Autoscheduling for Sparse Tensor Algebra with an Asymptotic Cost Model

Peter Ahrens
MIT CSAIL
Cambridge, MA, USA
pahrens@csail.mit.edu

Fredrik Kjolstad
Stanford University
Stanford, CA, USA
kjolstad@cs.stanford.edu

Saman Amarasinghe
MIT CSAIL
Cambridge, MA, USA
saman@csail.mit.edu

Abstract

While loop reordering and fusion can make big impacts on the constant-factor performance of dense tensor programs, the effects on sparse tensor programs are asymptotic, often leading to orders of magnitude performance differences in practice. Sparse tensors also introduce a choice of compressed storage formats that can have asymptotic effects. Research into sparse tensor compilers has led to simplified languages that express these tradeoffs, but the user is expected to provide a schedule that makes the decisions. This is challenging because schedulers must anticipate the interaction between sparse formats, loop structure, potential sparsity patterns, and the compiler itself. Automating this decision making process stands to finally make sparse tensor compilers accessible to end users.

We present, to the best of our knowledge, the first automatic asymptotic scheduler for sparse tensor programs. We provide an approach to abstractly represent the asymptotic cost of schedules and to choose between them. We narrow down the search space to a manageably small Pareto frontier of asymptotically non-dominating kernels. We test our approach by compiling these kernels with the TACO sparse tensor compiler and comparing them with those generated with the default TACO schedules. Our results show that our approach reduces the scheduling space by orders of magnitude and that the generated kernels perform asymptotically better than those generated using the default schedules.

CCS Concepts: • **Theory of computation → Complexity theory and logic**; *Database query languages (principles)*; • **Mathematics of computing → Mathematical software**.

Keywords: Sparse Tensors, Compilers, Asymptotic Analysis, Automatic Scheduling, Conjunctive Query Containment, Query Optimization

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06.
https://doi.org/10.1145/3519939.3523442

ACM Reference Format:
Peter Ahrens, Fredrik Kjolstad, and Saman Amarasinghe. 2022. Autoscheduling for Sparse Tensor Algebra with an Asymptotic Cost Model. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3519939.3523442

1 Introduction

Transformations like loop fusion or loop reordering can have large constant factor effects on the runtime of dense tensor programs [3]. However, the same transformations can have much larger asymptotic effects when tensors are sparse.

To see the effects of loop fusion, consider the sampled matrix multiply $A_{ij} = \sum_k B_{ik} \cdot C_{kj} \cdot D_{ij}$ where B and C are dense and A and D are sparse (SDDMM). When we multiply B and C first, then multiply by D, our runtime is $O(IJK)$, where I, J, and K are the dimensions of i, j, and k. If we fuse the two multiplications into one nested loop, our runtime is reduced to $O(\text{nnz}(D)K)$, where $\text{nnz}(D)$ is the number of nonzeros in D. Loop reordering matters too. As we illustrate in Section 3, loop reordering constitutes the main asymptotic difference between the three main algorithms for sparse-sparse matrix multiply (SpGEMM) [15, 29, 43]. Complicating matters, sparse tensors may be stored in different compressed formats with varied asymptotic behaviors. Making n random updates to a list of nonzero coordinates might take $O(n^2)$ time, but would only take $O(n)$ time if we used a hash table.

Research into sparse tensor compilers has led to simplified languages to express these tradeoffs and generate efficient implementations [5, 12, 14, 21, 22, 31, 33, 34, 37–39, 48, 51, 53, 54, 56]. These sparse tensor compilers separate **mechanism** (how code is generated from the high-level description) from **policy** (deciding what high-level description is best) [49].

Sparse tensor compilers leave policy to the user, which may be a burden for domain experts who are not performance engineers. Writing a good schedule takes time, especially when there are many kernels or the tensor compiler is unfamiliar to the user. Sparse systems must follow dense systems in moving towards automatic scheduling [2, 3, 45]. Automatic scheduling promises a realistic path towards integration into high-level systems like SciPy [58] or TensorFlow [1]. Whereas sparse tensor compilers have made performance engineers more productive, our automatic scheduler will make domain experts more productive.

We present, to the best of our knowledge, the first automatic scheduler for asymptotic decision making in sparse tensor programs. At its core is an asymptotic cost model that can automatically analyze and rank the complexity of sparse tensor schedules over all possible inputs to the program. Autoschedulers often make high-level decisions before considering fine-grained implementation details [3]. We ignore constant-factor optimizations such as sparse format implementations, cache blocking, or parallelization. We focus only on novel asymptotic concerns, such as loop fusion, loop reordering, and protocols. We introduce a precise intermediate representation, the Protocolized Concrete Index Notation (abbreviated CIN-P). Our method can detect cases where one program performs strictly more work than another across all input patterns, up to constant factors. The result is a frontier of asymptotically non-dominating programs, one of which being the best choice for any given input, up to constant factors. Programs in the (manageably small) frontier may be later embellished with constant-factor optimizations.

Tensor kernels are usually static from run to run. Thus, we design for an offline use case, where the autoscheduler runs only once per kernel and is given no information about the sparsity patterns of the inputs. In contrast, online autoschedulers execute at runtime, running once per input pattern and using the pattern to specialize the implementation just-in-time. Both regimes are important. While the runtime of online autoschedulers competes with the optimizations they deliver, the runtime of offline autoschedulers only competes with the equivalent developer effort required to write a schedule. Offline autoschedulers might run on a dedicated server as new schedules for kernels are requested by users, and new schedules could ship with each update to the tensor compiler. The TACO web scheduling tool has recorded only 2758 distinct tensor programs since 2017. These distinct schedules had a median of 3 indices and 6 tensors, and a mean of 3.17 and 6.63 tensors. Because sparse tensor programs are often small (only a few indices and tensors) and the offline scheduling use case affords an extensive amount of time to produce schedules, exponential-time solutions to difficult offline scheduling problems are within reach.

This paper makes the following contributions:

- We define a language (CIN-P) for the implementation of sparse tensor programs at a high level, specifying the loop structure, temporary tensors, and tensor formats. Our language separates the storage formats from how they should be accessed (the protocol).
- We model asymptotic complexity using abstract set expressions. We describe algorithms to derive the modeled complexity cost of CIN-P schedules and determine when one complexity dominates another.
- We use our cost model to write an asymptotic autoscheduler for CIN-P sparse programs. We enumerate equivalent programs of minimal loop nesting depth, filter these programs to the asymptotic frontier, and use a novel algorithm to automatically insert workspaces for transpositions and reformatting. We demonstrate that our asymptotic frontier is often several orders of magnitude smaller than the minimum depth frontier.
- We evaluate our approach on the subset of CIN-P programs supported by the TACO tensor compiler, demonstrating performance improvements of several orders of magnitude over TACO's default schedules.

2 Background

Tensor compilers provide a mechanism to automatically generate efficient code for simple loop programs that operate on tensors, or multidimensional arrays [5, 12, 14, 21, 22, 31, 33, 34, 37–39, 48, 53, 54, 56]. Sparse tensor compilers are specialized for the case where tensors are mostly zero and only nonzero elements are stored, making the problem especially complex. In addition to loop ordering problems of the dense case, the sparse code must also iterate over compressed representations of the input during the kernel computation. Sparse matrix representations have a long history of study, and are typically specialized to the kernel to be executed. We build on our previous work on the TACO sparse tensor compiler [21, 22, 31, 33, 34], though our work certainly applies to similar compilers such as COMET [54] or the MLIR sparse tensor dialect [11]. TACO simplifies compilation of sparse tensor programs by considering each dimension separately. A type system is used to specify whether each dimension is to be compressed (and therefore iterated over), or dense (and accessed with direct memory references). Workspaces, or temporary tensors, may be introduced to hold intermediate results. The loop ordering, workspaces, and sparse formats of the inputs together form the high-level description, a **schedule**, from which TACO generates code.

Different schedules may have different asymptotic effects that depend on the sparsity patterns of the input. TACO generates code using the properties that $a \cdot 0 = 0$ and $a + 0 = a$, so only nonzero values need to be processed, and when tensors are multiplied, only their shared nonzero values need to be recorded. In the first case, we can avoid computing a in its entirety. Workspaces may be inserted to cache intermediate results to avoid redundant computation, provide an effective buffer to avoid asymptotically expensive operations on sparse tensor formats, or perform filtering steps by exposing intermediate zero values early in a computation. The order in which sparse loops are evaluated can have asymptotic effects as well, since sparse outer loops may act as filters over their corresponding inner loops.

3 Motivating Examples

Three nested loops are required to compute a sparse matrix-matrix multiply (SpGEMM). The three main algorithms for

SpGEMM are distinguished by the order in which these loops are nested. We write SpGEMM as

$$A_{ij} = \sum_k B_{ik} \cdot C_{kj}.$$

The "Inner Products" approach processes this expression as a set of loops nested in i, j, k order, with the inner loop performing sparse inner products, merging nonzeros in rows of B and columns of C [43]. Even though we only need to multiply the shared nonzeros, merging the lists has a runtime proportional to their length, regardless of how many nonzeros they share. The improved "Outer Products" algorithm loops in k, i, j order, scattering (writing in no particular order) into A and only iterating through shared nonzeros[15]. Setting k to be the outer loop ensures that the nonzero i and j in inner loops share a value of k and correspond to necessary work. Gustavson's algorithm iterates in i, k, j order, representing a compromise between the two approaches that avoids the need to scatter into a two-dimensional output [29]. It should be noted that scattering operations incur asymptotic costs as well depending on the format used to store the output.

Loop fusion can be critical for sparse programs like sampled dense-dense matrix multiplication, written as

$$A_{ij} = \sum_k B_{ik} \cdot C_{kj} \cdot D_{ij}$$

where B and C are dense, but D is a sparse matrix. If we process this kernel with a dense matrix multiply followed by a sparse mask, like $A_{ij} = w_{ij} \cdot D_{ij}$ **where** $w_{ij} = \sum_k B_{ik} \cdot C_{kj}$, then the runtime is $O(I \cdot J \cdot K)$. If we instead process all three matrices at once, the runtime is $O(nnz(D) \cdot K)$.

On the other hand, inserting temporaries and avoiding loop fusion can be critical for kernels like the three-way pointwise sparse matrix product,

$$A_{ij} = B_{ij} \cdot C_{ij} \cdot D_{ij}.$$

Inserting a temporary like

$$A_{ij} = w_{ij} \cdot D_{ij} \textbf{ where } w_{ij} = B_{ij} \cdot C_{ij}$$

will avoid reading rows of D when the product $B_{ij} \cdot C_{ij}$ produces empty rows.

4 Protocolized Concrete Index Notation

In order to make our descriptions of sparse tensor algebra implementations more precise, we introduce the protocolized concrete index notation, an extension to concrete index notation [33]. Our notation starts with the tensors themselves. A more detailed description of TACO-style formats is given by Chou et. al. [21]. A **rank**-r tensor A of **dimension** $I_1, ..., I_r$ maps r-tuples of integers $(i_1, ..., i_r) \in 1 : I_1 \times ... \times 1 : I_r$ to values v. We often express this as $A_{i_1,...,i_r} = v$. Each position in the tuple is referred to as a **mode**. We represent sparse tensors using trees where each node $(i_1, ..., i_l)$ at **level** l in the tree represents a slice $T_{i_1,...,i_l,:,...,:}$ of the tensor that contains at least one nonzero. Each level is stored in a particular format. If the format is **uncompressed** (**u**), then for each node $(i_1, ..., i_{l-1})$ stored by the previous level, we store an array of all possible children $1 : I_l$. If the format is **compressed** (**c**), then we only store a list of nonzero children (and their locations). If the format is a **hash table** (**h**), we store the same information as the list format, but we use a hash table, enabling random access and insertion. We abbreviate our formats with their first letter and specify them as superscripts, read from left to right corresponding to the top down to the bottom of the compressed tensor tree. A matrix stored in the popular **CSR** format (rows are stored at the top level in an array, and columns as a list in the bottom level), would be written as $A^{\mathbf{uc}}$. If the modes are to be stored in a different order, we write the permutation next to the format, so **CSC** format (where columns are stored in the top) would be written as $A^{\mathbf{u(2)c(1)}}$. Another popular format, **DCSR**, stores both the rows and columns in compressed levels, written as $A^{\mathbf{cc}}$. DCSR is similar to a list of lists, which is useful when many of the rows are sparse.

