LZO Data Compression Algorithm on OpenCL for GPU's

A Project Report

submitted by

NIKETH BOREDDY

in partial fulfilment of the requirements
for the award of the degree of

BACHELOR OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

June 2016

THESIS CERTIFICATE

This is to certify that the thesis titled LZO Data Compression Algorithm on OpenCL for GPU's , submitted by **Niketh Boreddy**, to the Indian Institute of Technology, Madras, for the award of the degree of **Bachelor of Technology**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. Rupesh Nasre	**Prof. Krishna M. Sivalingam**
Research Guide	Head of Department
Assistant Professor	Professor
Dept. of Computer Science Engineering	Dept. of Computer Science Engineering
IIT-Madras, 600 036	IIT-Madras, 600 036

Place: Chennai

Date: 10th June 2016

ABSTRACT

KEYWORDS: LZO, OpenCL, GPU, Data Compression, Parallel Algorithms

In the following project we seek to develop a parallel version of the LZO(Lempel-Ziv-Obehumer) compression algorithm on the OpenCL(Open Computing Language) framework for execution on a GPU(Graphics Processing Unit). We will be looking into the different serial, parallel implementations of compression algorithms on CPU's, GPU's followed by a brief description of the original LZO algorithm and the basics of GPU architecture. An analysis on how the compression ratios and the execution times of the parallel version compare with that of the serial version of LZO is provided along with optimization techniques that improve the performance of the algorithm on GPU's. Overall we have been able to achieve a speedup of 1.5x over the serial lzo algorithm when we take into account the net time taken for execution of the program. Testing has been done on 11 files from different benchmarks to account for a variety in data and the compression ratio has been unaffected by the GPU version.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Data compression is the process by which information is conveyed by using fewer storage units than the original representation. Data compression finds many applications in the areas of data storage and transmission. As the number of applications that are involved with large volumes of data increases and as the use of computers extends to newer disciplines data compression techniques play an important role in improving the efficiency of such applications. A few examples would include storage of most types of multimedia data (audio, video and images), genetic compression algorithms that exploit the redundancy in the genetic data to sometimes achieve compression that allows for up to 95% reduction in the file sizes and transmission of data across I/O channels where the main bottleneck would be the bandwidth involved and hence any reduction in the file sizes would be an indirect improvement.

There are two main classes of compression algorithms, lossy and lossless. Lossy compression algorithms are employed when some loss of information is acceptable and there is no significant loss of information when perceived by the human senses although there might be some loss of detail which are mostly non-essential. Lossy compression is applied to multimedia data such as images, audio and videos.

Lossless compression is the process of encoding information such that one would be able to generate the exact original file from the compressed one. Such algorithms exploit the data redundancy that are generally exhibited in read-world data. The LZ family of compression methods are one of the most popular lossless compression methods, available in various flavours such as LZW, LZR, LZO, LZSS, LZ4 etc each often geared towards a particular type of input files with an optimal trade-off between the compression ratio and the compression speeds.

Among the lossless compression techniques there are two classes of algorithms, statistical and dictionary based algorithms. In statistical schemes the output string is determined based on the probability of occurrence of the input symbols whereas dictionary based schemes stores the previously read input strings in a dictionary and often make use of pointers for determining the location of the repeated inputs. The current algorithm under study LZO is a dictionary based one and we shall be studying it in greater detail in the coming chapters. Statistical algorithms generally require the same amounts of time for compression and decompression but for dictionary based algorithms decompression tends to be faster than compression and are more capable of achieving high compression ratios.

Although there are a lot of benefits from compression we must understand that compression in not free. Higher compression almost always demands higher computations and therefore requires more time. Optimizations that tend to improve the compression speeds usually compromise on the compression ratio and hence there is a delicate trade-off involved here. In the current project we will be try to implement a parallel version of the LZO algorithm for GPU devices in order to obtain a speedup without compromising on the compression ratios and free up the CPU for other important tasks. Since the output format for the parallel version of the code will be slightly different, a modified version of the decompression algorithm is also presented for decompression purposes.

CHAPTER 2

LITERATURE REVIEW

We will now have an overview of the previous work that has been done in the field of developing parallel versions of the LZ algorithms for GPU's and CPU's.

The usage of GPU's for compression algorithms is a relatively new area of research with most of the work happening in the last decade. Therefore not many novel and effective solutions for developing parallel versions of the compression algorithms are available. The following are the most relevant and cited publications.

2.1 Parallel Solutions for GPU's

Shunji Funasaka et al. 2015 (1): The following paper deals with the acceleration of the LZW(Lempel-Ziv-Welch) algorithm for TIFF(Tagged Image File Format) images using a CUDA(Computer Unified Device Architecuture)-enabled GPU. The modified algorithm runs the LZW compression on a set of consecutive row(s) of the image and the final compressed image is reconstructed by concatenating the resulting LZW codes into one with the help of the prefix sums of the lengths of the codes which in turn can be efficiently computed by a GPU. The algorithm also uses a hash table for implementing the back-pointer table(which is used for determining the matches in the LZW algorithm) that helps in reducing the required amount of storage space and improves speeds(due to reduced cache misses).
The experiments have been conducted using a NVIDIA GeForce GTX 980 (16 SMP's (Streaming Multi-Processors) each with 128 cores, 1.13GHz clock) and Intel Core i7 4790 (3.6GHz) using three different image files of size $\sim$12MB and compression on GPU is about 3 times faster than that on CPU with the compression ratio unaffected
Compression Speeds attained: $\sim$410MB/s

Yuan Zu and Bei Hua 2014 (2): In this paper the authors develop a new algorithm GLZSS which is a variant of LZSS(Lempel-Ziv-Storer-Syzmanski) and an improvement over CULZSS(3) which is itself an implementation of LZSS. GLZSS is developed in the CUDA framework. The authors identify the drawbacks of CULZSS which include poor and sometimes wasteful utilization of GPU resources, small block sizes that compromise on the compression ratio and bad data structures used for determining the matches. GLZSS counters these drawbacks by using a hash table for its dictionary followed by significantly larger block size (64KB). It also eliminates path divergence in the code so that serialization in the code is reduced and transfers the responsibilty of executing the branch instructions to the CPU instead. Decompression is also implemented on the GPU with important information being contained in the header of the compressed file
The experiments are conducted using a NVIDIA dual-GPU GTX590(each with 512 cores, 1.2GHz clock)(Only 1 GPU is used) and AMD A8-3870 quad-core CPU (0.8GHz) using 13 files of different types of data and sizes(32 - 200MB). The GLZSS algorithm turns out to be up to 2.5 times faster than CULZSS with compression ratio improved by 1.2-2X and about 1.25 times faster than the CPU implementation when all 4 cores are used for compression and about 5 times faster when only a single core is used. The compression ratio is however inferior when compared to traditional algorithms like gzip. The speeds of decompression are however not higher than that of the CPU mutlicore implementation. It is about 1.2x slower.
Compression Speeds atttained: $\sim$176MB/s
Decompression Speeds attained: $\sim$135MB/s

Bryan Ching 2014 (4): The following thesis develops a new algorithm PLZ(Parallel LZ) which is a variant of the LZ77 algorithm developed in the CUDA framework for GPU's. Traditionally LZ77 uses a sliding window buffer and a short unfactored suffix. The algorithm has been modified such that the LZ factorization is done on the entire string rather than a single window. This has been achieved in three steps, first the suffix array is constructed for the input and the ANSV(all nearest smaller values) values for each index of the input are computed. Finally the ANSV values are used for the LZ factorization and the compression is done by dividing the input into chunks and using their appropriate ANSV values. Both the suffix array and ANSV array computations use parallel GPU solutions provided by Deo and Keely(5), and Shun and Zhao(6) re-

spectively.
The testing has been done on a NVIDIA GTX TITAN Black(2880 CUDA cores and 0.98GHz clock). The data used consists of 13 files from publicly availabe datasets with sizes ranging from 50KB - 100MB. The parallel implementation of suffix array sees an improvement of 5x over the serial CPU solutions. It is important to note that the suffix array construction takes upto 80% of the time of the PLZ algorithm. The algorithm as a whole has a performance improvement of 17x over serial LZ-OG, LZ-ANSV algorithms and 1.2x improvement over PLZ3 algorithm which is the implementation of LZ77 on multicore CPU's.
Compression Speeds attained: $\sim$30MB/s

