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 Zoë Wood, Ph.D., Department of Computer Science
COMMITTEE MEMBER:	Assistant Professor Chris Lupo, Ph.D., Department of Computer Science
COMMITTEE MEMBER:	Professor Franz Kurfess, Ph.D., Department of Computer Science

ABSTRACT

Optimizing Lempel-Ziv Factorization for the GPU Architecture

Bryan Ching

TODO

ACKNOWLEDGMENTS

Thanks to:

- My parents.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Lossless data compression has the ability to reduce the 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 accoss I/O channels. Compression algorithms have a tradeoff, in that they require an additional computation to be done on the original data before a compressed version can be used. This computation 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 rates would generally be helpful. This work takes a look into speeding up those compression rates by performing the compression directly on a GPU, a graphics processing unit.

An increase of applications and algorithms are being developed to utilize the relatively new general purpose computing (GPGPU) apsect 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 many of the world's fastest supercomputers utilize GPGPUs in large clusters.

CHAPTER 2

Background

2.1 GPU Architecture

2.1.1 CUDA

2.1.2 Memory Model

2.1.3 Thread Model

2.1.4 Libraries and Parallel Primitives

There exist a variety of libaries that make development on CUDA more streamlined. These libraries provide a variety of uses for the CUDA programmer. 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/block/grid handling. 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 the quirks that come with it. Second, abstraction allows a problem to be continually optimized, while presenting a common API to use. This allows us to somewhat futureproof our implementation, since the libraries should be updated in the future against newer CUDA versions and hardware. Also, some problems may seem simple at first, but too complex for a user to implement every time. 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 min or sum of an array. Although these primitives may seem simple at first, there are many tricks 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 library. The Thrust library claims to resemble the STL and provide device-wide primitives to be used. 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. 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, like streams. That said, many of the algorithms are shared between CUB and Thrust. We decided to choose CUB for its claims at higher performance than Thrust and specific features unavailable in Thrust, like the primitives that work on the block level. 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. More specifically we make use of its algorithm for merge sorting. 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 to achieve this [?]. Merge Path realizes that if we imagined the two arrays on a grid, the merge sort follows a path through it. We can run diagonals throught the grid to find their intersection with this Merge Path, where all comparisons are true on one side and false on the other. This can be done quickly using a binary sort. We can now figure out exactly which sections of the arrays correspond to sections of the final merged array. With even divisions, we can evenly divide the final output among the threads, each knowing which sections of the input array they need. Those threads can then merge those sections, employing Merge Path if they so wish to.

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 or too big for the host system. The next is that it is feasible to write a higher performant 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 decide to ignore these caveats in our implementation.

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 tradeoff 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 for one over 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[**?**]. Their work, LZ77, used 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[?]. 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 tweaked their algorithm into LZ78 which instead provided an explicit dictionary that could be used for random lookup during decompression[?]. 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[?]. 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[?]. Today, it is used in popular archivers such as PKZip and RAR. Other variants, including LZMA and LZJB, change some aspect of the original algorithm to increase compression speed or the compression ratio. The focus of our thesis is on LZ77, and its LZ factorization.

Practical implementations of LZ77 use a sliding window buffer, a small section of the overall input, where the algorithm operates on a longer factored prefix and a short unfractored suffix. In practice, it has shown that 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 mix and match 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

The LZ factorization of a string S[n] decomposes S into factors S = w1w2wk where k¡=n, where each factor wi is either the longest factor that appears left of wi in S or is a new character. For example, the LZ factorization of string abbaabbbaaabab has the factorization a.b.b.a.abb.baa.ab.ab. Various algorithms have been compared experimentally in [?]. In general, LZ factorization algorithms all make use of a few common data structures and stages, the suffix array, the LCP array, the LPF array, and the PrevOcc array.

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 lexicicographically ordered array of integers of size n where each integer represents a suffix of S, so that suf[SA[0]] ¡ suf[SA[1]] ¡ . . . suf[SA[n-1]].

First introduced as 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 of Kark [?] 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. 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 Meo and Deeeley utilizes the skew algorithm. Inspired by most of their ideas, our work is also a reimplementation and benchmark of their algorithm.