The following paragraph formalizes concrete index notation, described in earlier work [33]. Examples are given in Figure 1. We use **index variables** to specify a particular element of a tensor. Tensors may be **accessed** by index variables $i...$ as $A_{i...}$. We can combine accesses into **index expressions** with function calls, such as the calls to + and · in $B_{ij} + 2 \cdot C_k$. An **assignment** statement writes to an element of a tensor, and takes the form $A_{i...} = expr$, where $expr$ is an index expression. An **increment** statement updates an element of a tensor, taking the form $A_{i...} \oplus= expr$ where $\oplus$ is a binary operator such as + or · and roughly meaning $A_{i...} = A_{i...} \oplus expr$, although we disallow the left hand side tensor from appearing on the right hand side of assignment or increment statements. The assignment statement "returns" the tensor on its left hand side, which can be used by a **where** statement. The statement *consumer* **where** *producer* first initializes the tensor to be returned by *producer*, then evaluates the statement *producer*, then makes the tensor it returns available for use in the scope of *consumer*, returning the tensor returned by *consumer*. Thus, workspace tensors are initialized just before the innermost **where** statement that contains them on the right hand side. The **forall** statement $\forall_{i...} body$ evaluates *body* over all assignments to the indices $i...$ and returns the tensor returned by *body*. However, when operands are sparse, we can skip some evaluations.

Adding to the existing concrete index notation, we introduce the notion of a **protocol**, used to describe how an index variable should interact with an access when that variable is quantified. The **step** (**s**) protocol indicates that the forall should coiterate over a list of nonzeros of the corresponding tensor, and substitute the default tensor value 0 into the body for the other values. The **locate** (**l**) protocol indicates that

$$\left(\forall_{ijk} A^{\mathbf{cc}}_{\mathbf{a}(i)\mathbf{a}(j)} \mathrel{+}= B^{\mathbf{cc}}_{\mathbf{s}(i)\mathbf{s}(k)} \cdot {C'}^{\mathbf{cc}}_{\mathbf{s}(j)\mathbf{s}(k)}\right) \textbf{ where } \left(\forall_{jk} {C'}^{\mathbf{cc}}_{\mathbf{i}(j)\mathbf{i}(k)} = C^{\mathbf{cc}}_{\mathbf{s}(k)\mathbf{s}(j)}\right)$$

↓

```
for i ∈ 𝒦(B) do
    for j ∈ 𝒦(C′) do
        for k ∈ 𝒦(B_i) ∪ 𝒦(C′_j) do
            if B_ik ≠ 0 ∧ C′_jk ≠ 0 then
                A_ij ← A_ij + B_ik · C′_jk
```

(a) Inner Products compilation (CIN-P → Pseudocode)

$$\left(\forall_{ij} A^{\mathbf{cc}}_{\mathbf{i}(i)\mathbf{i}(j)} = {A'}^{\mathbf{hh}}_{\mathbf{s}(i)\mathbf{s}(j)}\right) \textbf{ where } \left(\left(\forall_{kij} {A'}^{\mathbf{hh}}_{\mathbf{i}(i)\mathbf{i}(j)} \mathrel{+}= {B'}^{\mathbf{cc}}_{\mathbf{s}(k)\mathbf{s}(i)} \cdot C^{\mathbf{cc}}_{\mathbf{s}(k)\mathbf{s}(j)}\right) \textbf{ where } \left(\forall_{ki} {B'}^{\mathbf{cc}}_{\mathbf{i}(k)\mathbf{i}(i)} = B^{\mathbf{cc}}_{\mathbf{s}(i)\mathbf{s}(k)}\right)\right)$$

↓

```
for k ∈ 𝒦(B′) ∪ 𝒦(C) do
    if B′_k ≠ 0 ∧ C_k ≠ 0 then
        for i ∈ 𝒦(B′_k) do
            for j ∈ 𝒦(C_k) do
                A_ij ← A_ij + B′_ki · C_kj
```

(b) Outer Products compilation (CIN-P → Pseudocode)

$$\forall_i \left(\left(\forall_j A^{\mathbf{cc}}_{\mathbf{a}(i)\mathbf{a}(j)} = w^{\mathbf{h}}_{\mathbf{s}(j)}\right) \textbf{ where } \left(\forall_{kj} w^{\mathbf{h}}_{\mathbf{i}(j)} \mathrel{+}= B^{\mathbf{cc}}_{\mathbf{s}(i)\mathbf{s}(k)} \cdot C^{\mathbf{cc}}_{\mathbf{s}(k)\mathbf{s}(j)}\right)\right)$$

↓

```
for i ∈ 𝒦(B) do
    w ← 0                                   ▷ Initialize w
    for k ∈ 𝒦(B_i) ∪ 𝒦(C) do
        if B_ik ≠ 0 ∧ C_k ≠ 0 then
            for j ∈ 𝒦(C_k) do
                w_j ← w_j + B_ik · C_kj
    for j ∈ 𝒦(w) do
        A_ij ← w_j
```

(c) Gustavson's compilation (CIN-P → Pseudocode)

Figure 1. Input → output compilation examples various matrix multiply approaches. We compile CIN-P expressions to pseudocode. For feasibility of analysis, pseudocode is slightly simplified from what TACO would generate, see Section 5.2.

the forall should ignore this access for the purposes of determining which values to coiterate over. The list format only supports the step protocol, and the array format only supports the locate protocol, but the hash format supports both. We use separate protocols for writes. We say that a write is an **append** (**a**) protocol when we can guarantee that writes to that mode and all modes above it in the tensor tree will occur in lexicographic order. We say that a write is an **insert** (**i**) protocol otherwise. In effect, append and insert mirror the flavor of step and locate. We abbreviate our protocols with their first letter and specify them like functions surrounding indices in our access expressions, meant to specify how each mode of the tensor should be accessed. If we mean to read the mode indexed by i using step protocol and the mode indexed by j using locate protocol, we would write $A_{\mathbf{s}(i)\mathbf{l}(j)}$.

It helps to have some examples of protocolized concrete index notation and the kind of code it might generate. For this purpose, we introduce the notation

$$\mathcal{K}(A_{i\ldots}) = \{j | \exists_{k\ldots} A_{i\ldots,j,k\ldots} \neq 0\}$$

as a way to refer to the indices of nonzero slices within some slice of the tensor. These indices are usually stored in a list by the sparse level format.

Figure 1a compiles an example CIN-P expression for the inner products approach to matrix multiplication. The producer side of the where statement transposes C into a workspace C' so that the tree order in the consumer side agrees with the quantification order. The resulting pseudocode will iterate over $\mathcal{K}(B_i) \cup \mathcal{K}(C'_j)$, even though it only needs to iterate over $\mathcal{K}(B_i) \cap \mathcal{K}(C'_j)$. To improve the situation, we might choose to use the outer products algorithm, shown in Figure 1b where we have used a temporary hash format to handle the random accesses to A and a transposition of B to access k first. In this version, we have avoided repeating the filtering step for every i and j. Unfortunately, this version introduces a two-dimensional scatter, which can be expensive. Instead, we may choose to use Gustavson's algorithm, shown in Figure 1c. Gustavson's algorithm is our first example with a quantified where statement. This form of the algorithm is a practical improvement over the outer products formulation because the workspace is one dimensional, meaning that it can be implemented with a dense vector rather than a hash table. Note that workspaces are initialized just before executing the where statement that returns them.

5 Cost Modeling

In this section, we formalize the intuitions we gave in prior sections by describing a language for sparse asymptotic complexity. At a high level, we characterize the complexity of a kernel as the cardinality of the set of points in its iteration space, and reason about asymptotic domination by showing containment among iteration spaces as sets.

5.1 The Task Set Model

More precisely, we define our expressions of complexity as sets of **tasks**, where a task represents some constant-time unit of work associated with a point in the iteration space. We distinguish different tasks by the sets of indices that are in scope when they execute. For example, when $i = 3$, $j = 7$, and $k = 2$, and we execute the addition in $\forall_{ijk} a_i \mathrel{+}= B_{ij} \cdot C_{jk} \cdot d_k$, we incur a cost represented by the task $[3, 7, 2]$.

We represent a set of tasks over the indices $(i_1, ...i_r)$ with set builder notation

$$\{[i_1, ..., i_r] \in I_1 \times ... \times I_r \mid P\}$$

where P is a Boolean predicate expression over the indices. Our predicates will consist of clauses representing nonzero entries $A_{j...} \neq 0$ joined by logical conjunction ($\wedge$), logical disjunction ($\vee$), and the existential quantifier ($\exists$). These can be read as "and", "or", and "there exists", respectively. We may combine these set expressions with union ($\cup$) and intersection ($\cap$) operators. We will sometimes leave out the "$\neq 0$" for brevity. We use $i \in I$ as a shorthand for $i \in 1 : I$, but may omit the range of the index entirely for brevity.

The complexity of sparse kernels comes from two sources: computation tasks incurred in the numerical body of the loop, and coiteration tasks incurred due to iterating over multiple sparse tensor levels, since not all iterations lead to compute. Consider the sparse dot product $a \mathrel{+}= b^{\mathbf{c}}_{\mathbf{s}(i)} \cdot c^{\mathbf{c}}_{\mathbf{s}(i)}$.

for $i \in \mathcal{K}(b) \cup \mathcal{K}(c)$ **do**
  if $b_i \neq 0 \wedge c_i \neq 0$ **then**
    $a \leftarrow a + b_i \cdot c_i$

This code iterates over all the nonzeros in either b or c, but only multiplies the shared nonzeros. Thus, our dot product coiteration complexity is

$$\{[i] \in I \mid b_i \vee c_i\},$$

and the computation complexity is

$$\{[i] \in I \mid b_i \wedge c_i\},$$

Notice that the computation tasks are a subset of the iteration tasks. A sparse loop can function as a filter, considering many iterations but only executing the few relevant loop bodies. In our dot product example, the loop body is constant time. In other examples, such as SDDMM, the body of one loop might be another asymptotically significant loop. The same filtering effect seen in the dot product helps explain the difference between fused and unfused SDDMM, shown in Figure 2. By fusing the multiplications together, we can filter out irrelevant loops over k.

Existential quantifiers ($\exists$) are useful when we need to reason about whether loops contained in the body of another loop might be nontrivial. For example, the outermost loop of our fused SDDMM only processes a row i of D if there exists a nonzero in any column j_1 (if the row is nonempty).

Indices for dense loops do not appear in the predicate. For example, the index variable k in SDDMM does not participate in the predicates of the task sets, and is unconstrained. This is because the loops over k are dense, and iterate over all values of k. Adding an unconstrained k index is like multiplying the cardinality of the task set by K.

