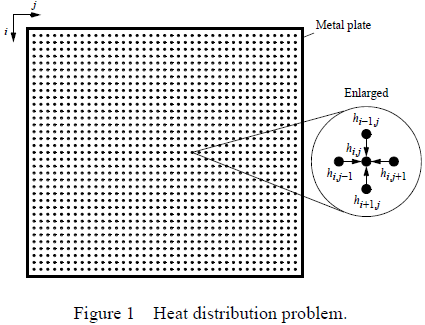
**WPI CS 525G Programming Assignment 2. Due before class on Wed Oct 26, 2011**

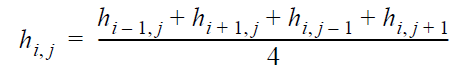
**Overview**

In this assignment, we will write a CUDA program to determine the heat distribution in a space using iterative computations on a GPU. We will start with a 2-dimensional space (square) and simple boundary conditions (walls at fixed temperatures). This program can then be modified to satisfy additional requirements.

**Determining Heat Distribution by a Finite Difference Method.** Consider an area that has known temperatures along each of its edges. The objective is to find the temperature distribution within the space. The temperature of an interior point will depend upon the temperatures around it. We can find the temperature distribution by dividing the area into a fine mesh of points, *hi*,*j*. The temperature at any inside point can be calculated as the average of the temperatures of its four neighboring points, as illustrated in Figure 1. For this calculation, it is convenient

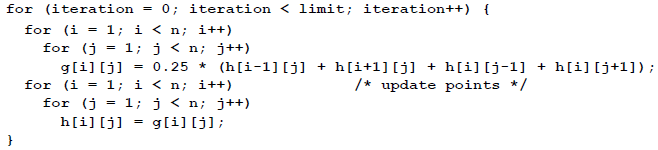


to describe the edges by points adjacent to the interior points. The interior points of *hi*,*j* are where 0 < *i* < *n*, 0 < *j* < *n* [(*n* - 1) x (*n* - 1) interior points]. The edge points are when *i* = 0, *i* = *n*, *j* = 0, or *j* = *n*, and have fixed values corresponding to the fixed temperatures of the edges. Hence, the full range of *hi*,*j* is 0 ≤ *i* ≤ *n*, 0 ≤ *j* ≤ *n*, and there are (*n* + 1) x (*n* + 1) points. We can compute the temperature of each point by iterating the equation



(0 < *i* < *n*, 0 < *j* < *n*) for a pre-defined number of iterations or until the difference between consecutive iterations at a point is less than some very small prescribed amount. This iteration equation occurs in several other similar problems; for example, with pressure and voltage. More complex versions appear for solving important problems in science and engineering. In fact, we are solving a system of linear equations. The method is known as the *finite difference* method. It can be extended into three dimensions by taking the average of six neighboring points, two in each dimension.

**Sequential Code.** Suppose the temperature of each point is held in an array h[i][j] and the boundary points h[0][x], h[x][0], h[n][x], and h[x][n] (0 ≤ *x* ≤ *n*) have been initialized to the edge temperatures. The calculation as sequential code could be

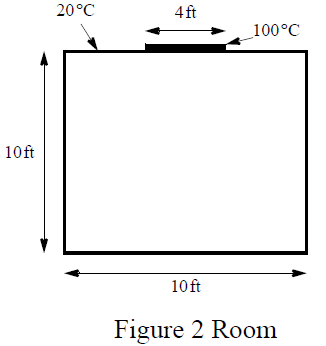


using a fixed number of iterations. Notice that a second array g[][] is used to hold the newly computed values of the points from the old values. The array h[][] is then updated with the new values held in g[][]. This is known as Jacobi iteration. Multiplying by 0.25 is done to compute the new value of the point rather than dividing by 4 because multiplication is usually more efficient than division. The usual methods to improve efficiency in sequential code carry over to GPU code and should be done where possible in all instances.

**Note:** It is possible to use the same array for the updated points (i.e. compute new values in place), thereby using some newly computed values for subsequent points (a Gauss-Seidel iteration) - this will converge significantly faster but may be difficult to implement on the GPU. However a sequential version should really use Gauss-Seidel iteration for comparison purposes when computing speedup factors

**Task 1 - Sequential Program**

Write a C program (on the CPU) to compute the temperature distribution inside the room shown in Figure 2 using Jacobi iteration. The room has four walls and a fireplace. The temperature of the wall is 20 degrees C, and the temperature of the fireplace is 100 degrees C. Divide the room into *N* x *N* points (including the boundaries), where *N* is an input variable and can vary. The values of the points are stored in an array.



**Task 2 - Basic CUDA Program**

Modify the sequential program in Task 1 into a CUDA program as follows.

1. Use dynamically allocated memory for the data arrays (h[N][N], g[N][N]) on both the CPU and GPU and add input statements to be able to specify N.
2. Implement a GPU version of the heat distribution calculation and verify that both the CPU and GPU versions produce the same (correct) results. Both your GPU and CPU implementations must be able to handle all values of *N* between 0 and 4000.
3. Explore different CUDA grid/block structures -- Add input statements to be able to read in different values for the CUDA grid/block structure:
   1. Numbers of threads in a block (T)
   2. Number of blocks in a grid (B)

(2-D grid and 2-D blocks). Include checks for invalid input. Your program should not run using invalid block/thread values.

1. Timing -- Add statements to time the execution of the code using CUDA events, both for

the host-only (CPU) computation and with the device (GPU) computation, and display

results. To display your results, simply print out a 6x6 grid of final values at the four

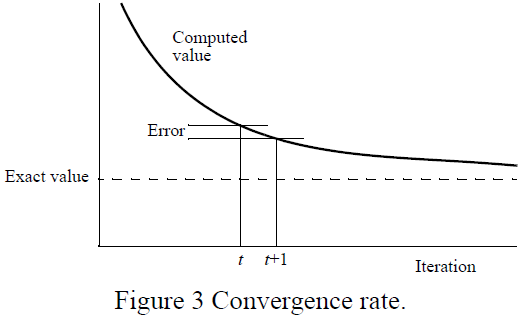
corners and at the center of the grid. Compute the speed-up factor.

1. Repeat your GPU implementation, part d and timings using shared memory and compare your timings and speedups with those with using global memory

Include print statements to show all input values and a 6x6 grid of your final values for the four corners of the room as well as the center . Experiment with different input values (at least ten different combinations of T, B, and N) and collect results. Create bar charts of your results and discuss your results. For instance, explain why certain values of T and B are optimal.

**Task 3 Termination Detection**

In the sample sequential code in Task 2, termination is set by a specific number of iterations. However the computed values may not have converged sufficiently towards the solution by that time. Re-write the CUDA code to terminate the computation when ALL values computed in iteration *t*+1 differ by those in iteration *t* by less than a value that is input, say *e*. Repeat the study in Task 2 with this CUDA program and comment on the results. Note that the above does not guarantee the computed values are accurate to ±*e*, see Figure 3.



**Grading**

Every task and subtask specified will be allocated a score so make sure you clearly identify each part you did. The computational efficiency and elegance of your solutions is will be a factor in grading.

**Submitting**

Write up your results in a separate Word document including a description of your code, discussion of your graphs, and conclusions. Test your program rigorously, clean up and comment your code properly. Your documentation doesn’t have to be long, but it should describe the main files generated and functions you implemented. Zip up your entire solution, including Visual Studio solution files, C/CUDA files and your Word documentation and email it to me ([emmanuel@cs.wpi.edu](mailto:emmanuel@cs.wpi.edu)) by the due date. Name your zip file as your FirstName\_lastName\_cs525g\_prog2.zip