OPTIMIZING LEMPEL-ZIV FACTORIZATION FOR THE GPU ARCHITECTURE

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Bryan Ching

June 2014

COMMITTEE MEMBERSHIP

TITLE:	Optimizing Lempel-Ziv Factorization for the GPU Architecture
AUTHOR:	Bryan Ching
DATE SUBMITTED:	June 2014
COMMITTEE CHAIR:	Professor Chris Lupo, Ph.D., Department of Computer Science
COMMITTEE MEMBER:	Professor John Seng, Ph.D., Department of Computer Science
COMMITTEE MEMBER:	Professor Zachary N J Peterson, Ph.D., Department of Computer Science

ABSTRACT

Optimizing Lempel-Ziv Factorization for the GPU Architecture

Bryan Ching

Lossless data compression is used to reduce storage requirements, allowing for the relief of I/O channels and better utilization of bandwidth. The Lempel-Ziv lossless compression algorithms form the basis for many of the most commonly used compression schemes. General purpose computing on graphic processing units (GPGPUs) allows us to take advantage of the massively parallel nature of GPUs for computations other that their original purpose of rendering graphics. Our work targets the use of GPUs for general lossless data compression. Specifically, we developed and ported an algorithm that constructs the Lempel-Ziv factorization directly on the GPU. Our implementation bypasses the sequential nature of the LZ factorization and attempts to compute the factorization in parallel. By breaking down the LZ factorization into what we call the PLZ, we are able to outperform the fastest serial CPU implementations by up to 24x and perform comparatively to a parallel multicore CPU implementation. To achieve these speeds, our implementation outputted LZ factorizations that were on average only 0.01 percent greater than the optimal solution that what could be computed sequentially.

We are also able to reevaluate the fastest GPU suffix array construction algorithm, which is needed to compute the LZ factorization. We are able to find speedups of up to 5x over the fastest CPU implementations.

ACKNOWLEDGMENTS

Thanks to:

- My advisor Chris Lupo, for all of his guidance and patience and for introducing me to the GPGPU.
- My parents, my family, and everyone who has supported me to where I am today.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Lossless data compression has the ability to reduce storage requirements, while still maintaining the integrity of the original data. Several advantages can be gained by reducing the size of data, including the relief of transfer across I/O channels. Compression algorithms have a trade-off, in that they require an additional computation to be done on the original data before a compressed version can be used. This can be computationally expensive and the cost to compress might require too much processing or time. In many cases and applications, the increase of bandwidth rates outweighs any other consideration, but the increase in compression speeds would generally be helpful. This work takes a look into speeding up those compression speeds by performing the compression directly on a GPU, a graphics processing unit.

Applications and algorithms are beginning to be developed and ported to utilize the relatively new general purpose computing (GPGPU) aspect of GPU technology. GPGPUs allow applications to run computations unrelated to graphics, while allowing for the exploitation of the massively parallel nature of GPUs. GPGPUs are becoming increasingly popular for high performance computing, and are often utilized in large clusters. GPGPUs are typically used as coprocessors, assisting the CPU by performing tasks assigned to it. Typically, all desktop computers have some form of GPU; often, they are not always being used or are underutilized. Developing an application that can run on the GPU allows us to

make use of this underutilized hardware. Being faster than a CPU implementation is just an added bonus.

Our contribution is a Lempel-Ziv factorization, a lossless compression algorithm, implementation that runs directly on the GPU. The first step of the factorization is the construction of a suffix array. We reimplemented and reevaluated the GPU suffix array construction algorithm by Deo and Keely [6]. Breaking down the problem so that it can be done in parallel, we were able to find a solution that found speedups of 12-18x over the fastest single threaded CPU solution, while only increasing the final compressed output by 0.01 percent.

This thesis is organized as follows. Chapter 2 defines GPU technology and the Lempel-Ziv factorization problem. Chapter 3 describes our GPU implementation of Lempel-Ziv factorization. Chapter 4 describes our evaluation and results. Chapter 5 presents our conclusions. Finally, Chapter 6 discusses further steps that could be taken.

CHAPTER 2

Background

2.1 GPU Architecture

2.1.1 CUDA

NVIDIA's Compute Unified Device Architecture (CUDA) is the dominant proprietary framework used to access and control NVIDIA GPUs. CUDA provides the toolchain required to create applications that run on the GPU. As noted earlier, GPUs are coprocessors to CPUs; CPUs must give the instructions to launch a GPU kernel, the body of code that each thread on the GPU runs. The CUDA engineer writes these kernels to be launched on the GPU.

There are various parameters that one can control when writing and launching a GPU kernel. The thread model and memory model are the most common of these parameters. The thread model defines how many and in what configuration a kernel launches its threads. There are four layers in the CUDA thread model: the grid, the block, the thread, and the warp. The grid is composed of all the CUDA threads, grouped into blocks. Various limits are imposed on the dimensions and sizes of these grids and blocks. These limits depend on the compute capability of the GPU, which closely coincides with the microarchitecture of the GPU. When launching a kernel, the engineer specifies exactly how many blocks should be launched and how many threads make up each block. Uncontrolled by

the engineer, the warp is a group of threads that execute in lockstep.

The memory model defines the different memory types available to the layers of the thread model. The first and largest is the global memory. It is the slowest of all the memory types but is accessible from all of the threads. Global memory is usually the advertised GPU memory size. Next is shared memory, which any thread in a block can access. Shared memory is much faster than global memory, but is limited in size per block. Lastly, registers are the fastest but are limited to an individual thread. These limits are also dependant on the compute capability of the GPU and the binary. The CUDA engineer usually tries to reduce the number of accesses to global memory and make use of the faster shared memory and registers.

2.1.2 Libraries and Parallel Primitives

A variety of libraries make development on CUDA more streamlined. Some provide a fast solution to a particular problem. Others provide layers of abstraction to hide the complexity of CUDA programming. This includes the transfer of memory and the thread model. In our implementation, we try and use libraries whenever possible. First, these libraries have been developed over a long time by people who are more familiar with the architecture and advanced optimization techniques. Second, abstraction allows a problem to be continually optimized, while presenting a common API to use. This allows us to somewhat future-proof our implementation, since the libraries should be updated in the future against newer CUDA versions and hardware. Lastly, the purpose of many libraries are to provide solutions to parallel primitives.

Parallel primitives change the way a programmer looks at implementing a

parallel algorithm on the GPU. Instead of having to create a totally new algorithm specific for the GPU, one can change their program to be a collection of parallel primitives. These parallel primitives are common operations that we see in parallel algorithms across any architecture. Some very well known operations are scans, like prefix sum, or reductions, such as finding the minimum or sum of an array. Although these primitives may seem simple at first, there are many optimizations used by these libraries to provide speed up. Many of these are CUDA specific and a beginner to intermediate CUDA engineer are likely to not know them.

One of the most widely used CUDA libraries is the Thrust [12] library, providing device-wide primitives. One of the key features of the Thrust library is the interoperability of different architectures and technologies (CUDA, OpenMP, TBB). Although this may be nice for portability, we decided to avoid the use of Thrust and use the more CUDA-specific CUB library [17]. CUB provides abstractions at all three layers of the CUDA thread model, the device, the block, and the warp. CUB is more aware of CUDA features. That said, many of the algorithms are shared between CUB and Thrust. We decided to choose CUB because it is higher performing than Thrust [17] and contains specific features unavailable in Thrust, like the primitives that work on the block and warp levels. Some of the primitives that we use in our implementation include an inclusive sum, device select, radix sort, and block reduce.

