CULZSS: LZSS Lossless Data Compression on CUDA

Adnan Ozsoy
Department of Computer & Information Sciences
University of Delaware
Newark, DE 19716
ozsoy@udel.edu

Martin Swany
Department of Computer & Information Sciences
University of Delaware
Newark, DE 19716
swany@cis.udel.edu

***Abstract*—Increasing needs in efficient storage management and better utilization of network bandwidth with less data transfer lead the computing community to consider data compression as a solution. However, compression introduces an extra overhead and performance can suffer. The key elements in making the decision to use compression are execution time and compression ratio. Due to negative performance impact, compression is often neglected.**

General purpose computing on graphic processing units (GPUs) introduces new opportunities where parallelism is available. Our work targets the use of the opportunities in GPU based systems by exploiting the parallelism in compression algorithms. In this paper we present an implementation of Lempel-Ziv-Storer-Szymanski(LZSS) lossless data compression algorithm by using NVIDIA GPUs Compute Unified Device Architecture (CUDA) Framework. Our implementation of the LZSS algorithm on GPUs significantly improves the performance of the compression process compared to CPU based implementation without any loss in compression ratio which can support GPU based clusters to solve bandwidth problems. Our system outperforms the serial CPU LZSS implementation by up to 18x, the parallel threaded version up to 3x and the BZIP2 program by up to 6x in terms of compression time, showing the promise of CUDA systems in lossless data compression. To give the programmers an easy to use tool, our work also provides an API for in memory compression without the need for reading from and writing to files, in addition to the version involving I/O.

***Keywords*-Lossless data compression, LZSS, GPU, CUDA**

I. Introduction

Making the best use of the expensive resources is crucial in high performance computing. Resources like memory, network bandwidth, or processing units are the key elements in achieving good performance and consumption of those resources needs to be carefully planned. Data compression helps to utilize space limited resources more efficiently. There are several algorithms on data compression; data deduplication, run-length encoding, dictionary coders [1], [2], Burrows-Wheeler Transform [3], statistical encoding [4], and others being used by programs to alleviate space usage. As nothing comes free, there are also some tradeoffs on the decision of using compression. One of the main issues is increase in running time. Obviously, additional computation results in longer total execution times.

The promising performance-per-dollar and performance-per-power ratios on non-graphic computations of GPUs has gotten the attention of many of those who were looking for affordable performance gains on data-parallel computations. General purpose GPUs (GPGPU) provide hundreds of cores that are programmable with NVIDIA's CUDA (Compute Unified Device Architecture) framework. This framework provides an API for programmers that exposes the underlying GPU architecture, which is a collection of virtualized SIMD processors and capable of efficiently switching between thousands of threads [5].

Since the advent of general purpose usage of GPUs, medical computing [6], [7], energy sciences [8], image/video processing [9], [10], finance [11] and many other problems in different areas have been ported on to GPU platforms in order to gain performance. In this paper we propose an implementation of Lempel-Ziv-Storer-Szymanski(LZSS) lossless data compression on NVIDIA GPUs (CULZSS). Our redesigned implementation for the CUDA framework aims to reduce the effect of compression time compared to CPU based compression implementations. Our work lets accelerators exploit the architectural strengths of GPUs.

The paper is divided into 8 sections: Section 2 gives background for the LZSS algorithm and GPU architecture. Section 3 describes the parallelism available in the algorithm and our implementation details. The performance analysis with benchmark setup details and results are represented in Section 4. Section 5 discusses the results and limitations. Section 6 describes the related work. Future work is in Section 7, and finally Section 8 concludes the paper.

II. Background

A. LZSS algorithm

The LZSS algorithm is a widely used compression algorithm which is implemented in several popular compression programs like PKZip, ARJ, LHArc, ZOO etc. with minor changes to the original algorithm. It is a variant of LZ77 [1] and a dictionary encoding technique with two buffers: a sliding window search buffer and an uncoded lookahead buffer [2]. Storer and Szymanski extended the work of Lempel and Ziv by using flags to indicate a coded or an uncoded character. This solution eliminated the redundancy

978-0-7695-4516-5/11 $26.00 © 2011 IEEE
DOI 10.1109/CLUSTER.2011.52

IEEE computer society

of LZSS which outputs an explicit character after each pointer, either coded or uncoded. This algorithm yields better compression ratio than LZ77 with the same computational complexity. The general algorithm is given in the following subsection.

1) Compressing strings: After initializing the buffers, characters are read from the input data to the uncoded data buffer. For every character in the uncoded buffer, the searching process looks for the longest substring in the search buffer that matches the lookahead buffer, starting with the first input character. If the match is long enough, then the program encodes the location and length of that substring into the output. If there is not enough match starting with the given input character, that character is written directly to output with a flag indicating no encoding was performed. The algorithm follows these steps until no characters are left. The minimum number of match is depending on the encoding of bits and in our case it is three. The encoding of two character match requires the same amount bytes if we directly output the two characters.

