OpenCL Buffers and Complete Examples

A Collaboration Between
David Kaeli, Northeastern University
Benedict R. Gaster, AMD

Instructor Notes

- This is a brief lecture which goes into some more details on OpenCL memory objects
  - Describes various flags that can be used to change how data is handled between host and device, like page-locked I/O and so on
- The aim of this lecture is to cover required OpenCL host code for buffer management and provide simple examples
- Code for context and buffer management discussed in examples in this lecture serves as templates for more complicated kernels
  - This allows the next 3 lectures to be focused solely on kernel optimizations like blocking, thread grouping and so on
- Examples covered
  - Simple image rotation example
  - Simple non-blocking matrix-matrix multiplication

Topics

- Using OpenCL buffers
  - Declaring buffers
  - Enqueue reading and writing of buffers
- Simple but complete examples
  - Image Rotation
  - Non-blocking Matrix Multiplication

Creating OpenCL Buffers

- Data used by OpenCL devices is stored in a “buffer” on the device
- An OpenCL buffer object is created using the following function

```
cl_mem  bufferobj = clCreateBuffer (
                  cl_context context,        //Context name
                  cl_mem_flags flags,        //Memory flags
                  size_t size,               //Memory size allocated in buffer
                  void *host_ptr,            //Host data
                  cl_int *errcode)           //Returned error code
```

- Data can implicitly be copied to the device using a host pointer parameter
  - In this case copy to device is invoked when kernel is enqueued

Memory Flags

- Memory flag field in clCreateBuffer() allows us to define characteristics of the buffer object

Memory Flag	Behavior
CL_MEM_READ_WRITE	Specifies memory read / write behavior
CL_MEM_WRITE_ONLY	
CL_MEM_READ_ONLY	
CL_MEM_USE_HOST_PTR	Implementations can cache the contents pointed to by host_ptr in device memory. This cached copy can be used when kernels are executed on a device.
CL_MEM_ALLOC_HOST_PTR	Specifies to the implementation to allocate memory from host accessible memory.
CL_MEM_COPY_HOST_PTR	Specifies to allocate memory for the object and copy the data from memory referenced by host_ptr.

Copying Buffers to Device

- clEnqueueWriteBuffer() is used to write a buffer object to device memory (from the host)
- Provides more control over copy process than using host pointer functionality of clCreateBuffer()
  - Allows waiting for events and blocking

```
cl_int clEnqueueWriteBuffer (
        cl_command_queue queue,                 //Command queue to device
        cl_mem buffer,                          //OpenCL Buffer Object
        cl_bool blocking_read,                  //Blocking/Non-Blocking Flag
        size_t offset,                          //Offset into buffer to write to
        size_t cb,                              //Size of data
        void *ptr,                              //Host pointer
        cl_uint num_in_wait_list,               //Number of events in wait list
        const cl_event *  event_wait_list,      //Array of events to wait for
        cl_event *event)                        //Event handler for this function
```

Copying Buffers to Host

- clEnqueueReadBuffer() is used to read from a buffer object from device to host memory
- Similar to clEnqueueWriteBuffer()

```
cl_int clEnqueueReadBuffer (
    cl_command_queue queue,                  //Command queue to device
    cl_mem buffer,                           //OpenCL Buffer Object
    cl_bool blocking_read,                   //Blocking/Non-Blocking Flag
    size_t offset,                           //Offset to copy from
    size_t cb,                               //Size of data
    void *ptr,                               //Host pointer
    cl_uint num_in_wait_list,                //Number of events in wait list
    const cl_event *  event_wait_list,       //Array of events to wait for
    cl_event *event)                         //Event handler for this function
```

- The vector addition example discussed in Lecture 2 and 3 provide simple code snipped for moving data to and from devices

Example 1 - Image Rotation

- A common image processing routine
  - Applications in matching, alignment, etc.
- New coordinates of point (x_1,y_1) when rotated by an angle Θ around (x_0,y_0)

$$x_2 = \cos(\theta) * (x_1 - x_0) - \sin(\theta) * (y_1 - y_0) + x_0$$
$$y_2 = \sin(\theta) * (x_1 - x_0) + \cos(\theta) * (y_1 - y_0) + x_0$$

