Assessing high-performance lightweight compression formats for Geospatial computation

Omar Tanner

Darwin College, University of Cambridge

This project expedites geospatial data processing by applying modern lightweight compression techniques to GeoTIFF-stored data, addressing the challenge of CPU bandwidth surpassing RAM bandwidths. We mitigate the impact of poor cache locality and the resulting memory bottlenecks by leveraging CPUs' superscalar capabilities and SIMD (Single Instruction, Multiple Data) instructions. By implementing SIMD-optimised compression, data remains compressed in RAM and closer to the CPU caches, facilitating faster access and alleviating memory constraints. Through multi-objective Pareto optimisation, the project identifies optimal compression schemes based on space and time efficiency. This approach achieves up to 29% speedups relative to uncompressed pipelines, which further increases to 78% when aggregations are fused into the decompression process. These speedups expedite the management and analysis of large datasets, crucial for addressing environmental challenges like climate change and ecosystem conservation. The project also comprehensively benchmarks and ranks compression algorithms on geospatial data, providing a resource for optimising geospatial pipelines and directing future research. The code used in our project is available at https://github.com/omarathon/compression-geospatial.

Keywords: **Data compression, remote sensing, geospatial computation, vectorisation, multi-objective optimisation, computer architecture**

Acknowledgements

My sincere thanks to my supervisors Prof. Anil Madhavapeddy and Dr. Sadiq Jaffer for their time, guidance, and support. I would like to extend my thanks to Cambridge University's HPC team for allocating 200,000 hours of compute time, 8000 of which were essential for my research. Finally, I would like to thank the Sensor CDT and the EPSRC for their funding, making this project possible.

1 Introduction

1.1 Research Problem and Project Aim

Petabytes of data collected from remote sensors such as satellites is often stored in the GeoTIFF file format. Geospatial piplines process this data and transform it into geographical insights [1]. These pipelines read/decode the data from the disk into RAM, transform it via operations on the CPU or GPU, and write/encode the transformed data back to RAM. Since CPU bandwidth already dominates memory bandwidth and their gap is ever increasing [2], the IO portion of the pipeline (encoding and decoding) bottleneck its overall processing speed [3]. This bottleneck is present in geospatial processing pipelines [4] and databases [5, 6]. Furthermore, CPU bandwidth dominates RAM bandwidth, leading a bottleneck from RAM. While modern CPUs are equipped with fast caches, transformations applied to geospatial data may not be cache-efficient [7]. Thus, data is read from slower, lower levels of the memory hierarchy, eventually reaching RAM and slowing down the pipeline [8], as shown in Figure 1. We aim to address the RAM bottleneck because the IO bottleneck has already been extensively addressed in geospatial pipelines [3, 5, 6, 9, 10].

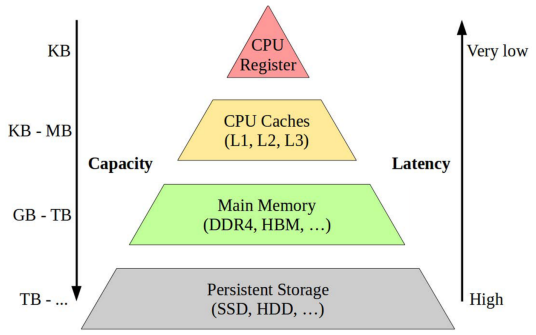


Figure 1: Pohl *et al.*'s [11] depiction of the computer memory hierarchy. We aim to move the data up the hierarchy, from RAM (highlighted in green) to the CPU caches/registers (highlighted in orange/red).

We aim to address the bottleneck by applying lightweight compression techniques to keep the data compressed, reducing the memory footprint, and selectively decompressing and re-compressing portions when operating on it. Lightweight compressors have grown in strength by exploiting the superscalar capabilities of modern CPUs [3], and often feature in database systems [6]. Now, they can outperform generic, 'heavyweight' techniques [3]. By keeping the data compressed in RAM, we reduce the impact of CPU cache misses, mitigating the impact of the bottleneck. We also aim to further exploit compression by transforming the data in its compressed form. By fusing operations on the data within decompression [12], we save CPU cycles and improve cache-locality, further mitigating the bottleneck.

1.2 Literature Review

Our review begins with an overview of how geospatial data processing. It examines key tools and formats like GeoTIFF and GDAL, and evaluates existing solutions for data processing challenges. We then shift the focus to explore compression algorithms, assessing them for efficiency and suitability. Lastly, we identify a gap in the

literature, highlighting the need for data compression in geospatial pipelines.

1.2.1 Geospatial Pipelines

In an era of geospatial 'big data' [13], remote sensors produce petabytes of geospatial data that is processed in computational pipelines, as depicted in Figure 2. These 'geospatial pipelines' comprise storing, processing and analysing geospatial data to produce insights for users such as scientists and policy-makers. As detailed by Holcomb *et al.* [1], existing cloud-based end-to-end systems comprise Google's Earth Engine [14] and Microsoft's Planetary Computer[1]. Despite their simple user interface, they lack flexibility and access guarantees. Custom end-to-end systems have been developed for greater control of the underlying systems. These often harness the Hadoop framework[2] with the MapReduce programming model to process the data in a distributed fashion. While useful in cluster computing contexts for e.g. data mining [13], these technologies face memory bottlenecks and can be complex to use [1]. GeoTIFF [15] is a commonly used file format for distributing geospatial data and GDAL[3] is the standard library used for transforming the data.

1.2.2 Existing Solutions

To address the RAM bottleneck, a traditional solution involves performing operations in a 'blocked' fashion [16]. This technique increases the spatial locality when accessing the data, in-tern improving CPU cache locality and the RAM bottleneck. However, it is is not applicable to all operations such as random sampling, and requires manual implementation for each operation. One can also increase the size of the CPU cache [8], however this is not feasible in settings with fixed resources. Equivalently, one can reduce the size of working set. Our proposed lightweight compression technique facilitates this. Our technique has recently been applied to databases [6], web search engines [17, 18] and integer sequences [3, 12], successfully speeding up their encoding and decoding phases. However, applications to geospatial data are absent. Finally, Li *et al.* highlight a cache-inefficient operation applied to raster datasets known as a reprojection, and optimise it with a GPU. The parallel nature of their optimisation aligns with our exploitation of SIMD instructions, however they do not utilise data compression or CPUs.

1.2.3 Data Compression

Research in data compression - encoding data with fewer bits - has shifted from expensive 'heavyweight' techniques to cheaper 'lightweight' techniques that exploit the superscalar capabilities of modern CPUs [3, 6, 12, 17, 18]. Cascading such 'lightweight' techniques by feeding the output of one compressor to the input of another can improve the compression factor [12]. As cascaded compressors form new compressors, we henceforth refer to a 'codec' as any cascade of compressors, including a single compressor. Lightweight codecs decode much faster than general 'heavyweight' codecs while maintaining reasonable compression factors [3]. Out of the six studies we reviewed, four highlight that there is 'no best algorithm', and that it depends on the data characteristics [5, 6, 12, 18]. Demonstrated by Zalipynis [5], while the GeoTIFF format natively supports heavyweight techniques such as DEFLATE and LZW, the CPU required for decompression must be balanced with the memory reduction they provide. After applying GeoTIFF's native codecs to environmental data in an array database system, DEFLATE performs the best, however none of their tested algorithms provide an overall speedup due to the algorithms' large CPU demands. For a speedup, they suggest investigating lightweight codecs.

With respect to lightweight compression, Damme *et al.*'s [12] 2017 paper surveys and benchmarks lightweight codecs for compressing general integer sequences. They categorise them into operating at the 'logical' and 'physical' levels. 'Logical-level' codecs reduce the magnitude of the values, and 'physical-level' codecs minimise the bits used to store the smaller magnitude values. Logical-level codecs include DELTA (replacing each value by its difference to its predecessor), Frame-of-Reference (FOR – replacing each value by its difference to a reference value), Dictionary (DICT – replacing each value by its index in a lookup table) and Run-Length-Encoding (RLE – replacing consecutive sequences/'runs' of values with the value and the number of repetitions/'length'). Physical-level algorithms are variants of Null-Suppresion (NS – omitting redundant zeros in the values' physical representations), and are the most-studied technique. These include SIMD-BP128 and SIMD-FastPFOR by Lemire *et al.*, and SIMD-GroupSimple from Zhao *et al.* after generalising Lemire *et al.*'s techniques: both state-of-the-art algorithms from 2015 exploiting SIMD instructions. They benchmark them on the GOV2 document collection: a dataset frequently used for evaluating integer compression. However, this dataset is not entirely representative of geospatial data. In addition to benchmarking the seven physical-level algorithms, they benchmark cascades of them with all four of the logical-level techniques. From their benchmarks they conclude that SIMD-BP128 is the best choice if the data exhibits good locality, and that cascades can improve the compression factor with a compromise of speed.

In 2021, Heinzl *et al.* build on their work, further evaluating a selection of their discussed algorithms in the context of an in-memory database. They uniquely assess the random-access capabilities of the algorithms and highlight open-source implementations which may be useful to adopt. However, their results may not extend to our geospatial context. Also in 2021, Pibiri *et al.* provide a

[1] https://planetarycomputer.microsoft. com
[2] https://hadoop.apache.org
[3] https://gdal.org

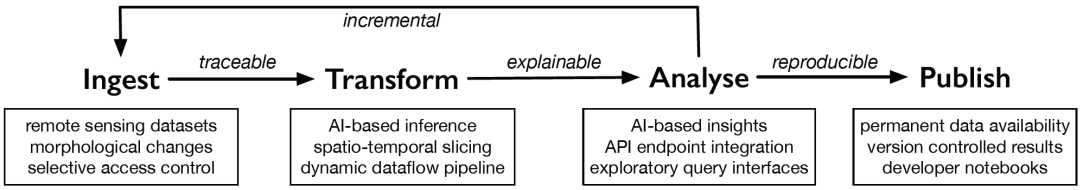


Figure 2: Holcomb *et al.*'s [1] vision of the dataflow in a planetary computing engine.

modern perspective on Damme *et al.*'s [12] 2017 study. Despite their comprehensive review and open-source implementations, they only benchmark a small subset, and exclude previously identified state-of-the-art algorithms such as SIMD-BP128. They also benchmark their algorithms on web-based datasets for optimising inverted indexes used by search engines. Again, their benchmarks may not extend to geospatial contexts. They identify the 'Roaring' and 'Slicing' algorithms as the most efficient due to their simple designs and exploitation of SIMD instructions.

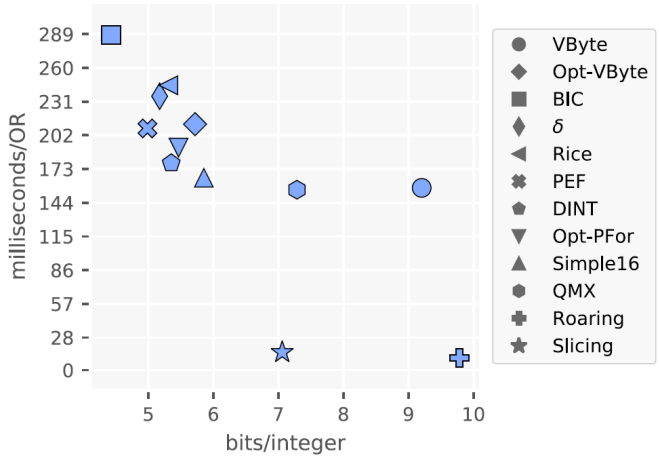


Figure 3: Pibiri *et al.*'s [18] space/time curves for the ClueWeb09 dataset.

The literature presents varying classifications for these algorithms, based on factors like data layout and alignment [17], technique level [12], and compression scope [18]. Despite these differences, common metrics like 'bits per integer' for compression factor and speed measurements in milliseconds provide a standardised framework for comparison [18]. The use of space/time plots [18], as shown in Figure 3, is particularly effective in visualising these comparisons. In such plots, the most desirable algorithms are located along the minimising Pareto front [19], as shown in Figure 4.

1.2.4 Research Gap

As highlighted in §1.2.1, Zalipynis compresses environmental data using GeoTIFF's native codecs, speeding up their pipeline by reducing IO. However, they do not apply the lightweight codecs highlighted in §1.2.2 to further

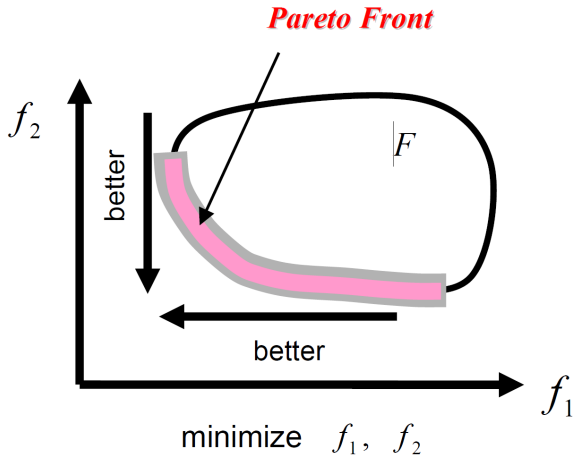


Figure 4: Ngatchou *et al.*'s visualisation of a two-objective Pareto front. When selecting the most desirable codecs, f_1 is 'space' (compression ratio), and f_2 is 'time' (compression/decompression time or overall execution time).

speed up their pipeline and suggest it as future work. Furthermore, while lightweight codecs have been applied to speed up web data processing pipelines [17, 18], they have not been applied to speed up geospatial pipelines. We address these two gaps. Our reviewed papers share the sentiment that compression performance depends the characteristics of the data being compressed [5, 6, 12, 18], thus the 'best' lightweight codecs for geospatial data is to-be-discovered in our study. Rapidly processing the increasing amounts of geospatial data from remote sensors is essential for timely insights into pressing environmental issues like climate change and biodiversity loss. Our advancements in data processing directly contribute to swift decision-making in areas such as deforestation tracking, habitat preservation, and climate monitoring [1].

1.3 Principal Findings

The key findings from our study are as follows:

1. We overcome the RAM bottleneck for random sampling operations with our method, providing relative speedups of up to 29%.

2. After fusing summing into decompression, we further overcome the RAM bottleneck, obtaining relative speedups of up to 78.7% for sums without a

transformation, and 49.7% for sums after a transformation.

3. Lemire's `SIMDBinaryPacking+VariableByte` codec provides the largest best-case speedup for pipelines without transformations (33.7%). For pipelines with transformations, Lemire's `simdcomp` provides the largest best-case speedups (17%).

4. We project the best codecs to fuse summing into decompression in the future are vectorised DELTA and TurboPFor's `xor+TurboPack256` codec.

2 Methods

In this section, we describe our geospatial pipelines and explain our application of compression to reduce their execution time.

2.1 Control Pipelines

Figure 5a shows the data flow in our unoptimised pipelines which we optimise in our study. In our experiments, these pipelines are our control. Our pipelines have a MapReduce structure: a common data processing framework in geospatial pipelines. Our pipelines first transform the data, akin to the 'map' part of MapReduce, and then aggregate the data, akin to the 'reduce' part of MapReduce.

Our pipelines process 2D rasters of `uint32` values, split into evenly-sized blocks. We perform the following transformations:

- **None**: No transformation - leave the data unchanged.
- **Shift**: Linearly sweep each value, and add a large value (2^{23}) to it. This transformation simply increases each value by 2^{23}. Increasing each value by a constant is commonly used to increase brightness or performed within more complex transformations.
- **Threshold**: Linearly sweep each value, and replace it by 0 if it is smaller than the mean of the values, and 1 otherwise. This transformation thresholds the data by the mean (which we assume is known a priori) and is commonly used for filtering.
- **Classification**: We form 8 equally-sized buckets spanning the range between the minimum and maximum value (which are assume are known a priori). Then, linearly sweep each value, and replace it by the index of the bucket it falls into. This transformation classifies each value by the bucket it falls into, and is commonly performed to compute histograms.
- **Smooth**: Linearly sweep each `uint32` value, and replace it by the mean of the window of the nearest 3 values (including itself). This transformation smooths the data and is a commonly used for blurring.

and the following aggregations:

- **Sum**: linearly traverse each `uint32` value, adding them all to a running `uint64` sum. Sums are common aggregations in geospatial pipelines, and can be used to compute other statistics such as means.
- **Random Sample**: read every `uint32` value in a random order. Random sampling is another useful aggregation in geospatial pipelines, performed within Monte Carlo methods.

We measure the execution time of our pipeline on a per-block basis. Assuming a pre-processing overhead, the per-block execution time, $\mu_T^{\text{exec}} \pm \sigma_T^{\text{exec}}$, for our control pipeline is given by:

$$\mu_T^{\text{exec}} = \mu_T^{\text{trans}} + \mu_T^{\text{agg}} \tag{1}$$

$$\sigma_T^{2\,\text{exec}} = \sigma_T^{2\,\text{trans}} + \sigma_T^{2\,\text{agg}} \tag{2}$$

where $\mu_T^{\text{trans}} \pm \sigma_T^{\text{trans}}$ is the time taken to transform a pre-processed block and $\mu_T^{\text{agg}} \pm \sigma_T^{\text{agg}}$ is the time taken to aggregate such a transformed block. In the case the pipeline performs no transformation, $\mu_T^{\text{trans}} = \sigma_T^{2\,\text{trans}} = 0$, and the execution time becomes just the aggregation time.

2.2 Test Data

We process intermediate geospatial data in GeoTIFF format from pipelines within the Cambridge Centre for Carbon Credits (4C)[4]. Such data is sourced from remote (satellite) sensors and NASA's Landsat datasets[5]. **Figure 6** shows one such file processed by our pipelines. **Appendix A** shows all four files (A, B, C and D). **Table 1** lists essential properties of each file. Despite the data in file A being floating point, we truncate its mantissa without data loss. If the file contains negative values, we add the absolute value of the minimum to each value to scale them to all-positive values. After doing so, every value in every file comfortably fits in a `uint32`.

