MPC: A Massively Parallel Compression Algorithm for Scientific Data

Annie Yang, Hari Mukka, Farbod Hesaaraki, and Martin Burtscher
Department of Computer Science
Texas State University
San Marcos, TX 78666
{ayang, mmsanthuhari, hesaaraki, burtscher}@txstate.edu

Abstract—**Due to their high peak performance and energy efficiency, massively parallel accelerators such as GPUs are quickly spreading in high-performance computing, where large amounts of floating-point data are processed, transferred, and stored. Such environments can greatly benefit from data compression if done sufficiently quickly. Unfortunately, most conventional compression algorithms are unsuitable for highly parallel execution. In fact, it is generally unknown how to design good compression algorithms for massively parallel systems. To remedy this situation, we study 138,240 lossless compression algorithms for single- and double-precision floating-point values that are built exclusively from easily parallelizable components. We analyze the best of these algorithms, explain why they compress well, and derive the Massively Parallel Compression (MPC) algorithm from them. This novel algorithm requires almost no internal state, achieves heretofore unreached compression ratios on several data sets, and roughly matches the best CPU-based algorithms in compression ratio while outperforming them by one to two orders of magnitude in throughput.**

Keywords - Lossless data compression; floating-point compression; massively parallel architectures; algorithm design; GPUs

I. INTRODUCTION

HPC cluster applications often process and transfer large amounts of floating-point data. For example, many simulations exchange data between compute nodes and with mass-storage devices after every time step. Most HPC programs retrieve and store large data sets, some of which may have to be sent to other locations for additional processing, analysis, or visualization. Furthermore, scientific programs often save checkpoints at regular intervals.

Compression can reduce the amount of data that needs to be transmitted and/or stored. However, if the overhead lowers the effective throughput, compression will not be used in the performance-centric HPC domain. Hence, the challenge is to maximize the compression ratio while meeting or exceeding the available transfer bandwidth. In other words, the compression and decompression have to be done in real time. Moreover, the compression should be lossless and single pass. Intermediate program results that are exchanged between compute nodes, for example, generally cannot be lossy. A single-pass algorithm is needed so that the data can be compressed and decompressed in a streaming fashion as they are being generated and consumed, respectively.

Some compression algorithms are asymmetric, meaning that compression takes much longer than decompression. This is useful in situations where data are compressed once and decompressed many times. However, this is not the case for checkpoints, which are almost never read, nor for intermediate program results that are compressed to boost the transmission speed between compute nodes, between accelerators and hosts, or between compute nodes and storage devices. Thus, we focus on symmetric algorithms in this paper.

Massively parallel compute GPUs are quickly spreading in HPC environments to accelerate calculations as GPUs not only provide much higher peak performance than multicore CPUs but are also more cost and energy efficient. However, utilizing GPUs for data compression is difficult because compression algorithms typically compress data based on information from previously processed words. This makes compression and decompression data-dependent operations that are difficult to parallelize. Thus, most of the relatively few parallel compression approaches from the literature simply break the data up into chunks that are compressed independently using a serial algorithm. However, this technique is not suitable for massively parallel hardware. First, the data would have to be broken up into hundreds of thousands of small chunks, at least one per thread, thus possibly losing much of the history needed to compress the data well. Second and more importantly, well-performing serial compression algorithms generally require a large amount of internal state (e.g., predictor tables or dictionaries), making it infeasible to run tens of thousands of them in parallel. As a consequence, the research community does not yet possess a good understanding of how to design effective compression algorithms for highly parallel machines.

At a high level, most data compression algorithms comprise two main steps, a data model and a coder. Roughly speaking, the goal of the model is to accurately predict the data. The residual (i.e., the difference) between each actual value and its predicted value will be close to zero if the model is accurate for the given data. This residual sequence of values is then compressed with the coder by mapping the residuals in such a way that frequently encountered values or patterns produce shorter output than infrequently encountered data. The reverse operations are performed to decompress the data. For instance, an inverse model takes the residual sequence as input and regenerates the original values as output.

To systematically search for effective and massive-parallelism-friendly compression algorithms, we synthesized a large number of compressors and their corresponding decompressors using the following approach. We started with a detailed study of previously proposed floating-point compression algorithms, broke them down into their constituent parts, rejected all parts that could not be parallelized well, and generalized the remaining parts as much as possible. This yielded a number of *algorithmic components* for building data models and coders. We then implemented each component using a common interface, i.e., each component can be given a block of data as input, which it transforms into an output block of data. This makes it possible to *chain* the components, allowing us to generate a vast number of compression-algorithm candidates from a given set of components. Note that each component comes with an inverse that performs the opposite transformation. Thus, for any chain of components, which represents a compression algorithm, we can synthesize the matching decompressor. Figure 1 illustrates this approach on the example of the four components named LNV6s, BIT, LNV1s, and ZE that make up the 6D version of our Massively Parallel Compression (MPC) algorithm.

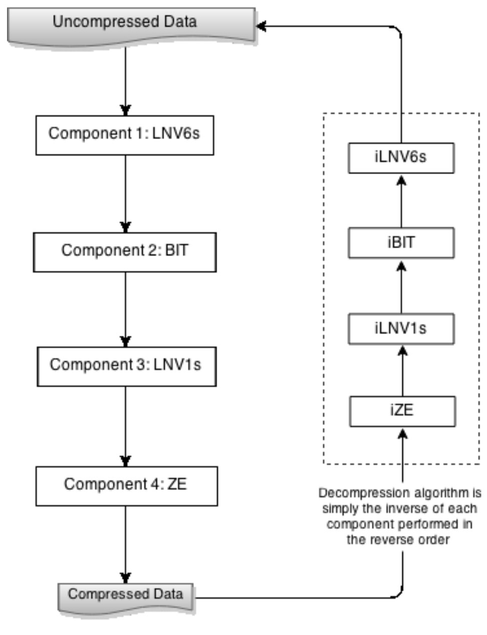

Figure 1. The four chained components that make up the six-dimensional MPC compression algorithm along with the corresponding four inverse components that make up the decompression algorithm

We use exhaustive search to determine the most effective compression algorithms that can be built from the available components. Limiting the components to those that can exploit massively parallel hardware guarantees that all of the compressors and decompressors synthesized in this way are, by design, GPU-friendly.

