Efficient Lossless Compression of Scientific Floating-Point Data on CPUs and GPUs

Noushin Azami
Texas State University
Department of Computer Science
San Marcos, Texas, USA
noushin.azami@txstate.edu

Alex Fallin
Texas State University
Department of Computer Science
San Marcos, Texas, USA
waf13@txstate.edu

Martin Burtscher
Texas State University
Department of Computer Science
San Marcos, Texas, USA
burtscher@txstate.edu

Abstract

The amount of scientific data being produced, transferred, and processed increases rapidly. Whereas GPUs have made faster processing possible, storage limitations and slow data transfers remain key bottlenecks. Data compression can help, but only if it does not create a new bottleneck. This paper presents four new lossless compression algorithms for single- and double-precision data that compress well and are fast even though they are fully compatible between CPUs and GPUs. Averaged over many SDRBench inputs, our implementations outperform most of the 18 compressors from the literature we compare to in compression ratio, compression throughput, and decompression throughput. Moreover, they outperform all of them in either throughput or compression ratio on the two CPUs and two GPUs we used for evaluation. For example, on an RTX 4090 GPU, our fastest code compresses and decompresses at over 500 GB/s while delivering one of the highest compression ratios.

CCS Concepts: **• Information systems → Compression strategies**; **• Computing methodologies → Massively parallel algorithms**.

Keywords: Data Compression; Lossless Compression; Floating-point Data; CPU and GPU Parallelization

ACM Reference Format:
Noushin Azami, Alex Fallin, and Martin Burtscher. 2025. Efficient Lossless Compression of Scientific Floating-Point Data on CPUs and GPUs. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3669940.3707280

1 Introduction

The amount of data being produced, transferred, and stored by scientific simulations (e.g., climate and cosmology) and instruments (e.g., particle accelerators and coherent light sources) is increasing rapidly and already often burdens or exceeds the available I/O bandwidth and storage capacity. For example, it is estimated that the Velociprobe beam line at APS will produce over 150 PB of raw data per year [15]. Data compression can reduce the needed amount of storage and its associated cost, but only if the compression ratio is sufficiently high. The data acquisition rate of the LCLS-II [24] light source will reach up to 250 GB/s [10]. Again, data compression can help, but only if the compression throughput is high enough for real-time operation. Interconnection speeds are also getting faster. For example, the most recent version of NVLink can achieve throughputs of up to 900 GB/s, and the latest PCIe bus specification supports up to 242 GB/s. Only few compressors can keep up with these throughputs, especially since they must operate at X times higher speeds, where X is the compression ratio, before the interconnect becomes the bottleneck.

Data can be compressed losslessly or lossily. Lossless compression yields lower compression ratios but recreates the original data exactly during decompression. It is indispensable in domains where preserving precision and accuracy is critical, such as in certain computational-physics, climate-modeling, and fluid-dynamics simulations, where lossy compression could introduce errors that affect the validity of the scientific findings. Other examples include engineering applications, where lossy compression might compromise the integrity of the designs, financial modeling and analysis, where lossy compression could lead to monetary losses, and medical imaging, where lossy compression might introduce artifacts that compromise diagnostic accuracy. In these and other domains, lossless compression is unavoidable for efficient storage, transmission, and processing of data.

Since scientific data is often generated and compressed on one system and decompressed and analyzed on another, it is important to support compatible compression and decompression across CPUs and GPUs. Yet, most current compressors only support either CPUs or GPUs. To make things worse, almost all compressors that compress well only deliver low speeds, and almost all compressors that compress and decompress quickly only deliver low compression ratios. Hence, the key challenge in lossless floating-point compression is *achieving a high compression ratio and a high speed at the same time on both CPUs and GPUs*.

To this end, we created four new lossless compression algorithms that we specifically designed to compress scientific floating-point data well and fast on CPUs and GPUs. We call them SPratio, SPspeed, DPratio, and DPspeed. The two "SP" algorithms target single-precision data whereas the two "DP" algorithms target double-precision data. They support two modes: the "ratio" mode focuses more on compression ratio, and the "speed" mode focuses more on throughput. Note that all 4 algorithms deliver a higher speed and a higher compression ratio than most of the prior work. The two modes simply give users a choice to obtain even better results.

We designed these algorithms by experimenting with a large number of combinations of data transformations until we found some that met our needs. We only considered transformations that we could efficiently implement on CPUs and GPUs to ensure a high throughput. Then we *enhanced* some of the transformations to boost the compression ratio. This approach led to the SPspeed, SPratio, and DPspeed algorithms. As it did not yield a satisfactory solution for DPratio, we created *novel* transformations specifically for the DPratio algorithm that deliver record compression ratios on GPUs while still running relatively quickly.

This paper makes the following main contributions.

- It introduces 4 new lossless compression algorithms that yield high compression ratios on single- and double-precision floating-point data.
- It describes compatible parallel CPU and GPU implementations of each algorithm that deliver high-throughput compression and decompression.
- It presents an extensive performance comparison of 19 compressors on 90 single- and 20 double-precision inputs from different scientific domains.
- It describes innovative new parallelizable data transformations and enhancements to previously known transformations that boost the compression ratio.

Our CPU and GPU compressors will be open-sourced on GitHub once the anonymization phase is over.

The rest of this paper is organized as follows. Section 2 summarizes the related work from which we draw ideas and to which we compare our algorithms. Section 3 explains the design, operation, and parallelization of our implementations. Section 4 details the experimental methodology. Section 5 presents, analyzes, and compares the compression ratios and speeds. Section 6 summarizes our work.

2 Related Work

Since we target lossless compression of floating-point data, we focus this section on such compressors. Some are special-purpose compressors designed for floating-point values. Others are general-purpose compressors designed to achieve high compression ratios across different data types. We compare our approach to compressors from both categories. We also compare to both CPU and GPU compressors. While the underlying algorithms are often compatible with both a CPU and GPU, we only use the implementations provided by the authors. Additionally, algorithms designed for the CPU are often much less performant when implemented on the GPU, and vice-versa.

2.1 Lossless floating-point compressors

FPzip [26] is a library that supports both lossy and lossless compression of scientific data. It exploits floating-point data coherency to predict values in the input, computes the residuals, stores the data as integers, and uses a fast entropy encoder to achieve not only high compression ratios but also fast compression and decompression.

ZFP [25] is designed for compressing multi-dimensional arrays of floating-point or integer values, supporting random-access reads and writes in constant time. Users can operate on these arrays normally since the compression is transparent. This is achieved through a software-defined cache where each value is compressed before storing it and decompressed upon loading it. ZFP exploits spatial correlation for effective compression. It is implemented on both CPUs and GPUs, but only the CPU version includes lossless compression.

FPC [8] is a CPU-based lossless compressor for double-precision data. It uses two hash tables to predict later values based on earlier values, selects the more accurate prediction, computes the difference between the predicted and the actual value, and compresses the result by replacing any leading zero bytes by a 3-bit value representing their count. A 1-bit value is also emitted to specify which of the two predictions was used. pFPC [9] is the parallel version of FPC. It is implemented using Pthreads and employs the same compression approach as FPC except it chunks up the data and applies the FPC algorithm in parallel to each chunk.

SPDP [11] is another CPU-based lossless compressor. It supports both single- and double-precision floating-point data. This compressor performs difference coding, byte shuffling, and Lempel-Ziv (LZ) coding [23]. Our algorithms do not utilize LZ because LZ is difficult to parallelize efficiently, especially for GPUs. However, we also make use of difference coding and byte shuffling. There is no GPU implementation of SPDP.

GFC [30] is a GPU-based lossless compressor for double-precision floating-point data. It computes the difference sequence, negates any negative differences, and encodes the sign bit together with a 3-bit count of the leading zero bytes in a nibble before removing those leading zero bytes. GFC compresses chunks of data in parallel. To achieve additional parallelism, the difference sequence is computed using values that appear at least 32 elements earlier in the input.

MPC [37] is a GPU algorithm for losslessly compressing single- and double-precision data. It is based on data transformations from other compression algorithms that are chained together and parallelized. MPC uses delta encoding and bit transposition (shuffling) to create many zero values, which are recorded in a bitmap and then eliminated from the value sequence. There is no CPU implementation of MPC.

nvCOMP [2] is a CUDA library that contains several parallel lossless compressors. Some of these compressors are specifically designed for compressing floating-point data, including Bitcomp, a novel algorithm designed by NVIDIA, and Asymmetric Numeral Systems (ANS) compression, an entropy coder that encodes symbols from the input data stream into a stream of bits using a reversible mapping [14].

Ndzip [21] is a lossless floating-point compressor for both CPUs and GPUs. It is based on high-throughput data transformations such as Lorenzo coding [19] and bit transposition. Ndzip supports both single- and double-precision data. Like our compressors, it is available in serial C++ code, parallel OpenMP code for CPUs, and parallel CUDA code for GPUs. NDzip is the only tested compressor other than ours that offers CPU and GPU compatibility. However, it requires the user to provide the dimensionality of the input data, which our algorithms do not need.

