Unified Compilation for Lossless Compression and Sparse Computing

Daniel Donenfeld
CSAIL, MIT
Cambridge, USA
danielbd@mit.edu

Stephen Chou
CSAIL, MIT
Cambridge, USA
s3chou@csail.mit.edu

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

***Abstract*—This paper shows 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.**

Our evaluation 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 $16.3\times$ faster than equivalent dense kernels generated by TACO, a tensor algebra compiler, and up to $16.1\times$ faster than OpenCV, a widely used image processing library.

***Index Terms*—lossless compression, compressed domain processing, sparse tensor algebra**

I. 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 [1], 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) [2] and compressed sparse fiber (CSF) [3], 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 [4] and compilers like TACO [5]–[7], 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 paper, 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 [7] and [6], 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 [8]. 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.

II. Background

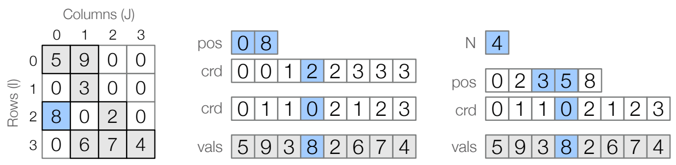
We briefly describe lossless compression and sparse programming, which are two distinct approaches for efficiently storing and computing with large data sets that contain repeated patterns. We also provide an overview of the TACO sparse tensor algebra compiler, which our technique extends.

A. 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.

1) RLE: RLE encodes any contiguous sequence of repeated values as a single copy of the repeated value followed by 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.

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.



(a) COO (b) CSR

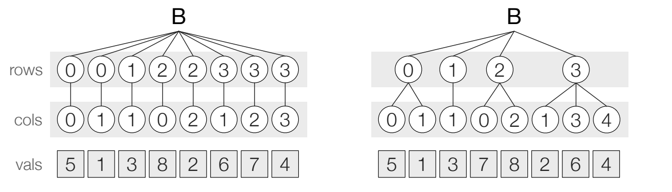
Fig. 1: The same tensor stored in two different formats.

Though most existing frameworks that utilize lossless compression work with compressed data by first decompressing the data before computing with it, [9] 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.

B. Sparse Programming

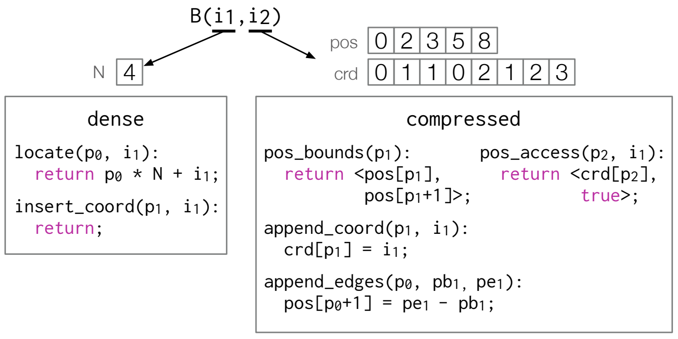
Sparse programming systems, on the other hand, are optimized to compute with tensors (multidimensional arrays) that contain mostly repeated zeros by storing such sparse tensors in specialized formats that avoid materializing the zeros. There exists many formats for storing sparse tensors, including CSR, CSF, and the coordinate format (COO) [10], as illustrated in Figure 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 [11]–[13] 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



(a) COO (b) CSR

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



```
dense

locate(p0, i1):
  return p0 * N + i1;

insert_coord(p1, i1):
  return;
```

```
compressed

pos_bounds(p1):           pos_access(p2, i1):
  return <pos[p1],          return <crd[p2],
          pos[p1+1]>;               true>;

append_coord(p1, i1):
  crd[p1] = i1;

append_edges(p0, pb1, pe1):
  pos[p0+1] = pe1 - pb1;
```

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

5	9	0	0
0	3	0	0
8	0	2	0
0	6	7	4

(a) Sparse tensor with mostly zeros

5	9	1	1
1	3	1	1
8	1	2	1
1	6	7	4

(b) Sparse tensor with mostly ones

5	9	5	9
3	3	4	4
2	0	2	0
0	7	7	7

(c) Dense tensor with repeated values/sequences

Fig. 4: 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.

