Floating-Point Data Compression Using Improved GFC Algorithm

Rui Wu[1] Sergiu M. Dascalu[1] Lee Barford[1,2] Frederick C. Harris, Jr[1]
[1] Department of Computer Science and Engineering, University of Nevada Reno, USA
[2] Keysight Laboratories, Keysight Technologies
{rui, dascalus, fred.harris}@cse.unr.edu lee barford@ieee.org

Abstract

Compression is widely used in both scientific research and industry. The most common use is that people compress the backup data and infrequently used data to save spaces. Compression is significantly meaningful for big data because it will save a lot of resources with the help of a good compression algorithm. There are two criteria for a good compression algorithm—compression ratio and time consumption. GFC is one of the fastest compression algorithms with a mediocre compression ratio, which is designed for real-time compression with the help of Graphics Processing Units (GPU). This paper introduces three methods to increase the speed of GFC algorithm by using the clzll function, removing if-else statements, and using multi-GPUs. The first and third methods improve the original algorithm performance. However, the if-else-removal method cannot always guarantee better results. The final compression speed is more than 1,000 gigabits/s, which is much faster than 75 gigabits/s—the original GFC algorithm speed.

Keywords: GFC; Lossless Compression; High-speed; Floating-Point Data

1 Introduction

Big data and its management is a hot topic for both businessmen and scientists. The digital era brings us many opportunities and also tons of problems. Almost every device keeps generating data all the time. For example, the Large Synoptic Survey Telescope (LSST) needs to manage over 100 PB of data [1]. The Facebook warehouse stores upwards of 300 PB with a daily incoming rate around 600 TB [2]. There are 300 hours of video material uploaded to YouTube every minute [3]. However, it is hard to manage and analyze big data. To uncover the "gold mines" buried in these datasets, researchers hold many conferences to resolve these hard big data problems, such as XLDB [4].

Compression is one of the keys to manage big data and it helps businessmen and scientists save resources. One of the most common rules is that the data management system will compress data if the data is not frequently used. If a compression algorithm compresses original data 20% smaller than before, it means people can save 20% spaces, which means a lot for petabyte-scale datasets. Therefore, a good compression algorithm is significant to a big data project. Also, compression is very significant for some big data web-based application. Dr. Holub and his colleagues introduced a method about how to transmit HD, 2K, and 4K videos with the low-latency network in their paper [5]. The core idea of this project is to compress and decompress JPEG efficiently with the help of GPUs. Figure 1 displays a simplified network diagram of the pilot deployment of their project [5].

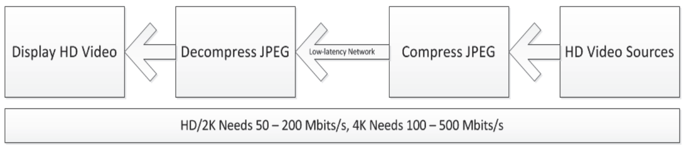


Figure 1. How to Transmit HD Videos with Low-latency Network

There are many mature and good CPU compression algorithms. Some of them are designed for image compressions, such as JPEG [6], some of them are designed for audio and video compression, such as MPEG [7], and some of them are for general use, such as LZ4 [8]. Also, some scientists tried to take advantage of GPU to increase the speed of CPU compression algorithms. For example, [9] tried to improve the Huffman compression algorithm using GPU.

GPU is short for Graphics Processing Unit. It is originally designed for computer graphics and image processing, and it is very popular in high-performance computing today. Also, there is a trend that scientists use multi-GPUs, instead of a single GPU to improve performances of different algorithms. However, GPU is not suitable for all kinds of the algorithm. If an algorithm is not parallelizable or highly divergent, it is better not to use GPU.

Here are some reasons that we chose GFC instead of other algorithms. First, GFC is one of the fastest existing lossless compression algorithms. The original algorithm is 75 gigabits/s [10]. It is gigabit, instead of "gigabyte", because the core ideas of GFC algorithm are based on bitwise operations. The speed is much faster than most other compression algorithms. For example, LZ4 is around 14.56 gigabits/s [8], which is much slower than wide-band network speed. If we do not choose a fast algorithm for high-speed web-based applications, the algorithm will slow down the throughput of these applications. Second, GFC is designed for GPU directly. In contrast to GFC, most of the GPU algorithms are converted from CPU algorithms, which means some compromises have to be made and it will have a negative impact on the algorithm performance most of the time. Third, GFC aims to compress large datasets, which is critical for both business and scientific uses.

