

Rapport: Implémentation d'un algorithme de traitement d'image avec OpenCL

Malik Carpanin, Iliès Sghiar, Elias Zeghdar

Ecole des Mines de Saint-Etienne

{malik.carpanin, ilies.sghiar, elias.zeghdar}@etu.emse.fr

30 mai 2025

Introduction

Dans le cadre de ce TP, après avoir testé l'efficacité d'OpenCL sur des programmes de sommes et de produits de matrices, on souhaite utiliser ce dernier pour appliquer des filtres à une image représentée par une matrice carrée $n \times n$. À travers plusieurs implémentations – de la plus naïve à la plus optimisée – nous mettons en évidence les gains de performance apportés par une exécution sur GPU ainsi que les stratégies d'optimisation possibles à différents niveaux du code. Le cas d'étude final porte sur l'application de filtres moyenieur et gaussien à une image, en exploitant pleinement la puissance de calcul parallèle.



Somme de deux matrices

Etant données deux matrices $A \in \mathcal{M}_{n,n}(\mathbb{R})$ et $B \in \mathcal{M}_{n,n}(\mathbb{R})$, on souhaite calculer la somme $C = A + B$.

$$(i, j) \in [1, n] \quad C_{i,j} = A_{i,j} + B_{i,j} \quad (1)$$

On imlémentera le Kernel du code de la façon suivante :

```
1  __kernel void somme_gpu(__global const float* A,
2                           __global const float* B,
3                           __global float* C,
4                           const int DIM)
5 {
6     // Recuperer les indices globaux
7     int i = get_global_id(0);
8     int j = get_global_id(1);
9
10    // Vérifier que nous sommes dans les limites de la matrice
11    if (i < DIM && j < DIM) {
12        // Calcul de l'index linearisé
13        int index = i * DIM + j;
14
15        // Effectuer l'addition
16        C[index] = A[index] + B[index];
17    }
18}
19}
```

