codeplay®
THE HETEROGENEOUS SYSTEMS EXPERTS

OpenCL -> Vulkan: A Porting Guide

Neil Henning

Munich Khronos Chapter Meeting - October 13th 2017

We'll cover:

- Why port from OpenCL -> Vulkan?
- Common problems when porting:
  - How does interfacing with multiple vendor drivers work?
  - How to allocate buffers & images?
  - Command queues are not like Vulkan's queues!
  - How to synchronize?
  - How to pass in kernels/shaders?
  - How to specify data to use in a kernel/shader?
  - How to read a buffer?
  - How to port OpenCL C kernels to Vulkan?

Why port from OpenCL -> Vulkan?

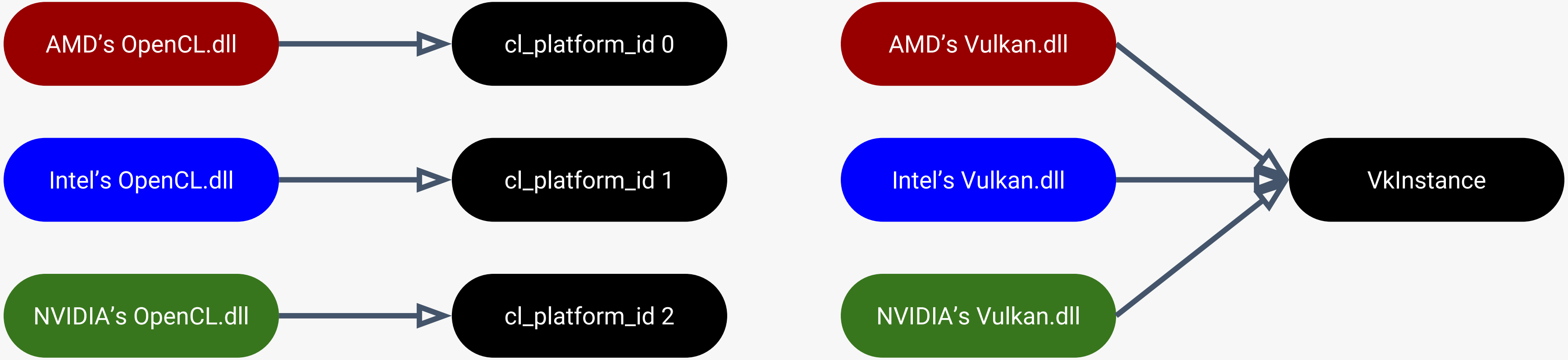
- Vulkan drivers for Windows/Linux/Android
- Vulkan is the **primary** API for graphics & compute on Android 7.0 onwards
- Great tooling for Vulkan (RenderDoc, Radeon GPU Profiler, Khronos Validation Layers, Khronos SPIR-V tools, etc)
- Vulkan SPIR-V is guaranteed to be available on all versions (unlike OpenCL SPIR-V where support is patchy)

Why port from OpenCL -> Vulkan?

- The Vulkan API is **significantly faster**
  - We’ve seen 3x improvement in just the API
  - The Validation Layers enable expensive checking to exist outside drivers
- Vulkan’s Validation Layers
  - Developed at a separate cadence to vendor drivers
  - Essentially gives you more frequent ‘driver’ updates on Vulkan
  - Code is open source so you can build yourself and debug!
- Vulkan drivers are updated much more frequently!
  - We’ve inherited the good practices from the graphics world to also benefit compute!

