University of Cape Town

Department of Computer Science

CSC4000W - Honours in Computer Science

Fast online predictive compression of radio astronomy data

Author:
Benjamin Hugo[1]

Team members:
Brandon Talbot
Heinrich Strauss

Supervisors:
A/Prof. James Gain
A/Prof. Patrick Marais

External Advisor:
Jason Manley

A thesis presented to the Department of Computer Science, UCT, in partial fulfillment of the requirements of the course CSC4000W, Honours in Computer Science.

October 25, 2013

	Category	**Min**	**Max**	**Chosen**
1	Requirement Analysis and Design	0	20	10
2	Theoretical Analysis	0	25	0
3	Experiment Design and Execution	0	20	5
4	System Development and Implementation	0	15	15
5	Results, Findings and Conclusion	10	20	15
6	Aim Formulation and Background Work	10	15	15
7	Quality of Report Writing and Presentation	10		10
8	Adherence to Project Proposal and Quality of Deliverables	10		10
9	Overall General Project Evaluation	0	10	0
Total marks		**80**		**80**

[1]The financial assistance of the National Research Foundation (NRF) towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the author and are not necessarily to be attributed to the NRF.

Abstract

This report investigates the fast, lossless compression of 32-bit single precision floating-point values. High speed compression is critical in the context of the MeerKAT radio telescope currently under construction in Southern Africa and Australia, which will produce data at rates up to 1 Petabyte every 20 seconds. The compression technique being investigated is based on predictive compression, which has proven successful at achieving high-speed compression in previous research. Several different predictive techniques (which includes polynomial extrapolation), along with CPU- and GPU-based parallelization approaches are discussed. The implementation successfully achieves throughput rates in excess of 6 GiB/s for compression and much higher rates for decompression using a 64-core AMD Opteron machine, achieving file-size reductions of, on average 9%. Furthermore the results of concurrent investigations into block-based parallel Huffman encoding and Zero-length Encoding are compared to the predictive scheme and it was found that the predictive scheme obtains approximately 4%-5% better compression ratios than the Zero-Length Encoder and is 25 times faster than Huffman encoding on an Intel Xeon E5 processor. The scheme may be well-suited to address the large network bandwidth requirements of the MeerKAT project.

Acknowledgements

I would like to acknowledge A/Prof. James Gain and A/Prof. 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 GiB of sample of KAT-7 output data for testing purposes.

Thirdly I would like to note that the majority of 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 MiB L3 cache memory. There are two additional GPU nodes with 4 Tesla M2090 GPU cards each. Each GPU card has 6 GiB 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 necessarily those of the NRF.

Glossary

3DNow! An AMD vectorized instruction set that extended the capabilities of Intel's MMX instruction set by adding supporting instructions for 32-bit floating-point processing.

Advanced Vector Extensions (AVX) An Intel extension to SSE instruction set that adds 256-bit registers (which may be used to perform simultaneous operations on up to 8 32-bit integers/floating-point values).

Arithmetic coding An entropy encoder that assigns sub-intervals of [0,1) to the symbols it processes. This encoder is closer to an optimal entropy encoder, achieving better compression than Huffman coding. See background chapter.

Bank conflict On a GPU a bank conflict occurs when multiple threads try to access the same bank of shared memory. Such accesses will then be completed sequentially. See design chapter.

BZIP2 A standard free compression utility that implements the Burrows-Wheeler Transform (which groups matching elements into runs, in order to perform Run-Length Encoding), along with Huffman encoding.

Coalesced memory access On a GPU a coalesced read occurs when the global memory calls of multiple threads are grouped together in order to share the latency of such a call between threads. This normally requires memory aligned access patterns. See design chapter.

Cohesion A software engineering term used to measure the the degree to which a component separates operations. Ideally a component should only house supporting operations to perform a single task to ease future maintenance and usability.

Compression Removal of redundancy according to some statistical model of the data. This redundancy may be repeated values or wasteful character encoding practices.

Compression ratio Ratio between output size and input size.

Compute Unified Device Architecture (CUDA) A programming framework used to compile programs for Nvidia GPUs.

Decompression The inverse of compression. Unpacks a file to its original state or a similar representation thereof (depending whether lossy or lossless compression is desired.

Decoupling A software engineering term used to measure the dependency of one component on another.

Discrete Cosine Transform (DCT) A finite sequence of samples can be represented as a sum of transformed cosine functions. This is one example of where transforms can be used to achieve level-of-detail-based compression. See background chapter.

Entropy The measurement of the information contained in a single base-n symbol (transmitted per unit time by some source).

Exclusive Or (XOR) A bitwise, binary (taking two inputs), operation that returns false if the inputs are identical.

eXtended Operations (XOP) An AMD extension to SSE instruction set that adds, amongst others, the ability to perform per-element bit-shifting on 4 integers contained in a 128-bit register.

General Purpose Graphical Processing Unit Programming (GPGPU) General purpose (for instance scientific computing) programming using SIMD style instructions on a GPU.

GNU ZIP (GZIP) A standard free compression utility that implements the DEFLATE (LZ-77 based) algorithm.

Graphical Processing Unit (GPU) Independent streaming processor that is tailored towards processing graphics.

HDF5 file format A hierarchical file format that supports the storage of large sets of numerical data and associated meta-data that is generally faster than storage using relational databases[2].

Huffman coding An entropy encoder that assigns shorter codes to frequently used symbols in a dataset. See background chapter.

IEEE 754 A widely employed standard defined for the storage of 32-bit, 64-bit or 128-bit floating point values. See background chapter.

Intel Multimedia eXtensions (MMX) A vectorized instruction set that added support for performing operations on two 32-bit integers through the 64-bit registers this technology added. The MMX instruction set was extended by the AMD 3DNow! extensions and later the SSE extensions.

KAT-7 Initial 7-dish prototype radio telescope array constructed near Carnarvon, South Africa.

Lagrange predictor A polynomial extrapolation technique used for prediction of successive values for smooth functions. See design section.

Lempel-Ziv (LZ) Compression A compression scheme where repeated sequences of symbols are encoded as distance-length pairs. See background chapter.

Lorenzo predictor A scheme that extends the parallelogram rule to higher dimensions which can be used for polynomial reconstruction. See design section.

[2]The HDF Group. Hierarchical data format version 5, 2000-2010. http://www.hdfgroup.org/HDF5.

Lossless Compression Can be inverted with no loss of information. The opposite will be lossy compression.

MeerKAT Successor to the KAT-7 prototype.

Merkle Damgard 5 (MD5) An iterative message digest algorithm that creates a fixed-length checksum of a message, with the intension of verifying data integrity. See section on approach feasibility.

Online Compression Compression is completed as part of the primary function of the system (and not afterwards). An online process is normally required to add no significant latencies to an existing system.

Open Systems Interconnection (OSI) network stack An abstraction of the layered nature of network protocols. From top to bottom the stack contains the following layers: application, presentation, session, transport, network, link and physical layers. A decompression operation may form part of the presentation layer in such a stack.

OpenMP A package that allows for easy parallelization of C and Fortran code using compiler flags.

Petabyte 1024 Terrabytes (TiB).

Pinned host memory A method used to allocate a non-swappable block of primary memory that improves memory transfers between a host machine and a GPU.

Pivot A pivot divides a dataset into 2 groups (useful in applications of fast sorting techniques). One group is larger than the pivot, while the other is smaller (or equal) to the pivot.

Pixel Picture element (or colour value). An image is a discretized raster of these pixels.

POSIX threads (pthreads) An API that allows for the creation of native process-based threading on Unix-like systems.

Prefix sum An element-wise accumulation of an array of values over some binary associative operator. See design chapter.

Quantization In terms of lossy compression this process refers to binning values in close proximity into groups.

Race condition A condition that occurs when two threads compete to update the same resource in which correctness depends on the order and timing of these updates. This constitutes non-deterministic behavior which invalidates the state of such a resource. See design chapter on how this affects a parallel packing algorithm.

Redundancy The difference between the optimal entropy of a dataset and the actual entropy.

Run-Length Encoding (RLE) A technique that clusters runs of symbols into length-based representations (aaabbbb is represented as 3a4b for example). See the introduction.

Scanline A horizontal row of pixels that stretches across the width of the image.

Shared memory In CPUs shared memory refers to the sharing of primary memory between multiple processor cores, or even multiple processors. In GPUs the threads within a single SM has access to dedicated fast memory that can be used to avoid calls to global device memory. See design chapter.

Simple Instruction Multiple Data (SIMD) Can apply to contexts where a large number of elements are subject to the same instruction. A good example where this style of programming can be used is vector addition.

Square Kilometer Array The combined low and high frequency radio telescope arrays constructed in Southern Africa and Australia will be collectively known as the SKA.

Streaming Multiprocessor (SM) A sub-component of Nvidia GPU architecture. Such a multiprocessor consists of many arithmetic units which operate in parallel. A GPU will normally have multiple of these multiprocessors which will operate independently of eachother unless a global synchronization barrier is imposed. See design chapter.

Streaming SIMD Extensions (SSE) A vectorized instruction set (see design and implementation chapters) that adds the capability of performing SIMD instructions on 128-bit registers (for example 4 32-bit integers) in one machine clock cycle.

Symmetrical algorithm In terms of compression and decompression this refers to executing a compression algorithm in reverse in order to achieve decompression.

Throughput input processed in GiB per second.

User Datagram Protocol (UDP) A fast networking protocol that sends data packets with no built-in support for verifying package integrity, and checking (and by extension correcting) transmission losses.

Warp of threads In a GPU each Streaming Multiprocessor is divided into atomic groups of threads that executes in parallel. See design chapter.

Zero-copy A support mechanism in the Nvidia API that allows for sharing pinned host, and device memory spaces.

ZIP A standard compression utility that implements a collection of LZ, entropy and transform methods that is useful when archiving a group of files.

Contents

List of Figures

1 Introduction

1.1 The KAT-7, MeerKAT and SKA

South Africa and Australia are the two primary hosting countries for what is to become the largest radio telescope array in the world, known as the Square Kilometer Array (SKA). 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 MeerKAT, will consist of approximately 90 dishes. These are to be erected in the central Karoo region of South Africa. The final phase will 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 GBit/s, while the lower frequency aperture arrays will produce up to 16 TBit/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 massive storage facilities. Data compression may prove useful to reduce both the network bandwidth requirements, as well as long term storage requirements.

1.2 Data compression & decompression

Data compression seek to encode data using fewer bits than that of the original encoding. By removing such redundant data it is possible to effectively reduce the size of the dataset. A simple illustration of this can be found in figure 1. It is clear that if this picture is stored naïvely as a 20x20 grid of 24-bit colours (8-bits for each of the red, green and blue channels) 1200 bytes of data will be written to disk. A simple, yet effective, technique to reduce redundancy in this example is to store each horizontal line (or *scanline*) of pixels as runs of colours, instead of individual colours. The first scanline can therefore be reduced to 9 white, 5 white, 5 orange, 1 white runs of pixels. This is known as *Run-Length Encoding*. However, this is not the only form of redundancy: if each colour is represented as a 24-bit value, but only 5 colours are used from the 2^{24} colours available, a lot of storage space is wasted. In this case only $\lfloor \log_2 5 \rfloor + 1 = 3$ bits are needed to store all 5 colours uniquely. This variable-length encoding is often accomplished using an entropy encoding scheme, which will be described to greater detail in the background section. Decompression can be thought of as the inverse operation to the compression process, as it reconstructs the original data from its encoded version.

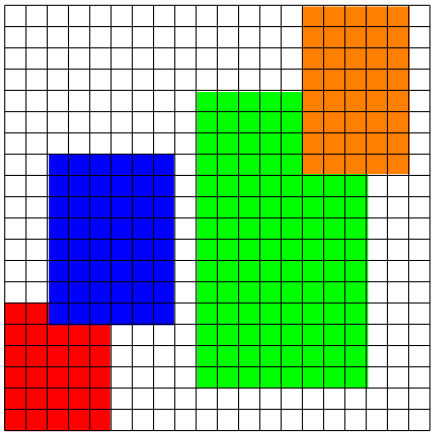


FIGURE 1: 5-colour image illustrating encoding redundancy. Instead of storing a 20x20 raster of colours, runs of colours (per row / column) can be stored, along with variable-length representations of the colours, instead of using 24 bits per colour.

1.3 Predictive compression of KAT-7 data

All compression techniques build on the central concept of reducing redundant data. The exact definition of this redundancy is, of course, context dependent. In the case of the KAT-7 / MeerKAT array this redundancy may be defined in terms of the coherence of consecutive observations from each *correlated* pair of dishes. Instead of building a single dish with a large collecting surface, which is obviously only feasible for relatively small collecting areas, the current trend of thought is to combine the collecting capabilities of hundreds of small dishes to artifically create an equivalent collecting area to that of a single large antenna. In simplest terms this correlation operator crosses the output between pairs of antennae and therefore scales quadratically as the number of antennae are increased.

If the observations are not particularly noisy it is reasonable to assume that the differences between consecutive values will be small, since every observed frequency will stay relatively constant, or grow and decay slowly. Rather than storing each value it is possible to store the differences between observations (each over a fixed time interval) instead. The array observes a large spectrum of frequencies over a large number of correlated pairs, as indicated in figure 2. Each of these correlated frequency observations can be considered as steps of a time series. This report investigates how linear prediction can be employed to predict consecutive values for each of these time series. The goal is to minimise the difference between consecutive values, which can then be encoded using fewer bits than the original 32-bit sample size that is received by the processing and storage cluster. Such a scheme must satisfy the additional requirements of being *lossless*, *online* and fast.

