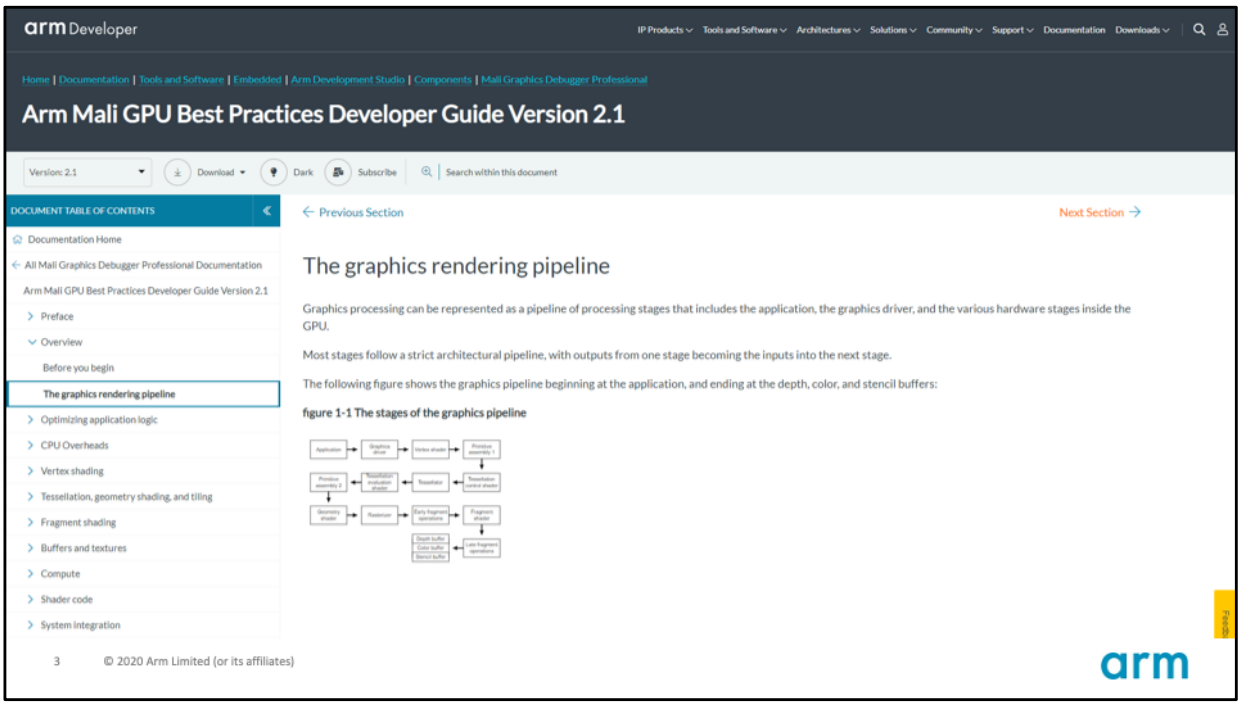

arm
Vulkan on Mobile Done Right
Moving Mobile Graphics
Jose Emilio Muñoz Lopez
SIGGRAPH 2020
© 2020 Arm Limited (or its affiliates)

Agenda

- Vulkan Samples introduction
- Render passes
- Immediate vs tile-based renderers
- Vulkan best practice
  - Load/Store operations
  - MSAA
  - Pipeline barriers
- Vulkan Samples framework
- Current and future work
- Further reading

arm

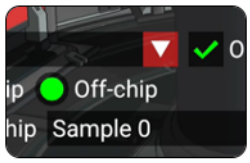
I will first introduce our Vulkan Samples, and after a quick recap on some Vulkan concepts I will take you through 3 examples of optimal API usage. Then I will close with a few details on the framework behind these samples and what we are doing with the samples next.



Here's Arm's developer guide. In here we collected many recommendations we gathered from our experience working with game developers and optimizing our hardware and software for graphics. We thought that the best way to understand these do's and don'ts was to try them for ourselves, and that's why we wrote the Vulkan Samples.

https://developer.arm.com/documentation/101897/latest

Vulkan samples



- Run samples
- Analyze counters
- Read tutorials
- Experiment on a mobile-optimized, multi-platform framework

https://github.com/KhronosGroup/Vulkan-Samples

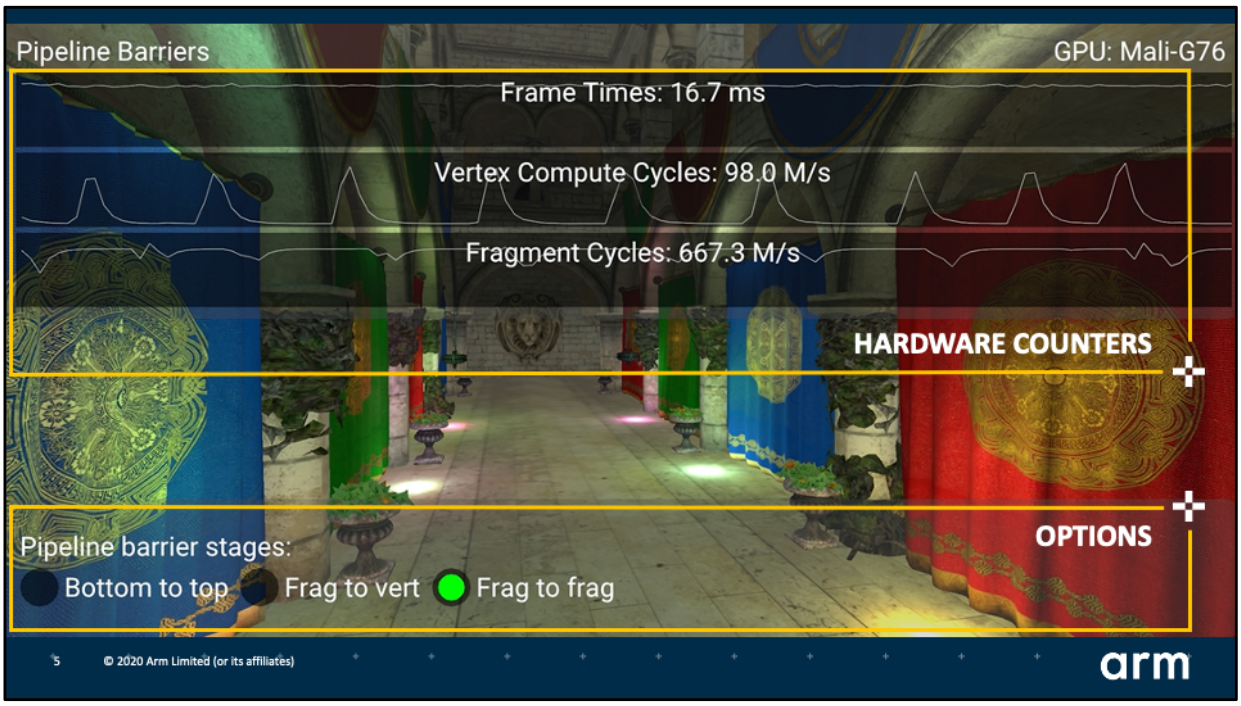
arm

The idea is that the samples should make it very easy to try out different ways of doing the same thing in Vulkan.

You can toggle between these different paths at run-time while monitoring the impact that they have on performance, thanks to the on-screen hardware counters. Finally, every sample is released alongside an article that explains the theory behind every best practice recommendation, with code snippets, diagrams and examples where necessary.

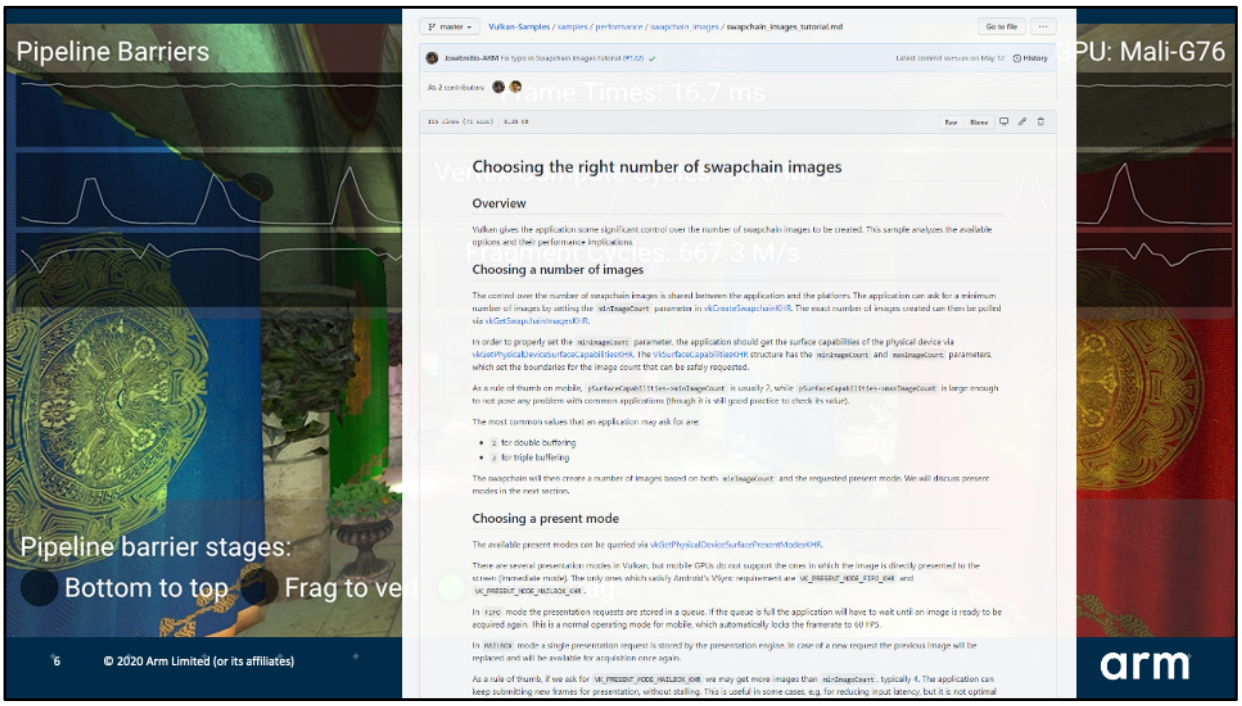
These have now been adopted by Khronos as the single repository for Vulkan Samples, so we are now working with other vendors and independent contributors to make it grow and improve.

https://github.com/KhronosGroup/Vulkan-Samples

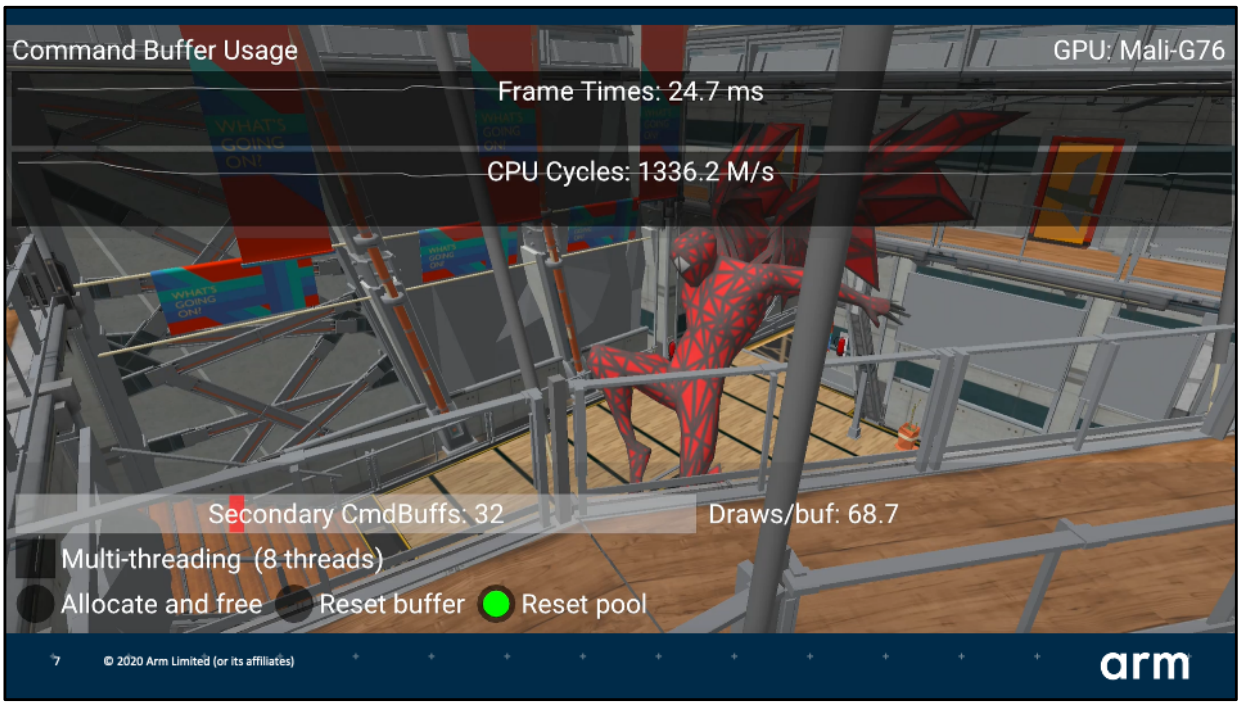


This is how they look like. At the bottom there will be some options to experiment with.

