Parallel Huffman Decoding: Presenting a Fast and Scalable Algorithm for Increasingly Multicore Devices

Beau Johnston and Eric C. McCreath
Research School of Computer Science,
The Australian National University,
Canberra, Australia
Email: beau.johnston@anu.edu.au, eric.mccreath@anu.edu.au

Abstract—**Huffman encoding provides a simple approach for lossless compression of sequential data. The length of encoded symbols varies and these symbols are tightly packed in the compressed data. Thus, Huffman decoding is not easily parallelisable. This is unfortunate since it is desirable to have a parallel algorithm which scales with the increased core count of modern systems. This paper presents a parallel approach for decoding Huffman codes which work by decoding from every location in the bit sequence then concurrently combining the results into the uncompressed sequence. Although requiring more operations than serial approaches the presented approach is able to produce results marginally faster, on sufficiently large data sets, then that of a simple serial implementation. This is achieved by using the large number of threads available on modern GPUs. A variety of implementations, primarily OpenCL, are presented to demonstrate the scaling of this algorithm on CPU and GPU hardware in response to cores available. As devices with more cores become available, the importance of such an algorithm will increase.**

I. Introduction

Huffman encoding [1] was developed by David Huffman and provides lossless compression on a sequence of symbols. Frequent symbols are given short binary codes, whereas, symbols that occur less frequently are given longer binary codes. This reduces the length of message, compressing the data. Figure 1 depicts a Huffman tree for encoding the string "Hello World". The encoded bit string is given in Table I. To decode such a bit string one simply steps through the bits of the encoded string while following the path down the tree. When a leaf node is hit the symbol at that leaf node is output and the process continues from the root of the tree. This is repeated until the entire sequence is decoded.

This algorithm is frequently used in many video, audio and image formats such as MP3, JPEG [2] and is featured in other common compression methods such as DEFLATE [3]. Given the significance of Huffman encoding, considerable research has been undertaken to explore approaches for improving decoding performance.

Separately, the trend in general purpose compute hardware is to have more cores. A brief survey of the history of core counts in Intel CPUs show the first inexpensive desktop based dual core processor to be the Prescott-256 Celeron D processor in 2004, by 2007 the Kentsfield Core 2 Quad offered 4 cores albeit in the form of 2 dual-core dies packaged in a multi-chip module. In the Core i7 range Clarksfield offered 4 cores in 2009, Gulftown with 6 cores in 2010, Haswell had 8 in 2015 and Broadwell had 10 by 2016. Meanwhile, in on the high-end server systems, Xeon processors have offered dual and quad variants of CPUs with core counts ranging from 6 with Dunnington in 2008, 8 with Beckton in 2010, through to today with core counts of 10, 12, 14, 15, 16, 18, 22, and 24 being achievable. The Xeon Phi offers 57-core, 60-core and 61-core processor variants and started shipping in 2013. This pattern of adding more, and not necessarily faster, cores is true with all other vendors and is increasingly so with GPU devices. Given this trend, it is desirable to have a Huffman decoding algorithm that is suitable for heterogeneous multicore processors.

Some research focusing on parallel Huffman decoding has been done, most notably Wang et al. [4] focused on a hardware implementation. In this approach parallelism on the bit-level is performed by walking a Huffman tree and using a single lookup table in order to achieve impressive results of a constant rate of decoding, up to 1 code per cycle. However, the research focus was to develop specialised single-core hardware for efficient Huffman decoding, whereas the focus of this paper proposes an algorithm that can perform on general purpose hardware using concurrency.

Some approaches use constraints on the Huffman tree which enables the tree to be more compactly represented and also decoded using index look up approaches. This, for example, is the case of Huffman encoding with JPEG in which codes of the same bit length are given encodings which are sequentially ordered, thus the encoding is not stored as a tree, rather just the sequence of symbols and the number of symbols of different bit lengths is stored [5].

Lin et al. [6] develop a time efficient approach which builds tables. This enables the decoding to proceed multiple bits at a time, with entries of these tables being able to decode multiple symbols in one step. They found there was an optimal size for these tables, since a small table would perform similar to the simple tree traversing approach. As the tables became larger performance would improve, however, there was a point at which cache misses for the table lookup would be detrimental to overall performance.

Lein and Iseman [7] propose a parallel approach for decoding Huffman and also evaluated how this approach can be applied to decoding JPEG images. The idea behind the approach is that if you start decoding on a bit that is not aligned with a bit boundary you will output the wrong symbol, however, after continuing to decode a number of symbols the approach will "probably" align with the correct bit boundary and from that point on the approach will output the correct sequence of symbols. So their parallel approach divides the input data up into n sections and each section is given to a processor which concurrently decode its given section. Once each processor has completed its section of data it will continue into the next section's data fixing any of the next processors decoding errors, the processor may stop when it sees that bit boundaries are aligned. In the worst case, the first processor may need to decode the entire sequence, however, generally this would not occur.

Edwards and Vishkin [8] propose work-optimal algorithms for Burrows-Wheeler compression and decompression, in this article they also present an algorithm for decoding Huffman data in $O(\lg n)$ time. The approach they present partitions the input bit sequence up into sections of length equal to maximum encoded bit sequence, then pointers from each bit in the partition to the starting bit in the adjacent partition are calculated. Pointers are merged using a prefix sum approach which also calculates the target location of decoded symbols, finally the symbols are decoded. Noting our approach is different in that it does not partition the bit sequence and also we have implemented and evaluated the performance of our algorithm.

Patel et. al. [9] explore how GPU's can be used for compression of lossless data. This does not have the same challenge as decompression, however, it is still of interest as serial implementations of compression approach are very fast and as such it is still a challenge gain performance advantage from GPU's.

Simple changes to the storage format can greatly help the implementation of parallel approaches. An example of this is Cloud et. al. [10] which divides the encoded data into independently compressible and decompressible blocks.

Less research has focused on parallel approaches that aim to decode the entire sequence, in many respects this is not unexpected as simple serial approaches for decoding Huffman are very fast with the performance approaching limitations based on memory transfer speeds. However, as we move to hardware that has 1000s of cores and memory transfer speeds increasing we may start to find such parallel approaches will start to outperform the fast and commonly used serial approaches.

