University of Derby

Department of Electronics, Computing & Mathematics

A project completed as part of the requirements for
BSc (Hons) Computer Games Programming

entitled

# Techniques for the utilisation of heterogenous multi-GPU configurations in realtime rendering

by

Andrew C. James

aandrew444@gmail.com

May 2019

## 1. ABSTRACT

Rapid realtime rendering is important to many applications such as games, simulations and computer aided design (CAD). In these scenarios a common way to improve rendering performance is to use more than one Graphics Processing Unit (GPU). However, many applications exhibit issues such as diminishing performance improvements with the addition of extra GPUs and instability. The introduction of low-level graphics Application Programming Interface (APIs) in recent years has provided more control over multi-GPU setups. This paper presents four techniques that have the potential to address the current performance issues with a focus on heterogenous configurations of two discrete GPUs. While this is the focus, many of the techniques discussed here can be scaled to support more than two GPUs with relative ease. The rendering performance of these techniques was evaluated using a purpose-built rendering engine.

## 2. TABLE OF CONTENTS

# 3. INTRODUCTION AND AIMS

The key goal of this paper is to evaluate techniques that allow more than one GPU in a system to be used together effectively to improve rendering performance. Using multiple GPUs for rendering has been possible since 1998 with the induction of scan line interweave by 3Dfx (Pabst, 1998). With the introduction of newer low-level graphics APIs in 2014 (Advanced Micro Devices, 2014), explicit multi-GPU programming has become possible and with it the ability to use heterogenous multi-GPU setups.

Many applications require smooth and responsive rendering and would benefit from improved rendering performance. Examples include games, simulations and computer aided design (CAD) work. Using multiple GPUs would allow higher framerates for games, this improves the experience and responsiveness. For professional applications such as CAD, larger model/assemblies could be worked on whilst maintaining application responsiveness and improving productivity. For simulations, this paper focuses on fast multi-GPU rendering rather than multi-GPU compute, but may still be of use.

This paper presents and analyses four techniques that utilise heterogenous multi-GPU setups. The focus is on two discrete GPUs with very different levels of performance. The design of the engine is presented with a focus on the implementation of the investigated techniques.

Each technique was tested at three different resolutions with technique specific variations. From these results a conclusion was drawn for each resolution and each technique. By analysing the results for all of the techniques a conclusion has been drawn detailing the best strategies for effective utilisation of heterogenous multi-GPU configurations. The limitations of the investigated techniques and their implementations are presented along with possible solutions. Finally, ideas for other techniques and improvements to the test engine are detailed.

## 3.1 AIMS

For many years, the only way to make use of multiple GPUs for realtime rendering was using homogenous GPU sets with techniques which were controlled by the graphics driver. With the introduction of DirectX 12 in 2014 (Microsoft Corporation, 2014) the use of heterogenous configurations is now possible and applications now have control of each GPU explicitly. This creates a need for techniques that can effectively utilise multiple GPUs of different performance in the same system.

To effectively develop and analyse the techniques in this paper, it was identified that a rendering engine was required that supported the following features:

- Support for heterogenous multi-GPU rendering.
  - Targeting native DirectX 12 on the Microsoft Windows 10 platform.
- An accurate performance capture system with low performance overhead.
- A fast and easy to use framework for inter-GPU transfer of resources.
- Robust and flexible synchronisation between the CPU and the engines/subsystems on each GPU (Copy, Compute and 3D).

At the time of writing there are no publicly available rendering engines that support the above features, so a custom engine was designed and implemented for this project.

With the key features implemented, the following techniques will be implemented and investigated:

- Multi-GPU shadow mapping
- Asynchronous multi-GPU shadow mapping
- Split frame rendering
- Split frame rendering with multi-GPU shadow mapping

The results will be gathered from a system with two GPUs of very different performance to test the scalability of the techniques. The results will be analysed to identify key factors that affect the performance for these multi-GPU techniques.

During this paper it is assumed that the most powerful GPU in the system is GPU 0 as this GPU needs to handle extra work related to displaying the final image and any passes that are not running in a multi-GPU configuration, such as UI or post-processing.

## 4. GLOSSARY

| | |
|---|---|
| GPU: | Graphics processing unit |
| CPU: | Central processing unit |
| FPS: | Frames per second |
| Frametime: | Time a single frame takes to render |
| PCIe bus: | The standard interface between a GPU and the rest of the system |
| Render pass: | A section of rendering work |
| Main pass: | The render pass that draws the scene that is displayed |
| SFR: | Split frame rendering |
| IDE: | An integrated development environment |
| VS: | The Microsoft Visual Studio IDE |
| Discrete GPU: | A GPU that is a separate physical card with its own local video memory |
| IGPU: | Integrated GPU, a GPU chip integrated on the same die as the CPU that uses a section of the CPU's memory for video memory |
| Linked GPUs: | Two or more GPUs linked by an extra physical connection and/or the graphics driver |
| Unlinked GPUs: | Two or more GPUs in the same system with no connection provided by the graphics driver |
| SLI: | Scalable Link Interface, Nvidia's linked GPU solution. |
| UE4: | Unreal Engine 4 (Epic Games, 2014) |
| GB: | Gigabytes |
| DirectX 12: | Microsoft's latest graphics API, it supports explicit heterogenous multi-GPU |
| Direct Lighting: | Light that travels directly from a source |
| Indirect Lighting: | Light that has bounced or been otherwise changed by the environment |
| GI: | Global Illumination, the indirect lighting generated by the scene and skybox |
| Core Clock: | GPU's core update rate |
| Memory Clock: | GPU's memory update rate |
| HD, (1080p): | High definition: 1920 by 1080 pixels |
| QHD, (1440p): | Quad High definition: 2560 by 1440 pixels |
| UHD, (2160p): | Ultra High definition: 3840 by 2160 pixels |
| Copy engine: | Hardware capable of copying data across the PCIe bus concurrently with other GPU work |
| DMA: | Direct Memory Access |
| Deferred Renderer: | A renderer that uses two passes, one for geometry and the other for lighting. |
| Forward Renderer: | A renderer that uses one pass for geometry and lighting |
| Render Target: | An image/memory that a GPU can write colour data to |
| Back buffer: | The screen's render target that is currently being written to |
| Shader: | A small program that runs on the GPU |
| Compute Shader: | Shader that runs compute work such as physics simulation, cannot use rendering hardware |
| Pixel shader: | Shader that runs to calculate the colour of a pixel |
| Geometry shader: | Shader that can edit or create geometry on the GPU |
| Millisecond (ms): | A thousandth of a second |

# 5. LITERATURE REVIEW

During the initial research for this project it became apparent that there had been very little work done in this area. Graphics APIs have supported explicit multi-GPU setups since Mantle (Advanced Micro Devices, 2014) in 2014, yet there has been slow adoption due to the complex nature of low-level APIs. In more recent years some research has been done however this is mainly investigating matched GPU sets. At the time of writing no work has been published about using multiple heterogenous GPUs to accelerate rendering performance.

Initial research started with a search for published papers relating to heterogenous multi-GPU rendering. Unfortunately, none were found directly relating to multi-GPU rendering, although a few were found that related to heterogenous multi-GPU compute work. However, these were focused on simulation workloads and so were of little use. Limited information was found from graphics API developers and GPU manufactures. This related almost exclusively to matched sets of GPUs rather than heterogenous GPUs. However, some of the techniques would work in a heterogenous setup and were included.

Due to the lack of relevant previous work this section provides an introduction to key topics in realtime rendering and how they relate to heterogenous multi-GPU rendering. Next, hardware connections used in multi-GPU setups are detailed and analysed. Finally, existing multi-GPU techniques and their suitability for use in heterogenous multi-GPU setups are explored.

## 5.1 GRAPHICS APIS:

This section details current Graphics APIs and their support for explicit multi-GPU programming. Graphics APIs provide a hardware independent interface that supports high performance control of graphics hardware. These APIs are the only way to interface with graphics hardware and the performance of the API directly impacts the applications performance. Many rendering engines support multiple APIs which can lead to compromised implementations of some APIs due to the large differences in architecture.

### 5.1.1 OpenGL

OpenGL (The Khronos Group, Inc., 1992) is one of the first “modern” graphics APIs having seen widespread use from the late 1990’s though to the present day. It is being phased out for more modern API’s like DirectX 12 and Vulkan (particularly on Linux). OpenGL has been the de facto standard for cross platform applications and has only recently been replaced by Vulkan on non-Windows platforms, the exception being Mac OSX which uses Apple’s propriety Metal API (Apple, 2019). OpenGL employs a state machine architecture, and this was a reasonable approach on single core/single thread applications with one GPU. However, with the advent of multi-core and multi-GPU systems this architecture is no longer suitable. OpenGL does not support explicit multi-GPU programming.

### 5.1.2 DirectX Up to Version 11

DirectX 11 (Microsoft Corporation, 2008) is the most common iteration of Microsoft’s Direct 3D Graphics APIs, it is an incremental improvement to DirectX 10 (and DirectX 9 before that). It is more similar to OpenGL than to the next DirectX iteration, DirectX 12. It uses the concept of the immediate context; all graphics commands are executed on this immediate context and only one can exist at once. The immediate context represents a single GPU, or a linked set of GPUs managed by the graphics driver. This prevents efficient threading or explicit multi-GPU.

### 5.1.3 Mantle

With the introduction of AMD’s Mantle (Advanced Micro Devices, 2014) API the industry has begun to shift to lower level graphics APIs with lower driver overheads such as Vulkan and DirectX 12. One of the key design goals with Mantle was to improve CPU and GPU performance (Davis, 2015) compared to other APIs. Mantle provides direct access to the GPU command buffers and explicit control over their submission and execution. This control allows for threading of command buffer generation (sometimes referred to as recording). The application has much greater control of resources and synchronisation which can lead to better overall performance.

Low-level APIs provide greater control and perform better than previous APIs. However, they need to be carefully implemented otherwise they can perform much worse and be unstable. This is first API to provide explicit access to the independent sections of GPU hardware namely the DMA engine(s) and compute pipelines. It is the first API to support explicit multi-GPU programming. In 2015, AMD announced that Mantle will no longer be developed (Ung, 2015) and released a programming breakdown of the API which greatly helped the Khronos group with the GLNext Project (Later renamed Vulkan).

### 5.1.4 Vulkan

Vulkan is very similar to DirectX 12 in concept and many engines support both. Vulkan is based on Mantle hence why AMD GPUs commonly perform better (Ung, 2015) when using it. However, at the time of writing there is no support for unlinked multi-GPU setups thus making the API unsuitable for use in the research documented in this paper. Vulkan is commonly referred to as the next version of OpenGL however Vulkan shares more similarities with DirectX 12 than OpenGL.

### 5.1.5 DirectX 12

In 2014, Microsoft announced DirectX 12 (Microsoft Corporation, 2014); the next iteration of DirectX, this release brought many changes to the API. The API gave developers much lower level access to hardware and reduced driver overhead. In DirectX 11 all graphics functions would go through the immediate context object. With DirectX 12 this has been removed entirely in favour of command lists, which are low level abstractions of GPU native command buffers. A command list is filled with rendering commands and then executed on a device's command queue. This change allowed command list generation, a commonly CPU performance limiting process, to be run in parallel across multiple threads. Device objects now map directly to physical hardware allowing explicit multi-GPU programming in both linked and unlinked GPU setups.

DirectX 12 also allows the developer direct access to GPU engines. A GPU engine is an abstraction of a hardware pipeline that can execute a fixed set of functions. A command queue object can be created for any GPU engine and there is no limit on the number of command queues linked to a GPU engine. It is typical of modern GPUs to only have one 3D hardware pipeline with a few compute engines and one copy engine. Nvidia Quadro series of workstation cards have had two DMA engines for a long time (Nvidia, 2010) and with the introduction of the 10 series, consumer grade cards now have two DMA engines as well (Nvidia, 2016). DirectX 12 allows each command queue to execute command lists concurrently as the synchronization of GPU and CPU is handled by the application. This is independent of the hardware's concurrent execution capabilities.

### 5.1.6 Pipeline State Objects

The pipeline state of a GPU is the collective name for all of the hardware settings used on a GPU. This includes what shader the GPU should use, whether to use a depth stencil etc. Changing the GPU pipeline can be very time-consuming process on the GPU and so should be avoided where possible.

One significant change brought with the new low-level APIs (Vulkan, DirectX 12 and Mantle) is the introduction of the pipeline state objects (PSO) which provides access to the GPU's pipeline state. These replace the individual state setting methods used in OpenGL and DirectX 11(see Code Sample 1). This change allows the driver to make hardware specific optimisations for switching state. DirectX 11 has an inefficiency where after each state setting method the driver needs to flush GPU caches so that the GPU is left in a state ready for commands. This means multiple cache flushes could occur in the code sample below. In the new PSO system only one flush would need to occur.

```
//D3D11
DeviceContext->VSSetShader(VertexShader, 0, 0);
DeviceContext->PSSetShader(PixelShader, 0, 0);
DeviceContext->RSSetState(RasterState);
//D3D12
CommandList->SetPipelineState(PSO);
```

*Code Sample 1: Example showing the difference in changing pipeline state between DirectX 11 and 12*

Many API's execute validation when a PSO is created, reducing the runtime overhead for changing GPU pipeline state. It is strongly recommended to developers that PSOs should be created before runtime or asynchronously at runtime as creation and validation can take a significant amount of CPU time (relative to frame time).

## 5.2 INTER-GPU DATA TRANSFERS

Almost all multi-GPU techniques require some graphics data to be transferred between GPUs (with the exception of AFR). The speed of this transfer significantly affects the performance of a given technique.

### 5.2.1 The PCIe Bus

Most modern GPUs use the Peripheral Component Interconnect Express (PCIe) 3.0 connection standard to interface with devices in the system. This has a maximum throughput of 1 gigabyte per second (GB/s) per lane per direction (PCI-SIG, 2018). Typically, GPUs take up 16 PCIe lanes (x16) so have a total bidirectional bandwidth of 16GB/s with the rest of the system.

Modern consumer grade processors, such as Intel i5s, i7s, AMD R7s, etc, only have enough PCIe lanes to run two GPUs at the slower x8 speed so the transfer speed is only 8 GB/s. This contrasts with the GPUs onboard memory (sometimes referred to as the local memory segment) being many times faster. For example, the Nvidia GTX 1080 video card uses Graphics Double Data Rate 5X (GDDR5X) memory which has a rated bandwidth of 320GB/s (Nvidia, 2016), a 20 times increase over PCIe x16. Also, PCIe can suffer from congestion from other devices thus slowing the transfer speed.

Currently almost all consumer grade GPUs do not allow other devices to write directly to their local memory segments, so any transfer must first be copied to system memory and then copied to the target GPU. For example, in the test system (defined in section 7.2) copying 2.86Mb of data takes around 4.33ms at a transfer speed of around 6.6GB/s. This was measured on the displaying GPU, so some bandwidth is taken for OS-GPU communication.

PCI-SIG has recently released the PCIe 4.0 specification which doubles the available bandwidth to 32GB/s for a x16 interconnect. The AMD X570 chipset for the ZEN 2 architecture is expected to launch with PCIe 4.0 support.
AMD has recently released the Radeon Instinct MI60 GPU for data centre applications using the PCIe 4.0 interface (Advanced Micro Devices, 2018). It is expected that PCIe 4.0 GPUs will become available in the consumer market over the coming years. The MI60 is a General Purpose-GPU (GP-GPU) designed to be used for general compute tasks in servers and has no video outputs. The MI60 allows direct access to its local memory segment over the PCIe bus which would greatly improve performance for the techniques investigated in this paper.

PCIe 5.0 is expected to be finalised by the PCI-SIG in 2019 and offers bandwidth of up to 64GB/s on a x16 interconnect. This will provide a notable improvement on consumer grade hardware as the bandwidth available to an x8 connection is 32GB/s

### 5.2.2 Other Connection Standards

Direct GPU to GPU connections provide many advantages over the PCIe bus. Firstly, they are normally faster. Secondly, the direct nature of the connection removes the need to copy data into host memory and out again, so the data is only copied once. Thirdly, they don’t suffer from congestion caused by other devices that could reduce throughput.

Nvidia has two technologies that provide direct GPU-GPU connection; NVLink and the SLI Bridge. NVLink is a direct physical connection between two or more GPU’s. Introduced in 2014 (wikichip.org, 2014) it has been used in their Quadro series of workstation cards and the recent GeForce RTX series. This link is faster than PCIe; on the latest Quadro card, the QUADRO RTX 8000, NVLink is rated at a transfer rate of 100GB/s, compared to 672 GB/s that its onboard GDDR6 memory is capable of. The NVLink connection is 6.25 times faster than a PCIe Gen 3.0 x16 link.

With Nvidia’s new RTX 20 series the NVLink Interconnect has been added to the consumer cards, this version of the link allows direct GPU transfers at around 50GB/s (Nvidia, 2018) when in a matched pair. This increase in bandwidth would improve the performance of many of the techniques investigated in this paper. Interestingly the GV100 built on the Volta architecture has double the NVLink transfer speed compared to the RTX Quadro series. This is due to the inclusion of two NVLink connections leading to a though put of 200GB/s (Nvidia Corporation, 2018). Although this transfer speed is nowhere near the speed of its High Bandwidth Memory 2 (HBM2) memory which is capable of transferring at around 874 GB/s (Nvidia, 2017) which is 4.4 times faster than NVLink and 55 times faster than a PCIe x16 connection.

Before the RTX 20 series Nvidia SLI required an additional connection of an “SLI bridge” cable. This connection is believed to be used for synchronization rather than data transfer (Anon., 2010). The fact it is only capable of a transfer rate of 1GB/s supports this idea (Nvidia Corporation, 2019).

## 5.3 CURRENT MULTI-GPU TECHNIQUES

Both AMD and Nvidia provide homogenous multi-GPU scaling in Crossfire (Xfire) and Scalable Link Interface (SLI) respectively. Both technologies link two identical GPUs together and provide access to direct GPU-GPU connections (if available). They are extremely similar and for most purposes are interchangeable.

### 5.3.1 Alternate Frame Rendering (AFR)

AFR is a techniques that alternates which GPU renders each frame so, frame one is rendered on GPU 0, frame two on GPU 1 and frame three on GPU 0 again (Nvidia Corporation, 2018). This increases the time each GPU has to render each frame by the number of GPUs in used. So, in a dual GPU setup the time is doubled. However the increasing popularity of temporal effects such as temporal anti-aliasing (Karis, 2014) has meant that AFR is a less viable technique (Sjoholm, 2017) as large amounts of data needs to be transferred between GPUs to run temporal effects.

This technique can suffer from micro stuttering where the time synchronisation between two or more GPUs is poor. In more modern SLI or Xfire implementations frame pacing (Marinkovic, 2016) is used to sync the two GPUs and reduce this issue. AFR has issues when applied to a heterogenous multi-GPU setup as the performance difference of the GPUs would have to be divisible by the number of GPUs to achieve the best effectiveness. Also, if the GPUs have different amounts of video memory either the smallest card’s framebuffer limits the useable memory of the other GPUs or the GPU with the smallest local memory size would have to page to system memory causing very poor performance defeating the goal of a multi-GPU setup.

### 5.3.2 Split Frame Rendering (SFR)

Split Frame Rending splits the frames rendering work spatially rather than temporally meaning the screen is split into sections which are rendered by each GPU and then stitched together by the displaying GPU for output. This eliminates micro stuttering which AFR can suffer from, however it requires a large amount of data to be transferred between GPUs. This means that PCIe transfer speed is the key limiting factor for SFR. There is potential duplication of data between GPUs such as per frame shadow maps etc.

This technique is suited to heterogenous multi-GPU setups as the frame can be split very finely giving lots more of control over the work splitting. However, distributing work using regions can lead to GPUs being underutilised as typically the shading complexity of a scene is not uniform. A variation of SFR that tries to address this issue is multi-GPU checkerboarding, instead of splitting the scene into regions, it is split on a per pixel basis. This splits the scene to share the complexity more evenly to each GPU.

### 5.3.3 Frame Pipelining

Shortly after DirectX 12 was released Nvidia published an article on a new multi-GPU technique called frame pipelining (Sjoholm, 2017). This works by splitting each frame’s work in half with the first card executing the first half and then the output being transferred to the second GPU to complete the frame. This technique is limited by PCIe transfer speeds as large chunks of data needs to be transferred. The copy step limits the achievable framerate as it takes around 14ms to copy. Nvidia demonstrated this technique on two GTX 980 TI’s in SLI however it could easily be utilised on a heterogeneous GPU set.

### 5.3.4 Heterogeneous Multi-GPU Techniques

#### 5.3.4.1 Microsoft DirectX 12 Sample

As part of the launch of DirectX 12, a developer example demonstrating heterogeneous adapters was released. This example used the first GPU to raster many triangles and then the render target was copied to the second GPU to be blurred and output to the screen. This is a very simple example demonstrating the potential performance improvement of explicit heterogonous multi-GPU in DirectX 12.

#### 5.3.4.2 Ashes of The Singularity

Ashes of the Singularity is a real time strategy game released by Stardock (Stardock, 2016). It was one of the first games to support DirectX 12 and heterogenous multi-GPU. For multi-GPU scaling it implements AFR and while it does support heterogenous GPU sets, it works best with GPUs of similar compute power (Williams & Smith, 2016). This showed off DirectX 12’s ability to utilise GPUs from different manufacturers in the same system in real world scenarios.

## 6. METHODOLOGY

This section details the rendering engine that was built to test the techniques. A custom rendering engine was developed as it was identified that none of the available rendering engines support heterogenous multi-GPU. It was considered to use UE4 as a starting point, however it quickly became apparent that it would take an un-feasibly large amount of work and time to implement the techniques. Other engines, such as Unity (Unity Technologies, 2005) were considered but discounted for the same reason.

The test engine is written in C++ as DirectX 12 natively is a C++ API. The test engine needed to support shadow mapping (see 6.1.2) as two of the techniques are based on this.

During this project it is assumed that GPU 0 is the most powerful GPU and displays the final image. GPU 1 is the less (or equally) powerful GPU and does not have a display attached.

## 6.1 RENDERING ENGINE DESIGN

### 6.1.1 Core Renderer Design

The test engine supports both forward and deferred rendering paths. It implements a physically based rendering (PBR) pipeline which uses the split sum approximation to compute Bidirectional Reflectance Distribution Function (BRDF) as used in Unreal Engine 4 (Karis, 2013). The BRDF defines how a surface should be rendered as it calculates how light would interact with it. This shading model was chosen as it is the current industry standard and allows the test engine to be more representative of a modern game engine.

Post processing is implemented using compute shaders however only bloom and colour correction are currently implemented. The colour correction pass writes the rendered output of the main pass to the screen’s back buffer and adjusts for exposure.

### 6.1.2 Shadow Mapping

Shadow mapping is an industry standard technique for rendering shadows in real time hence why it was implemented for this project. It renders the scene from the perspective of the light and writes the depth to a texture. When rendering the main pass, the distance from the light is calculated and compared with the generated depth texture, determining if the light can “see” the pixel and so whether it should be in shadow. Directional lights use one depth texture oriented in the direction of the light. For point lights a cube map is used, a cube map is six textures that together form a cube. This makes point lights much more expensive to render than directional lights as the scene needs to be rendered six times (once per direction) per light. A geometry shader is used to translate the geometry for each side of the cube map texture.

### 6.1.3 Multi-GPU Shadow Mapping

Shadow maps are rendered on GPU 1, then pre-sampled to a screen sized render target and copied to GPU 0. The pre-sampling pass calculates the visibility values for each light the same way the main pass does. When running the main pass, the pre-sample buffer is read from and applied to the corresponding light. The pre-sampling buffer is the same resolution as the back buffer although a lower resolution could be used to reduce the amount of data transferred and pre-sampling time. This would introduce visual artefacts. The render target uses one component of a texture per light so a single component 8-bit texture format stores one light’s data. This reduces the amount of data transferred across the PCIe bus which is a key limiting factor.

Due to limitations in DirectX 12, an 8-bit three component per pixel format is not available to be used for use when sampling three lights. Instead a four-component texture is used. This limitation is most likely due to hardware design optimisations. The pre-sample shader is compiled based on the number of pre-sampled lights for performance.

### 6.1.4 Asynchronous Multi-GPU Shadow Mapping

Asynchronous shadow mapping is a derivative of multi-GPU shadow mapping focused on hiding the render and transfer time of the shadow maps. It does this by pre-sampling the last frame’s shadow maps and transferring the pre-sample buffer to GPU 0. Once pre-sampling is complete, GPU 1 starts rendering the shadow maps for the next frame while GPU 0 is finishing the current frame’s rendering. This does lead to visual differences in the scene, as the shadows are one frame behind but, in most situations, this would not be very noticeable. This technique provides a way to improve framerates without reducing shadow map resolution and would be useful to games that demand a high framerate.

### 6.1.5 Split Frame Rendering (SFR)

The SFR implementation used in this project splits the screen horizontally by a ratio. The ratio determines the amount (in percent) of the frame is rendered on GPU 0 with the remaining being rendered by GPU 1. This is achieved using scissor regions, which indicate what pixels the GPU should render. Shadow maps are computed on each card. Once complete the result is copied back though host memory and merged with other sections of the frame to form the final image. Post-processing and UI are rendered on GPU 0 for simplicity. UI is normally rendered to an offscreen buffer which could be done in parallel to the copy operation, improving performance.

### 6.1.6 SFR With Multi-GPU Shadow Mapping

The goal of this technique is to reduce calculation of redundant data on each GPU: the shadow maps. For a reasonably balanced set of GPUs the four shadowing lights would be split equally between GPUs and the results pre-sampled and copied to GPU 0. The main drawback with this technique is the PCIe bandwidth as in addition to the already large SFR buffers two more pre-sampled shadow buffers (which are linked directly to screen size) need to be copied.

## 6.2 ENGINE LAYOUT

The design of the test engine has been heavily influenced by Unreal Engine 4 (Epic Games, 2014) with attempts made to address some of its weaknesses.

### 6.2.1 Engine Modularity

The engine is built around a modular architecture; each module is its own dynamic-link library (DLL) and Visual Studio project. A visual overview of engine's modules and layers is provided by figure 1.

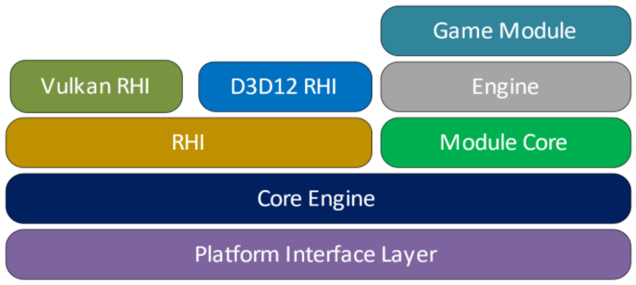

*Figure 1: Layout of Engine Modules and layers.*

The core engine is the most important section as it contains all the connecting code between modules and layers. This is built on the Platform Interface Layer which allows all modules above it to be platform independent. The module core is a key component of the core engine as it handles loading and unloading of modules.

The engine module contains higher level code such as AI, UI, scene management etc and provides an API for the game module. The game module contains the game specific code. The RHI handles both graphics API modules and makes the rendering code in the engine graphics API independent. The renderer is one large section of code that could be moved to its own module.