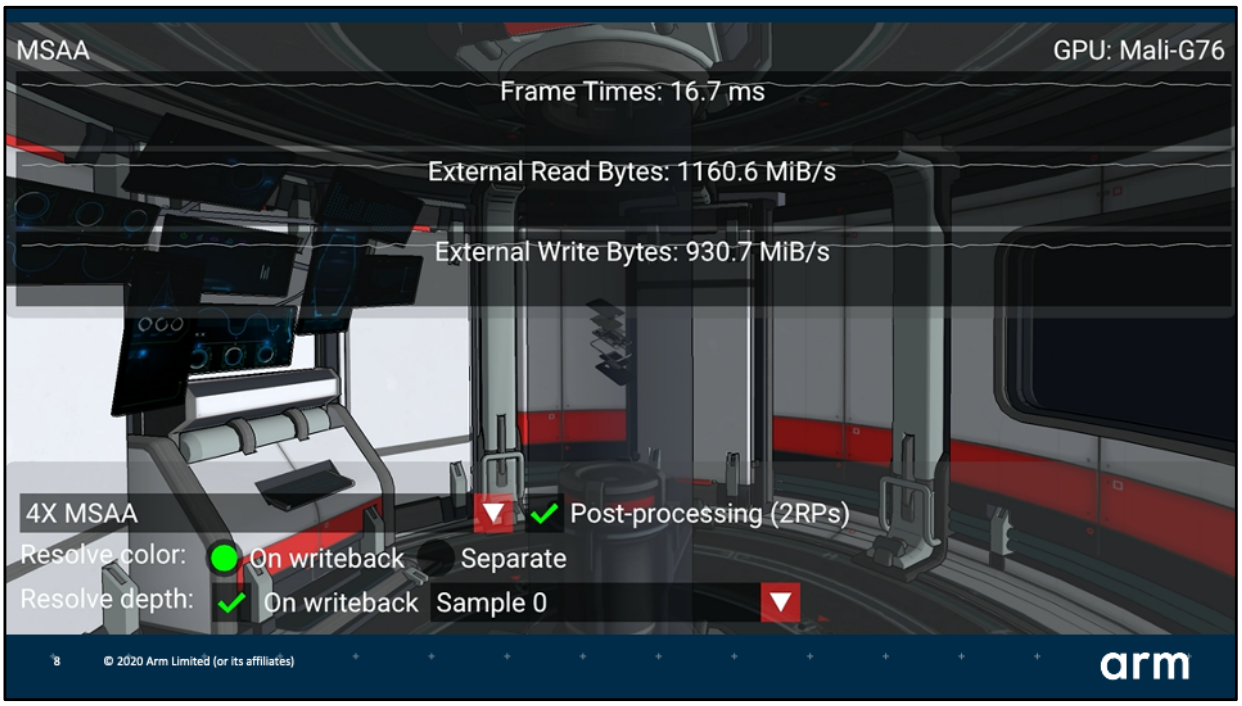
At the top samples show some stats running in real time, such as FPS, vertex and fragment cycles.

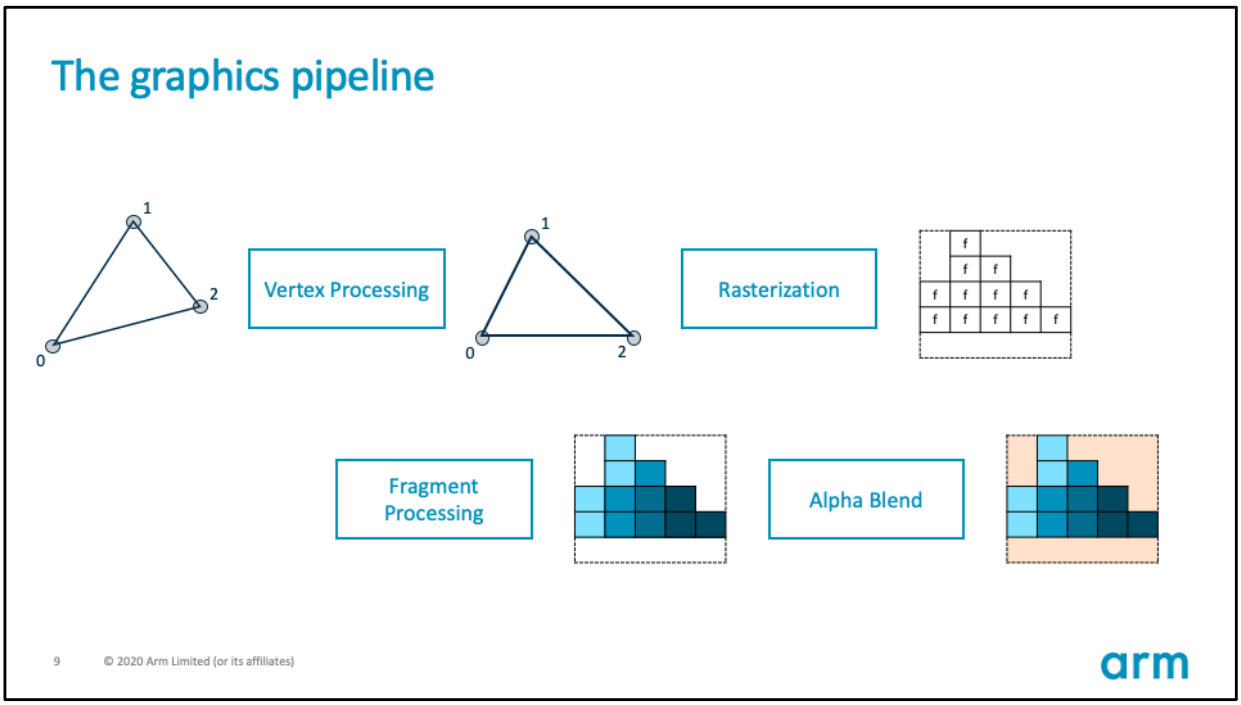


On GitHub every sample we release has a tutorial where we take you through the theory, overview of the sample configurations and expected performance, and at the end a summary of do's and don'ts and a guide to profile your own application using the tools available in the ecosystem.



You can also use one of three full scenes that we provide, or use your own as long as they are provided in the glTF format.


MSAA
GPU: Mali-G76
Frame Times: 16.7 ms
External Read Bytes: 1160.6 MiB/s
External Write Bytes: 930.7 MiB/s
4X MSAA
Post-processing (2RPs)
Resolve color:
On writeback
Separate
Resolve depth:
On writeback
Sample 0
8
© 2020 Arm Limited (or its affiliates)
arm



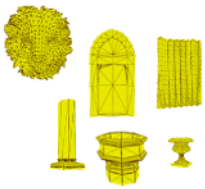
Arm's samples focus on performance, specially on considerations around limited power (and therefore bandwidth) on mobile devices.

Before we dive into the samples, let's go through a quick recap on GPU architectures and render passes.

You're probably familiar with the graphics pipeline.

For now let's focus on the Vertex and Fragment shading stages.

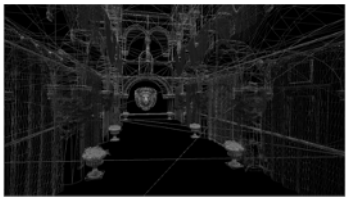
Renderpasses and subpasses

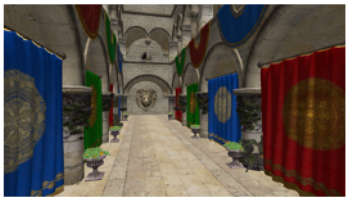


Vertex Processing

Rasterization

Fragment Processing





arm

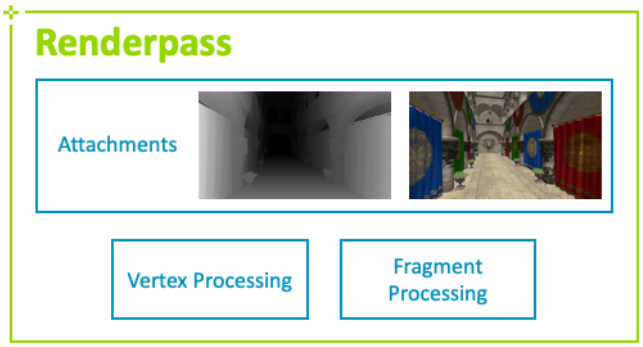
Consider models in the Sponza scene.

All vertices are transformed and projected in vertex processing.

The scene is rasterized.

And then, using 2 attachments, one for color and one for depth, the final image is computed.

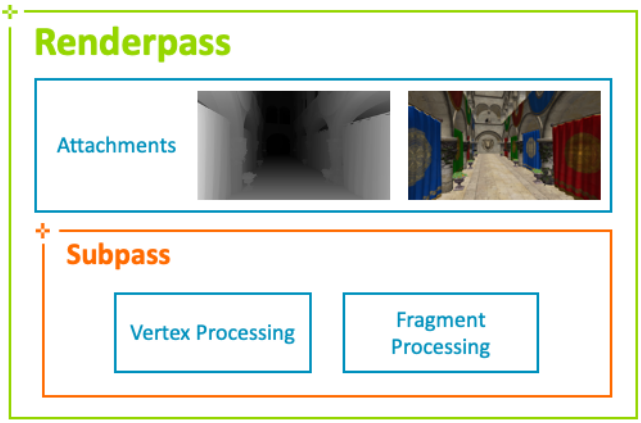
Renderpasses and subpasses



arm

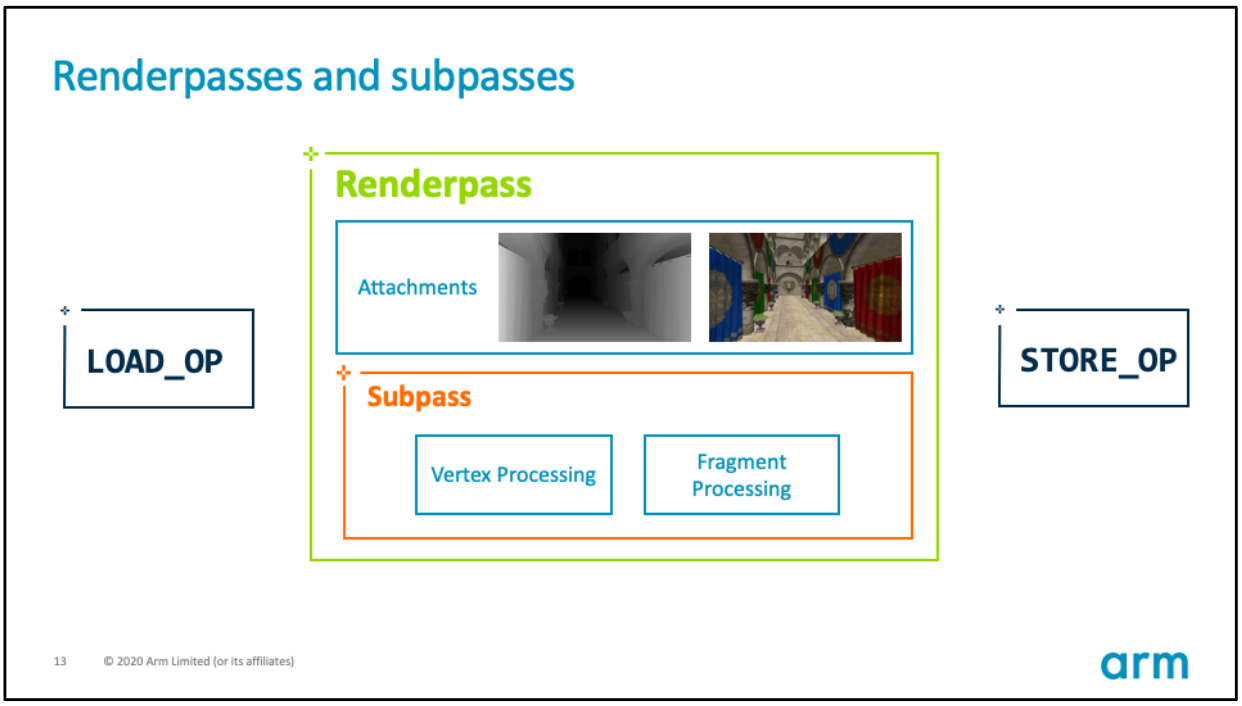
The collection of attachments and the way they are used is what is known in Vulkan as a Renderpass.

Renderpasses and subpasses



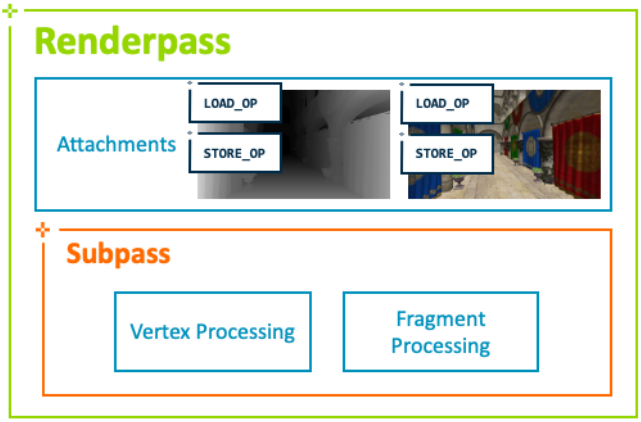
arm

Renderpasses consist of 1 or more Subpasses.



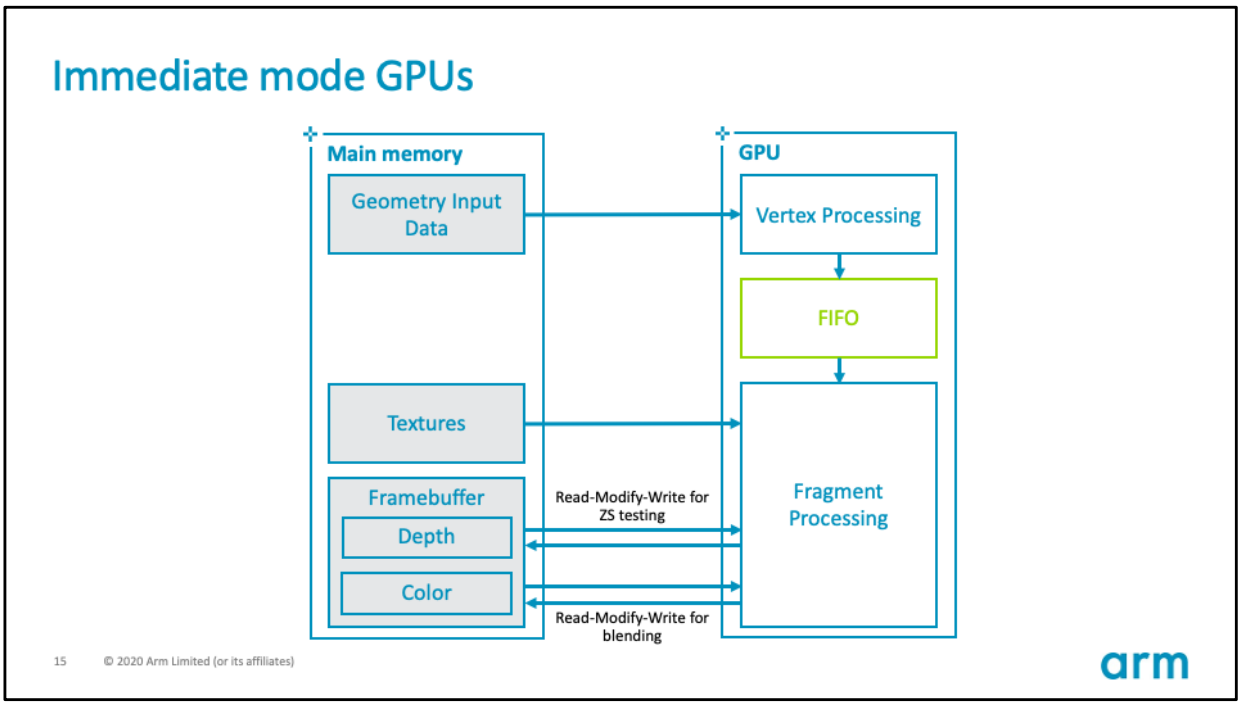
When defining a Renderpass and its attachments, in Vulkan we need to specify load and store operations, i.e. what to do with the attachment before and after rendering.

