University of Cape Town

Department of Computer Science

CSC4016W - Honours in Information Technology

A Fast Zero-Length Encoder for Astronomy Data

Author:
Heinrich Strauss

Team members:
Benjamin Hugo
Brandon Talbot

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

External Advisor:
Jason Manley

28 October 2013

A thesis presented to the Department of Computer Science, UCT, in partial fulfilment of the requirements of the course CSC4016W, Honours in Information Technology.

	Category	Min	Max	Chosen
1	Requirement Analysis and Design	0	20	**10**
2	Theoretical Analysis	0	25	-
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	-
Total marks		**80**		**80**

Acknowledgements

I would like to acknowledge and thank Associate Professor James Gain and Associate Professor Patrick Marais of the Department of Computer Science at the University of Cape Town for their careful and considered input on and supervision of the project.

Warm thanks also go to Jason Manley at the SKA South Africa offices in Pinelands, Cape Town for his pivotal role in formulating and defining the project. He has also kindly provided the data samples (raw KAT-7 output) for the testing phase of the project.

To my team members, Brandon and Ben, thank you for your input, patience and camaraderie during the project phase.

To family and friends, whose understanding of my absence and avoidance during this project does not go unnoticed, my heartfelt appreciation is sincerely shown.

Computations were performed using facilities provided by the University of Cape Town's ICTS High Performance Computing team: `http://hpc.uct.ac.za`

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.

Abstract

Observations from Radio Telescopes, such as the KAT-7 prototype to the Square Kilometre Array, can emit raw floating-point data at a phenomenal rate, often approaching 18 Tb/s. In order to keep the data as accurate as possible (for historical analysis up to 6 months after observation), we investigate zero-length encoding (ZLE) as a means to reduce disk cost and enable sufficient I/O throughput to be able to sustain a disk-write speed of 36-40 Gb/s (throughput; the current network bandwidth limit) and at a reasonable compression ratio in order to reduce cost of storage.

Whereas parallel team-research investigates methods known to provide better compression ratios (such as adaptive Huffman coding and predictive methods employing a parallelogram predictor), this paper's implementation focuses on throughput over ratio (since there is not sufficient redundancy with high locality to effect a major decrease in size).

Three attempts at implementations of ZLE are made: a naïve single-threaded prototype, an optimised and vectorised CPU implementation and a CUDA prototype. These are compared to concurrent team work and we determine that ZLE provides the required throughput at an almost negligible compression ratio (about 0.97:1.00) in the optimised CPU implementation.

Overall, Zero-Length Encoding and the concurrent research into Predictive Methods seem to satisfy the throughput requirement once the solution is scaled to 16 cores. The compression ratio is only attained through Arithmetic Methods, which do not perform at the required throughput levels by a few orders of magnitude but provide better compression ratios than general compression utilities.

Contents

List of Figures

Glossary of terms used in this report

- **Bit-twiddling**: The use of binary logic functions (AND, OR, XOR, NOT) and their combinations on integral value types to trade source-code readability for (usually impressive) algorithmic speed-up. The speed-up depends on the Arithmetic and Logic Units (ALUs) of a processor performing these native functions faster than a naive approach.
- **C++11**: The latest standard of the C++ programming language, ratified by the ISO/IEC in 2011. Introduces new language features such as generic programming (allowing a single function definition to be programmed with unspecified parameter data types) and enhanced multithreading support.
- **CUDA C**: A variant of the C programming language for GPGPU programming on NVIDIA CUDA-capable devices. Notably, recursion and dynamic parallelism were absent from initial releases, but features may be implemented in future with newer Compute Capability and hardware revisions.
- **GPGPU**: General-Purpose programming on Graphical Processing Units. A programming methodology in which specialised hardware commonly used for computer-based gaming or computer-aided design is repurposed to enable highly-parallel data processing. More recently, dedicated GPGPU platforms have become commercially available (e.g. the NVIDIA Tesla range of GPUs).
- **Fermi** The most widely-available (at the time of writing) NVIDIA architecture; This comprises all Geforce Series 200 and higher cards with GF*XYZ* as their model number.
- **HDF5**: Hierarchical Data Format, version 5; a common data exchange format for scientific data maintained by the HDF Group. Data access methods allow for "striding" of data reads (i.e. reading non-linear data-blocks which are stored at predictable offsets). It is the (incompatible) successor to the still commonly used HDF4 format, additionally providing split data files and unlimited data-set sizes.
- **Kepler** The newer K20- and K40-based cards; at the time of writing, only the NVIDIA GeForce TITAN, GeForce GTX 780 and Tesla K20, K20X and K40 cards have been released. Cards are identifiable by GK*XYZ* as their model number.
- **kb**, **Mb**, **Gb**, **Tb**: Kilo-, Mega-, Giga- and Tera**bits**, respectively using SI definitions ($\diamond \times 1000$)
- **kiB**, **MiB**, **GiB**, **TiB**: Kibi-, Mebi-, Gibi- and Tebi**bytes**, respectively using Binary SI definitions ($\diamond \times 1024 = 2^{10}$)
- **SSE**: Intel® Streaming SIMD Extensions; a means of applying a Single Instruction to Multiple Data in designated CPU registers on CPUs which support these extensions.

1 Introduction

1.1 SKA South Africa and its major projects (KAT-7, MeerKAT and the SKA)

The Square Kilometre Array (SKA) is a radio-telescope project currently in development across South Africa and Australia with a proposed collection area of approximately one square kilometre.

The currently active prototype is an array of seven 12-metre dishes, named KAT-7 (for Karoo Array Telescope). Commissioned in 2013, it precedes the 64-dish MeerKAT installation due for "first-light" (initial observations) in 2016. Further phases of the MeerKAT project will be completed in 2018.

The full complement of the Square Kilometre Array is expected to be completed in 2024. At this point, the array will be comprised of more than 3000 dishes, of which the low-frequency analysis collectors will be based in Australia and the medium- and high-frequency collectors will be based in South Africa. Together, these will cover a collection area of about 1 km^2.

The SKA South Africa offices, with whom we have had contact, is concerned ostensibly with the operations and operationalisation of the South African elements of the Square Kilometre Array. More information on the organisation can be found at their web presence[1] in the footnote below.

1.2 Data compression and decompression

The compressibility of data is determined by the amount of redundancy in a given amount of data. While redundancy will be formally defined in chapter 2.1.3, it may be useful to think of it as the amount of extraneous information in a given data-set. By removing redundancy, we decrease the amount of space necessary to store a given amount of data. Exploiting this to compress data requires knowledge of the structure of the source data, such as which elements are often repeated or which elements routinely follow other specific elements.

As an example, we take the poetry of William Blake, *The Tyger* in figure 1. Compression of this text uses both elements of the English language and the rhyming structure of the poem to compose a reasonably compressed (but by no means optimal) version of the poem. In networks where transfer costs are non-negligible, this can be transmitted at a much lower-cost (498 bytes if the dictionary is excluded) to the sender, but would be incomprehensible on the receiver without the decompression algorithm and dictionary.

On the receiver end, we use the dictionary of contractions sent as part of the compressed message or known beforehand to reconstruct the text. Since the input to the compression algorithm exactly matches the output from the decompression function, this is as a lossless compression. We could obtain better compression by, for example, removing punctuation, such as the ampersands, exclamation marks, question marks, commas and spaces. Alternatively, since the letter-case is almost determined completely by English grammatical rules and Christian conventions, this can be omitted from the compressed data and could be reasonably reconstructed on the receiver side given proper context. This is a form of lossy compression, since not all the uncompressed input would be accurately restored after decompression (that is, some information may be lost).

[1]SKA Web Presence: `http://www.ska.ac.za`

1.3 Comparison of approaches

With a view towards GPGPU implementations of the compression algorithms, three of the four avenues were actively pursued the three team members:

1. *Arithmetic methods* by Brandon Talbot,
2. *Predictive methods* by Benjamin Hugo; and
3. *Basic run-based methods* by myself, Heinrich Strauss.

The fourth avenue, transforms, were excluded given the relative scope and complexity. Arithmetic methods are also not expected to perform well in terms of throughput, based on previous attempts to parallelise an implementation of chemistry molecule data manipulation using GPGPU methods by our supervisors in 2012[14].

The first method largely attempts to address the compression ratio of the data first and the throughput as a secondary target. The last two pursued methods (with zero-length encoding discussed in this report) attempt to address throughput first, since they employ more rudimentary methods of data compression, which trades-off compression ratio for speed.

1.4 Compression of KAT-7 astronomy data

The data pipeline for KAT-7 is shown in figure 2. From left to right, raw data is collected by the seven dish-array, and manipulated using FPGAs to ensure data throughput to the processing nodes. The data rates when crossing from FPGA to commodity processing nodes can (theoretically) approach 16 Tb/second. While the current hardware is only capable of operating routinely at 40 Gb/s, it would be useful to store data from these observations for future analysis. Presently, this requires a sacrifice of data accuracy, which being observations of ephemeral data, cannot be reconstructed.

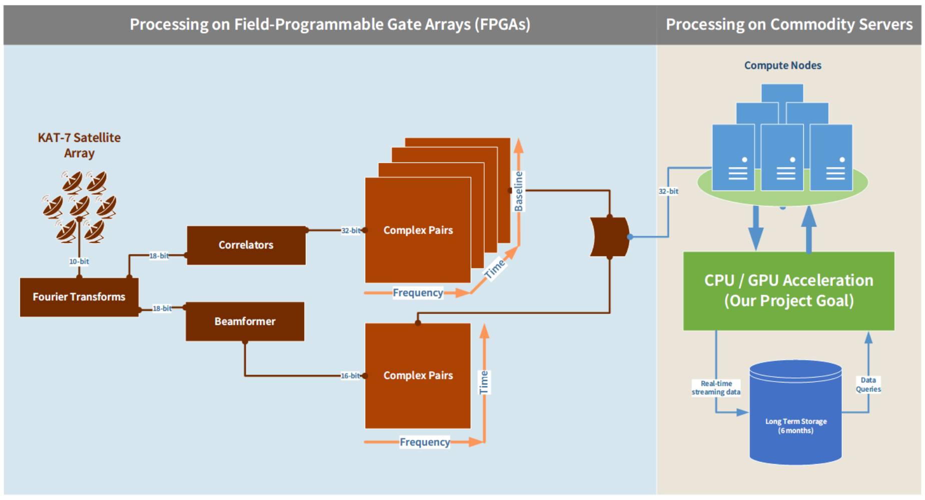


Figure 2: The current KAT-7 data pipeline. The green rectangle (right) indicates this project's location within the greater SKA data-collection pipeline. Pipeline elements on the left (blue background) are handled by FPGA computation and those on the right (yellow background) by General Purpose Computing.

The intent of this project is to determine whether we can reduce the network and disk I/O footprint of the collection process in order to store accurate data, which can be moved to the analysts requiring raw output. To this end, we attempt to transparently insert ourselves into the data pipeline at the green rectangle in the right of figure 2. This would result in lower hardware utilisation as well improved throughput, so less (or ideally no) data would have to be discarded for at least the 6-month post-observation period (which is considered long-term storage).

1.5 Compression algorithm properties and measurements

There are two major classes of compression algorithms: *lossless* and *lossy*. *Lossless compression* describes a scheme whereby it is possible to reconstruct the exact (uncompressed) input of the scheme. Archival copies of data and, more recently, lossless digital copies of digital media (such as music), are common examples of this type of compression. *Lossy compression* is used where data features can be "smoothed" to afford better compressibility of the data. Commonly this is applied to photographs, where gradient-filled areas can then be described more concisely, or lower fidelity audio files (such as MPEG-2 Layer 3 or Ogg Vorbis audio copies), where imperceptible audio artefacts may be culled from the stored copy. Software implementing lossy compression often allows the user to select a tunable amount of acceptable loss of quality to improve compression rates.

