High Performance Graphics

Zoe Wall - 40182161

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

School of Computing

April 10, 2018

Authorship Declaration

I, Zoe Wall, 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:

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

Several industry leading games developers are attempting to improve performance of their games through use of a technique called Asynchronous Compute. Asynchronous Compute can be described as leveraging scheduling queues directly on the GPU to allow general purpose calculations on the GPU (GPGPU) to be executed concurrently with rendering tasks. This project aims to provide an introductory analysis into the techniques of Asynchronous Compute using a low-level graphics API, Vulkan. Including, performance evaluations of such implementations.

This aim is achieved through the design, development and experimentation of a 3D rendered application with different Asynchronous Compute methodologies. For the purpose of this project, a resource intensive scene was rendered in real-time. It was not sufficient to be a scene that is only intensive to render, it also required an element of GPGPU. Within games technology, compute shaders can be used for such tasks as physics calculations. This forms the basis of the scene rendered: a N-body simulation.

An evaluation into the effects of Asynchronous Compute on rendering performance was completed with respect to a non-asynchronous baseline compute implementation. The results show that Asynchronous Compute can improve performance under certain conditions of workload, implementation type and GPU types. A proportional relationship was found to exist between the workload and the speed-up gained for the Transfer Buffer technique. This is a significant result, as it shows that the GPU was being utilised more efficiently.

Further research to include is investigations into larger more complex simulations covering multiple draw calls and pipeline changes.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank my supervisor Dr. Kevin Chalmers for all the words of encouragement and help throughout the project.

I would also like to thank Sam, for my late night 'Rubber Ducking' about anything BUT speed up and efficiency. The one thing I can't ask you about is the one thing I always need to talk about!

I would also like to thank all the lecturers at Edinburgh Napier University that have given me challenges and support throughout my rough 5 years in Edinburgh. After changing course multiple times, I have finally found the subject that I wish to make a career out of. Thank-you for all the advice and the encouragement to continue to learn and grow.

Original recipe lucozade, you were with me until the bitter end.

Chapter 1

Introduction

From the use of computer generated imagery in films and tv, to photo-realistic virtual reality experiences, high-detailed Computer Graphics applications are fast becoming a key instrument in entertainment and education [8]. To be able to keep up with this growing demand for more detailed images rendered in real-time, a focus of high performance computer graphics research is that of offloading general-purpose calculations onto the GPU (GPGPU) [40]. Currently, we are failing to exploit the potential of this hardware due to a lack of knowledge in how best to schedule these general-purpose tasks. This forms part of the motivation for the project. A recent development of NVIDIA GPU architecture [10], allows for a novel improvement on current GPGPU techniques, by allowing for concurrent processing of graphics and general tasks on the GPU - this is known as *Asynchronous Compute* [22].

Several *AAA* games have been released in the past few years that boast a performance increase through the use of Asynchronous Compute. However, evidence shows that Asynchronous Compute has an overhead associated cost (Async Tax) that can ”make or break performance” [1]. There is little to no published research on what exactly this cost is, and how to avoid it. Much of the reviews up to now have been competitive in nature, for example; focusing on, to what extent GPU vendors can leverage Asynchronous Compute to improve performance. There is little literature based on the techniques of Asynchronous Compute, and how it can be achieved providing a substantial challenge for this project.

1.1 Aim and Objectives

The aim of this project is to develop a 3D rendered application to analyse the effect of Asynchronous Compute on rendering performance.

The objectives of the work are as follows:

- Undertake a critical evaluation of the current literature and technology sur-

rounding Asynchronous Compute.

- Gain in-depth knowledge and understanding of a new low-level graphics API, Vulkan.
- Design and implement an application, using Vulkan, that attempts to leverage the use of Asynchronous work queues to improve performance.
- Gather experimental results of this application, following a scientific methodology.
- Analyse and evaluate the results and implementation with regards to performance, and ease of implementation.

1.2 Scope & Limitations

1.2.1 Scope

The deliverables of this project include a real-time rendered scene using Asynchronous Compute. This scene will have multiple modes, including a baseline non-asynchronous technique for performance benchmarking. Results of these experiments will form the basis of the final deliverable: an analysis of the simulation techniques. A N-body particle simulation, will form the scene to be rendered as this includes scope for acceleration with GPGPU.

Testing the overall throughput on a high-end NVIDIA Pascal GPU and researching how job scheduling and code optimisations may be able to help improve the performance, of Async, could be significant to the industry. However, this project does not aim to evaluate these optimisations with respect to multiple different hardware configurations, except in the case for profiling for proof of concept.

1.2.2 Limitations

Although the requirements of this project are to render a complex scene that requires physics and graphics calculations, this project will be limited to a basic particle system without rigid body collisions. One popular use of Compute shaders in computer graphics is that of post-processing effects. This too falls outwith the scope of the project, as a deep understanding of task scheduling is more important than having multiple image processing. Additionally, the deliverables do not specify a game engine. The only user-interactions required within the program will be to aid the performance testing.

1.3 Constraints

There are two major constraints to this project, apart from time: learning enough of Vulkan to be able to utilise the low-level API calls necessary for this project, and lack of current research into such methods of Asynchronous Compute.

An in-depth knowledge of Vulkan and GPGPU will be required for this project. Prior study in computer graphics and concurrent and parallel systems, will aid in this stage, however Vulkan is a more verbose and lower-level language than anything used previously so will take time to learn. Another limitation to this project is that the current research surrounding Asynchronous Compute techniques is sparse. Vulkan is still very-much in development and every effort will be made to keep up to date, providing it does not alter with the results collection.

1.4 Sources of Information

Games such as Doom, which had reports of rendering time gains of 3 to 5ms give context to the project [37]. However, the main sources of information will come from hardware vendors, tech-reviews and graphics documentation. Conference proceedings and presentations will also form a large part of the literature. To inform the experimentation and evaluation stages of the project, peer-reviewed parallelism journals and published books will be used.

1.5 Chapter Outlines

- **Background:** Includes technology review, and gives a short description of specialist topics.
- **Methodology:** Theoretical analysis and description of implementation methods used in completing the project milestones.
- **Results:** Presentation of aggregated results from experiments.
- **Evaluation:** Discussion of results and evaluation of the project as a whole
- **Conclusion:** Summary and reflection of the project with regards to the aim and objectives, including a future work section.

Chapter 2

Background

Much of the current literature within high performance computing investigates the use of GPUs for non-traditional workloads. Studies show that it can be beneficial to performance to offload certain types of computations onto the GPU (e.g [36], [20], [7]). This is known as General Purpose Computing on Graphics Processing Units (GPGPU). The GPU's unique architecture supports data parallel operations due to its many *Streaming Multiprocessors* (SMs). As pictured in Figure 2.1, each SM consists of a set of *Scalar Processors* (SPs) which execute the same instruction at the same time in perfect lock-step, working on different sets of data [16]. This is an example of data parallelism in the form of *Single Instruction Multiple Data* (SIMD), classified by Flynn's Taxonomy [28].

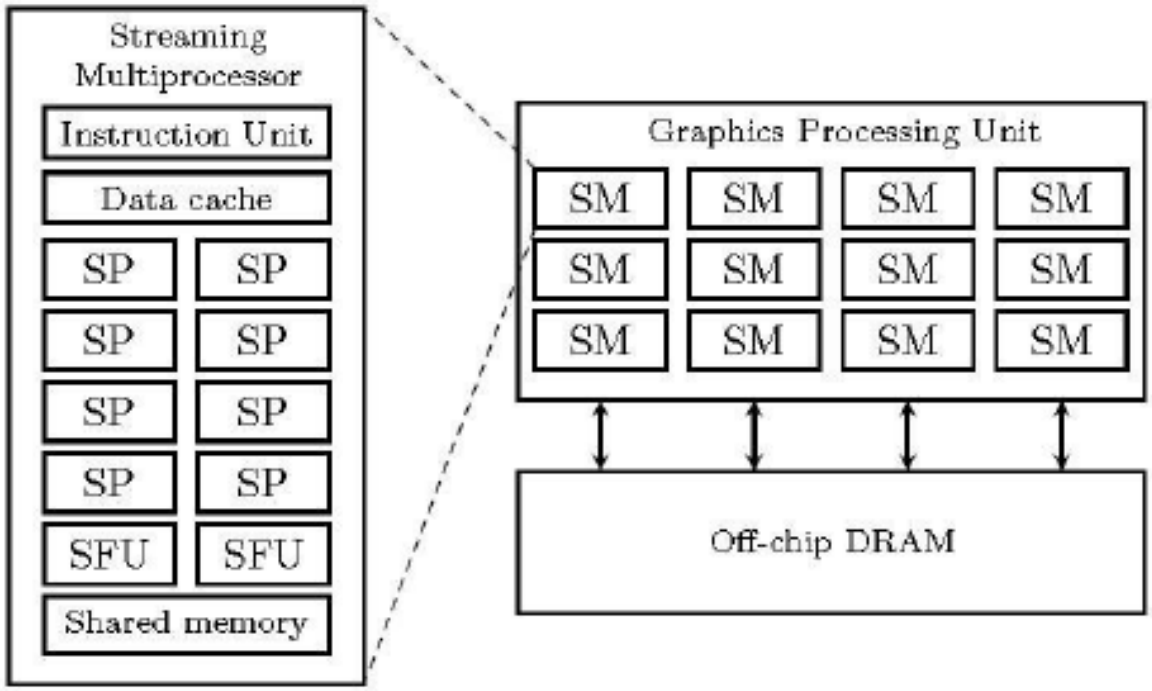

Figure 2.1: **Simplified Diagram of NVIDIA GPU Architecture** - NVIDIA GPUs typically consist of a set of streaming multiprocessors each containing scalar processors (SPs)

GPGPU can therefore exploit the architecture to create massively parallel solutions to accelerate certain programs. These certain problems that are well suited to processing on a GPU are known as ”embarrassingly parallel”, where components can be easily divided into workloads that can be executed concurrently [17]. A case study from 2008, conducted by Owens J. et al. consisted of simulating rigid body physics by performing integration steps on the GPU [35]. The study goes on to mention in the future work section, that ”one concern going forward, however, is the proprietary nature of the tools”. This is evident still, 10 years later: where the battle between AMD and NVIDIA is still going strong [25] and debugging and profiling on the hardware is still primitive. Or at least, it is for the public. GPU vendors are notorious for being secretive and misleading, and the case of Asynchronous Compute is no exception. There has been a lot of speculation and controversies in recent years of so called asynchronous compute. Which is an underlying motivation for this study.

2.0.1 Asynchronous Compute

Gaps between rendering tasks leave part of the GPU idling. Async compute aims to utilise these gaps by scheduling compute tasks concurrently. In the past, async compute has been extensively used on AMD graphics cards with Graphics Core Next (GCN) architecture. These AMD GPUs have Asynchronous Compute Engines (ACE), which are completely separate hardware processors within the GPUs that

are solely responsible for managing compute shaders, whilst the graphics command processor handles the graphics shaders [18]. In this case, having two command queues, one for graphics and one for compute tasks means that each queue can submit commands without waiting for the other tasks to complete.

However, the architecture for NVIDIA cards has always been different. Due to not having discrete compute processors, or a hardware scheduler, asynchronous compute has not been possible on their cards until recently. When the Maxwell 2 (900 series) architecture came out, NVIDIA claimed that it could handle async compute shaders by exposing 31 compute queues and the ability to mix them with the graphics queues [41]. However, there was a controversy as this was not inherently concurrent as it still had no hardware-level scheduler. The biggest problem for the cards is due to static load balancing. Static load balancing is where the GPU would allocate resources ahead of time and not be able to modify them on-the-fly. This meant that if the resources were badly allocated between streaming multiprocessors the performance could take a hit. See Figure 2.2 for a visual representation of this.

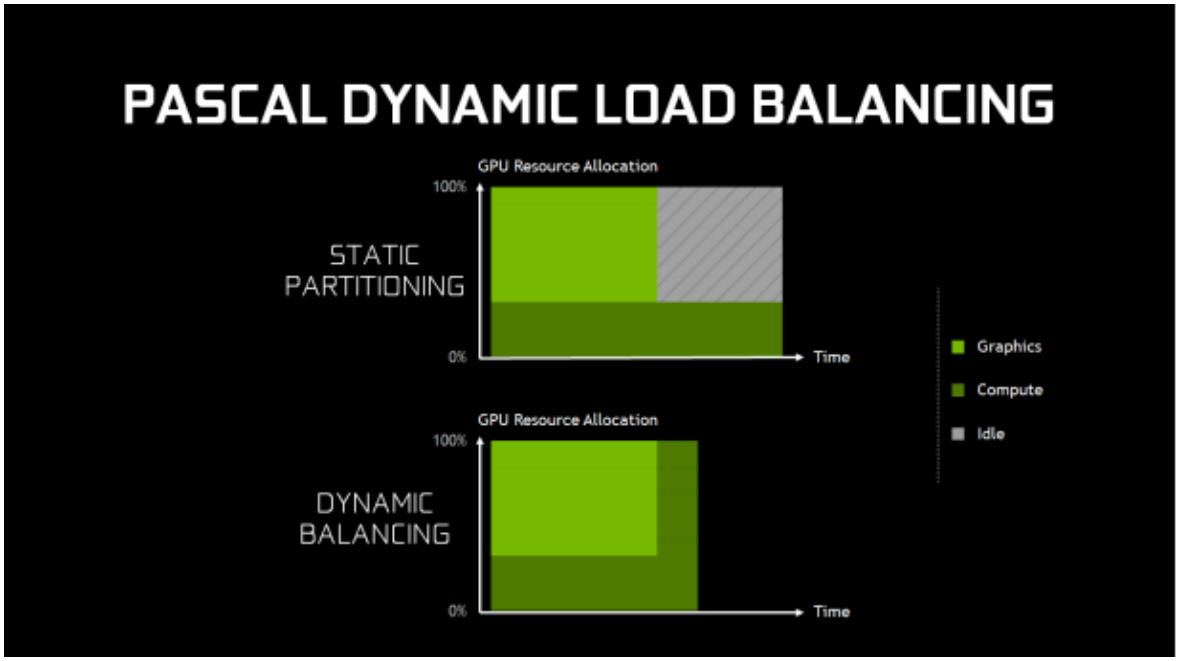

Figure 2.2: **Maxwell vs Pascal** - diagram showing the difference between the different architecture's resource allocation methods[10].

In 2016 when the Pascal (1000 series) architecture was released, it was announced that the GPUs would now have dynamic load balancing which meant that the resource allocation to the GPU can be altered on the fly to allow for asynchronous compute. Note the usage of the once idle time in figure 2.2.

Since this feature is a relatively new capability for NVIDIA cards, the intended research and evaluation of this project will be focused on testing the NVIDIA GeForce GTX 1080. In the future, this project can be expanded to include performance anal-

ysis of multiple types of GPU architectures and whether implementation can match that of the hardware advantage of AMD cards. However, this project first and foremost aims to exploit workload concurrency via async compute and whether or not a Pascal card's performance takes a hit depending on the implementation of Asynchronous Compute.

This project, if successful, will hopefully be useful to developers working on games and real-time simulations. There are very few released titles that currently use asynchronous compute technology. In fact, in researching the topic there seems to be only information and test results of two games: Ashes of The Singularity, and DOOM. Researching in such a new technology has proved to be rather difficult. Most research into compute has been implemented with CUDA for "embarrassingly parallel" problems. In searching for asynchronous compute for rendering performance, relevant results are hard to come by. In fact, the "compute problem" has been outlined in a call for research within "Open Problems in Real-time Rendering" [42].

2.0.2 Vulkan

In order to effectively use compute within this project, an in-depth knowledge of GPU architectures and computer graphics is required. This project requires the use of using a GPU for general-purpose computation via a traditional graphics API and graphics pipeline.

Vulkan, launched on February 16th 2016, is a graphics and compute Application Programming Interface (API) [46]. The Vulkan SDK was chosen to use for development as it is a unification of both graphics and compute functionality and is a relatively new technology. One disadvantage of using Vulkan over a different graphics API is that it is quite verbose. However the benefit of this is that it leads to a greater understanding of errors - what they are or how they occur. Also it gives a finer control over the GPU by requiring to explicitly state every function needed. The verbosity of Vulkan is why it is an ideal choice for this project. Due to the need to explicitly implement scheduling code.

2.0.3 Computer Graphics

Modern GPUs have a programmable part known as the Graphics Pipeline [2]. The pipeline stages that we are interested in for the purpose of this project are outlined as followed:

- The Vertex Shader
- The Pixel (Fragment) Shader.
- The Compute Shader

Computer generated images are based on collections of vertices called meshes that are sent to the *vertex shader*, which is a program or kernel that executes in lockstep. It is an example of Single Program Multiple Data model of execution. Each instruction for every vertex is executed concurrently (depending on the size of the wavefront) - you can only be sure that 64 threads are running at exactly the same time.

The fragment shader interpolates between these vertices and executes instructions per pixel outputting a buffer of colours that the rasteriser uses to display to the screen.

The compute shader acts very similar to a GPGPU compute kernel in other high performance computing areas. For the purpose of this project, the compute shader will be used for physics calculations, a widely used methodology in the gaming industry [21]. There are other optional programmable parts of the pipeline that can be used for optimisations such as the Geometry shader. However this is outwith the scope of this project and reserved for future improvement of the project.

2.1 Evaluation Methodology

The project consists of two main aspects of evaluation. The first, a quantitative evaluation of performance metrics. The second, a qualitative evaluation of the cost of implementation.

A widely accepted method for quantifying the performance of an optimisation are the metrics of speed up and efficiency. ”Speed-up is given by the Sequential time divided by the Parallel time. [23]” For the purpose of this project, speed-up can be calculated given the baseline time divided by the asynchronous time.

For the qualitative information, a 1985 study by Victor R. Basili, details some interesting metrics to be considered [5]. Objective metrics include absolute measures such as; the development time, the number of lines of code, or the number of changes. These metrics are seen to be product metrics, as they are a measure of the actual product developed. However, subjective metrics can sometimes be seen as more useful or reliable than objective metrics. Quality measures represent some form of value of the product. Some examples of quality measures that can be used in the evaluation of this project are, the ease of change, and the amount of code reusable. This study shows that the ease of change can be measured by two metrics; effort and confinement. It details the effort to change on a scale of *trivial* (less than 1 hour) to *formidable* (more than 1 month). Whereas confinement of changes is limited to one section, or more than one section.

2.2 Summary of Findings

These technologies are described further in the following chapters of this report. A brief summary of points to take from this section include:

- GPGPU can be seen to improve performance of embarassingly parallel computational tasks, such as physics for games.
- Asynchronous compute has been successfully used in several AAA titles, improving the performance of these games.
- NVIDIA Pascal architecture allows for concurrent processing of compute and graphics, forming the feasibility of the project.

Chapter 3

Methodology

The methodology as follows, is broken down into milestone stages taken from the initial project overview document - see Appendix .1. Step 5, the optimisation step introduces a separate technique than the first tested in Step 3. For clarification, each simulation has been given a name based on the implementation; baseline compute, asynchronous transfer buffer, asynchronous double buffer.

