UNIVERSITY OF CAPE TOWN

THESIS

Huffman Squeeze

Author:
Brandon J. TALBOT[1]

Group Members:
Benjamin Hugo
Heinrich Strauss

Supervisors:
A/Prof. James GAIN
A/Prof Patrick MARAIS
Jason MANLEY

A thesis submitted in fulfilment of the requirements
for the degree of Honours in Computer Science

in the

CUDA SQUEEZE
Computer Science

October 2013

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

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

University of Cape Town

Abstract

Science
Computer Science

Honours in Computer Science

Huffman Squeeze

by Brandon J. Talbot

Square Kilometer Array (SKA) is building a large radio telescopes array in South Africa which collects radio frequencies from objects in space. The radio frequency data is collected at 2 TB per second for many hours at a time, this means the final output file size lies in exabyte range. SKA have thus started plans to compress this data to alleviate huge expenses in storage and network speeds. The compression must not however negatively impact scan throughput, and so a Real Time compression scheme is required.

Since radio data is very noisy and many signals of interest are weak, it is desirable to retain all detail in the signal, which suggests using a lossless compression scheme. Since each scan ranges in the exabytes, but has few unique frequencies, an Entropy Encoding scheme seems to be the best approach for a chance of high compression ratios. There are 2 well known Entropy Encoding schemes, Arithmetic Coding which predominantly uses Dynamic Programming schemes and Huffman Coding which allows for many sections to be parallelised. Thus Huffman coding is the better candidate for the required SKA throughput.

The Huffman Coding algorithm we have designed for SKA uses GPUs and CPU parallelisation in order to achieve 71MB per second throughput on the tested system and a compression ratio of 41%. Which is better than the standard compression tools such as BZIP2 in both throughput and compression ratio but does not achieve real time compression for the SKA data.

Acknowledgements

I would to thank both my supervisors, A/Prof. James Gain and A/Prof. Patrick Marais, for their help throughout the project. Their insights truly did help me through the main processes and the final report of the project and are greatly appreciated.

I would then like to thank Jason Mannley who works at the SKA for all his help in the understanding of the SKA pipeline, and access to some SKA data.

Finally I would like to thank my project team member Benjamin Hugo who helped me work through sections code in order to find bugs or speed up certain processes.

Contents

List of Figures

Abbreviations

SKA	**S**quare **K**ilometer **A**rray
CPU	**C**entral **P**rocessing **U**nit
GPU	**G**raphical **P**rocessing **U**nit
RLE	**R**un **L**ength **E**ncoding
AHC	**A**daptive **H**uffman **C**oding
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
LZ	**L**empel **Z**iv
RAM	**R**andom **A**ccess **M**emory
SM/SMX	**S**treaming **M**ultiprocessor - the X is used for Kepler series SMs
CUDA	**C**ompute **U**unified **D**evice **A**rchitecture
STL	**S**tandard **T**emplate **L**ibrary
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
Mb	**M**ega**b**it
Gb	**G**iga**b**it
MB	**M**ega**B**yte
GB	**G**iga**B**yte
TB	**T**era**B**yte

Chapter 1

Introduction

1.1 Radio Astronomy

This report investigates the real time compression of radio telescope data. Radio Astronomy typically uses an array of large antennae in order to generate images of the sky. It does this by collecting the radio frequency data emitted by astronomical sources in the sky. Every object in space sends out some sort of radio frequency that can identify what the object is. Once a large range of frequencies have been collected they can be deciphered to determine exactly what is in that section of the sky.

Observations span many hours, since more data allows for a better reconstruction of the sky.

1.2 SKA and South Africa

Square Kilometer Array (SKA) currently have an array of antennae in South Africa called KAT-7. This array consists of 7 antennae and produces around 2TB of frequency data every second. Since each observation spans several hours, and data is received at 2TB per second, the final size of an observation may run into the exabyte range. The SKA project intends on upgrading KAT-7 into MeerKAT in 3 phases to end up with 90+ antennae, and the final SKA goal is a full Square Kilometer of antennae in South Africa. The data retrieval rate for the radio frequencies increase quadratically per antennae, thus the final square kilometer of antennae will produce 1PB every 20 seconds.

Since hard disk space of exabyte size is not cheap, the SKA project requires an efficient compression ratio. Since the data is received at such a high rate, the compressor must be able to compress a stream of data.

Currently SKA does not have any compression in place; this is mainly due to the very demanding process requirement of speed and accuracy for the project. Since radio frequency data could both contain external spikes and spikes that are caused by objects SKA are actually interested in, the data cannot be compressed through a Lossy compression algorithm.

In a lossless compression scheme the data is completely revertible without any loss of data. Lossy compression is useful in circumstances were subtle changes in data are acceptable such as removing high frequency data in images for jpeg compression[1].

1.3 Compression Scheme Chosen

Entropy Encoding[2] was chosen for this compression task since it is a lossless form of compression. Huffman Coding[3] was the selected algorithm within Entropy Encoders since it is the most suitable option for parallel design. Many other Entropy Coders use Dynamic Programming which is not easy to parallelise. Huffman Coding has a smaller Dynamic programming section and many procedures that could be parallelised.

Our Huffman Coder achieves very good compression ratios; where the output file is only 41% of the input data size, and compresses at a rate of 48MB/s on average (including Disk IO). This beats all the standard compression programs available, such as BZIP2 and GZIP in both speed and compression ratio. Compression schemes are usually compared through the Compression Ratio and Speed. In this report we will indicate Compression ratio as $\frac{\text{Output File size}}{\text{Input File Size}}$, and speed as the data rate in Megabytes per second that the compression algorithm can achieve.

1.4 Research Questions

The choice of several Huffman Coding inquiries to answer:

1. Is it possible to create a SKA Real Time compression algorithm using Huffman Coding?
2. Is Huffman Coding a better choice to compression schemes already available?
3. Is the compression ratio worth any time overhead it produces?

The first question is related to the speed of the algorithm in comparison to the arrival rate of the data blocks SKA radio dishes produce. For real time execution for SKA, the compression speed of the data needs to be approximately 5GB/s plus.

The second question relates to algorithms SKA could already use, those being BZIP, GZIP and many others. Our algorithm has to be faster to make this a viable alternative for SKA.

The final question is a very hard question to answer. If the compression takes too long it will not be worth while, but if it can produce a really good compression ratio and not slow the SKA pipeline too much, it may still be useful.

1.5 Report Structure

Chapter. 2 introduces the SKA's infrastructure, how compression schemes work, previous compression done using the Huffman Coding scheme and the fundamentals of parallel computing. Chapter. 3 outlines the design for the Huffman Coding algorithm for SKA, in terms of both an Adaptive and Dynamic approach. Chapter. 4 depicts the implementation for the SKA Adaptive Huffman coding scheme whereas Chapter. 5 depicts the Dynamic Huffman Coding scheme implementation. Chapter. 6 depicts our results and Chapter. 7 speaks to the final decision of the algorithms feasibility.

Chapter 2

Background

2.1 Square Kilometer Array in South Africa

SKA South Africa will soon commence construction on the largest array of radio telescope dishes in the world. As mentioned previously, SKA currently have a 7 Dish array called KAT-7 as part of its MeerKAT system. This array will be upgraded bit by bit to become the full square kilometer array.

The radio frequency data that arrives at 2TB/s from the KAT-7 array consists of floating point pairs, representing one real and one imaginary component. The current system uses FPGAs[4] to calculate the data at the base of each dish. This data is then sent to the main offices where further calculations are done to convert the data into the received floating point values. This transfer is done over a network with speeds of 36.6 Gigabits per second which averages to around 5GB/s.

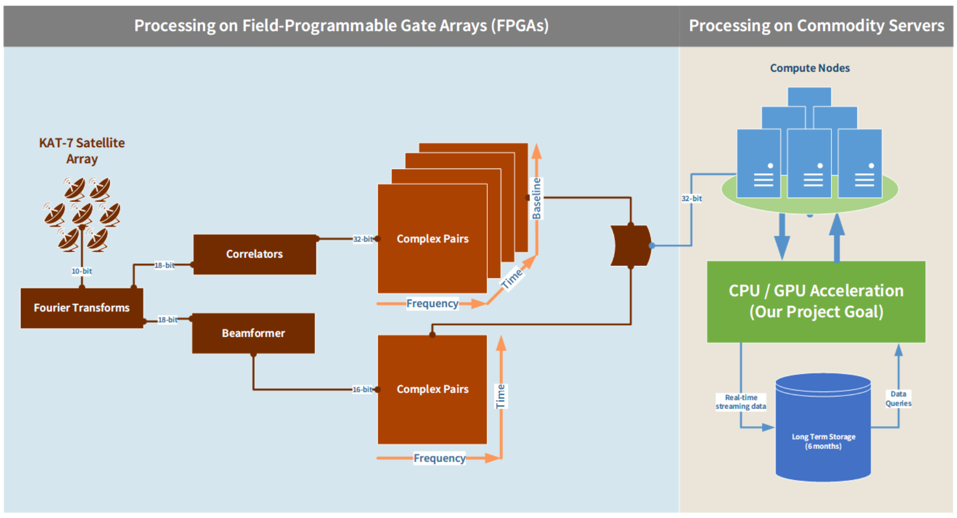

FIGURE 2.1: High Level overview of the MeerKAT pipeline, showing how the data is converted from radio frequency data into 32-bit floating point values before being stored.

The aforementioned floating point data tends to span a rather narrow frequency range, with very few values moving far out of the range(”spikes”). This means that there will be very few unique values in comparison to the total number of values contained in the output data. This is good as such data plays to an Entropy Encoders strengths, since entropy encoders use unique symbols in order to compress data.

Fig 2.1 shows the MeerKAT pipeline from the dishes to the storage location. It shows the conversion steps from frequency input being 10 bits to the final 32 bit floating point data output. The proposed compression step will take place between the computation nodes at the main office and the storage facility the data is to be stored.

2.2 Compression Schemes

Compression algorithms can be broken down into 4 major groups[5]. We discuss each of these groups below.

2.2.1 Basic Methods

Basic compression schemes are usually fast, but not very effective, since they tend to be processes that complex schemes use together for better compression ratios. The most widely used basic compression scheme is Run Length Encoding (RLE), which compresses data by finding repetitions of symbols and then representing the repetition by its symbol and the number of repetitions. Doing so reduces the number of values from more than 2 values to an integer and a symbol. If the value does not repeat, it only records that symbol as to not increase the size of the data.

Basic schemes work for both streaming and non-streaming data and could thus be suitable for SKA radio data. However since basic schemes seldom achieve good compression ratios, one was not chosen for this report.

