Multi-GPU Rendering with the open Vulkan API

Lars Olav Tolo[1], Ivan Viola[2], Atle Geitung[1], Harald Soleim[1] and Daniel Patel[1,3]

[1]Western Norway University of Applied Sciences
[2]TU Wien, Austria
[3]Christian Michelsen Research, Norway

Abstract

The Vulkan API provides a low level interface to modern Graphics Processing Units (GPUs). We demonstrate how to use Vulkan to send commands explicitly to separate GPUs for implementing platform,- and vendor independent multi-GPU rendering. We describe how to implement the sort-first and sort-last approaches to perform parallel rendering with Vulkan. We introduce an abstraction library which we have made available, and an application for multi-GPU rendering of meshes. Performance benchmarks have been performed in order to evaluate the implementation. We also show that we can utilize the additional GPU memory from multiple GPUs to render larger data sets than possible with a single GPU.

1 Introduction

Graphics Processing Units (GPUs) are dedicated hardware for accelerating computer graphics. To implement applications with computer graphics, we can utilize a graphics Application Programming Interface (API). These graphics APIs provide an interface to GPUs in a vendor neutral way. For demanding graphics rendering, using multiple GPUs can improve performance. However being able to use multiple GPUs from different vendors is not straight forward. Few graphics APIs have the features necessary to implement a general solution to multi-GPU rendering. With recent graphics APIs providing a lower level interface than before, developers have more freedom to manage details of the rendering process. This paper describes how to use Vulkan Application Programming Interface (API) to achieve multi-GPU rendering. We introduce an open source abstraction library for this purpose, and a prototype application that uses the library for sort-first and sort-last rendering of meshes.

Being able to select explicitly which GPU to assign work to, is important for implementing a solution for multi-GPU rendering. Core OpenGL does not support explicit selection of GPUs on the Windows operating system. To be able to assign work to each GPU explicitly with OpenGL, we have to rely on extensions specific to GPU vendors. AMD's GPU association extension [1] and NVIDIA's GPU affinity extension [2] provide the functionality of creating OpenGL contexts associated with

This paper was presented at the NIK-2018 conference; see http://www.nik.no/.

specific GPUs. Though AMD's extension target both consumer and professional GPUs [3], NVIDIA's extension is limited to their professional lineup (Quadro) only [4]. Hence, these extensions can not be considered a general solution to implement explicit multi-GPU rendering on consumer GPUs.

Explicit selection of GPUs is possible with Direct3D. Direct3D is however designed for the Windows operating system and is not an open API. However, Vulkan is an open API, which most likely will be supported by drivers and most operating systems for the foreseeable future.

Vulkan API launched 16th February 2016 and is "a new generation graphics and compute API that provides high-efficiency, cross-platform access to modern GPUs [...]" [5]. Contrary to OpenGL, where explicit control over GPUs is only available by extensions, with Vulkan, GPUs must be accessed explicitly. This provides the flexibility to implement multi-GPU rendering in a general solution for all Vulkan-capable GPUs, including consumer grade products.

2 Related work

Different ways of subdividing rendering into individual tasks that can be performed in parallel has been described by Molnar et al. [6]. They introduced the concepts of sort-first decomposition which is a partitioning of the screen into tiles, each being rendered in parallel, and sort-last decomposition which partitions the data instead, for instance the triangles of a 3D mesh. For both decompositions, the resulting renderings must be composited to one final image.

A large amount of research has been performed on rendering strategies when utilizing clusters of computers [7]. In such settings, low latency and scalability is central. Our work is orthogonal to these works as it can be used in any framework for utilizing multiple GPUs on each computer node.

Hardware-specific multi-GPU and multi-CPU rendering frameworks have been released, such as NVIDIA Optix [8] for NVIDIA GPUs, AMD RadeonRays [9] for AMD GPUs and Intel Embree [10] for Intel Xeon CPUs. These solutions offer APIs for raytracing and are tailored for specific hardware whereas ours interface the low level Vulkan API on any Vulkan compatible GPU.

NVIDIA's Scalable Link Interface (SLI) [11] and AMD's Crossfire [12] offer automatic, but limited utilization of several graphics cards residing in one computer. This is handled by the graphics drivers and does not require a change in the application itself. Both vendors offer two types of parallelization, alternate frame rendering where altenate frames are rendered by alternate GPUs, and split frame rendering where the rendering window is split and each GPU renders a part of it. SLI only works with identical NVIDIA GPUs while Crossfire can work with different AMD GPUs. Our solution can utilize any combination of Vulkan capable GPUs, regardless of vendor or model.

