Accelerating data compression using general purpose GPUs

Kristijan Ristovski
Ss. Cyril and Methodius University
Faculty of Computer Science and Engineering
Skopje, North Macedonia
kristijan.ristovski.1@students.finki.ukim.mk

Vladimir Zdraveski
Ss. Cyril and Methodius University
Faculty of Computer Science and Engineering
Skopje, North Macedonia
vladimir.zdraveski@finki.ukim.mk

***Abstract*—The amount of data has been exponentially rising over the years and data centers have invested heavily in research for solutions to efficiently transport and store the data. One of the fields that is crucial to solving this problem is compression of the generated data to reduce the amount of hardware needed for storage of said data. Multiple solutions have already been explored and proposed, and in this paper we experiment with GPU parallelization of compression algorithms to reduce the time required to compress data while maintaining the maximal possible compression ratio. The experiments in this paper explore the viability of using general purpose GPUs for accelerating at home data compression, with the possibility of scaling up to large data centers i.e. using GPU parallelization to reduce compression times.**

***Index Terms*—GPU, CUDA, Parallelization, Compression**

I. Introduction

Data compression has been around for a long time and is a widely researched topic for many years. It's a crucial part of data storage and transfer, especially in systems that utilize large amounts of data transmission. We divide data compression into lossy and lossless approaches. Audio and image data can tolerate some data loss and therefore can utilize lossy compression, but more sensitive data, such as scientific calculations or private user data, cannot tolerate losses during compression.

II. Related Work

Previous research in the field of lossless compression proposed the use of GPU based lossless compression for specific types of data. Fang et al. found that Run-Length encoding based compression yields a significant increase in performance for compressing databases [1], while O'Niel et al. suggested the GFC algorithm that's specifically targeted towards double-precision floating point data [2].

Balevic also worked on RLE using GPUs, but only got increase in performance compared to the serial implementation on the CPU [3]. The parallel algorithm for variable-length encoding used atomic operations and the shared memory for computing codeword offsets in a compressed data stream and cached the codeword in a look-up table in the low latency memory.

Huffman encoding has also been studied by Cloud et al. and showing that GPUs cannot significantly impact performance using the Huffman algorithm because of the algorithms deficiencies [4]. It's proposed that form optimal use of the GPUs SMID cores requires the complete elimination of if statements from the GPU sub-routine. Rahmani et al. presents an algorithm that managed to get a pretty sizable performance increase by parallel implementation of the Huffman algorithm by bypassing the constraints of the maximum codeword length and entropy [5]. With the amount of new research being conducted every day and with technology always improving, resulting data from studies can be extremely precise and can only be represented by the floating-point data type. Ratanaworabhan et al. compare a number of compression algorithms for scientific floating-point data and conclude that real-time compression and decompression is possible on modern hardware [6].

Works in the lossless family of algorithms, such as Lempel-Ziv family of lossless compression algorithms, suggest that its variations can benefit from parralelisation, as shown in Zu and Hua's work where they improve the LZSS algorithm by using their own approach called GLZSS and managed to get double speedup over the existing LZSS algorithm [7]. Concerning the efficiency of these algorithms there are works showing that efficiency is improved using multiple processors, as seen in Erdodi's paper where he implements the LZO algorithm using a CUDA enabled GPU which exploits the number of cores of the GPU and exceeds the CPUs efficiency when using the same algorithm [8].

As previously stated, multimedia like pictures, audio and video, is a data type that can tolerate some data loss. Utilising the weakness of some of the human senses and the human ability to fill in gaps given a context, lossy data can afford to be highly compressed, even at the cost of losing some of the nonessential data bits. For this type of data compression Patel et al. find that performance can be gained when using GPUs for lossy compression when algorithms are written with data level parallelism in mind [9].

There have been instances where algorithms intended for compressing lossless data have been used to compress lossy data. Funasaka et al. implement the LZW (Lempel-Ziv-Welch) compression algorithm to show that compressing TIFF images using a GPU yield much better performance compared to using the same algorithm on a CPU [10].

We also have data that can be compressed using both methodologies, and has pros and cons in each one. For example we have Franklin et al. that propose the HD-ODETLAP algorithm for lossy compression on high-dimensional arrays of data which tackle a pretty significant problem, compressing GIS (Geographic Information System) data [11]. GIS data can be highly compressed at the expense of reconstruction accuracy or be compressed using a lossless algorithm and maintain full accuracy.