Another interesting library that we make use of is Modern GPU, MGPU [3]. More specifically we make use of its algorithm for merge sorting, as we will see in our implementation in Section 3.1. One of the primary concerns when parallel programming is how to load balance a problem across the threads. MGPU makes use of a technique called Merge Path. A merge sort takes in two sorted arrays, A

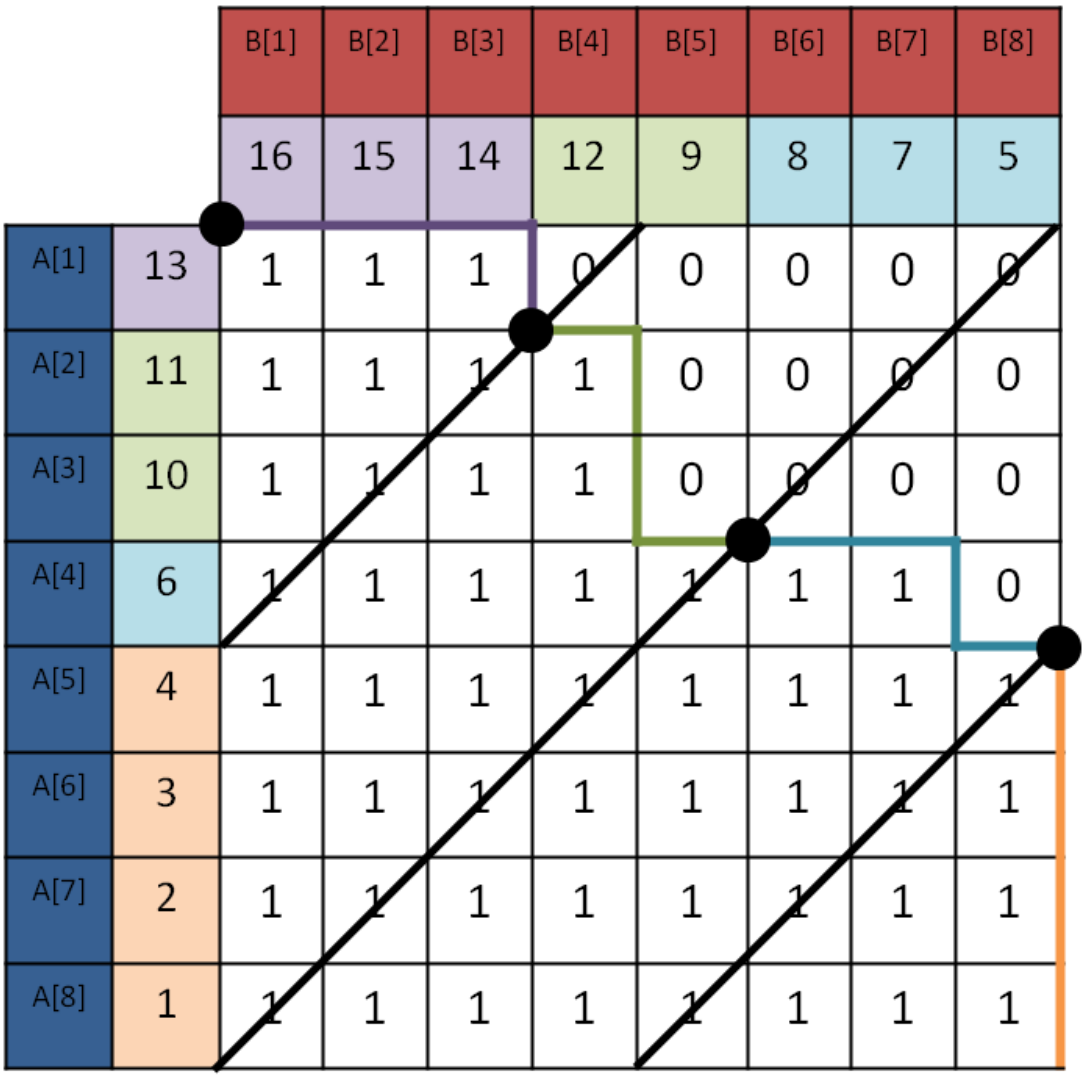

Figure 2.1: Merge Path partitioning scheme. Figure taken from [19]

and B, and outputs a final larger sorted array C with elements from both of the original arrays. First, let us imagine a merge matrix, a matrix where each row corresponds to an element from A and each column represents an element from B. If we were to try and run the merge sort sequentially, we'd start at the top left corner, and traverse the grid, moving right or down, depending on if A is greater than B at a specific element. Merge Path realizes that a path of traversal, the so-called merge path, is formed through the merge matrix during a merge sort.

Merge Path runs diagonals through the merge matrix and finds their intersection with the merge path. All elements on the diagonal before the intersection have a less than relationship. We can use this fact to quickly find the intersection using a binary sort. Using the intersections of the merge path, we now know which groups of elements from A and B are responsible for specific sections of C. If we use even divisions, we can then partition the work in a way that each thread is responsible for a constant number of elements from C and know which elements from A and B are required. Figure 2.1 shows an example merge matrix with a merge path and divisions. We refer you to the original paper of [19] for a more in-depth description of merge path.

There are some caveats when using these libraries. The first is that they provide an additional dependency to your application. Sometimes these libraries can be cumbersome to install on the host system. The next is that it is feasible to write a higher performing code with specific knowledge of the application. For example, knowing that part of the input array always appears in certain positions in the merged output could be utilized by the programmer. We opt for a more general approach at the cost of some potential performance improvement.

2.2 Compression

Compression, or more specifically data compression, is a field of computer science that is rich with applications. Compression allows us to reduce the size of the original data while still representing that original data. Compression techniques can be categorized as either lossless or lossy. Lossless compression tries to find repetitive or redundant patterns, while preserving all data allowing us to go back and forth from a compressed state to an uncompressed state without any data

loss. Lossless compression is often used to archive files but is also seen in many other fields, like genetics and executables. On the other hand, lossy compression tries to remove nonessential data from the source, often in a way that a human cannot even notice. In turn, this allows lossy compression to compress more than any lossless compression can since it is subjective what you can or cannot notice and how much is acceptable. Lossy compression algorithms, in contrast to lossless compression algorithms, do not preserve the original data, which now cannot be recreated. Some of the more common lossy compression algorithms are used as codecs to reduce video or audio sizes or as graphics formats. This includes household names like mp3 or jpeg. The focus of our work is on the Lempel-Ziv factorization, a lossless compression algorithm.

Let's first take a step back and discuss some terminology involved in evaluating a compression algorithm. The ratio between the sizes of the original uncompressed file and the compressed file is referred to as the *compression ratio*. The compression ratio tells us how much smaller the file has become after compressing. A high compression ratio indicates that the compressed file is much smaller than the original. Next, the *compression speed* is how long it takes for a compression algorithm to run. There is often a trade-off between the compression ratio and compression speed. Usually, having a faster compression algorithm may result in or is caused by having a smaller compression ratio. This works both ways. It is important to note that different users might have different requirements, leading them to choose one of these properties as more important than the other.

2.2.1 Lempel-Ziv

The seminal work on the Lempel-Ziv lossless compression algorithms is the original paper authored by Lempel and Ziv in 1977 [26]. Their work, LZ77, built

their final compressed output, called the LZ77 LZ factorization, by matching with previously parsed input. LZ77 is used in combination with Huffman coding to form the popular DEFLATE algorithm [7]. The DEFLATE algorithm is widely implemented and used; some of the most popular implementations include gzip, 7zip, zip, and the PNG file format. Lempel and Ziv later modified their algorithm into LZ78 which instead provided an explicit dictionary that could be used for random lookup decompression [27]. LZ77 and LZ78 would become the basis for a whole family of lossless data compression algorithms. Welch's work, LZW, improved the space efficiency of LZ78 by removing redundant characters and introducing variable encoding [25]. LZW is used today in Unix compress and in the GIF image format. LZSS improves on LZ77 by ensuring that the references replacing redundant symbols are indeed shorter than what they are replacing [24]. Today, it is used in popular archivers such as PKZip and RAR. Other variants, including LZMA and LZSS, change some aspect of the original algorithm to increase compression speed or the compression ratio. The focus of this thesis is on LZ77, and its LZ factorization.

Practical implementations of LZ77 use a sliding window buffer, a section of the overall input, where the algorithm operates on a longer factored prefix and a short unfactored suffix. In practice, using the sliding window produces comparable compression ratios while greatly increasing compression speed. We decide to calculate the LZ factorization of the entire string instead of just a window. In actuality, our final implementation is a compromise between these two. Throughout the paper, we will use the terms, LZ77 LZ factorization, LZ factorization, and ideal LZ factorization. The LZ77 LZ factorization and ideal LZ factorization refer to the LZ factorization of the whole string. In almost all cases, the term LZ factorization will also refer to the ideal LZ factorization. We

will now describe and define the LZ factorization of a string.

2.2.2 Lempel-Ziv Factorization

i	S[i]	nsv-lex	nsv-match	psv-lex	psv-match	LPF[i]	PrevOcc[i]	LZ
0	a	-1	-	-1	0	-1	0	
1	b	-1	-	0	-	0	-1	1
2	b	1	b	0	-	1	1	2
3	a	0	a	-1	-	1	0	3
4	a	2	-	0	abb	3	0	4
5	b	-1	-	1	bb	2	1	-
6	b	1	bbaa	2	bb	4	1	-
7	b	2	baa	4	-	3	2	7
8	a	3	aa	-1	-	2	3	-
9	a	3	aab	8	aa	3	3	-
10	a	0	ab	3	a	2	0	10
11	b	6	b	2	ba	2	2	-
12	a	10	ab	3	a	2	10	12
13	b	7	b	4	-	1	7	-

Table 2.1: SA, ANSV, LPF, PrevOcc, and LZ for S = abbaabbbaaabab

As previously described, Lempel-Ziv compression, a lossless data compression algorithm, tries to find repeated occurrences. Compression occurs by replacing these repeated occurrences with a pair of numbers representing the location of the previous occurrence and the length of the match. LZ77 compression finds these repeated occurrences by using a greedy left-to-right parsing, called the LZ factorization. The LZ77 LZ factorization is the breaking down of the input string

into substrings, referred to as factors, where each factor is either a repeated occurrence or a completely new character. The LZ factorization builds these factors from left-to-right using the previously factored prefix to search for previous occurrences. Formally, the LZ factorization of a string $S[n]$ decomposes S into factors $S = w_0 w_1 \dots w_k$ where $k <= n$, where a factor w_i is either the longest prefix that appears before w_i in S or is a new character.