3 Multi-GPU with Vulkan API

The Vulkan API provides explicit access to GPUs allowing the developer to allocate resources, and perform rendering and compute operations. Vulkan has no global state, as all state is represented by objects. This provides the opportunity to implement multi-threading without context switches, as there is no global context to take into account.

Vulkan, and most other graphics APIs implements graphics rendering using the graphics pipeline [13]. The graphics pipeline is a series of steps to produce pixels on the screen from 3D geometry. This pipeline is accelerated on the GPU to improve performance. With Vulkan, we have to initialize the resources necessary for rendering on a per-GPU basis. This adds some upfront work for the developer, compared to higher level graphics APIs like OpenGL. However, it also allows flexibility to schedule different rendering commands to different GPUs, which is necessary to implement a general multi-GPU implementation.

When implementing an application utilizing Vulkan, we first create a Vulkan instance [14, p. 38], from which we can enumerate physical devices [14, p. 44] (see Figure 1). These physical devices represents the Vulkan capable GPUs in our system. However all interaction with a GPU happens through a logical device [14, p. 56]. The logical device represents a subset of the functionality available for a physical device. When using multiple GPUs we create one logical device for each of them.

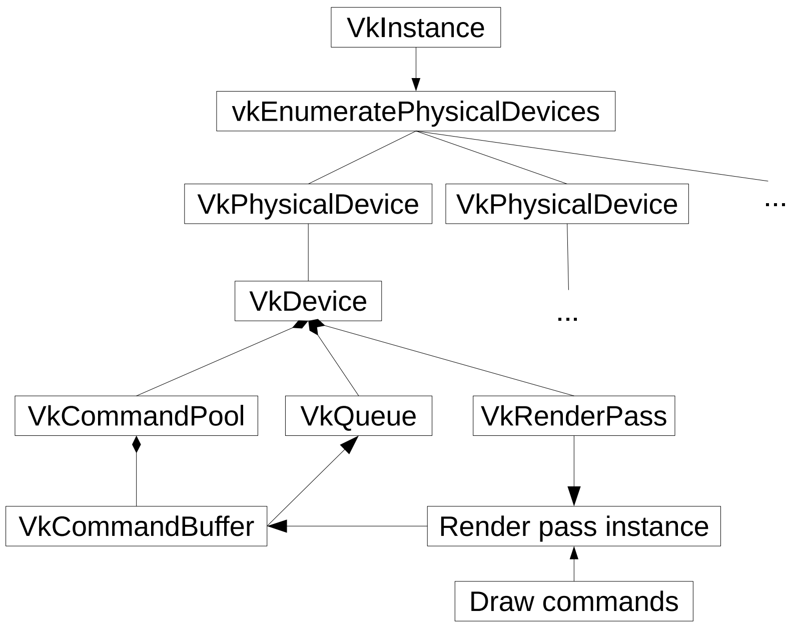


Figure 1: Vulkan objects used for multi-GPU rendering, and how they relate. The arrows show how we can use the API to record draw commands and the render pass instance they belong to into a command buffer. Command buffers are submitted to queues.

From a logical device we can create resources, allocate device memory, execute commands, bind resources to pipelines. We can perform rendering with graphics pipelines, and execute compute shaders with compute pipelines. Drawing with graphics pipelines must happen inside a render pass.

To execute commands on a GPU we must submit command buffers, which are batches of commands, to a queue. Command buffers are allocated from command pools [14, p. 70], which must be created from a logical device.

We have to specify which queues to create upon creation of the logical device. Queues belong to queue families that describe the capabilities of the queue, for instance graphics- and compute capabilities [14, p. 63]. Information about available queue families can be queried from the physical device. Multiple queues can execute commands in parallel, though the graphics driver must distribute the computational units of the GPU. Queue priorities allow us to hint to the driver which queues to prioritize more.

Render passes [14, p. 187] encapsulate attachments, subpasses and subpass dependencies. The attachments describe the layout of framebuffers that can be rendered to. Drawing commands can only be recorded within a subpass of a render pass instance. Multiple of these subpasses can be executed in parallel, and subpass dependencies provide synchronization guarantees between them. Upon starting a render pass instance, we have to specify a framebuffer object. This framebuffer encapsulates information about the specific attachments (texture, depth buffer, etc) that will hold the result from rendering.

Resources and memory are separated in Vulkan. There are two types of resources in Vulkan, buffers [14, p. 335], and images [14, p. 345]. Resources must be bound to memory objects before use. These memory objects [14, p. 293], are allocated from the logical device. The memory can either be device local (GPU memory), or host local (main memory). Device local memory is usually not visible for the host, so in order to transfer data to GPU memory, it must first be loaded into a host local staging buffer, which can then be copied to the device local memory through the API.