Renderpasses and subpasses



arm

We define these for each attachment.

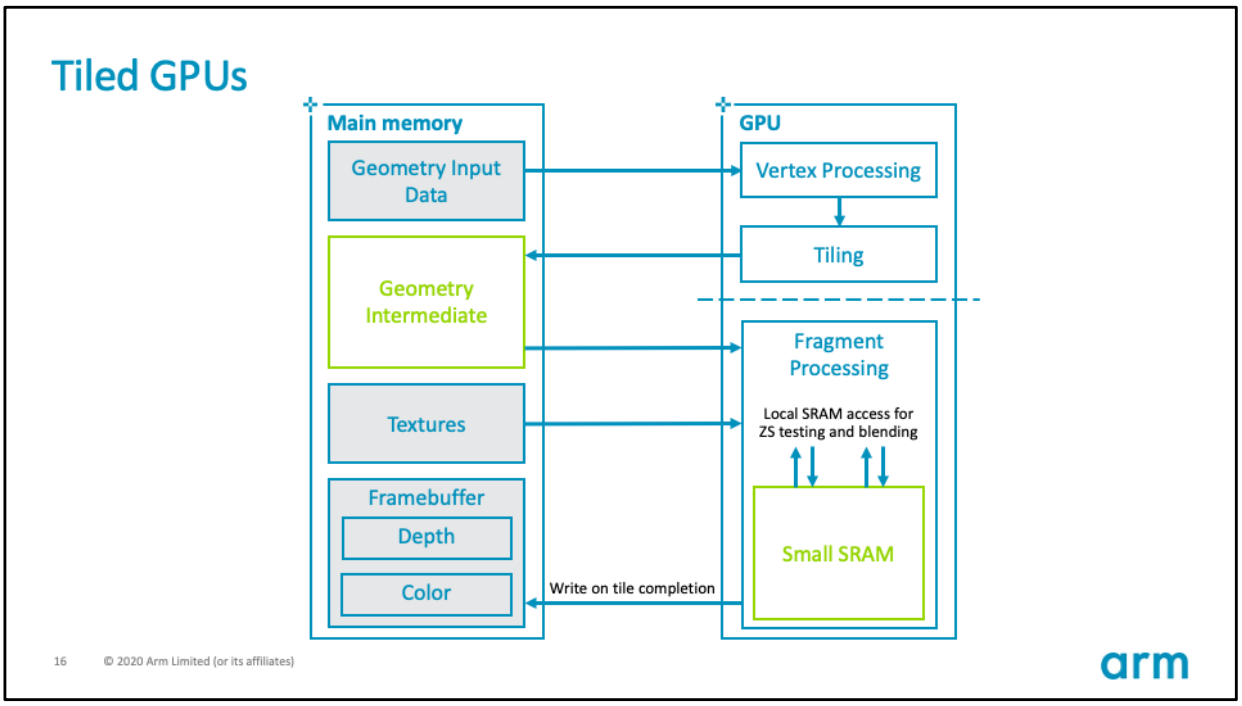


The GPU needs to load and store data to main memory, and here we represent these operations with arrows.

For immediate mode GPUs, found in desktop and console architectures, all the geometry is first processed and added to a queue.

Fragment processing then proceeds from this queue one draw call at a time, and for every pixel it will perform depth testing and color shading, reading from and writing to main memory as much as necessary.

These operations, depicted here by double arrows, can require exceptionally high bandwidth which is energy intensive.



Let's compare that to tile-based architectures commonly found in mobile.

Here we divide the screen into regions of pixels known as tiles, and the GPU rendering is then split into two phases:

- First, all the geometry is processed and assigned to tiles.
- Then for each tile we can execute all the required fragment operations using tile local memory, storing out the result only at the very end.

This can save a lot of bandwidth.

This concept of tile local memory is very important and the Vulkan API will let us optimize our rendering to take advantage of it.

arm

Load/Store operations

Load operations

- `loadOp` operations define how to initialize memory at the start of a render pass

- Clear or invalidate each attachment at the start of a render pass using LOAD_OP_CLEAR or LOAD_OP_DONT_CARE on mobile
- Do not clear an attachment inside a render pass using `vkCmdClearAttachments()`

```
VkAttachmentDescription color_attachment = {};
color_attachment.format  = VK_FORMAT_B8G8R8A8_SRGB;
color_attachment.samples = VK_SAMPLE_COUNT_1_BIT;

color_attachment.loadOp  = VK_ATTACHMENT_LOAD_OP_CLEAR;
```


The load operation refers to what to do with the attachment before rendering. The available options are LOAD if we need the previous contents, CLEAR if we want to start fresh, and DON'T_CARE if we can work with whatever is more optimal.

For immediate mode GPUs these might all perform similarly, but in mobile LOAD is much more expensive than CLEAR.

This is because as we saw, when working with tiles we can save bandwidth if we avoid unnecessary reads from main memory.

Note that LOAD_OP_CLEAR is different from using Vulkan's vkCmdClearAttachments, which will instruct the GPU to explicitly write out a clear value to the attachment in main memory, which is wasteful if we are going to afterwards write out a new value over it.

Therefore please don't use LOAD_OP_LOAD unless you really need to, and do not use vkCmdClearAttachments, try to use LOAD_OP_CLEAR or LOAD_OP_DONT_CARE instead

Store operations

- `storeOp` operations define what is written back to main memory at the end of a pass

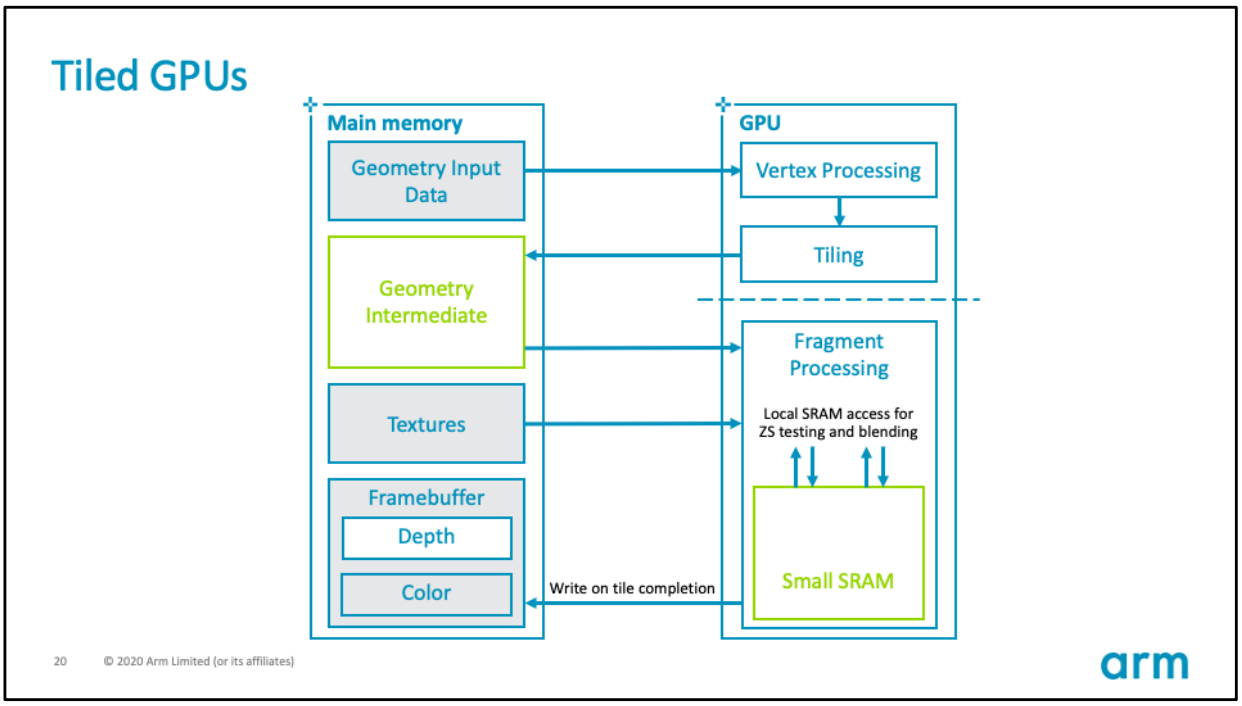
- If they are not going to be used further, ensure that the contents are invalidated at the end of the render pass using `STORE_OP_DONT_CARE` on mobile

```
VkAttachmentDescription depth_attachment = {};
depth_attachment.format  = VK_FORMAT_D32_SFLOAT;
depth_attachment.samples = VK_SAMPLE_COUNT_1_BIT;

depth_attachment.loadOp  = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
depth_attachment.storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
```


Similarly store operations define what to do with the attachment at the end of the render pass, which for color should be STORE as we need to present that color to the screen.

But what about the depth attachment, do we need it after the render pass is complete?

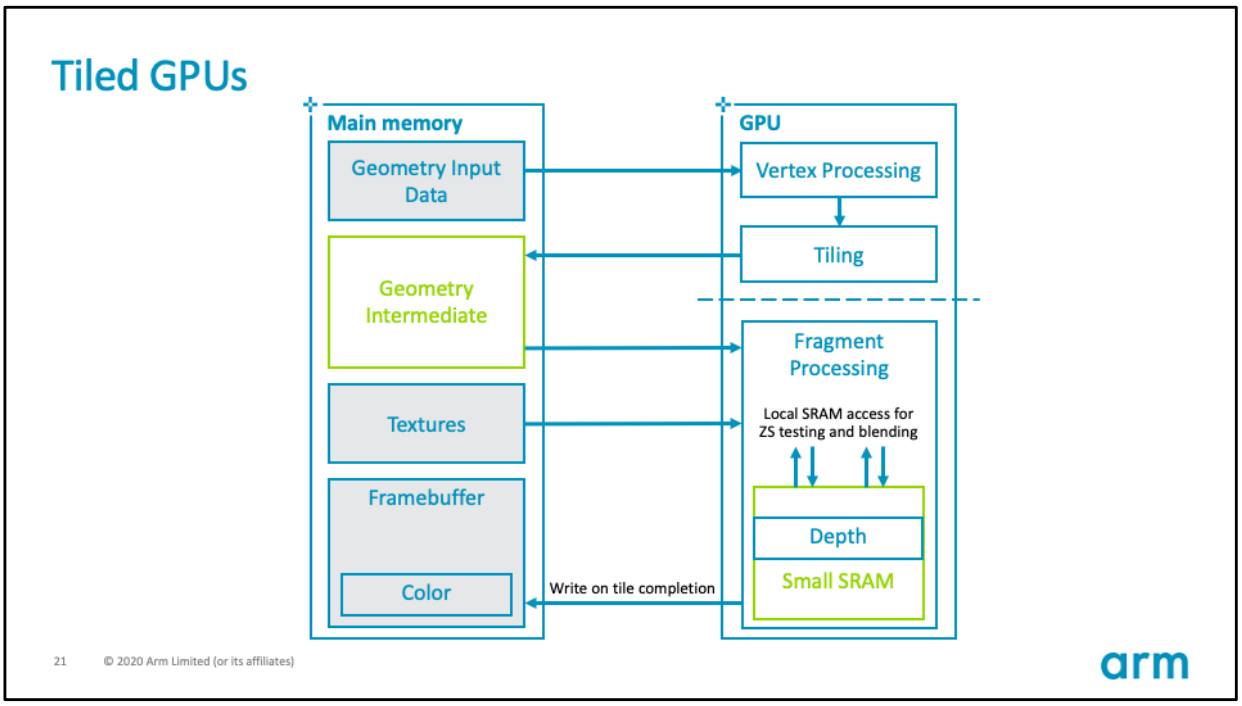
For depth the answer is almost always no, so using STORE_OP_DONT_CARE can save us a lot of bandwidth by avoiding writing it out to main memory.



If we go back to our tile-rendering diagram, you can see that we can perform depth testing using tile local memory.

Now, since we don't need to read the previous values of depth, we can use LOAD_OP_CLEAR or LOAD_OP_DONT CARE.

And, since we don't need to write out the depth values, we can use STORE_OP_DONT_CARE.



Therefore, depth does not need to be allocated at all in main memory, it can simply live temporarily in tile-local memory.

Vulkan gives us a way to request this optimization.

