Unified Compilation for Lossless Compression and Sparse Computing

by

Daniel Donenfeld

B.S., Cornell University (2017)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2023

Author .....
Department of Electrical Engineering and Computer Science
January 25, 2023

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

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Unified Compilation for Lossless Compression and Sparse Computing

by

Daniel Donenfeld

Submitted to the Department of Electrical Engineering and Computer Science
on January 25, 2023, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Achieving high performance for computations on tensors depends heavily on the formats used to store them. While sparse tensors are very common, there are more general patterns in data which can sometimes be better captured using lossless compression.

We show how to extend sparse tensor algebra compilers to support lossless compression techniques, including variants of run-length encoding and Lempel-Ziv compression. We develop new abstractions to represent losslessly compressed data as a generalized form of sparse tensors, with repetitions of values (which are compressed out in storage) represented by non-scalar, dynamic fill values. We then show how a compiler can use these abstractions to emit efficient code that computes on losslessly compressed data. By unifying lossless compression with sparse tensor algebra, our technique is able to generate code that computes with both losslessly compressed data and sparse data, as well as generate code that computes directly on compressed data without needing to first decompress it.

We evaluate two implementations of our techniques, using a prototype compiler based on TACO, and an implementation of our formats within Finch. Our evaluation using our TACO compiler shows our technique generates efficient image and video processing kernels that compute on losslessly compressed data. We find that the generated kernels are up to 16.3× faster than equivalent dense kernels generated by TACO, a tensor algebra compiler, and up to 16.1× faster than OpenCV, a widely used image processing library. Using our Finch formats, we see compression ratios up to 25× with run-time speedups up to 3.1× over dense computation for reduction computations.

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

Acknowledgments

This thesis includes work published in [20]. I would like to thank my advisor Professor Saman Amarasinghe for all his guidance. He has kept me on the right track when getting into the details, and encouraged me to think bigger when discussing ideas and future directions.

I would also like to thank Stephen Chou, whose mentorship and help made this project possible. His deep knowledge of TACO, tensor computations, and formats was invaluable and it was great working with him.

I would like to thank Willow Ahrens for many discussions on Tensor algebra, and helping with many technical questions around both Finch and Julia.

I would also like to thank all of the other members of the COMMIT group for their camaraderie. Especially with starting Graduate school during a pandemic, it was great to have virtual events which gradually have changed back to in-person.

I would like to thank my parents and brothers for all their love and support, and always asking about my research and being genuinely interested, even when my descriptions start out too technical.

Last but not least, I'd like to thank my wife Jamie for encouraging me to start this journey and for all of the unending love and support.

Contents

List of Figures

Chapter 1

Introduction

Data, either extracted from nature or artificially generated, are seldom random but often contain repeated patterns. Two distinct approaches, namely lossless data compression and sparse programming, have evolved over the years to take advantage of such repeated patterns, enabling large data sets to be transmitted, stored, and computed on efficiently while fully preserving the integrity of the data.

Lossless compression techniques work by using shorter code words to represent repeated patterns in the input, thus preventing them from having to be redundantly stored. Examples of lossless compression techniques include run-length encoding (RLE) and Lempel-Ziv (LZ77) compression [56], which are building blocks in many commonly used data formats such as PNG for images and ZIP for archive files.

By contrast, sparse programming, which is extensively used in linear/tensor algebra and array computing, exploits the fact that many tensors/arrays representing natural or synthetic data contain mostly zeros (i.e., are sparse). Sparse programming systems can exploit this property to reduce storage cost by storing sparse tensors in specialized data formats like compressed sparse row (CSR) [53] and compressed sparse fiber (CSF) [47], which make the zeros implicit and only explicitly store nonzero entries. Furthermore, sparse programming can reduce the cost of computing with large data sets by orders of magnitude by also exploiting algebraic properties of the computation. For instance, since multiplying any value by zero yields zero, multiplying two large sparse tensors can be done efficiently by only accessing and computing with

the nonzero entries of the two tensors.

Unfortunately though, lossless compression and sparse programming techniques have developed largely independently, and consequently existing libraries and frameworks that utilize these techniques suffer from various limitations. For one, except for in a few domain-specific cases, existing systems that utilize lossless compression techniques haven't progressed to directly compute on compressed data; instead, they must first decompress the data before computing with it. Meanwhile, existing sparse programming systems, including hand-implemented libraries like Intel MKL [27] and compilers like TACO [30, 16, 25], cannot efficiently store and compute with data that have many different repeated nonzeros, since sparse data representations only compress out zeros. Furthermore, existing systems cannot simultaneously compute with losslessly compressed data representations (like RLE and LZ77) and sparse data representations (like CSR and CSF) efficiently. A programming system that addresses all these limitations and that can efficiently compute with both sparse and compressed data requires well-optimized kernels to perform the computations. Such kernels are difficult and tedious to implement and optimize by hand, since they are typically much more complex than what are needed to perform the same computations with uncompressed data.

In this thesis, we propose a compiler technique to automatically generate efficient code that directly perform user-specified (tensor algebra) computations on any combination of losslessly compressed and sparse inputs on arbitrary types. The key idea behind our technique is to generalize the notion of sparsity by allowing different regions of a tensor to have different values that are treated similar to "zeros" (i.e., fill values) and compressed out in storage. We show how variants of algorithms like RLE and LZ77 can be viewed as sparse tensor formats that support this expanded notion of sparsity, allowing a compiler to reason about lossless compression techniques in exactly the same way as more typical sparse tensor representations like CSR. This, in turn, lets the compiler use the same mechanism to generate efficient code for computing with compressed data as well as to generate code for computing with sparse data. Specifically, our contributions include:

- A generalized notion of sparsity that allows repeated sequences of nonzeros to be compressed out (i.e., *fill regions*) and that allows a sparse tensor to have different fill values in different regions of the tensor (i.e., *dynamic fills*);
- New abstractions that capture lossless compression algorithms like RLE and LZ77 as sparse tensor representations that support dynamic fill regions; and
- A unified mechanism for generating efficient code that directly compute with losslessly compressed data and sparse data.

We implement our technique, which generalizes those described in [25] and [16], first as a prototype extension to the TACO sparse tensor algebra compiler. Our evaluation shows that our technique generates code that are up to 16× faster than both TACO-generated dense kernels and OpenCV [15]. While computing directly on compressed data is sometimes slower than processing densely stored data, we see that the former approach yields end-to-end speedups over the latter approach in all but one case (where the performance is equivalent), as the latter approach incurs overhead for decompressing and recompressing data. We also include implementations of our RLE formats within the Finch [7] compiler for the Looplets language. The Looplets language further generalizes our abstractions, and more allows the optimizations we describe to be more easily implemented. We show how our formats enable compression ratios up to 25× and run-time speedups up to 3.1× over dense computation for reduction computations.

Chapter 2

Background

We briefly describe lossless compression and sparse programming, which are two distinct approaches for efficiently storing and computing with large data sets that contain repeated patterns. We also provide an overview of the TACO sparse tensor algebra compiler, which our technique extends.

2.1 Lossless Compression

Lossless compression algorithms, such as RLE and LZ77 compression, work by using shorter code words to represent repeated patterns in the input, thus preventing the repeated patterns from having to be redundantly stored.

2.1.1 RLE

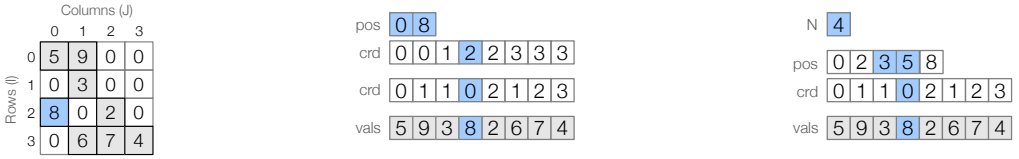
RLE encodes contiguous sequences of repeated values as a single copy of the repeated value along with a count of how many times the value is repeated. Thus, a sequence such as `1,1,1,3,3,3,3` can be encoded using RLE as $\langle$`1,3`$\rangle$,$\langle$`3,4`$\rangle$, which is a more compact representation. In this example, the count is stored explicitly, however there is significant flexibility in how the data and run lengths can be stored, as will be discussed when introducing our variants of RLE in Section 3.2.1 and Section 3.2.2. While RLE is a conceptually simple technique, it is not a single format, as there are

multiple ways to represent both the values and runs.

2.1.2 LZ77

LZ77 generalizes RLE by allowing repetitions of multi-valued sequences to be efficiently encoded. Data encoded using LZ77 consists of two kinds of tokens: *value tokens* and *repeat tokens*. When decoding LZ77-compressed data, value tokens, which store uncompressed subsets of the original data, are directly copied to the output. On the other hand, a repeat token $\langle$`c,d`$\rangle$, which consists of a count c and a distance d, is decoded by copying c values starting at an offset of d values from the end of the partially decoded output. A key aspect of the LZ77 algorithm is that d can be smaller than c in a repeat token, which implies that the sequence of values starting from offset d is repeated until c values are copied to the output. Thus, a sequence such as `1,2,3,1,2,3,1,2,3` can be encoded using LZ77 as `1,2,3,`$\langle$`6,3`$\rangle$, with all but the first occurrence of the sequence `1,2,3` encoded as repeats.

