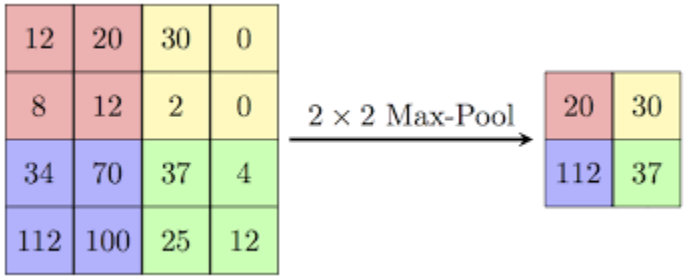
Ex2 - TPA
deadline : 18/06/2023 20:00 +0200

Write a parallel code in C++ 11 and OpenMP that solves efficiently the four main operations in a Convolution Neural Network (CNN):

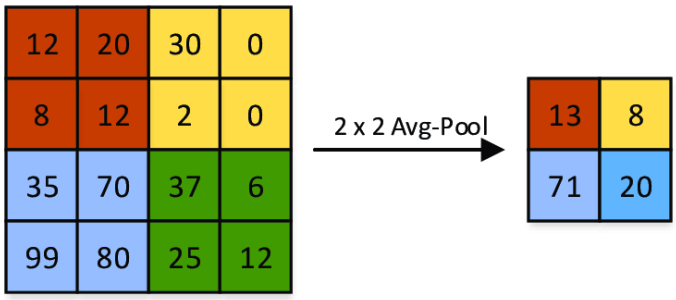
1 - MaxPool
2 - AvgPool
3 - ReLU6
4 - (simplified) Convolution

The operations act as follows:

MaxPool: this operation is used to **reduce** the resolution of an image (matrix). For each window (a 4x4 submatrix), MaxPool selects only the element with the highest value. The resulting matrix will then have ¼ the rows and ¼ the columns of the input matrix.
Given an input matrix of floating point values, the MaxPool returns a smaller matrix composed of the elements with the highest value in each submatrix.
For simplicity the input matrix is square and with sizes multiple of 4. In the image the submatrix is 2x2.



AvgPool: this operation is used to **reduce** the resolution of an image (matrix). For each window (a 4x4 submatrix), AvgPool calculates the average of the 16 elements in the window. The resulting matrix will then have ¼ the rows and ¼ the columns of the input matrix.
Given an input matrix of floating point values, the AvgPool returns a smaller matrix composed of the average among the elements of the submatrices.
For simplicity the input matrix is square and with sizes multiple of 4. In the image the submatrix is 2x2.



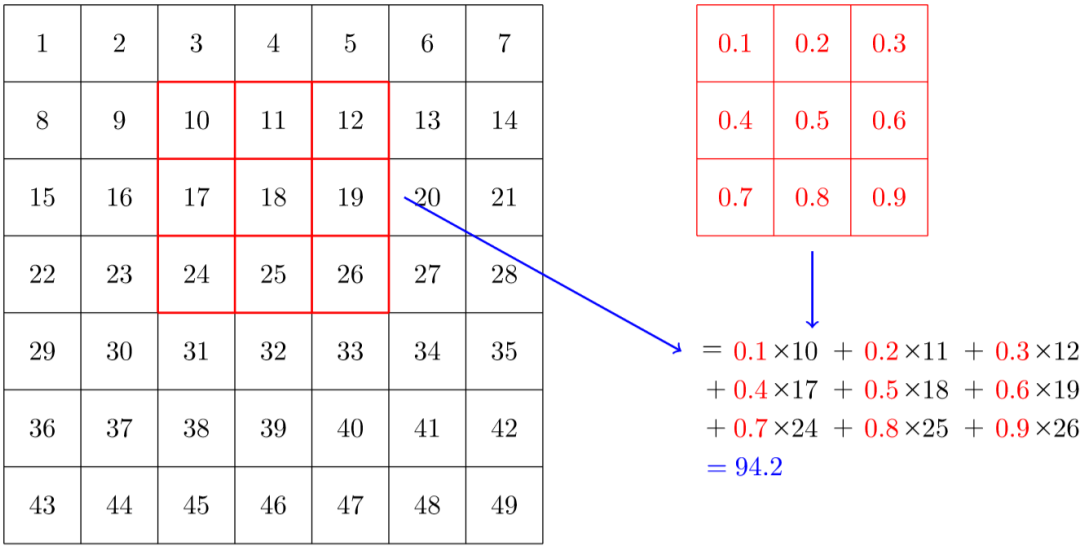
ReLU6: CNN normally operates with a reduced range of values. To improve performances, values smaller than 0 are put to 0 and values bigger than 6 are forced to 6. ReLU6, then, for each element in the input matrix will return:

- 0 if the element is < 0
- 6 if the element if > 6
- The value of the element otherwise.

Obviously, the output of ReLU6 is a matrix that has the same size of the input matrix.