Vulkan only supports shaders in the byte code format SPIR-V [15]. To use shaders, we must create shader modules [14, p. 229], which are combined in pipelines [14, p. 247]. We must bind a graphics pipeline inside a render pass instance before our draw commands. We can dispatch compute shaders with compute pipelines.

4 A Vulkan Abstraction Library

To take care of the low-level Vulkan details described above, we have made publicly available the library Boilerplate (BP) [16], a cross platform abstraction library implemented along with the multi-GPU application. BP consist of C++ wrappers of Vulkan objects, and abstractions hiding complexity of the API. The core module `bp` provides the wrappers and abstractions of Vulkan, and additional modules can be utilized for creating e.g. windows and scenes.

While many of the classes implemented with `bp` are simple wrappers of Vulkan objects, some of the classes provide a higher level abstraction that hides complexitites of Vulkan. Examples of the latter are devices, memory allocators, attachments, subpasses and render passes. To take care of the multi-GPU details of parallelization and compositing, the module `bpMulti` provides classes for implementing sort-first and sort-last rendering.

Instance and Device

The first step when implementing an application utilizing Vulkan API, is to create an instance object. If the intention is to use the `bpQt` module for creating a Qt window, a `QVulkanInstance` object, must be created. Otherwise, we can use the abstraction `bp::Instance` together with the `bpView` module, or other window libraries that can provide a Vulkan representation of the window.

The next step is to create a logical device, represented by `bp::Device` for each of the GPUs that should be used. We create devices with at least a graphics capable queue. We can also specify that the device should be capable of presenting its result to a specific window.

A `bp::MemoryAllocator` object is provided by the device, which is used to allocate memory for buffers or images. This abstraction is a wrapper of Vulkan Memory Allocator [17], an open source library from the GPUOpen initiative by AMD. The memory allocator allocates memory optimally in large chunks, rather than using small allocations for each resource.

Rendering with Boilerplate

The render pass abstraction provided by BP, `bp::RenderPass`, takes care of recording to a command buffer the commands for a render pass instance, and the subpasses within that render pass instance. Subpasses, represented by an abstract class `bp::Subpass`, must be added to the render pass before initialization.

Draw commands can only be executed inside a subpass of a render pass instance. To implement a subpass, we must inherit `bp::Subpass` and override the `render` method. Subpass dependencies can be added to a subpass with the `addDependency` method. Attachments to use in a subpass are represented by `bp::AttachmentSlot` objects. Subpass dependencies, as well as attachment slots must be set before adding the subpasses to a render pass.

When executing a render pass instance, we must provide a framebuffer, represented by `bp::Framebuffer`, that holds attachments for each of the attachment slots in use by the subpasses. The abstract class `bp::Attachment` represents a framebuffer attachment, and is inherited by `bp::Texture` and `bp::Swapchain`. A `bp::Texture` object encapsulates a single offscreen image, and a `bp::Swapchain` represents a series of images used to render to a window or display. A simple renderer would usually implement a swapchain as color attachment, and a texture as depth attachment.

To execute the commands on the GPU we must submit the recorded command buffer to the graphics-capable queue. When the commands have finished executing, our attachments will contain the rendering result.

5 Multi-GPU Implementation

We have made available the source code [18] of a prototype application that supports loading 3D meshes from file, and rendering with either sort-first or sort-last multi-GPU approaches. Each GPU renders its contribution to a texture. The textures are composited to the final frame by one of the GPUs, before it is presented to the screen. The current implementation executes five steps in order to render a frame:

1. Rendering the contributions for each GPU into textures.
2. Copy the contributing textures from the secondary GPUs to the host (CPU and main memory).
3. Redundant memory to memory copying step (discussed later) of the contributing textures.
4. Copying the textures from host to the primary GPU.
5. Combining the contributions into the final image in a compositing step.

If these steps are executed sequentially every frame, the overhead from the three copying steps would be a bottleneck for performance. In order to increase the frame rate, functional parallelism (pipelining) is introduced, such that some stages can execute in parallel. This approach increases frame rate, however the latency from starting rendering a frame until presenting it on the screen is not improved. See Figure 2 for a timeline of the rendering process across a few frames.

