SYCL Interoperability with DirectX and Vulkan via Bindless Images

Duncan Brawley, Codeplay

Duncan Brawley, Przemek Malon, Jack Kirk, Georgi Mirazchiyski, Peter Žužek, and Gordon Brown

Disclaimer – This is an experimental extension and so is subject to change

IWOCL 2024 Bindless Images presentation and slides

- Video Presentation:
  - https://www.youtube.com/watch?v=KfxiFRw3yAA

- Slides:
  - https://www.iwocl.org/wp-content/uploads/9301-Sean-Stirling-Codeplay.pdf

Agenda

- Intro
    - Overview of Bindless Images
    - Brief catchup of new features
- Importing Vulkan/DX12 memory into SYCL
- Exporting SYCL memory into Vulkan/DX12
- Importing Vulkan/DX12 semaphores into SYCL
- Exporting semaphores from SYCL into Vulkan/DX12
- Problems encountered and interop as separate extension
- Q&A

Motivation of Bindless Images

- SYCL 2020 images has too many limitations
  - DPC++ implementation not using texture hardware efficiently
  - Need to request access through accessors
  - Number of images must be known at compile time
  - No control over how images are stored on the device (layouts, encodings, USM, etc)
  - No mipmaps or cubemaps
  - No interop with graphics APIs

Highlights of Bindless Images

- Separation of image memory and the actual image object
  - Can use device-optimized memory layout, USM allocations from SYCL, or imported memory
  - RAII wrappers
- Images as opaque handles
  - No accessors required, vary number of images at runtime
- Flexible copy functions and flexible on-device access
  - Many options for copying and reinterpreting image data
- Additional image types
  - Mipmaps, cubemap, image arrays, etc.

New features since IWOCL 2024

- Explicit `fetch_image`, `sample_image`, `sample_mipmap`, etc. naming
- Sampled image arrays
- Extended image copies (device to device, image arrays, sub-copies)
- USM host image memory and copies
- Vulkan mipmap interop
- Limited 3 channel image support (Level Zero only)
- `gather_image` to get values used for linear interpolation

Backend Support

- CUDA Backend
  - Full Support – Everything in the bindless spec is implemented
- Level Zero Backend
  - Partial Support – 1-2-3D images, sampling, USM images, image arrays, 3-channel images
- HIP Backend
  - Basic support - 1-2-3D images, sampling

Blender using Bindless Images

- Initial changes have been pushed to Blender to allow SYCL backend use of Bindless Images
- Not fully upstreamed yet
- Works on CUDA, Level Zero and HIP
- Effort has been taken to optimize as much as possible
- Not currently using interop features
- Has been covered in more detail in previous presentation

