Parallel DEFLATE Decoding using GPGPU
COMP4560 - Single Semester Project

Matthew Whittaker

May 29, 2015

Contents

1 Why parallel DEFLATE decoding?

Data compression is everywhere. It reduces the costs of transmission and storage, by forming a more compact representation of data. One of the most popular compressed formats is DEFLATE. Examples of where DEFLATE occurs include .ZIP files, GZIP files, PNG image compression, and in the the Linux kernel.

There are many fantastic methods for encoding data more compactly and faster, even with simple formats such as DEFLATE. However, data is typically decompressed much more than it is compressed. An object needs to only be compressed once, but may be decompressed many times and by many different people. Despite this, it is not obvious how one can improve on decompression strategies, as encoded data formats are typically very prescriptive in how they map to decoded data.

We seek to improve on current decompression strategies, by modifying the algorithms to run in parallel over graphics processor cores.

2 The project's goal

This project's primary goal was to implement and assess the viability of highly parallel GPGPU (general purpose graphical processing unit) decompression. We focus on the DEFLATE compressed format, and hence on Huffman decoding which DEFLATE uses heavily.

This involves discovering whether or not any similar decoders exist, researching existing algorithms, designing my own algorithm, implementing this algorithm, and finally assessing the outcome: whether any speed up was or could be gained, and whether or not the whole exercise is viable in a practical context.

3 Background topics

There are several areas where some background knowledge is required to understand the approaches and context of this project. The main ones are Huffman encoding, LZ77 compression, the DEFLATE compressed format and OpenCL. We provide a brief discussion of each, which should be sufficient to understand the project. Extra material is suggested, but in all cases Wikipedia[13] provides a good start.

3.1 Huffman encoding

Huffman encoding is a widely adopted compression technique, which under certain assumptions (most notably statistical independence of symbols) is provably optimal[3]. The idea is that we have a string of symbols, and we wish to decide on a mapping of symbols to code-words (strings of 1's and 0's). Huffman's compression algorithm tells us how to choose these code-words such that the expected length of a coded string is minimised. The general strategy behind

this (and literally any other loss-less compression technique) is to assign shorter codes to more common things.

Beyond this, how the algorithm works need not be discussed, as we are interested in decoding, not encoding. For information on the encoding process, please see the Wikipedia page[13], but this is not required for a basic understanding.

What we must know is that an encoded string has an associated mapping from code-words to symbols, and given this mapping and the string we are able to decode the string. Mapping from code-words to symbols is typically represented as a tree (a Huffman tree). This is a binary tree, where if we begin at the root of the tree, and traverse the tree depending on the bits in our input string (say 0 for left, 1 for right), by the time we reach a leaf node, the value stored at this node will be the decoded symbol.

The Huffman algorithm ensures that no code-word will be the prefix of another. This ensures that if we start at the start of our code string and traverse through as we decode, we never incorrectly decode the start of one code-word to the symbol of the prefix code-word. This means that decoding from start to finish will always yield the correct result. This decoding process is highly sequential: we are forced to decode from start to finish. If we try and decode somewhere in the middle of our input data, we are likely to produce rubbish. This makes the job of parallel decoding non-trivial.

3.2 Lempel-Ziv 77[5]

LZ77 is a compression technique where to compress data we either store literal symbols, or we store a $(length, distance)$ pair. The pair tells us to look back $distance$ symbols in our already decoded data, and copy $length$ symbols from there to our decoded stream. This technique is highly effective for repetitive data.

We said that Huffman encoding is optimal where symbols are statistically independent, but it is very typical in the data humans and computers produce for things to be more likely to occur if they have already occurred recently. This violates independence, so Huffman encoding cannot take advantage of repetitiveness.

For example, if we have the input symbols 'a,a,b,a,a,b,b,a', Huffman encoding will produce an encoding that is a permutation of the encoding of 'a,a,a,a,b,b,b,b'. Hence, it will encode the two strings just as effectively. What we notice though, is that 'a,a,a,a,b,b,b,b' does not exhibit independence: where an 'a' has occurred, it is likely to occur immediately after, and similarly for 'b'.

In this example, LZ77 will produce output for the first string like: 'a,a,b,(2,3),b,b,a', but for the second string will produce 'a,(3,1),b,(3,1)', which is much shorter (if we encode our tuples efficiently). Interestingly we note here that LZ77 is a generalisation of the popular run length encoding[4].

3.3 DEFLATE

3.3.1 What is DEFLATE?

DEFLATE attempts to combine Huffman encoding with LZ77, to take advantage of data where some symbols are more common than others, and where repetitiveness is present. This is great for general purpose compression, but will fail (if used naively) for compression of data that exhibits other dependencies between symbols.

For example if we try to compress a photograph using DEFLATE, we will not find much repetitiveness, and the LZ77 aspect will not be effective. However, as is typical in photographs, adjacent pixels (both horizontally and vertically) are likely to be *similar*. So maybe DEFLATE would be effective if we apply some sort of differential transformation to the photograph before running DEFLATE. This is exactly what PNG encoding does[11].

