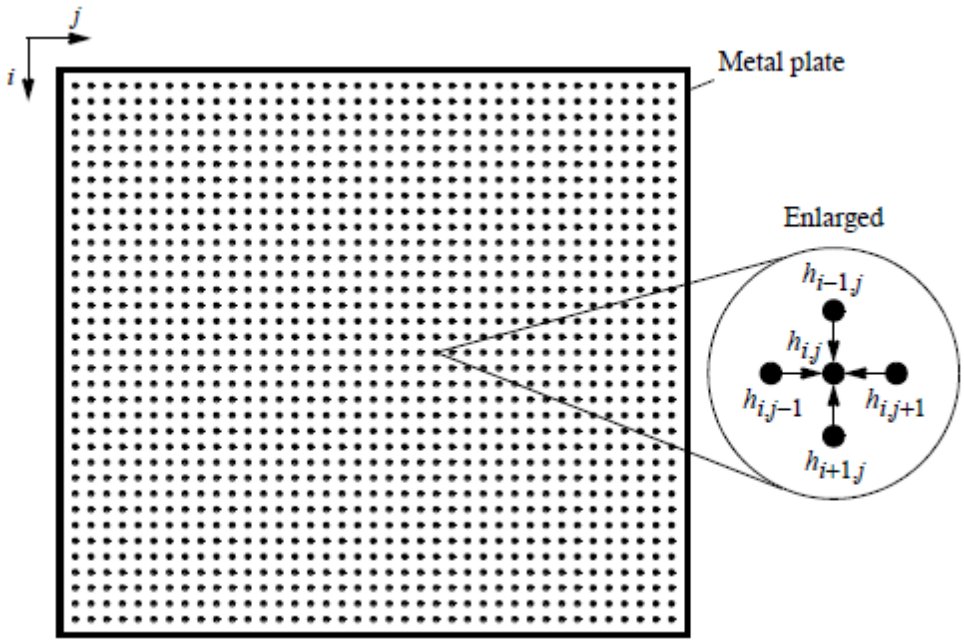
Graphics Processing Units (GPUs): Architecture and Programming
Programming Assignment # 2

In this programming assignment, you will implement a GPU friendly application and see yourself when a GPU friendly application will yield better performance than sequential version and when not. Also, we will try some optimizations from the ones we learned in class.

You will write CUDA program to determine the heat distribution in a space using synchronous iteration on a GPU. We have 2-dimensional square space and simple boundary conditions (edge points have fixed temperatures). The objective is to find the temperature distribution within. The temperature of the interior depends upon the temperatures around it. We find the temperature distribution by dividing the area into a fine mesh of points, $h_{i,j}$. The temperature at an inside point is calculated as the average of the temperatures of the four neighboring points, as shown in the following figure.



The edge points are when i = 0, i = n-1, j = 0, or j = n-1, and have fixed values corresponding to the fixed temperatures of the edges. The temperature at each interior point is calculated as:

$$h_{i,j} = \frac{h_{i-1,j} + h_{i+1,j} + h_{i,j-1} + h_{i,j+1}}{4}$$

(**0 < i < n** ; **0 < j < n**, remember that edge points have fixed values)

Stopping condition: You can have a fixed number of iterations or when the difference between two consecutive iterations (calculated as average among all points) is less than or equal to some number *e*. For this assignment, we will use fixed number of iterations.

Assume we have (N x N) points (including edge points), the initial situation is as follows:

- The edge points are fixed as follows:
  - Points (0, 0) to (0, N-1) inclusive have temperature of 100F. [Note: (x,y) means row x and column y].
  - Points (N-1, 0) to (N-1, N-1) inclusive have temperature of 150F.
- All other points are initialized to zero.

Remember that the boundary points do not change across iterations. This includes points (1,0) to (N-2, 0) and (1, N-1) to (N-2, N-1) that are fixed to 0.

There will be a fixed number of iterations ITR. In each iteration, each point in the grid will be updated. But be careful: The points at iteration i must use the neighbors values of iteration i, not the updated version. So, you may need to have a temporary storage for values of iteration i and you update the values in a different storage. Then, in the next iteration, the second storage becomes the new current and you update the other one.

You will implement two CUDA versions:

- GPU version: This is the traditional one where you dynamically allocate the array in the global memory, distribute work among threads and blocks, and do the iterations.
- Optimized GPU version: Same version as above but you add the tiling to reduce global memory access, as we saw in class.

There are some questions that you need to think about:

- Are there any communications going one between CPU and GPU? Or we can optimize this.
- Is it better to implement one big kernel to do the work, or a series of kernels called by the host?

As a help, we are providing you with a source code that contains the sequential code and the time measurement.

What you have to do:

1. [40 points: one for each version of the GPU] Write CUDA in the provided file heatdist.cu. The simplest way to compile your code is (be sure to do the steps mentioned in the course website first):

   nvcc -o heatdist heatdist.cu

   The generated binary will have the name: *heatdist.*
   Type: **./heatdist** and press enter to see the usage. Then start implementing your GPU version in the designated place in the *.cu file. Read the comments on that file before starting to code.
   The usage is simple, it takes 3 arguments:
   ./heatdist num iterations who
   num = dimension of the square matrix

iterations = number of iterations till stopping (1 and up)
who = 0: sequential code on CPU, 1: GPU version, 2:GPU optimized version

2. Experiment 1: Number of iterations is fixed to 100,000. As for the dimension, start with num = 1000 and then double in each experiment (i.e. 1000, 2000, 4000, 8000, 16000, … till the dynamic allocation fails), record the time taken by the CPU, the GPU, and the optimized GPU versions. The provided file already calculates the time for you and print it on the screen. Draw a bar-graph where x-axis is the dimension and y-axis is the speedup (= time CPU/time GPU). This means that for each dimension, you will have two bard-graphs: one for the GPU and one for the optimized GPU.

3. Experiment 2: Fix dimension to 1000 and vary the number of iterations starting from 1000 and doubling (1000, 2000, 4000, 8000, 16000, 32000, and 64000). Draw similar graph like the above but the x-axis this time is the number of iterations.

   [20 points, 10 per graph]

4. [40 points, 10 per bullet point] What are your conclusions regarding:
   a) At which dimension of N is GPU usage more beneficial? And what is your interpretation of that?
   b) When is the speedup (i.e. time of CPU version / time of CUDA version) at its lowest? And why? Answer this for both versions of CUDA you wrote.
   c) When is the speedup at its highest? And why? Answer this for both versions of CUDA you wrote.
   d) Which has more effect: number of iterations or the problem size? and why?

What to submit:

- The source code heatdist.cu
- A report that contains the graphs and answers to the questions mentioned above.

Put all the above in a zip file named by your netID.zip and upload them to Brightspace.

Important: Do this lab on one of the cuda machines and stick to it in all your experiments for this lab.
I hope that after implementing this programming assignment, **you become familiar with the following items** first-hand (You got some familiarity in lab 1.):

- How to allocate/free memory space at the device.
- How to optimize the code.
- When is the optimization beneficial and when not.
- Get a feel of when the problem size is enough to get better performance on device when the kernel is GPU friendly.

So …..

Have Fun!