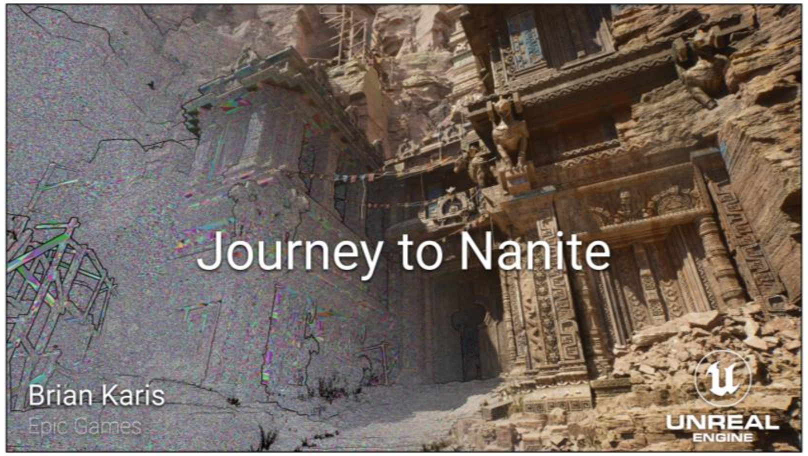
Vulkan for Compute

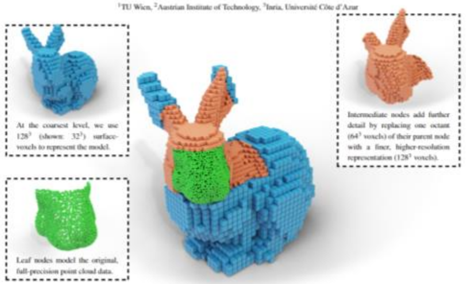
Why Compute?

- **Parallel performance**: GPU supports the CPU for non-graphics tasks
- **Unlocks features**: E.g., faster memory spaces (shared)
- **Educational**: reveals more about modern GPU architectures
- **Allows to focus on performance**: simpler setup, data in/data out
- Upcoming: hopefully, a healthy mix of foundations, examples and tools to get excited about using compute in the Vulkan API!

Compute in Graphics on the Rise



Nanite Presentation by Brian Karis at HPG 2022


High-Performance Graphics (2023)
J. Bikker and C. Gribble (Editors)

GPU-Accelerated LOD Generation for Point Clouds

Markus Schütz[1], Bernhard Kerbl[3], Philip Klaus[2], Michael Wimmer[1]

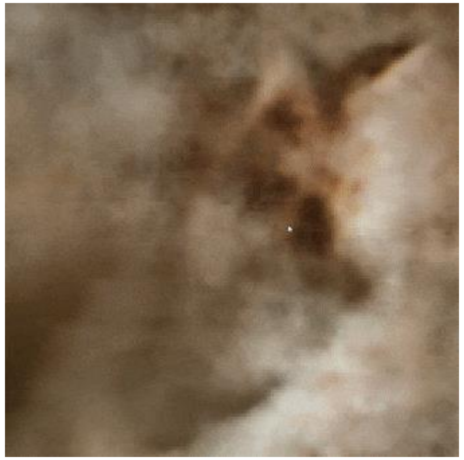
[1]TU Wien, [2]Austrian Institute of Technology, [3]Inria, Université Côte d'Azur

At the coarsest level, we use 128^3 (shown: 32^3) surface-voxels to represent the model.

Intermediate nodes add further detail by replacing one octant (64^3 voxels) of their parent node with a finer, higher-resolution representation (128^3 voxels).

Leaf nodes model the original, full-precision point cloud data.

Figure 1: The data structure is a hybrid voxel-point octree that utilizes color-filtered voxels for lower LODs, but displays the original point data at highest LOD. As most grid cells of each node are empty, surface-voxels are stored as vertices in a node's vertex buffer. During rendering, we aim to rasterize pixel-sized voxels to give viewers the impression that they are looking at full-precision point cloud data.



Instant-NGP Mueller et al., 2022 or **Nerfshop**, Jambon et al., 2023

DOI: 10.1111/cgf.14345
Eurographics Symposium on Rendering 2021
A. Bousseau and M. McGuire
(Guest Editors)
Volume 40 (2021), Number 4

Rendering Point Clouds with Compute Shaders and Vertex Order Optimization

Markus Schütz, Bernhard Kerbl, Michael Wimmer

TU Wien

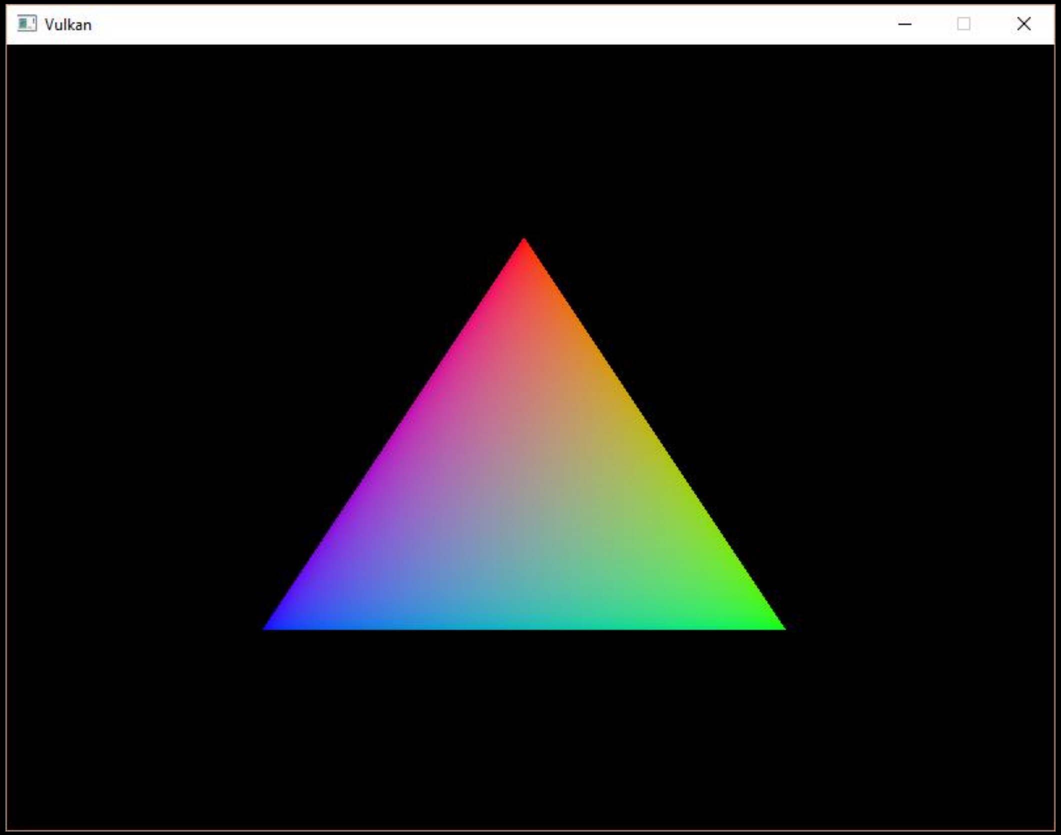
Figure 1: *Performance of GL_POINTS compared to a compute shader that performs local reduction and and early-z testing, and a high-quality compute shader that blends overlapping points. Retz point cloud (145 million points) courtesy of Riegl.*