Some basic concepts about GPU, such as grid, block, warp, and thread can be found in the paper [11] and Figure 2 displays a common GPU structure, which presents the relations between threads, blocks, and grids. Different GPU video card structures may be different from each other, but they all share some common features: if users want their GPU algorithms to perform best, they have to use all the threads in a warp; if different threads, in the same block, need to communicate with each other, programmers can use shared memory; if different threads, in different blocks, need to communicate with each other, programmers can use global memory.

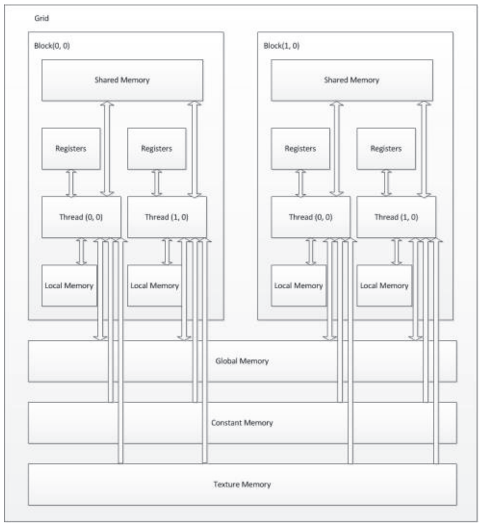


Figure 2. GPU Structure

The rest of this paper is organized as follows in the remaining part: Section II introduces the original GFC algorithm; Section III introduces our three methods to improve GFC algorithm; Section IV introduces the results and our opinions about these results; Section V concludes the main ideas of this paper.

2 Original GFC Algorithm

GFC is a lossless double-precision floating-point data compression algorithm. It is designed for GPU specifically. By using [12], GFC algorithm replaces 64-bit floating-point values with 64-bit integers. Therefore, GFC needs only integer operations, although it compresses floating-point datasets.

Overview of warp, block and chunk assignment of GFC is displayed in Figure 3. The uncompressed data is separated into r chunks and each chunk contains 32 doubles. Each chunk is processed by one warp in the GPU. After all warps finish compressing the assigned chunk, GFC combines all the results together, which is compressed data. The reason that each chunk contains 32 doubles is that there are 32 threads in each warp for most of GPU video cards and it is most effective when a program uses all the threads in a warp.

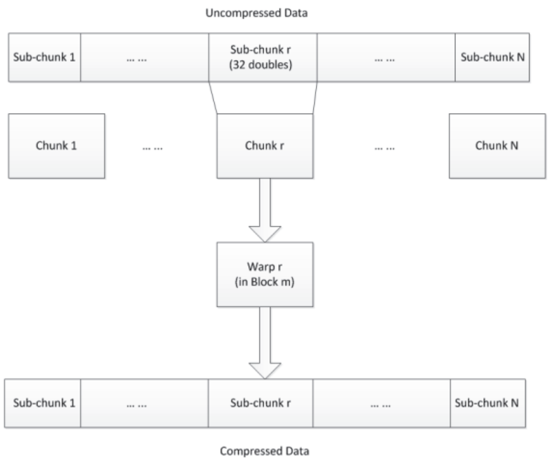


Figure 3. Overview of Warp, Block, and Chunk Assignment

Figure 4 presents the details about GFC compression algorithm. According to GFC, we need to subtract p, which is in the previous chunk, from i, which is in current chunk, and $p = 32 - (dim - I \% dim)$ [10]. Dim means "dimension" in this equation. If the subtraction is negative, we need to use operation—negate to make it positive. The magic part of GFC is the rectangle named residual in the bottom part of Figure 4. By counting the leading zeros of this part, removing these zeros, and adding the leading zeros metadata, GFC compresses the original datasets. The most significant theory behind GFC algorithm is that most scientific datasets interleave values from multiple dimensions [10]. For example, weather temperature will follow a pattern each year for most of the time, which means temperature scientific data can have many leading zeros by using GFC compression algorithm. Users need to find the interleave orders, gets the maximum leading zeros and removes them to have the highest compression ratio.