1.4 Compression algorithm properties and measurements

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

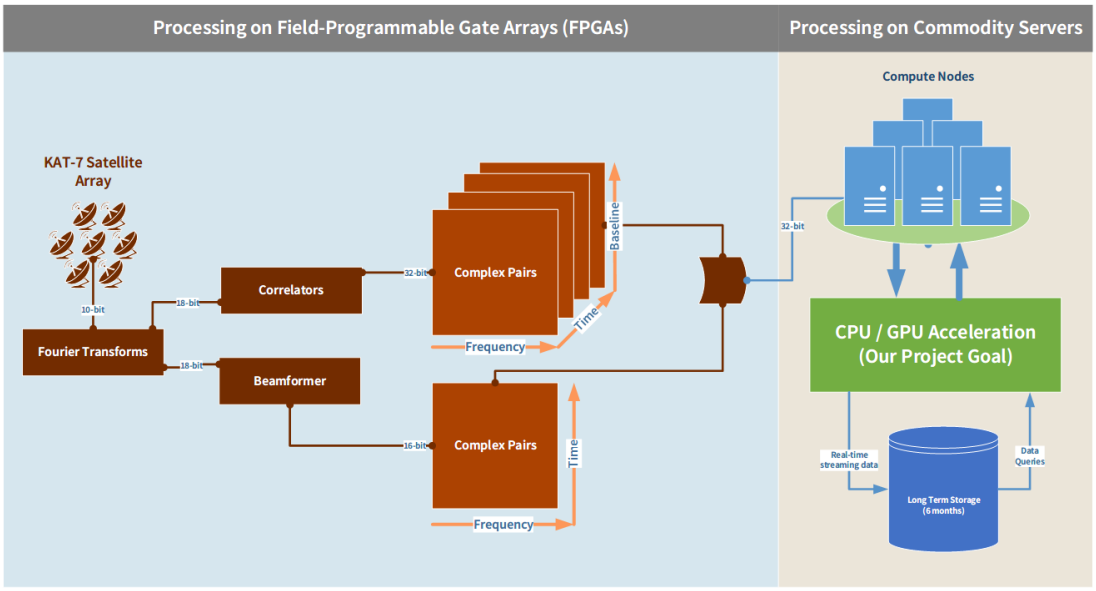


FIGURE 2: High-level overview of the MeerKAT pipeline. Although future work should focus on implementing compression near the front of the pipeline in order to save on network bandwidth, only the feasibility of achieving compression at linerates of 40 GiB/s are being considered in this investigation.

hand, discards unimportant data and achieves much higher compression ratios than lossless methods. Lossy compression is useful in many instances where subtle changes in data are not considered problematic. This includes the removal of high frequency data from images, sampling voice data at a lower rate than music, or employing 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. The results are sent off to subsequent processes such as transmission over a network or storage to disk. This is in contrast to an offline scheme where data is compressed as a separate process that does not form part of the primary work-flow of a system. An online process is normally required to be sufficiently fast that it does not to slow the overall data processing capabilities of a system.

Compression performance will be measured both in terms of effectiveness through a compression ratio described below [15, p. 10] and throughput. A compression ratio closer to 0 indicates a smaller output file and values greater than 1 indicate that the algorithm inflated the data instead of shrinking it. Refering back to the example on Run-Length Encoding (RLE) and variable length codes: using a standard compression utility known as BZIP2 the grid is compressed using RLE, in combination with the Burrows-Wheeler Transform and a variable length encoder known as a Huffman encoder (both will be fully described in the background section of this report) the size of the output is reduced from 1200 to a mear 84 bytes. The compression ratio will therefore be computed as

84 bytes/1200 bytes = 0.07; a saving of 93%.

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

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

1.5 Research questions

Due to the limited scope of the project this report will focus on evaluating algorithm performance rather than integration into the KAT-7 / MeerKAT processing pipeline. This research will include the construction of a parallel algorithm, as well as the investigation of the feasibility of porting this algorithm to the General Purpose Graphics Processing Unit (GPGPU)-accelerated nodes currently employed by the KAT-7 signal processing nodes, as illustrated in figure 2.

A predictive compression implementation has to meet two primary criteria: high throughput and effective compression ratios. These are outlined below:

1. Are predictive techniques fast enough? The algorithm should be able to achieve throughput rates of at least 40 GBit/s. These speeds represent Infiniband network speeds and are a reasonable first milestone towards achieving compression at line rates.

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 by the SKA office that the data may be too noisy to expect significant reductions, while maintaining the throughput rate mentioned above. Even though shaving a few percent off the output size may seem insignificant, in the context of the SKA this can translate back to a saving of 71.68 TiB every 20 seconds (if peak transfer rates of 1 PiB are achieved and compression ratios of 93% can be maintained). This may result in significant monetery savings, both in initial network and storage infrastructure, and long term maintenance costs on storage.

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

Next a breakdown of the most commonly used compression techniques is given. Thereafter we will present our design of a predictive compression scheme, different implementation strategies, along with technical details, and a section with results and discussion.

2 Background

2.1 Overview of data compression techniques

There are considered to be 4 broad categories of compression techniques [15]:

1. Basic methods include commonly employed methods such as *Run-Length Encoding* (RLE), *Move-to-Front Coding* and *Prediction-based* compression.
2. Lempel-Ziv stores repeating sequences of symbols as distance-length pairs.
3. Statistical methods, including Huffman and Arithmetic coding, encodes symbols as variable length codes, based on their probability of occurrence.
4. Transforms are usually combined with other forms of encoding including RLE and Statistical methods but maps a dataset into a form that is better suitable to these forms of encoding.

2.1.1 Basic methods

RLE, simply put, encodes runs of characters using some reserved character and a number indicating the length of the run. In the context of compressing floating-point data such a scheme would focus on encoding runs of zeros and ones at a bit level. A run of characters abbbbcccaddddddd can be encoded as a#4b#3ca#7d.

Move-to-front coding attempts to place frequently occurring symbols from an alphabet of symbols near the front of the alphabet. It encodes each incoming symbol as the number of symbols preceding it in the alphabet. The symbol is then moved to the front of the alphabet with the hope that the symbol will occur frequently in future input. After completion variable-length Huffman codes (discussed shortly) are assigned to each symbol, with shorter codes assigned to more frequent characters. This approach assigns small numbers to frequently occurring symbols and is known to significantly increase the effectiveness of Huffman encoding in the best case scenario. It performs slightly worse than standard Huffman in the worst case [15, ch. 1].

MTF coding is best explained using an example. Consider encoding the following string of characters: woooof. Initially the alphabet is defined as abcdefghijklmnopqrstuvwxyz. After completion the runs can be encoded using variable length codes and RLE.

Iteration	Alphabet	Encoding
woooof	abcdefghijklmnopqrstuvwxyz	22
w**o**ooof	wabcdefghijklmnpqrstuvxyz	22,15
wo**o**oof	owabcdefghijklmnpqrstuvxyz	22,15,0
woo**o**of	owabcdefghijklmnpqrstuvxyz	22,15,0,0
wooo**o**f	owabcdefghijklmnpqrstuvxyz	22,15,0,0,0
woooo**f**	owabcdefghijklmnpqrstuvxyz	22,15,0,0,0,7
Final state	fowabcdeghijklmnpqrstuvxyz	22,15,0,0,0,7

Another basic technique which is particularly relevant for application on numerical data is a predictive compression scheme. Such a compression scheme encodes the difference between each predicted and actual succeeding value. This can be employed quite successfully to compress data generated from time series [6], but does rely on data coherence, in the sense that the difference between consecutive observed values should be small (at a minimum be similar in magnitude).

2.1.2 Lempel-Ziv methods

Lempel-Ziv, commonly referred to as "LZ" or dictionary methods, is a class of algorithms with many variants. It 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 used in the GNU compression utility GZIP. GZIP implements the DEFLATE algorithm which is a variant of the first LZ scheme, LZ-77 [15, ch. 3]. LZ algorithms can furthermore be parallelized, as discussed by Klein et al. [9].

LZ-77 encodes these recurrent phrases as length-distance pairs. If a sequence of characters are found to be the same as a sub-sequence of characters in the search buffer, the distance to the start of that sub-sequence, together with the length of the match are encoded. The size of the search buffer can be up to 32 kiB in some implementations of the algorithm. In LZ-78 the recurring phrase is replaced by a reference to a dictionary item (constructed as the scanner progresses through the text). These dictionaries can then be further encoded using statistical methods [15, ch. 3].

The LZ family of compression techniques perform well on structured data, for example texts in the Canterbury Corpus[3], and Shanmugasundaram et al. [17] shows that methods derived from LZ77 and LZ78 perform reasonably well in this context. Detailed results will be given after a discussion on transforms.

2.1.3 Statistical methods

This class of algorithms normally uses variable-length codes to achieve an optimal (or near optimal) encoding of a 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).

Redundancy (R) is defined as the difference in entropy between the optimal encoding and the current encoding of a data item. It is expressed in the following equation (n is the size of a symbol set and P_i is the probability that a symbol c_i is transmitted from a source)[15, p. 46 - 47]:

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

As the name suggests these techniques use the probability of occurrence of a value to assign shorter codes to frequently occurring values in order to eliminate redundancy. This class includes two widely employed techniques: Huffman and Arithmetic coding. Huffman coding assigns shorter *integral-length* codes, while arithmetic coding assigns *real-length* subintervals of [0,1) to frequently occurring symbols [22][15, ch. 2].

[3]The Canterbury Corpus serves as a benchmark for lossless compression techniques. More information is available at `http://corpus.canterbury.ac.nz/`. Last accessed 23 October 2013.

Huffman coding counts the number of occurrences of a particular symbol, sorts this list in ascending order and merges the rarest pairs of symbols into a single *binary* sub-tree (a tree in which each node can have up to 2 children) and re-inserts the merged sub-tree into the list, while preserving the previously established order. The number of occurrences of each subtree will be the sum of occurrences of all its leaf nodes. Upon termination of this process only a single binary tree is left in the list. Each left-traversal adds a 0 to the encoding of a symbol, while each right-traversal adds a 1. When a leaf-node is reached the constructed encoding is an *unambiguous* code associated with the symbol at the leaf node. Refer to figure 3 for an example. In this case the symbol H will be represented as 11, while some of the least frequent symbols, such as B, will be represented by longer codes (0001 is associated with B).

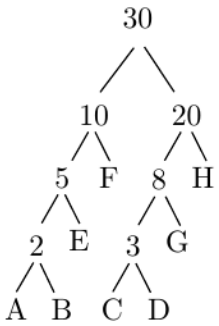


FIGURE 3: Sample of a completed Huffman-coding binary tree [15, p. 70]. The internal nodes represent the a number of times the symbols in the subtree below it occur over the entire dataset. These frequencies have to be determined before the tree can be constructed. This tree is used to assign unambiguous, variable-length codes, depending on the frequency a symbol occurs. More frequent codes like H and F gets assigned shorter codes. A left-traversal along the tree results in adding a zero to the left-most bit of this code, while a right-traversal adds a 1. Therefore 01 will be assigned to F and 101 to G

Instead of assigning variable, *integral*, length codes per symbol as is done by Huffman coding, Arithmetic coding encodes an entire message as a sub-interval of [0,1). As with Huffman a probability distribution has to be determined (or be specified) before the algorithm can be executed. The initial interval [0,1) is reduced by each consecutive symbol and the length of each sub-interval is proportional to the probability of occurrence of the symbol being read. Only the last symbol is treated differently: it gets assigned any value inside the remaining sub-interval. As the interval decreases the number of bits required to encode each consecutive interval increases. However, the average number of bits needed to store each symbol is closer to the entropy encoding of the dataset, since this average number can be a *real* value, unlike that of the per-symbol encoding of Huffman [15, ch. 2].

Both approaches lead to variable-size codes and 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. This can, for example, happen when processing streaming data and all compression has

to be completed on the fly. Arithmetic coding achieves higher compression ratios in both adaptive and non-adaptive cases than 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 [14, 20][15, ch. 2].

2.1.4 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 include, for example, *wavelet* transforms and the *Burrows-Wheeler Transform*.

A wavelet is a small, wave-like, function that is only non-zero over a very small domain and is useful in separating the low frequency components from the high frequency components of, for example, an image. JPEG2000 and DjVu are popular formats that use wavelet transforms. The dataset is generally sampled at increasing frequencies to produce a series of representations that gradually increases in resolution. When combined these representations will reconstruct the original dataset. A sample of the Discrete Cosine Transform of an image (as employed by JPEG2000) is shown in figure 4. Only the transformation coefficients need be stored in order to reconstruct each value in this series. The coefficients within this transformation can then be further compressed using other techniques, for example, Huffman coding and Run-Length Encoding, as is done in JPEG2000. If lossy compression (loss of accuracy which cannot be recovered after decompression) is tolerable, quantization can be used to discard unimportant values, such as the high frequency features of an image [18][15, ch. 5].

Another example of a useful transform is the Burrows-Wheeler transform which groups similar symbols together, by cyclically rotating blocks of symbols, in order to obtain n permutations of the block of length n. The rows in the resulting $n \times n$ matrix are then sorted, after which the runs in the last column can be encoded using RLE. This approach, coupled with move-to-front encoding and Huffman coding is successfully employed in the standard compression utility BZIP2 which will be used later in this investigation [15, ch. 8][16].

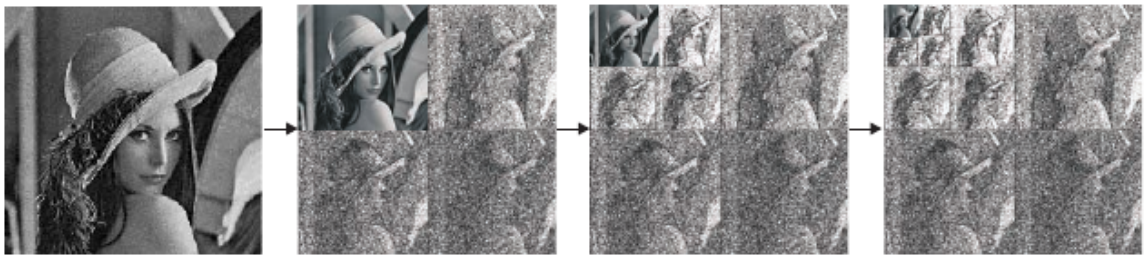


FIGURE 4: Sample output from a Discrete Cosine Transform employed by JPEG2000 [18]. It is easy to see that if lower levels of detail are required, some of the detail weightings stored in the lower portion of the transformation can simply be discarded and better compression ratios can be achieved, sacrificing quality.

Transforms are 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 consistently transferred, while higher resolution samples can be transferred upon request [19].

2.2 Comparison between statistical methods and LZ methods

The analysis in figure 5 is a summary compiled from works by Shanmugasundaram et al. [17]. It shows the average bits needed to encode each symbol in files taken from the Canterbury Corpus. The Canterbury Corpus website furthermore contains results for several common compression utilities including the GNU compression utility GZIP (based on LZ-77), as well as BZIP2, which employs the Burrows-Wheeler transform, RLE and Huffman coding. Extracts from the data provided on the site are shown in figure 6.

	Statistical methods			LZ-77 Derivatives				LZ-78 Derivatives		
RLE	Huff	A Huff	Arith	LZ77	LZSS	LZH	LZB	LZ78	LZW	LZFG
7.93	5.27	5.21	5.15	3.88	3.32	3.22	3.11	4.26	4.90	2.89

FIGURE 5: Table comparing average compression performance (bits per symbol) of statistical methods and the LZ family of methods [17]. RLE performs badly on most of the Canterbury Corpus datasets as can be expected, since structured English texts will not contain many runs of symbols. The LZ77 and LZ78 family performs well in most cases, except for LZW, which fairs poorly on some of the binary files provided in this corpus.

