Fast online predictive compression of radio astronomy data

Benjamin V. Hugo
Department of Computer Science
University of Cape Town
bennahugo@aol.com

ABSTRACT

TODO: add at end of writeup

1. INTRODUCTION

In this report we will investigate the feasibility of compressing radio astronomy data using a predictive compression scheme. All compression techniques build on the central concept of reducing redundant data. The exact definition of this redundancy is of course context dependent. It may take the form of repeated values, clustered values, wasteful encoding processes and many others. We also point out the difference between lossy and lossless compression, as well as online versus offline compression.

In a lossless compression scheme the compression is completely invertible with *no* loss of information after a decompression step is performed. Lossy compression on the other hand discards unimportant data and gives much higher compression ratios than lossless methods. Lossy compression is useful in many instances where subtle changes in data is not considered problematic. Some examples of this are the removal of high frequency data from images, sampling voice data at a lower rate than music or to employ a commonly used technique called *quantization* where data is simply binned into consecutive ranges (or *bins*).

An online compression scheme refers to a process of compressing data on-the-fly as it is being transmitted on the wire. The results are sent off to subsequent processes such transmission over a network or storage to disk. This is in contrast to to an offline scheme where data is compressed as a separate process which doesn't form part of the primary work-flow of a system. An online process is normally required to be fast enough, as not to slow the overall data processing capabilities of a system.

We will measure compression performance both in terms of effectiveness through a compression ratio described below [11, p. 10] and throughput. A compression ratio closer to 0 indicates a smaller output file and values greater than 1 indicates that the algorithm inflated the data instead of shrinking it.

$$\text{Compression ratio} := \frac{\text{size of the output stream}}{\text{size of the input stream}} \quad (1)$$

$$\text{Throughput} := \frac{\text{input processed (in GB)}}{\text{difference in time (in seconds)}} \quad (2)$$

We will now discuss the relevance of our problem and give a breakdown of the most commonly used compression techniques. Thereafter we will then discuss a detailed design of our predictive compression scheme, different implementation strategies along with technical details and a section with results and discussion.

2. BACKGROUND

2.1 KAT-7, MeerKAT and the SKA

South Africa and Australia are the two primary hosting countries for the largest radio telescope array in the world, known as the Square Kilometer Array. The SKA will give astronomers the opportunity to capture very high resolution images, over a wide field of view, covering a wide range of frequencies ranging from 70 MHz to 10 GHz. Upon completion in 2024 the array will consist of around 3000 dishes in the high frequency range and thousands of smaller antennae to cover the low frequency band. The South African branch of the SKA will be completed in 3 main phases. Phase 1 is a fully operational prototype 7-dish array called the KAT-7. The second phase, known as the MeerKAT, will consist of approximately 90 dishes. These are to be erected in the central Karoo region of South Africa. The final phase add the remaining dishes and increase the baseline of the telescope to roughly 3000 km.

Due to the high signal sampling rate it is expected that each of these dishes will produce data rates of up to 420 GiB/s, while the lower frequency aperture arrays will produce up to 16 TiB/s. These rates, coupled with the scale of the SKA, will require a processing facility capable of handling as much as 1 Petabyte of data every 20 seconds, necessitating for massive storage facilities. Innovative techniques are required to deal with this complex duel-requirement of high throughput rates while effectively reducing the large storage requirements by means of data compression. Refer to fig. 1 for an overview of the MeerKAT pipeline. Due to the limited scope of the project we propose that the compression step can be introduced as an additional step for each of the

cluster nodes, although it may be suitable for introduction on the Field Programmable Gate Arrays in future research.

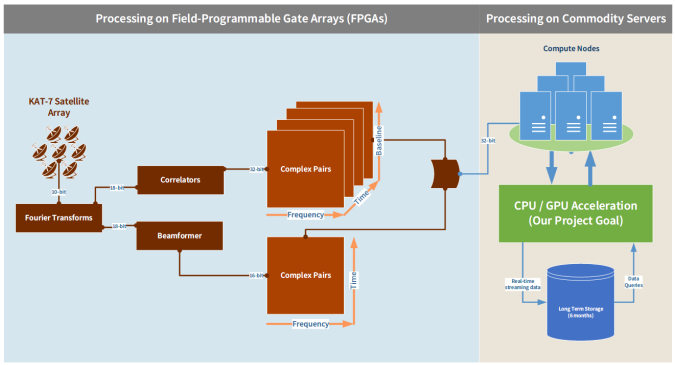

Figure 1: High-level overview of the MeerKAT pipeline

2.2 Overview of data compression techniques

There are considered to be 4 broad categories of compression techniques [11]. These are some basic methods, Lempel-Ziv methods, statistical methods and transforms.

