Adaptive Per-File Lossless Compression of Floating-Point Data

Andrew Rodriguez
Department of Computer Science
Texas State University
San Marcos, Texas, USA
andrew.rodriguez@txstate.edu

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

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

Abstract—**The large amount of floating-point data generated by scientific applications makes data compression essential for I/O performance and efficient storage. However, floating-point data is difficult to compress losslessly, and most compression algorithms are only effective on some files. In this paper, we study the benefit of compressing each file with a potentially different algorithm. For this purpose, we created AdaptiveFC, which is based on a tool that can chain data transformations together to generate millions of compression algorithms. AdaptiveFC uses a genetic algorithm to quickly identify an effective compressor in this vast search space for a given file. A comparison of AdaptiveFC to 15 leading lossless GPU compressors on 77 files from 6 datasets in the SDRBench suite shows that per-file compression yields higher compression ratios on average than any individual algorithm.**

Index Terms—**data compression, floating-point data, genetic algorithm, GPU processing**

I. Introduction

Scientific applications and instruments often emit huge amounts of floating-point data. For example, the Hardware/Hybrid Accelerated Cosmology Code (HACC) generates petabytes of data in a single simulation [1], and the Large Hadron Collider (LHC) produces approximately one petabyte of data per second [2]. Storing and processing such large amounts of data requires workarounds to be performant. In case of the LHC, a bank of processors located near the data-collection equipment makes real-time decisions about which data should be retained. The retained data is then compressed and archived. Users later decompress the data as needed. In such archival environments, users typically favor high compression ratios (and, to a lesser degree, fast decompression speeds) over fast compression speeds.

Effective data compression can reduce storage requirements, minimize data transfer times, and improve I/O speed in such environments. In many cases, it is important to preserve the data in its entirety (e.g., when computing certain derived quantities), which emphasizes the need for lossless compression. However, floating-point data is generally difficult to compress well in lossless mode. Although there have been compressors designed specifically for floating-point data, they are unable to achieve high compression ratios on the wide range of scientific data in existence because the values and patterns are highly application dependent. Thus, any one compression algorithm is unlikely to work well for all floating-point inputs.

In this paper, we study the benefit of compressing each floating-point file with a customized algorithm that is designed to achieve a high compression ratio on just this one input. For this purpose, we use LC [3], the successor of CRUSHER [4]–[7], both of which are tools that can automatically synthesize compression pipelines from a library of data transformations. We use it to generate compression algorithms that are tailored to a specific input and evaluate the benefit over using a single compressor for all inputs. Although we focus on floating-point data, our approach can be applied to any data type.

We named our approach "AdaptiveFC". It is based on a genetic algorithm (GA) to quickly search the millions of lossless compression algorithms LC can synthesize. It is orders of magnitude faster than an exhaustive search of all algorithms in the search space. We compare AdaptiveFC against 15 leading special-purpose and general-purpose GPU compressors on 77 floating-point inputs from 6 scientific datasets in the SDRBench suite [8]. We found that our approach is the top compressor overall and on 4 of the 6 datasets. The remaining two datasets both have a unique compressor that compresses its files the most, demonstrating that no single algorithm is a good choice for all datasets. This paper makes the following main contributions.

- It shows that different datasets prefer different compression algorithms and that per-file customized algorithms can deliver substantial benefits.
- It provides a detailed compression ratio comparison of 16 algorithms (including ours) on 6 floating-point datasets with 77 files from various scientific domains.
- It shows that AdaptiveFC yields the highest compression ratio on 4 of the 6 datasets, is in the top 3 compressors on the other 2 datasets, and compresses the most overall.

AdaptiveFC is freely available in open source as part of the LC Framework [3].

The remainder of the paper is organized as follows. Section 2 summarizes related work. Section 3 explains how AdaptiveFC works. Section 4 discusses the experimental evaluation methodology. Section 5 presents and analyzes the results. Section 6 concludes the paper.

II. Related Work

In prior work, we modified CRUSHER to perform *on-the-fly* synthesis of an optimized floating-point compressor for a given input [6]. This work also uses a GA, as well as several other techniques, to accelerate the synthesis. Moreover, it extracts a short segment of the input and utilizes only this segment to make the synthesis fast enough for real-time operation. In contrast, in this paper, we use the GA to perform a search with the entire input file and focus on compression ratio rather than synthesis speed.

Devarajan et al. showcase a dynamic and modular compression framework that intelligently applies compression libraries (bzip2, zlib, pithy, etc.) based on the characteristics of the input data [9]. The framework, named Ares, comprises three parts: the input analyzer, the main engine, and the output manager. The input analyzer employs static analysis and a dynamic feedback loop to describe the input data to the main engine, which then chooses the best compression library for the given input description. Finally, the output manager records how to decompress the data, checks correctness of the compression, and writes the compressed data to disk. Whereas Ares selects an algorithm from a small library of fixed compressors, our work builds customized pipelines out of a library of data transformations and can, therefore, select among millions of algorithms. Moreover, we use a GA for finding a good compression algorithm, whereas Ares employs a decision tree that requires off-line training.

