The Data in your program will look like this:

```
#define NUM_PARTICLES (1024*1024) // total number of particles to move
#define NUM_WORK_ITEMS_PER_GROUP 64 // # work-items per work-group
#define NUM_X_WORK_GROUPS (NUM_PARTICLES / NUM_WORK_ITEMS_PER_GROUP)

struct pos {
    glm::vec4; // positions
};

struct vel {
    glm::vec4; // velocities
};

struct col {
    glm::vec4; // colors
};
```

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

You can use the empty brackets, but only on the last element of the buffer. The actual dimension will be determined for you when Vulkan examines the size of this buffer's data store.
Remember the Graphics Pipeline Data Structure?

Here is how you create a Compute Pipeline Data Structure

A Reminder about Data Buffers

Creating a Shader Storage Buffer
Allocating Memory for a Buffer, Binding a Buffer to Memory, and Filling the Buffer

```cpp
VkMemoryRequirements vmr;
result = vkGetBufferMemoryRequirements(LogicalDevice, PosBuffer, OUT &vmr);

VkMemoryAllocateInfo vmai;
vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
vmai.pNext = nullptr;
vmai.flags = 0;
vmai.allocationSize = vmr.size;
vmai.memoryTypeIndex = FindMemoryThatIsHostVisible();

VkDeviceMemory vdm;
result = vkAllocateMemory(LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm);
result = vkBindBufferMemory(LogicalDevice, PosBuffer, IN vdm, 0); // 0 is the offset
```

Create the Compute Pipeline Layout

```cpp
Create the Compute Pipeline Layout

```

Create the Compute Pipeline

```cpp
Create the Compute Pipeline

```
Creating a Vulkan Data Buffer

```cpp
VkBuffer Buffer;
VkBufferCreateInfo vbci;
vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
vbci.pNext = nullptr;
vbci.flags = 0;
vbci.size = NUM_PARTICLES * sizeof(glm::vec4);
vbci.usage = VK_USAGE_STORAGE_BUFFER_BIT;
vbci.sharingMode = VK_SHARING_MODE_CONCURRENT;
vbci.queueFamilyIndexCount = 0;
vbci.pQueueFamilyIndices = (const int32_t*)nullptr;
result = vkCreateBuffer(LogicalDevice, IN &vbci, PALLOCATOR, OUT &posBuffer);
```

Allocating Memory and Binding the Buffer

```cpp
VkMemoryRequirements vmr;
result = vkGetBufferMemoryRequirements(LogicalDevice, posBuffer, OUT &vmr);
VkMemoryAllocateInfo vmai;
vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
vmai.pNext = nullptr;
vmai.flags = 0;
vmai.allocationSize = vmr.size;
vmai.memoryTypeIndex = FindMemoryThatIsHostVisible();
VkDeviceMemory vdm;
result = vkAllocateMemory(LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm);
result = vkBindBufferMemory(LogicalDevice, posBuffer, IN vdm, 0); // 0 is the offset
MyBuffer myPosBuffer;
myPosBuffer.size = vbci.size;
myPosBuffer.buffer = PosBuffer;
myPosBuffer.vdm = vdm;
```

Fill the Buffers

```cpp
# include <stdlib.h>
#define TOP 2147483647. // 2^31 - 1
float Ranf(float low, float high) {
    long random(); // returns integer 0 - TOP
    float r = (float)random(); // returns integer 0 - TOP
    return low + r * (high - low) / (float)RAND_MAX;
}
```
The Particle System Compute Shader

```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
};

layout( local_size_x = 64, local_size_y = 1, local_size_z = 1 ) in;
```

The Data gets Divided into Large Quantities call Work-Groups, each of which is further Divided into Smaller Units Called Work-Items

The number of work-items per work-group, set in the compute shader.
The number of work-groups is set in the `vkCmdDispatch(commandBuffer, workGroupCountX, workGroupCountY, workGroupCountZ );` function call in the application program.

The Data Needs to be Divided into Large Quantities call Work-Groups, each of which is further Divided into Smaller Units Called Work-Items

20 total items to compute:

```
#WorkGroups = GlobalInvocationSize
WorkGroupSize

5 = \frac{20}{4}
```

A Mechanical Equivalent...

http://news.cision.com
// Point, Velocity, Vector, Sphere, Plane definitions

#define POINT vec3
#define VELOCITY vec3
#define VECTOR vec3
#define SPHERE vec4
#define PLANE vec4

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

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

// update the position:
POINT pp = p + v*DT + .5*DT*DT*G;
VELOCITY vp = v + G*DT;

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

// bounce physics

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;

if( IsInsideSphere( pp, Sphere ) )
{
    float r = length( pp - Sphere.xyz );
    return ( r < Sphere.w );
}

void BounceSphere( POINT p, VELOCITY v, SPHERE s )
{
    VELOCITY vout = Bounce( v, normalize( p - s.xyz ) );
    pp = p + vout*DT + .5*DT*DT*G;
    Positions[ gid ].xyz = pp;
    Velocities[ gid ].xyz = vout;
}

// bounce on a plane
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;

void BouncePlane( POINT p, VELOCITY v, PLANE pl )
{
    VECTOR n = normalize( pl.xyz );
    vout = Bounce( v, n );
    Positions[ gid ].xyz = pp;
    Velocities[ gid ].xyz = vout;
}