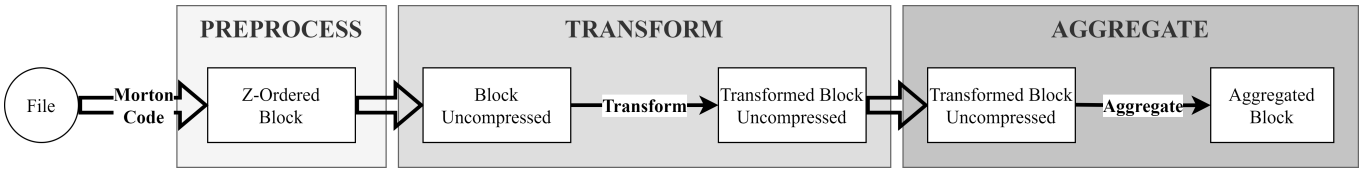
File	Data Type	Dimensions	Min	Max
A	float32	43200 x 17400	-9999.0	138893.0
B	uint8	37107 x 37368	0	6
C	int16	6000 x 6000	-9999	90
D	int16	6000 x 6000	-32768	2355

Table 1: Key attributes of our test files.

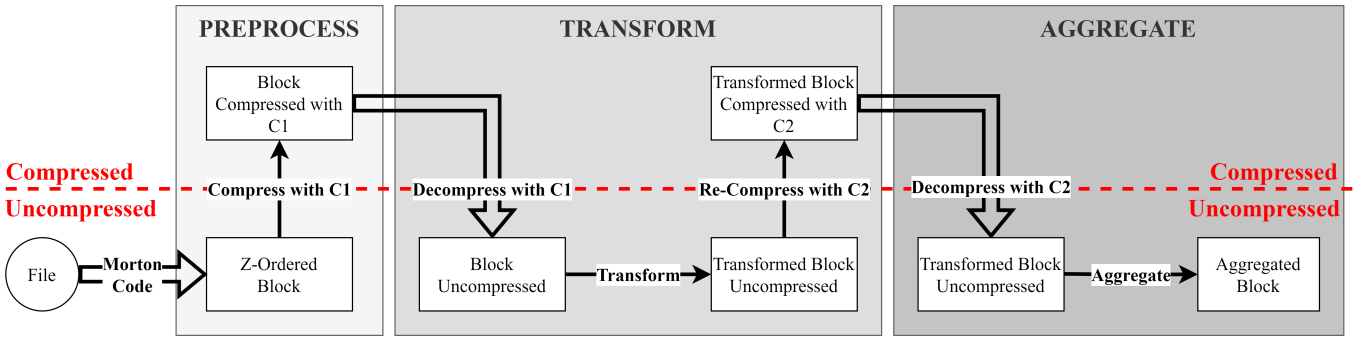
2.3 Pipeline Optimisation

With our control pipelines in place, we describe our methods for optimising them. Our method relies on keeping the blocks compressed in memory. By doing so, we keep the data higher in the memory hierarchy and reduce the

[4] https://4c.cst.cam.ac.uk
[5] https://landsat.gsfc.nasa.gov



(a) Unoptimised/control pipeline. This pipeline operates on blocks directly in their uncompressed form. Since Morton-coding blocks is part of the pre-processing step, we operate on a per-block basis.



(b) Optimised/experimental pipeline. This pipeline keeps blocks compressed in memory. Above the red line is a compressed grid of blocks. Below is a single uncompressed block. Only one block is uncompressed at a time to minimise memory usage. C1 and C2 are two separate (potentially composite) codecs.

Figure 5: Data flow in our unoptimised (a) and optimised (b) geospatial pipelines. Single-lined arrows represent operations on individual blocks. Double-lined arrows repeat the subsequent per-block operations over every block in-tern.

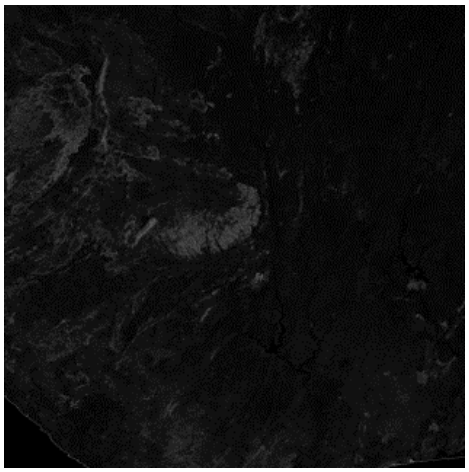


Figure 6: File C (brightened and down-scaled).

worst-case impact of cache misses. In our setting, we have too much data for the CPU caches. Hence, a lot of data resides in main memory. We aim to move more data into the CPU caches by compressing it and only decompressing single blocks at a time.

Figure 5b shows the data flow in our optimised pipelines. We first pre-process the file by splitting it into blocks, Morton coding each block, and compressing each block with C1, as depicted in Figure 7. After pre-processing, we have a grid of Z-ordered blocks each compressed with C1. As compression reduces the size of the data, this compressed grid occupies less memory than the uncompressed grid after the pre-processing step of the unoptimised pipeline (Figure 5a).

In our pipelines, we first Morton code the blocks in the pre-processing step to improve the compression ratio when compressing blocks. This works because computer memory is 1D. When implementing our pipelines, we map our 2D blocks to 1D contiguous memory locations. A naive approach for implementing this mapping is traversing the block in a row-major fashion. While this method has good spatial horizontal spatial locality, it has poor vertical spatial locality and large discontinuities between rows. Morton coding improves the spatial locality on both axes and reduces the worst-case discontinuities by recursively 'Z-ordering' the data, improving the compression ratio of our compressors as they exploit similarities in the data.

After pre-processing the data, we transform it on a per-block basis. For each block, we decompress it with C1, transform the uncompressed block, and re-compress the transformed block with C2. Note that we change the codec used to re-compress the data. This is because the performance of a codec depends on the characteristics of the data it is compressing. The old codec used to compress the untransformed data, C1, may not be the best after transforming it.

After transforming and re-compressing each block, we

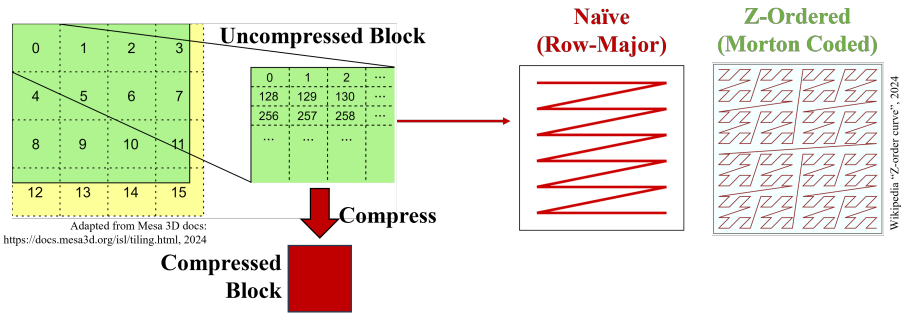


Figure 7: Raster (yellow) split into blocks (green) and compressed (red). Each block is Morton coded (right) and we show the naive row-major ordering before for comparison.

similarly aggregate the transformed blocks. For each transformed block, we decompress it with C2 and aggregate the uncompressed transformed block. Since aggregations do not change the contents of the blocks, we do not need to re-compress them after aggregating them. Note that we decompress the block with C2, since the transformed block was previously compressed with C2.

The per-block execution time, $\mu_T^{\text{exec}} \pm \sigma_T^{\text{exec}}$, for our optimised pipeline is the same as the per-block execution time for our unoptimised pipeline (Equations 1 and 2) but with the added overhead of compression and decompression. It is given by:

$$\mu_T^{\text{exec}} = \mu_T^{\text{D1}} + \mu_T^{\text{trans}} + \mu_T^{\text{C2}} + \mu_T^{\text{D2}} + \mu_T^{\text{agg}} \tag{3}$$

$$\sigma_T^{2\,\text{exec}} = \sigma_T^{2\,\text{D1}} + \sigma_T^{2\,\text{trans}} + \sigma_T^{2\,\text{C2}} + \sigma_T^{2\,\text{D2}} + \sigma_T^{2\,\text{agg}} \tag{4}$$

where $\mu_T^{\text{D1}} \pm \sigma_T^{\text{D1}}$ is the time taken to decompress a pre-processed block, $\mu_T^{\text{trans}} \pm \sigma_T^{\text{trans}}$ is the time taken to transform a pre-processed block, $\mu_T^{\text{C2}} \pm \sigma_T^{\text{C2}}$ is the time taken to re-compress such a transformed block, $\mu_T^{\text{D2}} \pm \sigma_T^{\text{D2}}$ is the time taken to decompress such a transformed block, and $\mu_T^{\text{agg}} \pm \sigma_T^{\text{agg}}$ is the time taken to aggregate such a transformed block. In the case of a 'None' transformation, we immediately aggregate the compressed pre-processed data, thus $\mu_T^{\text{C1}} = \mu_T^{\text{trans}} = \mu_T^{\text{C2}} = 0$.

In our optimised pipelines, μ_T^{trans} and μ_T^{agg} are smaller than in the unoptimised pipelines, since the uncompressed block data is higher in the memory hierarchy, with higher bandwidths and lower memory read/write overheads. Despite the additional overhead of compression/decompression in our optimised pipelines, we achieve an overall speedup when the time saved during transformation and aggregation dominates the compression/decompression overheads. Thus, for a speedup, we require codecs that are **fast** to minimise $\mu_T^{\text{D1}} + \mu_T^{\text{C2}} + \mu_T^{\text{D2}}$, but also **compress well** to minimise $\mu_T^{\text{trans}} + \mu_T^{\text{agg}}$.

2.4 Codec Selection

The best choice of codecs to use within a pipeline depends on the characteristics of the data flowing through it. While one can use heuristics to select their codecs (for example, using dictionary coding if the number of unique values is below a threshold), crafting such heuristics to consider both the space and time axes is a complex task likely to result in overlooking good codecs. To avoid this, we exploit our computational resources to comprehensively benchmark all our implemented codecs. From these benchmarks, we select the codecs to use in our pipelines that provide the greatest speedups.

After implementing our codecs, we benchmark them on every (file,transformation) combination. In these benchmarks, we split the file into 10,000 equally-spaced 256x256 blocks of `uint32` values. For each block, we load it and preprocess it by Morton-coding and transforming the data. Then, we individually measure the time taken to compress the block t_C, the time to decompress the compressed block t_D, and the number of bits occupied by the block's compressed form b_C. From b_C we compute the compression ratio $r_C = b_C/(256 * 256 * 32)$. We average these statistics over every block to obtain the mean and variance for the compression time $\mu_C \pm \sigma_C$, decompression time $\mu_D \pm \sigma_D$ and compression ratio $\mu_r \pm \sigma_r$.

Then, we use these benchmarks to select the best codecs for each optimised pipeline, C1 and C2. For a specific (file,transformation,aggregation) combination, consider the optimised pipeline performing that transformation and aggregation. From Equation 3, C1's contribution to the execution time is $\mu_T^{\text{D1}} + \mu_T^{\text{trans}} \propto \mu_D + \mu_r$. While $\mu_T^{\text{D1}} \propto \mu_D$ directly follows from our benchmarks, $\mu_T^{\text{trans}} \propto \mu_r$ follows from the fact that if a codec produces smaller compression ratios, then the compressed grid with C1 occupies less memory, reducing the memory read/write bottlenecks when performing the transformation and in-tern reducing its execution time. Thus, to minimise the execution time, we select C1 as the codec which minimises the decompression time μ_D and compression ratio μ_r. Since compression algorithms trade-off space and time, there is rarely a codec which minimises both μ_D and μ_r simultaneously. Thus, we estimate the best candidates for C1 as the codecs lying on the minimising Pareto front of μ_D against μ_r' – the 'Pareto codecs'.

Similarly, C2's contribution to the execution time is $\mu_T^{\text{C2}} + \mu_T^{\text{D2}} + \mu_T^{\text{agg}} \propto \mu_C + \mu_D + \mu_r$. Hence, the Pareto codecs for C2 are those lying on the minimising Pareto front of $\mu_C + \mu_D$ against μ_r. Given our individual sets of Pareto codecs for C1 and C2, we refer to the specific combination of C1 and C2 as a 'dual codec'. For C1 $\in \phi$ and C2 $\in \psi$, our 'dual Pareto codecs' $\in \phi \times \psi$.

Having used our compression benchmarks to obtain our set of dual Pareto codecs for our (file,transformation,aggregation) pipeline, we search over every dual Pareto codec to find the specific optimal combination of C1 and C2, denoted by C1* and C2*, which minimises the execution time of the pipeline:

$$(\text{C1}^*, \text{C2}^*) = \underset{(\text{C1},\text{C2})\in\phi\times\psi}{\operatorname{argmin}} \; \mu_T^{\text{exec}} \tag{5}$$

In the case of pipeliens with no transformation, our pipeline uses a single codec C2, and $\mu_T^{\text{exec}} = \mu_T^{\text{D2}} + \mu_T^{\text{agg}} \propto \mu_D + \mu_r$. Therefore, our Pareto codecs ψ for pipelines without transformations is the minimising Pareto front of μ_C against μ_r. Thus, the optimal Pareto codec C* is obtained by searching over the single Pareto codecs:

$$\text{C}^* = \underset{\text{C}\in\psi}{\operatorname{argmin}} \, \mu_T^{\text{exec}} \tag{6}$$

Indeed, one can also obtain the optimal codecs by searching over *all* combinations of codecs, skipping our Pareto benchmarks. However, since we implement a large number of codecs, this search is intractable with our computational resources. This intractable computation is clear from the fact that the number of dual codecs grows quadratically with the number of codecs. Hence, for tractability we narrow the search to our Pareto codecs.

Given the best codecs for each optimised pipeline, we compare our best pipelines' execution times with their unoptimised baselines to assess the speedups provided by our optimisations.

2.5 Fusing Aggregation into Decompression

Our pipelines in Figure 5 show aggregation as a separate step after decompression. However, Damme *et al.* [12] show that sum aggregations can be fused into codecs' decompression stage to compute the sum faster. For SIMD-optimised codecs, they replace the vectorised stores at the end of decompression to vectorised sums. This method exploits the fact that the decompressed data is available in vector registers at the end of decompression, gaining the speedup from vectorised adds without the overhead of initially loading the data into vector registers. For RLE, they sum the products of each (value,length) pair. Both of these methods benefit from improved cache-locality as we piggy-back a pass over the data to aggregate it. We apply these optimisations to our codecs, as explained in §2.6.1.

After fusing aggregation into decompression, we make $\mu_T^{\text{agg}} \approx 0$ and increase μ_T^{D2} in Equation 3. Due to the vectorisation and cache-locality benefits of fusing aggregations, the reduction in μ_T^{agg} dominates the increase in μ_T^{D2}, providing a net reduction to the pipeline execution time μ_T^{exec}.

2.6 Implementation

Following our description of our pipelines and optimisation methods, we describe their implementation.

2.6.1 Codecs

At the core of our implementation is our library of codecs, implemented in C++ with AVX512 and SSE4.2 vectorisation intrinsics. We abstract each codec as a `StatefulIntegerCodec`, which facilitates code-reuse for codecs which compress to different internal representations. A simplified definition of the `StatefulIntegerCodec` class is given below:

```
class StatefulIntegerCodec {
public:
  virtual void encodeArray(const uint32_t *in,
      ↪ const size_t length) = 0;
  virtual void decodeArray(uint32_t *out, const
      ↪ size_t length) = 0;
}
```

An example of using a `StatefulIntegerCodec` to compress and decompress an array of integers is given below:

```
StatefulIntegerCodec codec = /* ... */;
uint32_t data[10] = /* ... */;
// Compress 'data'.
codec.encodeArray(data, 10);
// Now, 'codec' stores the compressed data.
// Decompress 'data'.
uint32_t data_back[10];
codec.decodeArray(data_back, 10);
// Now, 'data_back' stores the decompressed data
    ↪ .
```

Note that the above example works regardless of the internal representation that the codec compresses to (these can vary in terms of bit-width, additional metadata, etc.), mitigating the need to adapt user-code to each specific codec.

Table 2 lists our custom-implemented logical-level codecs, vectorised in the same way as Damme *et al.* [12]. **Figure 8** lists our open-sourced physical-level codecs.

Codec	Description
custom_delta_vecavx512	DELTA vectorised with AVX512.
custom_for_vecavx512	FOR vectorised with AVX512.
custom_rle_vecavx512	RLE vectorised with AVX512.
custom_dict_vecavx512	DICT vectorised with AVX512.

Table 2: Custom logical-level codecs. Note we also have unvectorised and SSE4.2 implementations (suffixed with `unvec` and `vecsse` respectively) but exclude them as the AVX512 implementations dominate their performance.

Codec Name/Prefix	Number of Codecs	Fast?	Author	Source
FastPFor_	~30	Y	Lemire et al.	https://github.com /lemire/FastPFOR
TurboPFor_	~20	Y	Michael Stapelberg	https://github.com /powturbo /TurboPFor-Integer-Compression
simdcomp	1	Y	Lemire et al.	https://github.com /lemire/simdcomp
MaskedVByte	2	Y	Lemire et al.	https://github.com /lemire/MaskedVByte
streamvbyte	1	Y	Lemire et al.	https://github.com /lemire/streamvbyte
FrameOfReference _Turbo	1	Y	Lemire et al.	https://github.com /lemire/FrameOfReference
LZ4	1	N	Yann Collet	https://github.com/lz4/lz4
DEFLATE	1	N	Jean-loup Gailly and Mark Adler	https://www.zlib.net
Zstd_	3	N	Yann Collet	https://github.com /facebook/zstd

Figure 8: Open-sourced physical-level codecs. We include slow heavyweight codecs for completeness. Our `Zstd_` codecs have 3 compression levels: 1, 3 and 5.

Since logical and physical-level codecs can be cascaded to further compress the data, we also implement a generic cascade, which takes two `StatefulIntegerCodecs` and produces a new `StatefulIntegerCodec` which cascades the two codecs internally. A simplified definition of the cascade is given below:

```
class CompositeStatefulIntegerCodec : public
    ↪ StatefulIntegerCodec {
private:
    std::unique_ptr<StatefulIntegerCodec>
        ↪ firstCodec;
    std::unique_ptr<StatefulIntegerCodec>
        ↪ secondCodec;
public:
    CompositeStatefulIntegerCodec(
        std::unique_ptr<StatefulIntegerCodec>
            ↪ first,
        std::unique_ptr<StatefulIntegerCodec>
            ↪ second)
        : firstCodec(std::move(first)),
            ↪ secondCodec(std::move(second)) {}
    /* Methods from StatefulIntegerCodec... */
}
```

where its `encodeArray` and `decodeArray` sequence the methods from `firstCodec` and `secondCodec`. We name such cascades `[+]_codec1+codec2` where `codec1` and `codec2` are the names of the first and second codecs.