The following related works target program execution traces, heterogeneous files, images, and databases instead of floating-point data. Burtscher and Sam developed TCgen, a tool that generates customized trace compressors based on a user-provided configuration of one or more predictors [10]. TCgen then translates this description into C source code that is optimized for the specified trace format and predictors. Hsu and Zwarico present an automatic synthesis technique for compressing heterogeneous files [11]. 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 added in this phase.

Fang et al. investigate how to compress database information using GPUs to overcome the transfer overhead [12]. They employ a compression planner along with a cost model of the GPU to identify an optimal combination among nine different compression schemes and use a rule-based method to automatically prune the search space. Kattan and Poli propose a system that employs genetic programming to find optimal ways to combine standard compression algorithms [13]. They group similar data chunks together and label each group with the best compression algorithm for its chunks. We also utilize a GA. However, they use fewer components than we do and, as in Kattan and Poli's as well as Devarajan et al.'s work, each component is an entire compression algorithm, whereas our components are finer grained and represent parts of a compression algorithm.

Jia et al. demonstrate a light field (LF) image compression framework that employs a generative adversarial learning network (GAN) [14]. To encode a given light field, the sub-aperture images (SAI), which make up the LF, are sparsely sampled. The unsampled SAIs are then predicted from the sampled SAIs using a GAN, providing better redundancies. The predicted SAIs are then compressed.

Mitra et al. propose a methodology for compressing fractal images using a GA [15]. Initially, fractal codes are computed for each domain block. Then, these blocks are classified into two types based on the variability of the pixels in each block. A block belongs to the smooth type if its variance is below a given threshold and is considered rough otherwise. The purpose of this classification is to obtain higher compression ratios and to reduce the encoding time. The final step uses a GA to find a good match for the rough blocks. Wu and Lin use a similar approach with three classes [16].

Several other papers have been published that employ a GA for image compression, primarily to speed up the compression. Vences and Rudomin use it to compress sequences of images [17], Wu et al. [18] improve upon Vences and Rudomin's approach, and Boucetta and Melkemi describe how to transform the RGB planes of a color image into more suitable spaces using a genetic algorithm [19].

III. Approach

Many compression algorithms can be decomposed into individual data transformations, which we call components. We implemented many such components in the LC framework, which can chain them in any combination, use the resulting pipeline to perform compression, and measure the compression ratio. This gives us the ability to search for effective pipelines for any given input. However, for pipelines with more than a few stages, searching all combinations is intractable. In the following subsection, we describe how we use a GA to perform the search for a good pipeline in a reasonable amount of time.

A. AdaptiveFC Operation

GAs are inspired by evolution and natural selection and work well for many optimization problems [20], [21]. A population of solutions to the optimization problem is evolved over generations by applying biologically-inspired operators such as crossover and mutation. For AdaptiveFC, the optimization problem is as follows: given an input file, find a pipeline in the search space that results in a high compression ratio. This means that the population consists of pipelines, where the components act as the "genes" of the individual. Typically, the first population is randomly generated and then evolved over a number of generations. AdaptiveFC employs this approach and starts with random pipelines.

a) Evaluation: Evaluation ranks each individual (i.e., each pipeline in the population) by assigning a numerical value. This value is called "fitness" and is computed by a fitness function. For AdaptiveFC, the fitness function takes a file and a pipeline as arguments and returns the compression ratio of that pipeline on that input. In other words, the fitness of an individual is its compression ratio.

b) Elitism: Elitism is the process by which the fittest members of the current population are copied into the next population, preserving good pipelines from generation to generation. This group of fittest individuals, called the elite, is calculated by cubing the fitness of each member (to increase the distance between the fitness values) and keeping all members with a cubed fitness within a factor of ϵ of the fittest individual's value, where ϵ is a parameter in the range $[0.0, 1.0]$. The best pipeline of every generation is always copied into the new population, no matter the value of ϵ, to preserve the highest compression ratio. Figure 1 provides an example of elitism.

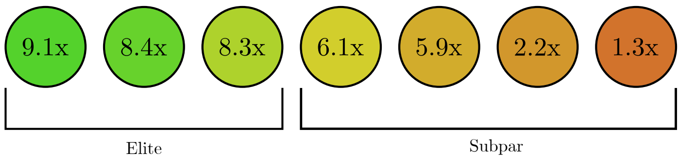

Fig. 1: Example of elitism with a cutoff value $\epsilon = 0.25$. Each circle represents a pipeline and lists the corresponding fitness (i.e., compression ratio). The elite pipelines all have a cubed compression ratio within 25 percent of the fittest pipeline.

With the new population so far consisting of the elite from the current population, the next steps, selection and crossover, generate additional pipelines until the new population matches the size of the current population. If the current and the new population are already the same size after elitism, selection and crossover are skipped.

c) Selection: Selection is the process of picking individuals in the population to act as "parents" for the crossover operation. Selecting fitter individuals is preferred for improving the solution quality. However, less fit individuals should still have a chance to be picked for genetic diversity. Since the following crossover operations require two parents, the desired selection method is run twice, with replacement. AdaptiveFC supports two selection methods: tournament and roulette wheel.

