COMP 426

Project

Design Documentation

Christopher Mukherjee

6291929

December 9, 2014

OpenCL Optimization

In this assignment, I took the code that was developed in Assignment 3 and optimized it by replacing the CUDA data parallelization with OpenCL data parallelization using the GPU. I also attempted to integrate OpenCL task parallelization using the CPU, but was unable to find any code that could and should be optimized using this method. Instead, I was able to find another section of code that could be data parallelized. This section was the Display function, which handles the displaying of the cells to screen. However, I was unable to successfully implement the necessary OpenGL calls from within the OpenCL kernel code, so I had to abandon this optimization strategy. Lastly, though I was not able to find a place to successfully use the CPU kernel, I left the setup for the CPU kernel in my code. Therefore, my code implements both a GPU kernel and a CPU kernel, but the CPU kernel remains unused.

Architecture

The entirety of the program is contained in one C++ file (.cpp). This file consists of a main function, 5 GLUT-related helper functions, 1 general helper function to begin the updating of the cells, 2 OpenCL helper functions, 1 OpenCL GPU kernel function, and 1 OpenCL CPU kernel function. The individual functions are:

**main**: This function is called when the program is initially run. It represents the main control thread of the application.

**Initialize**: Initialization function for GLUT.

**Keyboard**: GLUT-related function that handles keyboard buttons being pressed.

**MouseClicks**: GLUT-related function that handles mouse buttons being clicked.

**RenderBitmapString**: GLUT-related function that renders bitmap strings to display text on screen

**Display**: Function that displays the cells and text in a window on screen using GLUT

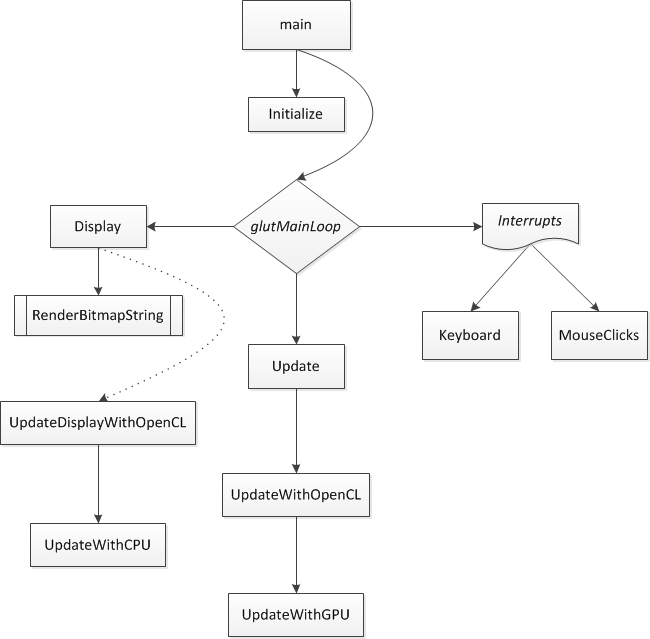
**Update**: Function that uses OpenCL to update the cells in parallel. At the end of the function, the function calls itself (to update again).

**UpdateWithOpenCL**: Helper function for using OpenCL to update cells in parallel. Launches the OpenCL GPU kernel.

**UpdateDisplayWithOpenCL**: Helper function for using OpenCL to update the display of the simulation in parallel. Launches the OpenCL CPU kernel. (Unused)

**UpdateWithGPU**: Updates each cell state using the GPU kernel.

**UpdateWithCPU**: Updates the display of the simulation using the CPU kernel. (Unused)

Control Flow Diagram:

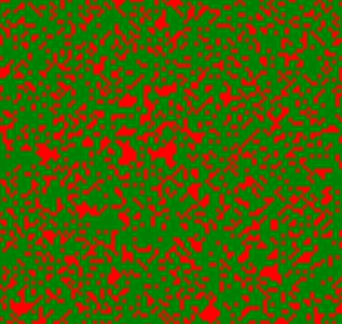
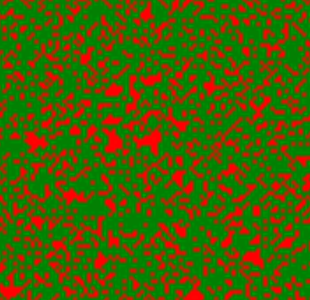
The OpenCL GPU kernel is used in order to parallelize the updating of the cells. Using the GPU kernel, the program can take advantage of the hundreds of threads available, instead of restricting itself to the threads available on the CPU. I developed this program on my MacBook Pro computer using Xcode in Mac OS X. My MacBook has a quad-core Intel Core i7 processor with hyper-threading. Thus, the CPU can run a total of 8 threads in parallel (one of which is taken up by the OS itself). It also has an AMD Radeon HD 6490M graphics processor, which is both OpenGL and OpenCL compatible. The GPU has hundreds of threads. Thus, as stated above, the GPU kernel can take advantage of these hundreds of threads, instead of restricting itself to only 7 threads.