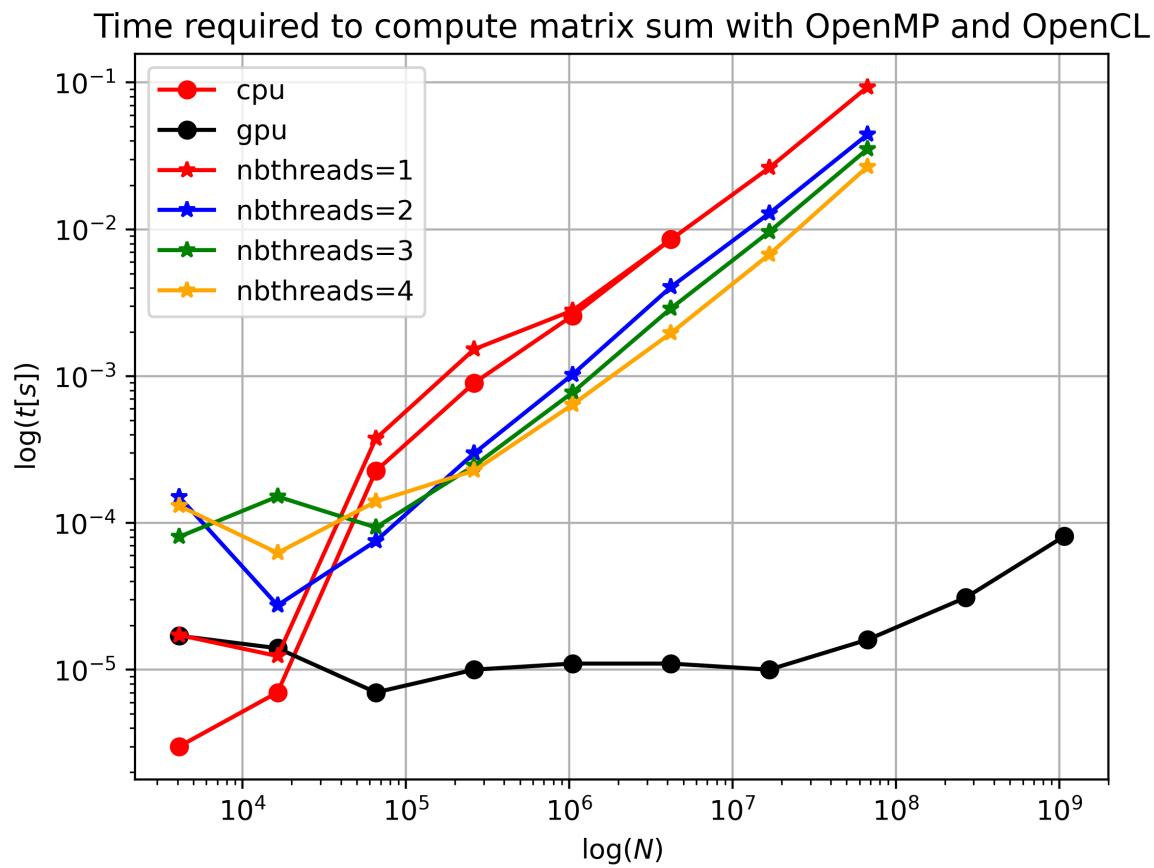


FIGURE 1 – Comparaison des temps d'exécution

Cette discréttisation permet de construire un système linéaire dont la solution approchera celle de l'équation de Poisson sur la grille considérée.

Produit de deux matrices

Etant données deux matrices $A \in \mathcal{M}_{n,n}(\mathbb{R})$ et $B \in \mathcal{M}_{n,n}(\mathbb{R})$, on souhaite calculer la somme $C = A \times B$.

$$(i, j) \in [1, n] \quad C_{i,j} = \sum_{k=1}^n A_{i,k} + B_{k,j} \quad (2)$$

On imlémentera le Kernel du code de la façon suivante :

```
1 __kernel void produit_gpu(__global const float* A,
2                           __global const float* B,
3                           __global float* C,
4                           const int DIM)
5 {
6     int row = get_global_id(0);
7     int col = get_global_id(1);
8
9     if (row < DIM && col < DIM) {
10         float sum = 0.0f;
11         for (int k = 0; k < DIM; k++) {
12             sum += A[row * DIM + k] * B[k * DIM + col];
13         }
14         C[row * DIM + col] = sum;
15     }
16 }
```