The compression scheme may also be classed as *online* or *offline*, where the former allows the data to be reconstructed in realtime by subsequent processes in a pipeline. An added benefit of online schemes is that a user may seek through data to a specific point, if the data is arranged chronologically. Offline schemes may provide better compression ratios, at the expense of additional resources (storage and memory) to reconstruct the data. [21]

A common real-world example of this is in the differences in random file access in the `zip` and `RAR` compressed file implementations. In the former, a single file can be extracted without linearly reading through the entire archive, whereas the latter requires all previous files in the catalogue to be read-through first. This gives the impression that decompression is slower, although it is largely an artifact of the compressed storage methodology.

Salomon [16, p. 10] defines the concepts of *Compression ratio* and *throughput* as:

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

$$\text{Throughput} \triangleq \frac{\text{input processed (in GB; SI units)}}{\text{difference in time (in seconds)}} \tag{2}$$

A compression ratio near 0.0 indicates a small output file (good compression) and a value greater than 1.00 indicates data inflation instead of compression. Inflation can be due to the algorithm or the structure of the data, so an algorithm might perform well on audio data, for example, but may inflate picture data more frequently than compressing it.

Data which is approximately random does not provide good compressibility[16, p. 71]. This underpins the compression ratio disparity between lossless and lossy data and is evident when comparing audio in a noisy environment to audio taken in a controlled, silent environment.

Since the accuracy of data collected is of importance to the researchers collecting it, lossy compression is excluded for consideration from the project.

1.6 Research questions

In order to modularise the project (and hence simplify its implementation in a live environment), the software is designed as a shim[2] between the data collection and storage roles within the data pipeline.

[2]shim: a small library that transparently intercepts and modifies calls to an API, usually for compatibility purposes. *cf.* ca. 1860, "thin piece of material used for alignment or support"

This allows the project to focus on optimising the algorithms rather than the actual storage subsystem implementation.

There are two key-performance indicators for the proposed shim: Throughput and Compression Ratio. Of these throughput is the more pertinent, given the expected data rates, as a decrease from the data collection rate would lead to loss of data if the throughput is below the disk writing speed. Compression ratio would reduce the operational cost of the storage system, especially once scaled out to the full project capacity.

We intend to answer the following questions through our research:

1. Can we achieve effective throughput rates of at least 40 Gb/s (data line-rate; 36 Gb/s nominal data throughput)? These represent the network and disk subsystem speeds (based on the Ethernet- and InfiniBand-based architectures) currently in use at the SKA South Africa offices.
2. Can we reduce the storage requirement significantly by using data-compression? Reductions on the order of 10—20% (as mentioned in initial project meetings with the SKA) would translate equivalently to the financial cost of the disk subsystem. The ability to achieve this is dependent on the algorithm used and the data presented, as it may be too random to effectively compress.
3. Can throughput rates be traded for compression ratio by using different algorithms? If there is enough gain from the compression ratio, this may be traded for throughput if this is not sufficiently fast.

Based on the scope-limit of the project, we will focus only on the algorithmic performance and not on integration into the SKA data-processing pipeline (figure 2). Three iterations will be detailed: a naïve single-threaded implementation, a SIMD-vectorised and optimised multi-threaded implementation and a GPGPU implementation for use on nvidia devices. The project is concerned nominally with the feasibility and impact of such compression routines and not with immediate operationalisation of these implementations.

In the background chapter, we detail the compression techniques employed throughout the project. The Design and Methodology chapter (ch. 3) will focus on how the development and testing platforms were constructed. The Implementation chapter (ch. 4) deals with describing the intricacies inherent in the design and implementation. Finally, the Results and Discussion (ch. 5) chapter will detail the raw findings and place them into context.

2 Background

2.1 Overview of data compression techniques

Four main categories of compression schemes are prominent, as noted by Salomon[16]. These are basic methods (on which we focus in this report), Lempel-Ziv (LZ) methods, statistical methods and transforms.

2.1.1 Basic Methods

The most intuitive method of compression commonly employed is Run-Length Encoding. This focuses on eliminating sequential redundancy but grouping consecutive like symbols into "runs." The symbol and the length of the run are then recorded and this can trivially reconstruct the original data.

Since the runs investigated would have redundancy at the bit-level, it is necessary to compress the runs of zeroes and ones. This yields a small dictionary, but requires a fast and efficient packing scheme in order to adequately compress data, since the size of each dictionary entry is less than 1 byte, the nominally elemental data storage unit. Naturally, this would require a great deal of bit-field manipulation ("bit-twiddling").

2.1.2 Predictive Methods

Predictive compression schemes attempt to synthesise the next symbol based on patterns previously observed. If the data has regular patterns (often found in time-series data), this type of compression could encode the difference between adjacent symbols (if it is smaller than the actual data) and store that instead of the actual data. Again, the source data can be completely constructed from the compressed data and the algorithm.

Lempel-Ziv (LZ) methods, credited to Abraham Lempel and Jacob Ziv[22], are also known as dictionary methods.

While many algorithms derive from the original methods discovered, LZ77 and LZ78 (so-named due to them being created in 1977 and 1978), these methods are considered some of the most popular and pervasive adaptive techniques still in modern use. In their basic form these methods use searches and lookahead buffers to encode patterns using fixed-size codes (in a lookup "dictionary"). When the underlying set of patterns is not known in advance (by a previous pass to determine these) or is temporal (with the amount of data processed), an adaptive variant of these technique becomes useful. This would result in adjacent blocks having smaller and distinct dictionaries.

The GNU compression utility `gzip` implements the DEFLATE algorithm[6] which is a variant of the first Lempel-Ziv compression scheme, LZ77. The newer `bzip2` uses the Burrows-Wheeler Transform to achieve better compression rates at the expense of CPU time.

LZ77 encodes these patterns as (length,distance) pairs. If a sequence of characters is 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 is encoded. The size of the search buffer can be up to 32 kB in some implementations of the algorithm [16], although the larger the search buffer, the slower the implementation would be in practice.

2.1.3 Statistical Methods

This class of algorithms normally uses variable-length codes to achieve an optimal encoding of dataset . In Information Theory, this optimal encoding is described as an entropy encoding. Entropy is defined as the measurement of the information contained in a single base-n symbol (as transmitted per unit time by some source, derived from the definition due to Claude Shannon in his 1948 paper[18]).

Redundancy is defined by Salomon[16] as the difference in entropy between the optimal encoding and the current encoding of a data set and 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):

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

These methods use historic probability of occurrences to limit redundancy. Two well-known methods are Huffman-coding and Arithmetic Coding.

Huffman Coding, named for David Huffman[11], assign short integral codes to symbols, with the most frequently occurring symbols being assigned the shortest-length symbols. It operates by determining the frequency of occurrences of each symbol, sorting it and placing it into a binary tree, with the leaves denoting the encoded symbols.

It starts with the rarest occurring symbols and places them into a binary tree, with `0` denoting a left-traversal and `1` denoting a right traversal. It then places the sub-tree back into the list of symbols, while preserving the established sub-tree ordering. The process is then repeated until a single binary tree remains, where the number of occurrences of each subtree is the sum of the occurrences of the nodes below it.

The encoding for each leaf-node is unique and the desired property of having shorter representations for frequently occurring symbols is achieved.

Refer to figure 3 for a worked example based on a poem by William Blake. The symbol for a space will be represented as `00` while some of the much-less frequent symbol *k* obtains a representation `110101111` in this particular example.

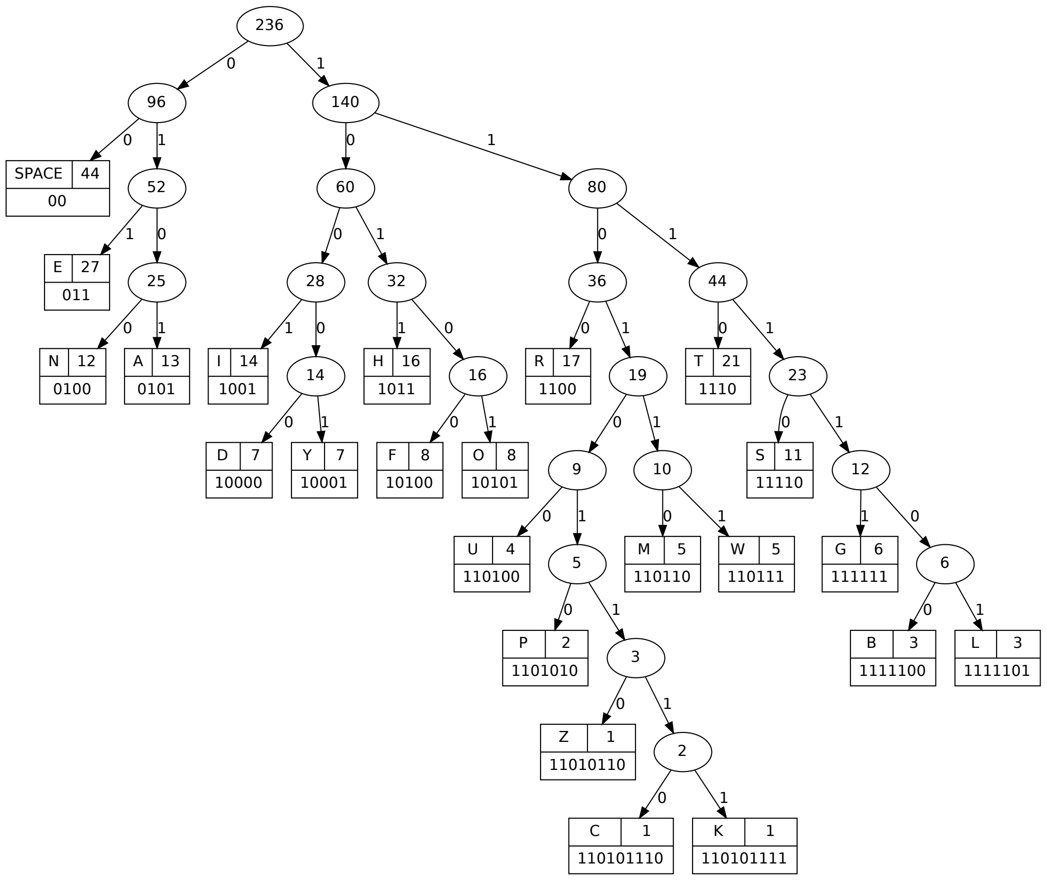


Figure 3: Sample of a completed Huffman-coding binary tree for the first two verses of *The Tyger* by William Blake, excluding punctuation (source: `http://huffman.ooz.ie`). Note that the most frequent symbol is space (00) and the other frequencies follow in the expected order (E,{T,A,I},...) which is natural for letter frequency occurrence in English. The least frequently used characters have the longest representation (K, for example has representation 110101111)

Arithmetic coding uses floating-point values in the range $[0, 1)$ to encode the entire message as proportional subintervals instead of assigning integral values to store the probability of encountering a symbol.

As with Huffman a probability distribution has to be known before the algorithm can be executed. The initial interval $[0, 1)$ is reduced by each consecutive symbol. The length of the sub-interval is proportional to the probability of occurrence of the symbol being read, which must be updated upon reading each symbol. The final symbol encountered is assigned an arbitrary value inside the remaining sub-interval.

As the interval between symbols decreases, the number of bits required to encode each consecutive interval increases. However, the average number of bits needed more accurately reflects the entropy encoding of the message, since this average number can be floating-point value, unlike that of the integral, per-symbol encoding of Huffman [16, ch. 2].

Both these arithmetic approaches emit variable-length codes. There are also adaptive implementations of both these methods, where the frequency distribution table changes rapidly or is not possible to know ahead of time. This allows these approaches to be used on streaming data (with no possibility of a second pass over the data once the frequency distribution has been calculated), as we will require for this project or data where the structure changes in a block-wise manner (for example encoded video, followed by audio structured block-wise).

