

# P. Threads

## The Problem

Multithreading is a solution to the growing problem of reduced gains from increased clock speed in processors. This example will focus on ray tracing a single sphere with some basic lighting to demonstrate the effects that multithreading can have on solving this problem. The Samaritan renderer used in this example creates a texture from an array of pixel data, which is transferred via a staging buffer to the GPU. This happens inside the `CreateTexture` function, shown on the left.

```
metcode Renderer_VLH::CreateTexture()
{
    VkResult result = VK_ERROR_UNKNOWN;
    RetCode retcode = APP_ERROR;

    const uint32_t width = n_SphereInExtent.width;
    const uint32_t height = n_SphereInExtent.height;
    std::vector<uint32_t> initialPixelData;
    initialPixelData.reserve(width * height, 0xffffffff);

    VkBuffer stagingBuffer;
    VkDeviceMemory stagingBufferMemory;
    VkDeviceSize size = width * height * sizeof(uint32_t);

    VkMemoryProperties memoryFlags = VK_MEMORY_PROPERTY_HOST_COHERENT_BIT |
        VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT;
    CreateStagingBuffer(stagingBuffer, memoryFlags, size, stagingBufferMemory);
    AssertRetCode == APP_OK, "Error: Failed to create staging buffer for texture!";

    void* data;
    result = vkhMemoryMapDevice(stagingBufferMemory, 0, size, 0, GData);
    Assert(result == VK_SUCCESS, "Error: Failed to map memory for texture staging buffer!");

    memcpy(data, initialPixelData.data(), static_cast<size_t>(size));
    vkhMemoryUnmapDevice(stagingBufferMemory);

    retCode = CreateImage
    {
        static const uint32_t i(width)
        static const uint32_t j(height)
        VK_FORMAT_R8G8B8A8_SRGB, VK_IMAGE_LAYOUT_UNDEFINED, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, m_Texture);
    Assert(result == APP_OK, "Error: Failed to transition image layout!");
    retCode = CopyBufferToImage(static_cast<uint32_t>(width), static_cast<uint32_t>(height), stagingBuffer, m_Texture);
    Assert(result == APP_OK, "Error: Failed to copy buffer to image!");

    // Prepare the image for shader access
    Assert(result == APP_OK, "Error: Failed to transition image format to VK_FORMAT_SHADER_READ_ONLY_OPTIMAL, m_Texture");
    Assert(result == APP_OK, "Error: Failed to transition image to shader readable format!");
    vkhDestroyBuffer(m_Device, stagingBuffer, nullptr);
    vkhDestroyMemory(m_Device, stagingBufferMemory, nullptr);
    return result == VK_SUCCESS ? retcode : APP_ERROR;
}
```

```
glm::vec4 Renderer_VLH::UpdatePixel(const glm::vec2& _pixelCoord)
{
    // Cast rays from the camera position to the texture to determine pixel colour.
    const glm::vec3 cameraPosition = { 0.f, 0.f, 1.f };
    const glm::vec3 rayDirection = { _pixelCoord, -1.f };
    const glm::vec3 lightDirection = glm::normalize(glm::vec3(-2.f, -2.f, 1.f));

    const float radius = 0.5f;

    const float a = glm::dot(rayDirection, rayDirection);
    const float b = glm::dot(cameraPosition, rayDirection) * 2.f;
    const float c = glm::dot(cameraPosition, cameraPosition) - radius * radius;

    const float d = b * b - 4.f * a * c;

    // Stop early if there are no possible quadratic solutions, i.e. the ray missed the sphere.
    if (d < 0.f)
    {
        return glm::vec4(0.f, 0.f, 0.f, 1.f);
    }

    // Solve the quadratic equation to find the distance to the sphere, i.e. the length of the ray.
    // t0 = (-b + glm::sqrt(d)) / (2.f * a);
    // t1 = (-b - glm::sqrt(d)) / (2.f * a);
    // a will always be negative, so t1 will always yield the closest hit.
    float t1 = (-b - glm::sqrt(d)) / (2.f * a);

    glm::vec3 nearestHit = cameraPosition + rayDirection * t1;

    // Adjust sphere colour space to be between 0 and 1.
    glm::vec3 hitNormal = glm::normalize(nearestHit);
    hitNormal = hitNormal * 0.5f + 0.5f;

    // Control light intensity, uses max to reject negative values
    const float finalIntensity = glm::max(0.f, glm::dot(hitNormal, -lightDirection));
    hitNormal *= finalIntensity;

    return glm::vec4(hitNormal, 1.f);
}
```

