High-Throughput Lossless Compression on Tightly Coupled CPU-FPGA Platforms

ABSTRACT

Data compression techniques have been widely used to reduce the data storage and movement overhead, especially in the big data era. While FPGAs are well suited to accelerate the computation-intensive lossless compression algorithms, big data compression with parallel requests in nature poses two challenges to the overall system throughput. First, scaling existing single-engine FPGA compression accelerator designs already encounters bottlenecks which will result in lower clock frequency, saturated throughput and lower area efficiency. Second, when such FPGA compression accelerators are integrated with the processors, the overall system throughput is typically limited by the communication between a CPU and an FPGA.

In this work we propose a novel multi-way parallel and fully pipelined architecture to achieve high-throughput lossless compression on modern Intel-Altera HARPv2 platforms. To compensate for the compression ratio loss in a multi-way design, we implement novel techniques, such as a better data feeding method and a hash chain to increase the hash dictionary history. Our accelerator kernel itself can achieve a compression throughput of 12.8 GB/s (2.3x better than the current record throughput) and a comparable compression ratio of 2.03 over standard benchmarks. Our approach enables design scalability without clock frequency drop and also improves the performance per area efficiency (up to 1.5x). Moreover, we exploit the high CPU-FPGA communication bandwidth of HARPv2 platforms to improve the compression throughput of the overall system, which can achieve an average practical end-to-end throughput of 10.0 GB/s (up to 12 GB/s for larger input files) on HARPv2.

1. INTRODUCTION

Data compression techniques have been widely used in datacenters to reduce the data storage and network transmission overhead. The Deflate data compression algorithm [1] is one of the most widely used algorithms at the core of many lossless compression standards such as ZLIB [2] and GZIP [3]. However, prior profiling results [4] using the standard Calgary Corpus [5] datasets show that the ZLIB [2] software can only achieve a compression throughput of 38 MB/s under the fastest configuration. This compression throughput is too low and significantly diminishes the performance benefits from reduced data storage and transmission.

The increasing demand for efficient data compression has stimulated a number of studies on compression acceleration in CPUs [6], ASICs [7] and FPGAs [4, 8–10]. Due to their high performance, lower power, and reconfiguration flexibility, FPGAs have attracted increased attention from the community. The opportunity of designing a high-throughput custom memory architecture on FPGAs makes it a promising candidate for Deflate accelerators. Recent studies [4, 10] have demonstrated impressive compression results on FPGAs, which are competitive to ASIC implementations. One study from Altera [4] implements a Deflate accelerator using OpenCL and leverages the Altera OpenCL compiler to automatically generate the final FPGA bitstream. This OpenCL implementation can process 15 bytes per cycle at a clock frequency of 193 MHz, thus achieving a compression throughput of 2.84 GB/s. Another study from Microsoft [10] using a hardware description language (HDL) scales the design to process 32 bytes per cycle at a maximum clock frequency of 175 MHz. It achieves a record compression throughput of 5.6 GB/s on FPGAs.

However, most prior studies [4, 10] only report the theoretical compression throughput for the FPGA kernel itself. In practice, an FPGA needs to read its input data from a CPU's DRAM, perform the compression, and then write the output data back to the CPU's DRAM. This CPU-FPGA communication can introduce significant overhead, especially in the mainstream PCIe-based loosely coupled CPU-FPGA platforms [11]. Indeed, two recent studies [12, 13] observe a significant degradation of the compression throughput for the overall PCIe-based CPU-FPGA platform. They only achieve marginal improvement compared to a multicore CPU implementation for big data workloads that usually perform compression on different data partitions concurrently.

Even given higher CPU-FPGA communication bandwidth, scaling current single-engine FPGA accelerator designs also encounters the bottleneck. As the latest Microsoft design [10] presented, each time it scales the number of bytes processed per cycle (BPC) by 2x, the resource utilization will increase by around 3x. Moreover, even provided with more area resources in larger FPGAs, further scaling BPC (e.g., from 32 to 64) for a single compression engine will increase the critical path and degrade the clock frequency, making the total throughput saturated. As a result, there will be a large degradation of performance per area efficiency when further scaling BPC in a single-engine design.

In this paper we present a novel multi-way parallel Deflate compression accelerator design where each way represents a well-optimized and fully pipelined Deflate engine. Such a parallel architecture improves the overall design scalability as the clock frequency of the design is not affected by the number of Deflate engines. It also improves the performance-per-area efficiency since the resource utilization goes almost linearly with the throughput.

However, the multi-way parallel design comes at a cost of degraded compression ratio due to the fact that compression opportunities in one compression engine may disappear as the matching records reside in another engine. To maintain a comparable compression ratio to prior studies, we provide novel optimizations within each single-way Deflate accelerator engine that are not implemented in prior studies, including 1) a better data feeding method to reduce the loss of dictionary records, 2) a hash chain to increase the hash dictionary history. Moreover, we also break the clock frequency bottleneck in current designs by introducing the double bank design instead of the double clock design used in [10], and reverse multiplexer designs in hash memory update.

By parallelizing up to four Deflate engines on HARPv2, we can compress up to 64 bytes of data per cycle with a fixed clock frequency of 200 MHz, at a compression ratio of 2.03. That is, our FPGA Deflate accelerator can achieve a peak compression throughput of 12.8 GB/s, which, to the best of our knowledge, is the best published result. Compared to the record accelerator design by Microsoft [10] with 5.6 GB/s throughput, we achieve 2.3x higher throughput, 1.5x performance-per-area efficiency, more scalability (e.g., no clock frequency degradation and less design efforts), and a comparable compression ratio (96%). In addition, on the HARP platform that has the same Stratix V FPGA as the Microsoft design [10], our accelerator design can achieve a throughput of 9.6 GB/s, with a 1.4x better performance-per-area efficiency.

Finally, we also explore the impact of CPU-FPGA communication bandwidth on system-level compression throughput. We wrap our FPGA accelerator with the CPU software invocation and abstract it as a software library on modern Intel-Altera HARP and HARPv2 platforms. With the CPU-FPGA communication bandwidth significantly increased, we achieve an average practical end-to-end compression throughput of 3.9 GB/s on HARP and 10.0 GB/s (up to more than 12 GB/s for large input files) on HARPv2. This shows that the compression design is rather powerful in real-world applications. We also plan to open source our design to the community in the near future.

2. BACKGROUND

In this section we first discuss prior studies about compression accelerators. Then we introduce the lossless Deflate algorithm—which includes the LZ77 algorithm [14] and static Huffman encoding [15]—and review its details for existing single-engine FPGA implementation.

2.1 Prior Studies and Their Limitations

FPGAs have attracted increased attention from the community in the past decade based on their high performance, energy efficiency and flexibility. Many efforts have been made to increase the compression accelerator's throughput and scalability.

