IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 25, 2020, accepted March 22, 2020, date of publication March 30, 2020, date of current version April 14, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2984191

High-Throughput FPGA-Based Hardware Accelerators for Deflate Compression and Decompression Using High-Level Synthesis

MORGAN LEDWON, (Member, IEEE), BRUCE F. COCKBURN, (Member, IEEE), AND JIE HAN, (Senior Member, IEEE)
Department of Electrical and Computer Engineering, University of Alberta, Edmonton, AB T6G 2R3, Canada

Corresponding author: Morgan Ledwon (ledwon@ualberta.ca)

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) under Project RES0018685 and Project RES0038166.

ABSTRACT The Deflate compression algorithm provides one of the most widely used solutions for lossless data compression. Field-programmable gate arrays (FPGAs) are commonly used to implement hardware accelerators that speed up computation-intensive applications. In this article, FPGA-based accelerators for Deflate compression and decompression are described. These accelerators were specified in C++ and synthesized using Vivado High-Level Synthesis (HLS) for a Xilinx Virtex UltraScale+ series FPGA and a system clock frequency of 250 MHz. The proposed compressor processes data at a fixed input throughput of 4.0 GB/s and achieves a geometric mean compression ratio of 1.92 on the Calgary corpus benchmark files using static Huffman encoding. While not the first compressor synthesized using high-level synthesis, our design achieves a 25% greater throughput and an 11% greater compression ratio than the only other published design that uses Vivado HLS. The proposed decompressor design achieves average input throughputs of 196.61 MB/s and 97.40 MB/s, for statically and dynamically encoded Calgary corpus files, respectively. This is the first published decompressor design that is synthesized using high-level synthesis and provides performance that is comparable to that of the best published designs, having static throughputs 11% higher and dynamic throughputs only 10% lower than the expertly-optimized design sold by Xilinx.

INDEX TERMS Deflate algorithm, lossless compression, LZ77 compression, hardware accelerator, FPGA-based accelerator, high-level synthesis.

I. INTRODUCTION

Data compression enables more efficient utilization of fixed data communication bandwidth and storage space. While some types of data can tolerate the loss of information resulting from lossy compression (e.g., audio, video, image files), many types of data cannot and must be compressed losslessly (e.g., text, source code, financial data). Deflate [1] is an important lossless data compression algorithm that is used in many compressed file formats (e.g., .zip, .gz, .png files) as well as the Hypertext Transfer Protocol (HTTP) [2].

In Deflate compression implementations, there is a trade-off between the compression ratio (i.e., the ratio of the input data size to the size of the compressed output) and the compression throughput. Compressors that offer the most compression tend to be slower than designs that maximize the compression speed. At the system level, the compression process can be accelerated through task-level parallelism by using multiple compressors. Unfortunately, the standard Deflate compressed format hinders parallelism in the decompression process within each task, requiring other methods of acceleration.

Increasingly, the Central Processing Units (CPUs) in conventional computers are enhanced with heterogeneous computing resources such as field-programmable gate arrays (FPGAs). FPGAs allow accelerators to be designed that exploit the structure and parallelism of computation-intensive applications. High-level synthesis is a technology that allows FPGA designs to be specified in a high-level programming language, instead of a hardware description language, with

The associate editor coordinating the review of this manuscript and approving it for publication was Victor Sanchez.

the objective of increasing designer productivity. Using high-level synthesis, however, requires designing at a higher level of abstraction and implies sacrificing a relatively large degree of control over the final design. High-level synthesis software tends to have a steep learning curve to use effectively, which can make creating a design that is both functional and efficient a challenging process.

In this article, we describe two FPGA-based accelerators: one for performing Deflate compression and the other for Deflate decompression. Both accelerator designs were created using high-level synthesis, from C++ source code, and target Xilinx FPGAs using a 250-MHz system clock. In the compressor design, meeting timing objectives at this relatively high frequency likely requires sacrificing some compression ratio (compared to previous designs). Our main design goal was thus to investigate ways to optimize this trade-off and to maximize both the compression throughput and compression ratio. In the design of the decompressor, our main goal was to find sources of exploitable parallelism in order to accelerate the process and maximize the decompression throughput. The main contributions of this article are as follows:

- We propose a compressor design that compresses data at a fixed input throughput of 4.0 GB/s while achieving a geometric mean compression ratio of 1.92 across the widely used Calgary corpus benchmark files [3] using static Huffman encoding. This compressor design provides both higher throughput and higher compression ratios compared to [4], the other published compressor design implemented using Vivado HLS.
- We propose a decompressor design with average input decompression throughputs of 196.61 MB/s and 97.40 MB/s, for statically and dynamically encoded Calgary corpus files, respectively. These values are higher than most other reported decompressors while being achieved for the first time using high-level synthesis.

This article is structured as follows: Section II reviews the Deflate file format and the compression and decompression processes. Sections III-A and III-B survey the published work on Deflate compression and decompression. Section IV describes the compressor design while Section V describes the decompressor. Section VI reports and discusses the performance results on the Calgary corpus and suggests directions for future work. Lastly, Section VII provides a concluding summary.

II. BACKGROUND INFORMATION ON DEFLATE

Deflate is the concatenation of two other compression algorithms: byte-level compression using LZ77 encoding [5] followed by bit-level compression using Huffman encoding [6]. In LZ77 encoding, the input data file is scanned to find repeated strings of bytes. When a match is found with an earlier string of bytes, the repeated string is replaced with a length-distance pair, representing the length of the matched string and the distance it was found earlier within the input data stream, as shown in Fig. 1. In Deflate, matching strings

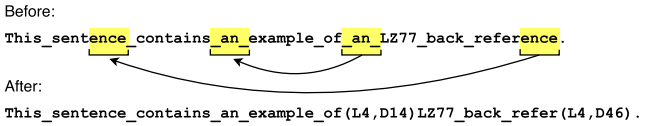

FIGURE 1. An example of LZ77 compression where repeated strings of characters are replaced with length-distance pairs.

can have lengths of 3 to 258 bytes and distances of up to 32,768 bytes back. The string search is typically performed by hashing into a history of previously encountered strings (i.e., a hash table) instead of by exhaustive comparison. To search for similar strings, a string of bytes from the current position in the data is input to a hash function and the resulting hash signature is then looked up in the hash table, which contains the positions of previously hashed strings.

Following LZ77 encoding, the data, now composed of unmatched literal bytes and length-distance pairs, is split into blocks of arbitrary size and each block is Huffman-encoded in one of two ways: using static (fixed) codes or dynamic codes. In static encoding, the literals and length-distance pairs, which shall be referred to as symbols, are encoded using pre-defined code tables given in the Deflate standard [1]. In dynamic encoding, custom Huffman code tables are created for each block based on the frequency of the literals and length-distance pairs within that block. Compressing a block using dynamic codes takes longer but allows a greater compression ratio to be achieved.

