GhostSZ: A Transparent FPGA-Accelerated Lossy Compression Framework

Qingqing Xiong* Rushi Patel* Chen Yang* Tong Geng* Anthony Skjellum† Martin C. Herbordt*
*Dept.of Electrical and Computer Engineering, Boston University
†Simcenter & Dept. of Computer Science and Engineering, University of Tennessee at Chattanooga

***Abstract*—High-performance computing (HPC) applications often generate enormous amounts of data that must be transferred for check-pointing, *in situ* processing, or post-execution analysis. To reduce the related network traffic and storage consumption, lossy compression schemes that target scientific data are often used. SZ compression emerged three years ago and has gained much attention because of its high compression ratio. However, performing SZ compression can take half a day per Terabyte of data; this could be a drawback to adoption.**

We propose *GhostSZ* an FPGA framework for accelerating tasks in SZ at line rate, and so transparently. The critical problem to be overcome is the tight data dependence central to SZ. GhostSZ solves this with a data transfer path having novel staged hardware. We test our implementation with both synthetic and real HPC application data and show $9.5\times$-$80\times$ core versus pipeline speedup over the optimized production version running on a state-of-the-art CPU and $8.2\times$ per chip. Much of the variance in performance is due to the FPGA already running at line rate and so benefiting less from optimizations applicable to the CPU only on the most favorable data sets. The significance of this work is the possibility of a major reduction in required networking and storage in HPC installations. For example, using GhostSZ, fewer than 10 FPGAs would be sufficient to handle the entire I/O bandwidth of the top entry on the latest IO-500 list.

***Index Terms*—Lossy Compression, In-Line Acceleration, FPGAs, High-Performance Computing**

I. Introduction

Scientific simulations continue to grow ever larger and produce increasingly extreme amounts of data: the Community Earth Simulation Model generates hundreds of TBs of data in a few tens of seconds [1]. Transferring and storing such large data sets introduces heavy network traffic and can easily exceed available storage capacity and network bandwidth. Compression is central to mitigating this problem.

Traditional deduplication of redundant files and lossless compression methods (e.g., LZ77 of Gzip [2]) guarantee complete data recovery, but can provide a poor compression ratio on scientific data that are usually single/double precision floating point. In recent years lossy compression methods have been explored to provide a higher compression ratio, especially for such scientific data; adoption in check-pointing, *in situ* processing, and post-execution data analysis is a growing trend. A fundamental assumption is that the introduction of small errors into simulated reality is generally worth the massive compression ratio that lossy compression enables.

The state-of-the-art lossy compression tool for HPC scientific simulations is perhaps SZ, which produces compression ratios $2\times$-$10\times$ higher than the next best method [3]. SZ exploits two characteristics of scientific data. The first is that adjacent physical data, e.g., in an electrostatic or climate model, are likely to be correlated. The second is that many values in the IEEE 754 floating point (FP) format need few bits, especially when close to zero. SZ is thus a multistage framework with the first being curve fitting, the next an FP-specific transform, and the final being a pass of lossless compression. These operations are very costly, however; in one application it takes one CPU one hour to generate one TB of data, but more than half a day to compress it, and several hours to decompress [3]. One key reason for the slow performance is simply the large number of flops required, but data dependencies during curve-fitting are also a problem as they inhibit vectorization.

Ideally the data would be compressed in-line at the point of generation and decompressed at the point of use. In the last few years FPGAs have become ubiquitous in large data centers to deal with just such problems. Both Smart-NIC [4] and Bump-in-the-Wire [5], [6] models are appropriate here. Besides configurability and high performance, other FPGA advantages include external connectivity and tight coupling of application logic with the communication data plane [7]–[10].

In this paper, we propose **GhostSZ**, a framework that contains a novel hardware design based on SZ. GhostSZ is a five-step framework that contains: offline configuration, linearization, curve-fitting, quantization, and Gzip. Of these, all but offline configuration can be part of SZ. One other change is replacing the binary representation in the original SZ with the quantization method found in later versions of SZ; we find that this best exploits FPGAs' attributes.

Our primary contributions are, first, the GhostSZ system, which contains novel design features and is the first implementation following SZ compression on FPGAs; and second, the improvement in performance over the newest software version of SZ (SZ-2.0.2.0) by $10\times$-$85\times$ (core versus pipeline) and $8.2\times$ per chip with a similar peak signal-to-noise ratio (PSNR) and 25% higher compression ratio on average. The significance is that GhostSZ is able to compress data with nearly no impact on time-of-flight and with modest hardware cost. This could substantially reduce requirements for network

This work was supported in part by the NSF through awards CNS-1405695, CCF-1618303, and CCF-1821431; by the NIH through award 1R41GM128533; by a grant from Red Hat; and by Intel through donated FPGAs, tools, and IP.

bandwidth and storage consumption for scientific computing.

The remainder of this paper is organized as follows: We cover related work in § II. Next, we give the background on SZ compression in § III. In § IV we present our hardware design and the corresponding optimizations. We then describe our testbed, the input data sets used, and the baselines in § V. We follow up in § VI with results and analysis. Lastly, we offer conclusions and future work in § VII.

II. Related Work and Overview

There has been little previous work on FPGA implementation of lossy compression, and what there is has been for methods related to image processing and not for scientific computing [11]–[15]. Santos et al. [13] implement the LCE algorithm (lossy compression for Exomars) that has similar procedures as SZ. LCE contains a 2D predictor and a quantization module; however, it is designed for image data thus does not require long-latency floating point arithmetic operations. Also, LCE uses a simple predictor that takes the mean-value of two neighboring points.

In the domain of scientific computing, previous lossy compressors [16]–[18] appear to have been superseded by SZ [19], so that is our focus. Since the original publication [3], the SZ group has provided many optimizations [20]–[25] such as for increasing fidelity (reducing the compression error), increasing the compression ratio, supporting users' diverse error-bounds, and improving performance.

In this first study we have two goals. The first is to provide a framework for current and future modules. The second is to implement the most important of these modules. For experimental comparisons, we always use the latest SZ (SZ-2.0.2.0) which includes all the optimizations mentioned above, some of which are available by specifying configurations.

