

A comparison of compression techniques for scientific floating-point data.

Craig Thomson

School of Computing Science

Sir Alwyn Williams Building

University of Glasgow

G12 8RZ

A dissertation presented in part fulfillment of the requirements of the Degree of Master of Science at the University of Glasgow

7.9.2018

**Abstract**

Weather simulations use huge amounts of data from different sources such as temperature, pressure and humidity. In recent times it’s been seen that the performance of simulations on highly parallel, high processing bandwidth hardware such as GPUs and FPGAs massively improves performance, to the rate where researchers are actively looking to port parts or even complete codebases of weather simulators to programming frameworks, such as OpenCL, that support execution on these hardware accelerators. Given that these simulations use huge amounts of data, when executed on hardware accelerators there is usually a bottleneck created form the accelerator communicating with the CPU to receive more data that it couldn’t previously fit onto the accelerators on-board memory. This project investigates the performance of compression on data similar to that which is used for weather simulations. It evaluates it in terms of compression size and processing time on CPUs with the idea that this work could later tested on GPUs and later on implemented for a hardware accelerated port of a weather simulator.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic form.

Name: CRAIG THOMSON Signature: CRAIG THOMSON

Acknowledgements

Thanks to Wim Vanderbauwhede, my project supervisor for guiding me throughout the project and spending time resolving all of the queries and issues that I had.

Thanks to Waqar Nabi for taking time out of his busy schedule to help me with report drafts, especially when he helped over the weekend, outside of normal working hours.

Thanks to DXC Technology for allowing me the flexibility to balance working full time with the project and project meetings.

Contents

Chapter 1 Introduction 1

Chapter 2 Analysis 2

2.1 Literature Review 2

2.1.1 Weather Simulators 2

2.1.2 Using accelerators for weather simulators 2

2.1.3 General compression techniques 3

2.1.4 Scientific floating-point compression 3

2.1.5 Lossy compression and weather simulators 4

2.2 Data Analysis 4

Chapter 3 Design and Implementation 9

3.1 Run length encoding 9

3.2 ZFP 9

3.3 Fixed 24 Bit 10

3.4 Non-byte aligned 11

Chapter 4 Testing and Evaluation 14

4.1 Testing 14

4.2 Evaluation Criteria 14

4.3 Results 15

4.3.1 Compression ratios 15

4.3.2 Compression overhead 18

Chapter 5 Conclusion 20

5.1 Overview 20

5.2 Future Work 20

Chapter 6 References 22

Appendix A Run length pseudocode 1

# Introduction

Weather simulations have become a very important task that we rely on everyday for our decision making. Looking at the results of a weather simulation, for example by visiting the BBC weather website for your local area, can make you change your plans for the day or even the week. Weather simulations are more than just your local weather forecast, they can be used to forecast tropical cyclones and even for forecasting the weather in space [1]. Weather simulations are a problem within the field of Computational Fluid Dynamics (CFD) and are performed using mathematical models of different natural phenomena and huge amounts of input data.

Weather simulations were primarily run on CPUs but there’s recently been a move to accelerate simulations on heterogenous platforms involving GPUs and FPGAs, after performance improvements have been observed from their processing capabilities and their parallelism [2, 3, 4]. Parts of weather simulators have been ported to OpenCL [20] or NVidia’s CUDA framework for execution on heterogenous hardware. An issue with using device acceleration is the overhead of data transfer between the host CPU and device, or between the device and its external memory. While this is issues is relevant for GPUs, it is more pronounced for FPGAs which typically have a much slower off-chip memory bandwidth and have a very limited amount of on-chip memory. The throughput performance of simulations running on such acceleration devices thus tends to be memory bound.

This project investigates the performance of various compression methods on data like that used in weather simulations and CFD problems. Data is collected and analyzed from the Large eddy simulation (LES) and is compressed, decompressed and manipulated. The performance of the compression algorithms is accessed in terms of the overhead in size and time. The algorithms are written in C with the motivation that they can easily be ported to OpenCL for further testing and application.

The remainder of this report is laid out as follows: chapter 2 of this report contains a literature review of related concepts and details the analysis of the datasets used. Chapter 3 details the design, implementation and reasoning behind the algorithms implemented. Chapter 4 discusses the testing of the implementation and the evaluation of the results. Finally, chapter 5 provides concluding remarks and discusses future work.

# Analysis

This chapter sets the scene by exploring the background to weather simulators, hardware acceleration and compression, establishing the motivation to the project. Then a detailed analysis of the data collected and used for this project is presented.

## Literature Review

### Weather Simulators

Weather simulations have more applications than accurately producing short term forecasts of the days weather. In recent years weather has become more unpredictable and extreme due to global warming [5], which creates a demand for simulations to be accurate and run quickly, numerous times to keep up with the observed state of the area simulated, potentially saving lives and money by avoiding losses due to damage. By being able to simulate weather quickly, particularly in areas with impending natural disasters (e.g. cyclones) lives can be saved [6]. On a smaller scale, they can even be used by farmers to try to predict which crops to grow based on the predicted weather and climate [7].

Weather simulations are performed using numerical weather prediction (NWP) models, which utilize equations from physics, chemistry and fluid dynamics. Examples of two simulators are the Weather Research and Forecasting (WRF) model [8], used heavily in academia, and the Met Offices Unified Model [9]. WRF is primarily written in Fortran, it takes in input data, gets configured and then it runs its forecast model. The WRF receives data such as the simulation duration, latitude, longitude, temperature, air pressure and wind speed[a]. It’s given data for a specified area which is treated as a grid, it performs its various functions and transformations based on the input data and produces output for the given history intervals.

Large Eddy Simulation (LES) is a mathematical model for simulating atmospheric air currents [11]. It’s been used for simulating urban boundary-layer flows, which is the wind flow over urban areas with large obstacles such as tall buildings [12] and is used within NWP.

