Heat Transfer in a Metal Plate

1. Summary:

In this exercise you are asked to implement a GPU-accelerated application which simulates the heat propagation on a conductive metal plate. The progress of the simulation will be displayed through a 2D thermal imaging animation. The exercise aims to familiarize you with:

- Converting a studied physical phenomenon into a numerical simulation
- Using a GPU to speed up a parallel computing application
- Implementing a load balancer to split work between the CPU and the GPU for additional performance
- Using interoperability between GPU computing and graphics APIs to display the results of the numerical simulation

2. Physical Context:

Consider an environment which contains a 2D heat-conductive metal plate. The initial temperature of the plate and the surrounding environment (air) is T_R. Then we introduce a very small but constant heat source which provides a constant temperature T_S. The heat source is placed on the plate and starts warming it up. Because of the plate's conduction, the heat will progressively spread on the surface, as caused by the induced motion of atoms: the atoms closest to the heat source receive thermal energy and begin to vibrate, triggering collisions with other atoms; energy passed from these collisions makes other atoms vibrate and so on.

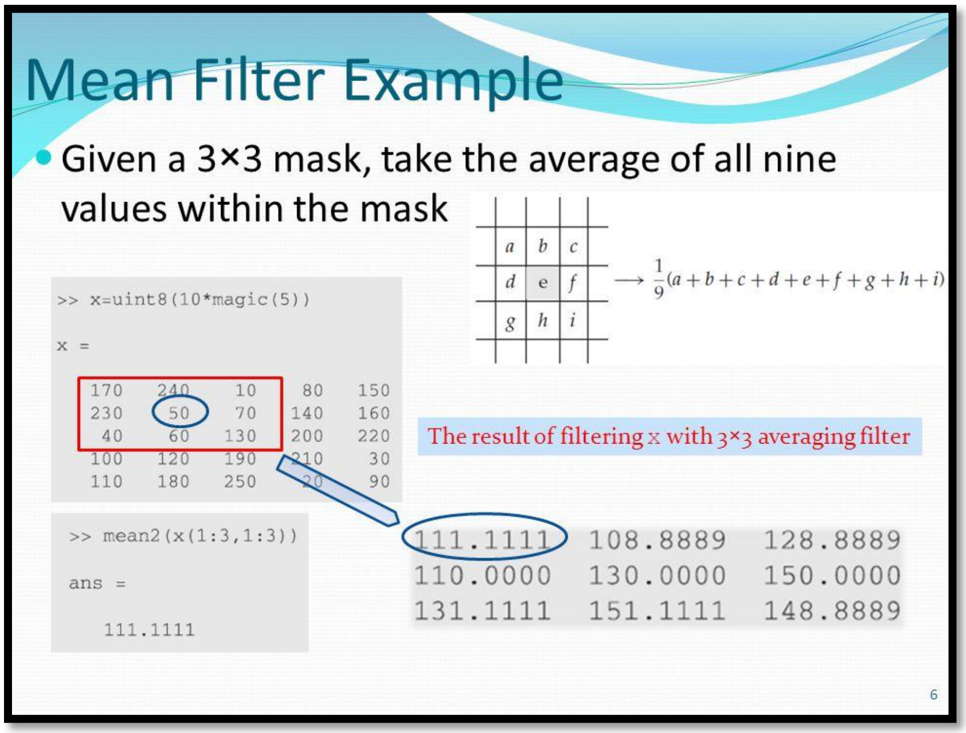
The heating of the plate is a process that begins with the introduction of the heat source and stops when the system reaches a thermal equilibrium (a state where there are no more variations in temperature in any point of the metal plate). In more technical terms, the process is called *transient heat conduction* and its equilibrium state at the end is called *steady-state conduction* (read more about both here). Throughout the time, the process is governed by the *heat equation* (read about it here).

3. Simulation details:

To simulate the heat transfer process described above, we will consider the metal plate to be a matrix of size `M x N`. Given a moment of time `t` we consider `P[x,y](t)` to represent the (instantaneous) heat of point `(x, y)` in that moment (with the constraints $0 \le x < M$ and $0 \le y < N$).

To simulate the initial condition, we consider the heat source to be placed at some given point (x_0, y_0), meaning that $P[x_0, y_0](t) = T_S$ (constant over time). To model the surrounding environment, we considered the matrix to be "bordered" by one additional row or column on each side, holding only the value T_R in each point (also constant over time).

Given the cause of heat propagation as explained in the previous chapter, at any given moment in time `t` we will consider that the heat of any point is affected only by the heat values of its 8 direct neighboring points. For the points of the edge of the matrix (where `x` is `0` or `M - 1` and where `y` is `0` or `N - 1`) some neighbor points will be on the "borders" (always having the constant value T_R).



Mean Filter Example

- Given a 3×3 mask, take the average of all nine values within the mask

$$\begin{array}{|c|c|c|} \hline a & b & c \\ \hline d & e & f \\ \hline g & h & i \\ \hline \end{array} \longrightarrow \frac{1}{9}(a+b+c+d+e+f+g+h+i)$$

```
>> x=uint8(10*magic(5))

x =

  170   240    10    80   150
  230    50    70   140   160
   40    60   130   200   220
  100   120   190   210    30
  110   180   250    20    90
```

```
>> mean2(x(1:3,1:3))

ans =

   111.1111
```

The result of filtering x with 3×3 averaging filter

```
111.1111   108.8889   128.8889
110.0000   130.0000   150.0000
131.1111   151.1111   148.8889
```

6

Considering the state of the matrix `P` at some moment `t`, we now need to determine its state at the next moment, `t + 1`. For this we will use a simple *3 x 3 mean filter* (an image processing example is given here). The filter will be applied independently to each point of matrix `P(t)` in order to generate matrix `P(t + 1)`. For the application, consider a point `P[x,y](t)` and all of its 8 direct neighbors. Application of the mean filter (which simply calculates the average of all 9 points) is given by the following (intuitive) formula:

$$P[x,y](t+1) = \frac{1}{9}\sum_{j=-1}^{1}\sum_{i=-1}^{1} P[x+j,y+i](t)$$

The earlier image provides a numerical example (you can read more about it here). It is now clear to see that calculating `P(t + 1)` from `P(t)` can be done in parallel with ease (namely, any point $P[x_1, y_1](t)$ can be calculated in parallel to any other point $P[x_2, y_2](t)$) – in fact, this is an example of an *embarrassingly parallel* application.

The heat transfer simulation effectively starts with `P(t = 0)` and proceeds to calculate `P(t = 1)`, then `P(t = 2)` and so on, until a *state of convergence* is reached. This state corresponds to the thermal equilibrium state described in the previous chapter and occurs when there is no more significant change between `P(t + 1)` and `P(t)`. Numerically, this means that the sums of all absolute differences between the new and current state of `P` fall below a (small) threshold:

$$\left(\sum_{x=0}^{M-1}\sum_{y=0}^{N-1} | P[x,y](t+1) - P[x,y](t) | \right) < \varepsilon$$

The initial state corresponds to the metal plate sitting in ambient air without the heat source being applied; therefore, in this state the matrix `P(t = 0)` contains the value T_R in all points.

4. Deliverables and Scoring:

You are asked to implement the following tasks:

1) Take as input `M`, `N` (the dimensions of the temperature matrix for the metal plate), x_0, y_0 (the point where the heat source is placed), T_S and T_R. Run the simulation by calculating `P(t)` (namely, `P(1)`, then `P(2)` and so on) as explained earlier *in parallel* using a GPU computing device of your choice and your preferred GPU computing API (OpenCL or CUDA). Provide a notification when convergence is reached. **(45 points)**

2) To visualize the simulation, you will need to display `P(t)` as a 2D thermal field image (an example is here). To convert a temperature value to RGB for the display you may use the algorithm found here, or any other alternative of your choice. To generate the image, you must use the interoperability with OpenGL or DirectX available in the OpenCL or CUDA APIs. (**30 points**)

3) Implement a load balancing mechanism, where a percentage `F%` of values is calculated on the GPU and the remaining percentage of `(100 - F)%` is calculated on the CPU in parallel (using `pthreads` or any other API of your choice). The value of `F` is given as input ($0 \leq F \leq 100$) and may be changed dynamically by the user throughout the simulation. To see the effects of when balancing the computation load you will display the average time needed to calculate a single simulation step (that is, full calculation of `P(t)` for a given moment `t`). The time will be displayed in the console or together with the RGB image, depending on your choice. (**15 points**)

4) Allow the user to change the problem configuration dynamically (i.e. the users may change the position of the heat source by providing a different position (x_0, y_0), they may change the heat source temperature T_S or the air temperature T_R, however the matrix dimensions `M` and `N` will remain constant throughout the simulation). The simulation should not be restarted from the beginning when changing the configuration. **(10 points)**