TP06
Run a compute shader

Robin Faury

02/05/20

Abstract

In this practical work, we will see how bind our GPU buffer and apply our shader on it.

1 Descriptor set

The descriptor set is the handle to store binding information before a draw call. Those information are described in a layout. In the same way that command buffer, descriptor set cannot be directly create. They need a descriptor pool.

1.1 Descriptor set layout

On our shader we declare a buffer as input bound at the location 0. In this section, we'll see how to link out GPU buffer to the shader. We need to create a VkDescriptorSetLayoutBinding to describe where the buffer is stored and for what purpose.

```
VkDescriptorSetLayoutBinding descriptorSetLayoutBinding = {};
descriptorSetLayoutBinding.binding = 0;
descriptorSetLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
descriptorSetLayoutBinding.descriptorCount = 1;
descriptorSetLayoutBinding.stageFlags =VK_SHADER_STAGE_COMPUTE_BIT;
descriptorSetLayoutBinding.pImmutableSamplers = nullptr;
```

We can now create the VkDescriptorSetLayoutCreateInfo with its sType set to VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO. This object just need the pointer to the descriptorSetLayoutBinding and the number of descriptor (1 for our case). If you need more buffer as input in your shader, you just have to add extra VkDescriptorSetLayoutBinding and increase the number of bindingCount. As a suggestion, you can store you VkDescriptorSetLayoutBinding in a vector.
The last step is to call the vkCreateDescriptorSetLayout to get the handle of the VkDescriptorSetLayout. As usual you should delete your handle at the end of your program using vkDestroyDescriptorSetLayout.

1.2 Descriptor pool

On the TP04, we saw that we need a command pool to allocate a command buffer. In this case, we will use a VkDescriptorPool to allocate a VkDescriptorSet. First we need a VkDescriptorPoolSize object to describe the size of the pool. In our case the type is still VK_DESCRIPTOR_TYPE_STORAGE_BUFFER and the descriptorCount is 1. We can use this object as parameter of the VkDescriptorPoolCreateInfo.

```
VkDescriptorPoolCreateInfo descriptorPoolCreateInfo = {};
descriptorPoolCreateInfo.sType  = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
descriptorPoolCreateInfo.poolSizeCount = 1;
descriptorPoolCreateInfo.pPoolSizes = &descriptorPoolSize;
descriptorPoolCreateInfo.maxSets = 1;
```

Finally call vkCreateDescriptorPool for the pool creation. As usual throw an error if the result isn't VK_SUCCESS and destroy the handle at the end of your application using vkDestroyDescriptorPool.

1.3 Allocate the descriptor set

Destroying a pool implicitly frees all objects allocated from that pool. Specifically destroying VkDescriptorPool frees all VkDescriptorSet objects that were allocated from it. To allocate a descriptor set we need an allocator info:

```
VkDescriptorSetAllocateInfo descriptorSetAllocateInfo = {};
descriptorSetAllocateInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
descriptorSetAllocateInfo.descriptorPool = descriptorPool;
descriptorSetAllocateInfo.descriptorSetCount = 1;
descriptorSetAllocateInfo.pSetLayouts = &descriptorSetLayout;
```

Call the vkAllocateDescriptorSets to allocate your descriptor set. Don't forget to check the result function.

1.4 Fill the descriptor set

The last step is to give to our descriptor set the VkBuffer and its parameters. Create a VkDescriptorBufferInfo and fill it with the handle of your GPU buffer. We want to use all data of the buffer. To do that, you need to put 0 in the offset and bufferSize in the range.
To write into a descriptor set we need a VkWriteDescriptorSet:

```
VkWriteDescriptorSet writeDescriptorSet = {};
writeDescriptorSet.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
writeDescriptorSet.dstSet = descriptorSets;
writeDescriptorSet.dstBinding = 0;
writeDescriptorSet.dstArrayElement = 0;
writeDescriptorSet.descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
writeDescriptorSet.descriptorCount = 1;
writeDescriptorSet.pBufferInfo = &descriptorBufferInfo;
```

Finally, just update your descriptor set:

```
vkUpdateDescriptorSets(logicalDevice, 1, &writeDescriptorSet, 0, nullptr);
```

