2011 IEEE International Parallel & Distributed Processing Symposium

Bandwidth Reduction Through Multithreaded Compression of Seismic Images

Ahmed A. Aqrawi and Anne C. Elster
Norwegian University of Science and Technology
Department of Computer and Information Science
Trondheim, Norway
aqrawi@stud.ntnu.no, elster@idi.ntnu.no

***Abstract*—One of the main challenges of modern computer systems is to overcome the ever more prominent limitations of disk I/O and memory bandwidth, which today are thousands-fold slower than computational speeds. In this paper, we investigate reducing memory bandwidth and overall I/O and memory access times by using multithreaded compression and decompression of large datasets. Since the goal is to achieve a significant overall speedup of I/O, both level of compression achieved and efficiency of the compression and decompression algorithms, are of importance. Several compression methods for efficient disk access for large seismic datasets are implemented and empirically tested on on several modern CPUs and GPUs, including the Intel i7 and NVIDIA c2050 GPU. To reduce I/O time, both lossless and lossy symmetrical compression algorithms as well as hardware alternatives, are tested.**

Results show that I/O speedup may double by using an SSD vs. HDD disk on larger seismic datasets. Lossy methods investigated include variations of DCT-based methods in several dimensions, and combining these with lossless compression methods such as RLE (Run-Length Encoding) and Huffman encoding. Our best compression rate (0.16%) and speedups (6 for HDD and 3.2 for SSD) are achieved by using DCT in 3D and combining this with a modified RLE for lossy methods. It has an average error of 0.46% which is very acceptable for seismic applications.

A simple predictive model for the execution time is also developed and shows an error of maximum 5% vs. our obtained results. It should thus be a good tool for predicting when to take advantage of multithreaded compression. This model and other techniques developed in this paper should also be applicable to several other data intensive applications.

***Keywords*-Multithreading; GPU compression; I/O acceleration; large datasets; SSD Disk**

I. Introduction

With the introduction of accelerators and ever faster and more parallel processors, the gap between bandwidth and computational throughput is growing even larger causing a further challenges. As stated by Hennessey and Patterson [8] current trends show that the gap will only widen in the future as illustrated in Figure 1. This is especially the case for data-intensive algorithms.

There are both hardware and software alternatives for optimizing the I/O bandwidth. One can upgrade the hardware platform such as using generally faster SSDs (solid state disks) rather than HDDs (Hard disk drives). However, SSDs are both expensive vs HDDs and although about twice as fast, as illustrated later in this paper, they also have bandwidth limitations similar to HDDs. This paper investigates another alternative which accelerates the I/O process by using fast and efficient compression algorithms that take advantage of the computational power of the modern multithreaded system including multi-core CPUs and GPUs. As we have already discovered in our previous work [2], [16], [11], [1], [17], and as it has been shown in several other cases [12], the computational power of modern GPUs are often superior to the CPU. Our work explores the computational capabilities of the GPU for compression both as an accelerator and a supporter for the CPU during computations.

The case of using compression to improve I/O time has some unique properties. In the usual compression scenario, one is aiming for high compression rates and neglect the execution time cost to

1530-2075/11 $26.00 © 2011 IEEE
DOI 10.1109/IPDPS.2011.330

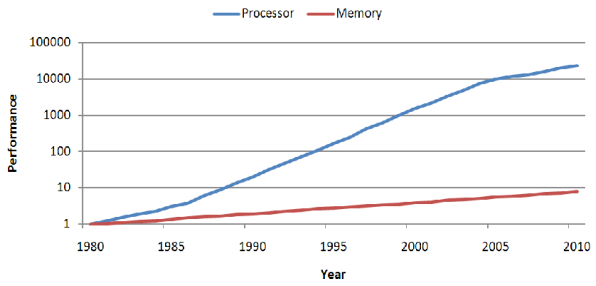

Figure 1. Increasing gap between processor and memory speeds (with permission from D. Patterson [13].)

achieve this compression, which is typical of effective lossless compression methods such as LZMW [15] or lossy methods such as GenLOT [4] with transform coding [14]. However, **when optimizing for I/O, one not only needs efficient compression rates, but also fast compression algorithms.** The next section describes the compression methods studied followed by some words about related seismic imaging and filtering. Compression computations and trade-off are the discussed, followed by the predictive model we developed. The rest of the paper analyzes and discusses our work followed by detailed conclusions and ideas for future work.

A. Compression Methods

Lossless compression is when one does not loose any data during compressing, which is optimal in cases where one is very concerned about data accuracy. In lossy compression on the other hand, some data is lost during compression. In seismic filtering, it is acceptable to lose some data, as long as one maintains two decimal accuracy. In this paper, both lossless and lossy compression algorithms are considered. Our main focus is on simpler, faster, lossless algorithms and transform coding because of their proven usefulness on seismic data [5].

Lossy methods investigated include variations of DCT in several dimensions, and combining these with lossless compression methods such as RLE (Run-Length Encoding) and Huffman encoding. Since seismic data may tolerate lossy compression, compression algorithm based on the DCT (Discrete Cosine Transform) [6] [10] [7] is evaluated as well combining it for 3D data with a modified RLE. As will be seen from our results this lossy method proved to be the fastest for large seismic dataset.