### 6.2.2 Engine Toolchain

The test engine uses a separate build tool application written in C# which utilises CMake to generate the project files. This allows much greater flexibility as CMake is cross IDE and platform. This tool uses C# files in the engine source code to define module properties, such as libraries, additional include directories etc. These are all named "<module name>. Build.cs" for clarity. "Target.cs" are used to define properties about the overall project.

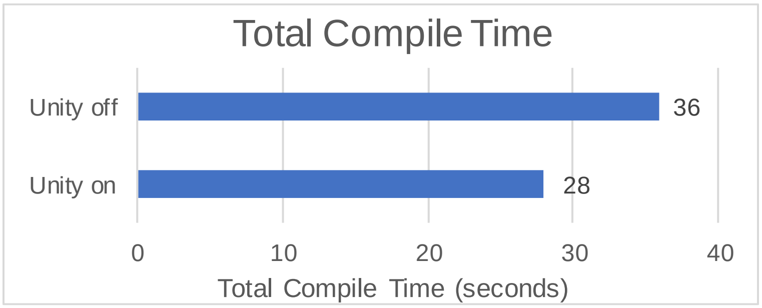

*Figure 2: Performance difference with and without unity build*

Unity build (not to be confused with Unity the game engine) is used on this project to speed up compilation (See *Figure 2*). This is implemented using Visual Studio 2017's experimental unity build feature (Buik, 2018), which requires the build tool to manually edit the generated project files. Unity build is used in many large projects notably Unreal Engine 4 (Epic Games, 2014). It works by merging collections of implementation files (.cpp or .c) together to form larger compilation units which are more efficient for the complier to process. This system can cause compile issues with files that define local functions or include the same headers. While this provides a small improvement curently, larger projects will see significant improvments to complie times.

## 6.3 ENGINE DESIGN

### 6.3.1 Platform Interface Layer

The platform interface layer abstracts all communication with the operating system and provides a wrapper for any platform specific functionality. This allows the test engine to be easily ported to another OS by simply filling out the generic functions with OS specific ones. This layer uses a compile time polymorphism with a generic platform base class providing standard library implementations of functions (where possible). The Windows header file is very large and defines many symbols with common names. Notably, min and max are defined as macros which cause conflicts with std::min and GLM::min/max. To avoid this issue the windows header is only included in the windows platform interface. This has the added benefit of reducing compile times for the project.

### 6.3.2 Render Hardware Interface (RHI)

The RHI is influenced by Unreal Engine 4 (Epic Games, 2014) with the key improvement being the explicit handling of device objects. The engine for this project initially supported OpenGL and DirectX 11. Once the need to use DirectX 12 was decided, the support for both OpenGL and DirectX 11 was removed as the differences between the API architectures would have caused conflicts.

The RHI abstracts the underlying graphics API's functions to provide an interface that is API independent. These higher-level objects also encapsulate API functionality for ease of use and compatibility with other APIs. This allows all rendering code to be API independent, so it only has to be written once.

Device objects map directly to a physical GPU and allow multi-GPU techniques to run independently of the underlying graphics API. In a linked multi-GPU setup, a device context is still created however its node index is set to address the node in the linked set it is representing. This means that the rendering code does not see any difference between linked and unlinked GPU sets and can use a mixture of both types.

One important aspect of this project is copying graphics data between GPUs, the RHI provides helper functions to copy data between GPUs and handles the associated synchronisation. Due to hardware limitations all inter-GPU data must first be copied to the CPU's memory and then copied to the target GPU, doubling transfer times. Currently the RHI does not handle linked adaptors where direct GPU to GPU transfer is possible.

### 6.3.3 Synchronisation

DirectX 12 does not handle command queue synchronisation however it provides objects and functions to synchronise them. Fences are used to synchronise command queues with other GPU command queues and the CPU. Fences halt execution of the GPU or CPU until signalled that execution can continue.

The validation layer provides checking for synchronisation issues and is very useful in tracking them down as the artefacts generated can be very subtle/hard to associate with synchronisation issues. GPUs rely heavily on pipelining so GPU stalls, where the pipeline is out of commands to execute, causes performance issues and needs to be avoided at all costs. To deal with this the CPU is normally two frames ahead of the GPU (or with triple buffing, three frames ahead). Ensuring the GPU pipeline is always filled at the expense of frame latency. This increases memory usage as some resources need to be double buffered such as screen render targets and command allocators.

### 6.3.4 Performance Analysis

The test engine uses DirectX 12 timestamp queries to capture timing data. A time stamp query writes the current GPU timestamp to a buffer. A timestamp is stored at the being and the end of a section of GPU work, from this the duration of the work is calculated.

To provide visual performance information a GPU timeline was implemented. This graph shows sections of work on all command queues and devices as bars (See Figure 3). One of the key metrics in multi-GPU performance is the time spent waiting for another GPU to finish. To measure this a timestamp is taken on the waiting device at the end of the last set of work and another when the other GPU has finished copying data and the waiting GPU can continue.

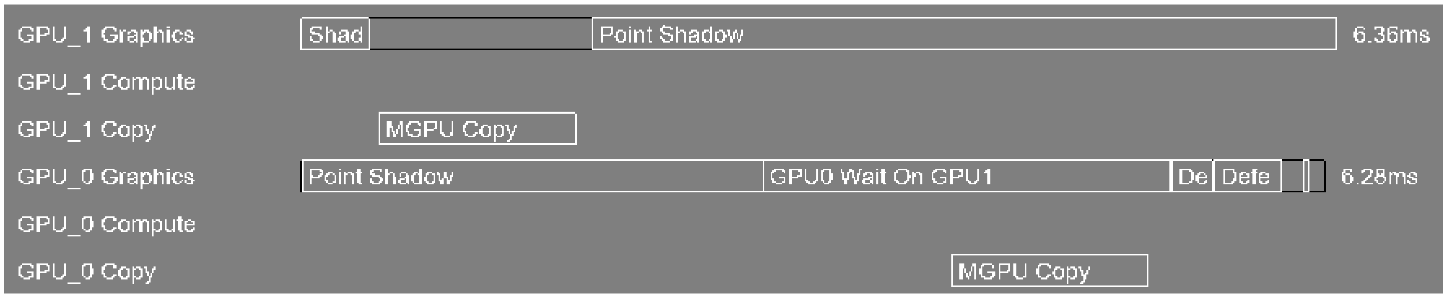

*Figure 3: A screen capture from the engine showing an example of the GPU timeline.*

For example, Figure 3 shows that the "MGPU Copy" task blocks the graphics queue increasing the frametimes. This was corrected by overlapping the graphics and copy work.

The test engine implements NVAPI (Nvidia, 2019) from Nvidia, which allows access to Nvidia specific functions and provides an API to gather information about Nvidia GPUs in the system. This allows the engine to read the utilisations and clock speeds of each GPU providing valuable information about their performance.

### 6.3.5 Shader System

Like Unreal Engine 4 (Epic Games, 2014), this engine has the concept of global and material shaders. A global shader is not involved in mesh rendering, for instance, one used for shadow mapping or post processing. A material shader is used to render meshes in the scene during the main pass.

Material shaders are compiled by the engine from a node system. All shading is implemented in the node system so that both forward and deferred renderers use the same code which is adapted for the different pipelines. This means that the output of both renderers is the same.

*In Shader_bloom.h:*
```
DECLARE_GLOBAL_SHADER(Shader_Bloom);
```
*In Shader_bloom.cpp:*
```
IMPLEMENT_GLOBAL_SHADER(Shader_Bloom);
```
*In Shader_Line.cpp:*
```
DECLARE_GLOBAL_SHADER_PERMIUTATION(Shader_Line_2D_ON, Shader_Line, bool, true);
DECLARE_GLOBAL_SHADER_PERMIUTATION(Shader_Line_2D_OFF, Shader_Line, bool, false);
```

*Code Sample 2: Example showing declaration of a global shader and permutation*

A global shader needs have the `DECLARE_GLOBAL_SHADER` macro defined which will register the shader (see Code Sample 2). This allows use of a shader to be as simple as requesting the shader from the complier with a single function. The shader complier handles the lifetime of all shaders and permutations.

## 7. TESTING METHODOLOGY

This section details how the techniques were tested and explains the metrics gathered.

## 7.1 PERFORMANCE DATA GATHERING

### 7.1.1 Offline Performance Capture

To test GPU performance, offline performance capture is normally used which improves the reliability and accuracy of results. However, none of the existing solutions support unlinked GPUs. Offline capture ensures that the GPUs are not idle during the test, since all the graphics commands were captured in advance. So, CPU performance would not affect the results. Also, it would provide more detail about the performance of each GPU.

Nvidia Nsight was evaluated due to its “multi-GPU” support but, it quickly became apparent that this only applied to Nvidia GPUs in SLI. Attempts were made to capture data from the test engine, however the Nsight capture engine crashes when any commands are sent to the extra GPU.

The Visual Studio Graphics Debugger was used for debugging during development, but it does not support multi-GPU as it only allows the application to create one device object.

## 7.2 TARGET SYSTEM

The results presented in section 8 were captured on the following system:

- AMD Ryzen 7 1700x CPU @ 3.5 GHz
- 16GB Dual Channel DDR4 Ram @ 2166 MHz
- ASUS Prime x370-Pro motherboard using the X370 chipset.
- Microsoft Windows 10 (build 1803).
- GPU 0 is a Nvidia GeForce GTX 1080
- GPU 1 is a Nvidia GeForce GTX 660

### 7.2.1 Target System Performance

The test system’s chipset and CPU only provide a maximum of 16 PCIe lanes for PCIe slots (WikiChip, 2018). This causes both GPUs to run at the slower PCIe x8 speed rather than the faster x16. To improve repeatability the target system was tested to find the maximum values for certain metrics important to the tests.

Max GPU clock was measured after ten minutes of stress testing in the aida64 stress testing application (finalwire, 2019). Both GPUs were stressed at the same time to ensure the thermal environment is as similar as possible to a real-world scenario. To compare relative performance of the GPUs, the GPU score from the PassMark performance benchmark was used (PassMark Software, 2019). A GTX 1060 was included as an example of a GPU that has more similar performance to the GTX 1080 than a GTX 660.

| | **GPU** | | |
|---|---|---|---|
| **Metric** | **GTX 1080** | **GTX 660** | **GTX 1060** |
| **Max. core clock** | 1974 (peak 1987) MHz | 1136 MHz | 1885 MHz |
| **Memory** | 8GB | 2GB | 6GB |
| **Max. transfer speed (measured)** | 7.9GB/s | 7.9GB/s | - |
| **PassMark** | 12187 | 4676 | 9094 |
| **Score percent of GPU 0** | 100% | 38% | 75% |

*Table 1: Specifications of relevant metrics for the 3 GPUs used.*

## 7.3 TESTING SCENARIO

In order to test the techniques a benchmark was created; this is a scene that is representative of an average gameplay environment. It includes four shadowing point lights and an average amount of geometry. During the benchmark the camera is moved between points in the scene to capture a more realistic performance indication than a static camera. The benchmark is automated to improve consistency. After each resolution was tested the render targets were resized and the benchmark then waits three seconds to allow framerates to stabilise before continuing. Before each benchmark run, the performance counters are cleared to avoid data from previous runs affecting results.

When a different multi-GPU method is needed the application is restarted, this is due to techniques needing to be setup during engine initialization and to eliminate any errors or performance issues that could be caused by runtime switching. The resolution defined in the results is the render target resolution of the main render target. This was used instead of screen size as the test system does not have a 2160p monitor available. However, for testing purposes the difference is negligible. Only a small amount of post-processing is run at screen resolution, which is constant throughout all results.

Three common resolutions were tested; 1080p, 1440p and 2160p as they represent common resolutions of monitors in the consumer market. The lower resolution of 720p was excluded as it is now a rare resolution, only used in games with a need for high performance or on lower end machines.

## 7.4 PERFORMANCE METRICS

During the benchmark tests, performance timers were recorded and when the run finishes are written to a CSV file. Results used in the tables are averaged over the benchmark. The performance timers capture in nanoseconds, but the results use milliseconds as this is more common for benchmarking and provides adequate resolution. Any higher resolution gives more variation due to the nature of GPU processing and scheduling variation.

All the benchmarks are completely GPU limited as the CPU performance is not being investigated in this project. The maximum CPU time observed was around 1.4 milliseconds (ms) which is much lower than the GPU time of the fastest test; around 5 ms.

GPU utilisation is not exclusive to the engine as the operating system will use a small amount of processing power to render the desktop/any other windows. However, this is negligible and is only present on GPU 0 as GPU 1 does not have a display attached.

### 7.4.1 GPU Boost

GPU boost is a technology that allows a GPU to dynamically adjust its core frequency to provide better performance. It does this when it detects available thermal and power headroom, up to a predefined frequency (unless changed manually). When a GPU’s utilisation is low it will start to reduce its core clock to conserver power, this negatively affects performance and should be avoided. Due to the nature of GPU boost it is possible to have a peak in frequency that is sustained for a short time before it stabilises to a lower frequency when the GPU has heated up. For example, The GTX 1080 in the test system boosted to 1987MHz before settling down to 1974MHz during sustained load. GPU boost does not consider the copy engine’s utilisation when determining clock speed. So, if lots of time is spent waiting on the copy engine the GPU will throttle down.

### 7.4.2 Metric Details

Each techniques analysis has its own set of variables that were investigated and analysed. Some were unique to the technique, but many were common and are explained below. Some result labels are omitted in graphs as they are inconsequential to the technique or too small to show.

| **Metric** | **Explanation** | **Expected Value** | **Units** |
|---|---|---|---|
| Frametime | Total time the frame took to render | Lower is better | ms |
| Point shadow | Time taken to calculate shadow maps for any lights resident on GPU 0 | Lower is better | ms |
| Point shadow on GPU 1 | Time taken to calculate shadow maps for any lights resident on GPU 1 | Lower is better | ms |
| Shadow pre-sample | Time taken to sample the shadow maps on a given GPU so that they can be transferred to another one. | Lower is better | ms |
| Inter-GPU copy time | Time taken to copy data to another GPUs local pool (though host memory). | Lower is better | ms |
| Inter-GPU copy time (pipeline timing graph) | Time taken to copy to or from host memory and the GPU's local pool. | Lower is better | ms |
| Wait on GPU 1 | Time spent on GPU 0 waiting for GPU 1 to complete work before rendering can continue. This includes transfer time. | Lower is better<br>0 is best. | ms |
| Main Pass | Time taken to shade the scene. This pass is dependent on all shadow lights from both GPUs. | Lower is better | ms |
| Main Pass on GPU 1 (SFR Only) | Time taken to shade the scene on GPU 1. This pass is dependent on all the shadow lights in the scene. | Lower is better | ms |
| GPU 0 Utilisation | Percentage of time where GPU 0 is considered busy in the last one second interval. | Higher is better, will not exceed 100 | % |
| GPU 1 Utilisation | Percentage of time where GPU 1 is considered busy in the last one second interval. | Higher is better, will not exceed 100 | % |
| GPU 0 Core Clock Stall Factor | The percentage reduction of GPU 0's core clock frequency from maximum. This gives an indication of the GPU throttling down due to stalling. | Lower is better 0% means no core clock stalling occurred | % |
| GPU 0 Core Clock | GPU 0's core clock frequency | Higher is better | MHz |
| GPU 1 Core Clock | GPU 1's core clock frequency | Higher is better | MHz |
| Inter-GPU Transfer Size | Size of data transferred between GPU's in a given frame. | Lower is better | MB |

*Table 2: Metrics present in results.*

To improve readability each technique has a shortened name:

| **Full Name** | **Short Name** |
|---|---|
| Multi-GPU shadow mapping | Multi-GPU shadows |
| Asynchronous multi-GPU shadow mapping | Asynchronous shadows |
| Split frame rendering | SFR |
| Split frame rendering with multi-GPU shadow mapping | SFR shadows |

*Table 3: Full technique names and their shortened names.*

## 8. REALTIME PERFORMANCE RESULTS

This section contains the results and analysis of the techniques investigated in this paper.

## 8.1 RESULTS LAYOUT

#### 8.1.1.1 Rendering Time Distribution

The rendering time distribution graph contains a breakdown of rendering times for sections of the pipeline that are most relevant to the investigated technique. These metrics are from both GPUs and are displayed in the approximate order of execution.

#### 8.1.1.2 GPU Utilisation

The GPU utilisation graph shows the utilisation of both GPUs during the test and the core clock stall factor of GPU 0. These metrics are important as they indicate how well the technique is utilising the available hardware. Below this graph there is a table listing the size of inter-GPU data transferred in a frame and the core clock values of both GPUs. GPU utilisation is an indication of how much of the GPU's power is being used by the technique.

A utilisation percentage above 90% is deemed good, above 80% is acceptable and anything below 80% is poor. It is the goal of any application to make full use of the hardware and with utilisation below 80% lots of performance is lost and will cause the GPU to begin throttling down which further reduces performance.

#### 8.1.1.3 Render pipeline time distribution

This graph shows the relation of time between the two GPUs. This is important to visualise what is happening in the technique and offers an easy way to see why a technique performs as it does. The key factor here is the length of GPU 1's execution time, if it is lower than GPU 0, the technique should provide an improvement over a single GPU (excluding issues relating to different work needed in multi-GPU techniques). This is indicated by the "Wait on GPU 1" metric.

If GPU 1's execution time is greater than GPU 0's, the technique is slower than a single GPU. If the wait time is significant GPU 0 will start throttling down which will affect frametimes significantly. In some timelines, the inter-GPU copy time is overlapped, this indicates that there is data being transferred both ways.

## 8.2 SINGLE GPU RESULTS

The graph below shows the single GPU results for all tested resolutions

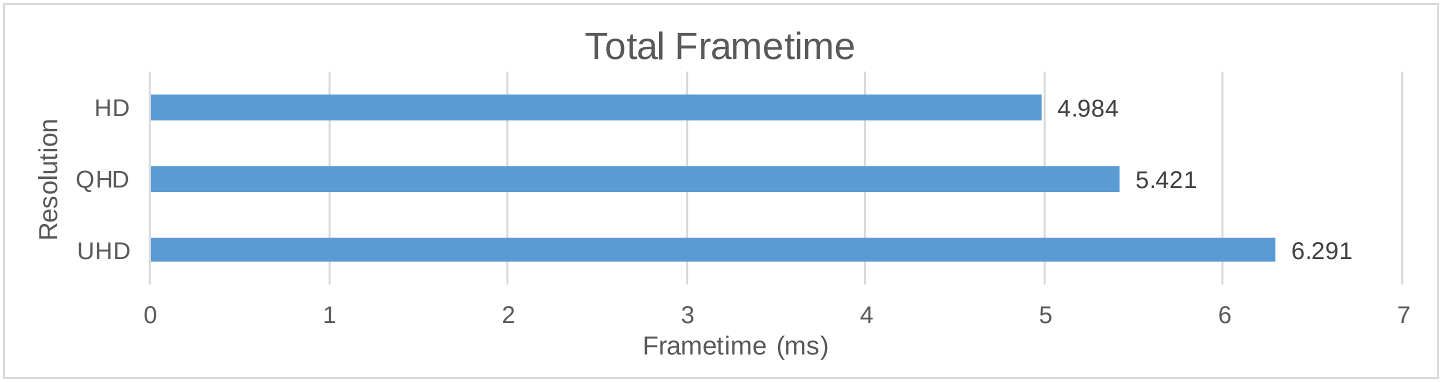

*Figure 4: Total frametimes for single GPU operation*

## 8.3 MULTI-GPU SHADOW MAPPING

This section contains the results and analysis for Multi-GPU shadow mapping. (see 6.1.3). The variable investigated is the number of lights calculated on GPU 1.

### 8.3.1 Multi-GPU shadow mapping with one light on GPU 1

For this variation, the shadow map for one light is computed on GPU 1 and the other three lights on GPU 0. The pre-sample buffer is sized for one light.

#### 8.3.1.1 Results

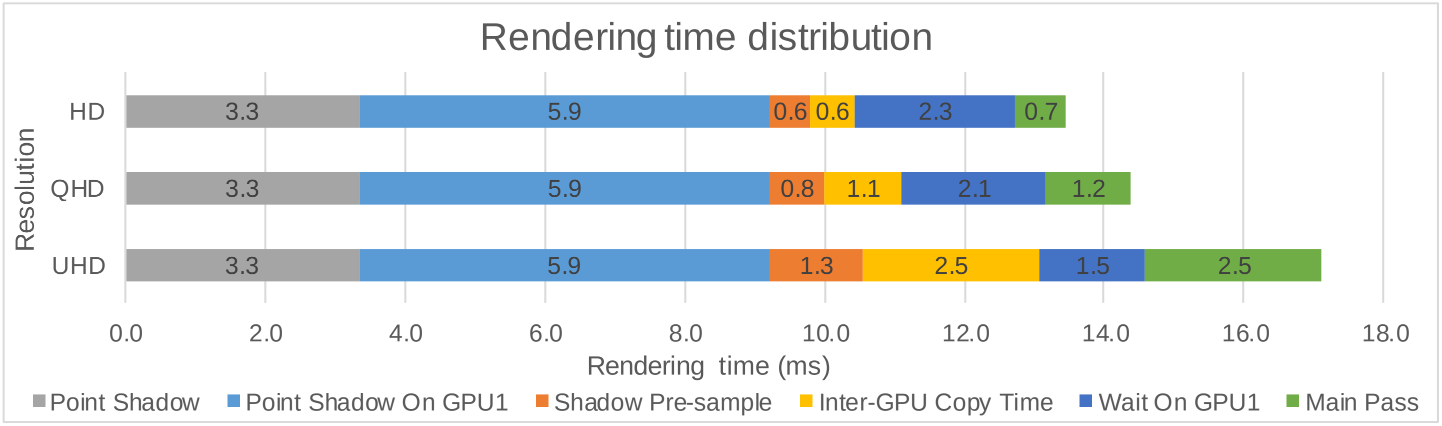

*Figure 5: Rendering time distribution for multi-GPU shadow mapping with one light on GPU 1*

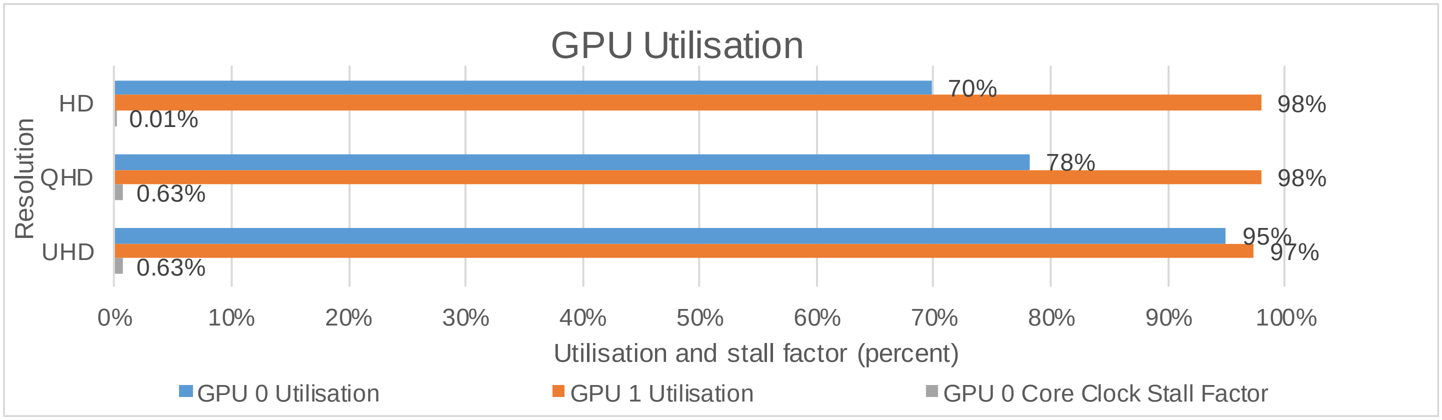

*Figure 6: GPU utilisation for multi-GPU shadow mapping with one light on GPU 1*

| **Metric** | **HD** | **QHD** | **UHD** |
|---|---|---|---|
| Inter-GPU transfer size (MB) | 0.207 | 0.369 | 0.829 |
| GPU 0 clock speed (MHz) | 1974 | 1962 | 1962 |
| GPU 1 clock speed (MHz) | 1137 | 1137 | 1137 |

*Table 4: Clock speed and Inter-GPU transfer size for multi-GPU shadow mapping with one light on GPU 1*

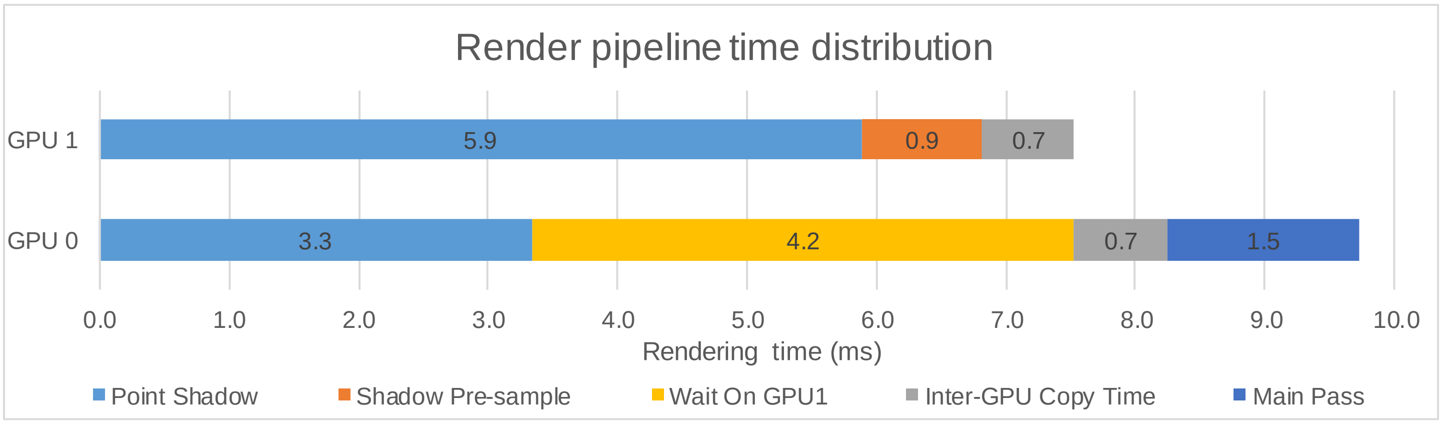

*Figure 7: Render pipeline time distribution for multi-GPU shadow mapping with one light on GPU 1*

#### 8.3.1.2 Analysis

This variation does not provide an improvement to frametimes versus single GPU operation as GPU 0 is waiting for almost 4ms on GPU 1 to render and transfer shadow data. The time spent transferring data is short, so this variation is not PCIe limited. Due to the short time that GPU 0 is idle it does not throttle down however utilisation at HD is low showing a loss in performance. At UHD the increased pixel shading work causes GPU 0 to have very good utilisation.