For example, the LZ factorization of string $S = abbaabbbaaabab$ has the factorization $a.b.b.a.abb.baa.ab.ab$, where $w_0 = a, w_1 = b, w_2 = b, w_3 = a, w_5 = abb, w_6 = baa, w_7 = ab, w_8 = ab$. If we were to naively solve the LZ factorization of the string S, we would start at the first suffix, $suf_0 = abbaabbbaaabab$, and try to find the longest prefix of suf_0 that appears previously. Since no prefix appears previously, the first factor is the first character of the suffix, $w_0 = a$. We then move on to the second suffix, $suf_1 = bbaabbbaaabab$. Again, no prefix appears previously and the first character is used again, $w_1 = b$. Now, we look at the third suffix, $suf_2 = baabbbaaabab$, and check if any prefix matches any of the previously factored string, ab. The third suffix does have prefix that matches previously. Only the first character b has appeared previously, so the next factor $w_2 = b$. Similarly, the next factor $w_3 = a$, because again only a appears previously. For the fifth factor, w_4, we need to look at the suffix, $suf_4 = abbbaaabab$, and the previously factored input, $abba$. This time, the prefix abb has appeared previously in $abba$, so the next factor is $w_4 = abb$. We can then move on to the suffix, $suf_7 = baaabab$, and continue until the whole string is factored.

The LZ factorization can be encoded simply using a sequence of pairs. One encoding scheme uses the pair with a position of previous occurrence and the length of the match or a character, if the length is 0. We can store the positions of previous occurrence into a Previous Occurrence (PrevOcc) array and

the length of the match into a Longest Previous Factor (LPF) array. The string abbaabbbaaabab encoded using this scheme would output:

(PrevOcc, LPF) = [(a, 0),(b, 0),(1, 1),(0, 1),(0, 1),(2, 3),(0, 3),(10, 2)]

Another encoding scheme uses the pair of original position, stored in the LZ array, and the position of the previous occurrence (LZ, PrevOcc).

(LZ, PrevOcc) = [(0, a),(1, b),(2, 1),(3, 0),(4, 0),(7, 2),(10, 0),(12, 10)]

Note that these two schemes produce the same number of pairs and each can be derived from the other. The goal of the LZ factorization algorithm is to fill these arrays. For the purpose of this thesis, we need not worry about the encoding scheme, as we will output all three arrays.

A naive LZ factorization can begin parsing the input from left-to-right filling the above arrays, by naively performing a string match with the prefix of the unfactored input and every position in the previously factored input. LZ factorization algorithms try and improve the efficiency of the LZ factorization by improving this string matching. Various LZ factorization algorithms have been compared experimentally in [1]. In general, efficient LZ factorization algorithms all make use of a few common data structures, the suffix array (SA) and the All Nearest Smaller Values (ANSV) arrays. The suffix array sorts every suffix of the input which the ANSV arrays can then utilize to reduce the number of positions that need to be checked for each factor. Tables 2.1 and 2.2 shows these structures for the string abbaabbbaaabab.

2.2.2.1 Suffix Array

The suffix array is a common data structure in string matching algorithms. The suffix array SA of S is a lexicographic ordering of integers of order n where each integer represents a suffix of S, so that $suf_{SA[0]} < suf_{SA[1]} < . \ . \ . \ suf_{SA[n-1]}$.

sa	suf-lex	nsv-lex	psv-lex
8	aaabab	3	-1
9	aabab	3	8
3	aabbbaaabab	0	-1
12	ab	10	3
10	abab	0	3
0	abbaabbbaaabab	-1	-1
4	abbbaaabab	2	0
13	b	7	4
7	baaabab	2	4
2	baabbbaaabab	1	0
11	bab	6	2
6	bbaaabab	1	2
1	bbaabbbaaabab	-1	0
5	bbbaaabab	-1	1

Table 2.2: Suffix array and ANSV in lexicographic order for the string abbaabbbaaabab

First introduced as a space efficient alternative to suffix trees, the suffix array can fully replace the suffix tree with the use of additional data structures, such as the LCP array. Suffix arrays can be used to quickly find and match strings in a dictionary. This ability has a wide variety of applications from string searches to data compression to bioinformatics.

There exist many suffix array construction algorithms (SACA). The skew algorithm [14] uses a divide and conquer approach to construct a partial suffix array to infer the rest of the positions. The pseudocode of the SACA that we

will use is presented in Figure 3.1. Essentially, the skew algorithm divides the suffixes into two groups. A suffix array is constructed using the larger group, which holds 2/3 of the suffixes. A quick check is used to see if these suffixes can be quickly sorted using their first three characters. If this sort does not create the suffix array, due to non-unique suffixes, then the algorithm recurses until the suffix array is constructed. The smaller group can then be sorted using inference and merged with the larger group. Running in linear time, the skew algorithm has also been studied in parallel. The fastest known construction of suffix arrays on the GPU by Deo and Keely [6] utilizes the skew algorithm. Inspired by most of their ideas, the work in this thesis is also a reimplementation and benchmark of their algorithm.

Table 2.2 shows the constructed suffix array for the string *abbaabbbaaabab*.

2.2.2.2 ANSV and LZ Factorization Calculation

Let us first formally define the LPF (Longest Previous Factor) and lcp (longest common prefix). The lcp of any two strings is the length of the common prefix between the two strings if any. For example, the $lcp(aaabab, aabab) = 2$. The LPF, longest previous factor, array holds the lengths of the longest previous factors at any position i. In other words, LPF[i] holds the maximum lcp of $\text{suf}_{\text{SA[i]}}$ and all suffixes less than i.

In our example,

$$LPF[2] = max(lcp(SA[2], SA[1]), lcp(SA[2], SA[0]))$$

.

A naive LZ factorization algorithm may work by calculating the LPF for every position, by calculating the lcp with every previous suffix. It can be seen that

```
1: procedure LazyLZFactorization(S, n, PSV, NSV)
2:     i ← 1
3:     while i ≤ n do
4:         a LZ factor starts here
5:         if lcp(i, PSV) ≥ lcp(i, NSV) then
6:             LZ Factor = (PSV,lcp(i,PSV))              ▷ Pair (PrevOcc,LPF)
7:         else
8:             LZ Factor = (NSV,lcp(i,NSV))
9:         end if
10:        if LPF = 0, PrevOcc = -1, and the character is inserted
11:        i ← i + max(LPF, 1)
12:    end while
13: end procedure
```

Figure 2.2: LZ Factorization Pseudocode

this naive algorithm runs in $O(n^3)$ time. This can be bounded to $O(n^2)$ time using the knowledge that the total length of all lcp's is N. In [4], the number of positions that a suffix needs to be compute lcp with is reduced to the PSV and NSV. The NSV, next smaller value, and PSV, previous smaller value, make up the ANSV, all nearest smaller values, problem. We only need to previous suffixes and only the closest ones since the suffix array is in lexicographic order. Computing the ANSV problem can be done linearly and sequentially using a stack-based algorithm found in [9]. By reducing the number of suffixes to compute lcp against to two, this observation now reduces the problem to an O(n) time complexity.

Table 2.1 shows the constructed suffix array and ANSV arrays (PSV,NSV) for the string abbaabbbaaabab. We can take a look at where the NSV and PSV values for suf_{10}=abab is found. Simply looking at the suffix array, we can see that the NSV of suf_{10} is suf_0, because 0 is the first smaller value after 10. Likewise, the PSV of suf_{10} is suf_3, because 12, the previous value, is greater than 10, but 3, which comes before 12, is smaller.

With these NSV and PSV arrays, a naive algorithm may try and calculate the LPF for every position. With the LPF array filled at every position, the LZ factorization can be quickly found [4]. More recent LZ factorization algorithms have decided to forgo this intermediate step and directly calculate the LZ factorization. If we examine Table 2.1, we can see that the LZ factorization of the string *abbaabbbaaabab* can be reduced to 8 positions. The positions where a factor do not begin, position 5 for example, does not need to calculate the LPF. It also does not need to calculate the ANSV, but that calculation is relatively inexpensive and may come from the generation of the other values anyway. Referred to as lazy LZ factorization in [13], the LPF value is only calculated at the start of a factor. Figure 2.2 shows the basic pseudocode for calculating the

LZ factorization given the PSV and NSV arrays using this lazy method. Again, we need only check the PSV and NSV positions for the lcp and take the larger of the two. Finally, we store the position of the larger value into the Previous Occurrence (PrevOcc) array.

