An Investigation into improving multi-GPU applications with Data compression

Sam Serrels - 40082367

Submitted in partial fulfilment of
the requirements of Edinburgh Napier University
for the Degree of
BSc (Hons) Games Development

School of Computing

April 25, 2016

Authorship Declaration

I, (Insert Name eg. Norman Stanley Fletcher), confirm that this dissertation and the work presented in it are my own achievement.

Where I have consulted the published work of others this is always clearly attributed;

Where I have quoted from the work of others the source is always given. With the exception of such quotations this dissertation is entirely my own work;

I have acknowledged all main sources of help;

If my research follows on from previous work or is part of a larger collaborative research project I have made clear exactly what was done by others and what I have contributed myself;

I have read and understand the penalties associated with Academic Misconduct.

I also confirm that I have obtained informed consent from all people I have involved in the work in this dissertation following the School's ethical guidelines.

Signed:

Date:
April 25, 2016

Matriculation no:
40082367

Data Protection Declaration

Under the 1998 Data Protection Act, The University cannot disclose your grade to an unauthorised person. However, other students benefit from studying dissertations that have their grades attached.

Please sign your name below one of the options below to state your preference.

The University may make this dissertation, with indicative grade, available to others.

The University may make this dissertation available to others, but the grade may not be disclosed.

The University may not make this dissertation available to others.

Abstract

This project aims to research the viability of compressing data on a Graphics Processing Unit(GPU) before sending it to another GPU to reduce the transfer time and bandwidth utilisation. The resulting data will be used to analyse the suitability of implementing compression methods into existing GPU workloads, such as real-time rendering, or distributed general purpose computation.

Contents

List of Tables

List of Figures

Acknowledgements

This dissertation would not have been possible without the help of the following great people:

Dr Kevin Chalmers, for supervising this project and always providing the best of advice.

Felix Jarvis, for letting me take apart his computers, and then fixing them.

Hugh Tennent, for inspiration and his great work in 1885.

William McEwan, for his exploits in 1856.

The Games Lab Family, for being the best and least-productive group of friends and colleagues I have ever known.

1 Introduction

Data Communication is one of the core obstacles to overcome in computation applications, even with the world's fastest computers; data moving between them is limited to the speed of light. This does not look like a problem until the speed of the processor is taken into account; at an example speed of 3Ghz, in the time taken for one clock cycle, light would have traveled only approximately 10 centimeters. Coincidently, this about the maximum distance for any two major components in a modern computer system. This is not accounting for bandwidth limitations, communication overhead, and the fact that speed of electric signals varies between 50% to 99% of the speed of light depending on the communication medium.
Communication time between internal components has been a problem that has up to now been solved with faster processing chips, protocol efficiency improvements and more physical data lanes. The speed mismatch of the destination components has always been the true bottleneck (CPU 3GHz, vs the speed of main memory 200MHz), leaving the communication link to only need upgrading when the relative speed of all other components has increased enough to reach the limits of the link.
With component speed reaching a plateau due to the physical limits of silicon being reached, the computer hardware industry is adapting parallelism to keep performance advancing forward. This means that the issue of communication is now possibly the biggest issue, as with more logical processing units comes a monumental increase in the need for passing and sharing data throughout systems.
This dissertation examines the GPU, to see how it communicates with the CPU, and specifically with other GPUs. The aim of this report was to test the theory that the spare computational cycles available to the GPU could be used to encode and decode data before and after communication in such way which would increase overall throughput. Firstly, the different technologies and methods of transferring data between GPUs is analysed and documented, followed by an implementation of various compression techniques and the resulting performance data presented and evaluated.
The ambition of this dissertation is to provide a documented analysis of the currently available tools and their capabilities, an overview on the operation of the GPU hardware, working and measured reference implementations for data compression on the GPU, and advice for integrating such a system into an existing application or product.

1.1 Motivation

With the introduction of new rendering APIs to control GPU hardware i.e Vulkan and DX12 (detailed in the Background section), it is now possible to utilise the power of multiple GPUS in new ways to achieve greater rendering performance. This comes at a convenient time as the emerging Virtual Reality and UHD display technologies require significantly more rendering throughput than the current best in class GPUs can deliver individually.
The ability to have fine grained control over multiple GPUs comes with the previously unexplored problem area: how and when to transfer data between the units. This is a problem that is shared by GPGPU applications, albeit with less of a priority placed on nanosecond to nanosecond operations, and more placed on overall job time and net throughput.

1.2 Aims and Objectives

- Evaluate the capabilities and performance of existing GPU APIs with regard to transferring data between devices.
- Implement compression and decompression algorithms on the GPU and document their performance

1.3 Research Questions

- How are inter-device data transfers handled by the different APIs and hardware?
- What are the major constraints and bottlenecks for inter-device transfers?
- it feasible to implement data compression into an application with the current existing hardware and APIs?
- With regards to both hardware and software, what would need to change to allow for compression to be/become more effective?

1.4 Scope

1.4.1 Deliverables

- An implemented application to test the various methods of compression and to output performance data.
- Documented findings and test results of different implementation methods
- A report into the technical possibilities and limitations of compression and data transfer using GPUs.
- A report into on the viability in using compression in actual applications.

1.4.2 Boundaries

This project will only consider sending and receiving data between GPU devices, it will not take into account the creation of the data or when/how it became available to be transferred. Although the report does contain speculation on how a compression scheme could be integrated into a larger application, this was not implemented or tested.

1.4.3 Constraints

During the majority of the project timespan, the newest GPU APIs were unavailable, therefore this project makes use of the pre-existing CUDA and OpenCL technologies. DirectX 12 was publicly released midway through the project and was utilised for some small initial tests, however the project was deeply tied to CUDA by this time and so a framework switch was not feasible. The Vulkan API was released to the public after the implementation was completed, so it was also not possible to use this for anything other than theoretical reference.
Data compression is vast and well established field of research, this project is focussing on

the implementation and testing of a select sample of existing compression solutions, this project does not attempt to create a new compression algorithm. This project requires access to compatible hardware, i.e two modern identical GPUS, as vendor support for the OpenCL standard varies, a complete scientific approach would be to acquire a large matrix of GPUs for results, this was not possible for this project.

1.5 Report Structure

TODO

2 Background

2.1 Graphics Processing Units

Initial GPU products started as co-processor expansion boards for computers in the late 1980s, their use was primarily for acting as display-output hardware with a small amount of memory for storing the display data. Soon after, 2D transform operations were incorporated as the GPU industry started to gain speed. Due to a large numbers of companies releasing competing products, the following years were a race to release products with more features. Market fragmentation with operating system and hardware exclusivity deals eventually caused to most of the companies producing GPUs to be merged and bought over.
By 1995, the focus of the computer graphics industry had moved to 3D acceleration, producing chips that could process 3D mathematical operations needed for rendering 3D games significantly faster than the CPU could. Software standardisation was required to reduce the number of proprietary graphics protocols that developers would have to implement. The OpenGL standard was created as an open industry standard for sending commands to GPUs. Microsoft's own Direct3D API (Part of the DirectX suite of gaming APIs) was created and maintained for the Windows operating system, the extra commercial support of DirectX allowed it to initially surpass OpenGL in terms of features and games industry adoption.
The following ten years upto 2005 saw improvements in all aspects of computing hardware; GPUs were entering into a near yearly development cycle, with each new line of products vastly outperforming the last. Nvidia first introduced their Scalable Link Interface(SLI) technology in 2004 which allowed two or more identical GPUs to be used concurrently for graphics rendering. This was handled solely by the Nvidia GPU driver and the running game or application has no control over which of the cards graphics commands were sent to. ATI (now AMD) soon released their similar CrossFire technology which also allowed combining of multiple GPUs in a single system. With the ability to combine GPU chips, ATI and Nvidia began releasing single cards that contained two GPU chips as high-end enthusiast products.
The architecture of the GPU chips at this point had become highly complex due to the amount of supported graphical features, controlling the GPUs as state machines from the application had become cumbersome. To remedy this, the industry moved to a system where the GPU would run entire small C language programs supplied by the application. DirectX developed the "High-Level Shading Language" (HLSL), similar to Nvidia's "C for Graphics" (Cg) Language, which was also similar to the "OpenGL Shading Language" (GLSL).
By this time (around 2005/2007), GPU chips had become significantly more complex than the desktop CPUs they would be paired with (AMD Athlon 64 X2 6000+ CPU : 243 million transistors vs Nvidia GeForce 8800 GTX GPU: 681 million transistors). As the method of programming the GPUs had moved to a C like programming language, GPU manufactures took the route of simplifying the chip architecture by reducing redundant silicon. Processing that used to take place on separate parts of the chip would now take place on the same area, on general purpose processing units (named Cuda Cores by Nvidia, and Stream Processors by AMD). This simplification, named "Unified Shader Architecture", allowed for a more general purpose chip design and thus made room for faster and more efficient GPUs.
The move to a more unified and general purpose chip design additionally brought in

the ability to run general purpose code on the GPU. Nvidia released the CUDA SDK, which allowed non-graphical application to access and use the computation resources of the GPU. AMD (who had purchased ATI in 2006) implemented the OpenCL standard on their hardware, Nvidia then later also added support for OpenCL, however CUDA remains a technology that will only run on Nvidia products.
Due to the data-heavy nature of 3D mathematics, GPUs had naturally evolved to be highly parallel to process the vast quantities of rendering data needed for gaming. This parallel design was completely different to the design of CPUs, which would have at most 8 cores. While CPU cores are capable of SIMD instructions, and hyperthreading could effectively double the core count, this was nowhere near the quantity of the thousands of cores inside the GPU.
With the introduction of General Purpose Computing on the GPU (GPGPU), this highly parallel design allowed the GPU to tackle specific data-parallel computational problems at a vastly increased rate to the CPU. This promoted GPUs from just graphics processors to valuable tools for a wider range of industries. Nvidia and AMD have since produced products that use the same chips as the GPU products, but with no output or video hardware, these "High Performance Computing (HPC) accelerator cards" could be used in datacenters or workstation machines.

2.2 Existing GPU APIs

To make use of GPUs, an abstraction layer is required to shield developers from the differences in hardware between different GPU products and manufacturers. Currently there is a small number of industry standardised APIs available and in use for programming and controlling GPUs, the main differences between them is primarily which hardware manufactures and operating systems support them.

2.2.1 Rendering APIs

As the initial and primary use of GPUs is for rendering 3D graphics, the oldest and most used APIs are also built for rendering. With technologies like compute shaders becoming more mainstream, the capabilities of these APIs are starting to include an ever increasing amount of GPGPU functionality, but their purpose remains first and foremost for rendering.