III. Solution Architecture

In this paper we will be comparing the execution times of three different compression algorithms on different processing units i.e. on the CPU, where we will be running the

algorithm in sequential mode, and on the GPU, where we will be running the algorithm in parallel mode. Figure 1 presents a general overview of both of these scenarios.

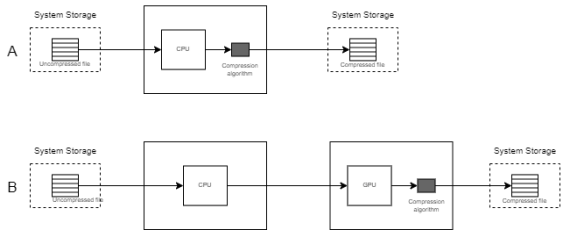

Fig. 1. Scenarios A and B

Suppose we have a large data set comprised of different types of data (audio, video, image or text data) stored in system storage (HDD or SSD). If we are running any of the compression algorithms on a file from the data set using Scenario A, the file first has to be loaded from the system storage to the main memory i.e. the RAM. After the file is put into RAM, the CPU reads it line by line and passes it through the compression algorithm which compresses the data according to the specified compression algorithm and loads the compressed file into RAM, then the resulting file is written to the system storage for the user to access.

Using Scenario B we attempt to improve upon scenario A by using the GPU as the processor for the compression algorithm to utilise to improve compression times. The process starts similarly as scenario A i.e. the desired file is loaded from system storage to RAM where the CPU can send the file using PCIe connections to the GPU. The GPU then runs the received data through the compression algorithm and generates a compressed file which is sent back to the RAM and written to the system storage.

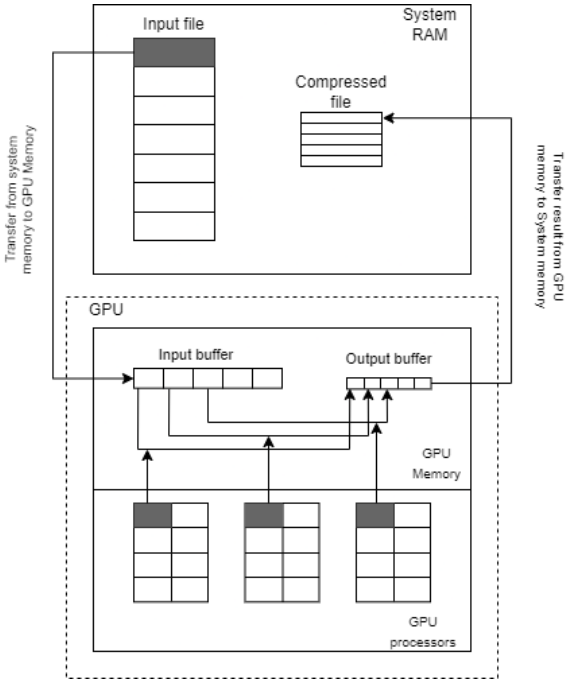

Fig. 2. GPU thread assignment

In Figure 2 we observe, in detail, the way that GPU assigns work to its multiprocessors. The desired file is loaded into RAM and gets copied to the GPUs input buffer where the work is split into threads. The individual threads and are assigned a GPU processing core which executes the compression algorithm and sends the result to the output buffer which collects all of the individual outputs and sends them to the RAM where the compressed file is temporarily stored before it gets permanently stored in the system storage.

In both of these scenarios we evaluate the time it takes to get a compressed file from the moment the command is executed using the *Measure-command* in the Windows terminal (equivalent to *time* in UNIX based systems) and the compression ratio of each file. All of the tests will be run on the same machine with an AMD Ryzen 5 3600 CPU with 6 cores and SMT (Simultaneous Multi-Threading) enabled, an Nvidia GTX 750Ti with 2GB of memory and 640 CUDA cores, a HDD which spins at 7200RPM and 16GB of RAM.

The compression algorithms which we will be using for these benchmarks are:

- BZip2, an open-source program that uses the Burrows-Wheeler algorithm, Run-length encoding, Huffman coding and Delta encoding to compress a desired file. It's much more efficient than traditional compression algorithms but it's noticeably slower.
- LZMA aka the Lempel–Ziv–Markov chain algorithm is mainly used for lossless data compression that uses a dictionary compression algorithm whose output is encoded with a range encoder utilising a complex model to make a probability prediction for each bit. This algorithm is also noticeably slower that traditional algorithms, but can compress much larger data inputs and is also characterised with a fast decompression time.
- PPMd i.e. the Prediction by Partial Matching algorithm, is an algorithm that chooses how to compress the data further in the receiving stream by comparing it with the data it has most recently encountered. This algorithm is also lossless and is effective at compressing text files that contain natural language text but at the cost of a relatively large amount of RAM consumption because of the needs of the prediction model.

