ARM® Mali™ Application Developer Best Practices

Version 1.1

Developer Guide

ARM® Mali™ Application Developer Best Practices

Developer Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
0100-00	27 February 2017	Non-Confidential	First release for version 1
0100-01	06 March 2017	Non-Confidential	Second release for version 1
0100-02	17 March 2017	Non-Confidential	Third release for version 1
0101-00	26 June 2017	Non-Confidential	First release for version 1.1

Contents

ARM® Mali™ Application Developer Best Practices Developer Guide

Preface

This preface introduces the *ARM® Mali™ Application Developer Best Practices Developer Guide*.

It contains the following:

- *About this book* on page 6.
- *Feedback* on page 8.

About this book

This book is for the ARM® Mali™ Application Developer Best Practices for Mali GPUs.

Product revision status

The r*m*p*n* identifier indicates the revision status of the product described in this book, for example, r1p2, where:

r*m* Identifies the major revision of the product, for example, r1.

p*n* Identifies the minor revision or modification status of the product, for example, p2.

Intended audience

This book is for application developers who are developing for the Mali GPU.

Using this book

This book is organized into the following chapters:

Chapter 1 Application development best practices for Mali GPUs
This chapter introduces best practices for ARM Mali GPUs.

Appendix A Revisions
This appendix describes the changes between released issues of this book.

Glossary

The ARM Glossary is a list of terms used in ARM documentation, together with definitions for those terms. The ARM Glossary does not contain terms that are industry standard unless the ARM meaning differs from the generally accepted meaning.

See the *ARM Glossary* for more information.

Typographic conventions

italic
Introduces special terminology, denotes cross-references, and citations.

bold
Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`
Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`monospace`
Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`monospace italic`
Denotes arguments to monospace text where the argument is to be replaced by a specific value.

`monospace bold`
Denotes language keywords when used outside example code.

`<and>`
Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

SMALL CAPITALS
Used in body text for a few terms that have specific technical meanings, that are defined in the *ARM Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Additional reading - appdev

Information published by ARM and by third parties.

See *http://infocenter.arm.com* for access to ARM documentation.

ARM publications

This book contains information that is specific to this product. See the following documents for other relevant information:
Developer resources:

https://developer.arm.com/graphics

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to *errata@arm.com*. Give:

- The title *ARM Mali Application Developer Best Practices Developer Guide*.
- The number ARM 100971_0100_02_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Note

ARM tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Chapter 1
Application development best practices for Mali GPUs

This chapter introduces best practices for ARM Mali GPUs.

It contains the following section:

1.1 Application developer best practices for Mali GPUs

This information is for the expert developer audience, familiar with Vulkan and OpenGL ES API programming. The technical details given here are for information only. Use with care.

A graphics system can be represented as a pipeline of stages, performance problems can arise in each of these stages.

At each stage we outline topics of interest. Each topic has a detailed explanation, with actionable "dos" and "don'ts" which should be considered in application development. In addition to the dos and don'ts, we state the impact of failing to follow that topic's best practice and include debugging advice which can be used to troubleshoot each performance issue.

Each topic contains an anchor link to a justification which explains the recommendation.

Application Developer Best Practices for Mali GPUs

This information is for the expert developer audience, familiar with Vulkan and OpenGL ES API programming. The technical details given here are for information only. Use with care.

System level overview

A graphics system can be represented as a pipeline of stages, performance problems can arisn each of these stages. At each stage we outline topics of interest. Each topic has a detailed explanation, with actionable "dos" and "don'ts" which should be considered in application development. In addition to the dos and don'ts, we state the impact of failing to follow that topic's best practice and include debugging advice which can be used to troubleshoot each performance issue. Each topic contains an anchor link to a justification which explains the recommendation.

```
[#JUST1]
```

CPU - Driver mapping and unmapping memory

This section is Vulkan only.

Vulkan adds support for far more sophisticated buffer mapping implementations compared to OpenGL ES. Multiple memory types with different caching mechanisms are exposed, as well as being able to persistently map memory in to the CPU address space.

The Mali driver exposes 3 memory types on Midgard architecture GPUs:

- DEVICE_LOCAL_BIT | HOST_VISIBLE_BIT | HOST_COHERENT_BIT
- DEVICE_LOCAL_BIT | HOST_VISIBLE_BIT | HOST_CACHED_BIT
- DEVICE_LOCAL_BIT | LAZILY_ALLOCATED_BIT

On Bifrost architecture GPUs, the following memory types are exposed:

- DEVICE_LOCAL_BIT | HOST_VISIBLE_BIT | HOST_COHERENT_BIT
- DEVICE_LOCAL_BIT | HOST_VISIBLE_BIT | HOST_COHERENT_BIT | HOST_CACHED_BIT
- DEVICE_LOCAL_BIT | LAZILY_ALLOCATED_BIT

These four different types serve different purposes.

Coherent, not cached

This is the default memory type as HOST_VISIBLE | COHERENT is guaranteed to be supported. It is great for streaming out data to GPU buffers since write-combine in ARM CPUs can buffer up small bursts of write-outs which go straight to memory, so for passing data from the application to the GPU it is the most appropriate buffer type.

Cached, incoherent

While write-out from CPU is great even without host caching, readbacks in to the CPU desperately need cached memory. Throughput for cached readbacks with `memcpy()` have been observed to be 10x faster due to the ability prefetch in to the CPU cache. However, in this case since the memory is incoherent, manual cache management is required to ensure that the CPU and GPU always correctly access the current version of the data. To write to incoherent memory from CPU vkFlushMappedRanges is required, and vkInvalidateMappedRanges is required to safely read back data from GPU.

Cached and coherent

This memory type is supported by Bifrost only, but requires the chip memory system to implement full coherency so many not always be available. In this mode memory coherency as well as CPU caching are enabled, which means no manual cache management is necessary, and the use of cache means that CPU readback performance is also optimal.

Coherency has a small power cost, so we prefer uncached, incoherent for data which is never read back onto the CPU.

Lazy allocated

This is a memory type designed to only be backed by virtual address space and never physical memory since the memory should never be normally accessed. This is intended for resources which live entirely in the tile-buffer such as G-buffer attachments and depth/stencil buffers. If the memory is written to for some reason, the memory will be backed when it's accessed, but this could create stalls.

Do

- Use HOST_VISIBLE | COHERENT to stream out data from CPU to GPU. #JUST24
- Use HOST_VISIBLE | COHERENT to back static GPU resources. #JUST24 #JUST9
- Use HOST_VISIBLE | CACHED to back memory which will be read by the CPU, including the COHERENT flag if the host system supports it. #JUST24
- If writing to non-cached memory use `memcpy()` or make sure your writes are contiguous to get best efficiency from the CPU write-combine unit. #JUST24
- Persistently map CPU visible buffers which are accessed often (uniform buffer data buffers, vertex data streaming); mapping and unmapping buffers has a CPU cost associated with it. #JUST24

Don't

- Read data from uncached memory. #JUST24 #JUST9
- Use transient memory for anything other than framebuffer attachments which only live in the tile buffer. #JUST24

Impact

Using incorrect memory types may cause an application slowdown due to increased CPU processing cost, in particular for readbacks from uncached memory which can be an order of magnitude slower than cached reads.

Debugging

The critical part from a performance perspective is using cached memory for any kind of readback. When reading any buffer, add asserts which check that any CPU readbacks come from cached memory.

Debugging issues related to cache coherency can be very difficult; we recommend designing your interfaces to encourage implicit invalidates/flushes as needed.

Pipeline creation

This section is Vulkan only.

To speed up pipeline creation, use a single pipeline cache for all vkCreate*Pipeline calls, and serialize it to disk so it can be reused the next time the application is started. Calling vkPipelineCache is thread-safe and only needs to lock when the internal data structure is accessed.

Neither pipeline creation nor command buffer building benefit from using derivative pipelines. The pipeline creation flag VK_PIPELINE_CREATE_DERIVATIVE_BIT and the basePipelineHandle/basePipelineIndex parameters are ignored by the driver.

Do

- Use a pipeline cache to speed up pipeline creation. #JUST25
- Serialize the contents of the pipeline cache and reuse it for the next run. #JUST25

Don't

- Spend time on derivative pipelines; they are no-ops on Mali. #JUST25

Impact

Shaders take some time to compile and link, so failing to cache pipelines will result in increased CPU load due to the need to compile and link the same shaders multiple times.

Failing to serialize the pipeline cache will result in the need to recompile and relink every time the application is restarted, which will increase application load times.

Debugging

N/A

Command buffers

This section is Vulkan only.

The command buffer usage flags affect performance. For best performance, the VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT flag should be set. Performance is reduced if the VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT flag is set..

Do

- Set the VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT flag by default. #JUST26
- Use command buffer reuse if the alternative is replaying the exact same commands every time on the CPU anyway. #JUST26
- Consider building per-frame command buffers when reuse is considered to avoid simultaneous command buffer use. #JUST26

Don't

- Set the VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT flag unless needed. #JUST26
- Use command pools with the RESET_COMMAND_BUFFER_BIT flag. This prohibits driver from using a single large allocator for all command buffers in a pool. #JUST27

Impact

Extra CPU overhead can be observed if flags are not used appropriately.

