Arm Mali GPUs Best Practices Developer Guide

Recommendations for efficient API Usage

Pete Harris

June 2019

Table of Contents

Introduction

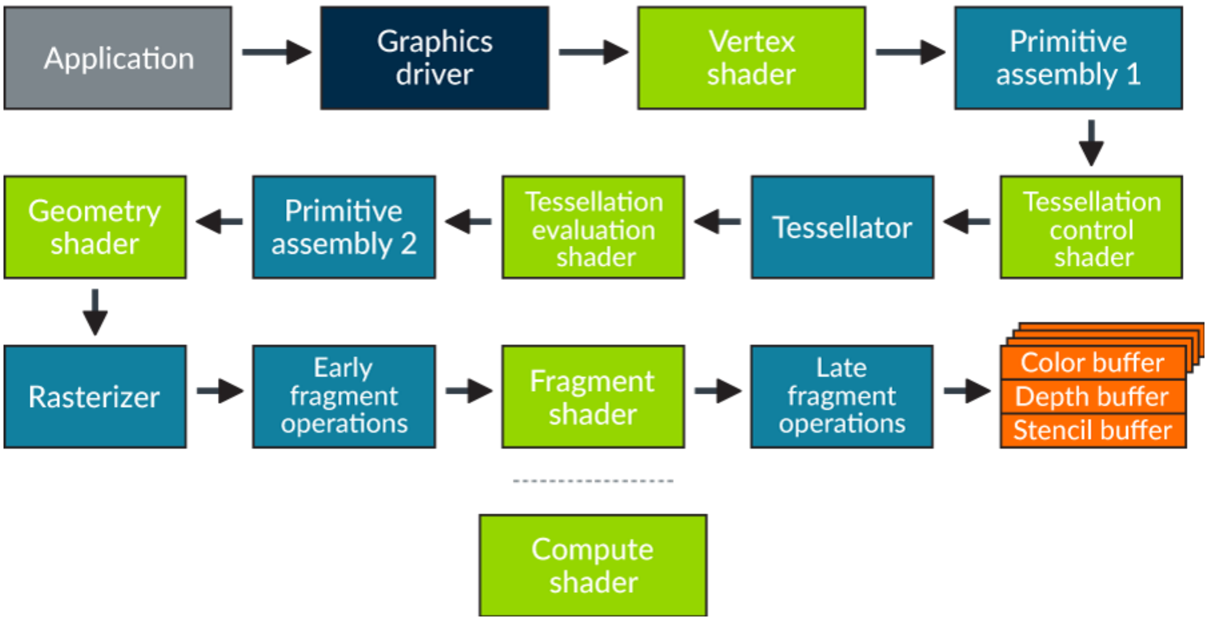
This article is a designed as a quick-reference, so assumes that the reader is familiar with using the underlying APIs; other articles will explore specific topics in more detail and take more time to explain key concepts to developers still learning the ins and outs of the APIs.

Warning

These recommendations aim to provide the best practices for Mali GPUs, however real-world applications can be complicated and there are always exceptions to this generalized advice. We highly recommend making measurements of any applied optimizations to check they are performing as intended on your target device(s).

Article structure

Graphics processing can be represented as a pipeline of processing stages that includes the application, the graphics driver, and the various hardware stages inside the GPU itself. Most stages follow a strict architectural pipeline, with outputs from one stage becoming the inputs into the next. Compute shaders are the one exception to this strict pipeline, as they simply write results back to resources stored in system memory, and therefore their outputs can be consumed by any stage in the pipeline which can consume those resources.



We have structured this article with topic ordering following this pipeline. Each topic provides recommendations which should be applied to processing running in that that pipeline stage. There are some generic topics such as shader program authoring advice which can apply to multiple pipeline stages; we've split these out into dedicated sections at the end of the document.

For each best practice topic, we provide an explanation of the recommendation, and actionable *do* and *don't* technical points which should be considered during development.

Where possible we also document the likely impact of not following the advice, and possible debugging techniques which could be applied.

Info

We provide relatively terse advice in this document for sake of brevity; for example, "*Don't use* `discard` *in fragment shaders*". There are inevitably cases where you will need to use a feature which we do not recommend using, some algorithms simply require it. Don't feel constrained by our advice to never use a feature, but in these cases at least start your algorithm design with the awareness that there might be an underlying performance implication for that algorithm that you should keep an eye on.

Application logic

At the start of each frame the application makes calls on the CPU to drive the graphics stack. The first possible source of performance issues is therefore the software running on the CPU, which may either be inside the application code, or inside the graphics driver. This section looks at advice to minimize the CPU load incurred inside the graphics driver.

Draw call batching

Committing draw calls to the command stream is an expensive operation in terms of driver CPU overheads, in particular for OpenGL ES which has a higher run-time cost per draw call than Vulkan.

Loading the render state for a draw call in to the hardware has a fixed cost which is amortized over the GPU threads which are executed for it. Draw calls containing small numbers of vertices and fragments cannot amortize this cost effectively, so applications with high volumes of small draw calls may be unable to fully utilize the available hardware performance.

Both of these issues encourage draw call batching; merging rendering for multiple objects using the same render state into a single draw call, reducing the total number of draw calls needed to render each frame and therefore reducing CPU overheads and power consumption.

Do

- Batch objects to reduce draw call count.
- Use instancing when drawing multiple copies of a mesh, using the flexibility instancing gives over static batches to more aggressively cull instances which are guaranteed to not be visible.
- Batch even if not CPU limited to reduce system power consumption.
- Aim for fewer than 500 draw calls a frame in OpenGL ES.
- Aim for fewer than 2000 draw calls a frame in Vulkan.

Note that these draw call count recommendations are very rough rules-of-thumb; the available CPU performance can vary widely from chipset to chipset.

Don't

- Make batches so large that culling and render order are badly compromised.

- Render lots of small draw calls, such as single points or quads, without batching.

Impact

- Higher application CPU load.
- Reduced performance if CPU limited.

Debugging

- Profile application CPU load.
- Trace API usage and count calls per frame.

Draw call culling

The fastest draw calls which an application can process are the ones which are discarded before they even reach the API because they are guaranteed never to be visible. Remember once a draw call hits the API it will, at a minimum, require vertex shading to get clip-space coordinates before primitives can be culled.

Do

- Cull objects which are out of frustum; e.g. bounding box frustum checks.
- Cull objects which are known to be occluded; e.g. portal culling.
- Find a balance between batching and culling.

Don't

- Send every object to the graphics API regardless of world-space position.

Impact

- Higher application CPU load.
- Higher vertex shading load and memory bandwidth.

Debugging

- Profile application CPU load.
- Trace API usage and count calls per frame.
- Use GPU performance counters to verify tiler primitive culling rates. We normally expect ~50% of triangles to be culled because they are inside the frustum but back-facing, higher culling rates than this may indicate draw call culling problems in the application logic.

Draw call render order

GPUs can most efficiently reject occluded fragments by using the early-zs test. To get the highest fragment culling rate from the early-zs unit, render all your opaque meshes first in a front-to-back render order, and then render your translucent meshes in a back-to-front render order over the top of the opaque geometry to ensure alpha blending works correctly.

Mali GPUs since Mali-T620 include an optimization called Forward Pixel Kill, which helps mitigate the cost of any fragments which are occluded and not killed by early-zs. However, do not rely on it alone; early-zs is always more efficient and consistent in terms of its benefits, and will work on older Mali GPUs which do not have the FPK feature.

Do

- Render opaque objects in a front-to-back order.
- Render opaque objects with blending disabled.

Don't

- Use `discard` in the fragment shader; this forces late-zs.
- Use alpha-to-coverage; this forces late-zs.
- Write to fragment depth in the fragment shader; this forces late-zs.

Impact

- Higher fragment shading load.

Debugging

- Render your scene without your transparent elements and use GPU performance counters to check the number of fragments being rendered per output pixel. If this is higher than 1.0 you have some kind of opaque fragment overdraw present which early-zs could remove.
- Use the GPU performance counters to check the number of fragments requiring late-zs testing, and the number of fragments being killed by late-zs testing.

Avoid depth prepasses

One common technique on PC and console games is using a depth prepass. In this algorithm the opaque geometry is drawn twice, first as a depth-only update, and then as a color render which uses an `EQUALS` depth test. This technique is designed to minimize the amount of redundant fragment processing which occurs, at the expense of doubling the draw call count and processed vertex count.

For tile-based GPUs such as the Mali GPUs, which already include optimizations such as FPK to reduce the redundant fragment processing automatically, the cost of the additional draw calls, vertex shading, and memory bandwidth nearly always outweighs the benefits. This is an "optimization" which actually ends up reducing performance in all cases we have seen it used.

Do

- Use draw call render order to maximize the benefits of early-zs.

Don't

- Use depth prepass algorithms as a means to remove fragment overdraw.

Impact

- Higher CPU load due to duplicated draw calls.
- Higher vertex shading cost and memory bandwidth due to duplicated geometry.

OpenGL ES GPU pipelining

OpenGL ES exposes a synchronous rendering model to the application developer, despite the underlying execution being asynchronous, whereas Vulkan exposes this asynchronous nature directly to the application developer to manage. In either case it is important that the application keeps the GPU fed with work, avoiding behaviors which drain the pipeline and starve the GPU of work (unless the desired target frame rate has been reached, of course).

Keeping the GPU busy not only means that you will get the best rendering performance from the platform, but also that you avoid hitting performance oscillations caused by the platform dynamic voltage and frequency scaling (DVFS) logic thinking that the CPU or GPU is under-utilized.

Do

- Do not let the GPU go idle unless the target performance is reached.
- Pipeline any use of fences and query objects; don't wait on them too early.
- Use GL_MAP_UNSYNCHRONIZED to allow use of glMapBufferRange() to patch a safe region of a buffer which is still partially referenced by in-flight draw calls.
- Pipeline any glReadPixels() calls to read asynchronously into a pixel buffer object.

Don't

- Use operations which enforce the synchronous behavior of OpenGL ES:
    - glFinish()
    - Synchronous glReadPixels()
    - glMapBufferRange() without GL_MAP_UNSYNCHRONIZED on buffer still referenced by a draw call
- Use glMapBufferRange() with either GL_MAP_INVALIDATE_RANGE or GL_MAP_INVALIDATE_BUFFER; due to a historical specification ambiguity these flags will currently trigger the creation of a resource ghost.
- Use glFlush() because this may force render passes to be split; the driver will flush as needed.

Impact

- Pipeline draining will, at a minimum, result in a loss of performance as the GPU will be partially idle for the duration of the bubble.
- Possible performance instability, depending on the interaction with the platform's DVFS power management logic.

Debugging

- System profilers such as DS-5 Streamline can show both CPU and GPU activity. Pipeline drains of this nature are normally clearly visible as periods of busy time oscillating between the CPU and GPU, with neither being fully utilized.

Vulkan GPU pipelining

Mali GPUs can run vertex/compute work at the same time as fragment processing from another render pass. Well performing applications should always ensure that they are not creating bubbles in this pipeline unnecessarily.