Time required to compute matrix product with OpenMP and OpenCL

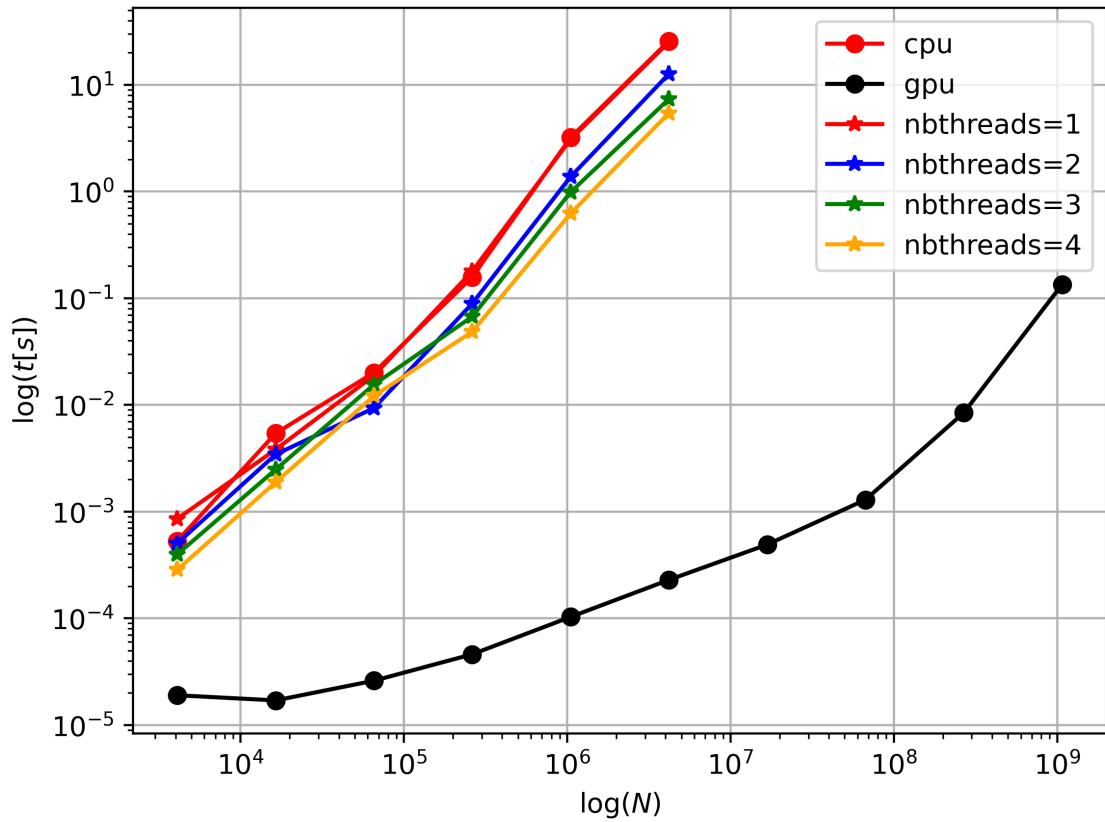


FIGURE 2 – Comparaison des temps d'exécution

Cette discréttisation permet de construire un système linéaire dont la solution approchera celle de l'équation de Poisson sur la grille considérée.

Filtre moyenneur

L'objectif du TP est d'appliquer un **filtre** à une image (une peinture de Manet).



FIGURE 3 – Enter Caption

On désigne par (i, j) la position du pixel courant et par N la moitié de la taille du filtre, carré de taille $(2N + 1) \times (2N + 1)$. Dans le cas du filtre moyenneur, celui-ci est formulé mathématiquement par la relation suivante :

$$g(i, j) = \frac{\sum_{k=-N}^N \sum_{l=-N}^N f(i + k, j + l)}{(2N + 1)^2} \quad (3)$$

```
1 __kernel void copy_image(__global const uchar *imageInput,
2                         __global uchar *imageOutput,
3                         int width, int height, int N)
4 {
5     int x = get_global_id(0);
6     int y = get_global_id(1);
7 }
```

```

8     if (x >= width || y >= height) return;
9
10    int sumR = 0;
11    int sumG = 0;
12    int sumB = 0;
13    int sumA = 0;
14    int count = 0;
15
16    // Parcourir un voisinage autour du pixel (x, y)
17    for (int dy = -N; dy <= N; dy++) {
18        for (int dx = -N; dx <= N; dx++) {
19            int nx = x + dx;
20            int ny = y + dy;
21
22            // Vérifier que le voisin est dans l'image
23            if (nx >= 0 && nx < width && ny >= 0 && ny < height) {
24                int nindex = (ny * width + nx) * 4;
25                sumR += imageInput[nindex];
26                sumG += imageInput[nindex + 1];
27                sumB += imageInput[nindex + 2];
28                sumA += imageInput[nindex + 3];
29                count++;
30            }
31        }
32    }
33
34    int index = (y * width + x) * 4;
35
36    // Calculer la moyenne pour RGBV
37    imageOutput[index] = (uchar)(sumR / count);
38    imageOutput[index + 1] = (uchar)(sumG / count);
39    imageOutput[index + 2] = (uchar)(sumB / count);
40    imageOutput[index + 3] = (uchar)(sumA / count);
41}

```

```

1 #define PI 3.14159f
2
3 __kernel void copy_image(__global const uchar4 *imageInput,
4                         __global uchar4 *imageOutput,
5                         int width, int height, int N_filtre)
6 {
7     int2 coord = (int2)(get_global_id(0), get_global_id(1));
8
9     if (coord.x < width && coord.y < height) {
10         float4 sum = (float4)(0.0f);
11
12         int nb_of_pixel = 0;
13
14         for (int i = -N_filtre; i <= N_filtre; i++) {
15             for (int j = -N_filtre; j <= N_filtre; j++) {
16                 int nx = coord.x + i;
17                 int ny = coord.y + j;
18
19                 if (nx >= 0 && nx < width && ny >= 0 && ny < height) {
20                     int idx = ny * width + nx;
21                     uchar4 pixel = imageInput[idx];
22                     sum += convert_float4(pixel);
23                     nb_of_pixel++;
24                 }
25             }
26         }
27     }
28
29     imageOutput[coord.x * height + coord.y] = sum;
30 }

```

```

25     }
26 }
27
28 // Moyenne des pixels du filtre
29 sum = sum / (float)nb_of_pixel;
30
31 // Conversion float4 vers uchar4
32 uchar4 result = convert_uchar4_rte(sum);
33
34 int out_idx = coord.y * width + coord.x;
35 imageOutput[out_idx] = result;
36
37 }