Here is an example to illustrate the encoding process:

Original	Encoded
0: I meant what I said	0: I meant what I said
20: and I said what I meant	20: and(12,7)(7,8)(2,5)
44:	30:
45: From there to here	31: From there to (51,4)
64: from here to there	47: f(46,4)(51,8)(50,5)
83: I said what I meant	55: (24,19)
Total characters 102	Total characters 56

Figure 1. LZSS Encoding Example

The left side is the original text with the total characters up to that line given in numbers at the beginning of each line. The encoded version of the text is on the right side. Encoding is shown with two numbers in parentheses. The first number in the parentheses is the offset and the second one is the length of the match.

2) Decompressing Strings: The decompression process is a straightforward decoding that involves read and write without any search. The encoding flags are read to find out which characters are being encoded. If a flag indicates encoded, the number of the characters and the starting position is gathered from the encoded part. Then the number of characters with the given position is written from the sliding window to the output file or memory. If it is not encoded, the character is output directly. Decompression consumes less memory resources and computing time compared to compression.

B. GPU architecture

GPUs are massively parallel computing units that offer high parallelism and memory bandwidth in a low cost, energy efficient platform [12]. GPUs employ significant multithreading. This is achieved by a set of multiprocessors, called streaming multiprocessors (SMs), that exist in GPU architecture. Each SM contains a set of SIMD processing units called streaming processors (SPs).

Within the new series of NVIDIA CUDA family called Fermi, there are up to 512 CUDA cores, which are organized in 16 streaming multiprocessors of 32 cores in each GPU [13]. There is barrier synchronization for inter-thread communication and a threading unit to schedule warps of threads. This enhanced multithreaded platform gives many opportunities for parallel computation.

NVIDIA GPU architectures implement a hierarchy of memory types; including global, constant, texture, shared memory and registers. Global memory, texture memory, and constant memory are accessible by all threads. Threads in the same thread block share the shared memory, and each thread has private registers and local memory. Because of the limited amount of shared resources (register and shared memory usages per thread block), it can be a limiting factor for CUDA programs and needs special attention to fully utilize the GPUs. [14]

Any code that needs to run on CUDA architecture is called kernel. Before and after the kernel execution, the memory needs to be explicitly copied to the GPU memory. There is an API with special function calls to communicate between the separate address spaces of host CPU and the GPU device.

CUDA allows developers to use C as a high-level programming language with the benefit of ease of programming in a familiar environment rather than learning a new programming language. This gives a motivation for porting already written programs in CUDA with minimal extensions and makes researchers curious about exploiting the massive parallel environment.

III. Implementation Details

The LZSS algorithm has been implemented in different versions. The serial CPU implementation of LZSS was mainly adapted from Dipperstein's work [15]. To be fair to the CPU implementation and give the opportunity to use parallelism, we also implemented a CPU threaded version of the LZSS algorithm using the POSIX threads. The CUDA version is adapted from the serial implementation. There are two different versions implemented in CUDA which differ in the distribution of the work on the computing units. We will discuss in detail further in this section.

Both CPU and CUDA versions are implemented considering only in-memory or with I/O capabilities. The in-memory compression is meant to work in applications that perform compression on the fly without involving any I/O. To be able to compare the results with well known compression utilities, an I/O version is also included.

In-memory compression gives a simple abstraction for users. The interfaces in Figure 2 are part of our compression library that take advantage of the installed GPU hardware and can be dynamically loaded. The library gets initialized

when loaded, detects GPUs, and determines capabilities on the system. Then, when Gpu_compress() is called, it takes the given buffer pointer and copies it to the GPU, compresses it into the given memory region, and returns the calling process a pointer to the compressed data and its length. The last parameters for the functions are compression parameters. Currently these parameters only include CULZSS version selection. In the future, window size and number of threads per block can be added. These parameters give the programmer the ability to find the best configurations for his/her dataset and work load.

```
Gpu_compress( *buffer, buf_length,
      **compressed_buffer,
      &comp_length,
      compression_parameters);

Gpu_decompress(*buffer, buf_length,
      **decompressed_buffer,
      &decomp_length,
      compression_parameters);
```

Figure 2. API Interface

From an application perspective, such as in a network application, the input data resides in a memory buffer that needs to be compressed at one gateway of the network and decompressed at the egress gateway, so the data looks the same going in as coming out.

The other version is the I/O version which is a standalone compression program. It follows the same flow except reading from and writing to the given files.

A. CPU implementation

The serial CPU implementation is a straightforward implementation of the algorithm. The algorithm is described in the background section. The threaded version exploits parallelism by using the independent behavior of the data processing in the algorithm. Each thread is given with some chunk of the file and the chunks are compressed concurrently. After each thread compresses the given data, individual compressed chunks are reassembled to form the final output. For this version, we used POSIX Threads.

B. CUDA Implementation

In the CUDA implementation we have decided to explore two different approaches. In the first version, the idea is very similar to Pthread implementation. It offloads the work into each thread in each block by giving them a small piece to work. In the second approach, we exploit the algorithm's SIMD nature to enhance parallelism suitable for CUDA architecture. The work that is distributed among block threads is the matching phase of the compression for a single block (Figure 3).

