



**ENSICAEN**

ÉCOLE NATIONALE SUPÉRIEURE D'INGÉNIEURS DE CAEN  
& CENTRE DE RECHERCHE

# Rapport de projet 2A Informatique

Implémentation de 5 fonctions de traitement d'images  
sur processeurs graphiques

Gautier Boëda, Steven Le Rouzic

Mars 2015

Une grande école pour réussir

ENSICAEN  
6, boulevard Maréchal Juin – CS 45 053 – F- 14050 Caen Cedex 4  
Tél. +33 (0)2 31 45 27 50  
Fax +33 (0)2 31 45 27 60

## Sommaire

1. Présentation du projet	2
1.1. Présentation de l'entreprise	2
1.2. Présentation des filtre à implémenter	2
1.3. Problématiques d'optimisation	5
1.4. Technologies GPU	5
2. Solutions proposées	6
2.1. Étude de marché, des architectures et des API	6
2.2. Prototypes Pandore	6
2.3. Librairie écriture/ouverture d'images	6
2.4. Programme de test	6
2.5. Les implémentations	7
3. Réalisation	8
3.1. Librairie d'ouverture et d'écriture d'images	8
3.2. Programme de tests	8
3.3. Implémentation canonique sur GPP	8
3.4. Vectorisation via les extensions SSE	13
3.5. Parallélisation via Intel TBB	20
3.6. Conclusion sur les implémentations sur GPP	25
3.7. Implémentation sur GPU avec CUDA	26
3.8. Implémentation sur GPU avec OpenCL	33
3.9. Conclusion sur les implémentations sur GPU	37
4. Conclusion	42
5. Références	44
6. Lexique	45

## 1. Présentation du projet

### 1.1. Présentation de l'entreprise

Ce projet a été proposé par ADCIS (Advanced Concepts in Imaging Software), une entreprise de la région Caennaise. Cette entreprise développe des logiciels de vision par ordinateur qui se veulent être extrêmement novateurs et performants dont notamment la suite logicielle Aphelion™. En plus de cela, elle met ses compétences à la disposition de ses clients pour le développement d'applications "clés en main" dans différents domaines tels que la sécurité, le cosmétique, la science de la terre, etc. Plus d'informations peuvent être trouvées sur [www.adcis.com](http://www.adcis.com).

### 1.2. Présentation des filtres à implémenter

La première partie de la problématique du sujet était l'implémentation de 5 filtres graphiques. Les trois premiers d'entre eux sont des filtres plutôt simples : addition, inversion et seuillage. Voici des exemples de ces filtres :



*Application d'un filtre d'inversion*



*Addition de deux images*

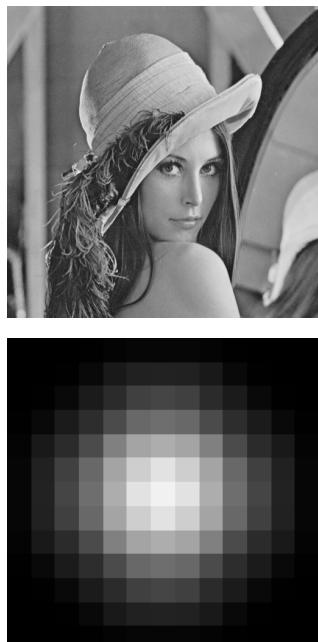


*Seuillage d'une image*

Les deux derniers sont des filtres plus compliqués qui consistent en l'utilisation de pixels adjacents pour calculer la valeur d'un pixel : la convolution et l'érosion.



*Application d'une érosion*



*Application d'une convolution*

### 1.3. Problématiques d'optimisation

La problématique majeure du projet reposait sur l'optimisation de l'exécution de ces filtres. Cela ne consistait pas en une optimisation mathématique mais en une optimisation logicielle. L'intérêt était de savoir si une implémentation de ces filtres sur GPU (Graphics Processing Unit) permettait une exécution plus rapide comparée à une implémentation sur GPP (General Purpose Processor). Les principales différences peuvent être résumées comme tels :

- GPU : Ils possèdent de grandes capacités de parallélisation grâce à la présence d'un très grand nombre de coeurs de calcul. Cependant ces coeurs sont très simples. Ils sont dit *in order*.
- GPP : Ils possèdent bien moins de coeurs mais ces coeurs sont plus complexes et plus intelligents pour du code comportant beaucoup de branchements. Ces coeurs sont dits superscalaires.

Pour cela, la parallélisation de ces filtres permettrait en théorie une exécution plus rapide. Cela consiste en l'exécution de l'algorithme du filtre sur plusieurs pixels en même temps. Les GPU sont des périphériques ayant une très haute parallélisation. L'objectif est ainsi de tirer partie de cette faculté pour observer le temps d'exécution de nos filtres.

### 1.4. Technologies GPU

Les GPU sont des périphériques dont l'architecture est hautement parallèle contrairement aux architectures à GPP ayant une très faible parallélisation (jusqu'à 8 coeurs à l'heure actuelle). Ils sont généralement utilisés pour le traitement graphiques comme leur nom l'indique mais peuvent aussi bien servir pour du calcul générique. Les GPU ont des limitations mémoires ainsi que des limitations en terme de parallélisation. Tout ceci est détaillé dans le rapport annexe "Marchés et architectures à GPU Intel et NVidia".

## 2. Solutions proposées

### 2.1. Étude de marché, des architectures et des API

Afin de cadrer le sujet du projet qui porte en particulier sur les technologies GPU, il a été en premier lieu nécessaire d'effectuer des recherches concernant ces technologies. Ces informations servent à la fois de point de repère concernant le vocabulaire et les concepts associés aux GPU, mais aussi de guide au choix de technologies. Ce rapport présente donc le vocabulaire et les concepts associés aux GPU, l'historique du marché des dernières années et à venir sur les technologies NVidia et Intel et une explication sur les deux API de programmation sur GPGPU auxquelles nous nous sommes intéressés : CUDA et OpenCL. La lecture de ce rapport séparé est très fortement recommandée pour la compréhension de celui-ci.

### 2.2. Prototypes Pandore

En première approche, nous avons implémenté les cinq filtres graphiques avec la librairie Pandore, une librairie de traitement d'image développée au GREYC, groupe de recherche en informatique et en image rattaché au CNRS. Cette implémentation nous a permis de vérifier notre compréhension des algorithmes à implémenter sans se soucier de problématiques de chargement de données ou de structures de données.

### 2.3. Librairie écriture/ouverture d'images

Pour nos implémentations indépendantes de Pandore, il nous a fallu en premier lieu créer une librairie capable d'ouvrir et d'écrire des images. Nous avons choisi le format d'image Targa pour sa simplicité. Nous avons donc choisi un format d'image avec des valeurs entières codées sur huit bits, qui correspond au format le plus utilisé. Cette librairie fait l'objet de tests unitaires afin de valider le fonctionnement de ses différentes fonctionnalités.

### 2.4. Programme de test

Afin de valider chacune de nos implémentations par rapport aux images de référence générées à l'aide du prototype implémenté avec Pandore, nous avons écrit un programme de test qui compare les images de sortie des différents implémentations et donne le pourcentage de différence entre les deux images (0% étant une correspondance parfaite) et la différence moyenne entre les pixels non égaux des deux images. Ces tests ont été automatisés à l'aide de l'outil GNU Make.

