SCIT

School of Computing & Information Technology

CSCI376 – Multicore and GPU Programming

Execution and Memory Model

This tutorial demonstrates the concept of work-items and work-groups, along with how to obtain relevant information within a kernel. This tutorial also illustrates different OpenCL memory address spaces and how they can be used.

Work-items and work-groups

Have a look at the code in Tutorial5a.

First, have a look at the kernel code in the 'workitem_id.cl' file. A kernel function must be defined using the `__kernel` keyword, with the return type as `void`. The kernel accepts an input and an output array (note that kernels can only return data through global arguments).

```
__kernel void workitem_id(__global int* input,
                          __global int* output) {
```

In the next few lines, the kernel uses a number of OpenCL built-in kernel functions to obtain work-item and work-group information. The argument that is passed to these functions is the dimension that is being queried. In this example, the values are all 0 because the program only uses 1-dimension.

```
   int global_id = get_global_id(0);
   int offset = get_global_offset(0);
   int global_size = get_global_size(0);
```

The first line gets the work-item's global ID. Each work-item has a unique global ID to identify itself in the solution index space.

The second line gets the offset value in the first dimension. Global IDs will start from this offset value.

The third line gets the number of work-items that were created when the kernel was launched.

You will remember from lectures that a kernel is a "blueprint" for work-items. Whereas work-items are the instantiation of a kernel. Many work-items can be created from a kernel. The offset and the number of work-items that will be created is set in host code when enqueueing the kernel for execution using the `enqueueNDRangeKernel` function. Since this is set in host code, the kernel does not have this information. As such, the built-in functions shown above are the mechanism that OpenCL provides for each work-item to be able to obtain this information.

The next few lines demonstrate a number of similar built-in functions.

```
int local_id = get_local_id(0);
int group_id = get_group_id(0);
int num_groups = get_num_groups(0);
int group_size = get_local_size(0);
```

The work-items are organised into work-groups. Each work-group has a unique work-group ID. Within a work-group, work-items have a local ID to identify itself within the work-group. While the global ID is unique for each work-item, i.e. there will not be any duplicate global IDs, work-items in *different* work-groups can have the same local ID. You will see this later when you run the program.

Hence, the `get_local_id` function returns a work-item's local ID. The `get_group_id` function returns the ID of the work-group that the work-item is executing in. The `get_num_groups` function returns the total number of work-groups, and the `get_local_size` function returns the number of work-items per work-group.

The next few lines of code checks whether the work-item is the first work-item (i.e. if the work-item's global ID is the same as the offset). This means that only one work-item will run the code in the body of the `if` statement, which sets the output array to some of the values obtained from the code above.

```
if(global_id == offset)
{
   output[0] = global_size;
   output[1] = offset;
   output[2] = num_groups;
   output[3] = group_size;
}
```

Note that the code will still work, if you omit the `if` statement since this information in this particular example is the same for all work-items. The difference is that without the `if` statement, *all* work-items will execute the code. Each work-item will simply overwrite values written by other work-items. In this example, this is fine because those values are the same for all work-items.

```
int index = (global_id - offset + 1) * 4;

output[index] = global_id;
output[index+1] = group_id;
output[index+2] = local_id;
output[index+3] = input[global_id];
```

The last few lines of code will be executed by each work-item. The index for the output array is computed based on the global ID and offset values. This is so that each work-item will only write to its respective output data space, and will not overwrite values written by other work-items. Also note that the last value outputted, i.e. `input[global_id]`, takes a value from the input array. The index used to lookup the input array is the work-item's global ID. This illustrates how each work-item can refer to its respective input data space.

Now have a look at the host code. If you have done the previous tutorials, then most of the code should be familiar. Hence, we will only look at some of the important sections.

Near the top, there are a number of values that have been defined.

```
#define OFFSET 8
#define GLOBAL_SIZE 16
#define LOCAL_SIZE 4

#define OUTPUT_SIZE (GLOBAL_SIZE + 1) * 4
```

These will be used to allocate memory and for launching the kernel.