2.3 Related Works

Lossless data compression on the GPU is a field that has yet to be fully investigated. Many lossless data compression algorithms are application specific.

Although few, there does exist work on porting general purpose lossless data compression algorithms to CUDA. CULZSS ports the LZSS algorithm, a sibling to LZ77, to the GPU with success [21]. The common optimization on these ports is the use of pipelining, where the input is broken up to be worked on individually, using CUDA streams, CUDA's ability to concurrently copy partial data and run kernels. Many of their original algorithms allow for this. Many of these applications also make use of a sliding window, as seen in most LZ77 implementations. Implementations using the sliding window do not know or make use of the whole input. This could lead to larger compressed files.

To the best of our knowledge, this is the first attempt to calculate the LZ factorization on the GPU. Although we will not be calculating the ideal LZ factorization, as described later, the knowledge of the whole input string is still utilized. The project by Shun and Zhao [23] is a multicore CPU parallel implementation of the LZ factorization. They provide one of the first and most recent parallel implementations of the LZ factorization, where many of the inspirations throughout our project and implementation derive from. In their project, they were able to show a O(n) work algorithm with significant speedups on a multicore

CPU. Most of our implementation matches their's, except on the GPU; however, we do not calculate the whole LPF string, and make use of the lazy LZ factorization technique described earlier. The cost to calculate the ideal LZ factorization in a parallel fashion, as they did, was too great for GPU. Calculating every LPF position was a very memory intensive task that we found took too long on the GPU due to the high memory latency. This was the primary cause to the usage of the PLZ, which we'll describe in Chapter 3.

CHAPTER 3

Implementation

Our implementation is structured in three main steps: the construction of the suffix array, the calculation of ANSV for every index, and finally the generation of the LZ factorization.

3.1 SA

The construction of the suffix array stays true to the algorithm used by Deo and Keely [6]. In Figure 3.1, we show the pseudocode used by Deo and Keely. One of the intermediate steps is to sort suffixes based on their first three characters. We used CUB's implementation of radix sort to facilitate this. In line 6, we need to check if the sorted triplets are unique. To accomplish this, we used a combination of small custom kernels and CUB primitives. Finally as mentioned in Section 2.1.2, we used the MGPU library to facilitate the merge sort.

3.2 ANSV

To calculate the needed ANSV values we used the parallel algorithm from Shun and Zhao [23].

The first step is to build a balanced binary tree, where the leaves are elements from SA, and the ancestors are the minima of their children. Although more

```
1: procedure COMPUTESA(s, sa, n)
2:     initMod12();    ▷ Kernel to set flags at 2/3. DeviceSelect to get s12,sa12
3:     radixSort(s12);                                          ▷ DeviceRadixSort
4:     radixSort(s12);
5:     radixSort(s12);
6:     lexicRankOfTriplets();  ▷ Custom kernel to check unique. Inclusive Sum
       to count. Custom kernel to get s12.
7:     if !allUniqueRanks then
8:         computeSA(s12, sa12);                                        ▷ Recursion
9:         storeUniqueRanks();                                          ▷ Kernel
10:    else
11:        computeSAFromUniqueRank();                                   ▷ Kernel
12:    end if
13:    radixSort(s0);
14:    mergeSort(s0, s12);                          ▷ Merge Path + Merge Sort
15: end procedure
```

Figure 3.1: Suffix Array Construction Pseudocode from [6]. Comments add details from our implementation.

efficient algorithms may exist, we decide to take a simpler naive approach and launch a kernel at each level. Each thread in the kernel calculates for a node the minimum of its two children and stores it into a 1d array. A 2d array would be easier to index into, but much more difficult to allocate. A 1d array has additional benefits of improved memory coalescing and better cache performance.

The suffix array is then divided into even divisions. Each thread uses a stack, in the form of an array, and traditionally solves ANSV for their division. Because

```
1: procedure ANSV
2:     for each level of MinTree do                    ▷ Bottom Up Construction
3:         MinTree(level);          ▷ Build level by calculating minima of children
4:     end for
5:     ANSVKernel(mintree, chunkSize)
6: end procedure
```

```
1: procedure ANSVKernel
2:     chunk ← threadID * chunkSize          ▷ Each thread gets a unique chunk
3:     ANSVLinear(chunk)
4:     if ( thenchunk detects no PSV/NSV)        ▷ ANSVLinear may be wrong
5:         checkMinTree(mintree)                    ▷ Manually check MinTree
6:     end if
7: end procedure
```

Figure 3.2: ANSV Pseudocode

each thread can only see their division, many of the positions will think there is no smaller position, while they may exist in the next or previous division. To compensate for this, each thread will manually check each position that did not find a smaller value using a search on the previously generated binary tree.

This algorithm will generate the ANSV arrays for each index, although not every index is needed in the final LZ factorization. We did experiment with the idea of solving the ANSV problem for a specific index only when needed, but found that in most cases, this was only a little faster or much slower than generating every ANSV value at once.

3.3 LZ Factorization

The final step is to calculate the LZ factorization.

At first, we attempted to follow the parallel algorithm of Shun and Zhao [23]. In their work, the LPF array is calculated for every position, and then the LZ factorization is solved using a parallel list ranking algorithm. We found that the calculation of the LPF array at every index to be too computationally expensive and wasteful, even on the GPU. Instead, our work will also employ the lazy LZ factorization, mentioned in [13]. The biggest problem with the lazy LZ factorization was that it is incredibly sequential. Since it is impossible to know what entries will exist in future points in the LZ factorization, it is a hard problem to parallelize.

3.3.1 PLZ

We propose breaking away from the ideal LZ factorization and using a Parallel LZ factorization (PLZ). Using PLZ, the string S will be broken into chunks of

i	S[i]	NSV[i]	PSV[i]	LPF[i]	LZ
0	a	-1	-1	0	0
1	b	-1	0	0	1
2	b	1	0	1	2
3	a	0	-1	1	3
4	a	2	0	3	4
5	b	-1	1		
6	b	1	2		
7	b	2	4	3	7
8	a	3	-1		
9	a	3	8		
10	a	0	3	2	10
11	b	6	2		
12	a	10	3	2	12
13	b	7	4		

Table 3.1: chunk size c=7, divisions d=2, LZ=8

size c to be worked on individually. Each thread will be assigned a chunk and traditionally calculate the LZ factorization on it. The LZ factorization calculated by each thread will be entered unmodified into the final LZ factorization. The main advantage of this is being able to parallelize the problem, while not incurring too many penalties on the compression ratio. By doing this, we are also able to limit the amount of work any one thread will do, in an attempt to load balance. There are several disadvantages that may appear, all of which depend on the original input string. There is a chance for the PLZ LZ factorization to be larger

than the ideal LZ factorization. There is also an unlikely chance for them to be exactly the same.

Recall that an entry into the final LZ factorization indicates the start of a factor. The ideal LZ factorization is a sequence of longest previous factors. When we use PLZ, we are breaking down the LZ factorization into chunks to be worked on in parallel. At the start of each chunk, we insert a first entry into the LZ factorization. We then continue calculating the LZ factorization using the traditional sequential algorithm. At the end of the chunk, we stop the string comparisons and cut off the current factor. By stopping the string comparisons, we are able to limit the amount of work needed to process a chunk. This also means that factors are limited in length to the chunk size.

Various scenarios may occur when using PLZ, which we will first explain and later show with example in Tables 3.1, 3.2, and 3.3. The first scenario occurs when that first entry is in the same position as an ideal LZ factorization factor. The LZ factorization of that chunk will then be the same as the ideal LZ factorization. If every first entry is in the same positions as an ideal LZ factorization, the final PLZ LZ factorization will be exactly the same as the ideal LZ factorization.

The next scenarios occur when a first entry is not in the same position as an ideal LZ factorization factor. This would happen when the chunk splits in the middle of a ideal LZ factorization factor. In these scenarios, the last factor in the previous chunk will no longer be the longest. The LPF of that last factor will be shorter than the ideal LZ factorization. When the LZ factorization is calculated on this chunk, the next factor may or may not start at an ideal LZ factorization factor. We then begin calculating the LZ factorization of that chunk, starting at that first entry. If any of the calculated factors begin at the start of the ideal LZ factorization factor, then all factors after will also match the ideal LZ

factorization. It is impossible to know a priori which of these scenarios will occur, since they are all dependent on the input string.

The next question to be answered is deciding the chunk size c. A larger chunk size could reduce the chances for a larger factorization and increase compression ratios. On the other hand, a smaller chunk size would more evenly distribute the work among the GPU threads, and in turn should increase compression speeds. This is a trade-off that should be left to the user. In our implementation, we have the option to define an arbitrary size for the chunk size c or for a number of divisions d of the input string. Some optimal sizes might be to use divisions that are multiples of the number of multiprocessors. In any case, it is impossible to predict the compression ratio, and different users will have different priorities.

