Institutionen för systemteknik
Department of Electrical Engineering

Examensarbete

Evaluation of multi-threading in Vulkan

av

Axel Blackert

LiTH-ISY-EX-ET--16/0458--SE

2016-09-12
Abstract

Today processor development has a lot of focus on parallel performance by providing multiple cores that programs can use. The problem with the current version of OpenGL is that it lacks support for utilizing multiple CPU threads for calling rendering commands. Vulkan is a new low level graphics API that gives more control to the developers and provides tools to properly utilize multiple threads for doing rendering operations in parallel. This should give increased performance in situations where the CPU is limiting the performance of the application and the goal of this report is to evaluate how large these performance gains can be in different scenes. To do this evaluation a test program is written with both Vulkan and OpenGL implementations and by rendering the same scene using different APIs and techniques the performance can be compared. In addition to evaluating the multithreaded rendering performance the new explicit pipelines in Vulkan is also evaluated.
Definitions

CPU – Central Processing Unit
GPU – Graphical Processing Unit
API – Application Programming Interface
FPS – Frames Per Second
GLSL – OpenGL Shading Language
1 CONTENT

1.1 Background .................................................................................................................. 1
1.2 Purpose ........................................................................................................................ 1
1.3 Task .............................................................................................................................. 1
1.4 Limitations ................................................................................................................... 1
2 Theory ............................................................................................................................. 2
2.1 Khronos Group ............................................................................................................ 2
2.2 Vulkan .......................................................................................................................... 2
2.3 Driver overhead ......................................................................................................... 3
2.4 Platform independence ............................................................................................. 3
2.5 Multithreading ............................................................................................................ 3
2.5.1 No global state ........................................................................................................ 4
2.5.2 No synchronization in the drivers ......................................................................... 4
2.5.3 Separation between generation and execution of rendering ................................. 5
2.6 Rendering pipeline .................................................................................................... 5
2.7 Instancing .................................................................................................................... 6
2.8 3rd party libraries ....................................................................................................... 6
2.8.1 GLM ...................................................................................................................... 6
2.8.2 Assimp .................................................................................................................. 6
2.8.3 GLEW .................................................................................................................. 6
2.8.4 TSBK07 utilities .................................................................................................... 6
2.8.5 Sascha Willems utilities ......................................................................................... 6
2.9 Previous Work ............................................................................................................ 7
3 Method ............................................................................................................................ 8
3.1 The resulting program ................................................................................................. 8
3.2 Technical Terms .......................................................................................................... 8
3.2.1 Vertex and fragment shaders ................................................................................ 8
3.2.2 Shader uniforms ................................................................................................... 8
3.2.3 Push constants ...................................................................................................... 8
3.2.4 Command buffer ................................................................................................. 8
3.2.5 Command pool ..................................................................................................... 8
3.3 Implementation details .............................................................................................. 8
3.3.1 Rendering interface .............................................................................................. 9
3.3.2 Creating the renderer ........................................................................................... 9
3.3.3 Recording performance ......................................................................................... 10
3.3.4 Vulkan renderer .................................................................................................. 10
INTRODUCTION

1.1 BACKGROUND
With the trend in central processing units development lately being very focused on parallelism and multi core performance it have become apparent that the current version of OpenGL is holding the graphic cards performance back. The current version of OpenGL is unable to efficiently use multiple threads for rendering commands which can mean that a rendering intensive program is limited by the single core performance of the CPU. Apart from the bad multi-thread support OpenGL suffers from an old design that works like a big state machine with the drivers doing a lot of heavy lifting.

Vulkan is a new graphics API that’s designed from the ground up to fully support multi-threading and it gives greater control to the developer by providing lower level access to the graphics card. Vulkan shifts a lot of responsibility from the driver to the developer, for example by removing unnecessary error checking in the driver, which reduces driver overhead. This increased developer responsibility can offer performance increases if used correctly. Vulkan also features a new explicit rendering pipeline system that should reduce the overhead when changing pipeline states.

1.2 PURPOSE
With Vulkan being a new technology it is of great interest to the graphics programming industry for it to be evaluated and tested as early as possible after the release. Developers have to decide if it is worth to adapt Vulkan in their graphics engines. In order to perform the decision it is vital to know what performance increases there are to gain and if it is worth the investment in time and money.

The goal of this paper is to give an overview on how well Vulkan is able to perform in a set of scenarios and present the possible increases in performance compared to the latest version of OpenGL.

1.3 TASK
To do the required performance comparisons a simple test framework will be implemented with two different backend rendering systems using Vulkan and OpenGL. By rendering exactly the same scenes containing the same models with both APIs and changing parameters a fair comparison should be possible. The performance metric that will be used is how fast the program can render a scene, measured in frames per second (FPS).

The thesis questions to be examined are:

- How much of a performance increase does Vulkan yield when using multithreading?
- Is the new pipeline management more efficient?
- What is the experience of implementing a Vulkan application?

The first two thesis questions will be answered using the results from the performed tests while the last question will be answered subjectively from my own experience of using Vulkan.