2 Compute pipeline

The pipeline is the mix of the descriptor and the shader module. We will first create the VkPipelineLayout using vkCreatePipelineLayout and this struct:

```
VkPipelineLayoutCreateInfo pipelineLayoutCreateInfo = {};
pipelineLayoutCreateInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
pipelineLayoutCreateInfo.setLayoutCount = 1;
pipelineLayoutCreateInfo.pSetLayouts = &descriptorSetLayout;
```

This pipeline should allow us to pass constants to the shader. On the shader we define bufferSize as an unsigned int. Create a VkPushConstantRange object to store the constant information.

```
VkPushConstantRange pushConstantRange = {};
pushConstantRange.stageFlags = VK_SHADER_STAGE_COMPUTE_BIT;
pushConstantRange.offset = 0;
pushConstantRange.size = sizeof(unsigned int);
```

And set the object to the pipelineLayoutCreateInfo:

```
pipelineLayoutCreateInfo.pushConstantRangeCount = 1;
pipelineLayoutCreateInfo.pPushConstantRanges = &pushConstantRange;
```

Create the VkPipelineLayout using vkCreatePipelineLayout and don't forget to destroy the handle of the pipeline layout at the end.
Next, we can create the structure relative to the shader module.

```
VkPipelineShaderStageCreateInfo pipelineShaderStageCreateInfo = {};
pipelineShaderStageCreateInfo.sType =
        VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
pipelineShaderStageCreateInfo.stage = VK_SHADER_STAGE_COMPUTE_BIT;
pipelineShaderStageCreateInfo.module = shaderModule;
pipelineShaderStageCreateInfo.pName = "main";
```

Finally, we can create our compute pipeline using the vkCreateComputePipelines function and its VkComputePipelineCreateInfo struct input (sType = VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO). Fill the structure with pipelineShaderStageCreateInfo and pipelineLayout.

```
VkComputePipelineCreateInfo computePipelineCreateInfo = {};
computePipelineCreateInfo.sType = VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO;
computePipelineCreateInfo.stage = pipelineShaderStageCreateInfo;
computePipelineCreateInfo.layout = pipelineLayout;

VkPipeline pipeline;
vkCreateComputePipelines(logicalDevice, VK_NULL_HANDLE, 1, &computePipelineCreateInfo
```

3 Command buffer

On the TP04, we saw how to create a command pool and allocate command buffer dedicated for buffer transfer. Here, we will make a similar work. Create a command pool object with the default parameters. Allocate a command buffer and create a VkCommandBufferBeginInfo in the same way as the TP04. We can now fill the command buffer:

```
vkBeginCommandBuffer(commandBuffer, &commandBufferBeginInfo);
vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, pipeline);
vkCmdBindDescriptorSets(
        commandBuffer,
        VK_PIPELINE_BIND_POINT_COMPUTE,
        pipelineLayout,
        0, 1,
        {descriptorSet},
        0, nullptr);
vkCmdPushConstants(
        commandBuffer,
        pipelineLayout,
        VK_SHADER_STAGE_COMPUTE_BIT,
        0,
        uint32_t(sizeof(unsigned int)),
        &bufferSize);
vkCmdDispatch(commandBuffer, bufferSize, 1, 1);
vkEndCommandBuffer(commandBuffer);
```

4 Run the Command buffer

Now we finally have all what we need to run our process. The GPU buffer handle and its data are on the GPU memory, the shader is store on the GPU memory, we have the layout description of how meta data is stored and we have the command buffer to describe the sequence of our process.

4.1 Fence

Fences are a synchronization primitive. We can add a fence to the submit function to track the statue of the process.

```
VkFenceCreateInfo fenceCreateInfo = {};
fenceCreateInfo.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
```

```
vkCreateFence(logicalDevice, fenceCreateInfo, nullptr, &fence);
```

4.2 Submit and wait fence

We can now submit it into the queue. Create a VkSubmitInfo with all members set to zero or nullptr, but commandBufferCount and pCommandBuffers.

```
vkQueueSubmit(queue, 1, {submitInfo}, fence);
```

We need to wait the end of the process. During the copyBuffer we used the waitIdle function. That means we are waiting until the GPU is inactive. In this case, we will wait until the fence it hit.

```
vkWaitForFences(logicalDevice, 1, {fence}, true, uint64_t(-1));
```

The last value is the timeout. -1 means no timeout. After that, we can destroy the fence using vkDestroyFence.