2.2.1 *Basic methods*

The more intuitive methods include commonly employed methods such as Run-Length Encoding (RLE), which, simply put, encodes runs of characters using some reserved character and a number indicating the length of the run.

Another basic technique which is particularly relevant for application on numerical data a predictive compression scheme. Such a compression scheme encodes the difference between each predicted succeeding value and the actual succeeding value. This can be quite successfully employed to compress data generated from time series [5].

2.2.2 *Lempel-Ziv methods*

Also commonly referred to as "LZ" or dictionary methods is a class of algorithms with many variants, and is one of the more popular *adaptive* techniques in modern compression utilities. In their simplest form these methods normally use both a search- and lookahead buffer to encode recurrent phrases using fixed-size codes. An adaptive compression technique is useful in circumstances where the probability distribution of the underlying dataset is not known in advance or may change over time. One example of such an LZ method is the GNU compression utility Gzip which implements the Deflate algorithm [11, ch. 3].

2.2.3 *Statistical methods*

This class of algorithms normally uses variable length codes to achieve an optimal (or near optimal) encoding of dataset. In information theory this optimal encoding is described as an *entropy* encoding. Entropy is the measurement of the information contained in a single base-n symbol (as transmitted per unit time by some source). Mathematically redundancy is defined as follows (n is the size of a symbol set and P_i is the probability that a symbol c_i is transmitted from a source)[11, p. 46 - 47]:

$$R := \log_2 n + \sum_{1}^{n} P_i \log_2 P_i \tag{3}$$

As the name may suggest these techniques use the probability of occurrence to assign shorter codes to frequently occurring values in order to eliminate redundancy. The class of statistical methods include two widely employed techniques known as Huffman and Arithmetic coding respectively. Huffman coding assigns shorter *integral-length* codes, while arithmetic coding assigns *real-length* subintervals of [0,1) to frequently occurring symbols [15][11, ch. 2].

Both approaches lead to variable-size codes. Both techniques have adaptive versions which are useful in situations where the probability distributions change or have to be estimated. This approach is also applicable to the situation where the symbol table has to be computed on the fly, because it is impossible to perform multiple passes over data. Arithmetic coding is considered to approach a true entropy encoder and achieves higher compression ratios in both adaptive and non-adaptive cases when compared to Huffman coding. It should be pointed out that the decompression step of Arithmetic coding is slow and unsuitable for cases where fast access is required [10, 14][11, ch. 2].

2.2.4 *Transforms*

As the name suggest it can be useful to transform a dataset from one form to another in order to exploit its features for the purposes of compression. Such transformations includes, for example, wavelet transforms. As the name suggests a wavelet is a small wave-like function that is only non-zero over a very small domain and can be used to represent, for example, the high frequency components in an image (JPEG2000 and DjVu are popular formats using wavelet transforms). The coefficients within this transformation can then be further compressed using other techniques, for example, Huffman coding. If lossy compression (loss of accuracy which cannot be recovered after decompression) is tolerable, quantization can be used to discard unimportant values (for example the high frequency features of an image) [12][11, ch. 5].

Transforms are furthermore particularly useful where multiple levels of detail are desired. An example of this may include the transfer of scientific data over a network for real-time analysis and observation. Low resolution samples can be constantly transferred, while higher resolution samples can be transferred upon request [13].

2.3 IEEE 754

The IEEE 754 standard of 2008 defines 3 interchange formats (32-bit, 64-bit and 128-bit). See figs. 2 , 3. We will only refer to the 32- and 64-bit representations later in the paper. Each of these have a common construction with the following subfields:

- A 1-bit sign
- A w-bit length biased exponent
- A (d-1)-bit significand, where the leading bit of the significand is implicitly encoded in the exponent.