III. Introduction to SZ Compression

SZ compression is a lossy compression tool initially targeting scientific data, where a high compression-ratio is hard to achieve with state-of-art compression tools. SZ adopts domain knowledge for compression and allows users to define an error-bound based on allowable data variation. It contains four major steps [26]: 1) linearizing the data; 2) using a best-fit curve fitting model to predict data points based on preceding points; 3) compressing unpredicted data by analyzing their binary representation; and 4) further compression using Gzip.

Along with these four steps, SZ also contains various performance/accuracy optimization methods (implemented in different tool versions). In this paper, we maintain the four step arrangement and keep the first two and the last steps mostly as they are, but we strategically select the quantization method as the third step for best exploiting FPGA capabilities.

A. Linearization of Multidimensional Arrays

SZ targets scientific data where the data are typically stored as multidimensional arrays, for instance, temperature data for each point in 3D space. The second step of the SZ sequence, linearization, requires serialized input and ideally uses the intrinsic memory sequence of the data array as the input to the curve-fitting model.

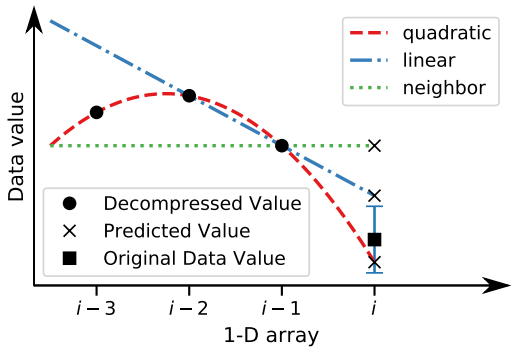

Fig. 1. Illustration of best-fit curve-fitting model. The decompressed values are used as the proceeding values for fitting. In this figure, the quadratic fitting result is within the error bound and is the best fit.

B. Best-Fit Curve Fitting

As the data enter one by one, the curve fitting model tries to predict the next data point from the preceding neighbors' values; if successful, it can then represent the data point as an encoding of the fitting model. SZ mainly uses three fitting models, corresponding to 0th, first, and second orders. The first model predicts the current point as the preceding neighbor's predicted value. The second uses linear interpolation; it predicts the current point i from the two preceding values, following $X_i = 2X_{i-1} - X_{i-2}$ (X denotes the predicted value). The third uses quadratic interpolation using the three preceding values, following $X_i = 3X_{i-1} - 3X_{i-2} + X_{i-3}$. The fitting models are illustrated in Fig. 1.

In this scheme, the entire stream can be compressed simply as the first value followed by a record of the interpolation order that best represents each following point. Assuming that the fitting model type can be represented with 2 or 3 bits, the data is compressed from 32-64 bits to 2-3 bits. If the predicted value cannot meet the error bound requirement, then the fitting fails, the data point is encoded as *unpredicted*, and the procedure goes to the next compression step.

C. Floating-point Data Binary Representation Analysis

For the unpredicted data points, SZ analyzes the IEEE 754 representations and reduces the number of bits. This is possible because the closer the data is to zero, the fewer mantissa bits it requires for representing the data within the error bound. SZ first normalizes the unpredicted data to the middle-value ($\frac{1}{2}(min+max)$) for making them closer to zero, then removes the insignificant bits in the mantissa, and finally uses an XOR-leading-zero-based floating point compression method.

D. Adaptive Error-Based Quantization

Alongside the binary representation method just described, new versions of SZ compression [22] provide an error-based quantization method for compressing the unpredicted data. During the first phase of curve-fitting, if the predicted result exceeds the error bound, SZ shifts the result by a multiple

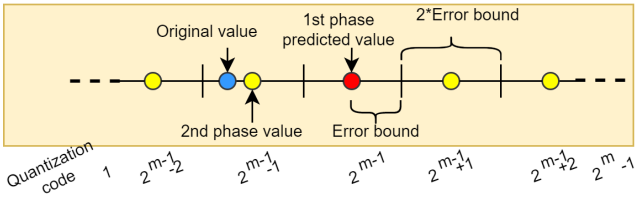

Fig. 2. Error-bound quantization illustration.

of 2∗error-bound. When this result is used, the number of error bounds shifted up or down is stored along with the fitting model (m bits represent 2^m different shift amounts, see Fig. 2). Therefore some unpredicted points in Step 2 can now be predicted with the use of added intervals. The size of m must be chosen with care: the bigger m is, the more types of shifts this method provides; but the more storage is consumed for storing m. SZ uses an adaptive method to decide how big m should be by evaluating the prediction hit rate and giving the user suggestions as to a preferred size.

E. Further Compression with Gzip

SZ further compresses the data by using the lossless compressor Gzip on the compressed byte stream produced from the previous steps. The results from previous steps are 2-bit encodings, m-bit quantization codes and the unpredicted data; these usually result in a high repetition count for each type of code, which Gzip handles well (LZ77 algorithm [2]).

F. Error-Bound Support

Lossy compressors generally use error-bounds to control data distortion. It indicates the biggest allowed difference between the original and decompressed values. There are many types of error-bounds such as absolute, relative, and point-wise. SZ currently supports all three.

IV. Design and Implementation

In this section, we describe the design and implementation of the GhostSZ compression method. A goal was to make minimal changes when introducing FPGA acceleration. The linearization step is unchanged as it exploits data locality and it avoids extra performance overhead from the linearization algorithm. We currently support absolute error-bounds when implementing multiple GhostSZ pipelines on the FPGA. A single GhostSZ pipeline can be configured with one absolute value as the error-bound but can be adapted to use a "regional" error-bound.

A. Five-step GhostSZ Compression Framework

When mapping software algorithms to FPGA hardware designs, their modularity and synchronization properties are significant to the success of the design. We describe the five steps of the GhostSZ FPGA framework:

I *Offline configuration*: The user selects parameters for the to-be-compressed data file: grid data dimension size, data type (float/double), error-bound, and optionally how many pipelines to use. This step is essential for GhostSZ to configure the hardware design (e.g., width of each data file, FIFO depth required).