2.2.2.2 LCP and LPF Array

The LCP, longest common prefix, array is an auxiliary structure to the suffix array that provides the longest common prefix between successive suffixes in SA. Formally, position i in the LCP array, LCP[i] = lcp(suf[SA[i-1]],suf[SA[i]]). The LCP array can be expensive to compute, but recent algorithms have found that the LCP array can be constructed during the construction of the suffix array.

The LPF, longest previous factor, array holds the lengths of the longest factors at any position i. In other words, LPF[i] holds the maximum lcp of suf[sa[i]] and all suffixes less than i.

2.2.2.3 LZ Factorization Calculation

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 this naive algorithm runs in O(n3) time. This can be bounded to O(n2) time using the knowledge that the total length of all lcp's is N. In [?], the number of positions that a suffix needs to be lcp against is reduced to the PSV and NSV. We only need to check longer suffixes, number smaller, and only the closest ones since lexi order. The NSV, next smaller value, and PSV, previous smaller value, make up the ANSV, all nearest smaller values, problem. Computing the ANSV problem can be done linearly and sequentially using a stack-based algorithm found in CITE GABOW. This observation now reduces the problem to O(n) time.

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 [?]. More recent LZ factorization algorithms have decided to forgo this intermediate step and directly calculate the LZ factorization. If we examine Table X, we can see that the LZ factorization of the string abbaabbbaaabab can be reduced to 8 positions. The positions where a factor does 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 [?, ?, ?], the LPF value is only calculated at the start of a factor. The recent algorithms [] make use of this fact for simplicity and speedup. Figure shows the basic pseudocode for calculating the LZ factorization given the PSV and NSV arrays using this lazy method. The PrevOcc array simply holds the position or suffix where the LPF occurs.

Finally, the LZ factorization can be encoded simply with a position of previous occurence and the length of the match or a character, if the length is 0. It can also be encoded using the pair of position and previous occurance. Practical compression schemes might be encoded in triplets with the position, length, and the first letter of mismatch.

```
procedure LazyLZFactorization($S, n, PSV, NSV$)
    $i \leftarrow 1$
    while $i \leq n$ do
        a LZ factor starts here
        if $lcp(i, PSV) \geq lcp(i, NSV)$ then
            LZ Factor = (PSV,lcp(i,PSV))                ▷ Pair (PrevOcc,LPF)
        else
            LZ Factor = (NSV,lcp(i,NSV))
        end if
        if LPF = 0, PrevOcc = -1, and the character is inserted
        $i \leftarrow i + max(LPF, 1)$
    end while
end procedure
```

Figure 2.1: LZ Factorization

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. The key observation on these ports is the use of pipelining. Most if not all of these ports split up the data to run their individual algorithm on. Many of their original algorithms allow for this. Benefits can be found using CUDA streams to concurrently copy partial data and running kernels. This is a feature not available in LZ factorization. Many of these applications also make use of a sliding window, as seen in most LZ77 implementations. Don't know 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 Jshun [**?**] 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. They provide many of the inspirations throughout our project and implementation. 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 BLZ, which we'll describe later.

CHAPTER 3

Implementation

There exists three main steps in my implementation, 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 meely [?]. The pseudocode for the skew algorithm is presented in. Smaller individual kernels can be used to first.

To sort the partial suffixes using radix sort, I used the CUB implementation of radix sort. After those partial suffixes are sorted, they need to be checked for uniqueness. To calculate the prefix sum, I used the device wide CUB implementation. Room for improvement. Transfer back to the cpu to control recursion Used cub for istriple kernel ModernGPU merge sort

3.2 ANSV

To calculate the needed ANSV values I used the parallel algorithm from jshun [?].

The first step is to build a balanced binary tree, where the leaves are elements

```
procedure COMPUTESA($s, sa, n$)
    initMod12();    ▷ Kernel to set flags at 2/3. DeviceSelect to get s12,sa12
    radixSort($s12$);                                        ▷ DeviceRadixSort
    radixSort($s12$);
    radixSort($s12$);
    lexicRankOfTriplets();    ▷ Custom kernel to check unique. Inclusive Sum
  to count. Custom kernel to get s12.
    if !allUniqueRanks then
        computeSA($s12, sa12$);                                    ▷ Recursion
        storeUniqueRanks();                                           ▷ Kernel
    else
        computeSAFromUniqueRank();                                    ▷ Kernel
    end if
    radixSort($s0$);
    mergeSort($s0, s12$);                         ▷ Merge Path + Merge Sort
end procedure
```