## 2.5. Les implémentations

Une fois ces tâches réalisées, nous avons pu nous concentrer sur le développement des filtres. Nous avons commencé par le C canonique sans recherche d'optimisation. C'est le cas le plus basique et que nous avons utilisé comme point de repère pour les mesures de performances des autres implémentations. Nous avons continué nos implémentations sur GPP en vectorisant les algorithmes en utilisant les extensions vectorielles SSE du processeur (extension SIMD du jeu d'instruction x86/x64). Nous avons ensuite parallélisé les algorithmes précédemment vectorisés à l'aide la librairie Intel TBB. Cette version nous a alors donné une bonne idée des performances optimales que nous pouvions obtenir sur CPU. Nous nous sommes alors concentrés sur les technologies GPU : CUDA et OpenCL. L'API CUDA nous a permis de programmer sur GPU NVidia et OpenCL sur NVidia et Intel. La variété de GPU sur lesquels nous avons testé ces implémentations nous a donné une bonne idée des performances qui peuvent être obtenues avec les différentes gammes de GPU disponibles sur le marché.

## 3. Réalisation

### 3.1. Librairie d'ouverture et d'écriture d'images

La librairie de lecture et d'écriture d'images au format Truevision Targa non compressée a été implémentée en C canonique. Les données chargées sont des entiers codés sur 8 bits. Les images peuvent avoir entre un et trois canaux. Pour des raisons qui seront expliquées lors de l'implémentation avec extension SSE, les données chargées sont alignées sur 16 octets. Cette librairie fait l'objet de tests unitaires vérifiant le bon chargement et le bon enregistrement des images.

### 3.2. Programme de tests

Dans le cadre de ce projet, au vu du nombre d'implémentations à effectuer, il paraissait important de pouvoir valider rapidement les implémentations et d'automatiser ces validations. Pour cela, nous avons développé un programme qui permet de tester les implémentations en comparant les images de sortie des implémentations avec les images de références générées avec Pandore. Le programme de test renvoie la différence constatée entre les deux images, c'est à dire le nombre d'erreur et la valeur moyenne de différence pour chaque pixel erreur. Ces deux facteurs permettent de vérifier si l'erreur est due à une approximation due à l'implémentation ou une erreur dans l'implémentation. En effet, une erreur moyenne de 1 (Un pixel contient une valeur de 0 à 255) permet de considérer l'implémentation valide contrairement à une erreur moyenne très élevée. Ces tests sont exécutés automatiquement via l'outil GNU Make.

### 3.3. Implémentation canonique sur GPP

La première implémentation du GPP des algorithmes a été réalisée en C canonique, c'est-à-dire l'implémentation la plus simple, sans recherche d'optimisation. Aucune difficulté particulière n'a été rencontrée lors de cette étape. Pour la convolution, nous avons utilisé le même format de fichiers que celui utilisé par Pandore pour stocker les valeurs de la matrice de convolution. Afin de réaliser des mesures de performances en temps utilisateur (temps réel passé, contrairement au temps processeur, le nombre de cycles que le processeur a passé sur le traitement uniquement), nous avons utilisé l'instruction x86 rdtsc qui retourne le nombre de cycles d'horloge écoulés depuis le dernier démarrage de la machine. Nous avons en plus mesuré la cadence du processeur au lancement du programme afin de convertir ce nombre de cycles en temps. Le temps est alors le temps réel associé au traitement, et non pas le temps en nombre de cycles alloués à son exécution et peut donc varier selon la charge courante du CPU. Par conséquent les mesures ont été faites en conditions de moindre occupation du CPU. Sur ordinateur portable, ces mesures peuvent être très imprécises à cause de mécanismes d'économie d'énergie, il était donc nécessaire de désactiver ces derniers avant d'effectuer toute mesure.

L'addition entre deux images consiste en l'addition pixel par pixel entre les deux images. Comme nous travaillons sur des entiers, nous devons gérer le débordement. Notre approche a été de saturer le résultat. C'est-à-dire d'assigner au résultat la valeur maximale qu'il peut contenir si la somme des deux pixels la dépasse. L'inversion consiste à soustraire à la valeur maximale pouvant être contenue par chaque pixel la valeur de chaque pixel. Le seuillage consiste à assigner chaque pixel à 0 ou à la valeur maximale pouvant être contenue par chaque pixel suivant qu'il dépasse une valeur seuil fixée à l'avance. Ces trois algorithmes se font pixel par pixel et ne dépendent d'aucune autre valeur que la leur.



*Images seuillées avec des seuils à 25%, 50% et 75%*

L'érosion consiste pour une image binaire (c'est-à-dire ne contenant que des 0 et des 1) à assigner la valeur de chaque pixel à 0 ou à 1 suivant que la région de l'image d'entrée délimitée par une forme appelée élément structurant centrée sur le pixel courant contient des pixels à 0 ou uniquement des pixels à 1. Par extension, l'érosion sur une image non binaire consiste à assigner à chaque pixel le minimum des pixels contenu dans la région délimitée par l'élément structurant. Voici un exemple d'optimisation en pseudo-code.

```

Entrées : entrée : Image
            struct : Élément structurant
Sorties : sortie : Image

Pour chaque pixel p de l'image entrée faire
    valeur := 255
    région := région de entrée centrée sur p délimitée par struct
    Pour chaque pixel r de région faire
        Si entrée[r] < valeur alors
            valeur := entrée[r]
        Fin si
    Fin pour chaque
    sortie[p] := valeur
Fin pour chaque

```

Cette méthode fonctionne aussi pour les images binaires et, dans un souci d'uniformisation, est celle que nous avons implémenté. De plus, nous avons choisi de nous limiter pour des raisons de simplicité à un élément structurant de forme rectangulaire. Il est possible ensuite d'implémenter presque aussi facilement l'érosion pour n'importe quel élément structurant en vérifiant dans un masque bi-dimensionnel si le pixel courant devrait être considéré comme faisant partie de la région ou non. Les dépassemens qui peuvent être rencontrés en bordure d'image sont gérés en ne considérant pas les points de l'élément structurant hors de l'image.



*Érosion d'une image à 255 valeurs et d'une image binaire par un élément structurant carré de 5x5.*

Enfin, la convolution consiste pour chaque pixel de l'image, en ayant défini une matrice de taille impaire appelée noyau de convolution, à assigner à chaque pixel de l'image la moyenne pondérée par le noyau de convolution des valeurs des pixels entourant le pixel courant dans les limites du noyau de convolution centré sur le pixel courant (d'où la nécessité d'avoir un noyau de taille impaire).