Though most existing frameworks that utilize lossless compression work with compressed data by first decompressing the data before computing with it, [52] describes an approach for performing streaming computations directly on LZ77 compressed data. The key idea behind the approach is that many computations preserve repetitions that exist in the input, so one can avoid recomputing with repeated data by copying repetitions directly into the output. So to increment every element in the LZ77-compressed sequence `1,2,3,`$\langle$`6,3`$\rangle$, for instance, one can directly compute on the compressed representation by simply incrementing the value tokens and copying over the repeat token. This produces the output sequence `2,3,4,`$\langle$`6,3`$\rangle$, which correctly encodes the result of the computation as if it was performed on the decompressed input and then recompressed.



(a) COO (b) CSR

Figure 2-1: The same tensor stored in two different formats.

2.2 Sparse Programming

Sparse programming systems, on the other hand, are optimized to compute with tensors (multidimensional arrays) that contain mostly repeated zeros by storing such sparse tensors in specialized formats that avoid materializing the zeros. There exists many formats for storing sparse tensors, including CSR, CSF, and the coordinate format (COO) [10], as illustrated in Figure 2-1. These formats use different data structures to store coordinates of nonzeros and differ in how stored values can be efficiently iterated and accessed, but all of these formats share the key characteristic that only nonzero entries are explicitly stored in memory. Sparse programming systems exploit this characteristic along with algebraic properties of the computation (such as the fact that multiplication by zero always yields zero) in order to avoid computing with zeros, thereby minimizing execution time. Sparse tensor formats also reduce data movement as only nonzeros and their coordinates are loaded from memory.

While most existing sparse linear/tensor algebra libraries only support sparse tensors that have zeros as fill values (i.e., the compressed out values), some sparse programming systems, such as GraphBLAS [37, 28, 17] and TACO, support an extended notion of sparsity where any value can be the fill value. A single value—the fill value—across the entire data can be elided. Under this model, one can also optimize computations other than multiplication or addition when the fill value of the sparse operands equals the computation's annihilator (i.e., any value that, when operated on, produces itself as the result). For example, computing the element-wise maximum of two sparse tensors that have ∞ as fill values can be done by only accessing and computing with the finite entries of the tensors, since the max function has ∞ as its annihilator (i.e., $\max(\infty, c) = \infty$ for any c).

(a) COO (b) CSR

Figure 2-2: Coordinate hierarchy representations of the same sparse tensor stored in two different formats.

2.3 Sparse Tensor Algebra Compilation

Chou et al. [16] describe how sparse tensors stored in different formats can be represented by *coordinate hierarchies* with varying structures that capture how stored nonzeros are physically organized and encoded in memory. Figure 2-2 shows coordinate hierarchies that represent the same tensor stored in two different formats. Each level in a coordinate hierarchy encodes stored coordinates along one dimension of the tensor, and each path from the root to a leaf of the coordinate hierarchy represents a stored nonzero.

Under the coordinate hierarchy abstraction, sparse tensor formats can be decomposed into *level formats* that each stores a level of a coordinate hierarchy. As Figure 2-3 illustrates, for instance, the CSR format can be expressed as a composition of the *dense* level format, which stores the row dimension of a matrix, and the *compressed*[1] level format, which stores the column dimension. Different level formats may use different data structures to store tensor dimensions. The dense level format, for example, encodes coordinates along a dimension as a contiguous range from 0 to N. By contrast, the compressed level format stores coordinates of nonzeros in segments of a `crd` array, with the bounds of each segment encoded in a `pos` array. However, all level formats implement the same interface, which exposes the level format's *capabilities* as sets of *level functions* that describe how underlying data structures can be accessed. For instance, the compressed level format supports the *coordinate position iteration* capability and the *coordinate append* capability. The coordinate position iteration

[1]The name of the compressed level format does not imply that it utilizes (lossless) compression, but rather that coordinates of zero entries are omitted.

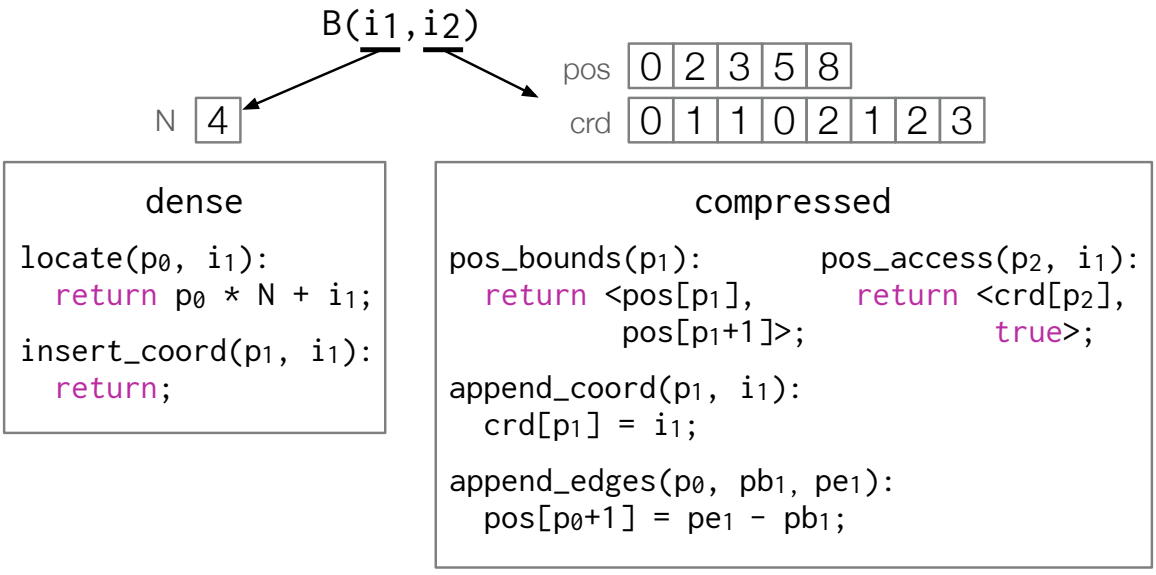


Figure 2-3: Decomposition of CSR into level formats, and corresponding level functions that specify how the associated data structures can be efficiently accessed and assembled.

capability is implemented by two level functions (`pos_bounds` and `pos_access`) that together describe how coordinates stored consecutively within a coordinate hierarchy level can be efficiently iterated. Similarly, the coordinate append capability is implemented by a set of level functions (`append_coord` and `append_edges`) that together describe how coordinates of nonzeros can be appended to a coordinate hierarchy level.

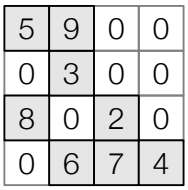
The coordinate hierarchy abstraction lets a compiler generate efficient code to compute with sparse tensors in arbitrary formats, without any of the formats being hard-coded into the compiler. In particular, the compiler can first emit code that invokes the format's capabilities in order to traverse coordinate hierarchies that represent the operands. Then, invocations of those capabilities can simply be replaced by the operand format's implementations of the capabilities, resulting in code that is specialized to the operands' formats.

Chapter 3

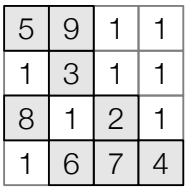
Representing Generalized Fill Values

As mentioned previously though, existing sparse tensor algebra compilers such as TACO only effectively support data representations like CSR and COO that compress out a single fill value. However, as Figure 3-1 illustrates, tensors that arise in many application domains often contain multiple distinct values (or even sequences of values) that are repeated. Animated videos and cartoon images, for instance, often contain many regions of duplicated pixels, with each region having pixels of a different color.

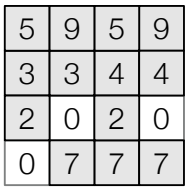
In this section, we first propose a generalization of fill values that can more efficiently encode repetitions of multiple distinct values in sparse tensors. We further propose new level formats that use variations of RLE and LZ77 in order to losslessly compress stored values, and we show how these level formats can be viewed as formats that efficiently store generalized fill values. Finally, we describe an extension to the level format abstraction described in [16] that fully captures how generalized fill values stored in our new level formats can be efficiently accessed and modified. Our technique is applicable to both integral and floating-point data, though it is better suited to integral data since small fluctuations in otherwise identical floating-point values can result in little or no exact repetition.



(a) Sparse tensor with mostly zeros

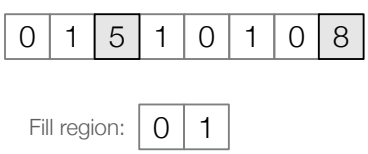


(b) Sparse tensor with mostly ones

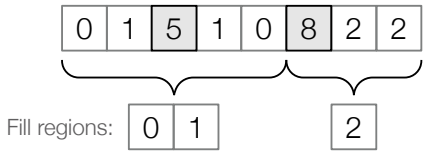


(c) Dense tensor with repeated values/sequences

Figure 3-1: Tensors that represent real-world data may contain various kinds of repetitions. Existing sparse tensor algebra compilers like TACO can efficiently handle sparse tensors that contain mostly any single specific value (zero or nonzero) but cannot efficiently work with tensors that contain many distinct repeated (sequences of) values.



(a) Fill regions



(b) Dynamic fills

Figure 3-2: Our generalization of fill values supports non-scalar fills (fill regions) and different fills for different parts of a tensor (dynamic fills).

3.1 Non-Scalar and Dynamic Fills

While sparse tensors are typically modeled as consisting of a single value (i.e., the fill value) that can be compressed out in storage, we generalize this model in two ways by introducing the concepts of *fill regions* and *dynamic fills*.

3.1.1 Fill Regions

Fill regions generalize the notion of sparsity by allowing for non-scalar fills. Conceptually, a fill region is a simply a sequence of values that is tiled over an entire tensor. At coordinates where there are no other explicitly defined values, the tensor assumes the values of the fill region. The number of values in a fill region is referred to as the fill region's *size*. (A scalar fill value can be viewed as a fill region of size 1.) Figure 3-2a shows an example of a sparse tensor with a fill region of size 2 and illustrates how sequences of values can be replicated across tensors as fill regions.