Utility	Bits Per Character	Comp. Speed	Decomp. Speed
BZIP2 (100k blocks)	2.31	4.01	1.32
BZIP2 (600k blocks)	2.23	3.85	1.31
BZIP2 (900k blocks)	2.23	3.98	1.37
GZIP	2.54	1.72	0.81
GZIP (Fast)	2.91	1.10	0.79
GZIP (Best)	2.53	4.19	0.72

FIGURE 6: Table comparing DEFLATE (LZ77-based) and BZIP2 (combination of the Burrows-Wheeler transform, RLE, MTF and Huffman coding). Although BZIP2 is significantly slower it achieves better compression results than GZIP using the Canterbury Corpus.

2.3 Overview of predictive compression

Previous research [13, 3, 6, 11, 12, 4, 8] in predictive compression has yielded good results both in terms of compression ratio and speed. The common line approach is to predict successive values with reasonable precision. 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 exclusive or operator or integer subtraction operation. Machine instructions to count the leading zeros can be found on AMD, newer Intel Processors and CUDA architectures. 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 [11, 8] generalizes the well known parallelogram predictor by extending the parallelogram predictor to arbitrary dimensions. More detail on the parallelogram predictor will be given in the implementation section, but it is worth noting that this approach is particularly useful for compressing large polygon meshes used in computer graphics and engineering design. Other approaches include the use of prediction history (via a simple lookup table construction) as suggested by Burtscher et al. [13, 3, 4]. Burtscher et al. report a throughput of up to 670 MiB/s on a serial CPU implementation of their dictionary-based predictive compressor (*Fpc*) and 1.36 GiB/s as a block-based parallel implementation on a 3 Ghz Intel Xeon quad-core processor [4]. An even simpler scheme by O'Neil et al. [12] reportedly achieved throughputs of up to 75GiB/s for compression and up to 90GiB/s decompression, implemented in CUDA. In this scheme only the difference between successive values is encoded (this will clearly only work if pairs of data points vary very little from one time step to the next). It should be noted that the speeds achieved are on the post-processing of results, already stored in graphics memory and thus do not include the cost of transfering from device to host.

The implementations by [12, 13, 3, 4, 6] target *double* precision floating-point data which is described by the IEEE 754 standard (revised 2008). The design of these encoders rely on the underlying structure of the these 64-bit double precision floating-point numbers (specifically when computing the difference between predicted and actual values). The IEEE 754 standard defines three interchange formats (32-bit, 64-bit and 128-bit). See figures 7 & 8. Each of these have a common construction with the following subfields:

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

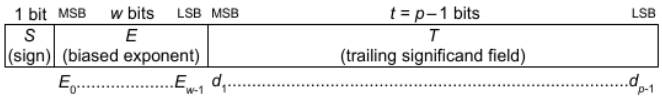


FIGURE 7: IEEE Interchange floating-point format [1]. If two numbers are of the same magnitude and differs very slightly, the biased exponent bits, along with the top bits if the significant will be the same.

The primary scheme proposed in this paper takes its inspiration from the work of O'Neil et al. [12], but will operate on 32-bit IEEE 754 *single* precision floating-point values

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

FIGURE 8: Specifications for the 32-bit and 64-bit interchange formats

(this is due to the fact that the correlation step in the MeerKAT pipeline transforms the data into 32-bit values). The uncompressed data itself is also structured slightly differently to the model used by O'Neil and Burtscher et al. Instead of compressing each neighboring value (or block of values), the proposed compressor will operate over consecutive time slices. As figure 2 shows the data is distributed in time, frequency and correlated pairs. The predictor is based on frequency (per correlation), since there should be some correlation between observations on a specific frequency. If the data is very noisy, little to no compression will be achieved.

As pointed out in previous research [6, 11] using any floating-point operations in the prediction step can cause an irreversible loss of information due to floating-point rounding errors. A simple alternative is proposed by Engelson et al. [6]: floating-point memory is simply treated as integer memory and all operations performed on that memory are integer operations. This approach ensures the predictive scheme achieves lossless compression. In light of this, only schemes that conform to this integer approach will be considered. Engelson et al. give an example of how floating-point numbers can be treated with integer operations. Figure 9 shows 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 exclusive or operation to extract the remaining, seemingly random, residual bits. Therefore if an accurate prediction can be made for consecutive numbers, a reasonable saving in terms of compression ratio can be obtained.

An exclusive or operation is a *binary* operation (taking two operands). It returns false if both operands are the same. If $\oplus$ represents the exclusive or operation then inversion can be achieved as follows: $A \oplus B = C \implies (C \oplus A = B) \wedge (C \oplus B = A)$. Furthermore, the exclusive or operation is commutative, meaning: $A \oplus B = B \oplus A$. The inversion property is obviously critical to achieve decompression since the difference can be exclusive ored with the observation at time $t-1$ to obtain the observation at time t.

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

FIGURE 9: Treating 64-bit IEEE 754 double precision floating-points as integer memory [6]. Due to the fact that the values are all of the same magnitude the most significant bits are equal. After exclusive oring pairs of these values the three bytes of leading zeros can be discarded, and the count stored as a prefix

3 Design and Methodology

3.1 Overview

In light of the research questions and the limited scope of this report, focus will be placed on the usefulness of a predictive data compression step in the context of the MeerKAT project. This will be limited to an investigation of whether a predictive compression algorithm can operate at the desired line rate of 40 GBit/s and whether it provides reasonable compression ratios. This preliminary nature of the investigation precludes the implementation of the algorithm as part of the networking stack employed in the KAT-7 pipeline. Reasonable compression in this context means achieving compression ratios comparable to industry standard tools like GZIP and BZIP2. The possibility of using alternative predictive schemes to trade some throughput for a better compression ratio will also be investigated. Such alternative schemes will include the following (notes on their efficient implementation will be provided in the implementation section):

1. A Lagrange polynomial extrapolation is a gradient-based method that can be used to predict values for functions that can be well-approximated by a n-element polynomial. The method employs only integer multiplication, addition and division.
2. A parallelogram predictor is a simple predictive scheme employing only integer addition and subtraction to make a prediction based on the previous 3 observed values.
3. Moving mean (using only integer addition and division) & median predictors. The median predictor selects the $\left(\frac{m}{2}\right)^{th}$ largest element from m previously observed values, which makes this approach less sensitive to noise, compared to a mean or polynomial extrapolation method.

The key success criteria for this investigation are the following:

1. Throughput in excess of 40 GBit/s is achieved by both the compressor and decompressor.
2. Effective compression ratios, comparable to those achieved by GZIP and BZIP2 are achieved.

3.2 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 the creation of several different implementations of the CPU version (for example SSE, different linear predictors and different parallelization approaches) and simplify swapping one implementation for another. Interactions between components must, thus, be kept to a minimum.

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

The architecture described in figure 10 is used for the benchmarking platform. All the external dependencies depicted in the component model are freely available and are actively maintained. Technical risk in using these libraries is negligible. The benchmark process is relatively simple and its flow is depicted in figure 11. The implementation details will be provided in the next chapter.

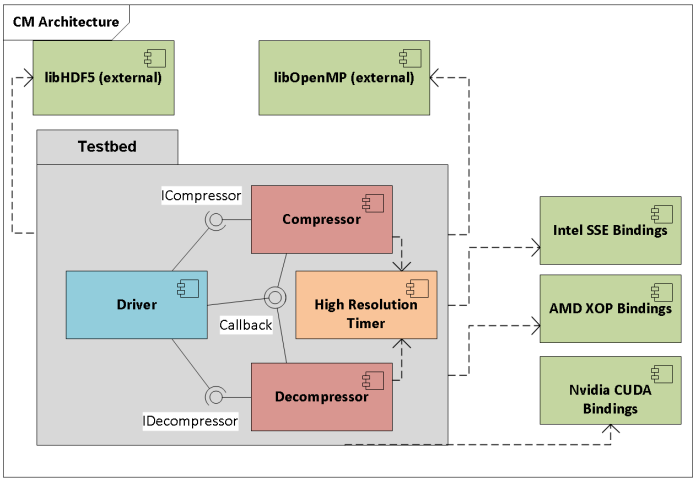


FIGURE 10: Architecture of the loosely coupled benchmarking platform, including external library dependencies. Dependencies are indicated with dashed arrows and interfaces are specified using ball-and-socket notation (for example the ICompressor interface).

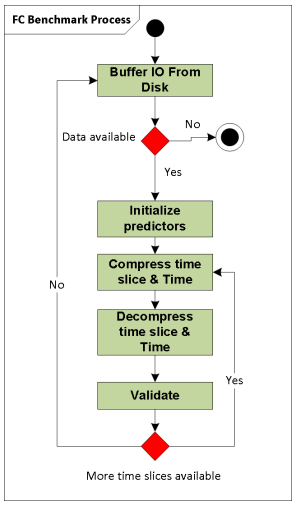


FIGURE 11: Behavioral model of the benchmarking platform. In order to optimize reading from HDF5 files, for testing purposes, data is buffered in blocks of 1 GiB and distibuted in time to the compressor (and by implication the decompressor), as would be the case in a data received over a network. The MeerKAT engineering team plans to distribute the data, first in time and then frequency to a large cluster of commodity servers.

3.3 Packing algorithm

The compacting algorithm being proposed is based on the approach taken by O'Neil et al. [12]. Their approach to compressing floating point values focus on a GPU-based implementation. In their implementation they divide a sequence of 64-bit elements into 32-bit blocks. This is because on GPUs groups of threads (usually 32, depending on architecture) are executed in lockstep. The difference between corresponding elements are then computed, leading-zero-compacted and stored along with the sign-bit and byte count.

Our algorithm is very similar, but operates on correlated 32-bit data and stores the difference between corresponding values at time $t-1$ and t. It is almost *symmetrical* in its compression and decompression stages. In this context symmetry means the algorithm is executed in the opposite direction when performing decompression. The primary difference is that the leading zero count does not have to be computed for each element being decompressed, since it is stored in a compacted prefix stream by the compressor. Refer to figure 12 for more details.

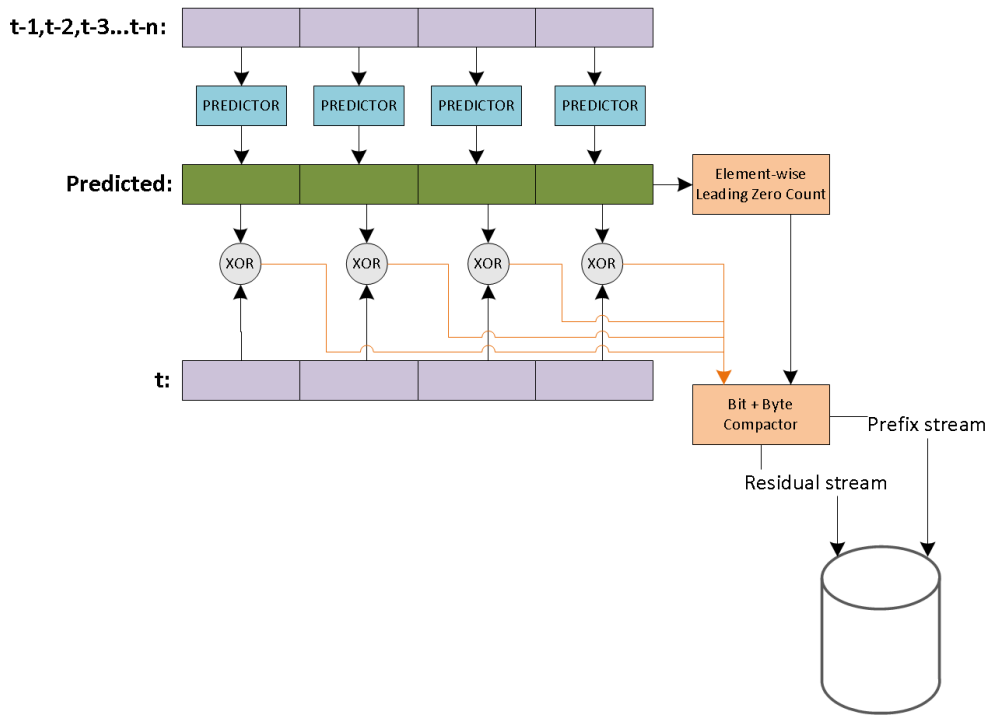


FIGURE 12: Prediction and compaction process proposed for compressing SKA 32-bit IEEE 754 floating-point data, distributed in time. The predictor can use any number of previous time-slices to make its prediction, but this will increase the memory requirement of the predictor. Once a prediction is made the prediction is exclusive ored with its corresponding observation at the time-slice currently being processed. If the differences between predictions and the actual values are small the most significant bits of this difference will be a run of zeros, which is counted by a leading-zero counter. The count is stored as a prefix (compacted at bit level). The remaining bits, or residuals can be compacted at either bit or byte level.

The algorithm structure in figure 12 is used because it allows for easy swapping between different predictors. The predictor in this case can be any n-element predictor. As mentioned earlier the predicted value is exclusive ored with the actual value to obtain a residual which can be compacted at either bit or byte level. This compaction simply removes the leading zeros from each leading zero count and stores each residual as a fixed-length prefix count. The number of bits needed to store n leading zeros can be calculated as $\lfloor \log_2 n \rfloor + 1$ bits where $n \in \mathbb{N}_{>0}$. It remains to find the residual compaction scheme that gives the best 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 the leading zero counts. The opposite is true for byte-packing schemes. Since the prefixes are always short in our case a bit-packing scheme is used to store these fixed-length codes. Basic pseudo code is given in the implementation section.

Suppose the following two (converted) float values have to be compacted (the left-most byte is the most significant byte):

```
#1: 00000000 01100010 01101000 11001001
#2: 00000000 00000000 11110001 11110010
```

In this scenario the compaction of the residuals is done at byte level and the prefixes done at bit level (assume three bits are needed to store up to four bytes of leading zeros):

```
RESIDUALS:
01100010 01101000 11001001 11110001
11110010
PREFIXES (VALUES ARE PADDED UP TO ENSURE ONLY COMPLETE BYTES ARE STORED):
00101000
```

The first value only has one byte of leading zeros, and its associated prefix is therefore $1_{10} = 001_2$. Similarly the second value has two bytes of leading zeros, its prefix therefore is $2_{10} = 010_2$. The remaining two bits are simply padding, since the atomic unit of storage is one byte (eight bits).

3.4 Parallelizing compression and decompression steps

This investigation considers three approaches to achieve parallelization on a CPU. This list is motivated by a trade-off in compression ratio and overheads, as well as the block-based approaches taken in previous research, specifically that of Klein, Burtscher & O'Neil et al. [4, 12, 9]

3.4.1 Element-wise parallel execution

Even though the residuals are of variable length it is possible to pack them in parallel. However, there is an additional overhead to *both* the packing and unpacking process,

since this approach requires a *prefix sum* to be computed for the array of element-wise leading zero counts.