2.2.2 Statistical Methods

Statistical methods use mathematical principals in order to change, or map the data into a smaller representation that can be reversed. The two most widely used schemes under statistical methods are Entropy Encoders and Predictive methods.[6]

Predictive schemes use probabilities of the next symbol being a certain value in order to compress data. This is done by predicting the next value and then comparing it to the actual next value. This is most commonly done by taking the difference between the predicted value and the actual value and saving that difference in a more compact binary representation. The difference value will in most cases be smaller than the actual value since in most cases that difference will have more zeroes starting from the most significant bit in the binary representation. These zeroes can be truncated in the recorded value.

Entropy Encoders will be discussed in depth in the next section since the chosen algorithm for this paper falls into this category.

2.2.3 Transformations

Transformation methods typically use Wavelets or Fourier Transforms to change data into a format that can either allow for better compression, or the ability to remove data that will not noticeably affect the signal. Due to this removal of data, such transformation methods are used primarily for visual data where a small artefact will not cause a huge problem[7].

2.2.4 Dictionary Methods

Dictionary methods use sorting and pattern finding algorithms to find unique sequences of data or repeated patterns. These methods then use one of the other schemes in order to compress these patterns of symbols, rather than compressing each symbol on its own. This usually gives a better compression ratio that using each scheme on its own. Due to the requirement of finding patterns that are utilised in the data more than once, the data has to be known in advance and thus streaming data tends to not work very well.
The most widely used pattern finding methods for Dictionary methods are from the Lempel Ziv (LZ) compression scheme[8].

2.3 Entropy Encoders

Entropy Encoders use the probabilities of each symbol being in the data in order to find more compact representations for each symbol, or for the entire data set. Huffman Coding and Arithmetic Coding[9] are the most widely known Entropy coders.

Huffman coding was the algorithm chosen for this report and will be discussed in depth in the next section.

Arithmetic coding calculates the probability of each symbol within the data and represents this through a range of values 0 and 1. For example: If the symbol *2* was found to comprise 20% of the data, it would be all the numbers from 0 to (not including) 0.2. Once all the probabilities are found, they are sorted from least probable block starting at 0, to the most probable block ending at 1. The data is then read in order, taking each symbol, finding the block representing that symbol and recreating the table between that blocks starting and ending numbers. This is repeated until the last value has changed the tables starting and ending values. Either the higher or lower value of the table is then used to represent the entire dataset with a table of all the symbols and their probabilities. The data is decompressed in the same way, except that each next table is chosen by were the given decimal value was found[10].

The requirement of repeated steps with the correct order means that Arithmetic coding is a Dynamic Programming problem[11], and thus not easily parallelisable and so unlikely to be accelerated very much by a parallel implementation which is a high priority for this task.

2.4 Huffman Coding Overview

Huffman Coding uses the probability of each unique symbol to determine codes (binary sequences) that map each symbol to a new binary representation. It gives the symbol that occurs the most frequently the shortest code. It does this most effectively through the use of a tree structure. Huffman Coding is most widely used for JPEG compression[1] and MPEG-2/4 AAC (MPEG audio channel) compression[3].

Once the Huffman coder has determined the codes for all unique symbols, it replaces each symbol with the code. These codes are placed into a single binary sequence in order to save the data with a table representing each unique symbol and its respective probability. The decompression scheme generates the same tree structure used to generate the code, through the use of the table of unique symbols and respective probabilities in order to take the binary sequence and convert that into its original symbols.

For instance, lets define each symbol as a single character, then try to compress the following data "abra-cadabra".
Firstly the data needs to be binned (each unique char counted):

- a - 5
- b - 2
- r - 2
- c - 1
- d - 1

The tree is then constructed using the probabilities of each character, since "a" has the most occurrences it needs to be the leaf node closest to the root node of the tree (the node that has no children, closest to the root of the tree). "b" and "r" have to be the next 2 nodes closest to the root and "c" and "d" can be the furthest from the root. The easiest way to do this is to take the 2 values in our list that have the lowest

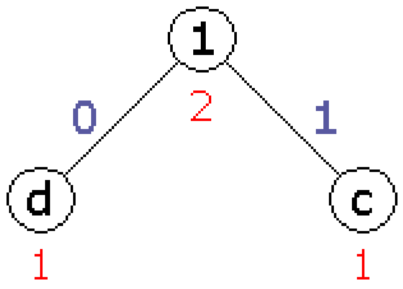

FIGURE 2.2: Huffman Coding tree example for "abracadabra" after the first step

counts and create a node to join them called a joiner with its weight being the children's weights added together as shown in 2.2.

We then add that joiner node "1" to the list with its given weight:

- a - 5
- b - 2
- r - 2
- 1 - 2

The same process is repeated, joining 1 and r together to create the next tree 2.3.

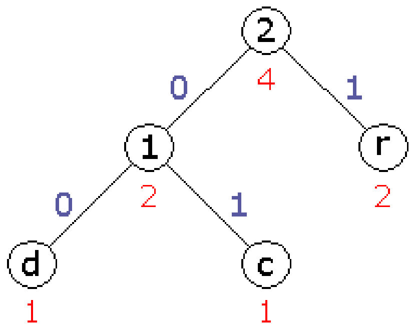

FIGURE 2.3: Huffman Coding tree example for "abracadabra" after the second step

The weight list is now changed to:

- a - 5
- b - 2
- 2 - 4

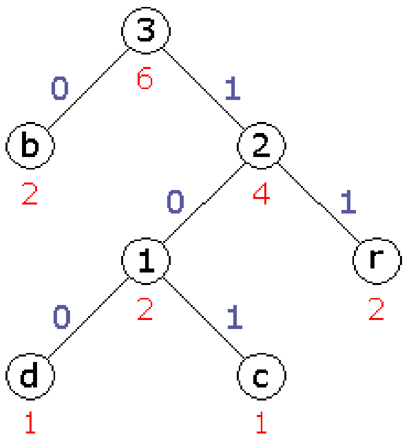

FIGURE 2.4: Huffman Coding tree example for "abracadabra" after the third step

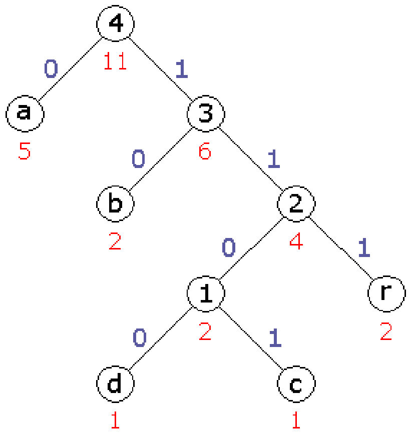

FIGURE 2.5: Huffman Coding tree example for "abracadabra" after the forth step

Since there is still more than 1 item in the weights list another 2 more joinings will have to occur shown in figures: 2.4 and 2.5

Note that the nodes "B" and "A" were placed on the left of the joiner node, this is done since all children nodes of joiners must follow a simple rule: the left child must have a smaller or equal weight than the right node.

From the final tree the codes for our symbols can be calculated, the Blue numbers along each line joining the nodes indicates the binary value added for that path. Thus the code for "a" is "0", but the code for "c" is "1101". Thus the codes table is:

- a - 0
- b - 10
- r - 111
- d - 1100

- c - 1101

The next step is to replace each character in our data with its code, the output full binary sequence will be:

a	b	r	a	c	a	d	a	b	r	a
0	10	111	0	1101	0	1100	0	10	111	0

Giving us a total length of 23 binary digits, this means 3 bytes will be needed to save these 23 binary digits, as a byte is made up of 8 binary values. This shows that the output binary sequence will be smaller than the 11 bytes for the original data.

2.5 Huffman Coder Extensions

Huffman Coding has been extended in many ways to both improve speed and memory usage for specific data. JPEG encoders[1, 12] have been accelerated by using a Hash Map once the codes were constructed rather than using the tree and looking for the symbols leaf node. This allowed the tree to be completely deleted and thus save memory.

Other changes involve removing weighting totals for each node. These are usually used to stop a tree from becoming too large and thus generating codes larger than the original input symbols.

Adaptive Huffman coding is an extension used to make Huffman Coding work on streaming data, since the probability cannot be calculated if the entire dataset cannot be read in advance. It achieves this by having a recalculation step each time a symbol is encoded and its probability changes. Each time a Symbol is encoded, that symbol now occurs more often in the file than it previously did, it is thus more probable than a previous symbol and needs its binary code to be shortened before it is used again. The tree is thus re-arranged each time a symbol is encoded. The decompression follows the same process for each decoding step.

2.6 Parallelism

Parallel execution is the process of running many instances of code at the same time. In order for code to run in parallel and still have the desired effect, that code has to be disjoint. This means the processes in the code that will be run in parallel cannot require data that is changed by one of the other parallel processes. Many issues arise with parallel code. Issues typically arise when each parallel process needs to update the same variable, or they all need to execute one process, then wait for all to finish before continuing the other processes. Using Atomic variables when a single variable needs to be updated fixes synchronization issues by making all parallel processes wait while the variable is being changed by one of the processes. The second issue is fixed by a synchronisation barrier. This forces all parallel processes to wait until every parallel process encounters the barrier.

Parallelisation can be done using the Central Processing Unit (CPU) and the Graphical Processing Unit (GPU), we will now discuss each of these architectures.

2.6.1 Parallelism via the CPU

CPU threads are the most commonly used approach to accelerate execution since it tends to be allot simpler to code and allows for the full use of the computer's Random Access Memory (RAM). CPUs also tend to have larger cache sizes than GPUs but the CPU will have far fewer processing cores, meaning the CPU cannot process as many threads at the same time.

The CPU can only run the same number of processes (threads) as cores it contains in parallel to its full potential, since if more threads are run at the same time a process scheduler has to schedule many threads to be run by a single core causing overheads.

Further parallelisation for the CPU could be achieved by using SSE (Streaming SIMD instructions). These instructions allow for faster processing of certain algorithms. SSE allows for vectorised data to be processed in parallel. SSE only works for single basic instructions to be run on data at the same time. For example: if you need to divide 1000 values in an array by 2. You can call on the SSE methods and within a few instructions steps all 1000 divisions will be computed, rather than issuing 1000 division instructions sequentially. AVX is a slight upgrade to SSE allowing for larger data sets and memory blocks to be used.

2.6.2 Parallelism via the GPU

The GPU has a different architecture to the CPU. It has a single large memory block (the GPU RAM), and a separate cache, shared memory and texture memory for each SMX[13]. Whereas the CPU has a single cache used by all cores. Each GPU SMX also has the capability of running many threads at a time, meaning algorithms can be made parallel in blocks of memory, and then threads for that block of memory as well.

One of the problems with GPU parallelism is the requirement of small data blocks per SMX, since the SMX has very small cache or shared memory, and the overhead of sending data to the main graphics card RAM and back when processed.

There are two two main programming languages for programming parallel graphics processes: OpenCL and CUDA. OpenCL (Open Computing Language) is an open source language that works for both AMD and NVIDIA graphics cards. CUDA (Compute Unified Device Architecture) is designed for NVIDIA graphics cards and is easier to program than OpenCL. CUDA also has many libraries of already designed algorithms that have been optimized such as Thrust.

