Journal of Parallel and Distributed Computing 193 (2024) 104955

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc

Accelerating memory and I/O intensive HPC applications using hardware compression ☆

Saleh AlSaleh [a], Muhammad E.S. Elrabaa [a,*], Aiman El-Maleh [a], Khaled Daud [a], Ayman Hroub [b], Muhamed Mudawar [a], Thierry Tonellot [c]

[a] Computer Engineering Department & Interdisciplinary Research Center for Intelligent Secure Systems, King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia
[b] Department of Electrical and Computer Engineering, Birzeit University, Birzeit, Ramallah, Palestine
[c] EXPEC ARC (Advanced Research Center), Saudi ARAMCO, Dhahran, Saudi Arabia

ARTICLE INFO

Keywords:
High performance computing
Reconfigurable computing
FPGA accelerators
Data compression
Memory intensive applications
Hardware co-design

ABSTRACT

Recently, accelerator-based compression/decompression was proposed to hide the storage latency of high-performance computing (HPC) applications that generate/ingest large data that cannot fit a node's memory. In this work, such a scheme has been implemented using a novel FPGA-based lossy compression/decompression scheme that has very low-latency. The proposed scheme completely overlaps the movement of the application's data with its compute kernels on the CPU with minimal impact on these kernels. Experiments showed that it can yield performance levels on-par with utilizing memory-only storage buffers, even though data is actually stored on disk. Experiments also showed that compared to CPU- and GPU-based compression frameworks, it achieves better performance levels at a fraction of the power consumption.

1. Introduction

The use of GPUs and FPGAs in accelerating compute-intensive applications has seen significant growth in the past decade [1,2]. FPGAs offer more fine-grained control of the implementation allowing the trade off of power, speed, and accuracy to meet specific needs. In conventional FPGA acceleration, part or whole of the computation is offloaded to one or more FPGAs (as HW circuitry) attached to the host CPU node via a system bus i.e. PCIe, or a local area network. Even with the help of high-level synthesis (HLS) tools, it takes significant effort to develop, verify, and debug the FPGA circuitry. Though HLS tools expedite the design process, it can result in less optimum circuitry. In addition to High-speed, multiple lanes PCIe buses, the commodity HPC nodes where these FPGAs are attached to, usually have multiple sockets of high-end CPUs. For cost-effectiveness, these CPUs should be fully utilized and not just act as gateways for launching kernels on accelerators. Hence, this work represents a different strategy for accelerating HPC applications that run on such nodes. It allows the developers of such applications to take full advantage of high-end CPUs while circumventing the limitations of a finite-size RAM.

Traditionally, FPGA acceleration has worked best for low-latency streamed applications that did not require significant data storage on the FPGA or its board [3]. Other issues with conventional FPGA acceleration include; 1) synchronization delays between the FPGA and the host CPU, and 2) the need to transfer large data back and forth between the FPGA and the host CPU which requires the use of the FPGA board's memory resulting in extra latency. Ma et al. [4] proposed using on-FPGA direct-mapped caches with architectures that are optimized according to the data-access patterns to minimize communication overhead and memory conflicts with the CPU. That only helps if the FPGA kernel has high arithmetic intensity (i.e. data are re-used multiple times), not the case with many HPC applications.

Many HPC applications generate/consume large amount of intermediate floating point data that need to be moved back and forth several times across the node's storage hierarchy (HDDs, SSDs, NVMEs, and RAM). This causes significant delays to the application's compute kernels due to; 1) the use of valuable CPU cycles for system calls, 2) con-

☆ This work was supported by a Saudi Aramco contract No **6600011900**. Authors also acknowledge the support of King Fahd University of Petroleum & Minerals (KFUPM).

* Corresponding author.
E-mail addresses: salehs@kfupm.edu.sa (S. AlSaleh), elrabaa@kfupm.edu.sa (M.E.S. Elrabaa), aimane@kfupm.edu.sa (A. El-Maleh), khaled.daud@kfupm.edu.sa (K. Daud), aahroub@birzeit.edu (A. Hroub), mudawar@kfupm.edu.sa (M. Mudawar), thierrylaurent.tonellot@aramco.com (T. Tonellot).

https://doi.org/10.1016/j.jpdc.2024.104955
Received 22 May 2023; Received in revised form 26 May 2024; Accepted 7 July 2024

tention in main memory since all PCIe transactions go through the RAM (DMA), and 3) cache pollution due to the load/store instructions moving the data across the storage hierarchy. Many data reduction strategies for HPC applications have been proposed [5]. Data compression is one of these strategies, however, most works assume that data is generated/calculated and compressed once, but read and decompressed many times later i.e., applications that read the data do not incur the compression's performance penalty. This may apply to applications that capture large amount of data and store it for later use, but not a wide range of other HPC applications (e.g. simulators). Lossless compression/decompression of floating point numbers was proposed in [6] to accelerate HPC applications running on the FPGA by effectively increasing the on-board memory bandwidth to match the circuit's throughput. The compression was based on linear prediction and variable length encoding which yields modest compression ratios (even more for non-linear data). To predict non-linear relationships, other researchers proposed the use of hash functions instead [7].

This work proposes using FPGA-based data compression/decompression to accelerate HPC applications by providing them with seemingly infinite in-memory buffering of their intermediate data. Data is continuously sent from the in-memory buffer to the FPGA for compression before sending it to the node's storage. Compressed data are later sent to the FPGA for decompression before they are sent to the CPU's memory buffer for further processing. Depending on the compression efficiency (compression ratio and latency), this could reduce the storage latency significantly, overlap data storage (i.e., I/O) with the main kernel computation on the CPU, relief the CPU from performing compression/decompression, and reduce memory contention and cache pollution. The feasibility of this strategy was illustrated in this work on a typical HPC application that dynamically generates/consumes large amount of floating point data, namely, 3-D seismic imaging using Reverse Time Migration (**RTM**). RTM is comprised of three main compute kernels (phases): 1) Forward propagation (**FP**) of the 3-D wave field of the induced surface seismic waves using a suitable velocity model, 2) Backward propagation (**BP**)of the received surface waves with the same velocity model, and 3) superposition (correlation) of both using an imaging condition. The complete wave equation is used and the entire wave field, including multiple reflections, is used. The first two tasks involve solving finite difference equations (using an 8th order 25-point stencil) generating large amount of data that need to be correlated in the third task. Data generated during **FP** need to be stored so it can be correlated with data generated during **BP**.

Researchers have proposed conventional acceleration of RTM on FPGAs [1] and on GPUs [8] with considerable speedups over CPUs. Unfortunately, the majority of efforts that compare FPGA/GPU acceleration over CPUs, involve CPUs with limited RAM (e.g., 16 GB), and/or limited number of cores (e.g. 4). This tends to amplify the effect of conventional FPGA acceleration for non-streamed applications.

The proposed acceleration framework is implemented on a production-quality multi-threaded implementation of 3-D RTM imaging provided by Saudi Aramco (called *RTMLab*). RTMLab comprises many features that facilitate different run-time experimentation scenarios (e.g. synthetic data generation, wave field size, threading options, storage options, velocity models, etc.). One of these features allows users to select among three storage policies for the intermediate data generated during **FP**; 1) in-memory policy (i.e., the whole data is kept in the CPU's RAM), 2) disk policy (data is stored/retrieved in/from the disk, uncompressed), and 3) compressed-disk policy (data is compressed and then stored/retrieved and decompressed in/from the disk). Data is generated/stored/retrieved as frames (a frame is a 3-D wave field generated for each time step).

This work has the following contributions:

1. An unconventional acceleration framework for HPC applications based on hiding the disk storage latency of intermediate generated/ingested data via on-FPGA compression/decompression resulting in performance levels on-bar with using in-memory storage,
2. A novel low-latency hardware-friendly scheme for real-time compression/decompression of floating point numbers with point-wise fixed relative error. The FPGA implementation achieved high throughput, low latency and very low FPGA resource utilization,
3. Validation of the proposed acceleration scheme using an actual HPC application (RTM for Seismic imaging) showing performance levels similar to using in-memory storage.

Next, section 2 provides a review of FPGA and GPU-based data compression. An overview of the proposed framework is presented in section 3 along with the proposed fixed relative error (**FRE**) compression scheme. Section 4 provides a detailed description of the FPGA implementation. Details of the experimental setup used to evaluate the effectiveness of the proposed framework against CPU and GPU based compression are presented in section 5 followed by the experimental results in section 6. Comparison with other FPGA-based compression implementations in terms of resource utilization and throughput are also presented in section 6. Finally, conclusions are presented in section 7.

2. Review of FPGA/GPU compression

The majority of works on floating point data compression has focused on lossless compression when data accuracy is of utmost importance. Examples of lossless compression techniques include LZ4 [9] and GZIP [10] which are dictionary-based. Other lossless compression methods rely on using liner prediction and encoding using variable length codes [11,12] and [6]. FPZIP [12] relies on point-wise relative error bound to support lossy to lossless compression. The predicted and actual values are transformed to an integer representation during which the least significant bits can be optionally truncated if lossy compression is desired. To improve compression efficiency for non-linear data, other researchers proposed the use of hash functions to predict non-linear relationships [7].

Several researchers proposed different implementations of the Deflate compression algorithm (GZIP) on FPGAs [13] [14] and on GPUs [15]. Even though these implementations achieved various degrees of throughput performance, the compression efficiencies (ratio of original/compressed data sizes) were modest for floating point numbers (~2 to 2.5x).

Similarly, several lossless compression FPGA implementations based on variations on the dictionary-based Limpel-Ziv (LZ) scheme have been proposed. An early direct HW implementation of the LZ4 algorithm was presented in [16]. The compression efficiency was not reported, but the throughput was estimated to be ~220 MB/s. The same group later developed a better version that focused on parallelization of the search and match portion of the LZ4 algorithm, thus increasing the throughput ten folds [17]. The compression ratio was very low though (~1.5). A HW implementation that utilized 16 dictionaries in parallel to optimize and accelerate the search step of LZ4 was reported in [18]. Still, the highest achieved compression ratio was 2.69 and the maximum throughput was a modest 500 MB/s. Finally, a HW implementation of a modified LZ4 algorithm was presented in [19]. Data format was modified to improve the HW throughput. Two implementations with slight differences in the search and match module were realized. The second implementation used more FPGA DSP macros but less other resources and achieved a throughput of around 500 MB/s and a maximum compression ratio of 2.05.