In total, including composite and non-composite codecs, we have **294 codecs**.

To fuse summing into decompression, we apply the method described in §2.5 to the `custom_rle_vecavx512`, `FastPFor_SIMDPFor+VariableByte` and `simdcomp` codecs. We select these codecs due to their high projected speedups in §3.3.2 and implementation convenience. Instead of *replacing* vectorised stores with sums, we add a vectorised sum alongside each store. This is because the decoding phase of `FastPFor_SIMDPFor+VariableByte` has exceptions [3] requiring accessing previously decompressed data, so we store the decompressed data alongside the running sums. We maintain a separate sum for these exceptions, and subtract this sum from the overall sum to correct the sum by the exceptions.

Since our sum can overflow 32 bits, we maintain the lower 32 bits and upper 32 bits of the sum separately. The last stage of decompression writes the vectorised output data in `__m128i OutReg` to `uint32_t *_out` like so:

```
__m128i *out = (__m128i *)(_out);
_mm_storeu_si128(out++, OutReg);
```

To aggregate the stored data, we maintain running lower and upper sums in `__m128i* sum_lo` and `__m128i* sum_hi` and append the below line after each `storeu` instruction:

```
agg_sums(OutReg, sum_lo, sum_hi);
```

where `agg_sums` has the following implementation:

```
static void agg_sums(__m128i OutReg, __m128i*
    ↪ sum_lo, __m128i* sum_hi) {
    __m128i OutReg_lo = _mm_unpacklo_epi32(
        ↪ OutReg, _mm_setzero_si128());
    __m128i OutReg_hi = _mm_unpackhi_epi32(
        ↪ OutReg, _mm_setzero_si128());
    *sum_lo = _mm_add_epi64(*sum_lo, OutReg_lo);
    *sum_hi = _mm_add_epi64(*sum_hi, OutReg_hi);
}
```

After aggregating the sums in `__m128i* sum_lo` and `__m128i* sum_hi`, we compute the `uint64 sum` as follows:

```
uint64_t sum =
    _mm_extract_epi64(sum_lo, 0) +
        ↪ _mm_extract_epi64(sum_lo, 1) +
    _mm_extract_epi64(sum_hi, 0) +
        ↪ _mm_extract_epi64(sum_hi, 1);
```

2.6.2 Programs

We implement the following programs in C++ for our experiments:

- `test_comp`: A program used to test the validity of our codecs. We randomly generate arrays of integers between 0 and 2^{24} and verify that each codec can successfully recover the array after compressing and subsequently decompressing it. We cap the values at 2^{24} because some codecs cannot compress larger integers (for example, `FastPFor_Simple16` requires the values $\in [0, 2^{28})$).

- `bench_comp`: A program used to benchmark the compression time, decompression time and compression ratio of codecs on a GeoTIFF file with a given transformation. The program evenly samples 10,000 blocks across the file. For each block, it Morton codes the block, transforms it, compresses it, and decompresses it. We average the compression time, decompression time and compression ratio across each block to obtain $\mu_D \pm \sigma_D$, $\mu_r \pm \sigma_r$ and $\mu_C \pm \sigma_C$ respectively.

- `bench_pipeline`: A program used to benchmark a portion of an optimised pipeline. For a particular combination of codecs `C1, C2` and a particular combination of `(file,initialTransformation,transformation)` or `(file,initialTransformation,aggregation)`, the program forms a grid of 2000 evenly-spaced blocks across the file. We pre-process each block by Morton coding it, transforming it with `initialTransformation`, and compressing it with `C1`. We store each compressed block in a 'compressed grid'. Then, we sample each block in this grid 10 times, measuring for each block the time to decompress it with `C1`, the time to either perform the `transformation` or the `aggregation`, and the time to re-compress it with `C2`. We combine the averaged timings from this program to measure μ_T^{exec} and $\sigma_T^{2\,\text{exec}}$. Note that one can run the program with the 'custom_direct_access' to obtain baseline measurements for our unoptimised pipelines.

We compile each program with `gcc` version 11.4.0 and with flags `-O3 -msse4.1 -mavx512f mbmi2 -fopenmp`. We run our programs on a Linux server and locally in WSL. Additionally, we use the Libmorton library [20] for Morton codes, and nanosecond-precision wallclock times for time measurements via `std::chrono::system_clock`.

2.6.3 High Performance Computing

We run `bench_comp` and `bench_pipeline` in parallel over all codecs and pipelines on the Cambridge Service for Data-Driven Discovery (CSD3) High-Performance-Computer (HPC) cluster within SLURM jobs. We use machines on the `cclake-himem` partition with Intel Xeon processors and the following CPU cache sizes:

- L1d: 32KB,
- L2: 1MB,
- L3: 39MB,

totalling ∼40MB. Given our uncompressed grid is $2000 * 256 * 256 * 4/10^6 = 524$MB, without compression much of the data spills out of the CPU caches, enabling a speedup with compression.

2.7 Analysis

We perform data analysis in Python by reading the `.out` files from our SLURM jobs, parsing the data into a CSV, and analysing the CSV data using the `numpy`, `pandas` and `matplotlib` libraries.

3 Results

3.1 Space/Time Pareto Fronts

As explained in §2.4, we benchmark our codecs with respect to their compression ratio $\mu_r \pm \sigma_r$, decompression time $\mu_D \pm \sigma_D$, and decompression + recompression time $(\mu_D + \mu_C) \pm \sqrt{\sigma_D^2 + \sigma_C^2}$ on every `(file,transformation)` combination. We obtain such timings by running `bench_comp`. Then, we plot the space/time graphs for each `(file,transformation)` combination. The minimising Pareto fronts of these graphs are the best sets of codecs for our optimised pipelines.

We have 4 tiffs, 5 transformations (including none) and 294 codecs. Thus, we have 5,880 `(tiff,transformation,codec)` combinations. Due to the large number of combinations, we perform our benchmarks in parallel using our HPC.

Figure 9 shows our space/time plots and Pareto fronts for file A transformed with Classification. **Appendix B** shows all of our plots and Pareto fronts for each (tiff,transformation) pair.

From our space/time plots, we observe a large variance in compression ratio and wallclock time across our codecs. This observation is reinforced by the axes being logarithmic. This highlights the importance of selecting the correct codec, as their wallclock times and compression ratios vary significantly.

Furthermore, on average, the Pareto fronts contain 35x fewer codecs than the total collection. This highlights

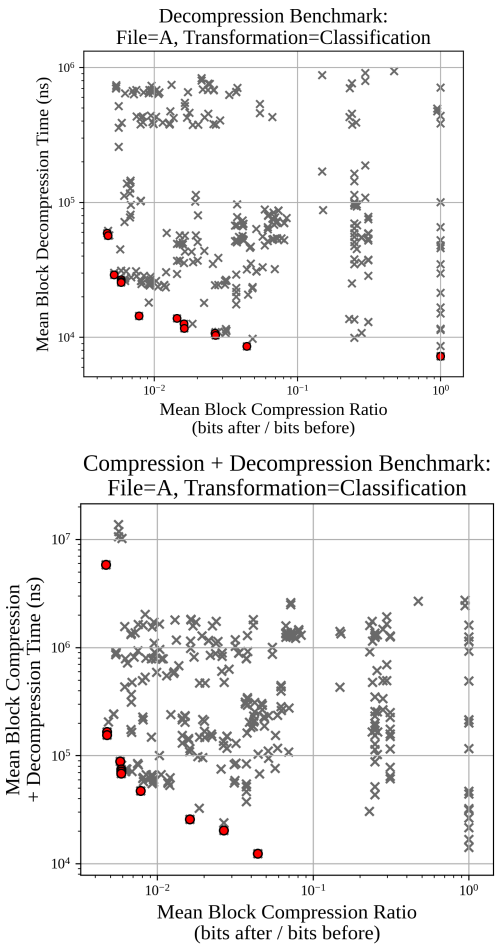


Figure 9: Space/time plots for file A transformed with Classification. Grey crosses are the benchmarks for each codec. Red circles are the codecs on the Pareto front. The names of the codecs on the Pareto front are detailed in Appendix B.

the significant reduction in search space for our subsequent pipeline optimisation experiments and the benefit of our Pareto optimisation method.

3.2 Pipeline Speedups

As explained in §2.3, to optimise our pipelines we keep the data compressed and decompress individual blocks when transforming or aggregating it. We use only the codecs on the Pareto fronts obtained in §3.1 to compress the data. In this section, we highlight our best speedups after searching over the codecs on the Pareto fronts to optimise our pipelines. For brevity, we use the term 'speedup' to refer to relative speedups, computed as $100 * (t_0 - t_1)/t_0$ where t_0 is the original time/duration and t_1 is the new time/duration.

First, we highlight the speedups for pipelines when performing random sampling aggregations. Then, we highlight the speedups for pipelines performing summing aggregations. For the speedups of summing pipelines, we also highlight the speedup with summing fused into the decompression stage of the pre-aggregation codec, as explained in §2.5.

We provide the full listing of the best codec and speedup for each `(file,transformation,aggregation)` combination in **Appendix C**. There, we also explain our computation of the mean and variance of speedups.

3.2.1 Random Sampling

Figure 10 shows the average speedup of our optimised pipelines performing random sampling aggregations against their non-optimised versions, averaged over all files. The reported mean speedups are such that, for each `(file,transformation)` combination, we search over its Pareto codecs to find those minimising the execution time of the pipeline. Then, we convert this minimal execution time to a best-found speedup relative to the non-optimised pipeline. We average the best-found speedups over each file to obtain a mean speedup for the transformation. Here, the error bars are deviations σ such that $\sigma^2 = \sigma_F^2 + \sigma_B^2$. This variance includes both the variance of the speedups between the files, σ_F^2, and the variance of the speedups between the blocks in each file, σ_B^2.

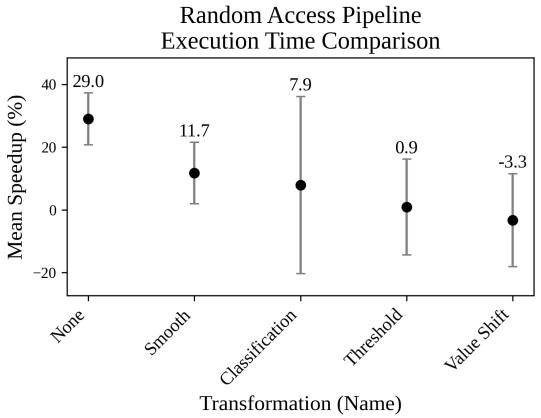


Figure 10: Mean speedup $\pm\sigma$, optimised vs non-optimised, of pipelines randomly sampling blocks, with and without a preceding transformation. The speedups are averaged over every file. For every `(file,transformation)` combination, we use our best-found execution time for the pipeline, obtained with our best-performing codec.

From Figure 10, we observe a 29% average speedup for pipelines randomly sampling data and performing no preceding transformation, and an 11.7% average speedup with our best-performing preceding transformation (Smooth). Most preceding transformations provide a net speedup. The only preceding transformation not providing a net speedup is Shift. Thus, we overcome the main memory bottleneck for random sampling operations with our method, providing speedups of up to

29%. While most of the errors are acceptable for comparison via means, the error for Classification is unacceptably large. As such, our best pipelines have volatile performance when randomly sampling Classification-transformed data, and the result for Classification should be ignored.

3.2.2 Summing

Figure 11 shows the average speedup of our optimised pipelines performing summing aggregations against their non-optimised versions, averaged over all files. We report the means and deviations in the same way as for the Random Sampling speedups. We show the speedups before and after fusing summing into decompression.

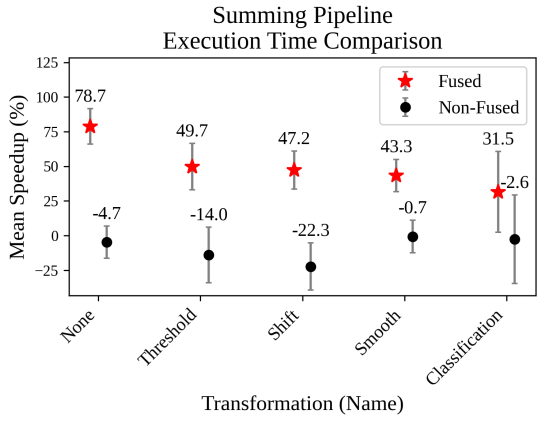


Figure 11: Mean speedup $\pm\sigma$, optimised vs non-optimised, of pipelines summing blocks, with and without a preceding transformation. The speedups are averaged over every file. Black and red points are before and after fusing summing into decompression respectively.

From Figure 11, we observe the speedups for summing blocks are much larger after fusing summing into decompression. After fusing summing into decompression, we have an overall speedup of 78.7% without a transformation and 49.7% with our best-performing transformation (Threshold). However, without fusing summing into decompression, we observe no overall speedup without or with any transformation. Thus, fusing summing into decompression is necessary to obtain speedups with our method, and doing so produces the largest observed speedups in our experiments.

Obtaining no speedup without fusing summing into decompression is not surprising, since our implementation of summing blocks is cache-efficient. Comparatively, we observe net speedups with random accesses since the access pattern is cache-inefficient, increasing the benefit of keeping the data compressed in memory. Therefore, the cache-efficiency of the aggregation is an important factor determining the possible speedups from our method, with less cache-efficient aggregations producing larger speedups.

While most errors are acceptably small for comparison via means, the error for Classification is unacceptably large, replicating the volatility that we observed with random sampling. Again, the result for Classification should be ignored.

3.3 Codec Rankings for Pipeline Speedups

In this section, we rank our codecs based on their provided speedups to our pipelines. These results serve to inform users of our method and inform future optimisation work, enabling these optimisations to be focused on the best-performing codecs.

We rank the codecs separately for pipelines without a transformation and with a single transformation. For pipelines without a transformation, we report single codecs. For pipelines with a single transformation, we report dual codecs, in the form `codec1` → `codec2`.

We rank codecs with the following metrics, where a larger rank implies a better codec:

- **Normalised ranking**: for each codec, we sum the mean speedups it provides to every pipeline it was benchmarked on. After computing the total speedup for each codec, we normalise their total speedups between 0 and 1 to obtain their normalised rankings. This metric captures both the performance and versatility of the codecs, ensuring higher-ranked codecs provide relatively large speedups across multiple pipelines.

- **Max speedup rankings**: for each codec, we compute its maximum mean speedup across every pipeline it was benchmarked on, and rank the codec with the mean and variance of this particular speedup. This metric captures the best-tested performance of the codecs across the pipelines, ensuring higher-ranked codecs can provide the largest speedups - under narrow conditions.

3.3.1 Rankings Without Fusing Aggregation

Figure 12 and Figure 13 show our normalised and max speedup rankings of single codecs applied to pipelines without transformations. We compute these speedups by searching over each `(file,aggregation)` pipeline performing no transformation. For each Pareto codec, we measure its speedup relative to the non-optimised pipeline. We aggregate the speedups provided by these codecs to rank them.

Figure 14 and Figure 15 show our normalised and max speedup rankings of dual codecs applied to pipelines with single transformations. We compute these rankings in a similar way to the pipelines without transformations. However, instead of aggregating the speedups of every Pareto codec, we only consider a speedup if it is the best speedup over every codec other applied to the pipeline. This ensures only the best codecs have non-zero ranks,

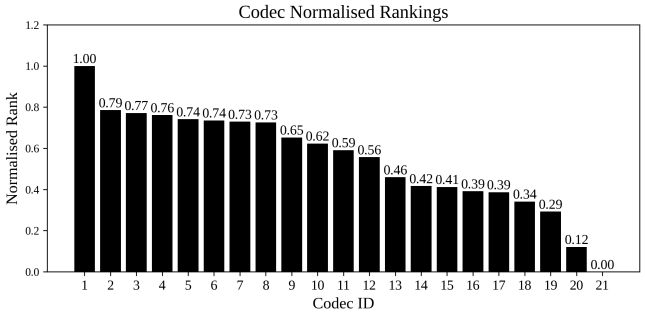


Figure 12: Normalised rankings of codecs for pipelines with no transformation. For these codecs, aggregation is not fused into decompression. The names corresponding to each Codec ID are given in Appendix D.1, Table 5.

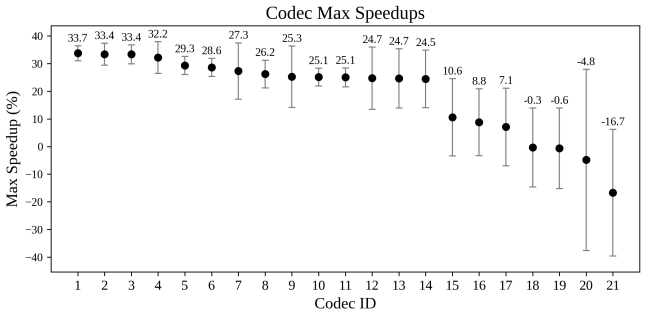


Figure 13: Max speedups of codecs for pipelines with no transformation. For these codecs, aggregation is not fused into decompression. The names corresponding to each Codec ID are given in Appendix D.1, Table 6.

narrowing the ranking to avoid overwhelming our visualisation of the rankings.

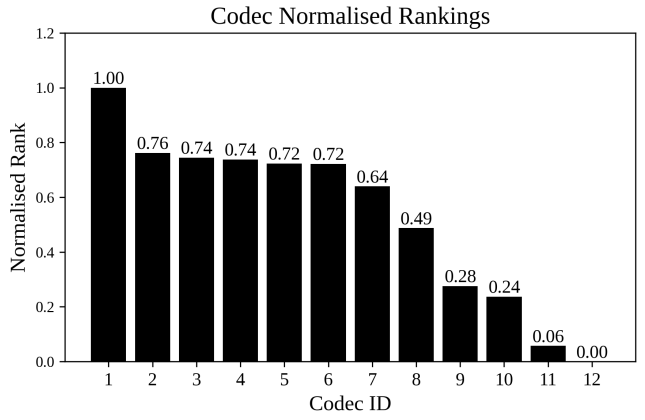


Figure 14: Normalised rankings of codecs for pipelines with a single transformation. For these codecs, aggregation is not fused into decompression. The names corresponding to each Codec ID are given in Appendix D.1, Table 7.

