REAL-TIME RENDERING WITH HETEROGENEOUS GPUS

by

Xiao Lei

A Thesis

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Master of Science

Department of Computer Graphics Technology

West Lafayette, Indiana

May 2020

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Vetria Byrd, Chair

Department of Computer Graphics Technology

Dr. Bedrich Benes

Department of Computer Graphics Technology

Dr. Tim McGraw

Department of Computer Graphics Technology

Approved by:

Dr. Nicoletta Adamo

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

3D	Three-Dimensional
API	Application Programming Interface
CPU	Central Processing Unit
FPS	Frames Per Second
GPU	Graphics Processing Unit
iGPU	Abbreviation of integrated GPU
dGPU	Abbreviation of discrete GPU
GUI	Graphical User Interface
HMD	Head Mounted Display
MB	Megabyte
ms	millisecond
OpenGL ES	OpenGL for Embedded Systems
SDK	Software Development Kit
SPIR	Standard Portable Intermediate Representation
VMA	Vulkan Memory Allocator

ABSTRACT

Lei, Xiao. M.S., Purdue University, May 2020. Real-time Rendering with Heterogeneous GPUs. Major Professor: Dr. Vetria Byrd.

Over the years, the performance demand for graphics applications has been steadily increasing. While upgrading the hardware is one direct solution, the emergence of the new low-level and low-overhead graphics APIs like Vulkan also exposed the possibility of improving rendering performance from the bottom of software implementation.

Most of the recent years' middle- to high-end personal computers are equipped with both integrated and discrete GPUs. However, with previous graphics APIs, it is hard to put these two heterogeneous GPUs to work concurrently in the same application without tailored driver support.

This thesis provides an exploration into the utilization of such heterogeneous GPUs in real-time rendering with the help of Vulkan API. This paper first demonstrates the design and implementation details for the proposed heterogeneous GPUs working model. After that, the paper presents the test of two workload offloading strategies: offloading screen space output workload to the integrated GPU and offloading asynchronous computation workload to the integrated GPU.

While this study failed to obtain performance improvement through offloading screen space output workload, it is successful in validating that offloading asynchronous computation workload from the discrete GPU to the integrated GPU can improve the overall system performance. This study proves that it is possible to make use of the integrated and discrete GPUs concurrently in the same application with the help of Vulkan. And offloading asynchronous computation workload from the discrete GPU to the integrated GPU can provide up to 3-4% performance improvement with combinations like UHD Graphics 630 + RTX 2070 Max-Q and HD Graphics 630 + GTX 1050.

1. INTRODUCTION

1.1 Background

Modern generation graphics APIs like Vulkan and DirectX 12 have certain characteristics in common: low-level and low overhead. For the previous generation of graphics APIs like OpenGL and DirectX 11, graphics drivers handle a major part of task scheduling and resource management, which are hidden from application developers. With the arrival of modern generation APIs, these responsibilities were transferred to application developers. Such change makes the application more complex, but it also brings more explicit control to the application level.

The predecessor of Vulkan, OpenGL, is originally designed for hardware architecture a few decades ago. At that time multi-core and parallelism are not major considerations for personal computing. But nowadays, multi-core and parallelism are common in personal computing devices, which makes the state-machine-based OpenGL less friendly and less efficient towards modern hardware architectures. In recent studies like (Dobersberger, 2015) and (Lujan, Baum, Chen, & Zong, 2019), it has become evident that the old graphics API's mechanism - handling higher-level tasks in drivers, is causing a visible and unnecessary burden on CPU.

Comparing with OpenGL, Vulkan has improvements in various ways (Kenwright, 2017): more explicit control, multi-threading friendly, discrete state access, bindless graphics, resource memory info, resource barrier, and acceleration for applications. All these new features make Vulkan based applications capable of doing a whole lot more than previous graphics applications. The exposure of low-level command and resource control makes resource and task management across vendor-independent graphics hardware become possible, and this inspired the idea of utilizing modern personal computers' heterogeneous GPUs (integrated and discrete) feature to obtain potential rendering performance improvement.

1.2 Problem Statement

With previous graphics API like OpenGL, an application running on a computer equipped with both integrated and discrete GPUs cannot make use of both GPUs without context switching. This makes either the integrated or discrete GPU remain idle while the other one is loaded. Such

application cannot release the computer's full potential performance by enabling all available hardware.

This study is not the first to come up with the idea of utilizing both integrated and discrete GPUs to improve system performance. A study with similar intention has been carried out for HMDs (Peek, Wünsche, & Lutteroth, 2014), in which the researchers utilized integrated GPU to do asynchronous image warping. In another experiment (Yeung, 2015), the authors claimed a noticeable framerate improvement with the discrete plus integrated GPUs combination in the Unreal Engine 4 Elemental demo. However, these studies either did not provide implementation details or did not cover comparisons on different hardware combinations. Furthermore, these studies are all Direct3D based and none of the public studies to date has addressed how the situation would be with Vulkan. There is also the fact that Direct3D is not open and works only on Microsoft platforms, while Vulkan is open and cross-platform friendly.

1.3 Significance of the Problem

Graphics is an essential component for many practical fields, such as scientific visualization, video games, interactive media, and computer-aided design. The performance of the graphics rendering stage significantly impacts the overall performance of the entire system.

Over the years, not only the hardware and developer tools are evolving, but the graphics application fields are also continuously increasing their complexities. For instance, the field of visualization is rapidly growing in both diversity and dimension and such growth is also posing major challenges to visualization tools (Thorvaldsdóttir, Robinson, & Mesirov, 2013).

Some researchers are already exploring in utilizing Vulkan's low-level features to improve application performance. (Zhang, Chen, Johan, & Erdt, 2018) are utilizing Vulkan's parallel opportunities to achieve higher performance city rendering. In the evaluation study conducted on the graphics rendering server (Lujan, Baum, Chen, & Zong, 2019), the results showed that Vulkan can save a significant amount of energy while maintaining the same performance. All these works have proven that the proper use of Vulkan can bring significant improvement to rendering performance.

Although making integrated GPU to assist discrete GPU would not be expected to bring significant improvement to overall performance since the performance of integrated GPU is

usually much lower than discrete GPU and there also exist communication overheads, any potential performance improvement on existing hardware could be appreciated.

1.4 Purpose Statement

The purpose of this study is to identify, design and evaluate the methods that could allow the integrated GPU to assist real-time rendering through offloading workloads from the discrete GPU with Vulkan implementation. And, to measure and analyze the performance difference between the heterogeneous GPUs implementation and the discrete-GPU-only implementation.

This study aims to answer the following research questions:

1. Is it possible to make use of the integrated and discrete GPUs concurrently in the same graphics application with the help of Vulkan?
2. What kind of workload offloading strategy can make better use of this heterogeneous GPUs solution?
3. What performance difference can be expected from this solution and what is the possible explanation?

1.5 Definitions

The following definitions apply to this study.

1. Vulkan: A low-overhead, cross-platform 3D graphics and computing API released by Khronos Group. (Khronos Group, 2019)
2. Integrated GPU: The GPU that shares system memory and usually resides on the same chip as the CPU. (operational definition)
3. Discrete GPU: The GPU that comes with its own dedicated graphics memory and normally exists on a standalone chip. (operational definition)
4. GPU Performance: The speed that the GPU is able to complete certain computational tasks. It is an intrinsic property determined by the hardware. (operational definition)
5. Rendering Performance: The speed that the application is able to generate correct visual contents. (operational definition)
6. Presentation: The process of GPU communicating with display hardware and outputting results to the screen. (operational definition)

1.6 Assumptions

This study is based on several assumptions to reduce variance and narrow down the scope of the topic. These assumptions can be categorized as:

1. Hardware assumptions:
    a) The performance of the integrated GPU is lower than the discrete GPU. By lower, it means it would take a longer time to complete the same job on the integrated GPU than on the discrete GPU.
    b) There is no architectural optimization on the tested computers. Meaning that none of the test computers have any intrinsic advantage of utilizing heterogeneous GPUs.
    c) Potential power consumption increase is acceptable, and performance improvement is prior to power conservation and temperature control.
2. Implementation assumption:
    a) There is no fatal flaw in the design and implementation of the validation program that would invert the results.
3. Test case assumptions:
    a) The test cases are GPU-bound.
    b) The test cases can represent real-life applications to a significant extent.

1.7 Limitations

1. This study assumes that the performance of the integrated GPU is significantly lower than the discrete GPU. The findings may become invalid if the performance of the integrated GPU is close to or higher than the paired discrete GPU.
2. The test cases used in the study will not be able to fully simulate the complex rendering scenario in real industry application, which could make the result over-promising.
3. This study failed to validate that the integrated GPU can improve rendering performance through offloading screen space output workloads from the discrete GPU.
4. The time analysis involved in this study is coarse-grained and can be affected by system environments.
5. Power consumption, memory consumption, and thermal performance are not monitored during the study.

6. The possible performance variation with integrated GPU resource being occupied by other programs is not measured.

1.8 Delimitations

1. This study aims for computers equipped with both integrated and discrete GPUs. It does not apply to devices with multiple discrete GPUs.
2. This study does not focus on the proposal of generic methods that apply to multi-GPU rendering.
3. Due to limited hardware accessibility, the result evaluation only runs on a limited number of personal computer types, and no AMD GPU is tested.
4. The heterogeneous GPUs working model only has an implementation with Vulkan API.
5. Since there already exist more mature version of best practices of Vulkan API, such as (SAMSUNG, 2019) (Subtil, Rusch, & Fedorov, 2019) (Tolo, 2018), this paper would not cover much detail in code-level implementation.
6. This study is not covering comparison with Direct3D or Metal and does not focus on parallel or distributed computing.

1.9 Summary

This chapter introduced the background of this study and stated the problem that remains to be resolved. The significance and purpose of the study were demonstrated. This chapter also listed the definitions and assumptions that the study is basing upon while addressing the limitations and delimitations of the study.

2. REVIEW OF LITERATURE

2.1 Methodology

This study aims to identify, design and evaluate the methods that could allow the integrated GPU to assist real-time rendering through offloading workloads from discrete GPU with Vulkan implementation. Key concepts include multi-GPU rendering, Vulkan API, and high-performance real-time rendering.

The reviewed articles are coming from multiple sources. Databases include ACM Digital Library, IEEE Xplore Library, Eurographics Digital Library, Google Scholar, Journal of Computer Graphics Techniques, and Vulkan Specifications. Key terms relating to multi-GPU, integrated and discrete, Vulkan, and high-performance rendering are used during the search procedure. ACM SIGGRAPH and IEEE Transaction on Visualization and Graphics are the main sources of high-quality literature. Articles from proposal feedbacks are also included. Reference books include Vulkan Cookbook and Game Engine Architecture.

2.2 Supporting Research Problem

In an earlier multi-GPU volume rendering research (Stuart, Chen, Ma, & Owens, 2010), the researchers showed that their proposed system scales with respect to the number of GPUs if given enough work. Similarly, in the recent study of multi-GPU rendering with Vulkan API (Tolo, 2018), the author presented that the multi-GPU approach could increase the overall performance by reducing the amount of work for each GPU when there are sufficient workloads. In the future work section, Tolo mentioned that “performance can be improved by implementing optimizations for increasing the GPU utilization” (Tolo, 2018, p. 68).

In the technical blog (Yeung, 2015) which has a very similar topic with this proposed study, the author introduced the multi-adapter feature supported by DirectX 12 and its application with integrate plus discrete GPUs system. Although the performance gain is demonstrated in the blog, the portion of offloaded postprocessing work is not provided, and whether offloading other workloads can further increase overall performance is not discussed.

Another integrated GPU utilization study was conducted for HMDs (Peek, Wünsche, & Lutteroth, 2014). The research employed Direct3D 11 to implement the system that utilized

integrated GPU to do asynchronous image warping. Their system can perform a warp in 4.2ms at 1920×1080 resolution on an Intel HD Graphics 2000 GPU at the cost of a 1.5ms increase in application render time. Furthermore, the authors identified that the usage of system resources' impact on the performance should be further investigated.

In an exploratory study of the Vulkan API (Shiraef, 2016), based on multiple available benchmarks, the author concluded that Vulkan does provide performance benefit over OpenGL and Vulkan based application is less CPU bound comparing to OpenGL application. However, the margin of improvement at the time (2016) was still minimal.

2.3 Supporting Research Purpose

A recent study evaluated the performance and energy efficiency of OpenGL ES and Vulkan (Lujan, Baum, Chen, & Zong, 2019). The authors not only demonstrated that Vulkan can reduce more than 50% of CPU power consumption without degrading performance, but also that when power is not a limiting factor, Vulkan is able to achieve 3 times of frame rate than OpenGL ES. Although their study is targeted for rendering servers, these findings would still be valid enough to show that choosing Vulkan to improve rendering performance is the right direction.

The multi-GPU rendering study (Tolo, 2018) mentioned in 2.2 Supporting Research Problem demonstrated how to implement platform and vendor-independent multi-GPU rendering by using Vulkan to send commands explicitly to separate GPUs. Besides, in Tolo's study, an abstraction library that supports multi-GPU application was also provided. This study proves that the concept of utilizing Vulkan to combine the use of GPUs from different vendors is viable.

In an earlier study regarding load-balancing in multi-GPU systems (Chen, Villa, Krishnamoorthy, & Gao, 2010), the authors proposed a task-based dynamic load-balancing solution for single and multi-GPU systems implemented as task queue scheme, which avoids using expensive synchronization locks. On the multi-GPU system, their solution was able to achieve "near-linear speedup, load balance, and significant performance improvement over techniques based on standard CUDA APIs" (Chen, Villa, Krishnamoorthy, & Gao, 2010, p. 1). Their study supports the purpose that specific task distribution strategies could be identified to make better use of the heterogeneous GPUs solution.

Theoretical estimation of the heterogeneous GPUs model performance can be partially supported by Amdahl's Law, which has been restated and improved in many studies like (Hill &

Marty, 2008). This law can provide a theoretical speedup estimation model for parallel programs. For instance, if 10% of the task can be taken out from discrete GPU and the affected part becomes roughly 1.5 times as fast, the overall estimated speedup will be about 3.4%. Such figure may not exciting but is anyway an improvement to the device performance.

Coincidentally, right before this paper finished writing, Intel presented their multi-adapter solution with integrated plus discrete GPUs in GDC 2020 (Hux, 2020). In their presentation, the integrated GPU is used to speed up the asynchronous computation for nBody particle rendering with DirectX 12 backend implementation. Their presentation is a great complement to the topic that this paper is focusing on.

2.4 Supporting Research Methodology

Vulkan based high-performance application exploration are already emerging.

In a quantitative visualization research (Mustafin, Almaty, Akhmed-Zaki, & Turar, 2019), the researchers developed and presented a prototype of visualizer, in which they proposed a real-time visualization of large grid models with using of Vulkan API on the typical personal computers. For implementation, double buffering for vertex buffer and C++ 11 multithreading for drawing and copy commands are adopted. As input data, the results of each 100th iteration of the Jacobi method for solving the Poisson equation with 2D and 3D grids were taken. Their results supported that the double vertex buffering and multi-threading approach to the visualization of the grid model optimized the speed of the application. However, due to the quasi-experimental design nature of their study, their results are not rigor.

In the high-performance city rendering research (Zhang, Chen, Johan, & Erdt, 2018), the authors utilized Vulkan's parallel opportunities to achieve higher rendering performance. In their proposed rendering system, three operating parts are running concurrently. These three parts are rendering, recording draw commands and streaming texture images respectively. To ensure valid synchronization for command buffer recording, double command buffering was employed. Other approaches like view culling, texture streaming and texture compressing were also combined. As a result, they achieved a relatively high drawing performance under the intensive draw calls and massive data situation ("At 1920×1080 resolution, a view submitting 302K, 723K, 5.04M draw calls runs at 119FPS, 57FPS, 12FPS respectively." (Zhang, Chen, Johan, & Erdt, 2018, p. 2)).