Altera [4] implemented an Deflate accelerator using OpenCL that processes 15 bytes per cycle at a kernel frequency of 193 MHz. They achieved a 2.84 GB/s throughput and 2.17x compression ratio at the expense of using 47% of the logic and 70% of the RAM. Microsoft [10] expanded the compression throughput to 5.6 GB/s by scaling up to 32 byte/cycle at a clock frequency of 175 MHz, which achieved a record of compression throughput on FPGAs. Further scaling of these two designs will be even harder because the designs already consume large area, and as shown by [10], the growth of FPGA area usage is much faster than the incremental bytes processed per cycle. The kernel's running frequency may also drop as the FPGA area usage increases, which will degrade the performance improvement.

IBM proposed a multi-way parallel compression engine design based on the 842B algorithm [8]. They implemented a single compression engine that processes 8 bytes per cycle at a clock frequency of 125 MHz and achieves a compression ratio of 1.96. By applying four engines they can get a throughput of 4 GB/s, but the compression ratio will be further sacrificed based on our study. Furthermore, there are no system interface and data feeding methods to support the multi-way parallel compression kernel, and thus no in-depth analysis or solid implementation when the compression engine scales to process a larger equivalent data window size—as will be done in our study. Later on, IBM implemented another Deflate accelerator [9] which achieved a throughput of 16 bytes/cycle at 250 MHz, i.e., 4 GB/s. However, its scalability is limited due to certain architectural choices like a 256-port hash table. Another Xpress9 compressor [16], targeting high compression ratio, integrated seven engines to support heavily multi-threaded environments. However, its throughput is only limited to 200 to 300 MB/s.

A quantitative comparison of our work to the recent IBM [9], Altera [4], and Microsoft [10] implementations will be presented in Section 5.2.

2.2 Deflate Algorithm and Review

The lossless Deflate algorithm [1] mainly includes two stages: first, it performs the dictionary-based LZ77 [14] compression; second, it performs the Huffman encoding [15] to compress at bit level.

2.2.1 Algorithm Overview

The LZ77 [14] compression algorithm scans the incoming byte stream and compares the new input with the entries in a dictionary which is populated by past input data. After finding a common string of length L, this repeated section of the incoming string is replaced with a (L, D) pair, where D is the distance between the history and the incoming string. Then, the incoming string is recorded in the dictionary for future references. The Deflate format limits distances to 32K bytes and lengths to 258 bytes, with a minimum length of 3.

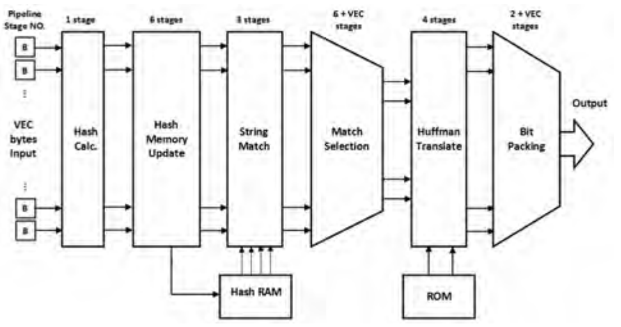


Figure 1: Single-engine fully-pipelined Deflate accelerator architecture

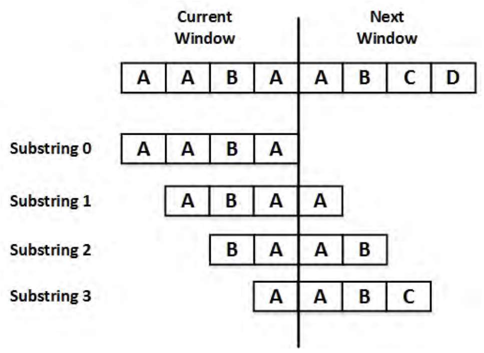


Figure 2: Current and next window of input string to be compressed

Huffman encoding is known to produce the minimum length encoding given the alphabet and the relative frequency of all characters. The Deflate algorithm has a dynamic Huffman encoding option, where the algorithm creates a Huffman code with the relative frequency produced by the current input, resulting in the minimum length encoding for the given input's LZ77 result. However, it requires frequency tracking and must be done after the LZ77 phase. Dynamic Huffman encoding is typically used in higher levels of ZLIB and GZIP standards, where a high compression ratio is favored over throughput. On the other hand, the Deflate algorithm also allows a static Huffman encoding option, where the Huffman code is generated by a golden frequency and is statically available. We use the static Huffman encoding option in this paper in order to enable the fully-pipelined accelerator design.

2.2.2 Review of Existing Single-Engine Implementation

Our initial single-engine accelerator architecture is shown in Figure 1, which is similar to [4] and [10]. We summarize the implementation of six major stages in this subsection and refer the audience to [4] and [10] for more details. Let us denote the bytes we are currently processing "the current window," and *the number of bytes processed in each clock cycle "VEC," which represents the compression throughput*.

Stage 1: Hash calculation. In each cycle the pipeline extracts all the substrings of length *VEC*, starting from every byte in the current window, and indexes each substring to its corresponding history memory using hashing. For the example of VEC=4 in Figure 2, it extracts four substrings (each with length 4) for the current window. VEC-1 more bytes from the next window are required for extracting substrings starting from later bytes of the current window. These substrings are then hashed and later matched with history data to implement the LZ77 algorithm. *The length of VEC represents the substring lengths to be compared, and therefore the maximum match length*. It seems the better maximum match

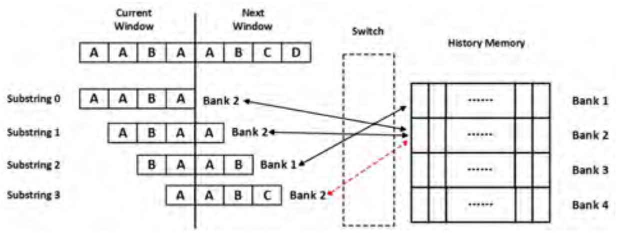


Figure 3: Hash (bank) conflict solution 1: double clock, proposed by [10]

length, the better the compression ratio. However, the study in [10] shows no compression ratio improvement when increasing VEC from 24 to 32. This is because in most of the standard benchmarks there is no match whose length is larger than 24, or such a match cannot be detected in the proposed algorithm. This observation is important as it enables us to start from a relatively small VEC design (e.g., VEC=16) with negligible compression ratio loss.