When we discussed Huffman encoding and LZ77, techniques were described, but actual data representation was purposely vague. DEFLATE is a data format that tells us precisely how to store our Huffman trees, LZ77 tuples and encoded data, so that it is both compact and can be reliably decoded.

The process for decoding DEFLATE data is quite involved, so we will not go into exact detail. For specifics, I direct you to the actual standard[1]. It is not too long, and is quite clear. I based my implementations[17, 18] entirely on this document.

3.3.2 Technical details of DEFLATE

DEFLATE breaks data up into chunks. Each chunk is either compressed using a combination of Huffman and LZ77 techniques, or is a raw chunk, which just presents the literal data. The purpose for this is to make the compression more 'dynamic'. If the input's symbol probability distribution changes, or its dependencies change, the chunking allows Huffman trees to be updated, or for blocks that cannot be compressed to be left raw. Another advantage of this is that to do Huffman encoding, two passes over the data are required. This is not always possible, where not all data in the stream is available. Using chunks means that encoding can begin when not all input data is present.

Each chunk has some bits that define the chunk type (raw or compressed), as well as a bit that says whether or not it is the last chunk in the stream.

A raw chunk contains the size of its data in bytes, followed by that size of raw data.

A compressed chunk is much more elaborate. First, we will discuss how a compressed chunk represents its Huffman trees. The DEFLATE format puts two constraints on the Huffman trees used (this is taken verbatim from the RFC[1]):

- All codes of a given bit length have lexicographically consecutive values, in the same order as the symbols they represent;
- Shorter codes lexicographically precede longer codes.

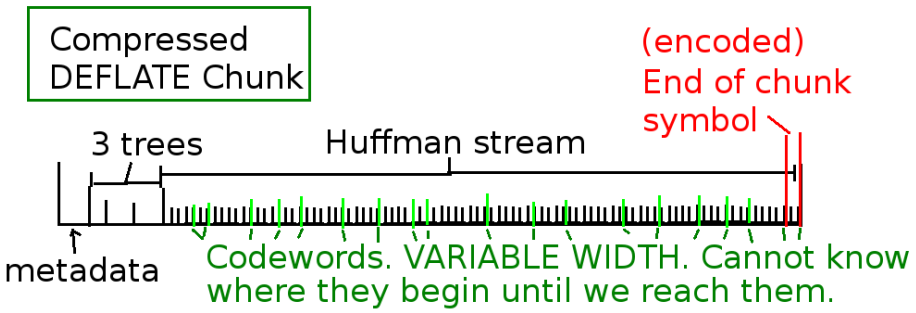

Figure 1: What each chunk looks like. A DEFLATE stream is nothing more than a concatenation of these chunks.

It can be shown[1] that given these constraints, and given the length of each symbol's code-word, one can uniquely build a tree from those lengths. This is powerful because it means DEFLATE only needs to provide a list of lengths, which makes tree representation very compact.

Each DEFLATE chunk uses three Huffman trees, and at the beginning of a chunk, the number of symbols that actually appear in the data are provided for each of these trees.

Then, the first tree is provided using the code length encoding described. This forms a tree, where each symbol in the tree represents either a code length or some repetition of code lengths. This tree is used to encode the code lengths of the other two trees. In this way our Huffman trees are encoded themselves using a Huffman tree.

The literal/length tree is then decoded using the tree just discovered. Each symbol in this tree represents either a literal byte, an end of block symbol or a LZ77 length.

The distance tree is then encoded using the first tree. Each symbol in this tree represents a LZ77 look-back distance.

The actual encoded data follows. Symbols are decoded from this data until an end of block symbol is found, at which point if this chunk had the last chunk bit set , the data is fully encoded. Otherwise, a new chunk, follows. If this new chunk is compressed, it will contain a new set of trees which is important to note when doing parallel decoding, as it makes it much more difficult.

If a decoded symbol is a literal byte, this is simply put into the output and a look-back buffer. If a decoded symbol is a LZ77 length, a predefined number of bits are then read which are added to the length represented by the symbol. A single symbol is then read from the distance tree that was built. Depending on the symbol found, a predefined number of bits are then read which are added to the distance represented by the symbol. Then *length* bytes from *distance* far back in the look-back buffer are output, and each output byte is also put into the look-back buffer.

Huffman symbols are used to either represent literal bytes, or LZ77 encodings. This is the technique used to combine the two compression strategies, into something more flexible.

Everything described about DEFLATE was fully accurate, but not fully detailed. For full detail, please read IEEE's RFC[1].

3.4 OpenCL

OpenCL (Open Computing Language) is a framework that allows one to execute the same code across many computing platforms such as CPUs and GPUs. The OpenCL standard is provided by The Khronos Group.

The only reason to use OpenCL is the performance benefit that these more difficult to interface with devices can provide. In our case, we use it to execute code on GPUs, which are able to run code in parallel over a huge amount of cores (the number is unclear as OpenCL does not specify how workers map to GPU units, and GPU designers are very secretive). This can result in incredible performance benefits if we are able to approach our problem in a way well addressed by the GPU architecture.