Since floating-point computations are prevalent on highly parallel machines and floating-point data tend to be difficult to compress, we decided to target this domain. In particular, we implemented 24 highly parallel components in the CUDA C++ programming language for GPUs and employed our approach on single- and double-precision versions of 13 real-world data sets. Based on a detailed analysis and generalization of the best four-stage compression algorithms we found for each data set as well as the best overall algorithm, we were able to derive the MPC algorithm that works well on many different types of floating-point data.

MPC treats double- and single-precision floating-point values as 8- or 4-byte integers, respectively, and exclusively uses integer instructions for performance reasons as well as to avoid the possibility of floating-point exceptions or rounding inaccuracies. This means that positive and negative zeros and infinities, not-a-number (NaN), denormals, and all other possible floating-point values are fully supported.

The first stage of MPC subtracts the n^{th} prior value from the current value to produce a residual sequence, where n is the dimensionality of the input data. The second stage rearranges the residuals by bit position, i.e., it emits all the most significant bits of the residuals packed into words, followed by the second-most significant bits, and so on. The third stage computes the residual of the consecutive words holding these bits. The fourth stage compresses the data by eliminating zero words. The precise operation of MPC, how it was derived, and why it works well are explained in Sections 4.D and 5.A.

MPC is quite different from the floating-point compression algorithms in the current literature. In particular, it requires almost no internal state, making it suitable both for a massively parallel software implementation as well as for a hardware implementation. On several of the studied data sets, MPC outperforms the general-purpose compressors bzip2, gzip, and lzop as well as the special-purpose compressors pFPC and GFC by up to 33% in compression ratio. Throughput evaluations show our CUDA implementation running on a single K40 GPU to be faster in all cases than even the parallel pFPC code running on twenty high-end Xeon CPU cores. Moreover, the double-precision throughput of MPC exceeds that of the PCI-Express bus linking the GPU to the CPU in our system, making real-time compression possible for results that are computed on a GPU before they are transferred to the host or the network interface card (NIC), and making real-time decompression possible of compressed data that are streamed to the GPU from the CPU or the NIC.

This paper makes the following main contributions.

- It presents the MPC lossless compression algorithm for single- and double-precision floating-point data that is suitable for massively parallel execution.
- It systematically evaluates 138,240 combinations of components to determine well-performing compression algorithms within the given search space.
- It analyzes the chains of components that work well to gain insight into the design of effective parallel compression algorithms and to predict how to adapt them to other data sets.
- It describes previously unknown algorithms that compress several real-world scientific numeric data sets significantly better than prior work.
- It demonstrates that, in spite of substantial constraints, MPC's compression ratios rival those of the best CPU-

based compressors while yielding much higher throughputs than (multicore) CPU-based compressors.

- It makes the CUDA implementation of MPC available at http://cs.txstate.edu/~burtscher/research/MPC/.

The rest of this paper is organized as follows. Section II describes contemporary GPU architecture and why it is more difficult to implement compression algorithms on such hardware than on CPUs. Section III summarizes related work. Section IV provides an overview of the system, the floating-point data sets, and the algorithmic components we use for our evaluation. Section V presents the MPC algorithm and discusses its performance. Section VI concludes the paper.

II. GPU Architecture

This section provides a brief overview of the architectural characteristics of the Kepler-based Tesla K40 compute GPU we use and explains the features that make it difficult to implement compression algorithms on such a massively parallel device. CUDA programs require hierarchical parallelization across threads as well as across thread blocks of up to 1024 threads. The K40 consists of 15 streaming multiprocessors (SMs) to which the thread blocks are mapped. Each SM contains 192 processing elements (PEs) for executing the threads. Whereas each PE can run an individual thread of instructions, sets of 32 PEs are tightly coupled and must either execute the same instruction (operating on different data) in the same cycle or wait. The corresponding sets of 32 coupled threads are called warps. Warps in which not all threads can execute the same instruction are subdivided by the hardware into sets of threads such that all threads in a set execute the same instruction. The individual sets are serially executed, which is called branch divergence, until they re-converge. To maximize performance, branch divergence has to be avoided, but it is generally difficult to implement compression algorithms in a manner such that sets of 32 threads always follow the same control flow.

The K40's memory subsystem is also built for warp-based processing. If the threads in a warp simultaneously access words in main memory that lie in the same aligned 128-byte segment, the hardware merges the 32 reads or writes into one coalesced memory transaction that is as fast as accessing a single word. Warps accessing multiple 128-byte segments result in correspondingly many individual memory transactions that are executed serially. Hence, uncoalesced accesses are slower, but it is unclear how to write compression algorithms such that sets of 32 threads always access words from the same 128-byte segment.

III. RELATED WORK

A. *Floating-Point Compressors*

This section summarizes related work on lossless floating-point compression. We extracted the basic idea behind many of our algorithmic components from these papers.

Lindstrom and Isenburg discuss real-time compression of floating-point grid data for speeding up I/O operations [1]. They use a Lorenzo predictor and map reals to unsigned integers. We also exclusively use integer representation and operations in MPC.

Burtscher and Ratanaworabhan's FPC algorithm targets double-precision values [2]. It predicts the integer interpretation of the 64-bit values using an FCM and a DFCM predictor. The two predictions are XORed with the true value. The result with more leading zeros is compressed using leading-zero byte counts. The authors also published a parallel version of their compression algorithm, called pFPC [3], with which we compare our approach in the result section. We did not include the FCM and DFCM predictors in our study because they require large tables, which is problematic in a massively parallel implementation where every thread would need such a table. However, we adopted the XOR idea.

