IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661, p- ISSN: 2278-8727Volume 13, Issue 1 (Jul. - Aug. 2013), PP 122-127
www.iosrjournals.org

Lossless LZW Data Compression Algorithm on CUDA

Shyni K[1], Manoj Kumar KV[2]
[1]*(Computer Science Department, Government Engineering College, Thrissur, India)*
[2]*(Professor in Computer Science Department, Government Engineering College, Thrissur, India)*

Abstract : *Data compression is an important area of information and communication technologies it seeks to reduce the number of bits used to store or transmit information. It will efficiently utilizes the memory spaces and allows to transmit data within a limited bandwidth. Most compression process is achieved by removing data redundancy while preserving information content. Data compression algorithms exploit some characteristics to make the compressed data smaller than the original data. Every data compression process is working with well defined algorithm. Data compression on graphics processors (GPUs) has become an effective approach to improve the performance of main memory. CUDA is a parallel computing platform and programming model invented by NVIDIA. It enables dramatic increases in computing performance with graphics processing unit (GPU).Data compression algorithms on CUDA provides better compression process. In this paper, we implement the most power full algorithm LZW on CUDA architecture. Due to the parallel characteristics of GPU, compression process time is very less than the CPU environment.*
Keywords – *Cuda, Gpu, Lzss, Lzw, Lzo.*

I. INTRODUCTION

Compression is an operation that is used frequently on personal computers. One such reason is storage space conservation, particularly in consumer devices where storage is limited, such as in personal media players. Data is also compressed in order to facilitate backup operations, allowing a lower cost of storage media compared to storing uncompressed data. Further, compression is used to save bandwidth during large file transfers over the Internet. Optimizing compression operations through parallelization could enable several interesting applications. Modern graphics processing cards contain hardware that can be exploited for parallel programming. GPUs are optimized for vector processing of graphics data, and contain on the order of hundreds of parallel units, and up to thousands of threads [4].However, the GPU can also be extended for general parallel computing. NVIDIA‟s CUDA framework provides a hardware and software architecture for general purpose parallel programming on GPUs. Parallelizing compression algorithms on CUDA is interesting because GPUs are common on many personal computers, and provide a way of scaling the parallelization to hundreds of compute units. It will be interesting to discover how these architectural considerations affect the speedups of the various compression algorithms.

II. BACK GROUND

Data compression techniques can be divided into two major families; lossy and lossless. Lossy compression methods achieve better compression by losing some information. When the compressed stream is decompressed, the result is not identical to the original data stream. Lossless compression consists of those techniques guaranteed to generate an exact duplicate of the input data stream after a compression. This is the type of compression used when storing database records, spread sheets, or word processing files. Lossless data compression is generally implemented using one of two different types of modeling: statistical or dictionary-based. Statistical modeling reads in and encodes a single symbol at a time using the probability of that character's appearance. It reads in input data and looks for groups of symbols that appear in a dictionary. If a string match is found, a pointer or index into the dictionary can be output instead of the code for the symbol. The longer the match, the better the compression ratio. Instead they select strings of symbols and encode each string as a token using a dictionary. The dictionary holds strings of symbols and it may be static or dynamic (adaptive).The former is permanent, sometimes allowing the addition of strings but no deletions, whereas the latter holds strings previously found in the input stream, allowing for additions and deletions of strings as new input is being read [2].We will be focusing only on the lossless data compression using dictionary.

A. Lemple-Ziv Algorithms

It is not a single algorithm, but a whole family of algorithms proposed by Jacob Ziv and Abraham Lempel in 1977 and 1978.Lempel-Ziv methods are popular for their speed and economy of memory. The 1977 algorithm is referred to as LZ77uses the past of the sequence as the dictionary.