OpenCL defines library functions as well as a programming language that is like a basic version of c. The library functions provided allow one to query platforms, and use these platforms to compile and build 'kernels', which are the unit of computation OpenCL uses. These kernels are then uploaded by the platform software to one or more devices to be executed.

The platforms (ICD, for installable client driver) are often proprietary software, such as drivers provided by AMD for their graphics card. The ICD loader is the library that provides a standard interface to the system. It is the ICD loader that our code links to.

For our purposes, we upload kernels to a GPU, which we pass data to. These kernels are run on thousands of cores at the same time, and each core executes the same code. The only difference between each worker is that each worker has a different ID, which it uses to decided exactly what work it is to do.

It is difficult for workers to communicate with one another, and difficult for different workers to perform hugely differing tasks. OpenCL excels in performing the same computation on lots of different pieces of data at once, but for more complex problems it can suffer.

It is also worth noting that OpenCL provides no dynamic memory management. For example if we don't know how big our output will be, or how big some data will be ahead of time, we can run into issues.

4 Possible approaches

It is not clear how one should approach the problem of parallel DEFLATE decoding, due to the highly sequential nature of the data. Each chunk's data is a Huffman stream, so one seems to be forced to decode from the start to the end, linearly. Decoding in the middle of a Huffman stream is likely to yield

garbage output, unless you happen to begin decoding at the start of a code-word. However, there is no way to know for sure where a code-word starts until you have decoded all previous data.

Even worse, each chunk uses a different set of Huffman trees, and there is no way to know when a chunk ends unless you decode all the data up to that end, as the end of chunk symbol is also encoded. This means that not only might starting in the middle of the stream not be at the start of a symbol, but the Huffman trees could have changed, or that data might represent more Huffman trees or raw data, etc.

The approaches we will discuss here focus on decoding the data stream in parallel, but leave the trees to be decoded by the CPU. Decoding the trees represents a small amount of work compared to decoding the stream due to their small, bounded size, so this is a small concession.

4.0.1 Decoding everywhere

In this strategy, the nth worker begins decoding at the nth bit. In this way all the workers focus their effort at the beginning of the stream. Once each worker decodes a symbol, the workers must detect the correctly decoded symbols and aggregate their results before sending them back to the CPU. How this should be done in a way that utilises all the workers, and is of reasonable time/space complexity is not obvious.

This method was not investigated more, in order to pursue the more obvious and simpler strategy to be described next.

4.0.2 Breaking the stream up

This is the strategy that is further investigated in this report. The first strategy was described to present an alternative, and as a thought exercise.

In this strategy, the Huffman stream is broken up into evenly spaced chunks. Each worker begins decoding at the start of their assigned chunk. Most workers will produce garbage output, but interestingly they will usually naturally synchronise. If the worker finishes decoding a symbol in a bit position where a true symbol started, it will have synchronised. If each symbol has a chance of synchronising, then as the worker decodes, the chance of it having not synchronised yet decreases exponentially.

Synchronisation cannot always occur however. If an unsynchronised worker finishes decoding a symbol at the same place as a symbol begins, this implies that the last code-word they used is a suffix of the last actual code-word. This means that for synchronisation to occur, the set of code-words cannot be suffix free. It turns out though, that the vast majority of encodings are not suffix free[6]. In the rare case that the encodings are suffix free, the algorithm will work, but only workers that begin on a symbol boundary (such as the first worker) will do work. This implies a speed decrease.

The workers must use some data structures or communication mechanisms to detect when they have synchronised with the worker ahead of them, and then

stop.

The primary problem with this strategy is that breaking the stream up is wasteful when a worker begins past the end of the current chunk. There is no way to know how large a chunk is before it is decoded, and so we cannot focus our efforts only on the current chunk.

There are two approaches to resolving this problem. Firstly, estimating the size of the chunk is not that hard. There is overhead in each chunk for the trees stored, so this tends to mean chunks must be above some critical mass to have any gain over using a raw chunk. Also, the two encoders tested use predictable chunk sizes. Golang's inbuilt DEFLATE encoder uses always close to 16KB chunks, and GNU's GZIP, which probably uses zlib uses always close to 32KB chunks.

If we can accurately guess for example that chunks are 32KB, and we have 1000 workers, then each worker will have 32B of data to work with. We can do some simple calculations to estimate the probability of each worker synchronising under certain assumptions. If we assume that each symbol is about 4 bits long, then each worker will have 64 symbols to work through. We can estimate the probability of synchronisation at each symbol as 286/287, as there are 287 symbols, and so if input and output streams are independent, and each symbol is uniformly probable (not actually true but okay for estimates). Then the chance of a worker synchronising is about $1 - (286/287)^{64} = 20\%$. This means that about 20%, or 200 workers get work done, which could be potentially be a huge amount faster than using a single CPU, even with such small chunks.