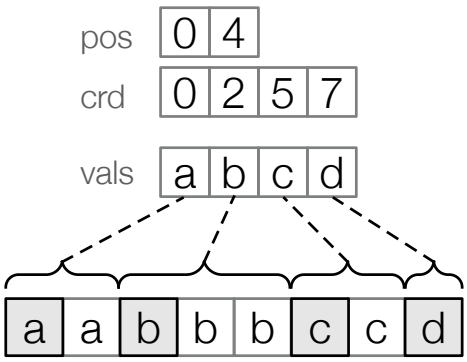


Figure 3-3: An example of a vector stored in the RLE level format, with each run of identical elements represented as a defined value followed by fills that have the same value. This variant of RLE explicitly stores the start and end coordinates of each run (other than the end coordinate of the last run, which is assumed to be the size of the stored dimension). The length of each run can be computed by taking the difference between the start and end coordinates.

3.1.2 Dynamic Fills

Dynamic fills generalize the notion of sparsity by allowing for different parts of a tensor to have different fill values (or, more generally, fill regions). Conceptually, a sparse tensor with dynamic fills can be represented as a set of defined values in the tensor and a map from subsets of the tensor to their corresponding fill values/regions. Figure 3-2b shows an example of a sparse tensor that contains two distinct fill regions.

3.2 Lossless Compression as Level Formats

As mentioned previously, RLE and LZ77 are two examples of commonly used lossless compression algorithms. We propose three new level formats that implement variants of these algorithms, and we show how these level formats can be viewed as storing tensors with generalized fills.

3.2.1 Run Length Encoded (RLE)

Run length encoding formats typically explicitly store the run-length associated with each value, either in a single data stream, or with the values and run lengths stored separately. In Figure 3-3, we demonstrate a variant of RLE as a level format that can

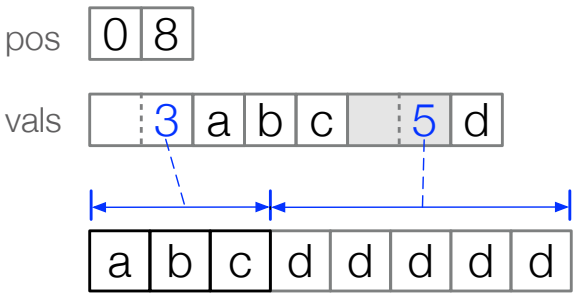


Figure 3-4: An example of a vector stored in the Pack-RLE format. Raw values are represented as defined values, and repetitions, encoded using a repeat token, are represented as fill values.

efficiently store a one-dimensional tensor (vector) containing many distinct runs of repeated values. This level format uses the same data structures as the compressed level format (in particular, the `crd` and `pos` arrays) to store the coordinates of defined values. In contrast to the compressed level format though, which simply interprets each stored coordinate (and associated value) as a nonzero, the RLE level format additionally interprets each stored coordinate as a point in the tensor where the fill value changes to being the stored value. This defines the run-lengths implicitly, using the coordinates from the `crd` array to store when the value stored changes, instead of storing an explicit length. When iterating over a tensor stored in our RLE format, the value at each non-defined coordinate can then be assumed to be the last explicitly-stored value that was accessed. In this way, distinct runs of repeated values stored in our RLE format can simply be interpreted as dynamic fill values.

The RLE level format can be used to represent any dimension of a tensor. When used to store the innermost dimension of a tensor, the level format efficiently stores repetitions of scalar fill values. When used to store other dimensions, however, the level format can efficiently store repetitions of fill regions. For example, color images can be viewed as $W \times H \times 3$ tensors, with the innermost dimension representing the three color values for each pixel. By storing the H dimension as RLE, one can efficiently represent repetitions of entire pixels instead of just individual color values.

3.2.2 RLE Variant (Pack-RLE)

The level format described in Section 3.2.1 is one possible implementation of RLE, however it does not have the best performance in all cases. For example, when there a fewer runs in data, a format which can store sequences of incompressible values can have a better compression ratio by avoiding the storage of many coordinate values. We implemented a second RLE format named *Pack-RLE*, based on the Packbits format [4], which can store both incompressible sequences and runs.

Packbits uses a single header byte to distinguish between sequences of uncompressed bytes and a repeated byte. This format is limited to compressing sequences of bytes with sequences or runs of length 128. We extend this format, shown in Figure 3-4, by using a two byte header, which encodes the length of either a run or sequence using the number of values. This allows our format to work for any type, and to represent longer sequences and runs when compared to Packbits. A high bit of zero in the header indicates the remaining bytes encode a count n, where the following n elements are distinct uncompressed values. A high bit of one in the header encodes the run length c using the remaining bits of the following element following the header. As shown in Figure 3-4, each run can be interpreted as a point in the tensor where the fill region dynamically changes to the repeated element. The uncompressed sequences are then defined entries in the tensor.

3.2.3 LZ77

Figure 3-5 shows how a level format that implements a variant of LZ77 can efficiently store a vector that contains many repeated sequences of values. The level format stores both values and repeats as sequences of elements within the values array. As with the RLE format described in Section 3.2.2, our LZ77 format uses a two byte header to differentiate between uncompressed sequences and repeated values. Raw values are represented by a two-byte element that has a high bit of zero and that encodes a count n using the remaining bits, followed by n elements that each stores a distinct uncompressed value. On the other hand, each repeat token $\langle$`c,d`$\rangle$ is repre-

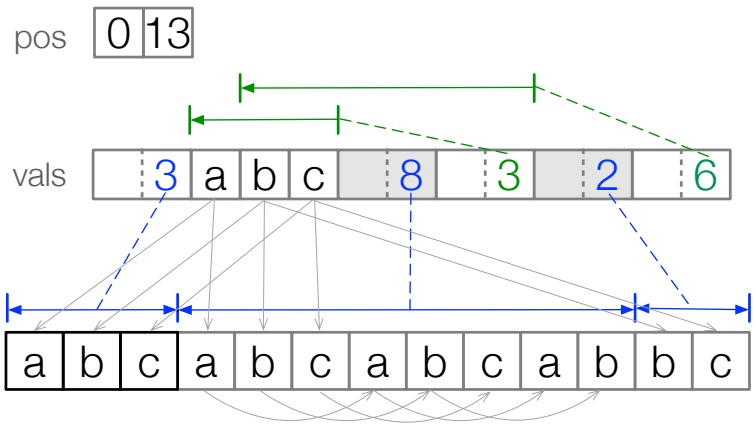


Figure 3-5: An example of a vector stored in the LZ77 level format, with raw values represented as defined values and repetitions (encoded by repeat tokens) represented as fill regions. The LZ77 sequence is `a,b,c,⟨8,3⟩,⟨2,6⟩`. Elements shaded gray in the `vals` array have high bits of one and denote the starts of repeat tokens. This variant of LZ77 stores distances d that represent relative offsets within the values array as opposed to offsets within the partially decoded sequence of elements.

sented by a two-byte element that has a high bit of one and that encodes c using the remaining bits, followed by another two-byte element that stores d. As Figure 3-5 illustrates, the repeat token can then be interpreted as a point in the tensor where the fill region dynamically changes to being the c values starting at d bytes prior in the values array. The raw values, on the other hand, can be interpreted as defined entries of the tensor.

3.3 Tracking Dynamic Fill Regions

In order to generate code to iterate over level formats that encode dynamic fills, a compiler must be able to emit code that keeps track of the current fill region. To enable this, we extend the level format abstraction with a new capability that captures how the fill region can be tracked at runtime during iteration. We define this capability as a single function:

```
fill_region(p_k, i_1, ···, i_k, vals) -> <sp, sz, found>
```

This function takes, as inputs, a position $\texttt{p}_k$ in a coordinate hierarchy level, the coordinates ($\texttt{i}_1$, ..., $\texttt{i}_k$) of the subtensor encoded at position $\texttt{p}_k$, and a reference `vals`

to the values array of the tensor. And, as outputs, the function is expected to return whether or not the fill region changes at position $\texttt{p}_k$ (i.e., `found`) and, if so, also return the new fill region itself (stored in an array of size `sz` referenced by the pointer `sp`).

The capability to keep track of fill regions can be implemented for the RLE level format as follows:

```
fill_region(p_k, i_1, ···, i_k, vals):
  return <&vals[p_k * size], size, true>
```

Since each stored coordinate also implicitly encodes a point where the fill region changes, the function always returns `found` as true and sets the new fill region to be the segment of the values array that corresponds to the coordinate stored at position $\texttt{p}_k$. When the level format is used to store the innermost dimension of a tensor, the value of `size` is 1 as the repetitions are of scalar values. However, when the level format is used to store other dimensions, `size` instead reflects the number of values that are in each subtensor stored by the level format. So if, for instance, the RLE level format is used to store the H dimension of a $W \times H \times 3$ tensor (and the innermost dimension is stored densely), then a value of 3 for `size` would reflect that repetitions are fill regions of size 3.

We implemented this capability for the Pack-RLE level format as follows:

```
fill_region(p_k, i_1, ···, i_k, vals):
  if ((load_uint16(vals, p_k) >> 15) & 1) {
    int count = load_uint16(vals, p_k) & 32767
    return <&vals[p_k + 2], count, true>
  }
  return <0, 0, false>
```

As our Pack-RLE level represents both repeated values and sequences, it first checks the 16 bit header to determine if a run is encoded at position p_k. If the header represents a sequence of values, the function returns `found` as false. Otherwise, the repeated value is immediately next to the header, with the count value read from the remaining 15 bits in the header.