In both adaptive and classical cases, arithmetic coding provides better compressibility than Huffman coding. The decompression algorithm for Huffman coding also does not lend itself well to online decompression[16, ch. 2], so this may preclude deployment of this algorithm if fast decompressed access to the data is required.

2.2 GPGPU programming primitives

2.2.1 Threads, blocks and grids

Massively parallel tasks, those with hundreds to thousands of parallel threads of execution, are divided into blocks of threads which are executed simultaneously on a Streaming Multiprocessor Unit (or SM). We concern ourselves with the Compute Unified Device Architecture (CUDA) created by nVIDIA and its programming language, CUDA C, derived from C with a few additions and limitations, as noted in the glossary.

A *thread* of execution is a nominal unit of work which can be independently handled by a single processor. On general-purpose machines, multiple threads are usually handled through context-switching on the CPU, in which the state of a thread is maintained for the time which it has exclusive access to the CPU (usually on the order of nano- or microseconds) and then swapped out for the state of another thread. To the average end-user, it appears as though multiple threads are running concurrently, whereas a single thread is in operation at any given time for a single processing core. In the GPGPU architectures, a large number of cores (on the order of hundreds to thousands) are available per Streaming Multiprocessor (SM), so that as many tasks execute simultaneously (provided there are no inter-thread dependencies).

Because threads may need to co-ordinate resources (such as memory), and the number of possible interactions grows quadratically with the number of threads, it is necessary to reduce the impact of such interactions on execution speed. A *block* of threads is a collection of execution threads which have a means of communicating with one another (usually through shared memory). By thresholding the number of interactions during the application design phase, the number of threads can scale transparently as more powerful processing hardware becomes available, since blocks can be assigned to any free processing unit.

By splitting the desired workload into threads of a particular block size, we can determine the number of blocks required to complete the desired task. These are arranged in a *grid* (often a 2D array-like structure), which allows addressing of the individual blocks and reconstitution of the individual results into the combined final result.

2.2.2 Kernels

A *kernel* is a body of computation that is performed multiple times by many threads on similar data. In CUDA, multiple threads run a kernel simultaneously.

For example, in the code sample (figure 4):

```
FOR j = 1 to number_of_rows
  FOR i = 1 to number_of_columns
    tableOut[j][i] = tableIn[i][j] // THIS IS THE KERNEL
  END FOR // i
END FOR // j
```

Figure 4: An example of a programmatic kernel for transposing a matrix.

the line `tableOut[j][i] = tableIn[i][j]` is the kernel of the code. As such, we may think of a kernel in general to be the body of a loop. In GPU processing, the looping constructs are usually handled by the geometry called at kernel invocation time, which is comprised of the number of processing blocks and number of threads per block (and optionally the amount of shared memory and processing stream identification).

In CUDA C, this is specified as:

```
__global__ void kernel_function(void* gpu_input_1, uint64_t gpu_input_size){
    ...
    // Do function on GPU
    ...
}

__host__ void call_kernel (void* func_param_1, uint64_t func_param_2){
    ...
    // Prepare data for GPU and copy input to GPU using
    // cudaMemcpy(dst,src,sz,cudaMemcpyHostToDevice);
    ...
    kernel_function <<< blocks,threadsPerBlock >>> (func_param_1, func_param_2);
    ...
    // Prepare data from GPU and copy output to Host using
    // cudaMemcpy(dst,src,sz,cudaMemcpyDeviceToHost);
    ...
}
```

Figure 5: Calling a CUDA kernel using CUDA C. The <<< `blocks,threadsPerBlock` >>> segment is called the launch configuration and specifies how many GPU threads should be launched on the CUDA device.

In CUDA C, all kernels return `void` (in other words, any output must be obtained from changes effected by reference to one or more of the of the kernel function arguments called) in the current CUDA Compute modes, but this is may change in future versions. Accessing the results of the kernel is usually done through functions such as `cudaMemcpy(...)` into a host-allocated memory block mirroring the data-structure on device once the kernel has completed its work.

The most useful GPU functional primitives that we intend to use are **map**, **reduce** and **stream filtering**. The former two are usually combined (and referred to as map/reduce) when a task is parallelisable. The *map* operation splits the overall task into parallelisable subcomponents and applies the kernel function to every subcomponent. The *reduce* operation will recombine the individual outcomes from processing the individual subcomponents into the expected output format. *Stream filtering* involves some additional branching in the reduce operation, whereby certain criteria may exclude subcomponents' kernelised results from being recombined into the final result (for example, exceeding a threshold) and so can be thought of as an obvious generalisation of the reduce operation.

2.3 IEEE 754–2008 data structures

The IEEE 754–2008 standard defines a format for storing and manipulating floating-point numbers (in 32-bit, 64-bit and 128-bit sizes). Although the official names are binary32, binary64 and binary128 (in reference to their bit sizes), we refer to these as *single-precision*, *double-precision* and *quad-precision floats*. Figure 6 shows the bit-layout of a binary64 structure and figure 7 shows the limitations of precision in binary and decimal. Each of these have a common construction with the following subfields:

- A 1-bit sign
- A μ-bit exponent
- A $(d - \mu - 1)$-bit significand, where the leading bit of the significand is implicitly encoded in the exponent and d is the bit-size of the structure.

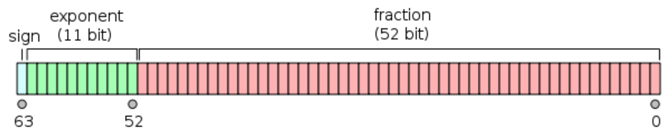


Figure 6: IEEE Interchange floating-point format for a binary64 structure (source: Wikimedia Commons). Note that the most-significant bit indicates the sign and the least significant bits the significand (or fraction).

Size	*Exponent width*	*Significand Precision*	*Approximate Decimal Precision*
32-bits	8 bits	23 bits	$6 - 9$ decimal places
64-bits	11 bits	52 bits	$15 - 16$ decimal places
128-bits	15 bits	112 bits	$33 - 36$ decimal places

Figure 7: Limitations on range and accuracy of the IEEE binary32, binary64 and binary128 floating-point number interchange formats. Note that significand precision (number of digits) more than doubles for each bit-wise double binary-type.

Since these structure natually fall on a byte-boundary (being multiples of 8 bits), this is presented for informational purposes only. The read data will be manipulated solely with the CPU's Arithmetic and Logic Unit (ALU). This has the complimentary side-effect of precluding any Floating-Point Processing Unit (FPU) rounding concerns, which is pivotal to lossless compression.

Should the data need to be accessed as its native (floating-point) value, the memory block can be converted in place through array-casting and the read values will accurately be presented as floating-point numbers.

2.4 Bit-manipulation (or "twiddling")

Logical expressions are often expressed concisely by packing multiple 1 bit long items into an integral-type variable, such as a byte or integer. This variable then becomes known as a bit-field.

Individual elements are committed to or extracted from a bit-field using logic functions and bit-shifting, a technique known as bit manipulation or "bit-twiddling". For example, if `bf` has the representation `0b00101001` and one want to check the third-least significant bit (0-indexed), one could use `((bf) & (1<<3))` to obtain a non-zero value, which means that it is set. Alternately, setting the fifth-least significant bit (0-indexed) to 1 can be accomplished using `bf |= (1 << 5)` and the fourth to 0 (redundant in this example) by using `bf &= inverse(1 << 4)`.

Often, these manipulations are macroed or inlined and hand-assembled to provide readable source-code and processor-efficient implementations.

Additionally, multiplication or division by powers of two are trivially (and efficiently) computed using only bit-shifting. 3×16 can be calculated using `0b00000011 << 4` (since $\log_2 16 = 4$) which is 48.

Bit-functions cannot natively function on floating-point values, so to manipulate those, the memory must be accessed as though it were integral data, using a `byte*` pointer, for example.

3 Design and Methodology

3.1 Overview

The key success criteria for the this implementation are the following:

1. Throughput on the order of 36-40 Gb/s (4.5-5 GiB/s) is achieved by the compressor, and perhaps less-critically the decompressor.
2. Effective compression ratios, comparable to those achieved by `gzip`, `bzip2` and `pbzip2` are achieved, although this is highly unlikely to be achieved using only zero-length encoding.

3.2 Packing algorithm

The compressed data structures is stored as described below in figures 8 and 9:

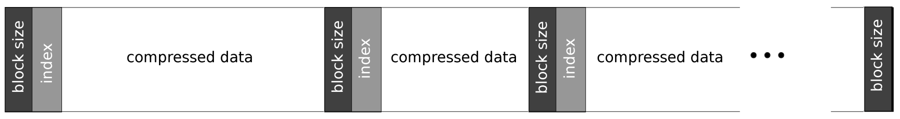


Figure 8: Compressed block layout diagram. The number of blocks in the compressed file and their index ordering is not known ahead of time, but can be inferred by reading all blocks until the end-of-file marker (and guard-block, final right) is encountered.

```
8 bytes: block_size (including headers) or 0x0000000000000000 for guard-block
8 bytes: block_index
block_size elements (1-2 bytes each):
      0x00              (unused)
      0x01 - 0xFE       (literal characters)
      0xFF - C_MARKER   marks a compressed run of zeroes, followed by:
           - 0x00       (an embedded null)
           - 0x01 - 0xFC (length in bytes of a run of zeroes)
           - 0xFD - 0xFF (unused)
```

Figure 9: The format of a compressed data block. There is an increase of 16 bytes (2×8-byte `unsigned long int`) over the raw compressed data, used for ordering the uncompressed data. The ultimate block in a compressed file consists of a single 8-byte block with only `block_size` included and equal to `0x0000000000000000`

The end of file is marked with a guard-block marker (indicating a `block_size` of zero bytes). This allows the decompressor to back-out gracefully in the event that a block which contains the end-of-file marker is read.

3.3 Avoiding the FPU

Data manipulated on the Floating-Point Unit (FPU) is subject to various rounding caveats. There are rounding 5 modes in the IEEE-754–2008 specification

- round to nearest: where ties round to the nearest even digit in the required position (the default mode)
- round to nearest: where ties round away from zero (more commonly used in binary-coded decimal mode)
- round up, toward $+\infty$: negative results thus round toward -0.
- round down, toward $-\infty$: negative results thus round away from -0.
- round toward zero (truncation; it is similar to the common behavior of float-to-integer conversions, which convert -3.9 to -3 and 3.9 to 3)

There are two zero-values, $+0.0$ and -0.0, which are not equal by the IEEE-754 standard. There is also an issue comparing two otherwise equal floating-point numbers

There is also a fast-math mode for FPUs in which hardware does not conform to the IEEE-754–2008 standard, but the instructions are performed natively in hardware. This sacrifices accuracy for speed, which is not a possibility for this project, since a stated goal is to have lossless compression.

There is a representation for "Not-a-Number" ("NaN") where a mathematical operation is not defined ($\frac{f}{0.0}$, for example). These generate an FPU exception which can be either quiet or signaling.[1]. Signaling exceptions should be trapped and remediated (if possible), or the program termination adequately explained, as a good practice.

By treating the read data as characters (or integral-type values), the accuracy of the retrieved data is maintained and the pitfalls mentioned above avoided. Floating-point rounding issues, such as those where calculating an average results in values outside the interval between two numbers[12] may need special case handling. While floating-pojnt instructions can be vectorised, the actual value as a floating-point number is irrelevant to this implementation, as long as it can be accurately reconstructed from the compressed data.

Fast-math mode on x86-based processors is not always IEEE 754–2008 compliant, especially for values which require more bits of accuracy than the x87 co-processor can handle[19, p. 6]. To this end, we will treat all memory buffers containing floating-point numbers as integral, so that compiler enhancements do not potentially taint calculations on these. Since we never operate on the values of the stored data, there can be no algorithmic impact from this methodology.

3.4 Parallelising compression and decompression steps

The data is read in blocks of a pre-defined size from disk. The offset of these blocks in the source data is easily computable, since they are multiples of the block-size and limited above by the size of the file on disk.