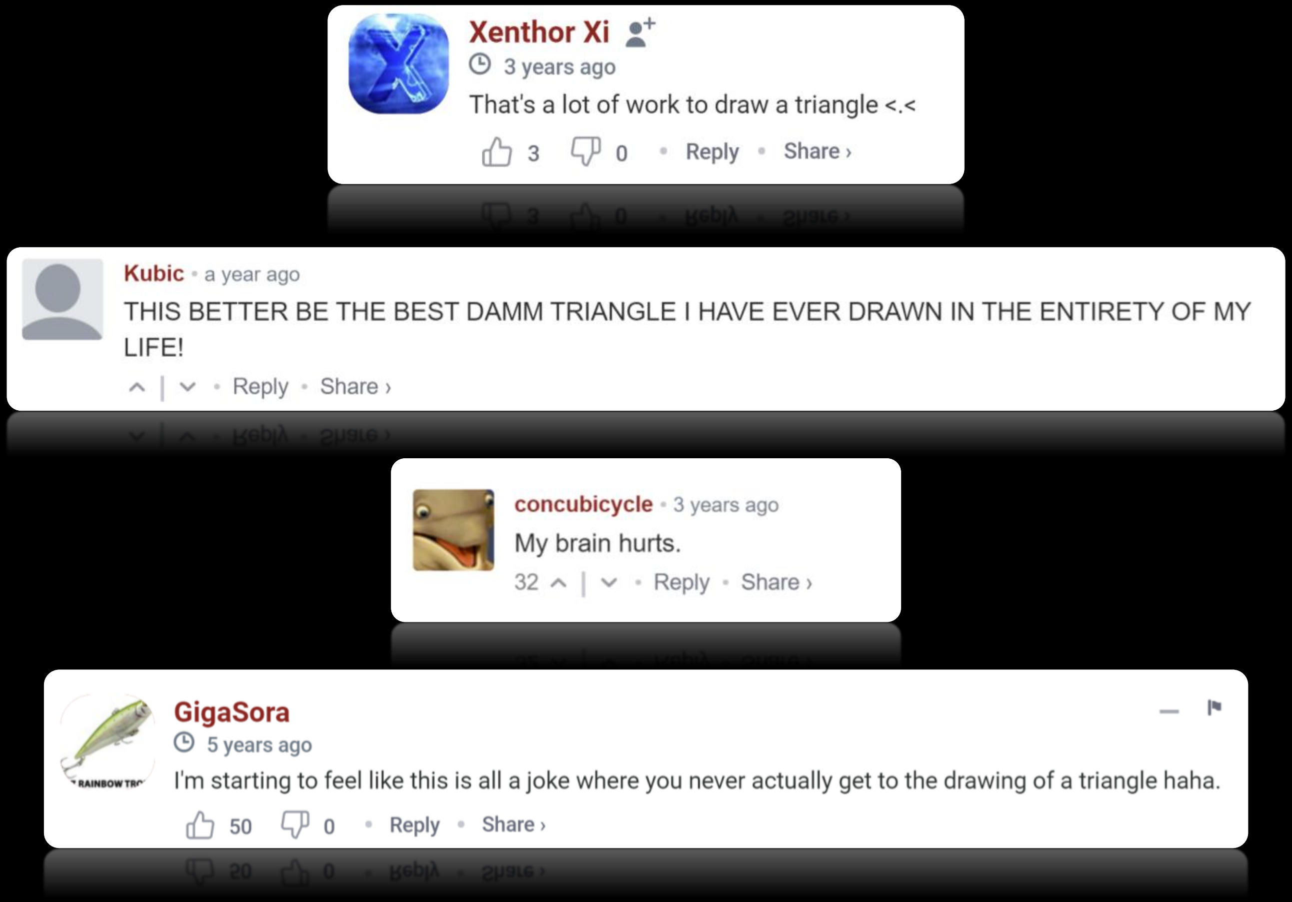
Abstract

In this paper, we present several compute-based point cloud rendering approaches that outperform the hardware pipeline by up to an order of magnitude and achieve significantly better frame times than previous compute-based methods. Beyond basic closest-point rendering, we also introduce a fast, high-quality variant to reduce aliasing. We present and evaluate several variants of our proposed methods with different flavors of optimization, in order to ensure their applicability and achieve optimal performance on a range of platforms and architectures with varying support for novel GPU hardware features. During our experiments, the observed peak performance was reached rendering 796 million points (12.7GB) at rates of 62 to 64 frames per second (50 billion points per second, 802GB/s) on an RTX 3090 without the use of level-of-detail structures.
We further introduce an optimized vertex order for point clouds to boost the efficiency of GL_POINTS by a factor of 5× in cases where hardware rendering is compulsory. We compare different orderings and show that Morton sorted buffers are faster

3D Gaussian Splatting for Real-Time Radiance Field Rendering (SIGGRAPH '23)

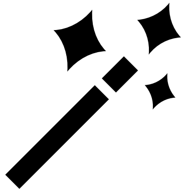
„Hello Triangle“ with Vulkan Graphics





Why so verbose?

- Your modern CPU is a magical place…



- The GPU is comparably simple. But not exactly a state machine!
- To achieve high efficiency, it needs to exploit work scheduling, caching, re-layouting of memory
- OpenGL/DirectX 11: can make excellent use of your GPU, **but** you are likely to **perceive a version** of it that is simply outdated

Tool Tip: Vulkan.hpp

- C++-style extension, which includes
  - Constructors for structs with very reasonable default arguments
  - RAII-style wrappers for almost anything (i.e., smart pointers)
  - Clean namespaces for flags and enums (help to avoid errors)

- Usually shortens code length and reduces risk of errors!

- Let’s look at the code and see what we understand already!

Logical Device Creation

```
float priority = 1.0f;
VkDeviceQueueCreateInfo queue_create_info  = {};
queue_create_info.sType                    = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
queue_create_info.queueFamilyIndex         = 0;
queue_create_info.queueCount               = 1;
queue_create_info.pQueuePriorities         = &priority;

const char* enabled_extensions[1]          = { "VK_KHR_swapchain" };

VkDeviceCreateInfo create_info             = {};
create_info.sType                          = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
create_info.queueCreateInfoCount           = 1;
create_info.pQueueCreateInfos              = &queue_create_info;
create_info.enabledExtensionCount          = 1;
create_info.ppEnabledExtensionNames        = enabled_extensions;

VkDevice device;
VkResult result = vkCreateDevice(physical_device, &create_info, nullptr, &device);
CHECK_VULKAN_RESULT(result);
```

Creating a Logical Device: vulkan.hpp

```
float priority[] = { 1.0f };
```

```
std::vector<vk::DeviceQueueCreateInfo> queue_create_infos {
        vk::DeviceQueueCreateInfo({}, 0, 1, priority)
};
```

No need to set structure type!

Specify params in constructor!

```
std::vector<const char*> deviceExtensions = { VK_KHR_SWAPCHAIN_EXTENSION_NAME };
```

Don’t need to remember exact string!

“Unique” objects automatically cleaned up!

```
vk::UniqueDevice device = physicalDevice.createDeviceUnique(
        vk::DeviceCreateInfo({}, queue_create_infos, {}, deviceExtensions)
);
```

Instead of passing (length, pointer), pass std::vectors instead!

Setting up your first compute pipeline

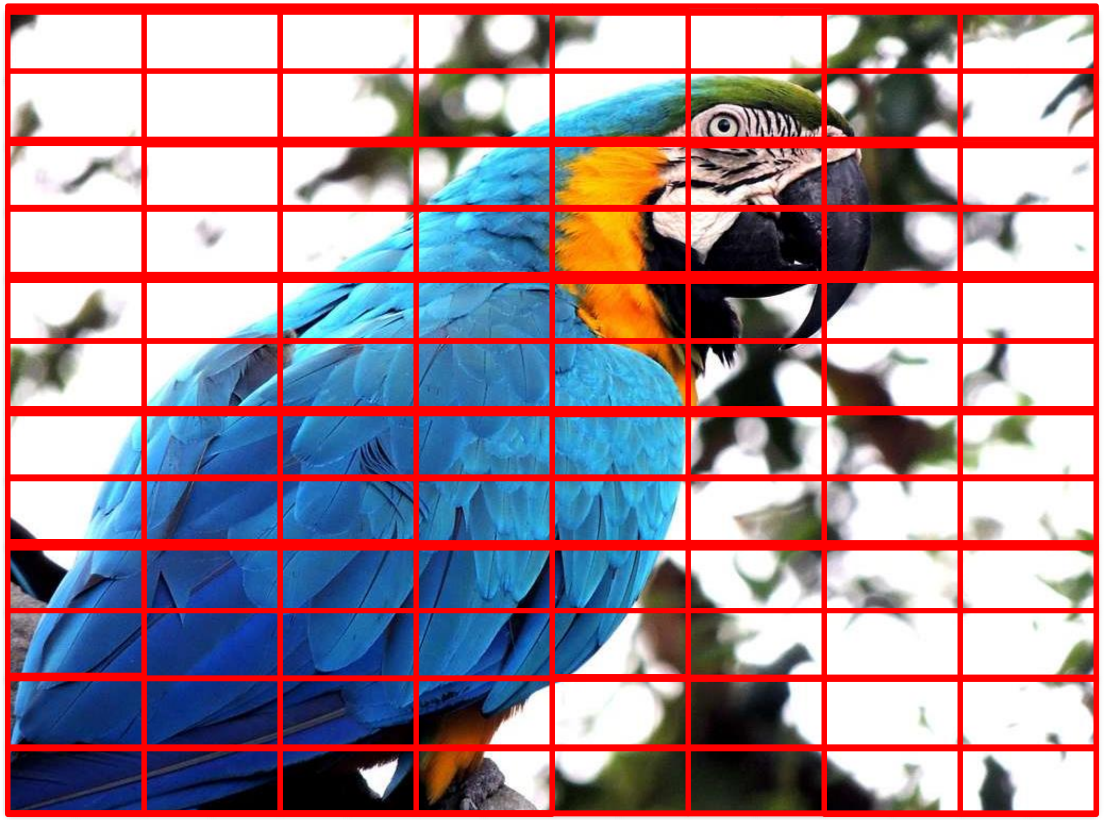
- You will need to
  - Have a file with GLSL code to execute
  - Get a compiled version of your code to SPIR-V before you run
  - Read the compiled SPIR-V file at runtime
  - Create a module from the SPIR-V code
  - Create a basic compute pipeline with that module
  - Record a command buffer that runs said pipeline
  - Make a submission of the command buffer to a queue
  - Synchronize, to make sure your GPU jobs have finished ☺

Setting up your first compute pipeline

- You will need to
  - Have a file with GLSL code to execute
  - Get a compiled version of your code to SPIR-V before you run
  - Read the compiled SPIR-V file at runtime
  - Create a module from the SPIR-V code
  - Create a basic compute pipeline with that module
  - Record a command buffer that runs said pipeline
  - Make a submission of the command buffer to a queue
  - Synchronize, to make sure your GPU jobs have finished ☺