```

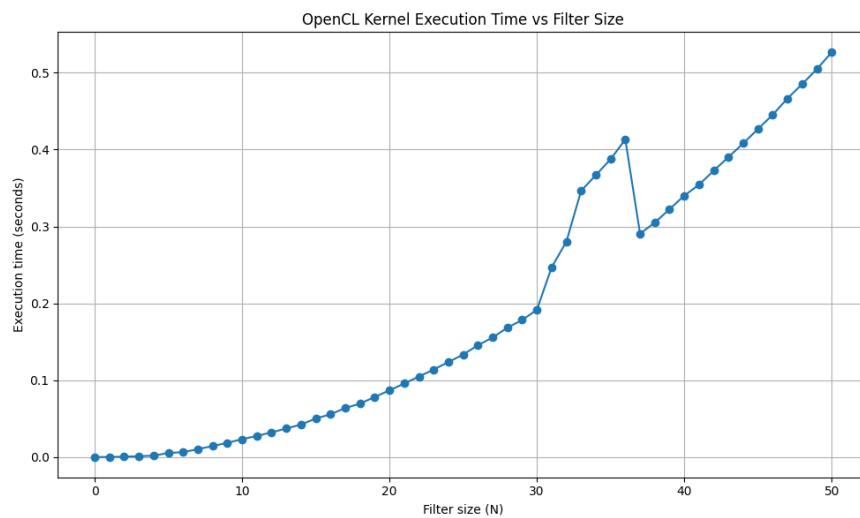


FIGURE 4 – Implémentation naïve

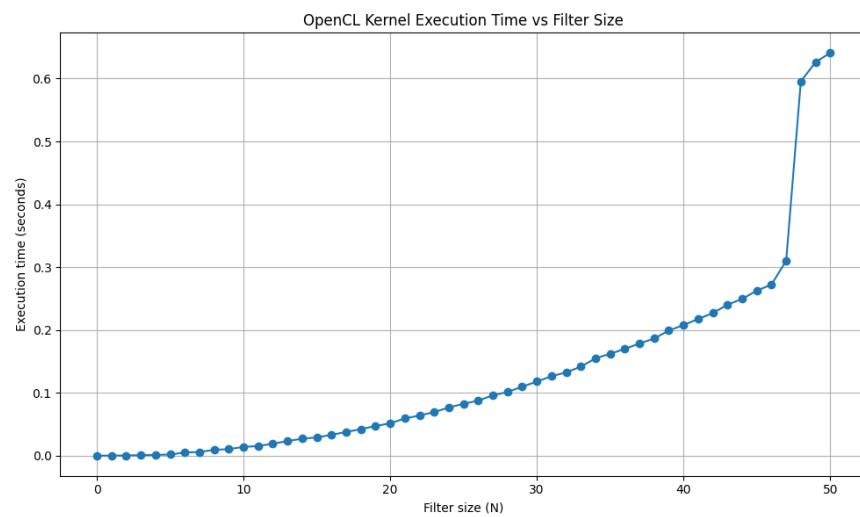


FIGURE 5 – Implémentation avec uchar4



FIGURE 6 – Résultats de l’application du filtre moyenieur avec différentes valeurs de N

Filtre Gaussien

Pour le filtre Gaussien, il est formulé mathématiquement par la relation

$$g(i, j) = \frac{1}{\underbrace{\sum_{k=-N}^N \sum_{l=-N}^N \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{k^2+l^2}{2\sigma^2}}}_{\text{facteur de normalisation}}} \sum_{k=-N}^N \sum_{l=-N}^N \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{k^2+l^2}{2\sigma^2}} f(i+k, j+l) \quad (4)$$

(a) Approche naïve dans le kernel

L’approche la plus naïve : on fait calculer à chaque work item N^2 exponentielles. C’est évidemment très gourmand en temps de calcul, mais fournit un premier résultat.

```

1 #define PI 3.14159f
2
3
4 __kernel void copy_image(__global const uchar4 *imageInput,
5     __global uchar4 *imageOutput,
6     int width, int height, int N_filtre)
7 {
8     int2 coord = (int2)(get_global_id(0), get_global_id(1));
9
10    if (coord.x < width && coord.y < height) {
11        float4 sum = (float4)(0.0f);
12
13        float denom = 0.0;
14        float sigma=1.0f; //1 10
15        float sigmasq=sigma*sigma;
16
17
18        for (int i = -N_filtre; i <= N_filtre; i++) {
19            for (int j = -N_filtre; j <= N_filtre; j++) {
20                int nx = coord.x + i;
21                int ny = coord.y + j;
22                int distancesq=i*i+j*j;
23                float localVal = (1.0f/(sqrt(2*PI)*sigma)) * exp(-distancesq/(2*
24                    sigmasq));
```

```

25     if (nx >= 0 && nx < width && ny >= 0 && ny < height) {
26         int idx = ny * width + nx;
27         uchar4 pixel = imageInput[idx];
28         sum += localVal*convert_float4(pixel);
29         denom+=localVal;
30     }
31 }
32 }

34 // Moyenne des pixels du filtre
35 sum = sum / (float)denom;

37 // Conversion float4 vers uchar4
38 uchar4 result = convert_uchar4_rte(sum);

40 int out_idx = coord.y * width + coord.x;
41 imageOutput[out_idx] = result;
42 }
43 }
```



