Compute Shaders

Here is how you create a much-simpler Compute Pipeline

Start with Creating the Data Buffers

This is a Particle System application, so we need Positions, Velocities, and (possibly) Colors

```cpp
layout( std140, set = 0, binding = 0 ) buffer Pos
{
    vec4 Positions[ ]; // array of structures
};

layout( std140, set = 0, binding = 1 ) buffer Vel
{
    vec4 Velocities[ ]; // array of structures
};

layout( std140, set = 0, binding = 2 ) buffer Col
{
    vec4 Colors[ ]; // array of structures
};
```

You can use the empty brackets, but only on the last element of the buffer. The actual dimension will be determined for you when OpenGL examines the size of this buffer’s data store.
A Reminder about Data Buffers

Creating a Shader Storage Buffer

Vulkan: Allocating Memory for a Buffer, Binding a Buffer to Memory, and Writing to the Buffer

Fill the Data Buffer
And, since we have Data Buffers, we will need Descriptor Sets to Create the Pipeline Layout

![Diagram of Descriptor Set Layouts and Push Constants]

Create the Compute Pipeline Layout

```cpp
VkDescriptorSetLayoutCreateInfo vdslc;
vdslc.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
vdslc.pNext = nullptr;
vdslc.flags = 0;
vdslc.bindingCount = 1;
vdslc.pBindings = &ComputeSet[0];
result = vkCreateDescriptorSetLayout( LogicalDevice, &vdslc, PALLOCATOR, OUT &ComputeSetLayout );

VkPipelineLayoutCreateInfo vplci;
vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
vplci.pNext = nullptr;
vplci.flags = 0;
vplci.setLayoutCount = 1;
vplci.pSetLayouts = ComputeSetLayout;
vplci.pushConstantRangeCount = 0;
vplci.pPushConstantRanges = (VkPushConstantRange *)nullptr;
result = vkCreatePipelineLayout( LogicalDevice, IN &vplci, PALLOCATOR, OUT &ComputePipelineLayout );
```

Create the Compute Pipeline

```cpp
VkPipelineShaderStageCreateInfo vpssci;
vpssci.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
vpssci.pNext = nullptr;
vpssci.flags = 0;
vpssci.stage = VK_SHADER_STAGE_COMPUTE_BIT;
vpssci.module = computeShader;
vpssci.pName = "main";
vpssci.pSpecializationInfo = (VkSpecializationInfo *)nullptr;

VkComputePipelineCreateInfo vcpci[1];
vcpci[0].sType = VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO;
vcpci[0].pNext = nullptr;
vcpci[0].flags = 0;
vcpci[0].stage = vpssci;
vcpci[0].layout = ComputePipelineLayout;
vcpci[0].basePipelineHandle = VK_NULL_HANDLE;
vcpci[0].basePipelineIndex = 0;
result = vkCreateComputePipelines( LogicalDevice, VK_NULL_HANDLE, 1, &vcpci[0], PALLOCATOR, &ComputePipeline );
```

The Particle System Compute Shader -- Setup

```cpp
#version 430
#extension GL_ARB_compute_shader : enable
layout( std140, set = 0, binding = 0 ) buffer Pos {
  vec4 Positions[ ]; // array of structures
};
layout( std140, set = 0, binding = 1 ) buffer Vel {
  vec4 Velocities[ ]; // array of structures
};
layout( std140, set = 0, binding = 2 ) buffer Col {
  vec4 Colors[ ]; // array of structures
};
layout( local_size_x = 64, local_size_y = 1, local_size_z = 1 ) in;
```
#define POINT vec3
#define VELOCITY vec3
#define VECTOR vec3
#define SPHERE vec4

const VECTOR G = VECTOR(0., -9.8, 0.);
const float DT = 0.1;
const SPHERE Sphere = vec4(-100., -800., 0., 600.);

uint gid = gl_GlobalInvocationID.x; // the .y and .z are both 1 in this case

POINT p = Positions[gid].xyz;
VELOCITY v = Velocities[gid].xyz;

POINT pp = p + v*DT + 0.5*DT*DT*G;
VELOCITY vp = v + G*DT;

Positions[gid].xyz = pp;
Velocities[gid].xyz = vp;

The Particle System Compute Shader – The Physics

VELOCITY Bounce( VELOCITY vin, VECTOR n )
{
    VELOCITY vout = reflect( vin, n );
    return vout;
}

VELOCITY BounceSphere( POINT p, VELOCITY v, SPHERE s )
{
    VECTOR n = normalize( p - s.xyz );
    return Bounce( v, n );
}

bool IsInsideSphere( POINT p, SPHERE s )
{
    float r = length( p - s.xyz );
    return ( r < s.w );
}

The Particle System Compute Shader – How About Introducing a Bounce?