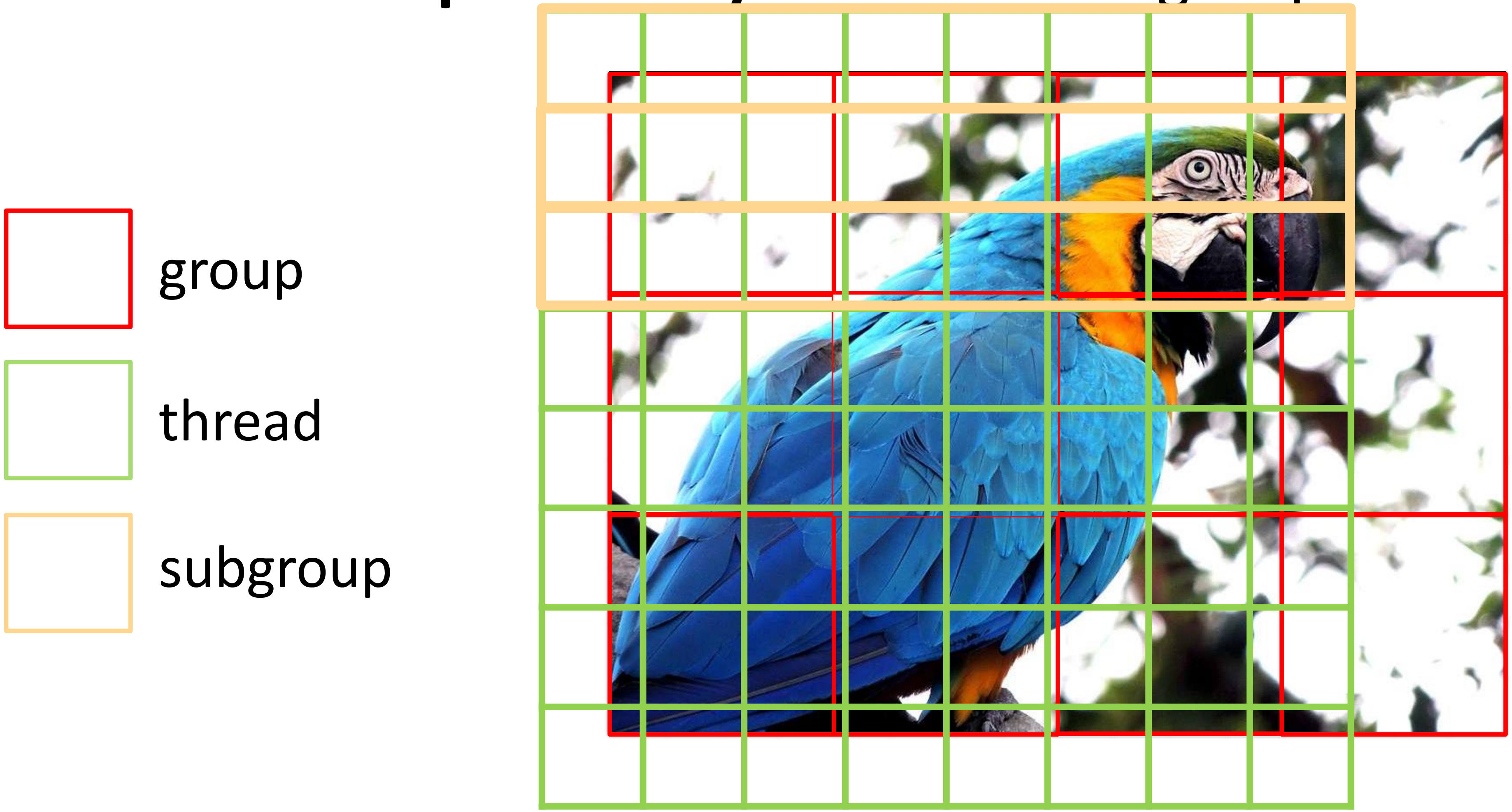
Defining Compute Job Domain

- Large GPU compute jobs are split up into smaller, **potentially** collaborative groups of threads



The Launching Hierarchy

- Large GPU compute jobs are split up into smaller, **potentially** collaborative groups of threads



Learning the GPU Compute Lingo

- The same concepts are used in most any GPU API

GLSL	HLSL	CUDA	Meaning
group	thread group	block	A group of collaborative threads
subgroup	wave	warp	SIMD width (hardware-defined)
shared	groupshared	shared	Fast memory for use within group
(unnamed)	dispatch	grid	Several groups, forming a compute job
uniform	cbuffer	constant	Read-only memory, best accessed uniformly
…	…	…	…

GLSL File

```glsl
#version 450  // version, defines capabilities

#extension GL_EXT_debug_printf : require   // extensions, if need non-default behavior

layout(local_size_x = 1, local_size_y = 1, local_size_z = 1) in;  // dimensions of each work group

void main()   // main routine to run (must be void)
{
    int x = gl_GlobalInvocationID.x;    // instructions (read built-in thread ID)
    int y = gl_GlobalInvocationID.y;
    int z = gl_GlobalInvocationID.z);
    debugPrintfEXT("Hello from thread %d %d %d!\n", x, y, z);   // Print a message!
}
```

- Save that file and compile with glslc.exe (comes with Vulkan SDK)
- `glslc hello_world.comp –o hello_world.comp.spv`

Setting up your first compute pipeline

- You will need to
  - Have a file with GLSL code to execute
  - Get a compiled version of your code to SPIR-V before you run
  - Read the compiled SPIR-V file at runtime
  - Create a module from the SPIR-V code
  - Create a basic compute pipeline with that module
  - Record a command buffer that runs said pipeline
  - Make a submission of the command buffer to a queue
  - Synchronize, to make sure your GPU jobs have finished ☺

Read your Compiled Binary File

- A thousand ways to do this in C++, but here is a good start

```
std::vector<uint32_t> loadShader(const char* path)
{
  std::ifstream infile(path, std::ios::binary | std::ios::ate);
  if (!infile.good())
    throw std::runtime_error("Unable to open shader file \"" + std::string(path) + "\"");
  std::streamsize size = infile.tellg();
  std::vector<uint32_t> buffer((size + 3) / 4);
  infile.seekg(std::ios::beg);
  infile.read((char*)buffer.data(), size);
  return buffer;
}

int main()
{
  ...
  std::vector<uint32_t> shaderCode = loadShader("hello_world.comp.spv");
  ...
```

Setting up your first compute pipeline

- You will need to
  - Have a file with GLSL code to execute
  - Get a compiled version of your code to SPIR-V before you run
  - Read the compiled SPIR-V file at runtime
  - Create a module from the SPIR-V code
  - Create a basic compute pipeline with that module
  - Record a command buffer that runs said pipeline
  - Make a submission of the command buffer to a queue
  - Synchronize, to make sure your GPU jobs have finished ☺

Create a Module from Loaded SPIR-V code

- Let's see vulkan.hpp in action

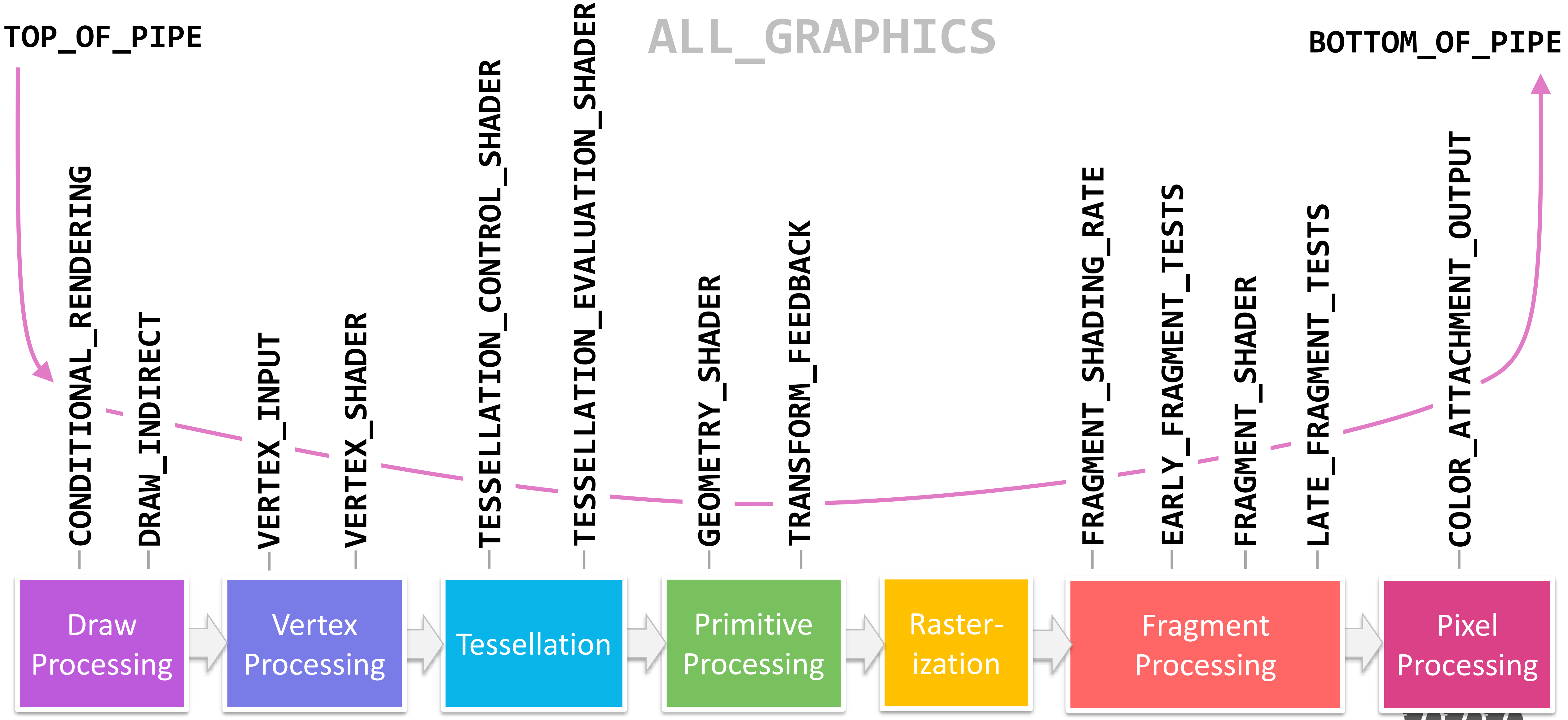
```
auto module = device->createShaderModuleUnique(vk::ShaderModuleCreateInfo({}, shaderCode));
```

- Simple things can often be a oneliner with vulkan.hpp
- Avoids explicit creation of `vkShaderModuleCreateInfo` struct
- The module is **unique**: works like a smart pointer, at the end of the **scope**, it will be deleted

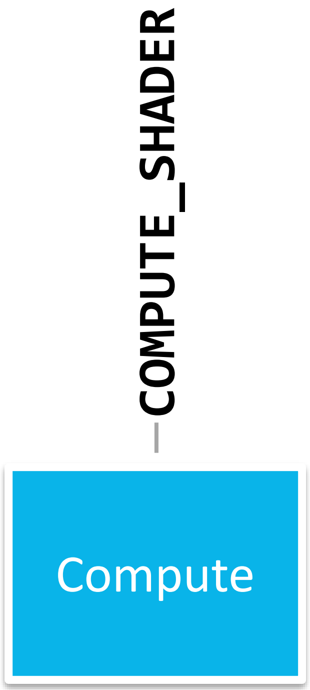
Setting up your first compute pipeline

- You will need to
  - Have a file with GLSL code to execute
  - Get a compiled version of your code to SPIR-V before you run
  - Read the compiled SPIR-V file at runtime
  - Create a module from the SPIR-V code
  - Create a basic compute pipeline with that module
  - Record a command buffer that runs said pipeline
  - Make a submission of the command buffer to a queue
  - Synchronize, to make sure your GPU jobs have finished ☺

Pipeline Stages of a Graphics Pipeline



Pipeline Stages of a Compute Pipeline



Creating a Basic Compute Pipeline

```
vk::PipelineShaderStageCreateInfo stageInfo(  // Describe a stage for our pipeline (there is only one)
{},                                            // No flags are needed: just a normal pipeline
vk::ShaderStageFlagBits::eCompute,             // A compute pipeline!
*module,                                       // Use our shader module in the pipeline
"main“                                         // Use its void main() function as entry point
);

auto pipelineLayout = device->createPipelineLayoutUnique({});
// We want nothing special for the layout. This calls default constructor of create info!

auto pipelineCache = device->createPipelineCacheUnique({});
// We want nothing special for the cache. This calls default constructor of create info!

vk::ComputePipelineCreateInfo pipelineInfo{ {}, stageInfo, *pipelineLayout };
// Explicitly make a create info that uses the stage description and our layout. *, because it is a pointer ☺

auto result = device->createComputePipelineUnique(*pipelineCache, pipelineInfo);
if(result.result != vk::Result::eSuccess)
  throw std::runtime_error("Failed to make pipeline!") // do something similar to notify user of error

auto pipeline = std::move(result.value); // Must std::move it, because it is a smart (unique) pointer. All done!
```