Blender is a registered trademark (®) of the Blender Foundation in EU and USA

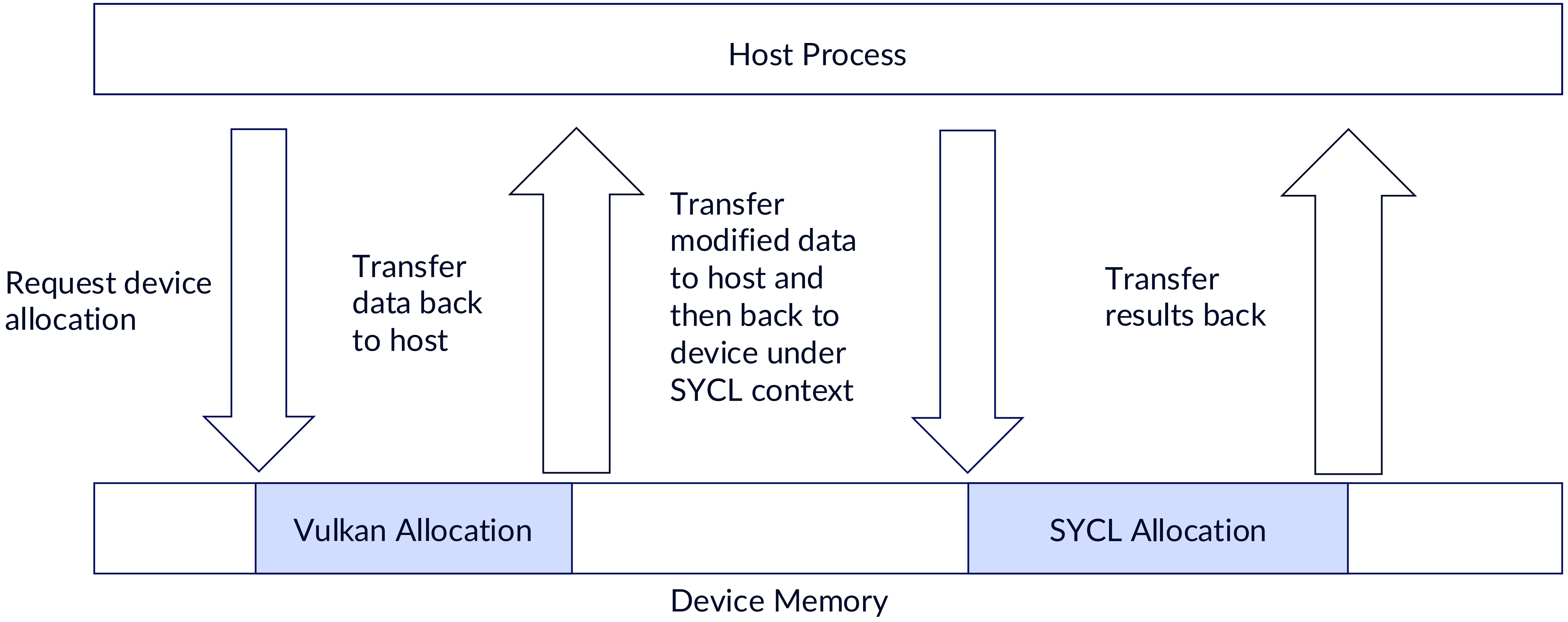
Importing Vulkan/DX12 memory into SYCL

Why is interop between SYCL and Vulkan/DX12 needed?

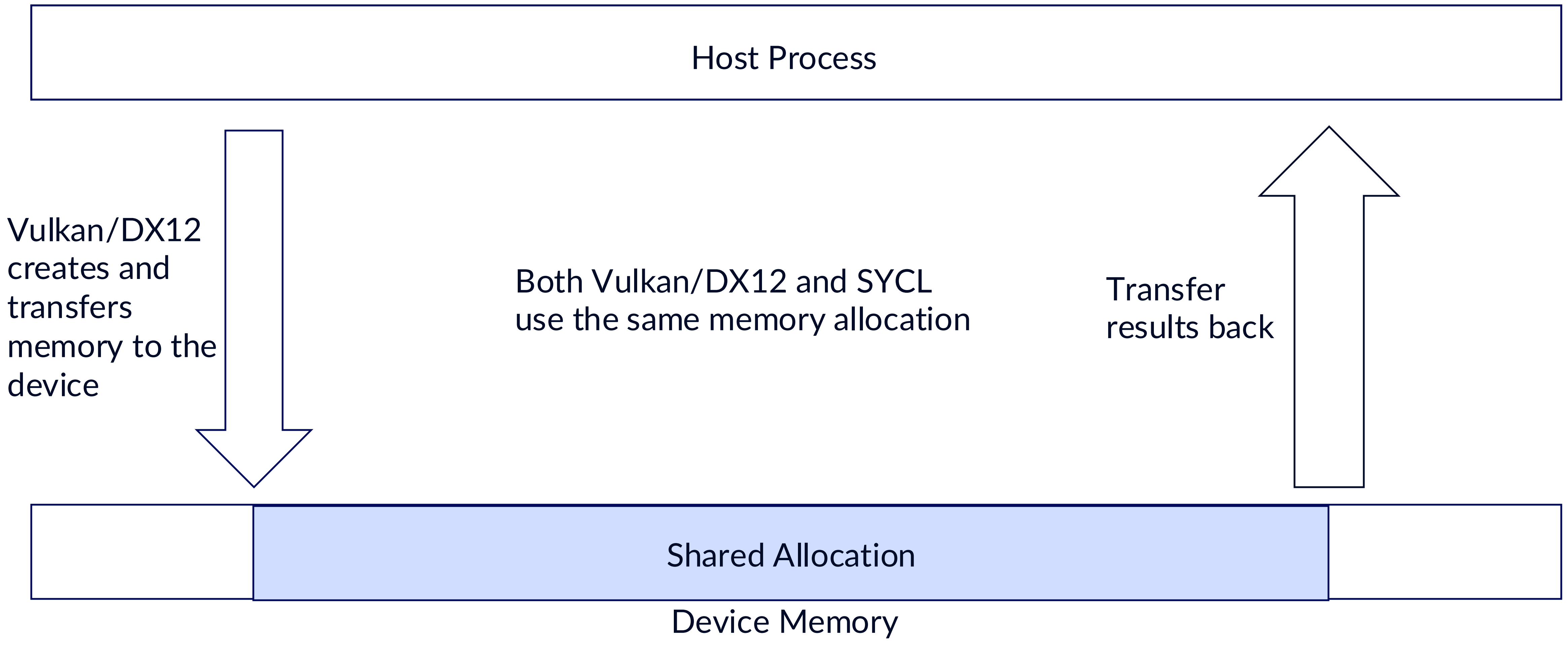
- No copies!
  - Otherwise, would need to introduce additional copies
- Easier leveraging of existing Vulkan/DX12 libraries in SYCL and vice versa
- We have designed the API to be generic and applicable to other external APIs

Vulkan is a registered trademark and the Vulkan Portability logo is a trademark of the Khronos Group Inc.

