UOW AUSTRALIA

SCIT

School of Computing & Information Technology

CSCI376 – Multicore and GPU Programming

Kernel Execution

The aim of this tutorial is to introduce you to basic kernels and sending kernels to a device for execution.

Global IDs and offsets

Have a look at the kernel code in global_id.cl in Tutorial4a.

The code gets a work-item's global ID and offset using built-in OpenCL kernel functions:

```
int i = get_global_id(0);
int offset = get_global_offset(0);
```

It then computes an array index for where it should store the global id as output:

```
int index = i - offset;
```

Each work-item will display its own global ID

(NOTE: printf in the kernel should only be used for debugging purposes).

```
printf("Global ID = %d\n", i);
```

Finally, the work-item's global ID is stored in the output array using the index that was previously computed:

```
output[index] = i;
```

Now, have a look at the host code in tutorial4a.cpp. The description below highlights some key sections of the code.

The offset and number of work-items to create are defined as:

```
#define OFFSET 0
#define NUM_OF_WORK_ITEMS 10
```

The vector and buffer to store the output are declared:

```
std::vector<cl_int> result(NUM_OF_WORK_ITEMS);
cl::Buffer outputBuffer;
```

An empty buffer is created using:

```
outputBuffer = cl::Buffer(context, CL_MEM_WRITE_ONLY,
       sizeof(cl_int) * NUM_OF_WORK_ITEMS);
```

The buffer is set as the kernel argument using;

```
kernel.setArg(0, outputBuffer);
```

Next, the kernel is enqueued for execution. The offset sets the starting global ID offset, while the globalsize sets the number of work-items to create:

```
cl::NDRange offset(OFFSET);
cl::NDRange globalSize(NUM_OF_WORK_ITEMS);

queue.enqueueNDRangeKernel(kernel, offset, globalSize);
```

If the following command is used instead of enqueueNDRangeKernel, only 1 work-item will be created:

```
queue.enqueueTask(kernel);
```

Once the kernel has completed its execution, we can read the output to host memory:

```
queue.enqueueReadBuffer(outputBuffer, CL_TRUE, 0,
      sizeof(cl_float) * NUM_OF_WORK_ITEMS, &result[0]);
```

We can then output the results:

```
for (int i = 0; i < NUM_OF_WORK_ITEMS; i++)
{
      std::cout << "Work-item " << i << "'s global ID: " << result[i] << std::endl;
}
```

Matrix-vector multiplication

Have a look at the code in the tutorial4b.cpp and matvec.cl files in Tutorial4b. The program performs a 4×4 matrix multiplication with a 4×1 vector. This is done by creating 4 work-items. Each work-item will access a row of the matrix (i.e. 4 elements), perform a dot product with the vector (also contains 4 elements), then store the results in an output buffer.

The following is the code in the kernel:

```
__kernel void matvec_mult(__global float4* matrix,
                          __global float4* vector,
                          __global float* result) {

   int i = get_global_id(0);
   result[i] = dot(matrix[i], vector[0]);
}
```

The data and buffers were declared as:

```
std::vector<cl_float> mat(16), vec(4), result(4), correctResult(4);
cl::Buffer matBuffer, vecBuffer, resultBuffer;
```

The data for some of the vectors were initialised as follows:

```
for (int i = 0; i < 16; i++)
```

```
{
	mat[i] = i * 2.0f;
}
for (int i = 0; i < 4; i++)
{
	vec[i] = i * 3.0f;
	correctResult[0] += mat[i] * vec[i];
	correctResult[1] += mat[i + 4] * vec[i];
	correctResult[2] += mat[i + 8] * vec[i];
	correctResult[3] += mat[i + 12] * vec[i];
}
```

The buffers are created using:

```
matBuffer = cl::Buffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
	sizeof(cl_float) * 16, &mat[0]);
vecBuffer = cl::Buffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
	sizeof(cl_float) * 4, &vec[0]);
resultBuffer = cl::Buffer(context, CL_MEM_WRITE_ONLY, sizeof(cl_float) * 4);
```

Notice that the matrix buffer contains 16 elements, whereas the vector buffer and the result buffer contain 4 elements. The buffers are set as kernel arguments as follows:

```
kernel.setArg(0, matBuffer);
kernel.setArg(1, vecBuffer);
kernel.setArg(2, resultBuffer);
```

The kernel is enqueued in the command queue using:

```
cl::NDRange offset(0);
cl::NDRange globalSize(4);

queue.enqueueNDRangeKernel(kernel, offset, globalSize);
```

This means that the kernel will execute on 4 work-items.