As can be seen from the above review of lossless compression schemes, they barely achieve more than 2x compression, which is not sufficient to provide the required reduction in storage latency. Lossy compression techniques of floating-point data (e.g., ISABELA [20], Squeeze or SZ1.4 [21]) can achieve higher compression ratios. ISABELA, however, needs to sort the data before using B-spline interpolation to compress it, something that reduces its suitability for direct GPU/FPGA implementation. It also has a lower compression ratio than SZ1.4 as it has to store indices for each data point. Squeeze (SZ) uses best-fit curve

prediction based on previously predicted neighboring points. The relative error, however, is taken as a fraction of the whole data range. This means for data with very large dynamic range, small data values will be set to zero and completely lost. Hence, in this case the only way to get a certain point-wise accuracy without prior knowledge of the range, is to assume it maximum and set the relative error to a small enough value. When that is done such that the point-wise relative error is bound to something like 1 ∼ 2% the compression ratio becomes ∼ 5 ∼ 6x. In lossy compression mode, FPZIP [12] adds simple truncation to increase the compression ratio at the required precision. As with SZ, for 3-D data, it requires the surrounding points to make the prediction which requires non-uniform memory access and in the case of HW implementation, significant on-FPGA buffers to hold several planes of 3-D data. FPZIP, however, uses shorter stencil for prediction which would result in relatively smaller buffer and better performance at similar compression ratios.

GPU implementations of lossy compression based on both SZ and ZFP were evaluated in [22] and [23]. ZFP only supports Fixed Rate Mode for GPU CUDA, which does not allow the relative or absolute error to be specified and limited. ZFP is a lossy block compression scheme that can achieve higher compression ratios than SZ, especially for data with many blocks of zeros (each of such blocks is replaced by 1-bit). These implementations achieved very high compression/decompression throughput by assuming that the original data is already in the GPU's memory, i.e., the main application runs on the GPU (so no CPU-GPU communication latency). Moreover, the data is divided into chunks that match the number and size of warps, thus fully utilizing the GPU's resources. They also reported high compression ratios by using a fixed absolute error. This eliminated the need for any pre-processing of the data to determine the error bound that would guarantee a maximum fixed relative error (i.e., a maximum relative error for any point in the data).

GhostSZ [24] and waveSZ [25] are both FPGA implementations of modified versions of SZ1.4. GhostSZ uses more FPGA resources and achieves significantly lower compression ratios than waveSZ. To improve resource utilization and throughput, waveSZ pre-processes data on the host CPU to achieve ∼1 GB/s/core throughput. This would slow down the HPC application generating the data. A HW-SW codesign of a modified ZFP compression algorithm was reported in [26]. The group testing part of the original ZFP algorithm was replaced by a new algorithm that increased the number of compressed bits to 16. Again, the implementation utilized the HARP platform (tightly coupled CPU-FPGA in the same package) to increase the throughput.

Finally, the authors of this article have explored high-throughput FPGA implementation of a modified version of ZFP, DE-ZFP [27]. DE-ZFP utilized dictionary-based encoding instead of embedded encoding to maximize performance. However, the compression ratios and throughput of these lossy compression techniques were for fixed absolute errors (ZFP-V used 1E-3, DE-ZFP reported results for 1E-3 and 1E-6). For applications that generate/use data with large dynamic range such as RTM Seismic imaging, this is not acceptable as many image features would be lost. Furthermore, albeit achieving good throughput, all FPGA implementations of ZFP suffered from high latency (∼10 s of ns [27]), making them unsuitable for our objective of accelerating HPC applications by reducing storage latency. In summary, the following can be observed:

1. The best lossy compression algorithms (i.e., SZ and ZFP) achieve ∼5x compression ratios when true point-wise relative error bounds are used,
2. Most GPU/FPGA implementations either assume the data is already in the accelerator's RAM, or there is no application running on the host's CPU. Thus, synchronization issues (and thus the impact) with host applications are not considered. Furthermore, point-wise (e.g. SZ) and block-based (e.g. ZFP) compression require neighboring data. For 3-D data, this means irregular data access pattern in the host RAM, thus reducing memory bandwidth and possibly causing cache pollution which would in turn negatively impact the host's application performance.

In this work, a simple lossy compression scheme based on constant relative error has been developed. It was developed with the specific objective of providing extremely low-latency compression/decompression of in-memory data before storing it on disk or reading it back to the RAM. This would then be utilized to accelerate HPC applications running on the host CPU(s) by reducing their I/O latency to that of using in-memory storage. It does not require any host pre-processing, nor does it need to store the data on the FPGA, and yet achieves compression ratios on-par with the best in-class lossy algorithms at the same accuracy levels. Furthermore, it is a point-wise scheme that only require the previous data point. Hence by reading the data from the host RAM, and sending it to the FPGA, in the same order it is stored in, memory bandwidth is maximized and the potential of cache pollution is also minimized. All these attributes, reduce the negative impacts on the host application to a minimum.

3. Overview of the proposed acceleration framework

3.1. Basic strategy

The main strategy for accelerating HPC applications that generate and ingest large intermediate data is to use an accelerator-based Compression Engine (CE) to compress/decompress intermediate data before storing it in the node's disk or when reading the data back. When intermediate data is streamed to the accelerator in a full asynchronous fashion, the accelerator-based compression/decompression can proceed in parallel with the computation on the host where the host threads only store/load data in/from the memory. Provided that the compression/decompression latency is less than the compute threads' unit of execution time (e.g. a loop that generates/consumes data in chunks), the storage latency is completely removed from the execution time and performance levels similar to using an in-memory data buffer can be attained.

In this work, two Compression Engines were evaluated; an NVIDIA A100 GPU and an Intel Arria 10 GX FPGA. Both platforms allow overlapping of host communications with computations. High-end FPGAs usually provide a combination of hard and soft PCIe interface macros for communication with the host. Moreover, some form of a SW run-time environment is used on the CPU side in the form of APIs (Application Program Interface) for communication between the host program and the on-FPGA accelerator. These run-time environments do affect the main application's execution time. Thus, in this work, we used the open source communication framework *Reusable Interface Framework for FPGA Accelerators* (RIFFA) [28] for the CPU-FPGA communication. It provides simple data transfer APIs on the CPU side (statically compiled library with no run-time environment) and a FIFO-based streaming interface on the FPGA side on top of the hard PCIe macros available on high-end FPGAs. RIFFA provides up to twelve independent fully-duplex *channels* between software threads on the CPU and the HW threads on the FPGA.

To minimize the impact on the HPC application, the following was done:

1. New threads, dubbed *servant* threads, are added to the original application's code to run in parallel with the worker threads and handle the tasks of streaming data to/from the Compression Engine (CE), and storing/retrieving data to/from the node's disk. They utilize the typical barrier found in data-parallel applications at the end of each iteration of the main kernel's that synchronizes the worker threads. Data is also read/written from the host's RAM in the same order as they are stored in to maximize memory bandwidth. Hence, if the CE's compression/decompression latency are smaller than the worker threads' loop execution time, they latter won't have to wait for the servant threads at the barrier and can just read/write data

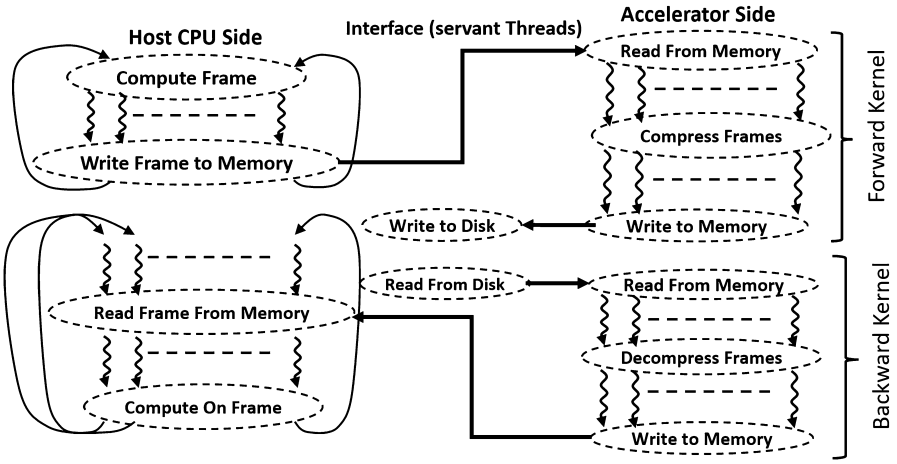

Fig. 1. Illustration of the proposed HPC application acceleration framework.

from/to the memory buffer all the time, completely eliminating the I/O overhead. For applications with higher throughput, more FPGA engines can be added,

2. Servant threads and the Compression/Decompression Engines were all designed to operate asynchronously (independent and in parallel) and using separate logical channels. Thus, multiple FPGA compression/decompression *channels*, with separate dedicated servant threads, can be implemented to serve several compute kernels simultaneously,
3. To reduce memory latency, servant threads are only spawned after the first chunk (we call it a frame) of data is generated or read from the disk by the worker threads. This ensures proper setup of the data generated/ingested by the different worker threads in the RAM attached to the socket where they run on (as data is placed in the socket's RAM of the thread that uses it for the first time). This reduces the relatively slower data transfers between the sockets (through the quick path interconnect, QPI),
4. The servant threads are also pinned to a dedicated core on the CPU socket that controls the PCIe lanes that host the GPU or FPGA board. Again, this reduces the communication latency to/from the GPU/FPGA by not having to go through the QPI.

Hence, every time worker threads generate a frame, it is copied to one of two memory buffers (every iteration, we switch between the two buffer pointers, i.e. the ping- strategy) for the servant threads to take care of compressing the data and storing it on the disk. Similarly, when the worker threads need a frame from the disk, another set of servant threads are spawned to get the frame from the disk, decompress it, and place it in one of two memory buffers for the worker threads to process it (while the next frame is being retrieved into the other memory buffer). Thus, the compute kernels on the CPU are spared from performing compression/decompression on intermediate data that does not fit in the node's RAM. Also, the impact of the servant threads on the performance of the kernel's worker threads is minimized and they would be able to maintain performance levels very close to that of using in-memory data storage.

Fig. 1 shows the proposed acceleration framework as applied to the *RTMLab*. The main application's kernels use many worker threads to compute/correlate the frames with standard data parallelism and a barrier synchronization at the end of each frame generation. The Compression Engine (CE) can be CPU, GPU, or FPGA based compressor/decompressor.