It is possible to parallelize stages executing on a single GPU by submitting commands to different queues. This has been utilized to perform rendering on the primary GPU in

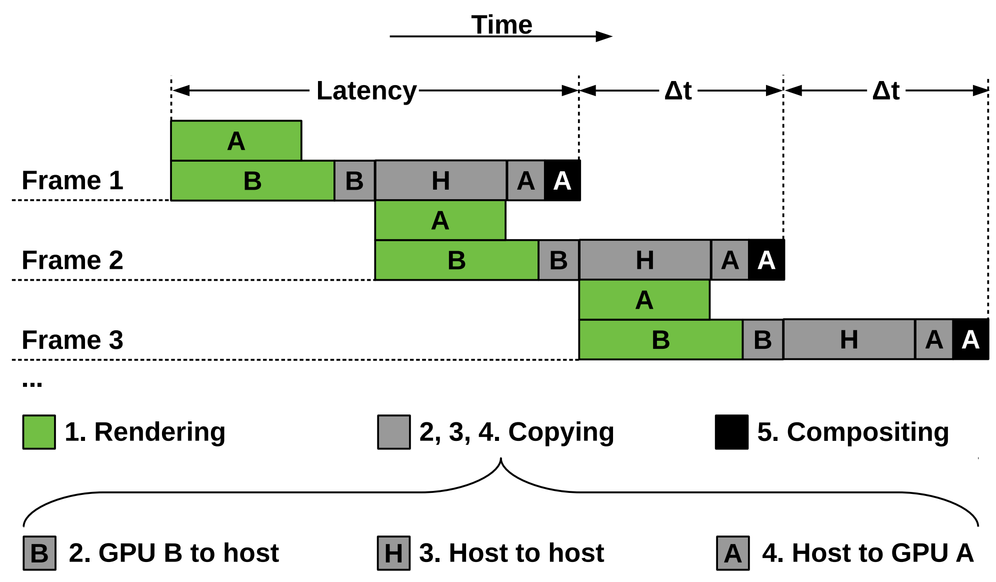


Figure 2: Illustration of a timeline for two GPUs A and B over three frames. H represents the host CPU and main memory. White steps are rendering, gray steps are copying, and black steps are compositing.

parallel with the last copying step, but requires the primary GPU to support a separate transfer-capable queue. When using multiple queues, the GPU and driver software takes care of scheduling the submitted tasks. Without using multiple queues, the rendering step and the compositing step on the primary GPU could be submitted in the same batch, by implementing two subpasses in a single render pass. This approach could be useful for GPUs that only support a single graphics capable queue, as multiple subpasses can in some cases be executed in parallel (depending on GPU and driver implementation).

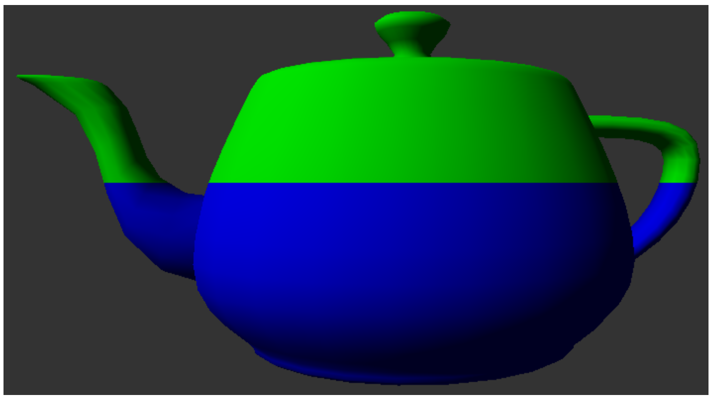
Two multi-GPU approaches are supported, sort-first and sort-last. For both approaches, the GPUs render their contributions into textures. Each GPU needs two textures used as framebuffer attachments: the color attachment and the depth attachment. In our implementation, the color attachment is implemented as a 32-bit RGBA texture and the depth attachment as a 16-bit depth texture. Both are represented by `bp::Texture` objects.

To render a mesh loaded by the application, a subpass have been implemented by inheriting `bp::Subpass`. This subpass is then executed in a render pass instance by each of the GPUs. After rendering the contributions of each GPU to textures, and copying the contributions from the secondary GPUs to the primary GPU, they must be composited into a final image. A subpass executed within a render pass, performs this compositing step. The compositing and parallelization of multi-GPU rendering is handled by compositor classes provided by the module `bpMulti`.

Sort-first

The sort-first approach is implemented by partitioning the screen into horizontal tiles of fixed height (see Figure 3a). Each GPU renders its tile to the framebuffer consisting of the color- and depth attachment textures. Then the color textures rendered on the secondary GPUs are copied into the primary GPU before composition. The textures contain the finished rendering for each tile, which must be copied into their correct screen positions.

For this, we draw a rectangle for each of the contributions, while sampling the color from the texture in the fragment shader.


(a) Screenshot illustrating the sort-first approach.

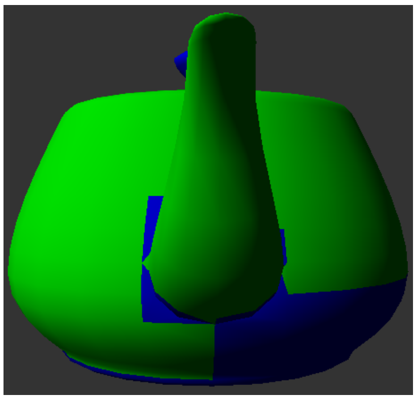

(b) Screenshot illustrating the sort-last approach.