(a) $N = 10 \quad \sigma = 0, 1$



(b) $N = 10 \quad \sigma = 1$



(c) $N = 10 \quad \sigma = 10$



(d) $N = 50 \quad \sigma = 0, 1$



(e) $N = 50 \quad \sigma = 1$



(f) $N = 50 \quad \sigma = 10$

FIGURE 7 – Application de l'approche naïve du g-filtre Gaussien

(b) Approche moins naïve dans le kernel

On commence à affiner, on se sert de la parité par rapport à k et l de chaque terme du facteur de normalisation. On divise par 4 le nombre de calcul mais on est toujours en $O(N^2)$ par work item.

```

1 #define PI 3.14159f
2
3 __kernel void copy_image(__global const uchar4 *imageInput,
```

```

4           __global      uchar4 *imageOutput,
5           int width, int height, int N_filtre)
6 {
7     int2 coord = (int2)(get_global_id(0), get_global_id(1));
8
9     if (coord.x < width && coord.y < height) {
10        float4 sum = (float4)(0.0f);
11
12        float denom = 0.0;
13        float sigma=2.0f;
14        float sigmasq=sigma*sigma;
15
16        for (int i = 0; i <= N_filtre; i++) {
17            for (int j = 0; j <= N_filtre; j++) {
18                int nxp = coord.x + i;
19                int nyp = coord.y + j;
20
21                int nxm =coord.x - i;
22                intnym = coord.y - j;
23                int distancesq=i*i+j*j;
24                float localVal = (1.0f/(sqrt(2*PI)*sigma)) * exp(-distancesq/(2*
25                               sigmasq));
26
27                if (nxp >= 0 && nxp < width && nyp >= 0 && nyp < height) {
28                    int idx = nyp * width + nxp;
29                    uchar4 pixel = imageInput[idx];
30                    sum += localVal*convert_float4(pixel);
31                    denom+=localVal;
32                }
33                if (nxm >= 0 && nxm < width &&nym >= 0 &&nym < height) {
34                    int idx =nym * width + nxm;
35                    uchar4 pixel = imageInput[idx];
36                    sum += localVal*convert_float4(pixel);
37                    denom+=localVal;
38                }
39                if (nxp >= 0 && nxp < width &&nym >= 0 &&nym < height) {
40                    int idx =nym * width + nxp;
41                    uchar4 pixel = imageInput[idx];
42                    sum += localVal*convert_float4(pixel);
43                    denom+=localVal;
44                }
45                if (nxm >= 0 && nxm < width && nyp >= 0 && nyp < height) {
46                    int idx = nyp * width + nxm;
47                    uchar4 pixel = imageInput[idx];
48                    sum += localVal*convert_float4(pixel);
49                    denom+=localVal;
50                }
51
52
53            }
54        }
55
56
57        // Moyenne des pixels du filtre
58        sum = sum / (float)denom;
59
60        // Conversion float4 vers uchar4
61        uchar4 result = convert_uchar4_rte(sum);

```

```

62     int out_idx = coord.y * width + coord.x;
63     imageOutput[out_idx] = result;
64 }
65 }
66 }
```