The 2-dimensional area of 1024 x 768 cells is represented by one global 2-dimensional array. This array is then copied into two 1-dimensional OpenCL device memory objects. This way I can use one array in device memory to read from, and one array in device memory to write to. Therefore during each update in the OpenCL kernel, all calculations are performed on a static array, instead of a dynamically changing one. The values of the write array in device memory are then updated to the global array before the next call to the OpenCL update functions. The reason I used 1-dimensional OpenCL device memory objects is in order to simplify the data structures so that they can be more easily worked with in the GPU kernel. Lastly, the reason I used array data structures is because the size of the problem was already known. This means that resizing (which would be a problem for an array) would never be needed. Therefore, all I would need to do is change the values of specific cells, which is easy to do with an array.

Test Scenarios

**Test Scenario #1:** Directly healing a single cancer cell

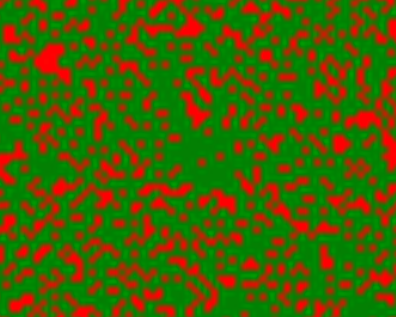
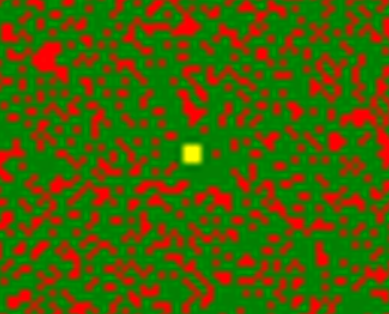
**User Input:** Click on a red cancer cell.

**Expected Result:** Medicine is injected into the cancer cell and is instantly absorbed. The cancer cell is healed and turns green.

**Test Scenario #2:** Injecting medicine into a healthy cell

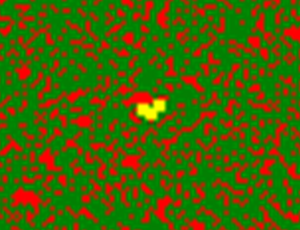
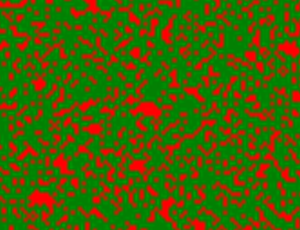
**User Input:** Click on a green healthy cell.

**Expected Result:** Medicine is injected into the healthy cell and the cell becomes yellow. The medicine is not absorbed and moves radially outwards by one cell position in each direction. Note that all the surrounding cells turn into yellow medicine cells regardless of their previous state.



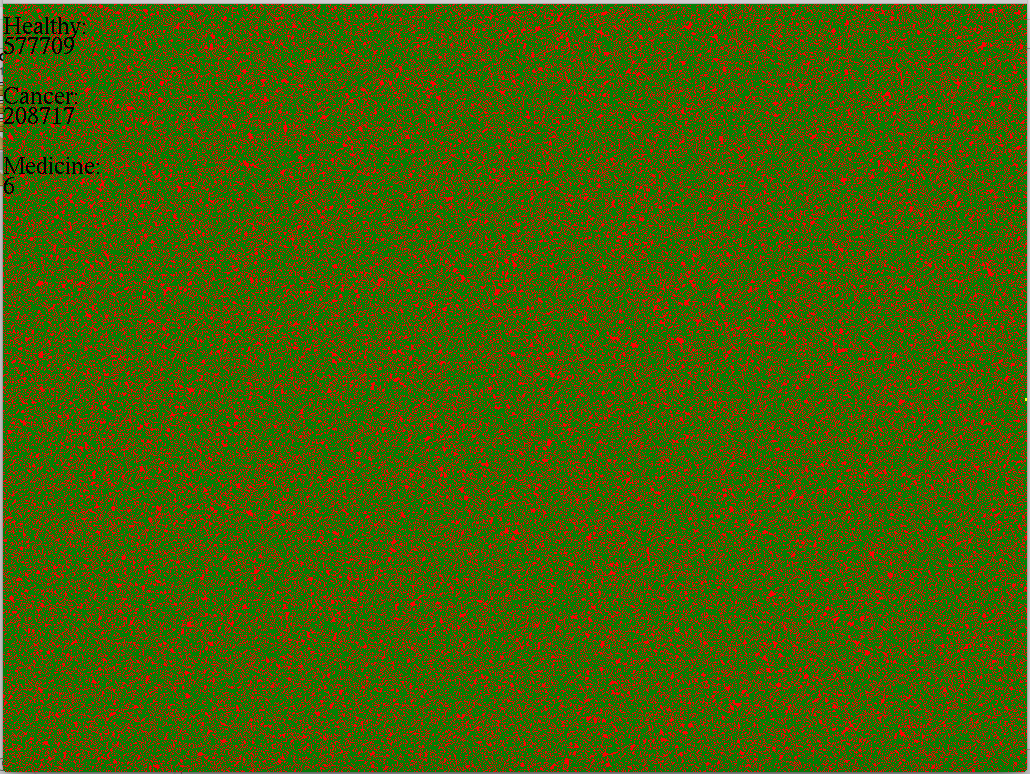
**Test Scenario #3:** Indirectly healing a single cancer cell

**User Input:** Click on the cells around a single cancer cell. Be sure to click at a distance of two (2) cells away due to the radial expansion of the medicine. It should take at least three (3) clicks in order for the cancer cell to be sufficiently surrounded by medicine.

**Expected Result:** The cancer cell is cured and turns into a green healthy cell due to the high number of surrounding yellow medicine cells. The surrounding yellow medicine cells remain as they are.

**Test Scenario #4:** Injecting medicine into a healthy cell at the edge of the area

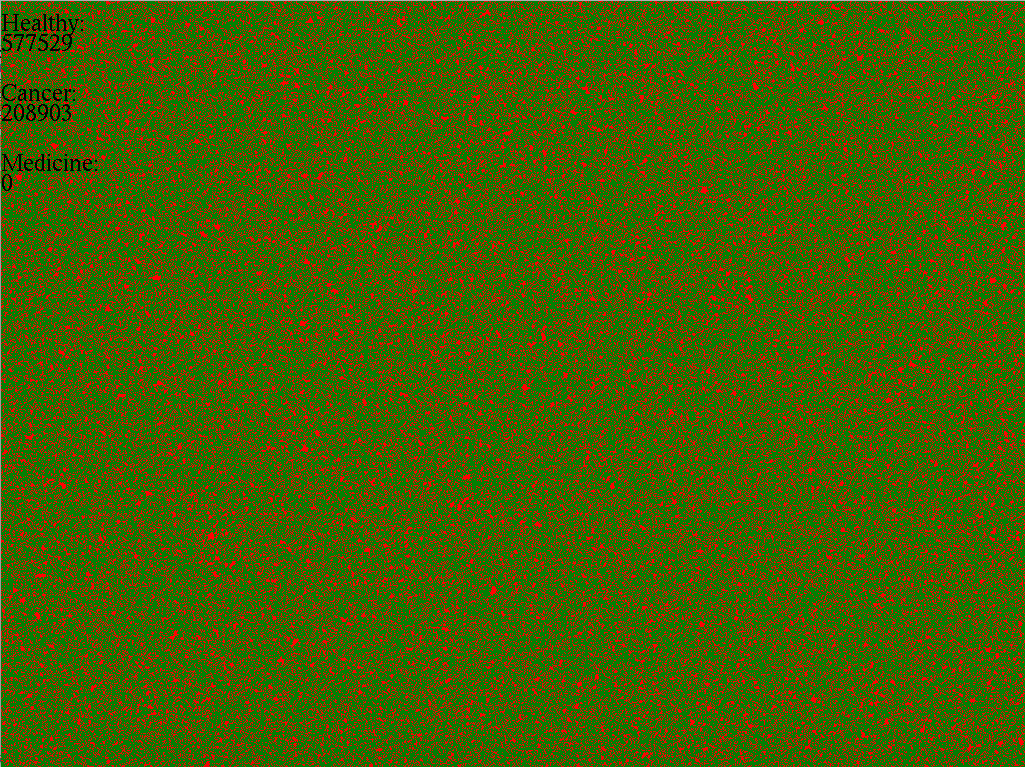
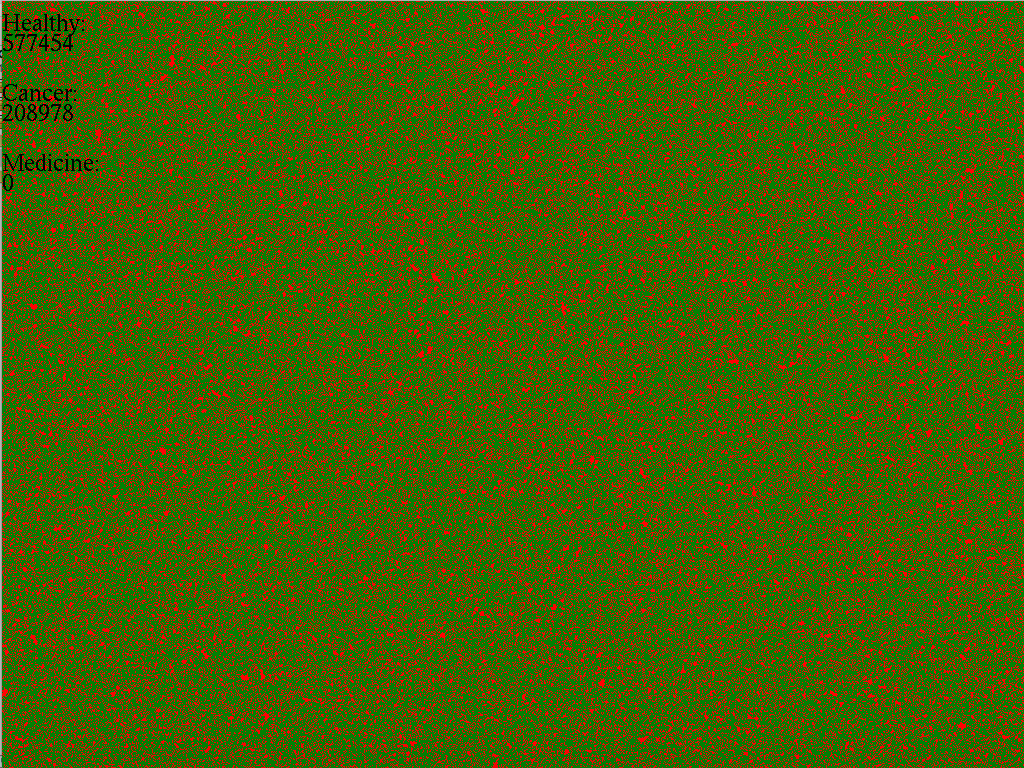
**User Input:** Click on a green healthy cell that is at the edge of the area (either on the left, right, top, or bottom edges, or in a corner).