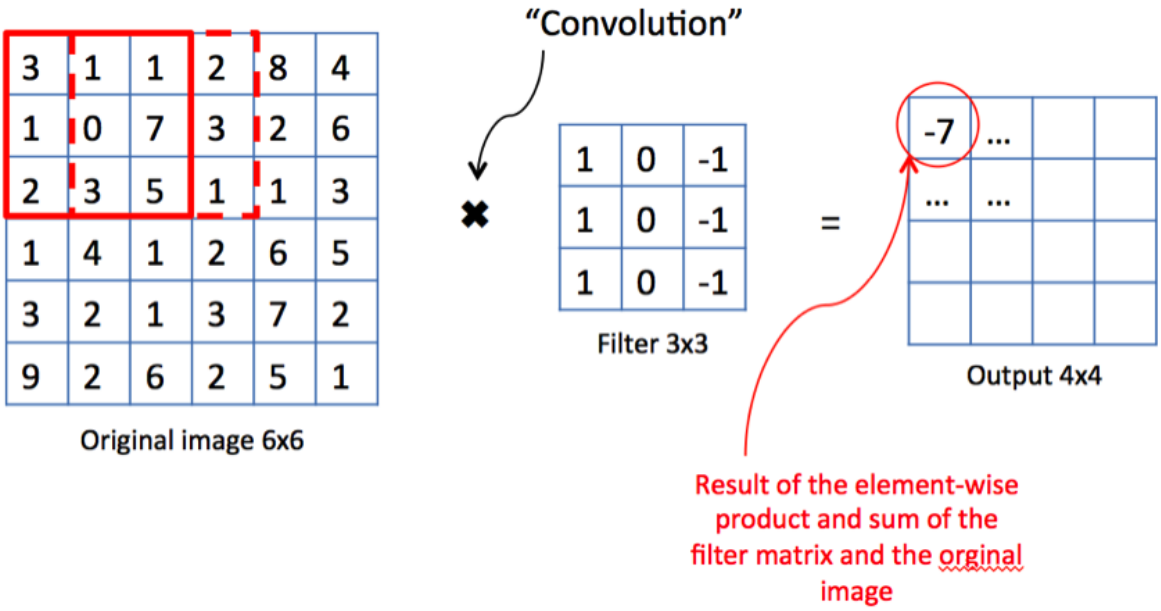
Convolution: this is the most interesting and complex operation in a CNN. With convolution the CNN extracts information (feature map) from the input image that is then used to detect and classify objects. We will implement a **simplified** convolution on a matrix of float rather than on an image (working on an image is possible, but requires to deal with heathers and descriptors that are well out of the scope of this course. You could try with a raw image of RGB values, just for fun. jpg, png, or any format requires to understand the compression: *do not try it*).
The simplified convolution is performed as **multiplication and accumulation** between a **kernel** (4x4 matrix of fixed values) and a 4x4 submatrix of the input (the input matrix will be square, bigger than 4x4, and with sizes multiple of 4). The result of the multiplication and accumulation is **one floating value** obtained accumulating (adding) all the elements of the multiplication. In other words:
1 - you instantiate conv_tmp as a floating value to 0 (remember to instante to 0).
2 - you perform the sub_matrix[i,j] * kernel[i,j] multiplication, which is a floating point value x.
3 - you accumulate the result in conv_tmp += x.

*** the pics have kernels 3x3, we will work on 4x4 kernels ***



You need to do so for all the rows and columns of the 4x4 kernel and submatrix.

The convolution is performed sliding the kernel in the input matrix as follows:



To simplify the code it is **not** necessary to add zero-padding to the edge of the original image, thus the output matrix will be **smaller** (you need to understand the size of the output matrix) than the input matrix.

Implementation details:
Inputs: file with input matrix (size 128x128) and kernel (size 4x4)

The goal is to first have a working sequential code for the four operations. Then, parallelize the operations that can be efficiently parallelized. Pay special attention to *data races* (more threads requesting the same input) and to *concurrency/conflicts* (multiple threads updating the same output). As a general suggestion, to achieve the best performance you should group in one thread (or in multiple threads executed in the same CPU) all the operations that work on the same input data. This avoids costly data copy to multiple locations.

You need to create:
a heather file (.h) with the following functions

- MaxPool that receives a matrix (assuming square matrix) as an input and returns the pointer to a new matrix of smaller size.
- AvgPool that receives a matrix (assuming square matrix) as an input and returns the pointer to a new matrix of smaller size.
- ReLU6 that updates the input matrix, overriding the values that are smaller to 0 or bigger than 6.
- Convolution that receives a matrix (square and multiple of 4x4) and a filter (4x4) of floating point and returns the result matrix of convolution.

An OpenMP file with the implementation of the functions

A main file for testing the functions. The main function will:
- read the input matrix from a text file (matrix.txt)
- apply MaxPool to the input and save the result in a file
- apply AvgPool to the input and save the result in a file
- apply ReLu6 to the input and save the matrix in a file
- read the kernel from a text file (kernel.txt)
- apply convolution and save the result in a file

For each of the four operations *measure the execution time of the sequential implementation* (to simply, simply set the number of threads to 1) *and of the best parallel implementation* (degree of parallelism and threads distribution) you achieve. Write your consideration in a **PDF document to add to the submission.**

********** extra points **********

To gain extra points, you can:
1 - for the convolution operation discuss the threads distribution and degree of parallelism dependence on the input size matrix.

2 - consider the "zero padding", by performing convolution in the whole input matrix, till the last column and the last row. Please do not add 3 extra rows and 3 extra columns of zeros in the input matrix but try more smart solutions. The output matrix will have the same size as the input matrix.