Our notation can also precisely characterize the differences between our example approaches to sparse matrix multiplication in Figure 3. Notice that the coiteration cost of the k loop in the inner products algorithm is the largest task set (it contains all the others). This is because it computes a sparse dot product for each row-column pair of A and B. This incurs a lot of filtering to determine which k values are shared. Gustavson's algorithm and the Outer Products algorithm are better choices because they move k to an outer loop, performing this filtering at a higher level of the loop nest. Compare the difference in coiteration and compute in our dot product example to that of the k loop and the compute in Inner Products.

$\forall i$	$\{[i] \mid \exists_{j_1} D_{ij_1}\} \cup$
$\forall j$	$\{[i,j] \mid D_{ij}\} \cup$
$\forall k$	$\{[i,j,k] \mid D_{ij}\} \cup$
$A_{ij} \mathrel{+}= B_{ik} \cdot C_{kj} \cdot D_{ij}$	$\{[i,j,k] \mid D_{ij}\}$

(a) Fused SDDMM

$\forall i$	$\{[i] \mid \exists_{j_1} D_{ij_1}\} \cup$
$\forall j$	$\{[i,j] \mid D_{ij}\} \cup$
$A_{ij} \mathrel{+}= w_{ij} \cdot D_{ij}$	$\{[i,j] \mid D_{ij}\}$
where	
$\forall i$	$\{[i]\} \cup$
$\forall j$	$\{[i,j]\} \cup$
$\forall k$	$\{[i,j,k]\} \cup$
$w_{ij} \mathrel{+}= B_{ik} \cdot C_{kj}$	$\{[i,j,k]\}$

(b) Non-Fused SDDMM

Figure 2. Some example implementations of SDDMM and the corresponding asymptotic costs for a DCSR matrices A and D and dense matrices B and C. Again, we have split the concrete index notation onto multiple lines to associate each statement with its corresponding task set. Notice that the triply nested loop is restricted to only the nonzero values of D in the unfused computation.

5.2 Accuracy of the Model

Any cost model is a balance between accuracy and complexity. Simple models may make unrealistic assumptions about the inputs and how they interact (such as nonzeros being uniformly distributed). More complex models might not lead to useful insights without extensive runtime information.

We chose our model because the cardinality of a task set is proportional, within constant factors, to the runtime of our pseudocode. Each operation takes constant time and is associated with one task. The number of different operations associated with each task is also bounded, since there are a finite number of operations at each level of each loop nest. This allows us to abstract over the various costs of different operations and the different times, or places, they may occur.

$$
\begin{aligned}
&\forall i && \{[i] \mid \exists_{k_1} B_{ik_1}\} \cup \\
&\quad \forall j && \{[i,j] \mid \exists_{k_1,k_2} B_{ik_1} \wedge C_{k_2 j}\} \cup \\
&\quad\quad \forall k && \{[i,j,k] \mid B_{ik} \vee C_{kj}\} \cup \\
&\quad\quad\quad A_{ij} \mathrel{+}= B_{ik} \cdot C_{kj} && \{[i,j,k] \mid B_{ik} \wedge C_{kj}\}
\end{aligned}
$$

(a) Inner Products SpGEMM

$$
\begin{aligned}
&\forall i && \{[i] \mid \exists_{k_1} B_{ik_1}\} \cup \\
&\quad \forall k && \{[i,k] \mid \exists_{j_1} B_{ik} \vee C_{kj_1}\} \cup \\
&\quad\quad \forall j && \{[i,j,k] \mid B_{ik} \wedge C_{kj}\} \cup \\
&\quad\quad\quad A_{ij} \mathrel{+}= B_{ik} \cdot C_{kj} && \{[i,j,k] \mid B_{ik} \wedge C_{kj}\}
\end{aligned}
$$

(b) Gustavson's SpGEMM

$$
\begin{aligned}
&\forall k && \{[k] \mid \exists_{i_1,j_1} B_{i_1 k} \vee C_{kj_1}\} \cup \\
&\quad \forall i && \{[i,k] \mid \exists_{j_1} B_{ik} \wedge C_{kj_1}\} \cup \\
&\quad\quad \forall j && \{[i,j,k] \mid B_{ik} \wedge C_{kj}\} \cup \\
&\quad\quad\quad A_{ij} \mathrel{+}= B_{ik} \cdot C_{kj} && \{[i,j,k] \mid B_{ik} \wedge C_{kj}\}
\end{aligned}
$$

(c) Outer Products SpGEMM

Figure 3. Some example implementations of SpGEMM and the corresponding asymptotic costs for DCSR matrices with step protocol. We split the concrete index notation across lines to associate each statement with its corresponding task set. Each loop iterates over a set of possible nonzero index locations, and only executes its body when the body might be nontrivial. For example, in the "Outer Products SpGEMM" kernel, the loop over k iterates over values of k that might be nonzero in either B or C, but only executes the body when the value of k is shared. Thus, the loop over nonzero i in B is also constrained to these shared values of k.

Our cost model does not consider hardware components like caches, vector units, and multicore or GPU processors. We consider these factors to be orthogonal to the decisions our asymptotic model empowers us to make. Our model represents the total amount of work done by a schedule. The total work is unchanged by cache blocking, loop unrolling, or parallelization and load balancing strategies, which only affect constant factors, and can likely be applied regardless of loop ordering. For example, all three variants of SpGEMM in Figure 3 can be parallelized. Thus, we leave tuning of hardware features as future work, but observe that the schedules produced by our work provide an excellent starting point.

Ignoring constant factors has consequences, of course, but they are limited. Even though one schedule might be asymptotically superior, it could be empirically outperformed by another on small inputs, by a limited constant factor. On the other hand, there may be a class of inputs on which our asymptotically superior schedule outperforms the other by an arbitrarily large factor as input grows, overcoming any constant-factor differences. Choosing an asymptotically inferior schedule could be a big mistake, while choosing the asymptotically superior schedule would at worst be a small mistake. In this sense, an asymptotically superior schedule can be considered to be safe. We leave the question of whether and when to consider asymptotically dominating schedules as future work.

In some cases, sparse tensor compilers use heuristic optimizations to avoid iterating over the full union of nonzero indices in a loop. For example, TACO stops a sparse dot product after the last nonzero in either vector is reached, regardless of whether the other vector has nonzeros [34]. Future tensor compilers might perform other heuristic improvements on intersections, for instance using a binary search to skip over several contiguous nonzeros when possible [9]. In these cases, our cost model should be considered an upper bound on the total work, which is sufficiently accurate in most practical cases [15]. We believe that accounting for these cases precisely would make our model too complicated to be useful, with little benefit. In fact, it might be impossible to choose between any of the costs. Optimizing for best case behavior under heuristic optimizations might result in a schedule which is vulnerable to worst-case inputs.

6 Asymptotic Domination

We use several algorithms for autoscheduling with our asymptotic cost model. First, we formalize what it means for one complexity to asymptotically dominate another. Then, Section 6.1 describes an algorithm to automatically check for domination. Section 6.2 describes ways to make our analysis more reflective of real world inputs and use cases, and Section 6.3 describes a way to filter out only the best complexities. Finally, Section 6.4 describes an algorithm to automatically determine the asymptotic complexity of a schedule, so that we might filter possible schedules by their complexity.

We say that a runtime f asymptotically dominates a runtime g with respect to a sequence of inputs x_n if for every constant $c > 0$, there exists m such that $f(x_n) > c \cdot g(x_n)$ for all $n > m$. This definition relies on the choice of inputs for the two algorithms whose runtimes we consider. For instance, when computing the expression $a_{ij} = b_{ij} \cdot c_{ij} \cdot d_{ij}$, one might compute $b_{ij} \cdot c_{ij}$ first, avoiding the need to traverse d when b and c are disjoint. However, a similar argument could be made for grouping c and d first.

We say that a task set f is asymptotically dominated by another task set g when f is contained in g but g is not contained in f, across all possible inputs.

We determine containment by converting our task sets to the standard "union of conjunctive queries" form [16, 35, 36], then using known approaches to check for set containment. In the language of the query containment literature, a conjunctive query is defined as the set of satisfying variable

assignments (expressed as a tuple of values) to an existentially quantified conjunction of Boolean functions on the variables. We write conjunctive queries as

$$\{(i_1, i_2 ...) \mid \exists_{j_1, j_2 ...} A_{k_1,k_2,...} \wedge B_{l_1,l_2,...} ...\}$$

The tuple $(i_1, i_2, ...)$ is referred to as the **head**, $A_{k_1,k_2,...}$, and $B_{l_1,l_2,...}$ are **clauses**, and our tensor variables $A, B, ...$ themselves are **predicates**. In our clauses, the predicates may be indexed by any combination of quantified or head variables.

As a running example, consider the inner products matrix multiply in Figure 3a. The techniques in this section will show how to prove that the coiteration on the k loop is the most frequently executed operation in the kernel, as its task set $\{[i, j, k] \mid B_{ik} \vee C_{kj}\}$ contains the other sets such as $\{[i, j] \mid \exists_{k_1,k_2} B_{ik_1} \wedge C_{k_2 j}\}$.

While tasks may be defined by different numbers of index variables in different orders, we would like a task set like $\{[i, j, k]\}$ to contain the task set $\{[j, i]\}$. We therefore interpret each task $[i...]$ as a shorthand for all tuples $(j...)$ where the variables $j...$ are a subset of the variables $i...$, arranged in any order. We also add existential quantifiers to any variables that are not present. Thus, we expand $\{[i, j] \mid \exists_{k_1,k_2} B_{ik_1} \wedge C_{k_2 j}\}$ into

$$\begin{aligned}
&\{(i, j) \mid \exists_{k_1,k_2} B_{ik_1} \wedge C_{k_2 j}\} \cup \\
&\{(j, i) \mid \exists_{k_1,k_2} B_{ik_1} \wedge C_{k_2 j}\} \cup \\
&\{(i) \mid \exists_{i,k_1,k_2} B_{ik_1} \wedge C_{k_2 j}\} \cup \\
&\{(j) \mid \exists_{j,k_1,k_2} B_{ik_1} \wedge C_{k_2 j}\} \cup \\
&\{() \mid \exists_{i,j,k_1,k_2} B_{ik_1} \wedge C_{k_2 j}\}.
\end{aligned}$$

Similarly, $\{(i, j) \mid \exists_k B_{ik} \vee C_{kj}\} \subset \{[i, j, k] \mid B_{ik} \vee C_{kj}\}$.

Our interpretation of tasks as queries must also respect the sizes of dimensions. Formally, we constrain the dimensions of head variables by interpreting each dimension itself as a Boolean function which is true when the index is in range. Thus, our query $\{(i, j) \mid \exists_k B_{ik} \vee C_{kj}\}$ is further expanded to $\{(i, j) \mid (\exists_k B_{ik} \vee C_{kj}) \wedge I[i] \wedge J[j]\}$. This is especially important when an index is unconstrained, or when we have more than one index of the same dimension (usually a result of symmetry). For instance, the task set $\{[i \in I, j \in I]\}$ would imply the query $\{(i, j) | I[i] \wedge I[j]\}$, of size I^2. For brevity, we will leave these dimension clauses out of subsequent query expressions. Notice that we do not need to worry about the dimension of existentially quantified variables since they do not contribute to the size of our task sets.