OpenGL The Open Graphics Library (OpenGL) is a specification authored by the Kronos Group, an industry lead consortium dedicated to creating open standards for accelerated devices. OpenGL does not provide any software or libraries, it is purely a specification that must be implemented and supported by the GPU manufacturers. OpenGL has implementations on virtually all devices and operating systems, including mobile devices, therefore it remains the obvious choice for multiplatform graphics development.
TODO: Compute shaders

DirectX DirectX is a collection of libraries intended for the videogames industry. Created by Microsoft, the implementation is proprietary and only runs on Windows and XBox products. The primary benefit to DirectX over OpenGL is that each version is not fully backwards compatible with the last, this does not initially seem like a benefit, but

it has allowed Microsoft to iterate faster, and keep the specification small which improves conformance testing with devices.
Having Microsoft's financial and development support behind the API has enabled it to stay ahead of OpenGL in terms of features, tool support, and hardware compatibility. As DirectX only runs on the Windows operating system, Microsoft has can couple the low level components of the API and the operating system together closely in a way that would be very difficult for a multi platform API. TODO: direct Compute

2.2.2 GPGPU APIs

For using a GPU for general purpose programming, the majority of the features included in OpenGL or DirectX are not needed, furthermore tasks like retrieving data back from a GPU is a convoluted endeavour as the APIs are designed for sending data not and not receiving.
There are now two main APIs for GPGPU programming, OpenCL and Cuda, they both can operate a GPU independently and can also interoperate with rendering APIs.

OpenCL OpenCL is very similar to OpenGL, but lacking the rendering commands. It is also entirely a reference specification with no supplied implementation. OpenCL has implementations on most of the common GPU hardware.

CUDA Cuda is a proprietary API library supplied by Nvidia, while Nvida has supplied drivers for most common Operating systems, it is limited to Nvidia hardware products only. The feature set is virtually identical to OpenCL with some minor differences.

2.2.3 API Style

The APIs are fairly low level and verbose, and supplied as C libraries. However a major difference to normal C code and writing GPU code is that the programmer does not own and cannot directly access any GPU resource. Memory allocations have to be requested from the driver, which will return an identifier to use the memory location in the future rather than a direct pointer to the memory. The hardware driver decides the layout and position of items on the hardware, and in most cases, when to transfer data to and from the GPU. This behaviour is unseen by the programmer and can lead to difficulties when debugging code.
The simplified standard workflow for GPU programming is:

1. Setting initial state & querying device capabilities
2. Creating space for resources (buffers)
3. Sending resources to the GPU (compute data, textures, 3d meshes)
4. Compiling and sending programs (Shaders/Kernels) the the GPU
5. Telling the GPU to execute the programs
6. Waiting for completion
7. Sending results to the screen / Return results to the CPU

8. Updating resources, repeat from step 5

Shaders / Kernels The user programs that execute on the GPU, that modify and process data are named Shader Programs for graphics, and Kernels for general purpose computing. They are written in an API specific language, generally a C language derivative with extra operations and syntax. These programs are executed as many instances in parallel, generally there will be one instance for every piece of data to compute, or every vertex in a model, or every pixel on a screen. There are limitations to the complexity of these programs, as they are executed completely in parallel, the hardware can send the same instruction to each processing unit all at once. This causes major performance issues with branch operations as both paths of a branch will have to be taken and one path discarded afterwards. Similarly, recursive operations are not possible and loops in general are discouraged. A GPU may have thousands of cores, but each core has a fraction of the power of a single CPU core, so they should not be treated as one. Simple instructions, making simple programs, running all at once is the key to GPU performance.

High Level GPGPU APIs C++AMP is a standard originally created by microsoft to create parallel applications in a high level environment, without the need to communicate or program for a specific GPU. Programs are written in C++ with extra directives to state how the work should be divided between processing units. The C++AMP implementation will then perform all the necessary initialisation and execution on the GPU.
While C++ AMP is just a standard, it has been designed to use DirectCompute as a backend, therefore C++AMP cannot do anything that would not be possible (albeit with substantially more code) in DirectX with Compute Shaders. However in theory any implementation of the standard could target any backend, such as OpenCL or Cuda.
Single-source Heterogeneous Programming for OpenCL (SYCL), is an abstraction layer standard from the Kronos Group with similar aims to C++ AMP. At the current time of writing it is not in common use and the implementations are in the early stages of development.
The aim of these high level abstraction layers is to allow parallel programs to be created without knowledge of the hardware that it will run on. At the time of this dissertation, these technologies are still new and lacking full implementations, so they were not used to implement the project.

2.3 New GPU APIs

While GPU hardware has evolved and rapid pace, graphics APIs have evolved substantially slower. New features have been added over time, but the overall design of the APIs have not changed, this is not completely true for DirectX, which can and has modernised internal systems over the various releases. DirectX 11 was especially different from previous versions, as it aimed to increase support for parallelism by allowing the application to give more detailed information to the driver. However the OpenGL Specification, which was designed for maximum compatibility, has grown in size and relies on hardware specific extension to support non standard features. This has caused portability issues as a system does not only have to support the OpenGL standard, it may also have to support specific non standard extensions to run the latest applications.
Old and legacy code is one of the two major issues with DirectX and OpenGL, the other major issue is driver overhead. The GPU hardware driver has to do a substantial error

checking, resource management, and scheduling of the hardware, all without any knowledge of what the application is doing or what it will do next. To achieve this, there has been limitations placed on what and when an application can request from an API. The application can give hints to the API to increase performance, such as marking a resource as read-only, but these are only hints. The true major limitation is that API calls can only be called from a single processing thread. An application may be running on many CPU cores, but when it comes time to communicate with the GPU, this can only be done by one of these cores, in one thread.
Application designers have attempted to mitigate the effects of this by processing data in a way which does all it can to minimize GPU communication, but with modern games making upwards of 5000 calls to the GPU, the limits of application-side optimisations on modern hardware are being reached.
A refresh and redesign of the graphics APIs was needed to take full advantage of the advances made by modern CPUs, GPUs, and Operating Systems. Work on a solution to this was first publicly shown by AMD with their "Mantle" project, which allowed for low level control of specific AMD products to increase the performance of games. A small number of large "Triple-A" games adopted Mantle, this demonstrated the viability of such an API, but with it limited to just a subset of AMD products, something else was needed to bring the benefits to all platforms.
There now exists two new graphics APIs, each a significant departure from the their predecessor's design. These new API were publicly released during the course of this project, with the full benefits and use cases are still yet to be determined. What can be determined at this point is that with the new fine-grained low level control over GPUS, code can run faster and more efficiently, providing performance benefits for the desktop user, and power saving benefits for the mobile user. Crucially this also allows for control over GPU data communication that has not been possible before, which was one of the primary motivations for this project. With the new capabilities there will be questions about how best to use them,questions which this report intends to attempt to answer.

2.3.1 DirectX 12

This version of Microsoft's API was released alongside Windows 10 on the 29th of July 2015. The primary aim of DirectX 12 was to reduce driver and API overhead, it achieved this by creating methods and constructs for the application to use to inform in detail what it intends to do to the driver.
These constructs can be saved reused to cut down on the amount of state changes the hardware is required to do. The application can now also generate GPU commands from multiple threads, allowing for better distribution of work across the CPU cores. A Critical new feature introduced with DirectX 12 is Multi-adapter functionality, allowing multiple GPUs to be used separately for graphics processing, alongside methods for communicating between GPUS. This had not been possible beforehand and is allowing for completely new render pipeline designs. Most modern CPUs include a graphics unit which usually lies dormant if a dedicated GPU is installed in the system, with DirectX 12 the integrated GPU could be used concurrently with a dedicated GPU on the same graphics application. This is discussed in detail in the implementation section

2.3.2 Vulkan

Vulkan is the successor to OpenGL, authored by the Khronos group, it shares many of the same advances as DirectX 12. The major advantage that Vulkan has over OpenGL is the removal of legacy items and a general cleanup in the specification, in addition of new low-level features. Vulkan still remains a crossplatform API, including mobile platforms, which is a difficult task for a low-level device-agnostic API. The release came with a major push towards increasing tool quality and conformance testing to break away from the main issues that plagued OpenGL; compatibility issues and implementations not following the specification completely.

2.3.3 Mantle

Starting point for vulkan, now deprecated.

2.3.4 Metal

Apple's version of vulkan for mobile devices

2.3.5 The place of new the rendering APIs

At the time of this report both, Vulkan and DirectX 12 are still brand new, industry adoption is looking positive but only time will tell. With the new capabilities of these low-level APIs comes substantially more work for the developer if they wish to make full use of everything the specification provides. Part of the motivation and inspiration for these APIs has come from the games console industry, wherein the developer has virtually full control over the hardware. As every console uses the same hardware and operating firmware, developers can use highly specific operations to achieve the greatest performance. With Vulkan and Directx 12 aiming to bring these benefits to a wide range of hardware and software platforms, compatibility testing is going to become critical and hardware vendors are going to have to make sure that their provided drivers adhere completely to the API specification on all devices. In summary, these new APIs unlock a large degree of potential performance, but require a large amount of careful work to use it. Tools, game engines, and library support will make things easier in time, but the full use case an adoption compared with the older more-usable APIs is yet to be determined.

2.4 Multi-GPU Use for Graphics and Gaming

Computer systems with more than one GPU was once only an area for gaming enthusiasts, it is now becoming more commonplace with many professional tools making use of GPGPU acceleration and the higher demands of the emerging Virtual Reality technologies. Until recently it has not been possible to control more than one GPU in a single rendering/graphics application, the distribution of work between GPUs is the responsibility of the hardware driver. This has the benefit of abstraction for the developer who can gain extra performance without changing any code, with the downside that the performance gain is limited as the driver can only use generic algorithms, and does not have knowledge of the program code. It is also common for modern CPUs to have a low-power

on-chip GPU, paired with a separate higher power and faster GPU, this is used in laptops to reduce power usage by disabling the high power GPU when not needed. It has also not been possible to use the CPU GPU and a separate GPU together in one application, until now.

SLI: As the driver could not alter the order or timing of the incoming graphics commands from the application, the best option to increase performance was to duplicate all instructions and data across all cards and use

2.5 GPU and Multi-GPU Use for General Purpose Computing

2.6 GPU Data Communication

2.7 Data Compression

2.7.1 Parallel Data Compression

3 Implementation

3.1 Plan

The implementation comprised of two stages, the first stage was to investigate the existing methods of GPU communication. These methods will be measured and analysed for their effectiveness in different use cases. After this stage is completed, work would then commence on the implementation of compression algorithms.

3.1.1 Initial Investigation of Communication methods

The true goal of this stage was to find an API that is most suitable for implementation compression, to find this, each different GPU technology was tested. A standard set of tests were implemented in each technology and the results were compared, the comparison looked at two categories:

Measuring Total Timespan Synchronization between the GPU and CPU is required at multiple points during the standard workflow of executing a kernel. Furthermore GPU state has to be initially setup by the application to be able to run any task, altogether this results in a significant time expenditure. Measuring only the data transfer times would not give an accurate analysis of the overall application performance. Ease of use is another factor taken into account when testing out different APIs. although this is not easily quantifiable, a fast api could be penalised if it was overly complex to use. The goal of this report is to recommend an implementation so other factors other than just time have to be considered and reported.