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).

C. Sparse Tensor Algebra Compilation

Chou et al. [6] 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 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 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 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.

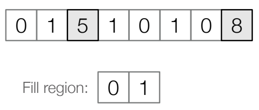
[1]The name of the compressed level format does not imply that it utilizes (lossless) compression, but rather that coordinates of zero entries are omitted.

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.

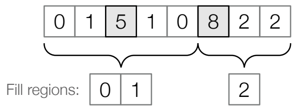
III. Representing Generalized Fill Values

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 4 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 [6] 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



(a) Fill regions



(b) Dynamic fills

Fig. 5: Our generalization of fill values supports non-scalar fills (fill regions) and different fills for different parts of a tensor (dynamic fills).

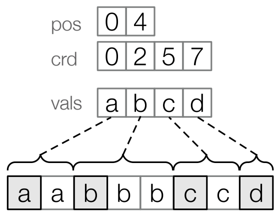


Fig. 6: An example of a vector stored in the RLE level format, with each run of identical elements represented as a defined value followed by fills that have the same value. This variant of RLE explicitly stores the start and end coordinates of each run (other than the end coordinate of the last run, 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.

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 few or no exact repetition.

A. 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*.

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 5a 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.

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 5b shows an example of a sparse tensor that contains two distinct fill regions.

B. Lossless Compression as Level Formats

As mentioned previously, RLE and LZ77 are two examples of commonly used lossless compression algorithms. We propose two 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.

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 6, we demonstrate a variant of RLE as a level format that can 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.

2) LZ77: Figure 7 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. 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 $\langle \texttt{c,d} \rangle$ is represented 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 7 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.

C. 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:

$\texttt{fill\_region}(\texttt{p}_k,\ \texttt{i}_1,\ \cdots,\ \texttt{i}_k,\ \texttt{vals})$

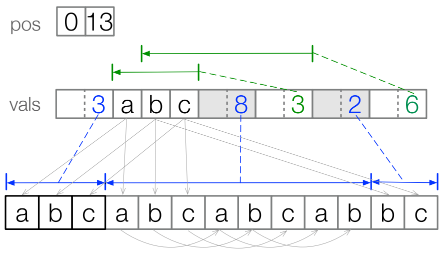


Fig. 7: An example of a vector stored in the LZ77 level format, with raw values represented as defined values and repetitions (encoded by repeat tokens) represented as fill regions. The LZ77 sequence is `a,b,c,⟨8,3⟩,⟨2,6⟩`. Elements shaded gray in the `vals` array have high bits of one and denote the 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.

```
  -> <sp, sz, found>
```

This function takes, as inputs, a position $\mathtt{p}_k$ in a coordinate hierarchy level, the coordinates ($\mathtt{i}_1$, ..., $\mathtt{i}_k$) of the subtensor encoded at position $\mathtt{p}_k$, 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 $\mathtt{p}_k$ (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(pk, i1, ···, ik, vals):
  return <&vals[pk * 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 $\mathtt{p}_k$. 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.

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

```
fill_region(pk, i1, ···, ik, vals):
  if ((load_uint16(vals, pk) >> 15) & 1)
  {
    int count = load_uint16(vals, pk])
                    & 32767
    int dist = load_uint16(vals, pk + 2])
    int size = MIN(count, dist)
    return <&vals[pk - size], size, true>
  }
  return <0, 0, false>
```

The function must first check for the type of the token that is stored at position $\mathtt{p}_k$, 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 uses the count and the distance encoded by the repeat token in order to determine the new fill region, which corresponds to the replicated values.

D. 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(pk, sp, sz, cnt, vals)
  -> void
```

This function takes, as arguments, the current end position $\mathtt{p}_k$ 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 the LZ77 level format encodes fill regions as repeat tokens, the fill append capability can be implemented for the level format by code that simply appends a repeat token to the values array, as follows:

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

As we will show in Section IV-C, 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.

IV. Code Generation for Generalized Fills

In this section, we show how our technique uses the abstractions we defined in Section III 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.

A. TACO Code Generation

Our technique, which builds on the technique described in [7], 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 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.

B. 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 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 8 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 8 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 9 shows an example of how our technique applies the optimization to generate efficient code that computes on RLE-compressed data.

C. 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 10 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 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 8 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 [9]. Lines 42–50 in Figure 8 compute the L necessary values, and lines 52–58 invoke the `append_fill_region` level function. Similarly, Figure 9 shows how our technique applies the same optimization to generate code that directly computes on RLE-compressed data.

D. 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 11. 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.

When there are multiple fill regions involved in a reduction computation, we can further optimize the computation. 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)) {
        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++;
  }
```