### Using accelerators for weather simulators

As previously mentioned, simulators were typically run on CPUs, but it’s recently been seen that performance can be improved when even just a part of a simulation is run using hardware acceleration with GPUs or FPGAs. It’s been seen that the scalar advection module of the WRF runs up to 7x faster on a GPU than on a CPU. Even with the expensive cost of transferring data between the host memory and the accelerator at the start of program execution, there was still a speed up of 2x for the code that ran on the CPU and GPU compared to just a CPU [2]. ASUCA is a weather model like WRF, ported to GPU produced similar results to its CPU counterpart with an 80-fold speedup [14].

Programming accelerators generally required specialized languages and frameworks. OpenCL [25] is a framework developed by the Khronos group for programming code for heterogenous systems. OpenCL provides an API for both C and C++ development. Within OpenCL the host machine sets up the memory objects to be transferred and “kernels”, which is code written for the accelerator, runs on the accelerator hardware.

### General compression techniques

Compression is the process of encoding information in such a way that the original file can be represented using less space and that the process or fully or partially reversable (lossy). There’re 2 different types of compression algorithms, lossy, where parts of the data or precision is lost, and lossless, where the decompressed information is exactly the same as the uncompressed version. Compression algorithms are judged on 2 main criteria, the compression ratio and the compression overhead (time to compress and decompress). Compression is extremely important for storing data, it makes the managing and transferring of data significantly faster. An example of a lossless data compression technique is run-length encoding [15]. Run-length encoding works extremely well on data which contains many repeated values. An example being the string “AAAAAAAAAA” would be converted to “10A”, cutting 10 characters down to 3. There’re many popular compression algorithms, each suited to a specific job. The ZIP file format is most suited to the lossless compression of files and directories, .JPEG is an extremely popular lossy compression method for digital images. More recent compression techniques utilize probabilities and statistics of the data to achieve better ratios [26].

### Scientific floating-point compression

As mentioned, different types of data are more suitable to different compression techniques. When dealing with data used in and extracted from weather simulators, there’s a large amount of it and it is highly precise and dense. Algorithms such as ZIP or JPEG wouldn’t perform well compressing it because they’re suited to different types of data. A successful algorithm (1.2 – 4.2x compression ratio) for compressing doubles uses a combination of 2 techniques; it attempts to predict the next value it’s trying to compress based on previous values, it then stores the difference between the prediction and the value [16]. ZFP is an open source library for compressing floating point arrays in either a lossy or lossless way achieving SOTA throughput [18]. ZFP works by breaking an n-dimensional array into independent blocks of 4^n where each block in handled individually from every other block. The values within these blocks are converted to integers and then transformed in a way like this used for JPEG compression, reordered and stored.

### Lossy compression and weather simulators

Due to the high precision of the values used in weather simulators one can expect that lossless compression methods are favored for compressing the data. Contrary to this, it’s been observed that for running the Lorenz ’95 model on FPGAs that it runs faster on the hardware accelerated platform and that there is not a significant increase in error when losing precision and representing data to a fixed-point representation [19].

## Data Analysis

To evaluate the performance of compression on scientific floating-point data there needs to be a dataset. This project uses data generated from a Fortran and OpenCL port of the Large Eddy Simulator (LES). As previously mentioned, the LES simulates wind flow over urban areas with obstacles such as buildings and skyscrapers. A project for the LES [20] was modified to print out the values of a pressure array at pre-defined time in the simulation e.g. one quarter complete, halfway through or complete. The code takes approximately 4 hours to execute on a 2nd generation Intel i5 processor. This code produces 2,025,000 floating point values to at most 9 decimal places, representing pressure values in a 150x150x90 3-dimensional space. The pre-defined time was modified in order to generate datasets at various different points of the simulation. These datasets were then stripped of comments and whitespace, leaving a 2,025,000-line file. See **Figure 1** below for a representation of the area represented.

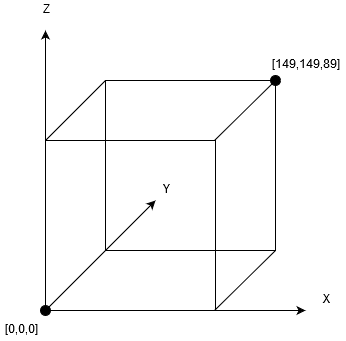


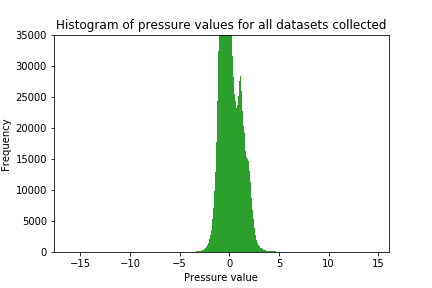
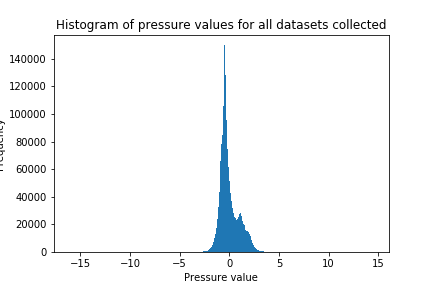
Figure 1 – 3D representation of pressure data extracted, values at z=89 represent pressure at top of the grid whilst values at z=0 represent pressure at bottom.

