Class COMP 426 – NN

Instructor: Professor R. Jayakumar

Student ID: 27739656

Name: Said-Mansour Maqsoudi

Assignment 3 documentation

Goal of the assignment

The goal of the assignment was to either use CUDA or OpenCL to port some of the computation that you previously performed on the CPU, to now be performing on the GPU. Because of the fact that I did not have an NVIDIA GPU, I had to opt for using OpenCL to accomplish the main goal. OpenCl requires that one installs an SDK to have the required libraries in the program. Because I had an AMD GPU I decided to install the OpenCL SDK provided by AMD.

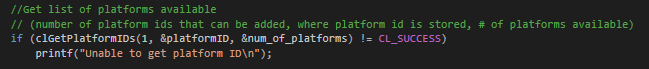
After including the necessary OpenCL libraries on visual studio, I had to decide where the code of the host program would reside. This is the code that will initialize the kernels and command queues as necessary to use threads on the GPU. Because my parallel medicine cell injections were performed by pressing the space bar, I decided that the host code will reside in a function that is called when the space bar is pressed. So the first thing I will try to replace is the TBB mechanism used in assignment 2. I first replace the TBB control thread in the keyboard function with a method that has the host code for OpenCL. So this time pressing the spacebar will call the injectMedicineCells() method that contains the host code for the OpenCL implementation.

The host code

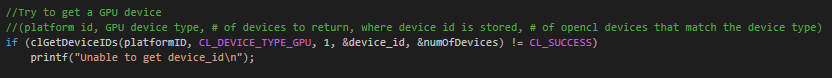
The host code is where a lot of important OpenCL commands will take place. First I created declarations for various OpenCL variables that will be needed when setting up the kernel commands.

OpenCL’s execution setup process

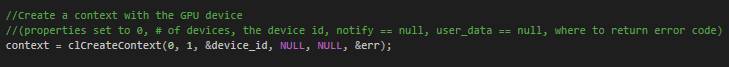
The first thing we have to do is to get the platform ID. The platform ID will help us to query for devices that are located on the machine. With OpenCl getting the platform ID can be done by using the function clGetPlatformIDs. For most of these steps I put these functions inside an “if statement” to check if the action was successful.



The next step was to try to get the machines GPU device. In this case we will use OpenCL’s clGetDeviceIDs() method. We will pass the platformID which was acquired in the previous step and set the second argument to “CL\_DEVICES\_TYPE\_GPU” in order to specifically get the GPU device.



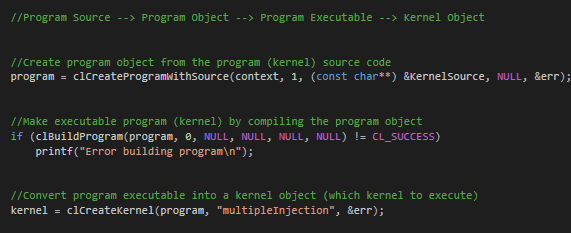
With the device ID acquired we can now create an OpenCL context. The OpenCL context is the environment where the OpenCL program runs. In OpenCL this can be created using the clCreateContext() function.



Once we have a context we can create a command queue and associate it with the context and the device ID. This will be used for queuing commands, particularly kernels, and certain events if necessary. OpenCL provides the clCreateCommandQueue() function to create the queue.

../../../Volumes/Untitled/images/command%2

The next steps involved taking the program source (kernel) turning it into a program object, compiling the program object and making it into a program executable and then converting the program executable into a kernel object. The entire process involves three OpenCL functions namely, clCreateProgramWithSource(), clBuildProgram(), and clCreateKernel(). Doing these steps gave us our kernel object.



Next I will create several buffers for the output of the kernel. In my case 8 buffers corresponding to the 8 directions that medicine cells travel in. This is done with the clCreateBuffer() function provided by OpenCL. Then I will use the clSetKernelArg() function to assign all of the arguments that the kernel will have. In this case, my kernel function will have 17 arguments. Eight of those arguments will be for the output buffers mention earlier. The rest of the parameters will be used for random number inputs to the kernel function, this because we cannot have the rand() function inside the kernel. These random numbers include random coordinates on the grid to inject medicine cells, as well as a random number of cells to inject.

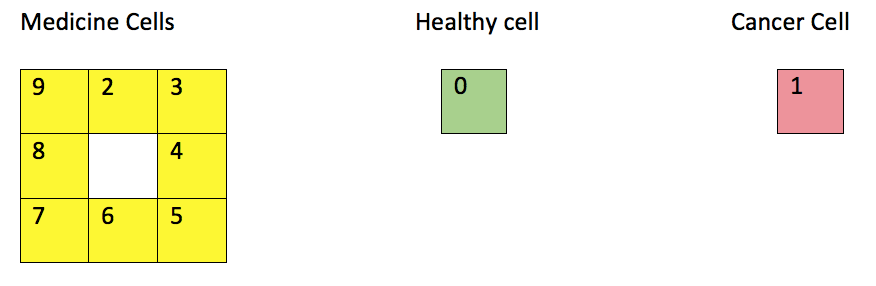
Next we have to enqueue the kernel into the command queue. To do this we use the function clEnqueueNDRangeKernel(), which includes as arguments the command queue and kernel object. After queueing the kernel, we can expect the kernel to execute immediately, so I included a function after this called clFinish(command\_queue) to wait for the completion of the command on the GPU. So after this function we know that the GPU is done computing.

Next I copy the results from the output buffer to the empty arrays I created earlier. Then I print a message to the console saying that the computation on the GPU is complete. From then I now have the new calculated coordinates of where the medicine cells should be located when injected. Then I simply update the main 2d array of cells with this information.

Finally I use the function clReleaseMemObject() on several objects to avoid memory leaks. This marks the end of the host code that will run on the CPU.

The kernel function

For the kernel function I originally planned to copy the entire 500x500 2D array as input for the kernel and then perform the calculations directly on the 2D array. However, doing it this way caused a lot of problems. Mainly the program would crash immediately when starting. I tried several ways to mitigate this, the only solution seemed to be making the 2D array much smaller, that is to a 10x10 2D array. However, for the purposes of the simulation I found that this was not acceptable. So I proceeded to try a different strategy for the kernel. Instead of using the entire 2D array as input, I would simply use the coordinates of injections as input. Doing so I could use the kernel to calculate the location of each medicine cell based on the original coordinate of injection. Then I would return these new coordinates in multiple arrays that will later be used to update the main 2D array. To explain why I needed to use multiple output arrays as buffers was because of the way my medicine cells worked. They are numbered 2 to 9 and each number perform its own specific movement. Ex: 2’s only move upwards. My strategy of medicine cells is illustrated below.



So for me having 8 output buffers for the medicine cells made things easier to manage.

Although this new kernel seemed to be a much better alternative to the original kernel, I still experience several issues with on occasion the screen going completely black. I believe some of these new issues have to do with the fact that I’m running fairly low end hardware.