Programme Name: **Kuwahara filter using Numba Cuda**

Course Code:

Course Name: **Advanced HPC**

Assignment: **Final project**

Date of Submission: **08 january 2023**

Submitted By: Josselin Peniguel

Student Name: **Josselin Peniguel**

IUKL ID:

Semester: **3rd Semester**

Intake: **October 2022**

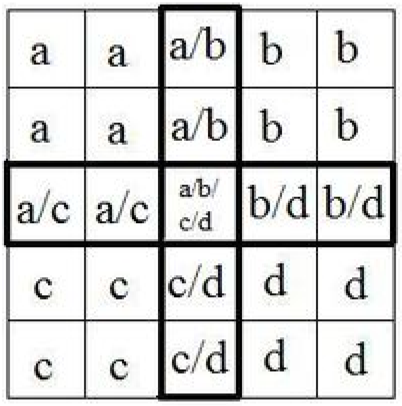
Submitted To: Dr. Tran Giang Son

Faculty Name: **USTH**

Department: **ICT**

1 Introduction

The Kuwahara filter is a digital image processing technique that is used to smooth images and reduce noise. It is named after its inventor, Nobuyuki Kuwahara. The Kuwahara filter works by dividing the image into a grid of overlapping regions, and then calculating the mean and standard deviation of the pixel values within each region. The filter then replaces the pixel values in each region with the mean value, resulting in a smoothed image. This filter, is particularly effective for preserving the edges in an image, as it does not smooth across regions with large gradients. As such, it is often used in applications where edge preservation is important, such as medical imaging and computer vision. For this project, we had to code a version of this filer using the Numba library in python, that allows us to use the GPU acceleration, in our python programs.



This image shows the spliting of the defined sized windo into four subwindows.

Numba is a just-in-time compiler for Python that can be used to optimize code for performance. Numba includes support for running on CUDA-enabled GPUs, which allows for significant acceleration of code that can be parallelized. To use numba to accelerate image filtering in Python, we can define a function that will implement the filtur and then use the numba.cuda.jit decorator. It allows us to compile the function for execution on the GPU. This can be useful because GPUs are designe to perform many calculations in parallel, and image processing algorithms often involve large amounts of data that can be processed simultaneously. By using numba cuda to accelerate the implementation of the filter, we can significantly reduce the time it takes to process the image. Numba will automatically parallelize the function by dividing the input data into chunks and assigning each chunk to a separate thread for processing. In the case of an image filter, this means that each thread can be responsible for processing a group of pixels in the image.using numba cuda can significantly reduce the time it takes to apply the filter to the image. This is because each thread can work independently on a different group of pixels, allowing the GPU to process multiple pixels at the same time. In contrast, when running the filter on a CPU, each pixel must be processed sequentially, which can be much slower.

2 Ideas and strategy

When thinking about this project, I knew that it was kind of hard debugging using numba. This is why I first started to develop the project using the CPU. To do this, I first started o code this type of code, which makes a lot of iterations and matrices creation :

```
def submatrices(matrix):
    submatrices = []
    mean = 0
    submean = []
    templist = []
    std = []
    for i in range(0, len(matrix) - 2, (len(matrix) // 2)):
        for j in range(0, len(matrix[0]) - 2, (len(matrix) // 2)):
            submatrix = []
            for row in matrix[i : i + ((len(matrix) // 2) + 1)]:
                submatrix.append(row[j : j + ((len(matrix) // 2) + 1)])
                for el in row[j : j + ((len(matrix) // 2) + 1)]:
                    mean += el
                    templist.append(el)

            std.append(np.std(templist))
            submatrices.append(submatrix)
            submean.append(mean / ((len(matrix) // 2 + 1) ** 2))
            mean = 0
            templist = []
    return submean, std, submatrices


matrix = [
    [1, 2, 3, 4, 5, 6, 7],
    [8, 9, 10, 11, 12, 13, 14],
    [15, 16, 17, 18, 19, 20, 21],
    [22, 23, 24, 25, 26, 27, 28],
    [29, 30, 31, 32, 33, 34, 35],
    [36, 37, 38, 39, 40, 41, 42],
    [43, 44, 45, 46, 47, 48, 49],
]

matrix2 = [
    [1, 2, 3, 4, 5, 6, 7],
    [8, 9, 10, 11, 12, 13, 14],
    [15, 16, 17, 18, 19, 20, 21],
    [22, 23, 24, 25, 26, 27, 28],
    [29, 30, 31, 32, 33, 34, 35],
    [36, 37, 38, 39, 40, 41, 42],
    [43, 44, 45, 46, 47, 48, 49],
]



mean, std, matrices = submatrices(matrix)
print(mean)
print(std)
print(matrices)

min_std = min(std)
index_min = std.index(min_std)
mean, std, matrices = submatrices(matrix2)
mean_RGB = mean[index_min]
print(mean_RGB)
```

This code works pretty okay, but it is very very long to run and not efficient at all. So I decided to move on to GPU use pretty quickly. I am now going to describe the code that you can find in this repository.

My final code, on the other end, converts a given input image in the RGB color space to the HSV color space using the rgb_to_hsv_kernel function. It then applies a Kuwahara filter to the image in the HSV color space using the kuwahara_filter_kernel function.The rgb_to_hsv_kernel function is a CUDA kernel that converts an image from the RGB color space to the HSV color space in parallel on the GPU. It takes in an input image in the RGB color space and an output image in the HSV color space, and converts each pixel in the input image to the corresponding pixel in the output image.

The kuwahara_filter_kernel function is also a CUDA kernel that applies the Kuwahara filter to an image in the HSV color space in parallel on the GPU. It takes in three input images: an image in the RGB color space, an image in the HSV color space, and an output image in the RGB color space. It also takes in a window size parameter, which specifies the size of the overlapping subwindows to use when applying the filter. Both the rgb_to_hsv_kernel and kuwahara_filter_kernel functions are decorated with the @cuda.jit.

The kuwahara_filter_kernel function uses the standard deviation function to calculate the minimum SD of the four subwindows, and then assigning the mean RGB value of this subwindow to the 'current' pixel. Here is the formula I used :

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

where:

- σ is the standard deviation
- N is the number of values in the sample
- x_i is the i^{th} value in the sample
- μ is the mean of the values in the sample

I coded this function for each pixel parallely treated by different thread like this :

- x = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
- y = cuda.threadIdx.y + cuda.blockIdx.y * cuda.blockDim.y

Thanks to these 2 coordinates, I was able to create 2 nested loops per subwindow, and then stored inside a tuple variable the std of each of the subwindows. I then selected the window where the std was the minimum and computed the RGB mean of this window, directly in the output RGB image.

3 Results and optimisation

Overall, this project was very interesting and harder that it seemed. I encountered a lot of obstacles and difficulties at the beginning because I was not going in the right direction. Indeed, I had a hard time understanding how to convert my CPU algorithm to my GPU algorithm. In the end, I succeeded at getting interesting and working GPU algorithm. Meanwhile my code is working great, there are a few optimizations I could think about while coding. I think the loops I am doing to calculate the different stds are great, but could be more efficient. Indeed, there are many loops, and I think this could be avoided if we change slightly the formula used (standard deviation), to something that doesn't need a 'temp' variable stored on the computer, during all the iterations. Reducing this formula would greatly reduce the number of loops. But this would also slightly change the algorithm. Finally, making a working shared memory algorithm could also be a good optimisation for this code, because it would greatly reduce the number of stds and RGB mean for the different subwindows that we have to calculate. It would allow the thread to communicate the subwindows mean and std values to eachother. Unfortunatly, I didn't have time to try this out.