We can normalize to the union-of-conjunctive-queries form because our tasks have been defined as sets of tuples with Boolean predicates that contain only existential quantifiers, conjunctions, and disjunctions. Using De Morgan's law, we can convert conjunctions of disjunctions into disjunctions of conjunctions, and we can convert set expressions over disjunctions into unions of set expressions. We can move conjunctions and disjunctions inside of existential quantifiers by renaming the quantified variables.

While $\{(i, j) \mid \exists_{k_1,k_2} B_{ik_1} \wedge C_{k_2 j}\}$ is already in the appropriate form, we would rephrase $\{(i, j) \mid \exists_k B_{ik} \vee C_{kj}\}$ as $\{(i, j) \mid \exists_k B_{ik}\} \cup \{(i, j) \mid \exists_k C_{kj}\}$.

6.1 Domination Checking Algorithm

A conjunctive query P is contained in another query Q if and only if there exists a homomorphism h from the variables of Q to those of P [16]. A variable mapping $h : Q \to P$ is a homomorphism if applying h to the head of Q gives the head of P and if every clause $A_{k_1,k_2,...}$ in Q has a corresponding clause $A_{h(k_1),h(k_2),...}$ in P.

Furthermore, a union of conjunctive queries $P_1, P_2, ...$ is contained in a union of conjunctive queries $Q_1, Q_2, ...$ if each P_i is contained in at least one Q_j. This implies a straightforward algorithm for checking containment. For each conjunctive query in P, we attempt to find a conjunctive query in Q that contains it. We determine this containment by performing a backtracking search for homomorphisms, maintaining the partial homomorphism as we process each clause in turn. At each clause P_i, we make a choice of which clause Q_j will cover it. If we run into a variable conflict in the homomorphism, we backtrack and try a different Q_j. This algorithm might take exponential time, which is to be expected as query containment is an NP-hard problem. However, since our tensor programs are of small constant size and we are designing for an offline use case, the cost is often justified.

In our example queries, we can show $\{(i, j) \mid \exists_{k_1,k_2} B_{ik_1} \wedge C_{k_2 j}\} \subseteq \{(i, j) \mid \exists_k C_{kj}\}$ by mapping i to i, j to j, and k to k_2.

Recall that each task is semantically expanded into tuples of all subsets of its indices, and all permutations thereof. In order to avoid exponential increases in the input size of what is already an exponential time containment algorithm, we represent this expansion lazily. Thus, given two task sets $P = \{[i_1, i_2, ...] \mid A\}$ and $Q = \{[j_1, j_2, ...] \mid B\}$, we can say that $P \subseteq Q$ if our algorithm can find a homomorphism h from Q to P where $\{i_1, i_2, ...\} \subseteq \{h(j_1), h(j_2), ...\}$. It would be sufficient to perform a separate homomorphism search for each mapping of the head of P to a subset of the head of Q. However, since our homomorphisms must respect the dimension of head variables, we need only consider head variable mappings that preserve dimensions (e.g. mapping $i_1 \in I_1$ to $j_1 \in I_1$, and not $j_2 \in I_2$, for instance). To make this computation easier, our algorithms store the representation of dimension in the head of the query itself. This also helps to avoid cluttering the predicates with such extra clauses. These optimizations do not change our definitions of containment in any way. Rather, they make our algorithms practical by losslessly compressing the query representation and avoiding enumeration of an exponential number of invalid head variable mappings.

6.2 Sunk Costs and Assumptions

As presented so far, set containment analysis is more strict than what we would like. For example, in Figure 3, the set containment metric prefers the outer products algorithm

over Gustavson's algorithm when all matrices are DCSR, even though both algorithms are quite practical and Gustavson's requires less intermediate memory as it stores a temporary row or column instead of a temporary matrix. The reason is that when C is all zero, Gustavson's algorithm still iterates over the entirety of B whereas the outer products algorithm does not. After simplification, observe that substituting a zero C into the Gustavson's complexity gives $\{[i,k] \mid B_{ik}\}$, whereas substituting a zero C into the outer products complexity gives $\{[k] \mid \exists_{i_1} B_{i_1 k}\}$, a smaller set. Practitioners would say that this observation is not useful, since we should not expect operands to be all-zero, and even if this case did occur, simply iterating over the nonzeros of B is not a significant issue, as the user has likely already iterated over B at least once to store it in memory.

To account for these observations, we extend our analysis to reflect common sunk costs and assumptions. We use the term **sunk cost**, from the field of economics, to refer to a cost that has already been paid. Assume we are to compare two costs P and Q. If there are any sunk costs S (such as the linear-time costs of reading all the inputs), we add those costs to our queries and instead compare $P \cup S$ to $Q \cup S$. If there are any assumptions A, we add those assumptions to the predicates of our queries and compare $\{P \mid A\}$ to $\{Q \mid A\}$. Throughout the rest of the paper, we will take the time to read sparse (not dense) inputs and the time to iterate over any single dimension as sunk costs. We will also assume sparse inputs are nonempty.

6.3 Building a Frontier

Some schedules might be asymptotically better on some inputs, but not others. Our definition of asymptotic domination detects when one schedule is always better than another. We compare several different candidate kernels based on their asymptotic complexities, and use reciprocal containment checks to identify a frontier of non-dominating implementations. All kernels with the best complexity for any class of inputs are guaranteed to be in the frontier, which can be much smaller than the full universe of schedules. The asymptotic frontier for SpGEMM contains the outer product and Gustavson's algorithms, but not inner products since its complexity dominates the former two.

Our algorithm builds a frontier starting from empty and processing each kernel in turn. If the current kernel dominates any kernel in the frontier, then we discard it. Otherwise, we add the current kernel to the frontier and remove any other kernels that dominate it. This algorithm avoids a strictly quadratic number of containment checks by only comparing programs to the current frontier instead of the universe, but we cannot guarantee any bounds on the intermediate size of the frontier so this improvement is only heuristic. The algorithm is displayed in Figure 4.

Our algorithm performs asymptotically more pairwise complexity comparisons than the number of input programs.

$frontier \leftarrow []$
for $(S, P) \in universe$ **do**
  for $(T, Q) \in frontier$ **do**
    if $P \subseteq Q$ and not $Q \subseteq P$ **then**
      Add (S, P) to frontier if not added already
      Remove (T, Q) from frontier

Figure 4. Our algorithm for producing an asymptotic frontier. In practice, the size of the frontier does not grow too large, and the algorithm performs much less than the worst case quadratic number of asymptotic comparisons.

Therefore, it makes sense to simplify the complexity expressions to help improve the speed of the comparisons.

We add our sunk costs and assumptions to each runtime and normalize the resulting terms before running our frontier algorithm. This avoids renormalization during each comparison. Then, we reduce the size of our query through query minimization [16]. Assuming we have some union of conjunctive queries $Q_1 \cup Q_2 \cup ...$, we consider each Q_i. If it happens that $Q_i \subseteq Q_j$, then we can leave Q_i out. Let $Q_i = \{(k...) \mid C_1 \wedge C_2 \wedge ...\}$. We iteratively try to leave each clause C_l out in turn. If leaving C_l out does not change the query (recall that mutual containment is equality), we can safely ignore it. The overall process leaves us with a query of minimal size.

6.4 Automatic Asymptotic Analysis

With the goal of using our asymptotic analysis as part of an automated scheduler, we describe an algorithm to compute the asymptotic complexity of an input tensor program in protocolized sparse concrete index notation.

Our algorithm performs an abstract interpretation over each node of the program, using a Boolean predicate to describe the set of iterations currently being executed (referred to in our pseudocode as the "guard"). We also construct Boolean predicates that represent the nonzero locations written to during the course of executing the program (referred to as the "state"). When we write $state(A_i) \leftarrow state(A_i) \vee \exists_j B_{ij}$, we mean to update our state of A_i to add any nonzero locations in the pattern $\exists_j B_{ij}$. Because tensors may be written and read by index variables of different names, we rename head variables of states by mode number.

We start our traversal at the topmost node with no bound variables, a guard set to true, and a state filled with default inputs ($state(A_{i...}) = A_{i...}$ for all sparse inputs A).

The input sparse format usually does not add any information about the sparsity pattern. Regardless of first few level formats of a tensor, if the last level format is sparse, the whole tensor might have any possible sparsity pattern. The particular format would only clarify situations involving tensors with a dense final dimension, a rare case where rows are either entirely dense or entirely sparse. Therefore, we

only track the sparsity patterns for sparse inputs, knowing fully uncompressed inputs are always nonzero.

When we encounter a forall node over an index i, we collect all of the accesses in the body that access i with step protocol. Since our sparse program will coiterate through these tensors, we output the task set $\{[j...] \mid \exists_{k...} A_{l...}\}$ for each access $A_{l...}$, where j is the set of bound variables and $k \leftarrow l \setminus j$. Each access might be zero or nonzero. As it coiterates over all nonzeros in each of the tensors, our sparse code will only execute the body in cases where there are nonzero operands that need processing. Thus, our abstract interpretation iterates over every combination of zero or nonzero for each tensor, substituting the zeros into the body and simplifying before recursing. When we recurse, we use prior assumptions about zeroness as a guard on the set of iterations that each recursive call corresponds to. If no tensors access i with step protocol, then we can simply recurse on the body after adding i to the set of bound variables.

When we encounter a where node, we add a zero-initialized workspace to the state of the producer before processing the producer side, and we make that new tensor state available when we process the consumer side.

When we encounter an assignment statement $A_{i...}$ += ..., we output $\{[bound...] \mid guard\}$ to reflect the work performed by this statement. We also update the state of $A_{i...}$ to add in the writes represented by $guard$.

Our algorithm is summarized in Figure 5. A benefit of our algorithm to analyze complexity is that it will extend to alternate fill values and operators which are not + or ·, extensions to the TACO compiler explored in earlier work [31]. Nothing in our algorithm is specific to the choice of 0 or the operators we have chosen in our examples.

```
tasks ← ∅
state ← predicates initialized with input tensors
procedure Complexity(node, bound, guard)
    if node matches ∀_i body then
        steppers ← all step protocol accesses of i in body
        bound′ ← (bound..., i)
        cases ← {body}
        for A_{j...} ∈ steppers do
            expand cases by substituting A_{j...} → 0
            k ← (j \ bound′)
            iters ← {[bound′...] | guard ∧ ∃_{k...} state(A_{j...})}
            tasks ← tasks ∪ iters
        for body′ ∈ cases do
            simplify body′ using zero-annihilation
            guard′ ← guard
            for A_{j...} ∈ steppers do
                if A_{j...} is in body′ then
                    k ← (j \ bound′)
                    guard′ ← guard′ ∧ ∃_{k...} state(A_{j...})
            Complexity(body′, bound′, guard′)
    else if node matches cons where prod then
        state[result(prod)] ← 0
        Complexity(prod, bound, guard)
        Complexity(cons, bound, guard)
    else if node matches A_{i...} += ... then
        computes ← {[bound...] | guard}
        tasks ← tasks ∪ computes
        state(A_{i...}) ← state(A_{i...}) ∨ guard
end procedure
```

Figure 5. Our algorithm for analyzing the complexity of programs in protocolized concrete index notation. The *tasks* variable is a global that records the total complexity, and the *state* variable is a global dictionary that we use to hold the nonzero pattern of each tensor over the course of the program.

