Microprocessors and Microsystems 90 (2022) 104453

Contents lists available at ScienceDirect

Microprocessors and Microsystems

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

DE-ZFP: An FPGA implementation of a modified ZFP compression/decompression algorithm☆

Mahmoud Habboush, Aiman H. El-Maleh *, Muhammad E.S. Elrabaa, Saleh AlSaleh

Computer Engineering Department, King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia

ARTICLE INFO

Keywords:
Hardware implementation
Lossy compression
FPGA acceleration

ABSTRACT

In this work, we present DE-ZFP: a hardware implementation of modified ZFP compression and decompression algorithms on a Field Programmable Gate Array (FPGA). It can be used to accelerate applications running on a host CPU that generates large volumes of floating point data. The proposed design uses dictionary-based encoding (DE) in lieu of ZFP's original embedded encoding to maximize hardware performance. Furthermore, the block encoder logic was optimized such that the loss of compression efficiency due to DE remains within 4%–13% of the original ZFP software implementation, with up to 19x improvement in throughput.

1. Introduction

Many applications are producing huge amount of data in the form of floating-point numbers. This is especially true for high-performance computing (HPC) applications. Many enterprises (such as universities and research centers) are building HPC clusters using commodity hardware (CPUs + Accelerators + Storage) to handle their heavy applications. Scientific applications such as the Coupled Model Intercomparison Project by the Community Earth system produced nearly 2.5 PB of data, and post analysis introduced almost 170 TB of additional data [1]. Movement of such data represents a significant performance bottleneck in HPC systems. A typical strategy to alleviate this problem is to use data compression/decompression on node-attached accelerators to reduce the amount of data to be moved without causing delays to the main applications running on these nodes.

Lossless compression techniques such as LZ4 [2] and GZIP [3] are dictionary-based, while others use linear prediction combined with variable length encoding [4,5].

Dictionary-based techniques provide limited compression. Similarly, for non-linearly related data, linear prediction techniques fail to provide acceptable compression ratios. To predict non-linear relationships, Burtscher et al. [6] used hash function. Still, lossless compression schemes have barely achieved more than 2x compression. This has led to more acceptance of lossy compression for scientific HPC applications. Several lossy compression techniques for floating-point data have been proposed. ISABELA [7] sorts data and then uses B-spline interpolation in performing compression but suffers from low compression ratios as it has to store data index for each point. Squeeze (SZ1.1) [8] uses best-fit curve fitting models based on using preceding neighbor, linear curve or quadratic curve fitting. It has the advantage of providing error-bounded compression. In [9], an improved version of Squeeze (SZ1.4) has been proposed which significantly improves the percentage of predicted points based on neighborhood values along multi-dimensions. Huffman coding is used to encode the generated set of prediction points to maximize compression. SSEM [10] compression scheme has been proposed for lossy compression of floating-point values for application-level checkpoint/restart. It is based on wavelet transformation, quantization and encoding. Lossy techniques such as ISABELA [7] and SZ [8,9] tightly rely on the smoothness of the data to make reasonable predictions; however, simulation data usually manifests sharp data changes. Such data behavior notably affects the prediction accuracy and therefore degrades the quality of compression. While SSEM [10] attempts to lessen the dependency on the smoothness of data, it fails to control the precision of compression based on a user-specified threshold. ZFP [11] is a versatile compression technique that is loosely based on the algorithm described in [12]. ZFP relies on a new orthogonal block transform and embedded encoding scheme which offers efficient compression for multidimensional floating-point arrays by partitioning the data into blocks of 4^d values (e.g. $4 * 4 * 4$ values in 3D). ZFP achieves better and faster compression than other techniques [13]. In contrast to prediction-based compression techniques, ZFP can be parallelized and pipelined rendering it a better candidate for hardware implementation.

☆ This research was funded by KFUPM under project No. DF191015.

* Corresponding author.
E-mail addresses: s201659120@kfupm.edu.sa (M. Habboush), aimane@kfupm.edu.sa (A.H. El-Maleh), elrabaa@kfupm.edu.sa (M.E.S. Elrabaa), salehs@kfupm.edu.sa (S. AlSaleh).

https://doi.org/10.1016/j.micpro.2022.104453
Received 9 November 2020; Received in revised form 23 October 2021; Accepted 15 January 2022
Available online 29 January 2022

Field-Programmable Gate Arrays (FPGAs) have demonstrated huge performance and power advantages over CPUs and GPUs for HW implementation of streamed applications that requires fine-grain parallelism [14]. FPGAs are commodity integrated circuits that are prefabricated and can be configured to implement PCIe-attached HW accelerators. FPGAs enable implementations with highly flexible fine-grained parallelism and associative operations such as broadcast and collective response. Several high throughput FPGA-based hardware implementations have been proposed in the literature for implementing lossless compression such as LZ4 [15–17] and GZIP [18–20]. On the other hand, there are only few lossy compression hardware implementations in the literature. GhostSZ [21] and waveSZ [22] both implement modified versions of SZ1.4 but waveSZ outperforms GhostSZ. waveSZ is based on a HW/SW co-design implementation using OpenCL [23] that utilizes a wavefront memory layout [22]. Pre-processing of the data is performed on the host CPU while the main steps of SZ1.4 are pipelined on the FPGA. It took 4 parallel pipelined engines (i.e. 4 parallelism) for the HW implementation to reach the bandwidth limit of a 4-lane PCIe Gen 3 connection to the host CPU. In [24] a co-design implementation of a modified ZFP lossy compression, dubbed ZFP-V is proposed. The inherently serial embedded coding scheme of ZFP was replaced with a more hardware-optimized scheme using a variable-length header. It was implemented on a special heterogeneous platform (Xeon processor + Arria 10 FPGA) that features FPGA-CPU connection through Quick Path Interface and PCIe channels, shared RAM, and coherency. Still, a single pipeline implementation barely outperformed a 4-threaded SW implementation on the host CPU. For most of the benchmarks reported, the HW implementation did not even approach the communication limits of any of the interconnection fabrics of the used platform (QPI or the 8-lane PCIe Gen 3 channels). The same group used the same technique in [25] (but called it sZFP) to improve a stencil computation speed on an FPGA by compressing/decompressing the data before/after writing/reading it to/from the FPGA board's RAM. They divide the data into independent chunks that can be processed by independent pipelines (that each had similar throughput as their previous implementation) which improved throughput significantly. As with their previous work, ZFP-V, they included decompression implementation, which is needed in HPC applications that generate huge intermediate data that need to be stored and then later used (e.g. Seismic Simulations).