2.2 Lossless general-purpose compressors

Gzip [1] is a lossless CPU compressor. It is based on LZ77 and Huffman coding. Bzip2 [32] is another lossless general-purpose compressor for CPUs that tends to compress better than Gzip but is slower. It is based on the Burrows-Wheeler Transform (BWT), run-length encoding (RLE) [27], and Huffman coding.

As mentioned, we compare our implementations to compressors from the nvCOMP library, some of which are general purpose, including Cascaded, which uses RLE, delta encoding, and bit packing [36]. nvCOMP also contains other general-purpose compressors such as Deflate [13], which is a combination of Huffman and LZ77, and Gdeflate, which is a novel algorithm based on Deflate with more efficient GPU decompression. It further contains SNAPPY, a compressor similar to LZ4.

We also evaluate Zstandard [12], a parallel CPU compressor that is based on LZ77 [23], ANS, and Huffman [18] coding. Furthermore, we compare our GPU compressor to the GPU implementation of Zstandard in nvCOMP.

2.3 Lossless graph compression

Compression of in-memory graph data structures has been studied as a way to improve the performance of graph algorithms [5, 33]. MPLG, which is a GPU implementation of this idea, is fast enough to make real-time decompression possible [3]. It consists of a lossless compression algorithm for eliminating leading zero bits, making it a fast entropy coder that is GPU friendly. We mention it here because we use a modified version of MPLG as the final data transformation in some of our algorithms.

3 Approach

Our SPspeed and SPratio algorithms process single-precision floating-point values as 32-bit integers, and our DPspeed and DPratio algorithms process double-precision floating-point values as 64-bit integers to guarantee lossless operation. Note that they do not convert (e.g., round) the floating-point values to integers. Instead, they treat the IEEE 754 floating-point word as an integer word, i.e., they load the values bit-for-bit into an integer variable and then process the data using integer operations only as is done by many other lossless floating-point compressors [7, 16, 17, 35].

Figure 1 lists the stages of our four new compression algorithms. For decompression, the inverses of the stages are invoked in reverse order. The following subsections explain the data transformations performed by each stage as well as how they are implemented and parallelized.

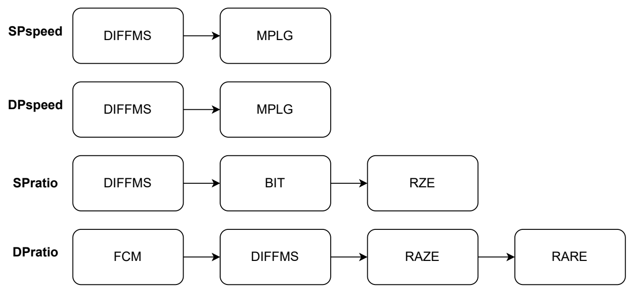

Figure 1. The stages (transformations) of our 4 algorithms

We designed these algorithms with the help of the LC framework [4], which can automatically synthesize data compressors. We used it to generate over 100,000 algorithms, the best of which we then analyzed. This analysis led to the creation of the DIFFMS, RZE, FCM, RARE, and RAZE transformations described below to boost the compression ratio while maintaining a high throughput.

Our four algorithms target scientific data that is relatively smooth. The values can be positive, negative, or a mix of positive and negative. For smooth data, that is, inputs where the differences between consecutive values are small in magnitude, the DIFFMS and FCM stages transform the input into a sequence of small positive values with many leading zeros. The purpose of the remaining stages is to maximally compress such sequences. Hence, we do not expect our algorithms to compress non-smooth data particularly well.

However, the wide range of scientific inputs we use for evaluation (see Section 4) tend to be quite smooth, normal, and centered around zero [38].

Except for FCM, all stages in all four algorithms operate on chunks of 16 kilobytes. We choose this size so that we can fit two chunk buffers in the GPU's shared memory and the CPU's L1 data cache. Each chunk is independent and can be compressed/decompressed in parallel. To cap the worst-case expansion, the compressor emits the original data for any chunk that it cannot compress and marks it as such. On the CPU, we dynamically assign the chunks to the threads to maximize the load balance. On the GPU, we dynamically assign the chunks to the thread blocks, which not only balances the load but also enables the use of fast shared memory (a software-controlled L1 data cache) and allows for further parallelization within a thread block.

3.1 SPspeed and DPspeed

Since SPspeed and DPspeed target a high compression and decompression throughput, the underlying algorithms consists of just two fast data transformations.

The DIFFMS transformation computes the difference (modulo 2^{32} in SPspeed and 2^{64} in DPspeed) between each integer value and the preceding value in the input and records the result in magnitude-sign format. Scientific datasets often contain values within a narrow range of each other, meaning their exponents are similar, which are stored in some of the most-significant bits. Computing the integer difference turns these exponents into values that cluster around zero. Since the differences can be positive or negative, the resulting values might contain many leading '0' bits or many leading '1' bits. This is why we change the format from two's-complement to magnitude-sign representation, which converts values with leading '0' bits and values with leading '1' bits into values with only leading zeros.

Figure 2 illustrates this transformation on the example of three 32-bit single-precision values. The most significant bit of each input value is the sign bit and the next 8 bits denote the exponent. The first element in a chunk is preserved as-is (i.e., as if 0 was its preceding value). Recall that we treat each value as a 32-bit integer. As outlined in Figure 2, the first step of this transformation produces a positive and two negative values, the latter of which contain many leading '1' bits. The second step changes the leading ones to zeros while retaining the leading zeros of the first value as shown at the bottom of Figure 2. This is accomplished with the following reversible transformation: $(data << 1)\hat{}(data >> 31)$, where the right shift is a signed shift that replicates the sign bit. Note that the sign bit is stored in the least significant position. DPspeed works similarly but on 64-bit values.

The goal of the first stage is to create values that hopefully contain many leading '0' bits. This enables us to use a modified version of MPLG [3], a high-speed lossless algorithm for eliminating leading zeros, as the second and final transformation in SPspeed and DPspeed. MPLG first finds the maximum value in each data chunk, counts the number of leading '0' bits in the maximum, and then eliminates that many bits from all values in the chunk. To improve the compression ratio, we enhanced MPLG as follows. If the maximum has no leading zeros, which renders MPLG ineffective, we apply another two's-complement to magnitude-sign conversion to the values in the chunk. Note that this conversion is meaningless in the sense that the data is no longer in two's-complement format. We simply use the conversion because it is a fast and reversible transformation that often manages to produce a few leading zeros where there were none before, thus boosting the effectiveness of MPLG. Again, this is done at 32-bit granularity in SPspeed and at 64-bit granularity in DPspeed.

Figure 3 illustrates the operation of MPLG on three 32-bit values.The first value is the maximum. Since it has 12 leading zeros, MPLG eliminates the 12 leading bits from each value, which are highlighted in red. The resulting 20-bit values are then concatenated, as shown at the bottom of Figure 3.

There are more leading '0' bits that could be eliminated from the second and third values. However, MPLG keeps the number of eliminated bits fixed to make independent and parallel decompression of each value possible. As a partial remedy, we divide each 16 kB data chunk into 32 512-byte subchunks. This enables us to efficiently process each subchunk by a warp and improves the compression ratio by allowing the MPLG stage to use a different number of leading zeros for each subchunk.

SPspeed and DPspeed parallelization: The OpenMP CPU implementation of SP/DPspeed is parallelized across the data chunks. The encoder works as follows. Each running thread request the next available chunk from the worklist, performs the two transformations on it, outputs the compressed size, busy-waits for the write position from the thread processing the prior chunk, adds the compressed size to this position, sends the result to the thread processing the next chunk, and then writes the compressed output to the received write position. The decoder works as follows. It first computes the prefix sum over the compressed chunk sizes, yielding the aforementioned write positions (which are now read positions). Then, each thread independently processes a compressed chunk at a time, running the inverse of the two transformations in the opposite order to recreate the original data. No write positions need to be communicated as the decompressed chunk sizes are known a priori.

The CUDA GPU implementations of SP/DPspeed are parallelized in the same way except the chunks are assigned to the thread blocks rather than the individual threads. We use Merrill and Garland's variable look-back strategy [28] to quickly communicate the write position to the next thread block. To also exploit parallelism within each thread block, we further parallelized the two stages as follows. In the encoder, the DIFFMS transformation is embarrassingly parallel.

