Massively Parallel Huffman Decoding on GPUs

André Weißenberger
Johann Wolfgang Goethe University
60323 Frankfurt am Main, Germany
andre.weissenberger@stud.uni-frankfurt.de

Bertil Schmidt
Institute of Computer Science
Johannes Gutenberg University
55128 Mainz, Germany
bertil.schmidt@uni-mainz.de

ABSTRACT

Data compression is a fundamental building block in a wide range of applications. Besides its intended purpose to save valuable storage on hard disks, compression can be utilized to increase the effective bandwidth to attached storage as realized by state-of-the-art file systems. In the foreseeing future, on-the-fly compression and decompression will gain utmost importance for the processing of data-intensive applications such as streamed Deep Learning tasks or Next Generation Sequencing pipelines, which establishes the need for fast parallel implementations. Huffman coding is an integral part of a number of compression methods. However, efficient parallel implementation of Huffman decompression is difficult due to inherent data dependencies (i.e. the location of a decoded symbol depends on its predecessors). In this paper, we present the first massively parallel decoder implementation that is compatible with Huffman's original method by taking advantage of the self-synchronization property of Huffman codes. Our performance evaluation on three different CUDA-enabled GPUs (TITAN V, TITAN XP, GTX 1080) demonstrates speedups of over one order-of-magnitude compared to the state-of-art CPU-based Zstandard Huffman decoder. Our implementation is available at https://github.com/weissenberger/gpuhd.

CCS CONCEPTS

• **Information systems** → **Data compression**; • **Computing methodologies** → **Massively parallel algorithms**; • **Mathematics of computing** → *Coding theory*;

KEYWORDS

Data compression, GPUs, CUDA, Huffman Decoding

ACM Reference Format:
André Weißenberger and Bertil Schmidt. 2018. Massively Parallel Huffman Decoding on GPUs. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3225058.3225076

1 INTRODUCTION

Since its introduction in 1952, Huffman coding [17] has been adopted as an entropy coding stage for a wide variety of compression schemes, including different standards for multimedia compression, like JPEG and MP3, as well as for general purpose compression applications, like DEFLATE [7] (ZIP) or gzip [8]. At this point, alternative coding methods such as arithmetic coding [22] or Asymmetric Numeral Systems (ANS) [9] exist and also provide superior compression results compared to Huffman coding. Nevertheless, Huffman's method remains popular as an integral part of recent compression software, such as Brotli [1] and Zstandard [3], due to its favorable cost to performance ratio.

In recent years, data compression gained significant importance in the field of high performance computing. Efficient storage solutions are needed by file systems supporting compression features, hence increasing effective memory bandwidth. Furthermore, data compression techniques hold the potential of accelerating network communication in Big Data and HPC applications. Hadoop [10], and similar processing frameworks allow for the use of different algorithms to compress and decompress the data exchanged throughout the network, leading to significantly lower communication overhead. With modern HPC applications relying on multi-core and GPU architectures, it seems natural to parallelize data compression and decompression as well. Specifications and implementations of multi-threaded compression formats already exist. Recently, Zstandard has been extended to be capable of compressing and decompressing archives utilizing multiple CPU cores.

Regarding GPUs, potential performance improvements are considerable: applications relying on graphics processors and similar accelerators are often bottlenecked by network- and PCIe bandwidth. With sufficiently fast compression software for GPUs, data transfer costs could be reduced by compressing and decompressing data directly in Video RAM.

Huffman coding is frequently used for entropy coding. Thus, the question arises whether the Huffman encoding and decoding procedures can be parallelized and implemented on GPU architectures. In this paper, we will put our emphasis on parallel decoding. Improvements of decompression speeds are particularly important in application scenarios where data is compressed only once but decompressed frequently. However, parallelization of Huffman decoding is made challenging by the fact the location of a decoded symbol depends on the locations of all predecessors. As a consequence, existing GPU implementations have altered Huffman's original method by splitting data into independent chunks which can be compressed and decompressed independently. Even though this approach enables parallelism, it is not compatible with the mentioned file formats and also reduces compression efficiency.

In this paper, we present the first GPU-based decoder that is compatible with Huffman's original method. The solution is based on a special property of Huffman codes which can be exploited to enable parallel decoding: the self-synchronizing property. Performance evaluation shows that our implementation is able to achieve speedups of over one order-of-magnitude compared to the state-of-art CPU-based Zstandard Huffman decoder implementation on three different GPUs (TITAN V, TITAN XP, GTX 1080).

The rest of this paper is organized as follows. Section 2 provides necessary background information on Huffman coding and reviews related work. The self-synchronizing property of Huffman codes is discussed in Section 3. Section 4 presents our GPU-based decoder and focuses on practical implementation of the algorithm using CUDA. Section 5 presents experimental results. Section 6 concludes.

2 BACKGROUND

2.1 Huffman Coding

Let Σ be a source, sequentially emitting symbols from a finite alphabet $\mathcal{A} = \{a_0, \ldots, a_{n-1}\}$ of cardinality n. Each symbol is emitted with a certain probability $P(a_i)$. Thus, a probability vector can be defined for Σ, representing all of the source probabilities: $\vec{p}(\Sigma) := (P(a_0), \ldots, P(a_{n-1}))^T \in [0, 1]^n$.

In practice, a file saved on a hard drive is a common example for such a source. $\vec{p}(\Sigma)$ can be determined by counting the occurrences of each entry in the file.

In the following, the source is assumed to be memoryless, meaning that the probability of a certain symbol being emitted by Σ is conditionally independent on previously emitted symbols.

A finite length string of concatenated symbols $s_0, \ldots, s_{m-1}$ emitted by Σ is referred to as a message. Huffman coding works by replacing each symbol s_i with an appropriate binary codeword c_i, leading to a compressed message $c_0, \ldots, c_{m-1}$. A Huffman encoder assigns shorter codewords to symbols that are more common in the source, and longer codewords to symbols that are less common. This results in a binary representation which generally consumes fewer bits than the original message.

The codewords can be extracted from a binary tree constructed using the source probabilities from $\vec{p}(\Sigma)$. For example, consider a source Σ_1 with source alphabet $\mathcal{A}_1 := \{A, B, C, D, E\}$ and a corresponding probability vector

$$\vec{p}(\Sigma_1) := \frac{1}{8}(3, 1, 1, 1, 2)^T .$$

The process of constructing the binary tree starts by creating a leaf node L_i for each symbol of the alphabet. Leaf nodes, in the following referred to by the symbol they represent, shall feature two attributes: $L_i.s$, which contains the symbol and $L_i.p$, which contains the corresponding probability. The set of all leaf nodes is referred to by L. Internal nodes I_i do not contain any symbol, but also have a probability $I_i.p$ as well as a left and right child: $I_i.l$ and $I_i.r$.

