A Scalable High-Bandwidth Architecture for Lossless Compression on FPGAs

Jeremy Fowers*, Joo-Young Kim* and Doug Burger*
*Microsoft Research
Redmond, WA, USA

Scott Hauck*†
†Department of Electrical Engineering
University of Washington, Seattle, WA, USA

Abstract—**Data compression techniques have been the subject of intense study over the past several decades due to exponential increases in the quantity of data stored and transmitted by computer systems. Compression algorithms are traditionally forced to make tradeoffs between throughput and compression quality (the ratio of original file size to compressed file size). FPGAs represent a compelling substrate for streaming applications such as data compression thanks to their capacity for deep pipelines and custom caching solutions. Unfortunately, data hazards in compression algorithms such as LZ77 inhibit the creation of deep pipelines without sacrificing some amount of compression quality. In this work we detail a scalable fully pipelined FPGA accelerator that performs LZ77 compression and static Huffman encoding at rates up to 5.6 GB/s. Furthermore, we explore tradeoffs between compression quality and FPGA area that allow the same throughput at a fraction of the logic utilization in exchange for moderate reductions in compression quality. Compared to recent FPGA compression studies, our emphasis on scalability gives our accelerator a 3.0x advantage in resource utilization at equivalent throughput and compression ratio.**

Keywords-**FPGA; data compression; LZ77; Huffman encoding; hardware accelerator; Xpress; high throughput;**

I. INTRODUCTION

Data compression plays an important role in computer systems by reducing resource usage in both storage and networking in exchange for some computational cost. The recent computing paradigm shift from personal to cloud has emphasized the role of compression for building efficient storage systems.

Lossless compression often exploits statistical redundancy in data patterns and applies a representation to eliminate it. The Lempel-Ziv (LZ) [1] compression method finds duplicate strings from earlier in the data and replaces the strings with pointers. On the other hand, Huffman encoding [2] collects frequency information for symbols and assigns fewer bits to commonly used symbols to reduce the average codeword length. DEFLATE [3], which combines LZ compression and Huffman encoding to achieve higher compression ratios, is one of the most popular algorithms for lossless storage and is utilized in many variations such as GZIP [4], ZLIB [5], XPRESS [6] and 7ZIP [7].

Streaming data applications, which includes high throughput compression, are typically well-matched to hardware accelerators such as FPGAs and ASICs. Unfortunately, hardware compression studies from the previous decade [8]–[10] have suffered from low throughput performance. However, recent efforts from industrial vendors [11]–[14] have demonstrated 10 GB/s and 4 GB/s lossless compression on ASICs and FPGAs, respectively, proving that these devices are indeed a compelling substrate for compression. These accelerators are useful for both higher throughput and for alleviating the computational cost of compression from the CPU.

While these new accelerators provide an order-of-magnitude throughput increase over modern CPUs, they are resource intensive and will not scale to significantly higher bandwidths on existing hardware. In this paper, we present a scalable high-bandwidth compression accelerator for reconfigurable devices. We introduce a hardware-amenable compression algorithm and demonstrate an area-efficient mapping to a scalable pipelined architecture. The results show that we achieve the same 16 bytes/cycle of throughput as the IBM implementation at 3.1x less area. Additionally, our design is the first FPGA architecture that scales up to 32 bytes/cycle in a single engine on a modern Stratix V FPGA.

The remainder of the paper is organized as follows: In Section II, we detail our compression algorithm and the challenges associated with hardware compression. Next, Section III provides an overview of the pipelined compression architecture. Each module from the architecture is detailed in Section IV, then Section V provides experimental results and analysis. Lastly, Section VI offers conclusions and suggestions for future work.

II. COMPRESSION ALGORITHM

The widely-used DEFLATE compression algorithm includes data-hazards and other features that make it challenging to implement in scalable hardware. Our compression algorithm modifies DEFLATE with the dual goals of enabling data- and pipeline-parallelism while minimizing sacrifices to compression quality. In this section, we will first summarize DEFLATE algorithm, then describe the modifications we made for parallel execution and hardware pipelining with a few newly introduced parameters. It is worth noting that these modifications are made only with regard for hardware mapping, and are not intended to benefit software performance.

