

Compte rendu TP CHP programmation avec GPU

Aymen IMAD

2024-05-31

Objectif du TP:

L'objectif du TP consiste à introduire et à appliquer la technologie OpenCL pour le traitement d'images, en utilisant des plateformes de calcul hétérogènes comme les CPU et les GPU. Le TP est structuré en plusieurs étapes, dont la principale est la manipulation d'images par des filtres de convolution. L'objectif est de commencer par comprendre et compiler un code de base fourni, qui copie une image. Ensuite, de mettre en œuvre deux types de filtres : un filtre moyen et un filtre gaussien, en codant et optimisant ces filtres pour une exécution rapide. Le but final est d'optimiser le code pour maximiser la vitesse d'exécution tout en maintenant l'exactitude des résultats.

Etape 1: Filtre moyenneur:

Pour donner un peu de contexte dans notre environnement de travail on posséde une “directory” nomée chp_gpu qui contient un fichier imageCopyFilter.cpp qui posséde les paramètre de lecture et écriture d'image et un autre fichier copyimage.cl qui posséde tout noyaux (Kernel) utilisés. On posséde une image manet.png et le résultat sera stocké dans l'image resultat.png .

implémentation:

pour une image si $f(i,j)$ donne la valeur du pixel à la position (i,j) (un 4-uplet [rouge, vert, bleu, transparence]) alors appliquer un filtre moyenneur serait d'appliquer la transformation suivante (où N la moitié de taille de filtre):

$$g(i,j) = \frac{\sum_{k=-N}^{N} \sum_{l=-N}^{N} f(i+k, j+l)}{(2N+1)^2}$$

Le filtre moyenneur-approche naïve:

une approche naïve du code consiste à coder le Kernel de la façon suivante:

```

kernel void moyenne_image_naive(__global const unsigned char *imageInput,
                                __global      unsigned char *imageOutput)
{
    // Get the index of the current element to be processed
    const int2 coord = (int2)(get_global_id(0), get_global_id(1));
    int rowOffset = coord.y * width * 4;
    int index = rowOffset + coord.x * 4;

    float mean0 = 0.0;
    float mean1 = 0.0;
    float mean2 = 0.0;
    float mean3 = 0.0;
    int compteur = 0;

    if ((coord.x - HALF_FILTER_SIZE >= 0) && (coord.x + HALF_FILTER_SIZE < width)
        && (coord.y - HALF_FILTER_SIZE >= 0) && (coord.y + HALF_FILTER_SIZE < height)) {
        for (int i = coord.x - HALF_FILTER_SIZE; i <= coord.x + HALF_FILTER_SIZE; i++) {
            for (int j = coord.y - HALF_FILTER_SIZE; j <= coord.y + HALF_FILTER_SIZE; j++) {
                rowOffset = j * width * 4;
                index = rowOffset + i * 4;
                mean0 = mean0 + (float)(imageInput[index]);
                mean1 = mean1 + (float)(imageInput[index + 1]);
                mean2 = mean2 + (float)(imageInput[index + 2]);
                mean3 = mean3 + (float)(imageInput[index + 3]);
                compteur++;
            }
        }
        mean0 = mean0 / compteur;
        mean1 = mean1 / compteur;
        mean2 = mean2 / compteur;
        mean3 = mean3 / compteur;

        rowOffset = coord.y * width * 4;
        index = rowOffset + coord.x * 4;

        imageOutput[index]      = (int)(mean0);
        imageOutput[index + 1] = (int)(mean1);
        imageOutput[index + 2] = (int)(mean2);
        imageOutput[index + 3] = (int)(mean3);
    }
    else {
        if (coord.x < width && coord.y < height) {
            imageOutput[index]      = imageInput[index];
            imageOutput[index + 1] = imageInput[index + 1];
            imageOutput[index + 2] = imageInput[index + 2];
            imageOutput[index + 3] = imageInput[index + 3];
        }
    }
}

```

voici les illustration de l'image manet.png avec une taille du filtre N=10 et N=50



Figure 1: image originale



Figure 2: filtre moyenneur naïf avec N=10

Le filtre moyenneur avec uchar4:

Le type **uchar4** est un type spécifique souvent utilisé dans des bibliothèques de calcul parallèle comme CUDA (Compute Unified Device Architecture) pour manipuler des pixels RGBA de manière plus efficace. **uchar4** regroupe quatre canaux (R, G, B, A) en un seul type, ce qui permet d'effectuer des opérations sur quatre valeurs simultanément. Le kernel moyenne_image_vect s'écrit comme suit:

```
kernel void moyenne_image_vect(__global const uchar4 *imageInput,
                               __global uchar4 *imageOutput)
{
    const int2 coord = (int2)(get_global_id(0), get_global_id(1));
    int rowOffset = 0;
    int index = 0;

    float4 mean = (float4) (0,0,0,0);
    int compteur = 0;
```



Figure 3: filtre moyenneur naïf avec N=50

```

if ( (coord.x>HALF_FILTER_SIZE) &&
    (width-coord.x>HALF_FILTER_SIZE) &&
    (coord.y>HALF_FILTER_SIZE) &&
    (height-coord.y>HALF_FILTER_SIZE) )
{
    for (int i = coord.x - HALF_FILTER_SIZE; i < coord.x + HALF_FILTER_SIZE+1; i++) {
        for (int j = coord.y - HALF_FILTER_SIZE; j < coord.y + HALF_FILTER_SIZE+1; j++) {
            rowOffset = j * width;
            index = rowOffset + i;

            mean = mean + convert_float4(imageInput[index]);
            compteur = compteur+1;
        }
    }
    mean = mean/compteur;

    rowOffset = coord.y * width;
    index = rowOffset + coord.x;

    imageOutput[index] = convert_uchar4(mean);
}
else {
    rowOffset = coord.y * width;
    index = rowOffset + coord.x;

    imageOutput[index] = imageInput[index];
}
}

```

et de même pour N=10 et N=50 les images se présentent comme suit:

Comparaison entre les deux approches:

bien que pour le résultat, les images retournées paraissent identiques mais il y a une différence au niveau de temps pris pour l'exécution dans chacunes des deux cas, voici l'évolution du temps d'exécution des deux

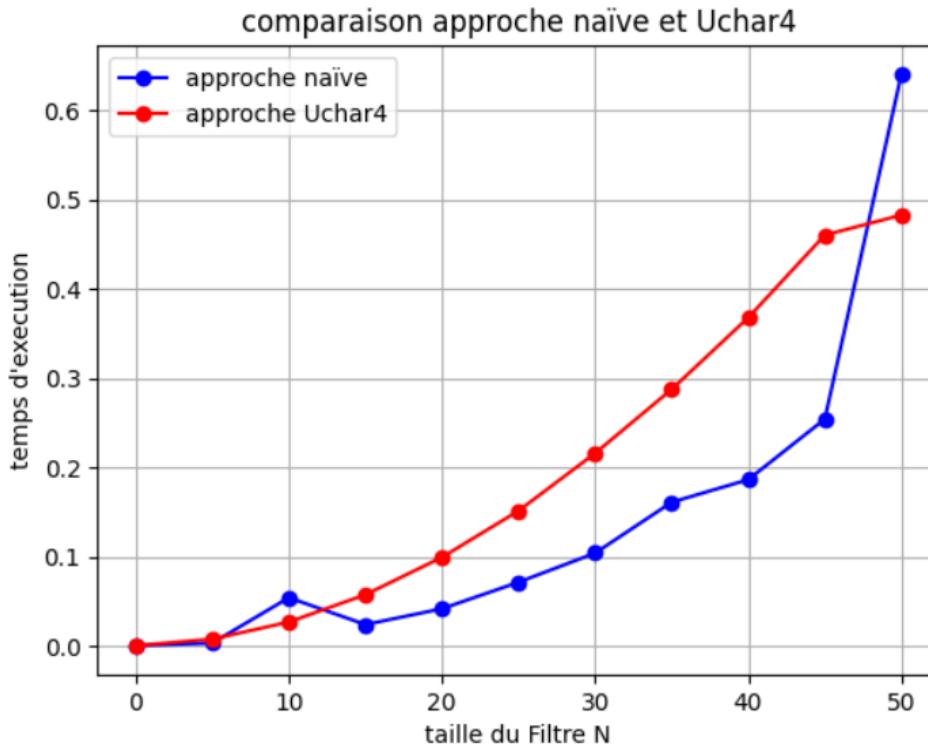


Figure 4: filtre moyenneur avec uchar4 pour N=10



Figure 5: filtre moyenneur avec uchar4 pour N=50

kernels, en fonction de N pour l'approche naïve et avec uchar4:



On remarque que les temps d'exécution sont relativement comparables pour N petit mais à partir de N=50 le temps d'exécution pour l'approche naïve explose, contrairement à l'approche Uchar4 qui elle continue d'augmenter mais d'une cadence faible., ce qui attendu puisque avec la méthode Uchar4 le traitement des images est procédé d'une façon parallèle et donc plus rapide.