A prefix sum, *prescan* can be defined on any binary associative operator, $\oplus$ over a sequence with I as its identity. If $A = [a_0, a_1, \ldots, a_{n-1}]$ is a sequence then the scan is defined as $scan(A) := [I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus ... \oplus a_{n-2})]$.

In the context of the packing scheme proposed here, the prefix scan is defined over the normal integer addition operator. An example of this would be $scan[2, 1, 3, 4] = [0, 2, 3, 6]$ under normal integer addition. The used-bit counts are saved to an array, A, after which the scan operation is computed on the array in order to obtain an, in-place, element-wise accumulation. Therefore the algorithm has a *constant* memory overhead (uses little extra memory). Each index $A[i]$ will therefore store the starting position (in bits) of the residual for the i^{th} element [2]. This can be illustrated by an example of compacting four differences in parallel:

```
#1: 00000000 01100010 01101000 11001001
#2: 00000000 00000000 11110001 11110010
#3: 00000000 00000010 11010111 11111110
#4: 00000000 00000000 00000000 00101101

The leading zeros are counted, subtracted
from 32 and stored:
C = [23,16,18,6]
A scan operation is completed:
C = scan(C)
C will now be equal to [0,23,39,57]

If the residual stream is compacted at bit
level and indexed from bit 0, each element
can be packed by a different thread:
11000100 11010001 10010011 11100011
^0                      ^23
11100101 01101011 11111111 01011010
       ^39                  ^57
The last zero is simply padding to fill
the byte.
```

This scan operation can be computed in parallel as suggested by Ladner, Cole and Blelloch et al. [10, 5, 2]. Additionally a work-efficient CUDA version of the algorithm discussed in GPU Gems 3 [7] is based on Blelloch's approach. In this version the algorithm is optimized to avoid *bank conflicts* (see the section on GPU architecture) and has been shown to be up to 6 times faster than a sequential CPU implementation for large arrays.

Ladner et al. suggests an algorithm that has a time complexity of $O(\log n / \log \log n)$ to compute the prefix sum over n elements, but requires $n \log \log n / \log n$ processors. Cole et al. suggests an improvement to this approach, achieving a time complexity of

$O(\log n / \log_2 n)$, but imposes the restriction that the n elements being processed are each $O(\log n)$ bits in length. Blelloch's approach has a complexity of $O(n/p + \log_2 p)$, where p is the number of processors in a system, but only requires the length of the input array to be a power of 2, and is therefore more flexible than the other approaches. Blelloch's parallel algorithm will be described in the implementation section.

After the indices have been accumulated using the prefix sum algorithm it is easy to see how different threads can pack residuals at the correct positions. Both a bit and byte packing scheme will, however, require *atomic* operations to ensure against the *race condition* that arises when multiple threads write to the same memory location simultaneously. This memory-level-synchronization mechanism adds additional overhead in terms of wasted machine clock cycles.

3.4.2 Block-based approach

Incoming data can also be separated into blocks, after which multiple of these blocks can be processed 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. However, this approach may have a detrimental effect on the compression ratio since the length of the residual array of each block has to be stored.

3.4.3 Vector intrinsics augmentation

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 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. A combination of the Intel SSE and AMD XOP instruction sets, will however, provide enough support to execute most of the algorithm using vector intrinsics. The XOP instruction set extends the SSE instruction set by adding these operations as one of its primary features. Adding these instructions will, however, make the implementation dependent on the architecture of the underlying machine, and therefore less portable.

If this approach is successful the vectorized code should be extended to use the latest Intel AVX 2 instruction set in which Intel introduces element-wise shift operations and offset-loading operations as part of future work. The AVX family of instructions can compute up to 8 SIMD operations in parallel per core. All these vectorized instruction sets make use of a very limited number of 128- and 256-bit registers (SSE and AVX, respectively). It remains to be investigated whether it is worthwhile to implement the proposed packing algorithm using SSE and XOP instructions.

3.5 Porting the implementation to CUDA

General Purpose programming using Graphics Processing Units (GPGPU) provides a massively multi-core environment, with hundreds of cores, for performing SIMD-styled processing on large numbers of elements. GPU architectures differs significantly from those of multi-core CPUs: memory and cache-memory layout is completely different (each group of cores has dedicated shared, cache and texture memory), direct manipulation of cache and shared memory is possible, individual cores on the GPU are less powerful than normal CPU cores (each core does not have its own dedicated floating point processor) and these cores are furthermore grouped into *warps* of threads, where each Streaming Multiprocessor (SM) consists of multiple warps. Figure 13 provides a detailed overview of the microchip die structure of a Nvidia Fermi generation GPU.

Programming for GPU architectures brings about a host of associated challenges. For starters, each generation of GPUs have widely differing architectures, each making many architecture specific optimizations. Each generation, for instance, will have both a different ratio and configuration of arithmetic cores to floating-point and special operation cores per SM. Additionally, each generation has widely differing memory controller layouts. These may determine how to schedule the execution of warps of threads, when the warps are swapped in/out, along with many other properties. Furthermore, atomic groups of cores (half-warp of threads on most modern architectures) are executed in lockstep.

A basic CUDA implementation will be provided 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. Adequate use of shared memory will be required and the implementation will take precautions to *coalesce* memory accesses and to avoid *bank conflicts* when possible.

Coalesced memory accesses are made if a continuous chunk of memory is accessed by a half-warp of threads at the same time. Normally these accesses have to be made in a uniform pattern that is aligned perfectly with the structure of global memory. Individual memory calls to global graphics memory is a very costly operation (and therefore random accesses have to be avoided). Coalesced accesses are made simultaneously by the memory controller. Such accesses retrieve / write memory in larger chunks, which counters any latencies associated with such an access.

As for bank conflicts, in GPU architectures (at the very least with Nvidia cards) *shared* memory is divided up into banks. When multiple threads within the same half-warp try to access the same bank of memory, the memory accesses will be serialized. The general approach to avoid such a situation is to add padding between blocks of shared memory.

More details will be provided in the implementation section, but it is clear that a combination of the block-based and element-wise parallel approaches, taken with the CPU version, will be required to make adequate use of the GPU architecture. This proposed 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 no unnecessary overheads due to costly card-wide synchronization steps are

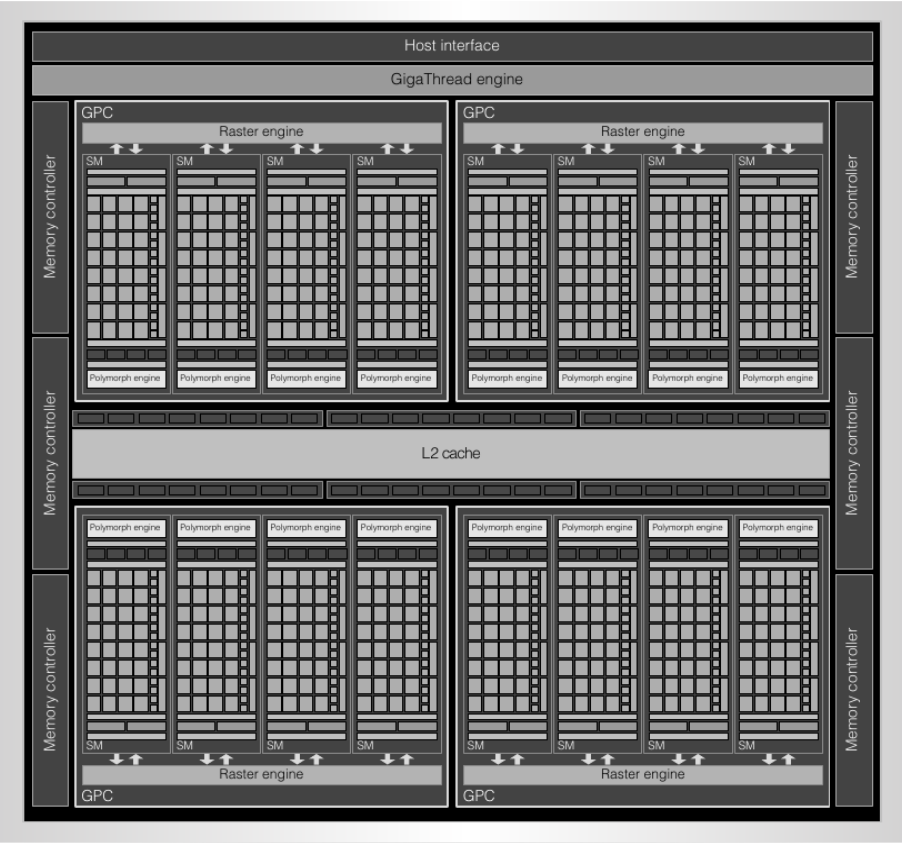


FIGURE 13: Architecture of the Fermi family of Nvidia GPUs [23]. The GPU consists of multiple Streaming Multiprocessors (SMs), which in turn is divided into groups of cores (called warps). Each of these warps will have dedicated floating point arithmetic units, along with an allocation of fast special operations units, containing many instructions that are commonly used for graphics processing (for instance a dedicated square root instruction is provided, because it is used for vector normalization). Each half-warp of threads are executed in lockstep, all performing the same instruction, and each SM in turn operates in parallel.

introduced. Instead, all atomic operations and synchronization steps will be performed per-SM only.

3.6 Test data

The correlated data is provided in the form of HDF5 files. Each of these files stores a three dimensional array of complex pairs. The first dimension is time. The second is signal frequency (as 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). The size of the last dimension is specified as twice the number of permutations of $n + 1$ (where n is the number of antennae in the array). The headers of MeerKAT 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.

It should be noted that the headers cannot be compressed with the algorithm being proposed and an LZ variant / entropy encoder may be better tailored for this task. The headers vary between 13-14% of the total HDF5 file size. However, the data is only written to these formats after it is received by the processing cluster (see figure 2). Therefore, if the algorithm being proposed is integrated into the front end of the pipeline, as shown in figure 2 (as part of future work) it may still provide a significant saving in network I/O.

3.7 Structure and scope of the benchmarks

Some of these HDF5 files measure more than 30GiB in size, each containing many time-stamps (some span over 4000 frequencies). It is reasonable to 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 network stack currently employed by the KAT-7 prototype. Due to the limited scope of this project focus will be on analyzing the performance of the algorithms themselves and not measuring any disk I/O, except when comparing against other compression utilities for the sake of fairness.

This report will focus on the following analysis:

1. A comparison between a CPU implementation of the parallel prefix sum-based algorithm and the blocked approach will be made.
2. For whichever method is determined most fit in step 1, a detailed breakdown of how the algorithm performs on different numbers of cores will be provided.
3. The effectiveness of a parallelogram predictor, a Lagrange extrapolation predictor [6] and a moving mean and median scheme will be compared. Each of these methods will be described in greater detail in the implementation chapter.
4. The feasibility of implementation using the Intel SSE and AMD XOP vectorized instruction sets will be investigated.
5. Benchmarks will be undertaken of the algorithm, against the standard programs used for comparison: GZIP, BZIP2 and ZIP. In this case the disk I/O will be included for the sake of fairness.
6. The feasibility of a CUDA implementation, as outlined earlier, will be explored.
7. Finally, results from concurrent research being conducted into entropy encoding and run-length encoding will be compared to those achieved in this investigation. This comparison will include a breakdown of performance in terms of both throughput and compression ratios.

4 Implementation

4.1 Overview

In this section we discuss the technical details surrounding our implementation of a parallel predictive compression algorithm on both multicore CPUs and GPUs. First the basic packing algorithm is discussed, along with technical details on alternative predictive schemes, then details on achieving parallelization is provided and finally the tradeoffs of using each of the alternative predictive schemes and parallelization are discussed.

4.2 Basic algorithm

In order to give a more detailed description to the algorithm illustrated in figure 12 in the previous section, the pseudo code for the compaction process is given in figure 14. After a linear predictive step is performed and the predicted value exclusive ored with the actual value the leading zeros are counted. The number of bytes worth of leading zeros ($\frac{\text{lzc}}{8}$) is computed and stored as fixed-length prefixes, each three bits long. Whenever division is performed it refers to performing integer division. The starting position of each prefix in an array of 32-bit values are therefore calculated as: $\frac{\text{index}\times 3}{32}$. The bit position in each of these prefix stores are the remainder of this division operation modulo 32. Since 3 bytes does not divide 32, some of the prefixes will overflow the store on which its first bits are stored. It is therefore necessary to save the remaining bits into the next index in the prefix array. All bit-shifting operations performed, are *logical* left and right shifts. These shifts do not preserve the sign bit, as opposed to arithmetic shifting operations. When a right shift is performed a zero is added as the most-significant bit. The process is very similar for residual array, except that, because the residuals varies in length the accumulated number of bits used up to the current iteration have to be tracked.

The decompression algorithm is remarkably similar, except that the leading zero count is not computed, but is instead retrieved from the prefix array. Only the shift operations are inverted to retrieve the bits (that may be split over two consecutive elements) in both the prefix and residual arrays and the prediction step is moved towards the end of the routine, where the predicted value is exclusive ored with the residual to restore the original, uncompressed, value.

4.3 Alternative linear prediction schemes

The most basic prediction scheme simply encodes the difference between observations at time t and $t+1$. This difference is given as $O[t]$ xor $O[t-1]$. The downside of this basic predictor is that it is particularly sensitive to the structure of the underlying data. It will only work if *most* consecutive values vary slowly over time. Using an n-element predictor instead may help mitigate the effects of noise. However, as pointed out earlier: any operations over these n-elements have to be integer operations, and not floating point

```
SUBROUTINE Compact(FLOAT[] data, UNSIGNED INT32 numElements,
                   out UNSIGNED INT32[] residuals,
                   out UNSIGNED INT32[] prefixes)
  //ASSUMPTION: 3 BITS NEEDED TO STORE UP TO 4 BYTES OF LEADING ZEROS
  //ASSUMPTION: COMPACTION IS DONE INTO 32 BIT INTEGER ARRAY
BEGIN
  UNSIGNED INT32 bitPosInResidualArray = 0
  FOR index = 0 TO numElements - 1 DO
    /*
    This prediction step requires up to "n" arrays of the
    same size of "data".
    */
    UNSIGNED INT32 predictedValue = linearPredict(index)
    UNSIGNED INT32 difference = predictedValue XOR data[index]
    UNSIGNED INT8 leadingZeroCount = clz(difference)
    //save the prefix:
    prefixes[index*3 INT_DIV 32] = SHIFT (leadingZeroCount INT_DIV 8) \\
        to the index*3 MODULO 32 th bit
    IF there are remaining bits that couldn't be fitted THEN
        prefixes[index*3 INT_DIV 32 + 1] = SHIFT remaining bits that \\
        couldn't be fitted in the previous operation to the first \\
        bit of the next byte
    END IF
    //save the residual:
    residuals[bitPosInResidualArray INT_DIV 32] = SHIFT difference to \\
        the bitPosInResidualArray MODULO 32 th bit
    IF there are remaining bits that couldn't be fitted THEN
        residuals[bitPosInResidualArray INT_DIV 32 + 1] = SHIFT \\
          remaining bits that couldn't be fitted in the previous \\
          operation to the first bit of the next byte
    END IF
    //increment position in the residual array
    bitPosInResidualArray += leadingZeroCount
  END FOR
END SUBROUTINE
```

FIGURE 14: Basic pseudo code for compacting residuals at byte-level and prefixes at bit-level in serial

operations, due to the rounding errors they may introduce. All floating-point memory is therefore cast to integer-typed memory before values are retrieved.

