ndzip: A High-Throughput Parallel Lossless Compressor for Scientific Data

Fabian Knorr, Peter Thoman and Thomas Fahringer

Distributed and Parallel Systems Group
University of Innsbruck, Austria
{fabian,petert,tf}@dps.uibk.ac.at

Abstract

Exchanging large amounts of floating-point data is common in distributed scientific computing applications. Data compression, when fast enough, can speed up such workloads by reducing the time spent waiting for data transfers. We propose ndzip, a high-throughput, lossless compression algorithm for multi-dimensional univariate regular grids of single- and double-precision floating point data. Tailored towards efficient implementation on modern SIMD-capable multicore processors, it compresses and decompresses data at speeds close to main memory bandwidth, significantly outperforming existing schemes. We evaluate this novel method using a representative set of scientific data, demonstrating a competitive trade-off between compression effectiveness and throughput.

Introduction

In distributed and high performance computing (HPC), interconnect bandwidth continues to be a significant limiting factor, manifesting across buses, storage, and network links [1]. Data compression can help mitigate this issue, but candidate algorithms are subject to a number of unique constraints: (i) the performance of both compression and decompression needs to be at least as fast as the transfer or storage medium; (ii) without additional per-application metainformation, the algorithm needs to be lossless; (iii) data commonly consists of n-dimensional grids of floating point values, which are smooth in some dimension but individually unique, and can therefore be difficult to compress with general-purpose algorithms.

We introduce **ndzip**, a new high-throughput lossless compression algorithm designed specifically for n-dimensional grids of floating point data. Our concrete contributions are as follows: (1) a new compression algorithm, ndzip, based on a fast, parallel integer approximation of a well-known predictor combined with a hardware-friendly block subdivision scheme; (2) a high-performance multi-level parallel implementation of ndzip, leveraging both SIMD and thread-level parallelism; (3) an in-depth performance evaluation across a large set of representative HPC data, with comparisons against the state of the art in both specialized floating point compressors and general purpose compression schemes.

Background

Related Algorithms fpzip [2] uses the Lorenzo predictor [3] to exploit smoothness in the direct neighborhood of points within a scalar n-dimensional grid, compacting the small residual values using a range coder. The scheme exhibits high compression efficiency, especially for single-precision values.

FPC [4] uses a pair of hash-table based value predictors to compress an unstructured double-precision data stream. It offers a tunable parameter trading compression effectiveness for speed. The thread-parallel pFPC variant [5] allows further prioritization of compression throughput by processing input data in chunks.

SPDP [6] combines one-dimensional prediction with an LZ77 variant to compress both single- and double-precision data without specialization for either format.

MPC [7] is a fast compression scheme for GPUs. A simple one-dimensional value predictor is combined with a bit-regrouping scheme for zero-bit elimination in the residual which maps well to the targeted hardware.

The APE and ACE compressors [8] adaptively select from multiple value predictors to decorrelate data points in an n-dimensional grid from their already-processed neighbors. Residuals are compacted using a variant of Golomb coding.

Value Prediction Individual values from scientific floating-point data commonly exhibit high entropy in the low-order mantissa bits and seldom repeat exactly, lowering efficiency of the traditional dictionary coder approach. Instead, specialized methods try to predict values from already processed data and only encode residuals.

SPDP and MPC use simple fixed-stride value predictors, remembering a history of k values and predicting each point with the k-th most recently encoded value.

FPC and pFPC maintain larger internal state with a pair of hash table based predictors to exploit repeating patterns in values and value deltas.

The floating-point Lorenzo predictor employed by fpzip estimates the value on one corner of a length-2 hypercube within an n-dimensional space by summing up all other corners reachable via an odd number of edges and subtracting those reachable via an even number. The prediction accuracy increases with smoothness of the data and is exact when the hypercube is an implicit polynomial of degree $n - 1$.

The APE and ACE predictors expand on this idea by using higher-dimensional polynomials in each dimension, which increases prediction accuracy at the expense of greater computational cost.