- By rotating the image about the origin (0,0) we get

$$x_2 = \cos(\theta) * (x_1) - \sin(\theta) * (y_1)$$
$$y_2 = \sin(\theta) * (x_1) + \cos(\theta) * (y_1)$$

- Each coordinate for every point in the image can be calculated independently

Original Image

Rotated Image (90^o)

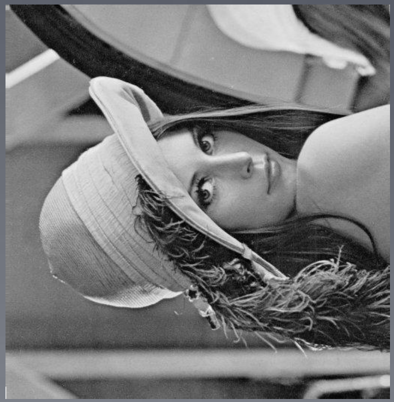
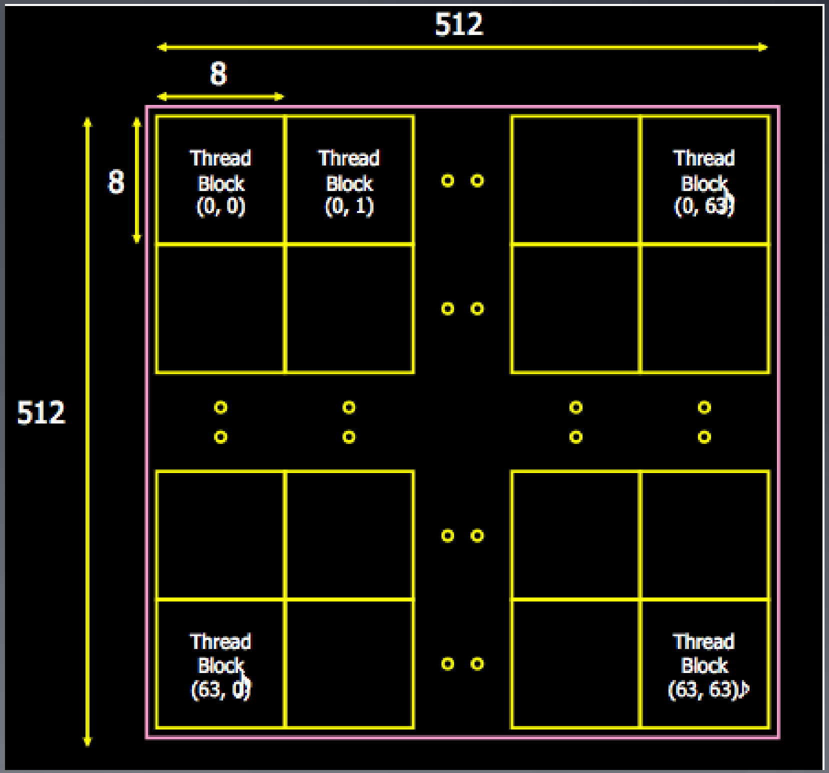


Image Rotation

- Input: To copy to device
  - Image (2D Matrix of floats)
  - Rotation parameters
  - Image dimensions
- Output: From device
  - Rotated Image
- Main Steps
  - Copy image to device by enqueueing a write to a buffer on the device from the host
  - Run the Image rotation kernel on input image
  - Copy output image to host by enqueueing a read from a buffer on the device



The OpenCL Kernel

- Parallel portion of the algorithm off-loaded to device
  - Most thought provoking part of coding process
- Steps to be done in Image Rotation kernel
  - Obtain coordinates of work item in work group
  - Read rotation parameters
  - Calculate destination coordinates
  - Read input and write rotated output at calculated coordinates
- Parallel kernel is not always this obvious.
  - Profiling of an application is often necessary to find the bottlenecks and locate the data parallelism
- In this example grid of output image decomposed into work items
  - Not all parts of the input image copied to the output image after rotation, corners of I/P image could be lost after rotation

