Gzip on a Chip: High Performance Lossless Data Compression on FPGAs using OpenCL

Mohamed S. Abdelfattah*
Dept. of Electrical & Computer Engineering
University of Toronto
10 King's College Rd.
Toronto, ON, Canada
mohamed@ece.utoronto.ca

Andrei Hagiescu and Deshanand Singh
Altera Toronto Technology Center
150 Bloor Street West
Toronto, ON, Canada
{ahagiesc,dsingh}@altera.com

ABSTRACT

Hardware implementation of lossless data compression is important for optimizing the capacity/cost/power of storage devices in data centers, as well as communication channels in high-speed networks. In this work we use the Open Computing Language (OpenCL) to implement high-speed data compression (Gzip) on a field-programmable gate-arrays (FPGA). We show how we make use of a heavily-pipelined custom hardware implementation to achieve the high throughput of ~3 GB/s with more than 2× compression ratio over standard compression benchmarks. When compared against a highly-tuned CPU implementation, the performance-per-watt of our OpenCL FPGA implementation is 12× better and compression ratio is on-par. Additionally, we compare our implementation to a hand-coded commercial implementation of Gzip to quantify the gap between a high-level language like OpenCL, and a hardware description language like Verilog. OpenCL performance is 5.3% lower than Verilog, and area is 2% more logic and 25% more of the FPGA's available memory resources but the productivity gains are significant.

1. INTRODUCTION

In this paper, we study the implementation of hardware data compression on FPGAs motivated by its potential for use both in data centers and in communication networks. A recent report [22] stated that companies are spending 12% of their IT budget on storage and this cost is doubling every two years. This trend motivates the usefulness of lossless data compression in data centers to reduce the size of stored data and the associated cost and energy of hard disks and memories [13]. In communication networks there is a similar need for lossless data compression to reduce the size of transmitted data over a communication channel and therefore use its bandwidth more efficiently [3, 19]. For both of these applications, fast data compression is required to keep up with disk read/write speeds or the speed of communication networks.

*With Altera Toronto Technology Center during this work.

IWOCL '14, May 12–13 2014, Bristol, United Kingdom.

http://dx.doi.org/10.1145/2664666.2664670

DEFLATE [21] is a very popular standard for lossless compression algorithms. It forms the basis for widely used compression packages such as Gzip, pkzip and zlib. Given the need for high performance implementations to keep up with network and storage device data rates, several previous works have addressed implementations targeting high throughput. These implementations can be classified in 3 categories:

- **CPU Implementations** – Intel [4] demonstrated a highly optimized implementation of DEFLATE running on a Core™ i5 650 processor at 3.20 GHz. The results showed that a single core could sustain a data rate of 2.7 Gigabits/s with an average compression ratio of 2.18x.
- **FPGA Implementations** – Hardware implementations of FPGA-based DEFLATE algorithms have been previously studied in the literature [14, 18, 20, 23]. The best known results have been reported in a recent work by IBM researchers [18]. Their implementation on an Altera StratixV A7 FPGA is able to sustain a throughput of 3 Gigabytes/s [15]. Note that the IBM implementation is the highest performance hardware implementation that has been publicly reported.
- **ASIC Implementations** – Several companies have created dedicated ASIC devices [6, 7, 9–11] that address the needs of high performance lossless compression. The AHA3642 [11] provides the best reported results: 20 Gigabits/s compression throughput while providing a 3.6x compression ratio. Intel's 89xx series of communication chipsets also offers similar levels of performance [10].

The highest performing implementations above were based on either ASIC or FPGA technology. While ASICs are the most efficient, the circuits are completely fixed and cannot be changed after the chip is produced. FPGAs provide a level of configurability that enables custom implementations suited to the particular needs of an application. For example, compression cores can be integrated with a variety of different interfaces which communicate with external devices. In addition, various algorithmic choices can be altered depending on the characteristics of the data being processed. This flexibility may enable even higher levels of compression than a more general fixed approach. However, previous FPGA implementations were written in a hardware description language such as Verilog HDL or VHDL which are akin to assembly language for hardware. This makes FPGA design time-consuming and dif-

ficult to verify. Instead, this paper proposes an OpenCL [17] implementation of Gzip. OpenCL is a 'C'-based language intended for application acceleration on heterogeneous systems. We demonstrate that the use of OpenCL for FPGA implementation enables incredible productivity gains while maintaining high efficiency as the generated system matches or exceeds the speed and compression quality of prior work. The OpenCL implementation of Gzip will be available for public download[1] and should serve as valuable framework for researchers to quickly explore variations of lossless compression algorithms on FPGAs.

In the following section we briefly describe the Gzip compression algorithm. Section 3 summarizes important features of the Altera SDK for OpenCL [12] and the design architecture. Section 4 describes the implementation and optimization of Gzip for FPGA, and Section 5 outlines preliminary compression quality and performance results.

2. GZIP

Gzip implements the DEFLATE algorithm for compression; this consists of two parts, LZ77 compression and Huffman encoding.

2.1 LZ77 Compression

This compression algorithm replaces repeat occurrences of bits with a reference to their previous occurrence [24]. Consider Fig. 1 for instance. The algorithm traverses the sentence serially, one character at a time in this case, and looks for repetitions. When " word " is found the second time, it is replaced with a pointer to the previous occurrence of it. this pointer consists of a marker @, match length and match offset. The match length is the length of the match being replaced – in this case it is 6 bytes including spaces. The match offset is the distance to the previous occurrence of the word.