1.4 LIMITATIONS
The work will be limited to 3D graphics and the testing will be done on the Windows operating system. The APIs compared will be Vulkan and OpenGL. The features to be compared are multithreading and pipeline management. To not present too many tests the multi-threaded testing will be limited to the testing of one thread and four threads. Only one vertex and pixel shader will be used in the tests.
2 Theory

In this chapter the theory needed to follow the rest of the report is presented. Starting with some background information about Vulkan and the Khronos Group and then going into the features of Vulkan that, as claimed, results in performance increases. With each presented feature the technical details about them is also explained. At last there are sub-chapters about the 3rd party libraries used in the project along a brief review of previous work on the subject.

2.1 Khronos Group

The creators of Vulkan is the Khronos Group which is a non-profit consortium. Khronos Group works with developing and maintaining new APIs for media with an open specification and free for developers to use. The consortium has many companies as members worldwide including AMD, Google, Apple, ARM, Intel, Samsung, Sony, Huawei, Nvidia, Epic Games, Nokia, Qualcomm, Vivante and Imagination [1].

The consortium was created 2000 and in 2006 they took control over the OpenGL specification. Other projects that Khronos Group runs are COLLADA, OpenCL, OpenKode, OpenVG, Spir, WebGL and EGL. Their latest project is Vulkan which will be the focus of this paper.

2.2 Vulkan

The development of Vulkan started in July 2014 when the Khronos Group had a meeting at Valve Coorporations and the project was announced the same year at the SIGGRAPH conference. Vulkan was officially released 16 February 2016. Many core features in Vulkan are taken from the discontinued low level graphics API Mantle that AMD created. One of the reasons to Mantles lack of success was that it only could run on the Windows platform. In a way Vulkan takes the best parts of Mantle and makes it entirely platform independent. [2]

The idea of a low level graphic API with more fine-grained control over the hardware is nothing new. It has been used for a long time on gaming consoles such as Xbox, Playstation and Nintendo. The
difference between gaming consoles and the PC however is that the developers know exactly which
hardware their code will be running on, making it easier to optimize it. The gaming consoles
architecture can also be built in a more specific manner since they have a more limited area of use
than a PC [11].

Vulkan is meant to bring the advantages of a console-like low level graphics API and still support
multiple platforms.

2.3 DRIVER OVERHEAD

The problem with graphics programming is that there are is an increasingly large amount of different
graphics cards that has to be supported. Vulkan currently supports graphics cards from Nvidia and
AMD from 2012 and forward, on both Windows and Linux [18], which from Nvidia alone means
~100 different graphics cards. To make it possible for a developer’s code to run on different graphics
card without actually having to worry about the hardware an extra layer sits between the hardware and
the developer’s code, the drivers.

In the current generation of graphics APIs (OpenGL 4.XX and DirectX 11) the drivers do a lot of
heavy lifting including doing error checks during runtime. These error checks are meant to catch
errors made by the programmer but they take crucial time to perform. According to Neil Trevett et al.
[3] it shouldn’t be the driver’s responsibility to find errors but instead the developers. By removing
the responsibility of finding errors in the drivers Vulkan prioritizes performance over ease of use.

This idea is used extensively in Vulkan by giving the developer more responsibility and control
which, if used correctly, can result in better performance. With great power comes great
responsibility. During development finding bugs and errors is very important so Vulkan offers
something called “validation layers” that can be enabled during the development phase to detect
errors and disabled in the released version to not affect the performance. The different layers are not
part of the Vulkan core but instead offered as extensions, making them entirely optional to include.

2.4 PLATFORM INDEPENDENCE

Vulkan is meant to be able to run on every platform and every graphics card that offers driver support.
To make Vulkan able to render graphics on every operating system Vulkan needs a surface to render
on. How this surface is retrieved is will differ between the different operating systems and to solve
this problem Vulkan uses platform dependent modules called Window System Interface (WSI). WSI
is an extension and not part of the Vulkan core [4]. All platforms that want to have support for Vulkan
will need to provide their own implementation of their platform specific WSI.

Windows, Linux and Android all officially provide their own WSI extension to make Vulkan run on
their platform. Worth noting is that Apple currently doesn’t provide any official WSI extension but
instead they offer a 3rd party Vulkan wrapper on top of their own graphics API called Metal.

2.5 MULTITHREADING

As a big focus of the CPU development is on multithread performance the current generation of
OpenGL loses a lot of performance due to the lack of multithreaded support. OpenGL does not allow
rendering calls to be made from more than one thread [6] which makes it inconvenient to utilize
multiple CPU cores for rendering commands. In the early versions of OpenGL this wasn’t a problem
since the CPUs then weren’t as parallel but today this old design of OpenGL can heavily limit the
performance if an application is CPU limited [5].
Greg Nott [13] have performed some extensive multithread testing with OpenGL and the result was that it decreased the performance rather than increasing it. The problem is that only one OpenGL context can be bound at any time so the driver has to change to the correct context, depending on the active thread, when executing a rendering command, resulting in significant loss of performance. The option is to synchronize the threads with locks and mutexes so only one thread can render at a time. This removes the context switching but instead there is added overhead from the synchronization between the threads. Pinheiro et al. [20] mentions that while it is possible to use multithreading in DirectX 9 and DirectX 10, the added overhead of manually synchronizing critical code can be so significant that in the end the performance instead decreases.