Anders L. V. Nicolaisen 2013 (7): The thesis aims at developing a parallel version of the LZSS algorithm called CANLZSS which is an improvement over the existing CULZSS(3) as mentioned earlier. The code has been developed on the CUDA platform and the author presents 4 alternative techniques that can be used for determining the matches which include KMP(Knuth-Morris-Pratt) algorithm, hashtables, BST's(Binary Search Trees) and suffix trees, all of which provide an improvement over the sequential search algorithm of LZSS. As simple data structures need to be used for GPU's the algorithm uses KMP for string matching. As stated earlier CULZSS can output overlapping matches which need to be resolved by CPU and the author uses an improved bit packing mechanisms to reduce the load on the CPU.
Testing has been done on a nVidia GeForce GT 620M(96 CUDAcores and 0.625GHz clock) and Intel(R) Core(TM) i7 CPU (1.9GHz). The datasets used consists of 3 files(E.coli genome, Bible, a facts book)with filesizes in the range of 2-4MB. The algorithms shows improvements of 6x/2x, 63x/7x, 24x/6x on the three files over the serial CPU LZSS and CULZSS algorithms respectively.
Compression Speeds attained: $\sim$23MB/s

Ritesh A. Patel et al. 2012 (8) The following paper develops a parallel version of the Bzip algorithm for execution on GPU's The approach parallelizes the three stages of the algorithm: 1.The BWT(Burrows Wheeler Transform) which is used for generating a reversible re-ordering of strings such that the new format is easier for compression as it generally contains repeated runs of the same character. Merge Sort is implemented in parallel for recombining the transforms of individual blocks to construct the BTW of the entire input . 2. the MTF(Move to Front) transform improves the effectiveness for statistical compression techniques, in this case for Huffman encoding. Although the algorithm is serial, it is parallelized by performing the transform on chunks of the input string, followed by recombination to generate the original transform. Not much parallelization has been achieved in the Huffman encoding part and the construction of the tree remains largely serial in nature.
Testing uses a Intel Core i5 CPU (3.2GHz) and NVIDIA GTX 460(336 cores and 1.35GHz clock). Three files (linux source code and 2 wiki file dumps) with file sizes in the range of 100-200MB are used. The GPU version is about 2.78 times **slower** than the CPU implementation. Most of the time is consumed in the merging of blocks in the BTW transform due to repeated access to the global memory for sorting the strings lexicographically as well as lack of parallelism for constructing the huffman trees account for the decrease in performance of the parallel Bzip implementation.

L. Erdődi 2012 (9) The paper proposes 3 algorithms which are somewhat a gradual improvement over one another for developing a parallel version of the LZO algorithm for GPU's. The algorithms stress on accessing the global memory in a coalesced fashion in order to ensure parallelism in the threads. Not many implementation details are given about the GPU algorithm
Experimentation has been done using NVIDIA GTX580(512 cores and 1.54GHz clock) and Intel Core i7-2600 CPU (3.4GHz clock) have been used for testing and no information is provided regarding the files used for compression. The paper claims to have attained up to 1.2x speedup when compared to the CPU implementation. Results are shown for decompression on GPU as well but once again no information is provided regarding the decompression algorithm
Compression Speeds attained: $\sim$900MB/s
Decompression Speeds attained: $\sim$1350MB/s

R.L. Cloud et al. 2011 (10): The paper attempts to develop a parallel algorithm for huffman compression from previous literature cited in the papers(11),(12) and simplifies the output format of the huffman compression so that the variable length encoding can be accounted for and decompression is made easier for parallelization. This decomposition of the compressed data into fixed size blocks has also been cited for use in the paper of Ritesh Patel et al(8) although it does not make use of the parallel huffman code generation algorithm that this paper mentions.
The testing is done on NVIDIA GeForce GTX 285 (240 cores and 1.5GHz clock) and Intel Core i7 Extreme Edition 965(quad core and 3.2GHz) and no information is provided on the testdata that has been used for the evaulation. The paper claims speedups of about 1.6x and 2x for compression and decompression respectively over the CPU implementations
Compression Speeds attained: $\sim$500MB/s
Decompression Speeds attained: $\sim$300MB/s

2.2 Parallel Soultions for CPU's

Jason Kane and Qing Yang 2012 (13): The paper deals with the parallelization of the LZO algorithm for CPU's. Several optimizations have been proposed that make use of the CPU architecture for improving the baseline performance of the LZO algorithm itself as well as optimizations for speeding up the algorithm at the cost of the degrading the compression ratio. These optimizations would include using SIMD instructions for copying the unmatched data, increasing the offset to search for matches, forcing the matches and data reads to be cache alingned and parallelizing the compression of blocks by using the multicore system.
Testing has been done a Intel 3.4GHz i7-2600 (4 physical cores) CPU with 8 different files of sizes in the range of 1-40GB and the results show that near linear speedup of 3.9x is achieved for the multi-core version of LZO and when the optimizations are added in the speed goes up to 5.4x times the serial code but the compression ratio is affected adversely and is about 1.4-2.3 times larger than the original compressed size
Compression Speeds attained: 1.5GB/s(3.9x Speedup)

Jeff Gilchrist 2003 (14): A parallel version of the Bzip algorithm called Pbzip is developed by the author which is an implementation of the bzip alogorithm for taking advantage of multiple processors. The algorithm works by dividing the input into smaller blocks and each block is handled by a separate processor and the paper also talks about implementing a parallel sort for improving the speeds of the BWT but has not been implemented.
The algorithm has been tested on different processors and a near linear speedup is attained on all multicore processors.

After looking at the literature we can conclude that not all types of algorithms are suitable for compression on GPU. Some algorithms like Bzip for example can attain much higher speedups with relatively less effort on CPU's but fail to improve performance even after a lot of optimizations on GPU's. Therefore it is better to optimize the algorithm for better utilization of resources on the GPU rather than trying to construct a new algorithm.
Another important point to note is the compression speeds attained. Although some algorithms might claim large magnitudes of speedups it is important to notice the absolute values of the compression speeds and ratios that are achieved as these are a function of many variables which include the type of processor/GPU's, files upon which the algorithms are run. The compression speed provided above are mostly approximate values.

CHAPTER 3

GPU ARCHITECTURE

In this chapter we will be having a brief overview of the important aspects of the GPU architecture which will give us a better idea on how to modify the LZO algorithm in order to improve the utilization of resources in a GPU.

Although GPU's were originally designed to for image processing and computer graphics where large blocks of data could be processed in parallel there are many other areas where the computation power of the GPU can be put to use. GPGPU's(General Purpose GPU's) have become popular wherein one can run a set of specified functions(kernels) on the GPU's which generally involve massive vector operations inoder to obtain an improvement in the performance. GPU's are suited for computations that exhibit data-parallelism which would exploit the SIMD(Single Instruction Multiple Data)architecture of the GPU and are not very efficient in executing instruction that have a lot of branches. It is better to execute such instructions on a CPU instead.

In order for the CPU to issue instructions and communicate with GPU's, c API's such as CUDA or OpenCL are used. CUDA is designed specifically for nVidia GPU's whereas openCL is designed for a wide range of devices including CPU's, GPU's, FGPA's etc. In the project we will be designing our implementation in OpenCL.

In the coming sections we will mainly be focusing on the organization of memory and execution model of the GPU.

3.1 Execution Model

In the OpenCL platform model the host(CPU) has access to a set of compute devices(GPU's). Each compute device consists of a set of compute units(SMP's) and each compute unit in turn is a collection of processors (CUDA cores). The host code sends commands for execution on the compute devices. Generally serial code is executed on the CPU and the GPU is used when parallel code need to be executed.