During the **FP** phase, frames generated by the worker threads are sent to the Compression Engine, where they are compressed, sent back to the RAM via DMA (Direct Memory Access), then written to the disk. After the first frame is generated, the master thread spawns both sending and receiving servant threads along with the normal worker threads. Worker threads would be generating the next frame (storing it in the 2nd memory buffer) while the servant threads handle the previous frame. All threads are synchronized at the barrier before the two buffers' pointers are swapped so the next frame is generated in the first frame's place. Thus, from this point on, no new servant threads are spawned before the previous ones are done with the previous frame. In **FP** phase, the servant threads send/receive frames/compressed frames to/from the Compression Engine (CPU, GPU, FPGA). No explicit synchronization between the two servant threads is required in performing these tasks. The receiving servant thread which stores the compressed frames in a disk file, also stores the size of each compressed frame so they can be individually retrieved later on.

The **BP** kernel ingests the frames that were generated in **FP** in reverse time order, last to first (hence the name reverse time migration). So now, the sending servant thread retrieves the compressed frames from the disk and sends them (one by one) to the Compression Engine's decompression channel. The receiving servant thread receives the decompressed frame from the Compression Engine and stores it in the BP memory frame buffer (the BP kernel only uses the last two frames at a time, hence these are just stored in the memory). These two servant threads are spawned first (before the **BP**'s worker threads) when the **FP** kernel terminates, to prepare the last frame generated by the **FP** kernel before the **BP** kernel starts. From that point on, the servant threads are synchronized at the same barrier of the **BP** compute threads. Hence, previously spawned servant threads have to finish before new ones are spawned to prepare the next frame and so on.

3.2. Efficient FPGA fixed relative error compression/decompression

Examining HPC applications that deal with temporal modeling (i.e. transient modeling) of multi-dimensional spatial data (e.g. wave fields as in RTM), the following is observed:

1. Produce fairly smooth data (spatially and temporally) with large dynamic range. I.e., at any time step, most points' values are identical/close to the surrounding points (spatial smoothness). Thus, encoding the difference between adjacent points can reduce the data size while retaining high accuracy.
2. Regular pattern of disk reads/writes (of frames of spatial data/time step). So even with variable length encoding, data is naturally delimited because of the fixed frame size.

As such, a simple Fixed Relative Error (FRE) compression scheme for single-precision floating point data was developed. Data are compressed to variable-length code words (from 2 to 27-bits) as follows (Table 1):

Table 1
Summary of proposed relative error lossy compression scheme.

Float Type	Condition	Encoding Scheme	No. of Bits
Denormalized	exponent = 0	'010' + sign bit + 23-bits mantissa	27 bits
Normalized	Equal Floats: sign bit, 1st 6 bits of mantissa and the exponent of current and previous points are identical	'00'	2 bits
	Exponents absolute difference ≤ 9	'1' + sign bit + 6-bit MSBs of mantissa + (1-9 bits) encoding of exponents absolute difference including the sign of the difference (if it is > 0) Exponent absolute difference encoding: '0' if absolute difference is 0 '10' if absolute difference is 1 '110' if absolute difference is 2 '1110' if absolute difference is 3 '11110' if absolute difference is 4 '111110' if absolute difference is 5 '11111100' – '11111111' if absolute difference is 6-9	9 to 17 bits
	Exponent absolute difference > 9	'011' + sign bit + 8-bit exponent + 6-bit MSBs of mantissa	18 bits

1. For denormalized floats (zero exponent), the exponent is removed, the mantissa and sign bit are kept as is, and '010' is added. This is the longest code word (27 bits) but it preserves denormalized points as they may have some significance for the application.
2. For normalized floats: Depending on the required point-wise fixed relative error (between the decompressed value and its original value) bound, the mantissa is simply truncated to the appropriate number of the most significant bits (e.g., 6-bits for a 2^{-6} or ∼1.5% relative error). The exponents are encoded using variable length encoding (1 to 9 bits) as ***a difference*** relative to the previous point's actual exponent (not the encoded one). This is for exponent absolute difference ≤9. If the exponents absolute difference is ≥ 10, the whole 8-bit exponent is kept as is.
3. In case of equal successive floats (same sign, equal truncated mantissa, and exponents), a 2-bit code '00' is used to encode them.

The exponent code words in Table 1 were devised based on statistical analysis of many seismic shots. As expected, the probability of each exponent absolute difference was found to be inversely proportional to the absolute difference. If FRE was to be applied to an application with a peculiar data statistics (e.g. some larger exponent differences have higher probabilities than some smaller differences), the code words could be interchanged between these different values to maintain the higher compression efficiencies. For lower relative error, smaller number of mantissa bits can be truncated (exponent encoding remains unchanged) with no impact on performance at all and only minimal loss in compression efficiency. For example, truncating after 7 mantissa-bits (instead of 6) for seismic data, reduces the average compression ratio by ∼7%. It should be noted that most compression works set error bounds relative to the overall data range (not to the points' original values) which results in setting small-valued points to zero (and possible loss of fidelity).

The proposed FRE scheme resulted in an average compression ratio of ∼ 5X, which is on-par with the much more sophisticated compression techniques (e.g., SZ) at the same point-wise relative error bound but with about two orders of magnitude less resource utilization when implemented in HW. This is because this encoding technique is very simple to implement as a data-flow model, requires no data storage, and can be pipelined to yield high throughput implementation with minimal resources and latency.

4. FPGA implementation of FRE compression/decompression (FPGA-FRE)

The proposed scheme was implemented on an Intel's Arria 10 GX development board that communicates with the host CPU via DMA using 4-lane PCIe Gen-3 and the RIFFA framework [28]. Fig. 2 shows the top level FPGA implementation including the PCIe IP core, RIFFA wrapper, and the logical compression/decompression channels. In this work only two channels were implemented (one for compression and one for decompression), each with a data input width of 128-bits (4 floats). A major challenge for designing the compression/decompression channels was to maximize the utilization of the PCIe data bandwidth (4 GB/s). That meant data had to be streamed in and out of the FPGA as 128-bit flits at 250 MHz frequency. Hence, after compression, the variable-sized compressed data had to be packed into 128-bit flits before being streamed back to the CPU. Similarly, compressed data received for decompression as 128-bit flits, had to be unpacked into variable-sized words (2 to 27 bits) before being decompressed into 32-bit floats, then packed again into 128-bit flits and sent back to the CPU. The developed FPGA designs accomplished these tasks under 10 ns of total on-FPGA latency. The achieved throughput and latency were more than enough to implement the acceleration scheme by almost eliminating the I/O latency of the worker threads on the host CPU.

4.1. The compression channel

Fig. 3 shows the block diagram of the compression channel. Floating point data are received into an input FIFO which also provides the back pressure, when needed, for the PCIe link. The Compression Engine is a 2-stage pipeline that receives a single floating-point number every cycle, encodes it to a variable length code, then compact these variable length words into 128-bit flits. The compaction circuit, dubbed Bit Compactor, is a special buffer that allows pushing/popping variable length data words (the width of the pushed/popped word is specified during the write/read operations). *Bit Compactor1* first packs the compressed data words (with lengths from 2 to 27 bits) into 32-bit words, which are then packed into 128-bit flits by *Bit Compactor2* for the PCIe output FIFO interface. The output FIFO then transfers these flits to the host whenever the host is ready to receive them. Each chunk of compressed data is appended with a 32-bit header that specifies the number of compressed floats in the chunk. This crucial for the subsequent decompression.

Three finite state machines (FSMs) control the different activities of the compression channel (Downstream communication including back pressure, Compression Engine, and Upstream communication). The Upstream FSM controls popping the 128-bit flits from Bit Compactor2, storing them into the output FIFO, and then sending them to the host. Individual FSMs and control signals are not shown in Fig. 3 for the sake of readability.

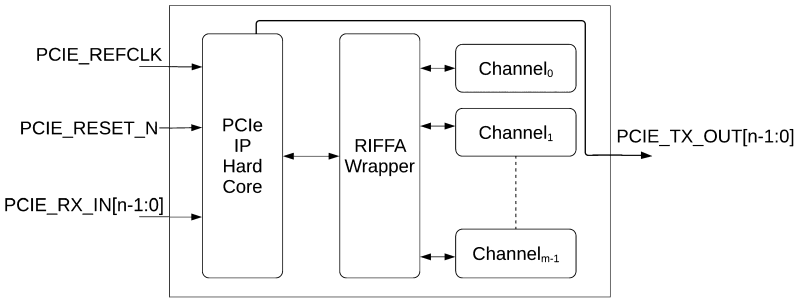

Fig. 2. Top-level diagram of the FPGA-FRE circuitry.

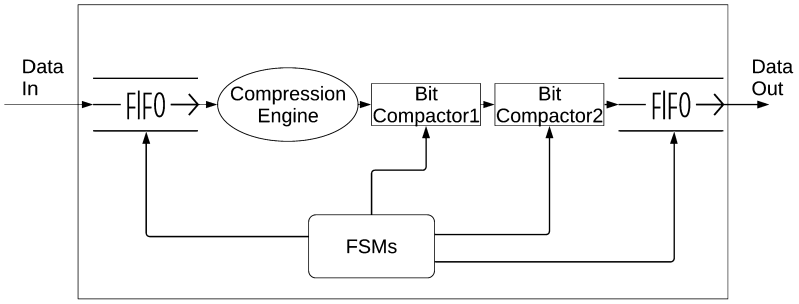

Fig. 3. Block diagram of the compression channel.

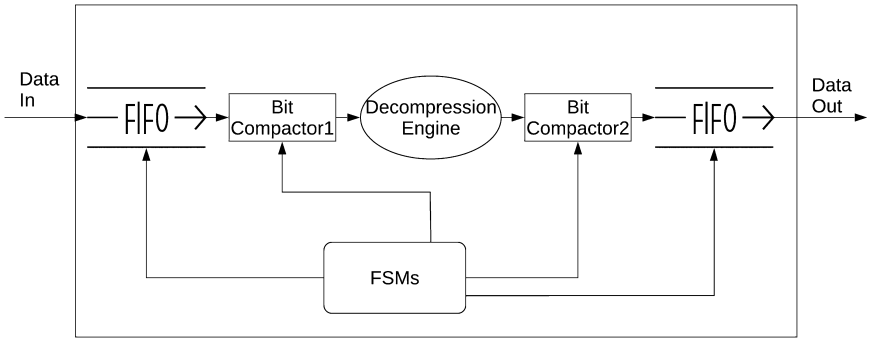

Fig. 4. Block diagram of the decompression channel.