Debugging

Evaluate every use of any command buffer flag other than the ONE_TIME_SUBMIT_BIT flag, and review that it's a sensible use of the flag combination.

Evaluate every use of vkResetCommandBuffer and see if vkResetCommandPool can't be used instead.

Descriptor sets and layouts

This section is Vulkan only.

Mali supports the minimum required number of simultaneous bound descriptor sets (4).

Mali GPUs currently require a single descriptor set per draw, sorted by resource type, so the driver has to merge all bound descriptor sets when draw calls are constructed. Updating one out of the potential 4 descriptor sets therefore involves internal table copies/rebuilding. The merged descriptor set is cached, so the first draw call after a descriptor change will have a higher overhead than following ones. The larger those sets are, the more costly the update on the CPU side will be.

With current drivers descriptor set pool allocations are not actually pooled, so calling vkAllocateDescriptorSets in the hot path is discouraged.

Do

- Try to pack the descriptor set space as much as possible. #JUST41
- Update already allocated descriptor sets, instead of resetting descriptor pools and reallocating. #JUST43
- Prefer reusing already allocated descriptor sets, and not updating them with same information every time. #JUST43
- Prefer DYNAMIC_OFFSET UBOs/SSBOs if you plan to bind the same buffer with just different offsets and the alternative is building more descriptor sets. #JUST41

Don't

- Leave holes in the descriptor set, i.e., make them sparse. #JUST41
- Don't leave unused entries - you will still pay the cost of copying and merging. #JUST41
- Allocate descriptor sets from descriptor pools in hot paths. #JUST43
- Use DYNAMIC_OFFSET UBOs/SSBOs if you never plan to change the binding offset; there is a very small additional cost for DYNAMIC_OFFSET. #JUST41

Impact

Using descriptor sets sub-optimally can increase CPU draw call overhead.

Debugging

Monitor the pipeline layout for holes, and generate a warning.

Monitor if there is contention on vkAllocateDescriptorSets, which will probably be a performance problem if it occurs, and generate a warning.

Allocating memory in Vulkan

This section is Vulkan only.

The vkAllocateMemory allocator is not designed to be used directly for frequent allocations; assume that all allocations to vkAllocateMemory are heavy-duty kernel calls to allocate pages.

Do

- Use your own allocator to sub-manage block allocations. #JUST44

Don't

- Use vkAllocateMemory unless it is to back your heap allocators with large blobs. #JUST44

Impact

Using vkAllocateMemory frequently will increase application CPU load.

Debugging

Monitor the frequency of calls to vkAllocateMemory at runtime, and ensure it is not being called often.

Vertex shading

Using index buffers

Starting with Bifrost, Mali has two geometry flows: the more recent index driven shading model (IDVS), and the older index scanning based methods. This impacts how index buffers should be laid out, in particular in the older index scanning case.

Do

- Ensure that an index buffer references every value between the min and max used index values; i.e. if the minimum index value is N and the maximum is M for a mesh, ensure that every index value in the range N-M is used in the mesh index buffer. #JUST16
- Keep index buffers as static as possible. #JUST16
- Optimize index locality for a post-transform cache. #JUST9

Don't

- Use indices which sparsely access vertices in the vertex buffer. #JUST9 #JUST16
- Implement geometry level-of-detail (LOD) by sparsely sampling vertices from the full detail mesh; create a contiguous set of vertices for each detail level required. #JUST9 #JUST16
- Use client-side index buffers. #JUST17

Impact

Inefficient mesh indexing due to index sparseness or poor spatial locality will typically show up as additional GPU vertex processing time and/or memory bandwidth, with severity depending on the complexity of the mesh and how the index buffer is laid out in memory.

Use of frequently changing index buffers, or use of client-side index buffers, will manifest as increased CPU load due to the need to scan and/or transfer the memory into a driver owned buffer.

Debugging

Scan through your index buffers prior to submission and determine if an index buffer sparsely accesses the vertex buffer, flagging buffers which include unused indices.

Attribute precision

Full FP32 "highp" precision of vertex attributes is very rarely needed. Good asset pipelines keep the data at minimum acceptable precision in final output to conserve bandwidth and memory footprint at run-time.

OpenGL ES and Vulkan can express various forms of attributes which fits every precision need:

- 8-bit/16-bit normalized
- 8-bit/16-bit int -> float conversion
- 8-bit/16-bit/32-bit int
- RGB10A2
- FP16

Do

- Use the lowest possible precision for attributes. Mali hardware can do conversion to FP16/FP32 for free on data load. #JUST21 #JUST8

Don't

- Always use FP32 for everything because it's simple. #JUST9 #JUST8

Impact

Using higher attribute precision than neccessary will manifest as higher memory bandwidth and memory footprint, as well as reduced vertex shader performance.

Debugging

Architectural design. Instrument an application to see if it's using lower-precision attribute types. A well-written application will typically have a mix of precision types.

Instanced vertex buffers

Both OpenGL ES and Vulkan have support for attribute instance divisors. However, there are some hardware limitations related to when instanced vertex buffers can be used well.

Do

- Group all attribute data using instanced attributes into a single interleaved vertex buffer. #JUST6
- Use instanced attributes to work around limitation of 16K uniform buffers. #JUST7
- If feasible, try to use a power-of-two divisor, i.e. number of vertices per instance should be power-of-two. #JUST6
- Prefer gl_InstanceID based instance data over instanced attributes if the suggestions here cannot be followed. #JUST6
- Prefer instanced attributes if instance data can be represented with smaller data types (i.e. not full FP32). #JUST8

Don't

- Use more than one vertex buffer which is instanced. #JUST6

Impact

The impact of getting this wrong is very subtle since it impacts on-chip caching of descriptors. In extreme cases it can reduce performance by a few percent.

Debugging

Statically validating this from the application side is the easiest option.

Designing the optimal vertex attribute layout

Starting with the Bifrost architecture, vertices can be shaded with an index driven flow. In the index driven flow, positions are shaded first, then, varyings are shaded for the visible vertices which survive culling. This design choice guides us to the optimal vertex buffer layout.

Do

- Use a dedicated vertex buffer for positions. #JUST12
- Interleave every non-position attribute in a single buffer. #JUST12
- On Midgard GPUs in OpenGL ES, attributes which are passed through the vertex shader unmodified should be moved to an interleaved third vertex buffer as well. #JUST13
- Keep the vertex buffer attribute stride tightly packed. #JUST9
- Consider specializing optimized versions of meshes to remove unused attributes for specific uses; e.g. generating a tightly packed buffer for shadow mapping which consists of only the position data. #JUST9

Don't

- Use one buffer per attribute. Each buffer takes up a slot in the buffer descriptor cache, and more bandwidth will be wasted fetching data for culled vertices. #JUST9
- Pad interleaved vertex buffer strides up to power-of-two to "help" the hardware. #JUST9

Impact

The impact here is mostly bandwidth, and using vertex buffers wrongly could potentially disable certain driver optimizations.

Debugging

This can be debugged by inspecting API state fairly easily.

Selecting right precision for varying outputs

Varying output from vertex shading is written to main memory on Mali. Applications need to consider the precision of varying outputs.

Typical types of varying data which usually have sufficient precision in mediump:

- Normals
- Vertex color
- Camera-local positions and directions
- Texture coordinates for reasonably sized textures (up to roughly 256x256, 512x512)

Common types of varying data which usually needs highp:

- World-local positions
- Texture coordinates for large textures
- Texture coordinates which "slide", this needs a lot of sub-texel precision in UV coordinates

Do

- Use mediump qualifier on varying outputs if the precision is acceptable. #JUST1

Don't

- Use varyings with more components and precision than needed. #JUST1

Impact

Getting this wrong will impact GPU bandwidth and system power consumption.

Debugging

This issue will show up as a larger bandwidth than expected. A good way to debug it is to force every varying to mediump and studying the difference of the bandwidth measured. If bandwidth is a bottleneck, reducing the bandwidth from varyings should impact overall FPS, otherwise this will probably just impact power consumption.

Matching geometry amount to resolution and screen size

Primitives and vertices are far more expensive than fragments, so you want to make sure that you get many pixels worth of fragment work for each primitive that is rendered. The Mali Utgard family of GPUs does not have a unified shading model, which means vertex processing and fragment shading can be bottlenecked independently. Starting with the Midgard architecture, vertex and fragment shading can happen concurrently on the shader cores. Modern content tends to be far more intense on fragment processing. Mali in general favors fragment due to being a tiled architecture, while high geometry detail is more expensive as a trade-off.

Do

- Balance your scene such that each primitive in the scene creates on average over 10-20 fragments. A budget of ~50-100k triangles for a 720p frame is a good starting point. #JUST20#JUST1
- Use common techniques such as normal mapping to reduce required geometry load for same visual quality. #JUST20
- Think about geometry LOD for distant objects; distant objects are the prime source of micro-geometry. #JUST20

- Favor improved lighting effects and textures to increase quality rather than increasing geometry count. #JUST20

Don't

- Render micro-geometry. #JUST1

Impact

Having far too much geometry in the scene can quickly make you bandwidth bound as well as triangle-setup/rasterizer bound in fragment.

