Automatic Optimization of Sparse Tensor Algebra Programs

by

Ziheng Wang

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

May 2020

Author .....
Department of Electrical Engineering and Computer Science
May 18, 2020

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

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

Automatic Optimization of Sparse Tensor Algebra Programs

by

Ziheng Wang

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2020, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I attempt to give some guidance on how to automatically optimize programs using a domain-specific-language (DSLs) compiler that exposes a set of scheduling commands. These DSLs have proliferated as of late, including Halide, TACO, Tiramisu and TVM, to name a few. The scheduling commands allow succinct expression of the programmer's desire to perform certain loop transformations, such as strip-mining, tiling, collapsing and parallelization, which the compiler proceeds to carry out. I explore if we can automatically generate schedules with good performance.

The main difficulty in searching for good schedules is the astronomical number of valid schedules for a particular program. I describe a system which generates a list of candidate schedules through a set of modular stages. Different optimization decisions are made at each stage, to trim down the number of schedules considered. I argue that certain sequences of scheduling commands are equivalent in their effect in partitioning the iteration space, and introduce heuristics that limit the number of permutations of variables. I implement these ideas for the open-source TACO system. I demonstrate several orders of magnitude reduction in the effective schedule search space. I also show that for most of the problems considered, we can generate schedules better than or equal to hand-tuned schedules in terms of performance.

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

Acknowledgments

I would like to thank Saman Amarasinghe and Fredrik Kjolstad for all the support and mentorship in the past year. I would also like to thank Ryan Senanayake, Stephen Chou and Changwan Hong for their helpful discussion and insights. Finally I would like to acknowledge the support from all of my family and friends.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, many domain specific languages (DSLs) have been developed to describe scientific or machine learning computations. These DSLs typically target programs that can be expressed as loop nests. For example, the pairwise forces computation in molecular dynamics simulations involve a double for loop over all the atoms. Image filtering algorithms involve 2D stencil computations which can be expressed as a double for loop over the filter elements. Notable recent DSLs include Halide (originally developed for image processing), TACO (sparse tensor algebra), Tiramisu (sparse/dense linear algebra) and TVM (deep learning) [18, 3, 5, 10].

In addition to a language to describe the computation using an algorithmic language, these recent DSLs typically include a way to describe the concrete implementation of the algorithm, which is called a **schedule**. A specification of the computation and the schedule completes the program. The idea is that programmer can change the schedule without impacting the computation, which allows rapid experimentation with different implementations to find the fastest one [17]. The ease of performance engineering is arguably why these DSLs exist. A lot of recent effort has focused on autoscheduling—automatically finding the best schedule for the DSL for a specific computation to get the best performance. The autoscheduler searches through the space of possible schedules to find the best. [13, 1]

A schedule is typically expressed by a sequence of scheduling commands, exposed by the scheduling API of the DSL, for example, unroll, reorder, or parallelize [18,

10]. These scheduling commands effect different loop transformations in the DSL. Many recent DSLs adopt this approach, including Halide, TACO, Tiramisu and TVM. Different systems go to different depths on autoscheduling. TVM for example, needs the user to manually specify the AST (template in TVM parlance). Auto-TVM then proceeds to find the best parameters for this template, i.e. tile size, block size etc [6]. The latest Halide autoscheduler generates this tree using a machine learning approach, trained on billions of fake programs [1].

The autoscheduler could be viewed as a compiler. Whereas a traditional compiler translates a piece of code into a sequence of instructions, the autoscheduler translates a tensor algebra problem into a sequence of these scheduling commands.

Modern compilers are designed with multiple optimization passes. These optimization passes makes the compiler modular and extensible. Each optimization pass produces an intermediate representation of the original piece of code. These intermediate representations contain strictly more detail than the original code – they encode choices made by the compiler at each stage to optimize the performance.

In this thesis, I will present autoscheduler whose design is inspired by the multi-stage heuristics-based optimizations in modern compilers, in contrast with the one-shot machine learning approaches typically employed today. Starting from a problem described in the tensor algebra notation, it makes a sequence of decisions, which incrementally specify more and more of the schedule. The sequence of decision involves how to partition the iteration space, how to reorder the index variables, which variables to parallelize, and finally what values tunable parameters take. At each step in this sequence, there are multiple choices to explore, expanding upon which we get even more choices at the next step, as shown in Figure 1-1a.

The first major stage is the generation of a list of **schedule templates**. A schedule template is a schedule without parameters such as split factors filled in, a.k.a. a loop transformation strategy without parameters. This stage could be divided into two substages: partition and assignment. In the first substage I decide how to cut up the iteration space. The second substage decides how to iterate over and through the chunks. We will go through this in detail later.

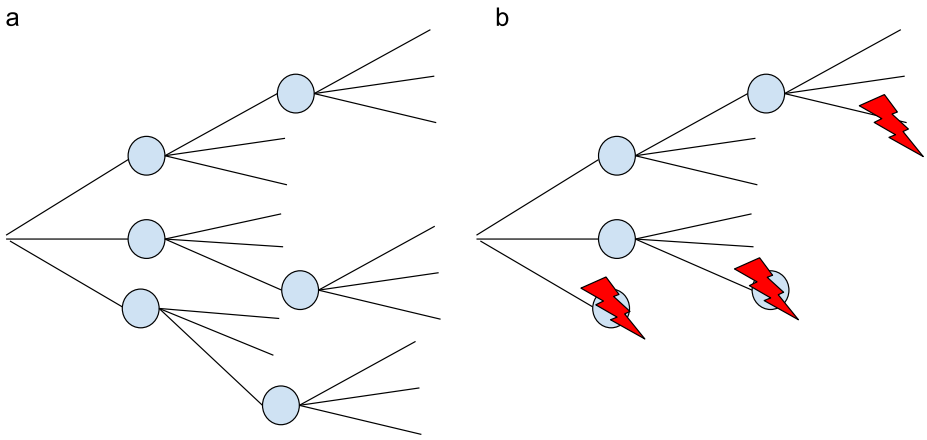

Figure 1-1: a) The autoscheduler has a sequence of stages, where each stage emit possible choices for the next stage to explore further. b) If we restrict some choices at some intermediate stage, it can greatly affect the number of schedules generated in the end. Generally the earlier we restrict choices the greater effect we have.

The second stage is filling in the numbers in each schedule template to generate schedules that can be lowered into code using a DSL. Needless to say, each schedule template can correspond to multiple schedules.

These two stages generate a list of candidate schedules. One now has to search through all the schedules we have generated. There are many approaches in literature, from recent neural network based approaches such as [6] to decades-old approaches like Thompson sampling. Figure 1-2 provides a visual description of the search space.

There is some confusion in the field pertaining to exactly what autoscheduling entails. Some works, for example autoTVM, define autoscheduling to be finding the best parameters for a fixed schedule template [5]. More recent works, [1] define autoscheduling to also include the generation of the schedule template. Similarly, in this work, I aim to both generate the schedule template and the schedule.

The key challenge in autoscheduling is that the list of viable schedules for a particular problem is often astronomically large. As a thought example, let's imagine

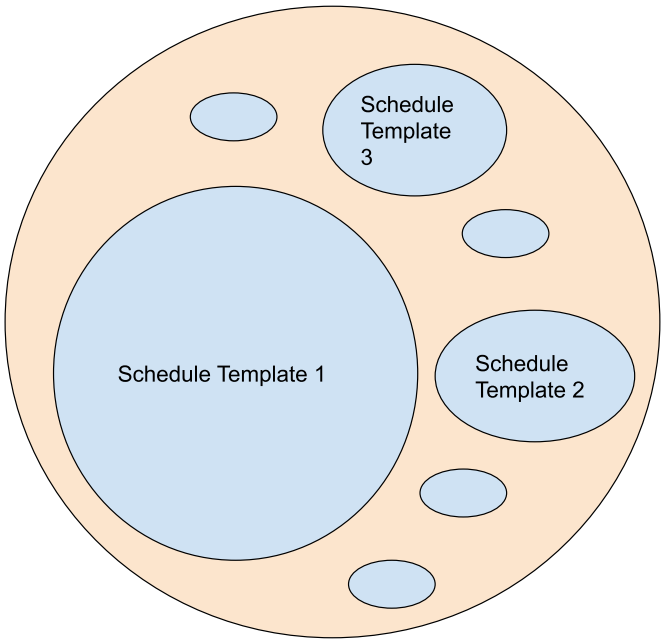

Figure 1-2: A visual illustration of the space of schedules. Each schedule template can generate a family of schedules, by changing the values of the tunable parameters. Different schedule templates can generate families of different sizes. Therefore, eliminating one schedule template might have a substantially larger effect on reducing the search space than eliminating another.

generating schedules for a particular program, say sparse matrix dense vector multiplication: $y(i) = A(i, j) \times x(j)$. This is probably one of the simplest sparse tensor algebra problems. In TACO, there are at least 1000 legal schedule templates for this simple program. each of these schedule templates contains one or more parameters. Assuming that there are at least three choices for each parameter, then each schedule template can give rise to at least three schedules. The search algorithm at the end would have to explore a space of three thousand schedules. If we assume that executing a single schedule takes three seconds, then searching through this entire space would take more than 2 hours. More complicated sparse tensor algebra problems can take significantly longer. One could invest a lot of effort into a good search algorithm that navigates this space efficiently, such as autoTVM or OpenTuner [5, 2]. To complement those approaches, we can also invest some effort into reducing the search space size.

The key to reducing the search space size is restricting the set of optimizations explored at each stage and substage, so that the total number of possible schedules generated at the end is small enough for the search to be feasible, as illustrated in Figure 1-1b. We will design these stages based on an understanding of the computation we are scheduling and the hardware we are targeting. I design the stages to be largely independent of each other. For the total number at the end to be small, the number of choices at each stage should be limited as much as possible. This means we can either generate fewer schedule templates, or we can limit the number of parameter choices for each schedule template. I claim that the first option is a lot more viable than the second option. It is difficult to claim that a set of legal parameter choices is better than another set of legal parameter choices for a particular schedule, assuming no knowledge of the input data and underlying hardware. However, it is relatively easier to claim that a particular schedule template is better than another schedule template, even without the parameter values filled in.

I now justify the counter-intuitive assertion I made at the end of the last paragraph. The schedule template makes qualitative statements about the properties of a schedule, while the actual parameters determine quantitative aspects. For example, the schedule template typically determines if a schedule is load-balanced, whereas the actual parameters might determine the amount of parallelism the schedule exposes. The schedule template determines the overall tiling strategy and iteration order, whereas the actual parameters determine the tile sizes. It is much harder to reason about the attractiveness of quantitative properties of schedules, whereas it's much easier to reason about their qualitative aspects. While quantitative properties can be used to drive the machine learning systems to search the schedule space, simple heuristics can be derived from qualitative properties to drastically reduce the search space size.

Let's now list some characteristics of bad schedule templates that we wish to eliminate. The first type is **illegal** schedule templates that simply do not compile in the DSL. On the grand scheme of things, they are actually not that bad, since if they never compile, we will not have to search parameters for them. However, they can

clutter up the schedule space, making meaningful analysis difficult. The second type is **redundant** schedule templates. This means two, or a group of, schedule templates all generate the same code. This scenario is painfully common in a lot of scheduling languages. Naive autoschedulers for example, could get caught endlessly reordering a couple of variables back and forth. How bad it is depends on how big the redundant group size is, and how large the parameter search space for each of those schedule templates is. The third type is **inefficient** schedule templates, which are almost certainly slower than some other schedule template. How bad they are correlate directly with how big their parameter search spaces are. A fundamentally flawed schedule template that captures a significant proportion of the final schedule search space is like a black hole that even the best search algorithms might have difficulty escaping from. For example, in Figure 1-2, half of the schedule space is derived from schedule template 1. If schedule template 1 is bad, then it would present such a "black hole".

Illegal schedule templates can be removed by following the scheduling language's rules. **Redundant** schedule templates can be removed by establishing functional equivalences between different sequences of schedule commands. These two types of schedule templates are relatively easy to remove. It is much harder, comparatively, to remove **inefficient** schedule templates. To do so, we have to introduce subjective criteria that good schedules probably fulfill, a.k.a. heuristics.

Up to this point, I have been talking in the context of some generic scheduling language and some generic DSL. While I hope the ideas I describe below can apply to many if not all of them, I will focus in particular on the DSL TACO [10]. TACO caters to sparse tensor algebra programs, which can be described by index notations [10]. I refer the unfamiliar reader to the multiple TACO publications for more information [10, 9, 7]. The scheduling API is described in [20].

In the following chapters, I will first describe the different stages of the autoscheduler in sequence (Chapter 2). I will first deal with partitioning dense and sparse iteration spaces into chunks in Sections 2-4. We will see that seemingly different sequences of scheduling commands can all describe virtually identical partition strategies. After

partitioning, we are left with a list of derived index variables from the original index variables. We will then explore how to reorder these derived index variables and parallelize them. We will find that the number of unique choices here are overwhelming even for the simplest of problems. To address this, I design hardware-specific trimming passes to eliminate inefficient reordering and parallelization strategies in Section 5. Finally, we will describe how one can search through all the schedules generated, treating the problem in an hierarchical multi-arm bandit setting in Section 6. In Chapter 3, I will evaluate my approach on three different sparse tensor algebra problems, SpMV, SpMM and MTTKRP on CPU and GPU. I will summarize my approach and compare it to related works in Chapter 4.

My specific contributions in this thesis are:

1. described heuristics to limit the number of strategies to partition sparse and dense iteration spaces into chunks

2. described heuristics to limit the number of ways to iterate through and parallelize the chunks

3. constructed an automated system that implements these heuristics to automatically write schedules for sparse tensor algebra problems in TACO.

Chapter 2

Scheduling Framework

2.1 Background

Before we start, let's discuss the TACO scheduling API we have at our disposal. For more detail, the reader is referred to [19, 20]. As mentioned before, TACO is primarily intended for sparse tensor algebra, and has scheduling commands to transform dense and sparse loops. A sparse tensor algebra problem can be expressed in **index notation**, e.g. $x(i) = A(i, j) \times y(j)$ for sparse matrix dense vector multiplication, where A is a matrix in the CSR format. i and j are called **index variables**. The ranges of the index variables define the **iteration space** of the program. They can either be dense, like i, or compressed, like j. We will discuss the subtleties involved in scheduling compressed loops in more detail later on. The important commands that we need to know are:

1. *reoder*: swaps the order of iteration over two or more directly nested index variables (permutes the order of for loops).

2. *split*: splits (strip-mines) an index variable into two nested index variables, where the size of the inner index variable is constant. We call the two nested index variables **derived index variables**.

