int
enqueue_kernel(
    queue_t queue,
    kernel_enqueueflags_t flags,
    const ndrange_t ndrange,
    void (*block)(void))

From the signature, we can see that as with host-side commands, a device-side
enqueue also requires a command-queue. The flags parameter is used to specify
when the child kernel should begin execution. There are three possible options with
the following semantics:

- CLK_ENQUEUE_FLAGS_NO_WAIT: The child kernel can begin executing
  immediately.
- CLK_ENQUEUE_FLAGS_WAIT_KERNEL: The child kernel must wait for the parent
  kernel to reach the ENDED before executing. In this case, the parent kernel has
  finished executing. However, other child kernels could still be executing on the
  device.
- CLK_ENQUEUE_FLAGS_WAIT_WORK_GROUP: The child kernel must wait for the
  enqueuing work-group to complete its execution before starting.
Creating and Passing a Device-Side Queue

```c
// Relevant host program

// Specify the queue properties
cl_command_queue_properties properties =
    CL_QUEUE_ON_DEVICE |
    CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE;

// Create the device-side command-queue
cl_command_queue device_queue;
device_queue = clCreateCommandQueueWithProperties(

    context,
    device,
    &properties,
    NULL);

... clSetKernelArg(kernel, 0, sizeof(cl_command_queue), &device_queue);
...

// Kernel

__kernel
void foo(queue_t myQueue, ...)
{
    ...
}

LISTING 5.9
Passing a device-side command-queue as a kernel argument.

OPTIONAL:
CL_QUEUE_ON_DEVICE_DEFAULT
added to properties ⇒ makes this the default Queue on the device
(See Listing 5.10)
Listing 5.10

A simple example showing child-kernel enqueues using block syntax.
void vecadd(__global int *A, __global int *B, __global int *C)
|
int idx = get_global_id(0);
|
|
As a toy example to illustrate argument passing, we can modify the vector addition so that the parent enqueues a child kernel to execute the work. Listing 5.11 shows argument passing when we are creating a block variable, and Listing 5.12 shows argument passing when we are using a block literal. Notice that in Listing 5.11 arguments are provided similarly to standard function calls. However, when we are using a literal in Listing 5.12, no arguments are passed explicitly. Instead, the compiler establishes a new lexical scope within the parent for the literal. While global variables are bound in the expected manner, private and local data must be copied. Note that pointers to local or private address spaces are invalid, as they do not have scope outside their work-group or work-item, respectively. However, creation of local memory regions for child kernels is supported and is discussed next. Because the return type of a kernel is always void, it never needs to be explicitly defined when declaring a block.

LISTING 5.11
Passing arguments using block syntax.
5.5 Device-side queuing

```c
__kernel
void parent_vccadd(__global int *A, __global int *B, __global int *C);
{
    kernel_enqueue_flags_t child_flags = CLK_ENQUEUE_FLAGS_NO_WAIT;
    ndrange_t child_ndrange = ndrange_id(get_global_size(0));

    // Only enqueue one child kernel
    if (get_global_id(0) == 0) {
        // Enqueue kernel from block literal
        enqueue_kernel(
                get_default_queue(),
                child_flags,
                child_ndrange,
                ^[int idx = get_global_id(0); C[idx] = A[idx] + B[idx];]:)
    }
}
```

Listing 5.12

Accessing arguments from lexical scope.

Dynamic local memory

When setting arguments using the host API, we can dynamically allocate local memory for a kernel by providing a NULL pointer to clSetKernelArg(). Since we do not have a similar mechanism for setting child-kernel arguments, the enqueue_kernel() function has been overloaded:

```c
int enqueue_kernel(
        queue_t queue,
        kernel_enqueue_flags_t flags,
        const ndrange_t ndrange,
        void (*)[block](local void *, ...),
        uint size0, ...)
```

To create local memory pointers, the specification defines the block argument to be a variadic function (a function that receives a variable number of arguments). Each argument must be of type local void *. Notice that in the declaration that this argument list replaces the existing void type. The enqueue_kernel() function is also variadic, ending with a variable number of parameters that represent the size of each local array. Listing 5.13 modifies the toy vector addition example to use local memory in order to illustrate the use of dynamic local memory allocation with block syntax.
// When a kernel has been defined like this, then it can be
// enqueued from the host as well as from the device

//_kernel
void child_vecadd(_-global int *A, _-global int *B, _-global int *C,
                  _-local int *local_A, _-local int *local_B,
                  _-local int *local_C)
{
    int idx = get_global_id(0);
    int local_idx = get_local_id(0);
    local_A[local_idx] = A[idx];
    local_B[local_idx] = B[idx];
    local_C[local_idx] = local_A[local_idx] + local_B[local_idx];
    C[idx] = local_C[local_idx];
}

//_kernel
void parent_vecadd(_-global int* A, _-global int*B, _-global int* C)
{
    kernel_enqueue_flags_t child_flags = CLK_ENQUEUE_FLAGS_NO_WAIT;
    nrange_t child_nrange = nrange_id(get_global_size(0));

    int local_A_mem_size = sizeof(int)*1 ;
    int local_B_mem_size = sizeof(int)*1 ;
    int local_C_mem_size = sizeof(int)*1 ;

    // Define a block with local memory for each local memory
    // argument of the kernel
    void (^child_vecadd_blk)(local int *, local int *,
                              local int * =
                              (local int *local_A, local int * local_B,
                              local int *
                              local_C)
    {
        child_vecadd(A, B, C, local_A, local_B,
                     local_C);
    };

    // Only enqueue one child kernel
    if(get_global_id(0)==0)
    {
        // Variadic enqueue_kernel function takes in local
        // memory size of each argument in block
        enqueue_kernel(
            get_default_queue(),
            child_flags,
            child_nrange,
            child_vecadd_blk,
            local_A_mem_size,
            local_B_mem_size,
            local_C_mem_size);
    }
}

LISTING 5.13

Allocating dynamic local memory within a child kernel.
Enforcing dependencies using events
When introducing device-side command-queues, we mentioned that the queues always operate as out-of-order queues. This implies that there must be some mechanism provided to enforce dependency requirements. As on the host, events are used to satisfy this requirement when kernels are enqueued directly from a device. Once again, the enqueue_kernel() function is overloaded to provide this additional functionality:

```c
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)(void))
```

The reader should notice that the three additional parameters, num_events_in_wait_list, event_wait_list, and event_ret, mirror event-related parameters from the host API functions.

The final signature of enqueue_kernel() provides support for events and local memory to be utilized together:

```c
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, ...
```

1. **__kernel**
2. **void child0_kernel()**
3. {
4.     printf("Child0: I will run first.\n");
5. }
6. **void child1_kernel()**
7. {
8.     printf("Child1: I will run second.\n");
9. }
10. **__kernel**
11. **void parent_kernel( )**
12. {
13.     kernel_enqueue_flags_t child_flags = CLK_ENQUEUE_FLAGS_NO_WAIT;
14.     ndrange_t child_ndrange = ndrange_ID(1);
15.     clk_event event;
16.     // Enqueue a kernel and initialize an event
17.     enqueue_kernel(get_default_queue(), child Flags, child ndrange, 0, NULL, &event, *(child0 kernel()));
18.     // Pass the event as a dependency between the kernels
19.     enqueue_kernel(get_default_queue(), child Flags, child ndrange, 1, &event, NULL, *(child1 kernel()));
20.     // Release the event. In this case, the event will be released
21.     // after the dependency is satisfied (second kernel is ready
22.     // to execute).
23.     release_event(event);
24. }

**LISTING 5.14**

An example showing device-side enqueuing with events used to specify dependencies.