SuperTaco: Taco Tensor Algebra Kernels on Distributed Systems Using Legion

by

Sachin Dilip Shinde

S.B., Massachusetts Institute of Technology (2018)

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 2019

Author ....
Department of Electrical Engineering and Computer Science
February 8, 2019

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

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

SuperTaco: Taco Tensor Algebra Kernels on Distributed Systems Using Legion

by

Sachin Dilip Shinde

Submitted to the Department of Electrical Engineering and Computer Science
on February 8, 2019, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Tensor algebra is a powerful language for expressing computation on multidimensional data. While many tensor datasets are sparse, most tensor algebra libraries have limited support for handling sparsity. The Tensor Algebra Compiler (Taco) has introduced a taxonomy for sparse tensor formats that has allowed them to compile sparse tensor algebra expressions to performant C code, but they have not taken advantage of distributed systems.

This work provides a code generation technique for creating Legion programs that distribute the computation of Taco tensor algebra kernels across distributed systems, and a scheduling language for controlling how this distributed computation is structured. This technique is implemented in the form of a command-line tool called SuperTaco. We perform a strong scaling analysis for the SpMV and TTM kernels under a row blocking distribution schedule, and find speedups of 9-10x when using 20 cores on a single node. For multi-node systems using 20 cores per node, SpMV achieves a 33.3x speedup at 160 cores and TTM achieves a 42.0x speedup at 140 cores.

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

Acknowledgments

I thank my thesis supervisor, Saman Amarasinghe, and my colleague, Fredrik Kjolstad, for their help throughout my time at MIT. Saman has guided me through the world of performance engineering and given me countless opportunities, and Fred's insights and feedback have been indispensable to me as a researcher.

Finally and most importantly, I thank my family. My parents, Pratima and Dilip, my sister, Ranjita, my brother-in-law, David, and my nephews, Bryce and Dylan. Their unwavering support pushes me to greater heights.

Contents

List of Figures

Chapter 1

Introduction

Tensors are a generalization of vectors and matrices to multiple dimensions, and permit algebraic operations that can be conveniently expressed using summation notation, e.g. $A_{ij} = \sum_k B_{ijk}c_k$. This tensor algebra is a powerful language for expressing computations involving multidimensional data, and is central in many practical applications and fields of study, such as chemometrics, signal processing, graph analysis, computer vision, and machine learning [1]. While numerous software tools and libraries have emerged for computing with tensor algebra, their capabilities have been limited when dealing with sparse tensors, which are tensors with many zeros that can be compressed to reduce storage space and computation. This is problematic considering many large real-world datasets are sparse tensors, such as Netflix ratings [2], Facebook ratings [3], and Amazon reviews [4].

Recently Kjolstad *et al.* have developed the Tensor Algebra Compiler (Taco) [5] [6], a library for compiling dense and sparse tensor algebra expressions into C code. Their work provides a taxonomy for tensor storage formats, which is capable of describing many widely-used sparse matrix formats such as compressed sparse row (CSR), compressed sparse column (CSC), doubly compressed sparse row (DCSR), Ellpack (ELL), and Diagonal (DIA), and generalizes to sparse tensor formats such as compressed sparse fiber (CSF). Given a tensor algebra expression and the tensor formats of the expression operands, their library can generate C code loop nests that compute the expression. While their generated code can take advantage of multicore systems using OpenMP, they have not taken advantage of distributed systems.

One of the difficulties of distributing the computation of Taco-generated code across nodes is that it requires the tensor data to be split into blocks and for those blocks to be distributed only to the nodes that make use of those blocks in their computations. The Legion programming system [7] is a runtime programming system for developing parallel applications and libraries which has abstractions that allow the expression of these block dependencies succinctly. Specifically, Legion tasks can be used to describe per-node computation, and Legion logical regions can be used to describe the tensor data blocks needed by those computations.

This work provides a method to generate Legion programs for distributing the computation of Taco tensor algebra expressions across distributed systems. This Legion program generator, which we call SuperTaco, is implemented using C/C++ for the generated Legion programs and Python with Mako templating for program generation. Our method introduces a *distribution schedule* for computation, which describes how the Taco computation is distributed among nodes, and is capable of expressing several different paradigms for distribution such as owner's compute [8]. We find that for large random tensors and row blocking distribution schedules, these generated programs exhibit a strong scaling efficiency near 50% for SpMV and 58% for TTM in the single-node case, achieving speedups around 10x on 20 cores. For the multi-node case with 20 cores per node, performance peaks with SpMV at 160 cores with a speedup of 33.3x (21% strong scaling efficiency), while performance peaks with TTM at 140 cores with a speedup of 42.0x (30% strong scaling efficiency). Adding more nodes past this point degrades efficiency due to node-to-node overhead. In these experiments, SpMV operated on a $50\,000\,000 \times 50\,000\,000$ sparse matrix (density 1×10^{-7}) and a length $50\,000\,000$ dense vector, while TTM operated on a $500\,000 \times 50\,000 \times 5000$ sparse tensor (density 1×10^{-6}) and a 32×5000 dense matrix.

Using the methods described in this work, tensor libraries can distribute tensor algebra computation across distributed systems through the Legion programming system. This work contributes to the future development of tensor libraries by (1) developing a technique for distributing tensor algebra computation across multiple nodes, (2) introducing a taxonomy for describing how to distribute tensor algebra computation via distribution schedules, and (3) providing a scaling analysis of the Legion programs generated through these methods.

Chapter 2

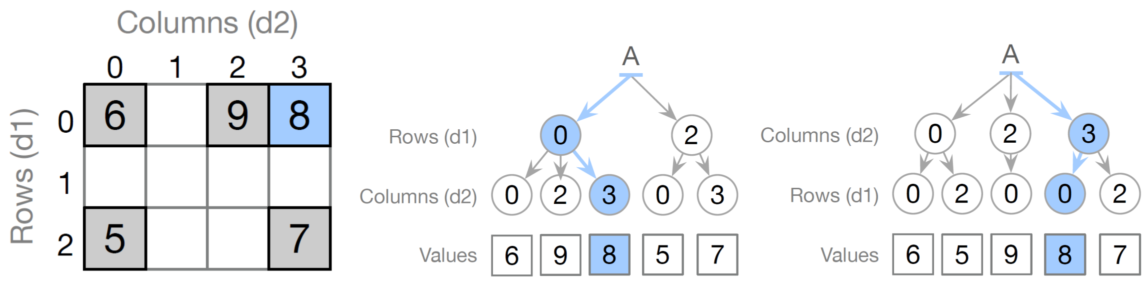
Background

This chapter gives an overview of the relevant concepts and terminology needed to talk about tensor algebra, Taco tensor storage formats, Taco code generation, and the Legion programming system in the context of distributing Taco computation. For more information regarding Taco code generation, please refer to Kjolstad *et al.* [5]; for more information regarding the Legion programming system, please refer to Bauer *et al.* [7].

2.1 Tensor Notation

The notation and terminology used for tensors is similar to that used in the Taco library. A tensor A of *rank* (or *order*) R is an association between R-tuples (or *coordinates*) $(i_0, i_1, \ldots, i_{R-1})$ of non-negative integers and tensor *entries* (or *components*). These components are written using tensor notation as $A_{i_0 i_1 \ldots i_{R-1}}$, where i_m is an *index variable* (or *index*). The position of an index variable in the tensor's tuple is known as its *mode*, so that the m^{th} mode refers to index variable i_m. The non-negative integers that are possible for a given mode are limited by the *size* (or *dimension*) of that mode, so that if mode m has dimension N_m, then $i_m < N_m$.

Using this notation allows us to express tensor algebra operations with summations, e.g. $A_{ij} = \sum_k B_{ijk} c_k$ represents evaluating tensor component A_{ij} for a given i and j by summing $B_{ijk} C_k$ for all values of index variable k. The index variables that



(a) Sparse matrix A (b) Storage format tree for index variable ordering (d_1, d_2) (c) Storage format tree for index variable ordering (d_2, d_1)

Figure 2-1: A 2nd-rank tensor $A_{d_1 d_2}$ and its possible storage format trees [5].

appear in the output tensor (i and j in this example) are known are *free variables*, while any index variables that don't appear in the output (k in this example) are known as *reduction variables*.

2.2 Taco Tensor Storage Formats

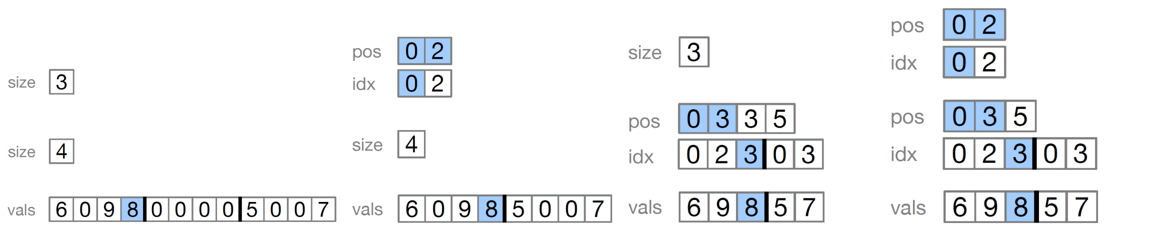
An important consideration when computing with tensors is the format in which the tensor data is stored, which can be specialized to accommodate underlying structure in the tensor data. In particular for sparse tensors (which contain a large percentage of zero components), the zeros can be compressed to reduce storage space and avoid computations that involve zeros. As mentioned previously, there are many commonly-used sparse storage formats, which compress the data in different ways. The Taco library introduces a taxonomy for constructing these storage formats from a set of base primitives.

For a tensor $A_{i_0 i_1 \ldots i_{R-1}}$, a Taco storage format consists of an ordering of the index variables (i.e. a permutation of $(i_0, i_1, \ldots, i_{R-1})$) and a declaration for each index variable as being stored as either a *sparse level* or a *dense level*. The ordering of the index variables allows us to express the non-zero components of the tensor using a tree, where each tree level corresponds to an index variable and the nodes on a level are the values of the corresponding index variable. Specifically, for an index variable ordering $(i_{\sigma_0}, i_{\sigma_1}, \ldots, i_{\sigma_{R-1}})$, there is a node on level k for each k-tuple $(j_0, j_1, \ldots, j_{k-1})$ for which there is a non-zero entry $(i'_0, i'_1, \ldots, i'_{R-1})$ in A satisfying $j_q = i'_{\sigma_q}$ for $0 \leq q < k$, and its parent is the node in level $k-1$ corresponding to tuple $(j_0, j_1, \ldots, j_{k-2})$. (Since a node's parent corresponds to its tuple's prefix, the node on level k need only show the value of j_{k-1}, i.e. the value of level k's index variable i_{σ_k}. The rest of the tuple is implicit from the path to the root.) The R-tuples corresponding to the leaf nodes of the tree then consist of all the non-zero entries of A. As an example, consider the sparse matrix $A_{d_1 d_2}$ as shown in Figure 2-1a. If the index variable ordering is (d_1, d_2), the storage format tree is given in Figure 2-1b; if the index variable ordering is (d_2, d_1), the tree is instead given by Figure 2-1c.

The Taco storage format effectively stores the format tree given by the index variable ordering, keeping a data structure for each level along with a data structure for the tensor component values. The data structure for each level depends on whether the corresponding index variable is declared dense or sparse. For dense levels, the data structure only consists of metadata, namely the dimension of the index variable. Conceptually, this is because dense levels assume that the corresponding index variable can take on all values in the range $[0, D)$ for index variable dimension D. More concretely, for a dense level k, it assumes that each node in level $k-1$ has D children, regardless of whether there's actually a non-zero entry in A satisfying the aforementioned conditions for k-tuple $(j_0, j_1, \ldots, j_{k-1})$. This allows for efficient storage of the dense level, but potentially inefficient storage of lower levels depending on the validity of the assumption.

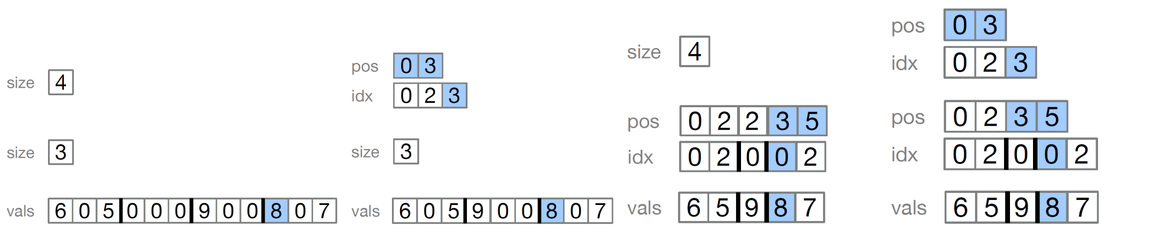
Sparse levels, in contrast to dense levels, only store the non-zero entries of the level. Sparse levels specifically keep a `pos` array and an `idx` array, where the `idx` array consists of several segments stored contiguously and the `pos` array keeps track of the positional bounds of the segments in the `idx` array. That is, segment i runs from `pos[i]:pos[i+1]` in the `idx` array, where the last element of the `pos` array is a sentinel equal to the length of `idx`. Each segment corresponds to a parent node in the previous level, and the segment itself stores the index variable values of the parent's children, in sorted order.

The data structure containing the tensor component values consists of a `val` array, with an entry for every leaf node in the tree. Note that there may be zero entries



(a) (dense$_{d_1}$, dense$_{d_2}$) (b) (sparse$_{d_1}$, dense$_{d_2}$) (c) (dense$_{d_1}$, sparse$_{d_2}$) (d) (sparse$_{d_1}$, sparse$_{d_2}$)

Figure 2-2: Possible storage formats for index variable ordering (d_1, d_2) [5].



(a) (dense$_{d_2}$, dense$_{d_1}$) (b) (sparse$_{d_2}$, dense$_{d_1}$) (c) (dense$_{d_2}$, sparse$_{d_1}$) (d) (sparse$_{d_2}$, sparse$_{d_1}$)

Figure 2-3: Possible storage formats for index variable ordering (d_2, d_1) [5].

in the `val` array, since dense levels assume that the number of children per parent node is equal to the corresponding index variable's dimension. Figure 2-2 shows the storage formats for index variable ordering (d_1, d_2), while Figure 2-3 shows the storage formats for index variable ordering (d_2, d_1).

Note that dense levels also implicitly consist of a `pos` and `idx` array, similar to sparse levels. In the dense case, each segment contains all the integers in the range $[0, D)$ for a level with dimension D, which makes `pos[i]` $= D \cdot i$ for segment i. Storing only the dimension D is just a form of compression here. This view makes iterating through tensors easier to conceptualize, since we can just think of each data structure as having an effective position pointer which points into the data structure's `idx` array (for the component value data structure, the effective position pointer just points into the `val` array). More concretely, for sparse level k, the effective position p_k satisfies `pos[`p_{k-1}`]` $\leq p_k <$ `pos[`$p_{k-1}+1$`]` and indicates index value `idx[`p_k`]`. For dense level k, the effective position p_k satisfies $\lfloor p_k/D \rfloor = p_{k-1}$ and indicates index value $p_k - D\lfloor p_k/D \rfloor$. For the component value, the effective position satisfies $p_k = p_{k-1}$ and indicates component value `val[`p_k`]`. Iterating through the tensor involves advancing these effective positions while maintaining the invariants. Note that given the effective position for the `val` array, we can uniquely determine the effective positions for the other levels; this makes sense intuitively, since the `val` array's effective position points to a component of the tensor.

2.3 Taco Code Generation

Using the tensor format taxonomy from the previous section, the Taco library can compile tensor algebra expressions involving tensors of given formats into C code. While an understanding of tensor storage formats is critical to understanding how to break tensor data into blocks and distribute those blocks across nodes, the specific details of Taco code generation are less relevant. The techniques used in this work largely treat Taco code generation as a black-box, and only modify the generated C code slightly to account for storage format modifications. This is one of the benefits of our methods, since it means changes to the internals of Taco code generation will not require significant changes to our methods. It also means that these techniques can be applicable to other tensor algebra libraries, provided they use similar tensor storage format taxonomy to Taco. In this context, it is more important to understand the nature and structure of the generated code than to understand how it is generated.

The Taco library takes a tensor algebra expression as input, written using the Einstein summation convention, and transforms each summation into a loop nest in C code. The storage formats of the tensors impose restrictions on the order of those loop nests for iteration to be efficient, and the Taco library may accordingly rearrange loop nests to satisfy those restrictions. We access the Taco library through the Taco command-line tool, which generates two functions for computing the tensor algebra kernel. The first is the `assemble()` function, which performs part of the computation to allocate space for the data structures in the output tensor and calculate the output indices. The second is the `compute()` function, which performs the rest of the computation to calculate the component values of the output tensor. Our Legion program generation routine uses the Taco command-line tool to generate `assemble()`

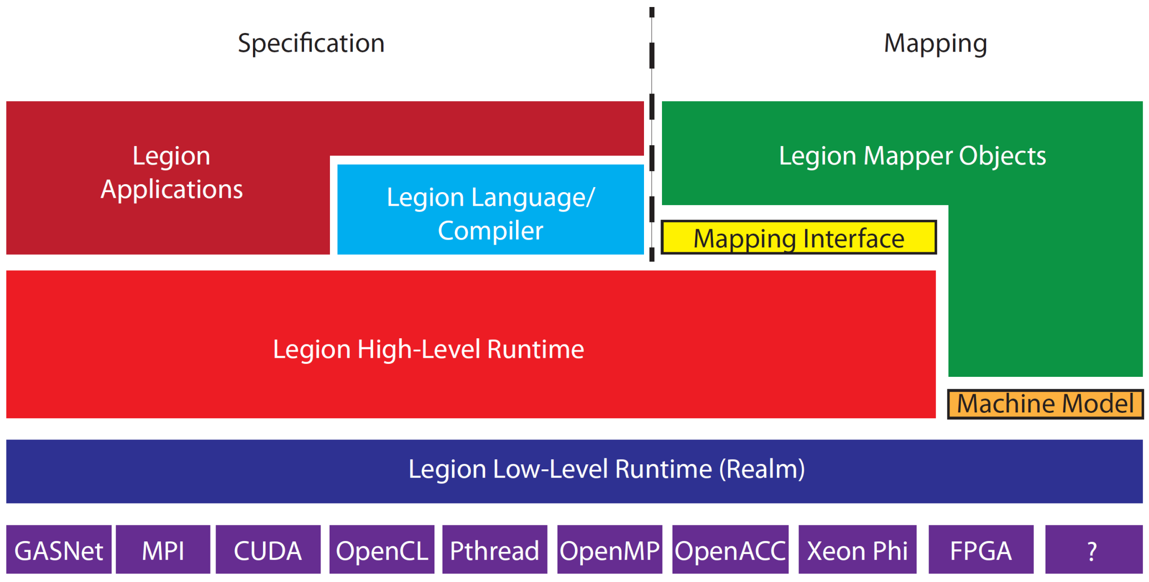


Figure 2-4: The Legion programming system software stack [9].

and `compute()` functions for the appropriate tensor algebra kernels, modifies them slightly to allow for computation using tensor blocks, and then emits the C code into the Legion program. For an overview of how Taco compiles tensor algebra expressions and the optimizations it uses, please refer to Kjolstad *et al.* [5].

