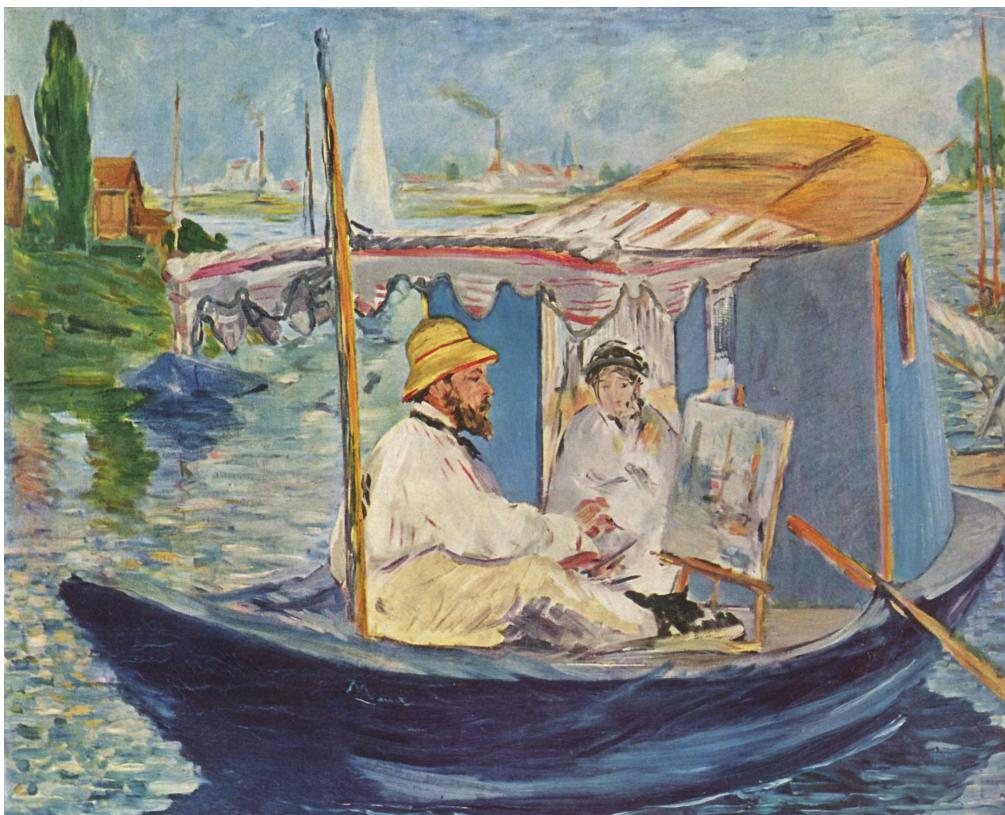


## TRAITEMENT D'IMAGE AVEC OPEN-CL :

### Introduction :

Le traitement d'image étant très couteux, ce TP a pour but d'optimiser le calcul filtrage d'une image. En effet, si on multiplie le nombre de pixels par la taille du filtre que nous allons programmer, les calculs s'alourdissent très vite et rendent le programme peu performant. Pour résoudre ce problème, nous avons utiliser le GPU (ayant plus de threads que le CPU) qui va parcourir l'image et la traiter en faisant les calculs en parallèles. Le filtre est ici, un filtre de convolution, que nous avons parallélisé afin de l'optimiser.



### Remarque :

J'ai réalisé les calculs de temps sur mon ordinateur personnel, un MacBook Pro.  
Néanmoins j'ai changé le utils.hpp dans le compte rendu pour qu'il compile sous linux.