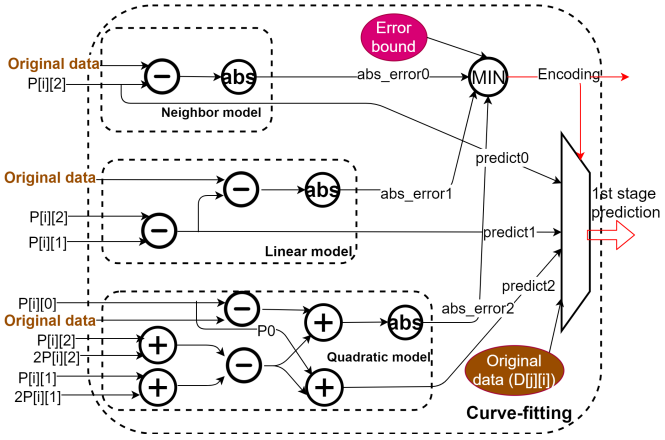

Fig. 3. Curve-fitting pipelines. Arithmetic operations are floating point.

II *Linearization* moves the to-be-compressed data to the FPGA designs based on the data location in memory. With the partitioning or blocking methods enabled, GhostSZ linearizes data in parallel to multiple pipelines by concatenating the read data from several blocks and sending them to different hardware pipelines.

III *Curve-fitting*: After the to-be-compressed data arrives at the hardware, it first goes through the curve-fitting pipeline, which tries to predict the original data with 3 fitting models. If successful, the output is a 2-bit encoding. If the prediction cannot meet the error bound, then information about the current data, e.g. the error, gets passed to the next step.

IV *Error-bound Quantization* takes the error calculated from the previous stage and calculates the quantization code as the output. If the error cannot be quantized, then the (all-zero) quantization code and the original data value are stored.

V *Gzip* hardware (IP) losslessly compresses all data.

These steps contain many design details which are described in the following subsections.

B. Hardware Design of Best-fit Curve-Fitting

The best-fit curve-fitting algorithm includes three fitting models which are implemented as three hardware pipelines as shown in Fig. 3. Each pipeline contains various floating point IPs for arithmetic operations.

1) Challenges: Mapping the fitting models to hardware encounters two major challenges: synchronization between pipelines and the introduction of *pipeline bubbles* by the feedback loop. To deal with the synchronization issue we insert parameterized shift registers. For instance, in the linear model shown in Fig. 3, the original data needs to wait for the output of the first subtractor for 14 cycles; this waiting is done by adding a shift register for the original data.

Pipeline bubbles are due to data dependencies between processing of successive data points. Data arrive on every cycle, but the longest fitting-pipeline takes 50 cycles (for generating the prediction and the 2-bit encoding). To fit the next data point, the previous prediction result is required; this

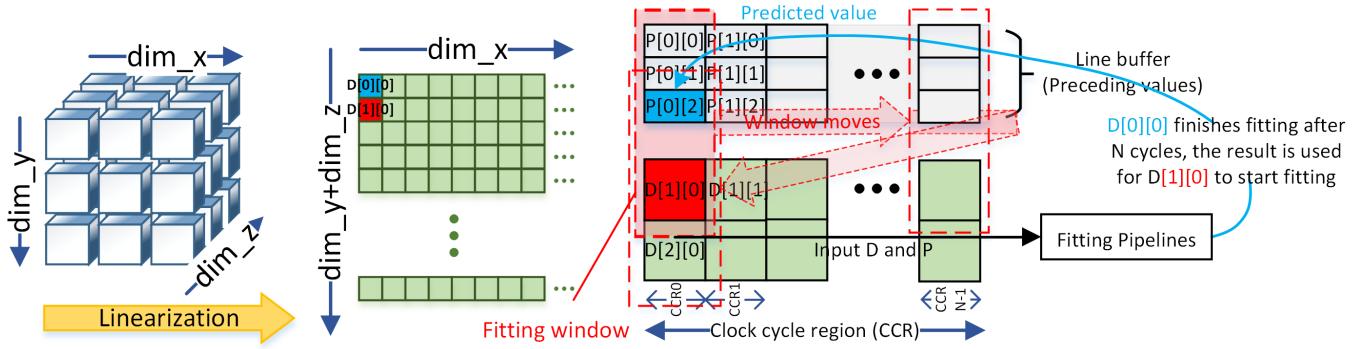

Fig. 4. Reorder fitting design. We fit orthogonal to the data streaming direction. The preceding values P are stored in the line buffers. Here, dim_x means x direction's size, and D denotes the original value. Data streams in from x direction, and the fitting window also moves along x direction.

results in a 50-cycle pipeline bubble. This is far too long to be overcome through simple data forwarding.

2) Reorder Fitting: To solve the pipeline bubble problem we take advantage of the multiple dimensions in the data; rather than fit the data in the streaming direction, we fit in an orthogonal dimension with a *reorder fitting method* (shown in Fig. 4). SZ works under the assumption that the grid's dimensions and types are known, and that neighboring points are correlated. Assuming data have similarly strong correlation in all dimensions, we can fit in any direction. If one dimension is strongly preferred, then a corner turning operation can be added.

In the reorder fitting design, a shift register array is used (similar to the line buffers in image processing) where each shift register stores 3 preceding values; the fitting window moves in the array among the critical cycle regions (CCR), following round-robin order. We give an example here with Fig. 4. In Cycle50, the fitter generates one result from 50 cycles ago for D[0][0] (D denotes the original value) and the corresponding shift register shifts to store the result as the new preceding value P[0][2]. Meanwhile, the fitting window takes the new proceeding values P[0] and tries fitting D[1][0]. The fitting window is designed as three staged-multiplexers for selecting the correct preceding values; this adds four extra stages to the critical loop length (CL).