Transient attachments

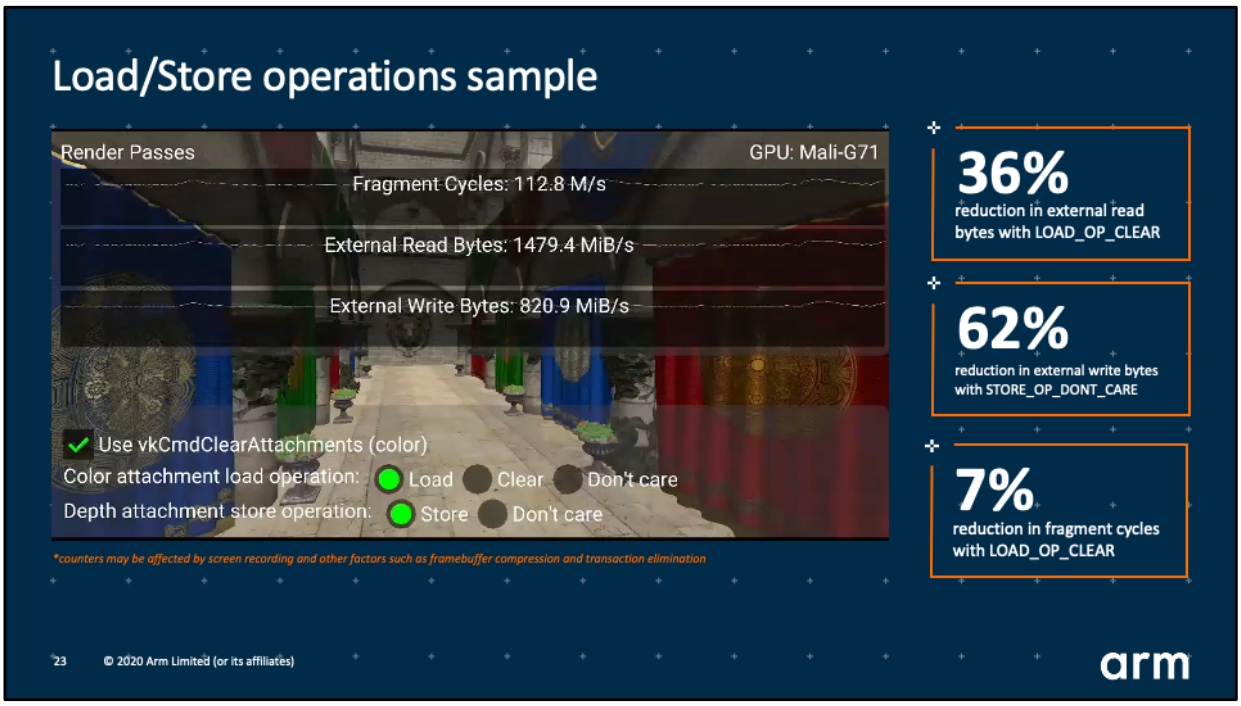
- Image usage flags: TRANSIENT_ATTACHMENT
- This tells the GPU that it can be used as a transient attachment
- It will only live for the duration of a single render-pass
- Additionally, it can be backed by LAZILY_ALLOCATED memory
- This way a tile-based renderer may avoid allocating external memory for it

```
VkImageCreateInfo image_info{VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO};
image_info.flags       = flags;
image_info.imageType   = type;
image_info.format      = format;
image_info.extent      = extent;
image_info.samples     = sample_count;
image_info.usage       = VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT;

VmaAllocation memory;
VmaAllocationCreateInfo memory_info{};
memory_info.usage = memory_usage;
memory_info.preferredFlags = VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT;

auto result = vmaCreateImage(device.get_memory_allocator(), &image_info, &memory_info, &handle, &memory, nullptr);
```


Apart from defining the right load/store operations, we also need to specify the TRANSIENT bit in the attachment's usage flags.

Then, we also need to add the LAZILY_ALLOCATED bit in the memory allocation flags.

This way a tile-based rendered may avoid allocating memory for this attachment.



We can see all this in action in the render passes sample.

https://github.com/KhronosGroup/Vulkan-Samples/blob/master/samples/performance/render_passes/render_passes_tutorial.md

(Video available in https://community.arm.com/developer/tools-software/graphics/b/blog/posts/vulkan-samples)

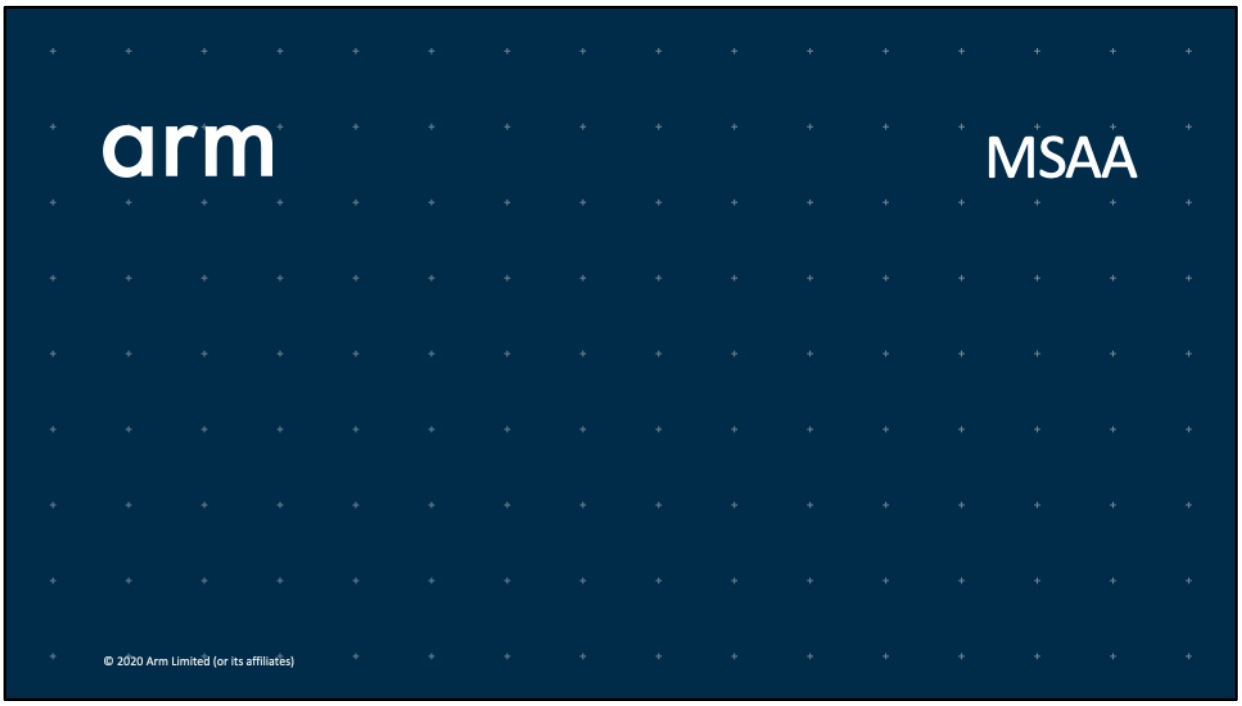
Focus on the External Read Bytes counter.

You can see how the number of reads dramatically decreases as we toggled the color load operation from load to clear.

Similarly, look now at External Write Bytes, and how it also decreases considerably when we toggle the depth store operation from store to don't care.

Finally you may also see how we can save fragment processing cycles by avoiding the use of vkCmdClearAttachments.

This represents significant savings in bandwidth and therefore power.



The MSAA sample illustrates another use-case for transient attachments.

Multisample anti-aliasing (MSAA)

arm

As you can see in this detail from one of our scenes, GPU rendering can sometimes result in jagged lines at model edges.

Multisample anti-aliasing (MSAA)

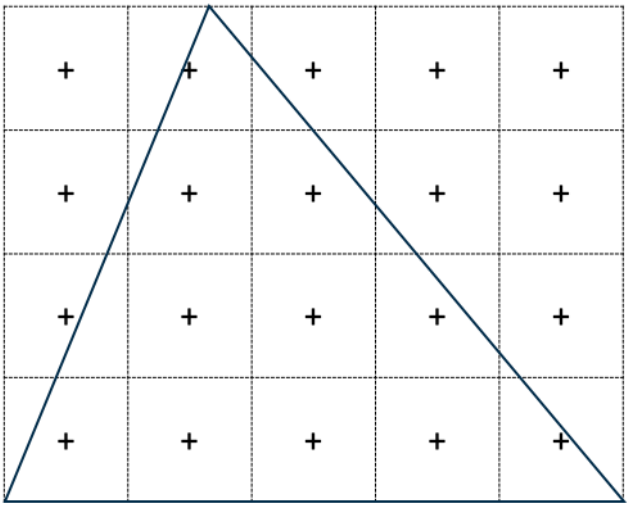
arm

We can smooth these out with MSAA.

Here you have the side-by-side comparison.

How does MSAA work?

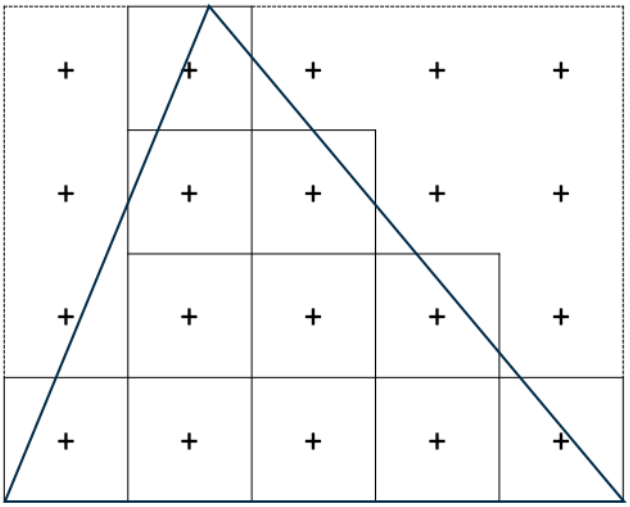
No MSAA



arm

With no MSAA, when we rasterize a polygon into pixels, we only look at the centre of the pixel.

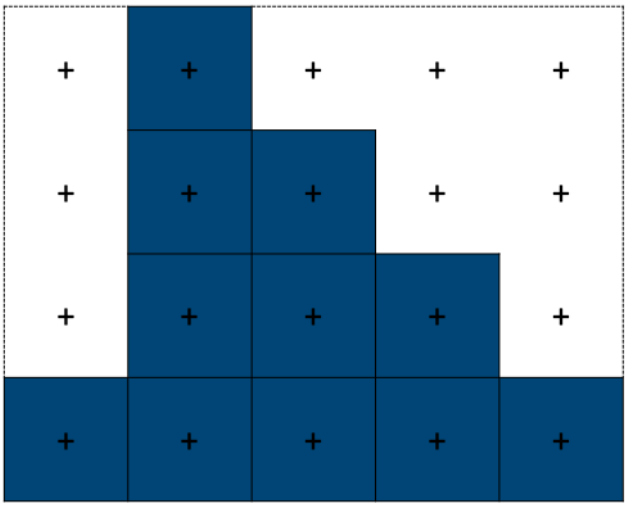
No MSAA



arm

And shade it only if the centre of the pixel lies within the polygon.

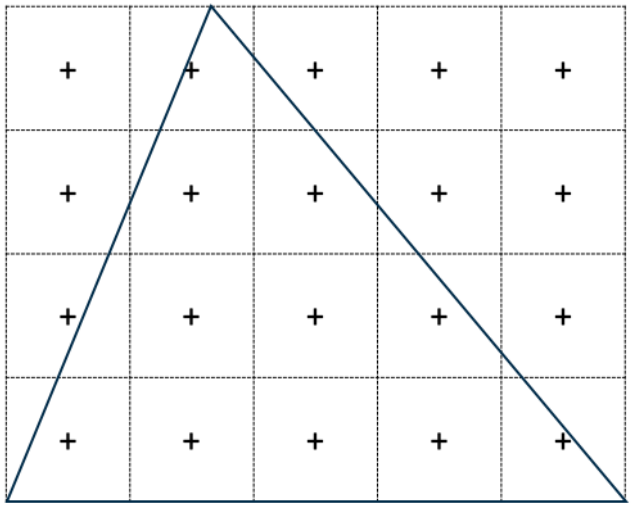
No MSAA



arm

In this case we save the result of the fragment shader in these pixels, and this staircase effect is what we then see as jagged edges.

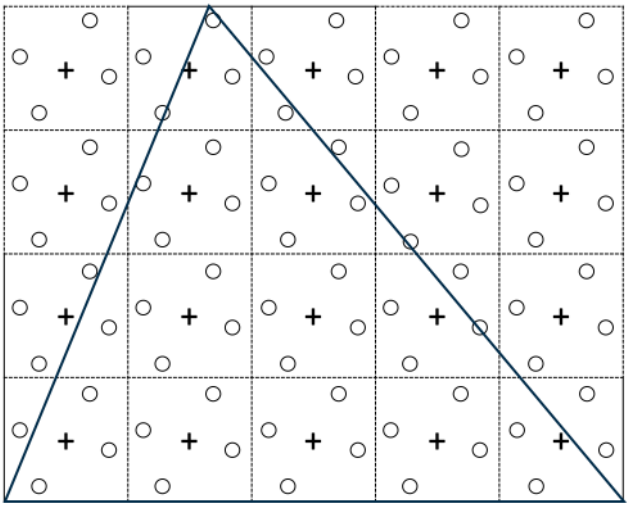
MSAA



arm

We can do better with MSAA, and instead of only checking a single point in a pixel, we look at multiple samples.

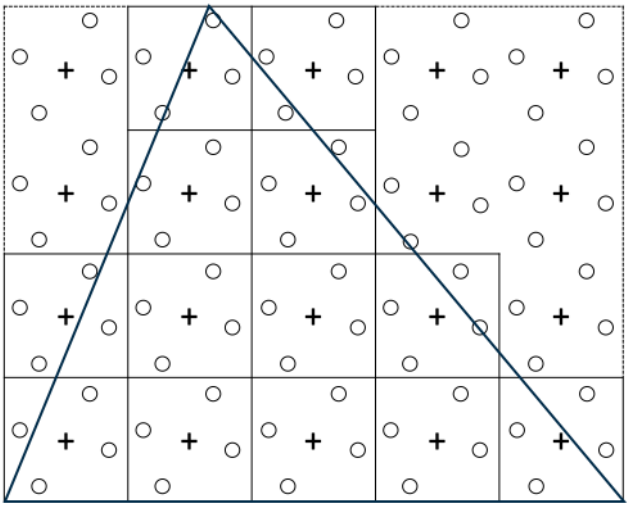
MSAA



arm

In this case we define the location of 4 samples and shade a pixel if any of these four samples lies within the polygon.