A method generated to run on the graphics card is called a Kernel. Only the host (CPU) can call on the graphics card to process a kernel except for the new NVIDIA graphics cards of compute level 3.5 and above, which added the functionality of kernels calling on other kernels. When a kernel is called the number of SMXs and threads per SMX needs to be allocated. The number of threads per SMX required for best efficiency varies greatly for each graphics cards due to hardware differences.

Thrust is a library of CUDA processes with data structures that help copy data to and from the graphics card. This allows for algorithms to be programmed more easily and faster than having to write your own kernel for each process. Thrust libraries also call on the correct number of SMXs and threads per SMX for the compute level it is compiled for, thus no need to determine the correct number of threads per SMX for the current hardware.

Chapter 3

Design

This chapter investigates how to modify Huffman Coding to suit the data and speed requirements of the SKA radio data.

3.1 Overview

Tests we ran with the standard Huffman Coding algorithm averaged around 2MB/s for compression speeds on Intel Ivy Bridge I7 950 @ 3.07GHz. Theoretically this speed should be multiplied by the number of cores used in a parallel approach that scales completely. Unfortunately in most cases this is not true, since there are many overheads to calling on parallel threads, as well as many sections of code that can not be run in parallel.
Studies show that Huffman Coding does not perform well on a GPU (Graphics Processing Unit) due to small cache sizes per SM/SMX and the requirement of using a Tree structures and the Dynamic programming required in many stages of the Huffman process[14]. Thus a CPU parallel approach will most likely achieve better compression speeds than a GPU approach, mainly due to memory access speeds and Host-to-Device memory transfer speeds.
Since the standard Huffman Coding algorithm only achieves 2MB/s and in best case we could only expect the speed to be multiplied by the number of cores a CPU contains, simply making the standard algorithm parallel will not suffice. We thus need to adapt the standard Huffman Coding algorithm to the SKA processing requirements.

As stated previously, the SKA data tends to span a narrow frequency range with very few spikes, and thus possess few unique values. This is to Huffman Coding's advantage as each code can be made smaller and the table will be smaller. This means that Huffman coding will most likely attain good compression ratio for the SKA data. The issue lies with the speed of the algorithm, and whether it is worth using Huffman coding over the standard algorithms, such as GZIP, BZIP, BZIP2 and WinRAR/WinZIP, the last

2 being being freeware most commonly used by the public. Further comparisons are required with other research approaches SKA have running, namely a parallel RLE and Predictive scheme.

3.2 Algorithms

In order to test the Huffman Coding scheme fully for streaming tasks, both Adaptive and Dynamic approaches need to be tested.

3.2.1 Huffman Coding

Huffman Coding (also known as Dynamic Huffman Coding) uses a tree structure in order to quickly generate binary sequences for input symbols. The symbols that occur the most frequently are assigned the shortest binary sequence. This method requires the ability to read all values in the file in advance in order to generate the required tree structure.

Huffman coding generates this tree by first binning (finding all unique values and how many of each unique value are in the file) the file and then using a Dynamic Programming scheme to generate the tree. Once the tree is constructed all leaf nodes correspond to unique values, and the closest leaf node to the root node of the tree corresponds to the unique value with the most occurrences. All inner nodes (non-leaf nodes) are just to help determine the structure of the tree.

Once the tree is complete, the binary sequence codes need to be constructed. The most efficient way to do this is is to start from the leaf nodes, then move up the tree checking if the node is a left, or right child adding the respective 0 or 1 binary values to the code of that leaf node, until the root node has been reached. Pseudo Code. 3.1.

The reverseArray() method is required since decompression requires a root to leaf binary sequence in order to determine the correct unique value associated to that binary sequence. Some algorithms skip this reverseArray() method since they use Break Point values in between each code. However break points tend to reduce the compression ratio and for the SKA data, reduce the processing speed too(see Chapters. 5 and 6).

For the Encoding procedure, each symbol value is taken and replaced with the binary sequence generated for that symbol. All the binary sequences are then placed after one another and written to a file as a sequence of byte values (8 binary bits at a time), with padding of 0s at the end to finish the last byte value. This allows the decoder to start at the beginning of the full binary sequence for the decoding process. The decoding process starts by moving down the tree according to the next binary value in the given sequence, 0 for left node movement and 1 for right node movement, this is done until a leaf node is reached, then that

```
bool [] generateCode(node leafNode)
BEGIN
        node parentNode = leafNode.parent;
        bool [] codeArray;
        WHILE (parentNode is not NULL (Exists)) do
                if (leafNode == parentNode.left)
                        codeArray.add(0);
                else if (leafNode == parentNode.right)
                        codeArray.add(1);

                leafNode = parentNode;
                parentNode = parentNode.parent;
        END WHILE

        reverseCode(codeArray);

        RETURN codeArray;
END
```

FIGURE 3.1: Pseudo code for the Code Generation method in Huffman Coding. This code generates all the codes for the leaf nodes.

number is added to the output values array and the process is started again from the root node until there are no binary values left to read. Pseudo Code. 3.2.

```
symbolType [] decode(bool[] fullBinarySequence)
BEGIN
        node treeNode = rootNode;
        symbolType [] symbols;
        // NOTE: padding must not add an extra symbol, so check for codeLength too
        int pos = 0;
        WHILE (pos > fullBinarySequence.length - shortestCodeLength)
                if (fullBinarySequence[pos] == 0)
                        treeNode = treeNode.left;
                else
                        treeNode = treeNode.right;

                IF (treeNode is leafNode)
                        symbols.add(treeNode.symbol);
                        treeNode = rootNode;
                END IF

                pos++;
        END WHILE

        RETURN symbols;
END
```

FIGURE 3.2: Pseudo code for the Huffman Coding decompression method.

The process of making Huffman Coding parallel entails parallelising the Binning method and the symbol to binary sequence swapping method, or by implementing a blocking scheme, where the data is split into blocks of equal size and compressed simultaneously. The blocking method is expected to be faster than the parallel method as there is a large dynamic programming section to Huffman coding.

The output file consists of a table associating each symbol and its count in order to reconstruct the same tree, followed by the full sequence of encoded binary values.

3.2.2 Adaptive Huffman Coding

Adaptive Huffman Coding (AHC) was designed so that Huffman coding can work on streaming data does and not require full knowledge of the data in advance.

The tree structure differs slightly from Dynamic Huffman Coding, as the furthest leaf node from the root node must always be a NULL or null placement holder (nullPTR), so that a new symbol can always be added to the tree.

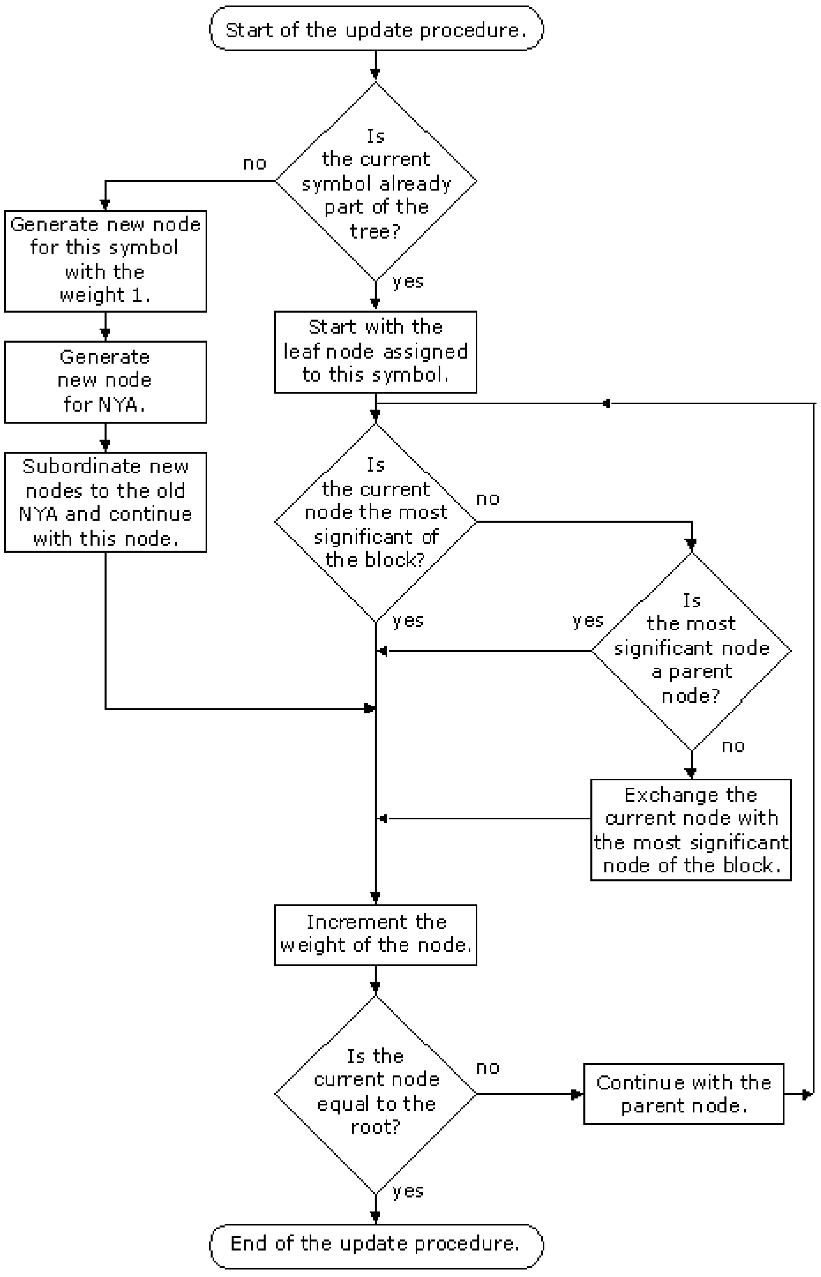
Start of the update procedure.
Is the current symbol already part of the tree?
no
yes
Generate new node for this symbol with the weight 1.
Generate new node for NYA.
Subordinate new nodes to the old NYA and continue with this node.
Start with the leaf node assigned to this symbol.
Is the current node the most significant of the block?
no
yes
Is the most significant node a parent node?
yes
no
Exchange the current node with the most significant node of the block.
Increment the weight of the node.
Is the current node equal to the root?
no
yes
Continue with the parent node.
End of the update procedure.