Specified functions called kernels can be executed on the GPU devices. Each kernel can be executed on a 1/2/3 dimensional computation domain. The kernel operates on a set of global work items(CUDA cores) and these work items can be grouped together into work groups. One or more work groups can execute concurrently on a compute unit(SMP's) depending upon the no of work items it holds and its memory requirements. The threads in a work group can communicate with one another with the help of local/shared memory but cannot directly interact with threads from other work groups(Global memory should be used in order to attain inter-workgroup communication). At any point of time only a single kernel can be executed on the GPU device.

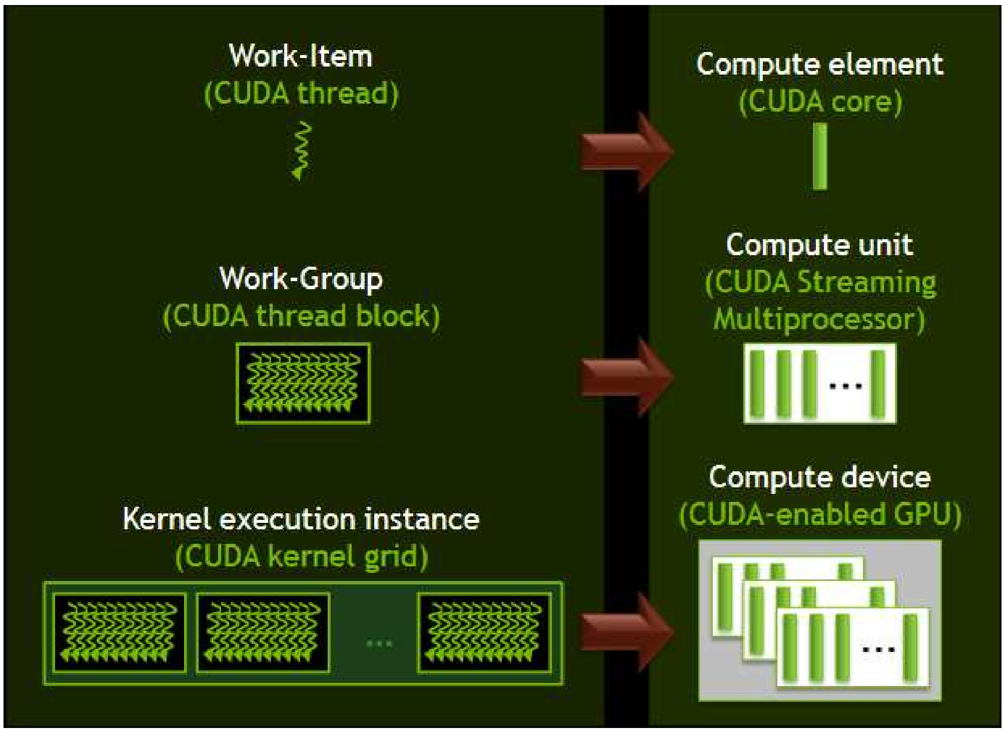


Figure 3.1: Mapping the exection model to the OpenCL Platform model(15)

3.2 Memory model

There are different layers of memory that are present in the GPU architecture each with different latencies and capacities. The OpenCL platform identifies four main types of memory in the compute devices.

Private memory is per work item and constitutes the registers of the particular work item. It is extremely fast and takes only a single cycle for accessing.
Local memory is shared by all the work items in a work group and is implemented on the chip and although slower than private memory it is much faster than the global memory and can be used for effective communication within the work group. However the amount of local memory is limited and should be used conservatively. In order to improve the performance of the local memory the memory access should be done in such a way that there are minimal bank conflicts i.e. threads should not try to read/write to consecutive addresses at the same time. This would result in serialization of code unless all the threads are reading the same memory location in which case the value would only be read once and broad casted.
Global memory is the main memory unit of the GPU to which all the work items have access. The CPU communicates with the GPU using Global memory and it has a large capacity in the order of GB's but is extremely slow, about a 100 times slower than the local memory and therefore one must try to minimize access to global memory as much as possible. In order for obtaining the best performance results memory access to global memory must be coalesced i.e. consecutive memory address must be accessed whenever possible at a given time. Notice the sharp difference in enhancing the performance of global and local memory.
Constant memory is also a part of the global memory with similar characteristics but has read-only permissions and is slightly faster than global memory.

The figure below gives a visual description of how the memory levels are arranged in a GPU and their interaction with the host device.

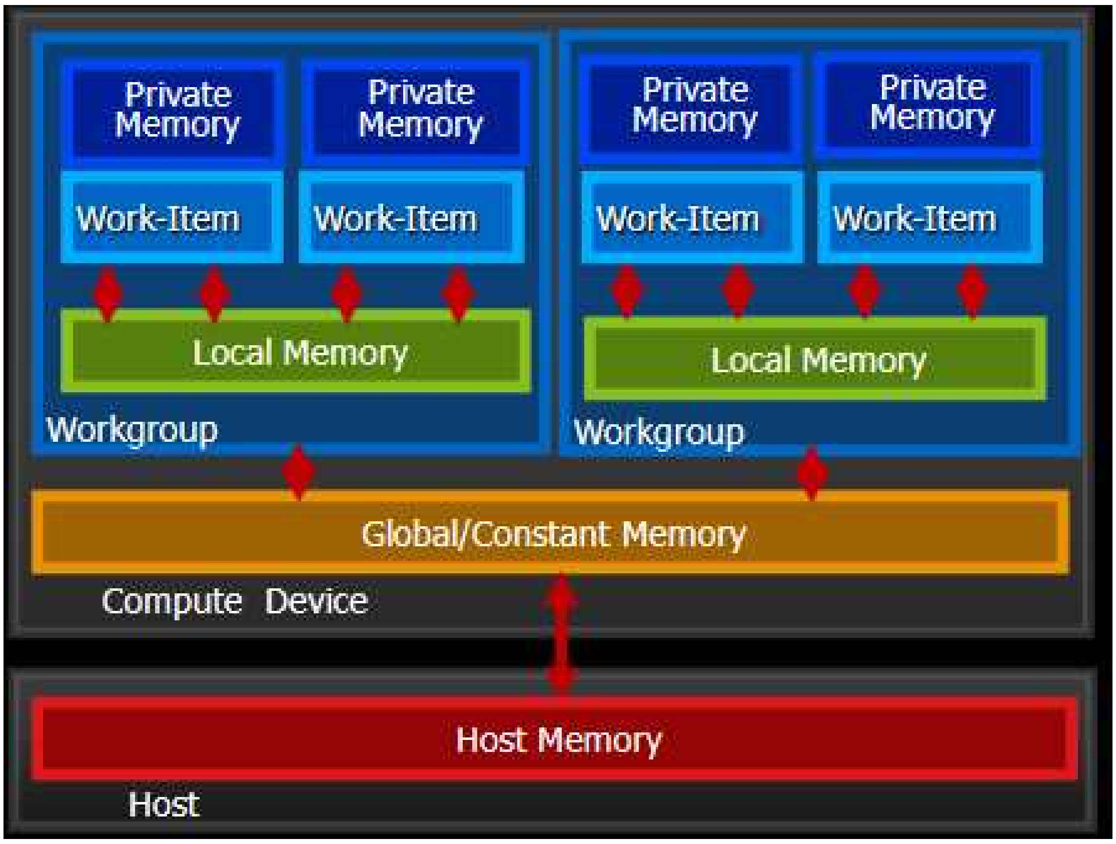


Figure 3.2: Memory model of a GPU(15)

CHAPTER 4

THE LZO ALGORITHM

In the following chapter we will be having a detailed description of the LZO compression and decompression algorithm. The lzo library consists of a set of algorithms that are specialized for attaining high compression and decompression speeds while compromising on the compression ratio and is one of the fastest algorithms around(16).
LZO has relatively simple operations in its algorithm and the most complex instructions would be the multiplication of two 32 bit integers. The algorithm in binary format is quite small and can fit into the level 1 instruction cache of modern computers.

Out of the different version of the lzo algorithms available in the lzo library the lzo_1x_15 algorithm has the highest compression speeds and we will be studying this algorithm.