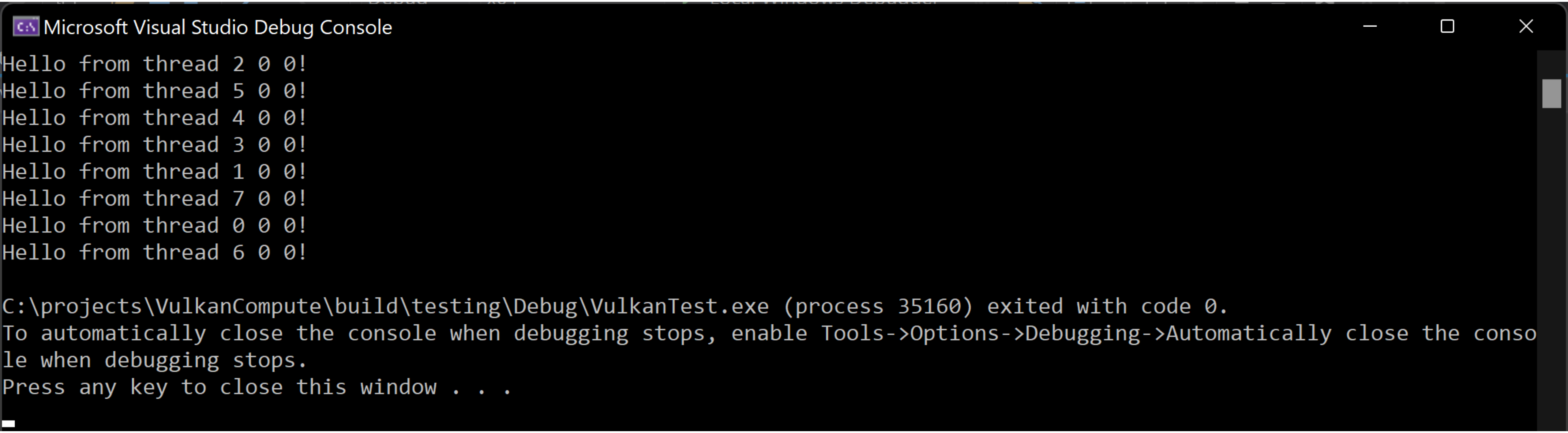
Setting up your first compute pipeline

- You will need to
  - Have a file with GLSL code to execute
  - Get a compiled version of your code to SPIR-V before you run
  - Read the compiled SPIR-V file at runtime
  - Create a module from the SPIR-V code
  - Create a basic compute pipeline with that module
  - Record a command buffer that runs said pipeline
  - Make a submission of the command buffer to a queue
  - Synchronize, to make sure your GPU jobs have finished ☺

Recording a Command Buffer to Dispatch Jobs

```
// Usually we have several command buffers, so it's an array
cmdBuffers[0]->begin({});

// Activate our pipeline as the active pipeline for compute jobs
cmdBuffers[0]->bindPipeline(vk::PipelineBindPoint::eCompute, *pipeline);

// Dispatch a compute jobs with 8 work groups
cmdBuffers[0]->dispatch(8, 1, 1);

cmdBuffers[0]->end();

// Submit and waaaait on the CPU for the job to finish running on the GPU
queue.submit(vk::SubmitInfo({}, {}, *cmdBuffers[0]));

device->waitIdle();
```

Hello World: Complete!

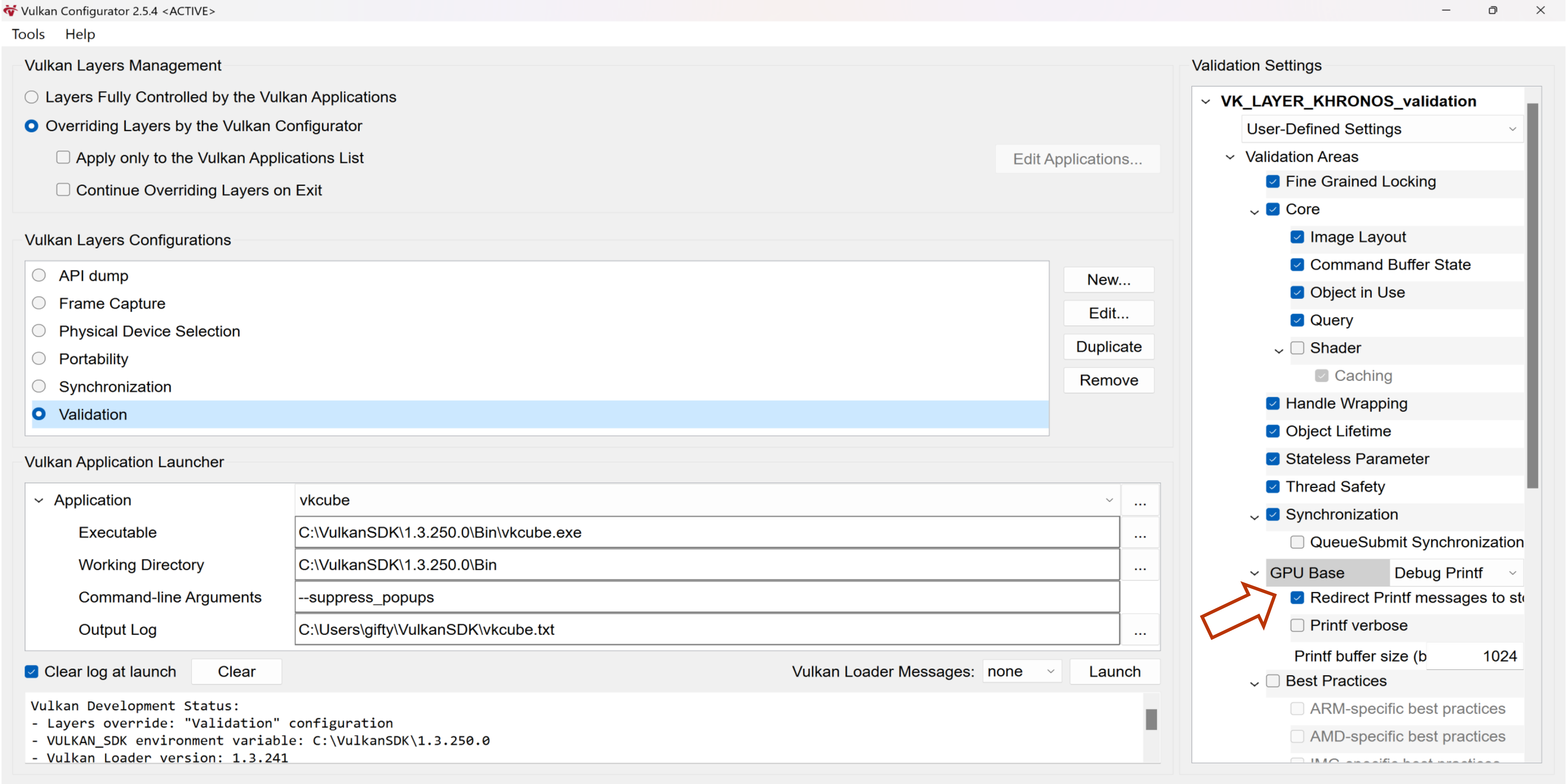


- It didn’t work?
  - Check your validation layer settings and outputs (console window)
  - Make sure the SPIR-V file has been loaded correctly (use a **debugger**!)
  - Don’t forget to synchronize at the end!

Tool Tip: VkConfig for Validation

- We mentioned that validation layers are helpful and modular
- So modular, they can be controlled by your environment too
- **Vulkan Configurator** sets up environment for you with a few clicks
- Can completely replace your validation layer setup
- Can also be used for easily enabling of debug `printf`

Tool Tip: VkConfig for Validation



```cpp
int main() {
    const vk::ApplicationInfo applicationInfo("Hello World", 0, nullptr, 0, VK_API_VERSION_1_3);
    const auto instance = vk::createInstanceUnique(vk::InstanceCreateInfo({}, &applicationInfo));
    const auto physicalDevice = instance->enumeratePhysicalDevices()[0];

    int family;
    const auto qProps = physicalDevice.getQueueFamilyProperties();
    for (family = 0; !(qProps[family].queueFlags & vk::QueueFlagBits::eCompute) && family < qProps.size(); family++);

    constexpr float priority[] = { 1.0f };
    const vk::DeviceQueueCreateInfo deviceQueueCreateInfo({}, family, 1, priority);
    const auto device = physicalDevice.createDeviceUnique(vk::DeviceCreateInfo({}, deviceQueueCreateInfo));
```

Vulkan Setup

```cpp
    const std::string print_shader = R"(
    #version 460
    #extension GL_EXT_debug_printf : require
    void main()
    { debugPrintfEXT("'Hello world!' (said thread: %d)\n", gl_GlobalInvocationID.x); })";

    const auto compiled = shaderc::Compiler().CompileGlslToSpv(print_shader, shaderc_compute_shader, "hello_world.comp");
    const std::vector<uint32_t> spirv(compiled.cbegin(), compiled.cend());
    const auto shaderModule = device->createShaderModuleUnique(vk::ShaderModuleCreateInfo({}, spirv));
    const vk::PipelineShaderStageCreateInfo stageCreateInfo({}, vk::ShaderStageFlagBits::eCompute, *shaderModule, "main");
    const auto pipelineLayout = device->createPipelineLayoutUnique(vk::PipelineLayoutCreateInfo({}));
    const vk::ComputePipelineCreateInfo pipelineCreateInfo({}, stageCreateInfo, *pipelineLayout);
    const auto [status, pipeline] = device->createComputePipelineUnique(*device->createPipelineCacheUnique({}), pipelineCreateInfo);
```