### 8.3.2 Multi-GPU shadow mapping with two lights on GPU 1

For this variation, two lights' shadow maps are computed on GPU 1 and two on GPU 0. The pre-sample buffer is sized for two lights

#### 8.3.2.1 Results

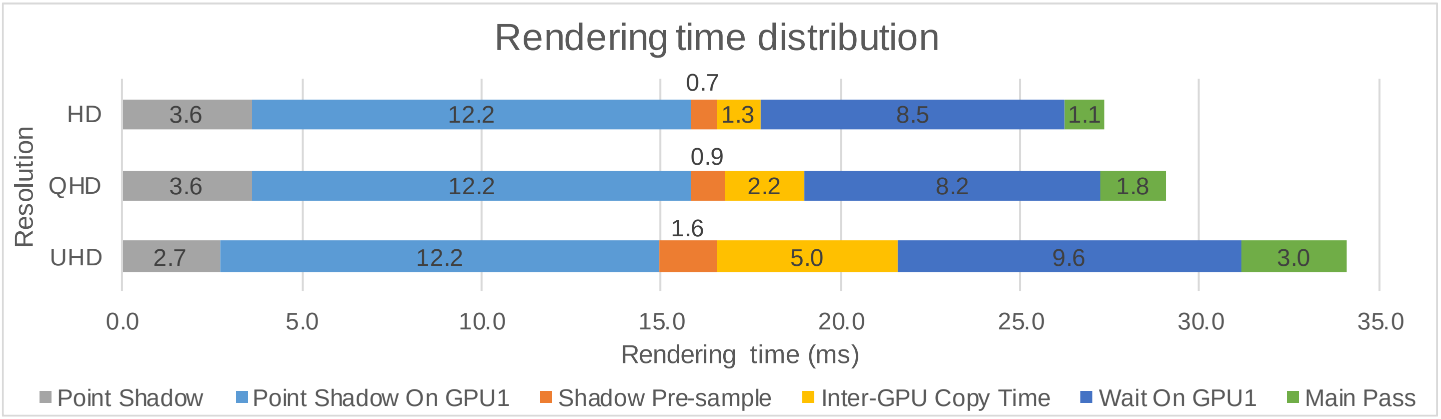

*Figure 8: Rendering time distribution for multi-GPU shadow mapping with two lights on GPU 1*

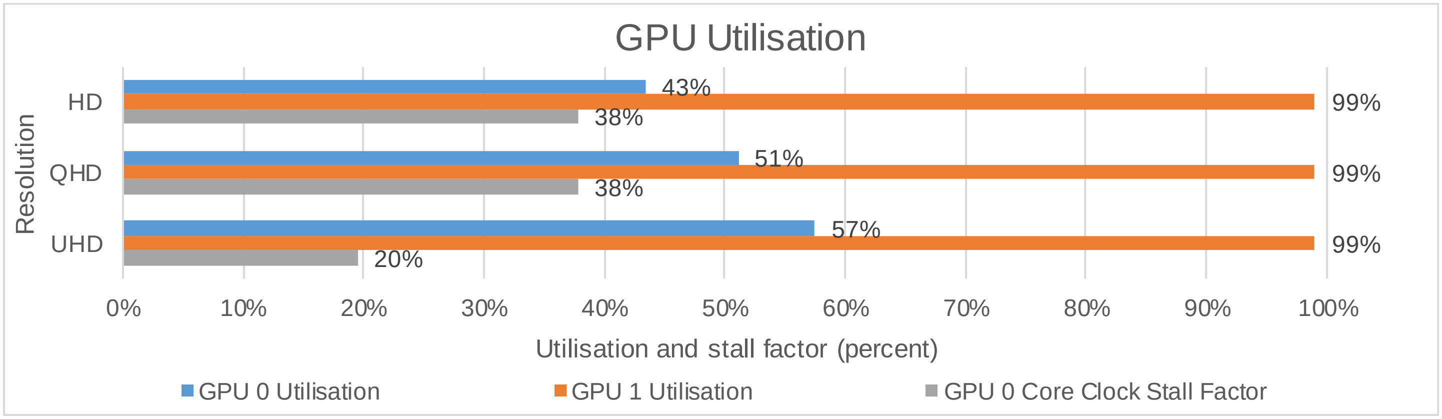

*Figure 9: GPU utilisation for multi-GPU shadow mapping with two lights on GPU 1*

| **Metric** | **HD** | **QHD** | **UHD** |
|---|---|---|---|
| Inter-GPU transfer size (MB) | 0.415 | 0.737 | 1.659 |
| GPU 0 clock speed (MHz) | 1228 | 1228 | 1589 |
| GPU 1 clock speed (MHz) | 1137 | 1137 | 1137 |

*Table 5: Clock speed and Inter-GPU transfer size for multi-GPU shadow mapping with two lights on GPU 1*

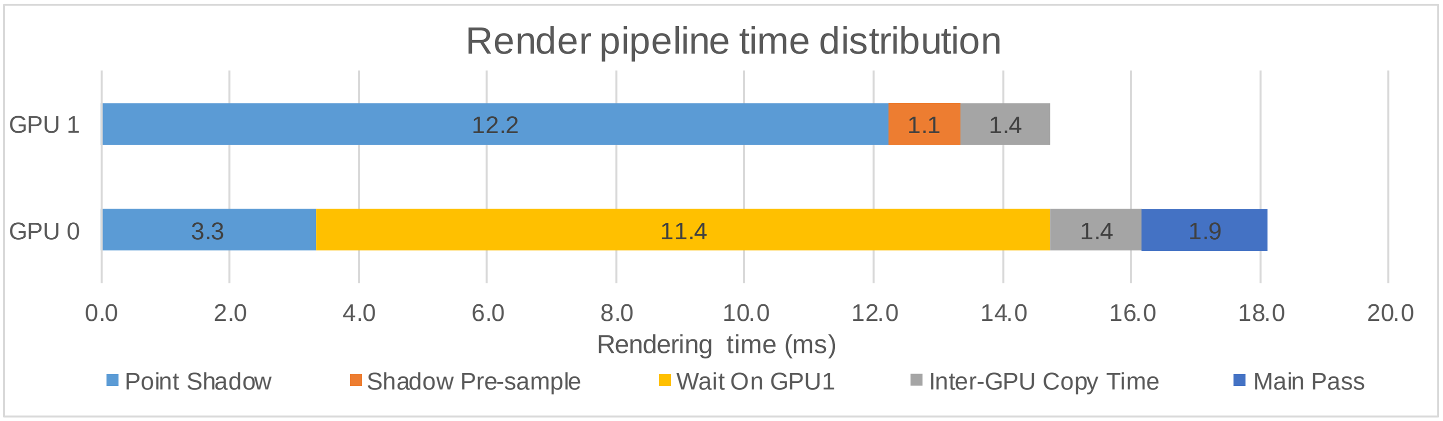

*Figure 10: Render pipeline time distribution for multi-GPU shadow mapping with two lights on GPU 1*

#### 8.3.2.2 Analysis

This variation displays worse overall framerates compared to 8.3.1 due to the increased time spent on GPU 1 rendering shadow maps. The time spent transferring data is still short, so this variation is not PCIe limited. Due to the time that GPU 0 is idle it throttles down causing worse overall performance. At UHD the increased pixel shading work causes GPU 0 to have slightly better utilisation and so it throttles less.

### 8.3.3 Multi-GPU shadow mapping with three lights on GPU 1

For this variation, three lights' shadow maps are computed on GPU 1 and one on GPU 0. The pre-sample buffer is sized for four lights due to texture formatting limitations (see 6.1.3).

#### 8.3.3.1 Results

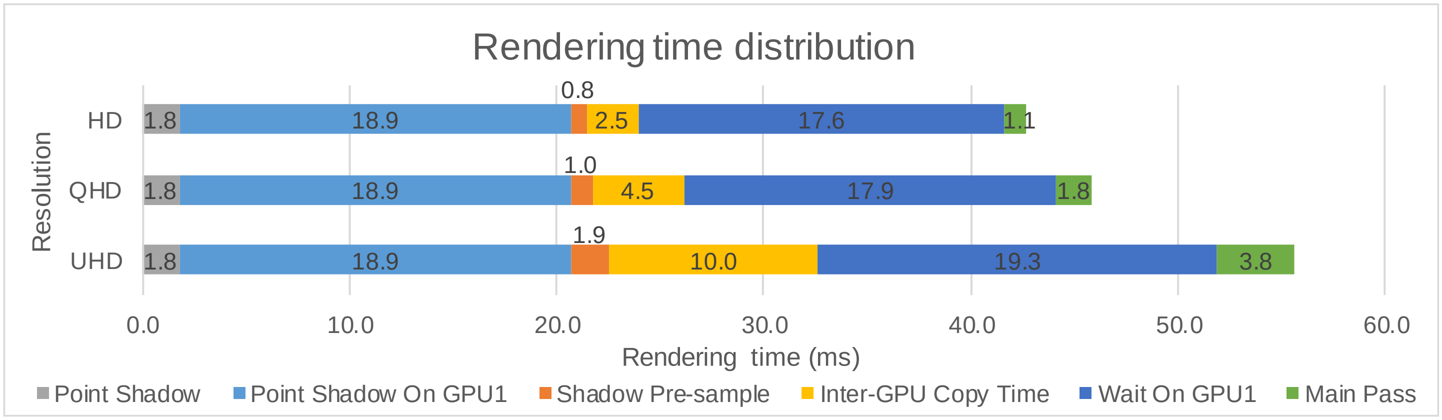

*Figure 11: Rendering time distribution for multi-GPU shadow mapping with three lights on GPU 1*

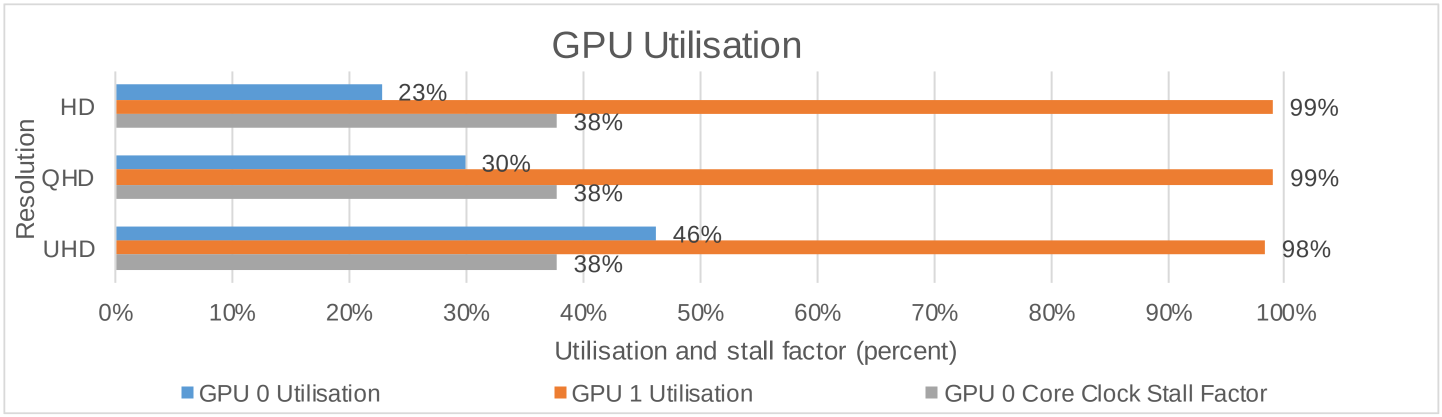

*Figure 12: GPU utilisation for multi-GPU shadow mapping with three lights on GPU 1*

| **Metric** | **HD** | **QHD** | **UHD** |
|---|---|---|---|
| Inter-GPU transfer size (MB) | 0.829 | 1.475 | 3.318 |
| GPU 0 clock speed (MHz) | 1228 | 1228 | 1228 |
| GPU 1 clock speed (MHz) | 1137 | 1137 | 1137 |

*Table 6: Clock speed and Inter-GPU transfer size for multi-GPU shadow mapping with three lights on GPU 1*

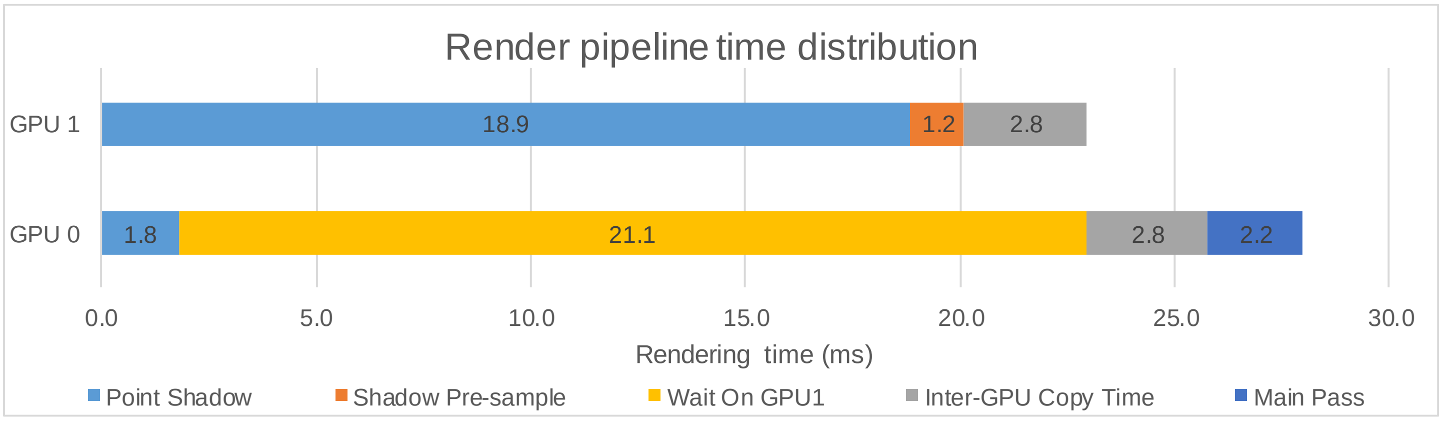

*Figure 13: Render pipeline time distribution for multi-GPU shadow mapping with three lights on GPU 1*

#### 8.3.3.2 Analysis

This variation displays worse overall framerates compared to 8.3.2 due to the increased time spent on GPU 1 rendering shadow maps. The time spent transferring data is significant and is contributing to the overall poor utilisation. Again GPU 0 throttles down and UHD has slightly better utilisation, but it is still very poor and throttles heavily.

### 8.3.4 Multi-GPU shadow mapping with four lights on GPU 1

For this variation, all four lights' shadow maps are computed on GPU 1 and none on GPU 0. The pre-sample buffer is sized for four lights.

#### 8.3.4.1 Results

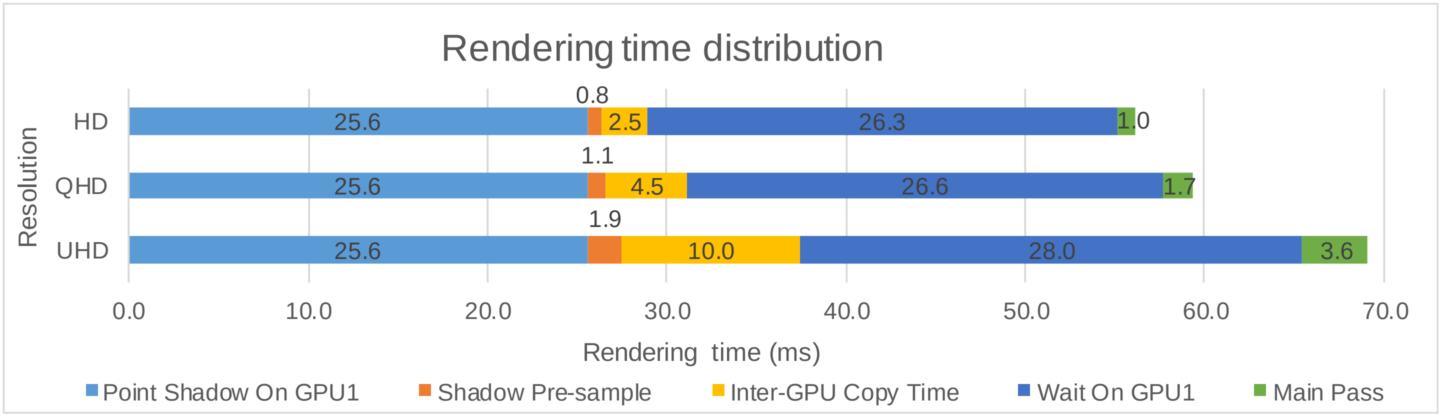

*Figure 14: Rendering time distribution for multi-GPU shadow mapping with four lights on GPU 1*

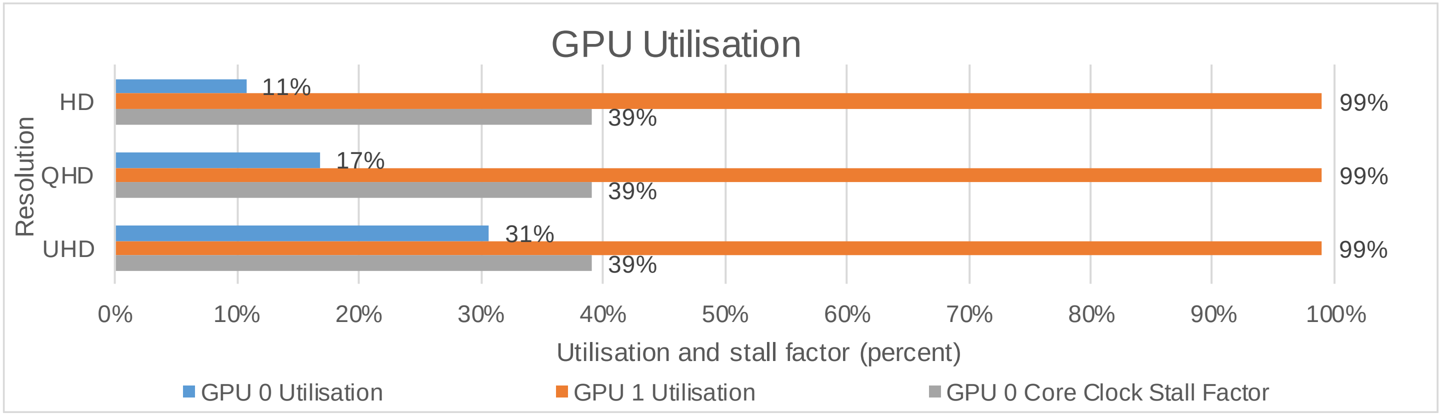

*Figure 15: GPU utilisation for multi-GPU shadow mapping with four lights on GPU 1*

| **Metric** | **HD** | **QHD** | **UHD** |
|---|---|---|---|
| Inter-GPU transfer size (MB) | 0.829 | 1.475 | 3.318 |
| GPU 0 clock speed (MHz) | 1202 | 1202 | 1202 |
| GPU 1 clock speed (MHz) | 1137 | 1137 | 1137 |

*Table 7: Clock speed and Inter-GPU transfer size for multi-GPU shadow mapping with four lights on GPU 1*

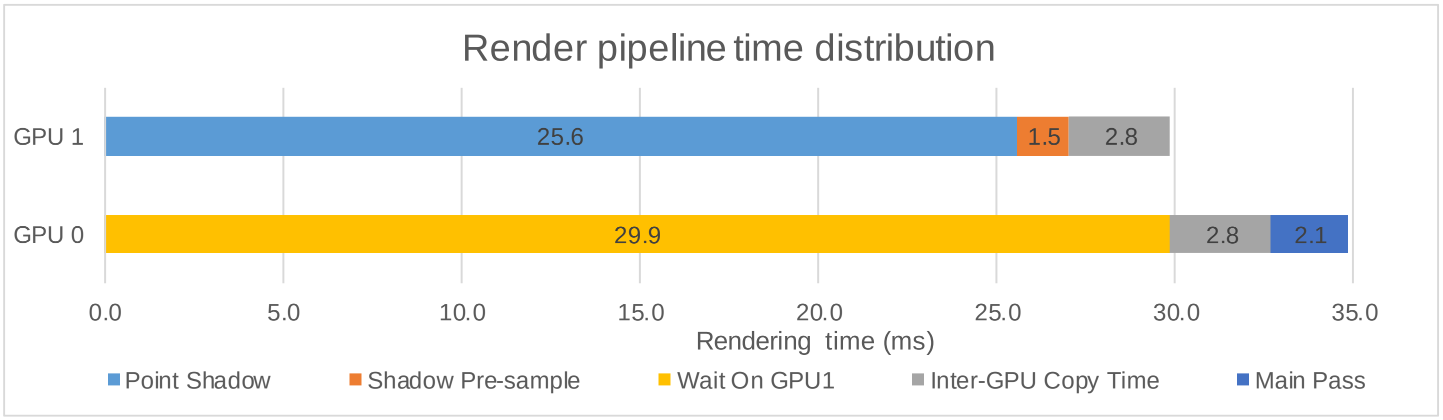

*Figure 16: Render pipeline time distribution for multi-GPU shadow mapping with four lights on GPU 1*

#### 8.3.4.2 Results analysis

This variation displays worse overall framerates compared to 8.3.3 due to the lengthy time spent on GPU 1 rendering shadow maps. Again GPU 0 throttles down and UHD has slightly better utilisation, but it is still very poor and throttles heavily. This variation is expected to perform badly as the lesser GPU (GPU 1) is performing most of the work.

### 8.3.5 Technique conclusion

#### 8.3.5.1 HD

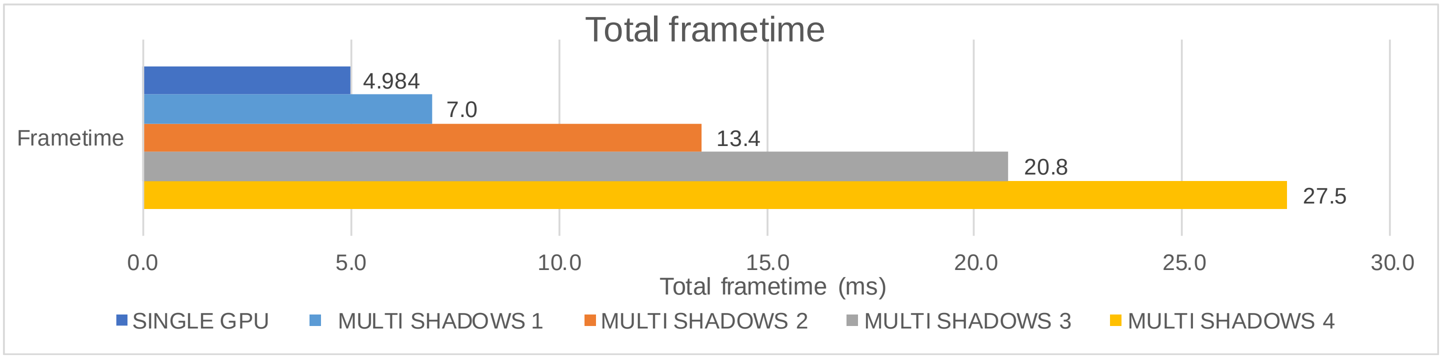

*Figure 17: Overall frametimes for multi-GPU shadow mapping at HD*

At HD, this technique does not provide any improvement over single GPU operation. This is due to the time spent rendering shadow maps on GPU 1 and transferring the data is longer than the time GPU 0 takes to render the other shadow maps. With the more lights added to GPU 1 the longer this time gets leading to GPU 0 stalling and throttling heavily.

#### 8.3.5.2 QHD

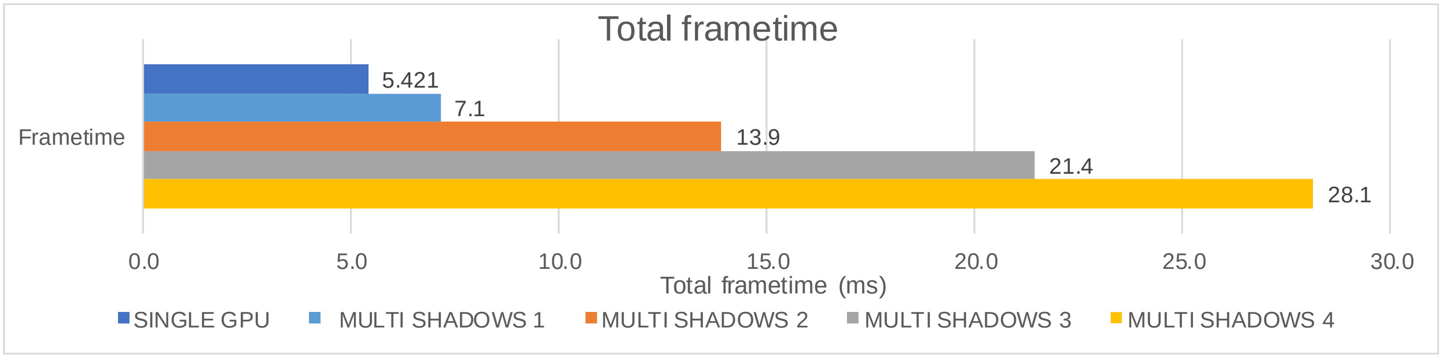

*Figure 18: Overall frametimes for multi-GPU shadow mapping at QHD*

QHD is very similar to HD with no improvement and overall worse frametimes with the more shadow maps rendered on GPU 1. The PCIe bus is starting to become a limiting factor due to the increased pixel count causing larger transfer sizes.

#### 8.3.5.3 UHD

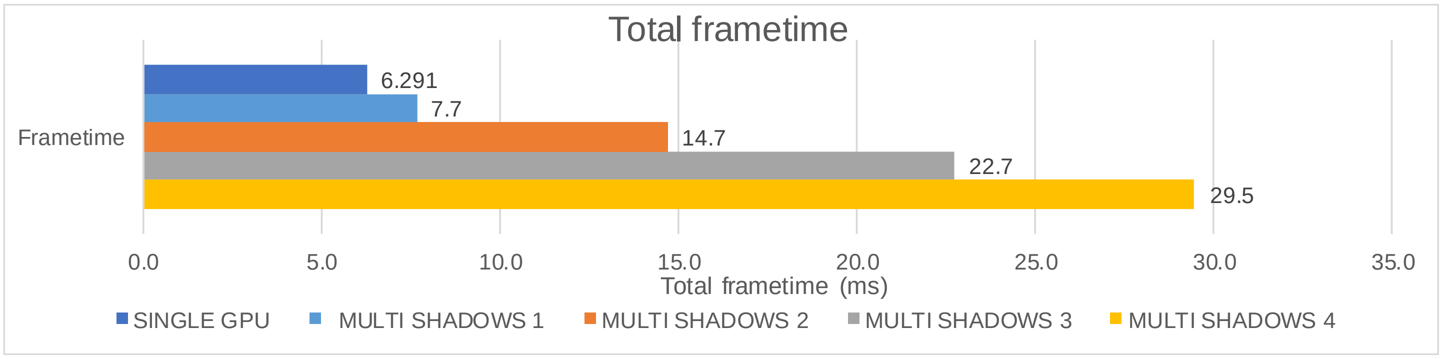

*Figure 19: Overall frametimes for multi-GPU shadow mapping at UHD*

UHD is even slower than QHD due to the increased resolution; this is because of the much longer transfer times of up to 10ms which causes poor overall utilisation of GPU 0 and very long frametimes.

