OpenCL

CS532 - High Performance Computing

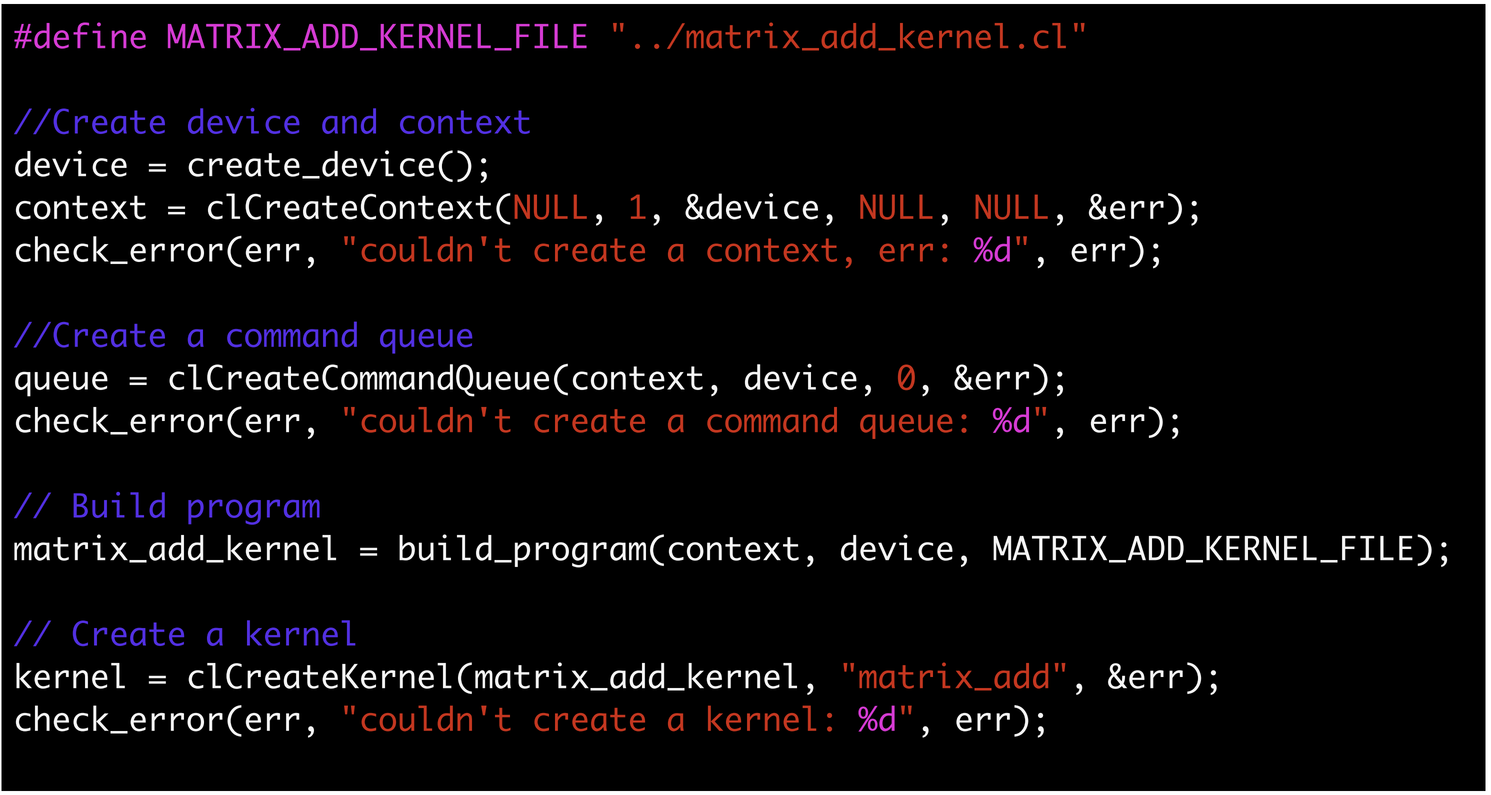
Example Code

- Code is from the matrix add OpenCL example on the course webpage.

OpenCL GPU Program Flow

1. create device
2. create context
3. create command queue
4. create kernel
5. copy memory to device
6. enqueue kernel
7. block for completion
8. copy memory to host

