Design and Evaluation of an FPGA-based Hardware Accelerator for Deflate Data Decompression

Morgan Ledwon
Electrical and Computer Engineering
University of Alberta
Edmonton, Canada
ledwon@ualberta.ca

Bruce F. Cockburn
Electrical and Computer Engineering
University of Alberta
Edmonton, Canada
cockburn@ualberta.ca

Jie Han
Electrical and Computer Engineering
University of Alberta
Edmonton, Canada
jhan8@ualberta.ca

***Abstract*—Data compression is an important technique for coping with the rapidly increasing volumes of data being transmitted over the Internet. The Deflate lossless data compression standard is used in several popular compressed file formats including the PNG image format and the ZIP and GZIP file formats. Consequently, several implementations of hardware accelerators for Deflate have been proposed. The recent availability of distributed field-programmable gate arrays (FPGAs) in the Internet cloud and the growing demand for decompressing compressed data that is streamed from remote servers make FPGA-based decompression accelerators commercially attractive. This paper describes an efficient implementation of the Deflate decompression algorithm using high-level synthesis from designs, specified in C++, down to optimized implementations for a Xilinx Virtex UltraScale+ class FPGA. When decompressing the Calgary corpus benchmark, our decompressor has average input (output) data throughputs of 70.7 (246.4) and 130.6 (386.6) MB/s for dynamically and statically encoded files, respectively. This performance is comparable to the 375 MB/s output throughput of Xilinx's state-of-the-art proprietary Deflate decompressor design.**

***Index Terms*—Deflate algorithm, lossless compression, hardware accelerator, FPGA-based accelerator**

I. Introduction

Due to the increasing popularity of cloud computing, the Internet must handle the ever growing volumes of data traffic being transmitted. Lossless data compression is used to effectively increase the Internet's communication bandwidth and storage capacity. While it is desirable to optimize both compression and decompression, there are many applications (e.g., multimedia streaming from a server) where decompression is performed much more frequently than compression as data is downloaded repeatedly by many users. This creates the demand for quick and efficient solutions to decompression.

Deflate [1] is one of the most widely used lossless compression algorithms. It underlies the .zip, .gz, and .png file formats, as well as the Hypertext Transfer Protocol (HTTP) [2]. Zlib [3] is an open-source library containing a reference implementation of Deflate, which is based on the unpatented utility gzip [4]. Deflate is generally very good at maximizing the compression ratio (i.e., the ratio of the size of the uncompressed data to the size of the compressed data) for typical data sources. Unfortunately, the output format used in Deflate was not designed to facilitate parallel decompression, which makes decompression difficult to accelerate. Specifically, Deflate decompression is inherently difficult to parallelize because of the serial process used during compression, which is reflected in the compressed data format. Compression can be accelerated by splitting the data stream into multiple chunks that can then be compressed in parallel (for example, in pigz [5], the parallel version of gzip), but parallelization of decompression is not easily done. A simple approach is to alter the Deflate format: if an index of the block locations is included in the compressed file and if back-references between blocks are prohibited, then the blocks within a file could be decompressed in parallel. Another strategy is to structure the data to suit the decompressor, such as by using a constant block size. However, to be Deflate-compliant, a decompressor must handle all of the standard Deflate formats [1].

Internet servers are commonly provided with parallel computing resources that can significantly speed up computationally demanding operations [6], [7]. Graphics processing units (GPUs) are used to accelerate computations that have suitable inherent parallelism. The single-instruction, multiple-data (SIMD) stream architecture of GPUs supplements the SIMD features that are provided in server central processing units (CPUs). Field-programmable gate arrays (FPGAs) provide flexible hardware reconfigurability, which allows accelerators to be designed and tailored to specific algorithms. Unlike GPUs, FPGAs can implement arbitrary algorithms that do not necessarily suit the SIMD model. Application-specific integrated circuits (ASICs) allow the design of customized accelerators that offer the greatest possible performance boost for given algorithms. Unlike ASICs though, FPGAs can be reconfigured, multiple times if necessary, to support a wide variety of algorithms.