Also, in 2018, in a research conducted on rendering framework for mobile multimedia (Gambhir, Panda, & Basha, 2018), the researchers presented the first Vulkan-based animation and effects engine for mobile video rendering. For 4K (3840×2160 resolution) videos playback at 30 FPS, the study observed "an increase of 30% in frame-rate, a decrease of 30% in memory consumption and export time, and a decrease of 20% in power consumption" (Gambhir, Panda, & Basha, 2018, p. 2), comparing with OpenGL ES. In their Vulkan based solution, methods that helped in contributing to the final performance include precompiled SPIR-V shaders, reusing recorded command buffer, staging buffer with device local memory, and avoiding unnecessary format conversions.

In the technical blog (Yeung, 2015) which shares a similar topic with this study, the author briefly explained that their solution is offloading some of the post-processing work to the integrated GPU and make both GPUs work in parallel. In the showcase examples provided alongside with the feature highlight, their discrete plus integrated combination beat the discrete-only setting by around 10% framerate performance (35.9 FPS vs. 39.7 FPS).

In the multi-GPU volume rendering research (Stuart, Chen, Ma, & Owens, 2010), the researchers suggested that if a GPU during the process is connected to a display, it would be more efficient to allow that exact GPU to output the image immediately after completing the final composition. Such suggestion is in accordance with the methodology design adopted in this study.

In the multi-threading evaluation study of Vulkan (Blackert, 2016), according to the results produced by the test program implemented both in OpenGL and Vulkan, a performance increase ranging from 8% and 69% can be observed depending on whether currently the application is CPU bound or GPU bound. This evaluation study demonstrated that if the application is CPU bound, using multiple threads for command buffer recording can significantly increase the performance of the application.

2.5 Summary

This chapter described the methodology that has been used in the review of literature process of this study. A list of articles supporting this study from the perspective of problem statement and purpose correctness is provided. Following that, several recent research that can either directly or indirectly inspire the methodologies that can be adopted in this study is examined.

3. METHODOLOGY

3.1 Research Type

The research presented in this paper is applied research, containing a mixture of exploratory and developmental elements. This research has the intention to establish priorities and provide improvement for future research design in the heterogeneous GPUs application field. In the meantime, this research holds a major part in proposing and evaluating methods that deal with utilizing heterogeneous GPUs in real-time rendering.

3.2 Development Tools

Some key development tools involved in this study include:

1. Vulkan SDK:

   The Vulkan Software Development Kit (SDK) provides the development and runtime components for building, running, and debugging Vulkan applications. This comprehensive SDK includes the Vulkan loader, Vulkan layers, debugging tools, SPIR-V tools, the Vulkan run time installer, documentation, samples, and demos. (LunarG, 2019)

2. GLFW:

   GLFW (Graphics Library Framework) is an Open Source, multi-platform library for OpenGL, OpenGL ES and Vulkan development on the desktop. It provides a simple API for creating windows, contexts, and surfaces, receiving input and events. (The GLFW Development Team, 2019)

3. RenderDoc:

   RenderDoc is a free MIT licensed stand-alone graphics debugger that allows quick and easy single-frame capture and detailed introspection of any application using Vulkan, D3D11, OpenGL & OpenGL ES or D3D12 across Windows 7 - 10, Linux, Android, Stadia, or Nintendo Switch. (Karlsson, 2019)

4. Visual Studio:

   Visual Studio is an integrated development environment (IDE) from Microsoft. It is used to develop computer programs, as well as websites, web apps, web services and mobile

apps (Wikipedia, 2019). It also provides basic profiling components for application development.

5. Vulkan Memory Allocator:

   The Vulkan Memory Allocator (VMA) library provides a simple and easy to integrate API to help allocating memory for Vulkan buffer and image storage. (GPUOpen, 2019)

6. SPIRV-Cross:

   SPIRV-Cross is a tool designed for parsing and converting SPIR-V to other shader languages. It also provides reflection API to simplify the creation of Vulkan pipeline layouts and modify and tweak OpDecorations. (Khronos Group, 2019)

7. GLM:

   GLM (OpenGL Mathematics) is a header-only C++ mathematics library for graphics software based on the OpenGL Shading Language (GLSL) specifications. (G-Truc, 2019)

The 3D models used during the test and validation process were obtained from McGuire Computer Graphics Archive (McGuire, 2017), The Stanford 3D Scanning Repository (Stanford Computer Graphics Laboratory, 2014), and Unity-Chan official resources (Unity Technologies Japan G.K., 2015).

3.3 Test Program Design

A demo real-time rendering program is developed based on OpenGL, Vulkan, and C++ for evaluation purposes. This program resembles a pure graphics engine and supports simple and complex 3D objects rendering with multiple post-processing techniques.

For early workflow validation and baseline performance comparison, an OpenGL counterpart exists in this implemented benchmark program, as shown in Figure 1. The Vulkan implementation and OpenGL implementation are abstracted as two independent drawing devices that can be controlled by the upper level of the program, but only one of them is activated at runtime. The abstracted drawing devices provide necessary communication interfaces with underlying graphics API, while the rendering logic and render resources are managed by renderers and upper application.

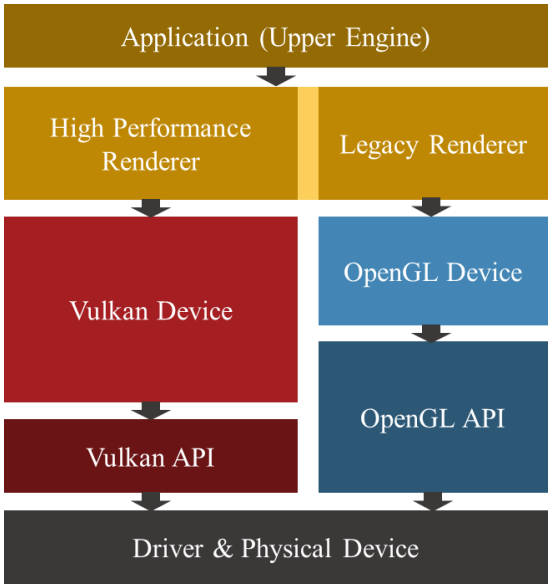


Figure 1. Structure of the demo program developed in this study

3.4 Communication Between GPUs

Vulkan has provided the interfaces to query available physical devices and wrap them up as ready-to-use logical devices. The integrated GPU (denote as iGPU) and the discrete GPU (denote as dGPU) can be encapsulated as two logical devices inside the Vulkan Device implementation, as illustrated in Figure 2.

Because the resources created on different logical devices are not interchangeable, the heterogeneous GPUs renderer manages two sets of resources: one set used by the dGPU and the other set used by the iGPU. Data from one GPU is transferred to the other GPU through explicit buffer copying.

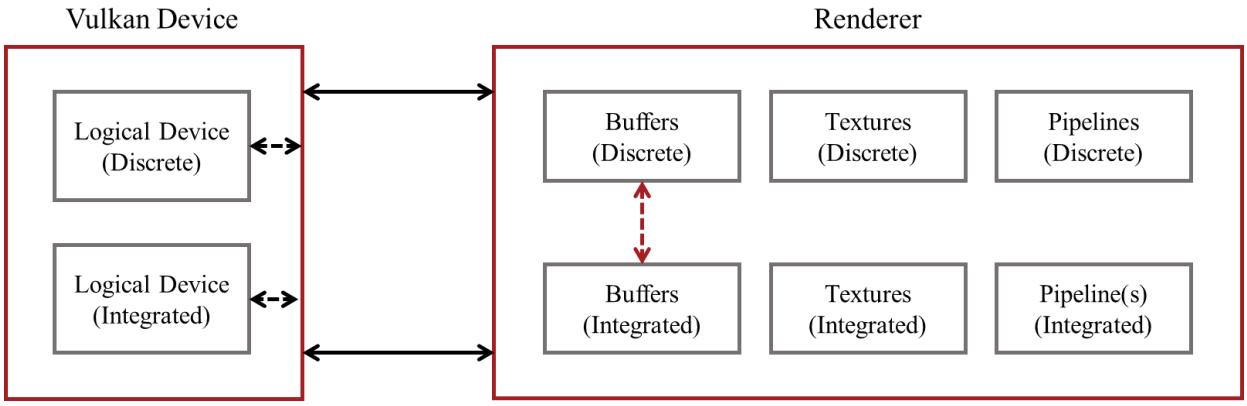


Figure 2. Logical device abstraction and resource management of the two GPUs

Two types of heterogeneous GPUs working models are designed and tested in this study. (1) The iGPU is directly participating in screen space output. More specifically, the iGPU is offloading the post-processing pass workload from the dGPU. (2) The iGPU is opted out of the screen space output pipeline and instead works on asynchronous computation.

The main reason to offload post-processing from dGPU is that the post-processing process has a predictable workload and resource consumption. If the iGPU is handling the beginning sections of the render process, it would also require allocating identical vertex buffers and possibly textures in system memory for every possible object in the scene, which is not desirable.

For the first type of working model, the data communication schema is demonstrated in Figure 3. Under this working type, the read back from graphics memory to system memory is enabled through staging buffers, and the iGPU also takes over the presentation responsibility. The reason why presentation should be handled by the iGPU is that: since the iGPU is handling the last sections of the render process, it should not write results back to dGPU memory as this would double the data transfer overhead for almost nothing in return.

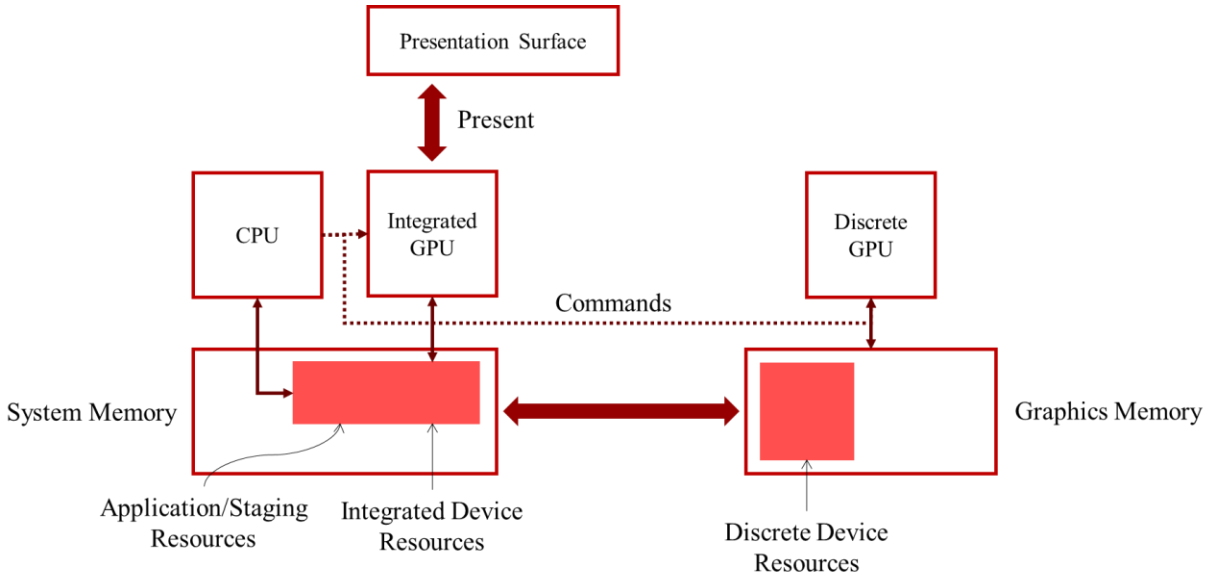


Figure 3. Integrated GPU directly participates in screen space output

For the second type of working model, the data communication schema is demonstrated in Figure 4. Under this working type, the presentation responsibility is returned to dGPU. There will

be no read back from the graphics memory as all data transfers are unidirectional from system memory to graphics memory.

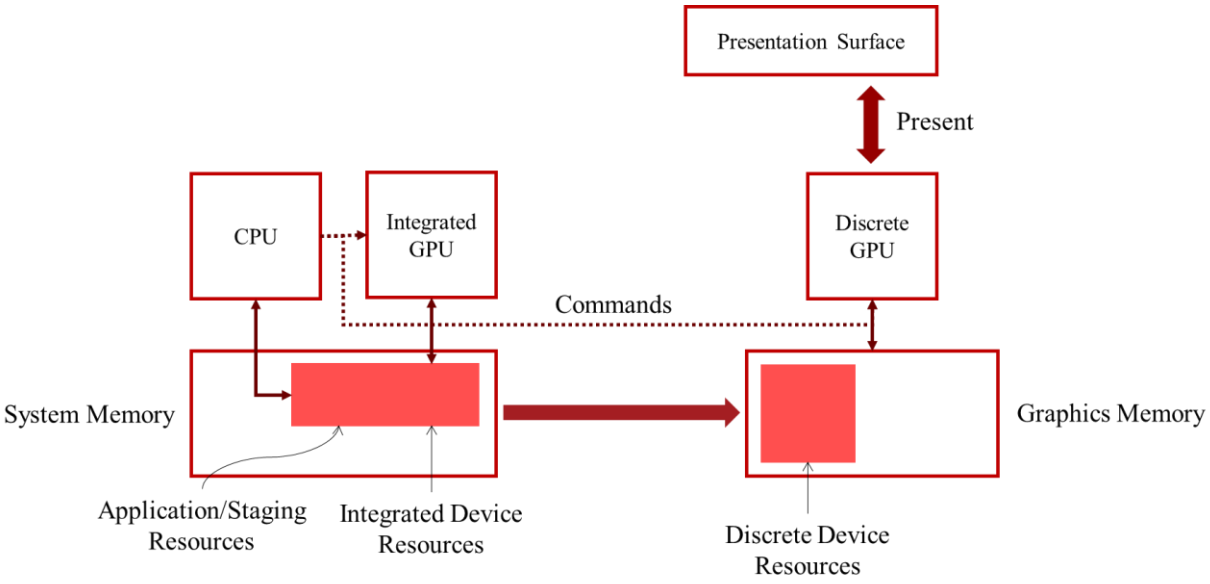


Figure 4. Integrated GPU works on asynchronous computation

3.5 Synchronization Primitive Selection

Vulkan has several built-in synchronization primitives: event, semaphore, and fence, each corresponding to a different granularity. Event provides synchronization between submitted commands in the same queue. Semaphore provides synchronization between commands submitted to different queues. Fence provides synchronization between GPU and CPU.

Fence and semaphore before Vulkan 1.2 in Vulkan core features are binary synchronization primitives, meaning that they only have two states: signaled and unsignaled. There are additional restrictions on these synchronization primitives making them less favorable to use: both semaphore and fence cannot be waited on before it is submitted to the queue, and fence cannot be used simultaneously across multiple threads.

With the arrival of Vulkan 1.2 in early 2020, a new type of semaphore – timeline semaphore became part of the core features of Vulkan. The timeline semaphore contains a superset of both the original semaphore and fence primitives while eliminating most of the unfavorable restrictions on the original primitives. It contains a monotonically increasing 64-bit integer value as its state, and both wait-before-submit and use across multiple threads behaviors are supported.

The timeline semaphore also has its restrictions and the most obvious one is that it is not compatible with window-system, meaning that any synchronization with presentation surface still needs to be done with traditional semaphore. Despite such restriction, timeline semaphore is widely used in the development process of this study as synchronization primitive.

