Algorithms for Compression on GPUs

Anders L. V. Nicolaisen, s112346

Tecnical University of Denmark
DTU Compute

Supervisors: Inge Li Gørtz & Philip Bille

Kongens Lyngby , August 2013
CSE-M.Sc.-2013-93

Technical University of Denmark
DTU Compute
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@compute.dtu.dk
www.compute.dtu.dk CSE-M.Sc.-2013

Abstract

This project seeks to produce an algorithm for fast lossless compression of data. This is attempted by utilisation of the highly parallel graphic processor units (GPU), which has been made easier to use in the last decade through simpler access. Especially nVidia has accomplished to provide simpler programming of GPUs with their CUDA architecture.

I present 4 techniques, each of which can be used to improve on existing algorithms for compression. I select the best of these through testing, and combine them into one final solution, that utilises CUDA to highly reduce the time needed to compress a file or stream of data.

Lastly I compare the final solution to a simpler sequential version of the same algorithm for CPU along with another solution for the GPU. Results show an 60 time increase of throughput for some files in comparison with the sequential algorithm, and as much as a 7 time increase compared to the other GPU solution.

Resumé

Dette projekt søger en algoritme for hurtig komprimering af data uden tab af information. Dette forsøges gjort ved hjælp af de kraftigt parallelisérbare grafikkort (GPU), som inden for det sidste årti har åbnet op for deres udnyttelse gennem simplere adgang. Specielt nVidia har med deres CUDA arkitektur formået at gøre programmering til grafikkort enklere.

Jeg præsenterer 4 teknikker, der hver især kan bruges til at forbedre allerede eksisterende kompressionsalgoritmer. Gennem test udvælger jeg de bedste, og sammensætter dem til én samlet løsning, der benytter sig af CUDA til kraftigt at nedsætte tiden nødvendig for at komprimere en fil eller strøm af data.

Til sidst sammenligner jeg den endelige løsning med en simplere sekventiel udgave af samme kompressionsalgoritme til CPU'en samt en anden GPU-løsning. Resultatet viser mere end 60 gange forøget hastighed for enkelte tilfælde af filer i forhold til den sekventielle algoritme, og op til 7 gange for den anden GPU-løsning.

Preface

This master's thesis has been prepared at DTU Compute from February 2013 to August 2013 under the supervision by associate professors Inge Li Gørtz and Phillip Bille in fulfilment of the requirement for acquiring an M.Sc. in Computer Science and Engineering. It has an assigned workload of 30 ECTS credits.

Acknowledgements

I would like to thank my supervisors for their guidance and incredible patience during the project. A special thanks to my girlfriend who waited for me in the new apartment 300 kilometres away. Also a thanks to old DDD3, who made me go through with this level of education.

Anders L. V. Nicolaisen

August, 2013

Contents

1 Introduction

Compression addresses the issue of reducing storage requirements of data and still maintaining the integrity of the information or what seems to be the same data.

When data needs to be interpret by human senses, compressed data may not need to be a complete representation of the original sequence of bytes. This is called *lossy compression* and is used on pictures and music, where some information can be omitted and still seem to be similar to the original by the human eye or ear.

Lossless compression is the process where information is not lost - mostly because the integrity of the data cannot tolerate it to be reduced. Instead, the data is stored differently, but maintaining the original information when decompressed accordingly. The basic principle is, that any non-random file will contain duplicated information, and by using statistical techniques on these duplicates,

Lossless compression is a widely researched area and is used in a variety of applications. One of the largest fields of application is the transfer of data over I/O channels. Bandwidth is always limited and pose as bottleneck, so a decrease in information needed to be transferred is always an improvement. This is especially the case with servers of webpages. In order to reduce the I/O needed for transferring the webpage or other files to the client, the webserver often use one or more of the most widespread compression methods, such as GZIP or BZIP2. As the names suggest, they both are somewhat related. In the case of these two, they, to some extent, implement the initial works of Lempel and Ziv from 1977 [17], also known as LZ77. The client then apply the reverse function of compression (decompression) to the downloaded material and ends

up with the original collection of data to be read or executed.

1.1 This Report

In this thesis I propose an algorithm for lossless data compression utilising graphic processing units (GPU) to attain a substantial speed-up and relieve the CPU for other tasks. The output format of the algorithm is required to be readable by simpler clients, which might not have a GPU at disposal, so the format should be able to be reversed by slower CPUs. Preferably should the output format be well-known and widely implemented, such that additional software is not needed.

The focus will be on fast compression of files and not fast decompression, as the format should be usable by already known algorithms.

1.2 Preliminaries

This section introduce some of the notation used throughout the report.

1.2.1 Notation

The term *character* (**char** in ANSI C) is often used throughout this report and represents an octet of bits. Implementations exists where a **char** is of 16 or 32 bits, but in this report it is always referred to as 8 bit.

Unless otherwise stated, all data (and files) are perceived as a one-dimensional array in convention with the ANSI C form: `type identifier [size] = {first, second, third, ..., size-1}`, with starting index at 0 (zero). Splitting files or data can simply denote dividing into several arrays or keeping multiple boundary indices. *Length* of data will therefore be defined as the size of the one-dimensional array.

The abbreviations *GPGPU* for "general purpose graphic processor unit", and the simpler *GPU*, are used arbitrarily throughout the report, but reference the same thing.

Also the term of *compressing* information can also be referred to as *encoding*. Even though the two words ordinarily have different meaning, in this context they both characterise the notion of reducing space requirements of data.

1.2.2 Evaluation

- **Speed**: The amount of time for the entire execution. It would in some cases be sufficient to measure the kernel execution of the GPU isolated, but some preliminary and subsequent work is usually needed, so the *speed* or *throughput* is measured as the total accumulated milliseconds; or seconds where appropriate.
- **Ratio**: The actual compression in percent. This is the difference between the size of the compressed output compared to the input data.

Formally we define:

- Execution time for preliminary work (T_{pre})
- GPU Kernel execution time (T_{kernel})
- Execution time for post-processing (T_{post})
- Count of encoded characters (N_{enc})
- Length of original data (N_{org})

Speed and *Ratio* can be defined as:

$$\begin{aligned} \text{Speed} &= T_{pre} + T_{kernel} + T_{post} \\ \text{Ratio} &= (N_{enc} \cdot 100)/N_{org} \end{aligned}$$

For both speed and ratio does it hold, that less is better. If a ratio is 100% or more, then no compression has been done, and the original input should be returned by the algorithm.

2 Lossless Data Compression

Two major approaches are used in lossless data compression: Statistical and dictionary methods. Statistical methods such as the *Run-Length Encoding* analyses the input and encodes it in terms of the *running length* of a character, word or sentence. See this example:

$$\begin{aligned} \textit{Input} &: AAABBCCDEEEEEEAAAAAAAAAAAAAAAAA \\ \textit{Output} &: 3A2B2C1D6E18A \end{aligned}$$

This form of compression is especially useful when working with highly-redundant data like images.

Dictionary methods keeps prior analysed portions of the text to search for existence of a current word and try to replace it with a smaller reference to the previous encounter. Examples hereof can be seen in chapter3.

2.1 Previous Works

I start by presenting an overview of the previous work in implementing and extending the algorithms for universal data compression, divided into sections of sequential and parallel solutions.

2.1.1 Sequential Solutions

Lempel and Ziv 1977 [17] (LZ77) used a dictionary encoding technique with two buffers: a sliding window search buffer and an uncoded look-ahead

buffer. More on this in chapter 3.

Lempel and Ziv 1978 [28] (LZ78) differ from their previous approach by constructing an explicit dictionary, that does not need the decoding of the entire corpus for reference, and can as such be used to do random lookup during decompression.

Welch 1984 [27] (LZW) this extension of the LZ78 removes redundant characters in the output, which then consist entirely of pointers. Welch also introduced variable-encoding, which further reduced the space requirement for the pointers, as the first element in the dictionary only took up 1 bit, and whenever all bit-positions per element where exhausted, an extra bit got introduced to the coming elements, until some prescribed maximum. Having an explicit dictionary proved efficient when the input had a final alphabet, such as the colours of an image. This led to the usage of LZW in the Graphics Interchange Format (GIF).

Burrows and Wheeler 1994 [5] presented a new variant of Huffman coding, which proved to have speed improvements compared to implementations of Lempel-Ziv at the time and still obtain close to best statistical compression. They used a technique to divide the input into smaller instances, and processing these blocks as a single unit. using simple compression algorithms

2.1.2 Parallel Solutions

Multithreading data compression can be done by splitting the input data up into several chunks, preferably into the same number of threads as available cores, and let each thread do the exact same sequential work, and after processing, merge the chunks back into a complete result.

This process can be done in disregard of the memory hierarchy, memory latency and synchronisation of the executing system, but would not be particularly efficient, so most implementations take these considerations into account, and do further division of workload to accommodate each level of memory during execution.

2.1.2.1 Parallel Solutions for CPU

All of the following use the idea of splitting up the work needed into the number of cores available and using POSIX Threads (Pthreads) for processing.

Gilchrist 2003 [10] extended the approach used in the **BZIP2** block-sorting compression application [24] and let blocks of input data be processed through the Burrows-Wheeler Transform[5] simultaneously on multiple processors using Pthreads. For each additional processor to distribute the algorithm in parallel, a speedup was seen. Trying the algorithm with 2 processors with steps of 2 up to 20 processors, the speedup was near linear and performed 88% better than the original **BZIP2** as baseline.