Whenever a pattern recurs within a predetermined window, it is replaced by a pointer to the beginning of its previous occurrence and the length of the pattern.LZ77 maintains a sliding window during compression. The window below is divided into two parts. The part on the left is called the search buffer. This is the current dictionary, and it always includes symbols that have recently been input and encoded. The part on the right is the look-ahead buffer, containing text yet to be encoded. In practical implementations the search buffer is some thousands of bytes long, while the look-ahead buffer is only tens of bytes long[1]. The 1978 algorithm is referred to as LZ78, uses a dictionary of previously encountered strings. The encoder outputs two-field tokens. The first field is a pointer to the dictionary; the second is the code of a symbol for example, (1,c).Tokens do not contain the length of a string, since this is implied in the dictionary. Each token corresponds to a string of input symbols, and that string is added to the dictionary after the token is written on the compressed stream [1].LZW is one of the subversion of LZ78.This paper will focus on LZW algorithm.

Limitations with LZ77:

❖ If the distance between two repeated patterns is larger than the size of the search buffer, the LZ77 algorithms cannot work efficiently.
❖ The fixed size of the both buffers implies that the matched string cannot be longer than the sum of the sizes of the two buffers, meaning another limitation on coding efficiency.
❖ Increasing the sizes of the search buffer and the look-ahead buffer seemingly will resolve the problems. Increases in the number of bits required to encode the offset and matched string length as well as an increase in processing complexity.

Limitations with LZ78:

❖ No use of the sliding window. Use encoded text as a dictionary which, potentially, does not have a fixed size.
❖ Each time a pointer (token) is issued; the encoded string is included in the dictionary.
❖ Once a preset limit to the dictionary size has been reached, either the dictionary is fixed for the future (if the coding efficiency is good), or it is reset to zero, i.e., it must be restarted.
❖ Instead of the triples used in the LZ77, only pairs are used in the LZ78. Specifically, only the position of the pointer to the matched string and the symbol following the matched string need to be encoded.

1) LZW algorithm.

This improved version of the original LZ78 algorithm is perhaps the most famous published by Terry Welch in 1984. LZW compression replaces strings of characters with single codes. It does not do any analysis of the incoming text. Instead, LZW builds a string translation table from the text being compressed. The string translation table maps fixed-length codes to strings. The string table is initialized with all single-character strings.

Whenever a previously encountered string is read from the input, the longest such previously-encountered string is determined, and then the code for this string concatenated with the extension character is stored in the table. The code for this longest previously-encountered string is output and the extension character is used as the beginning of the next word. Compression occurs when a single code is output instead of a string of characters. Although LZW is often explained in the context of compressing text files, it can be used on any type of file. However, it generally performs best on files with repeated substrings, such as text files. It removed the second item in the double (the index of the symbol following the longest matched string) and hence, it enhanced coding efficiency. In other words, the LZW only sends the indexes of the dictionary to the decoder [1].The actual algorithm as follows.

a) LZW compression Algorithm

Codes 0-255 in the code table are always assigned to represent single bytes from the input file. When encoding begins the code table contains only the first 256 entries, with the remainder of the table being blanks.

Compression is achieved by using codes 256 through 4095 to represent sequences of bytes. As the encoding continues, LZW identifies repeated sequences in the data, and adds them to the code table [1].

Initialize Dictionary with 256 single character strings and their corresponding ASCII codes;

```
Prefix ← first input character;
CodeWord← 256;
while(notend of character stream){
Char← next input character;
if( Prefix + Charexists in the Dictionary)
Prefix ← Prefix + Char;
else{
Output: the code for  Prefix ;
insertInDictionary(  (CodeWord ,  Prefix + Char) ) ;
CodeWord++;
Prefix ← Char;
}
}
Output: the code for  Prefix ;
```

b) LZW Decompression Algorithm

In LZW decompression algorithm, it needs to take the stream of code output from the compression algorithm, and use them to exactly recreate the input stream.