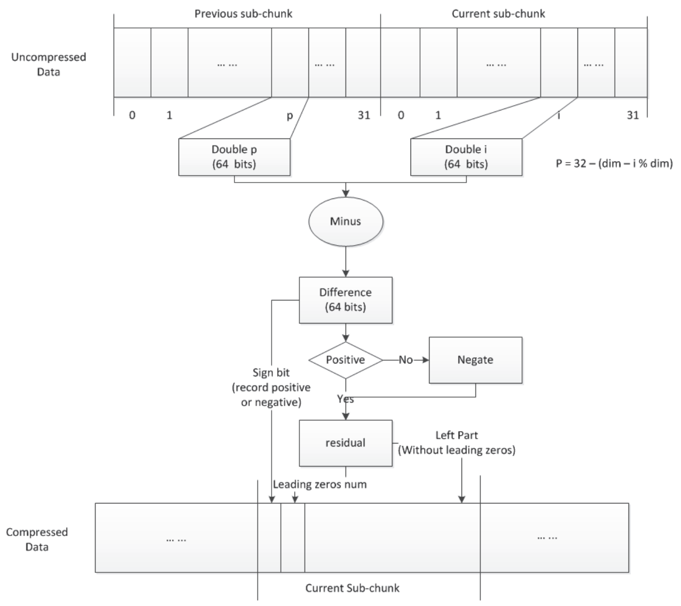


Figure 4. GFC Compression Algorithm

It is possible that the compressed data is larger than the original data using GFC compression algorithm if we choose a bad interleave dimensionality. For example, all the eight bytes of residuals are non-zeros and it results the output sub-chunk is 16 bytes larger than the original chunk, which is 6% larger than the original part [10]. Before users use GFC compression with their data, it is better to preprocess their data and find out the suitable data interleave dimensionality to obtain the best performance.

O'Neil and Burtscher created GFC and published this algorithm in [10]. They avoided using long if-else statements and assigned datasets reasonably according to the structure of GPU to improve the performance of their algorithm. If-else statements can slow down a program, especially GPU program. This is because of the structure of video cards. Each warp has 32 threads (for most video cards) and all these threads (in the same wrap) must execute the same instruction in one cycle [11]. When these threads execute If-else statements, some threads may fulfill the if statement and execute that part of the code, and the remaining threads will stay idle, which means threads are not fully used. Therefore, GFC avoids using long if-else statements.

3 Improved GFC Algorithm

We tried to improve the performance of GFC algorithm with three methods: 1) using clzll to count the leading zeros; 2) removing if-else statements in the program; 3) using multi-GPUs.

A. *Clzll*

In the summary and conclusions part of [10], the authors mentioned that they wrote their own function to count the leading zeros, because their video card was GTX-285 and it does not support clzll, which is used to count the number of consecutive leading zeros bits, starting at the most significant bit (bit 63) of x [13]. They believe GFC could be improved by using clzll to count the leading zeros to replace their code. We agree with their idea because professional programmers in Nvidia know secrets of their video cards. Therefore, it is not strange that their GPU functions are more suitable to the structure of video cards and more effective than our codes. The results in Section IV also prove this idea is right.

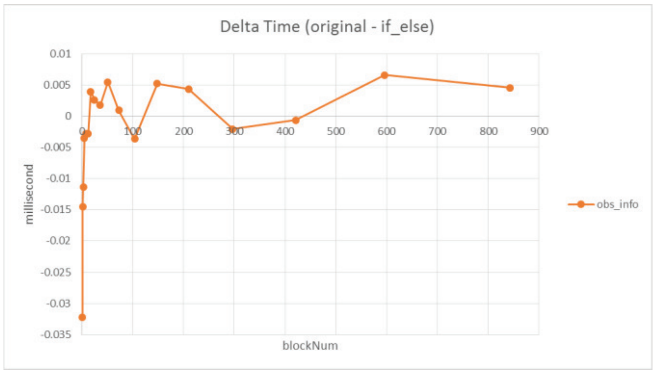


Figure 5. If-else-removal Time Delta

B. *If-else-removals*

In our opinion, if-else statements can slow down programs, especially for GPU programs. Because if-else statements will make some of the threads in a warp idle, when these threads cannot fulfill the if-else statement. Here is an example presented in Figure 6:

```
if $a < 3$ then
  $a = 7$
else
  if $a >= 3$ then
    $a = 5$
  end if
end if
```

Figure 6. If-else Statement Example

Each warp has 32 threads (for most current video cards). Only the threads that fulfill the condition, a > 3, they will execute a =7. Other threads will be idle till the whole warp goes through this if-statement.