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

```
procedure ANSV
    for each level of MinTree do                         ▷ Bottom Up Construction
        MinTree($level$);            ▷ Build level by calculating minima of children
    end for
    ANSVKernel($mintree, chunkSize$)
end procedure
```

```
procedure ANSVKernel
    $chunk \leftarrow threadID * chunkSize$         ▷ Each thread gets a unique chunk
    ANSVLinear($chunk$)
    if ( then$chunk$ detects no PSV/NSV)          ▷ ANSVLinear may be wrong
        checkMinTree($mintree$)                        ▷ Manually check MinTree
    end if
end procedure
```

Figure 3.2: ANSV Generation

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

from SA, and the ancestors are the minima of their children. Although more efficient algorithms may exist, I 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. (Something about memory locality and indexing into 1d array).

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 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

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	-

compensate for this, each thread will manually check each position that did not find a smaller value using RMQs 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. I did play around with 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.

3.3 LZ Factorization

The final step is to calculate the LZ factorization.

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		

At first I attempted to follow the parallel algorithm of jshun. In their work, the LPF array is calculated for every position, and then the LZ factorization is solved using a parallel list ranking algorithm. Like many more recent works, I found that the calculation of the LPF array at every index to be too computationaly expensive and wasteful, even on the GPU. Instead, my work will also employ the lazy LZ factorization, mentioned in [?]. 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.

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: Split=7,LZ=8

I propose breaking away from the ideal LZ factorization and using a BLZ. I define the ideal LZ factorization to be the original LZ factorization, calculated by starting at the first character and greedily solving the problem from left to right. This LZ factorization would be what is calculated from the original sequential algorithm, and should be the absolute best, hence ideal. In BLZ, the string S will be broken into chunks 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, I am 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 BLZ LZ factorization to be larger than the ideal LZ factorization. There is also an unlikely chance for them to be exactly the same. I would like to explain the different scenarios to you with an anology.

Let's imagine the LZ factorization of a string to be a stairway with floors. Generating the LZ factorization is like constructing said stairway and floors. Each floor represents an entry into the LZ factorization. Each stair represents a character match in the LPF for the floor beneath it. A sequential algorithm would have you start at the ground floor. At that point we know where the LPF is located. Of course in reality, we know of two possible locations where the LPF might be, but we'll simplify it to one for this anology. When you are on a floor, you, the builder, walk up a stair for each match. When the LPF no longer matches, you put a floor to represent a new entry and repeat. In the end, the number of stairs should match the number of original characters, and the number of floors represent the length of the ideal LZ factorization.

thm: The BLZ LZ factorization ¿= ideal. proof: Now we'll look at what happens when we use BLZ to parallelize the problem. First, we remove all the floors. Next, to split up the input, we insert ground floors at regular intervals. After, we travel the stairway inserting floors, like the sequential algorithm above. Finally, we stop when we get to the next ground floor. There are several scenarios that can occur. The first scenario occurs when an inserted ground floor is inserted where an existing floor used to exist. The LZ factorization calculated, starting at this ground floor, is exactly the same as the ideal LZ factorization, up until the next ground floor. The extreme of this scenario happens if every inserted ground floor is inserted where a floor used to exist. The BLZ factorization is then exactly the same as the ideal LZ factorization.

The next scenarios occurs when the inserted ground floors are inserted in between existing floors from the ideal. When this happens, the BLZ LZ factorization is automatically increased by at least one, from the new ground floor. The LPF of the previous floor is no longer the longest it can be. This also limits the maximum offset for any offset to be the chunk size. This new ground floor now performs the sequential algorithm as always. The next floor that is inserted by the builder may or may not be where a floor existed. This is entirely dependent on the input string and is impossible to predict. It is very likely though at some point, for the floors to again match the existing floors. At that point onward, the BLZ LZ factorization again matches with the ideal.

The next question to be answered is deciding the chunk size. 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 tradeoff that should be left to the user, in my opinion. In my implementation, I have the option to use arbitrary sizes or even divisions of the input. Some optimal sizes might be divisions of the input of multiples of the number of multiprocessors. In any case, it is impossible to predict the compression ratio, and different people will have different priorities.

