waveSZ: A Hardware-Algorithm Co-Design of Efficient Lossy Compression for Scientific Data

Jiannan Tian
The University of Alabama
Tuscaloosa, AL, USA
jtian10@crimson.ua.edu

Sheng Di
Argonne National Laboratory
Lemont, IL, USA
sdi1@anl.gov

Chengming Zhang
The University of Alabama
Tuscaloosa, AL, USA
czhang82@crimson.ua.edu

Xin Liang
University of California, Riverside
Riverside, CA, USA
xlian007@ucr.edu

Sian Jin
The University of Alabama
Tuscaloosa, AL, USA
sjin6@crimson.ua.edu

Dazhao Cheng
University of North Carolina at Charlotte
Charlotte, NC, USA
dcheng3@uncc.edu

Dingwen Tao*
The University of Alabama
Tuscaloosa, AL, USA
tao@cs.ua.edu

Franck Cappello
Argonne National Laboratory
Lemont, IL, USA
cappello@mcs.anl.gov

Abstract

Error-bounded lossy compression is critical to the success of extreme-scale scientific research because of ever-increasing volumes of data produced by today's high-performance computing (HPC) applications. Not only can error-controlled lossy compressors significantly reduce the I/O and storage burden but they can retain high data fidelity for post analysis. Existing state-of-the-art lossy compressors, however, generally suffer from relatively low compression and decompression throughput (up to hundreds of megabytes per second on a single CPU core), which considerably restrict the adoption of lossy compression by many HPC applications especially those with a fairly high data production rate. In this paper, we propose a highly efficient lossy compression approach based on field programmable gate arrays (FPGAs) under the state-of-the-art lossy compression model SZ. Our contributions are fourfold. (1) We adopt a wavefront memory layout to alleviate the data dependency during the prediction for higher-dimensional predictors, such as the Lorenzo predictor. (2) We propose a co-design framework named waveSZ based on the wavefront memory layout and the characteristics of SZ algorithm and carefully implement it by using high-level synthesis. (3) We propose a hardware-algorithm co-optimization method to improve the performance. (4) We evaluate our proposed waveSZ on three real-world HPC simulation datasets from the Scientific Data Reduction Benchmarks and compare it with other state-of-the-art methods on both CPUs and FPGAs. Experiments show that our waveSZ can improve SZ's compression throughput by 6.9× ~ 8.7× over the production version running on a state-of-the-art CPU and improve the compression ratio and throughput by 2.1× and 5.8× on average, respectively, compared with the state-of-the-art FPGA design.

CCS Concepts • **Computer systems organization → Reconfigurable computing**;

Keywords Lossy Compression; FPGA; Software-Hardware Co-Design; Scientific Data; Compression Ratio; Throughput.

1 Introduction

Today's large-scale high-performance computing (HPC) applications can generate extremely large volumes of scientific data. For example, the Hardware/Hybrid Accelerated Cosmology Code (HACC) [24] can simulate 1~10 trillion particles in one simulation and produce up to 220 TB of data per snapshot and a total of 22 PB of data during the simulation [55]. The large amounts of data are an imposing unprecedented burden on storage and interconnects of supercomputers [32] for both moving data to persistent storage and loading data for postanalysis and visualization. Data reduction therefore has attracted the attention of researchers for reducing the amount of data moved to and from storage systems. Decimation, which involves storing one snapshot every several time steps, undoubtedly degrades the temporal constructiveness of the simulation and also results in the

*Corresponding author: Dingwen Tao, Department of Computer Science, The University of Alabama, Tuscaloosa, AL 35487, USA.

PPoPP '20, February 22–26, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-6818-6/20/02…$15.00
https://doi.org/10.1145/3332466.3374525

loss of valuable information for postanalysis. Traditional data deduplication and lossless compression that have been widely used for reducing data redundancy suffer from their reduction capabilities. Because of nearly random ending mantissa bits of the floating-point data generated by HPC applications, deduplication typically reduces the size of scientific dataset by only 20% to 30% [38], while lossless compression usually achieves a reduction ratio of up to 2:1 [50]. These reduction ratios are far from, for instance, 10 that is needed by the Community Earth Simulation Model (CESM) [4].

Error-controlled lossy compression has been proposed recently for significantly reducing the data size while ensuring that the data distortion is still acceptable [52]. SZ [16, 52] is a state-of-the-art error-bounded lossy compression framework for scientific data (discussed in detail in Section 2.1), which often has higher compression ratio and better rate distortion than do other state-of-the-art techniques [32]. As shown in prior work [16, 52], however, SZ suffers from low compression and decompression throughput. Specifically, SZ's compression and decompression throughput on a single CPU core is about tens to hundreds of megabytes per second, which is far from enough for today's exascale applications or advanced instruments that may have extremely high data production rate. For instance, the data acquisition rate on LCLS-II can go up to 250 GB/s [7] such that the designers of the LCLS-II data system require a solution that not only can lead to relatively high compression ratios (e.g., 10:1) but also can have high compression throughput. As pointed out in [60], the key reasons for the low throughput of SZ are both its large amounts of floating-point operations and its data dependencies during the data prediction step.

Due to the lack of parallelism, it is hard to accelerate SZ by leveraging manycore techniques, such as employing GPU accelerator. GPU exhibits high throughput because of its massive SIMT (single instruction, multiple threads) parallelism, however, the tight dependency in the prediction and quantization procedure of SZ algorithm incurs expensive synchronizations across iterations in GPU execution. Moreover, during the lossless compression step of SZ algorithm (see Section 2.1 for more details), encoding and decoding each symbol according to the built Huffman tree [26] with different branches results in random memory access pattern. This causes serious divergence issue, inevitably leading to low GPU memory bandwidth utilization and performance.

Field programmable gate array (FPGA) accelerator offers many advantages, such as configurability, high energy efficiency, low latency, and external connectivity [21]. Hence, it is suitable for real-time processing such as streaming big data analytics. Adding FPGA to the existing manycore HPC system can also enable new use scenarios, such as ExaNet—an FPGA-based direct network architecture of the European exascale systems [3]. Inspired by these advantages, Xiong [60] et al. proposed an FPGA design for SZ lossy compression, named GhostSZ, combining a deprecated version of SZ (SZ-1.0 [16]) and the quantization technique as of SZ version 1.4 [52], and evaluated GhostSZ on Nek5000 benchmark datasets [1]. According to prior work [52], SZ-1.0 suffers from low prediction accuracy thus low compression ratios, especially for multidimensional datasets, because it adopts a 1D curve-fitting approach [16]. The modern versions (1.4+) of SZ [52] adopt Lorenzo predictor [28] instead of 1D curve-fitting as the predictor and use linear-scaling quantization to integerize the prediction error, which can significantly improve the compression ratio for 2D or higher dimensional datasets. Applying the Lorenzo predictor on FPGAs is challenging, however, since its inherited data dependencies during the prediction inhibit the fully pipelined design. Inspired by the existing techniques for dependency alleviation based on reshaping memory access pattern, such as loop skewing [58], PeerWave [5], and wavefront used in H.264 decoding [39], we propose to adopt the wavefront-like technique to eliminate the data dependency and efficiently pipeline SZ algorithm on FPGA, which is a starting point to achieve fine-grained parallelism in our hardware-algorithm co-design.

In this paper, we focus mainly on significantly improving the compression quality and performance (including both compression ratio and throughput) of modern SZ over the existing state-of-the-art hardware design. We propose a new fully pipelined hardware design named waveSZ based on the modern SZ model, and we optimize its performance for multidimensional datasets. waveSZ has four main steps: wavefront preprocessing, Lorenzo prediction, linear-scaling quantization, and `gzip`. Our contributions are summarized as follows.

- We adopt a wavefront memory layout to fit into SZ algorithm in order to alleviate the data dependency during the prediction for a higher-dimensional predictor, such as the Lorenzo predictor.
- We propose a co-design framework for SZ lossy compression called waveSZ, considering the wavefront memory layout and the characteristics of SZ algorithm (e.g., throughput, latency, ratio, parallelism), and carefully implement it on FPGA by using High-Level Synthesis (HLS).
- We propose a hardware-algorithm co-optimization to improve the performance, such as optimization via HLS directive and base-two algorithmic operations.
- We evaluate our proposed waveSZ on three real-world HPC simulation datasets from *Scientific Data Reduction Benchmarks* (SDRB) suite [48] and compare it with other state-of-the-art approaches on both CPUs and FPGAs. Experiments show that our waveSZ can improve the compression ratio and throughput by 2.1× and 5.8× on average, respectively, compared with the state-of-the-art FPGA implementation.

The rest of the paper is organized as follows. In Section 2, we discuss the SZ lossy compression framework in detail

and the prior FPGA-based SZ design and its limitations. In Section 3, we describe propose our novel design—waveSZ and optimization strategies. In Section 4, we present the evaluation results based on three real-world simulation datasets and compare waveSZ with other state-of-the-art approaches. In Section 5, we discuss related work. In Section 6, we present our conclusions and discuss our future work.

2 Background

In this section, we discuss the SZ error-bounded lossy compression and its prior FPGA-based design and limitations.

2.1 SZ Error-Bounded Lossy Compression

Many scientific applications require strict error-controlling scheme in lossy compression to achieve accurate postanalysis and scientific discovery, as well as high compression ratio. SZ [16, 52] is a prediction-based lossy compression framework designed for scientific data that can strictly control the global upper bound of compression error. Given a user-set error bound eb, SZ can guarantee $|d–d^\bullet| < eb$, where d and $d^\bullet$ are the original value and the decompressed value, respectively. It mainly involves four key steps: preprocessing, data prediction, linear-scaling quantization, and customized variable-length encoding followed by an optional lossless compression (such as `gzip` [14]).