There are some materials, such as [14], proving long if-else statements will also have a negative impact on the performance of normal programs. Therefore, we tried to remove if-else statements in GFC algorithm by using bitwise operations. Here is an example, as Figure 7 displays:

```
Algorithm 1 If-else-removal Example
1: if $b >= 2$ then
2:     return b++;
3: end if
```

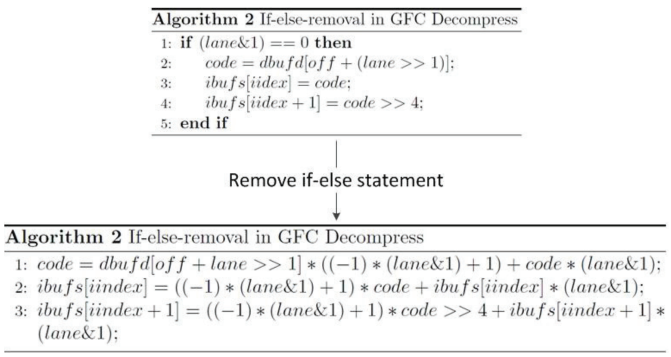
Remove if-else statement

```
Algorithm 1 If-else-removal Example
1: return $b = ((b-2) >> 31) + 1 + b$; ▷ " >> 31"means right shift for 31 bits
```

Figure 7. If-else-removal Example

">>31" means a right shift for 31 bits. For most cases, signed integers have 32 bits and left most bit is used for a sign (positive or negative). (b – 2)>>31 is -1 when b – 2 is negative and it is 0 when (b – 2)>>31 is positive. Therefore, the two statements are the same in Figure 7.

However, we found when if-else statement is short (for example, there is just one line of statement under "if"), the replacement of if-else statements with bitwise operations will slow down the program. We think it may be because something undisclosed in the compiler to optimize the program. The authors of [10] also tried to avoid long if-else statements in their program, except one part in the decompress kernel. Therefore, we replaced that part with bitwise operations as Figure 8 shows.



```
Algorithm 2 If-else-removal in GFC Decompress
1: if $(lane\&1) == 0$ then
2:     $code = dbufd[off + (lane >> 1)];$
3:     $ibufs[iidex] = code;$
4:     $ibufs[iidex + 1] = code >> 4;$
5: end if
```

Remove if-else statement

```
Algorithm 2 If-else-removal in GFC Decompress
1: $code = dbufd[off + lane >> 1] * ((-1) * (lane\&1) + 1) + code * (lane\&1);$
2: $ibufs[iindex] = ((-1) * (lane\&1) + 1) * code + ibufs[iindex] * (lane\&1);$
3: $ibufs[iindex + 1] = ((-1) * (lane\&1) + 1) * code >> 4 + ibufs[iindex + 1] * (lane\&1);$
```

Figure 8. If-else-removal in GFC Decompress

But, the method cannot guarantee better results all the time. Figure 5 displays the delta time between the original algorithm and the improved algorithm for a dataset named obs_info. When the line is above zero, it means the improved algorithm is faster. Even if the improved algorithm is better, the improvement is not really obvious. Therefore, we don't apply this method in the final improved algorithm.

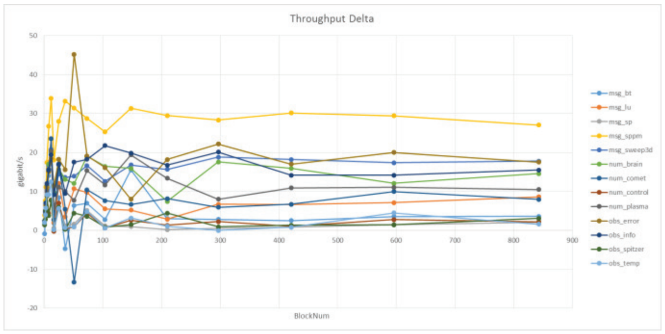


Figure 9. Clzll Throughput Delta

C. Multi-GPUs

After reading some GPU technique papers, we found that some authors try to improve the performance of an algorithm by parallelizing the algorithm and others try to enhance an algorithm by parallelizing tasks. For example, in [15], the author proposed to separate strings and assign a thread for each segment to increase the speed of Boyer-Moore algorithm. We also found there was a trend that scientists used multi-GPUs instead of a single GPU to improve their algorithms.