In the main() function, an input and output vector, and their corresponding memory objects, are declared. The contents of the input vector are initialised from 1000 to 1999.

```
	std::vector<cl_int> inputVec(1000);
	std::vector<cl_int> outputVec(OUTPUT_SIZE);

	cl::Buffer inputBuffer, outputBuffer;

	// initialise values
	for (int i = 0; i < 1000; i++)
	{
		inputVec[i] = 1000 + i;
	}
```

The memory objects for input and output are created.

```
	inputBuffer = cl::Buffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
		sizeof(cl_int) * inputVec.size(), &inputVec[0]);
	outputBuffer = cl::Buffer(context, CL_MEM_WRITE_ONLY,
		sizeof(cl_int) * outputVec.size());
```

Kernel argument are set to the respective memory object.

```
	kernel.setArg(0, inputBuffer);
	kernel.setArg(1, outputBuffer);
```

Next, enqueue the kernel with the values defined near the top of the code.

```
	cl::NDRange offset(OFFSET);
	cl::NDRange globalSize(GLOBAL_SIZE);
	cl::NDRange localSize(LOCAL_SIZE);

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

The code after this reads and displays the results.

Compile and run the code.

Check whether the output matches your understanding of how the code works based on these values:

```
#define OFFSET 8
#define GLOBAL_SIZE 16
#define LOCAL_SIZE 4
```

Note that it is recommended that the work-group size (i.e. local size: the number of work-items per work-group) be a multiple of 2 (i.e. 2^n) for memory alignment. Also, the number of processing elements per compute unit is typically a multiple of 2. In fact, some platforms will not allow work-group sizes that are not a multiple of 2. Global size should be a multiple of the work-group size.

In this example:

- The offset was set to 8, the number of work-items to create is 16. Hence, there will be 16 work-items with their global IDs going from 8 to 23.
- The number of work-items per work-group was set to 4. Hence, the local IDs are between 0 to 3 for each work-group.
- Since there each work-group has 4 work-items, there will be 16/4 = 4 work-groups. So the work-group IDs will range between 0 and 3.

You can try modifying the `OFFSET`, `GLOBAL_SIZE`, and `LOCAL_SIZE` values to see how the output will change based on values that you set.

Memory address spaces

This section deals with the code in Tutorial5b.

Note that due to limitations on Apple's implementation of OpenCL, this example will not work properly using a **CPU on a Mac**. If using a Mac, please run it on a GPU.

The OpenCL memory model defines a hierarchy of four memory spaces: global, constant, local and private. Each memory space has its own characteristics. This section shows an example that demonstrates these spaces and how to use them.

First, have a look at the `memory_spaces` kernel in the 'memory.cl' file.

```
__kernel void memory_spaces(__private int num1,
                            float num2,
                            __local float* shared,
                            __constant float* input,
                            __global int* output1,
                            __global float* output2) {
```

Note the parameters of the kernel.

- `num1` is a private integer, defined using the `__private` address space qualifier
- `num2` is a private floating point value. Without using an address space qualifer, it defaults to private.
- `shared` is a pointer to local memory, defined using the `__local` address space qualifier
- `input` is a pointer to constant memory, defined using the `__constant` address space qualifier
- `output1` and `output2` are pointers to global memory, defined using the `__global` address space qualifier. As previously mentioned, this is the only way to output data from a kernel.

Next a number of private variables are declared within the kernel. These use some of the built-in functions described in the previous section. The `__private` keyword can be omitted since the default is private.

```
__private int global_id = get_global_id(0);
int local_id = get_local_id(0);
int group_id = get_group_id(0);
int group_size = get_local_size(0);
```

The next line of code copies the content from global memory into local memory. Note that each work-item accesses its respective input data using its global ID, and stores the content in local memory using each work-item's local ID. Local memory is shared among all work-items within the same work-group.

```
shared[local_id] = input[global_id];
```

The following line synchronises between all work-items within the same work-group. It blocks subsequent code from executing until all work-items have arrived at the barrier. This is to ensure that all work-items have finished coping the data from global memory to local memory. Otherwise, if the data is used before some work-items have finished, the result will not be correct.

```
barrier(CLK_LOCAL_MEM_FENCE);
```

Finally, the `if` statement checks whether the work-item's local ID is 0. In other words, we only want the first work-item in each work-group to execute the code in the body. It sums the values in local memory (which can be accessed by all work-items within the same work-group), adds the value of `num2`, then puts this value in the output array. Note that each work-group will have a different result. Hence, the position in the output array is indexed using the work-group's ID.

```
if(local_id == 0)
{
      for(int i = 1; i < group_size; i++)
      {
         shared[0] += shared[i];
      }

   output2[group_id] = shared[0] + num2;
}
```

Looking at the host code, data and memory objects are declared.

```
std::vector<cl_float> inputVec(1000);
std::vector<cl_int> outputVec1(GLOBAL_SIZE);
std::vector<cl_float> outputVec2(NUM_WORK_GROUPS);

cl::Buffer inputBuffer, outputBuffer1, outputBuffer2;

// initialise values
for (int i = 0; i < 1000; i++)
{
       inputVec[i] = 0.1f * i;
}
```

Note the section of code before the memory objects are created. This is for Apple computers, to prevent an invalid work-group size error when running the code on a CPU. The code checks whether the device will create a work-group size of 1 for a given kernel (which will ignore local size in enqueueNDRangeKernel command).