B. Seismic Imaging

Oil exploration is heavily dependent upon seismic imaging, which is considered very data intensive. Seismic imaging is a method of exploring the layers of earth by using signal technology, which like ultrasound imaging, includes recording waves, reconstruction, filtering and analysis. It is the common case that seismic data is recorded, reconstructed and then stored for later filtering and analysis. In this paper, we will focus on the data after it is reconstructed. This part of the seismic process is termed seismic filtering.

The seismic filtering process is built upon two parts: 1) the transfer of data to a computation platform and 2) the actual filtering. In our previous work [2], off-loading computations to the GPU was explored for a typical filtering algorithm, namely 3D convolution.

Our initial results showed that the transfer time consumed 2% of the total execution time. As filtering was optimized by utilizing the computations capabilities of the GPU, the transfer time was 90% of execution time, making it the biggest limiting factor for further optimization. Another aspect worth noting is that this is not only a limiting factor for our work, but for all data intensive algorithms such as seismic processing.

II. Compression Computation and I/O Tradeoffs

Generally, the tradeoff when using compression to reduce I/O time is the time taken to compress and the resulting compression rate. Optimally, one would like to have a compression algorithm that both takes little time to execute and compresses the data well. This way one would reach the best speedup compared to the normal I/O process. In reality, however, there is a relation between the time one uses on compressing data *versus* the execution time– i.e. the more time one spends on compression, the more one is able to compress. This works, of course, only to a certain extent, that is, until the data is no longer compressible.

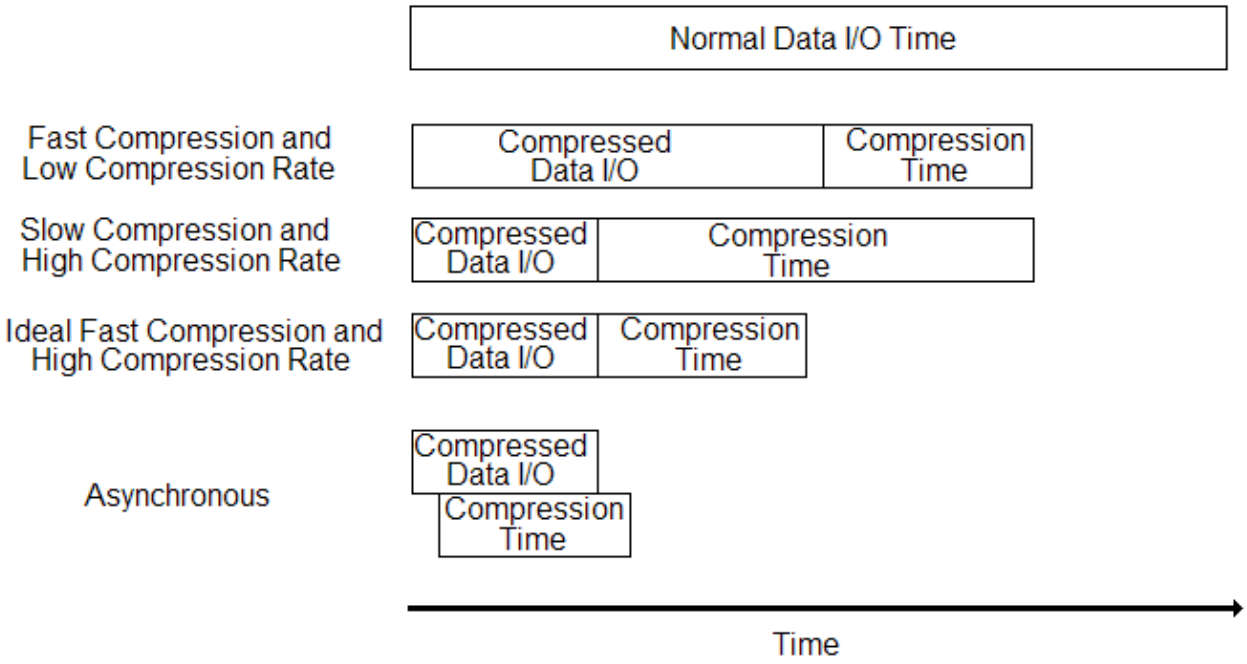

Figure 2. Execution time for combinations of fast/slow compression, high and low compression rates and asynchronous compression

Generally, when using compression, the main goal is to compress as much as possible while there is no time-limiting factor i.e. one can use a lot of time to achieve the best possible compression. In our case, however, we are limited by the normal I/O time. If the execution time of a compression algorithm exceeds that of standard I/O then it will never be able to achieve speedup, and therefore should use standard I/O. That is why we are aiming at using the computation power of the GPU to be able to run the heavier compression algorithms fast enough to gain I/O speedup. This also means that when comparing to faster I/O units such as the SSD disk, one has even more limited time and must therefore depend even more on fast compression algorithms, which limits compression options.