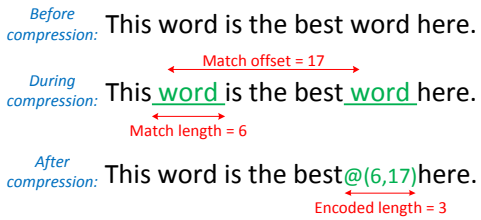

Figure 1: Example of LZ77 compression.

In LZ77, we only replace our matched word with a pointer if it results in compression. That means that the (marker/length/offset) must be smaller than the word being replaced – words of length 3 bytes are therefore never replaced for instance.

2.2 Huffman Encoding

After LZ77 compression, the uncompressed portion of the data is encoded using *dynamic* Huffman codes. This is slightly different from DEFLATE which encodes the length/offset from LZ77 with Huffman codes as well. Huffman encoding [16] replaces symbols (typically 1 byte but could be any length) in a data stream with codes such that the total size of the stream is reduced. The Huffman algorithm creates these codes optimally by constructing a Huffman tree [16]. This tree assigns shorter codes for more frequently occurring symbols, and longer codes for symbols that are seldom found. If this tree is updated for each new data set, this is known as a dynamic Huffman tree. Alternatively a static Huffman tree can be constructed based on statistical knowledge of the input data once, but this reduces compression quality. In our implementation, we create the Huffman tree based on the input data and we control how often we update it – this is described in more detail in Section 3.

[1] `http://www.altera.com/support/examples/opencl/opencl.html`

3. HARDWARE ARCHITECTURE

3.1 Altera SDK for OpenCL

Altera's SDK for OpenCL is a framework for using FPGAs to accelerate computation [12]. Altera's OpenCL compiler targets heterogeneous systems consisting of processors and FPGA devices. Common configurations include FPGA acceleration cards in server systems where the PCIe is the physical layer for communication. This compiler performs two main functions to create an FPGA system [5]. First, it automatically generates the platform IP that is required for communication between the host CPU and FPGA. In addition, it automatically creates the memory controllers and interconnect for FPGA communication with external memory. Second, the compiler translates OpenCL code to a hardware kernel circuit that executes on the FPGA. The connectivity between the generated hardware kernels and the platform IP is automatically handled without any manual intervention from the programmer.

Several new features of the compiler guided the design of Gzip and allowed the design to be completed in less than four weeks; we itemize them here:

- **Emulator** – This feature allows for functional verification of OpenCL algorithms before moving to FPGA implementation. Rather than waiting for full synthesis, placement and routing to be complete (which often takes hours), a programmer can experiment with algorithms on their desktop and have very fast iterations to debug their code.
- **Optimization report** –This feature uses static program analysis information to guide the user as to what the bottlenecks in their implementation will be. Common examples would include highlighting of complex loop carried dependencies which can slow down the processing rate of an OpenCL kernel.
- **Profiler** – In a similar way, the Profiler uses dynamic information from instrumented hardware to highlight performance bottlenecks. Useful information such as memory accesses with poor bandwidth efficiency are shown by this tool.

In OpenCL one can explicitly specify the level of parallelism through selecting the number of threads (or work-items) – this is supported by Altera's OpenCL compiler. However, the compiler also supports "single-threaded" OpenCL code; this is essentially C-code. The user does not specify the number of threads, nor does s/he specify whether variables go in private or local memory; instead, the compiler *infers parallelism automatically* and promotes/demotes variables between private and local memory automatically to better accelerate the application. This "single-threaded" code is more suitable for applications in which parallelism cannot be explicitly specified and there exists dependencies within the kernel. We use the "single-threaded" code for DEFLATE

because the algorithm is inherently serial (data is traversed byte by byte and is compared against previous bytes to look for matches) and the single-threaded code was a much more natural way to express the complex dependencies that exist in this algorithm.

3.2 Design Architecture

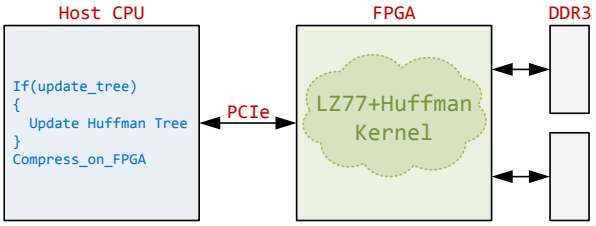

Figure 2: OpenCL system architecture.

Fig. 2 shows the overall architecture; the host portion of the implementation selects a byte to represent the marker and a suitable Huffman tree; it also sends to the FPGA the Huffman tree along with the data to be compressed. The host modifies the Huffman tree only if the "update_tree" flag is set; a new tree improves compression quality when the new data being compressed contains a different set of bytes.

The implementation in this paper uses an x86 based system as the host processor. However, we note that this implementation is extendable to standalone FPGA systems such as the case where data streams in and out of the FPGA through Ethernet cables. In these environments, new generations of FPGAs with embedded hard processors [8] can easily replace the functionality of the x86 host.

The kernel architecture is based on a Verilog implementation by IBM presented at ICCAD 2013 [18].

4. IMPLEMENTATION AND OPTIMIZATIONS

In this section we discuss implementation details and show how OpenCL code translates to hardware. Additionally, we describe performance and compression ratio optimizations that we performed.