 Dispatching the Compute Shader from the Command Buffer

const int NUM_PARTICLES = 100000;
const int NUM_WORK_ITEMS = 64;
const int NUM_WORK_GROUPS = NUM_PARTICLES / NUM_WORK_ITEMS;

vkCmdBindPipeline( CommandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, ComputePipeline );
vkCmdDispatch( CommandBuffer, NUM_WORK_GROUPS, 1, 1 );

Or,
vkCmdBindPipeline( CommandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, ComputePipeline );
vkCmdDispatchIndirect( CommandBuffer, Buffer, 0 );
The Bouncing Particle System Compute Shader – What Does It Look Like?

- A Specialization Constant is a way of injecting an integer or Boolean constant into an .spv-compiled version of a shader right before the final compilation.
- That final compilation happens when you call `vkCreateComputePipelines()`.
- Without Specialization Constants, you would have to commit to a final value before the SPIR-V compile was done, which could have been a long time ago.

Remember the Compute Pipeline?

- `VkShaderModule`:
  - `vkCreatePipelineLayout()`.
  - `vkCreateComputePipelines()`.

- `Shader Source`:
  - `SPIR-V Compile`.
  - `.spv File`.
  - `Specialization Constants`.
  - `Pipeline Shader Stage`.
  - `Final Compile`.

Specialization Constants

- `VulkanPipelineShaderStageCreateInfo`.
- `VkSpecializationMapEntry`:
  - `vsme.constantID = 0;` // one array element for each Specialization Constant
  - `vsme.offset = 0;` // # bytes into the Specialization Constant
  - `vsme.size = sizeof(int);` // size of just this Specialization Constant

In the compute shader:

```
layout( constant_id = 0 ) const int numXworkitems = 32;
layout( local_size_x = numXworkitems, local_size_y = 1, local_size_z = 1 ) in;
```

In the C/C++ program:

```
VkSpecializationMapEntry vsme[1]; // one array element for each Specialization Constant
vsme.constantID = 0; // # bytes into the Specialization Constant
vsme.offset = 0; // array this one item is
vsme.size = sizeof(int); // size of just this Specialization Constant
```

```
int numXworkitems = 64;
```

```
VkSpecializationInfo vsi;
vsii.mapEntryCount = 1;
vsii.pMapEntries = &vsme[0];
vsii.pData = &numXworkitems; // array of all the Specialization Constants
```
Linking the Specialization Constants into the Compute Pipeline

```c
VkSpecializationMapEntry vsme[1];
vsme.constantID = 0;
vsme.offset = 0;
vsme.size = sizeof(int);

int numXworkItems = 64;

VkSpecializationInfo vsi;
vsii.mapEntryCount = 1;
vsii.pMapEntries = &vsme[0];
vsii.dataSize = sizeof(int);
vsii.pData = &numXworkItems;

VkPipelineShaderStageCreateInfo vpssci;
vpssci.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
vpssci.pNext = nullptr;
vpssci.flags = 0;
vpssci.stage = VK_SHADER_STAGE_COMPUTE_BIT;
vpssci.module = computeShader;
vpssci.pName = "main";
vpssci.pSpecializationInfo = &vsi;

VkComputePipelineCreateInfo vcpci[1];
vcpci[0].sType = VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO;
vcpci[0].pNext = nullptr;
vcpci[0].flags = 0;
vcpci[0].stage = vpssci;
vcpci[0].layout = ComputePipelineLayout;
vcpci[0].basePipelineHandle = VK_NULL_HANDLE;
vcpci[0].basePipelineIndex = 0;

result = vkCreateComputePipelines(LogicalDevice, VK_NULL_HANDLE, 1, &vcpci[0], PALLOCATOR, &ComputePipeline);
```

"Linking the Specialization Constants into the Compute Pipeline"