3. *divide*: same as split, except that it's the size of the outer index variable that is constant instead of the inner one.

4. $fuse$: collapses two directly nested index variables, resulting in a new fused index variable that iterates over the product of the coordinates of the fused index variables.

5. *parallelize*: parallelizes a loop over a parallel unit, e.g. OpenMP thread, GPU block or GPU thread.

Compressed loops can be scheduled in the coordinate space or the position space, via the *pos* scheduling command. We will talk about position space later.

Now that we have introduced the terminology and listed the scheduling commands, this chapter will proceed to examine how we can use them to partition dense and sparse iteration spaces. We will largely explore the partitioning from a geometrical perspective, which will assist us in weeding out redundant schedules which express the same partitioning strategy with different sequences of scheduling commands.

2.2 The Hypercube Perspective

Let's take the example of matrix multiplication. The index notation is $C(i,k) = A(i,j) \times B(j,k)$. We can represent the iteration space with a cube, as illustrated in Figure 2-1. (Instead of a cube which suggest a solid object, perhaps I should emphasize that it's a 3D grid of points. In the following discussion, **"cube" is implicitly understood to be this 3D grid of points**.) Each edge corresponds to an index variable in the problem. The first matrix, which is sparse, is represented by the face with the edges i and j, where j is the sparse dimension. The second matrix, which is dense, is represented by the face with edges j and k. Each point inside this cube represents an individual multiplication. Since the problem is sparse, we do not need to iterate over every point inside the cube. Problems with more than 3 index variables can natually be represented by higher order hypercubes. This is a very polyhedral view of the world. A lot of disciples of this doctrine have written much better and possibly more lucid explanations of this perspective [27, 4]. This polyhedral perspective allows to visualize what partition strategies described by different sequences

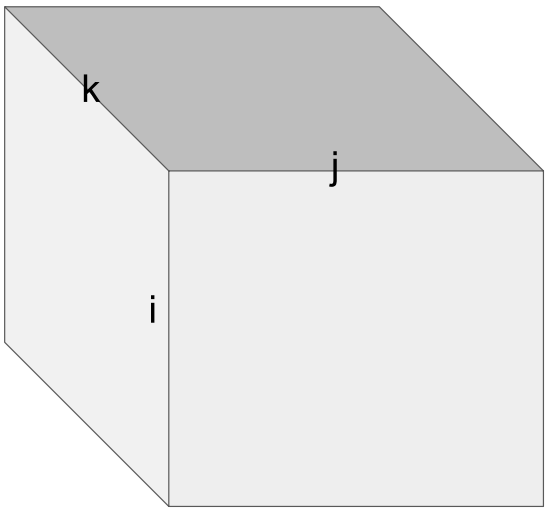

Figure 2-1: The hypercube corresponding to matrix multiplication $C(i,k) = A(i,j) \times B(j,k)$. If we discretize the edges, then this hypercube defines a grid of points, which correspond to the multiplication operations. This is exactly the same as in the Polyhedral model. Some points might not have to be visited if they correspond to multiplication by zero in the sparse case.

of scheduling commands do to iteration space, which will help us in weeding out redundant schedules.

We are faced with the task of taking this cube, which describes the sparse tensor algebra problem, and specifying how it is to be actually carried out on our hardware, be it CPU or GPU or potentially some FPGA accelerator. What does this entail? Assuming we have some parallel hardware device, then we have to specify for each point in this cube, two important attributes: **which** processor the point executes on and **when** it is to be executed on said processor among all the points assigned to it. The problem of autoscheduling can be thought of specifying these two things, for all the points in the hypercube.

It's important to note that the polyhedral model cares greatly about dependencies between execution instances, i.e. points inside the cube. If some points must occur after other points for example, then the number of transformations one can make

to the cube are greatly restricted. Fortunately, in tensor algebra problems, all of the execution instances are independent of one another. All of the multiplications in the problem can occur in any order or all at once, before the reductions happen. The reductions can also be done in any order. We thus do not have to worry about dependency analyses here, though it can potentially be introduced into my approach if necessary.

2.3 Dense Partition

In this section, we show how to think about partitioning a dense iteration space with the commands $fuse$, *split* and *divide*. We show that the command $fuse$ is effectively redundant, and we can proceed to partition each index variable independently using *split* and *divide*.

Before we proceed further, let's attempt to establish a formalism that describes the splitting we are going to be doing. Let us assume that the problem we are interested in have n index variables, which are called $X_1, X_2, ..., X_n$. Let's use $|X_i|$ to denote the maximum value that index variable can take (iteration bound). For a moment, let's assume that we cannot fuse dimensions. Then our job is to specify partitions $\pi_1.\pi_2...\pi_n$ to divide up the index variables. A partition is a function defined as $\pi_i : \{0, 1...|X_i|\} \rightarrow \{0, 1...d_i\}$, where we divide the range of index variable X_i into d_i non-overlapping subsets. Henceforth, we will abbreviate $\{0, 1...x\}$ as $range(x)$. Because the domain of π_i is discrete, we can also describe it as a sequence composed of its action on each element in $range(|X_i|)$: $(\pi_i(0), \pi_i(1)..., \pi_i(|X_i|))$.

The partitions on each edge induce a straightforward partition of the hypercube $\pi : \prod_{i=1}^{n} range(X_i) \rightarrow \prod_{i=1}^{n} range(d_i)$ through projection, as shown in Figure 2-2. This partitions the hypercube into $\prod_{i=1}^{n} d_i$ partitions. We can assign an n long tuple to each point in the cube specifying which partition it is in.

Let us focus on one single index variable, X_i and think about what does π_i have to be. In particular, we'd like to gain some intuition regarding what π_i should look like. Naively, of course, the number of ways to partition a set of $|X_i|$ numbers is truly

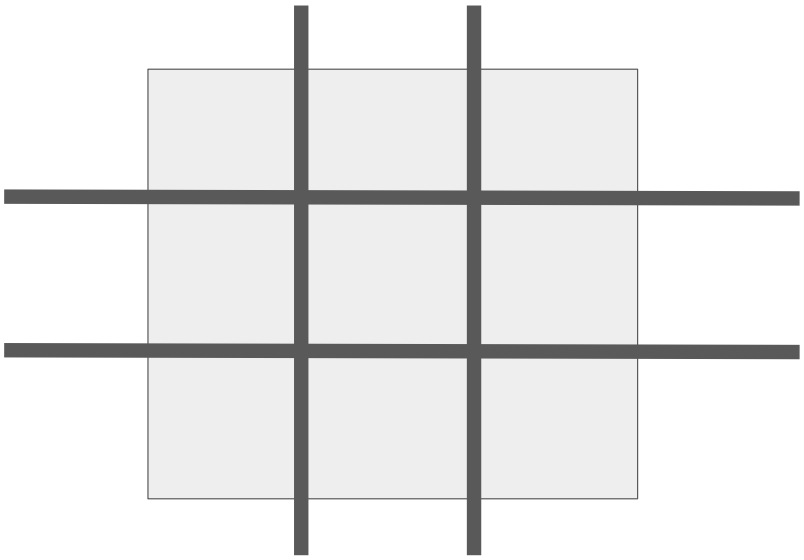

Figure 2-2: How we can induce a partition in 2 dimensions by partitioning each 1 dimensional-edge: $\pi(x_1, x_2) = (\pi(x_1), \pi(x_2))$.Higher dimensions are analogous.

massive. We'd like to restrict our attention to partitions that are 1) meaningful and likely to be useful and 2) achievable using a scheduling language.

Let's first consider the second condition, because it's much easier to reason about. What tools do we have to construct such a partition? As alluded to before we cannot just arbitrarily specify this function. Let's consider a scheduling function with two commands: split and divide. Split is a command that divides a loop into fixed sized inner loops, whereas divide divides a loop into a fixed number of outer loops. For more information, please refer to [20]. For example, $split(X_i, x_0, x_1, 4)$ will produce the following code (assuming that $|X_i|$ is divisible by 4):

```
for x0 = 0 ... |Xi| / 4
    for x1 = 0 ... 4
```

Whereas $divide(X_i, x_0, x_1, 4)$ will produce the following code:

```
for x0 = 0 ... 4
    for x1 = 0 .. |Xi| / 4
```

It is straightforward to see that the first command will produce the following partition on $range(X_i)$:

$(1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3...|X_i|/4, |X_i|/4, |X_i|/4, |X_i|/4)$

It's important to see that the first command can also be used to produce another partition:

$(1, 2, 3, 4, 1, 2, 3, 4...1, 2, 3, 4)$.

Let's call x_0 and x_1 **derived** index variables. The first partition implicitly assumes that we index the partitions with x_0. However, we could also index the chunks with x_1. If we insist that in a two-level loop, the outer loop iterates over the chunks and the inner loop iterates within a chunk, then this could be described by $split(X_i, x_0, x_1, 4).reorder(x_1, x_0)$. This reorder can always be done, since the iteration bounds of the two variables being reordered are completely independent.

This partition illustrates that the partition doesn't necessarily have to be contiguous. What does the *divide* command produce?

$(1, 1, 1...1, 2, 2, 2...2, 3, 3, 3...3, 4, 4, 4...4)$ and

$(1, 2, ...|X_i|/4, 1, 2, ...|X_i|/4, 1, 2, ...|X_i|/4, 1, 2, ...|X_i|/4,)$.

In short, the *split* command can produce a **data dependent** number of **fixed** size **contiguous** chunks indexed by x_0 and a **fixed** number of **data dependent** sized **noncontiguous** chunks indexed by x_1. Noncontigous means that the elements in the chunk are not adjacent in memory, which could result in unfavorable spatial locality properties. The *divide* command can produce a data dependent number of fixed size noncontiguous chunks indexed by x_1 and a fixed number of data dependent sized contiguous chunks indexed by x_0. I advise the reader to spend a moment reflecting upon this.

To summarize, a partition can be characterized by three things: 1) whether or not the number of disjoint sets, d_i, is statically known. 2) whether or not the size of each disjoint set is statically known. 3) whether or not the disjoint sets are contiguous. All three describe important qualitative aspects of the schedule, and will play an important role in heuristics used to eliminate inefficient schedules. Let's give a preview here. In some parallel models, there are a limited number of parallel units available.

If we are to assign each disjoint set in the partition to a parallel unit, then we need to know the number of disjoint sets statically to ensure that there is sufficient parallel resources. Sometimes, parallel units strongly prefer to operate over a contiguous set of points in the hypercube, e.g. due to memory locality. Then we would prefer the disjoint sets to be contiguous.

Of course, all this discussion about data dependent loop bounds assumes that we do not know what $|X_i|$ is at compile time. If we do know, then it's evident that these two commands or more or less equivalent to each other. Both can produce either a fixed number of fixed size noncontiguous chunks or a fixed number of fixed size contiguous chunks. Some scheduling languages allow for user input of this kind of information. For example, we could use the *bound* command in TACO to explicitly specify what the value of a data dependent number is.

Obviously, there are other forms π_i could take, for example $(1, 1, 2, 2, 3, 3, 1, 1, 2, 2, 3, 3...)$. This partition strategy is achievable through repeatedly applying the *split* command. In fact, we can apply these four partition strategies above recursively and without end in an infinite loop to effect the death of any autoscheduler.

To simplify our discussion, let's restrict our attention to only these four partition strategies for now. We also include a fifth strategy, which just leaves the index variable as is. While this could be expressed as $split(X_i, x_0, x_1, 1)$, writing it this way produces overhead in the resulting code and could lead to overcounting the number of schedules in the end. This point will be elaborated on further later on. It is straightforward now to imagine that if each dimension of this hypercube is split in one of these five strategies, we could project the partitions to form a partition of the hypercube itself.

2.3.1 Addressing Fuse

Now let's think about the $fuse$ command. What does this do? Let's imagine that we can fuse some subsets of the index variables $X_1, X_2, ..., X_n$. Does that increase the repertoire of partitions of the hypercube at our disposal?

To start, let's imagine we can only fuse the first two axes, X_1 and X_2, to get a

new axis F. Now we can apply our four partition types to this axis F. We realize that none of the strategies actually produce a partition that is substantially different from what we can obtain from just partitioning the two axes independently. How so? Let's imagine the two partitions produced by *split*. Consider $split(F, x_0, x_1, K)$. To simplify our analysis, let's assume that $|X_1|$ and $|X_2|$ and K are all powers of two. Let's just consider the case where the chunks are contiguous.

Then there are two cases. The first case is if $K \leq |X_2|$. In this case, we realize that the resulting partition pattern can be achieved without fuse, by splitting the two axes separately and considering their projection, i.e. $split(X_1, x_{11}, x_{12}, 1)$. $split(X_2, x_{21}, x_{22}, K)$. Note that the projection would create a 2-D array of chunks indexed by x_0 and x_2, whereas splitting F would have resulted in a 1-D array of chunks, indexed by x_0. This is just a technical difficulty that could be resolved post-facto by fusing x_0 and x_2 in the former case. The second case is if $K|X_2|$. In this case, the equivalent operation is simply $split(X_1, x_{11}, x_{12}, K/|X_2|)$. An example illustration is shown in Figure 2-3. The argument for the case where the chunks are noncontiguous is exactly analogous. Note we could argue that we do not know the value of $|X_2|$ at compile time, thus there is no way to produce the equivalent command statically. However, as mentioned in Chapter One, the autoscheduler is preoccupied with producing a list of correct schedule **templates**, without the exact parameters filled in. We can assume that later, we will perform exhaustive autotuning on these parameters. As a result, though we cannot determine the value of $K/|X_2|$ statically, we can assume it will be explored in the autotuning process.

Now let's consider $divide(F, x_0, x_1, K)$. Again let's consider the contiguous chunk case. Again there are two cases. The first case is if $K \leq |X_1|$. Then in this case we realize that the command is exactly equivalent to $divide(X_1, x_0, x_1, K)$. The second case is if $K > X_1$. Then the command is equivalent to the following sequence of commands $split(X_1, x_{11}, x_{12}, 1).divide(X_2, x_{21}, x_{22}, K/|X_1|)$, followed by fusing x_{11} and x_{21}. Similarly, the argument for the noncontiguous case is analogous.