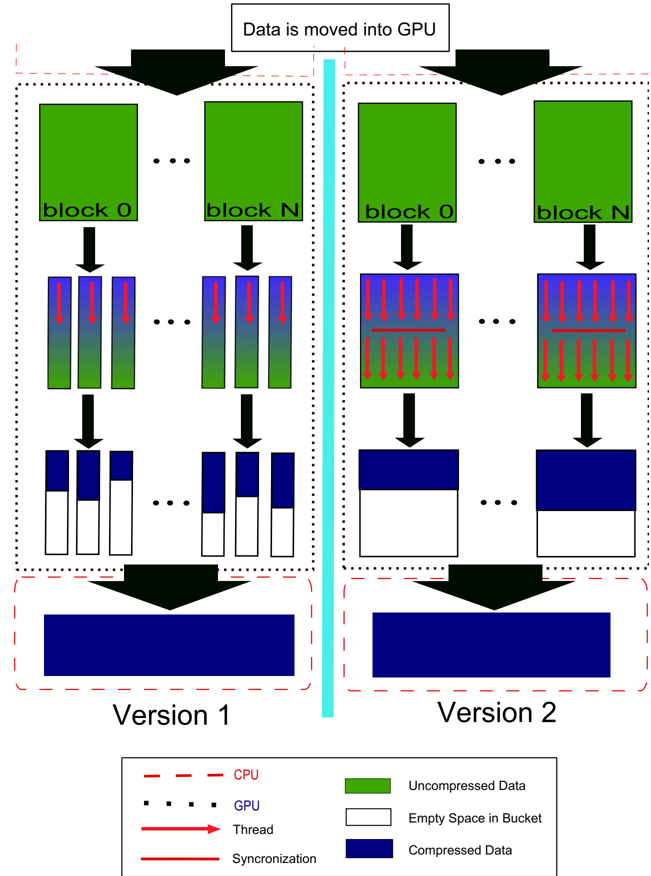

Figure 3. Overview of the two CULZSS Versions

1) CULZSS Version 1 : In the background section, LZSS algorithm steps show that the compression of characters is not dependent if the two starting characters of the substrings are not in the range of the same search buffer length. Because of this independent character of the algorithm, we can divide the input data into several blocks, and compress them in parallel.

The data is divided into chunks and distributed among blocks. Each block then splits this chunk into even smaller pieces. Each thread in the thread block receives a small portion of the input data and works on its own to compress that piece. This approach is used in parallelizing BZIP2 (PBZIP2) [16]. The compressed data is being overwritten onto each given chunk. After the compression is done, all the compressed chunks are combined sequentially and given back to CPU.

2) CULZSS Version 2 : In the second version we look into the algorithm to exploit parallelism if we can fully utilize the threads available on the GPU. CULZSS Version 1 makes use of the independent computational behavior among chunks. In this version, we observed that there is also independent computation in every single matching of the characters inside each chunk. The search for a substring match starts

with reading from the uncoded buffer and checks one by one with the window items. If there is a match, it checks the consequent items from each buffer to find a substring match. This process traces the whole window and the longest match is returned with the encoding information, length of substring and match offset. Item matching for each character does not depend on each other. Therefore, the matching computation can be done in parallel for each character in the uncoded lookahead buffer.

In the matching process each character is searched by a single thread throughout the window buffer. The search is linear, and each thread starts the search in the window buffer by an offset determined by the given thread id. This also gives an opportunity to avoid accesses with bank conflict to the shared memory by setting each thread with an offset of 4 characters(32 bytes) distance. After matching phase, uncoded lookahead buffer is needed to offload the encoded characters into the window buffer and load more from input data which resides in global memory. Loading to those buffers can also be done in parallel where each thread moves one item of the buffer. Coalescing access to global memory is also granted in this scenario.

In the serial implementation, the characters that are part of a substring matched before do not need to be searched again. They will be encoded and the uncoded head pointer will jump over them. However, in this second CUDA version, we don't have the information before ahead about which character is part of a substring. Thus we need to search for all characters and record the encoding information. One can argue that redundant searched characters may introduce more execution time. However, it is also a fact that all the threads spawn for matching at the same starting point and they do the search through the same size of buffer in parallel. All the threads compare the same number of characters. Therefore the redundant search for a character is being hidden by the work of other threads. In the performance analysis section we argue the effects of this argument.

There are several issues that arise with CULZSS Version 2's approach. One of the most difficult to overcome is that the window is sliding, meaning that after the match is done, the matched character from the uncoded buffer is moved to the end of window. The following character needs to see a window where the window head is incremented to one more than the previous point, and the last character checked for matching needs to be visible at the end of the window. In the serial, threaded and CULZSS Version 1 implementations, this is not an issue since the matching process is serial. However in this approach the matching accesses are parallel. As a solution, we extended both search window and uncoded buffers with the expected data for each thread. This way each thread sees a window that it is supposed to see in a serial implementation and a consequent buffer of uncoded items to match. Fortunately, this still grants us coalescing access to the buffers.