(c) Approche optimisée dans le kernel

On continue à optimiser : on calcul N exponentielles et on se sert de produits (bien plus rapide que le calcul d'exponentielles) pour tout calculer. On gagne un ordre de complexité.

```

1 #define PI 3.14159f
2
3 __kernel void copy_image(__global const uchar4 *imageInput,
4                         __global uchar4 *imageOutput,
5                         int width, int height, int N_filtre)
6 {
7     int2 coord = (int2)(get_global_id(0), get_global_id(1));
8
9     if (coord.x < width && coord.y < height) {
10         float4 sum = (float4)(0.0f);
11
12         float denom = 0.0;
13         float sigma=2.0f;
14         float sigmasq=sigma*sigma;
15         __private float expTab[100];
16
17         for (int i = 0; i<=N_filtre; i++) {
18             expTab[i]= exp(-(i*i)/(2*sigmasq));
19         }
20
21
22         for (int i = 0; i <= N_filtre; i++) {
23             for (int j = 0; j <= N_filtre; j++) {
24                 int nxp = coord.x + i;
25                 int nyp = coord.y + j;
26
27                 int nxm =coord.x - i;
28                 intnym = coord.y - j;
29                 int distancesq=i*i+j*j;
30                 float localVal = (1.0f/(sqrt(2*PI)*sigma)) * expTab[i]*expTab[j];
31
32                 if (nxp >= 0 && nxp < width && nyp >= 0 && nyp < height) {
33                     int idx = nyp * width + nxp;
34                     uchar4 pixel = imageInput[idx];
35                     sum += localVal*convert_float4(pixel);
36                     denom+=localVal;
37                 }
38                 if (nxm >= 0 && nxm < width &&nym >= 0 &&nym < height) {
39                     int idx =nym * width + nxm;
40                     uchar4 pixel = imageInput[idx];
41                     sum += localVal*convert_float4(pixel);
42                     denom+=localVal;
43                 }
44                 if (nxp >= 0 && nxp < width &&nym >= 0 &&nym < height) {
45                     int idx =nym * width + nxp;
46                     uchar4 pixel = imageInput[idx];
47                     sum += localVal*convert_float4(pixel);
48                 }
49             }
50         }
51     }
52 }
```

```

48         denom+=localVal;
49     }
50     if (nxm >= 0 && nxm < width && nyp >= 0 && nyp < height) {
51         int idx = nyp * width + nxm;
52         uchar4 pixel = imageInput[idx];
53         sum += localVal*convert_float4(pixel);
54         denom+=localVal;
55     }
56
57
58
59     }
60 }
61
62 // Moyenne des pixels du filtre
63 sum = sum / (float)denom;
64
65 // Conversion float4 vers uchar4
66 uchar4 result = convert_uchar4_rte(sum);
67
68 int out_idx = coord.y * width + coord.x;
69 imageOutput[out_idx] = result;
70 }
71 }
```

(d) Approche optimisée dans le code cpp

On cerne la plus grosse barrière de l'optimisation : refaire les calculs sur chaque work-items. Ici on décide de ne faire les calculs d'exponentielles qu'une fois pour tous les work-items, ce qui réduit drastiquement le nombre de calcul effectués.

```

1 #define PI 3.14159f
2
3 // kernel for copying an image using uchar4
4 __kernel void copy_image(__global const uchar4 *imageInput,
5                         __global uchar4 *imageOutput,
6                         int width, int height, int N_filtre, __global float *
7                           expTab, float sigma)
8 {
9     int2 coord = (int2)(get_global_id(0), get_global_id(1));
10
11    if (coord.x < width && coord.y < height) {
12        float4 sum = (float4)(0.0f);
13
14        float denom = 0.0;
15
16
17
18
19        for (int i = 0; i <= N_filtre; i++) {
20            for (int j = 0; j <= N_filtre; j++) {
21                int nxp = coord.x + i;
22                int nyp = coord.y + j;
23
24                int nxm = coord.x - i;
25                intnym = coord.y - j;
```

```

27     int distancesq=i*i+j*j;
28     float localVal = (1.0f/(sqrt(2*PI)*sigma)) * expTab[i]*expTab[j];
29
30     if (nxp >= 0 && nxp < width && nyp >= 0 && nyp < height) {
31         int idx = nyp * width + nxp;
32         uchar4 pixel = imageInput[idx];
33         sum += localVal*convert_float4(pixel);
34         denom+=localVal;
35     }
36     if (nxm >= 0 && nxm < width &&nym >= 0 &&nym < height) {
37         int idx =nym * width + nxm;
38         uchar4 pixel = imageInput[idx];
39         sum += localVal*convert_float4(pixel);
40         denom+=localVal;
41     }
42     if (nxp >= 0 && nxp < width &&nym >= 0 &&nym < height) {
43         int idx =nym * width + nxp;
44         uchar4 pixel = imageInput[idx];
45         sum += localVal*convert_float4(pixel);
46         denom+=localVal;
47     }
48     if (nxm >= 0 && nxm < width && nyp >= 0 && nyp < height) {
49         int idx = nyp * width + nxm;
50         uchar4 pixel = imageInput[idx];
51         sum += localVal*convert_float4(pixel);
52         denom+=localVal;
53     }
54
55
56
57     }
58 }
59
60 // Moyenne des pixels du filtre
61 sum = sum / (float)denom;
62
63 // Conversion float4 vers uchar4 (saturation automatique)
64 uchar4 result = convert_uchar4_rte(sum);
65
66 int out_idx = coord.y * width + coord.x;
67 imageOutput[out_idx] = result;
68 }
69 }
```