Measuring Transfer Timespan The raw throughput of transferring data from one GPU to another was measured for different file sizes and file types. Different types of data were used as there may be low-level compression built into the transport protocols and therefore different types of data must be tested to ensure an accurate analysis of throughput. These results are used for speculating on the theoretical throughput if there was no need for the CPU to intervene.

3.1.2 Software Framework Requirements

A framework was created to host and run the tests undertaken in this project, this was created to reduce development time of new tests and to manage common code across the different tests. The framework was designed with four major goals in mind:

Selection of tests and parameters Running and repeating tests should be easy and as intuitive as possible, available test should be selected from a menu, along with any parameters it requires with the option to use sensible defaults. After a test is completed, it should run any necessary cleanup and return to the menu

Standardised tests and results A standardised set of timing and time conversion functions should be implemented to be used across tests. This timing data should be collected and averaged before being printed to a fie.

Cross platform and API agnostic The framework should support atleast the standard Linux and Windows compilers and operating systems. The build system should check for the installed versions of OpenCL and CUDA, provide version information to the application and and correctly link the most recent version of the library provided.

Automated For quickly repeating tests and gathering results for multiple parameters, the framework should have the ability to process pre-specified parameters which can be specified via the command line.

3.1.3 Result Visualization and Comparison

With a framework for gathering results, a tool was needed to be able to compare tests against each other to spot both large differences between transmission methods and small differences as parameters are tweaked when optimising an individual method. This too would need to be able to read and understand the different sets of results that the framework would generate. Idieally this would also be connected to the automation of the framework so that results would be collected and visualised automatically.

3.1.4 Hardware

This project was able to make use of three separate hardware setups for deploying and testing, along with a few other auxiliary systems for testing and secondary tasks.

	Tesla Server	Lab Pc	Home Pc
OS	CentOS	Windows 7	Windows 10
GPU	2x Tesla K40	2x GTX980	AMD R9290, 2x AMD 5770
CPU	Intel i7-4770K @ 3.5GHz	Intel i7-4790K @ 4.0GHz	AMD FX8320 @ 4.0GHz

Table 1: Implementation Hardware

	Testing PC 1	Testing PC 2	Build Serve
OS	Ubuntu Desktop	Ubuntu Desktop	Ubuntu server
GPU	2x GTX970	AMD R9 390	N/A (openCL on CPU)
CPU	AMD FX8350 @ 4.2GHz	AMD Phenom X4 995 @ 3.0 GHz	Virtualized CPU, 2 cores @ 4GHz

Table 2: Other Hardware

3.2 Test Framework

3.2.1 Introduction

The application took the form of a command line application, navigated by a simple menu system to select options and input parameters, having live visualisations and graphical interface components was considered but was ultimately out of scope for this project. The program could be completely operated via command line arguments to allow for easy automation and repeating tests.

3.2.2 Portable Build and Deployment Considerations

CMake To allow for simultaneous deployments on both Linux and Windows platforms, the CMake build tool was utilised. CMake was used to generate either the Visual Studio Solution file or the GCC makefile from the source code. CMake also allowed for additional pre-build logic to be run to analyse the system capabilities.

Jenkins Build Server An external build server running the Jenkins Constant Integration software was created. When code was pushed to the version control host (GitHub), this would trigger a build with Jenkins. Jenkins would pull the code to the server and compile it, it would send a notification in the event of a build failure and would also log any errors. Static analysis tools (CppCheck) were also used on the code and logged in conjunction with the compiler output to maintain a high code standard. The build server had the ability to run OpenCL on its CPU via the Intel driver, this allowed for additional tests to spot memory issues with the Valgrind tool, this took a long time however so as the project transitioned to Cuda this activity was phased out.

The build server had access to the Tesla server and would run a remote agent to ensure the code compiled without issue on both systems. As the linux ecosystem has many different compiler and library versions, running the build on two different distribution flagged up a surprising amount of bugs.

Jenkins could also run any tests within the application after a successful build. This would have provided a large historical set of data as the application was developed, however this was found that it was not actually needed.

3.2.3 System Capability Testing

The pre-build process was designed to inspect the capabilities of a system and determine what APIs were available and their current versions, this information was passed onto the compiler to make sure that the program could be linked with the APIs correctly. This information was also made available to the application if it required it.

OpenCL Determining if a system has OpenCl drivers installed is a complex and convoluted task, after much research it was found that there is no accepted good technique. The task of ensuring OpenCL is installed, up to date, and compatible with an application is the responsibility of the user rather than the developer, as ultimately the developer may not have knowledge of the systems a given application will run on. This follows the

ethos of OpenCL, in that it should run everywhere without modification, in reality this does not work well due to driver differences, and the fact that a system can have multiple OpenCL drivers installed simultaneously. In this situation, different "platforms" are exposed to the application to chose first before it can chose any devices. Devices cannot be combined with devices from different platforms. This can work well, but an application must be linked with only a single OpenCL library, therefore on a system with multiple openCL libraries, one must be found and chosen.

An OpenCL library should look for and make available any other OpenCL platform on a system once the application is running, so it should not matter which library is chosen, however it was found that some libraries did not do this (Nvida+AMD on Linux, Nvidia on Windows 7). This raises the question that if the Nvidia Library is used to compile the application, then the application is moved to a different computer with no Nvidia hardware, would it still work? This is behavior that OpenCL specification has solutions for, and it seems that the newer the driver and library used, the less common these problems are, so this may be a case of bugs that already solved are or will soon be solved.

The method chosen by this implementation was to create a custom CMake module to search in known paths for versions of OpenCL. When a version was found, the header file was inspected to find the version of the library. After the search was completed the library that supported the newest version of OpenCL was chosen for linking with the application.

Cuda Cuda is only supported on Nvidia hardware, and while different versions of the Cuda SDK can be installed, finding the newest version to link with is a trivial task. This task was included in the same CMake module that searches for OpenCL.

Hardware Capabilities After choosing an OpenCL or Cuda SDK to link with to compile the program, there may still be incompatibilities if the actual hardware does not support all the features supported by the driver. Both APIs have methods to handle this, thankfully this is much easier when compared to the Graphics APIs, as the hardware will supply the highest version number of an API that it supports and this can be used to identify what can and can't be done on the hardware. With compatible features identified, the amount of cores, warp size, and memory capacity still need to be queried to make sure a given test can run on the GPU. This project used the newest and the best GPUS available to make sure this was not a problem.

3.2.4 Result Capture and Visualisation

Timing Features The framework made use of the C++ Standard Chrono Library for timing. Cuda provides a more precise timing method which was used where applicable. Timers could be started and stopped and then automatically added to a results set. The framework would run tests multiple times (default of 100) to get an accurate reading for each time. The times would be averaged and saved along with the raw timing for each run to a CSV file after a test had finished.

A single run of a test may have many separate steps that need to be timed separately, the framework supported this by providing a test the ability to create and name sets of timers.

The saved final file would have the times for each run for each set of test steps.

Test Step Visualisation Visualising an entire algorithm comprised of many steps required creating a new tool to show an overview of a test compared with another. The objective was to have the ability to compare individual steps of an algorithm rather than just the summed run time. This was implemented as a web app in JavaScript using the D3.js visualization framework. Result CSV could be uploaded to the site and would be added to the list of results to view, this was scripted to happen automatically on the Tesla server. A limitation of the tool is that it can not display two result sets if they have a different number of steps, different solutions to visualize this were tried but ultimately it always resulted in an unclear display.

3.3 Inter-Gpu Communication

"Discrete GPUs use dedicated high-bandwidth memory that exists in a separate address space. Moving data between the device address space and the host requires time-consuming transfers over a relatively slow PCI-Express bus. " ([1])

3.3.1 OpenCL

OpenCL was chosen as the first technology to be analysed, this was due to OpenCL's wide range of support platforms and therefore any advances made would have the greatest area for application. The address space of the GPU is not directly or even indirectly accessible from the application on the CPU. The application can create "buffers" to store data, GPU kernels can access the data in these buffers when executing, so they must be copied to the GPU. The application has no control over this (with one exception, explained later), it is completely up to the driver to identify when data would be needed on a gpu and initiate the copy.

> An OpenCL context can have multiple devices, and a memory object that is located on a device has a location on each device. To avoid over-allocating device memory for memory objects that are never used on that device, space is not allocated until first used on a device-by-device basis. For this reason, the first use of a memory object after it is created can be slower than subsequent uses ([1])

Optimisations can be done to ensure the copy is as fast as possible, by using "Pinned Memory" which is memory that is guaranteed to be in a static contiguous location within Main Memory. This allows the GPU to use a DMA transfer to quickly receive or transmit data to the CPU. However this is only a speed optimisation, the application still has no control over when the transfer will happen.

To transfer data from one GPU to another, they must write to and read from the same buffer. OpenCL does not provide low level cross-device/cross-queue synchronization barriers, so the application must use the GlFinish command to ensure a gpu has finished writing. This command has a significant overhead as it blocks the cpu until all current tasks on the GPU are finished, this means the cpu will be waiting for synchronization and cleanup operations to happen in addition to the kernel operations.

Once the application can be sure a GPU has finished writing to a buffer, it can then execute the kernels on the second GPU to read the data. Without even considering transferring data from card A to card B, the operation procedure to synchronize reading and writing has already taken a significant amount of time.

clEnqueueMigrateMemObjects As previously mentioned, there is no way to explicitly move a buffer from one GPU to another, or even move a buffer anywhere at all, as the placement of buffers is completely up to the driver. There exists an API function call named "clEnqueueMigrateMemObjects", which allows the application to express that it wishes a buffer to be associated with a specific device. Theoretically this should provided enough of a hint to the driver that it will initiate a transfer, however this is still only a hint and the OpenCL spec does not guarantee that a transfer will occur.

It was found that this function either provided no perceived speedup, or would actually crash the application in some specific combinations of hardware and software. There exists an AMD OpenCL extension to allow for direct GPU-GPU communication, but this is only available on the workstation FirePro GPU systems which could not be sourced for this project.

Although it was not proven scientifically, it is speculated that when the driver does schedule a transfer of data between GPUs, this transfer is routed through the CPU and possibly Main Memory rather than through the SLI or Crossfire Bridge. This could depend on the driver and hardware used, but it was observed in both AMD and Nvidia setups where the card bridge could be used, the transfer times strongly suggest they are not utilised.

3.3.2 Cuda

As Cuda is a technology that only operates on Nvidia hardware, it can afford to provide functionality that operates on a lower lever when compared to what OpenCL can offer. There is no concept of abstract buffers in Cuda, the application is in charge of allocated memory on the GPU and transferring data to and from it. The application cannot access the GPU memory directly, this must be done with Cuda functions, data is written and through "MemCopy" operations, the CPU cannot use direct pointers to GPU memory. Pinned memory is also provided in cuda, this can be written to from the CPU and quickly (compared to non pinned memory) accessed by the GPU.