In tournament selection, t pipelines are randomly sampled from the population. Of those t pipelines, the one with the highest fitness is selected to be a parent. In roulette wheel selection, each pipeline of the population has a probability of being selected that is proportional to its fitness. In other words, the better a pipeline compresses the input file, the more likely it will be selected. Similar to elitism, we cube the fitness before selection to increase the distance between the values.

d) Crossover: The crossover operation takes two parents and shuffles their genetic information to create offspring. AdaptiveFC provides two crossover methods: single-point crossover and masked crossover. Figure 2 shows examples of both methods.

Single-point crossover works by selecting a random component as the crossover point. The components to the right of the crossover point are swapped with the other parent, resulting in two offspring. Masked crossover works by randomly generating a bit for each stage in the pipeline. Both parents keep their components in the '0' stages and swaps their components with the other parent in the '1' stages.

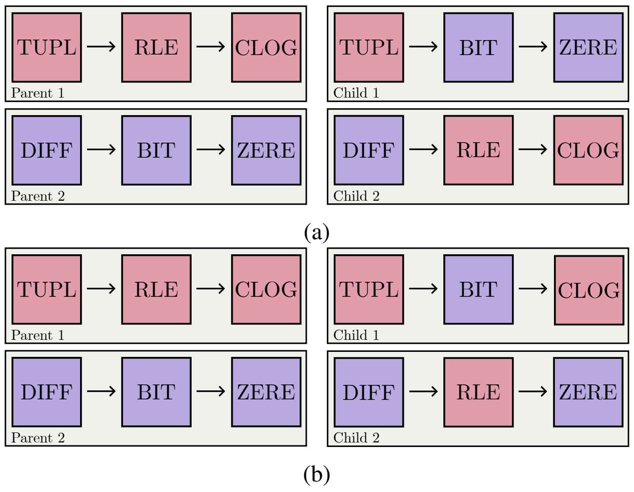

Fig. 2: Examples of crossover. The parents are on the left, and the children are on the right. (a) Example of single-point crossover. The crossover point is after the first component. (b) Example of masked crossover. The mask is '010'.

Once the offspring are created, they are included in the new population. Selection and crossover repeat until the new population has the same size as the old population.

e) Mutation: Mutation provides diversity by introducing randomness to the population. Each component of every pipeline in the new population changes into a random component with probability μ. For each pipeline in the new population, we iterate over the components and generate a random number in the interval $[0, 1)$. If this number is less than μ, we randomize the corresponding component. At the end, we add a non-mutated copy of the fittest individual from the current population to the new population. This ensures that the best compression ratio from generation to generation never decreases.

The overall process, starting with evaluation and ending with mutation, is repeated with the next generation. Once a fixed number of generations has been computed, the final generation is evaluated, and the best pipeline is outputted.

B. AdaptiveFC Parameters

AdaptiveFC allows the user to set the parameters of the GA, including the number of stages in each pipeline, the number of generations to compute, the size of the population, the elitism cutoff value, the mutation rate, the preferred selection method, the preferred crossover method, and the seed for the random-number generator used in the mutation and selection steps.

IV. Experimental Methodology

We evaluated all compressors on a system containing an AMD Ryzen Threadripper 2950X CPU with 16 cores and an NVIDIA GeForce RTX 4090 GPU with 24 GB of memory. The main memory has a capacity of 48 GB. The operating system is Fedora 37.

AdaptiveFC is implemented in Python 3 on top of LC. We ran AdaptiveFC for 140 generations with a population size of 20 and 5-stage pipelines using masked crossover and tournament selection. The mutation rate is 0.8, and the elitism cutoff is 0.1. Since AdaptiveFC's compression ratio depends on the choice of the random number seed, we ran it 9 times on each input using a fixed set of 9 distinct seeds. We report the median compression ratio from the 9 runs for each input.

We use the 6 datasets listed in Table I from SDRBench [8] as inputs. Each of the 77 files contains a binary sequence of single-precision floating-point numbers. We left out the double-precision and non-floating-point datasets from SDRBench. The CESM dataset includes both 2D and 3D inputs. We excluded the 2D data, as it is a subset of the 3D data.

TABLE I: Information about the SDRBench Inputs

Dataset	**Number of files**	**Domain**
CESM	33	Climate simulation
EXAALT	18	Molecular dynamics simulation
HACC	6	Cosmology particle simulation
NYX	6	Cosmology adaptive mesh hydrodynamics
QMC	2	Many-body ab initio Quantum Monte Carlo
SCALE	12	Climate simulation

Table II lists pertinent information about the GPU compressors we evaluated. Bitcomp includes four implementations (for byte and integer granularity and for sparse and dense inputs). Furthermore, LZ4 and Cascaded include two implementations each (for byte and integer granularity). We include all of these versions in our results, totaling 15 compressors that we compare against AdaptiveFC.