Initialization

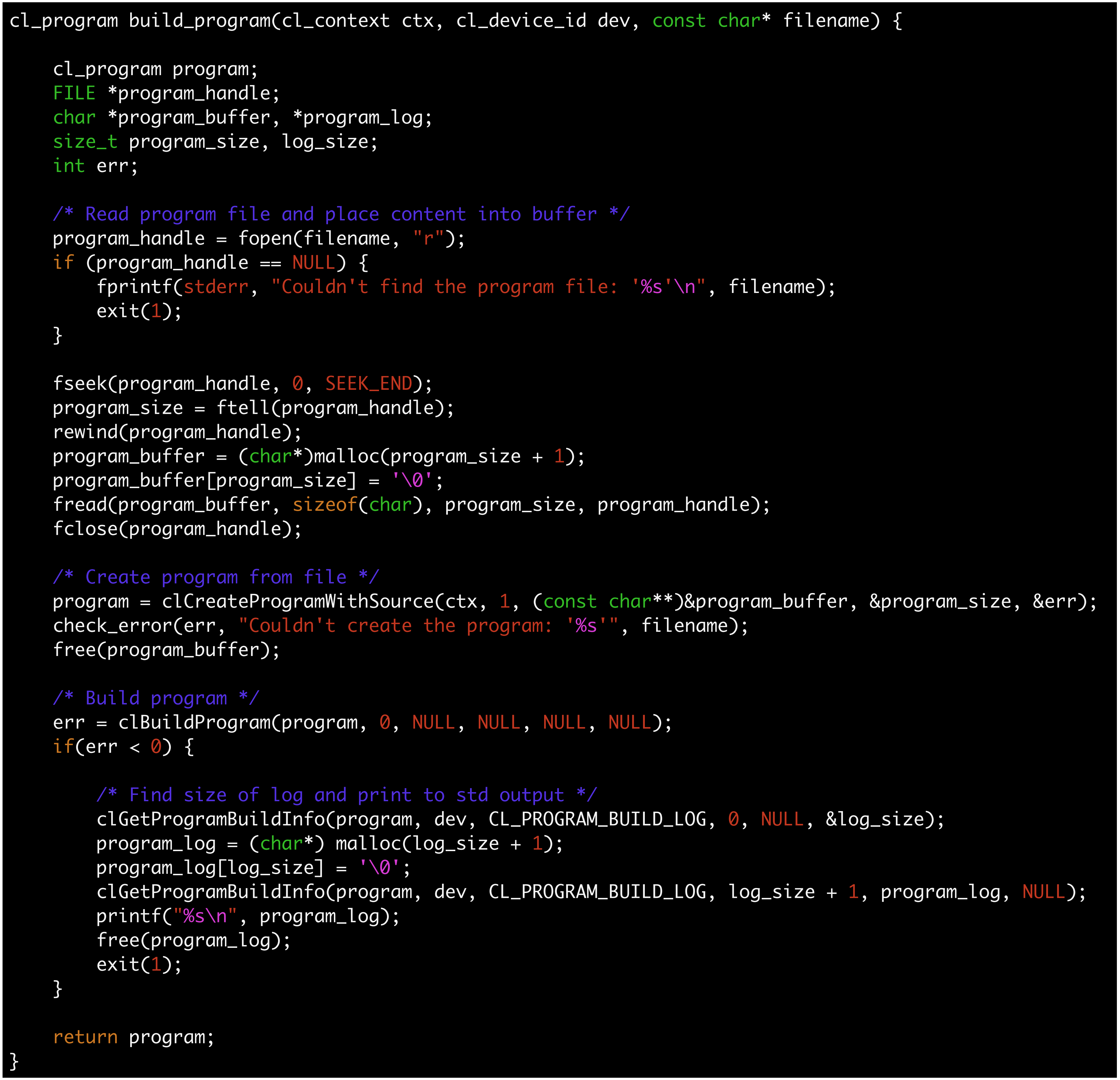


```
#define MATRIX_ADD_KERNEL_FILE "../matrix_add_kernel.cl"

//Create device and context
device = create_device();
context = clCreateContext(NULL, 1, &device, NULL, NULL, &err);
check_error(err, "couldn't create a context, err: %d", err);

//Create a command queue
queue = clCreateCommandQueue(context, device, 0, &err);
check_error(err, "couldn't create a command queue: %d", err);

// Build program
matrix_add_kernel = build_program(context, device, MATRIX_ADD_KERNEL_FILE);

// Create a kernel
kernel = clCreateKernel(matrix_add_kernel, "matrix_add", &err);
check_error(err, "couldn't create a kernel: %d", err);
```

- opencl_utils.cxx/ opencl_utils.hxx (on course webpage) provide create_device and build_program functions.

- The above creates a context, command queue, and program (kernel)

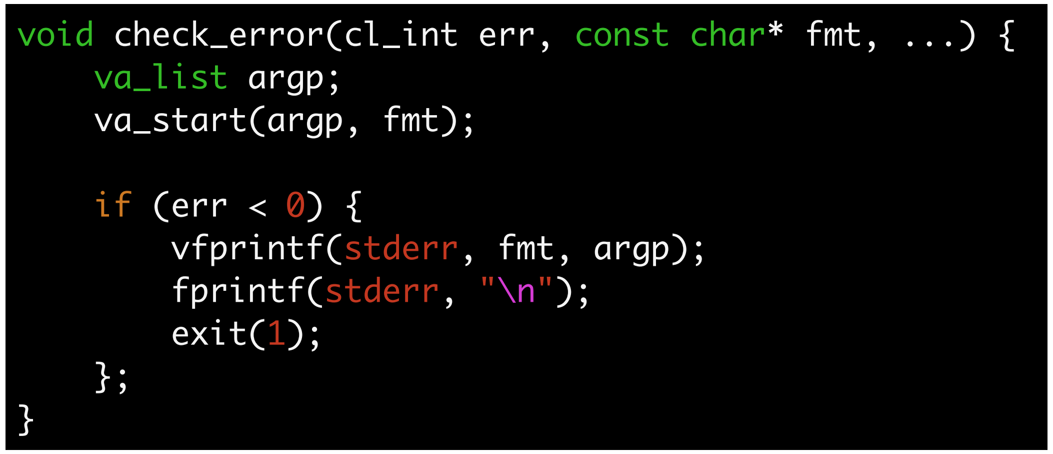
Creating a Kernel



```
cl_program build_program(cl_context ctx, cl_device_id dev, const char* filename) {

    cl_program program;
    FILE *program_handle;
    char *program_buffer, *program_log;
    size_t program_size, log_size;
    int err;

    /* Read program file and place content into buffer */
    program_handle = fopen(filename, "r");
    if (program_handle == NULL) {
        fprintf(stderr, "Couldn't find the program file: '%s'\n", filename);
        exit(1);
    }

    fseek(program_handle, 0, SEEK_END);
    program_size = ftell(program_handle);
    rewind(program_handle);
    program_buffer = (char*)malloc(program_size + 1);
    program_buffer[program_size] = '\0';
    fread(program_buffer, sizeof(char), program_size, program_handle);
    fclose(program_handle);

    /* Create program from file */
    program = clCreateProgramWithSource(ctx, 1, (const char**)&program_buffer, &program_size, &err);
    check_error(err, "Couldn't create the program: '%s'", filename);
    free(program_buffer);

    /* Build program */
    err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
    if(err < 0) {

        /* Find size of log and print to std output */
        clGetProgramBuildInfo(program, dev, CL_PROGRAM_BUILD_LOG, 0, NULL, &log_size);
        program_log = (char*) malloc(log_size + 1);
        program_log[log_size] = '\0';
        clGetProgramBuildInfo(program, dev, CL_PROGRAM_BUILD_LOG, log_size + 1, program_log, NULL);
        printf("%s\n", program_log);
        free(program_log);
        exit(1);
    }

    return program;
}
```

- opencl_utils.cxx/opencl_utils.hxx (on course webpage) provides a function to read a OpenCL kernel from a file, compile it and return a program.
- OpenCL is somewhat unique in that your code compiles the kernel for the particular device you are running it on.