During the Huffman encoding process, two code tables are used: one for encoding literals and lengths, and the other for encoding distances [1]. A special code for the end-of-block (EOB) symbol, which is used to indicate the end of a Deflate block, is included in the literal/length table. The static literal/length codes are from 7 to 9 bits long while the distance codes are fixed-length 5-bit codes. For dynamic encoding, both the literal/length and distance codes can be from 1 to 15 bits long. In both static and dynamic encoding, the length and distance codes may be followed by additional bits that correspond to an offset value that is added to the decoded base length or distance value. Length codes can be followed by 0 to 5 extra bits and distances can be followed by 0 to 13 extra bits. Besides static and dynamic Deflate blocks, there is a third type of block called a stored block. Stored blocks contain uncompressed data and are typically used when the data cannot be usefully compressed (e.g., already compressed data). Every Deflate block begins with a block header that identifies the type of the block.

III. LITERATURE REVIEW

A. COMPRESSION-RELATED WORK

State-of-the-art FPGA-based compression accelerators have a fixed-throughput pipeline that performs LZ77 string matching on multiple candidate strings in parallel [4], [7]–[10]. At every clock cycle, potential matches are looked up and compared to multiple substrings within a sliding window, with only the best matches found being used. The number of bytes read and the number of substrings matched are

determined by a key parameter called the Parallel Window Size (PWS). This process differs from how software compression is typically performed, in which multiple attempts may be made searching for potential matches for a single string in order to find the best one. As a result, the average compression ratios achieved by fixed-throughput FPGA-based designs are lower, around 2.00, compared to software compressors, which can achieve ratios of around 3.00. However, because only one clock cycle is spent searching for matches, the FPGA-based designs can achieve constant throughput values on the order of multiple GB/s, much greater than software compressors, for which the highest published throughputs are around 340 MB/s [11]. Most FPGA-based compressors also use only static Huffman encoding. This allows the literals and length-distance pairs to be Huffman encoded immediately following LZ77 encoding without the additional work of constructing dynamic Huffman code tables.

Reference [7] was one of the first proposed fixed-throughput FPGA-based compressor designs but this paper is unfortunately no longer accessible. The authors of [8] propose a similar design that was implemented using OpenCL [12] and that differs from other designs in that it utilizes a fully-connected hash dictionary. In this design, each substring in the sliding window has uncontested write access to its own hash dictionary and each hash dictionary is duplicated enough times so that it can be read for every substring. This dictionary architecture ensures that no hash conflicts can occur and, as a result, the compressor from [8] is able to achieve the highest compression ratios of all of the reported FPGA-based compressor designs, having a geometric mean compression ratio of 2.17 over the Calgary corpus [3]. With a clock frequency of 193 MHz and a PWS of 16 bytes, [8] reports an input decompression throughput of 2.84 GB/s.

In [9], a compressor is proposed that is scalable with the PWS value. The trade-off between the PWS value and the resulting compression ratio and area cost is then explored. It was found that as the PWS is increased, the compression ratio increases logarithmically (providing diminishing returns) while the area cost increases quadratically. Using a PWS of 16 bytes seems to be the most efficient trade-off as matches longer than 16 bytes are relatively rare in most types of data [9]. Unique to this design is the use of a double-clocked hash bank architecture, i.e., the hash banks are run at double the clock frequency compared to the rest of the compressor. This allows each hash bank to serve two requests from hashed substrings instead of only one, reducing the number of dropped substrings due to hash conflicts. For a design with a PWS of 16 bytes, [9] reports an input throughput of 2.80 GB/s and an average compression ratio of 2.05 across the Calgary corpus. When the PWS is scaled up to 32 bytes, the throughput increases to 5.60 GB/s and the average compression ratio increases only slightly to 2.09 at the cost of 2.8× more area.

The authors of [4] describe a compressor design similar to the above designs using Vivado HLS [13]. Unfortunately, however, few design details are provided. Using a 16-byte PWS, they achieved an input throughput of 3.20 GB/s (ignoring interface bottlenecks) and a geometric mean compression ratio of 1.73 on the Calgary corpus.

The authors of [10] used the results from [9] and determined that implementing multiple compression engines in parallel is a more efficient way to increase the system-level compression throughput while increasing the area cost only linearly. Consequently, they focused on implementing multi-core compressor designs. They also made improvements to the hash bank architecture, such as utilizing multiple hash banks chained together to give the hash banks ''depth'' allowing them to store multiple previous matches for each hash signature. Another improvement was to perform the match comparisons at the output of the hash banks to reduce the size of the multiplexers (MUXs) used. Since the match lengths can be from 0 to 16 bytes long, 5-bit MUXs can be used to route the resulting best match length back to each substring instead of using 16-byte-wide MUXs to route the entire strings before performing the comparisons. A single compressor core with a PWS of 16 bytes provides an input throughput of 3.20 GB/s and an average compression ratio of 2.10 on the Calgary corpus.

The authors in [14] focus on designing an LZ77 encoder that has two modes: one that prioritizes maximizing the throughput and the other that prioritizes maximizing the compression ratio. Their LZ77 encoder performs searches for matches and comparisons for a variable amount of time as opposed to the previously described fixed-rate designs, in which all searches and comparisons are done within a single clock cycle. This is similar to how software compressors, like zlib, function. In throughput-priority mode, fewer searches and fewer comparisons are made to save time. Conversely, in compression-ratio-priority mode, more searches and comparisons are done to increase the compression ratio. They also implement ''false history filtering'' in their design (from a previous paper of theirs [15]) in which string-tags are also stored along with string-positions in the hash table. Before performing string comparisons, the tags are checked to see if the strings are remotely similar or were two very different strings that happened to hash to the same signature. In the second case, the comparison does not need to be performed in order to save time since doing more comparisons increases the compression time. With a compression system composed of 16 LZ77 encoders and 4 Huffman encoders, [14] reports an average input throughput of 3.16 GB/s and an average static compression ratio of 2.73 on the Canterbury corpus [16] (in compression-ratio-priority mode). In [9], however, the authors are able to achieve average compression ratios of around 2.70, only 0.03 lower. This suggests that fixed-rate LZ77 encoders may be able to perform string matching almost as well as variable-rate encoders despite having much higher throughputs.

B. DECOMPRESSION-RELATED WORK

Due to the serial nature of the Deflate compressed format, accelerating the decompression process using task-level

parallelism is challenging. Because of the variable length of the Deflate blocks as well as the variable-length Huffman codes contained within, each block must be processed serially, one code at a time, until the EOB code is found. This prevents parallel decoding of the Huffman-encoded blocks. A further complication is that the LZ77 length-distance pairs are allowed to back-reference data from other previous Deflate blocks prior to the current one, which hampers the ability to perform LZ77 decoding on all of the blocks in parallel.