The parallel prefix algorithm (or sometimes referred to as a parallel scan algorithm) calculates the prefix sums on a list of n numbers in $O(\lg n)$ time using n processors[11]. It works by first calculating the total sum in an "up sweep" phase of the algorithm then "fills in the gaps in the calculation" via the "down sweep" phase. The parallel prefix algorithm may be generalised to any associative operator with an identity element. These algorithms have been efficiently implemented on GPUs[12], [13]. The algorithm proposed in this paper is similar in structure to a parallel prefix sum computations with both "up sweep" and "down sweep" phases, however, the operator used within our calculation is not associative.

TABLE I: The encoded binary string for "Hello World".

110	0000	01	01	001	100	0001	001	101	01	111
H	e	l	l	o		W	o	r	l	d

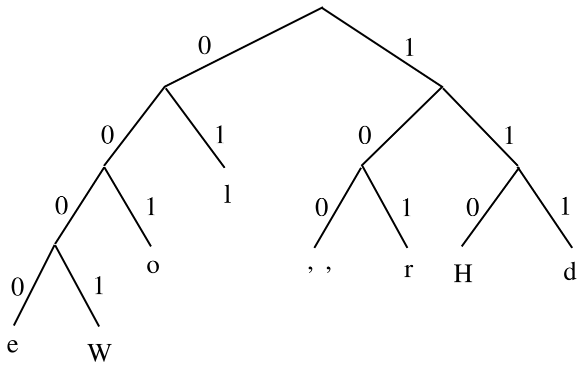


Fig. 1: Huffman tree for "Hello World".

In the following section, we describe our proposed parallel approach for decoding Huffman sequences and briefly discusses its algorithmic complexity and theoretical speedup. Section III describes the implementations developed to assess the Parallel Huffman algorithm.

Next Section IV introduces the data set and hardware used for benchmarking, additionally tuning parameters of Huffman decoding is discussed for both CPU and GPU hardware. The scaling of the algorithm on a range of CPU cores is also discussed. The performance of the parallel Huffman decoder on a range of decoding sizes is also mentioned.

Section V shows the experimental results of all implementations across the range of data sets, then an analysis of the kernels is presented so to evaluate the performance of each kernel. Next, the achieved performance is contrasted to the theoretical peak performance, then profiling occurs to determine if these theoretical deficiencies are actually shown in the experimental results. Finally, the section concludes by outlining the algorithmic and implementational deficiencies of the versions discussed.

The paper is concluded in Section VI with a discussion and some ideas for future directions this research could follow.

II. The Approach

In this section the proposed parallel Huffman decoding approach is presented. The bit string to decode is denoted H with the ith bit being H_i and $|H|$ being the length of the string. The result produced is a string of symbols with the jth symbol in the output being D_j. There are also a number of intermediate values used in the calculation. The first is the symbol when the bit sequence is decoded starting at bit i, this is denoted s_i. s_i nearing the end of the input string may not have sufficient bits to enable decoding to a symbol, in which case s_i is just set to ϵ. Noting a subsequence of s will be D. To help calculate this subsequence we also use a 2D array W where $W_{i,j}$ is the number of bits from bit i of the input which are used for encoding the next 2^j symbols. We also use an array P where P_i gives the index of D in which symbol s_i is placed. If s_i is not placed into the output, then P_i is set to -1. The minimum bit sequence length encoding is denoted h_{min} and the maximum h_{max}. h_{max} will be the height of the Huffman tree and h_{min} will be the minimum depth of the leaf nodes of the Huffman tree.

Algorithm 1 describes the parallel approach. The approach works by using 4 stages:

- Stage 1 initializes P, s, and $W_{i,0}$. The elements of P are all set to -1 except P_0 which is set to 0 as we know that the symbol decoded from the first bit will be place in the first element of the decoded string. s_i is calculated by application of Huffman decoding of one symbol using the provided Huffman tree starting at location i, the number of bits consumed in this decoding is stored in $W_{i,0}$. Noting there is no dependency between these operations and hence they may be done in parallel.
- Stage 2 creates the W table, each step uses the previous step's results to calculate the number of bits to move in the input string to go over twice as many symbols. The while loop in this stage will repeat $O(\lg(|H|))$ times. There is no dependency within instructions in the **dopar** loop as they use row $j-1$ of W to determine row j of W.
- Stage 3 uses the W table to determine which elements of s form part of the final decoded result and where to place them. The while loop in Stage 3 repeats the same number of times as did the while loop in Stage 2, using the rows of W in reverse order from which they were created. The result of this stage is recorded in P. Basically, if the symbol s_i is to be placed at index P_i within the output D then the symbol $s_{i+W_{i,j}}$ will be located at index $P_i + 2^j$ in the result. So initially, we know the location the first symbol is placed in the result (this was initialized in Stage 1, with $P_0 = 0$). Now the first time around the loop we can use W to find the location of the symbol to place approximately $\frac{1}{2}$ way along the result[1], this is recorded in P. The next time around the location of the symbols $\frac{1}{4}$ and $\frac{3}{4}$ way along the result would be found and also recorded into P, after these locations at $\frac{1}{8}$, $\frac{3}{8}$, $\frac{5}{8}$, and $\frac{7}{8}$ are found, etc. At the end of this process, all the positions are found, so P contains the indexes of where in the final result to place the decoded symbols form Stage 1. Noting if $P_i = -1$ then s_i does not form part of the result.
- Finally, Stage 4 forms the result into D. This is done using P to position symbols from s into D.

[1]The position would sit in the second half of the output sequence depending on the length of the output.

