TP08
Vulkan synchronisation

Robin Faury

02/19/20

Abstract

In this practical work, we will see how to run two compute shader with a write/read safe. There is two kind of synchronisation. The memory and the thread synchronisation. For the first one we will wait the end of the first compute shader before reading the buffer filled. For the other one we will duplicate the buffer to avoid write/read when we will want to read neighborhood.

1 Post effect - Linear blur

We want for some reason blur our outputed image. Modify your shader to apply this kernel at the end of the fractal generation. Out of image border will be ignored.

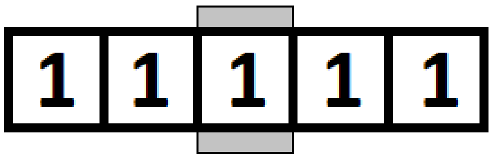

Figure 1: Kernel for linear blur

The output may be good but you can see artefacts like those:

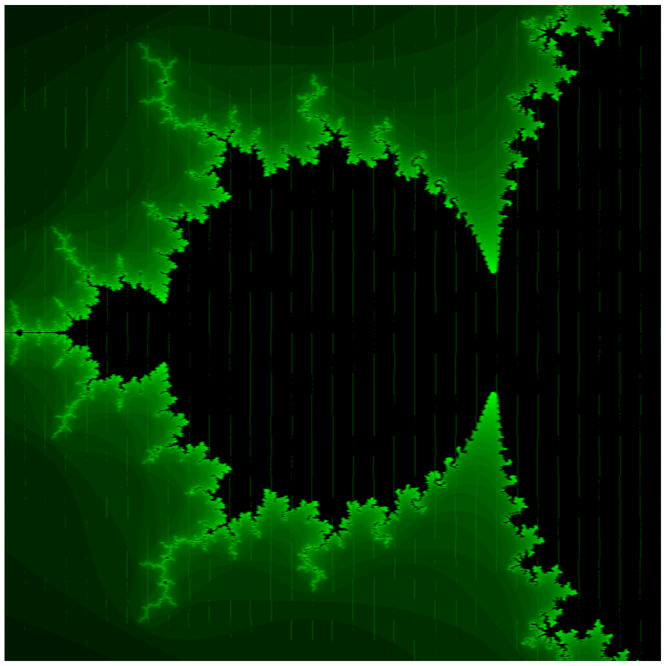

Figure 2: Mandelbrot set computed on GPU

This happen because we read some date while there are computed. We need to wait each shader invocations before reading the neighborhood.

2 synchronisation

We saw having one shader with a write and read behaviours cause artefact. So we need two compute shaders:

- The fractal generation
- The post effect

Create another compute shader to apply the post effect, its VkShaderModule and its VkPipeline. Fortunately, we can reuse all other object needed for the creation of the shader module and the pipeline because any layout has been changed. During the recording of command, simply bind the new pipeline, its descriptor set, its push constants and call its dispatch in the same way that the previous one. On the Vulkan specification, it's said that commands is executed in order, but they might finish out of order. That mean if your first shader is slow you may still having artefact.

2.1 Memory Barrier

In this section we will see how to use memory synchronisation. Barrier is the Vulkan system used for synchronisation. Right after the first dispatch we will create a memory barrier and we will add it to the command buffer:

```
VkMemoryBarrier memoryBarrier = {};
memoryBarrier.sType = VK_STRUCTURE_TYPE_MEMORY_BARRIER;
memoryBarrier.srcAccessMask = VK_ACCESS_SHADER_WRITE_BIT;
memoryBarrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;

vkCmdPipelineBarrier(
        commandBuffer,
        VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT,
        VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT,
        0,
        1,
        &memoryBarrier,
        0, nullptr, 0, nullptr);
```

This barrier force a synchronisation between the two compute shaders. For your information, inside a big pipeline a lot of optimisation can be done with barrier.
To test this barrier use a very simple post effect like invert the red and the green. Without the barrier you may try to apply a post effect on a pixel not computed yet.

2.2 In Out Buffer

For the linear blur we need to read neighborhood pixel. We saw how to write and read on the same buffer using two shader, but here we should read and write on the same shader. We need a thread synchronisation.

1. read neighborhood for each shader invocation
2. wait each shader invocation finnished
3. write the current pixel

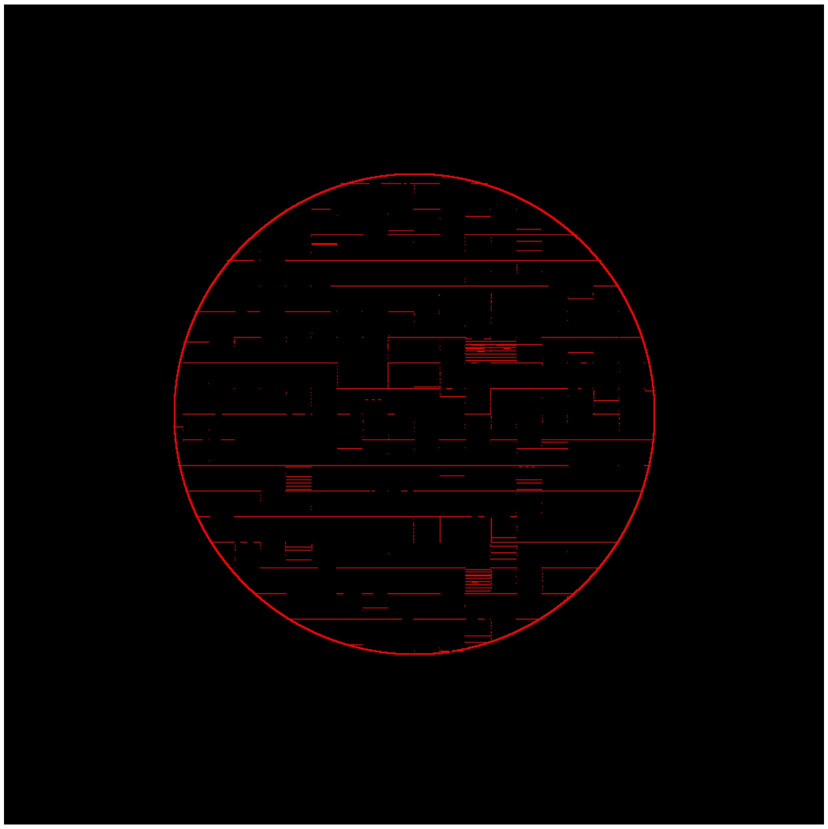

Figure 3: Sobel image processing without thread synchronisation

Unfortunately, Vulkan don't allow us to do that. The solution is to use two buffers. Create another GPUBuffer called GPUBufferOut with the same parameters as GPUBuffer. Modify the layout of your shader to bind two buffer on your compute shader. You can now read any pixel from the first buffer and populate the second.

3 Image processing

We can now create an image processing libriary. Create a shader to generate a circle and another one to apply the sobel filter.

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \qquad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

Figure 4: Sobel kernel

For each pixel apply this formula to get edge detection: $G = abs(G_x) + abs(G_y)$