The main reasons for pipeline bubbles in Vulkan are:

- Not submitting command buffers often enough and either starving the GPU of work, or restricting the possible scheduling opportunities. Tile-based renderers can be particularly sensitive to this because it is so important to overlap the vertex/compute work from one render passes with the fragment work from earlier render passes.
- A data dependency when a result from a pipeline stage in render pass `N` is consumed by an earlier stage in the pipeline by a later render pass `M`, without sufficient other work between `N` and `M` to hide the result generation latency.

Do

- Submit command buffers for processing relatively frequently; e.g. for each major render pass in a frame.
- If an earlier pipeline stage waits for results from a later pipeline stage from an earlier render pass, ensure that the result generation latency is accounted for by inserting independent workloads between the two render passes.
- Consider if data you depend on can be generated earlier in the pipeline; compute is a great stage for generating input data for vertex processing.
- Consider if processing depending on data can be moved later in pipeline; e.g. fragment shading consuming fragment shading tends to work better than compute shading consuming fragment shading.
- Use fences to asynchronously read back data to the CPU; don't block synchronously and cause the pipeline to drain.

Don't

- Unnecessarily wait for GPU data on either the CPU or GPU.
- Wait until the end of the frame to submit all render passes in one go.
- Create backwards data dependencies in the pipeline without sufficient intervening work to hide the result generation latency.
- Use `vkQueueWaitIdle()` or `vkDeviceWaitIdle()`.

Impact

- The impact can be very minor or can be very significant depending on the relative sizes and ordering of the workloads which are queued.

Debugging

- The DS-5 Streamline system profiler can visualize the Arm CPU and GPU activity on both GPU queues, and can quickly show bubbles in scheduling either locally to the GPU queues (indicative of a stage dependency issue) or globally across both CPU and GPU (indicative of a blocking CPU call being used).

Vulkan pipeline synchronization

Mali GPUs expose two hardware processing slots, each slot implementing a subset of the rendering pipeline stages and able to run in parallel with the other. To get best performance it is critical that the workload expressed to the GPU allows the maximum amount of parallel processing across these two hardware slots. The mapping of Vulkan stages to the Mali GPU processing slots is shown below:

Vertex/compute hardware slot

- VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
- VK_PIPELINE_STAGE_VERTEX_*_BIT
- VK_PIPELINE_STAGE_TESSELLATION_*_BIT
- VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT
- VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT
- VK_PIPELINE_STAGE_TRANSFER_BIT

Fragment hardware slot

- VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
- VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
- VK_PIPELINE_STAGE_TRANSFER_BIT

Vulkan places the application in control of how dependencies between commands are expressed; an application must make sure that a pipeline stage for one command has produced results before a stage for a later command consumes them. There are multiple primitives in the API available for command synchronization:

- Subpass dependencies, pipeline barriers, and events are used for expressing fine-grained synchronization within a single queue.
- Semaphores are used for expressing heavier weight dependencies across queues.

All of the fine-grained dependency tools allow the application to specify a restricted scope for their synchronization, with the srcStage mask indicating which pipeline stages must be waited for, and the dstStage mask indicating which pipeline stages must wait for synchronization before they can start processing. Correctly setting up a minimal scope for synchronization, setting srcStage as early as possible in the pipeline and dstStage as late as possible in the pipeline, is critical to getting parallel processing across the two Mali hardware processing slots. Semaphores allow control over when dependent commands run using pWaitDstStages but assume that the source stage is the worst case BOTTOM_OF_PIPE_BIT, so they should be used only when no fine-grained alternative is available.

At the low-level there are two kinds of synchronization needed for Mali GPUs: synchronization within a single hardware processing slot, and synchronization across the two hardware processing slots.

- Synchronization within a hardware processing slot is lightweight, as a dependency between rendering commands can be resolved by an execution order constraint.
- Synchronization from a srcStage running in the vertex/compute processing slot to a dstStage running in the fragment processing slot is free because fragment shading always comes after vertex/compute in processing in the macro-scale rendering pipeline, so this is also just an execution order constraint.

- Synchronization from a srcStage in the fragment hardware slot to a dstStage in the vertex/compute hardware slot can be very expensive because it can create a pipeline bubble unless the extra latency for srcStageresult generation is accounted for.

Note that the TRANSFER stage is a somewhat overloaded term in the Vulkan pipeline and transfer operations may be implemented by the driver in either hardware processing slot, so the direction of a dependency backwards or forwards through the pipeline is not always obvious. Which slot is used depends on the type of transfer and the current configuration of the resource being transferred. Transfers from buffer-to-buffer are always implemented in the vertex/compute processing slot, other transfers maybe implemented in either processing slot depending on the state of the data resource being written. Which processing slot is used may materially impact the pipelining of an application's rendering workload, so beware of this, and always review how well your transfer operations are performing.

Do

- Keep your srcStageMask as early as possible in the pipeline.
- Keep your dstStageMask as late as possible in the pipeline.
- Review if your dependencies are pointing forwards (vertex/compute -> fragment) or backwards (fragment -> vertex/compute) through the pipeline, and minimize use of backwards-facing dependencies unless you can add sufficient latency between generation of a resource and its consumption to hide the scheduling bubble it introduces.
- Use srcStageMask = ALL_GRAPHICS_BIT and dstStageMask = FRAGMENT_SHADING_BIT when synchronizing render passes with each other.
- Minimize use of TRANSFER copy operations – zero copy algorithms are more efficient if possible – and always review their impact on the hardware pipelining.
- Only use intra-queue barriers when you need to and put as much work as possible between barriers.

Don't

- Needlessly starve the hardware for work; aim to overlap vertex/compute with fragment processing.
- Use the following srcStageMask -> dstStageMask synchronization pairings because they will often cause a full drain of the pipeline:
  - BOTTOM_OF_PIPE_BIT -> TOP_OF_PIPE_BIT
  - ALL_GRAPHICS_BIT -> ALL_GRAPHICS_BIT
  - ALL_COMMANDS_BIT -> ALL_COMMANDS_BIT
- Use a VkEvent if you're signaling and waiting for that event right away, usevkCmdPipelineBarrier() instead.
- Use a VkSemaphore for dependency management within a single queue.

Impact

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

Warning

Note that the presence of two distinct hardware slots which are scheduled independently for different types of workload is one aspect where tile-based GPUs like Mali are very different to desktop immediate-mode renderers. Expect to have to tune your pipelining to work well on a tile-based GPU when porting content from a desktop GPU.

Debugging

- The DS-5 Streamline system profiler can visualize the Arm CPU and GPU activity on both GPU hardware slots and can quickly show bubbles in scheduling either locally to the GPU hardware (indicative of a stage dependency issue) or globally across both CPU and GPU (indicative of a blocking CPU call being used).

Pipelined resource updates

OpenGL ES exposes a synchronous rendering model to the application developer, despite the underlying execution being asynchronous. Rendering must reflect the state of the data resources at the point that the draw call was made which means that, if an application immediately modifies a resource while a pending draw call is still referencing it, the driver must take evasive action to ensure correctness. For Mali GPUs we try to avoid blocking and waiting for the resource reference count to hit zero; this normally drains the pipeline which is bad for performance. Instead we create a new version of the resource to reflect the new state, while keeping the old version of the resource – a ghost – live until the pending draw calls have completed and the reference count drops to zero.

This is very expensive: it requires at least a memory allocation for the new resource and cleaning up the old resource when it is no longer needed, and possibly also needs a copy from the old resource buffer into the new one if the update is not a total replacement.

Do

- N-buffer resources and pipeline your dynamic resource updates so that you are not modifying resources which are still referenced by queued draw calls.
- Use GL_MAP_UNSYNCHRONIZED to allow use of glMapBufferRange() to patch an unreferenced region of a buffer which is still referenced by in-flight draw calls.

Don't

- Modify resources which are still referenced by in-flight draw calls.
- Use glMapBufferRange() with either GL_MAP_INVALIDATE_RANGE or GL_MAP_INVALIDATE_BUFFER; due to a historical specification ambiguity these flags will currently trigger the creation of a resource ghost.

Impact

- Resource ghosting will typically increase CPU load due to the memory allocation overheads and possible need for copies to build new versions of resources.
- Despite appearances it does *not* normally increase memory footprint; the commonly used alternative is N-buffering the resources manually in the application logic, which will allocate just as many copies as the resource ghosting needed. What you may see instead is some instability in the memory footprint caused by constant allocation and freeing of the resource ghosts.

Debugging

- The DS-5 Streamline system profiler can visualize the Arm CPU and GPU activity. Failing to pipeline resource updates may show as elevated CPU activity and/or idle bubbles on the GPU while the CPU is fully loaded.

CPU overheads

Shader compilation

OpenGL ES only

Shader compilation and program linkage in OpenGL ES is expensive, and on the scale of attempting to render at 60 FPS can be an operation which introduces rendering hitches and dropped frames. For Mali GPUs assume that both shader compilation and program linkage may be slow, and so should be avoided in the interactive part of your applications.

Do

- Compile shaders and link programs when starting an activity or loading a game level.

Don't

- Attempt to compile shaders during interactive gameplay; you may get performance hitches
- Rely on the Android blob cache for interactive compile and link performance; that important first-use user experience will be poor as none of your shaders will be present in the cache.

Impact

- High CPU load and dropped frames if you attempt to compile and link shaders during the interactive portions of your application.

Pipeline creation

Vulkan only

Vulkan pipelines have similar performance implications to OpenGL ES shader compilation and linkage, except for the fact that the application developer is also responsible for providing persistent caching of compiled pipelines.

Do

- Create pipelines when starting an activity or loading a game level.
- Use a pipeline cache to speed up pipeline creation.
- Serialize the pipeline cache to disk and reload it when the application is next used, giving users faster load times and a better user experience.

Don't

- Create derivative pipelines using `CREATE_DERIVATIVE_BIT`; there is no benefit in the current Mali GPU drivers.

Impact

- High CPU load and skipped frames if you attempt to create pipelines during the interactive portions of your application.
- Failing to serialize and reload a pipeline cache means that the application will not benefit from reduced load times on subsequent application runs.

Allocating memory

Vulkan only

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

Do

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

Don't

- Use vkAllocateMemory() as a general purpose allocator.

Impact

- Increased application CPU load.

Debugging

- Monitor the frequency and allocation size of all calls to vkAllocateMemory() at runtime.

OpenGL ES CPU memory mapping

OpenGL ES only

OpenGL ES provides direct access to buffer objects mapped into the application address space, allowing them to be patched in-place, by using glMapBufferRange(). The Mali OpenGL ES drivers set up these mapped buffers as uncached on the CPU, which means that streaming writes to update the buffer are very efficient but reads by the CPU are very expensive.

Do

- Make write-only buffer updates.
- Write buffers updates to sequential addresses to benefit from merging short stores in the CPU write buffer.

Don't

- Read values from mapped buffers.

Impact

- Reading from uncached buffers will show up as increased CPU load in your application functions making the memory reads.

Vulkan CPU memory mapping

Vulkan only

Vulkan adds support for far more sophisticated buffer mapping implementations, giving applications more control over the memory types they use.

