COMP 426

Assignment 3

Design Documentation

Christopher Mukherjee

6291929

November 17, 2014

CUDA Optimization

In this assignment, I took the code that was developed in Assignment 1 and optimized it by replacing the multithreading achieved using the standard thread library with CUDA data parallelization. Instead of explicitly creating a few threads to manage the parallelization of the cancer simulation, CUDA allows us to use hundreds of threads by using GPUs along with the CPUs. In the CUDA program, data can be stored on the GPU buffers and instructions can be executed by launching a kernel on the GPU.

Architecture

The entirety of the program is contained in one CUDA file (.cu). This file consists of a main function, 5 GLUT-related helper functions, 1 general helper function to begin the updating of the cells, 1 CUDA helper function, and 1 CUDA 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.

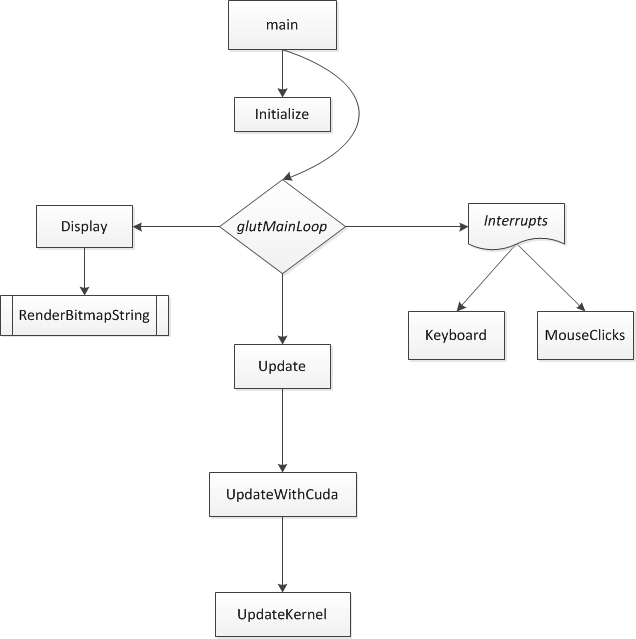
**RenderBitmapString**: GLUT-related function that renders bitmap strings to display text on screen

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

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

**UpdateWithCuda**: Helper function for using CUDA to update cells in parallel. Launches the CUDA kernel.

**UpdateKernel**: Updates each cell state.

Control Flow Diagram:

Techniques Used

The CUDA GPU kernel is used in order to parallelize the updating of the cells. The maximum acceptable block size on the machines in the multicore lab is 512 threads per block. Therefore, in my assignment, the kernel uses a block size (*dimBlock*) of 512 in order to achieve the greatest amount of optimization (by using the highest amount of parallel threads as possible). The grid size (*dimGrid*) uses the block size and scales it to the size of our entire simulation, which is 1024 x 768.

The 2-dimensional area of 1024 x 768 cells is represented by two global arrays. The reason I used arrays 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. The reason I used two arrays is to avoid errors to do with synchronization. I use one array to read from, and one array to write to. Therefore during each update, all calculations are performed on a static array, instead of a dynamically changing one. The values of the read array are then updated to the newly calculated values in the write array before the next call to the CUDA update functions.

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 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 also become green healthy cells.

**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:** A healthy cell turns into a cancer cell

**User Input:** Click on a red cancer cell that is surrounded by more than the majority (≥ 6) red cancer cells.

**Expected Result:** Medicine is injected into the cancer cell and is instantly absorbed. The cancer cell is temporarily healed and turns green. However, the cell quickly turns back into a red cancer cell due to the high number of surrounding red cancer cells.

**Test Scenario #6:** 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.

Most Important Part of Assignment

The most important part of this assignment is replacing the standard thread library with CUDA parallelization. Using CUDA, the program can use a much larger number of threads due to the fact that CUDA allows us to use the threads on the GPU along with the threads on the CPU. The simulation now runs hundreds of threads in parallel instead of less than ten. This means that the simulation is more parallelized and more optimized.

Most Difficult Part of Assignment

Personally, the most difficult part of the assignment was learning how to use CUDA. CUDA works much differently than any other programming technique I have ever used in the past. Therefore, I had a lot to learn for this assignment and had to conduct a lot of research and debugging (and some trial and error) in order to get everything to work properly. However, luckily for me, NVIDIA has a lot of documentation available online and on the computers in the multicore lab. Thus, with the help of this documentation, I was able to successfully complete this assignment.