A Unified Iteration Space Transformation Framework for Sparse and Dense Tensor Algebra

by

Ryan Michael Senanayake

S.B., Massachusetts Institute of Technology (2019)

Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2020

Author . . . .
Department of Electrical Engineering and Computer Science
January 27, 2020

Certified by . . . .
Saman Amarasinghe
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

A Unified Iteration Space Transformation Framework for Sparse and Dense Tensor Algebra

by

Ryan Michael Senanayake

Submitted to the Department of Electrical Engineering and Computer Science
on January 27, 2020, in Partial Fulfillment of the
Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This work addresses the problem of optimizing mixed sparse and dense tensor algebra in a compiler. I show that standard loop transformations, such as strip-mining, tiling, collapsing, parallelization and vectorization, can be applied to irregular loops over sparse iteration spaces. I also show how these transformations can be applied to the contiguous value arrays of sparse tensor data structures, which I call their position spaces, to unlock load-balanced tiling and parallelism.

These concepts have been prototyped in the open-source TACO system, where they are exposed as a scheduling API similar to the Halide domain-specific language for dense computations. Using this scheduling API, I show how to optimize mixed sparse/dense tensor algebra expressions, how to generate load-balanced code by scheduling sparse tensor algebra in position space, and how to generate sparse tensor algebra GPU code. As shown in the evaluation, these transformations allow us to generate code that is competitive with many hand-optimized implementations from the literature.

Thesis Supervisor: Saman Amarasinghe
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank Saman Amarasinghe, my research advisor, for all of his support and insight provided throughout the past two years. I would also like to thank Fredrik Kjolstad, whom I had the opportunity to work closely with and learn an enormous amount from. I consider myself extremely lucky to have been mentored by these two incredible professors. I also want to thank all of the other mentors, teachers, peers, and friends who have pushed me forward to this point and have supported and encouraged me every step of the way.

I was able to work closely with Changwan Hong and Shoaib Kamil who were coauthors on the paper that this thesis is based on. I have never worked with such brilliant collaborators. This research would also not have been possible without very helpful discussion, suggestions, and reviews from the following people: Stephen Chou, Michael Pellauer, Ziheng Wang, Ajay Brahmakshatriya, Albert Sidelnik, Michael Garland, Rawn Henry, Suzy Mueller, Peter Ahrens, and Yunming Zhang. Thank you so much for taking time out of your busy days to help!

Lastly, I want to thank my parents Michael and Rozanni, my brother Kevin, the cousins that I grew up with, Stefan and Hassan, and my partner Sammy. You have been the greatest of friends, teachers, and role-models. Out of all of the opportunities provided to me, I am most grateful for your presence in my life.

Contents

1 Introduction **13**

2 SpMV Example **19**

3 Coordinate & Position Iteration Spaces **23**

4 Intermediate Representation **31**

4.1 Iteration Graph Background 31

4.2 Iteration Graph Extensions 33

4.3 Provenance Graphs 34

5 Transformations **37**

5.1 Pos 39

5.2 Coord 40

5.3 Reorder 40

5.4 Fuse 41

5.5 Split 41

5.6 Divide 42

5.7 Parallelize 42

5.8 Bound 44

5.9 Unroll 44

6 Code Generation **45**

6.1 Merge Lattice Construction 47

6.2 Coordinate Recovery 48
6.3 Derived Bounds Propagation 52
6.4 Iteration Guards 54
6.5 Parallel and GPU Code Generation 55

7 Scheduling API 57

8 Evaluation 63

8.1 Methodology 63
8.2 Comparative Performance 65
8.2.1 SpMV CPU (Figure 8-1) 65
8.2.2 SpMV GPU (Figure 8-2) 65
8.2.3 SpMM CPU (Figure 8-3) 67
8.2.4 SpMM GPU (Figure 8-4) 68
8.2.5 MTTKRP CPU (Figure 8-5) 69
8.2.6 MTTKRP GPU (Figure 8-6) 70
8.3 Scheduling for GPUs 72
8.4 Scheduling for Load Balance 73
8.5 Scheduling for Maximal Parallelism 73
8.6 Scheduling for Locality 75

9 Related Works 77

10 Conclusion and Future Work 81

A Schedules 83

A.1 Scheduled TACO SPMV CPU (Figure 8-1) 83
A.2 Scheduled TACO SPMV GPU (Figure 8-2) 84
A.3 Scheduled TACO SPMM CPU (Figure 8-3) 85
A.4 Scheduled TACO SPMM GPU (Figure 8-4) 86
A.5 Scheduled TACO MTTKRP Order-3 CPU (Figure 8-5) 87
A.6 Scheduled TACO MTTKRP Order-4 CPU (Figure 8-5) 88

A.7 Scheduled TACO MTTKRP Order-5 CPU (Figure 8-5) 89
A.8 Scheduled TACO MTTKRP GPU (Figure 8-6) 90
A.9 SpMV Thread per Row on GPU (Section 8.3) 91
A.10 SpMV Warp per Row on GPU (Section 8.3) 91
A.11 SpMV on GPU with no Unrolling (Section 8.3) 92
A.12 SpMM on CPU Tiled (Section 8.6) 92
A.13 SpMM on CPU No Tiling Section 8.6) 93

List of Figures

1-1 Coordinate and position cuts of a matrix 14

1-2 Structure of the TACO-based prototype 15

2-1 Matrix-vector multiplication code examples 20

2-2 GPU Optimized SpMV 21

3-1 Dense matrix-vector multiplication coordinate and position spaces . . 24

3-2 Sparse matrix-vector multiplication coordinate and position spaces . . 24

3-3 Position space iteration of a 3-tensor 26

3-4 Tiling the coordinate space 28

3-5 Tiling the position space of the column coordinates 28

3-6 Tiling the position space of the row coordinates 28

4-1 Coordinate hierarchies and iteration spaces for intersection 32

4-2 Iteration graphs and iteration patterns for row-major and tiled iterations 34

4-3 Provenance Graph 35

5-1 Transformations applied to a row-major nested iteration 38

6-1 Color-coded generated code for parallel SpMV 46

6-2 Provenance graph depicting different recovery types 50

7-1 Sample schedule to parallelize SpMV 58

7-2 ParallelUnit and OutputRaceStrategy enum values 61

8-1 Comparative Performance: SpMV CPU 66

8-2 Comparative Performance: SpMV GPU 67
8-3 Comparative Performance: SpMM CPU 68
8-4 Comparative Performance: SpMM GPU 69
8-5 Comparative Performance: MTTKRP CPU 70
8-6 Comparative Performance: MTTKRP GPU 71
8-7 Evaluation: Load-balanced GPU strategy 74

Chapter 1

Introduction

Dense tensor algebra compilers, such as TCE, TVM and Tensor Comprehensions, build on decades of research on loop transformations for affine loop nests [48, 47, 26]. Sparse tensor algebra compilation was introduced by the TACO system, however it lacks an analogous sparse loop transformation framework [22].

Without a loop transformation framework, sparse tensor algebra compilers leave several optimization opportunities on the table. First, sparse tensor algebra expressions are often a mix of dense and sparse tensor algebra, with some operands stored in sparse data structures and some in dense arrays. Examples include (sparse matrix) $\times$ (dense vector) multiplication (SpMV) and (sparse matrix) $\times$ (dense matrix) multiplication (SpMM). Second, sparse tensors often have some dense dimensions, such as the blocked compressed sparse rows matrix format (BCSR) representing a sparse matrix containing dense blocks. Without a loop transformation framework, sparse tensor algebra compilers cannot optimize the dense loops in mixed sparse and dense expressions. Furthermore, current sparse tensor algebra compilers cannot apply loop transformations, such as strip-mining, loop reordering, and loop collapsing, to sparse loops. I will show that such transformations are possible and that they can be used to enable parallelization and make effective use of modern GPUs. Finally, I will show that by tiling the contiguous nonzero value arrays in sparse tensors, we can generate static load-balanced parallel code.

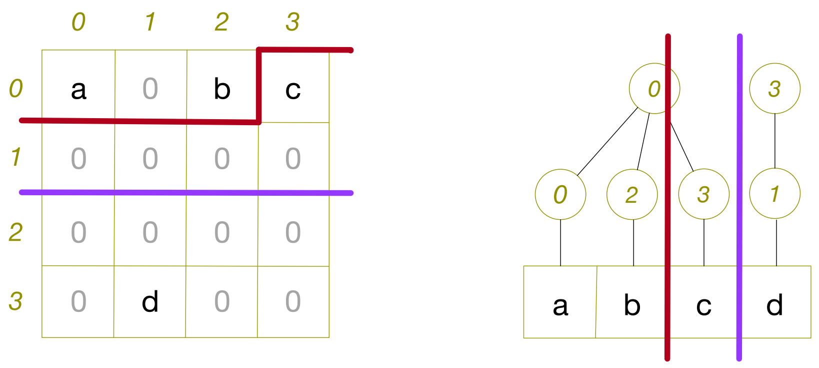


Figure 1-1: A coordinate cut (purple) and a position cut (red) in the coordinate iteration space of a matrix (left), and the same cuts in its compressed coordinate data structure. The coordinate cut cleanly divides the iteration space, while the position cut cleanly divides the values of the data structure.

Applying loop transformations to sparse loops is challenging as sparse loops coiterate data structures, sparse inputs may not support random access, and naively tiling the computation leads to load-imbalance. In contrast, dense loops iterate over a fixed range and index into data structures. As dense inputs can be randomly accessed, tiling the calculation of index values is sufficient to tile loads and stores. Sparse loops may coiterate any number of data structures and the resulting imperative code may contain while loops, conditionals and indirect memory references. Load-balancing a sparse computation to execute in parallel requires partitioning based on the sparsity pattern of data structures, which is not known until runtime. Efficient load-balancing is crucial to achieving high performance for GPUs, which makes generating sparse GPU code non-trivial.

In this thesis, I propose a unified loop transformation framework for loop nests with both dense and sparse loops that come from sparse tensor algebra. The loop transformations are applied to intermediate representations before generating imperative code. The transformations allow us to strip-mine, reorder, collapse, vectorize, and parallelize both dense and sparse loops subject to straightforward preconditions. Furthermore, the transformations enable strip-mining sparse loops both in the normal iteration space corresponding to dense loop nests and in the contiguous space given by the nonzeros of its compressed data structure (Figure 1-1).

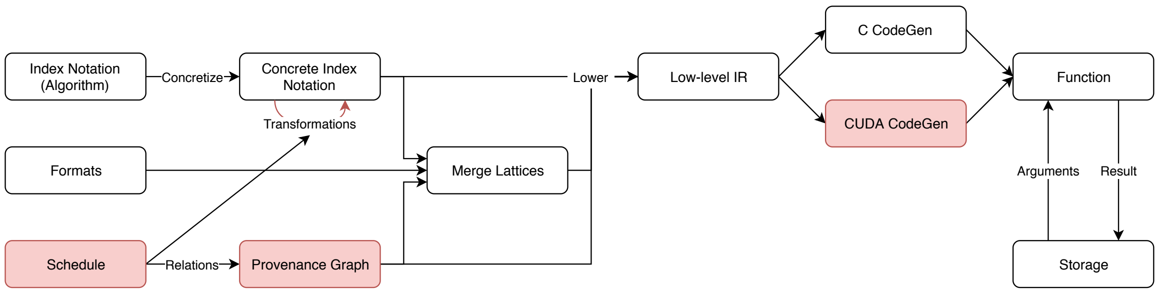


Figure 1-2: The structure of the prototype based on the TACO sparse tensor algebra compiler [22]. The user specifies the desired result using the Index Notation, Format, and Schedule languages. These are used in the construction of various intermediate representations until the final output Function is generated. The Function can be used to compute tensor algebra directly within TACO or can be used as a high-performance kernel in another application. Most components of TACO required modifications and the components highlighted red are new contributions of this thesis.

I have prototyped these ideas in the open-source TACO sparse tensor algebra compiler of Kjolstad et al. [22], which can generate code for any sparse tensor algebra expression. The structure of this prototype is shown in Figure 1-2. We expose the transformations as a loop scheduling API analogous to the Halide scheduling language [32]. Unlike Halide, however, the transformations can apply to both sparse and dense loops from mixed sparse and dense tensor algebra expressions. I leave automatic scheduling—developing an optimization system to automatically decide loop transformations—as future work. Experience from Halide demonstrates that an automatic scheduler is not necessary for a high-performance language to still be useful. Furthermore, recent work on creating ML-based automatic schedules for Halide can easily be targeted to my system. Such systems can be built on top of the transformations I propose here. With these transformations, for the first time, TACO is able to generate performance comparable code to the best hand-optimized libraries for GPUs.

My specific contributions are:

1. introduced position iteration spaces to complement the standard coordinate iteration spaces,
2. generalized standard loop transformations to sparse loops, and
3. constructed a CUDA GPU backend that allows generating high-performance CUDA kernels for all sparse tensor algebra expressions and formats supported by TACO.

The rest of this thesis is organized as follows:

Chapter 2 - SpMV Example describes the example problem of sparse matrix-vector multiplication (SpMV) and how this work can be applied to this problem to generate high-performance CPU and GPU kernels.

Chapter 3 - Coordinate & Position Iteration Spaces introduces position iteration spaces and describes how a computation has a coordinate space and multiple position spaces. A high-performance tiling strategy can require tiling multiple spaces as there are different benefits of tiling these two types of iteration spaces.

Chapter 4 - Intermediate Representation describes provenance graphs to track the derivation of transformed index variables and iteration graphs to describe nested and fused iterations.

Chapter 5 - Transformations presents an optimization framework that allows transforming the iteration space of a computation and mapping it to parallel hardware.

Chapter 6 - Code Generation describes extensions to the TACO code generator to support transformed sparse iteration spaces and CUDA GPU code generation.

Chapter 7 - Scheduling API details the transformation interface provided as a scheduling API in TACO.

Chapter 8 - Evaluation compares the performance of generated code against hand-tuned libraries. This chapter also highlights the need for automatic generation of these codes due to a large optimization space that is highly dependent on the sparsity pattern, size, hardware, and expression combination that is being optimized.

Chapter 9 - Related Works connects this work to prior work on optimizations and transformations for sparse matrix and tensor computations.

Chapter 10 - Conclusion and Future Work concludes with possible directions for future work.

Appendix A - Schedules provides the schedules used to generate all code presented in this thesis.

Chapter 2

SpMV Example

In this section, I demonstrate the capabilities of the loop transformation and code generation framework using a motivating example. The main target of this framework is all code expressing computations on mixed dense and sparse multidimensional tensor. For simplicity, though, I use the well-known example of matrix-vector multiplication: $a = Bc$, in which a vector c (a one-dimensional tensor) is multiplied by a matrix B (a two-dimensional tensor).

When all operands are dense, the code is straightforward. Figure 2-1a shows matrix-vector multiplication with dense operands (GEMV); the loops shown are easy to optimize, since they iterate over a regular data structure. However, when the matrix is sparse (SpMV), as shown in Figure 2-1b, iterating through the entries of the matrix becomes more complex. This is due to the fact that sparse matrices are most often stored in *compressed* structures[1], which avoid storing zero entries. More details about matrix storage are given in Section 3. The combination of complex iteration and indirect storage, common to most sparse matrix formats, makes subsequent loop transformations more difficult for sparse matrix-vector multiplication (SpMV).