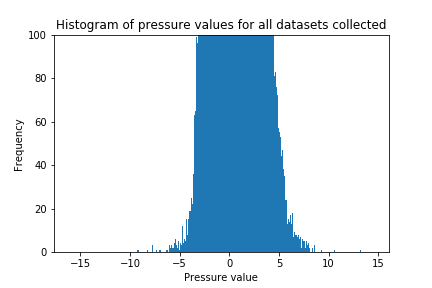
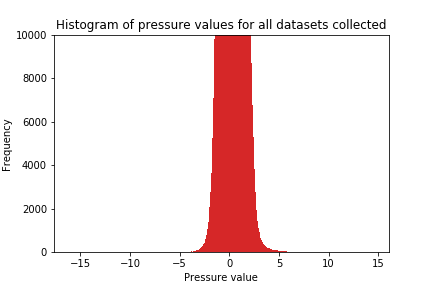
Data dumps from 10 different stages of the simulator were taken, each containing 2,025,000 32-bit floating point values, ~7.72Mb per file. Providing a large range of values to work with. A basic analysis was performed on each file to calculate values of interest such as the max, minimum and mean of the values in each data dump file, as seen in **Table 1** below.

|  |  |  |  |
| --- | --- | --- | --- |
| **Fraction of simulation complete** | **Min** | **Max value** | **Mean value** |
| 1/5 | -9.406430 | 11.404190 | -0.037463 |
| 1/4 | -16.119495 | 13.074379 | -0.039106 |
| 1/3 | -14.462990 | 13.204492 | -0.039257 |
| 2/5 | -9.083439 | 10.120451 | -0.033463 |
| 1/2 | -11.630094 | 12.651135 | -0.034482 |
| 3/5 | -10.504049 | 9.999895 | -0.035774 |
| 2/3 | -11.706923 | 12.863817 | -0.036299 |
| 3/4 | -10.953383 | 14.523329 | -0.036151 |
| 4/5 | -8.601760 | 11.882215 | -0.041041 |
| 1/1 (complete) | -10.930655 | 10.155290 | -0.037322 |

Table 1 – Analysis of values within datasets (data located in data/simulation\_datasets/)

From this point, all 10 data dump files were combined, and then various histograms were produced, allowing the distribution of values across all different states in the simulation to be visualized, seen in **Figure 2, 3, 4 and 5** below.





Figures 2, 3, 4 & 5 – Histograms of pressure values from all datasets with varying limits on frequency y-axis.

It can be seen that there’s a large distribution of values, primarily between -5 and 7.5, but with infrequent (<5 frequency) outliers such as -16 and 14, but there are a large number of very similar and repeated values. In addition to the histograms, 3D heatmaps were produced for the dataset representing pressure values halfway through the simulation. In order to see the trends in values at different levels, 3 different parts of the simulated area were cross sectioned and plotted. The areas cross sectioned were at the top of the area (z=89), the middle (z=45) and the bottom (z=0). The dataset used was for halfway through a complete simulation, this was because the values within it should be closer to the values present throughout all stages of the simulation. See **Figures 6, 7 and 8** below for the heatmaps.

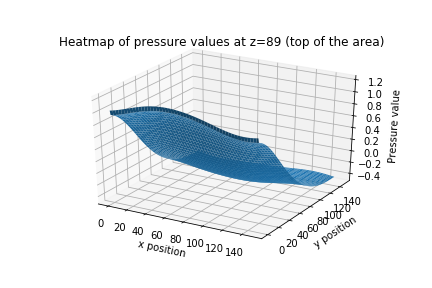


Figure 6 – Heatmap of pressure values at the top of the 3-dimensional area for the dataset halfway through a complete simulation.

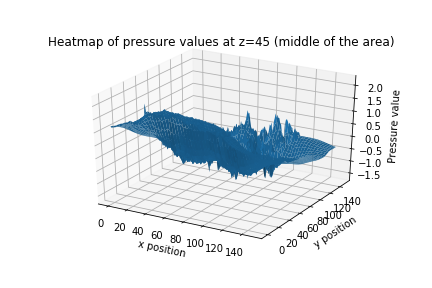


Figure 7 - Heatmap of pressure values in the middle of the 3-dimensional area for the dataset halfway through a complete simulation.

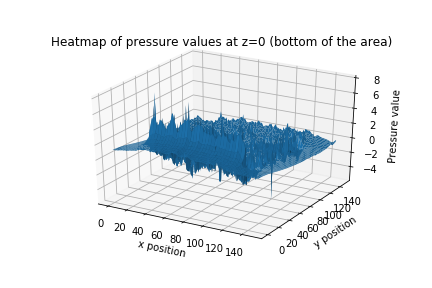


Figure 8 - Heatmap of pressure values at the bottom of the 3-dimensional area for the dataset halfway through a complete simulation.

It can be seen from the heatmaps that the values at the top of the 3D area, z=89, are quite similar and uniform, varying between ~-0.4 and ~1, and there is a trend of the values becoming smaller, a net change within the cross section of approximately ~-0.6. Whilst closer towards the ground, at z=49, the pressure values become much more unpredictable, varying from approx. ~1.1 to ~-1.4 and the values following a less uniform negative curve. At the ground level, z=0, values range massively from ~-4 to ~8 and are very unpredictable, where pressure values rapidly reach peaks and then dropping.

Following on, is chapter 3 which discusses the choices made for compression algorithms and their design and implementation.

# Design and Implementation

For this project all code was developed in C. Development was performed in C because of its high performance and so that if time permitted, the code could be ported to OpenCL and the compression performance could be evaluated further using a hardware accelerator. More details regarding an OpenCL port are in the Future Work section of Chapter 5. The code was written in a specific way trying to minimize dynamic memory allocation (malloc and calloc) because OpenCL doesn’t support memory allocation in this way, so the code could be more easily ported to OpenCL without too many major changes. This chapter covers the compression algorithms that were implemented as well as the reasoning for their choice and provides a high-level description of their implementation.

## Run length encoding

The first algorithm that was chosen for implementation and evaluation was run length encoding. It’s a lossless algorithm best suited to data with many sequential values that are the same. As it can be seen from the heatmaps generated, there is very little change in value at the top and middle level. This makes run length encoding appear as a candidate for this dataset that could achieve a good compression ratio (uncompressed size / compressed size). The pseudocode for a very simple implementation of run length encoding can be seen in **Appendix A**.

The implementation for run length encoding contains only slightly differences from the pseudocode. The methods in the source code for run length encoding are:

1. getRunlengthCompressedData(...)
2. getRunlengthDecompressedData(...)

In addition to the 2 functions for compression and decompression, a struct was made that simply contains a floating-point value and the number of times it’s exactly repeated.

