

The Khronos Open*^L APIs

Afrigraph 2009

Dave Shreiner
ARM/Khronos



Agenda

- The other Open*L Libraries
- OpenGL
 - a bit of history
 - new features for 3.0
 - deprecation model
- OpenCL
 - what is it, and what's it for?
 - architecture
 - kernel programming language

To Be Fair ...

- There are a few other APIs that match the “Open*L” glob
 - OpenGL ES
 - still at version 2.0 – the embedded space moves very slowly
 - OpenGL SC
 - may never move again – verification is very tedious
 - OpenSL ES
 - I think this has something to do with sound
 - ... but it isn't graphics, so I don't know anything about it 😊

The Road to OpenGL 3.0



A Brief Look Back

- OpenGL versions through time:
 - 1992 – 1.0
 - 1994 – 1.1
 - 1998 – 1.2 & 1.2.1
 - 2001 – 1.3
 - 2002 – 1.4
 - 2003 – 1.5
 - 2004 – 2.0
 - 2006 – 2.1
- 2007 – future “Long’s Peak” version announced to be OpenGL 3.0
- 2008 – OpenGL 3.0 announced
 - contains nothing that was described in “Long’s Peak”

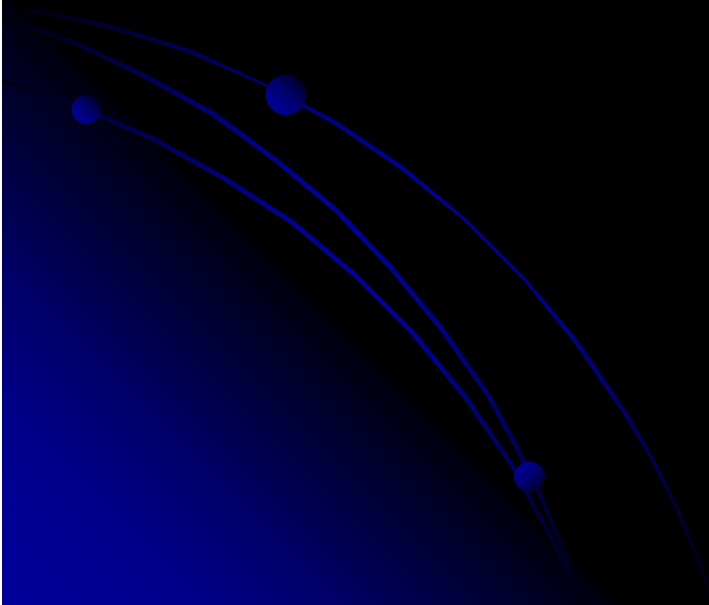
A New Design Methodology

- ARB controls OpenGL's evolution
 - a Khronos Working Group
- Majority of members thought things were taking too long
 - convergence of Long's Peak was always about 18 weeks from completion
 - growing concern about compatibility with existing OpenGL applications
 - Long's Peak was an entirely different API

A More Schedule Driven Approach

- Try to make a new OpenGL release every six months
- Work from a list of extensions for promotion into the core API
 - whatever can't be integrated is pushed to the next release
- OpenGL 3.0 was the first release using this model

What's New in OpenGL 3.0



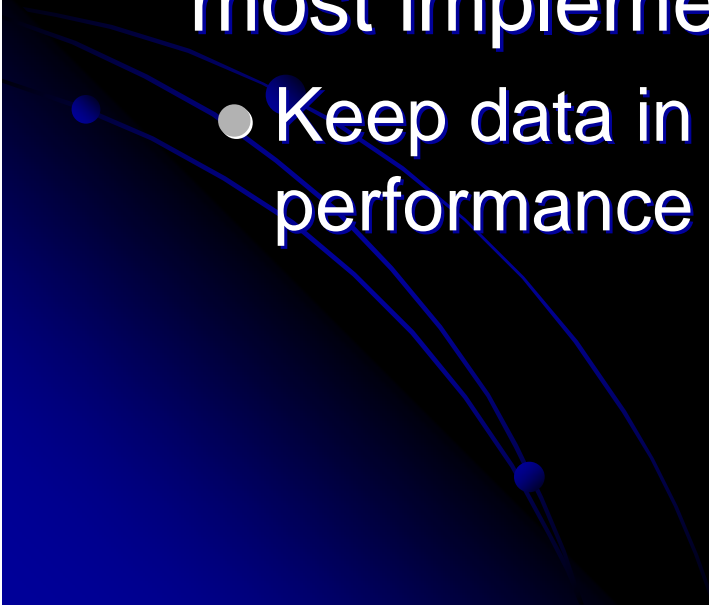
OpenGL 3.0

- New features for rendering
 - framebuffer objects
 - transform feedback
 - conditional rendering
 - ...
- New features in GLSL 1.30
- Deprecation Model

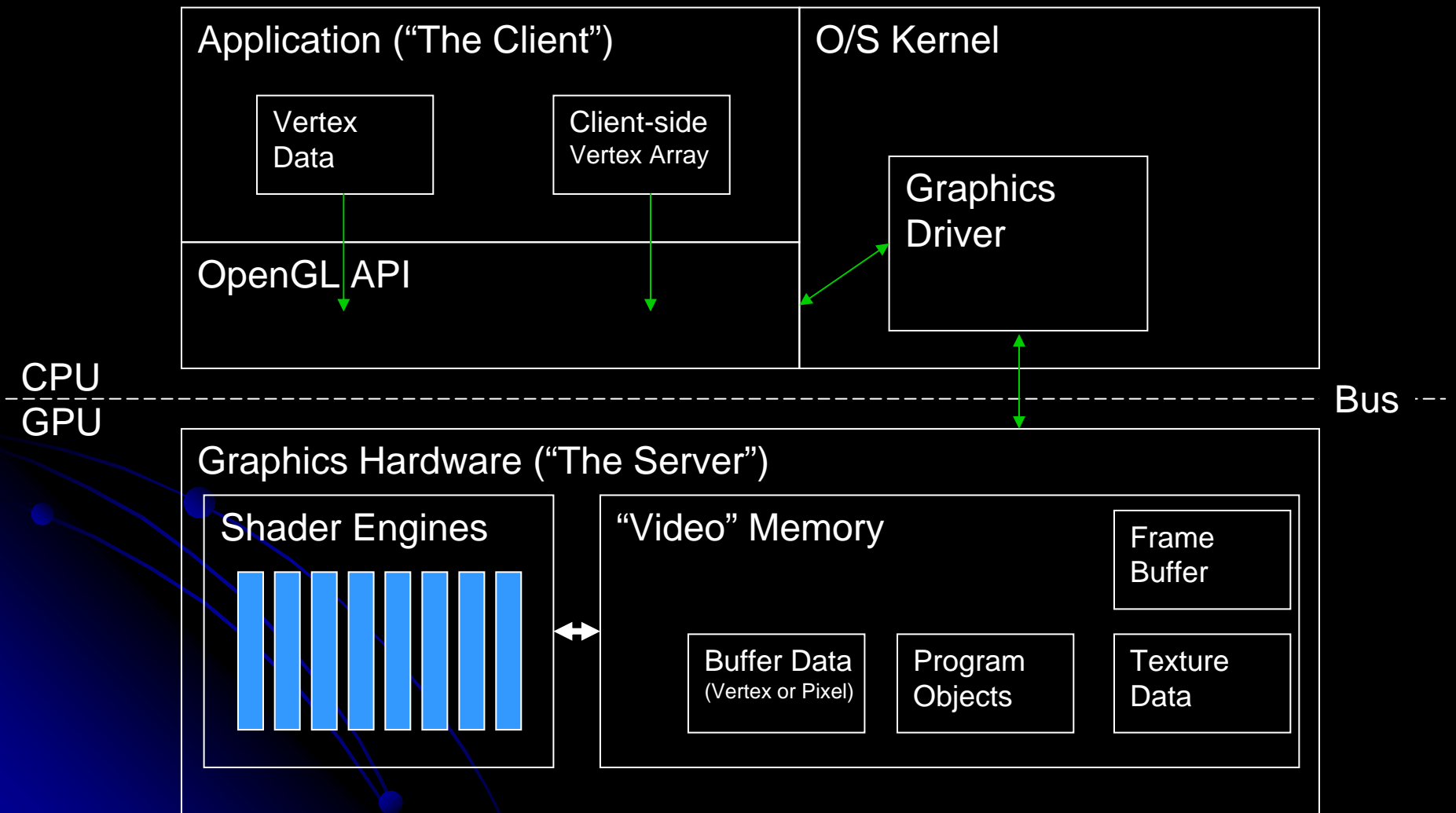
Of course, not everyone liked what was done:

[Hitler's YouTube commentary on OpenGL 3.0](#)

Motivations for New Features

- Much of the API design is for increased performance
 - really no new rendering effects, per se
 - System bus is usually the bottleneck for most implementations
 - Keep data in graphics server for best performance
- 

A Conceptual Model



Enhanced Buffer Management

- Prior to 3.0, you needed to either
 - update a buffer by calling `glBufferSubData`
 - or map the entire buffer by calling `glMapBuffer`
- **Either case caused potential buffer caching issues**
- **`glMapBufferRange` allows finer-grained control**
 - hints to the server what memory will need to be revalidated

Conditional Rendering

- Previously, *occlusion queries* would let you know if any samples passed the depth test
 - nice, but requires a round-trip to the server
 - slow, and potentially stalls the pipeline
- Conditional rendering is something like a display list with a built in occlusion query test

Conditional Rendering (cont'd)

Before:

```
GLuint occ;  
glGenQueries( 1, &occ );  
glBeginQuery(  
    GL_SAMPLES_PASSED, occ );  
// render occlusion test geom.  
glEndQuery( GL_SAMPLES_PASSED );
```

```
glGetQueryObjectiv( occ,  
    GL_QUERY_RESULTS, &n );  
if ( n > 0 ) {  
    // render occluded geom.  
}
```

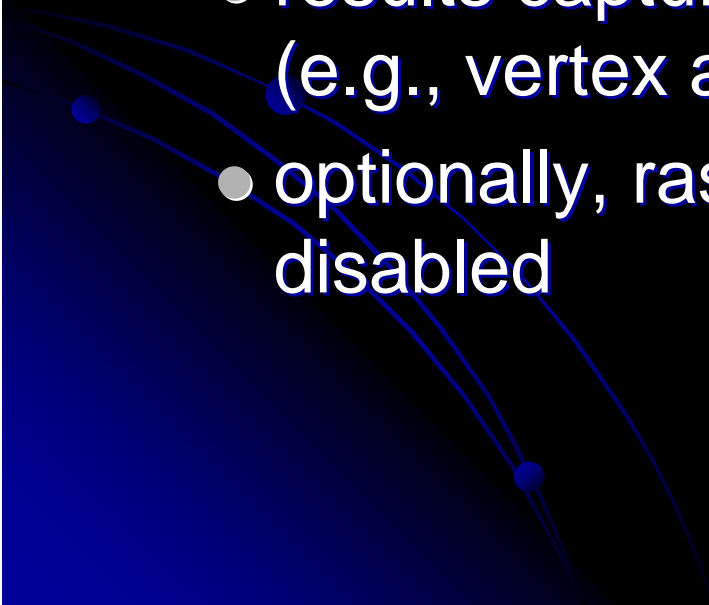
After:

```
GLuint occ;  
glGenQueries( 1, &occ );  
glBeginQuery(  
    GL_SAMPLES_PASSED, occ );  
// render occlusion test geom.  
glEndQuery( GL_SAMPLES_PASSED );  
  
glBeginConditionalRender( occ,  
    GL_QUERY_NO_WAIT );  
// render occluded geom.  
glEndConditionalRender( occ );
```

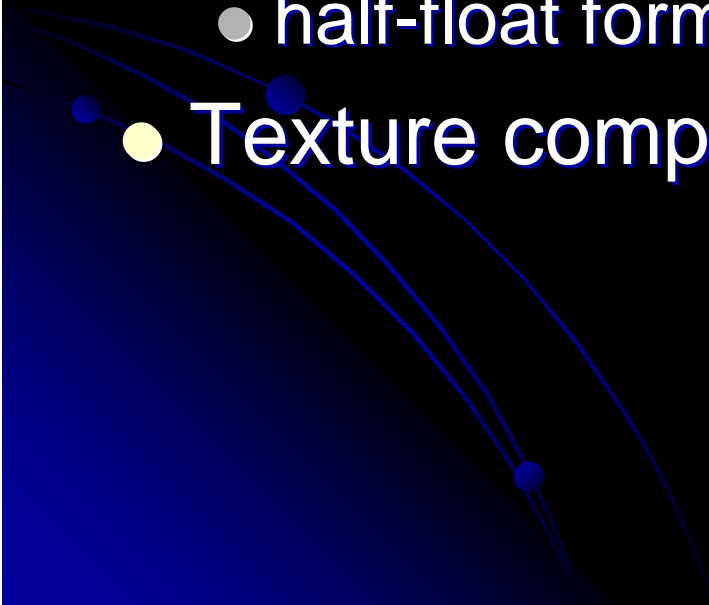
Framebuffer Objects

- Allows “compatible” buffer objects to be framebuffers (i.e., a rendering destination)
 - enables render-to-texture within OpenGL
 - allows rapid recycling of data within the pipe
 - render to a buffer, then bind that buffer as a texture
 - no copy required
 - Used with `gl_FragData` to specify **multiple-render-target (MRT) destinations**
- Per-object blending and color-writemask control

Transform feedback

- Shade vertices capturing transformed results
 - vertices are processed by bound vertex shader
 - results captured in a buffer object (e.g., vertex array object)
 - optionally, rasterization of primitives can be disabled
- 