Error Codes

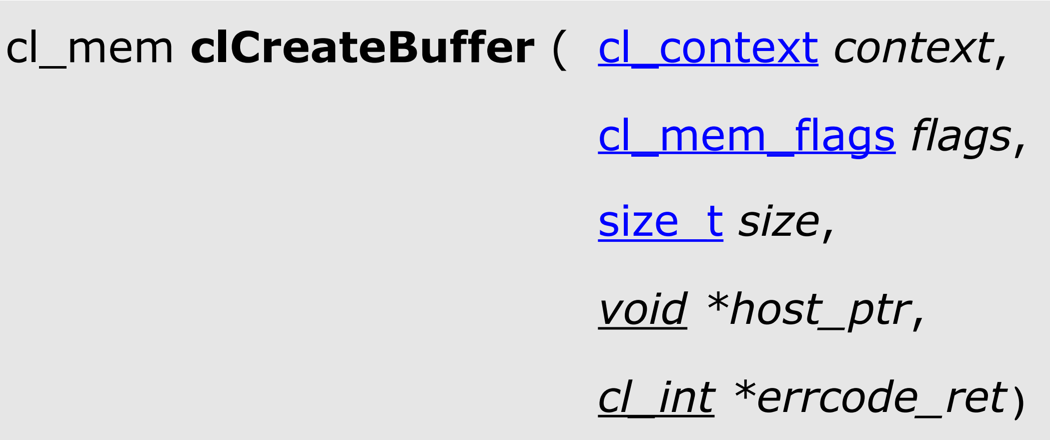


```
void check_error(cl_int err, const char* fmt, ...) {
    va_list argp;
    va_start(argp, fmt);

    if (err < 0) {
        vfprintf(stderr, fmt, argp);
        fprintf(stderr, "\n");
        exit(1);
    };
}
```

- OpenCL functions return error codes (in a *cl_int* type), which unfortunately you need to look up:

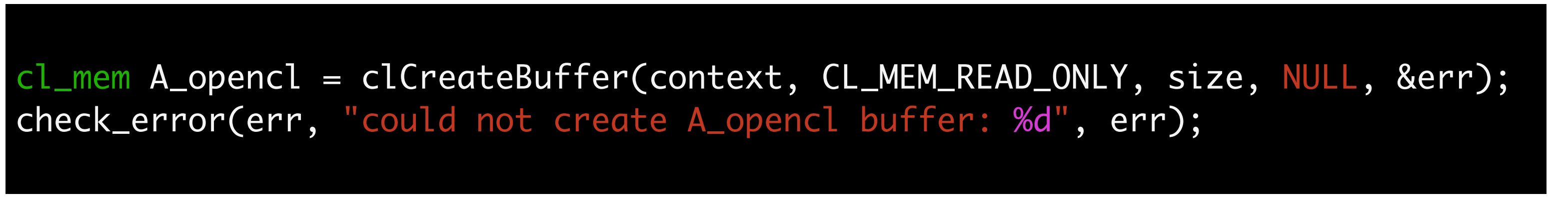
- https://streamhpc.com/blog/2013-04-28/opencl-error-codes/

Allocating Memory



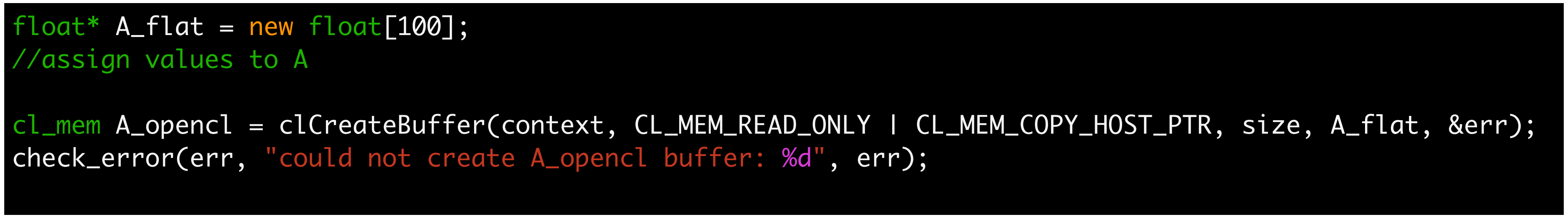
```
cl_mem clCreateBuffer ( cl_context context,
                        cl_mem_flags flags,
                        size_t size,
                        void *host_ptr,
                        cl_int *errcode_ret)
```

Allocate memory on device only.



```
cl_mem A_opencl = clCreateBuffer(context, CL_MEM_READ_ONLY, size, NULL, &err);
check_error(err, "could not create A_opencl buffer: %d", err);
```

Allocate memory and copy values from host.



```
float* A_flat = new float[100];
//assign values to A

cl_mem A_opencl = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, size, A_flat, &err);
check_error(err, "could not create A_opencl buffer: %d", err);
```

- Returns a "cl_mem" object which refers to the memory on the OpenCL device.
- CL_MEM_READ_ONLY goes to constant memory on a GPU.