For the fifth partition strategy where we don't do anything to the fused variable, the fuse is then not particularly useful. In fact, it is probably harmful, because you

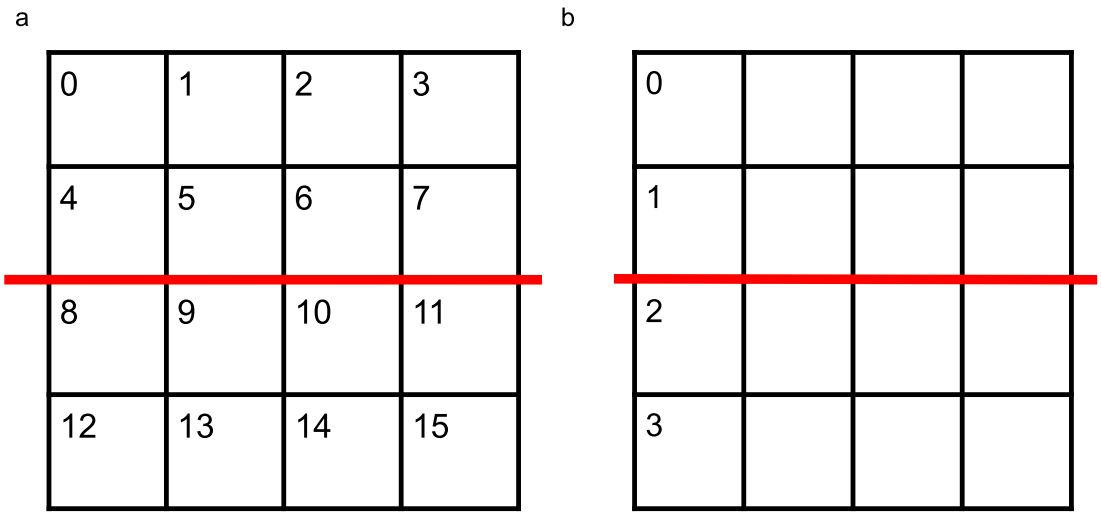

Figure 2-3: a) Splitting after fusing X_0 and X_1. b) Splitting just one axis and projecting this partition across the other one. We see that splitting the fused coordinate space of size 16 is equivalent to splitting just along one axis of size 4, and projecting that split across the other axis.

have to use the remainder and integer division operations to recover the original indices, which tend to be expensive on modern architectures.

We have just proved a rather significant result: in the dense case where things are powers of two, it is futile to consider fusion of 2 dimensions for expanding the number of ways to partition the 2-D iteration space. At most, we should only fuse the variables that are being used to index the chunks. In general, if things are not powers of two, this result will not hold exactly. However, one could show with some more complicated arithmetic that the sizes of the chunks in partitions of the fused variable are within a reasonable neighborhood of the sizes of the chunks achievable by partitioning both index variables independently. Let's ignore this for the sa, and write down the following theorem. The precise formal language is not as important as the intuition in the proof I just described.

Theorem 1 *Assuming $|X_1|, |X_2|$ are powers of 2. The set of partitions that can be achieved by fusing X_1 and X_2 and applying one of the five aforementioned strategies*

with K a power of 2 is contained in the set of partitions that can be achieved by partitioning X_1 and X_2 independently using one of the five strategies with K a power of 2, and projecting the result.

Following this theorem, we can use induction to prove the corollary, which asserts that $fuse$ does not need to be considered for all n dimensions.

Corollary 1.1 *Assuming $|X_1|, |X_2|...|X_n|$ are powers of 2. The set of partitions that can be achieved by fusing $X_1, X_2...X_n$ and applying one of the five aforementioned strategies is contained in the set of partitions that can be achieved by partitioning $X_1, X_2...X_n$ independently using one of the five strategies, and projecting the result.*

To prove this, we will use induction. The base case has been proven in Theorem 1. Let's assume this corollary holds for $X_1, X_2, ...X_{n-1}$. We see that $fuse(X_1, X_2, ...X_n)$ can be broken down to be the fusion of two index variables $fuse(X_1, ...X_{n-1})$ and X_n. By Theorem 1 all partitions on $fuse(X_1, X_2, ...X_n)$ can be expressed by projections of partitions on $fuse(X_1, ...X_{n-1})$ and X_n. But by the induction hypothesis, partitions of $fuse(X_1, ...X_{n-1})$ can be expressed as projections of partitions on $X_1, X_2, ...X_{n-1}$. Thus, all partitions on $fuse(X_1, X_2, ...X_n)$ can be expressed as projections of partitions on $X_1, X_2, ...X_n$.

Corollary 1.1 provides immense value to thinking about autoscheduling. We realize that partitioning the hypercube of a dense iteration space can be done one axis at a time without any interactions between them. This also suggests we can partition it in any order we see fit, since there is no dependence in the partitions specified for each independent index variable. Note that the corollary only establishes that the all partitions resulting from fusion can be produced from partitioning the index variables separately. It does not assert the opposite. In fact the opposite is not true: simply fusing all the index variables and partitioning that fused result will result in a loss of expressive power.

What does this suggest about the scheduling commands that we should issue? We see that for each index variable, we can use *split* or *divide*, and optionally reorder the resulting two variables. The analysis thus far has suggested that for $X_1, X_2, X_3, ..., X_n$,

we should first partition each of them independently, producing up to $2n$ variables, up to 2 from each one of the original index variable. Whereas reordering the 2 variables resulting from partitioning one particular axis results in different partition strategies, we can actually reorder all $2n$ variables thus produced. In the dense case, there are no external restrictions on the order of iterations of the variables, since their iteration ranges do not depend on one another.

We might wish not to split a variable and easily repeat the following analysis with fewer than $2n$ variables. Then we will have to consider potentially up to 2^n cases, where each variable could be split or not. This sounds unmanageable, but in reality n is typically very small, so this is okay. The astute reader will note that we can encapsulate "not splitting" an index variable by splitting it with split factor 1. This suggests that if we allow the search process to explore split factor 1, then we can just split all the index variables in a single case, and then let the search process decide if some index variables shouldn't be split after all. It seems like we should just produce a single schedule template with a bunch of split factors to fill in, and let the search process do its job, vs. produce a bunch of schedule templates with potentially fewer split factors to fill in, and search each of them.

While the latter strategy appears more complicated, it actually reduces the final search space. This is because if we allow a split factor of 1, then we are effectively introducing a useless loop into the schedule. However, we had already considered the set of valid permutations containing this loop, not knowing that it is useless. As a result, we will explore different permutations where the only difference is where this useless loop is executed in regards to other loops, and there is no difference. In short, some schedule templates might be redundant under certain parameter choices. This tradeoff is depicted visually in Figure 2-4.

The next question to be answered is how to globally reorder the up to $2n$ variables generated, and which of them to parallelize over. We will discuss that momentarily. Let's first extend the above analysis to sparse iteration spaces.

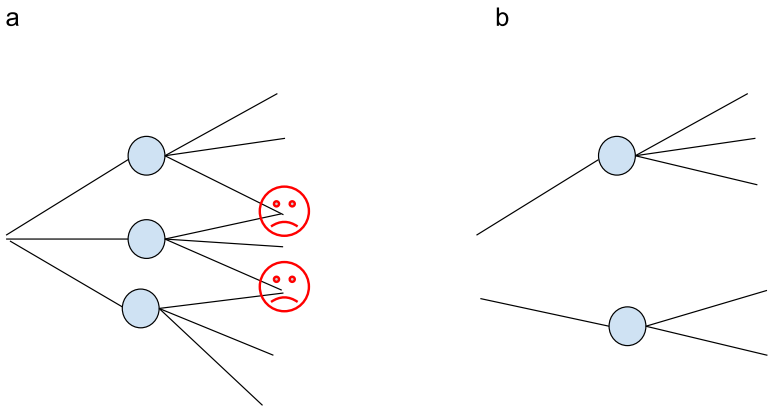

Figure 2-4: A comparison of two strategies. a) less choices at earlier stages could lead to potential redundant schedules in the end while b) more choices at earlier stages could actually lead to fewer viable schedules in the end.

2.4 Sparse Partition

In the world of sparse linear algebra, where holes might exist in the cube, partitioning the cube such that each partition has the same number of points is an unattainable luxury. Cutting the cube like we did above for the dense case is now called scheduling in *coordinate space*. We can divide each edge of the cube evenly, and hope that the the holes are evenly distributed in this cube. Then, each slice of the cube will roughly have the same number of holes, or the same number of points for us to process. However, this usually doesn't happen in sparse tensor algebra problems in science and engineering because the sparsity pattern is often highly structured.

We can get a bit creative, and perform cuts in the *position space* of an edge. What is a position space? It is described in detail in [20]. Briefly, you are partitioning the nonzeros of a dimension, guaranteeing that each partition has the same number of nonzeros. This kind of cutting has a lot of subtleties, an important one of which is that it might produce loops with bounds that depend on the iteration of another

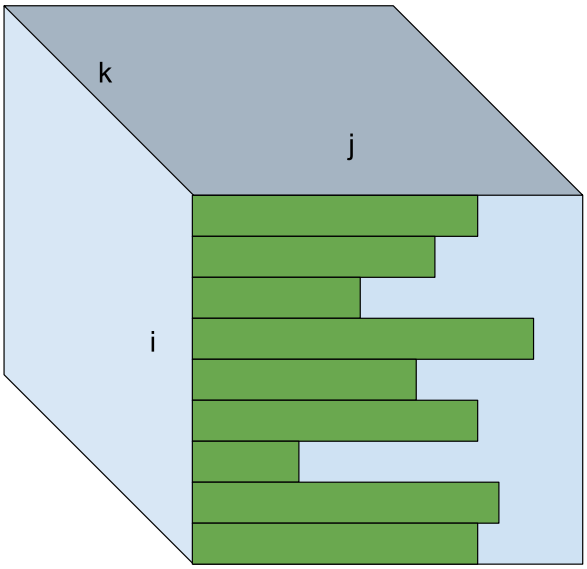

Figure 2-5: What the hypercube looks like is one of the dimensions, j is sparse, with corresponding dimension i.

loop. For example, if $A(i, j)$ is a sparse matrix in the CSR format, then iterating through j's position space means iterating through the nonzero columns. However the number of nonzeros columns change depending on the row, so we have the iterate over j afer i. The bound of the loop over j changes for each iteration of the loop over i, since there are different number of nonzeros in each row. This kind of loop bound is not merely a data dependent bound as we had seen in the dense case. In fact, this kind of loop bound imposes a fundamental ordering among the iteration variables, stemming from their ordering in the given data structure, which is absent in the dense case. This introduces quite a bit of complexity to our hypercube picture, as seen in Figure 2-5.

Indeed, let's return to our index variables, $X_1, X_2, ..., X_n$. For simplicity, let's consider just two of them, X_1 and X_2. Let's say that X_2 is sparse, and X_1 is the **corresponding dimension**, which is dense. Note that all sparse dimensions need to have a corresponding dimension in the same tensor that provides the sparse dimension with indexing information. For example in CSR, we need the dense row offsets array

to index into the compressed column indices array. The corresponding dimension could also be sparse, as in DCSR. In the original TACO formulation, one would say the corresponding dimension's node has an edge leading to this sparse dimension in the iteration graph [10].

We shall see that this limits the partitions we could do on $S = range(|X_1|) \times range(|X_2|)$. Let's keep on considering the case where X_1 is dense and X_2 is sparse. We will consider the other two cases (sparse-sparse, sparse-dense) later. Note again, that this assumes that X_1 and X_2 belong to the same tensor. We will briefly discuss the case where they belong to different tensors later. In short, when we encounter a pair X_1, X_2, we could classify them as belonging in 8 categories, depending on three Boolean flags: if X_1 is sparse, if X_2 is sparse and if X_1 and X_2 belong to the same tensor. Let's consider now one of the eight categories (dense, sparse, same), which is by far the most intricate case, here.

Before when both index variables are dense, we mentioned that there are four partition strategies for each index variable. Since we consider the projection of partitions on S, there are 16 ways to partition S. How many ways are there now?

Well of course, we could continue cutting in coordinate space and recover the previous 16 ways. However this is probably not a great thing to do, since the chunks might be very **load-imbalanced**. In the worst case, imagine all the nonzeros clusters around the start of X_2. Then a lot of chunks in the latter part of X_2 would be completely empty. In addition, due to the compressed storage format of X_2, partitioning the coordinate space means that when a parallel unit starts to iterate over a chunk, it would need to locate where to start in memory, since the memory is laid out in position space. This involves a binary search operation which could bring severe overhead. Again, more details are in [20].

It is important to note that this deficiency does not need to exist. If we imagine the scenario where the sparse matrix values are quite evenly distributed, such as in the cases encountered in deep learning for example, then this splitting strategy, or slight variants of it, could be perfect. The problem that arises from locating into the sparse matrix could also be solved if the sparse matrix is stored in some other

format. For example, these indices could be precomputed or quickly estimated from the sparse matrix data entries, if we know the distribution of the nonzero values. In short, we could proceed to schedule the problem as in the dense case, and simply take away the unnecessary work when lowering it into code.

However, the matrices we encounter in scientific computing typically do not have an even distribution of nonzeros. They are also typically not known statically. As a result, we have to deal with storage formats, the most efficient of which tend to store nonzeros in contiguous position space.

There are then in principle two ways to iterate over X_1 and X_2 efficiently. The first way is if we iterate over X_1 and then iterate over the position space of X_2, which we will denote by $pos(X_2)$. The second way is if we fuse X_1 and X_2, and iterate over the position space of the fused index variable. They roughly correspond to the following:

```
for i = 0 .. |X1|
    for j = start[i] ... start[i + 1]
```

and

```
i = 0
for j = 0 ... start[|X1|]
    while(start[i] < j)
        i = i + 1
```

Let's consider the first way first, and see what are the splitting strategies we can use. We are now splitting X_1 and $pos(X_2)$. We realize that we can partition $pos(X_2)$ in much the same way as we can partition a dense loop, since it's just a for loop, albeit with bounds that depend on x_1. This suggests that if we do the following: $split(X_1, x_0, x_1).pos(X_2).split(X_2, x_2, x_3)$ then we must iterate over x_2 only after we have iterated over both x_0 and x_1 and can calculate what the value of i is (where we are in X_1). This means that after splitting the two index variables to get four variables, we can no longer freely reorder them globally. This is good, because this will limit the number of permutations we need to consider!

It is important to note that if we split X_1 and X_2 separately and project the partitions to get a partition of S, the chunks might not be load balanced, i.e. they might have a different number of nonzero elements. This would occur if we'd split the position space of X_2 into chunks whose sizes are data-dependent, either by *split* or *divide*. Unlike in the dense case, the data-dependent no longer means $|X_2|$. $|X_2|$ in position space depends on which iteration we are in X_1! Now if the position space of X_2 is partitioned into fixed size chunks, the chunks will be load balanced. But unfortunately, as we will see, these partition strategies are not very amenable to parallelization so might get filtered out later on.