Fig. 8: Excerpt of code that our technique generates to add two LZ77 vectors, with the result also stored in LZ77.

```
  while (piB < B1_pos[1] && piC < C1_pos[1]) {
    int32_t iB = B1_crd[piB];
    int32_t iC = C1_crd[piC];
    int32_t i = min(iB,iC);
    if (iB == i) {
      BFillValue = (&(B_vals[piB]))[0];
    if (iC == i) {
      CFillValue = (&(C_vals[piC]))[0];
    if (iB == i && iC == i) {
      A_vals[piA] = B_vals[piB] + C_vals[piC];
    } else if (iB == i) {
      A_vals[piA] = B_vals[piB] + CFillValue;
    } else {
      A_vals[piA] = BFillValue + C_vals[piC];
    }
    A_crd[piA++] = i;
    piB += (int32_t)(iB == i);
    piC += (int32_t)(iC == i);
  }
```

Fig. 9: Code that our technique generates to add two RLE vectors, with the result also stored in RLE.

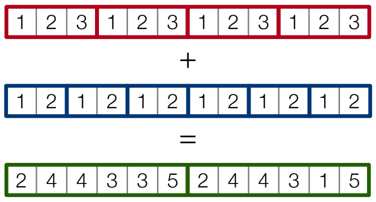


Fig. 10: When adding a vector containing repetitions of three elements to a vector containing repetitions of two elements, the resulting vector must contain repetitions of $LCM(2,3) = 6$ elements. Our technique exploits this to emit code that optimizes computations with fill values.

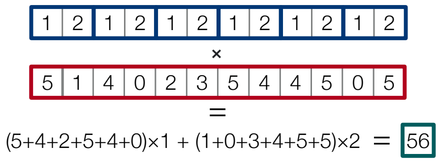


Fig. 11: When performing a reduction on two vectors, there is repeated multiplication from values within a fill region. Our technique is able to optimize reduction operations by factoring out the repeated multiplication.

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 IV-C. 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.

V. 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 [8]. We find that support for lossless compression is essential for memory usage and performance in many applications.

A. 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.

B. 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 12.

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.

C. Image Processing Applications

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

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 [14]. We report the geometric mean (geomean) speedup and size reduction compared to dense in Figure 13. Our RLE format has the largest geomean speedup of $16.3\times$ faster than dense and $16.1\times$ 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 $2.5\times$ over CSR.

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 [15]. We further filter the output image by applying a region-of-interest (ROI) mask, as done in [7]. 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 14. Using the RLE level format, we achieve a geomean speedup of $2.6\times$ over Dense, and $1.5\times$ 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 $1.9\times$ of the RLE format over OpenCV.

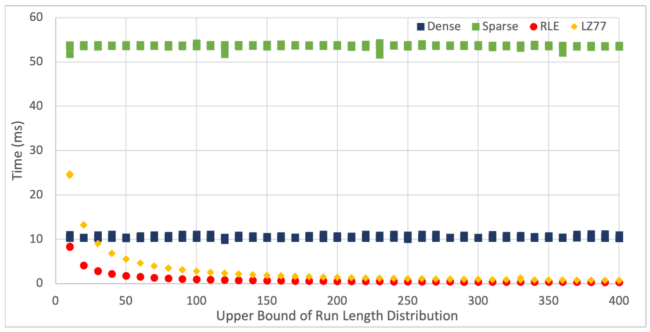
D. Video Processing Applications

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.