Streams Cuda allows for queuing GPU commands into a "stream", tasks in a stream will not execute until the previous task has completed, this synchronization occurs within the driver or on the GPU, not within the application. Data transfers can be queued after kernel completion to copy the data as soon as it is ready, the application will wait until the stream has finished rather than each task within it. Multiple streams can exist, however they must be associated with a single GPU for maximum performance. Synchronization barriers can be placed inside streams to synchronize across multiple streams, this means that GPU 2 could wait on GPU 1 to finish a job and then copy the from the card as soon as possible. This is a huge improvement over the abilities in OpenCL and this is main reason why Cuda was chosen as the API for the remainder of the project.

Peer to Peer Nvidia provide a method of copying data between GPUs directly over PCIe without routing through Main Memory, this is known as Peer to Peer (P2P) access. This is only supported on certain hardware and drivers versions. This is substantially faster than normal data transfers. However CPU synchronization is still needed at the start and end of the transfer.

Unified Virtual Addressing On high-end Tesla compute cards, Unified Virtual Addressing (UVA) technology is available. This merges the address space of the any GPUs and the CPU together. This is primarily an ease-of use feature as it can simplify and reduce code, Kernels can reference data on other devices and memory operations can move data around without doing as much address mathematics. This is mainly used for sharing data between the GPU and CPU, but when coupled with two GPUS and P2P operations it can give a small but noticeable increase in throughput. Using P2P memory copies with UVA was determined the fastest method of moving data from one card to another. Multiple copy operations can be executed at once vai streams, but it is not clear if these actually execute in parallel, are interleaved, or execute in sequence.

3.3.3 DirectX 12

The ability to use DirectX 12 became available midway through the project. DX12 has the ability to dispatch commands and data to specific GPUS. The major issue with Using DX12 at this time was that the drivers and documentation was still in a very early state. A working implementation that moved data from one GPU to another was implemented, however the results were not nearly as good as using Cuda. As DX12 is a full graphics API, integrating the data transfer into a games/graphics pipeline would be much simpler than with Cuda or OpenCL. As driver support and performance improves over time this method may become viable to integrate into a product, at the moment it only serves as a proof of concept that multi-GPU transfers can be done.

3.4 Gpu Compression

Any compression algorithm that runs on the GPU must be a highly parallel in nature, this is a hard task as compression in a data intensive operation that requires accessing data from the entire file at somewhat random times. If the data is split and compressed in chunks, the total compression ratio will always be less than an algorithm that can run across an entire file. Furthermore, joining the compressed chunks together is an operation that requires communication between each of the compressing elements, essentially a parallel reduction operation. This can be done completely with GPU kernels, but the data quantity involved makes this a slow process. It could be possible to skip the join stage and pass the data as junks to another GPU, which would get the data off the compressing GPU quickly. The data would still have to be combined by the receiving GPU which would also need extra metadata to do so, which increases the overall data transfer quantity.

3.4.1 Bit Reducer

To discover generic compression issues/bottlenecks and to get a baseline performance reading, a basic compression algorithm was created. The algorithm interprets data as 32bit integers and swaps swaps any leading zeroes in the binary representation with a 5 bit number representing the amount of bit removed. This was designed to make the best use of the parallelism of the GPU, as the data can be easily segmented into chunks of 32 bits, and each chunk can be processed independently. The difficult task is recombining the compressed chunks together as each chunk will be a separate size. This was managed by placing the size of the chunk into a shared memory location, creating a contiguous list of all chunk sizes. A single kernel will then read this list and compact the chunks together. Decompressing the data does not play to the strengths of a GPU as well as compressing, the input data has to be divided into individual chunks, this required reading sequentially through the data to pick out the size bits and data bits. Splitting the data randomly would not work as there is no way to tell a size bit from the data bits. This was remedied by adding additional data to the start of the entire data set, this contained the boundaries of a set of chunks within the output data. This data allowed the data to be divided correctly. This works well if the the compressing and decompressing gpus have the same amount of compute units and warp sizes and results in essentially a perfect reverse of the compression process with the cost of a few extra bytes of data.

Input data	2,4,6,8,0,1
Input Size	6x32 = 192 Bits
Chunks	4,5,6,8,0,1
Binary Encoding	100,101,110,1000,0,1
Size Bits	3,3,3,5,1,1
Size Bits Binary	11,11,11,101,1,1
Finished chunks	111001110111110,10110001011
Chunk Sizes	15,11
Chunk Sizes Binary	10101,10001
Final Binary data	10101,10001,111001110111110,10110001011
Final Size	36 Bits
Final Ratio	0.1875

Table 3: Bit Reducer Example Steps

3.4.2 GFC

"The GFC algorithm is a novel lossless compression algorithm for double-precision floating-point data targeted for parallel execution on a GPU." ([5])

GFC is a GPU compression algorithm with that achieves high compression ratios with high throughput compared to any other published work. The paper which describes the algorithm claims a total compression throughput of 75 Gb/s. this was chosen as a test case for GPU compression as primarily for the results and speeds that it provides, if this algorithm was found to not be fast enough (to gain speedup over raw transmission) then the chances of any algorithm working would be slim.

The GFC Algorithm works in a similar fashion to the much simpler Bit Reducer in that the data is split into blocks and processed independently, then combined with metadata

containing block information. GFC also includes Bit-Reduction code to remove unnecessary 0 padding. The primary method used by GFC when processing blocks is to use the assumption that consecutive data will be of similar magnitude. A number is assumed to be the same as the previous number, and then the error is recorded, if they are infact the exact same, the data needing stored will be minimal. In a worst case scenario where a number is the maximum int range from the predicted number, then the data to be stored will be the same size as the number with a few extra info bits.

> In the best case where all values are correctly predicted, the 32 64-bit values in a subchunk are compressed down to 16 bytes, resulting in an upper bound of 16 for the compression ratio. In the worst case where all residuals have eight non-zero bytes, the "compressed" output is 16 bytes larger per subchunk than the uncompressed input, resulting in a compression ratio of 0.94, i.e., an expansion by 6%. ([5])

GFC was designed for high compression throughput, and although the process could be reversed for decompression on the GPU, the use case for this algorithm was to get large amounts of data off of the GPU quickly and then it can be sent elsewhere i.e., over a network. It was found in this investigation that a high compression ratio and overall processing times are in the realm of the results described in the paper, however the compressed data is not in a format that can be easily transferred to and decoded by a decompressing GPU with some intermediary CPU work.

3.4.3 Lossy Image Compression

Video games use lossy compressed textures for rendering anywhere it is possible to do so, the speed benefits and memory savings make it an obvious choice. The degradation in individual texture quality is seen to be worth sacrificing for an increase in overall image diversity. In some cases (primarily old DirectX 8 era games), compression can increase image quality as game artists would have to downsample uncompressed textures to maintain a playable framerate. Compression can keep the low file size and also details in textures that would have to be cut to save space without compression. This has been the accepted practise for so long that Hardware and APIs have dedicated functions to enable image compression. This was looked at to see if the existing API functionality and hardware encoders could be used to compress and decompress procedural image data before transferring to another GPU.

S3TC S3 Texture Compression (S3TC) (commonly referred to as DXT after the integration into DirectX) is a set of lossy image compression algorithms designed for video games. The technique was first published by S3 Graphics Co for use in their Savage 3D GPU [4] .

> S3 Texture Compression is a lossy, fixed-rate, texture compression format. This style of compression makes S3TC an ideal texture compression format for textures used in hardware-accelerated 3D computer graphics. Following the integration of S3TC with Microsoft's DirectX* 6.0 and OpenGL 1.3, the compression format became much more widespread. ([2])

The compression works by dividing an image into blocks of 4x4 pixels, these segments are compressed independently of all other segments, allowing for easy parallelism. The

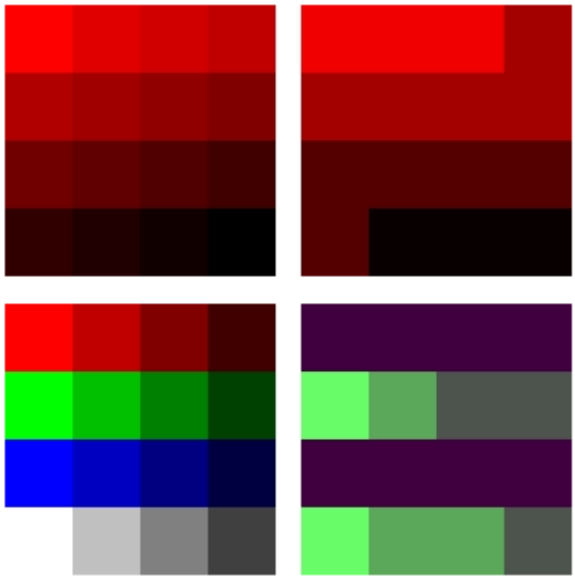

Figure 1: **S3TC, Single block** - Left: Original, Right: Compressed [3]

compression on the blocks involves finding two colours where every pixel in the block can fit on or close to a linear scale between the two colours. Then every pixel is rounded to the closest value on the linear scale. This works perfectly for blocks that are comprised of only shades of a single colour, if two or more colours with separate hues are present in a block, artifacts can occur. At a scale of 4x4 pixels, each block is relatively small and should not contain much image detail before compression, so artifacts should not be noticeable for large textures. There is a set of different versions of S3TC (DXT1, DXT3, DXT5), the difference being the amount of bits used and how alpha channels are handled, there is no fundamental difference to the algorithm for each version.

S3TC decoding has been built into the hardware design of GPUs for some time, it has also been integrated into the graphics APIS (still technically an extension in OpenGL) for around a decade. All game engines compress games texture during some art pipeline stage before the assets get into the game. Hardware support means that the compressed texture can be used in game code as if they were uncompressed without any additional code, the GPU manufactures can also tune the decoding algorithm to best fit the GPU hardware.

ATSC Adaptive Scalable Texture Compression (ATSC), is a newer compression format that combines many of the optimizations discovered by prior compression formats together to form a compression technique that provides much greater perceptive quality while remaining a small file size and fast GPU decompression. ATSC uses a block based approach, highly similar to S3TC, however the blocks do not need to be square. Each block is still a fixed size, so splitting the data for decompression is trivial. The main advances in ATSC is how the bits are allocated within each block. The allocation can be shifted depending on the contents of the block, for a block with similar colours, more data can be allocated to each pixel. For a block with multiple colours, more data can be allocated to the metadata and colour scale. With relatively small amount of bits to work with, changing the allocation would result in large changes i.e., an extra bit for each pixel would significantly reduce the remaining space for block metadata. ATSC makes use of

Bounded Integer Sequence Encoding to get around this problem and gain finer control over bit allocation. BISE leverages the spare bits available of the maximum number in a range and uses this to quantise the data down to a smaller size using higher base number systems (base 3 and 5). Essentially this can the effect of allowing fractional bits for each pixel.