## ZFP

Due to the promising compression ratios detailed in [18] for ZFP, basic compression functionality was implemented to see its performance on the LES datasets. Only basic compression functionality was implemented, this was because the ZFP C API only provides functions for whole compression and decompression amongst other things. Only the C++ API provides functionality to decompress individual indices in the compressed data structure. Nonetheless, ZFP was implemented so that the compression ratio for this particular type of data could be investigated. Basic functionality was implemented, most of the code was taken from ZFPs examples code on GitHub [20]. The sample code [20] was modified to handle 3 dimensional arrays of floats so that the compressed size would be returned. A single method was implemented for ZFP compression, the signature can be seen below.

1. zfpCompress(...)

## Fixed 24 Bit

After lossless compression techniques were implemented, lossy techniques were investigated, after seeing that losing precision for simulations doesn’t have an extremely detrimental effect on the end results [19]. The approach taken to lossless compression for this data was by keeping to byte aligned values, some precision could be lost to save a byte, results in 24 bits being used to store the compressed values.

By looking back at the analysis of the data, the largest value in the dataset was 14.523329 (rounded to 6 places from the ¾ simulation data) and the smallest value was -16.119495 (rounded and from ¼ simulation). By observing the maximum positive and negative values, if using 1 bit to represent sign, then only 5 bits are needed to represent any magnitude value. This would then leave 18 bits to represent the precision value. The precision value was treated like an integer as well. By using 18 bits, the largest value that could be represented is 262, 143. By attempting to represent values in 6 digits precision would be lost for anything above 262k, so the value after the decimal place to 5 digits was stored. This method can store any value in the dataset accurate to 5 decimal places and achieving a compression ratio of 1.33x. The algorithm works by taking the floating-point number and converting it into 2 separate integers, representing the values to be stored before and after the decimal place and the sign bit is stored. From here 3 bytes are used for storing the compressed number and the values for the sign bit, value before decimal and value after are shifted into place within the 3 bytes. Pseudocode for the compression steps can be seen below in and a worked example can be seen in **Figure 9**.

**for** each float value

**extract** the sign bit

**split** the float into 2 integers, representing the before and after decimal point

**shift** sign bit into left most byte of 3-byte array

**for** each bit of magnitude (before decimal value)

**shift** magnitude into 3-byte array

**for** each bit of precision (after decimal value scaled up)

**shift** precision into 3-byte array

**Value**: -15.08325

**Sign**: 1 = 0001

**Before decimal (5 bits)**: 15 = 1111

**After decimal (18 bits)**: .08325 \* 100,000 = 8325 = 00 0010 0000 1000 0101

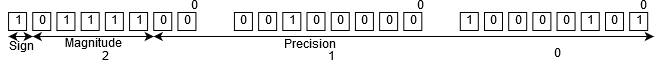


Figure 9 – Worked example of 24-bit compression for the value -15.08325. Values for before and after decimal are treated as integers and then the binary representation can be seen. After decimal is multiplied by 100,000 because that scales the value up to the limit of decimal places that can be stored. Layout within bytes can then be seen. The compressed representation starts being inserted into the most significant bit of the most significant byte (the left).

The code was implemented taking in the number of bits for the magnitude and for the precision so that they can be varied as desired, which allows the code to be catered to its specific datasets. There are 4 methods which have been implemented that are used for 24-bit compression and decompression, they are:

1. get24BitCompressedData(...)

1. get24BitDecompressedData(...)
2. getSingle24BitValue(...)
3. insertSingle24BitValue(...)

As can be seen, there’s a method for the compression and decompression of an array of values and functions for supporting single value compression and decompression from an array. Decompressing from 24-bit values back to floats works in the reverse way of compression. It takes in parameters such as the magnitude and precision value and works on a loop to unzip the 24 bits until there’s no target bits left.

## Non-byte aligned

An issue with compressing down to 24 bits is there’s potential to be using necessary bits for the storage. For example, with the current dataset if it’s known that there is an acceptable error with using values to 3 decimal places, then 5 bits are still needed for precision but only 10 for the precision. In total this would use 16 bits and provide a compression ratio of 2.0x. To support situations where the number of bits for compression isn’t a whole number of bytes, an algorithm that supports compression across byte boundaries is required.

The implementation for non-byte aligned compression is very similar to 24-bit compression. The main difference between them is that when performing non-byte aligned compression and decompression, there are additional variables needed to keep track of the current byte that’s being worked on, both compressed and decompressed as well as the current space available in the compressed data structure and in the floating-point value being compressed. In addition to this there’s additional conditional blocks that are used to decide whether its bits on the left of the current value’s binary representation, or bits on the right of it that need to be inserted to the compressed structure. The pseudocode for the high-level parts of the compression algorithm can be seen below.

**for** each value to be compressed

split float into 2 integers

insert sign bit

**if** sign bit fills up current compressed byte move to the next

**while** there’s bits of magnitude needing compressed

**if** bits left > space in byte

shift magnitude bits to the right and store

**else**

shift magnitude bits to the left and store

**while** there’s bits of precision needing compressed

**if** bits left > space in byte

shift precision bits to the right and store

**else**

shift precision bits to the left and store

Single value recompression works in a similar way to the list compression. The core difference being that the starting byte and index of the nth number that’s being updated needs to be calculated. Once the starting byte and index are known, the number of bits used for the number’s storage are set to 0 and then the new value is inserted, starting from the pre-calculated start byte and index. When performing decompression there’s an extra step with each extraction which is to use bitmasks to extract the certain bits targeted, this is because before the compressed value, there can be other values before and after, the magnitude before etc. and the bits for these that overlap in the same byte being worked on need to be ignored. This situation can be seen in **Figure 10** below.

