Energy Efficient Canonical Huffman Encoding

Janarbek Matai*, Joo-Young Kim†, and Ryan Kastner*
*Department of Computer Science and Engineering, University of California, San Diego
†Microsoft Research
{jmatai, kastner}@ucsd.com, jooyoung@microsoft.com

***Abstract*—As data centers are increasingly focused on energy efficiency, it becomes important to develop low power implementations of the various applications that run on them. Data compression plays a critical role in data centers to mitigate storage and communication costs. This work focuses on building a low power, high performance implementation for canonical Huffman encoding. We develop a number of different hardware and software implementations targeting Xilinx Zynq FPGA, ARM Cortex-A9, and Intel Core i7. Despite its sequential nature, we show that our hardware accelerated implementation is substantially more energy efficient than both the ARM and Intel Core i7 implementations. When compared to highly optimized software running on the ARM processor, our hardware accelerated implementation has approximately 15 times more throughput with 10% higher power usage, resulting in an 8X benefit in energy efficiency (measured in encodings/Watt). Additionally, our hardware accelerated implementation is up to 80% faster and over 230 times more energy efficient than a highly optimized Core i7 implementation.**

I. Introduction

Lossless data compression is a key ingredient for efficient data storage, and Huffman coding is amongst the most popular algorithm for variable length coding [1]. Given a set of data symbols and their frequencies of occurrence, Huffman coding generates codewords in a way that assigns shorter codes to more frequent symbols to minimize the average code length. Since it guarantees optimality, Huffman coding has been widely adopted for various applications [2]. In modern multi-stage compression designs, it often functions as a back-end of the system to boost compression performance after a domain-specific front-end as in GZIP [3], JPEG [4], and MP3 [5]. Although arithmetic encoding [6] (a generalized version of Huffman encoding which translates an entire message into a single number) can achieve better compression for most scenarios, Huffman coding is typically the algorithm of choice for production systems since developers do not have to deal with the patent issues surrounding arithmetic encoding [7].

Canonical Huffman coding has two main benefits over traditional Huffman coding. In basic Huffman coding, the encoder passes the complete Huffman tree structure to the decoder. Therefore, the decoder must traverse the tree to decode every encoded symbol. On the other hand, canonical Huffman coding only transfers the number of bits for each symbol to the decoder, and the decoder reconstructs the codeword for each symbol. This makes the decoder more efficient both in memory usage and computation requirements.

Data centers are one of the biggest users of data encoding for efficient storage and networking, which is typically run on high-end multi-core processors. This trend is changing recently with increased focus on energy efficient and specialized computation in data centers [8]. For example, IBM made a GZIP comparable compression accelerator [9] for their server system. We target the scenario where the previous stage of compressor (e.g., LZ77) produces multi-giga byte throughput with parallelized logic, which requires a high throughput, and ideally energy efficient, data compression engine. For example, to match a 4GB/s throughput, the Huffman encoder must be able to build up 40,000 dynamic Huffman trees per second assuming a generation of a new Huffman tree for every 100KB input data.

The primary goal of this work is to understand the tradeoff between performance and power consumption in developing a Canonical Huffman Encoder. In particular, we show the benefits and drawbacks of different computation platforms, e.g., FPGA, low-power processor, and high-end processor. To meet these goals, we design a number of different hardware and software implementations for Canonical Huffman Encoding. We developed a high performance, low power Canonical Huffman encoder based on high-level synthesis (HLS) tool for rapid prototyping. This is, to the best of our knowledge, the first hardware accelerated implementation of the complete pipeline stages of *Canonical Huffman Encoding (CHE)*. Additionally, we create highly optimized software implementations targeting an embedded ARM processor and a high-end Intel Core i7 processor. The specific contributions of this paper are:

1) The development of highly optimized software implementations of canonical Huffman encoding.
2) A detailed design space exploration for the hardware accelerated implementations using high-level synthesis tools.
3) A comparison of the performance and power/energy consumption of these hardware and software implementations on a variety of platforms.

The remainder of this paper is organized as follows: Section 2 presents related work. Section 3 provides algorithmic description of canonical Huffman encoding. In Section 4 and Section 5, we present detailed hardware and software implementations and optimizations, respectively. In Section 6, we present experimental results. We conclude in Section 7.

II. Related Work

Many previous works [10], [5], [11] focus on a hardware implementation for Huffman *decoding* because it is frequently used in mobile devices with tight energy budget, and the decoding algorithm has high levels of parallelism. On the other hand, Huffman *encoding* is naturally sequential, and it is difficult to parallelize in hardware. However, as we show in this paper, this does not mean that it is not beneficial

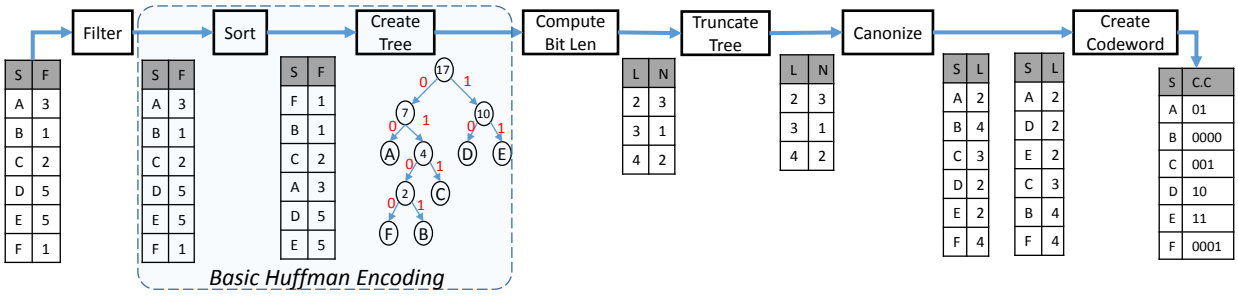

Fig. 1. The Canonical Huffman Encoding process. The symbols are filtered and sorted, and used to build a Huffman tree. Instead of passing the entire tree to the decoder (as is done in "basic" Huffman coding), the encoding is done such that only the length of the symbols in the tree is required by the decoder.

to pursue hardware acceleration for this application; when carefully designed, a hardware accelerator can be implemented in a high throughput and energy efficient manner.