### Plan :

**1/ Premier filtre : filtre moyenneur sur une taille N : mean\_filter**

**2/ Deuxième filtre : filtre GAUSSIEN :**

2.0/ Filtre Gaussien : filtre non optimisé : gauss\_filter

2.1/ Filtre Gaussien : première optimisation : gauss\_filter1

2.2/ Filtre Gaussien : deuxième optimisation : gauss\_filter2

2.3/ Filtre Gaussien : troisième optimisation : gauss\_filter3

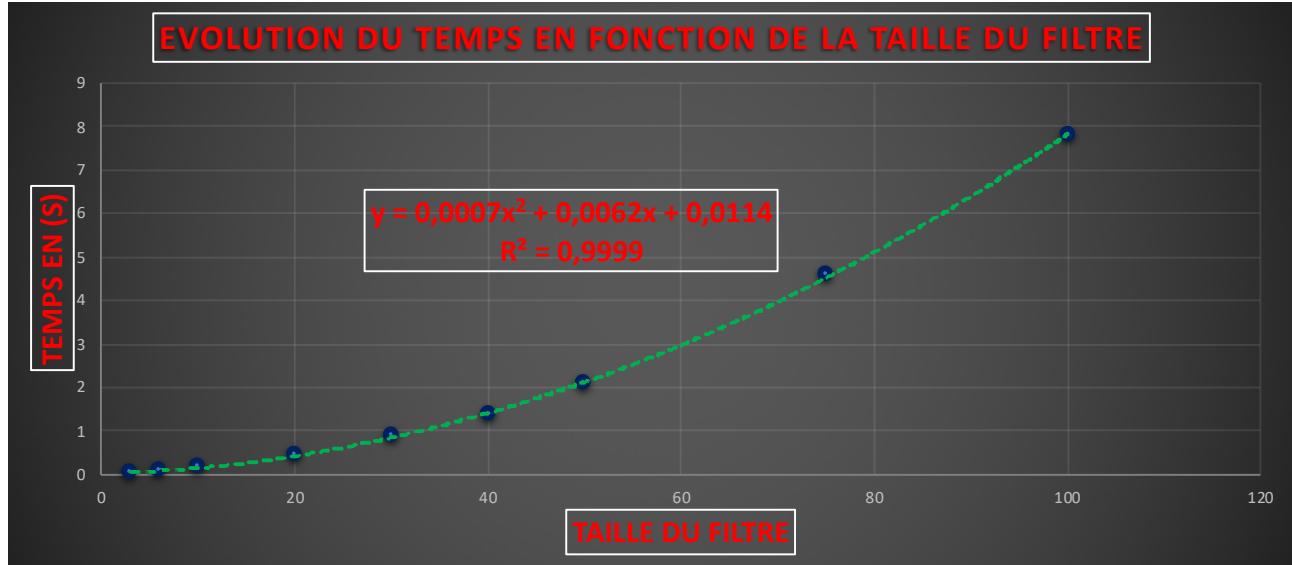
2.4/ Filtre Gaussien : troisième optimisation : gauss\_filter4

## 1/ Premier filtre : filtre moyenneur sur une taille N : mean\_filter

On obtient alors le résultat suivant :



Pour cette photo, on moyenne sur 10 cases (taille du filtre), le temps vaut : 0.1604 seconds



## 2/ Deuxième filtre : filtre GAUSSIEN :

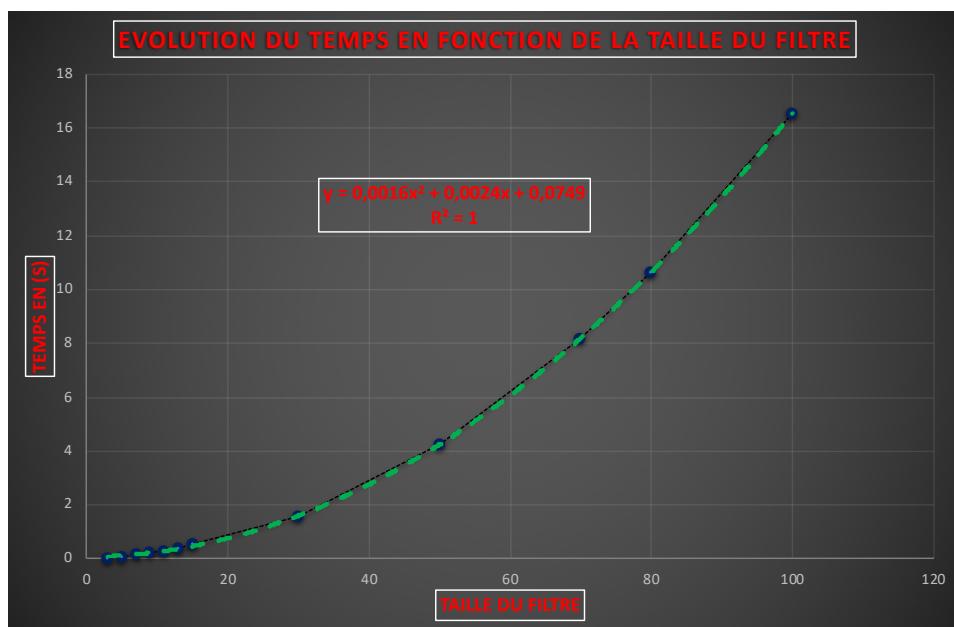
Remarque :