7 Autoscheduling

We use our asymptotic cost model to build an enumerative automatic scheduler. Our scheduler makes more coarse-grained decisions first, such as the structure of forall and where statements, working towards more fine-grained decisions such as the formats and protocols of the tensors and workspaces. Sections 7.1-7.9 each describe a stage in our autoscheduling pipeline. Our scheduler stops once we have asymptotically optimized sequential programs, but a more complete autotuner would make decisions regarding constant factors such as register or cache blocking and/or parallelization. Because we enumerate all possible choices at each stage of the pipeline, it is important to limit the number of choices we make at each stage, and filter candidate programs between stages so that the number of candidate programs does not grow too large. We will use the SpMV2 kernel (a repeated SpMV) on CSR matrices and a dense vector for the running example in this section. We write our CSR SpMV2 as

$$\forall_{ijk} a_i^{\mathbf{u}} \mathrel{+}= B_{ij}^{\mathbf{uc}} \cdot C_{jk}^{\mathbf{uc}} \cdot d_k^{\mathbf{u}}.$$

7.1 Enumerate Expression Rewrites

Our pipeline begins with a single pointwise index expression. We then consider associative and commutative rewriting transformations. These are rewriting rules of the form $(a + b) + c \to a + (b + c)$ or $(a + b) \to (b + a)$. If the expression contains + and ·, we might also consider distributive properties $(a + b) \cdot c \to a \cdot c + b \cdot c$ as well. The purpose of all this rewriting is to expose all grouping opportunities for the next stage. For example, we might first reassociate,

$$\forall_{ijk} a_i^{\mathbf{u}} \mathrel{+}= (B_{ij}^{\mathbf{uc}} \cdot C_{jk}^{\mathbf{uc}}) \cdot d_k^{\mathbf{u}} \to \forall_{ijk} a_i^{\mathbf{u}} \mathrel{+}= B_{ij}^{\mathbf{uc}} \cdot (C_{jk}^{\mathbf{uc}} \cdot d_k^{\mathbf{u}}),$$

then recommute,

$$\forall_{ijk} a_i^{\mathbf{u}} \mathrel{+}= B_{ij}^{\mathbf{uc}} \cdot (C_{jk}^{\mathbf{uc}} \cdot d_k^{\mathbf{u}}) \to \forall_{ijk} a_i^{\mathbf{u}} \mathrel{+}= B_{ij}^{\mathbf{uc}} \cdot (d_k^{\mathbf{u}} \cdot C_{jk}^{\mathbf{uc}}).$$

7.2 Enumerate Where Groupings

Once we have a set of possible expression trees, we enumerate nontrivial ways that where statements might be added to the expressions. These are transformations of the form

$$a_i \oplus\!= b_i \odot c_i \odot d_i \rightarrow a_i \oplus\!= b_i \odot w \textbf{ where } w \oplus\!= c_i \odot d_i,$$

where $\oplus$ distributes over $\odot$. We can also perform

$$a_i \oplus\!= b_i \odot c_i \odot d_i \rightarrow a_i \mathrel{+}= b_i \odot w \textbf{ where } w = c_i \odot d_i,$$

regardless of distributivity.

In our example, since + distributes over $\cdot$, we transform

$$\forall_{ijk} a_i^{\mathbf{u}} \mathrel{+}= B_{ij}^{\mathbf{uc}} \cdot (d_k^{\mathbf{u}} \cdot C_{jk}^{\mathbf{uc}}) \rightarrow \forall_{ijk} \left(a_i^{\mathbf{u}} \mathrel{+}= B_{ij}^{\mathbf{uc}} \cdot w \textbf{ where } w \mathrel{+}= d_k^{\mathbf{u}} \cdot C_{jk}^{\mathbf{uc}}\right).$$

Note that we do not add workspaces for single tensors, and only consider workspaces that record the result of a nontrivial operation between more than one tensor. When inserting workspaces at this stage, we do not yet need to consider the indices that the workspace access needs, as this will be derived later from the structure of foralls. We name the workspaces at this stage using De Bruijn indexing. After inserting where statements, our naming scheme allows us to normalize the pointwise expressions in each where and deduplicate our programs somewhat.

7.3 Enumerate Forall Nestings

Here, we consider all the different ways to move foralls from the outermost level into the pointwise expressions at the leaves of our program.

When we move an index into a where statement, it may need to move into the producer side, the consumer side, or both, depending on which side uses it and whether there is a reduction operator in the producer. If the producer has a reduction, then we move the loop into any side of the where that uses that index in an access. Otherwise, we move the loop into the consumer side always and into the producer side only if the producer uses that index in an access.

For instance, since the right hand side includes a reduction, we can move the k loop into just the producer side,

$$\forall_{ijk} (a_i^{\mathbf{u}} \mathrel{+}= B_{ij}^{\mathbf{uc}} + w \textbf{ where } w \mathrel{+}= C_{jk}^{\mathbf{uc}} + d_k^{\mathbf{u}}) \rightarrow \forall_{ij} ((a_i^{\mathbf{u}} \mathrel{+}= B_{ij}^{\mathbf{uc}} + w) \textbf{ where } (\forall_k w \mathrel{+}= C_{jk}^{\mathbf{uc}} + d_k^{\mathbf{u}})).$$

Then, since the i loop only occurs in the consumer side, we do not need to move it into the producer side,

$$\forall_{ij} ((a_i^{\mathbf{u}} \mathrel{+}= B_{ij}^{\mathbf{uc}} + w) \textbf{ where } (\forall_k w \mathrel{+}= C_{jk}^{\mathbf{uc}} + d_k^{\mathbf{u}})) \rightarrow \forall_j ((\forall_i a_i^{\mathbf{u}} \mathrel{+}= B_{ij}^{\mathbf{uc}} + w) \textbf{ where } (\forall_k w \mathrel{+}= C_{jk}^{\mathbf{uc}} + d_k^{\mathbf{u}})).$$

We could also move j into the where statement. We would have to move it into both sides since it is used on both.

$$\forall_j ((\forall_i a_i^{\mathbf{u}} \mathrel{+}= B_{ij}^{\mathbf{uc}} + w) \textbf{ where } (\forall_k w \mathrel{+}= C_{jk}^{\mathbf{uc}} + d_k^{\mathbf{u}})) \rightarrow ((\forall_{ji} a_i^{\mathbf{u}} \mathrel{+}= B_{ij}^{\mathbf{uc}} + w) \textbf{ where } (\forall_{jk} w \mathrel{+}= C_{jk}^{\mathbf{uc}} + d_k^{\mathbf{u}})).$$

After grouping the foralls, we enumerate all the different orderings of contiguous foralls (swapping $\forall_{ji}$ for $\forall_{ij}$ in our above example, for instance).

7.4 Filter by Maximum Nesting Depth

It is possible that our asymptotic cost model will not always recognize a triply-nested loop as dominating a double nested loop, because the triply-nested loop may perform better on specific sparsity patterns. However, we believe that most practitioners will want a program which performs well on relatively dense uniformly random inputs, so we restrict our focus to programs with minimum maximum nesting depth. We perform this filtering step as early in the pipeline as possible to reduce the burden on subsequent stages. There has been extensive research on heuristics to restrict the minimum maximum depth in tensor network contraction orders, but our focus is somewhat different [19, 27, 30]. We are interested in enumerating every program of minimum depth, and our programs are quite small. Most prior work focuses on finding a single minimum depth schedule for a very large network. Thus, we chose an enumerative approach.

At this step, we filter out depth-3 schedules such as

$$\forall_{ijk} a_i^{\mathbf{u}} \mathrel{+}= B_{ij}^{\mathbf{uc}} \cdot C_{jk}^{\mathbf{uc}} \cdot d_k^{\mathbf{u}},$$

in favor of depth-2 schedules like

$$\forall_j ((\forall_i a_i^{\mathbf{u}} \mathrel{+}= B_{ij}^{\mathbf{uc}} + w) \textbf{ where } (\forall_k w \mathrel{+}= C_{jk}^{\mathbf{uc}} + d_k^{\mathbf{u}})).$$

7.5 Name Workspaces and Indices

At this point, we can give our workspaces fresh names and compute the indices we need in their accesses. In the programs we have enumerated, a workspace linking the producer and consumer sides of a where statement needs to be indexed by the index variables shared by both the producer and consumer that are not quantified at the top of the where statement. If we are scheduling for TACO, we can remove workspaces with more than one dimension, since TACO does not support multidimensional sparse workspaces and dense multidimensional workspaces would be unacceptably large.

Notice that the workspace can be scalar in

$$\forall_j ((\forall_i a_i^{\mathbf{u}} \mathrel{+}= B_{ij}^{\mathbf{uc}} + w) \textbf{ where } (\forall_k w \mathrel{+}= C_{jk}^{\mathbf{uc}} + d_k^{\mathbf{u}}))),$$

but it needs to be a vector in

$$((\forall_{ji} a_i^{\mathbf{u}} \mathrel{+}= B_{ij}^{\mathbf{uc}} + w_j) \textbf{ where } (\forall_{jk} w_j \mathrel{+}= C_{jk}^{\mathbf{uc}} + d_k^{\mathbf{u}}))),$$

since the j is shared there.

7.6 Enumerate Protocols

At this stage, we normally enumerate all combinations of protocols for each mode for each access. However, since TACO does not support hash formats, almost any protocolization with more than two locates would require at least two uncompressed level formats, resulting in an unacceptable densification of the input. Thus, when we schedule for TACO, we protocolize with all step protocol or a single locate protocol at the first index to be quantified, the only two options that would not densify inputs. If we wanted to consider densification, we could add the induced storage overhead to the asymptotic runtime, but we reasoned that most practitioners would view these densifications as unacceptable.

One possible protocolization might look like

$$((\forall_{ji} a^{\mathbf{u}}_{i} \mathrel{+}= B^{\mathbf{uc}}_{ij} + w_j) \textbf{ where } (\forall_{jk} w_j \mathrel{+}= C^{\mathbf{uc}}_{jk} + d^{\mathbf{u}}_{k}))) \rightarrow ((\forall_{ji} a^{\mathbf{u}}_{\mathbf{i}(i)} \mathrel{+}= B^{\mathbf{uc}}_{\mathbf{l}(i)\mathbf{s}(j)} + w_{\mathbf{l}(j)}) \textbf{ where } (\forall_{jk} w_{\mathbf{a}(j)} \mathrel{+}= C^{\mathbf{uc}}_{\mathbf{l}(j)\mathbf{s}(k)} + d^{\mathbf{u}}_{\mathbf{l}(k)})))$$

7.7 Filter by Asymptotic Complexity

Finally, we can use our asymptotic cost model to filter the protocolized programs. Note that we assume that tensors will eventually be permuted to match the order in which index variables are quantified (concordant order). We also ignore storage formats at this stage, as we will add those in the next stage, and adding reformatting workspaces would needlessly complicate (but not change) the asymptotic complexity.

At this stage, our algorithm would choose for w to have locate protocol rather than step protocol in the kernel

$$((\forall_{ij} a^{\mathbf{u}}_{\mathbf{a}(i)} \mathrel{+}= B^{\mathbf{uc}}_{\mathbf{l}(i)\mathbf{s}(j)} + w_{\mathbf{l}(j)}) \textbf{ where } (\forall_{jk} w_{\mathbf{a}(j)} \mathrel{+}= C^{\mathbf{uc}}_{\mathbf{l}(j)\mathbf{s}(k)} + d^{\mathbf{u}}_{\mathbf{l}(k)})),$$

since the stepping over w involves merging nonzeros in w and each row of B, whereas locating over w means we only need to iterate over each nonzero in B.