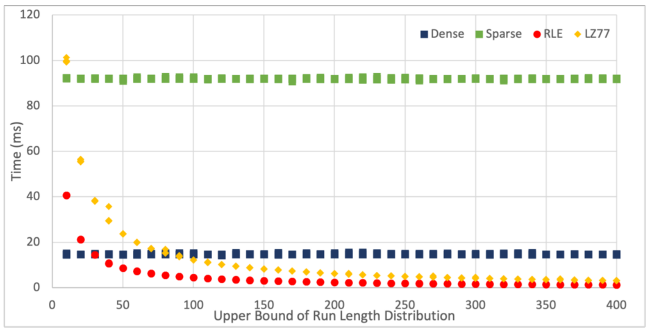
1) Brightening: This operation does element-wise addition with a constant and truncates the value at the maximum (i.e., 255, as all of the videos are in 8-bit color).

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.

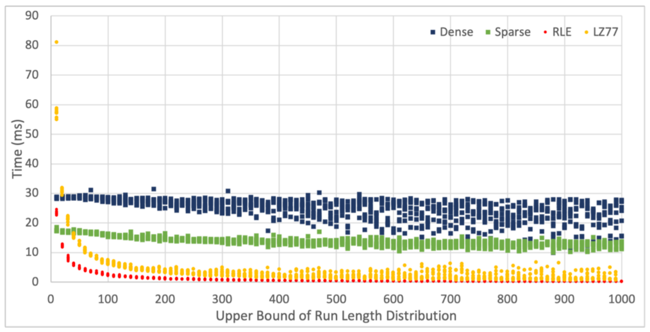
3) Results: Figure 15 shows the storage benefits of lossless compression, with both RLE and LZ77 storing up to an order of magnitude fewer explicit values than the dense representations. Figures 17 and 18 show the execution time



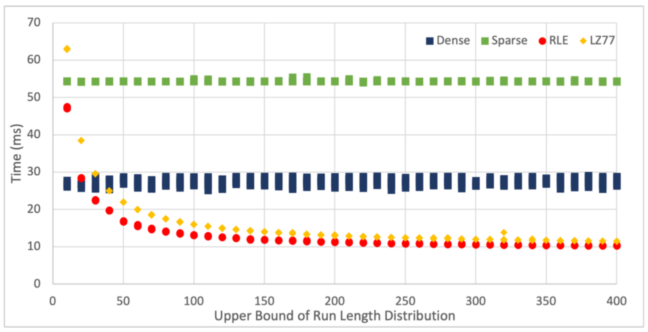
(a) Scalar multiplication



(b) Element-wise multiplication

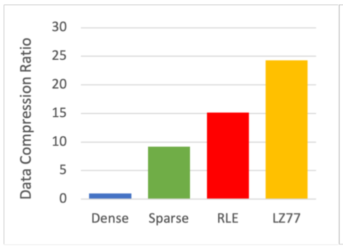


(c) Matrix-vector product

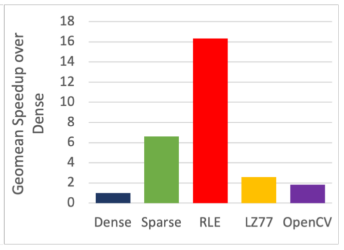


(d) Element-wise multiplication with a sparse mask

Fig. 12: Performance of micro-benchmarks on synthetic data.

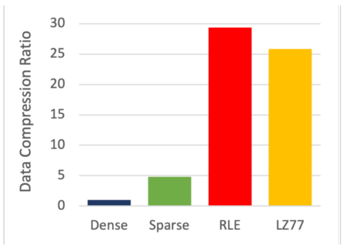


(a) Data compression ratio

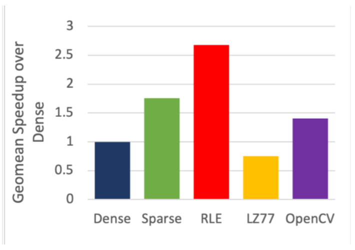


(b) Execution time speedup

Fig. 13: Results of alpha blending experiments.



(a) Data compression ratio



(b) Execution time speedup

Fig. 14: Results of edge detection experiments.

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 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 16. 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.