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

// bounce on a sphere
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;

void BounceSphere( POINT p, VELOCITY v, SPHERE s )
{
    VELOCITY vout = Bounce( v, normalize( p - s.xyz ) );
    pp = p + vout*DT + .5*DT*DT*G;
    Positions[ gid ].xyz = pp;
    Velocities[ gid ].xyz = vout;
}

void BouncePlane( POINT p, VELOCITY v, PLANE pl )
{
    VECTOR n = normalize( pl.xyz );
    vout = Bounce( v, n );
    Positions[ gid ].xyz = pp;
    Velocities[ gid ].xyz = vout;
}
Dispatching the Compute Shader from the Command Buffer

```c
#define NUM_PARTICLES (1024*1024)
#define NUM_WORK_ITEMS_PER_GROUP (NUM_PARTICLES / NUM_WORK_ITEMS_PER_GROUP)
...
vkCmdBindPipeline(CommandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, ComputePipeline);
vkCmdDispatch(CommandBuffer, NUM_X_WORK_GROUPS, 1, 1);
```

This is the number of work-groups, set in the application program. The number of work-items per work-group is set in the layout in the compute shader:

```c
layout( local_size_x = 64, local_size_y = 1, local_size_z = 1 ) in;
```

Displaying the Particles

```c
VkVertexInputBindingDescription vvibd[3]; // one of these per buffer data buffer
vvibd[0].binding = 0; // which binding this is
vvibd[0].stride = sizeof(struct pos); // bytes between successive structs
vvibd[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
vvibd[1].binding = 1;
vvibd[1].stride = sizeof(struct vel);
vvibd[1].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
vvibd[2].binding = 2;
vvibd[2].stride = sizeof(struct col);
vvibd[2].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
```

```c
layout( location = 0 ) in vec4 aPosition;
layout( location = 1 ) in vec4 aVelocity;
layout( location = 2 ) in vec4 aColor;
```

```c
VkVertexInputAttributeDescription vviad[3]; // array per vertex input attribute
vviad[0].location = 0; // location in the layout decoration
vviad[0].binding = 0; // which binding description this is part of
vviad[0].format = VK_FORMAT_VEC4; // x, y, z, w
vviad[0].offset = offsetof(struct pos, pos); // 0
vviad[1].location = 1;
vviad[1].binding = 0;
vviad[1].format = VK_FORMAT_VEC4; // nx, ny, nz
vviad[1].offset = offsetof(struct vel, vel); // 0
vviad[2].location = 2;
vviad[2].binding = 0;
```
Telling the Pipeline about its Input

We will come to the Pipeline later, but for now, know that a Vulkan Pipeline is essentially a very large data structure that holds (what OpenGL would call) the state, including how to parse its vertex input.

```
VkGraphicsPipelineCreateInfo vgpci;
vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
vgpci.pNext = nullptr;
vgpci.flags = 0;
vgpci.stageCount = 2;                // number of shader stages in this pipeline
vgpci.pStages = vpssci;
vgpci.pVertexInputState = &vpvisci;
vgpci.pInputAssemblyState = &vpiasci;
vgpci.pTessellationState = (VkPipelineTessellationStateCreateInfo *)nullptr;            // &vptsci
vgpci.pViewportState = &vpvsci;
vgpci.pRasterizationState = &vprsci;
vgpci.pMultisampleState = &vpmsci;
vgpci.pDepthStencilState = &vpdssci;
vgpci.pColorBlendState = &vpcbsci;
vgpci.pDynamicState = &vpdsci;
vgpci.layout = IN GraphicsPipelineLayout;
vgpci.renderPass = IN RenderPass;
vgpci.subpass = 0;                              // subpass number
vgpci.basePipelineHandle = (VkPipeline) VK_NULL_HANDLE;
vgpci.basePipelineIndex = 0;
result = vkCreateGraphicsPipelines(LogicalDevice, VK_NULL_HANDLE, 1, IN &vgpci, PALLOCATOR, OUT &GraphicsPipeline);
```

Setting a Pipeline Barrier so the Drawing Waits for the Compute

```
const uint32 bufferMemoryBarrierCount = 1;
vkCmdPipelineBarrier(
commandBuffer,
VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT, VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT,
VK_DEPENDENCY_BY_REGION_BIT, 0,  nullptr, bufferMemoryBarrierCount, IN &vbmb, 0,nullptr
);
```

Drawing

```
VkBuffer buffers[] = MyPosBuffer.buffer, MyVelBuffer.buffer, MyColBuffer.buffer
size_t offsets[] = { 0, 0, 0 }

vkCmdBindVertexBuffers(CommandBuffers[nextImageIndex], 0, 3, buffers, offsets);
const uint32_t  vertexCount = NUM_PARTICLES;
const uint32_t  instanceCount = 1;
const uint32_t  firstVertex = 0;
const uint32_t  firstInstance = 0;
vkCmdDraw(CommandBuffers[nextImageIndex], NUM_PARTICLES, 1, 0, 0
); // vertexCount, instanceCount, firstVertex, firstInstance
```

Setting a Pipeline Barrier so the Compute Waits for the Drawing

```
const uint32 bufferMemoryBarrierCount = 1;
vkCmdPipelineBarrier(
commandBuffer,
VK_PIPELINE_STAGE_VERTEX_INPUT_BIT, VK_PIPELINE_STAGE_VERTEX_SHADER_BIT,  // VK_DEPENDENCY_BY_REGION_BIT, 0, nullptr, bufferMemoryBarrierCount, IN &vbmb, 0,nullptr
);
```