4.2. The decompression channel

Fig. 4 shows a block diagram of the decompression channel. The input FIFO feeds the Bit Compactor1 module with 32-bit values which are read by the decompression engine. An FSM controls how many bits to read from Bit Compactor1 (2 to 32 bits) and reconstructs the decompressed float. This FSM also keeps track of the number of decompressed floats (based on the received 32-bit header). On average, each reconstructed float requires two reads from Bit Compactor1. This makes the decompression channel's throughput about half of the compression channel (as demonstrated in the experimental results). Bit Compactor2 receives the 32-bit floats from the decompression engine, compacts them into 128-bit flit, and pushes them into the output FIFO. Two more FSMs control receiving the downstream data and pushing it into Bit Compactor1, and popping the upstream data from Bit Compactor2 into the output FIFO and sending it to the host CPU, respectively.

5. Experimental setup

5.1. Overview of RTMLab

RTMlab, a seismic imaging simulator, starts simulations by forward propagating the injected surface waves using one of the several available velocity models and using a 3-D 8th order 25-point stencil. In this phase it generates a large number of 3-D wave fields (frames), one per time step, that constitute a single snapshot that needs to be stored. In the subsequent **BP** phase, it back propagates the received surface wave back in time. During this phase, it also reads the stored forward propagated frames (in reverse order), and correlate them with the **BP** generated frames to produce the 3-D sub-surface image. *RTMlab* allows controlling the size of the generated 3-D wave fields (hence the size of each frame and the snapshot). Table 2 summarizes the sizes of the data used throughout this work. *RTMlab* also has several options (called policies) for storing the intermediate snapshots generated during **FP**. To implement the proposed framework, we have added two more policies as listed below: 1) the *Memory* policy, where the entire uncompressed snapshots are stored in the system's memory, 2) the *Disk* policy, where snapshots are stored uncompressed in the node's local disk, 3) the *Compressed Disk (SZ)* policy, where snapshots are first compressed by the CPU using SZ 1.4 [29] with 2^{-6} point-wise relative error bound before being stored in the disk, 4) the *Compressed Disk (SW-FRE)* policy, which is similar to (3) except that a software version of the proposed FRE compression running on the CPU is used to compress the snapshots before storing them on the disk (with 2^{-6} relative error bound), and 5) the *FPGA-FRE)* policy, where the FPGA implementation of FRE is used to compress the snapshots before writing them to the disk.

Table 3 summarizes the four main sequential tasks in *RTMlab*. These include the main compute tasks (the FP and BP kernels) which are multi-

Table 2
Data sizes for the *RTMlab* models.

Model	Initial Data Size	Snapshots Size (Uncompressed)	Output Image Size
Regular	355.5 MB	97.6 GB	567.9 MB
Large1	430.7 MB	130.6 GB	759.8 MB
Large2	628.3 MB	224.9 GB	1.3 GB
Large3	1.3 GB	655.6 GB	3.7 GB

Table 3
RTMlab Main Computation Tasks.

Task	Definition
Forward Propagation	**FP** is the 1st Stencil-based Multi-threaded Computation Kernel
Backward Propagation	**BP** is the 2nd Stencil-based Multi-threaded Computation Kernel
I/O Read	Includes all the single-threaded mini tasks that are interleaved with the **BP** kernel (reading frames from memory or disk and decompression of compressed frames, for policies with compression)
I/O Write	Includes all the single-threaded mini tasks that are interleaved with the **FP** kernel (writing frames to memory or disk and frames compression for policies with compression)

Table 4
Machines Specifications.

Component	Machine 1	Machine 2
CPU	Dual Sockets Intel Xeon E5-2620 @ 2.40 GHz (12 Physical Cores, 24 Logical Cores/Threads), 15 MB L3 Cache	Dual Sockets Intel Xeon Gold 6136 @ 3.00 GHz (24 Physical Cores, 48 Logical Cores/Threads), 24.75 MB L3 Cache
RAM	512 GB DDR4 (16 x 32 GB DIMMs), 512 GB SWAP size	128 GB DDR4 (8 x 16 GB DIMMs), 128 GB SWAP Size
HDD	4x 2 TB in RAID 10 (RAID0+1) Configuration	256 GB NVME SSD (PCIe Gen3.0 x4) for OS and 2x2TB in RAID0 for data LTS
OS (Kernel)	Ubuntu 16.04 LTS (4.4.0-145)	Ubuntu 18.04 (5.3.0-40)

threaded, and other mini tasks such as moving the data to/from the main memory and the disk, compression, and decompression (for the policies that involve compression). The mini tasks were lumped into the I/O Read and Write tasks because their times are overlapped by the OS calls and cannot be separated. The total execution times, however, were measured from start to end of execution (excluding the initial setup) and ***not*** as the sum of different tasks' times.

5.2. Host nodes' configurations

Two commodity machines with the specifications listed in Table 4 were used to run the experiments. Machine 1 has 12 physical cores and 512 GB of RAM, while machine 2 has 24 cores and 128 GB of RAM. These differences allow for interesting experimentation with the proposed framework. Machine 2 has Advanced Vector Instructions (AVX-512) but *RTMlab* does not utilize any of them, so they do not affect the results. For the FPGA-FRE policy, an Intel Arria 10 GX FPGA (20 nm technology) development board was used. GPU based policies were run using NVIDIA A100 40 GB GPU (7 nm technology).

6. Experimental results

Many experiments were performed to verify if the proposed scheme makes (*RTMlab*) achieve performance levels similar to using an in-memory data storage policy while the data is actually stored on the disk. This included evaluating *RTMlab*'s baseline performance under different storage policies and with different compression engines. The **FP** and **BP** compute kernels were implemented with one worker thread per physical core as experiments showed significant performance degradation when more threads are used per core. Hence, the maximum number of threads was limited to the number of physical cores. For FPGA-FRE, one core was dedicated for all the servant threads. Experiments were repeated enough number of times for the variations in the reported average times to be less than 1%.

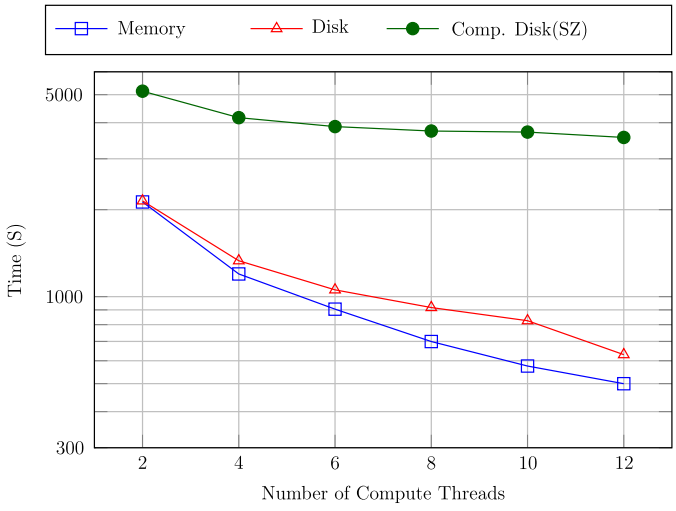

Fig. 5. Total execution time of *RTMLab* on machine 1 using the regular model for the three baseline SW-based storage policies.

6.1. Baseline performance of RTMlab

Figs. 5 and 6 show the total execution time of *RTMlab* versus the number of worker threads on machines 1 and 2, respectively. Three data storage policies were used; Memory (Uncompressed), Disk (Uncompressed), and Compressed Disk (CPU Based SZ1.4 with a relative error of 2^{-6}). The regular data size (~98 GB) was used in order to fit in both machines' RAMs (to enable the memory policy). Execution times of compression and decompression (both are single-threaded) were lumped into the I/O write and read times, respectively (Table 3). Figs. 7 and 8 show break down of the execution times of the four main computing tasks of *RTMlab*. These times are not overlapped, and were accumulated

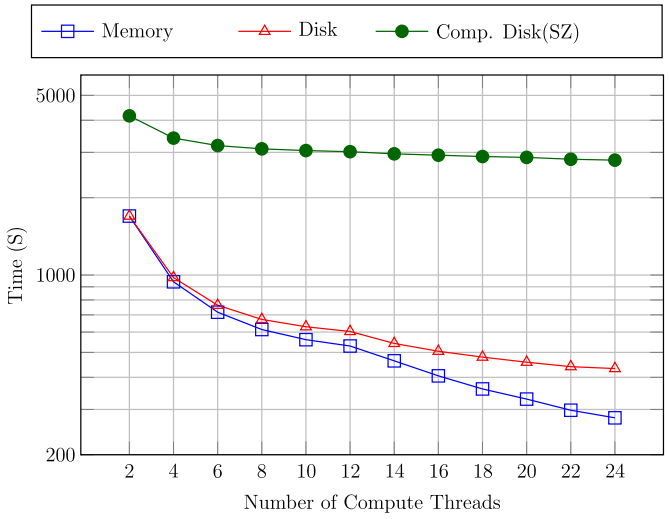

Fig. 6. Total execution time of *RTMLab* for the three baseline (SW) policies (regular model, machine 2).