For the next step, two nodes have to be selected. It is crucial to always select the nodes with the lowest probability out of all nodes available at a given point of time. This can be simplified by keeping the nodes in a min-priority queue Q, with the least probable nodes having the highest priority. After two nodes have been popped out from Q, a new internal node is created to become their parent node. The new parent is assigned the sum of the probabilities of its children and is added to Q afterwards. In the example, $B.p = C.p = 1/8 \leqslant X.p$ for all $X \in L$, so B and C are merged to become the children of a new internal node I_0, with $I_0.p = B.p + C.p = 2/8$. After the new node has been added to the queue, this step is repeated until only one node is left. This node is the root of the Huffman tree. Algorithm 1 represents a formal description of the process.

Algorithm 1: Constructing a Huffman tree

```
input  : A source alphabet $\mathcal{A}$ of size $n$ and a corresponding
         probability vector $\vec{p}$
output : An optimal Huffman tree for the source
1  $Q \leftarrow$ empty min-priority queue;
2  for $i \leftarrow 0$ to $n-1$ do
3      create node x; x.$s \leftarrow A_i$; x.$p \leftarrow \vec{p}_i$;
4      $Q$.insert_with_priority(x, $\vec{p}_i$);
5  end
6  while number of elements in $Q > 1$ do
7      a $\leftarrow Q$.get_min(); b $\leftarrow Q$.get_min();
8      create node I;
9      I.$l \leftarrow$ a; I.$r \leftarrow$ b;
10     I.$p \leftarrow$ a.$p$ + b.$p$;
11     $Q$.insert_with_priority(I, I.$p$);
12 end
13 return $Q$.get_min() ;    // root of the finished tree
```

Table 1: A Huffman table retrieved from the tree in Figure 1.

Symbol	Code	Frequency
A	00	3/8
B	10	1/8
E	11	2/8
C	010	1/8
D	011	1/8

Proceeding with the example, B and E are merged into I_1 with a probability of $3/8$, then I_1 is merged with A to I_2 with a probability of $5/8$, so only I_1 and I_2 remain in the queue. Finally, I_1 and I_2 are merged into the root node I_3, which has probability 1. The final tree is shown in Figure 1. All of the left edges in the tree are labeled with 0, all of the right edges are labeled with 1. The codewords can now be retrieved by concatenating the labels on the path from the root to each symbol, e.g. the codeword for symbol C is 010 in this example. Due to the way the tree is constructed, it is guaranteed that a codeword never can be a prefix of any other codeword. This property is referred to as the *prefix property* [1] of Huffman codes. Furthermore, Huffman codes are *complete*, meaning that all nodes in the Huffman tree either are leaves or have exactly two children. Thus, Huffman codes are an example of complete prefix codes.

Codewords are often stored in a Huffman table, together with their respective symbols. Table 1 shows the Huffman table for our example.

[1] It is beyond question that *prefix-free* would be a more suitable description, in literature, however, *prefix property* is the predominant term.

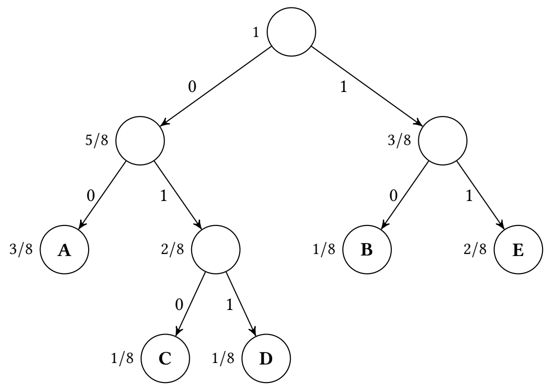

Figure 1: A simple Huffman tree, each node is shown with its respective probability at the left.

Now it is possible to encode messages from the source Σ_1. For example, the message **ABAEECDA** is encoded as 001000111101001100, just by replacing the symbols with their respective codewords. Because Huffman codes are prefix free, decoding is especially simple as a decoder can read the next symbol without knowing the length of the codeword. By comparing the first bits of the input to the entries in the Huffman table, the symbol can be identified and written to the output.

2.2 Related Work

In the past, different techniques dealing with GPU-based Huffman coding and decoding have been developed. Angulo et al. [2] implemented Huffman compression and decompression on GPUs for seismic data. In this approach, the encoding scheme is altered, such that chunk sizes are fixed at encoding time. The chunks are embedded into discrete packets, which are aligned to byte boundaries in the output stream. Although these modifications enable parallel decoding, the respective algorithms are obviously not suitable for decoding data generated with Huffman's original method, including the file formats mentioned in the introduction.

Recently, a lot of research regarding general GPU-based compression and decompression schemes has been accomplished. Sitaridi et al. [23] published a specification of a file format enabling GPU based compression and decompression, consisting of LZ77 compression and optional Huffman coding. In this format, the input is split into chunks, which are compressed and decompressed individually. Patel et al. [21] developed a similar block-based format, which, in addition to Huffman coding, relies on the Burrows-Wheeler and move-to-front transformation. However, practical decoding on the GPU proved to be slower than the corresponding CPU-based implementation for this approach.

Funasaka et al. [14, 15] presented a run-length- and dictionary based compression scheme for GPUs, however, their approach to entropy coding is only loosely related to Huffman's method.

CULZSS [20] is an implementation of the *Lempel-Ziv-Storer-Szymanski* (LZSS) algorithm for NVIDIA GPUs. Funasaka et al. also implemented the *Lempel-Ziv-Welch* (LZW) [13] algorithm using CUDA.

Klein and Wiseman [18] constructed a parallel algorithm for multi-core processors capable of decoding data that has been compressed using Huffman's original method. It relies on the so-called *self-synchronizing property*, which many Huffman codes possess.

Obviously, a parallel decoder exploiting the self-synchronizing property can only be useful if at least a major part of all codes encountered in practice possesses this property. In Section 3, we will further discuss this problem and conclude that almost all codes constructed by real-life encoders possess the self-synchronizing property.

3 SYNCHRONIZATION

Klein and Wiseman's decoder, as well as the GPU-based solution presented in this paper, exploit the *self-synchronizing property* of Huffman codes. A Huffman code which possesses the self-synchronizing property is called *self-synchronizing*. In the following, we will explain the effect of this property by giving an example. Afterwards, we will discuss under which conditions a Huffman code is self-synchronizing, and substantiate that codes that are not self-synchronizing are rarely encountered in practice.