Pour optimiser mes programmes je suis parti du principe que l'accès à la mémoire était moins couteux que le calcul et que les mémoires locales sont plus rapide que les mémoires globales, ainsi pour optimiser j'ai donc essayé de réaliser le plus de calcul possible dans le programme principal.

Programmation Open CL

Nous avons fait des comparaisons pour des filtres de taille en largeur de taille 3 à 15 (filtre donc de taille 3\*3 à 15\*15).

D'après la littérature, la valeur optimale pour sigma est  $(N-1)/6$  (avec N la largeur du filtre), pour garder le plus de détails sans avoir trop de flou.

2.0/ Filtre Gaussien : filtre non optimisé : gauss\_filter :

Remarque : Le filtre Gaussien a une complexité temporelle plus importante que le filtre moyenneur, vu les calculs réalisé cela est logique.

Observation :

Même si les temps de calcul restent faibles, elle augmente comme on le voit ci-dessus rapidement avec la taille du filtre.

Dans la suite du TP nous allons étudier plus précisément le filtre Gaussien pour essayer de l'optimiser le plus possible pour avoir le temps de calcul le plus faible.

2.1/ Filtre Gaussien : première optimisation : gauss\_filter1 :

Amélioration : Pour réduire le nombre de calcul de  $1/\sqrt{2\pi}$ , il est ajouté en argument du kernel.

Example :

Pour ce résultat nous avons pris les données suivantes :