"[BISE has] a significant saving, and has the somewhat weird property of assigning a non-integer number of bits - 2.33 - to each value[pixel]." ([6])

ATSC has hardware support in virtually all modern Desktop and laptop GPUS, and in most common mobile GPUs, it is expected to become the default compression standard due to the image quality it produces. It does take significantly more instructions to decode vs S3TC, but S3TC was designed for 1990's GPUS, with the extra processing power available to modern day GPUS, ATSC can be decoded at more than acceptable speeds.

Encoding and Decoding S3TC and ATSC Using these standard formats for this project seemed like an obvious choice, however it was found that compressed textures are designed to be used a certain way, and using it for the use of this project was unfortunately not feasible. GPUs can decode Compressed texture formats quickly, and transferring the compressed data from one card to another would not be any more complex than moving any other data. The issue that was discovered was the encoding stage. The accepted use scenario for video games is that the texture are compressed on the CPU before being sent to the GPU. This can be done at run time, using API calls (for some formats only), or by using an offline tool. The offline tools are designed to get the best image quality in the smallest file size. Time to encode is not a priority as these tools are only designed to be run once, then the compressed images are saved to disk along with other game files to be loaded at run time. While these algorithms in theory should be well suited to GPU architecture for encoding, this is a task that has not been commonly attempted. This project ran into difficulties getting an S3TC compressor working in CUDA, and unfortunately this feature set had to be cut from the final results.

4 Results

4.1 Data Transfer

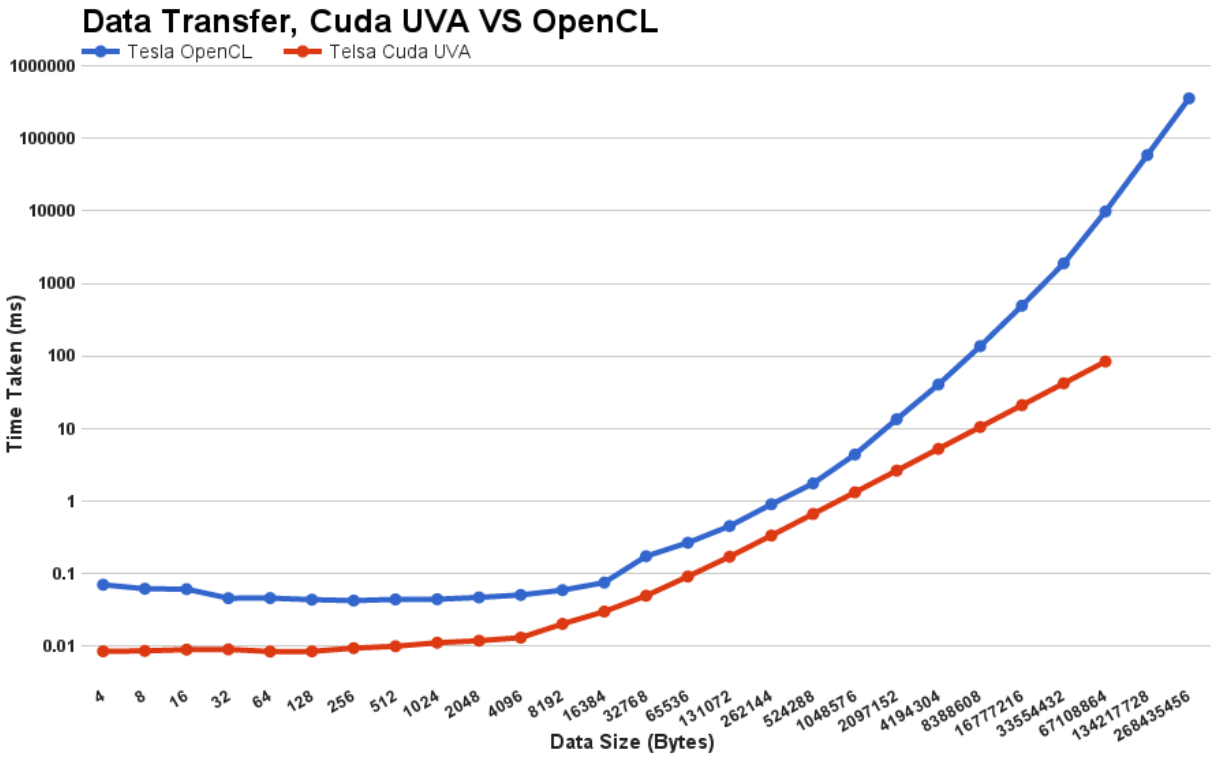

Figure 2: **Data Transfer: Cuda Vs OpenCL**

As previously mentioned, the difference between Cuda and OpenCL for transferring data is significant. This data (Figure 2) was recorded by sending data back and forth between two cards, and then taking the average time. Asserting that data is actually resident on a GPU is difficult with OpneCL as the driver is in charge of distributing data. Therefore a simple program has to be launched on each gpu which accesses the data to assure the data is in place. Ideally, the setup and running time of this program would not be included in the measured time, however there does not exit a way of accurately splitting this timing data from the transfer time so it was left in. To keep the test fair, a simple data program was also implemented into the cuda test and included in the timing data.

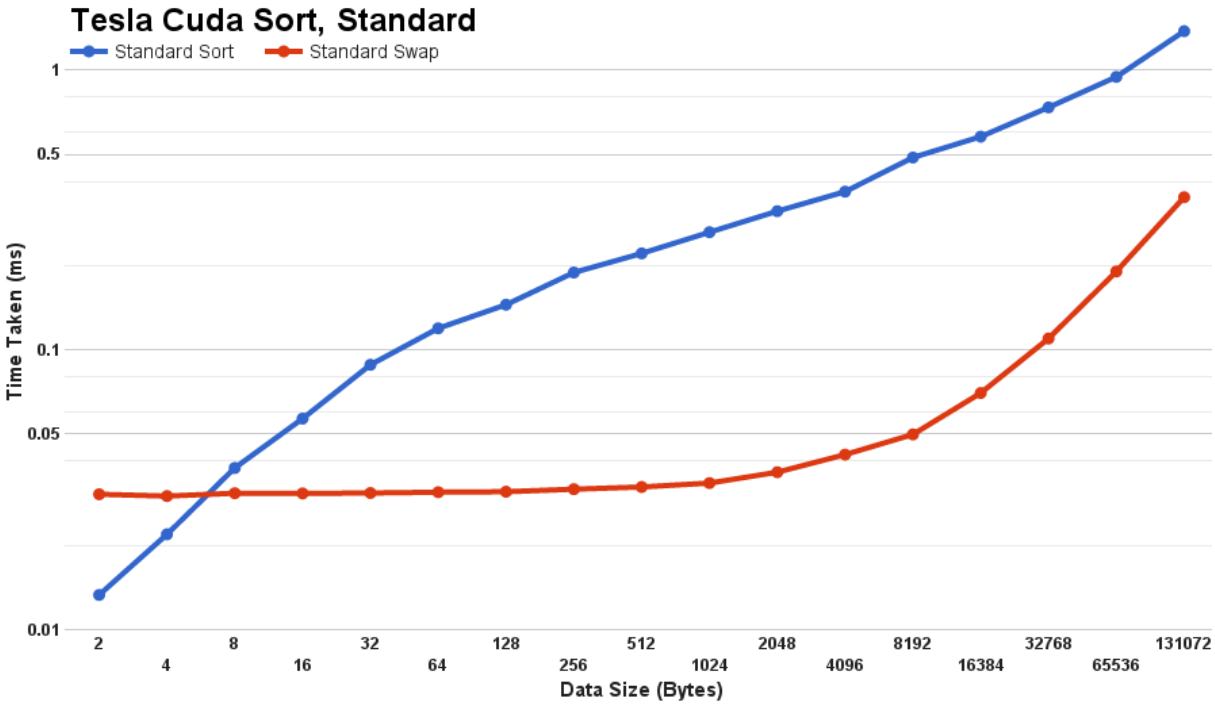

Figure 3: **Cuda Sort**

Figure 3 shows the results from running the parallel bionic sort. The sort data is the combined sum of all the sort stage running times, the same for swap. The swap time is mostly unaffected by the data size when compared to the sort time, which grows exponentially with the data size which is to be expected. This data shows that for a standard computational task which requires two gpus, the transfer time is probably sufficient.

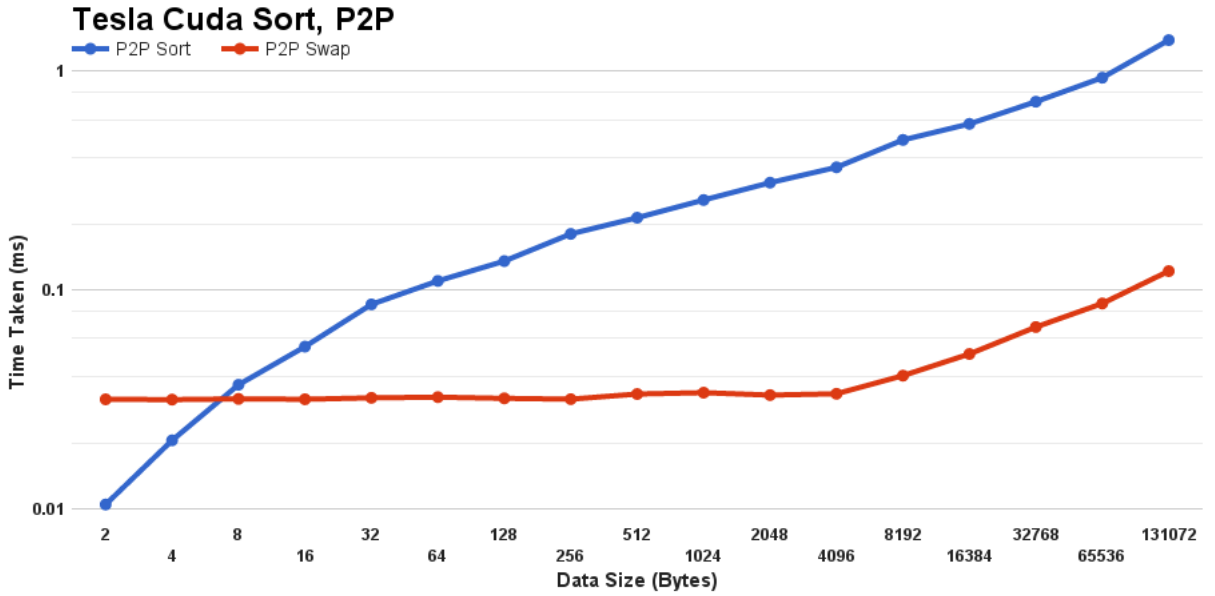

Figure 4: **Cuda Sort P2P**

Figure 4 shows the same sort test as figure 3, but with using the Peer to peer data transfer mode, which allows data to be transferred over the PCI bus directly between

cards. There is a noticeable and significant increase in transfer times, especially at the higher data sizes.

