Chapter 5

Understanding OpenCL’s Concurrency and Execution Model

Introduction

As discussed in Chapter 3, there is a wide range of devices supported by OpenCL. To achieve such wide support, it is vital that the memory and execution models for OpenCL are defined in such a way that we can achieve a high level of performance across a range of architectures without extraordinary programming effort. In this chapter, we delve deeper into these models, and in Chapter 6 we show how they map to a specific architecture that supports OpenCL.

Kernels, Work-Items, Workgroups, and the Execution Domain

OpenCL execution is centered on the concept of a kernel. A kernel is a unit of code that represents a single executing instance as written in the OpenCL C language. A kernel instance is at first sight similar to a C function: In the OpenCL C language, a kernel looks like C, it takes a parameter list, and has “local” variables (in a private address space, as we shall see) and standard control flow constructs. This single kernel instance is known as a work item in OpenCL terminology.

What makes the OpenCL kernel different from a C function is its parallel semantics. Any given kernel instance or work item defines just one sliver of a large parallel execution space. A kernel dispatch, initiated when the runtime processes the entry in an execution queue created by a call to `clEnqueueNDRangeKernel` on a queue object, consists of a large number of work items intended to execute together to carry out the collective operations specified in the kernel body. As the enqueue call suggests, this dispatch creates an `NDRange` (an n-dimensional range) worth of work items. An `NDRange` defines a one-, two-, or three-dimensional grid of work items, providing a simple and straightforward structure for parallel execution. When mapped to the hardware model of OpenCL, each work item runs on a unit of hardware abstractly known as a processing element, where a given processing element may process multiple work items in turn.

Within a kernel dispatch, each work item is independent. In OpenCL, global synchronization between work items is not defined. This relaxed execution model allows OpenCL to scale on devices possessing a large number of cores. However, this kind of hardware actually provides a hierarchy of execution devices, particularly a hierarchy of memory structures.

To flexibly support such devices, OpenCL divides the global execution space into a large number of equally sized one-, two-, or three-dimensional sets of work items called workgroups. Within each workgroup, some degree of communication is allowed. The OpenCL specification requires that an entire workgroup can run concurrently on an element of the device hierarchy known as a compute unit. This form of concurrent execution is vital to allow synchronization. Workgroups allow for local synchronization by guaranteeing concurrent execution, but they also limit communication to improve scalability. An application that involves global communication across its execution space is challenging to parallelize to multi-core devices with OpenCL. To satisfy global communications, the compute unit will be mapped onto a single core.

By defining larger dispatches than can execute concurrently, OpenCL kernels can scale onto larger and more heavily threaded devices on which more groups and more work items can execute at once. However, for performance reasons (just as with APIs such as OpenMP and MPI), it may make more sense to only issue enough work that you know can run and more directly control optimization.

As discussed in Chapter 2, OpenCL work items attempt to express parallelism that could be expressed using Win32 or POSIX threads or a more abstract mapping to threads such as OpenMP. The design of OpenCL takes that a step further, however, because the set of work items within a workgroup can be efficiently grouped into a smaller number of hardware thread contexts. This can be viewed as a generalization of single instruction multiple data (SIMD) or pipelined vector execution where long logical vectors execute over multiple cycles, but in the OpenCL case, subvectors can maintain their own program counters until synchronization points. The best example of this is on the GPU, where as many as 64 work items execute in lock step as a single hardware thread on a SIMD unit: On AMD architectures, this is known as a wavefront, and on NVIDIA architectures it is called a warp. The result is SIMD execution via lane wise programming, an arguably simpler development model than explicit use of SIMD instructions as developers are used to when using SSE intrinsics on x86 processors. Because of this SIMD execution, it is often noted that for a given device, an OpenCL dispatch should be an even multiple of that device’s SIMD width. This value can be queried through the `getInfo` functionality of the runtime as the parameter `CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE` to the `clGetKernelWorkGroupInfo` function.

OpenCL defines functions callable from within a kernel to obtain the position of a given work item in the execution range. Some of these functions take a dimension value, listed here as `uint dimension`. This refers to the 0th, 1st, or 2nd dimension in the iteration space as provided in the multidimensional `NDRange` parameters to the kernel enqueue:

- `uint get_work_dim()`: Returns the number of dimensions in use in the dispatch.
- `uint get_global_size(uint dimension)`: Returns the global number of work items in the dimension requested.
- `uint get_global_id(uint dimension)`: Returns the index of the current work item in the global space and in the dimension
requested.
* uint get_local_size(uint dimension): Returns the size of workgroups in this dispatch in the requested dimension.
* uint get_local_id(uint dimension): Returns the index of the current work item as an offset from the beginning of the current workgroup.
* uint get_num_groups(uint dimension): Returns the number of workgroups in the specified dimension of the dispatch. This is \( \frac{\text{get_global_size}}{\text{get_local_size}} \).
* uint get_group_id(uint dimension): Returns the index of the current workgroup. That is, the global index of the first work-item in the workgroup, dividing by the workgroup size.

As an example of execution of a simple kernel, take the following trivial kernel that executes over a two-dimensional execution space, multiplies an input array by 2, and then assigns it to the output. Figure 5.1 shows how this executes in practice. We can see that the calls to get_global_id and get_global_size return different values for each work item that refer to different points in the iteration space. In this trivial example, we use the position in the space to directly map to a two-dimensional data structure. In real examples, much more complicated mappings are possible, depending on the input and output structures and the way an algorithm will process the data.

__kernel void simpleKernel(__global float *a, __global float *b) {
    int address =
        get_global_id(0) +
        get_global_id(1) * get_global_size(0);
    b[address] = a[address] * 2;
}

Figure 5.1 Executing a simple kernel showing kernel instances in the grid.

OpenCL Synchronization: Kernels, Fences, and Barriers

In the OpenCL model at its simplest, individual work items execute independently. A write performed in one work item has no ordering guarantee with a read performed in another work item. Rather, OpenCL has both a relaxed synchronization model and a relaxed memory consistency model. Although the reality of hardware means that certain guarantees will be met, in a cross-platform API no such guarantee can be made. The solution is that OpenCL explicitly defines synchronization points where the programmer knows with certainty what the state of some part of the system is and can rely on that information to obtain expectations of behavior.