Debugging

An engine can easily keep track of which meshes it's rendering and with that track how many primitives are being rendered.

Draw call size and performance

The Mali hardware currently sees a small window of vertex and tiling work when working through a render pass. If draw calls are small enough, the small window of draw calls seen by the GPU might not have enough threads to fully saturate all the cores with work. Large draw calls allow the GPU to create enough threads to keep the cores busy with work.

Do

- When possible, make each draw call large enough that there are enough vertices which need to be shaded. #JUST45
- Consider instancing when it can help avoid very small draw calls. #JUST45

Don't

- Render individual objects of quads, cubes and similar very low-poly meshes. #JUST45
- For Vulkan in particular, avoid small indexed draw calls, prefer regular vkCmdDraw() for very small draw calls. #JUST45

Tessellation

Think carefully before using Tessellation

Tessellation is very powerful, but is also incredibly easy to misuse. The sole reason for its existence is to dynamically create geometry on the fly at a very fine dynamic granularity (using a patch as the main primitive).

A very common "misuse" of tessellation is to take high-detail geometry and make it even more detailed, creating micro-geometry and wasting incredible amounts of processing power and bandwidth. On mobile, where bandwidth is at a premium, tessellation should instead be seen as a way to reduce detail when it's not needed. Remember that tilers are excellent at fragment processing at the expense of geometry processing, so massively increasing the amount of triangles should not be a step taken lightly. There are almost always better alternatives to tessellation which will perform better in practice.

Before using tessellation, ask yourself these questions:

- Is it easier to just use a higher density static model?
- Will your tessellation implementation be able to reduce geometric complexity more aggressively than a simple static LOD scheme could?
- Will it be better than smooth geo-mipmap techniques such as "Continuous-LOD Morphing Geo-Mipmap"? #LINK1

Do

- Consider tessellation *only* if it helps significantly reduce overall vertex amount while maintaining same overall quality when compared to a static mesh. #JUST28
- Only add geometry detail where it counts, i.e. silhouette edges. #JUST20

- Cull patches early to avoid useless evaluation shading. #JUST29
- Consider each primitive from tessellation to be more expensive than primitives from regular vertex shading. #JUST28
- Use https://www.khronos.org/registry/gles/extensions/EXT/EXT_primitive_bounding_box.txt or frustum cull patches yourself in control shader. #JUST29

Don't

- Use tessellation until you have evaluated other possibilities. #JUST28
- Use tessellation with fixed tessellation factors; just pre-tessellate the mesh instead. #JUST29
- Use tessellation together with geometry shaders in the same draw call. #JUST28
- Use transform feedback with tessellation. #JUST28

Impact

Using tessellation to significantly increase triangle density will generally lead to worse performance, high memory bandwidth, and increased system power consumption.

Debugging

Always evaluate performance.

Geometry shading

Think carefully before using geometry shading

Most use-cases for Geometry shading are better handled by compute shaders. "Misuse" of geometry shaders can very easily lead to reduced performance and increased bandwidth.

Do

- Find a better solution to your problem. Geometry shaders are not your solution. #JUST30

Don't

- Use geometry shaders to "filter" primitives such as triangles and pass down even more attributes down to the fragment stage. This is a waste of bandwidth. #JUST30
- Use geometry shaders to expand points to quads; just instance a quad instead. #JUST21
- Use transform feedback with geometry shaders. #JUST30
- Use primitive restart with geometry shading. #JUST30

Impact

Using geometry shading will generally lead to worse performance, high memory bandwidth, and increased system power consumption

Debugging

Assert if geometry shaders are used.

Tiling

Topology formats and primitive restart

The fixed function tiler throughput is largely dependent on the number of indices/vertices it's processing.

Do

- Prefer strip formats over list formats as this reduces number of indices needed to represent a surface. #JUST10
- Use primitive restart if it helps achieving an efficient strip format. #JUST10
- Use offline tools to optimize meshes. #JUST9

Don't

- Use index buffers with low spatial coherency. This hurts caching. #JUST9

Impact

This is rarely a significant problem in practice, as long as job chain pipelining is working. If the fragment job chain is starving however, fix that issue first.

Debugging

Use Streamline to see if you're tiler bound. Not all geometry lends itself well to be optimized, so it might be easier to reduce geometry load instead instead of tuning tiler behavior.

Fragment shading

Render target management in GLES

This topic is GLES specific

A lClear() in GLES does not necessarily translate directly to a "clear" for a tile-based architecture. Tile based renderers can clear the tile when it starts rendering, which is a "free" clear. It is important to hit this fast path.

Another important consideration is that tile-based architectures work on whole frames at a time. Never switch back-and-forth between render targets, rendering a few draw calls each time. Bind a render target, clear it, and then make all of your draw operations in a single sequence, and then switch to the next target to render.

Vulkan enforces this style of behavior where you have to begin and end render passes as one atomic unit of work. GLES applications should always implement the same behavior.

Do

- Make sure you either clear or invalidate every attachment when you start rendering to a render target, unless you really, **really**, need to preserve the render targets data. #JUST5
- Make sure color write isn't masked when clearing. #JUST5
- Make sure depth write isn't masked when clearing. #JUST5
- Make sure stencil write mask is 0xFF (all bits) when clearing. #JUST5
- At end of a render pass before changing the render target, invalidate transient attachments which are no longer needed. Depth buffer is a typical case for this. #JUST5
- If rendering a frame to just part of a framebuffer, use scissor box throughout the entire frame to restrict area of clearing and rendering. #JUST5

Don't

- Ping-pong between rendertargets. Complete your render targets before moving on to the next. #JUST5
- Use glFlush() inside a render pass. #JUST5
- Use glFinish() inside a render pass. #JUST5
- Create a D24S8 texture and only attach depth or only attach stencil. #JUST5
- Modify resources which are used in the render pass. This triggers copy-on-write behavior. #JUST5

Impact

Impact here can be massive. It is low-hanging fruit which must be fixed in all applications.

Debugging

It is fairly easy to add logging to an application and see if it's doing these things correctly. Similarly, frame debuggers can do a good job of finding these issues.

Clearing attachments in Vulkan

> This topic is Vulkan specific

Vulkan has vkCmdClearColorImage() and vkCmdClearDepthStencilImage() which allows you to clear an image outside render passes. Vulkan renderpasses support all necessary features to be able to clear images for free on Mali. Clearing images outside render passes are both time consuming and bandwidth intensive, there is usually a better way to do it.

Do

- Clear an image when beginning rendering to a framebuffer, always use loadOp = VK_LOAD_OP_CLEAR for the attachment. #JUST5
- Use loadOp = VK_LOAD_OP_DONT_CARE when you don't really need to clear (e.g. blitting a full-screen quad anyway) and you want to be more friendly to immediate mode GPUs. #JUST5

Don't

- Clear an attachment inside a render pass with vkCmdClearAttachments(). This is implemented as a "clear quad". Always use loadOp = VK_LOAD_OP_CLEAR instead of CmdClearAttachments when possible. #JUST5
- Use loadOp = VK_LOAD_OP_LOAD, unless your algorithm relies on an initial framebuffer state. #JUST5
- Use vkCmdClearColorImage() or vkCmdClearDepthStencilImage() for any image which is used inside a render pass later. #JUST5

Impact

Getting this wrong has huge consequences for both bandwidth and performance. This is critical to get right.

Debugging

Debugging can be done by review for the most part. Everywhere a render pass is created, generate warnings or errors when LOAD_OP_LOAD is detected as well as any use of vkCmdClearColorImage(), vkCmdClearDepthStencilImage() and vkCmdClearAttachments() and review all these uses. A more low level debugging technique is studying number of fragment threads created.

Using multisampling correctly in Vulkan

> This topic is Vulkan specific

Just like clearing attachments properly in Vulkan, multisampling can be integrated fully with VkRenderPass, that is, Vulkan can do multisampled resolve at the end of a subpass. This resolve functionality is key to get optimal multisampling performance. There are many other things to do as well. Typically, after the multisampled image is resolved, we don't need the multisampled image anymore. Therefore, the multisampled image must be discarded by using STORE_OP_DONT_CARE. Since multisampled images are not written to or loaded from, we can use lazily allocated transient images.

Do

- Use lazily allocated multisampled images. #JUST23
- Use pResolveAttachments in a subpass to automatically resolve multisampled color buffer to single-sampled color buffer. #JUST22
- Only STORE_OP_STORE the single-sampled color buffer. #JUST22
- Use 4x MSAA. In typical cases this is very cheap compared to other quality-improving alternatives. #JUST22

Don't

- Use vkCmdResolveImage. Thishas a significant negative impact on bandwidth and performance. #JUST22
- Use STORE_OP_STORE for multisampled attachments. #JUST23
- Use LOAD_OP_LOAD for multisampled attachments. #JUST23
- Use more than 4x MSAA, as it is not full throughput. #JUST22

Impact

For multisampling, it is critical to get this right, or performance will suffer dramatically.

Debugging

Verify that all VkAttachments which are multisampled follow the loadOp/storeOp recommendations.

