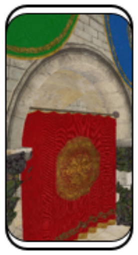
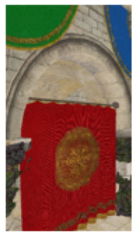
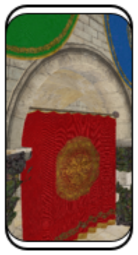
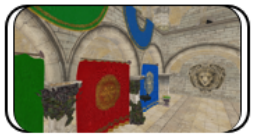
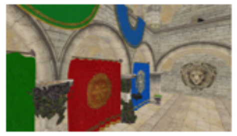
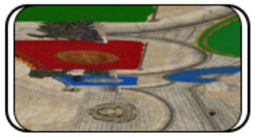
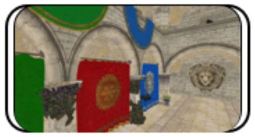
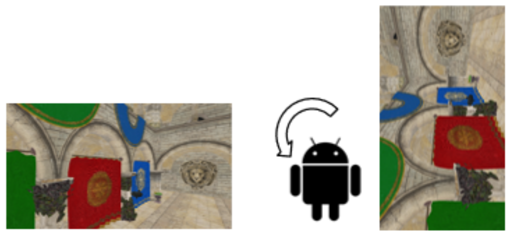
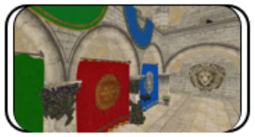
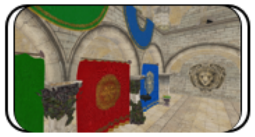
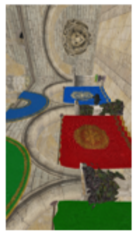
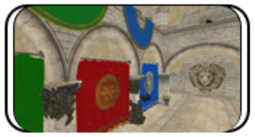
Appropriate use of surface rotation

Overview

Mobile devices can be rotated, therefore the logical orientation of the application window and the physical orientation of the display may not match. Applications then need to be able to operate in two modes: portrait and landscape. The difference between these two modes can be simplified to just a change in resolution. However, some display subsystems always work on the “native” (or “physical”) orientation of the display panel. Since the device has been rotated, to achieve the desired effect the application output must also rotate.

In OpenGL ES the GPU driver can transparently handle the logical rotation of window surface framebuffers, but the Vulkan specification has made this explicit in the API. Therefore in Vulkan the application is responsible for supporting rotation.

In this sample we focus on the rotation step, and analyse the performance implications of implementing it correctly with Vulkan.

Case	Desired outcome	Rendering events	Real outcome
No change in orientation			
Change in orientation, no rotation		 FRAME RENDERED ACCORDING TO CHANGE IN RESOLUTION	
Change in orientation, rotation by compositor		 FRAME RENDERED ACCORDING TO CHANGE IN RESOLUTION; ANDROID COMPOSITOR ROTATES THE OUTPUT	
Change in orientation, rotation by application		 FRAME RENDERED ACCORDING TO CHANGE IN RESOLUTION AND ORIENTATION	

Pre-rotation

The rotation step can be carried out in different ways:

- It can efficiently and transparently be handled in hardware by the Display Processing Unit (DPU), but this is only possible in those devices that support it.
- It can be handled by Android, by introducing a compositor pass that rotates the output using the GPU, or some other dedicated block. This is transparent to the application, but will have additional system-level costs such as extra memory bandwidth, or even GPU processing if the compositor uses the GPU as the rotation engine.
- It can be handled by the Vulkan application, by rendering into a window surface which is oriented to match the physical orientation of the display panel. We call this pre-rotation.

An application has no means to tell whether the current device can support a free rotation during composition, so the only guaranteed method to avoid any additional processing cost is to render into a window surface which is oriented to match the physical orientation of the display panel, thus removing the Android compositor step.