4.1 Shift in Data and Compute Hash

The first step is to load new data from global (DDR3) memory – the loaded data is then stored in on-chip registers. Listing 1 shows the details of how this is done. The on-chip register array "current_window" buffers data that is currently being processed. First, we shift the second half of "current_window" into its first half, then load VEC bytes from the global memory buffer "input".

```
1 //shift current window
2 #pragma unroll
3 for(char i = 0; i < VEC; i++)
4   curr_window[i] = curr_window[VEC+i];
5
6 //load in new data
7 #pragma unroll
8 for(char i = 0; i < VEC; i++)
9   curr_window[VEC+i] = input[inpos+i];
```

Listing 1: Load 16 bytes per cycle.

Fig.3 illustrates "current_window" before and after loading in new data. This will then allow us to process VEC substrings each cycle, by extracting portions of the "current_window" (each of length LEN) as shown. The parameter LEN determines the maximum match length possible in LZ77 compression.

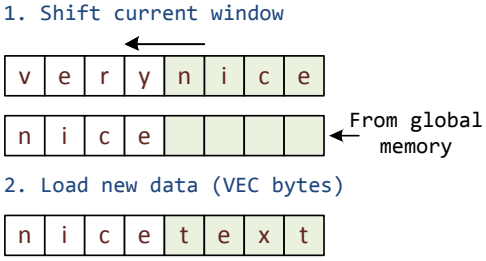

Figure 3: Shifting in new data into "current_window". (VEC=4)

Next, we want to search the previous text for substrings that partially or fully match the "current_window" substrings. Previous text is buffered in history dictionaries in local memory (in OpenCL terminology) or FPGA block RAM (BRAM). To lookup these dictionaries for possible matches, we use a hash value corresponding to each of the "current_window" substrings. A very simple hash function just uses the first byte of the substring thus guaranteeing that the dictionaries return candidate matches that all start with the same byte. For example, if the "current_window" substring is "nice", hash[nice] = n (remember that each letter is just an 8-bit ASCII number).

This simple hash function has two shortcomings: first, it creates an 8-bit value, that means it can only index 256 addresses while the dictionary BRAMs (M20Ks on Stratix-V FPGAs) have a depth of 1024 words. Second, it only contains information about the first byte (out of VEC bytes) in the current substring and hence the candidate matches returned from the dictionaries only resemble the "current_window" substrings in the first letter. To overcome these two shortcomings we experimented with the hash function and found that the following function improves absolute compression ratio by ~7% on average compared to the aforementioned simple hash:

$$\begin{aligned} hash[i] = {} & (curr\_window[i] \ll 2) \\ & xor\,(curr\_window[i+1] \ll 1) \\ & xor\,(curr\_window[i+2]) \\ & xor\,(curr\_window[i+3]) \end{aligned}$$

The improved hash function XOR's the first four bytes of the current substring, with the first byte shifted left by two bits, and the second byte shifted left by one bit. This creates a 10-bit hash value that is able to index the full depth of the M20K BRAM, and incorporates information about the first 4 bytes as well as their ordering. In testing different hashing functions, the emulator that came with the Altera OpenCL compiler was very useful as it allows testing the algorithm fairly quickly on the CPU.

4.2 Dictionary Lookup and Update

Using the hash value computed for each substring, we lookup candidate matches in history dictionaries. These history dictionaries buffer some of the previous text on-chip, and are implemented as a hierarchy of BRAMs – this local memory hierarchy is key to the high throughput of our LZ77 implementation. To look for matches

quickly, there are VEC dictionaries, each buffers some of the previous text. Using the hash value, each substring looks for a candidate match in each of the dictionaries; resulting in VEC candidate matches for each current substring.

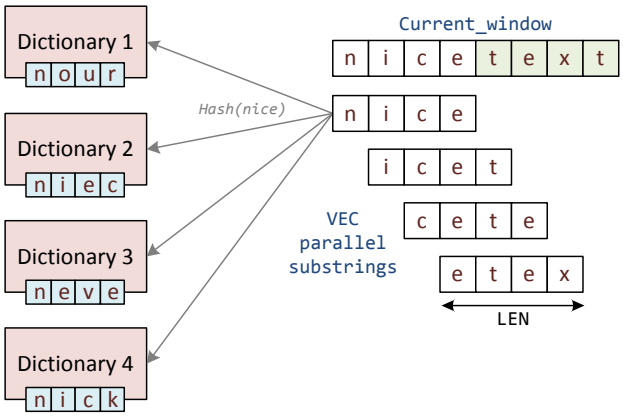

Figure 4: Each substring looks up candidate matches in VEC dictionaries. Each dictionary returns a candidate substring from history that resembles the current substring. (VEC=4)

The example in Fig. 4 shows that the word “nice" finds “nour", “nice", “neve" and “nick" as candidate matches – each of those comes from a different dictionary and they all occurred in the text before our current substring “nice". This lookup is repeated for the other substrings “icet", “cete" and “etex" as described in Listing 2 – “pragma unroll" on the outer loop tells the compiler to replicate the read ports on each dictionary BRAM so that there are exactly as many read (or write) ports on the physical BRAM to provide conflict-free accesses. In our case, each dictionary has VEC read ports and one write port. The inner loop in Listing 2 specifies the width of the read ports. In this case “pragma unroll" tells the compiler to coalesce the memory accesses into one wide access of LEN bytes per read/write port, and the generated on-chip memory supports that width to be able to load/store each substring in one access. In our implementation with VEC=16 and LEN=16, this local memory topology can load 64 16-byte substrings and store 16 16-byte substrings each cycle.

```
//loop over VEC current window substrings
#pragma unroll
for(char i = 0; i < VEC; i++)
 //load LEN bytes
 #pragma unroll
 for(char j = 0; j < LEN; j++)
 {
  comp_window[j][0][i]=dict_0[hash[i]][j];
  comp_window[j][1][i]=dict_1[hash[i]][j];
  ...
  comp_window[j][15][i]=dict_15[hash[i]][j];
 }
}
```

Listing 2: Lookup candidate matches in history dictionaries.