Adler and Gailly 2004 [19, 1] included into the **ZLIB** compression library in 2004 the **PIGZ** (pig-zee) algorithm, which is a parallel implementation with Pthreads using **GZIP**[18] in the same approach as Gilchrist.

Klein and Wiseman 2005 [14] improved the encoding and decoding times of Lempel-Ziv schemes such as LZSS and LZW. Found improvement in compression over the simple parallelisation, however, with a greater execution time. With 4 parallel processors, the proposed method only gained approximately a 2x speedup over the sequential implementation.

Kane and Yang 2012 [13] utilizing multi-core processors to parallelise block compression using the Lempel-Ziv-Oberhumer (**LZO**)[20] to pursue a performance proportional to the number of processor cores in the system. Gaining a 3.9x performance speedup on a quadcore processor without degradation of compression ratio, with the possibility for a speedup of 5.4x when compression ratio is not of primary importance.

In 2004 patented Google a "Parallel Compression and Decompression System"[9]. A piece of hardware designed for the reduction of data bandwidth and storage requirements for in-memory data to be used in a computer architecture. The purpose of the system, which consists of several compression/decompression (codec) *engines* to execute in parallel or in streams, to relieve the CPU from running software implementations. The *engines* of the system could in theory be using different kind of compression algorithms, but the patent itself only shows a variable-length encoding scheme implemented in hardware.
The purpose of the system was to be implemented in server structures to further utilise volatile memory and non-volatile storage. However, with the advent of GPGPUs in servers, and not just in desktop computers, the exact same could be attained by using the GPU-architecture to reduce the load of the CPU - and GPUs can even be used by much more versatile algorithms instead of just compression schemes.

2.1.2.2 Parallel Solutions for GPU

The use of GPGPU in algorithmic design is still a relatively new area of research, and therefore not many solutions has been made in the field of data compression. However, some ports of compression algorithms onto GPU have been made, and the following are the most, though relatively little, cited.

Notice the publication years of the research; it gives an indication of how recently GPGPUs have gotten the attention of scientific computing.

Balevic et al. 2008 [3], 2010 [2] parallellising the inherently serial *Variable-Length Encoding* onto a GPGPU. The paper presents the novel algorithm *parallel VLE* (*PAVLE*), which gives a 35x-50x speedup compared to a serial *Huffman code*[11] implementation.

Eirola 2011 [8] addressing the problem of scaling implementations from parallel CPU cores onto the GPU efficiently. As the GPU has hundreds of processors and a non-general memory accessing scheme, splitting the data into smaller chunks may not be feasible. Eirola singles out the parts of *BZIP2* compression and suggests GPU implementations for each. He does, however, not combine them in a single implementation, and therefore lacks the final results of speedup.

Ozsoy et al. 2011 [22] developed what they called *CULZSS*, short for CUDA LZSS, and it used a simple sequential search. They used the implementation of Dipperstein[7] without modifications to the algorithm itself, but they used the same approach as Gilchrist[10] and divided the input on kernel level into blocks and processed each block individually on multiple cores in the CUDA architecture. Two different versions of this approach were made, with the difference of how the matching of substrings in the lookup buffer were made. Speedup of 18x achieved compared to the serial implementation and 3x compared to a parallel CPU implementation of LZSS.

Ozsoy et al. 2012 [23] improved the work they had done on *CULZSS* and optimised the matching of the algorithm. This improvement was not a result of a newly devised algorithm, but rather a better use of the architecture of the CUDA framework. Furthermore, they found that the best block size of the input correlated with the memory available for each thread block in the GPU. The algorithm resulted in a 46.65% performance improvement compared to the serial versions of *GZIP* and *LZIP*, however, with a loss of compression ratio. The lesser compression of *CULZSS* is due to the additional use of *Huffman coding* after the initial *LZ77* in the *GZIP* and *ZLIB* algorithms, which ensures close to statistical compression.

The overall conclusion of the GPU implementations has been, that the best optimisations can be produced by reconstructing the algorithm for better utilisation of the GPU architecture and CUDA framework, and not necessarily by rethinking a different algorithm.

Even though, summarising the approaches, it can be seen, that all the implementations are based on already widely utilised compression schemes, and most of these are some variations of the works of *Lempel-Ziv*.

2.2 Approach Based on Previous Works

In this section I will outline the approach of the investigations in this project. I will use the original findings of Ozsoy et al.[22] to reproduce their results and further develop the algorithms used.
Especially will it be investigated if the choice of LZSS-implementation based on the work of Michael Dipperstein[7] could be made more efficient. Better results may be achieved by using the approach of others.

The original *CULZSS* will also serve as baseline for the evaluation of the final implementation as well as the sequential CPU implementation.

Even though both speed and compression ratio is part of the evaluation, speed (or *throughput*) will be the key element of success for a given improvement.

It is also imperative, that the output of the work is usable by simple, sequential CPU implementations, so decompression does not rely on the presence of a GPU if a stream or file is to be sent to another party.

3 Definition of LZSS

As can be seen in chapter 2, many implementations derive from the LZ77 algorithm. This is not due to superiority over LZ78, but because of the LZ78, and its derivatives, became patent-encumbered in 1984 by Unisys when they patented the LZW algorithm[6]. The patent expired in 2003, but did in the meantime put further research on hold.

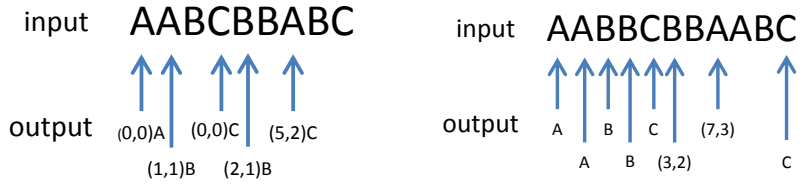
The encoding process of LZ77 is more computationally intensive for encoding with fast decoding, whereas the LZ78 balances resources between both encoding and decoding, and have a better compression ratio. Consequently, the two algorithms can be used in different scenarios, where data is to be decoded often in the case of LZ77, or data is seldom decoded and therefore should use less resources for storage in the case of LZ78.

`LZSS` (Lempel-Ziv–Storer–Szymanski) is an improvement of the `LZ77` with the difference of using a single prefix bit to represent whether the next bytes are a part of a reference or a single (uncoded) character. The length of the found common sequence with LZSS during encoding is ensured to always be greater than the size of a reference pointer - a minimum matching length - and with this invariant, the algorithm renders better compression, than on what it is based, as the `LZ77` had an overhead where reference pointers were longer than what they substituted, as the algorithm always outputted a reference pointer along with a, often redundant, character [26] (see figure 3.1).

The *reference pointer* is a code pair of the form:

`<B,L>` where $B = offset, L = length$

The *offset* is sometimes referred to as the start *position* of the match, but it is implemented as a an offset, which is the term used in this report.



(a) LZ77 **(b)** LZSS

Figure 3.1: Output example of the original LZ77 versus LZSS. Notice, that on LZ77 every output is a reference pointer with either a $length = 0$ and the current character, or $length > 0$ and the next character. The output for LZSS is with a minimum matching length of 2

The minimum matching length can be variable for different implementations, but it is crucial that the following is true:

$$min_{match} \geq \text{size}(\texttt{<B,L>})$$

where the function `size()` returning the number of bytes needed to represent the input.
This indicates a relation between `B` and `L`, as their combined size optimally should be a multiple of eight bits to utilise the entire byte. If a packing of bits are used (see section 6.2), then a calculation of

$$\text{size}(\texttt{<B,L>}) = m \cdot 8 - 1, \qquad \text{m} = \text{number of bytes}$$

could be used, as the prefix bit for representing the reference pointer could be a part of the byte.

This practise of using the least significant bit as an indicator for the following byte(s) has later been adopted by several other applications, one of the more recently is the Protocol Buffers[1] from Google and as a storage optimisation at Twitter[2].

3.1 The Compression Algorithm

The original dictionary encoding technique used in LZ77 (and in extension, LZSS) consists of two buffers: a sliding window search buffer and an uncoded look-ahead buffer (see figure 3.2). The search buffer is also called *history* buffer,

[1] https://developers.google.com/protocol-buffers/docs/overview
[2] https://blog.twitter.com/2010/hadoop-twitter

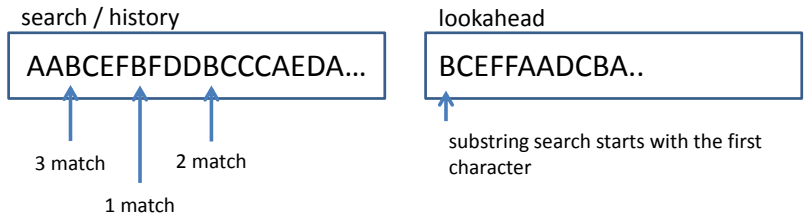


Figure 3.2: Example of the matching stage with sliding buffers

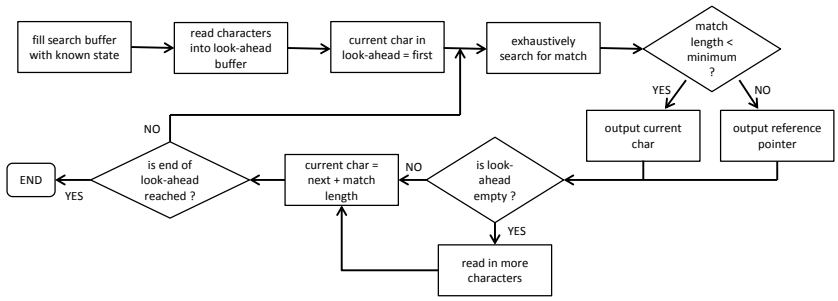


Figure 3.3: State diagram over the LZSS compression process

as it holds the "recently seen" characters.
As these buffers both "slide" over the same set of data - each processed character from the look-ahead will be put at the end of the search buffer - they are merely theoretical and can be implemented as a single buffer with pointers defining their boundaries.