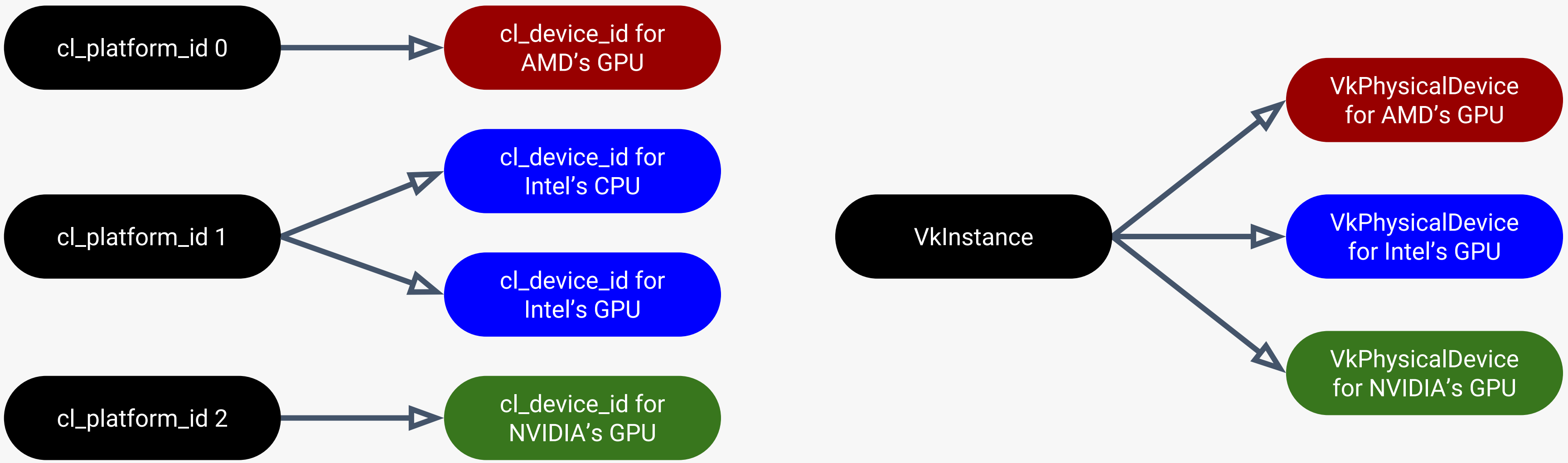
How does interfacing with multiple vendor drivers work?

- OpenCL - 1 or more platforms are 1:1 with vendor drivers
- Vulkan - an instance interfaces with any devices in all drivers



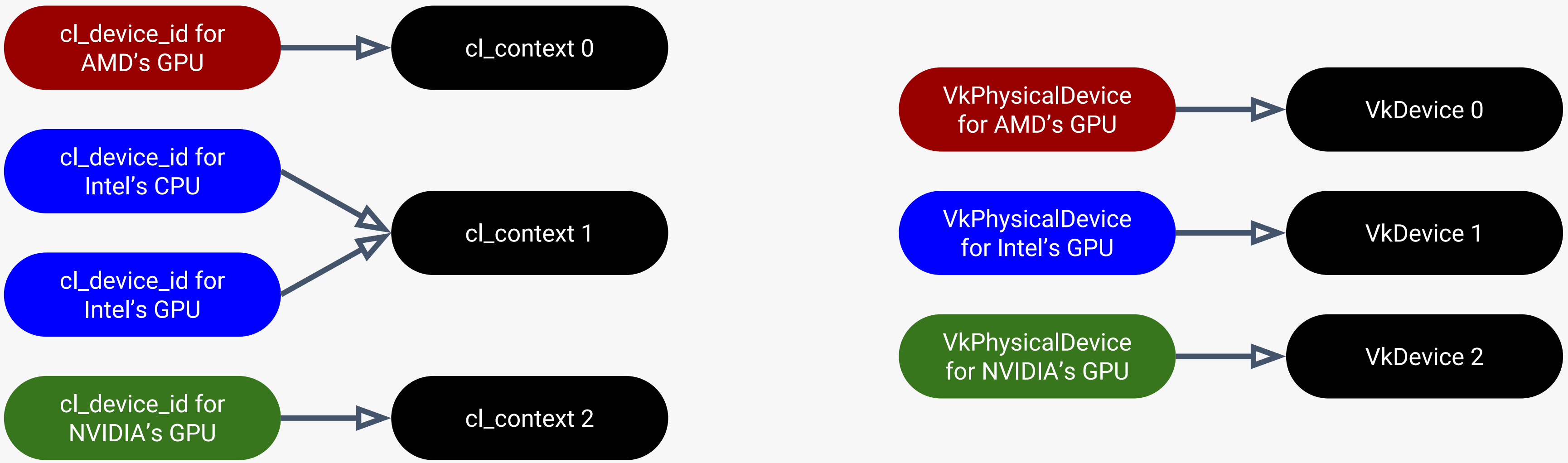
How does interfacing with multiple vendor drivers work?