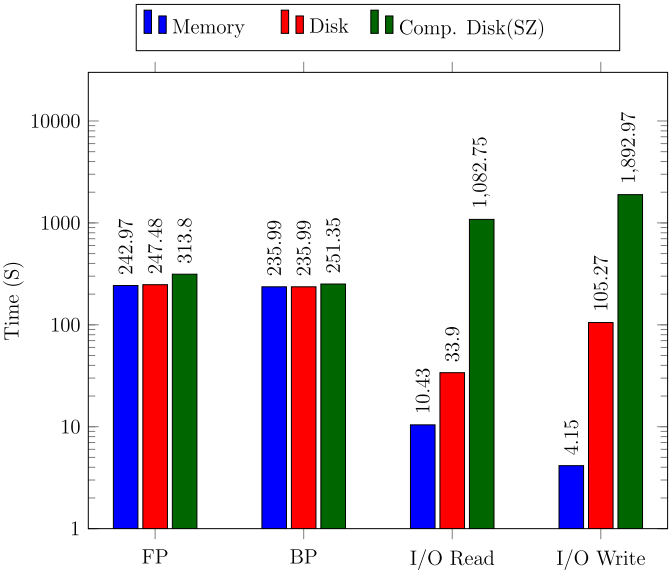

Fig. 7. Breakdown of *RTMlab* components' execution times at 12 threads for the three baseline (SW) policies (regular model, machine 1).

over the entire snapshots (the **FP** and **BP** kernels were each interleaved with the I/O Read/Write and Compression/Decompression tasks). The following can be observed from these results:

1. Unlike the Memory and Disk policies, the performance of the Compressed Disk (SZ1.4) does not scale up beyond 6 threads. This is because compression/decompression overhead represented ∼**89**% of the total execution time,
2. At low number of threads, both Disk and Memory policies yield similar performance due to the OS memory buffering of disk I/Os which hides their latency. At higher number of threads, however, disk I/O latency becomes more prominent as the delay of the parallel parts of the application becomes smaller, thus making the Memory policy ∼**70**% faster at 24 threads,
3. A noticeable kink in the total execution time graphs occurs at the point where the number of worker threads exceeds the physical core count of one socket. This is because as threads spread into the 2nd socket, memory access becomes non-uniform (threads access the other socket's RAM through QPI). This effect is more prominent for the disk policies (where frames need to be moved between the RAM and the disk),
4. I/O Write/Read times are much higher for the Compressed Disk policy as they include compression/decompression times,
5. Since both compute kernels perform similar stencil-based computations, their execution times for the Memory and Disk policies are basically the same. For the Compressed Disk policy, however, their execution times are affected differently. The **BP** kernel's performance (which is interleaved with SZ decompression) is hardly affected. However, **FB** kernel's execution time (interleaved with SZ compression)increased significantly; **29**% and **23.7**% for machines 1 and 2, respectively. The performance penalty is more for machine 1 due to its smaller RAM size,
6. At 12 threads each, machine 1 performance was slightly better (∼**6**% better) than Machine 2 due to its larger RAM. Even though Machine 2 had larger cache and higher clock frequency, these did not make a difference as the basic working set fits in both machines' caches anyway and both machines have similar RAM bandwidth.

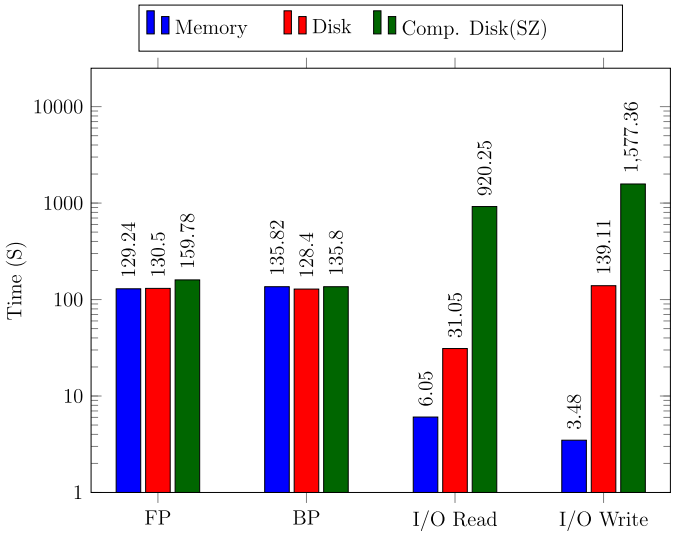

Fig. 8. Breakdown of *RTMlab* Components' execution times at 24 threads for the three baseline (SW) policies (regular model, machine 2).

6.2. Evaluating FRE impact on RTMlab's accuracy

Even though the relative error of **FRE** is bound to ∼1.5%, its impact on *RTMlab*'s accuracy had to be evaluated as decompressed frames are correlated with the generated frames in the back propagation phase. As such, the final image (after correlation with the last of 176 frames in back propagation) with and without **FRE** were compared for the regular data size (148,877,000 data points) for the medium model. Two measures were used to quantify the impact of FRE on the final image, relative error and Signal to Noise Ratio (SNR). SNR is often used to quantify the impact of reduced precision on seismic images (as added noise). SNR is evaluated from the ratio of the variance of the original image to that of the difference between the original and the one obtained with **FRE**: $SNR = 20 \times LOG\left(\frac{Vrms(ORG)}{Vrms(ORG-FRE)}\right)$. The SNR value was found to be **89.96 db** which is not only excellent but well above any other source of noise. Fig. 9 shows a histogram of the relative error. Over 90% of points in the final image had less than 2% relative errors. Examining the points that had relative error higher than 2% showed that they were all background noise to start with.

6.3. RTMlab performance under the proposed framework

RTMLab was run with the following additional policies; A SW-based Compressed Disk using **FRE** compression (**SW-FRE**), FPGA-based FRE

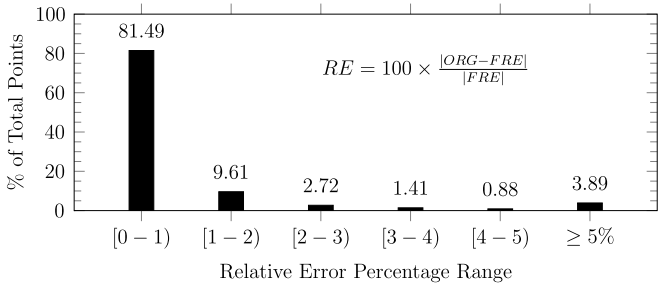

Fig. 9. Relative Error (RE) Histogram.

Table 5
Compression Engines (CE) Performance.

Compression Engine	Comp. Ratio	Comp. Thr. (GB/s)	Decomp. Thr. (GB/s)	Power (W)
FPGA-FRE	4.67	2.40	0.46	22
ZFP-GPU	5.16	0.63	1.11	63
cuSZ	10.91	0.28	0.24	82
nvcomp-gdeflate	1.78	14.50	96.63	61
nvcomp-lz4	1.69	17.84	123.06	67

compression (**FPGA-FRE**), and several existing GPU-based compression frameworks: 1) ZFP-GPU [30] Fixed Rate compression (it has no error bound, just fixed average compression rate which was set to ∼5.33x), 2) cuSZ [22] with 10^{-6} absolute error compression, 3) NVIDIA nvcomp GDeflate and NVIDIA nvcomp LZ4 lossless compressions [31]. Figs. 10 and 11 show the average total execution time versus the number of threads. The GPU based policies were only run on Machine 1 as the GPU board did not physically fit in Machine 2. For the FPGA-FRE and GPU based policies, the performance was evaluated using only two values of worker threads for each machine (5 and 11 on Machine 1, 11 and 23 on Machine 2). The servant threads were pinned to the socket with PCIe lanes to the FPGA/GPU boards. Two experiments were performed; one with worker threads on the remaining cores of that socket only, and another with additional worker threads on all cores of the other socket too (i.e., with only one fully utilized socket, and with both sockets fully utilized).

Figs. 12 and 13 show the execution times of the 4 computing tasks of *RTMlab* with the different compression engines (using maximum number of threads of each machine). Again, for the FPGA-FRE and GPU-based policies, the number of worker threads is actually less by 1 than the other policies since one core was dedicated for the servant threads.

Table 5 shows compression ratio, compression throughput, decompression throughput, and power consumption of the Compression Engines tested within *RTMlab* using the Medium Model on Machine 1. cuSZ consumed the highest power compared to all the others, while FPGA-FRE consumed least (about one-fourth of cuSZ's) with a very good compression ratio and compression throughput. ZFP-GPU only supports Fixed Rate mode where the absolute and relative errors cannot be controlled. nvomp-gdeflate and nvcomp-lz4 achieved highest throughput but being lossless, their compression ratios were modest.

The following is observed from these Figures:

- Again, SZ (with relative error mode) is the slowest. Its I/O Read and I/O Write times (include decompression and compression) represent approximately **83**% and **89**% of the total execution times on machines 1 and 2, respectively.
- SW-based **FRE** disk compression is considerably faster than SZ since it performs compression in one path over the data. Its I/O Read and I/O Write times are about **61**% and **71**% of the total execution times on machines 1 and 2, respectively. Thus, using SW-based FRE instead of SZ, reduced the compression time (and hence the I/O times) by ∼**3X** on both machines. Still, SW-FRE is significantly slower than the Disk policy which shows that SW-compression is **not** a viable practical option at this data size,
- The FPGA-based **FRE** performed the best among all policies that store data in the disk. Its I/O Read and Write times were less than SZ and SW-FRE times by up to **104X** and **26X**, respectively. Moreover, its I/O Read and I/O Write take significantly less portion of the total execution time (**2-3**% and **23**% on machine 1 and 2, respectively), thus approaching the I/O times of the Memory policy. It had significantly less impact on the main compute kernels' times (the **FP** and **BP** times were virtually unchanged),
- ZFP-GPU was run with fixed-rate of 6 which gave the closest compression ratio to the other Compression Engines (but higher error bound). It performed very close to the FPGA-FRE (within 7% difference),
- cuSZ had to be forked as a process (not as a library call from within the servant thread) due to its memory leaks [32]. This solved the memory leak problem but resulted in significant performance losses.
- nvcomp-gdeflate and nvcomp-lz4 had similar performance and outperformed the CPU based solutions (i.e. SZ, SW-FRE) and cuSZ. They

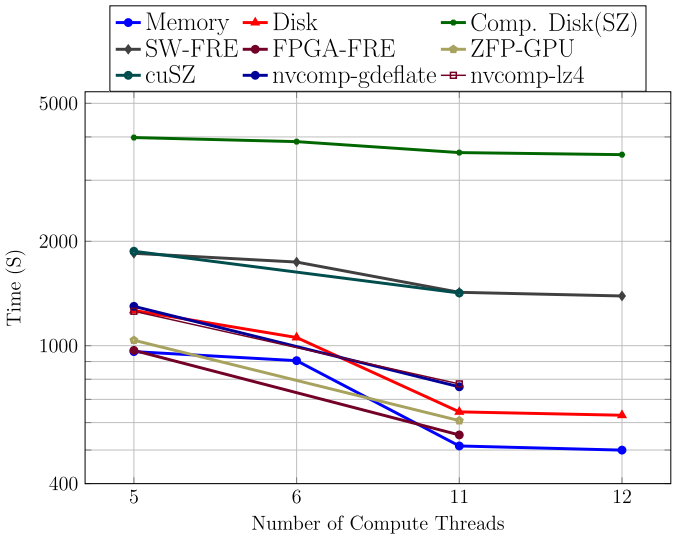

Fig. 10. Total execution time for all policies under the proposed framework using the regular model (Machine 1).