Given these rankings, for pipelines

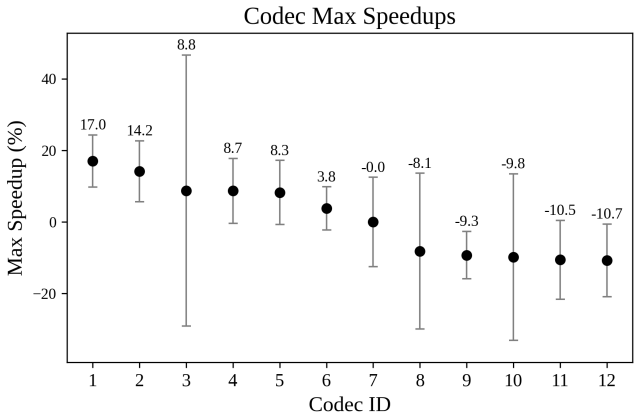


Figure 15: Max speedups of codecs for pipelines with a single transformation. For these codecs, aggregation is not fused into decompression. The names corresponding to each Codec ID are given in Appendix D.1, Table 8.

without transformations, we observe the `FastPFor_SIMDBinaryPacking+VariableByte` codec is ranked the highest and also provides the largest speedup (up to 33.7%). For pipelines with a single transformation, `simdcomp→simdcomp` is ranked the highest and also provides the largest speedup (up to 17%). The matching highest normalised and max speedup rankings for our two pipeline classes reinforces our assessment of the highest-ranking codecs.

The metrics do not agree for every rank. For pipelines with no transformations, the normalised and max speedup rankings disagree for the second-place codec. For pipelines with a single transformation, they disagree for the third-place codec. These disagreements highlight the nuanced performance differences between the codecs, necessitating caution when ranking them.

3.3.2 Projected Rankings after Fusing Aggregation

Having ranked the codecs' performance without fusing aggregation into decompression, we project the codec rankings after fusing these steps for summing pipelines. To make these projections, we remove the time to aggregate the data from the pipeline execution times, as aggregation is assumed to occur during decompression with negligible overhead (§2.5). We still take speedups relative to the non-optimised pipelines, hence aggregation times remain included in the non-optimised execution times. These projections inform the codecs for which we fuse aggregation into decompression.

Using our modified execution times, we rank the single and dual codecs in the same way as in §3.3.1, however only considering summing pipelines. **Figure 16** and **Figure 17** show our normalised and max speedup rankings of single, fused codecs applied to pipelines without transformations. **Figure 18** and **Figure 19** show our normalised and max speedup rankings of dual, fused codecs applied

to pipelines with single transformations.

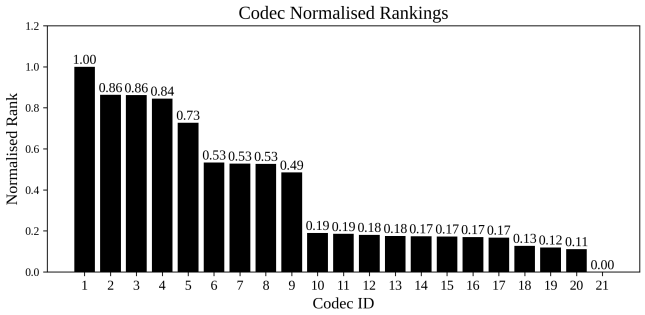


Figure 16: Normalised rankings of codecs for summing pipelines with no transformation. For these codecs, aggregation is not fused into decompression, however the execution time excludes aggregation time to project the rankings after fusing aggregation into decompression. The names corresponding to each Codec ID are given in Appendix D.2, Table 9.

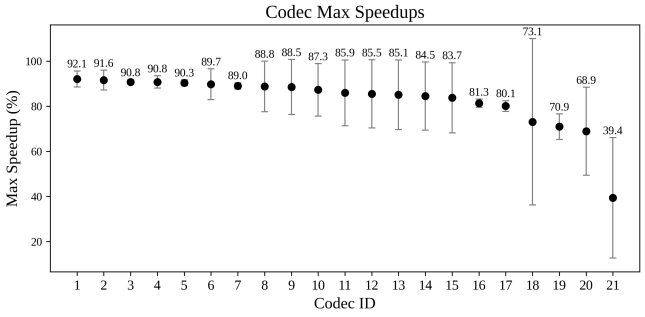


Figure 17: Max speedups of codecs for summing pipelines with no transformation. For these codecs, aggregation is not fused into decompression, however the execution time excludes aggregation time to project the max speedups after fusing aggregation into decompression. The names corresponding to each Codec ID are given in Appendix D.2, Table 10.

Given these rankings, we project `custom_delta_vecavx512` as the single codec with the highest normalised rank, and `TurboPFor_xor+TurboPack256` as the single codec providing the largest maximum speedup (92.1%) after fusing summing into decompression.

For dual codecs, we project `simdcomp`→`simdcomp` as the dual codec with the highest normalised rank, and `FastPFor_SIMDBinaryPacking+VariableByte`→`simdcomp` as the dual codec providing the largest maximum speedup (70%). Given that aggregation happens after decompressing with the second codec, we highlight the agreement in both ranking metrics for `simdcomp` as the codec to fuse summing into decompression.

Based on our projections and practicality, we select the below codecs to fuse summing into decompression. While high-ranking, we avoid modifying TurboPFor codecs due to the rigidity and volatility of their implementations.

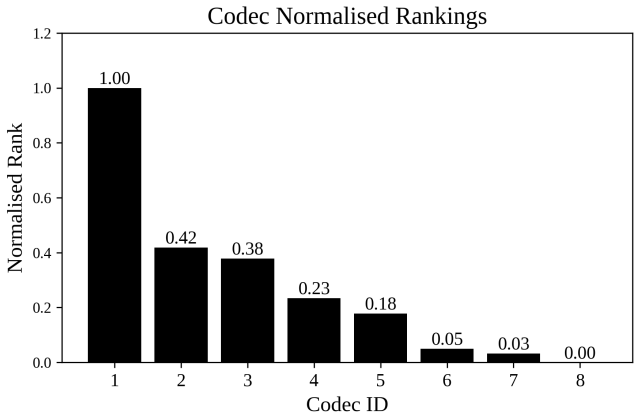


Figure 18: Normalised rankings of codecs for summing pipelines with a single transformation. For these codecs, aggregation is not fused into decompression, however the execution time excludes aggregation time to project the rankings after fusing aggregation into decompression. The names corresponding to each Codec ID are given in Appendix D.2, Table 11.

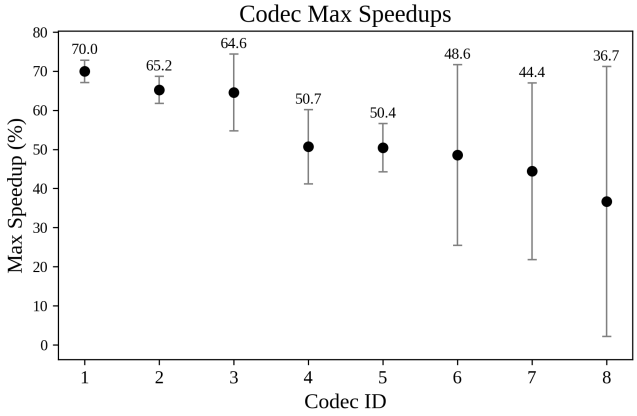


Figure 19: Max speedups of codecs for summing pipelines with a single transformation. For these codecs, aggregation is not fused into decompression, however the execution time excludes aggregation time to project the max speedups after fusing aggregation into decompression. The names corresponding to each Codec ID are given in Appendix D.2, Table 12.

1. `simdcomp` (rank 1 in the dual codec normalised rankings),
2. `custom_rle_vecavx512` (rank 3 in the dual codec normalised rankings),
3. `FastPFor_SIMDPFor+VariableByte` (rank 8 in the

dual codec normalised rankings).

4 Discussion

Our research applied modern lightweight compression to GeoTIFF-stored data, addressing the bottleneck arising from CPU bandwidth surpassing RAM bandwidth. Through SIMD-optimised compression, we successfully maintained data in a compressed form closer to the CPU caches, improving pipeline execution times. Pareto optimisation helped in identifying the most effective compression schemes, leading to substantial speedups in geospatial pipelines. We obtain relative speedups of up to 29% for pipelines transforming and subsequently randomly sampling data within blocks, and up to 78% for pipelines transforming and subsequently summing blocks, after fusing summing into the decompression process. Further, we comprehensively rank a broad range of codecs, narrowing the focus for future research.

However, the study is not without limitations. Due to implementation difficulties, we did not fuse the compression and decompression stages of our cascaded codecs. Thus, our cascades perform an additional unnecessary pass through the data, reducing their cache-efficiency and in-tern their performance. As such, our speedups are **underestimations** and can be further increased by fusing additional stages of the pipeline (for example, fusing transformations into decompression). Furthermore, we overlooked some codecs to fuse summing into decompression, namely our custom DELTA codec. Finally, we note the preprocessing overhead of our method. We did not quantify this overhead, and it could be useful to know when assessing the feasibility of our method in practice.

Our results extend Damme *et al.* [12]'s observed speedups in databases to geospatial contexts. Furthermore, we address the gap highlighted by Zalipynis [5] of lightweight compression applied to geospatial pipelines. We reinforce their conclusion that heavyweight compressors are not fit for accelerating in-memory geospatial computation, and provide evidence that lightweight codecs are fit-for-purpose, as more generally proposed by Lemire *et al.* [3]. Our approach not only mitigates the RAM bottleneck but also provides a framework for future exploration of lightweight compression in this domain.

The broader implications of our work are significant, offering a pathway to more efficient large-scale environmental data analysis. This is particularly relevant for real-time applications in climate change monitoring and ecosystem conservation, where rapid data processing can facilitate timely decision-making and intervention. Promisingly, since the gap between CPU and RAM continues to increase [2], the speedups by our method will increase over time.

Looking forward, applying the method to pipelines with additional transformations and aggregations presents a promising avenue for research. Given the observed benefits of fusing aggregation into decompression, fusing more stages of the pipelines could yield larger speedups. To this end, we suggest fusing transformations into decompression, fusing more aggregations into decompression, and fusing cascaded codecs into a single pass. Finally, in-practice the best codecs are not known a priori. For real-world adoption, we suggest implementing a 'query-engine' which uses heuristics to select the best codecs for the pipeline based on characteristics of the data to be processed.

References

1. Holcomb, A. *et al. A Case for Planetary Computing* arXiv:2303.04501 [cs]. Mar. 2023. `http://arxiv.org/abs/2303.04501` (2024).
2. Bjorlin, A. in *Infrastructure for Large Scale AI: "Empowering Open"* (2022). `https://www.opencompute.org/events/past-events/2022-ocp-global-summit`.
3. Lemire, D. & Boytsov, L. Decoding billions of integers per second through vectorization. en. *Software: Practice and Experience* **45,** 1–29. ISSN: 1097-024X. `https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2203` (2023) (2015).
4. Cresson, R. & Hautreux, G. A Generic Framework for the Development of Geospatial Processing Pipelines on Clusters. *IEEE Geoscience and Remote Sensing Letters* **13.** Conference Name: IEEE Geoscience and Remote Sensing Letters, 1706–1710. ISSN: 1558-0571. `https://ieeexplore.ieee.org/document/7572892` (2024) (Nov. 2016).
5. Zalipynis, R. A. R. *Evaluating Array DBMS Compression Techniques for Big Environmental Datasets* en. in *2019 10th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)* (IEEE, Metz, France, Sept. 2019), 859–863. ISBN: 978-1-72814-069-8. `https://ieeexplore.ieee.org/document/8924326/` (2024).
6. Heinzl, L., Hurdelhey, B., Boissier, M., Perscheid, M. & Plattner, H. Evaluating Lightweight Integer Compression Algorithms in Column-Oriented In-Memory DBMS. en (2021).
7. Li, J., Finn, M. P. & Blanco Castano, M. A Lightweight CUDA-Based Parallel Map Reprojection Method for Raster Datasets of Continental to Global Extent. en. *ISPRS International Journal of Geo-Information* **6.** Number: 4 Publisher: Multidisciplinary Digital Publishing Institute, 92. ISSN: 2220-9964. `https://www.mdpi.com/2220-9964/6/4/92` (2024) (Apr. 2017).

8. Rothberg, E., Singh, J. P. & Gupta, A. *Working sets, cache sizes, and node granularity issues for large-scale multiprocessors* in *Proceedings of the 20th annual international symposium on computer architecture* (Association for Computing Machinery, New York, NY, USA, May 1993), 14–26. ISBN: 978-0-8186-3810-7. https://dl.acm.org/doi/10.1145/165123.165126 (2023).

9. Alam, M. M., Ray, S. & Bhavsar, V. C. *A Performance Study of Big Spatial Data Systems* in *Proceedings of the 7th ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data* (Association for Computing Machinery, New York, NY, USA, Nov. 2018), 1–9. ISBN: 978-1-4503-6041-8. https://dl.acm.org/doi/10.1145/3282834.3282841 (2024).

10. Qin, C.-Z., Zhan, L.-J. & Zhu, A.-X. How to Apply the Geospatial Data Abstraction Library (GDAL) Properly to Parallel Geospatial Raster I/O? en. *Transactions in GIS* **18,** 950–957. ISSN: 1467-9671. https://onlinelibrary.wiley.com/doi/abs/10.1111/tgis.12068 (2024) (2014).

11. Pohl, C. & Sattler, K.-U. *Joins in a heterogeneous memory hierarchy: exploiting high-bandwidth memory* in *Proceedings of the 14th International Workshop on Data Management on New Hardware* (Association for Computing Machinery, New York, NY, USA, June 2018), 1–10. ISBN: 978-1-4503-5853-8. https://dl.acm.org/doi/10.1145/3211922.3211929 (2024).

12. Damme, P., Habich, D., Hildebrandt, J. & Lehner, W. *Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses)* en. 2017. https://openproceedings.org/2017/conf/edbt/paper-146.pdf (2023).

13. Praveen, P., Babu, C. J. & Rama, B. *Big data environment for geospatial data analysis* in *2016 International Conference on Communication and Electronics Systems (ICCES)* (Oct. 2016), 1–6. https://ieeexplore.ieee.org/abstract/document/7889816 (2024).

14. Gorelick, N. *et al.* Google Earth Engine: Planetary-scale geospatial analysis for everyone. *Remote Sensing of Environment. Big Remotely Sensed Data: tools, applications and experiences* **202,** 18–27. ISSN: 0034-4257. https://www.sciencedirect.com/science/article/pii/S0034425717302900 (2024) (Dec. 2017).

15. *OGC GeoTIFF Standard* https://www.ogc.org/standard/geotiff/ (2023).

16. Lam, M. D., Rothberg, E. E. & Wolf, M. E. The cache performance and optimizations of blocked algorithms. *ACM SIGPLAN Notices* **26,** 63–74. ISSN: 0362-1340. https://dl.acm.org/doi/10.1145/106973.106981 (2023) (Apr. 1991).

17. Zhao, W. X. *et al.* A General SIMD-Based Approach to Accelerating Compression Algorithms. *ACM Transactions on Information Systems* **33,** 15:1–15:28. ISSN: 1046-8188. https://doi.org/10.1145/2735629 (2023) (Mar. 2015).

18. Pibiri, G. E. & Venturini, R. Techniques for Inverted Index Compression. en. *ACM Computing Surveys* **53,** 1–36. ISSN: 0360-0300, 1557-7341. https://dl.acm.org/doi/10.1145/3415148 (2023) (Nov. 2021).

19. Ngatchou, P., Zarei, A. & El-Sharkawi, A. *Pareto Multi Objective Optimization* in *Proceedings of the 13th International Conference on, Intelligent Systems Application to Power Systems* (Nov. 2005), 84–91. https://ieeexplore.ieee.org/abstract/document/1599245 (2024).

20. Baert, J. *Libmorton: C++ Morton Encoding/Decoding Library* https://github.com/Forceflow/libmorton. 2018.

21. Seltman, H. *Approximations for Mean and Variance of a Ratio* https://www.stat.cmu.edu/~hseltman/files/ratio.pdf.

Appendices

A Pipeline Test Data

This appendix shows the test data / GeoTIFFs used in our experiments. We crop and brighten some images for clarity. Furthermore, we downscale the images due to their large file-size.

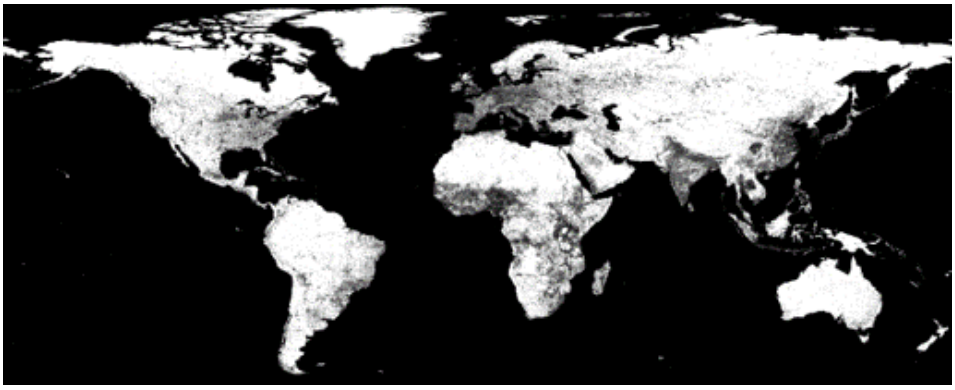


Figure 20: File A (original but downscaled).

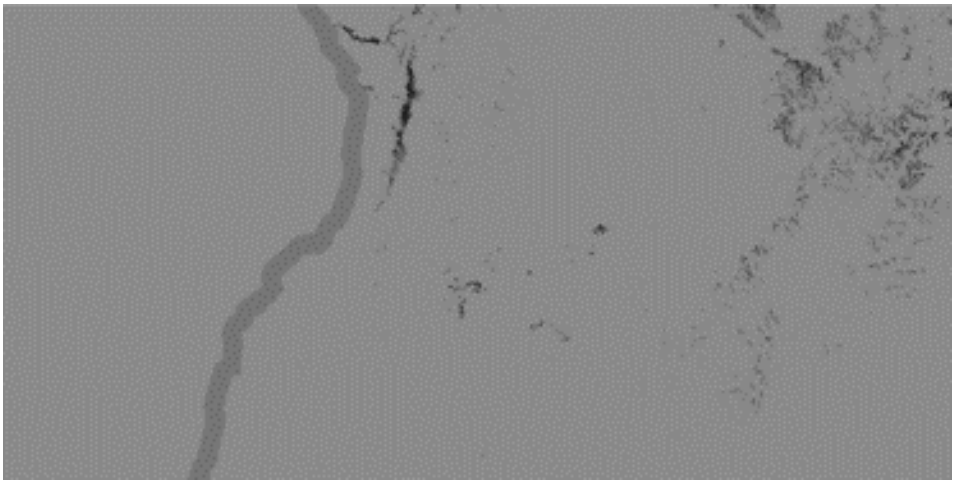


Figure 21: File B (brightened and cropped, downscaled).