1. Rendering Engine Using Vulkan
2. Rendering the Scene
3. Rendering Using Asynchronous Compute
4. Profiling
5. Optimisations for Asynchronous Compute
6. Evaluate Performance

3.1 Rendering Engine Using Vulkan

In the early stages of the project, the Vulkan Tutorial website was utilised to get started with the API [34]. Vulkan itself is a very verbose graphics API, which meant that in the beginning there was a large learning curve. Within Vulkan, every detail has to be stated explicitly for it to work. Unlike OpenGL, Vulkan has a validation layer framework which allows for debugging. This proved to be very useful as a learning and development tool. Another useful library used for the development was the Graphics Library Framework (GLFW), that “provides a simple API for creating windows, contexts and surfaces” [43]. This was used to create a window for rendering the graphics to. Overall, it took around 2,000 lines of code for an initial rendering window to display a triangle using GLFW and Vulkan. Listed below are detailed parts of this initial implementation, focusing on important topics unique within the

Vulkan API.

- Execution Model
- Rendering to a screen
- Graphics Pipeline
- Memory Considerations

3.1.1 Execution Model

Once initialised, the Vulkan API exposes *devices* to the developer. A device being a supported GPU, connected to the system. In Vulkan, it is necessary to explicitly state which device to use, as each device, whether it be integrated (part of the CPU) or dedicated, has different capabilities depending on the hardware. The chosen processing unit is then selected as a *physical device*. The focus of this project relies on executing code to multiple types of *queues*, which are exposed within this step. A *logical device* is then created to specify which features of the physical device you wish to use. In this example, separate queues are required as they have the ability to process work asynchronously to one another. Queues are allocated from *queue families*, which specify the type of operations the queues can perform. There are three queue families of interested for this project; compute, graphics and transfer. Theoretically, commands submitted to each queue can be executed asynchronously, which is required for this research. This is also why the project could only be completed with physical devices that support these types of queues, as different GPUs have support for different families of queues.

Vulkan is unique in its ability to schedule tasks through the use of buffers containing commands that are submitted to one of these queues. *Command queues* are allocated from a pool that is created - by specifying the type and amounts of each queue required. This step is similar to reserving a space on the current physical device, providing it is capable. To execute work on a queue, commands must be pre-recorded in memory and stored into a *command buffer*. These can then be submitted on a corresponding queue multiple times. Such commands fall into different *queue families* (categories). For the compute queue, the commands include kernel launches and for the graphics queue, they include draw calls. It is also here where you are required to set-up any barriers or semaphores that are required by the application. For example in a simple compute scenario, you need to ensure that the compute invocations have finished accessing the device memory before the graphics queue accesses it - avoiding a race condition.

3.1.2 Rendering to a screen

Once the window was initialised using GLFW, it was first necessary to create two new Vulkan components to allow for rendering to the GLFW window: a window surface and a swap chain.

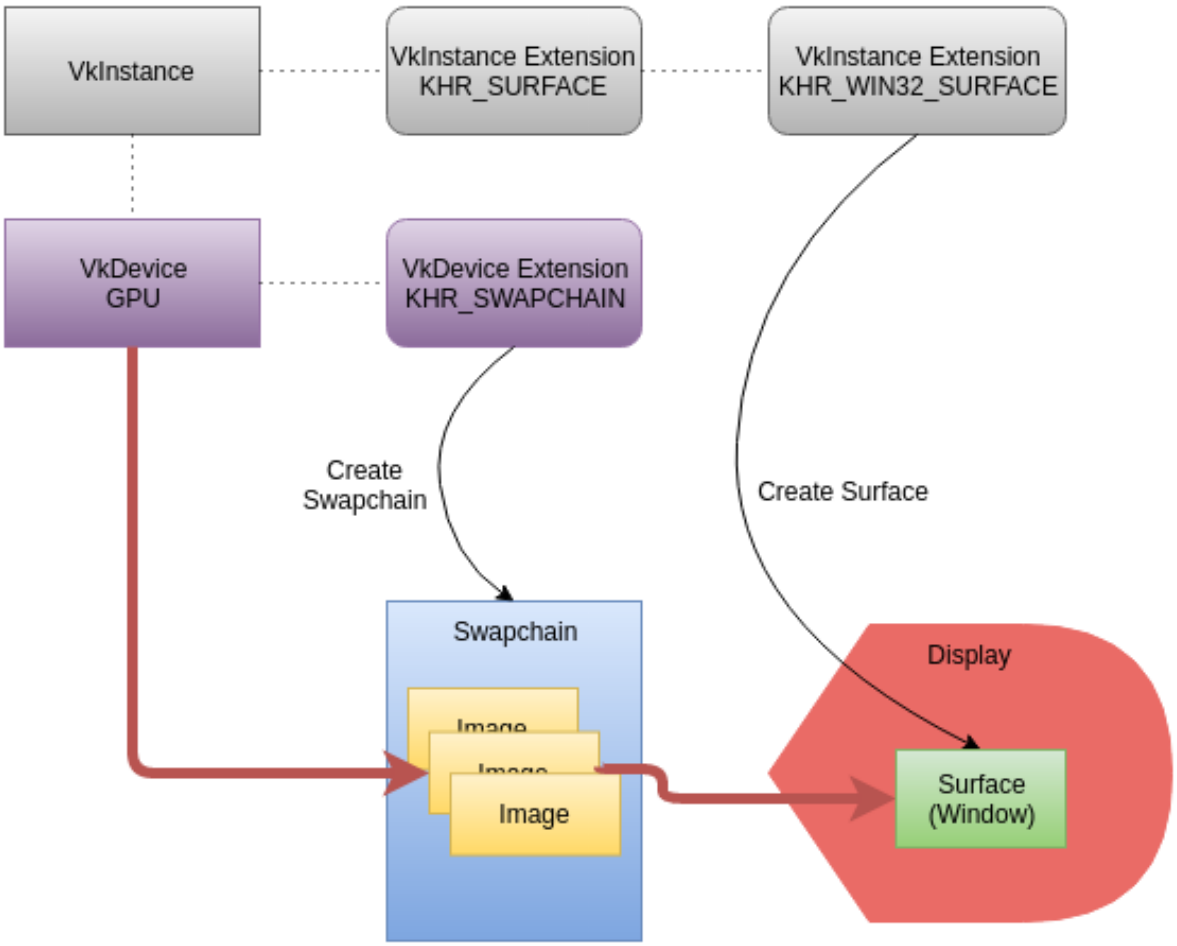

Figure 3.1: **A diagram to show the Vulkan swapchain in relation to the system.** - *The swapchain pictured here consists of 3 image views representing a technique called triple buffering* [19].

Window surfaces require platform-specific handles. An example of this is shown in Figure 3.1 - the Microsoft Windows Extension. A benefit of using GLFW for the window creation however, allows for easily interfacing with the windows system, avoiding having to set this up manually.

Another aspect of creating this render target is to ensure the logical device previously created allows for a presentation queue. There is an important distinction to be made here: a frame that has been rendered, means that it has been processed by the graphics pipeline, it does not mean that it is visible. A frame must be submitted to the *present queue* to be shown on screen. Therefore, rendering in this sense, does not equal displaying on screen.

The swap chain is essentially a buffer that holds multiple frames, also known as render targets. It is a collection of pointers to locations of blocks of device memory. These blocks of memory, or render targets, can either be processed by the graphics pipeline (rendered) or displayed on screen (submitted to the presentation queue). The amount of frames that a swap chain consists of depends on the type of rendering, for example: immediate mode, double buffering or triple buffering.

Typically, you would want at least two of these frames within the swap chain. This is known as double buffering. Double buffering allows for 1 frame to be rendered (drawn to by the graphics pipeline) whilst the other is being presented on screen. Once the rendering is finished, the buffers are ”swapped” so the screen updates at once - not pixel by pixel. This is to mitigate a problem with Immediate Mode - writing directly into video memory - which likely causes screen tearing.

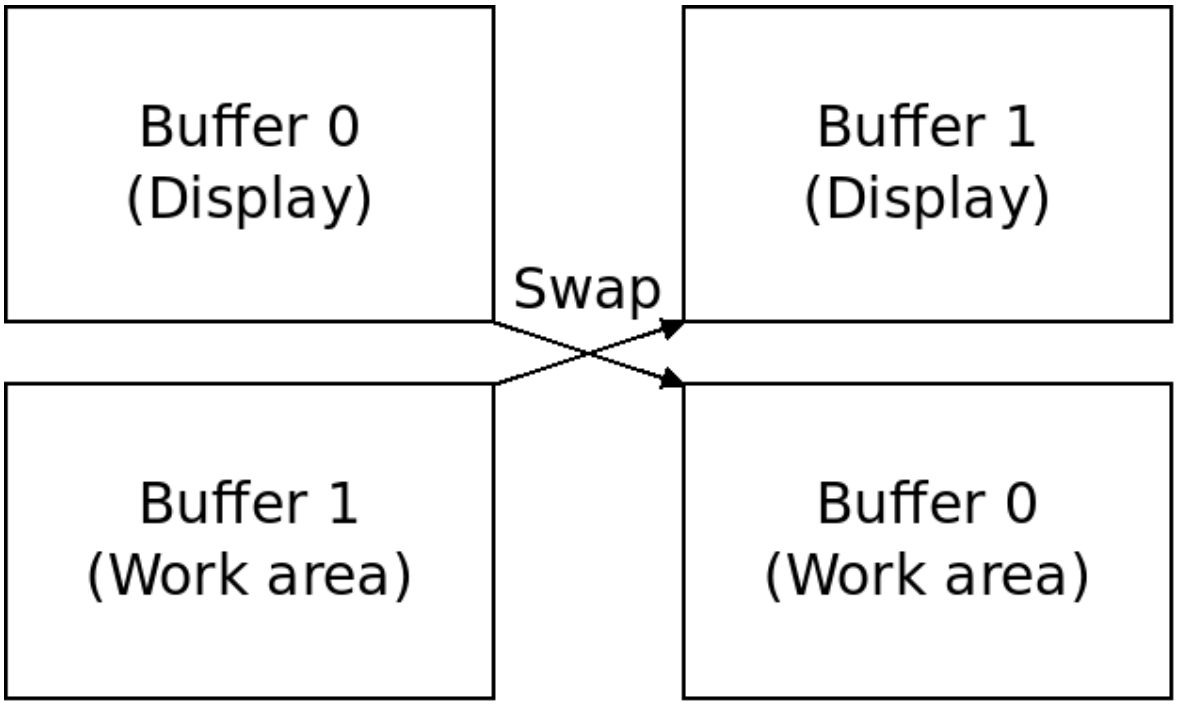

Figure 3.2: **A diagram of double buffering** - *In double buffering there are two copies of the frame at a time, one for modifying, one for displaying.* [14].

In the case of the first two simulations, triple buffering was used. Triple buffering allocates a third frame buffer, working additionally with the 2 others. Instead, to simplify things for the second asynchronous technique, double buffering was used - hence the name!

3.1.3 The Graphics Pipeline

The graphics pipeline (also known as the render pipeline) is a sequence of processes that take in vertex data (typically 3d models) and outputs pixels to a frame buffer. It is compromised of two different types of stages, fixed-function stages,

and programmable stages. The *fixed-function* stages of the pipeline are predefined, whereas the programmable stages (shaders) must be defined. Although, some of these programmable stages are optional, for this project, the tessellation and geometry shaders are not required.

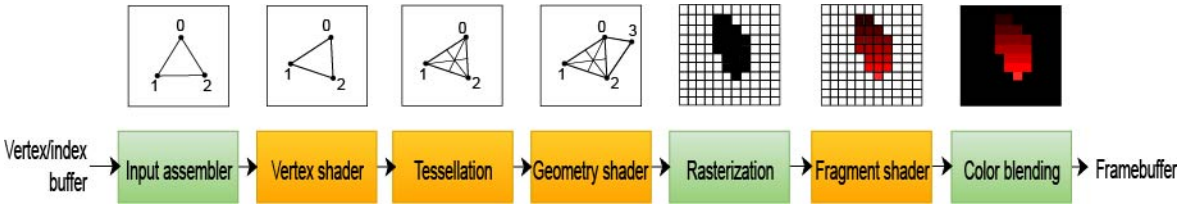

Figure 3.3: **A simplified view of the graphics pipeline** - *highlighted in green are the fixed-function stages, and highlighted in yellow are the programmable parts* [31].

In Vulkan, the pipeline must be explicitly created, and unlike other graphics APIs, it is almost completely immutable. This means that you must re-create the pipeline from scratch if you want to change some aspect of the implementation. For example: having different blend modes for different objects within the scene. However, as these operations are known in advance, the graphics driver can optimise this. Defining a graphics pipeline object requires both a *pipeline layout* and a *render pass* object.

Pipeline Layout

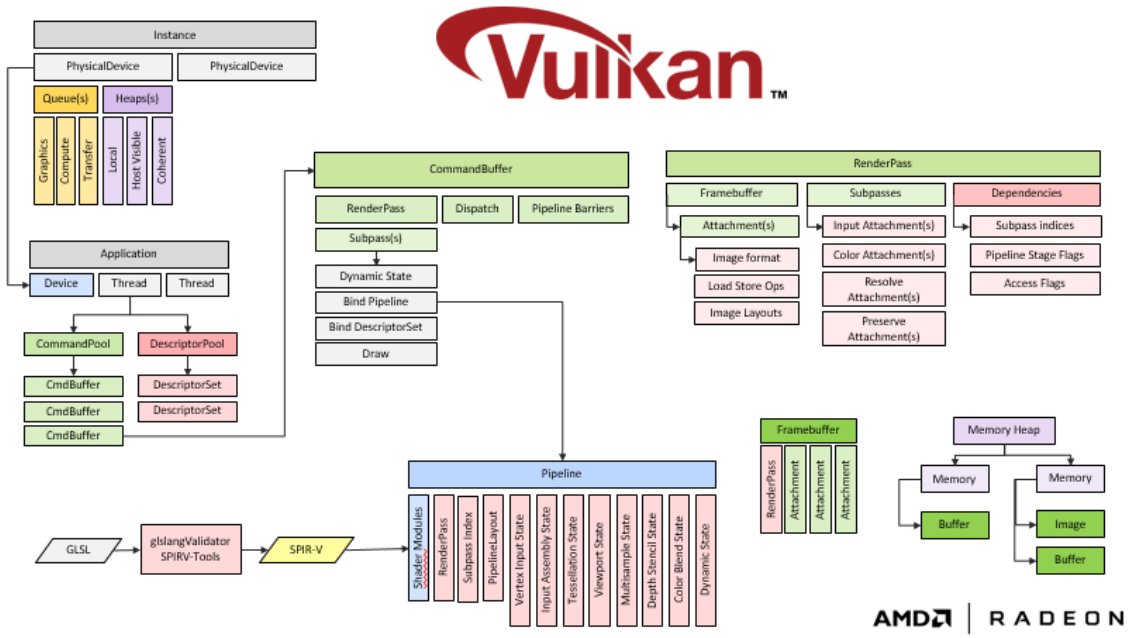

Figure 3.4: **Vulkan API overview** - *A diagram showing a high-level overview of the Vulkan API [30].*

A pipeline layout object, contains a list of objects called *descriptor sets*. A descriptor set, again allocated from a pool, is an object containing information that is required to pass the memory locations of the data to the processes. This information is passed to the set through a descriptor set layout [26]. The layout contains details about the type of buffer that is used, binding locations and the pipeline stages. Binding locations can be likened to that of channel inputs and outputs to the specified pipeline stage. For each buffer of data within the GPU, there must be a corresponding descriptor set.

Configuration of the fixed-function operations is also required in Vulkan, which occurs within the *pipeline layout* creation. These include rasterization operations such as: face culling - removing triangles that are on the inside of shapes [44]; and polygon modes - determining how fragments (pixels) are generated from the mesh [32].

For the programmable parts of the pipeline, shader modules are created. A shader is the name for a program that runs on the GPU. Vulkan expects these shaders to be written in byte-code known as Standard Portable Intermediate Representation for Vulkan (SPIR-V). SPIR-V is an intermediate language for parallel compute and graphics applications [3]. Included within the Vulkan SDK, is an application that converts the human-readable open standard for shading languages, GLSL, into SPIR-V. The shaders written for this project have all been written in GLSL before being converted into SPIR-V for use in the application.

Render Pass

Finally, before creating the pipeline, a render pass object must be defined. A render pass contains details of frame buffers and attachments that are required by the stages within the pipeline. A render pass should specify how many colour or depth buffers are required, and how their contents should be handled. For example, in using a texture, the image must be loaded into a VkImage object, and it is at this stage where you must determine how the pixels are interpreted, and how many times the image is sampled. For loading in the texture files into memory, a simple header-only library called 'stb_image' is used [4].

3.1.4 Memory Considerations

Almost all objects in Vulkan are initialised through the use of a specific CreateInfo struct, including the objects mentioned previously. An example of this in use can be seen in lines 3-17 of Listing 3.1. Each object must also be explicitly destroyed in the correct order at the end of the program to avoid memory leaks.

Creation of Buffers

Another unique aspect of Vulkan is that memory management for objects like buffers and texture images must be set-up from scratch. For this project, a helper function was written to aid in the creation of the various buffers required for the simulation.

Listing 3.1: "Create Buffer Helper Function"

```
1  void Renderer::createBuffer(VkDeviceSize size, VkBufferUsageFlags usage, VkMemoryPropertyFlags properties, ↩
       VkBuffer& buffer, VkDeviceMemory& bufferMemory)
2  {
3    // create struct as usual
4    VkBufferCreateInfo bufferInfo = {};
5    bufferInfo.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
6    // how big is the buffer (enough to store the size of 1 obj * how many)
7    bufferInfo.size = size;
8
9    // what purpose is this buffer going to be used for ( can use multiple things)
10   bufferInfo.usage = usage;
11
12   // buffer only used by the graphics queue so it can be exclusive
13   bufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
14
15   // create it
16   if (vkCreateBuffer(device, &bufferInfo, nullptr, &buffer) != VK_SUCCESS)
17     throw std::runtime_error("Failed to create buffer");
18
19   // Memory requirements
20   VkMemoryRequirements memRequirements;
21   vkGetBufferMemoryRequirements(device, buffer, &memRequirements);
22
23   // determine memory type to allocate
24   VkMemoryAllocateInfo allocInfo = {};
25   allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
26   allocInfo.allocationSize = memRequirements.size;
27   allocInfo.memoryTypeIndex = findMemoryType(memRequirements.memoryTypeBits, properties);
28
29   // allocate memory
30   if (vkAllocateMemory(device, &allocInfo, nullptr, &bufferMemory) != VK_SUCCESS)
31     throw std::runtime_error("Failed to allocate buffer memory");
32
33   // bind memory to the buffer (int is offset)
34   vkBindBufferMemory(device, buffer, bufferMemory, 0);
35 }
```

After a buffer has been created, the device memory must then be allocated to that buffer. Graphics cards have different memory types and distinct memory resources, so it could be a performance issue if the wrong type of memory is used. Another helper function was created to find the memory type suitable based on the requirements of the buffer and the requirements of the application [33].

Two types of buffers used within this project are vertex buffers and uniform buffer objects. The vertex buffer is what holds the geometry data of a certain object. Position, normal, colour, and texture coordinates, are the typical examples of data per vertex that this buffer contains. Offsets of this buffer, are bound to a location within the vertex shader to be processed.