$$\text{Sigma} = 100$$

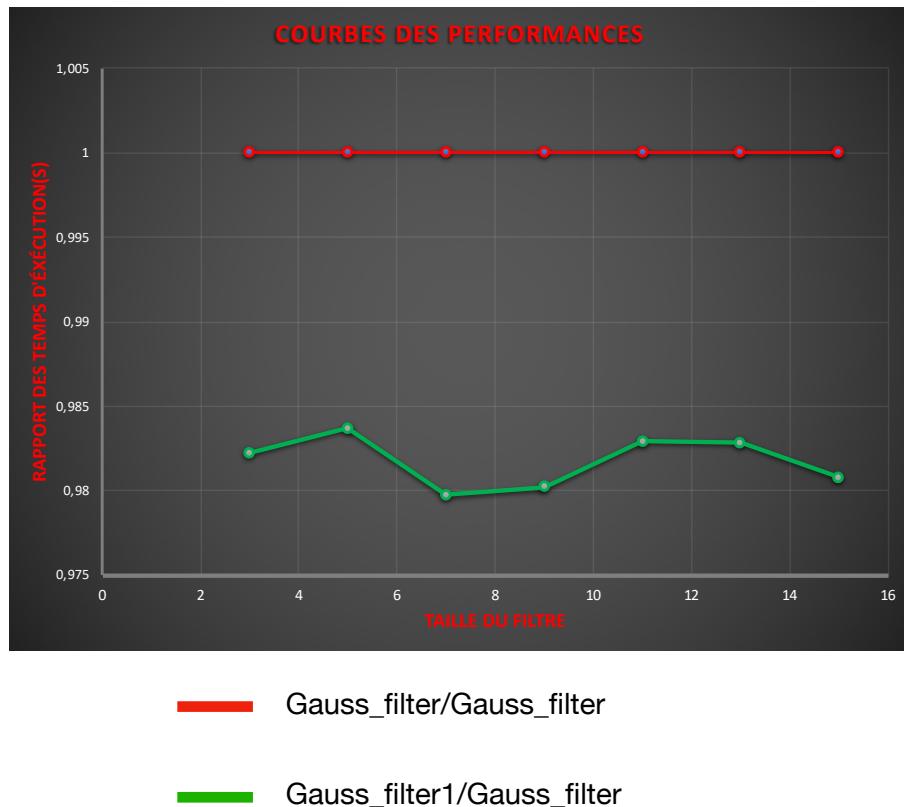
$$N = 10$$

Temps : 0.2859 seconds

3 sur 9

Pour garder une cohérence durant tous les calculs de temps nous avons pris les mêmes tailles de filtre  $N$  et  $\text{Sigma} = (N-1)/6$  pour pouvoir comparer aux mieux les programmes optimisent le filtre Gaussien.

Pour exploiter les résultats on a tracé le rapport entre la performance du filtre modifié (donc normalement optimisé) et celle du filtre dit « naif » en fonction de la taille du filtre.



Observation : Étant donné l'échelle des ordonnées, l'amélioration est très faible. On observe que l'amélioration s'améliore quand la taille du filtre augmente cela est du au fait que le calcul (se faisant sur le CPU) est plus rapide que les temps d'accès à la mémoire (en calculant sur les GPU).