1) **Preprocessing** SZ performs a preprocessing step, such as linearization in version 1.0, logarithmic transform for pointwise relative error bound in version 2.0 [31].
2) **Data Prediction** SZ predicts the value of each data point by a data-fitting predictor (such as the Lorenzo predictor [28]) based on its neighboring values. In order to guarantee that the compression error can be always controlled within the user-set error bound, the neighbor values used in the data prediction must be the decompressed values (i.e., the values used in the decompression) instead of the original data values. Such a constraint makes SZ hard to take advantage of the pipeline features in FPGA because the data cannot be predicted until its preceding points are compressed.
3) **Linear-Scaling Quantization** SZ computes the difference between the predicted value and original value for each data point and performs a linear-scaling quantization to convert the difference to an integer number based on the user-set error bound.
4) **Customized Variable-Length Encoding and Lossless Compression** SZ adopts a customized Huffman encoding algorithm and a lossless compressor (such as `Zstd` [61]) to reduce the data size significantly, because the integer codes generated by the linear-scaling quantization are likely distributed unevenly, especially when the data are mostly predicted accurately.

We list more technical details of each step for different SZ variants in Table 2. Specifically, the aforementioned four steps of SZ lossy compression can be decomposed into groups of functionalities, and each group may have optional or alternative designs for either better algorithmic features or better performance. In the data prediction step, for example, in order to estimate the data value based on its neighboring decompressed data points, multiple options exist, such as Order-{0,1,2} (i.e., neighboring, linear and quadratic) curve-fitting, Lorenzo predictor, and linear regression. The Lorenzo predictor and linear regression are designed for better prediction accuracy utilizing data correlation in high dimensional datasets; hence, they are platform agnostic. In comparison, Order-{0,1,2} curve-fitting in GhostSZ is delicately handled for FPGA pipelining (discussed in detail in Section 3.1). Prior independent work [18, 32, 36] shows that SZ leads the best compression quality among all the prediction-based compressors, therefore, we focus mainly on the SZ lossy compression model in this work.

We mainly focus on the SZ-1.4 instead of SZ-2.0 in that the 2.0 model is more effective only in the low-precision compression cases, which already lead to high compression ratios and rates in general. Recent studies [7] show that users often require a relatively high precision (or low error bound) for their scientific analysis. In this situation, lossy compressors may easily suffer from both low compression ratios and rates [32, 52], significantly degrading the overall data processing performance. Moreover, SZ-2.0 [32] has very similar (or slightly worse) compression quality/performance compared with SZ-1.4 when the users set a relatively low error bound. Accordingly, our design of the FPGA-accelerated SZ lossy compression will be based on SZ-1.4, by taking advantages of both hardware and algorithmic characteristics. Moreover, the current CPU version of SZ has no support on SIMD (single instruction, multiple data) vectorization such as SSE or AVX due to the inherited dependencies in SZ algorithm, so it has no specific improvement on the arithmetic performance. Accordingly, the CPU baseline used in our following evaluation will be based on the non-vectorized single-/multi-core implementation.

2.2 Prior FPGA-Based Design and Its Limitations

Few works have studied FPGA-accelerated lossy compression for scientific data. To the best of our knowledge, GhostSZ [60] is the first work that attempts to accelerate lossy compression based on the SZ model. However, it has several drawbacks in terms of compression ratio and throughput.

1) The prediction method used by GhostSZ is based on a deprecated design, namely, the Order-{0,1,2} 1D curve-fitting approach proposed in SZ-1.0 [16], because it suffers from low prediction accuracy on high dimensional datasets. Specifically, the Order-{0,1,2} curve-fitting approach performs three prediction methods (previous-value fitting, linear curve-fitting, and quadratic curve-fitting), which leverage the data correlation only along one dimension,

Table 1. Average compression ratio using SZ variants (visualized with scaled bars). SZ uses Lorenzo (ℓ) predictor, and GhostSZ uses modified Order-{0,1,2} predictor. Both compress at the error bound of 10^{-3}, relative to the data range.

	dimensions	GhostSZ	SZ-1.4
Climate CESM-ATM	1800×3600	7.9	31.2
Hurricane ISABEL	100×500×500	6.2	21.4
Cosmology NYX	512×512×512	6.6	33.8

leading to very limited prediction accuracy. Figure 1 shows the distributions of prediction errors (i.e., the difference between the real value and predicted value) based on the linear curve-fitting (denoted by CF-SZ-1.0) and the Lorenzo predictor (denoted by LP-SZ-1.4). We can clearly observe that the Lorenzo predictor has a higher prediction accuracy (lower prediction error) than does curve-fitting on 2D/3D datasets. More details can be found in [52].

2) The Order-{0,1,2} curve-fitting approach adopted by GhostSZ is a variant of the original version (denoted by CF-GhostSZ). CF-GhostSZ predicts the value for each data point based on the *predicted* values of previous points rather than the decompressed values. However, the predicted value without any error correction may lead to inaccurate prediction for the following data points and hence the overall low prediction accuracy, as illustrated in Figure 1.
3) The prediction method used by GhostSZ may cause a significant waste of FPGA computation resources and a workload imbalance issue. According to the Order-{0,1,2} 1D linear curve-fitting approach, GhostSZ needs to calculate three predicted values by different prediction methods and choose the bestfit value for the following quantization. As such, GhostSZ has to use three FPGA units to perform the prediction simultaneously for each data point, significantly wasting the FPGA computation resources. On the other hand, the three prediction methods have largely different computation workloads, causing a significant load imbalance issue on the corresponding FPGA units. For instance, the quadratic curve-fitting incurs twice the computation workload as that of linear curve-fitting; hence, the FPGA units assigned for the linear curve-fitting would stay idle much of time.

We evaluate GhostSZ on three representative real-world HPC simulation datasets from the Scientific Data Reduction Benchmarks (SDRB) suite [48], which has been widely used by many previous studies [7, 30–33, 53] in the scientific data reduction community. Table 1 shows that GhostSZ has a lower compression ratio than do SZ-1.4 [52] on all the three scientific datasets, under the same lossless compressor `gzip` and typical value-range-based relative error bound of 10^{-3} [15]. This is due to the much lower prediction accuracy in GhostSZ, as demonstrated in Figure 1.

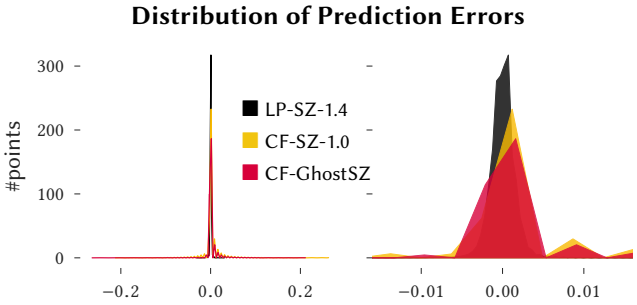


Figure 1. Distributions of prediction errors (right: zoomed in) using different predictors on a typical variable `CLDLOW` in CESM-ATM dataset.

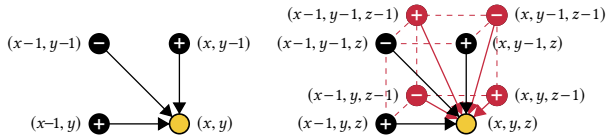


Figure 2. Single-layer 2D and 3D Lorenzo predictor. The signum $(+/-)$ of a value involved in the prediction is given by $(-1)^{L+1}$, where L is Manhattan distance from the current point. Left: four neighbor points for $\left(\ell^{\mathrm{I}}_{2\mathrm{D}}\right)$; right: eight neighbor points for $\left(\ell^{\mathrm{I}}_{3\mathrm{D}}\right)$.

3 Proposed Design of waveSZ

The key challenges that impede the SZ-1.4 from further acceleration are the tight data dependencies and stalls during the prediction: the prediction of each data point cannot be conducted until the decompression of its preceding neighbor points is finished. To alleviate this undesired stall, we adopt a wavefront memory layout [58]. We also propose several strategies based on HLS directive to optimize the performance based on the wavefront memory layout.

3.1 Wavefront Memory Layout

For demonstration, we first describe the inevitable data dependency and stall issue during the prediction step in the modern SZ model, by using the Lorenzo predictor as an example. We then discuss how the wavefront memory layout alleviates this issue.

As illustrated in Figure 2, $\left(\ell^{\mathrm{I}}_{2\mathrm{D}}\right)$ [52] can be formulated as

$$P_{(x,y)} = \mathrm{dot}\left(\begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} d_{x-1,y-1} & d_{x,y-1} \\ d_{x-1,y} & d_{x,y} \end{bmatrix}\right),$$

and $\left(\ell^{\mathrm{I}}_{3\mathrm{D}}\right)$ is formulated as

$$P_{(x,y,z)} = \mathrm{dot}\left(\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}, \begin{bmatrix} d_{x-1,y-1} & d_{x,y-1} \\ d_{x-1,y} & d_{x,y} \end{bmatrix}\right)_{(z-1)} + \mathrm{dot}\left(\begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} d_{x-1,y-1} & d_{x,y-1} \\ d_{x-1,y} & d_{x,y} \end{bmatrix}\right)_{(z)},$$

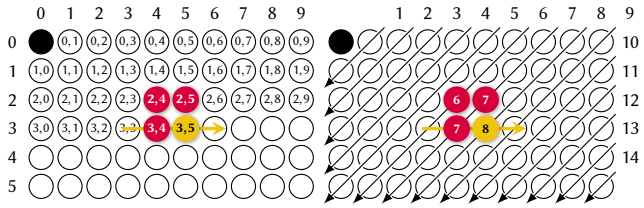
Based on these formulas, we can clearly observe that three and seven neighboring points will be involved in the computation for 2D and 3D cases, respectively, in order to predict each data point. With this feature, the prediction on one point cannot *start* until all its depending data points are fully processed through three steps: *prediction*, *quantization*, and *decompression* (denoted by PQD). In the following sections,

design goals	CPU	FPGA	base-2	base-10	base 10→2 mapping	blocking	logarithmic transform	memory layout transform	Order-{0,1,2}	Lorenzo (ℓ)	linear regression	OpenMP	explicit pipelining	line buffer	quantization	decompression writeback	prediction writeback	overbound check (s/w)	over/under-flow check (h/w)	predictor encoding	customized Huffman	gzip	Zstandard
	(P)		(P)		(P)	(P)		(P)	(P)			(S)	(S)	(S)			(S)		(S)				
	(P)				(P)	(P)	(P)			(P)	(P)				(P)	(P)		(S)					
version	**platform**		**base**		**preprocessing**					**prediction**					**lossy encoding**							**lossless**	
0.1—1.0	●			●				●		●						●						●	●	
1.4	●			●						●					●	●		●				●	●	
2.0+	●			●		●				●	●	●			●	●		●				●	★	★
GhostSZ	●	●		●				●		●			●	●	●		●		●	●			●	
waveSZ	●	●	★	★	●			●					●		●	●							●	

Table 2. SZ variants: functionality modules and design goals. ■ performance-oriented ■ data-quality-oriented
● required ★ exclusively optional (P) pan-platform (S) platform-specific

we will keep using 2D Lorenzo predictor to illustrate our proposed designs, which can be simply expanded to 3D or even higher-dimensional cases.