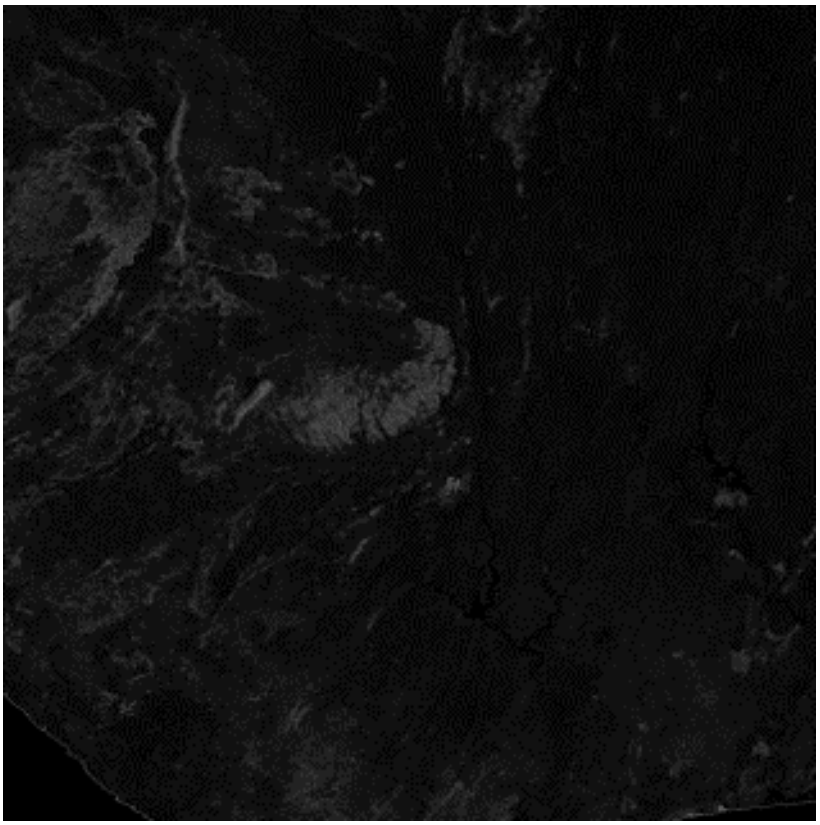
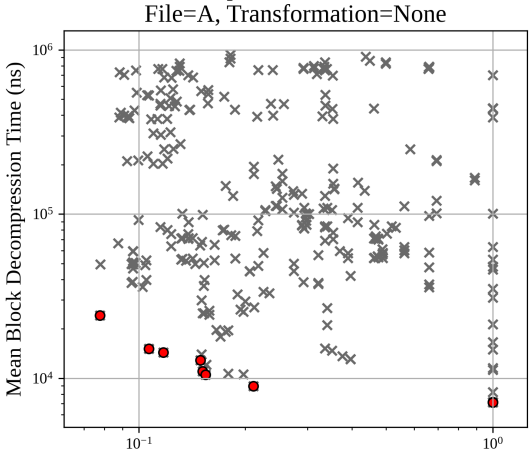


Figure 22: File C (brightened, downscaled).

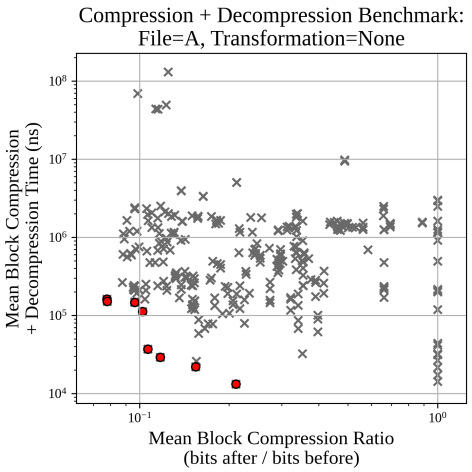
Figure 23: File D (original but downscaled).

B Space/Time Pareto Fronts

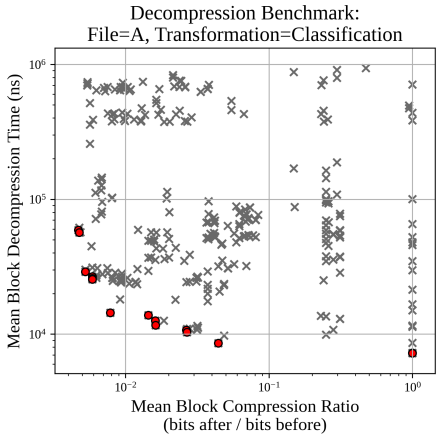
This appendix shows the space/time plots for every (file,transformation) combination. Grey crosses are the benchmarks for each codec. Red circles are the codecs on the Pareto front. The names of the codecs on the Pareto fronts are provided below the plots, ordered by smallest compression ratio first.



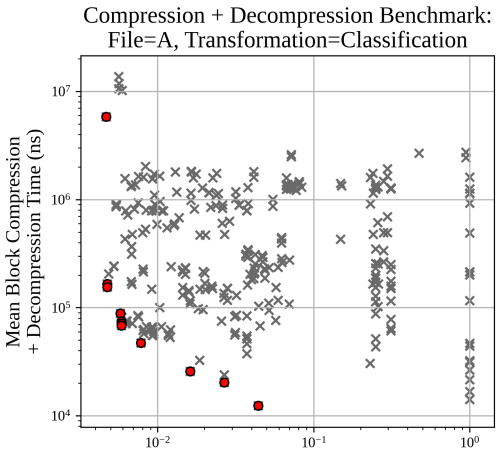
1: TurboPFor_Zigzag+TurboPFor256
2: TurboPFor_Zigzag+TurboPack256
3: TurboPFor_xor+TurboPack256
4: TurboPFor_TurboPFor256
5: FastPFor_SIMDFastPFor256+VariableByte
6: FastPFor_SIMDBinaryPacking+VariableByte
7: simdcomp
8: custom_delta_vecavx512
9: custom_for_vecavx512



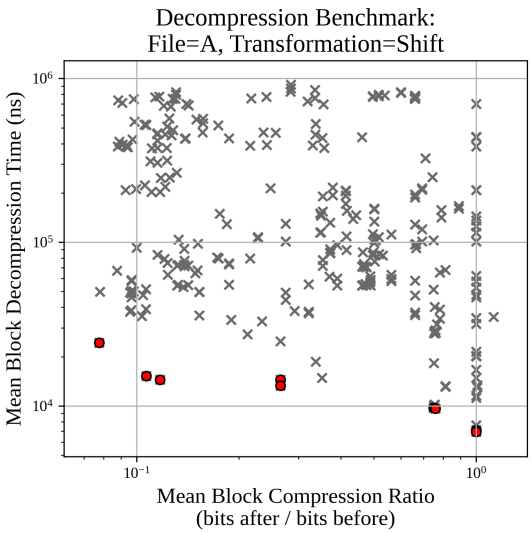
1: TurboPFor_Zigzag+TurboPFor256
2: TurboPFor_Zigzag+TurboVSimple
3: [+]_custom_delta_vecavx512+FastPFor_FastBinaryPacking32+VariableByte
4: [+]_custom_delta_vecavx512+FastPFor_SIMDBinaryPacking+VariableByte
5: TurboPFor_Zigzag+TurboPack256
6: TurboPFor_xor+TurboPack256
7: FastPFor_SIMDBinaryPacking+VariableByte
8: simdcomp



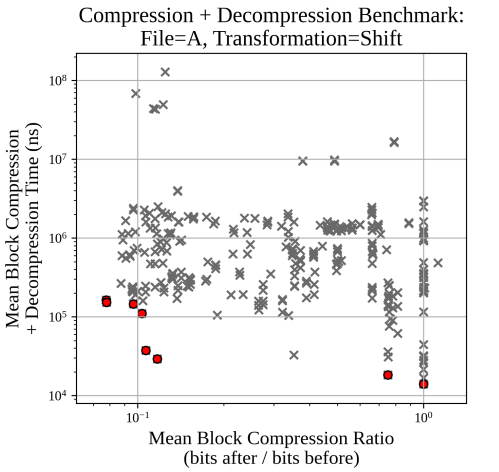
1: [+]_custom_rle_vecavx512+DEFLATE
2: [+]_custom_rle_vecavx512+Zstd_3
3: [+]_custom_rle_vecavx512+FastPFor_SIMDOPTPFor<4,Simple16>+VariableByte
4: [+]_custom_rle_vecavx512+FastPFor_FastPFor128+VariableByte
5: [+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor128+VariableByte
6: [+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor256+VariableByte
7: TurboPFor_TurboVSimple
8: TurboPFor_TurboPFor128
9: TurboPFor_xor+TurboPack256
10: TurboPFor_TurboPFor256
11: FastPFor_SIMDFastPFor256+VariableByte
12: FastPFor_SIMDBinaryPacking+VariableByte
13: simdcomp
14: custom_delta_vecavx512
15: custom_for_vecavx512



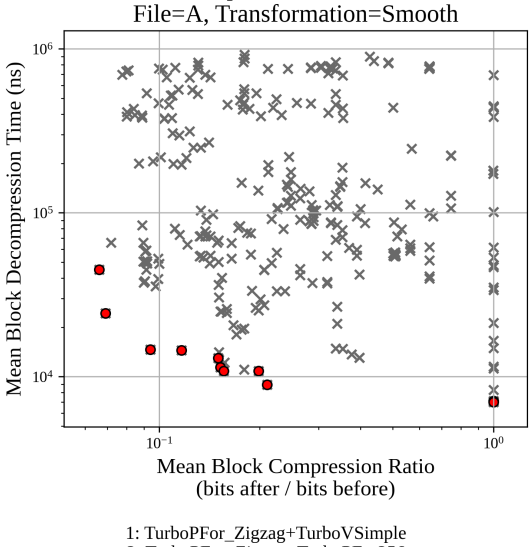
1: [+]_custom_rle_vecavx512+DEFLATE
2: [+]_custom_rle_vecavx512+Zstd_3
3: [+]_custom_rle_vecavx512+Zstd_1
4: [+]_custom_rle_vecavx512+FastPFor_SIMDSimplePFor+VariableByte
5: [+]_custom_rle_vecavx512+FastPFor_FastPFor128+VariableByte
6: [+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor128+VariableByte
7: [+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor256+VariableByte
8: TurboPFor_TurboVSimple
9: TurboPFor_xor+TurboPack256
10: FastPFor_SIMDBinaryPacking+VariableByte
11: simdcomp



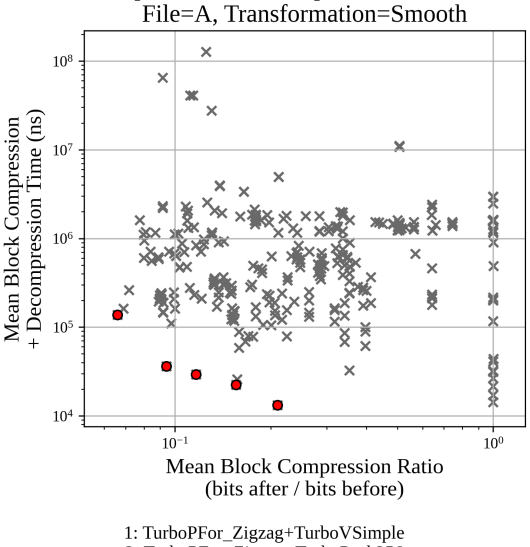
1: TurboPFor_Zigzag+TurboPFor256
2: TurboPFor_Zigzag+TurboPack256
3: TurboPFor_xor+TurboPack256
4: TurboPFor_TurboPFor128
5: TurboPFor_TurboPFor256
6: simdcomp
7: FastPFor_SIMDPFor+VariableByte
8: custom_delta_vecavx512
9: custom_for_vecavx512



1: TurboPFor_Zigzag+TurboPFor256
2: TurboPFor_Zigzag+TurboVSimple
3: [+]_custom_delta_vecavx512+FastPFor_FastBinaryPacking32+VariableByte
4: [+]_custom_delta_vecavx512+FastPFor_SIMDBinaryPacking+VariableByte
5: TurboPFor_Zigzag+TurboPack256
6: TurboPFor_xor+TurboPack256
7: simdcomp
8: custom_delta_vecavx512

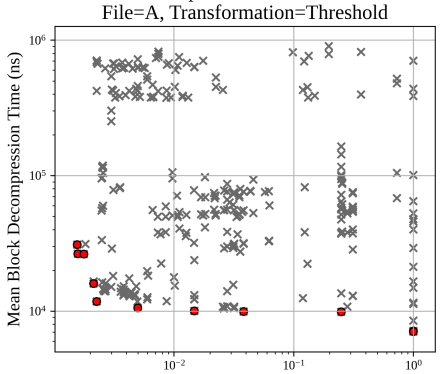


1: TurboPFor_Zigzag+TurboVSimple
2: TurboPFor_Zigzag+TurboPFor256
3: TurboPFor_Zigzag+TurboPack256
4: TurboPFor_xor+TurboPack256
5: TurboPFor_TurboPFor256
6: FastPFor_SIMDFastPFor256+VariableByte
7: FastPFor_SIMDBinaryPacking+VariableByte
8: FastPFor_SIMDPFor+VariableByte
9: simdcomp
10: custom_delta_vecavx512
11: custom_for_vecavx512



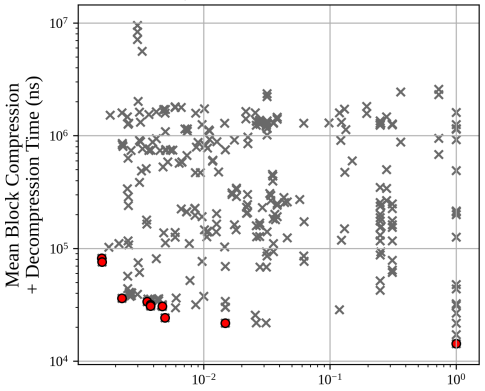
1: TurboPFor_Zigzag+TurboVSimple
2: TurboPFor_Zigzag+TurboPack256
3: TurboPFor_xor+TurboPack256
4: FastPFor_SIMDBinaryPacking+VariableByte
5: simdcomp

Decompression Benchmark:
File=A, Transformation=Threshold



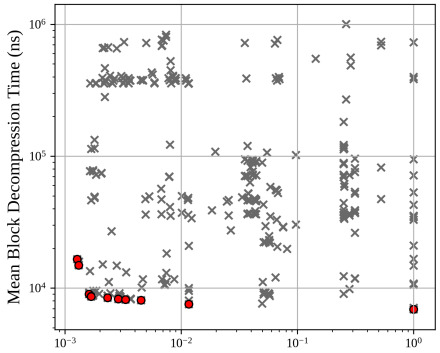
1: [+]_custom_rle_vecavx512+Zstd_1
2: [+]_custom_rle_vecavx512+Zstd_3
3: [+]_custom_rle_vecavx512+Zstd_5
4: [+]_custom_rle_vecavx512+FastPFor_SIMDOPTPFor<4,Simple16>+VariableByte
5: TurboPFor_TurboVSimple
6: TurboPFor_TurboPFor256
7: custom_rle_vecavx512
8: FastPFor_SIMDPFor+VariableByte
9: FastPFor_MaskedVByte
10: custom_delta_vecavx512
11: custom_for_vecavx512

Compression + Decompression Benchmark:
File=A, Transformation=Threshold



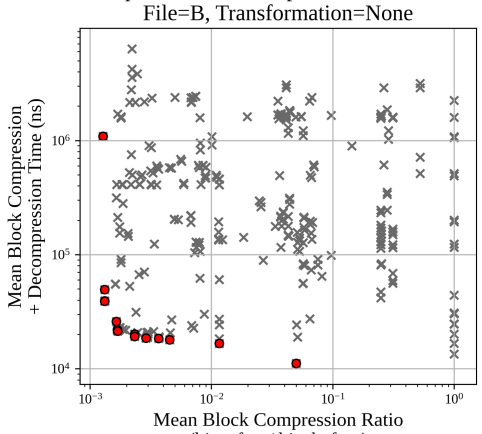
1: [+]_custom_rle_vecavx512+Zstd_1
2: [+]_custom_rle_vecavx512+Zstd_3
3: TurboPFor_TurboVSimple
4: [+]_custom_rle_vecavx512+FastPFor_FastBinaryPacking16+VariableByte
5: [+]_custom_rle_vecavx512+FastPFor_FastBinaryPacking32+VariableByte
6: [+]_custom_rle_vecavx512+FastPFor_VByte
7: [+]_custom_rle_vecavx512+FastPFor_varintgb
8: TurboPFor_xor+TurboPack256
9: custom_rle_vecavx512
10: custom_delta_vecavx512

Decompression Benchmark:
File=B, Transformation=None

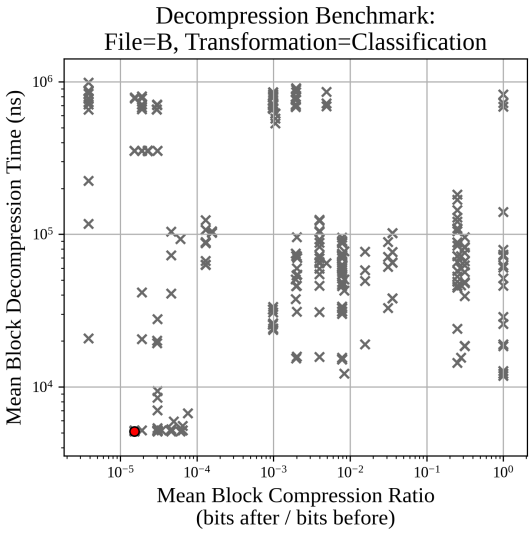


1: [+]_custom_rle_vecavx512+DEFLATE
2: [+]_custom_rle_vecavx512+Zstd_5
3: [+]_custom_rle_vecavx512+FastPFor_SIMDOPTPFor<4,Simple16>+VariableByte
4: [+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor128+VariableByte
5: [+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor256+VariableByte
6: [+]_custom_rle_vecavx512+FastPFor_FastBinaryPacking32+VariableByte
7: [+]_custom_rle_vecavx512+FastPFor_SIMDBinaryPacking+VariableByte
8: [+]_custom_rle_vecavx512+FastPFor_VarIntG8IU
9: [+]_custom_rle_vecavx512+simdcomp
10: custom_rle_vecavx512
11: custom_delta_vecavx512
12: custom_for_vecavx512