2.4 Legion Programming System

Legion is a parallel programming system that eases high-performance distributed application development through the use of abstractions that allow the programmer to explicitly express the locality and independence of program data [7]. Legion runs on top of the Realm runtime, which uses GASNet for high-performance networking on distributed systems; the full runtime stack is shown in Figure 2-4.

The main primitive in Legion is the *logical region*, an abstraction for storing program data similar to a table in a relational database, which uses an *index space* for rows and a *field space* for columns. The index space can be a single- or multi-dimensional array of points, either structured (indicating the index space is a collection of one or more dense Cartesian grids) or unstructured (indicating the index space is an arbitrary collection of keys or pointers). A field space consists of a collection

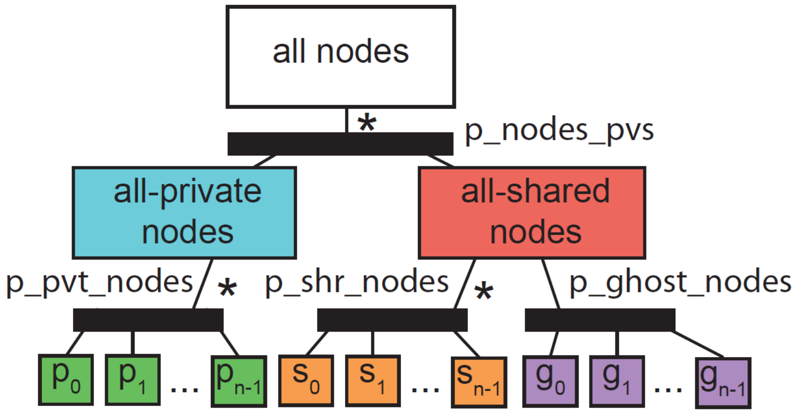


Figure 2-5: An example of a region tree, where the black bars represent partitions and the asterisks indicate disjoint partitions [7].

of named fields, each of which may either store a plain old data (POD) type or a compound type. Logical regions are effectively a cross product between a field space and an index space, and are first-class objects in Legion, i.e. they can be created and destroyed at runtime and stored in data structures.

Logical regions can be partitioned into logical sub-regions through *colorings*, which are maps from colors (which lie in a *color space*) to sets of points in the logical region's index space. Each color in the coloring map thus defines an index sub-space, which in turn defines a logical sub-region. If the index sub-spaces are *disjoint*, then the logical partition is considered disjoint; otherwise, the partition is considered *aliased*. These partitions create a region tree, which codifies the dependencies between logical sub-regions in the same tree (see Figure 2-4 for an example). It is through this primitive that Legion allows the programmer to indicate program data dependencies.

For performing computation, Legion uses *tasks*, which are functions that operate on logical regions and pass-by-value arguments. Legion tasks declare the regions they access through *region requirements*, which specify the fields of a region being accessed along with the *privilege* being used to access it (read-only, read-write, write-discard, or reduce). The privilege specifies the type of side effect the computation is allowed to have on the region. Region requirements thus allow the programmer to express the dependencies of computation on data.

Tasks can in turn launch sub-tasks asynchronously using a deferred execution model, which forms a tree of tasks with the top-level root task being the first to execute. The Legion runtime performs a dependency analysis on tasks to discover implicit parallelism by checking whether tasks are non-interfering on their logical regions; this allows the runtime to execute these tasks in parallel while maintaining sequential program order semantics. Legion requires that for a sub-task to declare privileges on a region, its parent task must possess the same privileges (or stronger) for the same fields and for the same logical region (or an ancestor in its region tree); this *containment property* allows the use of an efficient hierarchical scheduling algorithm for tasks, and makes Legion functional at coarse granularity (between tasks) while imperative at fine granularity (within a task). A sub-task that creates a logical region is granted read-write privileges on that region, and will pass those privileges to its parent task if it does not destroy the region by the end of its execution. Sub-tasks may additionally return values to their parent tasks in the form of futures, which can either be passed to other sub-tasks (causing those sub-tasks to delay execution until the future's result is available) or waited on in the parent task (causing the parent task to block until the future's result is available).

When a task requires access to data contained with a logical region, it has to instantiate that logical region in memory with some data layout; this process is called *mapping* the logical region, and the resulting data in memory is referred to as a *physical instance* of that logical region. When a task runs on a processor, the Legion runtime guarantees that the logical regions specified in the task's region requirements will be mapped to physical instances in that processor's memory. Tasks may also manually map logical regions during execution, but only if the task has the necessary privileges on the logical region; this is called *inline mapping*, and inline mapping launchers are similarly set up through region requirements. (This may block execution if the task attempts to access the instance before the logical region is mapped.) One consequence of these mapping semantics is that a task must manually unmap a logical region if it launches a sub-task that would require interfering privileges on that same logical region, since otherwise the sub-task would never execute.

Legion exposes scheduler decisions regarding how to map tasks to processors and how to map logical regions to processor memories using the mapper interface. *Mappers* are classes developed by the programmer that implement the mapper interface; the Legion runtime instantiates a mapper object for every mapper type registered with the runtime and for every processor in the system. Once a sub-task is launched, the mapper that corresponds to the processor running the parent task is queried with the sub-task using its `select_task_options()` call. During this call, the mapper can decide to do one of three things. It can map the sub-task inline, which causes it to be inlined into the parent task using the parent task's regions. It can choose to map the sub-task remotely, in which case the mapper selects a remote processor to send the sub-task to (whose corresponding mapper then maps the task). Finally, the sub-task can be sent to a remote processor but mapped locally, in which case the current mapper maps the sub-task and sends the results to the remote node, which then uses the results to immediately launch the task.

The act of mapping itself is carried out using the mapper's `map_task()` call. The mapper in this call decides the layout of the physical instances needed by the task, and for each physical instance ranks the physical memories with regards to which memory is the most desirable to instantiate the physical instance on. The mapper also has the ability in this call to specify that the task may by executed by any processor within a processor group, which provides a form of load-balancing to task execution. To learn more about the Legion runtime's software out-of-order processor (SOOP) and the algorithms it uses for dependency analysis and task scheduling, refer to Bauer's dissertation [9].

Chapter 3

Related Work

In this chapter, we turn our focus to previous works that have parallelized the computation of sparse tensor algebra. To the best of our knowledge, the Taco compiler is the first compiler capable of generating kernel code for general sparse tensor algebra expressions. Thus, distributing the computation of these expressions in the general case presents a novel problem. This makes it difficult to compare to prior work, which often has focused on parallelizing very specific tensor algebra expressions and has accordingly developed solutions that may not generalize to other expressions. Nevertheless, we have still gained insight from seeing approaches to distributing these computations.

3.1 SpMV Partitioning

Much effort has been put into optimizing the sparse matrix-vector (SpMV) kernel $a_i = \sum_j B_{ij} c_j$ due to its heavy use in scientific computing, which has given rise to several optimization techniques [10]. To exploit thread-level parallelism, the matrix data has to be partitioned in some way. Earlier work has suggested partitioning by row blocks, column blocks, or by sections of the sparse matrix data structure [10], while recent work has led to more complex, specialized techniques.

For example, using row splitting, long rows of CSR matrices are split into regularized sections and distributed to processors, with the partial sums aggregated later in a subsequent pass [11]. This has the benefit of limiting the amount of data assigned to any given thread and thus improves load balance compared to simply blocking along rows, but still possesses some load imbalance due to rows which are shorter than the regularized size. Another modern technique is nonzero splitting, which assigns equal amounts of non-zero data in CSR matrices to each processor, but requires a search in the `pos` array to find which rows the non-zero data sections begin and end on [11]. This improves load balancing by directly dividing up the non-zero data, but is still susceptible to imbalance if some non-zero data sections cover many more rows than other sections.

While these modern partitioning techniques improve load balancing for the SpMV kernel, they don't generalize past SpMV well since they require random access into the vector to be able to compute the output efficiently. More generally, these techniques are attempting to partition the set of tensor components into disjoint sets, and assigning these sets to processors in such a way that the time a processor spends iterating through its components is around the same per processor. However, iterating through the tensor's index variables in this complex fashion, while designed to be efficient for the partitioned tensor, may not be efficient for other tensors in the computation which depend on the same index variables. If any of these other tensors is similar in size to the partitioned tensor, the inefficiencies of this iteration will outweigh the benefits of the complex partitioning.

It is for this reason that we use tensor blocking based on index splitting, as described in Section 4.1. This in effect allows us to partition all the tensors simultaneously in such a way that iteration for a given processor is not rendered inefficient by the partitioning. At first glance, this suggests complex partitions may be salvageable through partitioning the other tensors to match in index variables, similar to index splitting. However, this does not necessarily result in a partition. For example, consider row splitting for SpMV but with a sparse vector; the parts of the sparse vector that correspond to the regularized row sections overlap each other and thus do not constitute a partition, meaning the approach would require extra work through duplicating data.

3.2 PETSc

PETSc is a suite of data structures and routines for developing parallel solutions to problems modelled by partial differential equations. The suite in particular is notable since it manages to provide a clean object-oriented abstraction for computing with sparse matrices and vectors in parallel without exposing the underlying MPI details [12]. The suite specifically provides a parallel sparse matrix type called `mpiaij` that uses row blocking to distribute the matrix data across processors, and further separates each row block into a diagonal block and an off-diagonal block (both stored using CSR). While the suite does not support sparse tensors, the suite makes computing parallel SpMV as simple as computing single-threaded SpMV since it doesn't expose the underlying parallel implementation; the user just calls `MatMult()` in both cases. SuperTaco is somewhat similar, as the tensor algebra expression remains unchanged by parallelizing the computation. However, SuperTaco uses a distribution schedule to control how computation is split among nodes; for PETSc, changing how computation is distributed would require changing the underlying matrix object to a different parallel implementation. In this sense, PETSc associates this property with the matrix, while SuperTaco associates it with the computation itself.

Chapter 4

SuperTaco

Chapter 2 gave an introduction to tensor storage formats, Taco code generation, and the Legion programming system. In this chapter, we describe our implementation of SuperTaco, a compiler that takes sparse tensor algebra expressions and generates Legion programs that distribute the computation of those expressions across homogeneous distributed systems. The SuperTaco command-line tool effectively wraps the Taco command-line tool, using it as a black-box to generate kernel code to run on each processor.

The SuperTaco command-line tool accepts similar arguments to the Taco command-line tool. At its core, it accepts a tensor algebra expression written using Einstein summation convention, such as `a(i) = B(i,j) * c(j)` to express $a_i = \sum_j B_{ij} c_j$. It requires that the storage formats of the tensor operands in the expression be specified. E.g. to specify that B_{ij} is a CSR matrix $(\text{dense}_{d1}, \text{sparse}_{d_2})$, we would use the format flag `-f:B:ds`. SuperTaco additionally accepts arguments that can control how it distributes computation, which we refer to as the distribution schedule.

The SuperTaco command-line tool itself is a Python script. During execution, it processes the arguments and distribution schedule, and generates a C++ Legion program from a template using the Mako templating library. SuperTaco does not dynamically generate all the code in the program; some routines, such as iterating through a tensor or binary-searching a tensor, are static, and are accordingly packaged in a static library that the generated Legion program uses. SuperTaco also at times

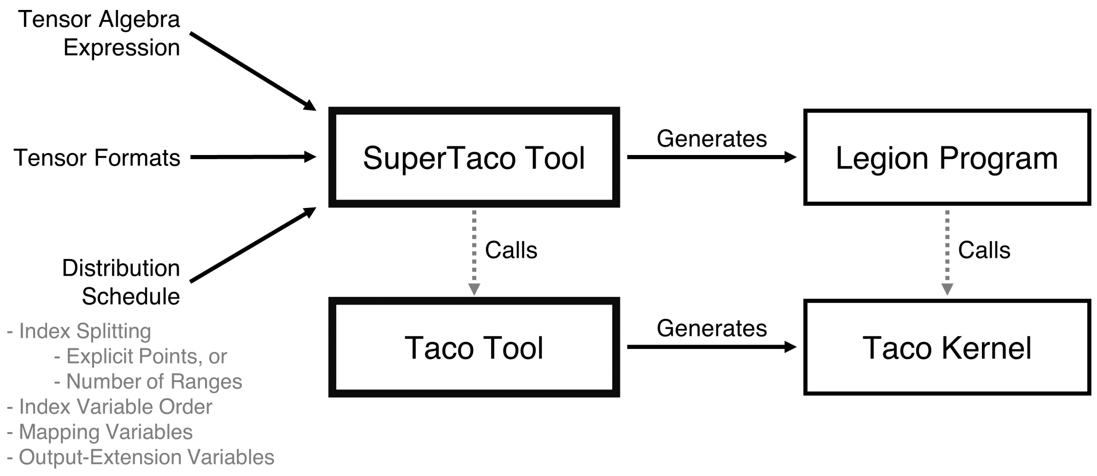


Figure 4-1: Overview of the SuperTaco command-line tool.

needs to compile tensor algebra kernels to single-processor C code; to do this, it calls the Taco command-line tool to generate the code, packs it into a C source file after slight modifications, and then has the Legion program call the functions in the file. Compiling the generated Legion program into an executable is performed using the standard Legion Makefile distributed to Legion application developers. An overview of the SuperTaco tool components are shown in Figure 4-1.

SuperTaco extracts parallelism from tensor algebra expressions by breaking tensors into blocks, and computing the requested kernel on those blocks instead of on the entire operand tensors at once. Since the requested kernel must be computed on many different combinations of these blocks, the kernel must be executed many times, and many of these executions can be run in parallel. However, dependencies can arise between the kernel executions depending on how the distribution takes place, e.g. if two of these kernel operations sum into the same output block.

A Legion program generated by SuperTaco consists of three phases: distributed tensor loading, distributed tensor algebra computation, and distributed tensor storing. In the first phase, a tensor (potentially larger than any single node's memory) is streamed through the nodes and loaded into blocks, which are evenly distributed over the nodes. In the second phase, the kernel is computed in parallel for various combinations of these blocks. In Legion, this amounts to a nest of for loops in the top-level task, with a kernel task launch in the innermost loop body. Through region requirements, the task launch receives references to the logical regions containing the blocks that the kernel task must operate on. When the kernel task finishes computing an output block, the block is packed in a logical region and passed back to the parent through a future. The parent task adds this future to the next kernel task that operates on that output block, which codifies the dependencies between kernel tasks. At the end of this phase, the Legion program adds together output blocks that were treated separately during kernel computation, but in reality refer to the same block (the extent to which this happens is one of components controlled by the distribution schedule). In the third phase, the tensor blocks are merged together via a distributed multi-way merge, and are then written back to disk.

Section 4.1 describes the theoretical notation and primitives needed to discuss tensor blocking and index splitting, while Section 4.2 describes how we augment Taco tensor storage formats to allow more efficient tensor block storage. In Section 4.3, we discuss a file format for storing the data structures of Taco tensor storage formats, and how to convert to this file type from the `.tns` file type. Section 4.4 discusses how our Legion programs load a tensor and partition it into tensor blocks in a distributed fashion. Section 4.5 presents the theory behind how a tensor algebra expression is transformed by tensor blocking, and describes the code generation techniques that SuperTaco uses to compute these transformed expressions along with the options that arise for structuring that computation, which is exposed through distribution schedules. Finally section 4.6 describes how our Legion programs merge the output tensor blocks into an output tensor in a distributed fashion and store it to disk.

4.1 Distributed Scheduling Primitives

For distributed systems, we require a notation that allows us to express tensor blocking and index splitting. *Index splitting* describes the process in which the range representing the possible values for an index variable, i.e. $[0, D)$ for an index variable of dimension D, is split into P contiguous *split ranges* $[0, s_0), [s_0, s_1), \ldots, [s_{P-2}, D)$

for a given non-decreasing tuple of $P-1$ *split points* $(s_0, s_1, \ldots, s_{P-2})$. Note that we allow here for the possibility that $P = 1$, so that the split points tuple is empty. For a given split range, let its index within its parent list of split ranges be known as its *split index*.

As part of its distribution schedule, SuperTaco accepts a list of index variables to be split, and for each variable its corresponding list of split points. Instead of supplying the split points for an index variable, a user can also supply a single number representing the number of split ranges to make, and SuperTaco will uniformly split the index variable into the requested number of ranges. If an index variable is not asked to be split, SuperTaco will split it with a single range of $[0, D)$.

Now suppose we have a tensor $A_{i_0 i_1 \ldots i_{R-1}}$ of dimensions $(N_0, N_1, \ldots, N_{R-1})$, where each index variable i_m has been split into P_m split ranges using the split points $(s_0^{(m)}, s_1^{(m)}, \ldots, s_{P_m-2}^{(m)})$. The splitting of these index variables define a *tensor blocking* of A; that is, we can split the components of A into $P_0 \cdot P_1 \cdot \ldots \cdot P_{R-1}$ tensor blocks $A^{I_0 I_1 \ldots I_{R-1}}$ for non-negative integers I_m satisfying $I_m < P_m$, where the components of the tensor $A^{I_0 I_1 \ldots I_{R-1}}$ consist of the components of A that have i_m in I_m-th split range of mode m for all m. In this notation, $I_0, I_1, \ldots, I_{R-1}$ represent the *block index variables* of tensor block $A^{I_0 I_1 \ldots I_{R-1}}$. We use bars to distinguish the index variables of $A^{I_0 I_1 \ldots I_{R-1}}$ from the index variables of A, so that the components of $A^{I_0 I_1 \ldots I_{R-1}}$ are written as $A^{I_0 I_1 \ldots I_{R-1}}_{\bar{i}_0 \bar{i}_1 \ldots \bar{i}_{R-1}}$.

In this sense, the index variable i_m has been split into two index variables I_m and $\bar{i}_m$, in which $i_m = p_m + \bar{i}_m$ where $[p_m, p_{m+1})$ is the I_m-th split range of index variable i_m. Note that it appears at first glance that $A^{I_0 I_1 \ldots I_{R-1}}_{\bar{i}_0 \bar{i}_1 \ldots \bar{i}_{R-1}}$ could be considered a tensor with rank $2R$ and index variables $(I_0, I_1, \ldots, I_{R-1}, \bar{i}_0, \bar{i}_1, \ldots, \bar{i}_{R-1})$, but this loses important information about the relationship between I_m and $\bar{i}_m$. Specifically, the possible values of $\bar{i}_m$ are determined by I_m via the I_m-th split range. If we discard this relationship, then the range of $\bar{i}_m$ becomes the full range of i_m (i.e. $[0, N_m)$), which would defeat the purpose of blocking. It is for this reason we must consider the block index variables separately when splitting index variables.

4.2 Taco Tensor Storage Format Augmentation

To efficiently store tensor blocks, there is one minor modification we need to make to Taco's sparse tensor storage formats. The dense level type currently assumes that the corresponding index variable can take values in the range $[0, D)$ for dimension size D. However, when splitting the range $[0, D)$, many ranges are created that don't begin at 0. For example, consider splitting into D ranges, so that the last range is $[D-1, D)$. The tensor block corresponding to this range can only take one value for this index variable, namely $D-1$. However, if the dimension is stored dense, then the tensor block is forced to use dimension size D, which assumes the index variable can take on D values. This illustrates that the dense level type is inefficient for index variables with high minimum values.