3) Data Correlation: For exploiting the correlation between data in one direction (e.g. y), it is necessary to guarantee that the preceding data and the current data points are neighbors in the grid. It is possible to parameterize the pipeline length (the number of shift registers in the array) as the x dimension's size (dim_x) to guarantee the correlation. When dim_x is smaller than the CL, to maintain the data correlation, we can either rearrange the data or reduce the CL by choosing a lower-frequency configuration for the floating point IPs. When dim_x is bigger than the CL, we extend the line buffers and the CL to be dim_x; thus extra resources are required for: the line buffer, the buffer for extending the three fitting models, and the bigger multiplexer for moving the fitting window. It is also possible to prevent the overhead and also maintain the correlation via partitioning the data in the x direction. In this case, the fitting pipeline keeps the original CL as the line buffer's size; this might introduce overhead in reordering the data in memory or more read/write cache misses. Since the data access patterns are regular-strided after partitioning, this overhead might be negligible with CPU prefetching. Partitioning also potentially brings performance benefits when combined with blocking; details of the benefits are in § IV-E.

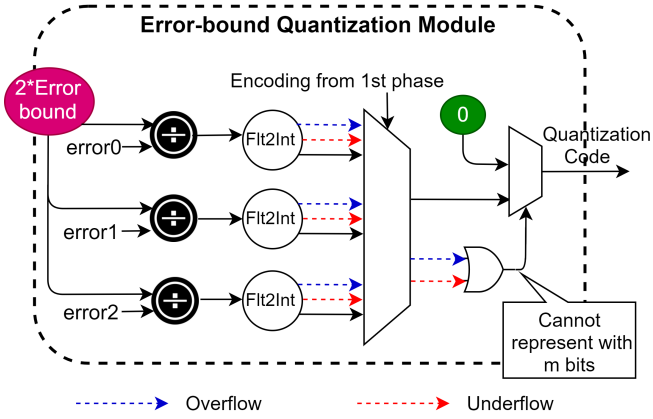

Fig. 5. Parameterizable error-bound quantization (PEBQ) module pipelines. The m is parameterizable by users.

C. Parameterizable Error-bound Quantization

For data points that cannot be predicted using the fitting models, SZ provides an adaptive error-bound quantization approach to compress those data points. For instance, if the distance between the prediction and the original data is about n times of the error-bound, SZ uses m bits to represent the distance and keeps the prediction result. How big m should be is decided adaptively in SZ. GhostSZ provides a parameterizable error-bound quantization (PEBQ shown in Fig. 5) approach that is similar to SZ's adaptive approach. In the PEBQ, m is set to a default value and can be changed by the user.

To fit the calculation of the quantization code into the design, we evaluate the representation of the quantization code and pick the one with the least resource/latency overhead. SZ's original quantization code, shown in Fig. 2, might result in extra computation on FPGAs for conversion when decompressing from the quantization code. Instead, we design

the calculation based on

$$Quantization\ code = toint\left(\frac{error}{2 \times Error\text{-}bound}\right). \quad (1)$$

The resulting quantization code is in two's complement. A positive quantization code indicates that the first-phase predicted value is bigger than the original value, while negative code indicates that it is smaller.

In the PEBQ hardware design, shown in Fig. 5, we maximally overlap the curve fitting pipeline with the PEBQ. We input the error result of each fitting model to PEBQ, and design three identical pipelines for calculating the quantization codes. In each pipeline, we use floating point IP to convert the division result to a variable-length integer (which can be parameterized, the default is set to 8 bits). From the IP, we output overflow and underflow signals for indicating the case that m bits cannot represent the error. The selection of the quantization codes from three pipelines is based on the best-fit result encoding from the first phase.

To provide an adaptive design, we feed back the hit rate of the PEBQ to the user. We use a register to record the hit count (when the quantization code is non-zero). We calculate the hit ratio with the hit count and the unpredicted data count. In the FPGA design, the change of m is translated to a new PEBQ configuration with an updated m value. Note that quantization of each unpredicted data point results in $m + 2$ bits; the leading 2 bits store the best-fitting model's encoding ($2b00$ for the first fitting phase, but the closest model's encoding for the second quantization phase). Since the PEBQ can be detached from the reorder-fitting model, it is possible to support partial reconfiguration of the PEBQ and keep the reorder-fitting model. This trade-off needs to be investigated further.

D. Gzip

There are many FPGA Gzip products on the market [27], [28]; we use an approachable design example [28] written in C++. We use Xilinx Vivado High-Level Synthesis [29] to generate the hardware design and adapt it to our Intel FPGA. The FPGA design includes most of the major components of the Gzip design including LZ77 and Huffman Encoding. We are aware that there has been much research about Gzip hardware designs, e.g., [30], [31]. The Gzip hardware can be easily replaced for further optimization.

E. Performance Optimization Method: Blocking

A method of blocking the input data is used to provide further performance improvement over the reorder-fitting design. Blocking the input data is an obvious optimization but requires additional logic. The first new component is to support streaming multiple blocks at the same time. The second supports the ability to stream into multiple blocks of output values and concatenate the blocks together. Since SZ takes domain knowledge of the data as an input for compression, it is reasonable to put the blocking component in the CPU; SZ gives users an option to set the blocking number.

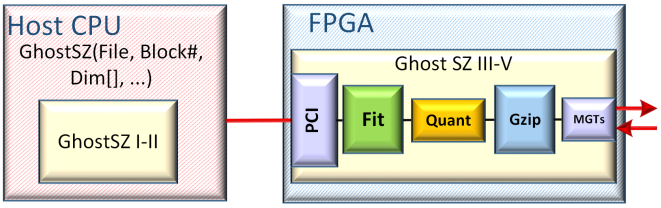

Fig. 6. Example system architecture with GhostSZ. MGTs are the multi-gigabit transceivers. Fit is the curve-fitting and Quant is quantization module.

How to block the data is a challenging problem. We mention the necessity of data partitioning in § IV-B2 when attempting to reduce FIFO overhead; this happens in the x dimension in our example (Fig. 4). Blocking can be implemented in all directions, preferably the highest dimension (z in the example) to reduce cache misses when splitting or writing back data. However it is also possible to block along the same direction, as partitioning requires a single solution for both issues/demands.

F. Decompression

Decompression is a significant component for the completion of a compressor. Many compression papers [3], [19], [31] mention little about decompression, since decompression usually follows the exact opposite flow of compression; this is also mostly true for GhostSZ. The only exception is that GhostSZ's hardware design of the decompressor has different input and output entities and contains a few different hardware arithmetic units. For instance, in the decompressor, the necessary outputs from the three models are three fitting results instead of the absolute value of the error. Also, the quantization module uses a multiplication operation to calculate the exact shifts recorded (instead of the division in Fig. 5). The exact shift is then accumulated to the selected output of the fitting models as the decompressed value.