**Expected Result:** Medicine is injected into the healthy cell and the cell becomes yellow. The medicine is not absorbed and moves radially outwards by one cell position in each possible direction, being sure to not expand outside of the boundaries of the area. Note that all the affected surrounding cells turn into yellow medicine cells regardless of their previous state.

**Test Scenario #5:** Simulation starts with ≥ 25% cancer cells randomly placed.

**User Input:** Restart the simulation several times (checking expected result each time).

**Expected Result:** The simulation starts with approximately the same amount of cancer cells each time (being ≈25%). The cells appear in different locations each time, due to the fact that they are randomly placed.

 Run #1 Run #2

Performance Analysis

While I was not able to perform a proper performance analysis for several reasons (described below), I do believe that implementing OpenCL did improve the performance of the cancer simulation. In previous implementations of the assignment (using threads, TBB, and CUDA), there was always some delay between clicking on a cell, and having that cell visually turn into a yellow medicine cell. This delay usually took over a second. However, using OpenCL, there is little or no delay in this action. Along with this improvement, the overall simulation seems to run quicker and smoother than before.

The reason that I was not able to perform a proper performance analysis is due to the fact that there are indeed a few factors that have changed between this implementation of the simulation and previous ones. First of all, as mentioned above, I have developed and tested the OpenCL implementation using Xcode natively on my MacBook. The CUDA implementation was developed and tested using Microsoft Visual Studio on the lab machines, which are running Windows. The thread and TBB implementations were developed in Microsoft Visual Studio on my Mac through a Parallels virtual machine running Windows 7. Thus, due to the variance in development IDE, testing OS, and computer hardware it is hard to truly test the performance improvement from one version of the simulation to another based solely on the code implementation.

Most Important Part of Assignment

The most important part of this assignment was integrating the GPU and CPU kernel code using OpenCL. Implementing both of these kernel calls allows for a heterogeneous multicore implementation of a previously homogeneous multicore program. This also allows for both data and task parallelization, making the program the most optimized it can possibly be (based on the hardware in use).

Most Difficult Part of Assignment

The most difficult part of the assignment, by far, was implementing OpenGL calls from within the OpenCL kernel. I was personally not able to successfully implement this. I was able to set up working OpenCL GPU and CPU kernels. However, when it came to attempting to optimize OpenGL tasks with OpenCL, I was not successful.