To remedy this, we augment the dense level type to include an offset T, so that the corresponding index variable can take values in the range $[T, T+D)$. This formulation allows us to use the dense level type to store tensor blocks efficiently. Similar to how dimension sizes must match when computing with tensors that have the same index variable, dimension offsets must also match. The changes that this imposes on Taco code generation is fairly minimal. In particular, if there's an index variable in the generated loop nests that is stored sparse in some tensor and dense in some other tensor, then any value read from an `idx` array corresponding to that index variable must be decremented by the dense dimension's offset. Similarly, any value written to an `idx` array corresponding to that index variable must be incremented by the dense dimension's offset prior to being written. These changes effectively add support for dense offsets to Taco-generated `assemble()` and `compute()` code by taking advantage of the fact that sparse dimension `idx` values only matter relative to the other index variable values. (This isn't true of dense dimension index values, since at times the dense index value is used as a stand-in for the distance between the dense index value and the dense initial index value due to the initial index value being assumed as zero in the generated code.)

4.3 Sparse Tensor File Format

While the Taco library defines a taxonomy for sparse tensor storage formats as described in Section 2.2, it does not define a file format for storing the associated data structures. A common existing file format for storing sparse tensors is the `.tns` file format, which stores a tensor entry and its indices on each line. That is, if a tensor maps R-tuple $(i_0, i_1, \ldots, i_{R-1})$ to component c, then the `.tns` file has a corresponding line reading "$i_0\ i_1\ \ldots\ i_{R-1}\ c$". This file format is sparse in the sense that zero-component entries are omitted. While `.tns` files do not necessarily have to have their lines be sorted by indices, in this work we assume that `.tns` files are sorted.

One of the downsides of this format is that it uses plaintext to store numbers, which takes up more space for larger numbers compared to a binary format (a 64-bit integer takes up the same space as 8 plaintext characters). This is particularly bad for large tensors, since storage bandwidth can be limiting for tensor loading. Furthermore, if we have a desired sparse tensor storage format, then the tensor must be converted into this format from the `.tns` file. For these reasons, we have created a binary file format for storing sparse tensors that mirrors the data structures defined by Taco's sparse tensor storage formats, which we call `.tcs`.

A `.tcs` file consists of a header section and a data section. The header stores metadata about the tensor and its dimensions, while the data section stores the `pos` and `idx` arrays (as 64-bit integers) for sparse dimensions along with storing the `val` array (as double-precision floats). More specifically, the header stores the rank, the file size, the type of each dimension (dense or sparse), the byte offsets of the arrays in the data section, and the dimension offset and size for dense dimensions. For tensor blocks, the header additionally stores for each dimension its split range endpoints, that range's corresponding index in the list of split ranges for that index variable, and the number of split ranges for that index variable. The file format being binary means that accessing any `pos`, `idx`, and `val` arrays only requires copying the relevant file sections to buffer space; this is an improvement over the `.tns` file format, where the relevant data would have to be parsed from plaintext. Moreover, the byte layout defined by the `.tcs` file format can alternatively be used as a memory layout, in which the header section is a struct with a variable-length array at the end that stores per-dimension metadata, and the data section is a series of arrays. In this sense, the `.tcs` format is as much a memory format as it is a file format.

We generally expect users to convert `.tns` files to `.tcs` files only when they first receive their dataset, so we do not attempt to distribute this conversion across processors. However, this conversion must be capable of dealing with tensors that cannot fit into a single node's memory. We thus perform this conversion with a two-pass streaming algorithm that computes header information in the first pass and populates the arrays in the data section in the second pass.

The streaming conversion algorithm takes as input the `.tns` file and a type for each dimension (dense or sparse). Note that we assume the index variable order for the storage format is the same as the order of variables in the `.tns` file; if not, the appropriate columns in the `.tns` file must be swapped and the `.tns` file resorted. During the first pass, we stream through the lines of the `.tns` file, keeping track of of the maximum and minimum value for each dimension (for dense dimensions whose offset and size are unspecified by input arguments, these are used to determine the offset and size). As we stream, we also keep a level size counter for each dimension, which represents how many storage format tree nodes are in a particular level. The counters all start at 0, and for every line, we determine the first level for which there is a difference between this line and the previous line; suppose it is level k (if there is no previous line, then we say it is level 0, i.e. the root node level). We then increment the level size counter for all levels $\geq k$, as the storage format tree splits into a branch at level k. After going through all pairs of consecutive lines, we can use these level size counters and the dense dimension sizes to determine the sizes of the `pos`, `idx`, and `val` arrays, which in turn lets us compute their byte offsets along with the file size.

The length of any sparse level's `idx` array is equal to its level size counter. The length of any sparse level's `pos` array can be found by finding the closest sparse level that's higher than the original sparse level (i.e. closer to the tree root), taking the length of its idx array, multiplying that by the dimension sizes of any dense dimensions between the original sparse level and its closest higher sparse level, and then adding one (due to the sentinel placed at the end of pos arrays). If there is no sparse level higher than the original sparse level, then we use the root node and assume an idx array length of 1. We can also use this method to determine the length of the val array if we just consider it as an extra level $R + 1$ past the leaf nodes, but for the val array we don't add a 1 since there is no sentinel.

For the second-pass of the conversion, we similarly stream through the lines of the .tns file, this time filling the data section of the .tcs file. Note that we write to the pos, idx, and val arrays sequentially, so we can stream output as well with buffers. Once again we iterate through every line and compare to its previous one, finding the first level k for which there is a difference between this line and the previous one, and keeping track of level size through level size counters. However, when we're processing a given line, we do not update the level size counters immediately; instead, we use these level size counters to write the appropriate values to the arrays in the data section, and then update the level counters afterwards.

Writing the appropriate values to the arrays once again involves comparing the current line to the previous line. As we iterate through the levels of a line, we keep track of an integer displacement (initialized to 0 at the start of the line) which tells us how many copies of a value need to be pushed onto the pos array. Starting at level k, we iterate through the remaining levels and do the following for each level. If the level is dense, we update the displacement by multiplying it by the dimension size, adding the current line's index value for the level, and subtracting the previous line's index value for the level. If the level is sparse, we push the value of the level size counter onto the pos array a total number of times equal to the displacement, push the current line's index value for the level onto the idx array, and then reset the displacement to 1. When we reach the component value at the end of the line, we push displacement minus one instances of 0 onto the val array and then push the actual component value onto the val array. The displacement here effectively accounts for the fact that dense levels assume that nodes in the tree have a number of children equal to the dimension size, which causes a subsequent sparse level to accordingly have zero children for those extra nodes that appeared in the dense levels. This process is slightly different for the first line in the `.tns` file, where we actually initialize the displacement to 1 and use a sentinel for the previous line in which sparse levels have an index value of 0 and dense levels have an index value of $D - 1$ for dimension size D. Similarly, at the end of the streaming algorithm we have to process an additional sentinel line where the displacement is once again initialized to 1 and all of the sentinel line's index values are 0 (for the `val` array here we only push the zeros since there is no component value).

More generally, this streaming algorithm effectively allows any iterator that iterates through the entries of a tensor in sorted order to convert those entries to the `.tcs` format. This will become useful in the next section in which we discuss the distributed loading of a tensor into tensor blocks.

4.4 Distributed Tensor Loading

The first phase of a Legion program generated by SuperTaco is loading each tensor into memory and partitioning them into tensor blocks. Since the tensor may not be able to fit into a single node's memory, this needs to be done in a distributed fashion. Note that most of the code for this phase does not need to be generated dynamically; such routines are packaged into a static library that the dynamically-generated Legion code calls.

During the first step (Section 4.4.1), the tensor on disk is conceptually split into pieces via load balancing, in such a way that each of the pieces are around the same size and can be easily iterated over. If we were to list in-order all the entries in a tensor, a *tensor piece* would be a contiguous section of this list; in this sense, we could also define it by a contiguous section of the `val` array. Each processor uses binary search to look for a point in the tensor such that the size of the tensor piece spanning the start of the tensor to that point is a desired size offset, where the desired offset for each processor is determined by the parent task. Once the location in the tensor is found, it is passed back to the parent Legion task, which then uses that information to define the tensor pieces by their endpoints. Each processor is assigned a piece.

Next, each processor will stream through its tensor piece, and place each entry it sees in the corresponding tensor block it belongs to. Since each processor is only iterating through a piece of the tensor, each processor will only be assembling tensor block pieces. This process requires two passes, and is correspondingly divided into two parts. The first part (Section 4.4.2) is for each processor to iterate through its piece in a streaming fashion, and gather partial header information for each tensor block. This partial header information is sent back to the parent task, which allows it to assemble the full `.tcs` header for each tensor block.

For the second part (Section 4.4.3), the parent task computes certain state information that the processors need to make the second pass, in which it assembles the tensor block pieces. It sends that state information to each processor, and then the processors make their second pass through their tensor piece to assemble the tensor block pieces. For the final step (Section 4.4.4), each tensor block is assembled on its designated node by gathering its tensor block pieces from the other processors.

4.4.1 Load Balancing

Before we can discuss load balancing, we need a clear way of defining the endpoints of a tensor piece, along with its size. Recall from Section 2.2 that we can uniquely identify a particular point in the process of tensor iteration via effective position pointers for each of the levels, essentially creating a vector of positions. Thus, by specifying a starting position vector and ending position vector, we can specify a piece of the tensor that can easily be iterated over. To load balance, we want to find a set of position vectors that divides the tensor into pieces of roughly equal size.

We would like to define the *tensor piece size* as the number of bytes in the data structures that would be traversed while iterating from the starting position to the ending position, which is a good approximation of the bandwidth consumed from iterating through such a piece. To calculate the size of a tensor piece, it is simpler to think of it in terms of the difference between the size offsets of the starting and ending position vectors, where the size offset of a position vector is the size of the tensor piece between the start of the tensor and that position.

Calculating the size offset given a position vector is straightforward. Note that the effective positions in the position vector give locations in the `idx` arrays and the `val` array, but they also give the locations in the `pos` arrays, since the location in a `pos` array is just the effective position of the previous level. Thus, given a position vector, we can easily determine the distance between the locations in these arrays and the starts of the arrays. We then define the size offset as the sum of these distances, when converted to bytes, for each of the sparse levels and the val array. (We leave out the distances from the `pos` and `idx` arrays for dense levels since these are only conceptual; dense levels only really store a small amount of metadata in the header.)

With these definitions made clear, we can now discuss how generated Legion programs load balance tensor loading. Since the program knows the file size of the tensor, it can determine the ideal size offsets that would evenly divide the tensor into pieces. The next step then for each size offset is to find a position vector that approximately achieves that size offset, which can be done through binary search. Our Legion program parallelizes this by launching a binary search task for each size offset, with the number of size offsets equal to the number of processors in the distributed system.

For the binary search task executed on each processor, the first step is to take a guess as to the effective position in the `val` array that achieves the desired size offset (recall from Section 2.1 that the effective position in the `val` array uniquely defines the other effective positions). To compute the size offset for the guess, we then need to determine the effective positions for all the levels, which we can do by going up the levels of the tree to the root. Recall that the location in an `idx` array is the same as the location in the `pos` or `val` array in the level below it, so determining the location in an `idx` array from lower levels is straightforward. The tricky part is determining the location in the `pos` array from the location in the same level's `idx` array. If the level is dense, this computation is just a straightforward floor division by the dimension size. However, if it is sparse, this requires a binary search to find the appropriate location in the `pos` array. Using this method, we can determine the entire position vector for our guessed effective position into the `val` array, and determine the actual size offset. We can then binary search the space of size offsets by querying more `val` array effective positions to eventually find the position vector near the desired size offset.

This process of finding the position vector that approximates the desired size offset can be sped up by using heuristic versions of binary search. In our case, we implemented binary search with a guess, in which the search accepts an initial guess and expands the search window exponentially around the guess until the desired point is in the bounds of the window, at which the point the window shrinks around the desired point exponentially as it would in traditional binary search. We specifically use a proportional guess based on size information, which has been sufficient for the tensor datasets we've used thus far.

Once a binary search task completes, it writes its resulting position vector into a logical region to share its output with its parent task, and then the parent task accordingly waits on all binary search tasks to finish before assembling the position start points and end points for each tensor piece. (The logical regions here are "flat", i.e. they have a 1D, contiguous index space and a field space with a single byte-wide column, which makes them suitable for raw data transfer. This is to optimize access performance, and all logical regions in this work should be assumed flat unless stated otherwise.)

4.4.2 Tensor Block Header Assembly

Now that we've created tensor pieces that are approximately equal in size, each processor in the system can iterate through its piece to find partial header information about the tensor blocks. Note that since each processor is only iterating through a piece of the tensor, it's also only iterating through a piece of each tensor block, and can accordingly only find partial information about the tensor. The information must be combined in the parent task to get a complete header for the tensor block.

Iterating through a particular piece of a tensor is relatively straightforward, and can be thought of as a traversal of the sparse format tree. Because we still move sequentially through the `pos`, `idx`, and `val` arrays, we can again stream through these arrays using buffers. We use a position vector to keep track of our locations in the tensor level data structures, and initialize to the starting position. We also use an index vector to keep track of the current index value for each level, and for sparse levels, we keep track of the right position endpoint from `pos`. For the dense dimensions, we initialize the index value to the sum of the effective position modulo the dimension size and the dimension offset. Iteration is performed using a while loop that keeps track of tree level and whether traversal is moving down or up; initially the tree level is 1 and traversal is moving down. The while loop only exits if the tree level reaches 0, or if the current position vector's last element (i.e. the effective position into the `val` array) becomes at least as large as the ending position vector's last element. For tree levels between 1 and R, the behavior of the loop differs depending on whether the level is dense or sparse. If the level is sparse and traversal is moving down, then we fetch the next right position endpoint from the `pos` array and determine whether the level's effective position is less than it (we also update the right position endpoint vector accordingly). If it is, then there are still children left in the corresponding segment, so we fetch the index value from `idx` corresponding to the effective position, increment the tree level, and keep moving down. If it's not, then there are no children left in the segment, in which case we decrement the tree level and move up. If instead we enter a sparse level while moving up, we increment the effective position and once again check whether our effective position is less than the right position endpoint, doing the same as before. If the level is dense and traversal is moving down, we need only increment the tree level and keep moving down. If instead we enter a dense level while moving up, we increment both the effective position and the index value, and check whether the index value is less than the sum of the dimension offset and the dimension size. If so, the segment still has children, so we increment the tree level and move down. If not, then the segment is finished, so we reset the index value to the dimension offset, decrement the tree level, and keep moving up. If our tree level becomes $R + 1$, then we've reached the `val` array in the tree, so we read the appropriate component from `val` using the effective position of the previous level. At this point, we have a full tensor entry from our index value vector and component, and can use a callback to decide how to handle the entry. After this, we begin moving back up and decrement the tree level.

We can augment this iteration process to further supply our callback with the split index for each dimension. That is, we can keep track of which split range each index value falls within. To do so, when an index value is increased from a position bounds check succeeding, we keep incrementing the split index until the new index value is within the bounds of the split index's range. If at any point we move down into a sparse or dense level (and a bounds check succeeds, if it exists), we reset the split index to 0 and again increment the split index until the new index value is within the bounds of the split index's range.

Using this procedure, we can iterate through the entries of a tensor piece and assign each one to its corresponding tensor block in the callback, as determined by the split index for each dimension of the entry. As for what information is gathered by the callback, it is similar to the first-pass of `.tns` to `.tcs` conversion; for each block, we keep vectors that store the level size, the minimum and maximum values, and additionally we keep copies of the first and last lines encountered for that block so we can account for edge effects when we merge this information together with that of other processors.

In the Legion program, we parallelize this procedure similar to how the binary search was parallelized. The parent task launches a block header child task for every tensor piece (as defined by its start and end position vectors), and the parent task receives from each processor the information they gathered while streaming via logical regions. For each tensor block, the parent task then reconstructs the tensor block's `.tcs` header based on the information it received from each processor about that block. (This primarily consists of pooling minima/maxima, adding together the level size vectors, and accounting for edge effects.) This essentially parallelizes the first-pass of `.tcs` conversion.

4.4.3 Tensor Block Piece Assembly

Now that the tensor block headers have been computed, the next step is for each processor to make a second-pass through its assigned tensor piece, and assemble the tensor block pieces for each of its tensor blocks. Similar to the previous section, this effectively uses the augmented iteration code to iterate through the entries of a tensor piece while determining which block each entry belongs to, and the callback resembles the second-pass of `.tcs` conversion to form the tensor block pieces. However, while the first-pass of conversion was easily parallelized, the second-pass requires more work.

In particular, during the second-pass of conversion the level size counters are built up over the course of iterating through the entire tensor, and during the processing of each line those counters are used when writing to the `pos` array. Since each processor only iterates through a piece of the tensor for loading, the level size counters aren't built up appropriately, which would make the values written to the `pos` arrays incorrect. Each processor instead needs to start with the level size counters that would have resulted from iterating from the start of the tensor to the start of that processor's assigned tensor piece. The parent task can thankfully provide this by computing partial sums over the level size counters it collected from each processor (and accounting for edge effects). The parent task additionally provides information to each processor about the size of the arrays for each of its tensor block pieces.

Hence, using this approach we are able to parallelize the second-pass of conversion, and generate the tensor block pieces on each processor. In the Legion program, we again parallelize this procedure similar to the binary search task and tensor header block computation, except this time the parent task doesn't access the output of the task, i.e. the tensor block pieces. The parent task does, however, have references to the logical regions holding that data, so that it may pass those logical regions to other child tasks.

4.4.4 Tensor Block Assembly

The final step of tensor loading is assembling the tensor blocks from their tensor block pieces. For each tensor block, this requires loading the tensor block header from the parent task, and loading the tensor block pieces from the other processors. The `pos`, `idx`, and `val` arrays within the tensor block pieces are simply concatenated to form the data section of the tensor block.

In the Legion program, parallelization is different than the previous steps. Instead of creating a child task for every processor, we create a child task for every block. The parent passes the appropriate tensor block header through a logical region when creating the child task. The parent task also passes logical regions containing the tensor block pieces to the child task assembling that tensor block. This means that the child task will not run until the tasks that generate the tensor block pieces have finished. Furthermore, Legion will take care of copying those tensor block pieces to the processor that will assemble that tensor block, so that when the child task runs, the pieces will be in its memory. After the child task has finished assembling the tensor block, it will destroy the logical regions containing the tensor block pieces, allowing them to be garbage collected.

Through these steps, we are able to assemble the tensor blocks in a distributed fashion. The original tensor is divided into pieces of equal size, which means the sum of the sizes of the tensor block pieces on any given node should be roughly the same in size, provided the tensor and index variable split ranges are not significantly skewed. To this extent, this tensor loading scheme is load-balanced, and should be able to handle tensors that can't fit into a node's memory. One aspect that has not yet been discussed is which nodes end up receiving which tensor blocks. This is affected by the block's *mapping key*, and will be explained in the next section on distributing the computation of tensor algebra.