The other approach to resolving the chunking problem is to use encoders that don't chunk often. They would produce output that is compatible with current decoders (i.e matches the DEFLATE standard), so switching to such encoders is easier than using an entirely new format. However, this would forgo the advantages of chunking.

5 Important data

Before fully designing an algorithm that implements the chosen strategy, important information had to be gathered. The primary information required are the disk IO speeds, current decoder speeds, encoded chunk sizes as created by typical encoders, and the overhead introduced by GPU memory transfer latency and throughput. The information found, as well as the relevance of this information will be discussed in the following subsections.

5.1 Disk IO speeds and current decoder speeds

For many applications, the stream to be decode will reside on disk. It must be read from disk, decoded, and rewritten to disk. This overhead cannot be improved on by parallelising decoding, but we can expect it to be improved upon by future disk/caching technologies.

It is important to have a good idea of disk overhead, because if it dominates the decoding times of existing decoders, then a performance increase cannot be expected by speeding up decoding unless streams do not reside on disk.

To measure this, simply transferring data from one file to another is not sufficient. We do not know what caching is happening, and we do not know whether or not decoders can read as fast as a program copying over large blocks.

To better measure disk IO, two decoders were run. The first run was not recorded. The second run was recorded. The operating systems disk cache was then cleared. This can be done on Linux with something like[7]:

```
echo 3 | sudo tee /proc/sys/vm/drop_caches
```

A final run was then recorded.

Huge differences were observed between the two recordings. We can infer that the disk IO time was *at least as large but maybe larger than* the difference in times. The lower bound proved to be sufficient.

A program[14] was written to wrap Golang's library DEFLATE decoder implementation[8]. GZIP was used to test zlib's decoder implementation. GZIP[9] is a compression format and program that is a simple wrapper around DEFLATE. It uses zlib[10] underneath, and so the GZIP decoder was used to test zlib's DEFLATE implementation. The results are as follows:

Implementation	Cached (s/GB)	Cache Clear (s/GB)	Implied Overhead (s/GB)
Golang	6	22	14
GZIP (zlib)	8	20	12

Notice that disk overhead dominates decoding time. This signals that there is not much to gain by parallelising, unless we are reading and writing to memory, which is a niche use case.

5.2 Typical chunk sizes

Before a DEFLATE chunk can be decoded, three Huffman trees must be decoded. The Huffman trees are of a bounded size, whereas the Huffman stream is not, hence tree decoding does not require a large amount of computing resources. The complexity of decoding the trees is substantial though. Hence it would be inappropriate to do this decoding in the limited and complex GPU environment, but rather it should be done on a single thread on the CPU.

The downside of this is that rather than sending all data to the GPU at once, we must send it in parts. An OpenCL kernel, chunk data and decoded streams must be sent for each chunk. This implies the same throughput overhead, but a potentially large latency overhead, as this approach would require many GPU transactions.

To estimate the size of this overhead in terms of seconds per gigabyte, we must know how large encoded chunks typically are, so we know how many transactions occur per gigabyte. We must also know the GPU memory transfer latency, which will be discussed in the next subsection.

Three programs were written to measure this overhead. A wrapper[15] for Golang's DEFLATE encoder[8] was written. A program was also written[16] to extract DEFLATE streams from GZIP files, according to the GZIP specification[2]. These two programs were used to produce encoded streams typical of Golang and zlib's encoder implementations.

Finally, a program[17] had to be written to decode a full DEFLATE stream using only the CPU. This was a huge undertaking, as it required full understanding of the complex format[1], and requires substantial tight, error prone, bit level code. It is a necessary exercise though, as learning the format properly is required before implementing a parallel version, and aids in debugging and testing the parallel version. This CPU decoder was used to report on the chunk sizes of Golang's and GZIP's encoded streams, produced by the previous two mentioned programs. This decoder was tested by decompressing compressed video files, and checking for differences. Video files are the recommended test case for decoders because they are large, and they contain a variety of compressed and non-compressed data.

The results were as follows:

Encoder	Encoded Chunk Size (kB)	Chunks/gB
Golang	16	62500
GZIP (zlib)	32	31250

These figures were very consistent across different files. The encoders were evidently targeting these chunk sizes.

5.3 GPU memory transfer

As discussed in the previous subsection, it is important to know GPU memory transfer latencies. It is also important to know throughput constraints: if throughput is too small, no approach will be viable as the whole encoded stream *must* be send to the GPU, and the decoded stream *must* be sent back.

Figures were drawn from a paper on the subject [12]. GPU memory latency was always close to $10\mu s$, and throughput was typically around $5gB/s = 0.2s/gB$. If we multiply the number of chunks per gigabyte by the latency, we find the total latency overhead per gigabyte. We also assume a decompressed size of $1.5\times$ input size. This is an assumed typical compression amount, no meaningful data can be gathered on actual compression rates as they are totally application specific. Throughput overhead is therefore estimated to be $2.5 \times 0.2s/gB = 0.5s/gb$.