Etape 2: Filtre gaussien:

implémentation:

De même, pour une image si $f(i,j)$ donne la valeur du pixel à la position (i,j) (un 4-uplet [rouge, vert, bleu, transparence]) alors appliquer un filtre gaussien serait d'appliquer la transformation suivante (où N la moitié de taille de filtre):

$$g(i, j) = \underbrace{\frac{1}{\sum_{k=-N}^N \sum_{l=-N}^N \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{k^2+l^2}{2\sigma^2}}} \sum_{k=-N}^N \sum_{l=-N}^N \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{k^2+l^2}{2\sigma^2}} f(i+k, j+l)}_{\text{facteur de normalisation}}$$

approche naïve du filtre:

Pour cette approche le code du Kernel est procédé comme suit:

```

kernel void gaussienne_image(__global const unsigned char *imageInput,
                            __global      unsigned char *imageOutput)
{
    // Get the index of the current element to be processed
    const int2 coord = (int2)(get_global_id(0), get_global_id(1));
    int rowOffset = coord.y * width * 4;
    int index = rowOffset + coord.x * 4;

    const float sigma = 10.0;
    const float beta = 1.0/(2.0*sigma*sigma);

    float4 gaussien = (float4) (0.0,0.0,0.0,0.0);
    float normalisation = 0.0;
    float coef;

    if ((coord.x - HALF_FILTER_SIZE >= 0) && (coord.x + HALF_FILTER_SIZE < width)
        && (coord.y - HALF_FILTER_SIZE >= 0) && (coord.y + HALF_FILTER_SIZE < height)) {
        for (int i = - HALF_FILTER_SIZE; i < HALF_FILTER_SIZE; i++) {
            for (int j = - HALF_FILTER_SIZE; j < HALF_FILTER_SIZE; j++) {
                rowOffset = (coord.y+j) * width * 4;
                index = rowOffset + (coord.x+i) * 4;

                coef = exp(-(float) (i*i+j*j)*beta);
                normalisation = normalisation + coef;

                gaussien = gaussien + coef * (float4)(imageInput[index],
                                                       imageInput[index + 1], imageInput[index + 2], imageInput[index + 3]);;
            }
        }
        gaussien = gaussien/normalisation;

        rowOffset = coord.y * width * 4;
        index = rowOffset + coord.x * 4;

        imageOutput[index]      = (int)(gaussien.x);
        imageOutput[index + 1] = (int)(gaussien.y);
        imageOutput[index + 2] = (int)(gaussien.z);
        imageOutput[index + 3] = (int)(gaussien.w);
    }
    else {
        if (coord.x < width && coord.y < height) {
            imageOutput[index]      = imageInput[index];
            imageOutput[index + 1] = imageInput[index + 1];
            imageOutput[index + 2] = imageInput[index + 2];
            imageOutput[index + 3] = imageInput[index + 3];
        }
    }
}

```

```

        }
    }
}
```

et en variant sigma de 0.1 à 1.0 à 10.0 et pour N prenant deux valeurs 10 et 50 les images sont présentée dans l'annexe:

approche moins naïve du Kernel:

pour cette approche on va parcourir i et j seulement entre 0 et N, et le code pour le kernet est codé comme suit:

```

kernel void gaussienne_image_naif(__global const unsigned char *imageInput,
                                   __global      unsigned char *imageOutput)
{
    // Get the index of the current element to be processed
    const int2 coord = (int2)(get_global_id(0), get_global_id(1));
    int rowOffset = coord.y * width * 4;
    int index = rowOffset + coord.x * 4;

    const float sigma = 0.1;
    const float beta = 1.0/(2.0*sigma*sigma);

    float4 gaussien = (float4) (0.0,0.0,0.0,0.0);
    float normalisation = 0.0;
    float coef;

    if ((coord.x - HALF_FILTER_SIZE >= 0) && (coord.x + HALF_FILTER_SIZE < width)
        && (coord.y - HALF_FILTER_SIZE >= 0) && (coord.y + HALF_FILTER_SIZE < height)) {
        for (int i = 0; i < HALF_FILTER_SIZE; i++) {
            for (int j = 0; j < HALF_FILTER_SIZE; j++) {
                rowOffset = (coord.y+j) * width * 4;
                index = rowOffset + (coord.x+i) * 4;

                coef = exp(-(float) (i*i+j*j)*beta);
                normalisation = normalisation + coef;

                gaussien = gaussien + coef * (float4)(imageInput[index],
                                                       imageInput[index + 1], imageInput[index + 2], imageInput[index + 3]);
            }
        }
        gaussien = gaussien/normalisation;

        rowOffset = coord.y * width * 4;
        index = rowOffset + coord.x * 4;

        imageOutput[index]      = (int)(gaussien.x);
        imageOutput[index + 1] = (int)(gaussien.y);
    }
}
```

```

        imageOutput[index + 2] = (int)(gaussien.z);
        imageOutput[index + 3] = (int)(gaussien.w);

    }

    else {
        if (coord.x < width && coord.y < height) {
            imageOutput[index] = imageInput[index];
            imageOutput[index + 1] = imageInput[index + 1];
            imageOutput[index + 2] = imageInput[index + 2];
            imageOutput[index + 3] = imageInput[index + 3];
        }
    }
}

```

approche optimisée du Kernel:

ici l'idée est tabuler dans un tableau de float les valeurs de $\exp(-k^2/(2\sigma^2))$ pour $0 \leq k \leq N$, et de faire des produits pour calculer le coefficient de ponderation $\exp((-k^2-l^2)/(2\sigma^2))$. Le code du Kernel se présente comme suit

```

kernel void gaussienne_image_coef(__global const uchar4 *imageInput,
                                  __global      uchar4 *imageOutput,
                                  __global const float *tab_coef)
{
    const int2 coord = (int2)(get_global_id(0), get_global_id(1));
    int rowOffset = 0;
    int index = 0;

    float4 gaussien = (float4) (0.0,0.0,0.0,0.0);
    float normalisation = 0.0;
    float coef;

    if ( (coord.x>HALF_FILTER_SIZE) &&
        (width-coord.x>HALF_FILTER_SIZE) &&
        (coord.y>HALF_FILTER_SIZE) &&
        (height-coord.y>HALF_FILTER_SIZE) )
    {
        for (int i = - HALF_FILTER_SIZE; i < HALF_FILTER_SIZE + 1; i++) {
        for (int j = - HALF_FILTER_SIZE; j < HALF_FILTER_SIZE + 1; j++) {
            rowOffset = (coord.y + j) * width;
            index = rowOffset + (coord.x + i);

            coef = tab_coef[abs(i)] * tab_coef[abs(j)];

            normalisation = normalisation + coef;
            gaussien = gaussien + coef*convert_float4(imageInput[index]);
        }
    }
}

```

```

        }

        gaussien = gaussien/normalisation;

        rowOffset = coord.y * width;
        index = rowOffset + coord.x;

        imageOutput[index] = convert_uchar4(gaussien);
    }
else {
    rowOffset = coord.y * width;
    index = rowOffset + coord.x;

    imageOutput[index] = imageInput[index];
}
}

```

dans le code imageCopyFilter.cpp on procéde comme suit en tabulant dans un tableau de float les valeurs de $\exp(-k^2/(2\sigma^2))$ pour $0 \leq k \leq N$. et puis on transfère ce tableau de la memoire du cpu vers celle du gpu avec clCreateBuffer On transfère ce tableau en argument avec SetKernelArg

```

/*status = clSetKernelArg(kernel, argIndex++, sizeof(cl_int), &image_width);
status = clSetKernelArg(kernel, argIndex++, sizeof(cl_int), &image_height);
status = clSetKernelArg(kernel, argIndex++, sizeof(cl_int), &filter_size);*/
status = clSetKernelArg(kernel, argIndex++, sizeof(cl_mem), (void *)&cl_tab_coef);

cout << "arguments assigned" << endl;

/*Step 10: Running the kernel.*/
size_t local_work_size[2] = {16, 16};
size_t global_work_size[2] = {RoundUp(local_work_size[0], image_width),
                           RoundUp(local_work_size[1], image_height)};

cl_event event=0;
cl_event *pevent=NULL;
cl_uint num_event =0;
status = clEnqueueNDRangeKernel(commandQueue, kernel, /* dimension of data*/ 2, /* offset */ NULL,
                                 global_work_size /*global_work_size*/, local_work_size/* local_work_size */,
                                 /* num_events */ num_event, /* wait_list */ pevent, /* event */ &event);
// ensure execution is finished
clWaitForEvents(1 , &event);
cl_ulong time_start, time_end;
double total_time;

clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START, sizeof(time_start), &time_start, NULL);
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END, sizeof(time_end), &time_end, NULL);
total_time = time_end - time_start;
cout << "Execution time: " << setprecision(4) << total_time/1E9 << " seconds" << endl;

/*Step 11: Read the output back to host memory.*/
unsigned char *buffer2 = new unsigned char[image_width * image_height * 4]();
// this initialization makes sure buffer2 is written
for (int i=0; i< image_width*image_height*channels; ++i)
    buffer2[i] = 0;