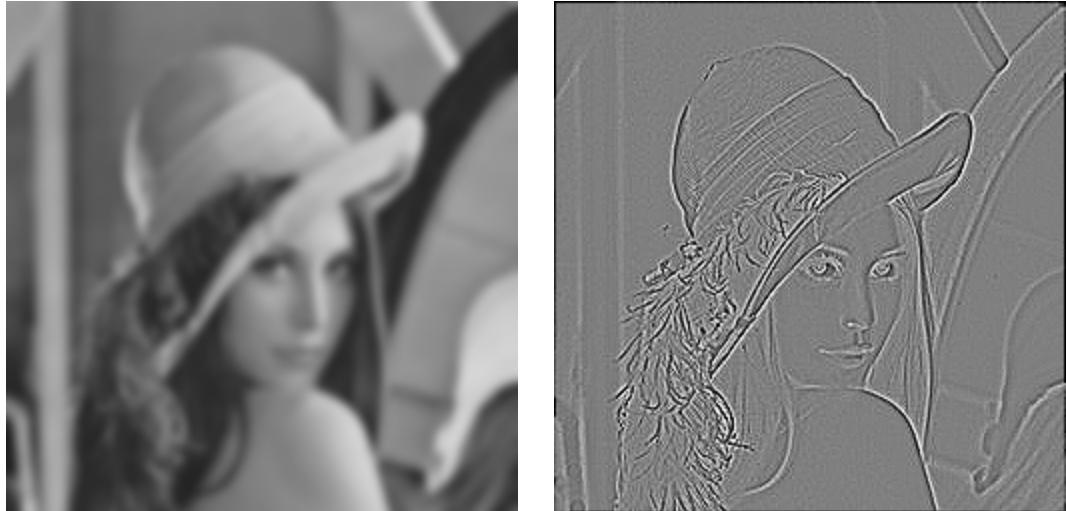
```

Entrées : entrée : Image
            kernel : Noyau de convolution
Sorties : sortie : Image

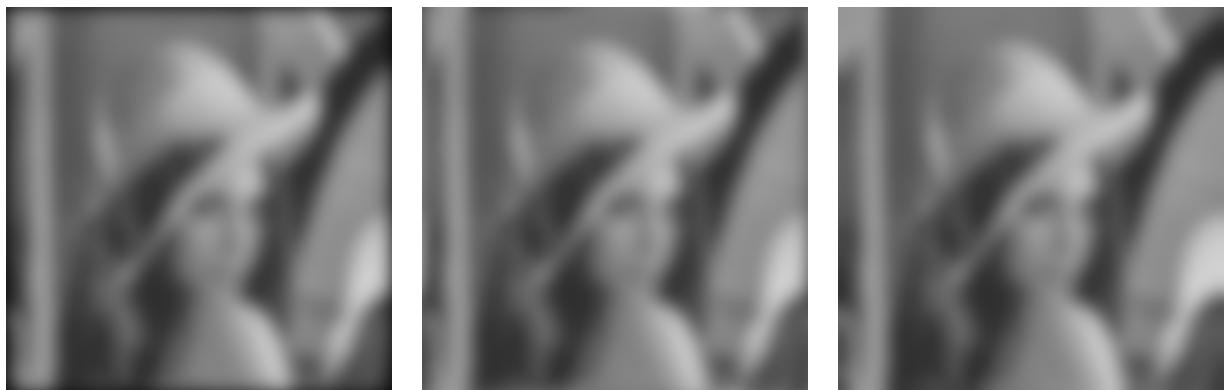
Pour chaque pixel p de l'image entrée faire
    valeur := 0
    Pour chaque valeur k de kernel faire
        valeur := valeur + kernel[k] * entrée[p + k]
    Fin pour chaque
    sortie[p] := valeur
Fin pour chaque

```

Les dépassemment rencontrés en bordure d'image peuvent être gérés de plusieurs manières : en assignant une valeur constante, par exemple 0, à ces pixels, en repartant de l'autre côté de l'image ou en prenant la valeur du pixel de l'image le plus proche. C'est cette dernière méthode qui donne les meilleurs résultats, et c'est donc celle que nous avons choisi d'implémenter.



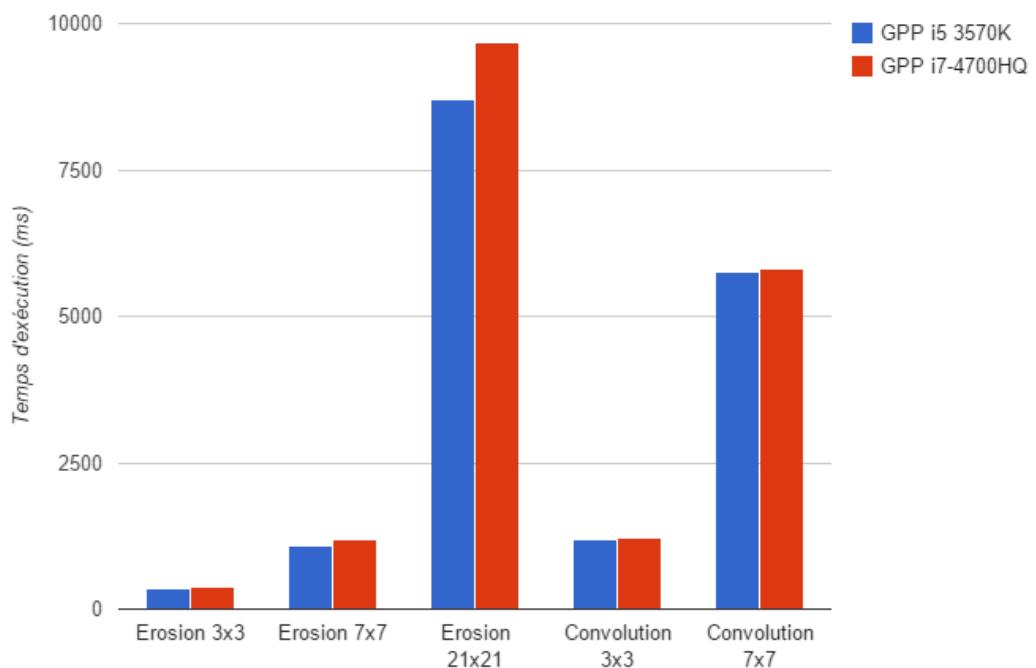
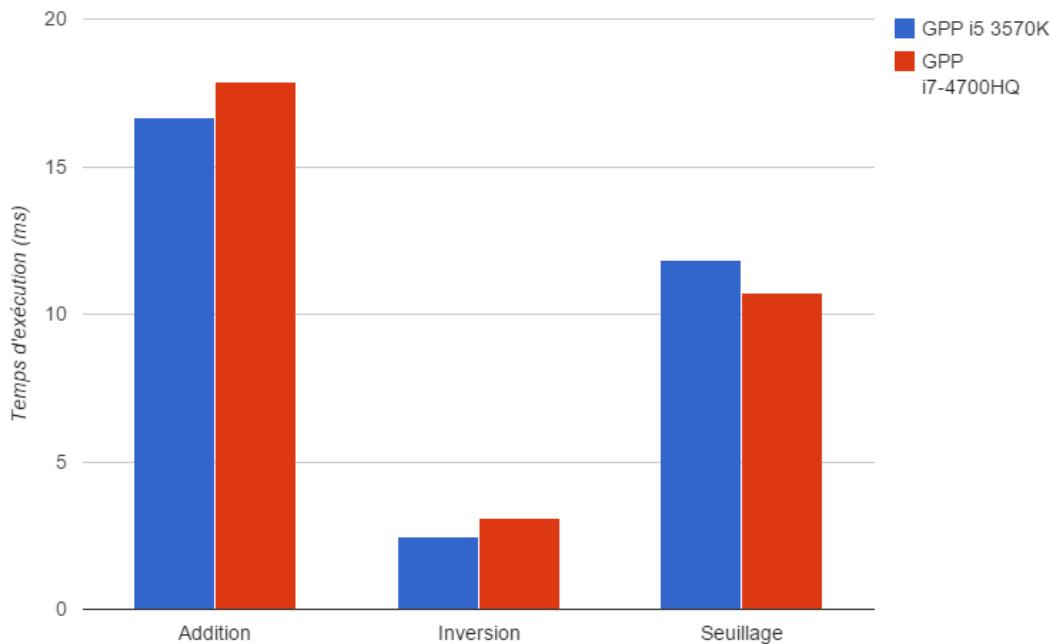
*Deux exemples de filtres par convolution : un flou gaussien et une détection de contours.*