VI. 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.

A. *Lossless Compression*

While this work focuses on LZ77-style compression [1], 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 [16], LZW [17] and LZSS [18], which encode repetition into a dictionary to avoid redundant storage. The second main class of lossless compression algorithms is entropy encoders, includes Huffman coding [19] and Arithmetic coding [20], [21]. 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 [22], ZIP [23], gzip [24], and bzip2 [25]. There are also

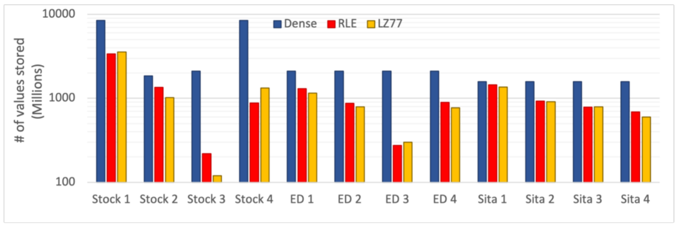


Fig. 15: Storage size of each saved file format for brighten.

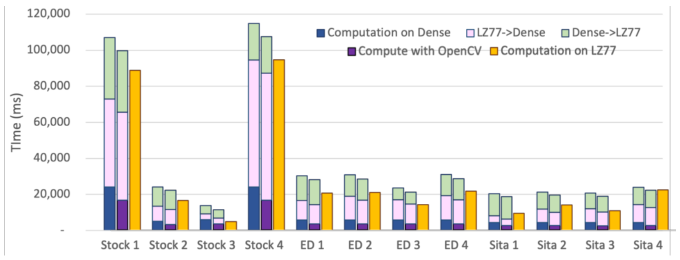


Fig. 16: Execution time of performing the subtitle computation where the files are saved in the LZ77 format.

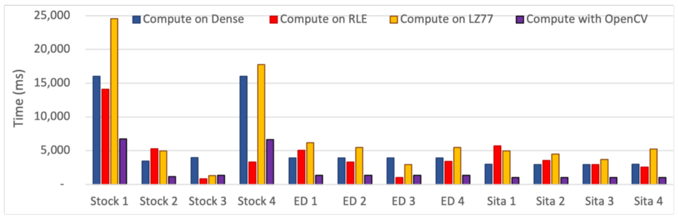


Fig. 17: Execution time of performing the brightening computation directly on the saved file format.

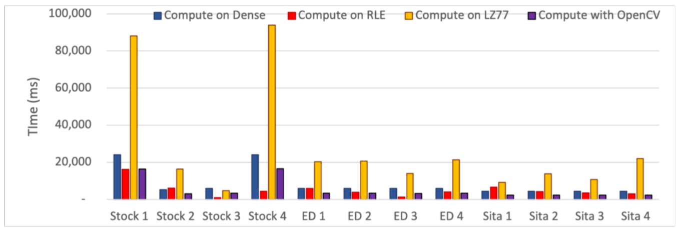


Fig. 18: Execution time of performing the subtitle computation directly on the saved file format.

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.

B. 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 [26]–[29], computing directly on compressed databases [30], [31], and computing directly on compressed video [9]. However, much of this work focuses on lossy compression schemes which use the Discrete Cosine Transform [32]–[36]. We do not consider this kind of compression in our work.

Two approaches for compressing matrices for SpMV were introduced in [37], 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 [38], 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 [39] and [40] 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 [39], 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 [38]. In [40], 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.

C. Sparse Programming

Our technique builds on the sparse tensor algebra compiler TACO [5]–[7], which implements the techniques described in Section II-C and Section IV-A. 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 [41], Bernoulli [42], SIPR [43], and CHiLL [44]. 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 [11]–[13] and CTF [45] 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 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, [46] describe a hierarchical fiber-tree abstraction for sparse tensor storage, similar to the coordinate hierarchy abstraction of [6]. The abstraction supports storing zeros using RLE, but it does not support lossless compression of nonzero elements.

VII. 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.

Acknowledgments

This research was supported by DARPA SDH Award #HR0011-18-3-0007, Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the aforementioned funding agencies.

References

[1] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, May 1977. [Online]. Available: http://ieeexplore.ieee.org/document/1055714/

[2] W. F. Tinney and J. W. Walker, "Direct solutions of sparse network equations by optimally ordered triangular factorization," *Proceedings of the IEEE*, vol. 55, no. 11, pp. 1801–1809, 1967.

[3] S. Smith and G. Karypis, "Tensor-matrix products with a compressed sparse tensor," in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*. ACM, 2015, p. 5.