Figure 3.3: Update Process for Adaptive Huffman Coding

The basic principal of AHC is to start with a root node and a nullPTR leaf node. We then start reading symbols. When a new symbol arrives, we add a new node with the weight of 1 to where the nullPTR node was, this node has 2 children, the nullPTR node and a new leaf node with the new symbol and a weight of 1 and running the update procedure on the parent node, when a symbol previously seen arrives run the update procedure on the leaf node containing that symbol.
Fig. 3.3 details this update procedure. It entails checking if the node needs replacing with a higher node since its weight will rival that higher node (one closer to the root node), increasing the nodes weight and running the same update procedure on the parent node until arrival at the root node.[15]

Decoding AHC requires the correct sequence of unique symbols as they were found when compressing and the full binary sequence. It decodes the same way as Dynamic Huffman Coding, with a slight difference: when arriving at the nullPTR node it places the next unique symbol in the read sequence the same way it would add a new symbol when compressing, also running the same update procedure on leaf nodes each time one is found.

Due to AHC having such a linear approach and a tree update procedure it is impossible to parallelise AHC. The only alternative would be to block the data in some way, thus defeating the purpose of AHC which is to support streaming and not to require the data in advance.

The output file for AHC comprises of the unique symbols sequence in the correct order of appearance, followed by the full binary sequence.

3.3 Prefix Sum

The major bottleneck within the Huffman coding algorithm is the conversion from the final binary sequence to byte array. Usually this cannot be done efficiently because each symbols binary sequence is in its own array and of arbitrary length.

A Prefix sum is an accumulation of elements in an array that can undergo a binary associative operation $\oplus$ where the identity value for the operation is I. If $A = [A_1, A_2, \dots, A_{n-1}, A_n]$, where n is the number of values in the array, then prefix sum is $PS = [I, A_1, (A_1 \oplus A_2), \dots, (A_1 \oplus A_2 \oplus \cdots \oplus A_{n-2})]$.[17]
For instance: If you have the following array of integers:

$$[1, 6, 9, 10, 2, 3, 12]$$

The prefix sum will be:

$$[0, 1, 7, 16, 26, 28, 31, 43]$$

Usually a Dynamic programming method is used to generate bytes in linear until no binary digits are left. We decided to try and speed this up by using a prefix sum to flatten a 2 dimensional binary sequence array into a one dimension binary sequence array in parallel, which allows for the flattened binary sequence array to be converted to a byte array in parallel. This should speed up the binary sequence to byte conversion.

3.4 GPU based Huffman Coding

As discussed earlier, GPU based Huffman coding algorithms do not fair well. However we decided to try a GPU-CPU combined algorithm using the Thrust libraries since they do not require allot of work to use and are already optimized. Since the Thrust libraries only contain Binning methods and prefix sums, these are the two sections of code we decided to test on the graphics card.

The binning method for thrust requires the number of unique values in advance, which in our case is not possible. Luckily Thrust contains other methods such as: `inner_product()` which can calculate the number of unique values in an array. Thus the final procedure required to bin the entire data set is: Firstly the data needs to be sorted using Thrusts sort method, this is because the `inner_product` method requires all unique values to be bunched together. The `inner_product` method is then run in order to initialize the `unique_Values[]` and `unique_values_count[]` arrays. The binning method can then be run to find and count the number of each unique value. Thrusts `reduce_by_key` method is used to do this.

The Thrust prefix sum method can be used to determine a exclusive scan of the lengths of each array in our 2 dimensional binary sequence array. Thrusts `exclusive_scan` method is run on an array containing all these lengths. Once the prefix sum is complete, the values in the prefix sum array are the starting positions of each array in the 2 dimensional binary sequence array. This allows us to initialize a flat binary sequence array and then flatten the 2 dimensional binary sequence array using the CPU.
Once the binary sequence array has been flattened, it can be converted into byte values to be written to a file.

3.5 Benchmarking

The process of testing the final stage of the SKA's Huffman coder will entail using sample data provided by the SKA. The main comparisons will be comparisons of parallel versions, and finally comparisons to standard compression algorithms. This will require multiple runs for each, on the same system. A time average for all algorithms over a minimum of 6 runs will be required for all test parameters. These parameters will be, 1 threads (or linear running), 4 threads (standard PC for public users), 8 threads and 16 threads. A combined GPU and CPU version will be constructed and will be compared to the equivalent CPU version (same number of CPU threads for each algorithm).

The final testing stages will be a comparison for SKA to determine the best possible scheme to use. This will compare all the SKA researched compression schemes. This comparison will be made fair by making all algorithms compress the same file.

Chapter 4

Adaptive Huffman Implementation

The SKA data is consists of floating point values only, and thus the AHC algorithm we designed was specified for floating point data.

4.1 Tree Structure

The tree structure for Adaptive Huffman Coding requires the ability to find the most significant node, or the leaf node closest to the root node with a specified weight. The fastest way to do this is by using a priority queue. The queue is kept in memory and contains all the leaf node pointers. This means however that the tree nodes all need another variable, depth. Unlike Dynamic Huffman Coding, the creation of the code can't happen at the end of the tree creation, since the tree is always changing. The code is thus generated while the nodes are created and changed when nodes move, all to save processing time.

The basic node structure for the tree is shown in Figure. 4.1: The value is kept and can be null, to indicate that it is a *joiner* node. The weight is required to construct the final tree. The depth of the node and the code is required for AHC tree construction and the usual links to that nodes parent and its left and right children.

4.2 Encoding Procedure

The encoding procedure is executed when a new value arrives. This process requires finding the nullPTR node. The search process can be accelerated by keeping a pointer to the nullPTR nodes parent at all times in the AHC class. Another process that has to happen each and every time a value is encoded, is locating the node that contains the same value, if there is one. We improved the speed of this step by keeping all unique values with a pointer to its leaf node in a Hash Map. The hash map can then be used to determine

```
Class Node
BEGIN
        // The constructor needs to add the new bit to the code
        Node(float data, Node parent, int width, int depth, bool[] code, bool newBit)
        BEGIN
                //Initialise all variables

                if (newBit not NULL)
                        code.add(newBit);
        END Constructor

        float data; // can be null

        int weight;
        int depth;

        bool[] code; // keep the code for all nodes

        Node Parent;
        Node left;
        Node right;
END
```

FIGURE 4.1: Pseudo code for the Adaptive Huffman Coding Node class used to construct the Huffman tree. The most important things to note are: The new depth and code variables, and the constructor must add a new bit to the code.

if the value has been encoded before and hence find the leaf node quickly.

The final approach that was used to speed the AHC algorithm was locating of the *Most Significant Node*. Another Hash Map, mapping node weights to priority queues is used. When a new value is encoded it is placed in the correct priority queue. If that priority queue does not exist it is created. When each nodes weight changes it is moved to the correct priority queue. These priority queues sort the data according to the node's depth value, the smallest depth being at the top. This means when the highest node with a specific weight is needed, it can be found by looking at the priority queue pointed to by the hash map with that weight value.

The encoding procedure starts by determining if the tree has been started or not, if not the root node and the first right child are created, leaving the left child as the nullPTR node, and the data is placed into the required hash maps for easy future finding. If the tree is already constructed we determine if the symbol to be added already has a node in the tree by using the constructed hash map. If it does exist then the code is copied and that node is updated, then the associated binary sequence is returned. If the symbol does not already have a node associated to it, one is constructed were the nullPTR is just as the root was created. A new joiner with the right child as the new node and the left child as the nullPTR. The required values are saved in their respective hash maps and the tree is updated as shown in the Algorithm. 4.2.

```
bool [] encodeValue(float Symbol){
        IF (The tree has not yet been constructed)
                Construct The Tree using the global root pointer

                // add to the leaf nodes hash map
                uniqueHaspMap[symbol] = rootNode.right
                listOfNewValues.add(symbol) //this is we keep the order of the unique values

                nullPTRParent = Parent of new nullPTR

                checkNodeFordepthHashMap(rootNode.right)

                updateNode(rootNode)

                RETURN symbol code
        END IF

        IF (the symbol has been encoded before)
                leafNode <- Get the leaf node associated to the symbol from the hash map

                bool[] code = leafNode.code // copied since the update process could change it

                updateNode(leafNode)

                RETURN code
        ELSE
                nullPTRParent.left <- new joiner node
                nullPTRParent.left.right <- new leaf node with the new symbol

                // copy the code so that the update process does not change it
                bool[] code = nullPTRParent.left.right.code;

                // add the the leaf node hash map
                uniqueList[symbol] = nullPTRParent.left.right
                // add to the order list
                listOfNewValues.add(symbol)

                nullPTRParent <- The new parent of the nullPTR

                checkNodeFordepthHashMap(nullPTRPArent.right)

                update(nullPTRParent)

                RETURN code
        END IF
}
```

Figure 4.2: Pseudo code showing the Adaptive Huffman Coding algorithm's encoding procedure, The algorithm takes in a symbol, adds it to the tree updating and swapping nodes if needed, and outputs the associated binary sequence for that symbol.

4.3 Update Procedure

The update process follows the same process as the standard AHC code; we use the new methods to find the most significant node. The updated method is shown in Figure. 4.3

```
update(Node node)
BEGIN
        IF (node is not the root node)
                mostSig <- getMostSignificantNode(node.weight); //find the highest node

                IF (mostSig not the node AND mostSig not the nodes parent AND mostSig not NULL)
                        swapNodes(node, mostSig); // this swaps position, codes, weights etc
                END IF
        END IF

        node.weight++;

        IF (node is not the root node)
                checkNodeFordepthHashMap(node); // check for change in the priority queue
                update(node.parent); // update the parent
        END IF
END
```

Figure 4.3: Pseudo code for the Adaptive Huffman Coding algorithm's update node procedure, updated to use the Find Most Significant Node method. This increase each nodes weight and does any node swaps that are needed to keep the symbol with the most occurrences closest to the root node.

4.4 Feasibility of AHC

Since each value has the possibility of swapping all the values within the tree, and updating the weights of nodes every single encoding procedure, parallelising any of the encoding procedures will cause synchronisation problems. If the update procedure is parallelised, the encoding tree could be constructed differently to the decoding tree, thus not decoding correctly. Another issue arises when one thread is updating the tree and busy increasing the weights of all the parent nodes, while another thread swaps these parent nodes after the previous update is at that parent node. Subsequent parent nodes that are updated will then be incorrect as they have moved.

Since a parallel version of the code is not viable and Adaptive Huffman is slow, averaging 10 times slower than Dynamic Huffman. Blocking (splitting data into blocks and running separate versions of the algorithm on each block at the same time) AHC in order to parallelise it is not worth while. It would be easier and faster to block the Dynamic Huffman algorithm.

Chapter 5

Huffman Coding Implementation

Since certain parts of the DHC algorithm can be executed faster when using CUDA than any CPU parallel system, we developed a CPU and combined GPU-CPU version of the algorithm was designed for SKA.

5.1 CPU Only Huffman Coding

The same tree used in AHC was used for Huffman Coding, but the Code and depth values are removed as they are not needed for the construction of the Huffman Tree or the codes.

Both a parallel algorithm and a blocked system were created for Huffman Coding. It was determined that the Blocked algorithm is approximately 10 times faster, and thus the parallel algorithm was abandoned to allow time to optimize the blocked version of the code.