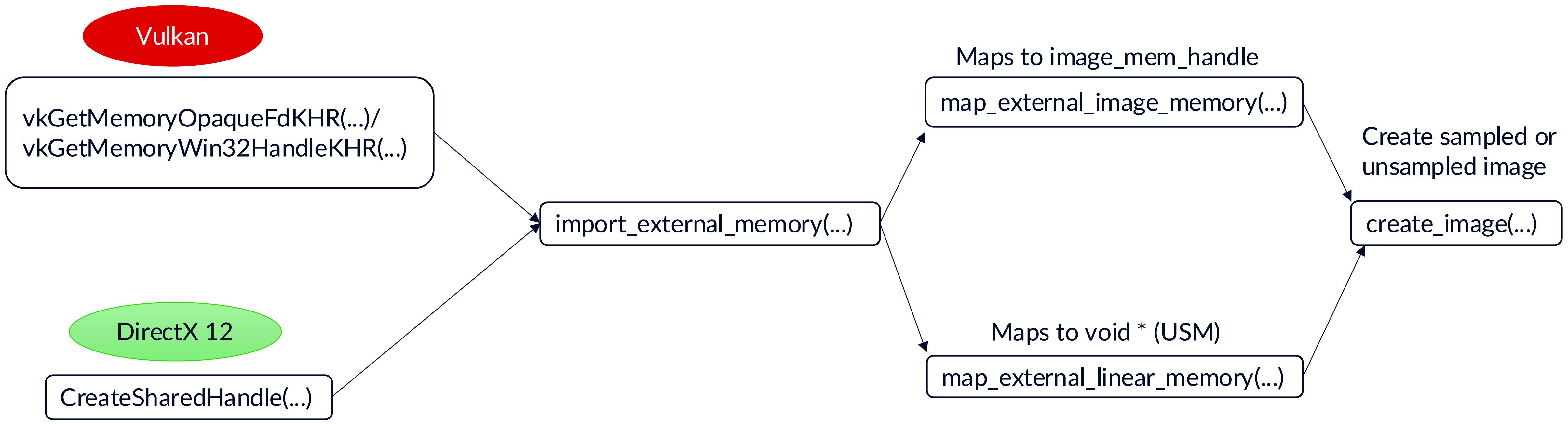
Without SYCL interop


Host Process
Request device allocation
Transfer data back to host
Transfer modified data to host and then back to device under SYCL context
Transfer results back
Vulkan Allocation
SYCL Allocation
Device Memory

With SYCL interop


Host Process
Vulkan/DX12 creates and transfers memory to the device
Both Vulkan/DX12 and SYCL use the same memory allocation
Transfer results back
Shared Allocation
Device Memory

Basic process of importing memory


Vulkan
vkGetMemoryOpaqueFdKHR(...)/
vkGetMemoryWin32HandleKHR(...)
DirectX 12
CreateSharedHandle(...)
import_external_memory(...)
Maps to image_mem_handle
map_external_image_memory(...)
Maps to void * (USM)
map_external_linear_memory(...)
Create sampled or
unsampled image
create_image(...)

Allocate and export Vulkan memory

Create memory in Vulkan

```
// Allocate memory in Vulkan
const size_t imgSize = numElems * sizeof(dataType) * NChannels;
VkDevice vulkanDevice = /* ... */;
VkDeviceMemory vulkanMemory = /* ... */;

// Export memory from Vulkan
#ifdef _WIN32
VkMemoryGetWin32HandleInfoKHR vulkanHandleInfo = /* ... */;
HANDLE vulkanMemHandle = INVALID_HANDLE_VALUE;
vkGetMemoryWin32HandleKHR(vulkanDevice, &vulkanHandleInfo, &vulkanMemHandle);
#else
VkMemoryGetFdInfoKHR vulkanHandleInfo = /* ... */;
int vulkanMemHandle = 0;
vkGetMemoryFdKHR(vulkanDevice, &vulkanHandleInfo, &vulkanMemHandle);
#endif
```

Export memory from Vulkan making it available for SYCL to import

Import Vulkan memory into SYCL

Describe what kind of memory is being imported

```
// Describe memory being imported
#ifdef _WIN32
syclexp::external_mem_descriptor<syclexp::resource_win32_handle> extMemDesc{
    vulkanMemHandle, syclexp::external_mem_handle_type::win32_nt_handle,
    imgSize};
#else
syclexp::external_mem_descriptor<syclexp::resource_fd> extMemDesc{
    vulkanMemHandle, syclexp::external_mem_handle_type::opaque_fd, imgSize};
#endif

// Import memory from Vulkan into SYCL
syclexp::external_mem externMem =
    syclexp::import_external_memory(extMemDesc, syclQueue);

// Map imported memory into SYCL image memory handle
syclexp::image_descriptor desc{imgSize, NChannels, channelType};
syclexp::image_mem_handle imgMemHandle =
    syclexp::map_external_image_memory(externMem, desc, syclQueue);

// Create SYCL image and use it as usual
syclexp::unsampled_image_handle imgHandle =
    syclexp::create_image(imgMemHandle, desc, syclQueue);
```

Import memory into SYCL

Create image memory handle from imported memory

Create image as usual

Allocate and export DirectX 12 memory

Create memory in DX12

```
// Allocate memory in DX12
ComPtr<ID3D12Device> dx12Device = /* ... */;
ComPtr<ID3D12Resource> dx12Texture = /* ... */;

// Export memory from DX12
HANDLE dx12SharedMemHandle = INVALID_HANDLE_VALUE;
dx12Device->CreateSharedHandle(dx12Texture.Get(), nullptr,
                               GENERIC_ALL, nullptr,
                               &dx12MemHandle);
```