In the simplest case, shown in Figure 2-1c, it is possible to tile SpMV by tiling the outer dense loop. This code is suitable for further optimizations such as parallelizing the loop over *i2*. Tiling by itself is also an important transformation to ensure that caches are used effectively. However, the resulting code of this tiling can be load-

[1]In this example, I use the compressed sparse row (CSR) storage.

```
for (int i = 0; i < M; i++) {
  for (int j = 0; j < N; j++) {
    int p = i * N + j;
    a[i] += B[p] * c[j];
  }
}
```

(a) Unscheduled GEMV

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

(b) Unscheduled SpMV

```
// IndexVar i1, i2;
// return stmt.split(i, i1, i2, 4);
for (int i1 = 0; i1 < CEIL(M,4); i1++) {
  for (int i2 = 0; i2 < 4; i2++) {
    int i = i1 * 4 + i2;
    if (i >= M) continue;

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

(c) Strip-mined SpMV

```
// IndexVar f, p;
// return stmt.fuse(i, j, f)
//            .pos(f, p, B(i, j));
int i = 0;
for (int p = 0; p < B_nnz; p++) {
  int j = B_crd[p];
  while (p == B_ptr[i+1]) i++;
  a[i] += B[p] * c[j];
}
```

(d) Position Iterating SpMV

Figure 2-1: Matrix-vector multiplication computed in several different ways, controlled by sparse iteration space transformations.

```
// IndexVar f, p, p1, p2, block, warp, thread, thread_nz, thread_nz_pre;
// TensorVar precomputed(Type(Float64, {Dimension(thread_nz)}), taco::dense);
// return stmt.fuse(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(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);
__global__ void gpuKernel(...) {
  ...
  int p_start = block * 8192 + warp * 512 + lane * 16;
  int i = binarySearch(B_ptr, block_row_starts[block], block_row_starts[block+1], p_start);
  double precomputed[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];
    precomputed[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_ptr[i+1]) i++;