The compression algorithm is then run on each block, which may result in data-inflation instead of compression. The compressed output file is then written (serialised) in the order in which independent threads of compression on independent blocks completes. This means that the compressed file will not always be written out exactly the same data order, but since the size of a compressed block is deterministic, the file size should remain constant for a given block size.

The compressed blocks are stored linearly with a header comprised of (compressed) block size and block-offset (from the initial source data). This allows for decompression of the block into the correct offset of the decompressed data regardless of the compressed block's location within the file. Once the decompressed data is written to disk, a cryptographically-secure hashing function is used to ensure that the decompressed data exactly matches the source data. An incorrect ordering of a particular block would result in mismatched file hashes after the compression and decompression routines are run.

Amdahl's Law, as popularised from his 1967 paper [3], claims that the speed-up factor for parallelising an algorithm is limited by the serial (unparallelisable) portion of that algorithm. Thus, if P is the fraction of the algorithm that is parallelisable and $1 - P$ is the fraction that is not, then the potential speed-up factor, S, for n threads is given by

$$S = \frac{1}{(1-P) + \frac{P}{n}} \tag{4}$$

The use of this as a guideline makes sense for this particular application, since the serial fraction of the algorithm is in the critical path for execution, particularly in the case of the decompression algorithm.

We thus need to select which parts of the program to optimise when parallelising. Since the search for zero-runs to compress is the only function in the critical path of the compressor which is parallelisable without impacting performance, it clearly is the place to focus. Since we are treating the raw data as integral bytes rather than IEEE 754 floats, we can use the CPU intrinsics available to operate bit-wise or (as necessary) to efficienctly vectorise logical operations.

3.5 Porting the implementation to CUDA

Due to the limitation of hardware for testing, we will concern ourselves with the Fermi architecture and not the latest Kepler architecture. The CUDA implementation design is intended to be independent of the architecture of the nVIDIA GPU platform. This allows testing to be completed on readily available Fermi-architecture GPUs. Throughput rate scale-up (although not perfectly) for the newer Kepler architecture

The processing architecture of Fermi-generation devices is described in figure 10 and the memory architecture of these in figure 11.

3.5.1 Processor architecture

GPU computing is intended to complement CPU computing and not replace it, since the two approaches are better suited towards diametrically opposed types of tasks: CPUs, for example, perform well with conditional branching, or where data locality is high, whereas GPUs perform well when a single program flow is applied to large numbers of data inputs.

In Fermi, the processor architecture is as shown in figure 10. The architecure consists of one or more Streaming Multiprocessors (SMs) with 32 computational cores each. Each of these cores can perform an integer or floating-point instruction in a clock cycle. They are supported by a hierarchy of memory, with sizes from 16 kibibytes to the a few Gibibytes, but with inverse correlation to access speed (i.e. larger size implies slower access).

A GPU function (or kernel) is compiled to inherently take advantage of the parallelism afforded here. The kernel operates simultaneously on up to 1536 threads, each processing what would be an iteration of a looping construct. If there are more than 1536 elements in a looping construct, the workload can be split across multiple SMs or scheduled once an SM becomes available. Within this block of threads ("thread block"), threads can communicate using shared memory.

The thread block is divided into warps[3] of 32 threads, which are the quantum of work dispatch within each SM. Provided computing and memory resources are available, multiple warps can execute concurrently on a single SM. Some operations refer to a half-warp, which is the first or last batch of 16-threads in a warp.

The Fermi device runs a single program at any given point in time. Switching applications can be accomplished in 25 μS, which allows a single device to seemingly simultaneously compute and render graphics. This is often used in newer gaming engines to facilitate physics simulations while rendering in-game images at a flicker-free refresh rate.

It is important to note that the hardware-level instruction-set differs between general-purpose computing processors and CPU processors. Functions such as Fused Multiply-and-Add (FMA; also known as SAXPY, single-precision $ax + y$) is a single instruction on the GPU architecture, but not on x86-64 based general purpose processors. This strongly correlates to speed-ups from CPU-to-GPU ports for particular types of problems (such as those in the domain of Linear Algebra). There are also cores denoted as special-function units (SFU in figure 10) which calculate trancendental (such as e^x) and trigonometric functions (such as $\sin x$, extensively used in graphics processing) natively in hardware.

[3]From weaving terminology: threads stretched lengthwise in a loom

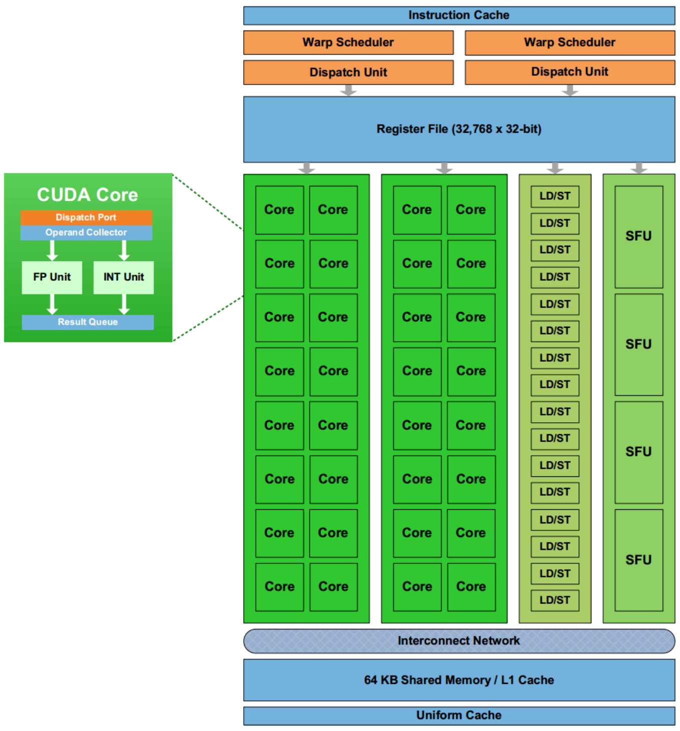


Figure 10: Architecture of the Fermi family of nvidia GPUs [8]. Note the large number of processing cores, as well as the Load/Store units and Special-Function Units.

3.5.2 Memory architecture

The memory architecture for the Fermi class of GPUs is detailed in figure 11.

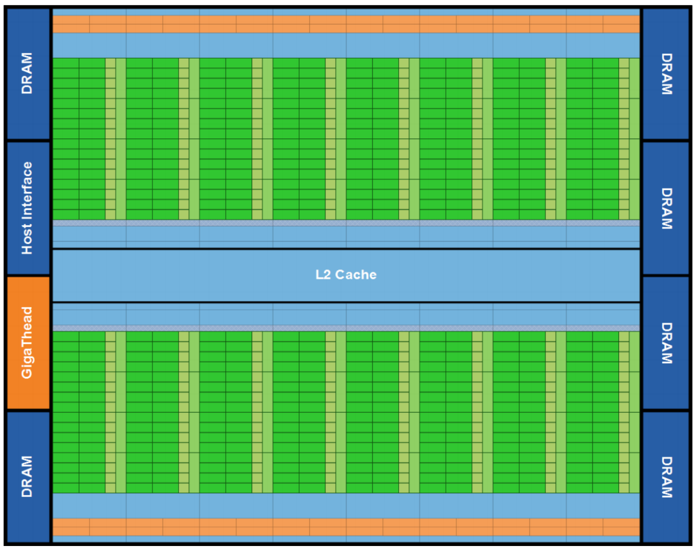


Figure 11: Memory bank layout of the Fermi family of nvidia GPUs [8]. This is expanded outward from figure 10 so that the location of L2 Cache and Global memory (marked DRAM) are visible.

The memory is divided by locality and size. In order of decreasing access speed, they are:

- Registers (fastest, around 8 Tb/s throughput)
- L1 and L2 caches (fast, but slower than registers, around 1.2 Tb/s throughput)
- Shared Memory (about as fast as L1 cache; needs special attention to prevent “bank-conflicts” which slow down kernels if access patterns are aligned to boundaries of powers of 2)
- Constant Memory (kernel read-only; intended for storing values that do not change over the execution of a kernel; as long as all threads in a warp read the same memory location, its speed is comparable to that of shared memory)
- Texture Memory (kernel read-only; used by shaders to store and manipulate images for 3D rendered applications; useful when threads require data which has good spatial locality, such as threads operating on neighbouring pixels in a texture would)
- Global Memory (slowest, but most abundant form of memory; the low speed can be mildly counteracted by using L1 cache and predictable access patterns within kernels, known as coalescing)

Memory is the most complex element to streamline in GPU programming, since an unintended slowdown can result based on a warp of threads’ memory-access patterns. Because all threads in a warp are executing concurrently, a memory exception requiring a read or write for a single thread will result in all other threads halting temporarily while the value is retrieved. This is most evident in branching-heavy kernels, which should be avoided for GPU programming.

With the largest amount of memory being global memory (which has an access latency on the order of hundreds of clock-cycles), wherever possible computation should be done within shared memory (limited to 48 kiB) to obtain any discernible speed-up from GPU programming.

3.6 Sourcing of test data for confirming viability of compression algorithms

Jason Manley at the SKA-SA offices provided test data were obtained from sampled from actual KAT-7 observations. This was comprised of about 167 GiB of HDF5 encapsulated data, of which the raw data comprised about 145 GiB. Four major datasets were used for development and testing. The observation starting times for these data sets are noted as:

- 05/29/2013 20:14:55: (HDF5 size: 15 MiB; raw data size: 12.25 MiB)
- 05/29/2013 18:49:30: (HDF5 size: 859 MiB; raw data size: 756.0 MiB)
- 06/03/2013 17:04:10: (HDF5 size: 2562 MiB; raw data size: 2182.25 MiB)
- 06/02/2013 09:25:14: (HDF5 size: 5060 MiB; raw data size: 4457.25 MiB)

In all figures and tables, the HDF5 size (not the raw data size) is used to label the individual data-sets.

We refer to three distinct representations of the data during the compression and decompression process:

- *Source Data* is the data as retrieved from the correlators before any processing occurs
- *Compressed Data* is the data which has run through the compression routine, but not the decompression routine; and
- *Decompressed Data* is the data output from the decompression routine. It must match (bit-wise) the Source Data for the compression to be considered successful.

After a full compressor/decompressor run, a cryptographically secure hash function is used to ensure that there are no differences between the uncompressed and decompressed data. Since these often operate block-wise (especially MD5 and the SHA-1 and SHA-2 families, which are based on the Merkle-Damgård construction), they are a natural fit for block-based data.

3.7 Structure and scope of the benchmarks

This report will focus on the following analyses:

1. A comparison between the a CPU implementation of the naive and the parallelised approach will be made.
2. A detailed breakdown of how the algorithm performs on different numbers of cores will be provided.
3. The feasibility of implementation using the Intel SSE and AMD XOP (if possible) vectorised instruction sets will be investigated.
4. Benchmarks of the algorithm against the standard utilities used for compression. This yields a comparison of throughput and compression ratio: `gzip`, `bzip2` and `pbzip2`. The disk I/O will be included in this case, in the interest of a balanced investigation.
5. The feasibility of a CUDA implementation, as outlined earlier will be determined.
6. Finally results from concurrent research being conducted into arithmetic and predictive methods will be compared to those achieved in this investigation. Aa breakdown of performance in terms of both throughput and compression ratios will be included.

3.8 Benchmarking platform

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

- It should be as *decoupled* as possible. This will assist in comparing several implementations of the CPU version and being able to invisibly replace implementations. Components should not depend on nuances of implementations.

- The components must be *cohesive*. They should perform only necessary functions and should be treated independent and atomic, in line with the UNIX philosophy.

Figure 12 describes the architecture for the benchmarking platform. More detailed implementation details will be provided in the Implementation chapter.

