GAVALDA Jason SCHOENBERGER Maxime

Projet “Vision” (HPC)

0 - Présentation du projet.

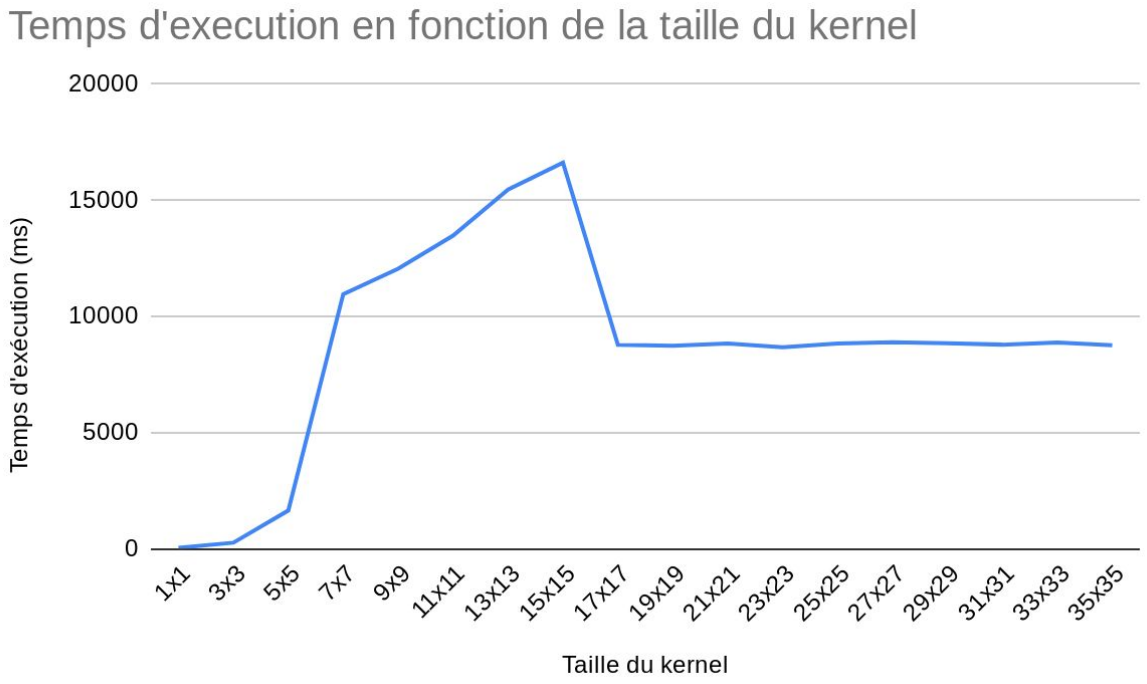
L’Objectif du projet est de développer un algorithme capable de calculer en temps réel le filtre sobel et le filtre médiane d’une image de webcam de façon efficace. Ici on partira d’une version naïve de sobel, et d’une version naïve du filtre médiane, et c’est en changeant l’algorithme, en déroulant les boucles et en réorganisant le code qu’on arrivera à un algorithme plus efficace, que nous comparerons au filtre sobel d’OpenCV.

Les tests et algorithmes sont utilisés sur un ordinateur windows 10 64 bits, avec un processeur 4 coeurs 3.60 GHz.

1 - Analyse du projet initial : mesure des performances des filtres issus d’OpenCV.

Le programme fourni pour ce TP renvoyait dans le terminal le temps d’exécution en millisecondes de l’algorithme de calcul du filtre médian, en fonction de la taille du kernel.

En faisant la moyenne d’une trentaine de mesures sur chaque plusieurs tailles de kernel, on obtient le graphique suivant :



On remarque ainsi que le temps d'exécution est assez faible pour des tailles inférieures à 5x5, mais que celui-ci bondit brutalement à partir de 7x7 pour augmenter linéairement jusqu'à la taille 15x15. Le temps d'exécution chute à nouveau brutalement à partir de la taille 17x17 pour rester constant sur les tailles de kernel suivantes que nous avons mesurées.

On en déduit que l'algorithme utilisé par OpenCV diffère selon la taille du kernel utilisé : pour des tailles de kernel inférieures à 15x15, il semble utiliser un algorithme linéaire, tandis que pour des tailles supérieures, il semble utiliser un algorithme en temps constant.