Stage 2: Hash memory update. The hardware compares these VEC input substrings to the records in the hash history memory, and replaces the old records with these input strings. The hash history memory is a dual-ported RAM that enables one read and one write per clock cycle. To help reduce bank access conflict, [10] suggested the hash memory runs at twice the clock rate of the rest of the system so each bank can handle two read requests in one system clock cycle. Figure 3 shows an example of bank conflict. Assuming strings "AABA," "ABAA," and "AABC" are all mapped to bank 2, in this case only the first two strings are allowed to access the bank. The switch part that connects each substring to its corresponding memory bank will become the critical path when we scale VEC to a higher number. It will limit the clock frequency of the design, since each time it needs to reorder VEC inputs while each input is a string of VEC bytes. The situation will be even worse when using a doubled-rate clock frequency since the switch also needs to be operated at the same clock frequency.

Stage 3: String match. The VEC records from hash history memory are matched with their corresponding input strings in parallel, and any match with match length smaller than 3 is rejected. Another switch is required to remap the VEC records to their counterparts, and it faces the same timing challenges as the one in Stage 2. According to the LZ77 algorithm [14], an (L,D) pair is emitted for each input position. If there is a match, then L represents match length, and D represents the distance between the history record and the current input string. If the match is rejected, then L is the byte value and D is 0. A special case that needs to be addressed is: if match length L is larger than D, L needs to be truncated to D to avoid overlapping.

Stage 4: Match selection. The matches in the current window could stretch to the next window and need to be pruned to avoid match overlapping between windows. This is implemented in two phases. First, in each cycle VEC matches from each byte in the current window will be compared to select a best match that extends farthest. The best match will be taken and any byte included in the best match will be covered. This comparison can be done in parallel. Then the bytes in-between the current best match and the best match extended from the previous window will be examined one by one to resolve the conflict between adjacent matches, this is referred to as lazy evaluation [10]. This phase is a loop behavior and can be implemented as a series of VEC pipeline stages.

Stage 5: Huffman translation. The following stage is Huffman encoding. The method of counting symbol frequency and using dynamic Huffman encoding no longer works because the Huffman packing pipeline must run simultaneously with LZ77 stages in a pipeline fashion. Therefore, static Huffman encoding is used in this design. Static Huffman encoding is nothing but a dictionary lookup, which is implemented as a ROM. The VEC (L,D) pairs can be looked up within VEC ROMs in parallel. After each (L,D) gets translated, we get a four-code tuple (Lcode, Dcode, Lextra, Dextra). *Lcode* and *Dcode* are the codes for literal and distance; *Lextra* and *Dextra* are the extra bits to encode literal and distances.

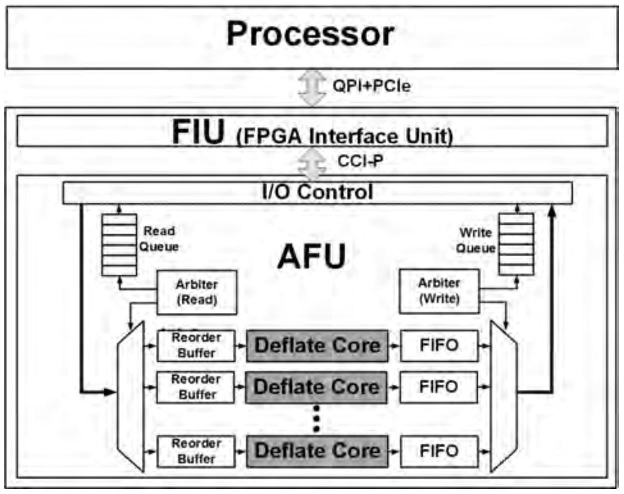


Figure 4: Overview of our multi-way parallel and fully-pipelined Deflate accelerator design

Stage 6: Bit packing. The last step involves packing the binary codes of different lengths together and aligning them to byte boundaries. This is because the length of the four-code tuple output ranges from 8 bits to 26 bits, while the data we finally stored are in byte format. Packing can be easily achieved by a series of shift-OR operations [10].

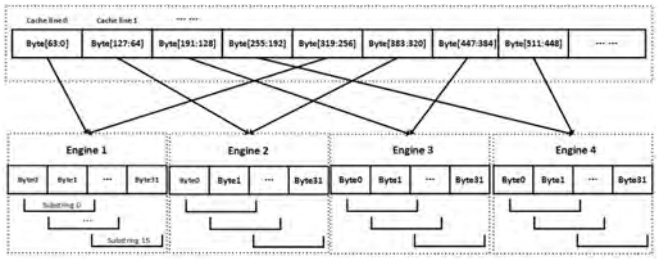
3. ACCELERATOR OPTIMIZATION

To improve the compression throughput and overall system efficiency of the FPGA Deflate accelerator design, we exploit a multi-way parallel design, where each accelerator engine can compress a relatively small amount of data concurrently. A system-level overview of our multi-way accelerator design is shown in Figure 4. The details of the CPU-FPGA interface will be introduced in Section 4.1. Our initial accelerator engine pipeline is similar to that in [4] and [10]. Note that for a single accelerator engine, as the data window size *VEC* increases from 16 to 32 bytes (per cycle), the ALM resource usage on FPGAs is increased nonlinearly by roughly 2.6x times, while the compression ratio is improved only by 3%, as observed both by us and the study in [10]; so using a data window size of 16 would be a good trade-off of resource usage and compression ratio. We will evaluate the design trade-offs in Section 5.

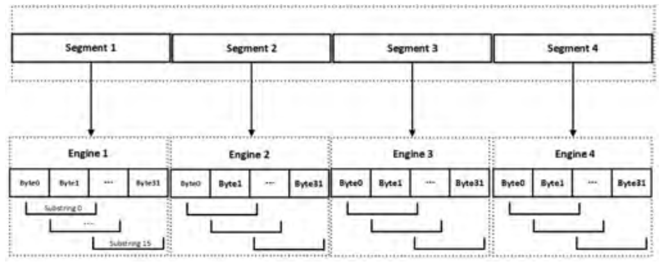
To compensate for the compression ratio loss in our multi-way design and improve the overall efficiency, in this section we will mainly present novel optimizations implemented in our accelerator. First, we propose a better data feeding method to multiple engines to reduce the loss of dictionary records and thus improve the compression ratio. Second, we propose single engine optimizations, including hash chains to improve the hash dictionary length and compression ratio, double bank design and switch optimization to improve clock frequency and resource utilization.

3.1 Multi-Engine Data Feeding Method

Since we propose a multi-way parallel FPGA Deflate accelerator design shown above in Figure 4, we need to divide the input file into multiple (four in our example) segments to feed each Deflate engine (core). There are two ways to fetch data for the FPGA accelerator. The first is cyclic data feeding, shown in Figure 5(a). Each time it fetches four small consecutive blocks (e.g., VEC bytes of data, or cachelines in HARP and HARPv2 platforms) and feeds them to the four parallel engines. The second is block data feeding, shown in Figure 5(b). It segments the entire input file into four large consecutive parts. Each time it fetches one (or multiple) cachelines from one of the four parts and feeds them into each compression engine.



