Appendix D - Examples

The examples described in appendix D are not considered as part of the official OpenCL specification. They are included to illustrate how to use the OpenCL APIs to execute a kernel on a device, and common algorithms such as matrix tranpose and reduction written as OpenCL kernels. These examples have been written with the goal of illustrating how to use OpenCL and should not be considered as examples of how to write performant OpenCL kernels.

D.1 A Simple OpenCL Kernel

This example demonstrates how a kernel can be written that operates on individual data elements of a memory object which is very similar to what is allowed in GLSL / HLSL. In this example we show a function that computes a dot product of two float4 arrays and writes result to a float array.

The OpenCL kernel source is:

```
__kernel void
dot_product (__global const float4 *a,
             __global const float4 *b, __global float *c)
{
    int gid = get_global_id(0);

    c[gid] = dot(a[gid], b[gid]);
}
```

The following code describes the OpenCL runtime calls the application must make to create appropriate memory objects, create program object and load the program source for the kernel described above, build the program executable, create the kernel object, load the appropriate argument values and execute the dot product kernel on the GPU device.

```
void
delete_memobjs(cl_mem *memobjs, int n)
{
    int i;
    for (i=0; i<n; i++)
        clReleaseMemObject(memobjs[i]);
}

int
exec_dot_product_kernel(const char *program_source,
                        int n, void *srcA, void *srcB, void *dst)
{
    cl_context        context;
```

```
cl_command_queue  cmd_queue;
cl_device_id      *devices;
cl_program        program;
cl_kernel         kernel;
cl_mem            memobjs[3];
size_t            global_work_size[1];
size_t            local_work_size[1];
size_t            cb;
cl_int            err;

// create the OpenCL context on a GPU device
context = clCreateContextFromType(NULL, CL_DEVICE_TYPE_GPU,
                                  NULL, NULL, NULL);
if (context == (cl_context)0)
    return -1;

// get the list of GPU devices associated with context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &cb);
devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb, devices, NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context, devices[0], 0, NULL);
if (cmd_queue == (cl_command_queue)0)
{
    clReleaseContext(context);
    free(devices);
    return -1;
}
free(devices);

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context,
                            CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                            sizeof(cl_float4) * n, srcA, NULL);
if (memobjs[0] == (cl_mem)0)
{
    clReleaseCommandQueue(cmd_queue);
    clReleaseContext(context);
    return -1;
}

memobjs[1] = clCreateBuffer(context,
                            CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                            sizeof(cl_float4) * n, srcB, NULL);
if (memobjs[1] == (cl_mem)0)
{
    delete_memobjs(memobjs, 1);
    clReleaseCommandQueue(cmd_queue);
    clReleaseContext(context);
    return -1;
```

```
}

memobjs[2] = clCreateBuffer(context,
                            CL_MEM_READ_WRITE,
                            sizeof(cl_float) * n, NULL, NULL);
if (memobjs[2] == (cl_mem)0)
{
    delete_memobjs(memobjs, 2);
    clReleaseCommandQueue(cmd_queue);
    clReleaseContext(context);
    return -1;
}

// create the program
program = clCreateProgramWithSource(context,
                  1, (const char**)&program_source, NULL, NULL);
if (program == (cl_program)0)
{
    delete_memobjs(memobjs, 3);
    clReleaseCommandQueue(cmd_queue);
    clReleaseContext(context);
    return -1;
}

// build the program
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
if (err != CL_SUCCESS)
{
    delete_memobjs(memobjs, 3);
    clReleaseProgram(program);
    clReleaseCommandQueue(cmd_queue);
    clReleaseContext(context);
    return -1;
}

// create the kernel
kernel = clCreateKernel(program, "dot_product", NULL);
if (kernel == (cl_kernel)0)
{
    delete_memobjs(memobjs, 3);
    clReleaseProgram(program);
    clReleaseCommandQueue(cmd_queue);
    clReleaseContext(context);
    return -1;
}

// set the args values
err = clSetKernelArg(kernel,  0,
                  sizeof(cl_mem), (void *) &memobjs[0]);
err |= clSetKernelArg(kernel, 1,
                  sizeof(cl_mem), (void *) &memobjs[1]);
err |= clSetKernelArg(kernel, 2,
```

```
                    sizeof(cl_mem), (void *) &memobjs[2]);

    if (err != CL_SUCCESS)
    {
        delete_memobjs(memobjs, 3);
        clReleaseKernel(kernel);
        clReleaseProgram(program);
        clReleaseCommandQueue(cmd_queue);
        clReleaseContext(context);
        return -1;
    }

    // set work-item dimensions
    global_work_size[0] = n;
    local_work_size[0]= 1;

    // execute kernel
    err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL,
                                 global_work_size, local_work_size,
                                 0, NULL, NULL);
    if (err != CL_SUCCESS)
    {
        delete_memobjs(memobjs, 3);
        clReleaseKernel(kernel);
        clReleaseProgram(program);
        clReleaseCommandQueue(cmd_queue);
        clReleaseContext(context);
        return -1;
    }

    // read output image
    err = clEnqueueReadBuffer(cmd_queue, memobjs[2], CL_TRUE,
                              0, n * sizeof(cl_float), dst,
                              0, NULL, NULL);
    if (err != CL_SUCCESS)
    {
        delete_memobjs(memobjs, 3);
        clReleaseKernel(kernel);
        clReleaseProgram(program);
        clReleaseCommandQueue(cmd_queue);
        clReleaseContext(context);
        return -1;
    }

    // release kernel, program, and memory objects
    delete_memobjs(memobjs, 3);
    clReleaseKernel(kernel);
    clReleaseProgram(program);
    clReleaseCommandQueue(cmd_queue);
    clReleaseContext(context);
    return 0; // success...
}
```