Because OpenCL runs on devices in which threading is managed by hardware, such as GPUs, in addition to operating system-managed threading models such as mainstream x86 CPUs, further care is taken to enable full concurrency. In an x86 thread, it is possible to attempt to lower a semaphore and block if the semaphore is unavailable, knowing that the operating system will remove the thread from execution and is free to schedule anything in its place with little in the way of resource constraints. On a GPU, applying the same trick in the GPU equivalent of a thread, the wavefront on AMD hardware, is problematic because the resources occupied are fixed. For example, removing one wavefront from execution does not free its resources, so it is possible to reach a situation in which a wavefront that is not yet able to fit on the device is required to free the semaphore before one that is already on the device is able to continue. Because the wavefronts on the device are waiting on that semaphore, they never get to execute and so the system deadlocks.

To circumvent this eventuality, OpenCL only defines global synchronization at kernel boundaries. That is, between one work item and another, there is no specified method of ensuring an ordering if those two work items are in different workgroups of the same kernel execution. To support sharing of data, mainly in local memory, between work items in the same workgroup, OpenCL specifies the barrier operation within the workgroup. A call to barrier within a work item requires that that work item cannot continue past the barrier until all work items in the group have also reached the barrier. This is a program-counter level restriction, which means that each barrier in the code is treated as a different execution barrier. As a result, when a workgroup barrier is placed within control flow in the kernel, all work items within the group must encounter that barrier. The net effect of this is that behavior of barriers within control flow that diverges between different work items in the group is undefined: On many devices, this leads to deadlock as work items wait for others that will never reach the barrier.

A simple example of OpenCL synchronization is shown in Figure 5.2. In this diagram, we see an initial kernel enqueue with four workgroups of eight work items each. Under the loosest interpretation of the OpenCL spec (i.e., ignoring
hardware implementations), the work items in each workgroup proceed at varying rates. On issuing the barrier instruction, the most advanced work item waits for all others to catch up, only continuing after all have reached that point. Different workgroups and specifically work items in other workgroups proceed with a schedule completely unrelated to that of the first workgroup until the end of the kernel. Between kernel dispatches, all work is guaranteed to be complete and all memory consistent. Then, the next kernel launches, with the same semantics.

Figure 5.2 Synchronization behavior in OpenCL. Within a single kernel dispatch, synchronization is only guaranteed within workgroups using barriers. Global synchronization is maintained by completion of the kernel and the guarantee that on a completion event all work is complete and memory content as expected.

If we assume that the kernels enqueued as 0 and 1 are produced from the same kernel object, the following kernel code and API calls could be expected to produce the behavior seen in Figure 5.2. In this case, the behavior we see from the work items is a simple wrapping neighborwise addition of elements in local memory, where availability of the data must be guaranteed before neighbors can be read. Note from this example that kernel arguments assigned to a kernel object are persistent and hence do not need to be repeatedly set. This is true of both the C and C++ APIs.

```c
// Host code
...
cl_mem input = clCreateBuffer(
    context,
    CL_MEM_READ_ONLY,
    10*sizeof(float),
    0,
    0);
cl_mem intermediate = clCreateBuffer(
    context,
    CL_MEM_READ_ONLY,
    10*sizeof(float),
    0,
    0);
cl_mem output = clCreateBuffer(
    context,
    CL_MEM_WRITE_ONLY,
    10*sizeof(float),
    0,
    0);
clEnqueueWriteBuffer(
    queue,
    input,
    CL_TRUE,
    0,
    10*sizeof(int),
    (void *)&hostInput,
    0,
    NULL,
    NULL);
clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&input);
clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&intermediate);
clSetKernelArg(kernel, 2, 2*sizeof(float), 0);
size_t localws[1] = {2};
size_t globalws[1] = {10};
clEnqueueNDRangeKernel(
    queue,
    kernel,
    1,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
Queuing and Global Synchronization

OpenCL is based on a task-parallel, host-controlled model, in which each task is data parallel. This is maintained through the use of thread-safe command queues attached to each device. Kernels, data movement, and other operations are not simply executed by the user calling a runtime function. These operations are enqueued onto a specific queue using an asynchronous enqueue operation, to be executed at some point in the future.

The commands enqueued into OpenCL’s command queues can be as follows:

- Kernel execution commands
- Memory commands
- Synchronization commands

All kernel execution and synchronization commands are enqueued asynchronously. Completion of a command from the point of view of the host program is only guaranteed at a synchronization point. The following are the primary synchronization points:
• A `clFinish` command that blocks until an entire queue completes execution
• Waiting on the completion of a specific event
• Execution of a blocking memory operation

The last option is the simplest, often used in simple OpenCL demos. The following is a program fragment that asynchronously enqueues a sequence of commands and requires a blocking memory operation to perform synchronization with the host:

```c
// Perform setup of platform, context and create buffers
...

// Create queue leaving parameters as default so queue is in-order
queue = clCreateCommandQueue(context, devices[0], 0, 0);
...
clEnqueueWriteBuffer(
    queue,
    bufferA,
    CL_TRUE,
    0,
    10 * sizeof(int),
    a,
    0,
    NULL,
    NULL);
clEnqueueWriteBuffer(
    queue,
    bufferB,
    CL_TRUE,
    0,
    10 * sizeof(int),
    b,
    0,
    NULL,
    NULL);
// Set kernel arguments
...
clEnqueueNDRangeKernel(
    queue,
    kernel,
    1,
    NULL,
    globalws,
    localws,
    0,
    NULL,
    NULL);
// Perform blocking read-back to synchronize
clEnqueueReadBuffer(
    queue,
    bufferOut,
    CL_TRUE,
    0,
    10 * sizeof(int),
    out,
    0,
    0,
    0);
```
The second parameter to `enqueueReadBuffer` reads `CL_TRUE`. This parameter makes the read buffer synchronous, such that it will block until the data has been copied back. To correctly copy back, all activities in the queue before the copy must have completed to correctly generate the data. Had we set that parameter to `CL_FALSE`, a further synchronization operation would have been needed. The simplest approach would have been to insert a `clFinish` operation on the queue:

```c
clEnqueueReadBuffer(
    queue, 
    bufferOut, 
    CL_FALSE, 
    0, 
    10 * sizeof(int), 
    out, 
    0, 
    0, 
    0);
clFinish(queue);
```

**Memory Consistency in OpenCL**

OpenCL synchronization applies not only to completion of work but also to correct visibility of memory. OpenCL follows a relaxed memory consistency model that allows it to be more efficiently mapped to a wide range of devices. In the OpenCL model, any memory object that is shared between multiple enqueued commands is guaranteed to be consistent only at synchronization points. This means that between two commands, consistency, and hence correctness of communication, is guaranteed at the minimum between elements in an in-order queue or on a communicated event from one command that generates the event to another that waits on it.

Even in this case, memory object consistency will be maintained only within the runtime, not visibly to the host API. To achieve host API correctness, the user must use one of the discussed blocking operations. For example, `clFinish` will block until all operations in the specified queue have completed and hence guarantee memory consistency of any buffers used by operations in the queue.

Even in this case, memory object consistency will be maintained only within the runtime, not visibly to the host API. To achieve host API correctness, the user must use one of the discussed blocking operations. For example, `clFinish` will block until all operations in the specified queue have completed and hence guarantee memory consistency of any buffers used by operations in the queue.

Between devices, the same consistency issues arise. Because memory objects are associated with contexts rather than devices, it is the responsibility of the OpenCL runtime to ensure that such objects are consistent across devices when data is shared and appropriate events occur. Data is moved from one device to another such that if a kernel is to be executed on a second device, any results generated on the first will be available when necessary. The completion of an event on the first data structure is the guarantee that the data is OK to move and no separate buffer copy operation is needed.

**Events**

Note that the command queue is constructed ignoring the final two parameters, which are left as default. One of the properties available for this bit field is to enable out-of-order execution of the queue (`CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE`). Thus, although the queue in the previous example is in-order, it is possible for a queue to execute out-of-order.

An out-of-order queue has no default ordering of the operations defined in the queue. If the runtime decides that it has, for example, a DMA engine that can execute in parallel with compute units, or that the device can execute multiple kernels at once, it is at liberty to schedule those operations in parallel with no guarantee that one completes before another starts. Similarly, if the runtime has multiple queues whether on the same device or, more obviously, on multiple devices, there is no default assumption about order of execution of elements of these multiple queues.

In either case, to correctly execute such a structure requires the construction of a task graph. In OpenCL, task graph creation is through event objects. OpenCL’s event model allows the construction of complicated graphs linking the tasks enqueued in any of the command queues associated with a given OpenCL context. A single event can be passed as the final parameter to the enqueue functions, and this event encapsulates the state of that enqueued command. Most important, the event registers the completion of the task along with the guarantee that all memory referenced by the task is consistent. A list of events can be passed to an enqueue function as a dependence list. This means that the command will not begin executing until all of the input events have completed. The following code is a repeat of the previous example, with an out-of-order queue and all dependencies explicitly defined. Figure 5.3 represents the same command sequence diagrammatically. As one can see in the example, a third approach for synchronizing with the host is to use an event directly. In this case, we see that we have called `wait()` on the read event:

```c
// Perform setup of platform, context and create buffers
...
// Create queue leaving parameters as default so queue is in-order
queue = clCreateCommandQueue(context, devices[0], 0, 0);
```
cl_event writeEventA;
cl_event writeEventB;
cl_event kernelEvent;
cl_event readEvent;

clEnqueueWriteBuffer(
    queue,
    bufferA,
    CL_TRUE,
    0,
    10 * sizeof(int),
    a,
    0,
    NULL,
    &writeEventA);

clEnqueueWriteBuffer(
    queue,
    bufferB,
    CL_TRUE,
    0,
    10 * sizeof(int),
    b,
    0,
    NULL,
    &writeEventB);

// Set kernel arguments
... size_t localws[1] = {2}; size_t globalws[1] = {10};
// Wait on both writes before executing the kernel
cl_event eventList[2];
eventList[0] = writeEventA;
eventList[1] = writeEventB;

clEnqueueNDRangeKernel(
    queue,
    kernel,
    1,
    NULL,
    globalws,
    localws,
    2,
    eventList,
    &kernelEvent);

// Decrease reference count on events
clReleaseEvent(writeEventA);
clReleaseEvent(writeEventB);

// Read will wait on kernel completion to run
clEnqueueReadBuffer(
    queue,
    bufferOut,
    CL_TRUE,
    0,
    10 * sizeof(int),
    out,
    1,
    &kernelEvent,
    &readEvent);
clReleaseEvent(kernelEvent);

// Block until the read has completed
clWaitForEvents(1, &readEvent);
clReleaseEvent(readEvent);

Figure 5.3 One enqueued command can depend on a set of enqueued commands through the events passed to the respective enqueue functions.

Command Queues to Multiple Devices

Understanding the synchronization capabilities and the host memory model of OpenCL is necessary for the management of multiple command queues. Multiple queues can be mapped to the same device to overlap execution of different commands or overlap commands and host–device communication. If we have multiple devices in a system (e.g., a CPU and a GPU or multiple GPUs), each device needs its own command queue.

Figure 5.4 shows an OpenCL context with two devices. Separate command queues are created to access each device. The following code shows how two command queues can be created within the same context. It is important to note that synchronization using OpenCL events can only be done for commands within the same context. If separate contexts were created for different devices, then synchronization using events would not be possible, and the only way to share data between devices would be to use clFinish and then explicitly copy data between buffer objects.