One thing that I did 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 I 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 value blockDim.x (32), and a mismatch is assigned the value threadIdx.x (0-31). The threads can then cooperatively min reduce the problem, 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 exist 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 soley 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. My implementation will use the parallel string comparison for evaluation.

Finally, it is shown how we can calculated the LPF. Because the LPF can occur at either the PSV or the NSV, we need to check both and pick the longer as mentioned before. Reaching the edge of the boundary of a chunk in the first search allows us to skip the second search.When the longer is found, we can insert that into a prevocc array.

The final LZ factorization, made up of relevant entries from the LPF and prevocc arrays, can be gathered by using the DeviceSelect and CudaMemcpy.

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: Split=4,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: Split=3,LZ=8

CHAPTER 4

Results

4.1 Experimental Setup

4.1.1 Test Machine

K40 Timings done cuda events

4.1.2 Data

Talk about datasets

4.1.3 Validation

Talk about compare to cpu implementations to validate

4.2 Suffix Array

The evaluation of the suffix array is actually a evaluation of a reimplementation of the fastest known GPU suffix array construction algorithm (SACA) by Deo and Meely[**?**]. The benefits and applications of the suffix array has already been detailed in chapter . . . Deo and Meely'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

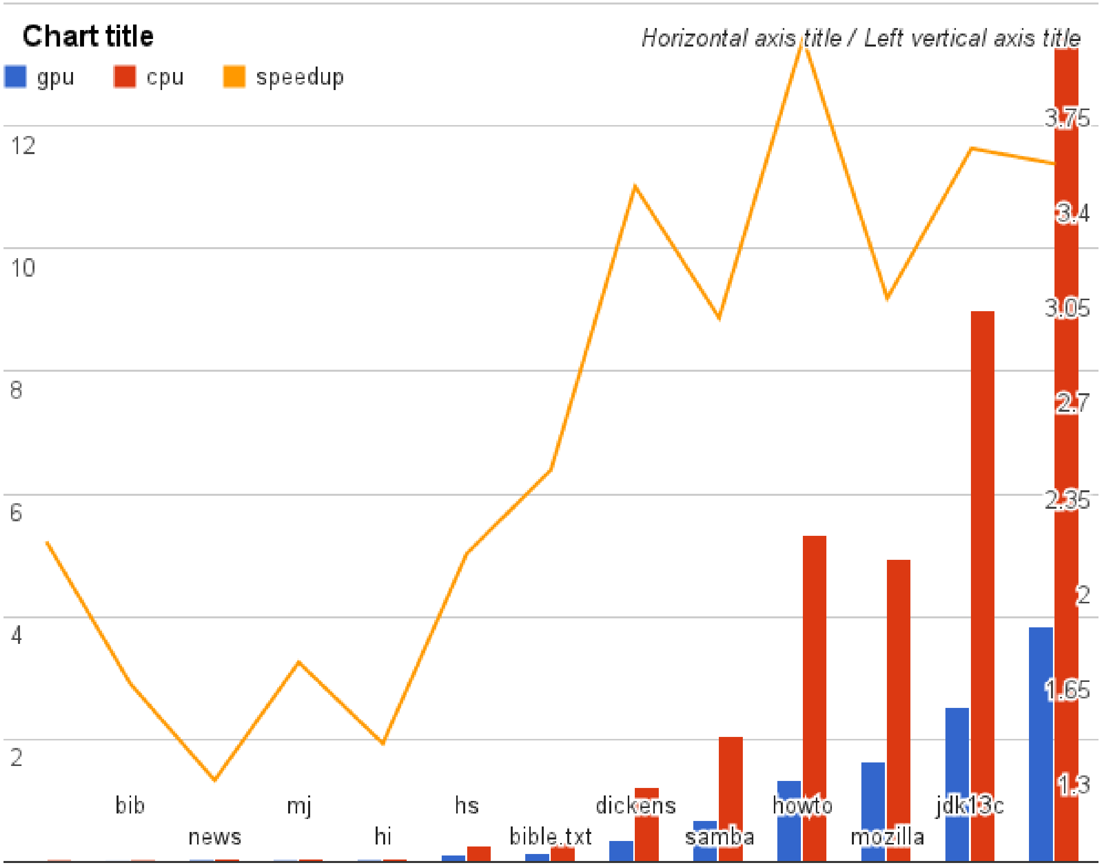

Figure 4.1: The basic structure of the graphics pipeline. Note the vertex processor, rasterizer, fragment processor, and termination in a frame buffer. Image taken from [?]

expected to be significantly different. We will be comparing out results to a set of SACA benchmarks found on the libdivsufsort wiki. That benchmark compares the fastest CPU SACA implementations on a variety of test files. We will compare our GPU implementaion against the fastest CPU time for each file. Files were picked to match closely with Deo and Meely's evaluation. GPU times include parsing the file, transfering the data both ways, and the construction of the suffix array.