The Mali GPU 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

... and 4 possible memory types on Bifrost architecture GPUs:

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

Each of these are useful for different purposes.

Not cached, coherent

The HOST_VISIBLE | HOST_COHERENT memory type is guaranteed to be supported and matches the default behavior described above for OpenGL ES buffers which is to use uncached memory on the CPU. It is the optimal type for resources which are write-only on the CPU, since it avoids polluting the CPU caches with data the CPU will never use, and writes use the CPU write buffer hardware to merge small writes into more efficient larger bursts which are sent on to the external memory device.

Cached, incoherent

The HOST_VISIBLE | HOST_CACHED memory type sets up the CPU-side memory mappings to use the CPU caches. This is ideal for resources which are mapped and read by the application software; read throughput for cached readbacks with memcpy() have been observed to be 10x faster due to the ability prefetch the data in to the CPU caches.

However, since the memory is incoherent with the GPU view of memory, calling vkFlushMappedRanges() is required when the CPU has finished writing data which needs to be sent to the GPU, and vkInvalidateMappedRanges() is required to safely read back data which has been written by GPU. Both of these will require the driver to insert manual CPU cache maintenance operations to ensure coherency between the CPU caches and the content of main memory, which can be relatively expensive. Use sparingly for resources which absolutely must be read back on the CPU.

Cached, coherent

This HOST_VISIBLE | HOST_COHERENT | HOST_CACHED memory type is only supported by the Bifrost GPUs, and even then, is only available *if* the chipset supports the hardware coherency protocol between the CPU and GPU. If the platform does not support hardware coherency the driver will expose HOST_VISIBLE | HOST_CACHEDinstead.

The use of cached memory on the CPU provides fast readbacks, and the use of hardware coherency means that the overheads of manual cache maintenance are avoided, so this is preferred over the HOST_VISIBLE | HOST_CACHED memory type when available.

The hardware coherency does have a small power cost, and it is not needed for the majority of resources which are write-only on the CPU; for these we would still recommend using the HOST_VISIBLE | HOST_COHERENT memory type which bypasses the CPU caches completely.

Lazily allocated

The LAZILY_ALLOCATED memory type is a special memory type designed to be backed by only virtual address space on the GPU, and never physical memory since the memory should be transient and never accessed outside of the local usage inside the GPU.

It is intended for resources which live entirely inside the GPU tile buffer memory and which have no use outside of the render pass which created them – such as depth/stencil buffers for simple renders and G-buffer attachments. The driver will automatically back these allocations with physical memory if required, but this could create stalls.

Do

- Use HOST_VISIBLE | HOST_COHERENT for immutable resources.
- Use HOST_VISIBLE | HOST_COHERENT for resources which are write-only on the CPU.
- When writing updates to HOST_VISIBLE | HOST_COHERENT memory use memcpy() or make sure your writes are sequential to get best efficiency from the CPU write-combine unit.
- Use HOST_VISIBLE | HOST_COHERENT | HOST_CACHED for resources which are read back on to the CPU if it is available, falling back to HOST_VISIBLE | HOST_CACHED if it is not.
- Use LAZILY_ALLOCATED for transient framebuffer attachments which only exist for the duration of a single render pass.
- Persistently map buffers which are accessed often, such as uniform buffer data buffers or dynamic vertex data buffers, as mapping and unmapping buffers has a CPU cost associated with it.

Don't

- Read back data on the CPU from uncached memory.
- Implement a suballocator which stores its CPU-side management metadata inside the memory buffer when the underlying buffer allocation is not cached on the CPU.
- Use LAZILY_ALLOCATED memory for anything other than TRANSIENT_ATTACHMENT framebuffer attachments.

Impact

- Increased CPU processing cost, in particular for readbacks from uncached memory which can be an order of magnitude slower than cached reads.

Debugging

- Check all buffers which are read on the CPU are backed by cached memory.
- Design interfaces for buffers to encourage implicit flushes/ invalidates as needed; debugging coherency failures due to missing maintenance operations without this type of infrastructure is very difficult and time consuming.

Command pools

Vulkan only

Command pools do not automatically recycle memory from deleted command buffers if the command pool was created without the RESET_COMMAND_BUFFER_BIT flag. Pools without this flag set will hold on to their memory without recycling it for new command buffer allocations until the pool is reset by the application.

Do

- Create Command pools with the RESET_COMMAND_BUFFER_BIT set, or periodically call vkResetCommandPool() to release the memory. Note that using RESET_COMMAND_BUFFER_BIT will force separate internal allocators to be used for each command buffer in the pool, which can increase CPU overhead compared to a single pool reset.

Impact

- Increased memory usage until a manual command pool reset is triggered.

Command buffers

Vulkan only

Command buffer usage flags affect performance. For best performance the ONE_TIME_SUBMIT_BIT flag should be set. Performance will be reduced if the SIMULTANEOUS_USE_BIT flag is set.

Do

- Use ONE_TIME_SUBMIT_BIT by default.
- Consider building per-frame command buffers when reuse is considered to avoid simultaneous command buffer use.
- Use SIMULTANEOUS_USE_BIT if the alternative is replaying the exact same command sequence every time in application logic; the driver can handle this more efficiently than an application replay, but less efficiently than a one-time submit buffer.

Don't

- Set SIMULTANEOUS_USE_BIT unless needed.
- Use command pools with RESET_COMMAND_BUFFER_BIT set. This prohibits driver from using a single large allocator for all command buffers in a pool, increasing memory management overhead.

Impact

- Increased CPU load will be incurred if flags are not used appropriately.

Debugging

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

- Evaluate every use of vkResetCommandBuffer() and see if it could be replaced with vkResetCommandPool() instead.

The current implementation of vkResetCommandBuffer() is more expensive than might be anticipated, and is currently equivalent to freeing and reallocating the command buffer.

Do

- Avoid calling vkResetCommandBuffer() on a high frequency call path.

Impact

- Increased CPU overhead if command buffer resets are too frequent.

Secondary command buffers

Vulkan only

The current Mali hardware does not have native support for invoking commands in a secondary command buffer, so there is additional overhead incurred when using secondary command buffers. It is expected that applications will need to use secondary command buffers, typically to allow multi-threaded command buffer construction, but it is recommended to minimize the total number of secondary buffer invocations. As with primary command buffers, we recommend avoid creating command buffers with the SIMULTANEOUS_USE_BIT because these incur much higher overheads.

Do

- Use secondary command buffers to allow multi-threaded render pass construction.
- Minimize the number of secondary command buffer invocations used per frame.

Don't

- Set SIMULTANEOUS_USE_BIT on secondary command buffers.

Impact

- Increased CPU load will be incurred.

Descriptor sets and layouts

Vulkan only

Mali GPUs support four simultaneous bound descriptor sets at the API level, but internally require a single physical descriptor table per draw call.

The driver has to rebuild this internal table for any draw call where one or more of the four API-level descriptor sets has changed. The first draw call after a descriptor change will have a higher CPU overhead than following ones which reuse the same descriptor set, and the larger the descriptor sets are the more expensive this rebuild will be.

In addition, with current drivers descriptor set pool allocations are not actually pooled so calling vkAllocateDescriptorSets() on a performance critical code path is discouraged.

Do

- Pack the descriptor set binding space as much as possible.
- Update already allocated but no longer referenced descriptor sets, instead of resetting descriptor pools and reallocating new descriptor sets.
- Prefer reusing already allocated descriptor sets, and not updating them with same information every time.
- Prefer UNIFORM_BUFFER_DYNAMIC or STORAGE_BUFFER_DYNAMIC descriptor types if you plan to bind the same UBO and/or SSBO with just different offsets and the alternative is building more descriptor sets.

Don't

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

Impact

- Increased CPU load for draw calls.

Debugging

- Monitor the pipeline layout for unused entries.
- Monitor if there is contention on vkAllocateDescriptorSets(), which will probably be a performance problem if it occurs.

Vertex shading

Index draw calls

Indexed draw calls are generally more efficient than non-indexed draw calls because they allow more reuse of vertices, for example on the seams between adjacent triangle strips. There are a number of recommendations which can improve the efficiency of indexed draw calls.

Do

- Use indexed draw calls whenever vertex reuse is possible.
- Optimize index locality for a post-transform cache.
- Ensure that an index buffer references every value between the min and max used index values, this minimizes redundant processing and data fetch.
- Avoid modifying the contents of index buffers, or use `glDrawRangeElements()` for volatile resources, otherwise the drivers must scan the index buffer to determine the active index range.

Don't

- Use client-side index buffers.
- Use indices which sparsely access vertices in the vertex buffer.
- Use indexed draw calls when mesh geometry is very simple such as a single quad or a point list, as little or no vertex reuse is possible.
- 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.

Impact

- Use of client-side index buffers will manifest as increased CPU load due to the need to allocate server-side buffer storage, copy the data, and scan the contents to determine the active index ranges.
- Use of index buffers with frequently modified contents will manifest as increased CPU load due to the need to rescan the buffer to determine the active index ranges.
- 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.

Debugging

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

Index buffer encoding

The index buffer data is one of the primary data sources into the primitive assembly and tiling process in Mali GPUs. The cost of tiling can be minimized by efficiently packing the index buffer.

Do

- Use the lowest precision index data type possible to reduce index list size.
- Prefer strip formats over simple list formats to reduce index list size.
- Use primitive restart to reduce index list size.
- Optimize index locality for a post-transform cache.

Don't

- Use 32-bit index values for everything.
- Use index buffers with low spatial coherency, because this hurts caching.

Impact

This is rarely a significant problem in practice, as long as draw calls are large enough to keep vertex shading and tiling pipelined cleanly, but small improvements like this one can accumulate into something significant when all added together.

Attribute precision

Full FP32 `highp` precision of vertex attributes is unnecessary for many uses of attribute data, such as color computation. Good asset pipelines keep the data at the minimum precision needed to produce an acceptable final output, conserving bandwidth and improving performance in return.

OpenGL ES and Vulkan can express various forms of attributes which fit every precision need, including 8-bit, 16-bit, and packed formats such as `RGB10_A2`.

Do

- Use FP32 for computing vertex positions; the additional precision is normally needed there to ensure stable geometry positioning.
- Use the lowest possible precision for other attributes; Mali GPU hardware can do conversion to FP16/FP32 for free on data load so make aggressive use of smaller formats and packed data formats if it would reduce bandwidth.

Don't

- Always use FP32 for everything because it's simple and what desktop GPUs do; you'll be leaving a lot of performance and energy efficiency on the table.
- Upload FP32 data into a buffer and then read it as a `mediump` attribute; it is simply a waste of memory storage and bandwidth as the additional precision is simply discarded.

Impact

- Higher memory bandwidth and memory footprint, as well as reduced vertex shader performance.

Attribute layout

Starting with the Bifrost architecture, vertices can be shaded using an index driven vertex shading (IDVS) flow; positions are shaded first, then varyings are shaded for the vertices of primitives which survive culling. Good buffer layout can maximize the benefit of this geometry pipeline.