The result of dictionary lookup is a set of candidate matches for each current substring stored in an array “compare_window" – this is used in the following step to look for matches for LZ77 compression. After dictionary lookup we update the dictionaries with the current substrings such that each substring is stored in a different dictionary. In Fig. 4 for instance, “nice" will be stored in dictionary 1, “icet" in dictionary 2, “cete" in dictionary 3 and “etex" in dictionary 4.

4.3 Match Search and Reduction

In this step, each “current_window" substring is compared to its candidate matches in “compare_window" and a match length is computed for each one as illustrated in Fig. 5. The “length" array in Fig. 5 and Listing 3 contains the number of matching bytes from the start of each current and compare windows. The largest value is then chosen and stored in the “bestlength" array – there is now one bestlength value for each “current_window" substring.

```
//loop over each comparison window
#pragma unroll
for(char i = 0; i < VEC; i++)
{
 //loop over each current window
 #pragma unroll
 for(char j = 0; j < VEC; j++)
 {
  //compare current/comparison windows
  #pragma unroll
  for(char k = 0; k < LEN; k++)
  {
  if(curr_window[j+k]==comp_window[k][i][j]
    && !done[j])
    length[j]++;
  else
    done[j] = 1;
  }
 }
 //update bestlength
 #pragma unroll
 for(char m = 0; m < VEC; m++)
  if(length[m] > bestlength[m])
   bestlength[m] = length[m];
}
```

Listing 3: Compare each “current_window" substring to its candidate matches in “compare_window" and find the length of each match.

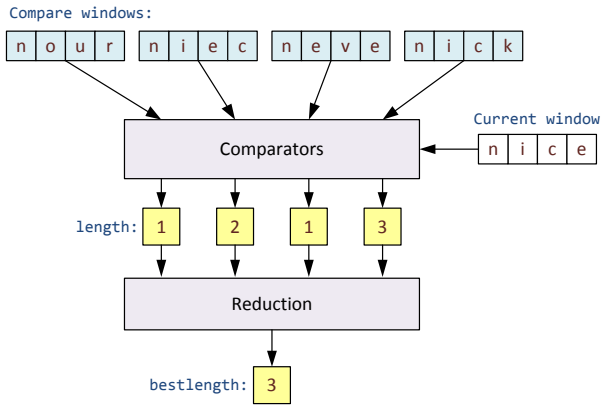

Figure 5: Each “current_window" substring is compared to its candidate matches in “compare_window" and the best match length is selected. (VEC=4)

Bad Coding Style

The code in Listing 3 consists of three nested loops; the innermost loop does a byte-by-byte comparison to find the match length. Listing 4 shows functionally equivalent code that works better on a

CPU but does not translate into efficient hardware. The reason is that the while-loop bounds cannot be determined statically so the compiler will not be able to determine how many replicas of the comparison hardware it needs to create. The compiler issues a warning and only one replica of the hardware is created – the loop iterations share this hardware and execute serially on it causing a big decrease in throughput.

```
//compare current/comparison windows
#pragma unroll
while(curr_window[j+k]==comp_window[k][i][j
    ])
  length[j]++;
```

Listing 4: Typical C-code targeting CPU processors does not necessarily compile into efficient hardware.

Area Optimization

Listing 5 demonstrates a subtle area optimization. The if-statement in Listing 3 gets translated by the compiler into a long vine of adders and multiplexers as shown in Fig. 6. This is because we need to know both the value of "length" and the condition of the if-statement before finding the new value of "length". Listing 5 removes this problem by using the OR-operator instead of addition to store the match length. Since the ordering of the OR operations doesn't matter, the compiler can create a balanced tree to group the operations on "length_bool" together. This reduces area for two reasons: first, it creates a shallower pipeline depth, meaning fewer registers and FIFOs will be required to balance the kernel pipeline. Second, shifters and OR gates require less resources than adders and multiplexers.

```
//compare current/comparison windows
#pragma unroll
for(char k = 0; k < LEN; k++)
{
 if(curr_window[j+k]==comp_window[k][i][j])
  length_bool[i][j] |= 1 << k;
}
```

Listing 5: An area efficient implementation of the innermost comparison loop in match search.

The resulting "length_bool" now contains an array of ones (and zeroes) instead of an actual number – if "length" was 3 for instance, "length_bool" will be 0111 where the ones indicate which bytes were equal between the current and compare window substrings. We leverage this one-zero encoding of "length_bool" (instead of the binary encoding of "length") in selecting the best length which results in further area savings. Overall, this area optimization reduces total logic utilization by ~31k logic elements, or 5% of the Stratix-V A7 FPGA device.

4.4 Match Filtering

The previous step creates a "bestlength" array of length VEC; each entry corresponds to one of the "current_window" substrings. The match filtering step now picks a valid subset of the "bestlength" array such that it maximizes compression; it consists of four steps:

1. Remove "bestlength" matches that are longer when encoded than the original. In Fig. 7 "bestlength[1]" is removed because its LZ-encoding will consist of 3 bytes at least (for marker,length,offset).