Difference Operators In a lossless compression context, floating-point subtraction is unsuitable for computing prediction residuals. Small-magnitude floating-point values usually do not manifest in short, compressible bit representations, and the format's limited precision make floating-point subtraction a non-bijective operation. Thus, all studied algorithms compute residuals on bit representations explicitly.

FPC and pFPC use a bit-wise XOR difference, while SPDP and MPC reinterpret the operand bits as integers and encode the result of integer subtraction. The APE and ACE schemes offer both variants. fpzip also uses integer subtraction, but negates the operand depending on the sign bit to improve continuity in the mapping.

Residual Value Encoding Accurate predictions yield small-magnitude residuals with many identical leading bits, i.e. zeros for the XOR operator and redundant sign bits for two's complement integer subtraction. Efficiently encoding these leading bits is the only data reduction mechanism employed by most studied schemes.

fpzip uses a range coder to compress the number of leading redundant bits, followed by a verbatim copy of the trailing bits. The near-optimal bit string produced by the range coder makes this approach very space-efficient. However, the required bit-granularity addressing is difficult to implement efficiently.

The APE and ACE schemes take a similar approach, but encode the number of redundant bits using a symbol-ranking Golomb code.

FPC and pFPC count the number of leading-zero bytes in a double-precision residual, using a fixed mapping to encode the run length together with the predictor selection in a four-bit value. The remainder, starting with the first non-zero byte, is output verbatim. This method is stateless and has an acceptable $1/16$ overhead in the incompressible case at the expense of wasting bits due to its lower granularity.

MPC splits the residual stream into chunks of 32 single- (or 64 double-) precision values, emitting the 32 (64) most-significant bits followed by the 32 (64) second-most significant bits, and so on. Zero-words are eliminated from the output stream and replaced with a 32- or 64-bit mask per chunk encoding the positions of all non-zero words. This method has very low $1/32$ ($1/64$) overhead in the incompressible case and is efficiently implemented on GPUs due to its word-granularity addressing, but requires all residuals within a chunk to have similar bit width to be effective.

SPDP starts with a regrouping scheme similar to MPC, but on a byte level. It follows up with a byte-granularity integer-subtraction difference operator and encodes the resulting stream using a LZ77-family coder. This can eliminate repeating patterns other than leading zeros and makes SPDP work on non-floating-point data as well.

Algorithm

We introduce ndzip, a block-transform-based lossless compression algorithm tailored towards a highly-efficient implementation on SIMD-capable multicore processors. By exposing data parallelism both within and across blocks, we are able to compress and decompress data near the memory bandwidth limit.

Our implementation supports up to three-dimensional grids of arbitrary extent, although the algorithm is easily generalized to higher dimensions. The expected input data format is a stream of little-endian IEEE 754 single- or double-precision values, and out-of-band metadata explaining data dimensionality and extent.

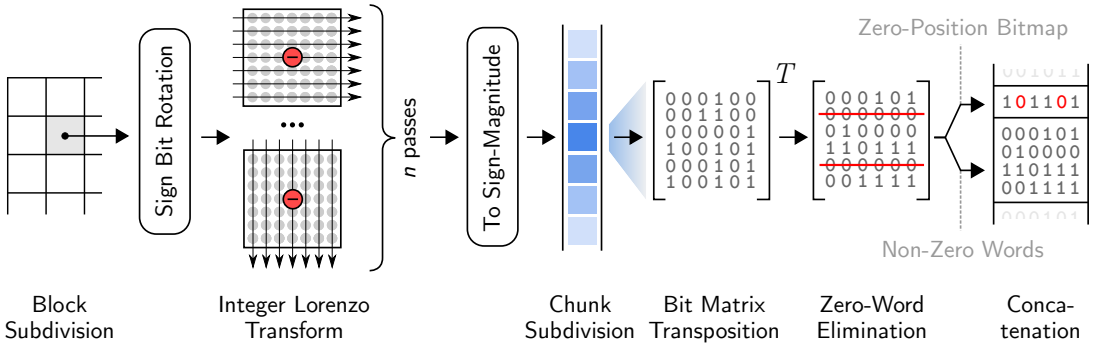

Figure 1: Overview of the ndzip compression pipeline

Overview Compression and decompression in ndzip are symmetric with regard to their algorithmic building blocks and performance characteristics.