Figure 3: Screenshots illustrating the sort-first and sort-last implementations running on two GPUs shown with two different colors.

Sort-last

Sort-last parallelization is implemented by partitioning the geometry (see Figure 3b). Each GPU renders its portion of the geomerty into textures of the same size as the render window. The contributing textures are copied from the secondary GPUs to the primary GPU. For sort-last, both the color- and depth textures must be copied, and the compositing must compare the fragment depths to decide from which texture to sample the color.

In order to composite the contributing textures into the final image, we draw a full screen quad for each contribution. In the fragment shader we sample the color from the color texture, and set the fragment depth to the value sampled from the depth texture. Then the depth test takes care of selecting the closest fragment.

Copying Contributing Textures

The time it takes to copy a texture from one GPU to another increases with resolution. Therefore the three steps for copying the contributing textures can quickly become a bottleneck.

The naive solution to copy a texture from one GPU to another, is to have a host-allocated staging buffer for each of the GPUs, then map these buffers and copy between them. Both of these staging buffers are located in main memory. Therefore copying between them is in theory a redundant step. The implemented copying process is to first copy from GPU memory to a staging buffer, then copy between the two staging buffers, before finally copying from the second staging buffer to the destination GPU.

In order to speed up the copying process for the redundant step, the memory is copied chunk-wise on multiple threads as a single thread is not able to utilize the entire memory bandwidth available. The tradeoff is high CPU usage.

To avoid the redundant copying step results in one less step to execute, increased frame rate and reduced CPU usage. For, we need a staging buffer shared between the source and destination GPUs. This could be possible using the Vulkan device extension `VK_EXT_external_memory_host` [14, p. 1247]. However, the extension requires driver support which is currently limited. AMD and NVIDIA have contributed in developing the extension [19]. Therefore it may be supported in future driver releases.

6 Results

In order to evaulate the implementation, performance have been measured for benchmarks tailored specific to the sort-first and sort-last approaches. Two different hardware configurations have been tested, both utilizing only dedicated GPUs. Testing Computer 1 (TC1) has three relatively powerful GPUs (Table 1a). Testing Computer 2 (TC2) has two GPUs from different vendors (Table 1b), which shows that the implementation is vendor agnostic and can work with any Vulkan capable GPUs.

CPU	Intel Core i7-6850K
RAM	64GB
GPU 1	Nvidia GeForce GTX 1080 8GB
GPU 2	Nvidia GeForce GTX 1080 8GB
GPU 3	Nvidia GeForce GTX 1080 8GB

(a) Specifications of testing computer 1 (TC1).

CPU	Intel Core i7-7700K
RAM	32GB
GPU 1	Nvidia GeForce GTX 760 2GB
GPU 2	AMD Radeon RX 460 2GB

(b) Specifications of testing computer 2 (TC2).

Table 1: Specifications of the testing hardware used for benchmarking.

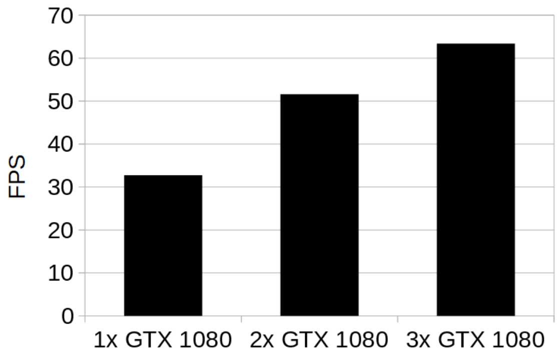
For both the sort-first and sort-last approaches, each GPU renders to off-screen framebuffers. The pixels from these framebuffers are combined to produce the final image. The difference between them is that the sort-first approach distributes pixels to render among the GPUs, while the sort-last approach distributes geometry. So a benchmark that shows performance increase for the sort-first approach, might not scale as well for the sort-last approach. This is why two different benchmarks have been implemented. All the benchmarks was executed with a screen resolution of 1024x768.