*Illustration des trois méthodes de gestion des bords pour la convolution.*

## Résultats de l'implémentation canonique

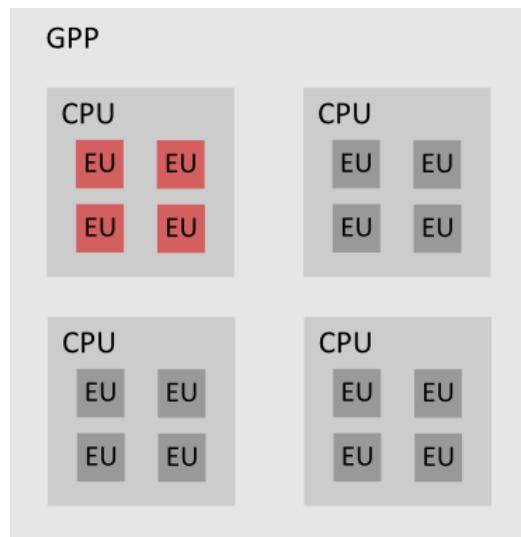
Toutes les mesures sont faites sur des images monochromes codées sur 8 bits de 4096 par 4096 pixels, soit environ 16 Mo.



On remarque principalement de ces mesures qu'il y a très peu de différences de performances entre les deux CPU utilisés. Le Core i5 est tout de même légèrement plus performante, connaissant leurs fréquences respectives. (2.4 GHz pour le Core i7, 3.5 GHz pour le Core i5).

### 3.4. Vectorisation via les extensions SSE

Afin d'améliorer au mieux les performances de cette implémentation, la première étape a été de vectoriser les algorithmes : nous avons utiliser les extensions vectorielles SSE (Streaming SIMD Extensions). La catégorie d'instructions SIMD (Single Instruction Multiple Data) établie par la taxinomie de Flynn en 1966 désigne, pour des ordinateurs dotés de capacités de parallélisme, le fait d'effectuer la même instruction sur plusieurs données à la fois et ainsi d'obtenir plusieurs résultats. Ces instruction travaillent donc sur des vecteurs de données, d'où leurs noms d'instructions vectorielles. Les deux extensions que nous avons utilisé sont les extensions SSE et SSE2 (datant respectivement de 1999 et de 2001 et ainsi considérées comme étant suffisamment répandues pour les utiliser sans trop de soucis de compatibilité. La vectorisation est une optimisation mono-coeur qui utilise les multiples unités d'exécution (EU) d'un même cœur.



Les instructions SSE (et par extension SSE2) travaillent avec des registres de 128 bits nommés `xmm0` à `xmm7`. Ainsi, ces instructions peuvent traiter par exemple 4 nombres flottants à simple précision ou encore 16 entiers à 8 bits. Dans un programme C, le type de variables contenu dans ces registres est `_m128i`. Le chargement des données vers et depuis les registres SSE se font via les instructions `_mm_load_si128` et `_mm_store_si128`.

**Code C canonique de l'addition :**

```
for (int i = 0; i < size; ++i) {
    short addition = entree1[i] + entree2[i];
    if (addition > 255)
        addition = 255;
    sortie[i] = addition;
}
```

Pour l'addition, une problématique que nous avions dans l'implémentation en C canonique était que, comme nous travaillons sur des entiers non signés de 8 bits, le résultat de l'addition pouvait dépasser la limite supérieure des valeurs pouvant être contenues (255). Ainsi nous devions effectuer l'addition dans une variable de plus de 8 bits afin d'effectuer un test de dépassement par la suite. SSE2 introduit l'instruction `_mm_adds_epu8` qui effectue l'addition de 16 entiers non signés sur 8 bits de façon "saturée", c'est-à-dire que s'il y a dépassement, la valeur est remise à 255. Une problématique qui apparaît alors avec l'utilisation de SSE est que les données qui doivent être chargées doivent être alignées sur 16 bits. C'est pour cela que l'allocation d'image dans la librairie d'ouverture et d'écriture d'image était alignée sur 16 bits. Pour gérer des images dont le nombre de pixels n'est pas multiple de 16, nous allouons également un peu plus de données afin que, si l'on doit charger des données dans un registre SSE mais qu'il n'y en a plus assez pour le remplir, nous ne dépassions pas la mémoire allouée.

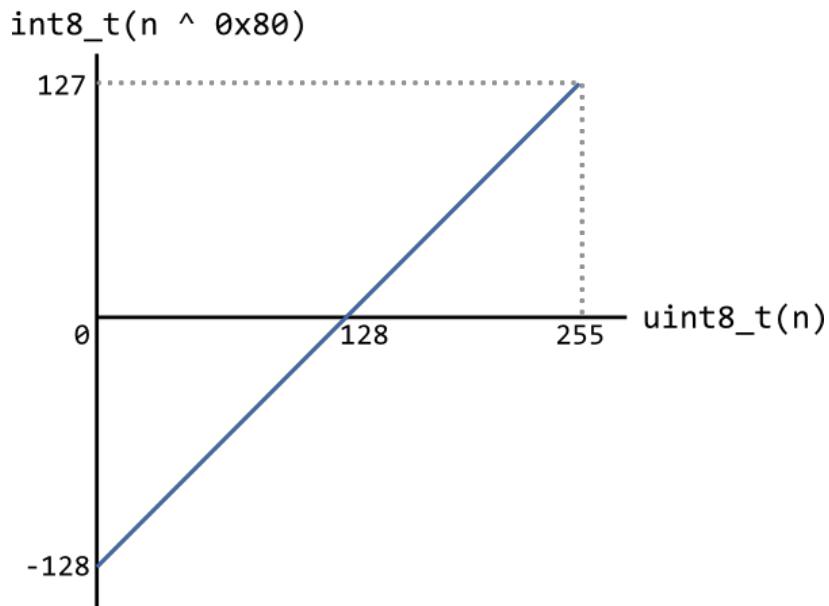
**Code C avec extensions SSE de l'addition :**

```
for (int i = 0; i < size; i += 16) {
    __m128i v1 = _mm_load_si128((__m128i *) &entree1[i]);
    __m128i v2 = _mm_load_si128((__m128i *) &entree2[i]);
    __m128i addition = _mm_adds_epu8(v1, v2);
    _mm_store_si128((__m128i *) &sortie[i], addition);
}
```

De façon classique, l'inversion de valeurs entières non signées sur 8 bits est faite par un ou exclusif avec la valeur 255. Ainsi, pour implémenter l'inversion avec l'extension SSE, nous avons commencé par placer tous les bits d'un registre à 1 grâce à l'instruction `_mm_setr_epi32`. Ensuite, il nous suffisait d'utiliser l'instruction `_mm_xor_si128` entre le registre contenant tous les bits à 1 et les données de l'image pour effectuer le ou exclusif (et donc l'inversion) de l'image.