Demo application

The sample application you can find here shows how you can handle rotations in your Vulkan application in a way that avoids using the Android compositor for rotation. It allows you to enable and disable pre-rotation at run time, so you can compare these two modes using the hardware counters shown on the display. In this section we will go through the code required to carry out pre-rotation. In the analysis section below we will explain the differences in more detail. Note that not all devices will show obvious differences, as more and more include a DPU capable of performing the rotation in hardware.

In a nutshell, below are the steps required to handle pre-rotation:

No pre-rotation	Pre-rotation
Destroy the Vulkan framebuffers and the swapchain	Destroy the Vulkan framebuffers and the swapchain

No pre-rotation	Pre-rotation
Re-create the swapchain using the new surface dimensions i.e. the swapchain dimensions match the surface's. Ignore the `preTransform` field in `VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR` . This will not match the value returned by `vkGetPhysicalDeviceSurfaceCapabilitiesKHR` and therefore the Android Compositor will rotate the scene before presenting it to the display	Re-create the swapchain using the old swapchain dimensions, i.e. the swapchain dimensions do not change. Update the `preTransform` field in `VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR` so that it matches the `currentTransform` field of the `VkSurfaceCapabilitiesKHR` returned by the new surface. This communicates to Android that it does not need to rotate the scene.
Re-create the framebuffers	Re-create the framebuffers
n/a	Adjust the MVP matrix so that: 1) The world is rotated, 2) The field-of-view (FOV) is adjusted to the new aspect ratio
Render the scene	Render the scene

Rotation in Android

Android added pre-rotation to their Vulkan Design Guidelines. However, by default, Android calls `onDestroy` when the screen is rotated. To disable this behavior and handle rotations in Vulkan, you must add the `orientation` (for API level 13 and lower) and `screenSize` attributes to the activity's `configChanges` in the Android manifest:

```
<activity android:name=".BPNativeActivity"
          android:configChanges="orientation|screenSize">
```

To track orientation changes, use Android's `APP_CMD_CONTENT_RECT_CHANGED` event:

```
void on_app_cmd(android_app *app, int32_t cmd)
{
        auto platform = reinterpret_cast<AndroidPlatform *>(app->userData);
        assert(platform && "Platform is not valid");
        switch (cmd)
        {
                case APP_CMD_INIT_WINDOW:
                {
                        app->destroyRequested = !platform->get_sample().prepare(*platform);
                        break;
                }
                case APP_CMD_CONTENT_RECT_CHANGED:
                {
                        // Get the new size
                        auto width  = app->contentRect.right - app->contentRect.left;
                        auto height = app->contentRect.bottom - app->contentRect.top;
                        platform->get_sample().resize(width, height);
                        break;
                }
                case APP_CMD_TERM_WINDOW:
                {
                        platform->get_sample().finish();
                        break;
                }
        }
}
```

Swapchain re-creation

When we rotate, first we must safely destroy the framebuffers:

```
vkDeviceWaitIdle(context.device);
destroy_framebuffers(context);
```

Then we need to sample the current transform from the surface:

```
VkSurfaceCapabilitiesKHR surface_properties;
vkGetPhysicalDeviceSurfaceCapabilitiesKHR(context.gpu,
                                          context.surface,
                                                &surface_properties);

uint32_t                      new_width         = surface_properties.currentExtent.width;
uint32_t                      new_height        = surface_properties.currentExtent.height;
VkSurfaceTransformFlagBitsKHR new_pre_transform = surface_properties.currentTransform;
```

We query the new width and the new height, which should just be swapped with respect to the previous orientation. However, when doing pre-rotation we want to preserve the dimensions of the images, as we are planning to rotate our geometry accordingly. `currentTransform` is a `VkSurfaceTransformFlagBitsKHR` value, which we can use to act on if the orientation has changed. When we re-create the swapchain, we must set the swapchain's `preTransform` to match this value. This informs the compositor that the application has handled the required transform so it does not have to.

```
if (new_pre_transform & VK_SURFACE_TRANSFORM_ROTATE_90_BIT_KHR ||
    new_pre_transform & VK_SURFACE_TRANSFORM_ROTATE_270_BIT_KHR)
{
        // Do not change the swapchain dimensions
        new_width  = surface_properties.currentExtent.height;
        new_height = surface_properties.currentExtent.width;
}
vkb::init_swapchain(context, new_width, new_height, new_pre_transform);
```