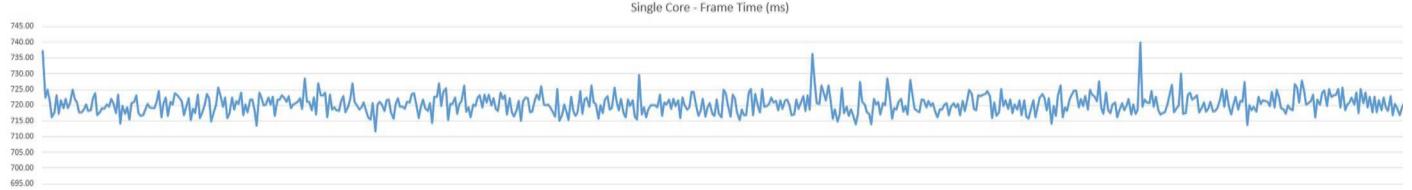
An array of pixel data is then manipulated using the `UpdatePixel()` function shown above on the right. This function is specifically designed to act as a kind of C++ fragment shader, which is why it returns a `glm::vec4` for the pixel colour.

```
uint32_t ConvertToInt(glm::vec4 colour)
{
    uint8_t r = static_cast<uint8_t>(colour.r * 255.f);
    uint8_t g = static_cast<uint8_t>(colour.g * 255.f);
    uint8_t b = static_cast<uint8_t>(colour.b * 255.f);
    uint8_t a = static_cast<uint8_t>(colour.a * 255.f);

    return (a << 24) | (b << 16) | (g << 8) | r;
}
```

This colour is then converted to a `uint32_t` inside `ConvertToInt()`, which manipulates the individual bytes of the resulting `uint32_t`. Once the pixel colours have all been calculated, the pixel array owned by the renderer is copied to the GPU in the same way the initial pixel data was in `CreateTexture()`.

As the graph below shows, this all takes a significant amount of time. Running on a single core, each update takes an average of 720.41 ms to draw one sphere.



## The Solution

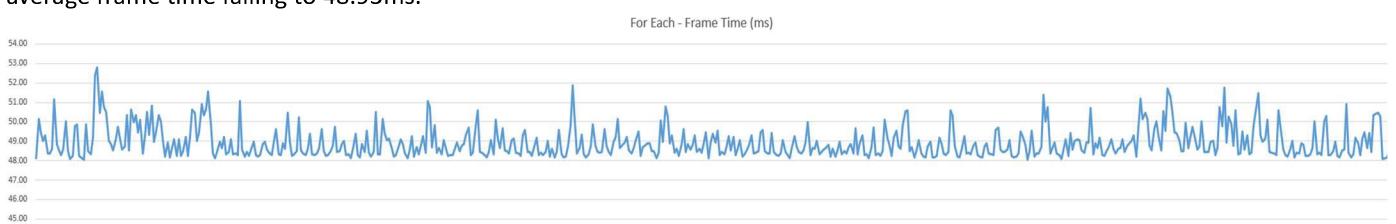
Each ray, and each pixel, are entirely independent from each other. This allows the algorithm to be threaded, allowing multiple CPU threads to calculate colours and update the array of pixels. The resolution of the pixel array remains constant, so as long as no threads overlap and try to work on the same rows, the task is relatively collision free.



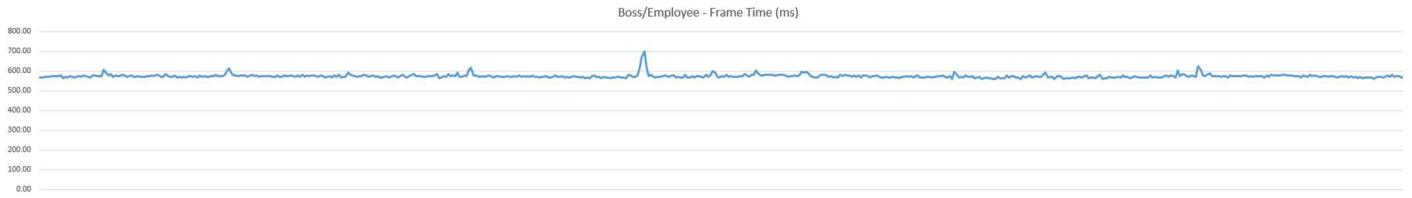
The Standard Template Library can help a lot in situations like this, and a great solution can be to use a simple `std::for_each` loop. `Std::for_each` can be overloaded with an execution policy using `std::execution`, in this case we'll use `std::execution::par`.