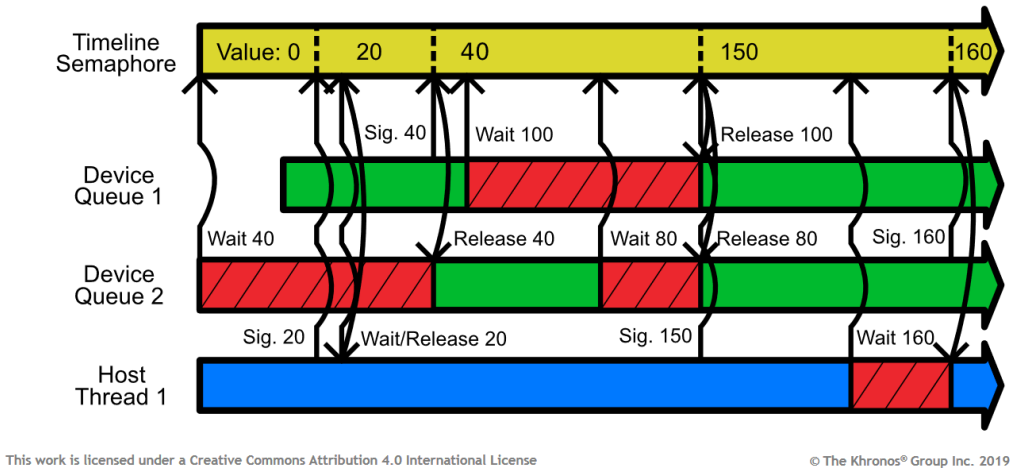


Figure 5. New Model: Timeline Semaphore Primitive. From *Introducing Timeline Semaphores*, by J. Jones, 2019, (Jones, 2019). Copyright 2019 by The Khronos Group Inc. Licensed under CC BY 4.0. No modification was made.

3.6 Render Graph

A render graph is an acyclic graph that describes the render passes behaviors and organizes their dependencies. It is particularly useful for low-level APIs like Vulkan as task scheduling for each pass can be easily assigned in parallel. The render graph mechanism implemented in this study is a primitive one, similar to the render graph described in (Persson, 2017). The pass dependencies still need to be specified explicitly by assigning immediately connecting pass(es), and automatic transient resources management as described in (Arntzen, 2017) is not yet implemented in this study.

The basic unit of the render graph is pass node. Each pass node composites of 5 types of components: node properties, render context, input resources, output resources and render pass function(s). Node properties include information like node identifier. Render context contains data such as associated renderer, command pools and list of objects to process. Input and output resources contain resources accessed during the pass, including images and buffers. Render pass function(s) defines the how operations should be executed during the render pass.

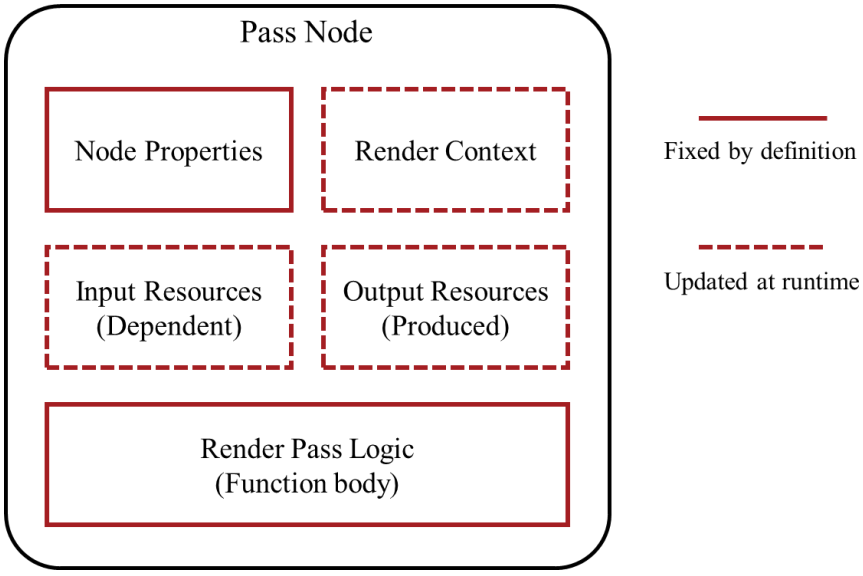


Figure 6. Composition of a node in the render graph

This study adopted several non-photorealistic rendering techniques to simulate a real-life graphics application, including: Toon Shading with Stylized Control based on (Barla, Thollot, & Markosian, 2006) and (Vanderhaeghe, Vergne, Barla, & Baxter, 2011); Surface Feature Enhancement based on (Vanderhaeghe, Vergne, Barla, & Baxter, 2011), (Vergne, Pacanowski, Barla, Granier, & Schlick, Radiance Scaling for Versatile Surface Enhancement, 2010) and (Vergne, Pacanowski, Barla, Granier, & Schlick, Light Warping for Enhanced Surface Depiction, 2009); Line Drawings via Abstracted Shading based on (Lee, Markosian, Lee, & Hughes, 2007). The full render graph constructed in this study is demonstrated in Figure 7.

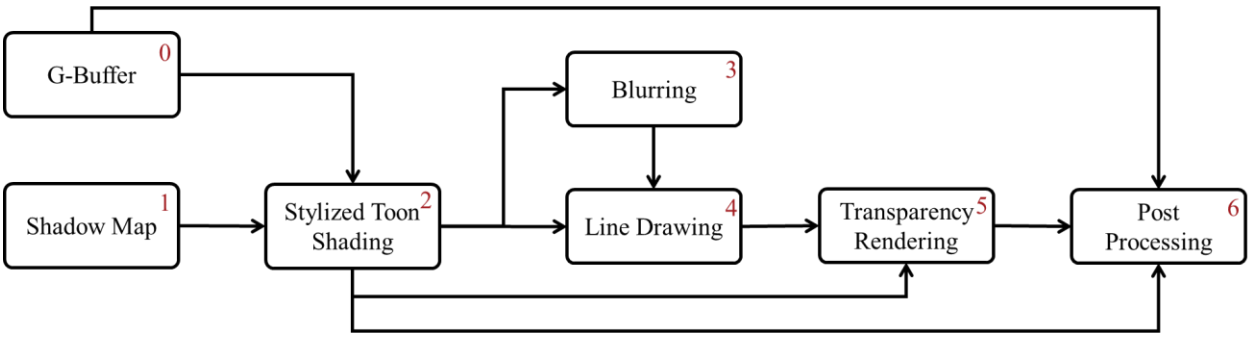


Figure 7. Render graph for single GPU (without asynchronous computation test)

The execution priority of the nodes can be determined by render graph traversal. The traversal time complexity is O(*n*), *n* is the number of nodes in the graph. This process only needs to run once at the very beginning, or every time the render graph is modified. The pseudo-code for the traversal process is provided below.

```
Render Graph Traversal
foreach node in render graph
    if node has no dependent node
        add node to starting node list
    end
end
foreach node in starting node list
    traverse(node)
end

function traverse(node)
if node has no dependent node or all dependent node was visited
    mark node as visited
    assign priority p to node
    increase p
else
    return
end
foreach node' dependent on node
    traverse(node')
end
```

After running the traversal on the given render graph, each node's priority can be derived as shown in Figure 7. Note that these priority values do not represent absolute execution sequence. The execution sequence would be determined by priority dependencies as depicted in Table 1.

Table 1. Node dependencies reflected in priorities

Priority	Dependent Priorities
0	(none)
1	(none)
2	{1, 2}
3	{2}
4	{2, 3}
5	{2, 4}
6	{0, 2, 5}

For each node that is pending execution, the renderer would first check whether all its dependent nodes have finished command recording or have already submitted for execution. The recorded command buffer(s) for the node would only be submitted to GPU if the condition is satisfied. Applying this mechanism to the render graph shown in Figure 7, the execution sequence of nodes with priority 0 and 1 are exchangeable, while the rest are pretty much sequential.

3.7 Render Graph for Heterogeneous GPUs

In this study, the render graph design itself does not change under heterogeneous GPUs mode, but rather the two GPUs would each own an exclusive render graph as shown in Figure 8 and Figure 9.

The consideration behind such arrangement is that because the two GPUs have non-interchangeable resources that must be explicitly transferred in the CPU side, there will be little benefit from mixing them in the same render graph. More importantly, for post-processing offloading as discussed in section 3.4, the integrated GPU would be working under a different cycle from the discrete GPU, which is further explained in section 3.10. Having only one render graph could not achieve the desired working model.

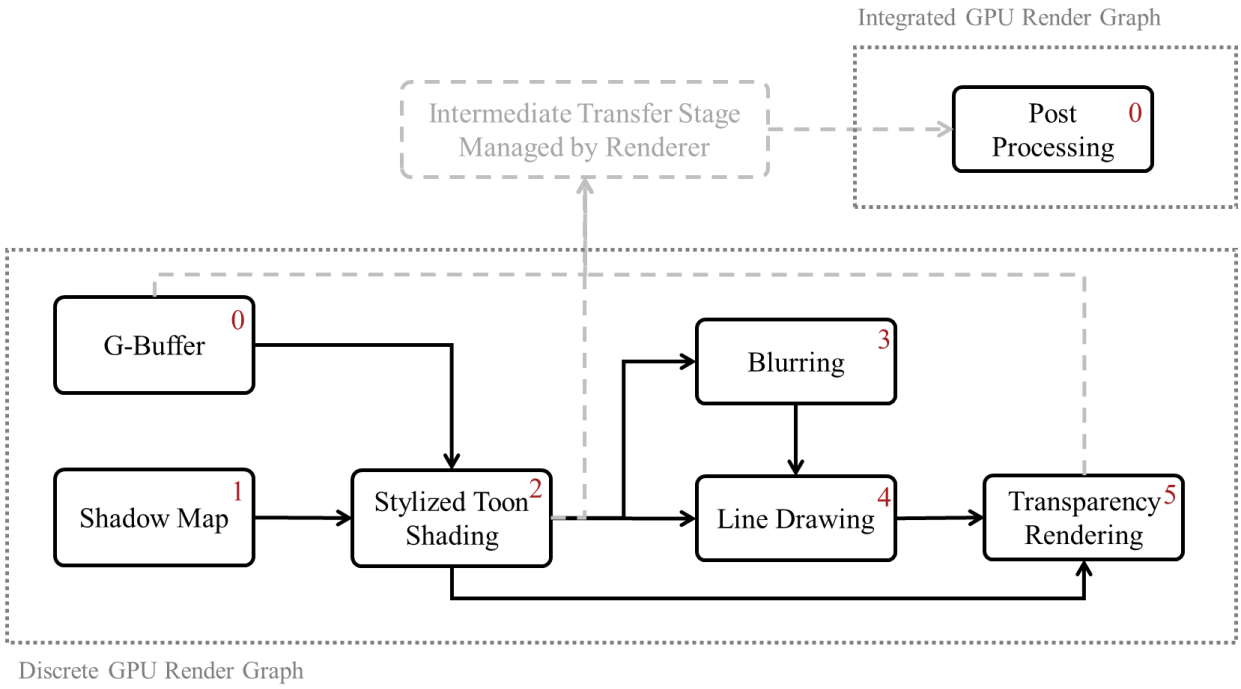


Figure 8. Render graphs of heterogeneous GPUs for offloading post-processing

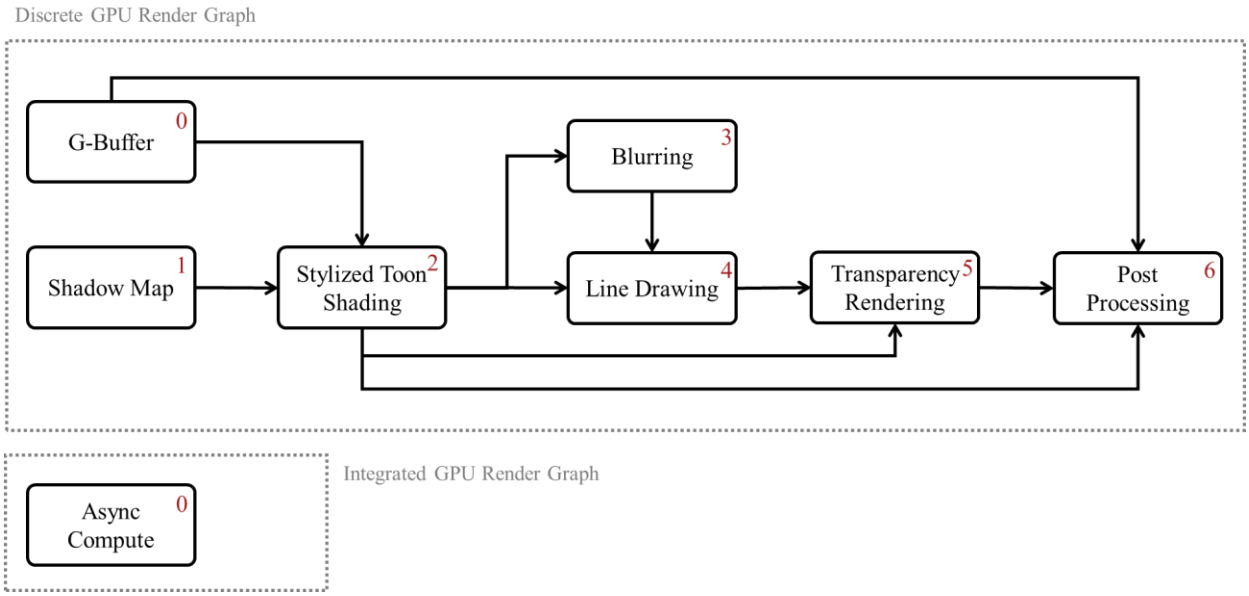


Figure 9. Render graphs of heterogeneous GPUs for offloading asynchronous computation

3.8 Multi-threading Design for Single GPU

One of the valuable features that APIs like Vulkan provide to the graphics developers is the improved compatibility with application-level multi-threading. This is also one of the major advantages Vulkan has over OpenGL.