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

void gpuKernel(...) {
  ...
  block_row_starts = search_block_starts<<<NUM_BLOCKS, 512>>>>(...);
  gpuKernel <<<B_nnz/4096, 512>>>(...);
  ...
}
```

Figure 2-2: GPU Optimized SpMV

imbalanced, because different rows may contain different numbers of entries. Logically, this approach tiles the i outer dimension of the matrix B while iterating through the i and j dimensions of the matrix.

The alternative approach is to instead directly iterate through the nonzeros of B rather than traverse the coordinate hierarchy with multiple loops. As sparse data structures are not defined for all coordinates, I find it useful to define the *positions* of a data structure as the dense index of the nonzero elements (ie the second position is the second coordinate & value pair that the data structure holds a nonzero value for). Therefore, I refer to iterating through the nonzeros of a data structure, as iterating through *position space*. Code for computing SpMV by iterating through position space is shown in Figure 2-1d. This style of iteration can be easily tiled while maintaining static load-balancing (segments of the loop of the same size contain the same amount of work) and can be used as the basis for high performance implementations on GPUs. Figure 2-2 shows an optimized GPU implementation of SpMV, which iterates through position space. This code requires a number of further transformations, including further tiling for GPU blocks, warps, and threads, as well as loop unrolling and precomputation. Transforming SpMV to iterate through position space, however, is the basis on which the rest of the transformations are applied.

While the code in Figure 2-1a-b can be generated by TACO, the extensions in this work enable the compiler to automatically generate the optimized implementations shown in the rest of the figure. In the rest of the thesis, I describe how I automate transforming code to iterate through position space for sparse tensors, and present a scheduling language users can use to explicitly apply this transformation along with others required to generate high performance sparse tensor code on CPUs and GPUs.

Chapter 3

Coordinate & Position Iteration Spaces

There are two types of iteration spaces: coordinate spaces and position spaces. A coordinate space is a multi-dimensional Cartesian combination of coordinates that encode each dimension of the iteration space. Transforming the coordinate space of a computation was first introduced by Leslie Lamport in 1974 [24]. Position spaces, on the other hand, are the positions along the space-filling curve created by imposing an ordering on these coordinates. There are many different orderings, but in this thesis I will only consider lexicographical orderings. Transforming the position space of a computation is newly introduced by this work.

Dense coordinate spaces, such as those that arise from dense linear and tensor algebra, can be visualized as a multi-dimensional lattice (i.e., a grid). Figure 3-1a shows a dense matrix-vector multiplication example $a = Bc$, and Figure 3-1b its two-dimensional $m \times n$ coordinate space, lexicographically ordered with the i coordinates before the j coordinates. The resulting loop nest, shown in Figure 2-1a, iterates over (i, j) coordinates and computes a position p to access the matrix using a strided formula $i * n + j$. Figure 3-1c shows the multiplication's one-dimensional position space, consisting of the positions along the lexicographical ordering of the coordinate space. The resulting loop nest would iterate over these positions p and compute the coordinates with the formulas $i = p/N$ and $j = p\%N$. As these are often expensive

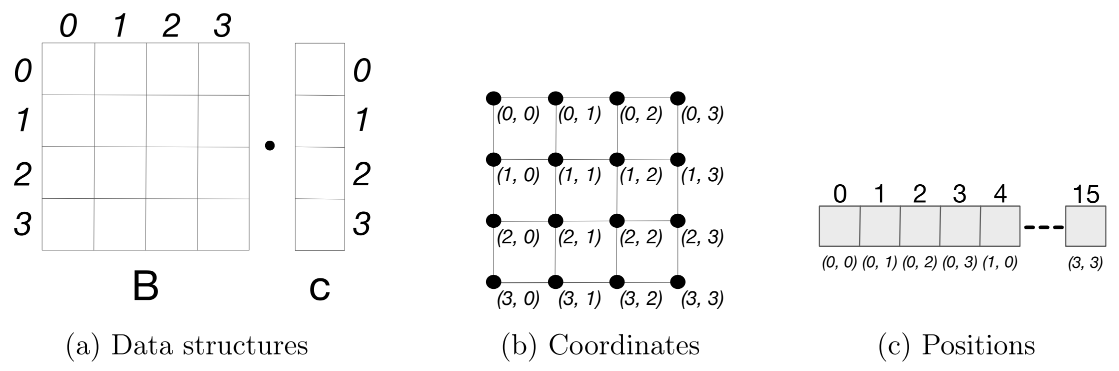


Figure 3-1: The dense matrix-vector multiplication ($a = Bc$) data structures, coordinate space and position space.

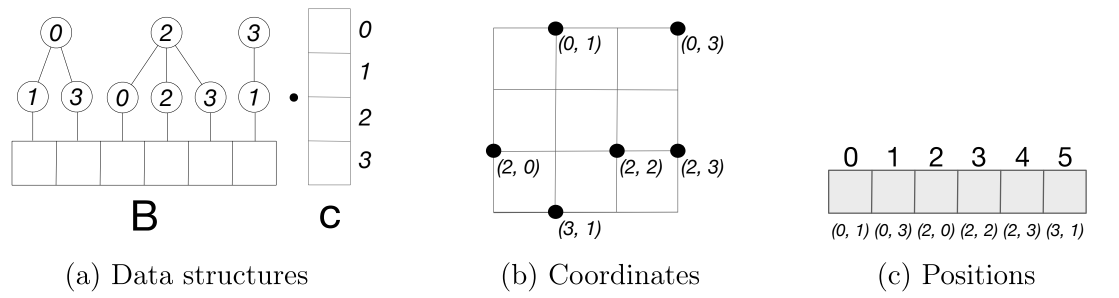


Figure 3-2: The sparse matrix-vector multiplication (SpMV) data structures, coordinate space and position space.

to compute, position iteration over a dense matrix primarily makes sense when the coordinates are not needed, such as when scaling a matrix.

Sparse iteration spaces appear when a loop iterates over one or more data structures that encode subsets of the coordinates in an iteration space dimension. Sparse tensors are stored in hierarchical compressed data structures. Figure 3-2a shows an example of a matrix B whose coordinates are stored in a compressed hierarchy that contains only those coordinates that lead to a nonzero matrix value. Note that, although we chose to show the coordinate hierarchy abstractly as a forest, in computer memory it would be stored as a concrete data structure, such as compressed sparse rows (CSR) or doubly-compressed sparse rows (DCSR). See Kjolstad et al. [22] or Chou et al. [10] for in-depth descriptions of the relationships between coordinate hierarchies and concrete matrix/tensor data structures.

Sparse coordinate spaces can be visualized as a multi-dimensional lattice with holes. The holes appear because the coordinate hierarchy data structures compress out coordinates corresponding to zero entries, and the sparse iteration space is described by the coordinates that are visited when loops iterate over these data structures. Figure 3-2b shows the sparse iteration space of the SpMV operation. The resulting loop nest that iterates over it was shown in Figure 2-1b. The outer loop iterates over the rows of B, stored as $(\text{coordinate}, \text{position})$ pairs. The positions are the locations of the coordinate at that level of the hierarchy, so the i coordinate 2 is stored at the second position after coordinate 0. The inner loop iterates over the nonempty (nonzero) components of the current row in the outer loop. The loops together iterate over the coordinate hierarchy of B and thus the sparse iteration space of Bc. (It is not necessary to iterate over dense c as the intersection resulting from the multiplication makes it sufficient to iterate over the smaller operand.)

Sparse position spaces are the sequence of nonzero coordinates in the order they are stored in a coordinate hierarchy, which may be thought of as a sparse space-filling curve through nonzero values. Their main advantage is that, although the coordinate space of a sparse tensor expression is sparse and irregular, its position space is a dense one-dimensional space that can be effectively tiled into equal-size blocks. This makes it possible to transform tensor expressions into statically load-balanced parallel code and to make effective use of vectorization and GPUs. Figure 3-2c shows the position space of the SpMV expression as a sequence of positions with coordinates attached.

Figure 2-1d shows the single loop that iterates through the sparse position space of the SpMV operation. Like the dense position space, the loop increments a position variable p. Based on the position it retrieves the coordinate j from the matrix B's coordinate array. In addition, it keeps track of the current coordinate i by incrementing it every time it reaches the end of a row. The increment of i is placed in a while loop to increment past empty rows. If there were more dimensions to the matrix B then the code would keep track of each coordinate above i using the same strategy.

Figure 3-3 shows how position space iteration generalizes to higher-order tensors. Instead of traversing the coordinate hierarchy in a typical top-down approach, the generated code instead traverses the positions of a specific level of the coordinate hierarchy (corresponding to a single dimension) directly. Once the end of a subtree is

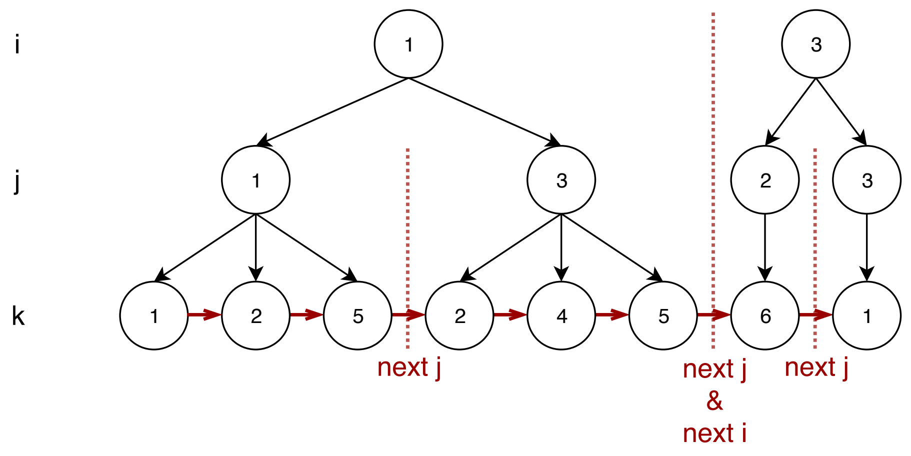


Figure 3-3: Position space iteration of a 3-tensor is shown by the red arrows running across the leaves of the coordinate tree. When the end of a subtree is reached, the i and j coordinate values are incremented accordingly.

reached, the appropriate upper-level coordinate values are incremented. Instead of requiring a conditional for every upper-level coordinate on every loop iteration, we can nest these conditionals, as a larger subtree can only end if the smaller right-most nested subtree also ends. This means that iterating over an n-order tensor requires at most $n-1$ conditionals, but most iterations will fail the first conditional and not have to evaluate the rest. If a coordinate level can have duplicate values (such as repeated row pointers for empty rows in CSR), then a while loop is necessary to skip past duplicate values; otherwise nested if statements are sufficient for unique values (such as in compressed sparse fiber, or CSF, which compresses out multiple levels of sparsity for higher-order tensors). The information needed to do this type of iteration is the same that is needed to perform a typical top-down iteration and therefore all sparse tensor formats that support top-down iteration can also be iterated in the position space.

We can tile the coordinate space or the position space of a sparse tensor algebra expression's loop nest. Tiling its coordinate space is conceptually straightforward and similar to how tiling works for dense loops. We simply split the dimensions of the sparse space into coordinate sets that are assigned to separate tiles. The challenge with tiling a sparse iteration space is that the tiles may not have the same number of nonzeros and will therefore have different sizes in memory. This means that a potentially expensive search is required to determine the starting location of each tile when iterating over them. Furthermore, varying sized blocks lead to different amounts of computation and thus load-imbalanced parallel execution. Figure 3-4 demonstrates the issue: although the cuts result in tiles with equal area, each tile contains a very different number of nonzero points in its iteration space. We see this clearly when projecting the cut to the coordinate hierarchy data structure of B, where the number of values in each tile substantially differs.

Tiling the position space of a sparse tensor algebra expression opens up new and exciting opportunities. The key property of position spaces is that they, in contrast to coordinate spaces, are always dense and contiguous. The tiles therefore have the same size and can easily be located. This lets us create statically load-balanced parallel code and makes it possible to generate code that is tuned for GPUs, which tend to benefit more from regular execution than CPUs. Figure 3-5 shows the effect of tiling in the position space on the sparse coordinate space of the SpMV example and on its data structures. The position tiles, in contrast to coordinate space tiles, are irregular in the coordinate space, but lead to equal-sized tiles in the position space. However, tiling in the position space introduces book-keeping code to keep track of coordinates above the cut (e.g., what row we are on in the SpMV example). As rows can also span multiple tiles, it is also necessary to deal with conflicting writes using synchronization instructions. Furthermore, we can only cut the position space with respect to the data structure of one of the operands of the expression. The resulting code, as we will see in Chapter 6, iterates over this operand, computes coordinates, and then finds the position of those coordinates in the other operands. Despite these drawbacks, I find that position tiles are often crucial, especially on inflexible compute platforms such as GPUs. It is a crucial strategy that I use to match the performance of well-studied high-performance GPU kernels (shown in Chapter 8), while generalizing to many tensor algebra expressions not previously studied.

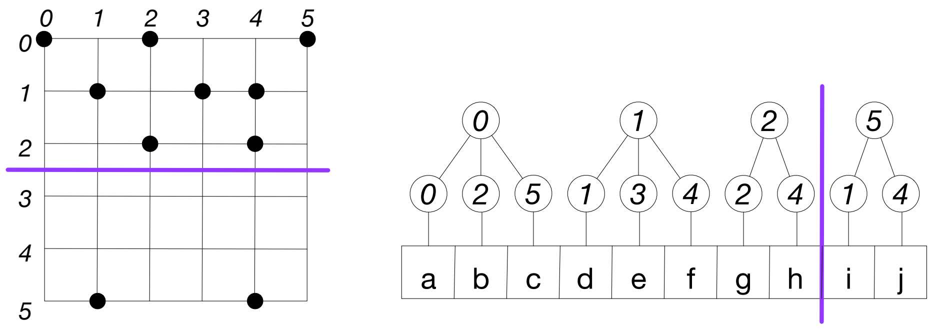


Figure 3-4: Tiling the coordinate space.

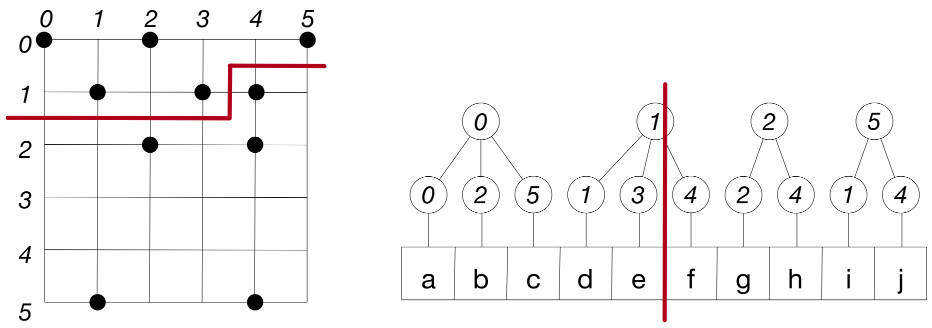


Figure 3-5: Tiling the position space of the column coordinates.

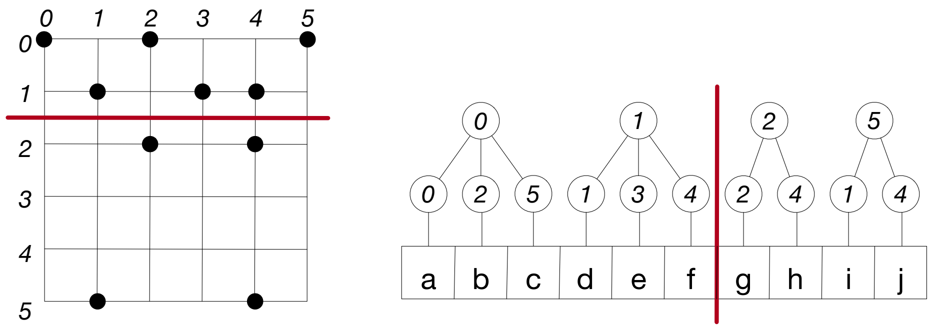


Figure 3-6: Tiling the position space of the row coordinates.

Finally, we can generalize position tiling to apply to any dimension of the iteration space, which corresponds to different levels of coordinate hierarchy in the data structure whose positions we are tiling. This lets us create tiles with a fixed number of coordinates in any dimension. For example, we can tile a SpMV operation in the position space of the rows of matrix B. This creates tiles that have the same number of nonempty rows; however, the rows themselves may have different numbers of nonzeros. Figure 3-6 shows a position cut in the row dimension of the iteration space, which corresponds to the first level of the coordinate hierarchy. The cut evenly divides the nonempty rows, whereas the position cut in the lowest level of the hierarchy evenly divides the nonzeros. The benefit of a cut in a higher level of the hierarchy is that the cut does not divide a row in two. This results in less book-keeping code and avoids the synchronization overhead to handle conflicting writes.

In this chapter, I introduced position spaces, which are dense and contiguous sections tied to a specific data structure. Coordinate and position spaces will be tiled by transformations described in Chapter 5. The Pos transformation will allow specifying a specific position space to schedule over and the $Coord$ transformation will allow further tiling a space that has already been tiled in position space. Chapter 6 will describe how code is generated to tile and iterate over these spaces using intermediate representations described in Chapter 4. Ultimately, these transformations and the concept of position spaces will be directly exposed to the user via a scheduling API described in Chapter 7. Tiling position space through this API will be essential to the high-performance GPU kernels evaluated in Chapter 8.

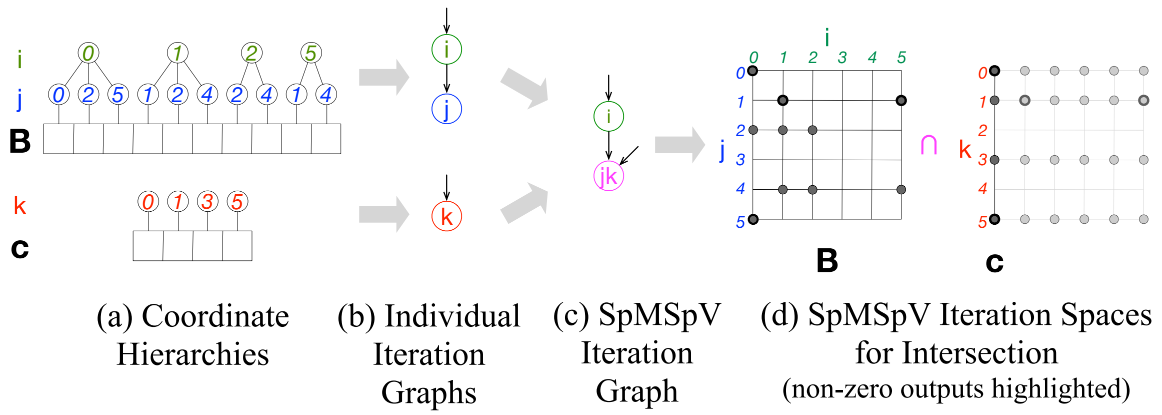
Chapter 4

Intermediate Representation

In order to carry out the iteration space transformations in this thesis, I extend the iteration graph intermediate representation of Kjolstad et al. [22] with the concepts of derived index variables, position index variables, and parallel index variables. An iteration graph describes the lexicographically ordered sparse iteration space that results from iterating over coordinate hierarchies, and transformations on the graph transform the corresponding iteration space. I also introduce the provenance graph intermediate representation, which formulates the relationships between derived index variables as a directed acyclic graph. This allows for specifying iteration graph code generation algorithms, described in Chapter 6, as directed traversals on a provenance graph. The code generation algorithms output loop nests that iterate over the space by coiterating over coordinate hierarchies. The transformations described in Chapter 5 directly transform iteration graphs and provide sufficient information in the SuchThat node to allow constructing a provenance graph.

4.1 Iteration Graph Background

We can symbolically describe an iteration space by a lexicographical ordering of index variables that represent dimensions, as in the polyhedral model. We extend this to a tree of such variables, where the concatenated ranges of the variables of each tree level together encode a dimension. This is analogous to an imperfect loop nest where an



i
j
B
k
c
i
j
k
i
jk
i
j
k
B
c
∩
(a) Coordinate Hierarchies
(b) Individual Iteration Graphs
(c) SpMSpV Iteration Graph
(d) SpMSpV Iteration Spaces for Intersection
(non-zero outputs highlighted)

Figure 4-1: Coordinate hierarchies and iteration spaces of a sparse matrix B and sparse tensor c are multiplied (SpMSpV).

outer loop contains two or more sequenced loops. These index variable trees describe the full, or dense, iteration space of a tensor algebra expression.

Sparse iteration graphs extend iteration variable trees with paths through index variable nodes. Each path represents a data structure's coordinate hierarchy. Different data structures contain different subsets of the iteration space. In tensor algebra these coordinate hierarchies, as we saw in the previous chapter, come from the data structures that encode the nonzero values of tensors. Figure 4-1a shows two abstract coordinate hierarchies for the matrix-vector multiplication example. The expression has two index variables that we choose to lexicographically order i before j. Figure 4-1b shows the two symbolic paths induced by these hierarchies. The matrix B enumerates both i and j coordinates and therefore its path go through both variables, where the vector c only enumerates j coordinates. Finally, Figure 4-1c shows the iteration graph that contains both paths, since the whole expression must coiterate over both coordinate hierarchies. The graph is annotated with an intersection operation between the two paths incoming on j. Thus, the iteration space of j is the intersection of each row of B and the vector c. It is an intersection because multiplying any value by zero yields a zero. It is therefore sufficient to iterate over those coordinates where both operands have a value in order to compute the output nonzeros. Conversely, a tensor addition would induce a union between incoming paths.

4.2 Iteration Graph Extensions

I extend iteration graphs with the concept of **derived index variables**. These new dimensions are added to an expression's iteration space by the *Split*, *Divide*, *Fuse*, *Pos*, *Coord*, and *Bound* iteration space transformations (described in Section 5) and can be in either coordinate space or position space. Figure 4-2 shows an iteration space before and after it has been tiled by splitting and reordering the index variables. The tiling increases the iteration space's dimensionality from two to four. All four of these index variables are derived as they do not appear in the original iteration space. Since we cannot cleanly visualize a four-dimensional space, however, we visualize the tiled iteration space in terms of its iteration order when projected onto the original iteration space. The nested iteration diagrams show the iteration of each index variable as differently colored arrows. For example, in the original iteration space, the iteration proceeds along the first blue arrow before it moves along the first red arrow to the next row. The unrolled iteration space shows the order of the overall iteration. Mapping between iteration spaces is important for code generation, since emitted loops iterate in the transformed space, while data structures must be accessed by coordinates in the original space. As well as allowing for derived index variables in the nodes of an iteration graph, I also allow tagging these nodes with additional information. I will leverage these tags to allow specifying hardware units to parallelize over and how to resolve data races.

I also add a new type of node to iteration graphs called a SuchThat node, which can optionally appear as the root of an iteration graph. This node provides context to the iteration graph that is needed to determine the iteration space represented by it. I leverage this new SuchThat node to store all transformations that yielded one or more new derived index variables. This allows for the code generator to determine how any given derived index variable is related to the original set of index variables in the untransformed iteration graph. These relationships are the edges of an index variable provenance graph, as described in the next section. This allows for trivially constructing the provenance graph of a given iteration graph as needed.

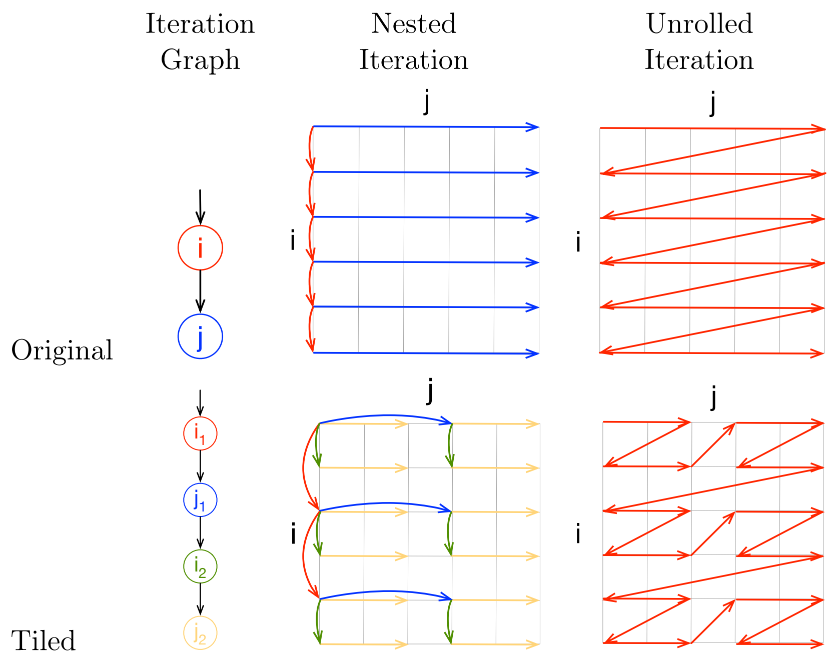


Figure 4-2: Iteration graphs, nested iteration orderings, and unrolled iteration orderings are shown for an untiled row-major iteration (top) and a tiled iteration (bottom).

4.3 Provenance Graphs

Index variable **provenance graphs** track the derivation of derived index variables from the original index variables in an index expression. Provenance graphs let us map between the transformed iteration space and the original space, so that the code generator can compute coordinates in the original space as needed to index into data structures. Figure 4-3 shows the provenance graph after creating a load-balanced SpMV kernel by tiling the expression in the position space. The derived index variables that represent the dimensions of the final iteration space, `p0` and `p1`, are tracked back to the index variables in the original space through the transformations they went through (transformations are described in the next chapter).

The transformations described in Chapter 5 create reversible relations between index variables. During code generation, we must build up expressions by following or reversing these relationships. An index variable provenance graph has a node for every index variable and for every relationship. Edges connect index variables to relationships. The edges are directed such that an index variable that is used in

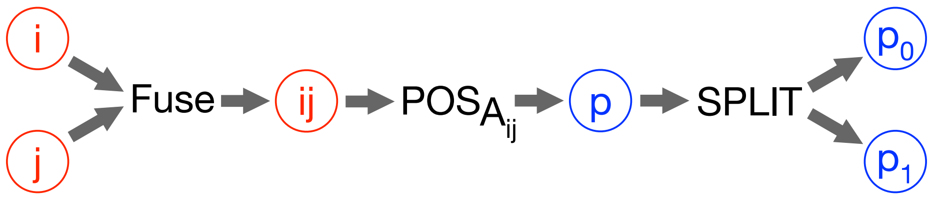


Figure 4-3: Index variable provenance graph after parallelizing SpMV in the CSR matrix's position space. The graph maps derived index variables back to the index variables they derived from. Blue variables are in position space.

a transformation points to the transformation and index variables created from a transformation are pointed to from the transformation. Variables are only created once and the set of transformations do not allow transforming variables that have already been replaced by a transformed variable. This means that all index variables in a provenance graph have at most one in-edge and one out-edge. Nodes that have no in-edges are dimensions in the original iteration space and nodes with no out-edges are dimensions in the transformed iteration space. I call these nodes respectively **underived** and **fully derived** and variables that are created from a transformation are **derived** from the variables used in the transformation. Untransformed index variables are considered both underived and fully derived.

By walking this graph, we are able to determine useful attributes about a given index variable. For example, an index variable is in position or coordinate space if there is a *Coord* or *Pos* transformation along its derivation path. Also, it can be determined if an index variable can be used to directly iterate over a position space by checking that it is never the outer variable in a *Split* or *Divide* transformation (which results in a strided iteration). As well as allowing us to determine properties about index variables, the code generation algorithm will use this graph to build up expressions needed to lower transformed iteration spaces.

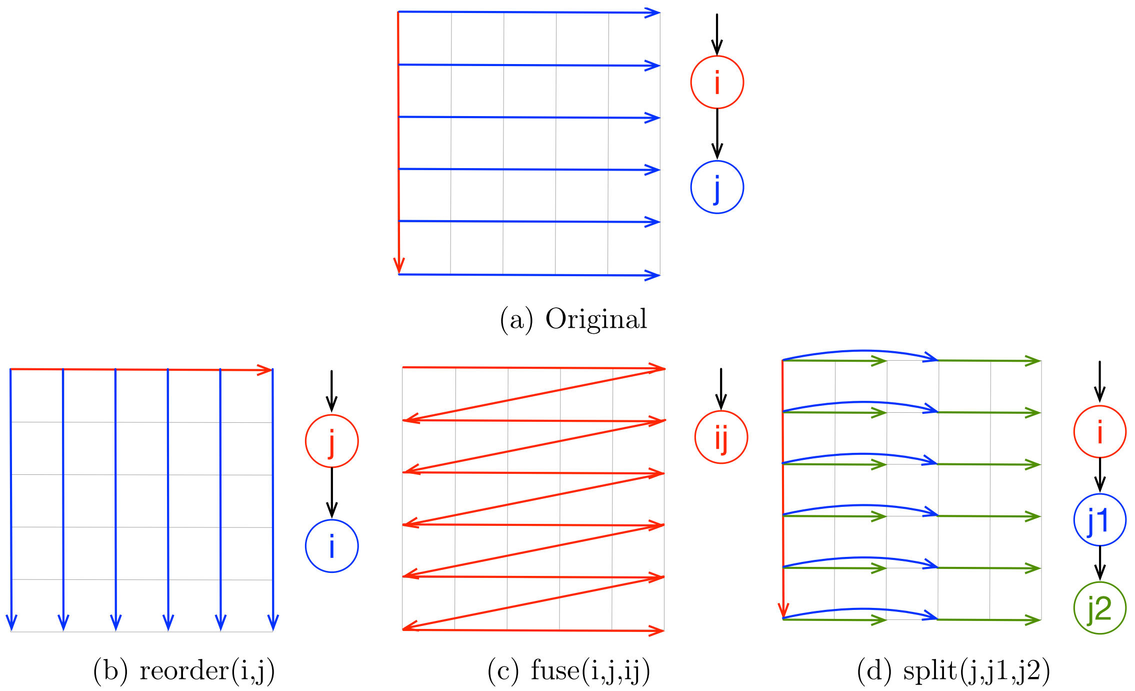
Chapter 5

Transformations

In this chapter, I will describe a transformation framework for sparse, dense, and mixed sparse/dense iteration spaces. The framework operates on the iteration graph intermediate representation discussed in the previous chapter. These transformations let us control the order of computation, so that we can optimize data access locality and parallelism.

The transformations in this thesis—*Coord*, *Pos*, *Reorder*, *Fuse*, *Split*, *Divide*, *Parallelize*, *Bound*, and *Unroll*—provide a comprehensive framework for controlling iteration order through sparse iteration spaces. Figure 5-1 shows the effect of the *Reorder*, *Fuse* and *Split* transformations on an i, j 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. All transformations apply to index variables in both the coordinate space and the position space, and the *Coord* and *Pos* transformations transition index variables between these spaces.

Key to my approach is that the transformations operate on the iteration graph intermediate representation before sparse code is generated. Similar to unimodular transformation space for dense loops [46], this 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,



(a) Original

(b) reorder(i,j) (c) fuse(i,j,ij) (d) split(j,j1,j2)

Figure 5-1: 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.

conditionals, and indirect accesses. The sparse code is then introduced when iteration graphs are lowered to code, as described in the next chapter.

Transformations need to be general enough to allow expressing high-performance techniques, but they also must be constrained enough to prevent producing incorrect code. If a user or autoscheduler cannot rely on the correctness of transformations, they are forced to test generated code against different sized inputs and concurrent interleavings. Not only does this make the code generator significantly less usable, it also can make autoscheduling searches intractable. If the following preconditions are checked, I believe that no transformation will ever lead to generated code that produces a different result. Formally proving this is left to future work.

Preconditions are checked during each transformation and the iteration graph to be transformed is assumed correct. If the supplied iteration graph is not valid then no guarantees can be made about the transformed iteration graph. In addition, all transformations have a precondition that the index variables to be transformed are in the provided iteration graph.

In Chapter 6, I specify per-transformation routines that will be used to build up generated code for transformed spaces. In Chapter 7, I explain how these transformation primitives are exposed to the user as a scheduling API. This scheduling API also enforces the preconditions specified in this chapter to ensure the correctness of the generated code.

5.1 Pos

Description The *Pos* transformation takes an index variable in the coordinate space and replaces it with a new derived index variable that operates over the same iteration range, but with respect to one input's position space. This has the effect of mode switching other commands such as *Split* and *Divide* to transform the position space rather than their default mode of operating on the coordinate space. If further commands are not used after this mode switch, this transformation has no effect. Even though the iteration graph now has a position variable as a node, it is important to realize that this does not signify that this forall node will iterate through all positions of the input. If the generated code's iteration pattern was changed by this transformation, it would be incorrect in the cases that iterating through the positions of one input does not iterate over all points where a computation is necessary. Instead the iteration graph produces the same generated code before and after the transformation and the mode switch only applies for further transformations.

Preconditions The index variable supplied to the *Pos* transformation must be in coordinate space. The *Pos* transformation also takes an input to indicate which position space to use. This input must appear in the computation expression and also be indexed by this index variable. If the index variable is derived from multiple index variables, these variables must appear directly nested in the mode ordering of this data structure. This allows for transforming multi-dimensional position spaces.

5.2 Coord

Description The *Coord* transformation takes an index variable in position space and replaces it with a new derived dimensional iterator that iterates over the corresponding iteration range in the coordinate iteration space. Similar to the *Pos* transformation, this causes for a mode switch for other transformations, but does not change the generated code by itself.

Preconditions The index variable supplied to the *Coord* transformation must be in position space.

5.3 Reorder

Description The *Reorder* transformation swaps two directly nested index variables in an iteration graph. This changes the order of iteration through the space and the order of tensor accesses. Figure 5-1b shows the effect of the *Reorder* transformation on a two-dimensional iteration space, in terms of iteration order on the original space. Whereas the original space was iterated through in row-major order, the reordering creates a new space that is equivalent to iterating through the original space in column-major order.

Preconditions The precondition of a *Reorder* transformation is that it must not hoist a tensor operation outside a reduction that it does not distribute over. Otherwise, this will alter the contents of a reduction and change the value of the result. 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.

5.4 Fuse

Description The *Fuse* transformation collapses two directly nested index variables. It results in a new fused index variable that iterates over the product of the coordinates of the fused index variables. This transformation by itself does not change iteration order, but it does change how the space is iterated. This transformation is useful, for example, to expose additional parallelism as one loop now iterates over all points that multiple loops would iterate over. Figure 5-1c shows the iteration order of the fused space in terms of the original space. The fused space is a one-dimensional space that is equivalent to iterating over the original space in a linearized order. The linearized order is determined by the nesting order. If the rows index variable is nested below the columns index variable, then the linearized order will be column-major (and the opposite nesting will be row-major).

Preconditions The *Fuse* transformation takes in two index variables. The second index variable must be directly nested under the first index variable in the iteration graph. In addition, the first index variable must be in coordinate space. To work with a multi-dimensional position space, it is instead necessary to fuse the coordinate dimensions and then use the *Pos* transformation. This allows us to isolate the necessary preconditions to the *Pos* transformation. The *Fuse* 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.

5.5 Split

Description The *Split* transformation splits (strip-mines) an index variable into two nested index variables, where the size of the inner index variable is constant. The size of the outer index variable is the size of the original index variable divided by the size of the inner index variable, and the product of the new index variables sizes therefore equals the size of the original index variable (or is slightly larger if the original index variable is not evenly divisible by the size of the inner index variable).

Note that in the generated code, when the size of the inner 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. Figure 5-1d shows the effect of the split transformation as an iteration order over the original space. The split replaces the j node with two nodes that together iterate over the same space. This transformation is useful for tiling to stream a workload over a fixed size resource such as a cache, CPU vector unit, or GPU thread-block.

Preconditions The split factor must be a positive non-zero integer.

5.6 Divide

Description The *Divide* transformation splits one index variable into two nested index variables, where the size of the *outer* index variable is constant. The size of the inner index variable is thus the size of the original index variable divided by the size of the outer index variable. The *Divide* transformation is important in sparse codes because locating the starting point of a tile can require an $O(n)$ or $O(\log(n))$ search. Therefore, if we want to parallelize a blocked loop, then we want a fixed number of blocks and not a number proportional to the tensor size. A common misconception is that *Divide* is simply the result of *Split* followed by *Reorder*. However, *Split-Reorder* results in a strided iteration, where consecutive iterations are split factor apart. *Divide*, like *Split*, keeps the same iteration order, but simply divides it into a constant number of chunks. This transformation is useful for statically load-balancing a fixed number of partitions of a workload such as parallelizing over all CPU cores.

Preconditions The divide factor must be a positive non-zero integer.

5.7 Parallelize

Description The *Parallelize* transformation tags an index variable for parallel execution. The transformation takes as an argument the type of parallel hardware to execute on. The set of parallel hardware is extensible and our current 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 Once a *Parallelize* transformation is used, no other transformations may be applied on the iteration graph as the preconditions for other transformations assume serial code. In addition there are sometimes hardware-specific rules to how things can be parallelized, such as that a CUDA warp has a fixed size of 32 threads or that to parallelize over CUDA threads one must also parallelize over CUDA thread-blocks. These hardware-specific rules are checked in the hardware-specific backend rather than before the transformation.

In addition to hardware-specific preconditions, there are preconditions related to coiteration that apply for all hardware. An index variable that indexes into multiple sparse data structures cannot be parallelized as it will become a while loop. Instead this loop can be parallelized by first strip-mining it with the *Split* or *Divide* transformation to create a parallel for loop with a serial nested while loop. Expressions that have an output in a format that does not support random insert can also not be parallelized. Parallelizing these expressions would require creating multiple copies of a data structure and then merging them, which is left to future work. Note that there is a special case where the output's sparsity pattern is the same as one of the inputs. For example, this is true of sampled dense-dense matrix multiply (SDDMM), tensor times vector (TTV), and tensor times matrix (TTM) kernels when choosing certain sparse data structures. This does not require creating multiple copies, but the implementation does not yet handle this special case.

Finally, there are preconditions related to data races during reductions. The *Parallelize* transformation allows for supplying a strategy 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. Other output race strategies will be described in Chapter 7.

5.8 Bound

Description The *Bound* transformation fixes the range of an index variable, which lets the code generator avoid inserting unnecessary conditionals and enables other transformations that require fixed size loops, such as vectorization. The *Bound* transformation allows specifying equality and inequality constraints. Additionally, it allows specifying if the range of an index variable is divisible by a constant.

Preconditions All provided constraints must hold for inputs of the generated code.

5.9 Unroll

Description The *Unroll* transformation tags an index variable to result in an unrolled loop with a given unroll factor. This reduces the amount of control flow logic at the cost of increased code size. Any early exit conditions for the loop will be hoisted outside of the loop before unrolling. This results in either a fully unrolled loop being executed or a normal loop with the early exit condition that is not unrolled.

Preconditions The unroll factor must be a positive non-zero integer.

Chapter 6

Code Generation

I extend the sparse tensor algebra code generator described by Kjolstad et al. [22] to support iteration graphs with derived index variables and to generate optimized CPU and CUDA GPU code. The TACO code generator generates an intermediate AST that is then targeted to a specific language and hardware by the backend. Previously, only untiled C code for CPUs could be generated. This C code generator was extended to allow scheduling over CPU vector units and threads. I also constructed a new CUDA backend and modified the code generation algorithm to generate an intermediate AST that contains loops that represent GPU blocks, warps, and threads. Lowering this AST to CUDA code follows a straightforward process as all high-level transformations have already been carried out. This provides the opportunity to more easily add additional backends to target new languages and hardware.

The existing code generator operates on iteration graphs and generates nested loops to iterate over each index variable. For each index variable, one or more loops are generated 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.

To generate loops that iterate, or coiterate, over derived index variables, the code generator must first compute their iteration domains. Figure 6-1 provides an