Generally, vertex shading takes each incoming vertex position and transforms it into a different coordinate space [39]. The incoming position for each vertex within the

mesh is normally based around the centre of the object at the origin, (0,0,0) in world space. World space relates to the coordinates of the entire 3D scene. A model matrix (specific to the object) translates, scales or rotates the object into a position in the 3D world. A view matrix is created with respect to the camera of the scene, and transforms a position from world space into camera space. Finally a projection matrix, transforms the position from camera space (3D) into screen space (2D). These transformations are the basic operations usually performed within the vertex shader.

To transform positions between coordinate spaces, the Model-View-Projection (MVP) matrix is required to be bound into the shader as a uniform. Uniform buffer objects are different to normal shader inputs and outputs as they do not change between executions of different shaders - within a particular rendering call. Uniforms cannot be altered within the shaders and must be updated per frame on the CPU before binding to the shaders. For example, to rotate an object, the MVP matrix needs to be recalculated according to the delta time, of each frame. As the uniform buffer is stored on the graphics card, to update this you must have a copy of the buffer on the CPU which is generally known as a Uniform Buffer Object. Refer to Listing 3.2 for an example of this.

Listing 3.2: ”Updating Uniforms on the GPU”

```
1 // map the memory from the uniform buffer object (cpu) to the device.
2 vkMapMemory(device, compute→ uboMem, 0, sizeof(compute→ ubo), 0, &compute→ mapped);
3 // copy the memory from host to device
4   memcpy(compute→ mapped, &compute→ ubo, sizeof(compute→ ubo));
5 // unmap the memory
6   vkUnmapMemory(device, compute→ uboMem);
```

3.1.5 Summary

This previous section details all the important steps taken to complete the first milestone. It was a large learning curve, as the Vulkan API is extremely verbose and lower-level than most other graphics APIs. Below is a screen capture of the output of this Vulkan test implementation.

3.2 Rendering The Scene

N-body simulations consist of thousands of dynamic particles. Typically, simulations use points to represent the particles, however for the purposes of this project, spheres are used instead to For this to be rendered on screen, a technique called instance rendering was used. Instance rendering or Instancing, is a technique that allows for rendering “multiple copies of the same mesh in different locations” [45]. This is a performance optimisation, as rendering multiple objects within a scene normally involves multiple draw calls with updates to uniform buffers in between them. This is very expensive, however instancing solves the need to do this.

For the simulation to work each particle in the scene must have a 'instance'position and a velocity. In additon to the standard vertex buffer containing the vertices of a sphere, formed around the world origin (0,0,0). The sphere mesh was generated procedurally rather than an imported model, so the amount of vertices used could be changed as a parameter in the experiments. To generate an approximate sphere out of triangles, the sphere is "subdivided around the Z axis into slices and along the Z axis into stacks" [24]. This method of subdivision allows for a minimum enclosed shape using only 54 triangles consisting of 3 stacks and 3 slices [13].

The positions of the vertices of the triangles are stored in a buffer, along with texture coordinates (uvs), normals and indices. Indices were used for rendering which allows for the renderer to re-use positions where a vertex is shared by multiple triangles. Shown below in Figure 3.5 are some screen captures noting the difference between spheres depending on the resolution of the mesh.

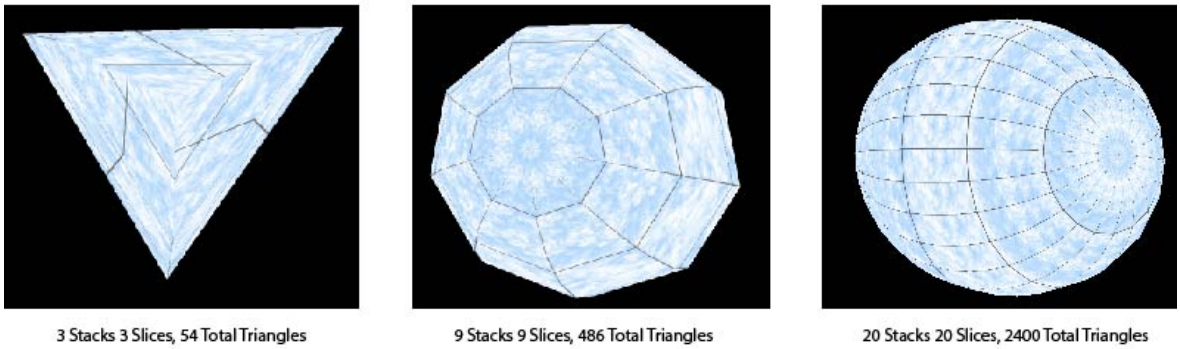

Figure 3.5: **An example of various mesh resolutions.** - *Procedurally generated spheres for use as particles in the simulation, each using more triangles exponentially.*

Instancing requires a second buffer, containing the positions and velocities for the separate instances of the spheres. This was generated using a random number generator, within the range -10 to 10, for both the x and y positions. This storage buffer is what is altered by the compute shader to change the positions of the particles.

3.2.1 Vertex Shader

Typically, a Uniform Buffer Object bound to a vertex shader would consist of the Model View Projection (MVP) matrix for efficiency reasons. However, due to the positions of the instanced particles being set by the compute shader, separate model view and projection matrices are bound instead. This implementation uses the maths library GLM [15]. See Listing 3.3 for an example of how the matrices are multiplied to achieve the correct screen position for each particle.

Listing 3.3: "**Example of Matrix Multiplication within the Vertex Shader**"

```
1  // translation matrix
2  vec3 v = instancePos.xyz;
3  mat4 m = mat4(1.0);
4  m[3] = m[0] * v[0] + m[1] * v[1] + m[2] * v[2] + m[3];
5
6  // scale matrix
7  mat4 s = mat4(1.0);
8  s[0] = s[0] * instancePos.w;
9  s[1] = s[1] * instancePos.w;
10 s[2] = s[2] * instancePos.w;
11
12   //Incoming model matrix is the rotation matrix.
13   // calculate TRS matrix
14 mat4 trs = m * s;
15 mat4 model = trs * ubo.model;
16   gl_Position = (ubo.proj * ubo.view * model) * vec4(inPos, 1.0);
```

Note the order of multiplication is important. Within this implementation, the instance position is converted into a translation matrix (line 4, in Listing 3.3). From the Translation, Rotation and Scale matrices, the Model matrix is calculated. The incoming vertex (inPos), is then multiplied by the MVP matrix to transform the position from world, to camera, to screen space. The full shader code can be seen in Appendix Listing 1.

3.2.2 Compute Shader

The purpose of the compute shader is to process the physics calculations of the N-body simulation. Each particle's velocity is influenced by the distance between it and every other particle in the scene. It was decided to implement a brute force algorithm due to it's $O(n^2)$ complexity. This would hopefully affect the results, as exponentially more work is required when the particle count increases. The compute shader program in full can be found in Appendix Listing 2.

3.2.3 Fragment Shader

The fragment or pixel shader is the last programmable step in the render pipeline. This takes in a fragment/pixel from the rasterizer stage and converts it into a colour and depth value. For this application, the fragment shader simply samples part of the texture that is bound to it, and outputs a final colour.

With these shaders completed, and then converted into SPIR-V. Each simulation type could now be implemented using this pipeline. First a baseline compute example was developed.

3.2.4 Baseline Compute Example

Implementation of the original compute example consists of: 1 memory buffer, 1-2 queues.

For this example, both the graphics and compute commands can be submitted to the same queue if it's family supports both operations.

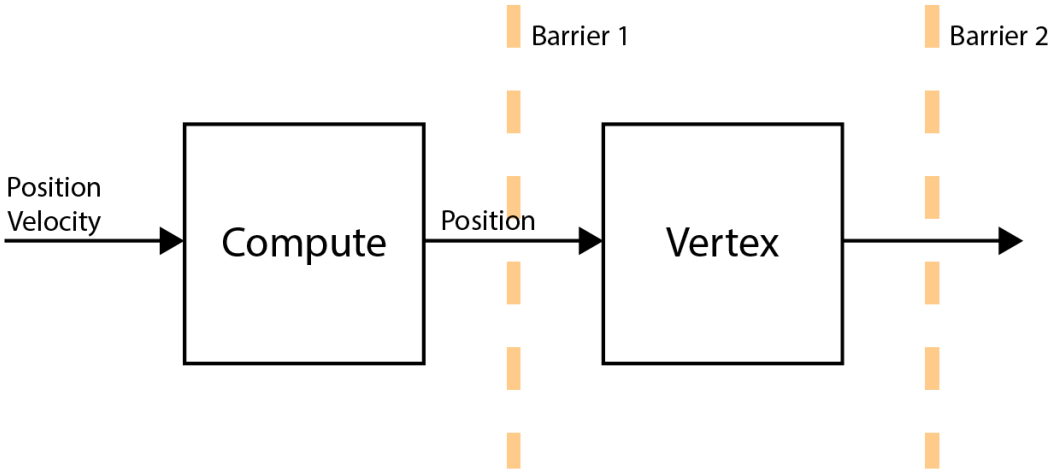

Figure 3.6: **A simplified diagram of the baseline compute simulation.** - *Procedurally generated spheres for use as particles in the simulation, each using more triangles exponentially.*

The instanced particle buffer, which consists of both the particle's position and velocity, is sent to the compute shader. There is a barrier in place that means the draw call will wait on when the compute shader has finished writing to the particle buffer, before being accessed by the vertex shader.

3.3 Rendering using Asynchronous Compute

A small number of examples of so called "asynchronous compute" shading have emerged on line, however the accuracy of these examples are not verified. One such example that was explored was used for inspiration for the first simulation [50].

3.3.1 Asynchronous Technique 1 - Transfer Buffer

The example classified asynchronous compute using a transfer buffer. This consisted of having 2 identical storage buffers for the positions and velocities of the particles (instance buffers). The first buffer is accessed only by the computer shader, where the second is accessed only by the graphics shader. Once the compute shader had altered the buffer for 1 frame, a transfer command was executed to copy the data from that buffer to the other storage buffer, which the vertex shader would read from. This transfer technique can be seen to be asynchronous as each of the shaders would be able to access their own memory at the same time.

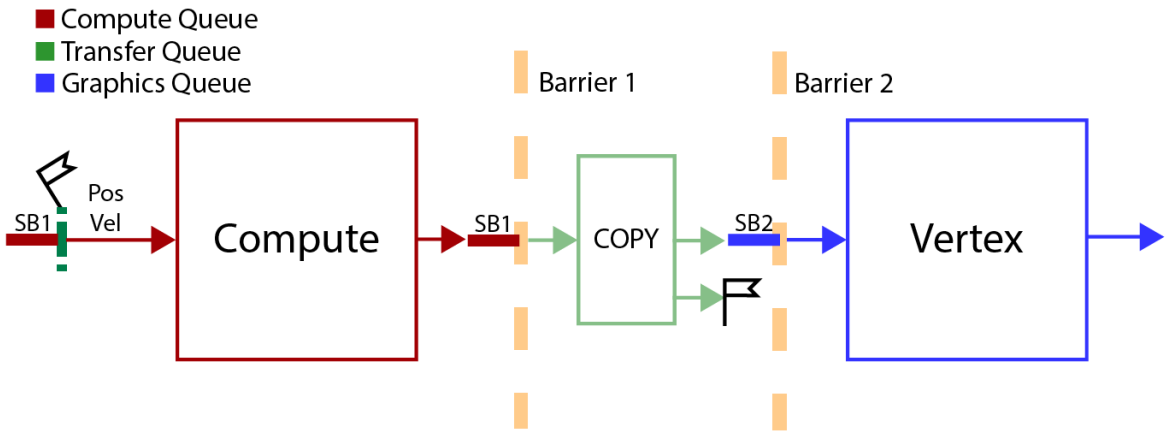

Figure 3.7: **A simplified diagram of the Asynchronous Transfer simulation.** - *The diagram shows each command queue in a different colour. There are two separate storage buffers containing the instance data (SB1 & SB2). The compute shader will wait for the compute fence to be reset (signaled by the completion of the copy operations). Pipeline barriers are implemented between the compute write and transfer read, and the transfer write and vertex read.*

To achieve this implementation, the command buffers had to be recorded differently from the baseline example. For the compute commands, the barriers were removed entirely, therefore the resulting compute commands were simplified - see Lines 14-23 of Listing 3.4.

Listing 3.4: "**Asynchronous Transfer Simulation Compute Command Buffer Recording**"

```
1  void trans_simulation::recordComputeCommands()
2  {
3    // create command buffer
4    // Compute: Begin, bind pipeline, bind desc sets, dispatch calls, end.
5
6    // begin
7    VkCommandBufferBeginInfo cmdBufInfo{};
8    cmdBufInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
9
10   if (vkBeginCommandBuffer(compute→ commandBuffer, &cmdBufInfo) != VK_SUCCESS)
11     throw std::runtime_error("Compute command buffer failed to start");
12
13   // bind pipeline & desc sets
14   vkCmdBindPipeline(compute→ commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, compute→ pipeline);
15   vkCmdBindDescriptorSets(compute→ commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, compute→ ↩
         pipelineLayout, 0, 1, &compute→ descriptorSet, 0, 0);
16
17   vkCmdResetQueryPool(compute→ commandBuffer, renderer→ computeQueryPool, 0, 2);
18   vkCmdWriteTimestamp(compute→ commandBuffer, VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT, renderer→ ↩
         computeQueryPool, 0);
19
20   // dispatch shader
21   vkCmdDispatch(compute→ commandBuffer, renderer→ PARTICLE_COUNT, 1, 1);
22
23   vkCmdWriteTimestamp(compute→ commandBuffer, VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT, renderer→ ↩
         computeQueryPool, 1);
24
25   // end cmd writing
26   vkEndCommandBuffer(compute→ commandBuffer);
27
28   // Set up memory barriers needed for compute and draw
29   recordTransferCommands();
```

```
30 }
```

The compute command buffer simply binds the necessary buffers and executes the same shader as described in the baseline simulation. However, the commands are only submitted on the queue once the compute fence has been reset. This is implemented by a pseudo spin-lock function, repeatedly checking if the transfer queue has finished. The transfer queue, is a separate queue family to both the compute and graphics queues to enable asynchronism. The transfer command buffer to be submitted on this queue, with the compute fence, consists of two pipeline barriers and a copy operation. The copy operation copies the values from the first instance buffer, into the second instance buffer. The first pipeline barrier, ensures that the compute shader has finished writing to the buffer before the transfer command reads it. Where as the second pipeline barrier, ensures that the vertex shader has finished reading the buffer before it overwrites the data. The graphics command buffer is exactly the same as within the baseline simulation.

3.4 Profiling

Due to the proprietary nature of GPUs and their drivers, publicly available profilers with the correct support for different devices are difficult to come by. Some profilers only work with certain vendors hardware, whilst others only support DX12 or OpenGL. A major limitation caused by the age of Vulkan, is that not only do many profilers not support it fully yet, but also it has been difficult to get consistent results due to driver patches and version updates. At the start of this project, the latest version of Vulkan was 1.0.61.1. When first attempting to profile, this was updated to 1.0.65.1 to keep it up to date. However, in an effort to keep the simulation results reliable, it was not upgraded further. From the time experiments were being recorded, two more updates have been released, and even more recently Vulkan 1.1 has now been launched [47].

At the time of writing, there was no profilers found that fully supported Vulkan running on Windows operating system using a NVIDIA GPU. Most profilers found had only support for a subset of these conditions, which is why the decision was made to source an AMD GPU to gather additional results.

3.4.1 GPUOpen

The only program to be found which would explicitly state results of asynchronous compute was GPUOpen. But only for AMD cards. This lead to sourcing of an AMD graphics card. It was reasoned that, if the profiler results showed asynchronous compute then it would be okay to assume that it would be approximately the same result for a NVIDIA card.

3.4.2 Summary of Preliminary Testing

Once this example had been converted to work with the original implementation, it was noted that due to the programmer not setting-up the queues correctly that this was actually not asynchronous. Tests were conducted by running the program through several profilers, (reference to profiling section below), which showed that the program did not work as stated.

At this point, it was decided to find an alternative simulation that fit the brief. Which leads us to the Asynchronous Double Buffering technique outlined below.

3.5 Optimisations for Asynchronous Compute

Not only at this stage were optimisations made for the transfer technique, another simulation was developed using a different technique. The implementation for this technique was based on a different example found on line [6].

3.5.1 Asynchronous Technique 2 - Double Buffer

The Asynchronous Double Buffering simulation requires much more set up that the first simulation. The implementation includes: 2 separate queues, one with graphics capability the other with compute capability; 4 command buffers, 2 copies of the compute commands and 2 copies of the graphics commands; and 2 copies of the instance buffer.

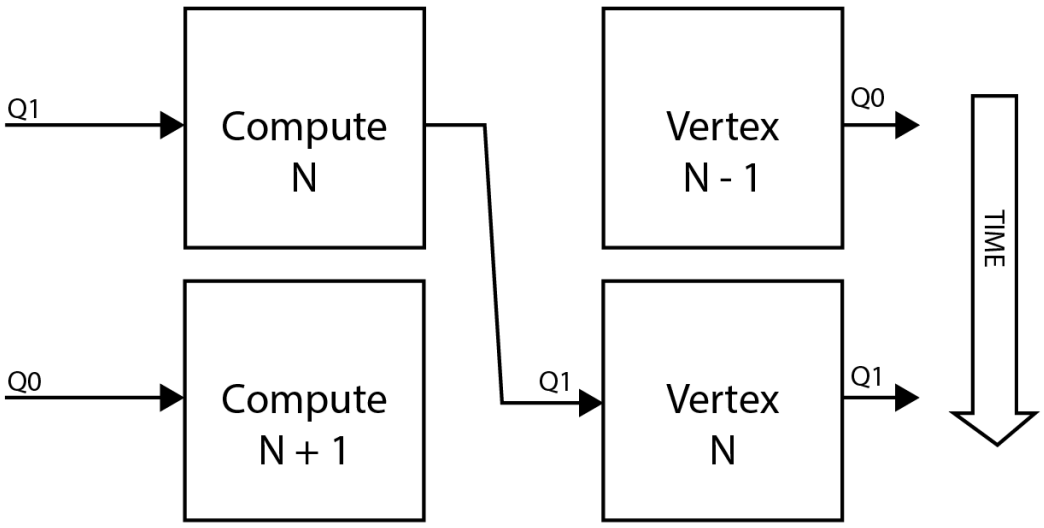

Figure 3.8: **A simplified diagram of the Asynchronous Double Buffer simulation.** - *The diagram shows two frames, the first is where the output of the first compute dispatch is not used until the next frame, whilst the vertex shader is processing the last frame's data.*

The way this simulation works is by swapping the command buffers and instance buffers each frame. The compute shader in this example has been slightly modified, to allow for binding two descriptor sets for both of the instance buffers. The first instance buffer is used as an input whilst the second instance buffer acts as an output, each frame this is swapped, to allow for updating of the older instance data. This negates the use of a copy queue (from the 1st async simulation). Listing 3.5 shows the frame method from this simulation.