Finalement le gain de temps venant de la parallélisation est perdu par les temps d'accès et d'envoie de calcul, il vaut mieux tout calculer en mémoire local.

## 2.2/ Filtre Gaussien : deuxième optimisation : gauss\_filter2 :

Amélioration : On garde l'amélioration du gauss\_filter1 et en plus on calcul les exponentiels avant le kernel. Pour ce faire on calcul dans un tableau de taille  $2*N+1$  l'exponentielle  $\exp(-l^2/(2*\text{sigma}^2))$ . On charge donc les calcul dans le kernel (qui sont donc initialement chargé par le tableau des calculs de l'exponentielle).



— Gauss\_filter/Gauss\_filter    — Gauss\_filter1/Gauss\_filter    — Gauss\_filter2/Gauss\_filter

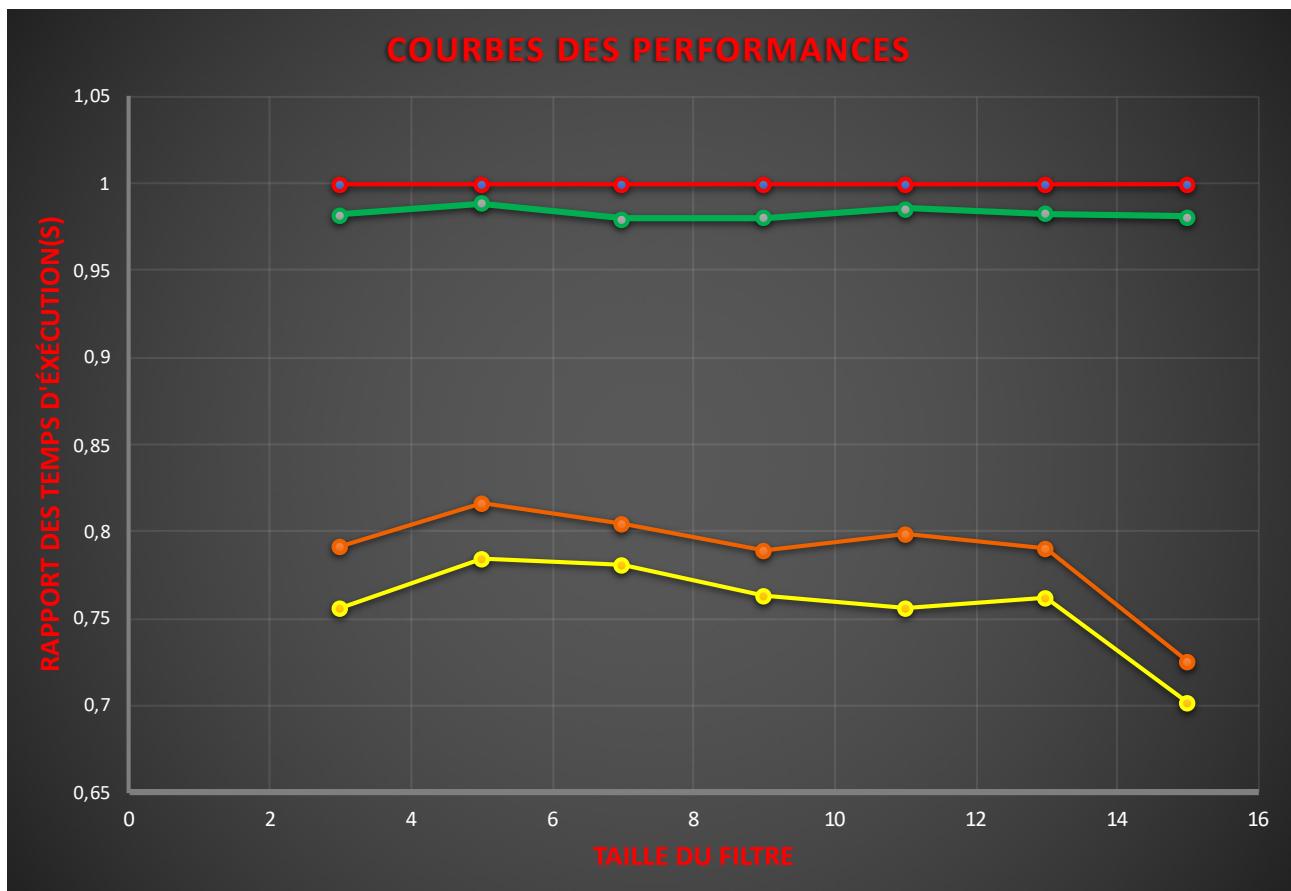
**Observation :** On a une nette augmentation des performances, réduit considérablement le temps de calcul, il s'améliore jusqu'à un facteur 0,75 du temps final, soit un gain de temps de 25% !