```
#pragma omp parallel for schedule(runtime, 1)
for (int p0 = 0; p0 < CEIL(B2_pos[B1_dim],16); p0++) {
  int i = 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;
}
```

Figure 6-1: Generated code for parallel sparse matrix-vector multiply. Red code recovers index variables, green code shows iteration bounds of derived index variables, and blue code depicts iteration guards.

example of a generated SpMV implementation that highlights derived loop bounds (green), iteration guards (blue), and index variable recovery (red). The following sections describe each of these extensions. These domains are computed by an iteration domain propagation algorithm, and they affect the bounds of generated loops (green in Figure 6-1). In cases where a fixed range index variable was split off from another index variable and its size does not evenly divide the original variable, the code generation algorithm also generates an iteration guard tail strategy (blue in Figure 6-1).[1]

[1]Note that I show the iteration guards inside the loop for readability, but in the implementation, the loop is cloned and an iteration guard determines which loop to enter.

To generate code for transformed sparse iteration spaces, I extend the TACO code generator to:

1. coiterate tiled and fused iteration spaces with a modified merge lattice construction algorithm (Section 6.1),
2. generate code to recover coordinates in the original iteration space from coordinates in the transformed iteration space (Section 6.2),
3. determine derived index variable bounds by propagating the bounds placed on the underived index variables (Section 6.3),
4. generate iteration guards to prevent iteration over invalid values from causing incorrect behavior (Section 6.4), and
5. generate SIMD vectorized, OpenMP CPU, and CUDA GPU code with reductions (Section 6.5).

6.1 Merge Lattice Construction

TACO uses merge lattices to optimize the coiteration of sparse ranges [22]. I extend the construction and optimizations on these merge lattices to support tiled and fused iterations. Merge lattices already support reasoning about intersections and unions of sparse tensor ranges where some ranges may become exhausted of values before others. I reuse these same concepts to add the additional functionality and therefore avoid making changes to the underlying merge lattice representation. As during lowering, the merge lattice construction process queries information from a provenance graph.

The prior merge lattice construction algorithm assumed that every node in an iteration graph performed a full coiteration of all data structures defined on the given dimension. For a tiled iteration, there are iteration graph nodes that iterate on an integer range rather than directly on tensors and there are nodes that coiterate on a segment of a dimension rather than the entire dimension (see Figure 4-2). The new algorithm uses a provenance graph to determine if an iteration graph node is iterating on an integer range and in this case produces the degenerate merge lattice that results in a simple for loop over the given index variable. When constructing a merge lattice for the coiteration of a segment of a dimension, the modified algorithm adds an additional intersecting data structure during construction, which is defined on values within the segment and zero otherwise. As this data structure should not actually be coiterated, (it does not exist), its iterator is only used to bound the iteration. In merge lattice terminology, the iterator of this data structure is a *ranger*, but not a *merger*. I also modify the merge lattice optimizations to still apply correctly for these segment coiterations. This formulation of tiled coiteration allows the reuse of much of the original merge lattice construction machinery.

When coiterating over a fused dimension, the modified algorithm constructs new iterators for this fused dimension that are equivalent to the iterators for the deepest nested dimension except that they are not nested within the iterator of the outer dimension. These iterators iterate through an entire level of a tensor rather than one segment of the level. Iterators in TACO are chained together such that all iterators have a parent iterator that iterates on the previous mode of the same data structure. The fused iterators skip the outer dimension in this chain. The values for the skipped iterators are recovered using the position space iteration strategy shown in Figure 3-3. This construction allows for us to merge data structures on fused dimensions just as we would on a single dimension. This does have the side effect however of losing the ability to merge on the outer dimension separately. For example, when performing element-wise multiply on two DSCR matrices A & B, we normally would be able to skip a row in B if it is empty in A, however iterating in the fused space does not allow this. I believe that position space iteration can be extended to allow this type of merge, but I leave this to future work.

6.2 Coordinate Recovery

In a transformed iteration space, dimensions are tiled and represented by derived index variables. The generated loops iterate over the coordinate ranges defined by these derived index variables. But the coordinate hierarchy data structures of the tensors contain and are accessed by coordinates in the original iteration space. The code generator must therefore emit code to map between these iteration spaces. I call this 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, which are stored in the original coordinate space. Recovering derived coordinates, on the other hand, is required when a coordinate in the original coordinate 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, and vice versa. 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 6-2), and

recover_derived, which computes the coordinate of an index variable from the variable it derives from and its siblings (green arrows in Figure 6-2).

Coordinate recovery may require an expensive search, so I define an optimization that computes the next coordinate faster than computing an arbitrary coordinate. Such tracking code implements iteration through recovered coordinates. The coordinate tracking code 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 during a fused iteration pattern (red stippled arrow in Figure 6-2).

Figure 6-1 uses the tracking optimization to track the `i` coordinate. It starts by finding the first `i` coordinate using a binary search, but then simply advances to the next `i` coordinate when it finds the end of a row segment.

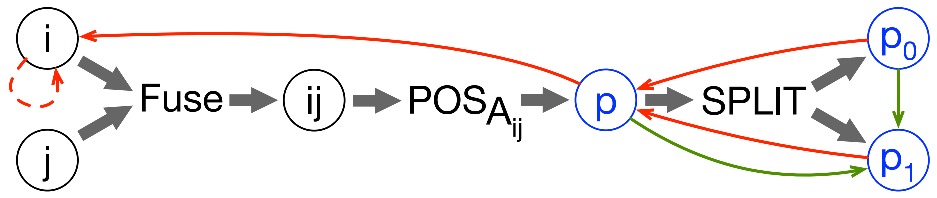


Figure 6-2: An index variable provenance graph annotated with arrows that depict different ways that an unknown index variable's coordinates can be recovered from known index variables. Red arrows depict original coordinate recovery and green arrows derived coordinate recovery.

Recovery functions are defined for each transformation that appears in the provenance graph:

Coordinate and Position Transformations I define two functions: `coord_from_pos` and `pos_from_coord`. The function `recover_original` for the position transformation and the function `recover_derived` for the coordinate transformation use the `pos_from_coord` function. The other recovery function for each of these transformations instead uses `coord_from_pos`. The function `pos_from_coord` searches the coordinate array of the underlying tensor to determine the position that indexes to this coordinate (or the coordinate directly preceding it) in the array. The function `coord_from_pos` indexes into the coordinate array at the given position index and retrieves the coordinate value. When iterating over a fused position space, the code generator is only able to retrieve the inner-most dimension's coordinate value and must search for upper-level coordinate values. This motivates the `recover_track` optimization.