4.5 Distributed Tensor Algebra Computation

In the previous section, we discussed how our generated Legion programs load tensors from disk into tensor blocks. The next phase of our Legion program is to compute the desired tensor algebra expression in a distributed fashion using these tensor blocks. This consists of two major steps.

The first step is the execution of the kernel itself. This is done in the top-level parent task through a nest of for loops, each of which iterates through a block index variable, and a child task launch in the innermost for loop body that executes the kernel. In this innermost loop body, the parent task adds region requirements to the kernel task launch for logical regions containing the tensor blocks that the kernel task must operate on, which is determined by the value of the block index variables in the for loop nest. The kernel task returns its output block logical region to the parent through a future, which the parent may then add to the next kernel task that sums into the same output block. A second step, which may or may not be necessary depending on the distribution schedule, is to sum together any output blocks that were treated as different during the first step, but in reality referred to the same output block. This is done in a distributed fashion by launching pair-wise summation tasks to form a tree-like dependency graph between tasks, with the final output block being returned from the summation task at the root of the tree.

To understand how this code is generated requires an understanding of how the computation of the tensor algebra expression is changed theoretically when blocking tensors, which is explained in Section 4.5.1. With this insight in hand, Section 4.5.2 talks about how those transformed expressions are implemented in Legion code, and accordingly how that code is generated from distribution schedules. Section 4.5.3 describes the code generation method for creating the kernel task, while section 4.5.4 describes the code generation method for creating the summation task. Section 4.5.5 talks about how tasks and data are mapped to particular nodes in Legion, and how this is controlled in code generation through the distribution schedule. Finally section 4.5.6 describes an issue posed by Legion's garbage collection to keeping memory use bounded, and how this is solved in code generation by emitting tensor block eviction code in the for loop nest.

4.5.1 Expression Transformation

As mentioned previously, we divide the computation of sparse tensor algebra expressions via tensor blocking and index splitting. In order to know what Legion code to generate for this distributed computation, we need to understand how blocking affects the computation of this expression.

Consider a simple example kernel such as SpMV, which is typically written as $a_i = \sum_j B_{ij} c_j$. To more explicitly represent the dependence on index variable i, we use concrete index notation [13] to write this as $\forall_i \left(a_i = \sum_j B_{ij} c_j \right)$. This says that we evaluate the expression surrounded by the $\forall$ for each i. Now consider row blocking, in which we split the index variable i into I and $\bar{i}$. This is effectively splitting the for loop $\forall_i$ into two for loops $\forall_{I=0}^{P_i-1} \forall_{\bar{i}=p_I}^{p_{I+1}-1}$, where P_i is the number of split ranges and the I-th split range is given by $[p_I, p_{I+1})$.

Note here that since we're not blocking columns, we're effectively splitting index variable j into one split range $[0, N_j)$. The $\sum_j$ term above similarly becomes $\sum_{J=0}^{P_j-1} \sum_{\bar{j}=p_J}^{p_{J+1}-1}$, but in this case the $\sum_{J=0}^{P_j-1}$ term becomes a no-op (since $P_j = 1$), and the $\sum_{\bar{j}=p_J}^{p_{J+1}-1}$ term becomes $\sum_{\bar{j}=0}^{N_j-1} = \sum_j$ (since the only split range is $[0, N_j)$). This means that for a non-split index variable like j, we can just treat it as a split index variable but with $J = 0$ and $\bar{j} = j$.

Splitting the for loops and writing our tensors using tensor block notation gives us our new expression after index splitting as

$$\forall_I \forall_{\bar{i}} \left(a_{\bar{i}}^I = \sum_j B_{\bar{i}j}^{I0} c_j^0 \right)$$

If we then parenthesize our expression as

$$\forall_I \left[\forall_{\bar{i}} \left(a_{\bar{i}}^I = \sum_j B_{\bar{i}j}^{I0} c_j^0 \right) \right]$$

we can interpret the outer for loop $\forall_I$ as a loop over processors, and the inner expression as what each processor needs to compute, i.e. the SpMV kernel computed on tensor blocks.

Now consider the case of column splitting. Our expression similarly becomes

$$\forall_i \left(a^0_{\bar{i}} = \textstyle\sum_J \sum_{\bar{j}} B^{0J}_{\bar{i}\bar{j}} c^J_{\bar{j}} \right)$$

We would like to hoist the $\sum_J$ term out of the expression into a $\forall_J$ loop so that we can interpret it as a for loop over processors, similar to how we did for $\forall_I$. However, doing so naively would create multiple copies of each $a^0_{\bar{i}}$, one on each processor, and each would only be a partial sum. To arrive at the actual value for $a^0_{\bar{i}}$, we need to sum together the partial values. We can express this transformed computation as

$$\forall_J \left[\forall_i \left(\bar{a}^{0J}_{\bar{i}} = \textstyle\sum_{\bar{j}} B^{0J}_{\bar{i}\bar{j}} c^J_{\bar{j}} \right) \right] ; \forall_i \left(a^0_{\bar{i}} = \textstyle\sum_J \bar{a}^{0J}_{\bar{i}} \right)$$

where we have created intermediate vectors, namely $\bar{a}^{0J}$, to hold the partial sums. More generally, if we have an expression of the form

$$\forall_{I_0} \forall_{I_1} \dots \forall_{I_{R-1}} \left[\forall_{\bar{i}_0} \forall_{\bar{i}_1} \dots \forall_{\bar{i}_{R-1}} \left(A^{I_0 I_1 \dots I_{R-1}}_{\bar{i}_0 \bar{i}_1 \dots \bar{i}_{R-1}} = \textstyle\sum_{J_0} \sum_{J_1} \dots \sum_{J_{R'-1}} (expr) \right) \right]$$

for free variables i_q and reduction variables j_q, we can hoist the block index variables J_q for those reduction variables out of the expression by creating intermediary tensors $\bar{A}^{I_0 I_1 \dots I_{R-1} J_0 J_1 \dots J_{R'-1}}$ according to

$$\begin{gathered}\forall_{I_0} \forall_{I_1} \dots \forall_{I_{R-1}} \forall_{J_0} \forall_{J_1} \dots \forall_{J_{R'-1}} \left[\forall_{\bar{i}_0} \forall_{\bar{i}_1} \dots \forall_{\bar{i}_{R-1}} \left(\bar{A}^{I_0 I_1 \dots I_{R-1} J_0 J_1 \dots J_{R'-1}}_{\bar{i}_0 \bar{i}_1 \dots \bar{i}_{R-1}} = expr \right) \right] ; \\ \forall_{I_0} \forall_{I_1} \dots \forall_{I_{R-1}} \left[\forall_{\bar{i}_0} \forall_{\bar{i}_1} \dots \forall_{\bar{i}_{R-1}} \left(A^{I_0 I_1 \dots I_{R-1}}_{\bar{i}_0 \bar{i}_1 \dots \bar{i}_{R-1}} = \textstyle\sum_{J_0} \sum_{J_1} \dots \sum_{J_{R'-1}} \bar{A}^{I_0 I_1 \dots I_{R-1} J_0 J_1 \dots J_{R'-1}}_{\bar{i}_0 \bar{i}_1 \dots \bar{i}_{R-1}} \right) \right]\end{gathered}$$

Going back to our SpMV example, if we split along both rows and columns, we would arrive at

$$\forall_I \forall_J \left[\forall_{\bar{i}} \left(\bar{a}^{IJ}_{\bar{i}} = \textstyle\sum_{\bar{j}} B^{IJ}_{\bar{i}\bar{j}} c^J_{\bar{j}} \right) \right] ; \forall_I \left[\forall_{\bar{i}} \left(a^I_{\bar{i}} = \textstyle\sum_J \bar{a}^{IJ}_{\bar{i}} \right) \right]$$

Splitting along j in this way allows us to further parallelize the computation of the expression at the price of having to combine the partial sums later. But there is another hidden cost; namely, the distributed system needs enough space to store the intermediary tensors. The tensor blocks $A^{I_0 I_1 \dots I_{R-1}}$ are just a partition of A, so the amount of space they take up should be only some small factor larger than A on average. However, the intermediary tensors $\bar{A}^{I_0 I_1 \dots I_{R-1} J_0 J_1 \dots J_{R'-1}}$ are not a partition, and may accordingly exceed the total space of tensor blocks $A^{I_0 I_1 \dots I_{R-1}}$ by a factor of $P_{J_0} \cdot P_{J_1} \cdot \dots \cdot P_{J_{R'-1}}$. Depending on how the indices J_q are split, the amount of space needed by the intermediary tensors may be too large.

Luckily, there is another way to restructure the computation that allows us to hoist the $\sum_J$ terms without increasingly many intermediaries. Returning to our SpMV example, we can instead transform

$$\forall_i \left(a_i^0 = \textstyle\sum_J \sum_{\bar{j}} B_{i\bar{j}}^{0J} c_{\bar{j}}^J \right)$$

by summing directly into the output, i.e.

$$\forall_i \left(a_i^0 = 0 \right); \forall_J \left[\forall_i \left(\bar{a}_i^0 = \textstyle\sum_{\bar{j}} B_{i\bar{j}}^{0J} c_{\bar{j}}^J \right); \forall_i \left(a_i^0 \mathrel{+}= \bar{a}_i^0 \right) \right]$$

where $\bar{a}_i^0$ is a temporary tensor (this tensor is the same size as a_i^0, and can be deleted once it is summed into a_i^0). Note that while this transformation does split the computation into P_J steps consisting of $\forall_i \left(a_i^0 \mathrel{+}= \sum_{\bar{j}} B_{i\bar{j}}^{0J} c_{\bar{j}}^J \right)$, these steps cannot be computed in parallel across J since each step writes to a_i^0. However, the act of splitting along j still decreases the size of tensor blocks depending on j, and thus this transformation is still useful in that it decreases the amount of memory needed by a processor to compute a step. Going back to our SpMV example, if we split along both rows and columns with this new hoisting method, we would arrive at

$$\forall_I \left[\forall_{\bar{i}} \left(a_{\bar{i}}^I = 0 \right) \right]; \forall_I \forall_J \left[\forall_{\bar{i}} \left(\bar{a}_{\bar{i}}^I = \textstyle\sum_{\bar{j}} B_{\bar{i}\bar{j}}^{IJ} c_{\bar{j}}^J \right); \forall_{\bar{i}} \left(a_{\bar{i}}^I \mathrel{+}= \bar{a}_{\bar{i}}^I \right) \right]$$

Hence, we have two methods to hoist a block reduction variable out of a tensor algebra expression. The first way affords parallelism at the cost of increased space from intermediary tensors and some increased time due to summing the intermediary tensors, while the second way doesn't afford parallelism but still decreases the size of the tensor blocks that use the index variable and thus decreases the memory needed by the processor to compute a step. These methods can be mixed and matched on reduction variables as well; consider an expression of the form

$$\begin{aligned}&\forall_{I_0}\forall_{I_1}\ldots\forall_{I_{R-1}}\Big[\\&\forall_{\bar{i}_0}\forall_{\bar{i}_1}\ldots\forall_{\bar{i}_{R-1}}\Big(A^{I_0I_1\ldots I_{R-1}}_{\bar{i}_0\bar{i}_1\ldots\bar{i}_{R-1}} = \textstyle\sum_{J_0}\sum_{J_1}\ldots\sum_{J_{R'-1}}\sum_{J'_0}\sum_{J'_1}\ldots\sum_{J'_{R''-1}}(expr)\Big)\Big]\end{aligned}$$

for free variables i_q and reduction variables j_q and j'_q. If we use parallel hoisting on J_q and non-parallel hoisting on J'_q, we arrive at

$$\begin{aligned}&\forall_{I_0}\forall_{I_1}\ldots\forall_{I_{R-1}}\forall_{J_0}\forall_{J_1}\ldots\forall_{J_{R'-1}}\left[\forall_{\bar{i}_0}\forall_{\bar{i}_1}\ldots\forall_{\bar{i}_{R-1}}\left(\bar{A}^{I_0I_1\ldots I_{R-1}J_0J_1\ldots J_{R'-1}}_{\bar{i}_0\bar{i}_1\ldots\bar{i}_{R-1}} = 0\right)\right];\\&\forall_{I_0}\forall_{I_1}\ldots\forall_{I_{R-1}}\forall_{J_0}\forall_{J_1}\ldots\forall_{J_{R'-1}}\forall_{J'_0}\forall_{J'_1}\ldots\forall_{J'_{R''-1}}\Big[\\&\forall_{\bar{i}_0}\forall_{\bar{i}_1}\ldots\forall_{\bar{i}_{R-1}}\Big(\bar{\bar{A}}^{I_0I_1\ldots I_{R-1}J_0J_1\ldots J_{R'-1}}_{\bar{i}_0\bar{i}_1\ldots\bar{i}_{R-1}} = expr\Big);\\&\forall_{\bar{i}_0}\forall_{\bar{i}_1}\ldots\forall_{\bar{i}_{R-1}}\left(\bar{A}^{I_0I_1\ldots I_{R-1}J_0J_1\ldots J_{R'-1}}_{\bar{i}_0\bar{i}_1\ldots\bar{i}_{R-1}} \mathrel{+}= \bar{\bar{A}}^{I_0I_1\ldots I_{R-1}J_0J_1\ldots J_{R'-1}}_{\bar{i}_0\bar{i}_1\ldots\bar{i}_{R-1}}\right)\Big];\\&\forall_{I_0}\forall_{I_1}\ldots\forall_{I_{R-1}}\left[\forall_{\bar{i}_0}\forall_{\bar{i}_1}\ldots\forall_{\bar{i}_{R-1}}\left(A^{I_0I_1\ldots I_{R-1}}_{\bar{i}_0\bar{i}_1\ldots\bar{i}_{R-1}} = \textstyle\sum_{J_0}\sum_{J_1}\ldots\sum_{J_{R'-1}}\bar{A}^{I_0I_1\ldots I_{R-1}J_0J_1\ldots J_{R'-1}}_{\bar{i}_0\bar{i}_1\ldots\bar{i}_{R-1}}\right)\right]\end{aligned}$$

Looking back at the SpMV example, regardless of how we hoist variables or what variables are split, the underlying kernel computed by each processor is still another SpMV, just using the tensor blocks instead of the full tensors. A similar observation holds true for the more general case shown above. However, we have been making an assumption about our tensor algebra expressions so far, specifically that the reduction variable summation can always be hoisted to the outside of the expression on the right-hand side. In some expressions, there are reduction variables that cannot be trivially hoisted; for example, consider $\forall_i\left(a_i = d_i + \sum_j B_{ij}c_j\right)$. Suppose we split along j to get $\forall_i\left(a^0_{\bar{i}} = d^0_{\bar{i}} + \sum_J\sum_{\bar{j}} B^{0J}_{\bar{i}\bar{j}}c^J_{\bar{j}}\right)$; the right-hand side is not equivalent to $\sum_J\left(d^0_{\bar{i}} + \sum_{\bar{j}} B^{0J}_{\bar{i}\bar{j}}c^J_{\bar{j}}\right)$. Hoisting $\sum_J$ doesn't work here because addition doesn't distribute over addition. One potential solution to this is to use a ternary conditional operator `?:` to ensure the addition only occurs once, i.e.

$\forall_i \left(a_i^0 = \sum_J \left(((J == 0)\,?\,d_i^0 : 0) + \sum_{\bar{j}} B_{i\bar{j}}^{0J} c_{\bar{j}}^J\right)\right)$. However, Taco doesn't currently support ternary conditional operators, and trying to modify the outputted C code to support it would be non-trivial. A simpler solution is to have one processor (e.g. the one corresponding to $J = 0$) compute $\forall_i \left(\bar{a}_i^{0J} = d_i^0 + \sum_{\bar{j}} B_{i\bar{j}}^{0J} c_{\bar{j}}^J\right)$ while the other processors compute $\forall_i \left(\bar{a}_i^{0J} = \sum_{\bar{j}} B_{i\bar{j}}^{0J} c_{\bar{j}}^J\right)$. However, this requires multiple calls to the Taco command-line tool, one for each kernel. Moreover, for each index variable that encounters this problem while being hoisted, the number of kernels needed doubles, so the compile time varies exponentially with the number of such index variables. We leave implementing support for this kind of reduction variable hoisting to future work, and assume that if a block index variable is hoisted over addition, that the variable has only one split range (in this case, no changes are needed since the summation is a no-op). Hence, we move forward under the assumption that the main kernel computed by each processor is the same as the original expression to be distributed; the only difference is that the per-processor kernel operates on tensor blocks.

This theory thus allows us to distribute our tensor algebra expression computation into executions of the same kernel but on tensor blocks, and is done by hoisting any split index variables out of the expression in the ways we have described. Note that even for index variables j that aren't split, we can treat them as split variables that only having one split range. This allows us to use the same transformations as before to hoist them out (some of the summations may be no-ops, but this doesn't affect correctness). This means we can consider all index variables to be hoisted. That is, we can use the convention that the $\forall$ loops surrounding the kernel iterate through every block index variable.

Using this convention, we have exposed two choices in the general-case expression. The first is how to order these $\forall$ loops, since they can be rearranged in any order; in SuperTaco's distribution schedule, this is referred to as the *index variable ordering* and must be a permutation of all index variables in the expression. The second choice is, for each reduction index variable, whether it is parallel hoisted or non-parallel hoisted. SuperTaco will by default assume reduction variables are non-parallel hoisted since it doesn't impose an extra space requirement for intermediate tensors like parallel hoisting does. For SuperTaco to parallel hoist a variable instead, the user must declare it as an *output-extension variable* of the distribution schedule. This is named as such since the intermediate tensors $\bar{A}^{I_0 I_1 \ldots I_{R-1} J_0 J_1 \ldots J_{R'-1}}_{\bar{i}_0 \bar{i}_1 \ldots \bar{i}_{R-1}}$ in parallel hoisting are similar to the output tensor $A^{I_0 I_1 \ldots I_{R-1}}_{\bar{i}_0 \bar{i}_1 \ldots \bar{i}_{R-1}}$, but extended along the dimensions $J_0, J_1, \ldots, J_{R'-1}$. With this, we have enough information to discuss SuperTaco's code generation for tensor algebra computation.