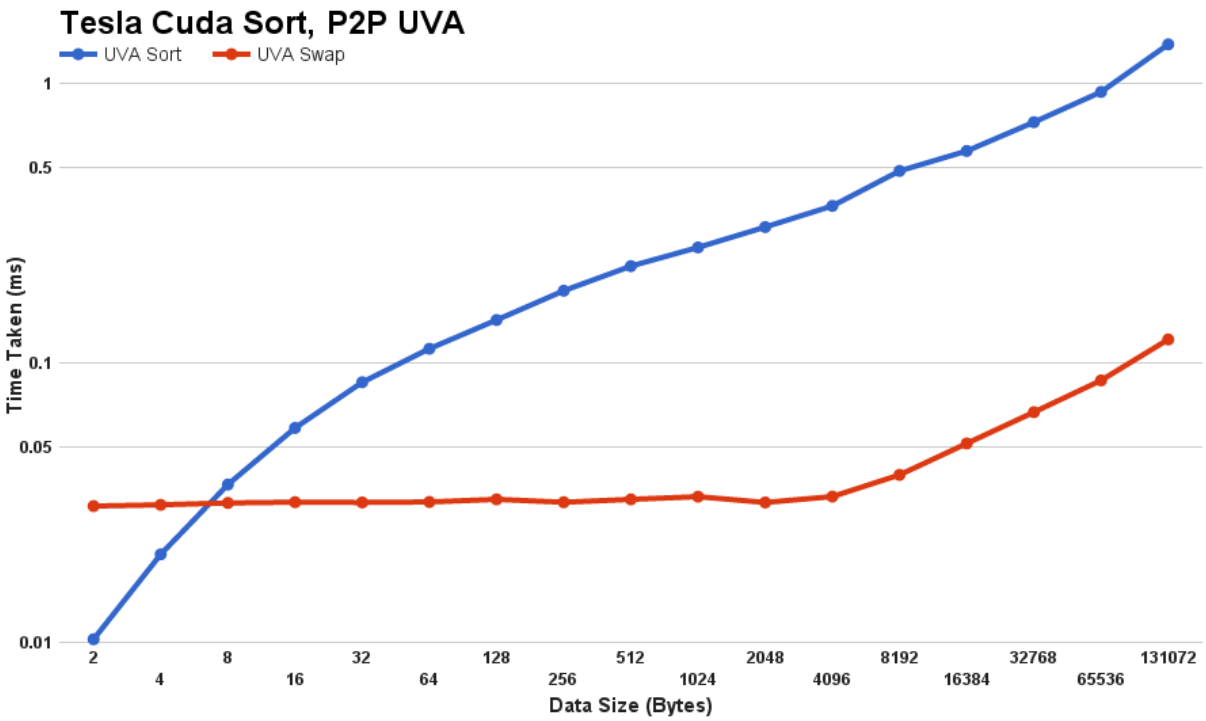

Figure 5: **Cuda Sort P2P+UVA**

Figure 5 shows similar data to 4, but with Unified Virtual addressing enabled. Thsi is mainly a software optimisation and doesn't result in much of a noticeable increase in transfer times, there is however a small boost to sort performance.

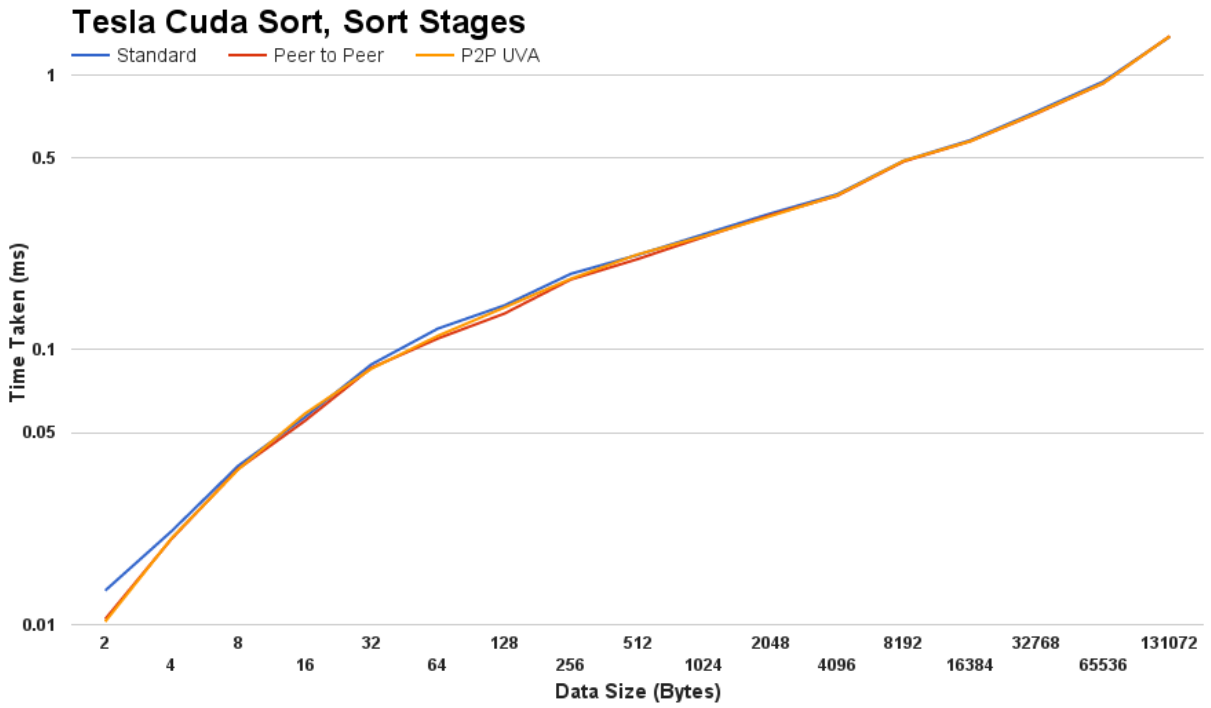

Figure 6: **Cuda Sort Comparison of Sort Stages**

Figure 6 combines the sort stages together for comparison. there is very little difference, other than both P2P and UVA always have a slight lead.

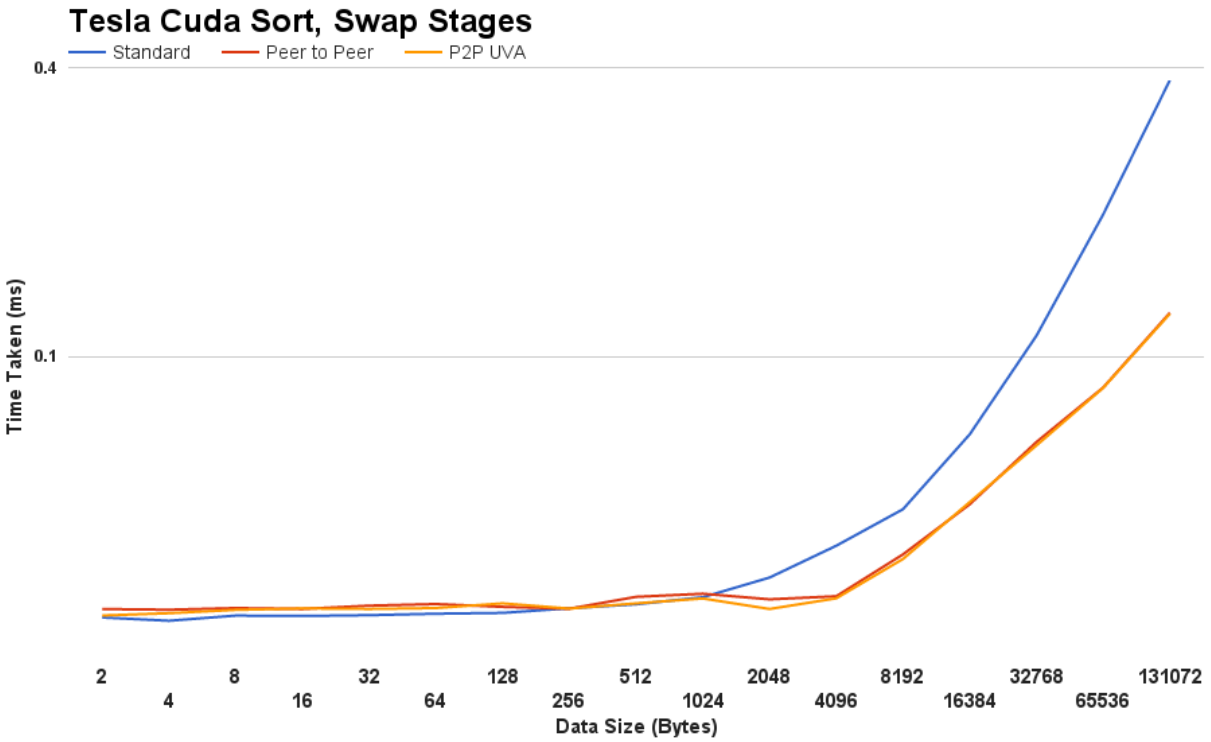

Figure 7: **Cuda Sort Comparison of Swap Stages**

Figure 7 combines the transfer stages together for comparison. Here the benefit of the P2P and UVA modes can clearly be seen, although only after the data size increase above 1024 bytes. UVA always has a slight advantage over P2P.