4.1 Compression

As the algorithm is a dictionary based one it tries to reduce the length of the files by replacing repeated occurrences of strings with tokens which give the match length and offsets for its previous occurrence. The compressed file can be expressed as a regular expression shown below:

$$[\{(\text{Token}_1)(Data)\}?(Token_2) \mid (Data)(Token_2)]^{+}..\ \text{F1}$$

Here **Token$_1$** is used for encoding the length of the Unmatched Data. the size of token1 varies linearly with the length of Unmatched Data. **Data** corresponds to the Unmatched data and copied as it is from the input file. **Token$_2$** contains information regarding the offset(location) of the match and the length of the match. Unlike Token$_1$ the size of Token$_2$ for storing the offset is limited to 2/3 bytes
In the regular expression we notice that sometimes Token$_1$ and Data might be absent. This scenario occurs when a new match occurs immediately after the old match ends

and therefore there would be no data to be copied and $\text{Token}_1$is rendered useless.
The second scenario where this can occur is when the size of the Unmatched data is very small and therefore instead of wasting space for Token_1 the information is encoded into some free space present in Token_2 of the previous entry.
More details can be found in the pseudo code of the algorithm given at the end of the section.

LZO is essentially a block compression algorithm i.e. compression is performed on one block at a time and as mentioned earlier since the size of Token_2 is fixed only a limited amount of data can be encoded for the match length and offset. Therefore in order to effectively utilize the bytes the max offset that can be encoded is 49,152 and therefore the block size is set to 48KB(0xC000).
Now we shall have a look at the core compression process of LZO.

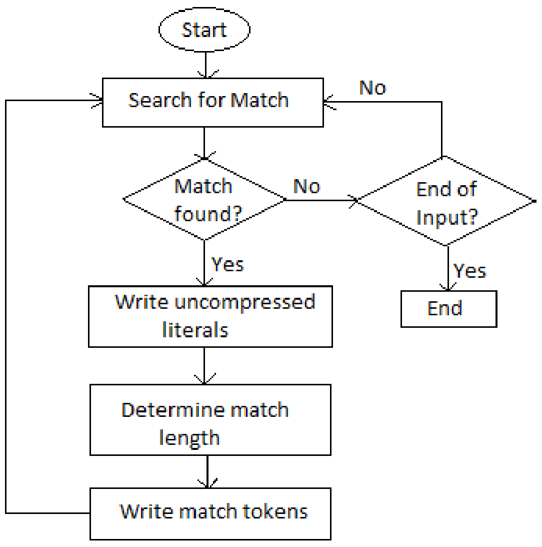


Figure 4.1: Control Flow of LZO(13)

As we can see in the figure the compression consists of 4 major steps.

1. Searching for a match
2. Output unmatched data
3. Determine match length
4. Write match tokens

We shall now examine each of these steps in detail.

4.1.1 Searching for a match

The LZO algorithm assumes that data exhibits spatial locality i.e. same data is often located nearby to one another. The pointer to the data where testing for a match is done is determined according to the equation given below

$$\text{ip} \mathrel{+}= 1 + ((\text{ip}-\text{ii}) >> 5) \quad .. \text{F2}$$

Where ip is a pointer to the input data where the match testing has to be done and ii is a pointer(to the input data) that points either to the beginning of the file or just after the last determined match. Here we can observe that initially when ip = ii the pointer is incremented in steps of 1 until ip-ii < 32. From then onwards ip is incremented by 2 until ip-ii < 64 and now ip is incremented in steps of 3 units. We can observe that as no matches are found ip will start to grow exponentially as shown in the figure below.

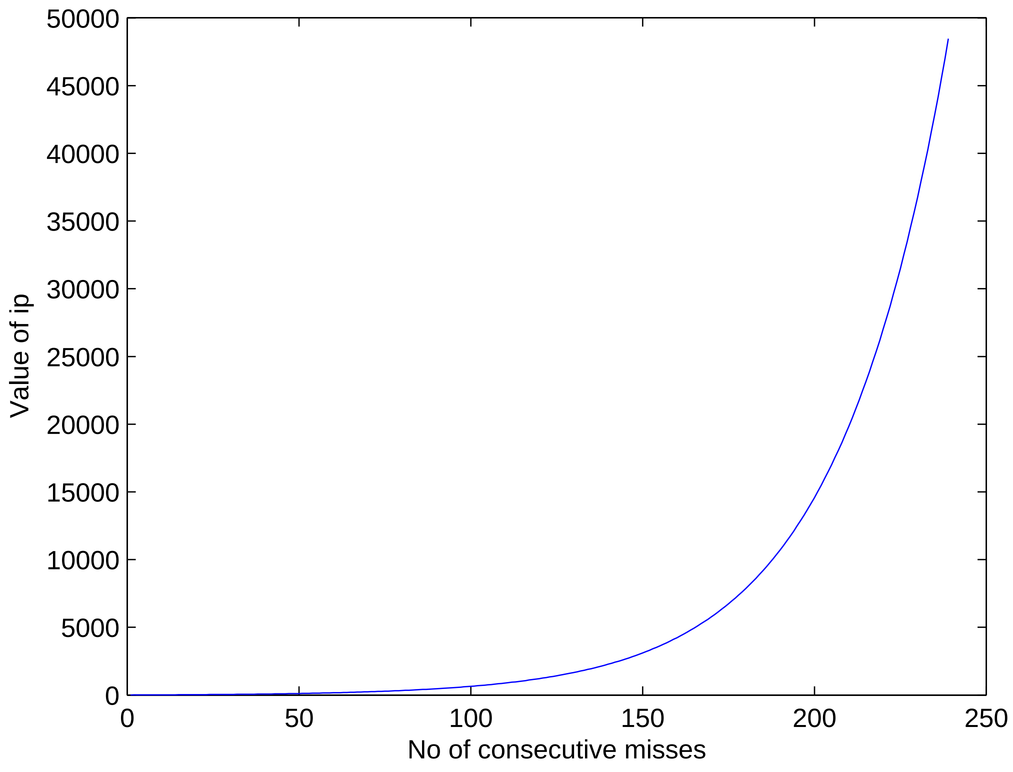


Figure 4.2: Exponential growth of the ip pointer

As we can see only within 240 consecutive misses ip gets incremented by 50,000 which is equal to the block size and therefore this indirectly accounts for the speed of the algorithm.

Now coming to the issue of match detection the algorithm uses a hash table implemented in the form of an array with 8192 entries. The key is determined my combining the first four characters starting from ip to form a 32 bit integer and multiplying it another 32 bit prime number. The last 13 bits of the product form the key and is used as an index into the hash table. The hash table entries store the offset of the data from the start of the block. Initially all the entries of the hash table are initialized to point to the start of the block, i.e. an offset of zero is stored. Multiplying with a large prime ensures that collisions in the hash table are not frequent but in case collisions do occur the values in the hash table are overwritten with the latest values.
The matching is done by computing the hash value, obtaining its corresponding value of offset from the hash table and checking if the current data and the data pointed by the hash table are the same or not. If a match is found or the end of input is reached the algorithm jumps to copying unmatched data, otherwise the offset value is updated in the hash table and ip is incremented according to the above equation.

4.1.2 Output unmatched data

Once a match has been determined now the unmatched data has to be written out to the output file. But first as discussed earlier the tokens need to be written which store information regarding the length of the unmatched data. The code for generating the tokens is given below.
Here **op** represents the pointer to the writing end of the file

```
len = ip-ii;            //length of unmatched data
if(len != 0){           //if len = 0, then no token is outputted
    if(len < 4){
        op[-2] = (op[-2] | len);
        //Store value of len in prev Token2
    }
    else{
        if(len < 19) *op++ = len-3;
        else{
            t = len - 18;
            *op++ = 0;
            while(t > 255){
                t -= 255;
                *op++ = 0;
            }
            *op = t;
        }
    }
}
```

Copying of data is done in either chunks of 1/4/8 bytes at a time depending upon which optimization is optimized for the CPU that it is operating upon.

4.1.3 Determining match length

String matching is done by comparing 8bytes from current input pointer and the hash table. XOR operation is performed and if the result is 0, the matching is continued, else it is stopped and and the exact match length is determined. Once again data can be fetched 1/4/8 bytes at a time depending upon the optimzations of the CPU.