Figure 1 shows all steps of the compression pipeline. It divides input data into fixed-size hypercubes and employs a multidimensional transform step to decorrelate data within blocks, yielding residuals with a shorter bit representation. Chunks of the

residual stream are then compacted by eliminating common zero-bits through a bit-matrix transposition scheme. The compacted chunks are stored alongside a header revealing positions of the compressed blocks within the output stream.

The decompression pipeline inverts each compression step by expanding the original chunks from their compressed form, repeating the bit-matrix transposition step to obtain residuals and finally applying the inverse block transform step to restore the uncompressed hypercube.

Block Subdivision Instead of attempting to process the entire n-dimensional grid of input data at once, ndzip subdivides it into small hypercubes which are compressed independently. Since the algorithm requires multiple passes over the data, this enables better use of processor caches at the expense of a minor loss in decorrelation efficiency. By eliminating all dependencies between blocks, this subdivision scheme exposes additional data parallelism to the implementation.

We choose a size of 4096 elements, which manifests as hypercube sizes of 4096^1, 64^2 or 16^3 for one to three dimensions, respectively. This corresponds to to 16 KB of memory for single- and 32 KB for double precision. Fixing the block size in advance allows the generation of highly-optimized machine code for the following steps.

When the grid extent is not a multiple of the block size, border elements are simply appended to the output uncompressed.

Integer Lorenzo Transform The floating-point Lorenzo predictor is highly effective on multidimensional data. However, the separate bit-pattern residual computation step requires the decompressor to reconstruct each prediction from already-decoded neighboring values, introducing dependences that limit parallelism.

We circumvent this problem in the novel Integer Lorenzo Transform, a multi-pass operation which directly computes an approximation of Lorenzo prediction residuals within the integer domain. Figure 2 illustrates this process. Both forward and inverse transform steps are in-place, and, in the case of multi-dimensional grids, data parallel.

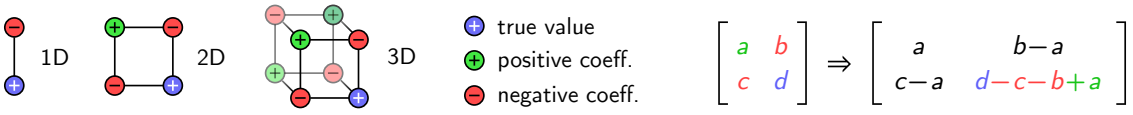

Figure 2: Computation of the Lorenzo prediction residual; example for point d in a 2D array

The first pass maps all floating-point values in a block to an integer representation. This mapping must be reversible and should ideally preserve small floating-point differences in the input as small integer differences in the output. Reinterpreting the bits of an IEEE 754 floating-point value as a two's complement integer of the same width already yields a monotone mapping in the exponent and mantissa bits. A discontinuity arises near zero due to the input's sign-magnitude representation of floating-point values. Inverting exponent and mantissa bits depending on the sign to obtain a two's complement integer as proposed by Lindstrom et al. [2] is not optimal since opposite-sign, similar-magnitude values usually share exponent bits. We found the best solution to be a simple left-rotation of the floating-point representation by one bit, which puts the sign at the least-significant position.

The following passes iteratively obtain the Lorenzo prediction residual of each value in the integer domain. For the one-dimensional case, this means replacing all

input values (but the first) with the difference to its predecessor, following Figure 2.

For the higher-dimensional case, we observe that residual computation is separable: remainders in two dimensions are calculated by first performing one-dimensional subtraction along each row, followed by one-dimensional subtraction along each column. This is trivially generalized to arbitrary n-dimensional grids, requiring n passes over the data. All subtraction operations within one pass are data-independent, so a fully parallel transform requires just $\mathcal{O}(n)$ steps.

The inverse operation, restoring input values from residuals, is separable in the same way. In each pass, it adds the restored predecessor to each residual value. Since this introduces a dependency chain along each row, the inverse transform passes are fully parallel in only $n - 1$ dimensions. Using a parallel scan for summation yields a complexity of $\mathcal{O}(n \log(k))$ steps for a block width of k.