3.2 Different Possible Methods for Searching the Dictionary of LZSS

As the LZSS is a dictionary compression, several methods for speeding the lookup can be applied, and has been in various variants of the LZ-family.

In the following I will describe some of the general lookup/search algorithms, that has been used in the LZ-family

- **Sequential search** With a search buffer of length h, and a word to search for w, renders a total search time $O(h \cdot w)$.

- **Knuth-Morris-Pratt**[16] A somewhat significant linear search optimisation of $O(n + w)$, with a lookup table precomputed over string length n and word length w.
- **Hashtable** Has a search complexity of $O(1)$, however, this comes at a cost of calculating hash-values, which always will be of at least w.
- **Binary search trees** Average search complexity of $O(log(n))$.
- **Suffix trees** The same theoretical lookup as KMP due to an initial built tree: $O(n + w)$.

All of these, except for the linear sequential search, promise faster lookup complexity, however, on the cost of memory. This makes them undesirable in some applications such as embedded systems, where memory is scarce. The sequential search can furthermore ensure fixed size memory, due to the bound buffers. Parallellising these methods also proves difficult, and the question always arises: *Is the benefits of larger memory footprint for faster lookup and the CPU cycles needed to maintain the structures really great enough for not using the simple buffer-approach?*
In some cases, the benefit does not show when parallellised, and Ozsoy et al.[22] states that in massive parallel architecture of GPUs, it seems better to use as simple structures as possible.

4 Graphical Processing Units

In this chapter I introduce relevant aspects of GPU architecture, especially focusing on the CUDA framework.

GPUs are specialised in compute-intensive and highly parallel computation, where a CPU is optimised for long-lived single-threaded applications. A modern CPU has several ALUs (arithmetic logic unit) for performing arithmetic and logical operations - a GPU has a considerable multitude more ALUs, which makes it especially well-suited for data-parallel computations with high arithmetic intensity.

CUDA (Compute Unified Device Architecture) is somewhat high-level GPU-programming and therefore seems simpler to programme, even though the standards OpenCL and OpenGl is much more widespread and implemented by a large variety of hardware manufacturers, whereas CUDA is a solely implemented by nVidia. This limits the usage of CUDA-code to graphic cards with nVidia technology, but benefit from the simplicity.

4.1 CUDA Architecture

A CUDA GPU is composed of a number of *streaming multiprocessors* (SM), each having a number of compute units called *streaming processors* (SP) running in lock-step. This enables SIMD-style execution of many concurrent threads (Single Instruction Multiple Data), but is, however, refined on the GPU into the SIMT (Single Instruction Multiple Thread). Instructions are issued to a collection of threads called a *warp*. As warp of threads execute in lock-step, the execution is most efficient when all threads of a warp agree on

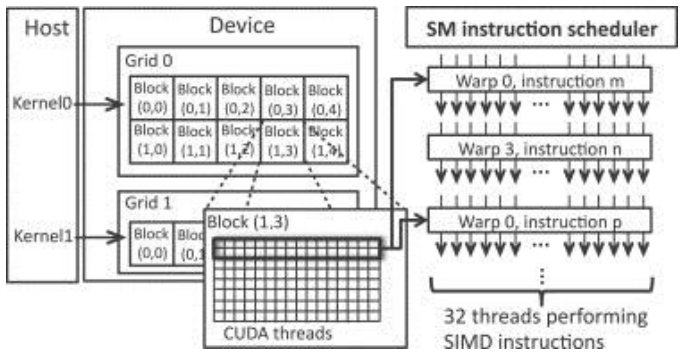


Figure 4.1: Taxonomy of the CUDA work partitioning hierarchy [1]

their execution path.

The CUDA programming model provides two levels of parallelism: coarse and fine-grained. On the *grid-level* is the coarse partitioning of work done by dividing the problem space into a grid consisting of a number of blocks. A block is mapped to a symmetric multiprocessor of which holds several threads at the fine-grained *thread-level*. The number of threads per block is limited by the GPU and in the case of nVidia GeForce GT 620M it is set to 256. Every block of threads can cooperate with each other by sharing data through shared memory and thread-*synchronisation* within a single block only.
A *warp* is the term of execution of a block of threads that are physically executed in parallel and is also a defined by the GPU in terms of how many threads can be executed concurrently, commonly 32.
A *kernel* is the set of functions and parameters that define the instructions to be executed.

4.2 Memory

The CUDA memory hierarchy is pictured in figure 4.2. *Registers* are the private memory per thread, while all threads within the same block can access the *shared* memory. All threads, disregarding block grouping, have read and write access to the *global* memory, and read-only of the *constant* and *texture*. Each type of memory has their justification as outlined in table 4.1.

[1] http://www.sciencedirect.com/science/article/pii/S0743731513000130
[2] `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html`

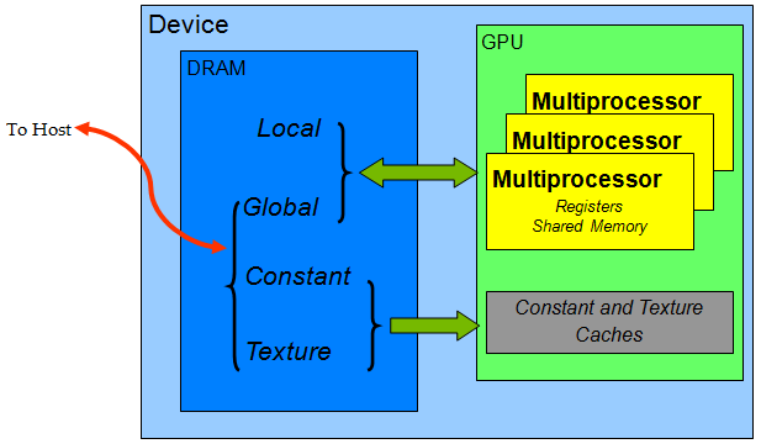


Figure 4.2: An illustration of the memory spaces on a CUDA device [2]

Type	Location	Access	Scope	Lifetime
Register	On chip	R/W	1 thread	thread
Local	Off chip	R/W	1 thread	thread
Shared	On chip	R/W	all threads in block	block
Global	Off chip	R/W	all threads + host	host allocation
Constant	Off chip	R	all threads + host	host allocation
Texture	Off chip	R	all threads + host	host allocation

Table 4.1: Memory types in CUDA architecture

The access time for registers is 1 cycle and for shared memory it is 2-4 cycles, so it quickly becomes expensive. When accessing global memory it costs 400-800 cycles with a penalty of 400 in best case. It can also be a good idea to pay attention to the local memory that each thread will allocate whenever registers are full. The local memory is as slow as global memory to access.

5 Reconstructing the CULZSS

As described in section 2.2, the work in this project will be based on the GPU implementation of **CULZSS**. I have unsuccessfully tried to get in touch with the authors of the original papers, and as a result I need to reconstruct their work in order to use it as a baseline for measuring any improvement proposed in this report.

In this chapter I will describe the different segments of CULZSS, that also will serve as a part of the implementation in later improved versions.

5.1 Pre-Processing

If the workload is to be evenly distributed to a series of CUDA-blocks, each with a fixed number of executing threads, then the division needs to be a multiple of the buffersizes defined beforehand. Otherwise the last CUDA-block to get a chunk of input assigned might get less than the algorithm expects. And with the GPU-kernel needing to be as general as possible, without too many branching, then it is crucial to ensure that input is within some expected bounds.

If the following does not hold for a predefined size of the buffers, or simply a set value per chunk, that the sequential algorithm works fast on, then the input must be enlarged - preferably with some value, that can be filtered later.

$$size(input) \equiv 0 \pmod{size(buffer)}$$

The length of the enlargement, or padding, is simply:

$$padding = (size(buffer) - (size(input) \bmod size(buffer)))$$

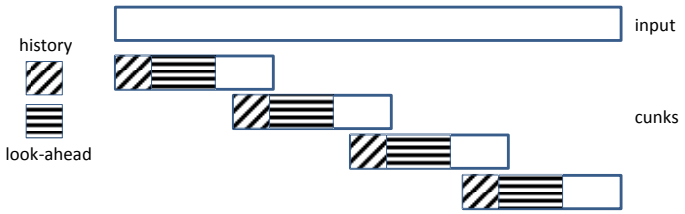


Figure 5.1: How the buffer is divided into several chunks, where each chunk is processed independently of each other. The history buffers overlap in order to not compromise on the compression ratio. The non-allocated part of the chunks will be covered as the "sliding" of the buffers approach the end of the chunks

The number of needed CUDA-blocks is simply assigned the division:

$$blocks = (size(input) + padding) \ / \ size(buffer)$$

5.2 Kernel

In the case of CULZSS, the kernel is a directly importation of the sequential code for LZSS by Michael Dipperstein[7], and differs in none of the crucial elements of the code.