There have been several hardware accelerated encoder implementations both in the ASIC and FPGA domain to achieve higher throughput for real-time applications [12], [13], [14], [15], [16]. Some designs [12], [13], [14], [16] focus on efficient memory architectures for accessing the Huffman tree in the context of JPEG image encoding and decoding. In some applications, the Huffman table is provided. In such cases the designs focused on the stream replacement process of encoding.

Some previous work develops a complete Huffman coding and decoding engine. For example, the work by Rigler et al [15] implements Huffman code generation for the GZIP algorithm on an FPGA. They employed a parent and the depth RAM to track the parent and depth of each node in tree creation. Recent work by IBM [9] designed GZIP compression/decompression streaming engine where they only implemented static Huffman encoder. This significantly lowers the quality of recompression.

This paper focuses on canonical Huffman encoding since our data center scenario must adapt the frequency distribution of new input patterns under the context of general data compression combining with Lempel-Ziv 77 [17]. This is well suited to canonical Huffman en coding. We provide comparison of a number of optimized hardware and software implementations on a variety of platforms and target energy efficiency.

III. Canonical Huffman Encoding (CHE)

In basic Huffman coding, the decoder decompresses the data by traversing the Huffman tree from the root until it hits the leaf node. This has two major drawbacks: it requires storing the entire Huffman tree which increases memory usage. Furthermore, traversing the tree for each symbol is computationally expensive. CHE addresses these two issues by creating codes using a standardized format.

Figure 1 shows the CHE process. The *Filter* module only passes symbols with non-zero frequencies. Then the encoder creates a Huffman tree in the same way as the basic Huffman encoding. The *Sort* module rearranges the symbols in ascending order based upon their frequencies. Next, the *Create Tree* module builds the Huffman tree using three steps: 1) it uses the two minimum frequent nodes as an initial sub-tree and generates a new parent node by adding their frequencies; 2) it adds the new intermediate node to the list and sorts them again; and 3) it selects the two minimum elements from the list and repeat these steps until one element remains. As a result, we get a Huffman tree, and by labelling each left and right edge to 0 and 1, we create codewords for symbols. For example, the codeword for A is 00 and codeword for B is 0101. This completes the basic Huffman encoding process. The CHE only sends the length of each Huffman codeword, but requires additional computation as explained in the following.

The *Compute Bit Len* module calculates the bit lengths of each codeword. It saves this information to a list where the key is length and value is the number of codewords with that length. In the example case, we have 3 symbols (A,D,E) with the code length of 2. Therefore, the output list contains L=2 and N=3. The *Truncate Tree* module rebalances the Huffman tree when it is very tall and/or unbalanced. This improves decoder speed at the cost of a slight increase in encoding time. We set the maximum height of the tree to 27.

Using output from the *Truncate Tree* module, the *Canonize* module creates two sorted lists. The first list contains symbols and frequencies sorted by symbol. The second list contains symbols and frequencies sorted by frequency. These lists are used for faster creation of the canonical Huffman codewords.

The *Create Codeword* module creates uniquely decodable codewords based on the following rules: 1) Shorter length codes have a higher numeric value than the same length prefix of longer codes. 2) Codes with the same length increase by one as the symbol value increases. According to the second rule, codes with same length increase by one. This means if we know the starting symbol for each code length, we can construct the canonical Huffman code in one pass. One way to calculate the starting canonical code for each code length is as follows: $for\ l = K\ to\ 1; Start[l] := [Start[l+1]+N[l+1]]$ where $Start[l]$ is the starting canonical codeword for a length l, K is the number of different code lengths, and $N[l]$ is the number of symbols with length l. In CHE, the first codeword for the symbol with the longest bit length starts all zeros. Therefore, the symbol B is the first symbol with longest codeword so it is assigned 0000. The next symbol with length 4 is F and is assigned 0001 by the second rule. The starting symbol for the next code length (next code length is 3) is calculated based on the first rule and increases by one for the rest.

In this paper, after calculating codewords, we do a bit re-

verse of the codeword. This is a requirement of the application on hand, and we skip the details due to space constraints.

The CHE pipeline includes many complex and inherently sequential computations. For example, the *Create Tree* module needs to track the correct order of the created sub trees, requiring careful memory management. Additionally, there is very limited parallelism that can be exploited. We designed the hardware using a high-level synthesis tool, and created highly optimized software for ARM and Intel Core i7 processors. In the following sections, we will report results, and highlight the benefits and pitfalls of each approach. We first discuss the hardware architecture and the implementation of the CHE design using HLS. Then we present the optimized software design of the CHE.