[4] Intel, "Intel math kernel library developer reference," 2020. [Online]. Available: https://software.intel.com/sites/default/files/mkl-2020-developer-reference-c.pdf.pdf

[5] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–29, Oct. 2017. [Online]. Available: https://dl.acm.org/doi/10.1145/3133901

[6] S. Chou, F. Kjolstad, and S. Amarasinghe, "Format abstraction for sparse tensor algebra compilers," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–30, Oct. 2018. [Online]. Available: https://dl.acm.org/doi/10.1145/3276493

[7] R. Henry, O. Hsu, R. Yadav, S. Chou, K. Olukotun, S. Amarasinghe, and F. Kjolstad, "Compilation of Sparse Array Programming Models," p. 30.

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

[9] W. Thies, S. Hall, and S. Amarasinghe, "Manipulating lossless video in the compressed domain," in *Proceedings of the seventeen ACM international conference on Multimedia - MM '09*. Beijing, China: ACM Press, 2009, p. 331. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1631272.1631319

[10] B. W. Bader and T. G. Kolda, "Efficient MATLAB computations with sparse and factored tensors," *SIAM Journal on Scientific Computing*, vol. 30, no. 1, pp. 205–231, 2007.

[11] T. Mattson, D. Bader, J. Berry, A. Buluç, J. Dongarra, C. Faloutsos, J. Feo, J. R. Gilbert, J. Gonzalez, B. Hendrickson, J. Kepner, C. E. Leiserson, A. Lumsdaine, D. Padua, S. Poole, S. Reinhardt, M. Stonebraker, S. Wallach, and A. Yoo, "Standards for graph algorithm primitives," in *IEEE High Performance Extreme Computing Conference*. IEEE, 2013, pp. 1–2.

[12] 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)*, 2016, pp. 1–9.

[13] T. A. Davis, "Algorithm 1000: Suitesparse:graphblas: Graph algorithms in the language of sparse linear algebra," *ACM Trans. Math. Softw.*, vol. 45, no. 4, Dec. 2019. [Online]. Available: https://doi.org/10.1145/3322125

[14] M. Eitz, J. Hays, and M. Alexa, "How Do Humans Sketch Objects?" *ACM Transactions on Graphics - TOG*, vol. 31, Jul. 2012.

[15] K. Somkantha, N. Theera-Umpon, and S. Auephanwiriyakul, "Boundary detection in medical images using edge following algorithm based on intensity gradient and texture gradient features," *IEEE Transactions on Biomedical Engineering*, vol. 58, no. 3, pp. 567–573, 2011.

[16] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, 1978.

[17] T. Welch, "A technique for high-performance data compression," *Computer*, vol. 17, no. 06, pp. 8–19, jun 1984.

[18] J. A. Storer and T. G. Szymanski, "Data compression via textual substitution," *J. ACM*, vol. 29, no. 4, p. 928–951, Oct. 1982. [Online]. Available: https://doi-org.libproxy.mit.edu/10.1145/322344.322346

[19] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[20] J. J. Rissanen, "Generalized kraft inequality and arithmetic coding," *IBM Journal of Research and Development*, vol. 20, no. 3, pp. 198–203, 1976.

[21] R. Pasco, "Source coding algorithms for fast data compression (ph.d. thesis abstr.)," *IEEE Trans. Inf. Theor.*, vol. 23, no. 4, p. 548, Sep. 2006. [Online]. Available: https://doi.org/10.1109/TIT.1977.1055739

[22] "7z Format." [Online]. Available: https://www.7-zip.org/7z.html

[23] PKWare, ".ZIP File Format Specification," Jul. 2020. [Online]. Available: https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT

[24] P. Deutsch, "Rfc1952: Gzip file format specification version 4.3," USA, 1996.

[25] "bzip2 : Home." [Online]. Available: https://sourceware.org/bzip2/

[26] A. Amir, G. Benson, and M. Farach, "Let Sleeping Files Lie: Pattern Matching in Z-Compressed Files," *Journal of Computer and System Sciences*, vol. 52, no. 2, pp. 299–307, Apr. 1996. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0022000096900239

[27] 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)*, Sep. 1999, pp. 89–96.