In this work, we present DE-ZFP: an FPGA-based implementation of a modified ZFP compression/decompression algorithm with the following novelties:

- The original embedded encoding scheme of ZFP (serial in nature) is replaced with a dictionary-based encoding that reduces the encoding time to a single clock cycle,
- To minimize the degradation in compression efficiency, the number of encoded blocks was minimized using specially-developed smart output logic
- Unlike some published HW implementations, both compression and decompression were implemented as independent channels on the FPGA. They can be operated in parallel to serve the needs of HPC applications that require the compression/decompression of intermediate results. With minimum resource utilization, this also allows implementing other computing accelerators on the same FPGA as independent channels,
- Custom RTL-level design of all circuitry (as opposed to HLS) is done to optimize performance, area, and power. Still, the developed blocks can be configured to provide fixed-accuracy or fixed-precision. This combined with a simple communication framework, enable the main application to use the compression/decompression HW using simple APIs. Hence, no need to use a run-time environment (as the case with OpenCL) on the host CPU that would impact its performance.

The rest of the paper is organized as follows. In Section 2, an overview of ZFP compression and decompression algorithm is presented followed by the proposed changes to the algorithm to improve its HW performance. In Sections 3 and 4, the proposed FPGA-based implementation of DE-ZFP is described. Section 5 examines the design interface. Experimental results with comparisons to previous works are given in Section 6 and conclusions are given in Section 7.

2. ZFP compression/decompression

ZFP compression algorithm [11,12] supports different compression modes: fixed-rate, fixed-precision and fixed-accuracy for error bounded systems. ZFP works on both, double and single precision floating-point numbers. ZFP algorithm is divided into 4 logical, independent steps that can be pipelined:

1. **Block-floating-point representation**: floating-point numbers are converted into a common exponent (the maximum unbiased exponent in the block), fixed-point format [26]. This maximum exponent is stored in the header of each block.
2. **Orthogonal block transform**: spatially correlated values are decorrelated using the lifting scheme [27] which involves 16 transforms performed on each plane. This results in close-to-zero coefficients that can be compressed efficiently
3. **Coefficient re-ordering**: signed integer coefficients are reordered so that statistically they appear in a roughly decreasing order [28]. Coefficients corresponding to low frequencies tend to have larger magnitude and are listed first. After reordering, the 2's complement integers are converted to their negabinary representation (using base -2). Small numbers in negabinary tend to have many leading zeros irrespective of their sign. For example, −1 as a 32-bit 2's complement value is represented as 0xffffffff, while in negabinary it is represented as 0x3.
4. **Embedded encoding**: The 64 floating point values of a block are transposed into 32 bitplanes (bitplane *j* is comprised of the 64 bits at bit position*j*). Bitplanes are then compressed losslessly exploiting the property that the coefficients tend to have many leading zeros that do not need to be encoded explicitly. Embedded coding encodes a bitplane from LSB to MSB as follows: if the bitplane is zero, a 0 is written and encoding ends. Otherwise, a 1 is written followed by all bits until a 1 is encountered, and written. Each time a bit is written, the bitplane is shifted to the right. Afterwards, the bitplane is tested again to find if it became zero or not, and so on. Encoded bitplanes are written from MSB to LSB. A cumulative variable ***n***, initially set to zero, is used in the encoding. At the beginning of encoding each bitplane, ***n*** bits are written from the LSBs (bit n-1 to 0), and the bitplane is right shifted by ***n*** bits. For every bit written from the bitplane, ***n*** is incremented. Using ***n*** to keep track of how many bits were written in the previous bitplane mitigates the overhead of embedded encoding as subsequent bitplanes tend to have at least ***n*** set bits to encode. Unfortunately, that also prevents parallelizing the bitplanes encoding (since each bitplane will depend on the previous bitplane's ***n*** value). Fig. 1 illustrates an example for compressing four 5-bit values.

2.1. Compression modes

In fixed-rate mode, the user specifies a *fixed* size per block, and the compression of bitplanes stops once it reaches the size limit. In fixed-precision, the user explicitly chooses the number of bitplanes to encode. Lastly, in fixed-accuracy the user specifies a tolerance, i.e. $|compressed\ value - original\ value| \leq tolerance$, and the number of encoded bitplanes is computed accordingly as $emax - minexp + 3d$, where d is the dimensionality of the data (in our case $d = 3$), and *minexp* is $\lfloor (\log_2 (tolerance)) \rfloor$.

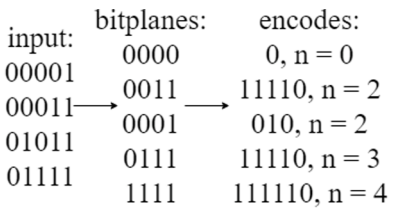


Fig. 1. Bitplane embedded encoding example.

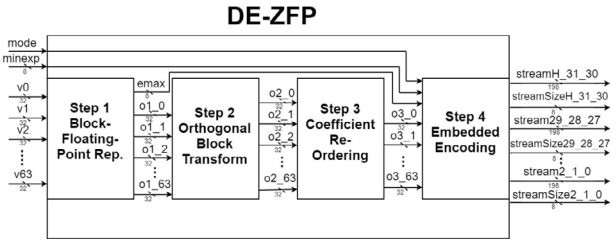


Fig. 2. DE-ZFP compressor block diagram.

2.2. Proposed modifications

The embedded encoding of the original ZFP algorithm is performed by reading bitplanes serially (bitplane by bitplane, and each bitplane bit by bit). This imposes a significant delay as encoding each bit will need a full cycle, and the dependency between the planes hinders parallelism in encoding. Hence, we designed a dictionary-based encoding that encodes all bitplanes in a single cycle. Further, we modified the output of the compressor by smartly removing blocks if they only contain zeros to keep the compression ratio percentage increase within 4%–13%.