.Figure presents the results comparing the CPU implementations to our GPU

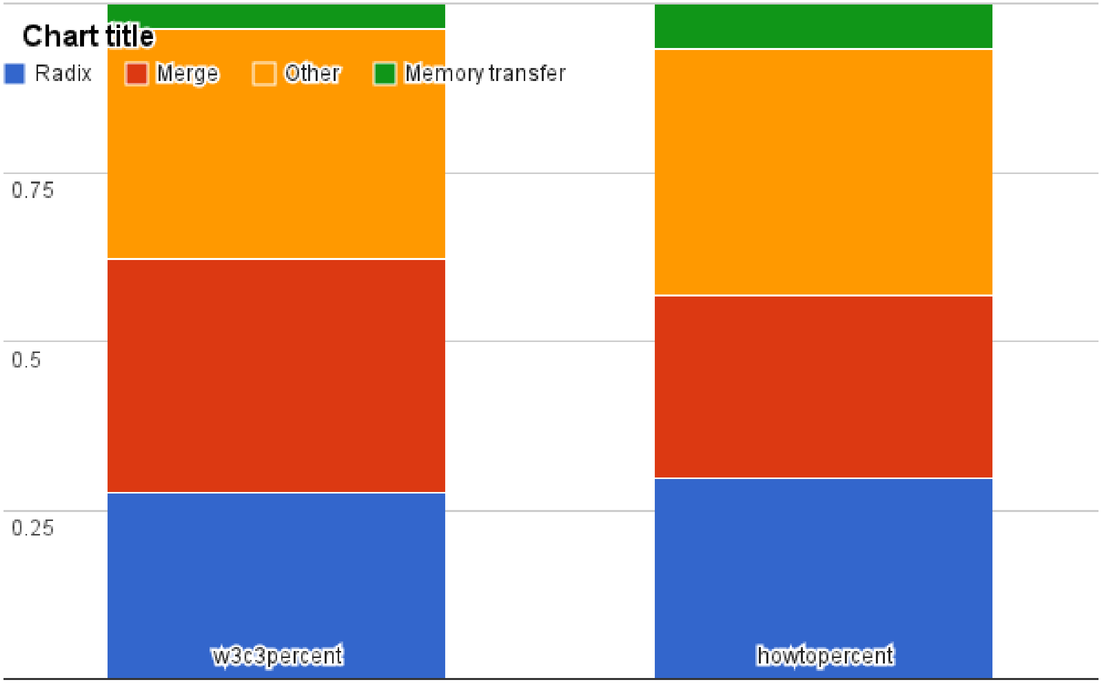

Figure 4.2: The basic structure of the graphics pipeline. Note the vertex processor, rasterizer, fragment processor, and termination in a frame buffer. Image taken from [?]

implementation. The first thing to note is that the CPU SACA benchmarks are significantly faster than those used by Deo and Meely. Our GPU implementation did not see the speedup of 35x that theirs did, but we still found around a 3-4x speedup for most files. We did not have their implementation or their raw result data to compare against. 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 input symbol seen in [?]. We found that after a certain point, our GPU implementation was achieving rates of around .03 to .04 ms per input symbol.

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

size	name	gpu	ms per input
53161	paper1	16	0.301
111261	bib	21.2	0.191
377109	news	36.8	0.098
448779	mj	30.9	0.069
509519	hi	36.1	0.071
3295751	hs	130.6	0.040
4047392	bible.txt	141.5	0.035
10192446	dickens	353.5	0.035
21606400	samba	693	0.032
39422105	howto	1339.7	0.034
51220480	mozilla	1642.6	0.032
69728899	jdk13c	2525.4	0.036
104201579	w3c2	3840.8	0.037

fully saturate the GPU, and the cost of intialization 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 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.