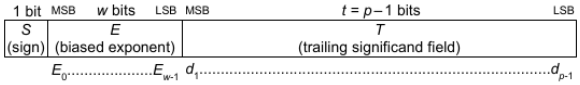

Figure 2: IEEE Interchange floating-point format [1]

Precision	Exponent width	Significant precision
32-bit	8 bits	23 bits
64-bit	11 bits	52 bits

Figure 3: Specifications for the 32-bit and 64-bit interchange formats

2.4 Overview of predictive compression

Previous research [9, 3, 5, 7, 8, 4] in this area has yielded good results both in terms of compression ratio and speed. The common line of thought is to predict successive values. Depending on the accuracy of the prediction, the difference between the predicted and actual value will be much smaller than the actual value itself. This difference can then be encoded using fewer bytes/bits of data (depending if compression occurs at a byte or bit level) by compressing the leading zeros after either an XOR or integer subtraction operation. Machine instructions to count the leading zeros can be found on AMD and newer Intel Processors. The leading zero count is then encoded as a prefix stream while the remaining bits/bytes are encoded as a residual stream.

Previous research suggests several different constructions of predictors. The use of a Lorenzo predictor [7] generalizes the well known parallelogram predictor used by [5]. This approach is particularly useful to compress large meshes. Other approaches include the use of prediction history (via a simple lookup table construction) as suggested in [9, 3, 4]. The later reports a throughput of up to 670 MB/s on a CPU implementation of their FPC compressor. An even simpler scheme [8] reportedly achieved throughputs of up to 75GB/s in its compression routine and up to 90GB/s in its decompression routine implemented in CUDA. In this scheme only the difference between successive values are encoded (this will clearly only work if the pairs of data points varies very little from one time step to the next). It is duly noted that the speeds achieved by [8] is on the post processing of results already stored in graphics memory. The implementations by [8, 9, 3, 4, 5] targets 64-bit IEEE 754 *double* precision floating-point data.

The primary scheme we propose takes its inspiration from [8], but will operate on 32-bit IEEE 754 *single* precision floating-point values. The uncompressed data itself is also structured slightly differently to the model used by [8]. Instead of compressing each consecutive value our compressor will operate over consecutive blocks of data. Although the predictor itself is very simple and may not yield as good a compression ratio as the schemes suggested by the other authors it should obtain the required throughput. Additionally we will test the effectiveness of a parallelogram predictor, a Lagrange extrapolation predictor [5] and a moving mean and median scheme. We will describe each implementation to greater extent in the implementation section.

As pointed out in previous research [5, 7] using floating point operations in the prediction step can cause an irreversible loss of information due to floating point rounding errors. A simple approach is proposed by [5]: floating point memory is simply treated as integer memory and all operations performed on that memory are integer operations. This approach assures us of achieving lossless compression and we will consequently only consider using schemes that conform to this approach. A simple case to illustrate this approach is made by [5]. It is clearly visible in fig. 4 that if two floating-point numbers are relatively close to each other in terms of magnitude it is possible to discard some repeated bytes of information after performing an XOR operation to extract the remaining, seemingly random, residual bits. Therefore if we can accurately predict consecutive numbers we should be able to make reasonable savings in terms of compression ratio.

		byte	1 2 3 4 5 6 7 8
a_1	2.3667176745585676	⇒	40 02 ef **09 ad 18 c0 f6**
a_2	2.3667**2**76745585676	⇒	40 02 ef **0e eb 46 23 2f**
a_3	2.3667**3**76745585676	⇒	40 02 ef **14 29 73 85 6a**

Figure 4: Treating 64-bit IEEE 754 double precision floating-points as integer memory [5]

2.5 Parallel prefix sums

The computation of a prefix sum is a necessary step in the parallelization of a packing algorithm [8]. A prefix sum (or *scan*) can be defined on any binary associative operator, $\oplus$ over an ordered set with I as its identity. If $A = [a_0, a_1, \ldots, a_{n-1}]$ is an ordered set then the prefix sum scan is defined as $scan(A) := [I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus ... \oplus a_{n-2})]$.

This scan operation can be computed in parallel. Blelloch suggests such an approach in [2]. We will discuss the algorithm in more detail in the section on implementation. Additionally a work-efficient CUDA version of the algorithm is discussed in [6]. In this version the algorithm is optimized to avoid bank conflicts and has been shown to be up to 6 times faster than a sequential CPU implementation for large arrays.

3. RESEARCH QUESTIONS

We are investigating the feasibility of adding a *online* predictive compression step to the existing KAT-7 / MeerKAT pipeline. Such a step has to meet at least two primary criteria: high throughput and effective compression ratios. These are outlined below:

1. Are predictive techniques fast enough? The algorithm should be able of achieving throughput rates of at least 40 GiB/s.