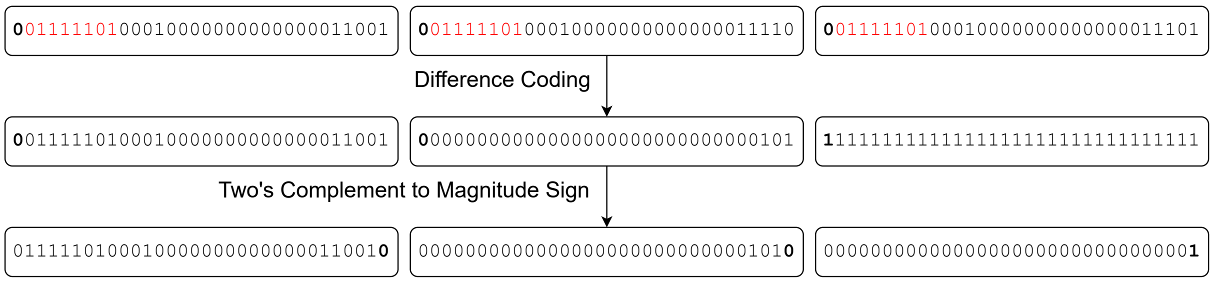

Figure 2. DIFFMS's difference coding and two's-complement to magnitude-sign conversion (actual magnitude is the sum of the represented magnitude and the sign); the sign bits are shown in bold print, the initial exponent bits are highlighted in red

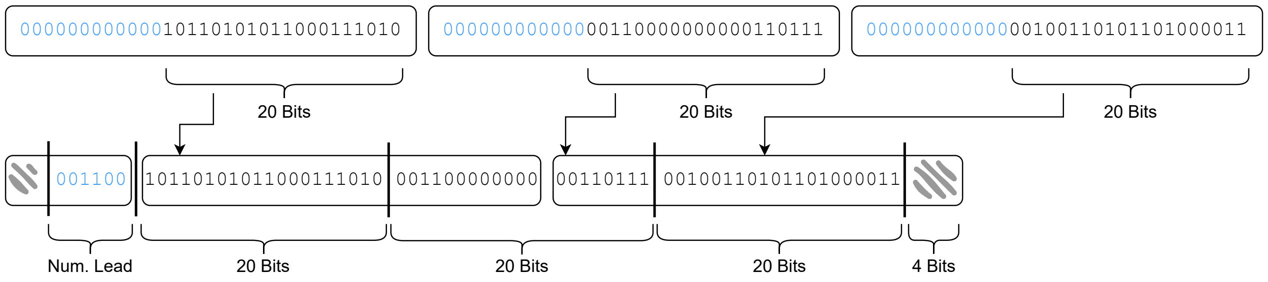

Figure 3. MPLG elimination of common leading zero bits

For the second transformation, we use an augmented version of the MPLG code [3]. In the decoder, we also use a modified version of the existing MPLG code. The difference decoding needed in the DIFFMS stage is implemented using a block-level parallel prefix sum that utilizes warp-level primitives and shared memory to achieve a high throughput. Note that both the encoder and the decoder keep all chunk data in shared memory between transformations to minimize accesses to the relatively slow main memory.

3.2 SPratio and DPratio

The SP/DPratio algorithms target a high compression ratio. To achieve this, they use more and slower stages that, taken together, generally compress better. Unlike the SP/DPspeed algorithms discussed above, which implement the same transformation but at different granularities, SPratio employs a different algorithm than DPratio.

SPratio: SPratio starts with DIFFMS, that is, the same transformation as SPspeed. It computes the difference between consecutive values and records them in magnitude-sign format. However, SPratio replaces the MPLG stage by two different stages to boost the compression ratio.

SPratio's second transformation, called BIT, performs a bit transposition (or bit shuffle). It groups the first bit of every value together, then all the second bits, and so on. Figure 4 illustrates this process on the example of three 32-bit values. Note that it continues the example from Figure 2.

As discussed, the DIFFMS transformation aims at producing values with many leading zero bits. Transposing the bits of these values places the most significant bits next to each other, often producing long runs of zero values, which are typically followed by gradually more random values that stem from the less significant bits.

We designed the third and last transformation in SPratio, which we call *Repeated Zero Elimination* (RZE), to compress such sequences well. RZE generates a bitmap in which each bit corresponds to a byte in the input. A cleared bit indicates that the corresponding byte is zero. Otherwise, the bit is set. All zero bytes are then removed from the input. Hence, the compressed output consists of the bitmap and the non-zero bytes from the input. The number of non-zero bytes depends on the data, but the size of the bitmap is fixed and represents a significant overhead.

Fortunately, the bitmap tends to be quite compressible as it typically starts with mostly '0' bits and ends with mostly '1' bits. Hence, we enhanced the transformation by employing a similar algorithm to repeatedly compress the bitmap. The only difference is that this algorithm identifies repeating bytes rather than zero bytes. In this way, the original bitmap of 16384 bits is reduced to 2048, then 256, and ultimately 32 bits. Only the final 32 bits and the non-repeating bytes from the larger bitmaps are emitted. Compressing the bitmap in this manner often substantially boosts the compression ratio.

Figure 5 illustrates how RZE works using the result from Figure 4 as input. Since RZE operates at byte granularity (to increase the chance of finding zero values), the middle portion of Figure 5 shows the same data as the top but separated into bytes. The numbers above the bytes indicate their index.

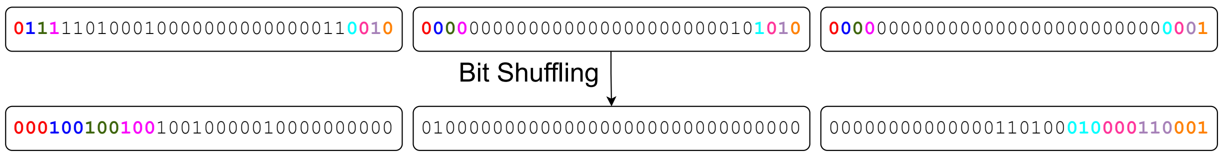

Figure 4. Bit transposition (shuffling) in the BIT stage; for inputs with more values, the resulting sequences of bits with the same color will be correspondingly longer

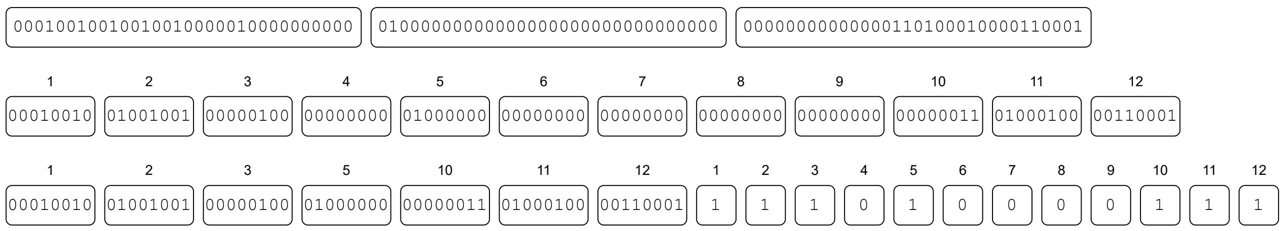

Figure 5. Repeated zero elimination (the repeated bitmap compression is not shown as it is ineffective on this small example)

RZE sets the bits in the bitmap according to whether the corresponding byte with the same index is zero. It then outputs all non-zero bytes followed by the compressed bitmap.

SPratio parallelization: Our OpenMP and CUDA implementations of SPratio are parallelized as described above for SPspeed. The only difference is how we internally parallelized the two new transformations for the GPU.

The BIT stage is parallelized in both the encoder and the decoder by grouping 32 values together to match the 32 bits in each value. Each group is assigned to a warp, which conveniently consists of 32 threads. We take advantage of fast CUDA shuffle operations to exchange data between the threads in a warp (without accessing memory) to implement the bit transposition in $log_2(32) = 5$ steps. The 64-bit version operates similarly but assigns 2 values to each warp thread.

The RZE encoder assigns multiples of 8 consecutive bytes to each thread. The threads then check if the bytes are zero and set the needed bits in the bitmap. For performance reasons, they do this 8 bits at a time. Moreover, each thread counts the number of non-zero bytes assigned to it. Then, all threads compute a block-wide parallel prefix sum on these counts. Finally, they output their non-zero bytes at the location determined by the prefix sum. Similar steps are executed repeatedly to compress the bitmap.

The RZE decoder operates analogously. It first decompresses the bitmap. Then, it assigns multiples of 8 consecutive (decompressed) bytes to each thread. The threads first count the number of non-zero bytes based on the bits in the bitmap. Next, they perform a prefix sum to determine the location where the non-zero bytes are stored. Finally, they recreate the original byte values, using a zero value if the bitmap contains a '0'. Otherwise, they obtain the non-zero byte from the compressed input and output it. Similar steps are used to decompress the bitmap.

As is done for the other data transformations, the encoder and decoder keep all data in shared memory within and between transformations to minimize main-memory accesses.

DPratio: DPratio employs a different algorithm than SPratio because using the same algorithm at 64-bit granularity yields unsatisfactory compression ratios. This discrepancy arises due to the generally higher randomness in the mantissa bits of double-precision data. As floating-point values undergo arithmetic operations in, for example, a simulation code, their bits tend to become more random [8], especially those far away from the (floating) binary point. Double-precision numbers, with their larger mantissas, therefore often exhibit more randomness in their least-significant bits. Since random bits cannot be compressed, we developed a new algorithm for compressing double-precision data.

Our evaluation of other compressors for double-precision data showed that FPC [8] delivers high compression ratios without using a complex algorithm. However, it employs two hash tables per thread, which is untenable on GPUs. As an alternative, we devised the related "Finite Context Method" (FCM) transformation for use in the first stage of DPratio.

