IWOCL 2024

The 12th International Workshop on OpenCL and SYCL

SYCL Bindless Images

Sean Stirling, Codeplay

Sean Stirling, Isaac Ault, Duncan Brawley, Przemek Malon, Alastair Murray, Chedy Najjar, and Peter Žužek

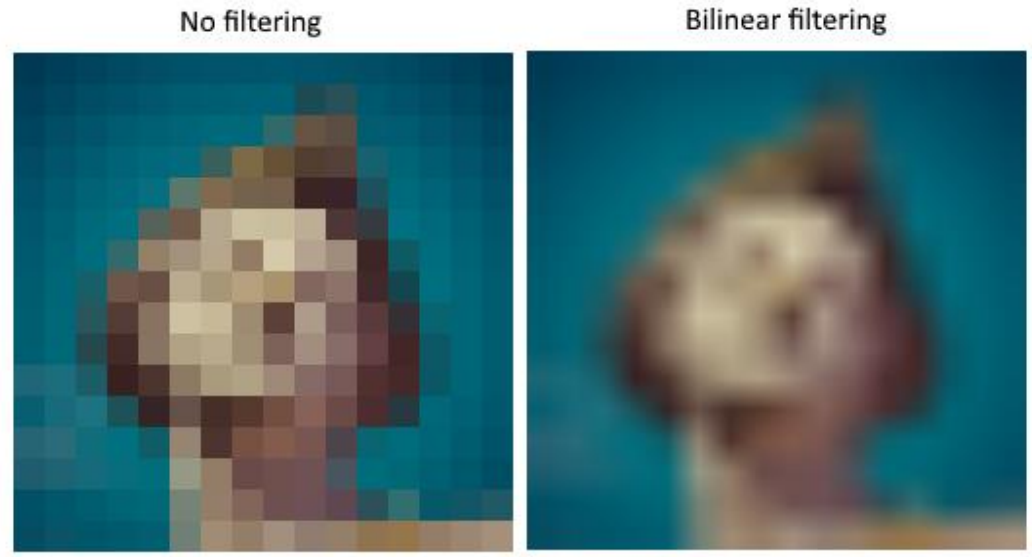
APRIL 8-11, 2024 | CHICAGO, USA | IWOCL.ORG

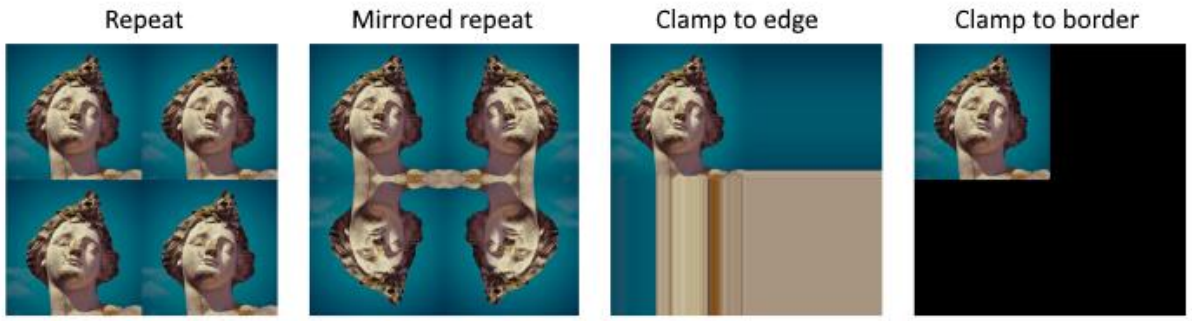
Agenda

- Intro
  - The Importance of Images
  - Bindless Textures – A Brief History
- Why Are Current SYCL Images Insufficient?
- SYCL Bindless Images
- Additional Interoperability Functionality
- Q&A

The Importance of Images

- Filtering
- Addressing Modes
- Texture Cache
- Image Types





[4] (Alexander Overvoorde, Texture mapping, Image view and sampler)