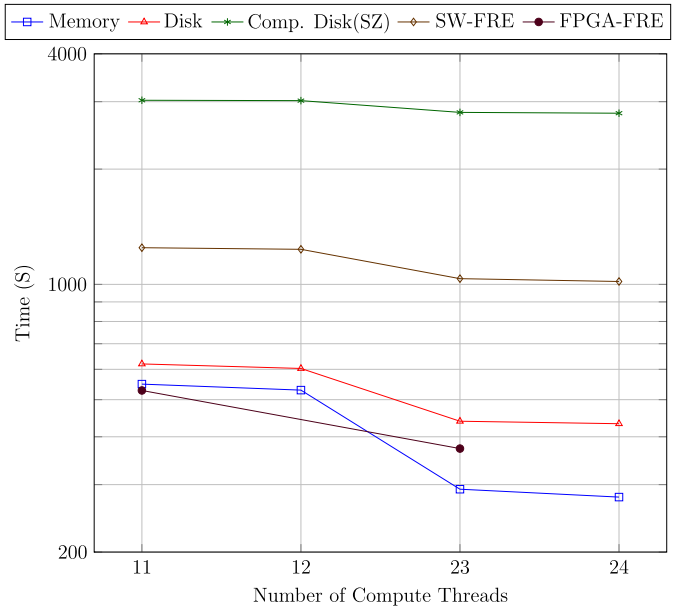

Fig. 11. Total execution time for all policies under the proposed framework using the regular model (Machine 2).

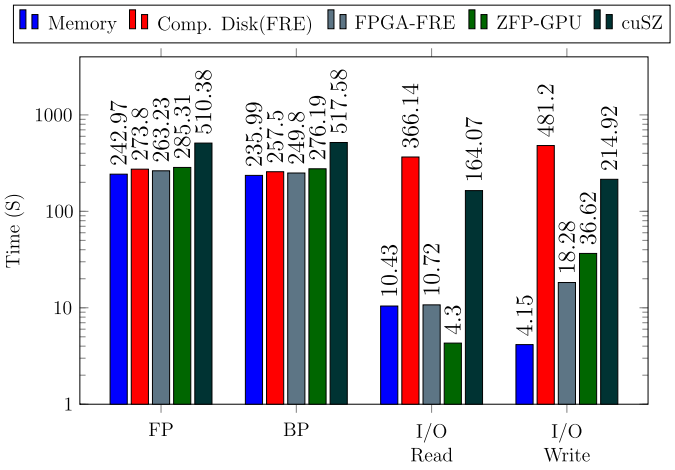

Fig. 12. Components' execution times at 11 threads for the FPGA-FRE, ZFP-GPU and cuSZ policies and 12 threads for the Memory and Compressed Disk (SW FRE) policies (regular model, machine 1).

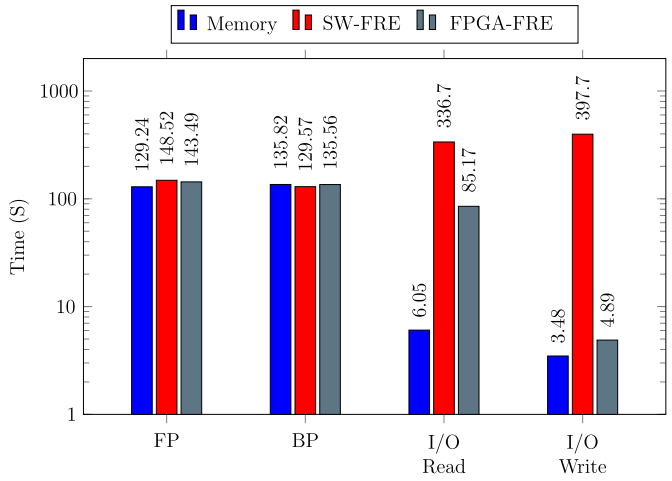

Fig. 13. Components' execution times at 23 threads for the FPGA-FRE policy and 24 threads for the Memory and Compressed Disk (SW FRE) policies (regular model, machine 2).

were slower than ZFP-GPU and FPGA-FRE however. Their performance is in line with the Disk policy where no compression is performed due to their low compression rates.

- FPGA-FRE with 11 and 23 worker threads, on machine 1 and 2, respectively, achieved **13 and 14**% less time than the Disk policy with 12 and 24 threads. Thus, even at this moderate data size where the RAM-based disk buffering (by the OS) is very effective, FPGA-FRE still offers a significant speedup over the Disk policy,
- When the application ran on a single socket, the proposed framework achieved almost identical performance to the Memory policy, something unattainable with SW-based compression. Spreading the compute threads over the two sockets slightly increased the I/O Read and Write times (servant threads need to move half the frame from/to socket 2 RAM through QPI),
- The I/O Read time on machine 2 increased significantly (~**23**% of the total time) due to the increased mismatch between the FPGA decompression's throughput (~458.58 MB/s) and the **BP** Kernel's throughput (~733 MB/s) at 23 threads. The main thread had to wait for the FPGA to finish the decompression before spawning another set of 23 **BP** worker threads. This problem could have been alleviated by using a second FPGA to overlap the decompression of two frames but that could not be done due to the size of the FPGA board.

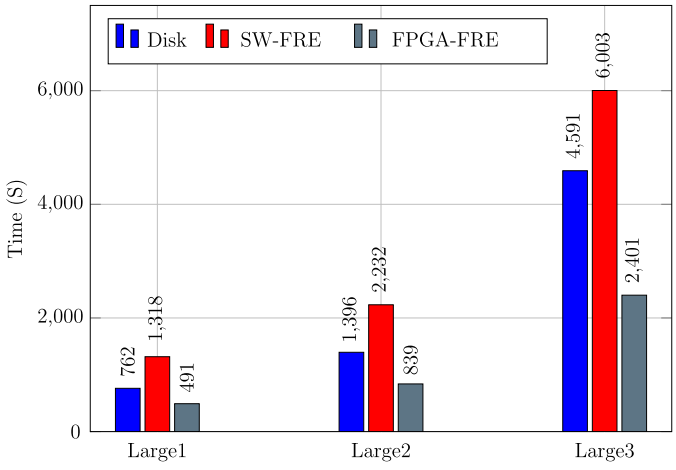

Fig. 14. Total execution times; FPGA-FRE, 23 threads, and the two disk policies at 24 threads.

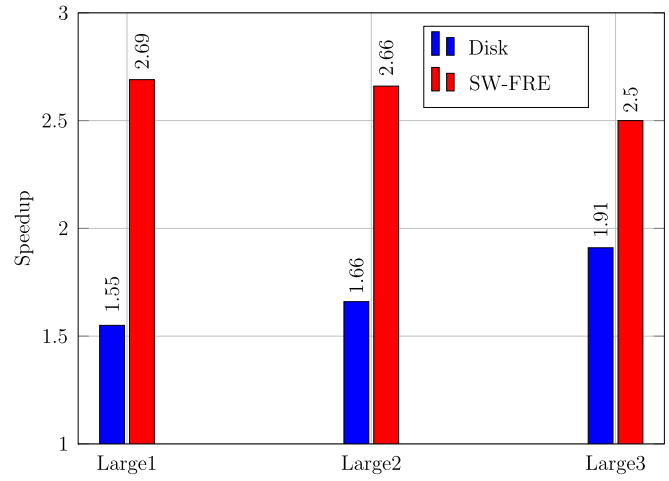

Fig. 15. Speedup (total execution time); FPGA based FRE, 23 threads over the two disk policies at 24 threads.

The above results showed that *RTMLab* under the proposed framework and FPGA-FRE can achieve performance levels that are very close to that with using a memory buffer, when the memory is actually insufficient to implement such a buffer. To demonstrate this, the three large models described in Table 2 (*Large1*, *Large2*, *Large3*) which cannot fit the memory, were used to evaluate the performance of the proposed framework against the Disk and Compressed Disk (using **SW-FRE**) policies. Only machine 2 was used as it has more physical cores and none of the large data models can fit in its RAM. This also reduces the benefit of RAM buffering of disk writes/reads by the operating system (OS).

Figs. 14 and 15 show the total execution times and speedups for the three policies. FPGA-FRE achieved the best execution time with up to ~**50**% improvement over the Disk policy even with one less worker thread. The performance gap between FPGA-FRE and the Disk policy increased with the data size, even when the number of threads is higher for the latter (from **13**~**14**% at the regular model to ~**50**% at the largest model). Hence, the proposed scheme becomes more advantageous as the data size increases. SW-FRE compression did not outperform the disk policy for any data size. This confirms the earlier conclusion that SW-based compression is not a viable option. These results demonstrate the effectiveness of the proposed framework to accelerate applications that generate large intermediate data that need to be ferried back and forth from/to the local disk storage.

Fig. 16 shows a breakdown of the execution time for FPGA-FRE for the large data models. All components' times increase monotonically

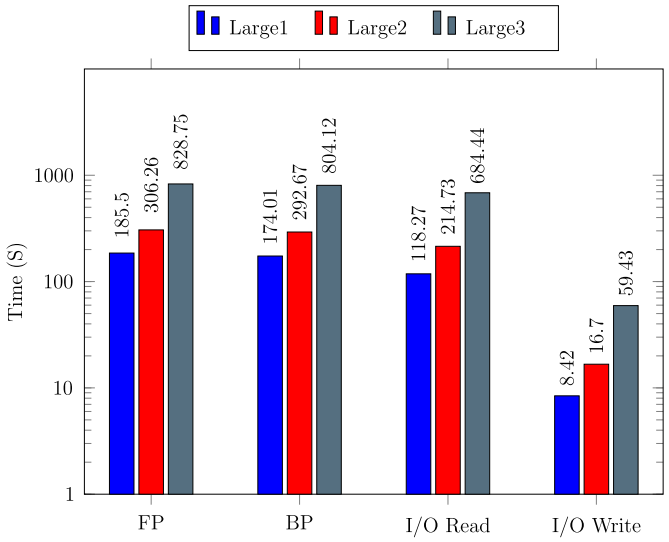

Fig. 16. Components' execution times with FPGA-FRE versus data size.