TABLE II: Information about the Evaluated Compressors

Name	**Version**	**Download Link**	**Category**
ANS	3.0.2	github.com/NVIDIA/nvcomp	General-purpose
Bitcomp	3.0.2	github.com/NVIDIA/nvcomp	Floating-point
Cascaded	3.0.2	github.com/NVIDIA/nvcomp	General-purpose
Deflate	2.3	datatracker.ietf.org/doc/html/rfc1951	General-purpose
Gdeflate	3.0.2	github.com/NVIDIA/nvcomp	General-purpose
LZ4	1.9.0	github.com/lz4	General-purpose
MPC	1.0	cs.txstate.edu/∼burtscher/research/MPC/	Floating-point
Ndzip	1.0	github.com/celerity/ndzip	Floating-point
SNAPPY	1.1.10	github.com/google/snappy	General-purpose
ZSTD	1.5.1	github.com/facebook/zstd	General-purpose

We use the compression ratio (CR), that is, the uncompressed file size divided by the compressed file size, as our main performance metric. We also use the compression and decompression throughput as performance metrics. For each dataset, we calculate the geometric-mean (gmean) CR, gmean compression throughput, and gmean decompression throughput across the files in the dataset. The timing measurements are performed by code we added before and after the compression and decompression sections of each compressor. For AdaptiveFC, we present two compression throughput measurements; one with the genetic algorithm search time included, and one without.

We also performed a parameter-space evaluation for the number of generations, the population size, and the mutation rate of the GA and analyze their impact on the compression ratio. For these experiments, the parameters that are not being tested have the same values as the previously mentioned defaults. We use a single file from the CESM dataset for these parameter-space evaluations, which we selected because it is representative of the median of all tested files in file size and in compression ratio. The range of values for the three studied hyperparameters is as follows: $[1, 200]$ for the number of generations, $[1, 50]$ for the size of the population, and $[0.00, 1.00]$ in 0.05 increments for the mutation rate. This evaluation informed us of ideal hyperparameter values to use for AdaptiveFC, discussed previously. However, note that choosing these exact values is not critical. For example, a mutation rate of 0.7 yields a similar compression ratio as a mutation rate of 0.8. Section V-D shows the results of this evaluation and provides more insight.

V. Results

A. Compression Ratio

In this subsection, we present the CR performance of the 16 GPU compressors overall and for each dataset. Figure 3 presents the gmean CRs over all datasets for all evaluated compressors. In these figures, the compressors are listed along the y-axis sorted by compression ratio, and the compression ratio achieved runs along the x-axis. Longer bars indicate higher compression.

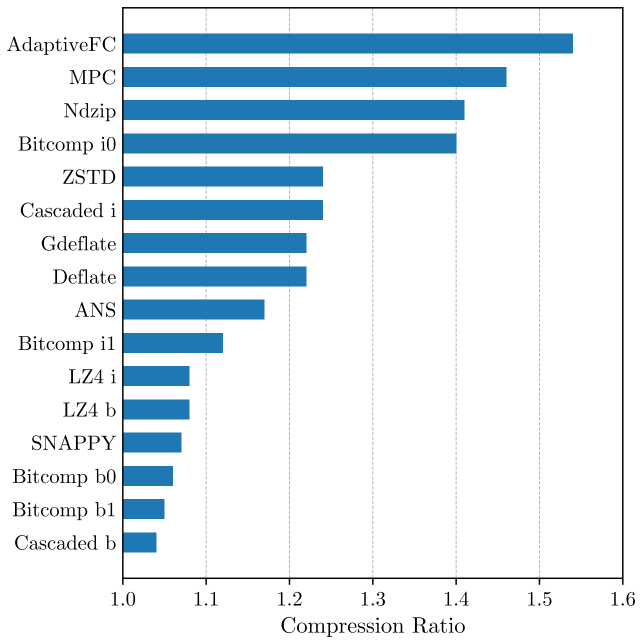

Fig. 3: Geometric-mean compression ratio for each compressor over all datasets.

AdaptiveFC performs the best, providing a gmean CR of 1.54. It is followed by MPC with a gmean CR of 1.46 and

Ndzip with a gmean CR of 1.41. The lowest CR on these datasets is 1.04.

AdaptiveFC yields the highest compression ratio, outperforming popular general-purpose compressors as well as specialized floating-point compressors. This highlights the benefit of using customized, per-input algorithms.

B. Individual Dataset Results

In this subsection, we present and discuss CR results separately for each dataset. Each figure shows the gmean CRs over all files in a dataset for the 5 best and 2 worst performing compressors, separated by a horizontal dotted line.

Note that, in these figures, the compressors with no bar do not manage to compress the files. The following paragraph lists the best-performing compression algorithms and the gmean CRs they achieve.

