OpenCL 2.0 Overview

Allen Hux
Intel Corporation

Goals

- Enable New Programming Patterns
- Performance Improvements
- Well-defined Execution & Memory Model
- Improve CL / GL sharing

Shared Virtual Memory

- In OpenCL 1.2 buffer objects can only be passed as kernel arguments
- Buffer object described as pointer to type in kernel
- Restrictions
  - Pass a pointer + offset as argument value
  - Store pointers in buffer object(s)
- Why?
  - Host and OpenCL device may not share the same virtual address space
  - No guarantee that the same virtual address will be used for a kernel argument across multiple enqueues

Shared Virtual Memory

- **clSVMAlloc - allocates a shared virtual memory buffer**
  - Specify size in bytes
  - Specify usage information
  - Optional alignment value
- **SVM pointer can be shared by the host and OpenCL device**
- **Examples**

```
clSVMAlloc(ctx, CL_MEM_READ_WRITE, 1024 * 1024, 0)

clSVMAlloc(ctx, CL_MEM_READ_ONLY, 1024 * 1024, sizeof(cl_float4))
```

- **Free SVM buffers**
  - clEnqueueSVMFree, clSVMFree

Shared Virtual Memory

- **clSetKernelArgSVMPointer**
  - SVM pointers as kernel arguments
  - A SVM pointer
  - A SVM pointer + offset

```
kernel void
vec_add(float *src, float *dst)
{
    size_t id = get_global_id(0);
    dst[id] += src[id];
}
```

```
// allocating SVM pointers
cl_float *src = (cl_float *)clSVMAlloc(ctx, CL_MEM_READ_ONLY, size, 0);
cl_float *dst = (cl_float *)clSVMAlloc(ctx, CL_MEM_READ_WRITE, size, 0);

// Passing SVM pointers as arguments
clSetKernelArgSVMPointer(vec_add_kernel, 0, src);
clSetKernelArgSVMPointer(vec_add_kernel, 1, dst);

// Passing SVM pointer + offset as arguments
clSetKernelArgSVMPointer(vec_add_kernel, 0, src + offset);
clSetKernelArgSVMPointer(vec_add_kernel, 1, dst + offset);
```

Shared Virtual Memory

- clSetKernelExecInfo
  - Passing SVM pointers in other SVM pointers or buffer objects

```
// allocating SVM pointers
my_info_t *pA = (my_info_t *)clSVMAlloc(ctx,
          CL_MEM_READ_ONLY, sizeof(my_info_t), 0);
pA->pB = (cl_float *)clSVMAlloc(ctx,
          CL_MEM_READ_WRITE, size, 0);


// Passing SVM pointers
clSetKernelArgSVMPointer(my_kernel, 0, pA);

clSetKernelExecInfo(my_kernel,
                    CL_KERNEL_EXEC_INFO_SVM_PTRS,
                    1 * sizeof(void *), &pA->pB);
```

```
typedef struct {
   …
   float *pB;
   …
} my_info_t;

kernel void
my_kernel(global my_info_t *pA, …)
{
    …
    do_stuff(pA->pB, …);
    …
}
```

Shared Virtual Memory

- **Three types of sharing**
  - Coarse-grained buffer sharing
  - Fine-grained buffer sharing
  - System sharing

Shared Virtual Memory - Coarse & Fine Grained

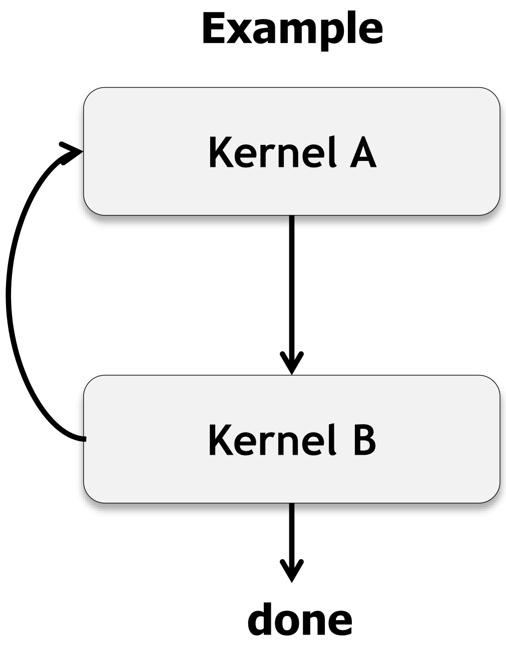
- **SVM buffers allocated using clSVMAlloc**
- **Coarse grained sharing**
  - Memory consistency only guaranteed at synchronization points
  - Host still needs to use synchronization APIs to update data
    - clEnqueueSVMMap / clEnqueueSVMUnmap or event callbacks
    - Memory consistency is at a buffer level
  - Allows sharing of pointers between host and OpenCL device
- **Fine grained sharing**
  - No synchronization needed between host and OpenCL device
    - Host and device can update data in buffer concurrently
    - Memory consistency using C11 atomics and synchronization operations
  - Optional Feature