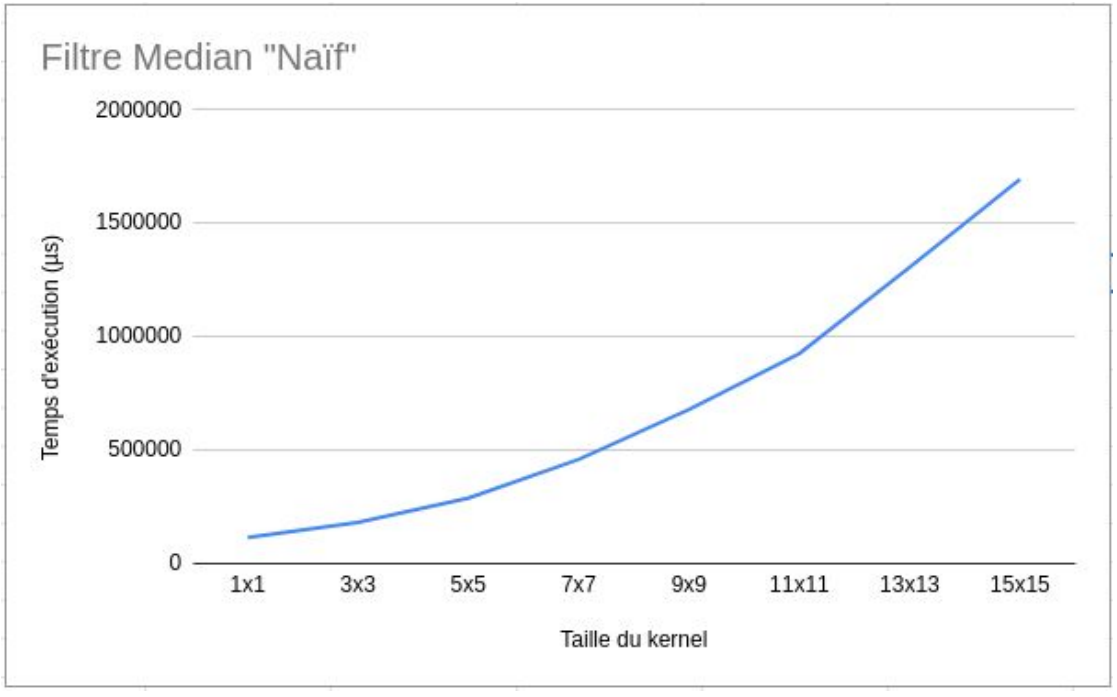
Ce genre d'utilisation de ces deux algorithmes concorde avec les travaux réalisés par Perrault et Hébert dans leur article *Median Filter in Constant Time* [1], dans lequel ils stipulent que leur algorithme en temps constant était plus performant à partir d'une certaine taille de kernel par rapport aux algorithmes linéaires et/ou logarithmiques ; cette taille variant selon les optimisations utilisées.

2 - Tests sur nos propres filtres.

2.1 - Filtre Médian

Nous avons testé les temps d'exécutions de notre propre algorithme "naïf" du filtre médian sur des kernels allant de 1x1 à 15x15 pixels.