Sort-first Benchmark

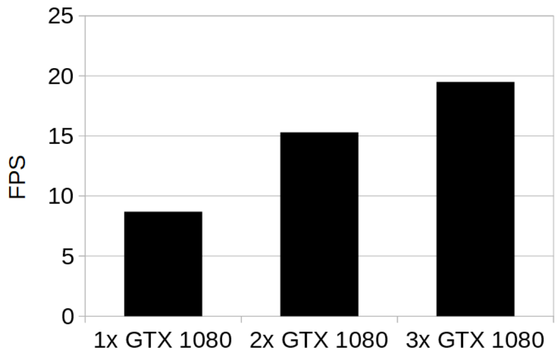
Since the sort-first approach distributes the pixels to render among the available GPUs, a benchmark suitable for evaluating the performance of the sort-first implementation would be demanding per pixel. One such use case is ray tracing. Ray tracing is a rendering technique that calculates the color per pixel by tracing light rays backwards into the scene. To simulate a per-pixel demanding workload, the implemented benchmark for the sort-first implementation draws a rectangle filling the framebuffer for each GPU, and performs demanding calculations in the fragment shader.

The performance results for the sort-first benchmark on TC1 (see Figure 4a), shows a 58% increase of frame rate from a single GPU to two GPUs. Adding a third GPU further increased the frame rate 22%, which is a 94% improvement compared to a single GPU. The GPU utilization was 79% and 65% for two and three GPUs respectively, compared to an ideal implementation.

On TC2 the sort-first results did not improve performance, as the difference in performance of the GTX 760 and the RX 460 was too large to get performance improvement in this benchmark. This is perhaps due to the difference in Raster Operations Pipeline (ROP) count, of which the GTX 760 has 32, and the RX 460 has 16. However for the demanding geometry implemented in the sort-last benchmark, the two GPUs performed similarly.



(a) Sort-first performance results.



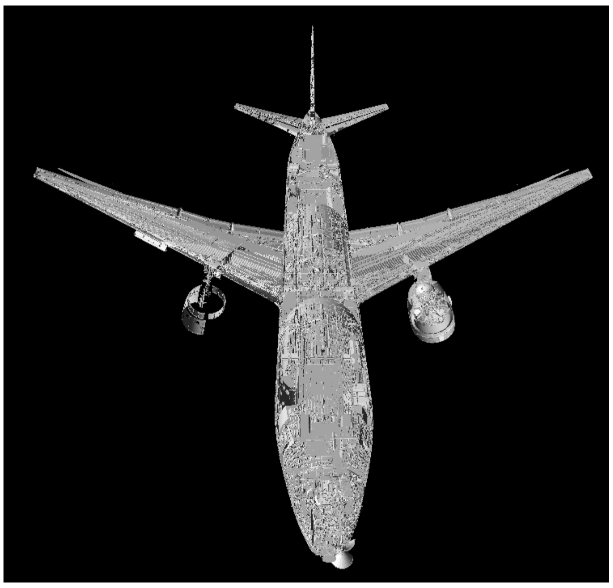
(b) Sort-last performance results.

Figure 4: Diagrams showing the performance for TC1. Performance is measured in Frames Per Second (FPS). Our solution successfully accelerates rendering when adding more GPUs

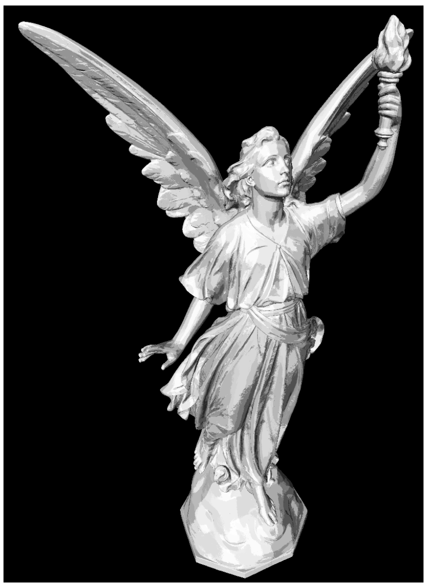
Sort-last Benchmark

The sort-last approach distributes geometry among the available GPUs. The benchmark used to evaluate the sort-last implementation is to draw a large enough amount of geometry, such that a single GPU is not powerful enough to achieve good performance. Then the geometry can be distributed between the GPUs to improve performance.

Since TC1 is significantly more powerful than TC2, the benchmark was run with different amount of geometry for each of the computers. TC1 was tested with a large mesh of a Boeing airplane (see Figure 5a), with more than 280 million triangles. This mesh was used with permission by the Boeing company. TC2 was benchmarked with a reconstructed mesh of a statue of Lucy (see Figure 5b), with more than 28 million triangles, provided by the Stanford 3D Scanning Repository [20]. The Lucy mesh was rendered twice, resulting in about 56 million triangles.



(a) Screenhot of the Boeing airplane model. Our copy of the model is missing some geometry, but it is still suitable for benchmarking.



(b) Screenshot of the reconstructed mesh of a statue of Lucy.

Figure 5: Screenshots of the meshes used for benchmarking the sort-last implementation.