```
#ifdef __APPLE__
       // MacOS: cannot run kernel using CPU
       size_t kernelWorkgroupSize =
              kernel.getWorkGroupInfo<CL_KERNEL_WORK_GROUP_SIZE>(device);

       // abort if kernel only allows one work-item per work-group
       if (kernelWorkgroupSize == 1)
              quit_program("Abort: Cannot run kernel, because kernel workgroup size is 1.");
#endif
```

Memory objects are then created as follows.

```
       inputBuffer = cl::Buffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
              sizeof(cl_float) * inputVec.size(), &inputVec[0]);
       outputBuffer1 = cl::Buffer(context, CL_MEM_WRITE_ONLY,
              sizeof(cl_int) * outputVec1.size());
       outputBuffer2 = cl::Buffer(context, CL_MEM_WRITE_ONLY,
              sizeof(cl_float) * outputVec2.size());
```

This is followed by setting the kernel arguments. Note that because the arguments at index 0 and 1 are private, they can only be primitive datatypes. The argument at index 2 is uses local memory, the code allocates a certain amount of space for it. The host cannot access local memory (i.e. cannot read/write to it), but can allocate memory for it.

```
       int a = 101;
       float b = 11.22;
       cl::LocalSpaceArg localSpace = cl::Local(sizeof(float) * LOCAL_SIZE);

       kernel.setArg(0, a);
       kernel.setArg(1, b);
       kernel.setArg(2, localSpace);
       kernel.setArg(3, inputBuffer);
       kernel.setArg(4, outputBuffer1);
       kernel.setArg(5, outputBuffer2);
```

The kernel is then enqueued for execution.

```
       cl::NDRange offset(0);
       cl::NDRange globalSize(GLOBAL_SIZE);
       cl::NDRange localSize(LOCAL_SIZE);

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

The results read by the host and displayed.

```
       queue.enqueueReadBuffer(outputBuffer1, CL_TRUE, 0, sizeof(cl_int) *
              outputVec1.size(), &outputVec1[0]);
       queue.enqueueReadBuffer(outputBuffer2, CL_TRUE, 0, sizeof(cl_float) *
              outputVec2.size(), &outputVec2[0]);
```

The displayed results will show

- The output data for each work-item.
- The result of the calculation for each work-group.

Vector data types

Tutorial5c demonstrates basic operations using vectors. Note that many operations on vector data types are performed on all components of the vector. The main highlights of this example are in the kernel.

First, a vector is declared an its contents are initialised:

```
int4 vec = (int4)(1, 2, 3, 4);
```

Next, the value of 4 is added to each element in the vector using:

```
vec += 4;
```

The content of the vector after this operation is (5, 6, 7, 8). The next line check whether element 3 is 7. If it is, bitwise AND the vector with another vector.

```
if(vec.s2 == 7)
   vec &= (int4)(-1, -1, 0, -1);
```

The content of the vector is now (5, 6, 0, 8). The next line replaces the first and second elements of the vector with the result of whether the third and fourth elements of the vector are less than 7. Note that for vectors TRUE is -1, and FALSE is 0.

```
vec.s01 = vec.s23 < 7;
```

The vector will now contain the values (-1, 0, 0, 8). The next couple of lines will divide the fourth element in the vector by 2 until it is less than or equal to 7:

```
while(vec.s3 > 7 && (vec.s0 < 16 || vec.s1 < 16))
   vec.s3 >>= 1;
```

Finally, the content of the vector is written to the output array. Note that the array is of type int4.

```
output[0] = vec;
```

Exercise

To better understand the concepts in this tutorial, try the following exercise:

Write an OpenCL program that uses a kernel (write the kernel code yourself) to fill in the contents of an output array in parallel. One of the kernel inputs is to be a number between 1 and 100. The kernel is to return the following results based on the value of the number:

- If the number is 1, the resulting contents of the array should be: 1, 2, 3, 4, 5,… until 1024
- If the number is 2, the resulting contents of the array should be: 1, 3, 5, 7, 9,… until 2047
- If the number is 3, the resulting contents of the array should be: 1, 4, 7, 10, 13,… until 3070
- …
- If the number is 100, the resulting contents of the array should be: 1, 101, 201, 301, 401,… until 102301

After kernel execution, display the resulting contents on screen.

References

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

- M. Scarpino, “OpenCL in Action: How to Accelerate Graphics and Computation,” Manning