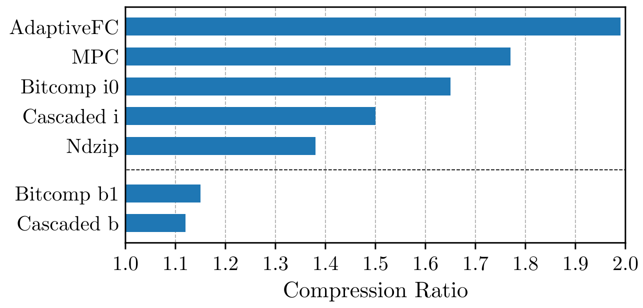

Fig. 4: Geometric-mean compression ratios on CESM dataset

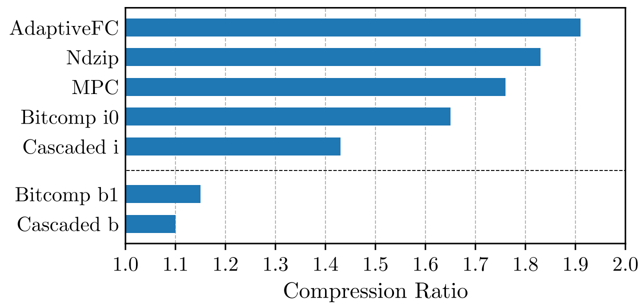

Fig. 5: Geometric-mean compression ratios on SCALE dataset

On the CESM dataset (Figure 4), AdaptiveFC yields the highest CR of 1.99. On the SCALE dataset (Figure 5), AdaptiveFC yields the highest CR of 1.91. On the HACC dataset (Figure 6), MPC performs the best with a CR of 1.72, followed by Bitcomp i0 with a CR of 1.49, and then AdaptiveFC with a CR of 1.48. On the EXAALT dataset (Figure 7), the top compressor is Ndzip with a CR of 1.42 and then AdaptiveFC with a CR of 1.38. On the NYX dataset (Figure 8), AdaptiveFC performs the best with a CR of 1.41. On the QMC dataset (Figure 9), AdaptiveFC provides the highest CR of 1.21.

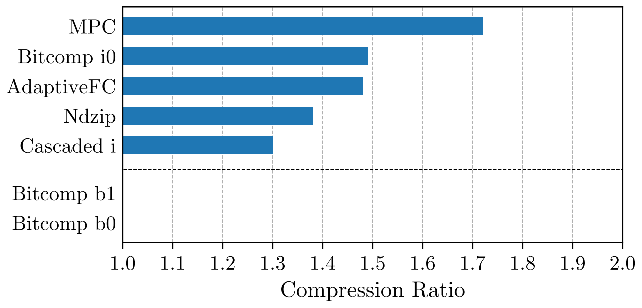

Fig. 6: Geometric-mean compression ratios on HACC dataset

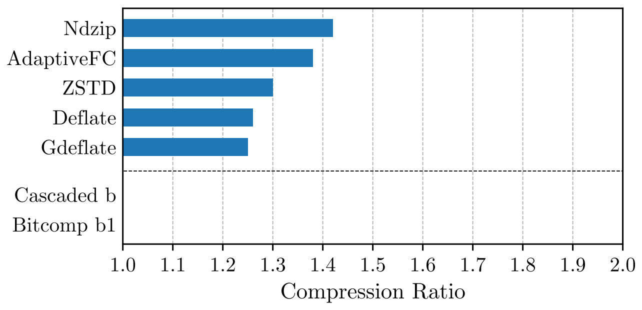

Fig. 7: Geometric-mean compression ratios on EXAALT dataset

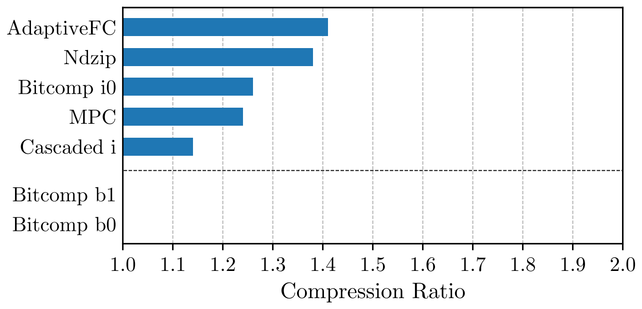

Fig. 8: Geometric-mean compression ratios on NYX dataset

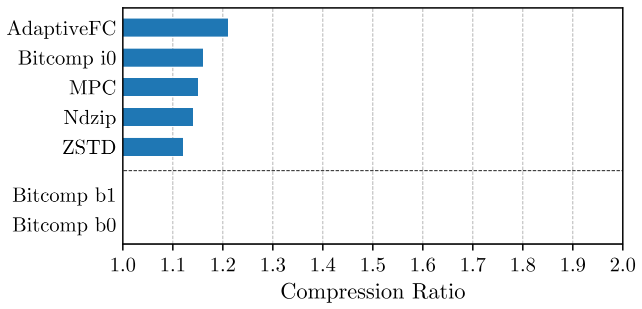

Fig. 9: Geometric-mean compression ratios on QMC dataset

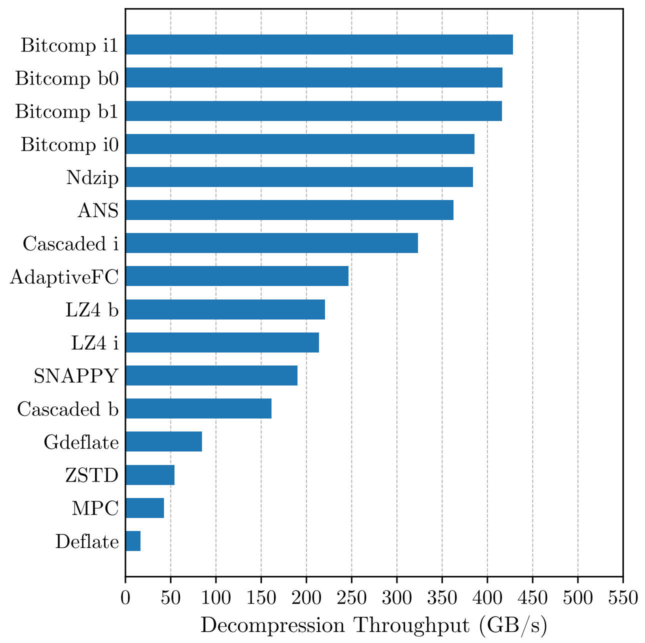

Fig. 10: Geometric-mean decompression throughput for each compressor over all datasets.