3. Implementation of DE-ZFP compression

The DE-ZFP compressor is implemented as four combinational blocks as shown in Fig. 2. It receives 64 single floats and generates a variable length compressed block and supports fixed-accuracy and fixed-precision modes. Our design is focused on minimizing the compression ratio percentage increase while maximizing throughput. The proposed hardware implementation of DE-ZFP compression/decompression on the FPGA is meant to provide data compression/decompression as a service to a main application on the host CPU. Additionally, a minimalist communication frame work, comprised of simple APIs, was used for communication between the host application and the FPGA circuitry that does not require any run-time environment. Our implementation investigates only single-precision floats.

3.1. Block-floating-point representation

This stage is made of two parts. The first part finds the maximum exponent in the block (*emax*) using a 6-level tree of comparators. The second part converts all values in the block to the Block-Floating-Point Representation with respect to *emax*. The original SW implementation of ZFP accomplishes this via floating-point multiplication which would be costly on FPGAs. Instead, each value is represented by left appending 3-bits (001) to its mantissa; the first 0 is for the extra precision needed by the transform stage, the second is for the sign bit and the 1 for the implicit leading bit, and right appending 6 zeros and then shifting the result to the right by the difference of *emax* and the input's exponent. If the input value is negative, the output will be converted to its 2's complement negative representation as shown in Fig. 3.

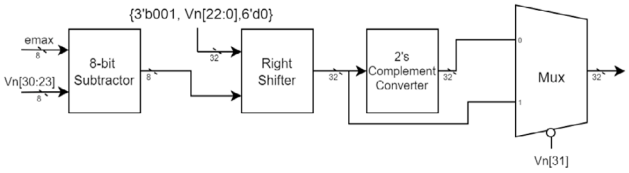


Fig. 3. Conversion of values to block floating point representation.

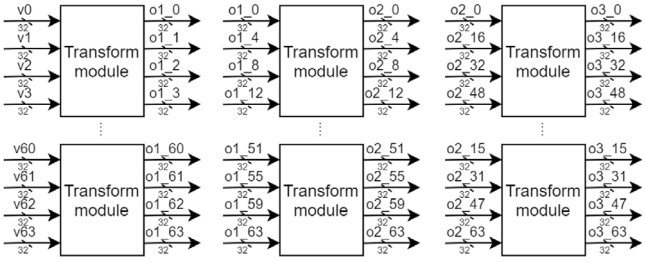


Fig. 4. Orthogonal block transform data path.

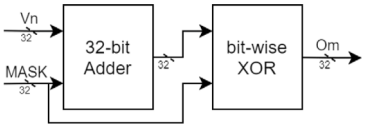


Fig. 5. Coefficient re-ordering block.

3.2. Orthogonal block transform

This stage takes 64 input values and generates the respective 64 transformed output values in three steps; one per plane. We designed a transform module that takes 4 input values and outputs the transformed values. The 1st stage comprises 16 parallel blocks that transforms the values on the x plane. This is followed by the 2nd stage which again uses 16 parallel blocks to transform the values on the y plane, followed by the 3rd stage for values on the z plane. The interconnections between transformer levels reflect the dimensional grouping as shown in Fig. 4.

3.3. Coefficient re-ordering

Coefficient reordering and conversion to the negabinary representation are done simultaneously. Each input is converted to its negabinary format and connected to the ordered output according to its frequency magnitude. All 64 values are processed in parallel as shown in Fig. 5, where *Om* is the output order corresponding to the *n*'th input order.

3.4. Dictionary-based encoding

Unlike the sequential embedded encoding in the original ZFP algorithm, dictionary-based encoding allows parallel lossless encoding of bitplanes while maintaining the same accuracy. As illustrated in Fig. 6, bitplanes are compared to an array of selected number of non-zero leading bits, counting the number of leading bits starting from the least significant bit until the last non-zero bit found. Each entry will have a respective code. The number of non-zero leading bits of the bitplane will be matched with the minimum number to represent those bits from the match array. The code associated with the match from the match array will be encoded, followed by m least significant bits from the bitplane, where m is the match value. The code of the bitplane is written from most significant to least significant.

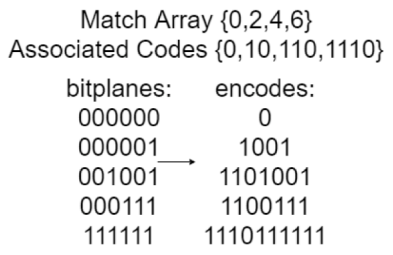


Fig. 6. Dictionary-based encoding example with 6-bit bitplanes.

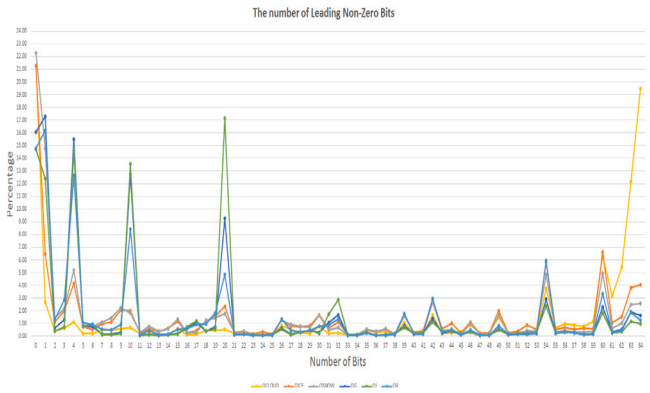


Fig. 7. Frequency of the number of non-zero leading bits.

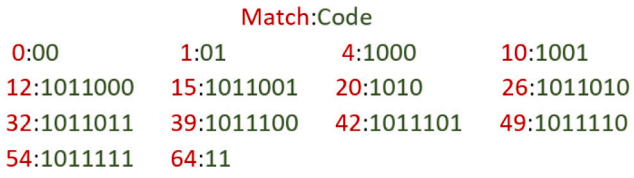


Fig. 8. Proposed dictionary-based encoding.

This scheme has no dependency between bitplanes, hence all bitplanes can be encoded in parallel. However, to minimize the degradation of compression efficiency incurred by writing the codes, match values and their associated codes must be wisely chosen. This obviously cannot be done for each data input, so we have selected optimum codes for different match values based on analyzing the number of leading non-zeros for many different scientific data sets. Only the bitplanes that are encoded using fixed-accuracy mode with tolerance 10^{-6} were considered, and all the all-zeros bitplanes (which are most common) have been excluded from this analysis. Analysis results for the 6 data sets used in this work [29] are shown in Fig. 7. Bitplanes with close to zero or 64 leading non-zero bits are prominently more common. Therefore, the encoding had to prioritize the end-points by giving them shorter codes. Further, certain bit lengths such as 10 and 20 are noticeably more common in all data sets analyzed. In addition, it was important to minimize the maximum encoded bitplane length to improve the worst-case compression scenario. The proposed encoding shown in Fig. 8 achieved the best compression ratio and combinational delay between many different explored variations.