```

```

size_t origin[3] = { 0, 0, 0 };
size_t region[3] = { image_width, image_height, 1};

status = clEnqueueReadBuffer(commandQueue, cl_output_image, CL_TRUE, 0,
                             image_width*image_height*channels* sizeof(unsigned char),
                             buffer2, 0, NULL, NULL);

if (status != CL_SUCCESS)
{
    std::cerr << "Error reading result buffer." << std::endl;
    cerr << getErrorString(status) << endl;
    return FAILURE;
}

// save resulting image into a file
saveImageFile(output_name.c_str(), buffer2, image_width, image_height);

/*Step 12: Clean the resources.*/
delete data;
delete buffer2;
status = clReleaseKernel(kernel);           //Release kernel.
status = clReleaseProgram(program);         //Release the program object.
status = clReleaseMemObject(cl_image);       //Release mem object.
status = clReleaseMemObject(cl_output_image); //Release mem object.
status = clReleaseMemObject(cl_tab_coeff);   //Release mem object.
status = clReleaseContext(context);          //Release context.

if (devices != NULL)
{
    free(devices);
    devices = NULL;
}

if (rtnValue == SUCCESS)
    std::cout<<"Passed!\n";
else
    std::cout << " Error in computation!\n";

return rtnValue;
}

```

les images retournée seront située dans l'annexe.

Comparaison entre les approches:

La courbe compare les temps d'exécution de trois approches différentes en fonction de la taille du filtre N . L'approche naïve (en bleu) montre des temps d'exécution qui augmentent progressivement avec la taille du filtre, atteignant un pic notable pour $N = 50$, indiquant qu'elle est la plus lente pour les grandes tailles de filtre. L'approche moins naïve (en rouge) présente une amélioration par rapport à l'approche naïve, avec des temps d'exécution globalement plus bas et une augmentation plus modérée et stable. L'approche optimisée (en vert) commence avec des temps d'exécution très bas pour les petites tailles de filtre, mais son temps d'exécution augmente de manière significative à partir de $N = 20$, devenant même plus élevé que les deux