Le seuillage a été un peu plus compliqué à implémenter à cause d'une limitation de SSE : il n'existe pas d'instruction pour effectuer des comparaisons entre entiers non signés, uniquement sur des entiers signés. Il fallait alors trouver une façon simple de transformer nos valeurs non signées pour conserver leur ordinalité lors de leur lecture en tant qu'entiers signés. Ceci a été fait en inversant le bit de poids fort des valeurs, c'est-à-dire en effectuant un ou exclusif par `0x80`, ce qui a alors le même effet que de soustraire 128 à ces valeurs. Cela transforme donc bien des valeurs entre 0 et 255 en des

valeurs entre -128 et 127 en conservant leur ordre. De plus, la comparaison est stricte uniquement, donc nous avons du soustraire 1 à la valeur de seuil car nous considérons la comparaison non stricte pour le seuillage. Nous effectuons alors la même opération à la valeur de seuil que pour les valeurs des pixels afin de la transformer en entier signé.



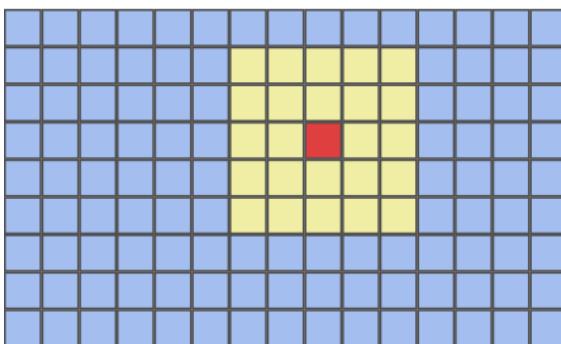
Nous remplissons alors un registre SSE de 16 fois la valeur `0x80`, et un autre de 16 fois la valeur de seuil grâce à `_mm_setr_epi8` et `_mm_setr_epi32`. Les valeurs de l'image sont alors chargées dans un registre SSE comme précédemment, on leur applique un ou exclusif par le registre rempli de `0x80` et enfin nous effectuons la comparaison des 16 valeurs par la valeur seuil grâce à `_mm_cmpgt_epi8` qui effectue 16 comparaisons de type “supérieur strictement” et qui place alors les valeurs de sortie à 0 ou à 255. Les résultats sont alors récupérés comme précédemment.

L'érosion a été l'algorithme le plus complexe à vectoriser. Cependant, nous savions que nous travaillions avec des éléments structurant carrés ce qui nous a amené à la méthode suivante : sachant que SSE nous fournit une instruction permettant de trouver les minimums terme à terme entre deux vecteurs de 16 entiers non signés sur 8 bits (`_mm_min_epu8`), nous avons commencé par mettre 16 fois la valeur 255 dans un registre. Ce registre contiendra par la suite les minimums par colonne de la région délimitée par l'élément structurant autour du pixel courant. Ensuite, pour chaque ligne de cette région, nous chargeons les valeurs de 16 pixels dans un registre SSE et effectuons un minimum avec les valeurs déjà contenues dans le registre SSE contenant les précédents minimums grâce à `_mm_min_epu8`. Une fois toutes les lignes traitées, le vecteur contient le minimum de chaque colonne pour la région donnée. Il suffit alors de trouver le minimum de ces 16 valeurs pour avoir la valeur à assigner au pixel courant pour l'érosion.

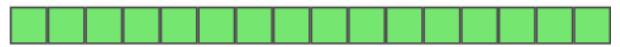
Voici sous forme de schéma l'exécution de cet algorithme :

(*Bleu = l'image, jaune = région de l'élément structurant, rouge = pixel courant, vert = registre SSE*)

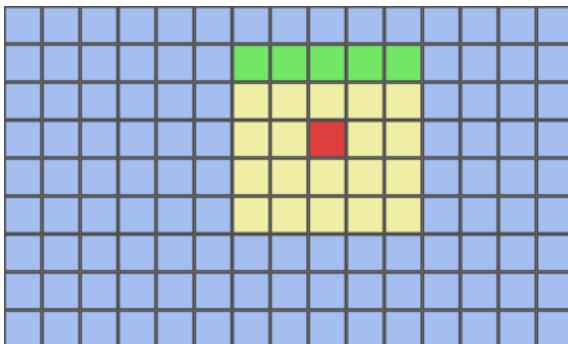
**Etape 1** (Placement du registre à 16 fois 255)



`0xffff...ff =`

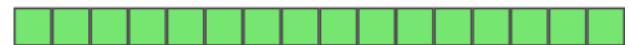
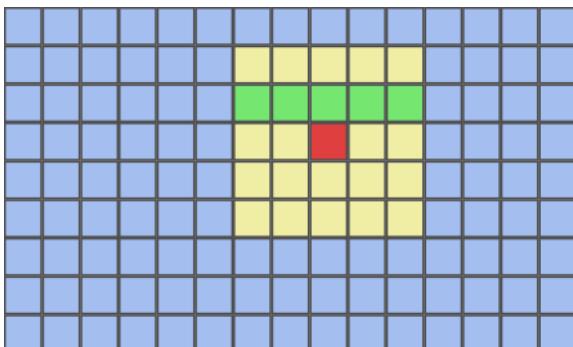


**Etape 2** (Minimum du registre avec la première ligne de la région)

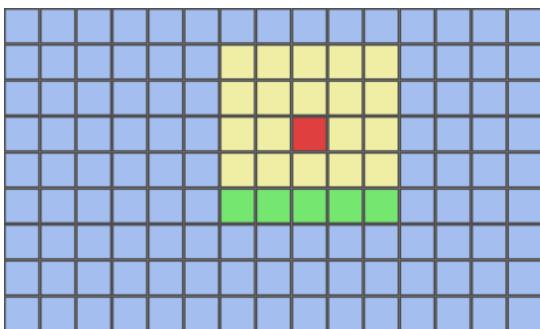


`_mm_min_epu8`

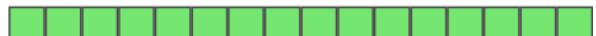
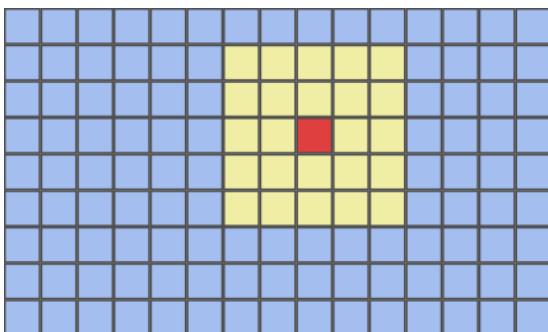
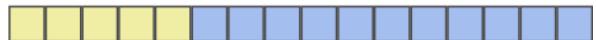
**Etape 3** (Minimum du registre avec la deuxième ligne de la région)



`_mm_min_epu8`

**Etape 4** (De même jusqu'à la dernière ligne)`_mm_min_epu8`

Le registre contient alors le minimum de chaque colonne de la région.