The authors of [17] make several alterations to the Deflate format in order to work around these limitations. Their format includes a file header containing indexes to all of the Huffman blocks; also, length-distance pairs are not allowed to reference other blocks, allowing both the Huffman and LZ77 decoding processes to be performed on all blocks in parallel. By exploiting these format changes, among others, they achieve significantly increased output decompression throughputs of over 13 GB/s using Graphics Processing Units (GPUs).

In [18], [19], and [20], the technique of performing speculative parallelization of the Huffman decoding process is investigated. By scanning ahead to find the EOB code of the current Deflate block, it may be possible to begin Huffman decoding the next block in parallel with the current one. However, the possibility of finding false-positive boundaries makes the process speculative and any speedups that this technique can provide are statistical. Also, the technique only helps to parallelize the Huffman decoding process and not the LZ77 decoding. As a result, the decompression speedups obtained in [18] and [19] are only slightly higher than sequential decompression. The authors of [20] attempt to parallelize the LZ77 decoding using a two-pass method in which length-distance pairs pointing to earlier Deflate blocks are skipped in the first pass and then resolved in the second pass. They also focused on decompressing files containing ASCII data only and exploited that knowledge in a filter when performing speculative parallelization in order to reduce the number of false-positive boundaries encountered. They reported an impressive input throughput of 611 MB/s.

References [21], [22], and [23] propose FPGA-based decompressor designs. In [21], the authors describe a decompressor that performs static Huffman decoding only. When decompressing statically compressed Calgary corpus files, they achieve an average output decompression throughput of 159 MB/s and a maximum of 206 MB/s. In [22], the proposed decompressor achieves a maximum input throughput of 125 MB/s. Unfortunately, the average values or the test files that were used were not disclosed. The authors of [23] propose a new dynamic Huffman decoder design. While the static Huffman codes can be easily decoded using a 512-index lookup table (LUT) for literals/lengths and a 32-index table for distances, the 1 to 15-bit dynamic codes require 32,768-index LUTs, one for literals/lengths and another for distances, to decode in the same way. The design in [23] complicates the decoding process slightly, but allows dynamic codes to be decoded using a 286-index LUT with one index per code. The decoding process involves determining the length of a code by comparing it to the base value of each code length before calculating the address to look up in the table. Their decompressor has a maximum throughput of 300 MB/s (it is unstated whether this is an input or output value) but no average values are given and the test files are not disclosed.

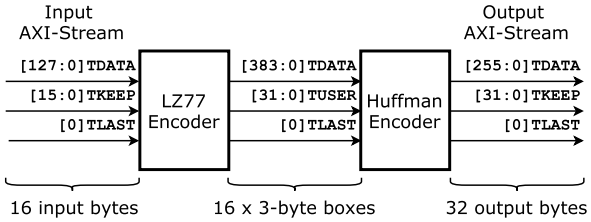

FIGURE 2. Compressor architecture.

The authors of [24] propose an FPGA-based LZ77 decoder design that is capable of processing multiple LZ77 commands (literals and length-distance pairs) at a time. This is done by cyclically partitioning the history buffer (as done in our design as well and explained further in Section V-C) so that multiple memory locations can be accessed simultaneously. Multiple LZ77 commands are read at a time and are broken down into individual read and write operations that are given to each memory partition. Relatively complex control logic is used to ensure that memory access conflicts (when multiple read and write operations access the same memory partition) and data dependencies (when a length-distance pair references data that has not been written yet) are resolved properly. Their design is capable of performing LZ77 decoding with output throughputs ranging from 4.40 to 7.20 GB/s on various files.

A proprietary decompressor intellectual property (IP) core created by Cast Inc. [25] can be purchased on Xilinx's website. The core is reported to have an average dynamic output throughput of 375 MB/s and an average static output throughput of 495 MB/s, though the test files and methods used to achieve these numbers are not disclosed.

IV. PROPOSED COMPRESSOR DESIGN

The proposed compressor design comprises two cores: an LZ77 encoder and a Huffman encoder, as shown in Fig. 2. Both cores are specified in C++ and synthesized in Vivado HLS for a target clock frequency of 250 MHz. The cores are connected using standard AXI-Stream interfaces [26]. As with previous designs, a PWS of 16 bytes is used and only static Huffman encoding is performed, providing a fixed input compression rate of 16 bytes per clock cycle. Therefore, the design of the Huffman encoder is simplified and most of the design effort was spent on the larger LZ77 encoder, which has greater impact on the compression ratio and throughput.

A. LZ77 ENCODER

With the compression rate held constant, as well as the PWS value and static Huffman encoding decided, there are still two

other factors that impact the compression ratio: the hash function and the hash bank architecture (i.e., the number and size of hash banks). Our original plan was to utilize the fully connected dictionary architecture from [8], thus avoiding hash conflicts, but this design was found to require too many Block Random-Access Memories (BRAMs). We instead used the hash bank architecture from [10], comprising 32 hash banks, each with a depth of 3 (providing the capacity to store 3 previous matches at a time), and 512 indexes.[1] Consequently, it became important to use a hash function that minimizes the probability of hash conflicts occurring (when two substrings attempt to access the same bank) as this strongly impacts the compression ratio when matches are dropped due to conflicts. When resolving conflicts, substrings from later positions in the window are given priority over earlier ones to coincide with the last-fit match selection heuristic used later.

With 32 hash banks containing 512 indexes each, we require a 14-bit hash function: the upper 5 bits address the banks and the lower 9 bits address the indexes within a bank. Out of all of the previous works described in Section III-A, only [8] describes the hash function used. We used their 10-bit hash function, which performs left-shift and XOR operations on 4 bytes, as a starting point in our search for an improved 14-bit function. We experimented with other operations to find a function that minimized the amount of hash conflicts and, therefore, provided the highest compression ratios. These operations included performing shifts, rotations, multiplication, addition, and XOR on various amounts of bytes. In total, we empirically evaluated about 100 different hash functions before settling on the one that was found to provide the highest geometric mean compression ratio across the Calgary corpus. This new hash function, which takes in 5 substring bytes as inputs, is shown in Algorithm 1.[2]

Algorithm 1:

Input: $curr\_window[2 \times PWS]$, array of 8-bit ints
Output: $hash[PWS]$, array of 14-bit ints

for $i \leftarrow 0$ **to** *PWS-1* **do**
  $hash[i] = (curr\_window[i] \times 31) \oplus (curr\_window[i+1])$;
  $hash[i].rrotate(4)$;
  $hash[i] = (hash[i] \times 3) \oplus curr\_window[i+2]$;
  $hash[i].rrotate(4)$;
  $hash[i] = (hash[i] \times 3) \oplus curr\_window[i+3]$;
  $hash[i].rrotate(4)$;
  $hash[i] = (hash[i] \times 3) \oplus curr\_window[i+4]$;

As done in [10], we perform the match comparisons at the output of the hash banks before multiplexing the best match length result back to the sliding window to minimize both the area and delay. For the match selection stages of the LZ77 encoder, we used the same last-fit heuristic as in [8], in which later potential matches within the window are kept over earlier ones. We also added match-trimming to the match selection: earlier matches that conflict with later matches are trimmed (if possible without trimming matches below 3 bytes) so that both matches can be kept. Following match selection, each location in the sliding window, which we will call boxes, contains an unmatched byte, a length-distance pair (where a match begins), or a matched byte (at locations following a length-distance pair). The boxes are each 3 bytes wide to accommodate length-distance pairs and are identified using a 2-bit TUSER signal. The final synthesized LZ77 encoder core produced by Vivado HLS has 43 pipeline stages.