As outlined in Figure 6, FCM starts by creating an array of pairs, where each pair corresponds to a value from the input. The first element of each pair is a hash of the three prior input values. The second element is simply the index (location) of the input value. Next, FCM sorts these pairs. Finally, it creates two scalar arrays (colored in Figure 6) that are populated as follows based on the sorted pairs.

For each pair, it checks whether one of the preceding four pairs in sorted order has the same hash and refers to the same floating-point value. If it does, we call it a match. Note that, due to the sorting, pairs with the same hash will be next to each other. Moreover, because the second element of each pair is the index, preceding pairs with the same hash always refer to earlier locations in the input. If there is no match,

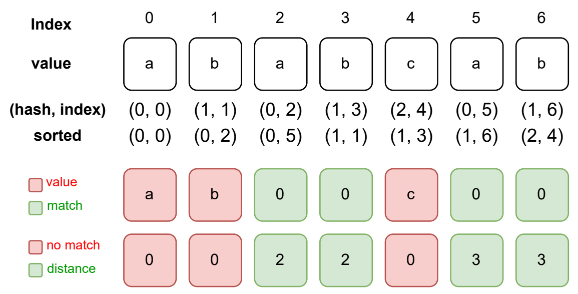

Figure 6. Visual representation of the FCM transformation: pairs with simplified hashes and indices are sorted to identify prior occurrence of the same value

the corresponding value in the first scalar array is set to the double-precision value from the input and the value in the second scalar array is set to zero. In contrast, if a match is found, the value in the first scalar array is set to zero and the value in the second scalar array is set to the distance to the matched value. This can be done independently for each input value (i.e., pair). Note that the two scalar arrays reflect the order of the original input values and not the sorted order of the pairs. The two arrays together require twice the size of the input, that is, this stage doubles the amount of data. However, the two arrays tend to be more compressible than the original data as half of their entries are zero and some of the double values have been converted to integer distances.

The second stage of DPratio is DIFFMS (i.e., it is identical to the first stage of DPspeed). It computes the difference sequence over the two arrays produced by the FCM stage and outputs the result in magnitude-sign format.

The third stage utilizes a new transformation we invented that is based on the aforementioned RZE transformation. However, as the word size of this stage is larger, the bitmap contains fewer bits and only requires 3 iterations to be compressed. Since double-precision values tend to have rather random bits in the least-signification positions, which are incompressible, we created the *Repeated Adaptive Zero Elimination* (RAZE) transformation, where the adaptive part is the key innovation. It treats the upper bits of each double separately from the lower bits and only applies RZE to the top k bits while always keeping the bottom $64 - k$ bits as shown in Figure 7. Importantly, RAZE automatically finds the optimal k value for each chunk, meaning it adapts to the data. This can be done without trying all 64 possibilities. Instead, RAZE creates a histogram of the leading-zero-bit counts of all values in the chunk. Then it computes the prefix sum over the 64 histogram bins. This yields useful counts because every value with m leading zero bits is also a value with $m - 1$, $m - 2$, etc. leading zeros. For each chunk, RAZE computes what the compressed size would be for each of the 64 counts, selects the k that minimizes the size, and then applies the RZE transformation to just the top k bits of each value.

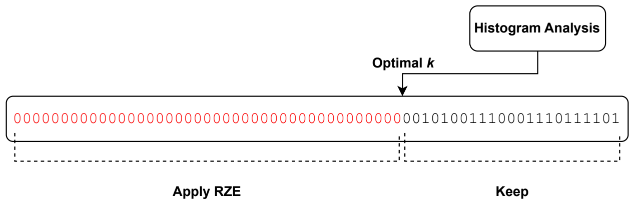

Figure 7. RARE and RAZE: find optimal k and apply RZE transformation to only the top k bits

The fourth stage of DPratio uses our new *Repeated Adaptive Repetition Elimination* (RARE) transformation, which is similar to RAZE except it does not check whether the top k bits are all zero but rather whether they are the same as in the prior value. Otherwise, RARE operates like RAZE. We included this stage to boost the compression ratio as the third stage eliminates zero bits but tends to produce values with identical bit patterns in the most-significant bits.

DPratio parallelization: The FCM encoder is easy to parallelize. All needed array elements can be computed in an embarrassingly parallel fashion. For the sorting, we use the CUB library that is included with CUDA. The decoder is parallelized as follows. For each array element, the corresponding thread reads the distance array at that index. If the distance is non-zero, the thread subtracts the distance from the index and tries again. It iterates until a zero distance is found. Then, the thread uses the resulting index to read from the value array and writes the result to the output array at the original index. Next, it executes a memory fence. Finally, it updates the non-zero distance at the original index to zero if needed. This indicates to the other threads that the current value is available for reading, i.e., their searches can stop at the current position. Whereas this approach sounds like it might be slow, it is actually a parallel implementation of the "find" operation in union-find [34] and, therefore, very fast in practice. In other words, the threads tend to not iterate often because other threads shorten the "chains" continuously.

The second stage is identical to the first stage of DPspeed and, therefore, parallelized in the same way.

The third and fourth stages are parallelized like RZE, i.e., the last stage of SPratio. The main difference is that the compressor first has to create the histogram, which it does in parallel by atomically incrementing the bins. Since we record the resulting k value in the compressed data, the decompressor can just read it and does not need the histogram.

4 Experimental Methodology

We evaluate the performance of ours and the 18 lossless compressors listed in Table 1. Among them, ANS, Bitcomp, FPC, FPzip, GFC, MPC, Ndzip, pFPC, SPDP, and ZFP are specifically designed for compressing floating-point data. In contrast, Bzip2, Cascaded, Deflate, Gdeflate, Gzip, LZ4, Snappy, and Zstandard are general-purpose compressors intended to effectively compress various types of data.

Ndzip stands out as the only compressor, aside from ours, that provides compatibility across CPUs and GPUs. Zstandard also offers implementations for CPUs (in the lzbenchmark [20] suite) and GPUs (in nvCOMP), but they originate from separate sources and are incompatible.

Table 1. Lossless compressors used in comparison

Device	Compressor	Datatype	Version	Source
CPU + GPU	Ndzip	FP32 & FP64	1.0	[21] [22]
	ZSTD	General	2.6	[2] [20]
GPU	ANS	FP32 & FP64	2.6	[2]
	Bitcomp	FP32 & FP64	2.6	[2]
	Cascaded	General	2.6	[2]
	Deflate	General	2.6	[2]
	Gdeflate	General	2.6	[2]
	GFC	FP64	2.2	[30]
	LZ4	General	2.6	[2]
	MPC	FP32 & FP64	1.2	[37]
	Snappy	General	2.6	[2]
CPU	Bzip2	General	1.0.8	[32]
	FPC	FP64	1.1	[8]
	FPzip	FP32 & FP64	1.3	[26]
	Gzip	General	1.1	[1]
	pFPC	FP64	1.0	[9]
	SPDP	FP32 & FP64	1.1	[11]
	ZFP	FP32 & FP64	1.0	[25]

We present results for two systems, both of which run Fedora 37. The first system is based on an AMD Ryzen Threadripper 2950X CPU with 16 hyperthreaded cores and has 48 GB of main memory. The GPU in this system is an NVIDIA GeForce RTX 4090 (Lovelace architecture) with 24 GB of global memory and 16,384 processing elements in 128 streaming multiprocessors (SMs). The second system is based on dual Intel Xeon Gold 6226R CPUs with 16 hyperthreaded cores each and has 64 GB of main memory. The GPU in this system is an NVIDIA A100 (Ampere architecture) with 40 GB of global memory and 6912 processing elements in 108 SMs. The GPU driver version is 525.85.05 on both systems.

We compiled our CPU codes using g++ version 12.2.1 with the $-O3$ $-march = native$ $-fopenmp$ flags. For our GPU codes, we used nvcc version 12.0 with the $-O3$ and $-arch = sm\_80$ flags for the A100 and the $-O3$ and $-arch = sm\_89$ flags for the RTX 4090. For the other compressor, we followed the instructions for compilation provided by the authors but adjusted the compute capabilities to our GPUs.

For the single-precision inputs, we used the Scientific Data Reduction (SDR) benchmark suite [31, 38]. This state-of-the-art suite was designed specifically for data compression research and contains a comprehensive set of real-world datasets from many scientific domains. To keep the experiments reasonable, we omitted some files. For CESM-ATM, we only use the 3D inputs because they contain the same data as the 2D inputs. For EXAALT, we only use the Copper dataset. For Hurricane ISABEL, we only use the raw (i.e., not cleared) data. Additionally, we excluded files that are not compatible with all tested compressors because they are either too large or in a proprietary file format. In total, we tested each code on 90 single-precision files from 7 scientific domains, including climate, molecular-dynamics, and cosmology simulations.

Given the small number of double-precision files in the SDR benchmark suite, we supplement the datasets with 13 additional files [6]. In total, we tested each code on 20 double-precision files from five scientific domains, including instrument data, simulation results, and MPI messages.

We compute the geometric-mean compression ratio, the geometric-mean compression throughput, and the geometric-mean decompression throughput for each of those 7 single-precision and 5 double-precision datasets and report the geometric-mean of all geometric-means for each compressor. We do this so as not to over-weigh the datasets that contain more files than others.