Using multipass well in Vulkan

This section is Vulkan specific.

A major feature of Vulkan is multipass, which enables you to exploit the full power of tile-based architectures with a standard API. There are several things an application needs to get right in order to take full advantage of multipass.

Mali is able to take color attachments and depth attachments from one subpass and use them as input attachments in a later subpass without going via main memory. This enables powerful methods to be used very efficiently:

- Deferred shading
- Programmable blending

The Vulkan driver is able to merge subpasses together in certain conditions.

The first point is G-buffer size. Mali is designed for 128-bit per pixel of tile buffer color storage (4x MSAA / 32bpp), however, this is not a strict limit. Larger G-buffers can be used at the expense of smaller tiles, which can reduce overall throughput and more bandwidth reading tile lists.

A sensible G-buffer layout could be:

- Light (B10G11R11_UFLOAT)
- Albedo (RGBA8_UNORM)
- Normal (RGB10A2_UNORM)
- PBR material parameters/misc (RGBA8_UNORM)

This fits neatly into the 128-bit budget.

Another point is reading the depth buffer on the tile. This is one of the few cases where image layout matters. Here's a sample multipass layout which hits all the good paths:

Initial layouts

- Light (UNDEFINED)
- Albedo (UNDEFINED)
- Normal (UNDEFINED)
- PBR (UNDEFINED)
- Depth (UNDEFINED)

G-buffer pass (subpass #0)

Output attachments

- Light (COLOR_ATTACHMENT_OPTIMAL)
- Albedo (COLOR_ATTACHMENT_OPTIMAL)
- Normal (COLOR_ATTACHMENT_OPTIMAL)
- PBR (COLOR_ATTACHMENT_OPTIMAL)

Light output should be attachment #0 in the VkRenderPass so it will occupy the first render target in hardware, this is significant in the light accumulate pass.

Depth attachment

- Depth/Stencil 24/8 (DEPTH_STENCIL_ATTACHMENT_OPTIMAL)

We include light here since we can write out emissive parameters from the opaque materials. There is no extra bandwidth for writing out an extra render target, unlike desktop, so we don't have to invent clever schemes to forward emissive via the other G-buffer attachments.

Subpass dependency

Here, we need to create a VkSubpassDependency which is by-region to allow the dependency to be per-pixel.

```
VkSubpassDependency subpassDependency = {};
subpassDependency.srcSubpass = 0;
subpassDependency.dstSubpass = 1;
subpassDependency.srcStageMask =
VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
| VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT
| VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT;
subpassDependency.dstStageMask =
VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
subpassDependency.srcAccessMask =
VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT |
VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;
subpassDependency.dstAccessMask =
VK_ACCESS_INPUT_ATTACHMENT_READ_BIT;
subpassDependency.dependencyFlags =
VK_DEPENDENCY_BY_REGION_BIT;
```

Lighting pass (subpass #1)

Output attachments

- Light (COLOR_ATTACHMENT_OPTIMAL)

This time, light should blend on top of the already emissive data. We can also blend transparent objects on top of this after all lights are complete.

Input attachments

- Albedo (SHADER_READ_ONLY_OPTIMAL)
- Normal (SHADER_READ_ONLY_OPTIMAL)
- PBR (SHADER_READ_ONLY_OPTIMAL)
- Depth (DEPTH_STENCIL_READ_ONLY)

Depth attachment

- D/S 24/8 (DEPTH_STENCIL_READ_ONLY)

A **very** important point here is that once we start read the buffer, we need to mark every depth/stencil attachment from that point as read-only. This allows certain optimization in DDK which can greatly improve multipass performance.

DEPTH_STENCIL_READ_ONLY is designed for this usecase, read-only depth/stencil testing while also using it as an input attachment.

Subpass merge considerations

The driver will merge subpasses if these conditions are met:

- If the number of unique VkAttachments used for input and color attachments in all considered passes is <= 8. Depth/stencil does not count towards this limit.
- Multisample counts are the same for all attachments.
- If the merge can save a write-out/read-back. Two unrelated subpasses which don't share any data do not benefit from multipass.
- If the formats are mergeable.

Do

- Use multipass. #JUST31
- Consider a 128-bit G-buffer budget for color. #JUST31
- Use by-region dependencies between subpasses. #JUST31
- Use DEPTH_STENCIL_READ_ONLY image layout exclusively for depth after g-buffer pass is done. #JUST32
- Use lazily allocated images for every attachment except for the light buffer (which is the only texture written out to memory). #JUST33
- Use loadOp = CLEAR or DONT_CARE for every attachment. Reading in a full G-buffer from memory will have a dramatic impact on bandwidth. #JUST33

Don't

- Store G-buffer data out to memory. This is a waste of bandwidth. #JUST33

Impact

Not using multipass correctly may force the driver to use multiple physical passes, which loses all benefits of the Vulkan multipass feature.

Debugging

Outside auditing the multipass implementation, the easiest way to debug multipass performance by far is to study hardware counters. Multiple physical passes will show up as more tiles rendered, and not using DEPTH_STENCIL_READ_ONLY correctly will show up as mysterious late-Z fragment threads.

Enabling MSAA the right way in OpenGL ES

This topic is GLES specific

Similar to Vulkan, to get optimal multisampling performance on OpenGL ES, we need to make sure we take advantage of on-tile resolve capabilities. Unlike Vulkan, there is no clear way of exposing this functionality in OpenGL ES as the only portable way to do multisampled resolve is through glBlitFramebuffer which implies flushing out the full multisampled image and resolving it. There is however an extension to get this functionality. https://www.khronos.org/registry/gles/extensions/EXT/EXT_multisampled_render_to_texture.txt. This extension can render multisampled to a single-sampled image by making the framebuffer multisampled. Similarly, render buffers can be made multisampled for depth.

A caveat of this extension is that multiple render targets (MRT) are not supported.

Do

- Use GL_EXT_multisampled_render_to_texture on Mali to render multisampled to a single-sampled image. #JUST22
- Use 4x MSAA. In typical cases this is very cheap compared to other quality-improving alternatives. #JUST22

Don't

- Use glBlitFramebuffer to implement multisample resolve. #JUST22
- Use more than 4x MSAA, as it is not full throughput. #JUST22

Impact

For multisampling, it is critical to get this right, or performance will suffer dramatically.

Debugging

Check if glBlitFramebuffer is used anywhere.

Using the early-ZS depth test to your advantage

Mali GPUs include optimizations for early depth and stencil testing, this means that fragments will not be spawned if they fail the depth or stencil test. To avoid excessive overdraw, it is important to consider drawing order. Newer Mali GPUs feature forward pixel-kill which alleviates the overdraw problem somewhat with opaque geometry, but having FPK does not eliminate the need to optimize render order. FPK can be good for removing the unavoidable inter-object ordering problems.

Do

- Render all opaque geometry front to back. #JUST11
- Aim to make use of early-Z as much as possible. #JUST11

Don't

- Use discard (alpha-testing) in the fragment shader unless really needed. #JUST11
- Use alpha-to-coverage unless really needed. #JUST11
- Write depth in the fragment shader unless really needed. #JUST11
- Use a depth-prepass as an optimization, it's not in almost all cases. #JUST11

Impact

Missing out on early-depth optimization is a critical performance flaw. It is vital to get as little overdraw as possible with opaque geometry.

Debugging

The two major ways to debug this is frame-debuggers like Mali Graphics Debugger and HW counters which tell you how many threads went late-Z and how many threads were killed late-Z. A high proportion of threads being late is a serious performance problem.

Minimize stencil value changes

Many stencil masking algorithms will toggle a stencil value between a small number of values (e.g. 0 and 1) when creating and using a mask. When designing a stencil mask algorithm which builds a mask using multiple draw operations, aim to minimize the number of stencil buffer updates which occur.

Do

- Use GL_KEEP rather than GL_REPLACE if you know the values are the same. #JUST11

Don't

- Write a new stencil value needlessly. #JUST11

Impact

Mask rendering which writes a stencil value cannot be rapidly discarded, which will introduce additional fragment processing cost.

Debugging

This is an algorithmic design issue which can only be analyzed by stepping through how a stencil buffer is being used.

Blending

Blending is generally quite efficient on Mali, but more efficient for some formats than others.

Blending with multisampling has some additional overhead since the blending needs to be done per sample.

Do

- Prefer blending on unorm formats. #JUST14
- Consider splitting large UI elements which include opaque and transparent regions into opaque and transparent portions which are drawn eparately, allowing early-zs and/or FPK to remove the overdraw beneath the opaque parts.

Don't

- Use blending on floating point formats unless needed. #JUST14
- Use blending on multisampled floating point framebuffers unless **really** needed. #JUST14
- Use blending with UI elements unless you're certain that you need blending. #JUST15
- Generalize UI code where you always blend and diable, by setting alpha to 1. #JUST15

Impact

Blending in general is fairly cheap on Mali due to it being on-chip, but blending means more fragments hitting the same pixel. The important performance cases to consider is when blending is enabled for cases where blending is not needed.

Debugging

The best tool for resolving these issues is frame debuggers like Mali Graphics Debugger. To determine that too much blending is a problem, you can look at #fragment threads spawned in HW counters.

