Optimizing Compression Schemes for Parallel Sparse Tensor Algebra

Helen Xu
Lawrence Berkeley National Laboratory
hjxu@lbl.gov

Tao B. Schardl
Massachusetts Institute of Technology
neboat@mit.edu

Michael Pellauer
NVIDIA
mpellauer@nvidia.com

Joel S. Emer
NVIDIA/MIT
jsemer@mit.edu

***Abstract*—This paper studies compression techniques for parallel in-memory sparse tensor algebra. We find that applying simple existing compression schemes can lead to performance loss in some cases. To resolve this issue, we introduce an optimized algorithm for processing compressed inputs that can improve both the space usage as well as the performance compared to uncompressed inputs. We implement the compression techniques on top of a suite of sparse matrix algorithms generated by taco, a compiler for sparse tensor algebra. On a machine with 48 hyperthreads, our empirical evaluation shows that compression reduces the space needed to store the matrices by over $2\times$ without sacrificing algorithm performance.**

***Index Terms*—compression, tensor algebra, parallel, multithreaded**

I. Introduction

Sparse matrix and tensor algebra operations underlie many applications in important domains such as scientific computing [1], data science [2], and graph analytics [3]. Tensors generalize vectors and matrices to arbitrary dimensions. Tensors from these domains are large and highly sparse–that is, almost all of their elements are zeroes. Because of the potential to leverage *ineffectual* multiplications by avoiding math and data transfers, considerable research effort has been devoted to processing them efficiently in parallel on a single shared-memory multicore. This paper studies compression mechanisms to improve the running time and space usage of parallel sparse tensor algebra in general.

The taco sparse tensor algebra compiler [4] provides a general mechanism to generate optimized implementations of arbitrary Einstein summations (einsums) [5] on sparse tensors of arbitrary dimensions. To handle the wide variety of sparse tensor representations, taco introduces the *level format* abstraction [6], which succinctly describes these representations as compositions of a few simple row-based formats.

But taco's existing row-based formats do not achieve the same space savings as state-of-the-art workload-specific compression schemes such as those for Sparse Matrix-Vector multiplication (SpMV) [7], [8], [9], [10], [11], [12], [13], [14]. Many compression schemes for SpMV are based on *delta encoding*, a classical technique that stores differences between consecutive elements [15]. Previous work introduces compression formats and decompression codes to take advantage of matrix structure [11], [12], [13], [14]. These schemes change the representation to fit substructures in specific matrices or generate code tailored to each matrix, which improves performance but requires additional preprocessing time. Furthermore, some schemes take advantage of two-dimensional patterns such as blocking.

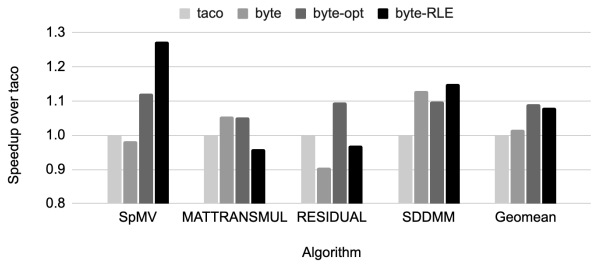
In this work, we demonstrate how to apply row-based compression schemes previously described for SpMV across generalized einsums, which requires tackling complex tradeoffs between performance and space. Directly applying simple compression schemes can trade space savings for performance on sparse matrix algorithms even when run in parallel. Sparse tensor algebra is often memory-bound [16], [17], and so intuitively there should be idle processor cycles available to deal with more advanced compression schemes without performance loss, or even with performance gain due to reduced memory transfers as compression rate improves. However, complex compression formats can become latency-bound depending on the structure of the input matrices [18]. Therefore, simply compressing the tensor representation may result in slowdown on rows containing only a few nonzeroes.

To overcome this performance loss due to compression, we introduce *byte-opt*, a novel optimized version of the *byte* format from the Ligra+ graph-processing framework [19] that saves space without sacrificing performance. The byte-opt format takes advantage of per-row structure during decoding without changing the underlying representation from byte. The Ligra+ paper also describes the *byte-RLE* format, which takes advantage of per-row structure, but changes the byte format to improve performance at the cost of space [8], [9], [10].