[28] T. Gagie, P. Gawrychowski, and S. J. Puglisi, "Approximate pattern matching in LZ77-compressed texts," *Journal of Discrete Algorithms*, vol. 32, pp. 64–68, May 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1570866714000719

[29] A. Amir and C. Benson, "Efficient two-dimensional compressed matching," in *1992 Data Compression Conference*. Los Alamitos, CA, USA: IEEE Computer Society, mar 1992, pp. 279,280,281,282,283,284,285,286,287,288. [Online]. Available: https://doi-ieeecomputersociety-org.libproxy.mit.edu/10.1109/DCC.1992.227453

[30] S. Aghav, "Database compression techniques for performance optimization," in *2010 2nd International Conference on Computer Engineering and Technology*, vol. 6, 2010, pp. V6–714–V6–717.

[31] D. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '06. New York, NY, USA: Association for Computing Machinery, Jun. 2006, pp. 671–682. [Online]. Available: http://doi.org/10.1145/1142473.1142548

[32] B. Smith and L. Rowe, "Algorithms for manipulating compressed images," *IEEE Computer Graphics and Applications*, vol. 13, no. 5,

pp. 34–42, Sep. 1993, conference Name: IEEE Computer Graphics and Applications.

[33] B. C. Smith and L. A. Rowe, "Compressed domain processing of jpeg-encoded imaages," *Real-Time Imaging*, vol. 2, no. 1, pp. 3–17, 1996. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1077201496900029

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

[35] Bo Shen, I. Sethi, and V. Bhaskaran, "DCT convolution and its application in compressed domain," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 8, no. 8, pp. 947–952, Dec. 1998. [Online]. Available: http://ieeexplore.ieee.org/document/736723/

[36] G. Mandyam, N. Ahmed, and N. Magotra, "Lossless Image Compression Using the Discrete Cosine Transform," *Journal of Visual Communication and Image Representation*, vol. 8, no. 1, pp. 21–26, Mar. 1997. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1047320397903230

[37] K. Kourtis, G. Goumas, and N. Koziris, "Optimizing sparse matrix-vector multiplication using index and value compression," in *Proceedings of the 5th Conference on Computing Frontiers*, ser. CF '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 87–96. [Online]. Available: https://doi.org/10.1145/1366230.1366244

[38] K. Kourtis, G. Goumas, and N. Koziris, "Improving the performance of multithreaded sparse matrix-vector multiplication using index and value compression," in *2008 37th International Conference on Parallel Processing*, 2008, pp. 511–519.

[39] A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald, "Compressed linear algebra for declarative large-scale machine learning," *Communications of the ACM*, vol. 62, no. 5, pp. 83–91, Apr. 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3318221

[40] F. Li, L. Chen, Y. Zeng, A. Kumar, J. F. Naughton, J. M. Patel, and X. Wu, "Tuple-oriented Compression for Large-scale Mini-batch Stochastic Gradient Descent," *Proceedings of the 2019 International Conference on Management of Data*, pp. 1517–1534, Jun. 2019, arXiv: 1702.06943. [Online]. Available: http://arxiv.org/abs/1702.06943

[41] A. J. C. Bik and H. A. G. Wijshoff, "Compilation techniques for sparse matrix computations," in *International Conference on Supercomputing*. ACM, Jul. 1993, pp. 416–424.

[42] V. Kotlyar, K. Pingali, and P. Stodghill, "A relational approach to the compilation of sparse matrix programs," in *Euro-Par Parallel Processing*. Passau, Germany: Springer, 1997, pp. 318–327.

[43] W. Pugh and T. Shpeisman, "Sipr: A new framework for generating efficient code for sparse matrix computations," in *Languages and Compilers for Parallel Computing*. Springer, 1999, pp. 213–229.

[44] A. Venkat, M. Hall, and M. Strout, "Loop and data transformations for sparse matrix code," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2015, 2015, pp. 521–532.

[45] E. Solomonik and T. Hoefler, "Sparse tensor algebra as a parallel programming model," *arXiv preprint arXiv:1512.00066*, 2015.

[46] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks," *Synthesis Lectures on Computer Architecture*, vol. 15, no. 2, pp. 1–341, 2020. [Online]. Available: https://doi.org/10.2200/S01004ED1V01Y202004CAC050