5 Result

You can now use your getData function to transfer the GPU buffer filled by the shader to the stage buffer and finaly to the CPU and save it as a BMP.

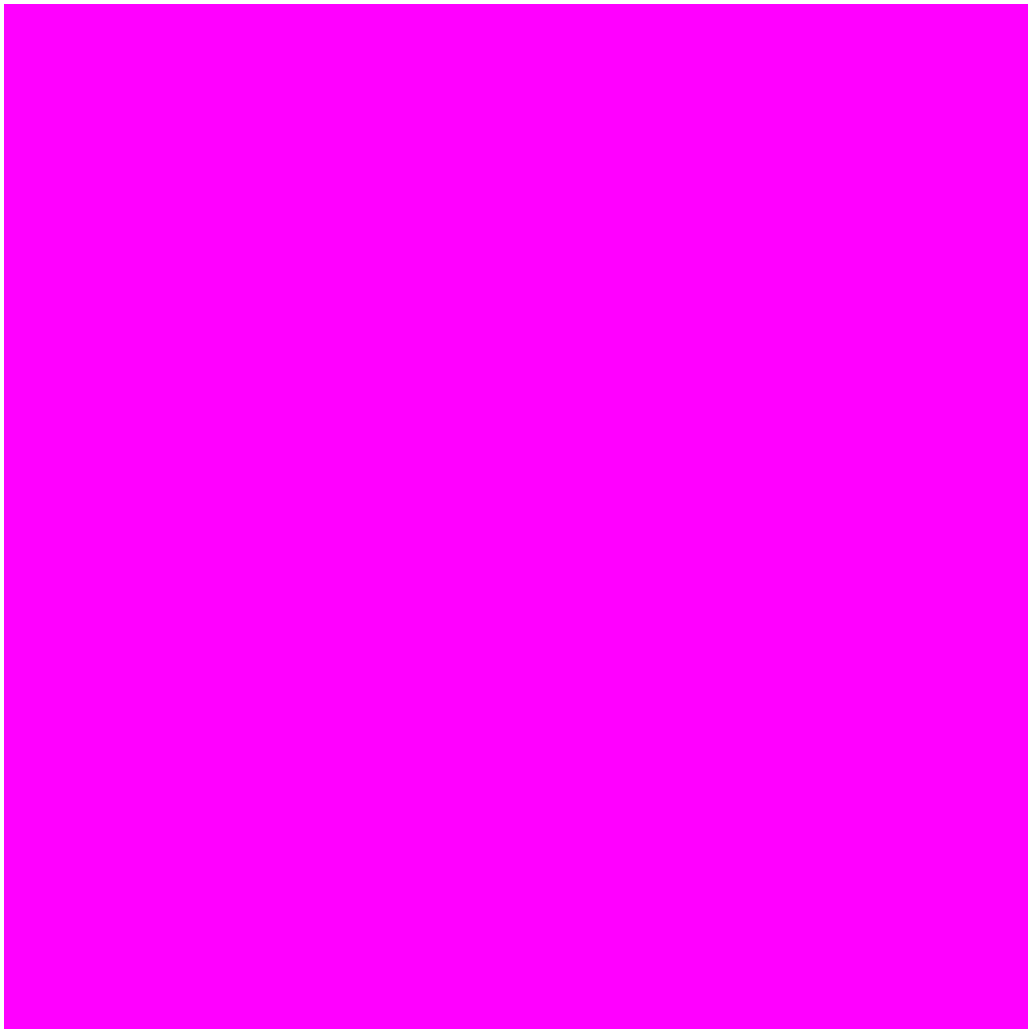

Figure 1: First image computed by Vulkan

6 Validation layer

Vulkan can create object with the given create info but he don't check if your step is consistent. We can add a validation layer to check that. This step can slow down you application. You must not use it in release.
Create a global variable to store the name of the validation layer.

```
const char * const validationLayer = "VK_LAYER_KHRONOS_validation";
```

On the create info of your instance set the enabledLayerCount to 1 and pass the address of the validationLayer into ppEnabledLayerNames. During the execution of your program, all unconsistants error will be printed.