Depth pre-passes

Depth pre-pass rendering is a technique where the geometry is rendered with depth-only, then shading is applied with EQUAL depth test in order to guarantee running the complex fragment shading pixels just once. Geometry is extra costly on tiled architectures, due to having to churn through tile-lists when rendering, doubling the amount of draw calls and more geometry bandwidth. Early-Z and forward pixel kill solve these issues much better on Mali, so a depth pre-pass adds additional overhead for little additional benefit.

Do

- Rely on early-Z testing with front-to-back rendering and forward pixel kill to avoid excessive overdraw. #JUST1

Don't

- Use a depth pre-pass. #JUST1

Impact

Bandwidth, CPU overhead, performance.

Debugging

N/A

Using sampler wrapping modes optimally

> This section is Vulkan only.

Bifrost hardware can more compactly encode sampler objects if the three wrapping modes S, T, R are the same. The GLES driver will typically repack sampler objects on-demand, but Vulkan will not.

Do

- Make sure sampler wrapping modes (S, T, R) all match, even if the sampler is only used for 2D textures. VkSampler doesn't know this up front! #JUST40

Don't

- Ignore the R wrapping mode even when it's not really used. #JUST40

Impact

Potential loss of GPU sampler performance.

Debugging

Assert if S, T, R differ (even for 2D textures!) and evaluate all cases.

Taking advantage of transaction elimination in Vulkan

> This section is Vulkan only.

When performing Transaction Elimination, the GPU compares the current frame buffer with the previously rendered frame and performs a partial update only to the particular parts of the frame that have been modified, thus significantly reducing the amount of data that needs to be transmitted per frame to external memory. The comparison is done on a per tile basis.

Transaction Elimination is used for an image if:

- Sample count is 1.
- Mip-levels is 1.
- Image usage has COLOR_ATTACHMENT_BIT.
- Image usage doesn't have TRANSIENT_ATTACHMENT_BIT (as these images should never be written out to memory anyway).
- A single color RT is being used and the effective tile size is 16x16. Mali-G51 and up can use multiple render targets.

Transaction elimination is one of the few cases where the driver currently cares about image layouts. Whenever the image transitions from a "safe" to an "unsafe" image layout, the TE buffer must be invalidated. "Safe" image layouts are considered as the image layouts which are read-only, or can only be written to by fragment. These layouts currently are:

- VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL
- VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL
- VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL
- VK_IMAGE_LAYOUT_PRESENT_SRC_KHR

All other layouts are considered unsafe as they allow writes to an image outside tilebuffer writebacks, and TE buffer and the color data will be mismatched.

TE buffer invalidation could happen as part of a VkImageMemoryBarrier in vkCmdPipelineBarrier/vkCmdWaitEvents or as part of a VkRenderPass if the color attachment reference layout is different from finalLayout.

Transitioning from UNDEFINED layout preserves the actual image layout of an image, so transitioning from UNDEFINED layout will not invalidate the TE buffer on its own.

Do

- Use COLOR_ATTACHMENT_OPTIMAL image layout for color attachments. #JUST33
- Try to stay in "safe" image layouts for color attachments. E.g. COLOR_ATTACHMENT_OPTIMAL SHADER_READ_ONLY preserves the TE buffer for the next render pass. #JUST33

Don't

- Transition color attachments from "safe" to "unsafe", unless required by the algorithm. #JUST33

Impact

External memory bandwidth.

Debugging

Best way to debug this is to study the "Tile writes killed by TE" hardware counter to see if transaction elimination is being used.

Indirectly, external memory bandwidth could also be studied.

Textures

Texture formats and impact on texture pipe throughput

The Mali texture mapper can spend variable amount of cycles on fetching a texel. Typically this is 1 texel/cycle on current hardware, but various cases can trigger multiple cycles needed to shade. Common cases which trigger more than 1 cycle:

- 3D, 2 cycles
- Twice Bilinear, 2 cycles
- Depth textures, 2 cycles on Midgard architecture GPUs, 1 cycle on Bifrost architecture GPUs
- 32-bit float textures, 2 cycles
- Cubemap, 1 load-store, 1 texture cycle

Do

- Consider texture formats and dimensions if you are texture fetch throughput bound. #JUST19

Don't

- Use higher precision formats unless needed. #JUST9

Impact

Impact largely depends whether the application is texture bound or not.

Debugging

To determine if you're texture fetch bound, first try to disable all use of trilinear filtering, then try forcing smaller texture formats which don't have multicycle properties. Other than that, looking at texture instructions fetches is probably the best way to understand what's going on.

Texture and sampler descriptors

Mali GPUs cache texture and sampler descriptors in a control structure cache, which can store variable numbers of descriptors depending on their content. To ensure the maximum cache capacity in terms of descriptor entries, and therefore the best performance, it is recommended to use the following descriptor settings. #JUST9

Do

For OpenGL ES:

- Set GL_TEXTURE_WRAP_(S|T|R) to identical values
    - Note that the GL driver can specialize the sampler state based on the current texture so, unlike Vulkan, there is no need to set GL_TEXTURE_WRAP_R for 2D textures
- Do not use GL_CLAMP_TO_BORDER
- Set GL_TEXTURE_MIN_LOD to -1000.0 (default)
- Set GL_TEXTURE_MAX_LOD to +1000.0 (default)
- Set GL_TEXTURE_BASE_LEVEL to 0 (default)
- Set TEXTURE_SWIZZLE_R to GL_RED (default)
- Set TEXTURE_SWIZZLE_G to GL_GREEN (default)
- Set TEXTURE_SWIZZLE_B to GL_BLUE (default)
- Set TEXTURE_SWIZZLE_A to GL_ALPHA (default)
- Set GL_TEXTURE_MAX_ANISOTROPY_EXT to 1.0, if the EXT_texture_filter_anisotropic filtering extension is available

For Vulkan, when populating the VkSamplerCreateInfo structure:

- Set sampler addressMode(U|V|W) so they are all the same
    - Note that addressModeW must be set to be the same as U and V even when sampling a 2D texture
- Set sampler mipLodBias to 0.0
- Set sampler minLod to 0.0
- Set sampler maxLod to VK_LOD_CLAMP_NONE
- Set sampler anisotropyEnable to VK_FALSE
- Set sampler maxAnisotropy to 1.0
- Set sampler borderColor to VK_BORDER_COLOR_FLOAT_TRANSPARENT_BLACK
- Set sampler unnormalizedCoordinates to VK_FALSE

… and when populating the VkImageViewCreateInfo structure:

- Set all fields in view components to either VK_COMPONENT_SWIZZLE_IDENTITY or the explicit per-channel identity mapping equivalent
- Set view subresourceRange.baseMipLevel to 0

It should be noted that the requirements for compact samplers conflict with the Vulkan specification's recommended approach for emulating GL_NEAREST and GL_LINEAR sampling for mipmapped textures. The Vulkan specification states:

There are no Vulkan filter modes that directly correspond to OpenGL minification filters of GL_LINEAR or GL_NEAREST, but they can be emulated using VK_SAMPLER_MIPMAP_MODE_NEAREST, minLod = 0, and maxLod = 0.25, and using minFilter = VK_FILTER_LINEAR or minFilter = VK_FILTER_NEAREST, respectively.

To emulate these two texture filtering modes for a texture with multiple mipmaps levels, while also being compatible with the requirements for compact samplers, the recommended application behaviour is to create a unique VkImageView instance which references only the level 0 mipmap and use a VkSampler with pCreateInfo.maxLod setting to VK_LOD_CLAMP_NONE in accordance with the compact sampler restrictions.

Note: Direct access to textures through imageLoad() and imageStore() in shader programs (or equivalent in SPIR-V) are not impacted by this issue.

Don't

The most common misconfiguration we have seen so far for Vulkan applications is applications setting maxLod to the maximum mipmap level in the texture chain, rather than VK_LOD_CLAMP_NONE. Doing this will force the hardware to use larger cache entries to store the descriptor, so fewer descriptors can concurrently be stored and performance may be reduced.

Impact

Frequent cache misses in the control structure cache may reduce throughput of the main texture filtering unit as shader threads will be unable to progress until their control data is available. If the texture unit is the critical path unit for the current tile, this may reduce overall application performance.

Debugging

Review texture and sampler descriptor settings against the best practice recommendations. Try forcing settings to the recommended settings and see if performance improves.

sRGB textures

sRGB textures are natively supported in Mali hardware and sampling, rendering and blending on sRGB surfaces come at no performance cost. sRGB textures have far better perceptual color resolution than UNORM formats at same bit depth.

Do

- Use sRGB textures when linearity in rendering is desired. #JUST21
- Use sRGB variants of compressed formats (e.g. ASTC) for albedo textures when linearity in rendering is desired. #JUST21
- Use sRGB framebuffers when linearity in blending is desired. #JUST21

Don't

- Use 16-bit linear formats when 8-bit sRGB can suffice. #JUST21

Impact

Not using sRGB where appropriate can reduce quality of rendered images, or unnecessarily increase bandwidth if wider formats are used in favor of sRGB.