OpenCL Kernel

```
__kernel  void image_rotate(
        __global float * src_data, __global float * dest_data,    //Data in global memory
        int W,    int H,                                          //Image Dimensions
        float sinTheta, float cosTheta )                          //Rotation Parameters
{
        //Thread gets its index within index space
        const int ix = get_global_id(0);
        const int iy = get_global_id(1);

        //Calculate location of data to move into ix and iy– Output decomposition as
mentioned
        float xpos = (  ((float) ix)*cosTheta + ((float)iy )*sinTheta);
        float ypos = (  ((float) iy)*cosTheta -  ((float)ix)*sinTheta);

        if ((     ((int)xpos>=0) && ((int)xpos< W)))             //Bound Checking
                && (((int)ypos>=0) && ((int)ypos< H)))
        {
                //Read (xpos,ypos) src_data and store at (ix,iy) in dest_data
                dest_data[iy*W+ix]=
                        src_data[(int)(floor(ypos*W+xpos))];
         }
}
```

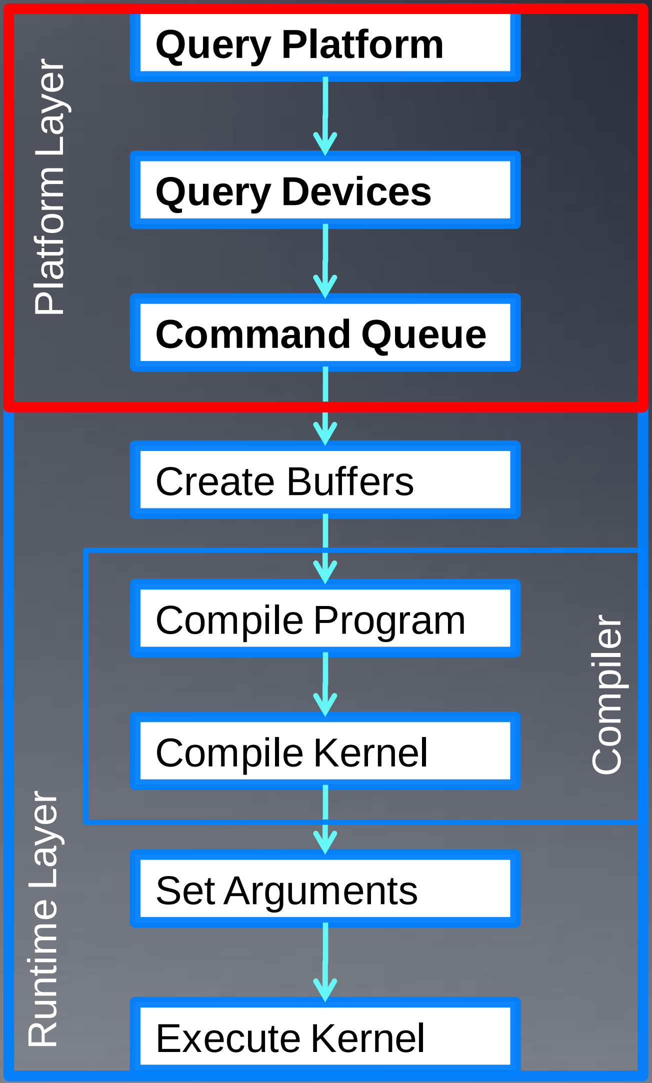
Step0: Initialize Device

- Declare context
- Choose a device from context
- Using device and context create a command queue

```
cl_context myctx = clCreateContextFromType (
                   0, CL_DEVICE_TYPE_GPU,
                        NULL, NULL, &ciErrNum);
```

```
ciErrNum = clGetDeviceIDs (0,
                     CL_DEVICE_TYPE_GPU,
                     1, &device,  cl_uint *num_devices)
```

```
cl_commandqueue myqueue ;
myqueue = clCreateCommandQueue(
                     myctx, device, 0, &ciErrNum);
```