In this paper we propose an FPGA-based hardware accelerator for Deflate decompression. Our design, which was developed using high-level synthesis, exploits the capabilities of FPGAs to speed up the decompression process. The main contributions of this paper are the following:

- We describe a Deflate-compliant decompressor design

This work was supported financially by the Natural Sciences and Engineering Research Council of Canada (NSERC) under projects RES0018685 and RES0038166. Design tools were provided through CMC Microsystems. The authors also gratefully acknowledge the technical advice and support of Eidetic Communications Inc., Calgary, AB, Canada.

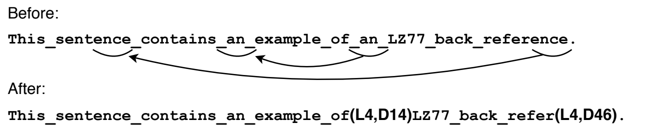
Before:
This_sentence_contains_an_example_of_an_LZ77_back_reference.
After:
This_sentence_contains_an_example_of(L4,D14)LZ77_back_refer(L4,D46).

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

that processes both static and dynamic blocks without making any assumptions about the compressed data.

- We take advantage of the steps of decompression that can be exploited using FPGA technology, for example, when decoding the Huffman codes and by copying multiple bytes of a length-distance pair at a time.
- We demonstrate the capabilities of using high-level synthesis to create a design that is efficient in both speed and area. Our design is able to achieve throughputs comparable to Xilinx's proprietary design [8].

The paper is structured as follows: Section II describes the Deflate file format and reviews related work on accelerating Deflate decompression. Section III describes our pipelined Deflate decompressor core for FPGAs. Section IV evaluates the performance of the decompressor. Lastly, Section V provides conclusions and directions for future work.

II. Background Information and Related Work

Deflate compression takes two steps: LZ77 encoding [9] followed by Huffman encoding [10]. A file to be compressed is first split into blocks of finite size which are then compressed using LZ77 encoding [1]. Strings of repeated characters in the data are replaced with LZ77 codewords in the form of *length-distance pairs*. These pairs indicate the *length* of a string and the *distance* it appeared earlier within the file (see Fig. 1). The Deflate format allows string matches of up to 258 bytes long and distances going back up to 32768 bytes. Note that these distances may cross block boundaries. After LZ77 encoding, the file is a series of data blocks containing literals (unmatched ASCII bytes) and LZ77 codewords, which together are called *symbols*. The symbols within each block are encoded using a Huffman code to compress the file further. The Huffman codewords are commonly called *codes*. One special code, called the *end-of-block (EOB) marker*, indicates the end of each block. Two code tables are used: one for lengths, literals, and the EOB marker, and the other for distances.

Deflate uses two different methods for Huffman encoding, static and dynamic. In *static encoding* the symbols are encoded using constant pre-defined code tables. In *dynamic encoding* the code tables are created during compression: they assign codes to the symbols based on the frequency that they appear in the data block, with the most frequently occurring symbols having the shortest codes. The Huffman code lengths are variable, with dynamic literal/length and distance codes being from 1 to 15 bits long. For static blocks, the literal/length codes are 7 to 9 bits long while the distance codes are 5 bits long. Table I shows the literal/length symbol library and their static codes. The length and distance symbol codes may then be followed by a number of extra bits, from 0 to 5 for lengths and 0 to 13 for distances. The *dynamic code tables* are not included in the compressed file itself, but the code length sequences that are used to recreate the tables are included. At the beginning of each block is a 3-bit header that specifies the type of block and also if it is the last block in the file (see Fig. 2). In a *dynamic block*, the header is followed by the compressed code length sequences for that block. Each dynamic block may be encoded differently. A third type of block, called a *stored block*, contains only uncompressed literals. Stored blocks are useful for encoding incompressible data, for example, already compressed data.