B. HUFFMAN ENCODER

The Huffman encoder receives PWS=16 boxes from the LZ77 encoder, statically encodes the contents of each using Read-Only Memories (ROMs), and then packs the encoded bits into an output window. Unmatched bytes are replaced with literal codes, length-distance pairs are replaced with length and distance codes (as well as their extra bits), and matched bytes are removed from the data stream. Although programmed to have PWS=16 stages and to encode and pack one box at each pipeline stage, Vivado HLS was able to automatically optimize the pipeline down to only 6 stages. Since it is possible for two consecutive encoded windows to be greater than 256 bits long, a 512-bit wide double buffer and a 256-bit output interface must be used to prevent overflowing or stalling.

V. PROPOSED DECOMPRESSOR DESIGN

Our pipelined decompressor core comprises four Vivado HLS synthesized cores: a Huffman decoder, an LZ77 decoder, a literal stacker, and a byte packer, as shown in Fig. 3. A first-in, first-out (FIFO) memory is placed in between the two decoders to alleviate stalling. An initial version of our decompressor was presented in [27]. Since then, improvements were made to the design of the Huffman decoder, as described in Section V-A. As with the compressor, each core is synthesized for a 250-MHz clock and uses AXI-Stream interfaces.

A. HUFFMAN DECODER

The Huffman decoder reads the Huffman encoded data from its input stream into a buffer called the accumulator. The bits in the accumulator are scanned and decoded to convert them back into literals and length-distance pairs. The first version of our Huffman decoder could decode a static literal in 3 clock cycles, a static length-distance pair in 4 cycles, a dynamic literal in 4 cycles, and a dynamic length-distance pair in 7 cycles. When decoding a length-distance pair (static or dynamic), two consecutive table lookups need to be performed: the first, to decode the length code and determine the number of extra length bits, and the second, to decode the distance code that follows. To reduce the delay caused

[1] When storing 128-bit (16-byte) words, at least four BRAM18K modules are required. These four BRAM18Ks are fully utilized when storing 512 128-bit words.

[2] In this algorithm, the '*rrotate*(4)' operation rotates the bits of the hash function four positions to the right.

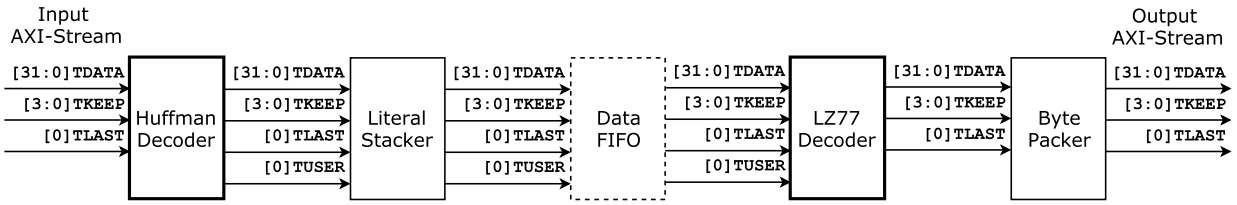

FIGURE 3. Decompressor architecture.

by two consecutive table lookups, we attempted to perform a preemptive lookup of the distance in parallel with the length. Since a static length code can be 7 or 8 bits long, which can then be followed by 0 to 5 extra length bits, there are 12 different possibilities. Of these 12 possibilities, there are actually only 7 different positions where the beginning of the distance code may occur (e.g., an 8-bit length followed by 2 extra bits and a 7-bit length followed by 3 extra bits will both have the same distance starting location). Since the static distance codes are all 5 bits long, we only need to perform 7 parallel distance code lookups alongside the length decoding. Once the length and number of extra length bits is determined, the correctly decoded distance is then selected. By doing this, along with making some other improvements to the static literal decoding, we reduced the time to decode both static literals and length-distance pairs to 2 clock cycles.

In dynamic decoding, the length codes may be 1 to 15 bits long and are followed by 0 to 5 extra bits, which gives 20 possible locations where the distance code may start. Since the distance codes may also be 1 to 15 bits long, for each starting position there are 16 possible positions where a distance code may end. However, since we are using the two-step dynamic decoding method from [23], we can perform the code base value comparison on all 20 distance code possibilities and find code lengths for all of them. While performing the length code lookup, we can then perform 20 parallel distance code lookups at the same time in a duplicated dynamic distance code table memory. As with static decoding, once the length is decoded and the number of extra length bits is known, the correctly decoded distance is selected. By doing this, we reduced the dynamic length-distance pair decoding time to 4 clock cycles. The dynamic literal decoding time remains unchanged at 4 cycles.

B. LITERAL STACKER

The purpose of the literal stacker module is to collect and pack together consecutive literal outputs from the Huffman decoder. Since the AXI-Stream data width is 4 bytes wide, up to 4 consecutive transfers containing literal bytes can be combined into a single transfer. This helps to utilize space in the FIFO more efficiently and also helps the LZ77 decoder to catch up on backlogs of data, which inevitably form in the FIFO while the LZ77 decoder is processing a length-distance pair.

C. LZ77 DECODER

The LZ77 decoder uses a circular buffer to record the last 32,768 bytes of data, which is the largest allowed distance that can be referenced. As literals are received, they are recorded in the buffer and written to the output stream. When a length-distance pair is encountered, the LZ77 encoder stops reading data from its input while it copies the string of bytes pointed to by the length-distance pair. The circular buffer in our design is cyclically partitioned into 4 separate dual-port BRAMs, allowing up to 4 literal bytes to be written to the circular buffer at once or up to 4 bytes of a length-distance pair to be copied at every clock cycle.

D. BYTE PACKER

The byte packer module is required because the LZ77 decoder can write multiple bytes per cycle. When fewer than 4 bytes are output, an AXI-Stream transfer with empty ''null'' bytes is created. To ensure a gapless data stream at the decompressor output, these null bytes must be removed. The byte packer retains transfers that contain fewer than 4 bytes and combines consecutive transfers so that only full 4-byte transfers are output (until the last transfer in the stream).