- OpenCL - each platform has N devices
- Vulkan - an instance has N physical devices



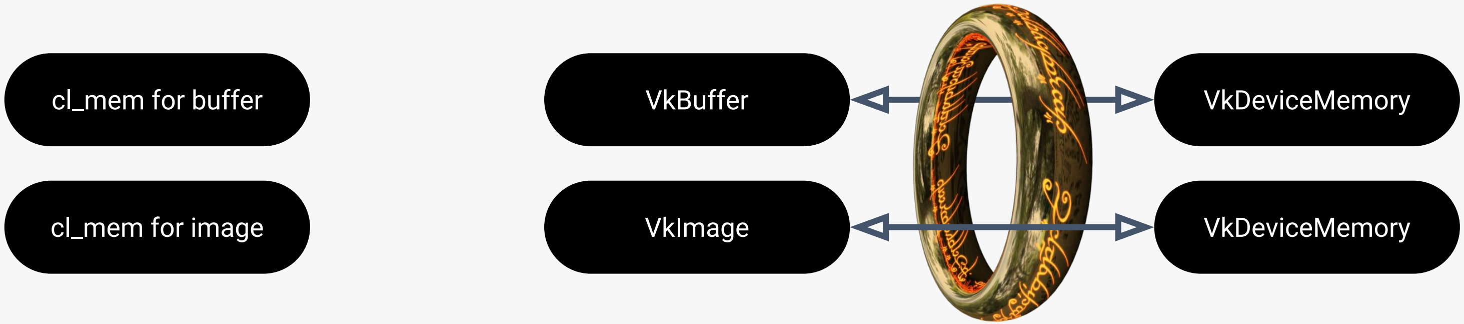
How does interfacing with multiple vendor drivers work?

- OpenCL - a context is created from N devices from a platform
- Vulkan - a device is created from a single physical device



How to allocate buffers & images?

- OpenCL - allocate buffer/image and get a ‘memory object’
- Vulkan - create buffer/image, allocate memory, bind them



OpenCL -> Vulkan: A Porting Guide

```
cl_mem buffer = clCreateBuffer(context
  CL_MEM_READ_WRITE, 1024, nullptr,
  &errorcode);

CL_CHK(errorcode);
```

```
vk::QueueFamilyProperties p =
  physicalDevice.getQueueFamilyProperties();

std::vector<uint32_t> idxs;
for (uint32_t k = 0; k < p.size(); k++)
  if (vk::QueueFlagBits::eCompute &
    p[k].queueFlags)
    idxs.push_back(k);

vk::BufferCreateInfo info;
info.setSize(1024);
info.setUsage(
  vk::BufferUsageFlagBits::eTransferSrc |
  vk::BufferUsageFlagBits::eTransferDst |
  vk::BufferUsageFlagBits::eStorageBuffer);
info.setSharingMode(
  vk::SharingMode::eConcurrent);
info.setQueueFamilyIndexCount(idxs.size());
info.setPQueueFamilyIndices(idxs.data());

vk::Buffer buffer;
VK_CHK(device.createBuffer(info,
  nullptr, &buffer));
```

OpenCL -> Vulkan: A Porting Guide

```
// Vulkan can be a little verbose ;)
```

```
vk::MemoryRequirements requirements =
 device.getBufferMemoryRequirements(buffer);

vk::PhysicalDeviceMemoryProperties props =
  physicalDevice.getMemoryProperties();

uint32_t typeIndex = 0;

vk::MemoryPropertyFlags flags =
  vk::MemoryPropertyFlagBits::eHostVisible;

for (uint32_t k = 0;
  k < props.memoryTypeCount; k++) {
  vk::MemoryType ty = props.memoryTypes[k];

  const vk::DeviceSize heapSize =
    props.memoryHeaps[ty.heapIndex].size;

  if ((flags & ty.propertyFlags) &&
    (requirements.memoryTypeBits & (1 << k))
    && (heapSize >= requirements.size))
    typeIndex = k;
}
```

OpenCL -> Vulkan: A Porting Guide

```
// Vulkan can be a little verbose ;)
```

```
vk::MemoryAllocateInfo info;

info.setAllocationSize(requirements.size);
info.setMemoryTypeIndex(typeIndex);

vk::DeviceMemory mem;

VK_CHK(device.allocateMemory(
  info, nullptr, &mem));

VK_CHK(device.bindBufferMemory(
  buffer, memory, 0));
```

How to allocate buffers & images?