We found the task—compression is parallelizable. "Parallelizable" means that we can separate the task into several parts and each part can be processed independently. GFC is a GPU algorithm and it uses both blocks and threads. Therefore, we need to assign a GPU for every segment to enhance the performance. So we tried to use multi-GPUs instead of single GPU and the basic idea is displayed in Figure 10. The uncompressed dataset is separated into N chunks, each chunk is processed by a GPU, and each GPU processes the assigned data with GFC algorithm. After all the GPUs finish their jobs, a CPU will combine the results together, which is the compressed data.

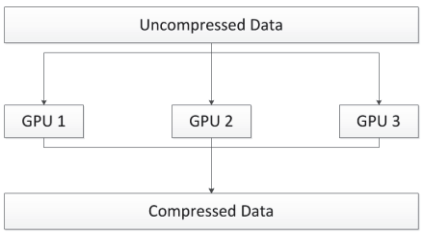


Figure 10. Multi-GPUs Method

4 Results

We did experiments with a Cubix machine, which has eight GeForce GTX 780 video cards, Intel(R) Xeon(R) CPU E5-2620 @ 2.00GHz, and PCI 3.0.

All the flowing experiment datasets are offered by Martin Burtscher, who is one of the authors of [10]. The datasets can be downloaded in [16]. From our experiences about GPU programming, the best results of different problems need different numbers of blocks and threads. After experiments with four of these datasets, we found that we need to use all the threads in the chosen number of blocks to get the best results (throughputs). Therefore, we only did experiments to find the best number of blocks for each dataset and used all the threads. All the experiments were ran 11 times and we chose the median value of these 11 results to be the final result. For example, in multi-GPUs part, we tested different numbers of blocks for a dataset named obs_info. We did the same experiment 11 times and finally found we should use 51 blocks and all the threads in these blocks to get the maximum throughput 1073.376 gigabits/s.

Because [10] mentioned that PCIe bus is too slow for GFC (compression speed is limited to 8GB/s [17]), O'Neil and Burtscher did not record the time of transferring data from CPU to GPU. Therefore, we did not do that for all the following experiments. We also compared decompressed files with original files to make sure that our methods do not change files.

D. Clzll

The first improvement is to use __clzll(), which is used to count the number of consecutive leading zeros bits, starting at the most significant bit (bit 63) of x [13]. The results are presented in Figure 9.

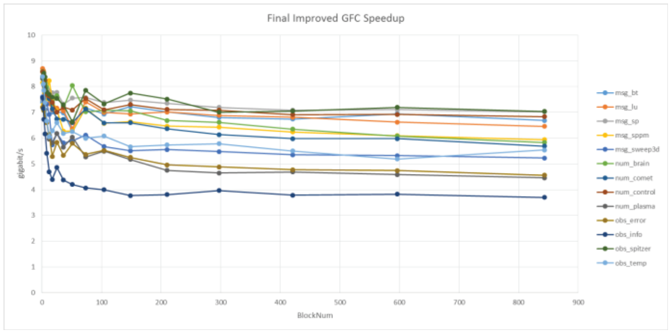


Figure 11. Speedup of Improved GFC Algorithm

In Figure 9, we subtracted original GFC's throughput from improved GFC's throughput. And we found most of the time, the deltas are above zero, which means the improved algorithms' throughput are better. This proves the idea that is introduced in Section III A.

E. Multi-GPUs

We did the experiments with one, two, four, and eight GPUs to study the relation between the number of GPUs and the speedup. We recorded time consumptions of each GPU and used the maximum time to be the final time consumption. For example, we used 8 GPUs and GPU1 spent T1, GPU2 spent T2 … GPU8 spent T8. The final time consumption was Max(T1, T2, … T8). We used the maximum time for the final time because we set up a synchronizing point, which resulted in GPUs waiting for others until all the GPUs finish their jobs. Table I displays the throughputs (gigabits/s) of a dataset named num_plasma. To save time, we did not do the experiment with block number from 1 to 1024. The step of BlockNum in Table I is int(sqrt(2)).

Table II presents the maximum throughputs of different number of GPUs. From this table, we can tell that the speedup is better with more GPUs. However, the relationship between the speedup and the GPU number is not linear. For example, 8-GPU speedup does not equal eight times 1-GPU speedup. In our opinions, this is because of the more GPUs we have, the more segment file will be generated (our program will separate the original file into N parts and each GPU is in charge of a segment). Our program needs to combine all the segment files together to be the file compressed file in the last compression step, which is done by a CPU sequentially. This step will use more time if we have more segment files.