**Etape 5** (Calcul de la valeur minimum contenu dans le registre sur les valeurs correspondant à l'image)`_mm_store_si128`

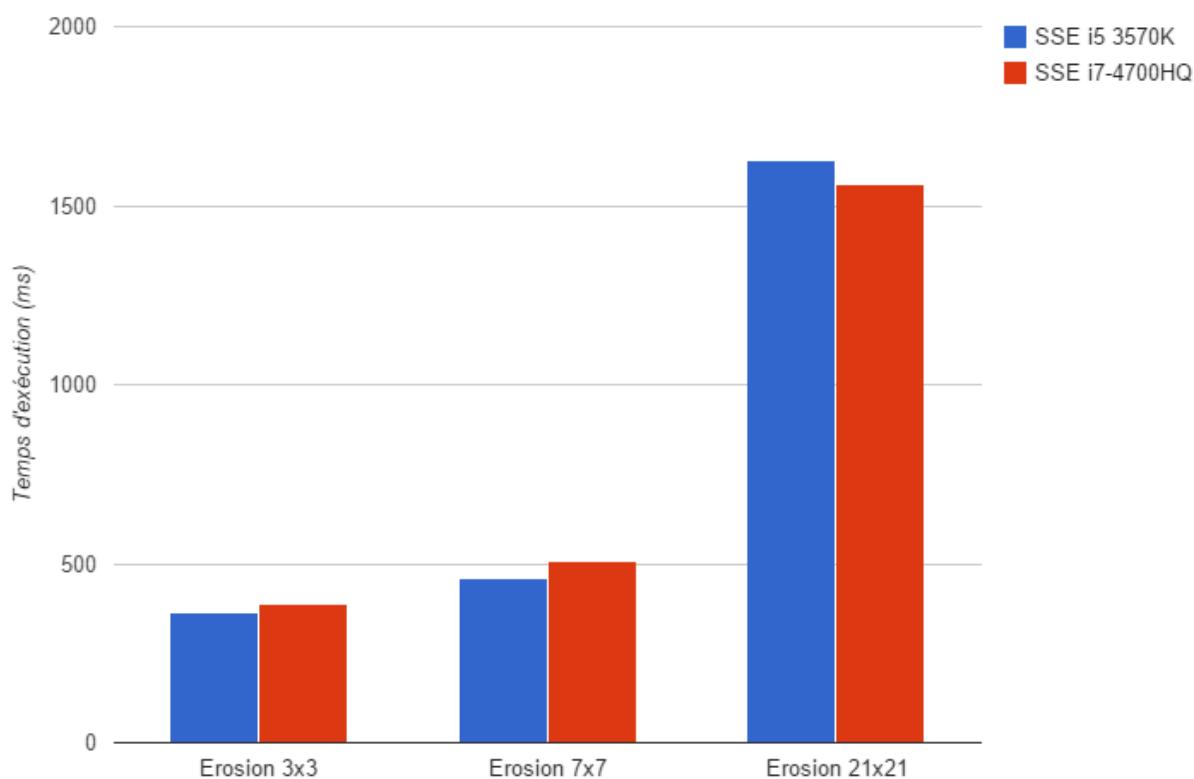
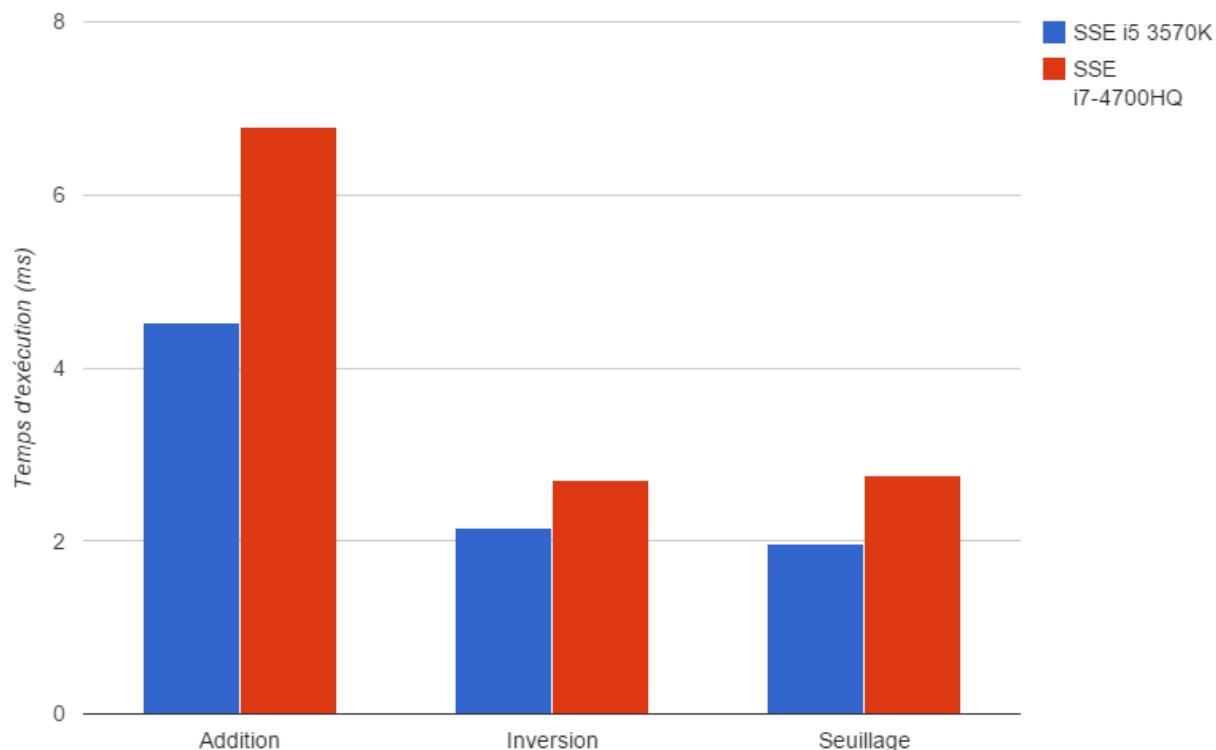
min =

On trouve alors le minimum de toute la région en cherchant la valeur minimum contenue dans le registre.