What needs to be taken care of, is the access of data, each thread should have. This means, that each CUDA-thread needs to be aware of where in the original input, which at this point resides in the slow global memory, the assigned portion of clear text reside. For improvement of speed, each thread is at first responsible for copying a portion of clear text into a shared buffer, which reside on the block-level of memory hierarchy. This shared buffer is then distributed virtually amongst the threads by using boundary pointers into the buffer, such that each thread work on its own portion of the original input.

5.3 Output Format of GPU Kernel

A second shared buffer for the output of the threads needs to be initialised. As it can be difficult to ensure the size of an output buffer for each thread, that needs to write into it, Ozsoy et al. devices a two-dimensional array as described in figure 5.2.

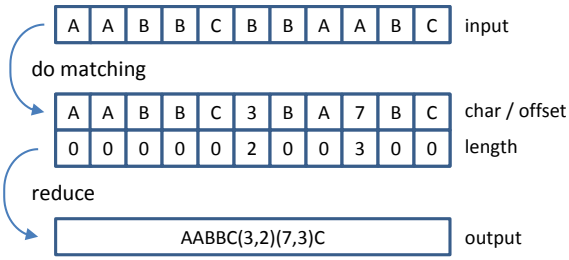


Figure 5.2: An illustration of the two-dimensional output form of the CULZSS, that afterwards needs to be stripped of the redundant data to produce the correct form of encoded information

The CUDA-threads, just as with the shared input buffer, needs boundary pointers to define, which part of the output buffer to write into. Consequently, an entire output buffer may be filled with empty entries, due to the fixed size of the buffer along with the reduced, encoded text of the compression. It is too branching to be efficient for the kernel to handle (and it would require expensive, dynamically allocated arrays), so it has to be dealt with on the CPU.

5.4 Post-Processing

The resulting output from the kernel needs to be pruned to get a result, that can be read from a different LZSS implementation - and more importantly, be written to a stream output.
This can be seen as the reduction step in figure 5.2.

The API proposed by Ozsoy et al. describes a function with a pointer to an output array and the length of the compressed text. As this output is ensured to be less than or equal to the input size (as defined in section 1.2.2), the provided pointer for an output array, could as well be the same as for the input array.

5.5 Final Note

As a final note of this chapter, the original version of CULZSS from [22] was later found at Indiana University GIT repository [1] and this implementation is used without modifications for all the subsequent testing.

As this original code was retrieved at the very end of the project, none of the original code has been used for the further development. Some implementation details differ from my reconstruction, but the test results are the same.

Because of this, some of the reconstructed CULZSS is used unmodified in the construction of the CANLZSS. All reference to CULZSS beyond this point in the report is to the original CULZSS from Ozsoy et al.

[1] `http://phi.cs.indiana.edu/gitweb/?p=Public/CUDA-Compression.git`

6 Improvement Proposals for CANLZSS

Of the reconstructed CULZSS implementation, primarily the pre-processing remains unchanged, and leaves the kernel and post-processing to be the point of improvements.

6.1 A Different Implementation of LZSS

The CULZSS was originally constructed using the serial implementation of *LZSS* by Michael Dipperstein[7], and Dipperstein has since been further developing this implementation several times - mainly by implementing other datastructures for searching the lookup buffer, as described in section 3.2.

However, by exploring other implementations of LZSS, one explicitly seems as a faster solution. The work of Okumura[21] uses a much simpler use of buffers - just one circular with boundary pointers - and the bit-packing used is more streamlined than that of Dipperstein. In addition to this, the Okumura uses far less instructions, so the complexity is less.
It is difficult to pinpoint the differences, though they exist, with simple algorithm samples, so the two implementations are presented in the appendix A.

In order to compare the two algorithms, I let them work on the same dataset, using the same parameters for buffer size and minimum length word before reference is output.
The results in figures 6.1 and 6.2 shows, that the Okumura implementation is overall superior in both *speed* and *compression ratio*.

	Filesize in kilobytes		
	Original	Dipperstein	Okumura
bib	108	52	45
book1	750	414	375
book2	596	280	254
geo	100	82	76
news	368	191	165
obj1	21	12	11
obj2	241	101	93
paper1	52	24	22
paper2	80	39	36
pic	501	103	96
progc	39	18	16
progl	70	23	21
progp	48	16	14
trans	91	34	30

Table 6.1: Using the Calgary Compression Corpus[12] to compare the two different sequential and single-threaded implementations of LZSS. The shown filesizes besides *Original* are post compression

Dipperstein	10.089
Okumura	4.893

Table 6.2: Total elapsed time for each algorithm to process the entire Calgary Compression Corpus. The time includes reading from the harddrive and writing the compressed file onto the harddrive. The implementations are compiled with the same GNU C Compiler without compiler flags

Based on these results, the Okumura implementation will be used as basis for the GPU-kernel in rest of improvement proposals in this project.

6.1.1 Handling the Buffers

At this point, the difference between the CULZSS and the base implementation pose some concerns to attend. The CULZSS uses a shared output buffer, that each thread writes results into. When the matching is finished, this shared buffer is copied to the global output buffer.
This is not a particular efficient approach, as (*shared*) block-level memory offers slower access than the local thread-level memory. Instead each thread has a local output buffer, that will be copied directly to the global output, when processing is finished. However, each thread does not have nearly as much memory as the block-level, so this thread-level buffer only needs to be of the size according to the workload assigned to each thread.

6.1.2 Dynamic Block/Thread Assignment

If the number of CUDA-blocks has to be able to be dynamically assigned in accordance to the size of the input text, then it must be considered how many blocks is assignable on the GPU, as this has an upper limit. The solution is to enlargen the chunk of input each buffer should manage, but even though the memory bounds are considerably larger for the block-level compared to thread-level, then this is also not limitless.
A true buffering scheme is added, exactly as it is done for ordinarily implementations, that reads from streams or filesystems. This implementation just uses global memory as the filesystem.

6.1.3 Handling the Padded Length

To save CPU cycles in the post-processing, looking for matches outputted due to the padded length of the input text, and removing these from the final output, a special **char** (*NULL*) is used in the padded part of the input. When this character is encountered in succession three times, the executing thread stops the matching process and waits for the rest of the block.
It does not give a speedup due to the warp execution, but prevent the algorithm to output extra matches.

6.2 Bit-Packing

The bit-packing used in Okumura uses two 8-bit variables: a `bit_buffer` and `bit_mask`. These variables are used in the process to hold single bits until the `bit_buffer` is filled and then output (either directly into a file or output buffer). The mask is initialised with `0x80` and thus holds a 1 as a *most significant pointer*. This pointer is marking into where the next bit is to be written on the `bit_buffer` and shifted for each bit write. When the mask is 0 (zero), the buffer is full and outputted.

Operating on only 1 bit at a time seems, however, inefficient when one knows the exact number of bytes to be written: a single char when no reference is found, and 2 bytes per reference pointer.

Instead I propose a faster bit-packing more suitable during post-processing (it is not efficient to let the GPU handle the bit-packing, as the output is in two dimensions, and still needs to be parsed individually by the CPU).

The `bit_buffer` and `bit_mask` is kept, as single bits still needs to be output as the prefix bit. In the example below, the `bit_buffer` already holds a single bit, as denoted by the `bit_mask` shifted one to the right. Thus only the first 7 bits of the character to write need to be output, denoted by 6.2.

bit_mask	0100 0000	(6.1)
(bit_mask « 1)-1	0111 1111	(6.2)
negated	1000 0000	(6.3)
character to write	1011 0010	(6.4)
right rotated	0101 1001	(6.5)

The («) denotes a left bit-shift. A single rotation of the character in 6.5 along with the mask from 6.2, the first 7 bits can be added to `bit_buffer`, and immediately output. With the negated mask from 6.3, the last bit can be put into the `bit_buffer`, and `bit_mask` remains unaltered after this entire operation.

By using this method, the amount of CPU operations for bit-packing is reduced by 60% (removing unnecessary checks and bit-operations).

6.3 Increase of Parallelism

In their paper from 2012[23] Ozsoy et al. proposed an optimised matching scheme to be used on the highly parallel GPU execution.

Algorithm 1 Matching of the LZSS

```
while i < buffer_end do
    if buffer[history_start + j] = buffer[i + j] then
        while MATCH do
            j = j + 1
        end while
        update matching information
    end if
    j = 0
end while
store matching information
history_start = history_start + match_length, i = i + match_length
```

Using their proposed optimisation, the matching will instead take place in *matching states*, that embraces the warp of execution as described in section 4.1.

Algorithm 2 Proposed optimisation of matching

```
while i < buffer_end do
    if buffer[history_start + j] = buffer[i + j] then
        j = j + 1 //matching state
    else
        update matching information
        j = 0 //exit matching state
    end if
end while
store matching information
history_start = history_start + match_length, i = i + match_length
```

The inner-loop is eliminated, which greatly reduces control-flow divergence. This also improves reading from the shared input buffer, as all threads consumes the same memory bank block, which then only uses a single memory call.

6.4 KMP Optimisation

In section 3.1 it is described how the sequential search is executed, and from this it seems somewhat intuitive to incorporate the same mechanics as **Knuth-Morris-Pratt**[16] published in 1977 when improving simple string matching.

Disregarding the precomputed lookup table, the main contribution was the observation, that the length of the current match needs not to be tested again, and can therefore be skipped before the next test of matching.