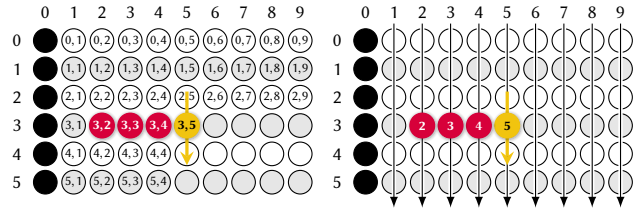
Figure 3a shows a partition of the dataset in the memory layout that is adopted by the original SZ. The highlighted four-point cell illustrates an example of an ongoing prediction: using the 2D Lorenzo predictor on $(2,2)$, $(2,3)$, $(3,2)$ to predict $(3,3)$. The prerequisite of predicting $(3,3)$ is that the decompression on the depending points $(2,2)$, $(3,2)$, $(2,3)$ is completed. Generally, for 1-layer 2D Lorenzo prediction, each data point depends on its upper and left three points in the memory layout (see the yellow cells in Figure 3a). After backtracking the data partition, we can visualize all the dependencies in Figure 3b. Specifically, given a pivot $(0,0)$ in a data partition, each data point $\boldsymbol{n}$ can calculate a Manhattan distance i between the pivot and itself. We use $(L_1(\boldsymbol{n},\boldsymbol{0}) = i)$ or simply $(L_1 = i)$ to represent this Manhattan distance. We note that the data point with $(L_1 = i)$ has dependencies on its two neighboring points with $(L_1 = i-1)$ and one upper-left neighboring point with $(L_1 = i-2)$, while it does not have any dependency with its upper-right and lower-left points with $(L_1 = i)$. In other words, *all the points with the same Manhattan distance do not have dependency with each other.*



(a) Original SZ cell indices. **(b)** Original data dependency in L_1.

Figure 3. Memory layout and data dependency in original SZ.

The current production SZ version implements the computation iterations in a simple double loop (for 2D cases) in terms of the original SZ cell indices. Thus, it must be against the dependency-free path, no matter what direction of the loop iteration is. For example, suppose the iteration moves forward from $(3,3)$ to $(3,4)$, $(3,4)$ $(L_1 = 7)$ cannot

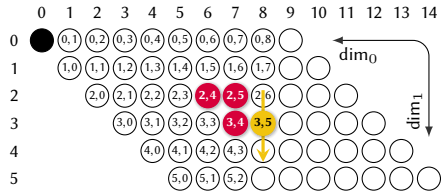


(a) GhostSZ cell indices. **(b)** GhostSZ data dependency in L_1.

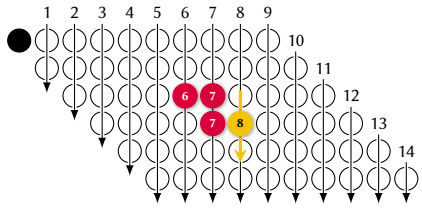
Figure 4. Memory layout and data dependency in GhostSZ.

start its prediction until $(3,3)$ $(L_1 = 6)$ finishes its decompression. Therefore, processing the data from the pivot $(0,0)$ in the original SZ faces inevitable stalls, significantly lowering its compression throughput. Although GhostSZ [60] manages to eliminate the dependencies by decorrelating the data into independent rows (as illustrated in Figure 4a), it suffers from relatively low compression ratios, especially for high-dimensional datasets. Specifically, GhostSZ ignores the correlation along the vertical direction so each row can perform the computation horizontally and independently. After the decorrelation, each row has its own starting point (or pivot), which is $(\star, 0)$. Similar to Figure 3b, we can visualize each point's Manhattan distance based on the rowwise pivot for GhostSZ. We can observe that the points within the same column have the same Manhattan distance; thus, the start of processing each point within the same column can be staged in a pipeline. GhostSZ designed this data decorrelation approach for higher compression throughput. However, the original 2D data partition is downgraded to 1D form, such that the data have to be predicted by using a 1D data predictor (such as the Order-{0,1,2} curve-fitting approach), which exploits the intrinsic correlation information only along one dimension. In summary, the design principle of GhostSZ focuses on improving the compression throughput while reducing the ratio unexpectedly (also verified in Section 2.2) compared with the modern SZ compression model.

Unlike GhostSZ, we adopt a wavefront [58] memory layout to reorganize the data access orders such that the modern SZ model can be applied to FPGAs without degradation of



(a) waveSZ cell indices.



(b) waveSZ data dependency in L_1.

Figure 5. Memory layout and data dependency of waveSZ.

compression ratios. Specifically, we note that we can preserve the data correlation only by radically changing the memory layout. As mentioned previously, data dependency is the root cause of stalls in accessing the data points that have not finished their decompression, as shown in Figure 3b. To resolve this issue, we reorganize/preprocess the memory layout into the wavefront form, as shown in Figure 5a. The wavefront memory layout can put the points with the same Manhattan distance (from the pivot $(0, 0)$) in a column, as shown in Figure 5b. Because the data points with the same Manhattan distance are not dependent on each other, the data points in the same column now are dependency free. Thus, when compression is performed vertically, there will be no data dependency and no stall. This design of dependency-free spatial mapping effectively avoids the undesired data decorrelation in GhostSZ from the perspective of algorithmic features and avoids wasting resources during the prediction with respect to the performance.

3.2 HLS-Based Co-Design

We design and optimize the SZ algorithm based on the coordination of the dependency-free memory layout (i.e., wavefront memory layout) and data access pattern (e.g., for-loop iteration in our case), of which the latter is programmed and supported by the hardware architecture. The tight correlation in the SZ prediction model and implementation can be expressed as threefold: (1) the interdimensional association of the dataset, which is a basis of the data reproducibility, (2) the algorithmic spatial and/or temporal dependency, which is discussed in Section 3.1, and (3) the hardware featured computational paradigms, of which we focus on pipelining.

As discussed in the preceding section, fully decorrelating the data would sacrifice the compression ratio. Instead, we propose to alleviate the data dependency by staging the prediction and quantization delicately, rather than adopting the simple instruction-level parallelization such as the single instruction, multiple threads model. Continuing with the 2D dataset example, the data points are iterated sequentially in the double for-loop (each fold for one dimension). To accelerate the whole computation, we adopt staging techniques such as loop unrolling and pipelining, since both of them can further exploit the parallelism of inner-loop iteration. More specifically, three key facts motivates us to use FPGA rather than CPU as our target hardware platform. (1) The degree of loop unrolling and pipelining is implicitly limited by the number of logic units (such as arithmetic logic unit) on the CPU, which prevents us from utilizing a deeper pipeline. (2) Loop unrolling and pipelining are usually controlled by the compiler rather than the program; in fact, no such primitive exists on the CPU platform that we can utilize to directly control pipeline, thus preventing us from designing a more sophisticated pipeline structure. (3) Even under the same pipeline structure, the FPGA has better performance than does the CPU, because it has more computing units than the CPU. Therefore, we use the FPGA as our target hardware platform supported by C/C++ based HLS [20] for the desired loop-unrolling factor and controllability of loop unrolling and pipelining, and hence higher throughput. C/C++ based HLS can provide good capabilities for implementing the lossy compression algorithm using the wavefront memory layout more programmable compared with VHDL and Verilog [13].

Figure 6 demonstrates our HLS-based design, which involves both a spatial memory layout and a temporal data access pattern. We implement the double loop in column-row order (inner-outer order). The compression iterator moves vertically from the top to the bottom first and then switches to the next column on the right. Since the vertical dependency has been removed, the time for the iterator moving down by one data point in the inner loop is the pipeline *Initiation Interval* (abbreviated as pII, i.e., the number of cycles between the starts of two adjacent loop iterations), which is set to 1 cycle in our case. We discuss the value of pII in the synthesis analysis (Section 3.3) . We use Λ to denote the number of points per column and Δ to denote the time (cycles) to process PQD for each point. Thus, the Λ points in a column can start every 1 cycle (due to pII = 1, according to the HLS synthesis report) sequentially, which costs Λ cycles in total. Ideally, in order to eliminate all the pipeline stalls, the time Δ for the iterator moving from the top of one column to the top of the next column should be equal to the Λ cycles, that is, $\Delta = \Lambda$. In other words, the Δ can be mapped perfectly onto the Λ points in the memory layout (temporal to spatial).