Cette forte amélioration est due à la reduction du nombre de calculs de l'exponentielle.

### 2.3/ Filtre Gaussien : troisième optimisation : gauss\_filter3 :

**Amélioration :** Ici on ajoute le calcul de l'exponentielle dans la mémoire locale car elle est plus rapide d'accès que la mémoire globale (en plus des améliorations précédentes). Le gain de temps doit être faible mais néanmoins observable.

**Remarque :** Pour chaque mesure du temps pour les tailles données j'ai réalisé 3 mesures (la valeur finale est la moyenne des trois), afin de réduire les incertitudes de calcul de temps (l'ordinateur est des fois plus rapide et des fois moins).

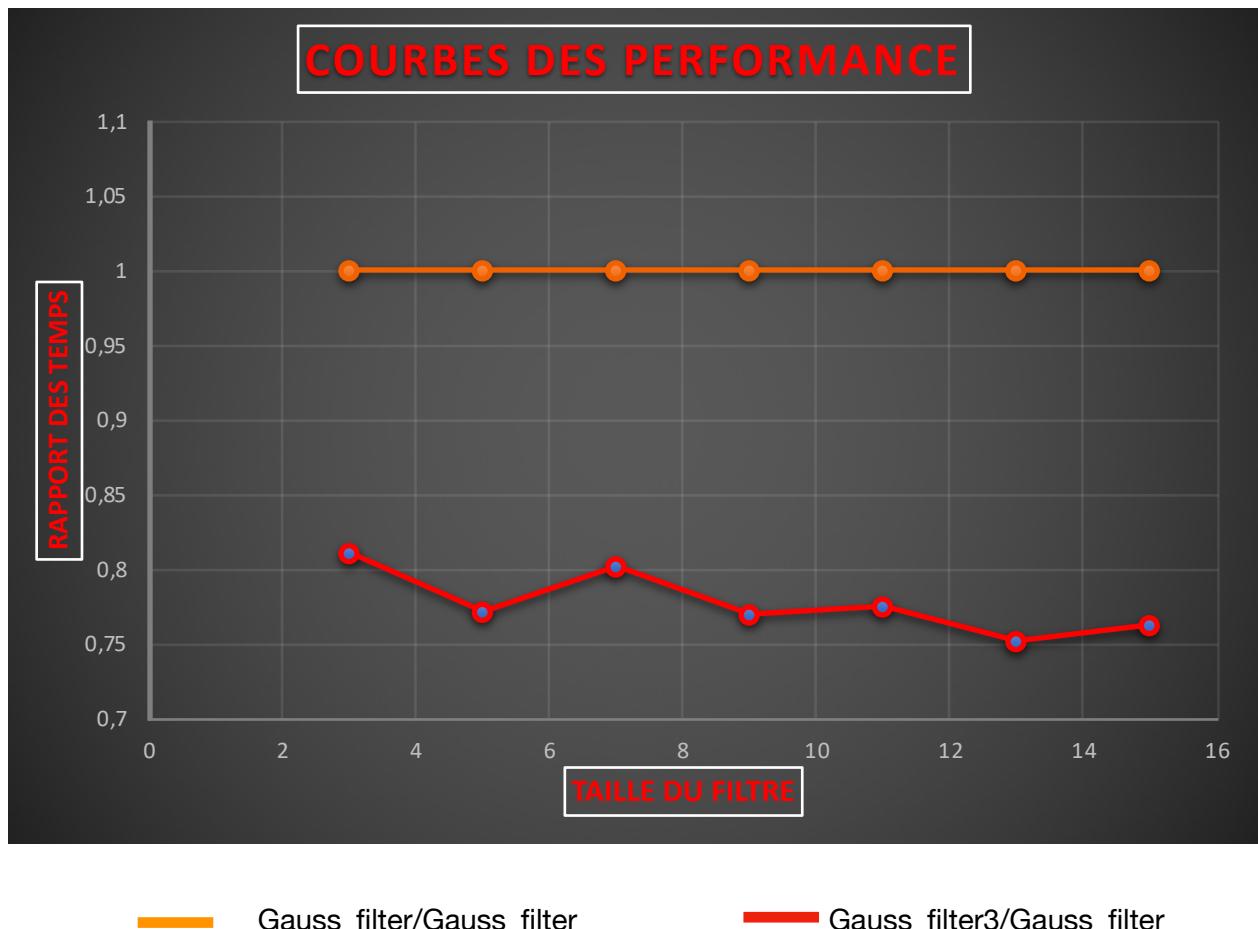


- Gauss\_filter/Gauss\_filter
- Gauss\_filter1/Gauss\_filter
- Gauss\_filter2/Gauss\_filter
- Gauss\_filter3/Gauss\_filter

Observation : On remarque une amélioration très faible, cela est cohérent, sur de gros calcul cela a tout de même un impact significatif.

→ Test avec une image de 3Mo (10 fois plus grande que l'image de Manet) pour voir si les résultats de l'optimisation est plus importante :