It is clear, that implementing this will theoretically increase the compression ratio (remember: less is better), as the following scenario could occur:

$$Text : BAAAABCDCDCD$$
$$Word : AAAB$$

This search would find a match starting from the first A, but stop the matching state, when the last A is met, as this ought to be a B, and thus the word would not be found in the text.

Never the less, tests on sequential code shows an improvement in execution time of $\approx 20\%$ when compressing the Calgary Corpus, however, with a slight increase in storage requirement of $\approx 1\%$.

It would be optimal, if the search did not include matching of words less than the longest match, but as the matching is executed on single characters and not entire words, this is not feasible with the current approach.

6.5 Implementation Details

Using an API similar to the proposed in CULZSS[22]:

```
1       Gpu_compress(*buffer, buf_lengt,
             **compressed_buffer, &comp_length,
             compression_parameters)
```

the implementation can be used as in-memory compression in applications that perform compression on-the-fly, like webservers, or other I/O intensive works, and ease whatever other tasks the CPU might need to handle.

The application can also be used as stand-alone, accepting files as input and writing the compressed file back as a output file.

7 Experiments and Results

7.1 Testbed Configurations

All testing have been done on a simple laptop with a general purpose GPU, and not a scientific GPU, which is optimised for extreme parallellisation, whereas GPGPUs consider tradeoffs between bus transfers and number of available cores[1] - and of course pricing.

The test machine has an nVidia GeForce GT 620M with 1GB dedicated RAM and CUDA version 2.1 along with Intel(R) Core(TM) i7 CPU 1.9GHz.

7.2 Datasets

As test data, the Calgary Compression Corpus[2] is used. The Calgary is a collection of different files - text inputs as well as non-text inputs - used as an agreed upon baseline for new implementations of compression algorithms. By using the same corpus as other researchers, the results are directly comparable.

When the file sizes of the Calgary are not sufficiently large, a Large Corpus[3] can be used to supplement. This corpus includes a version of the Bible, the complete genom of E. Coli and the CIA world fact book.

[1] `http://www.nvidia.com/object/gpu_science.html`
[2] `http://corpus.canterbury.ac.nz/descriptions/#calgary`
[3] `http://corpus.canterbury.ac.nz/descriptions/#large`

	Time in milliseconds			
	CAN_{first}	CAN_{bp}	CAN_{par}	CAN_{kmp}
bible	303	731	280	309
E.coli	266	262	273	271
world192	210	206	194	206

Table 7.1: Test results on *speed* of running the Large Corpus on each of the proposals. The timing is including reading and writing from/to disk

	Compression ratio			
	CAN_{first}	CAN_{bp}	CAN_{par}	CAN_{kmp}
bible	4.37%	4.25%	4.57%	4.34%
E.coli	4.18%	4.07%	3.99%	4.06%
world192	4.41%	4.31%	4.75%	4.31%

Table 7.2: Test results on *compression ratio* of running the Large Corpus on each of the proposals

7.3 Experimental Results of Proposals

Until now, each proposal has not been explicitly referred, but in the following naming will be be needed.

- CAN_{first} The base implementation of the Okumura-LZSS including the subsections of section 6.1.
- CAN_{bp} Improved bit-packing described in section 6.2
- CAN_{par} The improved parallelism from section 6.3
- CAN_{kmp} Usage of KMP-method, section 6.4

The tables 7.1 and 7.2 show the test results from each of the proposals. The results of CAN_{first} should be regarded as a baseline in the assessment of best proposals, as all the subsequent proposals build on top of this. All the results are based on the same parameters such as number of executing threads per block, number of blocks, and blocksize.

7.3.1 Part Conclusion on the Results

The purpose of this chapter is to find an ideal combined solution compiled of the best proposals into a single, final CANLZSS algorithm. When using the CAN_{first} as baseline, it seems clear, that CAN_{bp} gives all-round better results, so it can easily be incorporated in the final solution. Unfortunately, the last two, CAN_{par} and CAN_{kmp}, are mutually exclusive. They both try to improve the matching process. Yet it seems that CAN_{kmp} performs better in both speed an compression compared to the CAN_{first}, whereas CAN_{par} on average only performs better in execution time.

Further tests between CAN_{par} and CAN_{kmp} when increasing the number of threads per block only reveals a larger gap between the two, as CAN_{par} continues to perform better on the speed, but get worse compression ratio. CAN_{kmp}, on the other hand, gets speed and holds the compression ratio.

The ideal solutions seems to be a combination of CAN_{first}, CAN_{bp} and CAN_{kmp} into one final CANLZSS. This is the

8 Parameter tuning

Finding the optimal parameters for the execution is a key task in optimising performance. Especially with GPU applications, which are near impossible to analyse theoretically, so a series of automated tests to serve as empirical evidence can be devised.
The key parameters to test in this application is:

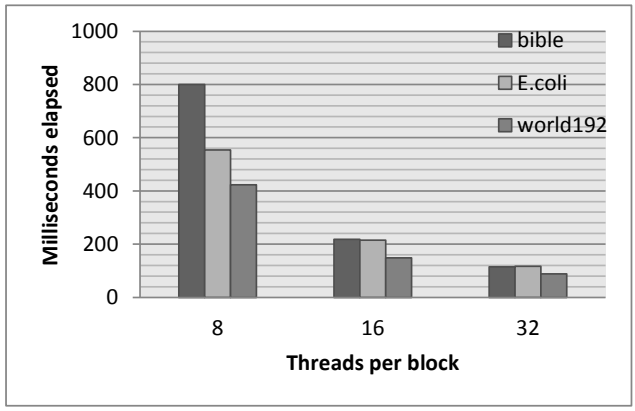
- Number of threads per block - needs to be a multiple of 2.
- The size of buffers used, which in turn has an impact on the number of CUDA-blocks allocated.

Tests are conducted in accordance to guidelines described in [25].

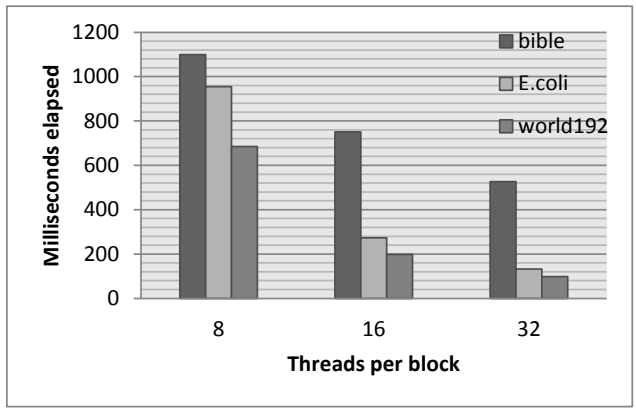
The results of the tests should reveal the most optimal settings of parameters in the CANLZSS, which then will be used in the final evaluation of performance. Success is based on lowest execution time.

Results are not feasible, when using only 4 threads and lower or using 64 and beyond, so these data are emitted from the graph. An upper and lower bound also showed when changing the buffer sizes. So the presented results in figure 8.1 are the feasible solutions.

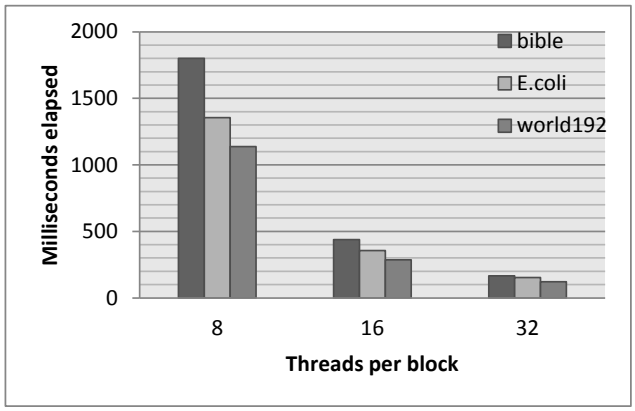
From the results it is somewhat evident, that the best result comes from a buffersize of 2048 bytes with 32 threads per block. So these are the parameters, that will be used in the final comparison.



(a) Buffer size of 2048 bytes



(b) Buffer size of 4096 bytes



(c) Buffer size of 8196 bytes

Figure 8.1: The result of the parameter tuning when adjusting the threads per block and buffer size. Due to the unfeasible solutions, number of threads below 8 and above 32 are omitted.

9 Performance Evaluation

9.1 Potential of Using GPUs for Compression

In this chapter I evaluate the potential of using graphical processors in compression schemes. The results found of the CANLZSS is compared to the sequential LZSS CPU algorithm and the CULZSS GPU implementation, which this project use as baseline. As the Okumura LZSS[21] implementation proved faster than the Dipperstein LZSS[7], the former is used as basis for this comparison. No comparison where made with a parallelised CPU implementation, as the CULZSS already have showed results of outperforming such algorithms, and consequently can act on behalf of this by induction.

Table 9.1 shows the performance comparison and outlines the speed-up over both the CPU and GPU implementations. This overview, however simple, demonstrate the potential of using GPU hardware in compression.

9.2 Comparing to Other GPU Solutions

It is difficult to compare with results of parallelised CPU and GPU versions of other compression algorithms, as some are designed to be fast in compression, but uses more decompression time. For true comparison, both compression and decompression times should be valuated along with the compression ratio.