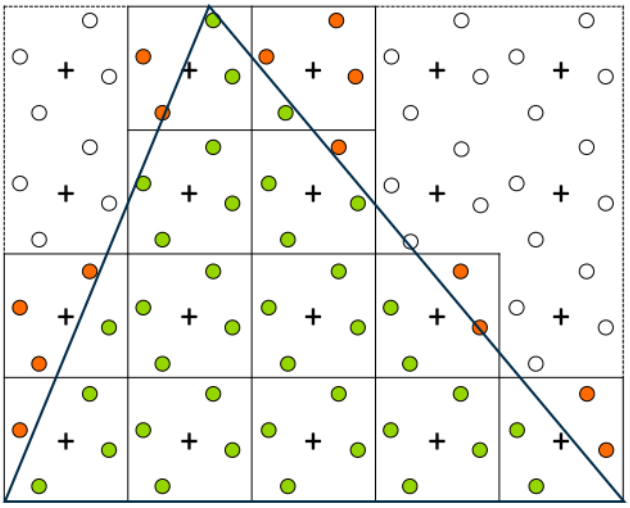
MSAA



arm

So now we are considering a couple more pixels around the edges.

MSAA



arm

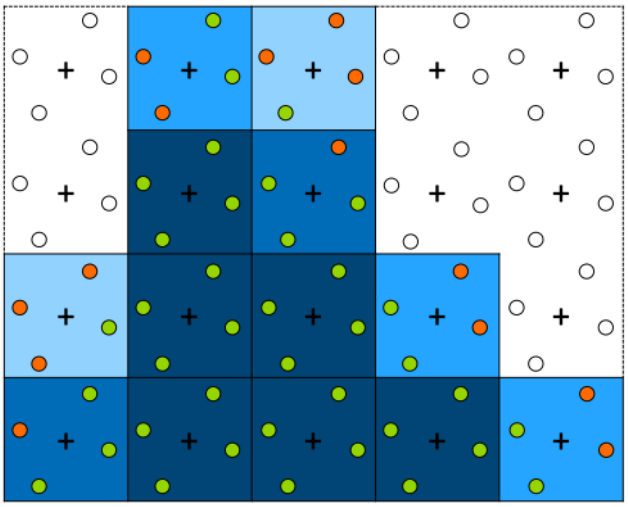
We still run the fragment shader only once, using the centre pixel coordinates.

We save this value only in the samples that lie within.

Then we average the value of all the samples.

For example if 2 out of 4 samples lie within the polygon, we'll apply 50% of the color.

MSAA



arm

Resulting in a smoother transition between the fragment color and the background at the edges.

The step where we average the sample is known as the colour resolve operation.

Resolve attachments

- Multisampled image may be transient
  - `loadOp = LOAD_OP_CLEAR`
  - `storeOp = STORE_OP_DONT_CARE`
  - Use `LAZILY_ALLOCATED` memory
- Use `pResolveAttachments` in a subpass to automatically resolve a multisampled color buffer into a single-sampled color buffer
- Avoid `vkCmdResolveImage()`: this has a significant negative impact on bandwidth and performance
- With `VK_KHR_depth_stencil_resolve` (core in Vulkan 1.2) the depth attachment may also be resolved in a similar fashion

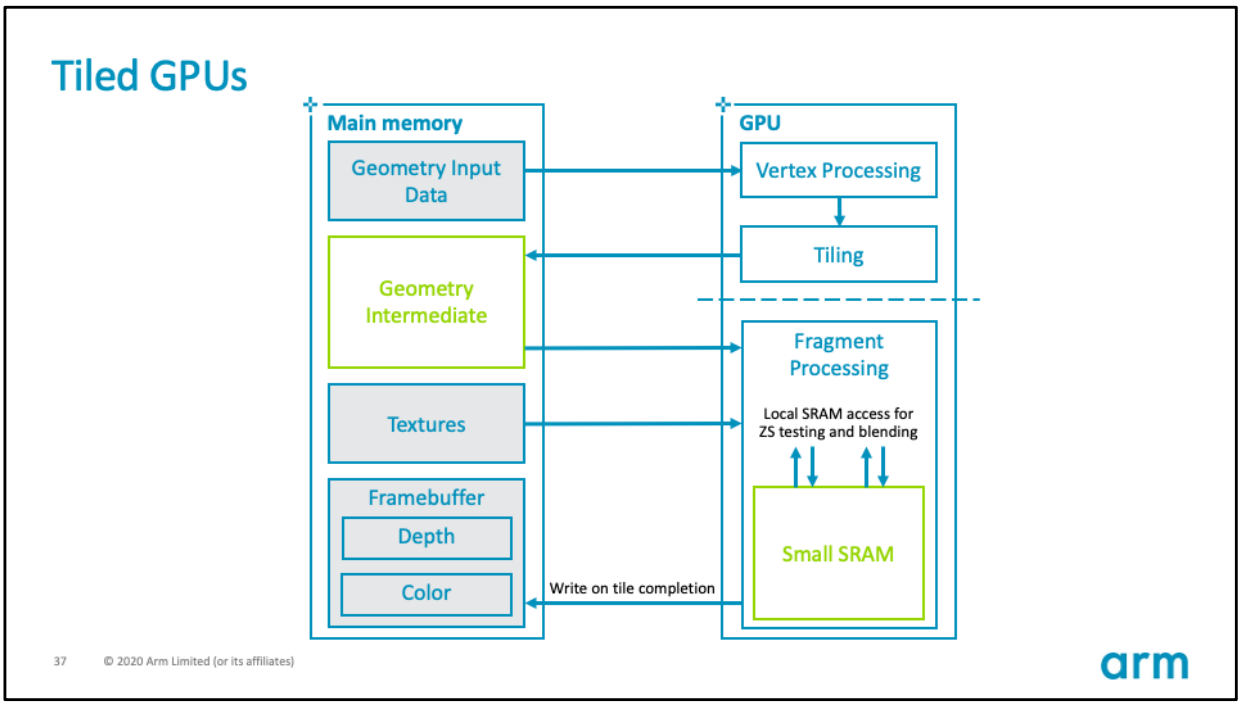
arm

If only the resolved, single-sampled attachment is required after rendering, the multi-sampled attachments may also be transient.

To do this in Vulkan as you remember we need to specify the right load/store operations as well as memory flags.

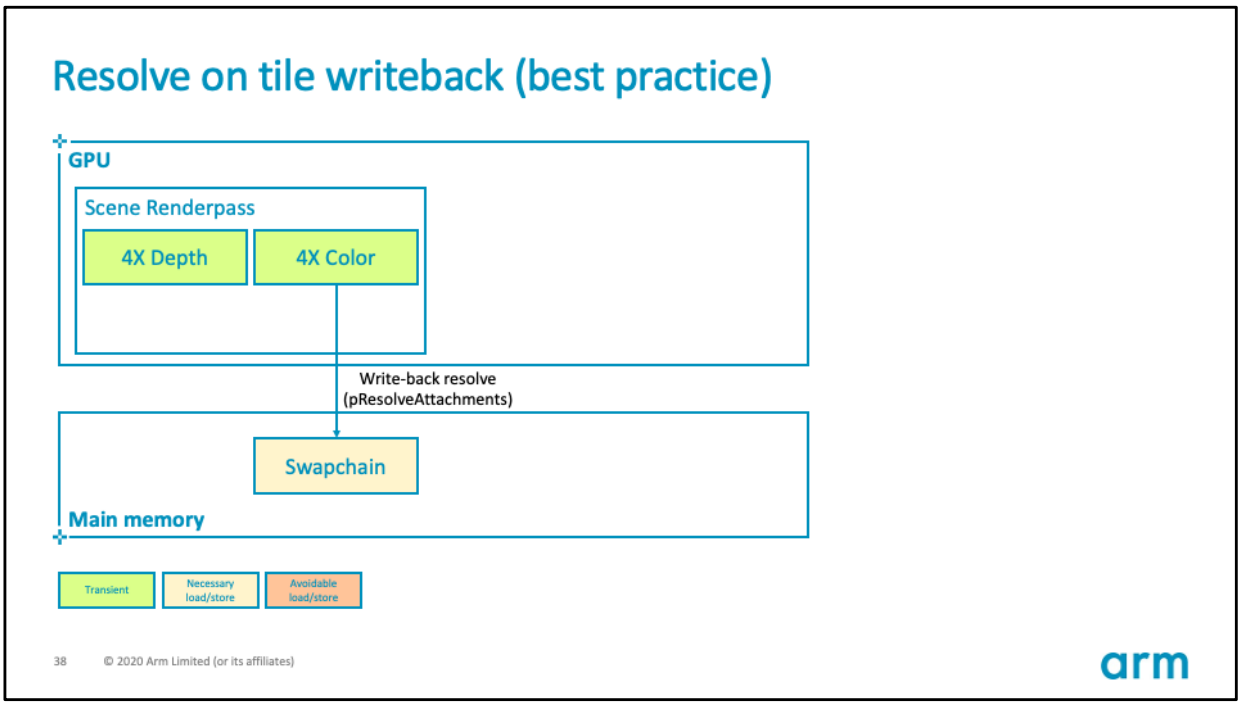
We also need to specify which attachment will be used as the destination of the resolve operation when creating the Renderpass.

Avoid using vkCmdResolveImage.



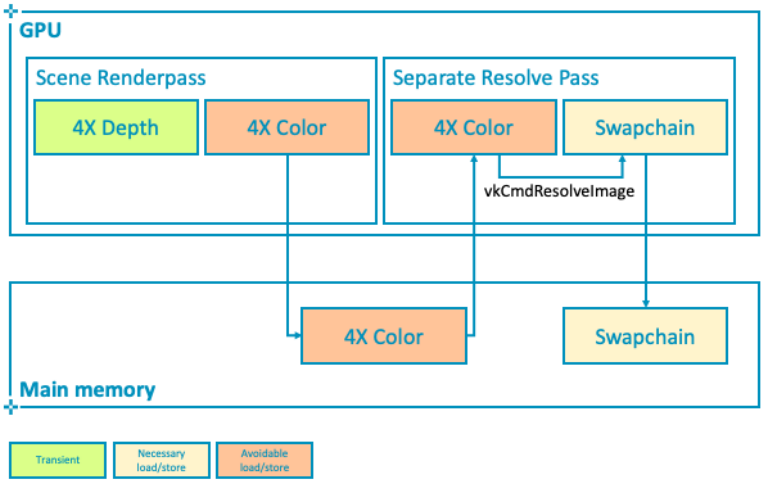
If we carry out this resolve operation in tile local memory, we do not need to write out the value of all the samples to main memory, and can therefore save bandwidth.

As we saw earlier with the depth attachment, multi-sample attachments can also be transient.



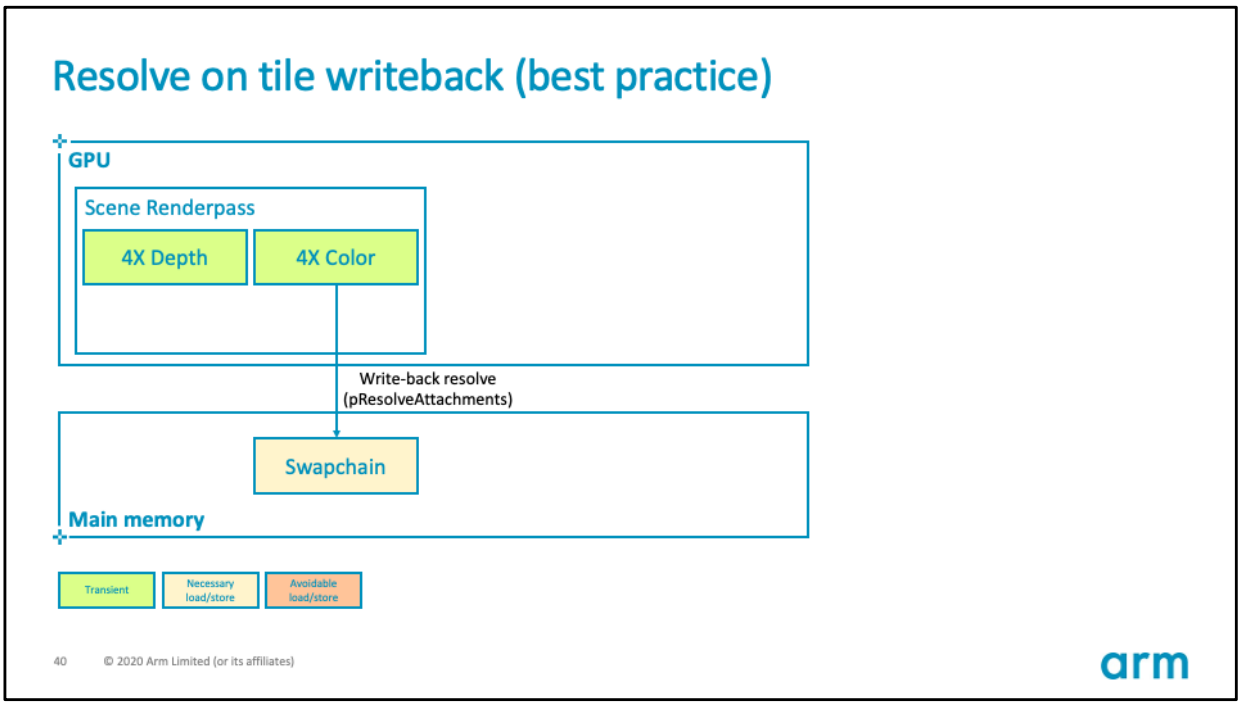
We can see here how we save bandwidth and avoid writing out the value of all samples to main memory, we will average them and write out only the result.

Resolve in a separate pass (avoid!)

