PARALLEL DATA COMPRESSION USING LZMA

[1]NANDAN PHADKE, [2]OMKAR BAHIRAT, [3]TEJASWI KONDURI, [4]CHANDRAMA THORAT

Department of Computer Engg., Rajarshi Shahu College of Engineering, Tathawade, Pune-33.
Email: nandanphadke@gmail.com, omkarbahirat1992@gmail.com, tejaswikonduri92@gmail.com, chandrama1684@gmail.com

Abstract— Now-a-days a tremendous amount of data is generated and shared per second, so there is real need of efficient data storage management and utilization of bandwidth. Data compression is one of solution to this problem. Compression ratio and time required for compression and decompression are two main pillars of data compression. If either of the two gets hampered then data compression may be termed as inefficient.

LZMA (Lample-Ziv-Markov) is one of the finest algorithms for data, to be more precise, text compression in terms of compression ratio. But we found that it has not got that much amount of success due to its comparatively high time complexity for compression. We are trying to improve its time complexity by implementing it in a parallel fashion. GPGPU has provided a well-defined and economic path of parallel implementation to a computer community.

Here we suggest an implementation of a general data compression algorithm that has been designed by keeping in mind the parallel nature of the computing devices that we will be using. This, as we have discovered over the past few months, not only decreases the compression time but also provides us with a compression ratio slightly better than that of LZMA.

Keywords— LZMA, CUDA, lossless data compression, GPU, BZIP2, GZIP.

I. INTRODUCTION

A. Losseless Data Compression

Lossless data compression is a class of data compression algorithms that allows the exact original data to be reconstructed from the compressed data. The term lossless is in contrast to lossy data compression, which only allows constructing an approximation of the original data, in exchange for better compression rates.

In this project we will mainly be utilizing the lossless data compression techniques of Dictionary based Data Compression and Arithmetic Encoding.

B. Literature survey

From the time when the dictionary based compression algorithm was proposed by Limpel and Ziv[1] they have been one of the most popular compression algorithms. The further modifications led to the creation of LZ77, LZSS, LZW and finally the LZMA in 1998.

LZMA is an algorithm introduced by Igor Pavlov in 1998. It was first used in the 7-zip program and is the default compression algorithm used in the 7z format. It has been under development since 1998 and provides a reference implementation of the algorithm along with the XZ encoder used in it. The files compressed by this format have the extensions .7z, .lzma or .xz.

LZMA is a modification over LZ77 using Range encoder. The dictionary size used in the algorithm is variable but is limited up to 4GB. It is known to have better compression ratio but usually the worst time complexity as compared to the rival BZIP2 and GZIP. Analysis shows[2][3] that for general data, that was taken from randomized source packages, that the compression ratio of LZMA was consistently the best but the compression time was also the worst. The analysis conducted by Lasse Collin[2] conclusively proves that GZIP gave the best speed for file size ratio, BZIP best compression ratio while LZMA excellent compression ratio and reasonable memory requirements.

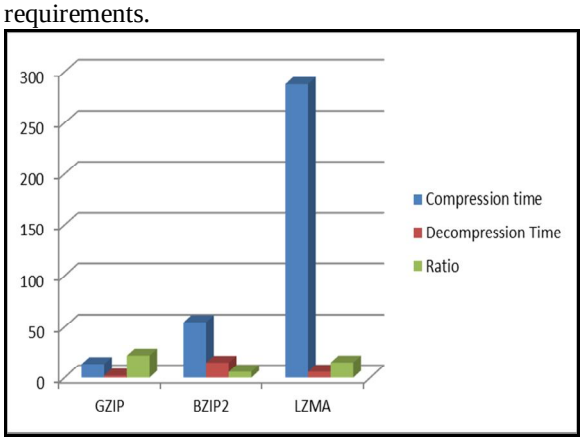

Fig 1. Comparison of the Compression, Decompression times and ratios of popular algorithms

The analysis conducted by Tobias Klausmann[3] also shows that most of the time LZMA has the best compression ratios but the worst compression time. This means that the real optimization required will be required at the compression end. Hence we understand that the LZMA has a two-fold advantage for the clients that will only decompress the files. These are:

1) The best average compression ratio. Meaning reduced network traffic.
2) Faster decompression than BZIP2.

This is the prime motivation for this project.