(e) Optimisation accès mémoire constante au lieu de globale

On optimise une dernière fois, en réduisant le temps d'accès au tableau des exponentielles en plaçant cette dernière comme variable constante (stockée donc dans la mémoire la plus rapide d'accès pour les work-items).

```

1 #define PI 3.14159f
2
3 // kernel for copying an image using uchar4
4 __kernel void copy_image(__global const uchar4 *imageInput,
5                         __global uchar4 *imageOutput,
6                         int width, int height, int N_filtre, __constant float *
7                           expTab, float sigma)
```

```

8     int2 coord = (int2)(get_global_id(0), get_global_id(1));
9
10    if (coord.x < width && coord.y < height) {
11        float4 sum = (float4)(0.0f);
12
13        float denom = 0.0;
14
15
16
17
18
19
20        for (int i = 0; i <= N_filtre; i++) {
21            for (int j = 0; j <= N_filtre; j++) {
22                int nxp = coord.x + i;
23                int nyp = coord.y + j;
24
25                int nxm = coord.x - i;
26                intnym = coord.y - j;
27                int distancesq=i*i+j*j;
28                float localVal = (1.0f/(sqrt(2*PI)*sigma)) * expTab[i]*expTab[j];
29
30                if (nxp >= 0 && nxp < width && nyp >= 0 && nyp < height) {
31                    int idx = nyp * width + nxp;
32                    uchar4 pixel = imageInput[idx];
33                    sum += localVal*convert_float4(pixel);
34                    denom+=localVal;
35                }
36                if (nxm >= 0 && nxm < width &&nym >= 0 &&nym < height) {
37                    int idx =nym * width + nxm;
38                    uchar4 pixel = imageInput[idx];
39                    sum += localVal*convert_float4(pixel);
40                    denom+=localVal;
41                }
42                if (nyp >= 0 && nyp < width &&nym >= 0 &&nym < height) {
43                    int idx =nym * width + nyp;
44                    uchar4 pixel = imageInput[idx];
45                    sum += localVal*convert_float4(pixel);
46                    denom+=localVal;
47                }
48                if (nxm >= 0 && nxm < width && nyp >= 0 && nyp < height) {
49                    int idx = nyp * width + nxm;
50                    uchar4 pixel = imageInput[idx];
51                    sum += localVal*convert_float4(pixel);
52                    denom+=localVal;
53                }
54
55
56            }
57        }
58
59        // Moyenne des pixels du filtre
60        sum = sum / (float)denom;
61
62        // Conversion float4 vers uchar4 (saturation automatique)
63        uchar4 result = convert_uchar4_rte(sum);
64
65        int out_idx = coord.y * width + coord.x;