To perform decompression, the two stages of encoding are reversed. First the Huffman codes within each block are decoded and then the LZ77 length-distance pairs are replaced with the original literals. Since the Deflate format does not include indexes that locate the compressed blocks, each block must be decoded serially; the boundary where one block ends and the next begins is unknown until the end of a block has been reached. By altering the Deflate format, the block boundaries could be indexed at the start of the compressed file, allowing each block to be identified and decoded in parallel. Since the length of Huffman codes is variable, each code needs to be decoded one at a time. It is possible, however, to search ahead for the next EOB code. Once a block boundary is found, the next block could begin decoding in parallel. This strategy is called *speculative parallelization* and it has been used to accelerate Huffman decoding [11], [12]. It is speculative because a false boundary can be found if the length and distance codes are followed by extra bits that mimic the EOB code.

The problem of accelerating LZ77 decoding remains. While each data block can be Huffman decoded independently if the block boundaries are known, the same cannot be done with LZ77 decoding because back-references can exist between blocks. An LZ77 distance can point to a string of literals in another block up to 32768 bytes away. A back-reference

TABLE I
Deflate Literal/Length Symbol Library and Static Codes

Symbol Type	Symbol Values	Static Code Lengths	Static Codes
Literal	0 - 143	8 bits	00110000 - 10111111
	144 - 255	9 bits	110010000 - 111111111
End-of-block	256	7 bits	0000000
Length	257 - 279	7 bits	0000001 - 0010111
	280 - 287	8 bits	11000000 - 11000111

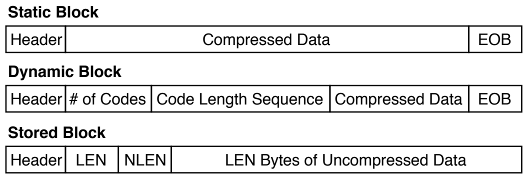

Fig. 2. Deflate block types and their contents (not to scale). A block does not always begin or end on a byte boundary (unless it is Stored).

may also point to another back-reference that has not yet been decoded. This hampers the ability to process each block independently. The authors of [7] solve this problem with an altered algorithm that compresses each block separately (removing inter-block dependencies) and that includes the option of disallowing nested back-references during compression (removing intra-block dependencies). Using this strategy with non-standard fixed-length Huffman encoding, they achieved impressive decompression speeds of over 13 GB/s.

The authors of [12] employed speculative parallelization on a cluster-computing platform. Using six nodes in parallel they achieved decompression throughputs ranging from 18 to 24 MB/s. Although they use speculative parallelization in the Huffman decoder, it is not described how they parallelize the LZ77 decoder. The authors of [11], who also utilize speculative parallelization in the Huffman decoder, rely on the presence of stored (uncompressed) blocks in the data stream to parallelize the LZ77 decoding. Zlib has the tendency to release stored blocks in sizes of at least 16 kB, meaning that two consecutive stored blocks in a stream can be used to fill the 32 kB history buffer of an LZ77 decoder. This allows multiple LZ77 decoders to operate in parallel without encountering back-references to blocks held in other decoders. They discovered, however, that the presence and distribution of stored blocks in a Deflate stream is not so regular and so they were only able to achieve speedups ranging from 1.24 to 1.80 times faster than standard sequential decompression using software.

Accelerating decompression using FPGAs has been reported, but not nearly to the same extent as compression. A variety of FPGA compression accelerators have been developed [13]–[16]. Commercial FPGA decompressor designs, like the one sold by Xilinx [8], report average decompression speeds of 375 MB/s (at the output), but these designs are proprietary. Reference [17] describes a two-core decompressor design for FPGAs similar to ours but their design is limited to static compressed files only. It uses a 512-index lookup table (LUT) to decode the 7 to 9-bit long static Huffman literal/length codes. They falsely assume that the decompressor speed would be the same if the dynamic code tables were created beforehand and given to the accelerator. This would in fact require not just one, but two 32768-index LUTs to be able to decode all possible 1 to 15-bit-long literal/length codes as well as all possible 1 to 15-bit-long distance codes. A solution to this problem is proposed in [18]. Since the Huffman codes of a particular length have consecutive values, their symbols can be stored in one continuous space in memory. If the base address of each code length is recorded (15 base addresses, one for each length from 1 to 15), a code can be located relative to its base address given its length. Thus a dynamic Huffman code can be decoded using a 286-index table for literals/lengths and a 32-index table for distances. We used the same approach, as explained in Section III-A.