aug 24 2008

4.3 ANSV

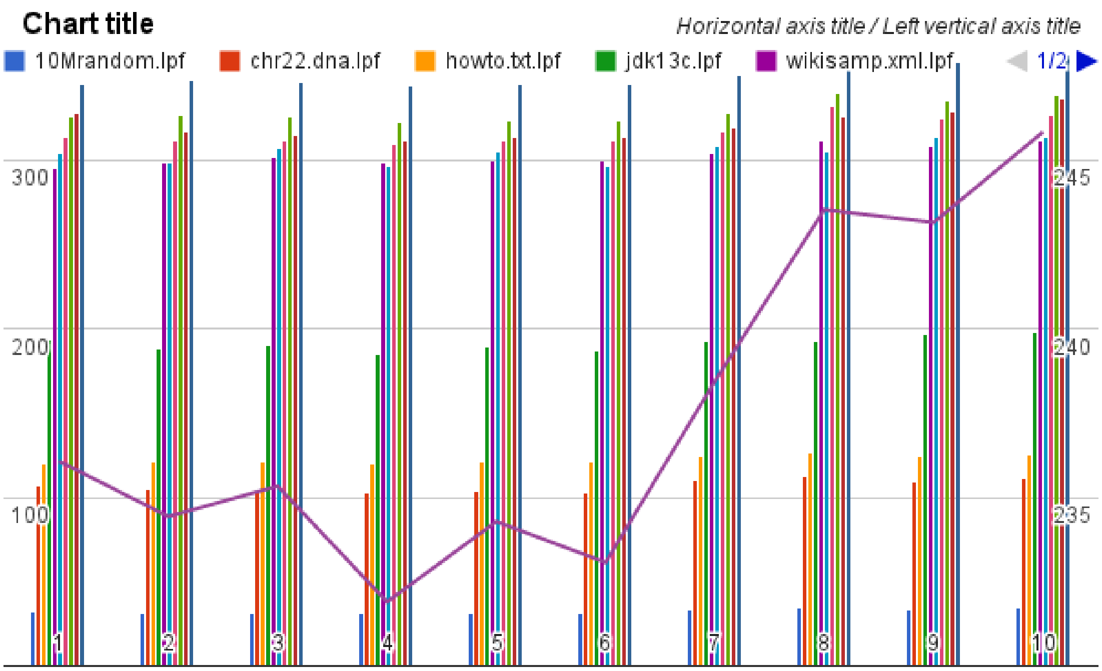

Figure 4.3: The basic structure of the graphics pipeline. Note the vertex processor, rasterizer, fragment processor, and termination in a frame buffer. Image taken from [?]

As discussed in chapter X, the 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 any outliers using a preconstructed binary tree.

Figure X 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. Altough not presented in this paper, a NVIDIA GT 650M found speedups at a much greater chunk size.

Different hardware have different memory latencies and other costs. I'll discuss this further in a future section.

Figure shows a profile of the ANSV generation in the two main steps, the construction of the binary tree and the chunk processing with a chunk size of 4. The construction of the min tree was no more than 4 percent of the overall ANSV generation. Several future optimizations were discussed earlier for the min tree, but seeing as it is only a small percentage of the overall runtime, time is probably better spent somewhere else. See Amdahl's law [?]. The bigger chunk of the runtime is in the chunk processing, as expected.

ms per input - ansv

4.4 BLZ

name	size	total	lz-og	lz-ansv	plz3	speedup og	speedup ansv	speedup
10Mrandom.lpf	9.5	371.1	4670	3970	437	12.58	10.70	1.18
chr22.dna.lpf	33.0	1385.8	22000	19400	1570	15.88	14.00	1.13
howto.txt.lpf	37.6	1620.3	25500	39400	1820	15.74	24.32	1.12
jdk13c.lpf	66.5	2940	41400	40400	2860	14.08	13.74	0.97
wikisamp.xml.lpf	95.4	4304.2	61400	59900	4030	14.27	13.92	0.94
w3c2.lpf	99.4	4988.5	84100	63100	4420	16.86	12.65	0.89
etext99.lpf	100.4	5182	75200	69900	4800	14.51	13.49	0.93
sprot34.dat.lpf	104.5	5210.9	72200	69000	4600	13.86	13.24	0.88
rctail96.lpf	109.4	5227.3	96500	70000	4770	18.46	13.39	0.91
rfc.lpf	111.0	5474	76600	72800	4830	13.99	13.30	0.88