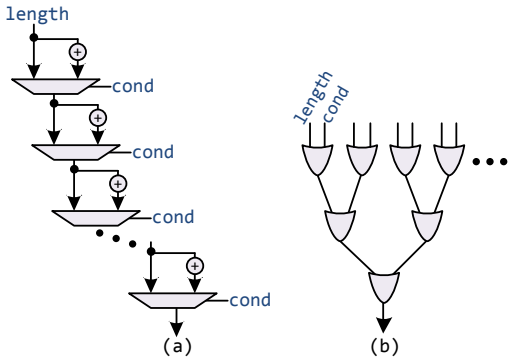

Figure 6: The code in Listing 3 produces a vine of adders/multiplexers as shown in (a), while using an OR-operator in Listing 5 allows the compiler to create a balanced tree of gates that uses lower FPGA resources.

2. Remove "bestlength" matches covered by the previous step. In Fig. 7 "bestlength[0]" is removed because the 'n' was already part of a code in the previous loop iteration.

3. Select non-overlapping matches from the remaining ones in "bestlength". We implement a bin-packing heuristic for this step; the one we choose to implement is "last-fit"; this selects the last match first ("bestlength[3]") then removes any "bestlength" that covers it ("bestlength[2]").

4. Compute the "first_valid_position" for the next step. This depends on the "reach" of the last used match – in the example in Fig. 7 the last match covers bytes 0, 1 and 2 in the following cycle so the "first_valid_position" in the following cycle is 3 as shown.

Loop-carried Computation

A variable is **loop-carried** whenever it is computed in loop iteration x and only used in the next loop iteration $x+1$. In our application, one of the loop-carried variables is "first_valid_position". In hardware this is implemented as a feedback connection between a later stage in the pipeline to an earlier stage in the pipeline as shown in Fig. 8a – "first_valid_position" is fed-back from stage 4 to stage 2 in match filtering. If this feedback path takes more than one cycle, this loop-carried computation causes the kernel pipeline to stall until it is completed.

Fig. 8b shows the kernel pipeline executing over time assuming that the "first_valid_position" computation takes three cycles. Loop iteration 2 is stalled in the first step until "first_valid_position" from loop iteration 1 is computed in the fourth step; this causes pipeline bubbles as illustrated in Fig. 8b. This also means that we can only start new loop iterations every three cycles – the *initiation interval* (II) of the loop is 3. For any FPGA design, our target should be to optimize this loop-carried computation such that we get an initiation interval equal to 1; this avoids pipeline bubbles – a design with II=1 has triple the throughput of a design with II=3. For a stallable pipeline such as the one generated by OpenCL; the loop-carried computations can be thought of as the critical path of the design.

In our application, the computation in Fig. 8a resulted in II=6; the Altera OpenCL compiler optimization report informs the user

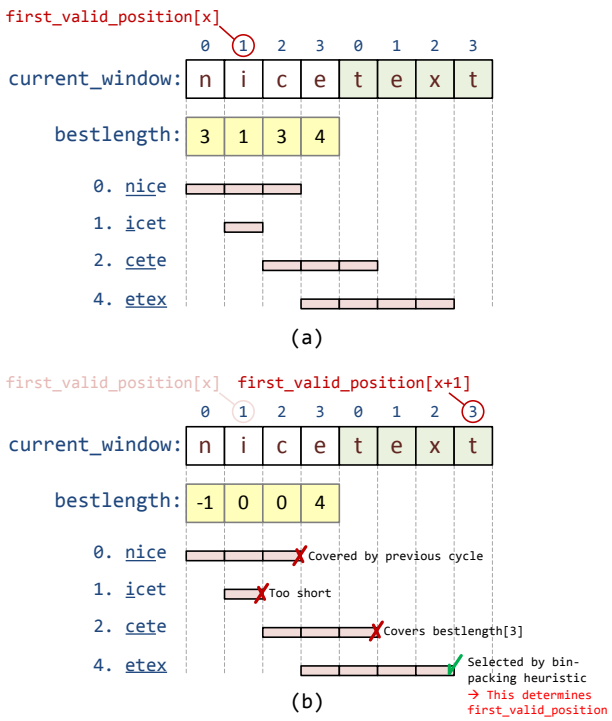

Figure 7: Bestlength array a) before, and b) after filtering. (VEC=4)

of the loop's II and points to the problematic variable in the user's source code – in this case it pointed to "first_valid_position. To optimize the computation, we take the bin-packing heuristic that filters "bestlength" off of the critical path by moving it after "first_valid_position" computation as shown in Fig. 8. This leads to an II=1 as desired, meaning we can process a new loop iteration every cycle, instead of every 6 cycles, effectively increasing throughput six-fold. However, we now have a design constraint on the bin-packing heuristic; it cannot alter the value of "first_valid_position" to maintain correctness. In other words, the "bin-packing" heuristic must always select the last valid entry of "bestlength" as shown in Fig. 7 – this is why we use last-fit bin-packing.