Do

- Use a dedicated vertex buffer for position data.
- Interleave every non-position attribute which is passed through the shader vertex unmodified in a single buffer.
- Interleave every non-position attribute which is modified by the vertex shader in a single buffer.
- Keep the vertex buffer attribute stride tightly packed.
- 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-related attributes.

Don't

- Use one buffer per attribute; each buffer takes up a slot in the buffer descriptor cache, and more attribute bandwidth will be wasted fetching data for culled vertices when they share a cache line with visible vertices.
- Pad interleaved vertex buffer strides up to power-of-two to "help" the hardware; it just increases memory bandwidth.

Impact

- Increased memory bandwidth due to redundant data fetches.
- Potential loss of performance caused by increased pressure on caches.

Varying precision

The vertex shader outputs – commonly called varying outputs – are written back to main memory for Mali GPUs because all geometry processing must be complete before fragment shading can commence. Minimizing the precision of the varying outputs therefore pays back twice over; once when written by the vertex shader, and again when read by the fragment shader, so is important to get right.

Typical uses of varying data which usually have sufficient precision in `mediump` include:

- Normals
- Vertex color
- Camera-space positions and directions
- Texture coordinates for reasonably sized non-tiled textures (up to approximately 512x512)

Typical uses of varying data which need `highp` precision are:

- World-space positions
- Texture coordinates for large textures, or textures with high levels of UV coordinate wrapping

Do

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

Don't

- Use varyings with more components and precision than needed.
- Use vertex shaders with outputs which are unused in the fragment shader.

Impact

- Increased GPU memory bandwidth.
- Reduced vertex shader and fragment shader performance.

Triangle density

The bandwidth and processing cost of a vertex is typically in order of magnitude higher than the cost of processing a fragment. Make sure that you get many pixels worth of fragment work for each primitive that is rendered, allowing the expense of the vertex processing to be amortized over multiple pixels of output.

Do

- Use models which create *at least* 10-20 fragments per primitive.
- Use dynamic mesh level-of-detail, using simpler meshes when objects are further away from the camera.
- Use techniques such as normal mapping to bake the complex geometry needed for per-pixel lighting into a simpler run-time mesh and a supplemental texture. The ASTC texture compression format includes compression modes explicitly designed for compressing normal maps, which can reduce bandwidth needs even further.
- Favor improved lighting effects and textures to increase final image quality, rather than using brute-force increases in geometry count.

Don't

- Generate microtriangles.

Impact

- High volumes of geometry can cause many problems for a tile-based renderer, both in terms of shading performance, memory bandwidth, and system energy consumption due to the memory traffic.

Debugging

- Monitor total primitive counts and sanity check against pixel counts; for a 1080p render pass you shouldn't really need more than 250K primitives (average of 16 fragments per front-facing triangle).
- Bifrost GPUs include a hardware performance counter to check the number of microtriangles being killed because they generate no sample coverage; if this number is more than a few percent review object meshes and consider more aggressive mesh level-of-detail selection at runtime.

Instanced vertex buffers

Both OpenGL ES and Vulkan have support for instanced drawing, using attribute instance divisors to determine how to partition a buffer to locate the data for each instance. There are some hardware limitations related to when instanced vertex buffers can be used well.

Do

- Use a single interleaved vertex buffer for all instance data.
- Use instanced attributes to work around limitation of 16KB uniform buffers.
- If feasible, try to use a power-of-two divisor, i.e. number of vertices per instance should be power-of-two.
- Prefer indexed lookups using [gl_InstanceID] into uniform buffers or shader storage buffers if the suggestions here cannot be followed.
- Prefer instanced attributes if instance data can be represented with smaller data types, as uniform buffers and shader storage buffers cannot utilize these denser data types.

Don't

- Use more than one vertex buffer which is instanced.

Impact

- Reduced performance for the impacted instanced draw calls.

Tessellation

Tessellation is very powerful, but as a somewhat brute-force technique it is also incredibly easy to misuse, in particular on tile-based GPUs where the output of the geometry processing is written back to main memory.

Do

- Always consider alternatives first:
  - For character meshes, consider simply adding more static mesh levels-of-detail.
  - For terrain meshes, consider smooth geo-mipmap morphing techniques.
- If you must use it:
  - Only add geometry where it benefits most; i.e. the silhouette edges of meshes.
  - Remember that tessellation can be used to *reduce* geometry complexity where not needed, not only to increase it.
  - Cull patches to avoid redundant evaluation shader invocations. Use theEXT_primitive_bounding_box extension, or frustum cull patches yourself in the control shader.
  - Avoid microtriangulation at all costs; remember that there are rapidly diminishing perceptual quality benefits to increasing triangle density below 10 fragments per primitive, especially on mobile displays which are often 500+ DPI.
  - Consider forcefully clamping your maximum tessellation factor to prevent accidental triangle count explosion.

Don't

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

Impact

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

Debugging

- It is very easy to turn on tessellation and fail to spot that it is generating millions of triangles because it is procedurally generated and somewhat hidden from the application. Always check the GPU performance counters to monitor how many triangles you are generating, and how many are microtriangles which are killed because they generate no sample coverage.
- Check performance with various levels of `min()` clamp applied to the tessellation factors in the control shader, allowing controlled exploration of the performance versus visual benefit trade-offs.

Geometry shading

As with tessellation shaders, think carefully before using geometry shading, considering you are working on a tile-based architecture which is sensitive to geometry bandwidth. Most use-cases for geometry shading are now better handled by compute shaders, as they are more flexible and give the application developer more ability to avoid the naive duplication of vertices which most geometry shaders will trigger.

Do

- Consider using more efficient alternatives, such as compute shaders.

Don't

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

Impact

- Using geometry shading will generally lead to significantly increased memory bandwidth, with the knock-on effect that will have on performance and energy efficiency.

Tiling

Effective triangulation

Tiling and rasterization both work on fragment patches larger than a single pixel; e.g. for Mali GPUs the tiling will use bins at least as large as 16x16 pixels, and fragment rasterization will emit 2x2 pixel quads for fragment shading. Best performance is achieved when mesh triangulation uses as few triangles as possible to cover the necessary pixels, in particular trying to maximize the ratio of triangle area to edge length.

Do

- Use triangles which are as close to equilateral as possible; this maximizes the ratio of area to edge length, reducing the number of partial fragment quads which are spawned.

Don't

- Use triangle fans or similar geometry layouts; the center point of the fan has a very high triangle density for very low pixel coverage per triangle loaded.

Impact

- Increased fragment shading overhead due to the triangle fetch and partial sample coverage of the 2x2 pixel fragment quads.

Debugging

- Mali Graphics Debugger contains mesh visualization tools, allowing the the outline of the mesh submitted to a draw call to be visualized in object-space.

Fragment shading

Efficient render passes

Tile-based rendering operates on the concept of render passes; each render pass has an explicit start and end and produces an output in memory only at the end of the pass. The start of the pass corresponds to initializing the tile memory inside the GPU, and the end of the pass corresponds to writing out the required outputs back to main memory. All of the intermediate framebuffer working state lives entirely inside the tile memory and is never visible in main memory.

OpenGL ES only

OpenGL ES does not have explicit render passes in the API; they are inferred by the driver based on framebuffer binding calls. A render pass for framebuffer n starts when it is bound as the GL_DRAW_FRAMEBUFFER target, and normally ends when the draw framebuffer binding changes to another framebuffer[1]. Applications should write their FBO usage to maximize the efficiency of this implicit render pass handling.

Do

- Clear or invalidate every attachment when you start a render pass, unless you really need to preserve the content of a render target to use as the starting point for rendering.
- Ensure that color/depth/stencil writes are not masked when clearing; you must clear the entire content of an attachment to get a fast clear of the tile memory.
- Invalidate attachments which are not needed outside of a render pass at the end of the pass *before* changing the framebuffer binding to the next FBO.
- If you know you are rendering to a sub-region of framebuffer use a scissor box to restrict the area of clearing and rendering required.

Don't

- Switch back and render to the same FBO multiple times in a frame; complete each of your render passes in a single glBindFramebuffer() call before moving on to the next.
- Use glFlush() or glFinish() inside a render pass; this will split the pass.
- Create a packed depth-stencil texture (D24_S8, D32F_S8) texture and only attach one of the two components as an attachment.

Impact

- Correct handling of render passes is critical; failing to follow this advice can result in significantly lower fragment shading performance and increased memory bandwidth due to the need to read non-cleared attachments into the tile memory at the start of rendering, and write out non-invalidated attachments at the end of rendering.

Debugging

- Review API usage of framebuffer binding, clears, draws, and invalidates.

Vulkan only

Vulkan render passes are explicit concepts in the API, with defined loadOp operations – which define how Mali GPUs initalize the tile memory at the start of a pass – and storeOp operations – which define what is written back to main memory at the end of a pass. In addition, Vulkan introduces the concept of lazily allocated memory which means that transient attachments which only exist for the duration of a single render pass do not normally even need physical storage.

Do

- Clear or invalidate each attachment at the start of a render pass using `loadOp = LOAD_OP_CLEAR` or `loadOp = LOAD_OP_DONT_CARE`.
- Set up any attachment which is only live for the duration of a single render pass as a `TRANSIENT_ATTACHMENT` backed by `LAZILY_ALLOCATED` memory and ensure that the contents are invalidated at the end of the render pass using `storeOp = STORE_OP_DONT_CARE`.

Don't

- Use `vkCmdClearColorImage()` or `vkCmdClearDepthStencilImage()` for any image which is used inside a render pass later; move the clear to the render pass `loadOp` setting.
- Clear an attachment inside a render pass with `vkCmdClearAttachments()`; this is not free, unlike a clear or invalidate load operation.
- Clear a render pass by manually writing a constant color using a shader program.
- Use `loadOp = LOAD_OP_LOAD` unless your algorithm actually relies on the initial framebuffer state.
- Set `loadOp` or `storeOp` for attachments which are not actually needed in the render pass; you'll generate a needless round-trip via tile-memory for that attachment.
- Use `vkCmdBlitImage` as a way of upscaling a low-resolution game frame to native resolution if you will render UI/HUD directly on top of it with `loadOp = LOAD_OP_LOAD`; this will be an unnecessary round-trip to memory.

Impact

- Correct handling of render passes is critical; failing to follow this advice can result in significantly lower fragment shading performance and increased memory bandwidth due to the need to read non-cleared attachments into the tile memory at the start of rendering, and write out non-invalidated attachments at the end of rendering.

Debugging

- Review API usage of render pass creation, and any use of `vkCmdClearColorImage()`, `vkCmdClearDepthStencilImage()` and `vkCmdClearAttachments()`.

Multisampling

For most uses of multisampling it is possible to keep all of the data for the additional samples in the tile memory inside of the GPU and resolve the value to a single pixel color as part of tile write-back. This means that the additional bandwidth of those additional samples never hits external memory, which makes it exceptionally efficient.

OpenGL ES only

Unfortunately there is no efficient way to resolve multisample data as part of the core OpenGL ES standard for render-to-texture render passes; you have to write the multisampled framebuffer back to main memory and then use a second pass by calling glBlitFramebuffer() to implement the resolve. As you might expect, this is very expensive as all of the intermediate samples must be written to memory and then re-read by the blit.