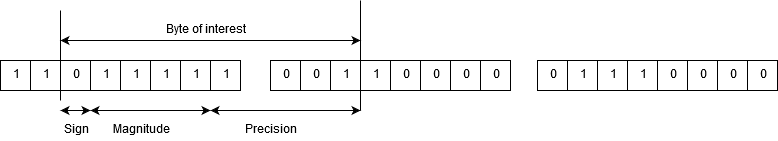


Figure 10 – Example of layout in memory of a non-byte aligned compressed value, showing the precision being spread across bytes and the value starting in a non-zero index of a byte.

There are 4 methods used to deal with non-byte aligned compression and decompression, which are listed below.

1. getVariableBitCompressedData(...)
2. getVariableBitDecompressedData(...)
3. getVariableBitDecompressedValue(...)
4. insertVariableBitValue(...)

The following section details the testing performed on the compression functions written. It also discusses the evaluation performed and presents the results produced.

# Testing and Evaluation

This chapter details the testing that has been performed on the created compression and decompression functionality. It then follows on to discuss how the functions were evaluated and presents the results.

## Testing

Unit tests were created for all compression and decompression functionality and for non-trivial helper methods, such as reading file data in. All unit tests are contained within the tests file and the unit tests work on data within the test\_datasets folder. Testing was performed using custom data because for most functionality created it would be too time demanding to verify the results produced are correct when working with the test files containing over 2 million values.

The primary focus of testing was on the 24-bit compression and non-byte aligned compression/decompression algorithms. For both 24 bit and non-byte aligned tests there are accompanying datasets and verification files. There are many datasets for each compression algorithm, each of which use different sizes of magnitude and precision for the compressed representation. By using different sizes, all conditional areas within the implementation are reached and the maximum values that can be represented both the positively and negatively can be tested.

Testing was performed using a minimal unit testing framework for C, called Minunit [13], which is contained in a single header file. The framework supports a range of testing functionality, but only asserts are used. For the 24 and non-byte aligned compression tests, the test data is read in and converted to compressed representation. The expected values of the unsigned bytes that are used for the compressed notation are taken from the expected file then compared to the result on a byte-by-byte basis. There’s a total of 12-unit tests provided in the tests file.

## Evaluation Criteria

There are two key aspects in which the compression methods previously detailed have been evaluated. The first way that the compression methods are evaluated is their compression ratio, which is a measure of how much the data is reduced. This is calculated from (uncompressed size / compressed size), the higher the number the better. The second criteria measured is the compression overhead, which is a value representing the cost of performing compression and decompression. The compression overhead is evaluated using various methods. It’s evaluated in the average time to compress all datasets, the average time to decompress all datasets and the effect of compression on a task. The use of a task for evaluation is because in the target environment, hardware accelerators, the datasets would be given to them already compressed, the operations they perform are many compressions and decompressions.

The task performed for analyzing overhead utilized all 10 datasets concurrently, it’s a function which performs a transformation on the values within them. All compression algorithms except ZFP and run length are used for this task. For every [x][y][z] combination of values (all 2,025,000 values) a transformation is performed. All 10 datasets are transformed at the same time, by performing the transformation for the value [i][j][k] in datasets 1 to 10, and then incrementing the k value, j and i. This results in over ~100,000,000 values being compressed and decompressed. The transformation algorithm run is a simplified version of successive over-relaxation algorithm, which can be seen below.

float tmp = (

value[i+1,j,k]

+value[i-1,j,k]

+value[i,j+1,k]

+value[i,j-1,k]

+value[i,j,k+1]

+value[i,j,k-1]

)/6.0

value[i,j,k] = value[i,j,k] + tmp;

As can be seen above, over 100,000,000 values would be required to be compressed and decompressed for this algorithm across all datasets due to the fetching of values surrounding the current value, in 3-dimensional space. The algorithm is suitable for evaluation because it represents the case that can occur in simulators, where transformations use data from different collections that may not be in cache, resulting in the slower fetching from on-board memory. This algorithm accesses values in 3-dimensional space that are beside the current value but physically can be stored a large distance apart.

## Results

In this section the results of the compression algorithms are presented. Firstly, the compression ratios are presented and then the overhead for the previously described task is analyzed.

### Compression ratios

The first part of the performance that was evaluated was the compression ratios, which is a measure of how much the data can be reduced to. Initially the uncompressed size was calculated, from the number of numbers multiplied by 32. Then each compression method was run, and the size calculated within the program. The results can be seen in **Table 2** below.

|  |  |  |  |
| --- | --- | --- | --- |
| **Compression technique** | **Size (bytes)** | **Size (mb)** | **Compression ratio** |
| None | 8,100,000 | 7.72 | N/A |
| Run length | 16,199,992 | 15.45 | 0.50 |
| Lossless ZFP | 5,662,768 | 5.40 | 1.43 |
| 24 bits (5 mag, 18 precision) | 6,075,000 | 5.79 | 1.34 |
| 21 bits (non-byte aligned, 5 mag 15 precision) | 5,315,625 | 5.07 | 1.52 |
| 18 bits (non-byte aligned, 5 mag 12 precision) | 4,556,250 | 4.34 | 1.78 |
| 15 bits (non-byte aligned, 5 mag 9 precision) | 3,796,875 | 3.62 | 2.13 |
| 12 bits (non-byte aligned, 5 mag 6 precision) | 3,037,500 | 2.90 | 2.67 |

Table 2 – Sizes of compressed dataset and compression ratios for dataset representing pressure values halfway through a simulation.

In terms of lossless compression schemes, lossless ZFP massively outperforms run length encoding. Even though there are many very frequent similar values (**Figures 2, 3, 4** and **5**) and the values are certain levels of the 3D space are similar at the top level, there doesn’t appear to be any common exact repetition of values. This causes run length encoding to perform extremely poorly because the representation for each compressed value is built from an integer represent the number of repeats and a float for the value. Lossless ZFP compression also outperforms the non-byte aligned lossy compression until the 5-bit magnitude, 15-bit precision results, once <=11 bits are saved per value, lossy compression outperforms ZFP. The lossy compression results for non-byte aligned demonstrate the net saving that can be achieved by losing 1 decimal place of precision, which is roughly 3.5 bits. By losing approximately 1 digit in precision, 0.72mb can be saved, approximately 10% of the original uncompressed size is the net reduction per loss of a digit of precision. The results of compression techniques have also been represented graphically, see **Figures 11** and **12** below.

