ASSIGNMENT #1

Programming Massively Parallel Processors

Due Date:	As specified on the Moodle
Topics covered:	CUDA programming model, Execution model
Deliverables:	The submitted project files are kernel.cu and cpu.cpp. The files should be submitted according to the stipulations set out in the course syllabus.
Objectives:	Learn how to use CUDA runtime API to write kernel functions.

Programming Statement

This is a CUDA C programming assignment. Students are expected to finish the programming (CUDA C/C++) for a given computation problem.

Problem Statement

In this assignment, we will write CUDA programs to determine a distribution in a 2D space using synchronous iteration on a GPU for the simulation of heat transfer problem. Assume that there is a 2-dimensional square space with $n \times n$ points, where the boundary edge points have fixed values. The objective is to find the value distribution within the 2D space. The value of the interior points depends upon the values of the points around it. We can find the value distribution of the interior points by dividing the area into a fine mesh of points, $v_{i,j}$. The value at an inside point is calculated as the average of the values of the four neighboring points. The edge points are the points (i,j) when $i = 0, i = n-1, j = 0$, or $j = n-1$, and have fixed values corresponding to the fixed values of the edges. The value at each interior point is calculated as:

$$v_{i,j}^{(K+1)} = \frac{v_{i-1,j}^{(K)} + v_{i+1,j}^{(K)} + v_{i,j-1}^{(K)} + v_{i,j+1}^{(K)}}{4} \tag{1}$$

where $0 < i < n-1$; $0 < j < n-1$, and K is the iteration number.

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. For this assignment, we will use the fixed number of iterations.

Assume we have $n \times n$ points (including edge points). The edge points have the value of 26.67, except points ((0, 10) to (0, 30) inclusive) have the value of 65.56. Assume that all internal points are initialized to zero.

During the calculation of the distribution, you will first initialize the values of the inputs for all the $n \times n$ points (including edge points) by the following function

```
void initPoints(float *pointIn, float *pointOut, uint nRowPoints)
```

In the above, **nRowPoints** is the width, n, for the points of 2D space area and the input is pointed by **pointIn**. As there is no calculation required for the edge points, you can set the output values (pointed by **pointOut**) of the edge points during initialization.

The CPU version for the calculation of the distribution is as follows:

```
heatDistrCPU(float *pointIn, float *pointOut, uint nRowPoints, uint nIter)
```

In the above you will apply the update based on the equation 1 over all the interior points for *nIter* iterations. For each interior points, you will use the inputs pointed by **pointIn** to calculate the outputs pointed by **pointOut**. According to the update equation, you will get the values of the surrounding 4 points to calculate the average result. At the end of each iteration, you should update next iteration input from the current iteration output.

In GPU version, you will use the following function to calculate the distribution with the given inputs pointed by *d_DataIn*:

```
void heatDistrGPU(float *d_DataIn, float *d_DataOut, uint nRowPoints, uint nIter)
```

where *d_DataOut* points to the outputs.

In the above function, for each of *nIter* iterations , you will firstly launch the calculation kernel

```
__global__ void heatDistrCalc(float *in, float *out, uint nRowPoints)
```

Once it is done, you need to call `cudaDeviceSynchronize()` to wait for all the threads in the grid to finish. Then, it is expected to use the outputs of K-th iteration calculation as the inputs in the $(K + 1)$-th iteration calculation so you will call the following kernel to update:

```
__global__ void heatDistrUpdate(float *in, float *out, uint nRowPoints)
```

After the update of the inputs for next iteration, you need to synchronize with GPU device again.

You are required to turn in your homework that is compilable under the Windows environment using Visual Studio.

What to do

Please install CUDA toolkit (version 13.0) on your own PC/laptop and VS 2022. Please refer to the Moodle link for the template that you can use to start your CUDA programming in Windows. You should also download **CUDA common include** on the Moodle and add the unzipped folder directory into **Project/Properties/CUDA C/C++ Additional Include Directories** for VS 2022. You can refer to the reference executable file A1.exe on the Moodle and rename it to heat.exe and run in Windows command prompt, as follows:

```
c:\yourdir>heat.exe 32 100
```

where 32 is the number of rows for points and 100 is the number of iterations. The reference executable prints out row-by-row. That means the values of the points (0,0), (0,1),...(0,nRowPoints-1) are printed out first, then (1,0), (1,1), ..., (1,nRowPoints-1). Please ensure your CPU code is correct and also compare the GPU code results with the reference. Your submissions may fail to match with the reference due to e.g. swapping column index with row index. Please note that as specified earlier, **nRowPoints** should be at least 32.

What you have to do:

1. Complete the sequential C version of the problem.
2. Write CUDA version of the above code.

3. Execute the programs in (1) and (2) with n = 100; 500; 1,000; 10,000. Assume 50 iterations.
4. Obtain the overall time taken (i.e. neither user nor system times). You may need to repeat the experiments 5 times and take the average to amortize any skew that may be caused by system load.
5. You may draw a bar-graph showing n (x-axis) and the time (y-axis) for the 2 versions of the program (2 bars per n on the same graph) to see the trend.
6. What are your conclusions regarding:
   - When is GPU usage more beneficial (at which n and why)?
   - When is the speedup (i.e. time of CPU version / time of CUDA version) at its lowest? And why?
   - When is the speedup at its highest? And why?
7. Repeat (4), (5), and 6 with 500 iterations.
8. What is the effect of increasing the number of iterations?

Please note that Step 3-8 are not required for submission.

You are suggested to set up a VS project based on the template on your own PC/laptop. Firstly you can finish the initialization function and CPU version for calculation and cross-check with the reference results. If it works correctly, you then develop GPU version. The template has provided the code to verify whether the calculation results match for your CPU and GPU code. You may use to check whether you GPU code is correct. Alternatively, yo may make use of **printf** to help you debug your code. Please note that the VPL is used mainly for you to auto-grade your submission rather than for testing and debugging purpose.

Grading Guideline

Please ensure that:

1. Functional Correctness: Produces correct result (CUDA C/C++ version is correct).
2. Coding:
   (a) Correct usage of CUDA library calls and C extensions.
   (b) Correct usage of thread id's in computation.
3. Pass the VPL test cases.

Your submission will be graded according to the test results. If your code fail to pass the test cases for $n = 32, \cdots, 1000$, zero mark will be given.

The list above is non-exhaustive.

The lecturer reserves the right to impose reasonable penalties for code that violates general practices or does not match the specification in an obvious way that has not been mentioned above. In exceptional cases, the lecturer reserves a discretionary right to allow resubmission or submission after the deadline.