#### 8.3.5.4 Overall Conclusion

Multi-GPU shadow mapping in the test system does not provide any improvement to frametimes compared to single GPU operation. However, with a more powerful GPU 1, it could. Any GPU that is fast enough to render a shadow map and transfer it in less time than GPU 0 takes to render the other shadow maps would provide an improvement over single GPU operation. At higher resolutions the PCIe bus is limiting frametimes so using compression on the shadow data would be very beneficial at the cost of quality.

## 8.4 ASYNCHRONOUS SHADOW MAPPING

This section contains the results and analysis for asynchronous shadow mapping (see 6.1.4). The number of point light shadows calculated on GPU 1 was the variable in this data set.

### 8.4.1 Asynchronous shadow mapping with one light on GPU 1

In this scenario, one light's shadow map is rendered on GPU 1 and three lights are rendered on GPU 0. The shadow buffer is sized so that only single light's data is stored to save transfer time.

#### 8.4.1.1 Results

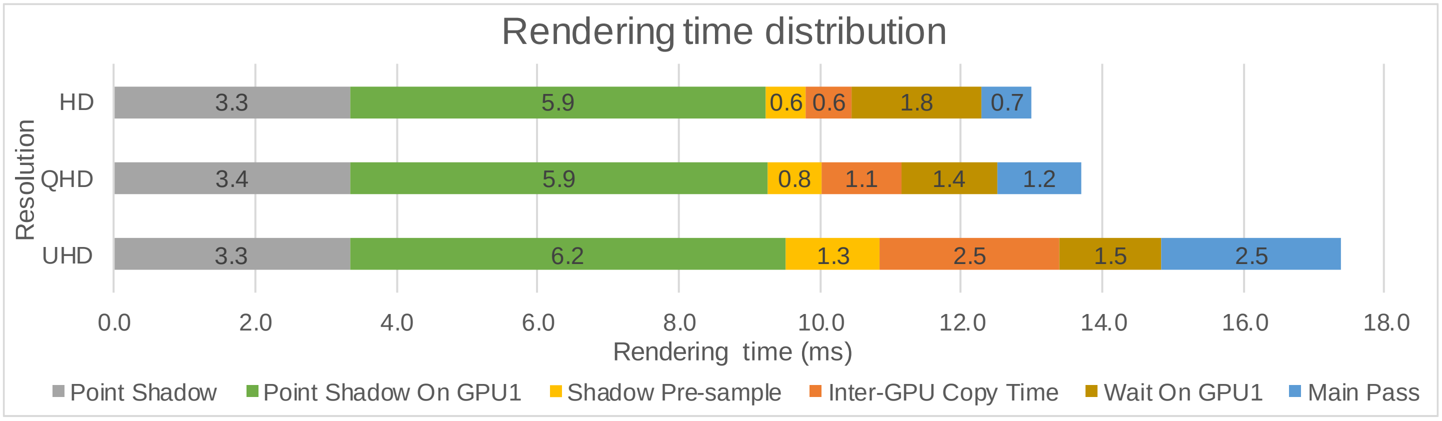

*Figure 20: Rendering time distribution for asynchronous shadows with one light on GPU 1*

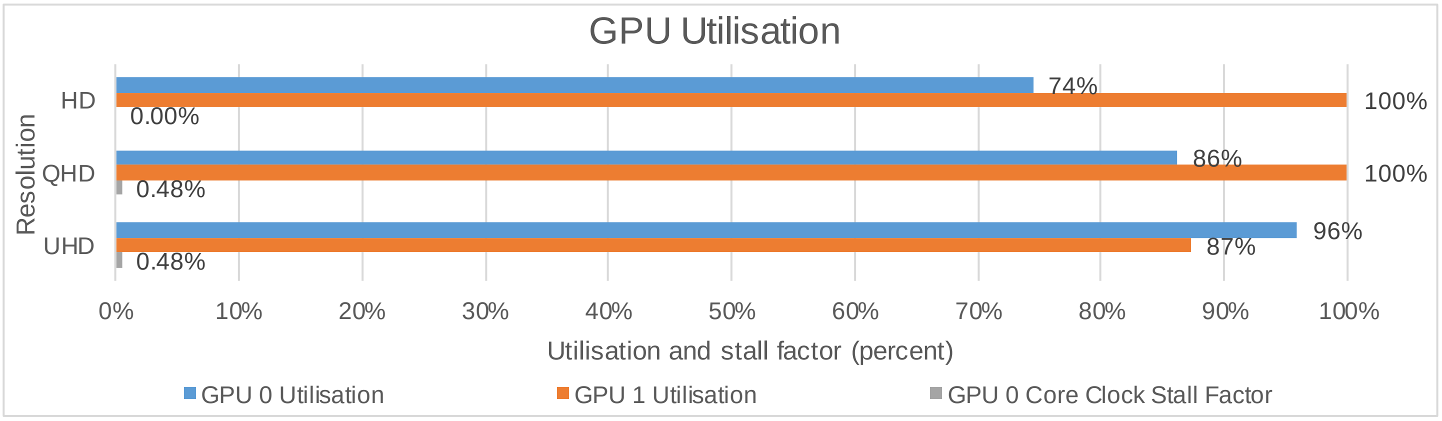

*Figure 21: GPU utilisations for asynchronous shadow mapping with one light on GPU 1*

| **Metric** | **HD** | **QHD** | **UHD** |
|---|---|---|---|
| Inter-GPU transfer size (MB) | 0.21 | 0.37 | 0.83 |
| GPU 0 clock speed (MHz) | 1971 | 1962 | 1962 |
| GPU 1 clock speed (MHz) | 1137 | 1137 | 1137 |

*Table 8: Data transfer size and clock speed for asynchronous shadow mapping with one light on GPU 1*

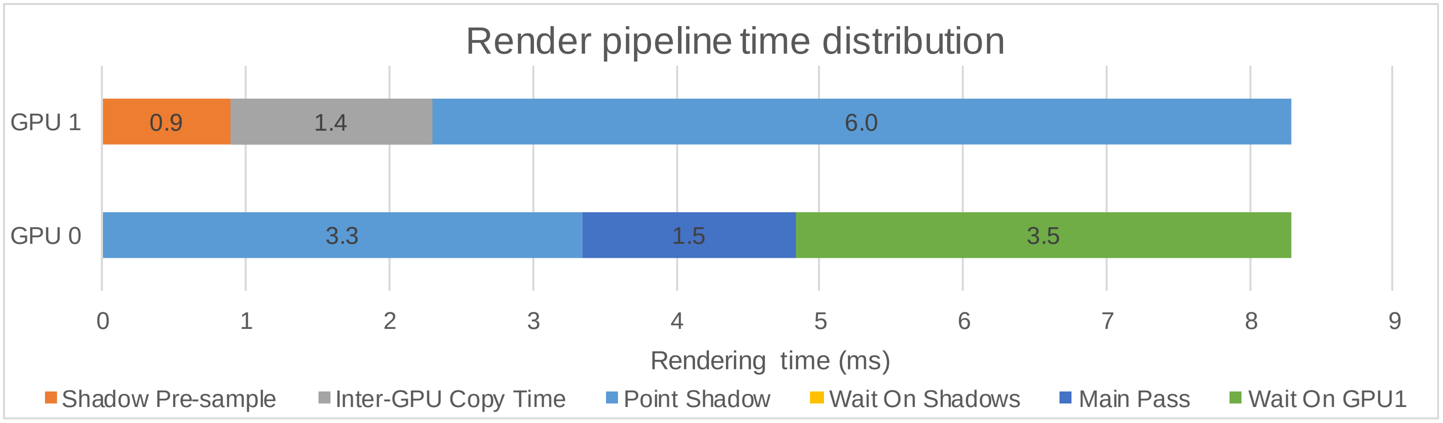

*Figure 22: Render pipeline time distribution for asynchronous shadow mapping with one light on GPU 1*

#### 8.4.1.2 Analysis

This variation does not provide an improvement to frametimes versus single GPU operation as GPU 0 is waiting for almost 3.5 ms on GPU 1 to finish rendering shadow maps. The time spent transferring data is short, so this variation is not PCIe limited. Due to the short time that GPU 0 is idle it does not throttle however utilisation at HD is low showing a loss in performance. The increase of transfer time at UHD causes GPU 1 to throttle down causing the longer shadow render time. The variation in stall factor between 0 and 0.48% is due to GPU boost reacting to the start of the benchmark, as it then settles during the higher resolution benchmarks. As it is less than 1% it is within normal deviation for clock speeds with GPU boost.

### 8.4.2 Asynchronous shadow mapping with two lights on GPU 1

In this scenario two lights shadow maps are computed on GPU 1 and two are computed on GPU 0. The shadow buffer is sized so that two lights' data is stored.

#### 8.4.2.1 Results

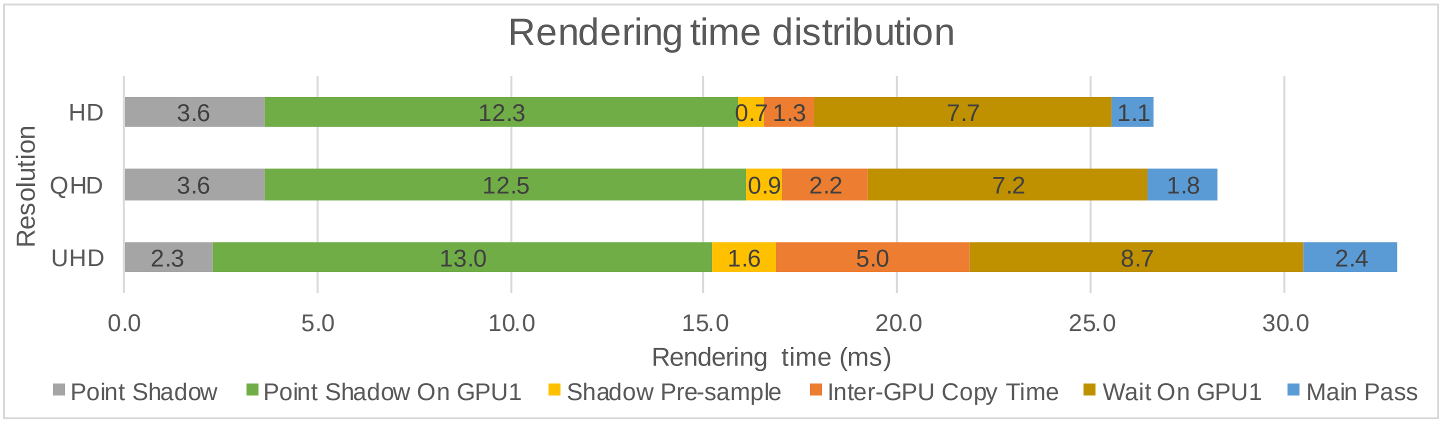

*Figure 23: Rendering time distribution for asynchronous shadow mapping with two lights on GPU 1*

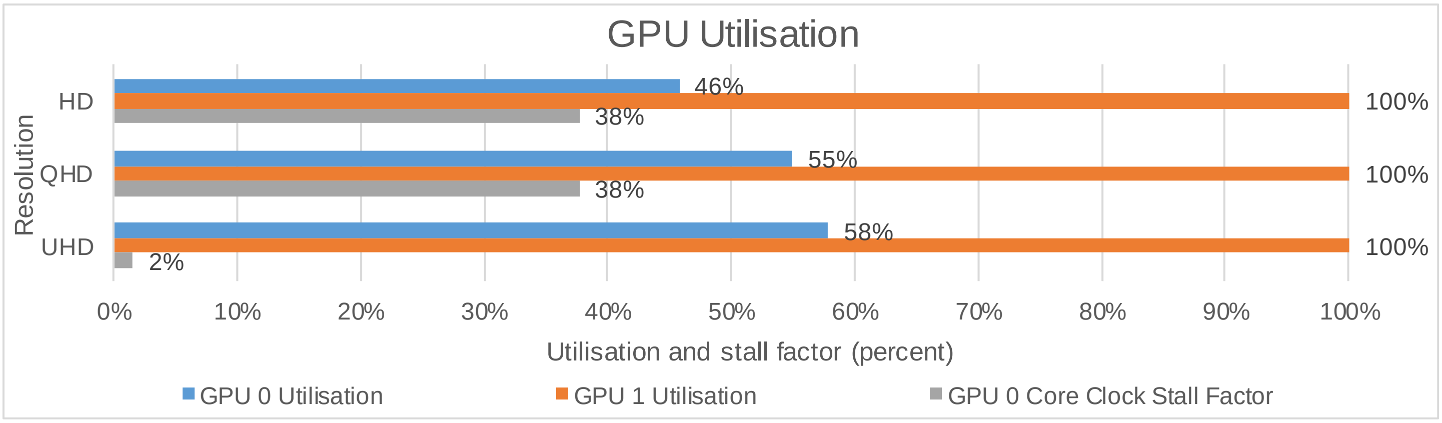

*Figure 24: GPU utilisation for asynchronous shadow mapping with two lights on GPU 1*

| Metric | HD | QHD | UHD |
|---|---|---|---|
| Inter-GPU transfer size (MB) | 0.415 | 0.737 | 1.659 |
| GPU 0 clock speed (MHz) | 1228 | 1228 | 1941 |
| GPU 1 clock speed (MHz) | 1137 | 1137 | 1137 |

*Table 9: Data transfer size and clock speed for asynchronous shadow mapping with two lights on GPU 1*

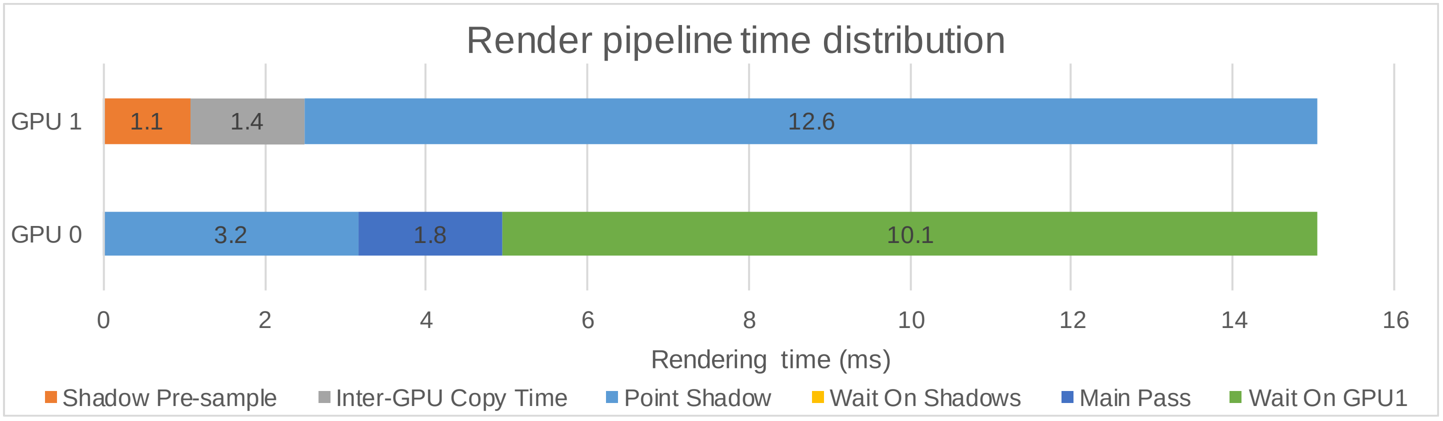

*Figure 25: Render pipeline time distribution for asynchronous shadow mapping with two lights on GPU 1*

#### 8.4.2.2 Analysis

This variation is slower than 8.4.1 as almost 12 ms is spent waiting on GPU 1 to finish rendering shadow maps. This time causes GPU to throttle down at HD and QHD, but not a UHD due to the higher pixel count. The time spent transferring data is still short, so this variation is not PCIe limited.

### 8.4.3 Asynchronous shadow mapping with three lights on GPU 1

In this scenario three light’s shadow maps are rendered on GPU 1 and one is rendered on GPU 0. The shadow buffer is sized for four lights’ data, this limitation is outlined in 6.1.3.

#### 8.4.3.1 Results

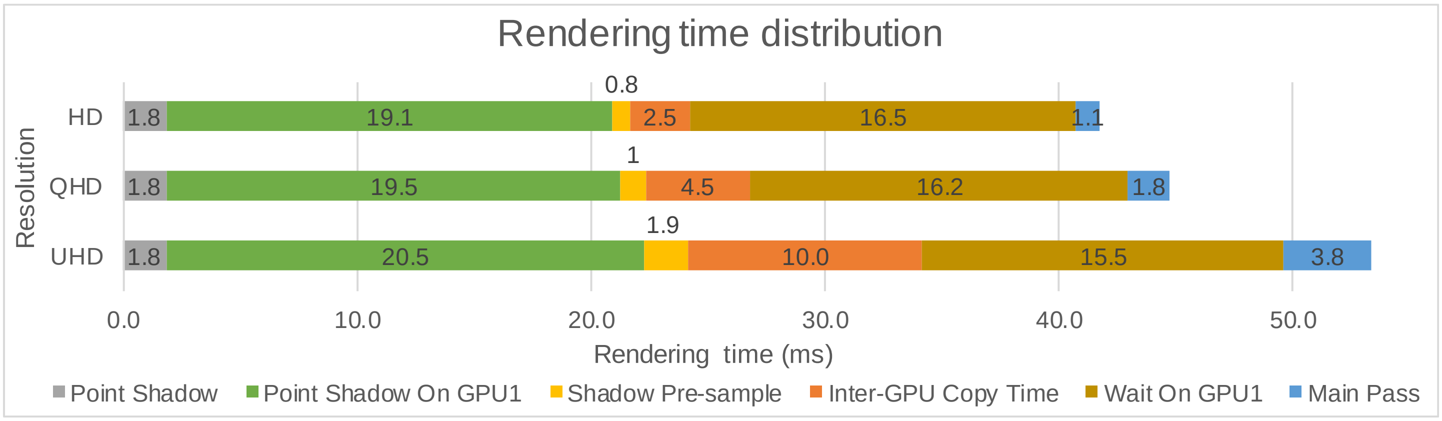

*Figure 26: Rendering time distrobution for asynchronous shadow mapping with three lights on GPU 1*

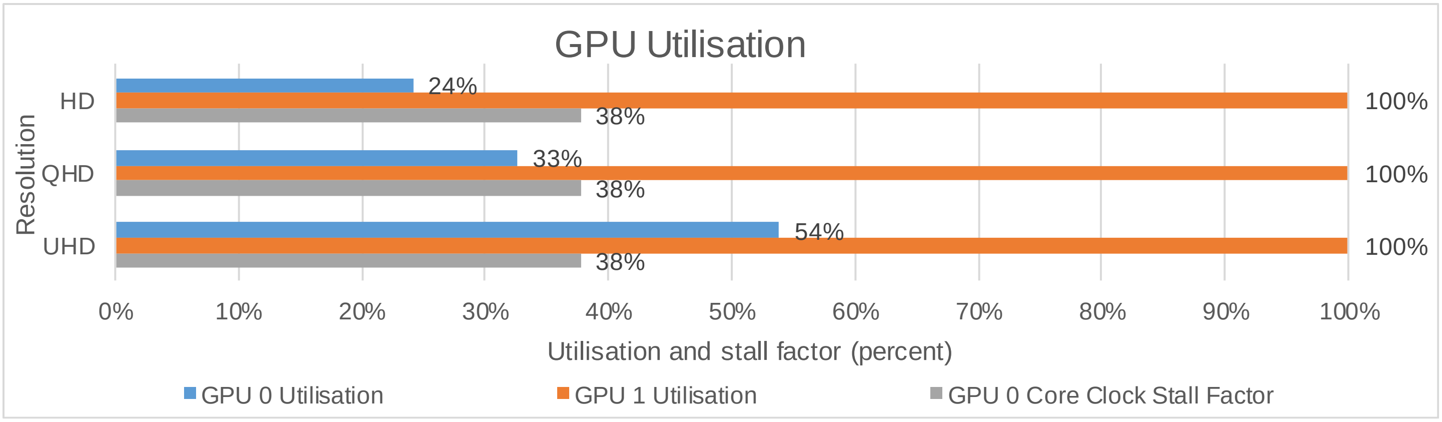

*Figure 27: GPU utilisation for asynchronous shadow mapping with three lights on GPU 1*

| **Metric** | **HD** | **QHD** | **UHD** |
|---|---|---|---|
| Inter-GPU transfer size (MB) | 0.829 | 1.475 | 3.318 |
| GPU 0 clock speed (MHz) | 1228 | 1228 | 1228 |
| GPU 1 clock speed (MHz) | 1137 | 1137 | 1137 |

*Table 10: Data transfer size and clock speed for asynchronous shadow mapping with three lights on GPU 1*

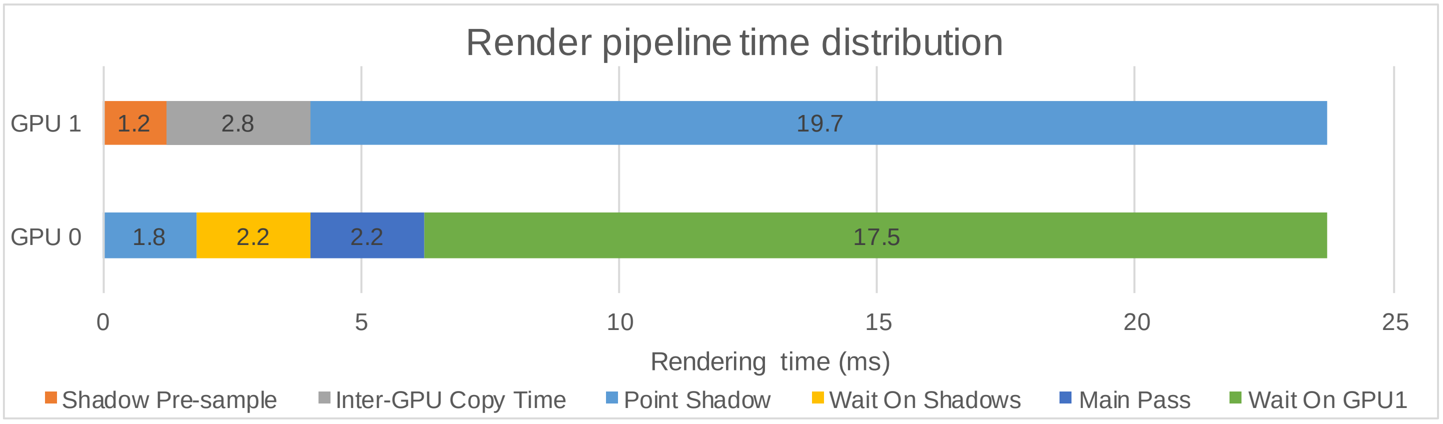

*Figure 28: Render pipeline time distribution for asynchronous shadow mapping with three lights on GPU 1*

#### 8.4.3.2 Analysis

This variation is slower than 8.2.2 as almost 18 ms is spent waiting on GPU 1 to finish rendering shadow maps. This time causes GPU to throttle down and have very poor utilisation. Due to a formatting limitation (see 6.1.3) the data transferred is the same as four lights. This size causes lengthy transfer times that reduce overall frametimes.

### 8.4.4 Asynchronous shadow mapping with four lights on GPU 1

In this scenario four lights shadow maps are computed on GPU 1 and zero on GPU 0. The shadow buffer is sized to fit four lights' data.

#### 8.4.4.1 Results

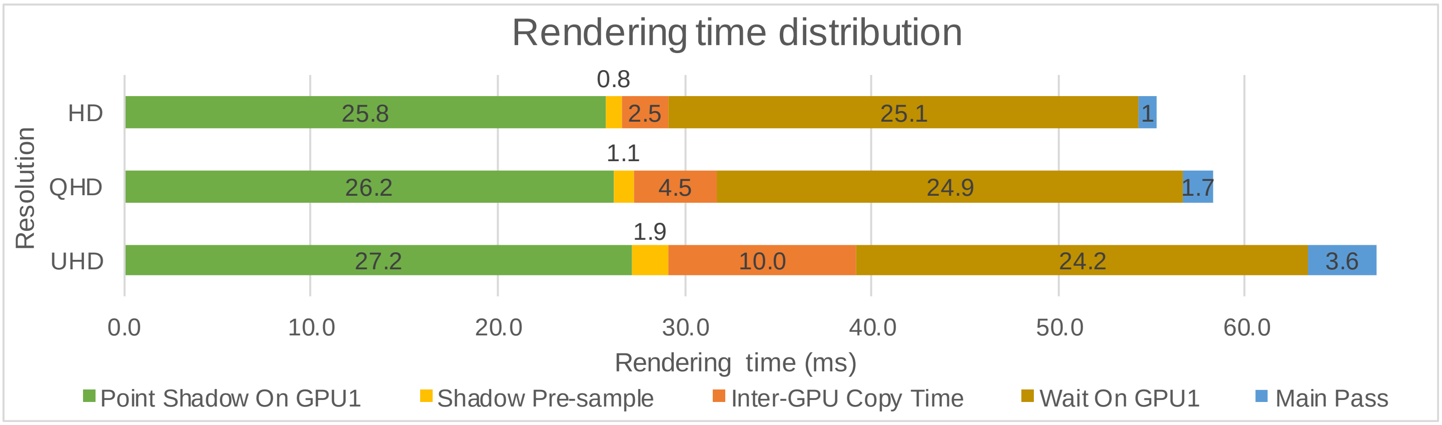

*Figure 29: Rendering time distrobution for asynchronous shadow mapping with four lights on GPU 1*

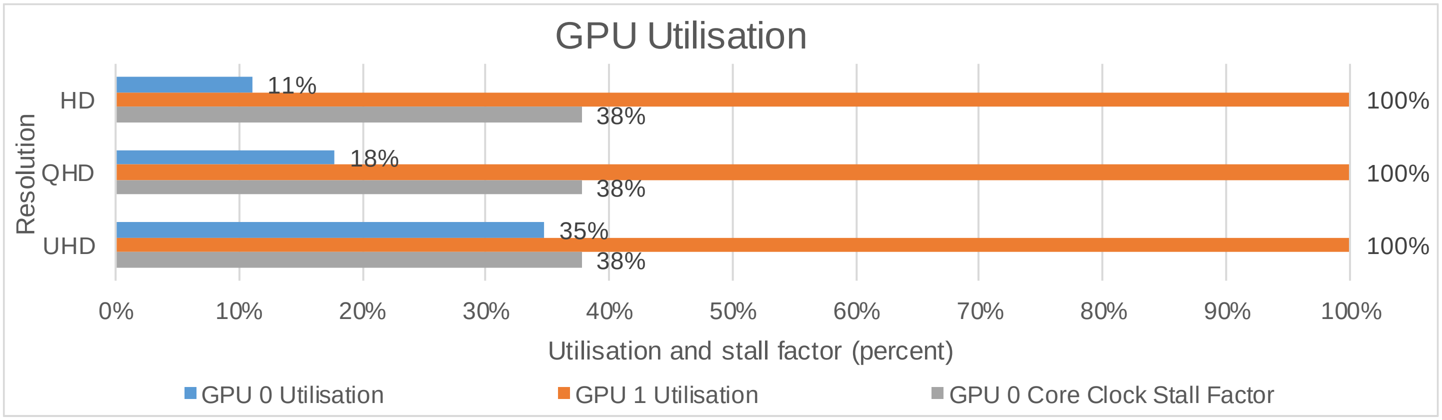

*Figure 30: GPU utilisation for asynchronous shadow mapping with four lights on GPU 1*

| **Metric** | **HD** | **QHD** | **UHD** |
|---|---|---|---|
| Inter-GPU transfer size (MB) | 0.829 | 1.475 | 3.318 |
| GPU 0 clock speed (MHz) | 1228 | 1228 | 1228 |
| GPU 1 clock speed (MHz) | 1137 | 1137 | 1137 |

*Table 11: Data transfer size and clock speed for asynchronous shadow mapping with four lights on GPU 1*

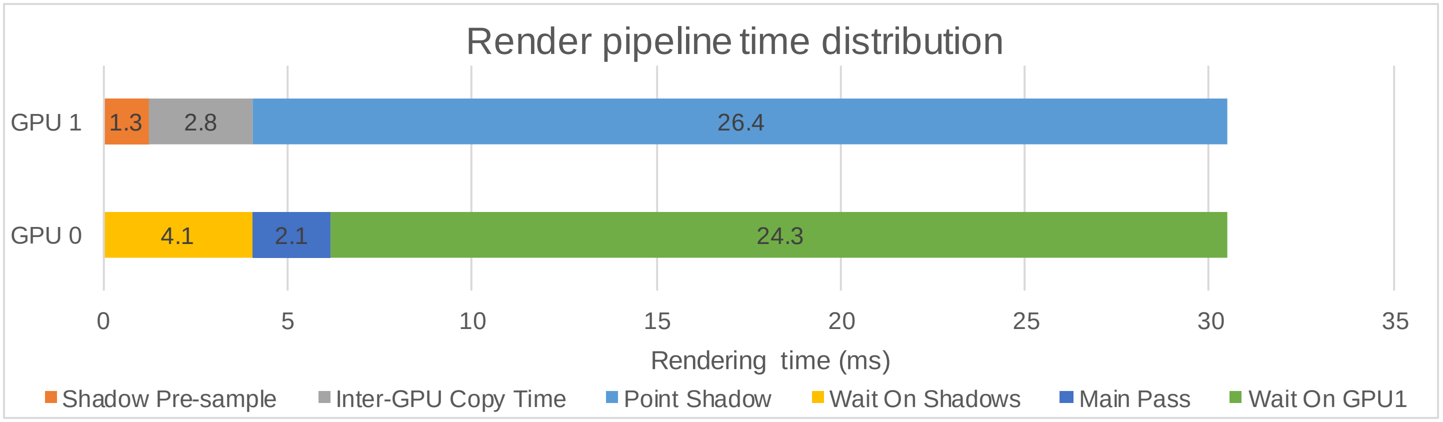

*Figure 31: Render pipeline time distribution for asynchronous shadow mapping with four lights on GPU 1*

#### 8.4.4.2 Analysis

This variation is slower than 8.2.3 as almost 25 ms are spent waiting on GPU 1 to finish rendering shadow maps. This time causes GPU to throttle down and have poor utilisation. The size of the inter-GPU transfer causes lengthy transfer times that affect overall frametimes. This variation is expected to perform badly as the lesser GPU (GPU 1) is performing most of the work.