To get optimal render-to-texture multisampling performance on OpenGL ES applications must use the [EXT_multisampled_render_to_texture]() extension. This extension can render multisampled data directly into a single-sampled image in memory, without needing a second pass.

Do

- Use 4x MSAA if possible; it's not expensive and provides good image quality improvements.
- Use EXT_multisampled_render_to_texture for render-to-texture multisampling.

Don't

- Use glBlitFramebuffer() to implement multisample resolve.
- Use more than 4x MSAA without checking performance, as it is not full throughput.

Impact

- Failing to get an inline resolve can result in substantially higher memory bandwidth and reduced performance; manually writing and resolving a 4xMSAA 1080p surface at 60 FPS requires 3.9GB/s of memory bandwidth compared to just 500MB/s when using the extension.

Debugging

- Review if glBlitFramebuffer() is used.

Vulkan only

Multisampling can be integrated fully with Vulkan render passes, allowing a multisampled resolve to be explicitly specified at the end of a subpass.

Do

- Use 4x MSAA if possible; it's not expensive and provides good image quality improvements.
- Use loadOp = LOAD_OP_CLEAR or loadOp = LOAD_OP_DONT_CARE for the multisampled image.
- Use pResolveAttachments in a subpass to automatically resolve a multisampled color buffer into a single-sampled color buffer.
- Use storeOp = STORE_OP_DONT_CARE for the multisampled image.
- Use LAZILY_ALLOCATED memory to back the allocated multisampled images; they do not need to be persisted into main memory and therefore do not need physical backing storage.

Don't

- Use vkCmdResolveImage(); this has a significant negative impact on bandwidth and performance.
- Use storeOp = STORE_OP_STORE for multisampled image attachments.
- Use storeOp = LOAD_OP_LOAD for multisampled image attachments.

- Use more than 4x MSAA without checking performance, as it is not full throughput.

Impact

- Failing to get an inline resolve can result in substantially higher memory bandwidth and reduced performance; manually writing and resolving a 4xMSAA 1080p surface at 60 FPS requires 3.9GB/s of memory bandwidth compared to just 500MB/s when using an inline resolve.

Multipass rendering

A major feature of Vulkan is multipass rendering, which enables applications to exploit the full power of tile-based architectures using the standard API. Mali GPUs are 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 algorithms — such as deferred shading or programmable blending — to be used very efficiently but needs a few things to be set up correctly.

Per-pixel storage requirements

Most of the Mali GPUs are designed for rendering 16x16 pixel tiles with 128-bit per pixel of tile buffer color storage, although some of the more recent GPUs such as Mali-G72 increase this to up to 256-bits per pixel. G-buffers requiring more color storage than this can be used at the expense of needing smaller tiles during fragment shading, which can reduce overall throughput and increase bandwidth reading tile lists.

For example, a sensible G-buffer layout that fits neatly into a 128-bit budget could be:

- Light: `B10G11R11_UFLOAT`
- Albedo: `RGBA8_UNORM`
- Normal: `RGB10A2_UNORM`
- PBR material parameters/misc: `RGBA8_UNORM`

Image layouts

Multipass rendering is one of the few cases where image layout really matters, as it materially impacts the optimizations which the driver is allowed to enable. 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`
- Depth: `DEPTH_STENCIL_ATTACHMENT_OPTIMAL`

Note that the light attachment, which becomes our eventual output, should be attachment #0 in the VkRenderPass so it will occupy the first render target slot in the hardware to gain a slight performance boost. We include light here as an output from the G-buffer pass since a render might want to write out emissive parameters from opaque materials. As we will merge the subpasses there is no extra bandwidth for writing out an extra render target, so – unlike desktop – we don't have to invent clever schemes to forward emissive light contribution via the other G-buffer attachments.

Lighting pass (subpass #1) input attachments:

- Albedo: SHADER_READ_ONLY_OPTIMAL
- Normal: SHADER_READ_ONLY_OPTIMAL
- PBR: SHADER_READ_ONLY_OPTIMAL
- Depth: DEPTH_STENCIL_READ_ONLY

An important point here is that once any pass has started to read from the tile buffer, every depth/stencil attachment from that point onwards should be marked as read-only. This allows optimizations which can greatly improve multipass performance; DEPTH_STENCIL_READ_ONLY is designed for this use case, read-only depth/stencil testing while also concurrently using it as an input attachment to the shader program for programmatic access to depth values.

Lighting pass (subpass #1) output attachments:

- Light: COLOR_ATTACHMENT_OPTIMAL

Lighting computed during subpass #1 should be blended on top of the already computed emissive data provided during subpass #0. An application could also blend transparent objects on top of this after all lights are complete if needed.

Subpass dependencies

Subpass dependencies between the passes must use a VkSubpassDependency which sets the DEPENDENCY_BY_REGION_BIT flag; this tells the driver that each subpass can only depend on the previous subpasses at that pixel coordinate and so there are guaranteed to be no out-of-tile data reads from previous subpasses.

For our example above, the subpass dependency setup would look like:

```
VkSubpassDependency subpassDependency = {};

subpassDependency.srcSubpass = 0;

subpassDependency.dstSubpass = 1;


subpassDependency.srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BI
T |

                        VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT |

                        VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT;


subpassDependency.dstStageMask = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT |
                          VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT |
                          VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT |
                          VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT;


subpassDependency.srcAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT |
                          VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT


subpassDependency.dstAccessMask = VK_ACCESS_INPUT_ATTACHMENT_READ_BIT |
                          VK_ACCESS_COLOR_ATTACHMENT_READ_BIT |
                          VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT |
                          VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT |
                          VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;


subpassDependency.dependencyFlags = VK_DEPENDENCY_BY_REGION_BIT;
```

Subpass merge considerations

The driver will merge subpasses if these conditions are met:

- If the color attachment data formats are mergeable.
- If merge can save a write-out/read-back; two unrelated subpasses which don't share any data do not benefit from multipass and will not be merged.
- If the number of unique VkAttachments used for input and color attachments in all considered subpasses is <= 8. Note that depth/stencil does not count towards this limit.
- The depth/stencil attachment does not change between subpasses.
- Multisample counts are the same for all attachments.

Do

- Use multipass.
- Use a 128-bit G-buffer budget for color.
- Use by-region dependencies between subpasses.
- Use DEPTH_STENCIL_READ_ONLY image layout for depth after the G-buffer pass is done.
- Use LAZILY_ALLOCATED memory to back images for every attachment except for the light buffer, which is the only texture written out to memory.
- Follow the basic render pass best practices, with LOAD_OP_CLEAR or LOAD_OP_DONT_CARE for attachment loads and STORE_OP_DONT_CARE for transient stores.

Don't

- Store G-buffer data out to memory.

Impact

- Not using multipass correctly may force the driver to use multiple physical passes, sending intermediate image data back via main memory between passes. This loses all benefits of the multipass rendering feature.

Debugging

- The GPU performance counters provide information about the number of physical tiles rendered, which can be used to determine if passes are being merged.
- The GPU performance counters provide information about the number of fragment threads using late-zs testing, a high value here can be indicative of failing to use `DEPTH_STENCIL_READ_ONLY` correctly.

HDR rendering

For mobile content the relevant formats for rendering HDR images are `RGB10_A2`(unorm), `B10R11G11` and `RGBA16F` (float).

Do

- Prefer `RGB10_A2` UNORM formats for rendering where small increases in dynamic range are required.
- Prefer `B10G11R11` for floating point rendering as it is only 32bpp and can save a lot of bandwidth compared to full fp16 float and can help keep your G-buffer budget within 128bpp.

Don't

- Use `RGBA16F` unless `B10G11R11` is demonstrated to be not good enough, or you really need alpha in the framebuffer.

Impact

- Increased bandwidth and reduced performance.
- May make fitting into 128bpp difficult for multipass rendering.

Stencil updates

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 `KEEP` rather than `REPLACE` if you know the values are the same.
- In algorithms performing pairs of draws – one to create the stencil mask, and one to color the unmasked fragments – use the second draw to reset the stencil value ready for the next pair, avoiding the need for a separate mid-pass clear operation.

Don't

- Write a new stencil value needlessly.

Impact

- A fragment which writes a stencil value cannot be rapidly discarded, which introduces additional fragment processing cost.

Blending

Blending is generally quite efficient on Mali GPUs, because we have the `dstColor`readily available inside the tile buffer on-chip, but it is more efficient for some formats than others.

Do

- Prefer blending on unorm formats.
- Always disable blending if you know an object is opaque.
- Keep an eye on the number of blended layers per pixel being generated; high layer counts can quickly consume cycles due to the number of fragments which need to be processed even if the shaders are simple.
- Consider splitting large UI elements which include opaque and transparent regions into opaque and transparent portions which are drawn separately, allowing early-zs and/or FPK to remove the overdraw beneath the opaque parts.

Don't

- Use blending on floating point framebuffers.
- Use blending on multisampled floating point framebuffers.
- Generalize user interface rendering code so you always leave blending enabled and "disable" it by setting fragment alpha to 1.0.

Impact

- Blending disables many of the big optimizations which can help remove fragment overdraw, such as early-zs testing and FPK, and so can have a significant impact on performance. This is particularly true for user interface rendering and 2D games, which often produce multiple layers of sprites.

Debugging

- Use Mali Graphics Debugger to step through the construction of a frame, monitoring which draw calls are blended and how much overdraw they are creating.

Transaction elimination

Vulkan only

Transaction elimination is a Mali technology to avoid framebuffer write bandwidth if the rendered color of a tile is the same as the data already in memory, only writing out tiles where a tile signature comparison with a signature buffer in memory fails. This technology is of particular benefit for user interfaces and games which contain significant amounts of static opaque overlays.

Transaction elimination is used for an image if:

- Sample count is 1
- Mipmap level is 1
- Image usage has `COLOR_ATTACHMENT_BIT`
- Image usage doesn't have `TRANSIENT_ATTACHMENT_BIT`
- A single color attachment is being used (except Mali-G51)

- The effective tile size, determined by pixel data storage requirements, is 16x16 pixels

Transaction elimination is one of the few cases where the driver actively cares about image layouts. Whenever the image transitions from a "safe" to an "unsafe" image layout, the transaction elimination signature buffer must be invalidated. "Safe" image layouts are considered as the image layouts which are either read-only or which can only be written to by fragment shading. These layouts 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 of the tile write path, which means the signature metadata and the actual color data may become desychronized. Signature buffer invalidation could happen as part of a VkImageMemoryBarrier, vkCmdPipelineBarrier(), vkCmdWaitEvents(), or as part of a VkRenderPass if the color attachment reference layout is different from the final layout.

Transitioning from UNDEFINED layout preserves the memory content of an image, so transitioning from UNDEFINED layout will not, by itself, invalidate the signature buffer unless some other triggering effect requires it.

Do

- Use COLOR_ATTACHMENT_OPTIMAL image layout for color attachments.
- Try to use the "safe" image layouts for color attachments to avoid unnecessary signature invalidation.