Bin-packing Heuristic

Both the hash function described in Section 4.1 and the bin-packing heuristic described here is a critical factor in determining compression ratio. Our loop-carried computation dictated the use of last-fit; however, it is very inefficient. This is why we optimize the last-fit heuristic by removing all the matches that have the same *reach* but smaller value than an existing match. Reach is how far the match extends and is computed as reach[i] = i+bestlength[i] as shown below. This eliminates inefficient "bestlength" entries without changing "first_valid_position" and results in an improvement of absolute compression ratio by ~8%.

$$\begin{aligned} bestlength[i] &= \{8,7,6,5\} \\ vanilla\ lastfit: bestlength[i] &= \{0,0,0,5\} \\ optimized\ lastfit: reach[i] &= \{8,8,8,8\} \\ optimized\ lastfit: bestlength[i] &= \{8,0,0,0\} \end{aligned}$$

4.5 Huffman Encoding

The final step is to encode the LZ77 symbols with Huffman symbols. The LZ77 algorithm produces two types of symbols: unencoded text (single-byte) and matches (multi-byte). Due to how we construct the matches, the total number of symbols to encode each iteration does not exceed $2 \cdot VEC$. We limit the length of each Huffman encoded symbol to 16 bits. This ensures that the result of the encoded stream fits in an array of $4 \cdot VEC$ bytes. The main challange is to concatenate the encoded symbols into contiguous data segments. Because the symbols can have an arbitrary number of bits, each symbol can shift to any position in the output segment. Also, the location where a symbol is shifted to depends on the location and length of the previous symbol.

```
uint code[2 * VEC];
ushort next[2 * VEC], segment[2 * VEC];
#pragma unroll
for (int i = 0; i < 2 * VEC; i++) {
  uchar bitPos = pos[i] % 16;
  code[i] = hufenc[data[i]] << bitPos;
}
#pragma unroll
for (int i = 0; i < 2 * VEC; i++) {
  next[i] = 0;
  segment[i] = 0;
  uchar bytePos = pos[j] / 16;
  #pragma unroll
  for (int j = 0; j < 2 * VEC; j++)  {
     bool upper = bytePos % (2 * VEC) == i;
     bool lower = bytePos % (2 * VEC) == i - 1;
     ushort crt = upper ? (code[j] >> 16) :
                  (lower ? code[j] : 0);
     bool useLater = (bytePos >= 2 * VEC) ||
           (lower && (bytePos >= 2 * VEC - 1);
     segment[i] |= useLater ? 0 : crt;
     next[i] |= useLater ? crt : 0;
  }
}
```

Listing 6: Bit-alignment of huffman codes

We defer writing an encoded segment to memory until it contains $4 \cdot VEC$ bytes, to ensure aligned memory accesses. Shorter segments will be updated during the subsequent loop iterations. We also carry across loop iterations the offset of the first empty bit in the segment. We track the location of each output symbol within the segment by adding the bit-length of all previous symbols. If an iteration completes a segment, it will write it to memory and start a new one.

We describe an architecture that shifts the encoded symbols to arbitrary bit positions within a segment. We use barrel shifters to pre-shift each symbol by amounts between 0 and 15. Next, we determine the destination of each pre-shifted symbol. Due to the pre-shifting, the destinations are aligned to 16-bit boundaries. We describe the placement of symbols into the segment as a fully unrolled loop nest. The outer loop iterates over all 16-bit locations in the segment while the inner loop iterates over all candidate symbols *crt*. Note that due to the pre-shift, the candidates are 32-bits while the segment locations are 16-bits. We split the symbol into an *upper* and a *lower* part which will land on consecutive locations. When a location match is identified, the value is or-ed with the current contents of that location, as multiple encoded symbols may land on disjoint bits from the same location. Our approach is described in Listing 6.

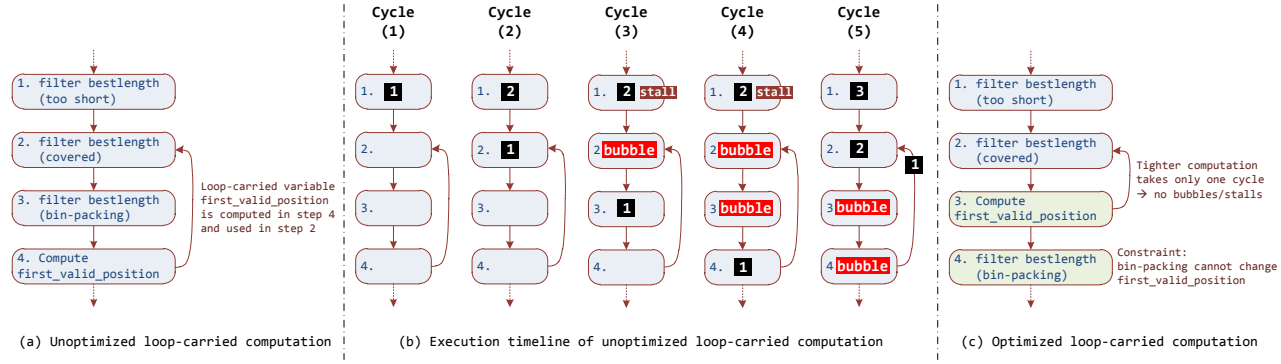

Figure 8: Loop-carried dependencies may cause a kernel pipeline to stall thus reducing throughput. By optimizing the loop-carried computation a high-throughput pipeline can be created.

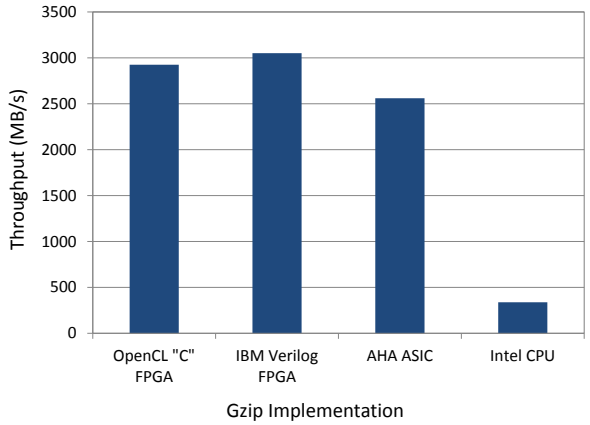

Figure 9: Comparison against commercial implementations of Gzip on FPGA, ASIC and CPU.