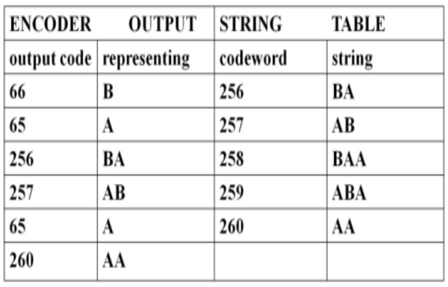
Decompression algorithm is shown as:

```
Initialize Dictionary with 256 ASCII codes and corresponding single character strings as their translations;
 PreviousCodeWord ← first input code;
Output:string(PreviousCodeWord) ;
Char ← character(first i nput code);
CodeWord ← 256;
while(not end of code stream){
CurrentCodeWord ← next input code ;
if( CurrentCodeWordexists in the Dictionary)
String  ← string(CurrentCodeWord) ;
else
String  ← string(PreviousCodeWord) + Char ;
Output:String;
Char ← first character of String ;
insertInDictionary(  ( CodeWord , string(PreviousCodeWord)
+ Char)   );
PreviousCodeWord ← CurrentCodeWord ;
CodeWord++ ;
}
```

In decompression algorithm, code will be searched in dictionary and its character will be output. Consider the BABAABAAA string. First 256 (0- 255) entries of the dictionary are already occupied with 256 ASCII characters. So start with 256th entry of the dictionary. Here B is already included so consider BA to check whether it is in dictionary or not, if it is not present in the dictionary then enter BA to the dictionary and encode the prefix B as 66 as so on. , see Table I

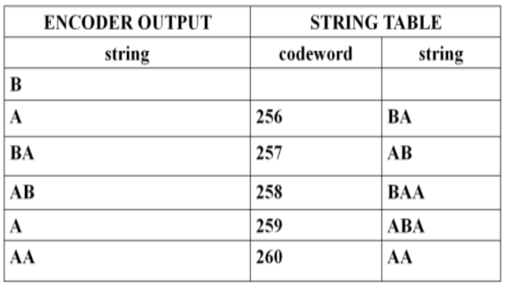
Encoded form of the above the input is **<66><65><256><257><65><260>.** The decompressor will then build a dictionary so that it can receive the indices that refer to the same symbol that are in the compressors dictionary.. For Example to encode the <65>, we know it is the letter 'A'. After decoding create the dictionary entry as previous output plus current output first letter in each decoding step. Here BA is added to dictionary. If dictionary entry of the current code is not decided yet then decoded form will be the combination of current output and current output first letter. For the code <260> decoded as 'AA' using this rule, we can see that it is correct during the dictionary entry creation for the same step, see Table II.

TABLE I. LZW COMPRESSION



ENCODER OUTPUT		STRING TABLE	
output code	representing	codeword	string
66	B	256	BA
65	A	257	AB
256	BA	258	BAA
257	AB	259	ABA
65	A	260	AA
260	AA		

TABLE II. LZW COMPRESSION



ENCODER OUTPUT	STRING TABLE	
string	codeword	string
B		
A	256	BA
BA	257	AB
AB	258	BAA
A	259	ABA
AA	260	AA

B. CUDA Architecture

In this section, we briefly introduce GPUs and CUDA, the underlying platform upon which our compression schemes are implemented. GPUs, originally designed for graphics rendering tasks, have evolved into massively multi-threaded many-core co-processors, as illustrated in Figure 1, for general-purpose computing. The GPU consists of many SIMD (Single Instruction, Multiple Data) multi-processors, all sharing a piece of device memory. CUDA, a general-purpose programming framework for NVIDIA GPUs, exposes the hierarchy of GPU threads. GPU threads exe-cute the same code of a kernel function concurrently on different data. Warps, each of which consists of the same number of threads, are scheduled across multiprocessors.

Within each multiprocessor, warps are further grouped into thread blocks. Threads in the same thread block share resources on one multiprocessor, e.g., registers and local memory (or called shared memory in NVIDIA's term).