The resulting PLZ LZ factorization when using PLZ with chunk sizes of 7, 4, and 3 can be seen in Tables 3.1, 3.2, and 3.3. These chunk sizes, from a string size $n = 14$, can result from divisions of $d = 2, 4, 5$. Table 2.1 can be used as reference with the whole LPF and PrevOcc arrays filled. First note that the LPF and PrevOcc arrays are not totally filled. As we are doing a lazy LZ factorization, not all values need to be computed, and this is shown accordingly. Next, notice that there are breaks within the table. These indicate chunks for a single thread, or block as we'll see soon, to work on. The next thing to notice are the bold elements in the LPF array. A bold element indicates that the value is no longer the LPF at that position and is changed from the reference LPF values. Recall that when using PLZ, the string matching stops at the end of a chunk.

In Table 3.1 with a chunk size of 7 and a division of 2, we can see that there are no changes in the LZ factorization. The split occurred at position $i = 7$, the beginning of a factor in the ideal LZ factorization. Therefore, we see no changes while being able to solve the problem in parallel.

Tables 3.2 and 3.3 begin to show changes from the ideal LZ factorization. In Table 3.2, we use a chunk size of $c = 4$ and a division of $d = 4$. Notice how the third chunk starts at position $i = 8$. Because the ideal LZ factorization did not have a factor starting at position 8, it can be determined that the PLZ LZ factorization is no longer the same as the LZ factorization. The LPF at position 8 is 2, so that the next factor starts at position 10. The ideal LZ factorization had a factor starting at position 10, so we are now back on track. Any additional factors calculated from this chunk should match that of the ideal LZ factorization. The resulting PLZ LZ factorization has a length of $l = 9$, 1 more than the ideal LZ factorization length of $l = 8$.

The last example in Table 3.3 shows a chunk size of $c = 3$ and a division of $d = 5$. Notice again that chunks 2 and 3, starting at positions 6 and 9 respectively, have a different factor from the ideal LZ factorization. A key thing to realize from this example is that the length is unchanged. Both the PLZ LZ factorization and the LZ factorization have a length, $l = 8$. By using PLZ, we were able to split the work into 5 to be worked on in parallel and compress the string to the same length as the ideal LZ factorization.

3.4 More LZ Factorization Optimizations

One thing that we have yet to cover is how we perform the string match. The naive operation is to do a character by character match until the prefix no longer matches with the LPF. One optimization that we have implemented is to instead do a parallel string comparison. A block of threads can load a group of characters into shared memory. The BlockLoad primitive from CUB is used to load a number of characters from the LPF and from the prefix into arrays for use in an individual

thread in the block. To simplify, we can imagine a block of 32 threads loading a chunk of 32 characters from the LPF and prefix. Each thread, responsible for a single index and two characters, then compares the two characters for a match. A match is assigned the block's dimensions (32), and a mismatch is assigned the thread's id (0-31). The threads can then cooperatively find the minimum in a block using reduction, with the BlockReduce primitive from CUB for example. If all 32 characters matched, the value 32 is returned to the first thread in the block, which controls all the logic. That thread will then set a flag to indicate to the rest of threads to continue with the comparisons. If there is a mismatch, the index of the mismatch is instead returned to the first thread, which can then stop the comparisons. Which implementation is faster is solely dependent on the data and the average factor length. Average factor lengths less than the number of characters loaded and compared in parallel may see faster speeds with just the naive comparisons. Our implementation will use the parallel string comparison for evaluation. In doing so, each chunk is worked on by a single block.

Finally because we are cutting off the matching at the end of the chunk, an optimization can be made to reduce the number of searches. Because the LPF can occur at either the PSV or the NSV, we usually need to check both and pick the longer. Reaching the edge of a chunk during the first search allows us to skip the second chunk.

The final LZ factorization, made up of relevant entries from the LPF and PrevOcc arrays, can be gathered by using CUB's DeviceSelect, which allows us to compact the arrays using a flag set at the start of each factor. This allows our final data transfer to require sending less data.

i	S[i]	NSV[i]	PSV[i]	LPF[i]	LZ
0	a	-1	-1	0	0
1	b	-1	0	0	1
2	b	1	0	1	2
3	a	0	-1	1	3
4	a	2	0	3	4
5	b	-1	1	-	-
6	b	1	2	-	-
7	b	2	4	3	7
8	a	3	-1	2	8
9	a	3	8	-	-
10	a	0	3	2	10
11	b	6	2	-	-
12	a	10	3	2	12
13	b	7	4	-	-

Table 3.2: chunk size c=4, divisions d=2, LZ=9

i	S[i]	NSV[i]	PSV[i]	PrevOcc[i]	LPF[i]	LZ
0	a	-1	-1	-1	0	0
1	b	-1	0	-1	0	1
2	b	1	0	1	1	2
3	a	0	-1	0	1	3
4	a	2	0	0	2	4
5	b	-1	1	-	-	-
6	b	1	2	1	3	6
7	b	2	4	-	-	-
8	a	3	-1	-	-	-
9	a	3	8	3	3	9
10	a	0	3	-	-	-
11	b	6	2	-	-	-
12	a	10	3	10	2	12
13	b	7	4	-	-	-

Table 3.3: chunk size c=3, divisions d=2 ,LZ=8

CHAPTER 4

Results

4.1 Experimental Setup

4.1.1 Test Machine

All measurements were gathered from a single machine with an NVIDIA Tesla K40c and NVIDIA GTX TITAN Black. The Tesla K40c and GTX TITAN Black are two of NVIDIA's higher end solutions. The GTX TITAN Black, which we'll now refer to as Black, has a 0.98 GHz GPU clock rate, 3.5 GHz memory clock rate, and 6 GB of memory. The Tesla K40c, now K40c, has a 0.88 GHz GPU clock rate, 3.0 GHz memory clock rate, and 12 GB of memory. The Black is faster than the K40c, but has significantly less memory. Both the CUDA runtime and driver version were 6.0. The binary was compiled using -O3 optimization and compute capability 2.0. We chose the more compatible compute capability 2.0 instead of 3.5, because we did not need any of the features of 3.5. Timings were recorded using the CUDA events API.

4.1.2 Data

Data for our evaluation was gathered from various publicly available datasets, often used in benchmarking lossless compression algorithms.

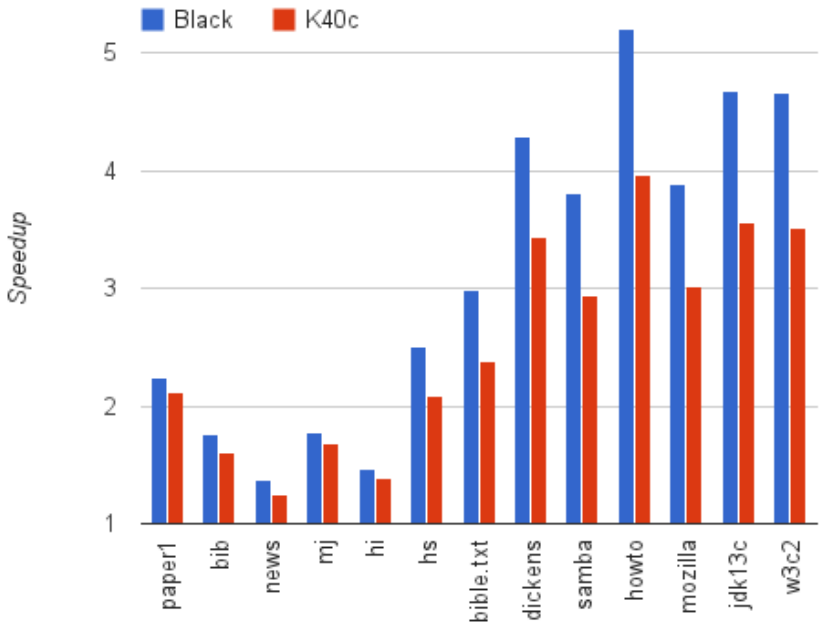

Figure 4.1: Speedup of suffix array construction on the GPU compared to the fastest CPU implementation