5. RESULTS AND COMPARISON

Fig. 9 compares our achieved throughput with the best known commercial implementations of Gzip on FPGAs [18] ASICs [11], and CPUs [4]. As the plot shows, our implementation is only about 5.3% slower than the best known FPGA implementation and 12% faster than the fastest commercial ASIC implementation. However, note that the ASIC implementation reports an average compression ratio of 3.6× on the Canterbury corpus [2], whereas our FPGA implementation achieves 2.43× on the same benchmark set. This is attributed to both the expert knowledge of the industrial vendor that we are comparing to, as well as the higher area budget available to ASICs. In the following subsections we focus on comparing to Intel's highly tuned CPU (OpenCL is 12× better), and IBM's hand-coded Verilog implementations to better understand the tradeoffs across design platforms (FPGA vs. CPU) and design abstractions (OpenCL vs. Verilog).

5.1 FPGA versus CPU

We compare our OpenCL FPGA implementation to the fastest known CPU implementation of Gzip hand-tuned by Intel engineers and makes use of hyper-threading [4]. Our implementation uses a 28-nm Stratix-V A7 FPGA device (25 W) while the CPU measurements were performed on a 32-nm Intel Core i5 650 processor (73 W for 2 cores) running at 3.2 GHz.

Table 1: Comparison between our OpenCL FPGA and the best CPU implementation of Gzip.

	Performance	Performance per Watt	Compression Ratio
OpenCL FPGA	2.84 GB/s	116 MB/J	2.17×
Intel Gzip	338 MB/s	9.26 MB/J	2.18×
Gap	8.5× faster	12× better	on par

Performance

Table 1 compares the performance, performance-per-watt and compression ratio across the two platforms. Even though the optimized CPU implementation runs at 3.2 GHz, it takes 9.6 cycles on average to process one byte. On the other hand, our FPGA implementation runs at a clock frequency that is approximately 20 times slower but is able to process 16 bytes every cycle making it 8.2× higher throughput, or 12× better when normalized to power consumption.

Compression Ratio

To evaluate compression ratio, we test our hardware with the Calgary corpus [1] to be able to compare to Intel's results. The geometric mean compression ratio over the corpus yields almost the same result thus validating our compression ratio with an industrial standard such as Intel's implementation. Note that our goal was to create a reference design with high throughput; compression ratio can be further improved by implementing smarter hashing functions for dictionary lookup/update, or by improving the match selection heuristic.

5.2 OpenCL versus Verilog

To evaluate the Altera OpenCL compiler, we compare our OpenCL implementation of DEFLATE to the Verilog implementation of IBM on which our design architecture is based [18]. This comparison quantifies the performance-efficiency vs. productivity tradeoff that a high-level language such as OpenCL offers – we expect a Verilog implementation to be of somewhat higher performance and efficiency because it is implemented in a low-level language that gives the user very fine control over the design. For the same reason, we expect OpenCL to have a much lower design effort and thus higher productivity.

Table 2: Comparison between OpenCL and Verilog for Gzip compression.

	Performance	Efficiency	Productivity
OpenCL Kernel	2.84 GB/s 193 MHz	47% logic 70% RAM	High (1 month)
Verilog	3.0 GB/s 200 MHz	45% logic* 45% RAM*	Low
Gap	5.3% slower	2% more 25% more	– –

*conservative area estimate based on chip image. [18]

We summarize the results in Table 2. Note that the designs are not identical, and that our design is a work-in-progress, and we are certain that both performance and efficiency can still be improved; however, we compare the results attained after 1 month of work on this reference design to be able to evaluate the performance/efficiency vs. productivity tradeoff of OpenCL compared to Verilog.

Performance

Both the Verilog and OpenCL implementations process 16 bytes per cycle but they run at different frequencies. To measure frequency and performance, we compile the design five times with different seeds and select the best one. The Verilog implementation runs "just under 200 MHz" [18], while our openCL implementation runs at 193 MHz – this causes throughput to be 5.3% lower for our implementation as shown in Table 2.

Efficiency

Table 2 also lists the area breakdown of the two designs; both designs are implemented on the same Stratix-V A7 chip. While the design architecture is more-or-less the same, the generated OpenCL kernel (for this case study) is a bit different to the hardware. The pipeline depth for the Verilog implementation is only 17 cycles [18] while our OpenCL kernel has a pipeline depth of 87 cycles. The throughput is agnostic to the pipeline depth, hence the OpenCL compiler makes heavy use of the ample registers available on modern FPGAs to optimize frequency as much as possible. However, this process is very conservative as it must account for any combination of operations in a kernel. Furthermore, a side-effect of this deep pipelining is that parts of the kernel are more heavily pipelined than others. To balance the pipeline, the OpenCL compiler automatically inserts FIFOs on the paths with lower latency – this is the main reason behind the discrepancy in the RAM utilization stated in Table 2. Because our kernel is very wide and includes many reductions, many FIFOs need to be inserted on paths that require fewer pipeline stages. Fig. 10 illustrates the pipelining added to a reduction tree where all the intermediate results are used downstream. This partly accounts for the higher logic and RAM utilization as some of these FIFOs end up in registers while the larger FIFOs utilize BRAM. Of course it is also expected that a highly-tuned commercial implementation to have smaller area than our OpenCL kernel. To quantify the area gap, we compute the relative increase in area of OpenCL compared to Verilog (Table 2 shows absolute increase of chip memory and logic resources) and take the geometric mean between logic and RAM resources – the OpenCL implementation has 7% higher area utilization compared to the tuned Verilog implementation of IBM.