Step1: Create Buffers

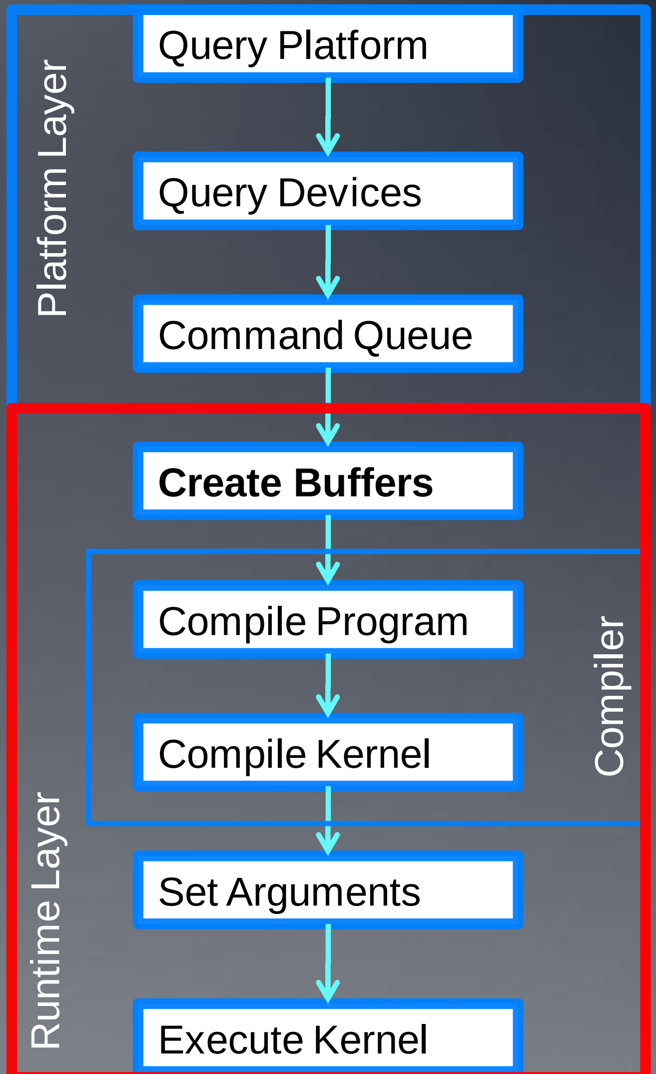
- Create buffers on device
  - Input data is read-only
  - Output data is write-only

```
cl_mem d_ip = clCreateBuffer(
                    myctx,
CL_MEM_READ_ONLY,
                    mem_size,
                    NULL, &ciErrNum);
```

```
cl_mem d_op = clCreateBuffer(
                    myctx,
CL_MEM_WRITE_ONLY,
                    mem_size,
                    NULL, &ciErrNum);
```

- Transfer input data to the device

```
ciErrNum = clEnqueueWriteBuffer (
             myqueue , d_ip, CL_TRUE,
             0, mem_size, (void *)src_image,
             0, NULL,  NULL)
```