The image above shows the difference in FPS when rendering a scene using single threaded OpenGL ES and multithreaded Vulkan in the Gnome Horde demo by Imagination Technologies. In the demo the camera moves around in the word fast and has to regenerate a lot of command buffers every frame. The increase in FPS is significant when looking at the graphs, almost six times the FPS, but as Ashley Smith [19] mentions the scenario is exaggerated to highlight Vulkans strengths.

In contrast to OpenGL, Vulkan is designed from the ground up to utilize multiple CPU cores when generating rendering commands. There are three main features that makes this possible in Vulkan.

### 2.5.1 No global state

In Vulkan the API functions requires that every required object is sent as function arguments, unlike in OpenGL where the driver manages global objects such as the physical device and the OpenGL context. A problem quickly arises when the driver manages the global state and you try to add support for multithreading. The global state can be modified from any of the different threads [8] and as described above, there can only be one OpenGL context bound at any time so the driver has to switch OpenGL context to the active threads context, which adds overhead. Hemaiyer et al. [10] claims that by using global objects you decrease the support for multithreading.

```c
vkCreateShader(device, &info, &shader);  // Vulkan
glCreateShader(GL_VERTEX_SHADER);   // OpenGL
```

The code snippet above shows an example of the difference between creating a shader in Vulkan and OpenGL. In the Vulkan function you have to explicitly send the current device as an argument while in OpenGL the driver is responsible to find the current threads device.

### 2.5.2 No synchronization in the drivers

In OpenGL the drivers handles the synchronization between different threads and guarantees that accessing a global object is valid at any time [7]. Because the driver can’t know which threads will be accessing what objects every function that possibly can modify data from multiple threads at the same time needs to be locked. This is inefficient and creates overhead. In Vulkan the application itself is entirely responsible for synchronizing threads instead of the driver which has no synchronization responsibility at all [3]. This creates more opportunities for the developers to optimize their code and
in many cases you can with good design avoid accessing an object from multiple threads at the same time.

### 2.5.3 Separation between generation and execution of rendering

Rendering commands includes API functions that bind vertex buffers, bind index buffers, bind shader resources and issues primitive rendering. Generating rendering commands is a CPU intensive task and should be done as little as possible. In OpenGL rendering commands are generated and executed at the same time with no possibility of separating them. `glDrawElements()` both sends rendering commands to the GPU and executes them immediately after. Since you often want to control which order objects are rendered you are stuck to only using one thread. In Vulkan on the other hand there is a clear separation between generating rendering commands and executing them. Commands are generated in “command buffers”, these command buffers are then submitted to a “command queue” and it is only at this point they are executed. The command buffers can be generated from multiple threads in parallel and this is important in Vulkans ability to increase the multithreaded performance.

Command buffers are generated in different threads and then submitted to a command queue in the main thread. The main thread is the thread that actually executes the rendering commands. Executing rendering commands is a relatively cheap CPU operation so it can be done from a single thread without losing significant performance, unlike generating rendering commands which is really CPU intensive.

### 2.6 Rendering pipeline

A rendering pipeline consists of the different stages required to render objects to the screen. Some of the steps are input assembly, rasterization, colour blending, viewport clipping, depth and stencil testing, vertex shader and pixel shader. With all these steps it is the graphics cards job to perform each step and then begin the next one. The programmer’s job is to feed data to the pipeline and configure the states for each stage. Sending data to be processed by the rendering pipeline is similar in both Vulkan and OpenGL, you bind buffers, textures and other data in similar fashion. However the two APIs have a distinctively different approach to configuring the many pipeline stages.

In OpenGL the different pipeline stages states are global states that are initialized with some default value and can be changed any time in the program with a function call. For example if you want to change which shader to use you can call `glUseProgram()` and if you want to change the culling order you call `glCullFace()`.

---

**Figure 3** [https://www.khronos.org/assets/uploads/developers/library/overview/2015_vulkan_v1_Overview.pdf](https://www.khronos.org/assets/uploads/developers/library/overview/2015_vulkan_v1_Overview.pdf)
In Vulkan on the other hand you have to explicitly create the entire pipeline by yourself and define every stages state on creation. When rendering you bind the pipeline and all states defined in the pipeline since creation will be used. Apart from the viewport and scissor state there is no way to change any individual state after creation. If you want to change any pipeline state you need to change to another already created pipeline. The reason for this is to do the time consuming validation checks of pipeline states at creation and not at runtime which results in better performance when reusing pipelines [16].

2.7 Instancing
Instancing is a special rendering technique that allows you to render many copies of the same model with a single draw call. This is great for reducing CPU overhead since normally each draw call would add overhead and this overhead can significantly decrease the performance when rendering large amounts of the same model (1000+). Rendering a few vertices is done quickly by the GPU and it may be done before the CPU have sent the next draw call, resulting in the GPU waiting on the CPU which is bad for performance. Every instance has to have the same vertex data but properties like position, rotation, scale and color can be modified between instances.

Instancing is important for this report because it gives great performance increases in the same cases as multithreading, when rendering a lot of simple geometry that is. Instancing almost completely removes the CPU overhead of the rendering commands while multithreading instead balances the overhead over multiple threads. Instancing and multithreading will be compared in the evaluation chapter, section 4.1.2 and 4.3, by rendering a scene containing many objects with low vertex count.