- Got to be **careful** with Vulkan image requirements
- TL;DR you can't map images across vendors reliably
  - The vk::MemoryRequirements of an image is allowed to restrict an image from host-visible memory
- Need to use staging buffers + copy-image-to-buffer + copy-buffer-to-image

Command queues are not like Vulkan's queues!

- Three parts to running things on a compute device:
  a. 1 or more commands you want to run
  b. Physical hardware to run on
  c. How to synchronize on device and with CPU
- OpenCL uses cl_command_queue's for a. & b., & cl_event's for c.
- Vulkan uses VkCommandBuffer's for a., VkQueue's for b., & VkFence's, VkSemaphore's, & VkEvent's for c.
- OpenCL *creates* command queues whereas Vulkan *creates* command buffers and **gets** queues from a device

OpenCL -> Vulkan: A Porting Guide

```
cl_int errorcode;

cl_command_queue command_queue =
  clCreateCommandQueue(context, device,
  CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE,
  &errorcode);

CK_CHK(errorcode);
```

```
uint32_t queueFamilyIndex = UINT32_MAX;

for (uint32_t i = 0; i < props.size(); i++)
  if (vk::QueueFlagBits::eCompute &
    props[i].queueFlags) {
    queueFamilyIndex = i;
    break;
  }
}

if (UINT32_MAX == queueFamilyIndex) {
  // ... error!
}

vk::Queue queue = device.getQueue(
  queueFamilyIndex, 0);
```

OpenCL -> Vulkan: A Porting Guide

```
// Vulkan can be a little verbose ;)
```

```
vk::CreateCommandPoolInfo info;

info.setQueueFamilyIndex(queueFamilyIndex);

vk::CommandPool commandPool;

VK_CHK(device.createCommandPool(
  &info, nullptr, &commandPool));

vk::CommandBufferAllocateInfo cbInfo;

cbInfo.setCommandPool(commandPool);
cbInfo.setLevel(
  vk::CommandBufferLevel::ePrimary);
cbInfo.setCommandBufferCount(1);

vk::CommandBuffer commandBuffer;

VK_CHK(commandPool.allocateCommandBuffers(
  &cbInfo, &commandBuffer));
```

How to synchronize?

- For all synchronization in OpenCL you use cl_event’s
- For Vulkan there are 3 methods of synchronization:
  - VkEvent’s within command buffers on a single queue
  - VkSemaphore’s between command buffers across all queues of a device
  - VkFence’s for the host CPU to wait on command buffers in flight on a device
- Both OpenCL and Vulkan allow you to wait on all commands in a queue to finish executing
- Vulkan also allows you to wait for an entire device to be idle!

OpenCL -> Vulkan: A Porting Guide

```
// Vulkan can be a little verbose ;)
```

```
vk::CreateFenceInfo fInfo;

vk::Fence fence;

VK_CHK(device.createFence(
  &fInfo, nullptr, &fence));

vk::CreateSemaphoreInfo sInfo;

vk::Semaphore semaphore;

VK_CHK(device.createSemaphore(
  &sInfo, nullptr, &semaphore));

vk::CreateEventInfo eInfo;

vk::Event event;

VK_CHK(device.createEvent(
  &eInfo, nullptr, &event));
```