The authors of [18] report a maximum decompression speed of about 300 MB/s but they do not give average speeds or specify if the results are for single-file decompression or multi-file decompression using multiple cores. The authors of [19] report a maximum decompression speed of 125 MB/s, however, they do not specify average speeds. The input files being decompressed are not given in [8], [18], or [19]. The closest comparable design to our own is in [17]; however, only results for static compressed files are given. When decompressing the Calgary corpus benchmark files [20], [17] reports a maximum output throughput of 206 MB/s with an average throughput of 159 MB/s. These numbers are thus inflated by about 2 or 3 times (i.e. by the compression ratio) compared to the input throughput.

III. Decompressor Design

Fig. 3 shows a block diagram of the proposed Deflate decompressor. The design comprises a Huffman Decoder, an LZ77 Decoder, and two byte-reordering modules: a literal stacker and a byte packer. All four modules were synthesized from a C++ specification using Vivado HLS [21] for a 250 MHz clock. Vivado HLS is a high-level synthesis tool capable of synthesizing designs that are expressed in C, C++, or SystemC. The first-in first-out (FIFO) memory was synthesized from a configurable block in the Vivado library. All five modules are interconnected with AXI-Stream interfaces [22], which provide unidirectional data flow through the modules. Each AXI-Stream interface comprises a 32-bit TDATA data bus, a 4-bit TKEEP bus, and a 1-bit TLAST signal. The TKEEP signals specify the valid data bytes and the TLAST signal identifies the last transfer. Our TUSER signal flags data packets containing length-distance pairs. The data FIFO can store up to 4096 literal stacks or length-distance pairs before stalling. A compressed file is streamed into the decompressor 4 bytes at a time and the decompressed file can be streamed out 4 bytes at a time.

A. *Huffman Decoder*

The Huffman decoder reads the next bytes from the compressed file and examines the block header. For a stored block, the decoder reads the block length and streams that many bytes from the input to the output. For a static block, the 9-bit literal/length codes are read and decoded using a 512-index static code table stored in a read-only memory (ROM). The 5-bit distance codes that follow address into a 32-index table. The Huffman decoder decodes a static literal in 3 clock cycles and a length-distance pair in 4 cycles. The decoder processes a dynamic block in two phases: first the dynamic code tables are assembled, then the block is decoded using those tables. To assemble the code tables in a dynamic block, the decoder reads the sequence of code lengths, calculates base values and addresses for each length, and assigns consecutive addresses to each code for each length. Code table assembly adds a significant delay when decompressing dynamic blocks compared to static blocks.

As mentioned above, instead of decoding by directly addressing every possible variable-length code, we use the decoder and code table architecture from [18]. Thus we use a 286-index table for literal/length codes and a 32-index table for distance codes. These two tables are implemented using

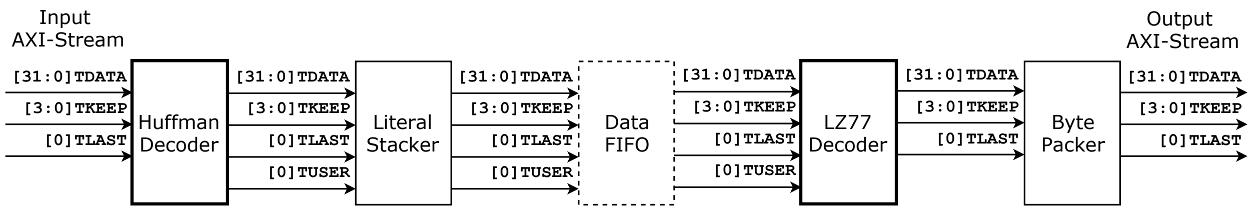

Fig. 3. Pipelined Decompressor Block Diagram.

block random-access memories (BRAMs) in the FPGA. The decoding process has two steps: first the number of bits in the code is determined, then the code address is calculated and the corresponding symbol is looked up. To decipher the code length, which can be from 1 to 15 bits long, the next 15 bits from the input stream are compared in parallel to the 15 different code length base values that were calculated during table construction. A leading-one detector returns the resulting bit length of the code.

