DKLZSS - A Dynamic KMP String Matching Method for Parallel LZSS Compression on GPGPUs

Vaibhav Tulsyan
Pune Institute of Computer Technology
Pune-411043, India

Aditya Sarode
Pune Institute of Computer Technology
Pune-411043, India

Aaryaman Vasishta
Pune Institute of Computer Technology
Pune-411043, India

Tarun Notani
Pune Institute of Computer Technology
Pune-411043, India

A.R. Sharma
Pune Institute of Computer Technology
Pune-411043, India

ABSTRAT
Fast data compression is gaining increasing importance in the recent times. Statistical compression methods and methods pertaining to textual substitution have been studied in great detail in the last few decades, however, the problem of compressing data at very high speeds with less compression trade-off still remains unsolved to a certain extent. General Purpose Graphic Processing Units (GPGPUs) are a powerful tool that allow large-scale parallel processing, owing to a large number of Streaming Multiprocessors. Recent studies on parallel methods for compression using textual substitution show that considerable speed-ups can be obtained by using the Lempel-Ziv-Storer-Szymanski(LZSS)[9] lossless data compression algorithm on GPUs.In this paper, a parallel, space-efficient compression algorithm using GPGPUs for LZSS compression, along with a dynamic variation of the Knuth-Morris-Pratt (KMP) string-matching algorithm[2] is presented. The algorithm splits the input data into disjoint data chunks and performs compression on each chunk using the Dynamic KMP algorithm, independent of the compression of other chunks.

Keywors
LZSS,KMP,CULZSS,GPU,GPU

1. INTRODUCTION TO THE LZSS (LEMPEL-ZIV-STORER-SZYMANSKI) ALGORITHM
The Lempel-Ziv-Storer-Szymanski(LZSS)[9] is a loss-less data compression algorithm that uses a Macro Encoding technique, also known as textual substitution, for compressing data. It factors out duplicate occurrences of a sub-string and replaces them with a pointer to the original occurrence of the sub-string. The pointer consists of the offset of the original sub-string from the start of the string, and its length.

The LZSS algorithm uses a 2-pointer approach find the longest matching previously occurred sub-string, for replacement. One pointer acts as a look-ahead, which is the starting point of the data

to be compressed. The other pointer scans the string prior to the look-ahead pointer, to find the longest matching replacement.

2. INTRODUCTION TO GPGPUS (GENERAL PURPOSE GRAPHIC PROCESSING UNITS)
Graphic Processing Units are highly parallel computation structures, consisting of several cores, which perform computations at a very high speed, in a parallel way. GPUs are being used for several applications other than video rendering and graphics today. The breed of GPUs created for general purpose computing are known as GPGPUs.

We provide experimental results of the algorithm comparing perfor- mance on a GPGPU with performance of the sequential algorithm on a CPU.

3. IMPLEMENTATION OF LZSS ALGORITHM ON GPGPUS
LZSS algorithm has been implemented on GPGPUs with a consid- erable amount of speed-up[5] as compared to the sequential algo- rithm. CULZSS is one approach, in which the entire data is divided into chunks. Each GPU core performs compression of a chunk, by splitting up the chunk further and assigning them to threads. Each thread uses the standard LZSS algorithm on the data assigned to it and overwrites the original data with the compressed data. Com- pressed data of all threads are then combined sequentially.

The CULZSS algorithm has the following limitations:

1. The worst-case time complexity for each thread to perform compression is $O|S|^3$), where S represents the data assigned to the thread and —S— represents the size of that data in bytes.[5]
2. Combination of the small data chunks, post-compression, is performed sequentially.

In this paper, the focus is on using a faster technique that improves the worst-case compression time by a linear factor, described in Section5.

4. KMP ALGORITHM FOR STRINGMATCHING
Knuth, Morris and Pratt developed an algorithm[2] to solve the problem of checking whether a string P occurs as a sub-string within a text T , in O(|P |) worst-case time complexity. Their al- gorithm is especially useful when the the text T is very large in size, but the string P is comparatively smaller. This problem is also sometimes referred to as finding a needle in the haystack.

In the KMP algorithm[2],a Failure Table structure π is build. This structure is used to store some information about the string P , that helps in finding the number of characters to skip (π(k)) whenever a mismatch occurs during string matching.

5. DYNAMIC KMP METHOD FOR STRING MATCHING IN LZSS (DKLZSS)

In this section,a new method, called Dynamic KMP string-matching algorithm is proposed, which speeds up the process of finding the longest replacement of a sub-string in the LZSS algorithm.

5.1 Construction of Failure Table

If KMP algorithm is used for finding the longest matching sub-string for the data D, as it iterate through the data, building the Failure Table π for each new byte would generally take $O(|D|)$ time, and would hence have a $O(|D|)^2$ worst-case time complexity.

However, this process is redundant and can be optimized such that as a new byte is processed, the Failure Table is updated in $O(1)$ time, and hence takes $O(|D|)$ time overall.