The `init_swapchain` function can be found in our framework. It uses the new width, height and transformation values, and it recycles the previous swapchain:

```
surface_properties.currentExtent.width  = new_width;
surface_properties.currentExtent.height = new_height;

VkSwapchainCreateInfoKHR info = {VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR};
(...)
info.imageExtent              = surface_properties.currentExtent;
info.preTransform             = new_pre_transform;
info.oldSwapchain             = old_swapchain;

vkCreateSwapchainKHR(context.device, &info, nullptr, &context.swapchain.handle);

if (old_swapchain != VK_NULL_HANDLE)
```

```
{
        vkDestroySwapchainKHR(context.device, old_swapchain, nullptr);
}
```

Finally, we re-create the framebuffers:

```
vkb::init_framebuffers(context);
```

Rotating the scene

When rotating our geometry, normally all we need to do is adjust the Model View Projection (MVP) matrix that we provide to the vertex shader every frame. In this case we want to rotate the scene just before applying the projection transformation:

```
float     aspect_ratio   = width / height;
auto      horizontal_fov = glm::radians(60.0f);
auto      vertical_fov   = static_cast<float>(2 * atan((0.5 * height) / (0.5 * width / tan(horizontal_fov / 2))));
auto      fov            = (aspect_ratio > 1.0f) ? horizontal_fov : vertical_fov;

glm::mat4 pre_rotate_mat = glm::mat4(1.0f);
glm::vec3 rotation_axis  = glm::vec3(0.0f, 0.0f, -1.0f); # Rotate anti-clockwise
if (context.swapchain.pre_transform & VK_SURFACE_TRANSFORM_ROTATE_90_BIT_KHR)
{
        pre_rotate_mat = glm::rotate(pre_rotate_mat, glm::radians(90.0f), rotation_axis);
}
else if (context.swapchain.pre_transform & VK_SURFACE_TRANSFORM_ROTATE_270_BIT_KHR)
{
        pre_rotate_mat = glm::rotate(pre_rotate_mat, glm::radians(270.0f), rotation_axis);
}
else if (context.swapchain.pre_transform & VK_SURFACE_TRANSFORM_ROTATE_180_BIT_KHR)
{
        pre_rotate_mat = glm::rotate(pre_rotate_mat, glm::radians(180.0f), rotation_axis);
}

glm::mat4 proj      = glm::perspective(fov, aspect_ratio, 0.01f, 1000.0f);
glm::mat4 view_proj = vulkan_style_projection(proj) * pre_rotate_mat * view;
```

```
(...)

vkCmdPushConstants(cmd, pipeline_layout->get_handle(), VK_SHADER_STAGE_VERTEX_BIT, sizeof(glm::mat4),
                   sizeof(glm::mat4), glm::value_ptr(view_proj));
```

For completion, here are the relevant sections of the vertex shader:

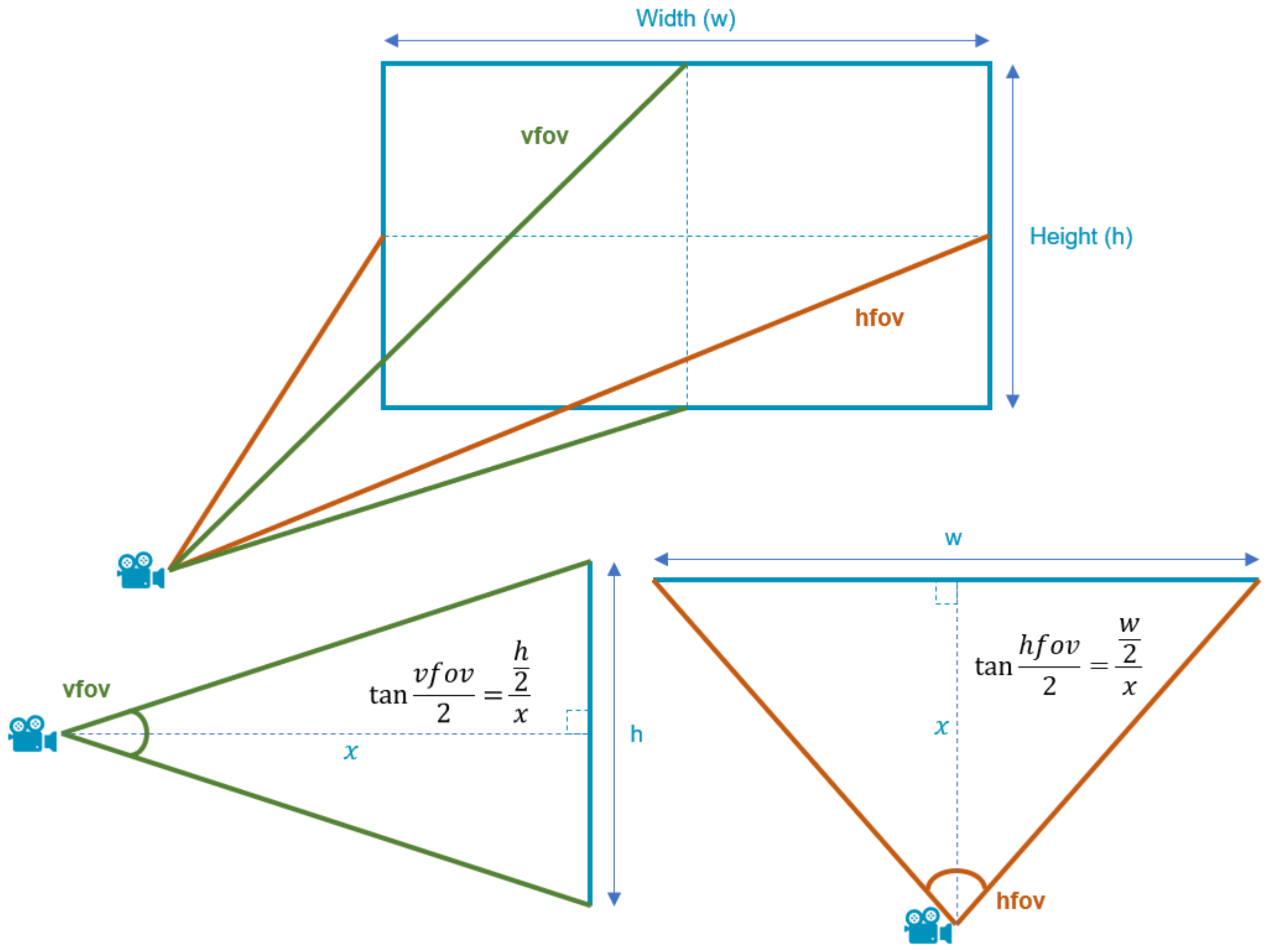
```
layout(location = 0) in vec3 position;

layout(push_constant, std430) uniform PushConstant {
        mat4 model;
        mat4 view_proj;
} vs_push_constant;

layout (location = 0) out vec4 o_pos;

void main(void)
{
        o_pos       = vs_push_constant.model * vec4(position, 1.0);
        gl_Position = vs_push_constant.view_proj * o_pos;
}
```