2.8 3rd Party Libraries

2.8.1 GLM
A C++ mathematics library based on the GLSL specification [15]. This means that the naming conventions are the same as in GLSL and that a vec3 in C++ corresponds to a vec3 in GLSL. It also contains convenient functions for matrix transformations like translations, rotations, look at etc. GLM works with both Vulkan and OpenGL and is used everywhere in the project.

2.8.2 Assimp
An open source library that imports different 3d model formats. Once models are imported they are all represented in the same way, no matter their original format. Extracting the required data and creating program specific structures from it is a convenient process.

2.8.3 GLEW
GLEW is a C/C++ extension loader library that exposes OpenGL core and extension functionality in a single header file.

2.8.4 TSBK07 utilities
OpenGL utility functions for loading shaders, models and textures written by Ingemar Ragnemalm for the course TSBK07 at Linköpings Universitet.

2.8.5 Sascha Willems utilities
Vulkan utility classes and functions for loading shaders, loading textures and convenient struct initializers written by Sascha Willems. Also contains a swap chain wrapper class that handles swap chain image presenting to a windowing system.
2.9 Previous Work
Simon Dobersberger [14] performed a study where he examined the driver overhead in OpenGL and DirectX to see how much it affected the performance. Some tests also included multithreading testing. He found out that there are multiple techniques in both OpenGL and DirectX 11 that significantly can reduce the driver overhead, instancing being one of them. However, the drawback with many of the techniques is that they lack flexibility. The study was performed last year which still makes it relevant. He mentioned himself that examining the driver overhead in Vulkan would be a suitable future work of his study.
3 METHOD

The goal of this work is to evaluate what performance increases can be observed when utilizing some of the new features of Vulkan, specifically multi-threading and the explicit pipelines. To do this evaluation a program containing several test cases is developed and the performance logs from the tests are used as the base for the evaluation in this paper. This chapter will explain some details of how the program was developed to make it replicable.

3.1 THE RESULTING PROGRAM

The program containing the Vulkan and OpenGL renderers and test cases is written in C++ using Visual Studio 2015 as development IDE. The resulting program renders scenes containing different objects with different properties, some of the test scenes contain high detail models and other lower detail models. By using keyboard shortcuts the scene, renderer and rendering technique can be changed during runtime. A camera is implemented so the view can be moved around, but during the tests the camera is set to a static position to ensure a fixed load. The window header contains information about the scene along with the current FPS. The FPS for every second is saved in memory and when a test is completed the average of the recorded FPS is calculated and printed to a file on the hard disk.

3.2 TECHNICAL TERMS

These are some of the technical terms required to understand in order to follow the explanation of the implementation details.

3.2.1 Vertex and fragment shaders
Both are part of the rendering pipeline. Vertex shaders transforms each vertex’s 3d position to a screen coordinate. The fragment shader calculates the colour properties for each pixel in the scene.

3.2.2 Shader uniforms
Unforms are global GLSL variables that can be accessed by the host program to pass data.

3.2.3 Push constants
Push constants is a new feature in Vulkan and they are an alternative way to communicate between C++ and GLSL outside of uniforms. They provide fast updates and can be used for variables that changes for each object, like the translation matrix.

3.2.4 Command buffer
Command buffers are represented by VkCommandBuffer in Vulkan. All rendering commands are recorded to command buffers. When a command buffer is generated and ready it is submitted to the graphics card.

3.2.5 Command pool
Command buffers are not allocated directly but instead from command pool objects, represented by VkCommandPool in Vulkan. To avoid synchronization problems it is appropriate to have one VkCommandPool for every thread.

3.3 IMPLEMENTATION DETAILS

This section will explain some of the details in the implementation with the goal to describe how the benchmarks were obtained.
3.3.1 Rendering interface
In order to make the different implementations easy to change between and to not rewrite unnecessary code a rendering interface is used. Both the Vulkan and OpenGL renderers inherits from this interface and since the interface is purely virtual they are forced to overwrite every function. This makes it very convenient to write the test cases since there is no need to think about which renderer we are testing, from the outside they work exactly the same. This is the implementation of the Renderer interface:

```cpp
class Renderer
{
    public:
        Renderer();
        ~Renderer();

        virtual void Cleanup(int numThreads) = 0;
        virtual void SetupMultithreading(int numThreads) = 0;
        virtual void Render() = 0;
        virtual void Update() = 0;
        virtual void Init() = 0;
        virtual void HandleMessages(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam) = 0;
        virtual void OutputLog(std::ostream& fout) = 0;
        virtual void AddModel(StaticModel* model) = 0;
        virtual void SetCamera(Camera* camera) = 0;
        virtual void AddObject(Object* object) = 0;

        virtual int GetNumVertices() = 0;
        virtual int GetNumTriangles() = 0;
        virtual int GetNumObjects() = 0;
        virtual std::string GetName() = 0;
        virtual int GetNumThreads() = 0;
        virtual Camera* GetCamera() = 0;

    private:
};
```

Figure 4 Renderer.cpp

The Vulkan and OpenGL renderers now simply inherits from this interface, implement the virtual functions and adds the necessary class members to work. Their implementations of the functions will be very different from each other. An interesting note here is that the Vulkan implementation is ~2100 lines of code while the OpenGL one is ~350 lines of code, excluding helper functions for texture loading.