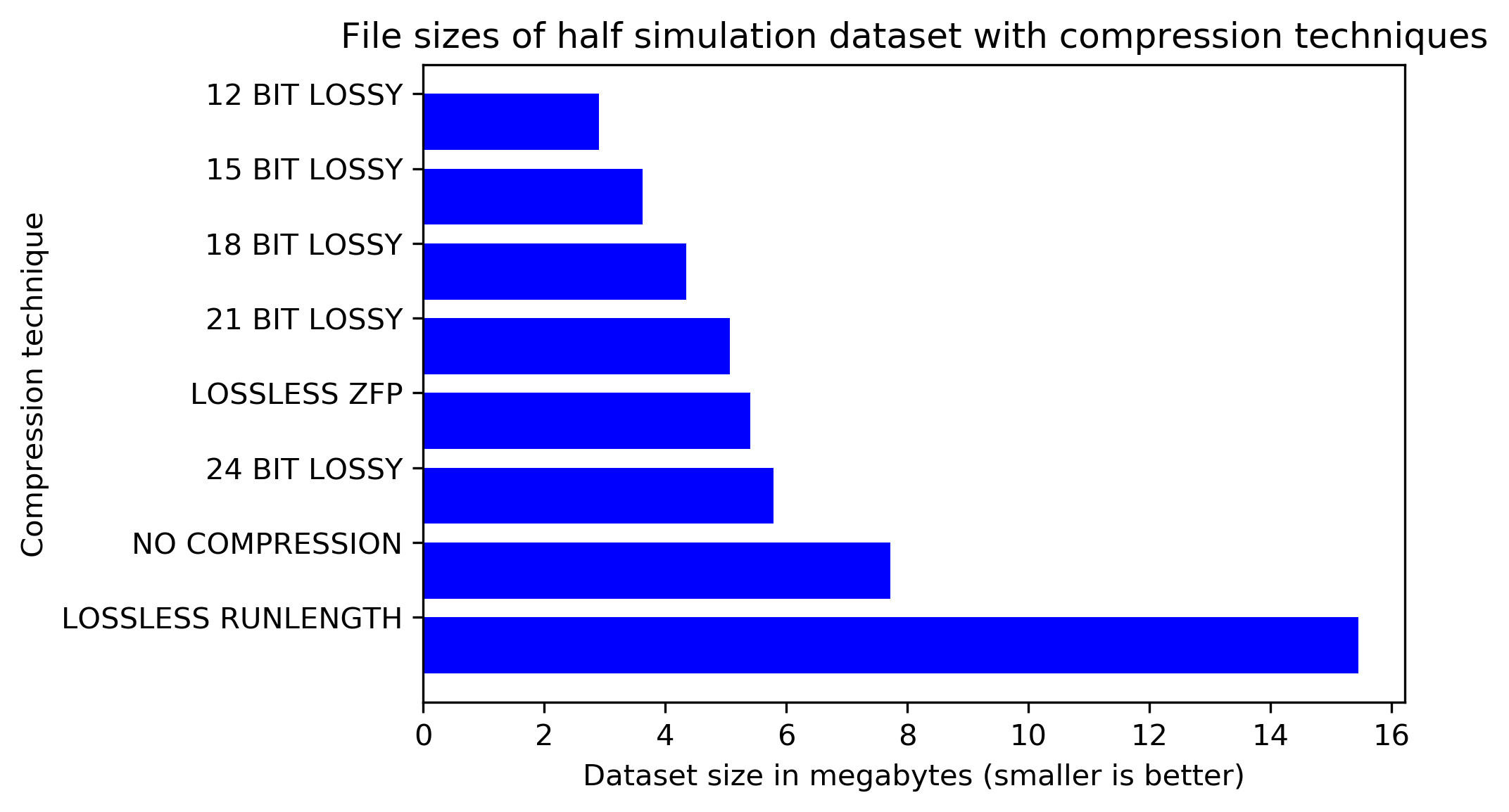


Figure 11 - Graphical representation of file sizes after compression for the dataset for halfway through a simulation.