The same capability can also be implemented for the LZ77 level format as follows:

```
fill_region(p_k, i_1, ···, i_k, vals):
  if ((load_uint16(vals, p_k) >> 15) & 1) {
    int count = load_uint16(vals, p_k) & 32767
    int dist = load_uint16(vals, p_k + 2)
    int size = MIN(count, dist)
    return <&vals[p_k - size], size, true>
  }
  return <0, 0, false>
```

As with the Pack-RLE level format, this function must first check for the type of the token that is stored at position $\texttt{p}_k$, since only repeat tokens encode points at which the fill region changes. If the token is a value token, then the function simply returns `found` as false. If the token is a repeat token though, then the function must calculate the new fill region size and position using the count and the distance encoded by the repeat token, which corresponds to the replicated values.

3.4 Appending Dynamic Fills

In order to support computations that store results in formats with dynamic fills, a compiler must also be able to emit code that inserts new fill regions into the output. To enable this, we further extend the level format abstraction with another new capability that captures how to append a new fill region to the output. We also define this capability as a single function:

```
append_fill_region(p_k, sp, sz, cnt, vals) -> void
```

This function takes, as arguments, the current end position $\texttt{p}_k$ of a level in the output's coordinate hierarchy representation, the new fill region to be appended (stored in an array of size `sz` referenced by `sp`), and the number `cnt` of output elements that this new fill region (assuming it encodes a repeated sequence of values) is actually meant to represent. Additionally, `vals` is a reference to the output values array.

The capability can be trivially implemented for the RLE level format as a no-op, since the format stores fill regions implicitly. On the other hand, since both the Pack-RLE and LZ77 level formats encode fill regions as explicit tokens in the tensor, the fill append capability can be implemented to set the token as appropriate.

The Pack-RLE level format implements this by appending the count with the high bit set.

```
append_fill_region(p_k, sp, sz, cnt, vals):
  store_uint16(vals, p_k, cnt | 32768)
  p_k += 2
```

For the LZ77 level format this capability is implemented by simply appending a repeat token to the values array, as follows:

```
append_fill_region(p_k, sp, sz, cnt, vals):
  store_uint16(vals, p_k, cnt | 32768)
  store_uint16(vals, p_k + 2, p_k - sp)
  p_k += 4
```

As we will show in Section 4.3, this enables a compiler to, by only reasoning about appending new fill regions, emit code that copies repetitions in the inputs directly to the output. This, in turns, makes it possible to generate code that directly compute on compressed data without first decompressing it.

Chapter 4

Code Generation for Generalized Fills

In this section, we show how our technique uses the abstractions we defined in Chapter 3 in order to generate code that compute on sparse tensors with dynamic fill regions and, by extension, generate code that efficiently compute on losslessly compressed data.

4.1 TACO Code Generation

Our technique, which builds on the technique described in [25], takes as input a tensor index notation statement that declaratively defines the tensor algebra computation to be performed. For instance, computation that alpha blends two tensors can be expressed in tensor index notation as $C_{ijc} = \alpha A_{ijc} + (1 - \alpha) B_{ijc}$, where the subscripts represent index variables used to access the modes of each tensor. To generate code, the compiler first lowers the tensor index notation statement down to concrete index notation, which is an IR that explicitly specifies the order of iteration over dimensions of the operands. So, for example, the alpha blending computation defined previously can be lowered to the concrete index notation statement $\forall i \forall j \forall c \; C_{ijc} = \alpha A_{ijc} + (1 - \alpha) B_{ijc}$.

The code generator then traverses the foralls (i.e., the $\forall$s) in order. For each forall (over dimension I), the code generator emits a loop (or multiple loops) that simultaneously iterates over the operands along dimension I. Within the loop(s), the

```
  while (piB < B1_pos[1] && piC < C1_pos[1]) {
    if (i == iB && iBVals == 0) {
      iB = B1Crd;
      if (load_uint16(B_vals, piB) >> 15 & 1)
            == 0) {
        iBVals = load_uint16(B_vals, piB);
        piB += 2;
        B1Crd += iBVals;
      }
      if (load_uint16(B_vals, piB) >> 15 & 1)
            == 1) {
        int32_t count =
            load_uint16(B_vals, piB) & 32767;
        int32_t dist = load_uint16(B_vals,
                                   piB + 2);
        BFillSize = MIN(B1Count, B1Dist);
        BFillRegion = &B_vals[piB - B1_dist];
        B1Crd += count;
        piB += 4;
        B1Found = true;
      } else {
        B1Found = false;
      }
      if (B1Found) {
        iB = B1Crd;
        BFillIndex = 0;
        BFillValue = BFillRegion[0];
      }
    }
...
    if (BVals == 0)
      BFillIndex = (BFillIndex + (i - iPrev))
                   % BFillSize;
...
    if (iB == i && iC == i && iBVals != 0 &&
        iCVals != 0) {
      store_uint16(A_vals, piA, 1);
      A_vals[piA + 2] = B_vals[piB] +
                        C_vals[piC];
      piA += 2;
      piA++;
    }
...
    else {
      int32_t lengthsLcm =
          LCM(BFillSize, CFillSize);
      int32_t coordMin = MIN(iB, iC);
      int32_t loopBound = i + lengthsLcm;
      int32_t startVar = piA;
      while (i < MIN(coordMin, loopBound)) {
        store_uint16(A_vals, piA, 1);
        A_vals[piA + 2] =
            B_vals[piB] + C_vals[piC];
        piA += 2;
        piA++;
        BFillIndex = (BFillIndex + 1)
                     % BFillSize;
        CFillIndex = (CFillIndex + 1)
                     % CFillSize;
        i++;
      }
      iPrev = i;
      if (MIN(coordMin, loopBound)
            == loopBound) {
        int32_t runValue = coordMin - i;
        store_uint16(A_vals, piA,
                     runValue | 32768);
        store_uint16(A_vals, piA + 2,
                     piA - startVar);
        piA += 4;
        i = coordMin;
      }
      continue;
    }
    piB += (int32_t)(iB == i);
    iB += (int32_t)(iB == i);
    piC += (int32_t)(iC == i);
    iC += (int32_t)(iC == i);
    iPrev = i++;
  }
```

Figure 4-1: Excerpt of code that our technique generates to add two LZ77 vectors, with the result also stored in LZ77.

code generator also emits if statements that, based on which operands actually contain defined values in a particular iteration of the loop, perform the specified computation with defined values from those operands (and fill values from the remaining operands). In addition, the code generator emits code that appends computed values to the result tensor.

4.2 Iterating with Dynamic Fills

To support computations on sparse tensors with dynamic fills though, our technique also emits code that keeps track of each input tensor's current fill value as the

```
1    while (piB < B1_pos[1] && piC < C1_pos[1]) {
2      int32_t iB = B1_crd[piB];
3      int32_t iC = C1_crd[piC];
4      int32_t i = min(iB,iC);
5      if (iB == i) {
6        BFillValue = (&(B_vals[piB]))[0];
7      if (iC == i) {
8        CFillValue = (&(C_vals[piC]))[0];
9      if (iB == i && iC == i) {
10       A_vals[piA] = B_vals[piB] + C_vals[piC];
11     } else if (iB == i) {
12       A_vals[piA] = B_vals[piB] + CFillValue;
13     } else {
14       A_vals[piA] = BFillValue + C_vals[piC];
15     }
16     A_crd[piA++] = i;
17     piB += (int32_t)(iB == i);
18     piC += (int32_t)(iC == i);
19   }
```

Figure 4-2: Code that our technique generates to add two RLE vectors, with the result also stored in RLE.

tensors are iterated over. In general, such code must keep track of each tensor's current fill region as well as track the position of the current fill value within the current fill region.

To keep track of a tensor B's current fill region, the generated code maintains a pointer (`BFillRegion`) to the start of the fill region and a variable (`BFillSize`) that keeps track of the size of the fill region. The generated code keeps these variables updated for each input tensor by invoking the `fill_region` level function whenever any element in the tensor is accessed. If the level function reports that the fill region for tensor B has changed, then `BFillRegion` and `BFillSize` are updated to store the new fill region returned by the level function. Lines 9–24 in Figure 4-1 show an example of code that our technique emits for tracking an LZ77 tensor's current fill region.

To track the position of a tensor B's current fill value within the current fill region, the generated code additionally maintains a variable (`BFillIndex`) that indexes into the fill region. This index is initialized to zero whenever the fill region changes, so that the index points to the start of the new fill region. Then, when iterating over the tensor, the generated code conceptually increments the index by one (potentially with wraparound) for every element of the tensor that follows the point where the fill region last changed. (To account for the fact that some elements may be skipped

when iterating over the tensor though, the generated code compensates by instead incrementing the index by the number of elements that were skipped.) Lines 27–28 and 47–48 in Figure 4-1 show an example of code that our technique emits for tracking the position of an LZ77 tensor's current fill value within the current fill region.

Our technique can additionally exploit properties of the operands' format in order to further optimize computations with dynamic fills. For example, when the size of a tensor's fill region is statically known to be one (as is the case with the RLE level format when used to store the innermost dimension, for instance), there is no distinction between the fill regions and fill values. Thus, in this case, the code generator does not need to emit code to keep track of the position of the tensor's current fill value within the current fill region. Preconditions that need to be satisfied for such an optimization can be checked by statically analyzing the definition of `fill_region` to see if the function returns the required values for `sz`. Figure 4-2 shows an example of how our technique applies the optimization to generate efficient code that computes on RLE-compressed data.