G. System Design

The system design is shown in Fig. 6. Steps I and II are on the host CPU side, and the hardware designs of Steps III-V are on the FPGA. The host CPU has the interface for using GhostSZ framework. When the data to be compressed are transferred to the FPGA, the compression happens in the FPGA hardware design. The FPGA is configured with the PCIe hardware design for communicating with the Host CPU. In real-life use cases, the compressed data needs to be transferred to other devices such as storage or another CPU. FPGAs have high-throughput multi-gigabit transceivers (MGTs), which enable fast data transfer to other devices. In this paper, we demonstrate the use case of data being transferred/compressed via FPGA and then looped back to the CPU. Other use cases are discussed below.

V. Experimental Setup

In this section, we describe the experimental setup including the testbed, baselines, input data, and evaluation metrics.

A. Testbed

Our testbed consists of two systems. The first contains one CPU node on the Boston University Shared Computing Cluster (SCC) [32]. The CPU is an Intel Xeon E7-8867 v4 running at 2.40GHz. The second system is an Intel Xeon E5-2620 v2 @ 2.10GHz connected to an Intel Arria10ax115 20nm FPGA via PCIe-gen3 (8 lanes). Software designs are compiled using gcc-7.2.0 -O3.

GhostSZ was developed with Intel Quartus Prime Pro 18.1 [33]. We use Intel floating point IPs for floating point arithmetic, and the IP configuration is set, when possible, for the highest frequency. We configure the number of quantization bits to 14 which allows PEBQ to quantize most of the unpredicted data (based on our tests). We vary the fitting pipeline's depth (the CL) to match the size of the x dimension of each input.

B. Baseline Codes

We have two comparison baselines: the most recent release of SZ (SZ-2.0.2.0) and our tailored implementation of the SZ algorithms, i.e., software versions of the selections of SZ we use in GhostSZ.

SZ-2.0.2.0 includes various optimization methods, such as 2D fitting as opposed to 1D fitting [22], binary representation analysis [3], and a linear regression-based predictor [24]. Each optimization might impact performance and compression ratio; therefore, comparing our hardware design only with SZ-2.0.2.0 is insufficient.

Tailored SZ contains the parts of the SZ algorithm used on GhostSZ and coded to match the hardware implementation. It is written in C++ and contains linearization, 1D fitting (in-order fitting as opposed to GhostSZ's reorder fitting), quantization with 14 bits, and a Gzip function call.

Parallel compression is enabled in two ways in SZ: 1) SZ is called independently by multiple processes and used to compress multiple files (batch-mode); and 2) Using OpenMP [34] in SZ for achieving parallelism. We experimented with the second case; the first is discussed in the results section.

C. Input Data

We tested our design with both synthetic data (SZ_Testdata) and real scientific/scientific simulation data from several domains, including the Eddy hydrodynamics simulator (HD) [35], climate simulations CICE-Month and CICE-Snow (CLI) [1], and astrophysics simulations Sky and Atlas (ASTRO) [36]. The data sources are in binary. Details are in Table I.

We ran Eddy and generated results for 2D grids with 128×128 points containing velocity, pressure, and coordinates. We gathered 10 iterations' data for 10 files in total. We downloaded the other data directly online. CICE-Month and CICE-Snow have 2D data with x dimension size of 384, but various y sizes. Sky is 2D with 256 points in the x direction; Atlas is also 2D with 104 points in the x direction. For testing parallel compression, we blocked Atlas evenly in the y direction; SZ-2.0.2.0 code blocked the data on both x and y directions (x direction only for 2 threads).

We use 10^{-4} as the error-bound for all input data; this is the default error-bound in the SZ configuration file.

Scientific data might be stored in various formats such as netCDF, HDF5, and some application-specific formats; processing these formats might need extra support. For example, CCSM data is stored in netCDF format; SZ recommends users to use the netCDF reader's library for reading this format and then calling SZ to compress the data. We preprocessed the input data formats from applications, and input the same data to two baselines and GhostSZ. Note that for almost any format, creating an in-line FPGA pre-processor is trivial.

D. Evaluation Metrics

GhostSZ was evaluated with respect to latency, compression ratio, and compression quality (errors). We also measured FPGA properties such as resource consumption and operating frequency.

1) Latency: For a fair comparison, we did not take the file reading/writing time into account to guarantee that time is not spent on disk I/O; rather that the entire latency is for doing the compression. Specifically, the latency measured for GhostSZ is the time between sending the first data from the host to receiving the last compressed result from the FPGA. For the two software baselines, the method is analogous.

2) Compression Ratio: is usually evaluated as $\rho = S_o/S_c$, where S_o is the original data size and S_c is the compressed data size.

3) Average Error in Compression: There are many ways to measure the error in compression, including the cumulative distribution of the error [3], the normalized root mean square of the error (NRMSE), and the peak signal-to-noise ratio (PSNR) [16], [22]. We used both NRMSE and PSNR. First, we calculated the root mean squared error (RMSE) with equation 2:

$$rmse = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(abs\_error_i)} \tag{2}$$

where n represents the total number of data points and abs_error_i denotes the absolute error value of data point i. Then $nrmse = \frac{rmse}{Rx}$ (Rx denotes the values' range of the original data). The PSNR was calculated with equation 3:

$$psnr = 20 \times \log_{10}(\frac{RX}{rmse}) \tag{3}$$

where PSNR represents the size of the RMSE relative to the peak size of the signal. In the compression error results, the smaller the NRMSE is, and the bigger the PSNR is, the higher the quality of the reconstructed data.

VI. Results and Analysis

Throughput performance, FPGA resource and clock frequency, and two metrics related to compression quality results are shown. We then discuss possible use-case scenarios and the performance scalability of GhostSZ.

TABLE I
INPUT DATA INFORMATION.