The performance results from the sort-last benchmark rendering the Boeing airplane on TC1 (see Figure 4b), shows a 76% improvement from one to two GPUs, and 27% improvement from two to three GPUs, which is a 124% improvement from one to three GPUs. The GPU utilization is in this case 88% and 75% for two and three GPUs respectively, compared to an ideal implementation.

The results from the sort-last benchmarks on TC2 (see Figure 6) yielded a performance increase of 52% when utilizing both GPUs, compared to the GTX 760 GPU alone, or 71% compared to the RX 460 GPU on its own. The GPU utilization is 80%.

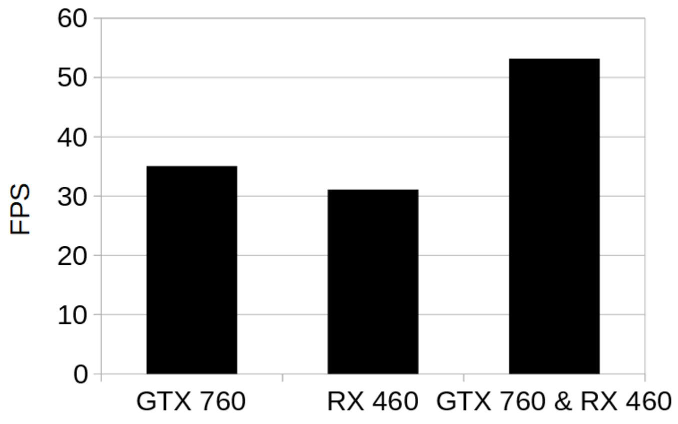


Figure 6: Diagram of the performance for TC2 for the sort-last benchmark. Performance is measured in Frames Per Second (FPS).

Utilizing Additional GPU Memory

While we have shown that we can utilize the sort-first and sort-last implementation to achieve higher framerates in specific scenarios, it is also possible to utilize the additional GPU memory available in a multi-GPU setup. We have performed rendering of geometry with high-resolution textures using the introduced sort-last implementation. We can render larger data sets using multiple GPUs than a single GPU, as we have access to more GPU memory.

The data set we used in order to perform rendering of geometry with high-resolution textures, consists of 3D models of the Beckwith Plateau (Book Cliffs, UT, USA) mountain side (see Figure 7). The Virtual Outcrop Geology group, Uni Research, Bergen is acknowledged for access to the 3D models used within this work. The data was acquired with the support of the Research Council of Norway and FORCE Sed/Strat group through the EUSA/SAFARI project (grant number 193059).

The data set consists of 12 sections of the mountain side with high-resolution textures. In total, the data set consists of more than 24 million triangles, and 4.3GB of textures in the compressed JPG format. The textures require more memory once uncompressed and transferred to the GPUs. We were able to render 3 sections with a single GTX 1080 GPU, on TC1. Utilizing all three GPUs, we were able to render 9 sections, at a framerate of 50FPS. In our testing we were not able to allocate the entire GPU memory available. This could be caused by fragmentation of the GPU memory.

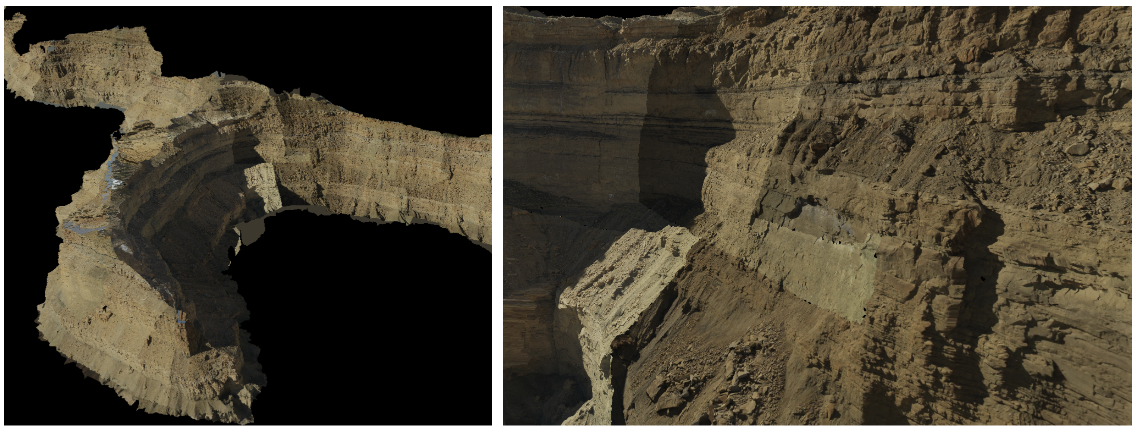


Figure 7: Screenshots from rendering the Beckwith Plateau, Book Cliffs, UT, USA. This geometry is too large to fit on a single GTX 1080 GPU.