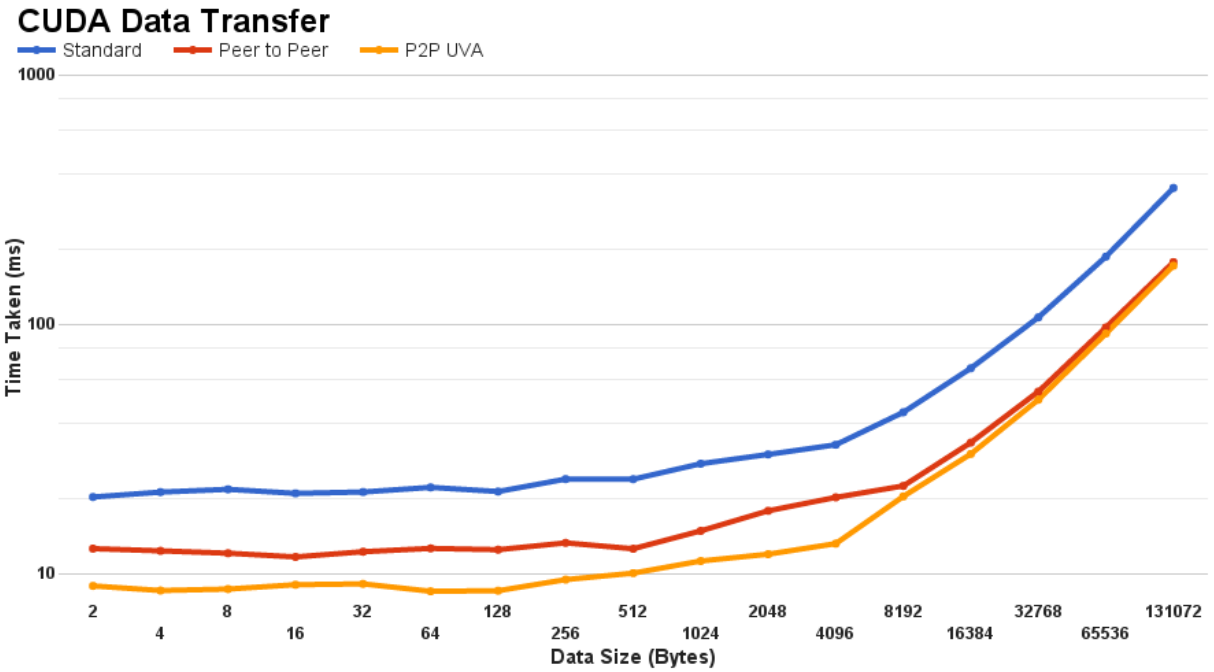

Figure 8: **Transfer Time for different Cuda Data Modes**

Figure 8 analyses the different transfer modes unter a pure data transfer test, similar to figure 2. This reinforces the finding form the sort tests and shows a clear speedup when using the more advanced transfer modes, furthermore UVA has a much more pronounced advantage. This is probably due to inefficiencies within the driver when dealing with data swaps.

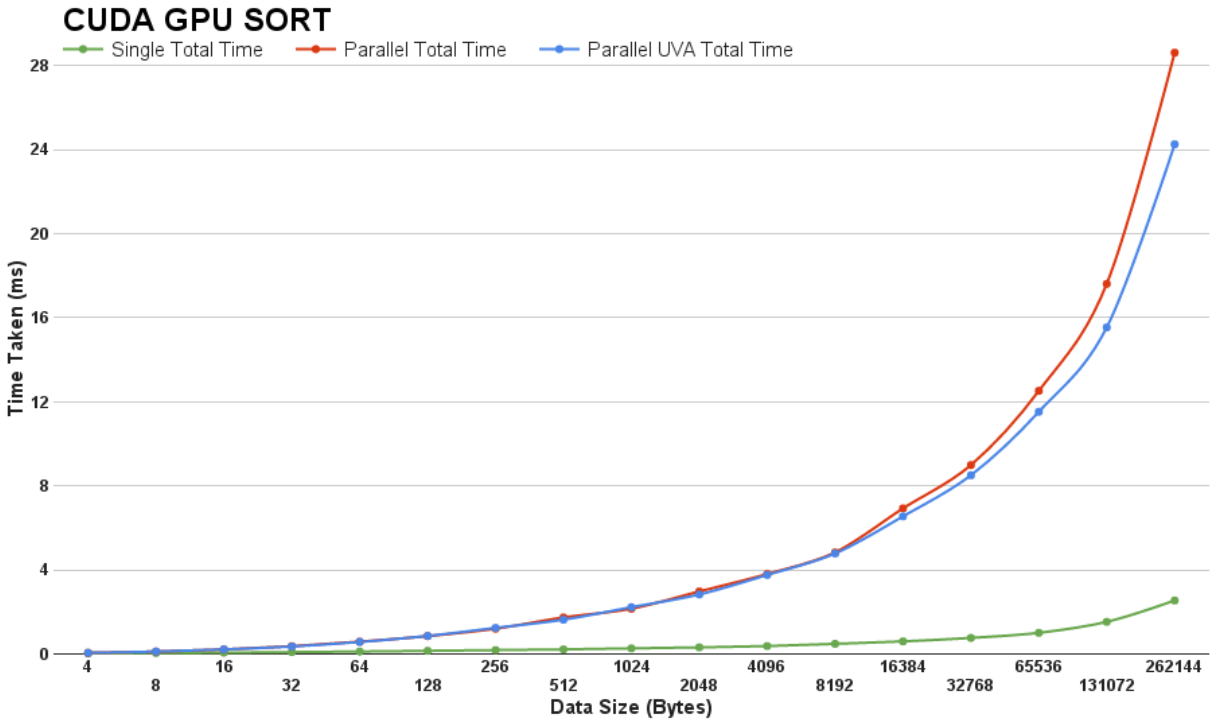

Figure 9: **Cuda Parallel Sort vs Sequential Sort**

Figure 9 compares running the parallel sort against a sequential version of the sort. Al-

though the sequential sort has to deal with double the amount of data to sort, it does not have to deal with multiple sort and swap stages.

4.2 Compression

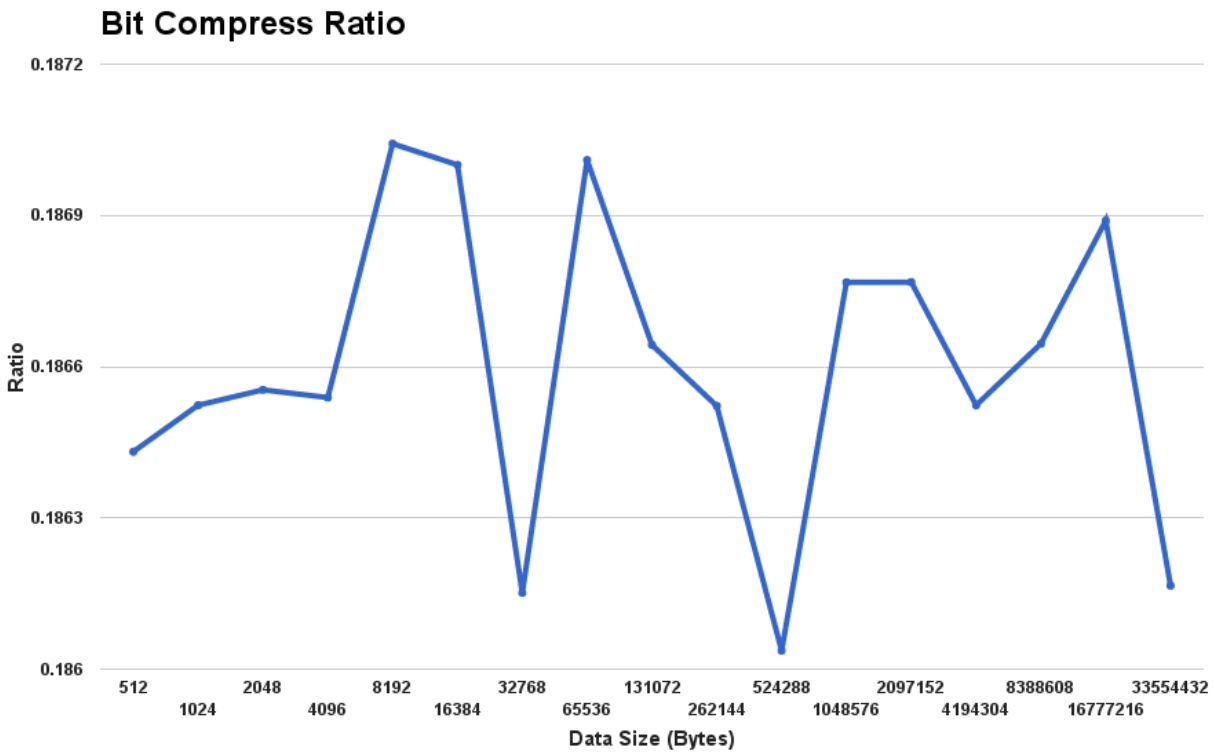

Figure 10: **Compression Ratio of the Bit Reducer Compressor**

Figure 10 displays the ratio achieved for different data sizes for the Bit reducer compressor. The ratio depends on the data contents, with lower numbers producing better compression. This input data used was random numbers, so the compression ratio does not convey much correlation to data size.

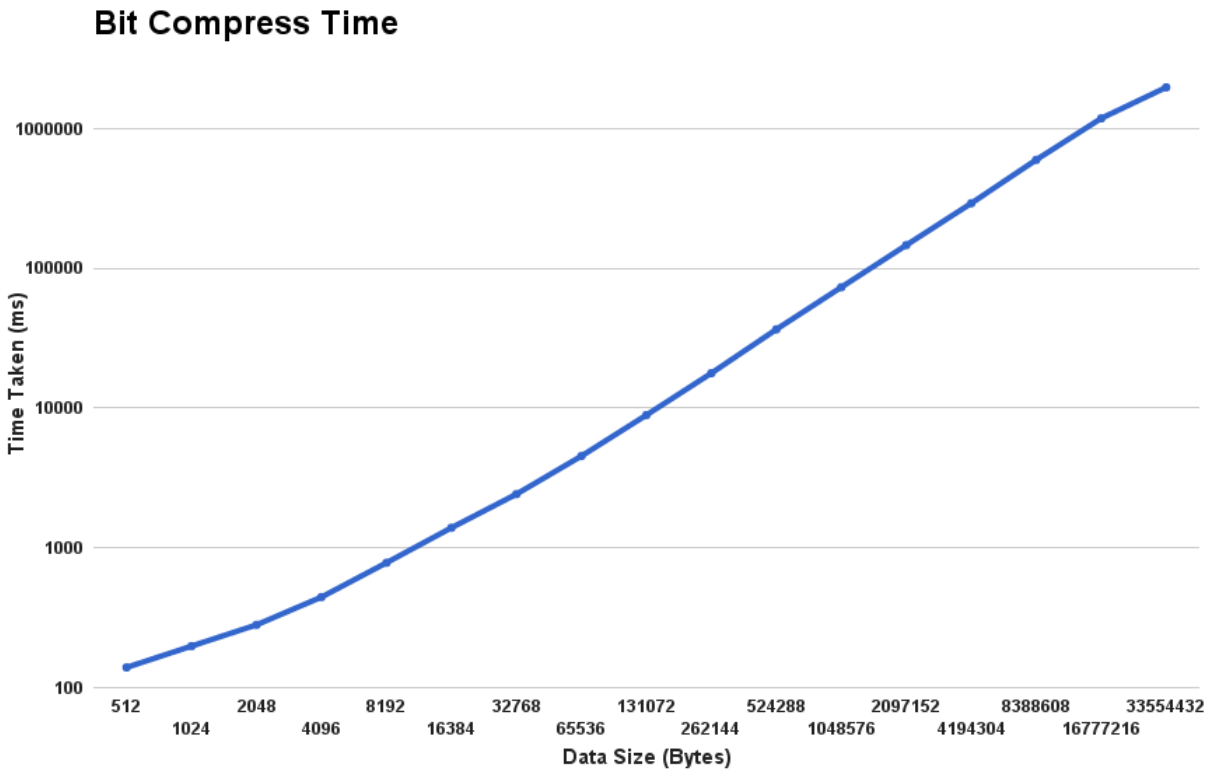

Figure 11: **Compression Time of the Bit Reducer Compressor**

Figure 11 shows the total time to compress the data, this shows the linear nature of the algorithm. Although the time does correlate well with the data size, with a minimum time to compress of 100 milliseconds, this algorithm is limited in it's use.