The Importance of Images

- Blender
- Cinebench 2024

Copyright Maxon

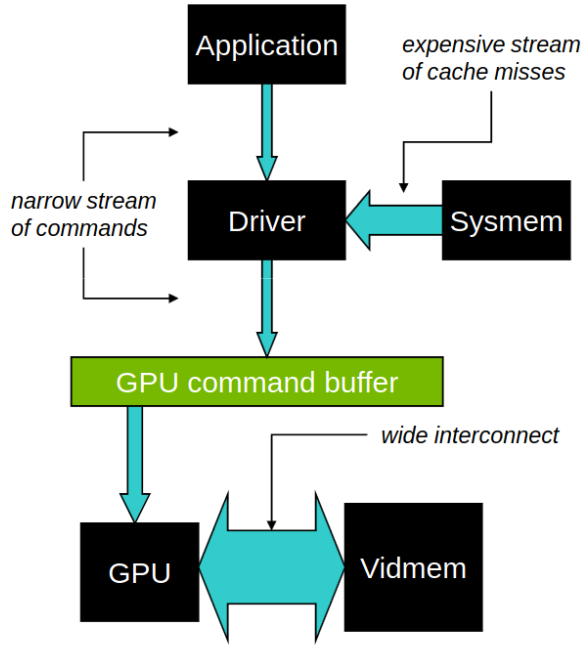
Blender is a registered trademark (®) of the Blender Foundation in EU and USA

Once Upon A Time...

Bindless Textures – A Brief History

- OpenGL introduced the binding of objects:
  - -> bound to context
  - -> bound to other container objects (e.g. vertex array objects)
  - Replaces context state variable manipulation
  - Far smaller API stream – order of magnitude speedup

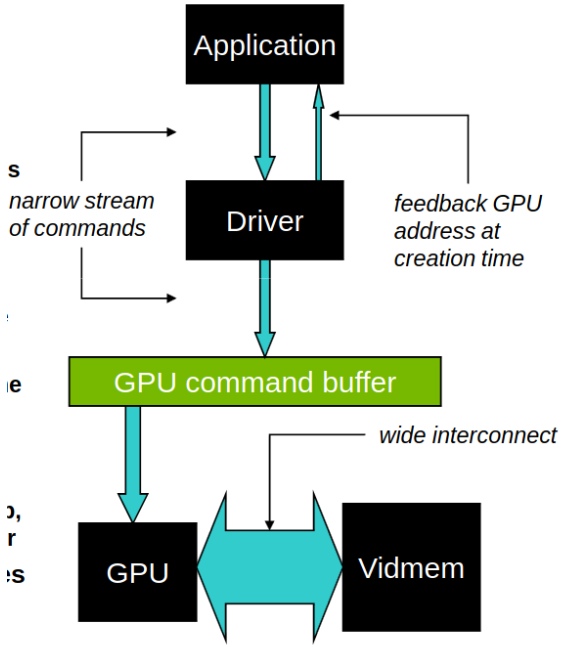
- The modern CPU bottleneck
  - Each bind – several reads of object data
  - Each read – multiple dereferences
  - Each dereference – likely CPU L2 cache miss



[2] (NVIDIA, Gernot Ziegler, Textures & Surfaces, CUDA Webinar, 2011, slide 4)

Bindless Textures – A Brief History

- Still want to use objects, but more directly
  - By GPU address (or handle) - driver feeds back at creation time
  - Driver no longer has to fetch GPU address from sysmem
- Huge speedup – 7x according to NVIDIA[1]
- Introduced:
  - OpenGL extension (2009)
  - CUDA as Bindless Textures (2012)



[3] (NVIDIA, Gernot Ziegler, Textures & Surfaces, CUDA Webinar, 2011, slide 5)

Present Day

Present Day

- All modern GPU APIs work this way
- SYCL images make use of these APIs as backends
  - Not much performance boost to be expected then
- So, what's the problem?
  - Googling Bindless Images will only lead to the preceding history lesson

SYCL Images

Why are current SYCL images insufficient?