Suppose that two decoders, A and B, are processing the message 001000111101001100 from Subsection 2.1. The process is illustrated in Table 2. The first row represents the indices of the respective bits, the second row contains the message itself. The last two rows contain the output of the decoders. Vertical bars are used to indicate codeword boundaries.

We assume that decoder A starts decoding the message from the beginning, i.e. at index 0. Decoder B shall start decoding at index 3, indicated by the arrow. As B starts processing the message at the suffix of another codeword, its decoding will initially be erroneous. B decodes the string *ADE*, while A correctly decodes *AEE*. However, bit 7 completes a codeword for both processes (indicated by the circle), so B decodes the next symbol, and hence the rest of the message correctly. Therefore, bit 7 is referred to as a *synchronization point*, as both decoders became synchronous.

After a synchronization point has been encountered, decoders will always generate the correct output. This property can be used to construct a multi-threaded decoder: When partitioning a message into multiple blocks, codewords are cut by block boundaries. Thus, the first symbols decoded in most of the blocks may be erroneous. However, threads can overflow to subsequent blocks and correct those errors until a synchronization point is encountered. The decoder can easily detect synchronization points by keeping track of codeword boundaries: when a codeword boundary detected by the current thread is aligned with a previously detected boundary, synchronization has been achieved.

There are two central questions regarding the capability of this approach: which sets of Huffman codes posses the self-synchronizing property, and what is the expected number of bits that have to be processed before a synchronization point is reached?

Klein and Wiseman [18] already presented a thorough analysis regarding the second question. They evaluated the expected number of bits E by

$$E := \frac{W}{P(S)},$$

Table 2: Illustration of the self-synchronizing property.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	0	**0**	**1**	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**0**	**1**	**0**	**0**	**1**	**1**	**0**	**0**
				↑					○									
Decoder A	A		B		A		E		E		C			D			A	
Decoder B				A		D			E		C			D			A	

where W is the *weighted average codeword length* and $P(S)$ represents the probability of the synchronization point being directly at the end of the codeword that was cut by the block boundary. In practical tests, this estimation proved to approximate the measured number of bits well. Synchronization was achieved after less than 73 bits on average for the nine tested plain text and binary files. Hence, as the second question has already been addressed, the remainder of this section will focus on the first question.

Assume an encoded message that is cut by a block boundary within a codeword, such that at least n bits remain after the cut. We shall consider the sequence of n bits following the cut point as a concatenation of two sequences, y and z. Let x denote the sequence preceding the cut point. z is referred to as a *synchronizing sequence* for x and y if the concatenations yz and xyz both are sequences of complete codewords. If this is the case, there is a synchronization point after the last bit of z. In the example of Table 2, x equals 001, y equals 00 and z equals 011.

Using this definition, Gilbert and Moore [16] differentiate prefix codes with a maximal codeword length of greater than 1 into three different categories. A given code is said to be

(1) *completely self-synchronizing*, if for each x and y there is a synchronizing sequence z.
(2) *never self-synchronizing*, if there is no z such that z is a synchronizing sequence for all x and y.
(3) *partially self-synchronizing*, if only for some x and y there is a synchronizing sequence z.

Furthermore, a sequence z is *universally synchronizing*, if *the same* sequence z is a synchronizing sequence for all x and y for a given code. It can be shown that a prefix code C is completely self-synchronizing, if, and only if, there exists a universal synchronizing sequence for C. In the example, the sequence 011 is universally synchronizing, which makes the Huffman code constructed in Subsection 2.1 a completely self-synchronizing code. Note that the root of the Huffman tree is reachable via the sequence 011 from any internal node.

A synchronizing sequence does not necessarily need to be a codeword as in the example, but its suffix has to. Thus, if a given code C is an *affix code*, meaning that no codeword in C is a proper suffix of any other codeword, C must be never self-synchronizing. It follows immediately that fixed-length codes are never self synchronizing. Given a fixed-length code $N := \{00, 01, 10, 11\}$ and an arbitrary message that was encoded using N, a decoder starting to process that message within a codeword will obviously never be able to regain synchronization. Nevertheless, messages encoded with fixed-length codes can easily be decompressed in parallel by simply choosing the block size as a multiple of the codeword length.

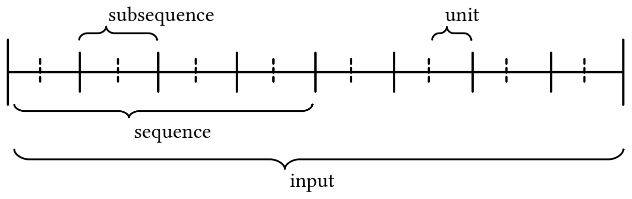

Figure 2: Exemplary input formatting for $N = 8$, $N_u = 2$, $B = 4$ and hence, $N_b = 2$.

Apart from the very common fixed-length codes, there also exist infinitely many variable-length codes that are never self synchronizing. However, the work of Gilbert and Moore revealed that those codes are nevertheless extremely rare, and that completely- or partially-self synchronizing codes occur much more commonly. It was stated further that it is very difficult to construct examples for never self-synchronizing codes without any deliberate intention. Klein and Wiseman [18] state that only for rare artificial distributions they were able to construct affix codes, which, in those cases, had to be carefully designed. Fraenkel and Klein [11] came to a similar conclusion in their work.

Another indicative result in regard to the self-synchronizing property is the following theorem, as defined and proven by Freiling et al. [12]: Let Q be the set of all complete prefix codes and let $\beta(n)$ be the number of codes in Q with n codewords. Let $\alpha(n)$ be the number of codes in Q with n codewords that posses a certain property. If

$$\lim_{n\to\infty} \frac{\alpha(n)}{\beta(n)} = 1,$$

we say that almost all codes in Q have that certain property.

The theorem states: *Almost all complete prefix codes have a self-synchronizing string.*

From those results, it can safely be concluded that the approach to massively parallel decoding presented in this paper can be applied effectively and reliably in practical applications.

4 MASSIVELY PARALLEL DECODING

4.1 Parallel Algorithm Design

The algorithm presented here divides the input data into N equally sized chunks, in the following referred to as subsequences. Each subsequence is further split into N_u smaller chunks, referred to as units. B adjacent subsequences are grouped into a sequence. If N is not a multiple of B, the last sequence will contain less than B subsequences. In consequence, the number of sequences N_b equals $\lceil N/B \rceil$. Input data is accessible at unit level. Figure 2 illustrates this division step for an example with eight subsequences.