We can see that, as well as rotating the scene, we also need to adjust the Field-of-view (FOV) used to calculate the projection matrix. The FOV is the extent of the observable world that can be seen, and it can be broken down into a horizontal component and a vertical component. To calculate the projection matrix, we use the aspect ratio, the FOV, a far clipping plane and a near clipping plane. Keeping the other factors constant, increasing the FOV for a given camera position results in a ‘zoom out’ effect. Similarly, decreasing the FOV results in a ‘zoom in’ effect. Therefore, if the aspect ratio changes, in order to keep the same ‘zoom level’, the FOV must be adjusted accordingly. Since rotating the screen is effectively a swap of width and height, having set a particular horizontal FOV in a landscape display means that we need to use the corresponding vertical FOV in a portrait display. The derivation of the formula used can be found below:


Width (w)
vfov
Height (h)
hfov
vfov
$\tan \frac{vfov}{2} = \frac{\frac{h}{2}}{x}$
x
h
w
$\tan \frac{hfov}{2} = \frac{\frac{w}{2}}{x}$
x
hfov

$$aspect\_ratio = \frac{w}{h} = \frac{\tan \frac{hfov}{2}}{\tan \frac{vfov}{2}}$$

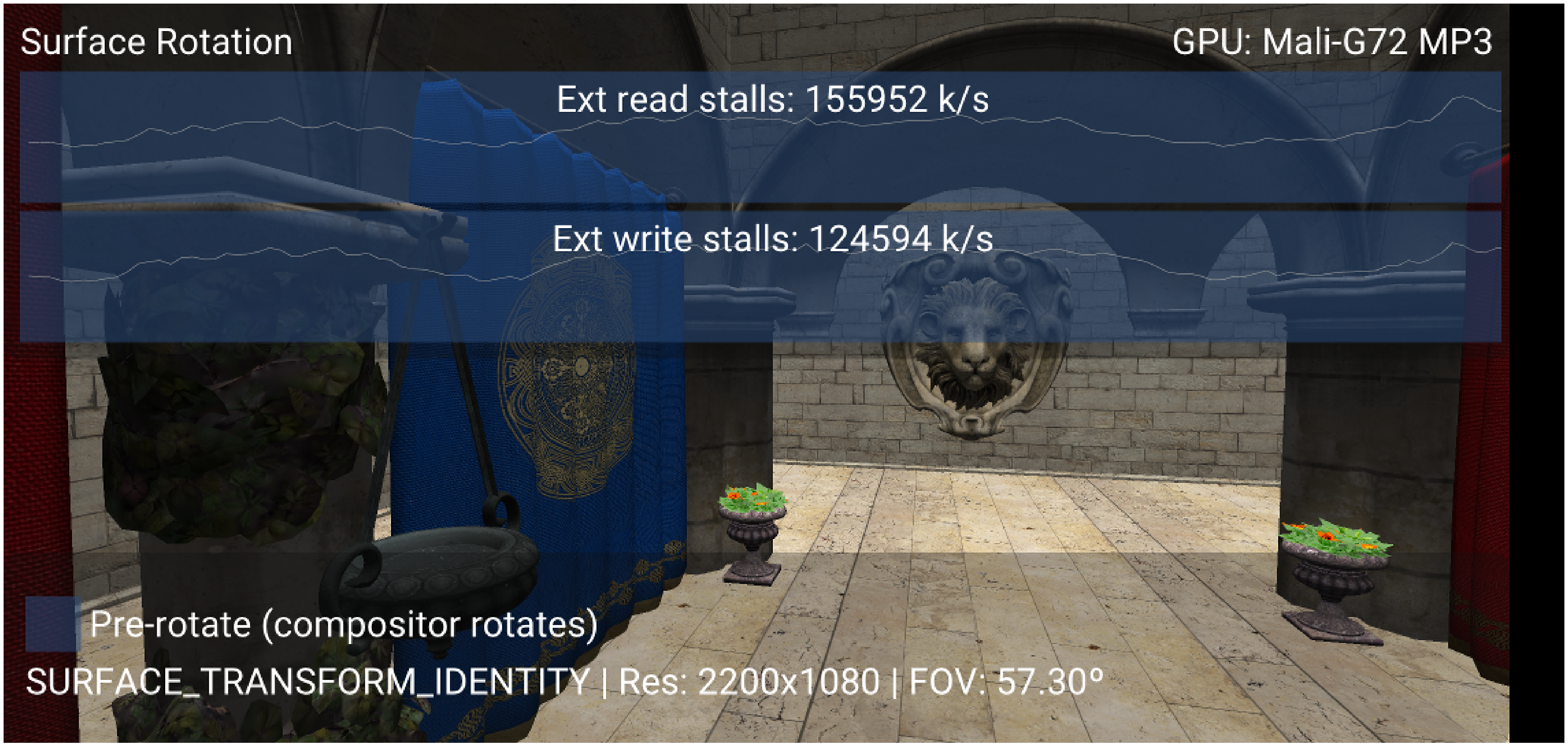
$$w \tan \frac{vfov}{2} = h \tan \frac{hfov}{2}$$