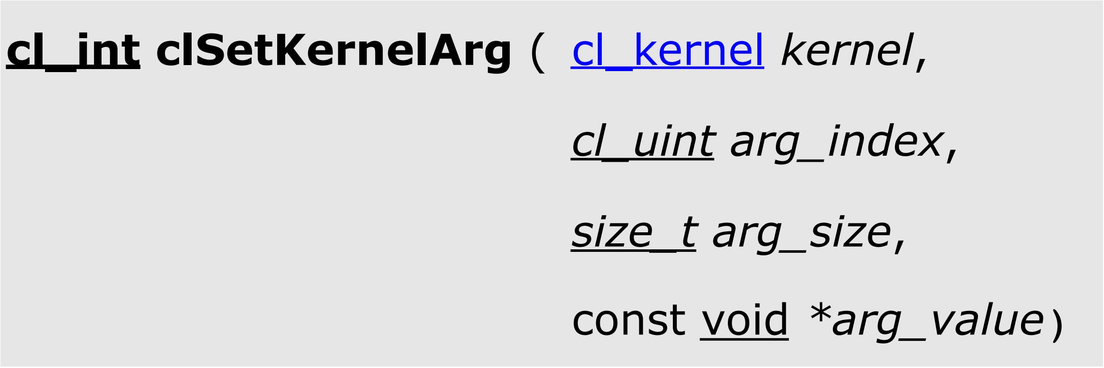
https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/clCreateBuffer.html

Allocating Memory Flags

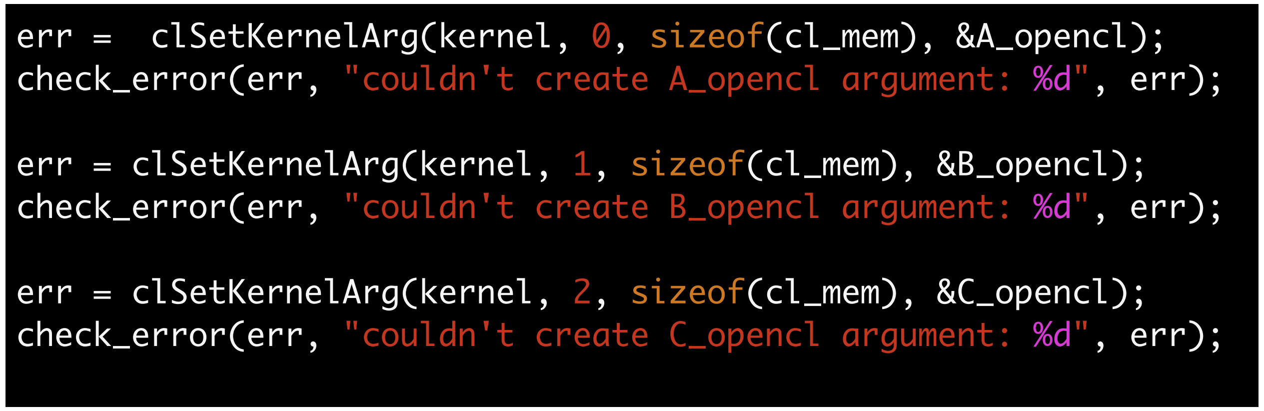
cl_mem_flags	**Description**
`CL_MEM_READ_WRITE`	This flag specifies that the memory object will be read and written by a kernel. This is the default.
`CL_MEM_WRITE_ONLY`	This flags specifies that the memory object will be written but not read by a kernel. Reading from a buffer or image object created with `CL_MEM_WRITE_ONLY` inside a kernel is undefined.
`CL_MEM_READ_ONLY`	This flag specifies that the memory object is a read-only memory object when used inside a kernel. Writing to a buffer or image object created with `CL_MEM_READ_ONLY` inside a kernel is undefined.
`CL_MEM_USE_HOST_PTR`	This flag is valid only if *host_ptr* is not NULL. If specified, it indicates that the application wants the OpenCL implementation to use memory referenced by *host_ptr* as the storage bits for the memory object. OpenCL implementations are allowed to cache the buffer contents pointed to by *host_ptr* in device memory. This cached copy can be used when kernels are executed on a device. The result of OpenCL commands that operate on multiple buffer objects created with the same *host_ptr* or overlapping host regions is considered to be undefined.
`CL_MEM_ALLOC_HOST_PTR`	This flag specifies that the application wants the OpenCL implementation to allocate memory from host accessible memory. `CL_MEM_ALLOC_HOST_PTR` and `CL_MEM_USE_HOST_PTR` are mutually exclusive.
`CL_MEM_COPY_HOST_PTR`	This flag is valid only if *host_ptr* is not NULL. If specified, it indicates that the application wants the OpenCL implementation to allocate memory for the memory object and copy the data from memory referenced by *host_ptr*. `CL_MEM_COPY_HOST_PTR` and `CL_MEM_USE_HOST_PTR` are mutually exclusive. `CL_MEM_COPY_HOST_PTR` can be used with `CL_MEM_ALLOC_HOST_PTR` to initialize the contents of the cl_mem object allocated using host-accessible (e.g. PCIe) memory.

https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/clCreateBuffer.html

Setting Kernel Arguments



```
cl_int clSetKernelArg ( cl_kernel kernel,
                        cl_uint arg_index,
                        size_t arg_size,
                        const void *arg_value)
```