Algorithm 1 DEFLATE Algorithm.
```
1: while (curr position<data size) do
2:     Hash chain build
           calculate hash value hv
           Prev[pos] ← Head[hv]
           Head[hv] ← pos
       String matching
           candidates from a hash chain vs. current
       Match selection
           if (either commit literal or match)
               Huffman encoding
               Move to next position
3: end while
```

Algorithm 2 Hash update algorithm.
```
for i = HTD − 1 to 0 do
    Candidate[i] ← HashTable[i][hv]
    if i ≥ 1 then
        HashTable[i][hv] ← HashTable[i − 1][hv]
    else
        HashTable[i][hv] ← pos
    end if
end for
```

A. DEFLATE Algorithm

As shown in Algorithm 1, DEFLATE uses a chained hash table implemented with *head* and *prev* tables to find duplicated strings from earlier in the data. For the current byte, it calculates a hash value by applying a hash function on the current 4-byte sequence. Then, the head table caches the most recent position for each hash value, while the prev table stores linked lists of matching positions. Each linked list starts with the second-most recent position in the hash value. As a result, the algorithm can traverse previous positions that have the same hash value through the prev table. The string matching examines the input data sequence with respect to strings from these candidate positions to find the longest match. To improve overall compression, the algorithm doesn't commit the matches immediately, but instead searches for another matching at the next position. If a longer match is found, the algorithm truncates the previous match to a literal and repeats this process of lazy evaluation until it encounters a worse match. Otherwise, it emits the previous match and skips forward by the match length; this repeats until it covers the entire input data set.

B. Parallelization Window

To parallelize the above algorithm, we perform the algorithm on multiple consecutive positions at the same time in a multi-threaded fashion. We call this spatial set of positions our *parallelization window* and the size of it (the number of positions) the *parallelization window size (PWS)*. Each position in the window executes the hash chain build and string matching processes independently. Among these positions, dependencies exists in the hash table update stage that performs write operations the on head and prev tables. Although we could solve this dependency using the chaining property of head and prev update, we solved it by re-designing the hash table structure, which will be discussed in the following sub-section. Furthermore, the string matching step can also execute in parallel, at the cost of invoking many concurrent read operations to the data memory.

Another dependency issue in our parallelization window approach occurs between neighboring windows. The match selection results from one window impact the match selection process in the following window, resulting in a data hazard for window pipelining. To resolve this problem, we start match selection by finding a match that extends into the next window, then communicate that match to the next window, and finally perform selection for the remaining window positions. Within independent windows we perform a lazy match selection evaluation found only in advanced DEFLATE versions.

C. Re-design of Hash Table

We made a series of significant modifications to the DEFLATE hash table to improve its amenability for hardware. First, we changed the head-prev linked list design to a multiple hash table design. In this design, the first hash table includes the latest positions for hash indexes, while the second hash table has the second latest, the third includes the third latest, and so on. While the original method is more resource efficient, this method has the advantage of retrieving previous positions simultaneously with no need to traverse the prev table. We refer to the number of hash tables with a parameter called *hash table depth (HTD)*, which is equivalent to the number of hash chain walks allowed in the original DEFLATE algorithm. The new hash update process is reflected in Algorithm 2.

Candidates positions with the same hash value can now access all of our hash tables at the same index. Furthermore, the hash tables are updated in a shifted fashion that throws out the oldest position, and has the current position as the latest. However, this simple hash read and update process has a severe hardware realization problem when it is applied to multiple positions simultaneously. As each position requires a single read and a single write for each table, the parallelization case requires PWS read and PWS write operations in a single cycle for each hash table. Given that multi-port memory designs are typically very expensive on FPGA devices [15], an alternative is needed to maintain scalability.

To resolve this problem, we propose a multi-banking solution that does not require any replicas for multi-port
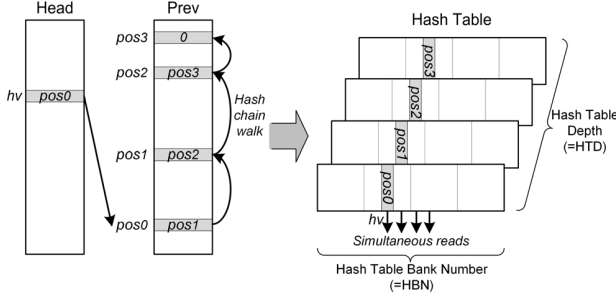
Figure 1. Hash table organization. Original (left) and hardware-optimized (right).