In the case of performing the compression and I/O operations synchronously, one would be limited by the execution time of both since they are performed after one another. This is always the case when performing on a single CPU core – everything is performed sequentially. While in the case of parallelism and using several threads on a multi-core CPU, one can perform the computations and I/O asynchronously, thus being only limited by the execution time of the part that is the most time consuming. This way one can hide either the I/O or the computations in by overlapping the two and the one with the highest execution time will be overshadowing the other. This gives more dimensions and possibilities to use more comprehensive compression algorithms even on fast I/O units, up to a certain point, of course. Similarly, if the compression algorithm is more time consuming than the original I/O time, then it will not give any speedup. See Figure 2 for details.

III. Predictive Model for I/O Compression

The new compressed I/O process can be modeled in two parts as given in Equation 1. It expresses that the I/O time is now the time it takes to read or write the compressed amount of data, $t_{compressedIO}$, plus the time it takes to compress or decompress the data. This concept reflects the synchronous model. One is also able to model this

asynchronously, which we will look at later.

$$t(n)_{SyncI/O} = t(n)_{compressedI/O} + t(n)_{compress/decompress} \quad (1)$$

A. Synchronous Model

When using the conceptual model and expressing it as a function of the original data size n, it will be as in Equation 2 for the case of reading from disk, and Equation 3 for when writing. Here $n_{compressed}$ is the amount of compressed data in bytes, which is dependent of the algorithm one uses to compress and can also be expressed as $n_{compressed} = n * C$, where C is the compression factor and n is the original data size in bytes. r_{disk} is, as earlier, the rate at which the disk reads or writes seismic data depending on the formula and is expressed in bytes per second. Whereas $r(n)_{decompress}$ is the rate, in bytes per second, n bytes are produced by the decompressing algorithm, and $r(n)_{compress}$ is the the rate at which n bytes are compressed. It is important to note the difference to avoid erronious estimates.

$$t(n)_{read} = \frac{n_{compressed}}{r_{disk}} + \frac{n}{r_{decompress}} \quad (2)$$

$$t(n)_{write} = \frac{n_{compressed}}{r_{disk}} + \frac{n}{r_{compress}} \quad (3)$$

The main difference between Equation 2 and 3 is that when compressing, the original data size is compressed into a compressed file size. While when decompressing, the compressed file size is used to reproduce the original file size. The difference is in the amount of reads and writes to memory. When compressing one reads the original file size n than writes $n_{compressed} = n * C$ amount of compressed data, where C is the compression factor dependent on the algorithm used. When decompressing it is the opposite situation, where one writes to memory more than reading. The total time used to compress or decompress symmetrically is about the same because one performs the same operations in reverse, but usually decompression is considered slower because one has to write more. In our case, we only decompress to memory to perform calculations and then compress the data again when storing it back to disk making the compression and decompression times quite similar. This is reflected in our results.

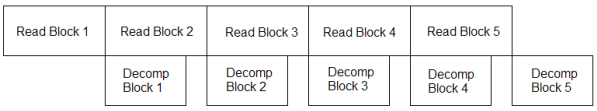

Figure 3. Asynchronous I/O Pipeline, with dominant I/O time

B. Asynchronously

When using more than one processor and parallelize code, one can split the reading and computations into different threads and perform them in parallel. This way one would hide a lot of the computation time with I/O time or the other way around depending on which step is the most time consuming. By reading the data asynchronously, one would have to divide it into blocks. Thus, one could read a small part and start performing computations on that, only to read the next part as these computations are being performed in parallel. This would hide these computations. In Figure 3, we illustrate the scenario of asynchronous reads by splitting the data into 5 blocks.

Conceptually this would result in the model presented in Equation 4, where b is the number of blocks and Max(x,y) is a function that returns the greater execution time between processes x and y. the other variables have been explained previously in the synchronous model.

$$t(b, n_b) = t(n_b)_{compI/O} + t(n_b)_{comp} + (b-1)MAX(t(n_b)_{compI/O}, t(n_b)_{comp}) \quad (4)$$

A clear advantage would be to increase the number of blocks to decrease the overhead from the first and last block of the asynchronous process. However, one has to keep in mind that all reads and writes have an overhead when invoked, which means that if one reads small amount of values per read than the read process will take much longer than reading a larger amount of sequential data.

Table I
COMPONENTS OVERVIEW IN SYSTEM 1 & SYSTEM 2

Component	system 1	system 2
CPU	Intel Q9957 2.81GHz	Intel i7 2.81GHz
RAM	8 GB DDR3	12 GB DDR3
Disk 1	750 GB 7200 rpm	Corsair SSD 256 GB
Disk 2	500 GB 10000 rpm	
GPU 1 (display)	NVIDIA Geforce 8600	No display
GPU 2	NVIDIA Tesla c1060	NVIDIA Tesla s1070
GPU 3	NVIDIA Tesla c2050	

This is why we try to fetch a large amount of data per read and use specific blocking techniques to do so. This was explored in our earlier work with seismic data [2]. Therefore there is a tradeoff between increasing block size and execution speeds.