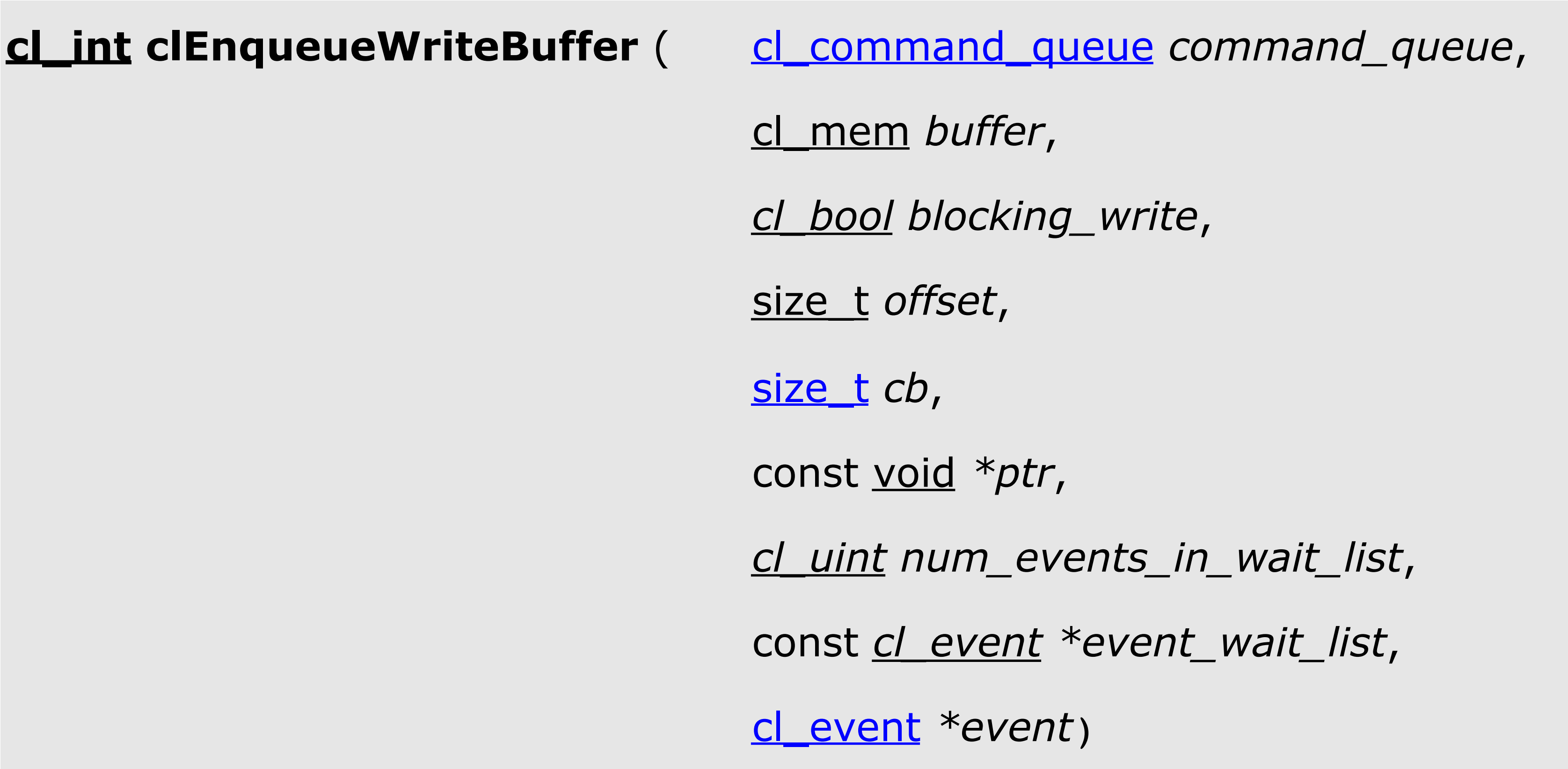


```
err =  clSetKernelArg(kernel, 0, sizeof(cl_mem), &A_opencl);
check_error(err, "couldn't create A_opencl argument: %d", err);

err = clSetKernelArg(kernel, 1, sizeof(cl_mem), &B_opencl);
check_error(err, "couldn't create B_opencl argument: %d", err);

err = clSetKernelArg(kernel, 2, sizeof(cl_mem), &C_opencl);
check_error(err, "couldn't create C_opencl argument: %d", err);
```

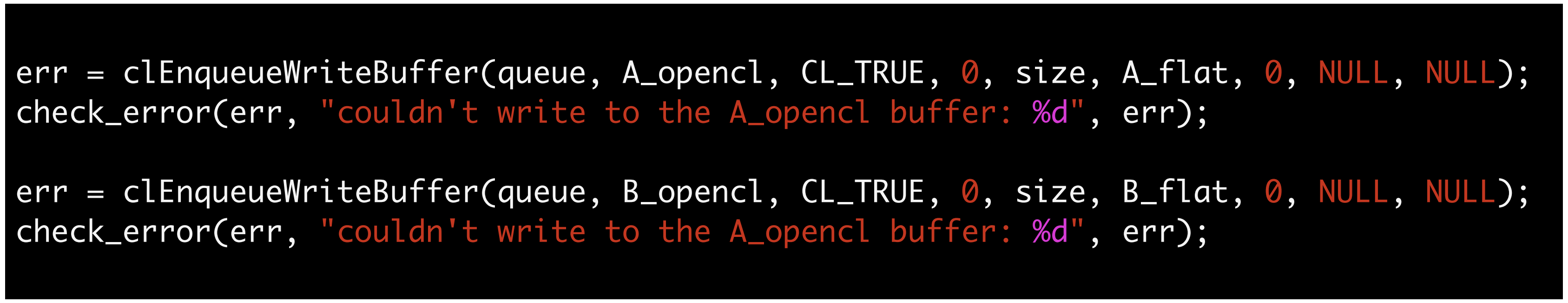
- You need to set up each argument to the kernel individually.
- *arg_index* is the position in the argument list
- *arg_size* is the size of the variable
- *arg_value* is the cl_mem variable

https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/clSetKernelArg.html

Transferring Memory to Device



```
cl_int clEnqueueWriteBuffer (   cl_command_queue command_queue,
                                cl_mem buffer,
                                cl_bool blocking_write,
                                size_t offset,
                                size_t cb,
                                const void *ptr,
                                cl_uint num_events_in_wait_list,
                                const cl_event *event_wait_list,
                                cl_event *event)
```