If you look at the code for the kernel, it accepts a pointer to a `float4` vector data type called `matrix` (this gets a row of the matrix), and a pointer to a `float4` vector data type called `vector` and a float pointer for the `result`. The following code in the kernel gets a work-item's global ID, which allows the kernel to access the appropriate data and write the results in the appropriate location:

```
int i = get_global_id(0);
result[i] = dot(matrix[i], vector[0]);
```

After the kernel has completed its execution, the reads the results from device to host, checks whether it is correct and displays the contents.

Vector addition

Have a look at the code in tutorial4c.cpp and vecadd.cl files in Tutorial4c.

The program adds the components of two arrays. The following is defined at the top of the file:

```
#define LENGTH 1000
```

This defines the number of elements in the vectors, which are declared as follows:

```
std::vector<cl_float> vectorA(LENGTH);
std::vector<cl_float> vectorB(LENGTH);
std::vector<cl_float> result(LENGTH);
std::vector<cl_float> correctResult(LENGTH);
```

Then three memory objects are declared:

```
cl::Buffer bufferA, bufferB, resultBuffer;
```

This is followed by code to initialise the `vectorA` and `vectorB` to random values. The `result` vector to all 0s, and the `correctResult` vector to the result of `vectorA + vectorB`:

```
for (int i = 0; i < LENGTH; i++)
{
	vectorA[i] = 1.0f + static_cast<float>(rand()) /
		(static_cast<float>(RAND_MAX / (100.0f - 1.0f)));
	vectorB[i] = 1.0f + static_cast<float>(rand()) /
		(static_cast<float>(RAND_MAX / (100.0f - 1.0f)));
	result[i] = 0.0f;
	correctResult[i] = vectorA[i] + vectorB[i];
}
```

Further down in the code, three buffer objects are created; two for the input data (created as read-only) and one for the result (write-only):

```
bufferA = cl::Buffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
	sizeof(cl_float) * LENGTH, &vectorA[0]);
bufferB = cl::Buffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
	sizeof(cl_float) * LENGTH, &vectorB[0]);
resultBuffer = cl::Buffer(context, CL_MEM_WRITE_ONLY, sizeof(cl_float) * LENGTH);
```

Then, the kernel arguments are set:

```
kernel.setArg(0, bufferA);
kernel.setArg(1, bufferB);
kernel.setArg(2, resultBuffer);
```

And the kernel is enqueued:

```
cl::NDRange offset(0);
cl::NDRange globalSize(LENGTH);    // work-units per kernel

queue.enqueueNDRangeKernel(kernel, offset, globalSize);
```

Note that this will create `LENGTH` work-items.

If you try using the following instead, the program will only add the first element of the arrays:

```
queue.enqueueTask(kernel);
```

This is because the above command is equivalent to:

```
cl::NDRange offset(0);
cl::NDRange globalSize(1);

queue.enqueueNDRangeKernel(kernel, offset, globalSize);
```

After the kernel has been executed on the device, the results are read from the buffer into the `result` vector and checked whether the results are correct.

Exercises

Try the following:

- In Tutorial4a, try changing the number of work-items and starting offset value.
- Chaining vector additions.

  Based on the code in Tutorial4c, *without changing the code in the kernel*, modify the code in tutorial4c.cpp (or write your own code) to chain vector additions. Your program should call the kernel 3 times to do the following:

  c = a + b

  d = c + e

  f = d + g

References

Among others, the material in this tutorial was sourced from:

- M. Scarpino, “OpenCL in Action: How to Accelerate Graphics and Computation,” Manning
- Khronos Group, “The OpenCL Specifications”, version 1.2
- S. McIntosh-Smith and T. Deakin, “Hands on OpenCL”