### 8.4.5 Technique Conclusion

#### 8.4.5.1 HD

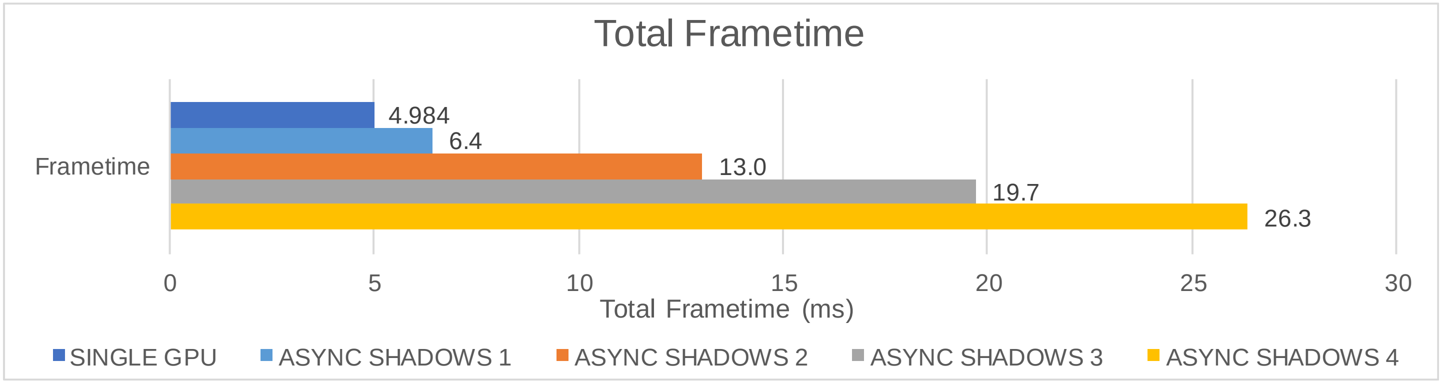

*Figure 32: Overall frametimes for asynchronous shadow mapping at HD*

At HD, asynchronous shadows should not be used as even one light being executed on GPU 1 leads to a loss in overall performance. This loss gets worse as more shadow maps are calculated on to GPU 1. In the single light configuration, the time wait time is not that long, so GPU 0 does not throttle. However, with two or more shadow maps being rendered on GPU 1, GPU 0 stalls causing severely decreased performance. This technique completely utilises GPU 1 with 100% usage recorded throughout all test scenarios.

#### 8.4.5.2 QHD

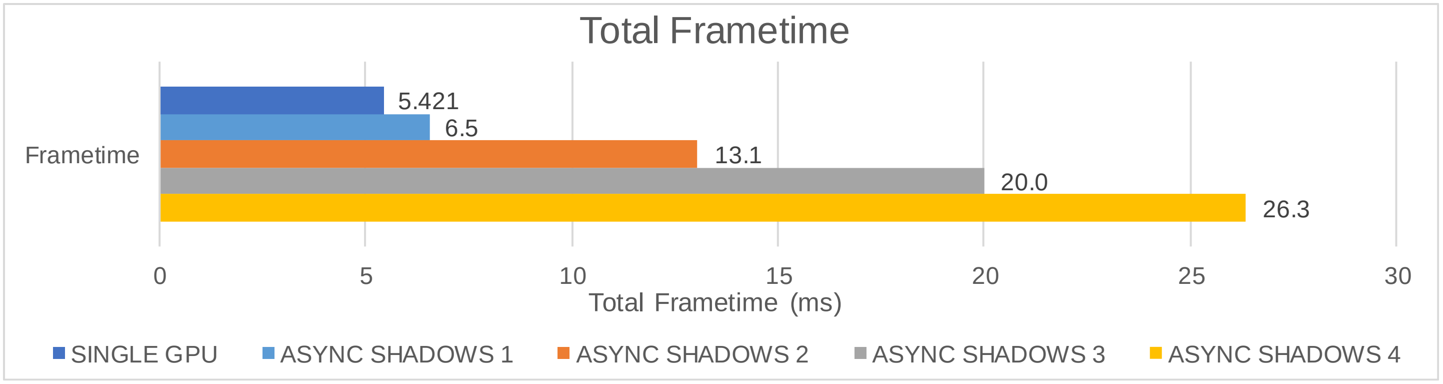

*Figure 33: Overall frametimes for asynchronous shadow mapping at QHD*

At QHD the outcome is very similar to HD with asynchronous shadow mapping not providing an improvement to overall frametimes. However, the gap between single GPU and asynchronous shadows is closer here (around 1.0 ms) as GPU 0 has more pixels to render, taking more time on the main pass allowing more time for GPU 1 to render the shadow maps. This is one of the strengths of this technique allowing work on GPU 0 to mask the time GPU 1 spends rendering shadows.

#### 8.4.5.3 UHD

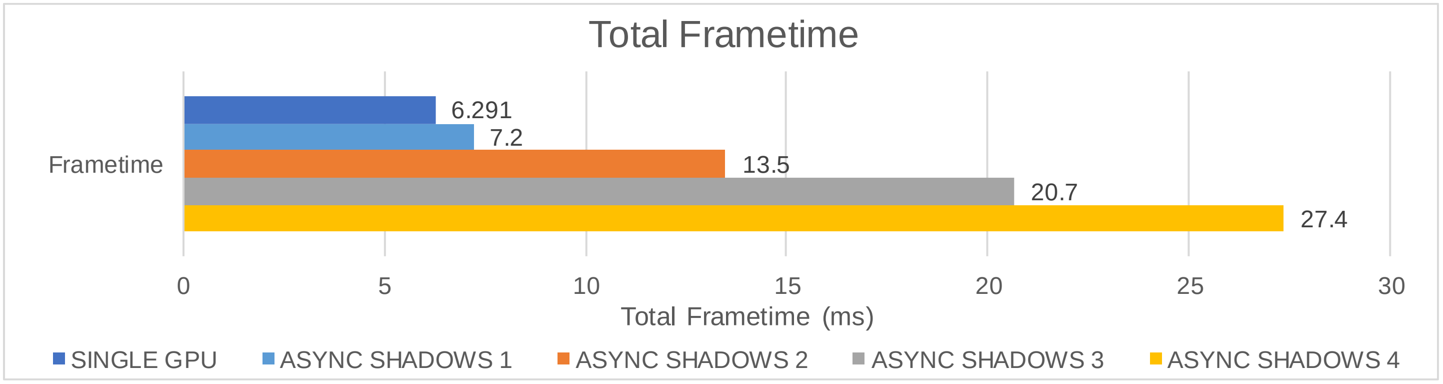

*Figure 34: Overall frametimes for asynchronous shadow mapping at UHD*

Again, at UHD the story is the same with no improvement to overall frame times. The gap against a single GPU is smaller than HD, due to the main pass taking slightly longer at the higher resolution giving more time to GPU 1 to calculate shadow maps. The gap between this technique and single GPU operation is the same due to the increase in transfer data size costing more than the increase in shading cost due to the higher resolution. Again GPU 1 is fully utilised thought the tests.

#### 8.4.5.4 Overall Conclusion

Asynchronous multi-GPU shadow mapping is an improvement to multi-GPU shadow mapping that provides a significant improvement to frametimes as the computations and transfer time of GPU 1 is hidden by the scene rendering on GPU 0. Unfortunately, in the test system this does not provide an improvement to frametimes over single GPU operation. However, with a faster GPU 1 (or a slower GPU 0) this technique could provide an improvement over single GPU operation. Although, at higher resolutions the PCIe bus transfer speed starts to bottleneck frametimes.

## 8.5 SPLIT FRAME RENDERING

This section contains the results and analysis for Split Frame Rendering (see 6.1.5). The variable changed is the ratio of the screen that is rendered on GPU 0.

### 8.5.1 SFR with a Ratio of 50%

For this technique, half (50%) of the screen is rendered with GPU 0 and half with GPU 1, the shadow maps are computed on both cards separately.

#### 8.5.1.1 Results

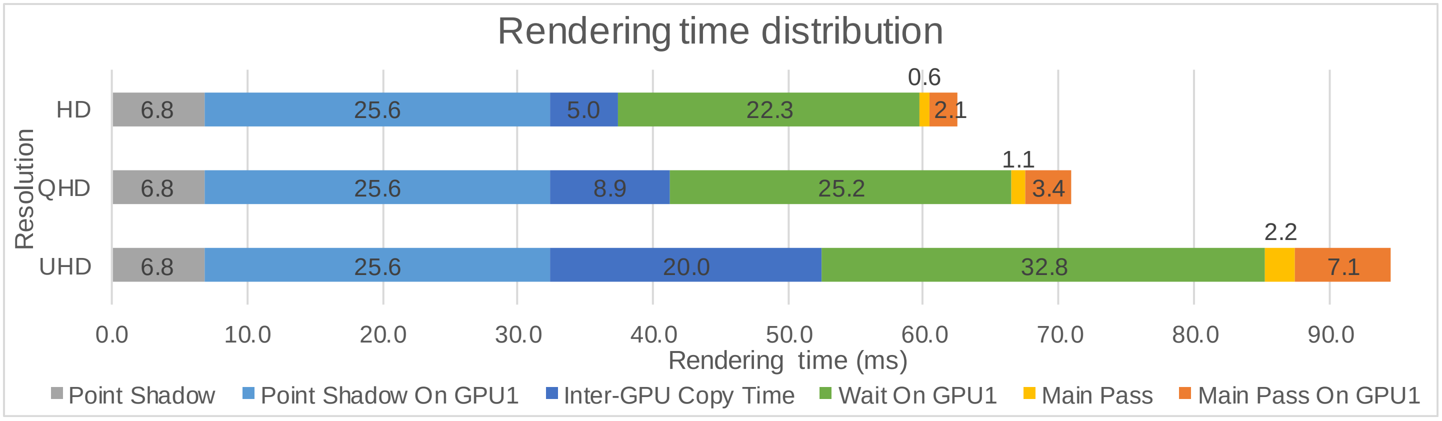

*Figure 35: Rendering time distribution for SFR with a ratio of 50%*

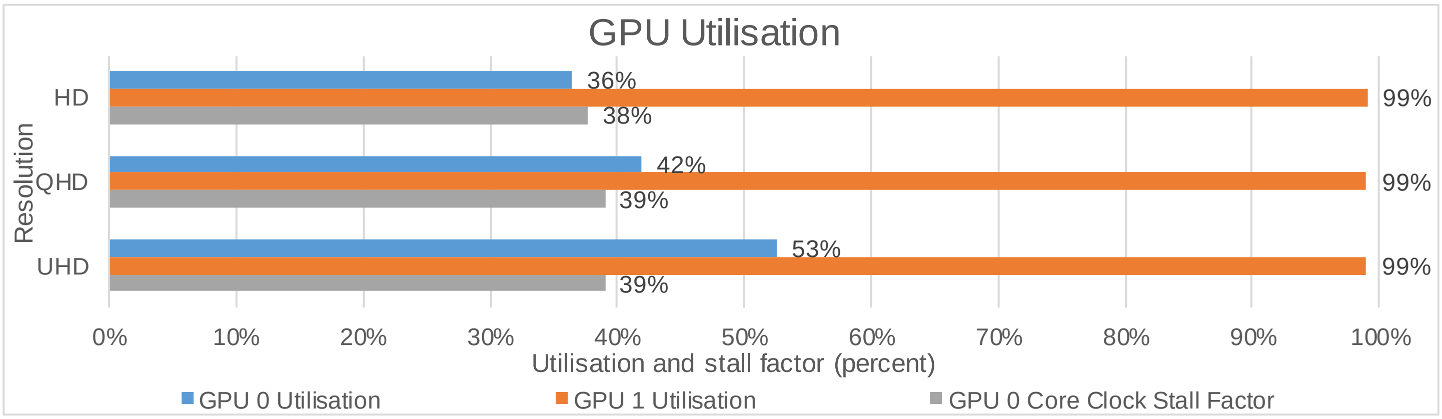

*Figure 36: GPU utilisation for SFR with a ratio of 50%*

| Metric | HD | QHD | UHD |
|---|---|---|---|
| Inter-GPU transfer size (MB) | 1.66 | 2.95 | 6.64 |
| GPU 0 clock speed (MHz) | 1228 | 1202 | 1202 |
| GPU 1 clock speed (MHz) | 1137 | 1137 | 1137 |

*Table 12: Data transfer size and clock speed for SFR with a ratio of 50%*

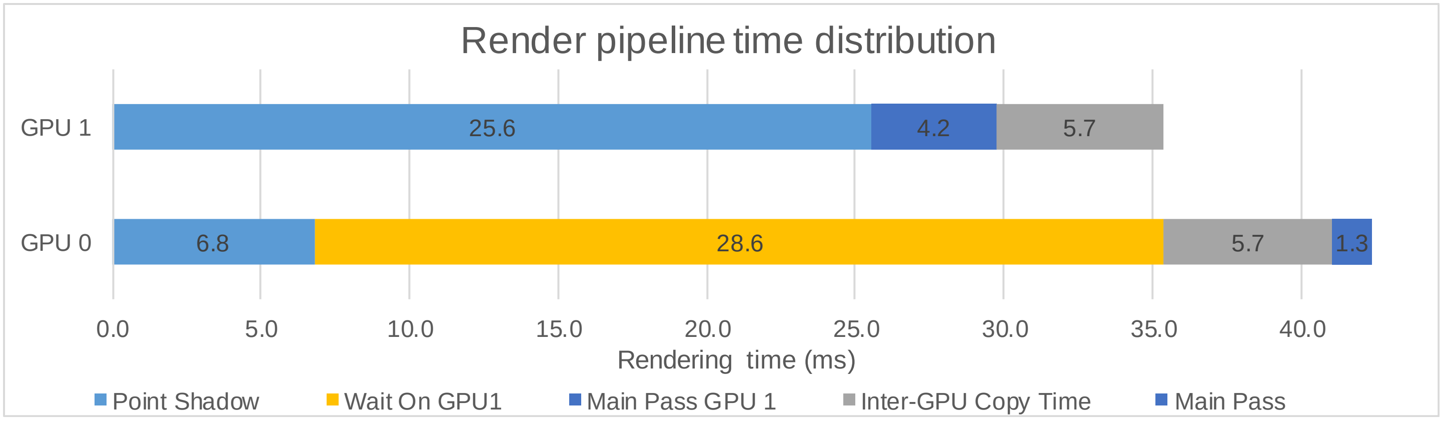

*Figure 37: Render pipeline time distribution for SFR with a ratio of 50%*

#### 8.5.1.2 Analysis

The results show that the overall frametimes are significantly longer than in single GPU mode. This is due to the time that GPU 1 takes to calculate the shadow maps being almost 3.75 times longer than GPU 0. This time difference causes significant stalling from GPU 0. The data that is transferred inter-GPU is large (6.6 MB) so the PCIe bus speed limits overall frametimes.

### 8.5.2 SFR Ratio with a ratio of 95%

For this technique, 95% the screen is rendered with GPU 0 and the other 5% with GPU 1, the shadow maps are computed on both cards separately.

#### 8.5.2.1 Results

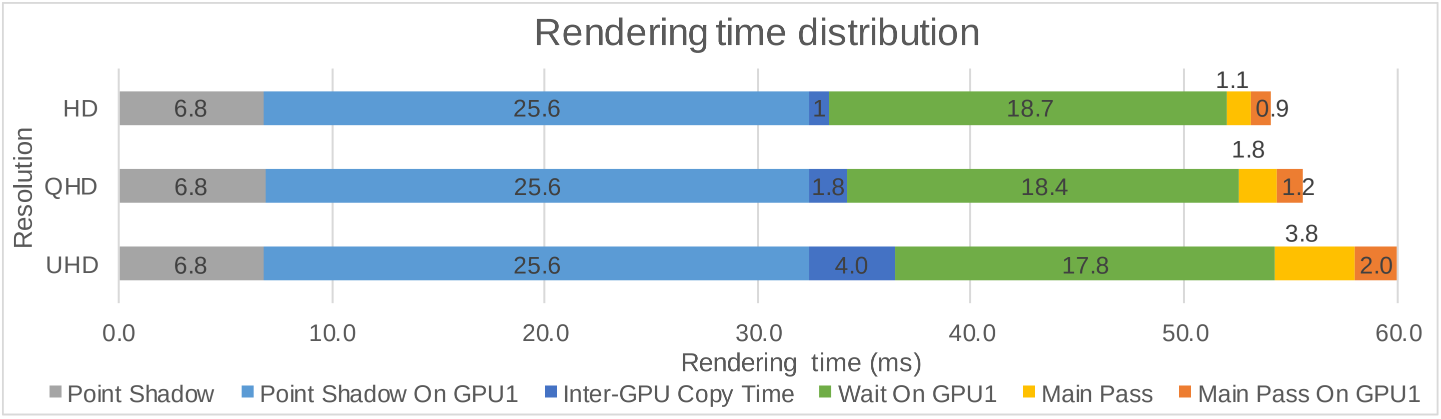

*Figure 38: Rendering time distribution for SFR with a ratio of 95%*

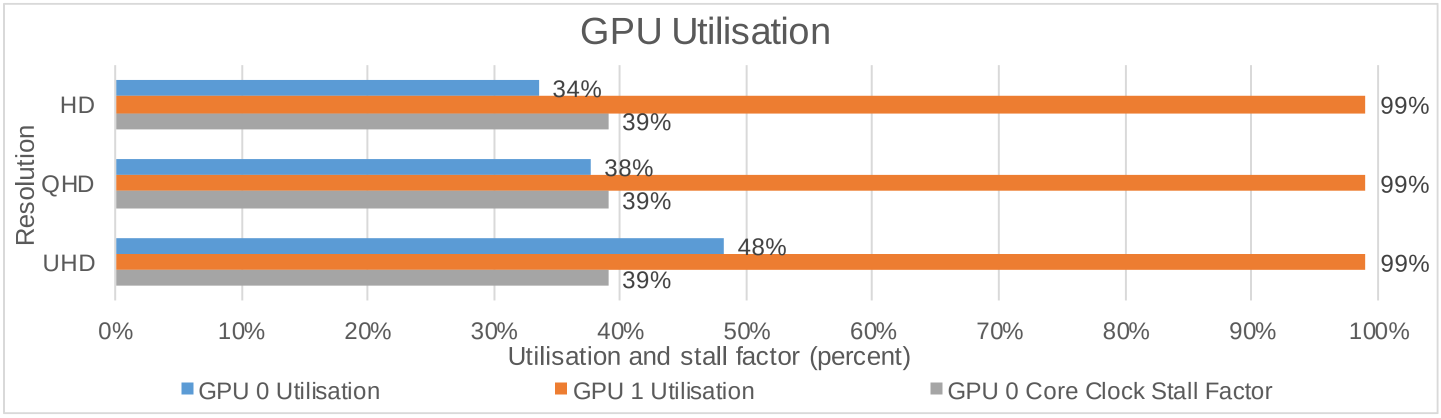

*Figure 39: GPU utilisation for SFR with a ratio of 95%*

| **Metric** | **HD** | **QHD** | **UHD** |
|---|---|---|---|
| Inter-GPU transfer size (MB) | 0.332 | 0.59 | 1.327 |
| GPU 0 clock speed (MHz) | 1202 | 1202 | 1202 |
| GPU 1 clock speed (MHz) | 1137 | 1137 | 1137 |

*Table 13: Data transfer size and clock speed for SFR with a ratio of 95%*

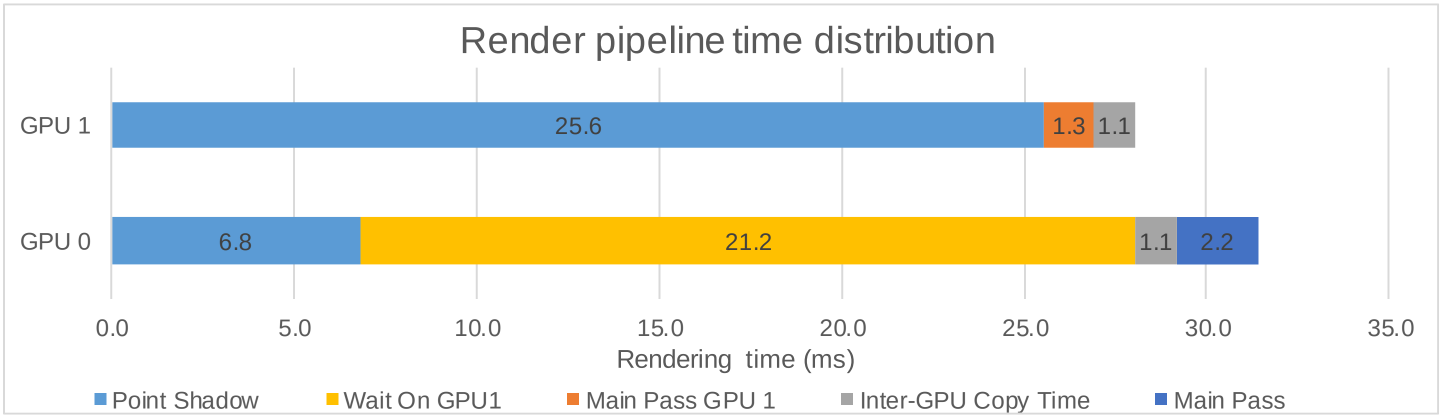

*Figure 40: Render pipeline time distribution for SFR with a ratio of 95%*

#### 8.5.2.2 Analysis

The results show that even at the maximum split towards GPU 0, the overall frametimes are significantly longer than in single GPU mode. This is due to the time that GPU 1 takes to render the shadow maps. This time difference causes significant stalling from GPU 0. Inter-GPU transfer time does not limit as the amount of data being transferred to GPU 0 is small, being only 5% of the image.

### 8.5.3 Technique conclusion

#### 8.5.3.1 HD

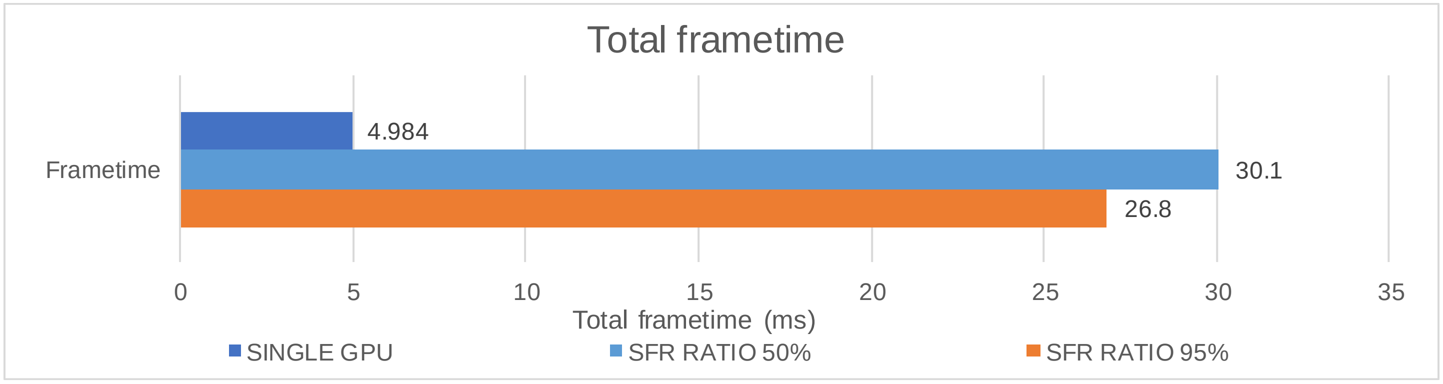

*Figure 41: Overall frametimes for SFR at HD*

At HD, this technique causes a large performance degradation due to the lower performance of GPU 1 compared to GPU 0. Split Frame Rendering favours more equally performing GPUs.