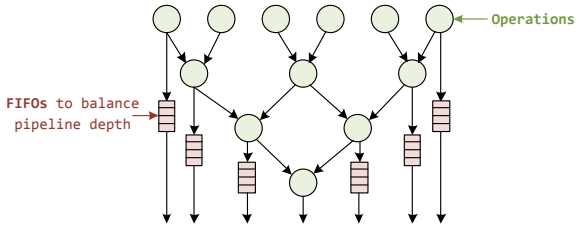

Figure 10: A reduction tree showing the insertion of FIFOs on paths with fewer stages to balance the pipeline depth.

Productivity

Compared to Verilog, performance only drops by 5.3%, and even though area is increased by 2% logic and 25% memory we believe OpenCL makes a compelling case for hardware designers. Similarly to how standard-cell ASIC design flow is typically used instead of full-custom layout for microelectronic circuits, we believe that hardware designers will migrate to the use of high-level languages like OpenCL for most designs. With OpenCL, this kernel was coded in one week and optimized in the following three weeks. OpenCL essentially makes hardware design as easy as writing software code. The more concise and portable C code is used instead of Verilog, and the emulator makes it very easy to test and verify algorithm modifications (such as different hashing functions).

6. CONCLUSION

We demonstrate that using a high-level compiler we can achieve competitive performance for GZIP compression, and significant productivity gains compared to traditional hardware design. The compiler's new features allowed fast iterations for various architectures, allowing the user to focus on the algorithm details. We were able to achieve a compression ratio of 2.17× on the Calgary corpus with throughput of 2.8 GB/s. This is only 5.3% lower than the best known hardware implementation of Gzip. Compared to the best CPU implementation, OpenCL FPGA performance-per-watt is 12× better. We aim to release our implementation as a reference design which can be improved even more than the reported results.

References

[1] Calgary Corpus. http://www.data-compression.info/Corpora/CalgaryCorpus/index.htm.

[2] Canterbury Corpus. http://corpus.canterbury.ac.nz/descriptions/.

[3] Hardware based GZIP Compression, Benefits and Applications. http://www.comtechaha.com/Uploads/GZIP-Benefits-Apps.pdf, 2008.

[4] High Performance DEFLATE on Intel Architecture Processors. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-deflate-compression-paper.pdf, 2011.

[5] Compiling OpenCL to FPGAs : A Standard and Portable Software Abstraction for System Design. http://www.fpl2012.org/keynote3.shtml, 2012.

[6] GZIP HW Accelerator. http://www.inomize.com/index.php/content/index/gzip-hw-accelerator, 2012.

[7] GZIP/GUNZIP Silicon IP Family. http://www.sandgate.com/new/static/QuickZIP%

20Family%20Product%20Brief%20%28V1.2a%29.pdf, 2012.

[8] Altera SoCs: When Architecture Matters. http://www.altera.com/devices/processor/soc-fpga/overview/proc-soc-fpga.html, 2013.

[9] GX 1700 Series. http://www.exar.com/common/content/document.ashx?id=21282&languageid=1033, 2013.

[10] Scaling Acceleration Capacity from 5 to 50 Gbps and Beyond with Intel QuickAssist Technology. http://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/scaling-acceleration-capacity-brief.pdf, 2013.

[11] AHA3642. http://www.aha.com/DrawProducts.aspx?Action=GetProductDetails&ProductID=38, 2014.

[12] Altera. OpenCL for Altera FPGAs: Accelerating Performance and Design Productivity. http://www.altera.com/products/software/opencl/opencl-index.html, 2012.

[13] D. Craft. A fast hardware data compression algorithm and some algorithmic extensions. *IBM Journal of Research and Development*, 42(6), Nov 1998.

[14] M. El Ghany, A. Salama, and A. Khalil. Design and Implementation of FPGA-based Systolic Array for LZ Data Compression. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 3691–3695, May 2007.

[15] P. Hofstee. The Big Deal about Big Data. In *Proceedings of the 8th IEEE International Conference on Networking, Architecture, and Storage*, July 2013.

[16] D. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[17] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.

[18] A. Martin, D. Jamsek, and K. Agarwal. FPGA-Based Application Acceleration: Case Study with GZIP Compression/Decompression Streaming Engine . In *International Conference on Computer-Aided Design (ICCAD)*, Nov 2013.

[19] I. Papaefstathiou. Titan II: an IPComp processor for 10Gbit/sec networks. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, pages 234–235, Feb 2003.

[20] S. Rigler, W. Bishop, and A. Kennings. FPGA-Based Lossless Data Compression using Huffman and LZ77 Algorithms. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 1235–1238, April 2007.

[21] D. Salomon, G. Motta, and D. Bryant. *Data Compression: The Complete Reference*. Molecular biology intelligence unit. Springer, 2007.

[22] S. V. Smith. Big Data creates big industry for storing data. http://www.marketplace.org/topics/business/big-data-creates-big-industry-storing-data, 2013.

[23] M. Tahghighi, M. Mousavi, and P. Khadivi. Hardware implementation of a novel adaptive version of Deflate compression algorithm. In *Proceedings of the 18th Iranian Conference on Electrical Engineering (ICEE)*, pages 566–569, May 2010.

[24] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.