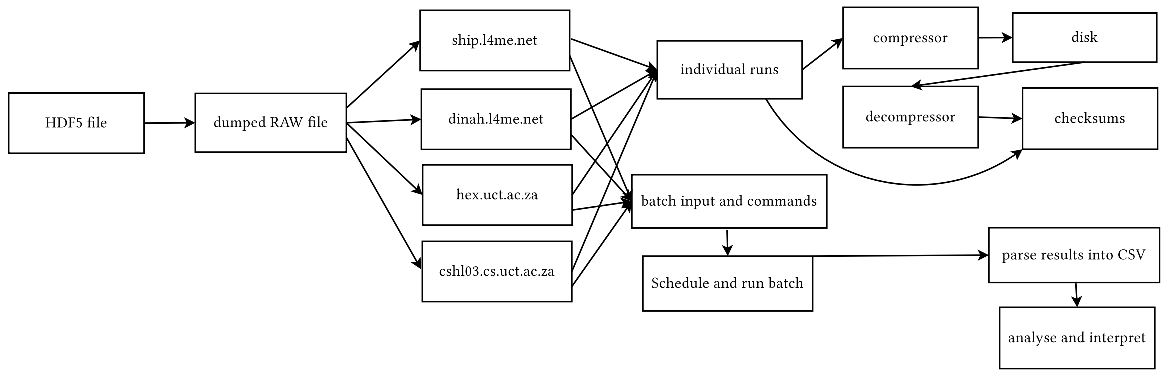


Figure 12: Benchmarking platform architecture

3.9 Testing platform: HEX.UCT.AC.ZA

While the hostname HEX.UCT.AC.ZA actually maps to the head-node (or job-management node with little computing power) for the UCT ICTS high-performance computing (HPC) cluster, it has become synonymous with the entire HPC cluster for both the CPU and GPU workloads. The operating system for the cluster is Linux Standard Base 4.0, which confirms to the POSIX specification and the Single UNIX Specification, amongst others.

The CPU computing cluster has 4 DELL C6145 nodes, each with four 16-core AMD Opteron 6274 CPUs, clocked at 2200 MHz with 16 MiB L3 cache memory per die. Each machine has 128 GiB of primary memory and is attached to a networked storage subsystem so that jobs and data can be uniformly mounted across the cluster.

The GPU cluster's two nodes have dual hex-core Intel Xeon E5-2620 CPUs, clocked at 2000 MHz with 15MB Cache, and each have 4 NVIDIA Tesla M2090 GPU cards (Fermi architecture) installed. The machines have 64 GiB of primary memory and each GPU card has 6 GiB of GDDR5 memory onboard and 2048 CUDA cores running at 1.3 GHz. This provides a combined output (over the GPU cluster) of around 10.64 PetaFLOPS.

3.10 Other machines used for testing

Two other machines were used for development and testing of the solutions:

- CSHL03.cs.uct.ac.za, with an Intel Core i7-3770 processor at 3.4 GHz, GK104 based NVIDIA GeForce GTX 670 card (Kepler architecture), 8 GiB primary memory and 900 GB traditional storage;
- SHIP, with an Intel Core i7-2720QM processor at 2.8 GHz, GF119M based NVIDIA Quadro NVS 4200M card, 16 GiB primary memory and 256 GB SSD storage; and
- DINAH, with an Intel Core 2 Q8400 processor at 2.6 GHz, GF106 based NVIDIA GeForce GTS450-OC card, 6 GiB primary memory and 1.5 TB traditional storage

The primary development operating system is Ubuntu 12.10 (x64), where tests were run on the standard Ubuntu Linux kernel 3.5.0-41 and software was compiled with GCC 4.7.1 and the CUDA 5.5 Toolkit.

These comprise the primary development and testing environment, with the tests for comparisons run on the UCT ICTS HPC Cluster detailed above.

4 Implementation

4.1 Basic algorithm

In order to give a more detailed description to the algorithm mentionedd in the previous section, the pseudo code for the block-wise zero-length compression algorithm is given below:

```
begin offset counter at 16
get block
    while characters available in block
    if char is C_MARKER,
          write FF00 to output
          advance input counter 1
          advance output counter 2
          proceed to next input char

    if char is ZERO
          for each zero-character in block,
              count and find first non-zero element.
          If run_length > 2
              write (FF,run_length) to output
              advance input counter loop times
              advance output counter 2 times
              proceed to next input char
          else
              write individual zeroes to output for runs less than 3
              advance input counter loop times
              advance output counter number of zeroes times
              proceed to next input char

    else
          write char to output
          advance input counter
          advance output counter
          proceed to next input char

  write length of compressed block to output position 0
  write index  of compressed block to output position 8
```

Figure 13: The compression routine in Pseudocode. Encountered zeroes which do not form part of a long-enough run are the most problematic case, since each number of consecutive zeroes needs to be handled individually.

Note that we only encode runs of length greater-than or equal to three bytes, since the disk-space cost of embedding a run is an additional byte and we would prefer run-lengths of length two to remain naïve for simpler (and faster) decompression.

Decompression follows analogously, with the caveat that the compressed file may not have the blocks written in numeric order:

```
get block
    for each character in input block
    if C_MARKER
        check for and write embedded null
            advance input counter by two
            advance output counter by one
        else
            check for and write run of zeroes
            advance input counter by two
            advance output counter by run_length
    else
        write input to output
        advance input  counter
        advance output counter
```

Figure 14: The decompression routine in Pseudocode

Since the data-sets are often too large to hold source, compressed and decompressed copies in memory, primary testing was completed on the 15 MiB dataset, with the 859 MiB dataset used for secondary verification of results before a test battery.

Boundary testing was accomplished using additional data-sets crafted with a hex editor. This also afforded an opportunity to compare the expected compressed outputs to ensure that all cases produced consistent results.

By selectively disabling routines (using compiler preprocessor directives), unit testing could also occur. A notable output from this phase is that certain compressed input may not be generated by the compressor, although the data described will be emitted by the decompressor in the correct, and expected, manner (specifically, encoded zero-runs of length 1, for example).

4.2 Alternative schemes considered

Initially a bit-based approach was pursued. This requires more CPU operations than a byte-based approach and could not embed run lengths without excessive data inflation, eliminating any compression benefit. Comparing bytes in assemble is efficient on the x86 architecture (three instructions) and was pursued instead, as confirmed by ANDERSON in [4, p. 26].

Storing a map of compressed regions which denote viable compressible blocks and storing this as a dictionary instead of inline markers and run-lengths was also pursued in the GPU-based approach. This is largely due to the possible benefit of only copying the dictionary back from the device.

4.3 Algorithm and data parallelisation

The algorithm falls into a class which is known as "embarassingly" or "pleasingly" parallel, since every individual data block cam be compressed on an independent processor as long as the input and output memory buffers can be accessed by that thread.

The buffer-block construction may, however, break a run of zeroes at the boundaries of these blocks, which has a negative impact on the compression ratio. In these cases, there are two possibilities:

1. Combine the adjacent runs to achieve better compression; or
2. Leave the multiple runs in place to achieve a stable block size.

Since the decompressor needs to allocate memory for a particular block prior to the decompression routine running, the first possibility would serialise the both writing of the compressed data and the decompression process at the block-boundaries and beyond the boundary of any (yet) uncompressed block when reading. It would also introduce inter-thread synchronisation complexities, which would decrease the throughput of each process thread (while waiting for inter-thread communication).

The second approach (using a stable block size) is the one pursued. This enables simplified parallel decompression, since the output memory offset is computable at compressed block-read time (where the offset and block-size are known).

4.4 Feasibility of including SSE and XOP intrinsics

The most processor intensive (and parallelisable) portion of the compression routine is determining the existence and length of a run of zeroes. By vectorising the loop, SSE2 intrinsics can compare 16 8-bit values simultaneously to another, possible zero, vector. This construct is known as an intrinsic and is converted to machine-code by the compiler, although aesthetically and programmatically it is superficially similar to a library call.

The use of this structure affords early-exit opportunities from the loop for incompressible portions of the input file, since the number of consecutive zeroes can be easily determined from the intrinsic output data-structure. The copy of the data can then be more efficiently done through a function/intrinsic call (to `memcpy()`) instead of individual byte assignments.

Similarly, the decompressor (naïvely) needs to compare each byte to find the `C_MARKER` (or run-substitution delimiter) bytes indicating zeroes and the substitution character. If the vector returned from the intrinsic is the zero vector, the entire block may be copied to the output without further investigation, since no expansion of the input would occur.

Analogous to the Intel-designed intrinsics are AMD-specific intrinsic extensions. These are named XOP (eXtended Operations) and are described in [10]. Where sensible, these can be used to complement the SSE instructions for improved performance on AMD Opteron-based platforms.

An important thing to consider is the alignment of local variables, since CPU intrinsics require that they be aligned to the size of the CPU registers to be used in the intrinsic calls. Since we were using SSE2 intrinsics, the alignment offset was 16 bytes (or 128 bits). This requires extends to their (byte-)offset in memory upon allocation, since the intrinsics can only access elements whose size and offset can be evenly divided by the designated register's size.

Willhalm, et al.[20] describes methods for vectorising lightweight compression (which zero-length encoding is considered to be a part of) on SAP databases.

4.5 GPGPU implementation in CUDA

Since we can massively parallelise the search for runs per byte (with early termination per byte eliminating the branches with terminated zero-runs), we take the following approach for compression, launching a kernel for each byte offset:

```
for each block
    read block from disk to host memory
    copy block to device
    <<<kernel_call>>> find runs of length >= 3
    <<<kernel_call>>> update (atomically) runs with tid (offset) and length
    copy back map<runs>,number_of_runs to host
    for n streams
        cudaMemcpyAsync (run[n], run[n+1]-1) // if required
    !! ensure that n+1 indicates end of block (from block size)
    write to disk number_of_runs,map<runs>,blocks (from host memory) (serialised)
```

Figure 15: The compression routine for CUDA in Pseudocode. The blocks written to disk are already in host memory, but for benchmarking purposes we copy the memory back.

This approach lends itself well to the block-based parallelisation approach pursued in the CPU implementation, with the caveat that shared memory reads will result in bank conflicts.

Since the `cudaMemcpy()` operations can be asynchronous across multiple streams (and therefore the ordering of the constituent memory block components is not deterministic), great care must be taken to serialise the write to disk. With the number of runs and the size of the structure storing these known (before the async `cudaMemcpy()`s), the structure can be written to disk with no risk.

Decompression will follow the following steps:

```
for each block
    copy the compressed structure to the device from host memory
    for each run
        cudaMemset() (to nullify the compressed runs)
    for each run/n streams
        do a cudaMemcpy() (D2D) of the uncompressed regions per stream
```

Figure 16: The decompression routine for CUDA in Pseudocode. Since the data is generally intact (apart from zero-runs) the Device to Device copy will be fastest.

4.6 Constraints on benchmarking environment

To exclude the influence of the external testing environment (which would likely be performing unrelated tasks to the test-cases) the following conditions are specified for the test runs:

1. Results for files in excess of 500 MiB will be recorded for benchmarking. This will stabilise the algorithm timings, since initial data loading (i.e. algorithm "warmup") costs become less obvious here. Additionally, at least one data set over 4 GiB will be used to ensure 64-bit algorithm parity above the 32-bit processing boundary (2^{32} bytes).
2. Memory swapping and task switching on the host for testing should be minimised. Wherever possible, a host for testing should be dedicated to the compression and decompression routines.

4.7 Distribution of compressible areas in the obtained data-sets

From casual data inspection, it is noted that the compressible areas are spread uniformly throughout the data sets, with an average zero run-length of about 4 bytes and are not concentrated in specific blocks.