Table II show the values of s, W and P when "Hello World" is decoded.

```
input : $H$ - bit string to decode, $T$ - the Huffman tree
output: $D$ - decoded string of symbols
// Stage 1 - Initialization
for $i = 0$ to $|H| - 1$ dopar
    $P_i \leftarrow (i == 0 ~?~ 0 : -1)$
    $s_i \leftarrow$ decode $H$ starting a position $i$ using $T$
    $W_{i,0} \leftarrow$ bits consumed when $H$ is decoded from $i$
odpar
// Stage 2 - Calculate $W$
$j \leftarrow 0$
while $W_{0,j} \neq -1$ do
    $j \leftarrow j + 1$
    for $i = 0$ to $|H| - 1$ dopar
        if $(W_{i,j-1} \neq -1) \wedge (W_{i+W_{i,j-1},j-1} \neq -1) \wedge$
        $(W_{i,j-1} + W_{i+W_{i,j-1},j-1} \leq |H|)$ then
            $W_{i,j} \leftarrow W_{i,j-1} + W_{i+W_{i,j-1},j-1}$
        else
            $W_{i,j} \leftarrow -1$
        end
    odpar
od
// Stage 3 - Calculate $P$
while $j > 0$ do
    for $i = 0$ to $|H| - 1$ dopar
        if $(W_{i,j-1} \neq -1) \wedge (P_i \neq -1) \wedge$
        $(i + W_{i,j-1} < |H|)$ then
            $P_{i+W_{i,j-1}} \leftarrow P_i + 2^{j-1}$
        end
    odpar
    $j \leftarrow j - 1$
od
// Stage 4 - Set result
for $i = 0$ to $|H| - 1$ dopar
    if $P_i \neq -1$ then
        $D_{P_i} \leftarrow s_i$
    end
odpar
```
Algorithm 1: Parallel Huffman Decoding

Using the Parallel Random Access Machine (PRAM) model with an Concurrent Read Exclusive Write (CREW) strategy for addressing read/write conflicts and assuming we have $|H|$ processors then the algorithm will execute in polylogarithmic time as it will complete in $O(\lg |H|)$ steps. This is because both Stages 1 and 4 can complete in a constant number of steps. Stage 2 will repeat the while loop at most $\lg(\frac{|H|}{h_{min}})$ times, and Stage 3 repeats its while loop the same number of times.

The speedup of the parallel algorithm is $\frac{|H|}{\lg |H|}$, and thus the efficiency is $\frac{1}{\lg |H|}$ as we are assuming we have $|H|$ processors.

Comparing the parallel to a simple serial approach one very important difference is the amount of space used. So the serial algorithm uses a constant amount of space whereas the

TABLE II: Values used when decoding “Hello World”.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
H_i	1	1	0	0	0	0	0	0	1	0	1	0	0	1	1	0	0	0	0	0	1	0	0	1	1	0	1	0	1	1	1	1
s_i	H		e	e	e	W	o	l	r	l		o	l	H		e	e	W	o	l		o	l	H	r	l	r	l	d	d	ϵ	ϵ
$W_{i,0}$	3	3	4	4	4	4	3	2	3	2	3	3	2	3	3	4	4	4	3	2	3	3	2	3	3	2	3	2	3	3	-1	-1
$W_{i,1}$	7	7	7	7	7	6	5	4	6	5	6	6	5	7	7	6	7	7	6	5	6	6	5	6	5	4	6	5	-1	-1	-1	-1
$W_{i,2}$	11	13	12	13	13	12	11	10	13	12	13	13	12	13	13	12	13	12	12	10	12	11	10	-1	-1	-1	-1	-1	-1	-1	-1	-1
$W_{i,3}$	24	26	25	26	25	24	24	22	24	23	25	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
$W_{i,4}$	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
P_i	0	-1	-1	1	-1	-1	-1	2	-1	3	-1	4	-1	-1	5	-1	-1	6	-1	-1	-1	7	-1	-1	8	-1	-1	9	-1	10	-1	-1

parallel algorithm presented uses $O(|H| \lg |H|)$. In terms of the scaling of the algorithm, this quickly becomes a significant consideration.

III. Implementations

A number of different implementations were developed enabling us to analyse the parallel algorithm. The implementations were tested for correctness on all the data sets used, this was done by comparing the produced result with the original uncompressed data. This process was automated within our evaluation framework and correctness tests were carried prior to any performance timing evaluations.

A. Serial Approaches

Two serial decoding approaches were developed. A simple decoder was implemented, known as Simple, which just reads a single bit at a time and uses a table to represent the tree. This approach has no constant set up time as the tree is stored and provided to the different approaches in this very table form. As such, for decoding short strings this simple decoding approach will be fastest. Also, a more sophisticated serial decoding approach, known as Bigtable, was implemented which creates a large table with 2^m entries where m is the height of the Huffman tree. So when decoding, the next m bits are used as the index for the lookup table. The entries of this table contain one or more symbols that are decoded with these bits of the index. As the next m bits may only decode a partial number of symbols entries, the table also contains the number of bits to decode the symbols it outputs (which may be m or fewer bits). So this table is looked up, then the symbol(s) are appended to the end of the output and the bit position within the input is moved on the required number of bits. Clearly, both these approaches are $O(n)$ where n is the number of bits in the input. The table approach uses fewer instructions to process a number of bits. Yet the downside of this large table approach is it takes time to create this table and as the size of the table increases it also no longer fits within the CPU cache which adversely affects performance. The algorithm presented in [6] sits between these two approaches balancing the size of the tables created.

B. Parallel algorithm executed on a single thread

The parallel algorithm was also implemented using a single thread. Basically, the “for dopar” loops in the algorithm was implemented using ordinary “for” loops. This gives us an idea of the speedup achieved by using the GPU, as we compare this parallel algorithm executed on a single thread with that of the same algorithm executed on the 1000s of threads provided by modern GPU cards. This also gives us an idea of how much extra, or wasted, computation we are doing as we compare the simple Huffman implementation with that of parallel implementation run on a single thread.

C. OpenCL

An OpenCL implementation was developed to examine the scaling of the parallel algorithm on differing core counts of CPU and it was also used for the GPU hardware. It was useful to perform a direct comparison of the Parallel algorithm executed on a single thread from Section III-B to multiple CPU cores, this experiment is presented in Section IV-D, and the evaluation to the theoretical scaling of the algorithm against experimental performance is presented in the analysis of Section V-B. Later the same implementation is run on the GPU to see if good scaling of performance is achieved as a response to having more compute hardware.

This OpenCL implementation first copies the bit string to decode over to the device and allocates the required memory on the device side for W, s, and D. The implementation then invokes approximately $2 + 2\lceil \lg(|H|) \rceil$ kernels. With each kernel corresponding to a “for dopar” loop in the algorithm. So the invocation of the kernel via the host provides all the required synchronisation. The work was divided up into a number of workgroups, within each workgroup there was a number of threads, and each thread was assigned a number of bits for which it was responsible. The selection of optimal workgroup size is discussed in Section IV-C. Once the calculation is completed, the result is copied back to the host memory. Once the OpenCL implementation is evaluated on the GPU each of these memory transfers is increasingly taxing, this is discussed in Section V-B.