Residual Value Encoding ndzip uses the same residual value encoding scheme as MPC [7], which can be implemented very efficiently on modern CPUs.

Since residuals use a two's complement representation, small values will start with a string of leading-zero or leading-one bits depending on their sign. To encode both through zero-elimination, residuals are first converted to a sign-magnitude representation by flipping all but the first bit whenever the residual is negative.

The scheme then divides the residual stream into chunks of 32 single-precision or 64 double-precision values and performs a 32×32 (64×64) bit matrix transposition on each chunk to group bits from the same position into words. All zero-words are stripped from the output, and a 32-bit (64-bit) header is prepended to each chunk, encoding the position of non-zero words as a bitmap.

Implementation

ndzip is tailored towards efficient use of SIMD units in modern CPUs. Although modern compilers can autovectorize certain code patterns, our implementation requires manual vectorization using platform intrinsics. We chose the AVX2 instruction set, available on most x86_64 processors since 2013. AVX2 vector registers are 256 bit wide and can thus operate on 8 single- or 4 double-precision values simultaneously.

Integer Lorenzo Transform Within an n-dimensional block, we identify the dimension with the fastest-changing index as the horizontal dimension.

In the horizontal dimension of the forward transform, 8 (4) residuals can be computed in parallel by loading two 256-bit registers with an offset of 4 (8) bytes and performing a single vector subtraction. This offset incurs one unaligned load, which newer processors do not penalize significantly.

The inverse exhibits limited parallelism in the horizontal dimension since a true dependence exists between adjacent values. Rows are still independent, but the transposition step required to load multiple values from a stride into one register would outweigh the speedup from parallel vector addition. We thus chose to only parallelize the inverse transform along $n - 1$ dimensions.

Implementing the higher-dimension passes of both forward and inverse transform is trivial. An aligned 256-bit load or store will operate on 8 (4) values from distinct, independent rows, allowing them to be processed in parallel.

All transform passes profit from software pipelining, i.e. the widening of vector instructions by operating on multiple registers in lock-step, to a small degree.

Residual Value Encoding A naive implementation of the bit matrix transposition step consists of a nested loop performing $32 \cdot 32 = 1024$ (or $64 \cdot 64 = 4096$) bitwise-and, bitwise-or and shift operations, or around 3 instructions per bit. This can be significantly improved by using vector operations to first reorder the input at a byte granularity and then extract neighboring bits in parallel.

The reordering step uses a sequence of AVX2 `vpunpck`, `vpshuf` and `vperm` instructions to rearrange input bytes such that the 32 (64) most significant residual bytes are followed by the 32 (64) second-most significant bytes, and so on. Clang 10.0.1 compiles this step to a sequence of 28 vector instructions for a 32×32 matrix and 241 instructions for a 64×64 matrix.

The extraction step then leverages the `vpmovmskb` instruction, which interprets a 256-bit word as a vector of 32 8-bit integers and extracts its 32 most-significant bits. Followed by a 1-bit left-shift of the vector register and a store of the 32-bit transposed column (or half-column, in the 64-bit case) to memory, this step produces 32 bits every 3 instructions. In total, the vectorized implementation requires $28 + 32 \cdot 3 = 124$ instructions for a 32×32 bit matrix and $241 + 64 \cdot 2 \cdot 3 = 625$ instructions for a 64×64 bit matrix to complete, a small fraction of the naive implementation.

Zero-words are eliminated using a simple loop with one branch per word. A quick but effective optimization lies in the observation that the compressed representation of an all-zero chunk is just an all-zero bitmap header. Common in highly-correlated input data, this pattern can be handled efficiently by zero-testing the chunk of residuals and conditionally skipping both transposition and zero-word elimination.

Thread Parallelism The scheme's subdivision into hypercubes exposes a high degree of parallelism between blocks that allows simple and efficient distribution of work among threads. Compression is an input-parallel problem, and while blocks can easily be transformed in parallel, more complex synchronization is required to concatenate the variable-length compressed representations in the output stream. Decompression on the other hand is output-parallel, making the implementation straightforward.