The rigid abstractions and lack of past development interest that define SYCL images have led to:

- Lacklustre Flexibility
- Tenuous Control
- Feature Sparsity
- Translation Difficulties for to-SYCL porting tools

Why are SYCL images insufficient?

- **Lacklustre Flexibility**
- Tenuous Control
- Feature Sparsity
- Translation Difficulties

- Forced to request access through accessors
- Number of images must be known at compile-time
  - No dynamic arrays
  - Virtual texturing is impossible
- Compile-time constraints on static arrays
  - Must have the same number of dimensions

Why are SYCL images insufficient?

- Lacklustre Flexibility
- **Tenuous Control**
- Feature Sparsity
- Translation Difficulties

- No control over how images are stored on device
  - No USM images
  - No choice between layouts/encodings (tile swizzle, linear/pitched, etc.)
- No distinction between device image data and the image itself
- Limited choice over how image data is copied to/from the device

Why are SYCL images insufficient?

- Lacklustre Flexibility
- Tenuous Control
- **Feature Sparsity**
- Translation Difficulties

- Lack of image types
  - No mipmaps
  - No cubemaps
- No sub-region copies
- No per-dimension addressing modes

Why are SYCL images insufficient?

- Lacklustre Flexibility
- Tenuous Control
- Feature Sparsity
- **Translation Difficulties**

- Poor mapping to CUDA
- Difficult translation for SYCLomatic

SYCL Bindless Images

SYCL Bindless Images – DPC++ Extension

- Inspired by the lightweight nature of CUDA Bindless Textures
  - Separate image data allocation from image creation
  - Treat all images as opaque handles

- Highly Flexible
- Full Control
- Feature Rich
- Simplistic to-SYCL translation

Image Memory

- Opaque image memory handle
  - Device optimized layout/encoding

```
/// Opaque image memory handle type
struct image_mem_handle {
  using handle_type = void *;
  handle_type raw_handle;
};
```

- RAII wrapper
  - Allocated image memory on construction
  - Deallocates image memory on destruction
  - Contains image descriptor

```
class image_mem {
public:

  // Constructors / Destructors / Operators
  image_mem() = default;
  image_mem(image_mem &&rhs) = default;
  // ...

  // Getters
  image_mem_handle get_handle() const;
  image_descriptor get_descriptor() const;
  // ...

protected:
  // impl
  std::shared_ptr<detail::image_mem_impl> impl;
  // ...

};
```

Image Descriptor

Note: bringing back SYCL 1.2.1 image channel order and type

- Represents all possible supported image properties
- Must be the same for allocation and creation
- Easy to construct
  - Prevents invalid/unsupported image property combinations

```
/// A struct to describe the properties of an image.
struct image_descriptor {
  size_t width;
  size_t height;
  size_t depth;
  image_channel_order channel_order;
  image_channel_type channel_type;
  image_type type;
  unsigned int num_levels;
  unsigned int array_size;

  // Constructors
  image_descriptor() = default;
  // ...
};
```

Image Memory Allocation

- Standalone

```
// Descriptor
syclexp::image_descriptor desc(width, image_channel_order::r, image_channel_type::fp32);

// Allocate
syclexp::image_mem_handle imgMemHandle = syclexp::alloc_image_mem(desc, dev, ctxt);

// Free
syclexp::free_image_mem(imgMemHandle, syclexp::image_type::standard. dev, ctxt);
```

- RAII wrapper

```
// Descriptor
syclexp::image_descriptor desc(width, image_channel_order::r, image_channel_type::fp32);

// RAII Allocate & Free
syclexp::image_mem imgMem(desc, dev, ctxt)
```

Image Memory Allocation

- Additional pitched allocation functionality
  - Pads image rows for optimized device memory access
  - USM – on device only
  - 2D images only

```
// Descriptor
syclexp::image_descriptor desc(
    {width, height}, sycl::image_channel_order::rgba,
    sycl::image_channel_type::fp32);

size_t pitch = 0;

// Returns the device pointer to USM allocated pitched memory
void* imgMemUSM = syclexp::pitched_alloc_device(&pitch, desc, dev, ctxt);
```