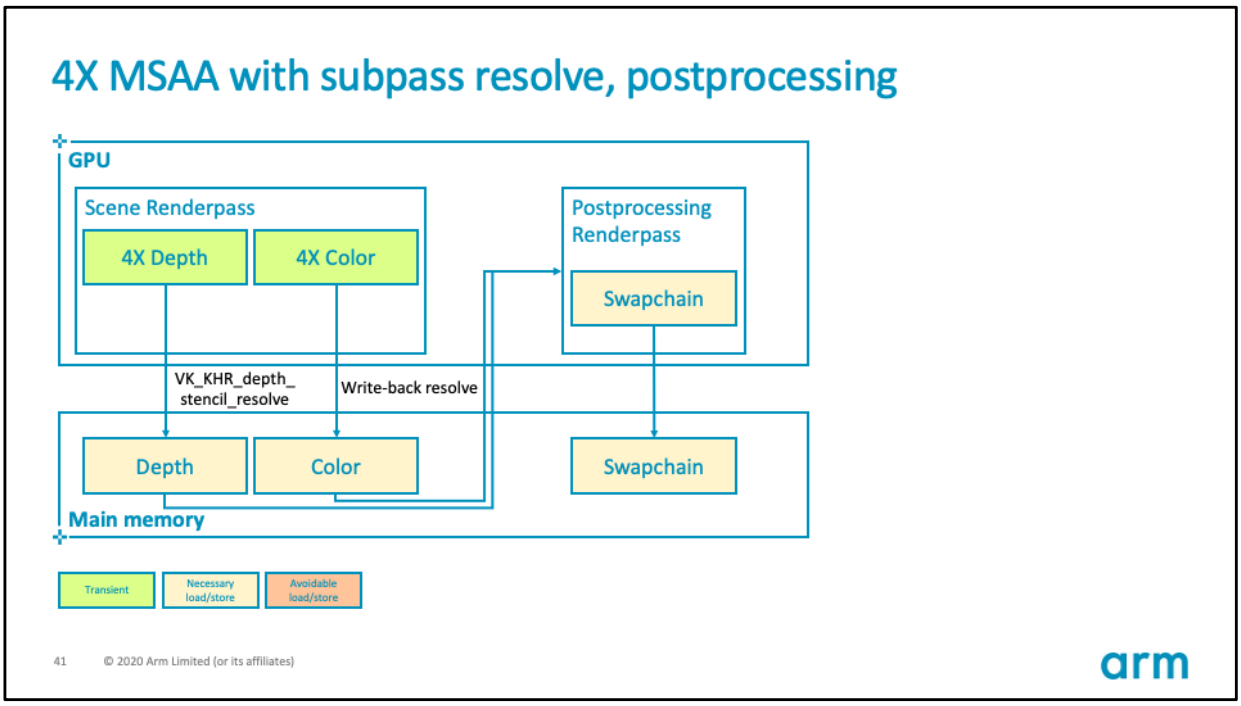


arm

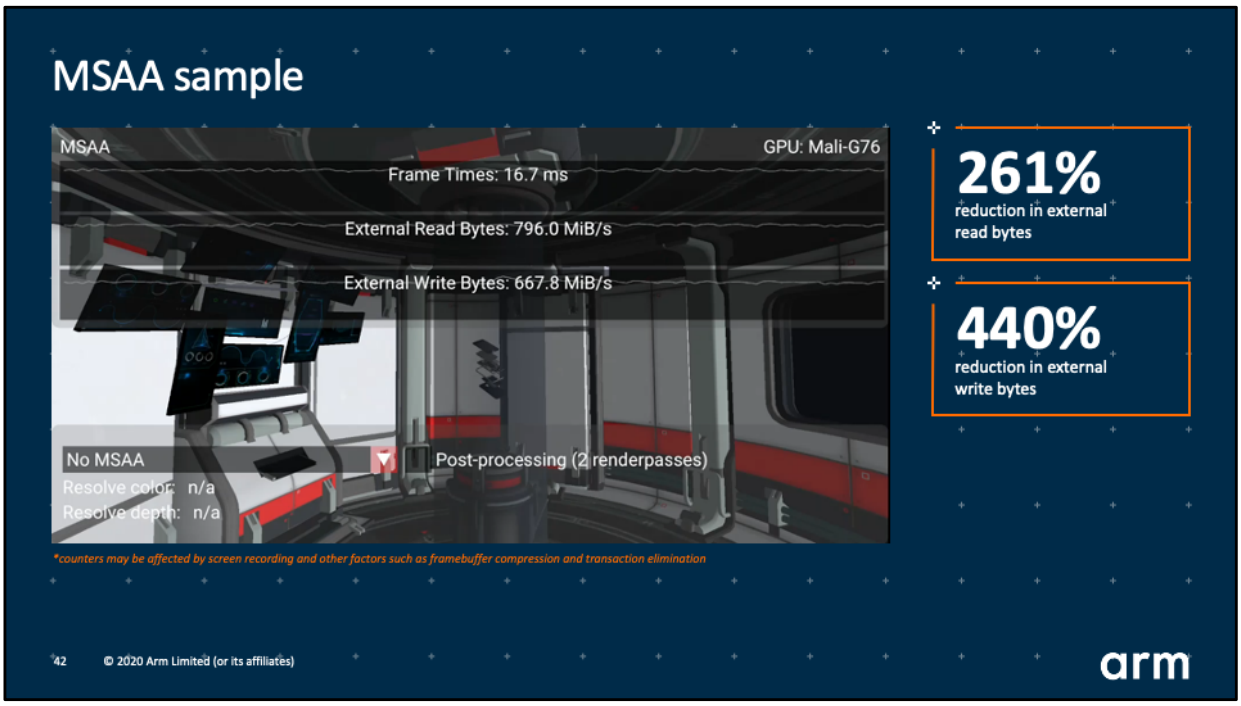
In contrast, if you use vkCmdResolveImage, the GPU will need to store the large multisampled attachment to main memory, and then read it back in a separate pass to resolve it, which is unnecessarily making use of a great deal of bandwidth and power.



Our sample also covers an use-case for resolving depth and using it in a second render pass.



For instance, a post-processing pass. To resolve depth we need to use VK_KHR_depth_stencil_resolve as shown in the sample.

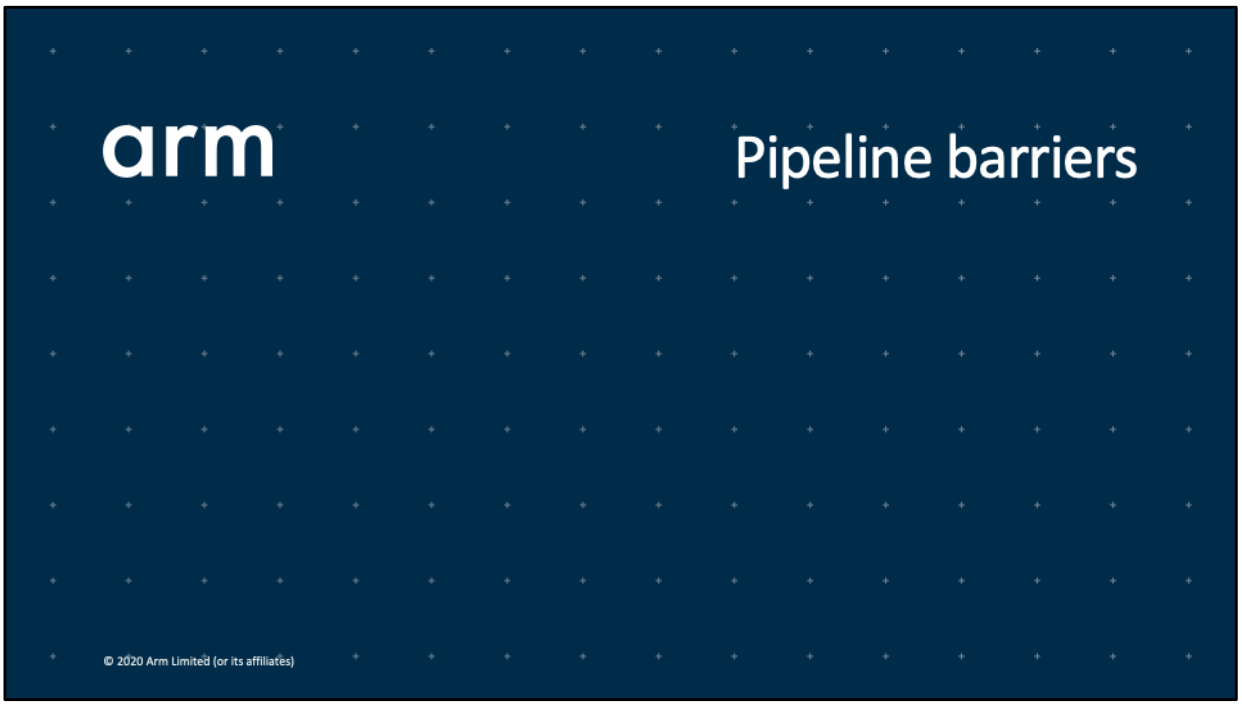


Our sample shows the bandwidth benefits with the help of the read and write counters.

https://github.com/KhronosGroup/Vulkan-Samples/blob/master/samples/performance/msaa/msaa_tutorial.md

(Video available in https://community.arm.com/developer/tools-software/graphics/b/blog/posts/vulkan-samples)

MSAA with 4 samples has little impact on bandwidth if we resolve on writeback. But as soon as we switch to vkCmdResolveImage, we introduce a separate pass and the bandwidth dramatically increases.



Something very important that we need to consider at this stage is how to define the existing dependency between the two render passes.

We cannot compute the post-processed color until we have the resolved attachments available.

This brings us to the complex topic of synchronisation. In this case we would use something known as pipeline barriers.

Pipeline stages

```
typedef enum VkPipelineStageFlagBits {
        VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT = 0x00000001,
        VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT = 0x00000002,
        VK_PIPELINE_STAGE_VERTEX_INPUT_BIT = 0x00000004,
        VK_PIPELINE_STAGE_VERTEX_SHADER_BIT = 0x00000008,
        VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT = 0x00000010,
        VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT = 0x00000020,
        VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT = 0x00000040,
        VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT = 0x00000080,
        VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT = 0x00000100,
        VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT = 0x00000200,
        VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT = 0x00000400,
        VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT = 0x00000800,
        VK_PIPELINE_STAGE_TRANSFER_BIT = 0x00001000,
        VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT = 0x00002000,
        VK_PIPELINE_STAGE_HOST_BIT = 0x00004000,
        VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT = 0x00008000,
        VK_PIPELINE_STAGE_ALL_COMMANDS_BIT = 0x00010000,
        VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT = 0x01000000,
        VK_PIPELINE_STAGE_CONDITIONAL_RENDERING_BIT_EXT = 0x00040000,
        VK_PIPELINE_STAGE_COMMAND_PROCESS_BIT_NVX = 0x00020000,
        VK_PIPELINE_STAGE_SHADING_RATE_IMAGE_BIT_NV = 0x00400000,
        VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_NV = 0x00200000,
        VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_NV = 0x02000000,
        VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV = 0x00080000,
        VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV = 0x00100000,
        VK_PIPELINE_STAGE_FRAGMENT_DENSITY_PROCESS_BIT_EXT = 0x00800000,
        VK_PIPELINE_STAGE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkPipelineStageFlagBits;
```


arm

Pipeline barriers work with pipeline stages.

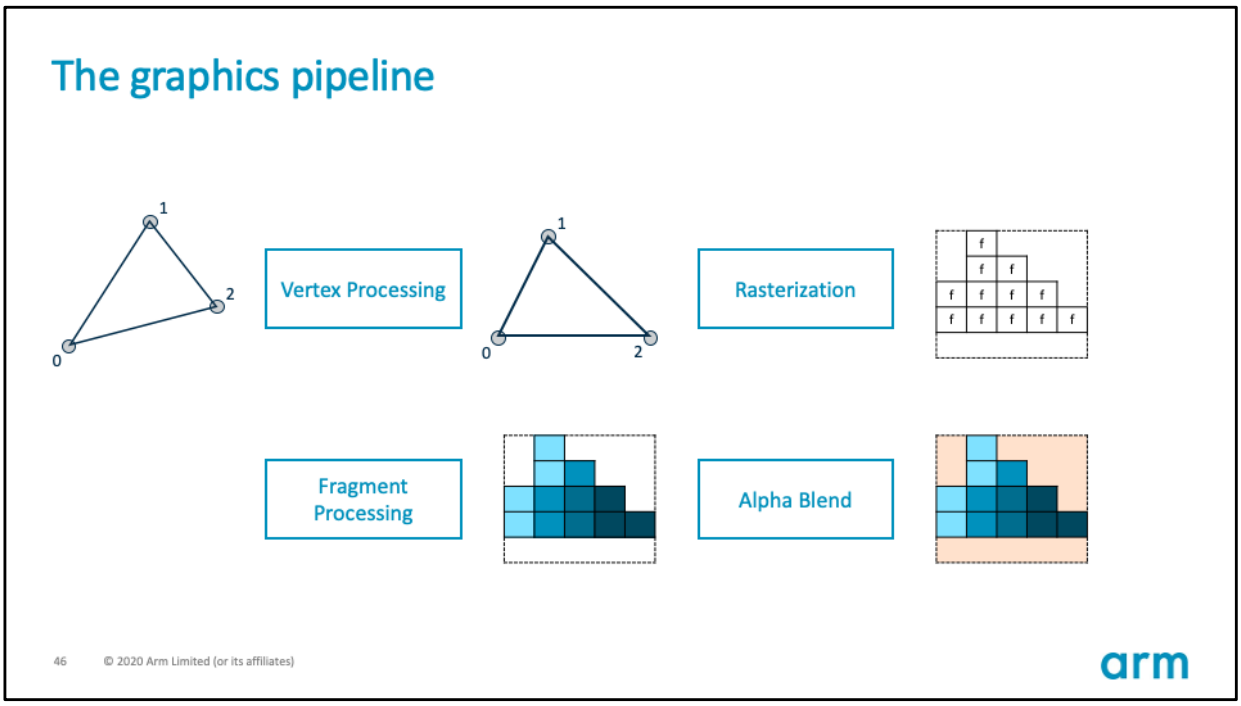
Here we have a Vulkan enumeration listing all the possible stages of the graphics pipeline.