Chen et al.'s work orders grid points of tetrahedral volume data to improve compressibility [4]. Their approach separates the "signed exponent" from the mantissa values. We include a similar component that groups the various bit positions from adjacent values so that all the sign bits, exponent bits, etc. can be compressed together.

Bicer et al. describe a framework that XORs values and leading-zero compresses the results [5]. As it operates at bit granularity, their approach works for both single- and double-precision data. The data are split into chunks, which are compressed independently. MPC also supports both single- and double-precision data and uses independent data chunks. However, each chunk is assigned to a different thread block in the GPU and all the threads in a block cooperatively compress and decompress the chunk.

Filgueira et al. focus on runtime compression of MPI messages, including floating-point messages [6]. They found lzop to work best on their synthetic integer and floating-point data that include a significant number of zeros because lzop is very fast. The user can select which compression algorithm to use for which data type. A later paper describes an extension that dynamically selects the most appropriate algorithm based on data type, including none for short messages [7]. Our approach is orthogonal to theirs and can be used to find good compression algorithms for various data types.

Schendel et al. introduce a pre-compression tool that improves the performance of general-purpose compressors on double-precision floating-point data. Their approach analyzes the compressibility of the data at byte-level granularity, determines the best compressor for the job and then identifies and removes hard-to-compress sections before piping the remaining data to the compressor [8]. Our approach searches for the best algorithms at word and byte granularity and produces standalone compression algorithms.

B. *Floating-Point Compression on GPUs*

There exists little prior work on GPU-based lossless floating-point compression. Balevic et al. designed a block-parallel arithmetic coder for post-processing scientific simulation data directly on the GPU before transfer back to the CPU [9]. Their approach achieved significant storage savings, but the compression overhead outweighed the resulting time savings.

Later, Balevic presented an algorithm for GPUs that is based on Huffman coding and that exploits atomic operations to enable variable-length code-word writes [10]. It relies on a parallel prefix scan to compute output positions. Several of our components also rely on prefix scans for parallelization.

O'Neil and Burtscher describe the GFC compression algorithm for GPUs [11]. We include this algorithm in our evaluation. GFC breaks up the data into chunks that are compressed independently by different warps. Our components work at the coarser granularity of thread blocks to retain larger "windows" of data, which greatly improves the resulting compression ratio.

Ozsoy and Swany implement LZSS for the GPU [12]. They achieve good speedups compared to serial LZSS implementations. However, their approach does not specifically target scientific data. LZSS also differs from MPC in that it requires tables and features a smaller window size.

While not targeting floating-point compression, the work by Patel et al. is related in so far as it also investigates the components of a compression algorithm and how to implement them efficiently on GPUs [13]. In particular, they study how to port BWT, MTF, and Huffman coding, which are the main components of the bzip2 algorithm. They found that none of these components are conducive to GPU parallelization, which demonstrates the need to find new algorithms.

C. *Generating Compression Algorithms*

Burtscher and Sam present TCgen, a tool that automatically generates customized trace compressors [14]. The user has to select one or more predictors for compression. TCgen then translates this description into C source code and optimizes it for the specified trace format and predictors. We use many more components in our study, including non-predictor components, all of which could be used to build a similar tool.

Kattan and Poli propose a system that employs genetic programming to find optimal ways to combine standard compression algorithms [15]. They group similar data chunks together and label each group with the best compression algorithm for its chunks. We also combine components. However, their components represent entire compression algorithms whereas our components are finer grained and represent parts of a compression algorithm.

Hsu and Zwarico describe an automatic synthesis technique for compressing heterogeneous files [16]. Each chunk of data is compressed using a different algorithm, which is determined using a statistical method. A compression history, required for decompression, is automatically generated and added in this phase. MPC only needs to record a single bit for the word size and a few bits for the dimensionality.

Fang et al.'s work is probably the closest to our approach. They investigate how to compress database information using GPUs to overcome the transfer overhead [17]. They employ a compression planner along with a cost model of their GPU to identify an optimal combination among nine different compression schemes. They use a rule-based method to automatically prune the search space. Our approach could benefit from their pruning techniques, but the results would no longer be guaranteed to be optimal within the search space. Also, they use fewer components and, as in Kattan and Poli's work, each component is an entire compression algorithm.

Note that chaining whole compression algorithms, as done in the above related works, is fundamentally different from chaining algorithmic components to build a compression algorithm, which is what we do. After all, the goal of a compression algorithm is to maximally reduce the number of bytes, which generally means that there are few exploitable patterns left in the output. This makes it difficult for the next compression algorithm in a chain to be effective. Our approach does not suffer from this problem. In fact, most of the algorithmic components we use do not reduce the number of bytes at all but transform the data to better expose patterns.

IV. METHODOLOGY

A. *System and Compilers*

We evaluated the tested compressors on one node of the Maverick supercomputer at the Texas Advanced Computing Center. Each compute node of Maverick contains two 10-core Intel Xeon E5-2680 v2 Ivy Bridge processors running at 2.8 GHz and 128 GB of main memory. The operating system is CentOS 6.4. We used the icc compiler version 14.0.1 with "-O3 -xhost -pthread" for the CPU implementations.

Each compute node further contains an Nvidia K40 GPU. It has 15 streaming multiprocessors with a total 2880 CUDA cores running at 745 MHz and 12 GB of global memory. We used nvcc version 6.0 with the "-O3 -arch=sm_35" flags to compile the GPU codes.

B. *Measuring Throughput*

For all special-purpose floating-point compressors, the timing measurements are performed by adding code to read a timer before and after the compression and decompression code sections and recording the difference. For the general-purpose compressors, we measure the runtime of compression and decompression when reading the input from a disk cache in main memory and writing the output to /dev/null. In the case of GPU code, we exclude the time to transfer the data to or from the GPU as we assume the data to have been produced there or to be needed there. Each experiment was conducted three times and the median throughput is reported. However, the three measured throughputs were very similar in all cases. The decompressed results are always compared to the original data to verify that every bit is identical.

