SCIT

School of Computing & Information Technology

CSCI376 – Multicore and GPU Programming

Parallel Image Processing

The aim of this tutorial is to introduce you to OpenCL image objects and functions. Refer to the lecture notes on parallel image processing for further details.

Reading and writing to BMP image files

To start, have a look at the code in bmpfuncs.cpp and bmpfuncs.h, which are located in the Tutorial8a folder. The code contains two functions.

The first will read the contents of a 24-bit colour bitmap image, and load it into an unsigned char array in RGBA format:

```
unsigned char* read_BMP_RGB_to_RGBA(const char *filename, int* widthOut, int* heightOut);
```

The second accepts image data in RGBA format and write the contents into a 24-bit colour bitmap image:

```
void write_BMP_RGBA_to_RGB(const char *filename, unsigned char* imageData, int width, int
height);
```

Reading from and writing to image objects in the kernel

The program in Tutorial8a is a basic program to load the contents of an image and pass this to the kernel. Each work-item will read from a single pixel and write this into the output image. The host program will read the contents of the output image and save it into an image file.

Examine the code in Tutorial8a.cpp and simple_image.cl.

On the host, the image data is stored in unsigned char arrays:

```
	unsigned char* inputImage;
	unsigned char* outputImage;
```

Image dimensions are stored in:

```
	int imgWidth, imgHeight, imageSize;
```

Image format and image objects are also declared:

```
	cl::ImageFormat imgFormat;
	cl::Image2D inputImgBuffer, outputImgBuffer;
```

The contents of a 24-bit RGB colour bmp image is read using the function described in the previous section, the width and height values are stored and memory is allocated for the output image. Note that the image that will be returned from the function is in RGBA format, and the output image will also be in RGBA format, hence its size is width * height * 4. The size is multiplied by 4 for the Red, Green, Blue and Alpha colour channels, which are 8-bits each.

```
inputImage = read_BMP_RGB_to_RGBA("lena.bmp", &imgWidth, &imgHeight);


imageSize = imgWidth * imgHeight * 4;
outputImage = new unsigned char[imageSize];
```

The image format is declared and initialized to `CL_RGBA` and the channel data type is set to `CL_UNORM_INT8`, so we will be dealing with colour values from 0.0 to 1.0 (0.0 is no intensity and 1.0 is maximum intensity for the respective colour channels) in the kernel. Note that the colour values range from 0 to 255 on the host (unsigned char), and 0.0 to 1.0 on the device (float).

```
imgFormat = cl::ImageFormat(CL_RGBA, CL_UNORM_INT8);
```

Next, the input and output 2D image objects need to be create and set as the kernel arguments:

```
inputImgBuffer = cl::Image2D(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
       imgFormat, imgWidth, imgHeight, 0, (void*)inputImage);
outputImgBuffer = cl::Image2D(context, CL_MEM_WRITE_ONLY | CL_MEM_COPY_HOST_PTR,
       imgFormat, imgWidth, imgHeight, 0, (void*)outputImage);


kernel.setArg(0, inputImgBuffer);
kernel.setArg(1, outputImgBuffer);
```

Then the kernel is enqueued. Note that this is a 2D kernel based on the width and height of the image, because the total number of work-items is one work-item per pixel.

```
cl::NDRange offset(0, 0);
cl::NDRange globalSize(imgWidth, imgHeight);

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

Once the computation is complete, read the images' contents back from the device to the host based on the dimensions of the output image:

```
cl::size_t<3> origin, region;
origin[0] = origin[1] = origin[2] = 0;
region[0] = imgWidth;
region[1] = imgHeight;
region[2] = 1;

queue.enqueueReadImage(outputImgBuffer, CL_TRUE, origin, region, 0, 0, outputImage);
```

Finally, write the contents of the output image to a bmp image using the `write_BMP_RGBA_to_RGB()` function. Note that this function accepts an input file, because it will copy the header format from the input file. In this example, it copies the header for a 24-bit RGB colour image.

```
write_BMP_RGBA_to_RGB("output.bmp", outputImage, imgWidth, imgHeight);
```

In the kernel function, a sampler is declared and initialized in constant memory.