Texturing Functionality Overview

- Texture arrays
 - Texture and renderbuffer formats
 - redefined one- and two-component formats
 - signed- and unsigned-integer texture
 - half-float formats
 - Texture compression mode
- 


Texture Arrays

- Store an array of n -D dimensional textures as an $(n+1)$ -D texture
 - for $n = 1$ or 2
- includes mipmap levels for each texture
- requires only a single texture sampler in shader
 - `sampler{1,2}DArray`



(image courtesy of Mark Kilgard)

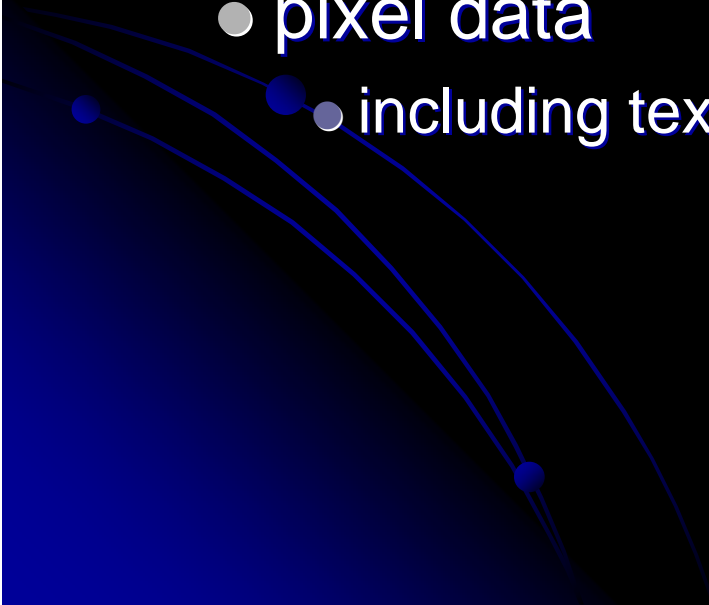
Texture and Renderbuffer Formats

- Non-normalized signed and unsigned integer internal color formats
 - integer values are preserved in their original format
 - new internal texture formats
 - introduces new sampler types for shaders
 - `isampler*`, and `usampler*`
- 

Texture and Renderbuffer Formats (cont'd)

- “R” and “RG” formats
 - adds one- and two-component formats
 - different than luminance & luminance-alpha
 - RGTC texture compression specifically for these formats
- Depth-stencil formats
 - allow access to packed depth/stencil formats

16-bit Half-float formats

- s10e5 floating-point value encoded in an unsigned short
 - Use with
 - vertex attribute arrays
 - pixel data
 - including textures and pixel-rectangles
- 

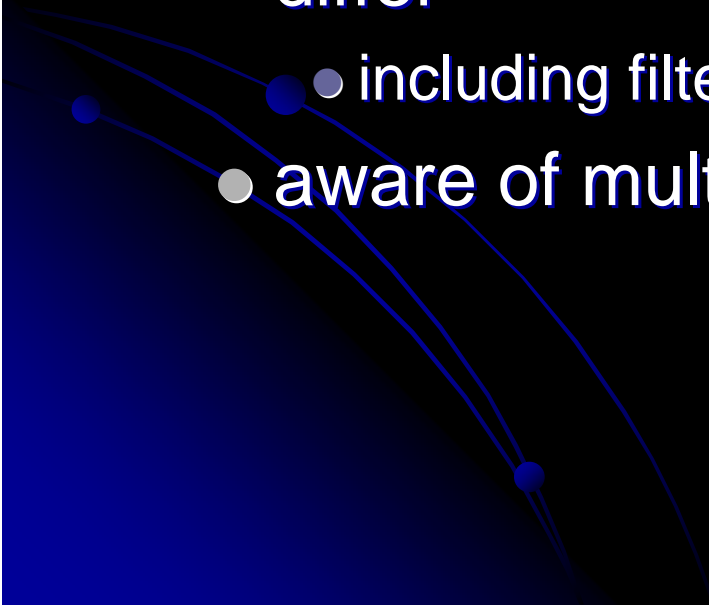
Vertex Array Objects

- Encapsulates multiple vertex attribute arrays into a single bindable object
 - “one bind to rule them all”
 - includes all related state
 - types, stride, enables, etc.
- Think texture objects
 - bind to create
 - set state
 - bind to use

Framebuffer Formats

- Floating-point color- and depth-buffers
 - linear-color space with a floating-point representation
 - 10-, 11-, and 16-bit floating-point
 - used with packed and shared-exponent formats
- sRGB color-buffer format
 - gamma-corrected color space
 - used to only be supported for textures

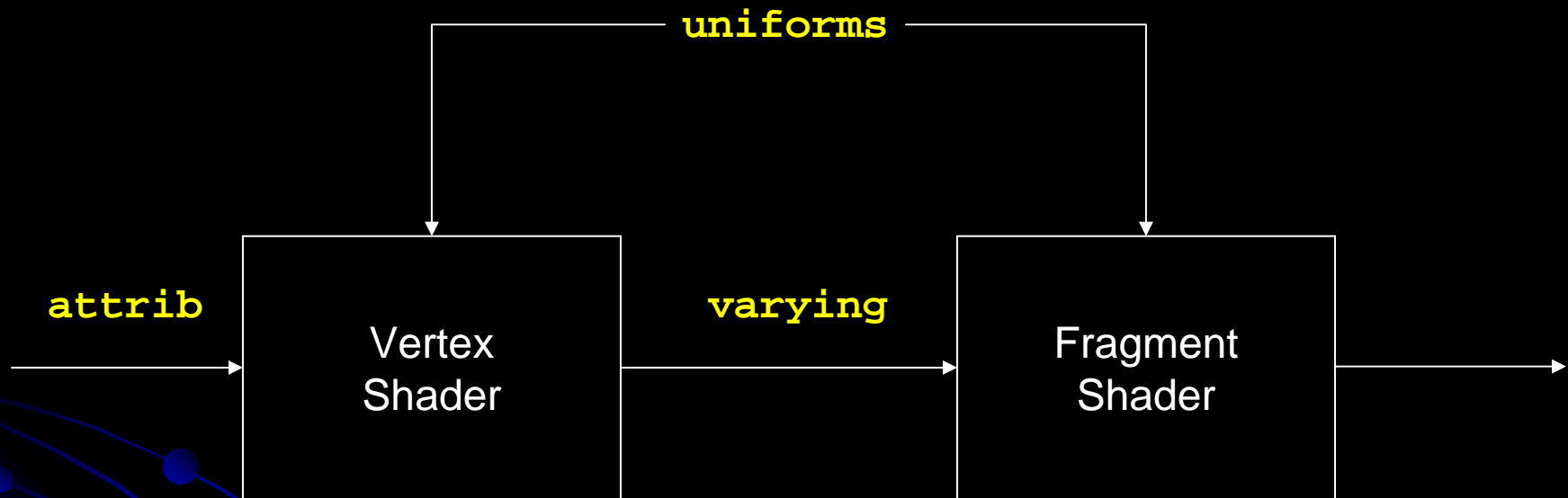
Framebuffer Blits

- `glBlitFramebuffer` – the Swiss Army knife of pixel copies
 - copies multiple buffers in a single call
 - automatically scales pixel rectangle if sizes differ
 - including filtering control
 - aware of multi-sample buffers
- 