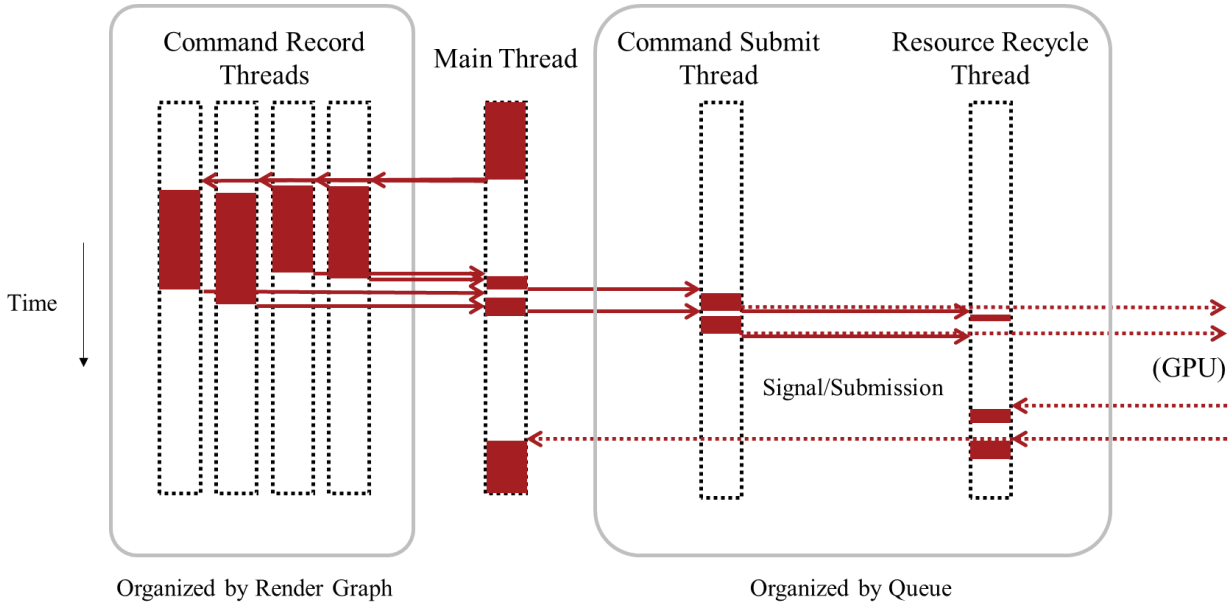


Figure 10. Multi-threading design for single GPU

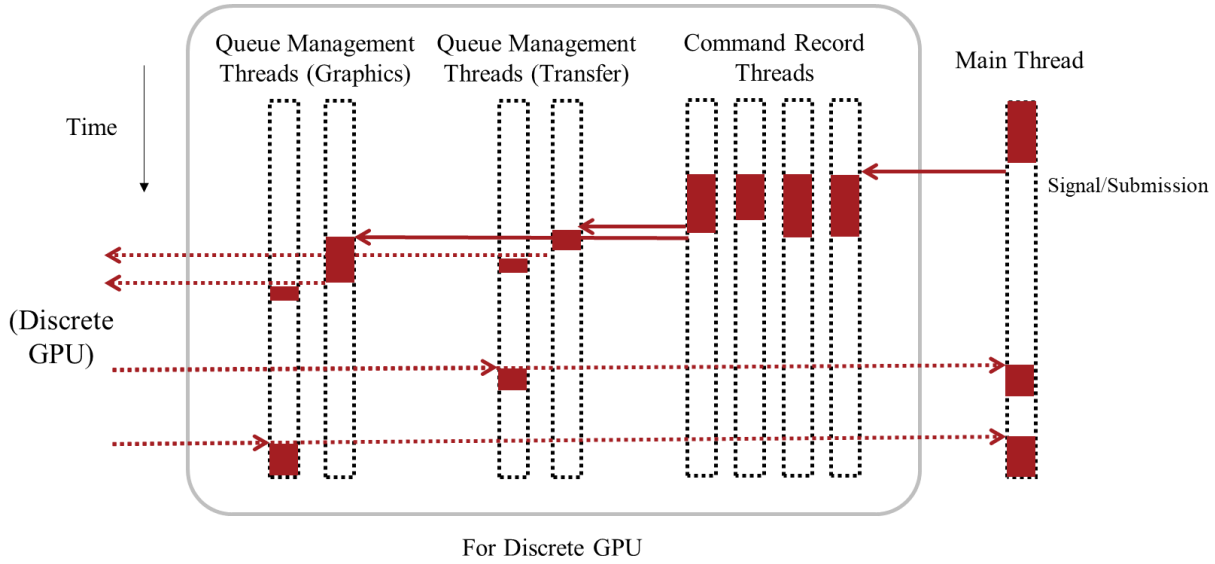


Figure 11. Multi-threading design for single GPU with asynchronous computation workload

With the help of render graph, multi-thread command recording can be easily implemented in the existing application. During one frame, each command-recording thread takes one pass node from the render graph by priority sequence and executes the pass function. All the command recording threads hold individual command buffer pools for command buffer allocation. The number of command-recording threads is configurable during renderer setup.

Two separate threads are utilized to manage command buffer submission and command resource recycling on each enabled type of queue. Transfer queue is also enabled when data transfer from graphics memory to system memory is involved. Note that the transfer queue only exists in dGPU, and on the tested Intel iGPUs there is no separate transfer queue.

Figure 12. Command submission sequence is supervised by the renderer (main thread)

As addressed in section 3.5, the recorded command buffers for each render pass need to submit in a certain sequence. Since recording with multiple threads, the finished command buffers may arrive in any sequence, the renderer running on the main thread / cycle management thread is responsible for checking and holding each arrived command buffer and ensuring they are submitted in the correct sequence.

For example, for a render graph shown in Figure 7, four command-recording threads take nodes with priority marked with 0-3 (denote by node 0-3) at the beginning. After some time, command buffers for nodes 1 and 2 finished recording. The main thread would check the dependency generated by the render graph, then determine that the command buffer for node 1 will be submitted immediately while the command buffer for node 2 should be retained as node 0 is not finished yet. The two command-recording threads continue to work on the tasks for nodes 4 and 5. After another while, command buffer for node 0 finished recording, and the main thread would then determine to submit command buffers for both node 0 and node 2 together in sequence.

3.9 Multi-threading Design for Heterogeneous GPUs

The multi-threading design for heterogeneous GPUs in this study is basically an extension to the design for single GPU case.

However, for offloading post-processing, there is a major difference between the two designs: the discrete GPU's work cycle would be managed by a dedicated thread other than the main thread. The main thread is in synchronization with presentation and thus bounded with integrated GPU. A detailed explanation is provided in section 3.10.

Transfer queue will be enabled to submit commands dealing with coping image outputs to staging resources.

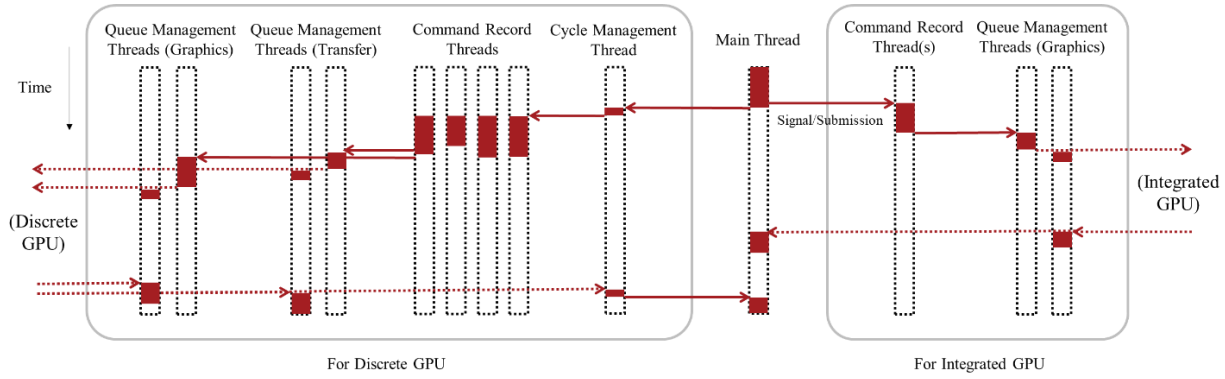


Figure 13. Multi-threading design for heterogeneous GPUs with offloading post-processing

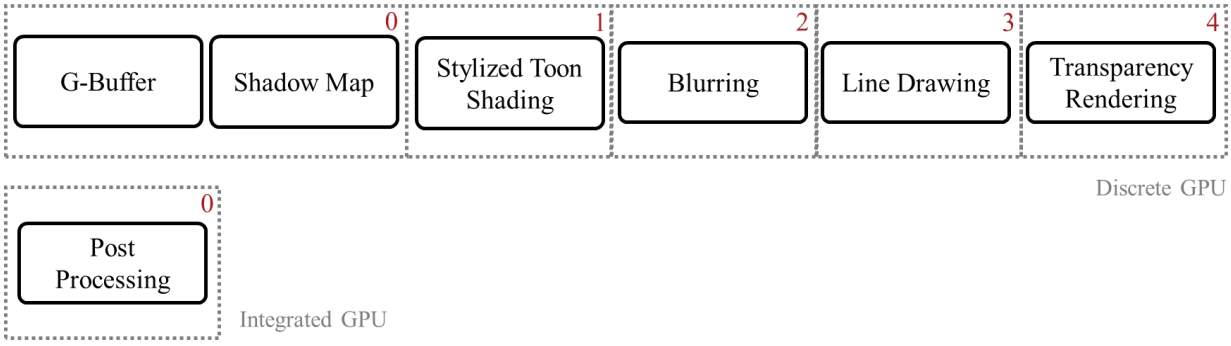


Figure 14. Command submission sequences are separate for two GPUs

For offloading asynchronous computation, the design would pretty much be an extended version of the single GPU case as illustrated in Figure 15. As the two GPUs work under the same cycle.

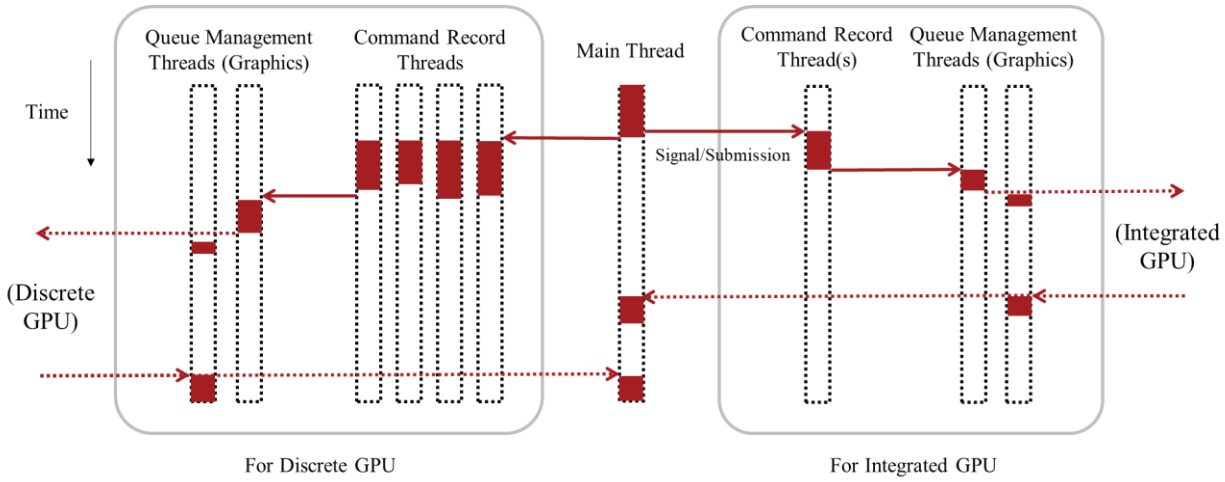


Figure 15. Multi-threading design for heterogeneous GPUs with offloading asynchronous computation

3.10 Work Cycle Design for Heterogeneous GPUs

Since it would take the iGPU longer time to complete the same task than the dGPU, it would be unwise to insert iGPU into dGPU's workflow directly.

When iGPU participates in screen space output, the solution adopted in this study is to separate two GPUs into two different work cycles, as illustrated in Figure 16. Discrete GPU works on a dedicated cycle that only involves rendering the contents from the beginning of a frame to everything before post-processing steps. Integrated GPU works on post-processing steps and everything involved in presentation. Since the iGPU directly controls when a frame is finally presented, the main thread is in synchronization with the iGPU, thus the dGPU would be managed through a dedicated cycle management thread.

The initial workflow is kick-started through enqueuing render tasks of both frame 0 and frame 1 to the dGPU. When dGPU finished rendering contents for frame 0, iGPU can continue its cycle for post-processing and presenting frame 0, while dGPU continue to work on frame 1.

When the iGPU works on asynchronous computation, the main thread will be in synchronization with the dGPU again. And as the iGPU is expected to complete its job inside the

same frame, two GPUs now work under the same cycle as illustrated in Figure 17. Therefore, there is no need for a dedicated cycle management thread for the iGPU.

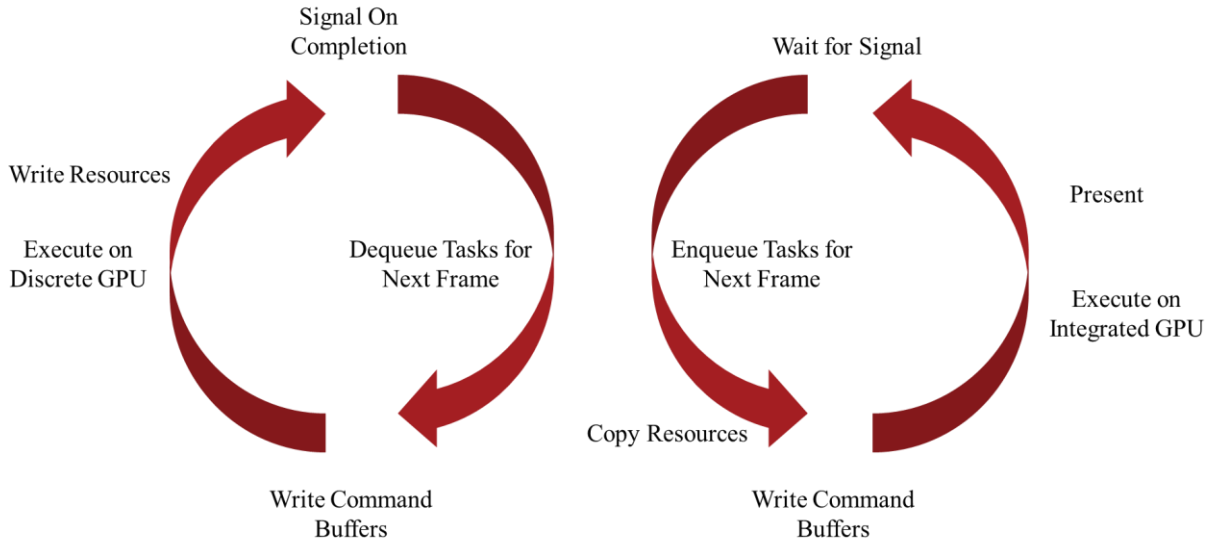


Figure 16. Two GPU work cycles with offloading post-processing

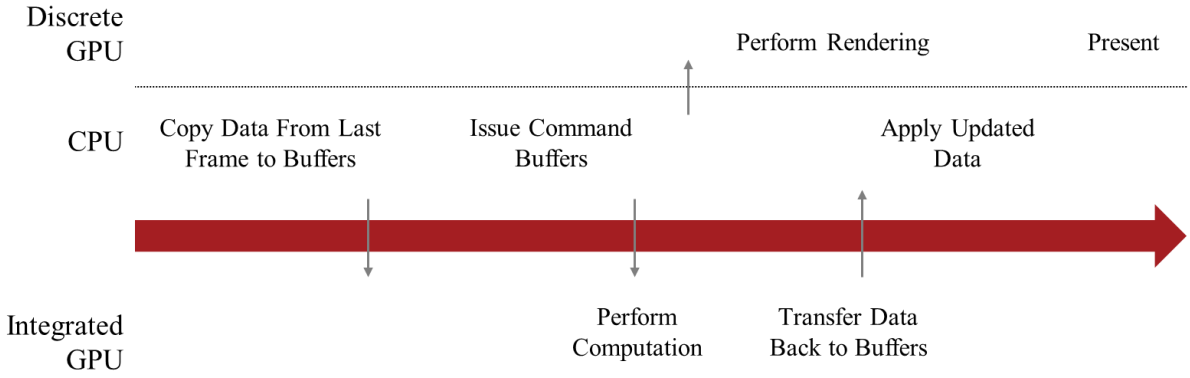


Figure 17. Two GPUs work under the same cycle when offloading asynchronous computation

3.11 Test Procedure Design

Two personal computers with different hardware combinations were used during the test. Their detailed specifications are given in Table 2. Theoretical performance data of the discrete GPUs was retrieved from the TechPowerUp website (TechPowerUp, 2020) (TechPowerUp, 2020).

Theoretical performance data of the integrated GPUs was retrieved from the WikiChip website (WikiChip, 2020) (WikiChip, 2020).