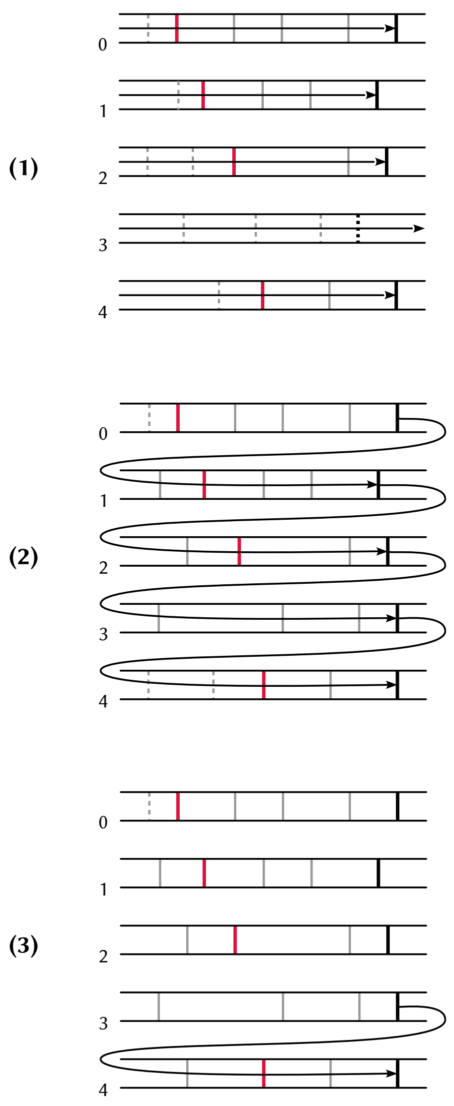

Figure 3: Illustration of Phase 1.

The decoder allocates simple arrays for input and output, as well as an auxiliary array of size N used to detect synchronization points, in the following referred to as `sync_info`. The elements of `sync_info` are structures, consisting of three integers: `(unit, bit, num_symbols)`. The element at index i in the `sync_info` array corresponds to the input subsequence of the same index. The `unit` and `bit` fields are used to save the presumed position of the last codeword's first bit in the respective subsequence. Analogously, the `num_symbols` field holds the presumed number of symbols in that subsequence.

A second auxiliary array is required to record in which sequences synchronization has been achieved. Corresponding to the number of sequences, its size equals N_b and the elements it contains are Boolean. In the following, this array will be referred to as `sequence_synced`. After all required data structures are set up, decoding consists of four phases:

Phase 1: Intra-sequence synchronization.
Phase 2: Inter-sequence synchronization.
Phase 3: Exclusive prefix sum over all `num_symbols` fields in the `sync_info` array to determine output array indices of decoded strings.
Phase 4: Write the output utilizing the information of the `sync_info` array.

Note that no output is written in phases 1 and 2, as their purpose is to locate the correct codeword boundaries inside each subsequence. In Phase 1, N threads are launched, each of which is assigned to decoding a different subsequence. Once a thread finishes decoding at the last complete codeword, it writes the position of the last codeword's first bit as well as the number of decoded symbols to the corresponding `sync_info` entry. In order to prevent data races, a thread has to wait at this point until all threads of the same thread block have finished decoding their assigned subsequence.

Algorithm 2: Phase 1, from a thread's perspective

```
input  : An array of subsequences SQ of size $N$
output : An array of triples sync_info of size $N$
current_subseq ← global thread id;
current_subseq_in_block ← local thread id;
last_codeword ← decode_subseq(SQ, sync_info,
    current_subseq, false);
++current_subseq;
++current_subseq_in_block;
__syncthreads();
sync_achieved ← false;
for $i$ ← 1 to $B$ do
    if not sync_achieved and current_subseq_in_block < $B$
    then
        last_codeword ← sync_info[current_subseq];
        current_codeword ← decode_subseq(SQ, sync_info,
            current_subseq, true);
        if last_codeword equals current_codeword then
            sync_achieved ← true;
        end
        else
            sync_info[current_subseq] ← current_codeword;
        end
    end
    ++current_subseq;
    ++current_subseq_in_block;
    __syncthreads();
end
```

Part (1) of Figure 3 shows an example of this situation: Five subsequences (horizontal bars) are being processed by five parallel threads (arrows) which pass several codeword boundaries (vertical bars). Threads 0, 1, 2, and 4 detect erroneous codewords (dashed) before reaching a synchronization point (red). In consequence, they decode the rest of the subsequence correctly, including the boundary of the last complete codeword (bold line). Subsequence 3 does not contain any synchronization point, hence all boundary positions detected by Thread 3 are incorrect, including the last one (bold and dashed).

Once threads have synchronized, they continue with overflowing to the next subsequence (illustrated in Part (2) of Figure 3) whereby decoding is picked up directly after the last complete codeword in the previous subsequence. Each thread then finishes decoding the current subsequence, possibly passing a synchronization point. In this case the position of the last codeword in the subsequence detected by the current thread will match the one detected by the previous thread. The current thread can validate that by consulting the `sync_info` array at the respective index. If the positions match, the thread disables itself. Otherwise, the thread waits for the other threads to finish decoding their subsequences and then overwrites the position data in the `sync_info` entry with its own values. The number of symbols detected is updated as well. Afterwards, all threads that are still active will overflow to the following subsequences as they have not passed a synchronization point yet. Thread 2 in Figure 3 corrects the position of the last codeword in Subsequence 3 but remains enabled as it was not possible to achieve synchronization. This procedure is repeated B times such that the

Algorithm 3: *decode_subseq*, decoding a single subsequence

```
input : An array of subsequences SQ of size $N$
        An array of triples sync_info of size $N$
        An index $i$
        A boolean value overflow
output: A triple of integers (unit, bit, num_symbols)
1  if overflow then
2      pos_last ← sync_info[$i-1$];
3      decode the subsequence at $i-1$, starting at pos_last;
4      overflow to the subsequence at $i$;
5  end
6  decode the subsequence at $i$;
7  unit ← index of the unit within the subsequence where the
     last bit of the last complete codeword was located;
8  bit ← index of the last bit;
9  num_symbols ← number of symbols found in the
     subsequence;
10 return (unit, bit, num_symbols)
```

first thread in a sequence would be able to reach the last subsequence. Thread 3 in Part (3) of Figure 3 is the only active thread after the first iteration, and detects synchronization in Subsequence 4, thereby completing Phase 1 for the example.