Listing 3.5: "**Asyncronous Double Buffer frame method**"

```
1  void double_simulation::frame()
2  {
3    renderer→ updateUniformBuffer();     // update gfx uniforms (MVP)
4    renderer→ updateCompute();       // update compute uniforms (delta time)
5    dispatchCompute();                 // submit compute
6    renderer→ drawFrame();             // render (submit gfx)
7
8    waitOnFence(renderer→ graphicsFence);
9    bufferIndex = 1 − bufferIndex;          // switch bufferIndex
10 }
```

3.5.2 Further Implementation

Once more knowledge had been acquired into the inner workings of Vulkan, by successfully implementing the third simulation. The transfer simulation was then updated to reflect true asynchronous compute. The reason why the original example could never be asynchronous was that the transfer, compute and graphics command queues were not distinct, and there were some validation errors in the barrier set-up [50].

Once these simulations were developed, and the results from the profiler showed positive for asynchronous compute. It was then down to experimentations.

To allow for the experiments to be run automatically, each simulation was combined into one single program. Choices such as GPU type, and simulation type could be entered as command line parameters, with the help of the "args" header only library [38]. See Figure 3.9 for a screen capture of the designed help screen.

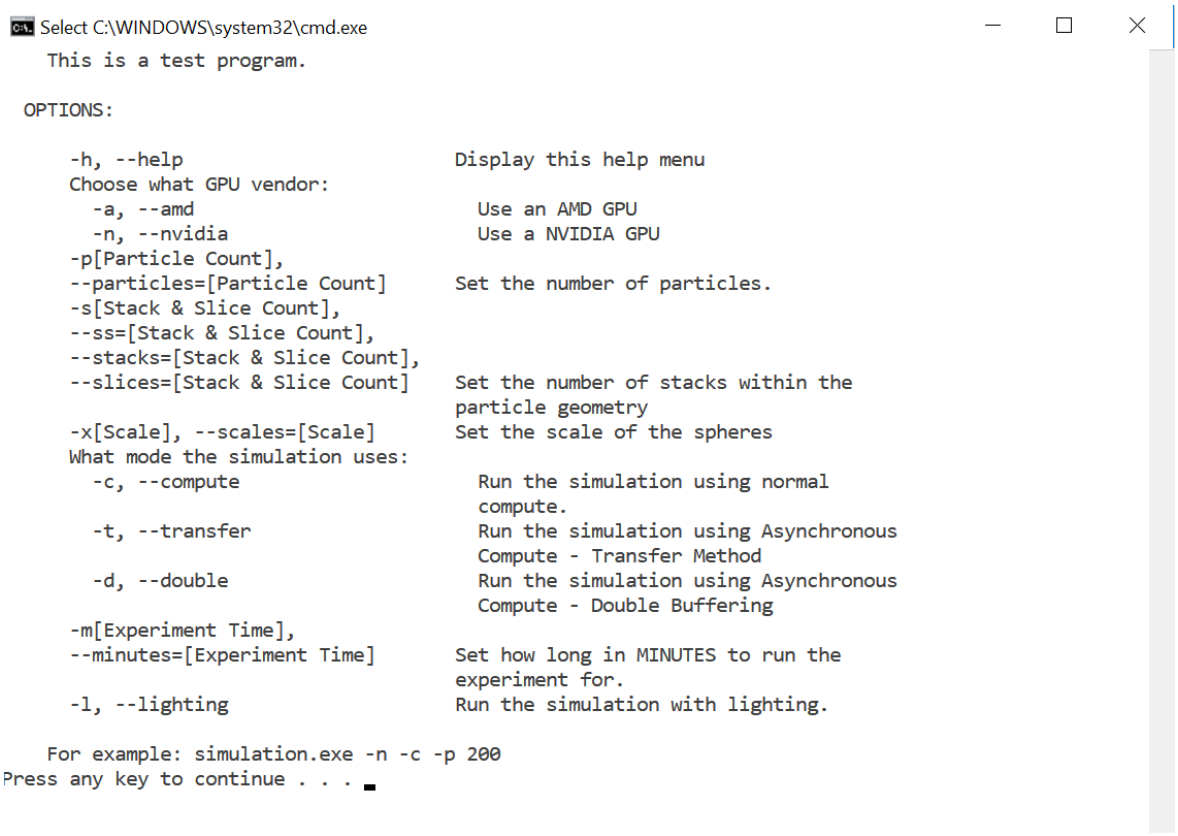

Figure 3.9: **A screen capture from the final application.** - *This shows the help screen for the command line parameters.*

The simulations could have been run separately, but by combining them into concrete implementations of an abstract simulation class meant that it was a lot easier to tell which methods needed to be altered, and which were unique to each technique.

Once the simulations were ready, it was then straightforward to implement a new vertex/fragment shader to include lighting calculations. This was to push the GPU harder, by having to do more operations.

The technique used for the lighting calculations is known as Phong Shading. This is per fragment shading, which means that the operations required to calculate the light were performed once per pixel. Phong shading was combined with normal mapping, to further increase the work.

Normal mapping is a technique in computer graphics to make objects appear more detailed than they are. It requires more data processing as the normals of the object need to be transformed into tangents and binormals. A separate texture, known as a normal map is sampled by the fragment shader, and this sampled texture determines the apparent height of the pixel. This technique doesn't change the actual geometry of the mesh, however it makes it appear so.

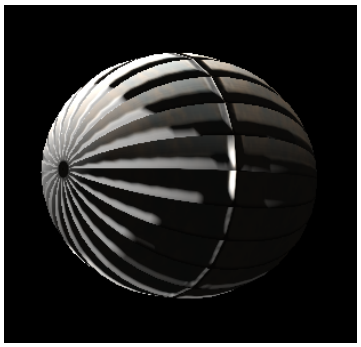

Figure 3.10: **An example of a particle using the normal mapping shader from the simulation.** - *The geometry of this mesh is a sphere, however lighting calculations make it appear to have ridges.*

3.6 Evaluating Performances

To evaluate the performance of each simulation, implemented within the code are extra features such as:

- Frame timings from the CPU.
- Time stamps from the GPU.
- Calculations of the difference between time stamps.
- Outputting the results to a csv.

Frame Timing

Listing 3.6, shows the implementation of the frame time measurements, taken from the main loop of the entire program. The C++ standard *chrono* library was used, as the high_resolution_clock type has the shortest tick period available [11]. The timing only includes the processing of each frame of the simulation, and not the processing or output of results.

Listing 3.6: "**Measuring the frame time**"

```
1  auto startTime = std::chrono::high_resolution_clock::now();    // start timer
2
3  sim→ frame();
4
5  frameCounter++;
6  auto endTime = std::chrono::high_resolution_clock::now();
7  auto deltaT = std::chrono::duration<double, std::milli>(endTime − startTime).count();
```

Time stamps

A feature of Vulkan that allows for time stamps to be recorded is set-up through a *query pool*. "Each query pool is a collection of a specific number of queries of a

particular type. [27]" In this instance, a time stamp query is used, that records the value of the timer "when all previous commands completed executing as far as the specified pipeline stage. [48]

Listing 3.7 shows the placement of these time stamp commands within the implementation of the command buffer recording methods. To begin recording, the query pool must first be reset before requesting the time stamp value be written to the pool. The *'vkCmdWriteTimestamp()'* method takes in the following parameters; the command buffer it is recorded to, the specified pipeline stage, and the querypool. To be able to meaningfully compare the timestamps, two separate query pools were used, one the compute queue, the other for the graphics queue. The given pipeline stage, VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT, was used, as this specifies the stage in the pipeline where operations generated by all commands up to this point complete execution [49]. Therefore, the time stamp command can be placed before and after the dispatch or draw call, to ensure the total command time can be calculated from these values.

Listing 3.7: **Positions of the timestamp query commands**within the recording of the compute command buffer method

```
1  // time compute
2  vkCmdResetQueryPool(compute→ commandBuffer, renderer→ computeQueryPool, 0, 2);
3  vkCmdWriteTimestamp(compute→ commandBuffer, VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT, renderer→ ↩
       computeQueryPool, 0);
4
5  // Dispatch the compute
6  vkCmdDispatch(compute→ commandBuffer, renderer→ PARTICLE_COUNT, 1, 1);
7
8  vkCmdWriteTimestamp(compute→ commandBuffer, VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT, renderer→ ↩
       computeQueryPool, 1);
```

The time stamps were retrieved from the corresponding query pools every frame, only after the frame time was recorded - so as to not interfere with the timing results. These results were used to calculate the timings of the individual compute and graphics commands, and to compare any overlapping times.

$$(results[C_{\text{END}}] - results[C_{\text{START}}]) \times \frac{timestampPeriod}{1 \times 10^6} \tag{3.1}$$

To convert the time stamp into a meaningful timing, Equation 3.1 was used. Where the time stamp difference between the start and the end of the command is multiplied by the time stamp period, and then converted into milliseconds. The '*time stamp period*' is a value unique to the graphics card used. Stored from the device property function used in the creation of the logical device. For the AMD card; this value was 40, whereas for the NVIDIA card; this value was 1.

Asynchronous Comparisons

As the profilers mentioned only supported AMD graphics cards, there needed to be a way of assessing whether the shaders were running concurrently on NVIDIA hardware. Comparing the time stamps from the compute dispatch call versus the graphics draw call allowed for evaluating the asynchrony of the operations, independent of GPU type. Whichever time stamp was the earliest, showed which command buffer started executing first. The end time of this first execution was compared against the start time of the other to give a calculated difference. If this value was positive, it meant that the execution of the shaders were overlapping and therefore asynchronous.

Output to CSV

Each time the simulation was run, the program was designed to output the results of these timing results to a CSV file. The files created were named depending on the command line parameters of the experiment. If the file already existed, it would increment the test number until a new file could be created. An important thing to note, is the results were never stored in the heap memory of the program, the results from each frame were written out to the file. This may have slowed down the overall program, however these timings were not measured, so the results should not have been effected. This was to mitigate any problems with the program crashing before it could finish writing the data.

Testing

With the extra features in place, a Windows batch script was written to automate the testing process. As each configurable parameter could be changed by the command line arguments at execution time.

Configurations explored were that of:

- Number of Particles
- Stack/Slice Count
- Particle Scale
- Lighting

For each iteration of the simulation, and for each graphics card, experiments were run 10 times. At first, each simulation was timed for 1 minute. However, after some

tests giving over 20,000 frames worth of information, the testing was shortened to last for 18 seconds.

An example table from the output of an experiment can be seen in Appendix Table 1.

These results were aggregated and are shown in detail in the following section of this report.

3.7 Summary

The methods of this project can be easily split into stages of implementation planned in the IPO document. All of the milestones were achieved in turn, resulting in a deliverable of a C++ application that has 3 different running modes, using the Vulkan API. The libraries used in the final application were; GLM, GLFW, STB.

The three simulations modes within the final application are as follows: Baseline Compute, Asynchronous Transfer Buffer, Asynchronous Double Buffer.

These simulations were profiled, and tested on both AMD and NVIDIA hardware, and the results were recorded for a number of test configurations.

Chapter 4

Results

4.1 Approach

Each experiment was ran for 1 minute a total of 10 times. Every frame, the total time measured and timestamps from the GPU for the beginning and ending of both the dispatch and draw calls, were recorded. A python script was used to calculate averages for each test run. Below shows an aggregate of the results for each of the ten tests - where the standard error has been calculated using the square root of the average variance of each sample.

Excluded from the results are two popular metrics from the literature: frame rates and efficiency. Frame times were calculated to print to the console window, to help with debugging. However were not recorded due to their inaccuracy. Frame rate is a measure of the average frames rendered per second. An example of how this is inferior to frame time is if 60 frames rendered in the first half a second, then no frames rendered for the remainder, FPS would still be 60FPS. ”The frame time measurement provides more precise data as it keeps track of the time interval between every displayed frame. [9]”. Parallel efficiency, is the calculation of speed-up per hardware core. Due to a GPU having thousands of cores, efficiency calculations on the GPU are not seen to be as useful [12].

4.2 System Specifications

After gathering initial results on NVIDIA hardware, it was decided to also run similar tests on AMD hardware to validate whether the implementation was actually running asynchronously. GPUOpen, the most useful profiler with regards to Asynchronous Compute would only work with certain types of AMD graphics cards. It was assumed that if certain implementations were seen to be asynchronous, then they would have the ability to be asynchronous on NVIDIA cards. It is important to distinguish, that the performance of each card is not compared to each other as they are completely

separate bits of hardware. However, the AMD results are still useful to see how a different type of hardware handles the same simulation. See Table 4.1 for details of the system used.

Table 4.1: PC Specifications

CPU	Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz
RAM	16GB
OS	Windows 10 Pro V1709 (64-bit)

GPU	
AMD	**NVIDIA**
AMD Radeon (TM) R9 Fury	GeForce GTX 1080

4.3 Experimental Results

4.3.1 NVIDIA

Particle Count	Mean Frame Time (ms)	Standard Error
2	0.243	0.024
4	0.244	0.026
8	0.245	0.042
16	0.246	0.018
32	0.254	0.040
64	0.260	0.037
128	0.292	0.022
256	0.339	0.021
512	0.445	0.049
1024	0.898	0.036
2048	2.536	0.128

Table 4.2: Baseline Compute Tests with 20 Stacks/Slices with a scale of 0.02.

As expected, the higher the particle count, the longer the frame took to complete. Using the results shown in Table 4.2, as a baseline. The experiment was repeated for both asynchronous simulations, and the corresponding speed-up was calculated. The results can be seen in Table 4.3, for the Asynchronous Transfer simulation, and Table 4.4 for the Asynchronous Double Buffer simulation.

Particle Count	Mean Frame Time (ms)	Standard Error	Speed-up
2	0.210	0.020	1.158
4	0.212	0.022	1.155
8	0.211	0.033	1.158
16	0.214	0.020	1.151
32	0.218	0.029	1.163
64	0.228	0.033	1.139
128	0.259	0.015	1.129
256	0.306	0.016	1.108
512	0.412	0.047	1.079
1024	0.949	0.081	0.946
2048	2.755	0.186	0.920

Table 4.4: Asynchronous Double Buffer Simulation with 20 Stacks/Slices with a scale of 0.02.

Particle Count	Mean Frame Time (ms)	Standard Error	speed-up
2	0.229	0.061	1.063
4	0.229	0.062	1.067
8	0.231	0.064	1.059
16	0.235	0.059	1.047
32	0.236	0.072	1.077
64	0.251	0.074	1.034
128	0.260	0.053	1.121
256	0.307	0.060	1.102
512	0.402	0.093	1.108
1024	0.792	0.116	1.133
2048	2.158	0.449	1.175

Table 4.3: Asynchronous Transfer Simulation with 20 Stacks/Slices with a scale of 0.02.

The results for the Asynchronous Transfer simulation show a performance increase, ranging from 1.063x-1.175x speed-up. This is an interesting result, as it shows that the more work completed by the GPU - the more speed-up is achieved. This means that with this technique, the GPU is working more efficiently as the workload increases.

Whereas the results for the Asynchronous Double Buffer simulation, are the complete opposite. The speed-up ranges from 1.158x to 0.920x, which actually shows a decrease in performance for 1024 particles and above. This result is expected, due to the asynchronous tax mentioned in the literature.

Figure 4.1, is a comparison of the average frame time against the number of particles in the simulation, for each simulation type. The results show that up to 512 particles, the fastest frame time is measured from the Double Buffer simulation. After this particle number, the Transfer Buffer simulation becomes the fastest simulation.

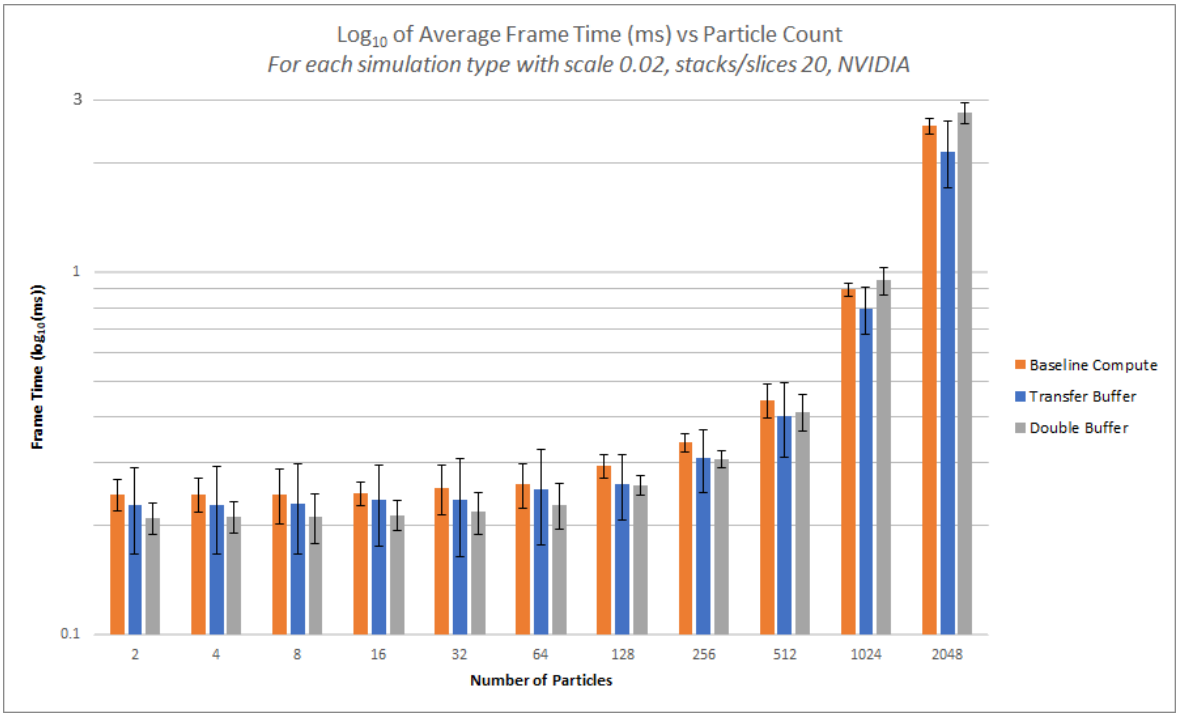

Figure 4.1: **A graph to show results of the NVIDIA experiments** - Detailing the difference in mean frame time (ms) between each simulation type for a range of particle numbers.

4.3.2 Particle Parameters

Two sets of parameters of the particles themselves were changed to see the effects. The stack and slice count effects how detailed the sphere mesh is, that is the number of triangles that constitute the shape. The higher the count, the more work for the GPU, due to more vertices being processed by the graphics pipeline. Altering the scale doesn't change the amount of work directly required for the vertex shaders. However, this was an attempt to increase the graphics rendering time, by forcing the GPU to draw the spheres overlapping each other. This was achieved by reducing the opacity of the final out-put colour of each pixel. This gave an interesting visual effect that can be seen in Figure 4.2.