Domain	Name	Source	Description	Total Size
Synthetic	SZ_Testdata	Unkown	Test floating point grid data in SZ-2.0.2.0 source code.	32.8KB
HD	Eddy	Nek5000 [35]	2D solution to Navier-Stokes equations.	1.97MB
CLI	CICE-Snow	CESM [1]	Community sea-ice simulation based on Community Earth System Model (CESM).	4.91MB
	CICE-Month			374MB
ASTRO	Sky	SDSS [36]	Data taken by the fourth phase of the Sloan Digital Sky Survey (SDSS) from 2014-2017.	305MB
	Atlas			2.67GB

TABLE II
COMPRESSION THROUGHPUT AND LATENCY FOR DIFFERENT INPUT DATA.

	Throughtput (MB/s) / Latency		
Input Data	SZ-2.0.2.0	Tailored SZ	GhostSZ
Eddy	16.7 / 118ms	39.8 / 49.5ms	808.9 / 2.5ms
CICE-Snow	10.3 / 475ms	55.4 / 88.7ms	821.9 / 6.0ms
CICE-Month	87.2 / 4.29s	85 / 4.4s	824.4 / 449ms
Sky	63.8 / 4.78s	35.5 / 8.6s	820.5 / 336ms
Atlas	83.7 / 31.9s	44.9 / 59.4s	833.9 / 3.2s

A. Performance results

In this section, we first show the results for a "per core" scenario: we use only one CPU core to run software SZ and one FPGA pipeline to run GhostSZ. We then show the on-chip parallel scenario: SZ enables OpenMP where up to 16 threads run concurrently and GhostSZ utilizes up to 8 hardware pipelines (with the current Arria-10 FPGA board, the practical limit is much higher).

1) Single-Core VS. Single-Pipeline: The compression latencies and throughputs for SZ, GhostSZ, and our reference code are shown in Table II. We observe that Tailored SZ, although it implements fewer functions than SZ, is usually slower. This is due to extra time spent in Gzip for Tailored SZ; it mostly requires quantization for each prediction result, it therefore leads to a larger input size to Gzip. We pick the faster of the two, SZ-2.0.2.0, for performance comparison. We observe the following.

- GhostSZ's throughput is $9.5\times$-$80\times$ of SZ-2.0.2.0.
- GhostSZ's latency is always proportional to the input data size; this demonstrates that the pipeline has no bubbles. The software versions do not show the same proportionality because compression time also depends on other factors such as correlation between neighboring points in the source data (e.g., the fraction of data points that can be fitted do not need the quantization step).
- The measured average throughput of Tailored SZ and SZ-2.0.2.0 is 52MB/s, which is a small fraction of a solid-state drive's (SSD) maximum bandwidth (around 700MB/s [37]). GhostSZ's average throughput is 821MB/s and exceeds the bandwidth of one SSD. However, in parallel network file systems where the bandwidth is much higher (up to 0.5TB/s [38]), all three designs can benefit from parallel compression to provide a higher throughput.

2) On-Chip Parallel Compression: SZ-2.0.2.0 supports parallel compression with OpenMP (OMP) for power-of-two thread counts; we varied the OMP thread count from 1 to 32 and measured the performance. For GhostSZ, we blocked the input data on y dimension according to the parallelism count, from 1 to 8 due to our 8-lane PCIe's bandwidth limitation. We ran two designs with Atlas, which is the largest data set we have, and which also gets a relatively high compression throughput from SZ. We show the parallel compression performance of Atlas data in Fig. 7.

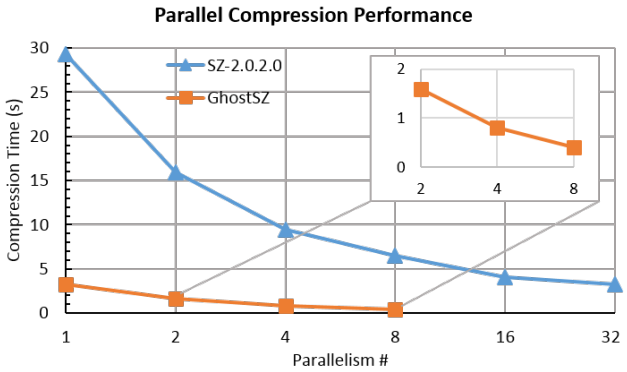

Fig. 7. Compressing Atlas data. Parallel compression time comparison between SZ-2.0.2.0 and GhostSZ. Parallelism # indicates the number of OpenMP threads used for SZ-2.0.2.0 and the number of GhostSZ hardware pipelines implemented.

From Fig. 7, we observe four major points: 1) GhostSZ's performance scales linearly with the pipeline count. The result is an aggregate throughput of 50gbps for eight pipelines. 2) SZ-2.0.2.0's performance does not scale linearly with the thread count; the reasons are most likely due to memory contention, but this needs to be examined further. 3) As a result, the advantage that GhostSZ has over SZ-2.0.2.0 increases from around $9\times$ to $16\times$ when both have $8\times$ parallelism. 4) When using 32 cores on the Xeon E7-8867 v4 CPU, 8-pipeline GhostSZ on an Arria10 FPGA still outperforms SZ-2.0.2.0 by $8.2\times$; this "per socket" improvement can be larger with a 16-lane PCIe and with another data set (e.g., Sky; it has a lower software SZ compression throughput than Atlas based on the single-core result).

B. Resource and Frequency Results

We present the FPGA resource consumption when implementing the GhostSZ with 104 as the line buffer width (for Atlas). The results are shown in Table III. The Gzip design consumes similar on-chip logic as one GhostSZ pipeline without Gzip. Scaling is limited by BRAM (RAM blocks) consumption by Gzip and the PCIe-gen3 communication bandwidth (8-lane). We measure a running frequency of 228 MHz with some degradation when using multiple pipelines (still around 200MHz).

TABLE III
RESOURCE UTILIZATION RESULTS. OTHERS INLCUDE MGTS, PCIE.

	GhostSZ (No_Gzip)	Gzip	Others	Total %	Total for 8 pipelines %
ALMs	9062	10726	15791	8%	42%
DSP	10	0	0	<1%	5%
BRAM	1	234	219	16%	76%
RX+TX	0	0	8	36%	36%

TABLE IV
COMPRESSION RATIO RESULTS COMPARISON. GHOSTSZ PERFORMS SIMILAR TO SZ-2.0.2.0.