IV. Hardware Implementations

We created HLS architectures with different goals. *Latency Optimized* is designed to improve latency by parallelizing the computation in each module, and *Throughput Optimized* targets a high throughput design by exploiting task level parallelism. Since their block diagrams are very similar, we only present the block diagram of the *Throughput Optimized* architecture as shown in Figure 2. For the sake of simplicity, it only shows the interfaces with block rams (BRAMs). To create these designs (*Latency Optimized*, and *Throughput Optimized*), we start from a software C code which we name a *Baseline* design. Then we restructure parts of the code (*Restructured* design) as discussed below targeting efficient hardware architectures.

The input to the system is a list of symbols and frequencies stored in Symbol-Frequency (*SF*) BRAM. The size of *SF* is $48 \times n$ bits where 16 bits are used for symbol, 32 bits are used for frequency, and n is the number of elements in the list. The *Filter* module reads from the *SF* BRAM and writes the output to the next *SF* BRAM. Also it passes the number of non-zero elements to the *Sort* module. The *Sort* module writes the list sorted by frequency into two different *SF* BRAMs. Using the sorted list, the *Create Tree* module creates a Huffman tree and stores it into three BRAMs (*Parent Address*, *Left*, and *Right*). Using the Huffman tree information, the *Compute Bit Len* module calculates the bit length of each symbol and stores this information to a *Bit Len* BRAM. We set the maximum number of entries to 64, covering up to maximum 64-bit frequency number, which is sufficient for most applications given that our Huffman tree creation rebalances its height. The *Truncate Tree* module rebalances the tree height and copies the bit length information of each codeword into two different BRAMs with the size of 27, which is the maximum depth of the tree. The *Canonize* module walks through each symbol from the *Sort* module and assigns the appropriate bit length using the *BitLen* of each symbol. The output of the *Canonize* module is a list of pairs where list contains symbols and its bit lengths.

We implemented the algorithm on an FPGA using the Vivado High-level Synthesis (HLS) tool. The *Baseline* design has no optimizations. We developed a *Restructured* design on top of the *Baseline* design. After creating the restructured design, we optimize the restructured design for latency and throughput. To implement above hardware designs, we first profiled the algorithm on an ARM with different optimizations. Figure 6 shows initial (naive) running time of each modules of the design on an ARM processor. Among these, *Radix Sort*, *Create Tree*, *Compute Bit Length* are most computationally intensive. We focused our design space exploration on these sub modules and optimized them in HLS to generate an efficient design.

A. Radix Sort

The radix sorting algorithm arranges the input data for each radix from left to right (least significant digit) or right to left (most significant digit) in a stable manner. In a decimal system, the radix takes values from 0 to 9. In our system, we are sorting the frequencies, which are represented using a 32-bit number. We treat a 32-bit number as 4-digit number with radix $r = 2^{32/4} = 2^8 = 256$. In serial radix sort, the input data is sorted by each radix k times where k is radix ($k = 4$ in our case). Algorithm 1 describes the radix sorting algorithm which used counting sort to perform the individual radix sorts.

Algorithm 1 Counting sort

```
1:  HISTOGRAM-KEY:
2:  for i ← 0 to 2^r − 1 do
3:      Bucket[i] ← 0
4:  end for
5:  for j ← 0 to N − 1 do
6:      Bucket[A[j]] ← Bucket[A[j]] + 1
7:      temp[j] ← A[j]
8:  end for
9:  PREFIX-SUM:
10: First[0] ← 0
11: for i ← 1 to 2^r − 1 do
12:     First[i] ← Bucket[i − 1] + First[i − 1]
13: end for
14: COPY-OUTPUT:
15: First[0] ← 0
16: for j ← 0 to N − 1 do
17:     Out[First[i]] ← temp[j]
18:     First[i] ← First[i] + 1
19: end for
```

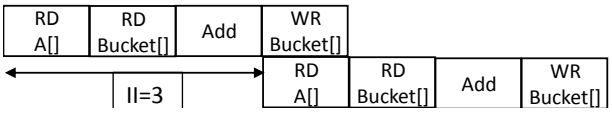

Fig. 3. A naively optimized code has RAW dependencies which requires an $II = 3$.

In order to implement parallel radix sort, we made two architectural modifications to the serial algorithm. First, we pipelined the counting sort portions (there are four counting sorts in the algorithm). This exploits coarse grained parallelism among these four stages of the radix sort architecture using the dataflow pipelining pragma in the Vivado HLS tool. Next we optimized the individual counting sort portions of the algorithm. In the current counting sort implementation there is a histogram calculation (Algorithm 1 line number 6). When synthesized with an HLS tool, this translates to an architecture which is similar to Figure 3. With this code, we achieve an

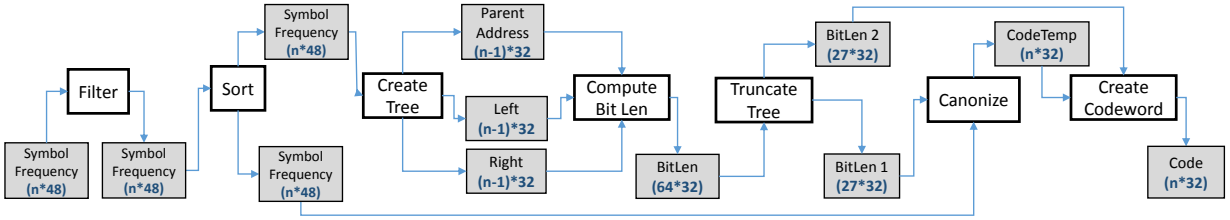

Fig. 2. The block diagram for our hardware implementation of canonical Huffman encoding. The gray blocks represent BRAMs with its size in bits. The white blocks correspond to the computational cores.