VI. PERFORMANCE RESULTS AND DISCUSSION

A. COMPRESSOR RESULTS

Our compressor core was implemented on a Xilinx XCVU3P-FFVC1517 Virtex UltraScale+ FPGA [28]. With a PWS of 16 bytes and a clock frequency of 250 MHz, the compressor has an input throughput of 4.0 GB/s. As mentioned in Section IV-A, the hash bank architecture comprises 32 banks, with 3 depth levels and 512 indexes per bank. The resulting area utilization of the compressor is shown in Table 1. The compressor core was tested using the Calgary corpus benchmark files [3] and the resulting compression ratios are shown in Table 2. The output of the compressor was verified by decompressing the files and comparing them to their originals. Our compressor achieves a geometric mean compression ratio of 1.92 across the Calgary corpus. Compared to the previous designs, shown in Table 3, this ratio is slightly lower than all other designs except [4]. Note that the results shown in this table are for single compressor core performance.

As mentioned in Section IV-A, we empirically evaluated about 100 different hash functions to find the one that provided the greatest geometric mean compression ratio across the Calgary corpus for the hash bank architecture that

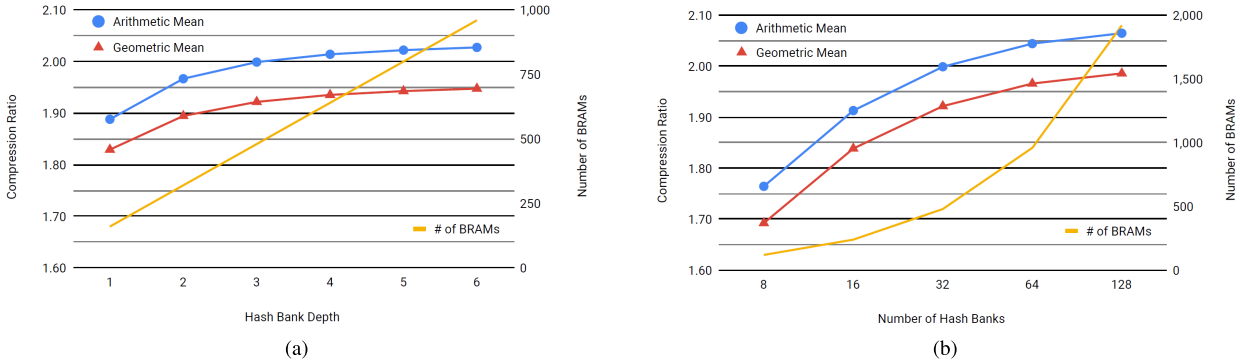

FIGURE 4. Effect of the hash bank architecture on the compression ratio and memory cost. a) Varying the hash bank depth with 32 hash banks and 16,384 total indexes. b) Varying the number of hash banks with 512 indexes per bank and 3 depth levels.

TABLE 1. FPGA resource utilization of the compressor.

	LUTs	FFs	BRAM Tiles
LZ77 Encoder	62,227 (15.79%)	46,225 (5.86%)	240 (33.33%)
Huffman Encoder	6,887 (1.75%)	3,554 (0.45%)	20.5 (2.85%)
Total	69,114 (17.54%)	49,779 (6.31%)	260.5 (36.18%)

TABLE 2. Compression results on the calgary corpus.

File	Uncompressed Size (bytes)	Compressed Size (bytes)	Compression Ratio
bib	111,261	56,450	1.97
book1	768,771	482,048	1.59
book2	610,856	325,400	1.88
geo	102,400	96,448	1.06
news	377,109	218,688	1.72
obj1	21,504	14,093	1.53
obj2	246,814	129,327	1.91
paper1	53,161	28,636	1.86
paper2	82,199	45,891	1.79
pic	513,216	137,428	3.73
progc	39,611	20,556	1.93
progl	71,646	30,510	2.35
progp	49,379	21,534	2.29
trans	93,695	39,795	2.35
		Arithmetic Mean	2.00
		Geometric Mean	1.92

was used. Across all of the different hash functions, most were able to provide geometric mean compression ratios of around 1.90 and no hash function was significantly better than the rest. Some of the hash functions tested, however, provided very poor hash bank distributions (i.e., they were prone to causing hash conflicts) and had poor compression ratios as a result. Since the Calgary corpus contains files of many different kinds of data, the chosen hash function is one that should work well for the compression of a wide variety of data types. If desired, a hash function could be developed that is optimized for a specific type of data (e.g., text) in order to even out the bank distribution when hashing and therefore provide better compression ratios for that type of data.

To investigate the effects that the hash bank architecture has on the compression ratio, a design space exploration on the various parameters was performed. In Vivado HLS, our hash bank architecture is specified as a three-dimensional array in the C++ code with the three dimensions being the number of hash banks, the size of each hash bank, and the depth of each hash bank. These three dimensions can be easily adjusted allowing the design to be re-synthesized and re-evaluated with respect to performance and hardware cost. Fig. 4 shows the effect on the compression ratio and the architecture cost in BRAMs when varying the hash bank depth and the number of hash banks. Fig. 4a shows that the hash bank depth can be scaled up with a linear increase in BRAM cost, while increasing the compression ratio only slightly. A much larger increase in the compression ratio can be obtained by increasing the number of hash banks (and the total number of indexes) but at an exponentially increasing BRAM cost, as shown in Fig. 4b. Increasing the number of hash banks or hash bank depth, however, is likely to require decreasing the clock frequency in order to accommodate the longer wiring as the current architecture proved difficult to meet timing at 250 MHz due to relatively long routing delays between the BRAMs. This illustrates the trade-off between the compression ratio and the throughput in the compressor design.

In the LZ77 encoder, we added match-trimming to increase the number of matches kept during the match selection stages. The addition of match-trimming increased the geometric mean compression ratio by 5.49% while adding 3 pipeline stages, increasing the total LUT usage by 3% and the total flip-flop (FF) usage by 1%. Compared to the other methods for increasing the compression ratio, this is a worthwhile trade-off. As mentioned in Section IV-A, the match selection stages use a last-fit selection heuristic, where later matches are chosen over earlier matches. Using a more complex match selection algorithm (e.g., one that considers the lengths of all of the matches in order to maximize the number of matched bytes in a window) at the cost of additional area or delay might yield more efficient match selection results and achieve higher compression ratios.

Performing dynamic Huffman encoding may be another way to significantly improve the compression ratio but this

TABLE 3. FPGA-based deflate compressor design comparison.