7.8 Add Workspaces to Transpose and Reformat

At this stage, we can reformat tensors and add formats to workspaces so that the tensor access order is concordant (the level order of the sparsity tree matches the order in which indices are quantified). This step helps us realize the asymptotic complexity we assumed our kernels had in previous steps. Reformatting operations are achieved by inserting workspaces that have the proper format. Since reformatting takes linear time in the size of the tensor, we do not need to consider reformatting in our asymptotic complexity. We choose formats for workspaces based on the set of protocols they must support. For instance, a workspace that is written to via append protocol and read via step protocol can be stored in uncompressed format, but a workspace that is written with insert protocol and read via step protocol must be stored in hash format. In our above example, we can choose uncompressed format for our workspace since it is written with append and read with locate.

$$((\forall_{ij} a^{\mathbf{u}}_{\mathbf{a}(i)} \mathrel{+}= B^{\mathbf{uc}}_{\mathbf{l}(i)\mathbf{s}(j)} + w_{\mathbf{l}(j)}) \textbf{ where } (\forall_{jk} w_{\mathbf{a}(j)} \mathrel{+}= C^{\mathbf{uc}}_{\mathbf{l}(j)\mathbf{s}(k)} + d^{\mathbf{u}}_{\mathbf{l}(k)})).$$

As an empirical optimization to save memory, notice that if there is a set of index variables $i\ldots$ that are quantified at the top of the expression consuming a tensor, all accesses to that tensor begin with $i\ldots$, and the corresponding modes do not need reformatting or transposing, we can insert a workspace to reformat just the bottom modes of the tensor. We can do the same for expressions which produce a tensor.

Because TACO only supports uncompressed vector workspaces, we compile outermost reformatting workspaces as explicit transposition and reformatting calls, separate from the kernel. When compiling for TACO, we only perform our workspace-simplifying optimization when it results in a one-dimensional workspace. TACO also only supports a single internal workspace, so we filter kernels at this step with more than one workspace. TACO workspaces are accessed with step protocol when they are the only tensor being read, and with locate protocol otherwise.

7.9 Extensions and Empirical filtering

At this point, one could employ additional cost models and transformations to enhance and choose between the asymptotically good skeleton programs produced by the pipeline. Transformations like parallelization, cache blocking, or register blocking might be employed. One might imagine runtime autotuning approaches that use sparsity patterns to make better informed choices between the remaining programs. Since these transformations are out of scope, we simply run all programs in the frontier on uniformly sparse square inputs and pick the best-performing one.

8 Evaluation

We implemented our autoscheduler[1][2] using the Julia programming language [10] and the SymbolicUtils library [25]. We evaluate our approach using the TACO tensor compiler to run the generated schedules. We evaluated our approach on the kernels SpMV, SpMV2, SpMTTKRP, SpGEMM, SpGEMM2, and SpGEMMH, described in Figure 6. TACO does not implement protocolized concrete index notation in its full generality, so we also ran a separate autoscheduling algorithm on the subset of schedules supported by TACO. After running a warmup sample to load matrices into cache and JIT-compile relevant Julia code, all timings are the minimum of 10000 executions, or enough executions to exceed 5 seconds of sample time, whichever happens first. We ran our experiments on an 12-core Intel®Xeon®E5-2680 v3 running at 2.50GHz. Turboboost was turned off. The generated TACO

[1] https://github.com/peterahrens/Pigeon.jl/releases/tag/pldi2022
[2] https://zenodo.org/record/6366296

Kernel	Description	Min-Depth Schedules	Non-Dominating Schedules	Min-Depth Schedules (TACO)	Non-Dominating Schedules (TACO)	Asymptotic Filter Runtime (Seconds)
SpMV	$a_i = \sum B_{ij} \cdot c_j$	8	4	4	4	0.0211
SpMV^2	$a_i = \sum B_{ij} \cdot C_{jk} \cdot d_k$	144	28	24	24	0.211
SpMTTKRP	$A_{ij} = \sum B_{ikl} \cdot C_{jk} \cdot D_{jl}$	3631104	timed out	384	23	0.29
SpGEMM	$A_{ij} = \sum B_{ik} \cdot C_{jk}$	96	12	16	4	0.0517
SpGEMM^2	$A_{ij} = \sum B_{ik} \cdot C_{kl} \cdot D_{jl}$	20736	292	32	4	0.104
SpGEMMH	$A_{ij} = \sum B_{ik} \cdot C_{jk} \cdot D_{jk}$	102272	204	144	4	0.107

Figure 6. Our test kernels, along with their descriptions and several statistics about our autotuning process. The "Min-Depth Schedules" column describes the number of schedules (with protocols and loop ordering for all tensors) of minimum maximum loop nesting depth. The "Non-Dominating Schedules" column describes the size of the frontier after asymptotic filtering of min-loop-depth schedules. The "Min-Depth Schedules (TACO)" and "Non-Dominated Schedules (TACO)" columns are the same, but restrict schedules to what the TACO tensor compiler can generate. The asymptotic filter runtime is the average time (in seconds) to filter a single TACO-compatible min-loop-depth kernel. Filtering the full universe for SpMTTKRP timed out after a few days.

kernels were executed serially. We do not measure the time to transpose or reformat inputs or outputs.

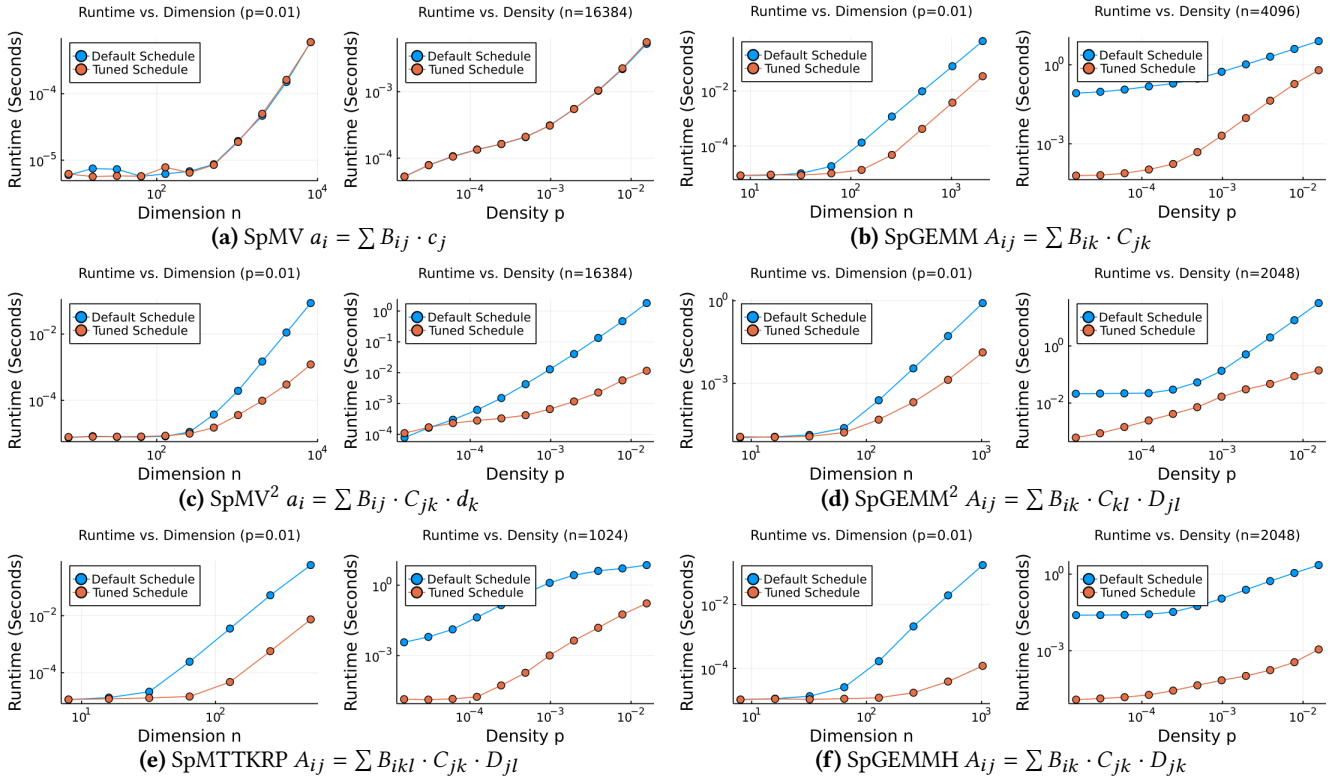
Kernel inputs used Compressed Sparse Fiber (CSF) format, meaning the first mode was dense and all subsequent modes were sparse. Vectors were therefore dense. All dimensions were the same size. Sparsity patterns were uniformly random. We benchmarked the asymptotic frontier on the same random sparse inputs ($\rho = 0.01$). We chose the dimensions by trying increasingly large powers of two, stopping when the default kernel exceeded 0.1 seconds of runtime.

Figure 6 contains statistics describing the size of the frontier before and after asymptotic filtering. As we can see, our asymptotic cost model was usually able to reduce the cardinality of the universe of min-loop-depth schedules by several orders of magnitude, even after restricting to TACO-compatible schedules. Our asymptotic filtering was able to process each candidate schedule in less than half a second (this number accounts for comparison between the candidate program and all other programs in the frontier, as well as the time required to simplify the asymptote during preprocessing). This time is independent of the size of the input to the program. To give a sense of scale for the runtime, running a single sparse matrix-vector multiply with TACO on a popular sparse matrix like "Boeing/ct20stif" of size $52,329 \times 52,329$ with $2,600,295$ nonzeros takes 9ms. When comparing so many schedules using empirical runtimes, it is important to ensure that results are statistically significant to avoid type 1 errors (false rejections of the null hypothesis that kernels perform identically). As the number of kernels grows, so too does the requisite number of samples required to compare them. If we wanted to run 100 trials on our single matrix, this would also take roughly a second per schedule. However, our method theoretically guarantees asymptotic domination across all inputs, whereas empirical evaluation is specific to the particular input matrix (or distribution of input matrices) under consideration. Furthermore, we can evaluate asymptotic domination before making choices about things like parallelization or cache blocking.

Figure 7 compares the autotuned schedules to the default schedules (created by nesting loops in alphabetical order of the indices). Figure 8 displays the autotuned schedules themselves. Since the default kernel for SpMV is already asymptotically optimal, we see no improvement. Our other autotuned kernels often improved on the defaults by several orders of magnitude, and we saw increasing speedups as the dimension increased. Speedups for SpMV^2 or SpGEMM^2 are likely due to a reduced loop nesting depth. These kernels saw speedups that increased with density, as the effects of loop depth become more obvious when more of the computation is dense. Speedups for SpGEMM, SpGEMMH and SpMTTKRP are likely due to improved loop ordering, moving filtering to an earlier step (the chosen algorithms are Gustavson-style). In these cases, speedups increased with sparsity, as the tuned kernels begin to asymptotically match the default when inputs become dense.

9 Related Work

Many autotuned libraries use runtime data to choose among a small set of possible programs [8]. Our approach is complementary, as it can automatically generate and prune this set of possible programs. The seminal ATLAS [23] and FFTW [24] libraries used empirical benchmarks to automatically choose between different kernels and to set parameters such as cache block or register block sizes in dense matrix algebra and Fourier transforms. The OSKI library [59] applied similar techniques to sparse matrix kernels, using sampling algorithms and affine cost models to set parameters. Runtime autotuning techniques subsequently enjoyed widespread adoption, allowing software developers to effectively specialize to new sparsity patterns [13, 41] and architectures [20, 62].