Pipeline stages

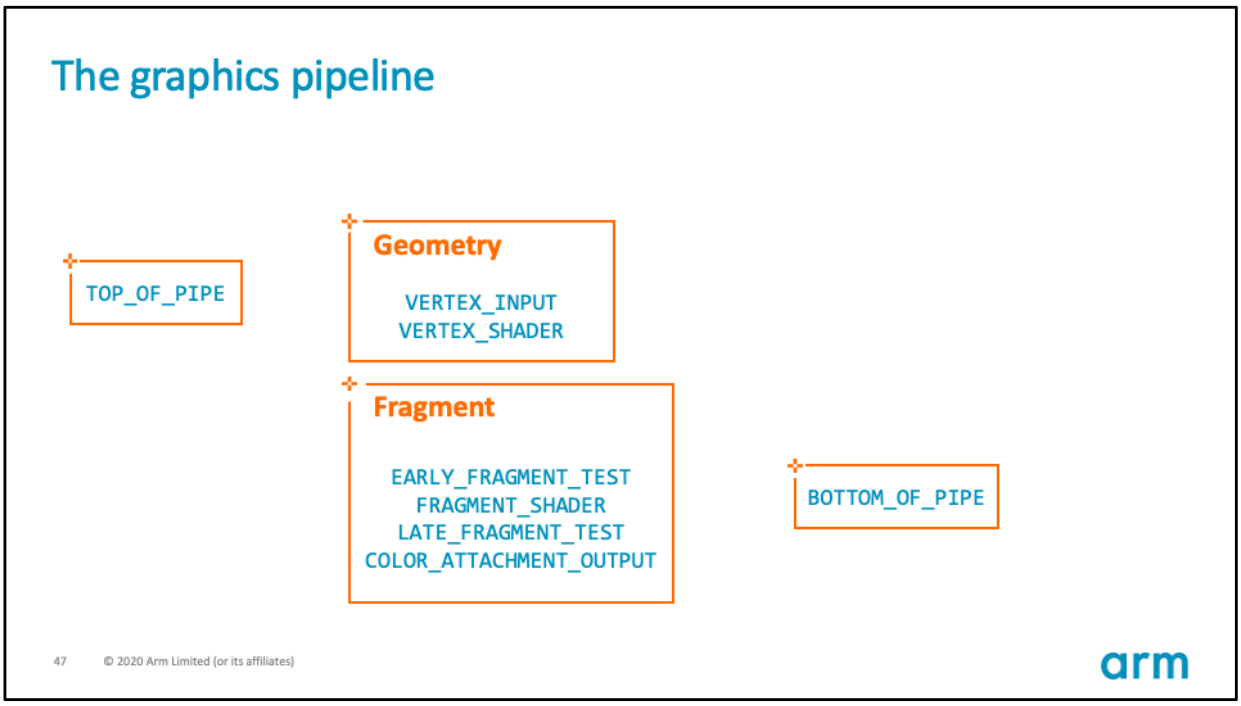
```
typedef enum VkPipelineStageFlagBits {
        VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT = 0x00000001,
        VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT = 0x00000002,
        VK_PIPELINE_STAGE_VERTEX_INPUT_BIT = 0x00000004,
        VK_PIPELINE_STAGE_VERTEX_SHADER_BIT = 0x00000008,
        VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT = 0x00000010,
        VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT = 0x00000020,
        VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT = 0x00000040,
        VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT = 0x00000080,
        VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT = 0x00000100,
        VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT = 0x00000200,
        VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT = 0x00000400,
        VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT = 0x00000800,
        VK_PIPELINE_STAGE_TRANSFER_BIT = 0x00001000,
        VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT = 0x00002000,
        VK_PIPELINE_STAGE_HOST_BIT = 0x00004000,
        VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT = 0x00008000,
        VK_PIPELINE_STAGE_ALL_COMMANDS_BIT = 0x00010000,
        VK_PIPELINE_STAGE_TRANSFORM_FEEDBACK_BIT_EXT = 0x01000000,
        VK_PIPELINE_STAGE_CONDITIONAL_RENDERING_BIT_EXT = 0x00040000,
        VK_PIPELINE_STAGE_COMMAND_PROCESS_BIT_NVX = 0x00020000,
        VK_PIPELINE_STAGE_SHADING_RATE_IMAGE_BIT_NV = 0x00400000,
        VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_NV = 0x00200000,
        VK_PIPELINE_STAGE_ACCELERATION_STRUCTURE_BUILD_BIT_NV = 0x02000000,
        VK_PIPELINE_STAGE_TASK_SHADER_BIT_NV = 0x00080000,
        VK_PIPELINE_STAGE_MESH_SHADER_BIT_NV = 0x00100000,
        VK_PIPELINE_STAGE_FRAGMENT_DENSITY_PROCESS_BIT_EXT = 0x00800000,
        VK_PIPELINE_STAGE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkPipelineStageFlagBits;
```


arm

For this presentation we will consider a subset of these, the ones I highlighted.



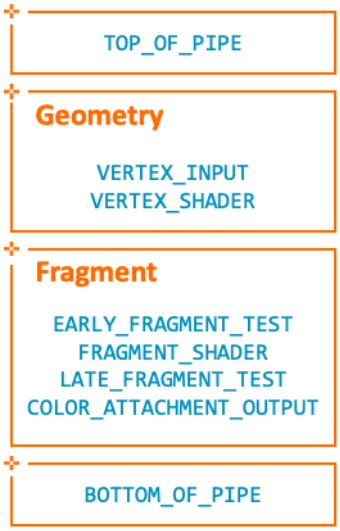
Roughly, this is how they map to the broader graphics pipeline that we saw earlier.



Commands will first go through the TOP_OF_PIPE stage.
Then we have the geometry stages, rasterization, and fragment stages, ending with BOTTOM_OF_PIPE.

The TOP_ and BOTTOM_OF_PIPE stages are helper stages. TOP signals that a command has been parsed, and BOTTOM that the command is retired.

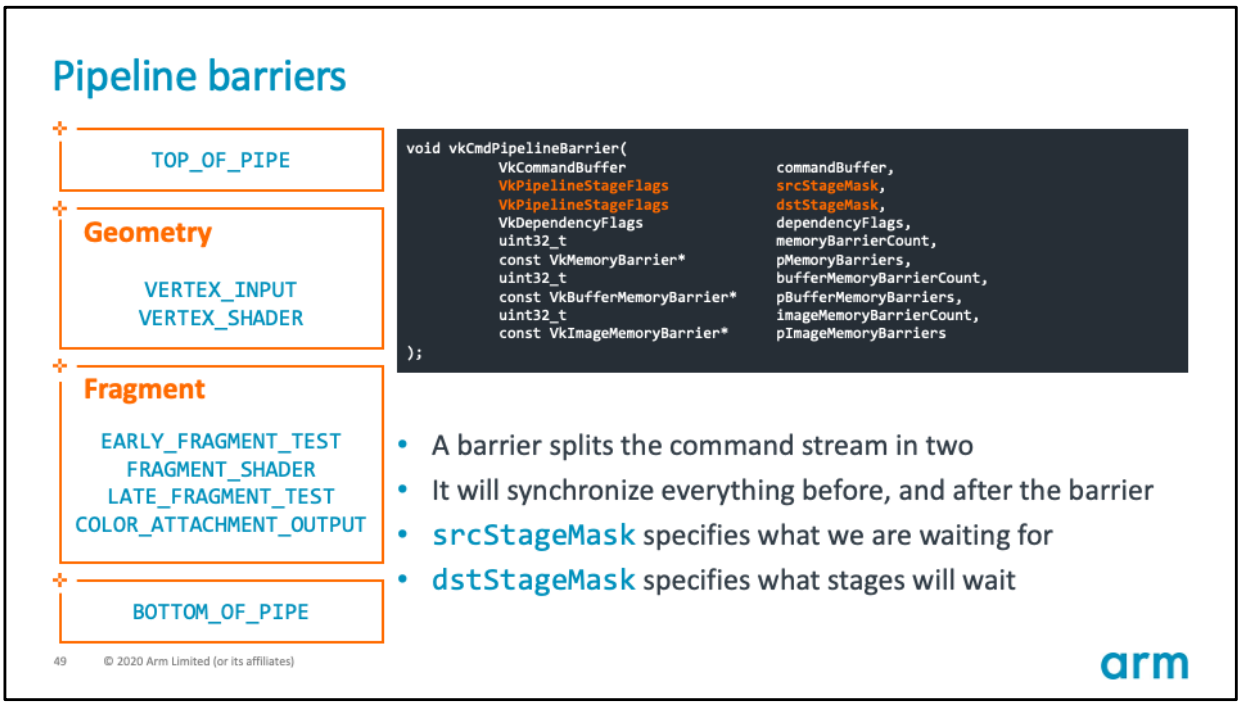
The graphics pipeline



In Vulkan, every command that we submit to a queue goes through some of these stages.

Commands may execute out of order across command buffers and even queue submissions.

In Vulkan we do not synchronize individual commands, instead we synchronize the work using these stages.



Vulkan offers the mechanism of execution barriers.

With a barrier we are dividing the command stream in two parts, and we are saying that all commands after the barrier must wait at a certain destination stage until all commands before the barrier have gone through a certain source stage.

Let’s illustrate it with an example.

Pipelining: avoid BOTTOM->TOP dependencies


```
vkCmdPipelineBarrier(
          command_buffer,
          VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT,
          VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT,
          0,
          0, nullptr,
          0, nullptr,
          1, &image_memory_barrier)
```


arm

In GPUs like Mali we find two processing slots to do vertex and fragment work in parallel.

Here we see the timeline as the CPU dispatches commands, and some go to vertex, others to fragment.

Let's say that each color here represents the work of a Renderpass, and we add one of these barriers between Renderpasses.

It is very important to synchronize the work to allow for maximum throughput.

In this case the destination stage is TOP_OF_PIPE, the very first stage of the pipeline. All commands after the barrier must wait here, which means they cannot start going through the pipeline at all, until all commands before the barrier have reached the source stage which is BOTTOM_OF_PIPE, the last stage of the pipeline.

This means that nothing can execute in parallel, we introduce bubbles and the work is heavily serialized.

This sort of barrier is inefficient.

Pipelining: avoid BOTTOM->TOP dependencies


```
vkCmdPipelineBarrier(
          command_buffer,
          VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT,
          VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,
          0,
          0, nullptr,
          0, nullptr,
          1, &image_memory_barrier)
```


Compare that to this other barrier.

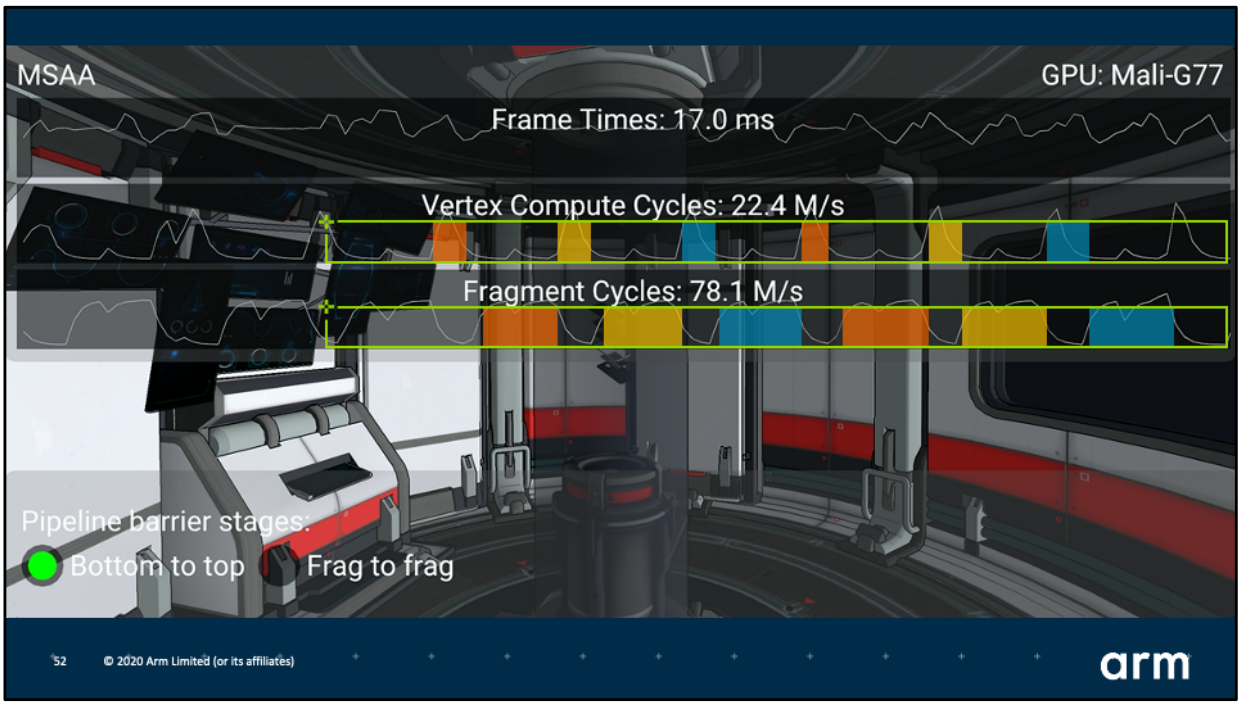
Here we are saying that commands after the barrier must wait at the fragment stage until commands before the barrier have also gone through the fragment stage.

This means that the vertex work from one Renderpass can proceed in parallel to the fragment work from a previous Renderpass.