This will indicate to the loop that the following code is parallelisable, with some caveats. For example, it's the programmer's responsibility to handle any shared data resources effectively to avoid results like the image on the left.

Its generic nature means that `std::for_each` isn't the perfect solution but provides decent improvements nonetheless, with average frame time falling to 48.93ms.



A naïve way to mimic std::for\_each would be a simplified Boss / Employee pattern. With the main thread acting as the boss who assigns work to a new employee for every job that comes in. The issue with this is that thread creation is not free, so the algorithm ends up taking much longer! With an average frame time of 574.56ms, with some readings peaking at around 700ms.



A better solution is to take inspiration from the std::for\_each implementation, which queries the number of hardware threads available. Assuming there's more than 1 thread the work is chunked up and sent using a call to \_Run\_chunked\_parallel\_work, which in turn uses a thread pool. (See image below)

```
template <class _Work>
void _Run_chunked_parallel_work(const size_t _Hw_threads, _Work& _Operation) {
    // process chunks of _Operation on the thread pool
    const _Work_ptr _Work_op{ _Operation };
    // setup complete, hereafter nothrow or terminate
    _Work_op._Submit_for_chunks(_Hw_threads, _Operation._Team._Chunks);
    _Run_available_chunked_work(_Operation);
}
```

A thread pool is a more efficient way to use multiple threads, because the overhead of thread creation doesn't increase with every job.

```
struct Job
{
    void* (*function)(void*); // function to complete
    void* args; // arguments to provide to that function
};

struct ThreadInfo
{
    glm::uvec2 resolution = { 0, 0 };
    uint32_t* container = nullptr;
    bool* containerFlags = nullptr;
    uint32_t rowID = 0;
};
```

Implementing the thread pool requires a job queue which can be added to by the main thread, and a system to wake threads when work is available. The full implementation can be found in ThreadPool.h, but some key pieces are shown here.

The concept of a job is simply the function to complete, and the argument to provide to that function. As a thread can only accept one argument, a custom structure was used to pass information to the executing thread.

```
ThreadInfo info{};
info.container = m_PixelData;
info.containerFlags = m_PixelRowDirtyFlags;
info.resolution = { width, height };
info.rowID = y;

Job job;
job.function = ProcessPixelRow;
job.args = (void*)&info;

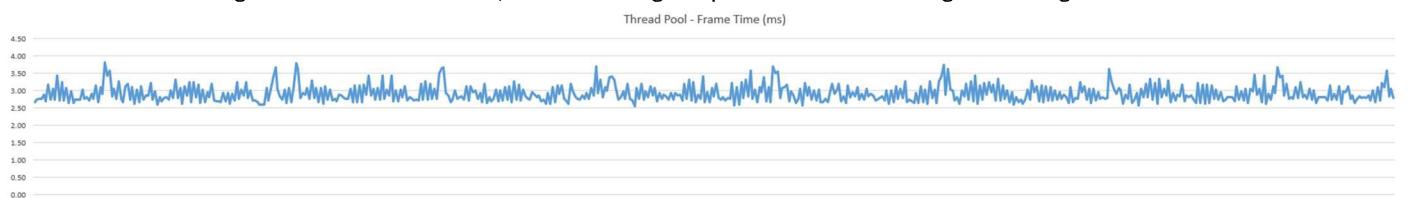
m_ThreadPool->AddJob(job);
```

When it's time to update the texture, a ThreadInfo struct, and Job struct are filled in, and then passed to the thread pool, which protects its queue with a max job size to prevent huge numbers of jobs being added to queue.

The thread pool then signals to a waiting thread that a job is available, and that thread processes the job inside the DoWork() function. (See ThreadPool.h, lines 71 – 97)

There's one more piece of the puzzle to add to stop the main thread adding huge numbers of jobs to the queue, a max queue count. Threads will take longer to process jobs than it takes to add them to the queue in any case, but without this, writing to the queue excessively means the threads can't pop jobs off because the queue is always locked.