For both the algorithms BZIP2 and LZMA there exists threaded versions that can be run on either Multi-core or Multithreaded CPU. The parallel

version of bip2 is PBZIP2 and it uses pthreads and achieves near-linear speedup on SMP machines.[5] The output of this compression is fully compatible with BZIP2 v1.0.2 or newer. There also is a distribution, MPIBZIP2 [6], aimed at cluster machines which was developed for distributed-memory message-passing architecture.

The first proposal for parallel implementation of the LZ algorithm was given by Selwyn Henriques and N. Ranganathan from University of South Florida in 1990[7]. They proposed a systolic architecture and algorithm implementing the LZ compression technique. The system was not on GPGPU but on embedded systems and was eventually implemented in VLSI using the CMOS 2-micron technology. More recently the LZ algorithm was proposed and implemented on Parallel and Distributed Systems by Sergio De Agostino of Sapienza University, Rome, Italy in 2011[8].

As far as LZMA is concerned we have LZMA2 where the implementation supports arbitrarily scalable multithreaded compression. Along with that it supports efficient compression of data which is partially incompressible. The XZ LZMA2 encoder processes the input in chunks (of up to 2MB uncompressed size or 64KB compressed size, whichever is lower), handing each chunk to the LZMA encoder. Although all these approaches are on parallelism of the popular data compression algorithms these are not designed for computing on a GPGPU.

II. DESIGN

The algorithm is designed to take any standard text file as an input and will compress it using our own algorithm using CUDA GPGPU.

A. Module

The application will contain following main modules:

- **a.** ***Input File***: We aim that our application should be useful for all text files irrespective of the size of the files. In order for this to be possible we will be dividing the files into a number of fixed size blocks and process these blocks independently.
- **b.** ***Dictionary***: In our analysis of the algorithm we found that creation of a temporary dictionary will be time consuming as it will cause overheads while copying the data to and from device to host memory. Thus we have eliminated the dictionary structure. Logically speaking we are using the input file itself as a "dictionary" and identifying redundancies and eliminating them.
- **c.** ***Range Encoder***: One of the main contributory of the great efficiency of the algorithm is the range encoder that is used after the elimination of the redundancies in the text. Moreover the encoding of individual words is independent of the words occurring prior or after them and thus we have parallelized the encoder in order to add to the efficiency of the algorithm.
- **d.** ***Packet Generator***: Any amount of compression ratio or time efficiency will be useless if we are not able to generate the decompressed file exactly equivalent to the input file. For this to be possible it is necessary to generate certain control information in the file. For this the compressed information will be stored as packets.
- **e.** ***Compressed File***: The output of our entire algorithm. This module is also responsible to calculate the total time taken for the completion of execution of the entire program and comparing it with the CPU-only version if the user has specified that option.

Diagrammatically the system design can be depicted as follows:

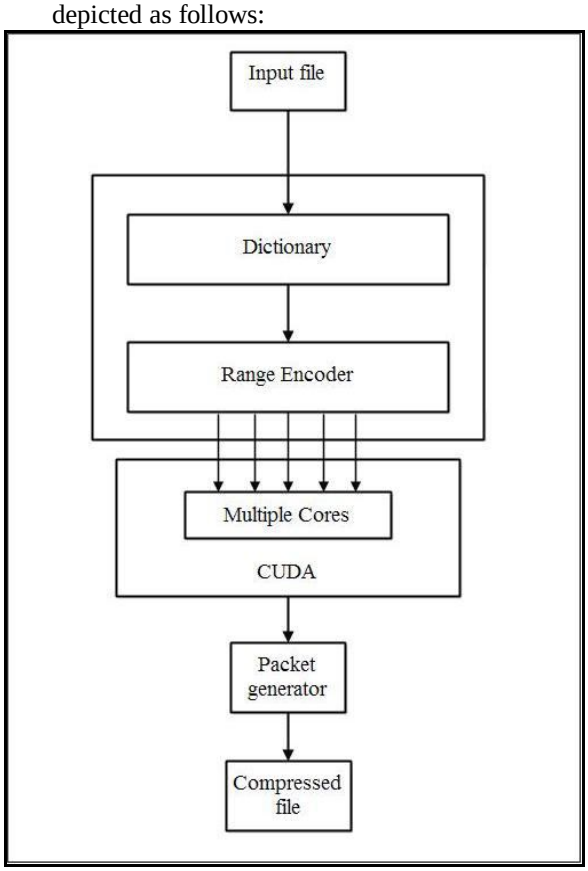

Fig 2. System Design

III. IMPLEMENTATION OF SYSTEM

A. *Hashing*

In order to reduce the length of actual text, the major role playing part is finding out the redundancy in the text to be compressed. The way in which it can be done is just go on tracing the text and keep the track of repeated text, so that it can be represented in some other way in order to achieve the compression.