The alternative to splitting X_2 in position space is fusing X_1 and X_2 and splitting the fused result. We can split the for loop that results from the fusion in the same four partition strategies that we have been using all along. Splitting it this way has a couple benefits. The first is that all the chunks are guaranteed to be load balanced. The second is that instead of producing 4 variables, it produces only 2, making our autoscheduling job a lot easier! (Or if you're manually writing a schedule, it's much easier to figure out how to order 2 variables than 4.)

Let's now remind ourselves of the assumption we made, we have been considering X_1 dense, X_2 sparse and X_1 and X_2 belonging to the same tensor. What happens in the other seven cases? We've already considered the two cases where X_1 and X_2 are both dense. There remains five cases, where at least one of them is sparse. Now if they belong to different tensors, then fusing is hopeless. We could still partition either (or both) sparse dimensions in its (their) position space. However, the ordering constraint will now be with respect to the corresponding dimension in their respective tensors. There remains two cases: (sparse, dense, same) and (sparse, sparse, same). We don't have much to say about the latter: it's easy to see that the two strategies above for (dense, sparse, same) would apply. What about (sparse, dense, same)? I argue that $fuse$ here is futile. The reason is that the partition it induces on S is too similar to what you could obtain by just partitioning the index variables independently. Perhaps a picture here should suffice: Figure 2-6. This situation is summarized in Table 2.1.

Having decided what to do when there are two index variables, let us think about

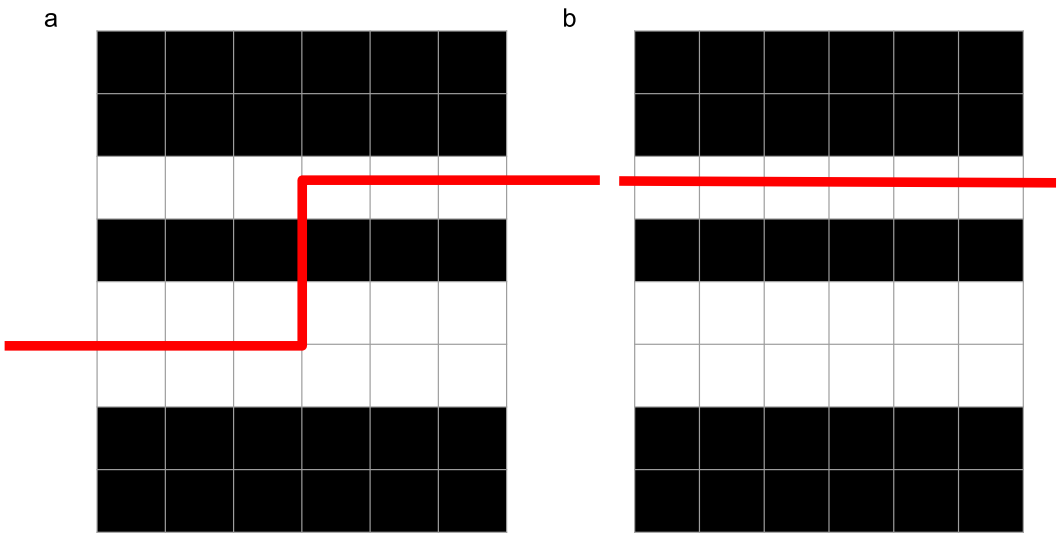

Figure 2-6: This is what a compressed-dense iteration space looks like. a) the result from splitting after fuse. b) the result if just splitting the compressed dimension. As we can see, it's quite similar, up to some difference bounded by the size of the dense dimension. If the compressed dimension has an even number of nonzeros, then it would've been exactly the same.

the case when there can be n of them, potentially belong to k different tensors. Let's first think about $fuse$. If we are going to fuse index variables, which ones we are going to fuse? Well, we should start by picking one of the k tensors with compressed dimensions. Then, we can potentially fuse some of the dimensions of that tensor. Let's imagine that this tensor has m dimensions, which could be a sequence of compressed or dense dimensions. Our analysis for the two dimensional case told us that fusion is only useful if we are fusing a sparse dimension directly into its corresponding dimension. Of course, the corresponding dimension might be sparse, and we could choose to fuse that into its own corresponding dimension! This quite severely limits the fusion that we are allowed to do inside a sparse tensor.

In addition, whichever group of dimensions we would like to fuse, it must end with a compressed dimension. If it ends with one or several dense dimensions, then the partitions that would result can be roughly produced by fusing until the last

Table 2.1: Applicability $fuse$ on X_1 and X_2 depending on their formats and if they belong to the same tensor.

Same/Different Tensors	X_1 format	X_2 format	$fuse$?
Same	Dense	Dense	No
Same	Dense	Sparse	Yes
Same	Sparse	Dense	No
Same	Sparse	Sparse	Yes
Different	Dense	Dense	No
Different	Dense	Sparse	No
Different	Sparse	Dense	No
Different	Sparse	Sparse	No

compressed dimension before the last dense dimensions and partitioning that fused variable and the dense dimensions separately. Of course, we could fuse multiple groups of variables in a single tensor. After we are done selecting a fusion strategy for a single tensor, we would take a look at what index variables are left unfused in the overall problem, select another tensor for fusion opportunities, and carry on.

This seems very complicated, but for lower-dimensional tensors we are likely to encounter in practice it's quite trivial. For example, let's consider SpMM: $C(i,k) = A(i,j) \times B(j,k)$, where A is sparse CSR. Then we can only fuse i and j. In MTTKRP: $A(i,j) = B(i,k,l) \times C(k,j) \times D(l,j)$ where k and l are sparse, we can either fuse k and l, or fuse i, k and l.

After the fusion strategy has been determined, we can proceed to split the index variables (replacing the variables being fused by a new variable) individually, each with the five partition strategies, just like what we did in the two dimensional case. Then, we will have to consider the allowed permutations among the resulting variables and pick groups of them for parallelization, just as in the dense.

2.5 Reordering and Parallelism

We now proceed to answer the most important question after the partitioning has been settled: in what order are we going to iterate over the derived variables. Instead of thinking about it in terms of functions on the range of the iteration variable and

the geometry of the hypercube, it is more expedient to think about it in terms of the order of the for loops in the generated code.

The fact that we can globally reorder all $2n$ split variables suggests that we should not locally reorder after splitting each variable to effect the two different partition strategies associated with either the *split* or *divide* command. We can just defer the reordering step until after we have partitioned all the axes, since we will always be able to express whatever local reorder we intended with a global reorder later. A global ordering of the $2n$ variables has an implied order for the two variables resulting from each index variable, which can determine if the partition of that index variable is contiguous or not. Note that in the global ordering, the two variables resulting from a index variable may not even end up next to each other. Nor might they be ordered in a way that suggests we are iterating over chunks in the projected partition.

For example, let's consider the case where we split three index variables X_1, X_2, X_3 into six variables$x_{11}, x_{12}, x_{21}, x_{22}, x_{31}, x_{32}$, where x_{11} and x_{12} result from splitting X_1, etc. Let's assume we use the first type of partition for all three axes, i.e. $split(X_1, x_{11}, x_{12}, K)$. Now most performance engineers would recognize that we are performing cache blocking, and put x_{12}, x_{22}, x_{32}, the inner loops of the tile, after x_{11}, x_{21}, x_{31}. For example, we could order the six variables as $x_{11}, x_{21}, x_{31}, x_{12}, x_{22}, x_{32}$. This is the most straightforward way to reflect the projected partition of the hypercube (which is just a cube here) – we have divided into chunks of smaller cubes by projecting the partitions on the edges, and we are iterating over the smaller cubes.

However, it is important to see that we don't have to reorder that way. An equally valid, and potentially equally performant way of reordering might be $x_{12}, x_{11}, x_{21}, x_{31}$, x_{22}, x_{32}. In this case we are going to iterate over X_1 fully before iterating over X_2 and X_3 in a blocked fashion. At least, this is a permutation that the autoscheduler should consider, and not just throw away.

The fact that the derived index variables can be globally reordered raises the interesting question of whether or not we should have categorized the two different partitions from either *split* or *divide* as a single partition. In particular, why bother ourselves with thinking about the difference between the two different partitions that

result from *split* or *divide*? Assume we have $split(X_1, x_1, x_2, K)$. Then the first partition have contiguous chunks that are indexed by x_1, whereas the second partition has noncontiguous chunks that are indexed by x_2. But this doesn't appear to matter, since the fact that the chunks are iterated over by different index variables is not reflected in the algorithm we described in the last paragraph.

This brings us to the next step of the problem: parallelism. What this means in practice, is that we would have to pick one out of the up to $2n$ variables, and parallelize it. We could also fuse several variables, and parallelize the fused result. We decreed in the very beginning that after we end up with chunks of the hypercube from projecting the partitions on each index variable, we will assign chunks to parallel units. Each parallel unit could be allotted multiple chunks, but a chunk cannot be divided across parallel units. This means that we must parallelize the index variables that are being used to index the chunks, i.e. x_{11} in the first partition strategy and x_{12} in the second partition strategy assigned to *split*. We don't have to parallelize all of them, i.e. assign a chunk its own parallel unit, but in the extreme case we can. Assuming we partitioned each of the n index variables, then we are left with n variables we can parallelize over and another n variables which we cannot. Let's call the n variables which we can parallelize over **parallelizable** variables.

At some point, we would have to choose what are the variables that we can parallelize. By differentiating the two partitions resulting from *split*, I am just making this decision at the partition stage. One could also presumably make the decision at this stage. This was just a presentation choice. The reader can think about this any way they prefer.

We are now faced with the momentous task of examining all the permutations of the variables, and for each permutation, specifying what should be parallelized. A naive solution would be to examine all $(2n)!$ permutations, and for each permutation, examine all $n+\binom{n}{2}+\binom{n}{3}+...+\binom{n}{n}$ choices of what to parallelize over. This assumes that we can fuse these $2n$ derived variables back together. If n happens to be 3, then this amounts to 5040 choices, all for just one partition strategy! (With 5 different partition strategies for each iteration variable, there are $5^3 = 125$ strategies.) Granted, some

other strategies that don't split all n index variables will have fewer derived index variables to reorder, but this still presents an infeasibly large search space.

2.5.1 Trimming Passes

Trimming the number of permutations and parallelism choices is at the heart of autoscheduling. Because parallelism choices and the loop choices are so intertwined, we will consider them together. But first, let's extend all of the above discussion to sparse tensor algebra.

We have now described how to reduce the problem of autoscheduling dense and sparse iteration spaces to considering permutations of variables and choosing what variables to parallelize over. The general strategy now, will be to come up with **trimming passes** that limits these choices. Each trimming pass is expressed as a condition that the permutations and parallelism must satisfy, based on either heuristics or hardware constraints. In particular, we will define a mathematical object called a **mapping** (naming inspired by Timeloop [16]). A mapping consists of a tuple of an ordered set, which contains the permutation, and an unordered set, which contains the variables over which we would like to parallelize. Note we have not discussed how we'd like to actually implement our desire to parallelize some variables in actual scheduling commands yet. This will depend greatly on the hardware architecture.

The philosophy of introducing trimming passes is also seen in [1] in the form of pruning the search space. Some of the rules mentioned in that work for Halide is also applicable to TACO. I believe that some of the rules that applies to TACO which I will mention here is also applicable to Halide.

Let's describe one particular trimming pass for example.

Trimming pass 0: Sparse Iteration

If we did not fuse a sparse dimension into its corresponding dense dimension, then we can only iterate over the data-dependent variable resulting from this sparse dimension (x_1 from $.split(X, x_1, x_2)$ or x_2 from $.divide(X, x_1, x_2)$) after all the variables from the

corresponding dense dimension have been iterated over. This is because the program needs to fully know where it is in the corresponding dense dimension to determine the iteration bounds for the sparse dimension.

Note that the analysis so far has made no assumption whatsoever about the hardware backend. It is a good time now to introduce the hardware and discuss their various characteristics. This work targets CPUs and GPUs, but I will also discuss how potentially other hardware architectures can be accommodated. For each hardware type, we consider the logical programming model, not the actual physical resources. For example, for GPU we consider the programming model exposed by CUDA. For CPU, we consider programming with OpenMP threads. This work will present some trimming passes for both CPUs and GPUs. To support a new hardware architecture, one would have to write their own trimming passes for it. Of course, one could also devise more trimming passes for CPUs and GPUs that I overlooked. I designed the concept of trimming passes to be composable, very similar to LLVM optimization passes.

In the OpenMP programming model, a loop can be parallelized by adding a schema. Then, different iterations of the loop would be assigned to different threads, either via a static round-robin assignment schedule or a dynamic first-come-first serve schedule. It is important to note that typically, the number of iterations far exceed the number of threads, so each thread can be expected to process a large number of iterations. This suggests that from the perspective of each thread, it still need to iterate over however many variables as there are in the specified permutation, albeit for one of them, it has to do fewer iterations.

In addition to the OpenMP parallelism, CPUs offer additional parallelism in the form of vector units, which consist of a collection of vector lanes that can operate in parallel. These vector units are highly powerful (potential of doing 16 floating point operations in one cycle in AVX512) and should be leveraged if possible. However, sometimes the best way to use them is to not make an effort, as smart compilers such as ICC can usually perform intelligent automatic vectorization. Convoluted

code intended to guide usage of the vector units might have an opposite effect on performance. Here we consider vector parallelism as another annotation on a loop. The code generation mechanism in the scheduling language will understand that the iterations of this loop should be distributed to separate vector lanes.

2.5.2 CPU

Let's talk about trimming passes for CPUs first. I will describe three obvious ones, and then describe some more advanced ones.

Trimming pass CPU-1: Parallelize One Variable

We realize that on the CPU, there is very limited parallelism. As a result, we will not consider parallelizing over more than one variable. In particular, we will not fuse multiple variables are parallelize the result. Note that by disallowing the fusion of derived index variables that iterate over the chunks, Corollary 1.1 no longer holds: fusing index variables and then splitting can produce partitions that splitting index variables independently can't. This trimming pass can be interpretted as saying that we are not going to consider those partitions.

Trimming pass CPU-2: Parallelize Outer Loop

We realize that in OpenMP, parallelizing inner loops often incur massive overhead. Combined with the fact that there is not that much parallelism to begin with, we will only consider parallelizing the outer loop. This rule is also mentioned in [1]. This also implies that if we had partitioned a sparse iteration variable in position space with $split(X, x_i, x_j, K)$, x_i will never get parallelized because it needs be iterated after the variables from X's corresponding dimension.