These predictions, however, still have to be fast and preferably have an average linear computational complexity, $\theta(n)$. The predictor will only store the previous n observations in a fixed-length queue. This implies that the memory overhead for the predictor is $n \times m$ where m is the total number of elements being processed at time t.

4.3.1 Lorenzo predictor

The n-dimensional Lorenzo predictor is a generalization of the two dimensional parallelogram predictor (see figure 15). It can be used to predict scalar functions that are polynomials of degree $n-1$ and has proven useful in scalar field compression [8]. In each of the 3 cases below only simple integer additions and subtractions are needed: the sum of the values in red are subtracted from the sum of the values in green to derive the value in blue. In the two-dimensional case this addition and subtraction is known as the *parallelogram rule*. The predicted value is computed from previous observations as as $P = O[t-1] + O[t-2] - O[t-3]$. The prediction operation remains simple and should not have a significant impact on the overall throughput. However, this approach will still be quite sensitive to noise.

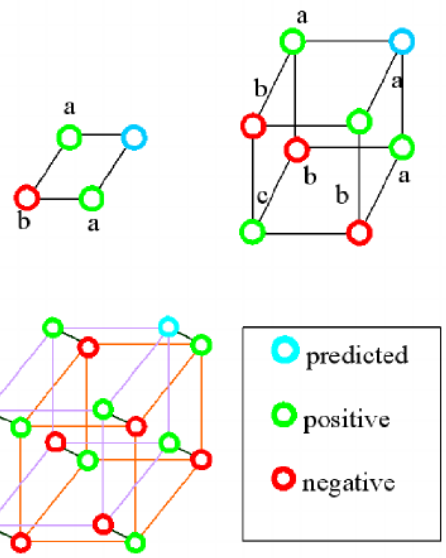


FIGURE 15: Lorenzo predictor for 2D (upper left), 3D (upper right) and 4D (lower left) data [8]. The sum of red values are subtracted from the sum of green values, to derive the value indicated in blue.

4.3.2 Moving average and moving mean

An integer-based moving mean is simple to implement and involves only a summation and an integer division, and therefore clearly has a linear computational complexity. However, the mean of a sample is sensitive to outliers, and therefore will not work well for noisy data. In contrast, using a median-based scheme, where the predicted value is the center value of a *sorted* list of observations may fair better. Unfortunately, a naive implementation of such a median-based scheme will be slow due to the overheads of sorting. Wirth [21] suggests an algorithm to select the median using a basic *pivoting* scheme in $\theta(n)$ time. A pivot divides a dataset into two groups. The elements in the group to the right of the pivot are larger than the pivot and the elements in the group to

the left of the pivot are smaller (or equal) to the pivot. The two groups do *not* necessarily have to be ordered. The median is chosen through the method indicated in figure 16 that selects the *k-th largest value* [21].

```
FUNCTION INT32 pivotMedian(INT32[] data, int n) {
    INT32 i, j, l, m, k = n/2 //k is the kth largest element
    INT32 x, s

    l=0 //lower bound
    m=n-1 //upper bound
    //swap elements in group L and M till the indexers cross
    WHILE l<m DO
        x=data[k]
        i=l
        j=m
        REPEAT
            //find 1 element in each sub-array that has to be swapped
            //to the other sub-array:
            WHILE data[i]<x DO i++
            WHILE x<data[j] DO j--
            IF i<=j THEN
                    //swap:
                s=data[i]
                data[i]=data[j]
                data[j]=s
                i++
                j--
            END IF
        WHILE i<=j
        if(j<k) l=i
        if(k<i) m=j
    END WHILE
    RETURN data[k] //mid-point in the array
}
```

FIGURE 16: Pseudo code for selecting the k^{th} largest element of a sequence.

4.3.3 Lagrange prediction

Lagrange extrapolation prediction is a gradient-based method that can be used to predict a value of a polynomial at $t+1$, given the values at $t \ldots t-n$. The method is commonly used for interpolation between known points on a curve, defined by some polynomial. Good extrapolations are only possible for *smooth* data.

If a function $f : [1 \ldots n] \rightarrow \mathbb{R}$ is evaluated over a time interval $[1 \ldots n]$ and its results stored as a sequence of values $a_1 \ldots a_n$ where $a_i = f(i)$, then such a sequence is called *smooth of order* m if for all $j > m$, a_j can be approximated with reasonable accuracy by extrapolating from the previous m values. Such smooth sequences normally arise as solutions of ordinary differential equations (ODEs) in simulation models [6].

The model implemented in this report uses *fixed-step* extrapolation (the time between consecutive observations is assumed to be of constant duration), but the method works just as well if these differences vary in length, as long as the underlying sequence can be approximated by some polynomial [6]. The method builds from *Lagrange's Rule*. It states, that for a given function, f, implicitly defined over the previous m observed values, given some extrapolation points, indexed by $x \in I = \{i \in \mathbb{N} | i \geq 0 \wedge i \leq m\}$ there exists some polynomial, φ_m such that $(\forall x \in I)\varphi_m(x) = f(x)$. The polynomial is defined as $\varphi_m(x^*) = \sum_{x \in I} L_x(x^*) f(x)$ where each coefficient, $L_x(x^*)$, can be computed by the equation below. Note that $x^* = m + 1$, the index of the unknown value of f, after the $1 \ldots m$ previously observed values.

$$L_x(x^*) = \prod_{k \in I, k \neq x} \left(\frac{x^* - k}{x - k} \right) \tag{4}$$

4.4 Algorithm parallelization and its implications for GPGPU architectures

As pointed out in the design section, the parallelization of the basic algorithm can be done by dividing incoming data up into fixed size chunks after which multiple blocks can be packed in parallel. However, this approach is detrimental to the compression ratio, since the size of each block's residual stream has to be stored. This effect will be worse if there are a large number of blocks. On the other hand the approach is relatively straightforward to implement and does not modify the structure of the basic algorithm at all.

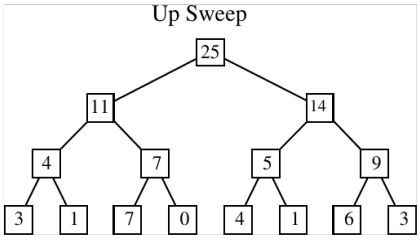
The alternative is to convert the basic algorithm into an element-wise parallel algorithm. As mentioned in the design chapter this requires computing a prefix sum in order to obtain the starting position of each residual in the residual stream. The serial prefix scan is a trivial case of a dynamic programming problem, but a parallelization approach is suggested by Blelloch [2]. Blelloch's approach has also been employed for GPU-based prefix sum calculation [7]. This parallel algorithm requires the length of the input array be a power of 2, but if it is not the case the input array is simply padded.

There are three primary steps to the algorithm, with a synchronization point between each. First an *up-sweep* operation is completed, then the last element is set to 0, and finally a *down-sweep* operation is carried out. Both the *up-sweep* and *down-sweep* operations can be represented as a *complete* binary tree of $\lceil \log_2 n \rceil$ fully-filled levels. The *up-sweep* and *down-sweep* operations (using integer addition) over a short sequence of integers are shown in figure 17.

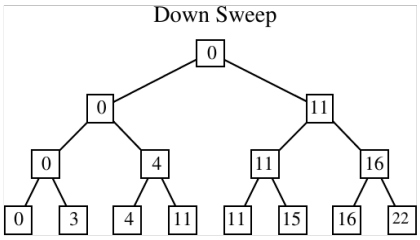
The up-sweep operation divides the array into pairs of elements and assigns each pair to a processor. It computes partial sums for each pair and recursively reduces pairs of partial sums. Upon completion of the up-sweep operation the array consists of many partial sums, which the down-sweep step then combine into a complete prefix sum, by recursively swapping and accumulating from the root down. It has been shown [2] that after completing a down-sweep, each node of the tree contains the sum of all preceding leaf nodes.

	Step	Array in Memory							
	0	[3	1	7	0	4	1	6	3]
up	1	[3	4	7	7	4	5	6	9]
	2	[3	4	7	11	4	5	6	14]
	3	[3	4	7	11	4	5	6	25]
clear	4	[3	4	7	11	4	5	6	0]
down	5	[3	4	7	0	4	5	6	11]
	6	[3	0	7	4	4	11	6	16]
	7	[0	3	4	11	11	15	16	22]

(A) Steps of the scan operation being executed in order on the sequence [3 1 7 0 4 1 6 3]




```
FOR d from 0 TO log2(length(A))-1 DO
  FOR i from 0 TO length(A) - 1 IN STEPS OF POW(2,d+1) IN PARALLLEL DO
    A[i + pow(2,d+1) - 1] += A[i + pow(2,d) - 1]
  END FOR
END FOR
```




```
A[length(A)-1] = 0
FOR d from log2(length(A))-1 DOWNTO 0 DO
  FOR i from 0 TO length(A) - 1 IN STEPS OF POW(2,d+1) IN PARALLLEL DO
    INT32 temp = A[i + pow(2,d) - 1]
    A[i + pow(2,d) - 1] = A[i + pow(2,d+1) - 1] //set left child
    A[i + pow(2,d+1) - 1] += temp //set right child
  END FOR
END FOR
//EFFICIENCY NOTE: Powers of 2 are efficiently implemented as:
POW(2,n) = (2 << n-1) //where "<<" is the logical left-shift operator
//and n is strictly greater than 0
```

(B) Tree representations with respective algorithms to compute each level in parallel (for both up- and down-sweeps)

FIGURE 17: Parrallel scan algorithm example and pseudo code [2]

The entire packing algorithm can now be converted to three steps, that can each be computed in parallel:

1. Perform a leading zero count and store the prefixes as discussed in the basic pseudo code, but store the leading zero count to an array, A. The prefixes are fixed in length, so placing them into the prefix stream in parallel does not pose a problem.
2. Compute $B = scan(A)$ in parallel
3. Each element, $B[i]$ is now the starting position (in bits) of the residual of element $D[i]$. This means the variable-length residuals can be placed into the residual stream in parallel.

The algorithm shown in figure 17 assumes execution takes place in a *shared-memory* architecture, where each processor has access to the memory of the other processors and that synchronization can take place across multiple processors. In CUDA the last condition is not easily met: the global synchronization required by the parallel prefix sum scan can only be achieved through a global memory barrier. However, this will cause a high number of context switches for every SM on the GPU (especially if the number of blocks are much larger than the number of available SMs). The alternative approach is to let each SM compute the prefix sum of a sub-sequence of numbers and store its last value of the accumulation as S_i. Each block total, S_i, is then distributed to the $i+1 \ldots n$ blocks of sub-sequence prefix sums in parallel [7].

This distribution step can be avoided by splitting the dataset into blocks, where each block is a multiple of the warp size in length. This effectively means that the data can be spread throughout the entire GPU in order to achieve a high *occupancy* of the available SMs, and each SM can then pack its block in parallel, by computing a short prefix sum only for leading zero counts of the block it is processing.

If this approach is taken in conjunction with the modifications described in GPU Gems 3 [7], a fast, work-efficient, parallel prefix sum scan can be implemented, that both minimizes overheads, and includes optimizations through the use of shared memory on the GPU. These modifications include adding padding to each shared memory index according to the *degree* of the bank conflict at the associated depth of the indexing tree shown in figure 17b, and letting each thread compute two values at a time. By doubling the strides between each successive level in the tree, the modifications ensures that double the number of threads can be executed on that level without bank conflicts. See figure 18 for an illustration of this solution.

The next complication for an element-wise packing implementation is the matter of atomic operations. This issue arises due to sharing an atomic memory location between multiple threads that require simultaneous access to that location. The following example illustrates a race condition between 2 threads, packing residuals into a 32-bit residual array: thread 0 tries to pack 00000000 00000000 00011101 01100111 into residual store $R[0]$ while thread 2 tries to split 00000000 01001010 01110001 01111111 into the same $R[0]$ and its successor, $R[1]$. Since $R[0]$ can only be written to by one thread at a time, either thread 1 will overwrite what has been written by thread 0, or the converse. This clearly causes irreversible loss of data. One way to avoid this is to lock the memory location for