Disabled threads need to wait for other threads in every iteration, but will not perform any further tasks. This is necessary for the algorithm to comply with the limitations of GPU architectures. In addition, if any thread reaches the end of the last subsequence in a sequence, it will disable itself regardless of whether synchronization was achieved or not. Hence, threads never overflow to the next sequence in this first phase. Pseudocode for the first phase is shown in Algorithm 2 and for the function to decode a single subsequence in Algorithm 3.

Phase 2 serves the purpose of achieving synchronization between sequences. N_b threads are launched. Each thread is assigned to a sequence, beginning with the second one, and starts to decode the first subsequence by continuing at the last subsequence of the previous sequence. Afterwards, each thread overflows to the following subsequences, correcting any erroneous information in the `sync_info` array. If a thread detects a synchronization point in Sequence i, it sets the flag at the i-th index of the `sequence_synced` array and disables itself afterwards. Analogously to Phase 1, threads wait for all other threads from the same thread block to finish with their subsequences before they continue decoding. Again, this procedure repeats B times, so each thread can reach the last subsequence of the sequence it was assigned to. Phase 2 is repeated until all flags in `sequence_synced` are set. As soon as this is the case, synchronization has been achieved for all subsequences and hence, `sync_info` contains the correct codeword positions and symbol counts. Pseudocode for the second phase is shown in Algorithm 4.

Algorithm 4: Phase 2, from a thread's perspective

```
input : An array of subsequences SQ of size $N$
        An array of triples sync_info of size $N$
        An array of flags sequence_synced of size $N_b$
output: The modified sync_info array
        The modified sequence_synced array
1  current_subseq ← (global thread id) $\cdot B + B$;
2  current_block ← thread block id +1;
3  sync_achieved ← false;
4  for $i$ ← 0 to $B$ do
5      if not sync_achieved then
6          last_codeword ← sync_info[current_subseq];
7          current_codeword ← decode_subseq(SQ, sync_info,
             current_subseq, true);
8          if last_codeword equals current_codeword then
9              sync_achieved ← true;
10             sequence_synced[current_block] ← true;
11         end
12         else
13             sync_info[current_subseq] ← current_codeword;
14         end
15     end
16     ++current_subseq;
17     __syncthreads();
18 end
```

Phase 3 performs a parallel exclusive prefix summation on the `num_symbols` fields in the `sync_info` array. When finished, the value of `num_symbols` at index m equals

$$\sum_{i=1}^{m} \texttt{sync\_info}[i-1].\texttt{num\_symbols}$$

with `sync_info[0].num_symbols` set to 0 afterwards. Now, for each subsequence `sync_info` contains the index at which the decoded string has to be written to the output. As parallel scan algorithms have already been subject to extensive research, they are out-of-scope for this paper.

In Phase 4, N threads are launched. Like in Phase 1, Thread i is assigned to decode Subsequence i. Using the information from `sync_info`$[i-1]$ and `sync_info`$[i]$, Thread i is able to correctly decode the contents of that subsequence and write them to the correct index in the output array. Algorithm 5 lists the pseudocode for the complete decoding process from the perspective of the host system.

Algorithm 5: GPU-based Huffman decoding

```
input  : An array in of compressed data
output : An array out containing the decoded data
1  divide in into units and subsequences;
2  allocate all arrays necessary;
3  transfer in to the device;
4  launch phase 1;
5  while there are unset flags in sequence_synced do
6      launch phase 2;
7  end
8  launch phase 3;
9  launch phase 4;
10 transfer the output from the device to out;
```

4.2 CUDA Implementation

For implementation and testing, NVIDIA's CUDA SDK has been used. However, as the algorithm does not rely on any specific CUDA features, implementations for related parallel programming interfaces such as OpenCL are conceivable.

In addition, we have created an implementation for multicore processors using native C++-11 threads. Here, the input is split into N sequences for N threads. Each thread is assigned to a sequence for decoding, and overflows to the next sequence. In this case, subsequences only serve the purpose of recording codeword boundaries.

We compare both implementations to Zstandard's state-of-the-art Huffman decoder. By default, this decoder only supports input of a size up to 128 kilobytes, so it had to be modified in order to remove this restriction for the tests. Using special codeword tables, Zstandard's decoder is capable of decoding one, two or four symbols with a single access to the table. We used the second of those variants in our tests. This decoder has a look-ahead distance of a certain number of bits. If two complete codewords are contained within that certain number of bits, both of them will be decoded with a single table access. However, if the codewords are too long, they will not fit entirely inside the look-ahead distance and only the first symbol will be decoded. Our CPU- and GPU-based implementations decode only one symbol per access.

For all three implementations, decoding works by looking up the next n bits from the input in a hash table using direct addressing, i.e. n bits are used as the key. The length of the next codeword, k, is retrieved and k bits are removed from the input. The corresponding symbol is stored together with its length. This technique works for a maximum codeword length of up to approximately 13, before the table becomes too large to fit inside the L1-Cache of the respective processor. Thus, the maximum codeword length has to be limited at encoding time. In practice, however, the impact on compression ratio is marginal in most of the cases. In our tests, a maximum codeword length of 11 has been used throughout, which also is the default for Zstandard. For the CUDA-based implementation, the table is kept in texture memory to achieve maximum throughput at random access. Input and output, as well as the `sync_info` and `sequence_synced` arrays are kept in global memory. The units a subsequence consists of are implemented as 32 bit wide unsigned integers. For all tests, decoded symbols have the size of one byte.

Phases 1, 2, and 4 have been implemented as individual CUDA kernels. To calculate the parallel prefix sum in Phase 3, the `exclusive_scan` method from the Thrust library [4] has been used.

Once all necessary arrays are allocated in global memory, the input is being copied to the device. Next, the kernel for Phase 1 launches. Threads decode the individual subsequences and update the `sync_info` array. Necessary thread block synchronization is realized by a `__syncthreads()` command.

Phase 2 executes immediately after Phase 1 has been completed. The kernel is launched repeatedly, until all flags in `sequence_sync` are set. The barrier inside the kernel is again realized by a `__syncthreads()` command. The barrier between kernel calls is implemented on the host. After each call, `sequence_sync` is checked for unset flags. If there are any, the kernel is relaunched. Even though this procedure is executed sequentially on the host system, the associated overhead proved to be only marginal. With CUDA 9 and higher, it is possible to synchronize threads from different thread blocks using cooperative groups. This feature allows threads to overflow from one sequence to the next without the need for interaction with the host, rendering multiple kernel launches unnecessary. However, inter-block synchronization can be fast even without the use of cooperative groups.

After Phase 2 is complete, the parallel prefix scan over the `sync_info` array is performed. When finished, the kernel implementing Phase 4 is launched, and all subsequences are being decoded using the correct starting positions. The output is written to global memory.