4.5.2 Code Generation

Recall from the previous section that after allowing for index variables to be split, the general-case transformed expression for describing the distributed computation of tensor algebra is

$$
\begin{aligned}
&\forall_{I_0}\forall_{I_1}\ldots\forall_{I_{R-1}}\forall_{J_0}\forall_{J_1}\ldots\forall_{J_{R'-1}}\left[\forall_{\bar{i}_0}\forall_{\bar{i}_1}\ldots\forall_{\bar{i}_{R-1}}\left(\bar{A}^{I_0 I_1 \ldots I_{R-1} J_0 J_1 \ldots J_{R'-1}}_{\bar{i}_0 \bar{i}_1 \ldots \bar{i}_{R-1}} = 0\right)\right];\\
&\forall_{I_0}\forall_{I_1}\ldots\forall_{I_{R-1}}\forall_{J_0}\forall_{J_1}\ldots\forall_{J_{R'-1}}\forall_{J'_0}\forall_{J'_1}\ldots\forall_{J'_{R''-1}}\Big[\\
&\forall_{\bar{i}_0}\forall_{\bar{i}_1}\ldots\forall_{\bar{i}_{R-1}}\left(\bar{\bar{A}}^{I_0 I_1 \ldots I_{R-1} J_0 J_1 \ldots J_{R'-1}}_{\bar{i}_0 \bar{i}_1 \ldots \bar{i}_{R-1}} = \mathit{expr}\right);\\
&\forall_{\bar{i}_0}\forall_{\bar{i}_1}\ldots\forall_{\bar{i}_{R-1}}\left(\bar{A}^{I_0 I_1 \ldots I_{R-1} J_0 J_1 \ldots J_{R'-1}}_{\bar{i}_0 \bar{i}_1 \ldots \bar{i}_{R-1}} \mathrel{+}= \bar{\bar{A}}^{I_0 I_1 \ldots I_{R-1} J_0 J_1 \ldots J_{R'-1}}_{\bar{i}_0 \bar{i}_1 \ldots \bar{i}_{R-1}}\right)\Big];\\
&\forall_{I_0}\forall_{I_1}\ldots\forall_{I_{R-1}}\left[\forall_{\bar{i}_0}\forall_{\bar{i}_1}\ldots\forall_{\bar{i}_{R-1}}\left(A^{I_0 I_1 \ldots I_{R-1}}_{\bar{i}_0 \bar{i}_1 \ldots \bar{i}_{R-1}} = \textstyle\sum_{J_0}\sum_{J_1}\ldots\sum_{J_{R'-1}}\bar{A}^{I_0 I_1 \ldots I_{R-1} J_0 J_1 \ldots J_{R'-1}}_{\bar{i}_0 \bar{i}_1 \ldots \bar{i}_{R-1}}\right)\right]
\end{aligned}
$$

In this section, we discuss how SuperTaco generates code to compute this transformed expression. From section 4.5.1, the expression above is not fully-defined; it requires the user to make a choice regarding how to order the $\forall$ loops (through the distribution schedule's *index variable ordering*) and a choice regarding which reduction variables get parallel-hoisted (through the distribution schedule's list of *output-extension variables*). The user must also decide how to split the index variables (if it all), which is also specified through the distribution schedule as described in Section 4.1. Given a distribution schedule with these properties, the transformed expression above becomes fully-defined, and SuperTaco can accordingly generate code for it.

The transformed expression here consists of three steps. The first one is just an initialization step, in which we initialize all intermediate tensors $\bar{A}^{I_0 I_1 \ldots I_{R-1} J_0 J_1 \ldots J_{R'-1}}_{\bar{i}_0 \bar{i}_1 \ldots \bar{i}_{R-1}}$

to 0. The second step is the kernel computation step, in which the kernel is computed for every appropriate combination of tensor blocks, and the results summed into the output intermediate tensor. More specifically, the kernel is computed for every integer point $(I_0, I_1, \ldots, I_{R-1}, J_0, J_1, \ldots, J_{R'-1}, J'_0, J'_1, \ldots, J'_{R''-1})$ in the Cartesian grid spanning from the origin to $(P_{i_0} - 1, P_{i_1} - 1, \ldots, P_{i_{R-1}} - 1, P_{j_0} - 1, P_{j_1} - 1, \ldots, P_{j_{R'-1}} - 1, P_{j'_0} - 1, P_{j'_1} - 1, \ldots, P_{j'_{R''-1}} - 1)$. At a given integer point, the specific tensor blocks used in the execution of a kernel are based off the values of the block index variables in the point's coordinates. The order of the $\forall$ loops determines the order in which these points are traversed in the grid. Finally in the third step, the intermediate tensors $\bar{A}^{I_0 I_1 \ldots I_{R-1} J_0 J_1 \ldots J_{R'-1}}_{\bar{i}_0 \bar{i}_1 \ldots \bar{i}_{R-1}}$ are summed together into the actual output tensor $A^{I_0 I_1 \ldots I_{R-1}}_{\bar{i}_0 \bar{i}_1 \ldots \bar{i}_{R-1}}$.

It is useful here to define a *mapping key function* $\sigma()$ on any tuple of block index variables $(I_0, I_1, \ldots, I_{R-1})$, such that $\sigma(I_0, I_1, \ldots, I_{R-1}) = I_{R-1} + P_{i_{R-1}} \cdot (I_{R-2} + P_{i_{R-2}} \cdot \ldots \cdot (I_1 + P_{i_1} \cdot (I_0)) \ldots)$. That is, the mapping key function generates from the tuple $(I_0, I_1, \ldots, I_{R-1})$ a unique index into an array of size $P_{i_0} \cdot P_{i_1} \cdot \ldots \cdot P_{i_{R-1}}$; we call this the *mapping key size*. If we have some total ordering to the block index variables, we can also define $\sigma()$ on sets of index variables instead of tuples, in which case it is equal to $\sigma()$ applied the tuple formed from sorting the set by the total ordering. Since our distribution schedule provides a index variable ordering, we shall use this ordering for our mapping key function. Another useful function is the inverse mapping key $\sigma^{-1}()$ function, which computes a tuple $(I_0, I_1, \ldots, I_{R-1})$ from a positive integer k through repeated division, i.e. $k = q_{R-1} \cdot P_{i_{R-1}} + r_{R-1}$, $q_{R-1} = q_{R-2} \cdot P_{i_{R-2}} + r_{R-2}$, $\ldots$, $q_1 = q_0 \cdot P_{i_0} + r_0$ for quotients q_m and remainders r_m. The tuple generated by $\sigma^{-1}()$ is then just $(r_0, r_1, \ldots, r_{R-1})$.

Before SuperTaco begins generating code, it does some preprocessing involving the distribution schedule. It determines the set of free variables $\{I_0, I_1, \ldots, I_{R-1}\}$ by taking the variables from the output tensor, it determines the set of parallel hoisted reduction variables $\{J_0, J_1, \ldots, J_{R'-1}\}$ by taking the variables from the list of output-extension variables, and it determines the set of non-parallel hoisted reduction variables $\{J'_0, J'_1, \ldots, J'_{R''-1}\}$ by taking the set of all index variables in the expression, and subtracting out the set of free variables and the set of output-extension variables. It then generates C/C++ functions that compute the mapping keys for each of these sets of variables (using the ordering given in the distribution schedule's index variable ordering), along with the mapping key sizes. To simplify notation, let the mapping key function for $\{I_0, I_1, \dots, I_{R-1}\}$ be σ_I and the mapping key size be λ_I, and similarly define σ_J, λ_J, $\sigma_{J'}$, and $\lambda_{J'}$. For each tensor, SuperTaco also calculates the mapping key and mapping key sizes of its set of variables. To simplify notation, let the mapping key function for the variables of tensor A be σ_A and the mapping key size be λ_A.

For the first step of the transformed expression, we generate code for

$$\forall_{I_0}\forall_{I_1}\dots\forall_{I_{R-1}}\forall_{J_0}\forall_{J_1}\dots\forall_{J_{R'-1}}\left[\forall_{\bar{i}_0}\forall_{\bar{i}_1}\dots\forall_{\bar{i}_{R-1}}\left(\bar{A}^{I_0I_1\dots I_{R-1}J_0J_1\dots J_{R'-1}}_{\bar{i}_0\bar{i}_1\dots\bar{i}_{R-1}}=0\right)\right]$$

However, instead of actually initializing an intermediate `.tcs` tensor to 0 and storing it inside a logical region, we will place a small sentinel value in the logical region instead. Later when the logical region is passed to a kernel task, the kernel task will check whether the sentinel is there. If it is, the kernel task will accordingly skip adding its output tensor block to a zero tensor, and just return its output tensor block to the parent task.

For this step's code generation, we initialize an array `intermediate_fts` of futures, with length equal to $\lambda_I \cdot \lambda_J$. This array will store the futures containing the logical regions that store the intermediate tensors. We then emit two nested for loops. The first for loop iterates through the mapping keys k_J for $\{J_0, J_1, \dots, J_{R-1}\}$, i.e. from 0 to $\lambda_J - 1$. The second for loop iterates through the mapping keys k_I for $\{I_0, I_1, \dots, I_{R-1}\}$, i.e. from 0 to $\lambda_I - 1$. In the innermost loop body, we create a logical region and store a sentinel in it (via an inline map), and pack it inside a future that is set to resolve immediately if waited on. We then compute the key into `intermediate_fts` as $k_I + \lambda_I \cdot k_J$, and then store the future to this location. Note that the overall loop order here doesn't matter since the intermediate tensors are all different, which is why the effective loop ordering used here isn't sorted according to the distribution schedule's index variable ordering.

For the second step of the transformed expression, we generate code for kernel computation, i.e.

$$\begin{aligned}
&\forall_{I_0}\forall_{I_1}\dots\forall_{I_{R-1}}\forall_{J_0}\forall_{J_1}\dots\forall_{J_{R'-1}}\forall_{J'_0}\forall_{J'_1}\dots\forall_{J'_{R''-1}}\Big[\\
&\forall_{\bar{i}_0}\forall_{\bar{i}_1}\dots\forall_{\bar{i}_{R-1}}\left(\bar{\bar{A}}^{I_0 I_1 \dots I_{R-1} J_0 J_1 \dots J_{R'-1}}_{\bar{i}_0\bar{i}_1\dots\bar{i}_{R-1}} = expr\right);\\
&\forall_{\bar{i}_0}\forall_{\bar{i}_1}\dots\forall_{\bar{i}_{R-1}}\left(\bar{A}^{I_0 I_1 \dots I_{R-1} J_0 J_1 \dots J_{R'-1}}_{\bar{i}_0\bar{i}_1\dots\bar{i}_{R-1}} \mathrel{+}= \bar{\bar{A}}^{I_0 I_1 \dots I_{R-1} J_0 J_1 \dots J_{R'-1}}_{\bar{i}_0\bar{i}_1\dots\bar{i}_{R-1}}\right)\Big]
\end{aligned}$$

This can be viewed as a nest of for loops with a child task launch in the innermost loop that computes

$$\begin{aligned}
&\forall_{\bar{i}_0}\forall_{\bar{i}_1}\dots\forall_{\bar{i}_{R-1}}\left(\bar{\bar{A}}^{I_0 I_1 \dots I_{R-1} J_0 J_1 \dots J_{R'-1}}_{\bar{i}_0\bar{i}_1\dots\bar{i}_{R-1}} = expr\right);\\
&\forall_{\bar{i}_0}\forall_{\bar{i}_1}\dots\forall_{\bar{i}_{R-1}}\left(\bar{A}^{I_0 I_1 \dots I_{R-1} J_0 J_1 \dots J_{R'-1}}_{\bar{i}_0\bar{i}_1\dots\bar{i}_{R-1}} \mathrel{+}= \bar{\bar{A}}^{I_0 I_1 \dots I_{R-1} J_0 J_1 \dots J_{R'-1}}_{\bar{i}_0\bar{i}_1\dots\bar{i}_{R-1}}\right)
\end{aligned}$$

which executes the kernel on the tensor blocks and adds the result into the intermediate tensor block.

For this step's code generation, we emit a nest of for loops, with a for loop for each index variable in the expression (even unsplit index variables). The order of the loops is the same as the distribution schedule's index variable ordering. In each for loop, the block index variable i runs from 0 to $P_i - 1$. In the innermost for loop body, we issue a task launch for the kernel task. For each input tensor A in the expression, the innermost loop body computes the mapping key for the tensor σ_A and uses it to index into an array that contains logical regions that store the tensor blocks of A (this was obtained during the distributed tensor loading step). This gives the logical region containing the appropriate tensor block of A needed for the kernel task launch, and that logical region is then added to the task launch through a region requirement (with read-only privilege). The innermost for loop body also computes the intermediate tensor block mapping key $k_I + \lambda_I \cdot k_J$, which it uses to index into `intermediate_fts` to get the future wrapping the logical region that contains the intermediate tensor block needed by the kernel task. It then adds this future to the kernel task launch to give it a reference to the logical region. Adding the future also ensures that this kernel task won't run until the previous kernel task operating on the same intermediate tensor block is finished. When the kernel task is launched, the return value of the task launch is a future wrapping the logical region that contains the new intermediate tensor block as computed by the kernel task. The innermost loop body then finally overwrites the old future in `intermediate_fts` with the new one.

There is one slight workaround needed in this step's code generation due to a quirk surrounding Legion index spaces. Index spaces in Legion are immutable, which means that once a logical region is created, its size cannot be changed. This is an issue if a parent task has a child task that returns an output of variable size that can't be easily bounded, specifically if the parent wants to pass that output as an input to another child task via futures. While the parent task will have the proper privileges needed to access the logical region once the child task ends, it cannot pass these privileges to the second child since the logical region is contained in a future. Thus, naively the second child task would not be able to access the logical region since it would not have the proper privileges (the inline map would fail). If the size were fixed, the parent could create the logical region itself, and just pass it to both child tasks using region requirements while stating the appropriate privilege in the region requirements. However, since the output is variably sized, the parent's logical region may not be big enough to contain the output.

Legion presents a workaround for this problem using partitioning and virtual mapping. Returning to the previous example, the parent task would create a very large logical region (larger than any relevant memory scale) and pass it to both tasks using region requirements, but declare that the region must be virtually mapped. This means that when the child tasks run, that region will not be instantiated as a physical instance. The first child task, after it computes the variably-sized output, partitions off a part of this large region, and stores the output in that logical region via an inline mapping. (This will only create a physical instance for the smaller partitioned region, and not create a physical instance for the parent region.) The first child task then returns the logical region via future, which is then passed to the second child task.

This time, however, when the second child task attempts to inline map the logical region, it will also pass the virtually mapped region it received from its parent task within the inline map's region requirement. This allows the inline map to succeed, since the second child task has privileges on an ancestor of the desired logical region.

We can use a similar solution in our code generation. That is, along with the `intermediate_fts` array, we create an `intermediate_lrs` array that contains the very large logical region that the corresponding logical region in the future was partitioned from. Going back to the first step of code generation (initialization), instead of creating the logical region for the intermediate tensor block the typical way via runtime, the innermost loop body creates a very large logical region and stores it into `intermediate_lrs` array using the key $k_I + \lambda_I \cdot k_J$. The innermost loop body then partitions a small portion of this region for the sentinel value, and inline maps the partitioned logical region to store the sentinel value in it before wrapping it in a future and placing it in `intermediate_fts`.

For the second step of code generation, the innermost loop body would fetch the very large logical region from the `intermediate_lrs` array using the key $k_I + \lambda_I \cdot k_J$, and then add it to the kernel task launch with a region requirement that specifies the region is virtually mapped (with read-only privilege). This would allow the kernel task to read the intermediate tensor block. The innermost loop body would then create a new very large logical region, add it to the kernel task launch with a region requirement that specifies the region is virtually mapped (with write-discard privilege), and then overwrite the logical region in the `intermediate_lrs` array using the same key $k_I + \lambda_I \cdot k_J$. This allows the kernel task to partition part of this logical region for storing its output.

With this, we've generated code for the distributed kernel computation. For the specifics of how the kernel task code is generated, see Section 4.5.3. We can now finally move to the third step of the transformed expression, in which we generate code for summing together the intermediate tensors into the output tensor. This is given by

$$\forall_{I_0}\forall_{I_1}\ldots\forall_{I_{R-1}}\left[\forall_{\bar{i}_0}\forall_{\bar{i}_1}\ldots\forall_{\bar{i}_{R-1}}\left(A^{I_0 I_1\ldots I_{R-1}}_{\bar{i}_0\bar{i}_1\ldots\bar{i}_{R-1}} = \sum_{J_0}\sum_{J_1}\ldots\sum_{J_{R'-1}}\bar{A}^{I_0 I_1\ldots I_{R-1} J_0 J_1\ldots J_{R'-1}}_{\bar{i}_0\bar{i}_1\ldots\bar{i}_{R-1}}\right)\right]$$

Note that the summations on the right-hand side are not restricted by order, and can be done in any manner. Naively, we could view this as a nested loop of only free variables, and compute the sum

$$\forall_{\bar{i}_0}\forall_{\bar{i}_1}\ldots\forall_{\bar{i}_{R-1}}\left(A^{I_0 I_1\ldots I_{R-1}}_{\bar{i}_0\bar{i}_1\ldots\bar{i}_{R-1}} = \sum_{J_0}\sum_{J_1}\ldots\sum_{J_{R'-1}}\bar{A}^{I_0 I_1\ldots I_{R-1} J_0 J_1\ldots J_{R'-1}}_{\bar{i}_0\bar{i}_1\ldots\bar{i}_{R-1}}\right)$$

in a child task. But this could potentially overflow memory for the node that executes the task. Moreover, the tensor sum would be pairwise for efficiency sake and not n-ary, so many of the intermediate tensor blocks for a child task would be waiting in memory to be added. A better solution is to launch summation tasks that sum a pair of tensor blocks, and form a tree of summation tasks with the root summation task returning the desired output tensor block. There will be a tree here for every output tensor block, so in order to hide the latency of summation, we should schedule the summation tasks by launching all summation tasks that are on one tree level before moving to the next level of summation tasks, moving towards the roots.

For this step's code generation, we emit three nested for loops, launching the pairwise summation task in the innermost loop body. The first loop iterates through stride lengths s, starting at 1 and doubling every loop iteration, and stopping when it's at least λ_J. The stride here represents what level of the tree is being iterated through, with stride of 1 being the leaves. The second loop iterates through the mapping keys k_J for $\{J_0, J_1, \ldots, J_{R-1}\}$. However, instead of going from 0 to $\lambda_J - 1$, it instead starts at 0 and increments by twice the stride s every loop iteration, stopping when at least $\lambda_J - s$. The third loop iterates through the mapping keys k_I for $\{I_0, I_1, \ldots, I_{R-1}\}$, going as usual from 0 to $\lambda_I - 1$.

In the innermost loop body, the mapping keys are computed for the two tensor blocks being summed together, which are $k_I + \lambda_I \cdot k_J$ for the first tensor block and $k_I + \lambda_I \cdot (k_J + s)$ for the second tensor block. These are used to index into the `intermediate_fts` and `intermediate_lrs` arrays to get to the futures wrapping the logical regions containing those tensor blocks, along with the very large logical regions from which they were partitioned. The innermost loop body adds the two futures to the summation task launch and adds the two logical regions to the task launch with virtually-mapped region requirements (with read-only privilege). The innermost loop body then creates a new very large logical region, adds it to the kernel task launch with a region requirement that specifies the region is virtually mapped (with write-discard privilege), and then overwrite the logical region in the `intermediate_lrs` array using the first tensor's mapping key $k_I + \lambda_I \cdot k_J$. The return value of the summation task launch is a future wrapping the logical region that contains the sum of the two input tensors. The innermost loop body finally overwrites the old future of the first tensor in `intermediate_fts` with the new one. After these for loops finish executing, the output tensor block for $\{I_0, I_1, \ldots, I_{R-1}\}$ will be located in a future-wrapped logical region in `intermediate_fts` using key σ_I. This generated code thus sums together the intermediate tensor blocks in a tree-like fashion into the output tensor blocks. For the specifics of how the summation task code is generated, see Section 4.5.4.

This section has introduced how SuperTaco generates code to compute distributed tensor algebra, but it isn't the whole picture. Section 4.5.5 describes how tasks are mapped to nodes, which is controlled by the distribution schedule and greatly affects data movement. Section 4.5.6 talks about an aspect of Legion garbage collection that could cause this code generation technique to result in memory overflow, and how we prevent that by placing tensor block evictions in the kernel computation's for loop nest.