Hashing has the property that it will give the unique values for different distinct inputs.

Exactly this property of hashing we have used to track the repetitions in the text to be compressed. Hashing will give us the same value for same inputs; hence here we can say that there is a chance of repetition of text. We are using the 2 bytes hashing i.e. the current character and the immediate next look-ahead character are given as a input to hash function and hash function will return the unique value. This unique value will act as a index to hash array in which distances of the bytes having the same hash value are stored.

Consider an **example**:-

abocd……..abc…….abohij……..ablmo…….. ab….
100 254 345 567 879

Assumption:-

1) Let hashing is done for 2 bytes. "h_arr[]" be the hash array used.
2) Let hash value calculated is 750 for bytes "ab"
3) It will get implemented in following manner:

Initially: h_arr[750] = NULL;

Step 1: First "ab" is encountered at distance 100 in string "abocd"; As hash value will be 750 we are going to put the distance 100 at h_arr[750]. For this we need to check the following condition

If (h_arr[750] == NULL) then
Put distance directly at location h_arr[750]
Else
Insert it at first position in the already existing list at index h_arr[750]

Hence here we got
h_arr[750] = {100};

Step 2: Next "ab" is encountered at distance 254 in string "abc";It will produce the same hash value 750. But h_arr[750] is not equal to NULL. Hence insert the distance in first position at index h_arr[750] Hence we get
h_arr[750] = {254,100}

Step 3: Next "ab" is encountered at distance 345 in string "abohij"; It will produce the same hash value 750. But h_arr[750] is not equal to NULL. Hence insert the distance in first position at index h_arr[750] Hence we get
h_arr[750] = {745,254,100}

Step 4: Next "ab" is encountered at distance 567 in string "ablmo";It will produce the same hash value 750. But h_arr[750] is not equal to NULL. Hence insert the distance in first position at index h_arr[750] Hence we get
h_arr[750] = {567,745,254,100}

Step 5: Next "ab" is encountered at distance 879 in string "ab"; It will produce the same hash value 750.But h_arr[750] is not equal to NULL. Hence insert the distance in first position at index h_arr[750] Hence we get
h_arr[750] = {879,567,745,254,100}

The maximum number of element that can be kept in the list varies as per compression ratio demand. More the number of elements more will be the compression ratio. But will increase the time for overall compression process.

As shown in example, the hash function is taking two bytes as an input; this can be changed to three or four bytes.

B. *Finding the maximum length matched*

This is crucial of the entire compression process. The hash array named "h_arr[]" created as shown above plays vital role in this process of finding the maximum length matched.

As we are tracing the text to be compressed for each two bytes, those are consecutive one; we are calculating the hash value. This hash value is giving us the index in "h_arr[]" at which the current distance to be stored.

If the distance to be stored is the first distance at the ith index the it is store as it is. But if it in the other way around then it means that there are already some distance/s stored at ith index. So here we got an opportunity to find out the repetitions which ultimately leads to a compressed text.

For all the available distance/s at the ith index in h_arr[] we will find out the maximum matched length distance. This distance will be stored with the length matched in the compressed file instead of the actual distance.

Consider an example:
Let the string to be compressed is as follows:-

"What is your name? What is your name? What is your name?"

100 119 138

Let, 100,119,138 are the distances of the 'W' as shown above.
Let, 780 is the hash value calculated for the bytes "Wh";
Initially: h_arr[780] = NULL;
So, when first "Wh" encountered at distance 100, the distance will be stored as it is. In this case the character 'W' will be copied into a compressed file as it is.
Hence status of the hash array h_arr[] is;
h_arr[780] = {100}

When "Wh" is encountered at second time at distance 119 then again the hash function will return the same value as previous i.e. 780.

But now h_arr[780] is not equal to NULL.

So at this point, as stated previously, we need to find out the maximum matched length distance.

If we start matching the text in character by character way then we will get the matched length 38.

Hence instead of writing the same text from distance 119 onwards we can just put the pair of [distance,length] and achieve the compression. This we will see in detail in upcoming topics.