(a) Cyclic data feeding



(b) Block data feeding

Figure 5: Data feeding from CPU to FPGA Deflate accelerator

Due to the fact that the similar data usually locate in nearby regions in a file, the compression ratio (measured by input file size divided by output file size) of the block data feeding is much higher than that of cyclic data feeding. Actually, the block feeding method suits the Deflate algorithm perfectly to compress large files since the Deflate format limits history distance to 32K bytes. For files much larger than 32K bytes, strings in the latter part will not be compared to the previous part with a distance longer than 32K bytes. Therefore, it makes minor impacts on compression ratio to segment large files for block data feeding, because only the compression of data strings in block boundaries might be affected. Compared to cyclic data feeding, the degradation on compression ratio due to block segmentation is negligible. On the other hand, cyclic data feeding offers better streaming capability. However, block data feeding can work as well for streaming applications by processing a collection of streaming data each time, for example, on the order of a few megabytes. Considering these, we will use the block data feeding as our default data feeding method.

3.2 Single Engine Optimization

In this subsection we will mainly present our new optimizations for the single compression engine to improve compression ratio, clock frequency, and resource utilization efficiency. These optimizations mainly focus on the major bottleneck stages 2 and 3 in Section 2.2.2, which 1) map the input strings that will be stored to the memory banks; 2) obtain and map the read results of previous strings from the memory banks for the inputs; and 3) compare the input strings and previous strings to get the match length and distance results. With all the optimizations, our single compression engine is fully pipelined, with an pipeline initial interval of one.

3.2.1 Hash Chain Implementation

As presented in Section 3.1, even given the block data feeding method, the compression ratio will drop to some extent—which means we need to compensate for the potential compression ratio loss in a single engine. Therefore, we increase the history dictionary size to find a better match. To achieve this, a hash memory chain, as shown in Figure 6, is implemented in the hash memory update stage. It is like a shift register, but in more of a register file format. Every cycle different depths of the chained memory all return a candidate string as the read output, and the current depth's read output is the input written into its next depth's memory. The substring in the current window will be stored into the memory in

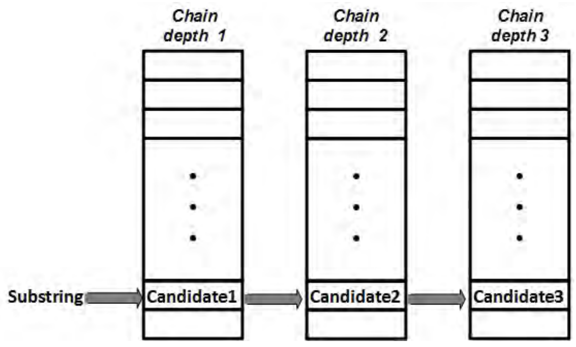


Figure 6: Hash memory chain for each hash history

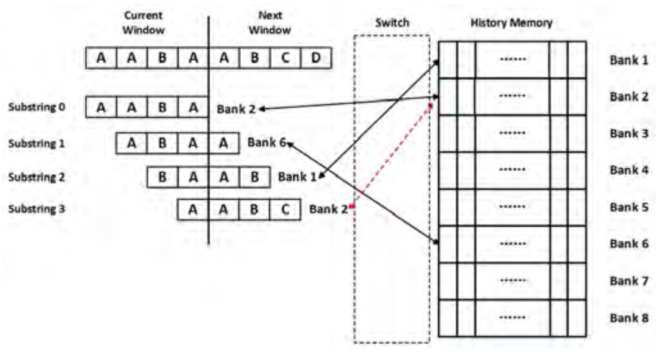


Figure 7: Hash (bank) conflict solution 2: double bank in this paper

the first depth of the chain. In the example, candidate 1, 2 and 3 are the output of each memory at different depths of the memory chain, and all of them will be compared to the substring to find the best match. After that, the current substring will be stored into chain depth 1, candidate 1 will be stored into chain depth 2, and so on.

3.2.2 Double Clock Design vs. Double Bank Design

To reduce the compression ratio drop caused by bank conflicts, Microsoft [10] uses a second clock whose frequency is twice the global clock to read and update the 16 (VEC) hash memory banks (dictionaries), which we refer as design 1. As will be evaluated in Section 5, the doubled clock frequency of the memory part will become the bottleneck of the whole design and significantly limits the system performance, especially when we integrate the idea of hash memory chain. We propose a second design to use a single clock while doubling the number of banks to 32 (2*VEC), as shown in Figure 7. As a result, we increase the length of the hash dictionary. This approach avoids the doubled clock frequency bottleneck and enables us to integrate more depth of hash chain.

3.2.3 Switch Optimization

As presented in Section 2.2.2, we implement the switches with multiplexers whose inputs are strings and select signals are the hash values. For the first set of switches that map the input strings which will be stored to the memory banks, we can use one 16-to-1 (VEC=16) multiplexer (MUX) for each hash chain as shown in Figure 8. To avoid the timing bottleneck, we pipeline the 16-to-1 MUXes into 2 stages using 4-to-1 MUXes.

For the second set of switches that map the read results of previous strings from the memory banks for the inputs, it is more complex. A straightforward way is using a 32-to-1 128 bit-width input MUX to select the record string from the 32 output ports of memory banks (hash table) for each depth of the chain, and then compare it with the input string, as shown in Figure 9. This way ensures that for those strings which cannot access the memory (e.g., bank conflict, their mapped banks have been accessed) are still able to measure if they are matched with the previous strings. Another way is to leave those strings as mismatched literals and just com-