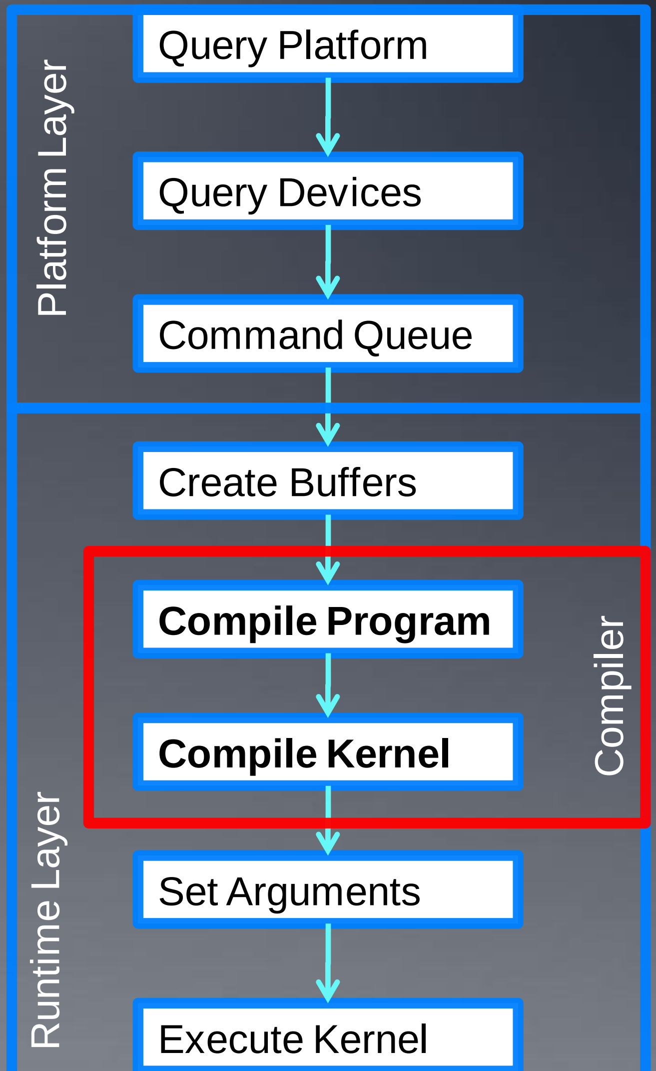


Step2: Build Program, Select Kernel

```
// create the program
cl_program myprog = clCreateProgramWithSource
                    ( myctx,1, (const char **)&source,
                    &program_length, &ciErrNum);
```

```
// build the program
ciErrNum = clBuildProgram( myprog, 0,
                    NULL, NULL, NULL, NULL);
```

```
//Use the "image_rotate" function as the kernel
cl_kernel mykernel = clCreateKernel (
                    myprog , "image_rotate" ,
                     error_code)
```

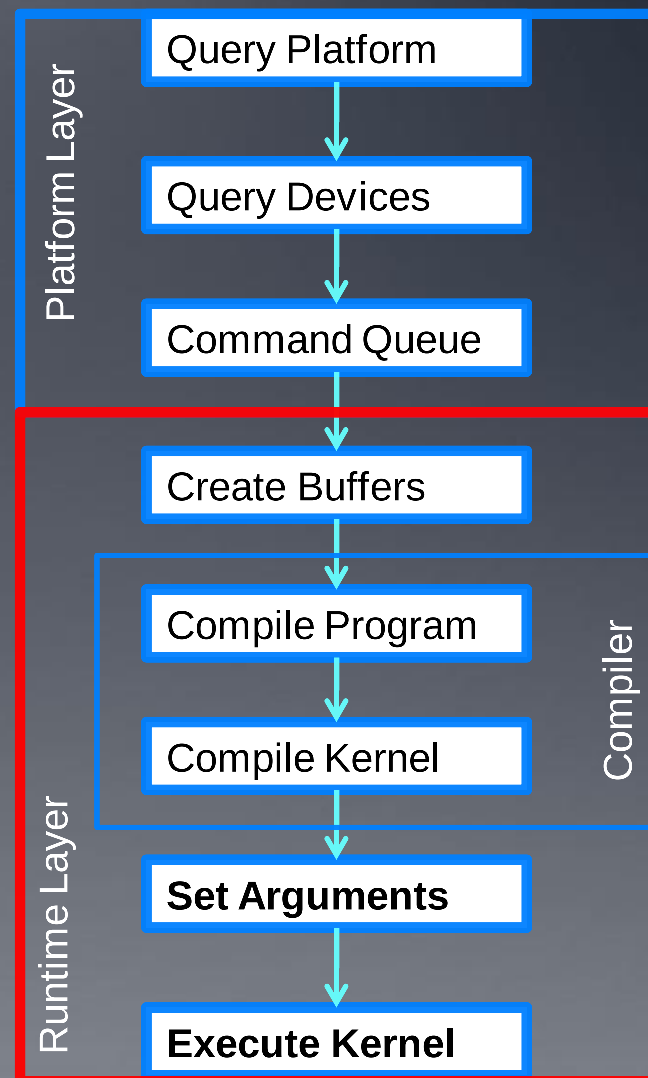


Step3: Set Arguments, Enqueue Kernel

```
// Set Arguments
clSetKernelArg(mykernel, 0, sizeof(cl_mem),
            (void *)&d_ip);
clSetKernelArg(mykernel, 1, sizeof(cl_mem),
           (void *)&d_op);
 clSetKernelArg(mykernel, 2, sizeof(cl_int),
            (void *)&W);
```

```
//Set local and global workgroup sizes
size_t localws[2] = {16,16} ;
size_t globalws[2] = {W, H};//Assume divisible by 16
```