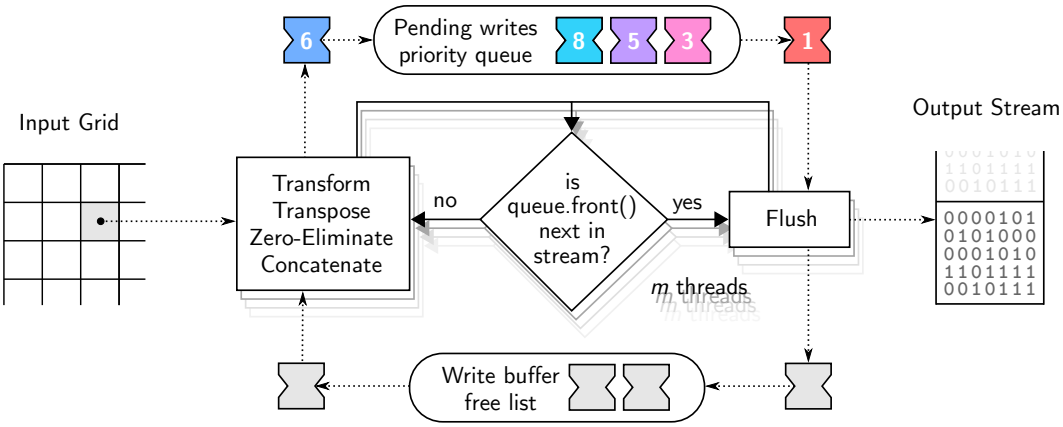

Figure 3: Producer-consumer scheme for thread-parallel compression

Figure 3 shows the parallel compressor, where a configurable number of identical threads work cooperatively on a set of pre-allocated write buffers in a producer-consumer scheme. In a loop, they load, transform and compact individual blocks, writing the compressed representation to a buffer taken from a shared buffer free list.

The filled buffer is then placed in a shared priority queue that orders buffers by their position in the output stream. When a task observes that the next item in the queue is the successor of the block most recently written to the output stream, it flushes the write, placing the buffer back into the free list.

For decompression, a simple work-sharing for-loop decompresses blocks by looking up the compressed stream offsets in the file header and decoding blocks in parallel.

Evaluation

Third-Party Software ndzip was compared against the original implementations of several floating-point compressors, namely fpzip 1.3.0[1], FPC 1.1[2], pFPC 1.0[2] and SPDP 1.1[2]. MPC, although related, was not considered since it only features a GPU implementation[2] and is thus not directly comparable to CPU algorithms. No implementation could be obtained for the APE and ACE compression schemes.

For context, we include state-of-the-art general-purpose compressors in our evaluation. zlib 1.2.11[3] is the de-facto standard implementation of the Deflate format. xzutils 5.2.5's liblzma[4] implements the expensive but effective LZMA scheme. LZ4 1.9.2[5] provides he throughput-optimized compressor of the same name. Zstd 1.4.5[6] is the reference implementation of the relatively new, efficient Zstandard compression scheme.

Test Data Compression strength and performance was evaluated on data of varying dimensionality from real-world applications [9], shown in Figure 4.

dataset	single	double	extent
msg_sppm	✓	✓	34,874,483
msg_sweep3d	✓	✓	15,716,403
snd_thunder	✓		7,898,672
ts_gas	✓		4,208,261
ts_wesad	✓		4,588,553
hdr_night	✓		8,192 × 16,384
hdr_palermo	✓		10,268 × 20,536
hubble	✓		6,036 × 6,014
rsim	✓	✓	2,048 × 11,509
spitzer_fls_irac	✓		6,456 × 6,389
spitzer_fls_vla	✓		8,192 × 8,192
spitzer_frontier	✓		3,874 × 2,694

dataset	single	double	extent
asteroid	✓		500 × 500 × 500
astro_mhd	✓		128 × 512 × 1024
astro_mhd		✓	130 × 514 × 1026
astro_pt	✓	✓	512 × 256 × 640
flow		✓	16 × 7,680 × 1,0240
hurricane	✓		100 × 500 × 500
magrecon	✓		512 × 512 × 512
miranda	✓		1,024 × 1,024 × 1,024
redsea	✓	✓	50 × 500 × 500
sma_disk	✓		301 × 369 × 369
turbulence	✓		256 × 256 × 256
wave	✓	✓	512 × 512 × 512