Export memory from DX12 making it available for SYCL to import

Import DirectX12 memory into SYCL

Describe what kind of memory is being imported

Create image memory handle from imported memory

```
// Describe memory being imported
syclexp::external_mem_descriptor<syclexp::resource_win32_handle> extMemDesc{
    dx12MemHandle,
syclexp::external_mem_handle_type::win32_nt_dx12_resource,
    dx12TexAllocInfo.SizeInBytes};

// Import memory from Vulkan into SYCL
syclexp::external_mem externMem =
    syclexp::import_external_memory(extMemDesc, syclQueue);

// Map imported memory into SYCL memory
syclexp::image_descriptor desc{imgSize, NChannels, channelType};
syclexp::image_mem_handle imgMemHandle =
    syclexp::map_external_image_memory(externMem, desc, syclQueue);

// Create SYCL image and use it as usual
syclexp::unsampled_image_handle imgHandle =
    syclexp::create_image(imgMemHandle, desc, syclQueue);
```

Import memory into SYCL

Create image as usual

Same process to import Vulkan and DirectX 12 memory into SYCL

Vulkan

```
// Describe memory being imported
#ifdef _WIN32
syclexp::external_mem_descriptor<syclexp::resource_win32_handle> extMemDesc{
    vulkanMemHandle, syclexp::external_mem_handle_type::win32_nt_handle,
    imgSize};
#else
syclexp::external_mem_descriptor<syclexp::resource_fd> extMemDesc{
    vulkanMemHandle, syclexp::external_mem_handle_type::opaque_fd, imgSize};
#endif

// Import memory from Vulkan into SYCL
syclexp::external_mem externMem =
    syclexp::import_external_memory(extMemDesc, syclQueue);

// Map imported memory into SYCL image memory handle
syclexp::image_descriptor desc{imgSize, NChannels, channelType};
syclexp::image_mem_handle imgMemHandle =
    syclexp::map_external_image_memory(externMem, desc, syclQueue);

// Create SYCL image and use it as usual
syclexp::unsampled_image_handle imgHandle =
    syclexp::create_image(imgMemHandle, desc, syclQueue);
```

DirectX 12

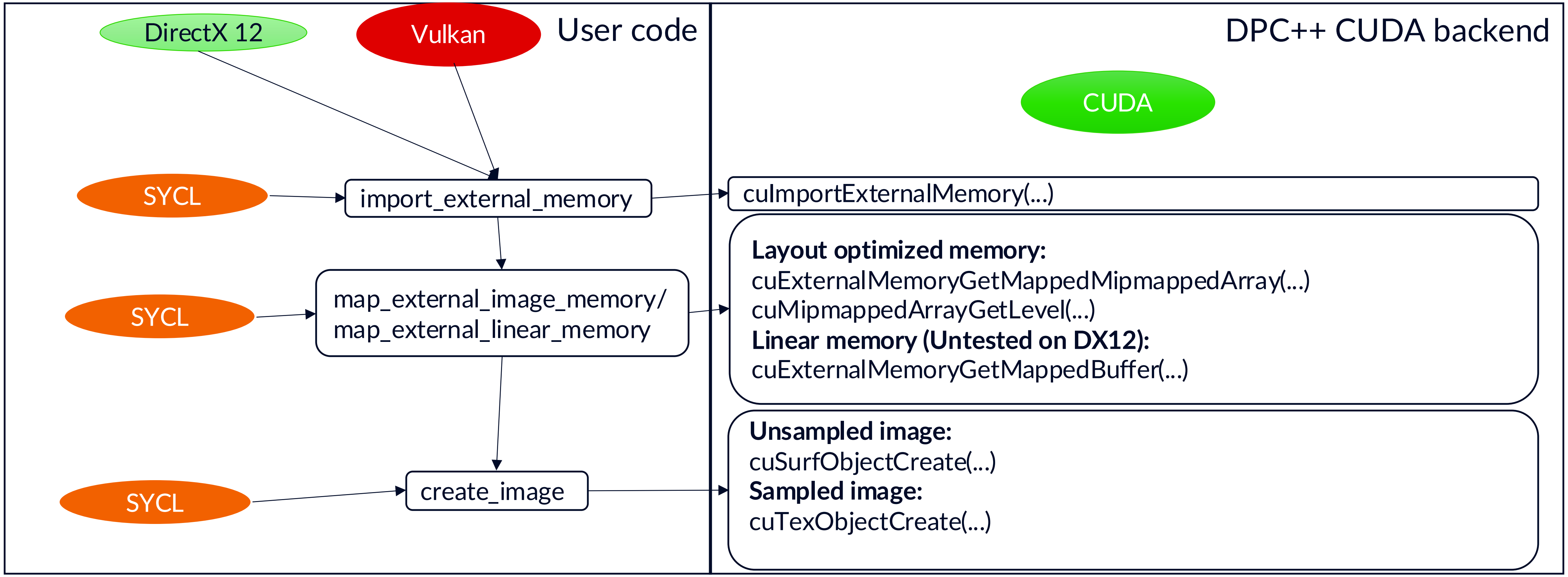
```
// Describe memory being imported
syclexp::external_mem_descriptor<syclexp::resource_win32_handle> extMemDesc{
    dx12MemHandle,
syclexp::external_mem_handle_type::win32_nt_dx12_resource,
    dx12TexAllocInfo.SizeInBytes};

// Import memory from Vulkan into SYCL
syclexp::external_mem externMem =
    syclexp::import_external_memory(extMemDesc, syclQueue);

// Map imported memory into SYCL memory
syclexp::image_descriptor desc{imgSize, NChannels, channelType};
syclexp::image_mem_handle imgMemHandle =
    syclexp::map_external_image_memory(externMem, desc, syclQueue);

// Create SYCL image and use it as usual
syclexp::unsampled_image_handle imgHandle =
    syclexp::create_image(imgMemHandle, desc, syclQueue);
```