3) CPU steps: There are a couple of additional steps needs to be handled by CPU in both CULZSS versions. After distributing the parts of the input data, the threads doing the compression process in GPU writes the compressed data on to its given bucket, returning partial full buckets. After all the threads finished, there is a final separate process to concatenate only the compressed data into a continuous stream of data by getting rid of the empty parts of the bucket. Through our tests we observe a very little overhead of doing this process in sequential so we leave this part serial.

In the second version of CUDA implementation, CPU handles more work than the CULZSS Version 1. In Version 1, CPU only moves the compressed data from each half full bucket, and combines them into one file. In Version 2, the matching phase is done in parallel for each character in input, therefore the previously described redundant searches needs to be eliminated from the encoded output. Because this is a data depended work, it follows a serial path and needs to be done on CPU. Since the output encodings are not known in advance, the flags for encoding will also be generated through this process. In Version 1, GPU handles this work. Separating the work between GPU and CPU actually brings an opportunity for CPU - GPU computation overlap, rather than performance degradation. This opportunity is described in detail in the optimization section.

C. Decompression

The decompression process is identical in both versions. Each character is read, decoded, and written into the output location, either in memory or in a file, according to the given encoding. The same independent behavior exists in the decompression process that lets us to make use of the data parallelism in CUDA. To distribute the work across the GPU cores, we need to identify which block of compressed data needs to be decompressed into the corresponding decompressed data block. To achieve this, we keep a list of block compression sizes that are recorded during compression process along with the compression data. The length of the list depends on the number of blocks that we distribute on CUDA threads. In our tests, we observe that this number is very small compared to compressed data size. Therefore it does not hurt the compression ratio.

D. Optimizations

There are a couple of optimizations and configurations we applied on the CUDA implementations to exploit the potential in GPU cards. We first looked at the global memory usage of the GPUs.

Global memory is accessed via 32, 64, or 128 byte memory transactions and for maximum performance, memory accesses must be coalesced as with accesses to global memory [5]. Anytime an access is needed to an address from a block, the entire block must be transferred. Coalesced accesses that fit into a block can be done by just one memory

Table I
COMPRESSION BENCHMARK AVERAGE RUNNING TIMES (IN SECONDS)

	Serial LZSS	**Pthread LZSS**	**BZIP2**	**CULZSS V1**	**CULZSS V2**
C files	50.58	9.12	20.97	7.28	4.26
DE Map	30.75	6.25	9.14	4.69	15.00
Dictionary	56.91	9.35	20.18	7.13	3.22
Kernel tarball	50.49	9.16	20.45	7.08	4.79
Highly Compr.	4.23	1.2	77.82	0.49	3.40

transaction. In the compression, the input data resides in the global memory after copied from CPU. Each CUDA block is responsible for a fixed sized chunk of that input data residing in global memory. Our implementation uses a 4KB block size. In the First Version each thread in a block is responsible for its chunk, resulting number of threads of chunks per block. In the Second Version each block is consuming one single 4KB chunk. To utilize the memory transactions from global to shared memory, we synchronize the threads. The access to the global memory is needed before each matching process. After the matchings are recorded, new data is moved into the uncoded buffer from global memory. In the second version each thread reads 1 byte of memory, where each thread reads subsequent bytes. In a 128 thread configuration it makes a block size of 128 bytes, a coalescing access to a one block size of 128 bytes and results in only one memory transaction in Fermi architectures.

A second important memory optimization is shared memory usage of the GPUs. In CUDA architecture shared memory can be accessed faster than global memory and in parallel if there is no bank conflicts. The shared memory is divided into banks and each bank can only address one dataset request at a time. If the thread requests are on different banks, all accesses are satisfied in parallel. If there are conflicts, the accesses are serialized [5]. In our first CUDA version, we moved the buffers to shared memory that we use repeatedly for searching substring match. This allowed us a 30% speed up over the global memory implementation. The second version's access pattern gives us more opportunities for better utilization of the shared memory. The access to the buffers are synchronized and organized such that successive accesses to the buffers are resulting in no bank conflict. The speed up analysis of the second version and first version is given in the performance section.

Other than memory access optimizations we also looked at configuration parameters, specifically the thread number per block. The shared memory region per block is a very limited resource in GPUs; therefore the thread count directly affects the available resources. However, choosing a smaller number of threads leads into a loss of performance because of having not enough working elements to exploit the power of GPUs.

In the tests, we see that 128 threads per block configuration is giving the best performance. Another configuration item is window size of the compression algorithm. The size affects the search time. Wider window size takes longer to search but increases the chance of having a better substring match. In our tests we get the best performance with the window buffer size of 128 bytes. This also gives us just enough number of bits to encode in a 16 bit encoding space with extended offset for Version Two. With the window buffer size of 128 bytes, possible offsets are upto 256 that leads to 8 bits for the match offset and 8 bits for the match length starting from that offset. A bigger buffer requires more bits to encode both for the match offset and encoding length.