Figure 4: Scientific sample datasets used in compressor comparisons

Environment Our test system was a 12-core, 24-thread AMD Ryzen 9 3900X processor with 64 GB of DDR4-3200 memory in a dual-channel configuration, running Linux 5.8.14 with the `ondemand` CPU frequency governor. The ndzip source code was compiled using Clang 10.0.1 and the `-O3 -march=native` optimization flags.

Methodology We define the compression ratio of a dataset as compressed size divided by uncompressed size in bytes, with lower ratios indicating stronger compression. This definition directly relates to the reduction of bandwidth required to

[1] `https://github.com/LLNL/fpzip/releases/tag/1.3.0`
[2] `https://userweb.cs.txstate.edu/~burtscher/research/{FPC,pFPC,SPDPcompressor,MPC}`
[3] `https://www.zlib.net/`
[4] `https://tukaani.org/xz`
[5] `https://github.com/lz4/lz4/releases/tag/v1.9.2`
[6] `https://github.com/facebook/zstd/releases/tag/v1.4.5`

transfer a dataset. Where ratios need to be aggregated, the unweighted arithmetic mean of compression ratios over datasets is reported.

Performance was evaluated by measuring wall-clock time to compress and decompress from and to system memory, excluding any file I/O and initialization overhead. Third-party implementations were adapted to allow in-memory operation where necessary. We report the throughput of uncompressed bytes per second, which translates to compression input and decompression output bandwidth. Measurements for each algorithm–dataset pair were repeated until the total runtime exceeded one second. Before each iteration, input data was evicted from the CPU cache.

Predictor Approximation Quality To evaluate effectiveness of the novel Integer Lorenzo Transform (ILT), we replace the transform step of our implementation with other prediction methods and compare the resulting compression ratios. The gain from exploiting multidimensionality is measured by transforming higher-dimensional datasets with an equivalent one-dimensional transform instead. The quality of the ILT approximation, which attempts to match the decorrelation efficiency of floating-point Lorenzo prediction (FLP), is assessed by replacing the ILT step with FLP and a difference operator. For all predictors and transforms, both a simple XOR difference and the rotate-and-subtract integer difference from ndzip are evaluated.

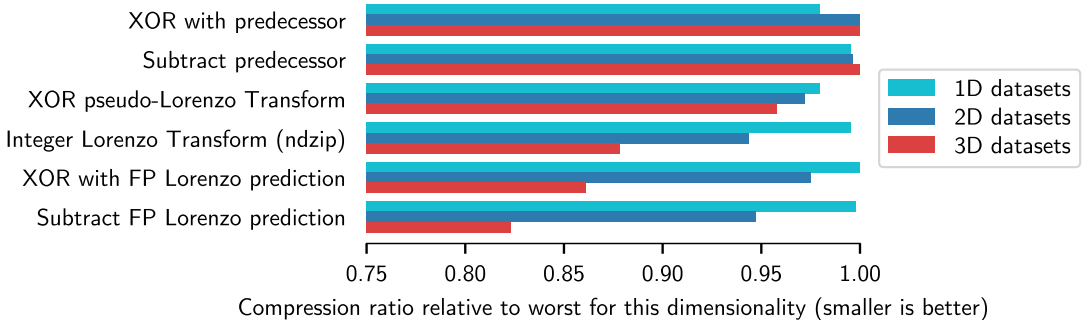

Figure 5: Relative compression ratios of different prediction and transform schemes

Figure 5 shows compression ratios averaged over all datasets of Figure 4 with the same dimensionality, scaled relative to the worst compression ratio observed in the respective dimension. For one-dimensional datasets, all schemes are roughly equivalent. Overall, FLP together with integer subtraction, similar to the scheme employed by fpzip, yields the best results, closely followed by XOR-remainder FLP and ILT. The worst options are the two one-dimensional predictors, showing that the Lorenzo-based components significantly benefit from higher dimensionality. Comparison with the XOR-difference variant of ILT, equivalent to calculating the XOR remainder of a length-2 hypercube for each value, demonstrates that careful choice of the remainder operation is essential for approximating the relevant features of FLP.