D. CUDA

A CUDA GPU implementation was developed, offering a comparison between OpenCL. This implementation mirrors that of the OpenCL version. Notationally, OpenCL workgroups are replaced with blocks, again the selection of the optimal block size was done experimentally and is discussed in Section IV-C. The timing results in the next section include the memory transfer time.

IV. Experimental Evaluation

The experimental evaluation was carried out on a high-end desktop computer with an Intel Skylake i7 6700K running at 4GHz with 4 physical cores (8 hyper-threaded) and 16GB of RAM with a memory bandwidth of 34.1GB/s. This desktop has an Nvidia Pascal GTX 1080 GPU with 2560 cores @1.6 GHz (~9 TFLOPS) along with 8GB RAM with a maximum theoretical bandwidth between this memory and the GPU of 320GB/s. All code was compiled with GCC version 5.4.0 and the host machine used a Linux Ubuntu 16.04.2 LTS Distribution with a kernel image version 4.4.0-81-generic. The CUDA implementation used CUDA version 8.0 and the OpenCL implementation was version 1.2. An OpenCL runtime on the NVIDIA GPU was provided by the CUDA 8 distribution and used the driver version provided in the package `NVIDIA-Linux-x86_64-375.66`. The Intel Skylake i7 CPU supports an OpenCL version 1.2 with the runtime version provided in the Intel `intel-opencl-cpu-r4.0-59481.x86_64` tarball.

Timing measurements presented are the minimum execution time of 25 collected runs, this is for each experimental setup. All timing measurements were collected using the `clock_gettime` function in the `CLOCK_MONOTONIC_RAW` setting and were provided by the Linux system library `time.h`.

The data sets used to evaluate the parallel Huffman algorithm are **paper1**, **news**, **book2**, and **kjv**, from [14] along with the simple "Hello World". These were compressed using the Huffman encoding and the encoded data along with the Huffman tree was stored. Table IV includes a summary of the size of these data sets. A test framework was developed which enabled different approaches to be applied to different data sets.

A. Algorithmic Scaling

A direct comparison around the overhead of the algorithm was made, this occurred by examining performance of the fastest serial implementation `serial` – bigtable, against the `PES` – parallel approach on one core. The analysis was performed on the largest dataset, `kjv`, and was the minimum time of 25 runs. For this comparison, the execution time was measured along with the number of instructions executed and total count of cycles taken to perform the computation. These hardware measurements required the use of PAPI, in particular the `PAPI_TOT_INS` and `PAPI_TOT_CYC` events. Additionally, the OpenCL version was used to examine the scaling of the algorithm in response to increasing the number of CPU cores, as is the case for all results presented on 2 or more cores.

When comparing the serial to the PES implementation of the algorithm, as shown in Table III, we see that the proposed algorithm requires 82× more instructions and 27× longer to complete. HT is an abbreviation for Hyper Threaded cores.

The number of instructions required seem to decrease when increasing the available cores. However, these PAPI measurements only present results solely on the initially instrumented core, thus the instructions required should be multiplied by the number of HT cores used. From this analysis we see that the algorithm regardless of the number of cores used consistently increases the amount of required computation by 72 – 82× on the `kjv` dataset. Yet, when we examine the decrease in execution time and the scaling of the algorithm in response to core count, it generates interest around whether improved scaling exists on a greater number of cores – as on a GPU.

TABLE III: Table: Scaling time, and hardware counters

# HT cores	Time (ms)	Instructions (10^9)	Cycles (10^9)
1 (serial)	53.312	0.269	0.228
1 (PES)	1435.316	21.375	5.688
2	1020.105	9.362	3.994
3	799.918	6.260	3.039
4	717.811	4.925	2.689
5	733.624	4.450	2.709
6	746.861	4.012	2.724
7	759.641	3.629	2.767
8	774.195	2.689	2.763

B. Enter GPUs

In order to achieve good performance on GPU architectures, we must partition the domain suitably. Selection of the optimal workgroup/block size was performed with experimental validation from considering known hardware characteristics. For instance, the GTX 1080 has 2560 CUDA cores but from the OpenCL perspective 20 OpenCL compute units are available. These correspond to the number of SMs (Streaming Multiprocessors) since each SM executes 128 threads in a block/workgroup in parallel and since 20 SMs execute concurrently (on the GTX 1080), 2560 threads/work items are being processed concurrently. Thus, we must have the minimum block size/workgroup size to be at least 128. We also, therefore, conclude that we must have at least enough work to provision 20 blocks/workgroups at any one time, or 2560 global threads active to fully utilise this hardware. Additionally, both the CUDA and OpenCL implementations support a secondary setting where each thread operates on multiple bits to decode. This was added to mitigate the overhead of thread generation, allowing a thread to perform decoding on more than one bit, this is known as the work per thread variable.

C. Tuning

Selecting the correct workgroup size significantly influences the computation times required to perform Huffman decoding, this results in teams of threads sharing memory thus choosing the best size improves cache usage. This parameter changes between each device and is sensitive to microarchitectural characteristics such as cache size and available registers.