Table 2. Specifications of the testing devices

	Computer A	**Computer B**
CPU	Intel Core i7 9750H	Intel Core i5 7300HQ
Discrete GPU	NVIDIA GeForce RTX 2070 with Max-Q Design	NVIDIA GeForce GTX 1050
Discrete GPU Theoretical Performance	5.460 TFLOPS	1.911 TFLOPS
Graphics Memory Transfer Rate	12.0 GB/s	7.0 GB/s
Integrated GPU	Intel UHD Graphics 630	Intel HD Graphics 630
Integrated GPU Theoretical Performance	441.6 GFLOPS	384.0 GFLOPS
System Memory	DDR4 2666 16GB 64-bit × 2	DDR4 2133 8GB 64-bit
System Memory Transfer Rate	42.66 GB/s	17.05 GB/s
Bus	PCI Express ×16 Gen3	PCI Express ×8 Gen3
Discrete GPU Diver Version	26.21.14.4236	26.21.14.4236
Integrated GPU Driver Version	26.20.100.7463	26.20.100.7870
Vulkan Runtime	1.2.131.2	1.2.131.2
Operating System	Windows 10, 1903	Windows 10, 1903

Both computers were configured to “performance” thermal mode to produce maximal and consistent hardware throughput. All background applications except for system services were cleared during the test.

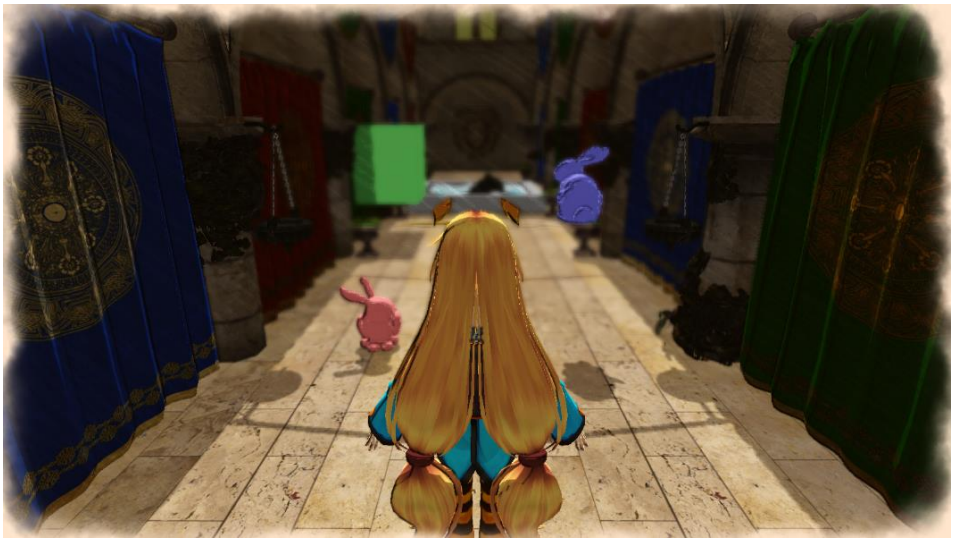


Figure 18. Test scene for offloading post-processing

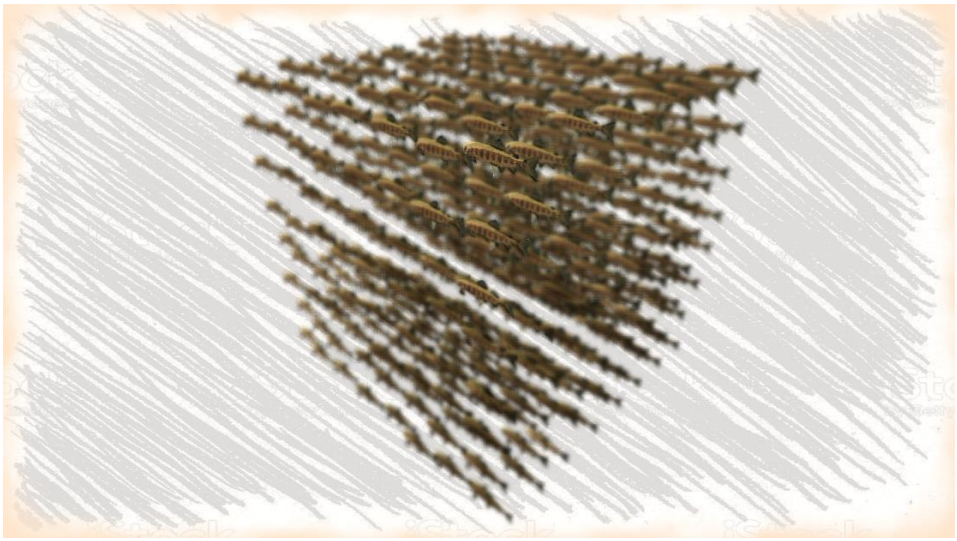


Figure 19. Test scene for offloading asynchronous computation

For offloading post-processing, a scene with around 588K triangles was used during the test, as shown in Figure 18. This scene generates 200 draw calls per frame, and a total of 237.75 MB textures were used. The post-processing pass includes a primitive depth of field effect and a screen space overlay shadow animation.

For the offloading asynchronous computation, a scene with around 3.24M triangles was used during the test, as shown in Figure 19. This scene contains 512 objects and generates 1547 draw calls per frame. In each frame, the positions of all the objects are put into asynchronous update

while the original values are used for rendering in the current frame. The position data are written and updated as a 32×16 texture in the GPU.

Performance was measured from both inside and outside the program. From inside the program, the total number of frames generated in 20 seconds is used to compute average frame time. From outside the program, RenderDoc and Visual Studio Diagnostic Tools were utilized to capture and analyze program performance. When measuring performance internally, all the external tools were disabled.

All tests were conducted under 1600×900 resolution with no framerate limit.

3.12 Summary

This chapter demonstrated the development tools and test cases used in the study. More importantly, this chapter presented feature design and methods adopted in the implementation and evaluation process of the study.

4. RESULTS AND DISCUSSION

4.1 Theoretical Analysis

The upper bound of performance improvement can be calculated by adding up the theoretical performance of both GPUs. For Computer A this upper bound would be 8.08%, and for Computer B this upper bound would be 20.09%.

Before performing a theoretical analysis on the optimal throughput of the system, an analysis of each pass/step of the rendering process is needed. The data required for the analysis were collected through RenderDoc on Computer A with dGPU running the program. The time consumption breakdown for each pass is shown in Figure 20 and Figure 21.

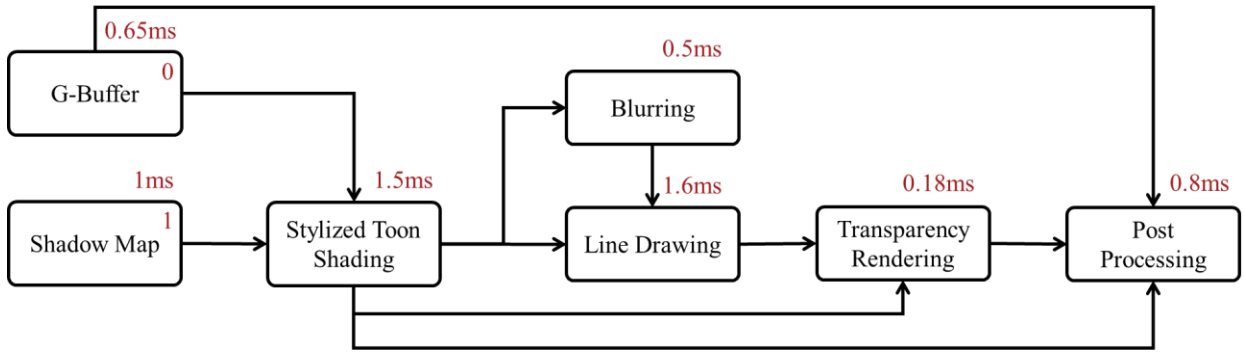


Figure 20. Time consumption breakdown of each pass for dGPU

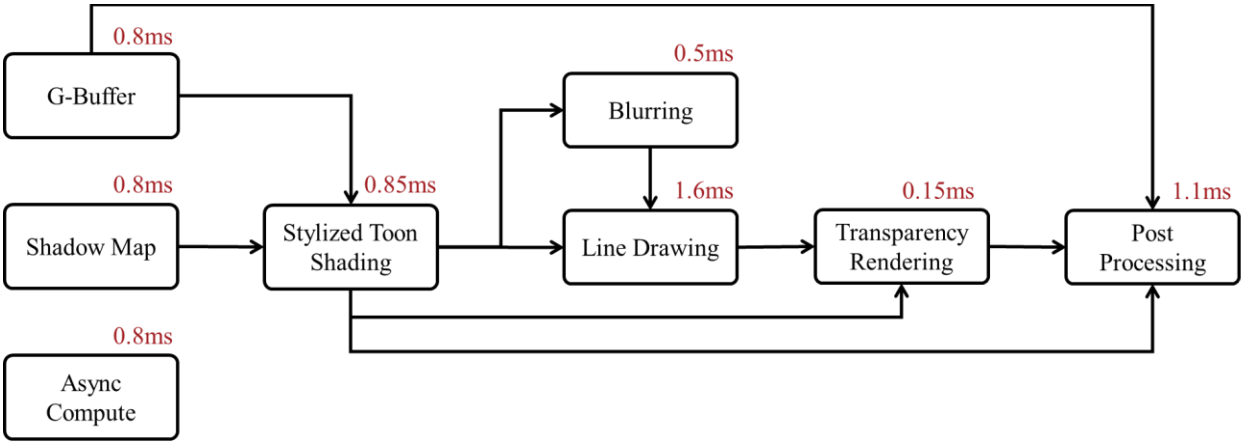


Figure 21. Time consumption breakdown of each pass for dGPU (continued)

From the collected data, it can be estimated that the post-processing step takes up about 12.8% of the total render passes time in offloading post-processing test. The asynchronous compute step takes up about 12.1% of the total render passes time in the asynchronous computation test.

When the post-processing pass is offloaded onto the iGPU, a texture data transfer overhead must append to the total time consumed by the dGPU.

The size of an uncompressed 1600×900 texture with RGBA 32-bit float format (the output from transparency rendering pass) is about 21.97 MB. On Computer A with optimal system memory bandwidth and minimal latency, it would take approximately 0.51ms to copy the texture from graphics memory to the system memory. Only one texture's copy time is calculated because optimally the transfer overhead of the other two input textures can be hidden by initiating transfer immediately after each pass, then only the last texture has an inevitable transfer overhead.

On Computer B, the post-processing step takes approximately 2.4ms to complete, while transferring the same 1600×900 texture to system memory would ideally take 1.29ms.

According to Amdahl's law, the overall speedup of the system can be estimated by:

$$Speedup(f, s) = \frac{1}{(1 - f) + \frac{f}{s}}$$

Where f is the fraction of computation that is enhanced, and s is the speedup of that fraction.

When applying Amdahl's law in the estimation, the transfer overhead can be treated as the remainder of the post-processing pass. For Computer A, the speedup s for the offloaded part would be 1.56, and the estimated overall speedup is therefore 1.048. For Computer B, the speedup s for the offloaded part would be 1.86, and the estimated overall speedup is therefore 1.063.

When the asynchronous computation is offloaded onto the iGPU, no transfer is needed from graphics memory to system memory. Optimally, the fraction of asynchronous computation can be treated as completely taken out from the dGPU and thus the speedup s for the offloaded part would be ∞. Then the theoretical overall speedup for Computer A would be 1.08 (upper bounded) and for Computer B would be 1.137.

All the estimation above assumes that the iGPU can complete the offloaded task within the time the dGPU completes all tasks in a frame and data transfer between memories can be saturated.

4.2 Baseline Performance of Vulkan Implementation

Both OpenGL and Vulkan implementations for single discrete GPU were tested on the scene shown in Figure 23 and Figure 18. The results are presented in Figure 22 and Table 3.

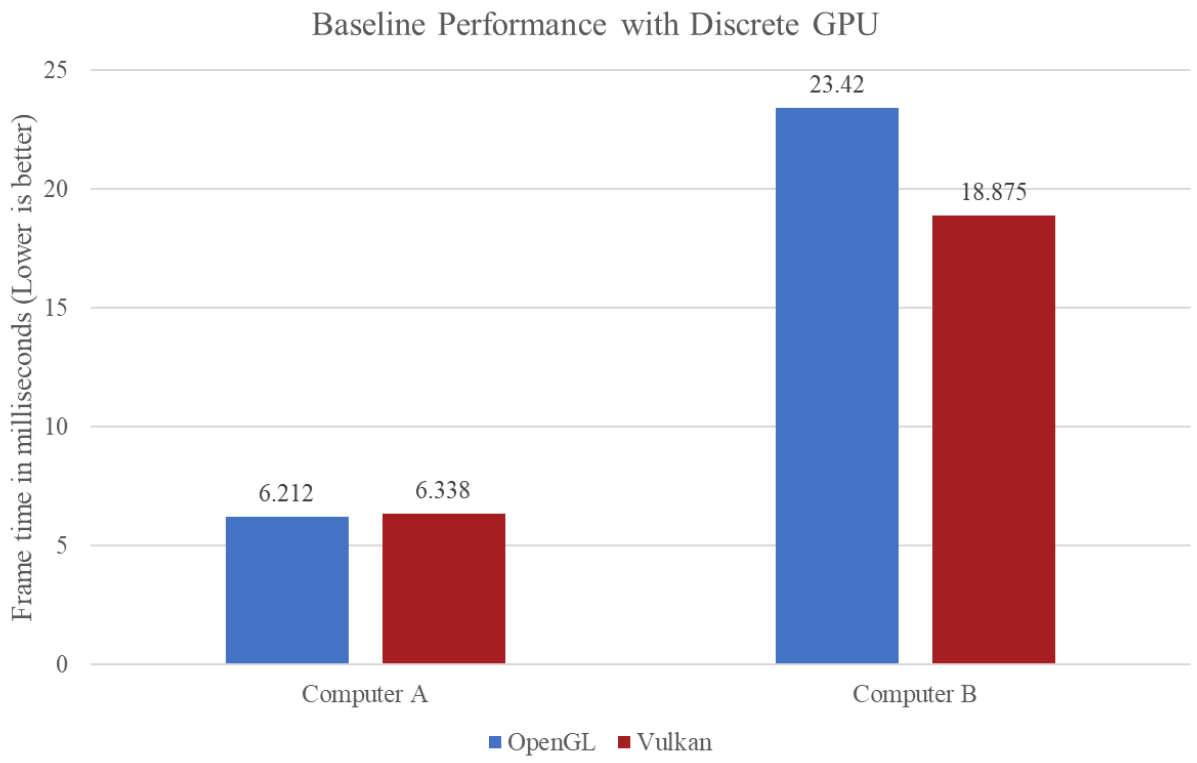


Figure 22. Average frame time with different API implementations

Table 3. Average frame time with different API implementations

API	Computer A	Computer B
OpenGL	6.212ms	23.420ms
Vulkan	6.338ms	18.875ms

Note that although Computer A has a slightly higher average frame time under Vulkan implementation, the observed program CPU usage is also around 60% lower than under OpenGL implementation (see appendix B). While on Computer B both API implementations have similar CPU usage. Also, note that this is the only test in this study where OpenGL is involved.

The results indirectly support that there is no fatal flaw in the Vulkan implementation for the discrete-GPU-only situation in the test program. Therefore, the results of heterogeneous GPUs implementation and discrete-GPU-only implementation can be directly compared.