Trimming pass CPU-3: No Useless Partition

If x_1 and x_2 are the two variables resulting from the same index variable, say X_i via either $split(X_i, x_1, x_2, K)$ or $divide(X_i, x_1, x_2, K)$ and neither is parallelized, then x_2

should not follow directly after x_1 in the permutation. If this is the case, then we should not have split X_i.

Trimming pass CPU-4: Concordant Iteration

Here let's examine our assumption that index variables can be freely reordered in the dense case. While there are no code legality concerns, there could be huge efficacy concerns. In particular, let's imagine a dense matrix stored in row-major format. Iterating through it row major will be a lot faster than iterating through it column major. This suggests that there is a strongly preferred order in which we iterate over each of the input tensors. How do we relate the order over the $2n$ variables resulting from the partitioning to the iteration order over the original index variables to respect these preferences? We seek to define an ordering on the original index variables, with the possibility for equality, in the case of simultaneous iteration from $fuse$.

We note that we can recover where we are in our iteration of our original index variable after we have encountered both of the variables it generated in the permutation. Say we got $split(X_1, x_{11}, x_{12}).split(X_2, x_{21}, x_{22}).split(X_3, x_{31}, x_{32}).reorder(\{x_{11}, x_{32}, x_{31}, x_{12}, x_{21}, x_{22}\})$. Then X_1 is recovered after the program encounters x_{11} and x_{12}, X_2 is recovered after x_{21} and x_{22}, and X_3 is recovered after x_{31} and x_{32}. This suggests that X_3 is iterated before X_1 before X_2 in this particular permutation.

In the case where there are sparse index variables and where $fuse$ is used, we might simultaneously recover two index variables. For example, if we have $fuse(X_1, X_2, F).split(F, x_0, x_1)$ then once we have encountered both x_0 and x_1 in our program, both X_1 and X_2 are recovered. From the perspective of other index variables, the iteration of X_1 and X_2 is simultaneous.

Now that we have figured out how to reason about the order of the original index variables, how do we pick the concordant ones? In general, it is infeasible to select an ordering of the original index variables that respects the data layout preferences for all the input tensors. The easiest counterexample to construe is matrix transposition: if you cater to the input matrix's format you will doom the output matrix's and vice versa. What we can do is to establish a concordancy score, which counts the number

of iteration preferences between pairs of index variables are obeyed in a particular iteration order of the index variables. We should only pick iteration orders with the best concordancy scores.

For example in SpMM: $C(i,k) = A(i,j) \times B(j,k)$ as an example, where j is compressed in A. , there are two dense data structures, C and B. Iteration orders that iterate over i first then j then k would have concordancy scores of 2, which is the best concordancy score achievable here.

Trimming pass CPU-5: Vector Variable

We limit the variables that can be vector parallelized to the last two variables in the permutation. In addition, they will have to be contiguous variables, i.e. they must be the inner variable resulting from a *split* or *divide*. While CPU AVX instructions can specify strides, strided vector instructions are nowhere as efficient as unstrided ones in terms of L1 cache locality.

2.5.3 GPU

Now let's consider what to do for GPUs. Different from CPUs, GPUs offer a massive amount of parallelism in the CUDA programming model. The programmer can launch tens of thousands of logically independent "threads" with options for cooperation and synchronization. A loop can be completely dissolved spatially across a parallelism dimension. For example, instead of iterating over a loop of size 1024, we could launch 1024 threads, each of which will handle one single iteration. However, each of those threads does not execute its assigned portion as efficiently as a CPU thread. In particular, lack of branch prediction hardware makes branching expensive, lack of thread-independent local data caches causes unpredictable cache thrashing between different threads and a lower clock rate makes executing code inherently slow. On GPUs, it is much more important to achieve some semblance of load balance between threads because the hardware often dictates that a group of threads cannot be retired until the slowest thread in that group has finished execution.

While there are many different philosophies to program GPUs efficiently, we here subscribe to the general philosophy that we should maximize the available parallelism and thus minimize the work per thread. As in the CPU case, let's assume we have fixed some permutations of the potentially up to $2n$ variables that we want to schedule over.

Importantly, we will treat the GPU warp as the basic parallel unit, similar to a CPU thread. We will think about parallelizing over warps and how they could be organized into thread blocks. How do we parallelize over warps? We can pick one of the n variables that iterate over chunks, and assign each (or a group of) iteration of it to a distinct warp. We could also potentially pick a group of the n parallelizable variables, fuse all of them, and assign each (or a group of) iteration of the fused variable to a distinct warp. After we have done that, let's think about how to consider the thread block level parallelism. Well now each warp will still iterate over the same list of variables, but for one of them, it will do a reduced number of iterations. Whatever we do, we will gain a variable which iterates over the warps.

We now have the opportunity to group GPU warps into thread blocks. The practical way to implement this would be to consider the variable that now iterates over the warps, and then partition it, assigning chunks to different thread blocks. While this method of considering block level and warp level parallelism is exhaustive in nature, not all parallelism recipes attained by this method is accomplishable.

Unfortunately, TACO does not give us this much freedom in scheduling warps and blocks. As we will describe in the trimming passes, we can not fuse variables resulting from *split* or *divide*, which suggests that we can only parallelize one of the n parallelizable variables, say x_i, over warps. We could then pick another parallelizable variable, say x_j to parallelize over blocks. This would correspond to fusing x_i and x_j and parallelizing the fused variable over warps, and then partitioning the warps such that they operate on different chunks of x_j. Of course, we could parallelize x_i itself further across blocks. These two parallelization strategies correspond to a very small subset of all strategies, which will assist us greatly in narrowing down the search space.

Once we decide what each warp does, it is relatively easier to decide what happens at the thread level. Each warp functions basically like a vector unit with 32 lanes. We will simply pick one of the variables, and iterate over it in a vector fashion.

Trimming pass GPU-1: No Useless Partition

Same as CPU.

Trimming pass GPU-2: No Hierarchial Tiling

This means that when we choose to parallelize a variable, we will completely "dissolve" it. Individual parallel units will no longer need to iterate over that variable, because each parallel unit will be assigned a unique iteration. We are using this trimming pass for three reasons: 1) the hardware supports massive parallelism, which we should cater to. 2) each parallel unit itself is less efficient at handling branching. 3) simplify the scheduling: if one or more variables are dissolved, then effectively it's position in the permutation no longer matters. 4) a very wise man once said that we should program GPUs as if caches don't exist.

Trimming pass GPU-3: Parallelize One Variable at Each Level

As described above, using variables derived from *split* and *divide* is basically not supported. As a result, we should consider only parallelizing one variable over warps. Similarly, we should only parallelize one variable over blocks. This could be the same variable as the one parallelized over the warps or another variable.

Before we proceed further, let's think about what kind of parallelism strategies we are still left to explore. Imagine that we partition X_1, X_2 and end up with the variables $x_{11}, x_{12}, x_{21}, x_{22}$ after our partitioning. Let's say that x_{11} and x_{21} are the parallelizable variables that iterate over the chunks in the partitions. Then the first parallelization strategy would be to pick one of those two, and parallelize it across both warps and blocks, for example the following schedule:.$split(x_{11}, block, warp, 4)$. $parallelize(block, GPUBlock).parallelize(warp, GPUWarp).reorder(x_{21}, x_{12}, x_{22})$. The second strategy would be to parallelize one of those variables over warps and the other

over blocks, for example: $.parallelize(x_{11}, GPUBlock).parallelize(x_{21}, GPUWarp)$. In the first case each warp would iterate over x_{21}, x_{12} and x_{22}, whereas in the second case each warp would iterate over x_{12} and x_{22}.

Trimming pass GPU-4: No Indeterminate Warps

We realize that in the second strategy, the number of warps per thread block is the same as the parallelization variable. Since the number of warps per thread block on the GPU is limited to 16, the size of this parallelization variable must not be data dependent. As a result, it must result from only two of the four partition strategies (it must be the inner variable of a split or the outer variable of a divide).

Trimming pass GPU-5: Enforce Load Balance

All GPU schedules are decreed to be load balanced. While we recognize that sometimes non load-balanced schedules could potentially have better performance, as there is some overhead associated with load balancing. For example, if we parallelize over a fused variable, then each parallel unit will have to perform a binary search [20]. However, on GPUs such an overhead is often justified as common practice, for example in the popular Merge SpMV strategy [12].

What this restriction means in practice, is that 1) you must not split compressed dimensions in coordinate space 2) you cannot parallelize the **corresponding dimensions** of any sparse variables. If you'd like to do so, you'd have to fuse that dimension into the sparse dimension first. For example, in the SpMM problem $C(k, i) = A(i, j) \times B(j, k)$, where A is sparse and j is the sparse dimension, you cannot parallelize i. Since j is sparse, different values of i will have a different number of nonzero j elements. If parallel units receive different chunks of i, then they are almost guaranteed to receive different numbers of nonzeros. You could parallelize k, on the other hand, even though k is also dense while still ensuring load balance.

Trimming pass GPU-6: Concordant Iteration

Same as CPU.

Trimming pass GPU-7: Contiguous Thread Iteration

An important principle in programming GPUs is memory coalescing. This basically states that threads should access contiguous memory addresses, up to some margin. (For example, threads can access stride-K addresses efficiently using the L1 cache if K is sufficiently small.) This is all the more important in sparse tensor algebra, since most kernels are memory bound. As a result, we impose the limit such that the threads within a warp can only be parallelized across contiguous variables.

Trimming pass GPU-8: Illegal Parameters

There are certain schedules where there are guaranteed to be no valid parameter settings, if certain split factors are guaranteed to be smaller than one. These can be safely excluded.

We have now finished describing the trimming passes we employ for CPU and GPU backend. What comes out of the trimming passes are lists of viable mappings that can be easily converted to schedule templates, through emitting the appropriate *parallelize* and *reorder* scheduling commands.

2.6 Search

I would like to briefly describe how one could search through the resulting list of schedules generated from this process. Let's remind ourselves that to generate the schedules, we did three things: determine the overall partition strategy, determine the reordering and parallelization strategy and finally fill in the parameter values. The search space can be grouped hierarchically based on the choices made at each of these three stages. This is a classic multi-arm bandit problem. The schedule templates correspond to different arms of the slot machine. We could try to exploit more parameter choices for a particular schedule template or explore different schedule templates.

It seems to be in fashion as of late to featurize this search space and use some form of machine learning guided optimization, e.g. [6, 1]. This approach is great but

requires significant engineering effort and compute resources. It is not hard to imagine how we could featurize our schedule templates given our detailed analysis up to this point. Perhaps features could include some qualitative aspects, such as what type of partitioning each index variable has undergone (contiguous vs. noncontiguous, fixed vs. data dependent chunk size, etc.), which index variables are paralellized over what kinds of parallel units etc. The features could also include some quantitative aspects, such as how big is the tile size, how big is the thread block size, etc. We could then directly use the methods in [5, 1] to perform a guided search on this search space.

However, there exists simpler alternatives. One of which is Thompson sampling. We could start with the same prior distributions of runtimes for all the schedules associated with a partition strategy. We can then start the exploration process. Since all the prior distributions are the same, we are equally likely to explore each partition strategy. Every time we pick a partition strategy, pick a schedule template from it and fill in some parameters to get a runtime. Then we update the belief distribution of runtimes for that partition strategy. We then reweight the likelihood to pick a select partition strategy based on this updated belief distribution.

The simplest strategy, however, is human guided search, which I employ. I discover that for many problems of interest, the space of partition strategies, and in turn, the space of schedule templates has been reduced so vastly through our meticulous trimming passes that it is feasible to search through them by hand. I strongly recommend this approach when approaching a new problem, especially if you are a generally astute programmer. In this case, the human replaces the Thompson sampler and updates belief distributions in her head.

Chapter 3

Evaluation

I implemented the strategy described in Chapter Two in Python. The schedules generated target TACO, a DSL for sparse linear algebra. When this work is being done, there is no *divide* command implementation yet so we only consider *split*. As described, we first specify the partition strategy, to generate a list of **split schedules**, which do not contain reordering or parallelization information. Then, each split schedule is expanded to consider all possible reordering and parallelization schemes to a list of schedule templates. Finally, each schedule template generates a few different schedules based on different tunable parameter settings.

Here let's present some preliminary results for different sparse tensor algebra problems that we consider. We should especially pay attention to how effective our strategies are in cutting down the search space of our schedule templates. The first step we did was to restrict the ways you could partition up the iteration space, eliminating fusing dense variables etc. Then we severely restricted the number of ways you could reorder and parallelize the resulting variables through the trimming passes.

In order to see our efficacy, we'd need a baseline system. Note that theoretically, every problem has an infinite number of viable schedule templates, e.g. accomplishable simply by reordering two index variables back and forth, or by splitting an axis by a factor of one as many times as you'd like, etc. A naive scheduling system that explores viable commands in a breadth-first or depth-first fashion without heuristics to prune choices is certain to end up with an infinite number of schedule templates.

Because I have no interest in implementing such a system, let's just study the efficacy of our trimming passes, after we have split up the iteration space in a finite number of ways in accordance with the methods proposed in this thesis. (A good reference point might be the work in [1], where they use literally billions of random schedules.)

The baseline system that we consider, is simply one that allows all possible permutations and parallelization choices of the variables resulting from each partitioning strategy. The number of choices can often be calculated analytically for each partition strategy based on the number of variables it produces. For CPUs, assuming that we end up with n variables, then there are $n!$ permutations times n^2 choices for parallelizing over threads and vector. (It is $n \times n$ instead of $n \times (n-1)$ since you could choose not to use vector parallelism). This suggests that naively there are $n^2 n!$ choices.

For GPUs, if we parallelize one variable across blocks and warps, there are n choices for this variable and $(n-1)!$ permutations for the remaining variables. There are $n-1$ choices for a variable to parallelize over threads and two ways to parallelize it, for a total of $2n(n-1)!(n-1) = 2n!(n-1)$ strategies. If we parallelize one variable across blocks and another across warps, then there are $n(n-1)/2$ choices for these variables and $(n-2)!$ permutation for the remaining variables. There are $n-2$ choices for a variable to parallelize over threads and two ways to parallelize it, for a total of $n(n-1)(n-2)!(n-2) = n!(n-2)$ strategies. So for GPU, there are naively $n!(3n-4)$ choices.

We evaluate the autoscheduler on three different sparse tensor algebra problems for CPUs and GPUs. Sparse matrix dense vector multiplication (SpMV), sparse matrix dense matrix multiplication (SpMM) and Matricized Tensor Times Khatri-Rao Product (MTTKRP). For SpMM and SpMV, we use a suite of 12 matrices from the UFlorida SparseSuite Collection [11]. These matrices are selected to be representative of different number of nonzero distributions, densities and sizes.