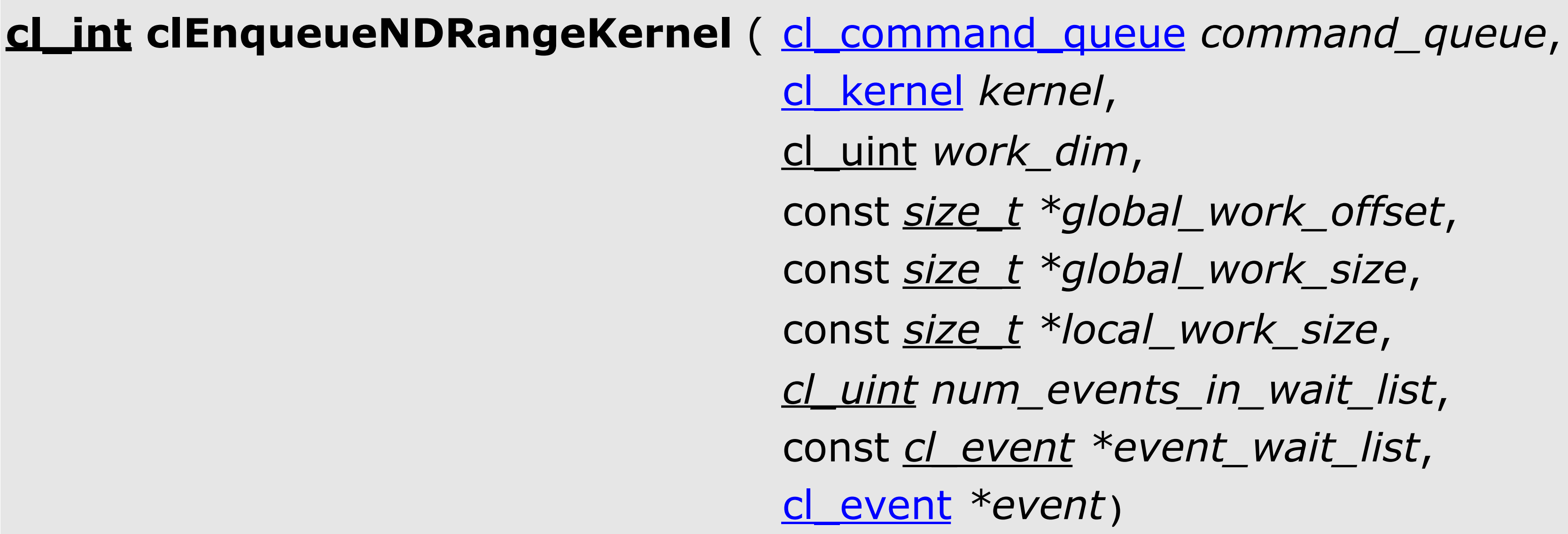
```
err = clEnqueueWriteBuffer(queue, A_opencl, CL_TRUE, 0, size, A_flat, 0, NULL, NULL);
check_error(err, "couldn't write to the A_opencl buffer: %d", err);

err = clEnqueueWriteBuffer(queue, B_opencl, CL_TRUE, 0, size, B_flat, 0, NULL, NULL);
check_error(err, "couldn't write to the A_opencl buffer: %d", err);
```

- *buffer* is the memory being written to on the host
- *offset* is the offset in bytes to start writing the memory in the buffer
- *cb* is the size (in bytes) to be written
- *ptr* is where the memory is being read from to be written

- Can ignore events_in_wait_list, event_wait_list and event for now.

https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/clEnqueueWriteBuffer.html

clEnqueueNDRangeKernel



```
cl_int clEnqueueNDRangeKernel ( cl_command_queue command_queue,
                                cl_kernel kernel,
                                cl_uint work_dim,
                                const size_t *global_work_offset,
                                const size_t *global_work_size,
                                const size_t *local_work_size,
                                cl_uint num_events_in_wait_list,
                                const cl_event *event_wait_list,
                                cl_event *event)
```

```
err = clEnqueueNDRangeKernel(queue, kernel, 3, NULL, global_size, local_size, 0, NULL, NULL);
check_error(err, "couldn't enqueue the kernel: %d", err);
```

- Used to load a kernel for use on GPU (or other device).
- *work_dim* is how many dimensions the work has (1, 2 or 3).
- *global_work_size* is the *total* number of items of work (this is different than a CUDA grid), *local_work_size* is the number of items in a local group (think a CUDA block).
- Can ignore events_in_wait_list, event_wait_list and event for now.

https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/clEnqueueNDRangeKernel.html

CUDA vs OpenCL work sizes

OpenCL

```
size_t* global_size = (size_t*)malloc(sizeof(size_t) * 3);
global_size[0] = 20;
global_size[1] = 20;
global_size[2] = 20;

size_t* local_size = (size_t*)malloc(sizeof(size_t) * 3);
local_size[0] = 5;
local_size[1] = 5;
local_size[2] = 5;
```

CUDA

```
dim3 dimGrid(4, 4, 4);
dim3 dimBlock(5, 5, 5);
```

- These two code samples will generate the same number of work items/threads. Note that the global_size for OpenCL is equal to the grid size * block size in CUDA.