```c
cl_uint num_devices;
cl_device_id devices[2];
cl_context ctx;

//Obtain devices of both CPU and GPU types
err_code = clGetDeviceIDs(  
    NULL,  
    CL_DEVICE_TYPE_CPU,  
    1,  
    &devices[0],  
    &num_devices);
err_code = clGetDeviceIDs(  
    NULL,  
    CL_DEVICE_TYPE_GPU,  
    1,  
    &devices[1],  
    &num_devices);

//Create a context including two devices
ctx = clCreateContext(0, 2, devices, NULL, NULL, &err);

cl_command_queue queue_cpu, queue_gpu;
//Create queues to each device
queue_cpu = clCreateCommandQueue(context, devices[0], 0, &err);
queue_gpu = clCreateCommandQueue(context, devices[1], 0, &err);
```

Figure 5.4 Multiple command queues created for different devices declared within the same context.

Multiple device programming with OpenCL can be summarized with two execution models usually seen in parallel programming for heterogeneous devices:

• Two or more devices work in a pipeline manner such that one device waits on the results of another, shown in Figure 5.5.

Figure 5.5 Multiple devices working in a cooperative manner on the same data. The CPU queue will wait until the GPU kernel is finished.
A model in which multiple devices work independently of each other, shown in Figure 5.6.

Figure 5.6 Multiple devices working in a parallel manner. In this scenario, both GPUs do not use the same buffers and will execute independently. The CPU queue will wait until both GPU devices are finished.

In the following code, the wait list orders execution such that the kernel on the GPU queue will complete before the CPU queue begins executing the kernel:

```c
// A collaborative - pipelined model of multidevice execution
// The enqueued kernel on the GPU command queue waits for the kernel on the CPU
// command queue to finish executing
cl_event event0_cpu, event1_gpu;
// Starts as soon as enqueued
err = clEnqueueNDRangeKernel(
    queue_gpu,
    kernel1_gpu,
    2,
    NULL,
    global,
    local,
    0,
    NULL,
    &event_gpu);
// Starts after event_gpu is on CL_COMPLETE
err = clEnqueueNDRangeKernel(
    queue_cpu,
    kernel2_cpu,
    2,
    NULL,
    global,
    local,
    1,
    &event_gpu,
    &event_cpu);
clFlush(queue_cpu);
clFlush(queue_gpu);
```

The following code shows an execution model in which the kernels are executed on different devices in parallel. A parallel multidevice example is shown in Figure 5.6, in which two GPUs process kernels independently. Separate buffers are required for the two devices because they can only execute in parallel if they do not share buffers.

```c
// Parallel multidevice execution
// We would need to create 3 command queues in this case
// 2 queues for 2 GPUs and 1 queue for the CPU
// The enqueued kernel on the CPU command queue waits
// for the kernels on the GPU command queues to finish
cl_event event_gpu[2];
// Both the GPU devices can execute concurrently as soon as they have
// their respective data since they have no events in their waitlist
err = clEnqueueNDRangeKernel(
    queue_gpu_0,
    kernel_gpu,
    2,
    NULL,
    global,
    ...)
```
local,
0,
NULL,
&event_gpu[0];
err = clEnqueueNDRangeKernel(
    queue_gpu_1,
    kernel_gpu,
    2,
    NULL,
    global,
    local,
    0,
    NULL,
    &event_gpu[1];
// The CPU will wait till both GPUs are done executing their kernels
// Two events in the CPU's waitlist
err = clEnqueueNDRangeKernel(
    queue_cpu,
    kernel_cpu,
    2,
    NULL,
    global,
    local,
    2,
    event_gpu,
    NULL);
clFlush(queue_gpu_0);
clFlush(queue_gpu_1);
clFlush(queue_cpu);

Event Uses beyond Synchronization

Due to the asynchronous nature of the OpenCL API, there is no good way for individual API calls to return error conditions or profiling data that relates to the execution of the OpenCL command rather than the setup of the queue performed by the enqueue function. Whereas the API calls can return error conditions relating to their parameters, error conditions relating to the execution of the OpenCL command itself can be queried through the event the command returns. Indeed, completion can be considered a condition similar to any other.

Event queries are performed through the `getInfo` function on an event. The following concepts can be queried through `getInfo`:

- **CL_EVENT_COMMAND_QUEUE**: Returns the command queue associated with the event (useful if events are being passed around a complicated program).
- **CL_EVENT_CONTEXT**: Returns the context associated with the event.
- **CL_EVENT_COMMAND_TYPE**: Returns the command associated with the event. This can be one of a list of types, including `CL_COMMAND_NDRANGE_KERNEL` and `CL_COMMAND_READ_BUFFER`.
- **CL_EVENT_COMMAND_EXECUTION_STATUS**: Returns the status of the command associated with the event. `CL_COMPLETE` is the event we wait on with `event.wait()`, but the command can be queued, submitted, or running as well. A negative integer value in this field is the method by which error codes that only arise when the command tries to execute are returned.

If the context was created with profiling enabled, `event.getProfilingInfo` allows the developer to obtain profiling information from the command. Profiling with events is discussed in Chapter 13.

User Events

User events are OpenCL’s method for allowing the user to enqueue commands that depend on the completion of some arbitrary task. The user event can be passed to OpenCL enqueue functions like any other event, but the execution status of the event is set explicitly.

For example, to ensure that a buffer is not overwritten by an asynchronous OpenCL read operation until the buffer is no longer in use, we could do something such as the following:
Event Callbacks

OpenCL allows a user to define callbacks invoked when events reach specific states. The callback function will be invoked for a specified execution status of a command in the queue. Event callbacks can be used to enqueue new commands. Callbacks can also be used to invoke host functions such as specialized CPU libraries. The `clSetEventCallback` function call is used to set a callback for an event:

```c
// Function call to set an event callback
cl_int clSetEventCallback (  
  //OpenCL event
  cl_event event,  
  //Event Status which invokes callback  
  cl_int command_exec_callback_type,  
  //Function pointer - parameter type shown  
  void (CL_CALLBACK *pfn_event_notify)(cl_event event, cl_int event_command_exec_status, void *user_data),  
  //Pointer to user data which is used by callback  
  void *user_data)
```

A usage scenario of OpenCL callbacks (Figure 5.7) includes applications in which the host CPU interacts tightly with a device such as a GPU. In such applications, usually the host would have to wait while the device is executing. This could reduce the system's efficiency. An alternative method would be to set a callback to a stub host function. The host could improve its efficiency by doing other work instead of spinning while waiting on the GPU.

![Figure 5.7 Using callbacks to enqueue data to device.](image)

The location of the `clSetEventCallback` is key in the following code segment. The function call is required after the `clEnqueueNDRangeKernel` because the `clSetEventCallback` function requires a valid event object that gets populated by the runtime.

The following code will return with an invalid event error code because the runtime has not populated an event with valid information when the callback is being set up:

```c
cl_event completionEvent;
// Wrong location to set callback in
errcode = clSetEventCallback(
  completionEvent,
```
Callbacks should be used with caution for the following reasons, as highlighted by the OpenCL specification:

- There is no guarantee that the callback functions registered for multiple execution status for the same event will be called in the exact order that the execution status of a command changes.
- The callback should be thread-safe and will be called asynchronously.
- The behavior of calling expensive system routines, or blocking OpenCL API calls such as `clFinish` from the callback function is undefined.

For these reasons, callbacks should be lightweight stubs, which call more complicated functions. The following is an example of setting a callback:

```c
// The callback can only get a single void* user_data pointer.
// As a workaround, a programmer can pass multiple
// arguments by wrapping them within a structure as shown
struct arg_block{
    data_type arg0;
    data_type arg1;
};
cl_event completionEvent;

// Simple example showing declaration of a callback
void CL_CALLBACK callbackFunction(
    cl_event event,
    cl_int cmd_exec_status,
    void *user_data) {
    // Use this function to invoke a host library
    arg_block *ipargs = (arg_block *)user_data;
    // Call host function
    host_function(ipargs->arg0, ipargs->arg1);
}

// Start Device computation
errcode = clEnqueueNDRangeKernel(
    command_queue,
    kernel,
    2,
    0,
    globalworksize,
    localworksize,
    NULL,
    &completionEvent);
```

// Set the callback such that callbackFunction is called when
// completionEvent indicates that the kernel
// has completed (CL_COMPLETE)
errcode = clSetEventCallback(
    completionEvent, callbackFunction, NULL);
One of the primary benefits of using the event-handling capabilities of OpenCL is that application-level behavior and synchronization requirements can be handled in a consistent manner on both CPU and GPU for multiple vendor implementations. This restricts device-specific tuning to only the compute kernels.

Native Kernels

An alternative to callbacks that is more cleanly integrated into the OpenCL execution model is to use native kernels. Native kernels allow standard C functions compiled with a traditional compiler rather than the OpenCL compiler to be executed within the OpenCL task graph, be triggered by events, and trigger further events.

The difference between enqueuing a native kernel and enqueuing a kernel is that rather than taking a cl_kernel object as an argument, the native kernel enqueue function, clEnqueueNativeKernel, takes a function pointer to a standard C function. The argument list is provided separately along with its size. Because OpenCL uses buffer and image objects, these are passed as kernel arguments, and it is useful to be able to pass these to native functions. This process is called unboxing, and it is handled by passing in a list of memory objects, in the argument mem_list, and a list of pointers, args_mem_loc, mapping into args where the unboxed memory pointers will be placed.

To illustrate the point, consider the following example, in which a native function expects an argument list containing five values, where the 0 and 2 indexes are set to integers 5 and 8, respectively, and the 1, 3, and 4 indexes are two buffer objects and an image object. This is shown in Figure 5.8. The corresponding code is as follows:

```c
cl_command_queue queue = clCreateCommandQueue(…);
cl_mem buffer1      = clCreateBuffer(…);
cl_mem buffer2      = clCreateBuffer(…);
cl_mem image       = clCreateImage2D(…);
// initialize buffers, images, and so on
size_t cb_args = 5;
num_mem_objects = 3;
void *args[5] = {(void *)5, NULL, (void *)8, NULL, NULL};
cl_mem mem_list[3] = {buffer1, buffer2, image};
void * args_mem_loc[3] = {&args[1], &args[3], &args[4]};
```

![Figure 5.8](image)

Finally, given a native function void foo(void * args), we can call clEnqueueNativeKernel:

```c
clEnqueueNativeKernel(
    queue,
    foo,
    args,
    cb_args,
    num_mem_objects,
    mem_list,
    args_mem_loc,
    0,
    NULL,
    NULL);
```

Command Barriers and Markers

An alternative method of synchronizing in an out-of-order queue is similar to the approach for synchronizing within a workgroup. In both cases, a barrier operation causes anything executing from the queue before the barrier to complete until activities after the queue can continue. A barrier includes no state and does not support an event of its own but sits in a queue guaranteeing ordering. A barrier is conceptually similar to calling clWaitForEvents from the host, but is managed internally to the runtime. Barriers are enqueued using the clEnqueueBarrierWithWaitList command, that takes a list of events to
wait on. If no events are provided the barrier waits for completion of all preceding commands in the command queue.

Markers are similar to barriers and are enqueued with the matching clEnqueueMarkerWithWaitList command. The difference between a barrier and a marker is that the marker does not block the execution of subsequent commands in the queue but simply triggers an event when its dependencies are satisfied.

Between these synchronization commands and the more general use of events, OpenCL provides the ability to produce sophisticated and complicated task graphs enabling highly complicated behaviors.

The Host-Side Memory Model

OpenCL devices such as GPUs and other accelerators frequently operate with memory systems separate from the main memory associated with the computer's primary CPUs. In addition, OpenCL's concurrency model supports a relaxed consistency in which global synchronization of memory is only defined on the completion of events and local synchronization on barrier operations. To support both of these features, OpenCL's memory objects are defined to be in a separate space from the host CPU's memory. Any movement of data in and out of OpenCL memory objects from a CPU pointer must be performed through API functions. It is important to note that OpenCL's memory objects are defined on a context and not on a device. That is, in general, moving data in and out of a buffer need not move data to any specific device. It is the job of the runtime to ensure that data is in the correct place at the correct time.

OpenCL's memory objects are divided into two types, where specific placement, layout, and format of these two types are defined by parameters. The two types of objects defined in the OpenCL specification are buffers and images.

Buffers

Buffer objects map very easily to the standard array representation that people expect in the host C program. Consider the following host code, which is legal C:

```c
float a[10], b[10];
for(int i = 0; i < 10; ++i){
    *(a+i) = b[i];
}
```

The example shows that we can access a and b either through pointers or using array access syntax. This is important because it implies that data is allocated sequentially, such that the i th element a[i] of array a is stored at location (a + i).

We can use sizeof operations on array elements to calculate offsets into arrays cast to pointers of different types. In low-level code, it is useful to have these features, and it is a natural expectation for a C-derived language. For example, the following OpenCL kernel code, taken from the Bullet physics SDK, allows us to perform flexible output into a vertex buffer by parameterizing with base pointer and strides. The position and normal arrays we receive are float4 in structure, and the output is a structure containing the position and normal information as well as other content that the kernel need not know about.