#### 8.5.3.2 QHD

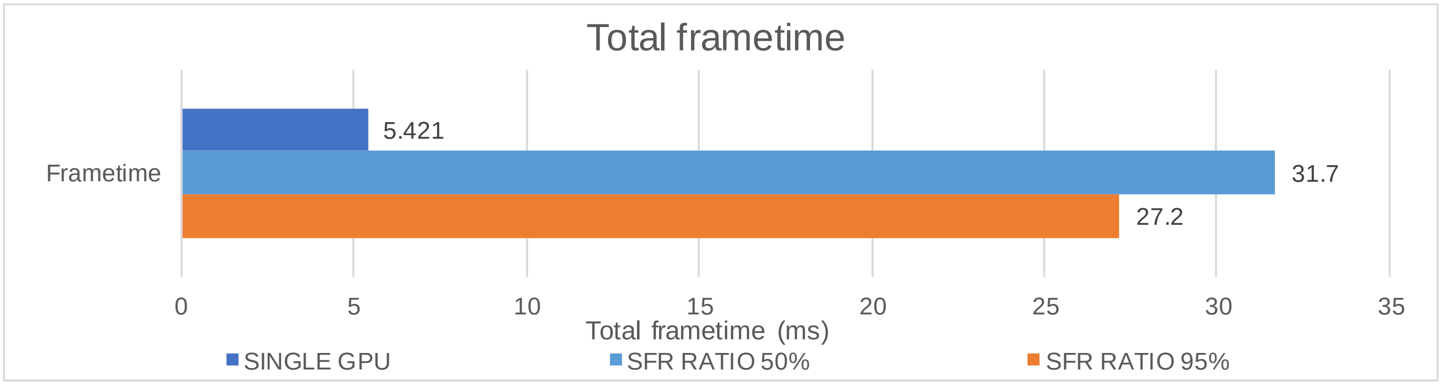

*Figure 42: Overall frametimes for SFR at QHD*

At QHD, the same is true as it was at HD. However, there is more data to transfer over the PCIe bus due to the higher resolution and more time is therefore spent copying data causing more GPU stalling to occur.

#### 8.5.3.3 UHD

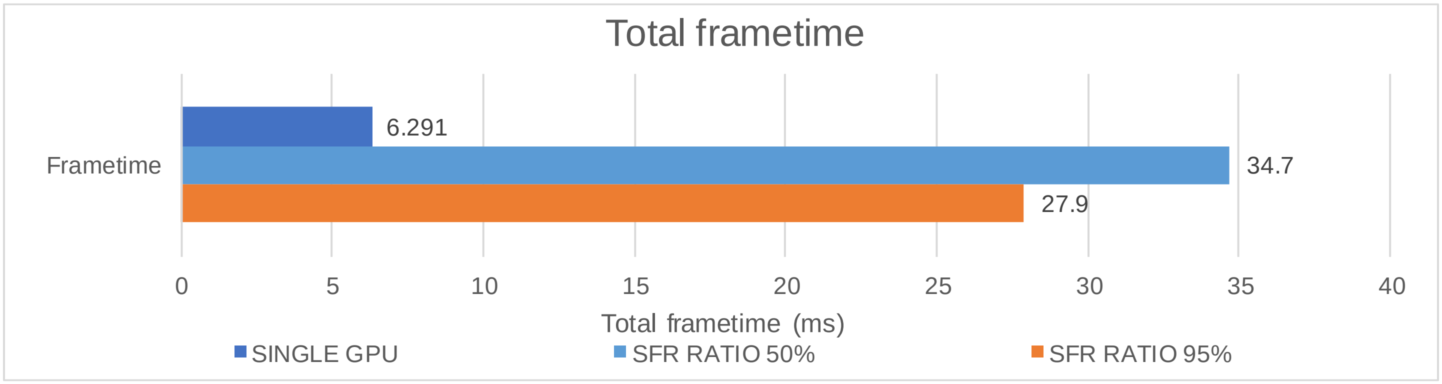

*Figure 43: Overall frametimes for SFR at UHD*

At UHD, the results get worse due to the larger amount of data to transfer for each frame. However overall utilisation is higher on GPU 0 due to the higher number of pixels to shade.

#### 8.5.3.4 Overall Conclusion

In the tested configuration SFR does not provide any improvement to frame times. This is due to the performance difference between GPU 1 and 0 being almost 3.75 times. A pairing of more similar power would yield much better results however at higher resolutions (mainly UHD) the speed of the PCIe bus will limit frametimes. One key factor is the rendering of shadow lights for the scene on GPU 1 taking around 25 ms. This computation is duplicated on both GPUs which is very inefficient.

## 8.6 SFR WITH MULTI-GPU SHADOW MAPPING

This section contains the results and analysis for split frame rendering with multi-GPU shadow mapping (see 6.1.6). The variations below change the ratio of the SFR split and the number of shadow lights computed on each GPU. If a shadow is not rendered on a GPU it is copied from the other GPU.

### 8.6.1 SFR Shadows with zero lights on GPU 1

In this scenario zero lights shadow maps are computed on GPU 1 and four on GPU 0. The shadow buffer is sized to fit four lights data.

#### 8.6.1.1 Results

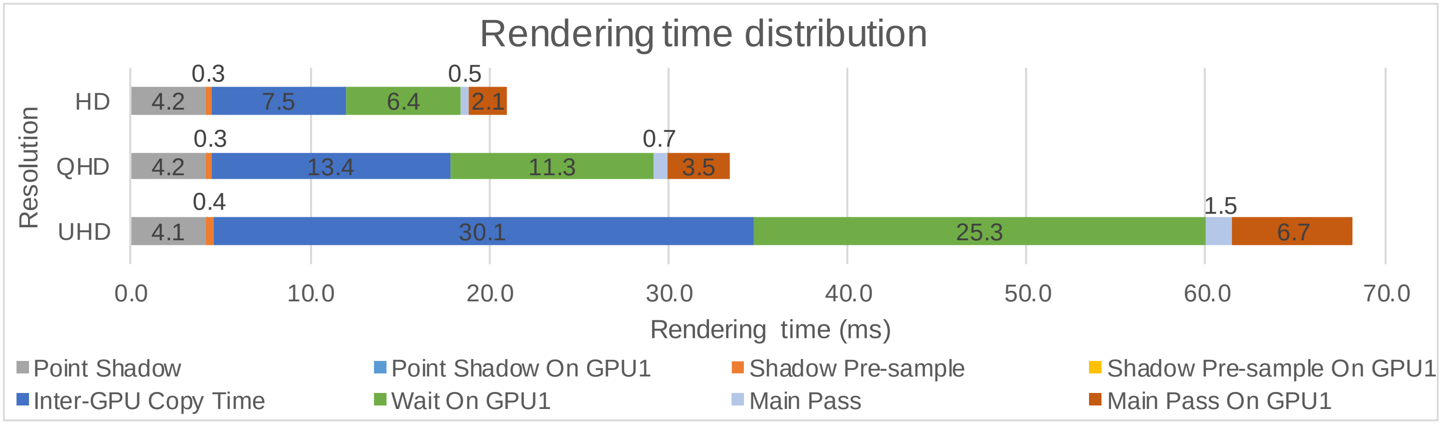

*Figure 44: Rendering time distribution for SFR with multi-GPU shadow mapping with zero lights on GPU 1*

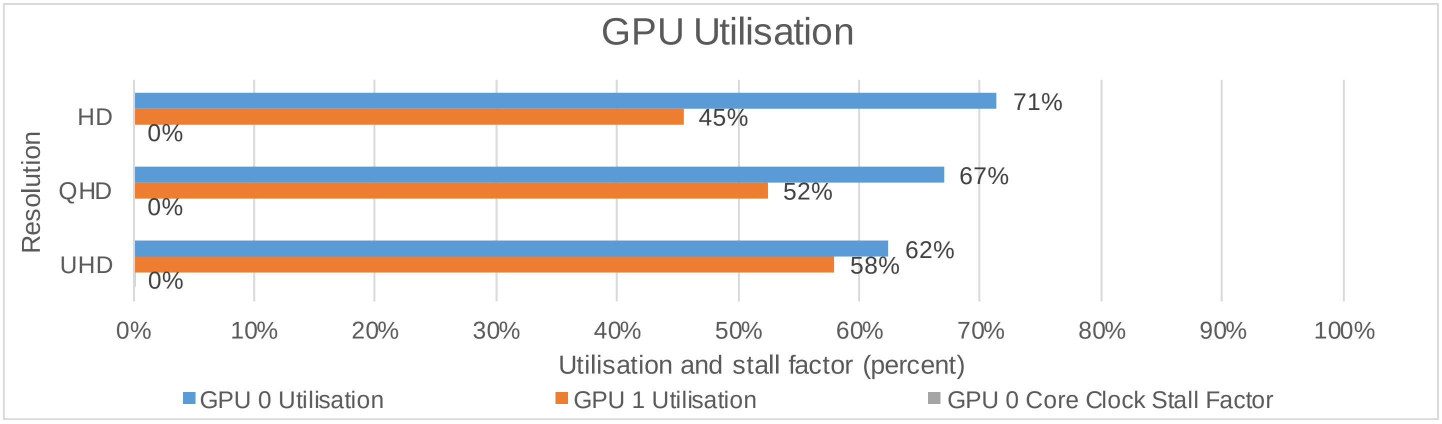

*Figure 45: GPU utilisation for SFR with multi-GPU shadow mapping with zero lights on GPU 1*

| Metric | HD | QHD | UHD |
|---|---|---|---|
| Inter-GPU transfer size (MB) | 2.488 | 4.424 | 9.953 |
| GPU 0 clock speed (MHz) | 1987 | 1987 | 1986 |
| GPU 1 clock speed (MHz) | 1032 | 1032 | 1133 |

*Table 14: Data transfer size and clock speed for SFR with multi-GPU shadow mapping with zero lights on GPU 1*

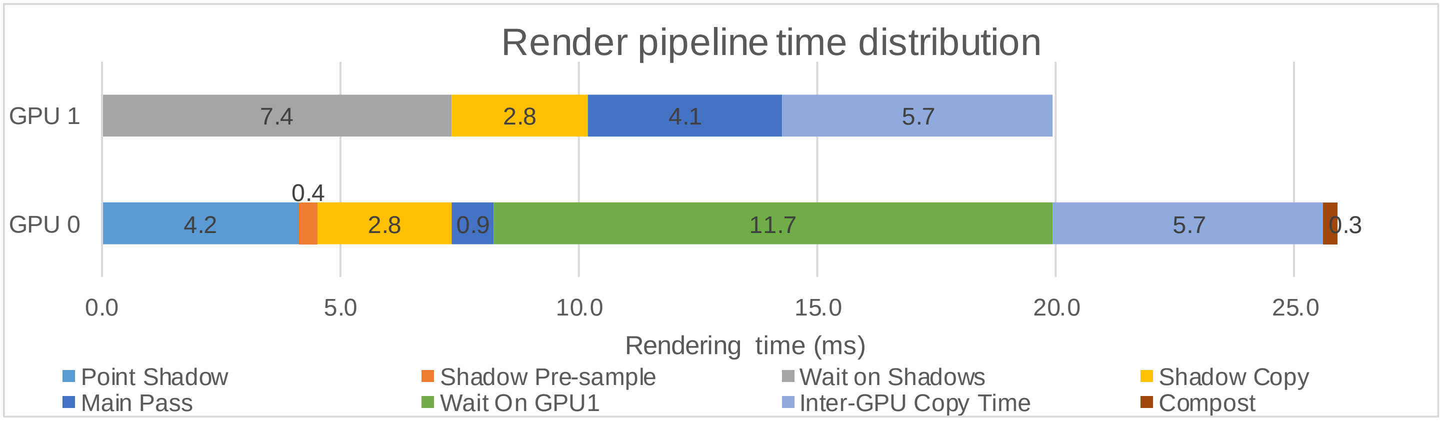

*Figure 46: Render pipeline time distribution for SFR with multi-GPU Shadow mapping with zero lights on GPU 1*

#### 8.6.1.2 Analysis

This variation has poor performance due to pipeline stalling on both GPUs. This is due to GPU 1 only running its section of the main pass, taking around 4ms to render. This lack of work causes GPU 1 to throttle down. In addition, GPU 1 must wait for the shadow maps to be computed on GPU 0 and copied over. Once GPU 1 has rendered the scene GPU 0 must wait for it to copied back which causes GPU 0 to throttle down. The copy time for this variation is almost 30ms at UHD, which is very long for a realtime application.

### 8.6.2 SFR Shadows with one light on GPU 1

In this scenario one lights shadow map is computed on GPU 1 and three on GPU 0. The shadow buffer is sized to fit five lights data, due to format limitations discussed in 6.1.3.

#### 8.6.2.1 Results

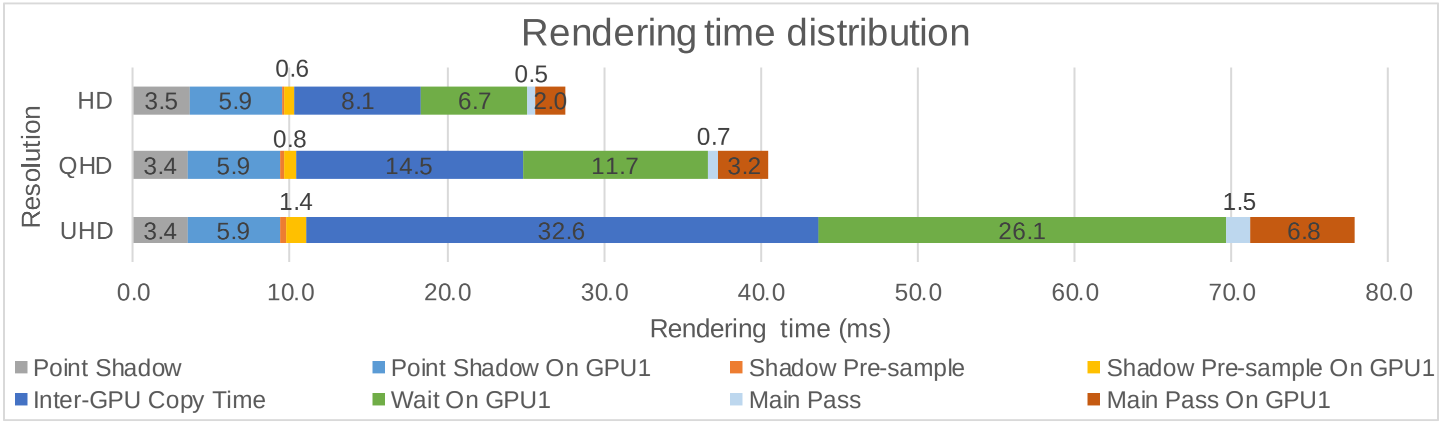

*Figure 47: Rendering time distribution for SFR with multi-GPU shadow mapping with one light on GPU 1*

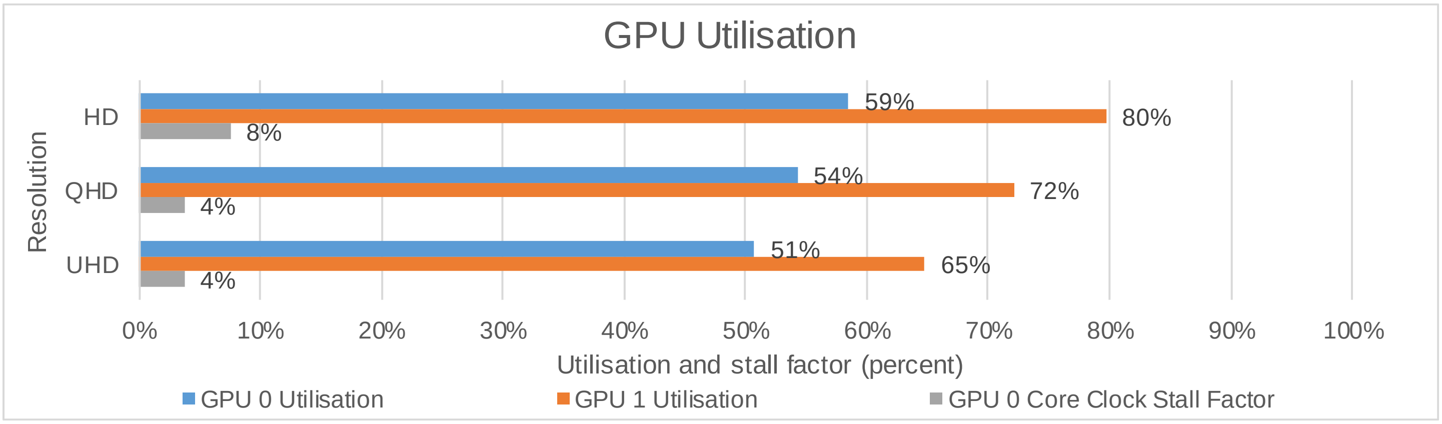

*Figure 48: GPU utilisation for SFR with multi-GPU shadow mapping with one light on GPU 1*

| **Metric** | **HD** | **QHD** | **UHD** |
|---|---|---|---|
| Inter-GPU transfer size (MB) | 2.696 | 4.792 | 10.783 |
| GPU 0 clock speed (MHz) | 1835 | 1911 | 1911 |
| GPU 1 clock speed (MHz) | 1137 | 1137 | 1137 |

*Table 15: Data transfer size and clock speed for SFR with multi-GPU shadow mapping with one light on GPU 1*

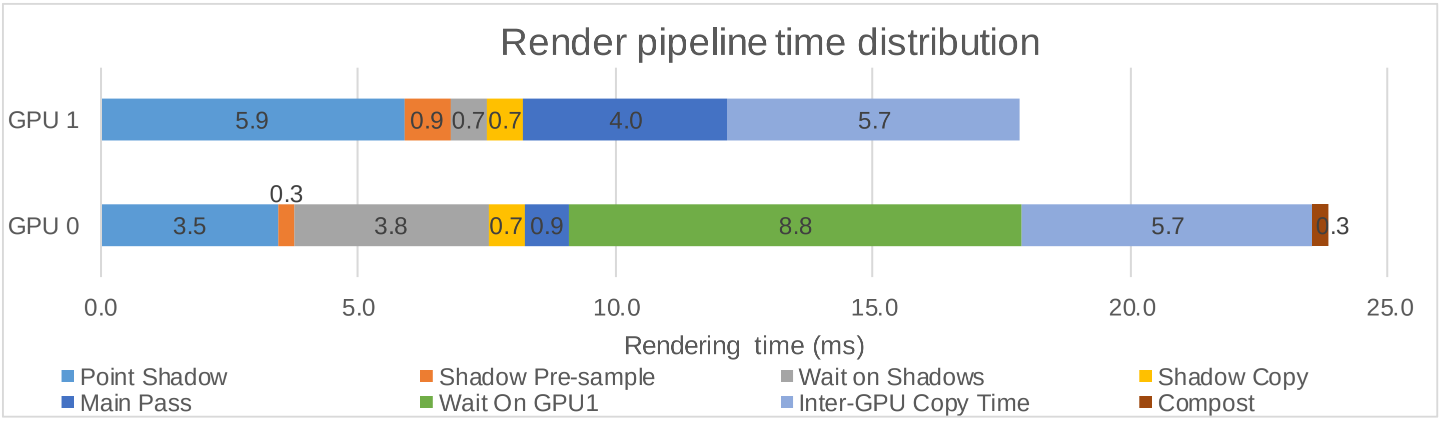

*Figure 49: Render pipeline time distribution for SFR with multi-GPU shadow mapping with one light on GPU 1*

#### 8.6.2.2 Analysis

This varation shows increased utilisations over 8.6.1 however both GPUs are waiting on the PCIe to transfer shadow data and final frame data. Again the amount of inter-GPU data is high here causing the stalling of both GPUs due to the long transfer times. Due to format limitations discussed in 6.1.3, this format transfers five lights worth of data instead of the needed four.

### 8.6.3 SFR Shadows with two lights on GPU 1

In this scenario two lights shadow maps are computed on GPU 1 and two on GPU 0. The shadow buffer is sized to fit four lights data.

#### 8.6.3.1 Results

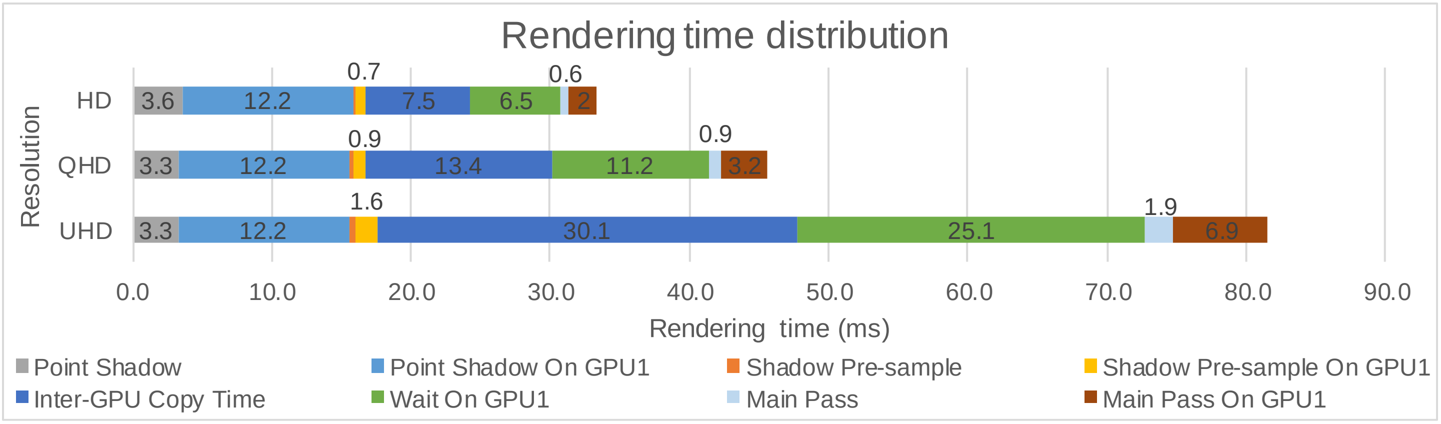

*Figure 50: Rendering time distribution for SFR with multi-GPU Shadow mapping with two lights on GPU 1*

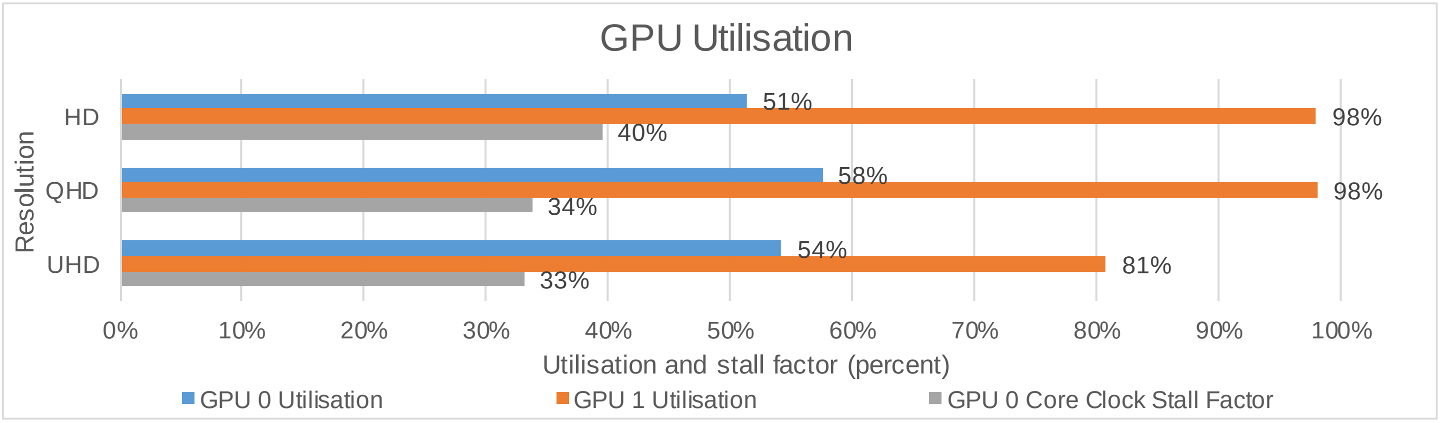

*Figure 51: GPU utilisation for SFR with multi-GPU shadow mapping with two lights on GPU 1*

| Metric | HD | QHD | UHD |
|---|---|---|---|
| Inter-GPU transfer size (MB) | 2.488 | 4.424 | 9.953 |
| GPU 0 clock Speed (MHz) | 1202 | 1316 | 1329 |
| GPU 1 clock Speed (MHz) | 1137 | 1137 | 1137 |

*Table 16: Data transfer size and clock speed for SFR with multi-GPU shadow mapping with two lights on GPU 1*

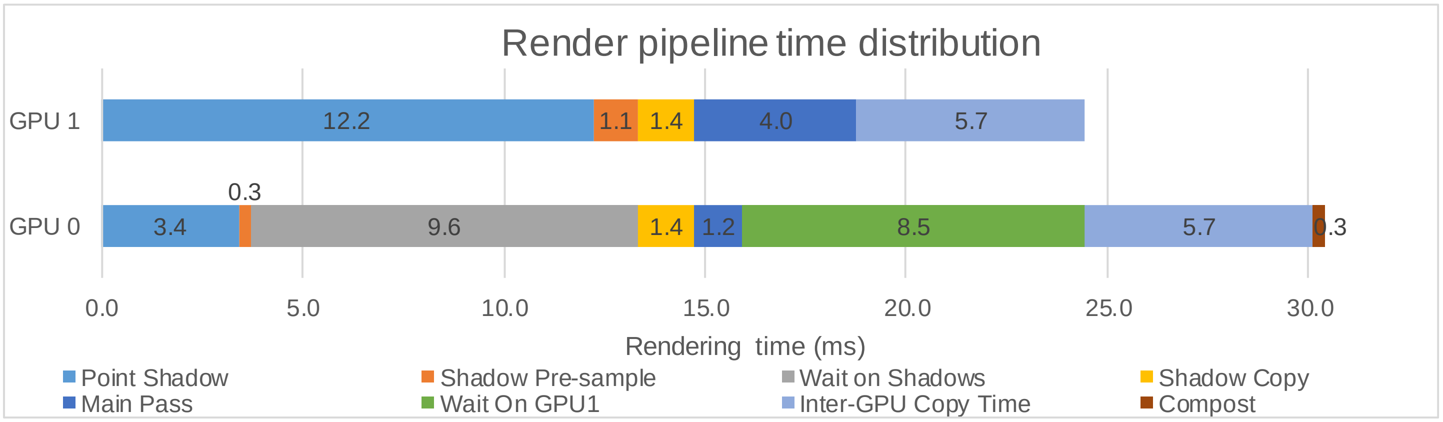

*Figure 52: Render pipeline time distribution for SFR with multi-GPU shadow mapping with two lights on GPU 1*

#### 8.6.3.2 Analysis

Due to format limitations discussed in 6.1.3, this variation has the smallest shadow data of all SFR shadow variations as there are two buffers each sized for two light's shadow maps. GPU 1 utilisation has improved again as more work is being done however both GPUs are still stalled by the PCIe bus at UHD.