4.3 Offloading Post-Processing

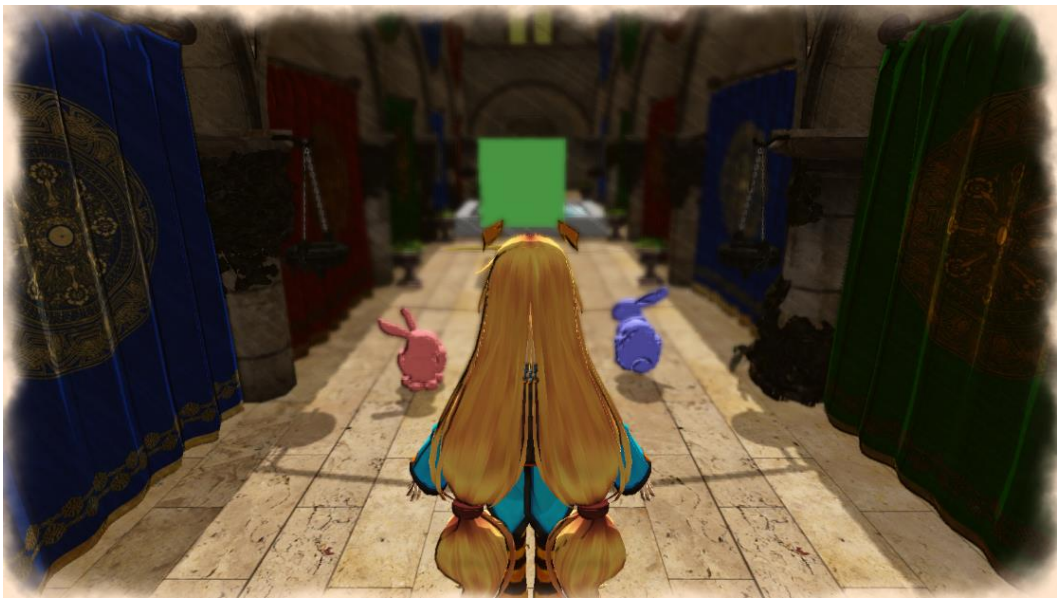


Figure 23. Test scene for offloading post-processing

When offloading post-processing and presentation workload to the iGPU, the performance measured on both test computers is presented in Table 4 and Figure 24.

Table 4. Average frame time with offloading post-processing

	Computer A	Computer A – with simulated pressure	Computer B
Discrete GPU	6.338ms	36.324ms	18.875ms
Integrated GPU	66.624ms	N/A	114.725ms
Heterogeneous GPUs	25.580ms	39.572ms	50.379ms
Improvement	**-303.5%**	**-8.9%**	**-166.9%**

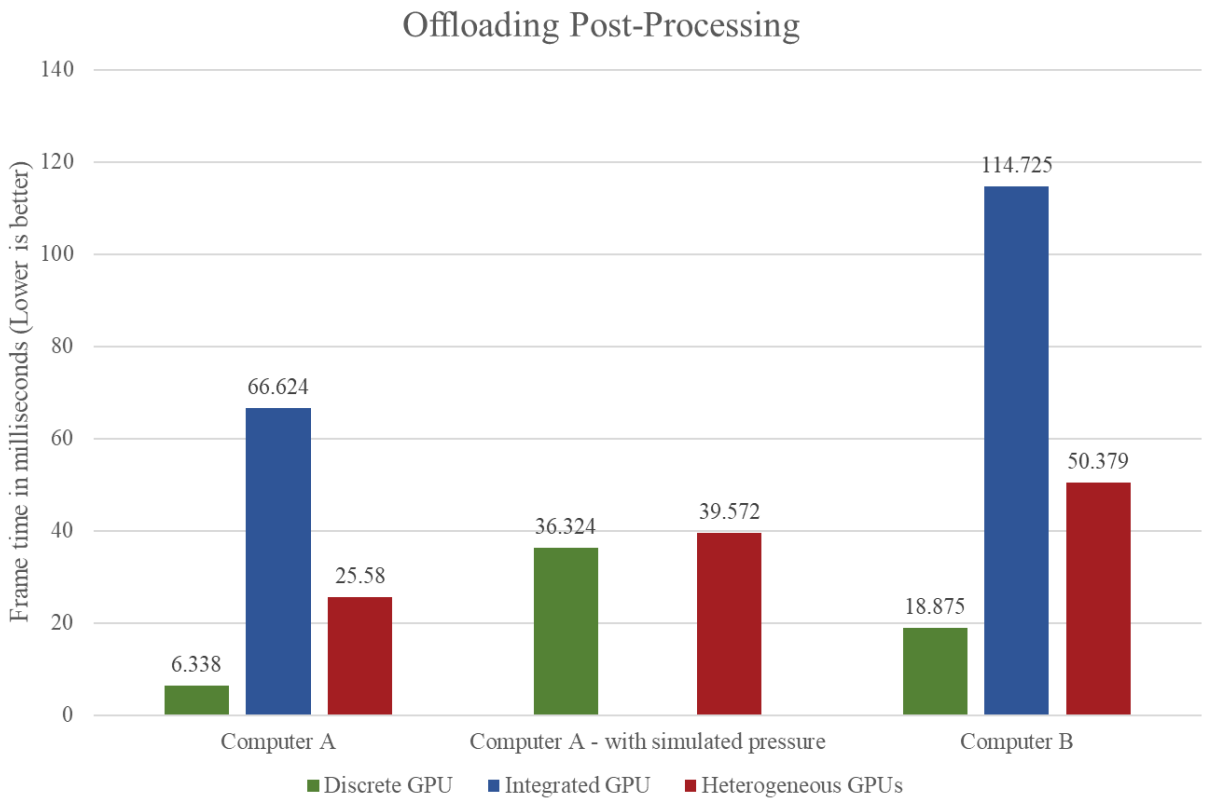


Figure 24. Average frame time with offloading post-processing

Contradictory to the theoretical analysis, a huge performance deterioration occurred with heterogeneous GPUs. However, this is not surprising with further analysis.

In the theoretical analysis section, the optimal bandwidth of system memory was used to do the calculation. Such bandwidth, however, is hardly achieved at runtime. As a reference, theoretically copying three 21.97 MB buffers inside system memory in Computer A would take 3.1ms, but in reality, a varying total time consumption between 5ms and 7ms was observed.

When transferring texture data from dGPU to integrated iGPU, ideally the texture data from graphics memory can be written directly into the corresponding texture memory area in system memory. However, this is not possible in current implementation as data transfer between different logical devices must go through staging buffers. This extra staging step produced an additional transfer overhead. Furthermore, the PCIe 3.0 bandwidth also bottlenecked memory transfer rate.

Besides the data copying overhead, signal latency between CPU and GPU, and latency within threads communication were also not considered in the theoretical analysis section. Adding up all these unconsidered overheads, the total communication overhead between two GPUs can

easily exceed the time consumed by the post-processing pass on dGPU. This effect is illustrated in Figure 25.

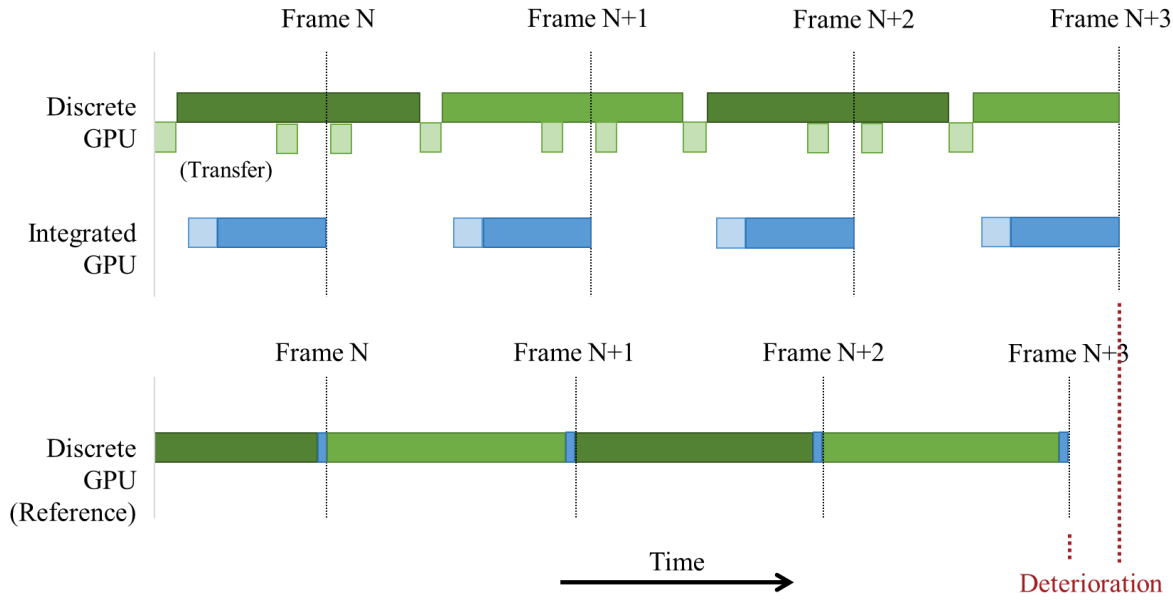


Figure 25. Data transfer overhead exceeds the offloaded workload fraction

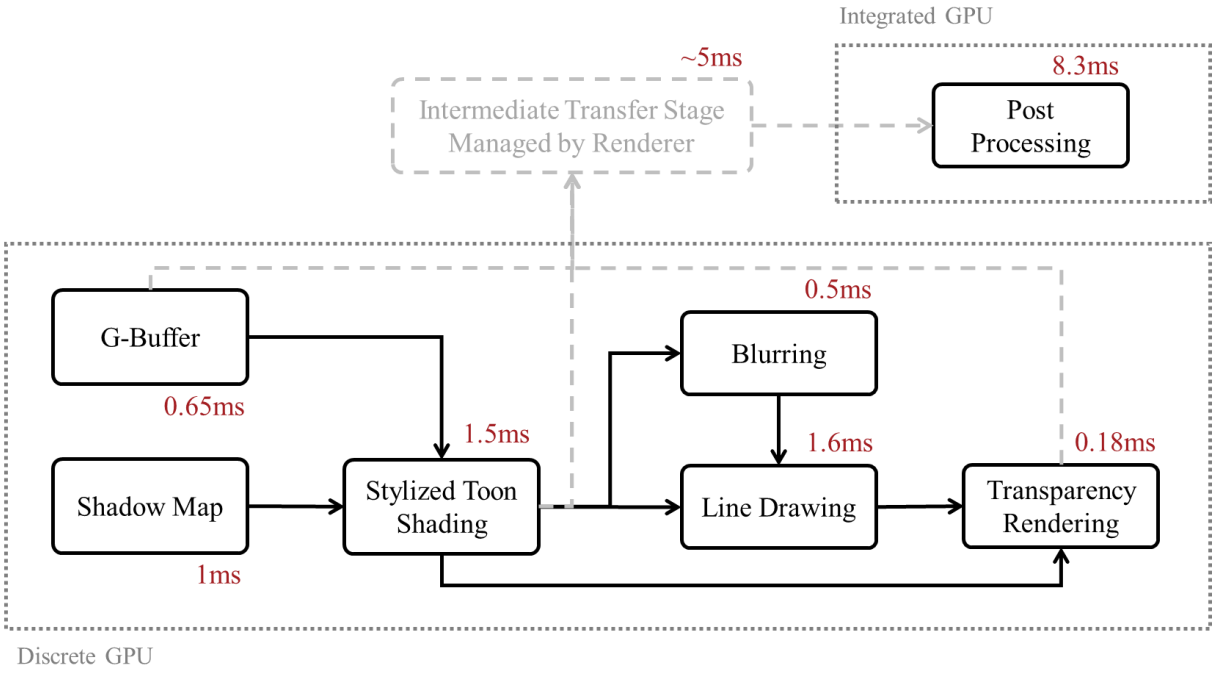


Figure 26. Time consumption breakdown for heterogeneous GPUs

The additional performance analysis from RenderDoc also revealed that the time consumed by the iGPU to complete the post-processing pass on Computer A exceeds the total time required by the dGPU to complete a full-frame as illustrated in Figure 26. This is the reason why when simulated pressure is added to the dGPU on Computer A, the percentage of performance deterioration is significantly reduced (pressure simulation is achieved through issuing overdraws on dGPU). And it is also the reason why the performance deterioration is less on Computer B since it inherently takes longer for the dGPU to complete a full frame.

After these analyses, it becomes clear that if performance improvement is to be achieved when the iGPU is offloading post-processing workload from the dGPU, two conditions must be satisfied:

1. The data transfer overhead between the iGPU and the dGPU must be less than the workload fraction taken out from the dGPU.
2. The iGPU must be able to complete the offloaded task(s) within the time required by the dGPU to complete a full frame.

Unfortunately, condition 1 can hardly be achieved on either Computer A or Computer B. Because covering the 5ms data transfer overhead alone would require the fraction of workload taken out from the dGPU to be greater than 5ms. According to the measured data, such a fraction can be predicted to take more than 50ms to complete on the iGPU. Which means the program would be running under at most 18 frames per second. Under such a low frame rate, it would hardly be practical to seize a less than 5% performance improvement for rendering only.

4.4 Offloading Asynchronous Computation

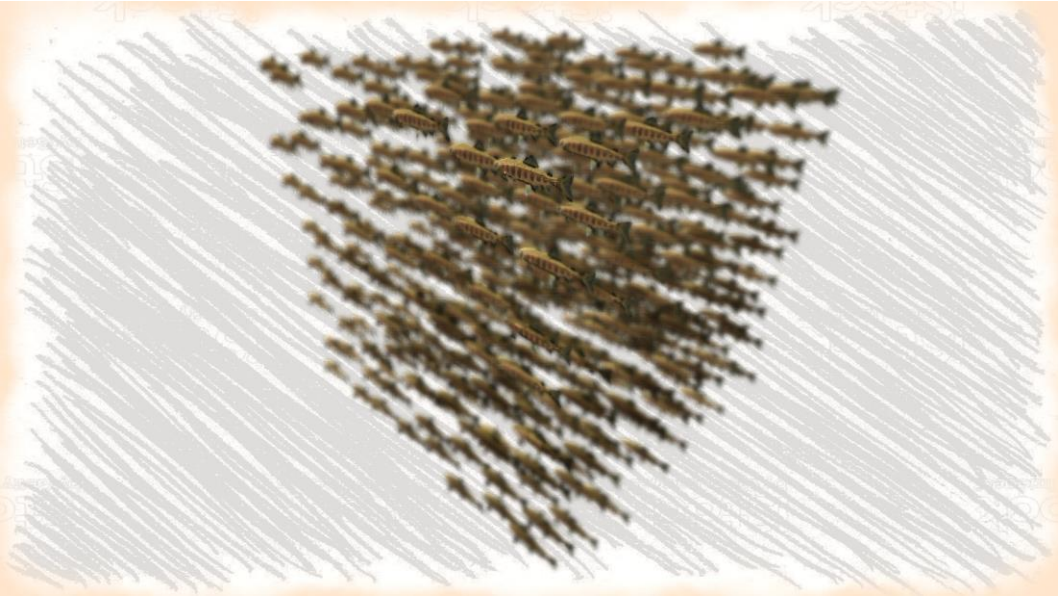


Figure 27. Test scene for offloading asynchronous computation.

Before diving into the heterogeneous GPUs test, the program was first tested in CPU computation mode to ensure that it is necessary to offload the workload from CPU to GPU. And the result is displayed in Figure 28, as both computers were running below 10 frames per second.