initiation interval (II) equal to 3 due to RAW dependencies. Ideally we want an $II = 1$. Since the histogram calculation is in a loop, achieving an $II = 1$ boasts performance by orders of magnitude. Achieving $II = 1$ requires an additional accumulator, which is shown in the pseudo HLS code in Listing 1. If the current and previous values of the histogram are the same, we increment the accumulator; otherwise we save the accumulator value to previous value's location and start a new accumulator for the current value. In addition to that, using dependency pragma, we instruct HLS to ignore RAW dependency.

```
1  #pragma DEPENDENCE var=Bucket RAW false
2  val = radix; //A[j]
3  if(old_val==val){
4    accu = accu + 1 ;
5  }
6  else {
7    Bucket[old_val] = accu;
8    accu = Bucket[val]+1;
9  }
10 old_val =val;
```

Listing 1. An efficient histogram calculation with II of 1.

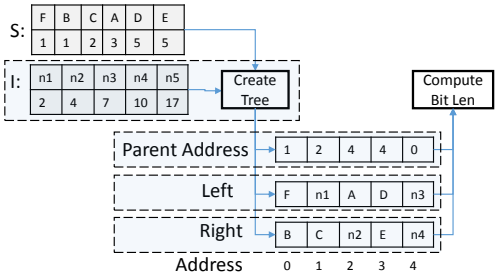

Fig. 4. The architecture for efficient Huffman tree creation. This architecture creates a Huffman tree in one pass by avoiding resorting of the elements.

B. Huffman Tree Creation

In order to create the Huffman tree, the basic algorithm creates a sub tree of the two elements with minimum frequency and adds intermediate node whose frequency is the sum of $f1$ and $f2$ to the list in sorted order ($f1$ and $f2$ are frequencies of those selected elements). This requires re-sorting the list each time when we add a new element to the list. At each step we remove the two elements with minimum frequencies and insert a new node with the aggregated frequency of those selected nodes. This means that the generated nodes are produced in non-decreasing sequence order. Thus, instead of adding the intermediate node to the sorted list, we used another BRAM to store the intermediate nodes in a FIFO.

With this modification, we eliminate the process of re-sorting. The architecture is shown in Figure 4. The S queue stores the input symbol/frequency list. The I Queue stores the intermediate nodes. The size of S is n, and size of I is $n-1$. *Create Tree* stores tree information on *Left*, *Right*, and *Parent Address* BRAMs. The changed algorithm works as follows. Initially, the algorithm selects the two minimum elements from the S queue in a similar manner to basic Huffman encoding. Then the algorithm adds the intermediate node $n1$ to the I queue. It selects a new element $e1$ from S queue. If the frequency of $e1$ is smaller than frequency of $n1$, we make $e1$ the left child. Otherwise, we make $n1$ the right child. If the I queue is empty (after selecting the left child), we select another element $e2$ from S and make it the right child. Then we add their frequency values, make a new intermediate node $n2$, and add it to I. This process continues until there is no element left in the S queue. If there are elements in the I queue, we create sub trees by making the first element the left child and second element the right child.

This eliminates need for resorting by using additional BRAM. While we are constructing the Huffman tree with this method, we store the tree information on three different BRAMs. The *Left* and *Right* BRAMs store the left and right children of each sub tree. The first left/right child is stored on address 0 of Left/Right. The *Parent Address* BRAM has the same address as its children but stores an address of the parent of that location. It points to the parent address of its children.

C. Parallel Bit Lengths Calculation

After storing tree information on *Left*, *Right*, and *Parent Address* BRAMs, calculating the bit length for each code is straightforward. The *Compute Bit Len* function starts from address 0 of *Left* and *Right* BRAMs and tracks their parents location from the *Parent Address* BRAM. For example, B and F have the same parent since they are both in address 0 in respective BRAMs. The address of the parent of B and F is located at 1 which is stored in the *Parent Address* BRAM at address 0. From address 1, we can locate the grandparent of F and B at address 2. From address 2, we can locate next ancestor of B and F at address 4. When we check address 4 and we find out it is zero, that means we have reached the root node. Therefore, bit length of F and B is equal to 4.

The data structures (*Left*, *Right*, and *Parent Address*) allow efficient and parallel bit length calculation. In these data structures, the symbols are stored from left to right, and we can track any symbol's parents to root starting from that symbol's position. In our design, we exploit this property and initiated parallel processes working from different symbols (Huffman

tree leaf nodes) towards the root node to calculate bit lengths of symbols in parallel. Figure 5 shows an example where two processes are working to calculate the bit lengths in parallel. Each process operates on data in its region, e.g., *Process 2* only needs data for symbol D and E symbols.

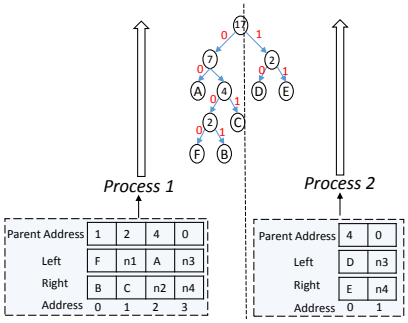

Fig. 5. A Parallel Bit Lengths Calculation. Each process has its own set of data which allows for fully parallel bit length calculation.