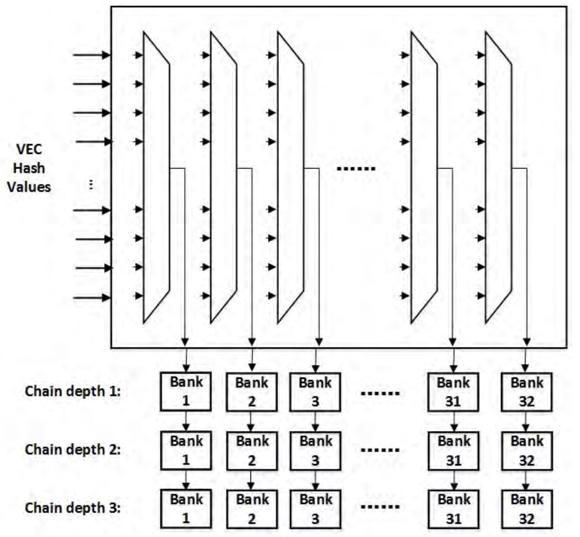


Figure 8: Input mapping to hash memory chain, i.e., memory banks

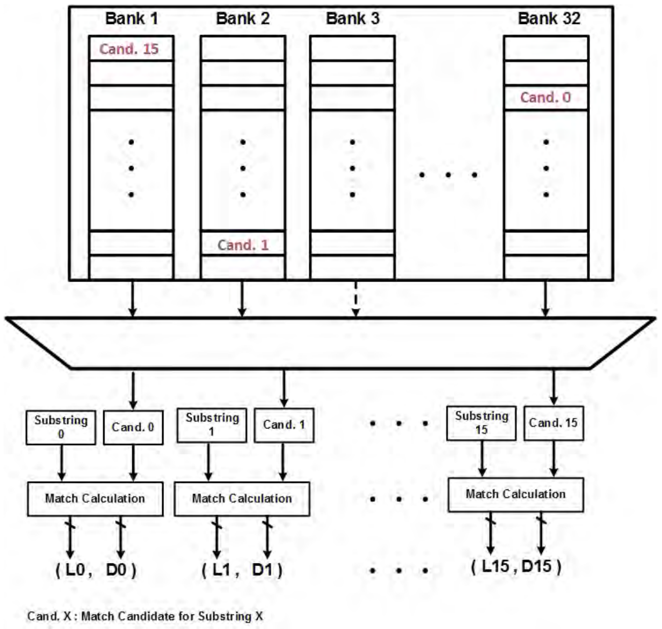


Figure 9: Memory bank (hash table) mapping to input option 1: string matching after 128-to-1 MUX

pare output port data of memory with the data delayed by 1 cycle from its input port, as shown in Figure 10. If the bank has not been accessed, (e.g., Banks 3), then the match calculation unit will generate a pair of (L=0, D=0) to indicate the case. The compared result is simply a 5-bit length data, which is much shorter than the original 128-bit data string. So we can use 5-to-1 MUXes to select the corresponding hash match results for each input string. In this design we employ the second option to avoid the overhead of multiplexers (reduce it from 128-to-1 MUX to 5-to-1 MUX) to improve the timing and resource consumption. The compression ratio only degrades by 3% since we have actually mapped the 16 input strings to 32 memory banks (e.g., design 2) or conceptually due to its double clock frequency (design 1). The second way eliminates the timing bottleneck even when we scale further up. For example, when we scale to process 32 bytes per cycle, we only need to use 32 6-to-1 MUXes, which introduces much less overhead and does no harm to the clock frequency of the design.

4. SYSTEM INTEGRATION

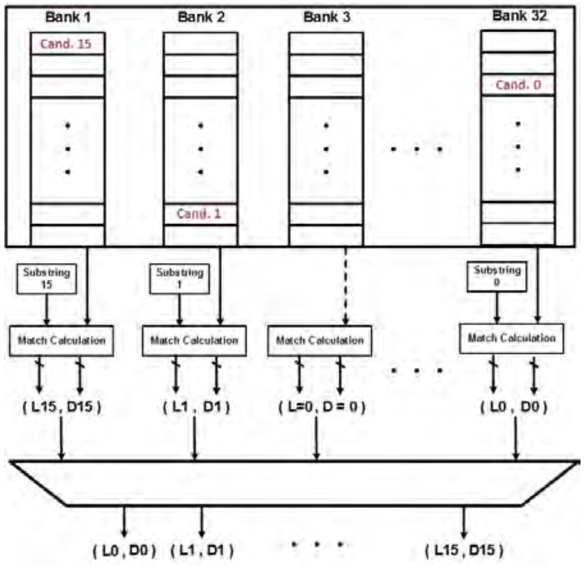


Figure 10: Memory bank (hash table) mapping to input option 2: 5-to-1 MUX after string matching

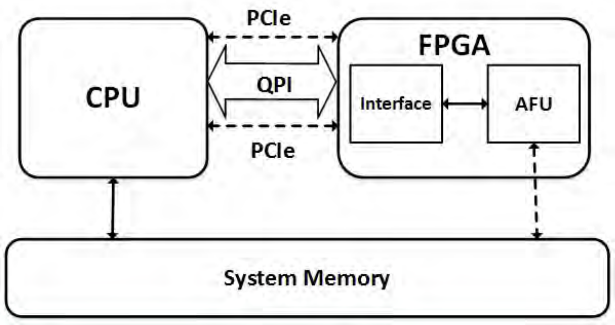


Figure 11: Intel-Altera HARP and HARPv2 CPU-FPGA architecture

In this section we first introduce some of the background on the recent Intel-Altera HARP [17] and HARPv2 [18] CPU-FPGA platforms that we leverage to achieve high end-to-end compression throughput. Then we present the CPU-FPGA communication flow and implementation of our multi-way accelerator architecture in the tightly coupled Intel-Altera HARPv2 CPU-FPGA platform. The integration flow is similar to the old-generation HARP platform.

4.1 CPU-FPGA Platform

The mainstream PCIe-based CPU-FPGA platforms use direct memory access (DMA) for an FPGA to access the data from a CPU. First, the FPGA needs a memory controller IP to read the data from the CPU's DRAM to its own DRAM through PCIe. And then the FPGA performs specified acceleration in its accelerator function units (AFUs). In fact, this DRAM-to-DRAM communication through PCIe can have a limited bandwidth in practice; for example, this practical PCIe bandwidth can be only around 1.6 GB/s (although the advertised bandwidth is 8 GB/s) according to the study in [11]. This makes it impractical to implement full-speed acceleration even if we have a high throughput compression accelerator on the FPGA side.

The recent Intel-Altera HARP platform uses the QuickPath Interconnect protocol (QPI). HARPv2 uses one QPI channel and two PCIe channels within a single package to connect the CPU cores and FPGA accelerators, as shown in Figure 11. And the CPU and FPGA can communicate using the shared memory to significantly increase the CPU-FPGA communication bandwidth. AFU can read/write data directly from/to system virtual memory through the core-cache interface (CCI). This architecture makes FPGA have

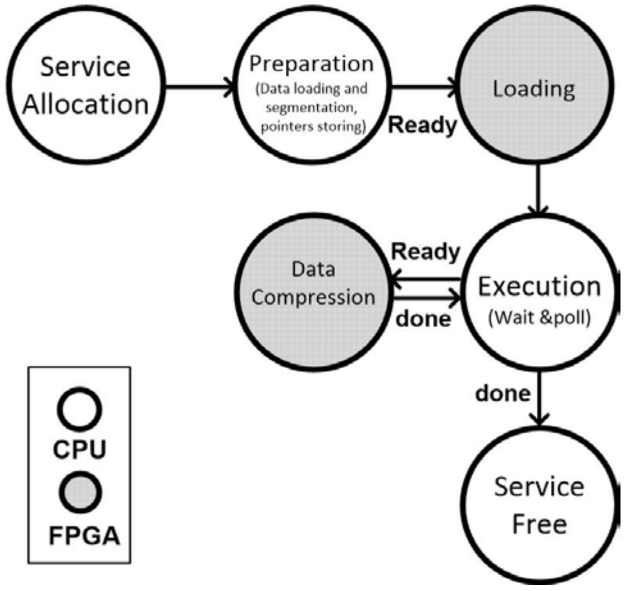


Figure 12: Flow chart of CPU-FPGA system execution

first-class access to the system memory and thus achieve a high-bandwidth and low latency for CPU-FPGA communication. According to the study in [11], HARP can provide 7.0 GB/s FPGA read bandwidth and 4.9 GB/s FPGA write bandwidth. Using the same benchmark method in [11], we find that HARPv2 has further improvement and can provide more than 15 GB/s FPGA read and write bandwidth. Such high CPU-FPGA communication bandwidth provides opportunities to achieve a much higher end-to-end compression throughput in practice.