Encoder	Total Latency Overhead (s/gB)	Total Throughput Overhead (s/gB)	Total GPU Transfer Overhead (s/gB)
Golang	0.625	0.5	1.125
GZIP (zlib)	0.3125	0.5	0.8125

The memory transfer overhead is in both cases around a second. Compared to the previously found decoder speeds of $8s/gB$ and $6s/gB$, this is small. There is plenty of room for our approach to save time, even with the latency overhead produced by sending data to the GPU one chunk at a time.

6 The chosen approach in detail

The approach used was based on the approach described in section 4.0.2. The results obtained in the previous section showed us that we can decode the metadata and trees for each chunk on the CPU, and the chunks one by one to the GPU without significant performance penalty.

The trees are to be decoded as discussed in section 3.3.2. These trees will be passed to the GPU as well as an estimated whole chunk. We should aim for our estimated chunk to be just larger than the actual chunk size. The details of your encoder's chunk sizes can be checked using the decoder provided[17]. If unsure, it is recommended that you send 17kB to your GPU. At this value, the majority of Golang's DEFLATE streams will be decoded in a single transaction, and GZIP's will be decoded in two.

The estimated Huffman chunk as well as the trees are to be sent to the GPU. The GPU should decode either up to the end of the stream, or as much as it can. If it did not receive the whole stream it should signal that the CPU send more data using the same trees. If it did receive the whole stream, it should tell the CPU how much of the input was consumed, so that the CPU knows where to begin decoding the next chunk.

At this point the approaches the CPU should take in decoding and interfacing with the GPU as well as the scope of the CPU's responsibilities have been described. All that is left is a description of exactly how the GPU can decode the data it is given in a highly parallel way, which is the heart of the problem.

6.1 The chunk decoding algorithm important notes

Please read and understand section 4.0.2 well before proceeding. Several aspects of what is required to make this approach concrete are described. Then, the algorithm is given in pseudo-code.

6.1.1 Synchronisation buffer

The workers are to each decode a section of the chunk, and will tend to synchronise. A mechanism for detecting synchronisation is required. The simplest solution, and the one adopted is to have a buffer with one byte for each bit in the input buffer. When a worker fully decodes a symbol, it places a token into the synchronisation buffer, in the position corresponding to the start of that code-word in the input chunk.

This serves several purposes:

- When a worker begins decoding a symbol, they check the bit position in the synchronisation buffer, and if they find something there, they know the symbol they are about to decode has been decoded already, and they have synchronised. They can then stop working, after taking a few measures to be discussed.

- If a worker sees an end of stream token in the input buffer, and is not yet synchronised, then the worker is no longer trying to synchronise with the worker directly in front of it, but with the worker two in front of it. This allows it to correctly update its data structures.

- More subtly, if a worker is in the middle of decoding a symbol, and it sees a token for a literal or LZ77 symbol in the buffer, it will use this to keep track of where the next worker is outputting at that point. This is necessary, because once we detect synchronisation, we have to know the position of the next worker's output buffer at that synchronisation point so that we can produce a full, sequential output.

The memory overhead of this buffer is equal to the size of the input data times 8. We recommended an input chunk size of 17kB, so this is an overhead of 136kB. Note that this data does *not* need to be sent to or from the CPU, it exists only in the GPU's personal memory, which is often gigabytes in size. Thus the overhead is negligible.

6.1.2 LZ77 representation

When a worker detects an LZ77 tuple, it will not be expanded. Instead, four bytes will be written to the output containing the length and the distance. The motivation for this is offloading complexity onto the CPU. The justification is that for the CPU to unroll these tuples is hardly (if at all) more expensive than copying data byte by byte. For example for the CPU to copy literal bytes from the output to disk, it will enter a tight, efficient loop. For the CPU to unroll LZ77 tuples from the output to the disk, the size and distance must be read, then it will enter a loop with a very similar number of machine instructions to write the bytes to disk.

The CPU needs a way to distinguish between a literal byte and a LZ77 tuple. The encoding used is as follows:

- LZ77 is encoded by writing a 0 byte, followed by the length-3 (to put the value between 0 and 255), followed by the distance in the next 2 bytes.

- Literal byte is encoding by writing a non-zero byte followed by the literal byte.

This allows us to put an upper bound on the size of the output buffer, which makes the limited memory management on the GPU easier to deal with. It is not clear whether or not this increases the data to be sent back from the GPU to the CPU. If the output data is heavy in literals, the size will be doubled, but

if it is heavy in LZ77 encodings, the size could be reduced drastically. Further analysis on this topic is suggested if using this approach, but time did not allow it.

6.1.3 Intermediary output buffers

Each worker needs a dedicated output buffer. We do not know the size of the output ahead of time, so these buffers must be sufficiently large. No dynamic memory management (think malloc()) is available by default. Thus, we find an upper bound on the size of the output and simply allocate this amount statically.

The upper bound is as follows: each literal byte can be encoded using as little as 1 bit, and is encoded on the output as 2 bytes. Each LZ77 byte can be encoded using as little as 2 bits (1 for the literal/length tree and 1 for the distance tree. They are encoded on the output as 4 bytes. Hence, in the worst case scenario our output is 2 bytes for every input bit.