D. Canonization

In *Canonize*, the algorithm has to create two lists; one sorted by symbols and another sorted by frequencies. In our design, we changed the *Canonize* module by eliminating the list which is sorted by frequencies. The second list is needed by *CreateCodeword* to track the number of codewords with the same length. We can track the number of symbols having the same code length by using 27 counters since any codeword need at most 27 bits. This optimization efficiently reduces the running time of the *Canonize* by half. Therefore, output of the *Canonize* is only one list in our design. However, it slightly increases the running time of *CreateCodeword*.

E. Codeword Creation

In addition to that, using dependency pragma, we instruct HLS to ignore RAW dependency. In the *Create Codeword* module, the algorithm does the bit reverse of each code length. Code lengths can be up to 27 bits. The software bit reverse does not synthesize to efficient hardware. Therefore, we coded an efficient bit reverse in HLS that results in logic as good as a custom bit reverse using the coding technique given in [18]. Listing 2 shows an example bit reverse for 27 bit number. Bit lengths can be up to 27-bits requiring to write twenty seven of these functions in our design. Since these functions are synthesized efficiently in HLS, we inline them in our design which increases performance with a slight increase of area.

```
1  #pragma pipeline
2  for(int k = 0; k < 27; k++) {
3    j = (j << 1) | ((i >> k) & 0x1);
4  }
```

Listing 2. Efficient bit reverse for 27-bit number

Restructured Design: This design includes manual restructured and the optimized designs for *Radix Sort*, *Create Tree*, *Compute Bit Length*, *Canonize*, and *Create Codeword* modules, which were described earlier in this section.

Latency Optimized Design: On top of the *Restructured* design, we pipeline computations in individual modules using the *pipeline pragma* in the high-level synthesis tool. The *pipeline pragma* pipelines the computations in a region exploiting fine grained parallelism. Once restructured, rest of the computations in individual modules of the CHE are sequential. e.g., computations in iterations execute dependent *read, compute, write* operations in each iteration of loop. This allows pipelining of only these primitive operations in each iteration of the loop. This is done by pipelining the most inner loop of each module.

Throughput Optimized Design: This design further optimizes the *Latency Optimized* design to achieve coarse grained parallelism. The goal of this design is to improve throughput by exploiting coarse grained parallelism among the tasks (through task level pipelining). We achieve task level pipeline in the high-level synthesis by using a *dataflow* directive. However, the current *dataflow* directive only works if the input/output of functions are read/written by only one process. We solved this issue by duplicating the input/outputs which are read/written by more than one process. Listing 3 shows the pseudocode. For example, the output of *Sort* is read by two processes (*Create Tree* and *Canonoize*). Therefore, we duplicated the output of *Sort* into two different arrays (BRAMs: *SF_SORT1, SF_SORT2*) inside the *Sort* module. This is shown in Listing 3 Line 9. For simplicity, we omitted BRAM duplication parts for the rest of the code in Listing 3. This incurs additional logic to duplicate this data as shown in Listing 4. This has some adverse effect on the final latency, but it improves the overall throughput.

```
1  CanonicalHuffman(SF[SIZE], Code[SIZE]){
2  #pragma dataflow
3  SF_TEMP1[SIZE];
4
5  SF_SORT1[SIZE];
6  SF_SORT2[SIZE];
7
8  Filter(SF, SF_TEMP1);
9  Sort(SF_TEMP1, SF_SORT1, SF_SORT2);
10 CreateTree(SF_SORT1, PA, L, R);
11
12 //Separate data in PA,L, R
13 //into PA1, L1, R1, PA2, L2, R2
14
15 //Parallel bit lenght calculation
16 ComputeBitLen1(PA1, L1, R1, Bitlen1);
17 ComputeBitLen1(PA2, L2, R2, Bitlen1);
18
19 //Merge BitLen1 and BitLen2 to BitLenFinal
20
21 TruncateTree(BitlenFinal, Bitlen3, Bitlen4);
22 Canonize(Bitlen4, SF_SORT2, CodeTemp);
23 CreateCodeword(Bitlen4, CodeTemp, Code);
24 }
```

Listing 3. Pseudocode for the throughput optimized design.

```
1  Sort(SF_TEMP1, SF_SORT1, SF_SORT2){
2  //Sort logic
3  SF_SORTED = ... ;
4  //Additional logic
5  for(int i=0;i<n;i++){
6    SF_SORT1[i] = SF_SORTED[i];
7    SF_SORT2[i] = SF_SORTED[i];
8  }}
```

Listing 4. Additional logic to support task level parallelism.

V. Software Implementations

The initial *Naive* software implementation is a functionally correct design which is not optimized for efficient memory management. In *Baseline* design, we used efficient memory management through optimized data structures to optimize the naive implementation. An example optimization is using pointers efficiently instead of arrays whenever possible. The code has the same functionality as the code in HLS. In addition to the memory optimization using pointers, we did following optimizations.

1) Software Optimization (SO): We do the same optimization for the *Canonize* and *Create Codeword* functions as we did in HLS implementation. This cuts the running time of *Canonize* by almost half.

2) Compiler Settings (CS): We do compiler optimizations on top of software optimization using -O3 compiler flag. These compiler optimization levels do common compiler optimizations such as common sub expression elimination and constant propagation. On top of that, we did manual loop vectorization and loop unrolling whenever it gives better performance.