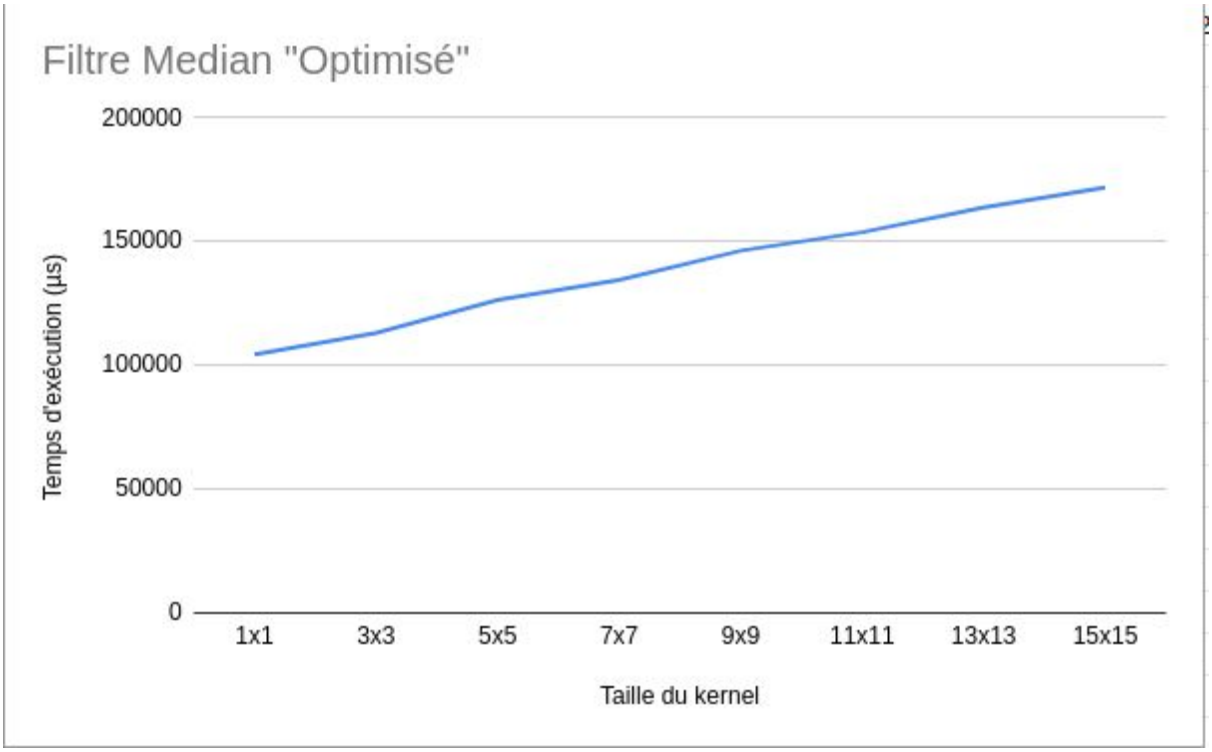
Le temps d'exécution est bien supérieur à celui utilisé par OpenCV, et la courbe indique clairement que l'algorithme n'a pas un temps d'exécution linéaire, l'augmentation du temps d'exécution elle-même tendant à devenir plus importante (déjà presque 2 secondes pour un kernel 15x15 !).



Pour changer le résultat, on reprend l'idée de l'algorithme en(k), c'est à dire plutôt que de construire un histogramme à chaque pixel, on reprend le même histogramme que l'on reconstruit en ajoutant et en enlevant une colonne.

Nous avons également essayé des déroulages de boucles sur les algorithmes auxiliaires utilisé par notre algorithme optimisé, mais cela n'a fait que rallonger de près de 10 millisecondes le temps d'exécution sur un kernel de taille 3, soit un temps d'exécution supérieur de 10 %.

Nous avons ensuite à nouveau évalué les temps d'exécution pour les mêmes tailles de kernel que précédemment, pour l'algorithme optimisé.



On remarque que cette fois que le temps d'exécution est plutôt linéaire selon la taille du kernel, ce qui est clairement une amélioration par rapport à l'algorithme précédant. Il reste néanmoins bien supérieur à l'algorithme d'opencv (de l'ordre de 10 fois supérieur), surtout en ce qui concerne les grands kernels, puisque nous avons établi que opencv utilisait un algorithme à temps constant pour ceux-ci.

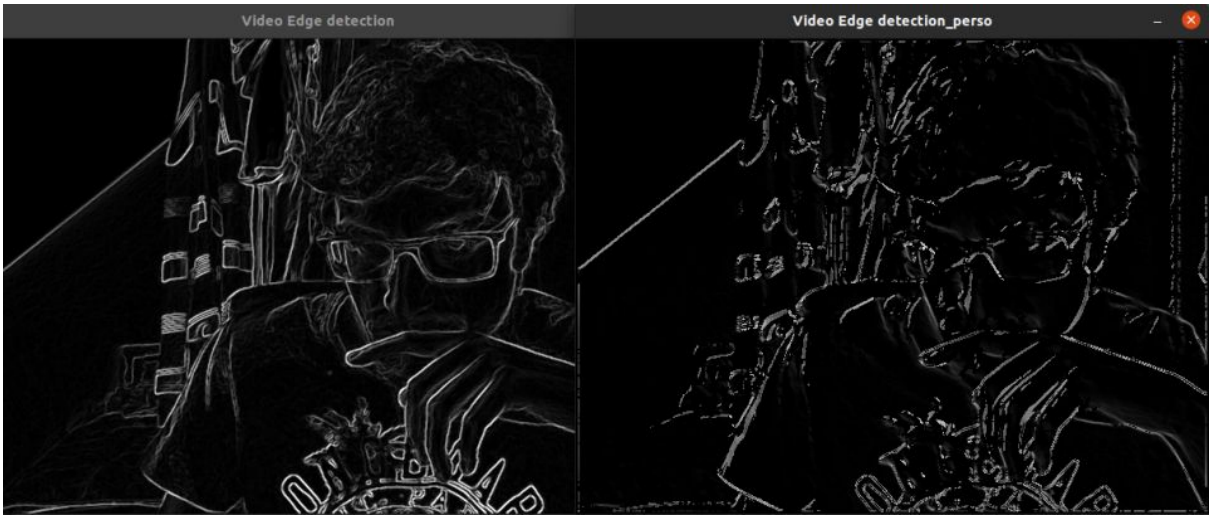
Si l'on choisit, par exemple, de prendre les temps d'exécution des deux algorithmes pour des kernels de tailles 3x3, on obtient un speedup de 175%, ce qui est clairement non-négligeable, mais pour un kernel de taille 15x15, celui-ci passe à 953% ! Ceci illustre donc bien clairement le caractère non-linéaire du 1er algorithme non optimisé.