To determine the optimal block size an experiment was conducted wherein the dataset was fixed to the **kjv** test, the work per thread variable was fixed to an arbitrary value (in this instance 64) and a trial-and-error evaluation of all block sizes were tested from 1 to 4096. However due to the limitations in CUDA on the GTX 1080 only values from 8 to 1024 executed

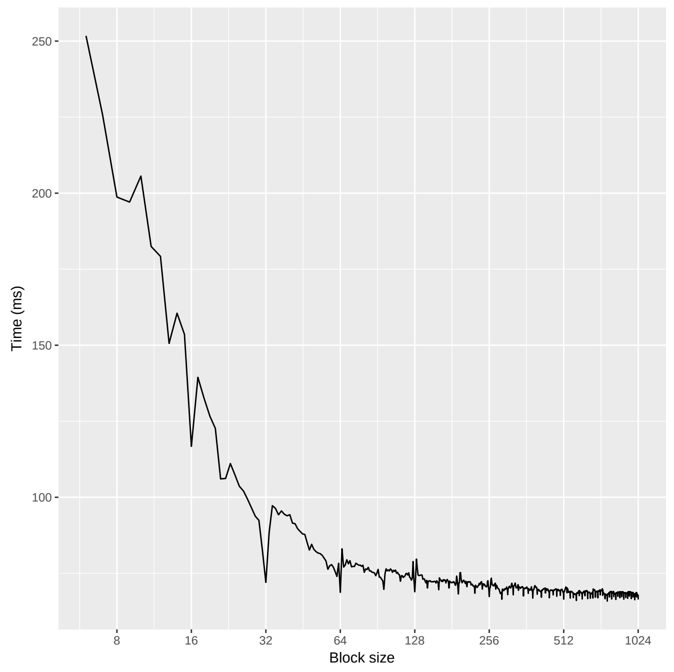


Fig. 2: Performance of the **kjv** dataset as the block size increases on the GTX 1080

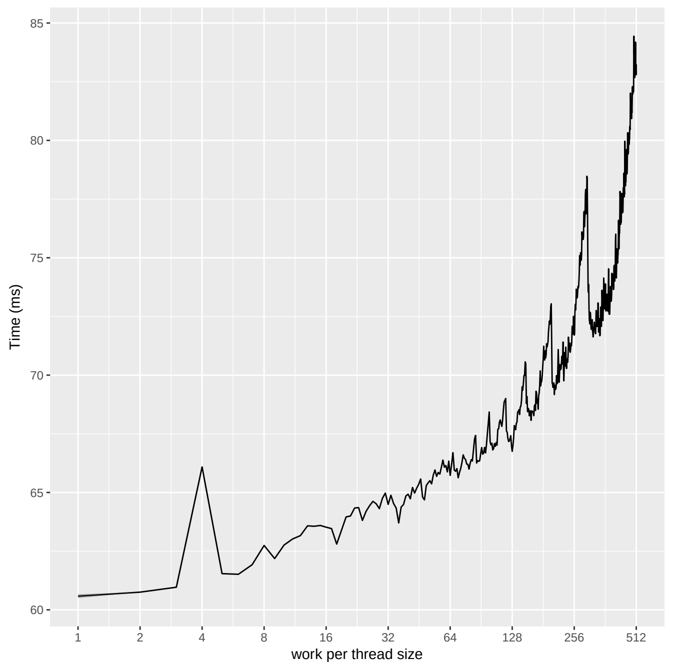


Fig. 3: Performance of the **kjv** dataset as the work per thread size increases on the GTX 1080

correctly. Any block size smaller than 8 would fail as there are more than 2^{16} blocks in one dimension, this restriction is imposed by the CUDA 2 compute capability. This is less important since the largest block size seems to yield the best performance.

Selection of the optimal workgroup/block size was performed with experimental validation from considering known hardware characteristics. For instance, the GTX 1080 has 2560 CUDA cores but from the OpenCL perspective 20 OpenCL compute units are available. These correspond to the number of SMs (Streaming Multiprocessors) since each SM executes 128 threads in a block/workgroup in parallel and since 20 SMs execute concurrently (on the GTX 1080), 2560 threads/work items are being processed concurrently. Thus, we must have the minimum block size/workgroup size to be at least 128. We also, therefore, conclude that we must have at least enough work to provision 20 blocks/workgroups at any one time, or 2560 global threads active to fully utilise this hardware. Additionally, both the CUDA and OpenCL implementations support a secondary setting where each thread operates on multiple bits to decode. This was added to mitigate the overhead of thread generation, allowing a thread to perform decoding on more than one bit, this is known as the work per thread variable. Both tuning parameters are discussed in the next section.

Figure 2 shows increasing larger block sizes for the CUDA implementation of the parallel Huffman algorithm running on the **kjv** test set. A major finding is that the larger the block size the shorter the execution time and better the performance. The GTX 1080 has a maximum block size of 1024 threads and as such, this is the optimal block size. Each SM performs the same instruction on a subset of this block size of 32 threads, known on Nvidia cards as a warp. The belief is that given the memory bound nature of the algorithm and relatively low amounts of computation, each warp performs the same instruction in the block, thus having a larger block size allows an SM to request a load or store from memory (which must miss in the cache and stall through to GPU RAM) instead of blocking and waiting for this memory access, another warp which has already loaded its values from RAM is ready to proceed. Thus, a larger blocksize allows more warps to hide memory access latency. In the OpenCL setting the optimal workgroup size is also 1024, this is not surprising as the implementation to CUDA is similar and both run on the same GPU hardware the GTX 1080.

Tuning also occurs in selecting the work per thread variable, this was initially introduced in the algorithm to mitigate the cost of creating threads which only perform work to decode one item. Setting the work per thread variable to be greater than 1 allows each thread to stride through multiple work items decoding several bits from the encoded bitstream. Following a similar experiment, the block/workgroup size was fixed to 1024, the **kjv** data set was used and a run for each work per thread was tested from the range of 1 to 4096.

From Figure 3 we can see that when the optimal block size of 1024 threads are used there is no benefit from having any work per thread. Thus, the final evaluated runtimes on all data sets are a block/workgroup size of 1024 and the work per thread being set to 1.