VI. Experimental Results

In this section, we present the performance, area, power and energy consumption results of our canonical Huffman encoding implementations for Xilinx Zynq FPGA, ARM Cortex-A9, and Intel Core i7 processors. All systems are tested with 256 and 704 symbol frequencies (as dictated by the of former LZ77 stage) as well as 536 as a median value in order to show scalability trends. The latency is reported in microseconds μs and throughput is reported in number of canonical Huffman encodings performed per second. In this work, Vivado HLS 2013.4 is used for the hardware implementations. The final optimized designs are implemented on a Xilinx Zynq FPGA (xc7z020clg484-1) and the functionality is tested using real-world inputs.

A. Software Implementations

ARM Cortex-A9: We used ARM Cortex-A9 667MHz dual-core processor with 32/32 KB Instruction/Data Cache and 512 Kbyte L2 cache. In order to get highest performance, we run our design on the processor without using an operating system. The latency results are calculated using the number of clock cycles times the clock period (1.49925 ns = 667 MHz). The number of clock cycles are collected using CP15 performance monitoring unit registers of ARM processor.

Figure 6 presents performance of software implementations running on the ARM processor. The initial *Naive* design is implemented without any optimizations. The *Baseline* design is optimized on top of the *Naive* design using efficient data structures and memory management on the ARM processor. On top of *Baseline* design then we apply Software Optimization (SO) and Compiler Setting (CS) in that order. In *SO* design, the run time of the *Canonize* module is reduced by 2.2X (114 us to 53 us). This added little overhead on the *CreateCodeword* module by increasing its run time from 75 us to 103 us. Overall, the running time of *SO* design is decreased from 622 us to 589 us. In the final design (*SO+CS*), the -O3 optimization decreases the running time of all modules, resulting in total running time of the design being decreased from 589 us to 220 us.

Intel Core i7: We also implemented the same software optimizations on a multi-core CPU implementation - an Intel Core i7-3820 CPU running at 3.6 GHz. Figure 7 presents performance of the various software implementations (*Naive*, *Baseline*, *SO* and *CS*). Due to fast running time of the algorithm on a powerful processor, the software optimization (SO) has very little impact on the final running time giving only 2 us of saving of total running time. The final optimized with *SO+CS* has the fastest running time.

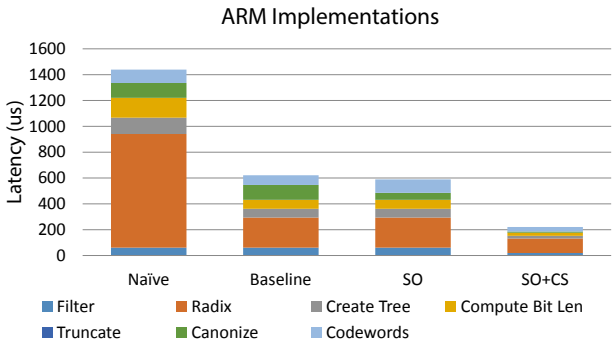

Fig. 6. The latency results of the various software implementations running on an ARM Cortex-A9 processor.

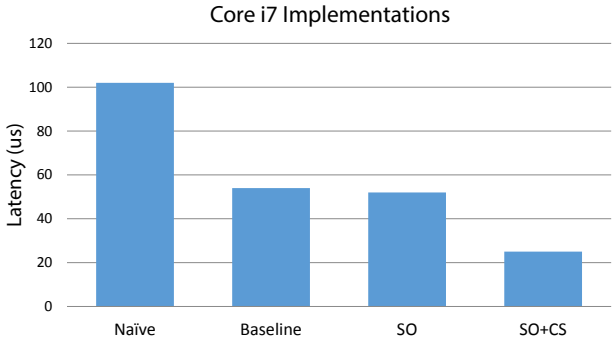

Fig. 7. The latency results of the various software implementations running on an Intel Core i7 processor.

B. Hardware Implementations

We synthesized the various hardware implementation described in Section IV (Baseline, Restructured, Latency Optimized, and Throughput Optimized). Table I gives the area utilizations and performance results for these different designs when using 704 input size (536 non zero elements), and Figure 8 shows their latency and throughput results.

TABLE I. Hardware area and performance results.

	Area		Performance		
	Slices	BRAM	Clock Cycles	II	Frequency
Baseline	2067	15	130977	130978	45
Restructured	761	14	90336	90337	173
Latency Optimized	1159	14	33769	33770	133
Throughput Optimized	1836	62	41321	3186	170

The number of slices in the *Baseline* design is reduced in *Restructured* design due to writing better HLS friendly code. (e.g., optimized bit reverse module). The *Latency Optimized*

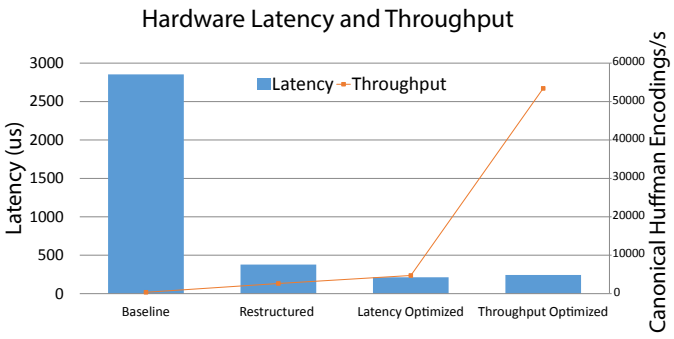

Fig. 8. The latency and throughput of hardware implementations.