$$\frac{hfov}{2} = \tan^{-1} \frac{w \tan \frac{vfov}{2}}{h}$$

$$hfov = 2 \tan^{-1} \left(\tan \frac{vfov}{2} \times \frac{w}{h} \right)$$

Performance impact

The `surface_rotation` Vulkan sample allows you to toggle between pre-rotation mode and compositor mode. Below is a screenshot of the sample running on a device that does not support native (DPU) rotation, but instead includes a separate 2D block which rotates the GPU output before presenting it to the display.


Surface Rotation

GPU: Mali-G72 MP3

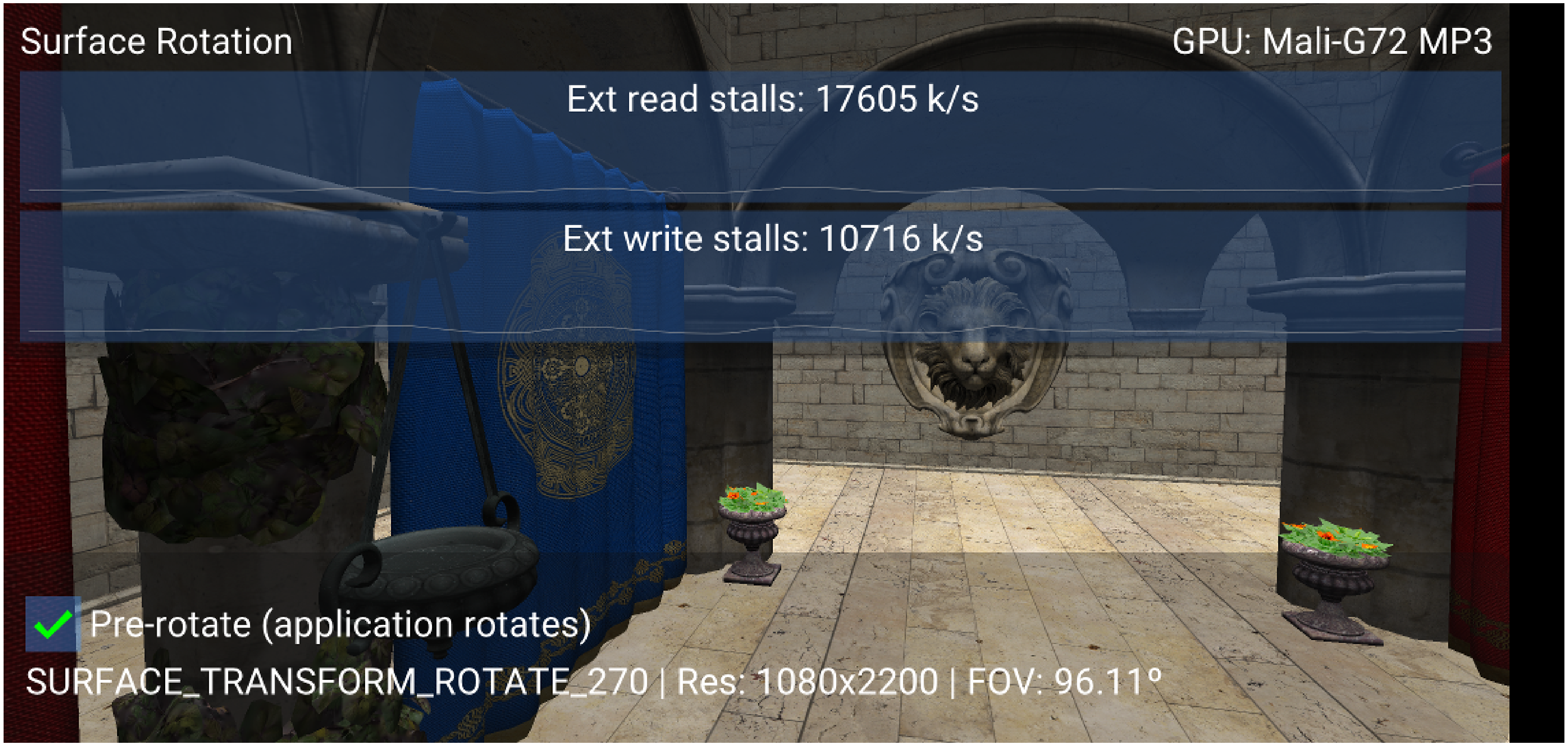
Ext read stalls: 155952 k/s

Ext write stalls: 124594 k/s

Pre-rotate (compositor rotates)

SURFACE_TRANSFORM_IDENTITY | Res: 2200x1080 | FOV: 57.30°

Compare this to the same scene rendered using pre-rotation:


Surface Rotation

GPU: Mali-G72 MP3

Ext read stalls: 17605 k/s

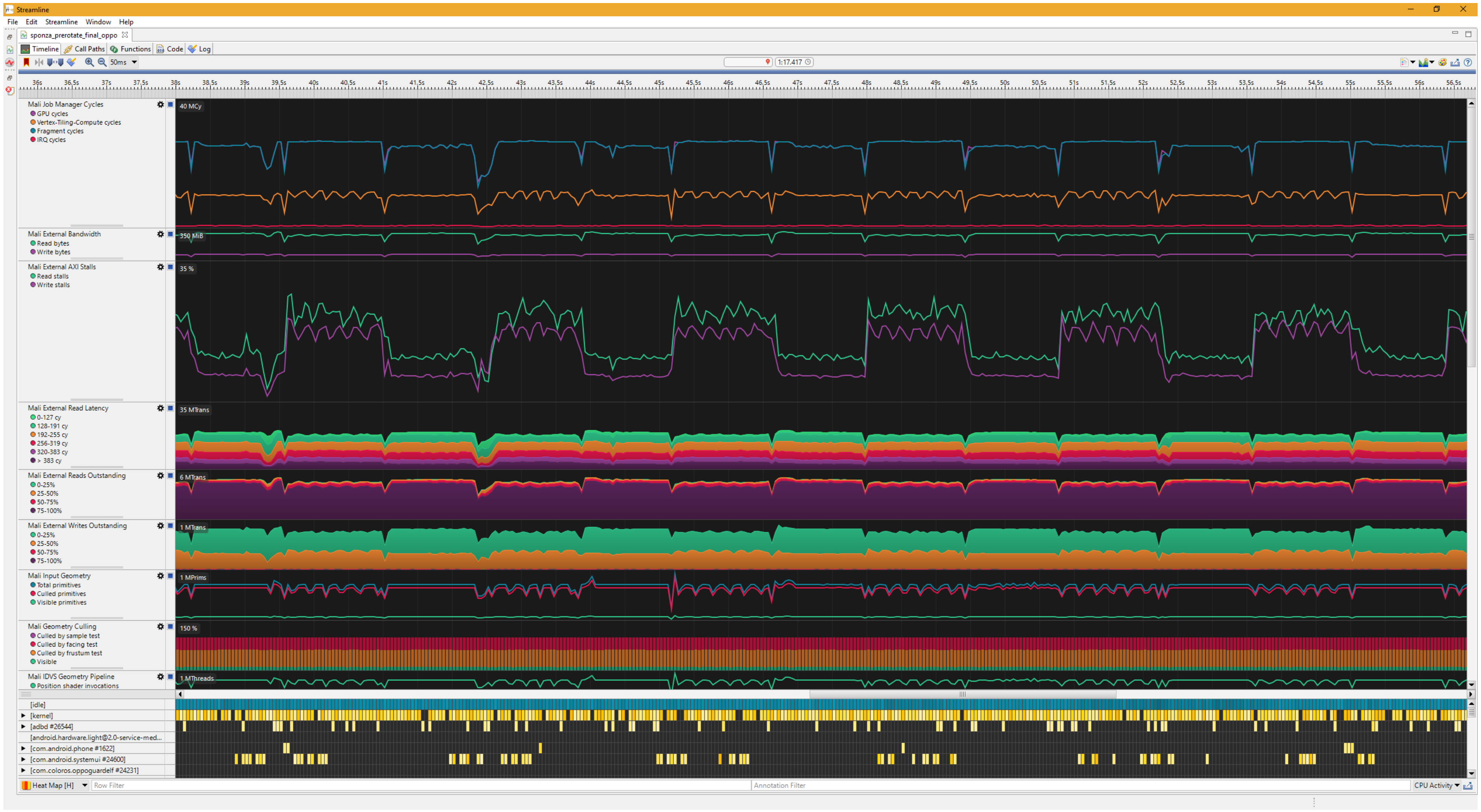
Ext write stalls: 10716 k/s

✓ Pre-rotate (application rotates)

SURFACE_TRANSFORM_ROTATE_270 | Res: 1080x2200 | FOV: 96.11º

As you can see there is a significant increase in the stall rate on the external memory bus if pre-rotation is not enabled, because the framebuffer is being read and written to the 2D rotation block. For this device the additional system memory bandwidth generated by the 2D block increases the use of external memory, which is visible as an increase in memory back-pressure seen by the GPU.

This is more obvious if we trace both modes using Streamline. If you enable all Mali counters and use the relevant template (Mali-G72 in this case) to visualize the data, we can see that we go from an average 12% read stall / 7% write stall to 22% read stall / 17% write stall. In the image below pre-rotation is enabled and disabled every second (using the auto-toggle option). The absolute traffic per cycle drops, but this is because of the drop in performance associated to the increased memory pressure.



In this case the 2D rotation block is using a significant portion of the bandwidth, causing a drop in performance. Note however that this scene is rendered in a memory-heavy fashion (no culling, no compressed textures) to make the effect of pre-rotation more visible. Even if your scene is not memory-heavy, the extra load on the system resulting from performing the rotation during composition will have a negative impact on the battery life of the device.

In order to save battery life in those devices without a rotation-capable DPU, always ensure that your Vulkan renderer performs pre-rotation.

Best-practice summary

Do

- To avoid presentation engine transformation passes ensure that swapchain `preTransform` matches the `currentTransform` value returned by `vkGetPhysicalDeviceSurfaceCapabilitiesKHR` .
- If a swapchain image acquisition returns `VK_SUBOPTIMAL_KHR` or `VK_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 engine’s transforms other than `currentTransform` are free; many presentation engines can handle rotation and/or mirroring but at additional processing cost. Note that Android will always return all transforms as supported, because the GPU is always available as a general purpose fallback.

Impact

- Non-native orientation may require additional transformation passes in the presentation engine. This may require use of the GPU or a dedicated 2D block on some systems which cannot handle the transformation directly in the display controller.

Debugging

- You may use a system profiler such as Streamline to spot extra memory loads in the GPU counters, either as a direct effect (GPU composition) or as a side-effect (memory pressure).