3.3.2 Creating the renderer
Game.cpp contains the class that creates the renderer, add all objects to the scene and sets up rendering parameters. Due to the use of a renderer interface the only code that changes when testing the different implementations is the creation of the renderer.

```cpp
// Test the OpenGL implementation
mRenderer = new VulkanLib::OpenGLRenderer(mWindow);
InitScene();

// Test the Vulkan implementation
mRenderer = new VulkanLib::VulkanRenderer(mWindow);
InitScene();
```

The InitScene() function contains a loop that loads models from disk and adds them to the scene at different locations. This is to make sure that the exact same scene is used in when testing both Vulkan and OpenGL.
3.3.3 Recording performance

As explained earlier performance is measured in Frames Per Second (FPS). To measure FPS the rendering loop contains both a time counter and a frame counter. The time counter gets incremented with the time each frame takes to render and the frame counter get incremented once per frame. Once the time counter reaches one second the value in the frame counter represents the FPS for that frame and is added to a vector. Then both counters are set to zero and the process begins again for the next FPS calculation. `std::chrono::high_resolution_clock` is used to get maximum accuracy when measuring time. In order to obtain a more reliable benchmark each test case runs for 60 seconds where the average FPS will be used.

In addition to the FPS of each test case the number of objects, renderer type, number of vertices and the number of threads used is printed to a text file. This is the data that is the base for the evaluation in the later part of this paper.

3.3.4 Vulkan renderer

The Vulkan renderer have support for multiple threads. To utilize these threads every thread is responsible for the command buffer generation of a set of the objects in the scene. When adding objects to the scene the objects are added to alternating threads, which in the end makes each thread contain NumObjects / NumThreads objects. Apart from containing a set of objects each thread also need their own command pool and command buffer.

In the rendering loop the main thread loops over the active threads and let each thread generate their own command buffer containing rendering commands for the specific set of objects the thread contains. The threads loops over their objects and calls `vkCmdBindVertexBuffers()`, `vkCmdBindIndexBuffer()` and `vkCmdDrawIndexed()` for each object to record to the command buffer. So if the application is using two threads and there are 100 objects in the scene then the command buffer of thread # 1 will contain the rendering commands for 50 of the objects and the command buffer of thread # 2 will contain the rendering commands for the remaining 50 objects. The main thread containing the rendering loop waits for each thread to be done generating their command buffers and then submits them to the GPU by calling `vkQueueSubmit`. The submitting of the command buffers is not done in parallel but compared to the generation of the command buffers its CPU time is not significant [+source].

3.3.5 OpenGL renderer

The OpenGL renderer is very barebones and the rendering loop simply iterates over each objects in the scene and calls `glBindVertexArray()`, `glUniformMatrix4fv()` and `glDrawElements()` for each object. The thing of note with the OpenGL renderer is that it does not use GLUT/GLFW for window creation. Instead it relies entirely on the Win32 for window and context creation. The reason for this is to utilize the same Win32 message loop that the Vulkan renderer uses. This is important since the performance of the program must be recorded in the exact same way for both renderer implementations, without relying on 3rd party functions from GLUT/GLFW and the risk of uncontrollable overhead.

3.3.6 Shader

An equivalent shader is used in both Vulkan and OpenGL for a fair comparison. The difference between the two is that the Vulkan vertex shader uses push constants for an objects world matrix while the Vulkan vertex shader uses a uniform. The vertex shader transforms the vertices to world coordinates and the fragment shader does phong shading with colours and no texture.

3.3.7 Camera

Camera.cpp contains the class that handles calculations of the view and projection matrices. Utilizing GLM for the vectors and matrix calculations. The camera can be controlled with the keyboard and
mouse and move around in the world. Both renderers use the same camera and during the performance tests the camera has the exact same position, target and aspect ratio.

### 3.3.8 Loading models
Models are loaded from disk using two different methods depending on the renderer implementation. The Vulkan renderer uses Assimp which make it possible to load a large amount of different 3D model formats. The OpenGL renderer on the other hand uses utility functions by Ingemar Ragnemalm from the TSBK07 course. I have examined the loaded data from the same model with both model importing methods and they yield identical vertex and index data, which is crucial for the performance testing.

To make loading many of the same model fast a model only has to be loaded once from disk and then it gets reused. However, each thread has to have their own unique copy of model memory to avoid synchronization issues.

### 3.4 TEST SYSTEM
The program will run the tests on a single computer with the following components.

- Windows 10 Pro (64 bit)
- Intel i5 750 @ 2.67 with four cores
- Radeon 280x 3GB VRAM
- 8GB RAM
- 16.7.3 Radeon driver

All tests will run at 1280x1024 resolution. The Vulkan version used is 1.0.2 and the OpenGL version used is 4.5.

### 3.5 TESTING

#### 3.5.1 Multi-threading tests
All the different tests will run for 60 seconds with no other applications running at the same time. When 60 seconds have passed the test ends and the average frames per second is calculated and the result is printed to a log file. When performing the tests all Vulkan debug layers are deactivated in order to provide maximum performance.

By using the Visual Studio Diagnostic Tool while running the program a diagram can be generated that displays the CPU and GPU utilization during runtime. The diagnostic tool will be used to test the utilization when using one and four threads with Vulkan and when using single threaded OpenGL. The generated diagrams are meant to give an overview over how using more threads will affect the CPU utilization. In the diagrams displayed the first 5 seconds are cropped to remove the loading of resources and initialization from the diagram.