Importing memory with the CUDA backend

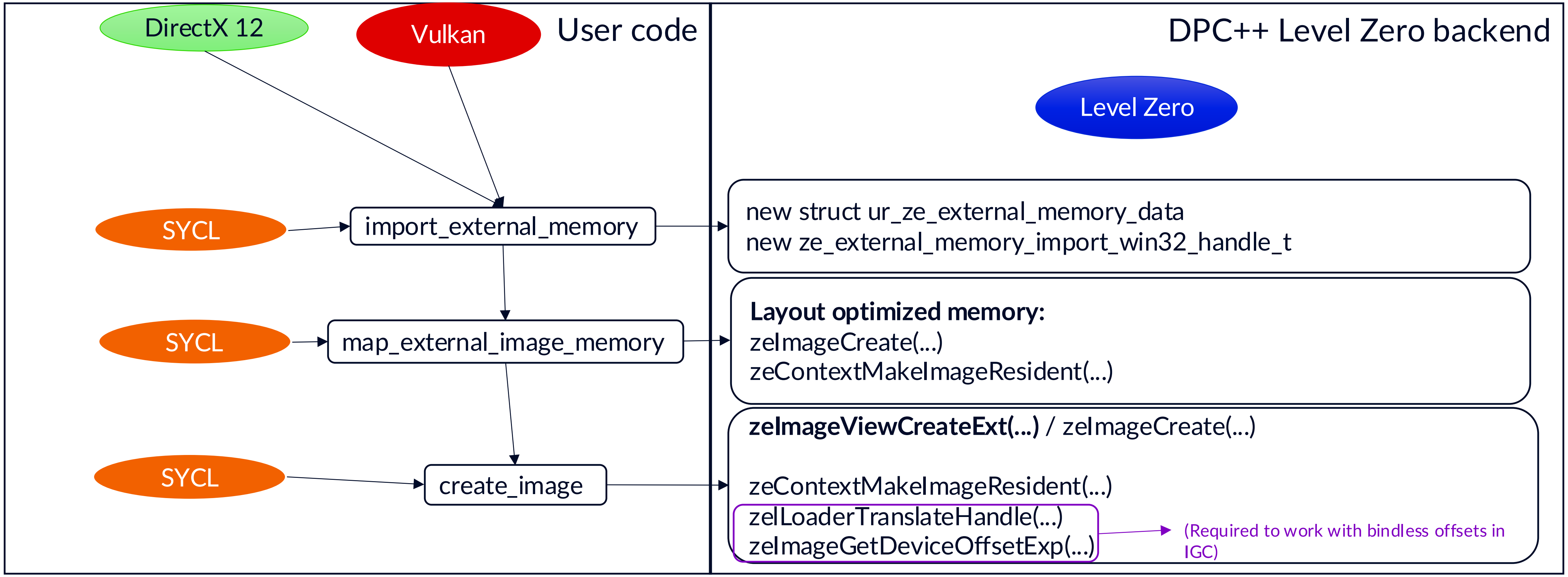


Layout optimized memory:
cuExternalMemoryGetMappedMipmappedArray(...)
cuMipmappedArrayGetLevel(...)
Linear memory (Untested on DX12):
cuExternalMemoryGetMappedBuffer(...)

Unsampled image:
cuSurfObjectCreate(...)
Sampled image:
cuTexObjectCreate(...)

https://github.com/intel/llvm/blob/sycl/sycl/source/detail/bindless_images.cpp
https://github.com/intel/llvm/blob/sycl/unified-runtime/source/adapters/cuda/image.cpp