The hardware implementation of this step is composed of the following operations: First, values are transposed into bitplanes. Second, bitplanes are encoded in parallel using the dictionary-based encoding scheme which will generate the encoded bitplanes, called streams, with a size signal for each stream. Based on the compression parameters, a 6-bit value *precision* will represent the number of bitplanes to encode, starting from the most significant bitplane (bitplane 31). For each bitplane that should not be encoded, its stream size will be logic And-ed with 0.

Joiner Circuit: After zeroing the sizes of unneeded streams, each three consecutive streams are joined together with respect to their sizes, and their sizes are added; streams 31 and 30 are joined with the header of the block. The header is 0 if the precision value is 0, i.e. the whole block will be decompressed with zeros, otherwise, 1 followed by the least significant 5 bits of the precision value, followed by *emax*. The header size will be either 1 or 14 bits. If the precision is 32, its 5 least significant bits will be 0s, this special case will be coded into the decompression circuit. To further optimize the compression ratio, we additionally consider the precision to be zero if the precision is 1 and bitplane 31 is only zeros, or if precision is 2 and both bitplanes 31 and 30 are only zeros, since both bitplanes are very likely to be zero. This approach allows DE-ZFP to reduce the size of a compressed block from 17 bits (in case of precision = 1) or 20 bits (in case of precision = 2) to a single bit, while representing the same block. This careful analysis helped reducing the compression ratio percentage increase for different data sets by around 10%. This stage results in 11 streams, each with a maximum size of 198 bits, since each encoded bitplane has a maximum size of 66 bits, as shown in Fig. 2. The 11 streams are concatenated with respect to their sizes, starting from the most significant stream, to form a compressed block.

4. DE-ZFP decompression implementation

The decompressor is designed in a similar manner to the compressor, i.e. based on four serial blocks. However, the first step of the decompressor is implemented as a sequential circuit, while the remaining three steps are still combinational. The decompressor receives a variable length encoded block, and outputs the decompressed block of 64 single precision floats.

4.1. Decompression of dictionary-based encoding

This stage is an 18-states FSM that generates 64, 32-bit output values. It first checks if the block is zero or not by reading the first bit; if 0 is read, then all values will be represented using zeros. Otherwise if the first bit is 1, the circuit reads the number of encoded bitplanes by reading the next 5-bit *precision* from the header of the compressed block. If the precision is zero, it will be replaced by 32. Then, it reads the next 8-bits that represent *emax*. Afterwards, it starts decoding the bitplanes one bitplane at a time, starting from bitplane 31, using if-else network to determine the code from the dictionary used for that particular bitplane. Depending on the code, the circuit will read the next *n* bits into a register of size 64 bits storing the decoded bitplane, where *n* is the value from the associated match array. After a bitplane is decoded into a 64-bit register, it will be injected into the 64 output registers, by inserting from right the *i'th* bit of the bitplane into output *i'th* register. *Precision* is decremented after decoding each bitplane. When *precision* is zero, each of the 64 output registers is shifted left by the difference between 32 and a registered *precision*, so that the first decoded bitplane values are now at bit position 31 of each output register. The circuit is interfaced with a variable read-length buffer that enables it to read variable number of bits each cycle as needed during the decoding process. Further details about this interface are given in Section 5.

4.2. Decompression of coefficient re-ordering

In this step, the values are re-ordered back to their sequential order from the frequency magnitude order. In addition, the values are converted back into the 2's complement representation from the negabinary representation. This is done by performing a bit-wise Exclusive-OR (XOR) between the input and the negabinary mask (0xAAAAAAAA), followed by subtracting the negabinary mask from the XOR result.

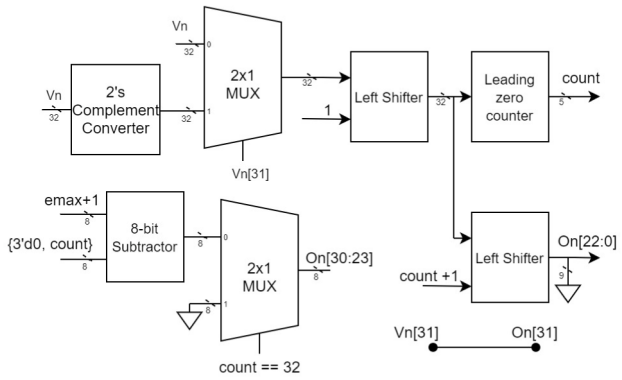


Fig. 9. Reconstruction of floating-point values.

4.3. Decompression of orthogonal block transform

Decompression of the orthogonal block transform is similar to the compression counterpart except that it reverses the order of transforms: 1st the z plane, then the y then the x, and the transform equations.

4.4. Decompression of block-floating-point rep

Fig. 9 shows how a floating-point value is reconstructed from its fixed-point 2's complement counterpart. The sign bit of the output float is set to the most significant bit of the 2's complement number, and if that bit is 1, we convert the 2's complement value to its positive format. The result is then shifted left by 1 bit to remove the extra precision bit. Afterwards, the number of leading zeros z of the shifted input is determined. If z is 32, then we set the exponent part to zero, otherwise we set it to the difference between $emax$+1 (to account for the implicit bit) and z. Finally, the shifted input is shifted again left by z+1 bits (plus one to remove the implicit leading bit). The mantissa of the float is set to the most significant 23 bits of the result. The mantissa is rounded according to IEEE754 nearest even rounding scheme, and if a carry out is generated during rounding, the fraction will be zero and the exponent is incremented by 1.

5. Interface design

The open-source PCI-e framework RIFFA [30] was used to provide communication between the main application on the host CPU and the compression/decompression engines via simple APIs. RIFFA API library is compiled with the host application and hence requires no run-time environment. Our design utilizes two channels, one per engine so they can operate in parallel. The PCIe link is a four-lanes GEN3 with an effective maximum throughput of 3.64 GB/s. The RIFFA interface to the circuit is 128 bits wide and works at a frequency of 250 MHz (i.e. it theoretically supplies/reads back 250 million 128-bit flits every second).