Compressor Efficiency Figure 6 compares throughput and compression ratios achieved by the examined compressor implementations on our test data. By design, not all algorithms can process both single- and double-precision values.

Some algorithms have one or more tunable parameters, in which case we report all configurations on the pareto front of the compression operation as a continuous line. Others schemes, notably including ndzip, have no tunable behavior.

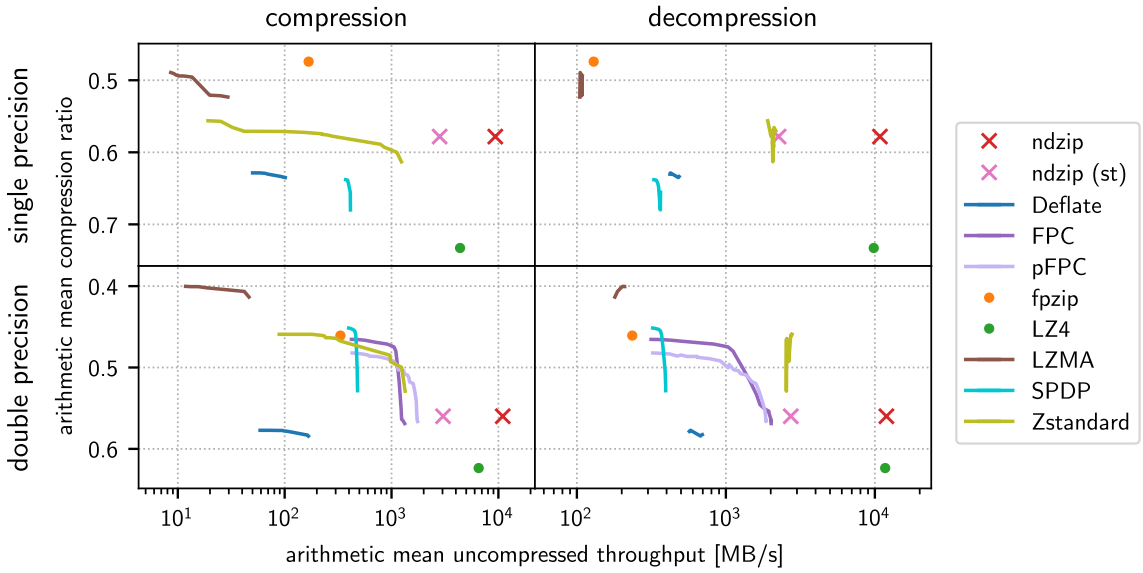

Figure 6: Compression ratio and throughput achieved by examined compressors

We observe that general-purpose algorithms can achieve high compression ratios on floating-point data, but only at the expense of significant computational resources. LZMA achieves the highest compression ratio on double-precision values and rivals the strongest single-precision compressor fpzip, while spending almost 90 minutes compressing our largest dataset from Figure 4. LZ4 achieves higher compression and decompression throughput than any other single-threaded algorithm reviewed, while also delivering the worst data reduction. Zstandard provides an exceptionally good trade-off, dominating Deflate and the specialized SPDP on single-precision data.

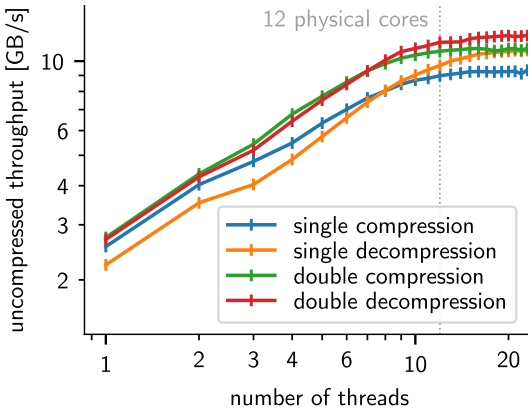

Figure 7: ndzip scaling up to 24 threads

Most specialized algorithms are able to outperform general-purpose schemes in at least one dimension. fpzip is the strongest single-precision compressor at the cost of only moderate throughput. For double-precision datasets, SPDP and FPC are able to slightly outperform Zstandard on compression but lose the throughput comparison for decompression, while fpzip provides no significant advantage in either step.