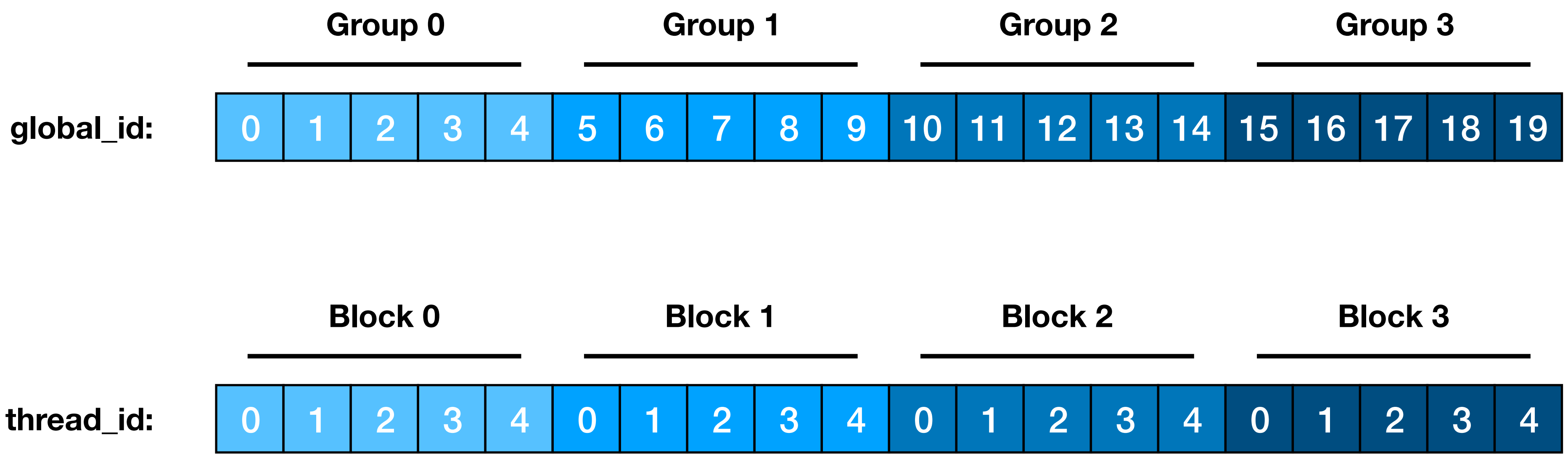
CUDA vs OpenCL work sizes

OpenCL

```
size_t* global_size = (size_t*)malloc(sizeof(size_t) * 1);
global_size[0] = 20;

size_t* local_size = (size_t*)malloc(sizeof(size_t) * 1);
local_size[0] = 5;
```

CUDA

```
int grid_size = 4;
dim3 block_size = 5;
```



	Group 0					Group 1					Group 2					Group 3				
global_id:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

	Block 0					Block 1					Block 2					Block 3				
thread_id:	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4

- In CUDA, thread_ids "reset" to 0 for each block, however in OpenCL, the global_id will be unique (in that dimension).

Blocking for Kernel Completion

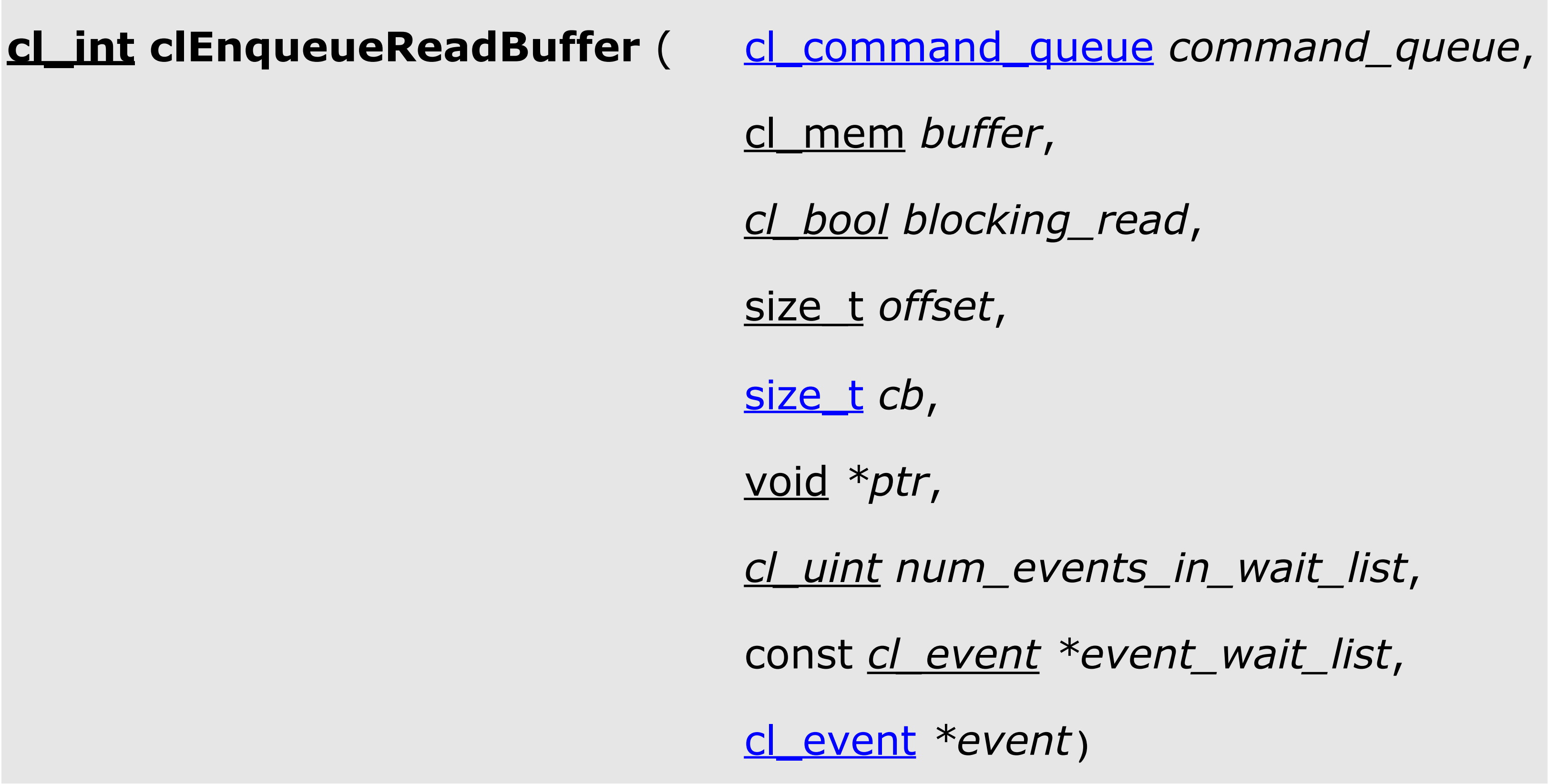
cl_int clFinish (cl_command_queue *command_queue*)

```
err = clFinish(queue);
check_error(err, "queue errored on finish: %d", err);
```