C. *Data Sets*

We use the 13 FPC data sets for our evaluation [2]. Each data set consists of a binary sequence of IEEE 754 double-precision floating-point values. They include MPI messages (msg), numeric results (num), and observational data (obs).

MPI messages: These five datasets contain the numeric messages sent by a node in a parallel system running NAS Parallel Benchmark (NPB) and ASCI Purple applications.

- msg_bt: NPB computational fluid dynamics pseudo-application bt
- msg_lu: NPB computational fluid dynamics pseudo-application lu
- msg_sp: NPB computational fluid dynamics pseudo-application sp
- msg_sppm: ASCI Purple solver sppm
- msg_sweep3d: ASCI Purple solver sweep3d

Numeric simulations: These four datasets are the result of numeric simulations.

- num_brain: simulation of the velocity field of a human brain during a head impact
- num_comet: simulation of the comet Shoemaker-Levy 9 entering Jupiter's atmosphere

- num_control: control vector output between two minimization steps in weather-satellite data assimilation
- num_plasma: simulated plasma temperature evolution of a wire array z-pinch experiment

Observational data: These four datasets comprise measurements from scientific instruments.

- obs_error: data values specifying brightness temperature errors of a weather satellite
- obs_info: latitude and longitude information of the observation points of a weather satellite
- obs_spitzer: data from the Spitzer Space Telescope showing a slight darkening as an extrasolar planet disappears behinds its star
- obs_temp: data from a weather satellite denoting how much the observed temperature differs from the actual contiguous analysis temperature field

TABLE I. INFORMATION ABOUT THE DOUBLE-PRECISION DATA SETS

	size (megabytes)	doubles (millions)	unique values (percent)	1st order entropy (bits)	randomness (percent)
msg_bt	254.0	33.30	92.9	23.67	95.1
msg_lu	185.1	24.26	99.2	24.47	99.8
msg_sp	276.7	36.26	98.9	25.03	99.7
msg_sppm	266.1	34.87	10.2	11.24	51.6
msg_sweep3d	119.9	15.72	89.8	23.41	98.6
num_brain	135.3	17.73	94.9	23.97	99.9
num_comet	102.4	13.42	88.9	22.04	93.8
num_control	152.1	19.94	98.5	24.14	99.6
num_plasma	33.5	4.39	0.3	13.65	99.4
obs_error	59.3	7.77	18.0	17.80	87.2
obs_info	18.1	2.37	23.9	18.07	94.5
obs_spitzer	189.0	24.77	5.7	17.36	85.0
obs_temp	38.1	4.99	100.0	22.25	100.0

Table I provides pertinent information about each data set. The first two data columns list the size in megabytes and in millions of double-precision values. The middle column shows the percentage of values that are unique. The fourth column displays the first-order entropy of the values in bits. The last column expresses the randomness of each data set in percent, i.e., it reflects how close the first-order entropy is to that of a truly random data set with the same number of unique values. For the single-precision experiments, we simply converted the double-precision data sets.

D. Algorithmic Components

We tested the following algorithmic components in our experiments. They are generalizations or approximations of components extracted from previously proposed compression algorithms. Each component takes a block of data as input, transforms it, and outputs the transformed block.

The input data are broken down into fixed-size chunks. Each chunk is assigned to a thread block for parallel processing. We chose 1024-element chunks to match the maximum number of threads per thread block in our GPU.

The **NUL** component simply outputs the input block. Its purpose will be explained below. The **INV** component flips all the bits. The **BIT** component breaks a block of data into chunks and then emits the most significant bit of each word in the chunk, followed by the second most significant bits, and so on. The **DIM*n*** component also breaks the blocks into chunks and then rearranges the values in each chunk such that the values from each dimension are grouped together. For example, DIM2 emits all the values from the even positions first and then all the values from the odd positions. We tested the dimensions ***n*** = 2, 3, 4, 5, 8, 16, and 32. The **LNV*n*s** component uses the ***n***th prior value in the same chunk as a prediction of the current value, subtracts the prediction from the current value, and emits the residual. The **LNV*n*x** component is identical except it XORs the prediction with the current value to form the residual. In both cases, we tested ***n*** = 1, 2, 3, 5, 6, and 8. Note that all of the above components transform the data blocks without changing their size. The following two components are the only ones that can actually reduce the length of a data block. The **ZE** component outputs a bitmap for each chunk that specifies which values in the chunk are zero. The bitmap is followed by the non-zero values. The **RLE** component performs run-length encoding, i.e., it replaces repeating values by a count and a single copy of the value. Each component has a corresponding inverse that performs the opposite transformation for decompression.

Since it may be more effective to operate at byte rather than word granularity, we also include the singleton pseudo component "**|**", which we call the cut, that converts a block of words into a block of bytes through type casting (i.e., no computation is necessary). As a result, we need three versions of each component and its inverse, one for double-precision values (8-byte words), one for single-precision values (4-byte words), and one for byte values. Each component operates on an integer representation of the floating-point data, i.e., the bit pattern representing the floating-point value is copied verbatim into an appropriately sized integer variable.

We chose this limited number of components because we only included components that we could implement in a massively parallel manner. Nevertheless, as the results in the next section show, very effective compression algorithms can be created from these components. In other words, the sophistication and effectiveness of the ultimate algorithm is the result of the clever combination of components, not the capability of each individual component. This is akin to how complex programs can be expressed through a suitable sequence of very simple machine instructions.