4.5.3 Kernel Task

The kernel task used in code generation consists of two steps: the main kernel is evaluated for specific tensor blocks, and then the temporary result is summed into a cumulative intermediate tensor block. First, the original kernel must be evaluated via

$$\forall_{\bar{i}_0}\forall_{\bar{i}_1}\dots\forall_{\bar{i}_{R-1}}\left(\bar{\bar{A}}^{I_0 I_1 \dots I_{R-1}}_{\bar{i}_0\bar{i}_1\dots\bar{i}_{R-1}} = expr\right)$$

This requires the parent task to supply the child task with the logical regions of the block tensor operands, which were attained during the distributed tensor loading step. Which block tensor operands are supplied depends on the values of the block index variables $(I_0, I_1, \dots, I_{R-1}, J_0, J_1, \dots, J_{R'-1}, J'_0, J'_1, \dots, J'_{R''-1})$ when the child task is executed, which is in turn dictated by the for loops in the parent task. These logical regions will be mapped at the start of task execution since they were passed to the kernel task via region requirements.

To generate the code to compute the kernel, we need only supply the original tensor algebra expression to the Taco command-line tool, and it will accordingly generate the C code for the `assemble()` and `compute()` functions to calculate $\bar{\bar{A}}^{I_0 I_1 \dots I_{R-1}}_{\bar{i}_0\bar{i}_1\dots\bar{i}_{R-1}}$ (recall from Section 4.2 that these functions require slight modification for handling dense dimensions efficiently). Note some metadata and pointer marshalling is required to setup the block tensor operands using Taco's native in-memory tensor format, but no large copies are needed here. The generated code thus marshalls the input tensors, and then calls `assemble()` and `compute()` to calculate the kernel.

The next step is to sum this temporary tensor into the previous intermediate tensor block with

$$\forall_{\bar{i}_0}\forall_{\bar{i}_1}\dots\forall_{\bar{i}_{R-1}}\left(\bar{A}^{I_0 I_1 \dots I_{R-1}}_{\bar{i}_0\bar{i}_1\dots\bar{i}_{R-1}} \mathrel{+}= \bar{\bar{A}}^{I_0 I_1 \dots I_{R-1}}_{\bar{i}_0\bar{i}_1\dots\bar{i}_{R-1}}\right)$$

To evaluate this expression, we do not directly sum into the intermediate tensor block as written. Instead we add the previous intermediate tensor block to the temporary tensor to generate a new intermediate tensor block. The parent task provides the old intermediate tensor block to the child task through a future containing a logical region, while the child task returns a future containing a logical region with the new intermediate tensor block.

Recall that since the logical region containing the previous intermediate tensor block is contained in a future, the logical region will not be automatically mapped before the child task runs. The child task must instead inline map the logical region (while passing a reference to the large ancestor logical region provided by the parent task to satisfy privilege checks). This may cause a delay since Legion will have to move the data while the child task is executing instead of before it executes. We hide some of this latency by initiating the inline map before executing the main kernel, so that the child task can do useful work while waiting for a physical instance of the old intermediate tensor block to become available to the processor.

To execute the summation to the old intermediate tensor block, we again call the Taco command-line tool, this time with the summation kernel

$$\forall_{\bar{i}_0}\forall_{\bar{i}_1}\ldots\forall_{\bar{i}_{R-1}}\left((\bar{A}^{new})^{I_0 I_1 \ldots I_{R-1}}_{\bar{i}_0\bar{i}_1\ldots\bar{i}_{R-1}} = (\bar{A}^{old})^{I_0 I_1 \ldots I_{R-1}}_{\bar{i}_0\bar{i}_1\ldots\bar{i}_{R-1}} + \bar{\bar{A}}^{I_0 I_1 \ldots I_{R-1}}_{\bar{i}_0\bar{i}_1\ldots\bar{i}_{R-1}}\right)$$

We then execute the resulting `assemble()` and `compute()` functions to calculate the new immediate tensor block. While marshalling the inputs to the two functions avoids large copies, marshalling the output of `compute()` into a logical region requires a copy of all the data arrays in the tensor block. The logical region itself, after being partitioned from a very large logical region provided by the parent task, must still be mapped inline for access. However, since this logical region was created on this node and has never been mapped before, the logical region is guaranteed empty and will inline map fast. After this computation is done, we also destroy the logical regions for the old intermediate tensor block and its ancestor so that Legion may garbage collect them.

One optimization we make to this kernel task is that we use a small sentinel value in the old intermediate tensor block future to indicate if there is no old intermediate tensor. This allows us to forgo initialization of the intermediate tensor block to the zero tensor, along with the addition of a tensor to the zero tensor. Instead, when the child kernel task sees the sentinel, it outputs the temporary tensor as the new intermediate tensor block and avoids the addition.

4.5.4 Summation Task

The summation task in code generation just computes a pairwise addition between its input tensors, similar to the last half of the kernel task. The logical regions containing the first and second tensor blocks are extracted from their futures, and inline mapped (passing in the appropriate ancestor logical regions for privilege checks). The task must then wait for both logical regions to map to physical instances before proceeding. Unfortunately there is no way to hide the latency in this scenario, since unlike with the kernel task there is no kernel to execute.

Once the logical regions have mapped, the summation task code calls the same `assemble()` and `compute()` functions used during summation in the kernel task. Again, the input tensor marshalling into Taco's native tensor format is cheap and avoids large copies, but marshalling the output into a logical region is expensive due to having to copy the `.tcs` data arrays. The output logical region needs to be inline mapped after being partitioned off from the very large logical region provided by the parent task, but similar to the case with the kernel task, this should be fast since it's never been mapped before. Once the computation is complete, the summation task destroys the logical regions belonging to the input tensors (and their ancestors) so they can be garbage collected by Legion, and returns the output logical region to its parent in the form of a future.

4.5.5 Task Mapping

One of the considerations left out so far in our discussion is how tasks are mapped to processors for execution, and where the data in logical regions is stored and when it moves around. Having an understanding of this is necessary to minimize unneeded data movement and express different schemes for distributing computation

Tasks are mapped to processors using Legion mapper objects, which are user-defined objects that Legion can query during runtime to let the user make mapping decisions (see Section 2.4 for an overview). You can pass information about where a task should map in a Legion program through the task launch API, which lets you tag the task with a mapper tag. When the runtime asks the mapper to map a task, it will pass the task's tag information along to the mapper. In this fashion, the Legion application can communicate with the mapper object about which processor the tag should map to.

Data movement in Legion, in contrast, is handled automatically. That is, if a Legion task runs on a processor that uses a logical region, Legion will automatically create a physical instance of that logical region on that node if it's not already on that node. That is, it will move the needed data by copying it from some other physical instance on another node. This means that to minimize data movement, tasks that use the same data should have a way of being mapped to the same node. SuperTaco implements a static mapper class that provides this functionality.

Before describing this mapper class, it's important to make a distinction between nodes and processors. It is often the case that a user doesn't need to map a task to a specific processor, but rather any processor on a node. For example, if a task is going to use a logical region and there's already a physical instance on some node, it doesn't matter which processor on that node that the task runs on. The physical instance won't need to be copied over the network provided the task is mapped to some processor on that node. For this reason, Legion provides a way to assign a task to a group of processors instead of a single processor. A task assigned to a group of processors will execute on whichever processor in the group becomes idle first, which effectively provides a form of load balancing over those processors.

Using this functionality, the SuperTaco mapper can assign a task to a given node ID instead of a specific processor. The SuperTaco mapper class subclasses Legion's default mapper class while overriding two methods of the default mapper. The first is `default_policy_select_initial_processor()`, which the default mapper class uses in `select_task_options()` to determine which processor to send the task to for remote mapping. In this function, the SuperTaco mapper checks the tag of the task for a binary flag. If the flag is unset, the SuperTaco mapper will map the task like the default mapper would. However, if the flag is set, the SuperTaco mapper will extract a node ID from the rest of the tag, and take it modulo the number of nodes to ensure it's a valid node ID. It will then give a processor on that node as the return value for `default_policy_select_initial_processor()`, which will cause the task to be remotely mapped at that node.

The second method of the default mapper class that the SuperTaco mapper overrides is `default_policy_select_target_processors()`, which the default mapper class uses in `map_task()` to determine which processors a task can map to. Once again, the SuperTaco mapper checks the tag of the task for a binary flag, and if unset, maps the task like the default mapper. If the flag is set, the SuperTaco mapper will return a list of all local processors for `default_policy_select_target_processors()`, which will cause the task to be load-balanced over the processors on that node.

The SuperTaco mapper thus allows tasks to be targeted towards nodes and load balanced over their processors via the mapper tag in task launches. However, this does not decide what function should be used for the mapper tag for a given task. Since we desire tasks that use the same data to have the capacity to map to the same node, it makes sense here to use the same labels we use for data in the mapper tag function. In this context, the data is tensor blocks, and the labels are index variables. A natural choice is then to use the mapping key function $\sigma()$ on some set of index variables as the mapper tag.

First consider how the tensor blocks are mapped. The node they reside on (at the start of computation) is the node they were assembled on, i.e. the node that the tensor block assembly task ran on. When this task is assembling a tensor block $A^{I_0 I_1 \ldots I_{R-1}}$, the natural choice for the mapper tag for this task would be $\sigma(\{I_0, I_1, \ldots, I_{R-1}\})$, since it would evenly distribute the tensor blocks across all nodes. We could potentially leave an index variable out, but then splitting along that index variable would be pointless since they would all end up on the same node regardless. For example, consider if a matrix B^{IJ} was had a mapper tag of $\sigma(\{I\})$ for tensor block assembly. All the blocks belonging to the same row would end up on the same node, which could have been accomplished by not splitting along J at all (and in that case, $\sigma(\{I\}) = \sigma(\{I, J\})$ since $J = 0$ and $P_j = 1$). Hence, in our generated Legion programs, the tensor block assembly task for $A^{I_0 I_1 \ldots I_{R-1}}$ uses a mapper tag function of

$\sigma(\{I_0, I_1, \ldots, I_{R-1}\})$.

For the kernel task, the mapper tag is determined by the distribution schedule's *mapping variables*. That is, the kernel task's mapping tag is the mapping key function $\sigma()$ applied to the set of the distribution schedule's mapping variables. This allows us to express a number of various task schedules. For example, the owner's compute rule, in which the node that stores an output block is responsible for performing the computations that sum into that block, can be expressed by making the mapping variables the same as the set of the output tensor's index variables. (This is the default behavior by SuperTaco if no mapping variables are given.) If we were computing SpMV $a_i = \sum_j B_{ij} c_j$ and had split over both i and j, then to prevent the data movement of the matrix blocks, we could use the set of mapping variables $\{i, j\}$. In that case, the input vector block would be copied to each node.

For the summation task, the mapper tag is the same as the mapper tag for the kernel task ($\sigma()$ applied to the set of the distribution schedule's mapping variables), with the exception that any non-parallel hoisted reduction variables J'_m that appear in the mapping key function are set to $P_{j_m} - 1$ in that function. We use this because at the end of the kernel computation phase, the intermediate tensor block $\bar{A}^{I_0 I_1 \ldots I_{R-1} J_0 J_1 \ldots J_{R'-1}}$ was last computed during a kernel task launch where the for loop variables $(I_0, I_1, \ldots, I_{R-1}, J_0, J_1, \ldots, J_{R'-1}, J'_0, J'_1, \ldots, J'_{R''-1})$ were set according to $(I_0, I_1, \ldots, I_{R-1}, J_0, J_1, \ldots, J_{R'-1}, P_{j'_0} - 1, P_{j'_1} - 1, \ldots, P_{j'_{R''-1}} - 1)$. That is, the free variables and output-extension variables were as specified by the block variables in $\bar{A}^{I_0 I_1 \ldots I_{R-1} J_0 J_1 \ldots J_{R'-1}}$, but the remaining variables had exhausted their for loops and accordingly had $J'_m = P_{j_m} - 1$. This means the node that launched the task (and thus the node that currently stores the intermediate tensor block) is determined by the kernel task's mapper tag for that point. Recall that the first tensor in the summation has mapping key $k_I + \lambda_I \cdot k_J$, meaning it corresponds to the intermediate tensor block $\bar{A}^{I_0 I_1 \ldots I_{R-1} J_0 J_1 \ldots J_{R'-1}}$, and that the summation output tensor is stored with the same key and ends up on the same node. This means that when the summation task launch executes, the second tensor is effectively transferred to the node containing the first tensor and the summation of the two tensors stays at that node, taking the place of the first tensor. Using this mapping technique, we can guarantee that each summation task only has to move one of its input tensors, which minimizes data movement.

With this, we have defined the mapper tag for each of the tasks relevant to the computation phase of tensor algebra distribution. For any remaining tasks, e.g. tasks during tensor loading or tensor storing, the tasks are distributed evenly across the nodes unless stated otherwise.

4.5.6 Tensor Block Eviction

One pitfall of Legion's approach to data is that physical instances will not be garbage collected unless either the logical region is destroyed, or the logical region is mapped to some node with read-write privileges, at which point all other physical instances become invalidated (or "evicted"). To understand how this becomes a problem, consider the SpMV example in the last section where we had split over both i and j. Suppose we used owner's compute for mapping, and knew that a node was big enough to hold a block B^{IJ}, but not an entire row B^{I}. Under owner's compute, a single node would be responsible for computing all parts of c^{I}, which means that over time, the node would eventually have loaded all the tensor blocks needed to compute c^{I}. That would include B^{IJ} for all J, which constitutes a row of J. Since none of those blocks would be garbage collected, this would result in an overflow of memory on the node.

To prevent this from happening, it is useful to think of all the kernel tasks that map to the same node by virtue of having the same mapping key; we call such kernel tasks a *task group*, and accordingly the task groups are labelled by the value of the kernel task's mapping key (equivalently, it could be labelled by the tuple of the distribution schedule's mapping variables that derived that mapping key). Tasks within a task group using different tensor blocks from the same tensor is what leads to multiple tensor blocks from the same tensor ending up on one node, which causes overflow.

A natural solution to this problem is to make sure that before a task in a task group loads a new tensor block from a tensor onto a node, that the old tensor block from that tensor is evicted from that node. However, recall that the only way to cause eviction, which is to open a logical region with read-write privileges, causes eviction from all nodes (and thus all task groups). That is, there is no way to evict a tensor block from a single task group; tensor block eviction is an all-or-nothing process.

This means that task groups to some extent must stay synchronized about their evictions; if not, they could potentially disrupt another task group that was relying on a particular tensor block to stay on a node for a while for efficient computation of the tasks within the task group. To this end, we desire to perform evictions in the loop nest of the kernel computation step, since it offers natural points of synchrony in each loop body.

As for where to evict a block tensor to, the solution that uses the least data movement is to have it stay on the node that constructed it. That is, when SuperTaco emits code to evict a tensor block, it will consist of a task launch with a region requirement with read-write privileges for the logical region containing the tensor block, and its mapper tag will be the same as the mapper tag for the tensor block assembly task (which from the previous section is the mapping key function $\sigma()$ applied to the index variables of the tensor). In this fashion, if the tensor was never moving in the first place, then the tensor will continue to not move by evicting it to where it was constructed. Note that the body of the evict task is empty; the only purpose of the task is to invalidate physical instances, and it's done that by the time it executes so it can immediately return.

The next question is where in the loop nest to place evictions. To answer this, consider a tensor block $A^{I_0 I_1 \dots I_{R-1}}$. Within a task group, certain block variables will stay "fixed", namely the block variables that are mapping variables. Only the non-mapping block variables of the tensor will change in the task group, and when any of them do, we need to evict the last tensor block accordingly. Inside the loop nest, this manifests when any of the non-mapping block variables increments in a loop header. However, we only need to evict in the deepest such loop, since that will in turn also evict for increments in the other non-mapping block variables. We also shouldn't evict in deeper loop nests, since that would cause unnecessary data movement through unnecessary eviction. Note that if all the block variables of the tensor are mapping, then there is no need to evict it.

Thus, for each tensor A, SuperTaco decides where to place the tensor block eviction for $A^{I_0 I_1 \ldots I_{R-1}}$ in the loop nest by taking a list of the tensor block index variables and subtracting out the set of mapping variables, and if the resulting set is non-empty, then finding the deepest loop body with a loop variable in the set of such non-mapping tensor block variables. However, SuperTaco can't just place an eviction task launch at the bottom of this loop body. This is because the tensor block may depend on block index variables that haven't appeared yet in the loop nest. Conceptually, the later loops containing a block index variable that the tensor depends on means that more than just a single tensor block has been accessed since the last eviction (one for each tuple of such later variables), and accordingly more than just a single tensor block eviction is needed. Hence, SuperTaco instead emits a for loop nest at the bottom of this loop body, with a loop for each of the tensor's block index variables that occurs later in the loop nest. At the bottom of this loop nest, is when SuperTaco emits the eviction task launch, as described above.

The tensor block evictions have now been placed within the loop nest so that the old tensor blocks within each task group will be evicted once a new tensor block needs to be loaded, as determined by one of the non-mapping tensor block variables incrementing. However, while the non-mapping tensor block variables incrementing signals when the tensor block evictions should occur, it doesn't force the next series of kernel task launches that use the incremented values to happen after eviction is complete. That is, there's nothing making the next series of task launches dependent on the tensor block evictions, so new tensor blocks could be loaded into a task group's node before the old tensor block is evicted. (Note that this is not true for past task launches and evictions, since the read-write privilege of evict task launches means they will occur after past task launches that use the same tensor blocks.)

SuperTaco solves this problem by manually implementing the dependencies using futures; this requires modifying the evict task so that it returns a dummy value instead of doing nothing. Note a kernel task launch is only dependent on the last tensor block eviction corresponding to its task group, which means they have the same values for the tensor block variables that are also mapping variables. Suppose for a tensor A, we call the set of its mapping tensor block variables $\{M_0, M_1, \ldots, M_{T-1}\}$; call its mapping key function $\sigma(\{M_0, M_1, \ldots, M_{T-1}\}) = \sigma_M$ and its mapping key size λ_M. Then SuperTaco generates code to create an array of futures `evict_fts` with λ_M length right before the kernel computation loop nest, and to initialize all futures to ones that immediately resolve. When an evict task launch for tensor A is executed, the mapping key σ_M will be computed and the future returned from the evict task launch stored in `evict_fts` using index σ_M. Additionally, in the kernel task launch loop body, SuperTaco adds code to compute σ_M and indexes into `evict_fts` with it to get back the evict task launch future, adding it to the kernel task launch.

By augmenting the kernel computation loop nest with these evictions, SuperTaco ensures that the tasks within a task group can never load two different tensor blocks from the same tensor simultaneously, and thus bounds the memory consumed by a task group at the cost of a loss in parallelism. Note that if the number of task groups (equal to the mapping key size of the kernel task's mapping key) is many times larger than the number of nodes, then many task groups will alias to the same node, which in turn may cause an overflow even if each individual task group is limited to a single tensor block from each tensor. Thus, care must be taken when choosing the number of split ranges for mapping variables.