The evaluation of the suffix array is actually an evaluation of a reimplementation of the fastest known GPU suffix array construction algorithm (SACA) by Deo and Keely [6]. The benefits and applications of the suffix array has already been detailed in Section 3.1 Deo and Keely's evaluation was done on an AMD Radeon GPU using OpenCL. Our results on a NVIDIA GPU using CUDA and CUB primitives with ModernGPU's merge path method to mergesort are not expected to be significantly different. We will be comparing our results to a set of SACA benchmarks found on the wiki of LibDivSufSort, one of if not the

size(MB)	filename	Black(ms)	K40c(ms)	CPU(ms)	$\text{Black}_{\text{Speedup}}$	$\text{Black}_{\text{ms/B}}$
0.05	paper1	15.2	16	34	2.2	0.286
0.11	bib	19.3	21.2	34	1.8	0.173
0.36	news	33.6	36.8	46	1.4	0.089
0.43	mj	29.2	30.9	52	1.8	0.065
0.49	hi	34.2	36.1	50	1.5	0.067
3.14	hs	108.5	130.6	272	2.5	0.033
3.86	bible.txt	112.9	141.5	338	3.0	0.028
9.72	dickens	282.8	353.5	1212	4.3	0.028
20.61	samba	536.9	693	2042	3.8	0.025
37.60	howto	1021.1	1339.7	5320	5.2	0.026
48.85	mozilla	1278.2	1642.6	4958	3.9	0.025
66.50	jdk13c	1928.2	2525.4	9010	4.7	0.028
99.37	w3c2	2896	3840.8	13486	4.7	0.028

Table 4.1: Runtimes(ms), speedups, and ms/B of datasets for evaluation of suffix array construction

fastest CPU SACA implementation [18]. That benchmark compares the fastest CPU SACA implementations on a variety of test files. We will compare our GPU implementation against the fastest CPU time for each file. Files were picked to match closely with Deo and Keely's evaluation. GPU times include parsing the file, transferring the data both ways, and the construction of the suffix array.

Figure 4.1 and Table 4.1 presents the results comparing the CPU implementations to our GPU implementation. The first thing to note is that the CPU SACA benchmarks are significantly faster than those used by Deo and Keely. Our GPU implementation did not see the speedup of 35x that theirs did, but we

still found around a 4-5x speedup for most files for Black. We did not have their implementation or their raw result data to compare against. Loosely comparing with the charts in their paper though, we find that our GPU implementation is at least on par if not faster.

Another interesting metric is the runtime in microseconds per Byte seen in [1]. We found that after a certain point, our Black implementation was achieving rates of around 0.02 to 0.03 ms/B. In comparison, [1] found results between 0.1 to 0.4 ms/B.

Like many other GPU algorithms, we found that smaller files did not see the greater speedups that larger files did. The likely cause is that smaller files cannot fully saturate the GPU, and the cost of initialization and data transfer could not be hidden by increased computations. This indicates that the GPU is not the all around solution for faster suffix arrays and the size of the input needs to be considered.

Figure 4.2 shows a profile of the SACA of the GPU implementation. Kernels other than those involved in the merging or sorting take the greatest percentage of time in both examples. These kernels have the most room for improvement, since they are less likely to deal with primitives and more likely deal with the setup and movement of data. The CUDA grid and block sizes could have a greater factor in the speeds and further optimization are more likely to see gains here.

4.3 ANSV

he ANSV algorithm divides the suffix array into chunks for each thread. These threads will then individually solve the ANSV problem on their chunk and solve

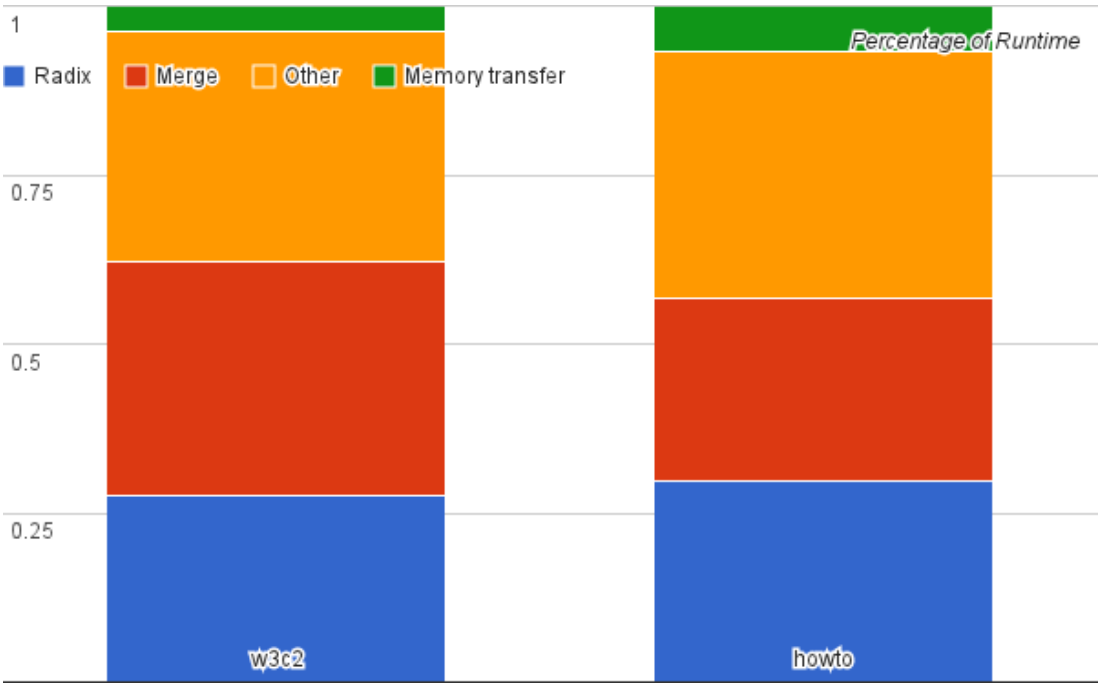

Figure 4.2: Profile of Suffix Array construction on the GPU

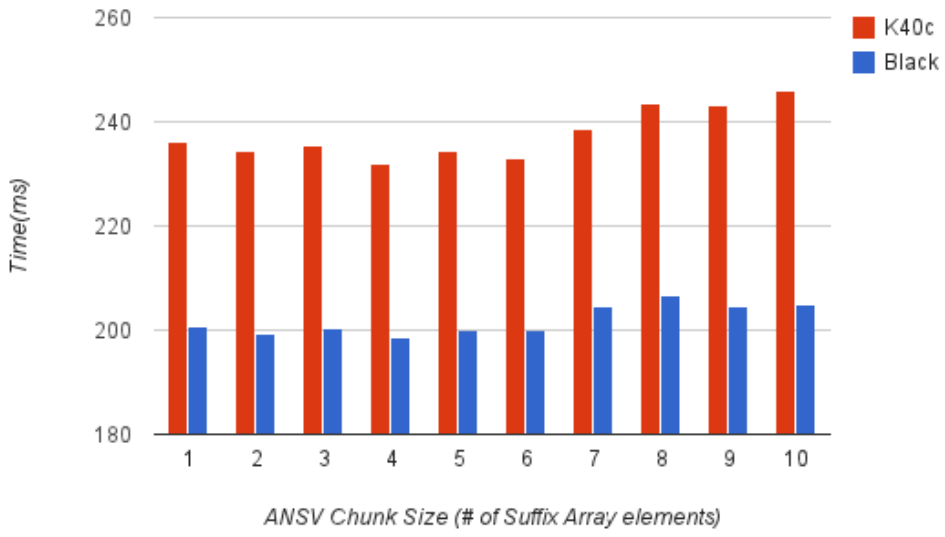

Figure 4.3: The effect of chunk size on ANSV runtime on the GPU

any outliers using a preconstructed binary tree.

Figure 4.3 shows the impact of changing the chunk size in the ANSV generation. For our setup we can see a noticeable speedup at a chunk size of 4. The chunk size of 4 is not a universal speedup for all NVIDIA GPUs. Although not presented in this paper, a mobile GPU, NVIDIA GT 650M, found speedups at a much greater chunk size. Different hardware have different memory latencies and other costs.

4.4 PLZ

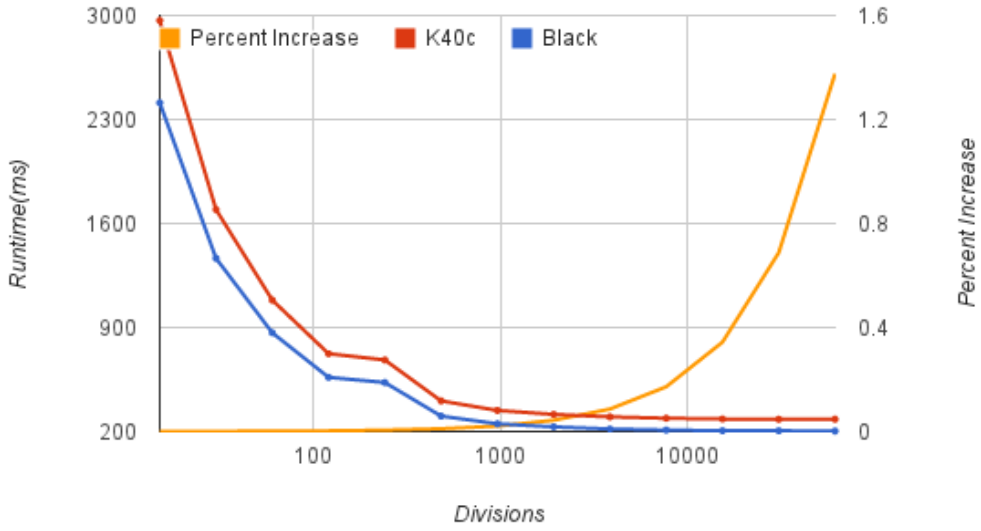

Figure 4.4: The effects of chunk size on percent increase and runtimes

To directly compare the generation of PLZ to algorithms and implementations generating the ideal LZ factorization would be unfair. The outputs are totally different, as the PLZ has lost an important property of the ideal LZ factorization, the LPF. The LPF in the PLZ are no longer the longest, as discussed

in our implementation. What can be done is a relative comparison to previous implementations. We will present the percent increase of the PLZ from the LZ factorization to help in the evaluation.