RAW data files are generated from the HDF5 sources using the following tool from the `hdfutils` Linux package:

```
h5dump -b LE -d /Data/correlator_data -o «output_file».dump «input_file».h5
```

Further analysis of the raw data extracted from the HDF5 files shows that the positive zero (four consecutive zero bytes) is categorically the most frequently occurring length of a run by several orders of magnitude in all sampled data files. This explains the low compressibility of the data using Zero-Length Encoding, since these runs are only reduced by half the size of their expanded representation.

5 Results and Discussion

5.1 Parallelisation approach

The performance of the block-based parallelisation scheme scales well, but not sub-linearly with the number of processing cores, as long as disk I/O is excluded (see figure 21.

The biggest issue with a block-based approach is serialisation of disk-writes, regardless of the ordering of blocks, since the compressed data cannot be interspersed on disk in order to be decompressible. An alternative scheme would be to write to separate files, although the thread_id would need to be assured of being unique throughout the data run to prevent unintended overwrites of another block's data upon decompression. This would, again, introduce inter-thread synchronisation issues updating a dictionary of (block,file_on_disk) mappings.

5.2 Vectorised intrinsics

Using intrinsics on the blocks seemed to slow down the compression algorithm instead of speed it up. This may be as a result of compiler optimisation (`gcc ... -O3 ...`) automatically applying the intrinsics and the manual inlining of the functions interacting with that optimisation. Compilation with `g++ ... -no-vec ...` shows a noticeable slowdown, so this is assumed to be the cause of the performance increase. A debug flag exists within the source-code to toggle this behaviour: `CONST_ROUTINE_EQUALS_INTR=1`

The main benefit of the intrinsified search is the early exit (and improved copy) of a vector of bytes which contains insufficient compressible bytes. These can then be copied using two intrinsic calls or a `memcpy()` call if they are not aligned to SIMD-register-sized byte boundaries.

5.3 Compression

An average compression ratio of 0.97 was obtained across the data-sets inspected at a throughput rate of 0.4-0.6 Gb/s per core in the vectorised CPU implementation. As can be seen in figure 17, the speeds far exceed those of general compression implementations, as was expected at the onset of the project.

Memory allocation was within the range of the general compression implementations for the cases tested (since a compressed block may expand to over double the size of the source data). Only `pbzip2` used comparatively inordinate amounts of memory (two decimal orders of magnitude more than any other algorithm.)

5.4 Decompression

Decompression speeds for ZLE mirror the compression speeds, since the size of a block of output memory is known ahead of time and only one block per thread needs to be allocated. Since all threads hand similar-sized blocks (and the master thread is simply coordinating assignment of the workload), there is no discernible bottleneck as the number of cores scales. It should be noted that memory allocation per thread would scale linearly with the number of cores, as in the case of compression.

5.5 Benchmarks against general compression utilities

For comparing various algorithms, we firstly ensured that the compressor was set to use the same number of threads as the general compressors (a single thread for `gzip` and `bzip2`; the number of cores present in the system for `pbzip2`).

The UNIX command `time` was used to view "wall-clock" timing for each process. While it should be noted that it is not monotonic and is susceptible to clock-drift and influence from synchronisation services such as (S)NTP, this was mitigated as far as possible during testing through multiple runs of the algorithm. Memory usage was obtained through polling the process list (the `ps` command) and recording the maximum value for each run. A script for automating this was obtained from the URL in the footnote[4] and will be included before the references.

Running against `gzip`, `bzip2` and `pbzip2` shows that the speed is better overall (since it is a more primitive algorithm) but the compression ratios are much worse compared to the standard algorithms. Figure 17 shows the magnitude of the speed/ratio trade-off:

	Time (s)	Peak Memory (KiB)	Ratio	Throughput (Mb/s)
bzip2	115.110	1436	0.499	55.09
gzip	32.009	620	0.643	198.13
pbzip2	22.43	30120	0.501	282.65
ZLE	3.74	3548	0.966	1693.78

Figure 17: Compression ratios against general compression algorithms (`gzip`, `bzip2`, `pbzip2`) on the 859 MiB data-set using 4-cores. Note the memory usage of `pbzip2`, which shows a 20-fold increase over `bzip2`, but fails to absolutely match its compression ratio. All timings and throughput values **include** disk I/O time (although the file was cached in memory by the Operating System after the initial warm-up runs).

`bzip2` uses a block size of between 100 kiB and 900 kiB (selectable through the command line switches `-1` through `-9`). This results in a stable memory allocation pattern, dependent only on the number of threads processing the source data (since each thread requires a separate memory allocation to store the uncompressed and compressed blocks).

According to the UNIX manual or "`man`" page for `bzip2`, this is $400\text{ kiB} + (8 \times \text{blocksize})$ for compression and $100\text{ kiB} + (4 \times \text{blocksize})$ or $100\text{ kiB} + (2.5 \times \text{blocksize})$ in a slower, but less memory intensive implementation which is selectable through a command-line switch (`bzip2 -s`).

5.6 Data analysis

By inspecting the raw data sources, we can determine whether elements of the algorithm can be improved, such as the character chosen to compress (nominally zero) or the marker chosen which would inflate. By examining the compressed data, we can determine which lengths of runs are most frequent.

[4]Process Maximum Memory checking script (memusg): `https://gist.github.com/netj/526585`

5.6.1 Character frequency distribution

Scanning the raw data and quantizing byte-aligned values as characters yields the character frequencies as in figure 18:

	15 MiB	859 MiB	2562 MiB	5060 MiB
Avg# occ. per non-zero char	46991	2887811	8379264	17048859
Total zero count	826465	56331641	151542748	326306261
Min# occurences (char)	21775 (`0x7C`)	352849 (`0x71`)	4321374 (`0xE8`)	6455150 (`0xFF`)

Figure 18: Frequencies of characters in raw input data. Zero bytes occur about 18-19 times more frequently than the average non-zero character.

It is clear that zeroes are the most frequently occurring byte, occurring 18-19 times more frequently on average across the data files than other characters.

As we will later see in figure 19, the most frequent run length is 4, indicating an IEEE 754−2008 binary32 positive zero value. Within the larger data sets, at least one run of 185 consecutive zero bytes was noted. This may correlate to a "blowout" (an abnormally strong signal, suggesting terrestrial-based interference) and its system-internal remediation, which would be to discard the observed values and emit zeroes into the output stream instead.

The least-frequently occurring character differs per data set. The knowledge of this character is important, since using it as a "run-marker" to indicate a sequence of zero bytes following would result in the best compression ratio. Embedding the run-marker into the compressed stream therefore inflates (doubles) the compressed file size increment for those bytes. Without loss of generality (and since it is not possible to know the least frequent character ahead of the compression run), we have used the character `0xFF` (ASCII `NBSP`; non-breaking space) in all implementations and tests, unless otherwise noted.

5.6.2 Zero-run lengths

As figure 19 shows, the most frequent run-length for zeroes is 4 bytes. The inflation from embedding nulls amounts to much less than 1% of the uncompressed data size, so this inflation is completely negated by the length of the runs of even 4 consecutive zeroes, which averages 1.59% of the uncompressed data size. If there were a more efficient way of encoding this (so that the input alphabet were only robbed of a marker and a run length of 4 could be denoted in a special manner), the compression gain for this special case would roughly double at the computational, and therefore throughput, expense of additional branching and handling for this specific case.

	15 MiB	859 MiB	2562 MiB	5060 MiB
`MARKER`	30325 (-7.95%)	421886 (-1.56%)	5816415 (-8.74%)	0 (-)
3	33 (0.01%)	135127 (0.50%)	2377 (0.004%)	198165 (0.13%)
4	201319 (106.55%)	25441706 (94.27%)	71948966 (108.05%)	150229168 (98.20%)
5	2211 (0.58%)	88794 (0.33%)	390942 (0.59%)	626610 (0.41%)
6	2788 (0.73%)	173568 (0.64%)	18008 (0.03%)	1339516 (0.88%)
7	0 (-)	10640 (0.04%)	240 (0.00%)	14345 (0.01%)
8	300 (0.08%)	480306 (1.78%)	43236 (0.07%)	559212 (0.37%)

Figure 19: Frequency of run-lengths of length 8 and below and their contribution towards compression savings (occurrences×(input size - output size)). The `C_MARKER (0xFF)` character, which inflates file size, is also included. Run lengths over 8 characters do exist, but generally do not meaningfully contribute to compression ratios. Curiously, the last file contains no embedded characters matching `C_MARKER`.

5.7 Feasibility of a CUDA implementation

While the runs of zeroes can be trivially identified and catalogued at the required speeds, the data still have to be recomposed into a compressed format and transferred to disk at comparable speeds. Given results from concurrent testing on the hardware available, memory throughput has been a notable bottleneck. If the claimed InfiniBand storage throughput speeds can be achieved in practice, this bottleneck can be dispelled once nVIDIA GPUDirect becomes a feasible approach for writing data to disk.

Ultimately the CUDA implementation for zero-length encoding was not finalised (owing to thread synchronisation issues), although insight into a potential upper bound for processing speed was gleaned. Since additional synchronisation would reduce the potential throughput,

The kernel throughput for finding the runs is on the order of 12-13 Gb/s per stream (when run on 1024 threads per block). Once memory copy (before writing to disk) is included, this drops to around 1 Gb/s.

We therefore concentrate (theoretically) on information from results in external research when discussing CUDA implementations of zero-length or run-length encoding.

5.8 Adjusting per-core block-size

There is little effect on throughput as the per-core block-size is adjusted. Even using "jumbo"-sized blocks of 64 MiB does not show a marked increase nor decrease in throughput. With smaller block-sizes (less than about 4 MiB), the probability of truncating a run of zeroes at the end of a block becomes higher, and this inflates the lower-bound for file-size. Figure 20 shows the effect of adjusting the block-size.

The block-sizes should always remain a factor of 4 bytes in order to fully encode an IEEE 754–2008 binary32 within a single block. To ensure intrinsics are able to function, this should be a multiple of 16 bytes (if SSE2 is used), ranging up to 64 bytes (if AVX-512 is used and intrinsics are extended to compensate for the possible longer run lengths).

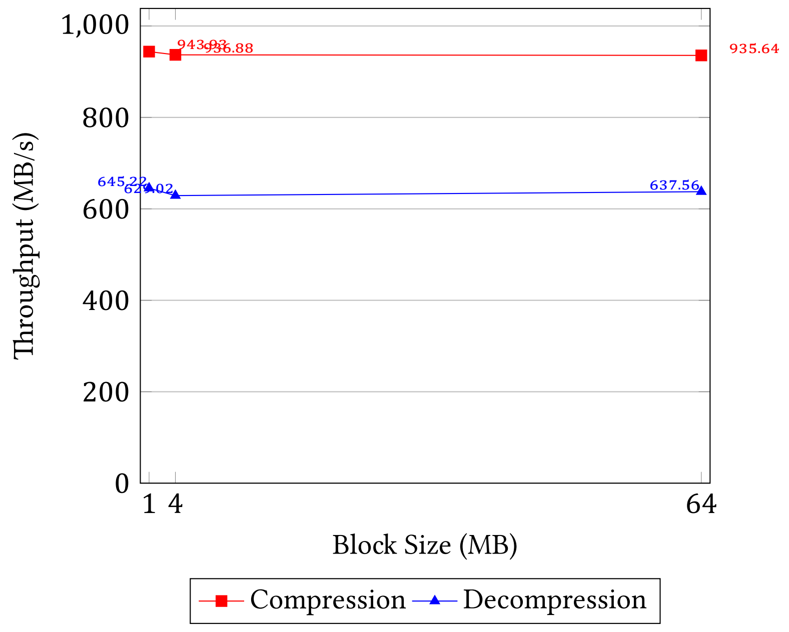


Figure 20: Impact of block-size on compression throughput

According to the UNIX "man" page on `bzip2`, there are diminishing returns on increasing it beyond 100-900 kiB, which is consistent with the results we've obtained.