```c
__kernel void OutputToVertexArray(
    const int startNode,
    const int numNodes,
    __global float *g_vertexBuffer,
    const int positionOffset,
    const int positionStride,
)
```
Manipulating Buffer Objects

Buffer objects are similar to malloc’d arrays, so their creation is relatively simple. At the simplest level, creation requires a size, a context in which to create the buffer, and a set of creation flags:

```
cl_mem clCreateBuffer(
    cl_context context,
    cl_mem_flags flags,
    size_t size,
    void *host_ptr,
    cl_int *err)
```

The function returns a buffer object, where the error code is returned through a variable passed by reference as the last parameter. The flags allow for various combinations of read-only/write-only data and allocation options. For example, in the following code, we create a read-only buffer that will be stored directly in a source array `a`, which is of the same size as the buffer. Note that memory in OpenCL is only guaranteed to be consistent at completion events of enqueued operations. As a result, when `CL_MEM_USE_HOST_PTR` is used, the runtime is still able to copy the data to the device, execute, and return it on completion because the data is guaranteed to have been synchronized after the kernel completion event. Any error value will be returned in `err`, which can be any of a range of error conditions defined in the specification. `CL_SUCCESS` is returned by any of the OpenCL functions when they complete successfully.

```c
cl_int err;
int a[16];
cl_mem newBuffer = clCreateBuffer(
    context,
    CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
    16*sizeof(int),
    a,
    &err);
if(err != CL_SUCCESS){
    // Do whatever error test is necessary
}
```

After creation, access to buffer objects is achieved through access functions. These functions are intended, like the rest of the OpenCL API, to be used asynchronously. That is, if we call `clEnqueueReadBuffer`, we cannot expect to be able to read the data from the target host array until we know—through the event mechanism, a `clFinish` call, or by passing `CL_TRUE` to `clEnqueueReadBuffer` to make it a blocking call—that the read has completed. Thus, for example, the following host code sequence does not guarantee that the two `printf` calls A and B generate different values even if `outputBuffer`’s content would suggest that it should. The `printf` of C is the only point in the code where the printed value is guaranteed to be that copied from `outputBuffer`.