Don't

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

Impact

- Loss of transaction elimination will increase external memory bandwidth for scenes with static regions across frames. This may reduce performance on systems which are memory bandwidth limited.

Debugging

- The GPU performance counters can count the number of tile writes killed by transaction elimination, so you can determine if it is being triggered at all.

Buffers

Buffer update

OpenGL ES only

OpenGL ES implements a synchronous programming model with each API call behaving as if the rendering triggered by earlier API calls has completed. In reality this is a complete illusion; rendering is handled asynchronously often completing milliseconds after the triggering API call was made. To maintain this illusion the driver must track the resources referenced by pending rendering commands and lock them to prevent modification until those rendering commands have completed. If the application attempts to modify a locked resource then the driver must take some evasive action, either draining the pipeline until the lock is released, or creating a new "ghost" copy of the resource to contain the modifications. Neither of these is free, and incurs overhead which can be avoided by the application.

Do

- Perform modification of buffers using glMapBufferRange() and MAP_UNSYNCHRONIZED to explicitly bypass the synchronous rendering and resource lock semantics.
- Design buffer use to cycle though N buffers, where the value of N is high enough to ensure that any resource locks have been released before a buffer is used again.
- Prefer complete buffer replacement using glBufferData() over partial replacement of buffers that are still referenced by pending commands.

Don't

- Use glBufferSubData() to modify buffers that are still referenced by pending commands.

Impact

- Pipeline draining stalls the application until the resource lock is released, causing no increase in CPU load but reducing GPU processing efficiency.
- Resource ghosting requires a new buffer to be allocated and any parts not overwritten to be copied from the original, which will increase CPU load.

Robust buffer access

Vulkan only

Vulkan devices support a feature called robustBufferAccess. When enabled this feature adds bounds checking to GPU buffer memory accesses, ensuring that accesses cannot fall outside of the buffer. This prevents hard failure modes, such as a GPU segmentation fault, but still leaves unpredictable rendering as the resulting behavior for out-of-bounds accesses is implementation defined.

Bounds checking on Mali GPUs is not always free; enabling this feature will likely cause some loss of performance, in particular for accesses to uniform buffers and shader storage buffers.

Do

- Use robustBufferAccess as a debugging tool during development.
- Disable robustBufferAccess in release builds, unless the application use case really needs the additional level of reliability due to use of unverified user-supplied draw parameters.
- If robustBufferAccess is required, use push constants in preference to uniform buffers. This reduces the number of buffer accesses required per frame, so fewer bounds checks are needed.

Don't

- Enable robustBufferAccess unless it is really needed.

- Enable `robustBufferAccess` without reviewing the performance impact.

Impact

- Use of `robustBufferAccess` may cause measurable performance loss.

Debugging

- Performance impact can be verified by comparing two test runs - one with `robustBufferAccess` and one without.
- The `robustBufferAccess` feature is a very useful debug tool during development. If your application is having problems with crashes or `DEVICE_LOST` errors being returned, enable the robust access feature and see if the problem stops. If it does, then you have a draw call or compute dispatch making an out-of-bounds access.

Textures

Sampling performance

The Mali GPU texture unit can spend variable amounts of cycles sampling a texture, depending on texture format and filtering mode. The texture unit is designed to give full speed performance for both nearest sample and bilinear filtered (LINEAR_MIPMAP_NEAREST) texel sampling.

Ignoring data cache effects, the cases which require additional cycles are:

- Trilinear (LINEAR_MIPMAP_LINEAR) filtering: 2x cost
- 3D formats: 2x cost
- FP32 formats: 2x cost
- Depth formats: 2x cost (Utgard / Midgard), 1x cost (Bifrost)
- Cubemap formats: 1x cost per cube face accessed
- YUV formats: Nx cost where N is the number of texture planes for old Mali GPUs. For the second generation Bifrost architecture cores (Mali-G51 onwards) YUV is 1x cost, irrespective of plane count.

For example, a trilinear filtered RGBA8 3D texture access will take 4 times longer to filter than a bilinear 2D RGBA texture access.

Do

- Use the lowest resolution you can.
- Use the narrowest precision you can.
- Use offline texture compression, such as ETC, ETC2, or ASTC, for static resources.
- Always use mipmaps for textures in 3D scenes to improve both texture caching and image quality.
- Use trilinear filtering selectively if your content is texture-rate limited; it gives the most benefits to textures with fine detail such as text rendering.
- Use mediump samplers; highp samplers may be slower due to wider data path requirements.
- Consider using packed 32-bit formats, such as RGB10_A2 or RGB9_E5, as an alternative to FP16 and FP32 textures if you need higher dynamic range.
- Use the ASTC decode mode extensions – where available – to lower the intermediate precision of the decompressed data. This can significantly improve filtering performance and energy efficiency on Mali-G77 onwards.
  - **OpenGL ES:** EXT_texture_compression_astc_decode_mode
  - **Vulkan:** VK_EXT_astc_decode_mode

Don't

- Use the wider data types unless needed.
- Use trilinear filtering for everything.

Impact

- Content loading too much texture data, sparse sampling due to missing mipmaps, or using wide data types, may experience issues due to either texture cache pressure, or general external memory bandwidth issues.

- Content making effective use of texture compression and mipmapping it typically limited by filtering performance rather than external memory bandwidth. This type of content will only see a measurable impact if the texture unit is the critical path unit for the shaders; if a shader is dominated by its arithmetic cost then the texture filtering may be hidden beneath that.

Debugging

- The GPU performance counters can directly show utilization of the texture unit to determine if your application is texture filtering limited, and external bandwidth counters can monitor traffic to the external system memory.
- Try forcefully disabling trilinear filtering and see if performance improves.
- Try forcefully clamping texture resolution and see if performance improves.
- Try narrowing texture formats and see if performance improves.

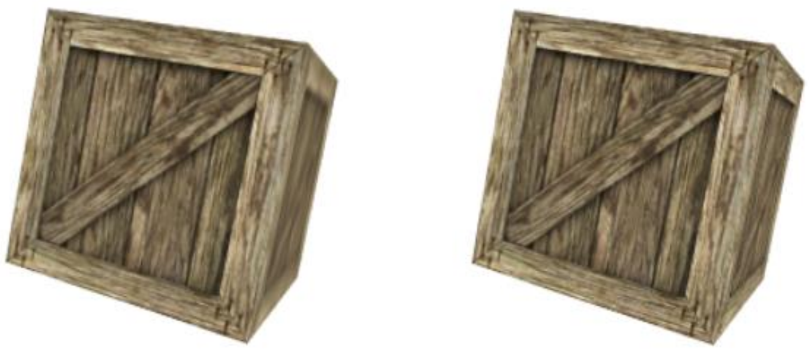
Info

The Mali Offline Compiler can present statistics about functional unit usage, which can help you identify if your important shaders are texture-rate limited. However, please note that the texture cycle counts from the offline compiler assume 1 cycle per texel performance as the compiler does not have visibility of the precise sampler or format which will be used. You will need to manually de-rate the texture performance based on your texture and sampler usage.

Anisotropic sampling performance

The latest generation of Mali GPUs support anisotropic filtering, which enables the texture sampling unit to take into account the orientation of the triangle when determining the sample pattern to use. This improves image quality, in particular for primitives which are viewed at a steep angle with respect to the view plane, at the cost of needing additional samples for a texture operation.

The figure below shows a cube textured as a wooden crate; the image on the left uses traditional trilinear filtering, and the image on the right uses a bilinear filter with 2x anisotropic filtering. Note the improved fidelity of the right-hand face of the cube when anisotropic filtering is used.



From a performance point of view the worst-case cost of anisotropic filtering is one sample per level of maximum anisotropy, which is application controlled, where samples may be bilinear (LINEAR_MIPMAP_NEAREST) or trilinear (LINEAR_MIPMAP_LINEAR). A 2x bilinear anisotropic filter therefore makes up to two bilinear samples. One significant advantage of anisotropic filtering is that actual number of samples made can be dynamically reduced based on the orientation of the primitive under the current sample position.

Do

- Start using a max anisotropy of 2 and assessing whether this gives sufficient quality. Higher numbers of samples will improve quality, but give diminishing returns which are often not worth the performance cost.
- Consider using 2x bilinear anisotropic filtering in preference to isotropic trilinear filtering; it is faster, and has better image quality in regions of high anisotropy. Note that you by switching to bilinear filtering you may see some visible seam at the decision point between mipmap levels.
- Use anisotropic filtering and trilinear filtering selectively for objects that benefit from it the most from a visual point of view.

Don't

- Use higher levels of max anisotropy without reviewing performance; 8x bilinear anisotropic filtering costs 8 times more than a simple bilinear filter.
- Use trilinear anisotropic filtering without reviewing performance; 8x trilinear anisotropic filtering costs 16 times more than a simple bilinear filter.

Impact

- Using 2x bilinear anisotropic filtering in preference to trilinear filtering will increase image quality and often also improves performance.
- Using high levels of max anisotropy may improve image quality but may also significantly degrade performance.

Debugging

- To debug a performance problem with texture filtering, first try disabling anisotropic filtering completely and measuring any improvement. Then incrementally increase the amount of max anisotropy allowed and assess whether the quality is worth the additional performance cost.

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.

Do

For OpenGL ES:

- Set `GL_TEXTURE_WRAP_(S|T|R)` to identical values
  - Note that the OpenGL ES 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 GL_TEXTURE_SWIZZLE_R to GL_RED (default)
- Set GL_TEXTURE_SWIZZLE_G to GL_GREEN (default)
- Set GL_TEXTURE_SWIZZLE_B to GL_BLUE (default)
- Set GL_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 should 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 LOD_CLAMP_NONE
- Set sampler anisotropyEnable to FALSE
- Set sampler maxAnisotropy to 1.0
- Set sampler borderColor to BORDER_COLOR_FLOAT_TRANSPARENT_BLACK
- Set sampler unnormalizedCoordinates to FALSE

... and when populating the VkImageViewCreateInfo structure:

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

It should be noted that the requirements for maximizing descriptor storage conflicts 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 SAMPLER_MIPMAP_MODE_NEAREST, minLod = 0, maxLod = 0.25, and using minFilter = FILTER_LINEAR or FILTER_NEAREST.

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 behavior is to create a unique VkImageView instance which references only the level 0 mipmap and use a VkSampler with pCreateInfo.maxLodsetting to LOD_CLAMP_NONE in accordance with the compact sampler restrictions.

Info

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

Don't

- Set maxLod to the maximum mipmap level in the texture chain; use LOD_CLAMP_NONE

Impact

- Reduced throughput of texture filtering.

sRGB textures

sRGB textures are natively supported in Mali GPU hardware; sampling from, and rendering or blending to an sRGB surface comes at no performance cost. sRGB textures have far better perceptual color resolution than non-gamma corrected formats at same bit depth, so their use is strongly encouraged.

Do

- Use sRGB textures for improved color quality.
- Use sRGB framebuffers for improved color quality.
- Remember that ASTC supports sRGB compression modes for offline compressed textures.