The data set and CPU benchmarks will be taken directly from the results in [23]. Specifically, we will compare our results to their benchmarks of LZ-OG, the most time efficient single threaded algorithm as seen in [20], LZ-ANSV, the sequential algorithm which computes the LZ factorization without every LPF value using lazy LZ factorization, and their contribution PLZ3, their parallel CPU algorithm using 40 cores with hyper-threading. LZ-ANSV is the closest sequential algorithm after the ANSV generation, while the ANSV generation algorithm comes from PLZ3.

The first and most important metric to look at is how the PLZ chunk size affects the final LZ factorization size. If the percent increase is too great, then the usage of PLZ is unacceptable. What percent increase is too great is a judgement that must be made by each user, as each user will have their own requirements. To pick the different chunk sizes, we decided to use number of divisions as the parameter, although we could have used the actual block size as mentioned before. More specifically we used multiples of the number of SMs (15). To try and get a wide range, we used powers of 2 to multiply.

The second most important metric is how the chunk sizes affect the runtimes. Since the suffix array construction and the ANSV generation are unrelated, we will keep our focus on the LPF time. This time assumes the suffix array and ANSV arrays are already present on GPU memory. It includes the generation of the necessary LPF and PrevOcc arrays, the isolation of the needed values using CUB's deviceSelect, and the data copy of those values back to the CPU. Many LZ factorization papers evaluate the runtime of their algorithm starting after the

suffix array is in memory. We will consider that runtime later.

Figure 4.4 presents the results with these two metrics together. We show the effect of percent increase and runtime as a function of the number of divisions. First, we notice that the percent increase grows linearly with the number of divisions. This trend is intuitive as each extra division has a chance to increase the final PLZ length if the division boundary occurs between the ideal LZ factorization. Next, we notice that the runtime generally decreases rapidly as we increase the number of divisions. At some point however, the rapid decreases stops and increasing the divisions further does not have as much effect on the runtime. As we can see in Figure 4.4, this occurs at around 480 divisions for both cards. At 480 divisions the datasets have 0.01 average percent increase. For some perspective, a file that compresses to 1 MB using the ideal LZ factorization would require an additional 105 bytes using PLZ. At this point, the Black has an average runtime of 303.7 ms, while the K40c has an average runtime of 406.8 ms. As we increase the number of divisions from 480 to 30720, Black's runtime decreases only 30 percent, while the percent increase changes over 150 percent to 1.54 percent. Similarly, K40c's runtime decreases only 32 percent. We will use this 0.01 percent increase with 480 divisions for the rest of this evaluation.

We now take a look at how our GPU implementation compares to the CPU LZ factorization implementations mentioned earlier. Table 4.2 tabulates the results and speedups found in our experiments. Our GPU implementation outperforms LZ-OG and LZ-ANSV on all the data sets. Black sees speedups between 15-24x (19.25x average) compared to LZ-OG and speedups between 13-19x (17.07x average) compared to LZ-ANSV. When compared to the 40 core PLZ3 implementation, Black performs comparatively with speedups of 1.1-1.5x (1.26x average). The slower, K40c sees speedups of 12-19x (15.02x) and 10-15x (13.32x) against

LZ-OG and LZ-ANSV respectively. K40c performed comparatively with PLZ3 having speedups and slowdowns no greater than 1.2x (0.98x average).

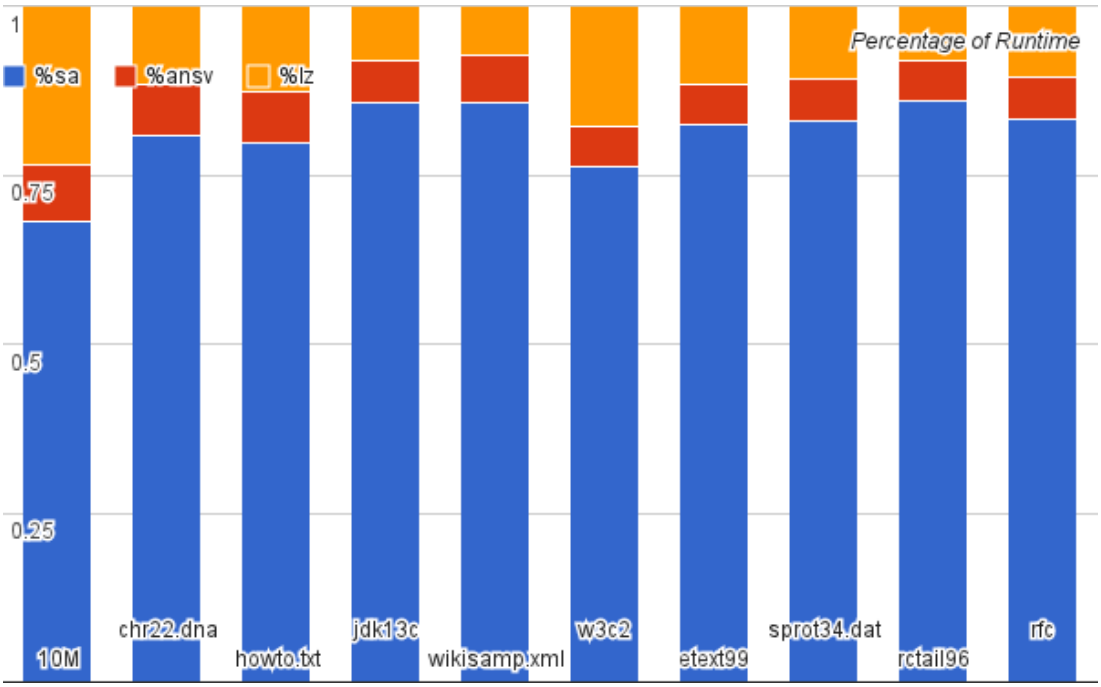

Figure 4.5: Profile of GPU implementation

Figure 4.5 shows a profile of the three main sections of our implementation, the SA, the ANSV, and the LZ. The majority of our implementation, like most LZ factorization implementations, spend most of their time constructing the suffix array. The suffix array construction takes on average 81 percent of the overall time.

Highly compressible inputs are excluded from the above results. These files incur an incredible space cost when using PLZ, as each division is likely to add an additional factor to the factorization. An additional factor with a highly compressible input could and probably will result in very high percent increases. Specifically, the file 10Midentical contained only one character. The resulting ideal LZ factorization contained 2 factors. Each PLZ division increased the LZ

factorization by 1 factor. With 480 divisions, we found 481 factors with the PLZ LZ factorization, amounting to an increase of 23950 percent.

filename	filesize (MB)	Black	K40c	lz-og	lz-ansv	plz3	lz-og Speedup Black	lz-ansv Speedup Black	plz3 Speedup Black	LZ	PLZ	percent increase
10Mrandom	9.5	0.3	0.4	4.7	4.0	0.4	15.85	13.47	1.48	1426311	1426496	0.013
chr22.dna	33.0	1.1	1.4	22.0	19.4	1.6	20.73	18.28	1.48	2461478	2461728	0.010
howto.txt	37.6	1.3	1.6	25.5	24.0	1.8	20.03	18.85	1.43	3063929	3064227	0.010
jdk13c	66.5	2.3	2.9	41.4	40.4	2.9	18.10	17.66	1.25	1209676	1210015	0.028
wikisamp.xml	95.4	3.4	4.3	61.4	59.9	4.0	18.21	17.76	1.20	2888810	2889040	0.008
w3c2	99.4	3.9	5.0	84.1	63.1	4.4	21.55	16.17	1.13	2340638	2341016	0.016
etext99	100.4	4.0	5.2	75.2	69.9	4.8	18.97	17.63	1.21	8306413	8306658	0.003
sprot34.dat	104.5	4.1	5.2	72.2	69.0	4.6	17.69	16.90	1.13	6395921	6396224	0.005
rctail96	109.4	4.1	5.2	96.5	70.0	4.8	23.54	17.07	1.16	3905843	3906149	0.008
rfc	111.0	4.3	5.5	76.6	72.8	4.8	17.81	16.93	1.12	5656068	5656367	0.005

Table 4.2: Sizes, running times(seconds), and speedups of datasets for evaluation of LZ factorization. GPU implementations use PLZ with 480 divisions

CHAPTER 5

Conclusion

We have presented an algorithm and implementation to calculate the Lempel-Ziv factorization on the GPU. We show the usage of PLZ in the calculation of the LZ factorization. Although this removed our ability to calculate the ideal LZ factorization that would be calculated in a traditional sequential algorithm, we found that using the PLZ found significant speedups on the GPU, while incurring a minimal space cost. Specifically in our evaluation, we found using 480 divisions only increased the LZ factorization by 0.01 percent. Using 480 divisions, we were able find speedups on an NVIDIA GTX TITAN Black of 15-24x over the sequential LZ-OG algorithm and up to 0.5x over the multicore parallel PLZ3. Using the PLZ, although not calculating the most space efficient ideal LZ factorization, could work well where time is a more important factor.