Reference	Language	Clock Frequency	PWS	Input Throughput	Pipeline Depth	Area Utilization	Compression Ratio
[7] (2013)	Verilog	200 MHz	16	3.00 GB/s	17	105,624 ALMs, 1,152 M20Ks	?
[8] (2014)	OpenCL	193 MHz	16	2.84 GB/s	87	110,318 ALMs, 1,792 M20Ks	(Geo.) 2.17
[9] (2015)	SystemVerilog	175 MHz	16	2.80 GB/s	58	39,078 ALMs, ? M20Ks	2.05
			32	5.60 GB/s	91	108,350 ALMs, ? M20Ks	2.09
[4] (2018)	C/C++	200 MHz	16	3.20 GB/s	?	83,583 LUTs, 65,009 FFs, ? BRAM18Ks	(Geo.) 1.73
[10] (2018)	Verilog	200 MHz	16	3.20 GB/s	?	38,297 ALMs, ? M20Ks	2.10
This work	C/C++	250 MHz	16	4.00 GB/s	49	69,114 LUTs, 49,779 FFs, 521 BRAM18Ks	(Geo.) 1.92

Note: Adaptive Logic Modules (ALMs) are the standard logic cells and M20Ks are the standard BRAM modules used by Intel/Altera FPGAs. Table entries occupied by "?" indicates information that is not disclosed.

would likely decrease the throughput because both the Huffman and LZ77 encoders cannot be run simultaneously at all times. This is because dynamic Huffman encoding cannot begin until the dynamic Huffman tables for a block have been created, which cannot start until the entire block has finished LZ77 encoding and the frequency of symbols has been measured. This type of architecture thus also requires elastic buffering between the two encoders to hold the LZ77 encoded data until the block is finished (see [29]).

The most directly comparable compressor is from [4], which was also synthesized from C/C++ using Vivado HLS. Our design has an 11% higher geometric mean compression ratio than [4], while using 17.3% fewer LUTs, 23.4% fewer FFs, and a clock frequency that is 50 MHz higher. Our design achieves higher compression throughput than most previous designs by using a 250-MHz clock. The only exception is the compressor from [9], which has a higher throughput when scaled up to a PWS of 32 bytes. As explained in Section III-A, increasing the compression throughput by increasing the PWS of one compressor is much less efficient for a compressor system than simply incorporating multiple compressors in parallel. This is likely the better option for achieving higher system-level throughputs if the slight drop in compression ratio can be tolerated. Increasing the clock frequency of our design above 250 MHz might be possible, but this would likely require shrinking the hash bank architecture and sacrificing some compression ratio.

B. DECOMPRESSOR RESULTS

Like our compressor, the decompressor was synthesized for a Xilinx XCVU3P-FFVC1517 Virtex UltraScale+ FPGA for a 250-MHz clock. Table 4 shows the resource utilization of the decompressor modules. The decompressor was tested using static and dynamic compressed versions of the Calgary corpus (i.e., containing only static or only dynamic Huffman blocks), which were compressed using zlib with the default compression settings, and the results are shown in Table 5. The output of the decompressor was verified by comparing the files to their originals. Our decompressor has average static and dynamic input throughputs of 196.61 MB/s and 97.40 MB/s, respectively. Output throughputs can be obtained by multiplying the input throughput by the compression ratio, giving us respective average static and dynamic output throughputs of 551.03 MB/s and 336.03 MB/s.

TABLE 4. FPGA resource utilization of the decompressor.

	LUTs	FFs	BRAM Tiles
Huffman Decoder	11,961 (3.04%)	7,497 (0.95%)	2 (0.28%)
Literal Stacker	407 (0.10%)	313 (0.04%)	0
FIFO	75 (0.02%)	103 (0.01%)	5 (0.69%)
LZ77 Decoder	2,239 (0.57%)	820 (0.10%)	8 (1.11%)
Byte Packer	1,009 (0.26%)	389 (0.05%)	0
Total	15,691 (3.98%)	9,122 (1.16%)	15 (2.08%)

The performance of our design is compared to that of other FPGA-based designs in Table 6. Compared to the static-only decompressor in [21], our static output throughput is 3.47 times higher over the Calgary corpus. Compared to the proprietary IP sold by Xilinx [25], our static and dynamic output throughputs are 11% higher and 10% lower, respectively. This is only a loose comparison, however, since the test files used in [25] are unknown. For [25], two areas are reported: the first when configured to perform both static and dynamic decompression and the second when configured to perform static-only decompression. In terms of area, our design uses twice as many LUTs but half as many BRAMs as the design from [25]. Since their design is proprietary we can only speculate, but the larger number of BRAMs may suggest the use of multiple decoders in parallel in their design. Part of the reason that our design uses more LUTs is due to the addition of joint length-distance decoding, which adds 4,500 LUTs, as explained in the next paragraph.

The Huffman decoder was upgraded (with respect to [27]) to be able to perform joint length-distance decoding. This reduced the time to process static literals from 3 to 2 cycles, static length-distance pairs from 4 to 2 cycles, and dynamic length-distance pairs from 7 to 4 cycles. This upgrade increased the size of the Huffman decoder by about 4,500 LUTs and 1,250 FFs, which increased the total FPGA LUT and FF usage by 1.15% and 0.16%, respectively. The average static and dynamic throughput values were thereby increased by 51% and 38%, respectively.

Having the literal stacker increases the compression throughput by anywhere from 0 to 9.7%, depending on the file, while requiring few LUTs and FFs and only adding

TABLE 5. Decompression performance on the calgary corpus.

	Dynamically Compressed Files				Statically Compressed Files			
Compressed File	Compressed Size (bytes)	Compression Ratio	Decompression Time (μs)	Input Throughput (MB/s)	Compressed Size (bytes)	Compression Ratio	Decompression Time (μs)	Input Throughput (MB/s)
bib	35,222	3.16	338.856	103.94	40,931	2.72	188.540	217.09
book1	313,576	2.45	2816.508	111.34	384,953	2.00	1494.016	(max) **257.66**
book2	206,658	2.96	1827.656	(max) **113.07**	243,843	2.51	1073.564	227.13
geo	68,427	1.50	802.436	85.27	80,949	1.26	392.556	206.21
news	144,794	2.60	1473.820	98.24	168,375	2.24	724.556	232.38
obj1	10,311	2.09	141.460	(min) **72.89**	11,138	1.93	68.336	162.99
obj2	81,499	3.03	882.212	92.38	88,949	2.77	455.476	195.29
paper1	18,552	2.87	179.348	103.44	21,670	2.45	98.048	221.01
paper2	29,754	2.76	272.664	109.12	35,499	2.32	155.076	228.91
pic	56,459	9.09	741.848	76.11	67,529	7.60	618.124	(min) **109.25**
progc	13,337	2.97	137.432	97.04	15,365	2.58	75.636	203.14
progl	16,249	4.41	157.108	103.43	18,603	3.85	108.496	171.46
progp	11,222	4.40	114.304	98.18	12,771	3.87	77.764	164.23
trans	19,039	4.92	192.120	99.10	21,424	4.37	137.516	155.79
			Average	97.40			Average	196.61

TABLE 6. FPGA-based decompressor design comparison.