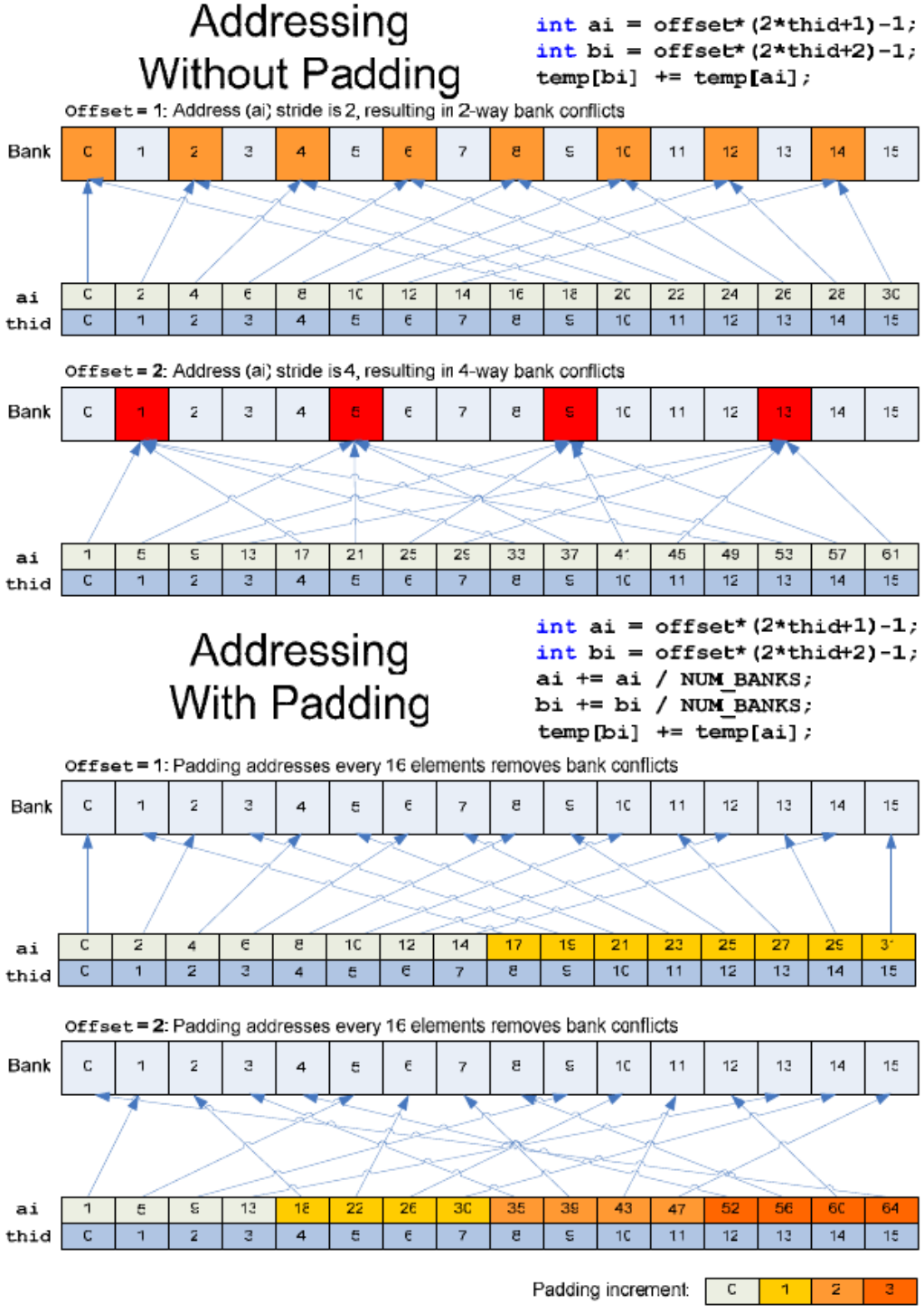


FIGURE 18: Applying padding to avoid a high degree of bank conflicts during the scan operation. The simple addressing scheme at the top results in bank conflicts, while the illustration at the bottom shows how no bank conflicts are encountered when shared memory addressing is padded [7].

reading and writing, while another thread is reading and writing to the location. This potentially serializes the memory access and can destroy parallelism.

One way to avoid two atomic operations per residual is to use an array of bytes as the atomic unit of the residual stream. However, since each element is 4 bytes in length, this may involve anything from 0 up to 4 global memory write operations, along with the necessary control logic, and this will result in *branch-divergence*. In the context of general GPU architecture this has potentially devastating consequences for simultaneous execution. As mentioned earlier each SM consists of many arithmetic units that are executed in atomic groups (each performing the same basic instruction). When some threads are required to perform one task, while others have to perform another, the SM is forced to let all the threads perform both tasks in sequence, each time discarding values from the threads that are not supposed to execute through a masking operation. This means that double the number of clock cycles are needed to perform a branch. This problem becomes worse if a 3- or 4-way branch is necessary. This branch-divergence, coupled with multiple costly memory accesses may prove worse than using an atomic operation.

Furthermore, it is possible to remove the atomic operations from the prefix compaction operation by storing only prefixes with a length that divides into the length of the closest atomic unit (byte, short or integer). If the compacting operation is constrained to packing up to 3 leading zero bytes of residuals, only 2 bits are needed to store this count per element. Every 16^{th} thread then has to store 16 leading zero counts simultaneously into a single 32-bit prefix store. In order to avoid multiple write operations to global memory, the 16 prefixes can be stored in a register before a call to global memory is made. This is accomplished through shifting and bit-wise oring each prefix into a 32-bit integer variable before writing that to global memory. The following provide an example of this:

```
C = [2,1,3,1,...] //10,01,11,01 in binary
UINT32 sixteenPrefixes = C[0]|(C[1] << 2)|(C[2] << 4)|(C[3] << 6)|...
This shifting and oring will result in the following computation:
                  (MSB)                          (LSB)
sixteenPrefixes = 00000000 00000000 00000000 00000000
                | 00000000 00000000 00000000 00000010
                | 00000000 00000000 00000000 00000100
                | 00000000 00000000 00000000 00110000
                | 00000000 00000000 00000000 01000000
                |                ...
At the end of the operation sixteenPrefixes will have the following value,
before being written to global address space:
........ ........ ........ 01110110
```

4.5 Feasibility analysis

4.5.1 Compression feasibility

As mentioned previously the packing of residuals can be done at either bit or byte level. When packing at a bit level $\lfloor \log_2 n \rfloor + 1$ bits are needed to represent n leading zeros. When packing up to 31 leading zeros 5 bits are needed, whereas 3 bytes of leading zeros can be packed using only 2 bits for each prefix, as explained earlier. Which scheme works better is entirely dependent on the properties of the data and how accurate the predictions are. From experimentation, we found packing at byte level (up to 3 bytes of leading zeros) to be the most efficient scheme. It gives compression ratios approximately 1% better than the closest bit-level packing scheme.

The compression ratios achieved when encoding the difference between successive observations by means of an exclusive or operation are not very high. An average compression ratio of 91% is observed. This shows that although the compression scheme is rather basic there is some coherence between successive observations.

The ratios mentioned above includes storing the following information:

- All observations at time $t = 0$ are stored in their uncompressed form, along with the total number of values per time step. Both the compressor and decompressor are initialized with this size and the values at this initial time step. When the number of observations changes (say due to a larger number of frequencies being sampled), both the compressor and decompressor have to be re-initialized.
- For the block-based approach the number of blocks must be stored if this value is not fixed.
- Since the residual arrays vary in length, the length of the residual array is stored before storing each block of residuals, per block if necessary.
- The prefix array is stored without a length field, due to the fact that the prefix array size can be computed from the number of elements.

The ratios, however, do not reflect additional header meta-information as found in the sample HDF5 files, and are solely based on the number of 32-bit floating-point values each HDF5 file contains.

4.5.2 Parallelization approaches and optimizations on the CPU

For the initial implementation we considered parallelizing the algorithm at the element level, using prefix sums as described in the previous section. Unfortunately, even when using up to 16 CPU threads through OpenMP the overheads of computing a prefix sum over all the elements turned out to be too costly on a CPU-based architecture. Although the implementation scale roughly linear in throughput per core, it only manages to achieve compression throughputs of around 400 MiB/s and decompression throughputs of around 600 MiB/s on a 16 core test. This is significantly slower than the required throughput of 5 GiB/s. It is duly noted that the throughput rates achieved in the element-wise

parallelization include several optimizations such as the removal of branches (for example branchless implementations of minimums and maximums are employed [4] for the logical bit shifting operations shown in the pseudo code), as well as the removal of some of the atomic operations, as discussed in the previous sections.

The failure of this approach in a CPU environment, however, does not necessarily preclude a similar approach being taken on a GPU. It has been shown [7] that a prefix sum can be computed up to six times faster on a GPU for large arrays of up to 16.7 million elements. This result, coupled with the fact that elements can be packed in a massively parallel environment, is reason enough to investigate a CUDA implementation. More details on this can be found later in this chapter.

The alternative approach to this element-wise parallel scheme, is to separate the data into blocks. There is one downside to this approach: it will degrade the compression ratio by a small amount depending on the number of blocks chosen (since the length of the residual array has to be stored). This approach proved more feasible than the element-wise parallel packing: the compressor achieves rates up to 1.98 GiB/s and the decompressor up to 3.66 GiB/s using 16 cores. Further optimizations included using a leading zero count assembler instruction [5] instead of loops to obtain the count. This optimization increases the throughput by up to 200 MiB/s on a quad-core processor. When the implementation is run for large files on a 32 core AMD Opteron platform the desired throughput of 5 GiB/s is achieved. A detailed breakdown is given in the results section.

4.5.3 Alternative prediction schemes

The basic difference scheme discussed in section 4.2 will not work well in the case where the sign bit of 2 consecutive observations differs (as previously mentioned the sign bit is the most significant bit of an IEEE 754 floating point representation, implying that there will be no leading zeros to encode). A solution to this is to dedicate an extra bit per prefix in order to store the sign value. From experimentation it was determined that this trade-off between storing an extra bit for every prefix and the number of signed differences is not worth encoding: a saving of less than 0.01% is made for a number of the sample datasets.

The next simple modification to the scheme is to test whether using normal integer subtraction fairs better than an exclusive or operation. Burtscher et al. point out that integer subtraction fairs much worse, both in terms of compression ratio, and throughput for their dictionary-based predictive compressor operating on 64-bit floating points values [3]. This accords with our findings on 32-bit KAT-7 data considered in this report. A

[4] Sean E. Anderson provides several bit-twiddled implementations of minimums, maximums and binary logarithms on his website: `graphics.stanford.edu/~seander/bithacks.html`. Last accessed: 3 October 2013

[5] `LZCNT rO,rI` is a machine instruction found on most AMD processors (since the introduction of their XOP vector instruction set), as well as newer Intel processors. Nvidia GPUs have similar support for leading zero counts.

loss of between 2% to 3% in compression ratio is observed. It is therefore better to adopt a simple exclusive or operation.

The integer mean predictor fairs poorly compared to the simple difference scheme by degrading the compression ratio by, from 3% to almost 7% (thereby inflating the data). The effect worsens when the number of prediction steps is increased. As pointed out earlier such a mean-based scheme is prone to outliers, which can be expected in noisy data. The median scheme was expected to fair better, and an implementation of Wirth's pivot-based algorithm confirmed this hypothesis. Similar or slightly smaller (roughly 1%) compression ratios to that achieved with the normal difference-based scheme are observed, depending on the number of prediction steps being used. However, although the pivoting algorithm has an average linear computational complexity, it does have a number of branches. This imposes a severe penalty on throughput: over 71% of the throughput is lost in the case of compression, while just short of 90% is lost when decompressing. The effect worsens as the number of predictive steps increase. In light of this result the median scheme is not a viable alternative to the basic difference scheme.

The Lagrange predictor is accurate for smooth data that can be well approximated by some polynomial of degree n, for example a sinusoid. However, the noise present in the KAT-7 data renders this method of prediction unsuitable for the purposes of compression. Similar to the mean-based predictor, the scheme inflates the data when up to ten predictive steps are chosen. Similarly, the parallelogram predictor also degrades the compression ratio by approximately 3%.

Other solutions not explored in this investigation include using a lookup table to store predictions. This approach is used in the dictionary-based predictor by Burtscher et al. [3, 4]. This scheme uses the same block-based parallel approach investigated in this report. However, our primary concern with this approach is that, as the number of observed values grow, so does the size of the lookup table. This approach is therefore not as well-tailored towards a GPU implementation as our difference-based scheme, because it will require more global memory accesses than a simple lookup of a single previous observation. In future work it may be worthwhile to investigate whether a CPU implementation of the dictionary approach will work on the sample data produced by the KAT-7 array.

4.5.4 Vector intrinsics

The Intel Multimedia Extensions (MMX), SSE and AVX, along with the original 3DNow, SSE extensions and XOP extensions added by AMD, form a family of vectorized instruction sets which add the ability to perform a single instruction on multiple data elements of the same type in parallel. The original SSE instruction set added eight special 128-bit registers. This adds the ability to perform both integer-arithmetic operations and floating-point operations on any continuous sequence of elements stored into these special registers. It imposes the special condition that the elements loaded into a single register are all of the same bit-length (64, 32, 16 or 8-bit values).

As the name suggests these special machine instructions are focused on providing greater flexibility for performing a single instruction on multiple data. A good example of where

a speedup can be expected is in parallelizing vector addition. However, in the case of implementing a packing algorithm, the original SSE instructions are not sufficient since they only allow bit-shifting multiple elements by the same amount. The packing algorithm requires that four 32-bit values each be shifted by a different amount. A simple solution is to employ the AMD XOP instruction set, which adds the ability to perform element-wise logical bit-shifting. The problem with this approach is that no Intel processor prior to the latest (at the time of writing) Haswell series supports this operation. This functionality is only added through the AVX 2 extensions.

The next major problems are the limited number of 128-bit registers available, as well as the fact that the packing algorithm does not have the same number of outputs as inputs. Although much of the implementation can be rewritten as SSE and XOP instructions, while the storage section can be written as four separate instructions, this avenue of research proved fruitless, because of both the limited number of registers available, and the overheads of employing both normal instructions and vectorized code (specifically conversion between the two). In both the compressor and decompressor, the augmented code halves the throughput achieved before the SSE and XOP instructions were added. This finding precludes using the AVX 2 instruction set, which adds several 256-bit registers, that can be used to perform 8 32-bit integer/floating point operations, since it will likely suffer from the same problems as the SSE- and XOP-augmented implementation.

4.5.5 GPGPU implementation using CUDA

The GPU implementation is structured slightly differently to the CPU implementation. Instead of only splitting the data into blocks, each block is now distributed to an SM, which processes the block in parallel as well. This approach ensures that a high occupancy is achieved (in other words, all the multiprocessors are used). Since the data is split into SM-sized blocks each SM is free to execute with no need for intra-SM synchronization.

The implementation of the packing algorithm is based around the parallel prefix sum implementation described in GPU Gems 3 [7]. The parallel prefix sum computes two elements at a time, making use of shared memory to store preliminary results for each parallel step of the scan algorithm. The block size, therefore, has to be both a power of two, as well as small enough in order for the leading zero count array, and its accumulation, to be stored in shared memory, with the necessary padding per level to avoid bank conflicts.

Since both the packing and unpacking algorithms are similar in construction, and for the sake of time constraints, only the packing algorithm was implemented in CUDA, as it provides a fair estimation of whether a decompression scheme is feasible. As mentioned previously, the GPU implementation can be viewed as a post-processing step, which is executed after existing signal processing routines have been performed. A general compacting kernel requires the use of atomics, but still achieves up to 5.2 GiB/s on a GT670, and up to 13.6 GiB/s on a GT780 when processing a sufficiently large number of observations (distributed over time as before), to fill the memory capacity of the GPU. It was found that the best throughput performance is given when block sizes of 256 elements are used. However, using such small blocks (and the associated increase in

header information) can result in a penalty in compression ratio of up to 2.7% compared to a CPU version which uses a small number of blocks by comparison.

Currently, the number of correlated antennae is limited to 7+1 sources, each with 2 polarizations. However, as the number of antennae grow (to around 3000) the number of correlations will exhibit a quadratic growth rate. It is therefore sensible to flood the SIMD environment of the cards with enough information to fill the on-board memory of the card completely, in order to mitigate the setup costs involved when launching device kernels. The current datasets are artificially extended by duplicating the observations at time t until they fill the memory of the card. It is reasonable to assume that the overall properties of the datasets are preserved if the number of antennae are increased.