However, two output buffers are required for each section. The reason is that once a worker finishes decoding its section, it moves into the next worker's section to try and synchronise. It cannot write to that worker's buffer though, as that data may be needed.

This puts our output buffer size at 4 bytes per input bit, or $32\times$ our input size. This seems like a huge overhead, however at 17kB of input data, this means output buffers of only 544kB which is insignificant compared to the memory available. This does pose a large problem though on how to send this data back to the CPU.

- This data could be sent back directly. This would increase our throughput requirements by $32\times$, which would create a totally crippling $0.2 \times 32 = 6.4$ seconds (see speed discussion section) of transfer time per gigabyte: more than is required by current decoders to the decoding.

- This data can be aggregated into a final output buffer, which is sent to the CPU. It should be aggregated in parallel by the workers, the logistics of which should not be complicated, as it is natural to copy buffers to other buffers in parallel. Not enough knowledge could be gathered about transferring data to and from a GPU using OpenCL to give a concrete description on how this is to be done. Further research on this topic is suggested, before implementing this approach.

Each output buffer must have associated with it the following meta-data. It must record the next output buffer used: when a worker synchronises with some worker ahead of it, there must be a natural transition from its output to whatever buffer that worker was using. It must also record the position in the next buffer of the following data. It must also record its size: how much of it has been used so far.

6.2 Chunk decoding pseudo-code

The following pseudo-code is to be executed by each worker at the same time on the GPU. It is assumed that each worker has a unique ID number between 0 and $no\_workers - 1$ which it is aware of, as is true in OpenCL.

A high level pseudo-code will be provided. It is intended to provide an more concrete idea of what is happening, but hides the intricacies of which there are a lot. After this, a very concrete and almost directly implementable pseudo-code will be provided.

6.2.1 High level code

```
decode:
    while true:
        if i am just starting this symbol, check the synchronisation buffer:
            if this buffer has an entry at my input location, link
            my current output buffer to that location and return

        if i am just starting this symbol, and my input position
        is in a new worker's input section, then:
            increment my output buffer, and set up a link from
            my last buffer to this one

        get a new bit
        if this bit completes my symbol
            if this is an end of chunk symbol, then stop
            else
                sync_buffer[symbol_start] = output_position;
                put the symbol in the output buffer
        otherwise continue
```

This did not mention how to figure out exactly where the output is at when you have synchronised, as this requires a lot of complex manipulation.

6.2.2 Implementable code

Lots of things in the following code will seem strange or unnecessary. Implementing this algorithm is wrought with subtleties, so if anything seems arcane, it is probably being done for a reason. A full CPU decoder[17] was implemented, as well as a single worker GPU decoder[18] which was to be adapted to a multi-worker decoder. Unfortunately time did not allow for this, and so this algorithm was not implemented and there may still be subtle bugs in it. Regardless, it provides a very strong starting point for someone wishing to pursue the subject.

```
decode(buffers,id,worker_count,input,literal_tree,distance_tree):
    buffer_location = id*2 /* 2 buffers per id */
    current_buffer = buffers[buffer_location]
```

```
next_workers_buffer = buffers[buffer_location+1]
next_workers_offset = 0

section_size = input_size*8/worker_count
input_pos = input_size*8/id /* input position in bits */
section_start = input_pos

start_of_symbol = true

codeword = 0

/* each iteration of this loop reads a bit and checks for complete symbol */
while true:
    if start_of_symbol:
        symbol_start = input_pos

        if sync_buffer[input_pos]: /* check synch buffer when start symbol */
            /* we have synchronised, set our buffer to point to next workers */
            current_buffer.next = next_workers_buffer
            current_buffer.next_offset = next_workers_offset
            current_buffer.size = buffer_position
            /* this is important */
            /* write zeros incase the last worker in this section wrote 1s */
            write_zeros_to_end_of_section(sync_buffer)
            return
        else:
            if input_pos-section_start > section_size :
                /* finished section, move onto next output buffer */
                /* wait so each worker works on 1 section at a time */
                WAIT_FOR_OTHER_WORKERS()
                ++buffer_location
                current_buffer.next = buffers[buffer_location]
                current_buffer.size = output_pos
                output_pos = 0
                section_start += section_size

    /* use sync buffer information to track next workers output position */
    else:
        case sync_buffer[input_pos] of:
        0 => do nothing
        literal =>
            next_workers_offset += 2 /* 2 bytes per literal */
        lz77 =>
            next_workers_offset += 4 /* 2 bytes per lz77 */
        end_of_chunk =>
```

```
        /* the next worker finished before we synchronised with it */
        /* try to synchronise with the next next worker instead */
        /* each worker is always 2 ahead of the previous one */
        /* it is difficult to see that this works without subtle bugs */
        next_workers_buffer = next_workers_buffer.position + 2
        next_workers_offset = 0

if next_workers_offset >= next_workers_buffer.size:
    next_workers_buffer = next_workers_buffer.next
    next_workers_offset = 0


start_of_symbol = false

/* this is important */
/* write zeros incase the last worker in this section wrote 1s */
sync_buffer[input_pos] = 0

/* read the next bit of the current codeword */
codeword.push(get_bit(input,input_pos++))
symbol = literal_tree[code_word]

/* did codeword form a symbol? */
if symbol != null:
    /* reset these two */
    codeword = 0
    start_of_symbol = true

    /* we found a symbol */
    if symbol.is_literal():
        /* record in sync buffer */
        sync_buffer[symbol_start] = literal
        /* output byte */
        current_buffer[output_pos++] = 1
        current_buffer[output_pos++] = symbol
    if symbol.is_lz77():
        /* finish decoding length */
        length = symbol + get_extra_bits(n) /* for value of n see RFC */
        distance_codeword = 0
        /* another loop to decode distance */
        while true:
            distance_codeword.push(get_bit(input,input_pos++))
            sync_buffer[input_pos] = 0
            distance = distance_tree[distance_codeword]
            if distance != null:
                distance += get_extra_bits(n) /* for value of n see RFC */
```

```
            break
        sync_buffer[symbol_start] = lz77
        current_buffer[output_pos++] = 0
        current_buffer[output_pos++] = length-3
        current_buffer[output_pos++] = low(distance)
        current_buffer[output_pos++] = high(distance)
    if symbol.is_end():
        sync_buffer[symbol_start] = end_of_chunk
        current_buffer.size = output_pos
        current_buffer.next = null
        /* this is important */
        /* write zeros incase the last worker in this section wrote 1s */
        write_zeros_to_end_of_section(sync_buffer)
        return
```