Reference	Area Utilization	Throughput
[21] (2007)	387 LUTs, 128 FFs, 18 RAMB16s	158.64 MB/s (avg., output) on static Calgary corpus
[22] (2009)	20,596 LUTs, 22 BRAM18Ks	125 MB/s (max, input) on unknown test files
[23] (2010)	Not disclosed	300 MB/s (max) on unknown test files
[25] (2016)	8,250 LUTs, 58 BRAM18Ks	375 MB/s (avg., output) on unknown dynamic test files
	5,392 LUTs, 21 BRAM18Ks	495 MB/s (avg., output) on unknown static test files
This work	15,691 LUTs, 9,122 FFs, 30 BRAM18Ks	97.40 MB/s (avg., input), 336.03 MB/s (avg., output) on dynamic Calgary corpus
		196.61 MB/s (avg., input), 551.03 MB/s (avg., output) on static Calgary corpus

two stages to the decompressor pipeline. The largest speedup of 9.7% was obtained when decompressing the static "pic" file. Files with larger compression ratios tend to have more and longer length-distance pairs, placing more stress on the LZ77 decoder, which causes the decompressor to benefit more from the literal stacker. Static files appear to benefit more than dynamic files because dynamic files must have their dynamic code tables reconstructed for every block, giving the LZ77 decoder time to catch up.

The performance of our Huffman decoder could possibly be improved further. The dynamic decoding pipeline might be reduced to 3 cycles if an optimized design were implemented using a hardware description language. The latency of the static decoding portion, however, would be difficult to reduce down to 1 cycle. To achieve this, the decoder would have to be pipelined with an initiation interval of 1 cycle, which would likely require substantially lengthening the clock period due to the dependence on the accumulator.

The LZ77 decoder could be improved by partitioning the circular buffer memory further, allowing more bytes of a back-referenced string to be copied at the same time. A decoder design capable of processing multiple LZ77 commands could also be used, as done in [24]. Generally, though, the current LZ77 decoder design is able to keep up with the Huffman decoder when performing dynamic decoding, so these techniques would not substantially increase the decompression throughput except for static compressed files or highly compressed files, like "pic", which place more stress on the LZ77 decoder. As reported in [9], string matches longer than 16 bytes are rare for most files. Consequently, increasing the LZ77 decoding width above 16 bytes (i.e., having more than 16 partitions) is not likely to be worthwhile.

VII. CONCLUSION

In this article we proposed two new FPGA-based accelerators for both Deflate compression and decompression. These accelerators were specified at a high level in C++ and synthesized using Vivado HLS for a Xilinx FPGA. Our compressor adopted the same 16-byte-per-cycle high-throughput design used in [4], [7]–[10]. We explored ways in which the compression ratio can be improved while also operating with a 250-MHz clock. This included varying the hash function as well as exploring the trade-off between the compression ratio and BRAM cost when scaling the hash bank architecture. Though our geometric mean compression ratio of 1.92 on the Calgary corpus was slightly lower than for most other hardware compressors, we achieved a working design with a 250-MHz clock and a PWS of 16 bytes, giving a constant compression throughput of 4.0 GB/s, which is greater than most other compressor cores. Compared to the other compressor design that was implemented using Vivado HLS, [4], our design has both a higher input throughput (800 MB/s greater) and a higher compression ratio (11% greater), while also occupying less chip area (with 17.3% fewer LUTs and 23.4% fewer FFs).

Our new decompressor design made improvements to our earlier design [27] to substantially improve the throughput. The improved Huffman decoder performs joint length-distance decoding in which the distance code is preemptively decoded at the same time as the length to reduce the overall decoding time for both static and dynamic codes. This allowed us to achieve average input decompression throughputs of 196.61 MB/s and 97.40 MB/s for static and dynamic compressed files, respectively. This gives us output

throughput values that are 11% higher (static) and 10% lower (dynamic) when compared to the proprietary IP core sold by Xilinx [25]. This is only a rough comparison, however, since the test files used in [25] were not disclosed.

The performance of our compressor and decompressor designs are competitive with other works, despite being implemented using high-level synthesis. The benefits of using high-level synthesis to implement our designs did not come easily, however. The lack of control over the details of the synthesized design sometimes made it difficult to create designs that functioned correctly while still performing well. Small changes to the C/C++ source code can change the resulting synthesized design in large ways, meaning that considerable experimentation was required when developing our code to find the syntax that generated the best results. More details on our experience using Vivado HLS can be found in [30]. Despite these challenges, Vivado HLS proved to be a powerful tool capable of producing efficient, high-performance Deflate compressors and decompressors.

ACKNOWLEDGMENT

Design tools were provided through CMC Microsystems.