Don't

- Use 16-bit linear formats to gain perceptual color resolution when 8-bit sRGB can suffice.

Impact

- Not using sRGB where appropriate can reduce quality of rendered images.
- Using wider float formats in place of sRGB will increase bandwidth and reduce performance.

AFBC textures

Many recent Mali GPUs support Arm FrameBuffer Compression (AFBC), a lossless image compression format which can be used for compressing framebuffer outputs from the GPU. Use of AFBC is automatic and functionally transparent to the application, but it is useful to be aware of some areas where it cannot be used and may require the driver to insert run-time decompression from AFBC back to an uncompressed pixel format.

Do

- Access textures and images previously rendered by the GPU as framebuffer attachments using the `texture()` access functions in shaders, not `imageLoad()` as this triggers decompression.
- When packing data into color channels – e.g. for a G-buffer – store the most volatile bits in the least significant bits of the channel to get the best compression rates.

Don't

- Use `imageLoad()` or `imageStore()` to read or write into a texture or image previously rendered by the GPU as a framebuffer attachment, as this triggers decompression.

Impact

- Incorrect use can trigger decompression
- Using wider float formats in place of sRGB will increase bandwidth and reduce performance.

Compute

Workgroup sizes

Mali Midgard and Bifrost GPUs have a fixed number of registers available in each shader core and can split those registers across a variable number of threads depending on the register usage requirements of the shader program. The hardware can split up and merge workgroups during shader core resource scheduling unless barriers and/or shared memory are used in which case all threads in the workgroup must concurrently execute in the shader core. Large workgroup sizes restrict the number of registers available to each thread in this scenario, which may force shader programs to make use of stack memory if insufficient registers are available.

Do

- Use 64 as a baseline workgroup size.
- Use a multiple of 4 as a workgroup size.
- Try smaller workgroup sizes before larger ones, especially if using barriers or shared memory.
- When working with images or textures use a square execution dimension – e.g. 8x8 – to exploit optimal 2D cache locality.
- If a workgroup has "per-workgroup" work to be done, consider splitting into two passes to avoid barriers and kernels which contain portions where most threads are idle.
- Measure; compute shader performance is not always intuitive.

Don't

- Use more than 64 threads per workgroup.
- Assume that barriers with small workgroups are free.

Impact

- Large workgroups may starve the shader core for work if a high percentage of threads are waiting on a barrier.
- Shaders which spill to stack will incur higher load/store unit utilization and may also see higher external memory bandwidth.

Shared memory

Mali GPUs do not implement dedicated on-chip shared memory for compute shaders; shared memory is simply system RAM backed by the load-store cache just like any other memory type.

Do

- Use shared memory to share significant computation between threads in a workgroup.
- Keep your shared memory as small as possible, as this reduces chances of thrashing the data cache.
- Reduce precision and data widths to reduce the size of the shared memory needed.

- Remember you need barriers when sharing shared data; a lot of shader code from desktop development will sometimes omit the barrier due to GPU-specific assumptions, but this is not safe!
- Remember that smaller workgroups are less expensive where barriers are concerned.

Don't

- Copy data from global memory to shared memory; this is useless on Mali GPUs and only serves to pollute the caches.
- Use shared memory to implement code like this:

```
if (localInvocationID == 0) {
    common_setup();
}

barrier();

// Per-thread workload here

barrier();

if (localInvocationID == 0) {
    result_reduction();
}
```

  - ... instead partition the problem into three shaders with fewer threads active in the setup and reduction shaders.

Impact

- Impact of using shared memory wrong is very application specific.

Debugging

- Analyzing performance with shared memory is generally very difficult; try out various approaches and check the performance impact.

Image processing

One common use case for compute shaders is image post-processing effects. Fragment shaders have access to many fixed-function features in the hardware which can speed things up, reduce power, and/or reduce bandwidth, so do not throw these advantages away lightly.

Advantages using fragment shading for image processing:

- Texture coordinates are interpolated using fixed function hardware when using varying interpolation, freeing up shader cycles for more useful workloads.
- Writeout to memory can be done via the tile-writeback hardware in parallel to shader code.
- No need to range check `imageStore()` coordinates, which might be a problem when using workgroups which don't subdivide a frame completely.
- Framebuffer compression and Transaction elimination are possible.

Advantages with compute for image processing:

- It can be possible to exploit shared data sets between neighboring pixels, which avoids extra passes, for some algorithms.
- Easier to work with larger working sets per thread, which avoids extra passes for some algorithms.
- For complicated algorithms such as FFTs, which would need multiple fragment render passes, it is often possible to merge into a single compute dispatch.

Do

- Prefer fragment shaders for simple image processing.
- Prefer compute shaders when you are able to be clever (and always measure!).
- Prefer `texture()` over `imageLoad()` for reading read-only texture data; this works well with AFBC textures rendered by previous fragment passes and also load balances the GPU pipelines better because `texture()` operations use the texture unit and `imageLoad\Store()` uses the load/store unit which is often already heavily used in comute shaders for generic memory accesses.

Don't

- Use `imageLoad()` in compute unless you need coherent read and writes within the dispatch.
- Use compute to process images that were produced by fragment shading; this creates a backwards dependency which can cause a bubble. Render passes pipeline more cleanly if render targets produced by fragment shading are consumed by fragment shaders in later render passes.

Impact

- Compute shaders may be slower and less energy efficient than fragment shaders for simple post-processing workloads, such as downscaling, upscaling, and blurs.

Shader code

Minimize precision

In addition to the data bandwidth benefits of using the minimal precision for inputs and outputs, Mali GPUs have full support for reduced precision in the shader core register file and arithmetic units. Using 16-bit precision is normally good enough for computer graphics, especially for fragment shading when computing an output color.

Both ESSL and Vulkan GLSL have support for marking variables and temporaries as having reduced precision with the `mediump` keyword. There is no benefit to using the `lowp` keyword for Mali GPUs; it is functionally identical to `mediump`.

Do

- Use `mediump` when the resultant precision is acceptable.
- Use `mediump` for inputs, outputs, variables, and samplers where possible.
- For angles use a range of `-PI` to `+PI` rather than `0` to `2PI`; this gains a useful bit of precision for `mediump` values because you make use of the floating-point sign bit.

Don't

- Test correctness of `mediump` precision on desktop GPUs; they typically ignore `mediump` and process it as `highp`, so you probably won't see any difference (functional or performance).

Impact

- Using full FP32 precision can affect performance and power efficiency.

Debugging

- Try forcing `mediump` for everything except the contributors to `gl_Position`, to see how much it might help.

Vectorize code

The Mali Utgard and Midgard GPU architectures implement SIMD maths units, exposing vector instructions to each thread of execution. The Mali Bifrost GPU architecture switches to scalar arithmetic instructions, but still implements vector access to memory.

Do

- Write vector arithmetic code in your shaders to get best performance from Mali GPUs. This becomes less critical with the introduction of the Bifrost architecture, but there are lots of Utgard and Midgard architecture devices out in users' hands.
- Write compute shaders so that work items contain enough work to fill the vector processing units.

Don't

- Write scalar code and hope the compiler will optimize it; it usually will, but it is most reliably vectorized if the input code starts out in vector form.

Vectorize memory access

The Mali GPU shader core load/store data cache has a wide data path capable of returning multiple values in a single clock cycle. For shader programs such as compute shaders which expose more direct access to the underlying memory it is important to make vector data accesses to get the highest access bandwidth from the data caches.

Do

- Use vector data types for memory access with a single thread.
- For Bifrost GPUs which run four neighboring threads – called a quad – in lockstep, access overlapping or sequential memory ranges across the four threads to allow load merging.

Don't

- Avoid scalar loads where possible.
- Avoid divergent addresses across the thread quad where possible.

Impact

- Many types of common compute program perform relatively light arithmetic on very large data sets and getting memory access right can have a significant improvement on the overall system.

Manual optimization

Code transforms which are legal in real-world mathematics may not be possible for the shader compiler to safely make because they could cause the appearance of a floating point infinity or a not-a-number (NaN) which would not have been present in the original program ordering, and thus trigger a rendering error. Where possible refactor your source code offline to reduce the number of computations needed, rather than relying on the compiler to apply the optimizations; it may not be able to optimize as much as you think.

Do

- Refactor source code to optimize as much as possible by hand based on your knowledge of values.
- Graphics isn't bit exact; be willing to approximate if it helps refactoring. For example `(A * 0.5) + (B * 0.45)` could be approximated by `(A + B) * 0.5` and save a multiply.
- Use the built-in function library where possible; in many cases it is backed by a hardware implementation which is faster and/or lower power than the equivalent hand-written shader code.

Don't

- Reinvent built-in function library in custom shader code.

Impact

- Reduced performance due to less efficient shader programs.

Debugging

- Use the Mali Offline Compiler to measure the impact of your shader code changes, including analysis of shortest and longest path through the programs.

Instruction caches

The shader core instruction cache is an often-overlooked area which can impact performance, but due to the number of threads running concurrently can be a critically important part to be aware of.

Do

- Prefer shorter shaders with many threads over long shaders with few threads, a shorter program is more likely to be hot in the cache.
- Prefer shaders which don't have control-flow divergence; this reduces temporal locality and increases cache pressure.
- Beware of fragment shading with many visible layers in a tile; shaders for all layers which are not killed by early-zs or FPK must be loaded and executed which can increase cache pressure.

Don't

- Unroll loops too aggressively, although some unrolling can help.
- Generate duplicate shader programs/pipeline binaries from identical source code.

Debugging

- The Mali Offline Compiler can be used to statically determine the sizes of the programs being generated for any given Mali GPU.
- The Mali Graphics Debugger can be used to step through draw calls and visualize how many transparent layers are building up in your render passes.

Uniforms

Mali GPUs can promote data from API-set uniforms and uniform buffers into shader core registers, which are loaded once per draw instead of once every shader thread. This removes a significant number of load operations from the shader programs.

Not all uniforms can be promoted into registers. Uniforms which are dynamically accessed, such as non-constant expression array indices, cannot always be promoted to register mapped uniforms unless the compiler is able to make them constant expressions – e.g. by loop unrolling a fixed iteration `for` loop.

Do

- 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.
- Promote uniforms to compile-time constants with `#defines` (OpenGL ES), specialization constants (Vulkan), or literals in the shader source if they are completely static.
- Avoid uniform vectors and/or matrices which are padded with constant elements which are used in computation but, for example, e.g. are always zero or one.
- Prefer uniforms set by `glUniform*()` (OpenGL ES), or push constants (Vulkan), rather than uniforms loaded from buffers.

Don't

- Dynamically index into uniform arrays.

- Over use instancing; instanced uniforms indexed by `gl_InstanceID` count as dynamically indexed and cannot use register mapped uniforms.

Impact

- Register mapped uniforms are effectively free; any spilling to buffers in memory will increase load/store cache accesses to the per thread uniform fetches.

Debugging

- The Mali Offline Compiler provides statistics about the number of uniform registers used and the number of load/store instructions being generated.

Uniform sub-expressions

One common source of inefficiency is the presence of uniform sub-expressions in the shader source; these are pieces of code which only depend on the value of literals and/or other uniforms, and so produce a result which is identical for all invocations.