Additional Copy Functions

- Copies to/from:
  - Opaque image memory handles
  - Pitched USM allocations
  - Host allocations
- HtoD, DtoH
  - DtoD (coming soon!)
- Sub-region copies!

```
q.ext_oneapi_copy(dataIn.data(), imgMemHandle, desc);
```

```
// Sub-region copy: copy over data to device (one quadrant)
sycl::range copyExtent = {width / 2, height / 2, 1};
sycl::range srcExtent = {width / 2, height / 2, 0};

q.ext_oneapi_copy(dataIn.data(), {0,0,0}, srcExtent,
                  imgMem.get_handle(), {0,0,0}, copyExtent);
```

Opaque Image Handles

- Opaque image handles
  - Two handle types
    - Unsampled images
    - Sampled images
- Image handle creation on USM allocated memory
  - Sampled images only
- Easy to create and use
- Must be destroyed when finished with

Opaque image handles

```
/// Opaque unsampled image handle type.
struct unsampled_image_handle {
  using raw_handle_type = pi_uint64;
  raw_handle_type raw_handle;
};
/// Opaque sampled image handle type.
struct sampled_image_handle {
  using raw_handle_type = pi_uint64;
  raw_handle_type raw_handle;
};
```

- Simple
  - Easily passed to the kernel
- The distinction creates useful compile-time constraints
  - No writing to sampled images!
- Maps well to SYCL 2020 and CUDA
- Two handles to represent all image types

Unsampled Image Handle

```
/// Opaque unsampled image handle type.
struct unsampled_image_handle {
  using raw_handle_type = pi_uint64;
  raw_handle_type raw_handle;
};
```

- Fetch
- Write
- No sampling
- No USM
  - Device optimized layout only

Sampled Image Handle

```
/// Opaque sampled image handle type.
struct sampled_image_handle {
  using raw_handle_type = pi_uint64;
  raw_handle_type raw_handle;
};
```

- Read only
- Hardware sampling capabilities
- Can be backed by
  - Device optimized layout/encoding
  - USM allocated memory
- Sampler is tied on creation

Creating Image Handles

Unsampled

```
// Create the unsampled image
// and return the handle
syclexp::unsampled_image_handle
  unsampImg = syclexp::create_image(
    imgMemHandle, desc, dev, ctxt);
```

Sampled

```
// Bindless sampler
syclexp::bindless_image_sampler samp(
    sycl::addressing_mode::repeat,
    sycl::coordinate_normalization_mode::normalized,
    sycl::filtering_mode::linear);

// Create the sampled image
// and return the handle
syclexp::sampled_image_handle sampImg =
  syclexp::create_image(
    imgMemHandle, samp, desc, dev, ctxt);
```

Bindless Sampler

- A new sampler object
- Exposes numerous & diverse image sampling capabilities
  - (=) Addressing modes
    - (+) Unique modes per dimension
  - (=) Coordinate normalization mode
  - (=) Filtering Modes
  - (+) Mipmap Filtering
    - (+) LOD filtering
    - (+) Anisotropic filtering
  - (+) Cubemap seamless filtering

```
// Bindless sampler
struct bindless_image_sampler {

  // Constructors
  // ...

  sycl::addressing_mode addressing[3];
  sycl::coordinate_normalization_mode coordinate;
  sycl::filtering_mode filtering;
  sycl::filtering_mode mipmap_filtering;
  syclexp::cubemap_filtering_mode cubemap_filtering;
  float min_mipmap_level_clamp = 0.f;
  float max_mipmap_level_clamp = 0.f;
  float max_anisotropy = 0.f;
};
```