(a) SpMV $a_i = \sum B_{ij} \cdot c_j$ **(b)** SpGEMM $A_{ij} = \sum B_{ik} \cdot C_{jk}$

(c) SpMV^2 $a_i = \sum B_{ij} \cdot C_{jk} \cdot d_k$ **(d)** SpGEMM^2 $A_{ij} = \sum B_{ik} \cdot C_{kl} \cdot D_{jl}$

(e) SpMTTKRP $A_{ij} = \sum B_{ikl} \cdot C_{jk} \cdot D_{jl}$ **(f)** SpGEMMH $A_{ij} = \sum B_{ik} \cdot C_{jk} \cdot D_{jk}$

Figure 7. A comparison between the default schedule (nesting the loops in alphabetical order, without inserting any workspaces), and the schedule chosen by our autoscheduler on sparse inputs with uniformly random sparsity.

Kernel	Description	Chosen Schedule
SpMV	$a_i = \sum B_{ij} \cdot c_j$	$\forall_{ij} a^{\mathbf{u}}_{\mathbf{i}(i)} \mathrel{+}= B^{\mathbf{uc}}_{\mathbf{l}(i)\mathbf{s}(j)} \cdot c^{\mathbf{u}}_{\mathbf{l}(j)}$
SpMV^2	$a_i = \sum B_{ij} \cdot C_{jk} \cdot d_k$	$\forall_{ij} a^{\mathbf{u}}_{\mathbf{i}(i)} \mathrel{+}= B^{\mathbf{uc}}_{\mathbf{l}(i)\mathbf{s}(j)} \cdot w^{\mathbf{h}}_{\mathbf{l}(j)}$ **where** $\forall_{jk} w^{\mathbf{h}}_{\mathbf{a}(j)} \mathrel{+}= C^{\mathbf{uc}}_{\mathbf{l}(j)\mathbf{s}(k)} \cdot d^{\mathbf{u}}_{\mathbf{l}(k)}$
SpMTTKRP	$A_{ij} = \sum B_{ikl} \cdot C_{jk} \cdot D_{jl}$	$\forall_j \left(\forall_i A^{\mathbf{u}(2)\mathbf{c}(1)}_{\mathbf{a}(j)\mathbf{a}(i)} = w^{\mathbf{h}}_{\mathbf{s}(i)}$ **where** $\forall_{lki} w^{\mathbf{h}}_{\mathbf{i}(i)} \mathrel{+}= B^{\mathbf{u}(3)\mathbf{c}(2)\mathbf{c}(1)}_{\mathbf{l}(l)\mathbf{s}(k)\mathbf{s}(i)} \cdot C^{\mathbf{cc}}_{\mathbf{s}(j)\mathbf{s}(k)} \cdot D^{\mathbf{cc}}_{\mathbf{s}(j)\mathbf{s}(l)} \right)$
SpGEMM	$A_{ij} = \sum B_{ik} \cdot C_{jk}$	$\forall_i \left(\forall_j A^{\mathbf{uc}}_{\mathbf{a}(i)\mathbf{a}(j)} = w^{\mathbf{h}}_{\mathbf{s}(j)}$ **where** $\forall_{kj} w^{\mathbf{h}}_{\mathbf{i}(j)} \mathrel{+}= B^{\mathbf{uc}}_{\mathbf{l}(i)\mathbf{s}(k)} \cdot C^{\mathbf{u}(2)\mathbf{c}(1)}_{\mathbf{l}(k)\mathbf{s}(j)} \right)$
SpGEMM^2	$A_{ij} = \sum B_{ik} \cdot C_{kl} \cdot D_{jl}$	$\forall_i \left(\forall_{jl} A^{\mathbf{uc}}_{\mathbf{a}(i)\mathbf{a}(j)} \mathrel{+}= w^{\mathbf{h}}_{\mathbf{l}(l)} \cdot D^{\mathbf{cc}}_{\mathbf{s}(j)\mathbf{s}(l)}$ **where** $\forall_{kl} w^{\mathbf{h}}_{\mathbf{i}(l)} \mathrel{+}= B^{\mathbf{uc}}_{\mathbf{l}(i)\mathbf{s}(k)} \cdot C^{\mathbf{uc}}_{\mathbf{l}(k)\mathbf{s}(l)} \right)$
SpGEMMH	$A_{ij} = \sum B_{ik} \cdot C_{jk} \cdot D_{jk}$	$\forall_j \left(\forall_i A^{\mathbf{c}(2)\mathbf{c}(1)}_{\mathbf{a}(j)\mathbf{a}(i)} = w^{\mathbf{h}}_{\mathbf{s}(i)}$ **where** $\forall_{ki} w^{\mathbf{h}}_{\mathbf{i}(i)} \mathrel{+}= B^{\mathbf{u}(2)\mathbf{c}(1)}_{\mathbf{l}(k)\mathbf{s}(i)} \cdot C^{\mathbf{cc}}_{\mathbf{s}(j)\mathbf{s}(k)} \cdot D^{\mathbf{uc}}_{\mathbf{l}(j)\mathbf{s}(k)} \right)$

Figure 8. The schedules chosen by our autoscheduler on sparse inputs with uniformly random sparsity.

Although runtime autotuning adds overhead to tuned kernels, these techniques are efficient enough to be employed in production libraries such as Cray LibSci, NVIDIA CuFFT or Intel MKL [8]. Recently, sparse autotuning approaches have investigated machine learning techniques as a way to predict sparse format performance from the input patterns [47, 65], or as a better way to explore the optimization search space [2, 3, 45]. Libraries like OpenTuner [4], HyperMapper [46], and ATF [50] provide user-friendly interfaces to generic autotuning functions like search and benchmarking.

Moving to tensors, the Tensor Contraction Engine [6] compiled code for dense tensor contractions, using tensor dimensions to calculate and minimize the operation count of the implementation at runtime. This NP-hard minimization problem is the subject of continued research [19, 27, 30]. Our loop-depth minimization problem is similar and our brute force approach might be accelerated by related algorithms.

The Halide image processing compiler supports a wider variety of dense tensor expressions. Several machine learning based automatic scheduling approaches were proposed for

transformations like cache blocking or loop fusion, treating tensor dimensions as constants [2, 3, 45, 49]. The TVM library applies Halide's approach with additional architecture-specific features, and provides a machine-learning cost model for similar transformations [17]. Wang explored scheduling approaches for the orthogonal problem of caching, tiling, and loop reordering in TACO schedules [34, 61].

Set-based representations are often used to describe implementations of general loop-based programs. The most popular example is the polyhedral framework, which represents the iterations of dense loops with affine loop bounds as abstract geometric polyhedra, and often uses integer linear programming to optimize these representations based on the dimension sizes [7, 28, 55]. The polyhedral model has been extended to the sparse case by Strout et al. [52, 53]. Their sparse polyhedral representations are used to describe the schedule itself, and not the complexity of the schedule.

Moving beyond autotuning, some approaches more broadly categorized under the Inspector-Executor model consider transformations not easily categorized by a few categorical or continuous parameters [52]. For example, some approaches consider arbitrarily permuting the iteration order of individual loops to promote cache reuse [32], even reorganizing individual iterations within a linear solver to expose parallel regions [18, 44, 57]. These approaches are highly specialized to the runtime input patterns, but can access a finer-grained space of implementation decisions.

Query optimizers for databases use related techniques. [16, 26, 35, 36]. In queries, column stores would be analogous to sparse tensors, joins to multiplications, and our insertion of temporaries as query planning. Query minimization reduces the number of terms in conjunctive query expressions, but is not used in practice since queries are not always conjunctive and minimization can be too expensive and database queries are likely to be compiled and executed only once. Our solutions for sparse tensor applications can use involved techniques like query minimization because our kernels are small and are often reused several times. Join reordering, a more widely used technique, considers the database structure at runtime to determine the best places to insert temporaries. We cannot use most query planning techniques because they require runtime information.

Sparse tensor algebra and relational databases are closely related. Kotlyar et al. implemented sparse tensor algebra using relational databases [37–39]. More recently, Luo et al. supported dense linear algebra using databases [42]. Koutsoukos et. al. proposed the reverse, using TVM to support database queries [40], and Yuan et. al. combined the representation of relational queries and tensor algebra [64].

Database query optimization techniques have been used to optimize some matrix programs. Yu et al. proposed integrating sparse matrix operations into relational queries, and proposed a few techniques to optimize the resulting join structures [63]. Wang et al. use conjunctive queries to minimize the number of terms in matrix linear algebra expressions, then assume a uniform sparsity distribution to model and optimize the cost of the resulting expression with integer linear programming [60]. Both techniques focus on optimizing the composition of several matrix kernels, rather than the implementation of the kernels themselves, so we view them as solving a complementary higher-level problem of chaining together the small kernels which we optimize. We believe our work to be unique in using conjunctive queries to express the cost of a sparse kernel, rather than the value of the tensors in the kernel.

10 Conclusions and Future Work

Our cost model represents the first automatically derived expression of asymptotic complexity for sparse tensor programs. We describe algorithms to determine when one program asymptotically dominates another, and produce a frontier of non-dominating programs.

In the future, our offline techniques could enhance online techniques, using runtime data to choose among the templates produced by the autoscheduler and further enhance them with constant-factor techniques such as cache blocking or parallel load balancing strategies. When all the choices are between constant improvements, a suboptimal choice or even a bad choice can be tolerated, making heuristic or machine-learning approaches possible. However, in the sparse domain, selecting a asymptotically bad choice, even rarely, is unacceptable. Our scheduler can provide an asymptotically sound set of schedules for heuristic autotuners to make constant-factor improvements upon, eliminating worst-case scenarios. We believe that this symbiotic relationship will be essential for a successful autoscheduling solution in the sparse domain.

Future work might consider extensions to our cost model and its applications. Our task set model could be applied to memory usage. Similarly, one might model expensive operations (such as hash table operations) separately from cheaper operations, to obtain further stratification among programs. Our model can also be used to evaluate or guide heuristic or machine-learning sparse tensor autoschedulers.

Sparse tensor compilers cannot expect end users to schedule their own programs. Automatic scheduling is necessary to fully abstract the details of sparse tensor compilation. The offline asymptotic decision making described in this paper is the last piece of infrastructure needed to make sparse tensor compilation accessible to the mainstream.

Acknowledgments

This work was supported by a DOE Computational Science Graduate Fellowship, DE-FG02-97ER25308, NSF Grant IIP-2044424, and the Applications Driving Architectures (ADA) Center, a JUMP Center cosponsored by SRC and DARPA.

References

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*.

[2] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. 2019. Learning to optimize halide with tree search and random programs. *ACM Trans. Graph.* 38, 4 (July 2019). https://doi.org/10.1145/3306346.3322967

[3] Luke Anderson, Andrew Adams, Karima Ma, Tzu-Mao Li, Tian Jin, and Jonathan Ragan-Kelley. 2021. Efficient automatic scheduling of imaging and vision pipelines for the GPU. *Proc. ACM Program. Lang.* 5, OOPSLA (Oct. 2021). https://doi.org/10.1145/3485486

[4] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U. O'Reilly, and S. Amarasinghe. 2014. OpenTuner: An Extensible Framework for Program Autotuning. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. https://doi.org/10.1145/2628071.2628092

[5] Gilad Arnold. 2011. *Data-Parallel Language for Correct and Efficient Sparse Matrix Codes.* PhD Thesis. University of California, Berkeley.