IV. PERFORMANCE ANALYSIS

A. Testbed Configurations

To evaluate how well the CUDA implementations works, we used a GeForce GTX 480 card with CUDA version 3.2 installed on a machine with Intel(R) Core(TM) i7 CPU 920 running at 2.67GHz. The CPU LZSS implementations are also tested on the same testbed. There are 5 sets of data used to test the programs.

B. Datasets

There are five sets of data chosen to test the programs. Each data set is 128 MB in size. The first set is a collection of C files. This dataset is chosen for a collection of text based input. The second set is taken from Delaware State Digital Raster Graphics and Digital Line Graphs Server. The DRGs and DLGs are produced by the United States Geological Survey (USGS) to represent images of topographic sheets, boundaries, hydrography, vegetative surface cover, non-vegetative features, roads and trails, railroads, pipe and trans lines, and manmade features which commonly used as basemaps for georeferencing and visual analysis [17]. The third data is English dictionary. It is chosen for none repeating text, since it is a list of alphabetically ordered not repeating words. The fourth data is part of the linux kernel tarball. Finally, we tested with a highly compressible, custom data set. It contains repeating characters in substrings of 20. It is chosen to see how well our program can run given the opportunity to compress in an optimal data for LZSS.

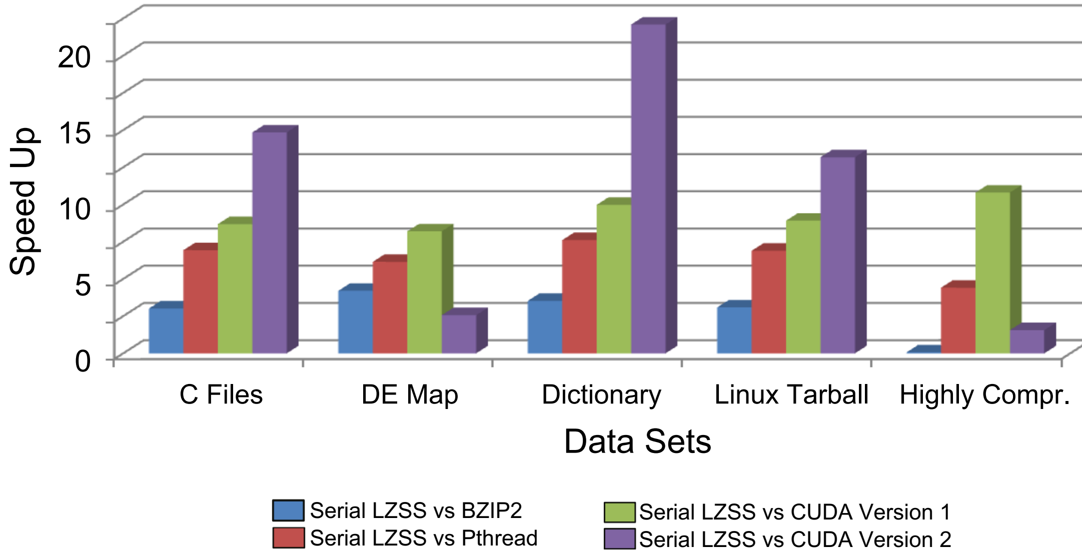

Figure 4. Compression speed up against the serial LZSS implementation compared to all other implementations

C. Compression results

This section shows several test results with compression and decompression. We measure these results by running on the five data sets, 10 times for each set. The results on Table I shows the average running results of the serial and Pthread LZSS, BZIP2 and two CULZSS versions on the data sets. Our first version achieve up to 7x to 9x speed up compared to serial LZSS and 20-30% speed up against Pthread LZSS. Second version achieve up to 18x to 12x speed up for the three data sets compared to serial LZSS and 2x to 3x against Phtread LZSS. For the other two sets of data(Delaware map and highly compressible) version 2 does better than serial implementation but do not achieve any speed up compared to Pthreaded (Figure 4). Compared to BZIP2 the speed up we get from first version is 2x to 3x on the first data set, but for the last data set which is highly compressible data, we get an enormous speed up of 160x. Version 2 gives a 5x to 6x speed up for the three data sets (C files, dictionary, and kernel tarball), a 25x speed up on highly compressible data, and no speed for Delaware map (Figure 4).

The compression ratios are very similar with the serial LZSS and the CULZSS Version 1 which shows that the limitation in the window size in CUDA implementation is not really a big concern on the compression ratio (Table II). For version 2, compression ratio increased by 5% to 8% on the three data sets where it is performing average of 2x times better. It is a result of limited shared memory and limited encoding space that avoids us to hold a bigger buffer and better compression ratio on version two.

It is expected that the poorest ratio is received on the English dictionary data set which was chosen for none repeating behavior. The BZIP2 is doing a better job on compression ratios by producing smaller compressed data in size. Bzip2 compressed data is two to three times smaller compared to first four data sets of serial LZSS and CUDA implementation. On the last highly compressible data set, it is 33 times smaller. Compared to time it takes 160x times slower for the last data set to compress and the 33x smaller comparison ratio, the bzip2 is five times less efficient compared to CULZSS Version 1. Although CUDA implementation produces larger compressed files compared to BZIP2, the performance gain in execution time is a lot better. On the other sets there is a proportional increase in the time with the decrease of the compressed file size between CUDA implementation and BZIP2.