```
// pos(i, B(i)) -> p
int i = B_crd[p];                              // recover_original
int p = search(B_crd, i);                      // recover_derived

// coord(p) -> p_coord
int p = search(B_crd, p_coord);                // recover_original
int p_coord = B_crd[p];                        // recover_derived
```

Fuse The function `recover_original` is defined separately for the outer and inner index variables. To recover the outer variable from the fused variable, the fused variable is divided by the size of the inner index variable's range. To recover the inner variable, division is replaced by modulo. The function `recover_derived` uses the simple equation fused = outer $*$ inner_size $+$ inner.

```
// fuse(i, j) -> f
int i = f / j_size;                               // recover_original i
int j = f % j_size;                               // recover_original j
int f = i * j_size + j;                           // recover_derived
```

Split *Split* defines the relationship original = outer $*$ split_factor $+$ inner. Solving for the various variables in this equation results in the definitions of `recover_original` and `recover_derived`.

```
// split(i, split_factor) -> i1, i2
int i  = i1 * split_factor + i2;                   // recover_original
int i1 = (i - i2) / split_factor;                  // recover_derived i1
int i2 = i - i1 * split_factor;                    // recover_derived i2
```

Divide *Divide* is similar to split except split_factor = dimension_size/divide_factor Once the size of the dimension is known and split_factor becomes a constant, the recovery functions are the same.

```
// divide(i, divide_factor) -> i1, i2
int i  = i1 * CEIL(i_size, divide_factor) + i2;   // recover_original
int i1 = (i - i2) / CEIL(i_size, divide_factor);  // recover_derived i1
int i2 = i - i1 * CEIL(i_size, divide_factor);    // recover_derived i2
```

Bound The *Bound* transformation simply passes through the value of the parent to the derived variable (and assumes that the value satisfies the *Bound*'s constraint). Therefore recovery is performed with the identity function.

```
// bound(i, ...) -> i_bound
int i = i_bound;                                   // recover_original
int i_bound = i;                                   // recover_derived
```

6.3 Derived Bounds Propagation

To determine the iteration domain of derived index variables, we propagate bounds through the index variable provenance graph (see Figure 4-3). I have defined propagation rules for each transformation 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.

The bound propagation functions for each transformation that appears in the provenance graph are defined below:

Coordinate and Position Transformations The bounds of a *Coord* transformation are found by indexing into the coordinate array with the current position bounds. The bounds of a *Pos* transformation are determined by searching for bounds of the original variable in the coordinate array and determining the position bounds that correspond to these coordinate bounds. We can avoid this search cost when the position transformation is applied to an index variable that spans the entire dimension (ie it has not been tiled). In this case, we can set the position bounds to run from 0 to the size of this position space.

```
// pos(i, B(i)) -> p
int p_min = search_ceil(B_crd, i_min);
int p_max = search(B_crd, i_max - 1) + 1;

// coord(p) -> p_coord
int p_coord_min = B_crd[p_min];
int p_coord_max = B_crd[p_max - 1] + 1;
```

Fuse Both the max and min bounds of the derived variable are equal to the bound of the outer variable multiplied by the size of the inner dimension and added to the bound of the inner variable. When both variables are bounded to fill the dimension, this results in the size of the derived dimension being a multiplication of the size of the two dimensions being fused.

```
// fuse(i, j) -> f
int f_min = i_min * (j_max - j_min) + j_min;
int f_max = i_max * (j_max - j_min) + j_max;
```

Split The iteration bounds for the inner variable always run from 0 to the split factor. The iteration bounds for the outer variable are determined by taking the original bounds and dividing them by split factor and then rounding the minimum bound down and the maximum bound up.

```
// split(i, split_factor) -> i1, i2
int i1_min = i_min / split_factor;
int i1_max = CEIL(i_max, split_factor);
int i2_min = 0;
int i2_max = split_factor;
```

Divide The same rules as *Split* are used except split factor is replaced by the size of the dimension divided by the the divide factor.

```
// divide(i, divide_factor) -> i1, i2
int i1_min = i_min / CEIL(i_max - i_min, divide_factor);
int i1_max = CEIL(i_max, CEIL(i_max - i_min, divide_factor));
int i2_min = 0;
int i2_max = CEIL(i_max - i_min, divide_factor);
```

Bound The *Bound* transformation replaces one or more of the bounds of the derived variable with a compile-time value.

```
// bound(i, value, MIN_EXACT) -> i_bound
int i_bound_min = value;
int i_bound_max = i_max;

// bound(i, value, MIN_CONSTRAINT) -> i_bound
int i_bound_min = (i_min < value) ? value : i_min;
int i_bound_max = i_max;
```

6.4 Iteration Guards

Unlike the coordinate tree iteration pattern that is generated for unscheduled computations, tiled iteration and position space iteration can not 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 at invalid values. This can lead to computing incorrect results or for the program to crash due to a segmentation fault. During position space iteration, it is necessary to determine when subtrees of the coordinate tree have been fully iterated (see Chapter 3) to prevent reducing values into the wrong location.

To prevent this incorrect behavior, iteration guards are generated. Tiled iteration guards cause an early exit of the loop and position space iteration guards determine when a subtree has ended and properly handles updating upper-level coordinate values, writing out results, and resetting temporaries. Position space iteration guards are generated depending on the format of the coordinate tree that is being iterated. The same process for generating loop bounds for a normal coordinate tree iteration is used to generate these guards. Similarly, the same process for writing results is used except the results are written only when the guard condition evaluates true instead of at the end of a nested loop. See Figure 6-1 for examples of both the position space iteration guards and tiled iteration guards.

Tiled iterations are guarded to ensure that no incorrect behavior occurs. Incorrect behavior occurs when an index variable is out-of-bounds of its own defined range and is used with the assumption that it is in-bounds. The original computation has dimensions that are defined by the sizes of input tensors. These dimensions can not be violated and a guard is placed if one of the underived index variables can possibly be out-of-bounds, as soon as the underived variable is recovered. This condition, however, is not sufficient to ensure correct execution of the code. Even if the original computation is never allowed to execute on an invalid point, generated code that indexes and searches over data structures to correctly coiterate could read invalid values or crash. It is therefore necessary to also generate a guard before an index variable that may be out-of-bounds is used to index or search over a data structure.

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 6.3. However, the parents of *Split* and *Divide* transformations 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* or *Divide* 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. In this way it can be determined if any given index variable is guaranteed to be in-bounds.

6.5 Parallel and GPU Code Generation

Parallelization and vectorization are applied to the high-level iteration graph IR and their safety can therefore be assured without heavy analysis. A parallelization strategy is tagged onto an index variable and the code generator generates parallel constructs from it, whether they are SIMD vector instructions, a parallel OpenMP loop, or a GPU thread array. The parallelization command can easily be extended with other parallelization strategies, and parallel code generators are easy to write as they only mechanically translate the common TACO lower-level intermediate representation, instead of having to also perform target-specific optimizations.

It is the responsibility of the parallel code generators to emit code that safely manages parallel reductions. I have implemented two strategies. The first strategy is to detect data races, by inspecting whether a reduction is dominated by an index variable that is summed over, and insert atomic instructions. A second strategy is to separate the loop into worker and reduction loops that communicate through a workspace [21] (e.g., an array). The threads in the parallel loop reduce into separate parts of the workspace. When they finish, the second loop reduces across the workspace, either sequentially or in parallel. I 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 each level of parallel hardware. For example, on a GPU the user can choose a loop separation strategy within a warp and atomics across warps.

Chapter 7

Scheduling API

I expose the sparse transformation primitives as a scheduling API in TACO, inspired by the Halide system for dense stencil computations [32]. The scheduling language is independent of both the algorithm language (used to specify computations) and the format language (used to specify tensor data structures). This allows users to schedule tensor computations independently of data structure choice while ensuring correctness for the overall algorithm. This separation further enables efficient execution on different hardware without changing the algorithm. I add the following member functions to the `IndexStmt` class, which implements the iteration graphs described in Section 4.1:

```
IndexStmt pos(IndexVar i, IndexVar p, Access a);
IndexStmt coord(IndexVar p, IndexVar p_coord);
IndexStmt reorder(IndexVar i, IndexVar j);
IndexStmt order(vector<IndexVar> reorderedVars);
IndexStmt fuse(IndexVar i, IndexVar j, IndexVar f);
IndexStmt split(IndexVar i, IndexVar i1, IndexVar i2, size_t size);
IndexStmt divide(IndexVar i, IndexVar i1, IndexVar i2, size_t size);
IndexStmt parallelize(IndexVar i, ParallelUnit pu, OutputRaceStrategy rs);
IndexStmt unroll(IndexVar i, size_t unrollFactor);
IndexStmt bound(IndexVar i, IndexVar i_bound, BoundType type, size_t val);
IndexStmt precompute(IndexExpr e, IndexVar i, IndexVar i_pre, Tensor w);
```

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

IndexVar i, j;
y(i) = A(i, j) * x(j);
IndexVar k, p, p0, p1;
IndexStmt stmt = y.getAssignment().concretize();
stmt = stmt.fuse(i, j, k)
           .pos(k, p, A(i,j))
           .split(p, p0, p1, CHUNK_SIZE)
           .parallelize(p0, ParallelUnit::CPUThread,
                            OutputRaceStrategy::Atomics);
y.compile(stmt);
y.assemble();
y.compute();
std::cout << y;
```

Figure 7-1: The schedule to parallelize SpMV over the nonzeros of the CSR matrix is shown. The transformations in this schedule correspond to the provenance graph in Figure 4-3 and it generates the code in Figure 6-1. Also depicts the implemented interface to use a schedule when computing a tensor result within the TACO library.

Iteration graphs are immutable data structures and so each of these member functions return a new transformed graph. This allows for chaining together scheduling API calls. An example schedule is shown in Figure 7-1. We first generate an iteration graph by using the `concretize` function and then apply successive transformations using scheduling language primitives.

These primitives directly correspond to transformations described in Section 5, but specifics of these API calls are described below:

`pos(IndexVar i, IndexVar p, Access a)` creates a new relationship, which specifies that the index variable `p` iterates the same range in position space as `i` iterates in coordinate space. This relationship is stored in the SuchThat node of the iteration graph. The access expression provided (ex. `B(i, j)`) is used to determine which position space that `p` should iterate over. The access expression provided must match an expression used in the original index notation that specifies the computation. The specific position space to use from this tensor is determined by comparing the derivation of `i` to the index variables used in the access expression. Using this transformation does not change the iteration pattern of the computation and is only used to mode-switch further transformations. More details and preconditions of this transformation are in Section 5.1.

`coord(IndexVar p, IndexVar p_coord)` creates a new relationship, which specifies that the index variable `p_coord` iterates the same range in coordinate space as `p` iterates in position space. This relationship is stored in the SuchThat node of the iteration graph. Using this transformation does not change the iteration pattern of the computation and is only used to mode-switch further transformations. More details and preconditions of this transformation are in Section 5.2.

`reorder(IndexVar i, IndexVar j)` returns a new iteration graph, which is identical except that the nesting positions of these two index variables have been swapped. More details and preconditions of this transformation are in Section 5.3.

`order(vector<IndexVar> reorderedVars)` is a higher-order command that takes a new ordering for a set of index variables that are directly nested in the iteration order. This command identifies the nested region of the iteration graph that contains all of the variables in `reorderedVars` out-of-order. Once it has identified this region and the current ordering, it uses the `reorder` command repeatedly to attempt to reach the desired order. If no set of reorderings exist that pass the preconditions of the `reorder` command, then this command will also fail.

`fuse(IndexVar i, IndexVar j, IndexVar f)` creates a new relationship, which specifies that the index variable `f` iterates the cartesian combination of the regions iterated by `i` and `j`. This relationship is stored in the SuchThat node of the iteration graph. Additionally, the corresponding iteration graph nodes to `i` and `j` are replaced by a new node that corresponds to `f`. More details and preconditions of this transformation are in Section 5.4.

`split(IndexVar i, IndexVar i1, IndexVar i2, size_t size)` creates a new relationship, which specifies that the index variables `i1` and `i2` together iterate the range iterated by `i` and that `i2` has a fixed range `[0, size)`. This relationship is stored in the SuchThat node of the iteration graph. Additionally, the iteration graph node corresponding to `i` is replaced by two nested nodes for `i1` and `i2`. More details and preconditions of this transformation are in Section 5.5.

`divide(IndexVar i, IndexVar i1, IndexVar i2, size_t size)` creates a new relationship, which specifies that the index variables `i1` and `i2` together iterate the range iterated by `i` and that `i1` iterates over a `size`-length range. This relationship is stored in the SuchThat node of the iteration graph. Additionally, the iteration graph node corresponding to `i` is replaced by two nested nodes for `i1` and `i2`. More details and preconditions of this transformation are in Section 5.6.

