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 17, 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:

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.
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

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

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.

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.