7 Analysis of performance increase

We have not shown a concrete implementation of parallel DEFLATE decoding that improves performance. We have shown however that given you are reading and writing from a magnetic hard disk, current decoding takes about 22 seconds per gigabyte, with at least 16 of this devoted to IO. Therefore the speedup cannot exceed around $6/22 \approx 30\%$.

If the data does not reside on disk however, but in memory, we have shown that GPU transfer times are small enough that even with a chunk by chunk decoding strategy, overhead is small enough that time can be gained if the GPU implementation is fast enough. GPU transfer overhead was about 1 second, which leaves potentially 5 seconds to be gained over current 6 second per gigabyte decoding. This could be a substantial gain, but it depends on how well a decoder can be implemented.

8 Practical analysis

- The intention of GPGPU platforms such as OpenCL is to run independent algorithms on multiple pieces of data at once. Huffman decoder however is highly sequential and highly dependent. The lack of memory management and general ill fit of the problem means that to implement this well will require a lot of time and resources. This is compared to the small speedup to be gained, unless you are in a niche application where you read and write your streams from memory.

- OpenCL, or any GPGPU computing platform introduces nasty dependencies. Relying on such a platform will reduce your application's portability, and introduce the large footprint of typically proprietary binaries. This means more trust must be outsourced to developers of GPU drivers, who are infamous for their lack of transparency.

- To properly implement parallel Huffman decoding on a GPU introduces *huge* complexity into your application. The difficulty of implementing this compared to a singled threaded CPU version cannot be emphasised enough. This complexity decreases maintainability and is very likely to introduce bugs into your software: more time spent for a less reliable solution.

9 Future developments

Future advances in technology, could potentially change the situation in several ways. These are of course purely speculative but a discussion is important.

It is very likely that significant speedups will come to disk access. This is primarily due to the increasing prevalence of solid state drives, but also advances in hardware and operating system (think caching) technologies will probably lead us to IO speed increases. This could make Huffman decoding viable, even for applications where streams are on disk.

CPU performance gains are experiencing diminishing returns. It is widely believed that the future of faster computers is in parallelisation. This means we can expect to see more appropriate (not intended for graphics) hardware platforms, that are more open better supported. It also implies increased research into tools such as compilers, languages and platforms that may make parallel programming more accessible or maybe in some cases transparently implemented.

This has two implications. Firstly, in the short term, more parallel hardware developments compared to single threaded hardware developments means more speed gains, giving our research an advantage.

More importantly though, as parallel hardware becomes more prevalent and accessible, we are likely to see compressed data formats that have some concession built in for parallel decoding. For example DEFLATE streams have their end of chunk symbol encoded. What if the chunk boundaries were told to us at the start of each chunk? Suddenly we know the chunk boundaries ahead of time and can simply put one worker on each chunk. This would renders our research redundant, as nobody wanting to do parallel decoding will be interested in DEFLATE if parallelisable alternatives are popular.

10 Recommendations

The best recommendation, that applies to the majority of applications is simply: don't bother. There are many reliable, bug free and fairly fast open source single threaded CPU decoders. Stand on the shoulders of the open source community and save yourself time and money.

If speedups are absolutely necessary, and you think parallel decoding is only the way to achieve that, then use another format. DEFLATE was clearly not

designed to be decoded in parallel, but could be trivially modified to make parallel decoding much easier.

If your application absolutely requires you decode DEFLATE streams, and is time critical, and your streams reside in memory, then try multi-threated CPU decoding. The potential speedups are less, but the development is substantially easier, and papers on the subject exist[6].