Importing memory with the Level Zero backend

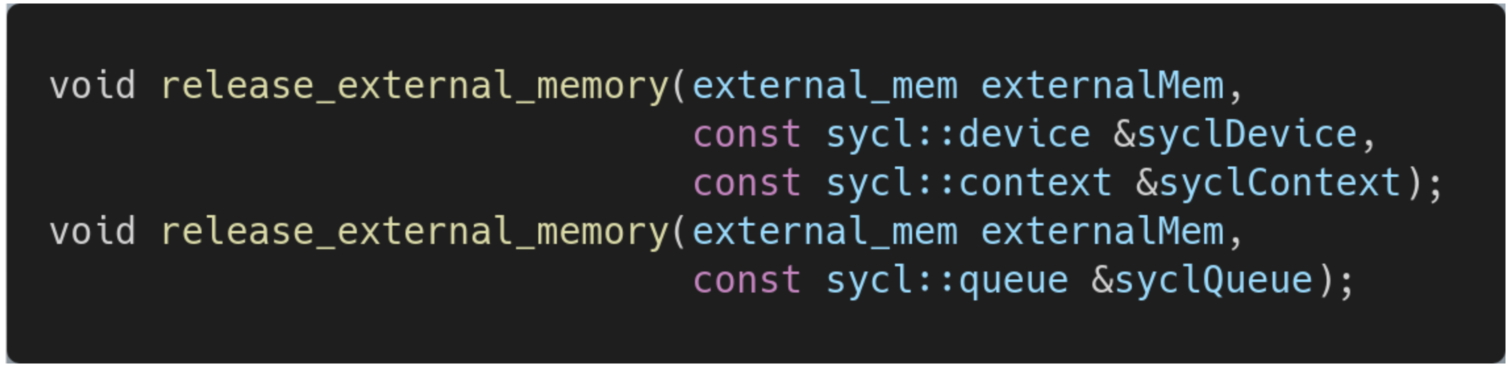


DirectX 12
Vulkan
User code
DPC++ Level Zero backend
Level Zero
SYCL
import_external_memory
new struct ur_ze_external_memory_data
new ze_external_memory_import_win32_handle_t
SYCL
map_external_image_memory
Layout optimized memory:
zeImageCreate(...)
zeContextMakeImageResident(...)
SYCL
create_image
zeImageViewCreateExt(...) / zeImageCreate(...)
zeContextMakeImageResident(...)
zeLoaderTranslateHandle(...)
zeImageGetDeviceOffsetExp(...)
(Required to work with bindless offsets in IGC)

https://github.com/intel/llvm/blob/sycl/sycl/source/detail/bindless_images.cpp
https://github.com/intel/llvm/blob/sycl/unified-runtime/source/adapters/level_zero/image.cpp

Destroying external memory handle

`external_mem` objects must be destroyed after using external memory in SYCL



```
void release_external_memory(external_mem externalMem,
                             const sycl::device &syclDevice,
                             const sycl::context &syclContext);
void release_external_memory(external_mem externalMem,
                             const sycl::queue &syclQueue);
```

Exporting memory from SYCL into Vulkan/DX12

- Currently being investigated
- We hope to make the proposal public soon
- Different processes and capabilities than importing memory
  - Backends handle exporting in different ways

Importing Vulkan/DX12 semaphores into SYCL

Semaphores

- SYCL having access to memory in Vulkan/DX12 is all well and good, but how can we ensure there is no inefficient waiting around?
- Semaphores are synchronization primitives that allow waiting for a condition to be met
- In order for semaphores to properly function, the sycl queue **must** be "in_order"
    - Otherwise kernel and semaphore execution order is not guaranteed

Vulkan/DX12 binary and timeline semaphores

Binary Semaphores

- Has only two states, signaled or unsignaled
- Can only be waited upon to switch to the signalled state
- Simpler, but can also increase complexity due to its simple nature requiring more binary semaphores than if a timeline semaphore was used
- Supported by **opaque_fd** and **win32_nt_handle**

Timeline Semaphores

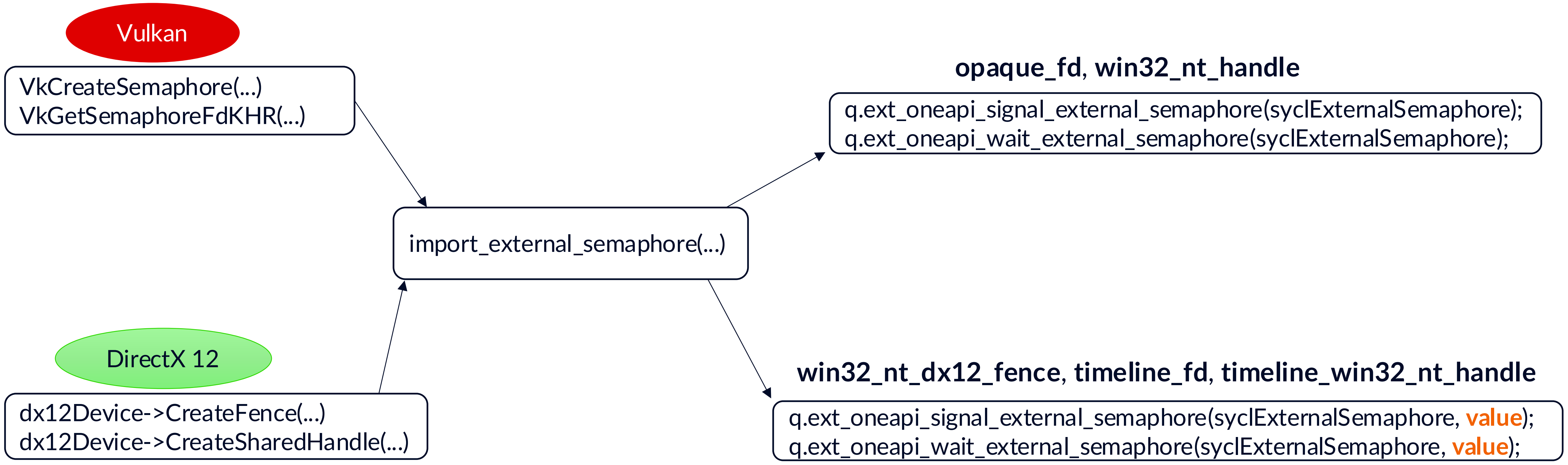
- Has a 64-bit integer value
- Can be waited upon to be a particular value
- Slightly more complex but allows for repeated use – use multiple waits and signals
- Supported by **win32_nt_dx12_fence**, **timeline_fd** and **timeline_win32_nt_handle**