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

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

[8] Prasanna Balaprakash, Jack Dongarra, Todd Gamblin, Mary Hall, Jeffrey K. Hollingsworth, Boyana Norris, and Richard Vuduc. 2018. Autotuning in High-Performance Computing Applications. *Proc. IEEE* 106, 11 (Nov. 2018). https://doi.org/10.1109/JPROC.2018.2841200

[9] Jérémy Barbay, Alejandro López-Ortiz, Tyler Lu, and Alejandro Salinger. 2010. An experimental investigation of set intersection algorithms for text searching. *ACM J. Exp. Algorithmics* 14 (Jan. 2010). https://doi.org/10.1145/1498698.1564507

[10] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. 2012. Julia: A Fast Dynamic Language for Technical Computing. *arXiv:1209.5145 [cs]* (Sept. 2012).

[11] Aart J. C. Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. 2022. Compiler Support for Sparse Tensor Computations in MLIR. *arXiv:2202.04305 [cs]* (Feb. 2022).

[12] Aart J. C. Bik and Harry A. G. Wijshoff. 1993. Compilation techniques for sparse matrix computations. In *Proceedings of the 7th international conference on Supercomputing*. https://doi.org/10.1145/165939.166023

[13] Aart J. C. Bik and Harry A. G. Wijshoff. 1994. Nonzero structure analysis. In *Proceedings of the 8th international conference on Supercomputing*. https://doi.org/10.1145/181181.181538

[14] Aart J. C. Bik and Harry A. G. Wijshoff. 1994. On automatic data structure selection and code generation for sparse computations. In *Languages and Compilers for Parallel Computing*. https://doi.org/10.1007/3-540-57659-2_4

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

[16] Ashok K. Chandra and Philip M. Merlin. 1977. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the ninth annual ACM symposium on Theory of computing*. https://doi.org/10.1145/800105.803397

[17] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*.

[18] Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2017. Sympiler: transforming sparse matrix codes by decoupling symbolic analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. https://doi.org/10.1145/3126908.3126936

[19] Lam Chi-Chung, P. Sadayappan, and Rephael Wenger. 1997. On Optimizing a Class of Multi-Dimensional Loops with Reduction for Parallel Execution. *Parallel Process. Lett.* 07, 02 (June 1997). https://doi.org/10.1142/S0129626497000176

[20] Jee W. Choi, Amik Singh, and Richard W. Vuduc. 2010. Model-driven autotuning of sparse matrix-vector multiply on GPUs. *SIGPLAN Not.* 45, 5 (Jan. 2010). https://doi.org/10.1145/1837853.1693471

[21] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format abstraction for sparse tensor algebra compilers. *Proc. ACM Program. Lang.* 2, OOPSLA (Oct. 2018). https://doi.org/10.1145/3276493

[22] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2020. Automatic generation of efficient sparse tensor format conversion routines. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/3385412.3385963

[23] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. 2001. Automated empirical optimizations of software and the ATLAS project. *Parallel Comput.* 27, 1 (Jan. 2001). https://doi.org/10.1016/S0167-8191(00)00087-9

[24] M. Frigo and S.G. Johnson. 1998. FFTW: an adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181)*, Vol. 3. https://doi.org/10.1109/ICASSP.1998.681704

[25] Shashi Gowda, Yingbo Ma, Alessandro Cheli, Maja Gwozdz, Viral B. Shah, Alan Edelman, and Christopher Rackauckas. 2021. High-performance symbolic-numerics via multiple dispatch. *arXiv:2105.03949 [cs]* (May 2021).

[26] Goetz Graefe. 1993. Query evaluation techniques for large databases. *ACM Comput. Surv.* 25, 2 (June 1993). https://doi.org/10.1145/152610.152611

[27] Johnnie Gray and Stefanos Kourtis. 2021. Hyper-optimized tensor network contraction. *Quantum* 5 (March 2021). https://doi.org/10.22331/q-2021-03-15-410

[28] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly — performing polyhedral optimizations on a low-level intermediate representation. *Parallel Process. Lett.* 22, 04 (Dec. 2012). https://doi.org/10.1142/S0129626412500107

[29] Fred G. Gustavson. 1978. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Trans. Math. Softw.* 4, 3 (Sept. 1978). https://doi.org/10.1145/355791.355796

[30] Albert Hartono, Qingda Lu, Thomas Henretty, Sriram Krishnamoorthy, Huaijian Zhang, Gerald Baumgartner, David E. Bernholdt, Marcel Nooijen, Russell Pitzer, J. Ramanujam, and P. Sadayappan. 2009. Performance Optimization of Tensor Contraction Expressions for Many-Body Methods in Quantum Chemistry. *J. Phys. Chem. A* 113, 45 (Nov. 2009). https://doi.org/10.1021/jp9051215

[31] Rawn Henry, Olivia Hsu, Rohan Yadav, Stephen Chou, Kunle Olukotun, Saman Amarasinghe, and Fredrik Kjolstad. 2021. Compilation of sparse array programming models. *Proc. ACM Program. Lang.* 5, OOPSLA (Oct. 2021). https://doi.org/10.1145/3485505

[32] Hwansoo Han and Chau-Wen Tseng. 2006. Exploiting locality for irregular scientific codes. *IEEE Transactions on Parallel and Distributed Systems* 17, 7 (July 2006). https://doi.org/10.1109/TPDS.2006.88

[33] Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, and Saman Amarasinghe. 2019. Tensor Algebra Compilation with Workspaces. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1109/CGO.2019.8661185

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

[35] Phokion G. Kolaitis and Moshe Y. Vardi. 2000. Conjunctive-Query Containment and Constraint Satisfaction. *J. Comput. System Sci.* 61, 2 (Oct. 2000). https://doi.org/10.1006/jcss.2000.1713

[36] George Konstantinidis and Jose Luis Ambite. 2013. Scalable containment for unions of conjunctive queries under constraints. In *Proceedings of the Fifth Workshop on Semantic Web Information Management - SWIM '13*. https://doi.org/10.1145/2484712.2484716

[37] Vladimir Kotlyar. 1999. *Relational Algebraic Techniques for the Synthesis of Sparse Matrix Programs*. PhD Thesis. Cornell.

[38] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. 1997. *Compiling parallel sparse code for user-defined data structures*. Technical Report. Cornell.

[39] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. 1997. A relational approach to the compilation of sparse matrix programs. In *Euro-Par'97 Parallel Processing*. https://doi.org/10.1007/BFb0002751

[40] Dimitrios Koutsoukos, Supun Nakandala, Konstantinos Karanasos, Karla Saur, Gustavo Alonso, and Matteo Interlandi. 2021. Tensors: an abstraction for general data processing. *Proc. VLDB Endow.* 14, 10 (June 2021). https://doi.org/10.14778/3467861.3467869

[41] Weifeng Liu and Brian Vinter. 2015. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. https://doi.org/10.1145/2751205.2751209

[42] Shangyu Luo, Dimitrije Jankov, Binhang Yuan, and Chris Jermaine. 2021. Automatic Optimization of Matrix Implementations for Distributed Machine Learning and Linear Algebra. In *Proceedings of the 2021 International Conference on Management of Data*. https://doi.org/10.1145/3448016.3457317

[43] John Michael McNamee. 1971. Algorithm 408: a sparse matrix package (part I) [F4]. *Commun. ACM* 14, 4 (April 1971). https://doi.org/10.1145/362575.362584

[44] Mahdi Soltan Mohammadi, Tomofumi Yuki, Kazem Cheshmi, Eddie C. Davis, Mary Hall, Maryam Mehri Dehnavi, Payal Nandy, Catherine Olschanowsky, Anand Venkat, and Michelle Mills Strout. 2019. Sparse computation data dependence simplification for efficient compiler-generated inspectors. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/3314221.3314646

[45] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.* 35, 4 (July 2016). https://doi.org/10.1145/2897824.2925952

[46] Luigi Nardi, Artur Souza, David Koeplinger, and Kunle Olukotun. 2019. HyperMapper: a Practical Design Space Exploration Framework. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. https://doi.org/10.1109/MASCOTS.2019.00053

[47] Israt Nisa, Charles Siegel, Aravind Sukumaran Rajam, Abhinav Vishnu, and P. Sadayappan. 2018. Effective Machine Learning Based Format Selection and Performance Modeling for SpMV on GPUs. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. https://doi.org/10.1109/IPDPSW.2018.00164

[48] William Pugh and Tatiana Shpeisman. 1999. SIPR: A New Framework for Generating Efficient Code for Sparse Matrix Computations. In *Languages and Compilers for Parallel Computing*. https://doi.org/10.1007/3-540-48319-5_14

[49] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/2491956.2462176

[50] Ari Rasch, Michael Haidl, and Sergei Gorlatch. 2017. ATF: A Generic Auto-Tuning Framework. In *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. https://doi.org/10.1109/HPCC-SmartCity-DSS.2017.9

[51] Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, Stephen Chou, Shoaib Kamil, Saman Amarasinghe, and Fredrik Kjolstad. 2020. A sparse iteration space transformation framework for sparse tensor algebra. *Proc. ACM Program. Lang.* 4, OOPSLA (Nov. 2020). https://doi.org/10.1145/3428226

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

[53] Michelle Mills Strout, Alan LaMielle, Larry Carter, Jeanne Ferrante, Barbara Kreaseck, and Catherine Olschanowsky. 2016. An approach for code generation in the Sparse Polyhedral Framework. *Parallel Comput.* 53 (April 2016). https://doi.org/10.1016/j.parco.2016.02.004

[54] Ruiqin Tian, Luanzheng Guo, Jiajia Li, Bin Ren, and Gokcen Kestor. 2021. A High-Performance Sparse Tensor Algebra Compiler in Multi-Level IR. *arXiv:2102.05187 [cs]* (Feb. 2021).

[55] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *arXiv:1802.04730 [cs]* (June 2018).

[56] Anand Venkat, Mary Hall, and Michelle Strout. 2015. Loop and data transformations for sparse matrix code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/2737924.2738003

[57] Anand Venkat, Mahdi Soltan Mohammadi, Jongsoo Park, Hongbo Rong, Rajkishore Barik, Michelle Mills Strout, and Mary Hall. 2016. Automating Wavefront Parallelization for Sparse Matrix Computations. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. https://doi.org/10.1109/SC.2016.40

[58] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C. J. Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, and Paul van Mulbregt. 2020. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nat Methods* 17, 3 (March 2020). https://doi.org/10.1038/s41592-019-0686-2

[59] Richard W. Vuduc. 2004. *Automatic performance tuning of sparse matrix kernels*. Ph. D. Dissertation. University of California.

[60] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. 2020. SPORES: sum-product optimization via relational

equality saturation for large scale linear algebra. *Proc. VLDB Endow.* 13, 12 (Aug. 2020). https://doi.org/10.14778/3407790.3407799

[61] Ziheng Wang. 2020. *Automatic optimization of sparse tensor algebra programs.* Thesis. Massachusetts Institute of Technology.

[62] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2007. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. https://doi.org/10.1145/1362622.1362674

[63] Yongyang Yu, Mingjie Tang, and Walid G. Aref. 2021. Scalable Relational Query Processing on Big Matrix Data. *arXiv:2110.01767 [cs]* (Oct. 2021).

[64] Binhang Yuan, Dimitrije Jankov, Jia Zou, Yuxin Tang, Daniel Bourgeois, and Chris Jermaine. 2021. Tensor relational algebra for distributed machine learning system design. *Proc. VLDB Endow.* 14, 8 (April 2021). https://doi.org/10.14778/3457390.3457399

[65] Yue Zhao, Jiajia Li, Chunhua Liao, and Xipeng Shen. 2018. Bridging the gap between deep learning and sparse matrix format selection. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. https://doi.org/10.1145/3178487.3178495