5.9 Scaling number of cores

The scaling dynamics of the algorithm can be seen in figure 21, where the speed increase is not quite linear, but rather a factor smaller than 1 of the speed. Naturally the scaling is limited by the speed of the input stream, which in the intended environment would be 36-40 Gb/s (either from network or disk array). The read speeds from disk are excluded from these benchmarks.

Threading overhead is introduced in the form of memory allocations for the input and compressed data for each of the individual running threads. This means that memory scales linearly with the number of cores, however given the block sizes chosen (maximum 64 MiB for these tests), the overhead is negligible.

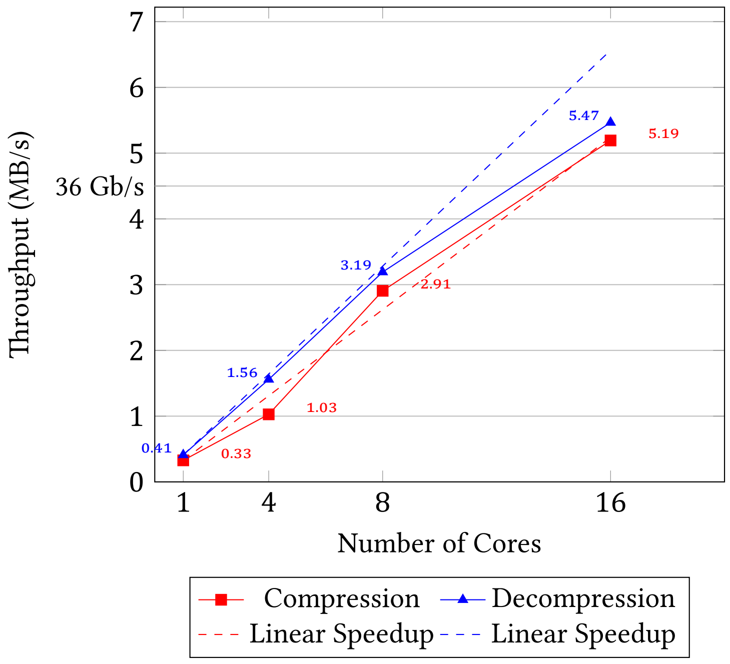


Figure 21: Impact of number of processor cores on compression throughput. We see that the outcome approximately linear.

5.10 Impact of disk I/O on throughput

When including the disk I/O, the throughput rates are predictably capped at the disk read/write rate, as seen in figure 22. The slow-down is equally noticeable regardless of the number of cores assigned to the compression process.

As the number of running threads increases, the OS disk-scheduler needs to queue more writes to the disk. Given native command queuing (NCQ) present in the SATA and SAS specifications, which reduces the waiting time of the system call to write to disk by offloading the system-call to the disk controller, and a fast disk subsystem, as in the machine SHIP which has a solid-state disk, the difference between a single-core and multiple cores (up to 16) is largely negligible. Once that buffer fills, however, additional requests to write to disk would block and the compression throughput would be decimated. This would be particularly evident on a general purpose disk-array once the disk-write cache is exhausted either by the compressor or other applications using it.

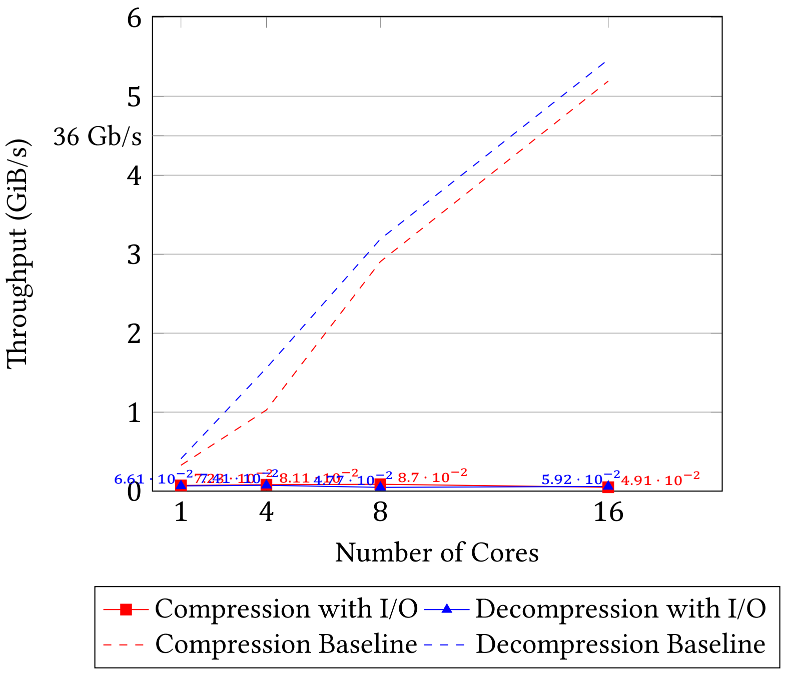


Figure 22: Impact of disk I/O on algorithm throughput. We note that throughput reduces by a factor of about 10^3 if disk access times are included.

5.11 General Observations

Two optimized results implemented by our team achieved the required speed and are thus considered potentially suitable for use within the data-processing pipeline. For offline storage, the arithmetic methods provide the most compressibility, even exceeding that of the standard `gzip` and `bzip2` algorithms for the SKA specific data sets. This is largely due to the small variation (approximately 9000 unique values, regardless of data-set size) of loating-point values in the data-sets.

The most promising algorithm for speed is Run-Length Encoding, which scales approximately linearly with the number of cores available (excluding secondary storage I/O).

5.12 Comparison to concurrent research

A lot of work regarding zero- and run-length encoding is database-related. Roth and Van Horn[15] have noted that run-length (and by association, zero-length) encoding impose a penalty on the reading of compressed table-spaces[2] if used on data with insufficient compressibility. A counter-example to this norm is the reduction in buffer-faults requiring a system call for disk I/O, as noted in Graefe and Shapiro[9].

While bit-twiddling was initially embarked upon, the number of operations outweighs the cost of comparing two bytes (which constitutes three x86 assembly instructions). Given the data-set sizes, this is a reasonable tradeoff, since calculating and aligning bits before allocating sufficient memory would require a full scan of the data prior to decompression. At present, a seek of the current compressed block-size is done to parallelise decompression, but this is only a read of 16 bytes for every block, which may have arbitrary size less than 2^{64} bytes, and not a full scan of the compressed data-file.

Internal to our team's investigations, we have the results for optimised implementations of all the intended compression algorithm classes.

Brandon Talbot's *Arithmetic Methods* results show good compression rates (ratio: 0.41) at the expense of speed, improving upon even `bzip2` for this particular data-set type. This would serve particularly well for storage of the data, since the HDF5 exports seen inflate the data considerably (approximately 11-12% of the file size is meta-data describing the data-strides in each HDF5 file), but at the current throughput rates it may not be feasible (since data has to remain in primary memory until written to disk). As expected, the decompression is completely linear and shows low output speeds of around 30 Mb/s, although this may be attributable to the disk-array access speed of the test-bed.

Benjamin Hugo's *Predictive Methods* results show slightly improved compression rates (ratio: 0.93) over Zero-Length Encoding, but the speed is not within the range required for real-time compression. The CUDA implementation's kernel shows promise at over 13 Gb/s so that it may be viable (with scaling over multiple GPUs) if the input stream can be adequately multiplexed.

Both these CUDA implementations suffer from memory-copying latency issues, however, which negate any speed improvement of the massively parallelised (CUDA) implementations. Even the hybrid CPU/GPU approach suggested for arithmetic methods do not provide speeds within the required range of around 36-40 Gb/s. While this may be attributable to hardware configuration, a sufficient variety of configurations was used for testing and none gave favourable results for the memory-copying phase.

Figure 23 summarises the outcomes of the parallel-running sub-projects.

6 Conclusion

We set out to investigate the possibility of an online (streaming) compressor capable of data-rates approaching 36-40 Gb/s, to be used for compression of data originating from the SKA arrays of radio-telescopes.

The following questions were posed at the onset:

1. Can we achieve effective throughput rates of at least 40 Gb/s (data line-rate; 36 Gb/s nominal data throughput)?

   As we have seen, this is certainly attainable for zero-length and predictive compression methods, but not for arithmetic methods, as initially estimated. The required amount of processing to produce sufficient output seems to be around 16 cores, each running around the 2 GHz level.

2. Can we reduce the storage requirement significantly by using data-compression?

   While promising results were obtained for arithmetic coding (again, as imagined with the caveat of the low throughput), the storage requirement can be exceeded. However, for zero-length encoding, the frequency of zeroes in a pattern that allows real-time decompression does not have the character frequency required for reductions on the order of 10%. At best, an estimate of around 94% may be possible, given the character distribution across the sampled sets.

3. Can throughput rates be traded for compression ratio by using different algorithms? If there is enough gain from the compression ratio, this may be traded for throughput if this is not sufficiently fast.

   Given the costs of obtaining the throughput at the required rate, this was not thoroughly investigated. As mentioned in the section on character frequencies, additional cores which are amenable to branching would result in more special cases being considered. These would correspond to symbols common as 8-bit-aligned byte-length strings inherent in IEEE 754–2008 floating-point numbers. Since the most obvious case is already handled, and the frequencies of other characters is mostly equi-distributed, there may be little value in increasing the branching level, since the sampled data is likely to change with more correlations and between astronomical observances.

While the naïve and parallel implementations respectively showed promise and attained their goal, the GPU-optimised version still needs to be proven viable, given the intricacies of moving data at consistent rates of gigabits or terabits per second.

The block-based parallelisation method would seem to scale well, provided adequate speed of access to memory is available. As long as disk I/O (which is inherently serial) is excluded, the compressor and decompressor would scale linearly in terms of memory required to compress larger amounts of data. Most importantly, the rate required can be achieved with about 16 cores.

Since runs of length longer than 4 are rare, the block-size has little impact on the compression ratio (since it would only negatively affect that if runs were truncated at block boundaries). The additional compression headers, however become notable at small block sizes, as the number of blocks increases. It would also have an impact on decompression, given that at least one read (16 bytes) and one seek is required per compressed block processed.

Given the instructions and structures necessary to efficiently encode runs, bytes are a good choice as a boundary for compressing this data. This leads to more maintainable code, since inlining of bit-manipulation functions does not need to be considered.

The choice of `C_MARKER` is arbitrary, since there seems to be no consistency in terms of least frequent character. The zero-byte, however, occurs routinely with a much higher frequency than random data, and so is a good choice for run-length detection.

Intrinsification of the zero-search function provides great improvements in throughput, particularly with the fast-exit capabilities afforded by combining 16, 32 or 64 elements and searching for zeroes. Extension of this to larger SIMD instructions (as they become available) should comparably improve performance. Performance improvements were noted even with just aligned memory declarations for the memory buffers.

Both zero-length and predictive encoding methods would therefore be suitable for reducing congestion on the network and disk subsystems, but are unlikely to provide long-term storage benefit. The latter requirement would be well addressed by the arithmetic methods, should that be a consideration. In particular, HDF5 as an interchange format with raw embedded data seems a wasteful solution for long term storage, since inflation of roughly 13% over the raw data size was observed.

7 Future Avenues of Research

7.1 Using nvidia GPUDirect

nvidia has a commercial product GPUDirect[5], which allows Remote Direct-Memory Access, enabling passing memory directly between devices and any of their PCIe neighbours. This has been used by Shainer, et al.[17] and Kim and Lee[13] to minimise the overhead of memory transfers through the CPU to a neighbouring GPU by writing directly across the PCIe bus to the other device, nominally a GPU attached to the sender's PCIe root-bridge.

If this can be extended to disk-array manufacturers (as seems plausible from the FAQ on the nvidia product page), the compression and decompression routines can then be modified to write to disk directly from the device memory, although this will still have to be serialised to prevent the race condition discussed in the Implementation chapter (ch. 4).