	Running time in milliseconds			
	CPU LZSS	GPU CULZSS	GPU CANLZSS	CPU/GPU Speed-up
bible	3299	991	543	6X/2X
E.coli	7406	778	118	63X/7X
world192	2248	324	92	24X/6X

Table 9.1: Performance comparison of the sequential CPU implementation of LZSS, the GPU implementation of CULZSS and the newly devised CANLZSS. The testbed configurations from section 7.1 still holds for these results. The timing include both reading and writing of files from the filesystem. The test have been conducted with the same parameters as found in chapter 8. The last column shows the speed-up achieved compared to the LZSS and the CULZSS

10 Conclusion

This project set out to examine the feasibility for using the CUDA framework to improve upon the speed of lossless data compression without neglecting the compression ratio. The focus of the project was to achieve substantial speed-up compared to a CPU implementation and the CULZSS GPU algorithm.

I have proposed a set of possible enhancements to the CULZSS, and evaluated the performance of each proposal in order to construct a final solution, that meets and succeeds the outlined criteria.

Tests have shown that the devices new algorithm called CANLZSS outperforms the serial LZSS implementation, of which it was based, by up to 63 times. The implementation also performs better than the baseline of CULZSS by up to 7 times.

The implementation offers an API for programmers to use in their applications, but can also be used as a stand-alone application for handling files as input and output the compressed data. With the offered API, and due to its compatibility with simpler CPU implementations, the application could serve well as a webserver implementation for swifter transfer of files to clients.

10.1 Further Work

An update of the post-processing for compatibility with more commonly used decompression algorithms, such as BZIP2 and GZIP used in modern browsers and standards on operating systems.

As stated in section 4.1, the local memory of the threads is extremely slow, so some work should be put in to further evaluate if the memory consumption of each thread could be optimised.

By rewriting the kernel into using OpenCL devices for it to be supported by various more graphic processors along with the possibility for collaborating with CPUs without any further modifications.

If the CANLZSS is to be used on webservers with multiple simultaneous connections to be handled, a pipeline processing scheme could be used for better utilisation of the many compute cores of the GPU.

Consider the problematics from section 6.4, make the matching possible on words instead of just single characters. Preferably of variable lengths to accommodate the need to search for bigger words per each match.
Some research should go in to test algorithms for fingerprinting of limited lengths such as [4, 15]

Bibliography

[1] Mark Adler. Pigz - a parallel implementation of gzip for modern multiprocessor, multi-core machines. `http://zlib.net/pigz/`. Last checked 11-04-2013.

[2] Ana Balevic. Parallel variable-length encoding on gpgpus. In *Euro-Par 2009–Parallel Processing Workshops*, pages 26–35. Springer, 2010.

[3] Ana Balevic, Lars Rockstroh, Marek Wroblewski, and Sven Simon. Using arithmetic coding for reduction of resulting simulation data size on massively parallel gpgpus. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 295–302. Springer, 2008.

[4] Philip Bille, Inge Li Gørtz, and Jesper Kristensen. Longest common extensions via fingerprinting. In *Language and Automata Theory and Applications*, pages 119–130. Springer, 2012.

[5] Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. Technical report 24, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, May 1994.

[6] Martin Campbell-Kelly. Not all bad: An historical perspective on software patents. *11 Mich. Telecomm. Tech. L. Rev. 191*, 2005. `http://www.mttlr.org/voleleven/campbell-kelly.pdf`.

[7] Michael Dipperstein. Lzss (lz77) discussion and implementation. `http://michael.dipperstein.com/lzss/index.html`. Last checked 19-06-2013.

[8] Axel Eirola. Lossless data compression on gpgpu architectures. *arXiv preprint arXiv:1109.2348*, 2011.

[9] Peter D Geiger, Manuel J Alvarez, Thomas A Dye, et al. Parallel compression and decompression system and method having multiple parallel compression and decompression engines, November 16 2004. US Patent 6,819,271.

[10] Jeff Gilchrist. Parallel data compression with bzip2. In *Proceedings of the 16th IASTED International Conference on Parallel and Distributed Computing and Systems*, volume 16, pages 559–564, 2004.

[11] D.A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[12] Tim Bell Ian Witten and John Cleary. The calgary compression corpus. `http://corpus.canterbury.ac.nz/descriptions/#calgary`. Last checked 20-06-2013.

[13] J. Kane and Qing Yang. Compression speed enhancements to lzo for multi-core systems. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pages 108–115, 2012.

[14] Shmuel Tomi Klein and Yair Wiseman. Parallel lempel ziv coding. *Discrete Appl. Math.*, 146(2):180–191, March 2005. `http://dx.doi.org/10.1016/j.dam.2004.04.013`.

[15] John Kloetzli, Brian Strege, Jonathan Decker, and Marc Olano. Parallel longest common subsequence using graphics hardware. In *Proceedings of the 8th Eurographics conference on Parallel Graphics and Visualization*, pages 57–64. Eurographics Association, 2008.

[16] Donald E Knuth, James H Morris, Jr, and Vaughan R Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6(2):323–350, 1977.

[17] A. Lempel and J. Ziv. A universal algorithm for sequential data compression. *Information Theory, IEEE Transactions on*, 23(3):337 – 343, may 1977.

[18] Jean loup Gailly and Mark Adler. The gzip home page. `http://www.gzip.org/`. Last checked 11-04-2013.

[19] Jean loup Gailly and Mark Adler. Zlib compression library. `http://www.dspace.cam.ac.uk/handle/1810/3486`, 2004. Last checked 11-04-2013.

[20] M.F.X.J. Oberhumer. Lzo real-time data compression library. `http://www.oberhumer.com/opensource/lzo/`, 2011. Last checked 19-06-2013.

[21] Haruhiko Okumura. Lzss encoder-decoder. `http://oku.edu.mie-u.ac.jp/~okumura/compression/lzss.c`. Last checked 19-06-2013.

[22] Adnan Ozsoy and Martin Swany. Culzss: Lzss lossless data compression on cuda. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 403–411. IEEE, 2011.

[23] Adnan Ozsoy, Martin Swany, and Arun Chauhan. Pipelined parallel lzss for streaming data compression on gpgpus. In *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*, pages 37–44. IEEE, 2012.

[24] Julian Seward. The bzip2 and libbzip2 official home page. `http://bzip.org/`. Last checked 11-04-2013.

[25] Hans Henrik Brandenborg Sørensen. Auto-tuning of level 1 and level 2 blas for gpus. *Concurrency and Computation: Practice and Experience*, 2012.

[26] James A Storer and Thomas G Szymanski. Data compression via textual substitution. *Journal of the ACM (JACM)*, 29(4):928–951, 1982.

[27] T.A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, 1984.

[28] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *Information Theory, IEEE Transactions on*, 24(5):530–536, 1978.