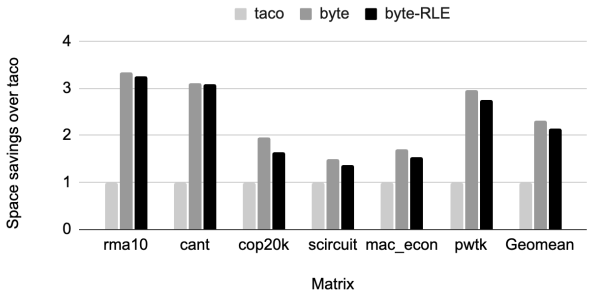
To experiment with a variety of sparse matrix workloads beyond SpMV, we integrate the *encoded formats* — byte, byte-opt, and byte-RLE — on top of a suite of codes generated from taco [4]. We use the original taco code as a baseline, which represents sparse matrices in the classical Compressed Sparse Row (CSR) [20] format. The original taco paper demonstrated that taco is competitive with or outperforms other state-of-the-art sparse linear algebra libraries such as MKL, uBLAS, and others.

Summary of results. Figure 1[1] demonstrates that the encoded formats are always smaller and generally faster than the baseline CSR representation. On average, the byte and byte-RLE formats are $2.3\times$ and $2.1\times$ smaller, respectively, than CSR. Meanwhile, although the encoded formats are on average

[1]We omit byte-opt from Figure 1(b) because it uses the same amount of space as byte.



(a) Speedup over original taco on all (48) hyperthreads.



(b) Space savings over original taco.

Fig. 1: Performance and space comparison with baseline taco.

about $1.1\times$ faster than CSR, there are algorithms that are substantially slower on byte and byte-RLE when compared to CSR. In contrast, byte-opt is always faster than the baseline while achieving the same space savings as byte.

Related work. Recently, Donenfeld *et al.* extended taco with RLE and LZ compression [21]. This work demonstrates the generality of the taco framework and the "level formats" abstraction, but the exact compression schemes are targeted for video applications and not for SpMV-like computations.

The SMASH compression format [22] for sparse matrix computations was shown to be slightly faster than taco's baseline CSR representation, but was studied sequentially. The SMASH paper focused on co-designing compression techniques for both hardware and software, while this paper focuses on software.

Finally, Shun *et al.* extended the Ligra graph-processing framework to run graph algorithms on encoded inputs [19]. Graph algorithms are also often memory-bound, so we draw inspiration from the compressed graphs literature for the compression schemes in this paper. However, the focus of this paper is on sparse matrix and tensor algebra.

II. Background

This section reviews preliminaries necessary to understand the compressed tensor representations in this paper. First, it defines tensor notation. Next, it describes the level format tensor representation abstraction and how to process formats with their "capabilities." Finally, it introduces the encoded formats that this paper studies.

Tensor notation. *Tensors* are multi-dimensional arrays of arbitrary *order* (dimensionality) N over some field $\mathbb{F}$, usually the real or complex numbers. Matrices are the special case of $N = 2$. Given a dimension i of a tensor A, we denote the *shape*, or number of possible elements along that dimension, with S_i. Therefore, an order-N tensor A can be formalized as $A \in \mathbb{F}^{S_1 \times S_2 \times \ldots \times S_N}$.

We address elements in a tensor with *points*, which are N-tuples of *coordinates* (one for each dimension). A coordinate in dimension i of a tensor A is an integer $x_i \in \{0, 1, \ldots, S_i - 1\}$. Each point has a *value* from $\mathbb{F}$. Furthermore, we denote the number of points in a tensor A with nonzero values as $\texttt{NNZ}(A)$.

Algorithm 1

`append_coord(`$\texttt{p}_k$`,` $\texttt{i}_k$`):`

1: `crd[`$\texttt{p}_k$`] =` $\texttt{i}_k$

Algorithm 2

`append_edges(`$\texttt{p}_k$`,` $\texttt{pbegin}_k$`,` $\texttt{pend}_k$`):`

1: `pos[`$\texttt{p}_k$`] =` $\texttt{pend}_k$ `-` $\texttt{pbegin}_k$

We use uppercase letters (e.g., A) to denote matrices, lowercase letters (e.g., x) to denote vectors, and Greek letters (e.g., α) to denote scalars.