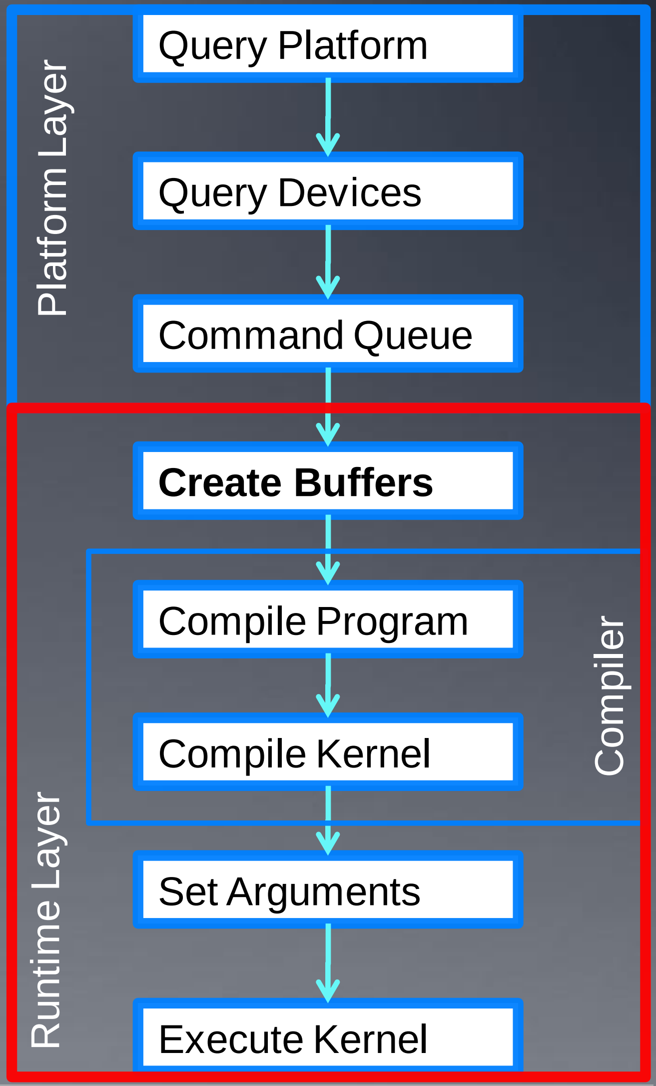
```
// execute kernel
clEnqueueNDRangeKernel(
            myqueue , myKernel,
            2, 0, globalws, localws,
            0, NULL, NULL);
```



Step4: Read Back Result

- Only necessary for data required on the host
- Data output from one kernel can be reused for another kernel
  - Avoid redundant host-device IO

```
// copy results from device back to host
clEnqueueReadBuffer(
            myctx, d_op,
            CL_TRUE,        //Blocking Read Back
            0, mem_size, (void *) op_data,
            NULL, NULL, NULL);
```



OpenCL Timing

- OpenCL provides “events” which can be used for timing kernels
  - Events will be discussed in detail in Lecture 11
- We pass an event to the OpenCL enqueue kernel function to capture timestamps
- Code snippet provided can be used to time a kernel
  - Add profiling enable flag to create command queue
  - By taking differences of the start and end timestamps we discount overheads like time spent in the command queue

```
cl_event event_timer;
clEnqueueNDRangeKernel(
            myqueue , myKernel,
            2, 0, globalws, localws,
            0, NULL, &event_timer);
```

```
unsigned long starttime, endtime;
```

```
clGetEventProfilingInfo( event_time,
      CL_PROFILING_COMMAND_START,
      sizeof(cl_ulong), &starttime, NULL);
```

```
clGetEventProfilingInfo(event_time,
      CL_PROFILING_COMMAND_END,
      sizeof(cl_ulong), &endtime, NULL);
```

```
unsigned long elapsed =
(unsigned long)(endtime - starttime);
```

Example 2
Matrix Multiplication

Basic Matrix Multiplication

- Non-blocking matrix multiplication
  - Doesn’t use local memory
    - Each element of matrix reads its own data independently
- Serial matrix multiplication