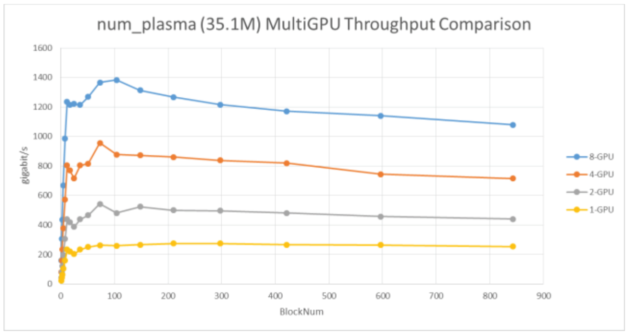


Figure 12. Multi-GPUs Throughput of Num_plasma

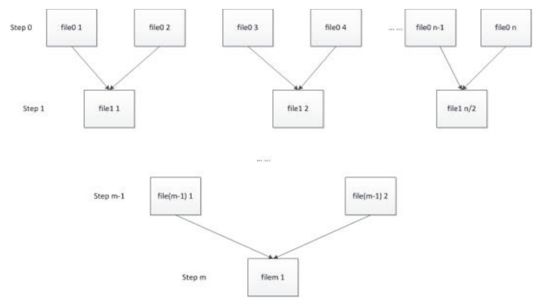


Figure 13. Segment Files Combination

Table I. NUM_PLASMA THROUGHPUTS

BlockNum	8-GPU	4-GPU	2-GPU	1-GPU
1	159.26	81.40	41.18	21.25
2	304.68	158.78	81.51	42.00
3	436.46	233.39	120.21	62.06
5	668.01	376.61	196.12	102.86
8	987.06	572.33	304.24	159.44
12	1,233.31	804.09	438.55	233.19
17	1,214.70	768.86	420.02	219.24
25	1,219.77	715.74	386.17	202.43
36	1,212.85	803.42	438.75	233.02
51	1,268.61	815.37	465.57	250.08
73	1,365.97	955.67	541.73	261.71
104	1,381.89	876.36	481.61	258.48
148	1,312.64	871.23	523.82	264.98
210	1,266.23	860.80	500.14	274.39
297	1,214.87	838.03	496.44	274.89
421	1,170.78	818.49	480.72	266.15
596	1,140.80	743.96	457.01	264.19
843	1,079.34	715.57	439.54	253.56

Table II. MAXIMUM THROUGHPUT

Name	Max Throughput (gigabits/s)	BlockNum	Speedup
8-GPU	1,381.89	104	5.03
4-GPU	955.67	73	3.48
2-GPU	541.73	73	1.97
1-GPU	274.89	297	1.00

Figure 12 visualizes the relation between the throughputs of each number of GPUs with a line chart. For each line in Figure 12, we found they went up first and then went down, which means that too many blocks will reduce the throughputs (gigabit/s) after a certain threshold. When the blocks number is small, N GPUs will increase the throughput almost N times. However, when the blocks number is increased, the speedup is less than N times. We think it may be because of the impact of blocks, as we just discussed. This negative impact will reduce the gap between each of the multi-GPUs results. Therefore, the final results are less than N times, when the blocks number is large.

F. Final Improved GFC Algorithm

Finally, we combined two methods—clzll and multi-GPUs together to improve GFC. We did experiments to datasets from [16] and obtained speedup results (the improved GFC algorithm over the original GFC algorithm) as Figure 11 presents.

The maximum speedup of the improved GFC algorithm is 8.705 and the maximum throughput of the improved GFC algorithm is 2454.603 gigabits/s, which is much faster than original GFC throughputs in [10]. Of course, the good result is partially because we used better hardware than the original GFC paper.

5 Conclusion & Future Work

In this paper, we introduced three methods to increase the speed of a lossless compression algorithm named GFC. These three methods are: 1) using clzll to count the leading zeros; 2) replacing if-else statements with bitwise operations in the program; 3) using multi-GPUs instead of a single GPU.

After some experiments with datasets downloaded from [15], we found 1) and 3) were effective and the maximum speedup is 8.705 and the maximum throughput of the improved GFC algorithm is 2,454.60 gigabits/s, by using 1) and 3) together. However, 2) cannot guarantee good results all the time.