Once the code length is known, the address for its symbol in the code table is the base address plus an offset, which is calculated by subtracting the code's base value from the code bits. The code table for literals/lengths is shown in Fig. 4. With the code address calculated, the corresponding symbol can be looked up in the table. If the symbol is a literal, the decoder writes it to the data stream. If the symbol is the EOB, the decoder returns to its initial state and begins decoding another block, if one is available. If the symbol is a length, the corresponding extra bits and base length value are looked up in a symbol table ROM. Any extra bits following the length are read from the stream and added to the base length to get the final length value. The distance code that follows, which can also be 1 to 15 bits long, is decoded in the same way as literals/lengths. Any extra distance bits are read and the actual distance value is calculated. When the final length and distance are known, they are written out to the data stream. The Huffman decoder decodes a dynamic literal in 4 clock cycles and a length-distance pair in 7 cycles. This is the minimum cycle latency that Vivado HLS synthesizes at 250 MHz.

B. Literal Stacker

The LZ77 decoder stops reading from its input stream while it is copying the bytes of a length-distance pair. During this

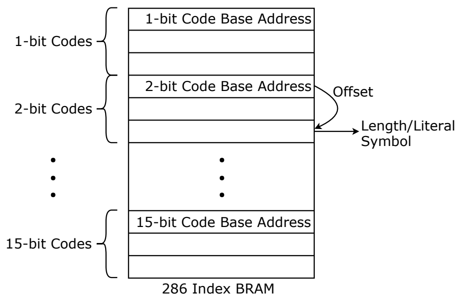

Fig. 4. Literal/Length Code Table.

time data will pile up in the data FIFO. Since the data stream is 4 bytes wide and since the LZ77 decoder can write 4 bytes to its memory in a cycle, we can more rapidly clear out the FIFO by grouping consecutive literals into stacks of 4. As data packets are written out from the Huffman decoder, the literal stacker will examine them and collect any literals into a stack. The presence of a length-distance pair in the stream forces the literal stacker to release any held literals in an incomplete stack. Thus a released literal data packet can contain 1 to 4 literals. TKEEP identifies the number of literals in the stack.

C. LZ77 Decoder

The LZ77 decoder uses the length-distance pairs from the Huffman decoder to recover the original stream of literals. It contains a circular buffer BRAM capable of storing 32768 literal bytes, the maximum distance in the Deflate format. The circular buffer is cyclically partitioned into 4 dual-port BRAMs, so 4 bytes can be read from and written to the buffer every clock cycle. The LZ77 decoder reads in a data packet from the FIFO and uses TUSER to identify literals or length-distance pairs. If the packet contains literals, TKEEP is analyzed and the corresponding number of literals, from 1 to 4, are recorded in the buffer and written to the output stream. If the packet is a length-distance pair, the decoder stops reading from the FIFO and begins copying bytes from the buffer back to the head of the buffer as well as the output stream. The distance specifies how far back in the buffer to read from, and the length specifies how many bytes are to be copied.