Debugging

N/A

Considering instruction cache

The I-cache is often overlooked, and it is often very difficult to identify and meaningfully optimize problems impacting it. However, there is some general advice which can be followed to minimize problems..

Do

- Prefer short shaders with many threads over long shaders with few threads. #JUST34

Don't

- Unroll huge full-resolution blur filters on Midgard. #JUST34

Impact

Debugging

It is easy to use Mali Offline Compiler to statically verify how many unique instructions are generated for a shader.

Floating point (HDR) rendering formats

The two relevant formats for rendering HDR images is B10G11R11_UFLOAT and R16G16B16A16_SFLOAT.

Do

- Prefer RGB10A2 UNORM formats for HDR rendering where small increases in dynamic range are required.
- If true floating point rendering is really required, then prefer B10G11R11 for HDR rendering as it is only 32bpp and can save a lot of bandwidth, and can keep your G-buffer budget within 128bpp. #JUST42

Don't

- Use FP16 unless B10G11R11 is demonstrated to be not good enough, or you need alpha in the framebuffer. #JUST42

Impact

Bandwidth, and performance in multipass. Exceeding 128bpp in multipass can have quite dramatic effects on performance.

Debugging

Evaluate your framebuffer formats. It's generally very easy to just try other formats to see the effect on performance.

Compute

Using compute for image processing

A common way developers use compute is to performance image processing. In general, fragment shaders enables many fixed-function features in the hardware which can speed things up. These advantages are lost when going to compute.

Advantages with fragment for image processing:

- Texture coordinates are interpolated for "free" when using varying interpolation.
- Writeout to memory can be done from tile writeout unit in larger 16x16 batches instead of scattered writes with imageStore().
- No need to range check imageStores, which might be a problem when using workgroups which don't subdivide a frame completely.
- Framebuffer compression and transactional elimination is possible.

Advantages with compute for image processing:

- It is sometimes possible to exploit shared data sets between neighbor pixels, which avoids extra passes.
- Easier to work with larger working sets per thread, this can allow reducing bandwidth due to less passes.
- For complicated algorithms like FFTs, compute is **far** better than fragment.

Essentially, it comes down to what kind of algorithm you need. The simpler the algorithms, the better fragment will be compared to compute and vice versa.

Do

- Prefer fragment for simple image processing. #JUST21
- Prefer compute when you are able to be clever (and always measure!). #JUST21
- Always prefer texture over images for reading read-only texture data. #JUST21 #JUST9

Don't

- Use imageLoad() in compute unless you need coherent reads in the dispatch (very rarely the case). Prefer texture() even in compute as it makes use of the (read-only) texture cache. #JUST9
- Use compute to process images that just completed in fragment. Once images have gone to fragment, prefer staying there. See #hardware-queue-pipelining

Impact

Overall throughput might be impacted, very hard to quantify without measurement of the specific case in question.

Debugging

N/A

Choosing good workgroup sizes

Mali Midgard and Bifrost cores have a fixed number of threads each per core and fixed number of vector registers which are shared among the threads. The hardware can split up and merge workgroups unless barriers and/or shared memory is used, so in practice the workgroup's number of threads does not matter much for completely parallel work. A possible exception could be when writing image data if you want to exploit the fact that textures are tiled in blocks, so all threads in a workgroup can become close in memory, consider an 8x8 workgroup instead of 64x1 for example, but that's more a case of organizing the memory access rather than the number of threads.

A good baseline value is 64 threads per workgroup. If barriers or shared memory are used, consider from 4 to 32 to have more independent workgroups on each shader core to avoid the worst pipeline bubbles. Later driver releases deal much better with very small workgroup sizes so it's worthwhile to experiment.

Do

- Consider 64 as a baseline value for the workgroup size. #JUST18
- If in doubt, try smaller workgroup sizes before larger ones. #JUST18
- When working with images, use square execution dimension (e.g. 8x8) to exploit optimal image tiling memory layout. #JUST9

- Use multiple of 4 number of threads in a workgroup. #JUST18
- If using barriers, consider between 4 and 32 threads per workgroup. #JUST18
- If a workgroup has "per-workgroup" work to be done, split into two passes and avoid large barriers. #JUST18
- Measure, performance behavior is not always intuitive.#JUST4

Don't

- Use more than 64 threads per workgroup. #JUST18
- Assume that barriers with small workgroups are "free" like on desktop. #JUST18

Impact

Impact can be quite significant. Always measure.

Debugging

N/A

Using shared memory well on Mali

Unlike some other architectures, Mali does not implement dedicated on-chip shared memory. It does instead rely on the load-store cache to get most of the benefits with shared memory, but this means there are some restrictions to get optimum benefit.

Do

- Keep your shared memory as small as possible, as this can reduce chances of spilling out to L2/main memory. 16-32 bytes per thread is a reasonable budget. #JUST9
- Use shared memory to share **significant** computation between threads in a workgroup. #JUST9
- Consider your workgroup sizes. Shared memory is more effective with smaller thread groups where barriers introduce less thread starvation. #JUST18
- Do as much work as you can with data before writing out to shared memory. #JUST29
- Write wide 128-bit vectors to shared memory when possible. Mali does not need "banked shared memory" trickery to get optimum performance. #JUST35
- Use barriers when sharing shared data, naive desktop-centric shader code will sometimes "skip" the barrier due to GPU-specific assumptions! #JUST36
- Use mediump and friends on shared memory blocks if possible to conserve memory. #JUST9

Don't

- Spend time copying from global memory to shared memory. This is useless on Mali and only serves to pollute load/store caches. #JUST9

- Use shared memory to implement code like

  ```
  if (localID == 0) { common_setup(); }
  barrier(); // real shader code
  ```

  Use a pre-pass instead with fewer threads active.

Impact

Impact of using shared memory wrong is very application specific.

Debugging

Analyzing performance with shared memory is generally very difficult. Trying out various approaches and checking performance is probably the only sensible way to approach it.

Shader core

Using uniforms well on Mali

Mali GPUs can promote data in uniform buffers to on-chip registers, which are loaded once instead of once every fragment. This potentially saves a lot of load-store operations in shaders. Dynamically accessed uniforms, such as array indices which are derived based on per-thread data values rather than constants, cannot be promoted to register mapped however, so application developers should try to structure their algorithms so dynamically accessed uniforms are kept to a minimum.

Do

- Prefer "plain" uniforms in GLES over uniform buffers where feasible. #JUST2
- Prefer moving uniform data to push constant registers in Vulkan. Push constants on Vulkan map naturally to register mapped uniforms. #JUST2
- If using UBOs, avoid dynamically accessing uniform data where feasible. #JUST2
- Keep your uniform data small. 128 bytes is a good rule of thumb for how much data can be promoted to registers in any given shader. #JUST3
- Measure your changes, shader-level optimizations don't always do what you expect to performance. #JUST4
- Promote uniforms to compile-time constants with specialization constants or #define if uniforms are completely static. #JUST29

Don't

- Use over-aggressive instancing and batching if it can help making more uniform accesses register mapped. There is a balance between draw calls (driver overhead) and shader core performance when it comes to uniform data. #JUST4

Impact

Over-using dynamically accessed uniforms may lead to load-store pressure. This can quickly make shaders load-store bound, which can reduce performance. This issue typically comes up in shader tuning after high-level optimizations are applied.

Debugging

A good way to verify if you're load-store bound is to use the Mali Offline Compiler. This can report number of cycles spent in arithmetic, load-store units and texturing as well as reporting number of uniform registers used. If uniform loads are an issue, load-store cycles will be the bottleneck, or close to being a bottleneck.

Using atomics well on Mali

Atomic operations is a staple of compute (and some fragment algorithms!), which allows many serial algorithms to be implemented on highly parallel GPUs with some slight modifications to the algorithms.

The fundamental performance problem with atomics is contention. Atomic operations from different shader cores hitting the same cache line will require snooping through L2 cache, which is expensive. Well tuned compute applications using atomics should aim to spread contention out by keeping the atomics operations local to a core. If a shader core "owns" a cache line in L1, atomics are very efficient on Mali. Atomics on shared memory by this nature is very cheap since all threads that have access to that shared memory is local to a single core, however this does not mean algorithms using shared memory (which is always local to a core) is necessarily better due to extra barriers needed.

Do

- Think about contention when using atomics. Is it possible to amortize the contention by having a single thread in the workgroup do the "global" atomic operation? #JUST37

Don't

- Use atomics if better multipass solutions are available. #JUST37

Impact

Heavy contention dramatically reduces overall throughput and scalability with number of cores.

Debugging

Debugging performance with atomics is quite difficult and often not intuitive to reason about. There are hardware counters for studying L1 cache snoops.

Using reduced precision

Mali GPUs have full support for reduced precision arithmetic and registers. FP16 / INT16 are usually good enough for computer graphics, especially in fragment shading. ESSL and Vulkan GLSL both have support for marking variables and temporaries as having reduced precision with the mediump keyword.

Do

- Use mediump when precision is acceptable, it is meaningful for Mali. #JUST38

Don't

- Test mediump precision on desktop GPUs and assume it'll be the same on Mali. Desktop GPUs typically ignore mediump and you won't see any difference. #JUST36 #JUST38

Impact

Needlessly using full FP32 precision can affect power and overall performance due to extra register pressure and spilling.