Algorithm 3 Head/tail algorithm.

for all window w[N] in input **do**
   Select a *tail match* that extends into w[N+1]
   if $N \neq 0$ **then**
      Receive *head match* from w[N-1]
      Trim *tail match* start to avoid *head match* overlap
      Invalidate positions in w[N] up to *head match* end
   end if
   Send *tail match* to w[N+1] as a *head match*
   Trim matches in w[N] to avoid *tail match* overlap
   Select matches and literals for remaining positions
end for

---

operation, while increasing read/write throughput up to by the factor of the number of banks, depending on how bank conflicts occur. We refer to the number of banks in the hash table as the *hash table bank number (HBN)*. Each bank performs hashing for positions whose hash modulo HBN falls into the index of the bank. Figure 1 compares the re-designed hash table organization with the original DEFLATE version.

Our re-designed hash table will encounter bank conflicts when hash values have the same modulo result. To achieve both a seamless hardware pipeline and a single read and a single write requirement per bank, we drop all of the conflicted inputs in the hash table after the second request from the smallest position. It is noteworthy that this dropping strategy actually solves the dependency problem mentioned in previous sub-section as well. However, we miss some possible candidate positions due to the dropping, and thus there is a loss in compression. We can mitigate the bank conflict by choosing a large enough HBN value since the hash function generates well-spread hash values. Based on our experiments, we picked an HBN value of 32 as it give a nice trade-off between conflict reduction effect and hardware cost for multi-banking.

D. Match Selection

Matches identified by string matching can overlap with each other. Match selection identifies which matches should be included in the output stream. The primary challenge with parallel match selection is that selecting each match requires knowledge of both past matches (do they preclude this match?) and future matches (would selecting this match preclude a better one later?). Therefore, a data hazard exists between windows if a match in one window can cover positions in the following window.

We enable parallel match selection by employing two techniques. The first, *match trimming*, reduces the size of matches to eliminate the dependencies between windows. However, we found that match trimming results in a 10-20% reduction in available compression quality. To improve quality at the cost of some resources we can apply our second technique, which we refer to as *head/tail selection*. Head/tail pre-processes each window by selecting one match that extends into the next window, then trims to the match candidates in both windows accordingly. We found that head/tail reduces the compression loss to only 2-6%.

The process for applying the head/tail technique is shown in Algorithm 3. Our selection approach uses the following heuristics for selecting matches. For tail matches, the selector chooses the longest match in the window that extends into the next window. For all other matches, we rely on the fact that the largest matches occur at low-indexed positions due to match trimming. The selector begins at the lowest-indexed position and compares its match length to its neighbor one index higher. If a position contains a longer match than its neighbor we select a match, otherwise we select a literal. Finally, trimming a match involves shifting its start position past the end of the head match and shifting its end position before the start of the tail match.

III. FULLY-PIPELINED ARCHITECTURE

Figure 2 shows the fully pipelined architecture of the proposed compressor with $22 + 2 * PWS + log_2(PWS)$ total stages. The compressor receives PWS bytes of data from its input source every cycle and directs them into our stall-free fixed latency pipeline. Thanks to this no-stall architecture, its input and output data rate are very simple to calculate; the input data rate is computed as (PWS x clock rate) bytes per second, while the output rate will be divided by that data set's compression ratio. The proposed architecture is composed of four major functional components: hash table update (hash calculation included), string match, match selection, and Huffman bit-packing. To sustain high data throughput while shrinking multiple bytes to compressed bits, we face the following design challenges for each component:

- Hash table update: The hash calculation module converts PWS bytes into hash values ranging from 0 to 64K-1, while also storing the bytes to data memory.
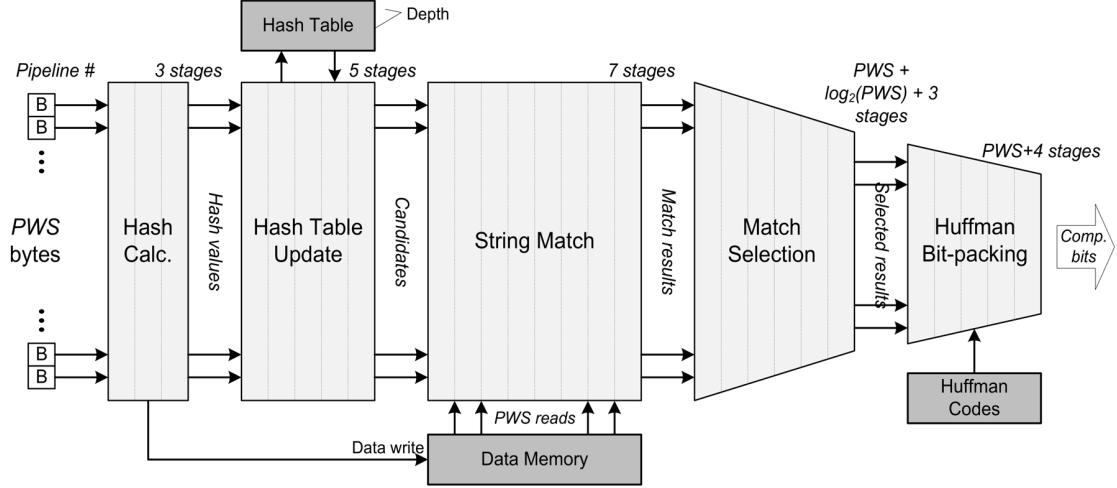
Figure 2. Fully-pipelined architecture.