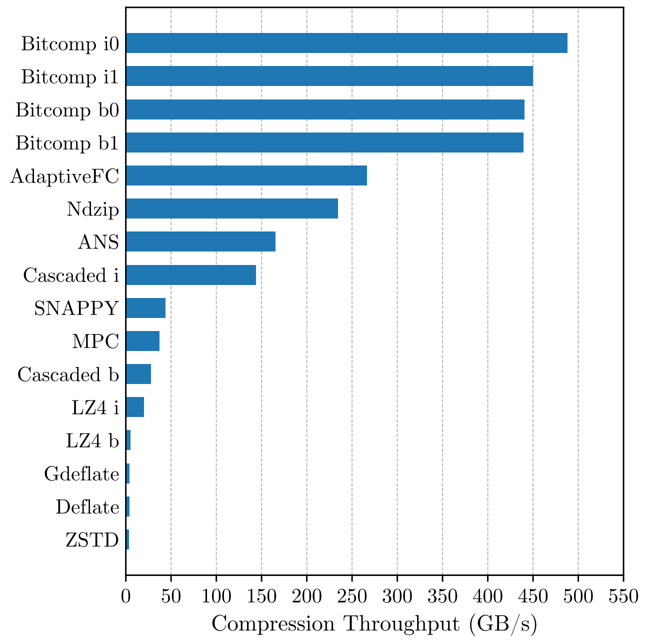

Fig. 11: Geometric-mean compression throughput for each compressor over all datasets. AdaptiveFC's search time is excluded from this graph.

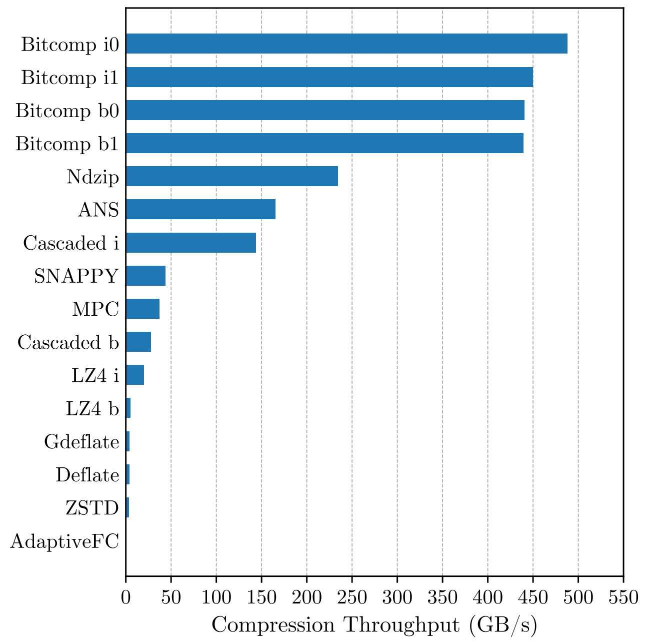

Fig. 12: Geometric-mean compression throughput for each compressor over all datasets. AdaptiveFC's search time is included in this graph.

Our approach compresses the files of 4 datasets the most, and is among the top 3 compressors for the other 2 datasets, illustrating the benefit of per-dataset customization of the compression algorithm. AdaptiveFC is the best or is among the best compressors for each dataset, which is not the case for the other compressors. There are several algorithms that perform well on some datasets and poorly on others. For example, Deflate and Gdeflate are among the top 5 compressors on the EXAALT dataset, but drop off in the other datasets. Similarly, ZSTD is among the top 5 compressors on the EXAALT and QMC datasets, but is not among the top 5 on the other datasets.

MPC outperforms AdaptiveFC on the HACC dataset. This is interesting because LC is capable of synthesizing an algorithm that is very similar to MPC. However, with the given hyperparameters and random seeds, AdaptiveFC does not find the compression pipeline that mimics MPC.

Ndzip outperforms AdaptiveFC on the EXAALT dataset. The main reason is that Ndzip employs a novel Integer Lorenzo Transform that is not available in LC. This transformation works particularly well on the EXAALT dataset.

C. Compression and Decompression Throughput

In this subsection, we study the compression and decompression throughput of the compressors over all datasets. In these figures, the compressors are again listed along the y-axis, and the throughput is listed along the x-axis. Longer bars indicate higher throughput.

Figure 10 presents the gmean *decompression* throughput for all evaluated compressors across all datasets. Bitcomp i0 performs the best, with a throughput of 428.2 GB/s. AdaptiveFC yields a throughput of 246.1 GB/s. The slowest compressor is Deflate with a throughput of 16.52 GB/s.

Figure 11 presents the gmean *compression* throughput for all evaluated compressors across all datasets. In this figure, AdaptiveFC does not include the search time in the throughput. Bitcomp i0 yields the highest throughput of 488.0 GB/s. AdaptiveFC achieves a throughput of 266.4 GB/s, and ZSTD yields the lowest compression throughput of 3.49 GB/s. Figure 12 presents the same compression throughput results but with AdaptiveFC including the search time in the measurement, yielding a throughput of 1.1 MB/s.