4.1.4 Writing match tokens

Now we need to output the match tokens that store the match length and offset. The code given below explains the different types of tokens that the algorithm uses. **m_off** represents the offset and **m_len** represents the match length.

```
if(m_len <= 8 && m_off <= 0x800){
    m_off -=1;
    *op++ = ( (m_len-1 << 5) | ((m_off & 7) << 2) );
    *op++ = m_off >> 3;
}
else if(m_off <= 0x4000){
    m_off -= 1;
    if(m_len <= 33) *op++ = ( 32 | (m_len-2) );
    else{
        *op++ = ( 32 | 0 );
        m_len -= 33;
        *op++ = 0;
        while(m_len > 255){
            m_len -= 255;
            *op++ = 0;
        }
        *op++ = m_len;
    }
}
```

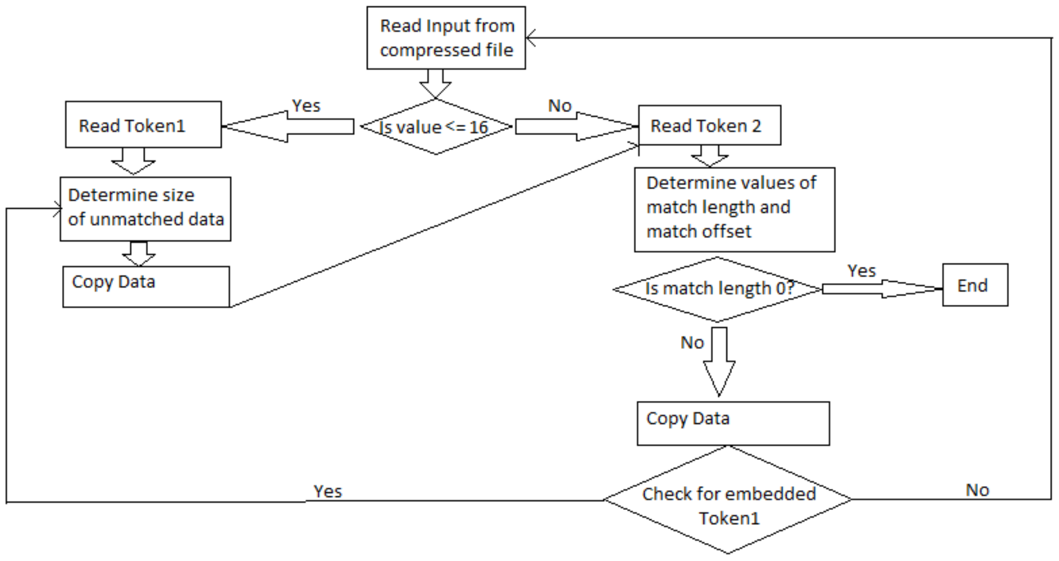
```
else{
    m_off -= 0x4000;
    if(m_len <= 9){
        *op++ = ( 16 | ((m_off >> 11) & 8) | (m_len-2));
    }
    else{
        *op++ = ( 16 | ((m_off >> 11) & 8) )
        m_len -= 9;
        while(m_len > 255){
            m_len -= 255;
            *op++ = 1;
        }
        *op++ = m_len;
    }
    *op++ = m_off << 2;
    *op++ = m_off >> 6;
}
```

Observe how in each case 2 bits are left unused in the 2nd byte from the end of the output. These bits are used for storing Token_1 in case the the length of the unmatched data is less than 4

4.2 Decompression

Once we have an understanding of how the tokens operate, decompression becomes a fairly trivial task. We can notice that Token_1 strings always start with a value < 16 and Token_2 strings start with a value >= 16. With the help of this information we can determine the length of the unmatched data appropriately if present and copy data directly from the compressed file to the output file.
Next the offsets and match lengths can be determined when we encounter Token_2 and this time the copying of data has to be done from within the output file generated till that moment using the offset and match length parameters obtained as the data to be written is already contained in the generated output.

The figure below gives a brief description of the working of the decompression algorithm.



Read Input from compressed file
Is value <= 16
Yes
No
Read Token1
Determine size of unmatched data
Copy Data
Read Token 2
Determine values of match length and match offset
Is match length 0?
Yes
End
No
Copy Data
Check for embedded Token1
Yes
No

Figure 4.3: Control Flow of LZO decompression

CHAPTER 5

PARALLELIZING LZO FOR GPU'S

We shall now be looking into the implementation details of the LZO algorithm on GPU architecture. We will be providing a summary of the optimizations and enhancements that were tried and their effects on the performance of the algorithm.

First we shall be dealing with the compression algorithm. We provide a list of optimization techniques each of which are a gradual improvement over the previous one and thereby steadily improve the performace. The basic GPU algorithm would have the CPU reading in the file, transferring the data to the global memory along with allocated memory for writing the output. After all the kernels have been executed the output is read back to the CPU and written to a file.

5.1 GPU Compression

Optimization#1: Parallelization of block compression This would be the most basic step in any parallelization technique of compression algorithms. The original LZO algorithm is serial in nature and therefore data is compressed only 1 block at a time. A GPU can overcome this limitation by simultaneously compressing a large number of blocks. However as LZO is a serial algorithm it ensures the continuity of the compression algorithm at the block boundaries, but this is not possible for implementation on GPU as we do not know beforehand how long the compression takes and would unnecessarily complicate the code. Therefore we simply append the token that signifies the end of compression in LZO for each block. This gives a small overhead for the compression ratio but as the block size is increased the effect is almost negligible. As we have modified the format of the compressed file, the original lzo decompressors will not be able to extract the file properly. Therefore a separate CPU, GPU implementation of the decompression algorithm is provided.

Optimization#2: Intra-Block Parallelization The next step would be to search for code snippets in the core compression algorithm where parallelization can be done. Here we can see that the writing of unmatched data literals into global memory can be parallelized. The same can be said about the code for determining the match length. In both of the above cases we can have individual threads writing to/reading from global memory and since different threads will be accessing consecutive elements of memory at a given time memory access is coalesced and execution happens in parallel.

Optimization#3: Hashtable .. Local vs Global memory Despite the above optimizations the GPU code is several orders slower than the CPU code. The main problem that was encountered was that we had used local memory for storing the hashtable entries. It was initially thought that since there would be frequent access to the entries of the hashtable along with its entire initialization which had to be done before at the beginning of the code it would be a good idea to use local memory as it is almost 100 times faster than global memory. The main issue with the above approach is that local memory is limited and a 8192 entry hashtable takes up 16KB for each block. Since the total amount of local memory that can be allocated is limited the maximum no of blocks that are executed in parallel is also reduced and this results in an under-utilization of the resources of the GPU. Reducing the size of the hashtable gives significant improvements in the running time but the compression ratios are adversely affected. Now we are presented with an option of either choosing a better hash function for improving the hit ratio or shifting the hashtable to global memory. The first idea is not very effective as the block consists of 49,152 entries which implies that there are 12,288 words and although not all words will be read into the table we can see that there will inevitably be conflicts as the size of the hasttable is reduced despite the hash function used.
Shifting the hasttable to global memory provides a very good speedup as more blocks can now be executed simultaneously and as mentioned above the initialization of the hashtables can also be done in parallel by coalescing memory accesses.

Optimization#4: Prefetching Input strings Another optimization that can be applied on the global vs local memory problem. We notice that while calculation hash values we form a word by combining 4 consecutive bytes from the input pointer. In the initial stages when the pointer is incremented by only 1 byte we will be repeatedly accessing the same data and since we are dealing with global memory large latencies are involved. We solve the problem by using local memory where the first 64 words are loaded into local memory. Once again we can make use of the threads in the work group to load the data in parallel effectively making it into a single memory access and for the rest of 32 iterations we can use the precomputed word values rather than accessing global memory. This optimization effectively replaces 4 global memory accesses with a single local memory access. Also the equation used for updating the search pointed is changed to:

ip += 1 + ((ip-ii)> >6) .. (F3)

which effectively increments the search pointer by 1 byte until the first 64 attempts. This helps in increasing the compression ratio but slightly degrades the compression speeds. Including all these changes in the code cuts down on the running time of the code by 7% for optimizing just a single instruction.