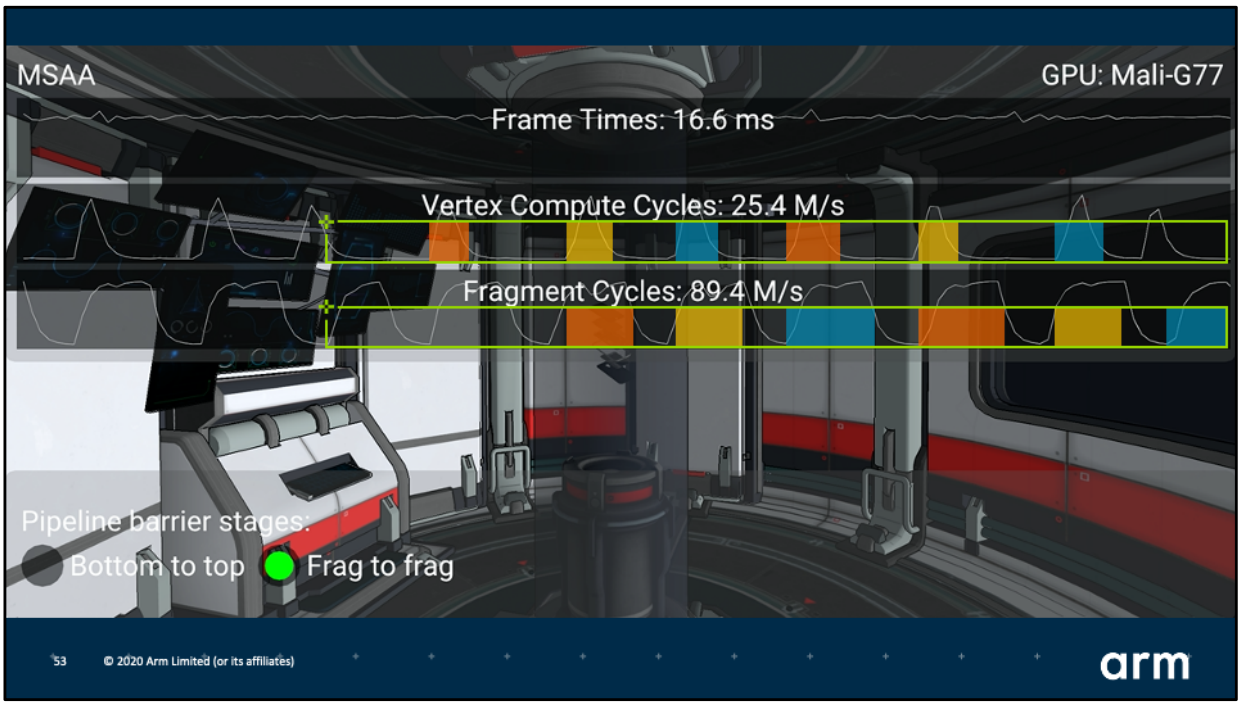
Therefore we avoid bubbles and maximize throughput.

Of course we are not saying that you should always use this barrier but try to avoid BOTTOM->TOP and find the one that best applies to your use case.

In this case since the vertex is not dependent on the fragment, this is the minimal correct barrier which covers the use case and hence avoids over-synchronizing.



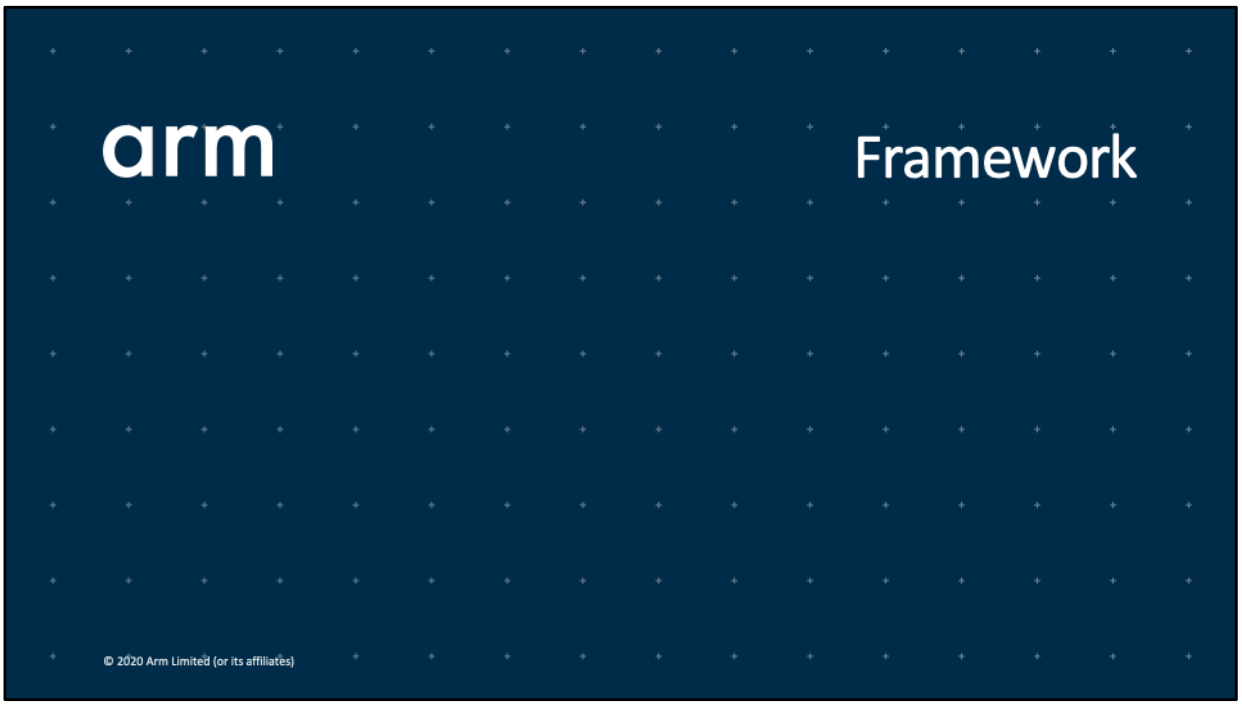
In the case of the MSAA sample this is how these bubbles can be visualized using the vertex and fragment cycle counters.

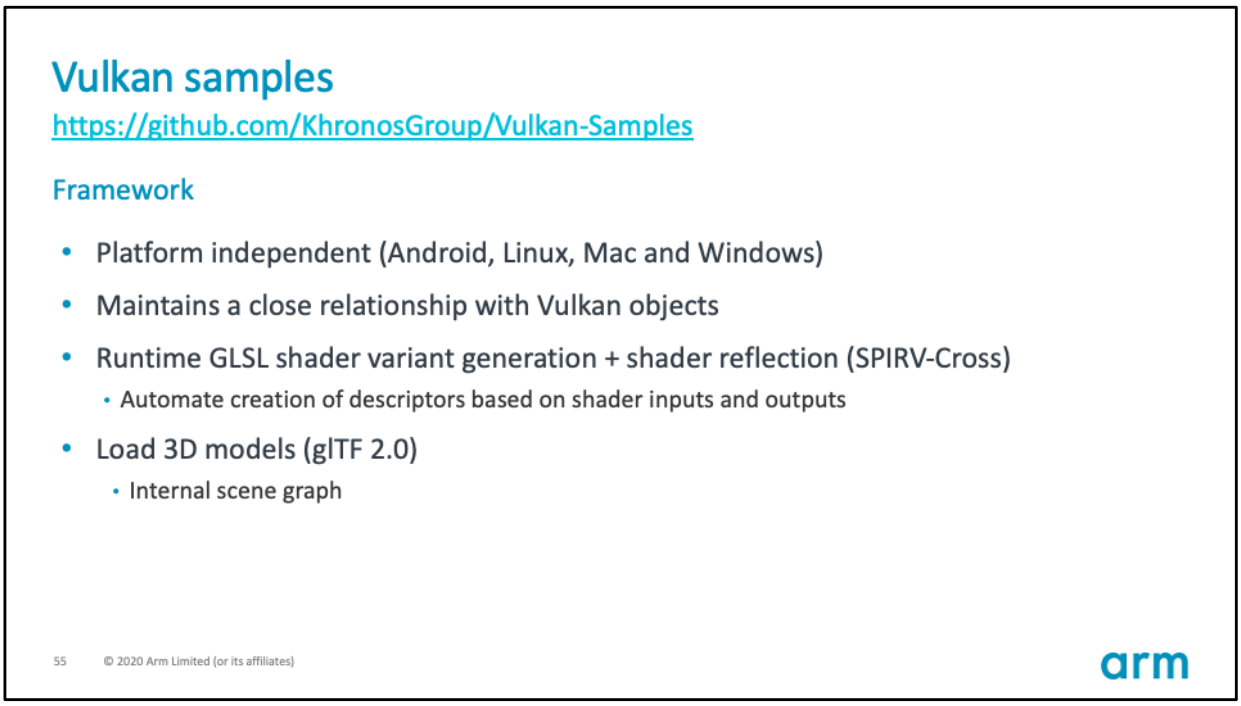


In the case of a fragment to fragment barrier, most of the vertex peaks occur in parallel to the fragment plateau, which is what we want.

This is a very simple scene, but in real games we have seen how a 1-line change to a pipeline barrier has improved FPS by up to 56%.

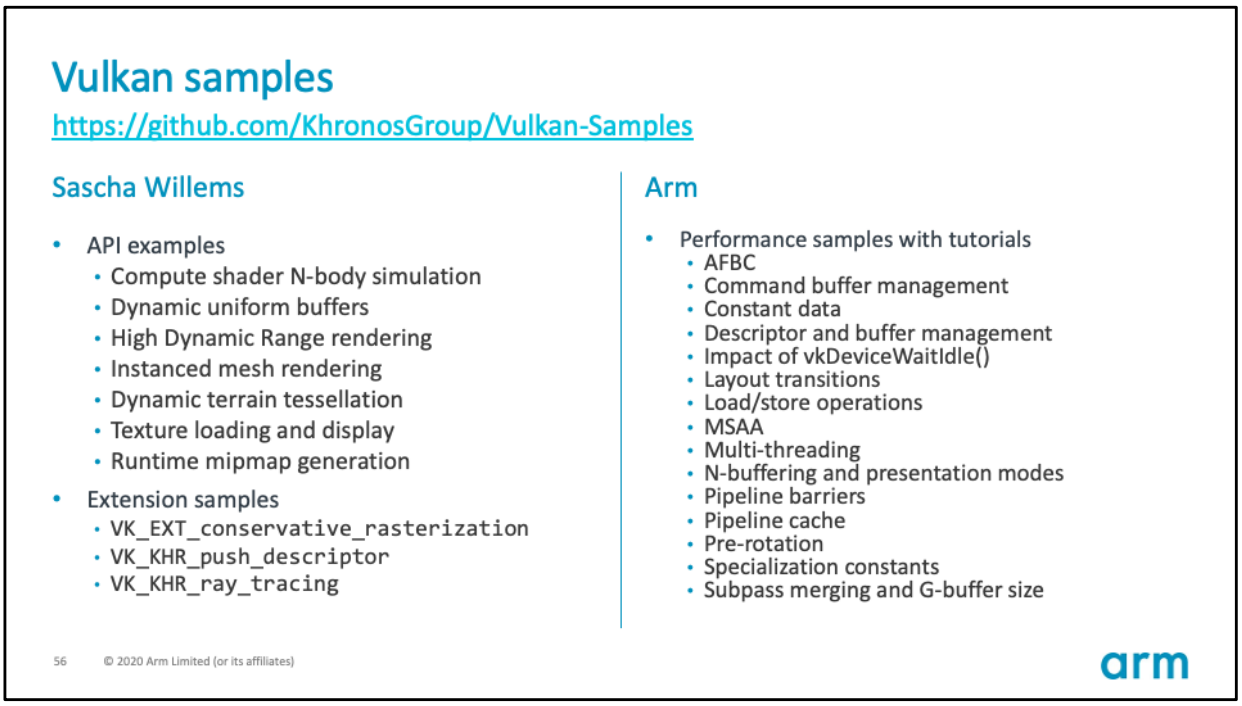
https://github.com/KhronosGroup/Vulkan-Samples/blob/master/samples/performance/pipeline_barriers/pipeline_barriers_tutorial.md


arm
Framework
© 2020 Arm Limited (or its affiliates)



Here are some of the main features to highlight about the framework underlying all the samples:

- It is multi-platform, enabling the samples to run on both desktop and mobile.
- It provides an API to encapsulate some of the objects and make them easier to handle.
- It implements shader reflection to automatically generate some of the necessary objects based on the shaders.
- Finally as I mentioned we integrated a glTF importer so you can use your own scenes.



These are all the samples that are available today.

We do get many requests and ideas for new samples and will prioritize which samples to work on based the community feedback.

Therefore we encourage you to check them out when you have the chance and let us know how to help you understand Vulkan and integrate these best practices in your applications.

Vulkan samples

https://github.com/KhronosGroup/Vulkan-Samples

Current and future work

- New samples
- Modularize framework into helper libraries
- Use framework for investigation of advanced rendering techniques
- Integrate, test and contribute to the Khronos validation layer best practice object

arm

Further reading

- Introduction
  - *Vulkan Guide*, Khronos Group
  - *GPU Framebuffer Memory: Understanding Tiling*, Samsung GameDev
  - *How does Vulkan compare to OpenGL ES?* by Pete Harris
- Best practice resources
  - *Arm Mali GPU Best Practices Developer Guide*
  - *Vulkan Samples Tutorials*
  - *Arm Mobile Studio*
  - *Writing an efficient Vulkan renderer*, GPU Zen 2, by Arseny Kapoulkine
- Vulkan synchronization
  - *Synchronization Examples* by Tobias Hector
  - *Yet another blog explaining Vulkan synchronization* by Hans-Kristian Arntzen
  - *Some Seriously Explicit Lessons in Vulkan*, Samsung GameDev and Croteam

arm

Introduction

Vulkan Guide, Khronos Group
GPU Framebuffer Memory: Understanding Tiling, Samsung GameDev
How does Vulkan compare to OpenGL ES? by Pete Harris

Best practice resources

Arm Mali GPU Best Practices Developer Guide
Vulkan Samples Tutorials
Arm Mobile Studio
Writing an efficient Vulkan renderer, GPU Zen 2, by Arseny Kapoulkine

Vulkan synchronization

Synchronization Examples by Tobias Hector
Yet another blog explaining Vulkan synchronization by Hans-Kristian Arntzen
Some Seriously Explicit Lessons in Vulkan, Samsung GameDev and Croteam

arm

Thank You
Danke
Merci
谢谢
ありがとう
Gracias
Kiitos
감사합니다
धन्यवाद
شكرًا
ধন্যবাদ
תודה

arm