`parallelize(IndexVar i, ParallelUnit pu, OutputRaceStrategy rs)` tags the iteration graph node corresponding to `i` to indicate that each iteration of the node's range is to be executed in parallel. Included in the tag is the parallel hardware unit to execute on and the strategy for dealing with output races. The implemented hardware units and strategies are shown in Figure 7-2. Based on the strategy additional transformations on the iteration graph may occur such as performing `precompute` transformations or tagging racing store instructions as atomic. Additionally, if the hardware element size does not evenly divide the number of loop iterations (such as is often the case when using CPU vector units), the loop is cloned and leftover loop iterations are handled serially. Section 6.5 provides further information about parallel code generation. More details and preconditions of this transformation are in Section 5.7.

`unroll(IndexVar i, size_t unrollFactor)` tags the iteration graph node corresponding to `i` to indicate that generated code for this node should unroll all loops. The `unrollFactor` parameter is optional and the loop is fully unrolled if omitted. Otherwise, `unrollFactor` iterations are unrolled. To be able to unroll loops that

`ParallelUnit::CPUThread` generates a pragma to parallelize over CPU threads

`ParallelUnit::CPUVector` generates a pragma to utilize a CPU vector unit

`ParallelUnit::GPUBlock` must be used with GPUThread to create blocks of GPU threads

`ParallelUnit::GPUWarp` can be optionally used to allow for GPU warp-level primitives

`ParallelUnit::GPUThread` causes for every iteration to be executed on a separate GPU thread

`OutputRaceStrategy::NoRaces` raises a compile-time error if an output race exists

`OutputRaceStrategy::Atomics` replaces racing instructions with atomics

`OutputRaceStrategy::Temporary` uses a temporary array for outputs that is serially reduced

`OutputRaceStrategy::ParallelReduction` uses reduction operations across a warp/vector

`OutputRaceStrategy::IgnoreRaces` allows the user to specify that races can be safely ignored

Figure 7-2: Implemented enum values for ParallelUnit and OutputRaceStrategy, which are used in the `parallelize` scheduling API.

contain iteration guards, the lowering process clones the loop and unrolls one copy of the loop. The lowering process then generates guards to prevent the unrolled loop from being executed if the guard would have caused an early-exit. This process is described in Section 6.4. More details and preconditions of this transformation are in Section 5.9.

`bound(IndexVar i, IndexVar i_bound, BoundType type, size_t val)` creates a new relationship, which specifies that the index variable `i_bound` iterates the same dimension as `i`, but with one of the iteration bounds replaced with a compile-time constant. This relationship is stored in the SuchThat node of the iteration graph. More details and preconditions of this transformation are in Section 5.8.

`precompute(IndexExpr e, IndexVar i, IndexVar i_pre, Tensor w)` creates a new relationship, which specifies that the index variable `i_pre` iterates the same range as `i`. This transformation is described in prior work [21], but by adding this relationship to the SuchThat node, it composes with the set of transformations described in this thesis. This transformation precomputes the expression described by `e` into a new tensor workspace `w`. The producer loop is over `i_pre` and the consumer loop is over `i`. In this way, the producer and consumer can continue to be scheduled separately. Depending on the parallel index variables that the precomputed index variable is nested inside, the code generator can choose to specialize what memory the temporary is stored in. On GPUs, if the precomputed index variable appears outside of the `ParallelUnit::GPUBlock` index variable, the temporary is placed in global memory. If it appears between the `ParallelUnit::GPUBlock` and `ParallelUnit::GPUThread` index variables, the temporary is placed in shared memory. And finally if the temporary is nested inside a `ParallelUnit::GPUThread` index variable, the temporary is placed either in registers or in local memory.

Chapter 8

Evaluation

I carry out experiments to compare the performance of code generated by this technique with different schedules to state-of-the-art library implementations of three important expressions: SpMV, SpMM, and MTTKRP. I choose these expressions because they have been heavily studied in the performance engineering literature. I show that it is possible to express schedules that have competitive performance for these expressions. This work has the additional advantage that it can be applied to all sparse tensor algebra expressions and formats supported by TACO and also allows creating multiple different schedules that take advantage of the specific sparsity patterns of the inputs.

I carry out several studies to highlight situations where the best schedule differs depending on the situation. For example, the best CPU and GPU schedules differ for the same computations, and the best GPU SpMV schedule depends on whether the computation is load-balanced or not. I carry out these studies on the simplest kernels that are sufficient to make the point. For most studies this is the SpMV kernel, except for the locality study in Section 8.6 where I use the SpMM expression, since it has two dense loops to tile over.

8.1 Methodology

I implement the transformation framework as an extension to the TACO compiler, which is freely available under the MIT license. To evaluate it, I compare the performance of code that has been optimized using the introduced transformations on CPU to Intel MKL 2020 [17], Eigen 3.3.7 [13], SPLATT 1.1.1 [36], and the original TACO system (commit 331188). On GPUs, I compare to cuSPARSE v9.0 [29], the Merge-Based SpMV implementation of Merrill and Garland [27], and hand-optimized GPU kernels presented by Nisa et al. [28]. For the comparative studies, I use all real-valued matrices from the SuiteSparse sparse matrix repository [11] and tensors from the Formidable Repository of Open Sparse Tensors and Tools (FROSTT) [35]. I exclude matrices and tensors, which can not fit in memory of the machine used to run the experiment. For GPU, I generate one code for order-3 MTTKRP and use the order-3 tensors in FROSTT and the 1998DARPA tensor from the 1998 DARPA Intrusion Detection Evaluation Dataset [18]. For sparse matrices, I use the standard compressed sparse row (CSR) format and for sparse tensors, I use the standard compressed sparse fiber (CSF) format. I also carry out studies to evaluate the value of a scheduling language. The load-balance study uses synthetic matrices designed to show at what load imbalance it makes sense to move to a statically load-balanced kernel. I have made all schedules available in the appendix.

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, I compile code that our technique generates using Intel icpc 19.1.0.166 with `-O3`, `-DNDEBUG`, `-march=native`, `-mtune=native`,`-ffast-math`, and `-fopenmp`. I 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 DGX system with 8 V100 GPUs with 32 GB of global memory, 6 MB of L2 cache and 128 KB of L1 cache per SM (80 SMs), and a bandwidth of 897 GB/s. Only one GPU is used at a time. I compile the generated code with NVIDIA nvcc 9.0.176 with `-O3`, `-gencode arch=compute_70,code=sm_70`, and `-use_fast_math`. I run GPU experiments 25 times and report median execution times recorded with CUDA events. The generated code relies on CUDA unified memory, but I modify the kernels to use device pointers and explicitly copy the memory as the libraries I compared to did not use unified memory. I exclude the time of copying and allocating memory and freshly copy all data to the device for each trial. The L1 cache is automatically flushed between kernel invocations and the L2 cache is not flushed between trials.

8.2 Comparative Performance

The first experiments validate that the performance of the code generated using our transformation framework performs well compared to other systems. These are shown in Figures 8-1–8-6.

8.2.1 SpMV CPU (Figure 8-1)

The generated code parallelizes SpMV with chunks of size 32. This strategy (Appendix A.1) performs well on larger matrices and achieves a 6.8% geometric mean speedup over MKL for the entire dataset. The Original TACO code performs better for smaller matrices and this code can be generated within Scheduled TACO with a different schedule. As a simple example of the power of generating multiple schedules for the same expression, if I use the Original TACO code for matrices with fewer than 20k nonzeros and the Scheduled TACO code otherwise, I achieve a geometric mean speedup of 22.0% over MKL and 20.2% over Eigen.

8.2.2 SpMV GPU (Figure 8-2)

The generated code parallelizes the nonzeros of the sparse matrix over GPU threads. Each thread also precomputes its results into registers and unrolls this loop before writing out the results. The provides additional instruction-level parallelism, which as discussed in Section 8.3 is crucial for performance. This schedule (Appendix A.2) exhibits better load-balancing than the cuSPARSE default strategy, but with slightly more overhead. The overhead of load-balancing causes for performance of already load-balanced matrices to be slightly degraded compared to cuSPARSE, but this strategy

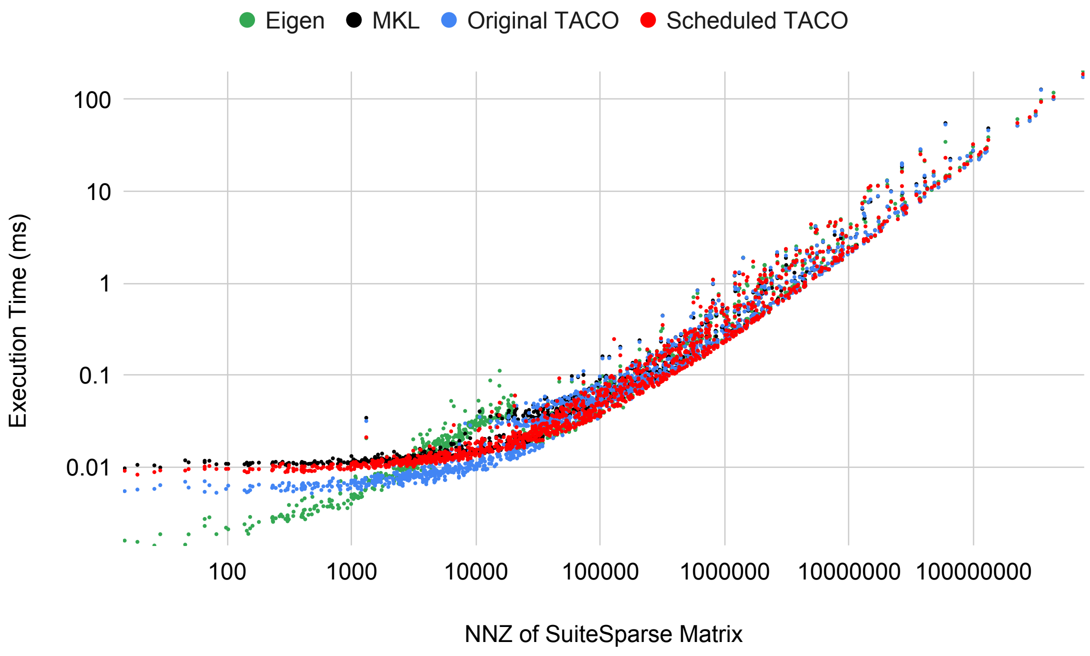


Figure 8-1: Generated SpMV CPU code (Scheduled TACO) compared to other high-performance implementations on all SuiteSparse matrices. [11] A log-scale is used to plot the execution time against the number of nonzeros in the matrix. The schedule used by Scheduled TACO is in Appendix A.1.

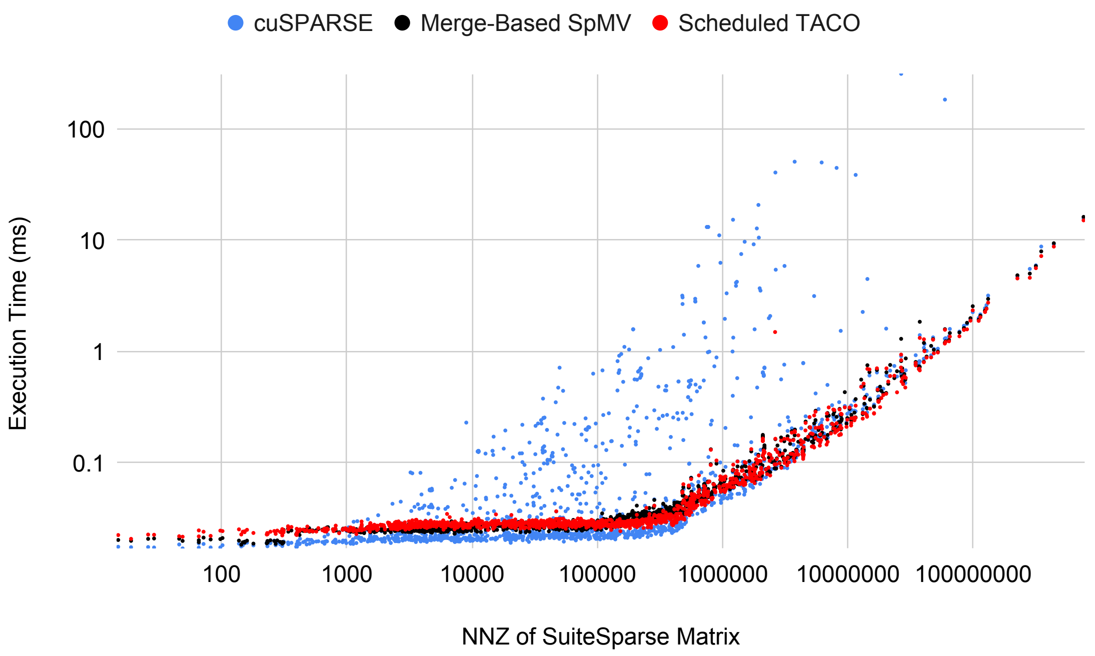


Figure 8-2: Generated SpMV GPU code (Scheduled TACO) compared to other high-performance implementations on all SuiteSparse matrices. [11] A log-scale is used to plot the execution time against the number of nonzeros in the matrix. The schedule used by Scheduled TACO is in Appendix A.2.

is far more resilient to the sparsity patterns of the matrices and is therefore more consistent. Scheduled TACO also requires additional kernel launches to perform its load-balancing scheme, which can dominate the performance for smaller matrices. The geometric mean speedup over cuSPARSE for this dataset is 39.0%. The Merge-Based SpMV strategy performs similarly to this strategy. They also load-balance the reads of empty row pointers in the CSR matrix, but this load-balancing incurs additional overhead. I achieve 99.0% of the geometric mean performance of Merge-Based SpMV while still retaining the benefits of a general optimization framework.