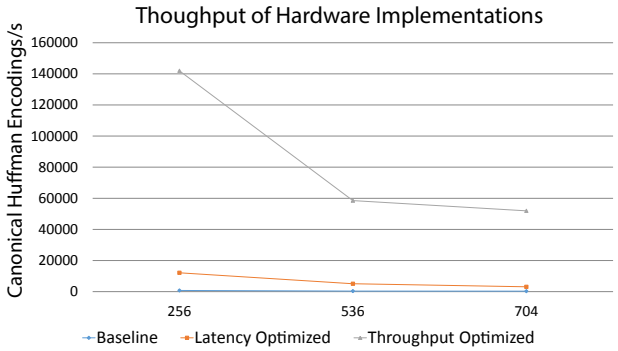

Fig. 9. The throughput of hardware implementation for different input sizes.

design has higher slices and higher performance due to fine grained parallelism. The *Throughput Optimized* designs use more BRAMs due to duplication of the input/output data for the purpose of overcoming limitations of *dataflow directive* in the HLS tool (each module can only have one read and write operation per port). The *Throughput Optimized* also has higher throughput due to the additional resources required to take advantage of both coarse grained and fine grained parallelism.

The latency $(clock\_cycles \times clock\_period)$ measures the time to perform one canonical Huffman encoding operation. The throughput $(Initiation\_Interval(II) \times clock\_period)$ is the number of canonical Huffman encoding operations per second. In the case of pipelining, the latency and throughput operations may not be equivalent. The latency reduces from *Baseline* design to *Latency Optimized* due to restructuring and pipelining. The latency of *Throughput Optimized* design increases from 212 μs to 242 μs (largely due to 33769 and 41321 clock cycles, respectively, though the clock period is also larger).

However, this *Throughput Optimized* design has better throughput than previous designs. The *Throughput Optimized* accepts new data every 3186 clock cycles while *Latency Optimized* design accepts new data in every 33770 clock cycles. Therefore, *Throughput Optimized* has higher throughput than the *Latency Optimized* design. Figure 9 shows the throughput for three designs for non-zero input sizes 256, 536 and 704. The y-axis shows the throughput (the number of canonical Huffman encodings per second) and x-axis shows input sizes.

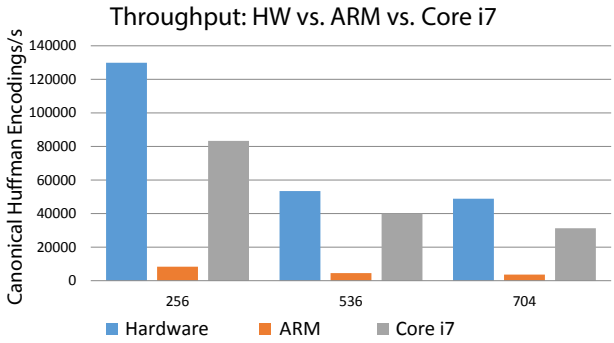

Fig. 10. Throughput for different input sizes: HW vs. ARM vs. Core i7

C. Hardware and Software Comparison

In this section, we discuss efficiency of our various hardware and software implementations in terms of four aspects: performance, power/energy consumption, productivity, and the benefits and drawbacks of designing using HLS tools as compared to directly coding RTL.

Performance: Figure 10 compares the throughputs of the best Xilinx Zynq FPGA design, ARM Cortex-A9 design, and Intel Core i7 design for three different input sizes. Overall, the hardware implementation has 13-16X better throughput than the ARM design due to exploiting task level parallelism. The hardware design achieves 1.5-1.8X speed-up over highly optimized design on a Intel Core i7 processor.

Power and Energy Consumption: The primary focus of this paper is to understand the energy efficiency of canonical Huffman encoding on different hardware and software platforms. We measured the ARM and FPGA portions of Zynq device power consumptions using power domains VCCPINT and VCCINT as described [19] in real time. The power consumption of ARM design is measured by running the design as a standalone application (i.e., without an operating system). The ARM implementation consumes around 344 mWatt in real time. The power consumption of FPGA part is measured in two ways; the first measurement is obtained by using Xpower tool which estimates around 380 mWatts. Then we calculated the power consumption of our design from real time *voltage* and *current* by running our design on a Zynq chip. Our design consumes maximum 170 mWatts among 250 runs. The Intel Core i7 power when running the canonical Huffman encoding is measured using Intel Power Gadget tool-kit in a real time[20]. The Core i7 consumes between 27-30 Watts when running the software.

Figure 11 (170 mWatt is used as FPGA power) shows the energy consumption of the different platforms. This is measured in the number of canonical Huffman encodings performed per Watt for the FPGA, ARM, and Core i7 designs. The FPGA is the most energy efficient. For three different sizes, hardware implementation has around 230X more encodings per Watt than the Core i7, and the ARM implementation has around 9X more CHEs than Core i7 design.