Optimization#5: Precomputing Hashtable entries This optimization is used in combination with prefectching input strings. Here we basically compute the hash values and the hastable entries for all the words that have been obtained earlier and populate the hasttable using these values. This allows us to introduce some degree of parallelism while searching for a match. Since hash tables are populated in advance there can be a scenario where a match is found such that the offset would be in the forward direction instead of backwards. In this case we simply interchange the match and search pointers such that the correct direction of offset is attained. An important point to note here would be that this will not always produce the same output file as we cannot exactly determine when the threads complete execution. In cases where two differnet words have the same hash value, only one of the pointers would be stored in the hashtable and it is not possible to determine which will be stored. This would be responsible for the randomness in the compressed file but all the generated outputs can be decompressed back into the original file without any errors. While writing code one should ensure that the the hashtables are populated in such a way that words closer to the current search pointer have their entries filled last so as to reduce the amount of unmatched data in case a match occurs. The main reason for the degradation of the compression ratio is that whenever a match is found in the forward direction the match pointer is advanced and the data that is overseen might have considerable redundant data which gets directly written to the output. This optimization also decreases the running time of the algorithm by 7% of its previous values and decreases the compressibility of a file.

The final Optimization presented uses a CPU-GPU hybrid algorithm for further enhancing performance

Optimization#6: The Hybrid Algorithm As we have seen earlier, the lzo algorithm has when implemented on the GPU has a significant number of branch instruction that are present while constructing the output tokens. This is believed to affect the performance of the algorithm as GPU is more suited towards executing SIMD type instructions rather than branch instructions. Therefore we decide to share the responsibility between the CPU and GPU where GPU executes the code that searches for matches and CPU performs the construction of the compressed file. The GPU determines the matches and communicates with the CPU using three arrays: Literals, Matchlens, Offsets that store the length of the unmatched data, matched data and the offset for each token pair format as shown above at (F1) This data is then directly used by the CPU to copy data from the input file and construct tokens appropriately. However we do not see the expected performance gain as the running time is almost double compared to the standalone GPU code. Although we achieve a 14% decrease in the running time for the GPU code the overhead involved with transferring data and CPU execution outweighs the benefits achieved. Therefore it is more economical to use either the CPU/GPU version rather than a hybrid algorithm.

5.2 GPU Decompression

An attempt has been made at trying to develop a GPU version of the compression algorithm but experimentation has shown that it is several orders of magnitude slower than the CPU implementation.
This is due to the extremely serial nature of the decompression algorithm. The main disadvantage is that the compressed file cannot be split into chunks so that decompression can be done in parallel as we do not know that exact boundaries of the blocks that were compressed earlier.
Therefore the GPU code that has been used is once again a hybrid algorithm where the CPU calculates the values of the match lengths and offsets. In order to ensure that both the GPU and CPU are working simultaneously a lightweight kernel was implemented whose primary objective was to copy the unmatched and matched data from the compressed file with the parameters generated by the CPU in each iteration. However this approach has a lot of drawbacks as one can see. The length of the unmatched, matched data generated for each token is generally quite small not exceeding a 100 characters in most cases and using a kernel for only copying data does not give us any additional advantage over copying data while using a CPU itself and additional overheads involved with calling a kernel make the scenario even worse.
While the copying of unmatched literals can be parallilized in the GPU the copying of matched data cannot be done so because data is copied from the generated output file itself and therefore the amount of data present in the output file becomes a limiting factor rather than the number of available threads for parallelization.
Although there are techniques by which these problems can be overcome we are quite skeptic that they would perform better than the CPU implementation. Details regarding these optimization are provided in the Future Works chapter and have currently not been implemented in the source code.

CHAPTER 6

EXPERIMENTS AND RESULTS

The experimental setup uses a NVIDIA Tesla K40c GPU(2880 CUDA cores across 15 SMP's, 745MHz Base Clock) and the processor is a Intel(R) Xeon(R) CPU E5-2650 v2 (8 cores and 2.60GHz Base clock).

The dataset used consists of 11 files from different backgrounds. The files include text files, dictionay, xml and html files, binary images and a scanned PDF. Most of these files have been obtained from the Mazini Corpus(17) and the Silesia Corpus(18). The file sizes vary from around 8MB - 150MB.
The Results Obtained are shown below.
We have split the running times of the algorithms on the GPU and CPU into the following sections:

1. Reading the Input File into CPU Main Memory (T_{Read})
2. Transferring Input Data into GPU Buffer ($T_{\text{CPU->GPU}}$)
3. Creating Output Buffers + Core Compression function (T_{core})
4. Reading Output Buffers from GPU ($T_{\text{GPU->CPU}}$)
5. Writing File (T_{Write})

There are additional timing parameters involved in the algorithm that include general initializations and freeing up of the parameters once the work has been done. These are not taken into account for both the CPU and the GPU implementation as when compression is performed on a large no of files at once the overhead involved in such initializations can be ameliorated.

We will be using two main measures of the compression speeds:

1. $\text{Speed}_{\text{core}}$: $\frac{SizeofInputFile}{T_{\text{core}}}$
2. $\text{Speed}_{\text{net}}$: $\frac{SizeofInputFile}{(T_{\text{Read}}+T_{\text{CPU->GPU}}+T_{\text{core}}+T_{\text{GPU->CPU}}+T_{\text{Write}})}$

Finally Compression Ratio(CR) is measured as follows: $\frac{SizeofCompressedFile}{SizeofCInputFile} * 100$
Therefore smaller the number, the better is the compression.

For comparison purposes we use 2 versions of the GPU algorithm: With Optimization#5 and without it. Optimization#6 has not been tested as it did not achieve the expected performance gain.
For each file we have 3 entries that correspond to:

1. GPU lzo without Opt#5
2. GPU lzo with Opt#5
3. CPU lzo

File sizes are given in MB, Timing parameters are given in ms and Speeds are given in the order of MB/s.

Different Parameters have been tested for the blocks to be compressed on the GPU such as the number of threads per block, block size, no of workgroups to be launced in each iteration. The current results have been obtained with block size set to 49,152, with 32 threads per work group and upto 1200 work groups are launched in each iteration. A lookahead of 128 bytes has been maintained for optimizations#4 and #5.

Filename	Size	T_{Read}	$T_{cpu\text{->}gpu}$	T_{core}	$T_{gpu\text{->}cpu}$	T_{Write}	$Speed_{core}$	$Speed_{net}$	CR
Random Words	108.35	53.16	59.81	597.07	59.15	90.66	181.47	126.01	77.59
		53.90	59.29	533.66	58.47	95.51	203.31	145.06	88.49
		48.85	-	447.62	-	969.60	242.06	73.90	77.92
arabic .pdf	150.07	68.77	81.77	111.20	81.57	145.68	1349.5	306.89	100.30
		65.73	82.36	117.52	82.39	129.81	1276.9	314.08	99.71
		71.60	-	62.93	-	1333	2384.7	102.26	100.30
etext99	100.4	45.80	54.87	442.50	54.53	81.89	226.89	147.77	61.24
		51.47	54.28	443.70	62.49	93.55	226.27	142.31	78.82
		47.0	-	396.02	-	692.48	253.52	88.42	61.31