We provide detailed timing analysis for the ideal case, as follows. Note that the global index for a point is not the row-column position after the wavefront memory layout preprocessing. Given a point (r, c) at row r and column c, we can derive its starting time and ending time to process PQD, respectively. Since the inner loop iterates vertically, the starting time of processing (r, c) can be calculated as the summation of running r outer (column-wise) loops and c

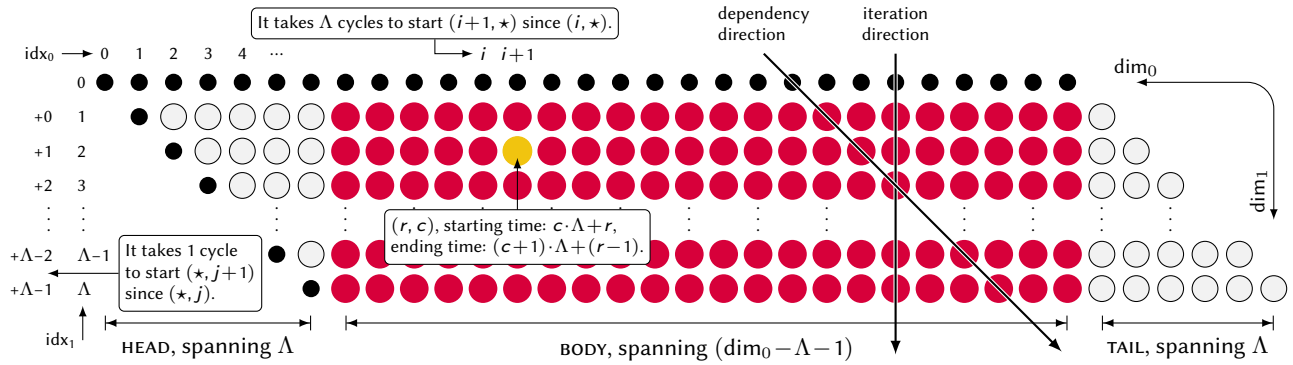


Figure 6. Memory layout with timing annotated. Λ stands for the vertical dimension of the data partition. Δ stands for the cycles of prediction, quantization, and decompression for each point.

inner (row-wise) loops. It takes one cycle to the next inner-loop iteration and Λ cycles to proceed to the next outer-loop iteration. Thus, the starting time of (r, c) against $(1, c)$ (the first point in the same column) is increased by $(r-1)$ cycles, and the starting time of $(1, c)$ against $(1, 1)$ (the first point in global) is increased by $(c \times \Lambda)$ cycles. Overall, the global starting time of (r, c) is $(c \times \Lambda + r)$. Also, we can derive the ending time of (r, c), which is Λ cycles after its starting time, in other words, $\big((c+1) \times \Lambda + r - 1\big)$. Note that the starting time of $(r, c+1)$ is one cycle after the ending time of (r, c).

From the perspective of programming, we further divide outer-loop columns into three groups: HEAD, BODY, and TAIL, as shown in Figure 6. Each columnwise inner loop in the BODY will perform the same floating-point operations on the same number of points; thus the loop in the BODY is called a "perfect" loop. However, the loops in HEAD and TAIL are "imperfect" because their execution times vary across different columns. We find that a stall does not occur in the "perfect" loop, which completely follows the aforementioned temporal-spatial mapping. In comparison, stalls still exist in the "imperfect" loop, but the number of points in HEAD and TAIL is much smaller than the number of points in the no-stall BODY. This partition can help the program eliminate the if-branch for calculating the ending index, because the number of points can be computed directly based on the column index for HEAD and TAIL.

In addition, in the original SZ, the boundary points in the first column and first row are marked as unpredictable data (blackened points), which are handled by a truncation-based binary analysis [16]. In comparison, our waveSZ directly passes the unpredictable data to the lossless compressor (i.e., `gzip`) instead of truncation for higher throughput. Since the linear-scaling quantization method using 16-bit quantization bins can cover most of the prediction errors (e.g., > 99%), few points would be affected by this change.

3.3 Hardware-Algorithm Co-Optimization

We propose a *hardware-algorithm co-optimization* strategy to optimize the performance not only on the hardware level but also on the algorithmic level. Figure 7 presents the overall system architecture based on our proposed waveSZ. Specifically, on the host CPU it conducts the preprocessing for the wavefront memory layout; and on the device FPGA it involves the hardware designs of fully pipelined Lorenzo prediction, linear-scaling quantization, and decompression with the Huffman coding. Note that the preprocessing step to form the wavefront memory layout is basically memory copy, which can be pipelined and overlapped with other steps on FPGA. The compressed data will be stored in another computing or storage device. Similar to prior work [60], we demonstrate the use case by transferring the compressed data from the FPGA back to the host CPU. In the real HPC scenarios, the use case can be accelerating the data transfer to the parallel file system through the I/O node.

To optimize the overall performance, we develop an HLS directive-based optimization method (hardware level) and reduce the time cost of running PQD by performing exponent-only operation (base-2 operation), detailed in the following text.

Optimization via HLS Directive The HLS code snippet for three groups of iterations is shown in Listing 1. Specifically, we mainly focus on optimizing the most computation-intensive loops. In our implementation, we mix-use HLS directives and HLS-supported C-syntax pragmas. For more direct demonstration, we show only the pragmas in the listed code. First, the three double-loops are labeled as `Head-`, `Body-`, and `Tail-`. The outer and inner loops for each group are labeled as `-H` (stands for "horizontal" iteration direction) and `-V` (stands for "vertical" iteration direction), respectively. This notation is used to mark the six loops for HLS-specific optimization. The pragma residing in the inner loop (e.g., (line 10) in Listing 1) is used for unrolling the inner-loop iterations. Ideally, the inner loop can be unrolled by a factor of the pipeline depth Λ for the perfect BODY loop. However, since the inner-loop termination conditions vary across HEAD and TAIL loops, these two groups of loops cannot be unrolled by the factor of Λ, and their unrolling factor will be determined by the synthesis tool automatically. Note that for pipelining (line 11), we set the pII to 1 cycle; however, if the pipelining cannot be performed with (pII = 1) during the synthesis, the

synthesis tool will relax the restriction of (pII = 1) to the smallest value to reduce the latency. In addition to the loop unrolling and explicit pipelining, we hardcoded the pipeline depth via template metaprogramming (line 2 in Listing 1). The hardware-resource utilization will be discussed in the evaluation section.

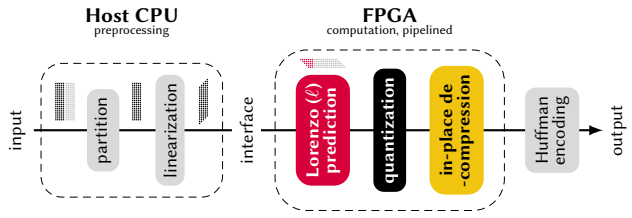


Figure 7. Overall system architecture based on waveSZ.

Listing 1. HLS code snippet for the {HEAD, BODY, TAIL}-loop iterations

```
1  template <typename T, typename Q, int PIPELINE_DEPTH>
2  void wave(int d0, int d1, /*dimensions*/
3            T*  data, Q* quant_code) {
4    /* the first row is the dependency. */
5    assert(PIPELINE_DEPTH == d0-1);
6    size_t h, v, NW, N, W, /*global idx*/_gi, _idx = 1;
7    HeadH: for (h=1; h<d0; _idx+=h+1, h++)
8      HeadV: for (v=0; v<h; v++) {
9  #pragma HLS unroll
10 #pragma HLS PIPELINE II = 1
11       NW = _idx+v-h; N   = _idx+v;
12       W  = _idx+v+1; _gi = _idx+(v+1)+(h+1);
13       T pred = Lorenzo_2D1L<T>(data, NW, N, W);
14       quantize<T, Q>(pred, data, _gi, quant_code); }
15   BodyH: for (h=d0; h<d1-1; _idx+=d0, h++)
16     BodyV: for (v=0; v<d0-1; v++) {
17 #pragma HLS unroll
18 #pragma HLS PIPELINE II = 1
19       NW = _idx+v-d0; N   = _idx+v;
20       W  = _idx+v+1;  _gi = _idx+(v+1)+d0;
21       T pred = Lorenzo_2D1L<T>(data, NW, N, W);
22       quantize<T, Q>(pred, data, _gi, quant_code); }
23   TailH: for (h=d1-1; h<d1-d0; _idx+=(d1+d0)-h, h++)
24     TailV: for (v=h-d1+1; v<d0-1; v++) {
25 #pragma HLS unroll
26 #pragma HLS PIPELINE II = 1
27       NW = _idx+v-d0; N   = _idx+v;
28       W  = _idx+v+1;  _gi = _idx+(v+1)+d0;
29       T pred = Lorenzo_2D1L<T>(data, NW, N, W);
30       quantize<T, Q>(pred, data, _gi, quant_code); } }
```

Base-2 Operation In compliance with the widely used IEEE-754 (2008 rev.) standard [11], a floating-point number, with regard to precision, can be represented as `float{16, 32, 64, 128, 256}`, [1] among which the *single-* and *double-precision* are hardware supported in the general-purpose systems. For example, 32-bit `float` is represented in bits as $(-1)^{\text{signum}} \times 2^{(\text{exponent}-127)} \times (1 \bullet b_{22} \cdots b_{00})$.

The original SZ adopts an arbitrary user-set error bound, which is likely not power-of-two, leading to disordered mantissa bits (mixed with 0 and 1) in its binary representation. For example, Table 3 shows typical decimal bases, which are

[1] Half-, single-, double-, quadruple-, octuple-precision, respectively.

Algorithm 1 Computation of *prediction*, *quantization*, and *decompression*.

input: current data point: $d_{x,y}$; dependencies: $d_{x-1,x-1}$, $d_{x-1,y}$, $d_{x,y-1}$; prediction: *pred*; max. quantizable #: *capacity*; precision p; radius r
intermediate: decompressed value: d_{re}; quantization code: *code*

```
pred  ← ℓ(d_{x-1,y-1}, d_{x-1,y}, d_{x,y-1})      ▷ 1-layer 2D Lorenzo (ℓ^I_2D)
diff  ← d_{x,y} − pred
d_ori ← d_{x,y}                                    ▷ register the original value
code° ← |diff/p| + 1
if code° < capacity then                           ▷ check if quantizable
    code° ← (diff > 0) ? code° : −code°
                ▷ change the signum (+/−) of code° according to diff
    code• ← CAST_TO_INTEGER(code°/2) + r
    d_re  ← pred + 2 × (code• − r) × p             ▷ SZ, waveSZ
    write back d_re  to where d_(0,0) was
    write back pred to where d_(0,0) was           ▷ GhostSZ
    return code• if |d_re − d_ori| ≤ p             ▷ overbound check
end if
write back d_ori to where d_{x,y} was
return 0                                           ▷ 0 for non-quantizable
```

represented with 0-1 mixed (non-zero) mantissa bits. Using such arbitrary numbers as divisors, the division operations used for integerization in the quantization step have to do full-featured arithmetic calculations, which is relatively expensive because it is usually handled by dedicated hardware such as DSP on FPGA and FPU on CPU. By tightening the user-set error bound to its nearest smaller power-of-two (e.g., 10^{-3} to $2^{-10} = 1/1024$), we can instead perform exponent-only calculations, i.e., addition and subtraction operations in exponent, which can further improve the overall throughput.