For compressors that support multiple levels, including CPU-Zstandard, Bzip2, Gzip, and SPDP, we evaluate all modes and present results for the fastest and best-compressing modes. Note that MPC requires the tuple size of the input, and FPzip, ZFP, and Ndzip need the dimensions of the input to work properly. We provided this information for all runs.

We measure the compression ratio by dividing the initial file size by the compressed size, and the compression and decompression throughput by diving the initial file size by the compression or decompression time. This time does not include reading/writing the data from/to secondary storage. To eliminate outliers, we use the median runtime of five identical runs for computing the throughputs. In all cases, a higher ratio or throughput indicates better performance.

Studying only the compression ratio or only the throughput does not fully describe the quality of a compression algorithm because, in many cases, a high compression ratio comes at the cost of a low throughput and vice versa. This is why we present the results in form of scatter plots showing either both the compression ratio and the compression throughput or both the compression ratio and the decompression throughput. In each such plot, we highlight the Pareto front [29]. All compressors that lie on this front are *optimal* in the sense that there is no other compressor that is both faster and compresses more.

5 Results

In this section, we compare the compression ratio, compression throughput, and decompression throughput of our 4 new algorithms to 11 lossless GPU and 9 lossless CPU compressors on 2 CPUs and 2 GPUs. We first present results for single-precision and then for double-precision data.

5.1 Single-Precision Data

GPU results: Figures 8 and 9 present the performance results of SPratio and SPspeed on the RTX 4090. In both figures, the y-axis lists the compression ratio. The x-axis shows the compression throughput in Figure 8 and the decompression throughput in Figure 9. Each data point reflects the performance of a compressor along the two dimensions. For example, Figure 8 shows that SPspeed reaches a geometric-mean compression ratio of 1.41 and a geometric-mean compression throughput of 518 GB/s on the 90 SDRbench inputs.

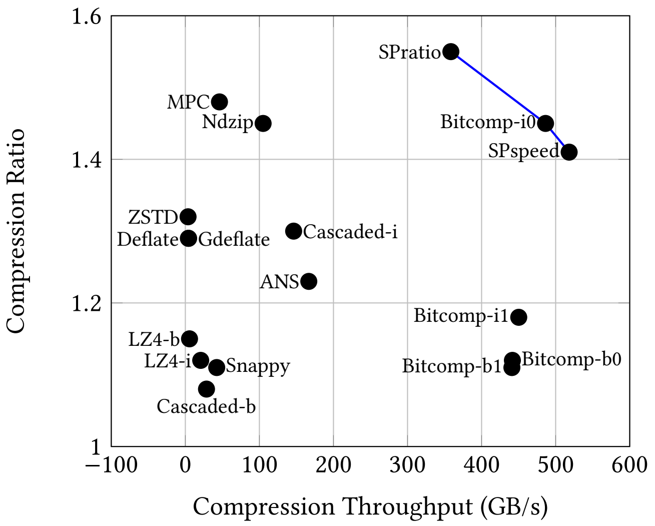

Figure 8. RTX 4090 compression ratio vs. compression throughput, including Pareto front, on single-precision data

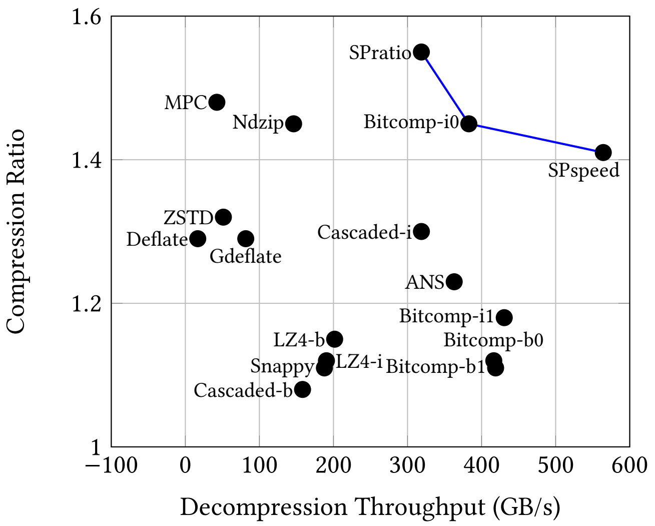

Figure 9. RTX 4090 compression ratio vs. decompression throughput, including Pareto front, on single-precision data

In both figures, the Pareto front includes SPratio, SPspeed, and Bitcomp-i0, indicating that these 3 algorithms outperform the others either in compression ratio, throughput, or both. Specifically, SPratio delivers the highest compression ratio whereas SPspeed delivers the highest throughput.

Since the throughputs are GPU dependent, we repeated the experiments on an A100 GPU that is based on an older architecture. Figures 10 and 11 show the results.

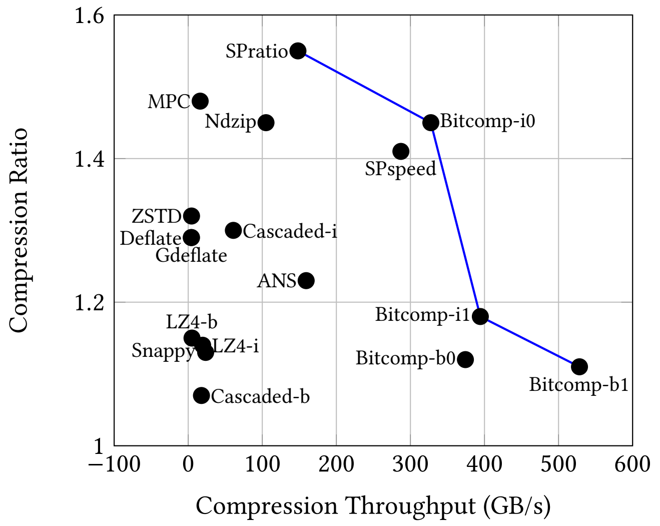

Figure 10. A100 compression ratio vs. compression throughput, including Pareto front, on single-precision data

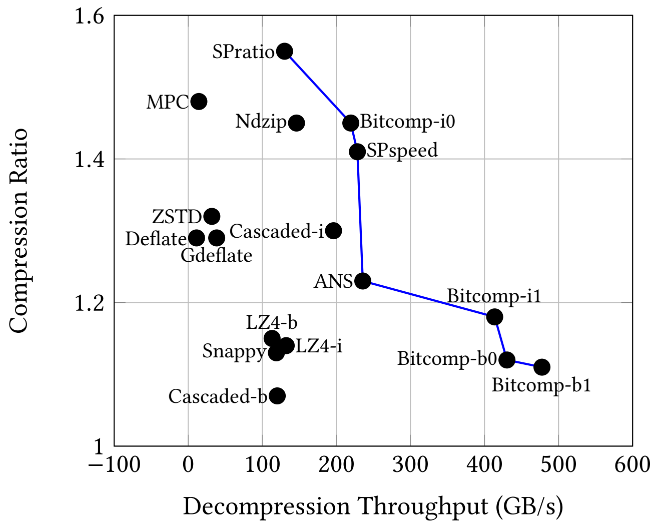

Figure 11. A100 compression ratio vs. decompression throughput, including Pareto front, on single-precision data

The Pareto front includes SPspeed in Figure 11 and SPratio in both figures, highlighting the good performance of our algorithms on both GPUs. Bitcomp-b0's decompressor and Bitcomp-b1's compressor and decompressor run faster on the A100 than on the RTX 4090. All other compressors and decompressors are faster on the newer GPU. Bitcomp-b appears to be particularly optimized for the A100, which has more shared memory and supports more threads per SM. In contrast, we optimized our compressors and decompressors for newer GPUs, which is why they deliver substantially higher throughputs on the RTX 4090.

Note that the nvCOMP compressors (including all versions of Bitcomp) produce multiple separately-stored compressed data chunks that are *not concatenated*. Skipping this important step, which is generally needed in applications that use compression, gives them a significant speed advantage over the non-nvCOMP compressors, including ours. Moreover, all nvCOMP and most of the other compressors we compare to in this subsection only work on GPUs, meaning their compressed data cannot be used on a CPU. In contrast, SPspeed, SPratio, and Ndzip support both types of devices and concatenate the compressed data into a contiguous memory block. Despite these extra features, our codes are among the fastest and compress more than the other GPU compressors.

CPU results: This subsection compares the performance of the CPU version of SPratio and SPspeed to 9 other lossless CPU compressors. Figures 12 and 13 show the results for the AMD Ryzen. Note that the x-axes of these charts use a logarithmic scale due to the vast difference in throughput between the various compressors.