Then, it must read candidates for PWS hash values from the multi-banked hash table, while resolving bank conflicts among the inputs. Simultaneously, it updates the hash table with new positions.

- String match: At each candidate position in the input the string matcher performs PWS independent matchings between the current string and previous strings. This parallel matching computation will require PWS reads of PWS bytes to the 64KB data memory.
- Match selection: We receive PWS candidate matches per cycle, which is too many to process combinatorially, and must perform lazy evaluation to select between them. Data hazards exist because each match may preclude other matches within a certain range of positions.
- Huffman bit-packing: We must create a PWS-aligned byte stream out of many Huffman encoded selection results. A large amount of buffering and bit shifting is required to align the data.

IV. MICRO-ARCHITECTURES

A. Multi-banking hash table

As we stated in Section II-C, we resolved the multiple read/write problem in the hash table with a multi-banking scheme that drops banking conflicts. Figure 3 shows the 5-stage pipelined hash table read and update module with two fully connected crossbar switches. The hash table module receives PWS input hash values per cycle from the hash calculation module. Based on their LSB values, it routes each input position to the corresponding bank. This problem is similar to the routing problem in a network switch, with the input positions corresponding to input ports and the bank numbers corresponding to output ports. Multiple input positions can send requests to a single bank and an arbiter chooses up to two requests per cycle. Thanks to a clock
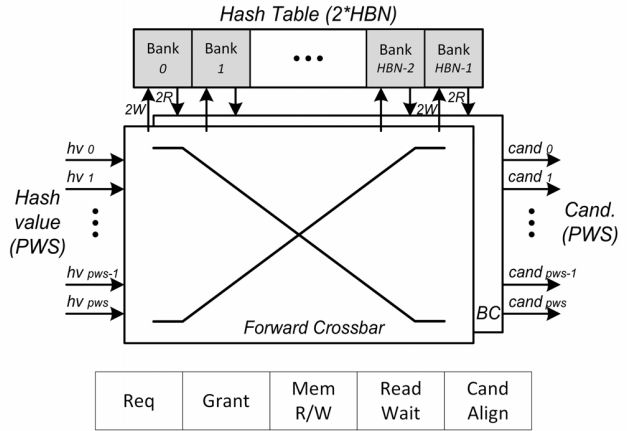

Figure 3. Multi-banking hash table update.

rate double that of the rest of the pipeline, each bank in the hash table can handle up to two hash updates involving two read and two write operations per cycle. Since each input hash value can update any channel of any bank in the hash table, we need a fully connected crossbar switch whose input and output port size is PWS and 2*HBN, respectively. After getting a grant from the bank, each input position accesses the granted bank to read the candidate position value and update it to the current position. For the single-depth hash table, we can perform read and write operations at the same cycle by configuring the memory output mode to read old data. In the case of multi-depth hash table, we need to wait for the read data to arrive, while resolving possible dependencies with forwarding without stalling. The read candidate positions arrive in two cycles from the banks and we re-align these to match the correct input positions. Therefore, another full crossbar switch is required to connect 2*HBN bank ports to PWS output ports. As a result, the
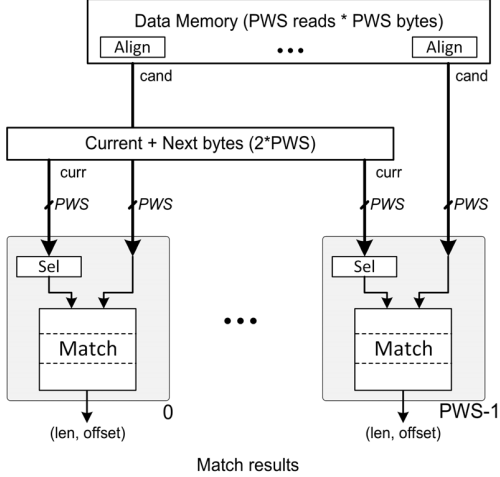
Figure 4. Parallel string matching.