IV. TEST PLATFORM

Our implementations have been tested on two machines that have different hardware specifications. The first system is one that is put together by us and runs on a Windows 7 operating system. A list of the hardware in the first system is described in Table I. It is worth noting that we have mentioned alternative options on the hardware such as the disk and GPU. This is mainly to underline that we have used the same system, but have changed to the alternative option such as when we tested the new Fermi architecture of NVIDIA. When it comes to the alternative disk we added another disk to add more variety to our I/O tests.

The second system is put together by the personnel of the group, which is a powerful system that has some of the newest hardware one can find, but since the systems is used as a server, we cannot easily change the hardware on this system. This is why we did not test the combination of SSD and Fermi, which also would have been interesting. The second system uses a Linux operating system and is accessed remotely. Note that the change in operating system can be a source of different results in that the operating system schedule tasks differently. An overview of the hardware of the second system is also given in Table I.

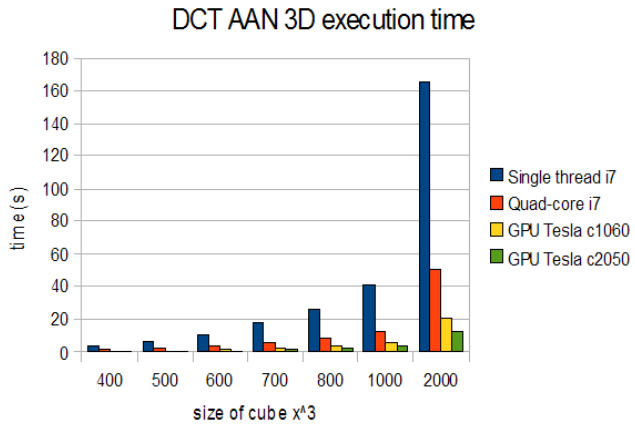

Figure 4. Execution time results for AAN DCT 3D algorithm

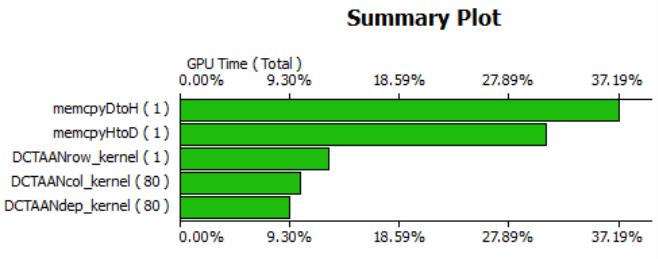

Figure 5. CUDA profiler snapshot of the DCT AAN 3D execution

V. RESULTS & ANALYSIS

In Figures 4 and 5 one can see the results we achieved at speeding up the DCT AAN algorithm in 3 dimensions. This is essential in the results we further achieved in I/O speedup.

The I/O speedup is for our tests are measured as in Equation 5 where we compare the new I/O time using compression to the normal sequential I/O time. The aim is to study the advantage the compression algorithms give for disk access, and to map which option is the most effective. We study two scenarios of I/O: one being synchronous and the other asynchronous. Our tests were also conducted on two hard disk drives (HDD) of varying transfer rates: One with 40 MB per second and the other with 70 MB per second. A solid state disk (SSD), which is of newer technology and has a speed of 140 MB per second for transfers, was