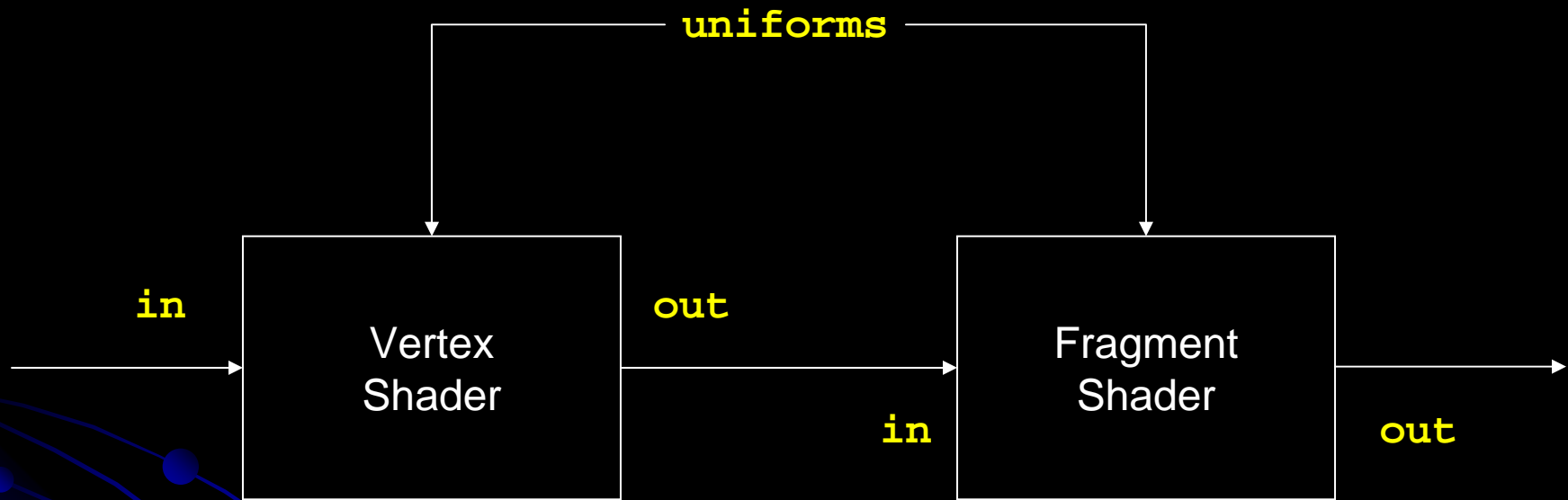
GLSL 1.30

- Shader support for all relevant new features
 - integer attributes, uniforms, and textures
 - integer shifts & masking operations
 - texture arrays
 - texture size queries: dimensions and dimensionality
- Improved flow control
 - switch statements
- Preprocessor enhancements
 - token pasting
- New built-in functions

GLSL 1.30 (cont'd)



GLSL 1.30 (cont'd)



The OpenGL Deprecation Model

- What's it mean?
- Context types
 - *full* context – all current features
 - *forward* context – deprecated features removed
 - GL_INVALID_OPERATION returned for any use of deprecated features
 - new {GLX,WGL,...} context creation routines

Features Deprecated in OpenGL 3.0

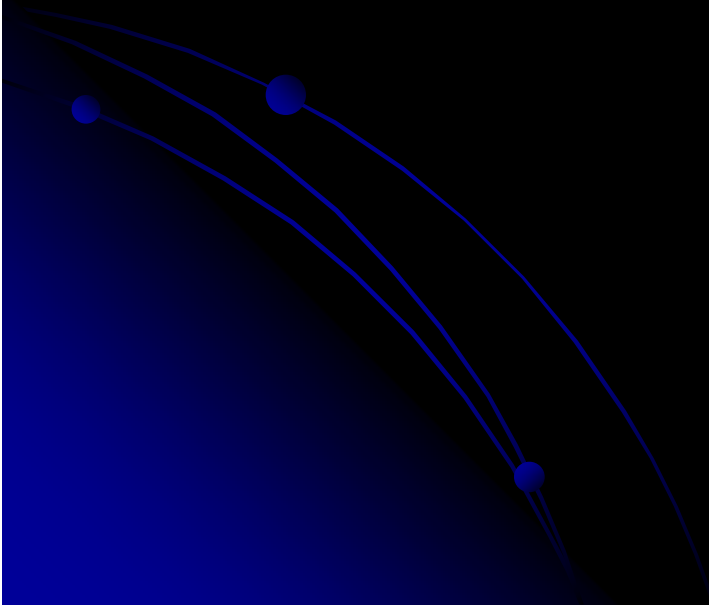
- Application generated object names
 - all objects must be *Generated now
- Color-index mode
- GLSL versions 1.10 and 1.20
- Fixed-function pipeline
 - Begin / End & Named Pointer arrays
 - GL_QUADS, GL_QUAD_STRIP, & GL_POLYGON
 - Matrix stacks
 - Texture coordinate generation
 - Lighting
 - client-side vertex arrays

Features Deprecated in OpenGL 3.0 (cont'd)

- Fixed-function texturing
 - no texture borders, GL_CLAMP mode
 - texture residency and priority
- Draw & copy pixels
 - `glWindowPos` & `glRasterPos`
- Alpha test
- Two-sided color selection
- Polygon mode & stipples
- Accumulation Buffers

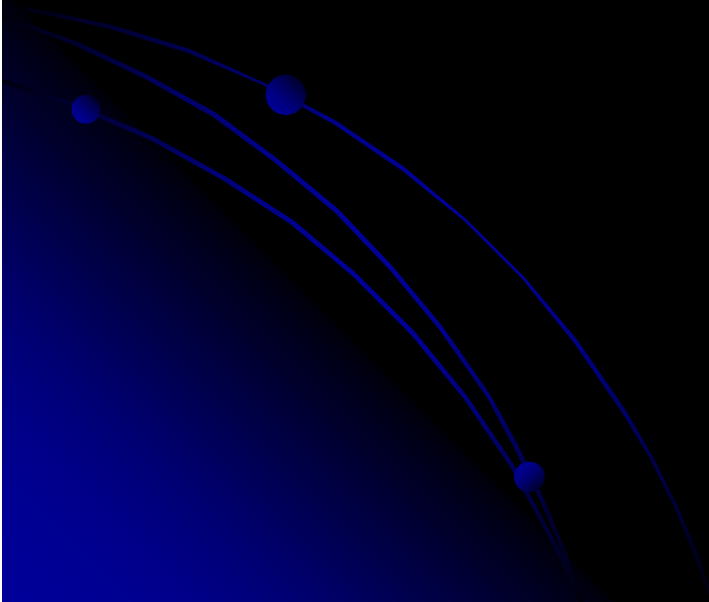
Features Deprecated in OpenGL 3.0 (cont'd)