For all phases, a thread block size of 128 proved to be optimal in terms of decoding speed on all tested devices. A subsequence size of 4 was found to be a good compromise between performance and memory consumption in most of the test cases.

Dramatic performance degradations were observed with increasing compression ratios. The reason for that is the increasing number of shorter codewords in the input, implying a higher number of write operations per thread, as subsequence size remains constant throughout decoding. The problem was solved by implementing a simple mechanism which determines a reasonable subsequence size a-priori.

As some adjacent threads in Algorithms 2 and 4 follow different execution paths inside the main loop, branch divergence may occur. However, restructuring the source code in order to keep more threads in a warp synchronized did not yield any significant performance improvement, as the kernel's main bottleneck is related to the device memory access latency.

5 EXPERIMENTAL RESULTS

5.1 Testing environment

We have used three different systems to evaluate the performance of our implementation:

A: Core i7-4790 CPU (4 cores in total operating at 3.60 GHz) running Ubuntu 16.04.4 LTS with Linux kernel *4.4.0-62*

B: Core i7-2700K CPU (4 cores in total operating at 3.50 GHz) with an attached Geforce TITAN XP GPU running Ubuntu 16.04.4 LTS with Linux kernel *4.13.0-37*.

C: Dual Xeon E5-2683v4 (32 cores operating at 2.10 GHz) with attached Geforce GTX 1080 GPU and TITAN V GPU running Ubuntu 16.04.4 LTS with Linux kernel *4.4.0-116*.

The CPU-based decoders (including Zstandard) are compiled with GCC, version 5.4.0, the GPU-based decoders are compiled with NVCC, v.*9.1.85* for the TITAN V and v.*8.0.61* for the remaining devices. System A is used to benchmark our CPU-based implementation, as well as Zstandard's decoder. Our reported GPU benchmarks do not include the cost of memory transfers between CPU and GPU; i.e. we place the compressed input and the decompressed output in GPU device memory and ignore PCIe transfers.

5.2 Test data

The following datasets are used for testing.

enwik9: [19] Excerpt of the English Wikipedia, consisting of *UTF8*-coded *XML* data (raw: 1 GB, compressed 0.65 GB, ratio: 1.52).

flan_1565: [5] Sparse matrix from the SuiteSparse Matrix Collection (raw: 1.6 GB, compressed: 0.78 GB, ratio: 2.0).

mozilla: [6] *tar*-compressed archive of Mozilla 1.0 (raw: 51.2 MB, compressed 40 MB, ratio: 1.28).

webster: [6] *1913 Webster Unabridged Dictionary* in HTML format (raw: 41.5 MB, compressed 26 MB, ratio: 1.6).

In addition to the real-world datasets, we have generated twenty different sets of random data, each of size 1 GB. For every generated file, the frequency of certain symbols was increased, leading to higher compression ratios and steeper Huffman trees. The ratio of the compressed data ranges from 1.2 to 7.6. Note that ratios greater than 4 are unrealistic for Huffman coding to achieve for real-world inputs.

5.3 Results

We have tested our CPU-based implementation on System A to evaluate multi-core performance at high clock speeds. The line graph in Figure 4 shows the throughput achieved for the `enwik9` and `flan_1565` datasets with respect to an increasing thread count up to 8, while threads 5 to 8 are hyper-threads. With 8 threads, a throughput of 716 MB/s is achieved for enwik9, and 649 MB/s for `flan_1565`. The results indicate that the decoder scales well over multiple cores and benefits from hyper-threading.

In Figure 5, the throughput of the CPU-based decoder (running 8 threads) is compared to the single- and double-symbol variants of the modified Zstandard Huffman decoder for the real-world datasets. In the bar chart, as well as in the remainder of this section, the single-symbol variant is referred to as `zstd_X1` and the double-symbol variant is referred to as `zstd_X2`. As `zstd_X1` is

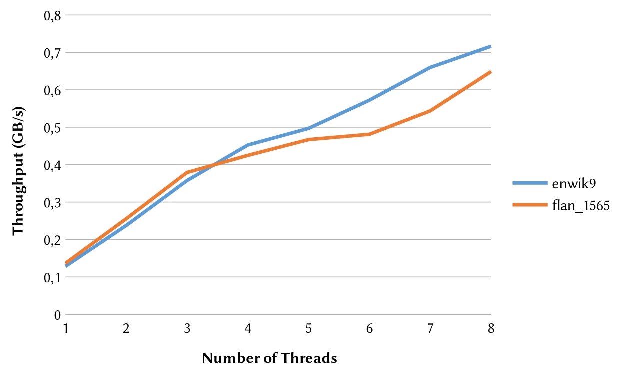

Figure 4: Throughput of the CPU based implementation for multiple threads

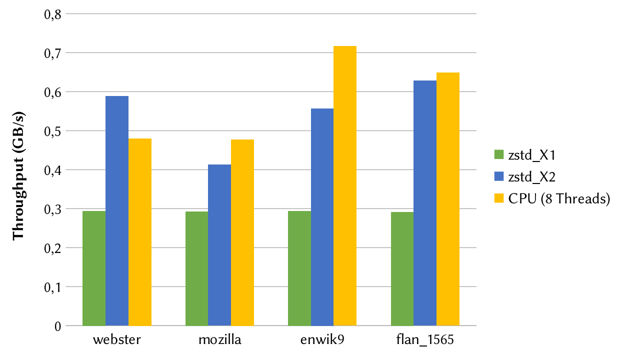

Figure 5: Throughput of the CPU based implementation (8 threads) compared to Zstandard's decoder

single-threaded and always decodes one symbol per table access, its throughput does not vary for different inputs. `zstd_X2` is also single-threaded, however, its performance increases for inputs with shorter codeword lengths, as it is the case with the `flan_1565` dataset. `zstd_X1` constantly delivers 294 MB/s, while `zstd_X2` delivers 412 MB/s for the `mozilla` dataset, 588 MB/s for the `webster` dataset, 716 MB/s for the `enwik9` dataset and 649 MB/s for the `flan_1565` dataset. For the `webster` dataset, `zstd_X2` outperforms our own decoder, and for `flan_1565`, the performance is comparable.

Figure 6 shows the speedup achieved by the GPU-based decoder relative to `zstd_X2` for the three tested GPUs. For all datasets, speedup greater than 10 can be achieved, with the TITAN V reaching values of up to 31.1 for the `webster` dataset and 27.9 for the `enwik9` dataset. Figure 7 shows the throughput achieved by the GPU-based decoder, again, relative to the values of `zstd_X2`. The TITAN V achieves a throughput of 18.3 GB/s for the `webster` dataset and 9.3 GB/s for the `mozilla` dataset. None of the GPUs achieve a throughput of less than 5 GB/s.