Format abstractions. A core part of the taco code-generation algorithm [4] is the *coordinate hierarchy* abstraction that composes level formats to define storage formats for tensors of arbitrary dimensions [6]. The coordinate hierarchy conceptually arranges a tensor's coordinates in a tree where each level stores the coordinates along one dimension of the tensor. Each root-to-leaf path in this abstraction reconstructs a point in the tensor. *Level formats* define how coordinates are stored (i.e., which types of data structures are used) at each level of the coordinate hierarchy.

The coordinate hierarchy and level-format abstraction provide a unified formal description that can be used to generate many canonical sparse matrix and tensor representations. For example, let us consider how to create the classical Compressed Sparse Row (CSR) [20] for sparse matrices with level formats. Storing a dimension i with the *dense* level format implicitly represents all possible coordinates in that dimension $x_i \in \{0, 1, \ldots, S_i - 1\}$ by storing all of its values. In contrast, the *compressed* level format explicitly stores only the coordinates with nonzero values. CSR can be described as the composition of the row dimension stored in the dense level format and the columns stored in the compressed level format. Level formats can describe many other tensor representations, such as COO, CSC, CSF, etc.

Level format capabilities. Each level format comes with a

Algorithm 3 `pos_bounds(p_k)`:

1: **return** `<pos[p_k], pos[p_k + 1]>`

Algorithm 4 `pos_access(p_k)`:

1: **return** `crd[p_k]`

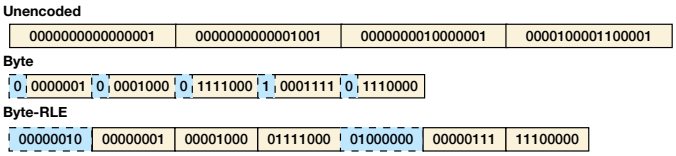


Fig. 2: An example of an unencoded list and how to compress it using with delta encoding using the byte and byte-RLE formats. In this example, coordinates can be up to 16 bits, and the byte and byte-RLE formats use 8-bit chunks. The blue dotted blocks in the byte format denote the continue bit. The blue dotted blocks in the byte-RLE format store the headers in 8 bits. The high-order 2 bits encode the number of bytes (up to 4) of each element in that group, and the remaining 6 bits encode the number of elements in the following group. For each of these values, the header stores the value minus one.

set of *capabilities* that define how to assemble and access it [6]. We describe the capabilities and implementation of the compressed level format since it is the starting point for the encoded formats in this paper. The compressed level format stores coordinates explicitly in a dimension in a `crd` array explicitly along with a `pos` array to keep track of the segment bounds. Assembling the two arrays uses two capabilities:

- The `append_coord` capability adds coordinates in a level at the end of the `crd` array (Algorithm 1).
- The `append_edges` capability stores the start and end of each row in the `pos` array (Algorithm 2).

Sparse tensor algebra relies on **efficient iteration through the nonzero coordinates**, which underlies core primitives such as intersection. The compressed level format supports coordinate position iteration with the following capabilities:

- The `pos_bounds` capability retrieves the locations of coordinates in a row (Algorithm 3).
- The `pos_access` capability accesses the coordinates in a row by position (Algorithm 4).

Encoded formats. *Delta encoding* is a canonical data compression technique that stores differences (deltas) between sequential elements rather than the full element [15]. Given a sorted array x of n elements, delta encoding results in a new array x' such that $x'[0] = x[0]$ and for all $i = 1, 2, \ldots, n-1$, $x'[i] = x[i] - x[i-1]$.

The *byte* format stores the deltas in byte codes [19], [23]. Byte codes are a type of *variable-length encoding*, or k-bit codes, which stores an integer as a series of k-bit *chunks* (here, $k = 8$). Each chunk uses one bit as a *continue bit*, which indicates if the following chunk starts a new element or is a continuation of the previous element.