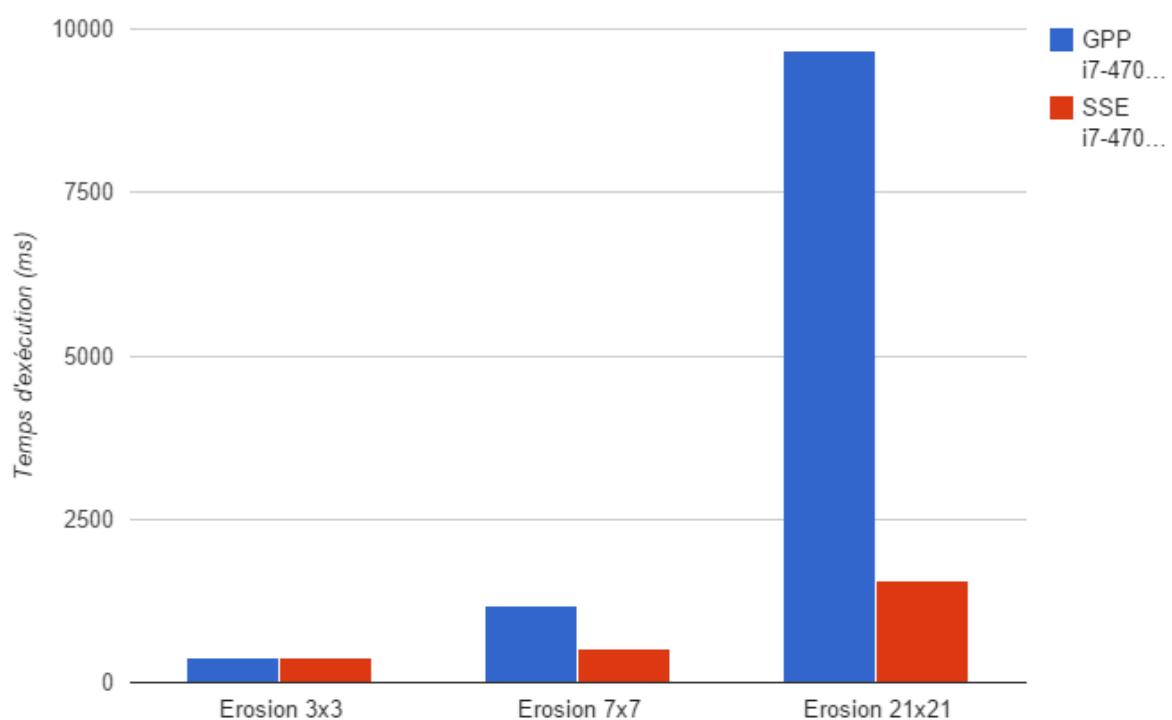
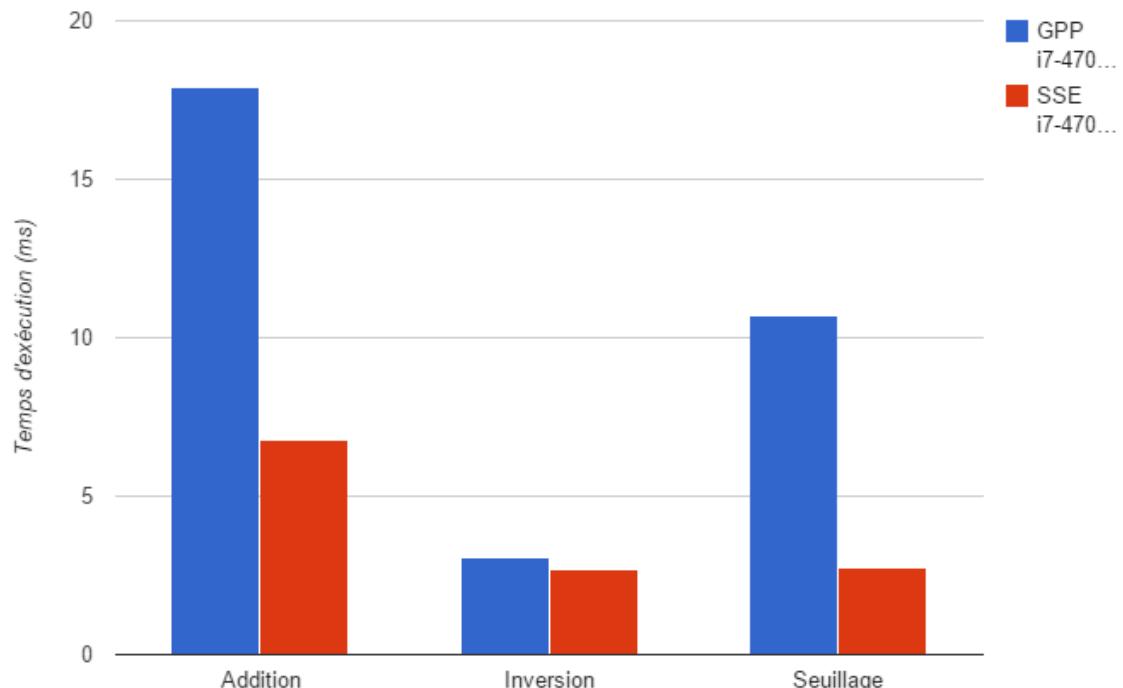
Trois problèmes se posent alors. Tout d'abord, il se peut que les données à charger ne soient pas alignées sur 16 bits, suivant la région que l'on est en train de traiter. Heureusement, SSE2 apporte l'instruction `_mm_loadu_si128` qui permet le chargement de données non alignées. Second problème, il se peut que la région à traiter ait une largeur de moins de 16. Dans ce cas, on effectue exactement les mêmes opération que dans le cas normal, mais au moment de trouver le minimum dans le vecteur final, nous ne considérons que les valeurs correspondant à la région à traiter. Enfin, une région peut avoir une largeur de plus de 16, dans ce cas nous effectuons ce processus autant de fois que nécessaire, jusqu'à ce que toute la région ait été traitée.

L'implémentation de la convolution avec les extensions SSE a été problématique dans le sens où les données étaient des entiers et les valeurs du noyau de convolution des nombres à virgule flottante. Deux approches auraient pu être prises : travailler sur des nombres à virgule flottante et ainsi avoir des résultats exacts au prix de ne pouvoir effectuer que 4 opérations à la fois avec les instructions SSE, ou travailler sur des entiers de 8 bits en normalisant les valeurs du noyau de convolution, au risque d'obtenir de l'imprécision. Pour des raisons de temps, nous n'avons pas implémenté la convolution avec les extensions SSE.

## Résultats de la vectorisation



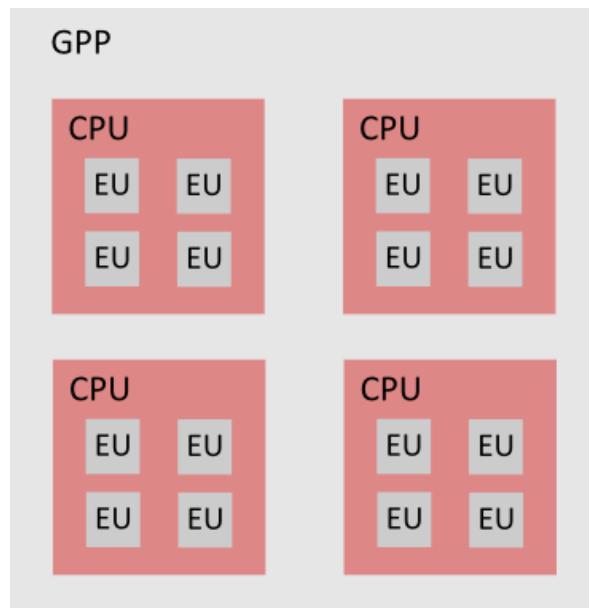
Tout comme pour les implémentations en C canonique, on ne remarque pas de réelle différence entre les performances des deux processeurs.



En comparant les performances des implémentations en C canonique et avec SSE sur un même processeur, le gain est alors évident. Le gain pour l'addition et le seuillage est un facteur de 3 à 5, en effet pour l'implémentation avec SSE, nous traitons beaucoup de données à la fois, et nous n'avons plus besoin des tests de débordement pour l'addition et du test de seuil pour le seuillage, le tout étant fait en une instruction. L'inversion montre moins de gain, mais comme elle ne consistait uniquement en un ou exclusif, ou pouvait se douter que le gain serait faible une fois pris en compte le temps de chargement vers et depuis les registres SSE. Le gain est très net pour l'érosion, et il est d'autant plus important que la taille de l'élément structurant est grande. En effet, notre implémentation fonctionne par lignes, puis par colonnes et s'approche donc d'une complexité en  $O(2n)$ , bien meilleure que la complexité en  $O(n^2)$  de l'implémentation canonique. On remarque cependant que pour une taille faible de l'élément structurant (3 par 3), le gain est plus faible voir inexistant. En effet nous effectuons alors environ 6 opération contre 9 pour l'implémentation canonique, mais il faut alors prendre en compte les temps de chargement vers et depuis des registres SSE de façon non alignée sur 16 bits. Au final, pour les algorithmes complexes, les gains peuvent aller du facteur 3 à 5. On pourrait s'attendre à des facteurs de 16 étant donné que l'on traite 16 valeurs à la fois, mais il faut également prendre en compte les opération de chargement depuis et vers les registres SSE.