We have also measured speedup and throughput for the artificial datasets. Figure 8 shows the throughput achieved by the CPU-based implementation, utilizing 8 threads, in comparison to `zstd_X1`

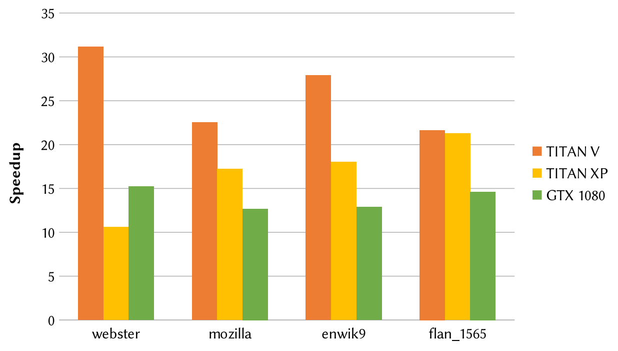

Figure 6: Speedup for real-life datasets against zstd_X2 on various GPUs.

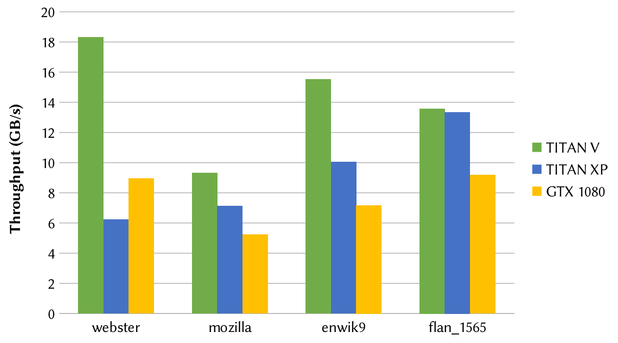

Figure 7: Throughput for real-life datasets on various GPUs.

and zstd_X2. Again, zstd_X1 delivers constant performance at 304.9 MB/s. zstd_X2 achieves 404 MB/s on the first dataset, which increases up to 669.8 MB/s at a compression ratio of 7.63. Our multi-threaded decoder delivers results from 775.3 MB/s on the first dataset, up to 980.8 MB/s on the last set. On average, zstd_X2 and our multi-threaded decoder achieve throughput rates of 646.23 MB/s and 867.1 MB/s respectively.

Figures 9 and 10 show the speedup and throughput of our GPU-based implementation relative to zstd_X2 for three different GPUs. At increasing compression ratio, the GTX 1080 on average achieves a speedup of 14.63 and a throughput of 9.34 GB/s. The TITAN XP achieves an average speedup of 21.32 and a throughput of 13.68 GB/s. The TITAN V achieves a speedup of 34.31 and a throughput of 21.89 GB/s on average. At a compression ratio of about 6, the performance of the TITAN V starts to diminish. For those compression ratios, the minimum subsequence size of 1 is chosen automatically, however, the subsequences are still to coarse to fully saturate the processing units of the TITAN V. This can be circumvented by choosing units of a size less than 32 bits.

6 CONCLUSION

In this paper, we have presented a fully parallel, GPU-based Huffman decoder. We have reasoned why almost all Huffman codes possess the self-synchronizing property, and presented a practical implementation using CUDA based on that property.

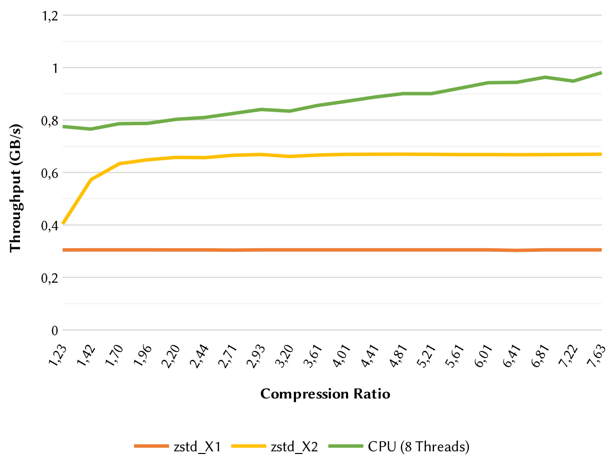

Figure 8: Throughput achieved by Zstandard and the multi-core variant.

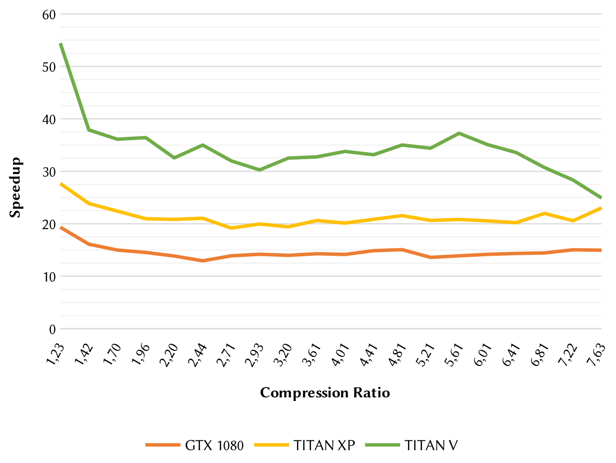

Figure 9: Speedup relative to Zstandard's Huffman decoder for various GPUs.

We have evaluated its performance using real-world and artificially generated datasets. On a NVIDIA TITAN V GPU, speedups of up to 30 over Zstandard's double-symbol decoder can be achieved.

Apart from being extremely fast, our decoder works with inputs encoded using Huffman's original method, and, is therefore suited to decode established formats like MP3 and JPEG files. Sodsong et al. [24] developed a GPU-based JPEG decoder. However, the codeword positions are sequentially detected on the host system. Employing our decoder, this step becomes unnecessary, making it possible to decode JPEG files using solely the GPU.