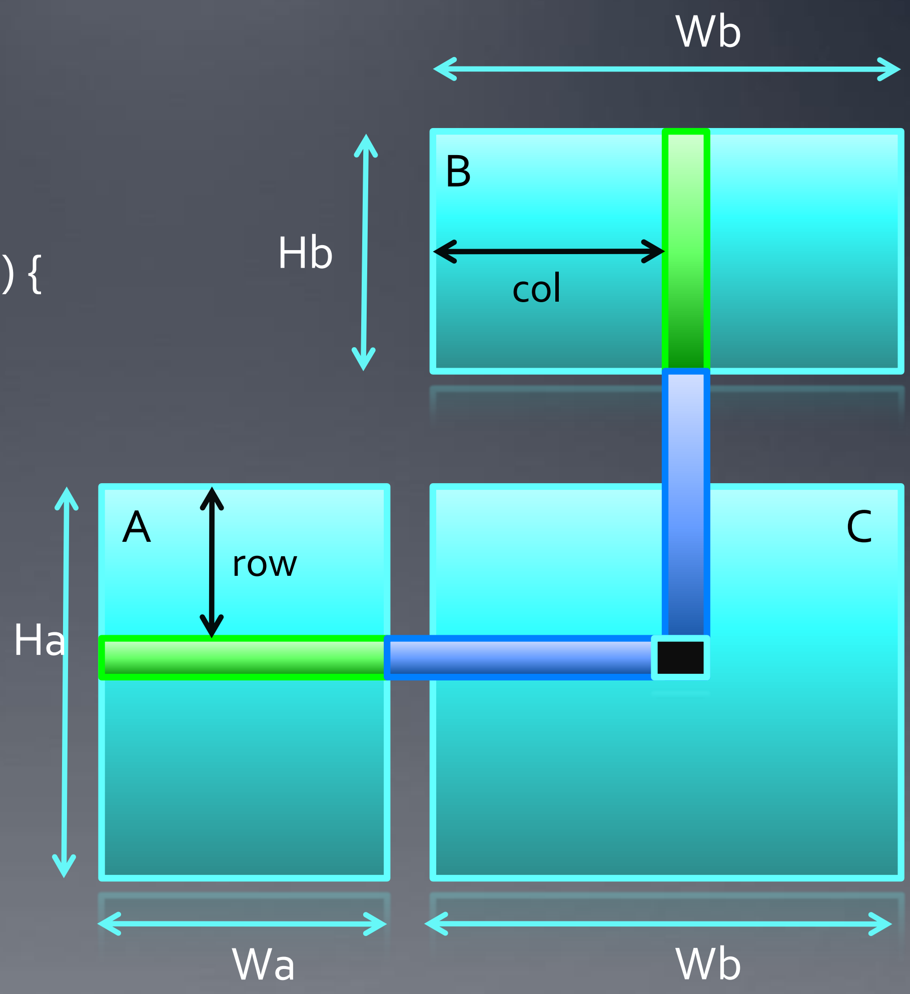
```
for(int i = 0; i < Ha; i++)
    for(int j = 0; j < Wb; j++){
        c[i][j] = 0;
        for(int k = 0; k < Wa; k++)
            c[i][j] +=  a[i][k] + b[k][j]
    }
```

- Reuse code from image rotation
  - Create context, command queues and compile program
  - Only need one more input memory object for 2nd matrix

Simple Matrix Multiplication

```
__kernel void simpleMultiply(
            __global float* c, int Wa, int Wb,
            __global float* a, __global float* b) {

    //Get global position in Y direction
    int row = get_global_id(1);
    //Get global position in X direction
    int col   = get_global_id(0);
    float sum = 0.0f;
    //Calculate result of one element
    for (int i = 0; i < Wa; i++)  {
            sum +=
                 a[row*Wa+i] * b[i*Wb+col];
    }
    c[row*Wb+col] = sum;
}
```



Summary

- We have studied the use of OpenCL buffer objects
- A complete program in OpenCL has been written
- We have understood how an OpenCL work-item can be used to work on a single output element (seen with rotation and matrix multiplication)
  - While the previously discussed examples are correct data parallel programs their performance can be drastically improved
- Next Lecture
  - Study the GPU memory subsystem to understand how data must be managed to obtain performance for data parallel programs
  - Understand possible optimizations for programs running on data parallel hardware like GPUs