5.1. Compressor channel

The compressor channel is responsible for receiving the uncompressed data (4 floats per cycle) and accumulates them to form 64-floats blocks and transfers them to the compression engine. The compressed data is sent back to the host. The first received flits specifies the ZFP mode, ZFP minexp and the block count. The front-end (accumulator) triggers the compressor whenever a block of 64-floats is available. To save area and power, the compressor itself is left as a multi-cycle unpipelined combinational block since the front-end needs at least 16 cycles to read a new block. The delay of the compressor block is ~42 ns or 11 cycles in the Arria-10GX FPGA used in this work. The front end will overwrite the block in the compression engine with a new block when it receives an acknowledgment. This simple handshake along with the large margin (16 cycles) allotted for the compressor means the design can be easily ported to other FPGAs. Another circuit relays the variable length output data of the compressor engine that represents the 11 streams to the output FIFO along with each stream's size. Once all streams are sent or a stream with a size of zero is encountered, this circuit will stop and wait for the next block. This circuit works in parallel with the compressor. Since the output FIFO size is 128-bits, the variable-size streams need to be compacted into 128-bit flits. A special bit-addressable FIFO buffer (BAB) [31] was developed. The input/output ports have associated controls that specify the size of the data to be pushed/popped. The first data written into the BAB is the input block count because it will be needed for decompression. Whenever the BAB has 128-bits of compacted data, it sends them to the host application. A special circuit detects the last compacted streams, pad them to 128-bits before pushing them into the BAB so they can be sent to the host.

5.2. Decompressor channel

The decompressor channel receives compressed data and decompresses the data block by block. The header of the compressed data is the block count. Another BAB is used as a front-end allowing 128-bit flits to be pushed in, and the variable-sized data to be popped out by the decompressor engine. A sequential circuit in the decompressor reverts the dictionary-based encoding (DeStreamify) and is followed by three combinational blocks that implement the remaining steps. Again, these blocks were implemented using multicycle approach with a delay of ~23 ns (6 cycles) on the Arria-10GX FPGA. The DeStreamify is based on the decompression of embedded encoding circuit but has 5 additional states (total 23) to handle multiple blocks and to communicate with the BAB. The DeStreamify uses the BAB to read variable data sizes depending on its state and which bitplane is being decompressed. When a block is ready for the combinational logic, it will assert a valid signal and wait for the block to be consumed. Once consumed, block count is decremented and if not zero, it will start decoding the next block. Otherwise if block count is zero, DeStreamify will consume any remaining bits in the BAB which are the padded bits added by the compression channel. Every 64 32-bit floats generated by the decompressor channel are grouped into 128-bit flits and sent back to the host.

6. Experimental results

6.1. Experimental setup

The proposed DE-ZFP was implemented on an Intel Arria 10 GX FPGA board connected to the host via a 4-lanes Gen3 PCIe link. The implementation was evaluated and compared to the latest official ZFP SW library (0.5.5, [11]) using fixed-accuracy mode and two tolerance values: 10^{-3} and 10^{-6}. DE-ZFP was also compared with other published FPGA-based implementations of lossy compressors. ZFP was run on a clean Linux machine (i7-7500u, 12 GB RAM, 250 GB NVMe SSD). ZFP was not run on a GPU because it only supports CUDA execution in fixed-rate mode and not the fixed-accuracy mode. The comparisons include compression efficiency (uncompressed data size divided by compressed data size), performance (throughput and power consumption), and resource utilization. To validate the DE-ZFP design, we compressed a set of files and then decompressed them using both the official ZFP SW library and our hardware implementation and verified that the results match under the same compression parameters. For our tests, we used the data shown in Table 1 [29]. Using RIFFA APIs, a small program was developed to send and receive original/compressed data to the compression/decompression channels on the FPGA and receive the compressed/decompressed data back. For compression, the data

Table 1
Comparison of compression efficiency (original size/compressed sizes) between DE-ZFP (HW) and ZFP (SW) (10^{-6} tolerance).

Data set	Dimensions	Data field	ZFP	DE-ZFP	Degradation
ISABEL Hurricane	500 × 500 × 2	QCLOUD	32.80	30.93	5.7%
		QICE	43.20	38.96	9.8%
		QSNOW	43.05	39.35	8.6%
		Average for all fields (20)			7.15%
		Std Dev for all fields (20)			2.00%
SCALE-LETKF	1200 × 1200 × 2	QG	42.27	37.30	11.8%
		QI	37.50	33.69	10.2%
		QR	31.75	28.06	11.6%
		Average of all fields (12)			8.55%
		Std Dev of all fields (12)			3.12%
NYX	512 × 512 × 2	Baryon den.	1.64	1.56	5.0%
		Temperature	1.40	1.34	4.6%
		Average of all fields (6)			4.81%
		Std Dev of all fields (6)			0.41%

Table 2
Comparison of compression Efficiency (original size/compressed sizes) between DE-ZFP (HW) and ZFP (SW) (10^{-3} tolerance).

Data set	Dimensions	Data field	ZFP	DE-ZFP	Degradation
ISABEL Hurricane	500 × 500 × 2	QCLOUD	262.86	226.88	15.86%
		QICE	745.82	659.28	13.13%
		QSNOW	797.86	706.37	12.95%
		Average for all fields (20)			11.82%
		Std Dev for all fields (20)			4.60%
SCALE-LETKF	1200 × 1200 × 2	QG	610.55	541.29	12.8%
		QI	573.13	491.82	16.53%
		QR	505.64	428.84	17.91%
		Average of all fields (12)			12.61%
		Std Dev of all fields (12)			5.96%
NYX	512 × 512 × 2	Baryon den.	3.37	3.13	7.45%
		Temperature	1.44	1.37	4.87%
		Average of all fields (6)			5.28%
		Std Dev of all fields (6)			0.98%

files and other related fields including the input's *endianness*, compression mode, compression parameter and data dimension are processed by the compression API on the host before sending the data to the compression channel on the FPGA. If the compression mode is fixed-accuracy, the parameter value should be the tolerance, otherwise, the number of bitplanes to encode is specified, between 0 and 32. The DE-ZFP compressor does not compress the input sequentially, rather, it compresses 64 spatially adjacent values of dimension 4 × 4 × 4. Hence, the compression API converts the input file to a series of blocks representation. Also, if one of the input dimensions is not a multiple of 4, it will be padded as in the software ZFP library. Finally the program will pre-append the compression header before sending it to the FPGA. For decompression, the program will read the compressed data, send it to the decompression channel on the FPGA, and receives the blocked decompressed data, de-block it and write it in the node buffer.