	SZ_2.0.2.0	GhostSZ
SZ_Testdata	20.71	35.93
Eddy	2.43	3.61
CICE-Snow	11.03	12.48
CICE	6.94	9.16
Sky	2.09	1.81
Atlas	3.39	3.20

C. Compression Ratio and Compression Error Results

Compression ratio results are shown in Table IV. SZ_Testdata has a high compression ratio in general; this is due to a large number of repetitions and that neighboring points are highly correlated. Overall in this data set, GhostSZ is able to provide an average 25% higher compression ratio than SZ-2.0.2.0.

Compression error results for four data sets are shown in Table V. In the results shown, the bigger the PSNR, the lower the NRMSE, and the better the compression quality. On average, GhostSZ's PSNR and NRMSE are both very similar to SZ-2.0.2.0's. Both GhostSZ and SZ-2.0.2.0 meet the error bound requirement of 10^{-4}.

D. Scalability Discussion

SZ compression may take hours or even half a day to compress 1TB of scientific data. Because GhostSZ's compression latency is proportional to the input data size, we are able to project the 8-pipeline GhostSZ's throughput (6.1GB/s) on the same size of data; GhostSZ only takes about 3 minutes. Since there are no bubbles in GhostSZ's pipeline, the latency comes entirely from data transfer. For an Arria 10 with 11 pipelines, GhostSZ has a throughput of 8.4GB/s. However, current generation FPGAs like the Xilinx Ultrascale+ have roughly $8\times$ the resources of the Arria 10. Since the pipelines run independently, this yields a throughput of over 60GB/s.

Perhaps the highest impact deployment scenario is for the GhostSZ FPGAs to be integrated into large-scale systems using MGTs as needed to fill the pipelines. Compression is likely to happen in parallel with each process running a part of the simulation and compressing its part of the data. Supercomputers generally use parallel file systems with peak bandwidth between tens to hundreds of GB/s [38]. Assuming the whole bandwidth of the top entry on the IO-500 list (0.5TB/s) is used for writing the compressed data, the software implementation of SZ needs at least 10k cores–assuming perfect scalability and each providing a throughput of 50MB/s–to reach the maximum capability of the storage. GhostSZ can do this with around 600 pipelines. Taking only the resource consumption into account (assuming use of 90% of Arria 10 resources, each Arria10 board fits 40 pipelines not including Gzip), this means when implementing GhostSZ with no Gzip hardware, we need only around 15 Arria 10 FPGAs. When adding Gzip, we need around 50 Arria 10 FPGAs or fewer than 10 Ultrascale+ FPGAs.

TABLE V
COMPRESSION ERROR RESULTS: NORMALIZED ROOT MEAN SQUARED ERROR (NRMSE) AND PEAK SIGNAL-TO-NOISE RATIO (PSNR).

	NRMSE		**PSNR (dB)**	
Application	SZ2.0.2.0	GhostSZ	SZ2.0.2.0	GhostSZ
SZ_Testdata	1.2E-5	1.1E-5	98.47	98.90
Eddy	9E-6	8E-6	101.07	101.45
CICE-Snow	1.0E-4	7.9E-5	79.91	82.02
CICE	9.2E-5	8.1E-5	77.9	79.8
Sky	2.1E-6	1.9E-6	127.83	128.36
Atlas	0	0	167.45	168.32

VII. CONCLUSION AND FUTURE WORK

SZ is a state-of-the-art lossy compression framework. However, due to the complexity of the computations, performing SZ compression might take half a day per Terabyte of data. In this paper, we propose GhostSZ, an FPGA framework for accelerating lossy compression tasks used by SZ at line rate, and so transparently.

GhostSZ is currently a five-step framework that contains: offline configuration, linearization, best-fit curve-fitting, quantization, and Gzip. We tested the system with scientific data, and showed $9.5\times$-$80\times$ speedup per compute unit and $8.2\times$ per chip for one input data. GhostSZ has an average 25% better compression ratio and a similar peak signal-to-noise ratio (PSNR) on average. Note that the FPGA results are limited by our test set-up that depends on a PCI bus. Higher throughput is possible by instead using the MGTs. Combining use of MGTs with mapping to current generation FPGAs increases throughput to 60GB/s with conservative assumptions.

Since the SZ compressor emerged, there have been many optimization methods available for supporting users' diverse error-bound demands and for increasing fidelity and compression ratio. As a part of our future work, we plan to selectively support these optimizations in the GhostSZ framework.

REFERENCES

[1] Community Earth Simulation Model (CESM), https://www.earthsystemgrid.org/, 2019.

[2] Gzip compression, http://www.gzip.org, 2019.

[3] S. Di and F. Cappello, "Fast Error-Bounded Lossy HPC Data Compression with SZ," in *Proceedings Int Parallel and Distributed Processing Symposium*, 2016, pp. 730–739.

[4] Mellanox, "Mellanox Introduces Programmable Network Adapter Product Line with Application Acceleration Engine," http://ir.mellanox.com/releasedetail.cfm?ReleaseID=883814 accessed 11/9/2015, 2015.

[5] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *49th IEEE/ACM Int. Symp. Microarchitecture*, 2016, pp. 1–13.

[6] J. Sheng, C. Yang, A. Caulfield, M. Papamichael, and M. Herbordt, "HPC on FPGA Clouds: 3D FFTs and Implications for Molecular Dynamics," in *Proc. IEEE Conf. Field Prog. Logic and Applications*, 2017.

[7] J. Sheng, B. Humphries, H. Zhang, and M. Herbordt, "Design of 3D FFTs with FPGA Clusters," in *Proc. IEEE High Perf. Extreme Computing Conf.*, 2014.

[8] A. George, M. Herbordt, H. Lam, A. Lawande, J. Sheng, and C. Yang, "Novo-G#: A Community Resource for Exploring Large-Scale Reconfigurable Computing Through Direct and Programmable Interconnects," in *Proc. IEEE High Perf. Extreme Computing Conf.*, 2016.