Note that, except for MPC, Ndzip and our AdaptiveFC algorithm, all other compressors belong to the NVcomp library. These compressors produce multiple compressed chunks that are *not concatenated* and, therefore, do not require an index to find the compressed chunks. Skipping these important steps, which are necessary in real-world applications, gives them a significant speed and a compression ratio advantage over MPC, Ndzip and AdaptiveFC.

The Bitcomp family of compressors outperforms the others in compression and decompression throughput. We do not know what algorithm Bitcomp employs as it is proprietary software. Although our approach ranks lower in throughput than Bitcomp and Ndzip, AdaptiveFC provides the highest compression ratios overall. Meanwhile, Bitcomp i0 and Ndzip fluctuate in CR performance per dataset, and rank lower overall. In some environments, compression and decompression speed is favored over compression ratio, while in others, such as in archival environments, compression ratio is typically favored over compression and decompression speed. These results highlight the tradeoff between compression ratio and compression and decompression speed.

D. Genetic Algorithm Parameter Search

This subsection presents the compression ratio performance of the genetic algorithm over a range of parameter values for the number of generations, the population size, and the mutation rate. Each figure shows the gmean CRs over the tested range of values for each hyperparameter. The parameter values run along the x-axis and the CR along the y-axis. As mentioned, these measurements are performed by running the genetic algorithm on just one file from the CESM dataset. Figures 13, 14, and 15 present the compression ratio over a range of values for the number of generations, the population size, and the mutation rate, respectively.

In Figure 13, the population of generation 0 contains all random algorithms as selection, crossover, and mutation have not yet occurred. In Figure 15, a mutation rate of 0.0 indicates no mutation takes place, leaving only selection and crossover. A mutation rate of 1.0 indicates that every component for every member of the population is randomized, essentially generating new random algorithms in each generation.

Unsurprisingly, higher numbers of generations result in higher compression ratios, eventually reaching a plateau as the CR gets closer to the optimal. At generation 0 the CR is 1.41, and the highest CR is 1.53 at generation 164. The

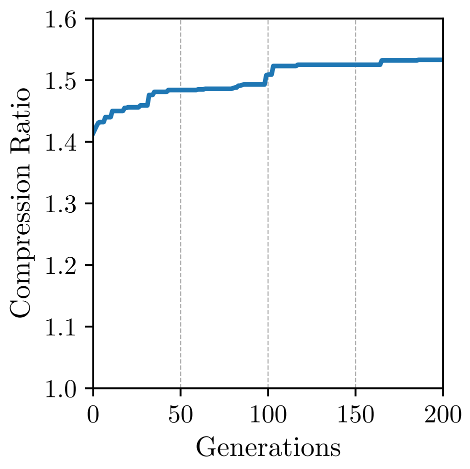

Fig. 13: Geometric-mean compression ratios over a range of values for the number of generations.

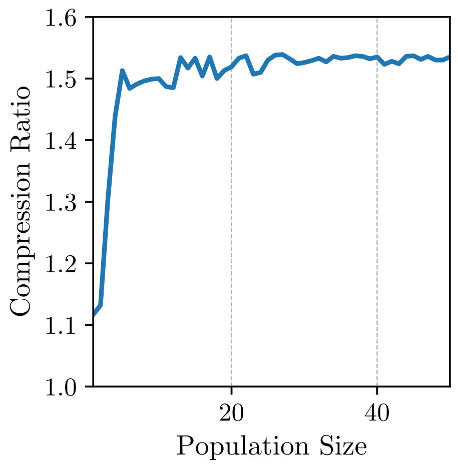

Fig. 14: Geometric-mean compression ratios over a range of values for the population size.

population size follows the same trend, where larger values result in higher CRs until plateauing. The lowest CR achieved is 1.12 with a population size of 1, and the highest CR achieved is 1.54 with a population size of 27. The mutation rate yields the highest CRs in the range 0.4–0.6, resulting in the highest CR of 1.53 with a mutation rate of 0.4, and the lowest CR of 1.48 with a mutation rate of 0.9, highlighting that too high or too low a mutation rate results in a less effective exploration of the search space. Selecting values for the hyperparameters is, in some cases, an investigation on which values give desirable results and, in other cases, a tradeoff between time and compression ratio.

VI. Conclusion

In this paper, we present AdaptiveFC, which generates a customized compression algorithm for any input. It uses a

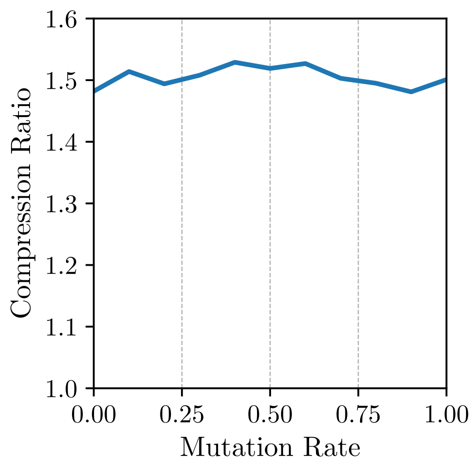

Fig. 15: Geometric-mean compression ratios over a range of values for the mutation rate.

genetic algorithm to efficiently search through a search space of millions of compression pipelines generated from a library of data transformations to quickly identify an algorithm that compresses a given input well.