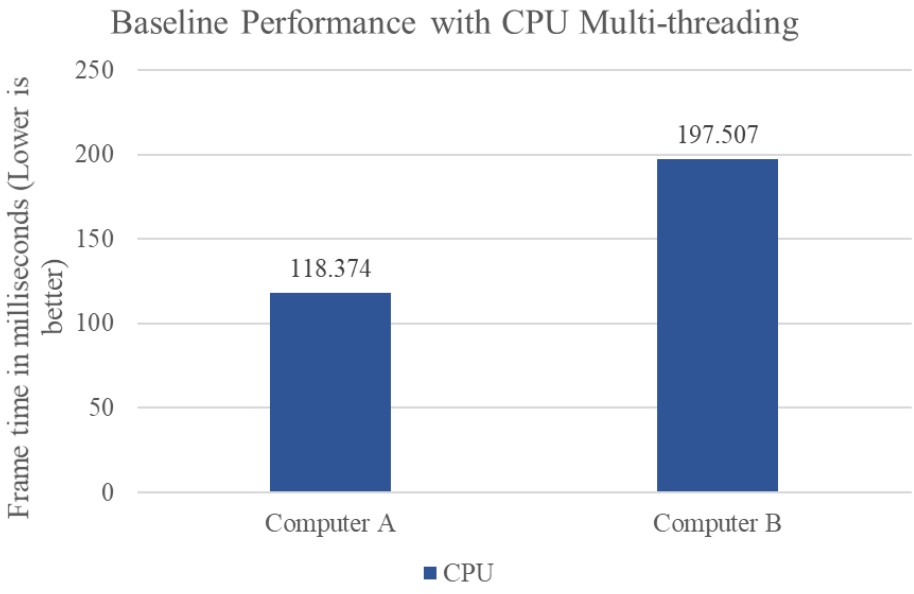


Figure 28. Average frame time with CPU multi-threading

When offloading the asynchronous computation to the iGPU/dGPU, the performance measured on both devices is presented in Figure 29 and Table 5. The code used to generate asynchronous computation content is provided in appendix A. Note that the load increase is not proportional to the actual time taken by the GPU to complete the computation.

This time, the heterogeneous GPUs are showing positive results closer to the prediction in theoretical analysis. As under this workload offloading strategy, there is no data transfer overhead between the iGPU and the dGPU, and both GPUs would only write the asynchronous compute results to the system memory.

The major overhead that was not included in the theoretical analysis is the extra CPU loads for issuing commands for the iGPU, which would delay the command recording for the dGPU. But this overhead can be well covered within the fraction taken out from the dGPU's workloads as illustrated in Figure 30.

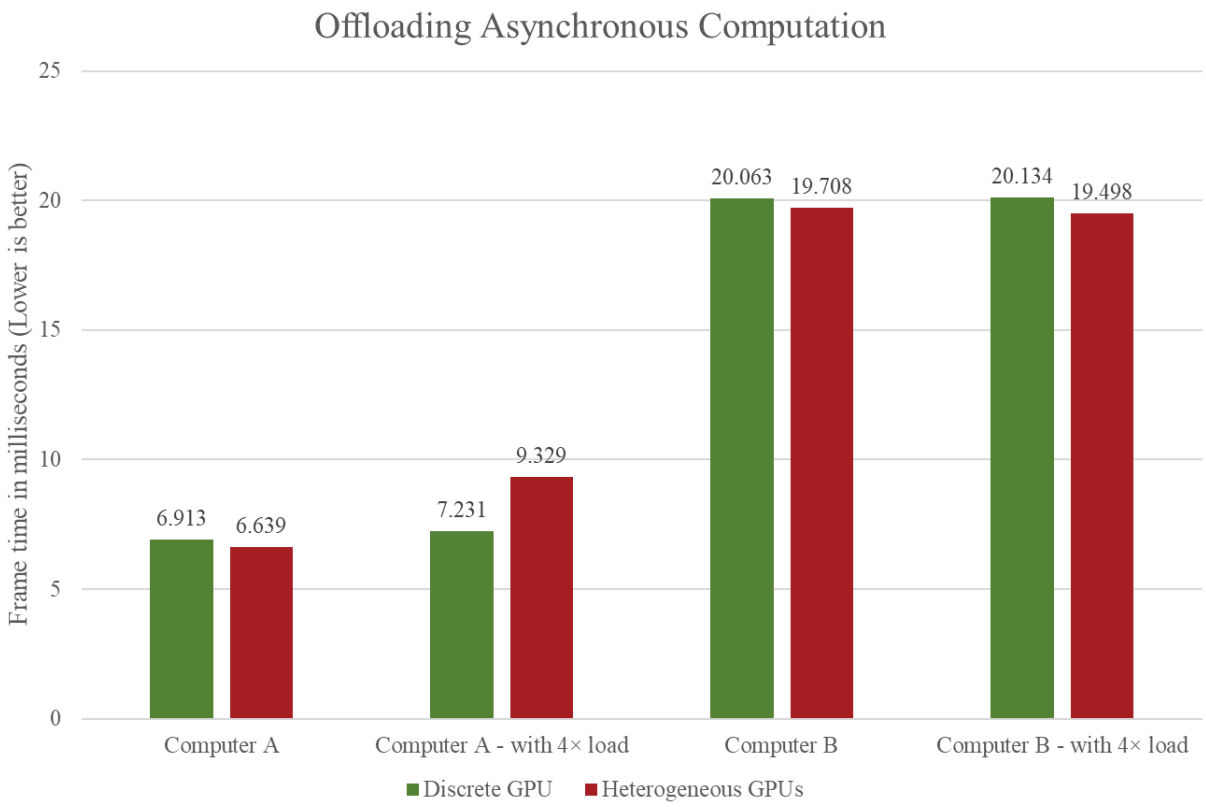


Figure 29. Average frame time with offloading asynchronous computation

Table 5. Average frame time with offloading asynchronous computation

	Computer A	Computer A – with extra load	Computer B	Computer B – with extra load
Discrete GPU	6.913ms	7.231ms	20.063ms	20.134ms
Heterogeneous GPUs	6.639ms	9.329ms	19.708ms	19.498ms
Improvement	**+4.1%**	**-29.0%**	**+1.8%**	**+3.3%**

The performance analysis from RenderDoc partially explained why performance decreased on Computer A while performance further improved on Computer B when the computational load was increased.

The original workload would take the iGPU on Computer A around 5ms to complete as illustrated in Figure 31, which is within the time required for the dGPU to complete a full frame. And as the computational load was increased on the iGPU, the iGPU became the bottleneck in the system and the dGPU had to halt and wait for the iGPU, resulting in the performance decrease. From the data, it can be inferred that it would now take the iGPU about 8ms to complete the task.

The iGPU on Computer B only has slightly worse performance than the iGPU on Computer A, thus the time required for the iGPU on Computer B to complete the same computational task can be expected to be well within 19ms.

Since the frame rate on Computer B is much lower than the frame rate on Computer A, the iGPU on Computer B was less loaded than the iGPU on Computer A as the computational task is issued per frame. It is possible that under normal computational load, the iGPU on Computer B was working on a lower frequency due to low workload (supported by observed iGPU usage). And the iGPU on Computer B was working on a higher frequency when the workload increased, resulting in a faster completion speed.

These observations suggest that when offloading asynchronous computation workload to the iGPU, the relative performance of the iGPU and application target frame rate should be carefully balanced.

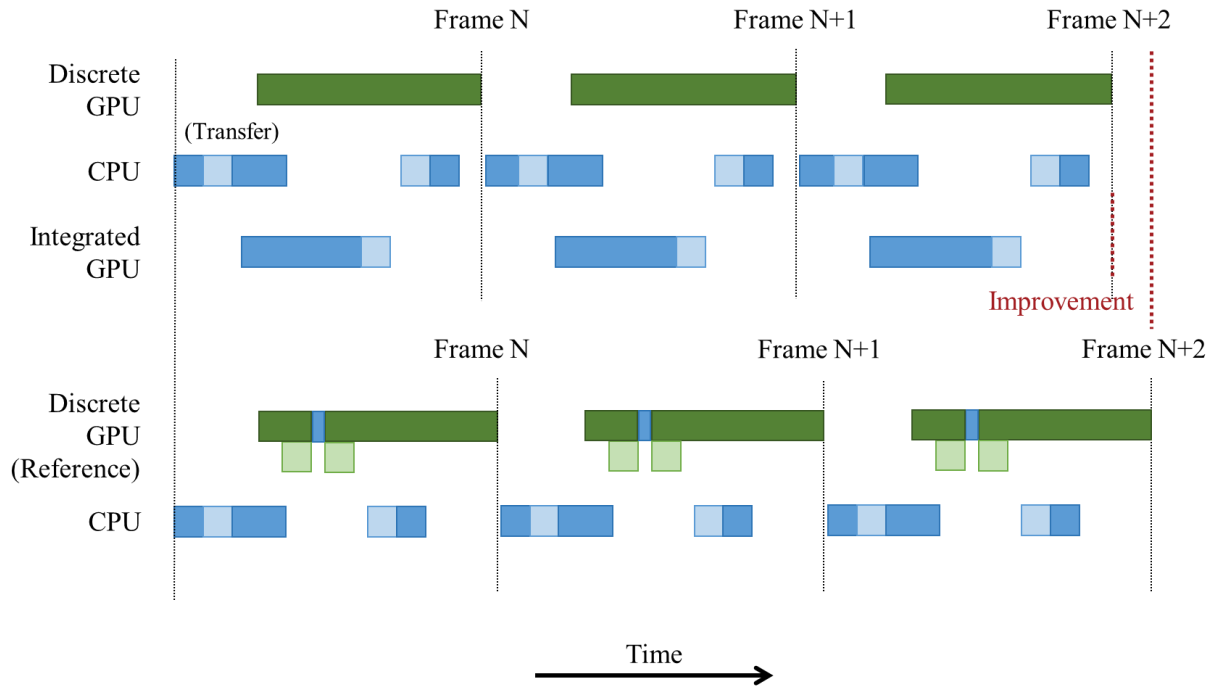


Figure 30. The overhead is less than the offloaded workload fraction

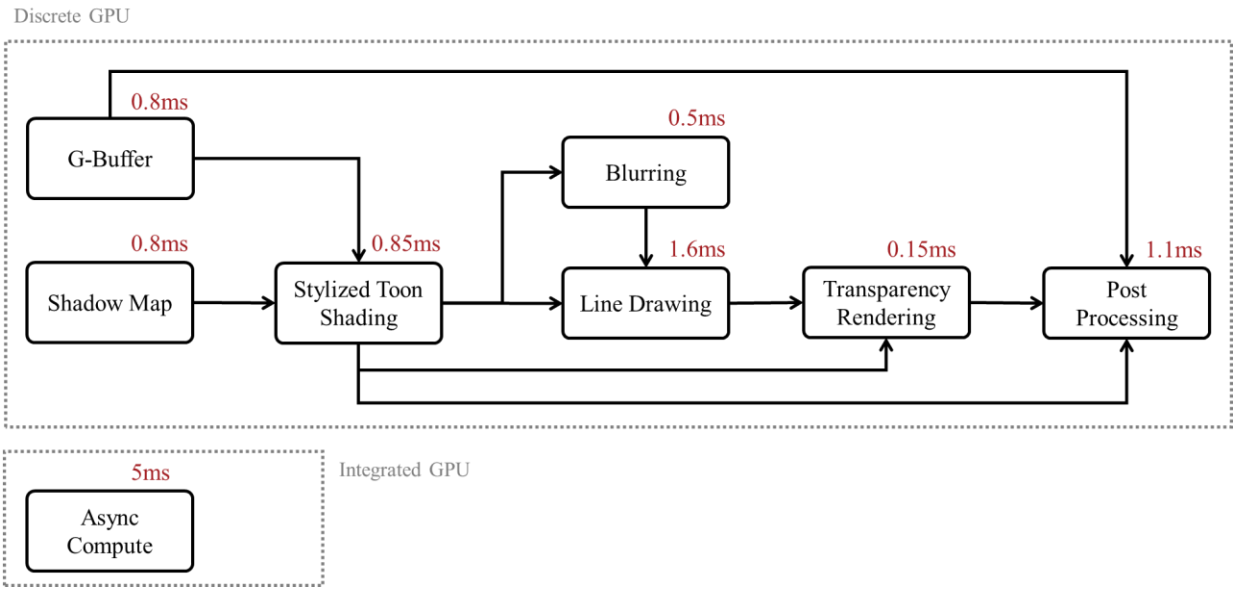


Figure 31. Time consumption breakdown for heterogeneous GPUs

4.5 Conclusion

With the conducted tests and analyses, although this study failed in obtaining performance improvement through offloading post-processing workload from the dGPU to the iGPU, performance improvement was successfully observed with offloading asynchronous computation from the dGPU to the iGPU. Also, the necessary conditions for improving performance through offloading screen space output workload / offloading asynchronous computation workload were explored and verified:

1. The data transfer overhead between the iGPU and the dGPU must be less than the workload fraction taken out from the dGPU.
2. The iGPU must be able to complete the offloaded task within the time required by the dGPU to complete a full frame unless the task is not frame-aligned.

This research proves that it is possible to make use of the integrated and discrete GPUs concurrently in the same application with the help of Vulkan. And offloading asynchronous computation workload from the discrete GPU to the integrated GPU can provide up to 3-4% performance improvement with combinations like UHD Graphics 630 + RTX 2070 Max-Q and HD Graphics 630 + GTX 1050.

4.6 Future Work

This study can be further improved by:

1. Try offloading screen space workload with low-end discrete GPU (such as NVIDIA GeForce MX150) that has a smaller performance gap from the integrated GPU.
2. Test and analyze the performance stability of the heterogenous GPUs working model when other GPU demanding program is present in the system.
3. Monitor the system power consumption and component temperature difference.
4. Test and analyze the sweet spot in the size of data transferred for the offloaded workload.

In addition, the approach used in this study can be improved by using buffer object to perform asynchronous computation directly, so that the buffer-image copy overhead can be eliminated. Also, in the case that resource copy command across different logical devices becomes supported in the future, speed up times can be expected to be enlarged through reducing the staging steps.

REFERENCES

Arntzen, H. K. (2017, August 15). *Render graphs and Vulkan — a deep dive*. Retrieved from Maister's Graphics Adventures: http://themaister.net/blog/2017/08/15/render-graphs-and-vulkan-a-deep-dive/

Barla, P., Thollot, J., & Markosian, L. (2006). X-Toon: An extended toon shader . *Proceedings of the 4th international symposium on Non-photorealistic animation and rendering*, (pp. 127-132).

Blackert, A. (2016). Evaluation of multi-threading in Vulkan. Linköping University.

Chen, L., Villa, O., Krishnamoorthy, S., & Gao, G. R. (2010). Dynamic Load Balancing on Single- and Multi-GPU Systems Long. *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)* (pp. 1-12). IEEE.

Dobersberger, S. (2015). Reducing Driver Overhead in OpenGL, Direct3D and Mantle. University of Applied Sciences Technikum Wien.

Gambhir, M., Panda, S., & Basha, S. J. (2018). Vulkan Rendering Framework for Mobile Multimedia. *SIGGRAPH Asia 2018 Posters* (p. 64). ACM.

GPUOpen. (2019, October 19). *Vulkan Memory Allocator*. Retrieved from https://gpuopen.com/gaming-product/vulkan-memory-allocator/

G-Truc. (2019, September 8). *OpenGL Mathematics*. Retrieved from https://glm.g-truc.net/0.9.9/index.html

Hill, M. D., & Marty, M. R. (2008). Amdahl's Law in the Multicore Era. *Computer*, 41(7), 33-38.