7.2 Intel MIC

Intel have presented a different approach to massive parallelisation through their Intel MIC ("Many Integrated Cores"; pronounced "Mike") architecture. The Intel Xeon Phi is manufactured as an add-in card with up to 61 processing cores running at 1.238 GHz (with a potential boost of up to 1.333 GHz) on-board[6], of which multiple units can be installed given sufficient expansion slots on a mainboard.

This means that the reasonably successful standard x86-64 architecture programming methods can be applied. Full performance on this platform is only not dependent on heavy SIMD utilisation, as shown by Cramer, et al.[5]. Given the purported memory bandwidth of 156 GB/s per co-processor, this should easily attain the 36-40 Gb/s requirement for feasibility.

7.3 NUMA/MPI/BeoWulf clustering

Non-Uniform Memory Architecture (NUMA) is a method of parallelising tasks by sharing memory and classifying it as either local or remote. This allows a processor to access more memory at the expense of speed. This memory would be classed between primary memory (RAM) and secondary storage (disk array) in terms of speed.

The most common implementation of this is the use of BeoWulf clustering, which is a generic term used to describe a network of commodity-grade machines used in a parallel fashion[7]. Often BeoWulf clusters use OpenMPI as the messaging interface, which is the same message-passing interface used on the hex.uct.ac.za computing cluster.

Given the required data throughput, it may be convenient to parallelise the data stream from the correlators to various machines on a fast multi-gigabit backbone and then process the results on those. Given the current architectural limitations on the PCIe bus, this technology is not viable at present (since the fastest commercially available PCIe backbone runs at 32Gb/s), but should the bandwidth increase considerably before the next phase of the SKA project, this is worth considering.

[5]https://developer.nvidia.com/GPUDirect

[6]http://ark.intel.com/products/75799/Intel-Xeon-Phi-Coprocessor-7120P-16GB-1_238-GHz-61-core

Code Listing

Script for determining memory usage (`memusg`)

Listing 1: memusg script (source: `https://gist.github.com/netj/526585`)

```bash
#!/usr/bin/env bash
# memusg -- Measure memory usage of processes
# Usage: memusg COMMAND [ARGS]...
#
# Author: Jaeho Shin <netj@sparcs.org>
# Created: 2010-08-16
set -um

# check input
[ $# -gt 0 ] || { sed -n '2,/^#$/ s/^# //p' <"$0"; exit 1; }

# TODO support more options: peak, footprint, sampling rate, etc.

pgid=`ps -o pgid= $$`
# make sure we're in a separate process group
if [ $pgid = $(ps -o pgid= $(ps -o ppid= $$)) ]; then
cmd=
set -- "$0" "$@"
for a; do cmd+="'${a//"'"/"'\\''"}' "; done
exec bash -i -c "$cmd"
fi

# detect operating system and prepare measurement
case `uname` in
Darwin|*BSD) sizes() { /bin/ps -o rss= -g $1; } ;;
Linux) sizes() { /bin/ps -o rss= -$1; } ;;
*) echo "`uname`: unsupported operating system" >&2; exit 2 ;;
esac

# monitor the memory usage in the background.
(
peak=0
while sizes=`sizes $pgid`
do
set -- $sizes
sample=$((${@/#/+}))
let peak="sample > peak ? sample : peak"
sleep 0.1
done
echo "memusg: peak=$peak" >&2
) &
monpid=$!


# run the given command
exec "$@"
```

References

[1] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008* (2008), 1–70.

[2] Abadi, D., Madden, S., and Ferreira, M. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data* (2006), ACM, pp. 671–682.

[3] Amdahl, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference* (1967), ACM, pp. 483–485.

[4] Anderson, S. E. Bit twiddling hacks. *CS Stanford,[Online]. Available: http://graphics. stanford. edu/˜ seander/bithacks. html# CountBitsSetTable.[Accessed 4 May 2012]* (2005).

[5] Cramer, T., Schmidl, D., Klemm, M., and an Mey, D. OpenMP Programming on Intel R Xeon Phi TM Coprocessors: An Early Performance Comparison.

[6] Deutsch, L. P. DEFLATE compressed data format specification version 1.3.

[7] Dmitruk, P., Wang, L.-P., Matthaeus, W., Zhang, R., and Seckel, D. Scalable parallel FFT for spectral simulations on a Beowulf cluster. *Parallel Computing 27*, 14 (2001), 1921–1936.

[8] Glaskowsky, P. N. NVIDIA's Fermi: the first complete GPU computing architecture. *NVIDIA Corporation, September* (2009).

[9] Graefe, G., and Shapiro, L. D. Data compression and database performance. In *Applied Computing, 1991.,[Proceedings of the 1991] Symposium on* (1991), IEEE, pp. 22–27.

[10] Hollingsworth, B. New "Bulldozer" and "Piledriver" Instructions.

[11] Huffman, D. A. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE 40*, 9 (1952), 1098–1101.

[12] Kahan, W., and Zuras, D. An open question to developers of numerical software. *Computer 38*, 5 (2005), 91–94.

[13] Kim, H.-J., and Lee, W. Multi GPU performance of conjugate gradient algorithm with staggered fermions. *arXiv preprint arXiv:1010.4782* (2010).

[14] Marais, P., Kenwood, J., Smith, K. C., Kuttel, M. M., and Gain, J. Efficient compression of molecular dynamics trajectory files. *Journal of Computational Chemistry 33*, 27 (2012), 2131–2141.

[15] Roth, M. A., and Van Horn, S. J. Database compression. *ACM Sigmod Record 22*, 3 (1993), 31–39.

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

[17] Shainer, G., Ayoub, A., Lui, P., Liu, T., Kagan, M., Trott, C. R., Scantlen, G., and Crozier, P. S. The development of Mellanox/NVIDIA GPUDirect over InfiniBand—a new model for GPU to GPU communications. *Computer Science-Research and Development 26*, 3-4 (2011), 267–273.

[18] Shannon, C. E. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review 5*, 1 (2001), 3–55.

[19] Whitehead, N., and Fit-Florea, A. Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs. *rn (A+ B) 21* (2011), 1–1874919424.

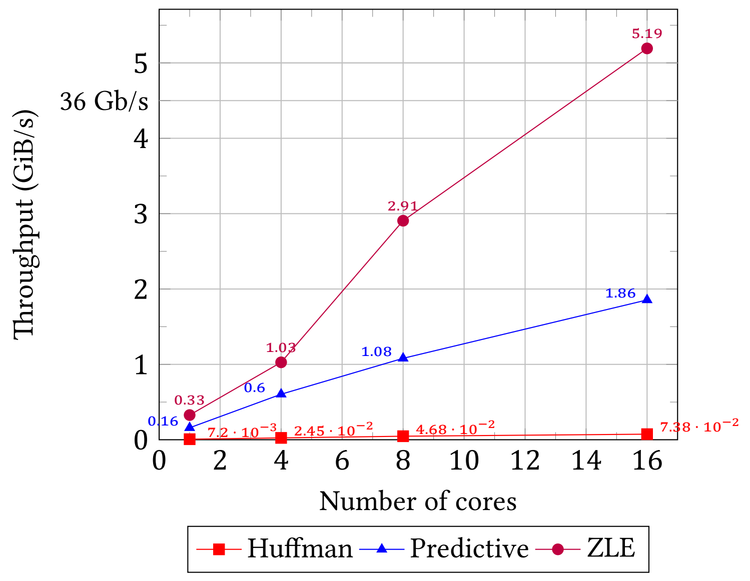
[20] Willhalm, T., Popovici, N., Boshmaf, Y., Plattner, H., Zeier, A., and Schaffner, J. SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. *Proceedings of the VLDB Endowment 2*, 1 (2009), 385–394.

[21] Yang, L., Dick, R. P., Lekatsas, H., and Chakradhar, S. Online memory compression for embedded systems. *ACM Transactions on Embedded Computing Systems (TECS) 9*, 3 (2010), 27.

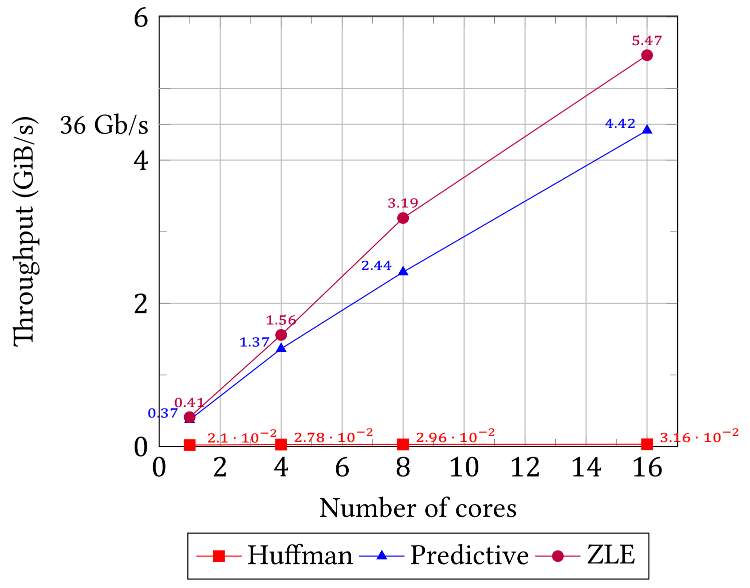
[22] Ziv, J., and Lempel, A. Compression of individual sequences via variable-rate coding. *Information Theory, IEEE Transactions on 24*, 5 (1978), 530–536.

Uncompressed	**Compressed**
Tyger! Tyger! burning bright	1,
In the forests of the night,	
What immortal hand or eye	
Could frame thy fearful symmetry?	Could 2,
In what distant deeps or skies	In 3, distant deeps or skies
Burnt the fire of thine eyes?	Burnt t4, f5, of thine eyes?
On what wings dare he aspire?	On 3, wings d6, 4, 7,5,?
What the hand dare sieze the fire?	W11, hand d6, sieze t4, f5,?
And what shoulder, & what art.	8, 3, shoulder, & 3, 9,.
Could twist the sinews of thy heart?	Could twist t4, sinews of thy 4,9,?
And when thy heart began to beat,	8, w4,n thy 4,9, began to beat,
What dread hand? & what dread feet?	W10, hand? & w10, feet?
What the hammer? what the chain?	W11, hammer? w11, chain?
In what furnace was thy brain?	In 3, furnace was thy brain?
What the anvil? what dread grasp	W11, anvil? w10, gr7,
Dare its deadly terrors clasp?	Dare its deadly terrors cl7,?
When the stars threw down their spears,	W4,n t4, stars threw down t4,ir sp12,,
And watered heaven with their tears,	8, watered 4,aven with t4,ir t12,,
Did he smile his work to see?	Did 4, smile his work to see?
Did he who made the Lamb make thee?	Did 4, who made t4, Lamb make t4,e?
Tyger! Tyger! burning bright	1,
In the forests of the night,	
What immortal hand or eye	
Dare frame thy fearful symmetry?	D6, 2,
	Dictionary:
	1, = Tyger! Tyger! burning bright#
	In the forests of the night,#
	What immortal hand or eye#
	2, = frame thy fearful symmetry?
	3, = what
	4, = he
	5, = ire
	6, = are
	7, = asp
	8, = And
	9, = art
	10, = hat dread
	11, = hat the
	12, = ears

Figure 1: A rudimentary dictionary-based example of compression on English poetry. By further encoding the dictionary symbols into the unused (in writing) ordinal numbers set, we can obtain a compressed size of 727 bytes (including dictionary) against an uncompressed size of 771 bytes. (This encoding is intended to be illustrative, not optimal.)



(a) Comparison of compression speeds achieved



(b) Comparison of decompression speeds achieved

Algorithm	**Huffman**	**Predictive**	**ZLE**
Ratio	0.41	0.933	0.969

(c) Comparison between compression ratios (raw data, excluding HDF5 metadata)

Figure 23: Comparison between parallel arithmetic, predictive and ZLE compressors (excluding Disk I/O)