Shader

```cpp
    const auto pool = device->createCommandPoolUnique(vk::CommandPoolCreateInfo({}, family));
    const vk::CommandBufferAllocateInfo allocateInfo(*pool, vk::CommandBufferLevel::ePrimary, 1);
    const auto cmdBuffers = device->allocateCommandBuffersUnique(allocateInfo);
    cmdBuffers[0]->begin(vk::CommandBufferBeginInfo{});
    cmdBuffers[0]->bindPipeline(vk::PipelineBindPoint::eCompute, *pipeline);
    cmdBuffers[0]->dispatch(8, 1, 1);
    cmdBuffers[0]->end();
    device->getQueue(family, 0).submit(vk::SubmitInfo({}, {}, *cmdBuffers[0]));
    device->waitIdle();
    return 0;
}
```

Dispatch

```cpp
int main() {
    const vk::ApplicationInfo applicationInfo("Hello World", 0, nullptr, 0, VK_API_VERSION_1_3);
    const auto instance = vk::createInstanceUnique(vk::InstanceCreateInfo({}, &applicationInfo));
    const auto physicalDevice = instance->enumeratePhysicalDevices()[0];

    int family;
    const auto qProps = physicalDevice.getQueueFamilyProperties();
    for (family = 0; !(qProps[family].queueFlags & vk::QueueFlagBits::eCompute) && family < qProps.size(); family++);

    constexpr float priority[] = { 1.0f };
    const vk::DeviceQueueCreateInfo deviceQueueCreateInfo({}, family, 1, priority);
    const auto device = physicalDevice.createDeviceUnique(vk::DeviceCreateInfo({}, deviceQueueCreateInfo));

    const std::string print_shader = R"(
    #version 460
    #extension GL_EXT_debug_printf : require
    void main()
    { debugPrintfEXT("'Hello world!' (said thread

    const auto compiled = shaderc::Compiler().Com
    const std::vector<uint32_t> spirv(compiled.cb
    const auto shaderModule = device->createShade
    const vk::PipelineShaderStageCreateInfo stage
    const auto pipelineLayout = device->createPip
    const vk::ComputePipelineCreateInfo pipelineC
    const auto [status, pipeline] = device->creat

    const auto pool = device->createCommandPoolUn
    const vk::CommandBufferAllocateInfo allocateI
    const auto cmdBuffers = device->allocateComma
    cmdBuffers[0]->begin(vk::CommandBufferBeginIn
    cmdBuffers[0]->bindPipeline(vk::PipelineBindP
    cmdBuffers[0]->dispatch(8, 1, 1);
    cmdBuffers[0]->end();
    device->getQueue(family, 0).submit(vk::SubmitInfo({}, {}, cmdBuffers[0]));
    device->waitIdle();
    return 0;
}
```

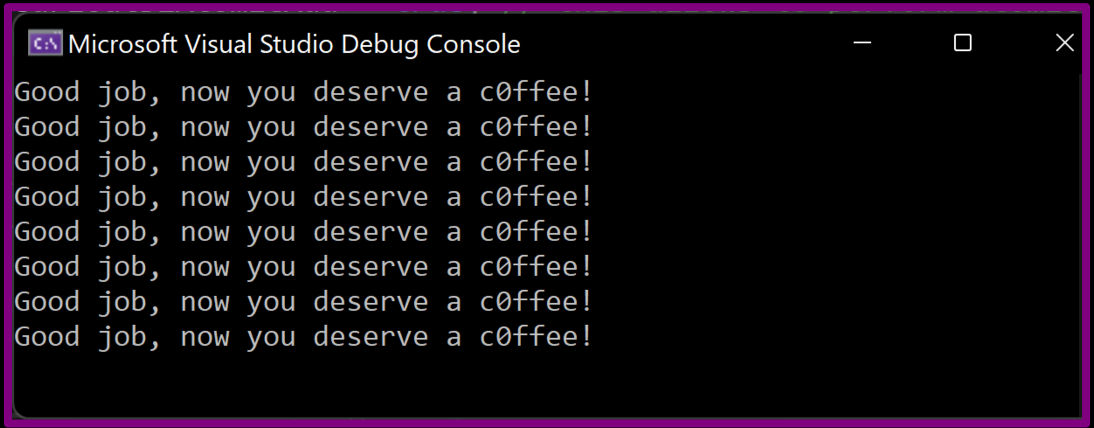
Vulkan Setup

Microsoft Visual Studio Debug Console

```
'Hello world!' (said thread: 4)
'Hello world!' (said thread: 1)
'Hello world!' (said thread: 7)
'Hello world!' (said thread: 3)
'Hello world!' (said thread: 5)
'Hello world!' (said thread: 6)
'Hello world!' (said thread: 2)
'Hello world!' (said thread: 0)

C:\projects\gpusim-vulkan\build\01_HelloGPU\Debug\01_HelloGPU.exe (process 5604) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically
close the console when debugging stops.
Press any key to close this window . . .
```

List of Coding Tasks

1. My Device(s)
2. Hello GPU
3. Uniform buffers
4. Storage buffers
5. Copying
6. Edge detector
7. Atomics
8. Point cloud renderer
9. Shared memory
10. Matrix Multiplication
11. Reduction
12. Staging Buffers
13. Final task of choice (Ray Tracer/Cloth Sim/MLP)



Stone Griffin, Downing College, Cambridge by Thomas Flynn, software-rendered. CC 4.0

Where to Find the Exercises

- https://github.com/bkerbl/vulkan-compute-exercises
- 12 hands-on programming exercises
- Requires CMake, recent Vulkan (1.3) installed with VMA
- Early alpha: glad if you test it and leave suggestions!
  - As a pull request, or
  - https://discord.gg/nSy5EgJvU (valid until July 26th, 2023)

Example: Uniform Buffers

- "Set up a uniform buffer ready to be filled with data"

```
auto usage = vk::BufferUsageFlagBits::eUniformBuffer;
auto props = vk::MemoryPropertyFlagBits::eHostVisible | vk::MemoryPropertyFlagBits::eHostCoherent |
vk::MemoryPropertyFlagBits::eDeviceLocal;
vk::BufferCreateInfo buffCreate({}, size, usage);
vk::UniqueBuffer buffer = device->createBufferUnique(buffCreate);
auto req = device->getBufferMemoryRequirements(*buffer);

uint32_t memoryIndex;
vk::PhysicalDeviceMemoryProperties memProps = physicalDevice.getMemoryProperties();
for (memoryIndex = 0; memoryIndex < memProps.memoryTypes.size(); memoryIndex++)
{
	vk::MemoryPropertyFlags flags = memProps.memoryTypes[memoryIndex].propertyFlags;
	if (req.memoryTypeBits & (1 << memoryIndex) && (flags & props) == props)
		break;
}
if (memoryIndex == memProps.memoryTypes.size())
	throw std::runtime_error("No suitable memory found!");

vk::MemoryAllocateInfo allocInfo(req.size, memoryIndex);
vk::UniqueDeviceMemory bufferMemory = device->allocateMemoryUnique(allocInfo);
device->bindBufferMemory(*buffer, *bufferMemory, 0);
```

Example: Storage Buffers

- "Set up a storage buffer ready to be filled with data"

```
auto usage = vk::BufferUsageFlagBits::eStorageBuffer;
auto props = vk::MemoryPropertyFlagBits::eHostVisible | vk::MemoryPropertyFlagBits::eHostCoherent |
vk::MemoryPropertyFlagBits::eDeviceLocal;
vk::BufferCreateInfo buffCreate({}, size, usage);
vk::UniqueBuffer buffer = device->createBufferUnique(buffCreate);
auto req = device->getBufferMemoryRequirements(*buffer);

uint32_t memoryIndex;
vk::PhysicalDeviceMemoryProperties memProps = physicalDevice.getMemoryProperties();
for (memoryIndex = 0; memoryIndex < memProps.memoryTypes.size(); memoryIndex++)
{
     vk::MemoryPropertyFlags flags = memProps.memoryTypes[memoryIndex].propertyFlags;
     if (req.memoryTypeBits & (1 << memoryIndex) && (flags & props) == props)
          break;
}
if (memoryIndex == memProps.memoryTypes.size())
     throw std::runtime_error("No suitable memory found!");

vk::MemoryAllocateInfo allocInfo(req.size, memoryIndex);
vk::UniqueDeviceMemory bufferMemory = device->allocateMemoryUnique(allocInfo);
device->bindBufferMemory(*buffer, *bufferMemory, 0);
```

- Let’s do this more compactly: use the industry standard VMA

```
…

VkBufferCreateInfo buffCreate = vk::BufferCreateInfo({}, size, vk::BufferUsageFlagBits::eStorageBuffer);
VmaAllocationCreateInfo info = { 0, VMA_MEMORY_USAGE_UNKNOWN, VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT |
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT | VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT};

VkBuffer buffer;
VmaAllocation vmaAllocation;
vmaCreateBuffer(allocator, &buffCreate, &info, &buffer, &allocation, nullptr);

…

vmaDestroyBuffer(allocator, vmaBuffer, vmaAllocation);
vmaDestroyAllocator(allocator);
```

We lost RAII ☹

Example: Edge Detector (Allocation)

- This is as verbose as it gets for allocation in the exercises

```
VkBufferCreateInfo infoCreate = vk::BufferCreateInfo({}, 2*sizeof(uint32_t), vk::BufferUsageFlagBits::eUniformBuffer),
srcCreate = vk::BufferCreateInfo({}, width*height*sizeof(uint32_t), vk::BufferUsageFlagBits::eStorageBuffer),
dstCreate = vk::BufferCreateInfo({}, width*height*sizeof(uint32_t), vk::BufferUsageFlagBits::eStorageBuffer);
VmaAllocationCreateInfo allocationInfo = { 0, VMA_MEMORY_USAGE_UNKNOWN, VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT |
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT | VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT };

VkBuffer infoBuffer, srcBuffer, dstBuffer;
VmaAllocation infoAllocation, srcAllocation, dstAllocation;

vmaCreateBuffer(allocator, &infoCreate, &allocationInfo, &infoBuffer, &infoAllocation, nullptr);
vmaCreateBuffer(allocator, &srcCreate, &allocationInfo, &srcBuffer, &srcAllocation, nullptr);
vmaCreateBuffer(allocator, &dstCreate, &allocationInfo, &dstBuffer, &dstAllocation, nullptr);
…
vmaDestroyBuffer(allocator, infoBuffer, infoAllocation);
vmaDestroyBuffer(allocator, srcBuffer, srcAllocation);
vmaDestroyBuffer(allocator, dstBuffer, dstAllocation);
vmaDestroyAlloactor(allocator);
```