Productivity: We spent about one month designing the algorithm on an FPGA using the Xilinx Vivado HLS tool. Approximately 25% of the time was spent on learning the algorithm and initial planning which also includes writing a C

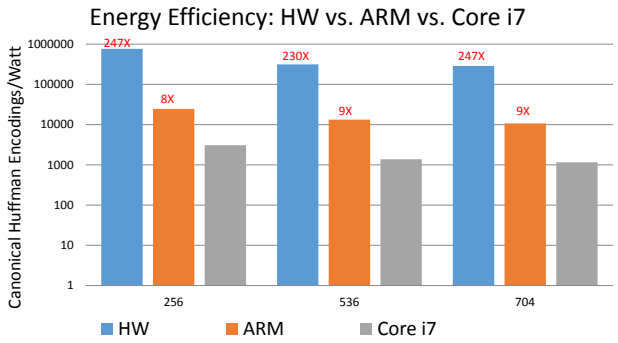

Fig. 11. Power efficiency for various input sizes: HW vs. ARM vs. Core i7

code for Core i7 and ARM processors. The rest of the time (approximately 75% of the total time) was spent on designing optimized FPGA solution using HLS. This includes time spent to create the Baseline, Restructured Latency Optimized, and Throughput Optimized implementations. Our final hardware design is parametrizable for different inputs which allows easy design space exploration in a short time while achieving higher throughput with significantly less energy consumption than the Intel and ARM processors.

HLS vs RTL: An obvious next step would be to implement the best hardware implementation using RTL to obtain even better results. This would likely increase the clock frequency, and provide better throughput and energy consumption. However, this is an expensive proposition in terms of designer time. For example, the tree creation/bit length calculation modules require careful coding if one decides to design them in RTL. HLS provides good results because limited parallelism in individual modules can be easily exploited by pipelining most inner loops in C, and that would be the primarily optimization target that one would exploit when writing RTL. Certainly, HLS provides a huge benefit in terms of design space exploration, and we advise first using the HLS tools to find a suitable micro architecture, and then develop the RTL from that micro architecture in order to optimize it further.

VII. Conclusion

One may assume this application is not suitable for hardware implementation because the algorithm is complex and sequential. On the contrary, the hardware implementation produces a superior result both in terms of throughput and energy efficiency. To demonstrate these, we developed a number of performance and low power design and implementation of canonical Huffman encoding in both hardware and software running on the Virtex Zynq FPGA, ARM Cortex-A9, and Intel Core i7 platforms. We demonstrated several optimization techniques for complex and inherently sequential applications (hard to code by hand due to complexity and sequentiality) such as Huffman tree creation can be easily done in HLS in a short amount of time with high performance and low power. Our final design has around 13-16X times higher throughput than highly optimized design on ARM and it is more energy efficient than design on high end multi-core CPU. We designed the systems on an FPGA and verified the functionality on a Zynq device. As future work, we plan to extend current work for different domains of applications such as JPEG encoding. We also plan to study more inherently sequential application designs using high-level synthesize tools.

VIII. Acknowledgment

The authors would like to thank Scott Hauck for discussion of the design and his valuable feedback for this work.

References

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

[2] B. P. Flannery, W. H. Press, S. A. Teukolsky, and W. Vetterling, "Numerical recipes in c," *Press Syndicate of the University of Cambridge, New York*, 1992.

[3] L. P. Deutsch, "Deflate compressed data format specification version 1.3," 1996.

[4] W. B. Pennebaker, *JPEG: Still image data compression standard*. Springer, 1992.

[5] B. Sherigar and K. RamanujanValmiki, "Huffman decoder used for decoding both advanced audio coding (aac) and mp3 audio," Jun. 29 2004, uS Patent App. 10/880,695.

[6] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Communications of the ACM*, vol. 30, no. 6, pp. 520–540, 1987.

[7] G. G. Langdon Jr *et al.*, "Arithmetic coding encoder and decoder system," Feb. 27 1990, uS Patent 4,905,297.

[8] J. Mars and L. Tang, "Whare-map: heterogeneity in homogeneous warehouse-scale computers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ACM, 2013, pp. 619–630.

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

[10] Z. Aspar, Z. Mohd Yusof, and I. Suleiman, "Parallel huffman decoder with an optimized look up table option on fpga," in *TENCON 2000. Proceedings*, vol. 1. IEEE, 2000, pp. 73–76.

[11] S. T. Klein and Y. Wiseman, "Parallel huffman decoding with applications to jpeg files," *The Computer Journal*, vol. 46, no. 5, pp. 487–497, 2003.

[12] Y.-S. Lee, T.-S. Perng, Hsu *et al.*, "A memory-based architecture for very-high-throughput variable length codec design," in *Circuits and Systems*, vol. 3. IEEE, 1997, pp. 2096–2099.

[13] A. Mukherjee *et al.*, "Marvle: A vlsi chip for data compression using tree-based codes," *Very Large Scale Integration Systems, IEEE Transactions on*, vol. 1, no. 2, pp. 203–214, 1993.

[14] H. Park and V. K. Prasanna, "Area efficient vlsi architectures for huffman coding," *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, vol. 40, no. 9, pp. 568–575, 1993.

[15] S. Rigler, W. Bishop, and A. Kennings, "Fpga-based lossless data compression using huffman and lz77 algorithms," in *CCECE*. IEEE, 2007, pp. 1235–1238.

[16] S.-H. Sun and S.-J. Lee, "A jpeg chip for image compression and decompression," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 35, no. 1, pp. 43–60, 2003.

[17] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *Information Theory, IEEE Transactions on*, vol. 24, no. 5, pp. 530–536, 1978.

[18] J. Zhang *et al.*, "Bit-level optimization for high-level synthesis and fpga-based acceleration," in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2010, pp. 59–68.

[19] "Zynq-7000 ap soc measuring zc702 power using standalone application tech tip." [Online]. Available: http://www.wiki.xilinx.com/

[20] "Intel power gadget." [Online]. Available: http://software.intel.com/en-us/articles/intel-power-gadget-20