5.1.1 Binning

The initial step for Huffman Coding is to Bin the data i.e calculate each unique symbol's probability. This is done by creating a hash map of symbols to symbol counts. The data is read and when a new symbol is found it is added to the hash map with a count of 1. If that symbol has been seen, before the count value for that symbol is increased by 1. This can be done in parallel with a few synchronisation checks. Firstly an atomic count variable is used, and a critical block is constructed to only allow one thread to construct a new symbol at a time.
The binning method developed for our Blocked Huffman coding algorithm is presented in Algorithm. 5.1

The binning algorithm uses a Hash Map of IEEE 32-bit floats to 32-bit integer value to save the count of each float value.

```
BinData(float[] data)
BEGIN
        FOR (int i = 0; i < numberFloats; ++i)
                float value = data[i];

                IF (FrequencyHashMap does not contain the value)
                                FrequencyHashMap[value] = 1;
                ELSE
                        FrequencyHashMap[value]++;
                END IF
        END FOR
END
```

FIGURE 5.1: Pseudo code for the Huffman Coding binning method. This finds and counts all the unique symbols.

5.1.2 Tree Construction and Code Generation

The next step is to construct the tree. As noted in Chapter. 3, a Dynamic programming scheme is used and so no parallel version could be designed. A Priority queue ordered on each tree node's weight is used. An output list of all leaf nodes is also kept for the code construction, as explained in Chapter. 3. The only difference is that the tree is deleted after the HashMap of symbol to Code is constructed. The generate tree method is presented in Algorithm. 5.2.

```
GenerateTree(Nodes[] leafNodes)
BEGIN
        PriorityQueue<Node> queue

        FOR (all values in FrequencyHashMap)
                node <- new leaf node containing the value
                queue.push(node)
                leafNodes.add(node)
        END FOR

        WHILE (the queue contains more than one node)
                Node a = queue.pop
                Node b = queue.pop

                Node parent <- new joiner node

                parent.left = a
                parent.right = b

                a.parent = parent
                b.parent = parent

                queue.push(parent)
        END WHILE

        DELETE the tree;
END
```

FIGURE 5.2: Pseudo code for the Huffman Coding Generate Tree method. This is how the Huffman tree is constructed using a Dynamic programming procedure with a priority queue to place symbols that appear more often closer to the root node.

The generation of the codes can be done in parallel when not blocking the data, as each leaf node's code can be constructed at the same time. Due to threads fighting for control of a CPU core when more threads than the number of cores on a CPU are used, it is not recommended to execute any parallel algorithm while running blocks in parallel.

5.1.3 Float-to-code swapping procedure

The next step is to put the codes in the correct order. This is most easily done by swapping the floats for the corresponding Hash Map code. The swapping can be done in parallel by giving each thread the same number of symbols to swap and then having them swap at the same time. The algorithm uses OpenMPI system to run the conversion in parallel. This is shown in Algorithm 5.3.

```
SwapValues(float[] data, bool[][] codes)
BEGIN
        // calculate the number of values to process at a time
        int32 numToProcessPerBlock = ceil(NumberFloats / NumThreads)

        // initialise the compressor with the correct frequency map
        compressor(FrequencyHashMAP)

        initialise the compressor tree

        # OMP parallel for
        LOOP (The number of threads)
                checkthat the number to process does not exceed the array bounds

                // swap the codes
                call on the compressor to swap the symbols for their codes for the current block
        END FOR
END

// The compress method in Huffman compressor
compress(float[] data, bool[][] codes, int32 num)
BEGIN
        FOR (int32 i = 0; i < num; ++i)
                codes[i] = CodesHashMap[data[i]];
        END FOR
END
```

Figure 5.3: Pseudo code for the Huffman Coding swap values method. This method swaps all the symbols for its respective binary sequence in parallel.

5.1.4 Binary Sequence to Char conversion

The final steps are to convert the boolean sequences into char arrays and place the required data into a file. There are 2 ways to do this: A Dynamic programming approach, and a prefix sum flattening parallel approach[16] which is shown to work very well.

The Dynamic Programming approach constructs bytes from the binary sequences in parallel using the 2 dimensional binary sequence array. It moves through each array in the 2D binary sequence array one bit at a time. Adding that bit to the current byte, until 8 bits have been added to the current byte. It then adds that byte to the byte array and starts with a fresh byte until there are no bits left. Algorithm. 5.4

```
ConvertToChar(bool[][] codes, char[] charArray)
BEGIN
        int32 count = 0;
        char c = 0;
        LOOP (the number of float symbols in the data)
                LOOP (all the bits in the current code)
                        // create the char bit by bit
                        IF (codes[i][z] == 1)
                                SHIFT 1 to indexed position
                        ELSE
                                SHIFT 0 to the indexed position
                        END IF

                        cnt++; // move to next bit

                        IF (a complete char has been constructed)
                                add the char to our output array
                                refresh the char
                        END IF
                END FOR
        END FOR
END
```

FIGURE 5.4: Pseudo code for the Huffman Coding Char Conversion method which converts the 2 dimensional binary sequence array into a byte array using a Dynamic Programming scheme.

This dynamic programming method was determined to be the slowest part of the entire algorithm, and thus causes a large bottleneck. We then decided to try the Prefix Sum flattening parallel approach found in [16]. This method uses a prefix sum to allow for a fast parallel process to flatten the 2D binary sequence array into a 1D binary sequence array. STL's in-place copy method was used to efficiently do this in parallel. Once the flattened array is constructed, the 1D binary sequence array can be converted into a byte array in parallel, shown in Algorithm. 5.5.

The use of a linear prefix sum for the CPU rather than try to make it parallel is from the Article. [17]. The authors of the article state that the CPU parallel prefix sum tends to slow down the prefix sum operation and thus a linear approach is faster on the CPU.

This version of the char construction method is around 5 times faster than the Dynamic programming version, which ensures the final CPU-only algorithm is more than 3 times faster.

The file is then constructed by writing the tables for the data, and then dumping the byte array from memory into the file, to save on writing costs.
In the blocked scheme there will be multiple tables and the charArrays will have to be constructed separately for each table so that the decompression can differentiate between the codes generated by each tree.

5.1.5 Decompression

The decompression process receives the tables and the character arrays. The parallel version of the algorithm will only have 1 table and 1 char array and can thus convert the char array back into boolean arrays

```
ConvertToChar(bool[][] codes, char[] charArray)
BEGIN
        // The prefix sum is first calculated
        // note an extra length is added to determine the total
number of binary values
        int32 [] prefixSum(codes.length + 1);

        prefixSum[0] = 0;
        int32 sum = 0;
        LOOP (the length of the code)
                sum += codes[i].length;
                prefixSum[i+1] = sum; // calculate the prefix
        END FOR

        // A flattened array of binary values is then calculated (remember padding)
        bool[] flatArray(ceil(prefixSum[codes.length]/8) * 8);

        # OMP parallel for
        LOOP (the number of codes in the 2D array)
                copy the code to the correct array starting position
        END FOR

        // finally construct the byte array

        # OMP parallel for
        LOOP (number of bytes to create)
                unsigned char c = 0;

                FOR (int z = 0; z < 8; ++z)
                        IF (flatArray[i << 3 + z] == 1)
                                SHIFT 1 to the index
                        ELSE
                                SHIFT 0 to the index
                        END IF
                END FOR

                charArray[i] = b;
        END FOR
END
```

FIGURE 5.5: Pseudo code for the Huffman coding char conversion method. This version of the char conversion method, uses a linear prefix sum to help flatten the 2D binary sequence array in parallel, for a parallel byte conversion.

in parallel by converting each char at a time. It then uses the same procedure as compression to construct the tree, this time neither deleting the tree nor constructing the hash map. The decompression procedure discussed in Chapter. 3 can then decompress the data.

For the blocked Huffman Coding version, each block can decompress at the same time but the conversion to char array will have to be serial for each block.

5.2 GPU and CPU combined Huffman Scheme

The GPU and CPU combined version of the code was implemented to find out if certain sections of the code could be accelerated by using the GPU rather than the CPU.

```
ConvertToChar(Thrust::device<float> data, Thrust::device<float> uniqueVals,
Thrust::device<int32> counts)
BEGIN
        // sort the data
        Thrust::sort(data.begin(), data.end());

        // once the data is sorted, number of unique values can be calculated
        int32 numUnique = Thrust::inner_product(data.begin(), data.end() -1
data.begin() + 1, int32(1), Thrust::plus<int32>(), Thrust::not_equal_to<float>());

        // set the size of the device arrays
        uniqueVals.resize(numUnique);
        counts.resize(numUnique);

        // find each unique value and count it.
        Thrust::reduce_by_key(data.begin(), data.end(),
Thrust::constant_iterator<int>(1), uniqueVals.begin(), counts.begin());
END
```

Figure 5.6: Pseudo code for the Thrust binning method. This method is to replace the CPU binning method to both speed up the compression as well as increase the compression ratio since fewer tables will be needed for each 5GB chunk.

5.2.1 Binning

The first section of code that could be parallelised is the binning process. There are many binning algorithms for CUDA. Thrust is a library for CUDA designed by NVIDIA which has all the normal c++ STL (Standard Template Library) algorithms and many more algorithms already available for use. A common use for Thrust is binning, as it is usually faster to bin data on the GPU than the CPU.

In order to use the Thrust binning algorithm most effectively, the full 5GB block that arrives should be binned rather than be split into blocks. This changes the swapping section of code to be the same as the non-blocked parallel CPU algorithm, where the compressor and its initialisation step are created outside of the swapping for-loop. Another change had to be made in the compressor class: the initialisation step should now *not* bin the data, as this is being done through Thrust calls.

The Thrust binning algorithm uses the original data, and 2 extra arrays for the unique symbols and the counts. The Thrust library provides both a device (GPU memory) array class as well as a host (CPU memory) array class that can interact with on another. This interaction also allows for easy memory copy to and from the device.
The algorithm as explained in the Chapter. 3 is shown in Figure. 5.6

The Thrust binning method finds all unique values and counts the number of unique values through the following procedure:

1. First the data is sorted to place all unique symbols together and in consecutive order.
2. A Thrust *inner product* is used to count how many unique values there are in the file, this is done by increasing an integer value by one each time a value is not equal to the one before it (since the data is sorted, it counts all unique symbols).
3. Finally a *reduction* operation is done on the data: Each time a new symbol is found, it is placed in the uniqueVals array, and a counting iterator is placed in the counts array, it then adds to that iterator each time that symbol is seen before.

Since the Thrust binning algorithm has many synchronisation issues, test show that regular Gaming graphics cards, GT/GTX series of NVIDIA graphics cards, beat the specialised, Tesla NVIDIA series, graphics cards in speed when running this CUDA code. This is most likely because the Gaming cards have much higher clock speeds than the TESLA cards. Even though the TESLA cards have allot more SMs and threads per SM, the synchronisation in the binning procedure causes most of the threads to halt and wait for another thread to release access to one of the variables. Thus, the higher clock speeds of the Gaming cards allow for the threads to release a variable in less time.