- Descriptor sets remain a little more verbose

Example: Edge Detector (Descriptors)

```
std::vector<vk::DescriptorPoolSize> sizes = {
{vk::DescriptorType::eUniformBuffer, 1},
{vk::DescriptorType::eStorageBuffer, 2}
};
vk::DescriptorPoolCreateInfo poolCreateInfo(vk::DescriptorPoolCreateFlagBits::eFreeDescriptorSet, 1, sizes);
auto descriptorPool = device->createDescriptorPoolUnique(poolCreateInfo);

std::vector<vk::DescriptorSetLayoutBinding> bufBindings = {
{0, vk::DescriptorType::eUniformBuffer, 1, vk::ShaderStageFlagBits::eCompute},
{1, vk::DescriptorType::eStorageBuffer, 1, vk::ShaderStageFlagBits::eCompute},
{2, vk::DescriptorType::eStorageBuffer, 1, vk::ShaderStageFlagBits::eCompute},
};
auto descriptorSetLayout = device->createDescriptorSetLayoutUnique(vk::DescriptorSetLayoutCreateInfo({}, bufBindings));
vk::DescriptorSetAllocateInfo descAllocInfo(*descriptorPool, *descriptorSetLayout);
vk::UniqueDescriptorSet descriptorSet = std::move(device->allocateDescriptorSetsUnique(descAllocInfo)[0]);

vk::DescriptorBufferInfo infoDesc(infoBuffer, 0, VK_WHOLE_SIZE);
vk::DescriptorBufferInfo srcDesc(srcBuffer, 0, VK_WHOLE_SIZE);
vk::DescriptorBufferInfo dstDesc(dstBuffer, 0, VK_WHOLE_SIZE);

std::vector<vk::WriteDescriptorSet> writes = {
{*descriptorSet, 0, 0, vk::DescriptorType::eUniformBuffer, {}, infoDesc},
{*descriptorSet, 1, 0, vk::DescriptorType::eStorageBuffer, {}, srcDesc},
{*descriptorSet, 2, 0, vk::DescriptorType::eStorageBuffer, {}, dstDesc}
};
```

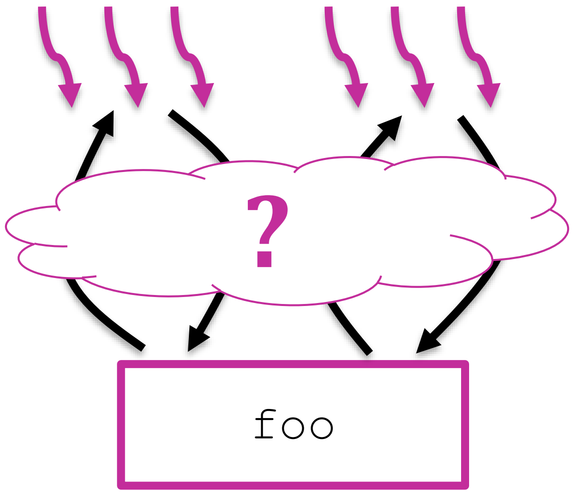
A New Hope?

- Descriptor remain some of the most verbose parts
- New extension `VK_EXT_descriptor_buffer`
- Allows you to treat descriptor set like a regular buffer
- Setting and updating descriptors becomes a `memcpy`
- Needs cutting-edge driver, detailed setup → not yet in core

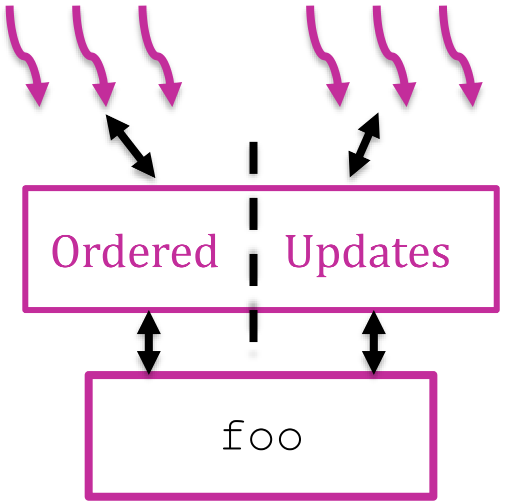
Atomic Operations

- Updates to the same memory problematic with many threads
  - Read/write may occur in arbitrary order, simultaneously, overlap?
  - Atomic operations are indivisible, visible and occur in some sequential order
  - Atomics where return value is unused are often just *reductions*

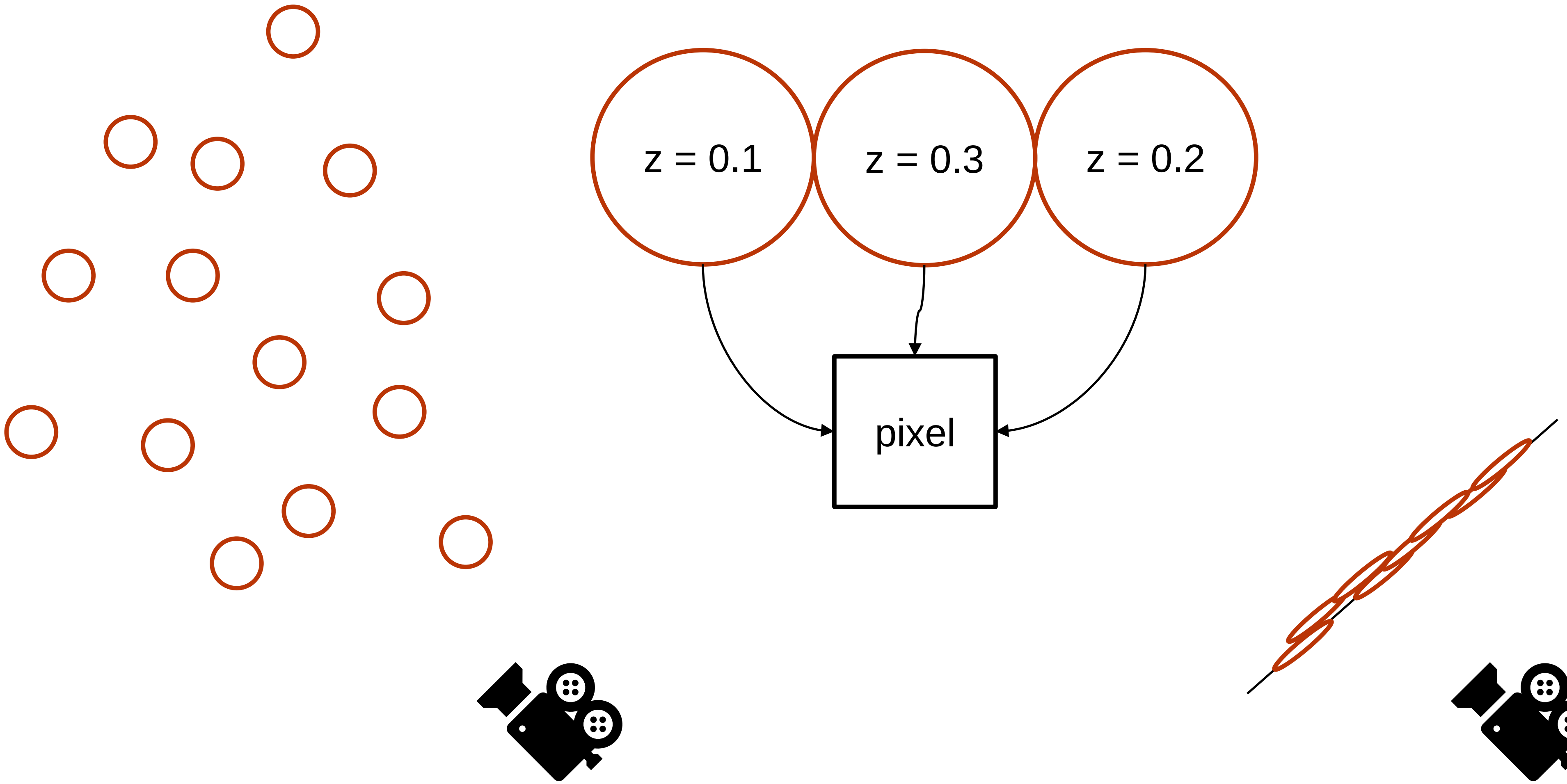
```
foo += bar;
```



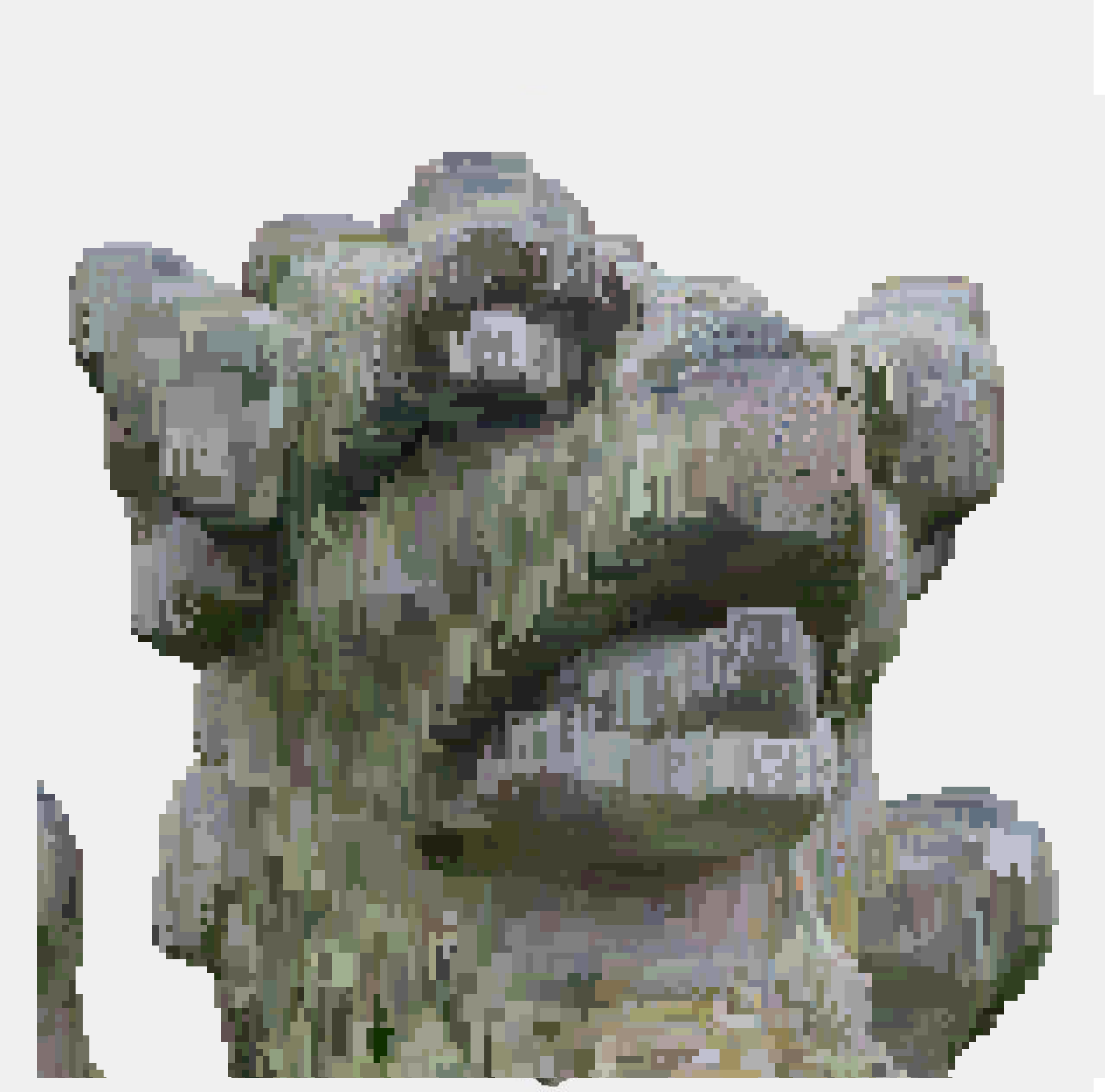

```
atomicAdd(&foo, bar);
```



Example: Atomics for Point Clouds

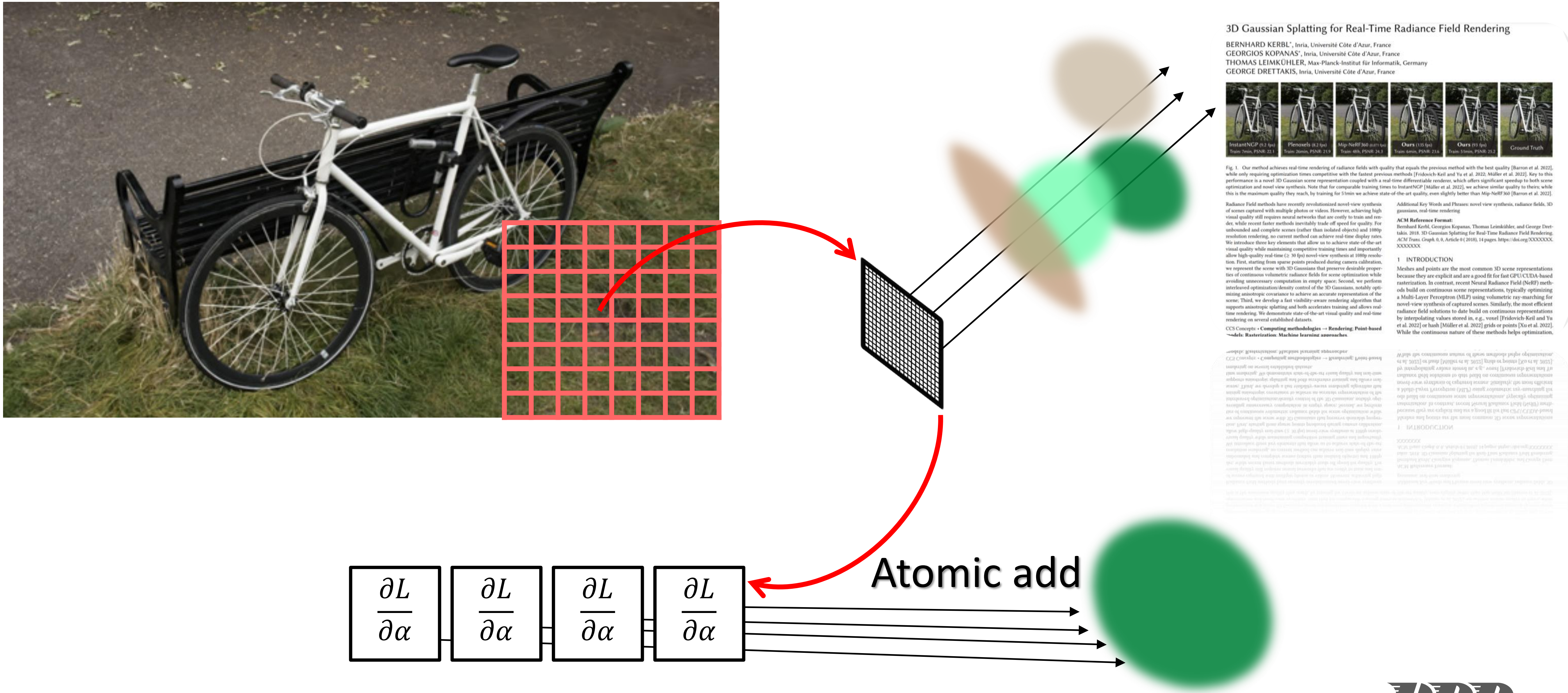


Correct vs Incorrect: Sometimes Subtle

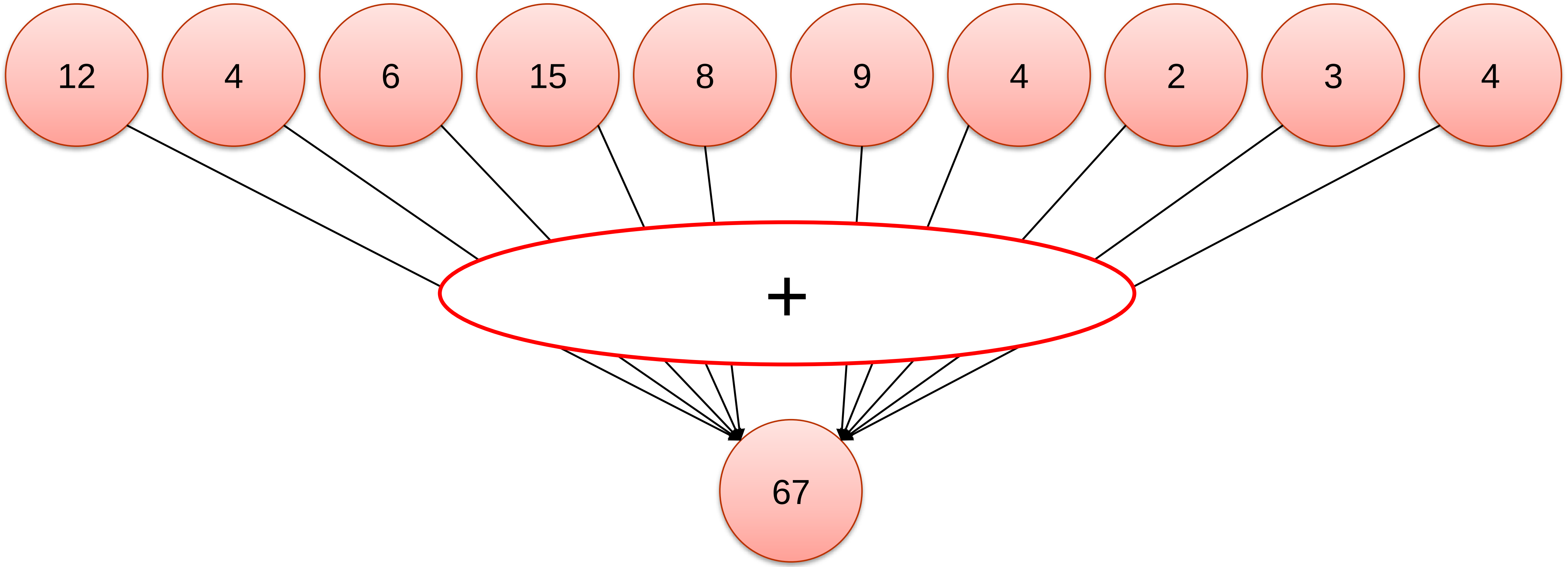


Stone Griffin, Downing College, Cambridge by Thomas Flynn, software-rendered. CC 4.0

Example: Atomics for Gradients



Use Case: Reduction



Example: Using Atomics for Reduction

```
#version 450

#extension GL_EXT_shader_atomic_float : require

layout (std430, set = 0, binding = 0) buffer InputBuff
{
    float data[];
} incoming;

layout (std430, set = 0, binding = 1) buffer OutputBuff
{
    float result;
} outgoing;

layout(local_size_x = 128, local_size_y = 1, local_size_z = 1) in;
void main()
{
    uint id = gl_GlobalInvocationID.x;
    atomicAdd(outgoing.result, incoming.data[id]);
}
```

20M floats

CPU Baseline:
63 ms

GPU Baseline:
85 ms

- *You could do this also in a fragment shader*. Why is it bad?

Exploiting Collaboration

- Many algorithms are embarrassingly parallel (e.g., ray tracing)
  - Each thread can work completely independently, no communication
  - Even a direct port to the GPU may accelerate them out of the box
  - Reduction is not one of them!

- If developers know how threads collaborate, more opportunities
  - The GPU is at its most powerful when it can reuse partial results
  - Cache utilization, **shared memory** and subgroup primitives
  - Competitive algorithms for sorting, reducing, analyzing or filtering

Example: Shared Memory & Reduction

```
shared float result_shared;

layout(local_size_x = 128, local_size_y = 1, local_size_z = 1) in;
void main()
{
    uint id = gl_GlobalInvocationID.x;

    if (gl_LocalInvocationID.x == 0)
        result_shared = 0;

    atomicAdd(result_shared, incoming.data[id]);

    if (gl_LocalInvocationID.x == 0)
        atomicAdd(outgoing.result, result_shared);
}
```

CPU Result:
42 (correct)

GPU Result:
35 (wrong)

- Smart!

```
shared float result_shared;

layout(local_size_x = 128, local_size_y = 1, local_size_z = 1) in;
void main()
{
    uint id = gl_GlobalInvocationID.x;

    if (gl_LocalInvocationID.x == 0)
        result_shared = 0;

    atomicAdd(result_shared, incoming.data[id]);

    if (gl_LocalInvocationID.x == 0)
        atomicAdd(outgoing.result, result_shared);
}
```

CPU Result:
42 (correct)

GPU Result:
35 (wrong)

- Smart! …so why doesn’t it work?