ndzip is the fastest specialized compressor and decompressor by a significant margin, with single-threaded execution ("st") achieving 2.8 GB/s (3.0 GB/s) in compression throughput and 2.2 GB/s (2.7 GB/s) decompression throughput for single (double) precision datasets. Thread-parallel execution peaked between 9 and 12 GB/s. On our test system, the maximum input bandwidth for a multithreaded, uncompressed copy operation is 16.3 GB/s as measured by the STREAM benchmark[7].

[7] `https://www.cs.virginia.edu/stream`

Our compressor delivers a lower compression ratio compared to some slower algorithms, but significantly outperforms its only throughput rival LZ4 in that regard.

Figure 7 shows the throughput of parallel ndzip scaling with the thread count, up to the 24 hardware threads of our test system.

Conclusion and Future Work

We demonstrated that by designing a specialized compression algorithm with characteristics of the target architecture in mind, excellent resource usage and a very competitive trade-off between compression rate and throughput can be achieved.

Based on our novel data-parallel Integer Lorenzo Transform of small hypercubes and a hardware-friendly residual coding scheme, the ndzip compressor makes use of both SIMD and thread parallelism to achieve compression and decompression speeds in excess of 10 GB/s on consumer hardware. Evaluation with real-world floating-point data shows that significant data reduction is possible with this approach.

Our implementation of ndzip is publicly available on GitHub[8].

With its small internal state and minimal synchronization requirements, a future GPU implementation of ndzip could be a great candidate for speeding up data transfers from and to devices by increasing the effective uncompressed bandwidth.

Acknowledgements

This research is supported by the FFG Bridge project INPACT (868018) and the D–A–CH project CELERITY, funded by FWF project I3388.

References

[1] N. Jain, A. Bhatele, S. White, T. Gamblin, and L. V. Kale, "Evaluating HPC networks via simulation of parallel workloads," in *SC '16: Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 154–165.

[2] P. Lindstrom and M. Isenburg, "Fast and efficient compression of floating-point data," *IEEE Transactions on Vis. and Comp. graphics*, vol. 12, no. 5, pp. 1245–1250, 2006.

[3] L. Ibarria, P. Lindstrom, J. Rossignac, and A. Szymczak, "Out-of-core compression and decompression of large n-dimensional scalar fields," in *Computer Graphics Forum*. Wiley Online Library, 2003, vol. 22, pp. 343–348.

[4] M. Burtscher and P. Ratanaworabhan, "FPC: A high-speed compressor for double-precision floating-point data," *IEEE Tr. on Computers*, vol. 58, no. 1, pp. 18–31, 2008.

[5] M. Burtscher and P. Ratanaworabhan, "pFPC: A parallel compressor for floating-point data," in *2009 Data Compression Conference*. IEEE, 2009, pp. 43–52.

[6] S. Claggett, S. Azimi, and M. Burtscher, "SPDP: An automatically synthesized lossless compression algorithm for floating-point data," in *2018 DCC*. IEEE, 2018, pp. 335–344.

[7] A. Yang, H. Mukka, F. Hesaaraki, and M. Burtscher, "MPC: a massively parallel compression algorithm for scientific data," in *2015 IEEE International Conference on Cluster Computing*. IEEE, 2015, pp. 381–389.

[8] N. Fout and K. Ma, "An adaptive prediction-based approach to lossless compression of floating-point volume data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 12, pp. 2295–2304, 2012.

[9] F. Knorr, P. Thoman, and T. Fahringer, "Datasets for Benchmarking Floating-Point Compressors," *arXiv e-prints*, p. arXiv:2011.02849, Nov. 2020.

[8] `https://github.com/fknorr/ndzip`