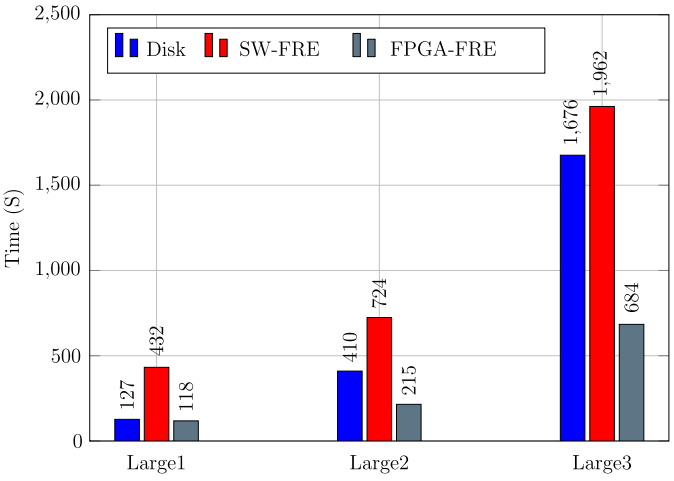

Fig. 17. I/O Read times; FPGA-FRE (23 threads) versus the two disk policies (24 threads).

with the data size (no component is affected differently). Figs. 17 and 18 give more insights into the performance of FPGA-FRE compared to the other policies. FPGA-FRE I/O Read speedup over the Disk policy was relatively modest (**1.1**X−**3.1**X) due to the decompression throughput mismatch with the **BP** kernel and the memory disk buffering by the OS. The I/O Read speedup increases with the data size (due to the diminished benefit of disk buffering). The I/O Write speedup of FPGA-FRE over the Disk policy was very significant ranging from **22.5**X to **43.9**X across the different data models. Even though the FPGA-FRE I/O Write speedup over the disk policy decreased as the model's size increased, it is a very small portion of the total execution time (unlike the I/O Read time) that it did not affect the overall speedup as evident from Fig. 15.

6.4. Resource utilization and throughput comparisons

Table 6 shows a comparison between the proposed **FRE** compression scheme with other published FPGA-based schemes (both lossy and lossless) in terms of compression ratios, resource utilization, and throughput. The throughput reported by most of the other works is *raw* throughput, with estimated from the maximum frequencies reported by the FPGA synthesis tool, FPGA board's DDR RAM throughput (i.e. assuming the data in the RAM), or for tightly coupled FPGA/CPU platform (e.g., HARP). I.e. no computing kernels running on the CPU and no

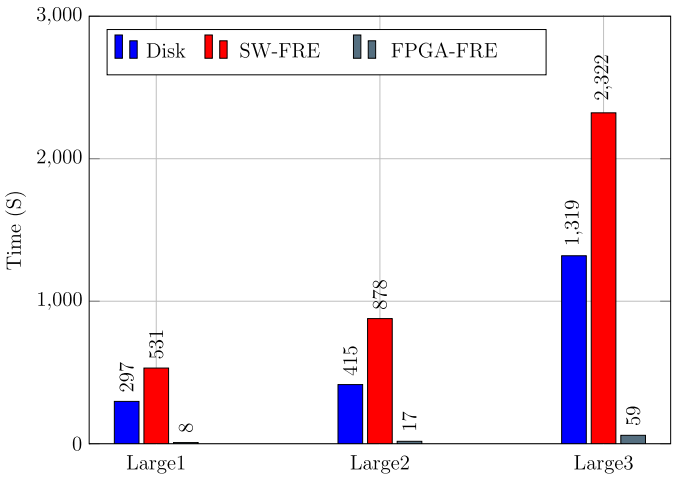

Fig. 18. I/O Write times; FPGA-FRE, 23 threads versus the two disk policies (24 threads).

communication overhead. Resource utilization of the compression and decompression engines is denoted as CE, and DE, respectively (the total resource utilization includes both engines plus the interface circuitry). Resource utilization is measured in ALMs (adaptive logic modules for Intel FPGAs; equivalent to 2.5 4-input LUTs), CLBs (configurable logic blocks, or slices; for Xilinx FPGAs, a CLB has two slices, and a slice has four lookup tables). Each ALM can function as two 4-input LUT or one big 8-input LUT. Proposed solution does not utilize any DSP or BRAMs. The resource utilization of the proposed scheme is orders of magnitude less than the other techniques with similar throughput.

As Table 6 shows, lossless compression techniques' compression ratios peaks at ∼2.5. The compression ratios reported for WaveSZ [25] are based on using SZ 1.4 (with GZIP) with a value-range-based relative error bound of 10^{-3}. This mode utilizes an error bound equals to the data value range (i.e., max - min) multiplied by 10^{-3}. For data with very large dynamic range (like seismic data), the error bound will be too large for the small values and hence they will all be predicted, resulting in a much higher compression ratio. This also requires extra pre-processing of the input data (on top of the linearization) on the host CPU, thus slowing down the compute kernels further.

SW-based SZ 1.4 lossy compression scheme [29] with point-wise relative error mode with a fixed 2^{-6} relative error (without GZIP) and FPZIP [12] with 26-bit precision were also included in Table 6 for reference. For both, and the proposed FRE scheme, the compression ratios are reported for two sets of data; Hurricane Isabel (average for all 13 fields), and actual seismic snapshots. The results show that SZ 1.4 has similar compression ratio to the proposed FRE scheme, but FPZIP has better throughput than SZ 1.4.

The other lossy compression implementation included in the table, is ZFP-V, a variation of ZFP [26]. It utilized an absolute error bound of 10^{-3} which basically encoded many blocks with small values as zero blocks (many of the files in the H. ISABEL benchmark had ∼90% zero blocks). Even though this led to a much higher compression ratio, the information loss is not acceptable for most applications. Still, this implementation used two orders of magnitude more resources than the proposed FRE scheme and achieved less throughput (using the same FPGA board used in this work).

7. Conclusions

In this work, a novel approach for accelerating memory & I/O intensive HPC applications that generate large amount of intermediary floating point data is proposed. Instead of offloading the application kernels to a GPU or an FPGA, an FPGA-based compression/decompression engines are used to compress intermediary data in parallel with

Table 6
Comparison between the proposed **FRE** compression/decompression and other published FPGA-based compression schemes.

	Compression Type	Data Type	Compression Ratio	Resource Utilization (ALMs, CLBs, or Slices)			Throughput (GB/s)
				CE	DE	Total	
Proposed (FPGA-FRE)	lossy	float	5.5 (H. ISABEL) 4.8 (seismic snapshots)	90 ALMs	311 ALMs	2,783 ALMs	2.40
SZ 1.4 (No GZIP) [29]	lossy	float	3.35 (H. ISABEL) 4.8 (seismic snapshots)	N/A	N/A	N/A	0.03
FPZIP [12] (26-bit prec.)	lossless/lossy	float/double	3.03 (H. ISABEL) 3.4 (seismic snapshots)	N/A	N/A	N/A	0.08
ZFP-V [26]	lossy	float/double	~38 (H. ISABEL)	N/A	N/A	115,344/132,432 ALMs	1.50
DE-ZFP [27]	lossy	float/double	~30.9 (H. ISABEL)	39,229 ALMs	22,306 ALMs	68,840 ALMs	3.48
WaveSZ [25]	lossy	float	13.2 (H. ISABEL)	N/A	N/A	8,208 ALMs	0.94
Ueno et al. [6]	lossless	float	2.51	1,285 ALMs	783 ALMs	2,068 ALMs	0.59
Fowers et al. [13]	lossless	text	2.09	N/A	N/A	108,350 ALMs	5.60
Qioa et al. [14]	lossless	text	2.03	N/A	N/A	162,828 ALMs	12.8
Lee et al. [18]	lossless	text	2.69	1,633 CLBs	N/A	N/A	0.06
Batrik et al. [16]	lossless	float	N/A	216 Slices	N/A	N/A	0.22
Batrik et al. [17]	lossless	text	1.5	9,656 ALMs	N/A	N/A	2.00
Liu et al. [19]	lossless	text	2.05	345 Slices	155	500 Slices	0.24

the compute kernels on the CPU before storing the data in the disk and decompress it before delivering it to the kernels. A simple low-latency HW-suitable lossy compression scheme with fixed relative error (FRE) was developed and implemented. This scheme achieved ~5X compression ratio at 2^{-6} or ~1.5% point-wise relative error, which is similar to SZ compression (at the same point-wise relative error bound). The relative error can be halved with minimal reduction in compression ratio (~7%) with no performance penalty at all. The FPGA compression/decompression is overlapped with the host's compute kernels. Experiments showed that the developed scheme has minimal effect on the host application (low memory contention and minimal cache pollution) and enables the HPC application to attain performance levels similar to using in-memory data storage, even though the data was actually stored in the node's disk. It outperformed direct disk storage (even with OS memory buffering of Disk I/Os) for all tested data sizes and the performance gap actually increased with the data size (~50% reduction in execution time for the largest tested data size). Experiments also showed that SW-based compression/decompression is impractical as it incurs huge performance penalties. At the same point-wise relative error bound, results showed that GPU compression achieves good compression ratio and similar throughput to FPGAs but at significantly more power. Finally, compared to published FPGA-based compression techniques, the proposed FRE compression scheme utilized much less FPGA resources than other techniques with similar throughput.

CRediT authorship contribution statement

Saleh AlSaleh: Writing – review & editing, Visualization, Validation, Investigation, Data curation. **Muhammad E.S. Elrabaa:** Writing – review & editing, Writing – original draft, Validation, Supervision, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Conceptualization. **Aiman El-Maleh:** Supervision, Investigation, Formal analysis, Conceptualization. **Khaled Daud:** Visualization, Validation, Investigation. **Ayman Hroub:** Methodology, Investigation, Formal analysis. **Muhamed Mudawar:** Formal analysis, Conceptualization. **Thierry Tonellot:** Supervision, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Muhammad E. S. Elrabaa reports financial support was provided by Saudi Aramco.

Data availability

The authors do not have permission to share data.