A 4-byte wide sliding window is used when reading and writing to the circular buffer, but any number of bytes from 1 to 4 may be read or written. The decoder is designed so that no addressing conflicts can occur, i.e. the same address will never be read and written in the same cycle. There are three possible scenarios that are handled to avoid addressing conflicts. If the distance is from 1 to 4, only a single read is performed on the buffer. The 1 to 4 read literals are stored in 4 registers which are then copied to the buffer head and the output stream. For example, if the distance is 1, only 1 literal is read from the buffer and a copy is stored in all 4 registers. This allows us to write back 4 copies of the same literal every clock cycle until the length is reached. The second scenario occurs when the distance is 5, 6, or 7 as the reading window will collide with the writing window during buffer access. To prevent this, the decoder keeps track of the number of bytes it is allowed to read before the reading window is reset back to its starting position. Any bytes past the distance are not read. For example, if the length is 10 and the distance is 5 and the string to be copied is "ABCDE", on the first iteration "ABCD" will be copied. On the next iteration only "E" is copied. On the next iteration, the sliding window is reset and "ABCD" is copied again. On the final iteration, "E" is written again. The third scenario occurs when the distance is 32768, the size of the circular buffer. In this scenario the read and write pointers will hold the same address. In this case, no writes are performed on the buffer and the literals are just copied from the buffer to the output stream. TKEEP identifies the valid bytes in the 32-bit TDATA output.

D. Byte Packer

The byte packer filters out null bytes from the LZ77 decoder output so that a continuous aligned stream of bytes is output from the decompressor. When a number of bytes less than the sliding window width is written, null bytes are output. The byte packer analyzes the received TKEEP signals and reorganizes the bytes as necessary. The data bytes are aligned in order and held back until four bytes can be output.

IV. Performance Results

The pipelined decompressor design was synthesized for a Xilinx xcvu3p-ffvc1517 Virtex UltraScale+ FPGA for a clock frequency of 250 MHz. The resource utilization of the decompressor is given in Table II. The decompressor was tested using files from the standard Calgary corpus [20] that were compressed using zlib. Using the default zlib compression settings, all of the files were compressed using only dynamic blocks. The corpus files were also compressed using the zlib static block option. For simplicity, the two-byte zlib header and the four-byte Adler-32 checksum footer were removed from the files before decompression. The compressed file sizes, compression ratios, and decompression results appear in Table III. The throughput (at the input) is defined as the compressed file size divided by the total time it took to decompress the file. The decompressor output throughput is the product of the input throughput and the compression ratio.

The file size, compression ratio, and the type of data being decompressed are all factors affecting the throughput. For dynamic files, the throughput has a maximum of 76.12 MB/s on file "book2", a minimum of 62.74 MB/s on "obj1", and an average of 70.73 MB/s. For static files, the throughput has a maximum of 151.64 MB/s on "book1", a minimum of 101.16 MB/s on "pic", and an average of 130.58 MB/s. The average output throughputs are 246.35 and 386.56 MB/s for dynamic and static files, respectively. Compared to the static decompression results in [17], our design is about 2 to 3 times faster. The throughput for the static files is roughly double that of the dynamic files because the Huffman decoder takes about half the time to decode a static code as it does a dynamic code. Also, when processing dynamic blocks, significant time is spent reading the code length sequences and building the dynamic code tables. This impacts the throughput of smaller files more as the fraction of time spent building the tables vs. decoding the codes is greater compared to larger files. The speed-up obtained when decompressing static files suggests that using static compression and sacrificing some compression ratio may often be a good trade-off. This speed-up extends to compression as well; the compression accelerators in [15] and [16] only perform static Huffman encoding in order to speed up compression. As shown in Table II, the resource utilization of the decompressor design is quite modest. Multiple decompressors could be employed in parallel in a multi-file decompressor system on the same FPGA to achieve even greater throughputs. A cluster of 32 decompressors would provide average input (output) throughputs of 2.24 (7.88) and 4.16 (12.37) GB/s for dynamic and static files, while using about 87% of the LUTs in the FPGA.

The current bottleneck is the Huffman decoder. During decompression, the data count in the FIFO rarely went above 20 for most files indicating that the LZ77 decoder was able to efficiently handle any backlog that started to form. The only exception to this is the file "pic", where the dynamic version caused the FIFO count to reach 1441 and the static version caused the FIFO to max out at 4096. The performance of the Huffman decoder might be improved by implementing it using a hardware description language like VHDL or Verilog. Higher clock frequencies may also be achievable. The LZ77 decoder could be improved by partitioning the circular buffer further and increasing the sliding window width. By reading and writing 8 or 16 bytes per cycle, the LZ77 decoder could copy longer length-distance pairs much faster. However, the benefits would only be seen with highly compressed files that actually have very large length-distance pairs, like "pic".