The result is an average frame time of 2.92ms, which is a huge improvement on the original average of 720.41ms.



In the final version, UpdateTexture uses an array of dirty flags to indicate if a row of the texture needs to be updated. These flags stop the main thread queueing unnecessary work, significantly reducing the number of jobs added to the queue. This further reduces the average frame time to 2.58ms, which is arguably not significant, but an improvement nonetheless.

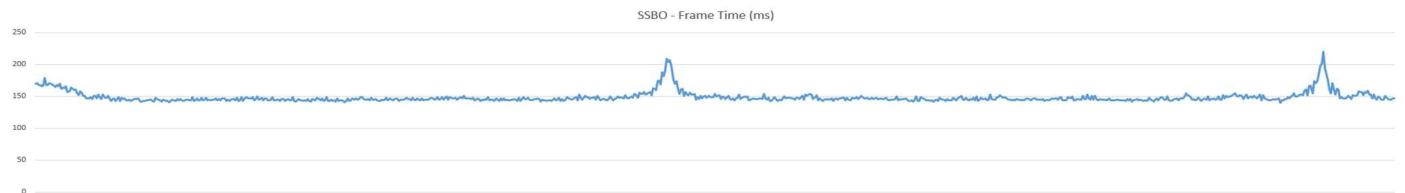
```
// Send each row of pixels to the threadpool
for (uint32_t y = 0; y < height; y++)
{
    if (m_PixelRowDirtyFlags[y] > 0)
    {
```

# Compute

## The Problem

Managing millions of particles on the CPU is comparatively slow. This can be demonstrated using a simple Shader Storage Buffer Object (SSBO) setup in Vulkan, in which the SSBO's are updated every frame on the CPU using the glm maths library. (See lines 355 – 390 in Renderer\_SSBO.cpp)

As the chart below demonstrates, the particle system updates in an average of 148.5 milliseconds. Notice the spikes in frame time that are also present. These are caused by increases in control divergence which will be discussed in the next section.



## The Solution

One solution is to leverage the GPU via a compute shader. Vulkan provides support for asynchronous compute but at least 1 queue family is guaranteed to have both the graphics and compute bits.

```
569 // We require at least one queue family that has the GFX bit. We also require the transfer bit for the staging buffer
570 // but as every queue that has the GFX bit is guaranteed to have the transfer bit we'll just check for that.
571 // At least one queue is guaranteed to have both the graphics and compute bits, we'll use that rather than async compute.
572 if ((queueFamily.queueFlags & VK_QUEUE_GRAPHICS_BIT) && (queueFamily.queueFlags & VK_QUEUE_COMPUTE_BIT))
573 {
574     _indices.graphicsFamily = i;
575 }
```

Async compute also requires advanced synchronisation methods beyond the scope of this assignment, whereas fences and semaphores are all that's needed for a combined graphics-compute queue, as seen in Renderer\_VLKN::DrawFrame in Renderer\_Compute.cpp. (Line 232 – 336, snippets shown in images below)

```
// Compute
vkWaitForFences(m_Device, 1, &m_vComputeInFlight[m_CurrentFrame], VK_TRUE, UINT64_MAX);
vkResetFences(m_Device, 1, &m_vComputeInFlight[m_CurrentFrame]);

vkResetCommandBuffer(m_vComputeCommandBuffers[m_CurrentFrame], 0);

PushConstantData data{};
data.deltaTime = _deltaSeconds;
data.elapsedSeconds = _TotalRenderTime;

retCode = RecordComputeCommandBuffer(m_vComputeCommandBuffers[m_CurrentFrame], &data);
Assert(retCode == APP_OK, "ERROR: Failed to record compute command buffer!");

submitInfo.commandBufferCount = 1;
submitInfo.pCommandBuffers = &m_vComputeCommandBuffers[m_CurrentFrame];
submitInfo.signalSemaphoreCount = 1;
submitInfo.pSignalSemaphores = &m_vComputeFinishedSemaphores[m_CurrentFrame];

result = vkQueueSubmit(m_ComputeQ, 1, &submitInfo, m_vComputeInFlight[m_CurrentFrame]);
Assert(result == VK_SUCCESS, "ERROR: Failed to submit compute!");
//
```

```
// Graphics
Assert(_pWindow, "ERROR: Cannot draw a frame without a valid window");
if (_pWindow)
{
    // Wait for all fences with no limit on how long we wait.
    result = vkhWaitForFences(m_Device, 1, &m_vImagesInFlight[m_CurrentFrame], VK_TRUE, UINT64_MAX);
    Assert(result == VK_SUCCESS, "ERROR: Failed to fences!");

    uint32_t imageIndex;
    result = vkhAcquireNextImageKHR(m_Device, m_Swapchain, UINT64_MAX, m_vImageAvailableSemaphores[m_CurrentFrame], VK_NULL_HANDLE, &imageIndex);

    // Usually happens because the window was resized.
    if (result == VK_ERROR_OUT_OF_DATE_KHR)
    {
        retCode = RecreateSwapchain(_pWindow);
        Assert(result == VK_SUCCESS, "ERROR: Failed to recreate swapchain!");
        return retCode;
    }

    // Check for successful image acquisition. Suboptimal is considered a success return code.
    Assert(result == VK_SUCCESS || result == VK_SUBOPTIMAL_KHR, "ERROR: Failed to acquire swap chain image!");

    // Reset the fence.
    result = vkResetFences(m_Device, 1, &m_vImagesInFlight[m_CurrentFrame]);
    Assert(result == VK_SUCCESS, "ERROR: Failed to reset fences!");

    vkResetCommandBuffer(m_vCommandBuffers[m_CurrentFrame], 0);
    RecordCommandBuffer(m_vCommandBuffers[m_CurrentFrame], imageIndex);
}

//
```