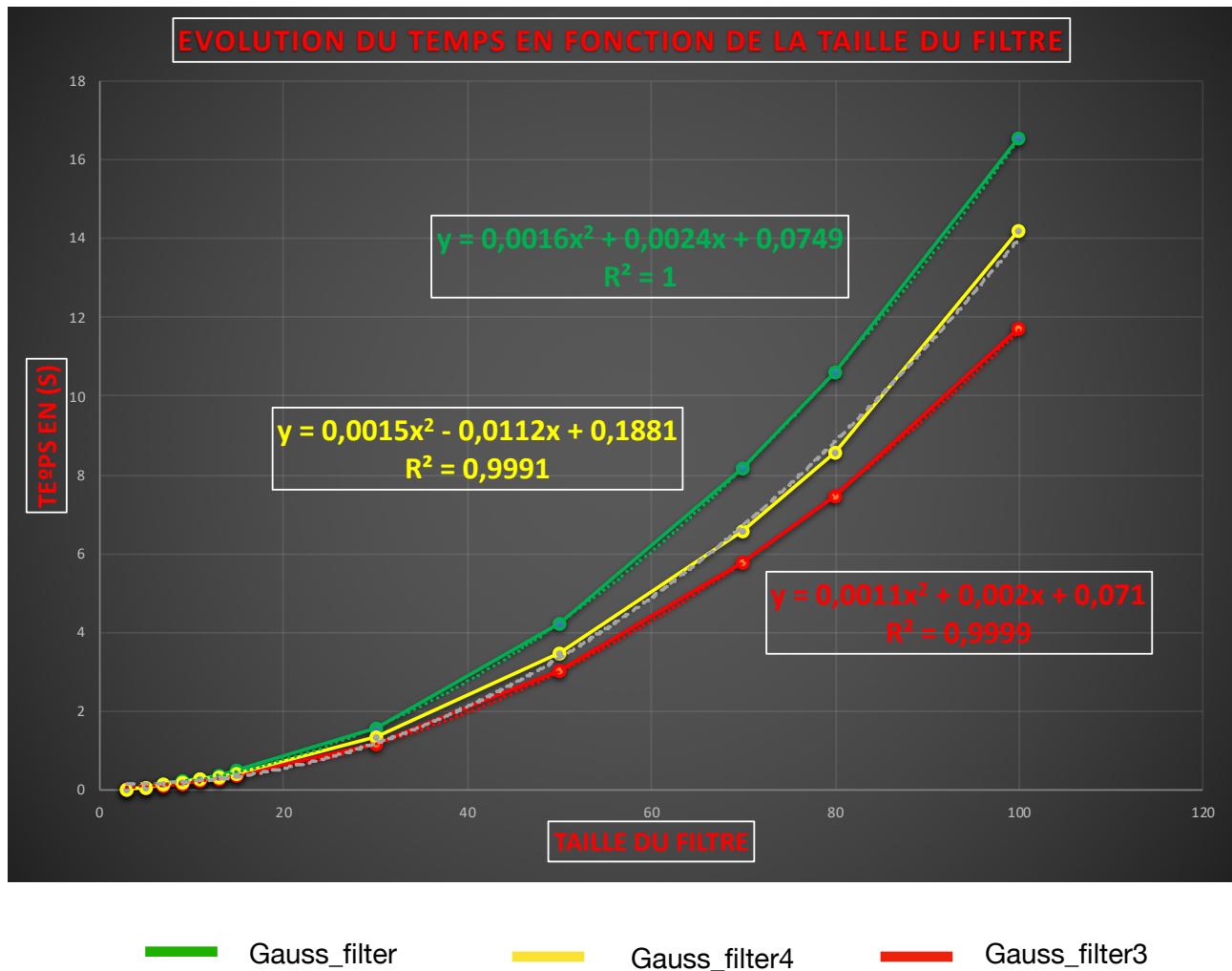




**Observation :** Les résultats sont similaires néanmoins on observe une amélioration de la performance quand on augmente la taille du filtre, ce qui est cohérent.

## 2.4/ Filtre Gaussien : deuxième optimisation : gauss\_filter4 :

**Amélioration :** Ici, on a utilisé le principe de vectorisation (en plus des améliorations précédentes). Cela permet de réaliser qu'un calcul au lieu de quatre, on travail sur un vecteur ayant R,G,B et la transparence. Cependant, le programme a été difficile à mettre en place (mais on a réussi !) Et ne permet pas une amélioration de temps, le programme met plus de temps !



Observation : Entre le programme non optimisé (gauss\_filter) et notre optimisation avant vectorisation (gauss\_filter3) on a une amélioration mais pas un changement de complexité asymptotique.

On remarque que la vectorisation (gauss\_filter4) n'apporte aucun intérêt dans ce cas, le temps a augmenté.

On remarque que le modèle polynomial d'ordre 2 est un très bon modèle.

## Conclusion :

Finalement, le fait de mettre la valeur et les calculs de l'exponentielle directement dans le programme principale a permis une grande amélioration de l'efficacité de notre programme (environ 25%), grâce à cette amélioration on pourrait traiter un plus grand nombre d'image pour le même temps.

*Programmation Open CL*

Pour conclure, OpenCL est un outil permettant de travailler sur le GPU et CPU en parallèle, il est très utile et performant pour le traitement d'images (beaucoup de « petit » calculs). Grâce à cela on peut plus finement gérer l'accès à une mémoire locale ou globale pour travailler sur le CPU ou GPU en fonction du type de calcul et ainsi optimiser le temps de calcul. Comme montré dans ce compte rendu une réflexion sur la structure de notre programme permet un travail d'optimisation pour réduire considérablement les temps de calcul (ce qui n'est pas négligeable sur un plus grand nombres de calcul à réalisé).