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

Recent growth in the commercial availability of consumer grade 3D user interface devices like the Microsoft Kinect and the Oculus Rift, coupled with the broad availability of high performance 3D graphics hardware, has put high quality 3D user interfaces firmly within the reach of consumer markets for the first time ever. However, these devices require custom integration with every application which wishes to use them, seriously limiting application support, and there is no established mechanism for multiple applications to use the same 3D interface hardware simultaneously. This thesis proposes that these problems can be solved in the same way that the same problems were solved for 2D interfaces: by abstracting the input hardware behind input primitives provided by the windowing system and compositing the output of applications within the windowing system before displaying it. To demonstrate this it presents a novel Wayland compositor which allows clients to create 3D interface contexts within a 3D interface space in the same way that traditional windowing systems allow applications to create 2D interface contexts (windows) within a 2D interface space (the desktop), as well as allowing unmodified 2D Wayland clients to window into the same 3D interface space and receive standard 2D input events. This implementation demonstrates the ability of consumer 3D interface hardware to support a 3D windowing system, the ability of this 3D windowing system to support applications with compelling 3D interfaces, the ability of this style of windowing system to be built on top of existing hardware accelerated graphics and windowing infrastructure, and its ability to support unmodified 2D interface applications windowing into the same

3D windowing space as the 3D interface applications. This means that application developers could create compelling 3D interfaces with no knowledge of the hardware that supports them, that new hardware could be introduced without needing to integrate it with individual applications, and that users could mix whatever 2D and 3D applications they wish in an immersive 3D interface space regardless of the details of the underlying hardware.

ACKNOWLEDGMENTS

Thanks to:

- My advisor Zoë Wood, for all of her guidance and wit.
- The wonderful people in the Wayland and QtWayland communities, without whom I would not have a functioning prototype.
- My parents, my family, and my girlfriend Katy, for supporting me always.

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

2.2 Compression

2.2.1 Lempel Ziv Factorization

The LZ factorization of a string S[n] decomposes S into factors S = w1w2. . . wk 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 ab-baabbbaaabab has the factorization a.b.b.a.abb.baa.ab.ab. This can be encoded simply with a position of previous occurence and the length of the match or a character, if the length is 0. Practical compression schemes might be encoded in triplets with the position, length, and the first letter of mismatch. 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, and the LPF array.

2.2.1.1 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]].

Suffix tree applications.

Suffix tree talk.

Various algorithms exist for the construction of suffix arrays. The skew algorithm of Kark [**?**] uses a divide and conquer approach to construct a partial suffix array to infer the rest of the positions. 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. Our work is also a reimplementation and benchmark of that algorithm inspired by most of their ideas.

2.2.1.2 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 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.

2.2.1.3 LZ Factorization Calculation

Previous works have computed the LPF array from the LCP array using ranged minimum queries. More recent works do a lazy computation.

The lz factorization can be computed by following the lpf array.

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 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 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.

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 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.

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. We can use either the LPF or the prevocc array as flags to indicate where the LZ factorization occurs. A final deviceSelect using flags from CUB can be used in a practical solution.