Table 3. Binary representation of decimal basis correspondence to 23 mantissa bits.

decimal bases	binary representation
0.1, or 1e−1	$(1.1001100110011\cdots)_2 \times 2^{-4}$
0.01, or 1e−2	$(1.0100011110101\cdots)_2 \times 2^{-7}$
0.001, or 1e−3	$(1.0000011000100\cdots)_2 \times 2^{-10}$
0.0001, or 1e−4	$(1.1010001101101\cdots)_2 \times 2^{-14}$
0.00001, or 1e−5	$(1.0100111110001\cdots)_2 \times 2^{-17}$
0.000001, or 1e−6	$(1.0000110001101\cdots)_2 \times 2^{-20}$
0.0000001, or 1e−7	$(1.1010110101111\cdots)_2 \times 2^{-24}$

4 Experimental Evaluation

In this section, we present our experimental setup (including the testbed, tested datasets, baselines, and performance metrics) and our evaluation results.

4.1 Experimental Setup

Experimental Platform We use two systems as our testbed. The first system is a CPU node from the PantaRhei cluster [41] from The University of Alabama, equipped with Intel Xeon Gold 6148 processors running at 2.4 GHz. The second system is a Xilinx Zynq-7000 SoC ZC706 FPGA platform. waveSZ is implemented with Vivado High-Level Synthesis [56] from the Vivado Design suite [17] (v.2019.1). We use Xilinx Floating-Point Operator IPs for floating-point arithmetic. The IP configuration is set for the highest frequency when it is possible. The default frequency is 156.25 MHz. We

adapt the pipeline configuration (such as pipeline depth) to the dimension of each dataset.

Test Datasets We conduct our evaluation and comparison based on three typical 2D and 3D real-world HPC simulation datasets from SDRB suite [48]: 2D CESM-ATM climate simulation [12], 3D ISABEL hurricane simulation [27], and 3D NYX cosmology simulation [40]. They have been widely used in much prior work [7, 31–33, 53] and are good representatives of production-level simulation datasets. Each dataset involves multiple snapshots and a diversity of fields. Table 4 presents all 105 fields across these datasets. The data sizes per snapshot are 2.0 GB, 1.9 GB, and 3.0 GB for the above three datasets, respectively.

Table 4. Real-world datasets used in evaluation.

	# fields	type	dimensions	example fields
CESM-ATM	79	`float32`	1800×3600	`CLDHGH`, `CLDLOW`
Hurricane	20	`float32`	100×500×500	`CLOUDf48`, `Uf48`
NYX	6	`float32`	512×512×512	`baryon_density`

Comparison Baselines We compare our waveSZ with two baselines: SZ-1.4 (the software version that we use in waveSZ), and GhostSZ [60]. For SZ-1.4, we adopt the default setting: 16 bits for linear-scaling quantization (i.e., 65,536 quantization bins), `best_compression` mode for SZ, and `best_speed` mode for `gzip`, which can lead to a good tradeoff between compression ratio and performance. We implement GhostSZ strictly based on [60]. We adopt the FPGA hardware design of `gzip` [59] developed by Xilinx for both GhostSZ and waveSZ. We use 16 bits to represent one quantization bin in waveSZ, so its number of quantization bins is up to 65,536. Although GhostSZ also uses 16 bits to represent one quantization bin, it can only generate at most 16,384 bins because it needs to use 2 bits to encode the bestfit curve-fitting approach in Order-{0,1,2}. This will increase the number of unpredictable data points, degrading the compression ratios in turn. For all the experiments, we use the typical value-range-based relative error bound of 10^{-3} [15]. Note that since waveSZ uses base-2 operations, we calculate the power-of-2 error bounds that are closest to the value-range-based relative error bound of 10^{-3}. We use GCC-7.3.0 to compile the software implementations.

Performance Metrics We evaluate each SZ variant using three metrics: *throughput*, *compression ratio*, and *peak signal-to-noise ratio* (PSNR). Throughput is defined as the number of data points processed (such as compression) per second. Similar to prior work [60], the latency of the entire compression process (excluding the file loading and dumping time) is measured as the period from the moment that FPGA/CPU receives the data through the moment that the whole compression is finished with output bytes. Compression ratio is defined as the ratio of original data size to the compressed data size. PSNR is a commonly used indicator to assess the distortion of data during lossy compression, which is defined as $\text{PSNR} = 20 \cdot \log_{10}\left[(d_{\max} - d_{\min})/\text{RMSE}\right]$, where N is the number of data points and $d_{\max}$ and $d_{\min}$ are the maximal and minimal values, respectively. RMSE represents *root mean squared error*, which is obtained by $\text{sqrt}\left[\frac{1}{N}\sum_{i=1}^{N}\left(d_i - d_i^{\bullet}\right)^2\right]$, where d_i and $d_i^{\bullet}$ refer to the original and decompressed data values, respectively. The larger the PSNR, the lower the RMSE, meaning lower distortion of reconstructed data and hence more accurate postanalysis.

4.2 Evaluation Results and Analysis

Table 5. Comparison of compression throughput (MB/s).

	waveSZ	GhostSZ	SZ-1.4
CESM-ATM	995	185	114
Hurricane	838	144	122
NYX	986	156	125

We first evaluate the throughput of waveSZ on FPGA with single lane and compare with the throughput of original SZ on CPU with single CPU core. We present the compression throughput results in Table 5, showing that waveSZ on FPGA has 6.9× to 8.7× higher compression throughput than does SZ-1.4 on the CPU, respectively. This is because of the low-data-dependency memory layout and highly pipelined design in waveSZ. Compared with GhostSZ, waveSZ can achieve 5.8× improvement on average due to its lower number of computing units used for prediction per point. We note that since waveSZ requires generating the decompressed value of each point, it needs a relatively deeper pipeline depth (i.e., Λ). As discussed in Section 3.2, the pipeline depth can be mapped onto a partition of the data in the "body"-loop iterations. Therefore, a deeper pipeline would affect only the latency of the boundary data points while maintaining the overall high throughput. Note that we only focus on the compression throughout in the evaluation because the compressed data would be transferred and stored to the parallel file systems after waveSZ compression, and users mainly use the SZ on CPU to decompress the data for postanalysis and visualization instead of FPGA.

Table 6 shows the FPGA resource consumption. Note that the table shows the utilization of three parallel PQD procedures in waveSZ, in order to match GhostSZ's Order-{0,1,2} procedure (with three predictors), taking into account the adequate hardware resource. We note that waveSZ consumes much smaller amount of resources than GhostSZ does on a single pipeline. We expect the scalability would be limited by `gzip`'s BRAM consumption (e.g., 303 as shown in the reference website [59]) and the PCIe bandwidth (e.g., 4× PCIe 2.1 operating at 5 Gb per lane in ZC706).

We then evaluate the throughput of waveSZ on FPGA with multiple lanes and compare with the throughput of original SZ with multiple CPU cores using its OpenMP implementation. In Figure 8, we present the parallel compression throughputs of SZ-1.4, GhostSZ), and waveSZ. Note that

since the 32-core throughput of SZ-1.4 is almost identical to its 40-core throughput, we only show the results up to 32 CPU cores. It is also worth noting that the OpenMP implementation in the modern versions (1.4+) of SZ supports only 3D datasets, thus, we only evaluate the 3D Hurricane and NYX datasets. We observe that the throughput of SZ-1.4 using OpenMP increases sublinearly with the number of CPU cores because of the context switch, and its parallel efficiency drops to as low as 59% at 32 cores. Figure 8 also illustrates that the throughputs of GhostSZ and waveSZ on FPGA can scale up linearly, and their parallelism/throughput would be limited by the hardware resource, such as BRAM capacity, number of PCIe lanes, and overall PCIe bandwidth.

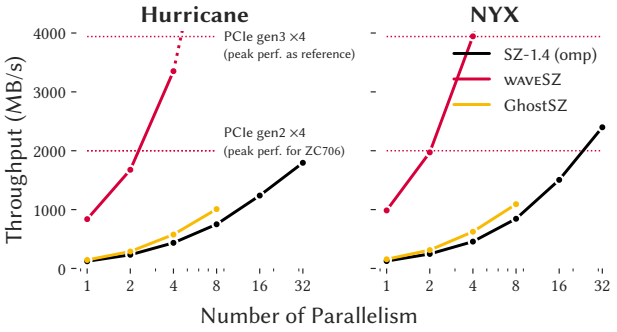


Figure 8. Parallel compression throughput (MB/s).

Table 7 compares the compression ratios among different compressors. The average compression ratio of waveSZ is 1.6×, 2.1×, and 2.7× higher than that of GhostSZ on the CESM-ATM, Hurricane, and NYX datasets, respectively. The main reason is that the Lorenzo predictor used by waveSZ has a higher prediction accuracy than does the variant of Order-{0,1,2} curve-fitting approach used by GhostSZ, as discussed in Section 2.2. We also observe that SZ-1.4 leads to higher compression ratios than do the FPGA-based versions (either waveSZ or GhostSZ because of the customized Huffman coding (i.e., 16-bit or 4 bytes per symbol) proposed by the original SZ model. Since FPGA versions for the customized Huffman coding algorithm do not yet exist, waveSZ and GhostSZ adopt the FPGA-based `gzip` developed by Xilinx [59] for simplicity, which may significantly limit the compression ratio. For demonstration purposes, we further apply the customized Huffman coding (denoted by H*) before the `gzip` (denoted by G*) in waveSZ. Table 7 shows that after applying the customized Huffman coding, waveSZ can get the similar compression ratios as SZ-1.4 (i.e., the software version of waveSZ). However, since designing a high-efficient customized Huffman coding on FPGAs is not the focus of this paper, we leave it for the future work.

We assess the compression errors for all four compressors. We note that they all satisfy the requirement of relative error bound 10^{-3}. As shown in Table 8, waveSZ has similar PSNRs on the tested datasets compared with SZ-1.4.