8.2.3 SpMM CPU (Figure 8-3)

The schedule (Appendix A.3) tiles the nonzero positions of each row of the sparse matrix to cache, vectorizes along the columns of the output, and parallelizes chunks of 8 rows. The additional overhead of this schedule causes for comparatively worse

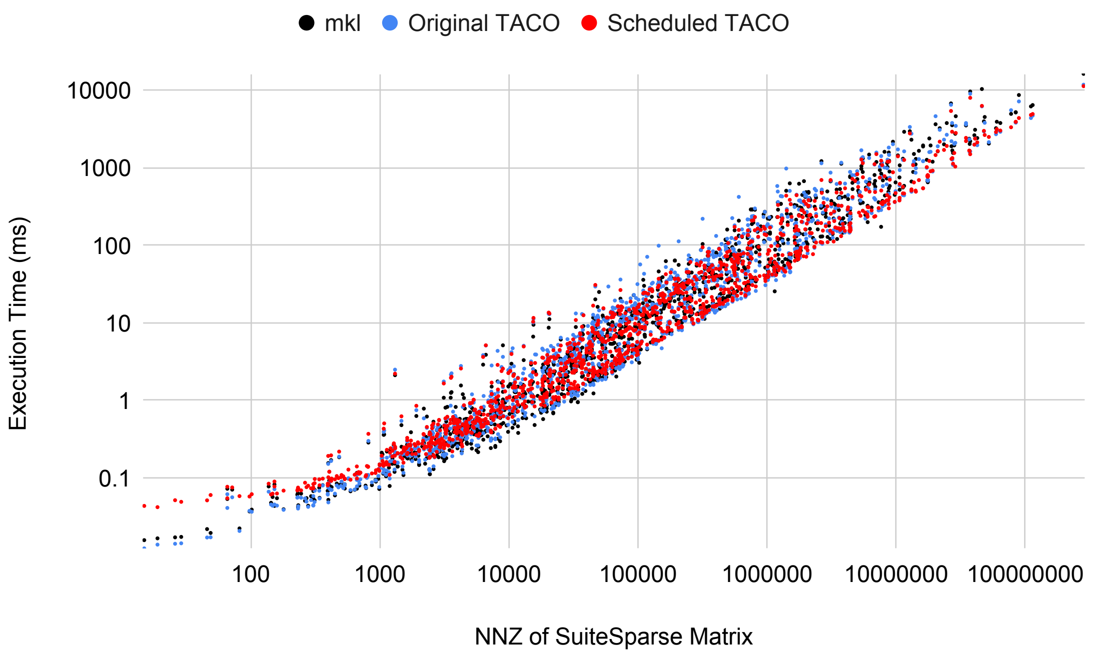


Figure 8-3: Generated SpMM CPU code (Scheduled TACO) compared to other high-performance implementations on all SuiteSparse matrices. [11] A log-scale is used to plot the execution time against the number of nonzeros in the matrix. The schedule used by Scheduled TACO is in Appendix A.3.

performance on small matrices with fewer than 15k nonzeros. The geometric mean speedup for the entire dataset over Original TACO is 7.3% and when the dataset is restricted to all matrices with greater than 15k nonzeros, this schedule achieves a 11.6% speedup. This schedule achieves 96.3% of MKL's performance. If I use the Original TACO schedule for matrices with fewer than 15k nonzeros and the Scheduled TACO schedule otherwise, I achieve a geometric mean 0.2% speedup over MKL for this dataset.

8.2.4 SpMM GPU (Figure 8-4)

For SpMM GPU (Appendix A.4), I tile 16 nonzero elements of the sparse matrix per GPU warp. I tile the dense matrix so that there are 4 tiles of 32 columns each. Each GPU warp loads a nonzero and each thread in the warp multiplies by a different column of the dense matrix. All threads reduce across its set of nonzeros and then

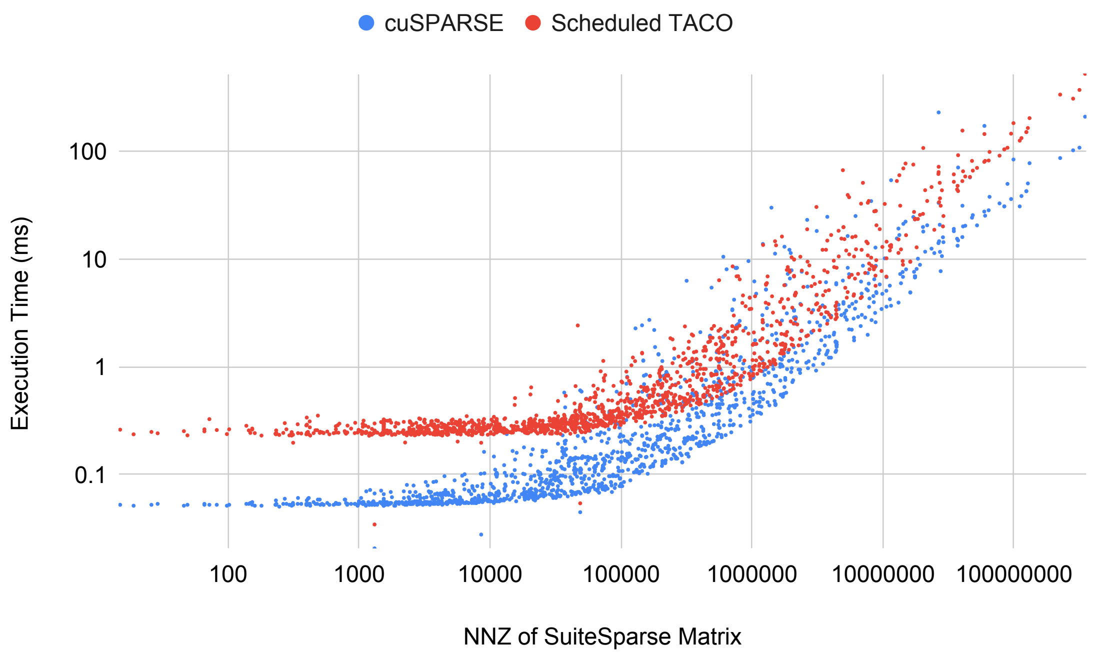


Figure 8-4: Generated SpMM GPU code (Scheduled TACO) compared to other high-performance implementations on all SuiteSparse matrices. [11] A log-scale is used to plot the execution time against the number of nonzeros in the matrix. The schedule used by Scheduled TACO is in Appendix A.4.

atomically add into the result matrix. I use a column-major output as cuSPARSE does not provide a kernel for a row-major output. This kernel imposes significant overhead for small matrices, but these matrices are often more efficiently computed on CPU anyway as they do not benefit from the parallelism offered by GPUs. For matrices with more than 1 million nonzeros in this dataset, I achieve 85.9% of the performance of cuSPARSE. cuSPARSE also modifies the GPU assembly generated by the nvcc compiler, which we are unable to do within this framework.

8.2.5 MTTKRP CPU (Figure 8-5)

I generate 3 different codes for order-3 (Appendix A.5), order-4 (Appendix A.6), and order-5 MTTKRP (Appendix A.7). In index notation, these kernels correspond to: $A(i,j) = B(i,k,l) * C(k,j) * D(l,j)$ (order-3), $A(i,j) = B(i,k,l,m) * C(k,j) * D(l,j) * E(m,j)$ (order-4), and $A(i,j) = B(i,k,l,m,n) * C(k,j) * D(l,j) * E(m,j) * F(n,j)$

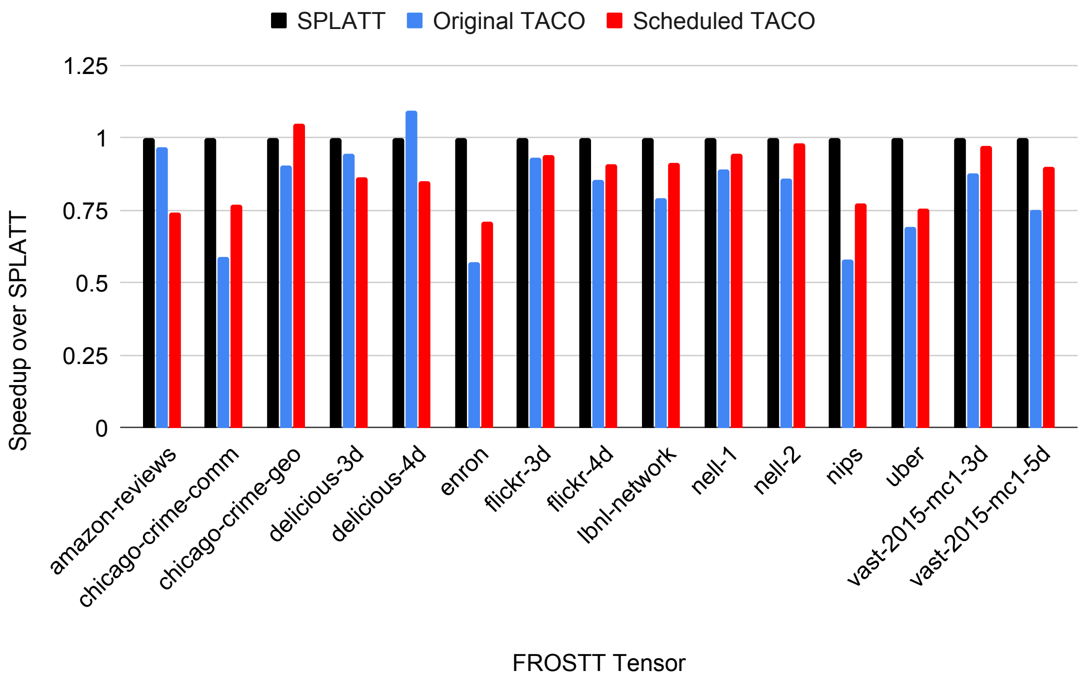


Figure 8-5: Generated MTTKRP CPU code (Scheduled TACO) compared to other high-performance implementations on all FROSTT tensors. [35] The speedup of implementations over the SPLATT baseline are plotted for all tensors. Different implementations are used depending on the order of the tensor. The schedules used by Scheduled TACO are in Appendices A.5, A.6, and A.7.

(order-5). All input tensors other than B have 32 columns and are stored densely. To match SPLATT, the first dimension of the B tensor is stored densely and all other levels are compressed. I use the `precompute` transformation to create 1, 2, or 3 j-sized workspaces for order-3, order-4, and order-5 MTTKRP respectively. This avoids redundant work, but imposes some additional overhead. I then parallelize the outer loop with chunks of 32. This provides a geometric mean speedup of 7.5% over Original TACO and achieves 86.7% of the performance of SPLATT.

8.2.6 MTTKRP GPU (Figure 8-6)

The schedule for MTTKRP GPU (Appendix A.8) parallelizes chunks of 4 nonzeros of the sparse tensor across GPU warps. Each thread in the warp multiplies the nonzero with values from the dense matrices corresponding to different j. Each thread reduces

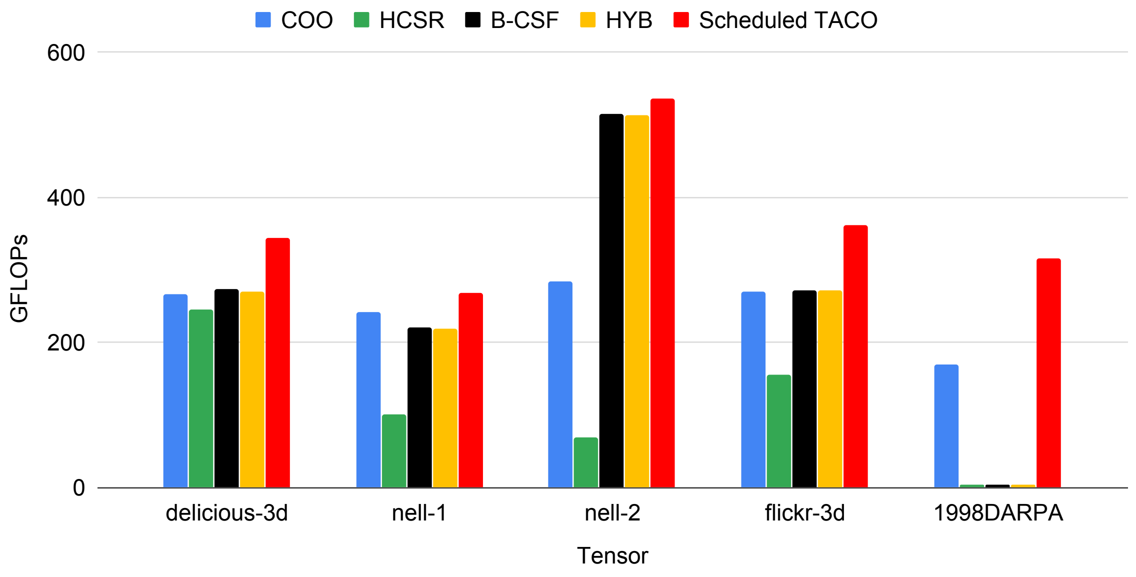


Figure 8-6: Generated MTTKRP GPU code (Scheduled TACO) compared to high-performance kernels written for different GPU-optimized tensor formats by Nisa et al. [28] Throughput in GFLOPs is reported for Order-3 FROSTT tensors and the 1998 DARPA Intrusion Detection Evaluation Dataset [35, 18] The schedule used by Scheduled TACO is in Appendix A.8.

across the nonzeros and then atomically adds into separate positions in the result tensor. This kernel is particularly difficult to write by hand given the multiple levels of searches required to tile the position space of a tensor that is not required for a matrix. I compare the performance of this code to four different kernels presented by Nisa et al. [28]. These kernels were targeted at a NVIDIA P100 (Pascal) device rather than the NVIDIA V100 (Volta) device, I used to run the experiments. The performance of GPU kernels tend to be tuned to a specific machine, which is an additional benefit of a generated kernel, which can be easily rescheduled, over a hand-tuned kernel. These kernels also take advantage of different formats that are built to load-balance well on GPU, while I use the standard CSF format and incur the runtime overhead of load-balancing. The Scheduled TACO kernel achieves a 29.0% geometric mean speedup over the best performance of all 4 kernels for each tensor. For the 1998DARPA tensor, the GFLOPs recorded for the HCSR, B-CSF, and HYB kernels are all less than 3.5. This is possibly due to the high skew in the number of nonzeros per outer-most dimension of the tensor.

8.3 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 the order of loads and to thread divergence, 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 6.9x worse than the warp-per-row GPU schedule on a matrix with four million randomly allocated nonzeros.

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 is a disaster on a GPU. Since threads in a warp execute separate rows, they cannot coalesce memory loads. This results in poor effective memory bandwidth and thus 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 thread divergence.

By contrast, more optimized GPU schedules are more carefully tiled. The warp-per-row schedule assigns an equal number of nonzero elements of the row to each thread and uses warp-level synchronization primitives to efficiently reduce these partial sums. The optimized SpMV schedule that I use in Figure 8-2 tiles the position space of the sparse matrix across threads. I 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.11.

8.4 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 8-2 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 8-7 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.

8.5 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 8.3 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, I execute the SpMV kernels generated from both schedules on a short and wide 100×100k 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 times faster on average across 10 runs.

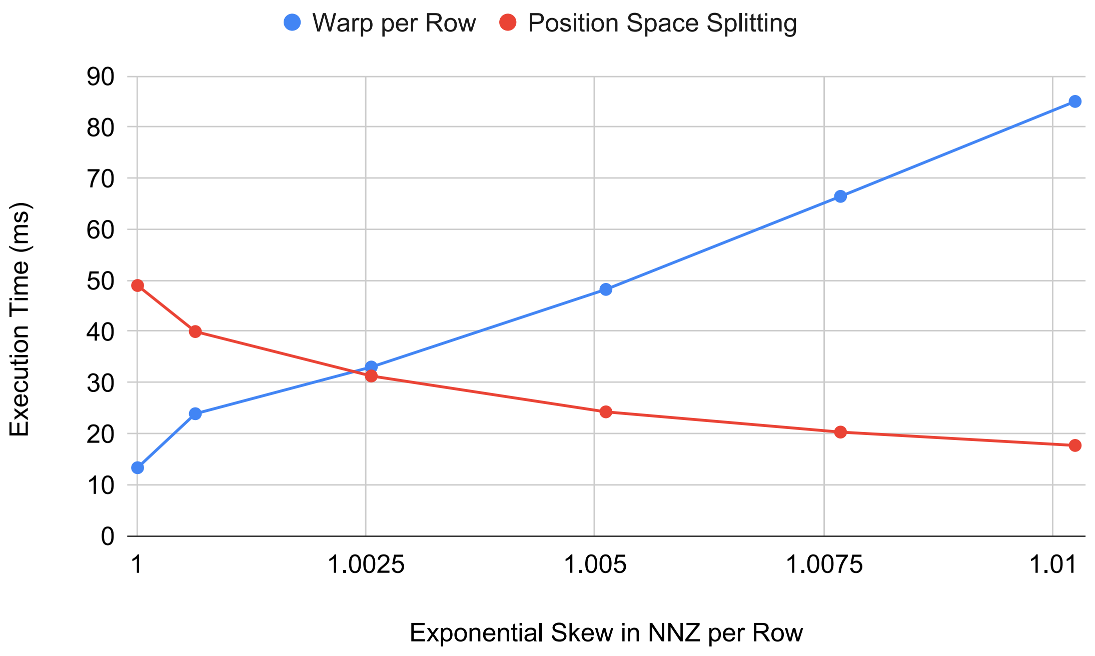


Figure 8-7: Performance comparison of warp-per-row and load-balanced SpMV on a matrix with a fixed number of nonzeros as the distribution per rows become more skewed. The number of nonzeros in row i is given by the formula $k * c^i$. c is the factor that appears on the x-axis of the graph (when $c = 1$, all rows have the same number of nonzeros). For every c, I choose k so that the number of nonzeros for the entire matrix is always 400 million. Nonzeros are generated uniformly random within a row and then all rows are shuffled. Schedules used to generate code in Appendices A.9 and A.10.