4.3 Appending Fill Regions

If the result of an element-wise computation is stored in a format that supports appending dynamic fill regions, our technique further optimizes the emitted code by minimizing the amount of computation with fill values. At coordinates where none of the input tensors have defined values, the generated code must use the input tensors' fill values to compute elements of the result. As Figure 4-3 illustrates though, since fill values of a tensor with fill region of size S repeat after every S elements (by definition), values of the result must therefore also repeat after every L elements, where L is the least common multiple (LCM) of the input tensors' fill region sizes. Thus, if more than L consecutive elements of the result are computed from just the input tensors' fill values, the generated code instead only computes the first L elements.

As the fill region sizes can be a dynamic property of the input tensors, this LCM computation must be done at runtime by the generated code. When computing with

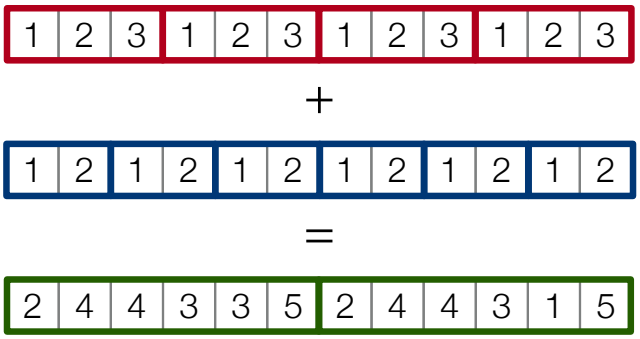


Figure 4-3: When adding a vector containing repetitions of three elements to a vector containing repetitions of two elements, the resulting vector must contain repetitions of $LCM(2,3) = 6$ elements. Our technique exploits this to emit code that optimizes computations with fill values.

formats with only static fill region size, such as RLE, are able to elide the LCM computation from the generated code by pre-computing its value statically. Then, the generated code appends those elements to the output tensor as a new fill region by invoking the `append_fill_region` level function.

Lines 38–59 in Figure 4-1 shows how our technique applies this optimization to generate code that directly computes on LZ77-compressed data and produces a compressed output without ever materializing uncompressed versions of the inputs or output, following the same general approach as [52]. Lines 42–50 in Figure 4-1 compute the L necessary values, and lines 52–58 invoke the `append_fill_region` level function. Similarly, Figure 4-2 shows how our technique applies the same optimization to generate code that directly computes on RLE-compressed data.

4.4 Optimizing Reductions

When computing the result of a reduction, such as $a = b_i c_i$, the compiler is able to reduce the computation cost by factoring out repeated multiplication as in Figure 4-4. When there is a fill region in one input tensor and dense values in another input tensor, we can factor out repeated multiplication by the values in the fill region. This optimization also applies with a scalar fill value, as in traditional sparse formats, and is beneficial when the fill value is not an annihilator.

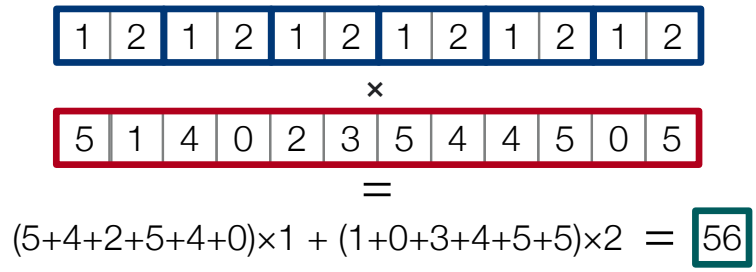


Figure 4-4: When performing a reduction on two vectors, there is repeated multiplication from values within a fill region. Our technique is able to optimize reduction operations by factoring out the repeated multiplication.

When there are multiple fill regions involved in a reduction computation, we can further optimize the computation. The compiler can first generate code to calculate the output obtained by performing the element-wise multiplication between the input tensors. Both the length of the resulting pattern, and the number of elements which are repeated are calculated as in section 4.3. The reduction of the element-wise intermediate result is multiplied by the number of repetitions to calculate the final value of reducing the input fill regions. This reduces the total cost of computing reductions when there are multiple fill regions.

For computations with both multiple fill regions and dense inputs, both of the above optimizations are applied. First, the generated code produces the element-wise output of the multiplication operation for all of the fill regions. This single repeated pattern can then be reduced with the dense outputs as described above, by factoring out repeated multiplication.

While reduction operations do not directly result in compressed outputs, the above optimizations reduce the computation cost of computing with compressed inputs. This is in addition to the reduced cost of data movement of compressed input tensors.

Chapter 5

Lossless Compression Using Finch

Our new abstractions make it possible to represent lossless compression formats within TACO, however it required significant engineering effort to produce a prototype compiler implementing our techniques. In addition to the new capabilities, to achieve the best performance on the LZ77 and Pack-RLE formats required additional extensions to efficiently iterate the densely stored regions. The optimizations described in chapter 4 also required integration into the compiler itself, though they apply to any new formats added. Composability with other new techniques, such as for ragged [23] or symmetric arrays [43] is also a challenge, as these techniques are separate compiler extensions which require significant effort to merge. A new langauge, Looplets [7], and its associated compiler Finch, aims to solve many of these issues with a new representation of array structure.

5.1 Background on Looplets

The Looplet language consists of *looplets* which describe abstract sequences of values, and are combined together to create formats using a level hierarchy as in TACO. In TACO, a level format can only iterate over non-zeros in a few standard ways, where looplets can be arbitrarily combined to represent values. Within a given level format, looplets are used to represent regions or sub-regions in that level, and must represent the full sequence of indices or values. Each region or sub-region is represented with

their absolute starting and ending index.

The Finch compiler lowers looplets into loop nests which can efficiently co-iterate many tensors. The values represented by each of the looplets can be static or dynamic. For example, the **run** looplet represents a single repeated value, and can be used to represent the fill value in a sparse tensor. The dense level in TACO can be represented using the **lookup** looplet, which represents an arbitrary sequence as a function of the index. There are also looplets to sequence regions, and to switch between different cases. The Looplets language can express a wide variety of Tensor formats through various combinations of the looplets which make up the language.

The Finch compiler also simplifies the implementation of optimizations, by expressing optimizations as rewrite rules which are applied when lowering tensor computations into loop nests. This makes it possible to generate comparable code to TACO, as there are included rewrite rules to statically remove computation with annihilator values. This also allows users to implement their own rewrite rules, for their own logic or functions without needing detailed knowledge of the compiler's internals.

5.2 Lossless Compression using Looplets

The Looplets language generalizes our techniques for representing sparsity within a tensor compiler. The **run** looplet can be used to implement the dynamic fill-values we describe, including appending fill values when generating compressed outputs. Using Looplets, each region of fill values within the tensor is replaced with a **run** which gets its value from the appropriate level data-structure. Furthermore, the **lookup** looplet can be used to represent the dense regions in both the Pack-RLE and LZ77 formats.

Currently, LZ77 is not expressible, as there is no looplet representing repeated regions of values, however this is a possible language extension. Both the RLE and Pack-RLE formats are described in the Looplets paper, and we evaluate their performance in chapter 6. We also implemented a new format, Repeat-VBL, which extends the VBL format to represent arbitrary runs in between dense blocks of values instead of a single default fill value. This required adding an additional array of repeated val-

```
@loop i ∈ start:stop b[j] += c * r[i] => if i ≠ j (b[j] += c * acc) where (@loop i ∈ start:stop acc += r[i])
```

Figure 5-1: The Finch rewrite rule to factor out repeated multiplication.

ues to the VBL format, and modifying the format to use the dynamic value from the array in the **run** looplet instead of the static fill value. This format can outperform our Pack-RLE format, depending on the input data, as it can represent arbitrarily long regions of runs and dense blocks, where Pack-RLE is limited by the header size.

To achieve the best performance, we also added an additional protocol for iterating the Pack-RLE and Repeat-VBL formats when the data has both runs and significant sparsity. Within the new Pack-RLE and Repeat-VBL formats, each run is split into two cases, a zero-run and a non-zero run. The compiler then automatically applies existing rewrite rules to elide the zero-run case. This produces code which dynamically checks the value of the run and only performs computation when necessary. This was necessary to achieve the best performance compared to sparse formats to further reduce the computation done.

We also add a rewrite rule implementing the optimization described in section 4.4 to factor out repeated multiplication. The new rule, shown in fig. 5-1, recognizes when a loop-invariant value is repeatedly multiplied with values in a dense region and automatically factors out the constant multiplication.

Chapter 6

Evaluation

We implement our technique as a prototype extension to TACO. We then evaluate it against TACO without our extension (which supports dense and sparse inputs/outputs, but not RLE or LZ77) as well as against the widely used image and video processing library OpenCV [15]. We find that support for lossless compression is essential for memory usage and performance in many applications. We also evaluate our techniques with in the Finch compiler.

6.1 Methodology

We ran our experiments on a dual-socket Intel Xeon E5-2680 v3 machine with 128 GB of main memory and running Ubuntu 18.04.3 LTS. All of our experiments were run single-threaded, with Turbo Boost disabled and execution restricted to a single socket using `numactl`. Unless otherwise noted, all of our experiments were run with a cold cache, and each experiment was repeated at least ten times.