To directly compare the generation of BLZ to algoritms and implementations

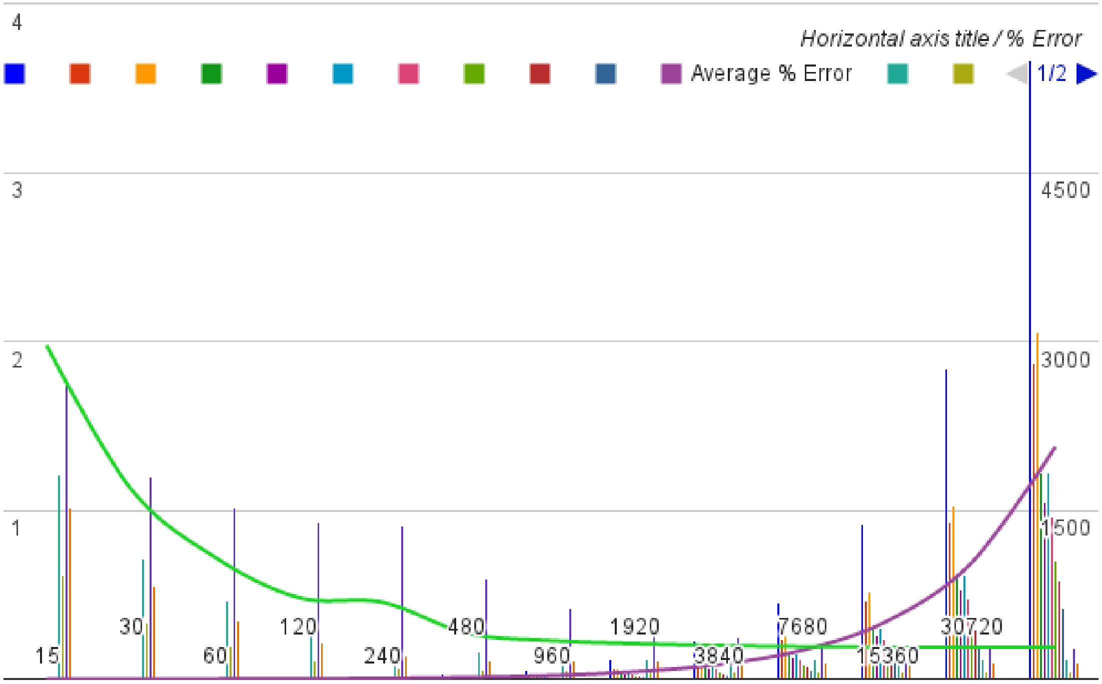

Figure 4.4: The basic structure of the graphics pipeline. Note the vertex processor, rasterizer, fragment processor, and termination in a frame buffer. Image taken from [?]

generating the ideal LZ factorization would be unfair. The outputs are totally different, as the BLZ has lost an important property of the ideal LZ factorization, the LPF. The LPF in the BLZ 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 BLZ from the LZ factorization to help in the evaluation.

The data set and CPU benchmarks will be taken directly from the results in [?]. Specifically, we will compare our results to their benchmarks of LZ-OG, the most time efficient algorithm as seen in [?], LZ-ANSV, the sequential algorithm which computes the LZ factorization without every LPF value using lazy LZ factorization, and their contribution PLZ3, a parallel CPU algorithm where much of our inspriration comes from, 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 BLZ block size affects the final LZ factorization size. If the percent increase, which we will now call percent error, is too great, then the usage of BLZ is unacceptable. What percent error 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 block sizes, I decided to use number of divisions as the parameter, although I could have used the actual block size as mentioned before. More specifically I used multiples of the number of SMs (15). To try and get a good spread I used powers of 2 to multiply.