- Attribute stacks
- Selection and feedback modes
- Evaluators
- GLSL `gl_` names

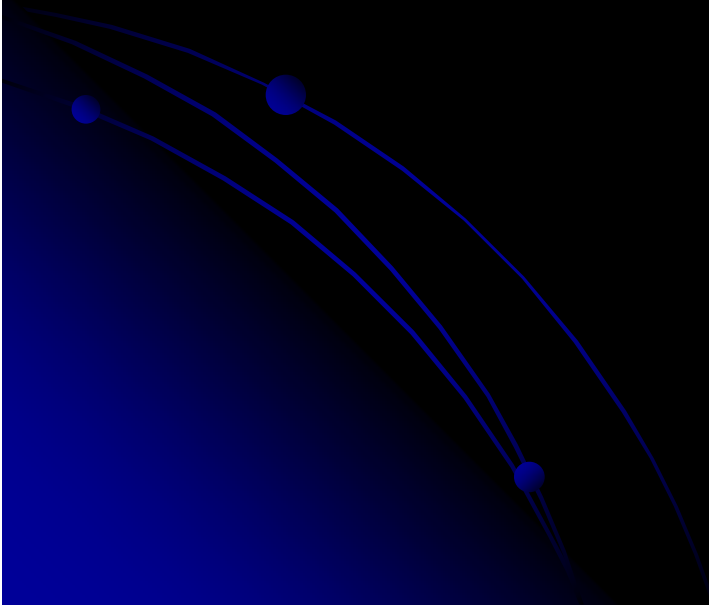


What's Important to Remember

- Nothing's removed in 3.0
 - it's fully backwards compatible
- This is just a list of things that *might* be removed in the future

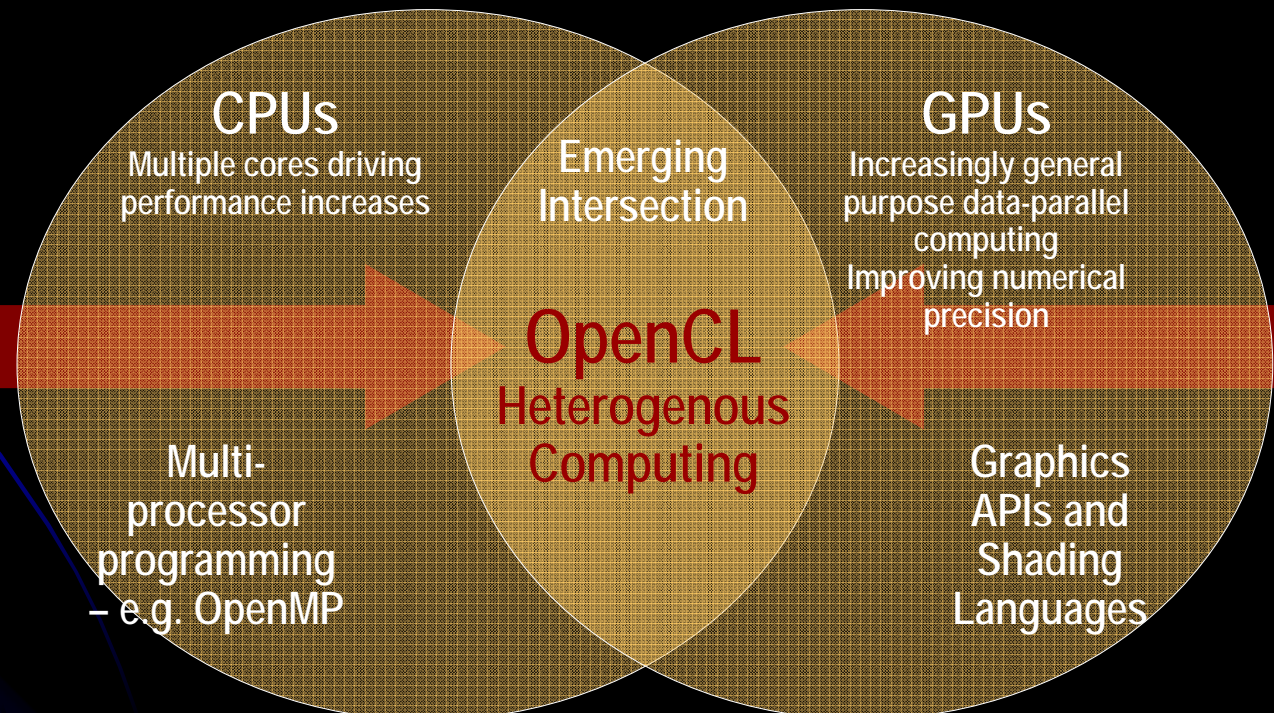


OpenCL



Why another API?

- Recognized the convergence of computational devices
 - GPUs as CPUs




Programming with OpenCL

- Will seem pretty familiar to graphics programmers
 1. create a suitable context
 2. query devices
 3. create kernel programs
 4. load data buffers
 5. specify division of work and execute kernel programs

Contexts and Queues

- Contexts are used to contain and manage the state of the “world”
- Kernels are executed in contexts defined and manipulated by the host
 - Devices
 - Kernels - OpenCL functions
 - Program objects - kernel source and executable
 - Memory objects

Contexts and Queues (cont'd)

- **Command-queue - coordinates execution of kernels**
 - Kernel execution commands
 - Memory commands - transfer or mapping of memory object data
 - Synchronization commands - constrains the order of commands
 - **Applications queue compute kernel execution instances**
 - Queued in-order
 - Executed in-order or out-of-order
 - Events are used to implement appropriate synchronization of execution instances
- 

OpenCL Kernels

- If you were a graphics person, you'd call them a shader
- however, they're not quite ...
 - different programming language
 - more flexible memory mode

```
__kernel void scaleVector( __global const float  *s,  
                           __global const float4 *a,  
                           __global          float4 *b )  
{  
    int id = get_global_id( 0 );  
  
    b[id] = s[id] * a[id];  
}
```

OpenCL C – The Kernel Language

- Subset of c99 with extensions
 - vector types: `<type>2`, `<type>4`, ..., `<type>16`
 - swizzle-type addressing
 - `float16 v; v.s02468ACE = ... // access all the even elements`
 - `float4 v; v.xyzw = ... // just like GLSL`
 - memory access controls
 - rich libm-like library

Language Highlights

- Function qualifiers
 - `__kernel` qualifier declares a function as a kernel
 - Kernels can call other kernel functions
- Address space qualifiers
 - `__global`, `__local`, `__constant`, `__private`
 - Pointer kernel arguments must be declared with an address space qualifier

Language Highlights (cont'd)