An important step for compute is to select an appropriate number of workgroups for the task. The max invocations per workgroup is queried by Samaritan when the physical device is selected.

```
// Retrieve the workgroup capacity of the selected GPU.
if (_PhysicalDevice)
{
    VKPhysicalDeviceProperties deviceProperties;
    vkGetPhysicalDeviceProperties(_PhysicalDevice, &deviceProperties);

    m_WorkgroupInvocations = deviceProperties.limits.maxComputeWorkGroupInvocations;
}
```

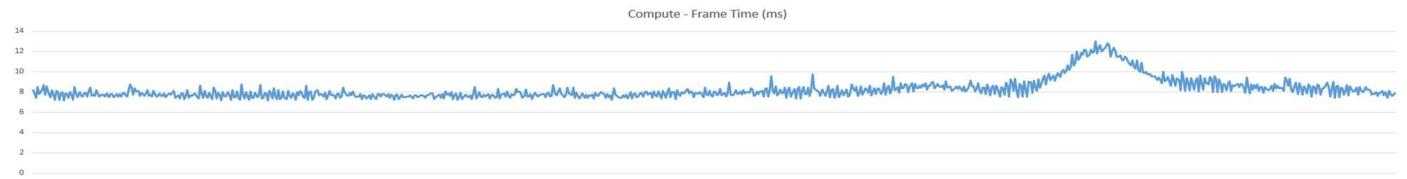
The maximum push constant capacity is also queried and used to score GPU candidates when selecting a device, although caution should be used here as only 128 bytes are guaranteed by the Vulkan standard so app mobility should be considered.

```
// ...and so are GPU's with a high push constant limit. Caution should still be used for
// cross-platform apps though as only 128 bytes are guaranteed by the Vulkan standard!
score += deviceProperties.limits.maxPushConstantsSize;
```

Before the compute dispatch command, the maximum number of particles is divided by the max invocations to give a suitable workgroup count. This means that the app should always saturate workgroups, with the possibility that 1 workgroup will pick up any stragglers. (Although this won't happen in this example because g\_MAX\_PARTICLES is guaranteed to be a multiple of m\_WorkgroupInvocations)

```
// Dividing the max particles by the max invocations per workgroup minimises the number of workgroups the application might underutilise.
// If g_MAX_PARTICLES is also kept as a product of X * m_WorkgroupInvocations then each workgroup will be saturated.
uint32_t workGroupCount = g_MAX_PARTICLES / m_WorkgroupInvocations;
if (g_MAX_PARTICLES % m_WorkgroupInvocations != 0)
{
    // Round up, although, as mentioned above we should avoid this situation in this example.
    // In a real-world example, 100% saturation is unlikely.
    workGroupCount += 1;
}
```

As the image below shows, the compute version is much faster in terms of milliseconds taken per update loop. With this example taking an average of 8.26ms per update. However, the frame time spikes seen in the previous example are still present.