The second most important metric is how the block 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 XX presents the results with these two metrics together. We show the effect of percent error and runtime as a function of the number of divisions. The key difference is A is geometric while B is linear. We have included averages as a convenience. First, we notice that the percent error grows linearly with the number of divisions. This trend is intuitive as each extra division has a chance to increase the final BLZ length if the division boundary occurs between the ideal LZ factorization. Next, we notice that the runtimes 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 XX, this occurs at around 480 divisions with an average of 378.3 ms. At this point, the percent error averages around .01 percent. For some perspective, a file that compresses to 1 Mb using the ideal LZ factorization would require an additional 105 bytes using BLZ. As we increase the number of divisions from 480 to 30720, the runtime only decreases 31 percent to 262.1 ms. Meanwhile, that increase increases the percent error over 150 percent to 1.54 percent error. Further increases of the number of divisions might actually begin to slowly increase the runtime, as we see the average increases ever so slightly to 262.6 ms when we double the divisions to 61440. We will now deem this .01 percent error acceptable and use these values as we continue the evaluation.

talk analysis

We now take a look at how our GPU implementation compares to the CPU LZ factorization implementations mentioned earlier. Table X tabulates the results and speedups found in our experiments. Our GPU implementation outperforms LZ-OG and LZ-ANSV on all the data sets. Our implementation sees speedups between 12x and 18x compared to LZ-OG and speedups between 10x and 14x compared to LZ-ANSV. When compared to the 40 core PLZ3 implementation, our GPU implementation performs relatively similar with speedups or slowdowns no more than 1.2x.

Figure X 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

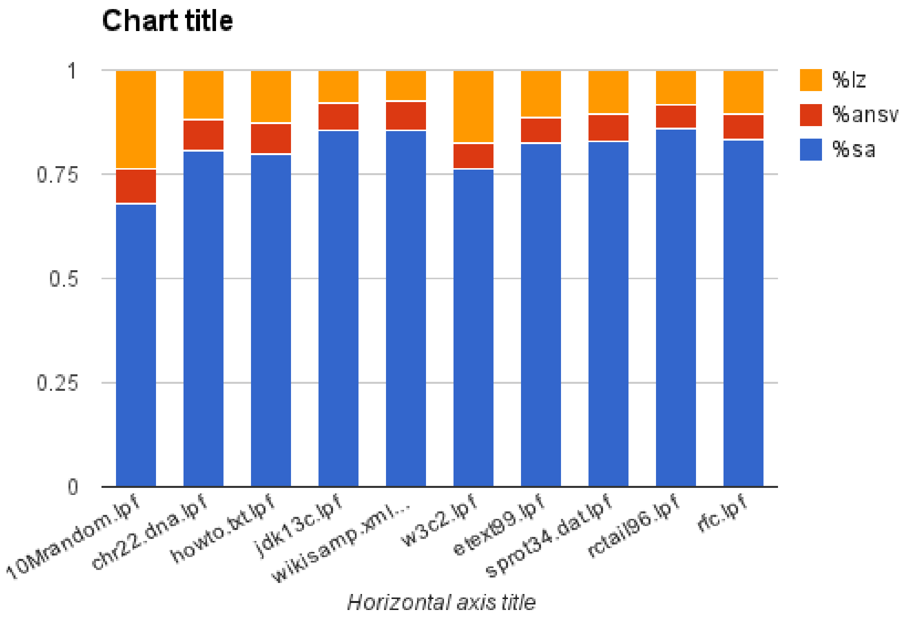

Figure 4.5: The basic structure of the graphics pipeline. Note the vertex processor, rasterizer, fragment processor, and termination in a frame buffer. Image taken from [**?**]

array. The suffix array construction takes on average 81 percent of the overall time.

Excluded highly compressable.

Comparison to gpu doesnt make sense.

CHAPTER 5

Conclusion

We have presented an algorithm and implementation to calculate the Lempel-Ziv factorization on the GPU. Our algorithm requires O(n) work and O(time). We show the usage of BLZ 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 BLZ found significant speedups on the GPU, incurring a space cost of at most .01 percent. Using the BLZ, although not calculating the most space efficient ideal LZ factorization, could work well in a more practical environment.

We have also presented a reimplementaion and reevaluation of the GPU suffix array construction algorithm of Deo and Meely []. 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 3-4x 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 factorizaton implementations, was just 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 decompressers can understand.

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.

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.

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 [?] has shown methods to reduce the space needed by LZ factorization algorithms by reusing the space required by auxillary 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.

A simpler optimization that could be added in future implementations is a more dynamic and adaptable kernel launch parameters. In our implementation, we left many of the kernel paramters 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. The first is to rewrite the implementation using OpenCL. OpenACC, GPU Ocelot Furthermore, the usage of BLZ could be examined on different platforms, where the cost to perform string comparisons are not as expensive, like a multicore CPU.