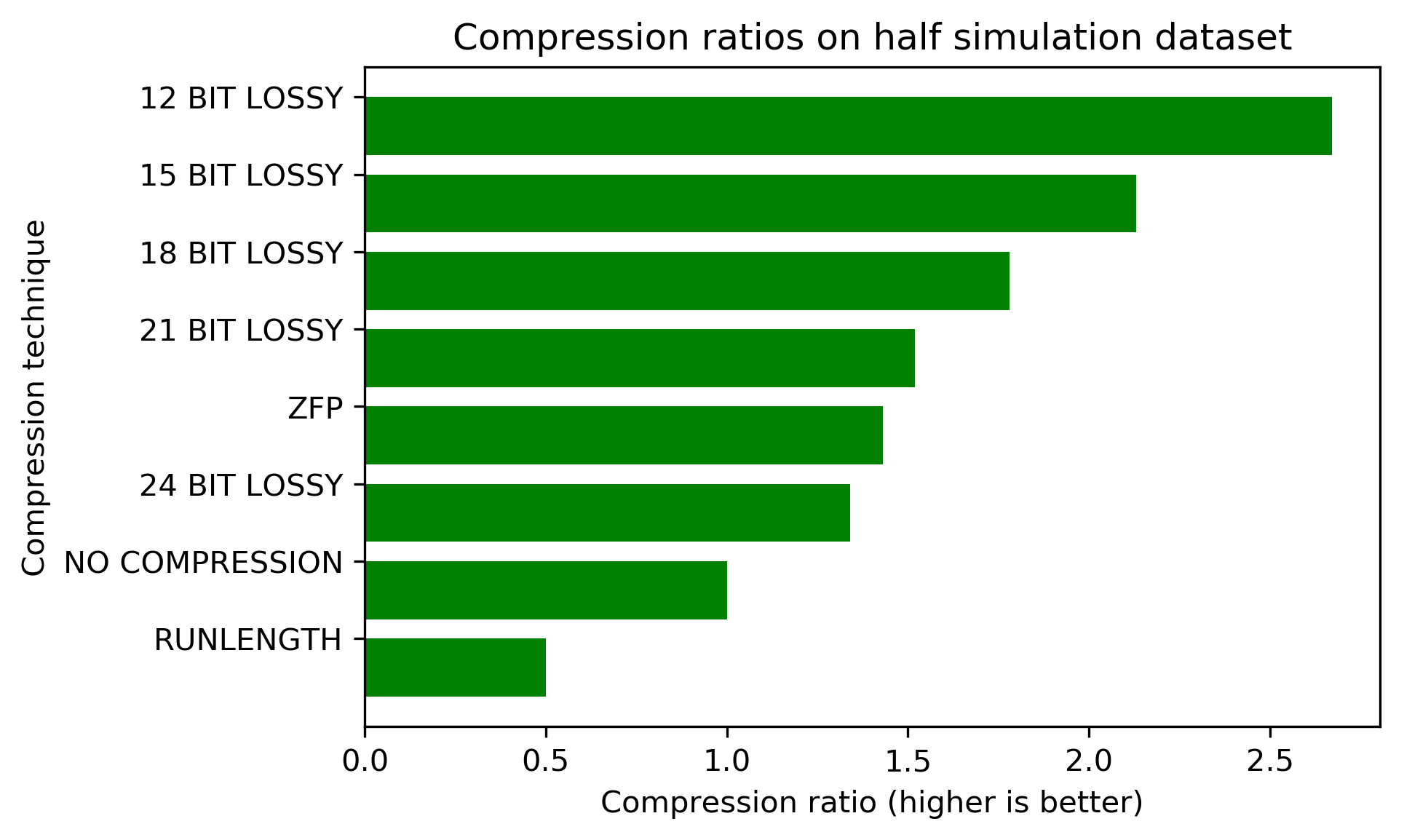


Figure 12 – Graphical representation of compression ratios achieved for the dataset for halfway through a simulation

### Compression overhead

All algorithms were analyzed in terms of average compression and decompression times of the entire dataset. The results of this analysis can be seen below in **Table 3**.

|  |  |  |
| --- | --- | --- |
| Compression algorithm | Average time to compress over 10 runs (seconds) | Average time to decompress over 10 runs (seconds) |
| Run length | 0.013171 |  |
| ZFP | 0.116863 | N/A (not implemented) |
| 24 Bit lossy (byte aligned) | 0.080418 | 0.158150 |
| 21 Bit lossy (non-byte aligned) | 0.117007 | 0.178427 |
| 18 Bit lossy (non-byte aligned) | 0.112390 | 0.174625 |
| 15 Bit lossy (non-byte aligned) | 0.111003 | 0.192604 |
| 12 Bit lossy (non-byte aligned) | 0.098646 | 0.200505 |

Table 3 – Results of various compression schemes for average time to compress and decompress for all 10 datasets.

It can be seen that run length compresses very quickly but has a poor compression ratio. Almost non-byte and byte aligned compression schemes outperform ZFP in terms of average speed to compress, more runs should be done in order to smooth out the 21 bit lossy compression that for some reason is a slight outlier. In terms of decompression, it’s a much costlier operation in terms of time to decompress than compress, and the less data that’s being decompressed takes longer as a whole. The results for the amount of time the compression takes to perform the aforementioned transformation can be seen below in **Table 4**.

|  |  |
| --- | --- |
| Compression Algorithm | Time to perform transformation (seconds) |
| Uncompressed | 0.514608 |
| 24 Bit lossy (byte aligned) | 65.408138 |
| 21 Bit lossy (non-byte aligned) | 96.339664 |
| 18 Bit lossy (non-byte aligned) | 90.633120 |
| 15 Bit lossy (non-byte aligned) | 79.909330 |
| 12 Bit lossy (non-byte aligned) | 72.295360 |

Table 4 – Results of various compression schemes for the time to perform transformation on data involving many individual compress and decompress calls.

It can be seen that there’s a prominent decrease in performance that comes with the compression and decompression of non-byte aligned values. As expected, all times for the transformation using compression have a significant increase in time as compared to uncompressed data. Comparing the times for the task with compressed data to uncompressed isn’t a valid comparison, due to the huge number of additional operations involved, instead, the transformation using compression algorithms should be compared to each other. There’s a significant speedup visible per digit lost of precision (~3.5 bits), and the use of non-byte aligned data would eventually outperform non-byte compression once approximately 5 digits of precision is lost.

# Conclusion

## Overview

The work performed in this project has successfully investigated the performance of different compression algorithms on a selection of different datasets, representing real world values. Evaluation metrics were chosen that can provide useful insight to the performance of these algorithms in the best replica of a real-world environment. If there was more time, the code would have been evaluated on an accelerator, but an acceptable compromise was made, a simulation similar to one on an accelerator was performed but on a CPU. The results are promising, the data compresses to a good ratio and compression/decompression are performed fairly quickly. In order to further evaluate the compression techniques, work needs to be performed on a hardware accelerated platform. The following section highlights some of the immediate future work that would need to be performed, if time allowed.

## Future Work

The work performed so far has fixed point and other compression schemes for this specific dataset from the Large Eddy Simulator. The next step to this project would be to port the code written to OpenCL. The code written for compression and decompression don’t use much dynamic allocation therefore it can be ported to OpenCL because OpenCL has no built-in dynamic memory allocation facilities, everything needs statically allocated. From here the code can be run on a hardware accelerator to see the performance increase and the new difference in efficiency in each compression/decompression method. From here, an existing issue which is also a motivation of this project may be experienced, a dataset too large for the hardware accelerator is attempted to be transferred over and because it’s too large, the transfer will perform poorly due to data transfer bandwidth being a bottleneck (or not perform at all on fixed memory FPGAs. A way past this would be to perform block based compression of the 3D pressure array. The 3D array (150x150x90) could be split into a pre-defined slices which can then be transferred to the accelerator. In addition to this a slice management module would need to be written to deal with the handling of moving and requesting slices from CPU to accelerator. This would allow larger simulations to be performed on the accelerator.