We have also presented a reimplementation and reevaluation of the GPU suffix array construction algorithm of Deo and Keely [6]. With the use of GPU libraries and parallel primitives, we were able to replicate their OpenCL results using CUDA on a NVIDIA GPU. On files greater than 10 MB, we found at least a 4-5x speedup over the fastest CPU implementations. This suffix array algorithm and implementation have many applications outside of data compression, most notably in bioinformatics.

CHAPTER 6

Future Work

Our implementation, like many other LZ factorization implementations, was a proof of concept to show compression speeds and compression ratio. Although we do output the correct pairs needed, we could take it further and encode them in a way that decompression implementations can understand. In doing so, we could create an actual utility to be used to compress actual data.

One aspect that was not considered in this thesis was the effect of having previous knowledge of the input. Specifically, what can we do if we know the alphabet of the input is limited. For instance during the suffix array construction, we do an initial sort of the 2/3 group using a 3 character prefix. To do this, we need to use three radix sorts. If we know exactly how many bits represent the largest character or integer in the alphabet, we can specialize the radix sort to only sort on those bits. If this is not possible, we could also check if three characters could fit into a smaller number of characters and perform a lesser number of radix sorts on them.

Recent work has looked at different ways to solve the ANSV problem. Work done in [4] and [10] has explored a technique called peak elimination to solve the ANSV problem. It is unclear whether this solution would parallelize and fit on the GPU architecture. They also found success using a single data structure to hold both the PSVs and NSVs to improve memory locality.

Another very important measurement that we did not consider was the space efficiency of our algorithm. As inputs, such as DNA sequences, grow larger and larger, it is important to make sure the algorithm is as space efficient as possible, so that the algorithm can scale. Recent work by [11] has shown methods to reduce the space needed by LZ factorization algorithms by reusing the space required by auxiliary data structures. It is especially important when working on the GPU, where hardware limits are stricter, memory is more sparse, and the communication overhead to go back and forth from the GPU to the CPU is expensive.

Multiple GPU support is becoming increasingly popular as GPU applications become more mainstream. Enabling multiple GPU support would allow our implementation to handle larger inputs. It would be interesting to investigate the added communication overhead and its effects on the overall performance. Multiple GPU support would also allow us to accompany high performance users, who have machines or clusters of machines with single or multiple GPUs.

A simpler optimization that could be added in future implementations is a more dynamic kernel launch parameters. In our implementation, we left many of the kernel parameters as program launch parameters for exhaustive trial and error. Other kernel parameters were also optimized specifically to the GPU we used for evaluation. Our implementation would still work with other GPUs, but different parameters might find faster compression speeds. One approach is to gather information about the GPU using the CUDA API before launching any kernels. We can then use that information to generate more sensible grid and block sizes. We can also use templating features for greater flexibility. Many CUDA libraries make use of this approach to great success.

NVIDIA CUDA is still a growing framework, as new hardware and new API

releases add additional features to ease or enable programmers. Recent releases have enabled dynamic parallelism and a unified memory. Dynamic parallelism allows GPU kernels to launch additional kernels from the kernels themselves. Traditionally, the GPU is used as a coprocessor and kernels must be launched from instructions on the CPU. Using dynamic parallelism, added overhead from communication between the GPU and CPU can me circumvented. Dynamic parallelism can also allow for easier or more load balanced parallelism. For example, during the final LZ factorization calculation, we could launch a new CUDA kernel to do string comparisons instead of having threads in a block working together. The benefits of unified memory in our current implementation is not clear. Since most of the data is generated and remains on the GPU, unified memory may only simplify the communication while not adding performance benefits. It would still be interesting to evaluate if these new features could provide speedups.

As of now, our implementation works only on NVIDIA GPUs through the use of CUDA. Although GPGPU development is dominated by NVIDIA CUDA on NVIDIA GPUs, the graphics market share includes many other significant vendors, including AMD and Intel. There are a variety of methods to create an implementation for use with those other vendors. One option is to rewrite the implementation using OpenCL. Other options include utilizing efforts such as OpenACC or GPU Ocelot. Furthermore, the usage of PLZ could be examined on different platforms, where the cost to perform string comparisons are not as expensive, like a multicore CPU.

BIBLIOGRAPHY

[1] A. Al-Hafeedh, M. Crochemore, L. Ilie, E. Kopylova, W. F. Smyth, G. Tischler, and M. Yusufu. A comparison of index-based Lempel-Ziv LZ77 factorization algorithms. *ACM Computing Surveys (CSUR)*, 45(1):5, 2012.

[2] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[3] S. Baxter. Modern GPU. `"https://nvlabs.github.io/moderngpu/"`, 2014.

[4] M. Crochemore and L. Ilie. Computing longest previous factor in linear time and applications. *Information Processing Letters*, 106(2):75–80, 2008.

[5] M. Crochemore, L. Ilie, and W. F. Smyth. A simple algorithm for computing the Lempel Ziv factorization. In *Data Compression Conference, 2008. DCC 2008*, pages 482–488. IEEE, 2008.

[6] M. Deo and S. Keely. Parallel suffix array and least common prefix for the GPU. *SIGPLAN Not.*, 48(8):197–206, Feb. 2013.

[7] L. P. Deutsch. Deflate compressed data format specification version 1.3. 1996.

[8] A. Ferreira, A. Oliveira, and M. Figueiredo. On the use of suffix arrays for

memory-efficient Lempel-Ziv data compression. In *Data Compression Conference, 2009. DCC'09.*, pages 444–444. IEEE, 2009.

[9] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 135–143. ACM, 1984.

[10] K. Goto and H. Bannai. Simpler and faster Lempel Ziv factorization. In *Data Compression Conference (DCC), 2013*, pages 133–142. IEEE, 2013.

[11] K. Goto and H. Bannai. Space efficient linear time Lempel-Ziv factorization for small alphabets. In *Data Compression Conference (DCC), 2014*, pages 163–172. IEEE, 2014.

[12] J. Hoberock and N. Bell. Thrust: A parallel template library. `"http://thrust.github.io/"`, 2010. Version 1.7.0.

[13] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In *Combinatorial Pattern Matching*, pages 189–200. Springer, 2013.

[14] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Automata, Languages and Programming*, pages 943–955. Springer, 2003.

[15] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In A. Amir, editor, *Combinatorial Pattern Matching*, volume 2089 of *Lecture Notes in Computer Science*, pages 181–192. Springer Berlin Heidelberg, 2001.

[16] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

[17] D. Merrill. CUB. "https://nvlabs.github.io/cub/", 2014. Version 1.3.1.

[18] Y. Mori. Libdivsufsort. "https://code.google.com/p/libdivsufsort/wiki/SACA_Benchmarks", AUG 2008.

[19] S. Odeh, O. Green, Z. Mwassi, O. Shmueli, and Y. Birk. Merge path-parallel merging made simple. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1611–1618. IEEE, 2012.

[20] E. Ohlebusch and S. Gog. Lempel-Ziv factorization revisited. In *Combinatorial Pattern Matching*, pages 15–26. Springer, 2011.

[21] A. Ozsoy and M. Swany. Culzss: LZSS lossless data compression on CUDA. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 403–411. IEEE, 2011.

[22] A. Ozsoy, M. Swany, and A. 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.

[23] J. Shun and F. Zhao. Practical parallel Lempel-Ziv factorization. In *Data Compression Conference (DCC), 2013*, pages 123–132. IEEE, 2013.

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

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

[26] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE TRANSACTIONS ON INFORMATION THEORY*, 23(3):337–343, 1977.

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

[28] Y. Zu and B. Hua. GLZSS: LZSS lossless data compression can be faster. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, GPGPU-7, pages 46:46–46:53, New York, NY, USA, 2014. ACM.