6.2. DE-ZFP versus SW ZFP compression efficiency

Three data sets with a total of 38 fields were used to evaluate the compression efficiency of the HW implementation of DE-ZFP against that of the original ZFP (SW) as shown in Tables 1 and 2, which reports the compression ratios. The results show that the degradation in compression efficiency (i.e. increase in compressed data size due to the dictionary-based encoding) average ranges from a minimum of 4.81% to a maximum of 8.55% with 10^{-6} tolerance, and a minimum of 5.28% to a maximum of 12.61% with 10^{-3} tolerance. We achieved a best case of 3.94% with 10^{-6} tolerance, and 4.51% with 10^{-3} tolerance.

Table 3
Throughput comparisons (10^{-6} tolerance).

Data file	Compression (GB/s)			Decompression (MB/s)		
	ZFP	DE-ZFP	Speed-Up	ZFP	DE-ZFP	Speed-Up
QCLOUD	0.65	3.47	5.3X	1009.18	1786.50	1.8X
ICE	0.62	3.49	5.7X	1012.39	1809.88	1.8X
QSNOW	0.59	3.49	6.0X	859.94	1809.82	2.1X
QG	0.48	3.41	7.2X	922.43	1752.98	1.9X
QI	0.47	3.40	7.3X	899.92	1752.73	1.9X
QR	0.44	3.39	7.8X	812.45	1752.27	2.1X
Baryon density	0.09	1.69	19.4X	127.84	681.94	5.3X
Temperature	0.08	1.52	19.1X	115.34	604.31	5.2X

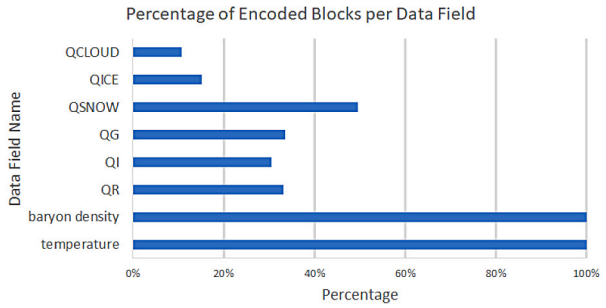


Fig. 10. Percentage of encoded blocks in each data file.

6.3. DE-ZFP throughput

The implemented multi-cycle compression engine has a maximum throughput of 5.4 GB/s (11 cycles to finish a 64-floats block at 250 MHz clock). If pipelined, it would process one block per cycle yielding a 59.6 GB/s. Also, data could be divided into independent chunks and multiple pipelines could be used in parallel. For our use case (where data is read/written from/to the CPU's RAM) the resulting throughput is, by far, beyond the 4-lane GEN3 PCIe link's capacity. Throughput measurements of DE-ZFP (HW) are shown in Table 3. We used 10^{-6} tolerance in our experiments to maximize computation in the compression & decompression process, as 10^{-6} tolerance will entitle encoding more bitplanes. The single-threaded SW ZFP throughput is also included in that table for comparison. DE-ZFP throughput was measured by dividing the total data size by total number of cycles (measured on the FPGA). This includes the communication time (both down link and up link) plus the compression time. ZFP throughput was measured for in-memory compression & decompression, i.e. without writing the output to disk to remove excess I/O time. The average of multiple runs per data field was measured. The number of runs ranged from 10 to 20 runs depending on the variation of results. The results are shown in Table 3. For data in which ZFP achieves high compression (first six files), DE-ZFP achieved up to 7.8x speedup. In fact, the DE-ZFP throughput for these cases approached the maximum throughput of the PCIe link of 3.64 GB/s. For decompression, the speedups are around 2x due to the sequential decoding of bitplanes. However, when the ZFP compression efficiency is low (last two files) our design exhibits higher speedups over the software version, reaching 19.4x for compression and 5.6x for decompression.

The variations in the speedup of DE-ZFP over ZFP is due to the way ZFP handles block compression in fixed-accuracy mode. When precision is found to be zero, ZFP terminates without further processing (it only finds emax) and starts the next block. In such cases, DE-ZFP would have limited speedup over ZFP because DE-ZFP needs to complete a full compression cycle. However, as more blocks and bitplanes are encoded, DE-ZFP offers higher speedup. This is illustrated in Fig. 10 which shows how the percentage of encoded blocks in each data field is relatively proportional to the speedup of DE-ZFP.

Table 4
DE-ZFP HW resource utilization.

Entity	ALM	Combinational ALUTs	Dedicated logic registers	ALM utilization
Compressor channel	39,229	56,312	10,563	9.2%
Decompressor channel	22,306	34,330	9763	5.2%
PCIe hard IP block	19	37	49	0.0%
RIFFA	7286	12,219	14,550	1.7%
Total	68,840	102,898	34,925	16.1%

Combinational ALUT represents what can be implemented by the combinational logic hardware of an Adaptive Logic Module (ALM).

Table 5
Power consumption of DE-ZFP implementation.

Component	Power
Transceivers' standby power[a]	2615.56 mW
Transceivers' dynamic power[a]	5109.41 mW
Core circuitry dynamic power	911.87 mW
FPGA static power dissipation	8407.04 mW
Total power	17,043.97 mW

[a]For four transceivers. These numbers would double if an 8-Lane PCIe link is used.

6.4. Resource utilization

The Arria 10 GX has 427,200 adaptive logic modules (ALMs) that are used to implement logic functions, arithmetic functions and register functions. The detailed resource utilization of the implemented DE-ZFP is shown in Table 4. In addition, the RIFFA interface used ~1 Mb of block memory.