Combined with the utilization graphs the recorded performance will be presented in frames per second along information about the renderer used, the number of objects and the number of vertices in the scene. A correlation between the CPU and GPU utilization and the performance will be examined.

The scene will be rendered with two different models, one with low detail and one with higher detail. The higher detail model contains 20 times more vertices than the low detail model and the goal of these tests are to find out what kind of performance increases multithreading yields in both scenarios.

All tests will use the same simple vertex and pixel shader. Using a more complex shader would increase the GPU load of the tests which in turn should result in less increase in performance when using multiple threads, similar to when rendering detailed models.
3.5.2 Pipeline management tests

In this test the new explicit pipeline used in Vulkan will be examined to find out what kind of performance changes there are when modifying pipeline stage states and how it compares to the OpenGL pipeline state modifications. In the tests there are three arbitrary states that will change between the renderings of every object, so 1000 objects in the scene means 3*1000 pipeline state changes every frame. The states that will be changed is the vertex shader, front face and culling face. In OpenGL this is done by calling `glUseProgram()`, `glFrontFace()` and `glCullFace()`. In Vulkan there will instead be two separate pipelines with different values for the shader, front face and cull facing states, changing the active pipeline is done by calling `vkCmdBindPipeline()`. 

As with the multi-threading tests these tests will also run for 60 seconds and the average FPS will be used as performance measurement. Since the goal of the explicit pipelines in Vulkan is to reduce CPU overhead this test will use a low detail model to make sure that the programs performance is CPU limited.
4 EVALUATION

In this chapter the results from the tests performed with the built application will be presented. The tests uses two different scenes, one using a low detail model and the other scene using a high detail model. First the results from running the test scene with the low detail model using single threaded OpenGL, single threaded Vulkan and four threaded Vulkan will be presented. After that the high detail model is tested in the same way. In addition to this the results from rendering both scenes using instancing will be presented. Finally the results from the pipeline state swapping test is shown.

4.1 LOW DETAIL MODEL TESTS

![Image of the low detail model test scene](image)

In the following tests a low detailed model with only 252 vertices will be used. It will be rendered 1000 times at different locations that are procedurally calculated. The simple shader that was described in the implementation will be used. If you have any experience with graphics programming you realise that instancing would be perfect to use when rendering this scene and as such instancing will be tested later in this chapter.
4.1.1 Single threaded Vulkan, low detail model

![Image of GPU utilization and CPU utilization graphs]

*Figure 6 Visual Studio Diagnostic Tool diagram with the low detail model scene using one threaded Vulkan*

Using only one thread gives very poor CPU utilization at around 21%. Since this test uses a very simple model with only 252 vertices, resulting in 252000 vertices totally in the scene, the graphics card is only utilized to around 26% since the CPU can’t keep up feeding vertices at the rate the GPU consumes them. This means that the program is CPU limited and should see great performance increase when utilizing more threads.

<table>
<thead>
<tr>
<th>Renderer</th>
<th>Vulkan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</td>
<td>1</td>
</tr>
<tr>
<td># objects</td>
<td>1000</td>
</tr>
<tr>
<td># vertices</td>
<td>252 000</td>
</tr>
<tr>
<td>Capture time</td>
<td>60 seconds</td>
</tr>
<tr>
<td>Average FPS</td>
<td>695</td>
</tr>
</tbody>
</table>

*Table 1 Recorded performance when using the low detail model scene with one threaded Vulkan*

The above are the recorded performance when rendering 1000 objects with Vulkan using one thread. Nothing can really be said about the average FPS recorded, running the test on a more powerful graphics card would result in a higher number, but it will act as a baseline for the other tests with the low detail model.

4.1.2 Four threads with Vulkan, low detail model

![Image of GPU utilization and CPU utilization graphs]

*Figure 7 Visual Studio Diagnostic Tool diagram with the low detail model scene using four threaded Vulkan*

With the same scene but now using four threads the CPU utilization is varying a lot over time but on average there is a great utilization increase from only using one thread. Due to this the GPU utilization have increased from 26% to 45% yielding a significant performance increase. However, even now the program is limited by a CPU that can’t deliver enough vertices to fully utilize the GPU. There is still a lot more performance to get out of the GPU.
The recorded performance have significantly increased when using four threads for command buffer generation. Going from 695 FPS to 1176 is a 69% performance increase. This increase in performance is directly dependent on the increase of CPU utilization as displayed in the diagram above. By using more threads on the CPU to generate rendering commands the GPU don’t have to wait for the CPU as much and the performance increases.

4.1.3 Single threaded OpenGL, low detail model

The utilization when using OpenGL is very similar to the utilization of single threaded Vulkan. The CPU is at around 25% utilization and the GPU at 27% utilization due to only one thread being used.

With the exact same scene the OpenGL renderer is 11% faster than single threaded Vulkan. This can be caused either due to better optimized drivers or inefficient use of Vulkan in my code. The latter being more likely due to Vulkan giving more control to the developer, providing increased performance if properly coded or decrease in performance if poorly coded.
4.2 **HIGH DETAIL MODEL TEST**

In these tests a model with greater detail will be rendered 1000 times. The model contains 5032 vertices resulting in totally 5 million vertices in the scene which means that the GPU will get stressed a lot more than in the low detail model tests.