```
__constant sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE |
      CLK_ADDRESS_CLAMP | CLK_FILTER_NEAREST;
```

The kernel accepts and input and output image. Each work-item calculates a pixel coordinate based on the 2 dimensional global IDs:

```
int2 coord = (int2)(get_global_id(0), get_global_id(1));
```

Using the coordinate and the sampler, the pixel is read from the image:

```
float4 pixel = read_imagef(src_image, sampler, coord);
```

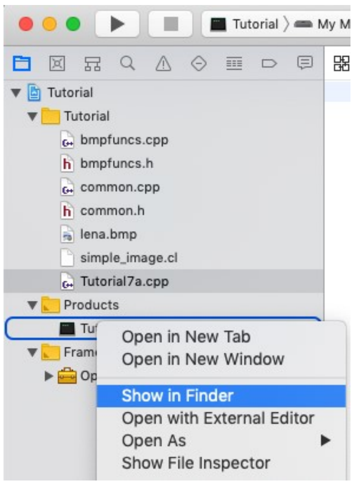
The pixel is then written into the output image at the exact some coordinate location:

```
write_imagef(dst_image, coord, pixel);
```

Compile and run the program.

In Visual Studio, the output.bmp image will be in the same folder where your code and input image is located.

In Xcode, open the following location in a finder. The output.bmp image will be in that folder.



Gradient image

The next example in the Tutorial8b folder shows simple colour manipulation in the kernel. Look at the code in Tutorial8b.cpp and gradient.cl. Note that the code in Tutorial8b.cpp is almost identical to Tutorial8a.cpp, only the input file name has changed white.bmp (which is an image with white pixels).

Only one line is added to the kernel:

```
pixel.xyz -= (float)coord.x/get_image_width(src_image);
```

The input image is a 24-bit colour image and the channel data type is set to `CL_UNORM_INT8`, so we will be dealing with colour values from 0.0 to 1.0 (if RGB are all 0.0, the colour is black, whereas if RGB are all 1.0, the colour is white). With the colour channels, the `float4` that will be returned will be (R, G, B, A), so we need to manipulate the RGB colour values i.e. `pixel.xyz`. Note that we ignore the A channel since we will not be using it. To change the gradient from white to black, we divide the coordinate (which is based on the work-item's global ID) by the image width this will produce a value from 0.0 to 1.0 for RGB respectively. Hence the final pixel value will gradually change from 1.0 to 0.0 for RGB respectively. This is then written into the output image.

Rotating an image

The example in Tutorial8c shows how to manipulate images by reading the value from a different pixel location and writing it to the destination pixel location. The code in Tutorial8c.cpp is very similar to Tutorial8a.cpp. The main difference is that the sin and cos of the rotation angle are calculated and passed to the kernel's private memory:

```
int theta = 45;
cl_float sin_theta = sinf(theta);
cl_float cos_theta = cosf(theta);
```

and

```
kernel.setArg(2, sin_theta);
kernel.setArg(3, cos_theta);
```

In the kernel, given a pixel location the kernel calculates a rotated coordinate and read a pixel value from this rotated coordinate. It then writes this value in the output image.

Image convolution

Image convolutions are commonly used in image processing. The example in Tutorial8d shows an example of applying a simple 3x3 convolution filter to an image. We will look at the code in the kernel as most things in the host program are the same.

In the kernel program, three different filters are declared and initialised (there are many other filters that can be used). The first thing the kernel does is to get the coordinates of the pixel from the global IDs:

```
int column = get_global_id(0);
int row = get_global_id(1);
```

Next, a number of variables are declared:

```
float4 sum = (float4)(0.0);   // This is to store the accumulated weighted sum
int filter_index =  0;        // To store the filter's array index
int2 coord;                   // To calculated pixel coordinates
float4 pixel;                 // This is to store the current pixel value
```

The next section consists of two for loops; the first to iterate over the rows and the second to iterate over the columns of the convolution. The values of the pixels at those locations are multiplied with the values from the filter and `sum` stores all the accumulated values. Note that in the following example, the `SharpeningFilter` is used, you can change this to use the `BlurringFilter` or the `vSobelFilter` to apply different filters to the image.

```
for(int i = -1; i <= 1; i++) {
     coord.y =  row + i;
```

```
        for(int j = -1; j <= 1; j++) {
            coord.x = column + j;

             /* Read value pixel from the image */
             pixel = read_imagef(src_image, sampler, coord);
             /* Acculumate weighted sum */
             sum.xyz += pixel.xyz * SharpeningFilter[filter_index++];
        }
   }
```

After this, the output image coordinates are defined and the pixel is written into the output image.

```
   coord = (int2)(column, row);
   write_imagef(dst_image, coord, sum);
```

Things to try

Try writing a program to do the following:

- Flip an image in the
    - Horizontal dimension (i.e. left become right, and vice versa)
    - Vertical dimension (i.e. top becomes bottom, and vice versa)
    - Both horizontal and vertical dimensions

References

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

- M. Scarpino, "OpenCL in Action: How to Accelerate Graphics and Computation," Manning Publications
- Gaster, B.R., Howes, L., Kaeli, D.R., Mistry, P. and Schaa, D., "Heterogeneous Computing with OpenCL," Morgan Kaufmann