### 8.6.4 SFR Shadows with three lights on GPU 1

In this scenario three lights shadow maps are computed on GPU 1 and one on GPU 0. The shadow buffer is sized to fit five lights data, due to format limitations discussed in 6.1.3.

#### 8.6.4.1 Results

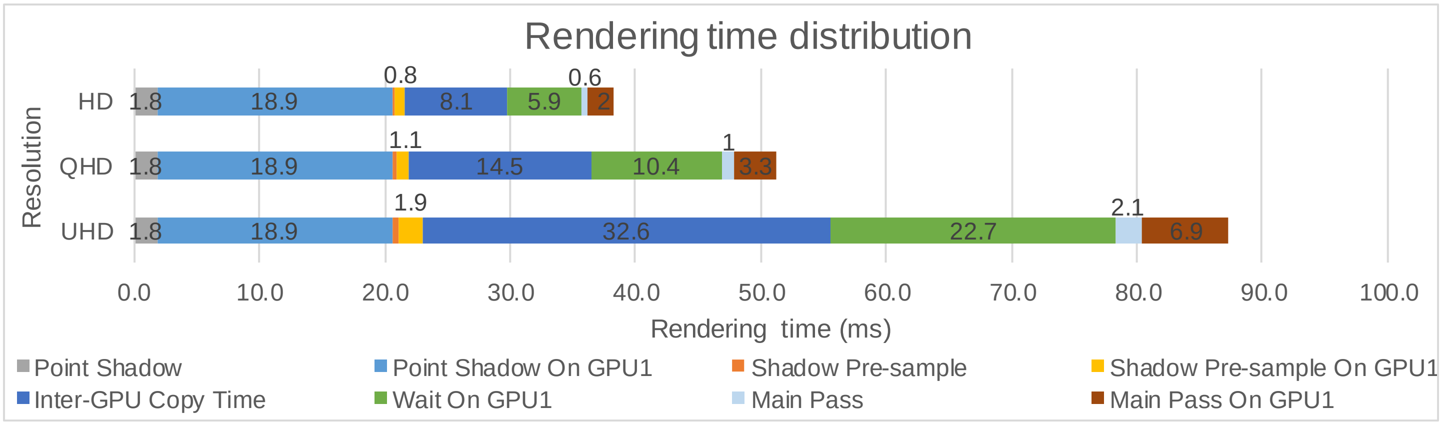

*Figure 53: Rendering time distribution for SFR with multi-GPU shadow mapping with three lights on GPU 1*

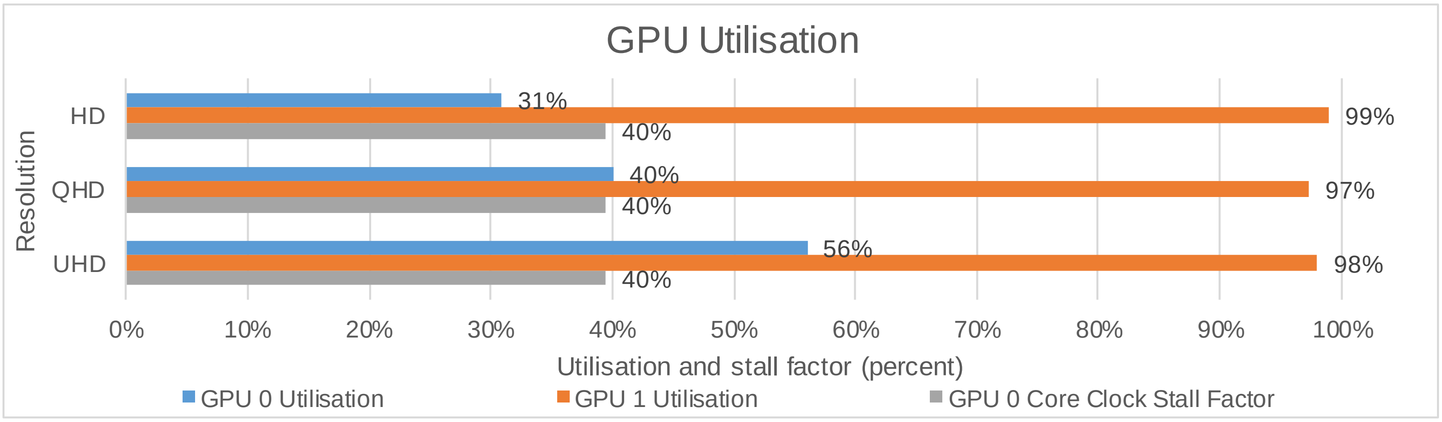

*Figure 54: GPU Utilisation for SFR with multi-GPU shadow mapping with three lights on GPU 1*

| **Metric** | **HD** | **QHD** | **UHD** |
|---|---|---|---|
| Inter-GPU transfer size (MB) | 2.696 | 4.792 | 10.783 |
| GPU 0 clock Speed (MHz) | 1202 | 1202 | 1202 |
| GPU 1 clock Speed (MHz) | 1137 | 1137 | 1137 |

*Table 17: Data transfer size and clock speed for SFR with multi-GPU shadow mapping with three lights on GPU 1*

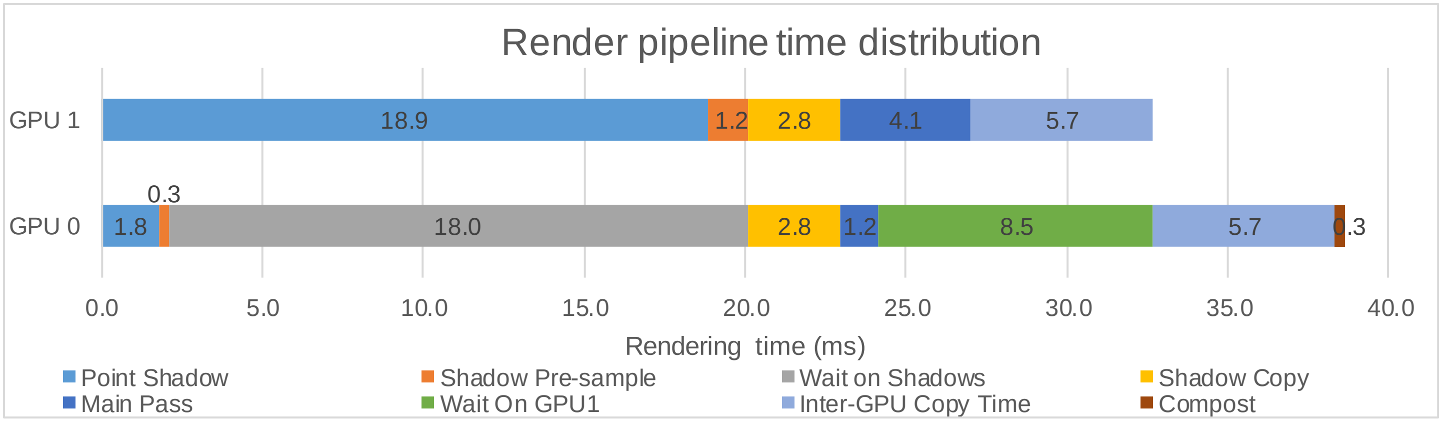

*Figure 55: Render pipeline time distribution for SFR with multi-GPU shadow mapping with three lights on GPU 1*

#### 8.6.4.2 Analysis

This variation is limited by GPU 1 rendering three light’s shadow maps. This time spent waiting on GPU 1 causes GPU 0 to stall and throttle down impacting overall frametimes. Additionally, lots of time is spent transferring data to GPU 0. Due to format limitations discussed in 6.1.3, this format transfers five lights worth of data instead of the needed four.

### 8.6.5 SFR Shadows with four lights on GPU 1

In this scenario four lights shadow maps are computed on GPU 1 and zero on GPU 0.

#### 8.6.5.1 Results

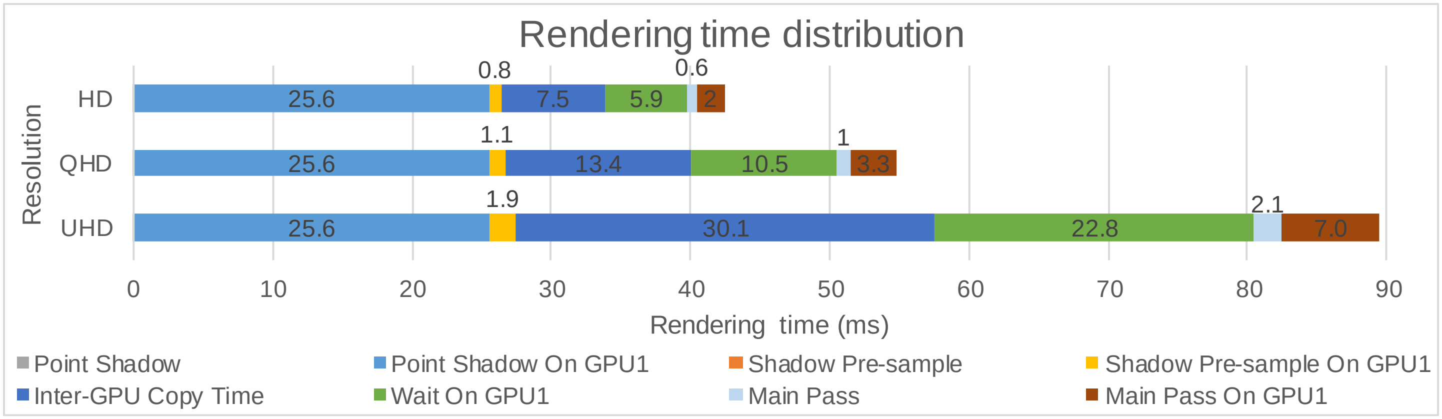

*Figure 56: Rendering time distribution for SFR with multi-GPU shadow mapping with four lights on GPU 1*

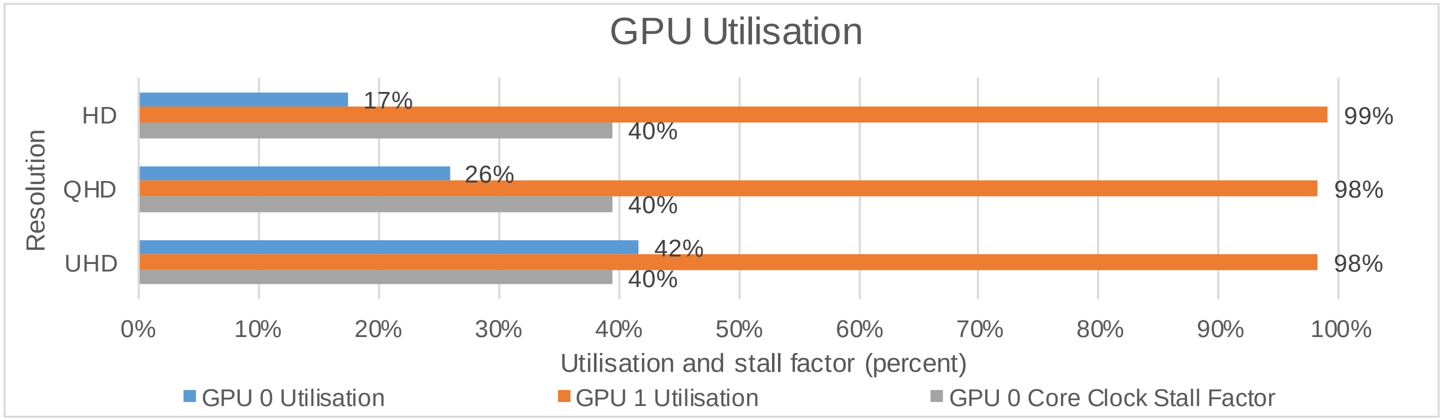

*Figure 57: GPU utilisation for SFR with multi-GPU shadow mapping with four lights on GPU 1*

| **Metric** | **HD** | **QHD** | **UHD** |
|---|---|---|---|
| Inter-GPU transfer size (MB) | 2.488 | 4.424 | 9.953 |
| GPU 0 clock Speed (MHz) | 1202 | 1202 | 1202 |
| GPU 1 clock Speed (MHz) | 1137 | 1137 | 1137 |

*Table 18: Data transfer size and clock speed for SFR with multi-GPU shadow mapping with four lights on GPU 1*

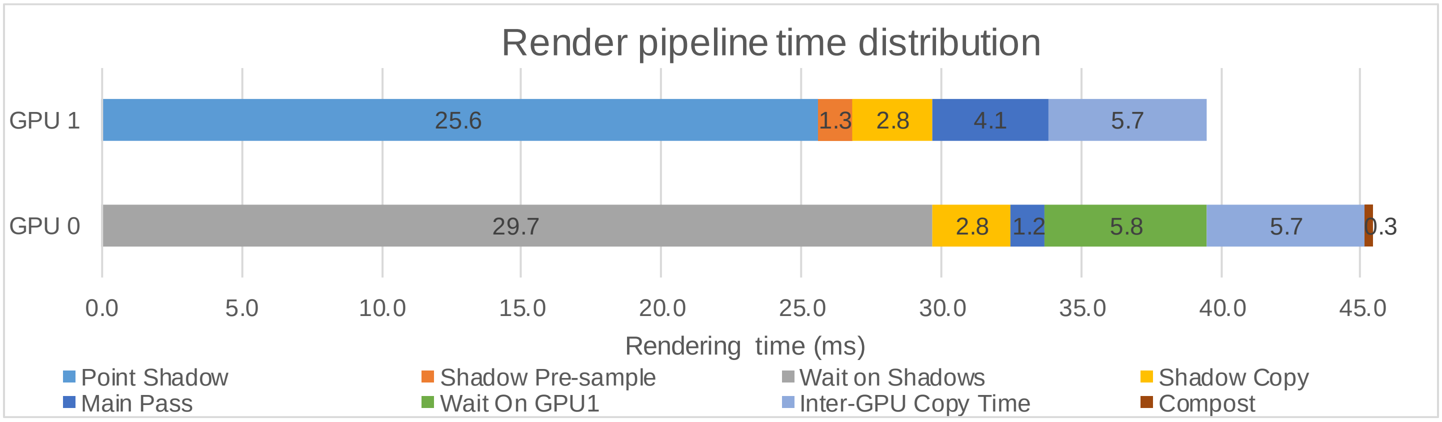

*Figure 58: Render pipeline time distribution for SFR with multi-GPU shadow mapping with four lights on GPU 1*

#### 8.6.5.2 Analysis

In this variation GPU 0 stalls heavily due to the time spent waiting on GPU 1. This variation could be more performant if GPU 0 had other work that took enough time to cover the time GPU 1 spends calculating shadows. This variation demonstrates the large performance difference between the two GPUs and how this factor affects this technique negatively. This variation is expected to perform badly as the lesser GPU (GPU 1) is performing most of the work.

### 8.6.6 SFR Shadows with a ratio of 95% and one light on GPU 1

In this scenario one lights shadow maps are computed on GPU 1 and three on GPU 0. The SFR ratio is set to 95%.

#### 8.6.6.1 Results

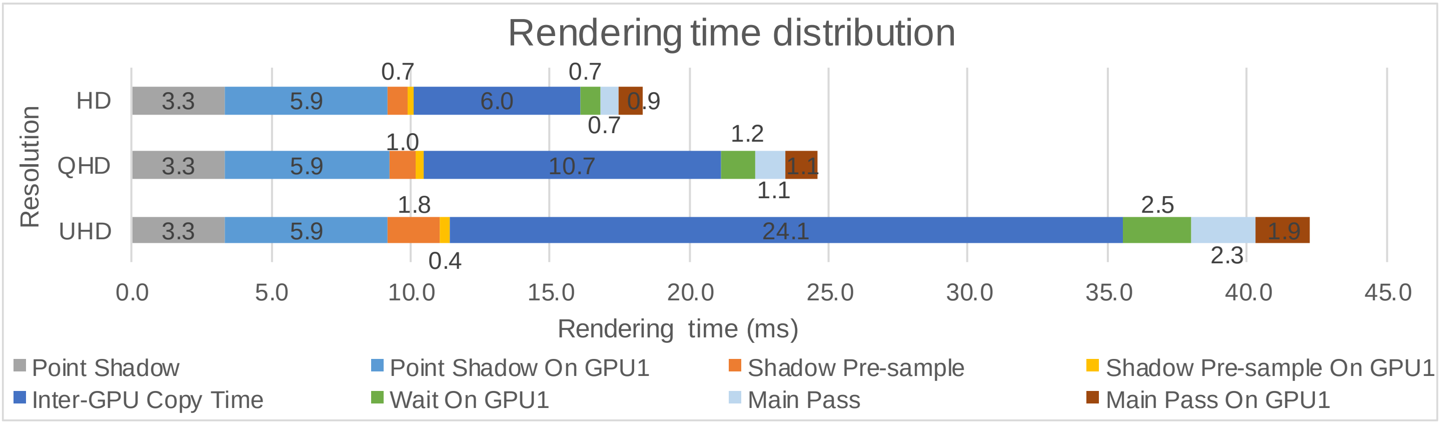

*Figure 59: Rendering time distribution for SFR with multi-GPU shadow mapping with one light on GPU 1 and a ratio of 95%*

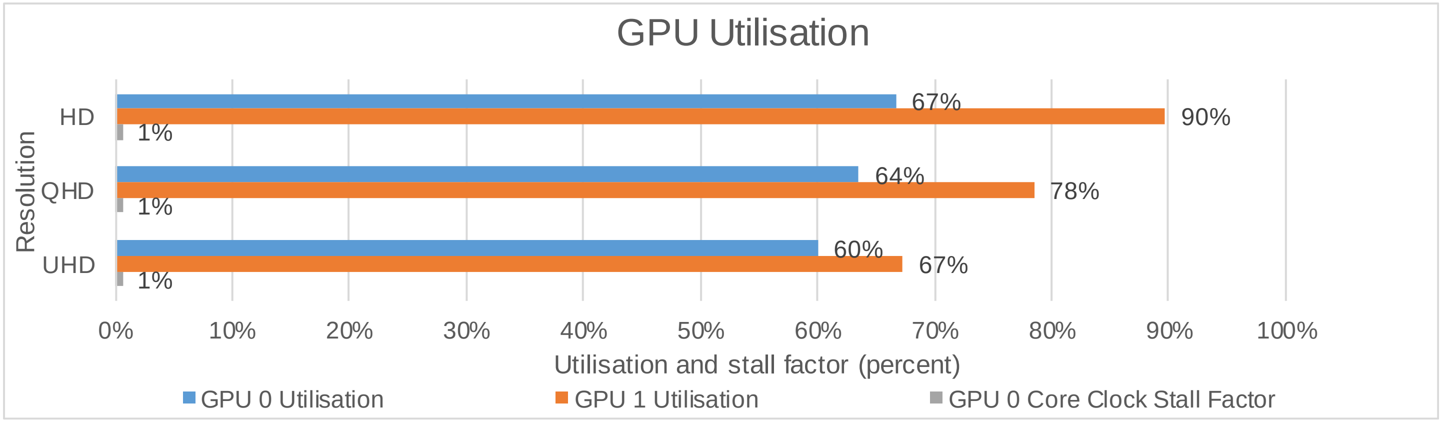

*Figure 60: GPU utilisation for SFR with multi-GPU shadow mapping with one light on GPU 1 and a ratio of 95%*

| **Metric** | **HD** | **QHD** | **UHD** |
|---|---|---|---|
| Inter-GPU transfer size (MB) | 1.991 | 3.539 | 7.963 |
| GPU 0 clock Speed (MHz) | 1974 | 1974 | 1974 |
| GPU 1 clock Speed (MHz) | 1137 | 1137 | 1137 |

*Table 19: Data transfer size and clock speed for SFR with multi-GPU shadow mapping with one light on GPU 1 and a ratio of 95%*

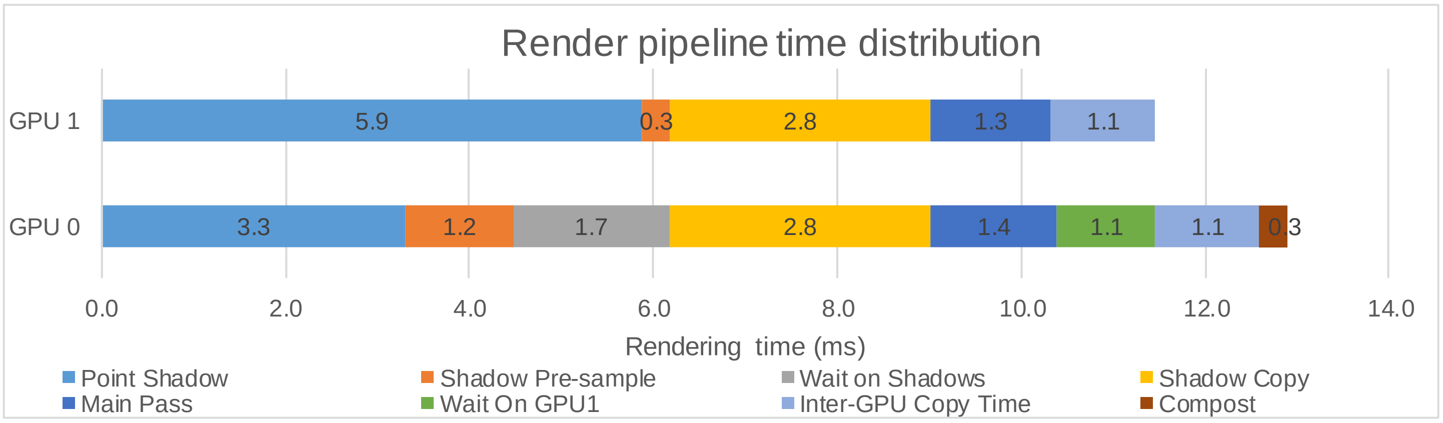

*Figure 61: Render pipeline time distribution for SFR with multi-GPU shadow mapping with one light on GPU 1 and a ratio of 95%*

#### 8.6.6.2 Analysis

This variation shows that with the significantly less powerful GPU 1, SFR shadows has serious issues with achieving good GPU utilisations. However, this variation does manage to avoid stalling both GPUs. The key performance limiting factor in this variation is the transfer across the PCIe bus. Executing one light on GPU 1 while it slows the shadow pass, prevents GPU 1 from stalling which would cause the main pass on GPU 1 to be significantly slower. The 1% stall factor in Figure 60 is caused by GPU frequency peaking in one of the variations causing the maximum to be higher than the sustained boost frequency for the GPU.

### 8.6.7 Technique conclusion

#### 8.6.7.1 HD

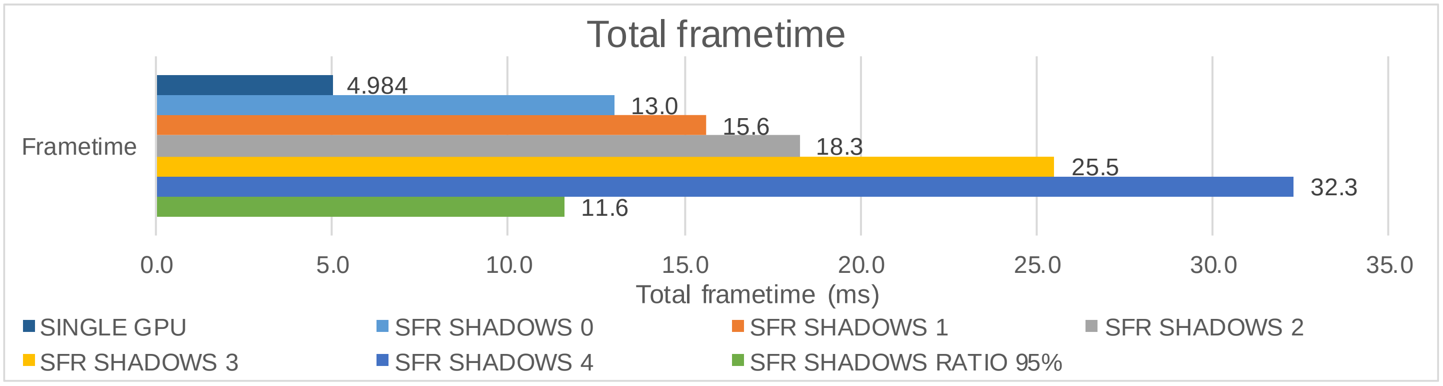

*Figure 62: Frametimes for SFR with multi-GPU shadow mapping at HD*

At HD, this technique suffers heavily from the lack of power of GPU 1 and shows some of the highest frametimes for HD in this paper. This is technique would be more effective with two similarly powered GPUs or a pair without a large performance difference. It is heavily limited by the PCIe bus and would benefit greatly from any methods of reducing data size.

#### 8.6.7.2 QHD

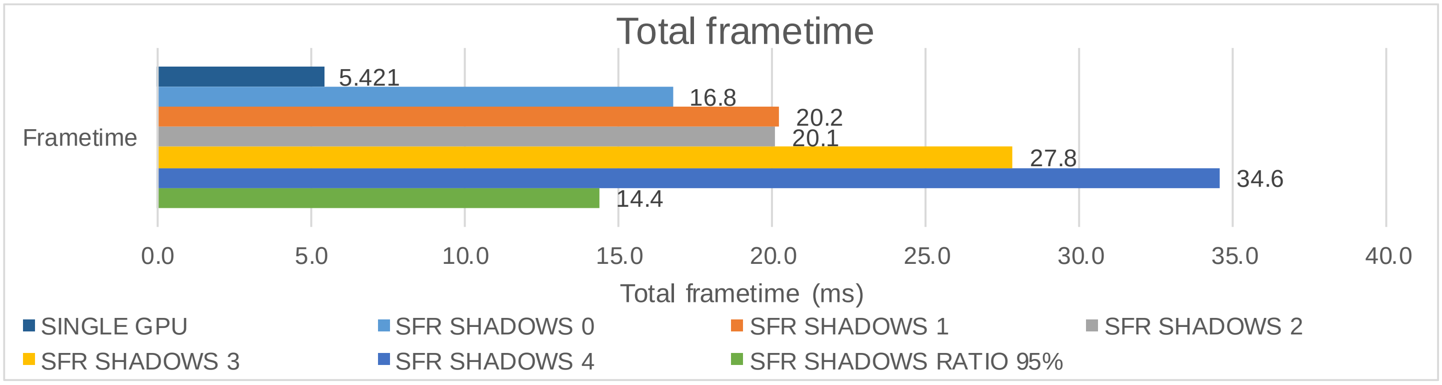

*Figure 63: Frametimes for SFR with multi-GPU shadow mapping at QHD*

There is an anomaly between shadows one and two which is due to the transfer of shadow data. As outlined in 6.1.3 when sampling three shadows on a GPU the buffer is sized for four lights, due to format limitations. This causes the data for five lights to be transferred compared to four in SFR shadows two. The same is true for SFR shadows three. As mentioned at HD, this technique transfers a lot of data across the PCIe bus which is its key limiting factor.