A speculatively parallelized design could be created to boost the speed of Huffman decoding using multiple decoders in parallel. The speed-ups obtained this way would be statistical, however, and might not be much in the average case. A parallel arrangement of multiple LZ77 decoders is also feasible but would be complex. The LZ77 decoders working in parallel would need to be tightly coupled and aware of the data held in other decoders. When a back-reference to another block is encountered by one decoder, it would need to stop and signal the decoder holding those bytes. That decoder would then need to stop and fetch the needed bytes (assuming it had decoded up to that point already) and send them back to the first decoder. Further work will consider this approach.

TABLE II
FPGA Utilization

Resource	Used	Total	Percent
LUTs	10,736	394,080	2.72%
Registers	6,334	788,160	0.80%
BRAM Tiles	14	720	1.94%

V. Conclusion

We used high-level synthesis to create, from a C++ specification, a compact FPGA-based hardware accelerator for the widely used Deflate data compression standard. Unlike most reported Deflate accelerators, our design can decompress any Deflate-compliant compressed file composed of statically or

TABLE III
CALGARY CORPUS BENCHMARK DECOMPRESSION RESULTS

	Dynamic Compressed Files				Static Compressed Files			
Compressed File	Compressed Size (bytes)	Compression Ratio	Decompression Time (μs)	Throughput (MB/s)	Compressed Size (bytes)	Compression Ratio	Decompression Time (μs)	Throughput (MB/s)
bib	35,222	3.16	471.948	74.63	40,931	2.72	292.220	140.07
book1	313,576	2.45	4197.576	74.70	384,953	2.00	2538.596	**151.64**
book2	206,658	2.96	2714.984	**76.12**	243,843	2.51	1643.424	148.37
geo	68,427	1.50	1040.708	65.75	80,949	1.26	668.172	121.15
news	144,794	2.60	2017.096	71.78	168,375	2.24	1266.768	132.92
obj1	10,311	2.09	164.356	**62.74**	11,138	1.93	109.392	101.82
obj2	81,499	3.03	1155.584	70.53	88,949	2.77	733.724	121.23
paper1	18,552	2.87	255.952	72.48	21,670	2.45	157.276	137.78
paper2	29,754	2.76	402.156	73.99	35,499	2.32	241.500	146.99
pic	56,459	9.09	896.580	62.97	67,529	7.60	667.552	**101.16**
progc	13,337	2.97	191.152	69.77	15,365	2.58	117.756	130.48
progl	16,249	4.41	224.060	72.52	18,603	3.85	138.320	134.49
progp	11,222	4.40	160.564	69.89	12,771	3.87	97.792	130.59
trans	19,039	4.92	262.912	72.42	21,424	4.37	165.472	129.47

dynamically encoded blocks. As shown in Table III, our design decompresses the 14 dynamically encoded files from the Calgary corpus with a maximum input throughput of 76.1 MB/s ("book2") and with average input (output) throughputs of 70.7 (246.4) MB/s. For statically encoded files the maximum input throughput increases to 151.6 MB/s ("book1") with average input (output) throughputs of 130.6 (386.6) MB/s. The expertly optimized, proprietary design from Xilinx outputs on average 3 bytes per clock cycle for an output throughput of 375 MB/s on most Xilinx FPGAs [8]. Unfortunately the design details, the design methodology, and the target FPGAs are not disclosed. However, our data throughputs are comparable even though we restricted ourselves to using high-level synthesis. Few other decompressor designs are available for comparison. The older design in [17] produced average output throughputs of 159 MB/s for statically compressed Calgary corpus files. Our compact design (see Table II) would allow many decompressor cores to be fit onto one FPGA. The mid-range UltraScale+ FPGA that we assumed could easily fit 32 decompressors producing average output throughputs of 7.88 and 12.37 GB/s for dynamically and statically encoded Calgary corpus files, respectively. These projected numbers would likely be reduced in practice because of data bandwidth limitations elsewhere in the server. Future work will focus on investigating speculative parallel decompressors, on hand-optimizing the designs of the Huffman and LZ77 decoder modules, and on implementing a Deflate compressor using high-level synthesis.