The approaches to removing the atomic operations from both prefix and residual compaction, as discussed previously, were implemented without much success. Branch-divergence and multiple write operations to global memory (in residual compaction) proved to be just as expensive as using atomic operations. The throughput remained roughly the same for both these approaches.

After considering all the kernel-level optimizations, including minimizing branch-divergence, using shared memory (and associated issues regarding bank conflicts) and investigating the removal of atomic operations, the issue of memory transfer between the device and host was taken into consideration. This copy operation should be taken into account in a scenario where post-processing is required, since no other operations are likely to follow a compression step, as well as due to the limitations in transfer speeds imposed by the PCI Express bus. Normally the cost of transfer can be hidden by executing multiple kernels while memory transfers are made through the use of asynchronous memory calls. However, due to the limited amount of memory on these cards, and the amount of data being processed, duplicating memory allocations in order to execute kernels and copy memory asynchronously is not a viable option. On both the GT670 and the GT780 the latency associated with a memory copy between each kernel launch proved to be very detrimental to the high throughputs seen for the execution of the compression kernel itself. When taking the transfer of the blocks of residuals and prefixes into account the compression throughput dropped to below 1 GiB/s.

Several different strategies were taken to minimize this latency, including transferring all the blocks of residuals and prefixes using a single copy operation (with the necessary padding between blocks of residuals to take into account the variable length of the blocks of residuals) and doing all the splitting on the host, *pinning* host memory, as well as *zero-copy* operations. Pinned memory simply allocates a block of memory in host RAM that cannot be swapped out to secondary storage, whereas zero-copy operations make memory transfers between host and device transparent, by mapping the host memory address space directly to the GPU memory address space. Such zero-copy operations are particularly useful when the memory allocation needed exceeds the limited on-board memory of a GPU.

Unfortunately, none of these approaches made a significant improvement in terms of throughput, although pinned memory showed a small 200-300 MiB/s increase in throughput if a large number of observations are transfered between the device and the host. In

light of these results, a post-processing GPU implementation may not be feasible unless the latency added by the memory copy operation can be hidden as part of the transfer cost between the host and RAID disk array (which can sustain transfer rates up to 4GB/s according to sources at the MeerKAT offices in Pinelands). Such a modification is, however, beyond the scope of the current investigation.

4.5.6 Notes on memory overheads

Due to the parallel algorithm being based on a block-based approach, all the observations at time t should be received before compression can commence. It will therefore be necessary to allocate enough space to accommodate buffering data, while compression is completed in a production environment. The difference-based predictive scheme, compacting up to 3 bytes at a time, requires at most $2n + \lceil 2n/8 \rceil + n + c$ 32-bit values every time the algorithm is executed. These numbers can be attributed to the following requirements:

- All observations at time t and $t - 1$ have to be stored.
- An array of length $\lceil 2n/8 \rceil$ is required to store the prefixes.
- An array of length n to store residuals, which includes padding in order to avoid unnecessary branch divergence within the algorithm.
- Some negligible amount, c, accounts for the rest of the program.

4.5.7 Notes on fault tolerance and error propagation

Due to the fact that the reconstruction of each observation at time t depends on the preceding, corresponding, observation at time $t - 1$, which in turn depends on the corresponding observation at time $t - 2$, error propagation could become a problem in a production environment, especially if network transmission uses a transport protocol without integrity checking, like the User Datagram Protocol (UDP).

If one of the residuals becomes corrupted (either through network error or disk corruption) each subsequent, corresponding, observation will be decoded to the wrong value. Since multiple observations can be written into a single 32-bit store, the observations in the immediate vicinity of the corrupted store (and their successors) will be affected. In addition, if one (or more) of the prefixes becomes corrupted, all the residuals after the first corrupted prefix, $P[c]$ will be decoded incorrectly as well. This is due to the fact that the accumulated prefix count over $P[0] \dots P[c] \dots P[m]$ is used to reconstruct the residual at index m. These errors will continue to propagate until a fresh initialization is done on both the compressor and decompressor.

In a production environment it may be necessary to include integrity checks using a *checksum* algorithm like the Merkle Damgard 5 (MD5) message digest, in order to validate output after all the observations at time t have been decoded. A message digest algorithm is simply a function that takes an arbitrary length input, M and maps it to a fixed length output, N. Any slight change in the input, M' will result in a substantially different

output, N'. The chances of a collision ($N = N'$) are very slim, meaning that the checksum provides a viable way of checking archive integrity. A checksum produced by MD5 is only 128 bits long and will not have a significant detremental impact on the compression ratio, since it only has to be saved for the entire set of observations at time t and not per block. Future work should investigate if such an integrity check can attain the throughput rates required. Alternatively, any errors can be mitigated by ensuring that the compressor and decompressor are reinitialized regularly.

4.6 Summary

We set out to describe the technical details of implementing a basic prediction algorithm, along with several alternative approaches to prediction and parallelization. The following table summarizes the results obtained from several implementation iterations:

Approach	**Throughput increase**	**Compression ratio increase**	**Success**
Basic compression and parallelization:			
Effective compression ratios	N/A	N/A	Depends, attains an average ratio of 91%
Element-wise parallelization	N/A	N/A	No
Block-based parallelization	N/A	N/A	Yes, attained 5+ GiB/s compressing and 9+ GiB/s decompressing
Alternative predictors:			
Storing dedicated sign-bit	No	Yes	Insignificant
Using integer subtraction instead of exclusive or	No	No	No
Integer mean predictor	No	No	No
Median predictor	No	Yes (1%)	No, significantly decreases throughput
Simple parallelogram rule-based prediction	No	No	No
Lagrange extrapolation predictor	No	No	No
Advanced parallelization techniques			
Vectorized instruction sets (Intel SSE and AMD XOP)	No	N/A	No
Post-processing GPU packing algorithm	Depends	N/A	Kernel achieves 13+ GiB/s. Not effective due to memory copy latencies.
Removal of atomics in GPU kernel	No	N/A	No

5 Results

5.1 Overview

We present detailed results on the algorithm throughput achieved by the basic difference-based predictor, compare the predictive compression scheme to standard utilities, as well as to techniques developed in concurrent research into MeerKAT data compression. The results are followed by a detailed discussion section.

5.2 Constraints on benchmarking environment

In order to eliminate the influence of the external environment from testing the following special conditions are specified:

1. Files in excess of 500 MB will be used for benchmarking. This greatly reduces variability in timings.
2. No other processes should make intensive use of primary memory once benchmarking has commenced. The sheer quantity of data being processed makes benchmarking a memory-bound operation.
3. Experiments will be repeated at least 10 times to ensure outliers can be removed before a mean is computed. Timings varying by more than roughly 10% to 15% from the average are considered outliers.

5.3 Algorithm throughput on a CPU architecture

The implementation of the block-based parallel difference predictive scheme was executed on one of the AMD Opteron 6274 nodes of the ICTS High Performance Cluster at the University of Cape Town. These nodes have four 16-core processors clocked at 2200 MHz with 128 GiB primary memory, clocked at 1333 MHz. The average algorithm throughput excluding any disk I/O, for a file of 216 x 4096 x 112 x 2 32-bit floats (756.0 MiB in total, excluding header information) is shown in figure 19. The average compression ratio achieved is 91.02% with a sample standard deviation of 2.01%.

5.4 Benchmark against standard compression utilities

The implementation was also benchmarked against the standard compression utilities GZIP, BZIP2 and ZIP on an 8 core Intel Xeon E5-2650 CPU[6] that supports up to 16 threads. The processor is clocked at 2000 MHz and has 64 GB primary memory clocked at 1600MHz. Disk I/O (network-based) was enabled for this run of the block-based parallel difference-predictive-scheme implementation in order to offer a fair comparison.

[6]This processor does not have the `LZCNT` machine instruction and is therefore much slower than the AMD Opteron.

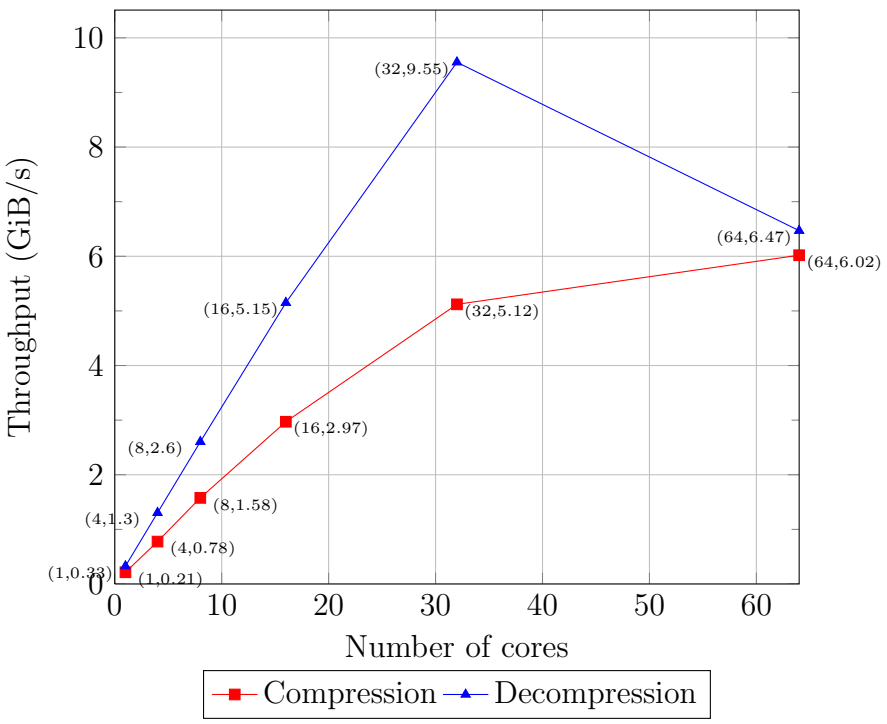


FIGURE 19: Algorithm performance using up to four 16-core AMD Opteron 6274 CPUs

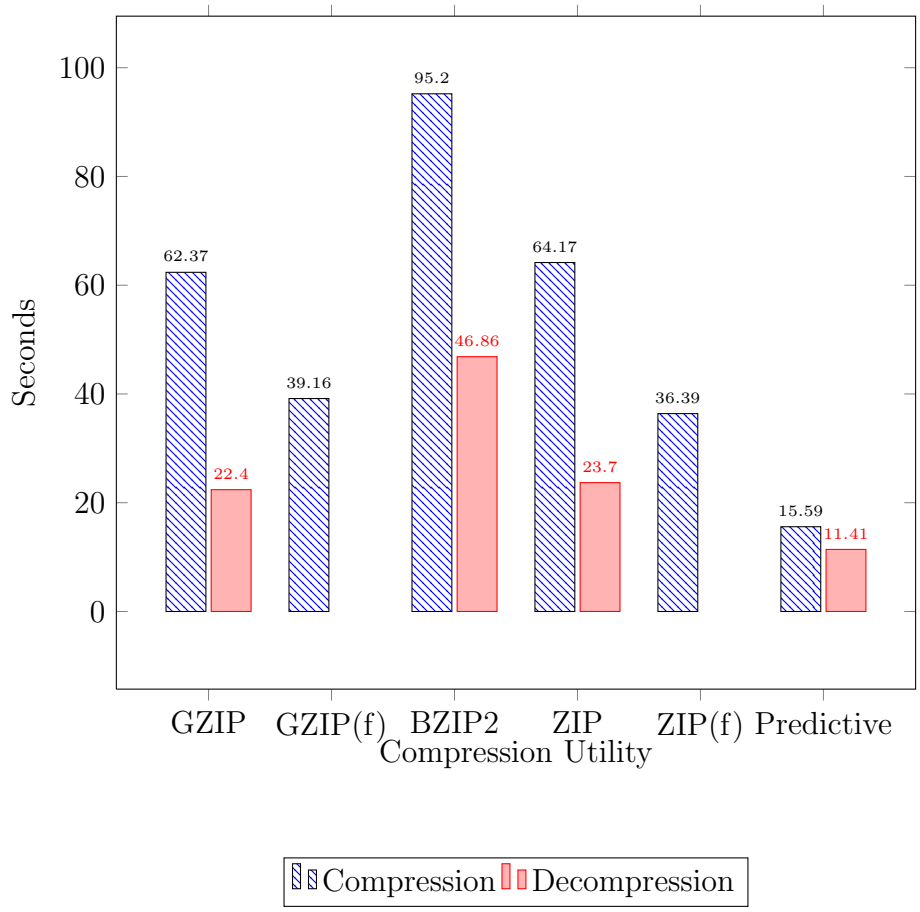
The performance comparison can be seen in figure 20. The experiment was run using the same file (including header) used for the throughput experiment. The utilities were executed using default compression and decompression settings. Using fast settings gives a 59.15% improvement (39.159s) for GZIP and a 76.36% improvement (36.386s) for ZIP (figure 20a). The compression ratio is decreased by nearly 30% in both instances (figure 20b). BZIP2 does not show an improvement in speed and the decompression times stay roughly the same for all the utilities. Only around 0.39s was spent on compression and 0.18s on decompression for the predictive scheme, the rest of the time was spent on disk I/O: reading from the original HDF5 files, writing a compressed file, reading the compressed file and finally writing a decompressed file.

5.5 Comparison to what was achieved in concurrent research

Research into both a Huffman entropy encoder, as well as a Zero-Length Coding scheme (Run-length encoding on binary data) achieved the results showed in figure 21 on the same file used in the previous experiments, on the same Intel Xeon E5-2650 CPU that was used in the comparison to standard utilities.