```

```

67     imageOutput[out_idx] = result;
68 }
69 }
```

(f) Comparaison des approches

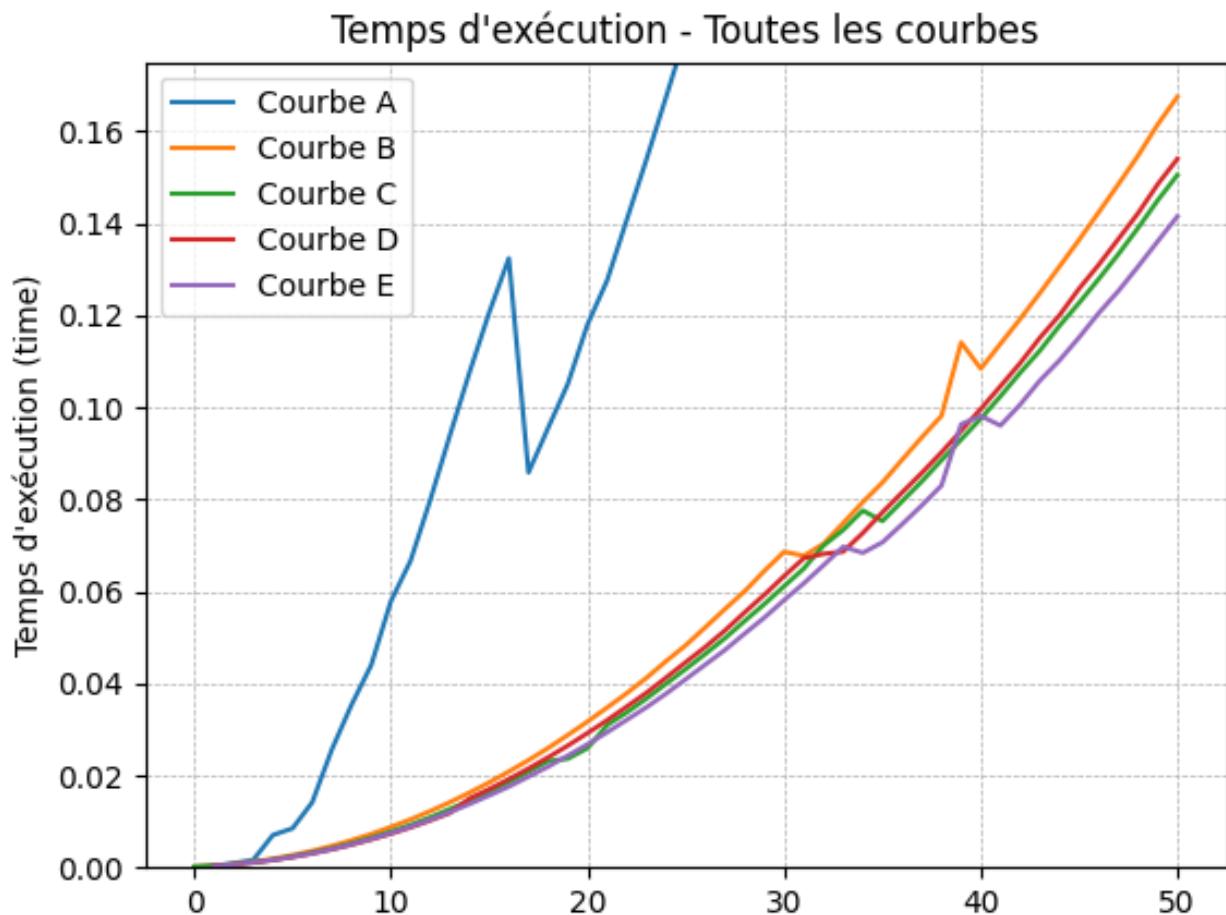


FIGURE 8 – Enter Caption

Etant donné le résultat plutôt suprenant que l'approche optimisée dans le code cpp (d) semble moins efficace que l'approche optimisée dans le kernel (c), on a voulu effectuer d'autres test qui ont au final confirmé celui-ci (voir graphe suivant). Au final Il s'explique probablement par des questions de hardware. On remarque tout de même que pour toutes les sections, des 'pics' de temps d'exécution apparaissent : selon nous, ceux-ci s'expliquent par le grand nombre d'utilisateur simultané. On notera pour information que le temps d'exécution passe du simple au double pour un même N et un même programme, justement à cause de ce grand nombre d'utilisateurs.

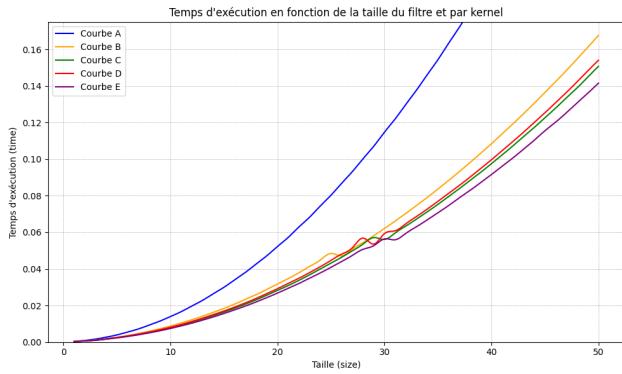


FIGURE 9 – Enter Caption

Conclusion

Ce TP nous a permis d'explorer concrètement les avantages du calcul parallèle sur GPU pour des opérations de traitement d'image. Les différentes implémentations testées – de la somme de matrices à l'application de filtres complexes – ont mis en évidence l'importance cruciale de l'optimisation, tant au niveau algorithmique qu'en ce qui concerne la gestion de la mémoire. Nous avons ainsi constaté que des choix judicieux, comme la pré-calculation des poids ou l'utilisation de la mémoire constante, peuvent considérablement améliorer les performances.