References

[1] W. Vanderbauwhede, K. Benkrid, High-Performance Computing Using FPGAs, Springer, 2014.
[2] D. Jones, A. Powell, C. Bouganis, P. Cheung, Gpu versus fpga for high productivity computing, in: 2010 International Conference on Field Programmable Logic and Applications, 2010, pp. 119–124.
[3] S. Neuendorffer, K. Vissers, Streaming systems in fpgas, in: Embedded Computer Systems: Architectures, Modeling, and Simulation, 2008, pp. 147–156.
[4] L. Ma, L. Lavagno, M.T. Lazarescu, A. Arif, Acceleration by inline cache for memory-intensive algorithms on fpga via high-level synthesis, IEEE Access 5 (2017) 18953–18974, https://doi.org/10.1109/access.2017.2750923.
[5] K. Duwe, J. Lüttgau, G. Mania, J. Squar, A. Fuchs, M. Kuhn, E. Betke, T. Ludwig, State of the art and future trends in data reduction for high-performance computing, Supercomput. Front. Innov. 7 (1) (2020).
[6] T. Ueno, K. Sano, S. Yamamoto, Bandwidth compression of floating-point numerical data streams for FPGA-based high-performance computing, ACM Trans. Reconfigurable Technol. Syst. 10 (3) (2017) 18, 10/gbq74m.
[7] P. Ratanaworabhan, J. Ke, M. Burtscher, Fast lossless compression of scientific floating-point data, in: Proceedings of the Data Compression Conference, DCC '06, IEEE Computer Society, Washington, DC, USA, 2006, pp. 133–142.
[8] R. Abdelkhalek, H. Calendra, O. Coulaud, G. Latu, J. Roman, Fast seismic modeling and reverse time migration on a gpu cluster, in: Proceedings of the 2009 High Performance Computing and Simulation, HPCS'09, Leipzig, Germany, 2009, pp. 36–43.
[9] Y. Collet, T. Matsuoka, Lz4 - extremely fast compression, https://lz4.github.io/lz4/.
[10] J. loup Gailly, M. Adler, The gzip home page, http://www.gzip.org/.
[11] M. Isenburg, P. Lindstrom, J. Snoeyink, Lossless compression of predicted floating-point geometry, Comput. Aided Des. 37 (8) (2005) 869–877, https://doi.org/10.1016/j.cad.2004.09.015.
[12] P. Lindstrom, M. Isenburg, Fast and efficient compression of floating-point data, IEEE Trans. Vis. Comput. Graph. 12 (5) (2006) 1245–1250, https://doi.org/10.1109/TVCG.2006.143.
[13] J. Fowers, J. Kim, D. Burger, S. Hauck, A scalable high-bandwidth architecture for lossless compression on FPGAs, in: 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines, 2015, pp. 52–59, 10/ghtcmk.
[14] W. Qiao, J. Du, Z. Fang, M. Lo, M.F. Chang, J. Cong, High-throughput lossless compression on tightly coupled CPU-FPGA platforms, in: 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2015, pp. 37–44, 10/ghtck6.
[15] F. Knorr, P. Thoman, T. Fahringer, Efficient lossless compression of scientific floating-point data on gpu, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '21, in: Article, vol. 93, Association for Computing Machinery, 2021, pp. 1–14.
[16] M. Bartík, S. Ubik, P. Kubalik, LZ4 compression algorithm on FPGA, in: 2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS), 2015, pp. 179–182, 10/ghj3vh.
[17] M. Bartík, T. Beneš, P. Kubalík, Design of a high-throughput match search unit for lossless compression algorithms, in: 2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC), 2019, pp. 0732–0738, 10/gh6mxd.

[18] S.M. Lee, J.H. Jang, J.H. Oh, J.K. Kim, S.E. Lee, Design of hardware accelerator for Lempel-Ziv 4 (LZ4) compression, IEICE Electron. Express 14 (11) (2017), 10/ghj3t8.
[19] W. Liu, F. Mei, C. Wang, M. O'Neill, E.E. Swartzlander, Data compression device based on modified LZ4 algorithm, IEEE Trans. Consum. Electron. 64 (1) (2018) 110–117, 10/ghj3vb.
[20] S. Lakshminarasimhan, N. Shah, S. Ethier, S. Klasky, R. Latham, R. Ross, N.F. Samatova, Compressing the incompressible with ISABELLA: in-situ reduction of spatio-temporal data, in: E. Jeannot, R. Namyst, J. Roman (Eds.), Euro-Par 2011 Parallel Processing, Springer, Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 366–379.
[21] D. Tao, S. Di, Z. Chen, F. Cappello, 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 (IPDPS), May 2017, pp. 1–2.
[22] J. Tian, S. Di, K. Zhao, C. Rivera, M. Hickman Fulp, R. Underwood, S. Jin, X. Liang, J. Calhoun, D. Tao, F. Cappello, Cusz: an efficient gpu-based error-bounded lossy compression framework for scientific data, in: Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques, PACT '20, Association for Computing Machinery, 2020, pp. 3–15.
[23] S. Jin, P. Grosset, C.M. Biwer, J. Pulido, J. Tian, D. Tao, J. Ahrens, Understanding gpu-based lossy compression for extreme-scale cosmological simulations, in: Proceedings of the 2020 IEEE International Parallel and Distributed Processing Symposium, IPDPS '20, IEEE, 2020, pp. 105–115.
[24] Q. Xiong, R. Patel, C. Yang, T. Geng, A. Skjellum, M.C. Herbordt, Ghostsz: a transparent fpga-accelerated lossy compression framework, in: 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2019, pp. 258–266.
[25] J. Tian, S. Di, C. Zhang, X. Liang, S. Jin, D. Cheng, D. Tao, F. Cappello, Wavesz: a hardware-algorithm co-design of efficient lossy compression for scientific data, in: Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 74–88.
[26] G. Sun, S. Jun, ZFP-V: hardware-optimized lossy floating point compression, in: 2019 International Conference on Field-Programmable Technology (ICFPT), 2019, pp. 117–125.
[27] M. Habboush, A.H. El-Maleh, M.E. Elrabaa, S. AlSaleh, De-zfp: an fpga implementation of a modified zfp compression/decompression algorithm, Elsevier Microprocess. Microsyst. J. 90 (4) (2022) 104453–104462, https://doi.org/10.1016/j.micpro.2022.104453.
[28] M. Jacobsen, D. Richmond, M. Hogains, R. Kastner, Riffa 2.1: a reusable integration framework for fpga accelerators, ACM Trans. Reconfigurable Technol. Syst. 8 (4) (2015) 22:1–22:23, https://doi.org/10.1145/2815631.
[29] S. Di, F. Cappello, Fast error-bounded lossy hpc data compression with sz, 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), https://doi.org/10.1109/ipdps.2016.11, May 2016.
[30] P. Lindstrom, Fixed-rate compressed floating-point arrays, IEEE Trans. Vis. Comput. Graph. 20 (12) (2014) 2674–2683, https://doi.org/10.1109/TVCG.2014.2346458.
[31] Nvcomp, https://developer.nvidia.com/nvcomp, Feb 2023.
[32] Szcompressor, Measuring memory footprint issue 76 szcompressor/cusz, https://github.com/szcompressor/cuSZ/issues/76.

Saleh Alsaleh received B.Sc. degree (Honos.) in Computer Science, and the M.A.Sc. degree in Computer Engineering from King Fahd University of Petroleum and Minerals (KFUPM), in 2016, and 2019, respectively. He is currently a lecturer with the Computer Engineering Department at KFUPM. His research interest includes Digital Design, FPGA, Embedded Systems and Computer Architecture.

Muhammad E. S. Elrabaa received M.A.Sc. and Ph.D. degrees in Electrical & Computer engineering from the University of Waterloo, Waterloo, Canada, in 1991 and 1995, respectively. From 1995 until 1998, he worked as a senior component designer with Intel Corp., Portland, Oregon, USA. He is currently a Professor with the computer Engineering department, King Fahd University of Petroleum & Minerals (KFUPM). His research interests include reconfigurable computing, cloud-based custom computing machines and Systems-on-Chip. He authored and co-authored numerous papers, a book and holds ten US patents.

Aiman H. El-Maleh received the B.Sc. degree (Hons.) in computer engineering from King Fahd University of Petroleum and Minerals, in 1989, the M.A.Sc. degree in electrical engineering from the University of Victoria, Canada, in 1991, and the Ph.D. degree in electrical engineering, with dean's honor list, from McGill University, Canada, in 1995. He is currently a Professor with the Computer Engineering Department, KFUPM. He was a Member of Scientific Staff with Mentor Graphics Corporation and the Leader in design automation, from 1995 to 1998. He holds seven U.S. patents. His research interests include synthesis, testing, and verification of digital systems, defect and soft error tolerance design, VLSI design, design automation, computer vision, machine learning, deep learning and efficient FPGA implementations of deep learning and data compression techniques.

Khaled Daus is a PhD student at the Information and Computer Science Department at KFUPM. He had obtained his MS degree from the Computer Engineering Department, KFUPM. His work is focused on coding and compression.

Ayman Hroub received the B.Sc. degree in Computer Systems Engineering from Birzeit University in 2008. He received the M.Sc. and Ph.D. degrees in Computer Engineering from KFUPM in 2011, and 2016, respectively. He was an Assistant Professor in Computer Engineering at KFUPM between 2016 and 2021. In 2018/2019 he visited TU Delft in The Netherlands as a Postdoctoral Researcher in Quantum and Computer Engineering Department. Currently, Ayman is an Assistant Professor in the Department of Electrical and Computer Engineering at Birzeit University. Ayman's research interests include computer architectures, in-memory computing, and secure architectures. He co-authored several papers and holds three U.S. patents.

Muhamed Mudawar obtained his B.Sc. degree with highest honors in Electrical Engineering from the American University in Beirut in 1986, and his Ph.D. degree with highest honors in Computer Engineering from Syracuse University in 1993. He is currently with the Computer Engineering department at King Fahd University of Petroleum and Minerals in Saudi Arabia. He was a faculty member in the Computer Science department at the American University in Cairo between 1993 and 2004. He was also a faculty member at King Abdullah University of Science and Technology (KAUST) in 2010. He is the author of numerous articles in journal and conference proceedings, and the sole author of three inventions (patents). His research interest includes parallel computer architectures, computer arithmetic, interconnection networks, parallel computing, and compilers.

Thierry Tonellot received a M.S. (1994) in Applied Mathematics from University Pierre et Marie Curie (Paris VI), France and a Ph.D. (2000) in Applied Mathematics from University Rene Descartes (Paris V), France. He was a Research Geophysicist with IFP, Rueil-Malmaison, France, for about ten years, where he worked in the field of seismic inversion and reservoir characterization. In 2006 he joined the Expec Advanced Research Center of Saudi Aramco in Dhahran, Saudi Arabia, as Research Geophysicist. His professional interests include high performance computing, seismic imaging and signal processing.