If you are still fixated on GPU decoding, and you have a significant economic incentive, then move forward at your own risk. It is still not clear whether or not substantial gains can be made, but it is clear that they will not be made without expertise and time. Such a development project would present a large risk and is likely to result in a buggier, less reliable product, if not failure.

11 References

[1] L. Peter Deutsch of Aladdin Enterprises, *RFC 1951 (DEFLATE)*, IEEE, May 1996.

[2] L. Peter Deutsch of Aladdin Enterprises, *RFC 1952 (GZIP)*, IEEE, May 1996.

[3] Wikipedia *Huffman coding*, `http://en.wikipedia.org/wiki/Huffman_coding`,

[4] Wikipedia *Run-length encoding*, `http://en.wikipedia.org/wiki/Run-length_encoding`

[5] Wikipedia *LZ77 and LZ78*, `http://en.wikipedia.org/wiki/LZ77_and_LZ78`

[6] S. T. Klein and Y. Wiseman, *Parallel Huffman Decoding with Applications to JPEG Files*, The Computer Journal, British Computer Society, February 2003.

[7] *How do you empty the buffers and cache on a Linux system?*, `http://unix.stackexchange.com/questions/87908/how-do-you-empty-the-buffers-and-cache-on-a-linux-system` Stack Exchange.

[8] Golang, *Package flate*, `http://golang.org/pkg/compress/flate/` Google.

[9] Wikipedia *gzip*, `http://en.wikipedia.org/wiki/Gzip`

[10] Wikipedia *zlib*, `http://en.wikipedia.org/wiki/Zlib`

[11] Wikipedia *Portable Network Graphics*, `http://en.wikipedia.org/wiki/Portable_Network_Graphics`

[12] Rune Johan Hovland, *Latency and Bandwidth Impact on GPU-systems*, Norwegian University of Science and Technology, December 2008.

[13] *Wikipedia*, `https://en.wikipedia.org/wiki/Main_Page`

[14] Matthew Whittaker, *artefact/aux/deflate.go*.

[15] Matthew Whittaker, *artefact/aux/inflate.go*.

[16] Matthew Whittaker, *artefact/aux/gzip.c*.

[17] Matthew Whittaker, *artefact/cpu_single_threaded*.

[18] Matthew Whittaker, *artefact/gpu_single_threaded*.

12 Appendices

12.1 Study Contract

A copy of the signed version was not made. If verification is required, Weifa can provide it.

Australian National University

INDEPENDENT STUDY CONTRACT

Note: Enrolment is subject to approval by the projects co-ordinator

SECTION A (Students and Supervisors)

UniID: __u5011509_____

SURNAME: _____Whittaker_____ FIRST NAMES: _____Matthew_____

PROJECT SUPERVISOR (*may be external*): ____Eric McCreath_____

COURSE SUPERVISOR (*a RSCS academic*): _____

COURSE CODE, TITLE AND UNIT: _____COMP4560 ___Advanced Computing Project (12 Units)_____

SEMESTER ☒ S1 ☐ S2 YEAR: _____2015_____

PROJECT TITLE:

Parallel Decompression using a GPU

LEARNING OBJECTIVES:
The student would gain a good understanding of GPGPU software development.
With a focus on looking at performance and scale relating to the decompression of DEFLATE.
Also it is expected that the student would gain general skills relating to: writing a report, and giving a seminar.

PROJECT DESCRIPTION:
The project will involve implementing parallel decompression of the DEFLATE format on a GPGPU. This will involve exploring different possible approaches and evaluating the overall performance and the bottlenecks in performance. This will be compared with the performance of current CPU implementation.
+ research current approaches for both parallel and serial implementation of the algorithm and include a summary of this in the report,
+ understand the algorithm and implement a simple serial approach,
+ provide a description of DEFLATE in the report,
+ consider different approaches that may be taken for parallel versions of the algorithm, complete

Australian National University

some simple analysis of them,
+ implement a GPU parallel version of DEFLATE,
+ evaluate the performance in terms of time and space, and
+ write the report.

ASSESSMENT (as per course's project rules web page, with the differences noted below):

Assessed project components:	**% of mark**	**Due date**	**Evaluated by:**
Report: name style: _____ (e.g. research report, software description...)	___45__ (60%)		
Artefact: name kind: _____ (e.g. software, user interface, robot...)	___45__ (30%)		
Presentation:	__10___ (10%)		

MEETING DATES (IF KNOWN):

STUDENT DECLARATION: I agree to fulfil the above defined contract:

…….. ……..
Signature Date

SECTION B (Supervisor):

I am willing to supervise and support this project. I have checked the student's academic record and believe this student can complete the project.

…….. ……..
Signature Date

REQUIRED DEPARTMENT RESOURCES:
Access to a GPU for completing performance evaluations.

SECTION C (Course coordinator approval)

Australian National University

…….. ……..
Signature Date

SECTION D (Projects coordinator approval)

…….. ……..
Signature Date