6.2 Computing on Compressed Data

We first exhibit the flexibility and performance benefits of our techniques with micro-benchmarks on synthetic data. We measure performance for the following computations:

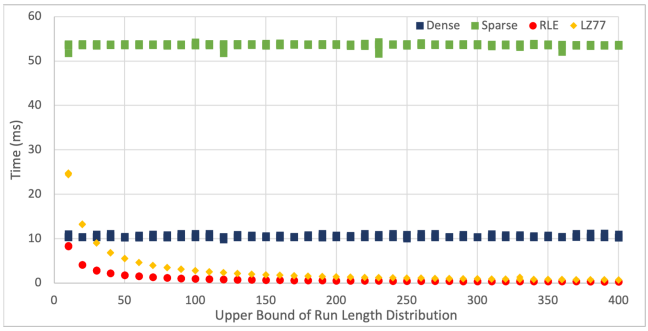
- Scalar multiplication, $A_{ij} = B_{ij} * c$
- Element-wise multiplication, $A_{ij} = B_{ij} * C_{ij}$
- Reduction (matrix-vector product), $A_i = B_{ij} * C_j$, where C is stored as an RLE vector
- Mixed operation (multiplication with a sparse mask), $A_{ij} = B_{ij} * C_{ij}$, where C is stored as CSR

We generate integer tensors by first sampling a random value uniformly from the range 0 to 255 and then determine the number of copies, or run length, of each value by sampling a random value uniformly from the range 1 to a defined upper limit. We generate ten random matrices for each run length upper bound of size $10,000 \times 1,000$. We show the execution time for each of the operations plotted against the upper bound of the run length, which approximates the compression factor in Figure 6-1.

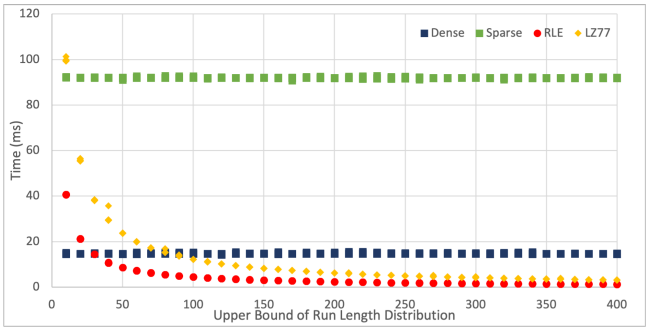
There is overhead to computing on compressed data, so for low compression ratios, depending on the computation, our technique is initially outperformed by computing on dense tensors. However there is a crossover point after which computing on both the RLE and LZ77 tensors is faster. For all of the kernels except matrix-vector product, computing with sparse matrices is significantly worse than computing on dense matrices, as the matrices have high density. However, the performance of matrix-vector product also depends on the compression of the RLE vector, which contributes to higher variability among the Dense and Sparse cases, and also makes computing on the CSR matrix faster. The LZ77 level format has higher iteration costs due to the complexity of the format, and it has no representation advantages over RLE on these generated tensors.

6.3 Image Processing Applications

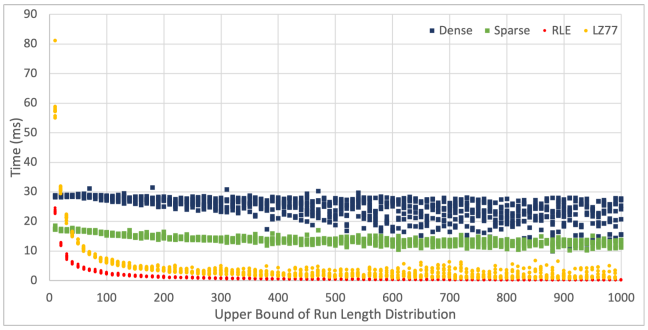
We evaluate our technique on two kernels used in image processing, namely alpha blending and edge detection.



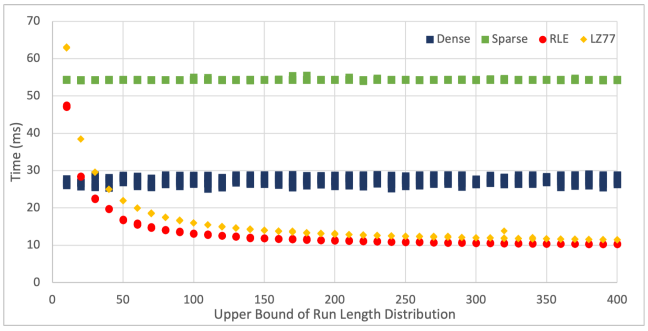
(a) Scalar multiplication



(b) Element-wise multiplication

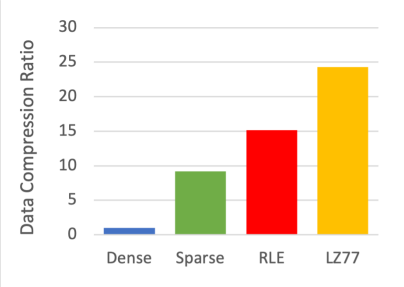


(c) Matrix-vector product

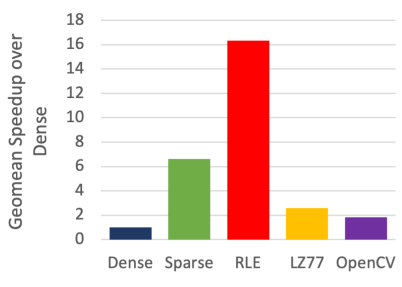


(d) Element-wise multiplication with a sparse mask

Figure 6-1: Performance of micro-benchmarks on synthetic data.

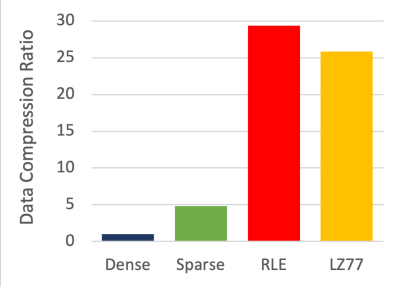


(a) Data compression ratio

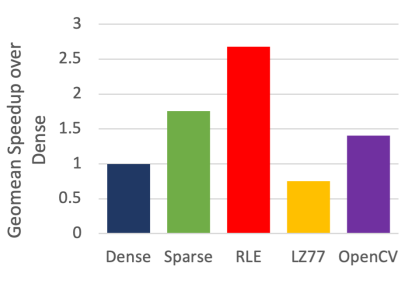


(b) Execution time speedup

Figure 6-2: Results of alpha blending experiments.



(a) Data compression ratio



(b) Execution time speedup

Figure 6-3: Results of edge detection experiments.

6.3.1 Alpha Blending

A common operation in the image processing domain is alpha blending, or the weighted element-wise sum of two images, represented by the following index statement $A_{ij} = \alpha B_{ij} + (1 - \alpha)C_{ij}$. We evaluate this operation on pairs of images pulled from a subset of 2000 images from the sketch dataset from [21]. We report the geometric mean (geomean) speedup and size reduction compared to dense in Figure 6-2. Our RLE format has the largest geomean speedup of 16.3× faster than dense and 16.1× faster than OpenCV. While these images are very sparse with most of the pixels being white background, they also have relatively large regions of black. Using the RLE format, we can gain additional speedups over traditional sparse computing, with a geomean speedup of 2.5× over CSR.

6.3.2 Medical Image Edge Detection

A common image processing algorithm used in many fields, including medical images, is edge detection. We implement boundary edge detection on MRI images as described in [49]. We further filter the output image by applying a region-of-interest (ROI) mask, as done in [25]. The expression we compute is $Out_{ij} = (A_{ij} \wedge ROI_{ij}) \oplus (B_{ij} \wedge ROI_{ij})$ where A and B are thresholded versions of the original image with $t_1 = 75\%$ used to compute A and $t_2 = 80\%$ used to compute B. The ROI used is generated by placing 4 rectangular regions of interest, each 40×40 pixels, in the center of the image 20 pixels apart.

We report the geomean size reduction of the input tensors and the geomean speedup over computing over Dense tensors in Figure 6-3. Using the RLE level format, we achieve a geomean speedup of 2.6× over Dense, and 1.5× over Sparse. While performing the computation using OpenCV is faster than the Dense computation using TACO, mainly due to hand vectorization, we still report a goemean speedup of 1.9× of the RLE format over OpenCV.

6.4 Video Processing Applications

We evaluate our technique on two kernels used in video processing, namely 1) compositing two videos together with a mask and 2) brightening the video. We use 12 video clips for our evaluation: four from the 3D-animated film Elephants Dream, four from the 2D-animated film Sita Sings the Blues, and four stock videos from Pexel. As all of these videos are in color, a third index variable c is used to index the additional mode of the input and output video frames.

6.4.1 Brightening

This operation does element-wise addition with a constant and truncates the value at the maximum (i.e., 255, as all of the videos are in 8-bit color).

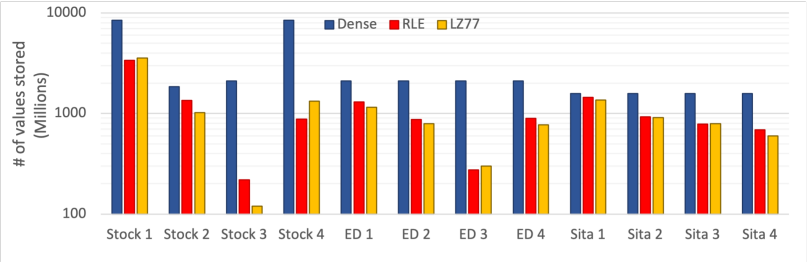


Figure 6-4: Storage size of each saved file format for brighten.