Let the position of the current byte of data be i - the data is partitioned into 2 parts $D[:i]$ and $D[i:]$ respectively, such that $D[:i]$ represents the prefix of D ending at position $(i-1)$ and $D[i:]$ represents the suffix of D starting at position i.

Let the Dynamic KMP Failure Table be represented by μ.A Failure Table pointer j is used to iterate through $\mu(j)$ to set the appropriate value for $\mu(i)$.

Algorithm 1 Failure Table Construction

```
1: global D, k, μ
2: procedure FAILURETABLE(i)                    . Computes μ(i)
3:     j ← i
4:     while True do
5:         if j == 0 then
6:             μ(i) ← 0
7:             break
8:         if D[μ(j) + k] == D[i + k] then
9:             μ(i) ← μ(j) + 1
10:            break
11:        j ← μ(j)
```

The MaxMatch variable is updated with this new value of match. To compute the value of the match variable, the Failure Table μ is used.

The FailureTable() method is called if the value of μ for a particular index is not computed yet. This is the basis on which the algorithm is termed as the Dynamic KMP Method.

Algorithm 2 Dynamic KMP

```
1: global D, k, μ
2: MaxMatch ← 0
3: procedure DYNAMIC KMP
4:     μ(0) ← 0
5:     index ← 0
6:     match ← 0
7:     while index + match < k do
8:         if D[index + match] == D[k + match] then
9:             match ← match + 1
10:            UPDATE MaxMatch
11:        else
12:            if match == 0 then
13:                index ← index + 1
14:            else
15:                if μ(match) not computed then
16:                    CALL FailureTable(match)
17:                index ← index + match − μ(match)
18:                match = μ(match)
```

5.3 DKLZSS Compression Function

The Compress() method takes the input data D as a parameter and stores the Compressed Data so far. It initializes the values of a global variable k, which is then used by the DynamicKMP() function to find out the maximum matching sub-string.

In this function, it is iterated over all k, where $k < |D|$ and keep updating the CompressedString variable.

Algorithm 3 DKLZSS

```
1: global D, k, μ
2: CompressedString ← φ
3: procedure COMPRESS(D)
4:     k ← 1
5:     while k < |D| do
6:         CALL DynamicKMP
7:         if MaxMatch ≥ 3 then
8:             k ← k+ MaxMatch
9:             UPDATE CompressedString with Pointer
10:        else
11:            k ← k + 1
12:            UPDATE CompressedString with Character
```

5.4 Dynamic KMP Method

This method finds the maximum matching sub-string for D[k :], inside D[: k].

The match variable keeps a count of the current value of matching length of sub-string obtained.

The M axM atch variable keeps a count of the Maximum Matching length we have so-far obtained for the sub-string starting at index k. If this length is greater than the MaxMatch variable, then

6 DKLZSS ON GPGPU

In order to improve the execution time of compression, the data is splitted into chunks and delegate the compression of each chunk of data using DKLZSS on different GPU threads, on various GPU cores. The SIMD architecture of GPUs is exploited by the parallel algorithm, so that the chunks perform compression independently, without any intermediate synchronization.

The input data D is stored in the GPU memory and the DKLZSS kernel is called on B blocks, with K threads per block. Each thread, hence, processes at most $|D|/(B * K)$ bytes of data.

In this scenario, each thread consists of thread-local data DT , that is, the chunk of data to be compressed. The thread stores the compressed data CT in a new memory location. The failure table

μT is stored in thread memory.

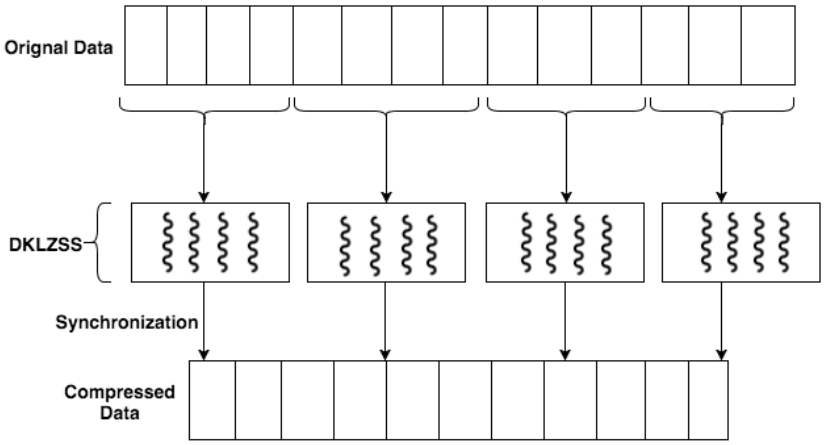

Fig. 1. Block Diagram for DKLZSS

Algorithm 4 DKLZSS - GPGPU

```
1:  Thread-Local D_T , k, μ_T
2:  CompressedString_T ← φ
3:  procedure COMPRESS(D_T)
4:      k ← 1
5:      while k < |D_T| do
6:          CALL DynamicKMP
7:          if MaxMatch ≥ 3 then
8:              k ← k+ MaxMatch
9:              UPDATE CompressedString_T with Pointer
10:         else
11:             k ← k + 1
12          UPDATE CompressedString_T with Character
```

After each thread completes compression, the compressed data from each thread is combined sequentially, to form the overall loss- less compressed data.