A Different LZSS implementations

A.1 Okumura

```
/* LZSS encoder-decoder  (c) Haruhiko Okumura */

/* http://interblag.com/lzss-compression-algorithm.html */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define EI 11  /* typically 10..13 */
#define EJ  4  /* typically 4..5 */
#define P   1  /* If match length <= P then output one character */
#define N (1 << EI)  /* buffer size */
#define F ((1 << EJ) + P)  /* lookahead buffer size */

int bit_buffer = 0, bit_mask = 128;
unsigned long codecount = 0, textcount = 0;
unsigned char buffer[N * 2];
FILE *infile, *outfile;

void error(void)
{
    printf("Output error\n");  exit(1);
}

void putbit1(void)
{
    bit_buffer |= bit_mask;
    if ((bit_mask >>= 1) == 0) {
        if (fputc(bit_buffer, outfile) == EOF) error();
        bit_buffer = 0;  bit_mask = 128;  codecount++;
```

```
    }
}

void putbit0(void)
{
    if ((bit_mask >>= 1) == 0) {
        if (fputc(bit_buffer, outfile) == EOF) error();
        bit_buffer = 0;  bit_mask = 128;  codecount++;
    }
}

void flush_bit_buffer(void)
{
    if (bit_mask != 128) {
        if (fputc(bit_buffer, outfile) == EOF) error();
        codecount++;
    }
}

void output1(int c)
{
    int mask;
    putbit1();
    mask = 256;
    while (mask >>= 1) {
        if (c & mask) putbit1();
        else putbit0();
    }
}

void output2(int x, int y)
{
    int mask;
    putbit0();
    mask = N;
    while (mask >>= 1) {
        if (x & mask) putbit1();
        else putbit0();
    }
    mask = (1 << EJ);
    while (mask >>= 1) {
        if (y & mask) putbit1();
        else putbit0();
    }
}

void encode(void)
{
    int i, j, f1, x, y, r, s, bufferend, c;

    //fill start of the buffer with a known state
    for (i = 0; i < N - F; i++) buffer[i] = ' ';
    //read in characters into the lookahead buffer
    for (i = N - F; i < N * 2; i++) {
        if ((c = fgetc(infile)) == EOF) break;
```

```
        buffer[i] = c;  textcount++;
    }
    bufferend = i;  r = N - F;  s = 0;//s = start of history
    buffer, r = end of the history buffer
    while (r < bufferend) {
        f1 = (F <= bufferend - r) ? F : bufferend - r;//makes sure
    we do not do bufferoverflow - might have used min()
        x = 0;  y = 1;  c = buffer[r];//x = offset, y = length of
    match
        for (i = r - 1; i >= s; i--)
            if (buffer[i] == c) {
                for (j = 1; j < f1; j++)
                    if (buffer[i + j] != buffer[r + j]) break;
                if (j > y) {
                    x = i;  y = j;
                }
            }
        if (y <= P) output1(c);
        else output2(x & (N - 1), y - 2);
        r += y;  s += y;
        //there is no more room left in the lookahead buffer
        if (r >= N * 2 - F) {
         //shift the buffer N backward
            for (i = 0; i < N; i++) buffer[i] = buffer[i + N];
            bufferend -= N;  r -= N;  s -= N;
            while (bufferend < N * 2) {
                if ((c = fgetc(infile)) == EOF) break;
                buffer[bufferend++] = c;  textcount++;
            }
        }
    }
    flush_bit_buffer();
    printf("text:  %ld bytes\n", textcount);
    printf("code:  %ld bytes (%ld%%)\n",
        codecount, (codecount * 100) / textcount);
}

int getbit(int n) /* get n bits */
{
    int i, x;
    static int buf, mask = 0;

    x = 0;
    for (i = 0; i < n; i++) {
        if (mask == 0) {
            if ((buf = fgetc(infile)) == EOF) return EOF;
            mask = 128;
        }
        x <<= 1;
        if (buf & mask) x++;
        mask >>= 1;
    }
    return x;
}
```

```
void decode(void)
{
    int i, j, k, r, c;

    for (i = 0; i < N - F; i++) buffer[i] = ' ';
    r = N - F;
    while ((c = getbit(1)) != EOF) {
        if (c) {
            if ((c = getbit(8)) == EOF) break;
            fputc(c, outfile);
            buffer[r++] = c;  r &= (N - 1);//(N - 1)
        } else {
            if ((i = getbit(EI)) == EOF) break;
            if ((j = getbit(EJ)) == EOF) break;
            for (k = 0; k <= j + 1; k++) {
                c = buffer[(i + k) & (N - 1)];
                fputc(c, outfile);
                buffer[r++] = c;  r &= (N - 1);
            }
        }
    }
}

int main(int argc, char *argv[])
{
    int enc;
    char *s;
    clock_t time = clock();

    if (argc != 4) {
        printf("Usage: lzss e/d infile outfile\n\te = encode\td =
    decode\n");
        return 1;
    }
    s = argv[1];
    if (s[1] == 0 && (*s == 'd' || *s == 'D' || *s == 'e' || *s ==
    'E'))
        enc = (*s == 'e' || *s == 'E');
    else {
        printf("? %s\n", s);  return 1;
    }
    if ((infile  = fopen(argv[2], "rb")) == NULL) {
        printf("? %s\n", argv[2]);  return 1;
    }
    if ((outfile = fopen(argv[3], "wb")) == NULL) {
        printf("? %s\n", argv[3]);  return 1;
    }
    if (enc) encode();  else decode();
    fclose(infile);  fclose(outfile);

    printf("time: %.2f \n", (double)(clock() -
    time)/CLOCKS_PER_SEC);
    return 0;
}
```