Debugging

Forcing mediump for everything and studying difference between full FP16 and FP32 can help figure out what gains there are to be had by using mediump.

Hardware queue pipelining

Making best use of vertex/fragment pipelining

The Mali GPUs pipeline compute/vertex work with fragment. Well performing applications should always ensure that they are not creating bubbles in this pipeline unnecessarily.

- CPU work (prepares frame N - 3)
- Vertex/tiling/compute (processes frame N - 2)
- Fragment (renders frame N - 1)
- Presentation (displays frame N)

Fragment and vertex work on Mali runs in two separate "queues". These internal queues can dispatch threads concurrently so it is important that the GPU is able to run both vertex work along with fragment shading work. The two most common problems when it comes to hardware queue pipelining is that the CPU is waiting for fragment data (glReadPixels and the like), and that compute or vertex depends on data from fragment. Reasoning for an in-order CPU pipeline applies here as well.

Do

- If an earlier pipeline stage waits for results in a later pipeline, ensure this latency is accounted for by waiting a few frames before using the result. #JUST39
- Use fences to asynchronously read back data to CPU from GPU. #JUST39
- Consider if data you depend on can be done earlier in the pipeline, compute is a great target for generating data for vertex processing. #JUST39

Don't

- Unnecessarily wait for GPU data. #JUST39
- Create backwards dependencies in the pipeline as this creates bubbles. #JUST39
- Use vkQueueWaitIdle(), vkDeviceWaitIdle() in runtime unless you specifically need it. #JUST39
- Use glFlush() or glFinish() in runtime unless you specifically need it. #JUST39

Impact

Reading back fragment shading results to CPU can be a **massive** performance problem as it breaks all high-level parallelism. Creating dependencies from fragment to vertex/compute is more subtle, but can be quite large performance problems as well.

Debugging

While glReadPixels and friends is very easy to check for, fragment -> vertex/compute hazards are more difficult to detect.

Using Pipeline Barriers correctly in Vulkan

This topic is Vulkan specific

Pipeline barriers in Vulkan is the go-to synchronization mechanism. It has a concept of pipeline stages which are synchronized with each other.

Vulkan workloads are spread over two hardware queues on the GPU. A hardware queue represents separate queues running concurrently on the GPU. The Vulkan pipeline model represents various stages.

Vertex/tiling/compute hardware queue

- VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
- VK_PIPELINE_STAGE_VERTEX_*_BIT
- VK_PIPELINE_STAGE_TESSELLATION_*_BIT
- VK_PIPELINE_STAGE_GEOMETRY_*_BIT
- VK_PIPELINE_STAGE_COMUPTE_SHADER_*_BIT
- VK_PIPELINE_STAGE_ALL_COMMANDS_BIT
- VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT

Fragment hardware queue

- VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT

- VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
- VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
- VK_PIPELINE_STAGE_ALL_COMMANDS_BIT
- VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT

From a high-level perspective, there are two kinds of synchronization, synchronization within a command buffer and synchronization between hardware queues. Synchronization within a queue is fairly lightweight, as it translates directly to dependency between commands.

Synchronization from the vertex/tiling/compute queue to the fragment one is "free" because fragment always comes after vertex/tiling in a rendering pipeline anyway. The cases to really watch out for is when pipeline stages in fragment are synchronized with pipeline stages in the vertex/tiling queue. This creates a pipeline bubble unless extra latency is accounted for.

Another aspect of pipeline barriers is transitioning image layouts. The cases where the driver currently uses image layouts are:

- PRESENT_SRC_KHR: Needed for presentation. Images in PRESENT_SRC_KHR cannot be used for anything except presentation.
- DEPTH_STENCIL_READ_ONLY_OPTIMAL: Used in multipass. Driver can use more efficient tile reads.
- COLOR_ATTACHMENT_OPTIMAL/SHADER_READ_ONLY_OPTIMAL: Used for purposes of transaction elimination on render targets.

Do

- Always consider if your dependencies are forward-feeding (vertex/compute -> fragment), or backwards (fragment -> compute/vertex). #JUST39
- Only use intra-queue barriers when you need to, put as much work as possible between barriers. #JUST39
- srcStageMask = ALL_GRAPHICS_BIT, dstStageMask = FRAGMENT_SHADING_BIT when synchronizing render passes with each other. #JUST39
- Keep your srcStageMask as early as possible in the pipeline. #JUST39
- Keep your dstStageMask as late as possible in the pipeline. #JUST39

Don't

- Needlessly starve vertex/compute for work. Vertex/compute can overlap execution with fragment! #JUST39
- Use VkEvent if you're signalling and waiting for that event right away, that's what vkCmdPipelineBarrier is for. #JUST29
- Ignore image layouts. While the impact of image layout usage is situational on Mali, other GPUs might not be that lax about it. #JUST36

Impact

Getting pipeline barriers wrong might either starve GPU (too much sync) or cause corruption (too little sync). Getting this just right is a critical component of any Vulkan application.

Debugging

Synchronization is arguably the hardest aspect of Vulkan, proceed with caution.

Windowing system - EGL / WSI

Using semaphores well in WSI

This section is Vulkan specific

A typical frame of Vulkan will look like:

- Create VkSemaphore (#1)
- AcquireImage, which signals semaphore and gives you the swapchain index #N
- Wait for fences associated with swapchain index #N
- Recycle VkSemaphore (#2)

- Build command buffer(s)
- Submit command buffer(s) to VkQueue signalling a release semaphore (#2), and **wait for semaphore (#1) from AcquireImage**
- Present, waiting for semaphore (#2)
- Repeat

The critical part of this is the text in bold, since we also need to specify which pipeline stages will actually need to wait for the WSI semaphore. Along with pWaitSemaphores[i], there's also pWaitDstStageMask[i]. This mask specifies which pipeline stages wait for the WSI semaphore. We only need VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT. We also need to transition our WSI image from either UNDEFINED or PRESENT_SRC_KHR layout to a different layout when rendering to it. This layout transition should wait for COLOR_ATTACHMENT_OUTPUT_BIT, which creates a dependency chain to the semaphore.

Do

- Use pWaitDstStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT when waiting for a semaphore from WSI. #JUST39

Don't

Impact

Getting this wrong can create large pipeline bubbles since vertex/compute will stall.

Debugging

Justifications

JUST1 - Tile based architectures vertex shade all geometry up front

Geometry data from tile based architectures can not fit in on-chip memory alone. Geometry data such as positions and varyings are stored in main memory before being used in fragment shading later.

This is a unique property of tile based architectures.

JUST2 - Mali GPUs can map uniform data to registers

Mali GPUs can preload uniform data to registers, as long as the uniform locations are known statically at compile-time.

JUST3 - Mali GPU register mapped space is limited

JUST4 - In general, hard measurement data is king

Good engineering practice is to always measure that improvements have actually improved anything, or at least, not made things worse.

JUST5 - Mali can do free framebuffer clears when a tile starts rendering

Doing this allows us to save a lot of bandwidth related to framebuffer clears which is the common case for most rendering. Applications must always aim to use the "fixed-function" clears available in the hardware and not use ad-hoc clearing methods.

JUST6 - Midgard has fewer cache entries for NPOT-divisors and images

JUST7 - Instanced attributes are implemented in hardware attribute fetcher

Outside the concerns outlied in #JUST6, instanced attributes are loaded the same way as regular attributes, making them very nifty for large-scale instancing where 16K of uniform buffer space is not practical.

JUST8 - Attribute fetching format conversion is implemented in hardware

Loading attributes is a free format conversion, this can be exploited for scenarios where you would otherwise need manual packing code to achieve same level of compression.

JUST9 - Cache is a precious resource

Like every modern compute architecture, caches must always be considered. Don't pollute caches with data which is not used before it's evicted.

JUST10 - Tiling performance is largely dependent on number of indices

The tiler is a serial HW unit which processes indices one-by-one. Processing triangle lists needs 3 indices per primitive while strips can operate at 1 primitive every 1 new index when at full rate.

JUST11 - Mali has an early-Z unit

Mali features an early-Z unit as well as a late-Z unit, it is very important to take advantage of the early-Z unit, since fragments which go late-Z might end up doing useless work.

JUST12 - Mali can do index driven vertex shading, starting with Bifrost

The Bifrost GPUs support an index driven vertex shading model. To fully exploit this, positions need to be tightly packed in cache lines so that it is possible to read only positions and still have 100% utilitzation of cache bandwidth.

JUST13 - Mali can do pass-through varyings in certain cases

In GLES, it is possible to pass-through varyings from vertex directly into fragment.

JUST14 - Mali has blending support in hardware

Mali features blending support in hardware, but only for fixed-point formats. Floating point blending is emulated, and is slightly slower. The performance hit for using blending is as "expected", more fragments are shaded.

Blending on 4x multisampled framebuffers is as efficient as 1x with fixed-point formats, but blending needs to be per-sample for floating point formats.

JUST15 - Generic code is harder to optimize

GPUs are good at very specific things. Do not write highly generic code and expect the GPU and compiler to figure everything out for you.