Table 6. Resource utilization from synthesis

	total	waveSZ	(%)	GhostSZ	(%)
BRAM_18K	1,090	9	0.84	20	1.83
DSP48E	900	0	0.00	51	5.67
FF	437,200	4,473	1.02	12,615	2.89
LUT	218,600	8,208	3.75	19,718	9.02

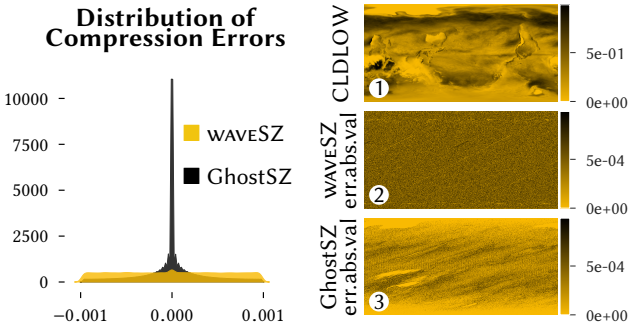


Figure 9. Compression errors analysis for waveSZ and GhostSZ on a typical variable `CLDLOW` in CESM-ATM. Left: error distribution. Right: (1) original data, (2) absolute value of waveSZ compression error, and (3) absolute value of GhostSZ compression error.

GhostSZ has a slightly higher PSNR than do the other three approaches, which can be explained by analyzing the distribution of compression errors. For instance, Figure 9 shows that the compression error of GhostSZ is distributed a little bit more concentratedly than that of waveSZ. The reason is that there are some regions with very similar values in the dataset (see the top and bottom area in Figure 9(1), such that the bestfit method in Order-{0,1,2} would always be the previous-value fitting in these areas. It may have a relatively high chance of predicting the data value accurately in such regions, which can be verified in Figure 9(3). By comparison, the Lorenzo predictor adopted by waveSZ has a better prediction accuracy overall (as shown in Figure 1) but relatively lower chance of getting a high prediction accuracy in the similar-value areas (as verified Figure 9(2), leading to an even distribution of compression errors.

Table 7. Comparison of compression ratio (visualized with scaled bar) with 10^{-3} relative error bound. H*: customized Huffman encoding (if in use, followed by `gzip`); G*: using `gzip`. Note that border points are counted as unpredictable data in waveSZ.

		CESM-ATM	Hurricane	NYX
GhostSZ		7.9	6.2	6.6
waveSZ	G*	12.3	13.2	18.3
	H*G*	29.4	20.3	34.8
SZ-1.4		31.2	21.4	33.8

Table 8. Comparison of PSNR (dB) with 10^{-3} relative error bound.

	GhostSZ	waveSZ	SZ-1.4
CESM-ATM	73.9	65.1	64.9
Hurricane	70.6	66.0	65.0
NYX	74.5	66.5	65.2

5 Related Work

5.1 Scientific Data Compression

Scientific data compression techniques have been studied for decades. Generally, data compressors can be split into two categories: lossless and lossy. Lossless compressors designed for floating-point data such as FPC [6] and SPDP [9] can guarantee the decompressed data completely lossless, but they cannot significantly reduce the simulation data size because of random ending mantissa bits of the floating-point values [23]. In absolute terms, the compression ratios of lossless compressors are around 2:1 [50] on scientific datasets, far lower than the user-desired level of around 10:1 [7].

Error-bounded lossy compressors have been developed as an important option to significantly reduce the scientific simulation data size while guaranteeing the decompressed data fully respects user-specified error bounds. Existing error-bounded lossy compressors are designed by either a prediction-based model [8, 16, 22, 29, 31, 35, 52] or a transform-based model [10, 34, 47, 54, 57]. Recent research studies illustrated that SZ (prediction-based model) and ZFP (transform-based model) are two leading lossy compressors for HPC scientific data. Lu et al. [36], for example, evaluated multiple compressors using nine HPC simulation data sets and concluded that SZ and ZFP are the two best compressors in their class. Furthermore, Tao et al. [53] proposed an online selection method that can automatically select the bestfit lossy compressor between SZ and ZFP, based on their observation that neither SZ nor ZFP can always lead to the best compression quality over the other across multiple fields in different data sets.

5.2 FPGA-Accelerated Data Compression

FPGA-based lossless compression has been studied for decades, especially because of increasing demand on compression throughput. Many lossless compression algorithms have been optimized and implemented on FPGAs. For example, Huang et al. [25] implemented the Lempel-Ziv algorithm on Xilinx FPGA chips. Rigler et al. [44] proposed a hardware implementation of the DEFLATE compression algorithm (including LZ77 algorithm and Huffman encoding) on Altera FPGAs. Their work, however, still suffers from relatively low compression throughput because the data hazards in compression algorithms such as LZ77 limit the creation of deep pipelines without sacrificing compression quality. To solve this issue, vendors developed several scalable high-throughput pipeline implementations [2, 19, 37] thanks to improvements in the gate number and I/O bandwidth. IBM [37] proposed an implementation of the DEFLATE algorithm for FPGAs with 4 GB/s throughput at 16 bytes/cycle. Altera [2] implemented LZ77 for FPGAs by using their OpenCL compiler and achieved the same throughput as IBM did. Microsoft [19] proposed a fully pipelined FPGA accelerator for the DEFLATE algorithm that can achieve a throughput of 5.6 GB/s. They also explored trade-offs between compression quality and FPGA throughput. Qiao et al. [42] further improved the throughput and overall system efficiency of the FPGA DEFLATE accelerator by exploiting a multiway parallel design. Besides DEFLATE, some other lossless compression algorithms such as Burrows-Wheeler Transform lossless compression [43] have been studied. Unlike these industry solutions only focusing on lossless compression, our lossy compression can significantly reduce the storage overhead due to much higher compression ratios.

However, few studies in the scientific community have investigated FPGA-accelerated lossy compression for scientific data. Existing work [45, 46, 49, 51] has focused mostly on lossy compression for image and video data rather than scientific data. Recently, GhostSZ [60] was proposed to accelerate a deprecated version of SZ lossy compression [16] (i.e., SZ-1.0) at line rate. Their experimental results, based mainly on synthetic HPC application data, showed 8.2× speedup by using on FPGAs compared with on state-of-the-art CPUs, illustrating the potential success of SZ lossy compression framework on FPGAs.

6 Conclusion and Future Work

In this work, we propose a highly efficient FPGA-based lossy compression approach called waveSZ, which can improve the compression throughput significantly for SZ. Our solution adopts the wavefront memory layout to alleviate the data dependency during the prediction for multidimensional predictors and datasets. We also propose HLS directive-based optimizations and use base-2 algorithmic operations to eliminate the expensive overbound check for each point. Experiments on three real-world HPC simulation datasets show that our proposed waveSZ can improve the compression throughput by 6.9×∼8.7× on average over the CPU version. Compared with the existing FPGA-based SZ, waveSZ can improve the compression ratio by 2.1× and the throughput by 5.8× averagely on the tested datasets and FPGA. We plan to implement the FPGA version for the customized Huffman encoding, which can further improve compression ratios especially for high-dimensional datasets.

Acknowledgment

We thank Xilinx for generous donations of software and hardware. This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations—the Office of Science and the National Nuclear Security Administration, responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, to support the nation's exascale computing imperative. The material was supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357, and supported by the National Oceanic and Atmospheric Administration under Grant No. NA18NWS46200438.

References

[1] NEK5000: a fast and scalable high-order solver for computational fluid dynamics. 2019. GZIP file format specification version 4.3. https://nek5000.mcs.anl.gov/. Online.

[2] Mohamed S Abdelfattah, Andrei Hagiescu, and Deshanand Singh. 2014. Gzip on a chip: High performance lossless data compression on FPGAs using OpenCl. In *Proceedings of the International Workshop on OpenCL 2013 & 2014*. ACM, ACM, Bristol, UK, 4.

[3] Roberto Ammendola, Andrea Biagioni, Fabrizio Capuani, Paolo Cretaro, Giulia De Bonis, Francesca Lo Cicero, Alessandro Lonardo, Michele Martinelli, Pier Stanislao Paolucci, Elena Pastorelli, Luca Pontisso, Francesco Simula, and Piero Vicini. 2018. Large Scale Low Power Computing System - Status of Network Design in ExaNeSt and EuroExa Projects. arXiv:cs.DC/1804.03893

[4] Allison H Baker, Haiying Xu, John M Dennis, Michael N Levy, Doug Nychka, Sheri A Mickelson, Jim Edwards, Mariana Vertenstein, and Al Wegener. 2014. A methodology for evaluating the impact of data compression on climate simulation data. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, ACM, Vancouver, BC, Canada, 203–214.

[5] Mehmet E Belviranli, Peng Deng, Laxmi N Bhuyan, Rajiv Gupta, and Qi Zhu. 2015. Peerwave: Exploiting wavefront parallelism on gpus with peer-sm synchronization. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, Austin, TX, USA, 25–35.

[6] Martin Burtscher and Paruj Ratanaworabhan. 2008. FPC: A high-speed compressor for double-precision floating-point data. *IEEE Trans. Comput.* 58, 1 (2008), 18–31.

[7] Franck Cappello, Sheng Di, Sihuan Li, Xin Liang, Ali Murat Gok, Dingwen Tao, Chun Hong Yoon, Xin-Chuan Wu, Yuri Alexeev, and Frederic T Chong. 2019. Use cases of lossy compression for floating-point data in scientific data sets. *The International Journal of High Performance Computing Applications* 33, 6 (2019), 1201–1220.

[8] Zhengzhang Chen, Seung Woo Son, William Hendrix, Ankit Agrawal, Wei-keng Liao, and Alok Choudhary. 2014. NUMARCK: Machine learning algorithm for resiliency and checkpointing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, New Orleans, LA, USA, 733–744.

[9] Steven Claggett, Sahar Azimi, and Martin Burtscher. 2018. SPDP: An automatically synthesized lossless compression algorithm for floating-point data. In *the 2018 Data Compression Conference*. IEEE, Snowbird, UT, USA, 337–346.

[10] John Clyne, Pablo Mininni, Alan Norton, and Mark Rast. 2007. Interactive desktop analysis of high resolution simulations: application to turbulent plume dynamics and current sheet formation. *New Journal of Physics* 9, 301 (2007), 1–29.

[11] IEEE Standards Committee et al. 2008. 754-2008 IEEE standard for floating-point arithmetic. *IEEE Computer Society Std* 2008 (2008), 517.