Algorithm 5 `append_coord_byte(p_k, c_k, i_k, i_{k-1})`:

1: `clength = encode_elt(i_k - i_{k-1})`
2: `c_k += clength`
3: `p_k++`

Algorithm 6
`append_edges_byte(row, pbegin_k, pend_k, cbegin_k, cend_k)`:

1: `val_pos[row] = pend_k - pbegin_k`
2: `crd_pos[row] = cend_k - cbegin_k`

Algorithm 7 `pos_bounds_byte(p_k)`:

1: **return** `<crd_pos[p_k], crd_pos[p_k + 1], val_pos[p_k], val_pos[p_k + 1]>`

Algorithm 8
`pos_access_byte(c_k)`:

1: **return** `decode_elt(crd, c_k)` ▷ Return the delta and how many bytes it took

The *byte-RLE* format [19], [8], [9], [10] avoids the need for a continue bit by grouping consecutive deltas that require the same number of bytes (8-bit chunks). Each group starts with an 8-bit header indicating the number of chunks each element requires and the number of elements in the group.

Figure 2 presents a worked example of these encoded formats.

III. Implementation

This section describes how to implement the encoded formats on top of sparse tensor algebra codes generated by taco.

Storage of byte and byte-RLE. The compressed level format originally used one `pos` array to store the offsets of each row in the `crd` array as well as the values array since they were the same offset. However, coordinates stored in byte and byte-RLE have variable size, which dissociates the offsets into the `crd` and `vals` arrays. Therefore, we replace the original `pos` array with two arrays: `crd_pos` and `val_pos`. The `val_pos` array stores the offsets into the values array, whereas the `crd_pos` array stores the offsets into the coordinate array stored in byte or byte-RLE format.

To store sparse matrices, we start with CSR and replace the compressed level format in the columns with the encoded formats.

Encoding. We describe how to build only the byte level format for simplicity; building the byte-RLE level format is similar. Instead of appending full coordinates, the byte level format appends deltas between coordinates. Furthermore, it must keep track of positions in both the coordinate and value array separately (Algorithm 5). To append an entire row to the tensor, the byte level format stores the start and end of

Algorithm 9

`read_row_byte(p_k):`

```
1: elt_so_far = 0
2: crd_idx, crd_end, pos_idx, pos_end =
   pos_bounds_byte(p_k)
3: while crd_idx < crd_end do
4:     delta, clength = pos_access_byte(crd_idx)
5:     elt_so_far += delta
6:     [Process coordinate]
7:     crd_idx += clength
8:     pos_idx++
9: end while
```

Algorithm 10

`read_row_byte_opt(p_k):`

```
1: while crd_idx < crd_end do
2:     if  crd_idx < crd_end - 4 && ((crd + crd_idx)
   & 0x80808080UL) == 0 then
3:         [Decode and process next four coordinates]
4:     else
5:         [Decode and process next coordinate]
6:     end if
7: end while
```

that row in both the `val_pos` array and `crd_pos` array (Algorithm 6).

Decoding. The encoded level formats support coordinate position iteration through the deltas between coordinates. The row iteration keeps track of the latest decoded coordinate to reconstruct the next coordinate from the next delta (Algorithm 9). The level formats access the start and end of the row in both the `crd` and `vals` arrays (Algorithm 7) and then decode the deltas one-by-one in that row (Algorithm 8).

Optimizing byte-format decoding. Although the byte format saves a significant amount of space over CSR, it incurs additional computational overhead to decode the continue bit in each byte and is therefore often slower than the baseline.

To reduce decoding overhead in the common case, we introduce *byte-opt*, a level format that uses the same representation as byte but optimizes the byte format's decoding operation by taking advantage of consecutive small deltas. Specifically, we note that one-byte differences do not require decoding the continue bit, so they can just be read directly from the `crd` array. Therefore, byte-opt contains a fast path in the decode loop to read four one-byte deltas at a time (Algorithm 10).

This decoding optimization takes advantage of naturally occurring consecutive groups of small deltas. For example, in the rma10 matrix used in the taco evaluation, about 87% of the four-byte groups contained only one-byte deltas. Similarly, in the cant matrix, about 81% of the deltas were one byte each. Although both byte-opt and byte-RLE take advantage of consecutive similarly-sized deltas, byte-opt avoids the space overhead of byte-RLE by applying this observation to the decode loop.