Types of Semaphores in SYCL

- Binary Semaphores
  - opaque_fd
  - win32_nt_handle
- Timeline Semaphores
  - win32_nt_dx12_fence
  - timeline_fd
  - timeline_win32_nt_handle

```
// Types of external semaphore handles
enum class external_semaphore_handle_type
{
  opaque_fd = 0,
  win32_nt_handle = 1,
  win32_nt_dx12_fence = 2,
  timeline_fd = 3,
  timeline_win32_nt_handle = 4,
};
```

Basic process of importing and using semaphores


Vulkan
VkCreateSemaphore(...)
VkGetSemaphoreFdKHR(...)
DirectX 12
dx12Device->CreateFence(...)
dx12Device->CreateSharedHandle(...)
import_external_semaphore(...)
opaque_fd, **win32_nt_handle**
q.ext_oneapi_signal_external_semaphore(syclExternalSemaphore);
q.ext_oneapi_wait_external_semaphore(syclExternalSemaphore);
win32_nt_dx12_fence, **timeline_fd**, **timeline_win32_nt_handle**
q.ext_oneapi_signal_external_semaphore(syclExternalSemaphore, **value**);
q.ext_oneapi_wait_external_semaphore(syclExternalSemaphore, **value**);

Allocate and export Vulkan semaphore

Create exportable semaphore

```
// Setup Vulkan device
VkDevice vulkanDevice = /* ... */;

// Create Vulkan semaphore
VkSemaphore vulkanSemaphore;
{
  VkExportSemaphoreCreateInfo esci = /* ... */;
  VkSemaphoreCreateInfo sci = {};
  sci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
  sci.pNext = &esci;
  vkCreateSemaphore(vulkanDevice, &sci, nullptr, &vulkanSemaphore);
}

// Export semaphore from Vulkan
#ifdef _WIN32
VkSemaphoreGetWin32HandleInfoKHR sgfi = /* ... */;
HANDLE vulkanSemaphoreHandle;
vkGetSemaphoreKHR(vulkanDevice, &sgfi, &vulkanSemaphoreHandle);
#else
VkSemaphoreGetFdInfoKHR sgfi = /* ... */;
HANDLE vulkanSemaphoreHandle;
vkGetSemaphoreWin32HandleKHR(vulkanDevice, &sgfi, &vulkanSemaphoreHandle);
#endif
```

Export semaphore from Vulkan making it available for SYCL to import

Import Vulkan semaphore into SYCL

Describe what kind of semaphore is being imported

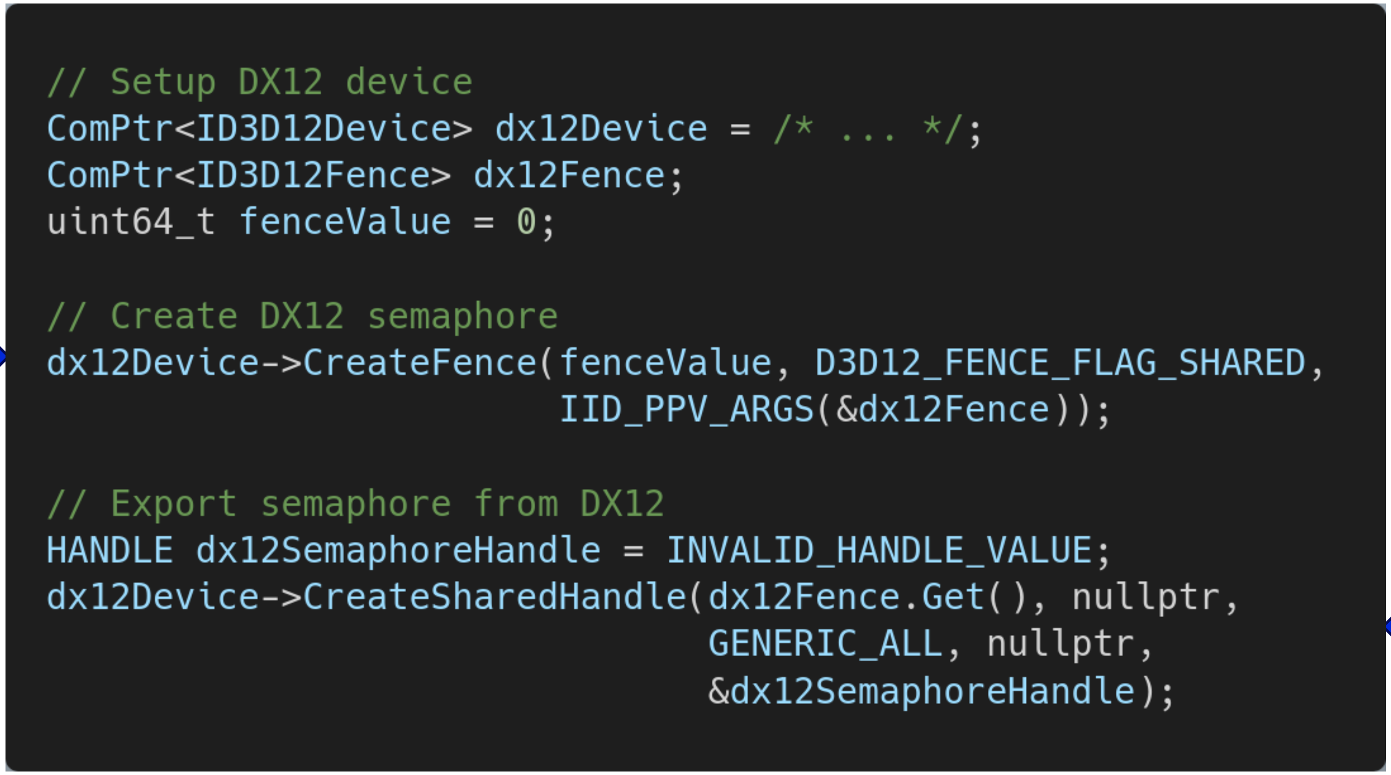
```
// Describe semaphore being imported
#ifdef _WIN32
syclexp::external_semaphore_descriptor<syclexp::resource_win32_handle>
    syclExternalSemaphoreDesc{
        vulkanSemaphoreHandle,
        syclexp::external_semaphore_handle_type::win32_nt_handle};
#else
syclexp::external_semaphore_descriptor<syclexp::resource_fd>
    syclExternalSemaphoreDesc{
        vulkanSemaphoreHandle,
        syclexp::external_semaphore_handle_type::opaque_fd};
#endif

// Import semaphore from Vulkan into SYCL
syclexp::external_semaphore syclExternalSemaphore =
    syclexp::import_external_semaphore(syclExternalSemaphoreDesc, syclQueue);
```