2. Are predictive techniques effective? The algorithm should reduce the size of transmissions by several percent and hopefully this reduction can take the form of double digit figures. It has, however, been pointed out that the data may be too noisy to expect great reductions, while maintaining the throughput rate we mentioned above.

3. Can throughput be traded for compression ratio using different predictors?

4. DESIGN AND METHODOLOGY

4.1 Overview

In light of our research questions and the limited scope of this report we will investigate the usefulness of a predictive data compression step in the context of the MeerKAT project. We will focus solely on investigating whether the algorithm can operate at the desired line rate of 40 GiB/s and whether it provides reasonable compression ratios. We will not investigate implementing the algorithm as part of the networking stack employed in the KAT-7 pipeline. By reasonable here we mean if we can get similar compression ratios to industry standard tools like Gzip and Bzip2. We will also investigate the possibility of using alternative predictive schemes to trade some throughput for a better compression ratio. These schemes will include the following (we will provide notes on their efficient implementation in the implementation section):

1. A Lagrange polynomial extrapolation.
2. A parallelogram predictor.
3. Moving mean & median predictors.

We will evaluate the success of our undertaking depending on the following criteria:

1. Throughput in excess of 40 GiB/s is achieved by both the compressor and decompressor.
2. Effective compression ratios, comparable to those achieved by gzip and bzip2 are achieved.

4.2 Packing algorithm

The compacting algorithm we propose is based on the approach taken by [8]. The algorithm is almost *symmetrical* in its compression and decompression stages. By symmetrical we infer that the algorithm is executed in the opposite direction when performing decompression. The primary difference is that we do not have to compute the leading zero count of each element being decompressed. Refer to fig. 5 for more details.

We use the construction specified in fig. 5 because we can easily swap out one predictor for another. The predictor in this case can be any n-element predictor. As we mentioned earlier the predicted value is XORed with the actual value to get a residual which can be compacted at either bit or byte level. This compaction simply removes the leading zeros from each residual and store each residual as a fixed-length prefix count. We note that the number of bits needed to store n leading zeros can be calculated as $\lfloor \log_2 n \rfloor + 1$ bits where $n \in \mathbb{N}_{>0}$. One aspect of this compression scheme that must be investigated is to determine which residual compaction scheme gives better results: packing at bit or byte level. Although a bit-packing scheme can pack an arbitrary number of zeros it requires longer prefixes to store these counts. The opposite is true for byte-packing schemes. Since the prefixes are always short we use a bit-packing scheme to store these fixed-length codes.

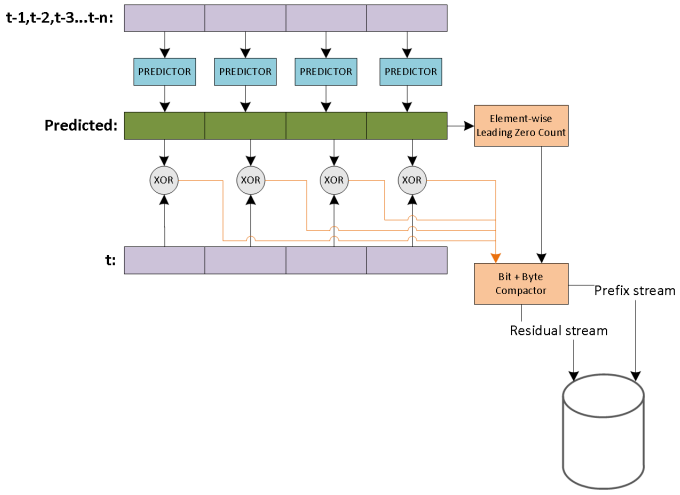

Figure 5: Prediction and compaction process

4.3 Parallelizing compression and decompression steps

We note that there are three ways this can be achieved:

1. Even though the residuals are of variable length it is possible to pack them in parallel. However, it is duly noted that this adds an additional overhead to *both* the packing and unpacking process. The process requires a prefix sum to be computed for the array of element-wise leading zero counts. As we pointed out in the background chapter this prefix sum can be computed in parallel. After the indexes have been accumulated using the prefix sum algorithm it is easy to see how different threads can pack residuals at the correct positions. We do, however, note that both a parallel bit and byte packing scheme will require *atomic* operations to ensure against the *race condition* that arises when multiple threads writes to the same memory location simultaneously. This memory-level-synchronization mechanism adds additional overhead in terms of wasted machine clock cycles.
2. We can separate incoming data into blocks and do multiple of these blocks in parallel (where each block is compressed/decompressed in serial). This overcomes the issue of additional overheads arising from computing prefix sums and using atomics, but it should be noted that this approach may have a detrimental effect on the compression ratio since we have to store the length of the residual array of each block. We will have to investigate which of 1 or 2 yields better results for CPU implementations.
3. A more fine-grained parallelization is possible using vectorized instructions. The various Intel SSE, Steaming SIMD (Single Instruction Multiple Data) instruction set extensions (up to version 4.2) provide us with the opportunity to perform 4 instructions (adds, multiplies, logarithms, rounding, etc.) simultaneously per core. However, the SSE instruction sets do not have any element-wise bit-shift operations. We propose using a combination of the Intel SSE and AMD XOP

instruction sets to achieve our goal. The XOP instruction set extends the SSE instruction set by adding these operations as one of its primary features. It is duly noted that adding these instructions make the implementation dependent on the architecture of the machine it is executed on (and therefore less portable). If this approach is successful we recommend that the vectorized code be extended to use the latest Intel AVX 2 instruction set in which Intel introduces element-wise shift operations and offset-loading operations. The AVX family of instructions can compute up to 8 SIMD operations in parallel per core. It is duly noted that these instruction sets make use of a very limited number of 128- and 256-bit registers respectively. We will need to investigate whether it is worthwhile to implement the proposed packing algorithm using SSE and XOP instructions.

4.4 Porting the implementation to CUDA

General Purpose programming using Graphics Processing Units (GPGPU) brings a host of associated challenges. These challenges arise primarily due to the widely differing architecture of GPUs if compared with the classical architectures of Central Processing Units, CPUs. Refer to fig. 6 for a detailed overview of the microchip die structure of a Nvidia Fermi generation GPU. It is also worthwhile to note that each generation of GPUs have widely differing architectures, making many optimizations architecture specific. Each generation, for instance, will have a different ratio and configuration of arithmetic cores to floating point and special operations cores per group of processing *CUDA cores* also known as *SMs* (or *SMXs* on the latest *Kepler* generation). Each generation also has widely differing memory controller properties. These may determine how many warps of cores can be executed, when the warps are swapped in/out, along with many others.

We will provide a basic CUDA implementation to see if doing compression as a post-processing step on signal data already loaded to GPU (as part of the current pipeline) is a feasible alternative to a regular multi-threaded CPU implementation. We will make use of shared memory (as fig. 6 shows GPUs have very large L2 caches available per SM) and our implementation will take adequate precaution to coalesce memory accesses and to avoid bank conflicts when possible. More details will be provided in the implementation section, but it is clear that a combination of the approaches (1 & 2) taken with the CPU version will be required to make adequate use of the GPU architecture. Our implementation will consider assigning a block of data to each SM, which will in turn process that block in parallel without regard to operations performed on other SMs. This ensures that we do not create unnecessary overhead synchronizing SMs (this is considered to be a very costly synchronization for this type of architecture). Instead all atomic operations and synchronization steps will be performed on a per-SM basis only.

4.5 Test data

The post-correlated data is provided to us in the form of HDF5 files. Each of these files store a 3 dimensional array of complex pairs. The first dimension is time. The second dimension is signal frequency (as we mentioned in the background section the telescope array will monitor a wide range of frequencies). The third dimension is a combination of two properties. Each number represents a correlation between two antennae. The antennae additionally have two modes of polarization (a horizontal and vertical polarization). We can specify the size of the last dimension as $2 \times ((n+1)\mathbf{P}2)$ elements where n is the number of antennae in the array. The headers of the HDF5 files contain meta information on the state of the telescope and what it is currently observing (this will determine the characteristics of the data itself as well as the range of frequencies being monitored). We state for the record that the headers cannot be compressed with the algorithm we propose. An LZ variant / entropy encoder may be better tailored for this task.