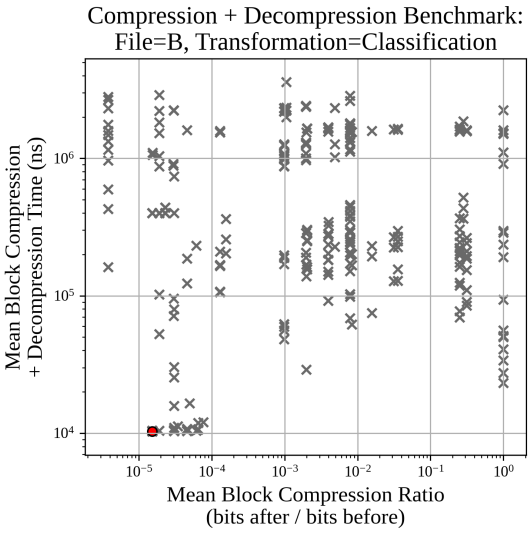
Compression + Decompression Benchmark:
File=B, Transformation=None



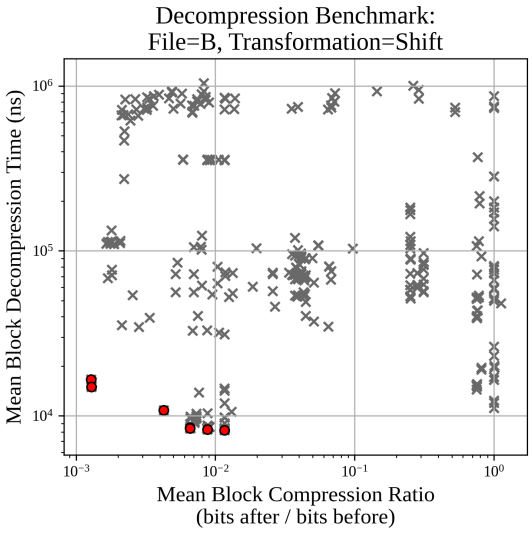
1: [+]_custom_rle_vecavx512+DEFLATE
2: [+]_custom_rle_vecavx512+Zstd_5
3: [+]_custom_rle_vecavx512+Zstd_3
4: [+]_custom_rle_vecavx512+Zstd_1
5: [+]_custom_rle_vecavx512+FastPFor_SIMDSimplePFor+VariableByte
6: [+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor128+VariableByte
7: [+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor256+VariableByte
8: [+]_custom_rle_vecavx512+FastPFor_FastBinaryPacking16+VariableByte
9: [+]_custom_rle_vecavx512+FastPFor_FastBinaryPacking32+VariableByte
10: [+]_custom_rle_vecavx512+FastPFor_SIMDBinaryPacking+VariableByte
11: [+]_custom_rle_vecavx512+STREAMVBYTE
12: [+]_custom_rle_vecavx512+simdcomp
13: custom_rle_vecavx512
14: simdcomp



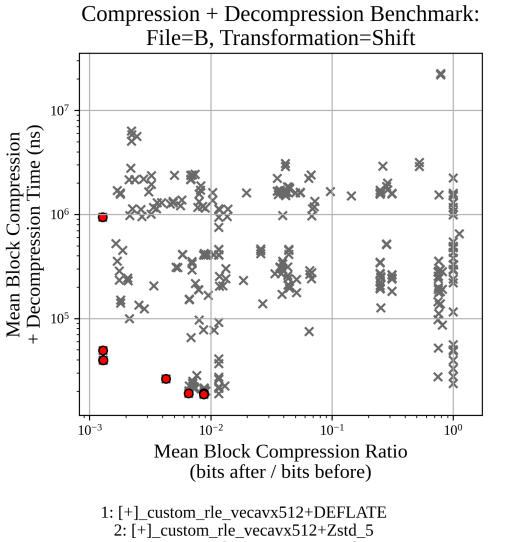
1: simdcomp
2: [+]_custom_rle_vecavx512+FastPFor_VariableByte



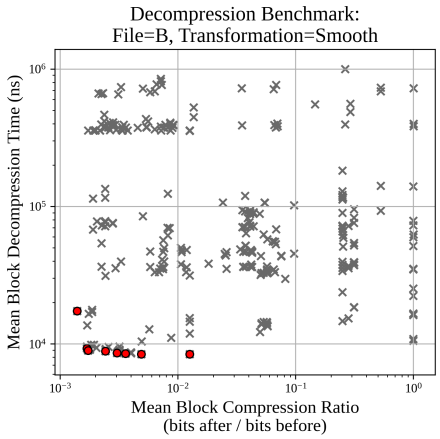
1: simdcomp
2: [+]_custom_rle_vecavx512+FastPFor_VariableByte



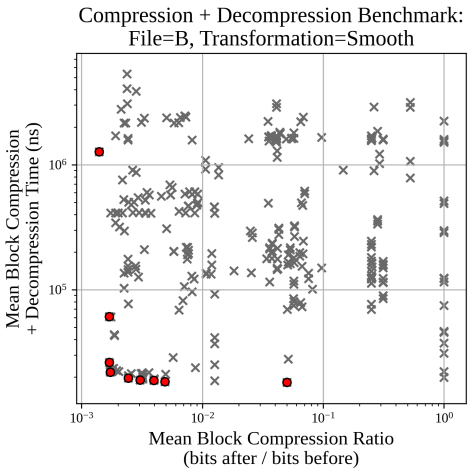
1: [+]_custom_rle_vecavx512+DEFLATE
2: [+]_custom_rle_vecavx512+Zstd_5
3: [+]_custom_rle_vecavx512+LZ4
4: [+]_custom_rle_vecavx512+STREAMVBYTE
5: [+]_custom_rle_vecavx512+FastPFor_streamvbyte
6: [+]_custom_rle_vecavx512+simdcomp
7: [+]_custom_rle_vecavx512+FastPFor_JustCopy



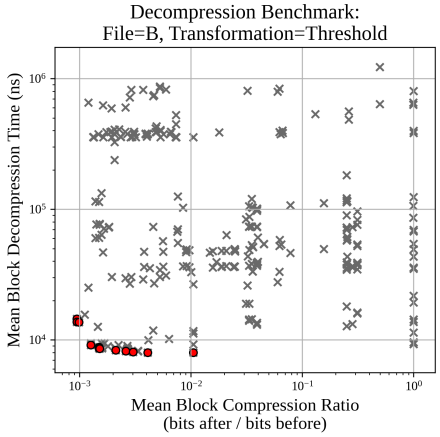
1: [+]_custom_rle_vecavx512+DEFLATE
2: [+]_custom_rle_vecavx512+Zstd_5
3: [+]_custom_rle_vecavx512+Zstd_3
4: [+]_custom_rle_vecavx512+Zstd_1
5: [+]_custom_rle_vecavx512+LZ4
6: [+]_custom_rle_vecavx512+STREAMVBYTE
7: [+]_custom_rle_vecavx512+FastPFor_SIMDBinaryPacking+VariableByte
8: [+]_custom_rle_vecavx512+simdcomp



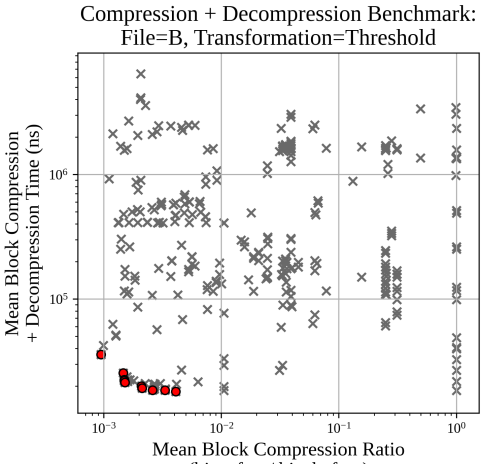
1: [+]_custom_rle_vecavx512+DEFLATE
2: [+]_custom_rle_vecavx512+FastPFor_SIMDOPTPFor<4,Simple16>+VariableByte
3: [+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor256+VariableByte
4: [+]_custom_rle_vecavx512+FastPFor_FastBinaryPacking32+VariableByte
5: [+]_custom_rle_vecavx512+FastPFor_SIMDBinaryPacking+VariableByte
6: [+]_custom_rle_vecavx512+FastPFor_VarIntG8IU
7: [+]_custom_rle_vecavx512+simdcomp
8: [+]_custom_rle_vecavx512+FastPFor_JustCopy



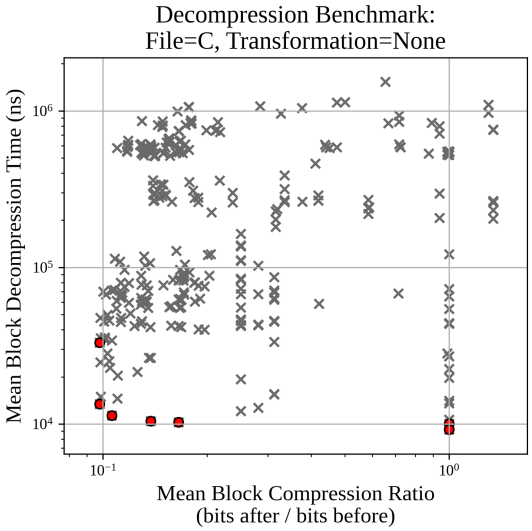
1: [+]_custom_rle_vecavx512+DEFLATE
2: [+]_custom_rle_vecavx512+FastPFor_OPTPFor<4,Simple16>+VariableByte
3: [+]_custom_rle_vecavx512+FastPFor_SIMDSimplePFor+VariableByte
4: [+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor256+VariableByte
5: [+]_custom_rle_vecavx512+FastPFor_FastBinaryPacking32+VariableByte
6: [+]_custom_rle_vecavx512+FastPFor_SIMDBinaryPacking+VariableByte
7: [+]_custom_rle_vecavx512+STREAMVBYTE
8: [+]_custom_rle_vecavx512+simdcomp
9: simdcomp



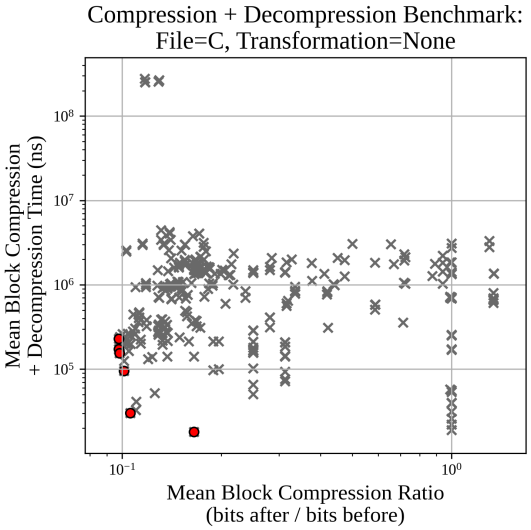
1: [+]_custom_rle_vecavx512+Zstd_1
2: [+]_custom_rle_vecavx512+Zstd_3
3: [+]_custom_rle_vecavx512+Zstd_5
4: [+]_custom_rle_vecavx512+FastPFor_SIMDOPTPFor<4,Simple16>+VariableByte
5: [+]_custom_rle_vecavx512+FastPFor_FastPFor128+VariableByte
6: [+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor128+VariableByte
7: [+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor256+VariableByte
8: [+]_custom_rle_vecavx512+FastPFor_FastBinaryPacking32+VariableByte
9: [+]_custom_rle_vecavx512+FastPFor_SIMDBinaryPacking+VariableByte
10: [+]_custom_rle_vecavx512+FastPFor_VarIntG8IU
11: [+]_custom_rle_vecavx512+simdcomp
12: [+]_custom_rle_vecavx512+FastPFor_JustCopy



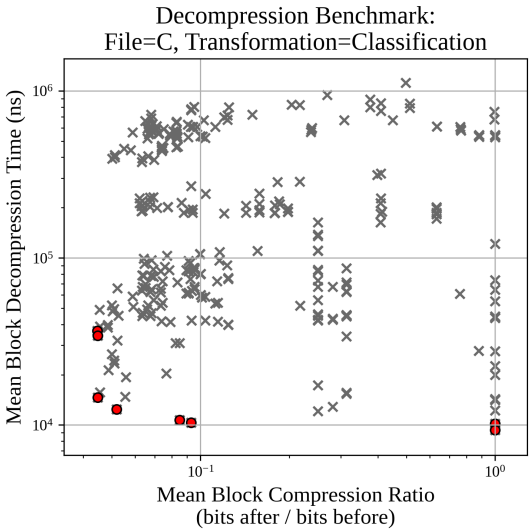
1: [+]_custom_rle_vecavx512+Zstd_1
2: [+]_custom_rle_vecavx512+FastPFor_SIMDSimplePFor+VariableByte
3: [+]_custom_rle_vecavx512+FastPFor_FastPFor128+VariableByte
4: [+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor128+VariableByte
5: [+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor256+VariableByte
6: [+]_custom_rle_vecavx512+FastPFor_FastBinaryPacking16+VariableByte
7: [+]_custom_rle_vecavx512+FastPFor_FastBinaryPacking32+VariableByte
8: [+]_custom_rle_vecavx512+FastPFor_SIMDBinaryPacking+VariableByte
9: [+]_custom_rle_vecavx512+STREAMVBYTE
10: [+]_custom_rle_vecavx512+simdcomp



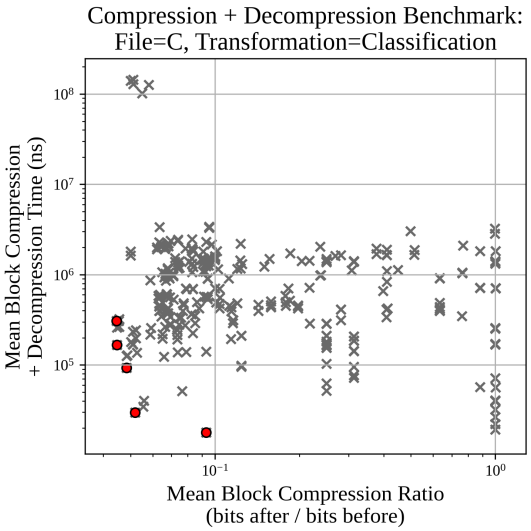
1: FastPFor_FastPFor256+VariableByte
2: FastPFor_SIMDFastPFor256+VariableByte
3: FastPFor_SIMDBinaryPacking+VariableByte
4: FastPFor_SIMDPFor+VariableByte
5: simdcomp
6: custom_delta_vecavx512
7: custom_for_vecavx512



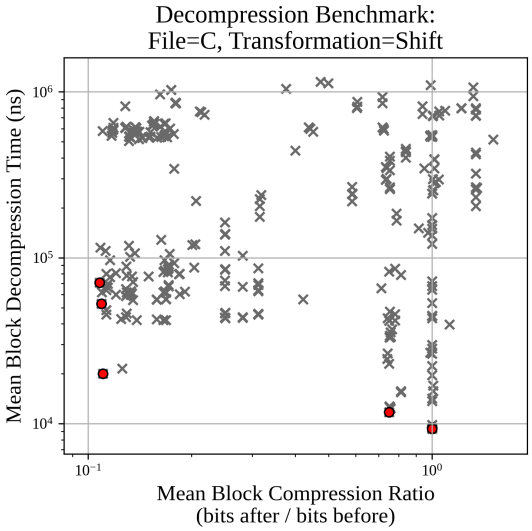
1: FastPFor_FastPFor256+VariableByte
2: FastPFor_SIMDFastPFor256+VariableByte
3: FastPFor_Simple8b
4: FastPFor_FastBinaryPacking32+VariableByte
5: FastPFor_SIMDBinaryPacking+VariableByte
6: simdcomp



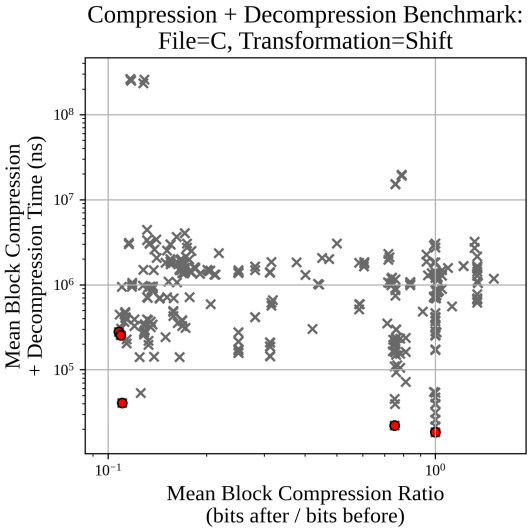
1: FastPFor_FastPFor256+VariableByte
2: TurboPFor_TurboVSimple
3: FastPFor_SIMDFastPFor256+VariableByte
4: FastPFor_SIMDBinaryPacking+VariableByte
5: FastPFor_SIMDPFor+VariableByte
6: simdcomp
7: custom_delta_vecavx512
8: custom_for_vecavx512



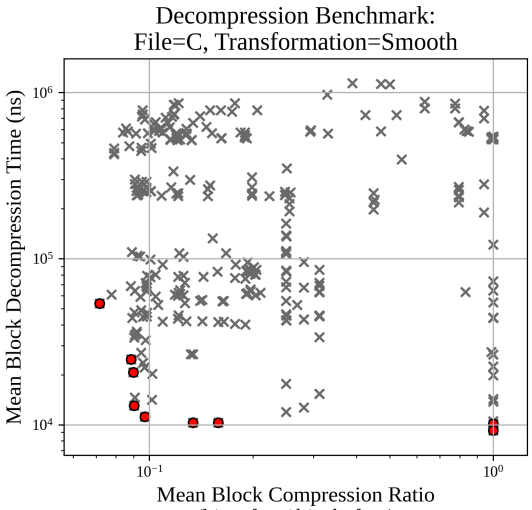
1: FastPFor_FastPFor256+VariableByte
2: TurboPFor_TurboVSimple
3: FastPFor_FastBinaryPacking32+VariableByte
4: FastPFor_SIMDBinaryPacking+VariableByte
5: simdcomp