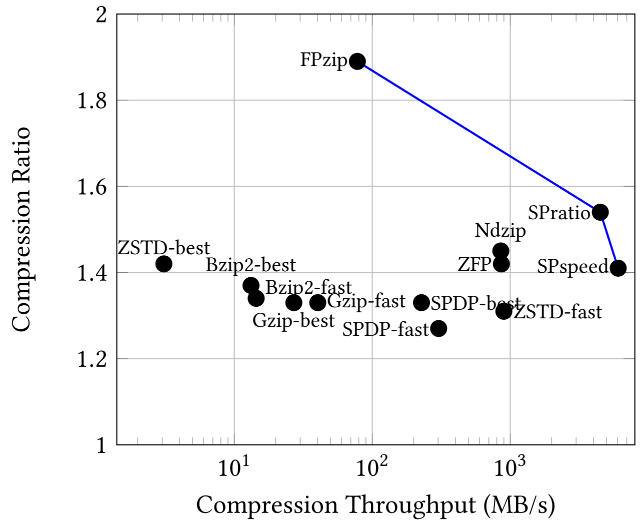

Figure 12. Ryzen compression ratio vs. compression throughput, including Pareto front, on single-precision data

The CPU versions of SPspeed and SPratio reach much higher compression and decompression throughputs than any of the other studied algorithms. Since the x-axis is logarithmic, the seemingly small benefit in throughput is actually quite large. Moreover, SPratio provides a higher compression ratio than the other algorithms except for FPzip, which yields by far the best compression ratio. Hence, the only compressors on the Pareto front are FPzip and both of our codes. SPspeed compresses 75 times faster and decompresses 55 times faster than FPzip.

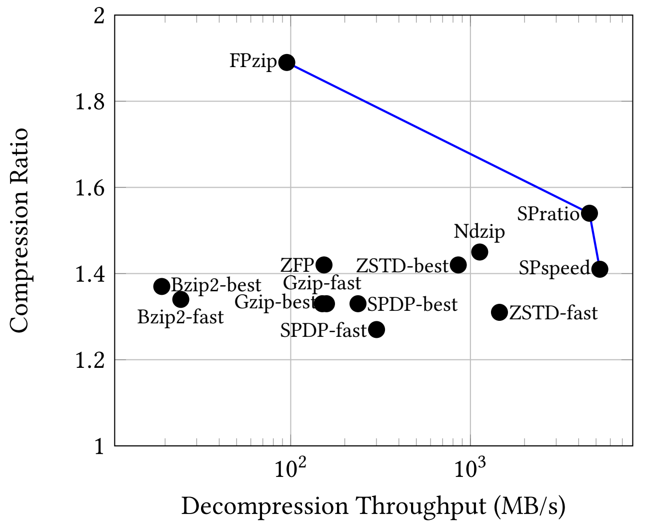

Figure 13. Ryzen compression ratio vs. decompression throughput, including Pareto front, on single-precision data

We repeated the CPU experiments on a second system from a different vendor that is based on an Intel Xeon. The results are not shown as they are qualitatively very similar to those from the Ryzen system. The main difference is that the throughputs are generally higher since the Xeon system contains two sockets with twice as many cores as the Ryzen system. Again, SPspeed and SPratio dominate the compression and decompression throughput. Together with FPzip, they are the only compressors on the Pareto front.

5.2 Double-Precision Data

GPU results: This section presents and discusses the double-precision performance of the GPU compressors. Figures 14 and 15 illustrate the RTX 4090 results.

DPratio stands out with a much higher compression ratio than the other tested GPU codes. This is mainly due to our new FCM transformation, which makes it possible to find repeating values in a GPU-friendly way even when they are far apart. However, this comes at the cost of a low throughput. Nevertheless, DPratio is on the Pareto front, which it shares with DPspeed. Bitcomp is also on the compression Pareto front but only achieves a compression ratio of 1.04, which is not particularly useful. As mentioned, Bitcomp's throughput is somewhat inflated because it does not concatenate the compressed data into a contiguous block of memory.

In Figure 15, which focuses on the compression ratio and decompression throughput, only DPratio and DPspeed are on the Pareto front, highlighting their good performance. Note that DPratio's decompression throughput is much higher than its compression throughput because no sorting is required in the FCM decoder. These results also show that

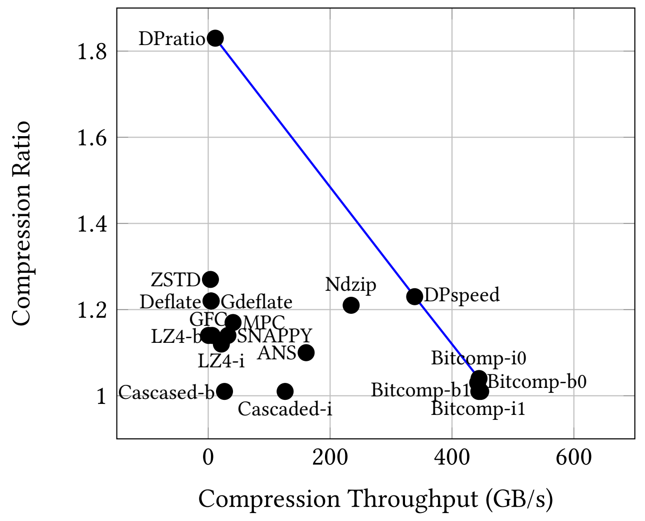

Figure 14. RTX 4090 compression ratio vs. compression throughput, including Pareto front, on double-precision data

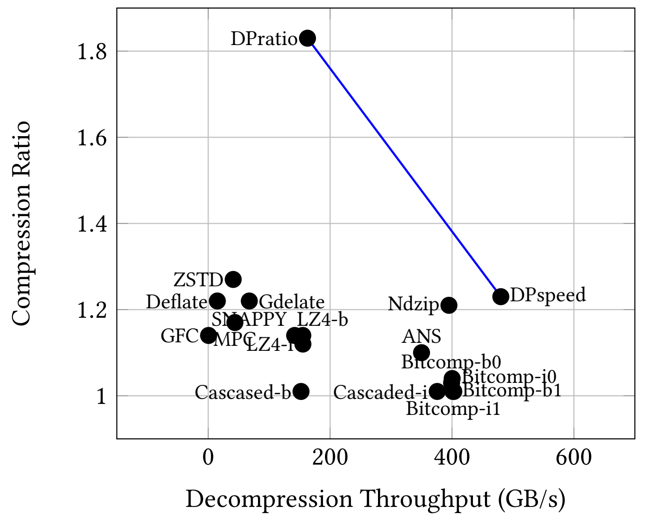

Figure 15. RTX 4090 compression ratio vs. decompression throughput, including Pareto front, on double-precision data

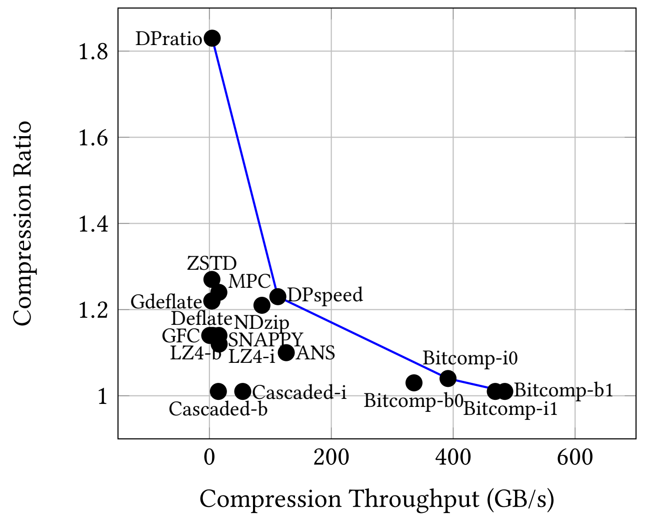

Figure 16. A100 compression ratio vs. compression throughput, including Pareto front on double-precision data

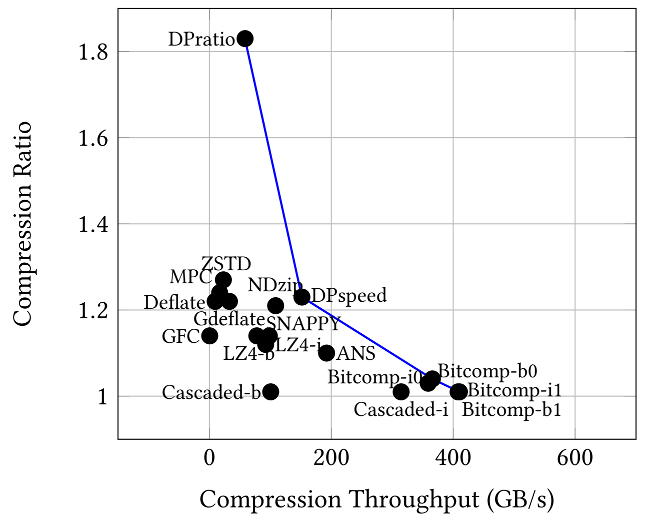

Figure 17. A100 compression ratio vs. decompression throughput, including Pareto front on double-precision data

the union-find approach (see the end of Section 3.2) is quite fast even on a GPU. Both of our algorithms provide a useful balance between compression ratio and decompression throughput, which is not the case for several other evaluated compressors. Moreover, as mentioned, most of the other GPU compressor's results cannot be used on a CPU. Only our codes and Ndzip provide full CPU/GPU compatibility.

Again, we repeated the same study on a second GPU for a more comprehensive assessment. Figures 16 and 17 show the results on the A100.