IV. Evaluation

This section evaluates the original CSR representation in taco as well as the encoded formats — byte, byte-opt, and byte-RLE — in terms of algorithm performance and space usage. We evaluate all formats on a suite of real-world compound linear algebra applications from the taco paper: SpMV, MATTRANSMUL, RESIDUAL, SDDMM (sampled dense-dense matrix multiplication). We compare the encoded formats to the original taco CSR representation to isolate the effects of compression on performance and space.

We find that the encoded formats slightly improve performance while significantly reducing space usage by over $2\times$. In parallel, the byte and byte-RLE formats trade off between space and performance: the byte format is slower but smaller compared to byte-RLE. The decoding optimization in byte-opt overcomes this tradeoff: it uses the same representation as byte, but is on average slightly faster than byte-RLE.

Experimental setup. We implemented all encoded formats in `C++` on top of the artifact from the taco paper [4]. We used 32-bit unsigned integers to store coordinates in the taco level formats (uncompressed and compressed) and bytes to store coordinates in the encoded formats. We parallelized the algorithms using Cilk [24] and the Tapir/LLVM [25] compiler based on LLVM [26] 14.

All experiments were run on a 48-core 2-way hyperthreaded Intel® Xeon® Platinum 8275CL CPU @ 3.00GHz with 189 GB of memory from AWS [27]. The machine has 1.5MiB of L1 cache, 48 MiB of L2 cache, and 71.5 MiB of L3 cache across all of the cores. To avoid non-uniform memory access (NUMA) issues across sockets, we ran all experiments on a single socket with 24 physical cores and 48 hyperthreads.

All times are the median of 5 trials after one warm-up trial. To clear the cache between runs, we read a large unrelated array bigger than the size of L3.

Inputs. Table I reports the sizes of all matrices used in the evaluation. We report the space used to store the coordinates. All tested matrices were gathered from the SuiteSparse Matrix Collection [28]. For all workloads except SDDMM, we use the same matrices from the compound linear algebra evaluation in taco (above the line in Table I). Since the corresponding dense matrices for SDDMM did not fit in memory, we chose several other matrices from SuiteSparse (below the line in Table I).

We generated random uniform 64-bit weights for all inputs in the evaluation. The magnitude of the weights does not matter because the values are uncompressed.

Algorithm performance. Tables II and III report the serial and parallel running times for each algorithm on each input matrix. Figure 1(a) reports the speedup relative to the baseline of original taco in parallel, and Figure 3(a) reports the speedup relative to the original taco in serial. We report the time to perform the algorithm without the time to build the representations because the cost of compressing the representation can be amortized over multiple sparse matrix computations. Furthermore, the compression step just involves one parallelizable pass through the data [19].

Matrix	*Rows*	*NNZ*	*NNZ/rows*	*orig.*		*byte*			*byte-RLE*		
				Size	*Bytes/nnz*	*Size*	*Bytes/nnz*	*C.R.*	*Size*	*Bytes/nnz*	*C.R.*
rma10	4.68E4	2.37E6	50.69	9.68	4.08	**2.90**	1.22	3.34	2.97	1.25	3.26
cant	6.25E4	2.03E6	32.58	8.39	4.12	**2.70**	1.33	3.10	**2.70**	1.33	3.10
cop20k	1.21E5	1.36E6	11.24	5.93	4.36	**3.04**	2.23	1.95	3.63	2.67	1.63
scircuit	1.71E5	9.59E5	5.61	4.52	4.71	**3.01**	3.14	1.50	3.32	3.46	1.36
mac_econ	2.07E5	1.27E6	6.17	5.92	4.65	**3.49**	2.74	1.70	3.84	3.02	1.54
pwtk	2.18E5	5.93E6	27.19	24.6	4.15	**8.31**	1.40	2.96	8.95	1.51	2.75
G66	9.00E3	1.80E4	2.00	0.108	6.00	**0.0988**	5.49	1.09	0.108	5.98	1.00
har_10NN	1.03E4	7.59E4	7.37	0.345	4.54	**0.194**	2.56	1.77	0.214	2.82	1.61
Kuu	7.10E3	1.74E5	24.45	0.723	4.16	**0.238**	1.37	3.04	0.246	1.42	2.94
fashion_mnist	1.00E4	7.92E4	7.92	0.357	4.51	**0.221**	2.79	1.62	0.245	3.10	1.46
nemeth26	9.51E3	7.61E5	80.02	3.08	4.05	**0.846**	1.11	3.64	0.863	1.14	3.57

TABLE I: Matrix sizes and size (in MB) of the representation of the coordinate hierarchy in original taco (orig.), byte, and byte-RLE (excluding values). C.R. denotes the compression ratio (space savings) compared to CSR. The bolded sizes are the smallest in each row.