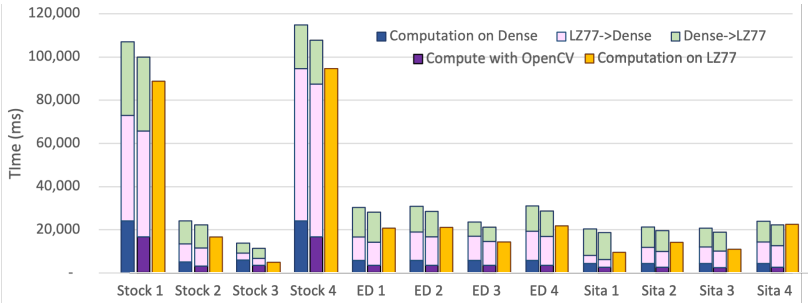


Figure 6-5: Execution time of performing the subtitle computation where the files are saved in the LZ77 format.

6.4.2 Compositing

We test this operation by compositing a subtitle image onto every frame in each of the videos. The index expression for performing this operation on each frame of the video is $Out_{ijc} = (F_{ijc} * M_{ij}) + (S_{ij} * !M_{ij})$ where F is the input frame, S is the grayscale subtitle image, and M is the Boolean mask.

6.4.3 Results

Figure 6-4 shows the storage benefits of lossless compression, with both RLE and LZ77 storing up to an order of magnitude fewer explicit values than the dense representations. Figures 6-6 and 6-7 show the execution time of computing directly on the given formats. While there are cases where the execution time of computing directly on RLE and LZ77 is faster then Dense or using OpenCV, in many cases the execution time of computing on the compressed data is slower. As storing video data as uncompressed is often extremely impractical, a fairer comparison is to the total

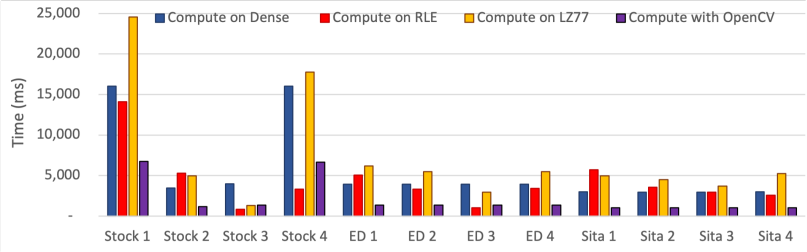


Figure 6-6: Execution time of performing the brightening computation directly on the saved file format.

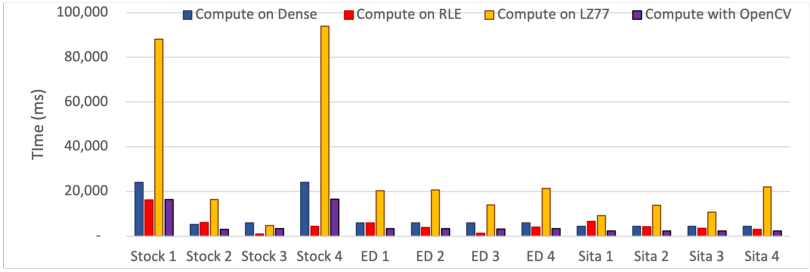


Figure 6-7: Execution time of performing the subtitle computation directly on the saved file format.

processing time, which includes both decompression and re-compression. We show a comparison of computing on dense tensors using both TACO and OpenCV and computing directly on LZ77 in Figure 6-5. Even though the computation time for LZ77 is much longer then any of the other formats, the time necessary for decompression and re-compression ensures that computing directly on LZ77 is still faster, in all but one case where the performance is equivalent.

6.5 Looplets and Finch

We evaluate the lossless compressed formats within Finch on the matrix-vector multiplication kernel $y_i = A_{ij} * x_j$, where only the matrix A is stored in either a sparse or lossless compressed format. As Finch is written in Julia, we run the benchmark at least ten times as described above, or for at least five seconds. However, we take the minimum time to account for the cost of JIT overhead from both the Julia runtime and Finch compiler for the kernels.

Many machine learning algorithms, including regression algorithms, use matrix

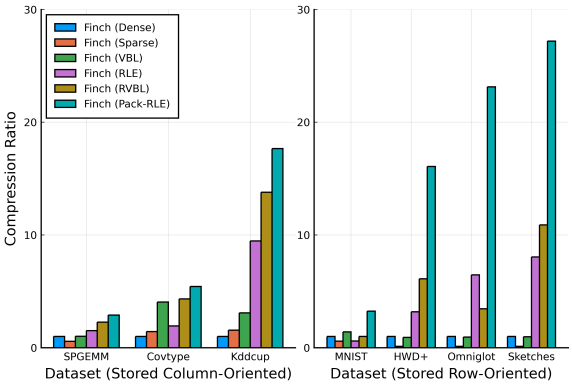


Figure 6-8: Compression ratio of matrices used for matrix vector product.

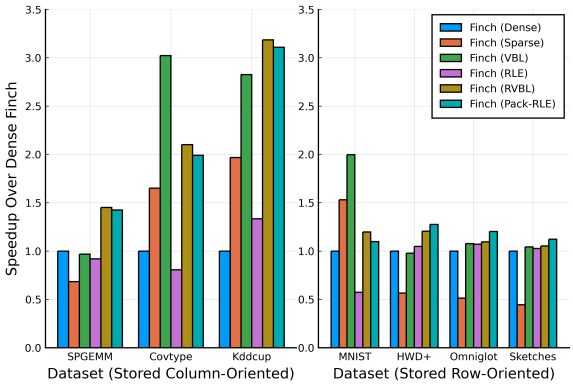


Figure 6-9: Relative time of matrix vector product.

vector products, and the datasets, while many may be sparse, are also compressible. The previous experiments only compressed along the rows of the matrices and tensors, however this approach is not as effective with many datasets. The first four datasets used, Covtype [13], Kddcup [3], and SPGEMM [38], are compressed along their columns. Compressing along their rows would be significantly less effective as each column represents distinct data for each sample, so in each column, there is significant potential for compression. The next four datasets are image-based, Omniglot [34], MNIST [18], HWD+ [11] and Sketches [21]. There is less potential for column compression with these datasets, and as shown in the previous image processing evaluation, significant potential for compression along the row dimension. The matrices are stored using both integral and floating point types, where appropriate.

We show the compression ratios achieved in fig. 6-8 and the relative speedup compared to dense computation in fig. 6-9. These datasets ranged in sparsity, while also having both runs and dense regions. Depending on the dataset, while one of the lossless-compressed formats always had the best compression ratio, the additional runtime complexity of these formats did not always result in the best performance. Computing column compressed resulted in higher runtime speedups for the datasets we evaluated, while the highest compression ratio observed was on the images datasets. While the long rows in these datasets were very amendable to compression, this likely resulted in poor cache utilization, as the vector being multiplied was extremely large.

We still observed performance roughly on par with dense computation, while having significantly higher compression. As these datasets are usually stored and transmitted in compressed formats, using our lossless compression formats would result in end-to-end speedups.

Chapter 7

Related Works

In this section, we describe related works on lossless compression algorithms and techniques for directly computing on compressed data (compressed domain processing), as well as describe related works on sparse programming.

7.1 Lossless Compression

While this work focuses on LZ77-style compression [56], it is one example of a general purpose lossless compression algorithm. LZ77 is part of a larger class of dictionary based compressors which includes LZ78 [57], LZW [55] and LZSS [50], which encode repetition into a dictionary to avoid redundant storage. The second main class of lossless compression algorithms is entropy encoders, includes Huffman coding [26] and Arithmetic coding [42, 39]. These algorithms compress data by replacing fixed length input symbols with code words whose length is determined by the probabilities of each input symbol in the data to be compressed.

Most general purpose compression formats use one of or both dictionary compression and entropy coding, including 7z [1], ZIP [40], gzip [19], and bzip2 [2]. There are also many more formats specialized for specific kinds of data, including audio, graphics, and video data. These formats can take advantage of specific properties of the data they store and compress, however they generally use dictionary and entropy coding as a part of their algorithms.

7.2 Compressed Domain Processing

There are many algorithms for compressed domain processing, which differ depending on the structure of the compressed data. For example there is prior work on pattern matching within compressed text [9, 29, 24, 8], computing directly on compressed databases [6, 5], and computing directly on compressed video [52]. However, much of this work focuses on lossy compression schemes which use the Discrete Cosine Transform [45, 46, 44, 14, 36]. We do not consider this kind of compression in our work.

Two approaches for compressing matrices for SpMV were introduced in [33], one based on compressing the index data structures using delta coding, and the second compressing the values by only storing unique elements and representing them with smaller indices.

There has also been research into compressing matrices for linear algebra. In [32], they propose techniques for compressing both the index and value data structures of sparse matrices for matrix-vector products. The index compression technique uses delta coding to reduce the size of the column coordinate array. The value compression stores a array of unique values and replaces the values array with indices into the array of unique values. Both [22] and [35] describe systems for compressing and computing on data matrices for common machine learning algorithms. In both cases are limited to a subset of matrix operations, and the specific compression formats they designed. In [22], they partition matrices into column groups and compress each group together, using both RLE and offset list encoding (OLE). In OLE, they store each value in a column with a list of coordinates it appears at, similarly to the value compression technique from [32]. In [35], they develop a dictionary compression scheme which does not compress data across row or column boundaries, however can represent repetition across a matrix by a common dictionary for each compressed matrix.

