# Homework 3
# Weighted Jacobi Iteration using CUDA

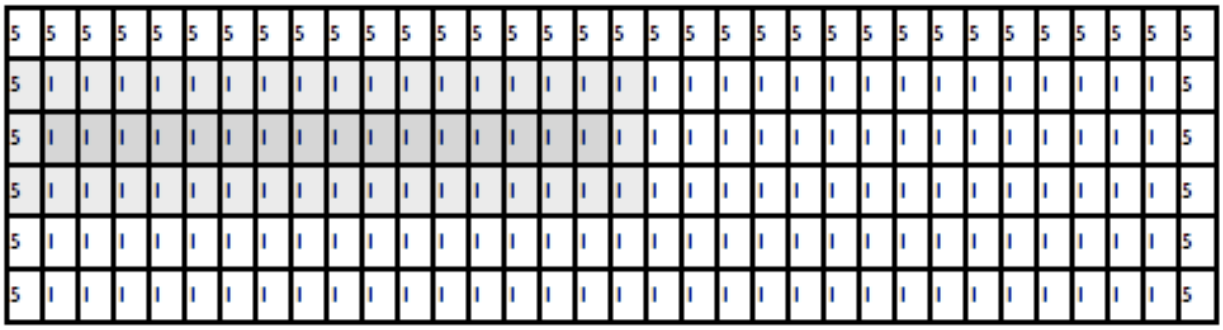
**Overview**

This assignment asks you to implement a variant of *Jacobi Iteration* in order to simulate the traversal of heat through a metal plate. The plate is represented by a 2D array of floats in which the border cells are considered to be of fixed temperature, and the interior cells have a variable temperature that is influenced by neighboring cells.

Our implementation of Jacobi Iteration just visits each of the interior cells and computes a new value by computing a weighted average of the 8 surrounding cells. The process is repeated over a user specified number of *passes*, after which the resulting array values may be displayed as text.

**Technical Summary**

The serial version of the code is very straightforward, and is provided for timing purposes. For the parallel version, you might start with a **1D block shape** with a length that is a multiple of 16 (or close to it). The provided kernel and host code uses this scheme, though you are free to explore other options.



The figure above shows the upper portion of a small example dataset with width of 34 elements. The set of shaded boxes represents the elements that will be read from while calculating the new values. The 1x16 region with darkest shading represents the elements that will have new values computed during one execution of a 1x16 block of threads. In practice, we'll read values from one copy of the array, and write new values to a different copy of the array.

New array values are produced by reading the 8 values surrounding an element (North, NorthEast, East, SouthEast, South, SouthWest, West, NorthWest), and calculating a weighted average. In order to achieve reasonable performance, your program should read the data values necessary for a block into ***shared memory*** from the slower global memory. Once this is done, you can do your calculations by reading from shared memory, and then write results back to global memory.

The Jacobi algorithm processes the array in multiple passes, and maintains pointers to two arrays, *old* and *new*. Each pass updates the values in *new* by reading values from *old*. After each pass, *old* and *new* are swapped. Examination of the serial code should prove very illuminating.

In the CUDA version, the kernel should probably implement a **single pass**. After each pass, the device pointers are swapped so that when the kernel is called again, the referenced arrays have exchanged roles. The

provided host code does this for you.

Lastly, when writing your kernel, be sure to handle the case where a thread wants to write values off the end of your array. You'll need to use if-statements for this and probably other purposes, so don't be afraid to do so.

**What is provided?**

Links to both the CUDA Programming Guide and Manual of cuda-gdb are available on EWU canvas under Files→Resources.

A start up package has been set up, which include a very simple kernel **k0** and the **supporting host code**. But, you have to carefully read the host code in order to understand the kernel configuration.

**Compiling and Debugging**

Notice that the provided **hw3_student** directory contains a Makefile that will help you build your code. As it stands, you should be able to type "make" and have an executable named *jacobi* automatically produced for you, provided there are no syntax errors. The *make* command can also take arguments. Typing "make clean" will remove all executable and object files associated with the project.

The makefile is setup to enable debugging by default, compiling with the -g and -G flags. This allows you to use the cuda debugger (called cuda-gdb), which is a modified form of gdb. Nvidia maintains documentation on this tool at their website.

After a successful build, you can run jacobi with the following arguments, where p determines whether results are printed, and width and height determine the size of the arrays used:

```
./jacobi threadsperblock passes width height [p]
```

If ***threadsperblock*** is zero, the serial version is run. Otherwise, this argument determines the number of threads in each block for the GPU kernel.

The provided kernel works correctly, but accesses global memory freely, eliminating any performance benefit over the serial case. Your job is to write code that reads necessary data into shared memory from global memory efficiently, performs the calculation, and then writes results back out to global memory. **Try your code with a variety of array and block sizes to make sure it is correct. (Use diff to compare the output of your parallel code and results of the provided serial code)**

When you're sure you have correct code, prepare a simple chart like the one below, using the tool of your choice. First, do a "make clean", and then rebuild your code with the following options:

make dbg="" cuda_dbg="" opt="-O3"

Test your performance with the following command lines, replacing **threadsPerBlock** and **n** with the values given in the chart:

```
./jacobi threadsPerBlock 1000 n n
```

For the given values of **n**, first collect execution times for the serial code by setting *threadsPerBlock* to 0. Put these in the row labeled "Time cost for Serial". Next, collect times for "Time cost for CUDA k0" and Time

cost for CUDA k1" by using the values for both n and *threadsPerBlock* as described in the chart. Compute a speedup using these numbers. (Is it really a true speedup, or a marketing trick?)

If there is anything special you'd like to point out about your implementation, include a brief paragraph or bullet list of features below your chart.

| **n** (array dimension) | 1600 | 1600 | 1600 | 1600 | 1600 | 1600 | 3200 | 3200 |
|---|---|---|---|---|---|---|---|---|
| threadPerBlock | 32 | 128 | 256 | 384 | 512 | 1024 | 256 | 128 |
| Time cost for Serial (ms) threadPerBlock=0 | | | | | | | | |
| Time cost for CUDA k0 (ms) | | | | | | | | |
| Speedup of K0 compared with Serial Code. | | | | | | | | |
| Time cost for CUDA k1 (ms) | | | | | | | | |
| Speedup of k1 compared with Serial Code | | | | | | | | |
| Speedup k1 compared with K0 | | | | | | | | |

**Submission**

Wrap up all your source code and your result report into a single zip file, submit the single zip file through EWU canvas.