Figure 5. Match selection logic.

module provides PWS candidate positions per cycle to the matching stage.

B. Parallel Matching

In this stage, we perform PWS parallel matchings between the current string and PWS previous strings to which those candidate positions refer, as shown in Figure 4. The current string is stored in pipeline registers, while the previous strings are fetched from the data memory. For the current string, we buffer up to the next window bytes (2*PWS bytes total) so that each position in the current window can have a full PWS byte sequence. The data memory that stores input bytes is designed to prepare vector data for matching. With multiple banks and a data aligner in its data read path, it provides PWS consecutive bytes from any input address. We replicate the data memory by PWS to support parallel matching, providing a total data bandwidth of (PWS*PWS*clock freq) bytes per second.

With two PWS bytes of strings available, the matching process is straightforward. It compares each byte of the two strings until they become different. As a result, we have up to PWS matching results, each of which is represented as a (length, offset) pair.

C. Match Selection

Our match selection pipeline (partially depicted in Figure 5) uses $PWS + log_2(PWS) + 3$ stages to process PWS input elements. Each element contains the literal for that position in the original input, along with match information: length, offset, and valid flag. Pipeline stages 1 through $log_2(PWS) + 2$ are tasked with selecting (with a pipelined max reduction), triming, and propagating the head/tail matches, while the remaining PWS+1 stages trim and select the remaining matches.
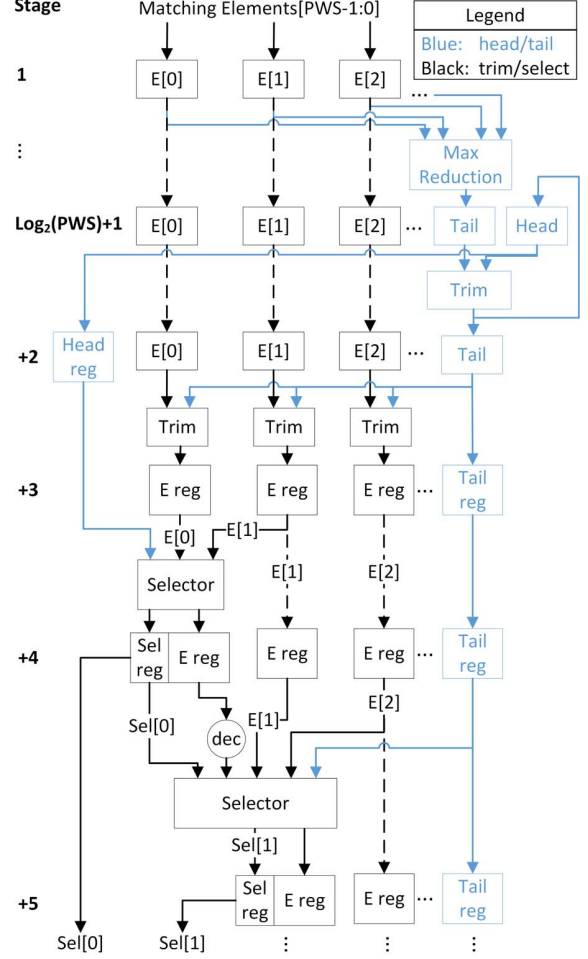
The selector modules depicted in Figure 5 perform the following tasks. First, they ensure that the match in the current position has not been precluded by a previous match, including the head match. Next, they determine whether the tail match should replace the match for the current position. If a selection is still needed, the selector compares the post-trimming match lengths for the current and next-indexed position, as described in Section II-D. Lastly, preclusions are carried between stages by decrementing length of the selected match. A stage's selector knows it must make a new selection when the preclusion value reaches zero.

Note that head/tail can be disabled to reduce resource usage by removing blue modules and wires in Figure 5 simply trimming matches to the end of the window.