Both figures exhibit similar patterns. DPspeed and DPratio are both on the Pareto front alongside Bitcomp and ANS. Again, some of the Bitcomp versions are faster on the A100 than on the RTX 4090. All other codes, including ours, are faster on the RTX 4090. Since no sorting is involved, DPratio's decompression throughput is much higher than its compression throughput. In general, the trends for the double-precision results are similar to those for the single-precision results discussed earlier.

CPU results: DPratio and DPspeed also include CPU versions. Figures 18 and 19 show the results on the Ryzen. The x-axes use a logarithmic scale to capture the large differences in throughput between the various compressors.

DPspeed has the highest throughput by a substantial margin (note the logarithmic x-axes). For example, it compresses and decompresses roughly 10 times faster than pFPC, which achieves a similar compression ratio.

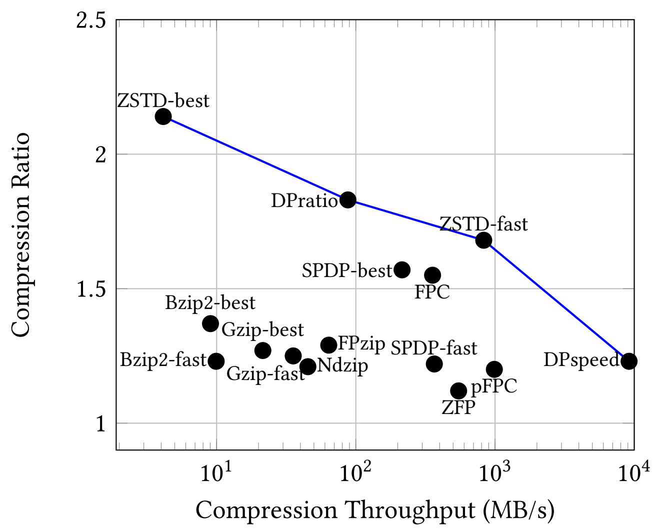

Figure 18. Ryzen compression ratio vs. compression throughput, including Pareto front, on double-precision data

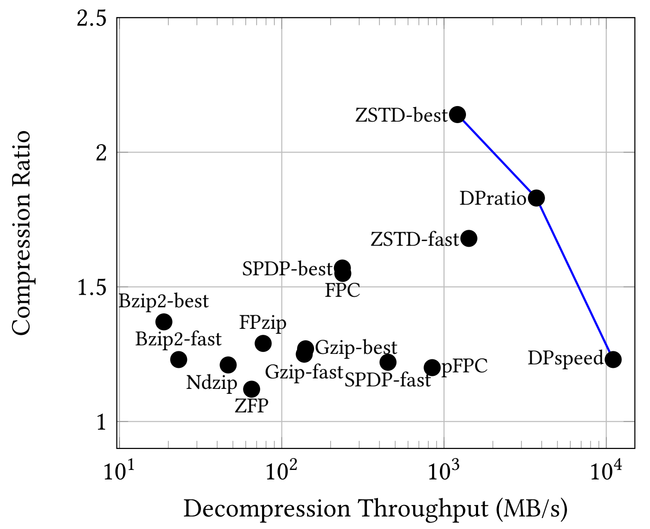

Figure 19. Ryzen compression ratio vs. decompression throughput, including Pareto front, on double-precision data

Although DPratio delivers a high compression ratio, the CPU version of Zstandard at its highest compression level reaches an even higher ratio albeit at a lower throughput. Due to the sorting in the FCM stage, DPratio's compression throughput is over an order of magnitude lower than its decompression throughput. This is why, on the compression side, the fastest version of Zstandard is also on the Pareto front. Recall that this implementation is from a different source than the GPU version and is incompatible with it.

Despite the sorting, DPratio is not among the slowest compressors. Moreover, it is the second fastest decompressor (after DPspeed), highlighting the speed of the union-find approach. Overall, DPratio provides a nice balance between throughput and compression ratio on the CPU.

We repeated the same study on the Xeon CPU. As before, the results exhibit very similar trends to those observed on the Ryzen CPU, with both DPspeed and DPratio appearing on the Pareto front along with Zstandard. Hence, we do not show figures for these results.

6 Summary and Conclusion

We introduce SPratio, SPspeed, DPratio, and DPspeed, 4 lossless compression algorithms tailored to single- and double-precision floating-point data. SP/DPratio prioritize a higher compression ratio whereas SP/DPspeed focus on maximizing compression and decompression throughput. Both modes deliver relatively high compression ratios and throughputs.

The four algorithms are based on new combinations of data transformations, some of which we have enhanced and some of which we have created. SPspeed and DPspeed are based on 2 fast data transformations: delta encoding with representation change and MPLG, an efficient approach for eliminating leading zero bits. SPratio encompasses 3 data transformations: delta encoding with representation change, bit shuffling, and repeated zero elimination. The "repetition" is an enhancement we designed to boost the compression ratio. DPratio comprises 4 data transformations: FCM, delta encoding with representation change, RAZE, and RARE.

FCM, RAZE, and RARE are new transformations we created. FCM builds pairs of hashed values and indices, sorts them, and then uses the result to encode the data in a format that is easier to compress. RAZE eliminates leading zero bits adaptively, while RARE eliminates common leading bits adaptively. Both of them optimally split the words into lower and upper pieces, where only the upper pieces undergo the leading zero or leading common-bits elimination. This is particularly useful on double-precision values, which tend to have rather random bits in the least-significant bit positions.

We selected these transformations because they yield a high compression ratio and can be implemented in parallel on CPUs and GPUs. In fact, all four of our algorithms have compatible CPU and GPU implementations, meaning users can compress their data on a CPU or a GPU and decompress the result on either a CPU or a GPU.

We compare the compression ratio, compression throughput, and decompression throughput of our four algorithms to 11 lossless GPU compressors and 9 lossless CPU compressors on 90 single-precision floating-point files from the SDRbench suite and 20 double-precision files. Our measurements on two GPUs from different generations and two CPUs from different vendors show that, with one exception, SP/DPratio and SP/DPspeed are always on the Pareto front.

SPratio delivers the highest compression ratio on the GPUs and the second highest ratio on the CPUs behind FPzip, which is over 55 times slower. Similarly, DPratio reaches by far the highest compression ratio on the GPUs and the second highest ratio on the CPUs behind Zstandard-best, which is several times slower. SPspeed yields the highest throughputs by large margin on the CPUs, is the fastest on one of our GPUs, and is faster than most compressors on the other GPU. Likewise, DPspeed is the fastest compressor and decompressor on the CPUs and faster than most GPU codes. Both of our GPU codes are outperformed in throughput (but not in compression ratio) in some cases by ANS and Bitcomp, both of which do not concatenate the compressed data chunks into a contiguous block of memory, which our compressors and all other non-nvCOMP compressors do.

Our implementations outperform Ndzip, the only other studied algorithm that provides CPU/GPU compatibility, in compression ratio and speed in most cases and on average. Overall, our algorithms are well-suited for environments where both a high throughput and a high compression ratio are needed, since they deliver up to 500 GB/s of compression and decompression throughput on an RTX 4090 GPU in combination with some of the highest compression ratios.

7 Acknowledgments

This work has been supported in part by the Department of Energy, Office of Science under Award No. DE-SC0022223 as well as by an equipment donation from NVIDIA Corporation.

References

[1] [n. d.]. Gzip: The GNU data compression utility. https://www.gnu.org/software/gzip/. Accessed: July 31, 2023.

[2] Year. nvCOMP: NVIDIA GPU Data Compression Library. https://github.com/NVIDIA/nvcomp. Accessed: July 31, 2023.

[3] Noushin Azami and Martin Burtscher. 2022. Compressed In-memory Graphs for Accelerating GPU-based Analytics. In *2022 IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 32–40.

[4] Noushin Azami, Alex Fallin, Brandon Burtchell, Andrew Rodriguez, Benila Jerald, Yiqian Liu, and Martin Burtscher. 2024. LC Git Repository. https://github.com/burtscher/LC-framework. Accessed: 2024-10-16.

[5] Maciej Besta, Dimitri Stanojevic, Tijana Zivic, Jagpreet Singh, Maurice Hoerold, and Torsten Hoefler. 2018. Log (graph) a near-optimal high-performance graph representation. In *Proceedings of the 27th international conference on parallel architectures and compilation techniques*. 1–13.

[6] Martin Burtscher. 2024. FPdouble: A Dataset of Double-Precision Floating-Point Numbers. https://userweb.cs.txstate.edu/~burtscher/research/datasets/FPdouble/.

[7] Martin Burtscher and Paruj Ratanaworabhan. 2007. High-throughput Compression of Double-precision Floating-point Data. In *2007 Data Compression Conference (DCC'07)*. IEEE, 293–302.

[8] Martin Burtscher and Paruj Ratanaworabhan. 2008. FPC: A High-speed Compressor for Double-precision Floating-point Data. *IEEE transactions on computers* 58, 1 (2008), 18–31.