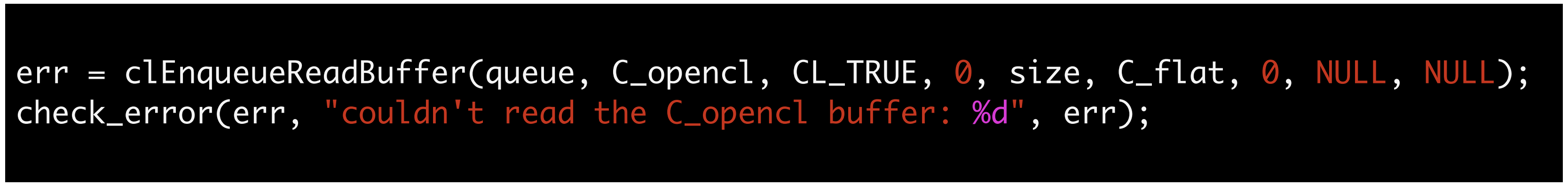
- *clFinish* blocks until all previously issued commands to the command queue have finished.

https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/clFinish.html

Reading Results to Host



```
cl_int clEnqueueReadBuffer (    cl_command_queue command_queue,
                                cl_mem buffer,
                                cl_bool blocking_read,
                                size_t offset,
                                size_t cb,
                                void *ptr,
                                cl_uint num_events_in_wait_list,
                                const cl_event *event_wait_list,
                                cl_event *event)
```

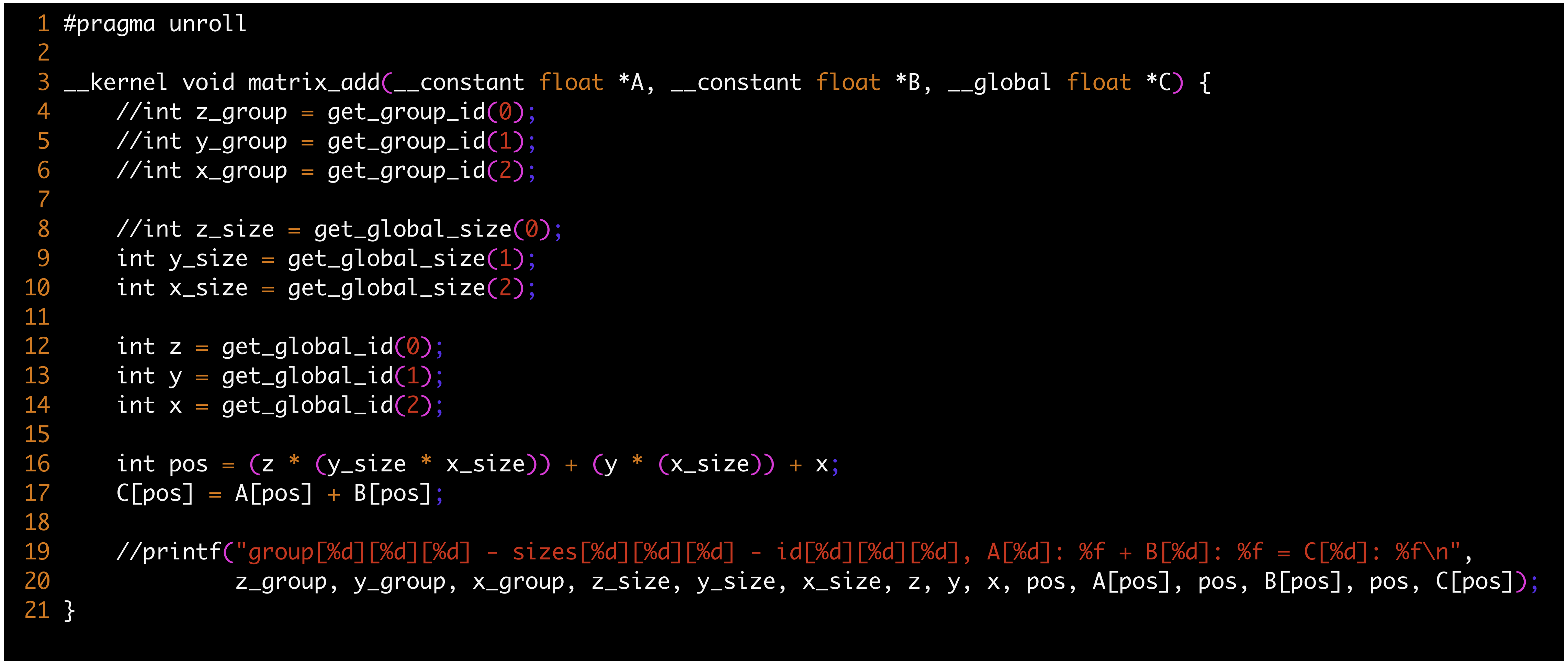


```
err = clEnqueueReadBuffer(queue, C_opencl, CL_TRUE, 0, size, C_flat, 0, NULL, NULL);
check_error(err, "couldn't read the C_opencl buffer: %d", err);
```

- Arguments are the same as as *clEnqueueWriteBuffer*, this is done after the kernel finishes (*clEnqueueNDRangeKernel* and *clFinish*).
- Memory is copied from the device (*buffer*) to the host (*ptr*).

https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/clEnqueueReadBuffer.html

Matrix Add Kernel



```
#pragma unroll

__kernel void matrix_add(__constant float *A, __constant float *B, __global float *C) {
    //int z_group = get_group_id(0);
    //int y_group = get_group_id(1);
    //int x_group = get_group_id(2);

    //int z_size = get_global_size(0);
    int y_size = get_global_size(1);
    int x_size = get_global_size(2);

    int z = get_global_id(0);
    int y = get_global_id(1);
    int x = get_global_id(2);

    int pos = (z * (y_size * x_size)) + (y * (x_size)) + x;
    C[pos] = A[pos] + B[pos];

    //printf("group[%d][%d][%d] - sizes[%d][%d][%d] - id[%d][%d][%d], A[%d]: %f + B[%d]: %f = C[%d]: %f\n",
             z_group, y_group, x_group, z_size, y_size, x_size, z, y, x, pos, A[pos], pos, B[pos], pos, C[pos]);
}
```

- __constant is constant memory, __global is global memory
- You can use the get_group_id, get_global_size and get_global_id functions to determine which thread you are and what the group sizes are.

https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/clEnqueueReadBuffer.html