Race Conditions

```
shared float result_shared;

layout(local_size_x = 128, local_size_y = 1, local_size_z = 1) in;
void main()
{
    uint id = gl_GlobalInvocationID.x;

    if (gl_LocalInvocationID.x == 0)
        result_shared = 0;

    atomicAdd(result_shared, incoming.data[id]);

    if (gl_LocalInvocationID.x == 0)
        atomicAdd(outgoing.result, result_shared);
}
```

RAW hazard

RAW hazard

- Compute enables powerful ways to communicate in GPU memory!
  - But with great power comes great responsibility…

Multi-threading and Memory

- Memory consistency is key for exchanging data between threads
- Compilers **cannot know** how threads will interact at runtime
- You must protect accesses to shared data, otherwise: data race!
- **x86 architectures**: maybe a dozen threads, forgiving memory model
- **GPU**: thousands of threads, nothing is forgiven!

Memory Consistency Models

- Are you comfortable (or even aware) of
  - Memory fences?
  - Relaxed atomic operations?
  - Release and acquire semantics?
  - Bonus points if you can explain “out-of-thin-air” (OOTA) values

- Vulkan defines a C++11-style memory consistency model
  - Modern GPUs **can** do atomics with release/acquire semantics
  - You can go back to not caring about OOTA values (explicitly banned)

Resource Tip: Memory Models

- “And what is a C++11-style memory model now?”
- Knowing the concepts is helpful for **anything** that’s multithreaded
- To get started, I personally recommend
  - Anthony Williams, “C++ Concurrency in Action”
  - Herb Sutter’s two-part talk “Atomic Weapons”

Device-side Synchronization: Barrier

- The threads in a work group are comparably cheap to synchronize
- `barrier()` is both an **availability** and **visibility** operation
- Its **scopes** include the entire work group
- That means, after it, any changes to memory made by a thread in the group before it will be visible to all other threads in the group

Reduction complete!

```
shared float result_shared;

layout(local_size_x = 128, local_size_y = 1, local_size_z = 1) in;
void main()
{
    uint id = gl_GlobalInvocationID.x;

    if (gl_LocalInvocationID.x == 0)
        result_shared = 0;
    barrier();

    atomicAdd(result_shared, incoming.data[id]);
    barrier();

    if (gl_LocalInvocationID.x == 0)
        atomicAdd(outgoing.result, result_shared);
}
```

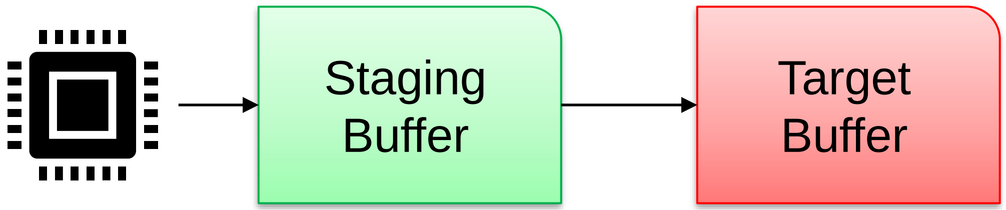
CPU Result:
42 (correct)

GPU Result:
42 (correct)

- Note: your experience might vary, less or more work to beat CPU

Host-side Synchronization : Staging

- Host-visible device memory is cool, but might have drawbacks
  - Might not be compatible with particular resources (type mismatch)
  - Probably limited (unless you have resizable BAR)
  - Might be slower for some operations



- Large data or repetitive use? A case for using a “staging buffer”
  - Host-visible memory usually supports copying from/to
  - Can create a slow, non-device-local, host-visible staging buffer A
  - Can create a fast, device-local, non-host-visible buffer B
  - Write from CPU to staging buffer A, then copy from there to B

Host-side Synchronization: Staging

- How? Simple, just drop the `MEMORY_PROPERTY_DEVICE_LOCAL`
- Let VMA figure out the rest
- Note: this buffer does NOT go in your descriptor sets!
- Don’t forget that copying between buffers should be synchronized

```
vk::MemoryBarrier copyBarrier(vk::AccessFlagBits::eTransferWrite, vk::AccessFlagBits::eShaderRead);

cmdBuffers[0]->pipelineBarrier(vk::PipelineStageFlagBits::eTransfer,
vk::PipelineStageFlagBits::eComputeShader, {}, copyBarrier, {}, {});
```

Correct Use of Synchronization

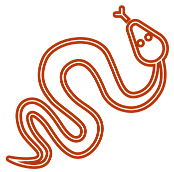
- Synchronization is tough
  - Except, not really: can use wooden mallets do get what you want
  - E.g., `vkDeviceWaidIdle`, writing one shader for every task
  - The most important point is that the program is **correct**

- ***Efficient*** synchronization is tough
  - Avoiding races with minimum overhead requires solid knowledge
  - A lot of work by community to improve it, practice makes perfect
  - Do you need to be that efficient? **Probably not!**

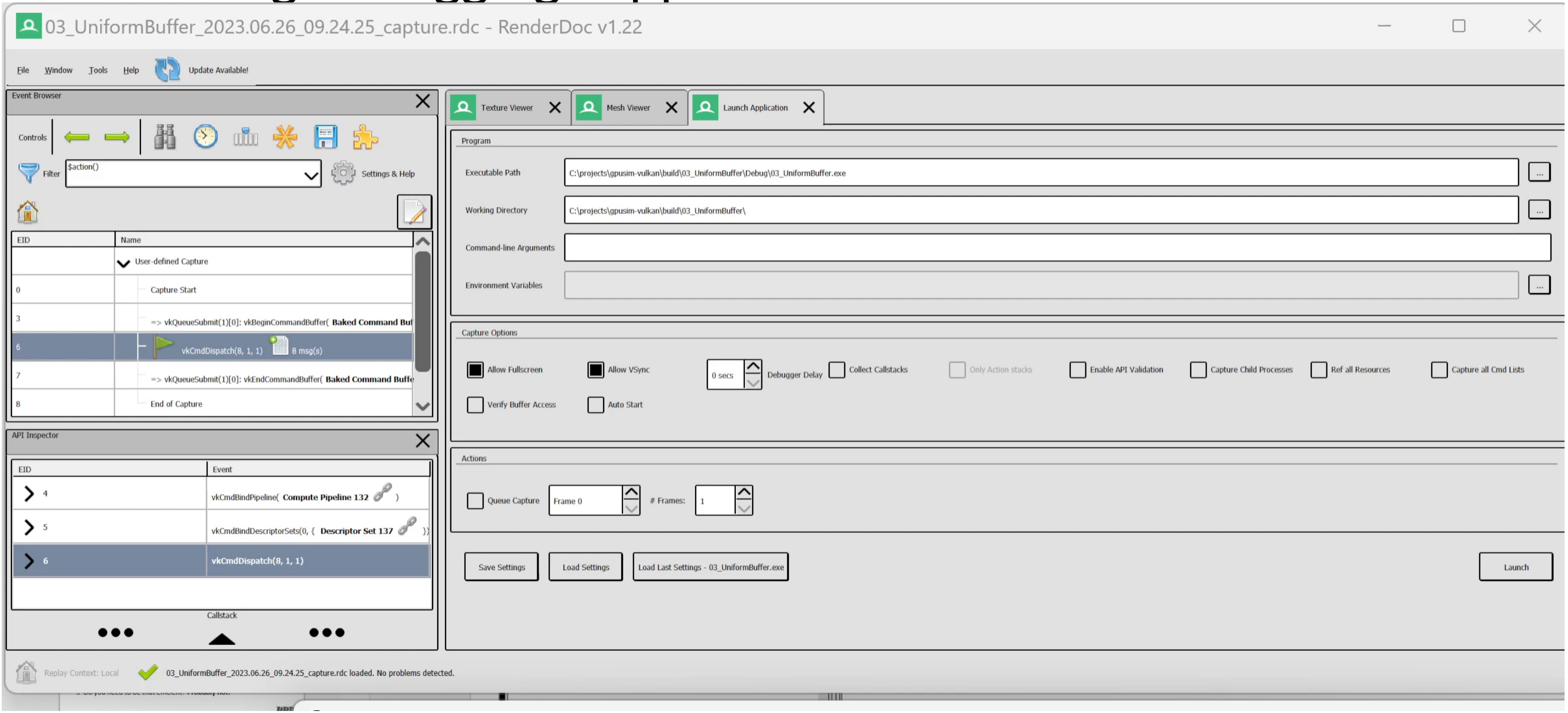
Profiling

- Very advanced profiling solutions exist. To get the most features, seek out vendor-specific *RGP* and *Nsight Graphics/Systems*

- Common mistake: over-optimizing your GPU code
  - In this room, mostly students and researchers
  - Be sure you understand the impact of your overall system
  - Never try to raise performance by improving what you THINK is slow
  - Training with PyTorch and with custom extensions that use the GPU? It’s probably unoptimized Python that’s slowing you down…



- Increasing debugging support across vendors with RenderDoc



Vulkan Compute Challenges

- General-purpose compute shaders can quickly become complex; ensuring performance is often key to success

- Great profiling with *RGP* and *Nsight Graphics*, **but limitations apply**
  - E.g., *Nsight Graphics* does not expose machine code instructions
  - Professional edition and agreement required

- *RenderDoc* provides basic debugging capabilities
  - Perfect for fixing arithmetic logic or data alignment errors
  - Not yet fully equivalent to the debuggers you have for *ROCm/CUDA*

Vulkan Compute Challenges

- A complete, C++11-like memory consistency model (shared and global), **but** currently no standardized forward progress guarantee
  - In development, many GPUs already seem to fulfill it

- Supports wide range of cutting-edge, vendor-specific features, **but** some popular ones missing (e.g., dynamic parallelism)
  - Available extensions often influenced by demand and vendor policies

- Useful (and fast!) libraries are starting to show up (e.g., *vkFFT)*
  but other APIs offer auxiliary libraries for containers, BLAS, AI…

Thanks!

- All those involved in the (long-term) process of developing material
  - Johannes Unterguggenberger and various students from TU Wien
  - Markus Steinberger, Michael Kenzel

- **High-Performance Graphics**

- The Khronos Group and its Community
  - Tim Lewis, for establishing connections and encouraging actions
  - Too many to name here! Check out the presenters of **Vulkanised** conference to get an idea of the people making contributions