6.5. Power consumption evaluation

Table 5 shows a detailed break down of the power consumption of the complete DE-ZFP circuitry using Intel's PowerPlay tool. As this table shows, the total core circuitry's dynamic power is 911.87 mW which represents only 5.3% of the total power. The high-speed transceivers and the FPGA's static power represent almost 95% of the total power consumption. Also, the core's dynamic power is obtained assuming that both channels are operating simultaneously at 250 MHz (i.e. each receiving and sending 1 GFloats/s simultaneously), something that cannot happen due to the shared PCIe link's maximum theoretical throughput of 4 GB/s. If a different interface is used (e.g. an 8-lane PCIe link), the core can process ~8 GB/s (compression and decompression) with energy consumption of 114.7 PJ/Byte and power efficiency (data volume processed) of 8.75 GB/s/W or 35.0 Gbits/s/W. This remarkable feat is mainly due to the adopted combinational implementation of the core compressor/decompressor functions and the avoidance of using excessive sequential circuits and the associated registers.

6.6. Comparison with other FPGA implementations

Since waveSZ [22] (based on SZ 1.4) uses relative error to control compression accuracy, we only compare our results with them in terms of throughput. A single compression core of waveSZ achieves less than 1 GB/s on the Hurricane data set, and it needs 4 cores to reach ~3.5 GB/s. Our design achieves ~3.5GB/s using a single compression core, i.e. 4 times faster than waveSZ. To our knowledge, ZFP-V [24] and sZFP [25] are the only complete FPGA implementation of ZFP. They reported resource utilization, compression ratios, and throughput but did not report power consumption. ZFP-V implemented a 3-D single-precision modified version of ZFP and reported numbers for a tolerances of 10^{-3}. While sZFP only implemented a 1-D double-precision modified version of ZFP and reported numbers for various tolerances (from 10^{-3} down to 10^{-6}). Further, sZFP uses 3 decompression pipelines to mitigate the sequential step of the decompressor.

Compression ratio: Of the compression ratio achieved by ZFP, DE-ZFP achieves a maximum of 96%, and a minimum of 88% with 10^{-6} tolerance, and a maximum of 95%, and a minimum of 80% with 10^{-3} tolerance. ZFP-V achieves similar maximum and minimum compression ratios (no explicit numbers are reported on compression ratio).sZFP however is reported to achieve a significantly lower compression ratio, as low as 22% for NWChem data set.

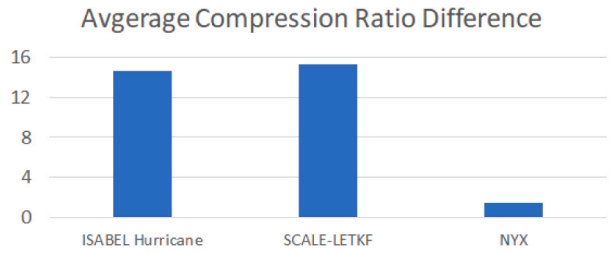


Fig. 11. Ratio of compression ratio increase when using a tolerance of 10^{-3} instead of 10^{-6}.

Table 6
Average throughput comparison.

Data set	ZFP-V Avg. Thr. (GB/s)	DE-ZFP Avg. Thr. (GB/s)
ISABEL Hurricane	1.8	3.48
SCALE-LETKF	2.1	3.40
NYX	1.7	1.61

Resource utilization: DE-ZFP resource utilization for the compressor and the decompressor channels is 14.7%, compared to 27% in ZFP-V. This improvement is due to the fact that DE-ZFP was developed using optimized RTL HDL as opposed to OpenCL used for ZFP-V. sZFP used a different FPGA platform but has generally lower resource utilization because it is designed for 1-D data.

Throughput: Table 6 shows a comparison of the average compressor throughput between DE-ZFP and ZFP-V for single-precision data sets. ZFP-V is a pipelined implementation on a hybrid platform with aggregated communication throughput of more than 22 GB/s between FPGA and CPU. Even though DE-ZFP is not pipelined, uses much smaller tolerance and has a limited interface throughput of 3.64 GB/s, it outperformed ZFP-V for two of the data sets by ~1.7x. Even for the 3rd data set, ZFP-V achieved very modest speedup over DE-ZFP. Comparing decompression throughput was not practical. Because using 10^{-3} instead of 10^{-6} (as in our implementation) yields higher compression ratios as illustrated in Fig. 11, for two of the data sets (15x improvement). This should also increase throughput since higher compression means less data to send back to the host CPU and less data to decode in decompression. Further, using 10^{-3} tolerance will result in more blocks represented as a single zero, and non-zero blocks would have fewer bitplanes. Hence the numbers will heavily depend on how much to decompress. In addition, ZFP-V did not report how many decompression pipelines they are using to decompress chunks. As was mentioned before, sZFP implemented a 1-D double-precision modified version of ZFP on a different platform with communication throughput of 8GB/s. Using 10^{-6} tolerance, sZFP average throughput was ~ 2.1 GB/s/pipeline and 1 GB/s/pipeline for compression and decompression, respectively.

7. Conclusion

An efficient FPGA-based implementation of a modified ZFP compressor and decompressor dubbed DE-ZFP was developed. The proposed implementation uses an optimized dictionary-based encoding instead of embedded encoding to achieve better performance on the FPGA. This leads to a slight degradation in compression efficiency but this degradation is minimized by optimizing the number of encoded blocks using smart output logic. The implementation supports two

modes for the compressor, fixed-accuracy and fixed-precision. Furthermore, we used PCI-e with RIFFA framework to interface the FPGA with the host and detailed the interconnecting modules. We also discussed the implementation compression ratio, throughput, resource utilization and power performance on Arria 10 GX FPGA. We compared the results to the software compressor where applicable. We also compared our work to other FPGA-based lossy implementations. DE-ZFP, if pipelined, has the potential to reach higher throughput, theoretically 59.6 GB/s, but it is capped by the maximum interface speed (3.64 GB/s).

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is supported by King Fahd University of Petroleum & Minerals under project No. DF191015. The authors acknowledge the help of Dr. Peter Lindstrom. Hurricane Isabel data produced by the Weather Research and Forecast model, courtesy of NCAR and the U.S. National Science Foundation. Data collection is a courtesy of DOE NNSA ECP project and the ECP CODAR project.

References

[1] P.J. Gleckler, P.J. Durack, R.J. Stouffer, G.C. Johnson, C.E. Forest, Industrial-era global ocean heat uptake doubles in recent decades, Nature Clim. Change 6 (4) (2016) 394–398, http://dx.doi.org/10.1038/nclimate2915.