Hux, A. (2020, March 17). *Multi-Adapter with Integrated and Discrete GPUs*. Retrieved from https://devmesh.intel.com/projects/multi-adapter-particles

Jones, J. (2019, July). *Introducing Timeline Semaphores.* Retrieved from Videos, Presentations, & Supporting Materials Archives - The Khronos Group Inc: https://www.khronos.org/assets/uploads/developers/library/2019-siggraph/Vulkan-04-Timeline-Semaphore-SIGGRAPH-Jul19.pdf

Karlsson, B. (2019, October 19). *RenderDoc*. Retrieved from https://renderdoc.org/

Kenwright, B. (2017). Getting started with computer graphics and the Vulkan API. *SIGGRAPH Asia 2017 Courses* (p. 5). ACM.

Khronos Group. (2019, October 19). *KhronosGroup/SPIRV-Cross*. Retrieved from GitHub: https://github.com/KhronosGroup/SPIRV-Cross

Khronos Group. (2019, October 27). *Vulkan Overview*. Retrieved from https://www.khronos.org/vulkan/

Lee, Y., Markosian, L., Lee, S., & Hughes, J. F. (2007). Line drawings via abstracted shading. *ACM Transactions on Graphics*, 18-es.

Lujan, M., Baum, M., Chen, D., & Zong, Z. (2019). Evaluating the Performance and Energy Efficiency of OpenGL and Vulkan on a Graphics Rendering Server. *2019 International Conference on Computing, Networking and Communications (ICNC)*, 777-781.

LunarG. (2019, October 19). *Vulkan® SDK - What's in the SDK - Where to Download*. Retrieved from LunarG Inc.: https://www.lunarg.com/vulkan-sdk/

McGuire, M. (2017, July). *Computer Graphics Archive*. Retrieved from https://casual-effects.com/data/

Mustafin, M., Almaty, K., Akhmed-Zaki, D., & Turar, O. (2019). Application of Vulkan technology for 3D Visualization of large computing data which change over the time. *Journal of Mathematics, Mechanics and Computer Science, 102*(2), 46-55.

Peek, E., Wünsche, B., & Lutteroth, C. (2014). Using Integrated GPUs to Perform Image Warping for HMDs. *Proceedings of the 29th International Conference on Image and Vision Computing New Zealand* (pp. 172-177). ACM.

Persson, T. (2017, August 28). *High-Level Rendering Using Render Graphs*. Retrieved from Our Machinery: https://ourmachinery.com/post/high-level-rendering-using-render-graphs/

SAMSUNG. (2019, October 18). *Vulkan Usage Recommendations*. Retrieved from SAMSUNG Developers: https://developer.samsung.com/game/usage

Shiraef, J. A. (2016). An exploratory study of high performance graphics application programming interfaces. The University of Tennessee at Chattanooga.

Stanford Computer Graphics Laboratory. (2014, August 19). *The Stanford 3D Scanning Repository*. Retrieved from http://graphics.stanford.edu/data/3Dscanrep/

Stuart, J. A., Chen, C.-K., Ma, K.-L., & Owens, J. D. (2010). Multi-GPU volume rendering using MapReduce. *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing* (pp. 841-848). ACM.

Subtil, N., Rusch, M., & Fedorov, I. (2019, June 6). *Tips and Tricks: Vulkan Dos and Don'ts*. Retrieved from NVIDIA Developer: https://devblogs.nvidia.com/vulkan-dos-donts/

TechPowerUp. (2020, March). *NVIDIA GeForce GTX 1050 Mobile Specs | TechPowerUp GPU Database*. Retrieved from TechPowerUp: https://www.techpowerup.com/gpu-specs/geforce-gtx-1050-mobile.c2917

TechPowerUp. (2020, March). *NVIDIA GeForce RTX 2070 Max-Q Specs | TechPowerUp GPU Database*. Retrieved from TechPowerUp: https://www.techpowerup.com/gpu-specs/geforce-rtx-2070-max-q.c3392

The GLFW Development Team. (2019, October 19). *GLFW - An OpenGL Library*. Retrieved from https://www.glfw.org/index.html

Thorvaldsdóttir, H., Robinson, J. T., & Mesirov, J. P. (2013). Integrative Genomics Viewer (IGV): high-performance genomics data visualization and exploration. *Briefings in bioinformatics*, 14(2), 178-192.

Tolo, L. O. (2018). Multi-GPU Rendering with Vulkan API. The University of Bergen.

Unity Technologies Japan G.K. (2015). Retrieved from UNITY-CHAN! OFFICIAL WEBSITE: https://unity-chan.com/

Vanderhaeghe, D., Vergne, R., Barla, P., & Baxter, W. (2011). Dynamic Stylized Shading Primitives. *:Proceedings of the 8th International Symposium on Non-Photorealistic Animation and Rendering*, (pp. 99-104). Vancouver, Canada.

Vergne, R., Pacanowski, R., Barla, P., Granier, X., & Schlick, C. (2009). Light Warping for Enhanced Surface Depiction. *ACM Transactions on Graphics*, 25:1–25:8.

Vergne, R., Pacanowski, R., Barla, P., Granier, X., & Schlick, C. (2010). Radiance Scaling for Versatile Surface Enhancement. *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, (pp. 143-150). Boston, United States.

WikiChip. (2020, March). *HD Graphics 630 - Intel - WikiChip*. Retrieved from WikiChip: https://en.wikichip.org/wiki/intel/hd_graphics_630

WikiChip. (2020, March). *UHD Graphics 630 - Intel - WikiChip*. Retrieved from WikiChip: https://en.wikichip.org/wiki/intel/uhd_graphics/630

Wikipedia. (2019, October 19). *Microsoft Visual Studio - Wikipedia*. Retrieved from https://en.wikipedia.org/wiki/Microsoft_Visual_Studio

Yeung, A. (2015, April 30). *DirectX 12 Multiadapter: Lighting up dormant silicon and making it work for you*. Retrieved from DirectX Developer Blog: https://devblogs.microsoft.com/directx/directx-12-multiadapter-lighting-up-dormant-silicon-and-making-it-work-for-you/

Zhang, A., Chen, K., Johan, H., & Erdt, M. (2018). High performance city rendering in Vulkan. *SIGGRAPH Asia 2018 Posters.* ACM.

APPENDIX A. CODE SNIPPET OF ASYNC COMPUTE CONTENT

```
float PseudoRand(vec2 co)
{
	return fract(sin(dot(co, vec2(12.9898, 78.233))) * 43758.5453);
}

vec4 UpdatePosition(vec4 origin, vec2 seed)
{
	vec4 result = origin;

	float coeff = 0.1f * PseudoRand(seed);

	vec4 pressureVal = vec4(0.0);
	for (int i = 0; i < 5000; i += 2)
	{
		pressureVal += 0.001f * vec4(
			PseudoRand(vec2(cos(PseudoRand(vec2(i, origin.z))) *
sin(PseudoRand(vec2(i, origin.y))), seed.y)),
			PseudoRand(vec2(sin(PseudoRand(vec2(i, origin.x))) *
cos(PseudoRand(vec2(i, origin.z))), seed.y)),
			PseudoRand(vec2(seed.y, cos(PseudoRand(vec2(i, origin.y))) *
sin(PseudoRand(vec2(i, origin.x))))),
			0);
	}
	for (int i = 1; i < 5000; i += 2)
	{
		pressureVal -= 0.001f * vec4(
			PseudoRand(vec2(cos(PseudoRand(vec2(i, origin.z))) *
sin(PseudoRand(vec2(i, origin.y))), seed.y)),
			PseudoRand(vec2(sin(PseudoRand(vec2(i, origin.x))) *
cos(PseudoRand(vec2(i, origin.z))), seed.y)),
			PseudoRand(vec2(seed.y, cos(PseudoRand(vec2(i, origin.y))) *
sin(PseudoRand(vec2(i, origin.x))))),
			0);
	}

	result += coeff * pressureVal;

	return result;
}
```

The complete codebase of this study is available from:

https://github.com/N7RX/CactusEngine/tree/heterogeneous

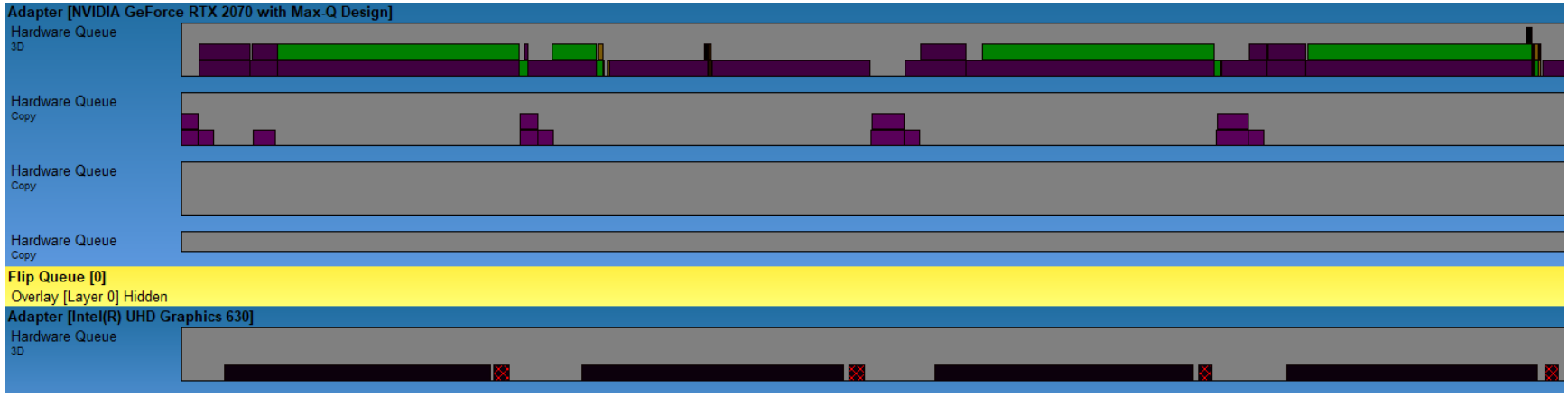
APPENDIX B. PERFORMANCE DATA COLLECTION

Specification	CPU	GPU 0	GPU 1	RAM	Windows	GPU 0 Driver	GPU 1 Driver	Vulkan Runtime	GPU 0 Direct Output
Notebook	i7 9750H	RTX 2070 Max-Q	UHD 630	DDR4 2666 Dual	Windows 10, 1903	26.21.14.4236	26.20.100.7463	1.2.131.2	Yes

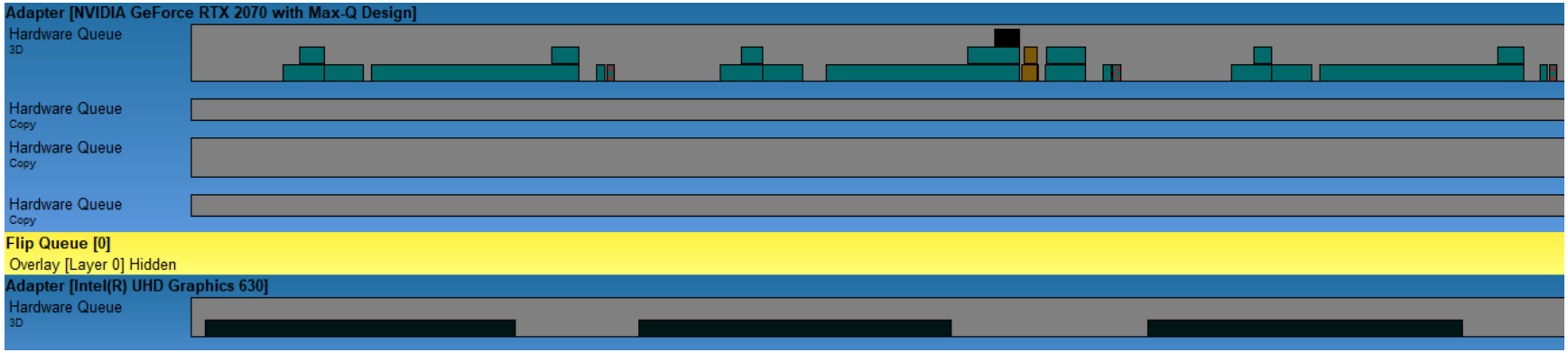
Frametime (ms)	1	2	3	Average	GPU 0	1	2	3	Average	GPU 1	1	2	3	Average	CPU	1	2	3	Average
Base Test																			
Vulkan	6.25	6.37	6.394	6.338		95%	94%	94%	94%							6%	5%	5%	5%
OpenGL	6.157	6.206	6.272	6.211667		99%	99%	99%	99%							15%	15%	15%	15%
Render Offload Test																			
Discrete	6.25	6.37	6.394	6.338		95%	94%	94%	94%							6%	5%	5%	5%
Integrated	66.67	66.628	66.573	66.62367							98%	98%	98%	98%		2%	2%	2%	2%
Heterogeneous	25.992	26.031	24.716	25.57967		28%	28%	30%	29%		86%	86%	85%	86%		8%	8%	9%	8%
Discrete - With Pressure	35.917	36.476	36.579	36.324		99%	99%	99%	99%							2%	2%	2%	2%
Heterogeneous - With Pressure	39.38	39.765	39.572	39.57233		97%	97%	97%	97%		74%	76%	76%	75%		7%	6%	6%	6%
Compute Offload Test																			
CPU	117.176	117.974	119.973	118.3743		32%	32%	30%	31%							51%	51%	52%	51%
Discrete	6.861	6.937	6.94	6.912667		80%	80%	81%	80%							16%	16%	16%	16%
Heterogeneous	6.595	6.687	6.636	6.639333		73%	74%	73%	73%		74%	73%	74%	74%		16%	16%	16%	16%
Discrete - 4x Load	7.131	7.243	7.319	7.231		94%	94%	94%	94%							16%	16%	16%	16%
Heterogeneous 4x Load	9.331	9.324	9.332	9.329		51%	51%	51%	51%		91%	91%	91%	91%		11%	12%	11%	11%
Date: 03/16/2020																			

Specification	CPU	GPU 0	GPU 1	RAM	Windows	GPU 0 Driver	GPU 1 Driver	Vulkan Runtime	GPU 0 Direct Output
Notebook	i5 7300HQ	GTX 1050	HD 630	DDR4 2133	Windows 10, 1903	26.21.14.4236	26.20.100.7870	1.2.131.2	No

Frametime (ms)	1	2	3	Average	GPU 0	1	2	3	Average	GPU 1	1	2	3	Average	CPU	1	2	3	Average
Base Test																			
Vulkan	18.881	18.853	18.89	18.87467		96%	96%	96%	96%							8%	7%	7%	7%
OpenGL	23.237	23.356	23.667	23.42		96%	96%	96%	96%							6%	7%	6%	6%
Render Offload Test																			
Discrete	18.881	18.853	18.89	18.87467		96%	96%	96%	96%							8%	7%	7%	7%
Integrated	114.579	114.796	114.799	114.7247							99%	99%	99%	99%		10%	11%	10%	10%
Heterogeneous	50.359	50.466	50.313	50.37933		43%	43%	43%	43%		74%	74%	74%	74%		25%	24%	25%	25%
Discrete - With Pressure																			
Heterogeneous - With Pressure																			
Compute Offload Test																			
CPU	199.644	196.424	196.453	197.507												93%	91%	90%	91%
Discrete	20.1106	20.02	20.058	20.06287		85%	86%	86%	86%							21%	21%	21%	21%
Heterogeneous	19.634	19.746	19.745	19.70833		85%	84%	85%	85%		33%	33%	33%	33%		20%	19%	19%	19%
Discrete - 4x Load	19.987	20.046	20.369	20.134		92%	93%	91%	92%							20%	21%	20%	20%
Heterogeneous 4x Load	19.485	19.505	19.503	19.49767		85%	85%	84%	85%		73%	72%	73%	73%		19%	19%	19%	19%
Date: 03/16/2020																			



Offloading Post-Processing



Offloading Async Computation