Our reported GPU benchmarks exclude the cost of transferring data between CPU and GPU using PCIe or NVLink technology. Transferring compressed data from host to device could be overlapped with kernel execution by a streaming approach. This would require partitioning the input data stream into batches and storing the sync_info entry of the CUDA thread responsible for the

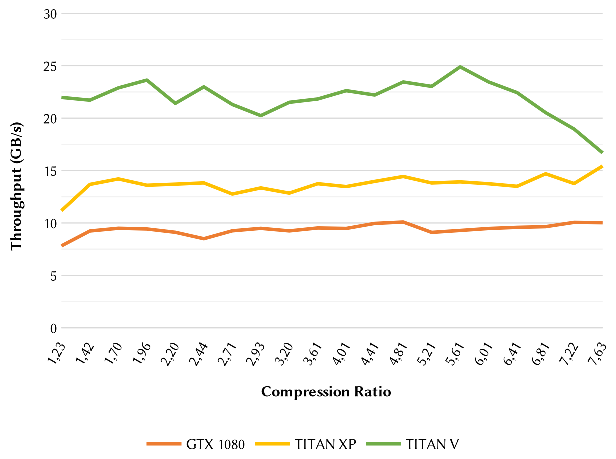

Figure 10: Throughput on various GPUs.

last subsequence of the current batch as input for the subsequent batch. Transferring decompressed data back to the CPU is often not required in application scenarios where further processing of the decompressed data is performed on the device.

Even further speed improvements could be achieved by using multiple GPUs. Exploiting this additional level of parallelism would in turn require communicating a limited amount of `sync_info` data between GPUs.

The implementation of our massively parallel Huffman decoder is available at https://github.com/weissenberger/gpuhd.

REFERENCES

[1] J. Alakuijala and Z. Szabadka. 2016. Brotli Compressed Data Format. RFC 7932. (July 2016). https://doi.org/10.17487/RFC7932

[2] C. A. Angulo, C. D. Hernández, G. Rincón, C. A. Boada, J. Castillo, and C. A. Fajardo. 2015. Accelerating huffman decoding of seismic data on GPUs. In *2015 20th Symposium on Signal Processing, Images and Computer Vision (STSIVA)*. 1–6. https://doi.org/10.1109/STSIVA.2015.7330430

[3] Y. Collet. [n. d.]. Zstandard - Real-time data compression algorithm. http://facebook.github.io/zstd. ([n. d.]). Retrieved April 03, 2018.

[4] NVIDIA Corporation. [n. d.]. Thrust | NVIDIA Developer. https://developer.nvidia.com/thrust. ([n. d.]). Retrieved April 03, 2018.

[5] T. Davis. [n. d.]. SuiteSparse Matrix Collection. https://sparse.tamu.edu/MM/Janna/Flan_1565.tar.gz. ([n. d.]). Retrieved April 03, 2018.

[6] S. Deorowicz. [n. d.]. Silesia compression corpus. http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia. ([n. d.]). Retrieved April 03, 2018.

[7] P. Deutsch. 1996. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951. (May 1996). https://doi.org/10.17487/RFC1951

[8] P. Deutsch. 1996. GZIP file format specification version 4.3. RFC 1952. (May 1996). https://doi.org/10.17487/RFC1952

[9] J. Duda, K. Tahboub, N. J. Gadgil, and E. J. Delp. 2015. The use of asymmetric numeral systems as an accurate replacement for Huffman coding. In *2015 Picture Coding Symposium (PCS)*. 65–69. https://doi.org/10.1109/PCS.2015.7170048

[10] The Apache Software Foundation. [n. d.]. Welcome to Apache Hadoop! https://hadoop.apache.org/. ([n. d.]). Retrieved April 03, 2018.

[11] A. S. Fraenkel and S. T. Klein. 1990. Bidirectional Huffman Coding. *Comput. J.* 33, 4 (1990), 296–307. https://doi.org/10.1093/comjnl/33.4.296

[12] C. F. Freiling, D. S. Jungreis, F. Theberge, and K. Zeger. 2003. Almost all complete binary prefix codes have a self-synchronizing string. *IEEE Transactions on Information Theory* 49, 9 (Sept 2003), 2219–2225. https://doi.org/10.1109/TIT.2003.815803

[13] S. Funasaka, K. Nakano, and Y. Ito. 2015. Fast LZW Compression Using a GPU. In *2015 Third International Symposium on Computing and Networking (CANDAR)*. 303–308. https://doi.org/10.1109/CANDAR.2015.20

[14] S. Funasaka, K. Nakano, and Y. Ito. 2016. Light Loss-Less Data Compression, with GPU Implementation. In *Algorithms and Architectures for Parallel Processing*, J. Carretero, J. Garcia-Blas, R. K.L. Ko, P. Mueller, and K. Nakano (Eds.). Springer International Publishing, Cham, 281–294.

[15] S. Funasaka, K. Nakano, and Y. Ito. 2017. Adaptive loss-less data compression method optimized for GPU decompression. *Concurrency and Computation: Practice and Experience* 29, 24 (2017), e4283. https://doi.org/10.1002/cpe.4283 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4283

[16] E. N. Gilbert and E. F. Moore. 1959. Variable-length binary encodings. *The Bell System Technical Journal* 38, 4 (July 1959), 933–967. https://doi.org/10.1002/j.1538-7305.1959.tb01583.x

[17] D. A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE* 40, 9 (Sept 1952), 1098–1101. https://doi.org/10.1109/JRPROC.1952.273898

[18] S. T. Klein and Y. Wiseman. 2003. Parallel Huffman Decoding with Applications to JPEG Files. *Comput. J.* 46, 5 (2003), 487–497. https://doi.org/10.1093/comjnl/46.5.487

[19] M. Mahoney. [n. d.]. Large text compression benchmark. http://mattmahoney.net/dc/enwik9.zip. ([n. d.]). Retrieved April 03, 2018.

[20] A. Ozsoy and M. Swany. 2011. CULZSS: LZSS Lossless Data Compression on CUDA. In *2011 IEEE International Conference on Cluster Computing*. 403–411. https://doi.org/10.1109/CLUSTER.2011.52

[21] R. A. Patel, Y. Zhang, J. Mak, A. Davidson, and J. D. Owens. 2012. Parallel lossless data compression on the GPU. In *2012 Innovative Parallel Computing (InPar)*. 1–9. https://doi.org/10.1109/InPar.2012.6339599

[22] J. J. Rissanen. 1976. Generalized Kraft Inequality and Arithmetic Coding. *IBM Journal of Research and Development* 20, 3 (May 1976), 198–203. https://doi.org/10.1147/rd.203.0198

[23] E. Sitaridi, R. Mueller, T. Kaldewey, G. Lohman, and K. A. Ross. 2016. Massively-Parallel Lossless Data Decompression. In *2016 45th International Conference on Parallel Processing (ICPP)*. 242–247. https://doi.org/10.1109/ICPP.2016.35

[24] W. Sodsong, M. Jung, J. Park, and B. Burgstaller. 2016. JParEnt: Parallel Entropy Decoding for JPEG Decompression on Heterogeneous Multicore Architectures. In *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'16)*. ACM, New York, NY, USA, 104–113. https://doi.org/10.1145/2883404.2883423