CUDA also exposes the memory hierarchy to developers. Multiprocessors share the device memory, which has a high bandwidth and high access latency. For example, NVIDIA GTX 280 GPU has a device memory of size 1 GB, a bandwidth of 141 GB/s, and a latency of 400 to 600 cycles. If threads in a half warp access consecutive device memory addresses, these accesses are coalesced into one memory access transaction. By utilizing the memory coalesced access feature, we can significantly reduce the number of device memory accesses, and improve the memory bandwidth utilization. Each multiprocessor also has a low latency and small sized local memory, and accesses to the local memory must be explicitly programmed through CUDA [6].Each block of threads can be scheduled on any of the available multiprocessors within a GPU, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on any number of multiprocessors as illustrated by Figure 2, and only the runtime system needs to know the physical multiprocessor count. A GPU is built around an array of Streaming Multiprocessors (SMs). A multithreaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors [3].

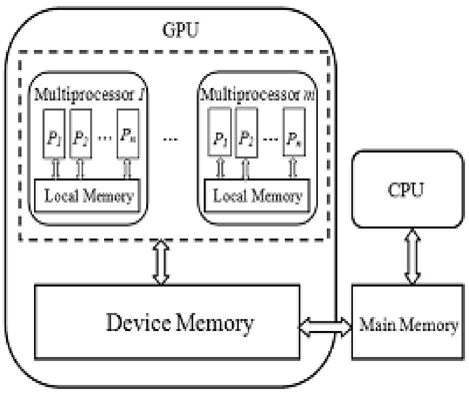


Fig 1. The GPU Manycore Architecture.

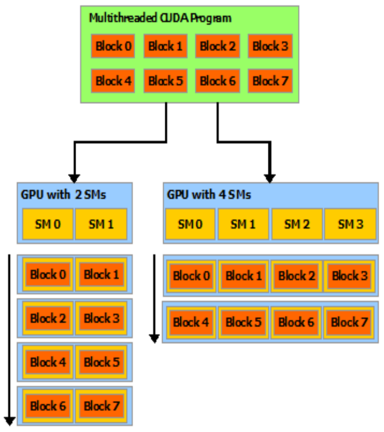

Multithreaded CUDA Program
Block 0 Block 1 Block 2 Block 3
Block 4 Block 5 Block 6 Block 7
GPU with 2 SMs
SM 0 SM 1
GPU with 4 SMs
SM 0 SM 1 SM 2 SM 3

Figure 2: Program blocks assignment to GPU with different number of Multiprocessors.

III. RELATED WORK

As data compression is an effective way to reduce the storage space and increasing the speed of data transmission it has an important role in information and communication technologies. Performance of compression process can be increased by using the opportunities in GPU based systems by exploiting the parallelism in compression algorithms. Porting lossless data compression algorithms on CUDA is a field that has not been fully investigated yet but some works have done.

CULZSS :LZSS lossless data compression on CUDA by Adnan Ozsoy and Martin Swany.[6]gives the implementation of LZSS lossless data compression algorithm on CUDA and analyses the performance with five different types of dataset, see Figure 3. 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. This 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, 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[1].

Another work is done by L. Erdődi , File compression with LZO algorithm[5]using NVIDIA CUDA architecture discusses the possible ways for the implementation of LZO for GPU Fermi architecture.

Three different algorithms are provided and compared and finally it is also shown that the use of GPU can significantly decrease the time of the file compression [7].

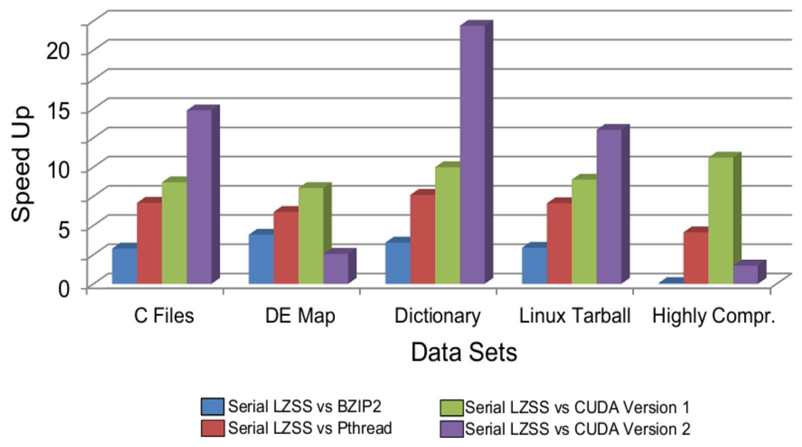


Figure 3: compression speed up against the serial LZSS implementation compared to all other implementations [6].