7 EXPERIMENTAL RESULT

7.1 Experimental Setup

1. NVIDIA GTX 900 series GPU supporting latest CUDA ver- sion.
2. Intel(R) Core(TM) i7 Haswell CPU - 2.5GHz

7.2 Observatios

A prototype for the DKLZSS algorithm was implemented. The Fig- ure 2) shows the comparative analysis between the LZSS algorithm and DKLZSS.

The X-axis denotes the respective chunk sizes, into which the data

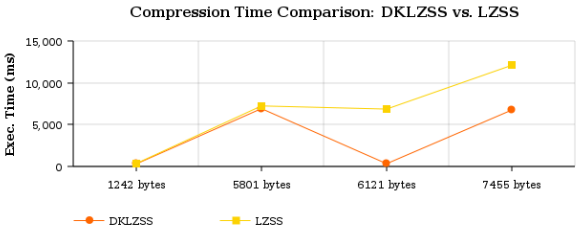

Fig. 2. Compression Time Comparison between DKLZSS and LZSS

was split, in bytes.

The Y-axis denotes the execution time in milliseconds.

The tests were performed on four images, with varying degree of repetitions on the byte-level.

It was found that in cases of high degree of repetitions in the input data, DKLZSS performs with a significant speed-up as compared to the LZSS algorithm.

8 CONCLUSION

In this paper,a parallel, space-efficient compression algorithm is presented.The algorithm uses a textual compression algorithm-LZSS, along with a dynamic variation of the Knuth-Morris-Pratt (KMP) string-matching algorithm.The algorithm splits the input data into disjoint data chunks and performs compression on each chunk using the Dynamic KMP algorithm, independent of the compression of other chunks.The compression of each data chunk happens in parallel manner on different core of GPGPU hence utilizing the parallel power of GPGPUs and acheiving the Compression at a

faster rate. The quick application of the proposed algorithm are real time web based services which requires high data compression rate such as live streaming applications or video chatting applications.

9 REFERENCES

[1] Ana Balevic. Parallel variable-length encoding on gpgpus. In Euro-Par 2009–Parallel Processing Workshops, pages 26–35. Springer, 2010.

[2] Donald E Knuth, James H Morris, Jr, and Vaughan R Pratt.Fast pattern matching in strings. SIAM journal on computing,6(2):323–350, 1977.

[3] SR Kodituwakku and US Amarasinghe. Comparison of lossless data compression algorithms for text data. Indian journal of computer science and engineering, 1(4):416–425, 2010.

[4] Adnan Ozsoy. Culzss-bit: a bit-vector algorithm for lossless data compression on gpgpus. In Proceedings of the 2014 International Workshop on Data Intensive Scalable Computing Systems, pages 57–64. IEEE Press, 2014.

[5] Adnan Ozsoy and Martin Swany. Culzss: Lzss lossless data compression on cuda. In Cluster Computing (CLUSTER),2011 IEEE International Conference on, pages 403–411. IEEE, 2011.

[6] Adnan Ozsoy, Martin Swany, and Anamika Chauhan.Pipelined parallel lzss for streaming data compression on gpgpus. In Parallel and Distributed Systems (ICPADS), 2012IEEE 18th International Conference on, pages 37–44. IEEE,2012.

[7] Adnan Ozsoy, Martin Swany, and Arun Chauhan. Optimizing lzss compression on gpgpus. Future Generation Computer Systems, 30:170–178, 2014.

[8] Akhtar Rasool and Nilay Khare. Parallelization of kmp string matching algorithm on different simd architectures: Multi- core and gpgpu's. International Journal of Computer Appli- cations, 49(11):26–28, 2012.

[9] James A Storer and Thomas G Szymanski. Data compres- sion via textual substitution. Journal of the ACM (JACM),29(4):928–951, 1982.

[10] Annie Yang, Hari Mukka, Farbod Hesaaraki, and Martin Burtscher. Mpc: A massively parallel compression algorithm for scientific data. In Cluster Computing (CLUSTER), 2015IEEE International Conference on, pages 381–389. IEEE,2015.

[11] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. Information Theory, IEEE Transactions on, 24(5):530–536, 1978.

[12] Yuan Zu and Bei Hua. Glzss: Lzss lossless data compression can be faster. In Proceedings of Workshop on General Pur- pose Processing Using GPUs, page 46. ACM, 2014.