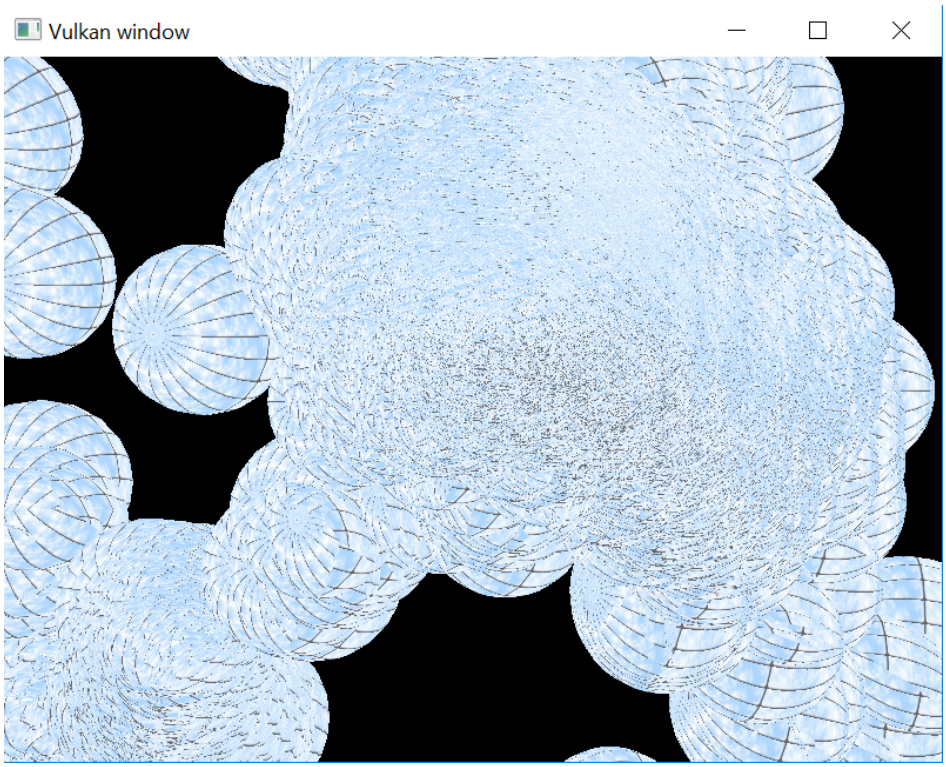

Figure 4.2: **A screen capture of the simulation with scale 2** - Note how the spheres overlap each other.

A table of the results from this experiment can be seen in Appendix Table 3, each simulation was ran with 256 particles on the NVIDIA graphics card. Figure 4.3, compares the results for the baseline compute example. The graph showing the same results for the Double Simulation can be found in Appendix Figure 1. However, the same conclusions can be drawn from both graphs. There is on average a difference of $34\mu s$ between scales 0.2 and 2. Where as, for the difference between scales 0.2 and 2 varies between $-0.4\mu s$ to $3.4\mu s$, this difference is negligible as it only represents very few clock cycles on the GPU.

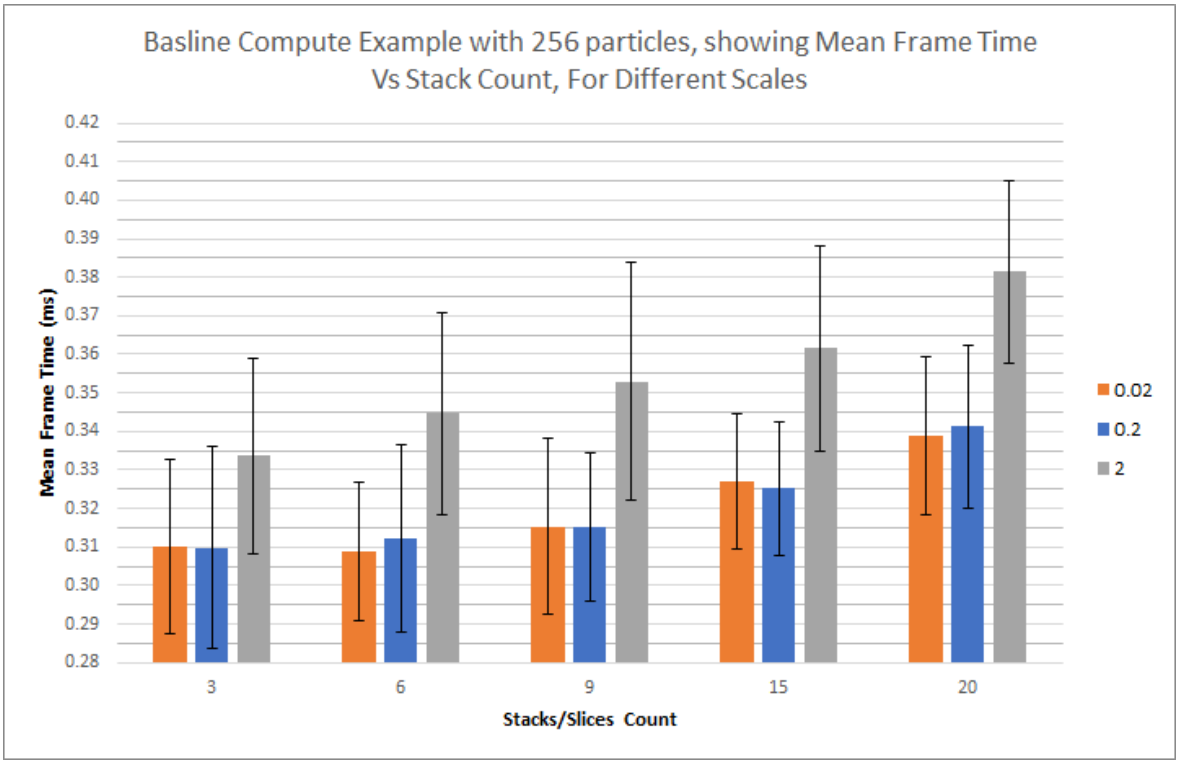

Figure 4.3: **A graph to show effects of stack count and scale on frame time for the baseline simulation** - 256 particles, baseline compute example on a NVIDIA GPU.

Included below in Figure 4.4 is the graph showing the effects of the parameter changes for the Transfer Simulation. This shows a slightly different result, however the trend is still the same. The difference between 0.02 and 0.2 is much less than the difference between 0.2 and 2. This graph also highlights the margin of error of these results being a lot larger than the other two simulations. The largest error from this data set is $\pm 0.07ms$, for 6 stacks/slices at a scale of 0.2.

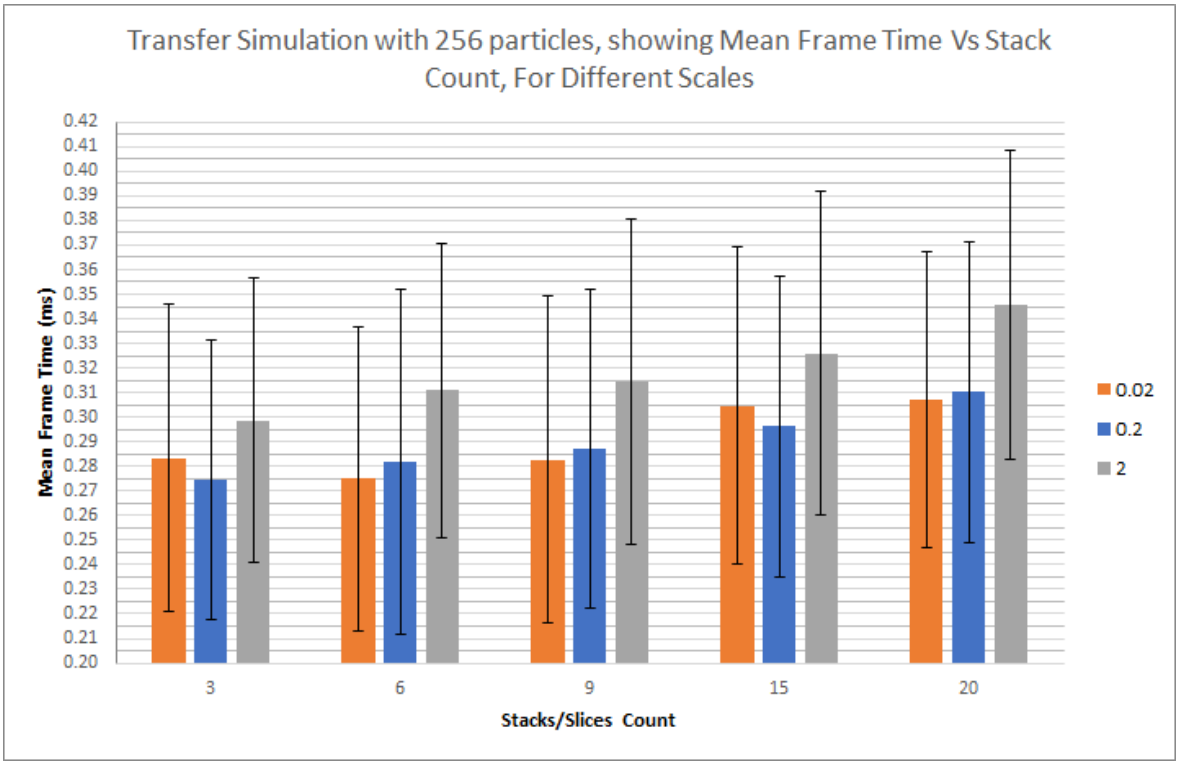

Figure 4.4: **A graph to show effects of stack count and scale on frame time for the transfer simulation** - 256 particles, baseline compute example on a NVIDIA GPU.

4.3.3 Lighting

The final configuration tested was the use of a different vertex and fragment shaders to include lighting calculations. A technique called normal mapping was used. The tables of aggregated results can be seen in Tables 4, 5, 6 of the Appendix.

Figure 4.5, shows the results of each simulation type, using the normal mapping shader. There are no surprises in the data, the trends are very similar to that of the original shading example. However the frame times are longer. An interesting figure to mention is that of the standard errors for the Transfer Simulation for particle count 8 and 128. These results have almost double the error margin of the other tests.

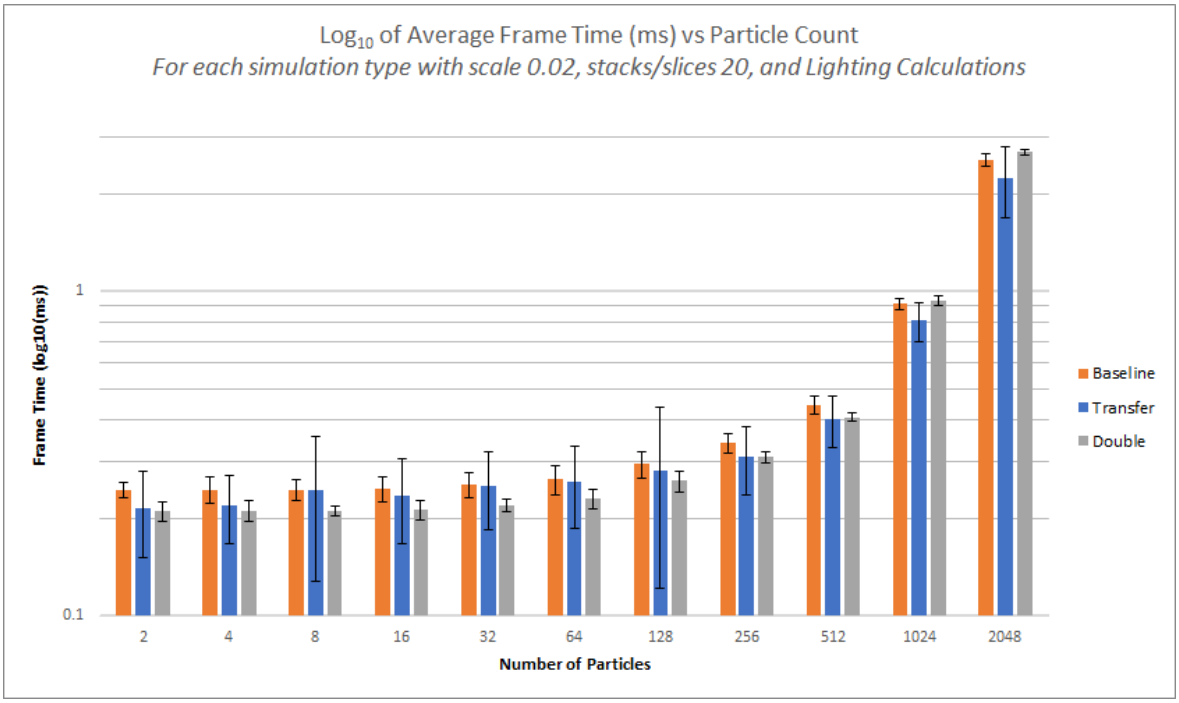

Figure 4.5: **A graph to show the effects of particle increase on frame time for each simulation, with lighting calculations.** - Each experiment was performed using the NVIDIA card, with a scale of 0.02 and stack/slice count of 20.

The differences in speed-up between using lighting and not are highlighted in Table 4.5. Interestingly, the times speed-up for the simulations including lighting was observed to be slightly higher in most cases. For the Double simulation, the same trend can be seen, where the increase of work results in a decrease in speed-up. The trend is also followed for the Transfer simulation, with the exception of the results including 2 and 4 particles.

	Transfer		Double	
Particle Count	**No Lighting**	**Lighting**	**No Lighting**	**Lighting**
2	1.063	1.133	1.158	1.158
4	1.067	1.116	1.155	1.159
8	1.059	1.007	1.158	1.162
16	1.047	1.045	1.151	1.157
32	1.077	1.007	1.163	1.158
64	1.034	1.013	1.139	1.147
128	1.121	1.047	1.129	1.127
256	1.102	1.100	1.108	1.109
512	1.108	1.104	1.079	1.091
1024	1.133	1.126	0.946	0.976
2048	1.175	1.137	0.920	0.947

Table 4.5: Speed-up comparisons between lighting and no lighting.

4.4 Profiling Results

Once the results of the experiments had been studied, the next stage was to explore each simulations through profiling tools. This was an attempt to correlate that the speed-up gained was due to Asynchronous Compute.

4.4.1 GPUOpen

GPUOpen, by AMD, gives the most detailed timing and occupancy information, than the other profilers that were attempted. However, it required very specific drivers and circumstances. There is no control over how long to capture the profile, so occasionally it would not capture a whole frame's worth of information. For example, occasionally the results would fail to record the draw calls, which were definitely happening - seen by the output to the render window.

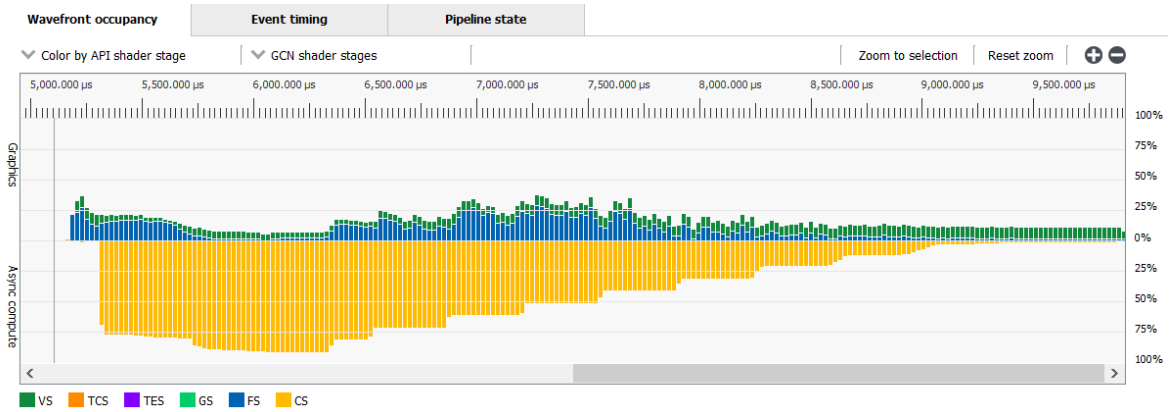

Figure 4.6: **A screen capture of a graph showing the wavefront occupancy of a frame from the double buffer simulation.** - The parameters of the simulation were: 2048 particles, 20 stacks/slices, scale 2.

Figure 4.6, shows part of the output displayed by the profiler, clearly showing asynchronous compute for this example of the simulation; 2048 particles, 20 stacks/slices and 2 scale. What is interesting to note is the dispatch call takes around $150\mu s$ due to a driver barrier launched after resetting the query pool - to time the start of the shader. This can be seen in more detail in Figure 2 of the Appendix. Which highlights the fact that there could be an offset within the timing results.

4.4.2 Visual Studio

Some more basic results could be concluded using the built in Microsoft Visual Studio GPU Profiler [29]. Unfortunately, it only has supported functionality for Microsoft's own DX12 applications. Trying this on the AMD card actually showed some interesting results, however for the NVIDIA results, see Figure 4.7,

Figure 4.7: **A screen capture of a graph showing the queues of the transfer buffer simulation on NVIDIA hardware.** - Both the blue and red sections within the queue are labeled as 'GPU Work', the green is labeled 'CopyResource'.

Rather unhelpfully, due to the implementation being in Vulkan and not DX12, it appears as if only the graphics queue is being used for 'GPU Work'. However, you could argue that it suggests both parts of the shaders are in use, whilst very fast context switches are occuring - note the small red section interleaved with the blue. This cannot unfortunately be used in any reliable way, however, running the same experiment with AMD hardware the results are as follows.

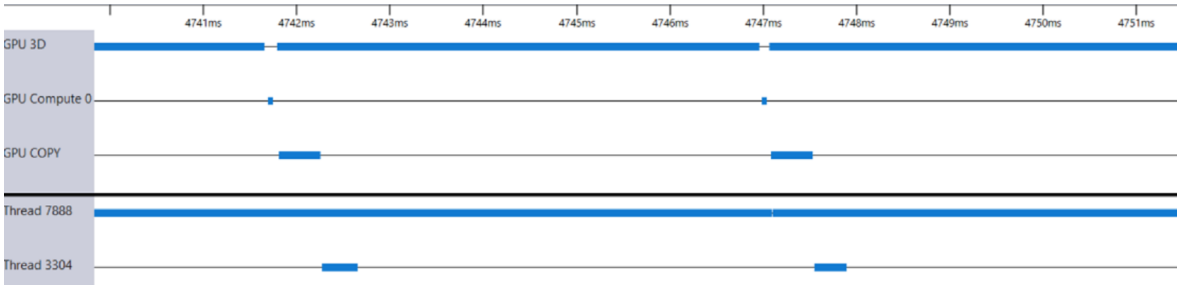

Figure 4.8: **A screen capture of a graph showing the queues of the compute buffer simulation on AMD hardware.** - This clearly shows each queue type and definite separate execution times for Graphics and Compute.

Figure 4.8 clearly shows that the baseline compute example was working as intended. There exists a gap between both compute and the graphics executions, which was the expected result. Likewise, the expected result can also be seen in Figure 4.9. For this simulation, there is a barrier set-up between the transfer queue and the compute queue. This is visible, as the transfer and compute queues never overlap in execution, however the Graphics and Compute do, which means it is Asynchronous Compute.

