**OpenCL, assignment 2**

# Introduction

In the previous assignment, you (hopefully) got OpenCL working on your system.

Since many systems have multiple OpenCL platforms aboard, it is important to choose the right one for your experiments. Therefore, we now first run a small program that will give an overview of the available OpenCL platforms on your system.

* Make a new empty OpenCL project (maybe the easiest way is to make a copy of a working project of week 1). Download the files CLGetPlatforms.cpp , opencl\_utils.cpp and opencl\_utils.h from SharePoint and add them to the project.
* Run the project. You should see all your OpenCL platforms listed, including their number (numbering starts from 0).
* Select the one you want to use (probably nvidia, AMD, or Intel) and remember its number.

**Make sure to use this platform in all your following OpenCL projects (in the call to clGetDeviceIDs).**

# About the Mandelbrot algorithm

This assignment is about the Mandelbrot fractal. There are many sites that explain the Mandelbrot algorithm, like <http://www.christineweb.be/VoorbeeldenVB/VBVBMandel.html> (in Dutch), or Wikipedia.

Fully understanding the Mandelbrot algorithm is not necessary for this assignment. The important thing are:

* Each pixel can be calculated on its own; in the calculation for one pixel, no information about any other pixel is needed.
* Calculating a pixel can take quite a lot of time.
* The application uses a color table, which is just a list of colors in BGR format.

|  |  |  |  |
| --- | --- | --- | --- |
|  | B | G | R |
| 0 | 0 | 0 | 20 |
| 1 | 0 | 0 | 25 |
| 2 | 0 | 0 | 30 |
| 3 | 0 | 0 | 35 |
| 4 | 0 | 0 | 40 |
| … | … | … | … |

For each pixel, at the end of the calculation we get one int value. This value is used as an index into the color table. The color on that index defines the color the pixel gets.   
For example: if the calculation for a certain pixel produces value 3, then the color of that pixel will be (0,0,35). To see which color that is, you could use MS Paint, button “Edit colors”.

# Tasks

* Open the project CPUMandelbrot, which can be found on SharePoint.   
  This program generates a bitmap file containing the fractal, and it also shows the time that the CPU needed to do the calculations.
* Run it to see what output it produces.  
  Note: this is not an OpenCL program; all calculations are done on the CPU.
* In the code, a number of constants are defined that can be used to shift the fractal, to zoom in etc: WIDTH, HEIGHT, OFFSET\_X, OFFSET\_Y, ZOOMFACTOR, MAX\_ITERATIONS, and COLORTABLE\_SIZE.  
  Try out a few different settings, and for each of these settings write down the reported calculation time, so we can compare it with the GPU implementation later.
* Now make an OpenCL implementation for the same Mandelbrot fractal. You can use the example of last week as starting point. The slides of last week (part 1) give an example of a kernel, and the typical structure of OpenCL code (last slide). Also the program that you ran in the beginning of this assignment (CLGetPlatforms) gives you some example code.
* If it works, also add some code to measure the time for the kernel to finish, and the time to move the image from GPU to CPU.   
  Make sure that you wait until the kernel is actually finished by using the function clFinish.
* Compare the calculation times for CPU and GPU with the same setting (WIDTH, HEIGHT, ZOOMFACTOR, etc).   
  The GPU should be noticeable faster.
* Make the variables x, y, xSqr and ySqr local and compare the execution times.

# Hints and tips

Some hints to make the OpenCL implementation:

* Reuse the code from the function mandelbrot\_frame as much as possible in the kernel (of course, you must make sure that the pixels will be calculated in parallel, and not sequentially, so remove the for tsatements).
* Also reuse the create\_colortable function, but copy the color table to the GPU with a code like this:
  + // Create new table on device
  + dev\_colortable = clCreateBuffer(context, CL\_MEM\_READ\_ONLY,   
     COLORTABLE\_SIZE\*sizeof(mandelbrot\_color), NULL, &err);
  + checkError(err,"Couldn't create colortable on device");
  + // Write the color table data to the device
  + err = clEnqueueWriteBuffer(queue, dev\_colortable, CL\_TRUE, 0,  
     COLORTABLE\_SIZE\* sizeof(mandelbrot\_color),   
     colortable, 0, NULL, NULL);
  + checkError(err,"Couldn't write colortable to device");
* The function checkError prints the error in a fairly readable way, which is more convenient than just a number.   
  This function is defined in opencl\_utils.cpp/opencl\_utils.h, so add these files to your project.  
  You might also want to check out these sites for more information about errors: <http://www.techdarting.com/2014/01/opencl-errors.html> or <https://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/errors.html>.
* Use this way of error-checking after all your OpenCL calls, which probably are (in this order):
  + clGetPlatformIDs
  + clGetDeviceIDs
  + clCreateContext
  + build\_program (no error checking needed here)
  + clCreateCommandQueue
  + clCreateKernel
  + clCreateBuffer
  + clSetKernelArg (a few times)
  + clEnqueueNDRangeKernel
  + clEnqueueReadBuffer
* In opencl\_utils, you can also find the convenient function build\_program, which
  + reads the kernel code from a file,
  + calls clCreateProgramWithSource,
  + calls clBuildProgram, and
  + prints all (if there are any) compiler errors in the kernel.
* For measuring the time, you can either use a timer on the host (with QueryPerformanceCounter) or a timer on the device (with clGetEventProfilingInfo).
* Don’t forget that 1/200 does give the result 0 (integer division). You need to convert at least one of the operands to float to get a float result: (float)1/200.