D. Huffman Encoding + Bit-Packing

Encoding the match selection results with static Huffman codes is a relatively simple process; however it is challenging to pack the subsequent outputs into a fixed-width output
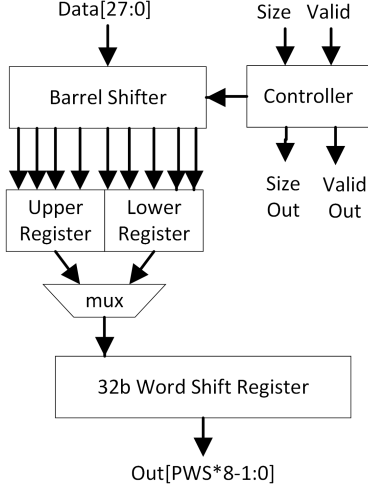
56

Figure 6. Window bit-packer.

bit stream. The encoder itself has two main actions: First, if the data is precluded by a previous match it is ignored. Second, if a match or literal is selected, the data is passed through a codebook ROM and the resulting data and size are passed to the packing module. PWS/2 dual-ported ROMs are used to process one output per window per cycle (PWS cycles for one window).

Packing the data is challenging because encoded outputs for a single selection vary from 7 to 28 bits, and each window can produce between one and PWS outputs. Therefore, the total output from one window can range between 12 and PWS*8 bits, representing a well-encoded match of size PWS and PWS literals, respectively.

Our bit-packing pipeline operates in two phases. First, PWS *window packers* are each responsible for collecting all of the outputs from one of the PWS parallel windows. Each cycle, one window packer will finish its window and send its data to the second phase, a unified *output packer*. The output packer accepts compressed windows and packs them into a PWS-bit output stream.

Our window packer implementation is depicted in Figure 6. It uses a 64-bit barrel shifter to align incoming data with data that has already been collected from the current window. The aligned values are ORed with the contents of the lower of two 32-bit registers in a double buffer, then stored back in the register. The packer's controller tracks the number of bits stored this way, and when a buffer is full its contents are sent to the 32-bit word shift register. Next, the barrel shifter continues to fill the upper register and uses the lower register for overflow. Our window packer uses PWS/4 shift registers, allowing it to pack up to 28 bits/cycle, and a total of PWS*8 bits, using a single 64-bit barrel shifter.

The output packer is a simpler version of the window packer that excludes the shift registers. It accepts inputs of up to PWS*8 bits and uses a PWS*16-bit barrel shifter to align them into a PWS*16-bit double buffer. When one side of the double buffers is filled, the buffer's data is sent as final compressed output.

V. EXPERIMENTAL RESULTS

A. Compression ratio

To evaluate the proposed compression algorithm, we chose 4 different data benchmarks covering a variety of data types: Calgary and Canterbury Corpus [16], Silesia Corpus [17] and the large text benchmark [18]. We use the publicly available ZLIB software implementation [5] of DEFLATE to provide a baseline for compression ratio and measure CPU throughput on a machine with a 2.3GHz Intel Xeon E5-2630 CPU and 32GB RAM.

Figure 7 shows the resulting compression ratios, calculated as input file size divided by output file size, with PWS varying between 8 and 32 and with head/tail selection included. We fixed the hash table bank number to 32 and varied the depth between 1 and 4. Figure 7 demonstrates that our compression ratio increases logarithmically with PWS. This behavior is expected because larger parallelization windows allow for longer matches. The compression gain achieved by increasing PWS for 8 to 16 is especially noteworthy across all data sets, since the PWS of 8 case simply does not produce enough matches due to the minimum match length of 4. Another trend in the graphs is that our compression ratio increases as the hash table depth grows larger. Deeper hash buckets result in better matches because we are able to keep more candidate positions available for comparison. A good observation for hardware implementation is that even increasing the depth to 2 results in a meaningful boost in compression ratio without requiring too large a hardware cost.
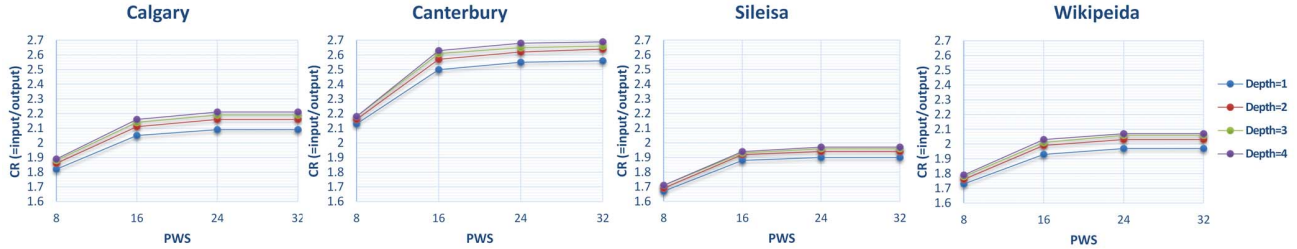
Figure 7.   Compression ratio vs. parallelization window size

B. Area-performance Trade-off

We implemented our architecture with SystemVerilog on a modern Stratix V FPGA at 175 MHz to evaluate the tradeoffs between area, throughput, and compression ratio. The HTD parameter is set to 1, as we found supporting higher values to be outside the scope of this work. Figure 8 presents the area requirements for each module across different PWS sizes and Table I gives the total area for each engine with and without head/tail selection. The results show that the design does not scale linearly, as certain modules—such as data memory and match selection—scale quadratically with PWS. This presents an interesting design space to users, who can achieve a target throughput by either applying one large engine or an array of smaller engines.