For MTTKRP, we evaluate on three tensors in the FROSTT dataset [21]. For SpMM, we use a random dense matrix with 128 columns. All evaluation except MTTKRP on GPU was done in double precision. MTTKRP on GPU was done in

single precision.

For each of these problems, we generate a list of schedules and search through all of the generated schedules exhaustively. Due to our aggressive trimming, the search space size is small enough such that we can afford some form of exhaustive search. The tunable parameters are chosen from 8, 16 and 32 for CPU and 4, 16, and 64 for GPU. The exact exhaustive search procedure differs by problem and will be elaborated below. The exhaustive search took anywhere between a few minutes to a day, using at most 48 cores.

For SpMV, we report the runtimes of baselines such as MKL [28] and Eigen [8] for CPU and cuSPARSE [14] and Merge SpMV [12] for GPU on the suite of problems. For SpMM the baselines are MKL for CPU and cuSPARSE for GPU. We also report the runtime of a handtuned schedule baseline. For each problem, we also report the best runtime found during the search as “autoscheduled TACO" results. Note that different problems might use different schedule templates and/or different values of tunable parameters. We do not use unscheduled TACO because the handtuned schedules used in each case are shown to have better or equal performance.

All CPU experiments are run on a dual-socket, 12-core with 24 threads, 2.5 GHz Intel Xeon E5-2680 v3 machine with 30 MB of L3 cache per socket and 128 GB of main memory, running Ubuntu 18.04.3 LTS. On CPU, the code is compiled using Intel icpc 19.1.0.166 with -O3, -DNDEBUG, -march=native, -mtune=native,–ffast-math, and -fopenmp. CPU experiments are cold cache.

All GPU experiments are run on an NVIDIA DGX system with 8 V100 GPUs. Only one GPU is used at a time. I compile the generated code with NVIDIA nvcc 9.0.176 with -O3, -gencode arch=compute_70,code=sm_70, and –use_fast_math.

3.1 SpMV

Index Expression: $y(i) = A(i, j) \times x(j)$

For SpMV, the autoscheduler is capable of generating five different split schedules, as listed in Table 3.1. These five split schedules lead to a total of 50 viable schedules

Table 3.1: SpMV split schedules

1	Empty
2	$.fuse(i, j, x_0).pos(x_0).split(x_0, x_1, x_2, K_0)$
3	$.pos(j).split(j, x_0, x_1, K_0)$
4	$.split(i, x_0, x_1, K_0)$
5	$.split(i, x_0, x_1, K_0).pos(j).split(j, x_2, x_3, K_1)$

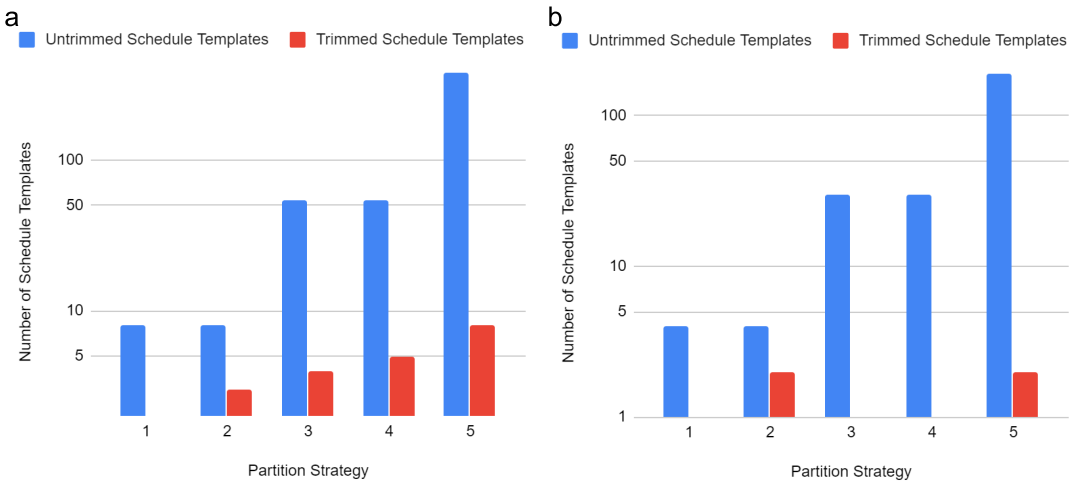

Figure 3-1: The results from trimming the search space of SpMV schedules for a) CPU and b) GPU.

for GPU and 87 viable schedules for CPU that were evaluated on the dataset. I search through all these schedules for all the problems in the benchmark, running them once on every problem to save time. For the best schedules, we proceed to run them 25 times across each problem in the benchmark to collect an average running time, which we report here.

For CPU, one of the schedules exactly correspond to the handtuned schedule in terms of the scheduling commands used and the tunable parameters. However, the autoscheduler uses a different OpenMP schedule than the handtuned schedule (static vs. dynamic), leading to performance differences in some cases. We see from Figure 3-3 that for some of the matrices the autoscheduler does better and for others it does

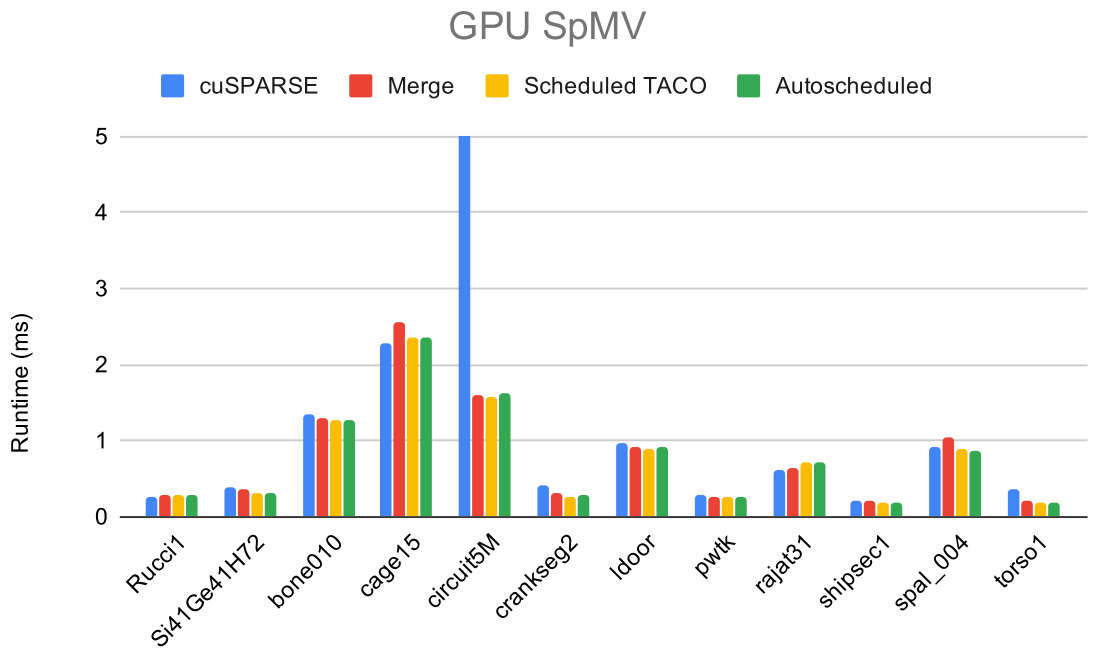

Figure 3-2: Runtimes for cuSPARSE, Merge SpMV, handtuned scheduled TACO and autoscheduled TACO

significantly worse. Interesting the performance of the autoscheduler is very similar to MKL, suggesting that MKL also uses a static load balancing strategy. The best schedule template found for the problems is either the handtuned schedule of a slight variation where one of the variables is vectorized.

For GPU, we see from Figure 3-2 that the best schedule that can be generated from the autoscheduler matches the handtuned schedule in all cases, which in turn is on-par with Merge SpMV and cuSPARSE in terms of performance. The best schedule template found for all the problems is exactly the handtuned schedule, though the parameters are different. The handtuned schedule

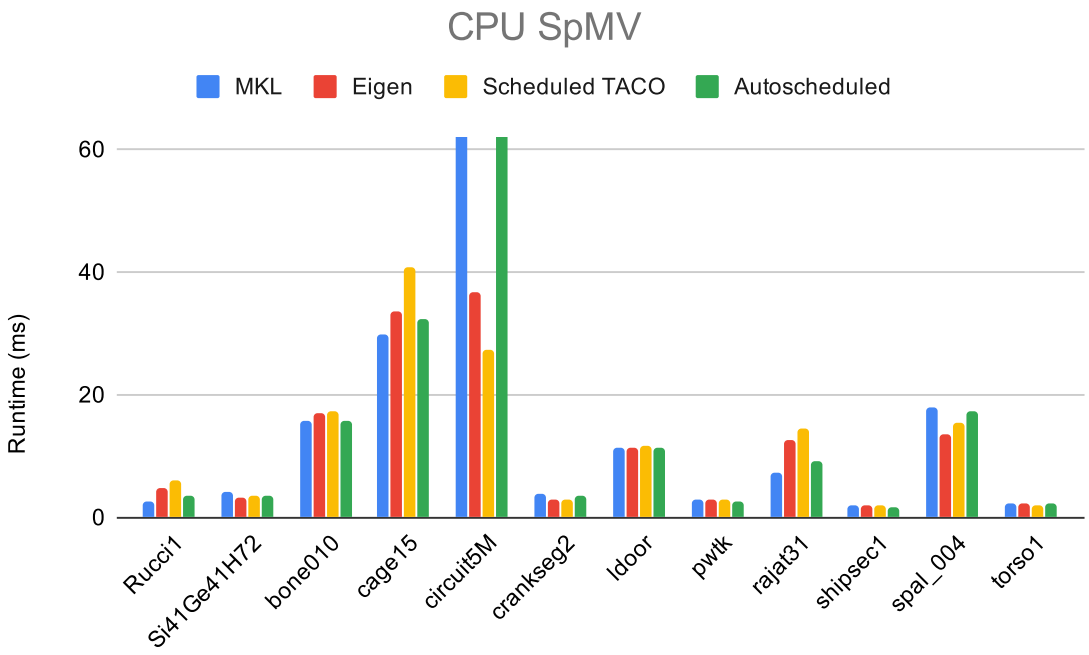

Figure 3-3: Runtimes for cuSPARSE, Merge SpMV, handtuned scheduled TACO and autoscheduled TACO

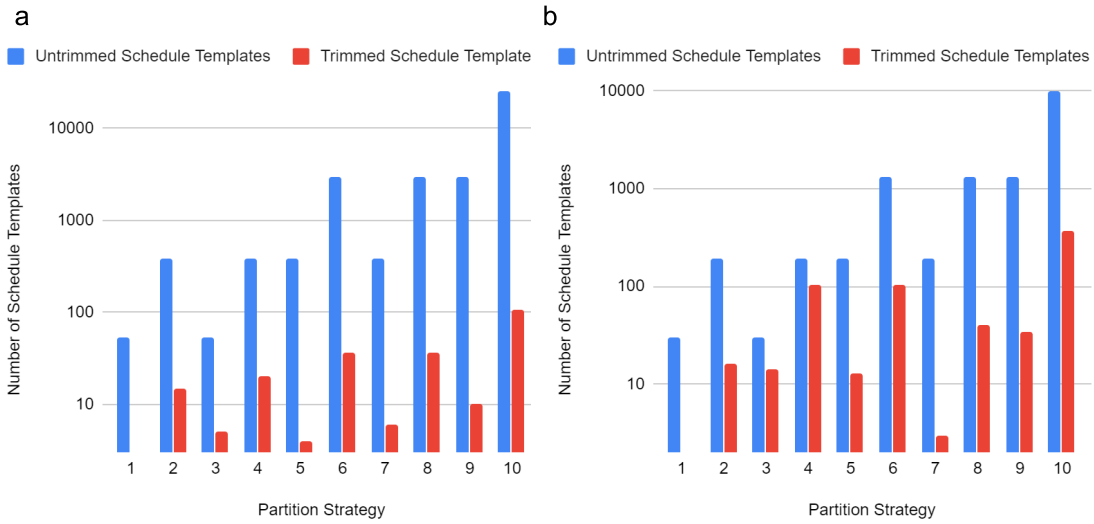

Figure 3-4: The results from trimming the search space of SpMM schedules for a) CPU and b) GPU.

3.2 SpMM

Index Expression: $C(k,i) = A(i,j) \times B(j,k)$ (GPU)

$C(i,k) = A(i,j) \times B(j,k)$ (CPU)

Table 3.2: SpMM split schedules

1	Empty
2	$.split(k, x_0, x_1, K_0)$
3	$.fuse(i,j,x_0).pos(x0,x_1).split(x_1,x_2,x_3,K_0)$
4	$.fuse(i,j,x_0).pos(x_0,x_1).split(x_1,x_2,x_3,K_0).split(k,x_4,x_5,K_1)$
5	$.pos(j,x_0).split(x_0,x_1,x_2,K_0)$
6	$.pos(j,x_0).split(x_0,x_1,x_2,K_0).split(k,x_3,x_4,K_1)$
7	$.split(i,x_0,x_1,K_0)$
8	$.split(i,x_0,x_1,K_0).split(k,x_2,x_3,K_1)$
9	$.split(i,x_0,x_1,K_0).pos(j,x_2).split(x_2,x_3,x_4,K_1)$
10	$.split(i,x_0,x_1,K_0).pos(j,x_2,A(i,j)).split(x_2,x_3,x_4,K_1).split(k,x_5,x_6,K_2)$

For SpMM, the autoscheduler is capable of generating ten different split schedules, as listed in Table 3.2. The tenth split schedule results in hundreds of schedule templates and thousands of schedules. I run each schedule template with a single

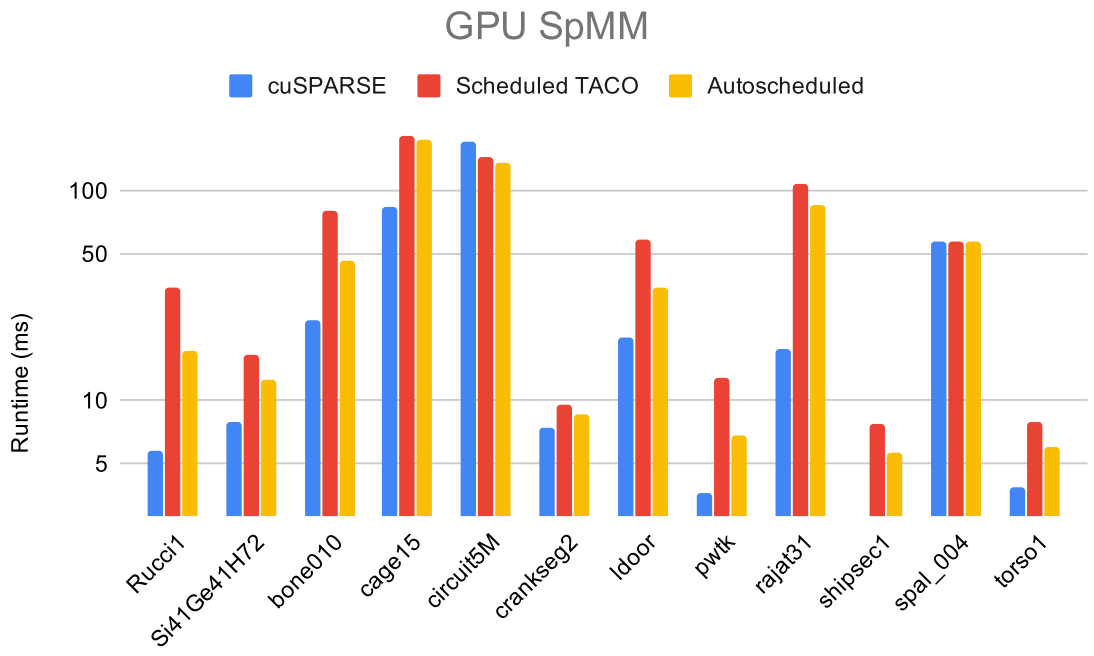

Figure 3-5: SpMM runtimes for cuSPARSE, handtuned scheduled TACO, and autoscheduled TACO.

parameter setting, and discover that none was able to come close in performance to schedules generated from other split schedules. As a result, the tenth split schedule is discarded as a "black hole" (a region of the search space that is immense but barren).