This Thrust binning procedure provides speed up of around 45% including all copy times to and from the Graphics card.
An important issue arose during the testing of this thrust binning algorithm. The Thrust::sort method is not an in-place sort, meaning it requires double the size of the data it is sorting in memory space. This is an issue for the SKA data as the current max memory size for NVIDIA graphics cards is 6GB and the SKA data arrives in 5GB chunks. There is thus insufficient memory space to sort the whole 5GB chunk. Two modifications were introduced to try and address this issue:
First, the data was sorted bit by bit using many thrust::sort calls on small blocks. Sadly due to each kernel call having an overhead time, the use of many thrust::sort calls caused the 45% speed up achieved to be lost.
The second modification attempts to fix the problem by utilizing Pinned memory. Pinned memory is the ability to link Host and Device memory together and allow copy between both the host and device to run on the fly during a kernel process. Thrust has the ability to use Pinned memory through the `Thrust::experimental::cuda::pinned_allocator` object. This is used for each host and device Thrust array declaration, to tell thrust to used pinned memory when dealing with the data in those arrays. This pinned memory allocation allows for the Thrust::sort method to sort data larger than half the size of the device memory, but causes the speed up to drop to 40% of the original CPU method, thus losing 5%.

5.2.2 Symbol to Code Swapping

The next parallel procedure that could be changed to a GPU algorithm is the process of swapping the floats to their respective codes. Many issues were found in the process of writing a kernel to do this swapping. In particular, CUDA cannot have pointer values pointing to other pointer values, only to data. This means that the Hash Map of symbols-to-codes cannot be sent as a data struct to the graphics card. We thus had to flatten the codes and symbols into separate arrays: an array for the floats, a long array for all the codes flattened after one another, and another array to state the starting position in the flattened code array for each unique symbol in the flattened float array.

After the flattening of all the arrays is done, the data could then be sent to the GPU and a kernel could be run. However since pointers could not be used, a hashing method is impossible. Each symbol had to be found by looping through the entire flattened float array. In the worst case, this means each thread has to loop through all the unique values before finishing.

Due to these issues, the kernel method was an average of 5 times slower compared to the normal CPU swapping procedure. Thus the GPU method for swapping was discarded.

5.2.3 Binary sequence to Character array conversion

The only part of the final prefix sum byte conversion procedure in the CPU-only method that could be changed to a graphics card method is the Prefix sum. Since the Prefix sum has been shown to work very well on the GPU[17] we decided to try the prefix sum using the given thrust method `Thrust::exclusive_scan`. The chang removes the linear CPU prefix sum and replaces it with the Thrust call as shown in Figure. 5.7.

The thrust implementation of the prefix sum achieves a speed up of around 26% compared to the CPU based character conversion method.

5.2.4 Feasibility of Huffman Coding

The final data rate achieved by the GPU-Huffman algorithm is around 71MB/s. Although this is much lower than the SKA 5GB/s required, tests show that it is much faster than the standard algorithms that are commonly used. The final average compression ratio for SKA data is 41% which is very good for long term storage of the data. Finally, as will be shown in the Chapter. 6, this compression ratio is better than any of the standard algorithms available.

This shows that GPU-Huffman is a feasible solution for long term storage of SKA data, but is not a good choice for pipe line compression to save on networking throughput rate.

```
ConvertToChar(bool[][] codes, char[] charArray)
BEGIN
        // The prefix sum is first calculated
        // note an extra length is added to determine the total
number of binary values
        int32 [] prefixSum(codes.length + 1);

        thrust::host<int32> prefixSum

        // copy the lengths to the host vector
        # OMP parallel for
        LOOP (the length of the 2D code array)
                prefixSum[i] = codes[i].length;
        END FOR

        // copy the lengths to device vector
        thrust::device<int32> prefixDev = prefixSum;

        // run the thrust prefix scan
        thrust::exclusive_scan(prefixDev.begin(), prefixDev.end());

        // copy the data back to the host
        prefixSum = prefixDev;

        // clear the graphics card memory as its no longer needed
        prefixDev.clear();
        prefixDev.shrink_to_fit();

        // A flattened array of binary values is then calculated (remember padding)
        // note the size of the last value does not include the last length
        bool[] flatArray(ceil((prefixSum[codes.length - 1] + codes[codes.length - 1].length)/8) * 8);

        // Rest of the code is the same as the CPU only version
END
```

Figure 5.7: Pseudo code showing the Huffman coding prefix sum char conversion. This version of the prefix sum char conversion method uses thrust to run the prefix sum in parallel rather than have it run linearly on the CPU.

Chapter 6

Results

This chapter covers many tests to determine if our Huffman coding algorithm is feasible for SKA data compression. Since external factors could cause incorrect results 2 requirements for testing were set:

- Each test must be run 10 times, with outlier times removed and all others averaged in order to exclude issues with other processes using the RAM or CPU cores allocated to the test process.
- Each test must be run on the same data, and hardware for each relevant test.

6.1 Tests

6.1.1 Feasibility of Huffman Coding in parallel

The first set of tests (Figure. 6.1) were run to determine if making Huffman Coding parallel is possible and worth the hardware requirements.
This test was run on a 1GB binary file of floating point data, with 1200 unique float values. Since this test was to determine the speed of process no output file was created and the time for reading the input file was discarded.
The UCT HEX cluster was used for this test. The machine specs for the HEX cluster is as follows:

- CPU - XEON E5-2650 @ 2Ghz (16 cores).
- RAM - 64GB of 1600MHz RAM.
- GPU - TESLA M2090 NVIDIA graphics card with 6GB DDR5 memory.

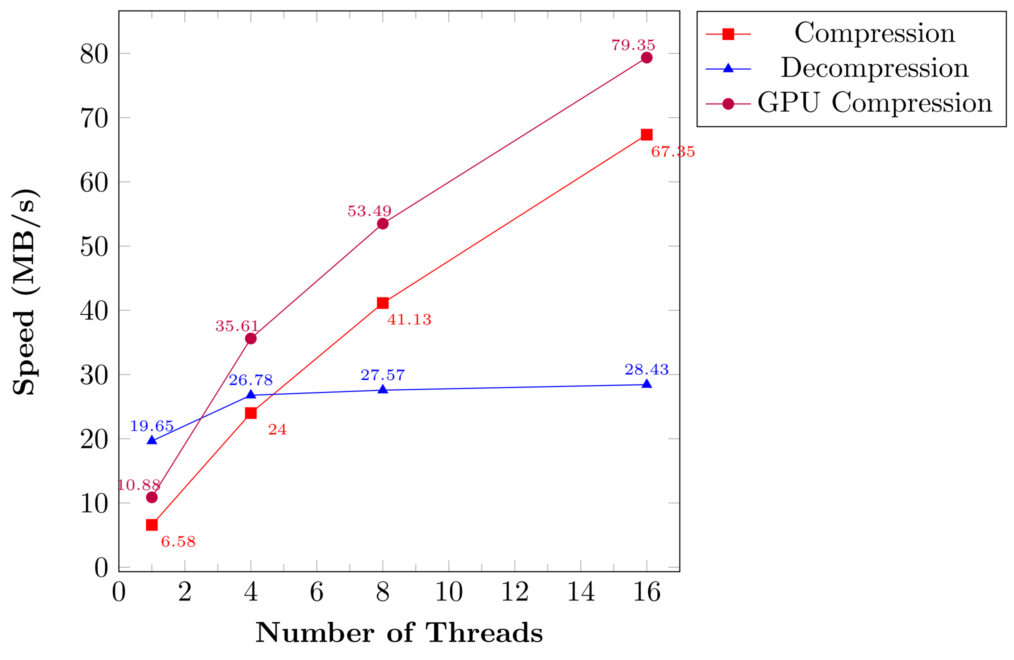

FIGURE 6.1: Graph showing speed up produced by increased threads for our Huffman Coding algorithm. This shows that parallelising the Huffman Coding algorithm is a feasible approach to increasing its throughput. And using the GPU to speed up small sections of the Huffman Coding algorithm does supply an increased throughput. It also shows that the decompression is only slightly accelerated by using parallelism.

6.1.2 Comparison to standard compression programs

The next test (Figures. 6.2 and 6.3) was performed to determine if the produced Huffman Coding algorithm performs better than compression algorithms already available on the market. Since the HEX cluster uses a networked hard drive and is not set up for high disk IO, these tests were run on a 1GB file of floating point data on a different system. The machine specs used to run these tests are:

- CPU - Intel Haswell core I7 4800MQ @ 2.7GHz (8 cores)
- RAM - 8GB of 1600Mhz RAM
- GPU - NVIDIA GTX 770M with 3GB DDR5 RAM
- Hard Drive: Western Digital 500GB hard drive (10000RPM)

6.1.3 Comparison of all SKA compression projects

The final test (Figures. 6.4,6.5 and 6.6) was run to compare all the currently running SKA compression projects, namely a Predictive Scheme, a RLE scheme and our Huffman Coding scheme. All algorithms in this test were run on the UCT HEX GPU cluster on a 756MB file of floats supplied by SKA.

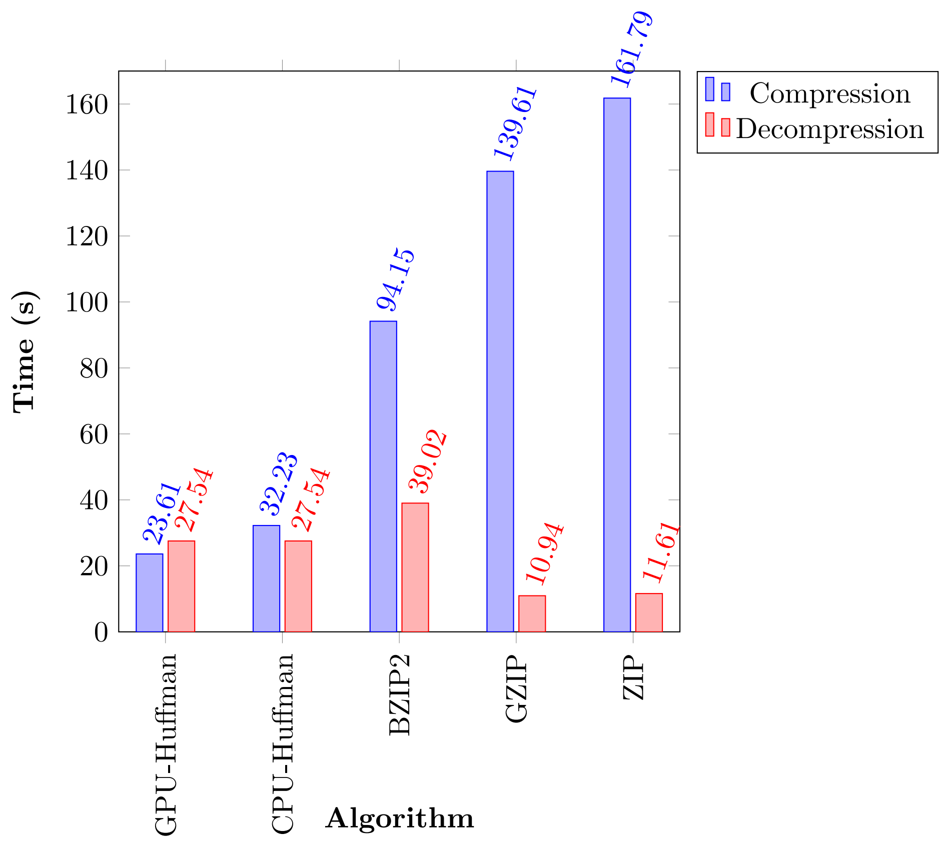

FIGURE 6.2: Graph showing compression times of GPU-Huffman and CPU-Huffman vs standard algorithms. This shows that both the CPU and combined GPU-CPU Huffman Coding algorithms compress faster than any standard compression tools available. It also shows that compared to BZIP2 it decompresses faster but it does not beat GZIP or ZIP in compression speeds. Our Huffman Coding algorithm is thus a much better algorithm for the SKA data

6.2 Discussion

6.2.1 Feasibility of Huffman Coding in parallel and for SKA

The first test shown in figure. 6.1 Shows that a near linear speed up for both the GPU and CPU compression is evident. This means that we can expect more threads to speed up the compression, up to a memory throughput limit. Decompression on the other hand, shows that the initial usage of multiple threads causes a huge speed increase. However since the only section of the algorithm that is parallel is the conversion from bytes read from the file to the full binary sequence, the speed up diminishes as we increase the number of threads.

Another noticeable feature is that the CPU compression speed is accelerating faster than the graphics card, this means that it may be possible for the high end Intel XEON Phi CPUs with threads greater than 100 could process the CPU only version of the generated Huffman Coding scheme faster than the GPU-CPU algorithm. Only testing on XEON Phi's and top Gaming Graphics cards together could prove

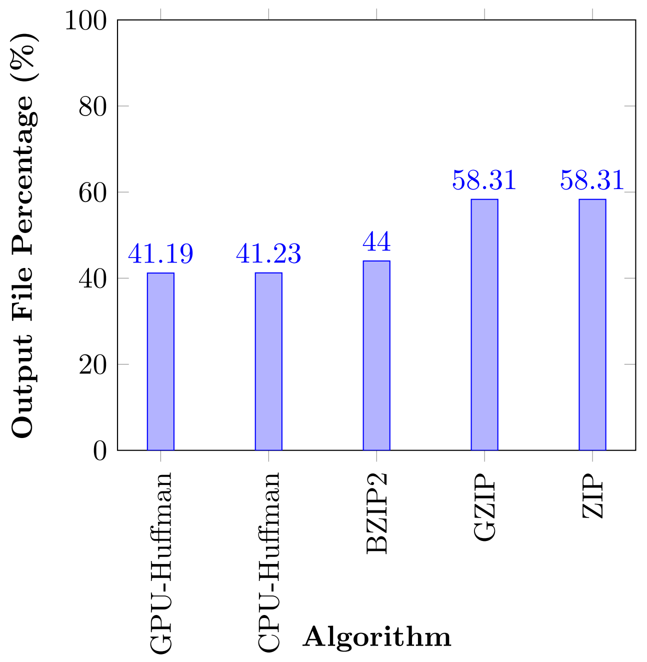

FIGURE 6.3: Graph showing compression ratios for GPU-Huffman vs standard algorithms. Both our Huffman Coding algorithms beat all the standard compression tools in compression ratio for the SKA data. The combined GPU-CPU method has a slightly better compression ratio than the CPU-only version since it only has to save 1 table and pad a single binary sequence unlike the CPU-only method. Huffman Coding is thus a much better algorithm for the SKA data.

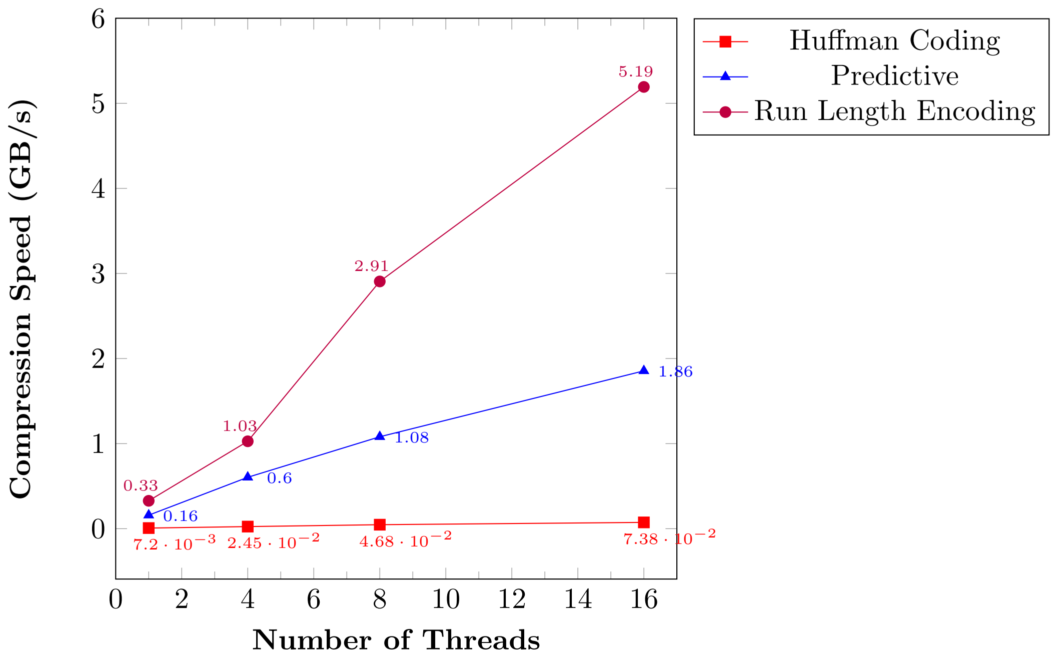

FIGURE 6.4: Graph showing compression speeds for all SKA research Projects. RLE is the fastest algorithm achieving the required 5GB/s throughput. The Huffman Coding algorithm is the slowest of all three algorithms. This shows that for SKA's required throughput RLE is the best candidate.

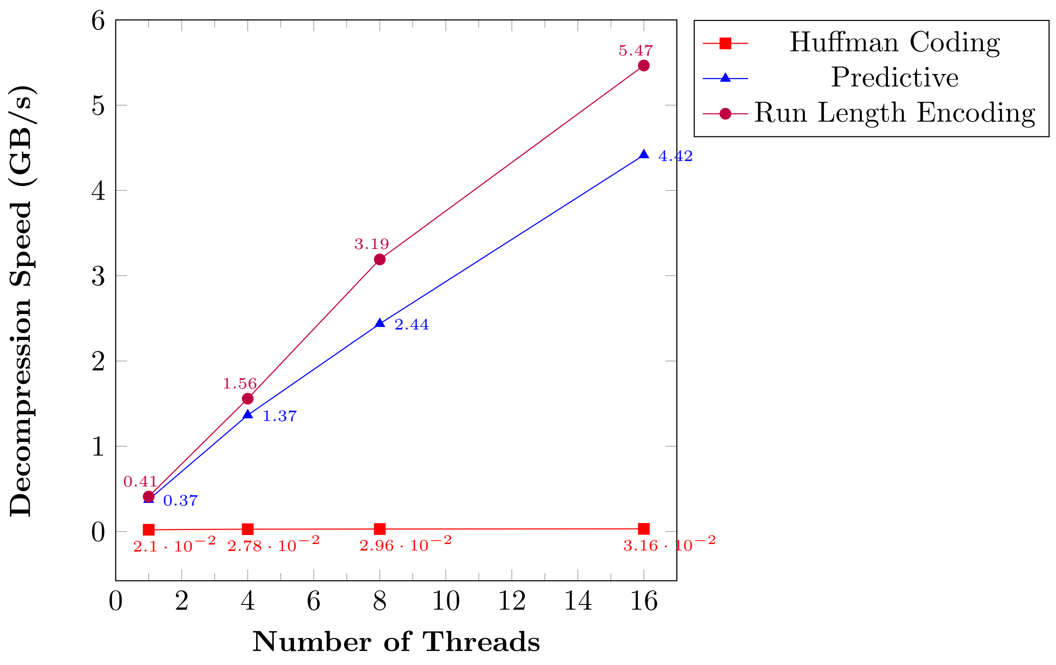

FIGURE 6.5: Graph showing decompression speeds for all SKA research Projects. RLE and Predictive schemes are greatly accelerated through the use of parallelism while Huffman Coding decompression tends to stay very linear. This shows that the Predictive and RLE algorithm can decompress the data faster than it was compressed, while the Huffman Coding algorithm tends be slower than its compression speed.

this, this is left for future work. However, one should note that when using the CPU only algorithm the table increases in size according to number of threads. The table size of the CPU only algorithm is: $NumThreads * numSymbolsUnique * 2 * 4 + 4$ bytes, thus the compression ratio is reduced the more threads used.

The top speed achieved is 79.3MB/s with 16 threads on a Tesla M2090 graphics card. If we assume that increasing the number of threads continues to cause a near to linear speed up, then 64 threads will achieve approximately 120MB/s and 128 threads around 140MB/s. This means with current hardware it is impossible to reach the transfer speed of the SKA data, which is 5GB/s. Thus Huffman Coding can not be done in real time for SKA data.