[12] Community Earth System Model (CESM) Atmosphere Model. 2019. http://www.cesm.ucar.edu/models/. Online.

[13] Philippe Coussy and Adam Morawiec. 2008. *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer Science & Business Media, Dordrecht, Netherlands.

[14] L Peter Deutsch. 1996. GZIP file format specification version 4.3.

[15] Sheng Di. 2019. https://www.mcs.anl.gov/~shdi/download/sz-2.0-user-guide.pdf. Online.

[16] Sheng Di and Franck Cappello. 2016. Fast error-bounded lossy HPC data compression with SZ. In *2016 IEEE International Parallel and Distributed Processing Symposium*. IEEE, Chicago, IL, USA, 730–739.

[17] Tom Feist. 2012. Vivado design suite. *White Paper* 5 (2012), 30.

[18] Ian Foster, Mark Ainsworth, Bryce Allen, Julie Bessac, Franck Cappello, Jong Youl Choi, Emil Constantinescu, Philip E Davis, Sheng Di, Wendy Di, et al. 2017. Computing just what you need: Online data analysis and reduction at extreme scales. In *European Conference on Parallel Processing*. Springer, Springer, Cham, 3–19.

[19] Jeremy Fowers, Joo-Young Kim, Doug Burger, and Scott Hauck. 2015. A scalable high-bandwidth architecture for lossless compression on FPGAs. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, Vancouver, BC, Canada, 52–59.

[20] Daniel D Gajski, Nikil D Dutt, Allen CH Wu, and Steve YL Lin. 2012. *High-Level Synthesis: Introduction to Chip and System Design*. Springer Science & Business Media, Boston, MA, USA.

[21] Tong Geng, Tianqi Wang, Chunshu Wu, Chen Yang, Wei Wu, Ang Li, and Martin C. Herbordt. 2019. O3BNN: an out-of-order architecture for high-performance binarized neural network inference with fine-grained pruning. In *Proceedings of the ACM International Conference on Supercomputing*. ACM, Denver, CO, USA, 461–472.

[22] Ali Murat Gok, Sheng Di, Alexeev Yuri, Dingwen Tao, Vladimir Mironov, Xin Liang, and Franck Cappello. 2018. PaSTRI: A novel data compression algorithm for two-electron integrals in quantum chemistry. In *IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, Belfast, UK, 1–11.

[23] Leonardo A Bautista Gomez and Franck Cappello. 2013. Improving floating point compression through binary masks. In *2013 IEEE International Conference on Big Data*. IEEE, Silicon Valley, CA, USA, 326–331.

[24] Salman Habib, Vitali Morozov, Nicholas Frontiere, Hal Finkel, Adrian Pope, Katrin Heitmann, Kalyan Kumaran, Venkatram Vishwanath, Tom Peterka, Joe Insley, et al. 2016. HACC: Extreme scaling and performance across diverse architectures. *Commun. ACM* 60, 1 (2016), 97–104.

[25] W-J Huang, Nirmal Saxena, and Edward J McCluskey. 2000. A reliable LZ data compressor on reconfigurable coprocessors. In *Proceedings 2000 IEEE Symposium on Field-Programmable Custom Computing Machines (Cat. No. PR00871)*. IEEE, Napa Valley, CA, USA, 249–258.

[26] D. A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE* 40, 9 (Sep. 1952), 1098–1101.

[27] Hurricane ISABEL Simulation Data. 2019. http://vis.computer.org/vis2004contest/data.html. Online.

[28] Lawrence Ibarria, Peter Lindstrom, Jarek Rossignac, and Andrzej Szymczak. 2003. Out-of-core compression and decompression of large n-dimensional scalar fields. *Computer Graphics Forum* 22, 3 (2003), 343–348.

[29] Sriram Lakshminarasimhan, Neil Shah, Stephane Ethier, Scott Klasky, Rob Latham, Rob Ross, and Nagiza F Samatova. 2011. Compressing the incompressible with ISABELA: In-situ reduction of spatio-temporal data. In *European Conference on Parallel Processing*. Springer, Springer, Berlin, Heidelberg, 366–379.

[30] Xin Liang, Sheng Di, Sihuan Li, Dingwen Tao, Bogdan Nicolae, Zizhong Chen, and Franck Cappello. 2019. Significantly improving lossy compression quality based on an optimized hybrid prediction model. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, ACM, Denver, CO, USA, 33.

[31] Xin Liang, Sheng Di, Dingwen Tao, Zizhong Chen, and Franck Cappello. 2018. An Efficient transformation scheme for lLossy data compression with point-wise relative error bound. In *IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, Belfast, UK, 179–189.

[32] Xin Liang, Sheng Di, Dingwen Tao, Sihuan Li, Shaomeng Li, Hanqi Guo, Zizhong Chen, and Franck Cappello. 2018. Error-controlled lossy compression optimized for high compression ratios of scientific datasets. In *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, Seattle, WA, USA, 438–447.

[33] Xin Liang, Sheng Di, Dingwen Tao, Sihuan Li, Bogdan Nicolae, Zizhong Chen, and Franck Cappello. 2019. Improving Performance of Data Dumping with Lossy Compression for Scientific Simulation. In *2019*

IEEE International Conference on Cluster Computing (CLUSTER). IEEE, IEEE, Albuquerque, NM, USA, 1–11.

[34] Peter Lindstrom. 2014. Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics* 20, 12 (2014), 2674–2683.

[35] Peter Lindstrom and Martin Isenburg. 2006. Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics* 12, 5 (2006), 1245–1250.

[36] Tao Lu, Qing Liu, Xubin He, Huizhang Luo, Eric Suchyta, Jong Choi, Norbert Podhorszki, Scott Klasky, Mathew Wolf, Tong Liu, et al. 2018. Understanding and modeling lossy compression schemes on HPC scientific data. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, Vancouver, BC, Canada, 348–357.

[37] Andrew Martin, Damir Jamsek, and K Agarawal. 2013. FPGA-based application acceleration: Case study with gzip compression/decompression streaming engine. *ICCAD Special Session C* 7 (2013), 2013.

[38] Dirk Meister, Jürgen Kaiser, Andre Brinkmann, Toni Cortes, Michael Kuhn, and Julian Kunkel. 2012. A study on data deduplication in HPC storage systems. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, IEEE, Salt Lake City, UT, USA, 7.

[39] M. A. Mesa, A. RamÃŋrez, A. Azevedo, C. Meenderinck, B. Juurlink, and M. Valero. 2009. Scalability of Macroblock-level Parallelism for H.264 Decoding. In *2009 15th International Conference on Parallel and Distributed Systems*. IEEE, Shenzhen, China, 236–243.

[40] NYX simulation. 2019. https://amrex-astro.github.io/Nyx/. Online.

[41] PantaRhei cluster. 2019. https://www.dingwentao.com/experimental-system. Online.

[42] Weikang Qiao, Jieqiong Du, Zhenman Fang, Michael Lo, Mau-Chung Frank Chang, and Jason Cong. 2018. High-throughput lossless compression on tightly coupled CPU-FPGA platforms. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, Boulder, CO, USA, 37–44.

[43] Weikang Qiao, Zhenman Fang, Mau-Chung Frank Chang, and Jason Cong. 2019. An FPGA-Based BWT accelerator for Bzip2 data compression. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, San Diego, CA, USA, 96–99.

[44] Suzanne Rigler, William Bishop, and Andrew Kennings. 2007. FPGA-based lossless data compression using Huffman and LZ77 algorithms. In *2007 Canadian conference on electrical and computer engineering*. IEEE, Vancouver, BC, Canada, 1235–1238.

[45] Jörg Ritter and Paul Molitor. 2001. A pipelined architecture for partitioned DWT based lossy image compression using FPGA's. In *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*. ACM, Monterey, California, USA, 201–206.

[46] Lucana Santos, José Fco López, Roberto Sarmiento, and Raffaele Vitulli. 2013. FPGA implementation of a lossy compression algorithm for hyperspectral images with a high-level synthesis tool. In *2013 NASA/ESA Conference on Adaptive Hardware and Systems (AHS-2013)*. IEEE, Torino, Italy, 107–114.

[47] Naoto Sasaki, Kento Sato, Toshio Endo, and Satoshi Matsuoka. 2015. Exploration of lossy compression for application-level checkpoint/restart. In *2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, Hyderabad, India, 914–922.

[48] Scientific Data Reduction Benchmarks. 2019. https://sdrbench.github.io/. Online.

[49] H Sofikitis, K Roumpou, Apostolos Dollas, and N Bourbakis. 2005. An architecture for video compression based on the SCAN algorithm. In *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*. IEEE, Napa, CA, USA, 295–296.

[50] Seung Woo Son, Zhengzhang Chen, William Hendrix, Ankit Agrawal, Wei-keng Liao, and Alok Choudhary. 2014. Data compression for the exascale computing era-survey. *Supercomputing Frontiers and Innovations* 1, 2 (2014), 76–88.

[51] Yinqi Tang and Naveen Verma. 2018. Energy-efficient pedestrian detection system: Exploiting statistical error compensation for lossy memory data compression. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26, 7 (2018), 1301–1311.

[52] Dingwen Tao, Sheng Di, Zizhong Chen, and Franck Cappello. 2017. Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization. In *2017 IEEE International Parallel and Distributed Processing Symposium*. IEEE, Orlando, FL, USA, 1129–1139.

[53] Dingwen Tao, Sheng Di, Xin Liang, Zizhong Chen, and Franck Cappello. 2019. Optimizing lossy compression rate-distortion from automatic online selection between SZ and ZFP. *IEEE Transactions on Parallel and Distributed Systems* 30, 8 (2019), 1857–1871.

[54] David Taubman and Michael Marcellin. 2012. *JPEG2000 image compression fundamentals, standards and practice: image compression fundamentals, standards and practice*. Vol. 642. Springer Science & Business Media, Boston, MA, USA.

[55] S. Crusan V. Vishwanath and K. Harms. 2019. Parallel I/O on Mira. https://www.alcf.anl.gov/files/Parallel_IO_on_Mira_0.pdf. Online.