These nine other split schedules lead to a total of 1039 viable schedules for GPU and 798 viable schedules for CPU. I search through all these schedules for all the problems in the benchmark, running them once on every problem to save time. For the best schedules, we proceed to run them 10 times across each problem in the benchmark to collect an average running time, which we report here.

The CPU results are reported in Figure 3-6. We see that for some matrices, the autoscheduler is slower than the handtuned schedule while for others it's slightly faster. The handtuned schedule is outside of the autoscheduler's search space, because the chosen variable ordering violates the concordant iteration heuristic. In addition, the autoscheduler generated schedules use a different OpenMP schedule as mentioned above in the SpMV section.

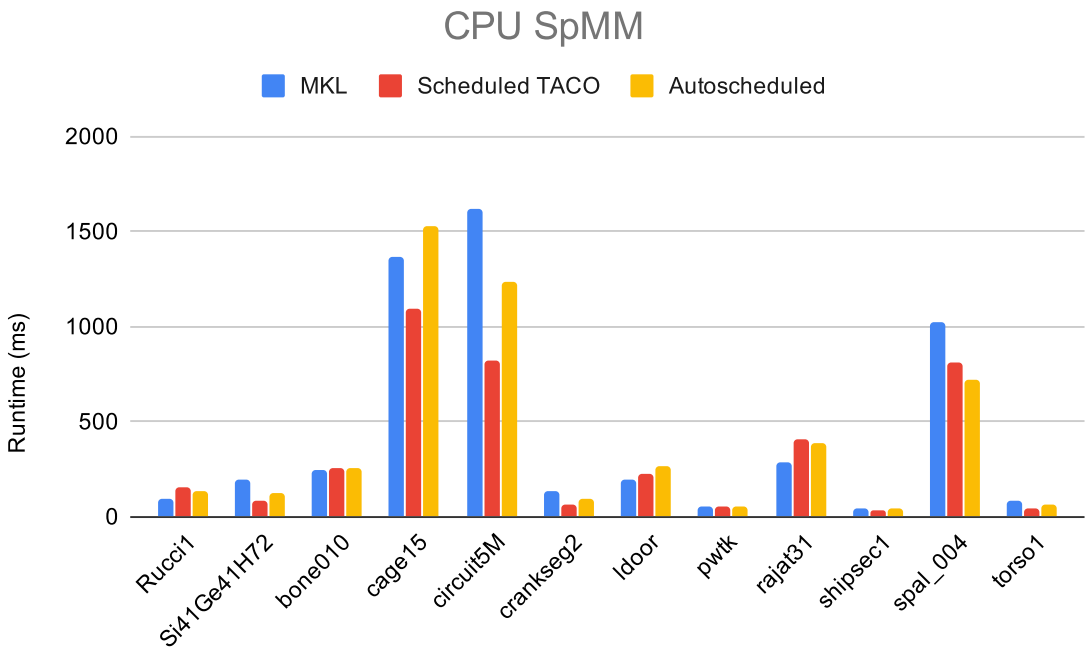

Figure 3-6: SpMM runtimes for cuSPARSE, handtuned scheduled TACO, and autoscheduled TACO.

The GPU results are reported in Figure 3-5. We see that the best schedule able to be generated by the autoscheduler is usually faster than the handtuned schedule and slower than cuSPARSE. However, this difference is mostly due to different tunable parameters being used. The best schedule template for each problem still matches exactly with the handtuned schedule.

3.3 MTTKRP

Index Expression: $A(i,j) = B(i,k,l) \times C(k,j) \times D(l,j)$.

On CPU, we manually specify the precompute command to hoist loop invariant code: $precomputedExpr = B(i,k,l) \times D(l,j)$, $A(i,j) = precomputedExpr \times C(k,j)$. We allocate a buffer to precompute j, called *precomputed*. Then we emit the following precompute command: $precompute(precomputedExpr, j, j, precomputed)$. In terms of the TACO IR, it inserts a where node after iterating over i and k to handle the precomputed buffer. Practically, this suggests that we can only reasonably reorder i and k so as to not interfere with the precompute, and we cannot partition j due to limitations in the scheduling API. This leads to much fewer explored schedules for MTTKRP on CPU, even though it has a more complicated index expression than SpMM and SpMV. Naive estimates suggest around 1000 possible schedule templates, which we trim to 27. This results in around 120 schedules with parameters filled in.

On GPU, we do not use precompute. All the index variables are thus free to be rearranged in whatever order, resulting in a massive potential search space. Naive estimates suggest more than 1 million possible schedule templates. After the trimming, only around 1000 are left. However, only around 200 legal schedules are generated in the end, as a lost of them fail with compile errors, despite the scheduling commands being semantically correct. Note that this is something that any autoscheduling system should consider: the autoscheduler is typically pushing the scheduling API of a DSL to its limits by exploring a deluge of possible schedules. Many semantically correct schedules could fail to compile because of compiler bugs that were simply not exposed during initial tests of the scheduling API with limited handtuned schedules.

Because there are too many split schedules for both CPU and GPU, we do not list them in a table as we do for the other two.

For CPU MTTKRP we find that the autoscheduler can generally match or exceed the handtuned schedule's performance. We observe not much variation in the performance of the schedules.

For GPU MTTKRP we see from Figure 3-8 that the best schedule discoverable

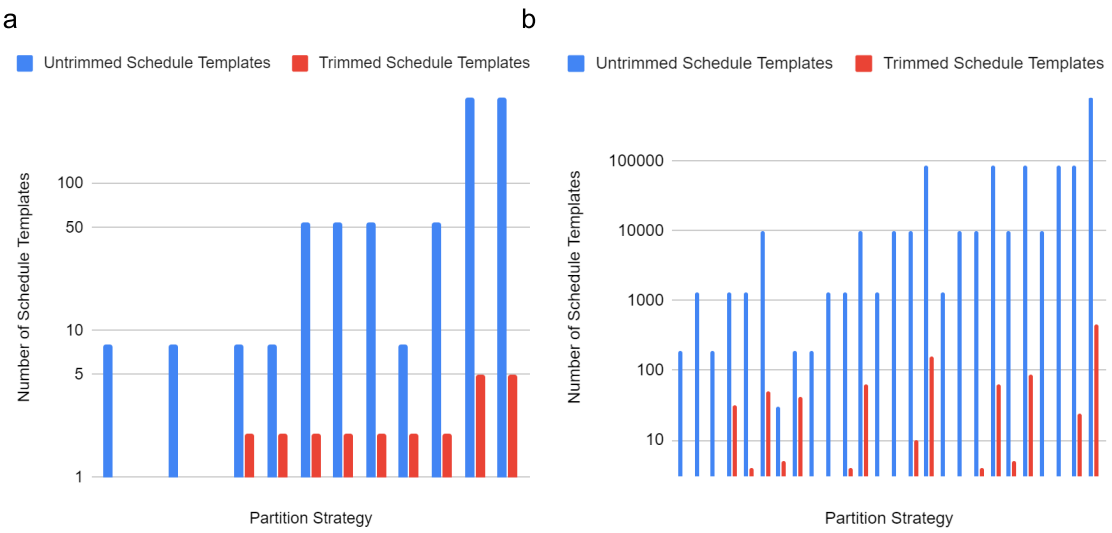

Figure 3-7: The results from trimming the search space of MTTKRP schedules for a) CPU and b) GPU.

by the autoscheduler is much worse than the expert code written by Nisa et al. [15] and the handtuned schedule. This is because the ordering of variables used in the handtuned schedule is excluded from the search space because of the concordant iteration heuristic for dense tensors, similar to the situtation for SpMM. In particular, the handtuned schedule iterates over j before k and l. However, in the handtuned schedule as well as the similar autoscheduler-generated schedules, the range of j is 32 and j is completely parallelized over threads in a warp, with each thread only iterating over 1 value of j. Theoretically then the position of j in the loop order should not change the iteration pattern, suggesting that the autoscheduler-generated schedules process the iteration space in the exact same way as the handtuned schedule. However, in practice, where this size-1 loop is put in the program has a strong impact in performance. Unfortunately this is below the level of abstraction used to construe the heuristics used in the autoscheduler, which reasons about the way the iteration space is traversed.

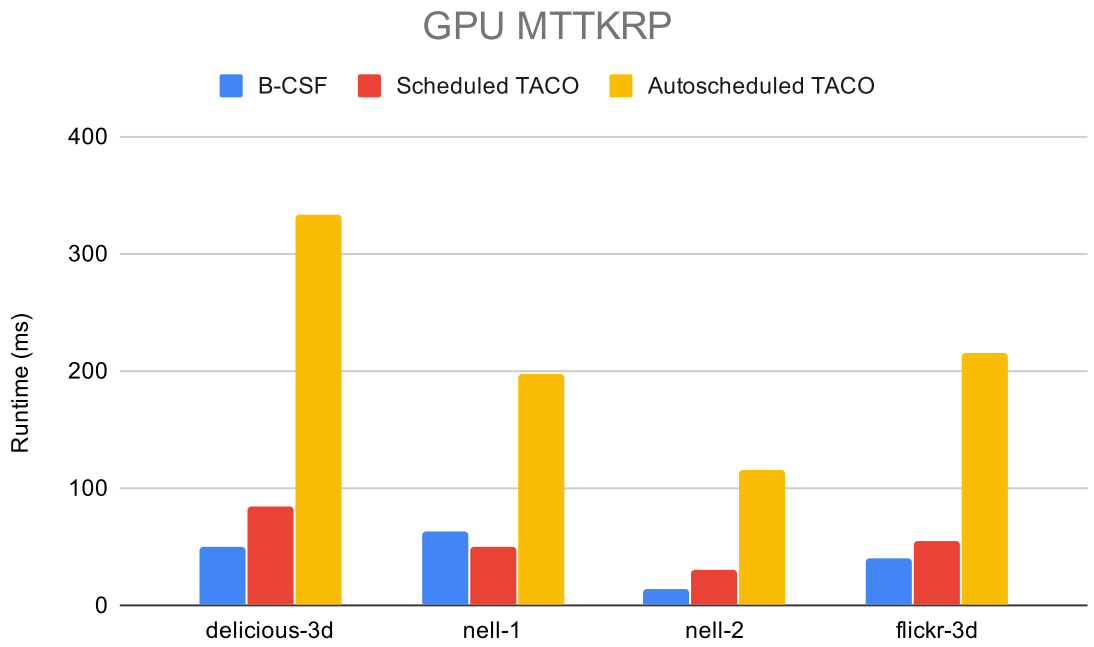

Figure 3-8: B-CSF Kernel by Nisa et al., handtuned schedule and autoscheduled TACO performance on 4 tensors.

3.4 Analysis and Future Work

To remind ourselves of the general procedure, first we fixed the partitions and generated a list of **split schedules**. Then for each split schedule we proceeded to consider all permutations and parallelization schemes to obtain a list of **schedule templates**. Finally for each schedule template we tried different parameter combinations to obtain the final **schedules**.

For SpMV, SpMM and MTTKRP, we more or less did some form of exhaustive search over all the final schedules generated on all the problems in the benchmark suite. For SpMM, we reasoned that one of the split schedules was a “black hole", so we threw away all of its schedule templates despite only experimenting with one parameter setting.

Here, we seek to gain a better understanding of what the space of schedules look like. Let’s remind ourselves of the bubble chart in Figure 1-2. We realize now that the bubble chart is hierarchical—there are different bubbles corresponding to different

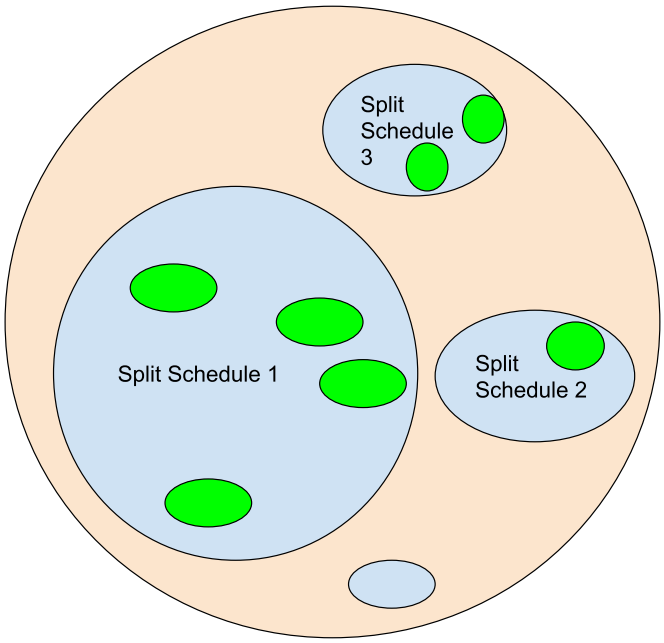

Figure 3-9: Bubble chart with more details filled in. The green circles represent the different schedules generated from the same schedule template. Note that schedule templates from different split schedules can generate different numbers of schedules because they have a different number of tunable parameters.

split schedules with differing sizes considering how many schedule templates they can generate. Inside each split schedule bubble there are smaller bubbles corresponding to different schedule templates. These bubbles are more or less of the same size, as the number of tunable parameter combinations associated to each schedule template depends solely on the number of tunable parameters, which depends only on the split schedule. Some permutation strategies might result in more illegal parameter settings, but those differences are expected to be minor. Here is a slightly updated bubble chart in Figure 3-9.