REFERENCES

[1] L. P. Deutsch, *DEFLATE Compressed Data Format Specification Version 1.3*, document RFC 1951, 1996. [Online]. Available: https://www.w3.org/Graphics/PNG/RFC-1951
[2] D. Salomon, *Data Compression: The Complete Reference*, 4th ed. London, U.K.: Springer, 2006.
[3] T. Bell, I. H. Witten, and J. G. Cleary, "Modeling for text compression," *ACM Comput. Surv.*, vol. 21, no. 4, pp. 557–591, 1989.
[4] J. Cong, Z. Fang, M. Huang, L. Wang, and D. Wu, "CPU-FPGA coscheduling for big data applications," *IEEE Des. Test. IEEE Des. Test. Comput.*, vol. 35, no. 1, pp. 16–22, Feb. 2018.
[5] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inf. Theory*, vol. 23, no. 3, pp. 337–343, May 1977.
[6] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proc. IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.
[7] A. Martin, D. Jamsek, and K. Agarawal, "FPGA-based application acceleration: Case study with gzip compression/decompression streaming engine," *ICCAD Special Session C*, vol. 7, p. 2013, Nov. 2013.
[8] M. S. Abdelfattah, A. Hagiescu, and D. Singh, "Gzip on a chip: High performance lossless data compression on FPGAs using OpenCL," in *Proc. Int. Workshop OpenCL (IWOCL)*. New York, NY, USA: ACM, 2014, pp. 4–9.
[9] J. Fowers, J.-Y. Kim, D. Burger, and S. Hauck, "A scalable high-bandwidth architecture for lossless compression on FPGAs," in *Proc. IEEE 23rd Annu. Int. Symp. Field-Program. Custom Comput. Mach.*, May 2015, pp. 52–59.
[10] W. Qiao, J. Du, Z. Fang, M. Lo, M.-C.-F. Chang, and J. Cong, "High-throughput lossless compression on tightly coupled CPU-FPGA platforms," in *Proc. IEEE 26th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr. 2018, pp. 37–44.
[11] V. Gopal, J. Guilford, W. Feghali, E. Ozturk, and G. Wolrich, "High performance DEFLATE compression on Intel architecture processors," Intel Corp., Santa Clara, CA, USA, White Paper 326352-001, 2011. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-deflate-compression-paper.pdf
[12] Khronos Group. (2019). *OpenCL Overview*. [Online]. Available: https://www.khronos.org/opencl/
[13] Xilinx Inc. (2019). *Vivado High-Level Synthesis*. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html
[14] S. Choi, Y. Kim, D. Lee, S. Lee, K. Park, Y. H. Song, and Y. H. Song, "Design of FPGA-based LZ77 compressor with runtime configurable compression ratio and throughput," *IEEE Access*, vol. 7, pp. 149583–149594, 2019.
[15] S. Choi, Y. Kim, and Y. H. Song, "False history filtering for reducing hardware overhead of FPGA-based LZ77 compressor," *J. Syst. Archit.*, vol. 88, pp. 110–119, Aug. 2018.
[16] R. Arnold and T. Bell, "A corpus for the evaluation of lossless compression algorithms," in *Proc. Data Compress. Conf. (DCC)*, Mar. 1997, pp. 201–210.
[17] E. Sitaridi, R. Mueller, T. Kaldewey, G. Lohman, and K. A. Ross, "Massively-parallel lossless data decompression," in *Proc. 45th Int. Conf. Parallel Process. (ICPP)*, Aug. 2016, pp. 242–247.
[18] H. Jang, C. Kim, and J. W. Lee, "Practical speculative parallelization of variable-length decompression algorithms," *ACM SIGPLAN Notices*, vol. 48, no. 5, p. 55, May 2013.
[19] Z. Wang, Y. Zhao, Y. Liu, Z. Chen, C. Lv, and Y. Li, "A speculative parallel decompression algorithm on apache spark," *J. Supercomput.*, vol. 73, no. 9, pp. 4082–4111, Sep. 2017.
[20] M. Kerbiriou and R. Chikhi, "Parallel decompression of gzip-compressed files and random access to DNA sequences," 2019, *arXiv:1905.07224*. [Online]. Available: http://arxiv.org/abs/1905.07224
[21] J. Lazaro, J. Arias, A. Astarloa, U. Bidarte, and A. Zuloaga, "Decompression dual core for SoPC applications in high speed FPGA," in *Proc. 33rd Annu. Conf. IEEE Ind. Electron. Soc. (IECON)*, Nov. 2007, pp. 738–743.
[22] D. C. Zaretsky, G. Mittal, and P. Banerjee, "Streaming implementation of the ZLIB decoder algorithm on an FPGA," in *Proc. IEEE Int. Symp. Circuits Syst.*, May 2009, pp. 2329–2332.
[23] J. Ouyang, H. Luo, Z. Wang, J. Tian, C. Liu, and K. Sheng, "FPGA implementation of GZIP compression and decompression for IDC services," in *Proc. Int. Conf. Field-Programmable Technol.*, Dec. 2010, pp. 265–268.
[24] J. Fang, J. Chen, J. Lee, Z. Al-Ars, and H. P. Hofstee, "Refine and recycle: A method to increase decompression parallelism," in *Proc. IEEE 30th Int. Conf. Appl.-Specific Syst., Archit. Processors (ASAP)*, Jul. 2019, pp. 272–280.
[25] Xilinx and CAST Inc. (2016). *GUNZIP/ZLIB/Inflate Data Decompression Core*. [Online]. Available: https://www.xilinx.com/products/intellectual-property/1-79drsh.html#overview
[26] *AMBA 4 AXI4-Stream Protocol*, document (ARM IHI 0051A (ID030510)), Arm Holdings, 2010.
[27] M. Ledwon, B. F. Cockburn, and J. Han, "Design and evaluation of an FPGA-based hardware accelerator for deflate data decompression," in *Proc. IEEE Can. Conf. Electr. Comput. Eng. (CCECE)*. Edmonton, AB, Canada: IEEE, May 2019, pp. 195–200.
[28] Xilinx Inc. *Ultrascale Architecture and Product Data Sheet: Overview*. Accessed: Aug. 2019. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf
[29] Y. Kim, S. Choi, J. Jeong, and Y. H. Song, "Data dependency reduction for high-performance FPGA implementation of DEFLATE compression algorithm," *J. Syst. Archit.*, vol. 98, pp. 41–52, Sep. 2019.
[30] M. Ledwon, "Design of FPGA-based accelerators for Deflate compression and decompression using high-level synthesis," M.S. thesis, Dept. Elect. Comput. Eng., Univ. Alberta, Edmonton, AB, Canada, 2019.

MORGAN LEDWON (Member, IEEE) received the B.Sc. degree in electrical engineering and the M.Sc. degree in electrical and computer engineering from the University of Alberta, Edmonton, AB, Canada, in 2016 and 2019, respectively.

His thesis research involved designing FPGA-based accelerators using high-level synthesis for Deflate compression and decompression. He is interested in the applications of hardware acceleration, cloud-computing, machine learning using neural networks, and the Internet of Things (IoT) systems.

BRUCE F. COCKBURN (Member, IEEE) received the B.Sc. degree in engineering physics from Queen's University, Kingston, ON, Canada, in 1981, and the M.Math. and Ph.D. degrees in computer science from the University of Waterloo, Waterloo, in 1985 and 1990, respectively.

He is currently a Professor with the Department of Electrical and Computer Engineering, University of Alberta, Edmonton, AB, Canada. From 1981 to 1983, he was a Test Engineer and a Software Designer at Mitel Corporation in Kanata, ON, Canada. In 2001, he was a Sabbatical Visitor at Agilent Technologies Inc., Santa Clara, CA, USA. From 2014 to 2015, he was also a Sabbatical Visitor with The University of British Columbia, Vancouver, BC, Canada. His research interests include the testing and verification of integrated circuits, application-specific hardware accelerators, applications of high-level synthesis and field-programmable gate arrays, heterogeneous parallel computing, stochastic and approximate computing, and genetic data processing.

Dr. Cockburn is a member of the Association for Computing Machinery and is registered as a Professional Engineer with the Association of Professional Engineers and Geoscientists of Alberta.

JIE HAN (Senior Member, IEEE) received the B.Sc. degree in electronic engineering from Tsinghua University, Beijing, China, in 1999, and the Ph.D. degree from the Delft University of Technology, The Netherlands, in 2004.

He is currently an Associate Professor with the Department of Electrical and Computer Engineering, University of Alberta, Edmonton, AB, Canada. His research interests include approximate computing, stochastic computing, reliability and fault tolerance, nanoelectronic circuits and systems, and novel computational models for nanoscale and biological applications.

Dr. Han was a recipient of the Best Paper Award at the International Symposium on Nanoscale Architectures (NanoArch) 2015 and the Best Paper Nominations at the 25th Great Lakes Symposium on VLSI (GLSVLSI) 2015, NanoArch 2016, and the 19th International Symposium on Quality Electronic Design (ISQED) 2018. He was nominated for the 2006 Christiaan Huygens Prize of Science by the Royal Dutch Academy of Science. He served as the General Chair for GLSVLSI 2017 and the IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT) 2013, and the Technical Program Committee Chair for GLSVLSI 2016, DFT 2012, and the Symposium on Stochastic & Approximate Computing for Signal Processing and Machine Learning, in 2017. He is currently an Associate Editor of the IEEE TRANSACTIONS ON EMERGING TOPICS IN COMPUTING (TETC), and the IEEE TRANSACTIONS ON NANOTECHNOLOGY, the IEEE Circuits and Systems Magazine, and *Microelectronics Reliability* (Elsevier).