Shared Virtual Memory - System Sharing

- **Can directly use any pointer allocated on the host**
  - No OpenCL APIs needed to allocate SVM buffers
- **Both host and OpenCL device can update data using C11 atomics and synchronization functions**
- **Optional Feature**

Nested Parallelism

- In OpenCL 1.2 only the host can enqueue kernels
- Iterative algorithm example
  - kernel A queues kernel B
  - kernel B decides to queue kernel A again
- Requires host - device interaction and for the host to wait for kernels to finish execution
  - Can use callbacks to avoid waiting for kernels to finish but still overhead
- A very simple but extremely common nested parallelism example



Nested Parallelism

- **Allow a device to queue kernels to itself**
  - Allow a work-item(s) to queue kernels
- **Use similar approach to how host queues commands**
  - Queues and Events
  - Functions that queue kernels and other commands
  - Event and Profiling functions

Nested Parallelism

- Use clang Blocks to describe kernel to queue

```
kernel void my_func(global int *a, global int *b)
{
    …
    void (^my_block_A)(void) =
            ^{
                size_t id = get_global_id(0);
                b[id] += a[id];
             };

   enqueue_kernel(get_default_queue(),
                  CLK_ENQUEUE_FLAGS_WAIT_KERNEL,
                  ndrange_1D(…),
                  my_block_A);
}
```

Nested Parallelism

```
int enqueue_kernel(queue_t queue,
                   kernel_enqueue_flags_t flags,
                   const ndrange_t ndrange,
                   void (^block)())

int enqueue_kernel(queue_t queue,
                   kernel_enqueue_flags_t flags,
                   const ndrange_t ndrange,
                   uint num_events_in_wait_list,
                   const clk_event_t *event_wait_list,
                   clk_event_t *event_ret,
                   void (^block)())
```

Nested Parallelism

- Queuing kernels with pointers to local address space as arguments

```
int enqueue_kernel(queue_t queue,
                   kernel_enqueue_flags_t flags,
                   const ndrange_t ndrange,
                   void (^block)(local void *, …), uint size0, …)

int enqueue_kernel(queue_t queue,
                   kernel_enqueue_flags_t flags,
                   const ndrange_t ndrange,
                   uint num_events_in_wait_list,
                   const clk_event_t *event_wait_list,
                   clk_event_t *event_ret,
                   void (^block)(local void *, …), uint size0, …)
```

Nested Parallelism

- Example showing queuing kernels with local address space arguments

```
void my_func_local_arg (global int *a, local int *lptr, …) { … }

kernel void my_func(global int *a, …)
{
    …
    uint local_mem_size = compute_local_mem_size(…);

    enqueue_kernel(get_default_queue(),
                   CLK_ENQUEUE_FLAGS_WAIT_KERNEL,
                   ndrange_1D(…),
                   ^(local int *p){my_func_local_arg(a, p, …);},
                   local_mem_size);
}
```

Nested Parallelism

- **Specify when a child kernel can begin execution (pick one)**
  - Don’t wait on parent
  - Wait for kernel to finish execution
  - Wait for work-group to finish execution

- **A kernel’s execution status is complete**
  - when it has finished execution
  - *and* all its child kernels have finished execution

Nested Parallelism

- **Other Commands**
  - Queue a marker
- **Query Functions**
  - Get workgroup size for a block
- **Event Functions**
  - Retain & Release events
  - Create user event
  - Set user event status
  - Capture event profiling info
- **Helper Functions**
  - Get default queue
  - Return a 1D, 2D or 3D ND-range descriptor

Generic Address Space

- In OpenCL 1.2, function arguments that are a pointer to a type must declare the address space of the memory region pointed to
- Many examples where developers want to use the same code but with pointers to different address spaces

```
void
my_func (local int *ptr, …)
{
    …
    foo(ptr, …);
    …
}
```

```
void
my_func (global int *ptr, …)
{
    …
    foo(ptr, …);
    …
}
```

- Above example is not supported in OpenCL 1.2
- Results in developers having to duplicate code

Generic Address Space

- OpenCL 2.0 no longer requires an address space qualifier for arguments to a function that are a pointer to a type
  - Except for kernel functions
- Generic address space assumed if no address space is specified
- Makes it really easy to write functions without having to worry about which address space arguments point to

```
void
my_func (int *ptr, …)
{
    …
}

kernel void
foo(global int *g_ptr, local int *l_ptr, …)
{
    …
    my_func(g_ptr, …);
    my_func(l_ptr, …);
}
```