5.6 Algorithm development and improvements

As discussed in the implementation section our investigation included several iterations of improvements. Figure 22 provides an overview of how different approaches performed



(A) Comparison of compression and decompression speeds

GZIP	GZIP(f)	BZIP2	ZIP	ZIP(f)	Predictive
0.52417	0.7470761	0.39415	0.52545	0.7470763	0.933001

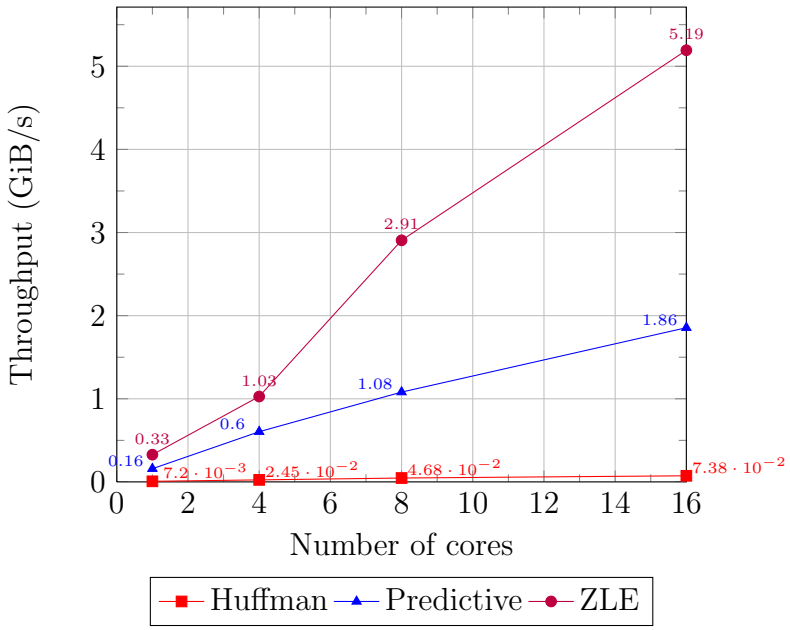
(B) Comparison of compression ratios

FIGURE 20: Comparison between the predictive scheme and the standard utilities GZIP, BZIP2 and ZIP

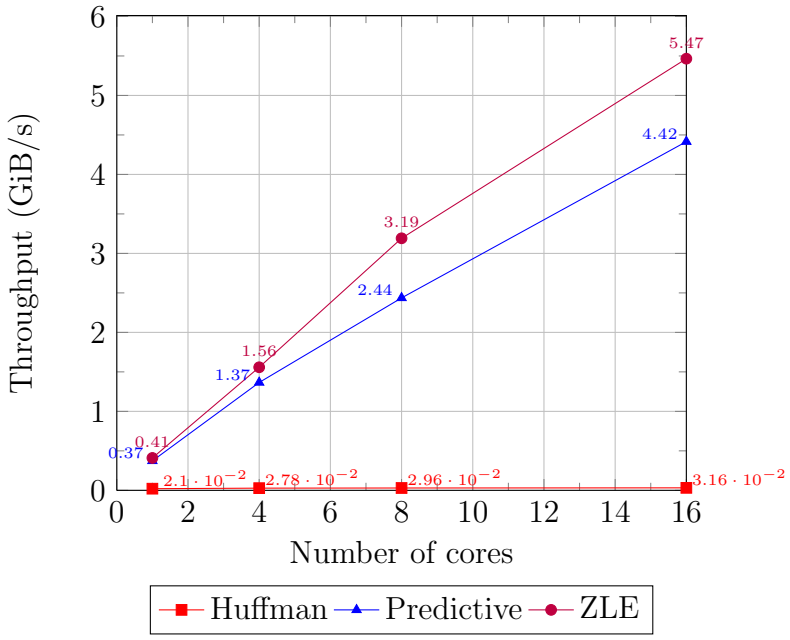
in relation to the fully optimized compressor and decompressor running on 16 threads on the AMD Opteron processor used in previous experiments. All other results are those obtained from running the solution on the same Intel Xeon E5 processor used previously. All these results excludes disk I/O operations.

5.7 Discussion

The block-based parallel approach achieves roughly a semi-linear scaling (doubling the number of cores almost doubles the throughput achieved) up to 32-cores for both the compressor and decompressor. After that point increasing the number of cores sees deminishing returns due to the overheads of processing smaller blocks in parallel, coupled



(A) Comparison of compression speeds achieved



(B) Comparison of decompression speeds achieved

Huffman	Predictive	ZLE
0.41	0.933001	0.969

(C) Comparison between compression ratios (on raw data, excluding h5 headers)

FIGURE 21: Comparison between Predictive Compressor, ZLE and Huffman coding (excluding Disk I/O)

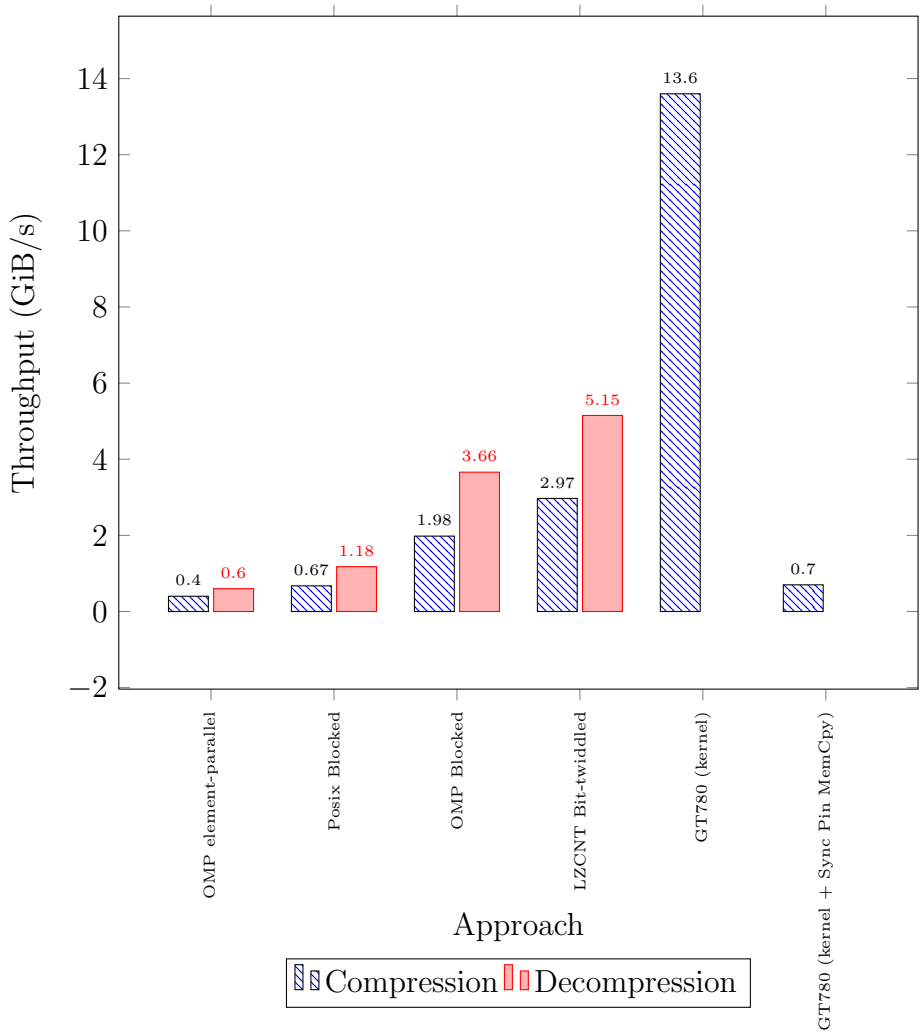


FIGURE 22: Comparison between several iterations of the implementation using 16 cores. The LZCNT optimizations are run on a slower AMD Opteron processor, but still outperforms a faster Intel Xeon E5 processor. For comparison the GPU kernel (with and without memory copy from device to host) is provided as reference.

with limited memory throughput capacity.

Although the compression and decompression algorithm is very similar in construction, the large difference in throughput between them can be attributed to the computation of the leading zero count per element in the compressor. Even though this leading zero count is computed using a dedicated machine instruction, the decompression step is roughly twice as fast as the compression step. Thus, faster decompression throughput is not surprising and corresponds to results reported for 64-bit double precision predictive compression schemes [4, 12].

Even when operating using only 16 threads on a processor without support for the leading zero count instruction, the predictive compressor implementation greatly outperforms the standard compression utilities set at their fastest settings. When common disk I/O is

subtracted the predictive compressor is roughly 50 times faster than ZIP (comparitively closest compression utility in speed). That said, the predictive compressor fails to achieve good compression ratios when compared to both the standard utilities, as well as the parallel Huffman encoder discussed in concurrent research. The relatively poor compression ratio can be attributed to noise in the data. Although there is some correlation between consecutive observations, the effectiveness of the predictor not only relies on consecutive observations having the same magnitude (the 1 byte exponents should be identical for most predictions), but the values should change very slowly in the top-most bits of the 23-bit significant. This limitation is not only apparent when compressing 32-bit single precision floating point values, but applies to 64-bit double precision floating point values as well.

When compared to the Zero-Length Encoder and Huffman coder on the Xeon E5 processor (without a leading zero count machine instruction) the predictive scheme shows roughly a 3.6% improvement in compression ratio compared to ZLE (this improvement may vary by 3%-4% depending on the characteristics of the input file), but the ZLE compressor is significantly faster (2.79 times faster than the predictive scheme using 16 threads). Additionally, the ZLE scheme does not have the memory requirement of storing observations at both time $t-1$ and t. Either one of these methods will help in reducing network bandwidth requirements. Although Huffman encoding does not meet the throughput requirements of an online compression scheme it is worth noting that this technique may be useful in addressing long-term storage requirements. The compression ratios produced by this scheme is on par with BZIP2, while being significantly faster (up to 2.9 times faster compressing and 1.43 times faster decompressing).

The difference in throughput of the implementation discussed in this report and those described in past research may be attributed to different processor speeds, memory clock speed and memory access patterns. The compressors mentioned in previous research operate on 64-bit double precision floating-point data and predictions are made on neighboring elements.

Over the course of our investigation into parallel predictive schemes several interesting observations were made: generally block-based packing approaches significantly outperforms parallel packing algorithms that operate at the element level. In the context of High Performance Computing it is better to use OpenMP to parallelize an implementation than process-based POSIX threads. This may be the result of the scheduling overheads associated with using process schedulers. Lastly we've considered implementing a packing algorithm using GPUs. We found that although such an implementation is possible, the latency associated with memory transfers between the host and device makes this approach infeasible.

In the context of the SKA, the implementation of a block-based parallel predictive scheme to reduce network bandwidth requirements is certainly attainable, provided that enough primary memory is available to store two time slices of observations, as well as the resulting prefix and residual arrays. Although the parallel predictive scheme is slower than the parallel ZLE scheme, it obtains significantly higher compression ratios. Long-term storage requirements is, however, better addressed by a Huffman encoder.

6 Conclusion

This investigation set out to evaluate the feasibility of using a predictive compression scheme to achieve high throughput compression on data produced by the MeerKAT radio telescope array, currently under construction in the central Karoo region of South Africa. The array will require processing facilities and network infrastructure capable of handling up 1 Petabyte of data every 20 seconds once completed. A fast compression scheme will be able to reduce both network bandwidth requirements, and ultimately, the storage space required to store data over extended periods of time. Some of the key research questions included:

1. Determining whether a predictive compression scheme achieves comparable results to that of standard compression utilities: we found that this goal was not attainable, but that the predictive compressor only produced compression ratios of 91% on average, in contrast to compression ratios of 39%.

2. Determining whether such a predictive scheme can operate at Infiniband network speeds of up to 5 GiB/s: the predictive compressor successfully met this target, while the decompressor well-exceeded these expectations.

3. Whether, using different predictors to trade throughput for improved compression ratios and vice-versa, is feasible: although several simple approaches were investigated, none of them had the desired outcome. This question remains open, as using dictionary-based prediction may still yield the desired tradeoff.

Although several parallel implementations were tested it was found that a block-based parallel approach achieved significantly better throughput rates (up to, and exceeding 5 GiB/s, using 32 cores) compared to an element-wise parallel approach in which the packing scheme is centered around the computation of a parallel prefix sum. This throughput was found to scale roughly linearly up to 32 CPU cores. Furthermore, employing a vectorized instruction set to enhance the performance of the block-based parallel approach proved infeasible. Additionally, merging the two approaches by splitting incoming data into blocks and processing those blocks in parallel on a GPU as a post-processing operation achieved high throughput rates of up to 13.6 GiB/s on a GT780. However, since such a post-processing operation should take the latencies associated with copying memory back to the host into account, GPU-based processing is ineffective if these latencies cannot be mitigated through asynchronous disk operations.

After investigating improvements in throughput focus shifted to improving the average compression ratio achieved, by taking the difference between successive samples. Such a basic difference scheme takes observations at time $t-1$ and exclusive ors them with observations at time t. It was found that using the exclusive or operator instead of normal integer subtraction was both faster and improved the average compression ratio. The scheme, however, failed to achieve similar compression ratios compared to the standard compression utilities GZIP, BZIP2 and ZIP, and only achieved savings of, on average, 9% in final file size. However, the predictive compression scheme is 20.801 seconds faster than the fastest compression utility (ZIP), when the latter operates at its maximum speed setting. When the common disk I/O latencies are subtracted from these timings

the solution is roughly 50 times faster than ZIP on a 16-thread implementation (without processor-based leading zero count support).

Using alternative predictors employing only integer arithmetic operations proved unsuccessful. These alternative predictors included basing each prediction on a mean, a median and polynomial extrapolation using both a parallelogram and Lagrange predictor. Only a median-based approach was found to give slightly better results (roughly 1%) using a small number of previous observations. However, even though a pivoting algorithm is employed to select the median for each prediction, the implementation is too slow to be considered a viable alternative to the simple difference-based predictor.

Both predictive or Zero-Length Encoding schemes therefore prove to be viable solutions for future implementation as part of the network stack employed by MeerKAT and ultimately the SKA. It remains to be investigated whether a predictive compression scheme, based on the use of lookup tables gives better results, although this may have considerable memory requirements depending on block size (see [13, 3, 4]). Using a Zero-Length Encoder on the residual stream should also be investigated as this may result in a slight improvement in compression ratio. The predictive scheme clearly outperforms regular compression utilities, as well as a Huffman entropy encoder in terms of throughput, but sacrifices a considerable amount of the possible compression ratio in doing so. An entropy encoder such as Huffman or a parallelization of a Lempel-Ziv scheme, as discussed by Klein et al. [9], may be better tailored to address the long-term storage requirements of the project.

7 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] Richard Cole and Uzi Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and computation*, 81(3):334–352, 1989.

[6] 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.

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

[8] Lawrence Ibarria, Peter Lindstrom, Jarek Rossignac, and Andrzej Szymczak. Out-of-core compression and decompression of large n-dimensional scalar fields. *Computer Graphics Forum*, 22(3):343–348, 2003.

[9] Shmuel Tomi Klein and Yair Wiseman. Parallel lempel ziv coding. *Discrete Applied Mathematics*, 146(2):180–191, 2005.

[10] Richard E Ladner and Michael J Fischer. Parallel prefix computation. *Journal of the ACM (JACM)*, 27(4):831–838, 1980.

[11] 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.

[12] 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.

[13] 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.

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

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

[16] Julian Seward. bzip2 and libbzip2. 1996.

[17] Senthil Shanmugasundaram and Robert Lourdusamy. A comparative study of text compression algorithms. *International Journal of Wisdom Based Computing*, 1(3):68–76, 2011.

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

[19] 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.

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

[21] N. Wirth. *Algorithms + Data Structures = Programs.* Prentice-Hall series in Automatic Computation, Englewood Cliffs, New Jersey, 1976.

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

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