Table II
COMPRESSION RATIOS (SMALLER IS BETTER)

	Serial	BZIP2	V1	V2
C files	54.80%	15.60%	55.70%	63.49%
DE Map	33.90%	11.80%	34.20%	33.35%
Dictionary	61.40%	34.50%	61.80%	65.09%
Kernel tarball	55.10%	16.90%	56.50%	62.59%
Highly Compr.	13.50%	0.40%	13.90%	6.34%

D. Decompression results

Decompression results are shown in the Table III. Both of the CULZSS versions use the same decompression implementation. These results are obtained from the in-memory decompression performance without I/O on the previous data sets. According to our results, we achieve 2.5X to 3.5X speed up for the decompression process compared to serial LZSS implementation. The speed up is lower than the speed up we gain from the compression phase. One

possible reason for that is decompression is not computation intensive process, instead mainly reading from and writing to memory. The potential in GPUs depend on the computational overload. The memory dominated work didn't give enough opportunities for the CUDA implementation to achieve better performance. Even in this case, CUDA implementation outperformed the serial implementation.

Compression ratios are very similar between the serial and CUDA implementations, especially for version one. It is important to keep the same (or very similar) level of compression ratios with a better running time. This indicates that using CUDA framework for compression does not introduce an additional storage and does not drop the compression ratios.

Table III
DECOMPRESSION BENCHMARK AVERAGE RUNNING TIMES (IN SECONDS)

	Serial LZSS	**CULZSS**
C files	1.79	0.53
DE Map	1.21	0.49
Dictionary	2.02	0.55
Kernel tarball	1.77	0.56
Highly Compr.	0.71	0.27

V. DISCUSSION - LIMITATIONS

Our work shows a promising speed up for the lossless data compression on CUDA platform. However there are some limitations that prevents us from achieving better performance. One of the limitations is the limited size of the shared memory per block that is available in CUDA architecture. The global accesses are very expensive compared to shared memory region. There is a 16KB shared memory space for all the threads in a block. In our implementation, mostly visited memory regions are the buffers in which we perform search for a matching substring. In the first version the limited space limits us to put the whole buffers into the shared memory regions in configurations where 256 to 512 threads are used per block.

Another limitation is the nature of LZSS algorithm. It is not explicitly data parallel algorithm which includes portions of code that cannot be parallelizable. In the first version, that limitation pushes us to look for parallelism in dividing the input data into chunks. The data blocks that are far enough between each other can be treated as independent. The decision on the length of the blocks is a configuration parameter. Making it smaller or larger only changes the number of CUDA blocks. The possibility of finding a better match depends on the size of sliding window and lookahead buffer, not directly to the size of blocks. We decided 4KB is a reasonable choice for an average size of a network packet. In the second version we manage to port the matching part into CUDA. As a consequence some of the work left to CPU after the kernel run, however this is more than an obstacle. This gives the opportunity to overlap CUDA and CPU computation.

In our tests we achieve a better execution time compared to both serial LZSS and BZIP2 implementations. The main goal being achieving a speed up against CPU based LZSS, BZIP2 is also used to compare our results with a well known, widely used program. Pthread implementation is also used to fully utilize CPU as well. The execution times are promising and outperforming both of them. However, in compression ratios BZIP2 is doing a better job, especially on highly compressible data suitable for LZSS algorithm. The CULZSS versions, especially the first version is doing a terrific job with speed up of 160x. One can argue the custom made file is not a representation of a real life dataset; however, it is added for completeness to show the capability of CUDA implementation in scenarios of compressing highly repeating data.

Except for the Delaware map and highly compressible data, CUDA version 2 is giving the best performance. Main reason for the better performance is better utilization of CUDA platform; coalescing accesses to global memory and efforts on avoiding bank conflicts to shared memory usage. The reason version 2 is performing poorly on the other two data sets is the nature of the data. As shown on Table II both data sets are highly compressible with LZSS algorithm. The matching is being done for all the characters in the input which cannot take advantage of skipping over the already encoded data. The other implementations save computation time from a match. This version is suitable and gives best performance gain mainly on files that are around 50% compressible data or less (Table II).

The two versions give us the opportunity to satisfy any data types, highly compressible or not. Users of our library can specify the version on the API call and the compression will be done by the specified implementation. This feature gives the ability to use the best matching implementation, instead of having on one implementation that suits for certain data types.

VI. RELATED WORK

The related work can be grouped into two sub groups; CPU based and GPU based.