4.6 Distributed Tensor Storing

The third and last phase of a Legion program generated by SuperTaco is merging the tensor blocks together and storing the data to disk. Similar to tensor loading, a tensor may not be able to fit into a single node's memory, so this needs to be done in a distributed fashion. Most of the code that makes up this component is static, and the routines share similarities with the ones for distributed tensor loading.

During the first step (Section 4.6.1), each of the block tensors is split into tensor block pieces via load balancing. Unlike tensor loading, actual tensor size is not used to determine where to split the tensor block into pieces. Instead, the top-level task partitions the space of index coordinates into equally sized pieces (*coordinate space pieces*), with the number of pieces equal to the number of processors; this gives a set of coordinates that would divide the the entire output tensor into roughly equally sized pieces provided the output tensor is not significantly skewed. The parent task creates a task for every output block, and each task iterates through its output block, finding the position vectors that yield index values close to the desired coordinates (but not going over). In this fashion, each output tensor block is split into tensor block pieces.

For the second step (Section 4.6.2), each processor is assigned a coordinate space piece, and gathers all the tensor block pieces that are within that same piece of space. Each tensor block piece comes with the offsets of each of its data arrays within the original data arrays to allow iteration. Each processor constructs a tournament tree, with each of the leaves being pointers to a tensor block piece and its iteration state. This allows for a multi-way merge between the tensor block pieces, which can create a piece of the output tensor. However, this process requires two parts.

The first part (Section 4.6.3) consists of each processor iterating through the entries of its output tensor piece via the tournament tree and gathering partial header information for the tensor. Similar to 4.4.2, the partial header information is sent back to the parent task, allowing it to assemble the full `.tcs` header for the output tensor. The parent task then creates the file on disk, and writes the header to it.

During the second part (Section 4.6.4), the parent task computes certain state information the processors need to make a second pass, in which it assembles the tensor pieces. The parent sends that state to the each processor, and the processors make their second pass to assemble the tensor pieces. For the final step (Section 4.6.5), each processor then must sequentially writeout its data arrays to disk.

4.6.1 Load Balancing

Unlike tensor loading, we cannot split the tensor blocks based on their size. Instead, tensor blocks must be split on their coordinate value, i.e. the values of their indices. The parent task determines the space of coordinates as a Cartesian grid, with each index variable bound by a minimum and maximum. We could potentially determine these minima and maxima from the split ranges, but this could be inaccurate since they're user-defined. Instead, during the first-pass of streaming through input tensors in Section 4.4.2, recall that each processor additionally collects minima and maxima information to send to its parent task. The parent task here is augmented to compute a global minima/maxima for the whole tensor, which gives minima/maxima on the corresponding index variables. Pooling then gives a minima and maxima per index variable.

For an index variable i, let the minimum and maximum values found be i_{min} and i_{max}, so that the index size is $Z_i = i_{k,max} - i_{k,min} + 1$. For an output tensor $A_{i_0 i_1 \ldots i_{R-1}}$, the parent task computes the size of the space of index coordinates as $Z = Z_0 \cdot Z_1 \cdot \ldots \cdot Z_{R-1}$. Next, the parent task computes the coordinates that give size offsets close to $Z/N, 2Z/N, \ldots, (N-1)Z/N$, where the size offset of a coordinate $(i_0, i_1, \ldots, i_{R-1})$ is $(i_{R-1} - i_{R-1,min}) + Z_{i_{R-1}} \cdot ((i_{R-2} - i_{R-2,min}) + Z_{i_{R-2}} \cdot \ldots \cdot ((i_1 - i_{1,min}) + Z_{i_1} \cdot ((i_0 - i_{0,min}))) \ldots)$. Note that this last function is the mapping key function, but with P_i replaced by Z_i and I_k replaced by $i_k - i_{k,min}$. Accordingly, computing the inverse to find the coordinates is the same as computing σ^{-1} as described in 4.5.2.

With this, the parent task has computed coordinates that span the range from $(i_{0,min}, i_{1,min}, \ldots, i_{R-1,min})$ to $(i_{0,max}, i_{1,max}, \ldots, i_{R-1,max})$, and are equidistant in coordinate space (i.e. if you sorted the list of integer points in the Cartesian grid between the min and max points, the coordinates would be equally spaced in the list). The parent task now launches a task for every output block (using the output block's mapping key), passing the set of coordinates to each task via logical region. Each child task then iterates through its output block as described in 4.4.2, and determines the position vectors that give index values closest to the given coordinates (without going over). These position vectors thus conceptually break each tensor block into tensor block pieces.

To actually break the tensor block into tensor block pieces, each child task then creates a logical region for every coordinate space piece, and then for each tensor block piece, copies over the full header and the appropriate sections of the data arrays into the corresponding logical region. (The position vectors are used here to determine the size of the data array sections.) The offsets of each of the sections of the data arrays in their original data arrays is stored as a vector, and packed into the header to allow for iteration. The headers here are updated to point at the new byte offsets within the logical regions. The child tasks destroy their original tensor block logical regions so they can be garbage collected, and each child task then returns the logical regions corresponding to the tensor block pieces by placing them in a containing logical region, which was passed to the child task by the parent. After the parent task launches all the child tasks, it then maps the containing logical regions inline and waits for them to become available. This gives the parent task the logical regions for all the tensor block pieces.

4.6.2 Tournament Tree

Now that each tensor block has been split into pieces by position vectors, each processor will create a tournament tree and pull in the appropriate tensor block pieces to create an iterator for a multi-way merge. In the Legion program, the parent task launches a tournament tree task for each coordinate space piece, and for each task launch adds to it the logical regions for all tensor block pieces that correspond to the same coordinate space piece.

When the child task executes, it creates a tournament tree within a logical region, where there are enough leaves for each of the tensor block pieces; the remaining leaves are negative infinity sentinels. This tournament tree can perform a k-way merge on the tensor block pieces by keeping in each leaf a position vector (and index vector) that points into the appropriate tensor block piece. In this sense, the tournament tree can be treated like an iterator; when a value is consumed, the branch corresponding to the value is emptied, the leaf node of the branch has its tensor block piece make one step forward in iteration to the next position vector, and then the branch is updated with the new index values from that new position. When a tensor block piece is exhausted, its leaf becomes negative infinity. When all leaves are exhausted, the root node will become negative infinity, signalling the iterator is exhausted. Resetting just involves resetting the position vectors of all the tensor block pieces, fetching the index values, and updating the tournament tree. Using this schema, the tournament tree is an iterator over the merged tensor block pieces. Note that since this merge is over all tensor blocks, then the result must be the piece of the tensor that lies in the coordinate space piece. The tournament tree is then an iterator over that tensor piece.

Also note that all the components of this tournament tree iterator (the tournament tree itself, each of the tensor block pieces) are logical regions. This means that they can be returned to the parent task via being packaged in a containing logical region, and as long as the parent launches the next task that needs that iterator on the same node (which can be accomplished via the mapper tag), there will be no movement of that data over a network. The child task thus returns the logical regions of the components that make up the tournament iterator to the parent task.

4.6.3 Tensor Header Assembly

The tournament tree from the previous section gave us an iterator over the tensor piece spanned by each coordinate space piece. For the next step, each processor gathers partial header information about the tensor piece it has, similar to how tensor block pieces were processed in Section 4.4.2. That is, we iterate through the entries of the tensor piece and gather information similar to the first-pass of `.tns` to `.tcs` conversion; we keep vectors that store the level size, minimum and maximum values, and additionally copies of the first and last lines encountered to account for edge effects when the information is merged together with that of other processors.

In the Legion program, this is parallelized similar to the last section. The parent task launches a tensor header assembly task for every tensor piece/coordinate space piece, passing along the tournament tree iterator logical regions, and the parent task receives from each processor the header information they gathered while iterating through the tournament tree (via logical regions). The parent then reconstructs the .tcs header for the output tensor based on the partial header information it received from each processor. (Again, this is primarily pooling minima/maxima, summing level size vectors, and accounting for edge effects.) The parent task then opens the file corresponding to the output tensor, and writes out its header to the file before closing it.

4.6.4 Tensor Piece Assembly

Now that the output tensor's `.tcs` header has been computed, the next step is for each processor to make a second-pass through its tensor piece and assemble the piece itself, similar to how tensor block pieces were processed in Section 4.4.3. Once again we iterate through the tournament tree, and the callback resembles the second-pass of `.tcs` conversion to form the tensor pieces.

Recall from Section 4.4.3 that the second-pass of `.tcs` conversion can't be as easily parallelized due to the level counters being wrong for a processor that only passes through a tensor piece instead of the whole tensor. We use the same solution from that section, i.e. the parent task computes partial sums over the level size counters it collected from each processor (and accounts for edge effects), and the parent additionally provides information to each processor about the sizes of the data arrays in that processor's tensor piece.

This allows the parallelization of the conversion's second-pass, and generates tensor pieces on each processor. In the Legion program, parallelization is similar to the previous section, but this time the parent task doesn't access the output of the task, i.e. the tensor pieces. The parent task does, however, have references to the logical regions holding that data, so it may pass those logical regions to other tasks. Also note that since the tournament tree iterators are no longer needed after the tensor pieces are assembled, the logical regions making up the tournament tree iterator are destroyed.

4.6.5 Tensor Writeout

The final step of tensor storing is writing the tensor pieces back to disk. This requires each processor to write its tensor piece to disk in-order, since the `pos`, `idx`, and `val` arrays within the tensor pieces can be concatenated to form the data section of the tensor. This cannot be done in parallel since the NFS filesystem does not allow processors on different nodes to write to the same file at the same time. Note that the generated code uses the same mapping key for this writeout task as the previous task, otherwise the entire tensor piece would move between nodes.

In the Legion program, a child task is launched per tensor piece with its corresponding tensor piece logical region, but the child tasks are sequential and in-order. The first child task holds the first tensor piece in its memory and writes out the first tensor piece to file, returning a future to the parent task with the disk offsets into each of the data arrays when its tensor piece ended. The parent task sends these offsets (via task argument) to the next child task to execute, which is responsible for writing out the second tensor piece starting at the data array offsets given by the first tensor piece. By repeating this process, the child tasks sequentially write their tensor pieces to file.

Thus, through these steps, we are able to store the tensor to disk in a distributed fashion. Note that our guarantees are not as strong here about load balancing compared to tensor loading, since we divide tensors based on coordinates instead of their actual size and accordingly any skew in the data density will cause imbalance. However, provided the output of the tensor operation is sufficiently random, this method of load balancing will be adequate.

Chapter 5

Evaluation

5.1 Experimental Methodology

To evaluate the performance of SuperTaco, we analyze the scalability of our generated programs for sparse matrix-vector (SpMV) multiplication $a_i = \sum_j B_{ij} c_j$ and tensor-times-matrix (TTM) multiplication $A_{ijk} = \sum_l B_{ijl} C_{kl}$. We specifically look at strong scaling, in which the problem size remains fixed but the number of cores/processors in the system increases. For each experiment, we first test processor scaling on a single node to test multicore performance. We then see how performance scales as we increase the number of nodes up to 10 while using 4 cores per node and then 20 cores per node. We only measure the execution time of the distributed computation, and not tensor loading or storing. This is because in practical applications, loading and storing are done as seldom as possible due to the unavoidable cost of disk I/O; data is kept in memory as long as it can be. To assess the overhead of splitting and data movement, we perform an additional experiment in which we increase the number of split ranges for a reduction variable (without output-extension) for an owner's compute schedule at fixed core count (5 nodes with 20 cores per node), and observe the corresponding slowdown.

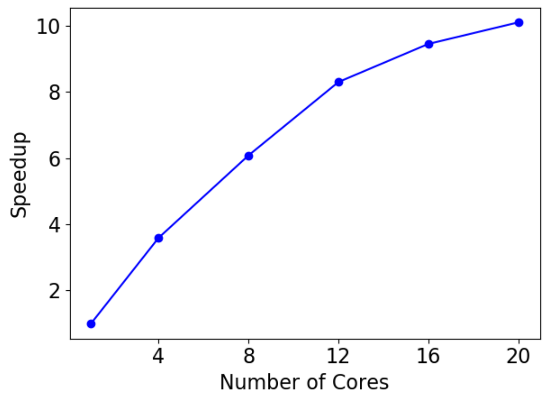
Evaluation was performed on an Infiniband cluster, with each node possessing two 12-core CPUs and 128 GB of memory. The CPUs were Intel Xeon E5-2695 v2 CPUs at 2.40 GHz, with 32 KB L1 data cache, 32 KB L1 instruction cache, 256 KB L2 cache, and 30720 KB L3 cache. Generated Legion programs and the Legion runtime were compiled with GCC 8.2.0 using `-O3 -march=ivybridge`. Legion is configured to run using GASNet for networking, which is in turn configured to use `ibv-conduit` for communicating over Infiniband. We reserved two cores per node as utility cores for Legion, leaving 22 cores per node for compute at maximum.

5.2 SpMV Results

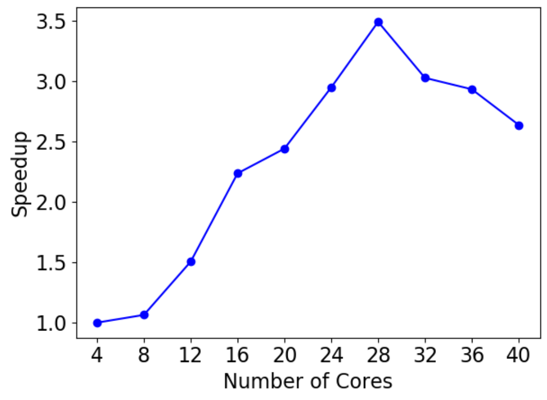
SpMV is an algebraic kernel $a_i = \sum_j B_{ij}c_j$ that appears widely throughout scientific and engineering applications. In this section, we see how the computation of SpMV scales when distributed using SuperTaco. The sparse matrix B_{ij} we use is formatted CSR (i.e. as $(\text{dense}_{d_1}, \text{sparse}_{d_2})$) while the vector c_j is dense. The matrix has dimensions of $50\,000\,000 \times 50\,000\,000$ and is randomly populated with a density of 1×10^{-6}. In `.tns` form it takes up 72 GB, while its `.tcs` form takes up 38 GB.

For the distribution schedule, splitting along j (columns) is expensive since it must either be sequential or require combining partial sums due to j being a reduction variable. There's not much benefit in splitting along j either, since the input vector c_j is relatively small compared to the sparse matrix, and any parallelism we get from using partial sums could also just be attained by splitting more along i (rows) instead. So for index splitting, we only split along i. We shall use a number of split ranges equal to the number of processors being tested. For the mapping variables of the schedule, we use i. This, along with j not being split, means that each node contains a row block of $B^{I0}_{\bar{i}j}$ and computes the corresponding output row block $a^{I}_{\bar{i}}$. The input vector c^0_i is not blocked, and correspondingly every node gets a copy of it. Since i is a free variable and accordingly in the output already, we can't extend along it, so the schedule has no output-extension variables. With this, we've completely described our schedule.

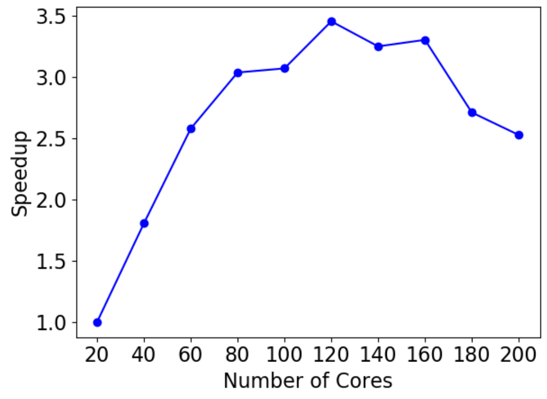
The results of our strong scaling experiments for SpMV are shown in Figure 5-1. For single-node, performance scales almost linearly for the first few cores and then degrades to 50% strong scaling efficiency, speeding up 10.1x when going from 1 to



(a) Scaling across processors for a single node.



(b) Scaling across nodes with 4 cores per node.



(c) Scaling across nodes with 20 cores per node.

Figure 5-1: Strong scaling analysis for SpMV.

20 cores. When we scale across nodes instead, we get similar efficiencies; for 4 cores per node, the strong scaling efficiency is 50% when going from 1 to 7 nodes due to a 3.49x speedup, and for 20 cores per node, the strong scaling efficiency is 58% when going from 1 to 6 nodes due to a 3.45x speedup.

For experiments past 6-7 nodes, the performance noticeably degrades as shown in Figures 5-1b and 5-1c. Note that this degradation occurs around a certain number of nodes and not a certain number of processors, indicating that node-to-node communication could potentially be to blame.

The distribution schedule above for SpMV minimized data movement, since the matrix rows and output vector rows were already assembled on the nodes they needed to be on for computation; the only tensor that needed moving was the input vector, which is relatively small. While this is good for efficient computation, it doesn't give us information about the costs of data movement. We also don't get information about the cost of index splitting; previously we only split as much as needed to make sure all processors were utilized. However, scenarios can arise in which we need to split more, e.g. if the tensor blocks don't fit into a node's memory otherwise.

Our next experiment thus performs SpMV using a suboptimal distribution schedule. In this case we split along i, but also along reduction variable j. To force parts of the matrix to move around the network, we set the mapping variable to only i (owner's compute). We don't use output-extension on j, since owner's compute here means different B^{IJ} can't be computed on simultaneously for the same I, i.e. tensor block evictions would serialize the task launches that output-extension along j would try to parallelize. For the index variable ordering, it's more efficient to use (j, i) since there is no parallelism along j. To ensure we don't underutilize or overutilize the processors, the kernel mapping key size should be around the number of processors, so we set the number of split ranges for i to be the number of processors. The only variable in our distribution schedule remaining is the number of split ranges for j, which will be varied during the experiment.

The results of this overhead analysis experiment are shown in Figure 5-2. Note that we use a smaller sparse matrix here of size $300\,000 \times 300\,000$ with a density of

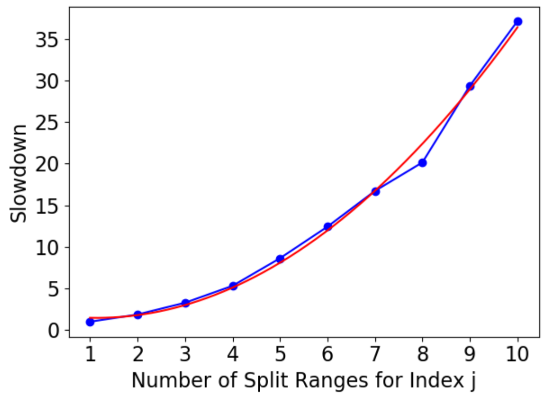


Figure 5-2: Overhead from increasing the number of splits along reduction variable j in SpMV. The red curve is a quadratic fit with equation $y = 0.445x^2 - 1.015x + 2.043$.

1×10^{-2}, since the scale of the slowdown makes experiments longer. The slowdown here is super-linear with the number of split ranges, and although it appears to almost be linear when the number of split ranges nears 10, the slowdown for 20 split ranges is 150x (compared to 37.1x at 10 split ranges). Using polynomial fitting determines that the curve is close to a quadratic fit (shown in red in Figure 5-2); the quadratic curve predicts a slowdown of 160x at 20 split ranges, which is close to the actual 150x. The overhead of splitting along the reduction variable for SpMV is thus nearly quadratic in the number of split ranges.