4.2.1 **Single threaded Vulkan, high detail model**

Rendering the scene using Vulkan with one thread gives a very different graph from the tests with the low detail model. Here the CPU utilization is low while the GPU utilization is very high. This means that even with only one thread the CPU supplies vertices at a rate that keeps the GPU busy, suggesting that the program is GPU limited.
<table>
<thead>
<tr>
<th>Renderer</th>
<th>Vulkan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</td>
<td>1</td>
</tr>
<tr>
<td># objects</td>
<td>1000</td>
</tr>
<tr>
<td># vertices</td>
<td>5.03 millions</td>
</tr>
<tr>
<td>Capture time</td>
<td>60 seconds</td>
</tr>
<tr>
<td>Average FPS</td>
<td>132</td>
</tr>
</tbody>
</table>

Table 4 Recorded performance when using the high detail model scene with one threaded Vulkan

With 20 times the amount of vertices in the scene the large drop in framerate is to be expected. However, as seen from the diagram above this time it’s the GPU that limits the FPS to 132 and not the CPU.

### 4.2.2 Four threaded Vulkan, high detail model

![Visual Studio Diagnostic Tool diagram with the low detail model scene using four threaded Vulkan](image)

Using four threads the utilization of both the CPU and GPU have increased slightly, but far from the same magnitude between one and four thread utilization difference rendering the low detail model.

<table>
<thead>
<tr>
<th>Renderer</th>
<th>Vulkan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</td>
<td>4</td>
</tr>
<tr>
<td># objects</td>
<td>1000</td>
</tr>
<tr>
<td># vertices</td>
<td>5.03 millions</td>
</tr>
<tr>
<td>Capture time</td>
<td>60 seconds</td>
</tr>
<tr>
<td>Average FPS</td>
<td>143</td>
</tr>
</tbody>
</table>

Table 5 Recorded performance when using the high detail model scene with four threaded Vulkan

The performance of using four threads have only increased by 8% from 132 FPS to 143 FPS. This shows that using more threads for generating rendering calls does not yield a significant performance increase in every scenario. When rendering a lot of high detailed models with many vertices the GPU is well utilized and it does not matter if the CPU can supply it with vertices at a higher rate, the programs performance is still limited by the GPU.
4.2.3 Single threaded OpenGL, high detail model

The utilization is very similar to the one threaded Vulkan utilization when rendering the high detail models with Vulkan, so the diagram generated by Visual Studio is omitted.

<table>
<thead>
<tr>
<th>Renderer</th>
<th>OpenGL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</td>
<td>1</td>
</tr>
<tr>
<td># objects</td>
<td>1000</td>
</tr>
<tr>
<td># vertices</td>
<td>5.03 millions</td>
</tr>
<tr>
<td>Capture time</td>
<td>60 seconds</td>
</tr>
<tr>
<td>Average FPS</td>
<td>182</td>
</tr>
</tbody>
</table>

*Table 6 Recorded performance when using the high detail model scene with one threaded OpenGL*

When rendering the high detail model OpenGL has a quite large lead in performance with 37% more FPS than the one threaded Vulkan test and 27% more FPS than the four threaded Vulkan test. This big difference in performance when rendering the high detail model is not expected and it’s hard to know what’s causing it. In the low detail model tests, OpenGL showed a slight performance increase over single-threaded Vulkan as well, and as in that case it is likely that it’s my implementation that is not optimized.

4.3 INSTANCING

By using instancing with Vulkan on a single thread the CPU overhead from the rendering calls are almost completely removed. The same low detail model scene and high detail model scene will be tested with and without instancing with the same resolution.

<table>
<thead>
<tr>
<th></th>
<th>One thread</th>
<th>Four threads</th>
<th>Instancing</th>
<th>Performance increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low detail</td>
<td>695 FPS</td>
<td>1176 FPS</td>
<td>2424 FPS</td>
<td>249%</td>
</tr>
<tr>
<td>High detail</td>
<td>132 FPS</td>
<td>143 FPS</td>
<td>144 FPS</td>
<td>9%</td>
</tr>
</tbody>
</table>

*Table 7 Recorded performance of instancing combined with the other Vulkan performance results*

From this table you see that instancing has the potential of the greatest performance increase, with the limitation of only being able to use the same model data for each instance. When rendering the high detail model the program is limited by the GPU it does not matter if you split up the CPU overhead in multiple threads or if you remove the overhead completely by instancing, there still won’t be any significant performance increase.

4.4 PIPELINE SWAPPING PERFORMANCE

Here the facing, culling and shader pipeline stage states will be modified between the rendering of each object and the performance is presented in two different tables.