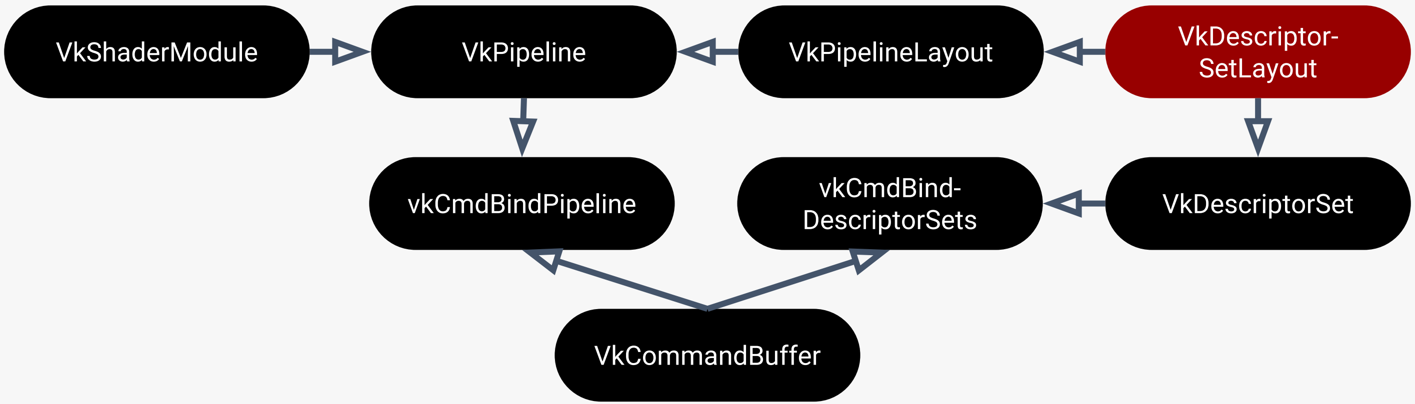
OpenCL -> Vulkan: A Porting Guide

```
uint32_t data = 42;

cl_event event;

CL_CHK(clEnqueueFillBuffer(
  command_queue, buffer, &data,
  sizeof(data), 0, 1024, 0, nullptr,
  &event));

// Will start any previous commands
// running
CL_CHK(clFlush(command_queue));

// Waiting for an event that is set
// from a previously enqueued command
// will start it running
CL_CHK(clWaitForEvents(1, &event));

// Or we can start the commands and wait
// on them all to finish
CL_CHK(clFinish(command_queue));
```

```
VK_CHK(commandBuffer.begin());
commandBuffer.fillBuffer(
  buffer, 0, 1024, 42);
commandBuffer.setEvent(event,
  vk::PipelineStageFlagBits::eAllCommands);
VK_CHK(commandBuffer.end());

vk::SubmitInfo submitInfo;
submitInfo.setCommandBufferCount(1);
submitInfo.setPCommandBuffers(
  &commandBuffer);

VK_CHK(queue.submit(1, &submitInfo, fence));

vk::Result status;
do {
  status = device.getEventStatus(event);
} while(vk::Result::eEventUnset == status);

VK_CHK(device.waitForFences(1, &fence,
  true, UINT64_MAX));

VK_CHK(device.waitIdle());
```

How to pass in kernels/shaders?

- The big gotcha coming from OpenCL to Vulkan is that it is **your** responsibility to explain the kernel interface
  - When creating a pipeline (the function to run on the device)
  - When creating a descriptor set (arguments to run with the function)



How to specify data to use in a kernel/shader?

- How data is passed to shaders differs vastly
  - OpenCL you set the arguments a kernel should execute with
  - Vulkan you create descriptor sets and then **bind** them during command buffer recording
  - Vulkan allows for the data used by a shader to change much more dynamically as a result

How to read a buffer?

- OpenCL has clEnqueueReadBuffer
  - Allows a buffer to be read and written to host memory in a command queue
- Vulkan has no equivalent construct!
  - Re-architect to map the memory and read that instead
- If you were reading a buffer then immediately reusing the buffer in the command queue
  - Use a temporary buffer
  - Copy the original buffer to the temporary (using vkCmdCopyBuffer)
  - Then map the temporary and read that instead