Figure 9 depicts the design space for area versus compression ratio at throughputs of 5.6, 2.8, and 1.4 GB/s. Each line represents a different throughput level, and the points along the line are the corresponding engine configurations (PWS x # engines). The data shows that a significant 15% improvement to compression ratio can be achieved if an additional 64% area can be spared for a PWS=32 engine.

C. Comparison

Table I compares our implementation, with PWS set between 8 and 32, to the fastest ZLIB mode, the IBM results from [11], and the Altera results from [12]. For a fair comparison, we used only Calgary Corpus that their results are also based on. We found that the high-throughput hardware accelerators outperformed the throughput of one CPU core running ZLIB by up to 2 orders of magnitude. However, all of the hardware implementations in Table I sacrifice some amount of compression ratio to improve DE-FLATE's hardware amenability. The discrepancy between our designs and the other hardware accelerators can be partially accounted for by the hash table design; the IBM and Altera designs keep more candidate positions than ours, which we will match with a depth of 2 in future work. Finally, we found our design to be significantly more resource-efficient than IBM's or Altera's, achieving 1.4-2.7x and 1.6-3.0x, respectively, better throughput/area across the various PWS settings. With further clock rate optimizations this lead would increase, because a PWS of 16 running at IBM's 250 MHz would result in the same 4 GB/s throughput.

VI. RELATED WORKS

Decades of research has investigated different compression algorithms [1]–[7]. High-bandwidth pipelined FPGA implementations have become interesting recently thanks to area and I/O bandwidth improvements. IBM's DEFLATE implementation for FPGAs [11] achieved 4 GB/s throughput at 16 bytes/cycle; However, certain architectural choices,
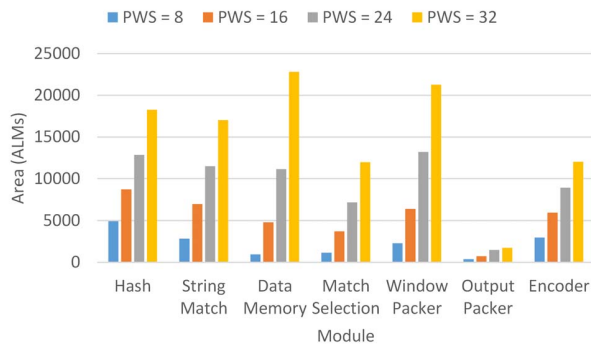

Figure 8.   FPGA area in ALMs required for each module at varying PWS (head/tail selection included).
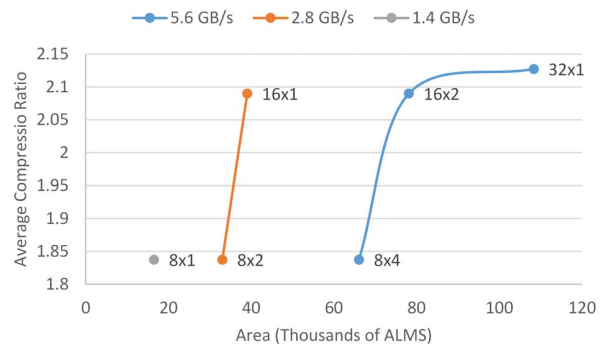

Figure 9.   Area vs. compression ratio at three throughput levels. Data points are labeled with the engine size x number of engines.

such as a 256-port hash table, limit its scalability. Altera [12] has also recently implemented LZ77 for FPGAs with their OpenCL compiler at the same throughput as IBM. However, they use an unscalable combinatorial match selection circuit. Our compression architecture improves on both of these implementations with significantly greater area efficiency and overall scalability.

An alternative approach from IBM is presented in [13], which makes significant sacrifices to compression quality to achieve hardware amenability. In particular, the 842B algorithm used in this work only allows matches of size 8, 4, and 2 and does not apply Huffman encoding. Our work significantly improves compression ratios by allowing larger matches and integrating static Huffman codes.