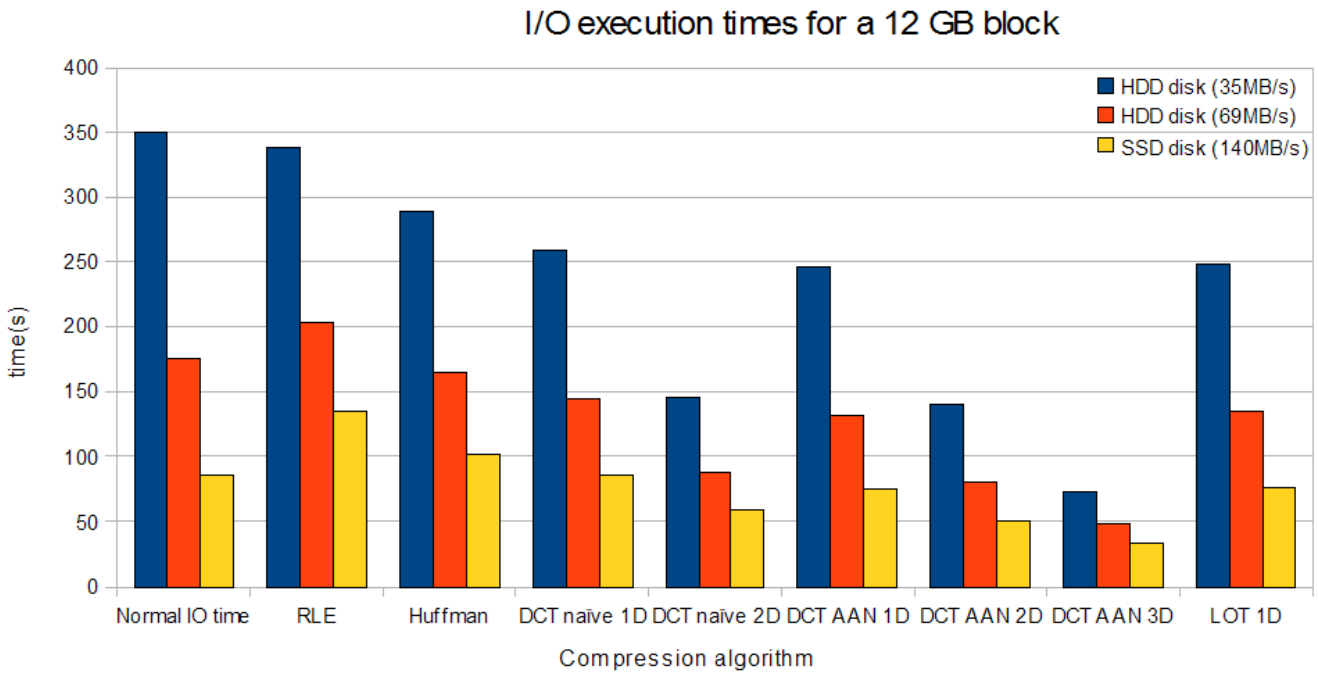

Figure 6. Execution time for synchronous I/O with CPU and GPU acceleration (CPU: Intel Q9958, GPU: NVIDIA Tesla c2050)

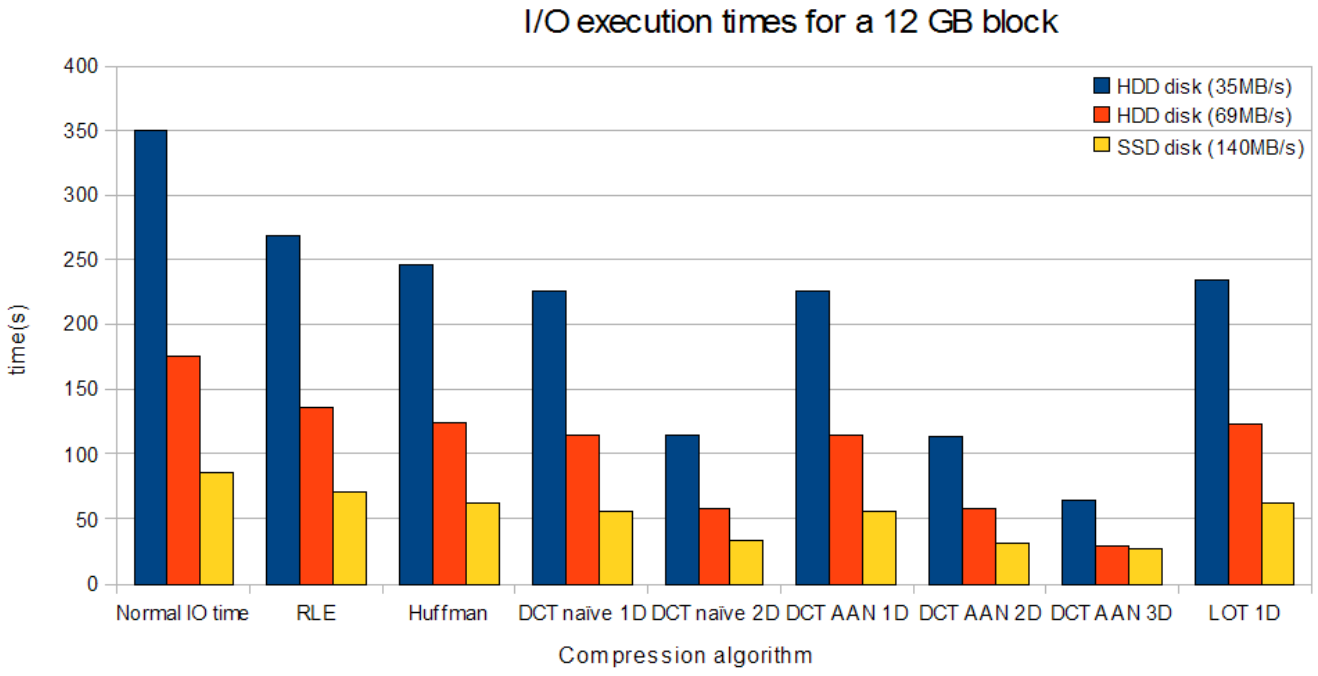

Figure 7. Execution time for asynchronous I/O with CPU and GPU acceleration (CPU: Intel Q9958, GPU: NVIDIA Tesla c2050)

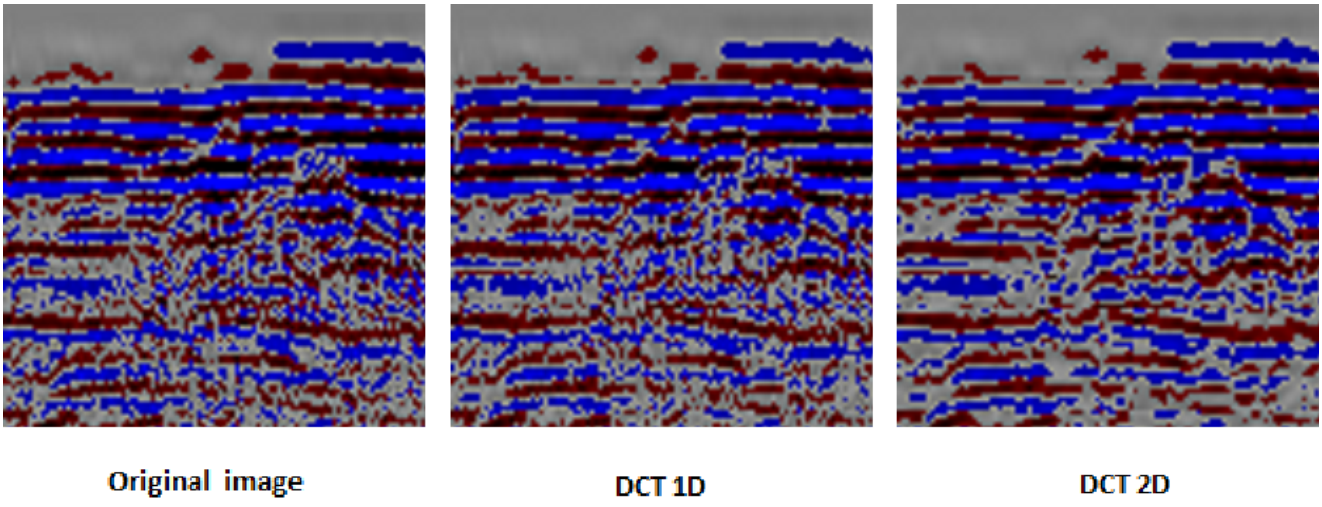

Figure 8. Visual results of data loss

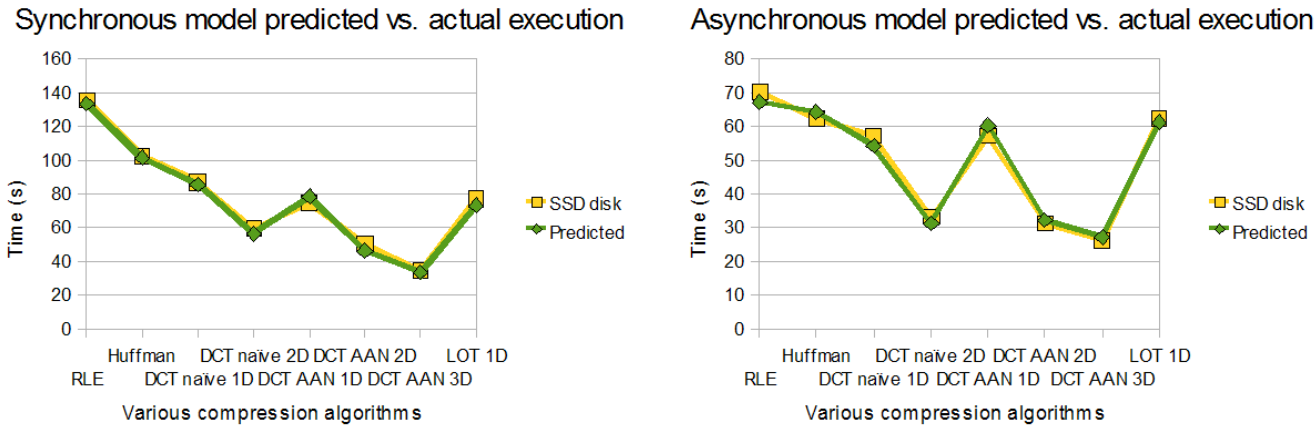

Figure 9. Predicted execution time for synchronous and asynchronous model

also tested. Note that it is twice as fast as the faster HDD disk, a figure later reflected in our results.

$$I/O\ speedup = \frac{Sequential\ I/O\ time}{I/O\ time\ w.\ compression} \quad (5)$$

Figures 6 and 7 display the I/O speedup results for our three different disks for each of the compression algorithms implemented for both our synchronous and asynchronous threaded implementations. Only the fastest resulting execution times are presented – that is, some are preformed on only the CPU and others with the aid of the GPU. The lossless algorithms as discussed earlier are the ones that benefit most from the CPU, and the transform encoding algorithms are run on the GPU for the same reasons. The CPU used to benchmark these results is the Intel Q9958, and the the GPU is the NVIDIA Tesla c2050. To see how they performed compared to other alternatives see [3].

The visual results of the data loss are displayed in Figure 8. This figure has been colored using a sharp threshold to make all changes evident, but in reality if one uses a gradient to color the seismic, the loss in close to non-visible.

A. Expected versus achieved results

Our model was tested by using a couple of measurements on a smaller block, and given those

results we estimate the execution time of the larger blocks of data. A graph representing the estimated and actual execution times for the synchronous and asynchronous model is shown in Figure 9.

Given that our models used are quite simple, the predictions were surprisingly close to the actual execution times and thus performed well. There are of course some errors that can occur, including some functions become more evident in their scaling such as standard C functions like memcpy() and memset(). But, the results prove that these estimates can be used with an error of +-5%, which is quite accurate given the simplicity of the models.

VI. Conclusions

This paper investigated how to reduce memory bandwidth and overall I/O and memory access times by using multithreaded compression and decompression of large datasets. Since the goal was to achieve a significant overall speedup of I/O, both the level of compression achieved and the efficiency of the compression and decompression algorithms, were of importance. Several compression methods for efficient disk access on both the CPU and GPU were implemented and empirically tested on large seismic datasets. To reduce I/O time, both lossless and lossy compression algorithms as well as hardware alternatives were tested. Compressed I/O and compression/decompression were also overlapped for maximum efficiency.

The lossless compression algorithms tried were RLE (run length encoding) and Huffman encoding. These were tweaked and optimized for compressing of seismic data, and resulted in a compression ratio of 0.83 and 0.71, respectively on all platforms tested. Both of these algorithms were chosen on the criteria that they are known to be fast. However, since they do not compress much of the data, the bandwidth bottleneck was still evident. Nevertheless, they resulted in 1.08 and 1.1 speedup respectively in disk access in the synchronous case, and 1.3 and 1.4 speedups respectively, in the asynchronous case. The implementations were tested two HDD disks with average transfer rates of 35MB/s and 70MB/s, respectively. Note, however, these lossless compression algorithms both gave negative speedup results when run on faster platforms such as SSD disk which averaged 140MB/s transfer rates. This showed that either the algorithms are slow or the compression is little. When tested on the GPU these algorithms performed even slower than on the CPU. This is likely because the GPU is slower on bit-wise operations and since both algorithms have a sequential nature, we are not able to take advantage of the vast parallel computation capabilities of the GPU. In other words, the CPU was superior in these cases.

Fortunately we were much more successful when it came to lossy compression. Seismic data is typically noisy, which makes it hard to compress. However, by filtering the noise and transforming the data to the frequency domain, one can achieve great compression rates with little error. We experimented with transformations in several dimensions and used algorithms that were usually used in image compression such as the DCT (discrete cosine transform), and the LOT (lapped orthogonal transform). The best compression rates were achieved by using the DCT in 3D and combining this with a modified RLE. This gave a compression ratio of 0.16. The transform in this case was performed on the GPU because of its parallel nature (which showed an 8 time speedup compared to the CPU), and the RLE was performed on the CPU because of its sequential nature.

Testing on HDD and SSD platforms, we were able to achieve respectively 3.7 and 2.5 speedups on the synchronous implementations, and a speedup of 6 and 3.2, respectively on the asynchronous model. This was done with an average error of 0.46% per float in the seismic data, which is within a reasonable loss of two decimal places. We later tested for LOT to see if we could reduce the error, but results show that this effects speedup and compression size more than the error term and that is why we did not look at this any further.

A mathematical model was further developed and empirically tested against our results. The model proved to be accurate, with up to 5% error in some cases, despite its simplistic nature. This model can be used to estimate running larger data sets, by measuring variables on smaller sets and

adapting the model to the device it runs on. The intention of the model is to give an estimate of execution time for I/O on a system using compression both synchronously and asynchronously, and it proved to be reasonably accurate for both.

VII. Future Work

Autotuning: One of the major challenges GPU programmers are facing is the blocking of the data such that separate threads can perform calculations on them. Presently, the programmer must match the block size and thread count to the capabilities of the GPU and the kernel. This greatly affects performance, and increasing thread counts or rewriting kernel for different work distribution can sometimes make the difference. This is a problem for both CUDA and OpenCL. NVIDIA has its CUDA calculator to help choose an optimal thread count to gain most occupancy, but it is also stated that anything above 50% occupancy will give an optimal execution situation. This comes from the scheduler that runs these kernels. A very useful thing to work on further is an auto-tuning program that is able to choose these factors for the programmer to optimize the running of the kernels on the GPU. This will make GPU programming easier and will automatically give the optimal running time. One could also experiment with assembly level autotuning to rewrite the kernels for efficiency.

3D visualization: During this work we have produced 2D images storing uncompressed data before visualizing it. This is of course not the case in a seismic application, which is why methods to present the data in its compressed format should be investigated. The challenge here is that the compression is non uniform, meaning that one does not know which compressed block has which part of the original image. This makes 3D representation challenging and less effective in some cases. 3D representation can be done with the help of the level of detail algorithms such that one selectively chooses what to show given the scope. Visualization is an area where compression will have great effects and present challenges for large data sets, where one has to decompress several blocks to select a few data points.

GenLOT: During this work we have been performing many transforms on the GPU in CUDA, and have mentioned how building upon the fast DCT would give room for more speedup. In our case of compression to beat I/O, it did not help speedup when we increased to the next step, namely LOT. But when looking at general compression of seismic data the GenLOT is used. A CUDA implementation of this would be interesting for testing how well the algorithm runs on the GPU. The preliminary tests show that good speedup is expected in running on the GPU.

Hardware: because of time limitations in this work, we were not able to optimize our code to run on the Fermi architecture. Although we have tested it and shown its performance, one can also look at cache optimizations for the new architecture. Another aspect that can be tested is multiple GPUs for compression. Although we tested that convolution scales almost perfectly by using several GPUs [2], the other transforms we have created in this work should be tested. Preliminary results show that the problem will scale well.

OpenCL: The end goal of our work would be to provide I/O compression for heterogeneous platforms in the form of a library. Using OpenCL may make the transition easier. There can be some optimization limitations since the compilers for C and CUDA are more optimal than that of OpenCL at the moment, which is why we avoided using it. In the future when this is no longer an issue, it is an good alternative to look into.

Finally, throughput computing remains a challenge on GPU systems [9], but as the CPUs and GPUs get integrated for more high-end systems, the bottleneck between CPUs and GPUs will shrink, and give us further opportunities to reduce the I/O bandwidth even further.

VIII. Acknowledgments

The authors would like to thank the NVIDIA Faculty Affiliation and Mad Scientist programs as well as the Department of Computer and Information Science at the Norwegian University of Science and Technology for providing support for the test beds used in this work. We would also like

to thank Dr. Victor Aarre and Mr. Christian Larsen from Schlumberger Ltd. for providing test data and the reviewers for their comments that helped improve this paper. Finally, we would like to thank Mr. Jan Christian Meyer for his last-minute help with the camera-ready version.

REFERENCES

[1] Eirik Aksnes and Anne C. Elster (both NTNU). Porus rock simulations and lattice boltzmann on gpus, in parallel computing. *From Multicores and GPU's to Petascale, Volume 19 Advances in Parallel Computing, Edited by: B. Chapman, F. Desprez, G.R. Joubert, A. Lichnewsky, F. Peters and T. Priol, April 2010, pp 533-535, hardcover,ISBN: 978-1-60750-529-7*.

[2] A. Aqrawi. 3d convolution of large datasets on modern gpus, masters fall project. Norwegian University of Science and Technology, 2009, http://www.idi.ntnu.no/~elster/pubs/index.html# aqrawi-msproject.

[3] A. Aqrawi. Effects of compression on data intensive algorithms, master of technology thesis. Norwegian University of Science and Technology, 2010, http://www.idi.ntnu.no/~elster/pubs/index.html#aqrawi-msthesis.

[4] R. L. de Queiroz, T. Q. Nguyen, and K. R. Rao. The genlot: Generelized linear-phase lapped orthogonal transform. *IEEE Transactions on Image Processing*, 1996.

[5] L. C. Duval, V. Bui-Tran, T. Q. Nguyen, and T. D. Tran. Genlot optimization techniques for seismic data compression. *IEEE Transactions on Image Processing*, 2000.

[6] R. Eidissen. Comparing cg and cuda implementations of selected transform algorithms. Dept. of Computer and Infor. Science, Norwegian University of Science and Technology, 2008.

[7] D. Haugen. Seismic data compression and gpu memory latency. Dept. of Computer and Infor. Science, Norwegian University of Science and Technology, 2009.

[8] J. Hennessy and D. Patterson. *Computer Architecture a quantitative approach*. Morgan Kaufman, third edition, 2003.

[9] Rune Hovland and Anne C. Elster (both NTNU). Throughput computing on future gpus in parallel computing. *From Multicores and GPU's to Petascale , Volume 19 Advances in Parallel Computing, Edited by: B. Chapman, F. Desprez, G. R. Joubert, A. Lichnewsky, F. Peters and T. Priol, April 2010, pp 570-577, hardcover, ISBN: 978-1-60750-529-7*.

[10] C. Larsen. Utilizing gpus on cluster computers. Dept. of Computer and Infor. Science, Norwegian University of Science and Technology, 2006.

[11] Anne C. Elster (NTNU) and Stephane Requena (Genci,Paris). *Parallel Computing on GPUs, in Parallel Computing: From Multicores and GPU's to Petascale , Volume 19Advances in Parallel Computing, Edited by: B. Chapman, F. Desprez, G.R. Joubert, A. Lichnewsky, F. Peters and T. Priol, April 2010, pp 533-535., hardcover, ISBN: 978-1-60750-529-7*.

[12] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J Phillips. Gpu computing. *Proceedings of the IEEE, Volume: 96 Issue: 5, On page(s): 879 - 899, ISSN: 0018-9219*, 2008.

[13] D. Patterson. University of California at Berkley. Personal Communication.

[14] W. B. Pennebaker and J. L. Mitchell. *JPEG Still Image Data Compression*. Van Nostrand Reinhold, 1st edition, 1993.

[15] D. Salomon. *Data Compression the complete reference*. Springer, fourth edition, 2007.

[16] Daniele Spampinato and Anne C. Elster (both NTNU). *Linear Optimization on Modern GPUs in Proceedings of 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2009), presented at Workshop on Multi-Threaded Architectures and Applications (MTAAP'09)*.

[17] Daniele G. Spampinato, Anne C. Elster, and Thorvald Natvig. Modeling multi-gpu systems in parallel computing. *From Multicores and GPU's to Petascale , Volume 19 Advances in Parallel Computing, Edited by: B. Chapman, F. Desprez, G. R. Joubert, A. Lichnewsky, F. Peters and T. Priol, April 2010, pp 562-569, hardcover, ISBN: 978-1-60750-529-7*.