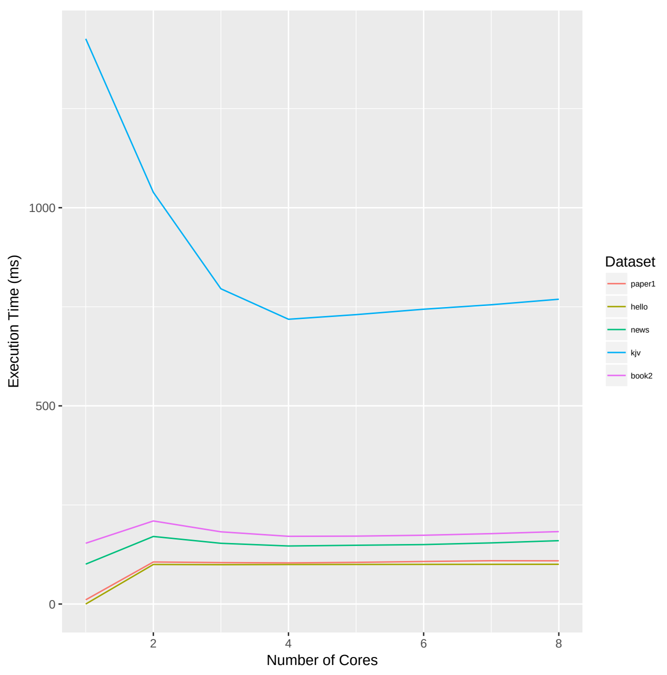


Fig. 4: Execution Times of Intel Skylake i7 6700K Cores across all Datasets, the Performance in Response to the Number of Cores

D. *Performance Scaling of Parallel Huffman in Response to Cores*

An experiment was devised to show the scaling of the algorithm, in particular, the OpenCL implementation in response to the number of CPU cores available. Tuning parameters for the CPU version were similar to the GPU version with the work per thread variable found to be optimal at 1 and the best workgroup/block size was found to be 128. Cores were physically disabled using the `cpuset` Linux tool, this allows processes to be bound to specific processors and memory node subsets. The identical OpenCL implementation was run on each configuration, from 1 to 8 cores. These findings are presented in Figure 4.

The x-axis tick that corresponds to 1 CPU core were generated using the parallel algorithm on a single thread implementation, this is why execution times for the smaller data sets appear lower than all other core counts, which were generated using the OpenCL implementation. The reason for this was a segmentation fault when querying the OpenCL platforms (during the `clGetPlatformIDs` function call) the reason for this is unknown. If it were possible to use the OpenCL implementation on one core the overhead of setting up OpenCL would be included in the computation times resulting in a more reasonable (roughly plus 100ms to Execution Time) on all one core results. Additionally, notice that execution time drops sharply for each additional CPU core used up to the fourth core. The Intel Skylake i7 6700K has 4 physical cores but 8 hyperthreaded ones, once we use more than the 4 physical cores we see a degradation in performance with execution times slightly increasing this is as there is enough work to fully utilise all the logical resources of the physical core, two hyperthreaded cores share the same compute resources of one physical core. Thus, we conclude that there is enough computationally intensive work to fully utilise all physical cores and we see no benefit of using the hyperthreaded cores, indeed we are only imposing more work in scheduling between cores.

E. *Performance over Decoding Size*

Table IV shows the decoding time for the 5 different approaches applied to the 5 different test sets. Noting the GPU approaches are comparable and in most cases better to the “Simple” approach on the larger data sets of **news**, **book2** and **kjv**. Although the “bigtable” approach performs better than either of these approaches. Notice that the CUDA implementation is on average 1.8ms faster than the OpenCL implementation, this is due to a larger overhead per kernel invocations from the host side (roughly 1μs in CUDA to 55μs in OpenCL).

Figure 5 explores the performance of the 5 approaches on the **kjv** test set as the amount to decode is increased. The leftmost graphic (a) shows absolute decompression times whereas the right plot (b) focuses on decompression times less than 100ms showing more detail on the GPU and serial decoder results. From this graph you can clearly see that the constant set up time for the “bigtable” approach is considerable, however, its linear constant factor is better than either of the other approaches. So for decoding large enough sequences this approach eventually does better than either the simple or parallel approaches. We also see that the Parallel Approach on a Single CPU core has linear scaling as work increases.

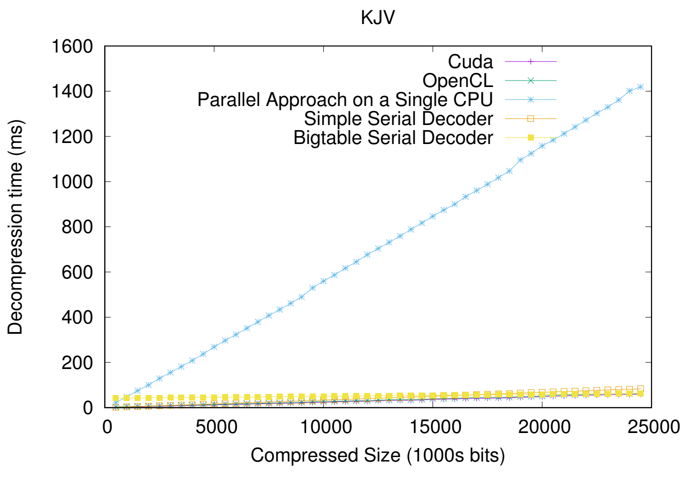
Since we see good scaling between available CPU cores as shown in Section IV-D and that good performance can be achieved using the Parallel Huffman algorithm given a suitable decoding size.

V. Results

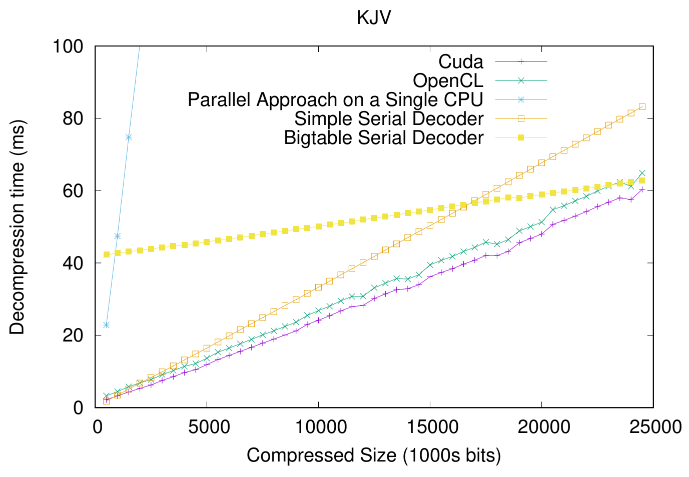
A. *Computation Characteristics*

The right-most 5 columns presented in Table V show the percentage of instrumented operations for the most frequently occurring instructions. They were generated using the instruction count feature of the `oclgrind` simulator by Price et al. [15], wherein instrumentation occurs on the Standard Portable Intermediate Representation (SPIR) of each kernel. Focus of this analysis is to see comparatively the computational structure of each kernel, from which we can infer the performance of the GPU architecture. Instrumentation is not perfect as instrumentation is on the LLVM Intermediate Representation (IR) rather than the final device specific binary, but most of the compiler optimisations have already taken place, so the same fundamental nature of computation is the same regardless of final device binary.