JUST16 - Current Mali hardware needs to know min/max of index data

Current Mali hardware needs to know the index range, either for allocating memory on the GPU or range of vertices to shade. The index buffer should specify as tight bounds as possible.

JUST17 - Client side buffers make drivers sad

Client side memory is generally bad. It forces new allocations every time the buffer is used and it must also be consumed immediately by the driver on draw time, creating stalls.

JUST18 - Number of threads available for execution on a shader core is finite

High-performing compute code always considers how parallel it is. Having a large amount of independent threads running at the same time allows higher utilization of the shader core.

Barriers typically mean that threads have to stay resident on the core because it has to wait for all threads within the group to complete. Having a smaller workgroup in this case means more independent workgroups can execute, making it more likely for there to be threads available for executing instructions at any one time.

JUST19 - Mali texture pipe is currently designed for 1 bi-linearly filtered quad / cycle

Current hardware is designed for one bi-linearly filtered quad per cycle. Trilinear and 3D textures both need to fetch two quads, which explains the two cycles needed. Depth formats need wide footprints in the texture pipe on Midgard.

JUST20 - Computer graphics is a domain full of speed hacks

If it looks good enough and runs efficiently enough, it's a winner.

JUST21 - Fixed function hardware exists for a reason

Fixed function parts of a GPU are designed to accelerate common cases, making use of them is a part of graphics programming.

JUST22 - Multisampled resolve on-tile is supported in hardware with no bandwidth hit

Mali GPUs support resolving multisampled framebuffers on-tile. Combined with tile-buffer support for full throughput in 4x MSAA makes 4x MSAA a very compelling way of improving quality with minimal speed hit.

JUST23 - Mali supports framebuffer attachments which live only in the tile buffer

An advantage with the tile buffer is that once a frame is done, there is no reason to write out certain data to main memory. Depth buffers and G-buffer values (in multipass) are prime candidates for this.

JUST24 - Mali Midgard GPUs are not cache coherent with CPU

When writing Vulkan applications for Midgard, it is vital to choose if you want coherency or cached memory. You cannot get both. Bifrost GPUs can also not be cache coherent, depending on the chipset implementation Applications will always need a fallback.

JUST25 - Pipeline caches on Mali avoid redundant shader compilation

The pipeline cache stores previously compiled pipelines. Pipelines coming from the same shader modules (but different rendering state) will very likely not have to be recompiled.

JUST26 - Command buffers can be modified by the driver

The optimal scenario is when a command buffer is only ever submitted once since the driver is able to modify the command buffer as the buffer is executed.

JUST27 - Multiple command buffers can be allocated from a single pool

Resetting all command buffers from a command pool is very cheap since only the pool allocator needs to be reset. If command buffers need to be individually reset, this optimization cannot be applied.

JUST28 - Mali is a tile based renderer, and tessellation is bandwidth heavy

As a tile-based renderer, Mali must write the post-tessellated geometry state back to main memory. This causes increased memory bandwidth.

JUST29 - Avoiding useless work is generally a good idea

Doing work which is eventually going to be thrown away is always a bad idea if it can be efficiently avoided. This also applies to doing the same work multiple times per frame.

JUST30 - Mali hardware implements geometry shading in software

As a tile-based renderer, Mali must write the geometry state back to main memory. This causes increased memory bandwidth.

JUST31 - Multipass maps really well to Mali's tile-based architecture

Multipass takes full advantage of tile-based architectures' ability to cheaply read previously rendered fragments.

JUST32 - Narrowing down your "usage" bits gives more room for optimization in driver

The less capabilities required, the easier it is to find optimal solutions.

JUST33 - Bandwidth drains battery life

Bandwidth is one of the most expensive things for power and battery life in a mobile device. Reducing bandwidth is very important in mobile applications.

JUST34 - I-cache is important as well

Just like L1 caches for data, i-cache should also be considered.

JUST35 - Wide writes are good

Writing wide words can help reduce the number of transactions to the memory system which is always good. Memory systems work on larger chunks of data at a time, and writing wide helps use the full bus. For Bifrost, it is also important to consider that transactions for all 4 threads in a quad can join together to create one joint transaction, as long as memory access is contiguous.

JUST36 - Just because it works on one GPU doesn't mean it's correct code

Code which happens to work for one GPU does not mean that the code is correct. It might still work differently on other GPUs. The most common error here is mediump, which is ignored on desktop GPUs.

JUST37 - Exclusive cache lines are happy cache lines

Memory coherency between multiple shader cores is not free. If one shader core writes to a cache line which other shader cores are currently accessing, synchronization needs to happen to make memory seem coherent. In the good case, only one core will access a particular cache line which gives good scalability to the cache system.

JUST38 - Mali supports reduced precision FP16/INT16/INT8 natively in hardware

Mali can take advantage of mediump (FP16 and INT16) which means wider vectors and more operations can potentially be done per cycle. It also reduces register pressure and can reduce bandwidth if used for varyings and attributes. Bifrost can also do SIMD within scalar registers, e.g. 2xFP16 instead of 1xFP32.

JUST39 - A GPU is a highly pipelined and asynchronous machine

Unlike CPUs, GPUs draw all their processing power from being able to overlap, pipeline long chains of work and make it highly parallel. GPUs work best when they can take full advantage of this.

JUST40 - Bifrost prefers single texture wrapping mode for all dimensions

Bifrost prefers when all wrapping modes are the same.

JUST41 - Current Mali GPUs support GLES-style linear bindings per type

The resource binding model in Mali GPUs map closely to how GLES does it, which means that descriptor sets might have to be merged to implement this.

JUST42 - Current Mali GPUs can render to floating point formats, but not natively

With on-tile buffers, Mali can render and blend to "any" format. B10G11R11 and FP16 can be rendered to at near-native rates.

JUST43 - Current drivers do not optimally implement descriptor pools

Allocating descriptor sets from a pool is like a regular allocation, and should be avoided in the critical paths.

JUST44 - Allocating Vulkan device memory is super expensive

But that's how it's designed, make your own allocators on top.

JUST45 - Current Mali GPUs are job based

Current Mali GPUs can only work on a small window of draws/dispatches at a time.

Mapping of justifications to graphics APIs

Justification	Utgard	Midgard	Bifrost	DDK version specifics
#JUST1	🟩	🟩	🟩	
#JUST2	🟨	🟩	🟩	
#JUST3	🟨	🟩	🟨	
#JUST4	🟩	🟩	🟩	
#JUST5	🟩	🟩	🟩	

#JUST6	⬜	🟩	🟩	
#JUST7	⬜	🟩	🟩	
#JUST8	🟨	🟩	🟩	
#JUST9	🟩	🟩	🟩	
#JUST10	🟩	🟩	🟩	
#JUST11	🟩	🟩	🟩	
#JUST12	⬜	⬜	🟩	
#JUST13	🟨	🟦	⬜	
#JUST14	🟩	🟩	🟩	
#JUST15	🟩	🟩	🟩	
#JUST16	🟩	🟩	🟩	
#JUST17	🟩	🟩	🟩	
#JUST18		🟩	🟩	
#JUST19	🟩	🟩	🟩	
#JUST20	🟩	🟩	🟩	
#JUST21	🟩	🟩	🟩	
#JUST22	🟩	🟩	🟩	
#JUST23	🟩	🟩	🟩	
#JUST24	⬜	🟥	⬜	
#JUST25	⬜	🟥	🟥	
#JUST26	⬜	🟥	🟥	
#JUST27	⬜	🟥	🟥	
#JUST28	⬜	🟩	🟩	
#JUST29	🟩	🟩	🟩	
#JUST30	⬜	🟩	🟩	
#JUST31	⬜	🟥	🟥	
#JUST32	🟩	🟩	🟩	
#JUST33	🟩	🟩	🟩	
#JUST34	🟨	🟩	🟨	
#JUST35	🟨	🟩	🟩	
#JUST36	🟩	🟩	🟩	
#JUST37	🟩	🟩	🟩	
#JUST38	🟩	🟩	🟩	
#JUST39	🟩	🟩	🟩	
#JUST40	🟨	⬜	🟥	
#JUST41	🟨	🟩	🟩	
#JUST42	🟨	🟩	🟩	
#JUST43	⬜	🟥	🟥	
#JUST44	⬜	🟥	🟥	
#JUST45	⬜	🟩	🟩	

Legend

⬜ No effect	🟨 Unknown	🟦 GLES only	🟥 Vulkan only	🟩 GLES and Vulkan

Appendix A
Revisions

This appendix describes the changes between released issues of this book.

It contains the following section:
- *A.1 Revisions* on page Appx-A-12.

A.1 Revisions

This appendix describes the technical changes between released issues of this book.

Table A-1 First release for r0p0

Change	Location	Affects
First release	-	-

Table A-2 Second release for r0p0

Change	Location	Affects
Second release - minor changes	-	-

Table A-3 Third release for r0p0

Change	Location	Affects
Third release - Document number updated	-	-

Table A-4 First release for r1p1

Change	Location	Affects
Addition of section: Texture and sampler descriptors	Textures	All