In serial, on average across all workloads, we find that byte is slower than the baseline, byte-opt is similar to the baseline, and byte-RLE is slightly faster than the baseline. The encoded formats were originally described specifically for SpMV because of SpMV's importance in scientific applications. However, all of the encoded formats are slower (between $1.1\times$–$1.4\times$ slower) than the baseline on SpMV because of the relative cost of decoding the sparse matrix compared to the other work in SpMV. Both byte and byte-opt are slower than byte-RLE for the SpMV-based algorithms (MATTRANSMUL and RESIDUAL) because byte-RLE has lower decoding overhead. Finally, the performance of SDDMM is similar for all of the formats because the kernel involves one dense dot product per nonzero, so the work is dominated by the matrix multiplication. In contrast, the SpMV-based workloads just perform one multiply and add per nonzero. We suspect that the serial execution of these algorithms cannot saturate the CPU's available memory bandwidth, which limits how much encodings can improve performance by reducing space.

In parallel, on average across all workloads, the encoded formats are faster than the baseline. However, the encoded formats are slower than the baseline on matrices with relatively few nonzeroes per row (e.g., cop20k, scircuit, and mac_econ) because the resulting compression ratio is also small, so the space savings are insufficient to outweigh the additional computational overhead. In contrast, the matrices with better compression ratio (e.g., rma10, cant, and pwtk) tend to result in better performance with the encoded formats.

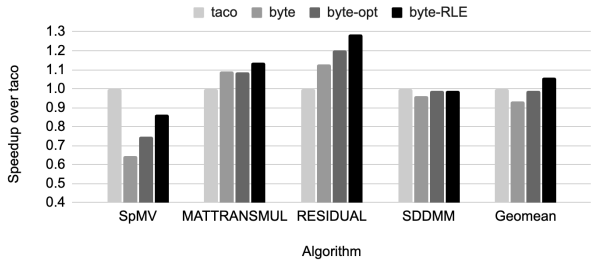
In general, the encoded formats achieve better performance-per-byte both in serial and in parallel compared to the baseline. For example, for parallel SpMV on rma10, CSR's time-space product is 0.89 (ms) $\times$ 4.08 (bytes per nonzero) = 3.63, whereas byte's time-space product is is 1.07 (ms) $\times$ 1.22 (bytes per nonzero) = 1.30.

Figure 3(b) reports the self-relative parallel speedup (T_1/T_{48}) for each format averaged across inputs for each workload. On SpMV, the baseline achieves $3.2\times$ speedup, while the encoded formats achieve up to $4.9\times$ speedup. Similarly, on SDDMM, the baseline achieves $14\times$ speedup, while the encoded formats achieve up to $16\times$ speedup. SpMV has relatively little parallelism compared to SDDMM, so all formats have less speedup on SpMV. In both cases, the encoded formats achieve better speedup than the baseline because they mitigate memory bandwidth limitations in parallel.

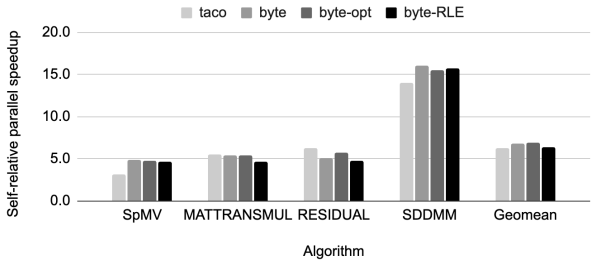
Space usage. Figure 1(b) and Table I report the space usage of the matrices in all of the formats. Both the byte and byte-opt formats use the byte representation, since byte-opt only changes the decode routine. On the matrices from the taco evaluation, the byte format is about $2.3\times$ smaller than CSR, whereas the byte-RLE format is about $2.1\times$ smaller than CSR. On the matrices used for SDDMM, both byte and byte-RLE are about $2\times$ smaller than CSR. Many of the matrices (e.g., cant, pwtk, nemeth, etc.) are narrow-banded or consist of narrow-banded blocks. This naturally-occuring matrix structure improves the compression ratio since nonzero elements are packed in nearby locations. Furthermore, the compression ratio improves with the number of nonzeroes per row because every matrix representation in this paper stores the row dimension in an uncompressed level format. The metadata required for the encoded formats at the row level is twice that of the compressed representation, but overall the compressed formats save space by reducing the size of the column level.