Across all 4 kernels we see that most of the instructions performed are computational or logical. Memory operations are in the minority, ranging from 8 - 5%. The `br` Branch



(a)



(b)

Fig. 5: Comparison of performance on the **kjv** test set as the decoding size increases

TABLE IV: Comparison of Performance

	hello	paper1	news	book2	kjv
Nodes	15	189	195	191	167
Compressed Size (bits)	32	266692	1971146	2946397	24585561
Uncompressed Size (bytes)	11	53161	377109	610856	5504597
Simple Decoder (ms)	0.0001	0.9202	7.2227	10.3469	78.8131
Bigtable Decoder (ms)	0.0003	0.7994	4.9678	7.1988	57.1711
Parallel approach on 1 core (ms)	0.0007	10.7710	101.9346	155.2715	1431.9591
OpenCL CPU Decoder on 4 cores (ms)	99.6895	103.7878	146.4306	170.8609	718.5004
OpenCL GPU Decoder (ms)	1.0115	2.4130	7.2368	9.1879	65.0892
CUDA GPU Decoder (ms)	0.4168	1.4128	5.7337	7.5538	60.5044

instruction makes up 17% of Stage 1 ranging up to 29% of the Stage 4 kernel, this simply means this branch instruction was hit and doesn't directly imply the penalty of thread divergence on GPU hardware or branch misprediction on the CPU, instead the overhead is expected to be very low as all kernels contain the same striding logic around the outer body determining the work per thread and workgroup size. `mov` instructions are quite common in all kernels 15% in Stage 1, 9% in Stage 2, 12% in stage 3 and 15% in Stage 4. Indeed, computationally intensive instructions are common across all kernels, with the `add` addition and `mul` multiplication operations typically taking 10% - 15% and 5% respectively, this is ideal, since this type of workload is typically well suited to the GPU architecture. Signed and Zero extension occurs roughly 7% of the time in Stage 1 and 2 and 3 and 4 respectively, again these instructions are essential and well-suited to the GPU. Finally, comparison instructions use up the remaining majority of all instructions with `icmp` integer comparison using 10-20% on all kernels. From examining the source code thread divergence when operating on a GPU should have negligible impact on performance.

The greatest performance bottleneck or overhead presented from this instruction analysis is the memory access operations initially discussed at 8% of Stage 2, this performance should incur greater penalties on the GPU due to slower memory access speeds for values located on GPU device memory (RAM) and is discussed in the next section.

B. Comparing Peak Theoretical and Experimental Performance

The instrumentation techniques used to acquire the results of Table V provide more than the computational composition of each of the 4 kernel stages, namely they offer exact counts of each type of instruction required to perform the decoding on the **kvj** dataset. From these instrumentation counts, we can determine the theoretical peak performance of the GTX 1080 GPU on this given workload. Whilst the oclgrind instruction counts were generated from the OpenCL version of the decoder, it is assumed that the count is similar to the CUDA implementation as these kernels are identical.

The results of Table V were generated using the optimal parameters for the GTX 1080, namely a block size of 1024 and work per thread of 1. Its assumed these numbers are still accurate when running on other devices, such as the 1080 GPU, since the fundamental nature of the task is the same and provided that the top-level IR instructions have many compiler optimisations stopping just short of mapping directly to the device specific binary instructions.

Stage 1's kernel was called twice, Stage 2's kernel was called 24 times, the kernel for Stage 3 was also called 24 times, Stage 4's kernel was called once. Thus, to get an approximation of the theoretical number of instructions required to perform Huffman decoding of the **kjv** data set, the following formula is applied:

$$\begin{aligned}\text{Instructions Required} &= \sum(\text{Stage 1}) \times 2 \\ &+ \sum(\text{Stage 2}) \times 24 \\ &+ \sum(\text{Stage 3}) \times 24 + \sum(\text{Stage 4}) \\ &= 1.2 \times 10^{11}\end{aligned}$$

where $\sum$ is the total sum of instructions from all operations.

The theoretical instructions per second of the GTX 1080 @ 1.607 GHz with 128 threads being processed on each of 20 Streaming Multiprocessors would be:

$$\begin{aligned}\text{Instructions Per Second} &= 1.607 \times 10^9 \times 128 \times 20 \\ &= 4.1e + 12\end{aligned}$$

In the more accessible metric as the Millions of Instructions Per Second (MIPS) this result is 4,113,920.

Therefore, the peak theoretical time to completion of the result on the GTX 1080 is determined as:

$$\begin{aligned}\text{Theoretical Time} &= \left(\frac{\text{Instructions Required}}{\text{Instructions Per Second}}\right) \times 10^3 \\ &= 29.6\text{ms}\end{aligned}$$

Comparing this to the achieved performance of the CUDA code which on the **kvj** dataset took 60.5 ms: $\frac{29.6}{65.8} \times 100 = 45\%$ peak efficiency compared to the OpenCL result took 65.0 ms: $\frac{29.6}{65.0} \times 100 = 46\%$ peak efficiency.

Table V shows the percent of compute time taken by each of the different stages of the algorithm. These experimental results were generated from profiling, using the Nvidia Visual profiler on the CUDA version of the decoder. It provides a closer investigation of the percentage of compute time spent in each kernel. Stage 1's two kernel call utilised 30% of the total GPU compute time, Stage 2's 24 kernel calls however used 49% of the total compute time, Stage 3's 24 invocations used 20% of all GPU compute time and Stage 4's one call utilised the GPU for 1% of all compute time. The entire computation took 65.8 ms.

Noting most of the computation time is spent in Stages 2 and 3, which involves reading and writing to the large W table. The efficiency of these memory transfers can be gauged again using the same profiling tool, with the larger writes of W (from host to device) taking 320 μs to write 3 MB resulting in a throughput of 9.6 GB/s. Many small writes from device to host are needed for each invocation in these stages, however, they have negligible effect on performance.