We investigate all four-stage compression algorithms that can be built from the above components. Due to the presence of the NUL component, this includes all one-, two-, and three-stage algorithms as well. Note that only the first three stages can contain any one of the 24 components described above. The last stage must contain a component that can reduce the amount of data, that is, either ZE or RLE. The cut can be before the first component, in which case the data is exclusively treated as a sequence of bytes, after the last component, in which case the data is exclusively treated as a sequence of words, or between components, in which case the data is initially treated as words and then as bytes. The five possible locations for the cut, the 24 possible components in each of the first three stages, and the two possible components in the last stage results in 5*24*24*24*2 = 138,240 possible compression algorithms that we evaluate.

TABLE II. BEST FOUR-STAGE ALGORITHM FOR EACH DATA SET, FOR ALL SINGLE-PRECISION DATA SETS, AND FOR ALL DOUBLE-PRECISION DATA SETS

	double precision		single precision	
dataset	CR	4-stage algorithm with cut	CR	4-stage algorithm with cut
msg_bt	1.143	`LNV1s BIT LNV1s ZE |`	1.233	`DIM5 ZE LNV6x | ZE`
msg_lu	1.244	`LNV5s | DIM8 BIT RLE`	1.588	`LNV5s LNV5s LNV5x | ZE`
msg_sp	1.192	`DIM3 LNV5x BIT ZE |`	1.362	`DIM3 LNV5x BIT ZE |`
msg_sppm	3.359	`DIM5 LNV6x ZE | ZE`	4.828	`RLE DIM5 LNV6s ZE |`
msg_sweep3d	1.293	`LNV1s DIM32 | DIM8 RLE`	1.545	`LNV1s DIM32 | DIM4 RLE`
num_brain	1.182	`LNV1s BIT LNV1s ZE |`	1.344	`LNV1s BIT LNV1s ZE |`
num_comet	1.267	`LNV1s BIT LNV1s ZE |`	1.199	`LNV1s | DIM4 BIT RLE`
num_control	1.106	`LNV1s BIT LNV1s ZE |`	1.122	`LNV1s BIT LNV1s ZE |`
num_plasma	1.454	`LNV2s LNV2s LNV2x | ZE`	1.978	`LNV2s LNV2s LNV2x | ZE`
obs_error	1.210	`LNV1x ZE LNV1s ZE |`	1.289	`LNV6s BIT LNV1s ZE |`
obs_info	1.245	`LNV2s | DIM8 BIT RLE`	1.477	`LNV8s DIM2 | DIM4 RLE`
obs_spitzer	1.231	`ZE BIT LNV1s ZE |`	1.080	`ZE BIT LNV1s ZE |`
obs_temp	1.101	`LNV8s BIT LNV1s ZE |`	1.126	`BIT LNV1x DIM32 | RLE`
Best	1.214	`LNV6s BIT LNV1s ZE |`	1.265	`LNV6s BIT LNV1s ZE |`

V. EXPERIMENTAL RESULTS

A. *Synthesis Analysis and Derivation of MPC*

Table II shows the four chained components and the location of the cut that the exhaustive search found to work best for each data set as well as across all 13 data sets, which is denoted as "Best". For Best, the compression ratio (CR) is the harmonic mean over the data sets.

1) *Observations about Individual Algorithms*

The individual best algorithms are truly the best, i.e., the next best algorithms compress successively worse (by a fraction of a percent). Hence, it is not the case that an entire set of algorithms performs equally well.

Whereas the last component (ignoring the cut) has to be ZE or RLE, ZE is chosen more often. This indicates that the earlier components manage to transform the data in a way that generates zeros but not many zeros in a row, which RLE would be better able to compress.

Interestingly, in most cases, the first three stages do not include any ZE or RLE component, i.e., they do not change the length of the data stream but transform it to make the final stage more effective. Clearly, these non-compressing transformations are very important, emphasizing that the best overall algorithm is generally not the one that maximally compresses the data in every stage. This also demonstrates that chaining whole compression algorithms, as proposed in some related work, is unlikely to yield effective algorithms.

There are several instances of DIM8 right after the cut in the double-precision algorithms and of DIM4 right after the cut in the single-precision algorithms. This surprised us as it utilizes the DIM component differently than anticipated. Instead of employing it for multi-dimensional data sets, these algorithms use DIM to separate the different byte positions from each four- or eight-byte word. Note that the frequently used BIT component serves a similar purpose but at a finer granularity. This repurposing of the DIM component shows that automatic synthesis is able to devise algorithms that the authors of the components may not have foreseen.

The very frequent occurrence of BIT indicates that the individual bits of floating-point values tend to correlate more strongly across values than within values. This might be expected as, for example, the top exponent bits of consecutive values are likely to all be the same.

Another interesting observation is that the cut is often at the end, i.e., it is not used. This means that the entire algorithm operates at word granularity, including the Best algorithm, and that there is no benefit from switching to byte granularity. This is good news because it simplifies and speeds up the implementation as only word-granular components are needed. NUL and INV are also not needed. Clearly, inverting all the bits is unnecessary and algorithms with fewer than four components (i.e., that include NUL) compress less well.

The LNV predictor component is obviously very important. Every listed algorithm includes it, and most of them include two such components. LNV comes in two versions, one that uses integer subtraction to form the residual and the other that uses XOR (i.e., bitwise subtraction). Subtraction is much more frequent, which is noteworthy as the current literature seems undecided as to which method to prefer.

In many cases, the single-precision algorithm is the same as the corresponding double-precision algorithm, especially when excluding the aforementioned DIM4 versus DIM8 difference immediately after the cut. This similarity is perhaps expected since the data sets contain the same values (albeit in different formats). However, in about half the cases, the algorithms are different, sometimes substantially (e.g., on msg_bt) and yield significantly different compression ratios. This implies that the bits that are dropped when converting from double to single precision benefit from different compression algorithms than the remaining bits. Hence, it would probably be advantageous if distinct algorithms were used for compressing the bits or bytes at different positions within the floating-point words.