[9] J. Sheng, Q. Xiong, C. Yang, and M. Herbordt, "Collective Communication on FPGA Clusters with Static Scheduling," vol. 44, no. 4, 2016.

[10] J. Sheng, C. Yang, and M. Herbordt, "High Performance Dynamic Communication on Reconfigurable Clusters," in *Proc. IEEE Conf. Field Prog. Logic and Applications*, 2018.

[11] J. Ritter and P. Molitor, "A Pipelined Architecture for Partitioned DWT Based Lossy Image Compression Using FPGAs," in *Proc Int Symposium on Field Programmable Gate Arrays*, 2001, pp. 201–206.

[12] T. W. Fry and S. A. Hauck, "Spiht image compression on fpgas," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 15, no. 9, pp. 1138–1147, Sep. 2005.

[13] L. Santos, J. F. Lpez, R. Sarmiento, and R. Vitulli, "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)*, June 2013, pp. 107–114.

[14] H. Sofikitis, K. Roumpou, A. Dollas, and N. Bourbakis, "An architecture for video compression based on the scan algorithm," in *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, April 2005, pp. 295–296.

[15] Y. Tang and N. Verma, "Energy-efficient pedestrian detection system: Exploiting statistical error compensation for lossy memory data compression," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 7, pp. 1301–1311, July 2018.

[16] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2674–2683, Dec 2014.

[17] N. Hübbe, A. Wegener, J. M. Kunkel, Y. Ling, and T. Ludwig, "Evaluating lossy compression on climate data," in *Supercomputing*, J. M. Kunkel, T. Ludwig, and H. W. Meuer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 343–356.

[18] S. Lakshminarasimhan, N. Shah, S. Ethier, S. Klasky, R. Latham, R. Ross, and N. F. Samatova, "Compressing the incompressible with isabela: In-situ reduction of spatio-temporal data," in *Proceedings of the 17th International Conference on Parallel Processing - Volume Part I*, ser. Euro-Par'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 366–379. [Online]. Available: http://dl.acm.org/citation.cfm?id=2033345.2033384

[19] S. Di and Y. Robert, "Toward an Optimal Online Checkpoint Solution under a Two-Level HPC Checkpoint Model," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 244–259, 2017.

[20] D. Tao, S. Di, Z. Chen, and F. Cappello, "In-Depth Exploration of Single-Snapshot Lossy Compression Techniques for N-Body Simulations," *IEEE Int Conference on Big Data*, no. 1, pp. 486–493, 2017.

[21] D. T. B, S. Di, Z. Chen, and F. Cappello, "Exploration of Pattern-Matching Techniques for Lossy Compression on Cosmology Simulation Data Sets," *ISC High Performance Workshops*, vol. 1, pp. 43–54, 2017.

[22] D. Tao, S. Di, Z. Chen, and F. Cappello, "Significantly Improving Lossy Compression for Scientific Data Sets Based on Multidimensional Prediction and Error-Controlled Quantization," *IEEE International Parallel and Distributed Processing Symposium*, 2017.

[23] X. Liang, S. Di, D. Tao, Z. Chen, and F. Cappello, "An Efficient Transformation Scheme for Lossy Data Compression with Point-wise Relative Error Bound," *Int Conference on Cluster Computing*, 2018.

[24] X. Liang, S. Di, D. Tao, S. Li, S. Li, H. Guo, Z. Chen, and F. Cappello, "Error-Controlled Lossy Compression Optimized for High Compression Ratios of Scientific Datasets," *IEEE BigData*, 2018.

[25] S. Di and F. Cappello, "Optimization of error-bounded lossy compression for hard-to-compress hpc data," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 1, pp. 129–143, Jan 2018.

[26] ——, "Optimization of Error-Bounded Lossy Compression for Hard-to-Compress HPC Data," *IEEE transactions on parallel and distributed systems*, vol. 29, no. 1, pp. 129–143, 2018.

[27] Intel Gzip Compression OpenCL Design Example, https://www.intel.com/content/www/us/en/programmable/support/support-resources/design-examples/design-software/opencl/gzip-compression.html, 2019.

[28] Xilinx, "Gzip," https://github.com/Xilinx/Applications/tree/master/GZip, 2019.

[29] ——, "Introduction to FPGA Design with Vivado High-Level Synthesis," https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf, 2013.

[30] J. Fowers, J. Kim, D. Burger, and S. Hauck, "A Scalable High-Bandwidth Architecture for Lossless Compression on FPGAs," in *International Symposium on Field-Programmable Custom Computing Machines*, May 2015, pp. 52–59.

[31] M. S. Abdelfattah, A. Hagiescu, and D. Singh, "Gzip on a chip: High performance lossless data compression on fpgas using opencl," in *Proceedings of the International Workshop on OpenCL 2013 & 2014*, ser. IWOCL '14. New York, NY, USA: ACM, 2014, pp. 4:1–4:9. [Online]. Available: http://doi.acm.org/10.1145/2664666.2664670

[32] Boston University, "Shared Computing Cluster (SCC)," 2019, https://www.bu.edu/tech/support/research/computing-resources/scc.

[33] Intel Quartus Prime Software Suite, https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html, 2019.

[34] L. Dagum and R. Menon, "Openmp: An industry-standard api for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan. 1998. [Online]. Available: https://doi.org/10.1109/99.660313

[35] NEK5000: a fast and scalable high-order solver for computational fluid dynamics, https://nek5000.mcs.anl.gov/, 2019.

[36] D. S. Aguado, R. Ahumada, A. Almeida, S. F. Anderson, B. H. Andrews, B. Anguiano, E. Aquino Ortiz, A. Aragon-Salamanca *et al.*, "The Fifteenth Data Release of the Sloan Digital Sky Surveys: First Release of MaNGA Derived Quantities, Data Visualization Tools and Stellar Library," *arXiv e-prints*, p. arXiv:1812.02759, Dec. 2018.

[37] W1zzard, "Seagate BarraCuda SSD 500 GB Review," 2019, https://www.techpowerup.com/reviews/Seagate/BarraCuda_SSD_500_GB/.

[38] IO-500: Virtual institute for IO, https://www.vi4io.org/std/io500/start#io-500, 2019.