[56] Vivado Design Suite User Guide: High-Level Synthesis. 2019. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug902-vivado-high-level-synthesis.pdf. Online.

[57] Gregory K Wallace. 1992. The JPEG still picture compression standard. *IEEE Transactions on Consumer Electronics* 38, 1 (1992), xviii–xxxiv.

[58] M. E. Wolf and M. S. Lam. 1991. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems* 2, 4 (Oct 1991), 452–471.

[59] Xilinx GZip. 2019. https://github.com/Xilinx/Applications/tree/master/GZip. Online.

[60] Qingqing Xiong, Rushi Patel, Chen Yang, Tong Geng, Anthony Skjellum, and Martin C Herbordt. 2019. GhostSZ: A transparent FPGA-accelerated lossy compression framework. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, San Diego, CA, USA, 258–266.

[61] Zstd. 2019. https://github.com/facebook/zstd/releases. Online.

A Appendix: Artifacts

Accessible via this Google Drive link (history archive) and this GitHub repository.

A.1 Preface

A list of claims from the paper supported by the artifact, and how/why The artifact is in support of compression ration related results, which are shown in Table 7 and 8. We provide the step-by-step instruction all the related datasets in this appendix.

A list of claims from the paper not supported by the artifact This artifact does not support the throughput shown in the paper, because it needs specific hardware and proprietary toolchain, which are not applicable to reviewers.

A.2 Getting Started

Please untar/unzip artifact archive to `/tmp`.

Listing 2. Setup: SZ-1.4

```
wget -P /tmp \
  https://github.com/disheng222/SZ/archive/v1.4.13.5.zip
cd /tmp; unzip v1.4.13.5.zip
cd SZ-1.4.13.5
./configure --prefix=/tmp/sz1.4_install
make && make install
```

Modify `/tmp/SZ-1.4.13.5/example/sz.config` by changing line 40 to `max_quant_intervals = 32768`

Listing 3. Setup: waveSZ and GhostSZ

```
tar zvf ppopp20_artifacts.tgz -C /tmp/ppopp20_artifacts
cd /tmp/ppopp20_artifacts
make
```

A.3 Data Preparation

In our paper, CESM-ATM Dataset1 (1.47 GB), Hurricane ISABEL Dataset (1.25 GB), and NYX Dataset (2.7 GB) are used. Both the links and the commands work. These datasets are hosted on https://sdrbench.github.io

Listing 4. Data Preparation

```
mkdir /tmp/SDRBench
wget -P /tmp/SDRBench/ \
  https://97235036-3749-11e7-bcdc-22000b9a448b.e.globus.org/
      ds131.2/Data-Reduction-Repo/raw-data/CESM-ATM/
      SDRBENCH-CESM-ATM-1800x3600.tar.gz
wget -P /tmp/SDRBench/ \
  https://97235036-3749-11e7-bcdc-22000b9a448b.e.globus.org/
      ds131.2/Data-Reduction-Repo/raw-data/Hurricane-ISABEL
      /SDRBENCH-Hurricane-ISABEL-100x500x500.tar.gz
wget -P /tmp/SDRBench/ \
  https://97235036-3749-11e7-bcdc-22000b9a448b.e.globus.org/
      ds131.2/Data-Reduction-Repo/raw-data/EXASKY/NYX/
      SDRBENCH-EXASKY-NYX-512x512x512.tar.gz
cd /tmp/SDRBench
for i in *.tar.gz; do tar zxf $i; done
```

Listing 5. SHASUM

```
(optional) SHASUM
SDRBENCH-CESM-ATM-1800x3600.tar.gz
0994a5bfc797f3faf3f092a807422accee24f91dfa073f528ab7b76114986af5
SDRBENCH-Hurricane-ISABEL-100x500x500.tar.gz
ca5f8003ab4a650ea2d546eeb7ad6efd55b6e14281b361b365add41d0e7fa5ac
SDRBENCH-EXASKY-NYX-512x512x512.tar.gz
9df0fb4450d45f3d9c7cd1b5b59e9a7c1ec6ea0ab736d3e5ae46b5c52118e058
```

A.4 Step-by-Step Instruction

A.4.1 Execution

Listing 6. Conducting SZ-1.4 Experiments

```
SZ=/tmp/sz1.4_install/bin/sz
DIR=/tmp/ppopp20_artifacts
BESTSPEED=/tmp/SZ-1.4.13.5/example/sz.bestSpeed

rm -f ${DIR}/SZ_{bestSpeed,bestCR}_{CESM,Isabel,NYX}.txt
touch ${DIR}/SZ_{bestSpeed,bestCR}_{CESM,Isabel,NYX}.txt
SZC="${SZ} -z -f -c ${BESTSPEED} -M REL -R 1E-3"
SZD="${SZ} -x -f -s"

cd /tmp/SDRBench/1800x3600
SIZE="-2 3600 1800"
for DATA in *.dat; do
  LOG="${DIR}/SZ_CESM.txt"
  echo ${DATA} >> ${LOG}
  ${SZC} -i ${DATA} ${SIZE} >> ${LOG}
  ${SZD} ${DATA}.sz -i ${DATA} ${SIZE} -a >> ${LOG}
done

cd /tmp/SDRBench/100x500x500
SIZE="-3 500 500 100"
for DATA in *.f32; do
  LOG="${DIR}/SZ_Hurricane.txt"
  echo ${DATA} >> ${LOG}
  ${SZC} -i ${DATA} ${SIZE} >> ${LOG}
  ${SZD} ${DATA}.sz -i ${DATA} ${SIZE} -a >> ${LOG}
done

cd /tmp/SDRBench/512x512x512
SIZE="-3 512 512 512"
for DATA in *.dat; do
  LOG="${DIR}/SZ_NYX.txt"
  echo ${DATA} >> ${LOG}
  ${SZC} -i ${DATA} ${SIZE} >> ${LOG}
  ${SZD} ${DATA}.sz -i ${DATA} ${SIZE} -a >> ${LOG}
done
```

Listing 7. Conducting waveSZ and GhostSZ Experiments

```
DIR=/tmp/ppopp20_artifacts
waveSZ=${DIR}/cpurun; GhostSZ=${DIR}/cpurun
rm -f ${DIR}/{waveSZ,GhostSZ}_{CESM,Hurricane,NYX}.txt
touch ${DIR}/{waveSZ,GhostSZ}_{CESM,Hurricane,NYX}.txt

cd /tmp/SDRBench/1800x3600
for DATA in *.dat; do
  echo ${DATA} >> ${DIR}/{wave,Ghost}SZ_CESM.txt
  $waveSZ 1800 3600 1 -3 base10 ${DATA} wave VRREL \
    >> ${DIR}/waveSZ_CESM.txt
  $GhostSZ 1800 3600 1 -3 base10 ${DATA} ghost VRREL \
    >> ${DIR}/GhostSZ_CESM.txt
done
```

```
cd /tmp/SDRBench/100x500x500
for DATA in *.f32; do
  echo ${DATA} >> ${DIR}/{wave,Ghost}SZ_Hurricane.txt
  $waveSZ 100 250000 1 -3 base10 ${DATA} wave VRREL \
    >> ${DIR}/waveSZ_Hurricane.txt
  $GhostSZ 100 250000 1 -3 base10 ${DATA} ghost VRREL \
    >> ${DIR}/GhostSZ_Hurricane.txt
done

cd /tmp/SDRBench/512x512x512
for DATA in *.dat; do
  echo ${DATA} >> ${DIR}/{wave,Ghost}SZ_NYX.txt
  $waveSZ 512 262144 1 -3 base10 ${DATA} wave VRREL \
    >> ${DIR}/waveSZ_NYX.txt
  $GhostSZ 512 262144 1 -3 base10 ${DATA} ghost VRREL \
    >> ${DIR}/GhostSZ_NYX.txt
done
```

A.4.2 Gather Statistics

The compression ratio is obtained from calculations. And all the datasets are interpreted as 2D datasets. For instance, a 100x500x500, with the order of higher dimension to lower dimension (or outer loop to inner loop in waveSZ implementation) is interpreted as 100×(250,000) 2D dataset. And all points at border can be left out when calculating compression ration (further details found below).

We have two ways of using lossless compression to archive the quantization

1) use `gzip` on the raw quantization code, which is exported as `{wave,ghost}.code`
2) use customized Huffman encoding and then use `gzip` on the Huffman-encoded quantization code, which is exported as `*.{wave,ghost}.huffman`

```
for i in *.{code,huffman}; do
  gzip -c --fast $i > $i.gz.fast
  gzip -c --best $i > $i.gz.best
done
```

And the compression ratio is obtained by

$$\frac{\text{original.size}}{\text{lossy.compression.size}}$$

if we do not count the border points as unpredictable data ("verbatim" in the final archive), and this is annotated with "maximal possible compression ratio". Otherwise, with border points counted as unpredictable data ("verbatim"), it is calculated from

$$\frac{\text{original.size}}{(\text{lossy.compression.size} + \#\text{border.points}) \times \texttt{sizeof(float32)}}$$

To get the useful information,

Listing 8. SZ-1.4: extracting compression ratio and PSNR

```
# source: /tmp/ppopp20_artifacts/SZ_{CESM,Hurricane,NYX}.txt
# compression ratio
cat SZ_{CESM,Hurricane,NYX}.txt \
  | grep compressionRatio \
  | awk -F'=' '{print $NF}'
# PSNR
cat SZ_CESM.txt | grep PSNR \
  | awk -F',' '{print $1}' \
  | awk -F' = ' '{print $NF}'
```

Listing 9. waveSZ and GhostSZ: extracting compression ratio and PSNR

```
# source: /tmp/ppopp20_artifacts/{waveSZ,GhostSZ}_{CESM,
    Hurricane,NYX}.txt
# get the lossless compressed archive size in filesystem
ls -al \
  | grep {ghost,wave} \
  | grep {huffman,code}.tgz \
  | awk '{print $5}'
# get PSNR
cat {waveSZ,GhostSZ}_{CESM,Hurricane,NYX} \
  | grep PSNR \
  | awk '{print $NF}'
```

Please also refer to the organized statistics in `ppopp_artifacts.{xlsx,ods}`, whichever as open format works. A script for collecting results is posted on the GitHub repository.