Point to be noted in this section is, if among all the distances available at ith index the maximum matched length is less than eight then there is no point in replacing the text by the pair. Because instead of compression it will lead to expansion. As any general character takes seven bits(1 BYTE) for storage eight characters will take (1*8) = 8 BYTES for storage. But pair will take more than or equal to the eight bytes for storage. Hence in such a situation rather than representing the text with pair it is more beneficial to keep the text as it is.

C. *Preparing the Pairs & Indexing*

1. Preparing the pairs

In simple words pairs are nothing but the combination of distance and length. Where, distance is the position at which the matching starts and length is nothing but the number of characters that matched. This pair will actually lead us to compression of text. The pair is replacement for the repeated text.

Table 1. Memory Requirements

Term	Data type used	Memory requirements (in bytes)	
		16 bit compiler	32 bit compiler
Distance	Unsigned integer	Two	Four
Length	Unsigned integer	Two	Four
Entire pair (distance, length)		Four	Eight

Let assume that we are using the 32 bit compiler. Hence the memory required by the pair is Eight bytes. Now in the further discussion we are going to see that, how these pairs will lead us to compress the text.

If the distance is say 200 and matched length is say 87 then the pair will be [200,87].

In this case if we store the 87 characters as it is then we will require (87*1) = 87 BYTES. As for each character there is requirement of 1 BYTE in ASCII for storage. And this memory requirement will be changing depending upon the length matched.

On the other hand, as we have previously calculated, we require only 8 BYTES each time for storing the pair of the same. In this way we can achieve the compression by storing the pair instead of the actual text for repeated text.

2. Preparing the index

The compressed file will consist of a combination of both, that is, actual text bytes and pairs. The indexing of the pairs is required in order to differentiate between the actual text bytes and the pairs. The index will consist of the distance and the number of consecutive pairs in the compressed file. During the decompression the indexing is required.

After application of all the three steps on the example2 which is:-

"What is your name? What is your name? What is your name?"

100 119 138

The compressed string will be: "What is your name? [100,119]"

If we compare both, original string and the compressed string then we will get following

Size of Original string = 57 bytes

Size of compressed string = (19+8) = 27 bytes

D. *Probability Computation*

We are employing a two phased approach in our algorithm: the first being dictionary based compression and then arithmetic encoding. Now arithmetic encoding is based on the fact that not all symbols may be present in the text and even if they are they do not occur with the same frequency thus letting us store the same amount of information in lower size by encoding it.

This means that we need to find out the number of symbols occurring and their probabilities. Ordinarily the technique used is by passing the entire file in one go for probability computation and then another pass for a distinct arithmetic encoding phase. But as we may imagine this requires a large amount of memory access and computation thus making algebraic encoding a computational bottleneck. The alternative as suggested by Whitten, Neal and Cleary[13] is the use of adaptive modelling where the probability will wary as input is encoded. But this has the major disadvantage that the algorithms, though complex, do not save us the extra computations required. Moreover not only are the computations for the encoding increased but also those for decoding increase by the same extent.

Thus we have adopted the novel scheme of parallel implementing the probability computation. In this we need to mould the **data parallel model of the GPU**(where 1 instruction executes on potentially several thousands of threads in hundreds of blocks) into a **instruction parallel model required** by probability computation. This has been achieved by using the ACSII values getting latched to the block and thread identifiers. Thus although a standard template of code will be written to launch the kernel, while executing the code will be modified based on the thread and block identifiers. This model not only

reduces the data dependency among different blocks but virtually eliminates it, thus leading to lower memory access delays. The fixed patter of memory access and the nature that the memory which is shared need only be read by the blocks lets us use the high speed texture memory provided by the CUDA framework.

E. *Arithmetic Encoding:*

It has been proved by Whitten, Neal and Cleary[13] and again by Howard and Vitter[14] that arithmetic encoding is far more efficient in lossless data compression than any of the contemporary methods like dictionary compression or the more famous Huffman encoding. We have implemented this as the last stage of data compression thus gaining a much better compression ratio than just the dictionary based methods.

The implementation of the arithmetic coding given by Whitten[13] in a format in which the model is distinct from the encoding also makes the problem modular in nature. This gives further scope to parallelism. It also increases the reusability of the entire code and further makes it generic in the sense that any proposed change in the model need not alter the implementation of the code.