Moreover, this shared system memory architecture eliminates explicit memory copies and increases the performance of fine-grained memory accesses. This proves to be rather useful in our multi-way parallel design where we need to read/write to multiple random memory locations in consecutive cycles.

4.2 CPU-FPGA Communication Flow and Interface Design

Figure 12 presents the flow of compression execution. At initialization, the CPU first allocates memory workspace which is shared by the CPU and the FPGA, as shown in Figure 13. This includes device status memory (DSM) workspace to keep track of the status of the FPGA accelerator, a source buffer for storing the source data, and a destination buffer to store the processed output data. After loading the source file into the source buffer, the CPU segments the source file equally into N blocks (four in our example) and then writes the segmented base addresses and block sizes to the FPGA. The initial setting being done, the CPU enters a wait state and polls the status bit in DSM until the FPGA sends an ending signal. On the other side, the FPGA accelerator begins execution after loading the addresses and source data block sizes. Each engine is allowed an equal share of time to send read requests for source data from the shared memory using a round-robin arbiter. Since read responses are out of order in the HARPv2 platform, re-order buffers are used to restore source data order before compressing. The processed results are kept temporarily in first-in-first-out (FIFO) buffers before writing back to the destination buffer in the shared memory. Another arbiter is used to control write access among the Deflate engines. After all the source data are processed, the FPGA will signal the CPU that all work has been done.

In HARPv2, the data transfer size for each read/write request can be 1,2 or 4 consecutive cache lines (64 bytes per cache line); and for maximum CPU-FPGA data bandwidth, our design uses 4 cache lines per request while other options are also supported. Read/write requests generated by the accelerated function unit (AFU) are

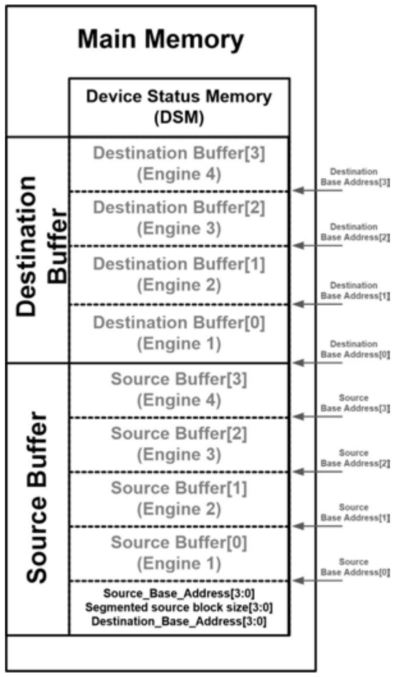

Main Memory
Device Status Memory (DSM)
Destination Buffer
Destination Buffer[3] (Engine 4)
Destination Buffer[2] (Engine 3)
Destination Buffer[1] (Engine 2)
Destination Buffer[0] (Engine 1)
Destination Base Address[3]
Destination Base Address[2]
Destination Base Address[1]
Destination Base Address[0]
Source Buffer
Source Buffer[3] (Engine 4)
Source Buffer[2] (Engine 3)
Source Buffer[1] (Engine 2)
Source Buffer[0] (Engine 1)
Source Base Address[3]
Source Base Address[2]
Source Base Address[1]
Source Base Address[0]
Source_Base_Address[3:0]
Segmented source block size[3:0]
Destination_Base_Address[3:0]

Figure 13: Shared memory space CPU-FPGA system

compliant with the core cache interface (CCI-P) which is the interface between the AFU and FPGA interface unit (FIU). The FIU is formed by Intel QPI and PCIes and implements the interface protocols for links between CPU and FPGA. The three physical links (one QPI and two PCIes) are configured to one virtual channel in our accelerator design, and the FIU will optimize the communication bandwidth by steering requests among the three physical links.

5. EVALUATION

5.1 Experimental Setup

We test our FPGA Deflate accelerators on the HARPv2 [18] platform (major platform) and use the Calgary Corpus [5] datasets as the benchmarks to measure the average compression throughput and compression ratio. To show how CPU-FPGA communication bandwidth may limit end-to-end throughput and avoid FPGA hardware generation gap, we also test some of the designs on the HARP platform as a comparison since HARP still has a very limited communication bandwidth and it uses the same Stratix V FPGA as previous studies [4, 10].

HARP Platform. HARP platform integrates an Intel Xeon E5-26xx v2 processor and an Altera Stratix V 5SGXEA7 FPGA module.

HARPv2 Platform. HARPv2 platform integrates a 14-core Broadwell EP CPU and an Altera Arria 10 GX1150 FPGA module.

5.2 Comparison of Deflate Accelerators

Since most prior studies only measure the performance of the FPGA Deflate accelerator, we first compare our accelerator design to state-of-the-art studies in Table 1. By using the novel multi-way parallel and fully pipelined accelerator architecture where each way features a single clock, 32 banks and a memory chain depth of 3 designs that can process 16 bytes per cycle, we can compress 64 bytes/cycle (4-way parallel) at a clock frequency of 200 MHz. This is more scalable than merely scaling up the data window size of a single Deflate engine, since the area increases nonlinearly with the data window size, and the maximum frequency the engine can achieve drops as a result of routing problems. In summary, our FPGA Deflate accelerator achieves a record compression throughput of 12.8 GB/s, which is 2.2x faster than the prior record

Table 1: FPGA Deflate accelerator comparison

Design	Frequency	Throughput	Compression ratio	Area(ALMs)	Efficiency (MB/s per kilo ALMs)
IBM [9]	250 MHz	4 GB/s	2.17	110,000	36
Altera [4] (Stratix V)	193 MHz	2.84 GB/s	2.17	123,000	23
Microsoft [10] (Stratix V)	175 MHz	5.6 GB/s	2.09	108,350	52
This work on HARP (Stratix V)	200 MHz	9.6 GB/s	2.05	134,664	71.3
This work on HARPv2 (Arria 10)	200 MHz	12.8 GB/s	2.03	162,828	78.6