[2] Y. Collet, T. Matsuoka, LZ4 - Extremely fast compression. URL: https://lz4.github.io/lz4/.

[3] J. loup Gailly, M. Adler, The gzip home page. URL: http://www.gzip.org/.

[4] M. Isenburg, P. Lindstrom, J. Snoeyink, Lossless compression of predicted floating-point geometry, Comput. Aided Des. 37 (8) (2005) 869–877, http://dx.doi.org/10.1016/j.cad.2004.09.015.

[5] P. Lindstrom, M. Isenburg, Fast and efficient compression of floating-point data, IEEE Trans. Vis. Comput. Graphics 12 (5) (2006) 1245–1250, http://dx.doi.org/10.1109/TVCG.2006.143.

[6] P. Ratanaworabhan, J. Ke, M. Burtscher, Fast lossless compression of scientific floating-point data, in: Proceedings of the Data Compression Conference, in: DCC '06, IEEE Computer Society, Washington, DC, USA, 2006, pp. 133–142, http://dx.doi.org/10.1109/DCC.2006.35.

[7] S. Lakshminarasimhan, N. Shah, S. Ethier, S. Klasky, R. Latham, R. Ross, N.F. Samatova, Compressing the incompressible with ISABELA: 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.

[8] S. Di, F. Cappello, Fast error-bounded lossy HPC data compression with SZ, in: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2016, http://dx.doi.org/10.1109/ipdps.2016.11.

[9] 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), 2017, pp. 1129–1139, http://dx.doi.org/10.1109/ipdps.2017.115.

[10] N. Sasaki, K. Sato, T. Endo, S. Matsuoka, Exploration of lossy compression for application-level checkpoint/restart, in: 2015 IEEE International Parallel and Distributed Processing Symposium, 2015, pp. 914–922, http://dx.doi.org/10.1109/ipdps.2015.67.

[11] P. Lindstrom, ZFP 0.5.5 documentation. URL: https://zfp.readthedocs.io/.

[12] P. Lindstrom, Fixed-rate compressed floating-point arrays, IEEE Trans. Vis. Comput. Graphics 20 (12) (2014) 2674–2683.

[13] Lawrence Livermore National Laboratory, 2019, URL: https://computing.llnl.gov/projects/floating-point-compression/zfp-compression-ratio-and-quality.

[14] A. Shawahna, S.M. Sait, A. El-Maleh, FPGA-based accelerators of deep learning networks for learning and classification: A review, IEEE Access 7 (2019) 7823–7859.

[15] 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.

[16] 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 advpub (2017) http://dx.doi.org/10.1587/elex.14.20170399.

[17] 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.

[18] J. Ouyang, H. Luo, Z. Wang, J. Tian, C. Liu, K. Sheng, FPGA implementation of GZIP compression and decompression for IDC services, in: 2010 International Conference on Field-Programmable Technology, 2010, pp. 265–268, http://dx.doi.org/10.1109/FPT.2010.5681489.

[19] A. Martin, D. Jamsek, K. Agarwal, FPGA-based application acceleration: Case study with GZIP compression/decompression streaming engine, in: International Conference on Computer-Aided Design (ICCAD), 2013.

[20] M.S. Abdelfattah, A. Hagiescu, 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, in: IWOCL '14, Association for Computing Machinery, New York, NY, USA, 2014, pp. 1–9, http://dx.doi.org/10.1145/2664666.2664670.

[21] 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, http://dx.doi.org/10.1109/FCCM.2019.00042.

[22] 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, in: PPoPP '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 74–88, http://dx.doi.org/10.1145/3332466.3374525.

[23] Intel Corporation, 2020, URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf.

[24] 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, http://dx.doi.org/10.1109/ICFPT47387.2019.00022.

[25] G. Sun, S. Kang, S.-W. Jun, BurstZ: A bandwidth-efficient scientific computing accelerator platform for large-scale data, in: Proceedings of the 34th ACM International Conference on Supercomputing, in: ICS '20, Association for Computing Machinery, New York, NY, USA, 2020, http://dx.doi.org/10.1145/3392717.3392746.

[26] A.W. Wegener, Block floating point compression of signal data, 2012, US Patent US20110099295A1. URL: https://patents.google.com/patent/US8301803B2/en.

[27] I. Daubechies, W. Sweldens, Factoring wavelet transforms into lifting steps, J. Fourier Anal. Appl. 4 (3) (1998) 247–269, http://dx.doi.org/10.1007/BF02476026.

[28] R. Wang, Introduction to Orthogonal Transforms: With Applications in Data Processing and Analysis, Cambridge University Press, 2012, http://dx.doi.org/10.1017/CBO9781139015158.

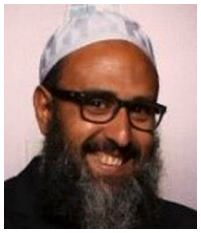
[29] N. Podhorszki, K. Mehta, S. Klasky, L. Wan, M. Wolf, Scientific data reduction benchmarks. URL: https://sdrbench.github.io/.

[30] M. Jacobsen, D. Richmond, M. Hogains, R. Kastner, RIFFA 2.1: A reusable integration framework for FPGA accelerators, ACM Trans. Reconfig. Technol. Syst. 8 (4) (2015) 22:1–22:23, http://dx.doi.org/10.1145/2815631, URL: http://doi.acm.org/10.1145/2815631.

[31] A. El-Maleh, S. AlSaleh, M.E.S. Elrabaa, A bit addressable register with variable write/read data widths, Arab. J. Sci. Eng. (2021).

Mahmoud Habboush received his B.SC. in Computer Engineering at King Fahd University of Petroleum and Minerals (KFUPM), in 2020. In his senior year, he started undergraduate research under the supervision of Profs. Aiman H. El-Maleh & Muhammad E. S. Elrabaa. His research interests include FPGA-based hardware accelerators, computer vision and deep neural networks.

Aiman H. El-Maleh received 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. His research interests include synthesis, testing, and verification of digital systems, defect and soft-error tolerance design, design automation, computer vision and deep learning. He received the Best Paper Award for the most outstanding contribution to the field of test at the 1995 European Design and Test Conference. He holds seven U.S. patents.



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 eight US patents.

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