IV. IMPLEMENTATION

Dictionary based algorithms on CUDA architecture is not implemented yet. LZW can be parallelized by coding each block independently. LZW is one of the powerful subversion of LZ78 dictionary based algorithm. Implementation of LZW dictionary based compression algorithm on GPU architecture will take lesser time compared to its implementation on CPU for some input data.

FILE SIZE	RUNNING TIME(CPU) /sec	RUNNING TIME (GPU)/ sec	COMPRESSION RATIO(CPU)	COMPRESSION RATIO(GPU)
215 KB	52.6394	29.5796	62.1857	57.8111
1,477 KB	205.0344	168.0480	91.1720	90.8867
2,015 KB	399.5633	225.6690	95.8258	94.8616

In this paper LZW algorithm working on CUDA will be simulated using Matlab2012 and the result is comparison study of the LZW algorithm in CPU and GPU architecture. Processing efficiency of GPU implementation will be much more efficient than CPU implementation. Following table shows the comparison study of the CUDA architecture with normal CPU architecture. From the figure GPU performance far better than CPU. For large size of files GPU takes less time than CPU, compression ratio also decreasing.

V. CONCLUSION

GPU has become a powerful device for the execution of data-parallel in which the same operations are carried out on many elements of data in parallel.LZO on CUDA improves the performance than the CPU implementation. Three algorithms are considered to compare the calculation efficiency. Algorithm #3 exceeds the efficiency of the CPU compression implementation. Using NVIDIA GTX580 (512 cores) graphical processing unit with Intel Core i7-2600 CPU with Algorithm #3 the compression process is 20% faster than with CPU. Performance of the algorithm on latest GPU processors gives much more results. NVIDIA GTX680 model possesses 1024 cores, provides calculation speed two times greater than CPU. LZSS on CUDA also improves the performance than CPU. This implementation is tested on several data sets and compared with the serial implementation. 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.LZW (Lempel-Ziv-Welch) which is an adaptive technique.

As the algorithm runs, a dictionary of the strings which have appeared is updated and maintained. The algorithm is adaptive because it will add new strings to the dictionary. Compared to sliding window dictionary based algorithms increases the speed of compression process because of this adaptive technique. Implementation of these dictionary based algorithms should produce better result.

Acknowledgements

The authors wish to thank all the reviewers of the work for their valuable suggestions and advices to successfully complete the work.

REFERENCES

Books:

[1] David Salomon*,"Data Compression The Complete Reference",*Third Edition, Department of Computer ScienceCalifornia State University, USA.

[2] Mark Nelson and Jean-loupGailly, *"Data Compression Techniques ",*College of Applied Studies, University of Bahrain.

[3] Wenbin Fang, Bingsheng He, Qiong Luo."*Database Compression on Graphics Processors*". Hong Kong University of Science and Technology.

Websites:

[4] "*NVIDIA CUDA Compute Unified Device ArchitectureProgramming Guide*" www.nvidia.com, 2012.

[5] M. F. X. J Oberhumer "LZO source code" www.oberhumer.com/opensource/lzo

Proceedings Papers:

[6] Adnan Ozsoy, Martin Swany, "*CULZSS: LZSS Lossless Data Compression on CUDA*," Department of Computer & Information Sciences University of Delaware Newark, DE 19716, 2011 IEEE International Conference on Cluster Computing.

[7] L. Erdődi, *"File compression with LZO algorithm using NVIDIA CUDA architecture*"LINDI 2012 • 4th IEEE International Symposium on Logistics and Industrial Informatics • September 5-7, 2012; Smolenice, Slovakia.