Kernel fetch, write, and HW sample

Unsampled

```
// Read image data from unsampled handle
sycl::float4 px1 =
  syclexp::fetch_image<sycl::float4>(
    imgIn1, int(id[0]));
sycl::float4 px2 =
  syclexp::fetch_image<sycl::float4>(
    imgIn2, int(id[0]));

sum = px1[0] + px2[0];

// Write to image using unsampled handle
syclexp::write_image<sycl::float4>(
  imgOut, int(id[0]), sycl::float4(sum));
```

Sampled

```
// Normalize cooridnate -- +0.5
// looks towards centre of pixel
float x = float(id[0] + 0.5) / (float)N;

// Sample image data from sampled handle
float px1 =
  syclexp::sample_image<float>(imgHandle, x);

// Fetch image data from sampled handle
float px2 =
  syclexp::fetch_image<float>(
    imgHandle, int(id[0]));
```

User Defined Types

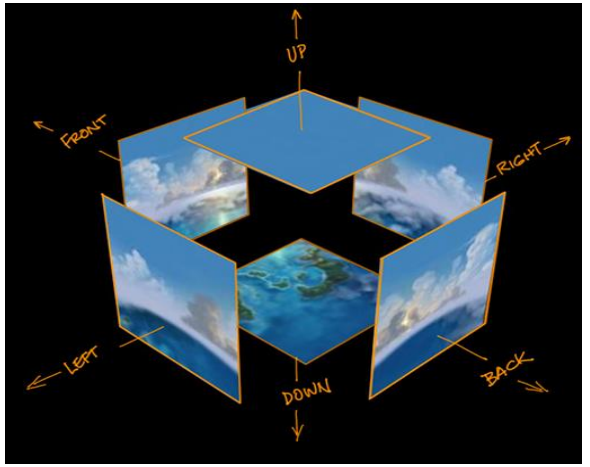
- Images can be read from, written to, and sampled using user defined types
- Must be trivially copyable
- Hint type required

```
struct my_float4 {
  float x, y, z, w;
};

...

// In the kernel
my_float4 myPixel{};

// User-defined unsampled fetch/write -- hint required
myPixel += syclexp::fetch_image<my_float4, float4>(unsampledImg, coords);
syclexp::write_image<my_float4, float4>(unsampledImg, coords, myPixel);

// User-defined sampled read -- hint required
myPixel += syclexp::sample_image<my_float4, float4>(sampledImg, floatCoords);
```

Varied Image Types

- Mipmaps
  - LOD Filtering
  - Anisotropic Filtering
- Image Arrays
- Cubemaps
  - Seamless Filtering

[4] (Alexander Overvoorde, Texture mapping, Image view and sampler)
Copyright Vulkan Tutorial



[5] (Scali, Cubemaps, 2013)

Varied Image Types

- Similar code between image types
- Just make sure the descriptor is constructed appropriately
- Some nuance required with mipmaps
  - Mipmaps are copied to on a level-by-level basis

Varied Image Types

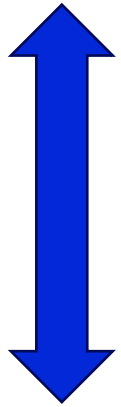
Descriptor

Allocate

Copy

Create

```
// Descriptor -- array size
unsigned int array_size = 2;
syclexp::image_descriptor desc(
    width, image_channel_order::rgba, image_channel_type::fp32,
    syclexp::image_type::array, /* array image type */
    1, /* mip levels */
    array_size /* number of images in array */);

// Allocate memory on device
syclexp::image_mem imgMem(desc, q);

// Copy over data to device
q.ext_oneapi_copy(input.data(), imgMem.get_handle(), desc);

// Create unsampled image handle to image array
syclexp::unsampled_image_handle imgHandle =
    sycl::ext::oneapi::experimental::create_image(
        imgMem, desc, dev, ctxt);
```



- Standard
- Arrays
- Cubemap

Varied Image Types

✓ -Supported
✗ -Won't support
⌛ -Coming soon

Mipmaps

- Unsampled ✗
- Sampled
  - Fetch ✗
  - LOD Filtering ✓
  - Anisotropic Filtering ✓