```c
int returnedArray[16];
```
cl_buffer outputBuffer;
cl_event readEvent;

// Some code that fills returned Array with 0s and invokes kernels
// that generates a result in outputBuffer
printf("A: %d\n", returnedArray[3]);
clEnqueueReadBuffer(
    commandQueue,
    outputBuffer,
    CL_FALSE,
    0,
    sizeof(int)*16,
    returnedArray,
    0,
    0,
    &readEvent);
printf("B: %d\n", returnedArray[3]);
clWaitForEvents(1, &readEvent);
printf("C: %d\n", returnedArray[3]);

This is a vital point about the OpenCL memory model. Changes to memory are not guaranteed to be visible, and hence memory is not guaranteed to be consistent, until an event reports that the execution has completed. This works both ways: In a transfer between a host buffer and a device buffer, you cannot reuse a host buffer until you know that the event associated with the asynchronous copy moving data into the device buffer has completed. Indeed, a careful reading of the OpenCL specification suggests that because buffers are associated with the context and not a device, a clEnqueueWriteBuffer enqueue, even on completion, does not guarantee to have moved the data to the device, only that it be moved out of the host pointer:

If blocking_write is CL_TRUE, the OpenCL implementation copies the data referred to by ptr and enqueues the write operation in the command-queue. The memory pointed to by ptr can be reused by the application after the clEnqueueWriteBuffer call returns.

OpenCL 1.2 specification section 5.2.2

However, unlike other API calls in OpenCL, the read and write buffer calls allow us to specify synchronous execution. Had we replaced the previous call with

cEnqueueReadBuffer(
    commandQueue,
    outputBuffer,
    CL_TRUE,
    0,
    sizeof(int)*16,
    returnedArray,
    0,
    0,
    &readEvent);

execution of the host thread would stall at the read buffer call until all execution had completed and the copy had been correctly performed.

OpenCL also supports of sub-buffer objects that allow us to divide a single buffer into multiple smaller buffers that may overlap and that can be read, written, copied, and used in much the same way as their parent buffer objects. Note that overlapping sub-buffers and the combination of sub-buffers and their parent buffer objects constitutes aliasing, and behavior is undefined in these circumstances.

Images

Images differ from buffers in three ways. Images are

• opaque types that cannot be viewed directly through pointers in device code;
• multidimensional structures; and
• limited to a range of types relevant to graphics data rather than being free to implement arbitrary structures.

Image objects exist in OpenCL to offer access to special function hardware on graphics processors that is designed to support highly efficient access to image data. These special function units do not always support the full range of access...
modes necessary to enable buffer access, but they may provide additional features such as filtering in hardware in a highly efficient manner. Filtering operations enable efficient transformations of image data based on collections of pixels. These operations would require long instruction sequences with multiple read operations and can be very efficiently performed in dedicated hardware units.

Image data is accessed through specialized access functions in the kernel code, as discussed later. Access to images from the host is not significantly different from access to buffers, except that all functions are expanded to support addressing in multiple dimensions. Thus, for example, `clEnqueueReadImage` is more like `clEnqueueReadBufferRect` than `clEnqueueReadBuffer`.

The major difference between buffers and images from the host is in the formats images can support. Whereas buffers support the basic OpenCL types and structures made from them, image formats are more subtle.

Image formats are a combination of a channel order and a channel type. Channel order defines the number of channels and the order in which they occur—for example, `CL_RGBA`, `CL_R`, or `CL_ARGB`. Channel type is selected from a wide range of storage formats from `CL_FLOAT` to less storage-hungry formats such as `CL_UNORM_SHORT_565`, which packs into a single 16-bit word in memory. When accessed from kernel code, reading from any of these formats results in upconversion to a standard OpenCL C type. The list of image formats can be queried by the API call `clGetSupportedImageFormats`.

Images offer an additional feature that enables optimizations in the runtime system and hardware that buffers may often not support. Whereas image data can be mapped to the host using the `clEnqueueMapImage` API call, and hence image data must have a certain format when viewed through the mapped host pointer, the semantics of the map operation allow for format conversion. This feature of OpenCL data structures enables the runtime system to perform transformations on data that it controls.

Image objects cannot be accessed through pointers on the device and cannot be both read and write within the same kernel. As a result, the transformations that the runtime system performs can be entirely opaque to the kernels executing on the OpenCL device: Transformations can significantly improve performance without affecting code correctness. This feature of images also removes the possibility of aliased data, allowing the hardware to cache images in situations in which buffers cannot be safely cached.

Take one common optimization as an example. Any given multidimensional data structure, of which an image is an example, must be mapped to a single dimensional memory address at some point. The obvious method, and indeed the method applied to multidimensional arrays in most programming languages, is a dictionary order in either column-major or row-major pattern. That is, \((x,y)\) comes before \((x+1,y)\), which comes long before \((x,y+1)\), and so on. The long distance in memory between \((x,y)\) and \((x+1,y)\) means that an access of consecutive addresses in the y-dimension stride ineﬃciently through memory hitting a large number of cache lines. In contrast, the fact that \((x,y)\) is adjacent to \((x+1,y)\) means consecutive accesses in \(x\) stride efﬁciently (and cause memory accesses to coalesce).

Z-order or Morton order memory layouts apply a mapping that preserves spatial locality of data points. Figure 5.9 shows that the data is stored in order \((0,0)\), \((1,0)\), \((0,1)\), \((1,1)\), \((2,0)\) and so on. By storing data according to its position in a Z-ordered mapping, we may hit the same cache line repeatedly when performing a vertical read. If we go further by laying out our computational work in a two-dimensional layout (as we see with the quads created in the graphics pipeline), we further improve this data locality. This sort of optimization is only possible transparently (and hence different optimizations can be performed on different architectures) if we offer the kernel programmer no guarantees about the relative locations of memory elements.

![Figure 5.9](image-url) Applying the Z-order mapping to a two-dimensional memory space.

We can go a step further with this approach. If we are executing on an architecture that does not have vector registers and does not perform vector reads from memory, we might wish `float4 a = read_imagef(sourceImage, imageSampler, location)` to compile down to four scalar reads instead of a single vector read. In these circumstances, it might be a more efficient use of the memory system to read from the same offset into four separate arrays instead of four times from the single array because the data in each separate array would exhibit better locality on each individual read operation.

### The Device-Side Memory Model

On OpenCL devices, the memory space is classified into four primary categories:

- **Global memory**
- **Local memory**
- **Constant memory**
- **Private memory**

These memory spaces are visualized in Figure 5.10. As discussed in Chapter 3, OpenCL is designed to run on a range of architectures. The purpose of arranging a memory hierarchy of this form is to allow OpenCL to perform efficiently on
such architectures. The actual meaning of each memory space in terms of a hardware mapping is very much implementation dependent. However they are mapped to hardware, as a programming construct, these memory spaces are disjoint. Furthermore, as shown in Figure 5.10, local and private are divided into disjoint blocks across compute units and work items. By defining separate layers of address space in this way, the mapping to hardware can efficiently use anything from relaxed memory consistency models with programatically controlled scratchpad buffers as seen on most GPU devices to fully coherent memory systems such as x86-based architectures.

![Figure 5.10](image)
The OpenCL memory spaces available on an OpenCL device.

The default address space for function arguments and local variables within a function or block is private. Pointer arguments to functions can be placed in one of the other address spaces depending on where the data comes from or where it is to be used. Note that the pointer itself is always in the private address space wherever the data lies.

The address spaces are strictly disjoint when used through pointers. Casting from one address space to another is not legal because this would imply either that the data lives at a globally accessible address or that the compiler would have to generate a copy to go with the cast, which is not feasible in practice. Image arguments to functions always live in the global address space, so we discuss images in those terms.

### Device-Side Relaxed Consistency

OpenCL’s relaxed consistency model applies within the kernel as well as between dispatches. Writes to memory are not guaranteed to be visible until the end of the kernel execution unless fence operations are used. As a result, we have a hierarchy of consistency:

- **Within a work item**, memory operations are ordered predictably: Any two reads and writes to the same address will not be reordered by hardware or the compiler.
- **Between work items and within a workgroup**, memory is only guaranteed to be consistent at a barrier operation.
- **Between workgroups**, there are no guarantees about memory consistency until completion of the kernel execution—that is, when the event reports completion.

Given the previous hierarchy, there is no requirement for the compiler to make anything but the last write to a given address visible outside a given work item. To allow some level of communication between work items within and between workgroups, OpenCL provides a set of fence operations. Even with these fences, there are no guarantees of ordering between work items.

Fences come in read, write, and read/write versions:

- `read_mem_fence(cl_mem_fence_flags flags)`
- `write_mem_fence(cl_mem_fence_flags flags)`
- `mem_fence(cl_mem_fence_flags flags)`

In each case, the fence is parameterized with flags specifying the address space it is fencing. The value of these flags is some combination of `CLK_LOCAL_MEM_FENCE` and `CLK_GLOBAL_MEM_FENCE`. The fence ensures that loads and/or stores issued before the fence will complete before any loads and/or stores issued after the fence. No synchronization is implied by the fences alone. The `barrier` operation supports a read/write fence in one or both memory spaces as well as blocking until all work items in a given workgroup reach it.

An alternative approach to ensuring that memory operations are correctly communicated between work items is to use atomic operations. These are particularly useful because they guarantee not only that a write update occurs but also that a read and write combined with some operation on the data occur without interruption from another work item. However, they are only defined on integer data due to the complexity (both in implementation and in comprehension) of performing floating point operations in most hardware. Atomic operations may be arithmetic operations, such as `int atomic_add(volatile __global int *p, int val)`, and data-only, such as `int atomic_xchg (volatile __global int *p, int val)`. In all cases, the atomic operation returns the original data that was at the memory location. Note that if the return value is ignored, the compiler is at liberty to use nonreturning atomic operations, which are far more efficient on many architectures.

### Global Memory

Global memory, defined in OpenCL C code by a pointer with the type qualifier `__global` (or `global`), or by one of the image types `image2d_t` or `image3d_t`, refers to data in a memory space consistently addressable by all compute units in the device. The two types of object differ in their scope and use cases.

The `__global` address space qualifier refers to a pointer referencing data in a buffer object. As noted previously, a buffer can carry any scalar data type, vector data type, or user-defined structure. Whatever the type of buffer, it is accessed at
typedef struct AStructure {
    float a;
    float b;
} AStructure;

__kernel void aFunction(__global AStructure *inputOutputBuffer) {
    __global AStructure* inputLocation =
        inputOutputBuffer + get_global_id(0);
    __global AStructure* outputLocation =
        inputOutputBuffer + get_global_size(0) + get_global_id(0);
    outputLocation->a = inputLocation->a * -1;
    outputLocation->b = (*inputLocation).b + 3.f;
}

Image objects, although conceptually in the __global memory space, are treated differently from buffers and are not mappable to __global pointers. Image objects can be one-dimensional, two-dimensional or three-dimensional and created using the image1d_t, image2d_t or image3d_t type qualifiers. Unlike buffers, images can be either read-only or write-only but never both within the same kernel. This is a result of the design of GPU hardware supporting very high-performance caching and filtering. Within kernel code, we specify which form of access we are using with the __read_only and __write_only access qualifiers on kernel image parameters.

Images are opaque memory objects. Although we can read or write the data based on addresses, we do not really know the relative memory locations of two different values in the image. As a result, and to support parameterization of the style of read, rather than accessing images through pointers, we use a set of built-in functions: read_imagef, read_imagei, read_imageui, write_imagef, and so on. Each of the image read functions takes three parameters:

float4 read_imagef(image2d_t image, sampler_t sampler, float2 coord)

The final address parameter can optionally be an int2 (or int4 if the image is of type image3d_t), and the precise meaning of the returned data depends on the image format. The OpenCL specification lists these options in full.

The first and third parameters to the read functions are self-explanatory, being the image object itself and the coordinate of the read. The second parameter is more complicated. This is a sampler object that defines how the image is interpreted by the hardware or runtime system. The sampler can be defined either by declaring a constant variable of sampler_t type within the OpenCL C source or by passing as a kernel parameter a sampler created in host code using the clCreateSampler function. The following is an example of the use of a constant-variable-declared sampler:

__constant sampler_t sampler =
    CLK_NORMALIZED_COORDS_TRUE | CLK_FILTER_LINEAR;
__kernel void samplerUser(image2d_t sourceImage, __global float *outputBuffer) {
    float4 a = read_imagef(sourceImage, sampler, (float2)(
        (float)(get_global_id(0)),
        (float)(get_global_id(1))));
    outputBuffer[get_global_id(1) * get_global_size(0) +
        get_global_id(0)] = a.x + a.y + a.z + a.w;
}

The value returned in the float4 vector depends on the image format specified on image creation. A CL_R image, for example, would only contain data in the x channel with 1.0 in the w (alpha) channel.

The write functions take a similar set of parameters, replacing the sampler with the value to write:
Local Memory

A subset of the architectures supported by OpenCL, including many of the GPUs and the Cell broadband engine, possess small scratchpad memory buffers distinct from the primary DRAM and caching infrastructure. Local memory in these cases is disjoint from global memory and often accessed using separate memory operations. As a result, data must be copied in and out of it programmatically. Depending on the architecture, this occurs either through DMA transfers (most efficiently accessed using the async_work_group_copy function) or by memory-to-memory copies. Local memory is also supported in CPU implementations, but it sits in standard cacheable memory; in such cases, use of local memory can still be beneficial because it encourages cache-aware programming.

Local memory is most useful because it provides the most efficient method of communication between work items in a workgroup. Any allocated local memory buffer can be accessed at any location by an entire workgroup and hence writes to the local array will be visible to other work items. Remember that OpenCL work items are conceptually, if not literally, executed independently.

Local memory is defined by the __local address space qualifier and can be defined either locally in the kernel or as a parameter. Both examples are shown in the following code:

```c
__kernel void localAccess(
    __global float* A,
    __global float* B,
    __local float* C)
{
    __local float aLocalArray[1];
    if(get_local_id(0) == 0){
        aLocalArray[0] = A[0];
    }
    C[get_local_id(0)] = A[get_global_id(0)];
    barrier(CLK_LOCAL_MEM_FENCE);
    float neighborSum = C[get_local_id(0)] + aLocalArray[0];
    if(get_local_id(0) > 0)
        neighborSum = neighborSum + C[get_local_id(0)-1];
    B[get_global_id(0)] = neighborSum;
}
```

Figure 5.11 shows a diagrammatic representation of the data flow in the previous code sample. Note that data will be read from global memory and written to the two local arrays C and aLocalArray at unpredictable times as the work items execute independently in an undefined order. The reality will be slightly more predictable on a given device because implementations will map to hardware in predictable ways. For example, on the AMD GPUs, execution occurs in lock-step over a wide SIMD vector, meaning that the read and write operations will have an ordering guarantee over the entire vector in the same way that they would over a single work item. However, this feature does not apply generally. In the general case, we must insert the barrier operation: Only at this barrier can we guarantee that all writes to local arrays, and the global memory reads that fed them, will have been completed across the workgroup such that the data is visible to all work items. Beyond this barrier, the data can be used by the entire workgroup as shown in the lower part of the diagram.

![Figure 5.11](image)

The pattern of data flow for the example shown in the “localAccess” code.

aLocalArray is at function scope lexically but is visible to the entire workgroup. That is, there is only one 32-bit variable in local memory per workgroup, and any work item in the group using the name aLocalArray has access to the same 32-bit value. In this case, after the barrier we know that work item 0 has written to aLocalArray and hence all work items in the group can now read from it.

The alternative method for creating local arrays is through a kernel parameter, as we see for array C. This version is created by a runtime API call. To allocate the memory, we call clSetKernelArg as we would for passing a global array to the kernel, but we leave the final pointer field as 0. We therefore allocate a per-workgroup amount of memory based on the third parameter but with no global object to back it up so it sits in local memory:

```c
clEnqueueWriteImage(queues[queue], image2d_t image, float2 coord, float4 color)
```