Comparing AdaptiveFC to 15 other GPU-based compression algorithms on 6 scientific datasets containing 77 files, we found that AdaptiveFC provides the highest compression ratio on 4 of the 6 datasets and yields the highest geometric-mean compression ratio overall, besting popular general-purpose as well as special-purpose compressors, highlighting the benefit of per-file adaptive compression.

For future work, other optimization-problem heuristics can be explored besides genetic algorithms. Moreover, AdaptiveFC can be enhanced with the addition of new components, potentially extracted from other well-performing compressors such as Ndzip, to further improve the compression ratios.

Acknowledgements

This work has been supported in part by the Department of Energy, Office of Science under Award Number DE-SC0022223.

References

[1] Salman Habib, Vitali Morozov, Nicholas Frontiere, Hal Finkel, Adrian Pope, and Katrin Heitmann, "Hacc: Extreme scaling and performance across diverse architectures," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–10.

[2] Ian Bird, "Computing for the large hadron collider," *Annual Review of Nuclear and Particle Science*, vol. 61, pp. 99–118, 2011.

[3] Burtscher et al., "LC-framework," https://github.com/burtscher/LC-framework/, 2024, Accessed: 2024-2-8.

[4] Annie Yang, Hari Mukka, Farbod Hesaaraki, and Martin Burtscher, "Mpc: a massively parallel compression algorithm for scientific data," in *2015 IEEE International Conference on Cluster Computing*. IEEE, 2015, pp. 381–389.

[5] J. Coplin, A. Yang, A. Poppe, and M. Burtscher, "Increasing telemetry throughput using customized and adaptive data compression," in *AIAA SPACE and Astronautics Forum and Exposition*, 2016.

[6] Martin Burtscher, Hari Mukka, Annie Yang, and Farbod Hesaaraki, "Real-time synthesis of compression algorithms for scientific data," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 264–275.

[7] Steven Claggett, Sahar Azimi, and Martin Burtscher, "Spdp: An automatically synthesized lossless compression algorithm for floating-point data," in *2018 data compression conference*. IEEE, 2018, pp. 335–344.

[8] Kai Zhao, Sheng Di, Xin Lian, Sihuan Li, Dingwen Tao, Julie Bessac, Zizhong Chen, and Franck Cappello, "Sdrbench: Scientific data reduction benchmark for lossy compressors," in *2020 IEEE international conference on big data (Big Data)*. IEEE, 2020, pp. 2716–2724.

[9] Hariharan Devarajan, Anthony Kougkas, and Xian-He Sun, "An intelligent, adaptive, and flexible data compression framework," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2019, pp. 82–91.

[10] Martin Burtscher and Nana B Sam, "Automatic generation of high-performance trace compressors," in *International Symposium on Code Generation and Optimization*. IEEE, 2005, pp. 229–240.

[11] William H Hsu and Amy E Zwarico, "Automatic synthesis of compression techniques for heterogeneous files," *Software: Practice and Experience*, vol. 25, no. 10, pp. 1097–1116, 1995.

[12] Wenbin Fang, Bingsheng He, and Qiong Luo, "Database compression on graphics processors," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 670–680, 2010.

[13] Ahmed Kattan and Riccardo Poli, "Evolutionary synthesis of lossless compression algorithms with gp-zip3," in *IEEE Congress on Evolutionary Computation*. IEEE, 2010, pp. 1–8.

[14] Chuanmin Jia, Xinfeng Zhang, Shanshe Wang, Shiqi Wang, and Siwei Ma, "Light field image compression using generative adversarial network-based view synthesis," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 1, pp. 177–189, 2018.

[15] Suman K Mitra, CA Murthy, and Malay Kumar Kundu, "Technique for fractal image compression using genetic algorithm," *IEEE transactions on image processing*, vol. 7, no. 4, pp. 586–593, 1998.

[16] Ming-Sheng Wu and Yih-Lon Lin, "Genetic algorithm with a hybrid select mechanism for fractal image compression," *Digital Signal Processing*, vol. 20, no. 4, pp. 1150–1161, 2010.

[17] Lucia Vences and Isaac Rudomin, "Genetic algorithms for fractal image and image sequence compression," *Proceedings Computacion Visual*, pp. 35–44, 1997.

[18] Ming-Sheng Wu, Jyh-Horng Jeng, and Jer-Guang Hsieh, "Schema genetic algorithm for fractal image compression," *Engineering Applications of Artificial Intelligence*, vol. 20, no. 4, pp. 531–538, 2007.

[19] Aldjia Boucetta and Kamal Eddine Melkemi, "Dwt based-approach for color image compression using genetic algorithm," in *Image and Signal Processing: 5th International Conference, ICISP 2012, Agadir, Morocco, June 28-30, 2012. Proceedings 5*. Springer, 2012, pp. 476–484.

[20] Goldberg DE, "Genetic algorithms in search," *Optimization, and Machine Learning, Addison Wesley*, 1989.

[21] John H Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*, MIT press, 1992.