#### 8.6.7.3 UHD

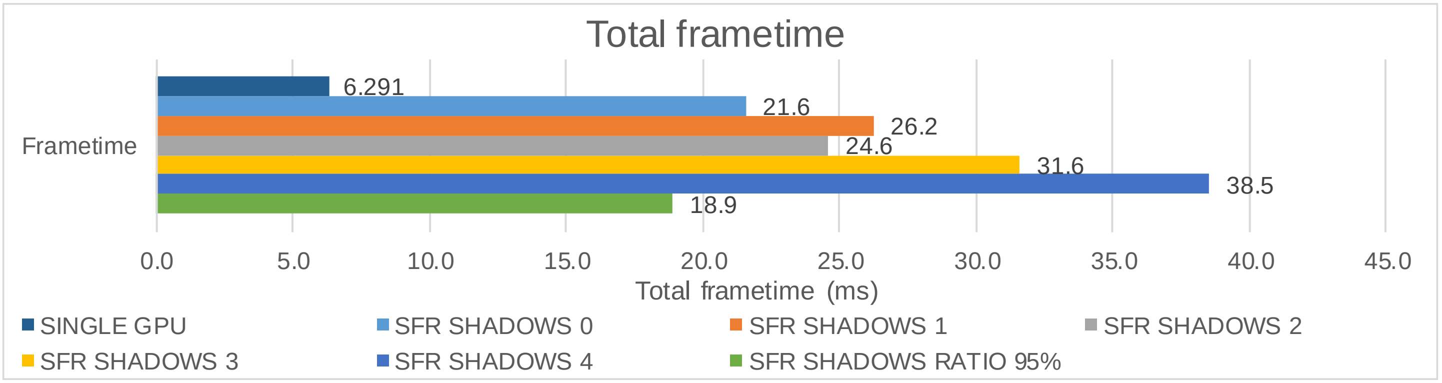

*Figure 64: Frametimes for SFR with multi-GPU shadow mapping at UHD*

At UHD, the PCIe bus severely limits this techniques speed and effectiveness especially due to the higher pixel count of UHD. 38.5ms is the highest frame time in this paper and equates to 25 frames per second (FPS) which is a significant performance loss considering the power of the hardware used and the workload.

#### 8.6.7.4 Overall Conclusion

SFR with multi-GPU shadow mapping attempts to reduce duplicate computation and is better than SFR in some cases however two key issues are present. The main issue is the length of transfer time. This is due to both shadow data and final frame data needing to be transferred which is a large amount of data. In the current rendering pipeline, there is no way to hide this time and so it directly affects frametimes. The other issue is GPU work distribution, as shown by SFR shadows with zero lights on GPU 1, when done incorrectly both GPUs can have very poor utilisation and so throttle down causing extremely long frametimes.

## 9. EXPECTED RESULTS FROM OTHER GPUS

From the results outlined in section 8 it is possible to predict the performance of other GPUs in these techniques. Two metrics were investigated in this section; shadow map rendering time and main pass rendering time. These were chosen as they represent the key limiting factors of the techniques investigated in this paper.

The GTX 1060 was not tested in the test machine listed in 7.2. However, the key variables have been kept effectively constant. The CPU used was sufficiently powerful so that it did not limit GPU performance. Of note, the GTX 1060 has a x16 PCIe connection compared to the test machine's x8. This would slightly affect speed of CPU side buffers being updated however this difference is negligible. For Figure 66 the results of each resolution have been averaged.

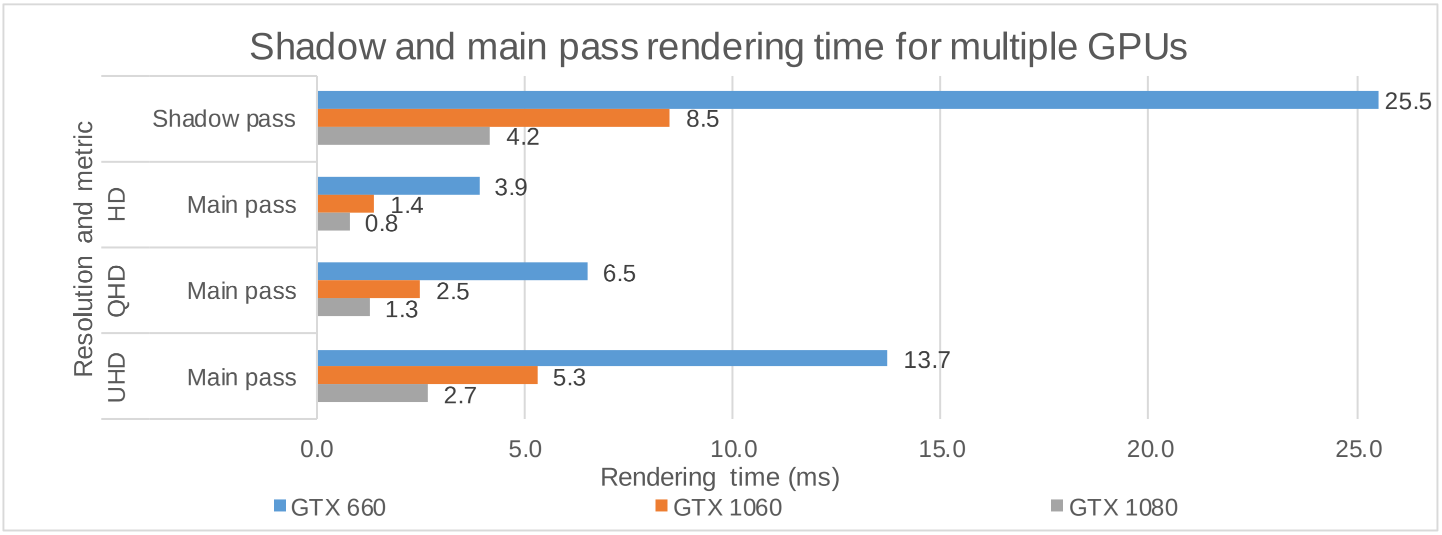

*Figure 65: Rendering times for a selection of GPUs*

Figures Figure *65* and Figure *66* show that the GTX 1060 is much faster than the GTX 660. This would allow the techniques investigated in this paper to provide an improvement to frametimes over single GPU operation. It shows that an even more powerful GPU 1 could be used to improve frametimes further. One key observation here, is that the relative performance of the GPUs is more important than the absolute performance when utilising heterogenous multi-GPU techniques.

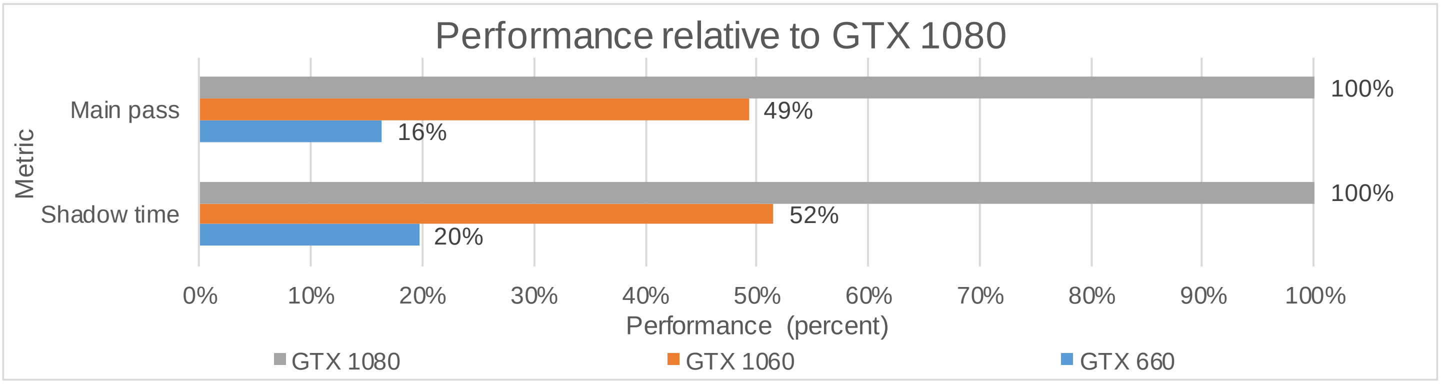

*Figure 66: Relative performance of GPUs for two render passes*

## 10. CONCLUSION

An analysis of the results presented in section 8 show that the multi-GPU techniques presented in this paper do not provide an improvement to overall frametimes in the test system. This is due the performance difference between the two GPUs being too extreme (3.75x).

Asynchronous multi-GPU shadow mapping was the only technique to get close to providing an improvement in frametimes versus single GPU operation. This was due to two key factors; the minimisation of inter-GPU data transfer and overlapping of as much GPU 1 work as possible with work on GPU 0.

As shown by the data in section 9, a GTX 1060 with a GTX 1080 would provide lower overall frametimes than a single GTX 1080. This is due to the GTX 1060 rendering shadow maps more than three times faster than a GTX 660 and displays a similar performance improvement when rendering the main scene. This result shows that many pairings of similarly powered GPUs will see an improvement to overall frametimes when using the techniques investigated in this paper.

## 10.1 KEY FACTORS

From the results set out in this paper, three key factors have been identified which determine the effectiveness of multi-GPU techniques to reduce frametimes.

1. The amount of data transferred across the PCIe bus should be kept to a minimum, as shown by the differences in GPU copy time between SFR and SFR with multi-GPU shadow mapping.
2. The pipeline should minimise the time spent waiting for another GPU to finish an operation. This is shown by the improvement that asynchronous shadow mapping provides over multi-GPU shadow mapping. One very effective strategy to achieve this is running work one frame behind, although for some workloads this would not be possible.
3. It is important for any multi-GPU technique to detect when two GPUs have such a large performance difference that no improvement would be provided or the inter-GPU connection is too slow, and the application should switch back to single GPU operation for optimal performance.

## 10.2 LIMITATIONS

### 10.2.1 PCIe bus

A key limiting factor in all the techniques investigated in this paper is the speed of the PCIe bus. In the tests only the inter-GPU data and any operating system data was transferred across it. In a game engine the PCIe bus would need to serve the streaming system for textures and other assets being loaded onto the GPU from host memory. This would negatively affect the transfer speed and overall framerates of techniques. This could be mitigated by scheduling the streaming copy work to execute when the graphics pipeline is not using the copy engine. However, this would cause reduced streaming performance and could cause assets to stream slowly affecting image quality.

### 10.2.2 Advanced rendering features

Due to the young age of the test engine, it does not support many advanced rendering effects such as screen space reflections or ambient occlusion. These effects would run on GPU 0 which would take up an amount of rendering time that would allow techniques such as asynchronous shadow mapping more time to render. With GPU 0 taking a few milliseconds longer to render, asynchronous shadow mapping with one light would provide an improvement over single GPU operation in the test system.

### 10.2.3 SFR

The engine currently only supports dual GPU configurations however it would be trivial to split the frame into more slices. When using more than two GPUs, the final output could be assembled in host memory. This would allow the display cost to be constant as only one copy is required per GPU. However, many consumer grade systems would not have enough PCIe lanes to run more than two GPUs at x8 PCIe speed. Some workstations CPUs and chipsets have more than 60 PCIe lanes such as the Ryzen Threadripper 2990WX (Cutress, 2018) so would be able to run such a configuration.

To find the best ratio, a dynamic ratio scaler was implemented which would change the ratio based on the main pass time of each GPU averaged over a period. This implementation is very naïve as it only considers a simple metric, however it completed its goal and found a better screen ratio. To improve the distribution of work the frame could be split based on frame complexity and assigned out in square regions to an array of many GPUs.

### 10.2.4 GPU Memory and SFR

In the test system the GPUs have very different memory configurations (see Table 1). If the scene required more memory than GPU 1 has available, it would need to page data to host memory to complete the frame which would cause extremely poor performance. This could be addressed using mega textures.

Mega textures use an atlas to texture the scene, this atlas only contains the needed textures to render the scene. Textures are added to the atlas based on a feedback buffer. This allows reduced memory usage and could allow SFR on two or more GPUs with different memory sizes to function without visual differences by limiting the mega texture memory allocation to the smaller card’s maximum memory. However, this could still cause underutilisation of the larger GPU’s memory in some scenarios.

### 10.2.5 Multi-GPU shadows

The pre-sample buffer’s size can limit overall frametime so attempts could be made to reduce its size. For instance, using a R8 texture and encoding the visibility of a light in each bit. This would allow 8 lights at 8 bits per pixel. However, this would prevent soft shadows as a pixel is either visible or not. It would be possible to encode a soft shadow in 2 bits however this would negatively affect softness of the shadows as only $4(2^2)$ different values could be represented.

### 10.2.6 Directional lights

Multi-GPU shadow mapping currently uses point source lights which are the most computationally complex form of shadowing light for shadow mapping. This was done to generate a sizeable chunk of GPU work consistent with other techniques that could benefit from execution across multiple GPUs. Directional lights are around a sixth of the cost of a point light and could have been used instead. Directional lights would have provided a finer gainer of control over the work splitting and would have helped make techniques more viable in the test system.

Currently the shadows only use a single sample; this causes the shadow to have a hard edge which for many cases is not physically accurate. Soft shadows use multiple samples taken across a small region to generate much softer and more accurate shadows in the final image. This adds shading complexity to shadow sampling in the main pass and the pre-sample pass. However, this would not require any extra inter-GPU data.

## 11. FURTHER WORK

## 11.1 OTHER HARDWARE

Further work could investigate the use of an Integrated GPU (IGPU) to boost the performance of a single discrete GPU. This would be useful on modern Intel and AMD platforms as they ship with an IGPU as standard. When a discrete GPU is present the IGPU is idle in most situations. One limitation would be the performance difference between the GPUs; with a lower end/entry level discrete GPU and a high powered IGPU an improvement to frametimes could be achieved. One added advantage to the IGPU is with inter-GPU transfers. As IGPUs use system (host) memory as their local memory segment, a discrete GPU could copy directly to and from it halving the transfer time compared to two discrete GPUs.

Another avenue for investigation is the use of other adaptor connection technologies such as NVLink which would allow direct GPU-GPU transfer at a much higher speed than with the PCIe bus. Currently the link is only active between matching pairs of supported GPUs however this is most likely a software limitation and thus could be disabled or worked around to allow other configurations to utilise it.

## 11.2 OTHER TECHNIQUES

One area of investigation would be utilisation of more than two GPUs. Many of the techniques discussed in this paper would scale to more GPUs however PCIe lanes limits and overall PCIe bandwidth would become major limiting factors.

### 11.2.1 Ray tracing

With the introduction of Nvidia's RTX line of GPUs there has been an increase in interest in real time raytracing and using multiple GPUs would help handle the very computationally intensive task. Either the extra GPU(s) ray trace and shade pixels into a render target which would then be copied back to GPU 0 or just ray trace and send the intersection data to GPU 0. The second approach could use compression and different formats to reduce the data transfer without sacrificing pixel quality. Raytraced reflections are a very good candidate to run asynchronously as there is little visual difference with reflections being a frame behind.

### 11.2.2 Multi-GPU compute

One area not investigated in this project was the use of multi-GPUs for more general compute tasks. This has been widely used for tasks such as path traced scene rendering for offline renderers. With access to additional GPUs that can run complete separately from GPU 0, work that takes multiple frames to complete would be possible to run without affecting frametimes. For example, path tracing (or photon mapping) is used in lightmap generation for Global Illumination (GI); this could be run on the extra GPU(s) to provide semi-realtime GI at a much higher quality than current real time GI techniques. Once the mapping is complete the result would be written to the texture that GPU 0 uses to render the scene. This texture would be used to render frames until the next result is ready. Additionally, current real-time GI solutions such as Light Propagation Volumes (LPVs) (Kaplanyan, 2009) could be run at a higher resolution without affecting frametimes.

More general compute applications would benefit from multi-GPU such as scene physics, cloth, fluids etc. An extra GPU could simulate the particles and even render the result. GPU 0 would then copy the texture/data buffer for use in the final frame. One side effect of copying through host memory would be its availably to the CPU, which could use the data as part of a larger simulation that is completed on the CPU. Splitting the work across multiple GPUs could be problematic, for instance, a single piece of cloth would difficult to split and maintain simulation accuracy.

### 11.2.3 Inter-GPU data compression

Compression could be used to reduce the amount of data transferred between GPUs. For image compression GPUs support multiple compressed formats such as DXT1 or BC5 etc. While GPUs can decode these formats rapidly, encoding is a more compute intensive task and could take too long to provide any improvement. Also, BC5 is a lossy codec meaning image quality would be lost. For raw data, such as shadow maps, data could be encoded into fewer bits reducing the accuracy of the data, but this might not affect the end image result significantly. One other benefit of bit compression is the very short compression time compared to all other compression techniques. One issue could be hardware support for bitwise operations in a compute shader.

## 11.3 TEST ENGINE IMPROVEMENTS

### 11.3.1 GPU Improvements

#### 11.3.1.1 Rendering performance

Currently a very simple form of caching for pipeline state objects is implemented in the test engine. This could be greatly improved by comparing all properties on the PSO and building PSOs that can be shared to reduce pipeline flushes caused by changing state. Another improvement would be grouping objects by PSO when rendering to further reduce switches.

DirectX 12 gives the responsibility of resource barriers to the application. Barriers change a resource's state or ensure that all writes are complete. Barrier minimisation is very important to an applications GPU performance as barrier transitions can have a high GPU cost. Currently, the engine has CPU side resource state shadowing which prevents duplicate transitions however during some operations resources state can decay. This implicit state decay could be used to further reduce the number of barriers used without compromising stability of the application.

#### 11.3.1.2 Memory management

Currently the engine has a very simple memory management system, a full management system should be implemented. This would allow the engine to remove unneeded resources from the GPU memory and would support resource streaming such as textures, meshes etc. Texture streaming is very important as textures can be very large and, in some cases (from a hard disk), could take a long time to load into memory. Streaming would load textures asynchronously over time as they are read off disk. For rendering primitives such as command lists or framebuffers, resource aliasing could be used to reduce the essential memory footprint of the engine. Resource aliasing allows the same memory to be used for multiple different resources (not at the same time). This is great for temporary framebuffers like the ones used for offscreen effects such as bloom etc.

Mega texturing (see 11.2) could be implemented to manage scene texture memory. However, the above strategy would still be used for rendering primitives such as framebuffers and command lists, etc.

### 11.3.2 CPU Improvements

While the CPU performance of the engine does not currently limit the GPU, the current workload is trivial compared to the work that a released game would do. Some systems in the engine are currently implemented in a way that is simpler and not focused on performance. One example is the performance capture system which currently uses many string comparisons which are slow compared to integer ones. Another example is the text renderer which would benefit from caching text into larger draw calls reducing CPU computation time. Additionally, improvements could be made to the rendering quality of the text.

Currently shaders are compiled synchronously however material shaders could be complied asynchronously. A default material shader would be used while the material shader is compiled, and the correct shader would then be swapped in. This would allow the engine to be ready much quicker when lots of material shaders need to be compiled. It would also allow live recompiling of material shaders for fast iteration.

#### 11.3.2.1 Threading

One key technique that modern engines make heavy use of is threading, as used in UE4 (Epic Games, 2014). Adding a render thread that runs one frame behind the gameplay/main thread would provide better frametimes at the cost of latency. UE4 uses a single render thread to handle all graphics commands for all devices, however a render thread per GPU could reduce CPU time in multi-GPU configurations.

DirectX 12 allows command lists to be recorded in parallel. For example, the main render pass could be recorded in multiple command lists across multiple threads reducing overall frametimes. A task graph architecture (University of California, Berkeley, 2019) would suit this workload very well as rendering each object has few dependencies. A task graph is a way of distributing tasks across multiple threads and handling their dependencies effectively. This architecture would allow scalability across many systems with differing core counts.

## 12. REFERENCES

Advanced Micro Devices, 2014. *Mantle White Paper.* [Online]
Available at: https://www.amd.com/Documents/Mantle_White_Paper.pdf
[Accessed 20 Jan 2019].

Advanced Micro Devices, 2018. *AMD Radeon Instinct™ MI60 Accelerator.* [Online]
Available at: https://www.amd.com/en/products/professional-graphics/instinct-mi60
[Accessed 5 Jan 2019].

Anon., 2010. *Using SLI Bridge to transfer data.* [Online]
Available at: https://devtalk.nvidia.com/default/topic/470111/using-sli-bridge-to-transfer-data-/
[Accessed 18 Feb 2019].

Apple, 2019. *Metal API.* [Online]
Available at: https://developer.apple.com/metal/
[Accessed 24 April 2019].

Buik, W., 2018. *Support for Unity (Jumbo) Files in Visual Studio 2017 15.8.* [Online]
Available at: https://blogs.msdn.microsoft.com/vcblog/2018/07/02/support-for-unity-jumbo-files-in-visual-studio-2017-15-8-experimental/
[Accessed 14 Feb 2019].

Cutress, I., 2018. *The AMD Threadripper 2990WX 32-Core and 2950X 16-Core Review.* [Online]
Available at: https://www.anandtech.com/show/13124/the-amd-threadripper-2990wx-and-2950x-review
[Accessed 1 Feb 2019].

Davis, S., 2015. *Mantle 101.* [Online]
Available at: https://community.amd.com/community/gaming/blog/2015/05/12/mantle-101
[Accessed 21 Jan 2019].

Epic Games, 2014. *Unreal Engine 4. [Software].* [Online]
Available at: www.unrealengine.com
[Accessed 20 Jan 2019].

finalwire, 2019. *AIDA64.[Software].* [Online]
Available at: https://www.aida64.com/
[Accessed 1 March 2019].

James, H., 2013. *Why Do Dedicated Game Consoles Exist?.* [Online]
Available at: https://prog21.dadgum.com/181.html
[Accessed 04 April 2019].

Kaplanyan, A., 2009. *Light_Propagation_Volumes.* [Online]
Available at: http://advances.realtimerendering.com/s2009/Light_Propagation_Volumes.pdf
[Accessed 24 April 2019].

Karis, B., 2013. *Real Shading in Unreal Engine 4.* [Online]
Available at: https://cdn2.unrealengine.com/Resources/files/2013SiggraphPresentationsNotes-26915738.pdf
[Accessed 27 Feb 2019].

Karis, B., 2014. *HIGH-QUALITY TEMPORAL SUPERSAMPLING.* [Online]
Available at: http://advances.realtimerendering.com/s2014/#_HIGH-QUALITY_TEMPORAL_SUPERSAMPLING
[Accessed 9 Jan 2019].

Marinkovic, S., 2016. *Radeon Software enables an incredible gaming experience with DirectX® 12 multi-GPU Frame Pacing.* [Online]
Available at: https://community.amd.com/community/gaming/blog/2016/10/04/radeon-software-enables-an-incredible-gaming-experience-with-directx-12-multi-gpu-frame-pacing
[Accessed 10 Jan 2019].

Microsoft Corporation, 2008. *Gamefest 2008 and the DirectX 11 announcement.* [Online]
Available at: https://blogs.msdn.microsoft.com/ptaylor/2008/07/28/gamefest-2008-and-the-directx-11-announcement/
[Accessed 10 Feb 2019].

Microsoft Corporation, 2014. *DirectX 12.* [Online]
Available at: https://blogs.msdn.microsoft.com/directx/2014/03/20/directx-12/
[Accessed 5 Feb 2019].

Nvidia Corporation, 2018. *nvlink bridges.* [Online]
Available at: https://www.nvidia.com/en-gb/design-visualization/nvlink-bridges/
[Accessed 21 Jan 2019].

Nvidia Corporation, 2018. *SLI.* [Online]
Available at: nvidia.com/sli.htm
[Accessed 21 Jan 2019].

Nvidia Corporation, 2019. *FAQ* **|** *GeForce.* [Online]
Available at: https://www.geforce.co.uk/hardware/technology/sli/faq
[Accessed 21 Feb 2019].

Nvidia, 2010. *NVIDIA QUADRO DUAL COPY ENGINES.* [Online]
Available at: https://www.nvidia.co.uk/docs/IO/40049/Dual_copy_engines.pdf
[Accessed 14 Feb 2019].

Nvidia, 2016. *GeForce_GTX_1080_Whitepaper_FINAL.* [Online]
Available at: https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf
[Accessed 25 Jan 2019].

Nvidia, 2017. *volta-architecture-whitepaper.* [Online]
Available at: http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf
[Accessed 19 Jan 2019].

Nvidia, 2018. *NVIDIA-Turing-Architecture-Whitepaper.* [Online]
Available at: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf
[Accessed 19 Jan 2019].

Nvidia, 2019. *NVAPI. [Software].* [Online]
Available at: https://developer.nvidia.com/nvapi
[Accessed 4 APR 2019].

Pabst, T., 1998. [Online]
Available at: https://www.tomshardware.com/reviews/diamond-monster-3d-ii,52-6.html
[Accessed 2 April 2019].

PassMark Software, 2019. *PerformanceTest.* [Online]
Available at: https://www.passmark.com/products/pt.htm
[Accessed 30 March 2019].

PCI-SIG, 2018. *PCI-SIG.* [Online]
Available at: https://pcisig.com/
[Accessed 4 Nov 2018].

Sjoholm, J., 2017. *Explicit Multi-GPU with DirectX 12 – Control, Freedom, New Possibilities.* [Online]
Available at: https://developer.nvidia.com/explicit-multi-gpu-programming-directx-12
[Accessed 9 Feb 2019].

Sjoholm, J., 2017. *Explicit Multi-GPU with DirectX 12 – Frame Pipelining, a New Alternative.* [Online]
Available at: https://developer.nvidia.com/explicit-multi-gpu-programming-directx-12-part-2
[Accessed 9 Feb 2019].

Stardock, 2016. *Ashes of the Singularity.* [Online]
Available at: https://www.ashesofthesingularity.com/
[Accessed 20 Feb 2019].

The Khronos Group, Inc., 1992. *Khronos OpenGL® Registry.* [Online]
Available at: https://www.khronos.org/registry/OpenGL/index_gl.php
[Accessed 11 Jan 2019].

Ung, G. M., 2015. *AMD's Mantle 1.0 is dead; long live DirectX.* [Online]
Available at: https://www.pcworld.com/article/2891672/amds-mantle-10-is-dead-long-live-directx.html
[Accessed 21 Jan 2019].

Ung, G. M., 2015. *Mantle is a Vulkan: AMD's dead graphics API rises from the ashes in OpenGL's successor.* [Online]
Available at: https://www.pcworld.com/article/2894036/mantle-is-a-vulkan-amds-dead-graphics-api-rises-from-the-ashes-as-opengls-successor.html
[Accessed 21 Jan 2019].

Unity Technologies, 2005. *Unity. [Software].* [Online]
Available at: https://unity.com/

University of California, Berkeley, 2019. *Task Graph.* [Online]
Available at: https://patterns.eecs.berkeley.edu/?page_id=609
[Accessed 30 March 2019].

wikichip.org, 2014. *nvlink.* [Online]
Available at: https://en.wikichip.org/wiki/nvidia/nvlink
[Accessed 19 Jan 2019].

WikiChip, 2018. *Ryzen 7 1700X - AMD.* [Online]
Available at: https://en.wikichip.org/wiki/amd/ryzen_7/1700x
[Accessed 27 Feb 2019].

Williams, D. & Smith, R., 2016. *Ashes of the Singularity Revisited.* [Online]
Available at: https://www.anandtech.com/show/10067/ashes-of-the-singularity-revisited-beta/4
[Accessed 21 Feb 2019].

# 13. INDEX OF FIGURES AND TABLES

## 13.1 CODE SAMPLES

## 13.2 FIGURES

## 13.3 TABLES