Let's shed some more light onto the schedule space by seeing what the performance distribution looks like. Let's examine the distribution of runtimes for SpMM on CPU, for two matrices Cage15 and Si41Ge41H72 in Figure 3-10 and Figure 3-11. Different colors correspond to different split schedules. For each of the split schedule, we show what percentage of the schedule templates that result from it is faster than a particular runtime. We see that interestingly, though the runtimes span a huge range across all

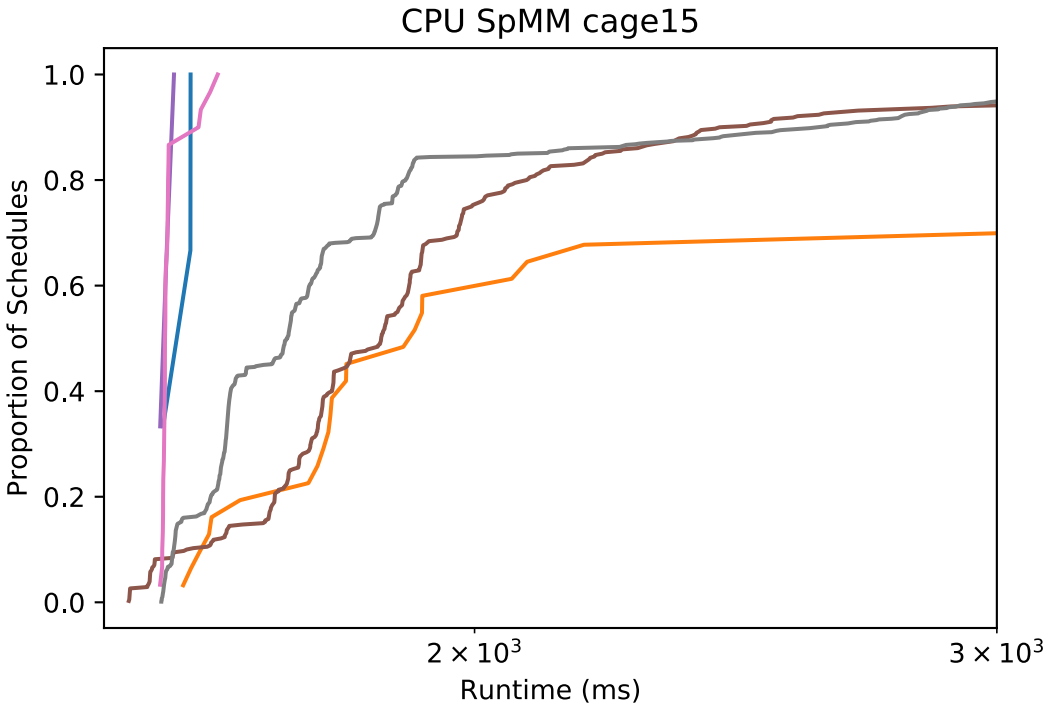

Figure 3-10: Distributions of runtimes for SpMM for cage15 matrix on CPU. Different colors correspond to different split schedules. Some colors overlay almost exactly on each other.

the schedules, they tend to cluster around a few values for some split schedules, i.e. the magenta, blue, green, purple and red ones. This suggests that despite our careful procedure at weeding out redundant schedules in the partition stage and trimming passes, there are still many redundant schedule templates left in the search space. Future work involves examining these clusters and introducing new trimming passes to remove these redundant schedule templates.

However, we note that the trimming passes have already severely cut the number of viable schedule templates for other split schedules, including sometimes the split schedule that results in the best performance. For both Si41Ge41H72 and Cage15, we see that the split schedule that resulted in the best performance generated very few schedules in the end. Introducing trimming passes that will impact every split schedule risks eliminating these few schedules altogether. Indeed, for both SpMM on CPU and MTTKRP on GPU, the handtuned schedule is outside of the search

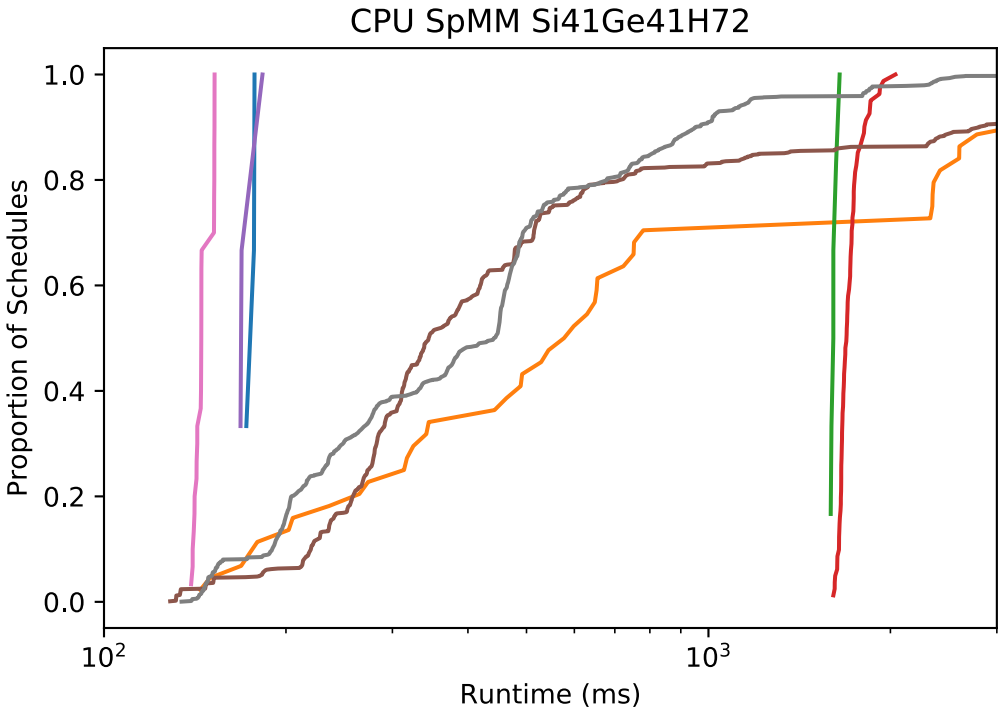

Figure 3-11: Distributions of runtimes for SpMM for Si41Ge41H72 matrix on CPU. Different colors correspond to different split schedules.

space due to the restrictive concordant iteration trimming pass. However, if that trimming pass is eliminated, the schedule space would be roughly five times the size. We have thus stumbled upon the fundamental compromise in autoscheduling: how do we eliminate schedules such that we don't toss out the baby with the bathwater. Different use cases could call for different solutions. If you are going to spend millions of CPU-hours repeatedly executing a single sparse tensor algebra expression (e.g. SpMV in iterative solvers), then it might be worth generating a massive search space and spend a lot of time finding the best schedule. If you just want to quickly find a schedule for an expression that you will execute once or twice, then you might want a much more constrained search space so picking a single schedule is a lot easier.

I'd like to make the observation that in both cases shown here (and in most other cases), it is quite difficult to find the best schedule in terms of performance, but it's relatively easy to find a schedule that performs reasonably well. This suggests that

the search space has plenty of local minima with a few global minima. The exhaustive search procedure is able to recover the global minimum, but the plots would suggest that even a random search would find a reasonably good schedule in just a few tries (the best brown bar in both plots). This suggests that if the user does not have much time to do tuning, he/she can just select a random schedule. Most likely it will perform decently. This in part points to the success at our trimming passes in removing inefficient schedule templates.

Chapter 4

Discussion and Related Work

Thinking about transforming the iteration space of a program through scheduling commands assumes a particular view of the iteration space of the program. Scheduling APIs transform the abstract syntax tree (AST) of the program, which often manifests itself as a loop nest. Put more plainly, this perspective treats the program as a sequence of for loops. Scheduling commands strip-mine, tile, reorder, unroll or parallelize those loops [18, 10]. The autoscheduler in this case tries to find the best "tags" for the different loop nests in the program. Thinking about the program in this way usually leads to studying ASTs. For example, [6] builds tree GRUs to find good schedules and [1] focuses on strategies to perform tree search on the AST. Certain information, such as iteration order, is conveyed straightforwardly by the AST. Other information, such as how the iteration space is partitioned, can be hard to glean from the AST. In particular, it is difficult to tell that two sets of for loops effect equivalent partitions of the iteration space by just looking at loop structure.

When examining the partitioning strategies of the iteration space, I believe that we should adopt a geometric perspective, one that is championed by the Polyhedral Model [26, 4, 27]. This model treats the work to be done inside the for loops as an execution instance, and visualizes a polytope of execution instances corresponding to the iteration space. The polyhedral approach champions another style of autoscheduling. The autoscheduler is formulated as the best affine transformation of the polytope under some chosen cost function. Typical cost functions balance data

reuse and parallelism, depending on the hardware target [27]. This takes the form of a linear program or an integer linear program and the best schedule can be solved for analytically. There are DSLs that adopt this approach, for example PCCG and Tensor Comprehension [27, 24]. There have been recent works attempting to extend this framework to support sparsity as well [22, 23, 25].

Here, the only thing I take from the Polyhedral Model is treating the iteration space like a grid of points. We can easily tell if two ASTs effect the same partition of the iteration space by looking at their geometrical effects on this grid. Indeed, we were able to derive equivalence relations of AST transformations from this perspective.

Instead of using affine transformations to determine the iteration order of the execution instances after the partitioning, we revert back to the AST view. Although we are able to describe iteration order in the Polyhedral Model using directed edges between execution instances, I find it much easier to construct heuristics governing iteration order in terms of the order of for loops. Perhaps, this is because it is easier to reason at a higher level about relations between index variables, instead of the execution instances.

Of course, one does not have to switch perspectives. One could specify equivalence relations of AST transformations purely through reasoning about loops, and the Polyhedral Model is rich enough to fully describe any heuristics governing iteration order of the execution instances. I just find switching perspectives more convenient.

This work is not the first in attempting to explore the synergy between polyhedral and AST based approaches. For example, the Tiramisu compiler is an interesting case. The DSL itself exposes certain affine transformations of the polytope as scheduling commands. However, it does not try to optimize for the best polytope transformation. Instead, the programmer is expected to supply the optimal schedule, in the AST scheduling style [3]. Recent efforts are underway in using machine learning in the AST style to build an autoscheduler, similar to the Halide effort. The underlying assumption here is that the polyhedral model expresses sensible transformations of the schedule, but explicit-cost-model based optimization is not optimal.

Chapter 5

Conclusion

This thesis provides an approach to reason about optimizing dense and sparse tensor algebra programs. We show that by reasoning about the computation and the hardware backend, we can introduce heuristics that greatly reduce the search space of viable optimization strategies. This approach is complementary with machine learning techniques to search over those strategies. While the implementation is centered around TACO's scheduling API, the approach could be generalized to other DSLs with scheduling APIs, such as Halide and Tiramisu, a promising direction of future work.

Bibliography

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

[2] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 303–316, 2014.

[3] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, pages 193–205. IEEE Press, 2019.

[4] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *International Conference on Compiler Construction*, pages 283–303. Springer, 2010.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: end-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799*, 2018.

[6] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*, pages 3389–3400, 2018.

[7] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):123, 2018.

[8] Gaël Guennebaud, Benoit Jacob, et al. Eigen. *URl: http://eigen. tuxfamily. org*, 2010.

[9] Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, and Saman Amarasinghe. Tensor algebra compilation with workspaces. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, pages 180–192. IEEE Press, 2019.

[10] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):77, 2017.

[11] Scott Kolodziej, Mohsen Aznaveh, Matthew Bullock, Jarrett David, Timothy Davis, Matthew Henderson, Yifan Hu, and Read Sandstrom. The suitesparse matrix collection website interface. *Journal of Open Source Software*, 4(35):1244, 2019.

[12] Duane Merrill and Michael Garland. Merge-based parallel sparse matrix-vector multiplication. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 678–689. IEEE, 2016.

[13] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)*, 35(4):83, 2016.

[14] M Naumov, LS Chien, P Vandermersch, and U Kapasi. Cusparse library. In *GPU Technology Conference*, 2010.

[15] Israt Nisa, Jiajia Li, Aravind Sukumaran-Rajam, Richard Vuduc, and P Sadayappan. Load-balanced sparse mttkrp on gpus. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 123–133. IEEE, 2019.

[16] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 304–315. IEEE, 2019.

[17] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Halide: decoupling algorithms from schedules for high-performance image processing. *Communications of the ACM*, 61(1):106–115, 2017.

[18] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Acm Sigplan Notices*, volume 48, pages 519–530. ACM, 2013.

[19] Ryan Senanayake. A unified iteration space transformation framework for sparse and dense tensor algebra. Master's thesis, Massachusetts Institute of Technology, 1 2020. http://rsenapps.com/senanayake-thesis.pdf.

[20] Ryan Senanayake, Fredrik Kjolstad, Changwan Hong, Shoaib Kamil, and Saman Amarasinghe. A unified iteration space transformation framework for sparse and dense tensor algebra. *arXiv preprint arXiv:2001.00532*, 2019.

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

[22] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. The sparse polyhedral framework: Composing compiler-generated inspector-executor code. *Proceedings of the IEEE*, 106(11):1921–1934, 2018.

[23] Michelle Mills Strout, Alan LaMielle, Larry Carter, Jeanne Ferrante, Barbara Kreaseck, and Catherine Olschanowsky. An approach for code generation in the sparse polyhedral framework. *Parallel Computing*, 53:32–57, 2016.

[24] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[25] Anand Venkat, Mary Hall, and Michelle Strout. Loop and data transformations for sparse matrix code. In *ACM SIGPLAN Notices*, volume 50, pages 521–532. ACM, 2015.

[26] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In *International Congress on Mathematical Software*, pages 299–302. Springer, 2010.

[27] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):54, 2013.

[28] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*, pages 167–188. Springer, 2014.