A. *CPU based*

There are several works on parallelizing data compression on thread level to improve the performance of the compression. Similar to our first version of CUDA implementation, Gilchrist's work parallelizes the BZIP2 program by dividing the input data into chunks and distributing them to threads [16]. PGIZ is another parallel implementation of compression tool, GZIP, which makes use of the multi cores in systems [18], [19]. Distributed computing techniques are also used to increase the performance for compression

process [20]. For high bandwidth and low latency on high speed networks, Franaszek et al. [21] proposes a parallel compression where multiple compressors jointly construct a dictionary to achieve compression performance similar to the sequential implementation.

B. GPU based

Lossy data compression is a field that has been investigated by GPU community. NVIDIA CUDA SDK has sample codes, utilities and, whitepapers on Image/Video Processing [22] and DirectX Texture Compressor (DXTC) for real-time hardware decompression of textures [23]. Using run-length encoding to improve the performance of remote rendering for interactive applications with the use of GPUs has been explored [24].

On lossless data compression recently O'Neil et al. [25] gives a CUDA implementation of specialized compression algorithm targeted specifically for double-precision floating-point data called GFC. These floating-point compressors aims for high performance data process for certain speed rates. On a prior work Balevic proposes a data parallel algorithm for variable length encoding with the new availability of atomic operations on GPGPUs [26]. This approach brings a 35x to 50x speed up over serial implementation.

Porting lossless data compression algorithms on CUDA is a field that has not been fully investigated yet. Our work aims for a better performing lossless compression program on not only specific well suited data but any data. Unlike previous attempts, our work gives promising performance compared to CPU implementations. Scientific applications can often be extremely data intensive. Many applications write to a file every few timesteps for subsequent visualization. Other long-running applications checkpoint their state to disk for restarting. Compression is also used in fields like Bioinformatics, where partial DNA repeats and palindromes are compressible [27]. This work shows potential benefits of GPUs for those kind of applications running on clusters.

VII. Future Work

For future work, we will investigate generalizing the implementation to operate on any size data set. To support any size, we need to divide the input data into chunks of powers of two sizes. The smallest part can be padded. The concurrent execution and streaming feature of new Fermi GPUs can be used to process those chunks. Another improvement can be a more detailed tuning configuration API that gives the ability to adjust the program for the needs of the user. If better compression ratio is required, an adjustable configuration of increased window size can help. For a faster execution but lesser compression ratio can be achieved by again playing with the buffer and bucket sizes.

Further, a combined CPU and GPU heterogeneous implementation can give benefits for the execution time. Since the chip designers are already looking into putting both in a die, for example AMD Fusion and Intel Nehalem processors, it can be a future proof application for the new trend. Although we could not receive any gains in our attempt to use multiple GPUs in a distributed fashion on a machine, a multi GPU implementation can also increase the performance. We could not have a chance to investigate the problem in detail, but we suspect the division of the GPUs by threads introduced thread overhead. There are also further improvement opportunities on the LZSS algorithm, like improved searching with better search algorithms, data structures suitable for CUDA implementation, etc. Last but not least, there are improvements to be made on the clumsier of partial work left on CPU. Both CULZSS versions leave the work of removing the half full buckets from the GPU generated data to CPU. Version 2 has additional encoding work left on CPU. These can be ported to GPU or hidden by overlapping computation with GPU kernel in a pipelining fashion.

VIII. Conclusion

In this paper we examined the feasibility to use CUDA framework for LZSS lossless data compression algorithm. Our main target was to outperform the CPU based implementation by using NVIDIA GPUs without losing any compression ratio. Our implementation is tested on several data sets and compared with the serial implementation. Tests showed that our work outperforms the serial LZSS by up to 18x and the Pthread LZSS by up to 3x in compression time and shows the promising usage of CUDA systems in lossless data compression. The compression ratios between the serial and CUDA implementations are very similar which concludes the CUDA implementation doesn't introduce any additional storage and doesn't drop the compression ratio.

This work, to our best knowledge, is first implementation of general purpose lossless data compression algorithm on CUDA that shows speed up. We also compared our results with a well known compression program, BZIP2, and outperformed up to 6x times on the general data sets, and 160x times on the custom made highly compressible data suitable for LZSS.

The implementation gives programmers an option for using either the in memory compression API or a standalone program which is accepting files as input and writing the compressed file back to the output file. The API also provides an option for CULZSS Version selection to choose the best performance for the given data set.

References

[1] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *Information Theory, IEEE Transactions on*, vol. 23, no. 3, pp. 337 – 343, May 1977.

[2] J. A. Storer and T. G. Szymanski, "Data compression via textual substitution," *J. ACM*, vol. 29, pp. 928–951, October 1982. [Online]. Available: http://doi.acm.org/10.1145/322344.322346

[3] M. Burrows and D. Wheeler, "A block sorting lossless data compression algorithm," Digital Equipment Corporation, Tech. Rep. SRC-RR-124, 1994.

[4] D. Salomon, *A Guide to Data Compression Methods*. Springer, February 8 2002.

[5] NVIDIA, *CUDA C Programming Guide*, 2012.

[6] V. Simek and R. R. Asn, "Gpu acceleration of 2d-dwt image compression in matlab with cuda," *Computer Modeling and Simulation, UKSIM European Symposium on*, vol. 0, pp. 274–277, 2008.