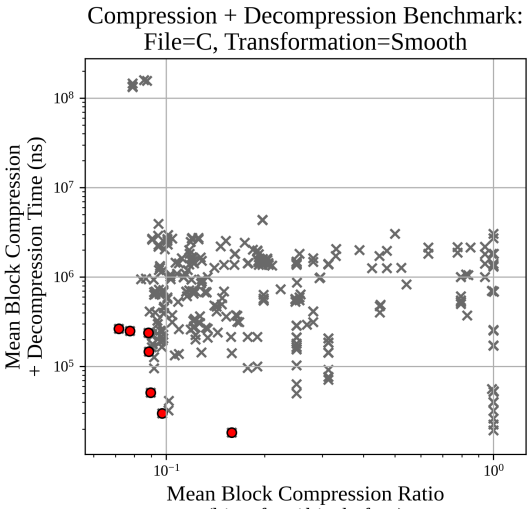
1: TurboPFor_Zigzag+TurboVSimple
2: TurboPFor_Zigzag+TurboPFor256
3: TurboPFor_xor+TurboPack256
4: simdcomp
5: FastPFor_JustCopy
6: custom_for_vecavx512



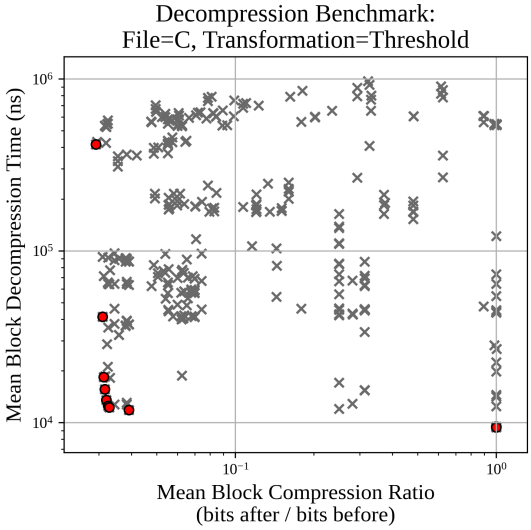
1: TurboPFor_Zigzag+TurboVSimple
2: TurboPFor_Zigzag+TurboPFor256
3: TurboPFor_xor+TurboPack256
4: simdcomp
5: custom_delta_vecavx512



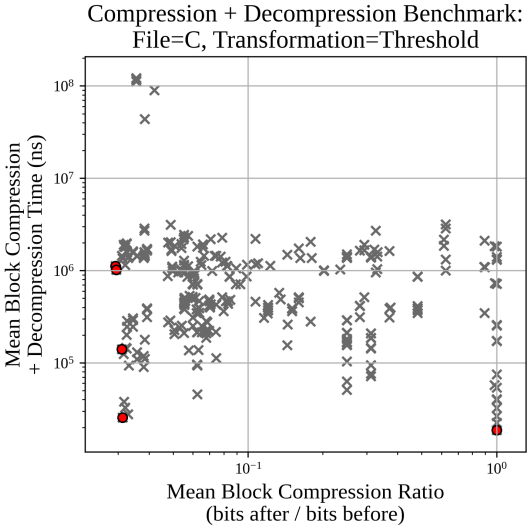
1: TurboPFor_Zigzag+TurboPFor256
2: FastPFor_Simple8b
3: TurboPFor_Zigzag+TurboPack256
4: FastPFor_SIMDFastPFor256+VariableByte
5: FastPFor_SIMDBinaryPacking+VariableByte
6: FastPFor_SIMDPFor+VariableByte
7: simdcomp
8: custom_delta_vecavx512
9: custom_for_vecavx512



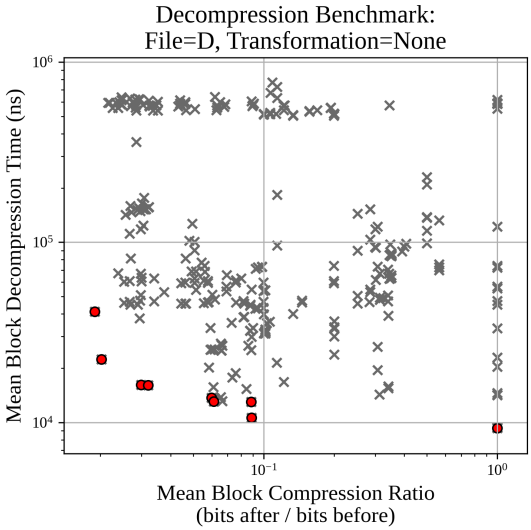
1: TurboPFor_Zigzag+TurboPFor256
2: TurboPFor_Zigzag+TurboVSimple
3: FastPFor_Simple16
4: FastPFor_Simple8b
5: TurboPFor_Zigzag+TurboPack256
6: FastPFor_SIMDBinaryPacking+VariableByte
7: simdcomp



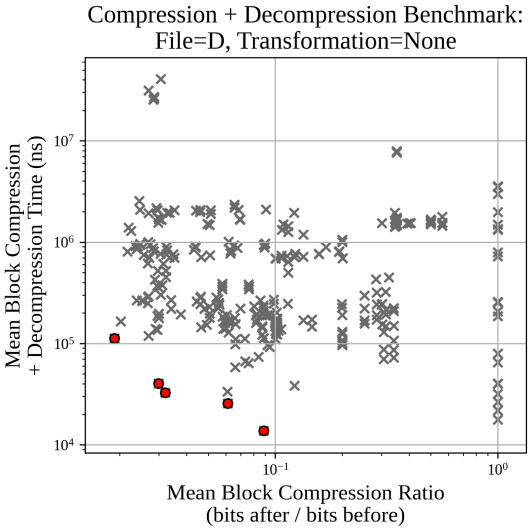
1: [+]_custom_rle_vecavx512+Zstd_3
2: FrameOfReference_Turbo
3: simdcomp
4: TurboPFor_TurboPFor256
5: TurboPFor_TurboPack256
6: FastPFor_SIMDFastPFor256+VariableByte
7: FastPFor_SIMDBinaryPacking+VariableByte
8: FastPFor_SIMDPFor+VariableByte
9: custom_delta_vecavx512
10: custom_for_vecavx512



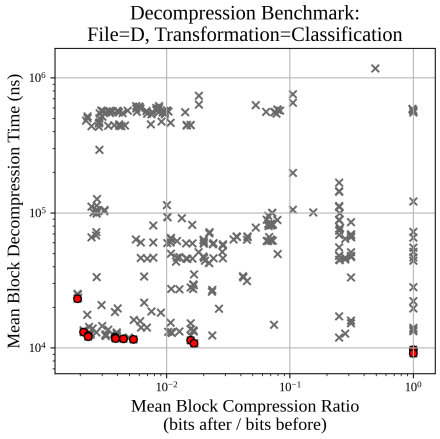
1: [+]_custom_rle_vecavx512+Zstd_3
2: [+]_custom_rle_vecavx512+Zstd_1
3: FrameOfReference_Turbo
4: simdcomp
5: custom_delta_vecavx512



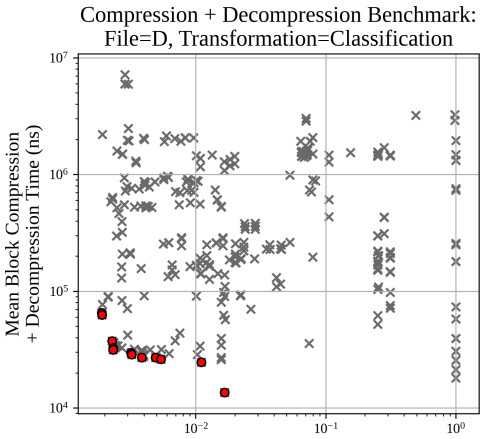
1: TurboPFor_Zigzag+TurboVSimple
2: TurboPFor_Zigzag+TurboPFor256
3: TurboPFor_Zigzag+TurboPack256
4: TurboPFor_xor+TurboPack256
5: FastPFor_SIMDFastPFor256+VariableByte
6: FastPFor_SIMDBinaryPacking+VariableByte
7: FastPFor_SIMDPFor+VariableByte
8: simdcomp
9: custom_delta_vecavx512



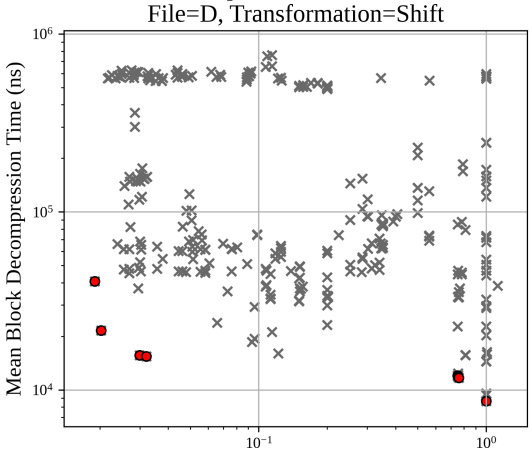
1: TurboPFor_Zigzag+TurboVSimple
2: TurboPFor_Zigzag+TurboPack256
3: TurboPFor_xor+TurboPack256
4: FastPFor_SIMDBinaryPacking+VariableByte
5: simdcomp



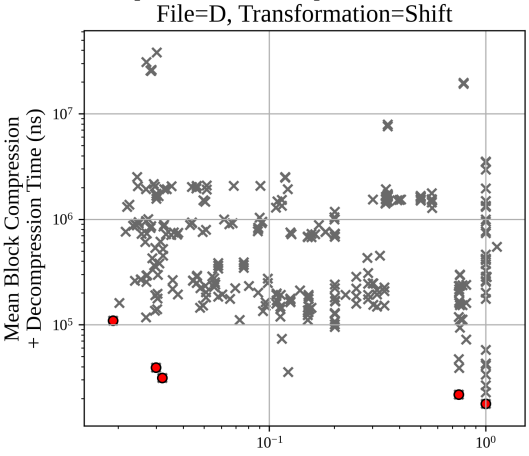
1: [+]_custom_rle_vecavx512+Zstd_3
2: [+]_custom_rle_vecavx512+FastPFor_SIMDOPTPFor<4,Simple16>+VariableByte
3: [+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor128+VariableByte
4: [+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor256+VariableByte
5: [+]_custom_rle_vecavx512+FastPFor_PFor2008+VariableByte
6: [+]_custom_rle_vecavx512+FastPFor_SIMDBinaryPacking+VariableByte
7: [+]_custom_rle_vecavx512+FastPFor_VarIntG8IU
8: [+]_custom_rle_vecavx512+simdcomp
9: [+]_custom_rle_vecavx512+FastPFor_JustCopy
10: simdcomp
11: custom_delta_vecavx512
12: custom_for_vecavx512



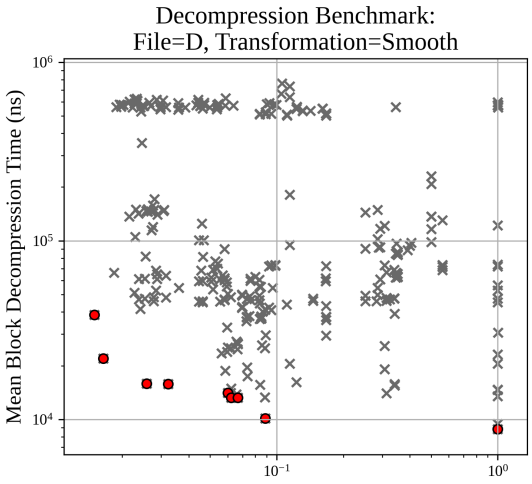
1: [+]_custom_rle_vecavx512+Zstd_3
2: [+]_custom_rle_vecavx512+Zstd_1
3: [+]_custom_rle_vecavx512+FastPFor_SIMDSimplePFor+VariableByte
4: [+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor128+VariableByte
5: [+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor256+VariableByte
6: [+]_custom_rle_vecavx512+FastPFor_FastBinaryPacking16+VariableByte
7: [+]_custom_rle_vecavx512+FastPFor_FastBinaryPacking32+VariableByte
8: [+]_custom_rle_vecavx512+FastPFor_SIMDBinaryPacking+VariableByte
9: [+]_custom_rle_vecavx512+STREAMVBYTE
10: [+]_custom_rle_vecavx512+simdcomp
11: FastPFor_SIMDBinaryPacking+VariableByte
12: simdcomp



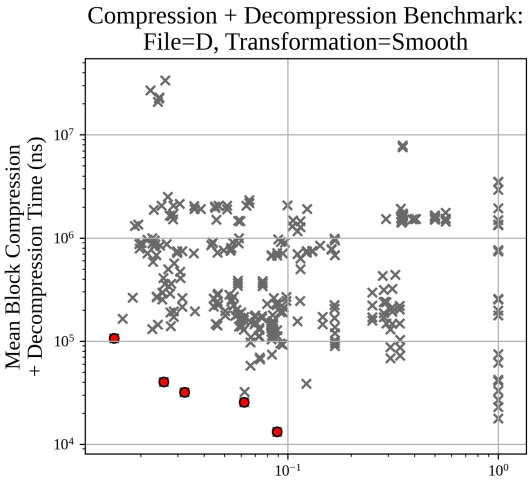
1: TurboPFor_Zigzag+TurboVSimple
2: TurboPFor_Zigzag+TurboPFor256
3: TurboPFor_Zigzag+TurboPack256
4: TurboPFor_xor+TurboPack256
5: simdcomp
6: FastPFor_SIMDFastPFor128+VariableByte
7: FastPFor_SIMDPFor+VariableByte
8: custom_delta_vecavx512



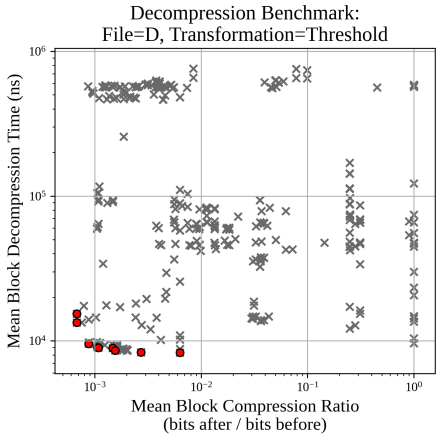
1: TurboPFor_Zigzag+TurboVSimple
2: TurboPFor_Zigzag+TurboPack256
3: TurboPFor_xor+TurboPack256
4: simdcomp
5: custom_delta_vecavx512



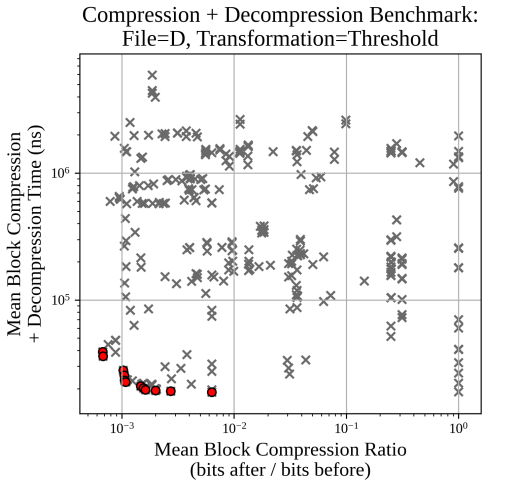
1: TurboPFor_Zigzag+TurboVSimple
2: TurboPFor_Zigzag+TurboPFor256
3: TurboPFor_Zigzag+TurboPack256
4: TurboPFor_xor+TurboPack256
5: FastPFor_SIMDFastPFor256+VariableByte
6: FastPFor_SIMDBinaryPacking+VariableByte
7: FastPFor_SIMDNewPFor<4,Simple16>+VariableByte
8: simdcomp
9: custom_delta_vecavx512



1: TurboPFor_Zigzag+TurboVSimple
2: TurboPFor_Zigzag+TurboPack256
3: TurboPFor_xor+TurboPack256
4: FastPFor_SIMDBinaryPacking+VariableByte
5: simdcomp



1: [+]_custom_rle_vecavx512+Zstd_1
2: [+]_custom_rle_vecavx512+Zstd_3
3: [+]_custom_rle_vecavx512+FastPFor_SIMDOPTPFor<4,Simple16>+VariableByte
4: [+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor128+VariableByte
5: [+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor256+VariableByte
6: [+]_custom_rle_vecavx512+FastPFor_FastBinaryPacking16+VariableByte
7: [+]_custom_rle_vecavx512+FastPFor_FastBinaryPacking32+VariableByte
8: [+]_custom_rle_vecavx512+simdcomp
9: [+]_custom_rle_vecavx512+FastPFor_JustCopy



1: [+]_custom_rle_vecavx512+Zstd_1
2: [+]_custom_rle_vecavx512+Zstd_3
3: [+]_custom_rle_vecavx512+FastPFor_SIMDSimplePFor+VariableByte
4: [+]_custom_rle_vecavx512+FastPFor_Simple16
5: [+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor128+VariableByte
6: [+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor256+VariableByte
7: [+]_custom_rle_vecavx512+FastPFor_FastBinaryPacking16+VariableByte
8: [+]_custom_rle_vecavx512+FastPFor_FastBinaryPacking32+VariableByte
9: [+]_custom_rle_vecavx512+FastPFor_VByte
10: [+]_custom_rle_vecavx512+STREAMVBYTE
11: [+]_custom_rle_vecavx512+simdcomp
12: custom_rle_vecavx512

C Pipeline Execution Times

This appendix shows, for each (file,transformation,aggregation) combination, the best-found Pareto codec, the pipeline's execution time $\mu \pm \sigma$ with the codec, and the speedup over the non-optimised pipeline. We first show the results without fusing sum aggregations into decompression. Then, we show the results with fusing sum aggregations into decompression.

For 'None' transformations, the pipeline uses a single codec, hence the codec is formatted as a single, potentially composite, codec. For other non-'None' transformations, the pipeline uses two codecs, hence the codec is formatted as `codec1` → `codec2` where `codec1` and `codec2` are both single, potentially composite, codecs.