2) *Observations about the Best Algorithm*

Focusing on the Best algorithm, which is the algorithm shown in Figure 1, we find that the single- and double-precision data sets result in the same algorithm that maximizes the harmonic-mean compression ratio. Hence, the following discussion applies to both formats. The most frequent pattern of components in the individual algorithms is "LNV*s BIT

LNV1s ZE", where the star represents a digit. Consequently, it is not surprising that the Best algorithm also follows this pattern. It is interesting, however, that Best uses a "6" in the starred position even though, with one exception, none of the individual algorithms do. We believe that the exhaustive search selected a six because six is the least common multiple of one, two, and three, all of which occur more often than six. In other words, the first component tries to predict each value using a similar prior value, which is best done when looking back *n* positions, where *n* is the least common multiple of the dimensionality of the various data sets. Hence, it is likely that larger *n* will work better, but we only tested up to $n = 8$.

With this in mind, we can now explain the operation of the Best algorithm. The job of the LNV6s component is to predict each value using a similar prior value to obtain a residual sequence with many small values. Since not all bit positions are equally predictable (e.g., the most significant exponent bits are more likely than the other bits to be predicted correctly and to therefore be zero in the residual sequence), it is beneficial to group bits from the same bit position together, which is what the BIT component does. The resulting sequence of values apparently contains consecutive "words" that are identical, which the LNV1s component turns into zeros. The ZE component then eliminates these zeros.

3) *Derivation of the MPC Algorithm*

With this insight, a good compression algorithm for floating-point data can be derived by adapting the first component to the dimensionality of the data set and keeping the other three components fixed. We named the resulting algorithm MPC for "Massively Parallel Compressor". It is based on the aforementioned "LNV*s BIT LNV1s ZE" pattern but uses the data-set dimensionality in the starred location. Note that we obtained the same pattern when using cross-validation, i.e., when excluding one of the inputs. MPC is identical to the best algorithm the exhaustive search found for several of the studied data sets. Even better, in cases where the actual dimensionality is above eight, MPC yields compression ratios exceeding those of the Best algorithm (cf. the Best results in Table II vs. the MPC results in Tables III and IV), which validates our generalization of the Best algorithm.

B. *Compression Ratios*

Tables III and IV show the compression ratios on the double- and single-precision data sets, respectively, for the general-purpose compressors bzip2, gzip, and lzop, the special-purpose floating-point compressors pFPC and GFC (they only support double precision), and for MPC. The highest compression ratio for each data set is highlighted in the tables. Since we want to maximize the compression ratio, we selected the command-line flags that result in the highest compression ratio where possible. For GFC and MPC, we specify the true data-set dimensionality on the command line.

GFC and MPC are parallel GPU implementations. pFPC is a parallel CPU implementation. The remaining compressors are serial CPU implementations.

All of the tested algorithms except lzop and GFC compress at least one data set best. MPC delivers the highest compression ratio on 5 double-precision and 8 single-precision data sets. This is a rather surprising result given that MPC is "handicapped" by being constrained to only utilize GPU-friendly components that retain almost no internal state. Only pFPC with 1-million-entry tables matches MPC in the number of data sets (5) on which it yields the highest compression ratio. However, MPC not only requires much less memory but also supports single-precision data, which pFPC does not. In fact, the pFPC algorithm is not suitable for a single-precision implementation.

MPC is clearly superior to lzop and GFC, both of which it outperforms on almost every data set in terms of compression ratio. Moreover, GFC also does not support single-precision data. pFPC and bzip2 outperform MPC on average. However, this is only the case because of two data sets, msg_sppm and num_plasma, on which they yield much higher compression ratios than MPC.

Most of the double-precision data sets are less compressible than the single-precision data sets. This is expected as the least significant mantissa bits tend to be the most random in floating-point values and many of those bits are dropped when converting from double to single precision.

TABLE III. COMPRESSION RATIOS ON THE DOUBLE-PRECISION DATA SETS

	HarMean	msg_bt	msg_lu	msg_sp	msg_sppm	msg_sweep3d	num_brain	num_comet	num_control	num_plasma	obs_error	obs_info	obs_spitzer	obs_temp
bzip2 --best	*1.321*	1.088	1.018	1.055	6.933	1.294	1.043	1.173	1.029	5.789	1.339	1.217	1.752	1.024
gzip --best	*1.239*	1.130	1.055	1.107	7.431	1.092	1.064	1.162	1.058	1.608	1.448	1.154	1.231	1.036
lzop -9	*1.158*	1.052	1.000	1.003	6.780	1.017	1.000	1.082	1.017	1.503	1.273	1.096	1.142	1.000
pFPC -1M	*1.365*	1.250	1.137	1.238	4.710	1.888	1.148	1.151	1.038	7.042	1.542	1.215	1.022	0.997
GFC	*1.179*	1.122	1.148	1.202	3.506	1.217	1.090	1.110	1.013	1.125	1.233	1.141	1.022	1.037
MPC	*1.248*	1.207	1.212	1.208	2.999	1.287	1.182	1.267	1.106	1.164	1.180	1.214	1.184	1.101

TABLE IV. COMPRESSION RATIOS ON THE SINGLE-PRECISION DATA SETS

	HarMean	msg_bt	msg_lu	msg_sp	msg_sppm	msg_sweep3d	num_brain	num_comet	num_control	num_plasma	obs_error	obs_info	obs_spitzer	obs_temp
bzip2 --best	*1.398*	1.129	1.041	1.141	8.741	2.355	1.113	1.117	1.043	8.652	1.338	1.327	1.394	1.049
gzip --best	*1.267*	1.179	1.086	1.200	9.605	1.151	1.128	1.151	1.080	1.383	1.466	1.200	1.188	1.079
lzop -9	*1.153*	1.075	1.000	1.083	8.634	1.033	1.003	1.086	1.016	1.223	1.246	1.129	1.077	1.000
MPC	*1.350*	1.336	1.440	1.385	3.813	1.534	1.344	1.178	1.122	1.345	1.298	1.436	1.047	1.114

C. *Compression and Decompression Speed*

Table V lists the average throughput in gigabytes per second on the 13 data sets for all tested compressors. As stated before, GFC and pFPC do not support single-precision data.