Another step could be taken is work on the advancement of the variable bit compressor to potentially achieve a better compression ratio. A revamped version of the variable rate compressor could be created which could work on a row by row or plane by plane basis. This compressor can calculate the average value of the subset of data that it’s working on. It could then store this average value and then every other value is stored as a sign bit and the absolute delta from this average value. For the datasets used in this project, it appears that this approach would work very well but there wasn’t enough time to implement and test this idea. This approach could work well for this dataset but not every dataset, for each if a dataset contained data evenly distributed between -7 and 7 then the overage average would be 0 and each magnitude value needs the same number of bits to represent it but there’s extra overhead for storing these average values.

With the port of 24 bit and variable bit compression to OpenCL, the next step would be to port ZFP to OpenCL for further testing, in order to test its performance and implement individual value compress/decompression. There’s two routes for this task. The first would be to analyze the ZFP code on GitHub and re-write it completely for OpenCL, removing any dynamic memory allocation as this is a feature not available on hardware accelerators, or facilitated by OpenCL. The other potentially faster option would be to change CUDA ported OpenCL code. There is a partial port of ZFP to CUDA which already deals with removing dynamic allocation of memory. The issue with this task is that the port is poorly commented and only partially done, so time would need to be spent learning CUDA, porting it and testing.

# References

[1] Toth, G., Sokolov, I., Gombosi, T. et al. (2005) Space Weather Modeling Framework: A new tool for the space science community – Journal of Geophysical Research, Space Physics.

[2] Vanderbauwhede, W., and Takemi, T. (2016) An investigation into the feasibility and benefits of GPU/multicore acceleration of the Weather Research and Forecasting model. Concurrency Computat.: Pract. Exper., 28: 2052–2072.

[3] Michalakes, John & Vachharajani, Manish. (2008). GPU acceleration of numerical weather prediction. Parallel Processing Letters. 18.

[4] J. Mielikainen, B. Huang, H. A. Huang and M. D. Goldberg, "Improved GPU/CUDA Based Parallel Weather and Research Forecast (WRF) Single Moment 5-Class (WSM5) Cloud Microphysics," in IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, vol. 5, no. 4, pp. 1256-1265, Aug. 2012

[5] Van Aalst, M. K. (2006), The impacts of climate change on the risk of natural disasters. Disasters, 30: 5-18.

[6] Paul, B.K. Nat Hazards (2009) 50: 289.

[7] Kant, Kamal & Telkar, Shivkumar & Gogoi, Meghna & Kumar, Deepak. (2017). Crop Simulation Models.

[8] Skamarock, W. C., J. B. Klemp, J. Dudhia, D. O. Gill, D. M. Barker, M. G Duda, X.-Y. Huang, W. Wang, and J. G. Powers, 2008: A Description of the Advanced Research WRF Version 3. NCAR Tech. Note NCAR/TN-475+STR, 113 pp.

[9] Met Office, Unified Model, 2018.

[10] Powers, J.G., J.B. Klemp, W.C. Skamarock, C.A. Davis, J. Dudhia, D.O. Gill, J.L. Coen, D.J. Gochis, R. Ahmadov, S.E. Peckham, G.A. Grell, J. Michalakes, S. Trahan, S.G. Benjamin, C.R. Alexander, G.J. Dimego, W. Wang, C.S. Schwartz, G.S. Romine, Z. Liu, C. Snyder, F. Chen, M.J. Barlage, W. Yu, and M.G. Duda, 2017: [The Weather Research and Forecasting Model: Overview, System Efforts, and Future Directions.](https://journals.ametsoc.org/doi/abs/10.1175/BAMS-D-15-00308.1) Bull. Amer. Meteor. Soc., 98, 1717–1737.

[11] Deardorff, J. (1970). A numerical study of three-dimensional turbulent channel flow at large Reynolds numbers. Journal of Fluid Mechanics,41, 453-480.

[12] Nakayama, H., Takemi, T. and Nagai, H. (2012), Large‐eddy simulation of urban boundary‐layer flows by generating turbulent inflows from mesoscale meteorological simulations. Atmosph. Sci. Lett., 13: 180-186.

[13] Pastor, D. S., MinUnit, Github. 2018.

[14] T. Shimokawabe et al., "An 80-Fold Speedup, 15.0 TFlops Full GPU Acceleration of Non-Hydrostatic Weather Model ASUCA Production Code," SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, New Orleans, LA, 2010, pp. 1-11.  
[15] Unknown, FileFormat.info. Run-Length Encoding (RLE)

[16] P. Ratanaworabhan, Jian Ke and M. Burtscher, "Fast lossless compression of scientific floating-point data," Data Compression Conference (DCC'06), Snowbird, UT, 2006, pp. 133-142.

[18] P. Lindstrom and M. Isenburg, "Fast and Efficient Compression of Floating-Point Data," in IEEE Transactions on Visualization and Computer Graphics, vol. 12, no. 5, pp. 1245-1250, Sept.-Oct. 2006.

[19] Düben PD, Russell FP, Niu X, Luk W, Palmer TN. On the use of programmable hardware and reduced numerical precision in earth‐system modeling. Journal of Advances in Modeling Earth Systems. 2015;7(3):1393-1408.   
[20] Lindstrom, P. ZFP, Github. 2018.

[20] Munshi, A., The OpenCL Specification, Khronos OpenCL Working Group, 2012.

###### Run length pseudocode

**set** current to value 0 in the array

**set** count to 1

**for** each value **i** in the array 1..n

**if** value[i] == current

count++

**else**

output+= current + count + ”,”

current = value[i]

count = 1