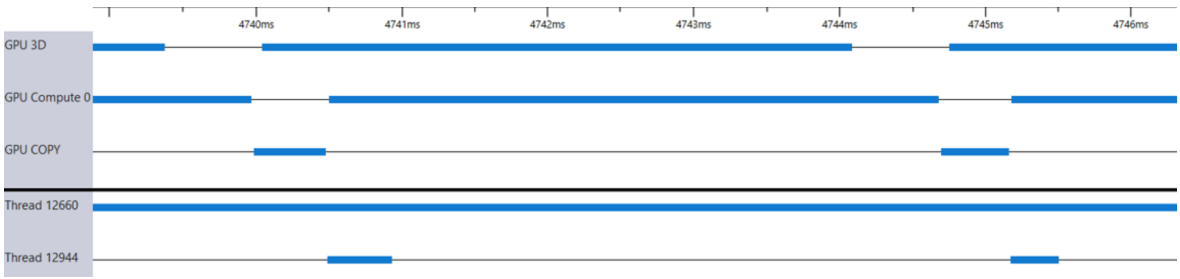

Figure 4.9: **A screen capture of a graph showing the queues of the transfer buffer simulation on AMD hardware.** - This clearly shows each queue type and definite separate execution times for Graphics and Compute.

4.4.3 NSIGHT

NSIGHT is NVIDIA's tool for graphics debugging and profiling. It is designed to integrate with Visual Studio, and boasts "the ultimate development platform for heterogeneous computing." That is of course, only if you are not using Vulkan. Unfortunately there was no conceivable way of capturing any results from the performance profiler, as it requires you to specify which API calls it should capture. Nonetheless, some results could be gained from using the Graphics Debugger. As there was no support for Vulkan, the queues it displayed per frame were merged into one, so there was no way of telling if it was running asynchronously, the transfer queue from the second simulation wasn't pictured at all. But some useful insights were gained into how the separate shaders were executed.

4.5 Asynchronous Comparisons

As the profilers were not conclusive enough to prove that NVIDIA was asynchronous, time stamps were also recorded at the start and end of both the compute and graphics launches. The tables below present the results highlighting the difference in milliseconds, between the two invocations. A positive result for difference, indicates that the two shaders were running concurrently, thus performing Asynchronous Compute.

	Scale	**2**				**0.02**			
	S/S	**3**		**20**		**3**		**20**	
PN	**Sim**	**Diff (ms)**	**SE**	**Diff (ms)**	**SE**	**Diff (ms)**	**SE**	**Diff (ms)**	**SE**
256	**T**	−0.430	0.029	−0.274	0.020	−0.373	0.025	−0.289	0.014
512		−0.543	0.009	−0.187	0.004	−0.434	0.009	−0.170	0.005
1024		−0.565	0.149	0.855	0.005	−0.446	0.011	0.772	0.007
2048		−0.539	0.014	4.554	0.009	−0.380	0.015	4.004	0.016
256	**D**	−0.530	0.004	−0.334	0.004	−0.448	0.005	−0.332	0.004
512		−0.533	0.003	−0.148	0.004	−0.436	0.007	−0.135	0.004
1024		−0.538	0.004	0.892	0.014	−0.406	0.009	0.834	0.006
2048		−0.511	0.004	4.426	0.031	−0.361	0.017	4.054	0.017

Table 4.6: A Table comparing timing differences between shader invocations for varying configurations of the lighting example. Tested on the AMD card.

A comparison of the results in Table 4.6 reveals Asynchronous Compute is taking place under certain conditions for both simulation types on the AMD GPU. When particle size is greater than 1024 and stack/slice count is 20, the results consistently show Asynchronous Compute. These results provide further support for the hypothesis that, below these parameters there is not enough rendering work to force the GPU to schedule the tasks asynchronously. An interesting relationship to draw here is that these two results not only show positively Asynchronous Compute, but

the same configurations were also the only results to return a speed-up of below 1.

	Scale	2				0.02			
	S/S	3		20		3		20	
PN	Sim	Diff (ms)	SE	Diff (ms)	SE	Diff (ms)	SE	Diff (ms)	SE
256	T	−0.033	0.054	−0.029	0.056	−0.033	0.058	−0.147	0.112
512		−0.030	0.073	−0.030	0.064	−0.030	0.053	−0.122	0.335
1024		−0.041	0.088	−0.047	0.084	−0.049	0.090	−0.082	0.109
2048		−0.086	0.164	−0.204	0.132	−0.198	0.150	−0.293	0.225
256	D	−0.021	0.014	−0.022	0.006	−0.023	0.007	−0.026	0.026
512		−0.022	0.014	−0.021	0.006	−0.022	0.007	−0.023	0.034
1024		−0.025	0.026	−0.022	0.007	−0.022	0.010	−0.020	0.015
2048		−0.021	0.026	−0.026	0.106	−0.020	0.009	−0.028	0.075

Table 4.7: NVIDIA LIGHTING

The comparison of the experimental results for the NVIDIA GPU can be found in Table 4.7. These results were not very encouraging, as on average, no asynchronous compute had been recorded. Although, implementing the program in such a way that it could be asynchronous, still gained a significant speed-up. To explore these results further, the differences and timings of the frames were plotted in Figure 4.10

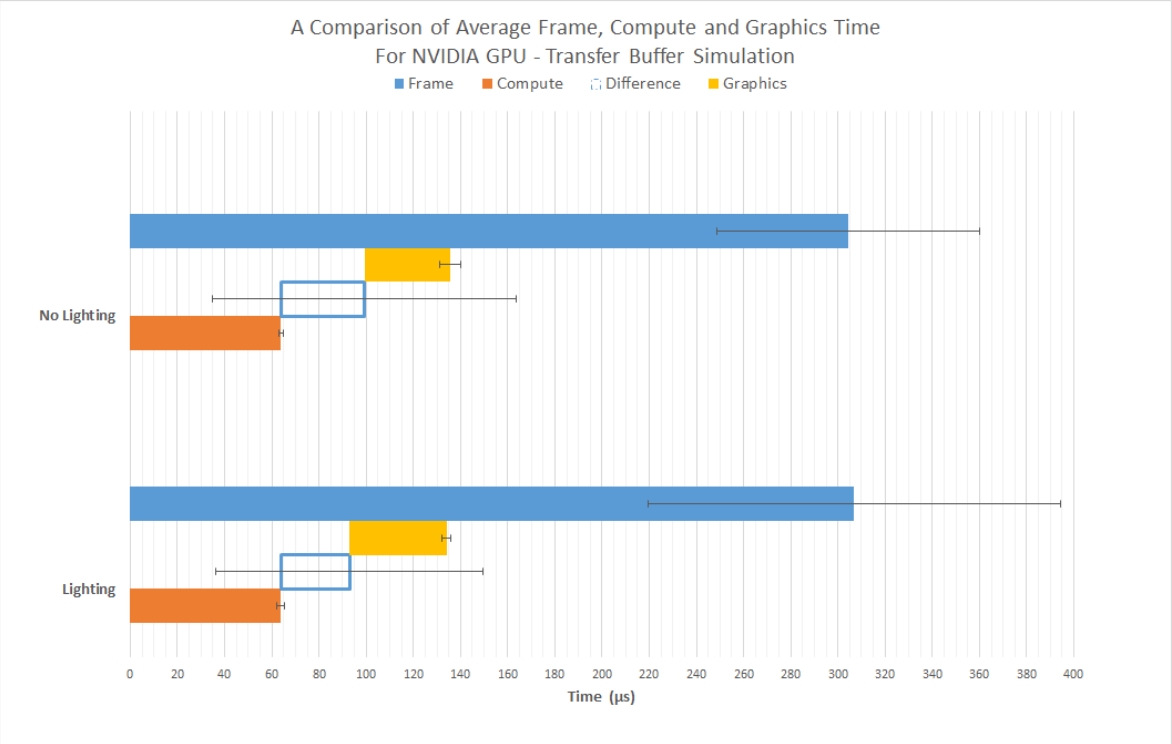

Figure 4.10: **A graph to compare the time between invocations of different shaders for the Transfer Buffer simulation** - these results were gathered using the NVIDIA GPU, for a particle count of 256, stack count of 20 and scale of 0.02.

With the results displayed in this way, it is notable that the time between the graphics

and compute shader invocations varies dependent on the work performed. In both the lit and unlit experiments, the compute shader is seen to take almost exactly the same time. This result is expected as the compute shader used is the same. The lighting calculations occur only in the rendering processes, which also accounts for the longer graphics time in the lighting example. However, the time difference for the lighting simulation was smaller than that of the unlit example. This is interesting, as it also further supports the hypothesis that the likelihood of asynchronous scheduling occurring increases with the workload. By comparing the data in such a way, some frames of the experiments were seen to be Asynchronous Compute. Albeit, these results were not frequent enough to change the overall conclusion, but can be used as evidence that Asynchronous Compute is achievable on a 10-series NVIDIA GPU.

4.6 Changes in implementation

In an effort to quantify the ease of implementation, the simulation classes can be compared. A metric from the literature includes time it took to implement. However, this was not accurately measured due to the complexity of the topic. It would be an unfair comparison, between the time to implement the two asynchronous techniques as they were not completed in parallel. Significantly more knowledge and understanding had been achieved by the second implementation stage that it was quicker to develop.

Detailed below are modified and added methods to the class definitions for the optimisations performed.

Descriptions of functions re-implemented by each class:

- Frame loop
- Allocation of compute command buffers
- Recording of compute commands
- Creation of buffer objects
- Compute dispatch call
- Memory clean-up

As can be seen in Table 4.8, the Double Buffer simulation had the most methods changed or added. This is a naive approach to approximating how much code has been modified as any of these over-ridden functions could be a lot more complex than any another. However, concise descriptions of class functions modified can help to build up a summary of the main Vulkan objects that were changed. It does

Transfer
Override: Creation of command pools
Recording of transfer commands
Compute copy function
Compute Transfer function

Double
Override: Recording of graphics commands
Recording of Render Command (singular)
Recording of Compute Command (singular)
Allocation of Graphics Command Buffers
Creation of Descriptor pool
Creation of Descriptor Sets

Table 4.8: Table describing unique changes and additions to functions for each simulation type.

not tell you whether the change was challenging or significant but nonetheless, the change can be quantified.

Containment of code changes can be easily recorded, due to the final application's class structure. The changes required for the transfer simulation are wholly implemented within its own instance. However, the changes required for the double buffer simulation were not that easily contained. Not only did the technique require multiple extra methods within its own instance, it also required a modified compute shader, and a completely modified render pass, to allow for double buffering of both the graphics and compute queues.

4.7 Summary

In summary, there is evidence to suggest that two Asynchronous Compute techniques were successfully implemented. These techniques tested positive for running concurrently on the AMD GPU through the use of a profiler. Where the results could not be proved conclusively, due to the limitations of the profilers, a performance increase was achieved in all experiments, except for that of the Double Buffer technique above 1024 particles. These exceptions, along with trends revealed between comparisons between lit and unlit scenes, suggests a directly proportional relationship between the likelihood of Asynchronous Compute scheduling and workload.

A difficult metric to measure is that of ease of implementation of the certain optimisations. It is hard to quantify such a task, especially regarding the steep learning curve of Vulkan and low-level graphics programming. However, the Double Buffering technique required the most change in the code-base, and was the least contained.

The results were limited to a maximum of 2048 particles, due to driver issues of the

given GPUs. This makes it difficult for reliable conclusions to be drawn, as there could be changes in the relationships and behaviour of these techniques at higher frame times.

Chapter 5

Evaluation

5.1 Discussion of Results

The first question in this study sought to determine whether Asynchronous Compute techniques could improve performance of a graphical simulation. Three techniques were developed, combined within an application simulating a n-body particle simulation. The results detailed in the previous section show that overall Asynchronous Compute techniques did indeed improve the performance, in certain circumstances.

5.1.1 Initial Render framework and baseline compute

The implementation of the initial rendering framework had much to be desired. The final product fit the purpose of the project, however rendering a more expensive scene would have strengthened the results.

One aspect of Vulkan, the project didn't have time to explore was that of *Push Constants*. Push constants are a different method for inputting values into a shader that can be seen to be more optimised than using multiple descriptor sets for each data buffer. If the project was to be repeated, this is definitely something worth exploring, alongside V-EZ [30]. After spending months of fighting with the low-level verbosity of the Vulkan API. On the 26th of March 2018, GPUOpen released a middle-ware option for a simplification to the Vulkan framework, V-EZ. Unfortunately, this was not available until the end of the project, so could not be used or explored. If this project was completed a year in the future, the focus of the project would have probably been a more complex simulation, as opposed to time spent learning the Vulkan framework. Nonetheless, the experience and knowledge gained from "doing it the hard way" has been invaluable.

What the project doesn't experiment with is testing for multiple changes to shaders, including several different draw calls. It would have been good to test this, and po-

tentially compare the results of this study with using compute for post-processing effects. Only 1 workgroup configuration was tested. Evaluating different configurations of these workgroups to the dispatch calls could have provided interesting results. However, this would have been a substantial project in itself.

5.1.2 Baseline Compute

After setting up the initial rendering engine using Vulkan, the implementation of the basic compute example was straightforward. There are no surprises in the results gathered for this technique, showing that it had been successfully implemented. The implementation was to a high standard, allowing for reliable and repeatable results to be recorded.

5.1.3 Transfer Buffer Technique

The results from the experiments showed that, for this technique, Rthe average frame time was lower than that of the baseline simulation for every configuration. Although, the results from this technique had the largest standard deviations compared to results from the other two techniques. This inconsistency could be related to the implementation itself. Some frames took a lot longer to complete than others, potentially because of the synchronisation barriers between the transfer buffer and the shaders. This theory is supported by comparing the differences in standard deviations between the frame, compute, and graphics timings. In the example of the aggregated data, it can be seen that the standard error for the mean frame time is at least 1 order of magnitude larger than that of the other averages, for this particular technique. As the frame time is the only measure that includes the synchronisations, this could well be the cause of the larger discrepancies within the data.

5.1.4 Double Buffer Technique

The results from the double buffer technique for NVIDIA graphics cards are difficult to draw conclusions regarding Asynchronous Compute. Though, this technique produced the most reliable proof of Asynchronous Compute using the AMD card. It is difficult to compare the two implementations due to this particular simulation being more complex and difficult to develop. Due to the increasing complexity of changes required in the code base to implement this method, only two swapchain buffers were used. Improved performance results may come from future implementation of this method using triple buffering. The cause of decrease in performance over the higher workloads is unclear. This could be attributed to any of the following: ‘Async Tax’ mentioned in the literature, not using triple buffering, or another unexplored cause. It is unlikely that the lack of triple buffering is the cause of this reduction in speed, due to the fact that typically triple buffering is more rendering intensive than double.

5.1.5 Overall Experimental Limitations

Unfortunately, the simulations did not work over 2048 particles, due to driver issues. When the simulation was initialised with a larger number of particles, the program threw a 'device lost' error caused by a crash in the graphics driver. There was not much to rectify in this example other than try back-dating the driver until the simulation was stable. This caused a hard limitation in the simulation of 2048 instanced particles, which could be attributed to the amount of particles that fit within the GPU's memory. More research into memory management techniques using a low-level graphics API would have been beneficial. Although, unfortunately there was not enough time in the scope of the project to include an analysis of shared memory techniques. Vulkan has features such as pipeline caching and shared memory resources within shader operation, however it would have been to complex to evaluate alongside the other aims of the project.

What is not covered by the implementation of the project that could cause a significant effect of the results was that of the compute dispatch call. This is a typical performance study within the parallelism and HPC literature. As a result of GPU architecture, operations to be performed are split into work-groups. The exclusion of an evaluation into this, was actually beneficial to this particular study as the dispatch call used was not optimal. The benefits of this are due to the low cost of the compute shader as a whole.

5.1.6 Cost-Benefit of Optimisation Technique versus Performance

Evaluating the worth of implementing such Asynchronous Compute techniques is difficult due to the nature of the results gathered. The main conclusion that can be drawn from this study with regards to this study is: different implementation of Asynchronous Compute yield different performance results depending on multiple attributing factors. An example of these factors, that the results of this study present, include: hardware architectural differences, and changes to the outputs of the graphics shaders. Both asynchronous techniques showed merits interesting trends on different results, for example; the Transfer simulation showed a potential for running asynchronously on NVIDIA hardware, whilst the double buffer simulation was more reliably asynchronous on AMD. It is for these reasons, combined with having no prior knowledge of Vulkan, that evaluating the cost of these techniques is difficult. As of this, costs of implementation can only be quantified with regards to total changes in the complexity, length, and readability of the code-base. The results of this evaluation conclude that the Double Buffer technique is more difficult to implement. However, a comparison of this cost against the value of the performance differences cannot be made without further research into limiting factors.

5.1.7 Profiling

One thing to be said about the profiling results is that they are not very reliable. As it is a snapshot of 1 frame, it is difficult to judge whether its consistent for every frame, especially with the Transfer Buffer simulation that appears to vary. They are useful to form a proof of concept for whether the implementations were actually running asynchronously. But due to the buggy and unrepeatable nature of these programs, including results of missing values for queues - it is hard to draw solid conclusions especially with the current lack of support for Vulkan and different hardware. Comparisons cannot be made into the utilisation of different drivers and architectures of GPUs which unfortunately appears to be make or break for the performance of these techniques.

A potential future study could include porting this project from Vulkan to DX12 to see the difference in results. However, this is not a light task. DX12 is another lower-level graphics API, exclusive to Microsoft Windows operating systems. It is a different type of challenge in itself compared to Vulkan. But results from the literature show similar results due to Asynchronous Compute.

5.1.8 So was it asynchronous?

One of the aims of this project was to prove that NVIDIA cards with a Pascal architecture could actually utilise Asynchronous Compute. Unfortunately, this result was inconclusive. The results from the experiments showed certain promise for the Transfer technique. Although the average result tended towards the shaders not being executed concurrently, there are some instances of frames where the results were positive. This tended to happen when the time taken for the draw command exceeded 4ms. An implication of this is the possibility that the compute only runs asynchronously when enough work is done by the GPU. Which was one of the hypothesis from the beginning of the project. Unfortunately, due to the scope of the project, exploration of this was not possible.

There was no conceivable way of making time for rendering a more graphical demanding scene, given the time-constraints. This is a real weakness of the project as a whole. The main audience for this research is aimed at games developers, where more realistic values of frame times can reach an upper constraint of approximately 33ms. The project therefore has room for improvement, by evaluating these techniques within a larger scale baseline simulation that reaches those longest acceptable frame times. Nevertheless, if the project did focus on creating these intensive scenes, it would have been more complicated to apply the techniques used. Therefore, it would have had a detrimental effect on the overall success of the project, as there would not be enough time to focus on gaining such an in-depth knowledge of the low-level operations of the GPU. Consequently the results of this project could be

used as preliminary results, or a proof of concept, for testing these more expensive simulations.

5.2 Evaluation in respect to the learning objectives

5.2.1 LO1: Manage a substantial individual project

The initial planning of the project can be seen throughout both the Initial Project Overview and the Week 9 documentation within the appendix. Substantial project milestones were documented from the outset of the project. These were all successfully completed. However the suggested timeframe of the optimisation and result gathering stages were underestimated. Albeit, the experiments could have been planned in more detail as using an automated scripting process gathered so many results that quickly got out of hand. Some of the results gathered have not been included in the evaluation of this project as they were not significant enough differences to comment upon. More planning at this stage would have been beneficial to allow for more time to analyse the results. For example, configurations including different display resolutions could have proved interesting results, as opposed to recording the changing stack/slice size for every particle count that ended up not being used.

Every week, meetings with the project supervisor were attended and the project overall was kept mostly on-track. For the first half of these meetings, the notes were written in a diary that is available on request. These meetings were gradually less important regarding the direction of the project as time moved on, as the understanding of Vulkan improved.