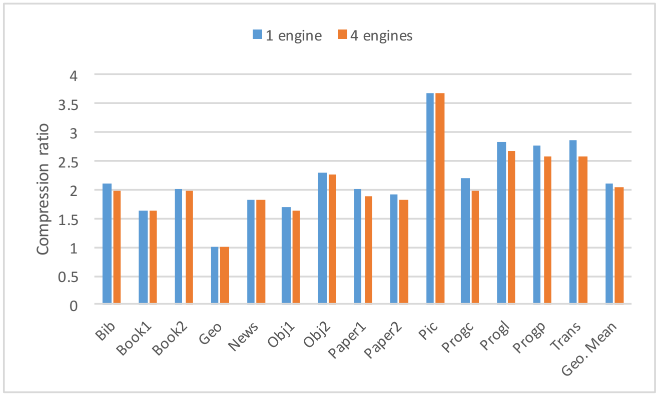


Figure 14: Compression ratio when scaling from 1 engine to 4 engines

in [10]. In addition, our design is also much more resource-efficient in terms of compressed MB/s per kilo ALMs, which is 1.5x efficient than the prior record in [10]. Please also note we measure our designs at a fixed clock frequency of 200 MHz, since the platform provides the fixed clock. In fact, due to our optimizations, our FPGA kernel can work at an even higher clock frequency.

The compression ratio drops by 4.7% compared to Microsoft's work [10], but it is still acceptable. This drop is due to the fact we divide the input file into four parts to feed into four different compression engines. If the data in one part should be matched better with the data in another part, then in our case the compression ratio drops.

To be fair about the comparison, we also break down the compression accelerator kernel speedup into two parts: improvement from the accelerator design and improvement from the FPGA technology advancement.

5.2.1 *Improvement from Accelerator Design*

Since previous studies [4, 10] both implement their designs on Stratix V FPGA, we also list the experimental result on HARP, which also uses Stratix V. Due to the total resource limitation, we can only put three Deflate engines on it. Still, we can achieve a compression throughput of 9.6 GB/s, which is 1.7x faster and 1.4x more efficient in performance per area over the record design [10]).

5.2.2 *Improvement from Technology Advancement*

HARPv2 uses a more advanced Arria 10 FPGA, which provides more area and enables us to integrate one more Deflate engine than HARP Stratix V FPGA. As a result, we can achieve 12.8 GB/s compression throughput on HARPv2. Table 2 lists the FPGA resource utilization on the HARPv2 platform, where ALMs are the main contributor. Note that HARPv2 interface itself occupies an additional 20% of the FPGA resource.

Table 2: FPGA resource utilization on HARPv2

Resources	Amount
ALMs	162,828 (38%)
Registers	248,188 (15%)
Block Memory Bits	16,177,152 (29%)

5.3 Scaling Effect

Figure 14 shows the corresponding compression ratio of Calgary Corpus benchmarks as we change the parallel engine number from 1 to 4. The single engine (VEC=16) achieves an average compression ratio of 2.11. When we increase the throughput to 12.8 GB/s with 4 engines, the average compression ratio drops by 3.7%. Note that the compression ratio of those large files (e.g., book1, book2, news and pic) only degrades by less than 1%. This result proves the analysis we present in Section 3.1 that the multi-way parallel Deflate compressor perfectly suits the applications where large files need to be compressed.

Table 3: Scaling results on HARPv2

Parallel Engine No	Throughput	Compression ratio	Area (ALMs)
1	3.2 GB/s	2.11	38,297
2	6.4 GB/s	2.07	78,604
3	9.6 GB/s	2.05	118,891
4	12.8 GB/s	2.03	162,828

Table 3 lists the area usage (and compression ratio) for scaling up engines, and the total area usage increases roughly linear. Considering that the area when we directly change the data window size of a single compression engine from 16 to 32 bytes will increase the ALM usage by 2.6x, it is also difficult to keep the frequency to a high value. This becomes the bottleneck for scaling in some cases where the system needs to run at a fixed clock frequency. For example, on HARP the system needs to run at 200 MHz. Exploiting the parallel engines can avoid this scaling problem as each engine is designed and placed separately.

5.4 Design Tradeoff Evaluation

The single compression engine design is important as it determines the baseline performance for the whole design. Thus, we explore the single engine design space to choose the best.

5.4.1 *Memory Chain Depth*

Table 4: Memory chain depth on HARPv2

Depth	Compression ratio	Area(ALMs)
1	1.92	31,360
2	2.04	34,761
3	2.10	38,297

We first compare the performance of different hash chain depths for a single engine in Table 4. Increasing one depth only augments 3,500 more ALMs, thanks to the MUX optimization, and there is more than a 9% improvement on the compression ratio, increasing depth from 1 to 3. Note that further increasing memory chain depth will not benefit the system performance much, and the compression ratio gain becomes marginal (less than 1.5% improvement measured). In fact, the dictionary strings read from each depth need to be compared with the same input substring simultaneously, so matching comparison of the input string and the output dictionary string from the deepest memory bank will become the critical path.

5.4.2 *Double Clock vs. Double Bank*

As presented in Section 3.2.2, a double clock design (design 1) is also implemented whose memory part uses a clock frequency that is twice the global clock. We use the same number of bank numbers as VEC (i.e., 16 banks), set the memory chain depth to be 3, and integrate 4 parallel engines. Table 5 summarizes the two designs. For the double clock design, since BRAM blocks are placed as arrays and the interconnect between two adjacent arrays will consume more time, the memory chain which occupies more than one single array can only run at a frequency of 340 MHz under the fastest configuration, thus limiting the system performance. Since

Table 5: Double Clock v.s. Double Bank on HARPv2

Design	Frequency	Throughput	Compression ratio	Area(ALMs)	Efficiency (MB/s per kilo ALMs)
Double clock design	150 MHz	9.6 GB/s	2.10	115,436	83
Double bank design	200 MHz	12.8 GB/s	2.03	162,828	78.6

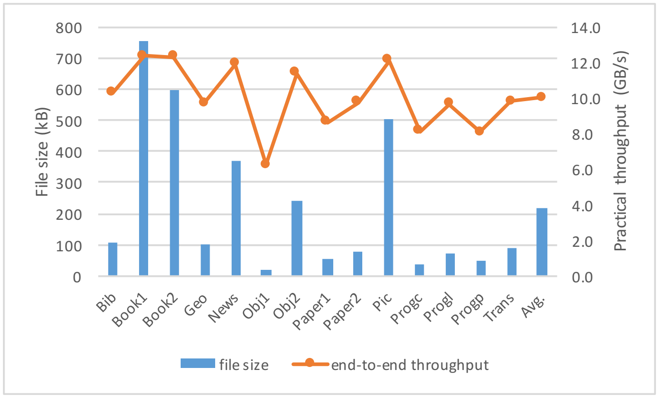


Figure 15: Compression throughput and ratio on HARPv2, under different file size of Calgary Corpus datasets

HARPv2 provides another clock group of 150/300 MHz, we take 150 MHz as the system clock for the design and achieve the equivalent throughput of 9.6 GB/s, which is only 75% of the performance of the single clock design with double banks.