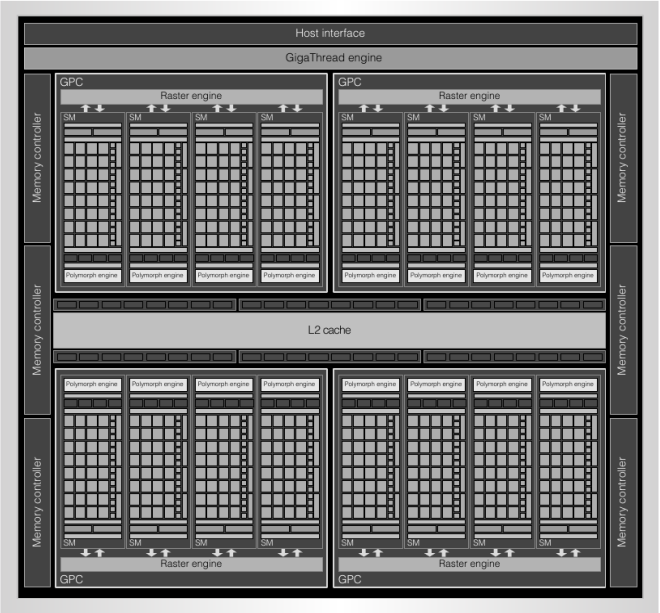

Figure 6: Architecture of the Fermi family of Nvidia GPUs [16]

4.6 Structure and scope of the benchmarks

Some of these HDF5 files measure more than 30GB in size and contains many timestamps, some over 4000 frequencies. We assume that once the number of correlations grow, emphasis will be placed on processing each time step (or portion thereof), as soon as it is received over the network. In future research the compressor may be integrated into the 7-layer OSI networking stack currently employed by the KAT-7 prototype. Due to the limited scope of this project we will only analyze the performance of the algorithms themselves and will not measure any disk I/O (except when comparing to other compression utilities for the sake of fairness).

We propose that the following results should be collected:

1. We will draw a comparison between the a CPU implementation of the parallel (prefix sum-based algorithm) and the blocked approach.

2. We will provide a detailed breakdown of how the algorithm perform on different numbers of cores of whichever

method is determined most fit in step 1.

3. We will investigate the feasibility of an implementation using the Intel SSE and AMD XOP instruction sets.

4. We will benchmark the algorithm against the standard programs used for comparison: gzip, bzip2, zip and rar. In this case the disk I/O will be included for the sake of fairness.

5. We will investigate the feasibility of a CUDA implementation as we outlined earlier.

6. Finally we will compare results from concurrent research being conducted into entropy encoding and run-length encoding. This will include a break down of performance in terms of both throughput and compression ratios.

In order to eliminate the influence of the external testing environment we specify the following special conditions:

1. Files in excess of 500 MB will be used for benchmarking. This will greatly reduce variability in timings.

2. No other processes should be making intensive use of primary memory once benchmarking has commenced. The sheer quantity of data being processed makes benchmarking a memory-bound operation.

4.7 Benchmarking platform

The benchmarking platform is designed with two primary software engineering goals in mind:

- It should be as *decoupled* as possible, making use of predefined interfaces. This will assist in providing several implementations of the CPU version (for example SSE, different linear predictors and different parallelization approaches) and simply swapping one implementation for another. Interactions between components must be kept to a minimum.

- The components must be *cohesive*. They should contain only relevant operations and should be considered as atomic units.

We propose that the architecture described in fig. 7 be used for the benchmarking platform. We will give more detailed implementation details in the next section. The benchmark process is relatively simple and its flow is depicted in fig. 8.