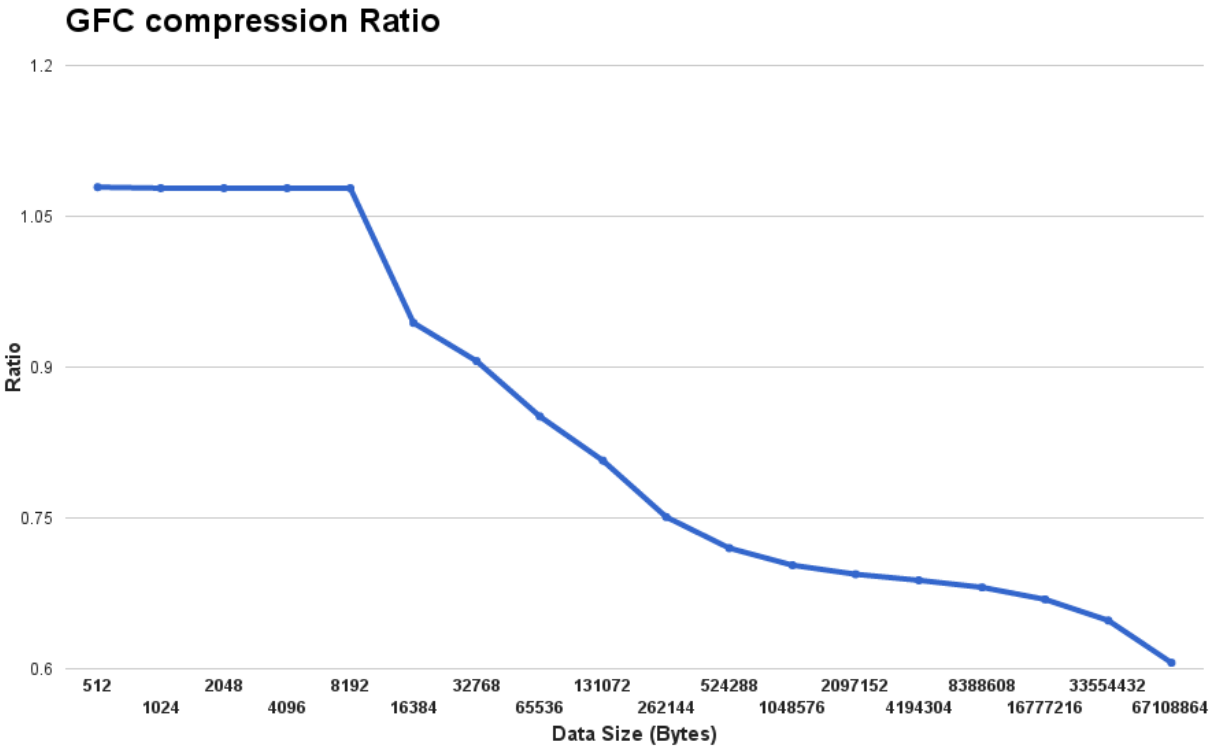

Figure 12: **Compression Ratio of the GFC Compressor**

Figure 12 displays the compression ratio archived via the GFC compressor. The algorithm

does not become effective until the data size increases above 8192, this is due to the design and overhead of the algorithm. As the data size increase, the compression ratio improves, unfortunately the size limitations of GPU memory prevented additional data sizes being tested to see when/if the ratio levels out.

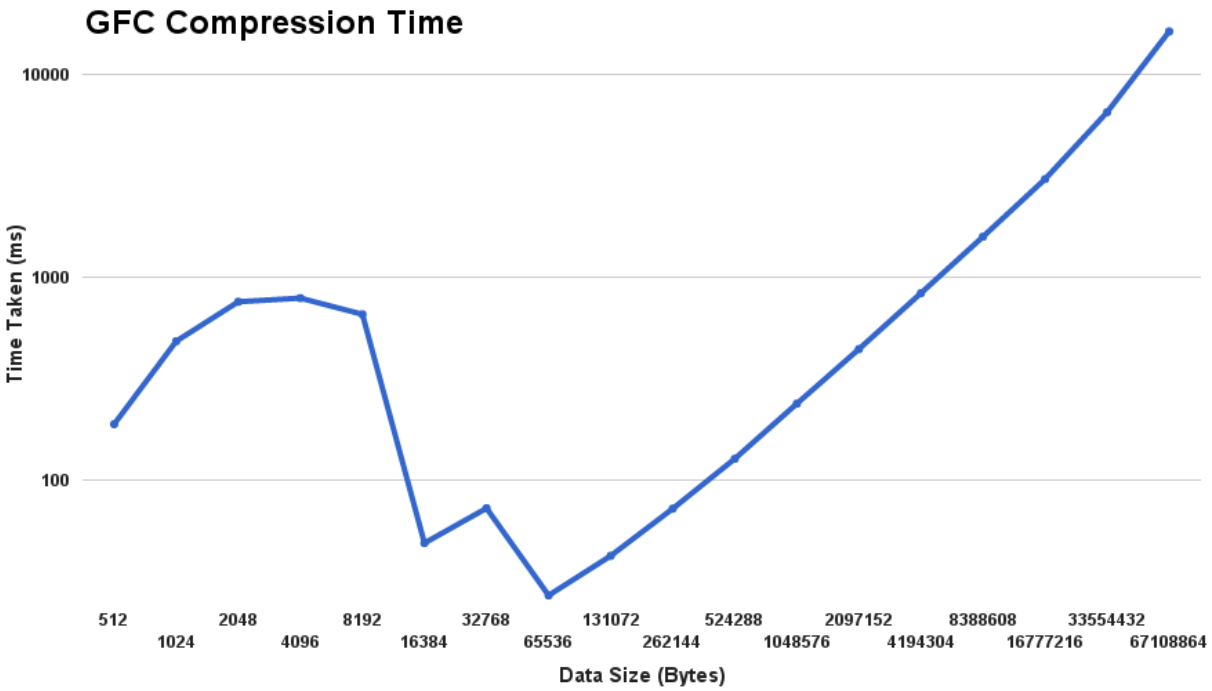

Figure 13: **Compression Time of the GFC Compressor**

Figure 13 Shows the time taken to compress data with GFC. As seen in the previous figure, anomalous data occurs before the data size increase above 8192. The compression time then increase somewhat linearly to the data size, the overall time is significantly less than the Bit Reducer compression.

Figure 14 is the most significant dataset produced, it shows taht speedup would be gained by compressing the data with GFC before transmission over transmitting the whole data uncompressed. This is a theoretical data based only on the compression and projected transmission times, as converting from the output format of GFC to data that could be sent and received would take additional time. This does however show that there is potentially enough compute power and API support to justify compression before transmission with high data volumes.

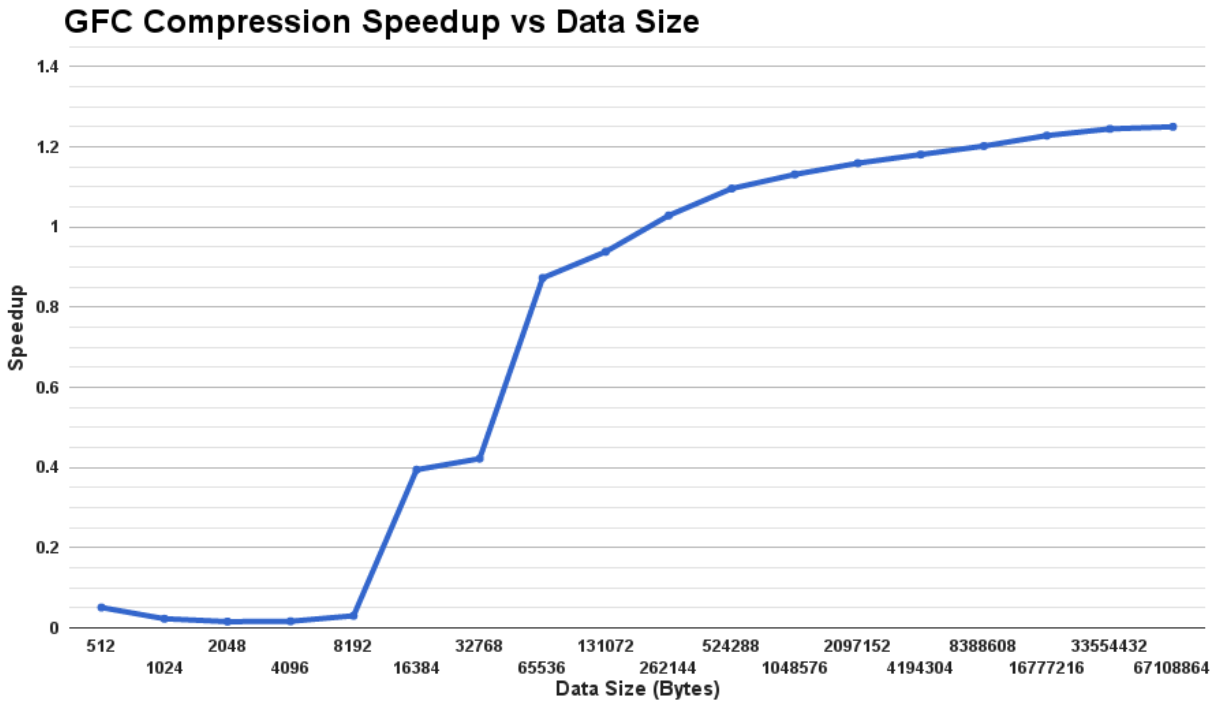

Figure 14: **Compression Speedup**

References

[1] AMD Inc. Opencl optimization guide, 2016.

[2] CRISTIANO F. Android* texture compression - a comparison study with code sample, 2016.

[3] FS developer Wiki. Dxt compression explained, 2016.

[4] Z. Hong, K.I. Iourcha, and K.S. Nayak. Fixed-rate block-based image compression with inferred pixel values, August 10 2004. US Patent 6,775,417.

[5] Molly A. O'Neil and Martin Burtscher. Floating-point data compression at 75 gb/s on a gpu. *GPGPU-4*, November 2010.

[6] Sean Ellis. Details of astc texture compression, 2016.

Appendices

A Project Overview

A.A Example sub appendices

...

B Second Formal Review Output

Insert a copy of the project review form you were given at the end of the review by the second marker

C Diary Sheets (or other project management evidence)

Insert diary sheets here together with any project management plan you have

D Appendix 4 and following

insert content here and for each of the other appendices, the title may be just on a page by itself, the pages of the appendices are not numbered, unless an included document such as a user manual or design document is itself pager numbered.