7.3 Sparse Programming

Our technique builds on the sparse tensor algebra compiler TACO [25, 16, 30], which implements the techniques described in Section 2.3 and Section 4.1. Without our extension, TACO does not support lossless compression techniques like RLE and LZ77 and cannot efficiently compute with data that contain many distinct repeated values or sequences. There also exist a number of other compiler techniques that can generate sparse linear algebra kernels given imperative implementations of their dense counterparts, including MT1 [12], Bernoulli [31], SIPR [41], and CHiLL [54]. Bernoulli uses a sparse matrix format abstraction that can also represent losslessly-compressed data, but this abstraction exposes losslessly-compressed data to the compiler as just fully-decompressed streams of nonzeros. Thus, Bernoulli cannot generate code that avoid redundant computations by directly computing on compressed data. Meanwhile, the other techniques only support sparse matrix representations that compress out zero elements, and thus they cannot generate code to compute with losslessly compressed data. In addition, GraphBLAS [37, 28, 17] and CTF [48] are examples of sparse linear algebra frameworks that support arbitrary semirings, which can have any value as the "zero" that is compressed out in storage. As with the technique of [25] though, such systems can only efficiently compute with data that contain mostly a single value, not data that contain many distinct repeated values or sequences.

Furthermore, in the context of domain-specific hardware design, [51] describe a hierarchical fiber-tree abstraction for sparse tensor storage, similar to the coordinate hierarchy abstraction of [16]. The abstraction supports storing zeros using RLE, but it does not support lossless compression of nonzero elements.

Chapter 8

Conclusion

This paper shows how to build a compiler for computing with both losslessly compressed and sparse tensors by generalizing the notion of sparsity to handle different repeated values within a single tensor. With our technique, the compiler is able to generate efficient code for varied computations. We observe speedups up to 16× over computing on dense data, and even in the worst case we observe speedups or equivalent performance over just decompressing the inputs, computing on dense data, and then recompressing the result.

Bibliography

[1] 7z Format.

[2] bzip2 : Home.

[3] KDD Cup 1999 Data. UCI Machine Learning Repository, 1998.

[4] Technical Note TN1023: Understanding PackBits, July 2008.

[5] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 671–682, New York, NY, USA, June 2006. Association for Computing Machinery.

[6] Sushila Aghav. Database compression techniques for performance optimization. In *2010 2nd International Conference on Computer Engineering and Technology*, volume 6, pages V6–714–V6–717, 2010.

[7] Willow Ahrens, Daniel Donenfeld, Fredrik Kjolstad, and Saman Amarasinghe. Looplets: A language for structured coiteration, 2022.

[8] A. Amir and C. Benson. Efficient two-dimensional compressed matching. In *1992 Data Compression Conference*, pages 279,280,281,282,283,284,285,286,287,288, Los Alamitos, CA, USA, mar 1992. IEEE Computer Society.

[9] Amihood Amir, Gary Benson, and Martin Farach. Let Sleeping Files Lie: Pattern Matching in Z-Compressed Files. *Journal of Computer and System Sciences*, 52(2):299–307, April 1996.

[10] Brett W Bader and Tamara G Kolda. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing*, 30(1):205–231, 2007.

[11] Cédric Beaulac and Jeffrey S. Rosenthal. Introducing a new high-resolution handwritten digits data set with writer characteristics. *SN Computer Science*, 4(1), nov 2022.

[12] Aart J. C. Bik and Harry A. G. Wijshoff. Compilation techniques for sparse matrix computations. In *International Conference on Supercomputing*, pages 416–424. ACM, July 1993.

[13] Jock Blackard. Covertype. UCI Machine Learning Repository, 1998.

[14] Bo Shen, I.K. Sethi, and V. Bhaskaran. DCT convolution and its application in compressed domain. *IEEE Transactions on Circuits and Systems for Video Technology*, 8(8):947–952, December 1998.

[15] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[16] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–30, October 2018.

[17] Timothy A. Davis. Algorithm 1000: Suitesparse:graphblas: Graph algorithms in the language of sparse linear algebra. *ACM Trans. Math. Softw.*, 45(4), December 2019.

[18] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[19] P. Deutsch. Rfc1952: Gzip file format specification version 4.3, 1996.

[20] Daniel Donenfeld, Stephen Chou, and Saman Amarasinghe. Unified compilation for lossless compression and sparse computing. CGO 2022, Apr 2022.

[21] Mathias Eitz, James Hays, and Marc Alexa. How Do Humans Sketch Objects? *ACM Transactions on Graphics - TOG*, 31, July 2012.

[22] Ahmed Elgohary, Matthias Boehm, Peter J. Haas, Frederick R. Reiss, and Berthold Reinwald. Compressed linear algebra for declarative large-scale machine learning. *Communications of the ACM*, 62(5):83–91, April 2019.

[23] Pratik Fegade, Tianqi Chen, Phillip B. Gibbons, and Todd C. Mowry. The CoRa Tensor Compiler: Compilation for Ragged Tensors with Minimal Padding. *arXiv:2110.10221 [cs]*, October 2021. arXiv: 2110.10221.

[24] Travis Gagie, Paweł Gawrychowski, and Simon J. Puglisi. Approximate pattern matching in LZ77-compressed texts. *Journal of Discrete Algorithms*, 32:64–68, May 2015.

[25] Rawn Henry, Olivia Hsu, Rohan Yadav, Stephen Chou, Kunle Olukotun, Saman Amarasinghe, and Fredrik Kjolstad. Compilation of sparse array programming models. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[26] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[27] Intel. Intel math kernel library developer reference, 2020.

[28] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, C. Yang, J. D. Owens, M. Zalewski, T. Mattson, and J. Moreira. Mathematical foundations of the graphblas. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9, 2016.

[29] T. Kida, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. A unifying framework for compressed pattern matching. In *6th International Symposium on String Processing and Information Retrieval. 5th International Workshop on Groupware (Cat. No.PR00268)*, pages 89–96, September 1999.

[30] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–29, October 2017.

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

[32] Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. Improving the performance of multithreaded sparse matrix-vector multiplication using index and value compression. In *2008 37th International Conference on Parallel Processing*, pages 511–519, 2008.

[33] Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. Optimizing sparse matrix-vector multiplication using index and value compression. In *Proceedings of the 5th Conference on Computing Frontiers*, CF '08, page 87–96, New York, NY, USA, 2008. Association for Computing Machinery.

[34] Brenden M. Lake, Ruslan Salakhutdinov, and Joshua B. Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

[35] Fengan Li, Lingjiao Chen, Yijing Zeng, Arun Kumar, Jeffrey F. Naughton, Jignesh M. Patel, and Xi Wu. Tuple-oriented Compression for Large-scale Mini-batch Stochastic Gradient Descent. *Proceedings of the 2019 International Conference on Management of Data*, pages 1517–1534, June 2019. arXiv: 1702.06943.

[36] Giridhar Mandyam, Nasir Ahmed, and Neeraj Magotra. Lossless Image Compression Using the Discrete Cosine Transform. *Journal of Visual Communication and Image Representation*, 8(1):21–26, March 1997.

[37] Tim Mattson, David Bader, Jon Berry, Aydın Buluç, Jack Dongarra, Christos Faloutsos, John Feo, John R. Gilbert, Joseph Gonzalez, Bruce Hendrickson, Jeremy Kepner, Charles E Leiserson, Andrew Lumsdaine, David Padua, Stephen Poole, Steve Reinhardt, Michael Stonebraker, Steve Wallach, and Andrew Yoo. Standards for graph algorithm primitives. In *IEEE High Performance Extreme Computing Conference*, pages 1–2. IEEE, 2013.

[38] Rafael Paredes, Enrique I& Ballester-Ripoll. SGEMM GPU kernel performance. UCI Machine Learning Repository, 2018.

[39] R. Pasco. Source coding algorithms for fast data compression (ph.d. thesis abstr.). *IEEE Trans. Inf. Theor.*, 23(4):548, September 2006.

[40] PKWare. .ZIP File Format Specification, July 2020.

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

[42] J. J. Rissanen. Generalized kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20(3):198–203, 1976.

[43] Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, Stephen Chou, Shoaib Kamil, Saman Amarasinghe, and Fredrik Kjolstad. A sparse iteration space transformation framework for sparse tensor algebra. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):158:1–158:30, November 2020.

[44] B. Shen and I. K. Sethi. Convolution-Based Edge Detection for Image/Video in Block DCT Domain. *Journal of Visual Communications and Image Representation*, 7:411–423, 1996.

[45] B.C. Smith and L.A. Rowe. Algorithms for manipulating compressed images. *IEEE Computer Graphics and Applications*, 13(5):34–42, September 1993. Conference Name: IEEE Computer Graphics and Applications.

[46] Brian C. Smith and Lawrence A. Rowe. Compressed domain processing of jpeg-encoded imaages. *Real-Time Imaging*, 2(1):3–17, 1996.

[47] Shaden Smith and George Karypis. Tensor-matrix products with a compressed sparse tensor. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, page 5. ACM, 2015.

[48] Edgar Solomonik and Torsten Hoefler. Sparse tensor algebra as a parallel programming model. *arXiv preprint arXiv:1512.00066*, 2015.

[49] Krit Somkantha, Nipon Theera-Umpon, and Sansanee Auephanwiriyakul. Boundary detection in medical images using edge following algorithm based on intensity gradient and texture gradient features. *IEEE Transactions on Biomedical Engineering*, 58(3):567–573, 2011.

[50] James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, October 1982.

[51] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. Efficient processing of deep neural networks. *Synthesis Lectures on Computer Architecture*, 15(2):1–341, 2020.

[52] William Thies, Steven Hall, and Saman Amarasinghe. Manipulating lossless video in the compressed domain. In *Proceedings of the seventeen ACM international conference on Multimedia - MM '09*, page 331, Beijing, China, 2009. ACM Press.

[53] William F Tinney and John W Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proceedings of the IEEE*, 55(11):1801–1809, 1967.

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

[55] T. Welch. A technique for high-performance data compression. *Computer*, 17(06):8–19, jun 1984.

[56] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

[57] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.