REFERENCES

[1] L. P. Deutsch. (1996) "DEFLATE compressed data format specification version 1.3". [Online]. Available: https://www.w3.org/Graphics/PNG/RFC-1951

[2] D. Salomon, *Data Compression: The Complete Reference*, 4th ed. Springer Science & Business Media, 2006.

[3] J.-l. Gailly and M. Adler. "zlib". [Online]. Available: https://zlib.net/zlib.html

[4] ——. "gzip". [Online]. Available: https://www.gzip.org/

[5] M. Adler. "pigz". [Online]. Available: https://zlib.net/pigz/

[6] J. Cong, Z. Fang, M. Huang, L. Wang, and D. Wu, "CPU-FPGA Coscheduling for big data applications," *IEEE Design & Test*, vol. 35, no. 1, pp. 16–22, 2018.

[7] E. Sitaridi, R. Mueller, T. Kaldewey, G. Lohman, and K. A. Ross, "Massively-parallel lossless data decompression," in *the 45th Int. Conf. Parallel Processing (ICPP)*, 2016, pp. 242–247.

[8] Xilinx and CAST Inc. GUNZIP/ZLIB/Inflate Data Decompression Core. [Online]. Available: https://www.xilinx.com/products/intellectual-property/1-79drsh.html\#overview

[9] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Info. Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[10] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proc. of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[11] H. Jang, C. Kim, and J. W. Lee, "Practical speculative parallelization of variable-length decompression algorithms," *SIGPLAN Not.*, vol. 48, no. 5, pp. 55–64, June 2013.

[12] Z. Wang, Y. Zhao, Y. Liu, Z. Chen, C. Lv, and Y. Li, "A speculative parallel decompression algorithm on Apache Spark," *Journal of Supercomputing*, vol. 73, no. 9, pp. 4082–4111, 2017.

[13] 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, 2013.

[14] 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 on OpenCL 2013 & 2014*, ser. IWOCL '14. New York, NY, USA: ACM, 2014, p. 4:9.

[15] J. Fowers, J. Y. Kim, D. Burger, and S. Hauck, "A scalable high-bandwidth architecture for lossless compression on FPGAs," in *IEEE 23rd Annu. Int. Symp. Field-Programmable Custom Computing Machines (FCCM)*, 2015, pp. 52–59.

[16] 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 *IEEE 26th Annu. Int. Symp. Field-Programmable Custom Computing Machines (FCCM)*, 2018, pp. 37–44.

[17] J. Lazaro, J. Arias, A. Astarloa, U. Bidarte, and A. Zuloaga, "Decompression dual core for SoPC applications in high speed FPGA," in *IECON 2007 - 33rd Annu. Conf. IEEE Industrial Electronics Society*, 2007, pp. 738–743.

[18] J. Ouyang, H. Luo, Z. Wang, J. Tian, C. Liu, and K. Sheng, "FPGA implementation of GZIP compression and decompression for IDC services," in *IEEE Int. Conf. Field-Programmable Technology (FPT)*. IEEE, 2010, pp. 265–268.

[19] D. C. Zaretsky, G. Mittal, and P. Banerjee, "Streaming implementation of the ZLIB decoder algorithm on an FPGA," in *IEEE Int. Symp. Circuits and Systems*, 2009, pp. 2329–2332.

[20] T. Bell, I. H. Witten, and J. G. Cleary, "Modeling for text compression," *ACM Computing Surveys (CSUR)*, vol. 21, no. 4, pp. 557–591, 1989.

[21] Xilinx Inc. (2019) "Vivado High-Level Synthesis". [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[22] ARM, "AMBA 4 AXI4-Stream Protocol," 2010.