5.2.2 LO2: Construct a focused problem statement and conduct suitable investigations

Before undertaking the project, it was known that the literature surrounding the problem area was sparse. However, this was not a deterrent due to the potential significance of the results. The project was sufficiently demanding, as the knowledge of a brand-new graphics API was required before the main aim of the project could be completed. The main aim of the project was to analyse the effects of leveraging command queues

Due to the age of the problem area, the surrounding literature was difficult to find. Most of the initial research into the topic was formed from news articles and blog posts as opposed to published journal papers. This research, of course, was not a reliable source of information, however it did help to inform the direction of the project.

5.2.3 LO3: Demonstrating professional competence

The aim of the project was to design, implement and evaluate an example of Asynchronous Compute. The final deliverable of the project consisted of a complex piece of software designed for research purposes. One part of the evaluation of the project was a subjective evaluation of the ease of the implementation of the asynchronous compute. Due to the software design of the final application, this evaluation was easier to achieve. This was because the application was split into a renderer class and concrete implementations of an abstract simulation class. The renderer class dealt with the main Vulkan functionality that is required by any typical rendering engine. Whilst the instances of the simulations contain methods that were overridden to provide the specific functionality. This meant that overall, the differences in implementations were exposed clearly for the evaluation.

Realistic technical goals were set within the planning stages of the project. As the problem area was so broad, the scope of the project had to be based on a reasonable estimation of personal ability. Prior knowledge of computer graphics, real-time rendering, and of experimental methods in concurrent and parallel systems, was definitely a pre-requisite for undertaking this project. In saying that, the challenge of tackling a new low-level graphics API was not one to be underestimated. There are many directions the project could have explored, for example; expanding on the simulation by including rigid-body physics (collision detection). However, it was decided that the project was demanding enough as it was, and the focus of the project was geared towards gaining an in-depth understanding of leveraging queues on the GPU. The extent to which the application provided a means of evaluating this goal is high.

Technical problems arose frequently during the course of this project, such is the nature of GPU programming. Whether it was an update to a graphics driver, or an unsupported profiler, each problem was unfortunately non-trivial. Anticipating for such problems in advance was a large part of the planning process, and where possible time to fix these problems were over-estimated. This allowed for still achieving the aims of the project on time. The stage taken from the planning process which allocated time for finding solutions to these difficult problems was part of the optimisation milestone. It was at this stage where the majority of bug fixes, and code re-factoring occurred. Any risks to the aims of the project outcomes were successfully mitigated by allowing time for this - which in turn improved the overall quality of the project.

Chapter 6

Conclusion

In this investigation, the aim was to develop a 3D rendered application to evaluate the effect of Asynchronous Compute on rendering performance. In completing this aim, two different approaches to Asynchronous Compute were explored: the Transfer Buffer technique, and the Double Buffer technique.

Both of these techniques were seen to be asynchronous, under certain conditions. The investigation of these conditions shows a correlation between workload and likelihood of asynchronism. The more work done, the more the shader executions overlap. For smaller workloads, the overhead cost associated with Asynchronous Compute outweighs the increases to performance it brings.

6.1 Reflection

Overall, the project was a success. The results show an increase in performance using certain asynchronous compute techniques.

For example, changing the number of triangles per sphere did not effect the time difference between invocations of the shader. This implies that NVIDIA have some sort of hardware scheduler that allows a fixed-time between each job before task-switching. Interestingly, a decrease in this difference measurement was recorded for the lighting example, suggesting that the timing has something more to do with the fragment stage of the graphics pipeline.

The experiments confirmed that there is a distinct difference between vendors in how the propriety drivers and hardware handle asynchronous commands. It is interesting to note that on the tested NVIDIA GPU, the more work, the higher the times speed-up of the transfer simulation, which suggests the GPU working more efficiently. This is potentially due to ability of asynchronous compute. Even though the GPU on average, didn't perform the tasks asynchronously, when the graphics time for a frame reached over $4ms$ an overlap of the shaders was recorded. This is significant, as

it could be proof that NVIDIA finally has support for true concurrency across queue families.

Although promising results, the study's biggest limitation is that the application was CPU bound. Unfortunately, the developed scene was not sufficiently detailed, and therefore not expensive to begin with. The conclusions of this study would have been strengthened by a more realistic benchmark simulation.

6.2 Future Work

A simulation that would require around $30ms$ to processes one frame, would have been a more realistic starting point for the evaluation. This would make a good basis for taking this study further. Tying directly into a future aim of exploring the effect of multiple dispatch calls on asynchronous performance.

A significant future development of this study would be to attempt to quantify the 'Async Tax', aiming to develop a new method of combining multiple current optimisation techniques. Defining where best to apply these techniques for maximum utilisation of the hardware. Additionally, identifying the costs of these techniques, and where possible ways to mitigate these costs.

Bibliography

[1] *Proc. Game Developers Conference (GDC, San Francisco, February-March 2017)*, Deep Dive: Asynchronous Compute, San Francisco, March 2017. Springer.

[2] T. Akenine-Möller, E. Haines, and N. Hoffman. *Real-Time Rendering, Third Edition*. CRC Press, 2016.

[3] Ben Ashbaugh, Adam Lake, and Maria Rovatsou. Khronos™ group. In *Proceedings of the 3rd International Workshop on OpenCL*, IWOCL '15, pages 16:1–16:23, New York, NY, USA, 2015. ACM.

[4] Sean T. Barrett et al. Stb libraries, 2017.

[5] Victor R Basili. Quantitative evaluation of software methodology. Technical report, 1985.

[6] Albin Bernhardsson. Asynchronous compute example, 2017.

[7] W. Blewitt, G. Ushaw, and G. Morgan. Applicability of gpgpu computing to real-time ai solutions in games. *IEEE Transactions on Computational Intelligence and AI in Games*, 5(3):265–275, Sept 2013.

[8] Chih-Fan Chen, Mark Bolas, and Evan Suma Rosenberg. Rapid creation of photorealistic virtual reality content with consumer depth cameras. In *Virtual Reality (VR), 2017 IEEE*, pages 473–474. IEEE, 2017.

[9] Michael J (Intel) Coppock. An often overlooked game performance metric: Frame time. *Intel Software: Developers Zone*, August 2015.

[10] NVIDIA Corporation. Whitepaper: Nvidia geforce gtx 1080. pages 14–17, 2016.

[11] cppreference. Date and time utilities. *CPP Reference*, April 2018.

[12] J B B Darmawan and S Mungkasi. Performance of parallel computation using cuda for solving the one-dimensional elasticity equations. *Journal of Physics: Conference Series*, 801(1):012080, 2017.

[13] Jackson Dunstan. Procedurally-generated sphere. *Mastering CSharp and Unity3D*, July 2012.

[14] Ian Finlayson. Gba graphics programming. *University of Mary Washington*, 2017.

[15] G-Truc Creation. Opengl mathematics, 2017.

[16] Mahder Gebremedhin, Afshin Hemmati Moghadam, Peter Fritzson, and Kristian Stavåker. A data-parallel algorithmic modelica extension for efficient execution on multi-core platforms. In *9th International Modelica Conference*, Sept 2012.

[17] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[18] Advanced Micro Devices Inc. Whitepaper: Amd graphics cores next (gcn) architecture. pages 12–15, 2012.

[19] LunarG Inc. Create a swapchain. *Vulkan Docs*, 2016.

[20] M. Joselli and E. Clua. Gpuwars: Design and implementation of a gpgpu game. In *2009 VIII Brazilian Symposium on Games and Digital Entertainment*, pages 132–140, Oct 2009.

[21] Mark Joselli, José Ricardo da Silva, Marcelo Zamith, Esteban Clua, Mateus Pelegrino, and Evandro Mendonça. Techniques for designing gpgpu games. In *Games Innovation Conference (IGIC), 2012 IEEE International*, pages 1–5. IEEE, 2012.

[22] David Kanter. Graphics processing requirements for enabling immersive vr. *AMD White Paper*, 2015.

[23] Jon Kerridge. *Using Concurrency and Parallelism Effectively - II*. Bookboon.com, 2015.

[24] Mark Kilgard. 11.1 glutsolidsphere, glutwiresphere. *OpenGL Documentation*, 1996.

[25] Paul Lilly. Intel joins forces with amd to battle nvidia. *PCGAMER*, November 2017.

[26] LunarG. Descriptor set layouts and pipeline layouts. *Vulkan Documentation Tutorials*, 2016.

[27] LunarG. 16.1. query pools. *Vulkan Documentation*, 2017.

[28] Timothy G Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming*. Pearson Education, 2004.

[29] Microsoft. Gpu usage. *Microsoft Visual Studio Docs*, 2017.

[30] Sean O'Connell. V-ez brings "easy mode" to vulkan. *GPUOpen*, March 2018.

[31] Alexander Overvoorde et al. Graphics pipeline basics. *Vulkan Tutorial*, 2017.

[32] Alexander Overvoorde et al. Pipeline layout: Rasterizer. *Vulkan Tutorial*, 2017.

[33] Alexander Overvoorde et al. Vertex buffer creation. *Vulkan Tutorial*, 2017.

[34] Alexander Overvoorde et al. *Vulkan-Tutorial*. May 2017.

[35] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[36] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113.

[37] Alessio Palumbo. Async compute praised by several devs; was key to hitting performance target in doom on consoles. *WCCFTECH*, June 2016.

[38] Taylor C. Richberger and Pavel Belikov. A simple header-only c++ argument parser library, 2016.

[39] Will Schroeder, Kenneth M. Martin, and William E. Lorensen. *The Visualization Toolkit (2Nd Ed.): An Object-oriented Approach to 3D Graphics*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[40] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. A performance analysis framework for identifying potential benefits in gpgpu applications. *SIGPLAN Not.*, 47(8):11–22, February 2012.

[41] Ryan Smith. The nvidia geforce gtx 1080 & gtx 1070 founders editions review: Kicking off the finfet generation. 2016.

[42] Natalya Tatarchuk. Open problems in real-time rendering. In *ACM SIGGRAPH 2016 Courses*, SIGGRAPH '16, New York, NY, USA, 2016. ACM.

[43] The GLFW Development Team. Glfw, 2016.

[44] The Khronos™ Group. Face culling. *OpenGL Wiki*.

[45] The Khronos™ Group. Vertex rendering - instancing, 2017.

[46] The Khronos™ Group. Vulkan api description, 2017.

[47] The Khronos™ Group. Khronos group releases vulkan 1.1. *Khronos News*, march 2018.

[48] The Khronos™ Group. vkcmdwritetimestamp(3) manual page. *Vulkan Specifications*, February 2018.

[49] The Khronos™ Group. Vkpipelinestageflagbits(3) manual page. *Vulkan Specifications*, February 2018.

[50] Sascha Willems, Bradley Austin Davis, et al. Asynchronous compute example, 2017.

Appendices

A Relevant Materials

Listing 1: ”Vertex.vert”

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(location = 0) in vec3 inPos;
layout(location = 1) in vec3 inColour;
layout(location = 2) in vec2 inTexCoord;

layout(location = 3) in vec4 instancePos;

layout(location = 0) out vec3 fragColour;
layout(location = 1) out vec2 fragTexCoord;

layout(binding = 0) uniform UniformBufferObject
{
    mat4 model;
    mat4 view;
    mat4 proj;
} ubo;

out gl_PerVertex
{
    vec4 gl_Position;
};

void main()
{
  vec3 v = instancePos.xyz;
  mat4 m = mat4(1.0);
  // transformation matrix
  m[3] = m[0] * v[0] + m[1] * v[1] + m[2] * v[2] + m[3];

  // scale matrix
  mat4 s = mat4(1.0);
  s[0] = s[0] * instancePos.w;
  s[1] = s[1] * instancePos.w;
  s[2] = s[2] * instancePos.w;

  mat4 trs = m * s;
  mat4 model = trs * ubo.model;
    gl_Position = (ubo.proj * ubo.view * model) * vec4(inPos, 1.0);
    fragColour = instancePos.xyz;
  fragTexCoord = inTexCoord;
}
```

Listing 2: ”Compute.comp”

```
void main()
{
    // Current SSBO index
    uint index = gl_GlobalInvocationID.x;
  // Don't try to write beyond particle count
    if (index >= ubo.particleCount)
    return;

    // Read position and velocity
    vec3 vVel = particles[index].vel.xyz;
    vec3 vPos = particles[index].pos.xyz;

  // calculate acceleration
  vec3 acceleration = vec3(0.0);

  // for each particle calculate force
  for (int i = 0; i < ubo.particleCount; i++)
  {
    if (index == i)
      continue;

    vec3 dist = particles[i].pos.xyz - particles[index].pos.xyz;
    vec3 direction = normalize(dist);

    // calculate acceleration with dampening constants
    acceleration += direction * GRAVITY / pow(dot(dist, dist) + SOFTEN, POWER);
  }

  float deltaT = max(0, ubo.deltaT);
  vVel += (deltaT * acceleration);

    vPos += vVel * deltaT;