The data set on which we will be running the test consists of a large text file that contains generated Lorem-ipsum text, an image with the RAW format and a regular JPEG image.

IV. Results

In this section we evaluate how quickly the files are compressed on our machine using different compression algorithms and see which compression algorithm and/or file type benefits the most from GPU acceleration.

As previously mentioned, our data set consists of a large text file with a size of 953 MB, an image with the RAW file format which has a size of 27.2 MB i.e a large uncompressed image that contains all of the data captured by the DLSR camera and an uncompressed BMP image with a size of 116 KB. All of the tests are repeated 4 times and an average time is calculated.

Figure 3 illustrates the time needed to compress the large text file. As we can see from the graph all of the algorithms execute faster when running GPU acceleration, but the time delta is minimal in most cases. The LZMA algorithm benefits the most from GPU acceleration which reduces execution time by 27.2 seconds but is slowest out of the three by quite some margin and reduces the file size to 203 MB. The

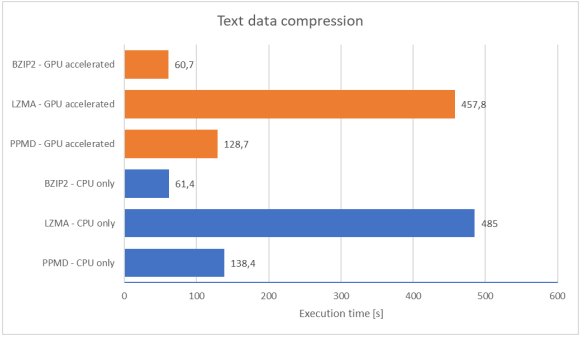

Fig. 3. Execution time of text compression

BZIP2 algorithm has the fastest execution time but also has the worst compression, only managing to reduce file size to 242 MB and the resulting file is 42MB larger than the most compressed file, i.e the resulting file from the PPMD algorithm with a resulting file size of 200 MB, but has the lowest benefit from GPU acceleration. The PPMD algorithm speeds up by 9.3 seconds when using the GPU and is the best out of the three algorithms when compressing textual data.

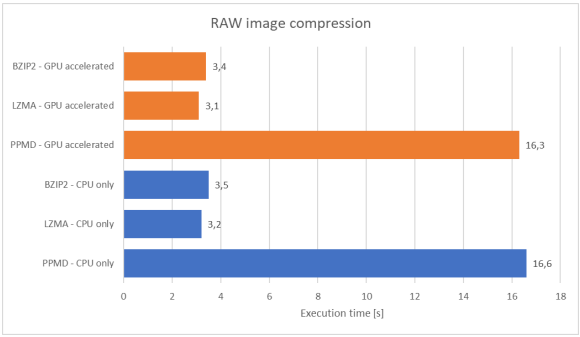

Fig. 4. Execution time of RAW image compression

In the graph in Figure 4 we observe negligible speedup when using the GPU and in the table we can see that all of the algorithms achieve the same amount of compression. This type of data i.e RAW images, is minimally compressible due to the nature of the contained data and results in all of the algorithms having the same compression result of 26.3MB. The purpose of these files is to edit the image without losing any crucial data like colour, ISO etc., so compressing the file RAW only compresses the overhead and metadata in the picture which results in minimal overall compression.

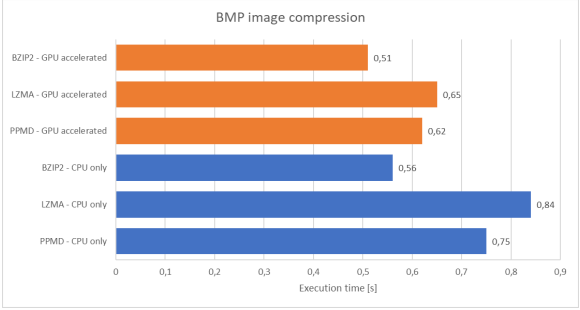

Fig. 5. Execution time of BMP image compression

Figure 5 illustrates the execution time of the compression of a BMP image. The very short execution times are a result of the overall small file size of the image and the fact that BMP images are highly compressible. The compression algorithms all gain from the GPU acceleration but it's very negligible, but the compressed file sizes are half of the original file size. The LZMA algorithm achieves the best compression with a resulting file size of 69.2 MB, while the PPMD algorithm achieves the worst compression with a resulting file size of 77.2 MB and BZip2 algorithm generates a file size of 74.8 MB, which shows that the PPMD algorithm is better optimized for compressing textual data.

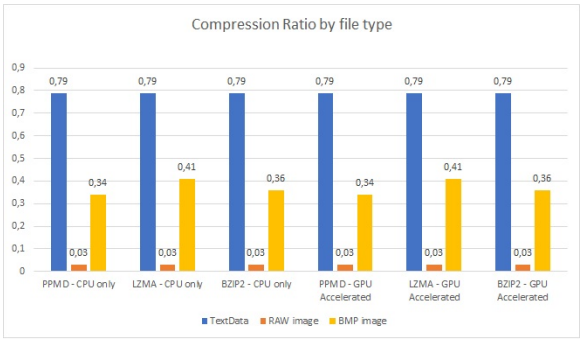

Fig. 6. Resulting compression ratios by file type

Figure 6 depicts the compression ratios of all of the files from the testing data set. We can see that the text data sample and the RAW image have a constant compression ratio that isn't affected neither by the algorithm nor whether the GPU is used or not. The only difference in the compression ratio is observed in the BMP image compression and it only differs based on the algorithm used, not the hardware used to compress the file.

V. Conclusion

In this paper, we compared the execution times and the compression ratios of different file types using three different compression algorithms and executing those algorithms on two different processing units, the CPU and the GPU. We observed a small speedup in execution when running the algorithms on the GPU versus running the same algorithms on the CPU, showing that some algorithms and file types benefit from running on parallel processors. This speedup is almost negligible in most user cases and doesn't make it viable in a personal working scenario, but may prove useful in professional work when compressing very large data sets, like user data bases or data resulting from a large experiment, which take a very long time to complete.

The results from our experiments confirm the results from already existing work i.e. it is possible to accelerate the execution process of compression algorithms when running them on parallel processors.

References

[1] W. Fang, B. He, and Q. Luo, "Database compression on graphics processors," Proceedings of the VLDB Endowment, vol. 3, no. 1-2, pp. 670–680, 2010

[2] M. A. O'Neil and M. Burtscher, "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, 2011, pp. 1–7.

[3] A. Balevic, "Parallel variable-length encoding on gpgpus," in European Conference on Parallel Processing. Springer, 2009, pp. 26–35.

[4] R. L. Cloud, M. L. Curry, H. L. Ward, A. Skjellum, and P. Bangalore, "Accelerating lossless data compression with gpus," arXiv preprint arXiv:1107.1525, 2011.

[5] H. Rahmani, C. Topal, and C. Akinlar, "A parallel huffman coder on the cuda architecture," in 2014 IEEE Visual Communications and Image Processing Conference. IEEE, 2014, pp. 311–314.
[6] P. Ratanaworabhan, J. Ke, and M. Burtscher, "Fast lossless compression of scientific floating-point data," in Data Compression Conference (DCC'06). IEEE, 2006, pp. 133–142.
[7] Y. Zu and B. Hua, "Glzss: Lzss lossless data compression can be faster," in Proceedings of Workshop on General Purpose Processing Using GPUs, 2014, pp. 46–53
[8] L. Erdodi, "File compression with lzo algorithm using nvidia cuda architecture," in 2012 4th IEEE International Symposium on Logistics and Industrial Informatics. IEEE, 2012, pp.251–254.
[9] P. Patel, J. Wong, M. Tatikonda, and J. Marczewski, "Jpeg compression algorithm using cuda," Department of Computer Engineering, University of Toronto, Course Project for ECE, vol. 1724, 2009
[10] S. Funasaka, K. Nakano, and Y. Ito, "Fast lzw compression using a gpu," in 2015 Third International Symposium on Computing and Networking (CANDAR). IEEE, 2015, pp. 303–308.
[11] W. R. Franklin, Y. Li, T.-Y. Lau, and P. Fox, "Cuda-accelerated hd-odetlap: lossy high dimensional gridded data compression," in Modern Accelerator Technologies for Geographic Information Science. Springer, 2013, pp. 95–111