Image Arrays

- Unsampled
  - Fetch ✓
  - Write ✓
- Sampled (coming soon!)
  - Fetch ⌛
  - Sample ⌛

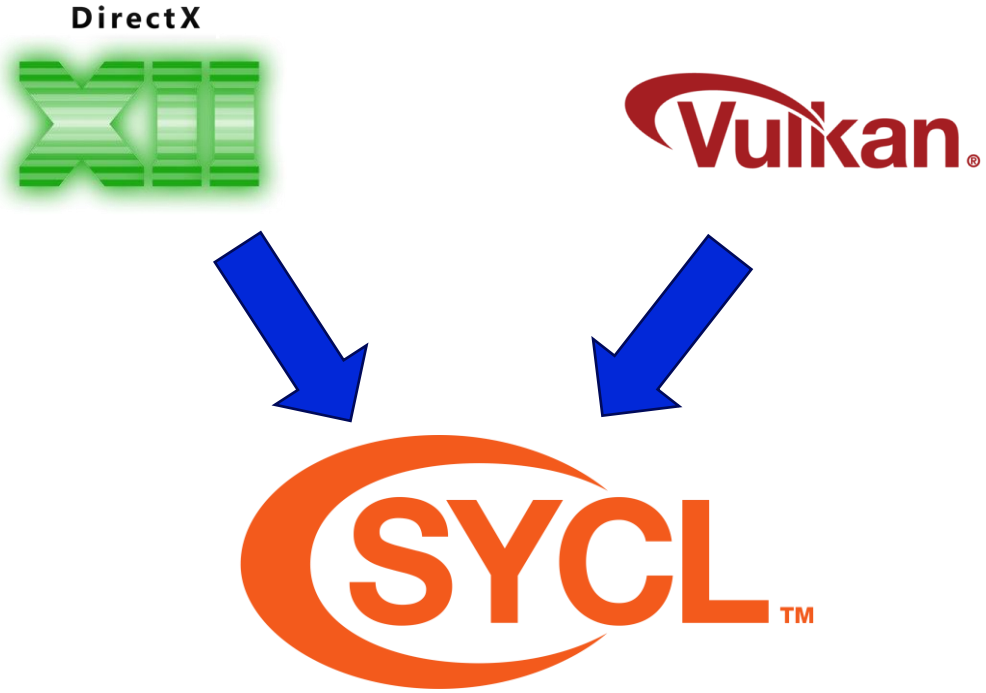
Cubemaps

- Unsampled
  - Fetch ✓
  - Write ✓
- Sampled
  - Fetch ✗
  - Sample ✓

One more thing...

Interop

- Additional interoperability functionality
  - Vulkan
  - DirectX12
- Import external image memory
  - No copies!
- Import external synchronization primitives
  - Wait on, or signal, external semaphores



Importing external memory

- Standard and mipmap images
  - Image arrays and cubemaps coming soon
- Simple two steps
  - Import the external memory
  - Map the imported memory to a usable memory object
    - USM pointer (coming soon)
    - image_mem_handle
- Free to create an image on the imported memory
  - Fetch/sample/write from/to the externally allocated memory

Importing external image memory

From external API

Declare the external mem descriptor

Map the interop handle to an image_mem_handle

```
int img_fd = /* exported from external API */;
int img_size = /* img size */;

// Image descriptor - mapped to external API image layout
syclexp::image_descriptor desc(dims, order, CType,
                               syclexp::image_type::standard);

// Setup the external mem descriptor
syclexp::external_mem_descriptor<syclexp::resource_fd>
    ext_mem_desc{img_fd, img_size};

// Create interop memory handles
syclexp::interop_mem_handle interop_mem_handle =
  syclexp::import_external_memory(ext_mem_desc, dev, ctxt);

// Map image memory handles
syclexp::image_mem_handle mapped_mem_handle =
  syclexp::map_external_image_memory(interop_mem_handle, desc,
                                     dev, ctxt);

// Create the image and return the handle
syclexp::unsampled_image_handle imgHandle =
  syclexp::create_image(mapped_mem_handle, desc, dev, ctxt);
```

Must match external API image layout

Create the interop handle

Create an image backed by imported memory!