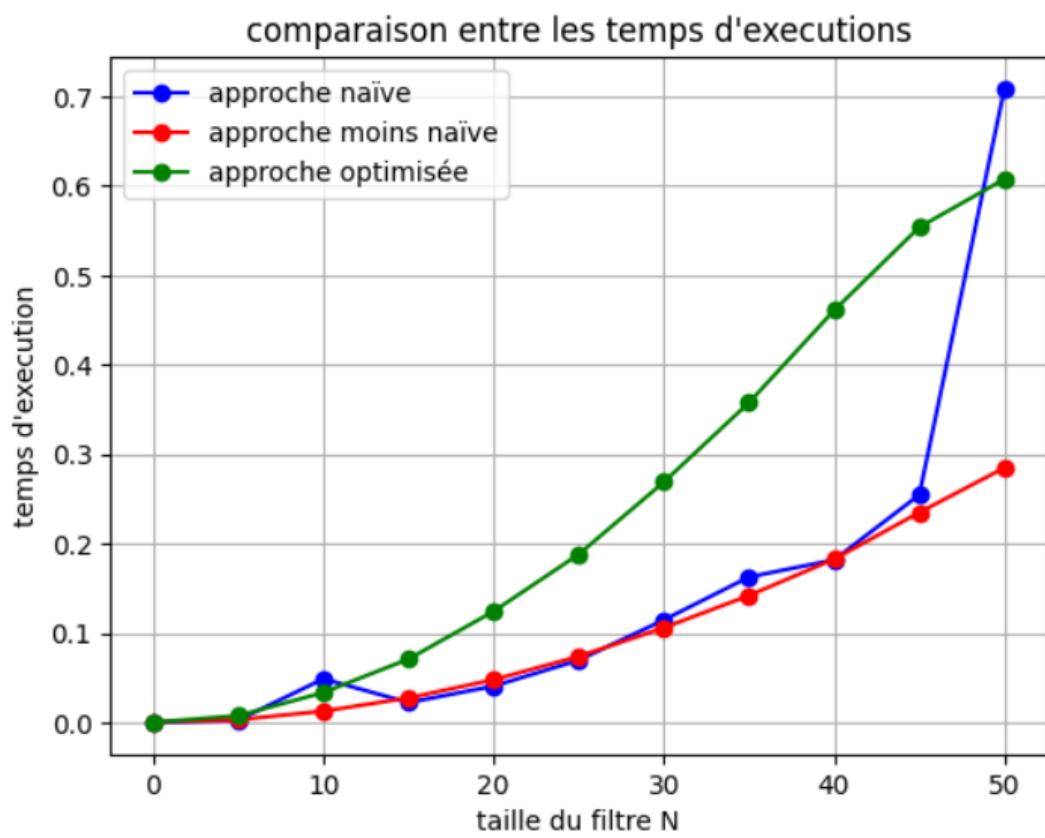


Figure 6: comparaison entre toutes les approches précédentes

autres approches pour les tailles de filtre les plus grandes. Ainsi, bien que l'approche optimisée soit la plus efficace pour les petites tailles de filtre, elle perd son avantage pour les filtres plus grands, où l'approche moins naïve devient la plus performante.

Annexes:



Filtre naïf N=10 sigma=0.1



Filtre naïf N=50 sigma=0.1



Filtre naïf N=10 sigma=1



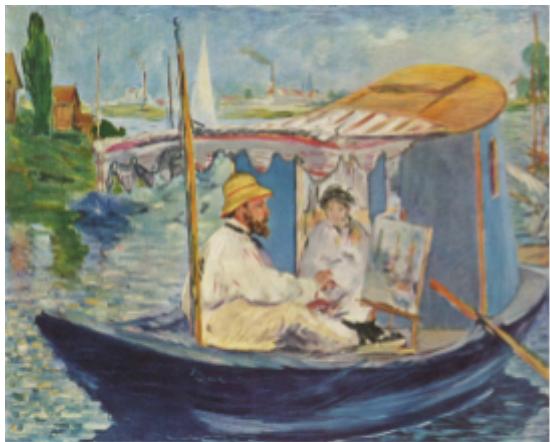
Filtre naïf N=50 sigma=1



Filtre naïf N=10 sigma=10



Filtre naïf N=50 sigma=10



Filtre moins naïf N=10 sigma=0.1



Filtre moins naïf N=50 sigma=0.1



Filtre moins naïf N=10 sigma=1



Filtre moins naïf N=50 sigma=1



Filtre moins naïf N=10 siama=10



Filtre moins naïf N=50 siama=10



Filtre optimisé N=10 sigma=0.1



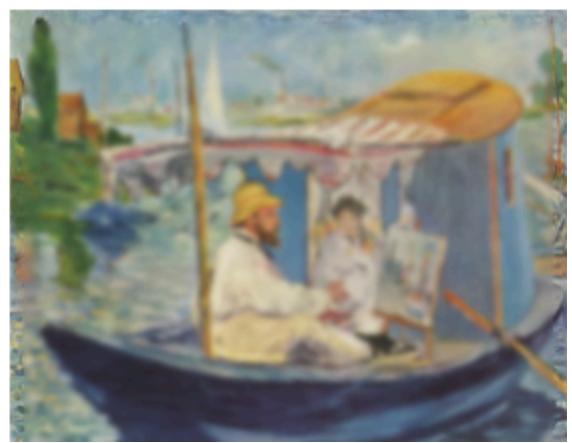
Filtre optimisé N=50 sigma=0.1



Filtre optimisé N=10 sigma=1



Filtre optimisé N=50 sigma=1



On remarque que plus sigma augmente plus le filtrage est performant.