    // Write back
    particles[index].pos.xyz = vPos;
    particles[index].vel.xyz = vVel;
}
```

Simulation Type	TRANSFER BUFFERS _ ASYNC							
Particles	256	Stack Count	20	Slice Count	20	Mesh Scale	0.02	
Frame	Frame Time (ms)	Compute TS Start	Compute TS	Compute Time	Graphics TS Start	Graphics TS End	Graphics Time	async?
1	0.5374	1.52E+18	1.52E+18	0.065536	1.52E+18	1.52E+18	0.038912	NO
2	0.108863	1.52E+18	1.52E+18	0.065536	1.52E+18	1.52E+18	0.038912	NO
3	0.241548	1.52E+18	1.52E+18	0.063488	1.52E+18	1.52E+18	0.036864	NO
4	0.104765	1.52E+18	1.52E+18	0.063488	1.52E+18	1.52E+18	0.03584	NO
5	0.089908	1.52E+18	1.52E+18	0.063488	1.52E+18	1.52E+18	0.03584	NO
6	0.141137	1.52E+18	1.52E+18	0.063488	1.52E+18	1.52E+18	0.03584	NO
7	0.263321	1.52E+18	1.52E+18	0.064512	1.52E+18	1.52E+18	0.036864	NO

Table 1: An example of the output of the application test.

Sim Type:		**NORMAL COMPUTE**							
Particles	**64**	**Stacks/Slices**		**20**		**Scale**	**0.02**		
Total Frames	**Mean FT (ms)**	**STDev**	**Variance**	**Mean CT (ms)**	**STDev**	**Variance**	**Mean GT (ms)**	**Stdev**	**Variance**
72 056	0.264	0.056	0.003	0.019	0.001	3.530×10^{-7}	0.014	0.001	1.03E-06
72 341	0.263	0.013	0.000	0.019	0.001	3.780×10^{-7}	0.014	0.001	1.24E-06
71 817	0.265	0.019	0.000	0.019	0.001	3.440×10^{-7}	0.014	0.001	1.03E-06
72 347	0.263	0.013	0.000	0.019	0.001	3.490×10^{-7}	0.014	0.001	1.21E-06
72 277	0.263	0.012	0.000	0.019	0.001	3.070×10^{-7}	0.014	0.001	1.01E-06
72 318	0.263	0.013	0.000	0.019	0.001	3.510×10^{-7}	0.014	0.001	1.22E-06
72 374	0.263	0.012	0.000	0.019	0.001	3.130×10^{-7}	0.014	0.001	9.98E-07
72 219	0.263	0.022	0.000	0.019	0.001	3.610×10^{-7}	0.014	0.001	1.25E-06
72 425	0.262	0.014	0.000	0.019	0.001	3.360×10^{-7}	0.014	0.001	1.02E-06
72 014	0.264	0.046	0.002	0.019	0.001	3.500×10^{-7}	0.014	0.001	1.04E-06
Averages	0.263	0.027	0.001	0.019	0.001	3.440×10^{-7}	0.014	0.001	1.11E-06

Table 2: An example of aggregated results from the python script.

Sim	S/S	Scale 0.02 Mean FT (ms)	std dev	Speedup	Scale 0.2 Mean FT (ms)	std dev	Speedup	Scale 2 Mean FT (ms)	std dev	Speedup
	3	0.310254	0.022617		0.309843	0.026147		0.333601	0.025473	
	6	0.308699	0.01791		0.312196	0.024193		0.344572	0.026162	
0	**9**	0.315354	0.022678		0.315109	0.019273		0.352888	0.030855	
	15	0.327038	0.017551		0.325169	0.017263		0.361514	0.026669	
	20	0.338787	0.020604		0.34119	0.021096		0.381513	0.023641	
	3	0.283258	0.062476	1.095302	0.274549	0.056615	1.128555	0.29867	0.057756	1.116956
	6	0.275072	0.061945	1.122244	0.281861	0.070175	1.107622	0.310762	0.059541	1.108797
1	**9**	0.28265	0.066471	1.115706	0.287252	0.064737	1.096978	0.314538	0.06629	1.121926
	15	0.304537	0.0645	1.073885	0.296361	0.061192	1.097204	0.325907	0.065927	1.109257
	20	0.307326	0.060143	1.10237	0.310199	0.061359	1.099906	0.345675	0.062949	1.103676
	3	0.273411	0.010742	1.134752	0.274817	0.014141	1.127452	0.299788	0.013372	1.112788
	6	0.275878	0.010641	1.118968	0.278866	0.014346	1.119519	0.31538	0.016776	1.092563
2	**9**	0.280438	0.014958	1.124506	0.280854	0.011385	1.121967	0.323271	0.030855	1.091617
	15	0.291993	0.012203	1.120018	0.292475	0.013556	1.111781	0.334332	0.021846	1.081303
	20	0.30577	0.015733	1.107979	0.307071	0.01258	1.11111	0.35402	0.025667	1.077659

Table 3: 256 Particle count, varying slice/stack count and scale.

Particle Count	Mean Frame Time (ms)	Standard Error
2	0.243	0.013
4	0.244	0.023
8	0.244	0.018
16	0.245	0.022
32	0.253	0.022
64	0.263	0.027
128	0.293	0.027
256	0.341	0.024
512	0.446	0.029
1024	0.914	0.033
2048	2.548	0.108

Table 4: NVIDIA Lighting Particle Increase Baseline with 20 Stacks/Slices with a scale of 0.02.

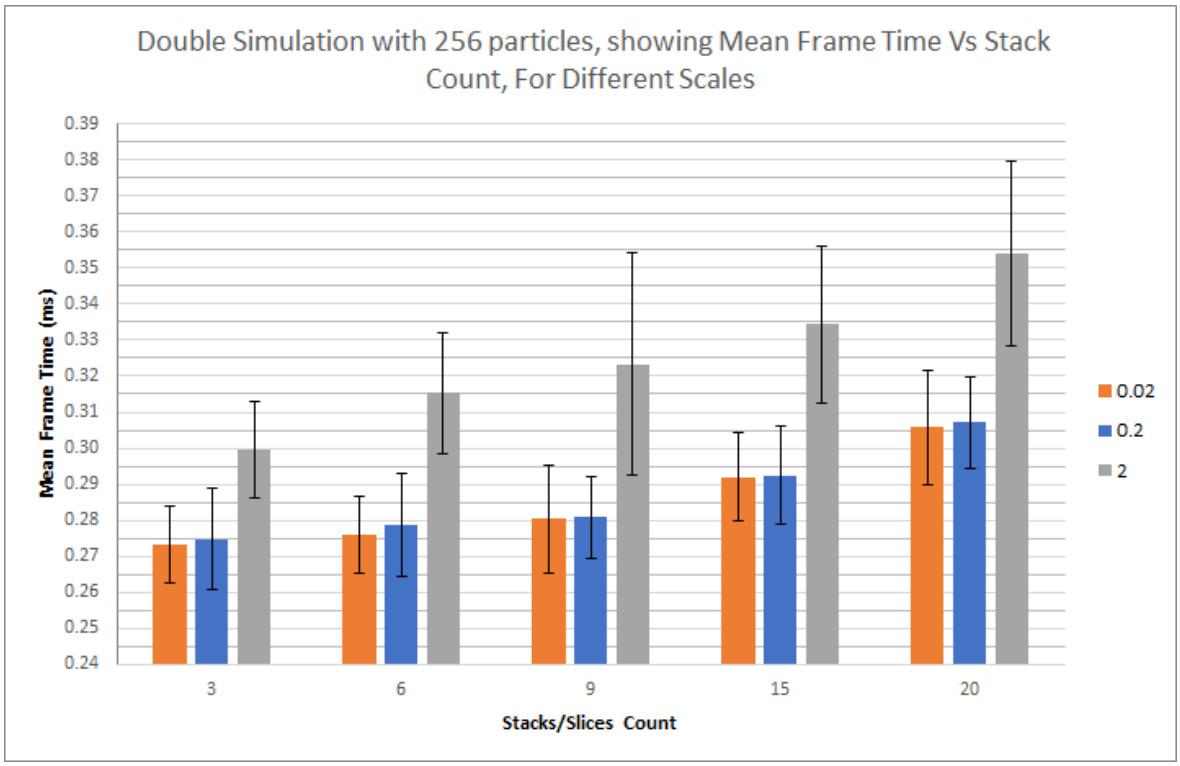

Figure 1: **A graph to show the effects of stack count and scale on frame time for the double simulation** - Each run was using 256 particles, using the NVIDIA card.

Particle Count	Mean Frame Time (ms)	Standard Error	Speedup
2	0.214	0.063	1.133
4	0.219	0.052	1.116
8	0.243	0.115	1.007
16	0.235	0.068	1.045
32	0.251	0.068	1.007
64	0.260	0.074	1.013
128	0.283	0.256	1.036
256	0.310	0.074	1.100
512	0.404	0.074	1.104
1024	0.811	0.110	1.126
2048	2.241	0.559	1.137

Table 5: NVIDIA Lighting Particle Increase Asynchronous Transfer with 20 Stacks/Slices with a scale of 0.02.

Particle Count	Mean Frame Time (ms)	Standard Error	Speedup
2	0.210	0.015	1.158
4	0.211	0.016	1.159
8	0.210	0.008	1.162
16	0.212	0.014	1.157
32	0.219	0.010	1.158
64	0.229	0.015	1.147
128	0.260	0.019	1.127
256	0.308	0.011	1.109
512	0.409	0.013	1.091
1024	0.936	0.037	0.976
2048	2.690	0.050	0.947

Table 6: NVIDIA Lighting Particle Increase Asynchronous Double with 20 Stacks/Slices with a scale of 0.02.

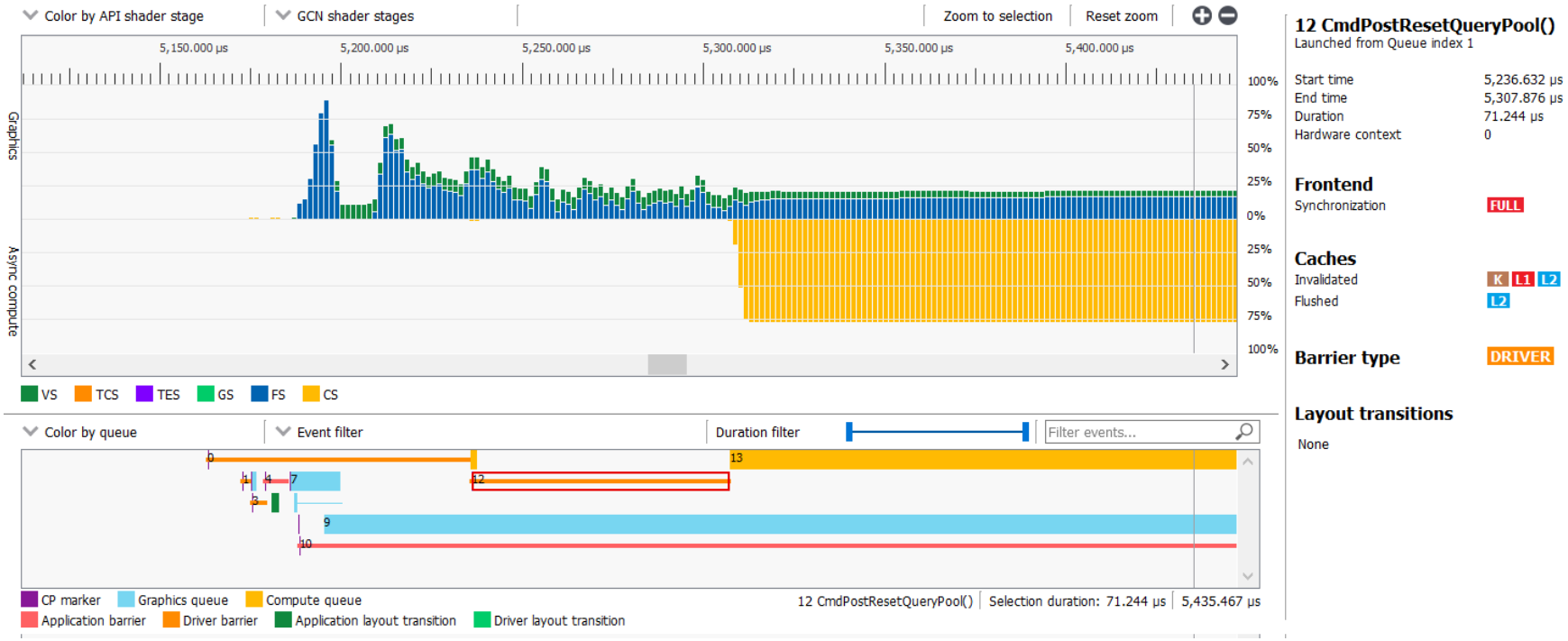

Figure 2: **A screen capture of a graph showing the wavefront occupancy of a frame from the double buffer simulation.** - The parameters of the simulation were: 2048 particles, 20 stacks/slices, scale 2.

B Initial Project Overview

.1 Overview of Project Content and Milestones

The project entitled ”High Performance Graphics” aims to analyse the effect of asynchronous compute shaders on game performance.

Asynchronous compute increases the performance of games in certain conditions. This project aims to understand these conditions by exploring optimisations.

By evaluating the trade offs between performance increase and ease of implementation of the optimisations needed to improve the capability of asynchronous compute shaders.

In the area of computer graphics, shaders are programs run on the GPU. A compute shader is different than other shader stages as it usually doesn't work on tasks directly related to rendering - they are more used for auxiliary calculations. Asynchronous compute is a technique that allows GPUs to perform graphics and compute work simultaneously. In the past, this has not been achievable on NVIDIA GPUs due to hardware limitations, however with the recent architectural changes, true async compute is now possible.

Milestones:

- Rendering engine using Vulkan
- Rendering something complex using compute shaders
- Render same scene using asynchronous compute shaders
- Analyse performance differences
- Optimise code for async
- Re-evaluate performance using these optimisations

.2 The Main Deliverables

The main deliverables include:

- A real-time rendered scene using asynchronous compute
- Performance analysis results
- A report evaluating optimisation techniques used
- A poster

.3 Target Audience for the Deliverables

An increase in performance for rendering will be beneficial not only for other researchers in the area of computer graphics, but for game studios, independent or other. A common problem with games that include async compute is that it doesn't necessarily always increase performance. This project will benefit game developers to understand why, or how to optimise their software for async compute shaders. Which in turn will aid developers who are porting their releases to console, as any bit of performance increase is sought after.

.4 The Work to be Undertaken

The first main task to be undertaken is to build a rendering framework in C++ using Vulkan and GLFW. This rendering framework should allow a complex scene to be rendered using compute shaders. Once this has been completed, performance testing should be undertaken to get a baseline for the experiment.

Afte a baseline has been recorded, asynchronous compute should be implemented and tested again. Testing on the same hardware under the same conditions is important for this as different hardware can effect this. Once analysing the performance difference, an attempt should be made in how to optimise the code to increase the performance of async. Then an evaluation of which of these techniques can be utilised with less changing of the particular engine design against performance.

.5 Additional Information / Knowledge Required

Thorough knowledge of compute shaders and the graphics pipeline will be required for this project, extending current knowledge of computer graphics and GPU programming. Learning a new rendering API will also be required. For the evaulation side of the project, research will be needed into how to analyse performance of software, specifically for real-time simulations. It would also be useful to learn more about job scheduling and optimisations.

.6 Information Sources that Provide a Context for the Project

Games such as Doom, which had reports of rendering time gains of 3 to 5ms, meaning it is invaluable for console optimisation. Information about how hardware manufacturers designed GPU hardware that is more compatible with true asynchronous compute, including the Vulkan API documentation. Published guidelines on best practises for using the systems and the new architecture. Articles and reviews from games companies who are trying to implement async within their releases, for example idSoftware, Oxide Games and Q-Games.

.7 The Importance of the Project

The project is significant as the technology is quite new for the games industry, for specifically Nvidia GPUs. AMD have had the hardware edge due to having specific Asynchronous Command Engines, whereas only the new Pascal architecture Nvidia cards have the capability. Before, the older architecture could not context switch as quickly as AMD. The reason it was chosen to do the project based on software improvements as opposed to hardware evaluations as lots of companies have tested the asynchronous compute latency across different hardware, but not performance.

So testing the overall throughput on a high-end Pascal card and researching how job scheduling and code optimisations may be able to help improve the performance, of async, is significant to the industry. For a Maxwell card, the drivers for Async Compute were not completely implemented so the question became how you would attempt to schedule simultaneous jobs, and whether they could run graphics and compute workloads concurrently. Now, Pascal cards have fixed this resource allocation issue, as they have implemented a dynamic load balancing system, which means better so async and true concurrency should be possible.

To further the project, it can be developed to then test on different hardware etc. For example, the increase seems extremely important to making console ports for games run smoother, as consoles are typically limited by the hardware async compute can improve the quality of these games.

.8 The Key Challenge(s) to be Overcome

Researching a relatively new subject, at least for Nvidia cards. Will have to devote time to understanding graphics cards architecture a lot more in depth to see how to utilise it to improve performance.

If the performance does not increase, there also needs to be more understanding of why. What are the bottlenecks? Are there any situations where this is not suitable? To ensure success, more parallel programming skills are needed for the evaluation and optimisation stages of the project. Another key challenge to covercome will be the implementation of compute shaders.

A Week 9 Interim Report

.9 Introduction

The main aim of the project entitled 'High Performance Graphics' is to develop and analyse the effect on rendering performance using asynchronous compute techniques.

In completing this project, an evaluation into the difficulties of implementation of certain asynchronous compute shaders will be recorded. This will be achieved by comparing the performance changes between using compute shaders and asynchronous compute.

This can be split into multiple objectives including; rendering a complex scene using compute and async compute; analysing performance differences; implementing optimisations for the code and re-evaulating the performance.

To achieve these aims, first a simulation using compute shaders must be implemented. This is to record a baseline for the investigation. Measuring performance by timing how long it takes for a frame to be rendered.

For the purpose of this project, a resource intensive scene must be rendered in real-time; for example a scene consisting of many vertices. It is not sufficient to be a scene that is only intensive to render, it also needs an element of general purpose calculations on the GPU (GPGPU). Within games technology, compute shaders can be used for such tasks as physics calculations or post-processing effects. The idea is to use an n-body particle simulation. In this instance, the compute shader will operate using two buffers: velocity and position of each particle. Within the compute shader, the velocities are used to perform calculations to approximate the new positions of the particles which will be then passed to the render pipeline. Doing this asynchronously means that this computation step will run in parallel to the rendering of the previous frame.

A Background

A compute shader is a program that runs on the GPU, in this case to accelerate rendering performance for a real-time game simulation. Asynchronous compute aims to run these programs truly concurrently alongside the rendering stages. For this example, the chosen simulation's performance will be investigated by modifying the number of particles rendered and by modifying the resolution of the spherical meshes that the particles are represented by, essentially number of triangles processed and rendered per frame.

An N-body simulation is a dynamic system of particles, approximating the movement

of the particles that interact with each other under a physical force, for example gravitational force.

$$\sum_{i \neq n}^{N} \frac{G m_n m_i}{{r_{ni}}^2} \tag{1}$$

The force acting on a body by another, is inversely proportional to the distance squared between the two bodies with mass. In the case of an N-body simulation the brute force algorithm has $\mathcal{O}(N^2)$ complexity, which means it is not feasible to use for a large number of particles. Due to the complexity of this algorithm, any increase in performance should be easily noticeable. Therefore the plan is to first implement a method which checks every particle for it's position relative to the current particle, and then to update the next frame whilst the GPU is rendering the initial frame. To take the project a step further, this can be updated to be a rigid-body simulation, including collision resolution of each particle.

B Asynchronous Compute

Gaps between rendering tasks leave part of the GPU idling. Async compute aims to utilise these gaps by scheduling compute tasks concurrently. In the past, async compute has been extensively used on AMD graphics cards with Graphics Core Next (GCN) architecture. These AMD GPUs have Asynchronous Compute Engines (ACE), which are completely separate hardware processors within the GPUs that are solely responsible for managing compute shaders, whilst the graphics command processor handles the graphics shaders [18]. In this case, having two command queues, one for graphics and one for compute tasks means that each queue can submit commands without waiting for the other tasks to complete.

However, the architecture for NVIDIA cards has always been different. Due to not having discrete compute processors, or a hardware scheduler, asynchronous compute has not been possible on their cards until recently. When the Maxwell 2 (900 series) architecture came out, NVIDIA claimed that it could handle async compute shaders by exposing 31 compute queues and the ability to mix them with the graphics queues [41]. However, there was a controversy as this was not inherently concurrent as it still had no hardware-level scheduler. The biggest problem for the cards is due to static load balancing. Static load balancing is where the GPU would allocate resources ahead of time and not be able to modify them on-the-fly. This meant that if the resources were badly allocated between streaming multiprocessors the performance could take a hit. See Figure 2.2 for a visual representation of this.

In 2016 when the Pascal (1000 series) architecture was released, it was announced that the GPUs would now have dynamic load balancing which meant that the re-

source allocation to the GPU can be altered on the fly to allow for asynchronous compute. Note the usage of the once idle time in figure 2.2.

Since this feature is a relatively new capability for NVIDIA cards, the intended research and evaluation of this project will be focused on testing the NVIDIA GeForce GTX 1080. In the future, this project can be expanded to include performance analysis of multiple types of GPU architectures and whether implementation can match that of the hardware advantage of AMD cards. However, this project first and foremost aims to exploit workload concurrency via async compute and whether or not a Pascal card's performance takes a hit depending on the implementation of async.

This project, if successful, will hopefully be useful to developers working on games and real-time simulations. There are very few released titles that currently use asynchronous compute technology. In fact, in researching the topic there seems to be only information and test results of two games: Ashes of The Singularity, and DOOM. Researching in such a new technology has proved to be rather difficult. Most research into compute has been implemented with CUDA for "embarrassingly parallel" problems. In searching for asynchronous compute for rendering performance, relevant results are hard to come by.

C Vulkan

In order to effectively use compute within this project, an in-depth knowledge of GPU architectures and computer graphics is required. This project requires the use of using a GPU for general-purpose computation via a traditional graphics API and graphics pipeline.

Vulkan, launched on February 16th 2016, is a graphics and compute Application Programming Interface (API) [46]. The Vulkan SDK was chosen to use for development as it is a unification of both graphics and compute functionality and is a relatively new technology. One disadvantage of using Vulkan over a different graphics API is that it is quite verbose. However the benefit of this is that it leads to a greater understanding of errors - what they are or how they occur. Also it gives a finer control over the GPU by requiring to explicitly state every function needed.

.10 Work so far

During the first few weeks of implementation, the underlying application was developed to set-up a Vulkan rendering framework to be able to output images to a screen. It has taken a reasonable chunk of time due to the level of understanding required for using the new Vulkan API. Below is a list of several learned methods:

- Command queues and the supported queue families of a device

- How to structure a render pass
- Creating the graphics pipeline
- Swapchains, image views and image buffers
- How to setup command pools with semaphores
- Allocating memory on the GPU
- Copying memory across from the host
- Vertex and Index buffers
- Uniform buffer objects
- Texture sampling and loading
- Depth buffering
- Instance rendering

The most challenging part of the implementation so far has been in understanding uniform buffer objects. Unlike OpenGL, uniform values are bound to the shader through description sets, which are created from a descriptor pool. This is an important step to learn as the way memory is copied to the GPU will affect performance if done incorrectly. Using separate command queues for compute and graphics work will rely on passing memory between shaders efficiently.

The main ground work of the implementation has been completed now, with a framework consisting of roughly two thousand lines of code to render a triangle on screen with textures and depth buffering. This initial research and implementation allows the project to continue as the actual simulation and compute shaders can now be implemented.

A Plan of Work

The next steps for the project now the initial implementation and rendering framework is in place, is to work on completing an n-body simulation using compute shaders.

Below are the milestones from the Initial Project Overview document, now with an estimated individual deadline, referring to university week numbers.

Milestones:

- COMPLETED - Rendering engine using Vulkan
- WEEK 13 - Rendering something complex using compute shaders

- WEEK 16 - Render same scene using asynchronous compute shaders
- WEEK 1 - Analyse performance differences
- WEEK 6 - Optimise code for async
- WEEK 8 - Re-evaluate performance using these optimisations

Overall, the aim is to finish the implementation before Christmas. This leaves the 3 week break before trimester two to begin the literature review and to start running tests. During trimester two, work will be done in writing the report and evaluation of the results. The majority of the report writing to take place between weeks 6 to 10. Aiming to finish the poster and dissertation for week 11. The plan for writing is to take each section at a time whilst working on the implementation of the optimization.

.11 Evaluation

There are two main aims to investigate within this project; an objective and a subjective evaluation. The objective evaluation will include details of how much the performance of the simulation has improved - if at all. Whereas, the subjective evaluation is how complex the implementation is. This can be measured by how much code needs to be changed and whether the increase in performance validates using the technique. A key outcome of this project will be to see whether the implementation of asynchronous compute shaders can be optimised for NVIDIA GPUs. During this semester, Concurrent and Parallel Systems will teach different techniques for evaluating performance that will aid in the coming stages of the project.

The current plan is to investigate different configurations, measuring performance by timing render frames. Exploring any changes to frame rate when either the resolution of the spheres, or frequency of them, are changed.

The evaluation of the ease of implementation will include assessing how much of the main game code will have to be re-structured. For example, a small performance increase may not be worth changing the entire code base and design of the game engine. Whereas the optimisations will be deemed easier to implement if they include inserting code rather than refactoring it.