[9] Martin Burtscher and Paruj Ratanaworabhan. 2009. pFPC: A Parallel Compressor for Floating-point Data. In *2009 Data Compression Conference*. IEEE, 43–52.

[10] Franck Cappello, Sheng Di, Sihuan Li, Xin Liang, Gok M. Ali, Dingwen Tao, Chun Yoon Hong, Xin-chuan Wu, Yuri Alexeev, and T. Frederic Chong. 2019. Use cases of lossy compression for floating-point data in scientific datasets. *International Journal of High Performance Computing Applications (IJHPCA)* 33 (2019), 1201–1220.

[11] Steven Claggett, Sahar Azimi, and Martin Burtscher. 2018. SPDP: An Automatically Synthesized Lossless Compression Algorithm for Floating-point Data. In *2018 data compression conference*. IEEE, 335–344.

[12] Yann Collet. 2016. Zstandard compression. https://github.com/facebook/zstd. *GitHub repository* (2016).

[13] Peter Deutsch. 1996. *DEFLATE compressed data format specification version 1.3*. Technical Report.

[14] Jarosław Duda. 2014. Asymmetric Numeral Systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding. In *Proceedings of the Data Compression Conference (DCC)*. 309–318. https://doi.org/10.1109/DCC.2014.43

[15] Thomas E. Fornek. 2017. Advanced Photon Source Upgrade Project preliminary design report. https://doi.org/10.2172/1423830

[16] Manuel Noronha Gamito and Miguel Salles Dias. 2004. Lossless coding of floating point data with JPEG 2000 Part 10. In *Applications of Digital Image Processing XXVII*, Vol. 5558. SPIE, 276–287.

[17] Florin Ghido. 2004. An efficient algorithm for lossless compression of IEEE float audio. In *Data Compression Conference, 2004. Proceedings. DCC 2004*. IEEE, 429–438.

[18] David A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE* 40, 9 (September 1952), 1098–1101. https://doi.org/10.1109/JRPROC.1952.273898

[19] Lawrence Ibarria, Peter Lindstrom, Jarek Rossignac, and Andrzej Szymczak. 2003. Out-of-core compression and decompression of large n-dimensional scalar fields. In *Computer Graphics Forum*, Vol. 22. Wiley Online Library, 343–348.

[20] inikep. 2023. Lzbench. https://github.com/inikep/lzbench.

[21] Fabian Knorr, Peter Thoman, and Thomas Fahringer. 2021. ndzip: A High-Throughput Parallel Lossless Compressor for Scientific Data. In *2021 Data Compression Conference (DCC)*. IEEE, 103–112.

[22] Fabian Knorr, Peter Thoman, and Thomas Fahringer. 2021. ndzip-gpu: Efficient lossless Compression of Scientific Floating-Point Data on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[23] Abraham Lempel and Jacob Ziv. 1977. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory* 23, 3 (May 1977), 337–343. https://doi.org/10.1109/TIT.1977.1055714

[24] Linac Coherent Light Source (LCLS-II). 2017. https://lcls.slac.stanford.edu/. Online.

[25] Peter Lindstrom. 2014. Fixed-rate compressed floating-point arrays. *IEEE transactions on visualization and computer graphics* 20, 12 (2014), 2674–2683.

[26] Peter Lindstrom and Martin Isenburg. 2006. Fast and efficient compression of floating-point data. *IEEE transactions on visualization and computer graphics* 12, 5 (2006), 1245–1250.

[27] David J.C. MacKay. 2003. Information theory, inference and learning algorithms. *Cambridge University Press* (2003). Chapter 4: "Source Coding and Compression".

[28] D. Merrill and M. Garland. 2016. *Single-pass Parallel Prefix Scan with Decoupled Look-back*. Technical Report NVR-2016-002. NVIDIA.

[29] Kaisa Miettinen. 1998. Theoretical Analysis of Multiobjective Optimization. *Doctoral Dissertation, Acta Polytechnica Scandinavica, Mathematics and Computing Series, No. 67* (1998). http://www.uio.no/studier/emner/matnat/ifi/nedlagte-emner/INF-MAT5380/v03/undervisningsmateriale/papers/pareto_front_definition.pdf

[30] Molly A O'Neil and Martin Burtscher. 2011. Floating-point Data Compression at 75 Gb/s on a GPU. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. 1–7.

[31] SDRBench Inputs https://sdrbench.github.io/, 2023. SDRBench Inputs. https://sdrbench.github.io/

[32] Julian Seward. 1996. Bzip2 and libbzip2. *avaliable at http://www.bzip.org* (1996).

[33] Julian Shun, Laxman Dhulipala, and Guy E Blelloch. 2015. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *2015 Data Compression Conference*. IEEE, 403–412.

[34] Natcha Simsiri, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. 2016. Work-efficient parallel union-find with applications to incremental graph connectivity. In *Euro-Par 2016: Parallel Processing: 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings 22*. Springer, 561–573.

[35] Bryan E Usevitch. 2003. JPEG2000 extensions for bit plane coding of floating point data. In *Data Compression Conference, 2003. Proceedings. DCC 2003*. IEEE, 451.

[36] Sheng Wu and Daniel Lemire. 2013. Fast Integer Compression Using SIMD Instructions. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 395–405. https://doi.org/10.1109/MICRO.2013.43

[37] Annie Yang, Hari Mukka, Farbod Hesaaraki, and Martin Burtscher. 2015. MPC: A Massively Parallel Compression Algorithm for Scientific Data. In *2015 IEEE International Conference on Cluster Computing*. IEEE, 381–389.

[38] Kai Zhao, Sheng Di, Xin Liang, Sihuan Li, Dingwen Tao, Julie Bessac, Zizhong Chen, and Franck Cappello. 2020. SDRBench: Scientific Data Reduction Benchmark for Lossy Compressors. In *International Workshop on Big Data Reduction (IEEE IWBDR20) in conjunction with IEEE International Conference on Big Data (IEEE BigData20)*.

A Artifact Appendix

A.1 Abstract

This artifact contains the code and script to generate compression-ratio and throughput results for the 4 algorithms the paper introduces. The results should be similar to the numbers shown in Figures 8 through 19 for SPratio, SPspeed, DPratio, and DPspeed, that is, the compression ratios should match exactly but the compression and decompression throughputs are system dependent.

A.2 Artifact check-list (meta-information)

- **Algorithm: SPratio, SPspeed, DPratio, and DPspeed**
- **Compilation: g++ and nvcc**
- **Data set: SDRBench**
- **Hardware: CPU and GPU**
- **Execution: Parallel**
- **Metrics: Compression ratio and throughput**
- **Output: Compression ratio vs. throughput scatter plots**
- **How much disk space required (approximately)?: 100 GB**
- **How much time is needed to prepare workflow (approximately)?: 180 minutes to download inputs**
- **How much time is needed to complete experiments (approximately)?: 100 minutes for the GPU and 200 minutes for the CPU**
- **Publicly available?: Yes**
- **Code licenses (if publicly available)?: BSD 3-Clause License**
- **Workflow automation framework used?: Python scripts**
- **Archived (provide DOI)?: https://doi.org/10.5281/zenodo.14061031**

A.3 Description

A.3.1 How to access. The artifact can be found at https://github.com/burtscher/FPcompress.

A.3.2 Hardware dependencies. The hardware required for this artifact is an x86 multi-core CPU and a CUDA-capable GPU. We used a 32-core Intel Xeon Gold 6226R CPU @ 2.9 GHz with hyperthreading enabled to run the CPU codes. To run the GPU codes, we used an NVIDIA RTX 4090. Using similar hardware should result in throughputs similar to those reported in the paper.

A.3.3 Software dependencies. The required software includes:

- The computational artifact from https://github.com/burtscher/FPcompress
- GCC 7.5.0 or higher
- OpenMP 3.1 or higher
- CUDA 11.0 or higher
- Python v3.4 or higher
- Matplotlib v3.6 or higher

A.3.4 Data sets. The data sets used in the artifact are downloaded as part of the installation process and can be found at https://sdrbench.github.io.

A.4 Installation

To install the artifact

- Clone the repository from https://github.com/burtscher/FPcompress
- Run './compile.py' to compile SPratio, SPspeed, DPratio, and DPspeed

A.5 Experiment workflow

- Clone the repository from https://github.com/burtscher/FPcompress
- Run './get_inputs_double.py' and './get_inputs_single.py' to download and set up the inputs used by the artifact
- Run './compile.py' to compile SPratio, SPspeed, DPratio, and DPspeed
- Run './run_experiments_double.py' and './run_experiments_single.py to produce the intermediate experimental results
- Run './chart_double.py' and './chart_single.py' to generate compression and decompression charts that look like Figures 8 through 19 but without the results for the third-party codes.
- View the charts, which can be found in the current directory under 'double_charts.png' and 'single_charts.png'

A.6 Evaluation and expected results

The evaluation of the results is accomplished by comparing the figures generated using this artifact to the SPratio, SPspeed, DPratio, and DPspeed results listed in Figures 8 through 19. The absolute values of the throughputs and the relative positions may be different based on the CPU and GPU used, but the compression ratios should be the same.

A.7 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae