The Australian National University
College of Engineering and Computer Science

Parallel LZ77 Decoding using a GPU

Emmanuel Morfiadakis

Supervisor: Dr. Eric McCreath

Submitted as part of COMP4560, in partial fulfilment of the requirements for the Bachelor of Advanced Computing at the Australian National University, May 2017

Abstract

Data compression, as a process, aims to satisfy the modern world's need for speed and efficiency by reducing the cost of storing and transmitting information. Over the past few years, there have been several attempts to improve the performance and reduce the execution times of older compression algorithms by adapting them to make use of parallel programming architectures and hardware. The prevalent trend, however, is to focus on compression; the reverse process of decompression is largely ignored.

In this report, we develop and compare two versions of the LZ77 decompression algorithm, one that executes sequentially on a Central Processing Unit (CPU) and one that runs in parallel on a GPU (Graphics Processing Unit). Out aim is to investigate the running times of the algorithms, and the memory transfer and thread synchronization costs of the GPU implementation, in an effort to determine if there are any significant performance gains by offloading this operation to the GPU for completion.

We experimentally show that the serial execution manages to constantly outperform the parallel one. This is due to the significant memory access and thread waiting times that arise when trying to resolve the data dependencies present in the LZ77 encoded input. Even though memory transfer times remain relatively constant as the input size grows, we conclude that the LZ77 decompression algorithm can not be parallelised efficiently with an approach that relies on in-memory synchronisation.

Acknowledgements

I would like to express my thanks to:

- My supervisor, Dr. Eric McCreath, for his guidance and mentoring over the duration of this semester, and for providing me with all the necessary resources and materials needed to complete this project.

- Dr. Weifa Liang and Dr. Peter Strazdins, for organising the community practise meetings and showing a genuine interest in ensuring the success of all student projects, including this one.

- My family and friends for their understanding, support, love, and most importantly patience.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In its modern form, data compression is an ubiquitous process that enables many complex activities, such as long-distance video communications, streaming movies through on-line services or physical media, and even browsing the Internet [1]. The field itself has been described as a necessity for our world, where speed and efficiency hold a central role [2], so it is not hard to imagine the benefits of improving the rate at which information is compressed and decompressed.

With the emergence of General-Purpose Graphics Processing Unit (GPGPU) computing in 2001 [3] and over the course of the last decade, new and faster ways to perform large scale calculations have appeared, that utilise the massively parallel architectures of GPUs over the conventional, sequential nature of CPUs [4]. GPUs excel at completing tasks that can benefit from data parallelism, that is, performing many independent, simple operations on large sets of data [5].

In this report, we explore the potential of parallelising the LZ77 decompression algorithm with the use of a GPU card. We develop two programs implementing the LZ77 decompression process, one written in C that performs the operation serially, and one written in CUDA that decompresses in parallel. Our aim is to determine whether the parallel implementation provides any performance advantages in terms of the overall execution time, by bench-marking it against the serial implementation.

Given that GPU execution introduces some undesirable overheads and bottlenecks [6] [7], we are also interested in investigating their effect in the running time of the parallel implementation. The first bottleneck is the memory transfer time, or the time required to move data from the CPU to the GPU for processing. The second one is the overhead related to the synchronisation cost of threads, the basic parallel entity in GPU programming.

1.1 Motivation

One of the most important applications of the LZ77 encoding/decoding scheme, along with Huffman Encoding, is in DEFLATE: an algorithm developed by Phillip Katz that is used extensively in applications such as the HTTP (Hyper-Text Transfer Protocol), image formats such as .MNG and .PNG (Multiple-image Network Graphics and Portable Network Graphics respectively), and .PDF (Portable Document Format) files [8]. Given the pervasiveness of DEFLATE, we are interested to see if offloading the decompression process of an encoded LZ77 input to a GPU is faster than performing the decompression on a CPU.

1.2 Contributions

Our main contribution is that, overall, the LZ77 decompression algorithm is a bad candidate for paralellisation, due to its inherent sequential nature.

We observe that our attempt to implement memory-based synchronisation for the parallel approach is limited by slow memory read/write/access times. In conjunction with thread stalls at synchronisation barriers, we fail to outperform the serial implementation.

Hence, we recommend against using in-memory synchronisation for this task, if the desired outcome is a fast and efficient parallel decoder. Instead, other options should be pursued.

1.3 Outline

We start by providing the necessary background information in Chapter 2. This includes the LZ77 compression algorithm and the emergence of parallel computing, with a focus on the CUDA programming language. We proceed by briefly presenting an overview of related research and other attempts of parallelising the algorithm in Chapter 3.

The main section of the report begins in Chapter 4, which provides a theoretical, high-level overview of the serial and parallel decoding algorithms and their design. In Chapter 5, we link theory and practise, and discuss the implementation of the algorithms in software, and how this process was affected by limitations of the utilised frameworks, in particular CUDA.

Finally, we describe the experimental setup and then proceed to present our results in Chapter 6, before concluding the report in Chapter 7 and providing suggestions for future work.

Chapter 2

Background Information

2.1 Data compression

The term "data compression" refers to the process of reducing the number of bits used in transmitting and storing information [9]. Even though this practise appeared as early as the invention of the Morse code in 1893, it wasn't until the late 1940s that this field started to receive attention from the mathematical community, a side-effect of the emergence of Information Theory [10].

According to Sayood [11], a data compression algorithm or technique actually refers to two separate algorithms: The first one, A_1, operates on an input X and generates an output X_c, that typically requires fewer bits to represent. A_1 is known as the compression algorithm. The second algorithm, A_2, is called the decompression algorithm and operates on X_c to produce an output Y. A summary of this mathematical model can be seen in Figure 2.1 below.

$$X_C = A_1(X)$$
$$Y = A_2(X_c)$$

Figure 2.1: A mathematical model of data compression

Ideally, the output Y of the decompression algorithm is identical to the input X of the compression algorithm; in this case, the compression scheme is referred to as “lossless” [11]. Text files are a typical example where someone would employ a lossless algorithm, where even one character change could lead to ambiguity.

If the output Y is only an approximation of X, then the algorithm is characterised as “lossy”: In this case, loss of information occurs during the compression process [11]. Lossy algorithms are typically used in situations where small inaccuracies are not catastrophic; as an example, video compression typically uses lossy algorithms.

2.2 The LZ77 Algorithm

2.2.1 An overview

The main compression scheme that this report is concerned with is the LZ77 algorithm, which was developed in 1977 by Jacob Ziv and Abraham Lempel [12]. LZ77 is an example of lossless compression: It attempts to reduce the size of the compression output by replacing recurring data sequences with references to data that appears earlier in the uncompressed input [13].

At every point, the algorithm maintains two first in, first out buffers, called the “sliding window” and the “lookahead buffer”. The sliding window is used to store the recent past output, while the lookahead buffer contains the data to be encoded next [13][14]. The algorithm attempts to find the longest matching sequence of symbols between these two buffers, starting with the first symbol in the lookahead buffer [15]. If a match is found, then the sequence in the lookahead buffer is replaced with a reference, usually a 3-tuple, containing the distance between the two sequences, the number of matched characters, and the first character that follows the sequence in the lookahead buffer [16].

Apart from its popularity as part of the DEFLATE algorithm, the now nearly 40-years-old LZ77 algorithm and some of its variants (such as LZR or LZSS) are still being utilised by programs such as pkzip, gzip, Stacker, and by operating systems such as Microsoft Windows [17].

The algorithm itself is relatively simple to understand, and pseudocode for the operation of the encoder can be found in Appendix 8.D.

2.2.2 A simple example

This section will provide a detailed explanation of the LZ77 encoding and decoding processes in the form of a simple example, based on material adapted from Jens Müller [16]. It is recommended that readers unfamiliar with these processes go through the examples. Individuals that have encountered these concepts before may choose to skip this section.

We'll first illustrate the encoding of the string "**lalaal**" with lookahead buffer and sliding window sizes of 2, and use 3-tuples to encode references to previous data. Table 2.1 contains the output of the encoding process as each character passes through the lookahead buffer, while each step is explained in detail in the next page.

Sliding Window	**Lookahead Buffer**	**Remaining Input**	**Output**
		lalaal	
	la	laal	(0, 0, l)
l	al	aal	(0, 0, a)
la	la	al	(2, 2, a)
aa	l		(0, 0, l)

Table 2.1: LZ77 Compression example

Step 1: The **Lookahead Buffer** is originally empty. The first two characters from the **Remaining Input** then slide into the **Lookahead Buffer** to fill it.

Step 2: The symbol at the start of the **Lookahead Buffer** is **l**, which is not present in the **Sliding Window**. Hence, we can't encode this character as a reference to previous data. We output **(0, 0, l)**, since no sequence has been matched. We only processed one character, so we slide the buffer contents by one position.

Step 3: The symbol at the start of the **Lookahead Buffer** is **a**, which is not present in the **Sliding Window**. Hence, similarly to step 1, we output **(0, 0, a)** to indicate that we failed to match a sequence. Since we only processed one character, we slide the buffer contents by one position.

Step 4: The contents of both buffers are the same sequence. Thus, we can encode **la** as the tuple **(2, 2, a)**. The first number indicates the distance between the matching sequences, while the second one indicates the length of the matched sequence. Finally, the character **a** indicates the first character following the sequence in the **Lookahead Buffer**: In this case it is **a**, the next character in the **Remaining Input**. Since we have processed three characters (**lal**), we slide the buffer contents by three positions.

Step 5: The symbol at the start of the **Lookahead Buffer** is **l**, which is not present in the **Sliding Window**. Hence, the output at this step is **(0, 0, l)**, similarly to steps 1 and 2. The buffer contents slide by 1 position, and since the lookahead buffer is now empty, the algorithm terminates (as we have no other characters to encode).

As such, the encoded output is (0, 0, l)(0, 0, a)(2, 2, a)(0, 0, l).

Working in reverse, we can easily decode the sequence:

Step 1: We replace each tuple[1] that represents only a single character with that character. In

[1]the terms resolving, processing, decoding, decompressing, replacing a tuple are used interchangeably in this report, and refer to the process of replacing a tuple with the data it references.

terms of the encoded output above, with replace the first, second and fourth tuples with **l**, **a**, and **l** respectively and we have **la(2, 2, a)l** .

Step 2: We replace all other tuples with non-zero lengths and offsets of the form **(x, y, z)** by copying the first **y** characters that are at an offset of **x** from the start of the tuple, and we append **z** at the end. Since tuples only contain characters that appear earlier in the input, the decoding order is from left to right. In this scenario, we only have one such tuple: **(2, 2, a)**. Its contents suggest: move two characters to the left, copy the next two characters, and replace this tuple by the copied characters and the third element of the tuple. In the case of **la(2, 2, a)l**, this means copy **la**, append **a** to it and replace the tuple with it:

$$\mathbf{la} + \mathbf{laa} + \mathbf{l} = \mathbf{lalaal}$$

which is the original string.

2.3 Graphics Processing Unit Computing

2.3.1 A short history

Historically, Graphics Processing Units, **GPUs** for short, have been associated with fast image rendering and videogames, according to Du et al. [3]. However, advances in the early 2000s sparked an interest in using GPUs as a general computational tool, instead of a dedicated device used for graphics rendering; the fact that some operations where able to execute faster on these devices, such as the LU factorisation for matrices, was enough to attract the interest of the scientific community.

In its early days, GPU computing was not an easy task: the lack of high-level frameworks meant that people interested in using these devices for computational purposes, and not graphics generation, had to burden themselves with knowledge of the underlyding graphics APIs and adapt their code to make use of fixed-function pipelines: GPU instructions that could only be parametrised to a certain extent [18] [19]. Some of the more widespread graphics frameworks that were adapted for early GPU computing include the openGL language by the Khronos group and DirectX by Microsoft [20].

As a result of this difficulty in using GPUs for computing, there were several attempts to produce more abstract frameworks, that removed the need of understanding the underlying graphics APIs in detail, with some notable early examples including Brook, Sh, RapidMind, and Accelarator [20]. However, even though these languages were an attempt to form an abstraction layer that would simplify interactions with GPUs, they were still unsuited as languages for general-purpose computation [20].

The first programming framework that was designed to cater to people that were looking to use GPUs for high performance computation and not graphics generation and processing came in the form of NVIDIA's Compute Unified Device Architecture in 2007, now known as CUDA[21]. CUDA was designed from its core to be a language focused entirely on parallel programming, and can thus represent concepts such as threads, shared memory, synchronisation and so on, that are prominent in concurrent and parallel programming paradigms [3].

Other high-perfomance languages for GPU computing followed shortly, including Microsoft's DirectCompute, and openCL by the Khronos group [3]. The frameworks that currently dominate this field include CUDA, openCL and Stream, with recent attempts to standardize and converge different General Purpose Graphics Processing Unit computing (GPGPU computing) frameworks [22].

2.3.2 The CUDA language

According to the official website [23], CUDA is a parallel computing platform and programming model by the NVIDIA corporation, which allows users to generate high-perfomance parallel code with minimal understanding of the underlying GPU architecture. It acts as an extension to the C, C++, and Fortran programming languages. It enhances these languages with the addition of parallel constructs and a high-level API that allows direct use of GPU devices. The framework itself is compatible with most mainstream operating systems, but only supports NVIDIA-manufactured GPUs.

CUDA programs distinguish between two different, heterogeneous systems. The first one comprises the CPU (one or multiple) along with its memory space, which is the traditional computing architecture. In CUDA terms, this system is called the “host”. The second system contains one or more GPUs with their memories. This system is called a “device”. Different parts of CUDA code can be configured to execute either on the host or on a device, as required [24].

2.3.3 CUDA and C

The contents of this section aim to present a simple CUDA program and explain some basic parallel programming concepts. The example is based on material adapted from Cyrill Zeller [24]. It is assumed that readers are already familiar with the C programming language, or the general structure of C-like languages. Appendix E in Section 8.E contains compilable source code that illustrates the concepts described in this section. The program attempts to increment each of the one million elements in an integer array a million times.

```
__global__ void addmill(...){
    ...
}

int main(){
    ...
    cudaMalloc();
    cudaMemcpy();
    addmill <<< blocks, block_size >>> (...)
    cudaMemcpy();
    cudaFree();
    ...
    return 0;
}
```

Figure 2.2: Typical structure of a CUDA program, illustrating memory management, kernel invocations, and function specifiers.

Overall, CUDA provides a small set of extensions and Application Programming Interfaces

(APIs) that enhance the C programming language and allow it to be used for general-purpose parallel computation. As such, most of the structure of a CUDA program resembles a C program. Figure 2.2 shows the basic skeleton of a CUDA program, where this similarity is apparent.

CUDA code can execute on either a device (GPU) or on a host (CPU). The specification of the target on which each function is executed is done through a new syntactic element, called a "**function identifier**", which is placed in front of the function definition. For example, the *__global__* and *__device__* identifiers indicate that a function will be executed on the GPU. No identifier means that a function executes on the CPU serially, as is the case for most programming languages.

Invocation of GPU functions can be done from anywhere in the code. However, it differs from a typical C function call. In addition to calling the function name and providing the appropriate parameters, the number of concurrent entities that will execute the function code must also be specified. In CUDA, these entities are threads, which are further grouped into blocks of threads. Specifying the number of threads that will execute a GPU function is done by a "**kernel invocation**" statement, placed between the function call name and the parameters. The syntax for this is simple: <<<**X, Y**>>> where **X** is the block number, and **Y** is the number of threads in each block. As such, the total number of threads that will execute the code inside the function is **X*Y**.

Of course, it is not only sufficient to establish the number of parallel entities that will execute a function; data and parameters also needs to be transferred between the CPU and GPU, as both entities have different memory spaces in CUDA. Memory needs to be allocated twice, once for the host and once for the device, and then copied to and from as required. In typical CUDA programs, the CPU is responsible for initialising data, which is then passed on to the GPU to be processed. At the end of processing, the data is returned back to the CPU, so it can be stored, printed, manipulated further, and so on.

Memory allocation and management on the GPU (device memory space) is done through the system calls *cudaMalloc*, *cudaMemcpy*, *cudaFree*, which serve functions similar to the C *malloc*, *memcpy*, *free* system calls. Given that we are dealing directly with memory, the parameters of functions that execute on the GPU are usually pointers to the data stored in the device memory space.

Summarising this brief overview, CUDA programs use **threads** to execute programs in parallel. Functions can be configured to run either on the **CPU** or the **GPU**. GPU processing requires appropriate **handling of memory**, given that the device and host memory spaces are **distinct**. There are more concepts around this programming paradigm, but what has been discussed so far will allow the readers to understand the basic structure of a CUDA program, and by extension, the rest of this report.

Chapter 3

Related Work

As mentioned in Section 2.3.1, mainstream use of GPUs for general purpose computing only started around 2007, when CUDA was launched. As such, literature that discusses implementations of decompression algorithms on GPUs is non-existent before this time period. However, there is a number of earlier publications that attempt to provide parallel versions of the LZ77 algorithm or its derivatives for abstract machines.

The earliest relevant publication dates to 1995. In their paper, "Optimal Parallel Dictionary Matching and Compression", Farach et al. [25] present a work-optimal algorithm that achieves a logarithmic ($O(logn)$) decompression time for dynamic dictionary compression, which is the type of compression the LZ77 algorithm uses. This is done by generating tree data structures, based on the compressed LZ77 strings, and then transforming the decompression process to a process of matching nodes with trees. Their algorithm was targetted towards Parallel Random Access Machines (PRAMs) with CRCW (Concurrent Read Concurrent Write) operations, which are general abstract entities used to model parallel computations. [26]

In a similar vein in 2004, De Agostino [27] utilised concepts from Graph Theory, in particular Eulerian Tours, to derive a $O(logn)$ time implementation of a parallel LZ77 (or LZ1, as it is called in the paper) decoder. Again, this implementation is purely theoretical and it is targetted towards an abstract machine, a PRAM EREW (Exclusive Read Exclusive Write) in this case.

Over the next few years, hardly any literature exists that describes parallel implementations of LZ77 on GPUs. The closest and most related pieces of work tend to focus on LZSS and LZW, which are derivatives of LZ77 and LZ78 respectively. In addition, these publications tend to investigate the prospects of parallelizing the compression process, rather than the decompression process, which is dealt with to a lesser extent [28] [29].

In particular, Ozsoy et al. [29] developed a CUDA-based parallel LZSS algorithm that manages to outperform other serial and parallel-threaded implementations by a factor of 3x and 18x respectively, without any losses in compression ratio . Furthermore, Funasaka et al. [28] were able to derive a parallel LZW implementation by using the same framework; their result was an algorithm that is 69.4 times faster than its sequential counterpart, whose main application is in the processing of big data .

The only publication that deals explicitly with both the LZ77 algorithm and GPU computing revolves around the DEFLATE algorithm, of which LZ77 is a part, as was discussed earlier. In their work, Sitaridi et al. [30] derive a parallel DEFLATE algorithm, called Gompresso, by using the CUDA framework. In particular, they investigate two different methods of simplifying the resolution of data dependencies between LZ77 tuples, which according to their paper, is the main challenge in parallelizing the algorithm. The first method involves a voting procedure that makes use of CUDA programming primitives to eliminate the synchronisation costs of resolving tuples that reference other tuples, while the second method attempts to eliminate synchronization costs during the encoding phase of the algorithm. Their first scheme, called "Multi-round resolution of nested backreferences" has been partially utilised in the implementation of the parallel algorithm in this project, and its basic operation will be described in Section 4.2.

Chapter 4

Algorithms

In this section, we'll provide an overview of the two main algorithms that this report is concerned with, which are the serial and parallel decoding procedures. A third algorithm was also developed, in order to assist with the generation of encoded test data and solidify the author's understanding of the LZ77 format. However, it is out of the scope of this report. Nevertheless, it is provided as part of the software artefact source files, if the readers wish to generate their own data and test the provided decoding software.

4.1 Serial Decoder

4.1.1 Data format

We are focusing on byte-level decoding of data, and not bit-level. That is, we will not deal with binary sequences, but rather with sequences of text, containing Latin characters, symbols, and numbers.

One simplification has been introduced with respect to the version of the algorithm presented in Section 2.2.1, to make the process of parsing the input data simpler: literals have been separated from tuples that reference previous sections of the encoded input, and are treated as separate entities instead. That is, tuples such as **(2, 2, a)** are instead represented as **(2, 2)(0, 0, a)**. This does affects the data compression ratio, but since we are not interested in the encoding capabilities of the LZ77 algorithm, but rather in the potential performance gains of a parallel decoding implementation, there are no unwanted consequences introduced by this simple change. For brevity, we'll refer to tuples of the form $(0, 0, c)$ where c is any character as literals, and we'll call tuples of the form (x, y) backreferences, where x, y are positive integers.

With these two constraints, the algorithm is capable of recognizing any sequence of encoded LZ77 tuples, as described previously in this report, provided they are valid and unambiguous. The input to the algorithm is a text file (*.txt*) containing a sequence of encoded LZ77 tuples. The precise format will be discussed in the next section. For now, we focus specifically on the serial decoding algorithm.

4.1.2 Overview

The algorithm implementation for the serial decoding procedure follows naturally from the description outlined in Section 2.2.1, and contains two different steps. The first step is responsible for parsing the text file containing the input, while the second step consists of the actual decoding procedure.

During the first step, the algorithm parses the encoded tuples from the specified input file. We extract the offset and length values from backreference tuples, and the literal values of literal tuples, which are stored in two arrays, named *coordinates* and *symbols*. A helper variable, called *pointer*, is used to keep track number of how many tuples have been processed and are stored in the arrays so far.

For the ith encoded tuple of the input stream that has the form (x, y, z), $symbols[i]$ will contain z, $coordinates[2 * i]$ will contain x and $coordinates[2 * i + 1]$ will contain y. In the case of backreferences, which have the form (x, y), then $symbols[i]$ will contain a space character. An example of the mapping between the text input and the two arrays can be seen in Figure 4.1.

This transformation is used to separate the processes of parsing and decoding the input, given that we are only interested in the latter. In addition, arrays can be used directly by both the parallel and serial algorithms. We have therefore eliminated any potential performance differences due to differing data structures between the two algorithms.

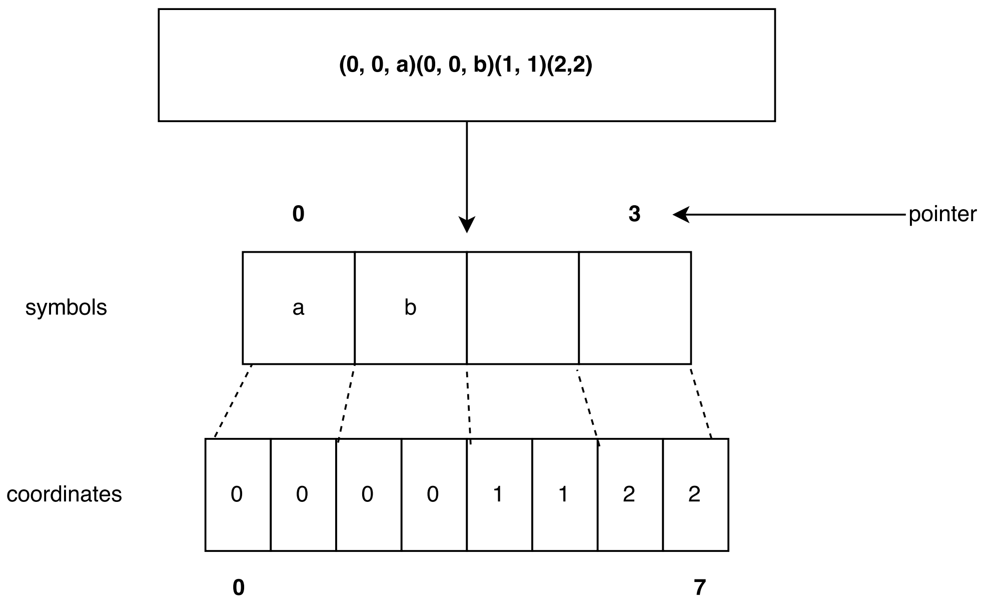


Figure 4.1: Array representations of the encoded tuples

The second step consists of the actual decoding process. As a starting point, we declare a new character array, called *output*, which will contain the decoded text sequence, and an integer variable, *position*, which points to the current writing position in the *output* array. Then, for the ith tuple with $0 \leq i < pointer$, whose elements are stored in $symbols[i]$, $coordinates[2 * i]$ and $coordinates[2 * i + 1]$, we have two options, based on whether or not the tuple is a backreference (which can be determined by the value stored in $coordinates[2 * i]$):

- If the tuple is not a backreference, then it is a literal, so we copy the content of $symbols[i]$ to the current output position and increment *position* by 1.
- Otherwise, resolving a backreference consists of copying the correct part of the already-decoded output. By using the data structures defined in this section, we have to copy the first $coordinates[2 * i + 1]$ characters starting at index $position - coordinates[2 * i]$ of the output to the current $position$. We then increment $position$ by $coordinates[2 * i + 1]$, which corresponds to the length of the backreference that was just resolved.

We do this for every tuple from left to right, until all of them have been processed. This procedure is summarized in Pseudocode form in Algorithm 1 below.

In order to ensure that we don't run out of space in array declarations, their upper bound l is set to be the length of the encoded input sequence. In general, this is sufficient, unless the size of the compressed sequence is much larger than the size of the uncompressed sequence. This case is not addressed in this algorithm.

4.1.3 Algorithm Pseudocode

```
Data: LZ77 Encoding of a text sequence
Result: Decoded text sequence
l := length of the encoded text sequence;
symbols := character array [0 .. l];
coordinates := integer array [0 .. 2 * l];
pointer := 0;
while not all tuples in the encoded input have been processed do
    t := next tuple to be processed;
    if t is a back-reference tuple then
        symbols[pointer] := "";
        coordinates[2 * pointer] := encoded length of t;
        coordinates[2 * pointer + 1] := encoded offset of t;
    else if t is a literal tuple then
        symbols[pointer] := literal in t;
        coordinates[2*pointer] := 0;
        coordinates[2*pointer + 1] := 0;
    pointer := pointer + 1;
end
output := character array[0 .. l];
position := 0 ;
for i = 0 to pointer do
    if coordinates[2 * i + 1] == 0 then
        output[position] := symbols[i];
        position := position + 1;
    else
        for c = 0 to coordinates[2 * pointer + 1] do
            output[position + c] := output[position + c - coordinates[2 * i]] ;
        end
        position := position + coordinates[2 * i + 1];
end
```

Algorithm 1: LZ77 Serial Decoding process

4.1.4 Algorithm analysis

We only consider the decoding procedure, which is of interest to us, and not parsing. Since we are resolving tuples from left to right, it is guaranteed that the portion of the output required to resolve a backreference tuple will be available before that tuple is reached. This serial processing guarantees that the algorithm will eventually terminate, once all tuples have been resolved.

In terms of performance, the algorithm resolves one tuple at a time, and the resolution process is constant, given that we are either copying a single character to the output, or copying from an earlier section of the output. In the second case, the number of copied characters is bounded by the size of the sliding window that was used in the encoding, which is a constant number. Therefore, for n encoded tuples, we have a linear complexity of $O(n)$.

In terms of space complexity, the algorithm uses a single array to store the final and all intermediate states of the decoded sequence. The size of this array is proportional to the size of input, the encoded text sequence. As such, space complexity is also linear: $O(n)$.

4.2 Parallel Decoder

4.2.1 Data format

The data format is exactly the same as in the serial decoder. Valid inputs to the serial decoder are also accepted by the parallel decoder.

4.2.2 Overview

Again, this algorithm is divided to 2 steps. During the first step, the input file is parsed and mapped onto two arrays, exactly as described in Section 4.1. However, in order to avoid complicating the pseudocode and to focus on the parallel concepts of the algorithm, we won't refer explicitly to these underlying data structures in this section. Instead, we'll adopt a more abstract model, and assume that all encoded tuples are stored as tuples in memory.

The second step consists of the decoding procedure. Instead of focusing on one tuple at a time, we attempt to decode them all in parallel. We'll use the term "thread" to refer to the concurrent entities responsible for performing the decoding operation in parallel. In this algorithm, each thread is responsible for decoding one encoded tuple.

The main hurdle in developing a parallel implementation is the resolution of backreferences. In particular, how can we be certain that the portion of the output required to resolve a back-reference has been decoded and is available, given that all threads operate independently and non-deterministically? This problem is illustrated in Figure 4.2. In the topmost encoded sequence, the tuples are independent and can be decoded in parallel. In the bottom sequence however, the middle four tuples require the resolution of the previous tuple first. This means that only the first and last one can be decoded in parallel. As such, synchronization between threads is essential in determining when a tuple can be resolved.

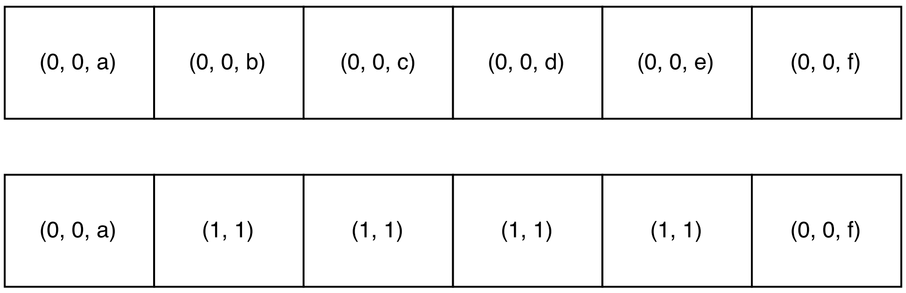


Figure 4.2: Tuples with nested backreferences force sequential decoding

In dealing with this problem, the algorithm adopts a similar procedure as to the Multi-Round Resolution (MRR) protocol used in Sitaridi et al [30]. The main idea is that the encoded sequence will be decoded in multiple rounds. On the first round, all literal tuples will be decoded. On the second round, backreferences of depth 1 will be resolved, followed by backreferences of depth 2 at the next round, and so on. In this scenario, the depth of a backreference refers to the number of backreferences that need to be resolved first, before it can be resolved. Literal tuples do not rely on other data, so they have a depth of 0.

Referring back to Figure 4.2, in the bottom sequence, the respective backreferences of the encoded tuples are $[0, 1, 2, 3, 4, 0]$, given that the four middle tuples depend on the previous one being decoded, before they can be resolved. Essentially, we are trying to decode every backreference as soon as the referenced data becomes available in the output, by grouping together tuples with the same number of data dependencies on other tuples.

Initially, and since decoding is done in parallel, each thread must determine its writing location on the output. As each thread is processing one encoded tuple, we use a parallel prefix sums calculation to determine the write location based on the encoded length of the tuple. For each literal tuple, the prefix is assigned to be 1, whereas for the backreference tuples, the prefix is the corresponding encoded length. Thus, each tuple is represented by the number of characters it will write to the output. We store these values in an array called *sums*, of size equal to the number of encoded tuples.

If we assume that there are n tuples and threads, and each thread calculates the prefix sum for one element, then this whole process can be completed in $\lg n$ steps. At each step, every element n of the *sums* array is updated according to the formula $sums[n] = sums[n] + sums[i - n]$. The parameter i starts as 1 and doubles at the end of each step. Figure 4.3 illustrates the operation of the algorithm for a small array.

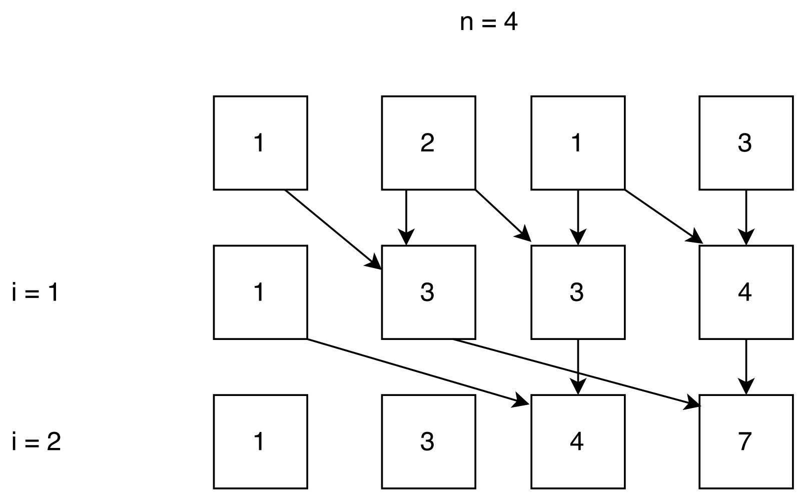


Figure 4.3: Parallel prefix sums calculation for an array of 4 elements

What the parallel prefix sums algorithm does is to determine the inclusive sum of the current and all the previous elements of the array. In this case, we are summing the number of characters written by all the preceding tuples. Thus, *sums*$[n]$ will contain the last index of the output that the nth tuple will write to. Therefore, by using the *sums* array for reference, each thread knows its writing location in the output.

We then proceed to the main decoding loop. At the beginning of each iteration, all threads synchronise, to determine whether or not they can proceed with the decoding of their respective tuples. If a thread is able to do so, it writes its contents at the location calculated by the prefix sums operation and informs the other threads that this part of the output is available, should other threads depend on it. Otherwise, the thread waits for the next round.

Synchronisation is done with a bit vector of n elements (where n is the number of encoded tuples in the input). This vector initially contains all 0s. Once the ith tuple has been resolved, the bit at position i of the vector is set to 1. Before the main decoding iteration begins, each thread must determine the range of tuples that need to be resolved first, before it can proceed. This information is stored as a *lower* and *upper* index. At the end of each iteration, the thread checks if the bit vector elements between these two indices containg consecutive 1s. If this is the case, it can proceed with decoding its tuple, since all referenced data is available. Otherwise, it has to wait for the next round.

Figure 4.4 showcases an example of how this process works. Determining the range of referenced tuples is done based on the location that each tuple will be written to, and its length and offset, which are parameters that are calculated in previous parts of the algorithm. In this figure, tuple T4 depends on T3 only, so the index range is $3 - 3$, if T1 has an index of 0. At the first round, all literal tuples are resolved, so the third element of the bit vector remains 0 to reflect that T3 was not resolved. In round 2 however, T3 is resolved since its a backreference with a depth of 1. As such, the third bit is flipped to 1. This indicates that at round 3, T4 will be able to resolve its encoded tuple.

Overall, there are as many resolution rounds as there are nested backreferences. Hence, the performance of this algorithm depends the maximum depth of the encoded tuples.

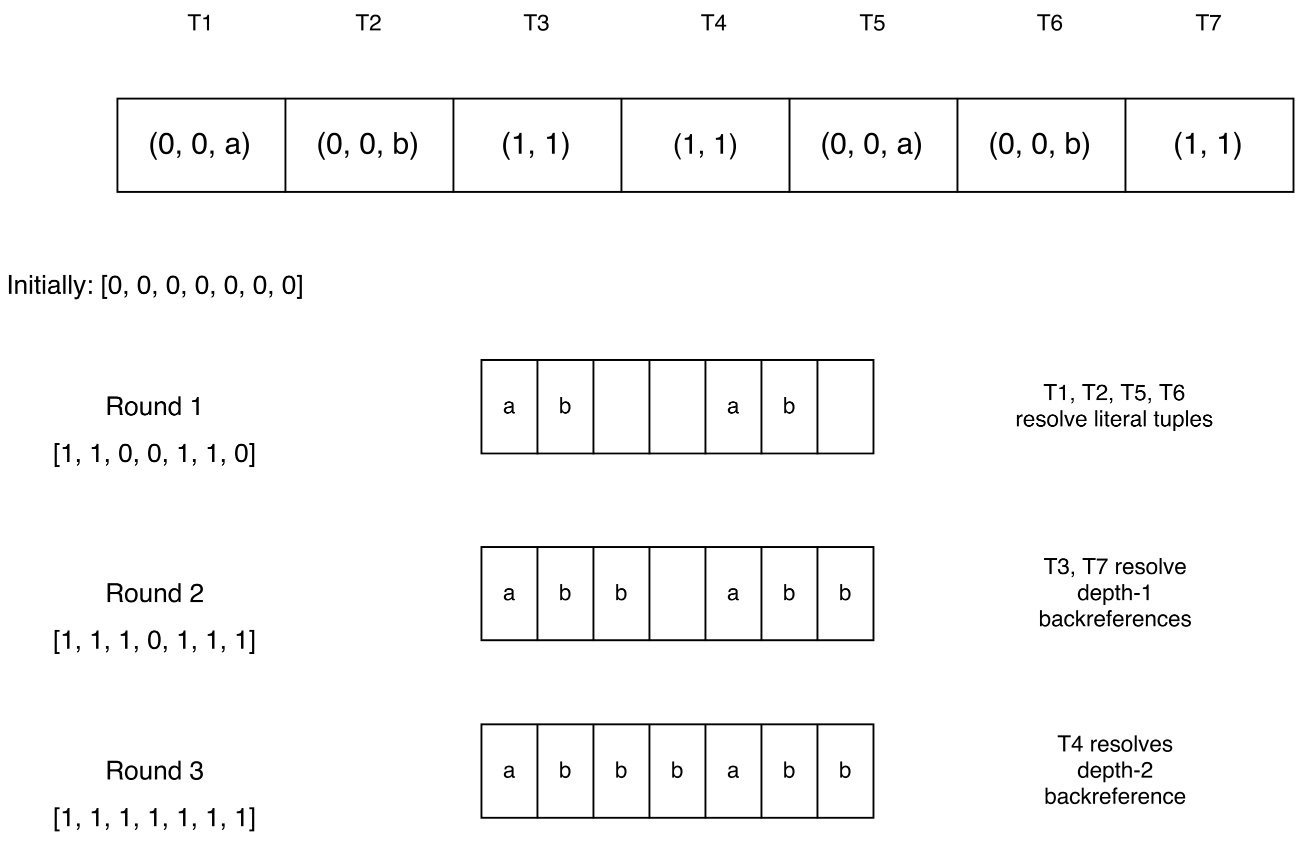


Figure 4.4: Resolving backreference depths in parallel

4.2.3 Algorithm Pseudocode

Algorithm 2 illustrates the decoding process, excluding the parsing part, which is identical to the procedure described in the previous section.

We start by calculating the prefix sums of the tuples, to determine their output position, and continue with the main decoding loop, which decodes the input in successive rounds. Each thread resolves one tuple.

4.2.4 Algorithm analysis

Termination is guaranteed by the fact that at least one tuple will be processed in each round. As such, the portion of the output required for resolving each encoded tuple will eventually become available. There are as many resolution rounds as there are nested backreference depths, so eventually all tuples will be processed, hence the algorithm terminates.

```
Data: LZ77 Encoding of a text sequence
Result: Decoded text sequence
l := length of the encoded text sequence;
output := character array[0 .. l];
pointer := number of encoded tuples in the input;
bit_vector := integer array [0 .. pointer];
sums := integer array [0 .. pointer] ;
for i := 1 to lg(pointer); par do
    sums := prefix_sum(sums) ;
end
while not all tuples decoded par do
    if (tuple not resolved AND consecutive 1s in bit_vector range) OR tuple is a literal
     then
        resolve tuple;
        copy contents to output index pointed by sums;
        set corresponding bit_vector value to 1;
    else
        wait for the next round;
end
```

Algorithm 2: LZ77 Parallel Decoding process

In the worst case scenario, each encoded tuple depends on the one preceeding it, except the first tuple, which has to be a literal. Then, the algorithm has no choice but to execute $n - 1$ resolution rounds, one for each tuple, until all of them are decoded. The resolution time is constant, since the most expensive operation is to check the bit vector and copy a finite number of characters to the output, which can be done in constant time. Hence, we have an upper time complexity bound of $O(n)$. The prefix sums calculation consists of $O(\lg n)$ steps, so the overall time complexity is again $O(n)$, where n is the number of encoded tuples in the input.

In terms of space complexity, in addition to the output array, we also make use of the prefix sums array, of size n, and the bit vector, of the same size. As such, the space complexity is again linear: $O(n)$.

Chapter 5

Implementation

The algorithms described above have been implemented as two separate programs. This section will expand on specific details of the software implementations.

5.1 Serial Decoder

5.1.1 System specifics

The serial decoder is implemented in the C programming language, in order to minimise performance deviations caused by the use of different programming languages, given that the parallel decoder is written in CUDA-C. In addition, this choice allows the re-usability of code for the operations shared between the two programs, such as I/O processing and parsing the encoded text.

Given that the main development environment consists of a Windows laptop and an Ubuntu desktop, the serial program was designed to compile and run without errors or warnings on Windows 10, and Ubuntu 17.04 "Zesty Zapus".

Timing the decoding step is done with the use of the "*windows.h*" and "*time.h*" libraries, to an accuracy of milliseconds.

5.1.2 Algorithm implementation

The serial algorithm described in Section 4.1 is implemented as is, without any modifications, in the form of a single function call which handles both parsing and decoding. For simplicity, the arrays *symbols* and *characters* used to contain the encoded tuples are created as members of a structure.

The input of the algorithm consists of a single *.txt* file, which contains a sequence of encoded tuples that can be continuous, or seperated by a newline ($\backslash n$) character. The tuple format must adhere to the rules in Section 4.1.1. If in doubt, the encoding utility will encode a given text file to the correct format.

5.1.3 Correctness

Testing for correctness was done through data sets, which are provided as part of the software artefact. These consist of LZ77 encodings of some sample text files. The original, uncompressed sequence, and the decoded output for each data set were then compared with the **diff** program, and in all cases the decoder was able to retrieve the original text contents.

In the case of malformed or unambiguous input, the program will halt execution and inform the user appropriately through the command line.

5.2 Parallel Decoder

5.2.1 System specifics

The parallel decoder has been implemented in CUDA-C version 8.0, originally targetted for the Microsoft Windows operating system, but has been adapted and can compile and execute on most Unix systems, including Ubuntu. However, due to limitations of the CUDA framework, only NVIDIA enabled GPUs can execute the program.

In addition, to simplify the process of copying data between the host and the device, the use of unified memory has been employed. This feature allows these two memory spaces to be treated as one in the code [31], and greatly reduces the number of *cudaMalloc*, *cudaMemcpy* and *cudaFree* statements which tend to become boilerplate code. Unfortunately, the use of Unified Memory requires the use of CUDA version 6.0 or later and a 64-bit system. Thus the program won't be able to run on the more common 32-bit architecture.

In summary, correct execution of this program requires an NVIDIA GPU on a Unix or Windows 64-bit operating system.

Collection of performance metrics and measurements was done with the use of the NVIDIA Nsight tool [32], which is available as part of the CUDA framework.

5.2.2 Algorithm implementation in software

Given some specific characteristics of the CUDA framework, the parallel decoding algorithm had to be adapted before it was implemented into software.

Due to the fact that the decoder program has to deal with an unknown number of encoded tuples, it is unrealistic to dispatch a single thread to decompress a single tuple. The number of threads that can run concurrently varies per GPU type, and it is entirely possible that the input sequence contains more tuples than there are threads to process [33]. In addition, due to the serial nature of the LZ77 encoding scheme, thread stalling is a potential issue. Tuples towards the end of the encoded text would have to wait longer for data to become available, and the threads handling them would therefore have to spend the majority of their time waiting.

The solution to these two issues is the use of striding to process the encoded tuples in blocks. Striding, as a GPU programming technique, refers to re-using a thread for processing multiple data elements [33]. The main parameter that determines how many times a thread will be re-used is called the stride.

This process can be understood by considering figure 5.1. In this scenario, we have 11 elements that need to be processed. A standard choice would be to deploy a single thread for each element, that is 11 threads. If we choose to stride, however, with a stride of, say, 4, only 4 threads are required. As illustrated in the figure, each of the striding threads starts at a different location, and processes each 4th element. For example, thread Th_1 starts at position 1 and processes the elements at positions $1, 5, 9$. Thread Th_2, similarly, starts at position 2 and processes elements $2, 6, 10$, and so on. Overall, there are three striding blocks, whose elements are processed by 4 different threads.

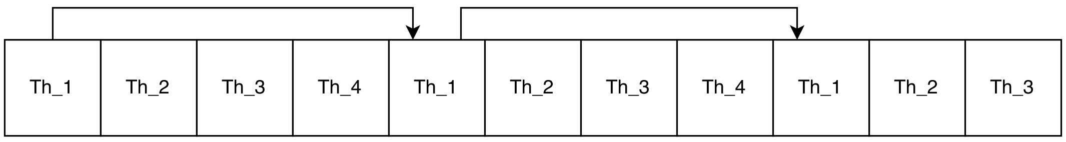


Figure 5.1: Example of striding with a stride of 4

In the software implementation, a constant number of threads is launched, that stride over the *symbols* array. Every thread processes one encoded tuple, and then waits for all the other threads in the striding block to decode their tuples. The stride then moves on to the next block.

As mentioned, striding helps ensure that the entirety of the output is processed, given the GPU-specific thread limit. In addition, by dividing the encoded tuples to striding blocks, we are also able to minimise the stalling of threads, which is now only affected by the local maximum backreference depth of each block, and not the global maximum backreference depth. Finally, the potential overhead of creating and destroying a large number of threads is also eradicated, by keeping the number of threads constant.

The other parts of the algorithm have been implemented as described in the previous section. For simplicity, however, the two arrays (*symbols*, *coordinates*) and the number of encoded tuples are passed to the GPU for encoding as part of a structure. This was done to reduce the number of memory allocation, copying, and error checking statements. This change is also reflected in the serial decoder program, where all necessary data structures for the decoding are members of a structure, in order to ensure consistency between both implementations for the non-decoding parts of the code.

Finally, the synchronisation bit vector has been implemented as a bit array that is reset at the end of each striding block. Given that the previous block are guaranteed to be decoded, it is only necessary to retain the status (processed or not processed) of the tuples in the current block. As such, the bit vector size is equal to the striding parameter, and we avoid the issue of having a potentially large array with large access times.

5.2.3 Correctness

Testing for correctness was, again, done through data sets which are provided as part of the software artefact. An uncompressed text was compressed with the encoding utility program, and then decompressed. The output was compared with the original uncompressed text with the **diff** program, and correctness was verified.

In the case of malformed or unambiguous input, the program will halt execution and inform the user appropriately through the command line.

One potential cause of trouble has to do with the execution of the program on a GPU that is used also used as a display driver. Some operating systems, such as Windows, contain a “watchdog” timer that monitors the execution of threads on the display device, and attempts to reset the ones whose execution time is over a certain threshold [34]. Given that decoding of some files, and hence thread execution time, might exceed this threshold depending on the size of the input, it is recommended that the program is executed on a GPU that is not used as the primary display driver. This will guarantee that the program will not be interrupted by the operating system, and will execute correctly.

Chapter 6

Experimental Results

6.1 Testing Environment

We are interested in the required time for decoding the input (including memory transfer times for the parallel implementation), so we disregard other parts of the program, such as parsing, writing the output to a file, and so on. Eight encoded files of sizes up to 56 MB comprise the data set that is used for both implementations.

For each data set, we run the decoding procedure for a total of 10 times and average the results. We start by comparing the parallel algorithm with different stride values, select the version with the best perfomance, and compare it with the serial implementation.

The experimental evaluation was carried out on a medium-range laptop computer, with an Intel Core i7 4510U running at 2 GHz and 4 GB of RAM and an NVIDIA GeForce 840M GPU, clocked at 1.029 GHz and with 2GB RAM.

6.2 Results

Table 6.1 summarises the experimental results.

	Bible	DNA	Paper	Paper2	Random	Lorem	HelloWorld	Char
Compressed file size (KB)	56385	12244	3440	483	232	131	0.112	0.009
Serial (ms)	22.68	16.05	2.73	0.89	0.22	0.13	0.02	0.02
Parallel - stride 16 (ms)	16673.18	4167.7	764.59	66.88	23.89	18.01	0.022	0.01
Parallel - stride 256 (ms)	2066.72	2096.17	276.87	37.22	5.89	14.08	0.021	0.01
Parallel - stride 1024 (ms)	2981.24	8437.7	806.69	96.12	7.75	61.9	0.021	0.01
Memory transfer time (ms)	0.256	0.158	0.152	0.143	0.135	0.121	0.120	0.118

Table 6.1: Comparison of performance between the two versions of the algorithm

6.2.1 Impact of the striding size

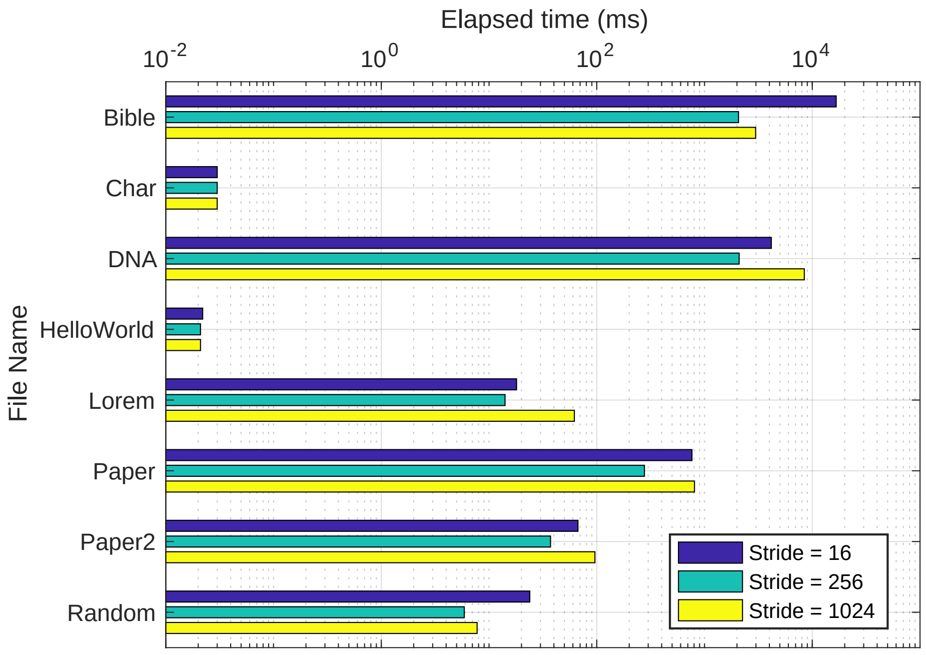


Figure 6.1: Comparison of execution times for the parallel decoder with different stride values

We first start by comparing the performance of the parallel algorithm for different striding sizes, to determine the optimal value for this parameter. Figure 6.1 illustrates the running time of the decoder for strides of sizes 16, 256, 1024. The value that yields the best performance is 256; given that this is the middle value, we can infer that there is a trade-off involved.

This situation can be explained by considering what happens in the other two cases. If the stride is too small, then there is a smaller number of threads that process every block of encoded tuples. This keeps synchronisation times low, but the input is divided to a very large number of small blocks, hence only a tiny portion of the input is decoded at every resolution round. In contrast, for increasing striding sizes, we are able to process a larger part of the input; however, more threads operate on each striding block and as a result, the synchronisation time is longer.

The value of 256 represents a compromise. At this striding size, the trade-off between synchronisation costs and the block size is balanced, and the overall performance is better when compared to the other cases.

6.2.2 Memory transfer overhead

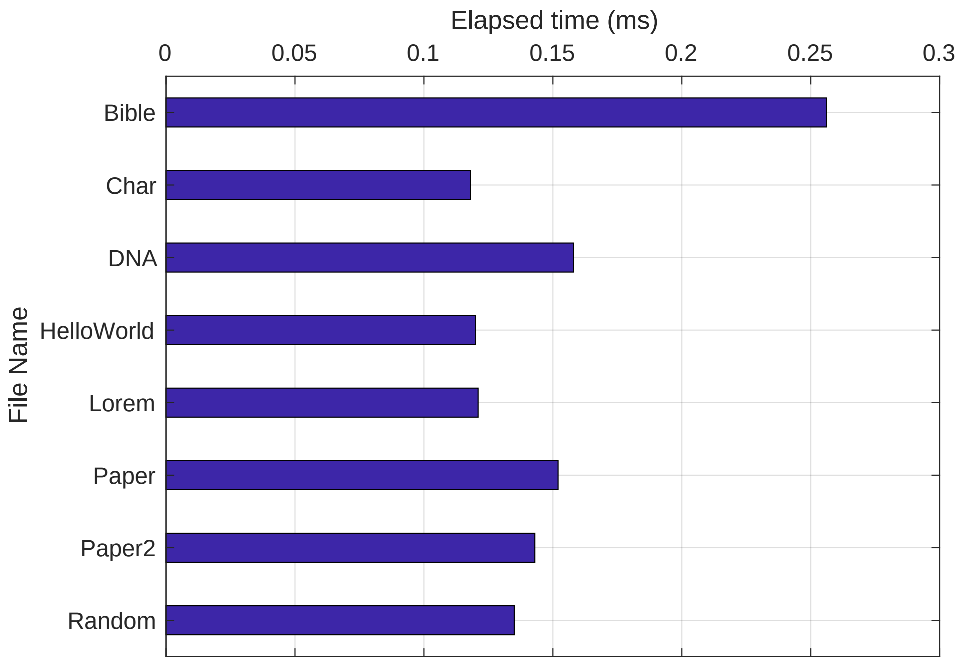


Figure 6.2: Memory transfer times for different files

Next, we consider the impact of memory transfer times to the overall decoding performance. This includes allocation and copying, but not deallocation, which is done after the GPU has completed the transfer of the decoded output to the CPU, and therefore does not affect the decoding procedure.

As illustrated in figure 6.2, the time taken to transfer data between the CPU and the GPU grows slowly as the size of the input increases. For the smaller data sets (Char, HelloWorld), the serial algorithm is able to complete the decoding procedure before the parallel decoder has a chance to finalise all data transfers. However, for all the other, larger input files, the memory transfer time is negligible; it is less than 1% of the parallel decoding time.

We can therefore conclude that memory transfer times only have an impact when the size of the input is small. In these cases, this overhead stalls the GPU long enough and the opportunity to outperform the CPU is gone. For larger inputs, however, the GPU memory times are almost insignificant.

6.2.3 Parallel versus serial performance

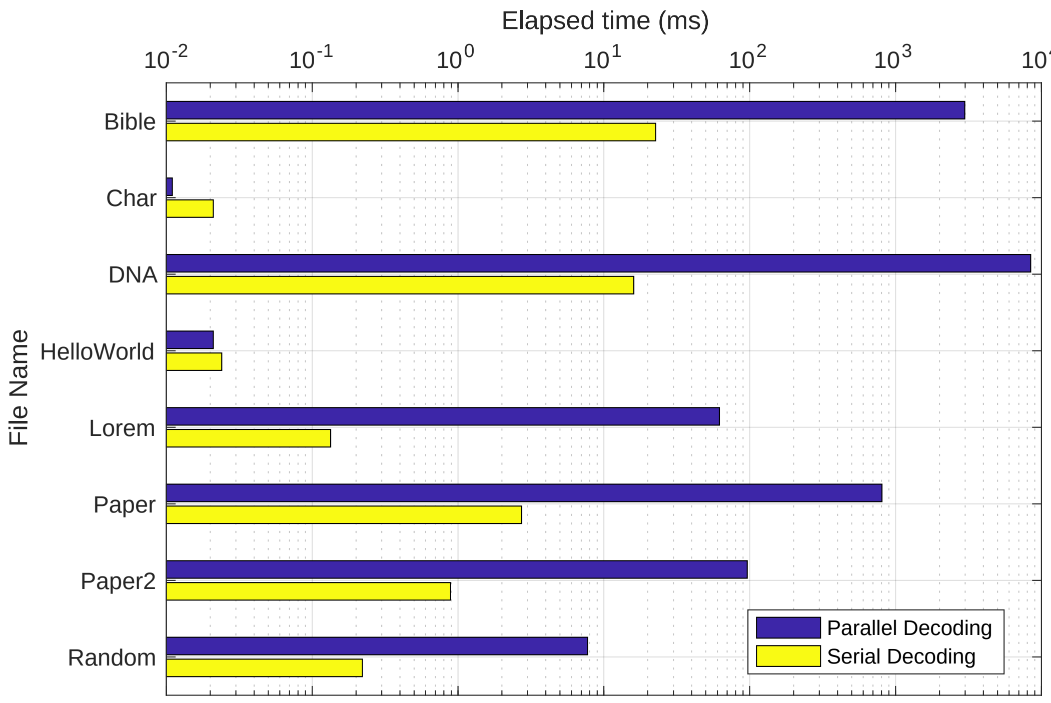


Figure 6.3: Comparison of execution times between parallel and serial implementations

Finally, we compare the parallel and serial implementations against each other in figure 6.3. We use the stride of 256, since it leads to the best parallel decoding time out of all the tested parameters.

We see that the serial implementation manages to outperform the parallel implementation with the exception of Char and HelloWorld, the two smallest data sets. However, in these two cases, the memory transfer overhead stalls the GPU long enough for the CPU to outperform it, as discussed previously. Overall, including the memory transfer overhead, it is evident that the serial implementation is much better in terms of the time required to decode the input.

We have already established that memory transfer times have no significant impact on the decoding time. In addition, the use of striding was employed to keep the number of threads constant, and to minimise the overhead of creating a large number of them. This indicates that the slowdown potentially occurs from synchronisation problems during the parallel decoding loop. We use the NVIDIA Visual Profiler tool to determine the causes of this issue in the next section.

6.3 Thread execution delays

Figure 6.4 contains the main causes that delayed the execution of the different groups of threads that are executed in parallel from the GPU hardware (called a "**warp**" of threads in CUDA terminology), as determined by the Profiler tool. These belong to the *parallel_decode* kernel, whose position in the execution timeline can be seen in Figure 6.5. Between the different data sets, there is a clear trend: three main issues are responsible for the majority of the delays, namely synchronization, execution dependencies, and memory dependencies.

Based on the official documentation [35], synchronization delays are caused when warps of threads are blocked at *_syncthreads*() barriers. In the parallel decoder implementation, this refers to the waiting time at the beginning and at the end of each backreference resolution round. As it is evident, the time spent idly between the decoding of a tuple and moving on to the next block for decoding is causing a significant performance degradation.

Execution dependencies are caused when "an input required by the instruction is not yet available" [35]. This essentially means that memory access is slow, and as a result, instructions cannot be executed because the required data has not yet been loaded from memory. Memory dependencies are similar in the sense that they indicate that reading/writing data to memory is taking longer than anticipated [35]. In the implementation discussed in this report, both of these issues are linked to how threads communicate with each other and determine if they can resolve their tuples. In particular, the operations of determining the bit vector range that must contain consecutive 1s, checking the thread's eligibility to decode its workload at the end of every resolution round, and copying and pasting data to a global array simultaneously are the parts of the parallel decoder that require significant memory read/write/access operations. These costs accumulate, and manifest as delays that stall warps of threads from executing. As evidenced from the profiler screenshots, these dependencies are the main cause behind the slow execution of the parallel decoder.

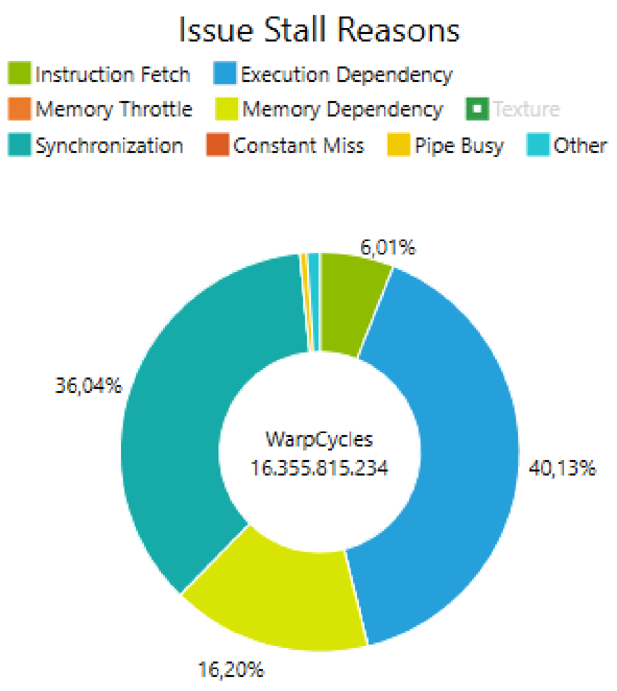
6.4 Summary

Initially, we arrived at the conclusion that memory transfer times become insignificant when the size of the input increases; this parameter scales slowly and can only be considered a bottleneck for small files, typically 200 KB or less.

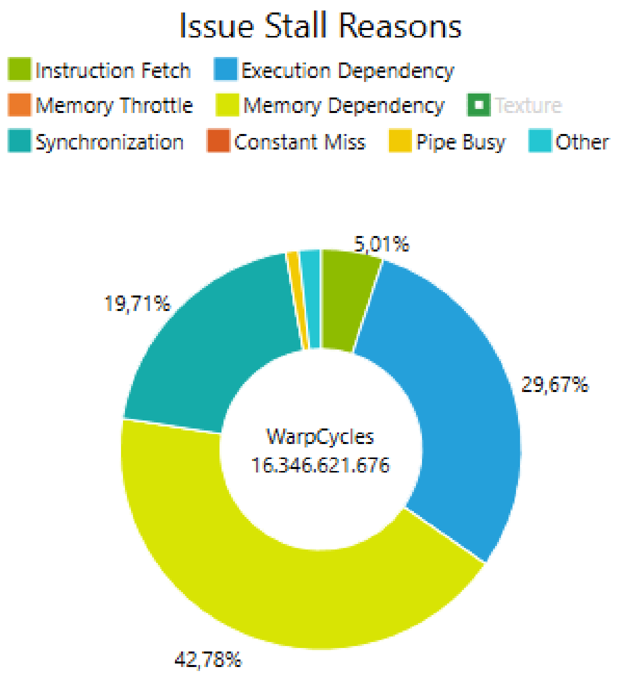
Overall, however, we have shown that a parallel implementation using in-memory synchronisation is slower than a serial implementation.

There are two causes for that. The first one is the time wasted by threads at every resolution round. Specifically, threads stay idle after they resolve their tuples, and before the stride moves on to the next encoded block. The time spent between these two states is significant, and accounts for 20-36% of the stalling in the execution of warps.

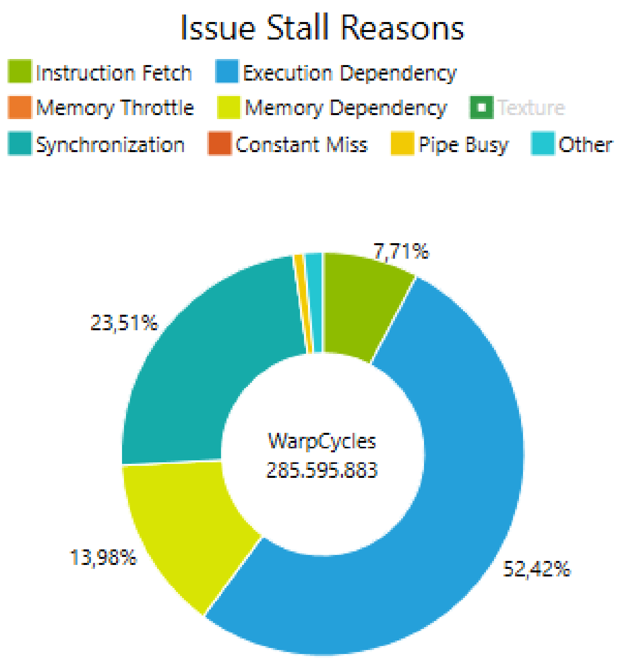
The second cause, which is responsible for over 65% of stalling, has to do with the large amount of memory operations required to synchronise the threads. The LZ77 encoded format contains many data dependencies, which do not permit the encoded input to be decoded independently in parallel. Hence, threads need to synchronise and communicate in order to determine their decoding round at each striding block. This process needs to be repeated often (during each resolution round) and ends up being the significant cause of slowdown for the parallel implementation.



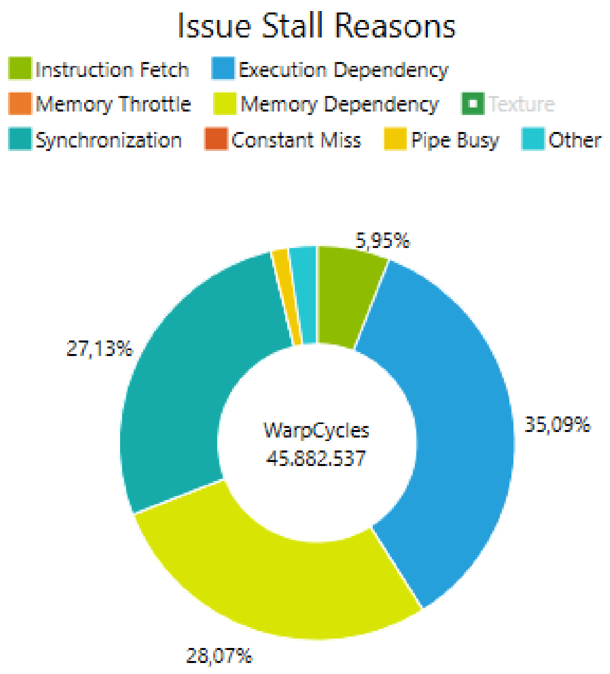
(a) DNA



(b) Bible



(c) Paper



(d) Random

Figure 6.4: Thread warp execution stalls for a stride of 256

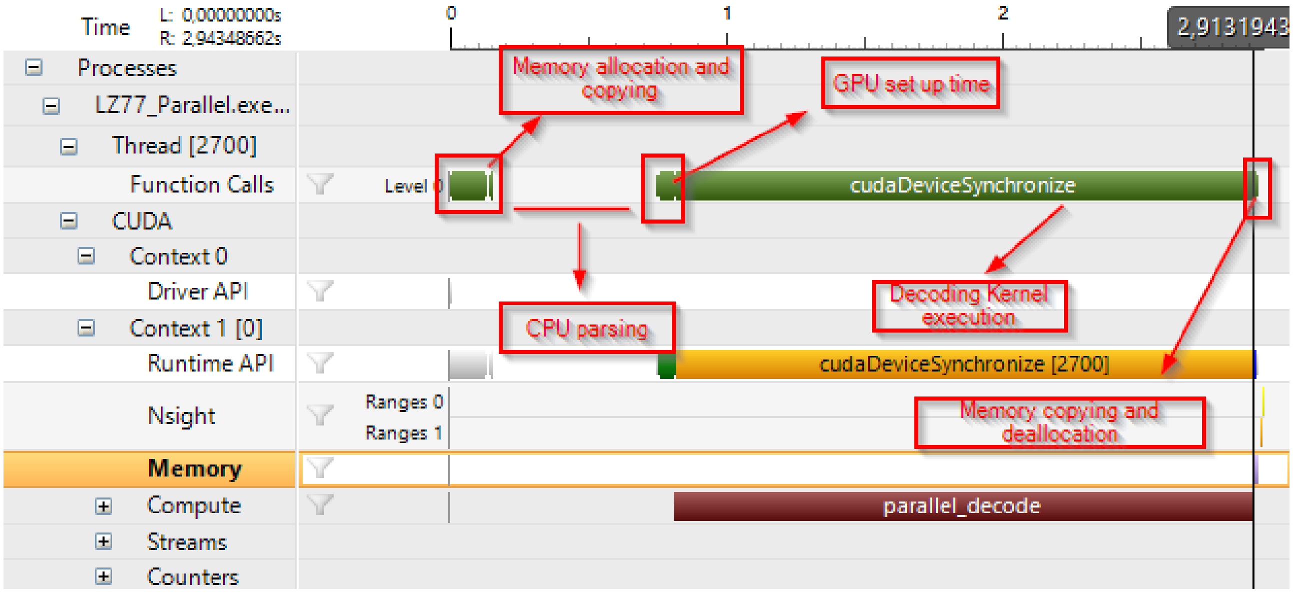


Figure 6.5: Timeline of GPU execution for the DNA data set

Chapter 7

Conclusion

We attempted to implement a parallelised version of the LZ77 algorithm and compare its running time with a serialised version, while also investigating the thread synchronisation cost and memory transfer times, that are typical bottlenecks of GPGPU computing. The conclusion we arrive at is that, due to the dependencies between the current output and the recently decoded output, the algorithm is a bad candidate for parallelisation, and is more suitable for a serial approach: The potential gains in performance and the relatively-constant memory transfer time are overshadowed by slow memory access times and moderate synchronisation costs, at least for the in-memory based synchronisation attempted in this paper.

As a final word, it seems unlikely that an efficient and fast parallel LZ77 algorithm can be derived without considering the underlying hardware architecture, or using a different synchronisation method, which is essential due to the data dependencies of the compression format. LZ77 decompression was designed to be executed sequentially, and this underlying assumption is expressed clearly when we consider the memory-intensive operations that are required to synchronise the threads at each decompression round.

7.1 Limitations

We only considered in-memory synchronisation in this paper, and we ignored any potential language-specific tools that could resolve some of the issues of our parallel implementation discussed above. For example, Sitaridi et al. [30] were able to achieve a parallel speedup by using warp functions that allow threads to synchronise with minimal memory accesses.

Furthermore, we did not properly investigate the effect of the maximum backreference depth on the overall process. In particular, most of the data sets are not random, but contain human text which is usually repetitive. Determining the connection between depth and decoding time would be useful in deciding whether or not the parallel decoding algorithm could efficiently be used on other data formats which exhibit less redundancy.

Finally, we only considered the prospect of improving the algorithm by executing it on a GPU in parallel, with a single framework. An approach with multiple CPUs or implementations in different parallel programming languages would provide a clearer view of the big picture, and allow us to draw specific conclusions.

7.2 Future work and extensions

As a first step, we would suggest eliminating the performance degradation in the presented parallel algorithm and investigating if the number of memory accesses and barriers that stall threads could be reduced. This could potentially result in an improved version of the resolution algorithm presented in this paper.

We also believe that investigating bit-level decompression would be something worth pursuing. Using bit strings could help improving the instruction-level parallelism of the GPU implementation.

Furthermore, we would advise on implementing the algorithm in different parallel architectures, and comparing them. We've only used CUDA in this paper, and there is a whole world of frameworks out there that may or may not be more appropriate for this type of problem. Some ideas for improvements include utilising more of the GPU (we only use a single block in this algorithm), attempting to overlap the operations of transferring memory and decoding (this could impact the performance for smaller files), or breaking down execution to multiple kernels (instad of just one), with the CPU acting as a synchroniser.

Finally, we suggest exploring the parallel compression aspect of this problem. In contrast to the decoding algorithm, encoding does not include any data dependencies and in theory, every character in the sequence could be encoded individually. Thus, we could still partially parallelise the LZ77 scheme, by focusing on compression instead of decompression.

Bibliography

[1] Khalid Sayood. *Introduction to Data Compression*, chapter 1, page 1. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edition, 2000.

[2] Joseph Lee. *Huffman Data Compression.* page 3., MIT, May 2007.

[3] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, 38(8):391 – 407, 2012. Available at `http://www.sciencedirect.com/science/article/pii/S0167819111001335`. Accessed: 29/03/2017.

[4] FluiDyna GmbH. What is GPU computing? no date. Available at `http://www.fluidyna.com/content/what-gpu-computing`. Accessed: 14/5/2017.

[5] Chris Paciorek. An Introduction to Using GPUs for Computation. 2014. Available at `http://www.stat.berkeley.edu/scf/paciorek-gpuWorkshop.html`. Accessed: 14/5/2017.

[6] Paul Visschers. *Reducing the overhead of data transfer in data-parallel programs.* page 3., Utrecht University, March 2012.

[7] Wu-chun Feng and Shucai Xiao. *To GPU Synchronize or Not GPU Synchronize?* page 1., Virginia Tech, May 2010.

[8] David Salomon. *Data Compression: The Complete Reference*, chapter 3, pages 230–231. Springer Science & Business Media, 2007. Available at `https://books.google.com.au/books?id=ujnQogzx_2EC&pg=PA230&lpg=PA230&dq=Deflate+popular&source=bl&ots=`

FpqttK6qmQ&sig=R1RGpu0MORJsJ0zKKSTapwmBaVQ&hl=el&sa=X&ved=0ahUKEwj86ej4_ _jSAhWqsVQKHbfcBQYQ6AEIRzAH#v=onepage&q=Deflate%20popular&f=false. Accessed: 28/3/2017.

[9] Mark Nelson and Jean-loup Gailley. *The Data Compression Book*, chapter 1, page 1. University of Bahrain, no date. Available at http://bit.ly/2mIotu2. Accessed: 28/3/2017.

[10] S. Wolfram. *A New Kind of Science*, chapter 10, page 1069. Wolfram Media, 2002, 2002. Available at https://www.wolframscience.com/reference/notes/1069b. Accessed: 28/3/2017.

[11] Khalid Sayood. *Introduction to Data Compression*, chapter 1, pages 3–6. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edition, 2000.

[12] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theor.*, 23(3):337–343, September 2006.

[13] Sebastian Deorowicz. *Universal lossless data compression algorithms.* page 3., Silesian University of Technology, 2003.

[14] LZ77 Compression Algorithm. Available at https://msdn.microsoft.com/en-us/library/ee441602.aspx. Accessed: 28/3/2017.

[15] Paul Brittan. Evaluating lossless data compression algorithms for use on mobile devices. Available at https://people.cs.uct.ac.za/~pbrittan/privacy_brittan_petzer/downloads/paul_litReview.pdf. Accessed: 30/3/2017.

[16] Jens Muller. Data Compression LZ77. *Universitt Stuttgart*, 2008. Available at http://jens.jm-s.de/comp/LZ77-JensMueller.pdf. Accessed: 30/3/2017.

[17] Lempel-Ziv universal data compression. Available at http://www.ece.northwestern.edu/~rberry/ECE428/Lectures/lec8_notes.pdf. Accessed: 28/3/2017.

[18] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses. Technical report, October 2006.

[19] David Davidović. The End of Fixed-Function Rendering Pipelines (and How to Move On). *gamedevelopment.tutsplus.com*, 2014. Available at `https://gamedevelopment.tutsplus.com/articles/the-end-of-fixed-function-rendering-pipelines-and-how-to-move-on--cms-21469`. Accessed: 17/5/2017.

[20] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370 – 1380, 2008.

[21] Derek Wilson Anand Lal Shimpi. NVIDIA's GeForce 8800 (G80): GPUs Re-architected for DirectX 10. *ANDANDTECH*, 2006. Available at `http://www.anandtech.com/show/2116/8`. Accessed: 30/3/2017.

[22] Mark Harris. GPGPU developer resources. *GPGPU.org*, 2011. Available at `http://gpgpu.org/developer`. Accessed: 30/3/2017.

[23] NVIDIA corporation. CUDA Parallel Computing Platform. *http://www.nvidia.com*, no date. Available at `http://www.nvidia.com/object/cuda_home_new.html`. Accessed: 30/3/2017.

[24] Cyrill Zeller. CUDA C/C++ Basics. *http://www.nvidia.com*, 2011. available at `http://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf` Accessed: 2/4/2017.

[25] Martin Farach and S. Muthukrishnan. Optimal parallel dictionary matching and compression (extended abstract). pages 244–253, 1995.

[26] Joseph F. JaJa. *PRAM (Parallel Random Access Machines)*, pages 1608–1615. Springer US, Boston, MA, 2011.

[27] S. De Agostino. Almost work-optimal PRAM EREW decoders of LZ compressed text. In *Data Compression Conference, 2003. Proceedings. DCC 2003*, pages 422–424, March 2003.

[28] Shunji Funasaka, Koji Nakano, and Yasuaki Ito. *A Parallel Algorithm for LZW Decompression, with GPU Implementation*, pages 228–237. Springer International Publishing, Cham, 2016.

[29] A. Ozsoy and M. Swany. CULZSS: LZSS Lossless Data Compression on CUDA. In *2011 IEEE International Conference on Cluster Computing*, pages 403–411, Sept 2011.

[30] Evangelia A. Sitaridi, René Müller, Tim Kaldewey, Guy M. Lohman, and Kenneth A. Ross. Massively-Parallel Lossless Data Decompression. *CoRR*, abs/1606.00519, 2016.

[31] NVIDIA corporation. Unified Memory in CUDA 6. *http://www.nvidia.com*, 2013. Available at `https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/`. Accessed: 5/5/2017.

[32] NVIDIA corporation. NVIDIA Nsight. *http://www.nvidia.com*, 2013. Available at `http://www.nvidia.com/object/nsight.html`. Accessed: 5/5/2017.

[33] Mark Harris. CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops. *http://www.nvidia.com*, 2013. Available at `https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/`. Accessed: 5/5/2017.

[34] NVIDIA Corporation. CUDA FAQ. *http://www.nvidia.com*, no date. Available at `https://developer.nvidia.com/cuda-faq`. Accessed: 5/5/2017.

[35] NVIDIA Corporation. NSIGHT Visual Studio Guide. *http://www.nvidia.com*, no date. Available at `http://docs.nvidia.com/nsight-visual-studio-edition/5.2/Nsight_Visual_Studio_Edition_User_Guide.htm#Analysis/Report/CudaExperiments/KernelLevel/IssueEfficiency.htm#IssueStallReasons`. Accessed: 12/5/2017.

[36] Data Compression LZ77. Available at `http://www.stringology.org/DataCompression/lz77/index_en.html`. Accessed: 24/5/2017.

Chapter 8

Appendix

8.A Study Contract

The final project description and the version of the study contract when it was signed can be found in the next page.

Australian National University

INDEPENDENT STUDY CONTRACT

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

SECTION A (Students and Supervisors)

UniID: u5587023

SURNAME: Morfiadakis FIRST NAMES: Emmanuel

PROJECT SUPERVISOR (*may be external*): Dr Eric McCreath

COURSE SUPERVISOR (*a RSCS academic*): ______________________________

COURSE CODE, TITLE AND UNIT: COMP4560, Advanced Computing Project, 12 units

SEMESTER [✓ S1] S2 YEAR: 2017 (12u)

PROJECT TITLE:

Parallel LZ77 decoding using a GPU

LEARNING OBJECTIVES:

In addition to the general COMP4560 learning goals:

The student would gain a good understanding of LZ77 compression and GPU software development, with a focus on looking at performance and scaling relating to the decoding. More generally, the project would strengthen the programming and problem solving abilities along with research skills associated with exploring approaches and ideas and then implementing, testing and evaluating these approaches and ideas.

Also it is expected that the student would gain general skills relating to: writing a report, and giving a seminar.

Australian
National
University

PROJECT DESCRIPTION:
The project will explore the LZ77 compression approach. This approach is used within DEFLATE which is used in gzip and png. LZ77 decoding works by copying previous subsection of a stream into the current output. This saves space as only a size and a offset is stored.   However, it also makes parallel decoding more challenging because of the data dependency within the main decoding loop.   A parallel approach exists which first determines the location for the decoded sequences and then determines the final decoding by iteratively forming the final result.   This enables the parallel implementation to be done in O(lg n) steps, assuming n is the length of the decoded output and also we have n processes.   Whereas the serial implementation is O(n).   This project will involve creating a framework for evaluating different implementations and then implementing both single CPU serial implementation along with a GPU parallel implementation. The performance and the performance bottlenecks will be evaluated for the proposed approach, in particular the analyses of memory transfer time along with synchronization costs will be evaluated.

The project report will contain:
+ An introduction to the topic.
+ A background section which describes the LZ77 format,
+ A section which provides a background to GPU computing.
+ A description of the algorithm for decoding
+ A description of the implementations.
+ Experimental chapter which: describes the hardware used for evaluation, the experiments done, and the results tabulated/graphed.
+ Conclusion/discussion/limitations/future work chapter.

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: Research Report (e.g. research report, software description...)	50%	May 26th	
Artefact: name kind: Software (Parallel and Serial Implementation) (e.g. software, user interface, robot...)	40%	May 26th	
Presentation:	10%	May 22nd	

MEETING DATES (IF KNOWN):
Weekly during the semester.

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:
+ Most of the development can be done on the students desktop, if not access to server machine with a GPU card would be made available.

SECTION C (Course coordinator approval)

…………………………………………………..	………………………..
Signature	Date

SECTION D (Projects coordinator approval)

…………………………………………………..	………………………..
Signature	Date

8.B Description of software

Code written by the author of this report:

`LZ77_Serial/LZ77_decode.c`

`LZ77_Serial/LZ77_decode.h`

`LZ77_Serial/LZ77_encode.c`

`LZ77_Serial/LZ77_encode.h`

`LZ77_Serial/decoder.c`

`LZ77_Serial/encoder.c`

`LZ77_Serial/Makefile`

`LZ77_Parallel/LZ77_decode.cu`

`LZ77_Parallel/LZ77_decode.cuh`

`LZ77_Parallel/decoder.cu`

`LZ77_Parallel/Makefile`

Code that was taken from other sources:

`LZ77_Serial/getopt.h`

`LZ77_Parallel/getopt.h`

All third party code uses are acknowledged in the `Originality.txt` files in the two subdirectories of the software artefact.

Testing for correctness was done with data sets, as explained in Chapter 4 of this report. These are submitted as part of the software artifact and are included in the file “Encoded Test Data.zip”. The same data sets were used in measuring the running times of the two algorithms presented in this report. The helper program used to generate the test data is also provided as part of the code artefact.

The experimental evaluation was carried out on a medium-range laptop computer, with an Intel Core i7 4510U running at 2 GHz and 4 GB of RAM and an NVIDIA GeForce 840M GPU, clocked at 1.029 GHz and with 2GB RAM.

Compilers include nvcc version 8.0, release 8.0.60 and Visual C++ Redistributable v.2015, as part of Visual Studio 2015.

8.C Software README

Due to the size and the formatting of the README file, it is not attached as an appendix. Instead, refer to the software artefact file `README.txt`, which contains information and examples on how to use the submitted software. A markdown version of these instructions is also provided in the `README.md` file.

8.D LZ77 Encoding Pseudocode

```
Data: String representation of data
Result: LZ77 encoding of the input
fill the lookahead buffer;
while lookahead buffer is not empty do
    p := longest prefix of the lookahead buffer in the sliding window;
    i := position of p in window relative to the lookahead buffer;
    k := length of p;
    X := first char after p in the lookahead buffer;
    output(i, k, X);
    move the first k characters from the buffer to the window;
    add the next k + 1 characters to the buffer
end
```

Algorithm 3: LZ77 Encoding process [36]

8.E Sample CUDA program

```
// nvcc -o addonegpu01 addonegpu01.cu

#include <stdio.h>
#include <cuda.h>
#define SIZE 1000000

// Kernel that executes on the GPU
__global__ void addmill(float *a, int N, int off)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x + off;
    int i;
    for (i=0; i<SIZE; i++) {
        a[idx] = a[idx] + 1.0f;
    }
}

// main routine that executes on the host
int main(void)
{
   printf("Starting!\n");

   float *a_h, *a_d;  // Pointer to host & device arrays
   const int N = SIZE;  // Number of elements in arrays
   size_t size = N * sizeof(float);
   a_h = (float *)malloc(size);         // Allocate array on host
   cudaMalloc((void **) &a_d, size);    // Allocate array on device

   // Initialize host array and copy it to CUDA device
   for (int i=0; i<N; i++) a_h[i] = (float)0.0;
```

```
  cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);

  // Do calculation on device:
  int block_size = 100;
  int n_blocks = 100;
  for(int i=0;i<SIZE/(block_size*n_blocks); i++) {
        addmill <<< n_blocks, block_size >>> (a_d, N, i*block_size*n_blocks);
  }

  // Retrieve result from device and store it in host array
  cudaMemcpy(a_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);

  // Print and check results
  for (int i=0; i<4; i++) printf("%d %f; ", i, a_h[i]);
  int okay = 1;
  double e = 0.1;
  for (int i=0; i<N; i++) okay = okay && (a_h[i] > (SIZE - e))
       && (a_h[i] < (SIZE + e));
  printf(": check : %s\n",(okay?"Okay":"Error"));

  // Cleanup
  free(a_h);
  cudaFree(a_d);
}
```