Generic Address Space - Casting Rules

- Implicit casts allowed from named to generic address space
- Explicit casts allowed from generic to named address space
- Cannot cast between constant and generic address spaces

```
kernel void foo()
{
    int *ptr;
    local int *lptr;
    global int *gptr;
    local int val = 55;

    ptr = gptr; // legal
    lptr = ptr; // illegal
    lptr = gptr; // illegal
    ptr = &val; // legal
    lptr = (local int *)ptr; // legal
}
```

Generic Address Space - Built-in Functions

- global gentype* to_global(const gentype*)
  local gentype* to_local(const gentype *)
  private gentype* to_private(const gentype *)
  - Returns NULL if cannot cast
- cl_mem_fence_flags get_fence(const void *ptr)
  - Returns the memory fence flag value
  - Needed by work_group_barrier and mem_fence functions

KHRONOS GROUP

C11 Atomics

- **Implements a subset of the C11 atomic and synchronization operations**
  - Enable assignments in one work-item to be visible to others
- **Atomic operations**
  - loads & stores
  - exchange, compare & exchange
  - fetch and modify (add, sub, or, xor, and, min, max)
  - test and set, clear
- **Fence operation**
- **Atomic and Fence operations take**
  - Memory order
  - Memory scope
- **Operations are supported for global and local memory**

C11 Atomics

- **memory_order_relaxed**
  - Atomic operations with this memory order are not synchronization operations
  - Only guarantee atomicity
- **memory_order_acquire, memory_order_release, memory_order_acq_rel**
  - Atomic store in work-item A for variable M is tagged with memory_order_release
  - Atomic load in work-item B for same variable M is tagged with memory_order_acquire
  - Once the atomic load is completed work-item B is guaranteed to see everything work-item A wrote to memory before atomic store
  - Synchronization is only guaranteed between work-items releasing and acquiring the same atomic variable
- **memory_order_seq_cst**
  - Same as memory_order_acq_rel, and
  - A single total order exists in which all work-items observe all modifications

C11 Atomics

- **Memory scope - specifies scope of memory ordering constraints**
  - Work-items in a work-group
  - Work-items of a kernel executing on a device
  - Work-items of a kernel & host threads executing across devices and host
    - For shared virtual memory

C11 Atomics

- **Supported Atomic Types**
  - atomic_int, atomic_uint
  - atomic_long, atomic_ulong
  - atomic_float
  - atomic_double
  - atomic_intptr_t, atomic_uintptr_t, atomic_ptrdiff_t
  - atomic_size_t
  - atomic_flag
- **Atomic types have the same size & representation as the non-atomic types except for atomic_flag**
- **Atomic functions must be lock-free**

Images

- **2D image from buffer**
  - GPUs have dedicated and fast hardware for texture addressing & filtering
  - Accessing a buffer as a 2D image allows us to use this hardware
  - Both buffer and 2D image use the same data storage
- **Reading & writing to an image in a kernel**
  - Declare images with the read_write qualifier
  - Use barrier between writes and reads by work-items to the image
    - work_group_barrier(CLK_IMAGE_MEM_FENCE)
  - Only sampler-less reads are supported

Images

- Writes to 3D images is now a core feature
- New image formats
  - sRGB
  - Depth
- Extended list of required image formats
- Improvements to CL / GL sharing
  - Multi-sampled GL textures
  - Mip-mapped GL textures

Pipes

- Memory objects that store data organized as a FIFO
- Kernels can read from or write to a pipe object
- Host can only create pipe objects

Pipes

- **Why introduce a pipe object?**
  - Allow vendors to implement dedicated hardware to support pipes
  - Read from and write to a pipe without requiring atomic operations to global memory
  - Enable producer – consumer relationships between kernels

Pipes - Read & Write Functions

- **Work-item read pipe functions**
  - Read a packet from a pipe
  - Read with reservation
    - Reserve n packets for reading
    - Read individual packets (identified by reservation ID and packet index)
    - Confirm that the reserved packets have been read
- **Work-item write pipe functions**
  - Write a packet to a pipe
  - Write with reservation
- **Work-group pipe functions**
  - Reserve and commit packets for reading / writing

Other 2.0 Features

- Program scope variables
- Flexible work-groups
- New work-item functions
  - get_global_linear_id, get_local_linear_id
- Work-group functions
  - broadcast, reduction, vote (any & all), prefix sum
- Sub-groups
- Sharing with EGL images and events

