SCIT
School of Computing & Information Technology

CSCI376 – Multicore and GPU Programming

Kernel Programming

The aim of this tutorial is to introduce some OpenCL built-in functions and operations on vectors. The programs in this tutorial are based on the content of this week's lecture and should be examined in conjunction with the lecture notes.

Enqueueing a kernel in two dimensions

Have a look at the code in Tutorial6a. This program shows a 2-dimensional example of using the OpenCL built-in functions to obtain work-item and work-group information from within the kernel. This allows work-items to determine their individual location in the index space.

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

In the host code, pay attention to the values that are passed when the kernel is enqueued on the command queue.

```
cl::NDRange offset(3, 5);
cl::NDRange globalSize(6, 4);
cl::NDRange localSize(3, 2);

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

Notice that this enqueues the kernel in two dimensions.

In the kernel code, the following lines get a work-item's global IDs in dimension 0 and dimension 1, respectively:

```
size_t global_id_0 = get_global_id(0);
size_t global_id_1 = get_global_id(1);
```

This gets the number of work-items that were created in dimension 0:

```
size_t global_size_0 = get_global_size(0);
```

The next lines get the starting global ID offsets in dimension 0 and dimension 1, respectively:

```
size_t offset_0 = get_global_offset(0);
size_t offset_1 = get_global_offset(1);
```

Then, these get a work-item's local IDs in dimension 0 and dimension 1, respectively, within its work-group:

```
size_t local_id_0 = get_local_id(0);
size_t local_id_1 = get_local_id(1);
```

The information obtained within the kernel above can be used to calculate a 1- dimensional output array index.

```
int index_0 = global_id_0 - offset_0;
int index_1 = global_id_1 - offset_1;
int index = index_1 * global_size_0 + index_0;
```

This index will be used for each work-item to store its result in the output array:

```
output[index] = f;
```

Notice that while the problem is solved in 2-dimensions, the result is written into a 1-dimensional output array.

The result for each work-item is simply its global IDs and local IDs.

```
float f = global_id_0 * 10.0f + global_id_1 * 1.0f;
f += local_id_0 * 0.1f + local_id_1 * 0.01f;
```

Compile and run the code. Check whether the output matches your understanding of global and local IDs.

OpenCL vector data types

The example in Tutorial6b demonstrates the use of OpenCL vector data types.

In the host code, two input vectors and an output vector is created by the host:

```
std::vector<cl_int> inputVec1(16);
std::vector<cl_int> inputVec2(16);
std::vector<cl_int> results(32);
```

The first input vector is initialised with values from 0 to 15, while the second input vector is initialised with values from -1 to 14.

```
for (int i = 0; i < inputVec1.size(); i++)
{
       inputVec1[i] = i;
       inputVec2[i] = i - 1;
       results[i] = -1;
}
```

Their respective buffer objects are created and associated with kernel arguments, before the kernel is enqueued. The global size is set to 4, so 4 work-items will be created.

Now have a look at the code in the kernel. The kernel accepts a scalar input array, an input array of vectors and a scalar output array.

```
__kernel void vectors(__global int* array,
                      __global int4* vec,
                      __global int* output) {
```

The work-item's global id is obtained and a number of vectors are declared:

```
int global_id = get_global_id(0);
int4 temp1, temp2;
int8 temp3;
```

Next, each work-item loads 4-elements from the input array into an int4 vector using the vload*n* function:

```
temp1 = vload4(global_id, array);
```

The content of `temp1` for the respective work-item is as follows:

Work-item 1: (0, 1, 2, 3)

Work-item 2: (4, 5, 6, 7)

Work-item 3: (8, 9, 10, 11)

Work-item 4: (12, 13, 14, 15)

Then, each work-item gets a 4-element vector from the array of input vectors:

```
temp2 = vec[global_id];
```

The content of `temp2` for the respective work-item is as follows:

Work-item 1: (-1, 0, 1, 2)

Work-item 2: (3, 4, 5, 6)

Work-item 3: (7, 8, 9, 10)

Work-item 4: (11, 12, 13, 14)

These contents are added with a value from the input array:

```
temp2 += array[global_id];
```

After the +=operation, the content of `temp2` for the respective work-item is now:

Work-item 1: (-1, 0, 1, 2)

Work-item 2: (4, 5, 6, 7)

Work-item 3: (9, 10, 11, 12)

Work-item 4: (14, 15, 16, 17)

The next section checks whether *all* elements in `temp2` are greater than elements in `temp1`. If so, set the content of `temp3` to the contents of `temp1` and `temp2`. Otherwise, set the content of temp3 to the contents of `temp1` and `temp2` in the reversed order:

```
if(all(temp1 < temp2))
   temp3 = (int8)(temp1, temp2);
else
   temp3 = (int8)(temp1.s3210, temp2.s3210);
```

Finally, store the content of the 8-element vector in the scalar output array at each work-item's respective output position:

```
vstore8(temp3, global_id, output);
```

After the kernel execution, the host application reads the results and displays this on screen.

Note that in the kernel, the code can be changed to:

```
__kernel void vectors(__global int* array,
                      __global int4* vec,
                      __global int8* output) {
```

Then replace:

```
   vstore8(temp3, global_id, output);
```

With:

```
   output[global_id] = temp3;
```

Shuffle and select

The shuffle and select functions fill an output vector with content from one or two input vectors, based on values in a mask vector. The code in Tutorial6c shows examples using these functions.

The host code is straightforward and should be familiar. It sets up the data and buffer objects, sets the kernel arguments and enqueues the kernel. Note that only 1 work-item is created. After the kernel has executed, the results are read and displayed on screen.

The kernel code shows examples of defining normal functions (i.e. non-kernel functions) in an OpenCL program, where these functions can be called by a kernel.

The following is an example of a function that accepts OpenCL vector data types and also returns a vector:

```
float8 shuffle_test(uint8 mask, float4 input)
{
    return shuffle(input, mask);
}
```

It uses the built-in OpenCL shuffle function to fill the content of an output vector. To do this, the shuffle function uses the content of a mask vector as indexes of an input vector.

The `shuffle_test()` function can be called from within the kernel:

```
   uint8 mask = (uint8)(0, 1, 2, 3, 1, 3, 0, 2);
   float4 input = (float4)(0.25f, 0.5f, 0.75f, 1.0f);
   *s1 = shuffle_test(mask, input);
```

The next example demonstrates the shuffle2 function. It works very similar to the shuffle function, where the difference is that the shuffle2 function accepts two input vectors instead of one input vector.

```
char16 shuffle2_test()
{
   // initialise mask and input vectors
   uchar16 mask = (uchar16)(6, 10, 5, 2, 8, 0, 9, 14, 7, 5, 12, 3, 11, 15, 1, 13);
   char8 input1 = (char8)('l', 'o', 'f', 'c', 'a', 'u', 's', 'f');
   char8 input2 = (char8)('f', 'e', 'h', 't', 'n', 'n', '2', 'i');

   return shuffle2(input1, input2, mask);
}
```

The next two examples demonstrate the select and bitselect functions. These select functions use bits in a mask vector to select elements from either the first input vector or from the second input vector.

The select function selects the corresponding element from the first input vector if the most significant bit (MSB) in the mask is 0, whereas it selects the corresponding element from the second input vector if the MSB in the mask is 1.

In the following code, the first element of the output vector will be the first element of the second input vector. This is because the MSB of the first element of the mask is 1. The MSB of the second element of the mask is 0, hence, the second element of the output vector will be the second element of the first input vector. The MSB of the third element of the mask is 1, hence, the third element of the output vector will be the third element of the second input vector. Finally, the fourth element of the output vector will be the fourth element of the first input vector, because the MSB of the fourth element of the mask is 0.

```
float4 select_test()
{
   // initialise mask and input vectors
   int4 mask = (int4)(-1, 0, -1, 0);
   float4 input1 = (float4)(0.25f, 0.5f, 0.75f, 1.0f);
   float4 input2 = (float4)(1.25f, 1.5f, 1.75f, 2.0f);

   return select(input1, input2, mask);
}
```

The difference between the select function and the bitselect function, is that the bitselect function uses corresponding bits in the mask to select bits from either the first or the second input vectors. If a bit in the mask is 0, the corresponding bit in the output vector will be the bit from the first input vector. Otherwise, the corresponding bit in the output vector will be the bit from the second input vector.

```
uchar2 bitselect_test()
{
   // initialise mask and input vectors
   uchar2 mask = (uchar2)(0xAA, 0x55);     // values in bits: 1010 1010 0101 0101
   uchar2 input1 = (uchar2)(0x0F, 0x0F);   // values in bits: 0000 1111 0000 1111
   uchar2 input2 = (uchar2)(0x33, 0x33);   // values in bits: 0011 0011 0011 0011

   return bitselect(input1, input2, mask); // result in bits: 0010 0111 0001 1011
}
```

Exercise

To gain a better understanding of the code, try changing values in the code and figure out how the changes will affect the output.

Start working on Assignment 2. The assignment is based on the OpenCL concepts covered in the lectures and tutorials.

References

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

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