[7] S. S. Stone, J. P. Haldar, S. C. Tsao, W.-m. W. Hwu, Z.-P. Liang, and B. P. Sutton, "Accelerating advanced mri reconstructions on gpus," in *Proceedings of the 5th conference on Computing frontiers*, ser. CF '08. New York, NY, USA: ACM, 2008, pp. 261–272. [Online]. Available: http://doi.acm.org/10.1145/1366230.1366276

[8] M. Moorkamp, M. Jegen, A. Roberts, and R. Hobbs, "Massively parallel forward modeling of scalar and tensor gravimetry data," *Computers & Geosciences*, vol. 36, no. 5, pp. 680 – 686, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/B6V7D-4YDT400-1/2/e4d65a17d0ad393deaf4aab529aa7390

[9] K. Zhang and J. U. Kang, "Graphics processing unit accelerated non-uniform fast fourier transform for ultrahigh-speed, real-time fourier-domain oct," *Opt. Express*, vol. 18, no. 22, pp. 23 472–23 487, Oct 2010. [Online]. Available: http://www.opticsexpress.org/abstract.cfm?URI=oe-18-22-23472

[10] M. Moazeni, A. Bui, and M. Sarrafzadeh, "Accelerating total variation regularization for matrix-valued images on gpus," in *Proceedings of the 6th ACM conference on Computing frontiers*, ser. CF '09. New York, NY, USA: ACM, 2009, pp. 137–146. [Online]. Available: http://doi.acm.org/10.1145/1531743.1531765

[11] A. Gaikwad and I. M. Toke, "Gpu based sparse grid technique for solving multidimensional options pricing pdes," in *Proceedings of the 2nd Workshop on High Performance Computational Finance*, ser. WHPCF '09. New York, NY, USA: ACM, 2009, pp. 6:1–6:9. [Online]. Available: http://doi.acm.org/10.1145/1645413.1645419

[12] M. Boyer, D. Tarjan, S. Acton, and K. Skadron, "Accelerating leukocyte tracking using cuda: A case study in leveraging manycore coprocessors," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, May 2009, pp. 1 –12.

[13] NVIDIA, *NVIDIA Fermi Compute Architecture Whitepaper*, 2010.

[14] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, ser. PPoPP '08. New York, NY, USA: ACM, 2008, pp. 73–82. [Online]. Available: http://doi.acm.org/10.1145/1345206.1345220

[15] *LZSS (LZ77) Discussion and Implementation*, Retrieved April 14, 2011 from http://michael.dipperstein.com/lzss/index.html.

[16] J. Gilchrist, "Parallel Compression with BZIP2," in *Proceedings of the 16th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2004)*. MIT, Cambridge, USA: ACTA Press, Nov. 9-11 2004, pp. 559–564.

[17] *Delaware State DRG and DLG Data*, Retrieved April 14, 2011 from http://www.rdms.udel.edu/drgdlg/.

[18] *Gzip*, Retrieved April 14, 2011 from http://www.gzip.org/.

[19] *Pigz*, Retrieved April 14, 2011 from http://zlib.net/pigz/.

[20] S. Pradhan, J. Kusuma, and K. Ramchandran, "Distributed compression in a dense microsensor network," *Signal Processing Magazine, IEEE*, vol. 19, no. 2, pp. 51 –60, Mar. 2002.

[21] P. Franaszek, J. Robinson, and J. Thomas, "Parallel compression with cooperative dictionary construction," in *Proceedings of the Conference on Data Compression*. Washington, DC, USA: IEEE Computer Society, 1996, pp. 200–. [Online]. Available: http://portal.acm.org/citation.cfm?id=789084.789497

[22] *CUDA SDK Code Samples*, Retrieved April 14, 2011 from http://www.nvidia.com/object/cuda_get_samples.html.

[23] I. Castao, *High Quality DXT Compression using CUDA*, 2007.

[24] S. Lietsch and O. Marquardt, "A cuda-supported approach to remote rendering," in *Proceedings of the 3rd international conference on Advances in visual computing - Volume Part I*, ser. ISVC'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 724–733. [Online]. Available: http://portal.acm.org/citation.cfm?id=1779178.1779261

[25] M. A. O'Neil and M. Burtscher, "Floating-point data compression at 75 gb/s on a gpu," in *Fourth Workshop on General Purpose Processing on Graphics Processing Units*, March 2011.

[26] A. Balevic, "Parallel variablelength encoding on gpgpus," in *Proceedings of the 2009 international conference on Parallel processing*, ser. EuroPar'09. Berlin, Heidelberg: SpringerVerlag, 2010, pp. 26--35. [Online]. Available: http://portal.acm.org/citation.cfm?id=1884795.1884802

[27] S. Grumbach and F. Tahi, "A new challenge for compression algorithms: genetic sequences," *Inf. Process. Manage.*, vol. 30, pp. 875–886, October 1994. [Online]. Available: http://dx.doi.org/10.1016/0306-4573(94)90014-0