7 Conclusion

We have shown that the prototype application for multi-GPU rendering with Vulkan API can perform well with the sort-first and sort-last approaches. We have also shown that we can utilize the additional GPU memory to render data sets that will otherwise not fit in the memory of a single GPU. The application is cross platform, and will work on most Vulkan capable GPUs. This is the first publicly available, heterogeneous multi-GPU solution implemented with Vulkan API. We believe this implementation has potential, and that performance improvements can be made by implementing pipelining of all the rendering steps, and support for the external memory Vulkan extension.

Future work

The main area of potential performance improvement is pipelining of the steps necessary. Currently there are five steps, three of which are copying steps from the secondary- to the primary GPU. Some pipelining have been implemented already, but more is possible. Having all five steps operate in parallel will increase the frame rate, and GPU utilization. However the latency will not improve. By implementing support for the external memory Vulkan extension, we can remove one of the copying steps, reducing some latency and possibly increase the frame rate, but also reduce CPU usage as the redundant copying step is removed.

Another area of improvement is to implement dynamic load balancing, such that the workload will be automatically distributed according to the performance of the GPUs. This way the faster GPU will do more work than the weaker one.

References

[1] N. Haemel. *AMD_gpu_association*. `https://www.khronos.org/registry/OpenGL/extensions/AMD/WGL_AMD_gpu_association.txt`. Accessed: 2017-11-24. Mar. 2009.

[2] B. Lichtenbelt. *WGL_NV_gpu_affinity*. `https://www.khronos.org/registry/OpenGL/extensions/NV/WGL_NV_gpu_affinity.txt`. Accessed: 2017-11-24. Nov. 2006.

[3] Advanced Micro Devices Inc. *AMD - GPU Association - Targeting GPUs for Load Balancing in OpenGL*. 2010.

[4] Derivative.ca. *Using Multiple Graphic Cards*. https://www.derivative.ca/wiki088/index.php?title=Using_Multiple_Graphic_Cards. Accessed: 2017-11-24.

[5] Khronos Group. *Vulkan*. https://www.khronos.org/vulkan/. Accessed: 2017-11-13.

[6] S. Molnar et al. "A sorting classification of parallel rendering." In: *IEEE computer graphics and applications* 14.4 (1994), pp. 23–32.

[7] Eilemann et al. "Parallel rendering on hybrid multi-gpu clusters." In: *Proceedings Eurographics Symposium on Parallel Graphics* (2012), pp. 109–117.

[8] Parker et al. "GPU ray tracing." In: *Communications of the ACM* (2013).

[9] AMD. *Introducing the Radeon Rays SDK*. Tech. rep. One AMD Place, Sunnyvale, CA 94088, 2016.

[10] Wald et al. "Embree: a kernel framework for efficient CPU ray tracing." In: *ACM Transactions on Graphics (TOG)* (2014).

[11] NVIDIA Corporation. *Introduction to SLI Technology*. https://www.geforce.com/whats-new/guides/introduction-to-sli-technology-guide. Accessed: 2017-11-15.

[12] Advanced Micro Devices Inc. *AMD Crossfire Technology*. https://www.amd.com/en/technologies/crossfire. Accessed: 2017-11-15.

[13] J. F. Hughes et al. *Computer graphics: principles and practice (3rd ed.)* Boston, MA, USA: Addison-Wesley Professional, July 2013, p. 1264. ISBN: 0321399528.

[14] Khronos Vulkan Working Group. *Vulkan 1.0.66 - A Specification (with all registered Vulkan extensions)*. 1.0.66. Nov. 2017.

[15] Khronos Group. *SPIR Overview*. https://www.khronos.org/spir/. Accessed: 2018-03-03.

[16] L. O. Tolo. *C++ Abstraction library for Vulkan API*. https://github.com/larso0/bp. Accessed: 2018-03-03.

[17] Advanced Micro Devices Inc. *Vulkan Memory Allocator*. https://github.com/GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator. Accessed: 2018-02-26.

[18] L. O. Tolo. *Vulkan Multi-GPU Application*. https://github.com/larso0/vmgpu. Accessed: 2018-03-03.

[19] M. Larabel. *Vulkan 1.0.66 Introduces Three New Extensions*. https://www.phoronix.com/scan.php?page=news_item&px=Vulkan-1.0.66-Released. Accessed: 2018-02-25.

[20] Stanford University. *The Stanford 3D Scanning Repository*. http://graphics.stanford.edu/data/3Dscanrep/. Accessed: 2018-03-01.