howto	37.59	17.33	20.69	167.69	18.61	30.5	224.16	159.14	51.81
		17.74	21.15	155.0	18.77	30.68	242.51	154.47	70.24
		21.30	-	138.27	-	193.44	271.86	106.48	51.85
jdk13c	66.50	30.49	35.95	181.47	39.60	19.92	366.45	216.30	21.25
		28.99	36.20	180.32	35.03	26.88	368.78	216.32	37.66
		36.31	-	108.14	-	213.96	614.94	185.54	21.28
rctail96	109.40	49.98	60.08	354.43	58.76	39.58	308.66	194.37	33.85
		46.98	60.25	344.10	59.40	57.17	318.02	192.64	47.66
		59.11	-	243.65	-	431.94	449.0	148.90	33.89
rfc	111.03	50.49	60.19	381.07	60.17	49.98	291.36	184.46	40.19
		48.02	60.82	364.23	60.43	60.91	304.83	186.79	59.06
		57.84	-	299.38	-	495.92	370.87	130.12	40.20
sprot32 .dat	104.54	47.52	57.61	407.43	55.89	51. 07	256.58	168.74	41.29
		52.55	57.12	367.99	64.69	58.12	284.08	174.10	54.53
		53.29	-	226.02	-	546.01	462.52	126.67	41.48
w3c2	99.37	44.45	54.49	286.11	53.28	35.86	347.31	209.55	25.19
		43.25	54.02	280.55	53.65	42.52	354.19	209.64	41.02
		53.78	-	181.60	-	191.65	547.19	232.70	25.25
webster	39.53	18.27	21.90	165.17	19.91	23.79	239.32	158.73	48.81
		20.58	21.79	156.37	22.23	25.52	252.79	160.37	68.82
		20.22	-	149.79	-	246.27	263.90	94.96	48.85
x-ray	8.08	4.64	4.25	54.41	3.97	13.45	148.50	100.09	99.49
		4.05	4.86	44.53	4.14	8.80	181.45	121.72	99.79
		4.65	-	4.27	-	125.78	1892.27	59.98	100.27

Table 6.1: GPU vs CPU: Timings, Speeds and Compression Ratios

Looking at the results we can conclude that there are some files on which lzo is unable to achieve any significant compression. These are mainly image based file (arabic.pdf and x-ray). The highest compression is achieved for xml, html files (jdk13c, rctail96, w3c2) followed by general text files. Out of the text files we can say that the more structured or less random the data is the better is the compression that can be achieved.

We can notice an anomaly in the running times of the CPU algorithm compared to the GPU version. The core compression algorithm is faster for the CPU version but for some reason the writing of the file seems to extremely slow and as a net result the GPU version is faster than the CPU. The code used for finding the running times of the CPU version has been included in the appendix for reference and testing purposes.
The highest compression ratio is attained for the GPU code with Optimization#4. This is only slightly better than that of the original lzo algorithm, because of the way the search pointer is updated in equation F3.However another anomaly is detected while compressing arabic.pdf where GPU code with Opt#5 has a better compression ratio that either of the other 2 algorithms.
GPU code with Opt#5 has a higher core compression speeds almost most of the time but is compromised on the compression ratio. Therefore this results in larger files being generated and the advantage is gained in the kernel execution is lost out at the writing of a larger file, especially in the case of highly compressible files. The core compression speeds are 1.06x faster on average and the Compressed files obtained are 1.34x larger than the ones attained without Opt#5.

Although the CPU core compression algorithm is significantly faster than that of the GPU kernel we will also be taking into consideration the time taken for reading and writing the files into consideration. Therefore we perform an analysis on the $\text{Speed}_{\text{net}}$ obtained. We observe that on an average the GPU code is about 1.5x faster than the CPU code with no loss in compression ratio. (Actually a very slight improvement).

CHAPTER 7

CONCLUSION

In the following project we have explored the LZO algorithm and examined several methods via which the compression speeds of the algorithm can be improved. We have tried to develop an Optimized algorithm for GPU's where slight improvements have been made in the copying of literal data, determining match lengths, and updating the search pointer for finding matches and all of this can be done without any loss in the compression ratio. This demonstrates the ability of GPU to perform on par with the CPU implementation. Overall the GPU algorithm has achieved a 1.5x speedup over the traditional serial algorithm. This shows that although there is not a phenomenal rise in the compression speeds, they are significant and therefore the GPU can be used alongside with the CPU as a co processor and can improve the overall throughput of the system. The GPU that we have used for testing uses has a 745MHz clock, and as faster GPU's arrive we can expect more and more speedup for the algorithm.
We can see that there is a lot of scope for the GPU for processing large amounts of data in parallel and when care has been taken to optimize data for GPU environments performance gains can be achieved.

7.1 Future Work

There is a lot of scope for further improvement in the algorithm. Not all the optimizations have been implemented in the code base and we shall be discussing some of them here. The current GPU that we have used has a preferred vector width for characters set to 1, had it been greater we would have exploited the GPU more as this would allow each thread to process more information with the same instruction set and make efficient use of the SIMD nature of the GPU. However one has to be careful while implementing these changes as OpenCL does not specify how these instructions are implemented and in some cases the execution can be done serially and not in parallel.
Other minor optimizations that can be done is to use the extension of the GPU available

for loop unrolling purposes etc. This would be more specialized towards the GPU being used as each GPU has a different set of extension that can be implemented.

Coming to the decompression algorithm many more optimizations can be tried out. In order to effectively parallelize the algorithm we will need prior information regarding the block boundaries in the compressed file. Therefore one can include some header information in the compressed file which gives us the length of the original file and the block boundaries so that each block can be decompressed independently.
As discussed in earlier sections since the writing of matched data is limited by the amount of data generated and not the number of threads being used, one can overcome this limitation by performing copying in serial until enough data has been generated and then use threads for copying data simultaneously. However the scenario where we have a small offset and a large match length is quite unlikely to be found in real world data unless we have repeating sets of a particular sequence of characters.
The hybrid algorithm for decompression can also be reimplemented wherein instead of using a kernel to decompress data for every token, we can use the CPU to generate token for an entire block and then hand it over to the GPU for decompression while the CPU can effectively work on generating tokens for another block while the previous one is being decompressed on the GPU. This solution can be used when we do not know the block boundaries in advance. This would reduce the overall overhead associated with generating kernels.

REFERENCES

[1] Funasaka, Shunji, Koji Nakano, and Yasuaki Ito.
Fast LZW compression using a GPU.
2015 Third International Symposium on Computing and Networking (CANDAR). IEEE, 2015.

[2] Zu, Yuan, and Bei Hua.
GLZSS: LZSS lossless data compression can be faster.
Proceedings of Workshop on General Purpose Processing Using GPUs. ACM, 2014.

[3] 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.

[4] Ching, Bryan.
Optimizing lempel-ziv factorization for the GPU architecture.
Diss. California Polytechnic State University San Luis Obispo, 2014.

[5] Deo and S. Keely.
Parallel suffix array and least common prefix for the GPU.
In Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming, pages 197-206. ACM, 2013

[6] Shun, Julian, and Fuyao Zhao.
Practical parallel lempel-ziv factorization.
Data Compression Conference (DCC), 2013. IEEE, 2013.

[7] Nicolaisen, Anders Lehrmann Vrønning
Algorithms for Compression on GPUs.
Technical University of Denmark, 2013.

[8] Patel, R. A., Zhang, Y., Mak, J., Davidson, A., & Owens, J. D
Parallel lossless data compression on the GPU.
IEEE, 2012.

[9] Erdodi, L
File compression with LZO algorithm using NVIDIA CUDA architecture.
Logistics and Industrial Informatics (LINDI), 2012 4th IEEE International Symposium on. IEEE, 2012.

[10] Cloud, R. L., Curry, M. L., Ward, H. L., Skjellum, A., & Bangalore, P
Accelerating lossless data compression with GPUs.
arXiv preprint arXiv:1107.1525 (2011).

[11] M. J. Atallah, S. R. Kosaraju, L. L. Larmore, G. L. Miller, and S. H. Teng.
Constructing trees in parallel.
In SPAA '89: Proceedings of the first annual ACM symposium on Parallel algorithms and architectures, pages 421–431, New York, NY, USA, 1989. ACM

[12] P. Berman, M. Karpinski, and Y. Nekrich.
Approximating huffman codes in parallel.
Journal of Discrete Algorithms, 5(3):479–490, 2007.

[13] Kane, Jason, and Qing Yang.
Compression speed enhancements to LZO for multi-core systems.
Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on. IEEE, 2012.

[14] Gilchrist, Jeff.
Parallel data compression with bzip2.
Proceedings of the 16th IASTED international conference on parallel and distributed computing and systems. Vol. 16. 2004.