- Work-item functions
 - Query work-item identifiers
 - `get_work_dim()`
 - `get_global_id()`, `get_local_id()`, `get_group_id()`
- Image functions
 - Images must be accessed through built-in functions
 - Reads/writes performed through sampler objects from host or defined in source
- Synchronization functions
 - Barriers - all work-items within a work-group must execute the barrier function before any work-item can continue
 - Memory fences - provides ordering between memory operations

Host Initialization Code Example (cont'd)

```
// create the program from the text string "program_source"
cl_program program = clCreateProgramWithSource(context, 1,
                                              &program_source,
                                              NULL, NULL);

// build the program
cl_int err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

// create the kernel
cl_kernel kernel = clCreateKernel(program, "scaleVector", NULL);

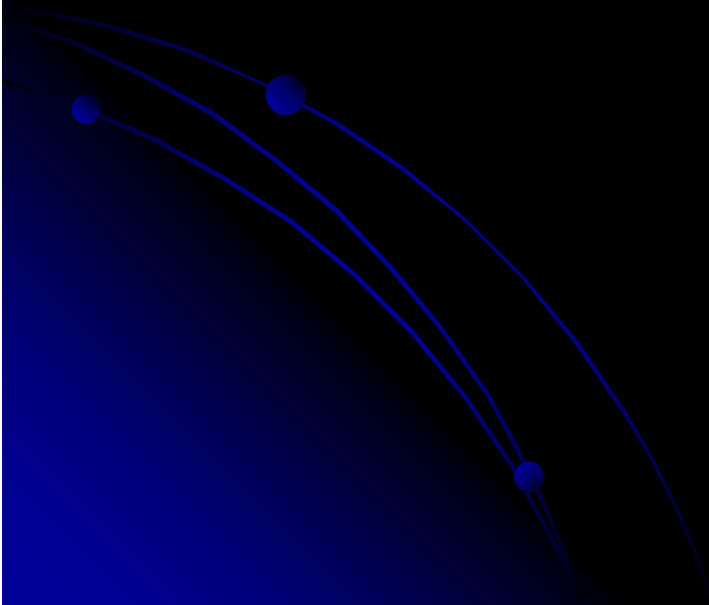
// set the args values
for ( int i = 0; i < 3; ++i ) {
    err |= clSetKernelArg(kernel, i, (void *)&values[i],
                          sizeof(cl_mem));
}
```

Host Initialization Code Example (cont'd)

```
size_t global_work_size[0] = n;

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL,
                              global_work_size, NULL, 0, NULL,
                              NULL);

// copy output array into host array "dst"
err = clEnqueueReadBuffer(context, values[2], CL_TRUE, 0,
                           n*sizeof(cl_float4), dst, 0, NULL,
                           NULL);
```



Kernel Objects

- Kernel objects encapsulate
 - Specific kernel functions declared in a program
 - Argument values used for kernel execution
- Creating kernel objects
 - `clCreateKernel()` - creates a kernel object for a single function in a program
 - `clCreateKernelsInProgram()` - creates an object for all kernels in a program

Kernel Object (cont'd)

- Setting arguments

- `clSetKernelArg(<kernel>, <argument index>)`
- Each argument data must be set for the kernel function
- Argument values are copied and stored in the kernel object

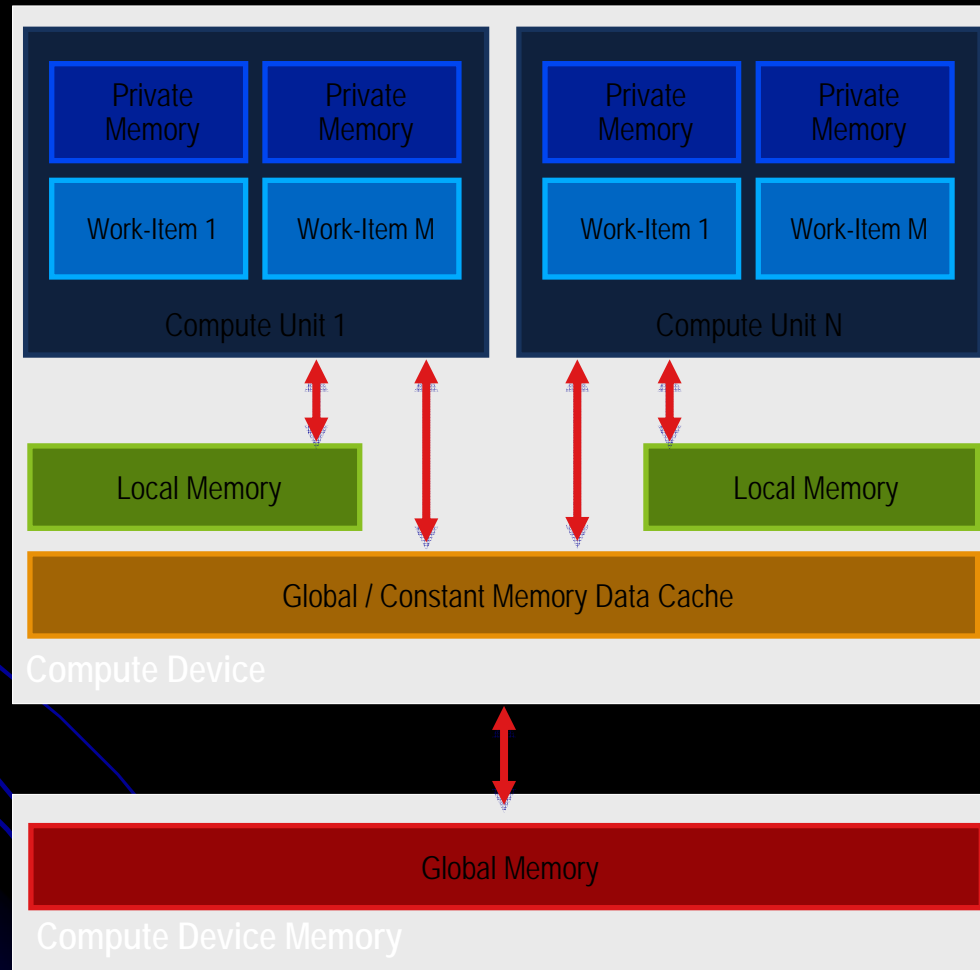
- Kernel vs. program objects

- Kernels are related to program execution
- Programs are related to program source