8.6 Scheduling for Locality

Sparse tensor algebra expressions can have dense loops that may be tiled for better temporal locality. I demonstrate this for the SpMM expression. Since a sparse matrix is multiplied by a dense matrix, the resulting kernel has two dense loops that can be tiled. I implement a tiled and an untiled schedule (Appendices A.12 and A.13) for this expression and run the resulting kernels on a 100k $\times$ 100k sparse matrix with an average of 1000 randomly distributed nonzeros per row multiplied by a 100k $\times$ 32 dense matrix. Not surprisingly, the tiled kernel performs 2.1x better.

Chapter 9

Related Works

There is a long history of optimizations and transformations for sparse matrix [45, 16, 11] and tensor computations. I divide prior work into the categories described in the following sections.

Sparse compilation The work by Bik and Wijshoff [8, 7] was an early attempt to apply compiler techniques to sparse matrix codes. They use a technique called guard encapsulation to turn dense loops over dense arrays into sparse loops over only nonzeros. The Bernouilli [23] system followed soon thereafter and lifts matrix codes to relational algebra, which is then optimized and emitted as sparse code. More recent works on compiling and transforming sparse loops have been done using the polyhedral model [40, 5, 41, 4]. These generally employ *inspector-executor* techniques, which combine run-time inspection of data with compile-time transformations. Venkat et al. [43] use the polyhedral model to turn dense loops with conditional guards into loops over a sparse matrix, enabling further optimizations including wavefront parallelism [44] and tiling of dense loops inside sparse codes [2]. Pugh et al. [31] designed SIPR, an intermediate representation for sparse compilation. Most recently, the TACO compiler generalizes sparse and dense tensor operations in a variety of formats, automatically generating code for any computation [22, 10, 21]. I build on TACO in this work, adding scheduling and GPU code generation.

Dense tensor compilation Recent works on dense tensor algebra have 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 [3] automatically optimizes dense tensor contractions and is developed primarily for chemistry applications. Libtensor [12] and CTF [39] cast tensor contractions as matrix multiplications by flattening and transposing tensors. In machine learning, TensorFlow [1] and other frameworks [30, 19] combine tensor operations to efficiently apply gradient descent for learning and are among the most popular packages used for deep learning. TVM [9] 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) [42] is another framework for defining new deep learning building blocks over tensors, utilizing the polyhedral model.

Scheduling Halide [33, 34] is a widely used domain specific language in industrial applications, partially due to its flexible scheduling language that lets users express how a high-level algorithm is efficiently executed. Many of our constructs are inspired by Halide, though we deal with sparse loops while Halide only considers dense loops. TVM uses a variant of Halide's scheduling language, modified for deep learning applications. Most recently, GraphIt [49] and Taichi [15] built scheduling languages for graph algorithms and sparse irregular spatial data structures, respectively. In the polyhedral framework, CHiLL [43] allows users to specify sequences of loop transformations, similar to a scheduling language.

Hand-optimized sparse tensor code Finally, researchers have studied how to manually optimize sparse linear and tensor algebra code for CPUs and GPUs. We will mention a few prominent examples. The Intel MKL library is a fast sparse linear algebra library for CPUs that employs some inspector-executor techniques to choose formats [17]. Bell and Garland describe a set of techniques for optimizing SpMV for several different data structures on GPUs, including the vectorized kernel we use in our evaluation [6]. Furthermore, Merrill and Garland showed how to develop a load-balanced SpMV implementation by generalizing a parallel merge algorithm [27]. The SPLATT library includes an efficient implementation of the MTTKRP kernel [37] and HiCOO explores new coordinate-based formats that further improves performance [25]. Nisa et al. describe techniques for how to effectively parallelize the MTTKRP kernel for GPUs [28]. Finally, the Cyclops library shows how to scale sparse kernels to distributed machines [38].

Chapter 10

Conclusion and Future Work

This thesis presents a comprehensive theory of 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 it 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. With this work, sparse tensor algebra is finally on the same optimization and code generation footing as dense tensor algebra and array codes.

While my prototype provides a system that is usable by experts and demonstrates the efficacy of the transformations described in this thesis, there is more work needed to allow this work to be used by non-experts. Autoscheduling is a necessity for users who are not performance engineers. Additionally, a higher-level scheduling API is needed to make standard optimization patterns apparent to users. The `pos` and `coord` commands in the scheduling API force the user to recognize the costs associated with mode switching between tiling different spaces. But they are also easily mistaken as commands that change how the space should be iterated. A higher-level API could hide these commands and expose this overhead in another way.

The current implementation is unable to parallelize computations with sparse results for reasons described in the *Parallelize* preconditions (Section 5.7). Many sparse computations rely on a sparse result so relaxing this precondition by allowing for assembling and merging per-thread copies of a tensor is also important future work. Finally, this work only supports the formats described in the original TACO paper [22]. Extending these optimization strategies to all formats described in [10] would greatly expand the number of useful kernels that can be generated.

This work has also been prototyped such that it is possible to explore extensions to new hardware and new optimization strategies. As most optimizations are done at a higher-level representation, specifics of a new hardware platform and language can be added with relative ease. Dense and sparse tensor algebra accelerators such as TPU [20] and ExTensor [14] are becoming increasingly numerous and prevalent. I expect that unifying the optimization strategies for these accelerators into a single framework will become increasingly necessary. While there are plenty of different optimization strategies to explore, I believe that the most important ones to explore first are those that consider how to more efficiently split up and load-balance a computation. In this thesis, I explore two basic transformations *Split* and *Divide*. These could be extended to handle optimization strategies such as dynamic load- balancing, using different optimization strategies based on the row length, bucketing rows by number of nonzeros, and splitting based on different types of searches such as the merge-path search [27]. If these transformations were extended to allow for a user-provided function to determine the split points, then these different strategies could be explored without making changes to the compiler.

Appendix A

Schedules

A.1 Scheduled TACO SPMV CPU (Figure 8-1)

```
int CHUNK_SIZE = 32;
IndexVar i("i"), j("j");
y(i) = A(i, j) * x(j);

IndexVar i0("i0"), i1("i1"), kpos("kpos"), kpos0("kpos0"), kpos1("kpos1");
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 8-2)

```
int NNZ_PER_THREAD = 7;
int NNZ_PER_WARP = 7 * 32;
int NNZ_PER_TB = 7 * 512;
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 SPMM CPU (Figure 8-3)

```
int CHUNK_SIZE = 8;
int TILE_SIZE = 8;
IndexVar i("i"), j("j"), k("k");
C(i, k) = A(i, j) * B(j, k);

IndexVar i0("i0"), i1("i1"), kbounded("kbounded"), k0("k0"), k1("k1");
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.4 Scheduled TACO SPMM GPU (Figure 8-4)

```
int NUM_COLS = 128;
int NNZ_PER_WARP = 16;
int NNZ_PER_TB = 16 * (512 / 32);
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.5 Scheduled TACO MTTKRP Order-3 CPU (Figure 8-5)

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

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

A.6 Scheduled TACO MTTKRP Order-4 CPU (Figure 8-5)

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

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.7 Scheduled TACO MTTKRP Order-5 CPU (Figure 8-5)

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

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.8 Scheduled TACO MTTKRP GPU (Figure 8-6)

```
int NUM_COLS = 32;
int NNZ_PER_WARP = 4;
int NNZ_PER_TB = 4 * (512 / WARP_SIZE);
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.9 SpMV Thread per Row on GPU (Section 8.3)

```
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.10 SpMV Warp per Row on GPU (Section 8.3)

```
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.11 SpMV on GPU with no Unrolling (Section 8.3)

```
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.12 SpMM on CPU Tiled (Section 8.6)

```
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.13 SpMM on CPU No Tiling Section 8.6)

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

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

Bibliography

[1] 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. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 265–283, Berkeley, CA, USA, 2016. USENIX Association.

[2] Khalid Ahmad, Anand Venkat, and Mary Hall. Optimizing lobpcg: Sparse matrix loop and data transformations in action. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 218–232. Springer, 2016.

[3] 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. Automatic code generation for many-body electronic structure methods: the tensor contraction engine. *Molecular Physics*, 104(2):211–228, 2006.

[4] Travis Augustine, Janarthanan Sarma, Louis-Noël Pouchet, and Gabriel Rodríguez. Generating piecewise-regular code from irregular structures. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 625–639, New York, NY, USA, 2019. ACM.

[5] Marouane Belaoucha, Denis Barthou, Adrien Eliche, and Sid-Ahmed-Ali Touati. FADAlib: an open source C++ library for fuzzy array dataflow analysis. *Procedia Computer Science*, 1(1):2075–2084, May 2010.

[6] Nathan Bell and Michael Garland. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. In *International Conference for High Performance Computing, Networking, Storage, and Analysis*, pages 18:1–18:11, Portland, Oregon, 2009. ACM.

[7] Aart J. C. Bik. *Compiler Support for Sparse Matrix Computations.* PhD thesis, 1996.

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

[9] 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. 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*, pages 578–594, Carlsbad, CA, oct 2018. USENIX Association.

[10] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Format Abstraction for Sparse Tensor Algebra Compilers. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):123:1–123:30, nov 2018.

[11] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), December 2011.

[12] Evgeny Epifanovsky, Michael Wormit, Tomasz Kuś, Arie Landau, Dmitry Zuev, Kirill Khistyaev, Prashant Manohar, Ilya Kaliman, Andreas Dreuw, and Anna I. Krylov. 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):2293–2309, 2013.

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

[14] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. ExTensor: An accelerator for sparse tensor algebra. *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, pages 319–333, oct 2019.

[15] Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics (TOG)*, 38(6):201, 2019.

[16] Eun-Jin Im and Katherine Yelick. Optimizing Sparse Matrix Computations for Register Reuse in SPARSITY. In *Computational Science âĂŤ ICCS 2001*, Lecture Notes in Computer Science, pages 127–136. Springer, Berlin, Heidelberg, May 2001.

[17] Intel. Intel math kernel library reference manual. Technical report, 630813-051US, 2012. `http://software.intel.com/sites/products/documentation/hpc/mkl/mklman/mklman.pdf`, 2012.

[18] Inah Jeon, Evangelos E. Papalexakis, U Kang, and Christos Faloutsos. Haten2: Billion-scale tensor decompositions. In *IEEE International Conference on Data Engineering (ICDE)*, 2015.

[19] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[20] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12. IEEE, 2017.

[21] Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, and Saman Amarasinghe. Tensor Algebra Compilation with Workspaces. In *International Symposium on Code Generation and Optimization*, pages 180–192, Washington, DC, feb 2019. IEEE Press.

[22] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The Tensor Algebra Compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):77:1 – 77:29, oct 2017.

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

[24] Leslie Lamport. The parallel execution of do loops. *Commun. ACM*, 17(2):83âĂŞ93, February 1974.

[25] Jiajia Li, Jimeng Sun, and Richard Vuduc. HiCOO: Hierarchical storage of sparse tensors. *International Conference for High Performance Computing, Networking, Storage, and Analysis*, pages 238–252, 2019.

[26] Kathryn S McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4):424–453, 1996.

[27] Duane Merrill and Michael Garland. Merge-Based Parallel Sparse Matrix-Vector Multiplication. *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, (November), 2016.

[28] Israt Nisa, Jiajia Li, Aravind Sukumaran-Rajam, Richard W. Vuduc, and P. Sadayappan. 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*, pages 123–133, 2019.

[29] NVIDIA V10.1.243. cuSPARSE Software Library, 2019.

[30] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

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

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

[33] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, 31(4):32:1–32:12, July 2012.

[34] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 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, pages 519–530, New York, NY, USA, 2013. ACM.

[35] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. FROSTT: The formidable repository of open sparse tensors and tools, 2017.

[36] Shaden Smith, Niranjay Ravindran, Nicholas Sidiropoulos, and George Karypis. Splatt: Efficient and parallel sparse tensor-matrix multiplication. In *2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 61–70, May 2015.

[37] Shaden Smith, Niranjay Ravindran, Nicholas Sidiropoulos, and George Karypis. SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication. In *IEEE International Parallel and Distributed Processing Symposium*, pages 61–70. IEEE, may 2015.

[38] Edgar Solomonik, Maciej Besta, Flavio Vella, and Torsten Hoefler. Scaling betweenness centrality using communication-efficient sparse matrix multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 47:1–47:14, New York, NY, USA, 2017. ACM.

[39] Edgar Solomonik, Devin Matthews, Jeff R. Hammond, John F. Stanton, and James Demmel. A massively parallel tensor contraction framework for coupled-cluster computations. *Journal of Parallel and Distributed Computing*, 74(12):3176–3190, December 2014.

[40] Michelle Mills Strout, Geri Georg, and Catherine Olschanowsky. Set and Relation Manipulation for the Sparse Polyhedral Framework. In *Languages and Compilers*

for Parallel Computing, Lecture Notes in Computer Science, pages 61–75. Springer, Berlin, Heidelberg, September 2012.

[41] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. The sparse polyhedral framework: Composing compiler-generated inspector-executor code. *Proceedings of the IEEE*, (99):1–15, 2018.

[42] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. Technical report, jun 2018.

[43] Anand Venkat, Mary Hall, and Michelle Strout. Loop and data transformations for sparse matrix code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 521–532, 2015.

[44] Anand Venkat, Mahdi Soltan Mohammadi, Jongsoo Park, Hongbo Rong, Rajkishore Barik, Michelle Mills Strout, and Mary Hall. Automating wavefront parallelization for sparse matrix computations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 41. IEEE Press, 2016.

[45] R. Vuduc, J.W. Demmel, K.A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply. pages 26–26. IEEE, 2002.

[46] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, Oct 1991.

[47] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. *SIGPLAN Not.*, 26(6):30–44, May 1991.

[48] Michael Joseph Wolfe. *Optimizing Supercompilers for Supercomputers*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1982. AAI8303027.

[49] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance graph dsl. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):121, 2018.