[15] Website:
`http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2011-04-14/06-intro_to_opencl.pdf`

[16] LZO Website:
`www.oberhumer.com/opensource/lzo/`

[17] MaZini Corpus:

`http://people.unipmn.it/manzini/lightweight/corpus/`

[18] Silesia Corpus:

`http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia`

APPENDIX A

LZO code

```
#include <lzo/lzoconf.h>
#include <lzo/lzo1x.h>
#include <sys/time.h>

#define USE_LZO1X 1

#define PARANOID 0


/* portability layer */
static const char *progname = NULL;
#define WANT_LZO_MALLOC 1
#define WANT_LZO_FREAD 1
#define WANT_LZO_WILDARGV 1
#define WANT_XMALLOC 1
#include "examples/portab.h"


/***********************************************************************

//
************************************************************************


int __lzo_cdecl_main main(int argc, char *argv[])
{
    struct timeval begin, checkpoint[10];
    gettimeofday(&begin, NULL);
```

```
int r;

lzo_bytep in;
lzo_uint in_len;

lzo_bytep out;
lzo_uint out_bufsize;
lzo_uint out_len = 0;

lzo_voidp wrkmem;
lzo_uint wrkmem_size;

lzo_uint best_len;
int best_compress = -1;

lzo_uint orig_len;
lzo_uint32_t uncompressed_checksum;
lzo_uint32_t compressed_checksum;

FILE *fp;
const char *in_name = NULL;
const char *out_name = NULL;
long l;



lzo_wildargv(&argc, &argv);

printf("\nLZO real-time data compression library (v%s
    , %s).\n",
        lzo_version_string(), lzo_version_date());
printf("Copyright (C) 1996-2015 Markus Franz Xaver
   Johannes Oberhumer\nAll Rights Reserved.\n\n");
```

```
    progname = argv[0];
    if (argc < 2 || argc > 3)
    {
        printf("usage: %s file [output-file]\n", progname
            );
        exit(1);
    }
    in_name = argv[1];
    if (argc > 2) out_name = argv[2];

    gettimeofday(&checkpoint[0], NULL);
    printf("Tag Calculation: %f ms\n", (double)(
       checkpoint[0].tv_usec - begin.tv_usec)/1000 + (
       double)(checkpoint[0].tv_sec - begin.tv_sec)*1000);

/*
 * Step 1: initialize the LZO library
 */
    if (lzo_init() != LZO_E_OK)
    {
        printf("internal error - lzo_init() failed !!!\n
            ");
        printf("(this usually indicates a compiler bug -
            try recompiling\nwithout optimizations, and
            enable '-DLZO_DEBUG' for diagnostics)\n");
        exit(1);
    }

    gettimeofday(&checkpoint[1], NULL);
    printf("LZO Initializations: %f ms\n", (double)(
       checkpoint[1].tv_usec - checkpoint[0].tv_usec)/1000
        + (double)(checkpoint[1].tv_sec - checkpoint[0].
       tv_sec)*1000);
```

```
/*
 * Step 2: allocate the work-memory
 */
    wrkmem_size = 1;
    wrkmem_size = (LZO1X_1_15_MEM_COMPRESS > wrkmem_size)
        ? LZO1X_1_15_MEM_COMPRESS : wrkmem_size;
    wrkmem = (lzo_voidp) xmalloc(wrkmem_size);
    if (wrkmem == NULL)
    {
        printf("%s: out of memory\n", progname);
        exit(1);
    }

/*
 * Step 3: open the input file
 */
    fp = fopen(in_name,"rb");
    if (fp == NULL)
    {
        printf("%s: cannot open file %s\n", progname,
           in_name);
        exit(1);
    }
    fseek(fp, 0, SEEK_END);
    l = ftell(fp);
    fseek(fp, 0, SEEK_SET);
    if (l <= 0)
    {
        printf("%s: %s: empty file\n", progname, in_name)
           ;
        fclose(fp); fp = NULL;
        exit(1);
    }
    in_len = (lzo_uint) 1;
    out_bufsize = in_len + in_len / 16 + 64 + 3;
    best_len = in_len;

/*
 * Step 4: allocate compression buffers and read the file
 */
    in = (lzo_bytep) xmalloc(in_len);
    if (in == NULL)
    {
        printf("%s: out of memory\n", progname);
        exit(1);
    }
    in_len = (lzo_uint) lzo_fread(fp, in, in_len);
    printf("%s: loaded file %s: %ld bytes\n", progname,
       in_name, (long) in_len);
    fclose(fp); fp = NULL;

    gettimeofday(&checkpoint[2], NULL);
    printf("Reading input file to CPU: %f ms\n", (double)
       (checkpoint[2].tv_usec - checkpoint[1].tv_usec)
       /1000 + (double)(checkpoint[2].tv_sec - checkpoint
       [1].tv_sec)*1000);

/*
 * Step 5: compress from 'in' to 'out' with LZO1X-1-15
 */
#ifdef USE_LZO1X
        out = (lzo_bytep) xmalloc(out_bufsize);
        if (out == NULL)
        {
        printf("%s: out of memory\n", progname);
        exit(1);
        }
        out_len = out_bufsize;
        r = lzo1x_1_15_compress(in,in_len,out,&out_len,
           wrkmem);
        if (r != LZO_E_OK)
        {
            /* this should NEVER happen */
            printf("internal error - compression failed:
               %d\n", r);
            exit(1);
        }
        printf("LZO1X_1_15: %8lu -> %8lu\n", (unsigned
           long) in_len, (unsigned long) out_len);

#endif /* USE_LZO1X */

    gettimeofday(&checkpoint[3], NULL);
    printf("Output Buffer creation + Execution: %f ms\n",
        (double)(checkpoint[3].tv_usec - checkpoint[2].
       tv_usec)/1000 + (double)(checkpoint[3].tv_sec -
       checkpoint[2].tv_sec)*1000);
    if (out_name && out_name[0])
    {
        printf("%s: writing to file %s\n", progname,
           out_name);
        fp = fopen(out_name,"wb");
        if (fp == NULL)
        {
            printf("%s: cannot open output file %s\n",
               progname, out_name);
            exit(1);
        }
```

```
            if (fwrite(out, 1, out_len, fp) != out_len ||
                fclose(fp) != 0)
            {
                printf("%s: write error !!\n", progname);
                exit(1);
            }
        }
        gettimeofday(&checkpoint[4], NULL);
        printf("Writing compressed file: %f ms\n", (double)(
            checkpoint[4].tv_usec - checkpoint[3].tv_usec)/1000
             + (double)(checkpoint[4].tv_sec - checkpoint[3].
            tv_sec)*1000);

/*
 * Step 6: verify decompression
 */
#if PARANOID
        lzo_memset(in,0,in_len);      /* paranoia - clear
            output buffer */
        orig_len = in_len;
        r = -100;
#ifdef USE_LZO1X
        if (best_compress == 1)
            r = lzo1x_decompress_safe(out,out_len,in,&
                orig_len,NULL);
#endif
#ifdef USE_LZO1Y
        if (best_compress == 2)
            r = lzo1y_decompress_safe(out,out_len,in,&
                orig_len,NULL);
#endif
        if (r != LZO_E_OK || orig_len != in_len)
        {
```

```
        /* this should NEVER happen */
        printf("internal error - decompression failed: %d
            \n", r);
        exit(1);
    }
    if (uncompressed_checksum != lzo_adler32(lzo_adler32
        (0,NULL,0),in,in_len))
    {
        /* this should NEVER happen */
        printf("internal error - decompression data error
            \n");
        exit(1);
    }
    /* Now you could also verify decompression under
        similar conditions as in
     * your application, e.g. overlapping assembler
         decompression etc.
     */
#endif

    lzo_free(in);
    lzo_free(out);
    lzo_free(wrkmem);

    gettimeofday(&checkpoint[5], NULL);
    printf("Freeing Resources: %f ms\n", (double)(
        checkpoint[5].tv_usec - checkpoint[4].tv_usec)/1000
         + (double)(checkpoint[5].tv_sec - checkpoint[4].
        tv_sec)*1000);

    return 0;
}
```