However, the area efficiency of the double clock design is slightly better than the double bank design since it reduces the use of memory banks and corresponding multiplexers. Another interesting benefit of the double clock design is that it gives a slightly better compression ratio. This is because raising clock frequency while reducing memory banks enables similar strings to have more chances at being mapped to the same bank.

5.5 End-to-End Compression Throughput

We measure the end-to-end compression throughput by inserting a counter on the FPGA side. The counter counts the total clock cycles from when FPGA sends the first request to read the memory to the time it sets the data valid bit in the memory—which indicates the FPGA work is done and all of the data has been written back to the memory. Note that on HARP and HARPv2, the CPU and FPGA share the same memory so that there is no additional memory copy overhead on conventional PCIe-based platforms. Denote the total clock cycles as t_{cycle} and the total amount of data fed to the compression engine as A_{data} bytes, then the end-to-end compression throughput can be measured as A_{data} /t_{cycle}.

The end-to-end overall compression throughput results are shown in Table 6. To show the overhead of CPU-FPGA communication, we first test our design on HARP, where the QPI bandwidth is limited (7 GB/s for read and 4.9 GB/s for write). We integrate 3-way parallel engines on HARP since the Stratix V FPGA module on HARP provides less resources and cannot accommodate more engines. The FPGA kernel throughput is 9.6 GB/s, while we can only achieve an average of 3.9 GB/s practical throughput based on the Calgary Corpus benchmarks.

Table 6: End-to-end compression throughput on HARP and HARPv2

Platform	FPGA Throughput	Practical Throughput
HARP	9.6 GB/s	3.9 GB/s
HARPv2	12.8 GB/s	10.0 GB/s

On the other hand, even considering the read and write latency, we can see that HARPv2 can achieve an average end-to-end compression throughput of 10.0 GB/s since it has over 15 GB/s CPU-FPGA communication bandwidth. We also list each benchmark's size and its corresponding practical throughput in Figure 15. If the file size is larger than 500 kB (e.g., book1, book2 and pic), then the overhead of communication latency will be negligible and the end-to-end throughput can be up to 12 GB/s.

6. CONCLUSION

In this work we designed an FPGA Deflate accelerator that can be easily scaled to achieve a record compression throughput of 12.8 GB/s while maintaining a relatively high compression ratio of 2.03. We presented methods for efficiently feeding data into the parallel compression engines, improving the resource utilization, augmenting compression ratio, and improving the clock frequency of the single fully pipelined compression engine. This is the first public work to integrate the compression accelerator with Intel-Altera HARP and HARPv2 platforms, where we leverage the high CPU-FPGA communication bandwidth to achieve high end-to-end system compression throughput. The end-to-end compression throughput we achieve is 3.92 GB/s on HARP and 10 GB/s (12 GB/s) on HARPv2. This shows that our compression accelerator design is a great fit for the new HARP and HARPv2 platforms.

References

[1] P. Deutsch, "DEFLATE Compressed Data Format Specification version 1.3," http://www.rfc-base.org/txt/rfc-1951.txt, 1996, [Online; accessed 23-Apr-2017].
[2] "Zlib Compression Library," http://www.zlib.net/, [Online; accessed 23-Apr-2017].
[3] "Gzip file format specification version 4.3," https://tools.ietf.org/html/rfc1952, [Online; accessed 23-Apr-2017].
[4] M. S. Abdelfattah, A. Hagiescu, and D. Singh, "Gzip on a Chip: High Performance Lossless Data Compression on FPGAs Using OpenCL," in *Proceedings of the International Workshop on OpenCL 2014*, ser. IWOCL '14. ACM, 2014, pp. 4:1–4:9.
[5] "The Calgary Corpus," http://corpus.canterbury.ac.nz/descriptions/#calgary, [Online; accessed 23-Apr-2017].
[6] V. Gopal, J. Guilford, W. Feghali, E. Ozturk, and G. Wolrich, "High Performance DEFLATE Compression on Intel Architecture Processors," http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-deflate-compression-paper.pdf, 2011, [Online; accessed 23-Apr-2017].
[7] "AHA3642 Compression Core," http://www.aha.com/DrawProducts.aspx?Action=GetProductDetails&ProductID=38, 2014, [Online; accessed 23-Apr-2017].
[8] B. Sukhwani, B. Abali, B. Brezzo, and S. Asaad, "High-Throughput, Lossless Data Compresion on FPGAs," in *Proceedings of the 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '11, 2011, pp. 113–116.
[9] A. Martin, D. Jamsek, and K. Agarwal, "Fpga-based application acceleration: Case study with gzip compression/decompression streaming engine," in *Special Session 7C, IEEE International Conference on Computer-Aided Design*, 2013.
[10] J. Fowers, J.-Y. Kim, D. Burger, and S. Hauck, "A Scalable High-Bandwidth Architecture for Lossless Compression on FPGAs," in *The 23rd IEEE International Symposium on Field-Programmable Custom Computing Machines*, May 2015.
[11] Y.-k. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei, "A quantitative analysis on microarchitectures of modern cpu-fpga platforms," in *Proceedings of the 53rd Annual Design Automation Conference*, ser. DAC '16, 2016, pp. 109:1–109:6.
[12] M. Huang, D. Wu, C. H. Yu, Z. Fang, M. Interlandi, T. Condie, and J. Cong, "Programming and runtime support to blaze fpga accelerator deployment at datacenter scale," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, ser. SoCC '16. ACM, 2016, pp. 456–469.
[13] J. Cong, Z. Fang, M. Huang, L. Wang, and D. Wu, "Cpu-fpga co-scheduling for big data applications," vol. PP, no. 99, 2017, pp. 1–4.
[14] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, May 1977.
[15] D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, Sept 1952.
[16] J.-Y. Kim, S. Hauck, and D. Burger, "A Scalable Multi-Engine Xpress9 Compressor with Asynchronous Data Transfer," in *Proceedings of the 2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '14, 2014, pp. 161–164.
[17] "Xeon+FPGA Platform for the Data Center," https://www.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf, [Online; accessed 23-Apr-2017].
[18] "Accelerating Datacenter Workloads," http://fpl2016.org/slides/Gupta%20--%20Accelerating%20Datacenter%20Workloads.pdf, [Online; accessed 23-Apr-2017].