These spikes are caused by control divergence within the compute shader. Particles are bounced off a barrier which changes size every frame. More particles trigger IsAtBoundary (see line 64 below) as the barrier radius shrinks because the particles simply take less time to travel from end to end, thus, the branching behaviour is exacerbated by the increased branching frequency, which also reduces branch predictability.

```
64    bool IsAtBoundary = !IsInCircle(radius, centre, particlesOut[index].position);
```

Vulkan introduced an extension for SPIR-V compilers (1.3 and up) which allows us to hint which branch will be taken on a subgroup scale, where a subgroup is a division of a workgroup, typically containing 32 invocations on Nvidia GPU's. This allows for optimisations of subgroups where every invocation within the subgroup is doing the same thing.

```
// Are all the particles in this subgroup at the boundary?
if(subgroupAll(IsAtBoundary))
{
    particlesOut[index].velocity = -particle.velocity;
    particlesOut[index].position = particle.position + particlesOut[index].velocity * speedDelta;
}
else if(!subgroupAny(IsAtBoundary)) // Are any of them at the boundary?
{
    // All the particles aren't at the boundary so just carry on as normal.
    return;
}
else // Then we're doing a mix, so fall back to the original solution
{
    if(IsAtBoundary)
    {
        particlesOut[index].velocity = -particle.velocity;
        particlesOut[index].position = particle.position + particlesOut[index].velocity * speedDelta;
    }
}
```

```
1 #version 450
2 #extension GL_KHR_shader_subgroup_vote: enable
```

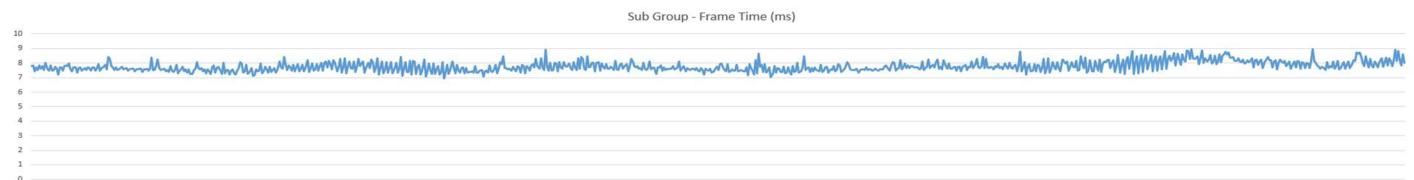
There are 3 outcomes to check.

subgroupAll(condition) checks if the condition is true for all the invocations of this subgroup.

If not, then subgroupAny(condition) checks if the condition is true for ANY of the invocations, if false, then it must be false for them all.

Finally, we must provide a fallback in case this subgroup is mixed, this is the worst case, but can't be worse than the original version. In this example, the particles either reverse, or do nothing, hence the early return if IsAtBoundary is false for all invocations in the subgroup.

As the graph shows, there is still a small frame time spike visible, but it is much better than the previous version. Average frame time also fell to 7.765ms, however, this is not quite enough to be considered statistically significant given the variation in spike duration and severity.



## References

The Samaritan app is a project I've worked on in my spare time this academic year. It's designed to be a simple framework for an app which creates a window using the win32 API, then renders using Vulkan. It is based on this archive [https://github.com/DomMc/Vulkan\\_LearningExercise](https://github.com/DomMc/Vulkan_LearningExercise) which is also my own work from the very start of the Zool project in level 6 when the aim was to get a triangle on a screen. That exercise was itself based on the Sascha Willems examples on GitHub.

<https://learning.oreilly.com/library/view/pthreads-programming/9781449364724/> - Pthreads Programming by Dick Butler, et al.

[https://en.cppreference.com/w/cpp/algorithm/execution\\_policy\\_tag](https://en.cppreference.com/w/cpp/algorithm/execution_policy_tag) - Reference for std::execution, inspiration for using this was originally drawn from this video by TheCherno - <https://www.youtube.com/watch?v=46ddlUImiQA>

[https://vulkan-tutorial.com/Compute\\_Shader#page\\_Compute-shaders](https://vulkan-tutorial.com/Compute_Shader#page_Compute-shaders) – Article by Alexander Overvoorde and Sascha Willems discussing headless compute, the Vulkan compute pipeline, and providing the design for which this compute particle system was based on.