Microsoft presented their Xpress compression accelerator [19] targeting high compression ratios with complex matching optimization and dynamic Huffman encoding. However, its throughput performance is limited to less than a GB/s due to the complex algorithmic flow.

Application-specific integrated circuits (ASICs) have also been used to accelerate compression. The latest AHA372 product performs 2.5GB/s throughput, while the next generation AHA378 product [14] is capable of up to 10GB/s on a sixteen-lane PCIe board. Our scalable FPGA implementation is complementary because it offers a compelling alternative for servers that already include an FPGA [20].

VII. CONCLUSION & FUTURE WORK

In this paper we presented a scalable architecture for high-bandwidth lossless compression on reconfigurable devices. To enable seamless pipelining in hardware, we resolved algorithmic dependencies by introducing a new hash table design and trimming matches. Although these changes sacrifice some amount of compression ratio, they enable our architecture to scale to 5.6 GB/s of throughput. We also detailed micro architectural components for the compression pipeline, including modules for hash table update, string matching, selection, and Huffman bit-packing in a scalable and resource-efficient way. Finally, we explored a design space of proposed architecture with parallelization window size and embraced area-performance trade-off relations. At the time of this writing, our architecture achieves the highest throughput and area-efficiency of any published high-bandwidth FPGA compressor.

Future work will investigate clock frequency optimizations to improve the throughput of a given PWS value. We also plan to integrate deep hash tables to improve the our compression ratio.

ACKNOWLEDGMENT

REFERENCES

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

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

[3] P. Deutsch. (1996) Rfc 1951 deflate compressed data format specification version 1.3. [Online]. Available: https://tools.ietf.org/html/rfc1951

[4] P. Deutsch. (1996) Gzip file format specification version 4.3. [Online]. Available: http://tools.ietf.org/html/rfc1952

[5] P. Deutsch and J.-L. Gailly. (1996) Zlib compressed data format specification version 3.3. [Online]. Available: http://tools.ietf.org/html/rfc1950

[6] Microsoft. (2014) Ms-xca: Xpress compression algorithm. [Online]. Available: http://msdn.microsoft.com/en-us/library/hh554002.aspx

[7] I. Pavlov. (2013) Lzma sdk. [Online]. Available: http://www.7-zip.org/sdk.html

[8] W.-J. Huang, N. Saxena, and E. J. McCluskey, "A reliable lz data compressor on reconfigurable coprocessors," in *Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2000, pp. 249–258.

[9] S.-A. Hwang and C.-W. Wu, "Unified vlsi systolic array design for lz data compression," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 9, no. 4, pp. 489–499, 2001.

[10] S. Rigler, W. Bishop, and A. Kennings, "Fpga-based lossless data compression using huffman and lz77 algorithms," in *Electrical and Computer Engineering, 2007. CCECE 2007. Canadian Conference on*. IEEE, 2007, pp. 1235–1238.

[11] A. Martin, D. Jamsek, and K. Agarwal, "Fpga-based application acceleration: Case study with gzip compression/decompression streaming engine," in *Special Session 7C, International Conference on Computer-Aided Design*. IEEE, 2013.

[12] M. S. Abdelfattah, A. Hagiescu, and D. Singh, "Gzip on a chip: High performance lossless data compression on fpgas using opencl," in *International Workshop on OpenCL*. ACM, 2014.

[13] B. Sukhwani, B. Abali, B. Brezzo, and A. Sameh, "High-throughput, lossless data compression on fpgas," in *Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2011, pp. 113–116.

[14] AHA378. (2015). [Online]. Available: http://www.aha.com/data-compression/

[15] C. E. LaForest and J. G. Steffan, "Efficient multi-ported memories for fpgas," in *International Symposium on Field Programmable Gate Arrays*. ACM, 2010, pp. 41–50.

[16] M. Powell. (2001) The canterbury corpus. [Online]. Available: http://corpus.canterbury.ac.nz/descriptions/

[17] S. Deorowicz. (2014) Silesia compression corpus. [Online]. Available: http://sun.aei.polsl.pl/ sdeor/index.php?page=silesia

[18] M. Mahoney. (2014) Large text compression benchmark. [Online]. Available: http://mattmahoney.net/dc/text.html

[19] J.-Y. Kim, S. Hauck, and D. Burger, "A scalable multi-engine xpress9 compressor with asynchronous data transfer," in *Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2014, pp. 161–164.

[20] A. Putnam *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *41st Annual International Symposium on Computer Architecture (ISCA)*, 2014.