6.2.2 Comparison to Standard Compression programs

The next test was done to determine if this Huffman Coding algorithm is better than any currently available to the SKA. Figures. 6.2 and 6.3 show the findings. The algorithm was tested against the top commonly used public available algorithms: BZIP2, GZIP and ZIP. As the graph shows, the GPU and CPU versions both beat all the standard algorithms in both speed and compression ratio. Both new algorithms are around 3 times faster than BZIP2 and slightly better in compression ratio. However the decompression speeds only beat the BZIP2 algorithm whereas GZIP and ZIP tend to decompress the data very quickly, but have

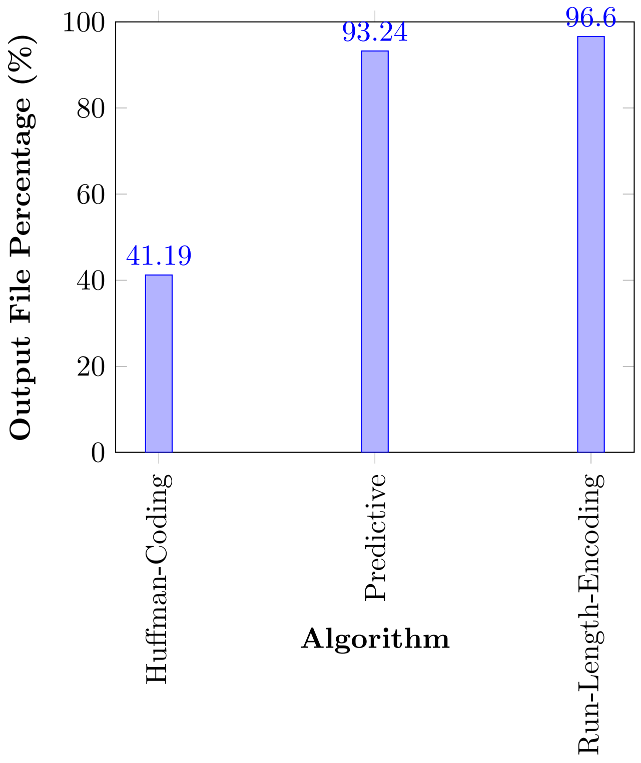

Figure 6.6: Graph showing compression ratios for all SKA research projects. Our Huffman Coding algorithm compresses the file allot more than both the Predictive and RLE algorithms, having over double the compression ratio. This also shows that the Predictive and RLE methods have a small compression ratio and thus will not be good for long term storage. Our Huffman Coding algorithm is thus the best candidate for SKA to use as along term storage compression tool.

terrible compression ratios. An interesting finding was that GZIP and ZIP compress .H5 (HDF) files faster than BZIP2 but BZIP2 compresses binary stored floats and plain text floats faster than GZIP and ZIP. The test was run on a file were the floats were stored as binary data due to this fact.
This shows that the created Huffman Coding scheme for SKA is much better than any standard compression program available to SKA currently, since it provides better speeds and compression ratio.

6.2.3 Comparison of SKA compression projects

The final test was run to determine if the other 2 SKA project's algorithms, a Predictive scheme and RLE, could beat the generated Huffman Coding algorithm. Figures. 6.4 and 6.5 show that both the RLE and predictive scheme beat the Huffman Coding scheme in throughput, RLE being the fastest and Huffman Coding being the worst for both compression and decompression. Out of all three, RLE is the only algorithm to achieve the throughput equal to the SKA transfer rate (Note: The predictive scheme does achieve the required throughput on Intel Hasswell and AMD processors, as these processors have the *LZCNT* leading zero count machine instruction which causes a good speed up for the predictive method).

However Figure. 6.6 shows that the Huffman coding algorithm supplies over double the compression ratio compared to the Predictive and RLE schemes. To determine which of the three SKA projects is the best algorithm overall is thus much harder. The RLE algorithm is the fastest, but supplies the worst compression ratio. Huffman Coding is the slowest, yet supplies the best compression ratio by far. Taking the SKA pipeline into account, Figure. 2.1 - we notice that we could place either the RLE or Predictive scheme between the Complex pairs and the compute nodes. This is were the data is sent over a slow network from the station were the telescope antennae are, and the main offices. Since the RLE and Predictive scheme are really fast there won't be a bottleneck issue and it will speed up the transfer times, but a 93+% compression ratio is very poor for long term storage. The Huffman Coding algorithm can thus be used for long term storage since it achieves a very good compression ratio.

Chapter 7

Conclusion

This report set out to determine if using a Huffman Coding scheme could achieve high throughput on data produced by the SKA, which is currently under construction in South Africa. Once completed the SKA infrastructure will produce up to 1 Petabyte of frequency data every 20 seconds. A fast compression scheme with a good compression ratio is thus required to both decrease the required network bandwidth and reduce storage space requirements.
The research aims were to determine if:

1. Is it possible to create a SKA Real Time compression algorithm using Huffman Coding?
2. Is Huffman Coding a better choice to compression schemes already available?
3. Is the compression ratio worth any time overhead it produces?

Several parallel algorithms were produced, blocked and parallel CPU version as well as a combined GPU-CPU algorithm. The combined GPU-CPU algorithm was found to be the best of the three for both compression ratio and throughput (achieving 73MB/s throughput and a 41% compression ratio on a 16-core system) compared to the blocked scheme which was around ¾ the speed and a fractionally worse compression ratio. Several methods were used to accelerate the algorithm: parallel swapping of data, Thrust implemented binning with pinned memory, a prefix sum implementation for flattening arrays in parallel and removal of tree structures to use hash map look ups. Many hardware differences were noticed: Since the algorithm is memory bound, the speed of the memory and motherboard played a huge factor. Furthermore, since most of the GPU based processes have many synchronization issues, gaming graphics cards (which have high clock rated but fewer cores) run much faster than specialist GPU Tesla cards. Although all processes that could be parallelised were done so, our Huffman Coding algorithm could not achieve the required data throughput. Huffman coding can thus not be run in Real Time for SKA data.

Once investigations in improving throughput concluded, compression ratio was considered. Attempts to use break points in the binary sequences ended up both reducing compression ratio and slowing throughput due to the process of having to pad data for each and every binary sequence. We also realised that the blocked CPU-only scheme reduces the compression ratio as more threads are utilized, since each thread creates its own table and has to pad more than 1 binary sequence. The initial Thrust library implementation did not allow for the full 5GB chunk to be sorted in place on any graphics card. However, the experimental thrust pinned memory allocator, which allows for data any size below the total graphics card ram to be sorted in place, improved results. This meant that only a single table and single binary sequence per 5GB chunk is needed. The algorithm thus produces a 41.18% average compression ratio for SKA radio data, which is better than any other algorithm available. It also beats the standard compression tools such as BZIP2 in data throughput. This means that our Huffman Coding algorithm is a better choice than any compression tool currently available.

SKA is experimenting with two other compression methods: a Predictive scheme and an RLE method. Both the Predictive and RLE method have higher throughput rates than our Huffman Coding algorithm but had over half the compression ratio. They both achieved the required 5GB/s throughput rate, but the Predictive method has a higher compression ratio than the RLE method by about 3%. This shows that for long term storage our Huffman Coding algorithm is the best algorithm available, even though it has a low throughput compared to the Predictive and RLE schemes. However the 93% compression ratio on the Predictive scheme does not allow for high storage space reduction. But the RLE or Predictive scheme could be used to speed up any network throughput without causing any bottlenecks in the pipeline.

Future work would entail adding a Lempel-Ziv scheme to the Huffman Coder in order to compress patterns rather than single symbols. Furthermore, an FPGA based Huffman coding algorithm could be tested for higher throughput.

Bibliography

[1] Pei-Man Yang, Jen-Kuei Yang, Wen-Shan Wang, and Shau-Yin Tseng. Fast jpeg huffman table restoring and decoding for embedded dsp implementations. In *Multimedia, 2009. ISM '09. 11th IEEE International Symposium on*, pages 118–123, 2009. doi: 10.1109/ISM.2009.18.

[2] Debin Zhao, Y. K. Chan, and Wen Gao. Low-complexity and low-memory entropy coder for image compression. *Circuits and Systems for Video Technology, IEEE Transactions on*, 11(10):1140–1145, 2001. ISSN 1051-8215. doi: 10.1109/76.954501.

[3] Han-Chang Ho and S. F Lei. Fast huffman decoding algorithm by multiple-bit length search scheme for mpeg-2/4 aac. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 2844–2847, 2010. doi: 10.1109/ISCAS.2010.5536972.

[4] W. Balid and M. Abdulwahed. A novel fpga educational paradigm using the next generation programming languages case of an embedded fpga system course. In *Global Engineering Education Conference (EDUCON), 2013 IEEE*, pages 23–31, 2013. doi: 10.1109/EduCon.2013.6530082.

[5] D. Salomon. *Data Compression.: The Complete Reference.* Springer-Verlag New York Incorporated, 2004. ISBN 9780387406978. URL http://books.google.co.za/books?id=FlWjiShUst0C.

[6] J.L. Nunez-Yanez and V.A. Chouliaras. A configurable statistical lossless compression core based on variable order markov modeling and arithmetic coding. *Computers, IEEE Transactions on*, 54(11): 1345–1359, 2005. ISSN 0018-9340. doi: 10.1109/TC.2005.171.

[7] I. Chaabouni, W. Fourati, and M.-S. Bouhlel. An improved image compression approach with combined wavelet and self organizing maps. In *Electrotechnical Conference (MELECON), 2012 16th IEEE Mediterranean*, pages 360–365, 2012. doi: 10.1109/MELCON.2012.6196449.

[8] Jing Tang and W. Szpanskowski. Average profile of the lempel-ziv scheme for a markovian source. In *Information Theory. 1997. Proceedings., 1997 IEEE International Symposium on*, pages 311–, 1997. doi: 10.1109/ISIT.1997.613235.

[9] Zhiwei Tang. One adaptive binary arithmetic coding system based on context. In *Computer Science and Service System (CSSS), 2011 International Conference on*, pages 1440–1443, 2011. doi: 10.1109/CSSS.2011.5974482.

[10] K. Murashita, N. Satoh, Y. Okada, and S. Yoshida. High-speed statistical compression using self-organized rules and predetermined code tables. In *Data Compression Conference, 1996. DCC '96. Proceedings*, pages 449–, 1996. doi: 10.1109/DCC.1996.488381.

[11] Jennie Si, Lei Yang, Chao Lu, Jian Sun, and Shengwei Mei. Approximate dynamic programming for continuous state and control problems. In *Control and Automation, 2009. MED '09. 17th Mediterranean Conference on*, pages 1415–1420, 2009. doi: 10.1109/MED.2009.5164745.

[12] G. Lakhani. Modified jpeg huffman coding. *Image Processing, IEEE Transactions on*, 12(2):159–169, 2003. ISSN 1057-7149. doi: 10.1109/TIP.2003.809001.

[13] R.A. Patel, Yao Zhang, J. Mak, A. Davidson, and J.D. Owens. Parallel lossless data compression on the gpu. In *Innovative Parallel Computing (InPar), 2012*, pages 1–9, 2012. doi: 10.1109/InPar.2012.6339599.

[14] M. Kawahara, Y.-J. Chiu, and T. Berger. High-speed software implementation of huffman coding. In *Data Compression Conference, 1998. DCC '98. Proceedings*, pages 553–, 1998. doi: 10.1109/DCC.1998.672291.

[15] Roger Seeck. Binary essence compression library. `http://www.binaryessence.com`. Accessed: 2013-05-17.

[16] Molly A. O'Neil and Martin Burtscher. Floating-point data compression at 75 gb/s on a gpu. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 7:1–7:7, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0569-3. doi: 10.1145/1964179.1964189. URL `http://doi.acm.org/10.1145/1964179.1964189`.

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