Do

- Minimize the number of *uniform-on-uniform* or *uniform-on-literal* computations; compute the final result of the uniform sub-expression on the CPU and upload that as your uniform.

Impact

- The Mali GPU drivers can optimize the cost of most uniform sub-expressions so that they are only computed once per draw, so the benefit isn't as big as it appears, but the optimization pass will still incur a small cost for every draw call which can be avoided by removing the redundancy.

Debugging

- Use the Mali Offline Compiler to measure the impact of your shader code changes, including analysis of shortest and longest path through the programs.

Uniform control-flow

One common source of inefficiency is the presence of conditional control-flow, such as `if` blocks and `for` loops, which are parameterized by uniform expressions.

Do

- Use compile-time `#defines` (OpenGL ES) and specialization constants (Vulkan) for all control flow, this allows compilation to completely remove unused code blocks and statically unroll loops.

Don't

- Use control-flow which is parameterized by uniform values; specialize shaders for each control path needed instead.

Impact

- Reduced performance due to less efficient shader programs.

Debugging

- Use the Mali Offline Compiler to measure the impact of your shader code changes, including analysis of shortest and longest path through the programs.

Branches

Branching is relatively expensive on a GPU because it either restricts how the compiler can pack groups of instructions within a thread or introduces cross-thread scheduling restrictions when there is divergence across multiple threads.

Do

- Minimize the use of complex branches in shader programs.
- Minimize the amount of control-flow divergence in spatially adjacent shader threads.
- Use `min()`/`max()`/`clamp()`/`mix()` functions to avoid small branches.
- Check the benefits of branching over computation – e.g. skipping lights which are above a threshold distance from the camera – in many cases just doing the computation will be faster.

Don't

- Implement multiple expensive data paths which are selected from using a `mix()`; branching is likely the best solution for minimizing the overall cost in this scenario.

Impact

- Reduced performance due to less efficient shader programs.

Debugging

- Use the Mali Offline Compiler to measure the impact of your shader code changes, including analysis of shortest and longest path through the programs.

Discards

Using a `discard` in a fragment shader or using alpha-to-coverage (which is really just an implicit discard), are is commonly used to implement for techniques such as alpha-testing complex shapes for foliage and trees.

These techniques force fragments to use late-zs updates, because it is not known if they will survive the fragment operations stage of the pipeline until after shading because we need to run the shader to determine the discard state of each sample. This can cause either redundant shading and/or pipeline starvation due to pixel dependencies.

Do

- Minimize the use of shader `discard` and alpha-to-coverage.
- Disable depth and stencil writes when using `discard` for culling lighting fragments when resolving the lighting for a deferred shading using precomputed depth buffer.
- Render alpha-tested geometry front-to-back with depth-testing enabled; this will cause as many fragments as possible to fail early-zs testing, minimizing the number of late-zs updates needed.

Impact

- Performance loss or bandwidth increase due to additional fragment shading costs.
- Performance loss due to pipeline starvation waiting for pixel dependencies to resolve.

Atomics

Atomic operations are a staple of many compute (and some fragment) algorithms, which allows many otherwise serial algorithms to be implemented on highly parallel GPUs with some slight modifications. The fundamental performance problem with atomics is contention; atomic operations from different shader cores hitting the same cache line will require data coherency snooping through L2 cache, which is expensive.

Well-tuned compute applications using atomics should aim to spread contention out by keeping the atomic operations local to a single shader core. If a shader core "owns" the necessary cache line in its L1 then atomics will be very efficient.

Do

- Consider how to avoid contention when using atomics in algorithm design.
- Consider spacing atomics 64 bytes apart to avoid multiple atomics contending on the same cache line.
- Consider whether it is possible to amortize the contention by by accumulating into a shared memory atomic, and then have one thread push the global atomic operation at the end of the workgroup.

Don't

- Use atomics if better solutions using multiple passes are available.

Impact

- Heavy contention on a single atomic cache entry dramatically reduces overall throughput and impacts how well problems will scale up when running on a GPU implementation with more shader cores.

Debugging

- The GPU performance counters include counters for monitoring the frequency of L1 cache snoops from the other shader cores in the system.

System integration

EGL buffer preservation

OpenGL ES only

Creating window surfaces configured with EGL_BUFFER_PRESERVED allows applications to logically update only part of the screen, always starting rendering with the framebuffer state from the previous frame rather than rendering from scratch.

On first glance this seems like an efficiency boost if the application only wants to update a subregion of the screen. However, in real systems the use of double or triple buffering means that what really happens is that rendering starts with a full screen blit from the window surface for frame n in to the window surface for frame n+1.

Do

- Use the EGL_KHR_partial_update extension in preference to EGL_BUFFER_PRESERVED to minimize client rendering; it actually allows incremental updates and true partial rendering of a frame.
- Use the EGL_KHR_swap_buffers_with_damage extension in preference toeglSwapBuffers() to minimize server-side composition overheads.
- If these extensions are not available, review the cost of EGL_BUFFER_PRESERVED against the cost of just re-rendering a whole frame. Simple UI content using compressed texture inputs and procedural fills may actually find it more efficient just to re-render the entire frame from the original source material.
- If you know an entire frame is overdrawn when using EGL_BUFFER_PRESERVED insert a full-screen glClear() at the start of the render pass; this will remove the readback of the previous frame into the GPU tile memory.

Don't

- Use EGL_BUFFER_PRESERVED surfaces without considering whether it makes sense for the content involved; always try EGL_BUFFER_DESTROYED and measure the benefits.
- Use EGL_KHR_partial_update or EGL_KHR_swap_buffers_with_damagefor applications which always re-render the whole frame; there is a small additional software cost.

Impact

- Unnecessary readbacks may reduce performance and increase memory bandwidth.

Android blob cache size

OpenGL ES only

The Mali OpenGL ES drivers use the EGL_ANDROID_blob_cache extension to persistently cache the results of shader compilation and linkage. For many applications this means that shaders are only compiled and linked when the application is first used; subsequent application runs will use binaries from the cache, and benefit from faster startup and level load times.

The Android blob cache defaults to a relatively small size (64KB per application), which is insufficient for many modern games and applications which may use hundreds of shaders. We recommend that system integrators significantly increase the maximum size of the blob cache for their platforms, in order to increase the number of applications which benefit from program caching.

Do

- Increase the size of the Android blob cache; for example, to 512KB or 1MB per application

Impact

- Games and applications which exceed the size of the blob cache will start up more slowly, as shader programs must be compiled and linked at runtime.

Swapchain surface count

Vulkan only

For Vulkan the application has control over the window surface swapchain, in particular deciding how many surfaces should be used. Nearly all mobile platforms use the display vsync signal to prevent screen tearing on buffer swap, which means that a swapchain containing only two buffers is prone to stalling the GPU if the GPU is rendering more slowly than the vsync period.

Do

- Use two surfaces in the swapchain if your application will always run faster than vsync; this will reduce your memory consumption.
- Use three surfaces in the swapchain if your application may run more slowly than vsync; this will give you best application performance.

Don't

- Use two surfaces in the swapchain if your application runs more slowly than vsync.

Impact

- Using double buffering and vsync will lock rendering to an integer fraction of the vsync rate, which will reduce performance if the application is rendering slower than vsync. For example, in a system with a 60FPS panel refresh rate, an application which is otherwise capable of running at 50FPS will snap down to 30FPS.

Debugging

- System profilers, such as DS-5 Streamline, can show a when the GPU is going idle. If the GPU is going idle and the frame period is a multiple of the vsync period, then this might be indicative of rendering blocking and waiting for a buffer to be released to be triggered by the vsync signal.

Swapchain surface rotation

Vulkan only

For Vulkan the application has control over the window surface orientation and is responsible for handling differences between the logical orientation of the window – which can vary on portable devices as the device is rotated – and the physical orientation of the display. It is nearly always more efficient for the presentation subsystem if the application to renders into a window surface which is oriented to match the physical orientation of the display panel.

Do

- To avoid presentation engine transformation passes ensure that swapchain `preTransform` matches the `currentTransform` value returned by `vkGetPhysicalDeviceSurfaceCapabilitiesKHR`.
- If a swapchain image acquisition returns `SUBOPTIMAL_KHR` or `ERROR_OUT_OF_DATE_KHR` then recreate the swapchain taking into account any updated surface properties including potential orientation updates reported via `currentTransform`.

Don't

- Assume that supported presentation engines transforms other than `currentTransform` are free; many presentation engines can handle rotation and/or mirroring but at additional processing cost.

Impact

- Non-native orientation may require additional transformation passes in the presentation engine. This may require use of the GPU on some systems which use the GPU as part of the presentation engine to handle cases the display controller cannot handle natively.

Debugging

- System profilers, such as the kernel integrated version of DS-5 Streamline, are able to attribute Mali GPU use to specific processes. This allows attribution of additional GPU workload to the compositor process, assuming that the GPU is being used by the compositor to apply the presentation transformation.

Swapchain semaphores

Vulkan only

A typical frame of Vulkan will look like:

- Create a `VkSemaphore` for the start of frame acquire (#1).
- Create a `VkSemaphore` for the end of frame release (#2).
- Call `vkAcquireNextImage()` gives you the swapchain index #N and associates with semaphore #1.
- Wait for all fences associated with swapchain index #N.
- Build command buffers.
- Submit command buffers rendering to the window surface to `VkQueue` telling them to wait for (#1) before they start rendering, and to signal (#2) when they have completed.
- Call `vkQueuePresent()`, configured to wait for semaphore (#2).

The critical part of this is setting up the wait on (#1) for any command buffers, 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 to wait for the final color buffer using `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 = COLOR_ATTACHMENT_OUTPUT_BIT` when waiting for a semaphore from WSI.

Impact

- Large pipeline bubbles will be created since vertex/compute will stall.

Window buffer alignment

Linear format framebuffers can suffer from dramatically reduced write performance if rows are insufficiently aligned. To ensure efficient write performance memory system allocated surfaces which are imported into the Mali driver should be aligned so that each row in a 16x16 tile can be written as a single AXI burst transaction.

Do

- Align rows in linear format framebuffers to a minimum alignment of either a multiple of 16 pixels or 64-bytes (whichever is smaller); e.g. for RGB565 framebuffers you can use 32-byte alignment, for RGBA8 and RGBA fp16 framebuffers you can use 64-byte alignment.
- Use power-of-two formats such as RGBX8 with a dummy channel in preference to RGB8 in cases where alpha is not required, as they have better memory alignment properties.

Don't

- Use any row alignment other than either a multiple of 16 pixels or 64 bytes.
- Use non-power of two framebuffer formats, such as true 24-bit RGB8.

Impact

- AXI bursts must be aligned to the burst size. Unaligned rows must be broken into multiple smaller AXI bursts to meet this requirement, which will make less efficient use of the memory bus and in many cases will cause the GPU to stall because it exhausts the pool of available transaction IDs.

Footnotes

1. FBO0 is a special case; its render pass ends when `eglSwapBuffers()` is called. ↩