Ci dessous, la comparaison entre le filtre médian d'opencv (à gauche) et notre filtre médian optimisé, avec une taille de kernel à 3x3 :

2.2 - Filtre de Sobel.

Notre algorithme de Sobel est décomposé en 2 parties :
Une première partie concerne le traitement de l'image sans les bords, en appliquant la convolution sur chaque pixel de l'image. Nous avons créé les deux algorithmes de convolution horizontale et verticale en dehors de l'algorithme principal pour plus de visibilité, mais le fonctionnement de l'algorithme de Sobel demeure identique. Ainsi, nous utilisons pour chaque pixel les deux algorithmes cités précédemment, et nous additionnons les valeurs absolues des deux résultats, (seulement si ces deux valeurs sont inférieures à une certaine intensité afin d'éviter des parasites sur l'image de sortie) conformément à l'approximation qui peut être faite du gradient.

Ci dessous, la comparaison entre l'algorithme d'opencv (à gauche) et le nôtre (à droite :



Nous avons ensuite testé le temps d'exécution de cet algorithme sur une moyenne de 6 mesures : 38520 µs d'exécution... contre 1373 µs pour l'algorithme d'opencv...

Nous avons amélioré l'algorithme de Sobel en reprenant le travail effectué lors d'un de nos TP précédents, qu'il a fallu adapter en C++, celui-ci ayant été effectué en assembleur. L'optimisation consistait d'une part à mêler les calculs du gradient horizontal et vertical de chaque pixel dans le but de limiter l'impact des dépendances en RAW : plutôt que d'effectuer le calcul du gradient horizontal, puis celui du gradient vertical, on effectue les deux de façon plus ou moins simultanée, c'est-à-dire en emboîtant les calculs intermédiaires indépendants à la suite pour limiter l'impact des dépendances sur le temps de calcul. D'autre part, elle consistait en un déroulage de boucle : plutôt que de traiter un pixel à la fois, nous en traitions 2, ce qui réduisait encore l'impact des dépendances dans nos calculs, puisque nous pouvions encore imbriquer d'autres calculs intermédiaires indépendants des calculs concernant le premier pixel.

En assembleur et en utilisant EduMips64, nous avions obtenu un speedup de 67% en ajoutant le forwarding.

En testant la vitesse d'exécution de cet algorithme en C++ sur 6 mesures, on obtient en moyenne 36730 µs d'exécution, soit un speedup de 5%.

A noter que nous ne calculons pas le speedup de la même manière en C++ qu'en assembleur : là où nous utilisions le rapport entre le nombre de cycles nécessaires sans optimisation et le nombres de cycles nécessaires après optimisation en assembleur, en C++ nous utilisons le rapport entre les temps d'exécution, avant et après optimisation, ce qui pourrait en partie justifier cet écart.

3 : Conclusion

Nous avons donc réalisé un filtre de Sobel fonctionnant en temps réel utilisant un filtre médian de notre conception, dont nous avons optimisé les codes sources à l'aide de réorganisation de code et de déroulage de boucles. Il a cependant été montré que ceux-ci étaient clairement moins performants que les algorithmes utilisés par OpenCV : notre filtre médian ayant un temps d'exécution plus de 10 fois supérieur et ayant une complexité linéaire, là où OpenCV est capable d'utiliser une complexité constante pour les plus grands kernels; et notre filtre de Sobel un temps d'exécution 20 fois supérieur à celui d'OpenCV, avec un kernel de 3.

Ressources :

[1] : *Median Filter in Constant Time*, Simon Perreault and Patrick Hébert