TABLE V. AVERAGE COMPRESSION AND DECOMPRESSION THROUGHPUT IN GIGABYTES PER SECOND

	double precision		single precision	
	compr.	decom.	compr.	decom.
bzip2 --best	0.01	0.02	0.01	0.02
gzip --best	0.02	0.15	0.03	0.15
lzop -9	0.01	1.93	0.01	1.43
pFPC -1M	1.43	1.05	n/a	n/a
GFC	32.28	31.47	n/a	n/a
MPC	10.78	7.91	5.81	4.23

lzop is the fastest CPU-based algorithm at decompression, but it is still three to four times slower than MPC. gzip is the fastest general-purpose compression CPU-based algorithm at compression, but it is hundreds of times slower than MPC. Thus, MPC not only compresses better than these algorithms but also greatly outperforms them in throughput.

pFPC, the overall best implementation in terms of compression ratio (on the double-precision data), is 7.5 times slower when running on 20 CPU cores than MPC. Yet, MPC compresses nearly as well. GFC is the fastest tested implementation (on the double-precision data sets). It is three to four times faster than MPC, which is expected as GFC is essentially a two-component algorithm that does not require prefix sums. Recall, however, that GFC compresses poorly.

The double-precision throughput of MPC far exceeds that of the PCI-Express bus linking the GPU to the CPU in our system, making real-time compression and decompression possible. The single-precision compression throughput approximately matches the PCI throughput, but decompression is a little slower. Note, however, that the evaluated MPC implementation is synthesized from general component code that is not optimized for this particular algorithm.

D. *Component Count*

Figure 2 illustrates by how much the compression ratio is lowered when reducing the number of chained components on the double-precision data sets. The bars marked "Best" refer to the single algorithm that yields the highest harmonic-mean compression ratio over all thirteen data sets. All other bars refer to the best algorithm found using exhaustive search for each individual data set. The single-precision results are not shown as they are qualitatively similar.

Except on two data sets, a single component suffices to reach roughly 80% to 90% of the compression ratio achieved with four components. Moreover, the increase in compression ratio when going from one to two stages is generally larger than going from two to three stages, which in turn is larger than going from three to four stages. In other words, the improvements start to flatten out, indicating that large numbers of stages are unlikely to yield much higher compression ratios (and would be slower) than the presented four-stage results. Note that, due to the presence of the NUL component, the algorithms that can be created with a larger number of stages represent a strict superset of the algorithms that can be created with fewer stages. As a consequence, the results in Figure 2 monotonically increase with more stages.

VI. SUMMARY, CONCLUSIONS & FUTURE WORK

The goal of this work is to determine whether effective algorithms exist for floating-point data compression that are suitable for massively parallel architectures. To this end, we evaluated 138,240 possible combinations of 24 GPU-friendly algorithmic components to find the best four-stage algorithm for each tested data set as well as the best algorithm for all thirteen data sets together, both for single- and double-preci-

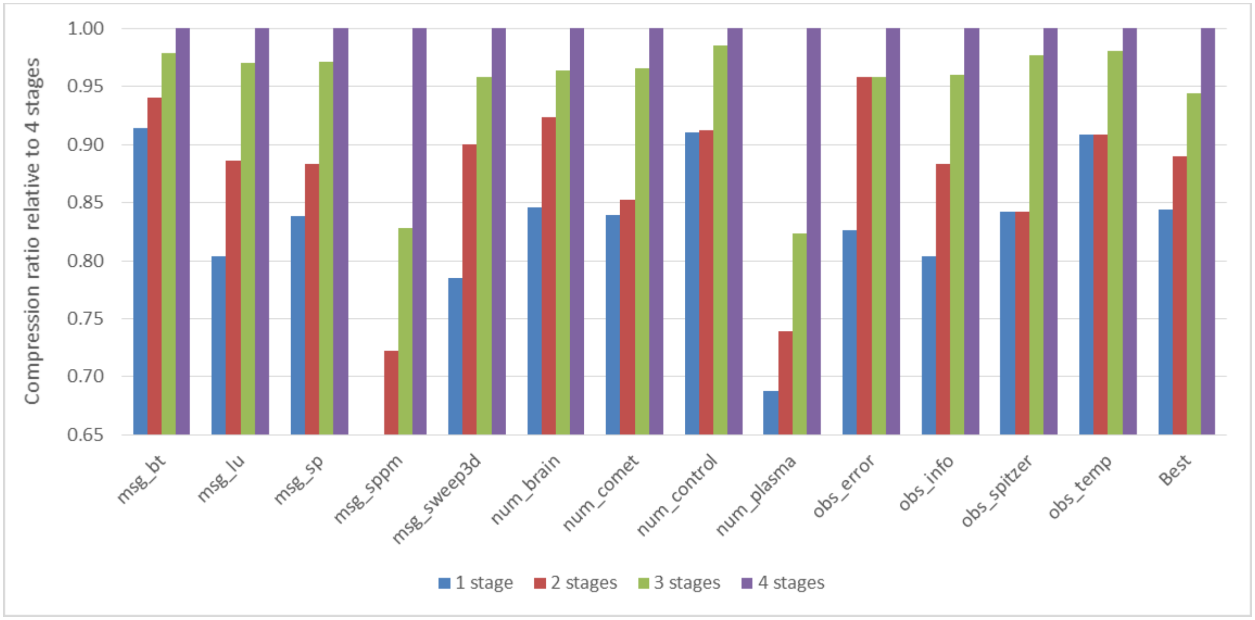

Figure 2. Best exhaustive-search-based double-precision compression ratios with 1, 2, and 3 stages relative to the best compression ratios with 4 stages; the y axis does not start at zero; the cut-off bar for msg_sppm reaches 0.4

sion representation. This study resulted in well-performing algorithms that have never before been described. A detailed analysis thereof yielded important insight that helped us understand why and how these automatically synthesized algorithms work. This, in turn, enabled us to make predictions as to which algorithms will likely work well on other data sets, which ultimately led to our MPC algorithm. It constitutes a generalization of the best algorithms we found using exhaustive search and requires only two generally known parameters about the input data: the word size (single- or double-precision) and the dimensionality.