4.4.1 Changing facing, culling and shader

<table>
<thead>
<tr>
<th>Renderer</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vulkan</td>
<td>352 FPS</td>
</tr>
<tr>
<td>OpenGL</td>
<td>135 FPS</td>
</tr>
</tbody>
</table>

*Table 8 Recorded performance when changing facing, culling and shader pipeline states*

When changing the facing, culling and vertex shader states for every object the Vulkan explicit pipeline design shows a significant performance advantage with a performance increase of 160%.
4.4.2 Changing facing and culling

<table>
<thead>
<tr>
<th>Renderer</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vulkan</td>
<td>354 FPS</td>
</tr>
<tr>
<td>OpenGL</td>
<td>580 FPS</td>
</tr>
</tbody>
</table>

*Table 9 Recorded performance when changing facing and culling pipeline states*

When only changing the facing and culling states the OpenGL program is 63% quicker. The performance penalty for changing pipelines in Vulkan seems to be consistent no matter which states that gets changed. OpenGL on the other hand is faster in some cases but when changing the shader state the performance drops significantly.
5 Conclusion

This chapter summarizes what the results from the evaluation chapter means and how significant the performance increases are. First the multithreaded performance is discussed, what it can mean for both developers and users, and also how it affects the use of instancing. Finally the result from the pipeline tests are discussed.

5.1 Multithreading

The test results shows that using multiple threads for command buffer generation significantly increases the performance if the program is CPU limited. In the test with the low detail model enabling four threads provided a performance increase of 69% with the hardware used in the test. This performance increase will vary greatly depending which hardware that is tested and what scene is used. The test with the high detail model only showed a performance increase of 8% between one and four threads, due to the program being GPU limited. As Pawel L. [17] puts it, “If someone doesn’t need multithreading or if the application isn’t CPU bound, OpenGL is enough and using Vulkan will not give any performance boost”, this is exactly what the multithreaded test results shows.

Using multiple threads for generating rendering calls yields the greatest performance increase when rendering large amount of simple geometry. This is also where the use of instancing provides great performance increases by completely removing the CPU overhead for all the rendering calls. In the test cases with the low detail model instancing provided a performance increase of 350% over standard rendering calls on one thread. Similar to four threaded command buffer generation instancing only provided 9% increased performance when using the high detail model.

The multithreaded support that comes with Vulkan has great potential to increase the performance of a program. For users that have an older multicore CPU that can’t keep up its single core performance with the new graphics cards this is especially true. By utilizing all the cores on the CPU it will be like unlocking free performance simply by using a well-engineered Vulkan based renderer.

As a developer the support for multithreading command buffer generation in Vulkan does not make instancing obsolete. In situations where possible use instancing, otherwise use multithreaded command buffer generation for the best performance.

5.2 Pipeline states

In the pipeline state change tests the explicit Vulkan pipeline design provided a more consistent performance decrease when changing states between each object. The OpenGL implementation was faster when only changing facing and culling but a lot slower when changing the vertex shader state. This is due to changing the vertex shader state being a heavier operation with more error checks, which in the case of OpenGL has to be performed at runtime. By doing these error checks at the pipeline creation Vulkan is able to provide better performance.

However, as a developer you should still avoid changing states as much as possible. By using techniques such as grouping where objects using the same pipeline are rendered after each other unnecessary pipeline changes can be avoided.

The downside of the explicit pipelines in Vulkan is that you have to plan which pipeline states that will be used ahead of time and create all the required pipelines at initialization. With OpenGL it’s convenient to be able to call a function that immediately changes an individual pipeline stage state.
5.3 EXPERIENCE OF USING VULKAN

The Khronos Group have not been hiding the fact that Vulkan is significantly more complex to use properly than OpenGL. It is one of the main concepts of Vulkan in fact. Giving the developers more control, for example by removing responsibilities from the driver, gives room for optimizations and performance gains for developers. What it also does is that it reduces the ease of use and increases the chance of shooting yourself in the foot, if you do not know what you are doing. This is my experience of writing the test program used in this report as well. You have to deal with a lot of low level code that in OpenGL was handled by the driver.

For example many structures in Vulkan has flags that explicitly needs to be defined at creation, depending on different usages, to tell the driver how the structure will be used. There are no magic in the driver that figures out what you want to do, instead you explicitly tell it what to do. This requires developers to have great knowledge of the architecture in order to use Vulkan properly.

Sending resources to shaders in Vulkan requires a lot more effort than in OpenGL. If you want to change the value of an integer in OpenGL you can simply call glUniform1i() and pass the uniform location and the value. In Vulkan you have to manage the device memory for the uniform yourself using VkDeviceMemory, making sure that the correct size and usage flags are used. When you want to update the uniform you must first map the device memory and then use memcpy(). You must also bind the uniform manually when you want to use it. But you don’t bind the uniform itself; instead you bind the descriptor set that the uniform is a part of.

These were just some examples of what I found made Vulkan hard to work with from my limited experience. Overall I believe Vulkan is more time consuming to use than OpenGL but once you got a code base, some own wrappers and more experience I believe that Vulkan will not be that intimidating any more.
6 FUTURE WORK

As future work it would be interesting to perform similar tests focusing on multithreading with Microsoft’s rival graphics API DirectX 12. Vulkan and DirectX 12 are both designed as low level graphics API’s and they have very much in common from both being inspired by AMD Mantle. Evaluating DirectX 12 and comparing the results with Vulkans performance would show which API currently has the lead in performance.

One feature of Vulkan is the support for many different platforms. As such extending this work to test more platforms would be a great addition. Seeing how the performance changes when running Vulkan on Windows vs Linux on the same hardware would be interesting. Testing the support of Vulkan on both AMD and Nvidia graphics card could also be made as future work.
7 REFERENCES