The final results we can take away from the profile investigation and the low ratio of achieved performance compared to the theoretical peak is that compute utilisation of the algorithm is quite low. The profile indicates as low as 25%, having such computation utilisation indicates that the multiprocessors are mostly idle given the current workload. We also see that there is low memory copy throughput, for this profile, it was at approximately 4% of the peak bandwidth of the GPU's RAM to internal memory.

C. Deficiencies

From the evaluation of performance shown in Section V-B there are some shortcomings of the parallel implementation of the algorithm when targeting the GPU architecture. Notably, the parallel method lacks overlapping memory copies which could be used to hide memory access latency. This overlapping could increase kernel concurrency and also computation which would result in a higher compute utilisation, which is greatly needed as the GTX 1080's multiprocessors were only active for 25% of the time.

VI. Conclusions

One contribution of this paper is the parallel algorithm, which at least in theory, is able to greatly improve the performance over that of serial approaches. Experimentally we have shown our approach to provide performance improvements, at least over a simple serial implementation. However, as the serial implementations are very fast there is only a small margin for improvement. This is expected to scale further in the future with the increasing core count. We show that OpenCL is a suitable implementation to demonstrate the scaling on heterogeneous devices of the parallel Huffman algorithm. The CUDA implementation is shown to be marginally faster, but this is mostly contributed to the overhead in host side API calls and there are many of these in the form of synchronisation points of the algorithm during decoding from the host.

The GPU implementation presented in the experimental section uses the parallel approach on the entire Huffman sequence to decoding. This limits the size of the sequences that may be decoded as the data, including the W table, must fix within the GPU's memory. The sequence to be decoded could be partitioned and the decoding could be streamed using a number of channels to overlap transfers and computation. Such a streaming approach would have the advantage that much larger sequences could be decoded. Also, the partition size could be increased to just saturate the GPU's parallel compute capabilities, this would minimise the depth of W table and overall reduce total memory transfers to and from the GPU's global memory.

As the computation of the algorithm is dominated by the creation and use of the W table, it is limited by memory transfer bandwidth. Looking at the W table produced for decoding "Hello World", as shown in Table II, many of the entries in this table are -1. This would generally be the case, so a simple way of improving performance would be to not store or load entries of the W table when they are known to be -1.

Another possible direction for improvement would be to attempt to compress the size of the W table reducing the memory bandwidth associated with Stages 2 and 3. This would be possible because the range of values for row j will be in

TABLE V: Profile of kernels over the **kjv** test set.

Kernel	total time	br	add	mov	icmp	other
Stage 1 - Initialization	30%	17.7%	6.5%	14.5%	13.3%	47.8%
Stage 2 - Calculate W	49%	23.7%	23.2%	8.8%	18.3%	25.7%
Stage 3 - Calculate P	20%	22.4%	16.3%	12.2%	20.3%	28.5%
Stage 4 - Set result	1%	29.0%	15.4%	15.4%	20.5%	19.5%

$[h_{min} * 2^j, h_{max} * 2^j]$, so one could store values offset from $h_{min} * 2^j$ using a reduced number of bits.

A modification to the algorithm could be made such that an associative operator for combining partial results is used. This would enable a direct implementation of the parallel prefix sum algorithm. This has the advantage of a smaller memory overhead which could improve performance. The approach also uses global memory for the tables and kernel launches for synchronisation. It would be interesting to explore how shared memory and inter-block synchronisation could be used to improved performance.

Finally, it will be interesting to see how the algorithm performs over embedded SoC devices or on a processor with an integrated GPU, such as those provided by AMD and Intel. It is expected that these devices will have better scaling since the shared memory is faster than communication over PCI-E.

In the interests of reproducibility, all code used in the generations of these findings can be found at the associated GitHub repository [16].

VII. Acknowledgements

We thank the reviewers for their valuable recommendations for improving this document.

References

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

[2] G. K. Wallace, "The JPEG still picture compression standard," *IEEE transactions on consumer electronics*, vol. 38, no. 1, pp. xviii–xxxiv, 1992.

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

[4] G. Wang, H. Zhang, and M. Lu, "Transformed hct for parallel huffman decoding," *International Journal of Circuit Theory and Applications*, vol. 43, pp. 1759—1774, 2015.

[5] G. K. Wallace, "The jpeg still picture compression standard," *Commun. ACM*, vol. 34, no. 4, pp. 30–44, Apr. 1991. [Online]. Available: http://doi.acm.org/10.1145/103085.103089

[6] Y.-K. Lin, S.-C. Huang, and C.-H. Yang, "A fast algorithm for huffman decoding based on a recursion huffman tree," *Journal of Systems and Software*, vol. 85, no. 4, pp. 974–980, Apr. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2011.11.1019

[7] 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.

[8] J. A. Edwards and U. Vishkin, "Parallel algorithms for Burrows–Wheeler compression and decompression," *Theoretical Computer Science*, vol. 525, pp. 10–22, 2014.

[9] R. A. Patel, Y. Zhang, J. Mak, A. Davidson, and J. D. Owens, *Parallel lossless data compression on the GPU*. IEEE, 2012.

[10] R. L. Cloud, M. L. Curry, H. L. Ward, A. Skjellum, and P. Bangalore, "Accelerating lossless data compression with GPUs," *arXiv preprint arXiv:1107.1525*, 2011.

[11] G. E. Blelloch, "Prefix sums and their applications," 1990.

[12] M. Harris, "Parallel prefix sum (scan) with cuda," 2007.

[13] S. Sengupta, M. Harris, and M. Garland, "Efficient Parallel Scan Algorithms for GPUs," NVIDIA Corporation, Tech. Rep. NVR-2008-003, Dec 2008.

[14] T. Bell, M. Powell, J. Horlor, and R. Arnold, *The Canterbury Corpus*, 2000 (accessed 22/8/2016). [Online]. Available: http://corpus.canterbury.ac.nz/index.html

[15] J. Price and S. McIntosh-Smith, "Oclgrind: An extensible opencl device simulator," in *Proceedings of the 3rd International Workshop on OpenCL*. ACM, 2015, p. 12.

[16] ProBeauNo, "BeauJoh/HuffmanDecoderOnGPUs: HuffmanDecoderOnGPUs V1.0," Oct. 2017. [Online]. Available: https://doi.org/10.5281/zenodo.1004749