V. Conclusion

This work demonstrates the potential for compression schemes from the SpMV literature to speed up other types of sparse linear algebra while saving space. Using the byte, byte-opt, and byte-RLE formats on top of code generated from taco, we are able to significantly reduce the space usage of the sparse matrices without sacrificing parallel performance. Future work includes integrating these compression formats into taco as well as exploring the impact of compression on tensor algebra in higher dimensions.



(a) Speedup over original taco when run on one hyperthread.



(b) Self-relative parallel speedup on 48 hyperthreads (T_1/T_{48}).

Fig. 3: Serial performance compared to original taco and self-relative parallel speedup.

Matrix	*SpMV* ($y = Ax$)								*MATTRANSMUL* ($y = \alpha A^T x + \beta z$)							
	orig.		*byte*		*byte-opt*		*byte-RLE*		*orig.*		*byte*		*byte-opt*		*byte-RLE*	
	T_1	T_{48}	T_1	T_{48}	T_1	T_{48}	T_1	T_{48}	T_1	T_{48}	T_1	T_{48}	T_1	T_{48}	T_1	T_{48}
rma10	4.26	0.89	4.64	1.07	3.47	0.86	2.89	**0.71**	4.30	0.86	3.56	0.83	3.23	0.77	2.88	**0.74**
cant	3.57	0.77	3.70	0.99	2.69	0.75	2.22	**0.63**	5.81	0.74	2.64	0.70	2.50	**0.64**	2.24	**0.64**
cop20k	3.74	**0.85**	7.56	1.14	7.70	1.14	6.59	1.08	5.94	**0.84**	5.37	0.88	5.70	0.97	5.69	1.04
scircuit	2.24	**0.81**	4.87	0.88	5.02	0.89	4.62	0.86	4.55	0.62	4.34	**0.60**	4.82	0.69	4.90	0.86
mac_econ	1.90	2.62	3.55	0.86	3.52	0.86	3.02	**0.77**	2.03	0.78	2.92	**0.56**	3.17	0.58	3.17	0.79
pwtk	7.03	1.28	10.62	1.89	7.81	1.42	6.86	**1.17**	6.45	1.15	8.10	1.22	7.74	**1.14**	7.10	1.15

Matrix	*RESIDUAL* ($y = b - Ax$)							
	orig.		*byte*		*byte-opt*		*byte-RLE*	
	T_1	T_{48}	T_1	T_{48}	T_1	T_{48}	T_1	T_{48}
rma10	4.32	0.83	4.07	0.97	3.24	0.75	2.93	**0.73**
cant	5.79	0.96	3.09	0.85	2.57	**0.61**	2.25	0.69
cop20k	5.44	**0.86**	7.13	1.00	7.19	0.98	6.86	1.12
scircuit	4.46	0.75	4.41	0.76	4.86	**0.70**	4.86	0.88
mac_econ	4.86	0.64	2.99	0.64	3.27	**0.59**	3.16	0.77
pwtk	7.45	**1.09**	9.00	1.62	7.61	1.10	6.93	1.13

TABLE II: Serial (T_1) and parallel (T_{48}) running times (in milliseconds) of compound linear algebra applications with the original taco (orig.) and the encoded formats. The bolded times are the lowest for each matrix on each workload.