Importing external synchronization primitives

- Even simpler one stage process
  - Import the external semaphore
- Wait on or signal the external semaphores

Importing external synchronization primitives

From external API

Import the semaphore

```
int semaphore_fd = /* exported from external API */;

// Setup external semaphore descriptor
syclexp::external_semaphore_descriptor<syclexp::resource_fd>
    ext_semaphore_desc{semaphore_fd};

// Import semaphore
syclexp::interop_semaphore_handle semaphore_handle =
  syclexp::import_external_semaphore(ext_semaphore_desc,
                                     dev, ctxt);

// Wait on or signal imported semaphore
q.ext_oneapi_wait_external_semaphore(semaphore_handle);
q.ext_oneapi_signal_external_semaphore(semaphore_handle);
```

Setup external semaphore descriptor

Wait on or signal the external semaphore!

Have we achieved our goal?

Have we achieved our goal?

Flexible

- No accessors or rigid abstractions
- Simple handles
  - Memory and Image
  - No constraints on how to store these handles – vs compile-time constraints on static arrays
- Dynamic arrays
  - Any number of images

Feature-rich

- Varied image types
  - Mipmaps
  - Image arrays
  - Cubemaps
- Sub-region copies
- Per-dimension addressing modes

Controllable

- A choice of how images are stored on device
  - USM backed images – Linear/pitched
  - Device optimized
- Distinction between device image data and the image handle
- Full choice over how/when image data is copied to/from the device

Translatable

- Conforms well to CUDA
  - And other compute APIs/languages
- Much simpler translation for SYCLomatic

Backend/Device Support

- NVIDIA/CUDA
  - Fully supported
- Intel/Level Zero
  - Level Zero extension
  - SPIR-V extension - https://github.com/intel/llvm/pull/12927
    - Converting handles to SPIRV Images & Sampler
  - Being implemented as I speak
    - A first revision will implement standard Bindless Images and interop functionality
    - Future work will include the additional image types
- AMD/HIP
  - Lower priority
  - Straightforward port from CUDA implementation
- OpenCL
  - No priority

DPC++ Extension

- Currently working towards revision 6
- View the spec here: https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/experimental/sycl_ext_oneapi_bindless_images.asciidoc

Future Work

- Support for bindless images in:
  - Level Zero & SPIR-V (Intel GPUs)
  - AMD HIP
- Performance
  - Closely work with Blender
  - Profiling, benchmarking, and optimization
- Future revisions
  - Combined image types (mipmapped arrays, mipmapped cubemap, etc.)
  - More interoperability
    - More image types (image arrays, cubemaps)
    - More APIs and resource types (KMT handles)
    - Export image memory (maybe)
- Prepare a KHR extension
  - To replace SYCL 2020 images
  - There's interest in integrating SYCL Bindless Images in SYCL Next

Citations

- [1] Bindless Graphics Tutorial https://www.nvidia.com/en-us/drivers/bindless-graphics/
- [2] NVIDIA, Gernot Ziegler, Textures & Surfaces, CUDA Webinar, 2011, slide 4, https://developer.download.nvidia.com/opengl/tutorials/bindless_graphics.pdf
- [3] NVIDIA, Gernot Ziegler, Textures & Surfaces, CUDA Webinar, 2011, slide 5, https://developer.download.nvidia.com/opengl/tutorials/bindless_graphics.pdf
- [4] Alexander Overvoorde, Texture mapping, Image view and sampler, https://vulkan-tutorial.com/Texture_mapping/Image_view_and_sampler
- [5] Scali, Cubemaps, 2013, https://scalibq.wordpress.com/2013/06/23/cubemaps/

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.

Q&A

Thank you!