In the future, we want to do more experiments to find out the rules between the performance and number of blocks & GPUs. For example, an equation can obtain the number of blocks & GPUs for a specific problem to get the best results (throughputs). The last step of our method is to combine all the compressed file segments into the final compressed file. This is done sequentially using a CPU core. We have designed a new method to do it parallel using multiple CPU cores. Figure 13 presents the details of this method. The basic idea is to use one CPU core to combine two compressed file segments. Therefore, we can use N CPU cores to combine 2N file segments in one step, which is faster than the sequential combination method.

ACKNOWLEDGEMENT

The authors of this paper acknowledge the help from O'Neil and Burtscher. They answered some hard questions about GFC by email and also offered us test datasets.

This material is based in part upon work supported by: The National Science Foundation under grant number(s) IIA-1329469, and by Cubix Corporation through use of their PCIe slot expansion hardware solutions and HostEngine. Any opinions, finds, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or Cubix Corporation.

References

[1] Cudré-Mauroux, P., Kimura, H., Lim, K. T., Rogers, J., Simakov, R., Soroush, E., ..., and Zdonik, S. (2009). A demonstration of SciDB: a science-oriented DBMS. In *Proceedings of the VLDB Endowment*, *2*(2), 1534-1537.

[2] Vagata, P. and Wilfong, K. (2014). Scaling the Facebook data warehouse to 300 PB (accessed 5/4/2015), https://code.facebook.com/posts/229861827208629/scaling-the-facebook-data-warehouse-to-300-pb/

[3] Google Inc. (2015). Statistics—YouTube (accessed 5/4/2015), https://www.youtube.com/yt/press/statistics.html

[4] Stonebraker, M., Becla, J., DeWitt, D. J., Lim, K. T., Maier, D., Ratzesberger, O., and Zdonik, S. B. (2009, January). Requirements for Science Data Bases and SciDB. In *Proceedings of the fourth biennial conference on innovative data system* , Vol. 7, pp. 173-184.

[5] Holub, P., Šrom, M., Pulec, M., Matela, J., & Jirman, M. (2013). GPU-accelerated DXT and JPEG compression schemes for low-latency network transmissions of HD, 2K, and 4K video. *Future Generation Computer Systems*,*29*(8), 1991-2006.

[6] Wallace, G. K. (1991). The JPEG still picture compression standard. *Communications of the ACM*, *34*(4), 30-44.

[7] Le Gall, D. (1991). MPEG: A video compression standard for multimedia applications. *Communications of the ACM*, *34*(4), 46-58.

[8] Collet, Y. (2013). LZ4-Extremely Fast Compression Algorithm (accessed 5/4/2015), https://code.google.com/p/lz4/

[9] Cloud, R. L., Curry, M. L., Ward, H. L., Skjellum, A., & Bangalore, P. (2011). Accelerating lossless data compression with GPUs. *arXiv Volume 3, 2009, p. 26 – 29.*

[10] O'Neil, M. A. and Burtscher, M. (2011). Floating-point data compression at 75 Gb/s on a GPU. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units,* ACM, p. 7.

[11] Luitjens, J. and Rennich, S. (2011). CUDA warps and occupancy. *GPU Computing Webinar*, *11*.

[12] Kahan, W. (1996). Lecture notes on the status of IEEE standard 754 for binary floating-point arithmetic. *Manuscript, May*. 30 pp.

[13] NuDoq. (2015). NuDoq – CUDAfy.NET (accessed 5/5/2015), http://www.nudoq.org/#!/Packages/CUDAfy.NET/Cudafy.NET/IntegerIntrinsicsFunctions/M/clzll

[14] Loinel, S. (2008). Does a lot of "if … else" statements slow down the code? (accessed 5/5/2015) https://software.intel.com/en-us/forums/topic/283268

[15] Jaiswal, M. (2014). Accelerating Enhanced Boyer-Moore String Matching Algorithm on Multicore GPU for Network Security. *International Journal of Computer Applications*, *97*(1), 30-35.

[16] Burscher, M. (2009). Martin Burscher / FPdouble (accessed 5/5/2015), http://cs.txstate.edu/~burtscher/research/datasets/FPdouble/

[17] Eirola, A. (2011). Lossless data compression on GPGPU architectures. *arXiv preprint arXiv:1109.2348*.