5. IMPLEMENTATION

TODO

6. FINDINGS

TODO

7. DISCUSSION

TODO

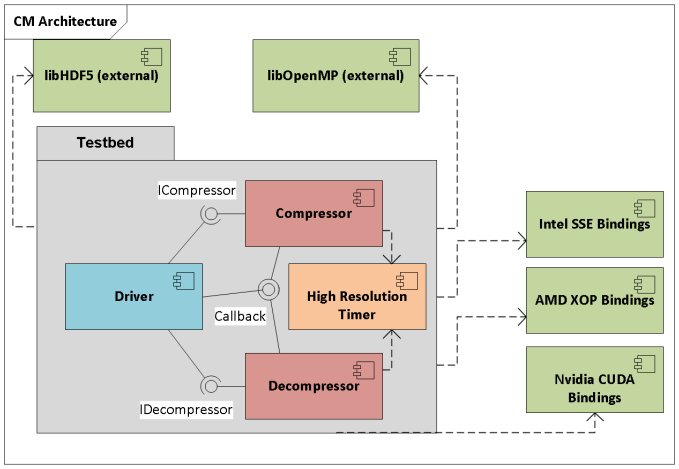

Figure 7: Architecture of the benchmarking platform

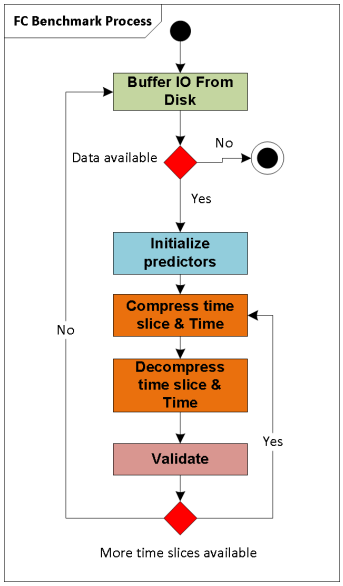

Figure 8: Behavioral model of the benchmarking platform

8. CONCLUSION

TODO

9. FUTURE AVENUES OF RESEARCH

TODO

10. ACKNOWLEDGEMENTS

I would like to acknowledge A/Prof. James Gain and Dr. Patrick Marais of the Department of Computer Science at the University of Cape Town for their continuous, expert, input on the project.

Secondly I would like to thank Jason Manley, a Digital Signals Processing specialist at the MeerKAT offices in Pinelands, Cape Town for providing us with technical information on the MeerKAT project. Jason has also kindly prepared a 100 GB of sample of KAT-7 output data for testing purposes.

Thirdly I would like to note that all tests were performed on

the ICTS High Performance (*HEX*) cluster at the University of Cape Town. The cluster has 4 DELL C6145 nodes each boasting 4 16-core AMD Opteron 6274 CPUs, clocked at 2200 MHz with 16 MB L3 cache memory. There are two additional GPU nodes with 4 Tesla M2090 GPU cards each. Each GPU card has 6 GB GDDR5 and 2048 CUDA cores. I want to especially thank Andrew Lewis from ICTS for his assistance and support during testing.

This research is made possible under grant of the National Research Foundation (hereafter *NRF*) of the Republic of South Africa. All views expressed in this report are those of the author and not the NRF.

11. GLOSSARY

TODO

12. REFERENCES

[1] Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.

[2] Guy E Blelloch. Prefix sums and their applications. 1990.

[3] M. Burtscher and P. Ratanaworabhan. Fpc: A high-speed compressor for double-precision floating-point data. *Computers, IEEE Transactions on*, 58(1):18–31, 2009.

[4] M. Burtscher and P. Ratanaworabhan. pfpc: A parallel compressor for floating-point data. In *Data Compression Conference, 2009. DCC '09.*, pages 43–52, 2009.

[5] Vadim Engelson, Dag Fritzson, and Peter Fritzson. Lossless compression of high-volume numerical data from simulations. In *Data Compression Conference*, pages 574–586. Citeseer, 2000.

[6] Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel prefix sum (scan) with cuda. *GPU gems*, 3(39):851–876, 2007.

[7] Peter Lindstrom and Martin Isenburg. Fast and efficient compression of floating-point data. *Visualization and Computer Graphics, IEEE Transactions on*, 12(5):1245–1250, 2006.

[8] Molly A. O'Neil and Martin 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*, GPGPU-4, pages 7:1–7:7, New York, NY, USA, 2011. ACM.

[9] P. Ratanaworabhan, J. Ke, and M. Burtscher. Fast lossless compression of scientific floating-point data. In *Data Compression Conference, 2006. DCC 2006. Proceedings*, pages 133–142, 2006.

[10] Gautam Ray, Jayant R Haritsa, and S Seshadri. Database compression: A performance enhancement tool. In *International Conference on Management of Data*, page 4, 1995.

[11] D. Salomon. *Data Compression.: The Complete Reference.* Springer-Verlag New York Incorporated, 2004.

[12] A. Skodras, C. Christopoulos, and T. Ebrahimi. The jpeg 2000 still image compression standard. *Signal Processing Magazine, IEEE*, 18(5):36–58, 2001.

[13] Hai Tao and Robert J. Moorhead. Progressive transmission of scientific data using biorthogonal wavelet transform. In *Proceedings of the conference on Visualization '94*, VIS '94, pages 93–99, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[14] Hugh E Williams and Justin Zobel. Compressing integers for fast file access. *The Computer Journal*, 42(3):193–201, 1999.

[15] Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Commun. ACM*, 30(6):520–540, June 1987.

[16] Craig M Wittenbrink, Emmett Kilgariff, and Arjun Prabhu. Fermi gf100 gpu architecture. *Micro, IEEE*, 31(2):50–59, 2011.