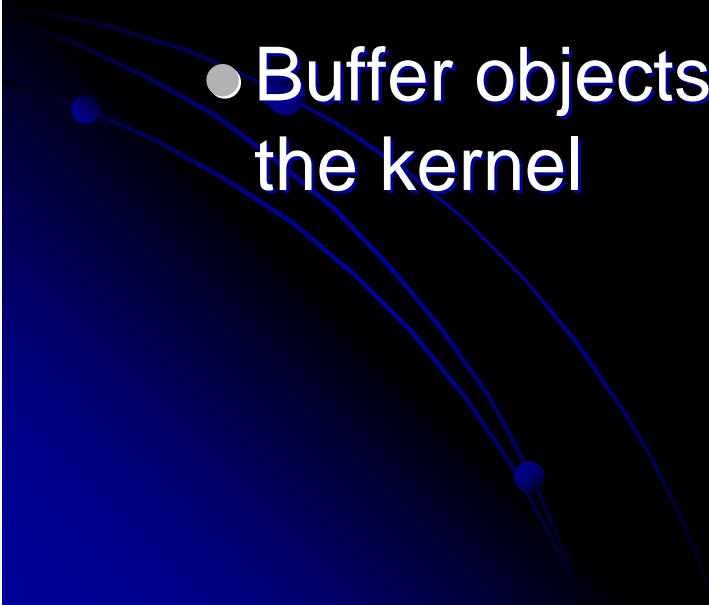
OpenCL Memory Layout

- kernel-accessed memory controlled by access modifiers
 - `__global` – accessible by all work-groups
 - `__local` – accessible only to members of a work-group
 - `__private` – local to a work-item's execution
 - `__constant` – constant global memory

OpenCL Memory Layout



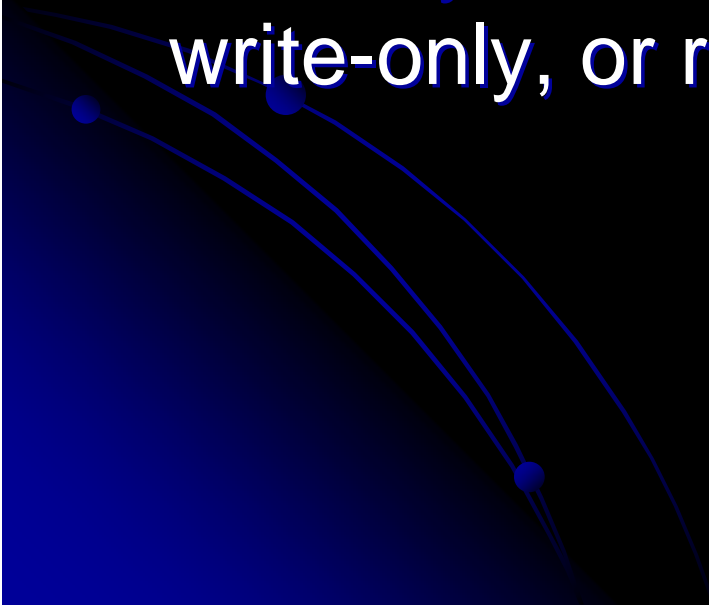
Memory Objects

- Buffer objects
 - One-dimensional collection of objects (like C arrays)
 - Valid elements include scalar and vector types as well as user defined structures
 - Buffer objects can be accessed via pointers in the kernel
- 

Memory Objects (cont'd)

- Image objects
 - Two- or three-dimensional texture, frame-buffer, or images
 - Must be addressed through built-in functions
- Sampler objects
 - Describes how to sample an image in the kernel
 - Addressing modes
 - Filtering modes

Creating Memory Objects

- `clCreateBuffer()`, `clCreateImage2D()`, and `clCreateImage3D()`
 - Memory objects are created with an associated context
 - Memory can be created as read-only, write-only, or read-write
- 

Creating Memory Objects (cont'd)

- Where objects are created in the platform memory space can be controlled
 - Device memory
 - Device memory with data copied from a host pointer
 - Host memory
 - Host memory associated with a pointer
 - Memory at that pointer is guaranteed to be valid at synchronization points
- Image objects are also created with a channel format
 - Channel order (e.g., RGB, RGBA ,etc.)
 - Channel type (e.g., UNORM INT8, FLOAT, etc.)

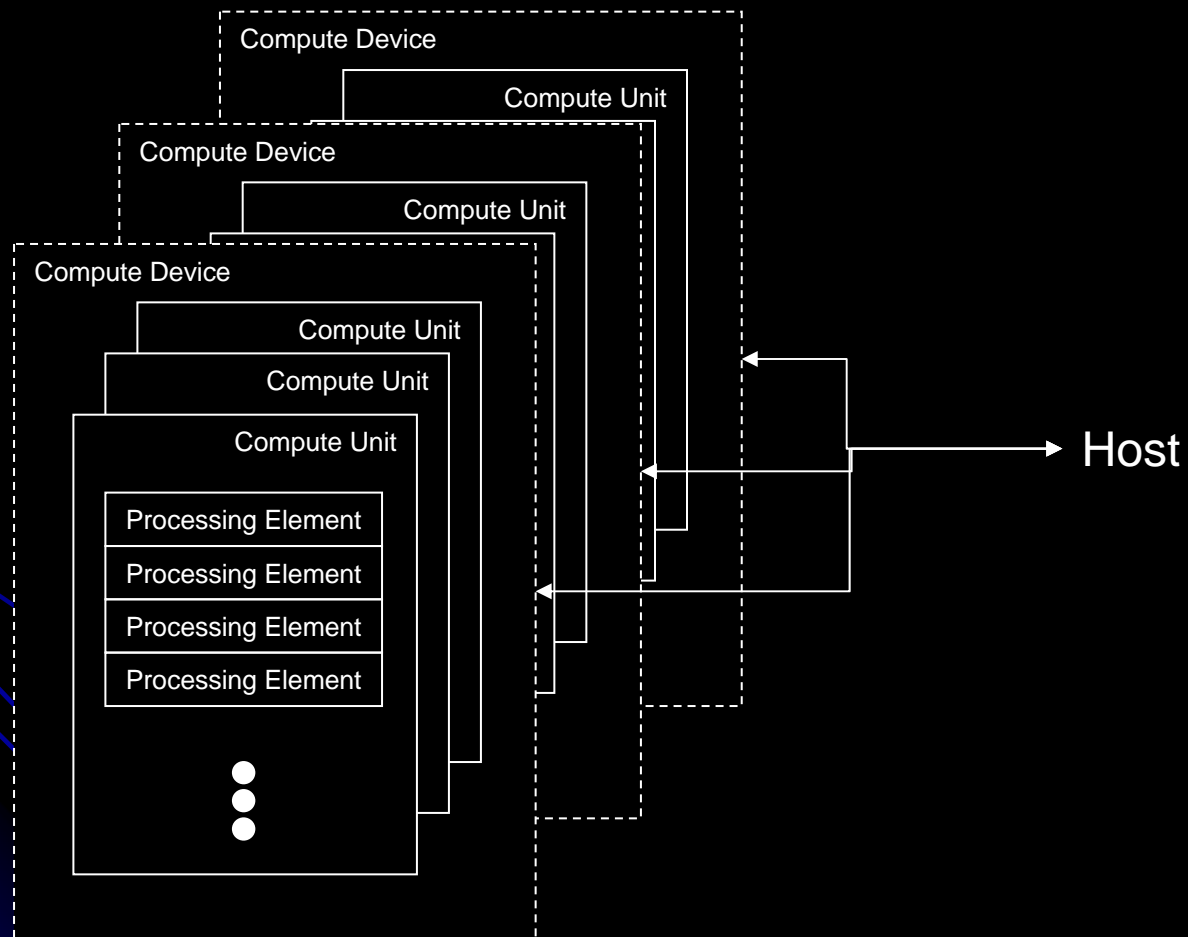
Manipulating Object Data

- Object data can be copied to host memory, from host memory, or to other objects
- Memory commands are enqueued in the command buffer and processed when the command is executed
 - `clEnqueueReadBuffer()`, `clEnqueueReadImage()`
 - `clEnqueueWriteBuffer()`, `clEnqueueWriteImage()`
 - `clEnqueueCopyBuffer()`, `clEnqueueCopyImage()`