5.3 TTM Results

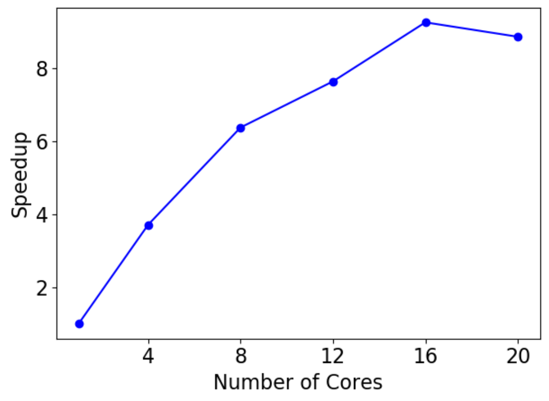
While SpMV is a ubiquitous kernel, it only involves matrices and vectors, and does not take advantage of Taco's ability to generate code for higher-order tensor expressions. The TTM kernel $A_{ijk} = \sum_l B_{ijl}C_{kl}$ involves a rank-3 tensor, and is central to the Alternative Least Squares (ALS) method of solving non-convex optimization problems [14]. In this section, we see how the computation of TTM scales when distributed using SuperTaco. The sparse tensor B_{ijl} we use is formatted CSF (i.e. as $(\text{sparse}_{d_1}, \text{sparse}_{d_2}, \text{sparse}_{d_3})$) while the matrix C_{kl} is dense. The tensor has dimensions of $500\,000 \times 50\,000 \times 5000$ and is randomly populated with a density of 1×10^{-6}, while the matrix is 32×5000. This is meant to be a slightly larger imitation of the

Netflix Prize tensor (approximately $480\text{K} \times 18\text{K} \times 2\text{K}$ with a sparsity of 5.79×10^{-6}), which has previously been used to evaluate TTM implementations [15]. In `.tns` form the tensor takes up 3.6 GB, while its `.tcs` form takes up 3.8 GB.

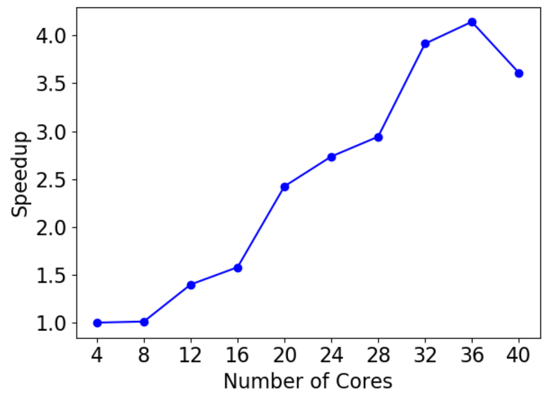
For the distribution schedule, splitting along the reduction variable l is again expensive since it must either be sequential or require combining partial sums, and the benefits in this context are low. We shouldn't split along k either, since k's dimension size is 32 and we may want to scale past that. The best candidates for splitting are then i and j. We could potentially split them both, but this would complicate analysis. Since the first mode is largest in dimension size, we choose to split i since there will be a less of a chance of equidistant split points resulting in skewed block sizes. For the mapping variables of the schedule, we just use i. Then in terms of mapping, each node contains a row block of $B^{I00}_{\bar{i}jl}$ and computes the corresponding output row block $A^{I00}_{\bar{i}jk}$. The input matrix C^{00}_{kl} is not blocked, and correspondingly every node gets a copy of it; this is fine since it only contains 160K entries as a dense matrix, which comes out to 1.28 MB for the `val` array. Since i is a free variable and accordingly in the output already, we can't extend along it, so the schedule has no output-extension variables. With this, we've completely described our schedule.

The results of our strong scaling experiments for TTM are shown in Figure 5-3. The trends are somewhat similar to SpMV. For single-node, performance scales almost linearly for the first few cores and then degrades to 58% strong scaling efficiency, speeding up 9.25x when going from 1 to 16 cores. Unlike the SpMV case there is a slight dip in performance at 20 cores; this could potentially be due to TTM requiring a much larger write bandwidth since it's writing to a tensor instead of a vector. More specifically, for this TTM computation the output tensor has 1.81 billion non-zero entries while the input tensor only has 125 million.

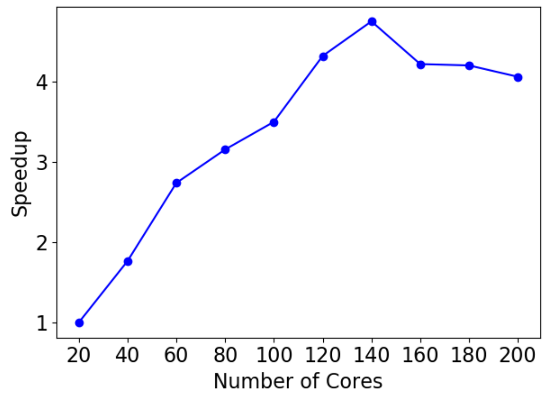
When we scale across nodes instead, the efficiencies are slightly different depending on whether we use 4 cores per node or 20 cores per node. In the former case, the strong scaling efficiency is 46% when going from 1 to 9 nodes due to a 4.14x speedup. In the latter, the strong scaling efficiency is 68% when going from 1 to 7 nodes due



(a) Scaling across processors for a single node.



(b) Scaling across nodes with 4 cores per node.



(c) Scaling across nodes with 20 cores per node.

Figure 5-3: TTM strong scaling analysis

to a 4.75x speedup. Lower cores per node led to less efficiency, but more nodes could be used before performance degradation.

Similar to the case of SpMV, the degradation in performance happens closer to a certain node count threshold than a processor count threshold. However in this case, the number of processors shifts where that threshold is slightly; more cores per node brings down the node threshold.

Once again our distribution schedule used above minimized data movement, since the tensor rows and output vector rows were already assembled on the nodes they needed to be on for computation; the only tensor that needed moving was the input matrix, which is only 1.28 MB. While this is good for efficient computation, it doesn't give us information about the costs of data movement. We also don't get information about the cost of index splitting.

Our next experiment thus performs TTM using a suboptimal distribution schedule. In this case we split along i, but also along reduction variable l. To force parts of the tensor to move around the network, we set the mapping variable to only i (owner's compute). We don't use output-extension on l, since owner's compute here means different B^{I0L} can't be computed on simultaneously for the same I, i.e. tensor block evictions would serialize the task launches that output-extension along l would try to parallelize. For the index variable ordering, it's more efficient to use (j, k, l, i), with (j, k) on the outside since they're not split, then l since there is no parallelism along l, then i since kernel tasks can be parallelized along it. To ensure we don't underutilize or overutilize the processors, the kernel mapping key size should be around the number of processors, so we set the number of split ranges for i to be the number of processors. The only variable in our distribution schedule remaining is the number of split ranges for l, which will be varied during the experiment.

The results of this overhead analysis experiment are shown in Figure 5-4. We use the same tensors and matrices here as the previous TTM experiment. The slowdown curve is much less smooth than the SpMV case. While the general trend is for the slowdown to get worse, there's a bump in the slowdown curve where it rises fast and then falls suddenly before increasing. Once again, the curve appears to be almost

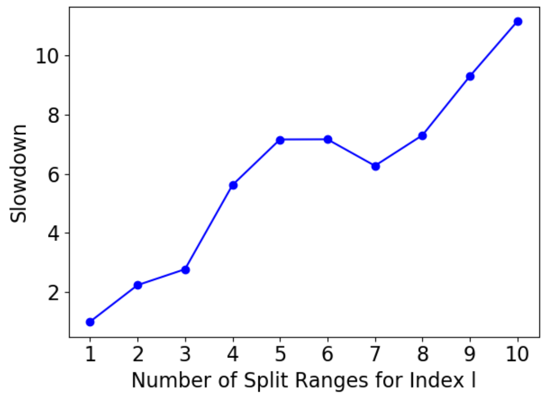


Figure 5-4: Overhead from increasing the number of splits along reduction variable l in SpMV.

linear when the number of split ranges nears 10, but the slowdown for 20 split ranges is 34.5x (compared to 11.2x at 10 split ranges). This isn't quite quadratic, but it's still super-linear. Moreover, the scale of the slowdown is much smaller than SpMV; at 10 split ranges SpMV is 3.3x worse, and at 200 split ranges SpMV is 4.3x worse. Overall, the overhead of splitting along the reduction variable for TTM is super-linear, but sub-quadratic and not as steep as the SpMV slowdown curve.

5.4 Discussion

The strong scaling efficiencies for the kernels we analyzed were generally consistent, with the TTM kernel coming out slightly higher at 20 cores per node but slightly lower at 4 cores per node. If we take the optimal conditions for each kernel, then the SpMV kernel achieves a 33.3x speedup from single-core performance (45.0 sec) by using 8 nodes with 20 cores per node (for an overall strong scaling efficiency of 21%), while the TTM kernel achieves a 42.0x speedup from single-core performance (55.9 sec) by using 7 nodes with 20 cores per node (for an overall strong scaling efficiency of 30%).

As for the performance degradation that occurs past a certain number of nodes, Legion's profiling tool points to a few potential causes. One is the inline mapping performed in the kernel task. When accessing a physical instance that has been mapped inline, the task can potentially yield to the scheduler when accessing the instance initially, even if the requested instance is in memory. This can delay the completion of the kernel task for some time, and potentially significantly increase compute time when compute time becomes small. It also delays the execution of any tasks that depend on this task to complete first.

Unfortunately the inline mapping cannot easily be avoided. Logical regions have a fixed size upon creation, so since the kernel task returns a variably-sized output tensor, the kernel task must create that logical region in its context and return it either through a future or inside another logical region that was provided to the task at launch. To avoid the inline mapping in the next kernel task that uses that logical region, the logical region must be provided in a region requirement to the task when it launches in the parent. But getting a reference to the logical region would require either waiting on the containing future or performing an inline map on the containing logical region. Both of these would stall the parent task, which would be worse than stalling the child task. One potential solution is to have the parent launch a helper task that depends on the containing future or has a region requirement for the containing logical region. The helper task would then launch the child task using the results of the containing future or a physical instance of the containing logical region, which would be guaranteed to be available to the helper task when it runs. This has the downside that the kernel task is no longer able to hide the latency of the logical region mapping with the kernel `assemble()` and `compute()` like it did before. Furthermore, for any kernel tasks that would have waited little time after yielding to resume, the helper task may impose more of a delay.

Another noticeable issue found by the profiling tool is that the starts of kernel tasks that should take place in parallel aren't synchronized. When a group of tasks that can be run in parallel is issued on a node, some of those tasks can start much later than the others, sometimes as long as 1–2 seconds. The discrepancy between start times is so long that the scheduler may execute multiple kernel tasks on the same processor while another processor waits for its first kernel task to execute. It's unclear what's causing these delays in some of the kernel tasks. It could be a delay in task mapping or a delay in other control signals in the Legion runtime. It's also possible there's a logical region dependency that hasn't mapped yet, and the kernel task is waiting on that dependency to resolve. However, in both kernels, the distribution schedules are such that very little tensor data needs to be transferred between nodes. Performance degradation appearing at a certain node count threshold and not a processor threshold supports the idea that this delay is at least in part caused by node-to-node communication issues.

With regards to the overhead analysis for reduction variable splitting, SpMV experienced a steep quadratic slowdown with increasing splits along the reduction variable while TTM had a gentler sub-quadratic slowdown. We expected at least linearity from the slowdown, since more reduction variable split ranges means more tasks that must be executed sequentially, and accordingly any fixed costs of those tasks (e.g. summation into the output tensor, copying output tensor data, Legion overheads) manifest as a linear cost in the number of reduction variable splits. What can lead to super-linearity is if the kernel task execution time increases with more splits, or if the time between kernel tasks (e.g. time spent waiting for data) increases with more splits. Given that the SpMV kernel requires much less work to execute than the TTM kernel, SpMV's increased slowdown with splitting could be due to the time spent computing the kernel being on the scale of other factors which cause delay linear in the number of splits, while the TTM kernel isn't as affected as heavily because its kernel compute time outweighs these factors. Determining what those factors are that cause super-linearity will require further study.

While SuperTaco has shown an ability to scale to 120–140 cores with 20–30% strong-scaling efficiency for various kernels, future work must be done to minimize performance degradation with further scaling and determine how to structure our generated Legion programs to avoid hidden latencies.

Chapter 6

Conclusion and Future Work

In this work, we have demonstrated a code generation technique for creating Legion programs to distribute Taco tensor algebra kernels. The technique is versatile in that it can structure the distributed computation in a variety of ways, which is controllable through a scheduling language. Our scaling analyses showed that our tested kernels achieved significant speedup for both multi-core and multi-node settings, but the scaling efficiency could be improved.

Looking forward, there are three areas of SuperTaco that need focus. The most pressing one is improving scalability by reducing overhead associated with the Legion runtime. In some cases, this may amount to changing what paradigms we use, how we structure tasks, or tweaking environment settings. In other cases though, it may require actively working with the Legion team to come up with a satisfactory solution. One example of this is Legion's garbage collection; if there was a method to mark a physical instance as invalid, we wouldn't have to resort to tensor block eviction, which forces all physical instances to be marked invalid instead of just one and forces tasks to synchronize. However, being able to invalidate a specific physical instance could be potentially unsafe if it's the only physical instance, and determining whether it's the last remaining physical instance in a distributed system could be costly. Finding a tenable middle ground for these kinds of problems will require effort.

The next area is the scheduling policy. While this work introduces a scheduling language, it doesn't formalize any policies of how to choose a distribution schedule in the general case. There are many metrics that could be used to gauge the performance of a policy, such as the amount of data movement incurred or running time. However, we have not developed efficient algorithms or heuristics for choosing schedules, and instead typically reason about it on a case-by-case basis.

The third area is evaluation. The Taco compiler is a very versatile tensor algebra library in that it can compile a wide variety of expressions to C code, and the choice added through distribution schedules means there's now a wide variety of ways to distribute those computations. However, very few of these cases have been tested and evaluated for performance. This is made worse by the fact that performance can depend on the sizes and data patterns of the tensors. Without more evaluations, it will be difficult to gauge optimal policy beyond theoretical models.

The Taco compiler was groundbreaking in its ability to compile general sparse tensor algebra expressions into competitive C code, and the SuperTaco project has the potential to extend those performance gains to a distributed setting. If further development can address Legion overhead, scheduling policy, and evaluation, then in the future SuperTaco may be competitive with hand-optimized distributed kernels.

Bibliography

[1] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Rev.*, vol. 51, no. 3, pp. 455–500, Aug. 2009, ISSN: 0036-1445. DOI: `10.1137/07070111X`. [Online]. Available: `http://dx.doi.org/10.1137/07070111X`.

[2] J. Bennett, C. Elkan, B. Liu, P. Smyth, and D. Tikk, "Kdd cup and workshop 2007," *SIGKDD Explor. Newsl.*, vol. 9, no. 2, pp. 51–52, Dec. 2007, ISSN: 1931-0145. DOI: `10.1145/1345448.1345459`. [Online]. Available: `http://doi.acm.org/10.1145/1345448.1345459`.

[3] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi, "On the evolution of user interaction in facebook," in *Proceedings of the 2Nd ACM Workshop on Online Social Networks*, ser. WOSN '09, Barcelona, Spain: ACM, 2009, pp. 37–42, ISBN: 978-1-60558-445-4. DOI: `10.1145/1592665.1592675`. [Online]. Available: `http://doi.acm.org/10.1145/1592665.1592675`.

[4] J. McAuley and J. Leskovec, "Hidden factors and hidden topics: Understanding rating dimensions with review text," in *Proceedings of the 7th ACM Conference on Recommender Systems*, ser. RecSys '13, Hong Kong, China: ACM, 2013, pp. 165–172, ISBN: 978-1-4503-2409-0. DOI: `10.1145/2507157.2507163`. [Online]. Available: `http://doi.acm.org/10.1145/2507157.2507163`.

[5] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, 77:1–77:29, Oct. 2017, ISSN: 2475-1421. DOI: `10.1145/3133901`. [Online]. Available: `http://doi.acm.org/10.1145/3133901`.

[6] S. Chou, F. Kjolstad, and S. Amarasinghe, "Format abstraction for sparse tensor algebra compilers," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, 123:1–123:30, Oct. 2018, ISSN: 2475-1421. DOI: `10.1145/3276493`. [Online]. Available: `http://doi.acm.org/10.1145/3276493`.

[7] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, Salt Lake City, Utah: IEEE Computer Society Press, 2012, 66:1–66:11, ISBN: 978-1-4673-0804-5. [Online]. Available: `http://dl.acm.org/citation.cfm?id=2388996.2389086`.

[8] S. Hiranandani, K. Kennedy, and C.-W. Tseng, "Compiling fortran d for mimd distributed-memory machines," *Commun. ACM*, vol. 35, no. 8, pp. 66–80, Aug. 1992, ISSN: 0001-0782. DOI: `10.1145/135226.135230`. [Online]. Available: `http://doi.acm.org/10.1145/135226.135230`.

[9] M. E. Bauer, "Legion: Programming distributed heterogeneous architectures with logical regions," PhD thesis, Stanford University, 2014.

[10] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, ser. SC '07, Reno, Nevada: ACM, 2007, 38:1–38:12, ISBN: 978-1-59593-764-3. DOI: `10.1145/1362622.1362674`. [Online]. Available: `http://doi.acm.org/10.1145/1362622.1362674`.

[11] D. Merrill and M. Garland, "Merge-based parallel sparse matrix-vector multiplication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16, Salt Lake City, Utah: IEEE Press, 2016, 58:1–58:12, ISBN: 978-1-4673-8815-3. [Online]. Available: `http://dl.acm.org/citation.cfm?id=3014904.3014982`.

[12] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Modern software tools for scientific computing," in, E. Arge, A. M. Bruaset, and H. P. Langtan-

gen, Eds., Cambridge, MA, USA: Birkhauser Boston Inc., 1997, ch. Efficient Management of Parallelism in Object-oriented Numerical Software Libraries, pp. 163–202, ISBN: 0-8176-3974-8. [Online]. Available: `http://dl.acm.org/citation.cfm?id=266469.266486`.

[13] F. Kjolstad, S. Kamil, and S. P. Amarasinghe, "Automatic generation of sparse tensor kernels with workspaces," *CoRR*, vol. abs/1802.10574, 2018. arXiv: `1802.10574`. [Online]. Available: `http://arxiv.org/abs/1802.10574`.

[14] S. Smith and G. Karypis, "Tensor-matrix products with a compressed sparse tensor," in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA3 '15, Austin, Texas: ACM, 2015, 5:1–5:7, ISBN: 978-1-4503-4001-4. DOI: `10.1145/2833179.2833183`. [Online]. Available: `http://doi.acm.org/10.1145/2833179.2833183`.

[15] O. Kaya and B. Ucar, "High performance parallel algorithms for the tucker decomposition of sparse tensors," *2016 45th International Conference on Parallel Processing (ICPP)*, pp. 103–112, 2016.