Matrix	*orig.*		*byte*		*byte-opt*		*byte-RLE*	
	T_1	T_{48}	T_1	T_{48}	T_1	T_{48}	T_1	T_{48}
G66	188.90	18.15	208.52	17.89	190.19	18.52	190.19	**16.99**
har_10NN	886.40	55.10	939.71	54.64	920.82	53.18	904.75	**53.17**
Kuu	1,302.46	68.20	1,296.83	62.65	1,313.03	**61.95**	1,313.94	64.26
fashion_mnist	921.59	86.31	944.83	59.77	912.23	59.42	912.92	**58.96**
nemeth26	7,515.85	474.89	7,651.98	476.06	7,538.73	477.28	7,623.50	**474.20**

TABLE III: Serial (T_1) and parallel (T_{48}) running times (in milliseconds) of SDDMM with the original taco (orig.) and the encoded formats. The bolded times are the lowest in each row.

ACKNOWLEDGMENT

Research was sponsored by the United States Air Force Research Laboratory and the Department of the Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Department of the Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

REFERENCES

[1] Y. Saad, *Iterative methods for sparse linear systems*, 2nd ed. SIAM, 2003.

[2] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *ISCA*, 2017.

[3] T. Mattson *et al.*, "Standards for graph algorithm primitives," in *HPEC*, 2013.

[4] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *OOPSLA*, 2017.

[5] M. Ricci and T. Levi-Civita, "Méthodes de calcul différentiel absolu et leurs applications," *Mathematische Annalen*, 1900.

[6] S. Chou, F. Kjolstad, and S. Amarasinghe, "Format abstraction for sparse tensor algebra compilers," *OOPSLA*, 2018.

[7] J. Willcock and A. Lumsdaine, "Accelerating sparse matrix computations via data compression," in *ICS*, 2006.

[8] K. Kourtis, G. Goumas, and N. Koziris, "Exploiting compression opportunities to improve SpMxV performance on shared memory systems," *TACO*, 2011.

[9] K. Kourtis *et al.*, "Improving the performance of multithreaded sparse matrix-vector multiplication using index and value compression," in *ICPP*, 2008.

[10] K. Kourtis, G. Goumas, and N. Koziris, "Optimizing sparse matrix-vector multiplication using index and value compression," in *CF*, 2008.

[11] M. Belgin *et al.*, "Pattern-based sparse matrix representation for memory-efficient SMVM kernels," in *ICS*, 2009.

[12] K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris, "CSX: an extended compression format for spmv on shared memory systems," in *PPOPP*, 2011.

[13] G. E. Blelloch, I. Koutis, G. L. Miller, and K. Tangwongsan, "Hierarchical diagonal blocking and precision reduction applied to combinatorial multigrid," in *SC*, 2010.

[14] A. Buluç, S. Williams, L. Oliker, and J. Demmel, "Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication," in *IPDPS*, 2011.

[15] S. W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, 1997.

[16] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris, "Understanding the performance of sparse matrix-vector multiplication," in *PDP*, 2008.

[17] W. K. Anderson, W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith, "Achieving high sustained performance in an unstructured mesh CFD application," in *SC*, 1999.

[18] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris, "Performance evaluation of the sparse matrix-vector multiplication on modern architectures," *The Journal of Supercomputing*, 2009.

[19] J. Shun, L. Dhulipala, and G. E. Blelloch, "Smaller and faster: Parallel processing of compressed graphs with Ligra+," in *DCC*, 2015.

[20] W. F. Tinney and J. W. Walker, "Direct solutions of sparse network equations by optimally ordered triangular factorization," *Proceedings of the IEEE*, 1967.

[21] D. Donenfeld, S. Chou, and S. Amarasinghe, "Unified compilation for lossless compression and sparse computing," in *CGO*, 2022.

[22] K. Kanellopoulos *et al.*, "SMASH: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations," in *MICRO*, 2019.

[23] I. H. Witten, A. Moffat, and T. C. Bell, *Managing Gigabytes (2nd Ed.): Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers Inc., 1999.

[24] Intel Corporation, *Intel Cilk Plus Language Specification*, 2010.

[25] T. B. Schardl, W. S. Moses, and C. E. Leiserson, "Tapir: Embedding recursive fork-join parallelism into LLVM's intermediate representation," *TOPC*, 2019.

[26] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *CGO*, 2004.

[27] Amazon, "Amazon web services," https://aws.amazon.com/.

[28] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *TOMS*, 2011.