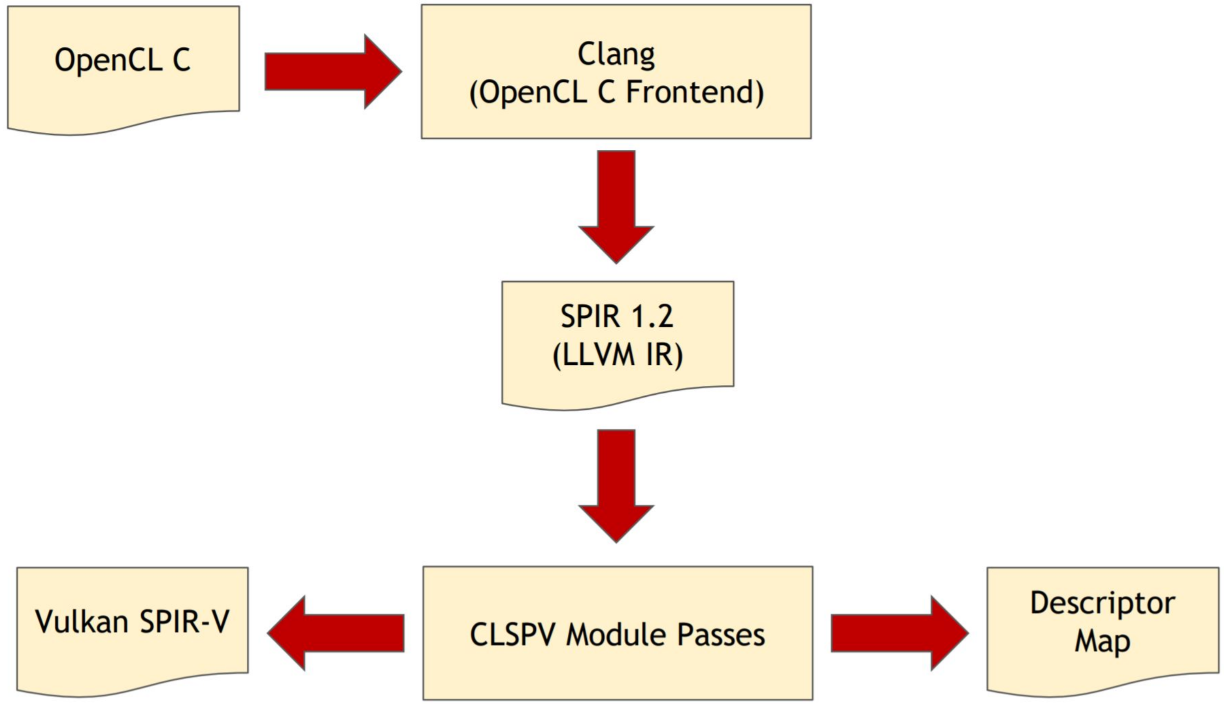
How to port OpenCL C kernels to Vulkan?

- OpenCL consumes OpenCL C language kernels
- Vulkan consumes SPIR-V
- How do we marry these?
    - We don’t want to rewrite thousands of lines into GLSL or HLSL
    - Want to keep supporting OpenCL too...
    - Use clspv - https://github.com/google/clspv

clspv

- Adobe/Codeplay/Google collaboration to port a subset of OpenCL C to Vulkan SPIR-V
- We've now ported ~1 million lines of production OpenCL C to Vulkan
- I've hoisted two slides by my colleague Ralph Potter from his Siggraph talk to show how the tool works

CLSPV Compiler



Example

```
kernel
void interleave(global float *dst,
                global float *src_a,
                global float *src_b)
{
  int id = get_global_id(0);

  global float *src =
    (id % 2) ? src_a : src_b;
  dst[id] = src[id / 2];
}
```

```
// Pointers to StorageBuffer src_a, src_b
%28 =   OpAccessChain %2 %24 %14 %14
%29 =   OpAccessChain %2 %25 %14 %14
// Load GlobalInvocationId
%30 =   OpAccessChain %11 %17 %14
%31 =   OpLoad %6 %30
// Src = (GlobalInvocationId & 1 == 0) ?
//           src_b : src_a
%32 =   OpBitwiseAnd %6 %31 %15
%33 =   OpIEqual %12 %32 %14
// Dynamically select between two pointers
%34 =   OpSelect %2 %33 %29 %28
// Load Src[GlobalInvocationId / 2]
%35 =   OpSDiv %6 %31 %16
%36 =   OpPtrAccessChain %2 %34 %35
%37 =   OpLoad %1 %36
// Store Dst[GlobalInvocationId]
%38 =   OpAccessChain %2 %23 %14 %31
        OpStore %38 %37
        OpReturn
```

clspv

- Some limitations
  - Doesn’t support 8 & 16 wide vector types
  - Vulkan doesn’t support 8-bit types (we have to emulate them)
  - Some OpenCL C built-ins are not supported
  - All restrictions documented here: https://github.com/google/clspv/blob/master/docs/OpenCLCOnVulkan.md

Conclusion:

- Good time to port to Vulkan
  - Android support
  - **3x** performance
  - Explicit APIs give huge control to **you**
- Some niggles to work through
  - Mapping images
  - Synchronization is more complex
- Biggest issue (porting shaders) solved with clspv
  - No need to rewrite OpenCL C kernels into GLSL or HLSL!

codeplay®

THE HETEROGENEOUS SYSTEMS EXPERTS

Questions?

@sheredom

neil@codeplay.com

codeplay.com