Given an optimised pipeline with execution time $\mu_\uparrow \pm \sigma_\uparrow$ and a non-optimised pipeline with execution time $\mu_\downarrow \pm \sigma_\downarrow$, the speedup $\mu_\Delta \pm \sigma_\Delta$ is given by:

$$\mu_\Delta = 100\frac{\mu_\downarrow - \mu_\uparrow}{\mu_\downarrow} = 100(1 - \frac{\mu_\uparrow}{\mu_\downarrow}) \tag{7}$$

$$\sigma_\Delta^2 \approx 100^2 \left(\frac{\mu_\uparrow^2}{\mu_\downarrow^2}\right)\left(\frac{\sigma_\uparrow^2}{\mu_\uparrow^2} + \frac{\sigma_\downarrow^2}{\mu_\downarrow^2}\right) \tag{8}$$

Equation 7 is obtained from basic statistics. Equation 8 is obtained by Seltman's approximation of the variance of a ratio [21] and assuming zero covariance.

3.1 Without Fusing Sum into Decompression

File	Transform	Aggregation	Best Codec	Execution Time (ns)		Speedup (%)	
				μ	σ	μ	σ
A	Smooth	Sum	simdcomp → simdcomp	252323.90	19614.89	-1.81	9
B	Smooth	Sum	custom_rle_vecavx512 → simdcomp	248599.15	23262.43	-6.57	11
C	Smooth	Sum	FastPFor_SIMDPFor +VariableByte → simdcomp	259383.62	14408.26	-4.64	8.28
D	Smooth	Sum	simdcomp → simdcomp	223581.65	12189.14	10.34	6.42
A	Smooth	Random Sample	simdcomp → simdcomp	242701.45	17423.69	13.91	7.02
B	Smooth	Random Sample	custom_rle_vecavx512 → simdcomp	234809.75	24115.39	7.26	10.7
C	Smooth	Random Sample	FastPFor_SIMDPFor +VariableByte → simdcomp	249777.42	18862.39	8.7	9.05
D	Smooth	Random Sample	simdcomp → simdcomp	223843.95	16160.12	17.05	7.26
A	Shift	Sum	simdcomp → simdcomp	176268.82	11157.03	-21.59	11.93
B	Shift	Sum	[+]_custom_rle_vecavx512 +FastPFor_SIMDOPTPFor <4,Simple16> +VariableByte → [+]_custom_rle_vecavx512 +FastPFor_ SIMDBinaryPacking +VariableByte	217667.8	38317.62	-34.51	24.4
C	Shift	Sum	FastPFor_SIMDBinaryPacking +VariableByte → simdcomp	191440.02	8171.68	-15.97	6.07

Table 3 continued from previous page

D	Shift	Sum	simdcomp → TurboPFor_xor +TurboPack256	187674.36	14520.6	-17.16	10.41
A	Shift	Random Sample	simdcomp → simdcomp	168578.62	9425.49	2.16	7.11
B	Shift	Random Sample	[+]_custom_rle_vecavx512 +FastPFor_SIMDOPTPFor <4,Simple16> +VariableByte → [+]_custom_rle_vecavx512 +FastPFor_ SIMDBinaryPacking +VariableByte	181140.4	36662.55	-9.82	23.25
C	Shift	Random Sample	FastPFor_SIMDBinaryPacking +VariableByte → simdcomp	179483.42	9339.28	-9.26	6.61
D	Shift	Random Sample	simdcomp → custom_delta_vecavx512	184435.1	8521.9	3.79	6.03
A	Classification	Sum	custom_for_vecavx512 → simdcomp	361298.62	99417.64	-1.95	43.01
B	Classification	Sum	custom_rle_vecavx512 → simdcomp	245763.63	22096.69	-6.41	11.22
C	Classification	Sum	FastPFor_SIMDPFor +VariableByte → simdcomp	487978.12	36965.07	-0.22	9.25
D	Classification	Sum	simdcomp → simdcomp	296488.13	90777.87	-1.99	44.42
A	Classification	Random Sample	custom_for_vecavx512 → simdcomp	346419.22	99849.09	8.76	37.86
B	Classification	Random Sample	custom_rle_vecavx512 → simdcomp	243855.94	21208.05	14.17	8.51
C	Classification	Random Sample	FastPFor_SIMDPFor +VariableByte → simdcomp	476831.42	35316.28	1.95	8.72
D	Classification	Random Sample	simdcomp → simdcomp	295705.26	90691.57	6.87	38.81
A	Threshold	Sum	simdcomp → TurboPFor_xor +TurboPack256	187998.49	14381.7	-10.73	10.15
B	Threshold	Sum	[+]_custom_rle_vecavx512 +FastPFor_SIMDFastPFor256 +VariableByte → [+]_custom_rle_vecavx512 +FastPFor_ SIMDBinaryPacking +VariableByte	191631.24	34071.35	-32.22	25.32
C	Threshold	Sum	FastPFor_SIMDPFor +VariableByte → custom_delta_vecavx512	171914.04	10697.18	-2.6	9.76
D	Threshold	Sum	simdcomp → custom_rle_vecavx512	190231.3	17969.51	-10.55	11
A	Threshold	Random Sample	simdcomp → custom_delta_vecavx512	189971.26	8860.78	3.47	5.41

Table 3 continued from previous page

B	Threshold	Random Sample	[+]_custom_rle_vecavx512 +FastPFor_SIMDFastPFor256 +VariableByte → [+]_custom_rle_vecavx512 +FastPFor_ SIMDBinaryPacking +VariableByte	189095.34	35497.17	-8.13	21.74
C	Threshold	Random Sample	FastPFor_SIMDPFor +VariableByte → custom_delta_vecavx512	186615.34	12884.23	8.26	8.94
D	Threshold	Random Sample	simdcomp → [+]_custom_rle_vecavx512 +FastPFor_SIMDSimplePFor +VariableByte	182728.69	20189.1	-0.01	12.49
A	None	Sum	FastPFor_SIMDBinaryPacking +VariableByte	135269.7	9135.9	-9.53	11.09
B	None	Sum	[+]_custom_rle_vecavx512 +simdcomp	121983.5	15718.81	4.51	13.2
C	None	Sum	FastPFor_SIMDBinaryPacking +VariableByte	106351.21	2582.89	-7.55	4.81
D	None	Sum	TurboPFor_xor +TurboPack256	109854.79	5726.49	-6.25	8.06
A	None	Random Sample	FastPFor_SIMDBinaryPacking +VariableByte	106447.8	2882.63	33.74	2.74
B	None	Random Sample	[+]_custom_rle_vecavx512 +FastPFor_ SIMDBinaryPacking +VariableByte	110074.5	15039.82	27.33	10.17
C	None	Random Sample	FastPFor_SIMDBinaryPacking +VariableByte	131806.7	7126.89	25.65	10.07
D	None	Random Sample	FastPFor_SIMDPFor +VariableByte	106031.1	3171.76	29.32	3.29

Table 3: Pipeline execution times and speedups with their best-found non-fused codecs.

3.2 With Fusing Sum into Decompression

File	**Transform**	**Aggregation**	**Best Codec**	**Execution Time (ns)**		**Speedup (%)**	
				μ	σ	μ	σ
A	Smooth	Sum	simdcomp → simdcomp	133655.60	21211.18	44.10	9.37
B	Smooth	Sum	custom_rle_vecavx512 → simdcomp	131990.58	26823.15	43.42	11.76
C	Smooth	Sum	FastPFor_SIMDPFor +VariableByte → simdcomp	150690.17	13917.15	33.94	7.22
D	Smooth	Sum	simdcomp → simdcomp	115062.55	16433.14	51.88	7.25
A	Shift	Sum	simdcomp → simdcomp	78775.86	5738.93	45.66	5.68

Table 4 continued from previous page

B	Shift	Sum	[+]_custom_rle_vecavx512 +FastPFor _SIMDOPTPFor<4,Simple16> +VariableByte → [+]_custom_rle_vecavx512 +FastPFor _SIMDBinaryPacking +VariableByte	73083.39	35413.44	48.81	25.01
C	Shift	Sum	FastPFor_SIMDBinaryPacking +VariableByte → simdcomp	79762.55	8162.79	50.42	5.35
D	Shift	Sum	simdcomp → simdcomp	76416.25	5503.01	43.89	5.37
A	Classification	Sum	custom_for_vecavx512 → simdcomp	236028.94	100244.71	28.75	38.86
B	Classification	Sum	custom_rle_vecavx512 → simdcomp	137139.70	19287.21	40.62	8.97
C	Classification	Sum	FastPFor_SIMDPFor +VariableByte → simdcomp	372762.59	35071.46	18.83	8.82
D	Classification	Sum	simdcomp → simdcomp	181146.78	91237.99	37.69	36.84
A	Threshold	Sum	simdcomp → [+]_custom_rle_vecavx512 +FastPFor _FastBinaryPacking32 +VariableByte	74735.45	23851.91	49.57	16.20
B	Threshold	Sum	[+]_custom_rle_vecavx512 +FastPFor_ SIMDBinaryPacking +VariableByte → [+]_custom_rle_vecavx512 +FastPFor_ FastBinaryPacking16 +VariableByte	83040.39	34136.67	41.64	24.16
C	Threshold	Sum	FastPFor_SIMDBinaryPacking +VariableByte → simdcomp	78115.16	3305.62	53.38	3.89
D	Threshold	Sum	simdcomp → [+]_custom_rle_vecavx512 +FastPFor _FastBinaryPacking16 +VariableByte	73508.25	18582.53	54.22	11.75
A	None	Sum	FastPFor_SIMDPFor +VariableByte	24154.81	14000.08	79.07	12.18
B	None	Sum	simdcomp	25472.21	14317.96	79.29	11.68
C	None	Sum	FastPFor_SIMDPFor +VariableByte	35399.73	2915.09	68.37	3.46
D	None	Sum	simdcomp	15849.13	13379.43	88.22	9.95

Table 4: Pipeline execution times and speedups with their best-found fused codecs.

D Codec Rankings

4.1 Rankings Without Fusing Aggregations

ID	Codec
1	FastPFor_SIMDBinaryPacking+VariableByte
2	FastPFor_SIMDPFor+VariableByte
3	[+]_custom_rle_vecavx512+simdcomp
4	TurboPFor_xor+TurboPack256
5	simdcomp
6	TurboPFor_Zigzag+TurboPack256
7	custom_rle_vecavx512
8	FastPFor_SIMDFastPFor256+VariableByte
9	TurboPFor_TurboPFor256
10	[+]_custom_rle_vecavx512+FastPFor_SIMDBinaryPacking+VariableByte
11	[+]_custom_rle_vecavx512+FastPFor_VarIntG8IU
12	TurboPFor_Zigzag+TurboPFor256
13	[+]_custom_rle_vecavx512+FastPFor_FastBinaryPacking32+VariableByte
14	[+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor256+VariableByte
15	[+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor128+VariableByte
16	TurboPFor_Zigzag+TurboVSimple
17	FastPFor_FastPFor256+VariableByte
18	custom_for_vecavx512
19	[+]_custom_rle_vecavx512+Zstd_5
20	[+]_custom_rle_vecavx512+DEFLATE
21	custom_delta_vecavx512

Table 5: Codec names for Figure 12.

ID	Codec
1	FastPFor_SIMDBinaryPacking+VariableByte
2	TurboPFor_TurboPFor256
3	simdcomp
4	TurboPFor_Zigzag+TurboPack256
5	FastPFor_SIMDPFor+VariableByte
6	FastPFor_SIMDFastPFor256+VariableByte
7	[+]_custom_rle_vecavx512+FastPFor_SIMDBinaryPacking+VariableByte
8	TurboPFor_xor+TurboPack256
9	TurboPFor_Zigzag+TurboPFor256
10	custom_for_vecavx512
11	custom_delta_vecavx512
12	custom_rle_vecavx512
13	[+]_custom_rle_vecavx512+FastPFor_VarIntG8IU
14	[+]_custom_rle_vecavx512+simdcomp
15	TurboPFor_Zigzag+TurboVSimple
16	FastPFor_FastPFor256+VariableByte
17	[+]_custom_rle_vecavx512+FastPFor_FastBinaryPacking32+VariableByte
18	[+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor256+VariableByte
19	[+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor128+VariableByte
20	[+]_custom_rle_vecavx512+Zstd_5
21	[+]_custom_rle_vecavx512+DEFLATE

Table 6: Codec names for Figure 13.

ID	Codecs
1	simdcomp → simdcomp

Table 7 continued from previous page

2	custom_rle_vecavx512 → simdcomp
3	simdcomp → custom_delta_vecavx512
4	custom_for_vecavx512 → simdcomp
5	FastPFor_SIMDPFor+VariableByte → simdcomp
6	FastPFor_SIMDPFor+VariableByte → custom_delta_vecavx512
7	simdcomp → [+]_custom_rle_vecavx512+FastPFor_SIMDSimplePFor+VariableByte
8	simdcomp → custom_rle_vecavx512
9	FastPFor_SIMDBinaryPacking+VariableByte → simdcomp
10	simdcomp → TurboPFor_xor+TurboPack256
11	[+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor256+VariableByte → [+]_custom_rle_vecavx512+FastPFor_SIMDBinaryPacking+VariableByte
12	[+]_custom_rle_vecavx512+FastPFor_SIMDOPTPFor<4,Simple16>+VariableByte → [+]_custom_rle_vecavx512+FastPFor_SIMDBinaryPacking+VariableByte

Table 7: Codec names for Figure 14.

ID	Codecs
1	simdcomp → simdcomp
2	custom_rle_vecavx512 → simdcomp
3	custom_for_vecavx512 → simdcomp
4	FastPFor_SIMDPFor+VariableByte → simdcomp
5	FastPFor_SIMDPFor+VariableByte → custom_delta_vecavx512
6	simdcomp → custom_delta_vecavx512
7	simdcomp → [+]_custom_rle_vecavx512+FastPFor_SIMDSimplePFor+VariableByte
8	[+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor256+VariableByte → [+]_custom_rle_vecavx512+FastPFor_SIMDBinaryPacking+VariableByte
9	FastPFor_SIMDBinaryPacking+VariableByte → simdcomp
10	[+]_custom_rle_vecavx512+FastPFor_SIMDOPTPFor<4,Simple16>+VariableByte → [+]_custom_rle_vecavx512+FastPFor_SIMDBinaryPacking+VariableByte
11	simdcomp → custom_rle_vecavx512
12	simdcomp → TurboPFor_xor+TurboPack256

Table 8: Codec names for Figure 15.

4.2 Projected Rankings after Fusing Aggregation

ID	Codec
1	custom_delta_vecavx512
2	FastPFor_SIMDBinaryPacking+VariableByte
3	simdcomp
4	FastPFor_SIMDFastPFor256+VariableByte
5	custom_for_vecavx512
6	TurboPFor_xor+TurboPack256
7	TurboPFor_Zigzag+TurboPack256
8	FastPFor_SIMDPFor+VariableByte
9	TurboPFor_Zigzag+TurboPFor256
10	TurboPFor_TurboPFor256
11	custom_rle_vecavx512
12	[+]_custom_rle_vecavx512+simdcomp
13	[+]_custom_rle_vecavx512+FastPFor_SIMDBinaryPacking+VariableByte
14	[+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor256+VariableByte
15	[+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor128+VariableByte

Table 9 continued from previous page

16	[+]_custom_rle_vecavx512+FastPFor_VarIntG8IU
17	[+]_custom_rle_vecavx512+FastPFor_FastBinaryPacking32+VariableByte
18	[+]_custom_rle_vecavx512+Zstd_5
19	FastPFor_FastPFor256+VariableByte
20	TurboPFor_Zigzag+TurboVSimple
21	[+]_custom_rle_vecavx512+DEFLATE

Table 9: Codec names for Figure 16.

ID	Codec
1	TurboPFor_xor+TurboPack256
2	TurboPFor_Zigzag+TurboPack256
3	FastPFor_SIMDBinaryPacking+VariableByte
4	simdcomp
5	FastPFor_SIMDPFor+VariableByte
6	TurboPFor_TurboPFor256
7	FastPFor_SIMDFastPFor256+VariableByte
8	TurboPFor_Zigzag+TurboPFor256
9	custom_rle_vecavx512
10	[+]_custom_rle_vecavx512+simdcomp
11	[+]_custom_rle_vecavx512+FastPFor_SIMDBinaryPacking+VariableByte
12	[+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor256+VariableByte
13	[+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor128+VariableByte
14	[+]_custom_rle_vecavx512+FastPFor_VarIntG8IU
15	[+]_custom_rle_vecavx512+FastPFor_FastBinaryPacking32+VariableByte
16	custom_for_vecavx512
17	custom_delta_vecavx512
18	[+]_custom_rle_vecavx512+Zstd_5
19	FastPFor_FastPFor256+VariableByte
20	TurboPFor_Zigzag+TurboVSimple
21	[+]_custom_rle_vecavx512+DEFLATE

Table 10: Codec names for Figure 17.

ID	Codecs
1	simdcomp → simdcomp
2	FastPFor_SIMDBinaryPacking+VariableByte → simdcomp
3	simdcomp → custom_rle_vecavx512
4	custom_rle_vecavx512 → simdcomp
5	FastPFor_SIMDPFor+VariableByte → simdcomp
6	[+]_custom_rle_vecavx512+FastPFor_SIMDOPTPFor<4,Simple16>+VariableByte → [+]_custom_rle_vecavx512+FastPFor_SIMDBinaryPacking+VariableByte
7	[+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor256+VariableByte → [+]_custom_rle_vecavx512+FastPFor_SIMDBinaryPacking+VariableByte
8	custom_for_vecavx512 → simdcomp

Table 11: Codec names for Figure 18.

ID	Codecs
1	FastPFor_SIMDBinaryPacking+VariableByte → simdcomp
2	simdcomp → simdcomp

Table 12 continued from previous page

3	simdcomp → custom_rle_vecavx512
4	custom_rle_vecavx512 → simdcomp
5	FastPFor_SIMDPFor+VariableByte → simdcomp
6	[+]_custom_rle_vecavx512+FastPFor_SIMDOPTPFor<4,Simple16>+VariableByte → [+]_custom_rle_vecavx512+FastPFor_SIMDBinaryPacking+VariableByte
7	[+]_custom_rle_vecavx512+FastPFor_SIMDFastPFor256+VariableByte → [+]_custom_rle_vecavx512+FastPFor_SIMDBinaryPacking+VariableByte
8	custom_for_vecavx512 → simdcomp

Table 12: Codec names for Figure 19.