As previously observed by Whitten, Neal and Cleary[13] the main problem with arithmetic encoding is not the compression ratio but the large time complexity required for encoding. Thus our implementation we have used the CUDA architecture for parallelization of the encoder. This is possible because just like video or image encoding and decoding the outputs are distinct for distinct blocks of data and are in no way interrelated. The only data that will be common to all blocks is the probability values. Moreover this data will only be read by all the threads, thus allowing us to place it in a high speed shared memory in the GPU, thus giving us an even better compression time along with larger amount of global memory to store both the uncompressed string as well as the encoded bytes. This means that all the CUDA Threads will be accessing the shared probability values and the independent blocks of data to be encoded by each thread. After this each thread will encode its block independently and in parallel. Thus we can achieve large amount of parallelism. Furthermore as the dependence or independence of the data present in various blocks is irrelevant, the threads may encode the data of multiple files in parallel.

F. *Assumptions and dependencies*

We have assumed that the data to be compressed in an unformatted Text file(.txt). This has allowed us to focus our attention to the issues of data compression rather than data formatting and support for constructs to handle this.

G. *Future Work*

This project can most trivially be expanded to support formatted data and a wide variety of files used by todays average user, most importantly the Microsoft Office files such as .doc, .docx,.xls, .xslx etc. Further enhances in the file header can be made to compress whole package or folders rather than just individual files.

CONCLUSION

Thus we have successfully developed a parallel algorithm that addresses the issues present in LZMA and rectifies them. This algorithm not only decreases the time required for compression as compared to other popular compression techniques but also achieves a slightly better compression ratio.

REFERENCES

[1] Jacob Ziv and Abraham Lempel, A Universal Algorithm for Sequential Data Compression in IEEE Transactions on Information Theory, 23(3), pp. 337–343, May 1977.

[2] Lasse Collin(2012, Oct. 12), A Quick Benchmark: Gzip vs. Bzip2 vs. LZMA. Available:http://tukaani.org/lzma/benchmarks.html

[3] Tobias Klausmann(2012, Oct. 12), Blog of an Alpha animal, Available: http://blog.i-no.de//archives/2008/05/08/index.html#e2008-05-08T16_35_13.txt

[4] Compression rating team(2012 Oct. 12), 7zip vs. Bzip2 vs. Gzip Available:http://compressionratings.com/comp.cgi?7-zip+9.12b++BZIP2+1.0.5++GZIP+1.3.3+-5

[5] Jeff Gilchrist (2012 Oct. 12), Parallel Bzip2 (PBZIP2) Data Compression SoftwareAvailable: http://compression.ca/pBZIP2/

[6] Jeff Gilchrist (2012 Oct. 12), Parallel Bzip2 Available: http://compression.ca/mpiBZIP2/

[7] Selwyn Henriques and N. Ranganathan, “A Parallel Architecture for Data Compression” Parallel and Distributed Processing, 1990. Proceedings of the Second IEEE Symposium pp264-266.

[8] Sergio De Agostino, “Lempel-Ziv Data Compression on Parallel and Distributed Systems” 2011, First International Conference on Data Compression, Communications and Processing, pp201-202

[9] Adnan Ozsoy, Martin Swany “CULZSS: LZSS Lossless Data Compression on CUDA” in 2011, IEEE International Conference on Cluster Computing2011, pp 403-411.

[10] Wave Access Enterprise Solutions (2012 Oct. 12), Available: http://waveaccessllc.blogspot.in/2011/04/breakthrough-in-cuda-data-compression.html

[11] Bill Dally (2012 Oct. 12), Project Denver Processor to Usher in a new era of computing. Available: http://blogs.nvidia.com/2011/01/project-denver-processor-to-usher-in-new-era-of-computing/

[12] Sergey Poznvakoff(2012 Oct. 12), GNU Tar 1.26 Creating and reading compressed archives. Available: http://www.gnu.org/software/tar/manual/html_node/gzip.html.

[13] Ian H. Willen, Radford m. Neal, and John G. Cleary, “Arithmetic coding for Data compression”, Communications of the ACM, June 1987, Volume 30, Number 6 (Page no: 520 – 540).

[14] Paul G. Howard, Jeffrey Scott Vitter, “Practical Implementations of Arithmetic Coding” Department of Computer Science, Brown University Providence, R.I. 02912-1910.