Manipulating Object Data (cont'd)

- Data can be copied between Image and Buffer objects
 - `clEnqueueCopyImageToBuffer()`
 - `clEnqueueCopyBufferToImage()`
- Regions of the object data can be accessed by mapping into the host address space
 - `clEnqueueMapBuffer()`, `clEnqueueMapImage()`
 - `clEnqueueUnmapMemObject()`

Compute Architecture



Kernel Execution

- A command to execute a kernel must be enqueued to the command-queue
- `clEnqueueNDRangeKernel()`
 - Data-parallel execution model
 - Describes the *index space* for kernel execution
 - Requires information on NDRange dimensions and work-group size

Kernel Execution (cont'd)

- **clEnqueueTask()**
 - Task-parallel execution model (multiple queued tasks)
 - Kernel is executed on a single work-item
- **clEnqueueNativeKernel()**
 - Task-parallel execution model
 - Executes a native C/C++ function not compiled using the OpenCL compiler
 - This mode does not use a kernel object so arguments must be passed in

OpenCL Execution Model

- Define N-Dimensional computation domain
 - Each independent element of execution in an N-Dimensional domain is called a *work-item*
 - N-Dimensional domain defines the total number of work-items that execute in parallel = *global work size*
- Work-items can be grouped together — *work-group*
 - Work-items in group can communicate with each other
 - Can synchronize execution among work-items in group to coordinate memory access

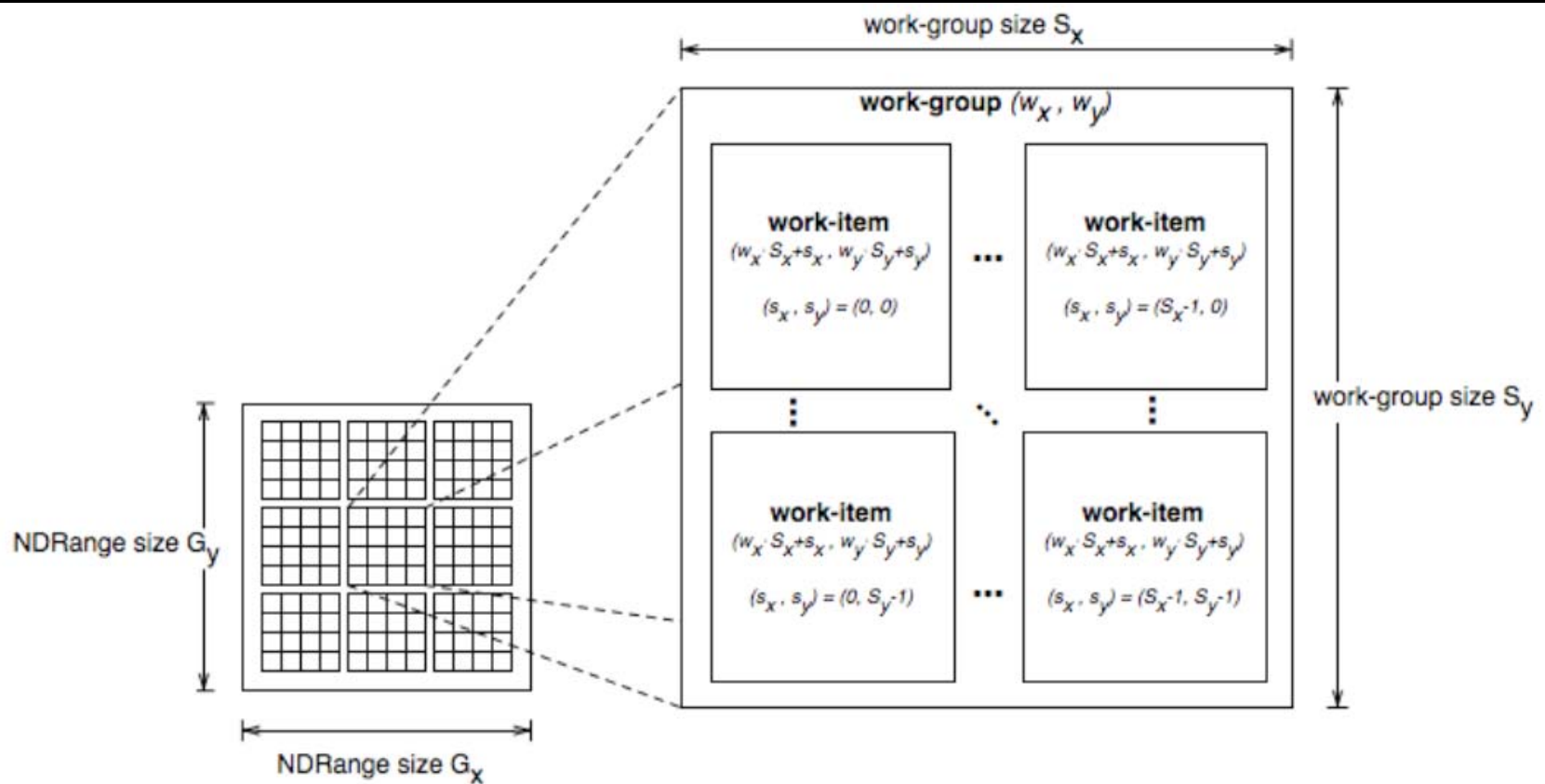
OpenCL Execution Model (cont'd)

- Execute multiple work-groups in parallel
 - Mapping of global work size to work-group can be implicit or explicit
- Compute kernels can execute task-parallel loads
 - Executes as a single work-item
 - A compute kernel written in OpenCL
 - A native C / C++ function

Kernel Execution

- The host program invokes a kernel over an index space called an ***NDRange***
 - NDRange can be a 1, 2, or 3-dimensional space
- A single kernel instance at a point in the index space is called a ***work-item***
 - Work-items have unique global IDs from the index space

NDRanges and Work-items



NDRanges and Work-items (cont'd)

- Work-items are further grouped into ***work-groups***
 - Work-groups have a unique work-group ID
 - Work-items have a unique local ID within a work-group
- Total number of work-items = $G_x \times G_y$
- Size of each work-group = $S_x \times S_y$
- Global ID can be computed from work-group ID and local ID

Thanks!

Questions?

shreiner@siggraph.org