A.2 Dipperstein

```
/***************************************************************************
*          Lempel, Ziv, Storer, and Szymanski Encoding and Decoding
*
*   File    : lzss.c
*   Purpose : Use lzss coding (Storer and Szymanski's modified
    lz77) to
*             compress/decompress files.
*   Author  : Michael Dipperstein
*   Date    : November 24, 2003
*
***************************************************************************
*   UPDATES
*
*   Date        Change
*   12/10/03    Changed handling of sliding window to better match
    standard
*               algorithm description.
*   12/11/03    Remebered to copy encoded characters to the
    sliding window
*               even when there are no more characters in the
    input stream.
*
***************************************************************************
*
* LZSS: An ANSI C LZss Encoding/Decoding Routine
* Copyright (C) 2003 by Michael Dipperstein (mdipper@cs.ucsb.edu)
*
* This library is free software; you can redistribute it and/or
* modify it under the terms of the GNU Lesser General Public
* License as published by the Free Software Foundation; either
* version 2.1 of the License, or (at your option) any later
    version.
*
* This library is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
* Lesser General Public License for more details.
*
* You should have received a copy of the GNU Lesser General Public
* License along with this library; if not, write to the Free
    Software
* Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
    02111-1307  USA
*
***************************************************************************/

/***************************************************************************
*                              INCLUDED FILES
***************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include "getopt.h"
```

```
/*****************************************************************
*                            TYPE DEFINITIONS
*****************************************************************/
/* unpacked encoded offset and length, gets packed into 12 bits
    and 4 bits*/
typedef struct encoded_string_t
{
    int offset;     /* offset to start of longest match */
    int length;     /* length of longest match */
} encoded_string_t;

typedef enum
{
    ENCODE,
    DECODE
} MODES;

/*****************************************************************
*                                CONSTANTS
*****************************************************************/
#define FALSE   0
#define TRUE    1

#define WINDOW_SIZE     4096   /* size of sliding window (12 bits)
    */

/* maximum match length not encoded and encoded (4 bits) */
#define MAX_UNCODED     2
#define MAX_CODED       (15 + MAX_UNCODED + 1)

/*****************************************************************
*                            GLOBAL VARIABLES
*****************************************************************/
/* cyclic buffer sliding window of already read characters */
unsigned char slidingWindow[WINDOW_SIZE];
unsigned char uncodedLookahead[MAX_CODED];

/*****************************************************************
*                             PROTOTYPES
*****************************************************************/
void EncodeLZSS(FILE *inFile, FILE *outFile);   /* encoding
    routine */
void DecodeLZSS(FILE *inFile, FILE *outFile);   /* decoding
    routine */

/*****************************************************************
*                              FUNCTIONS
*****************************************************************/

/*****************************************************************
*   Function    : main
*   Description: This is the main function for this program, it
    validates
*                the command line input and, if valid, it will
```

```
    either
*                encode a file using the LZss algorithm or decode a
97 *                file encoded with the LZss algorithm.
*   Parameters : argc - number of parameters
*                argv - parameter list
*   Effects    : Encodes/Decodes input file
*   Returned   : EXIT_SUCCESS for success, otherwise EXIT_FAILURE.
102 ****************************************************************************/
int main(int argc, char *argv[])
{
    int opt;
    FILE *inFile, *outFile;  /* input & output files */
107     MODES mode;

    /* initialize data */
    inFile = NULL;
    outFile = NULL;
112     mode = ENCODE;

    /* parse command line */
    while ((opt = getopt(argc, argv, "cdtni:o:h?")) != -1)
    {
117         switch(opt)
        {
            case 'c':       /* compression mode */
                mode = ENCODE;
                break;
122
            case 'd':       /* decompression mode */
                mode = DECODE;
                break;

127             case 'i':       /* input file name */
                if (inFile != NULL)
                {
                    fprintf(stderr, "Multiple input files not
    allowed.\n");
                    fclose(inFile);
132
                    if (outFile != NULL)
                    {
                        fclose(outFile);
                    }
137
                    exit(EXIT_FAILURE);
                }
                else if ((inFile = fopen(optarg, "rb")) == NULL)
                {
142                     perror("Opening inFile");

                    if (outFile != NULL)
                    {
                        fclose(outFile);
147                     }
```

```
                    exit(EXIT_FAILURE);
                }
                break;

            case 'o':       /* output file name */
                if (outFile != NULL)
                {
                    fprintf(stderr, "Multiple output files not
    allowed.\n");
                    fclose(outFile);

                    if (inFile != NULL)
                    {
                        fclose(inFile);
                    }

                    exit(EXIT_FAILURE);
                }
                else if ((outFile = fopen(optarg, "wb")) == NULL)
                {
                    perror("Opening outFile");

                    if (outFile != NULL)
                    {
                        fclose(inFile);
                    }

                    exit(EXIT_FAILURE);
                }
                break;

            case 'h':
            case '?':
                printf("Usage: lzss <options>\n\n");
                printf("options:\n");
                printf("  -c : Encode input file to output
    file.\n");
                printf("  -d : Decode input file to output
    file.\n");
                printf("  -i <filename> : Name of input file.\n");
                printf("  -o <filename> : Name of output file.\n");
                printf("  -h | ?  : Print out command line
    options.\n\n");
                printf("Default: lzss -c\n");
                return(EXIT_SUCCESS);
        }
    }

    /* validate command line */
    if (inFile == NULL)
    {
        fprintf(stderr, "Input file must be provided\n");
        fprintf(stderr, "Enter \"lzss -?\" for help.\n");

        if (outFile != NULL)
```

```
            {
                fclose(outFile);
            }

            exit (EXIT_FAILURE);
        }
        else if (outFile == NULL)
        {
            fprintf(stderr, "Output file must be provided\n");
            fprintf(stderr, "Enter \"lzss -?\" for help.\n");

            if (inFile != NULL)
            {
                fclose(inFile);
            }

            exit (EXIT_FAILURE);
        }

        /* we have valid parameters encode or decode */
        if (mode == ENCODE)
        {
            EncodeLZSS(inFile, outFile);
        }
        else
        {
            DecodeLZSS(inFile, outFile);
        }

        fclose(inFile);
        fclose(outFile);
        return EXIT_SUCCESS;
}

/***************************************************************************
*    Function    : FindMatch
*    Description: This function will search through the
    slidingWindow
*                 dictionary for the longest sequence matching the
    MAX_CODED
*                 long string stored in uncodedLookahed.
*    Parameters : windowHead - head of sliding window
*                 uncodedHead - head of uncoded lookahead buffer
*    Effects    : NONE
*    Returned   : The sliding window index where the match starts
    and the
*                 length of the match.  If there is no match a
    length of
*                 zero will be returned.
***************************************************************************/
encoded_string_t FindMatch(int windowHead, int uncodedHead)
{
    encoded_string_t matchData;
    int i, j;
```

```
    matchData.length = 0;
    i = windowHead;  /* start at the beginning of the sliding
    window */
    j = 0;

    while (TRUE)
    {
        if (slidingWindow[i] == uncodedLookahead[uncodedHead])
        {
            /* we matched one how many more match? */
            j = 1;

            while(slidingWindow[(i + j) % WINDOW_SIZE] ==
                uncodedLookahead[(uncodedHead + j) % MAX_CODED])
            {
                if (j >= MAX_CODED)
                {
                    break;
                }
                j++;
            };

            if (j > matchData.length)
            {
                matchData.length = j;
                matchData.offset = i;
            }
        }

        if (j >= MAX_CODED)
        {
            matchData.length = MAX_CODED;
            break;
        }

        i = (i + 1) % WINDOW_SIZE;
        if (i == windowHead)
        {
            /* we wrapped around */
            break;
        }
    }

    return matchData;
}

/**************************************************************************
*   Function    : EncodeLZSS
*   Description: This function will read an input file and write
    an output
*                file encoded using a slight modification to the
    LZss
*                algorithm.  I'm not sure who to credit with the
    slight
*                modification to LZss, but the modification is to
```

```
    group the
302 *                coded/not coded flag into bytes.  By grouping the
    flags,
    *                the need to be able to write anything other than
    a byte
    *                may be avoided as longs as strings encode as a
    whole byte
    *                multiple.  This algorithm encodes strings as 16
    bits (a 12
    *                bit offset + a 4 bit length).
307 *    Parameters  : inFile - file to encode
    *                 outFile - file to write encoded output
    *    Effects     : inFile is encoded and written to outFile
    *    Returned    : NONE
    ********************************************************************/
312 void EncodeLZSS(FILE *inFile, FILE *outFile)
    {
        /* 8 code flags and encoded strings */
        unsigned char flags, flagPos, encodedData[16];
        int nextEncoded;                /* index into encodedData */
317     encoded_string_t matchData;
        int i, c;
        int len;                        /* length of string */
        int windowHead, uncodedHead;    /* head of sliding window and
        lookahead */

322     flags = 0;
        flagPos = 0x01;
        nextEncoded = 0;
        windowHead = 0;
        uncodedHead = 0;
327

        /*************************************************************
        * Fill the sliding window buffer with some known vales.
        DecodeLZSS must
        * use the same values.  If common characters are used, there's
        an
        * increased chance of matching to the earlier strings.
332
        *************************************************************/
        for (i = 0; i < WINDOW_SIZE; i++)
        {
            slidingWindow[i] = ' ';
        }
337

        /*************************************************************
        * Copy MAX_CODED bytes from the input file into the uncoded
        lookahead
        * buffer.

        *************************************************************/
342     for (len = 0; len < MAX_CODED && (c = getc(inFile)) != EOF;
        len++)
```

```
    {
        uncodedLookahead[len] = c;
    }

    if (len == 0)
    {
        return;  /* inFile was empty */
    }

    /* Look for matching string in sliding window */
    matchData = FindMatch(windowHead, uncodedHead);

    /* now encoded the rest of the file until an EOF is read */
    while (len > 0)
    {
        if (matchData.length > len)
        {
            /* garbage beyond last data happened to extend match
length */
            matchData.length = len;
        }

        if (matchData.length <= MAX_UNCODED)
        {
            /* not long enough match.  write uncoded byte */
            matchData.length = 1;   /* set to 1 for 1 byte uncoded
*/
            flags |= flagPos;       /* mark with uncoded byte flag
*/
            encodedData[nextEncoded++] =
uncodedLookahead[uncodedHead];
        }
        else
        {
            /* match length > MAX_UNCODED.  Encode as offset and
length. */
            encodedData[nextEncoded++] =
                (unsigned char)((matchData.offset & 0x0FFF) >> 4);

            encodedData[nextEncoded++] =
                (unsigned char)(((matchData.offset & 0x000F) << 4)
|
                (matchData.length - (MAX_UNCODED + 1)));
        }

        if (flagPos == 0x80)
        {
            /* we have 8 code flags, write out flags and code
buffer */
            putc(flags, outFile);

            for (i = 0; i < nextEncoded; i++)
            {
                /* send at most 8 units of code together */
                putc(encodedData[i], outFile);
```

```
            }

            /* reset encoded data buffer */
            flags = 0;
            flagPos = 0x01;
            nextEncoded = 0;
        }
        else
        {
            /* we don't have 8 code flags yet, use next bit for
    next flag */
            flagPos <<= 1;
        }


    /**************************************************************
        * Replace the matchData.length worth of bytes we've
    matched in the
        * sliding window with new bytes from the input file.

    **************************************************************/
        i = 0;
        while ((i < matchData.length) && ((c = getc(inFile)) !=
    EOF))
        {
            /* add old byte into sliding window and new into
    lookahead */
            slidingWindow[windowHead] =
    uncodedLookahead[uncodedHead];
            uncodedLookahead[uncodedHead] = c;
            windowHead = (windowHead + 1) % WINDOW_SIZE;
            uncodedHead = (uncodedHead + 1) % MAX_CODED;
            i++;
        }

        /* handle case where we hit EOF before filling lookahead */
        while (i < matchData.length)
        {
            slidingWindow[windowHead] =
    uncodedLookahead[uncodedHead];
            /* nothing to add to lookahead here */
            windowHead = (windowHead + 1) % WINDOW_SIZE;
            uncodedHead = (uncodedHead + 1) % MAX_CODED;
            len--;
            i++;
        }

        /* find match for the remaining characters */
        matchData = FindMatch(windowHead, uncodedHead);
    }

    /* write out any remaining encoded data */
    if (nextEncoded != 0)
    {
        putc(flags, outFile);
```

```
        for (i = 0; i < nextEncoded; i++)
        {
            putc(encodedData[i], outFile);
        }
    }
}

/**************************************************************************
*   Function    : DecodeLZSS
*   Description: This function will read an LZss encoded input
    file and
*                write an output file.  The encoded file uses a
    slight
*                modification to the LZss algorithm.  I'm not sure
    who to
*                credit with the slight modification to LZss, but
    the
*                modification is to group the coded/not coded flag
    into
*                bytes.  By grouping the flags, the need to be
    able to
*                write anything other than a byte may be avoided
    as longs
*                as strings encode as a whole byte multiple.  This
    algorithm
*                encodes strings as 16 bits (a 12bit offset + a 4
    bit length).
*   Parameters : inFile - file to decode
*                outFile - file to write decoded output
*   Effects    : inFile is decoded and written to outFile
*   Returned   : NONE
**************************************************************************/
void DecodeLZSS(FILE *inFile, FILE *outFile)
{
    int  i, c;
    unsigned char flags, flagsUsed;     /* encoded/not encoded
    flag */
    int nextChar;                       /* next char in sliding
    window */
    encoded_string_t code;              /* offset/length code for
    string */

    /* initialize variables */
    flags = 0;
    flagsUsed = 7;
    nextChar = 0;


    /**************************************************************
    * Fill the sliding window buffer with some known vales.
    EncodeLZSS must
    * use the same values.  If common characters are used, there's
    an
    * increased chance of matching to the earlier strings.
```

```
    ********************************************************************/
    for (i = 0; i < WINDOW_SIZE; i++)
    {
        slidingWindow[i] = ' ';
    }

    while (TRUE)
    {
        flags >>= 1;
        flagsUsed++;

        if (flagsUsed == 8)
        {
            /* shifted out all the flag bits, read a new flag */
            if ((c = getc(inFile)) == EOF)
            {
                break;
            }

            flags = c & 0xFF;
            flagsUsed = 0;
        }

        if (flags & 0x01)
        {
            /* uncoded character */
            if ((c = getc(inFile)) == EOF)
            {
                break;
            }

            /* write out byte and put it in sliding window */
            putc(c, outFile);
            slidingWindow[nextChar] = c;
            nextChar = (nextChar + 1) % WINDOW_SIZE;
        }
        else
        {
            /* offset and length */
            if ((code.offset = getc(inFile)) == EOF)
            {
                break;
            }

            if ((code.length = getc(inFile)) == EOF)
            {
                break;
            }

            /* unpack offset and length */
            code.offset <<= 4;
            code.offset |= ((code.length & 0x00F0) >> 4);
            code.length = (code.length & 0x000F) + MAX_UNCODED + 1;
```

```
532
    /**************************************************************
            * Write out decoded string to file and lookahead.  It
    would be
            * nice to write to the sliding window instead of the
    lookahead,
            * but we could end up overwriting the matching string
    with the
            * new string if abs(offset - next char) < match length.
537
    **************************************************************/
            for (i = 0; i < code.length; i++)
            {
                c = slidingWindow[(code.offset + i) % WINDOW_SIZE];
                putc(c, outFile);
542             uncodedLookahead[i] = c;
            }

            /* write out decoded string to sliding window */
            for (i = 0; i < code.length; i++)
547         {
                slidingWindow[(nextChar + i) % WINDOW_SIZE] =
                    uncodedLookahead[i];
            }

552         nextChar = (nextChar + code.length) % WINDOW_SIZE;
        }
    }
}
```