Import semaphore into SYCL

Allocate and export DirectX 12 semaphore

Create semaphore



```
// Setup DX12 device
ComPtr<ID3D12Device> dx12Device = /* ... */;
ComPtr<ID3D12Fence> dx12Fence;
uint64_t fenceValue = 0;

// Create DX12 semaphore
dx12Device->CreateFence(fenceValue, D3D12_FENCE_FLAG_SHARED,
                        IID_PPV_ARGS(&dx12Fence));

// Export semaphore from DX12
HANDLE dx12SemaphoreHandle = INVALID_HANDLE_VALUE;
dx12Device->CreateSharedHandle(dx12Fence.Get(), nullptr,
                               GENERIC_ALL, nullptr,
                               &dx12SemaphoreHandle);
```

Export semaphore from Vulkan making it available for SYCL to import

Import DirectX 12 semaphore into SYCL

Describe what kind of semaphore is being imported

```
// Describe semaphore being imported
syclexp::external_semaphore_descriptor<syclexp::resource_win32_handle>
    extSemDesc{dx12SemaphoreHandle,
               syclexp::external_semaphore_handle_type::win32_nt_dx12_fence};

// Import semaphore from DX12 into SYCL
syclexp::external_semaphore syclExternalSemaphore =
    syclexp::import_external_semaphore(extSemDesc, syclQueue);
```

Import semaphore into SYCL

Destroying external semaphore handle

`external_semaphore` objects must be destroyed after using external semaphores in SYCL

```
void release_external_semaphore(external_semaphore semaphoreHandle,
                                const sycl::device &syclDevice,
                                const sycl::context &syclContext);

void release_external_semaphore(external_semaphore semaphoreHandle,
                                const sycl::queue &syclQueue);
```

Exporting semaphores from SYCL into Vulkan/DX12

- There is currently no capability to create semaphores in SYCL
- Neither CUDA or Level Zero have capabilities to create and export semaphores

Problems encountered when mapping CUDA interop API

- There are legacy APIs that are not generic, making mapping difficult at times
- Using CUDA, interop with OpenGL requires using CUDA graphics interoperability API, instead of CUDA external resource interoperability API
  - This former API is very specific to CUDA, making mapping difficult so we favour the latter
  - If we really need to, we will need a separate SYCL extension that is only applicable to CUDA and OpenGL interop

SYCL interop as separate extension

- Planning to have SYCL interop as a separate extension from SYCL Bindless Images extension
  - Possibly multiple separate extensions i.e. separate ones for importing and exporting memory

Future Work

- Exporting memory from SYCL into Vulkan/DX12
- Splitting external memory and semaphores into their own extension
- DX11 interop
- Additional image formats
- SYCL buffer and USM interop
- Additional synchronization primitives
- Use imported object with `host_task` to directly access interop resources with the backend API such as CUDA and Level Zero
    - i.e. pass an imported `CUarray` directly to a CUDA function using `host_task`

codeplay®

Disclaimers

A wee bit of legal

No product or component can be absolutely secure.

Your costs and results may vary.

Intel technologies may require enabled hardware, software or service activation.

© Codeplay Software Ltd.. Codeplay, Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

Q&A

Thank You!