It rivals the compression ratios of the best CPU-based algorithms, which is surprising because MPC is limited to using only algorithmic components that can be easily parallelized and that do not use much internal state. In contrast, some of the CPU compressors we tested utilize megabytes of internal state per thread. We believe the almost stateless operation of MPC make it a great algorithm for any highly parallel compute device as well as for FPGAs and hardware implementations, for instance in a network interface card.

Our open-source implementation of MPC, which is available at http://cs.txstate.edu/~burtscher/research/MPC/, greatly outperforms all other tested algorithms that compress similarly well, both in compression and in decompression throughput. Moreover, the throughput is, in most cases, sufficient for real-time compression/decompression of data transmitted over the PCIe bus.

Clearly, highly parallel, effective compression algorithms for floating-point data sets exist. Determining whether this is also the case for other types of data is important future work. To boost the throughput of MPC, we intend to replace the synthesized, component-based implementation with a tailored and tuned CUDA implementation. Should that not be enough to match the PCI throughput even for single-precision data, we may design a faster three-component algorithm that compresses a little less. To further improve the compression ratio, especially on the two data sets where other algorithms perform significantly better, we will try to add other GPU-friendly algorithmic components.

ACKNOWLEDGMENTS

This work was supported by the U.S. National Science Foundation under grants 1141022, 1217231, 1406304, and 1438963, a REP grant from Texas State University, and grants and gifts from NVIDIA Corporation. The authors acknowledge the Texas Advanced Computing Center for providing the HPC resources used in this study.

REFERENCES

[1] P. Lindstrom and M. Isenburg. “Fast and Efficient Compression of Floating-Point Data.” *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1245-1250. 2006.

[2] M. Burtscher and P. Ratanaworabhan. “FPC: A High-Speed Compressor for Double-Precision Floating-Point Data.” *IEEE Transactions on Computers*, 58(1):18-31. 2009.

[3] M. Burtscher and P. Ratanaworabhan. “pFPC: A Parallel Compressor for Floating-Point Data.” *Data Compression Conference*, pp. 43-52. 2009.

[4] D. Chen, Y.-J. Chiang, N. Memon, and X. Wu. “Lossless Geometry Compression for Steady-state and Time-varying Irregular Grids.” *IEEE Symposium on Visualization*, pp. 275-282. 2006.

[5] T. Bicer, J. Yiny, D. Chiuz, G. Agrawal, and K. Schuchardt. “Integrating Online Compression to Accelerate Large-Scale Data Analytics Applications.” *International Parallel and Distributed Processing Symposium*. 2013.

[6] R. Filgueira, D.E. Singh, A. Calderón, and J. Carretero. “CoMPI: Enhancing MPI-based Applications Performance and Scalability Using Run-Time Compression.” *EUROPVM/MPI*. 2009.

[7] R. Filgueira, D.E. Singh, J. Carretero, A. Calderón, and F. Garcia. “Adaptive-CoMPI: Enhancing MPI-based Applications - Performance and Scalability by using Adaptive Compression.” *International Journal of High Performance Computing Applications*, 25(1):93-114. 2011.

[8] E. R. Schendel, Y. Jin, N. Shah, J. Chen, C. S. Chang, S-H. Ku, S. Ethier, S. Klasky, R. Latham, R. B. Ross, and N. F. Samatova. “ISOBAR Preconditioner for Effective and High-throughput Lossless Data Compression.” *28th Annual IEEE International Conference on Data Engineering (ICDE),* pp. 138-149. 2012.

[9] A. Balevic, L. Rockstroh, M. Wroblewski, and S. Simon. “Using Arithmetic Coding for Reduction of Resulting Simulation Data Size on Massively Parallel GPGPUs.” *15th European PVM/MPI Users' Group Meeting*. Springer Verlag, pp. 295-302. 2008.

[10] A. Balevic. “Parallel Variable-length Encoding on GPGPUs.” *International Conference on Parallel Processing*. Springer-Verlag, pp. 26-35. 2009.

[11] M.A. O'Neil and M. Burtscher. “Floating-Point Data Compression at 75 Gb/s on a GPU.” *Workshop on General Purpose Processing Using GPUs*. 2011.

[12] A. Ozsoy and M. Swany. “CULZSS: LZSS Lossless Data Compression on CUDA.” *IEEE International Conference on Cluster Computing*, pp. 403-411. 2011.

[13] R. Patel, Y. Zhang, J. Mak, A. Davidson, and J. Owens. “Parallel Lossless Data Compression on the GPU.” *Innovative Parallel Computing*, pp. 1-9. 2012.

[14] M. Burtscher and N.B. Sam. “Automatic Generation of High-Performance Trace Compressors.” *International Symposium on Code Generation and Optimization*, pp. 229-240. 2005.

[15] A. Kattan and R. Poli. “Evolutionary Synthesis of Lossless Compression Algorithms with GP-zip3.” *IEEE Congress on Evolutionary Computation*, 1(8):18-23. 2010.

[16] W.H. Hsu and A.E. Zwarico. “Automatic Synthesis of Compression Techniques for Heterogeneous Files.” *Software: Practice and Experience*, 25(10):1097-1116. 1995.

[17] W. Fang, B. He, and Q. Luo. “Database Compression on Graphic Processors.” *Proceedings of the VLDB Endowment*, 3(1-2):670-680. 2010.