### 3.5. Parallélisation via Intel TBB

Suite aux implémentations précédentes sur GPP, l'aspect parallèle n'avait pas été encore abordé. C'est ce que va nous permettre Intel TBB (Threading Building Blocks) qui permet au développeur d'utiliser plusieurs outils dont la possibilité d'utiliser les différents coeurs du CPU. Même si le nombre de coeurs est faible (entre 1 et 8 actuellement), le gain attendu est assez conséquent : un facteur 8 au mieux dans le cas d'un processeurs à 8 coeurs. Ce type d'optimisation est donc multi-coeur.



Les implémentations étaient simples à effectuer car il suffisait juste de créer l'algorithme à exécuter sur chaque segment et lui indiquer le nombre de fois qu'il devait exécuter l'algorithme. Nous n'avons pas rencontré de problèmes du côté de l'accès en lecture et en écriture à la mémoire, seul un problème en temps de création de threads a été découvert pendant la conception. Un thread est une copie de l'algorithme s'exécutant sur un segment précis. En effet, ce coût n'est pas négligeable dans le cas d'algorithme ne comportant que peu d'instructions comme les 3 premiers filtres. La création du thread est alors plus coûteuse que l'exécution de l'algorithme sur un pixel. De ce fait, il s'est avéré après recherche que l'implémentation la plus intéressante était de couper l'image en segments suffisamment grands (Nous avons ici découpé notre image en ligne, donc 4096 pixels par thread pour une image 4096 par 4096).

#### **Exemple d'addition avec IntelTBB :**

```
class AddParallel {
public:
    AddParallel(Image *entree1, Image *entree2, Image *sortie) :
        entree1(entree1), entree2(entree2), sortie(sortie) {}

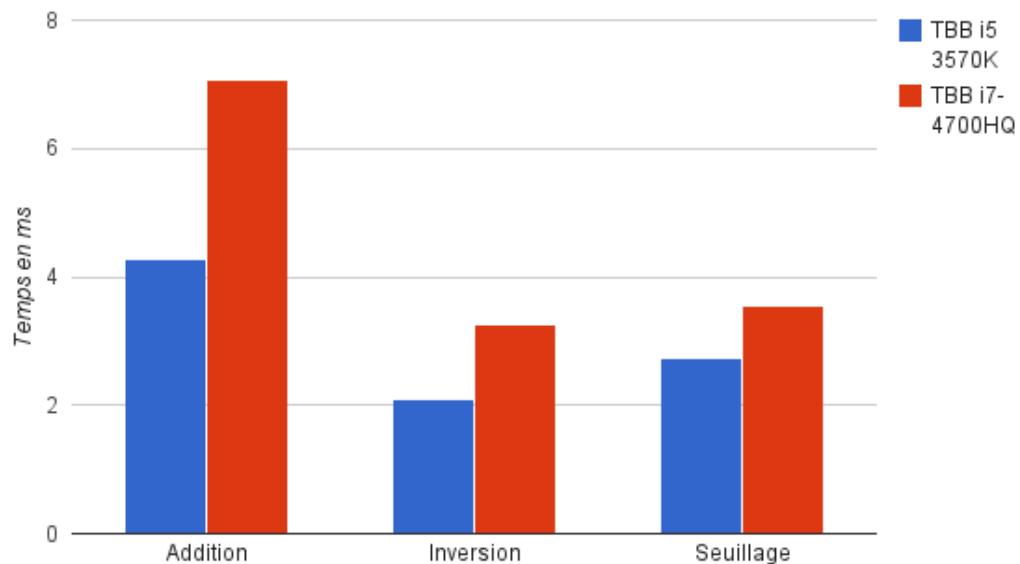
    void operator()(int i) const {
        for (int j = 0; j < entree1->largeur; ++j) {
            int pixel = j + i * entree1->largeur;
            short addition = entree1->data[pixel] + entree2->data[pixel];
            if (addition > 255)
                addition = 255;
            sortie->data[pixel] = addition;
        }
    }

private:
    Image *entree1;
    Image *entree2;
    Image *sortie;
};

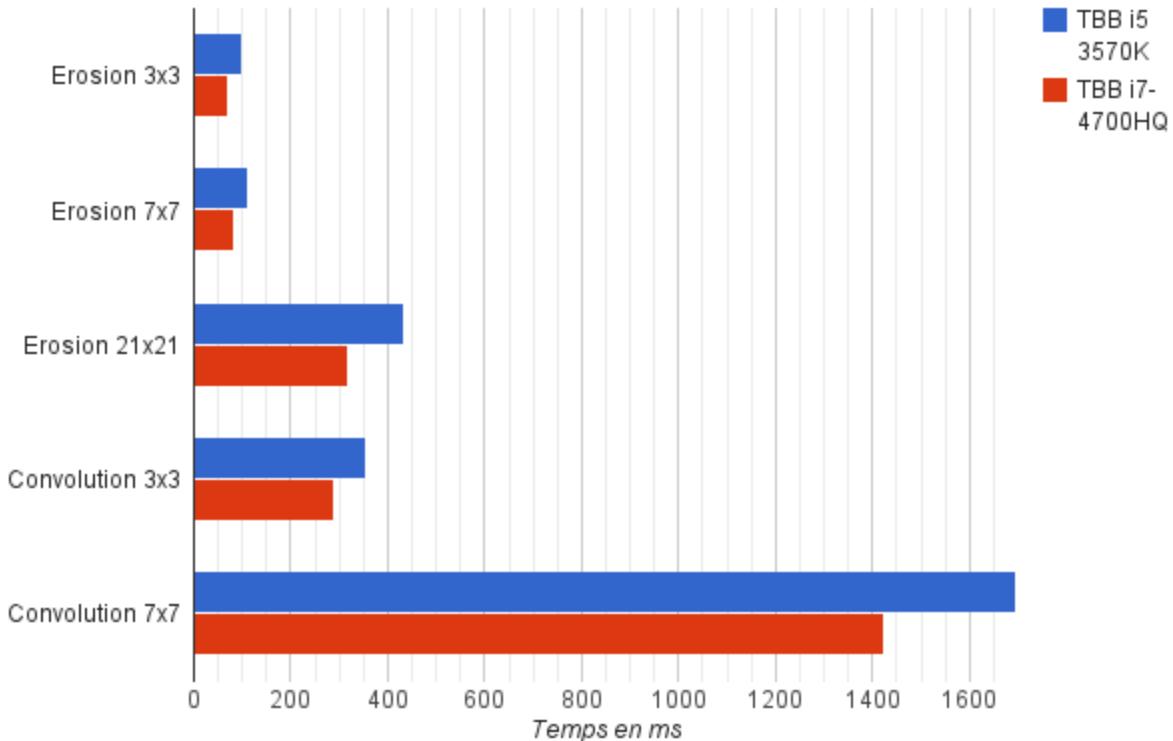
AddParallel addParallel(entree1, entree2, sortie);
tbb::parallel_for(0, entree1->hauteur, addParallel);
```

## Résultats de la parallélisation

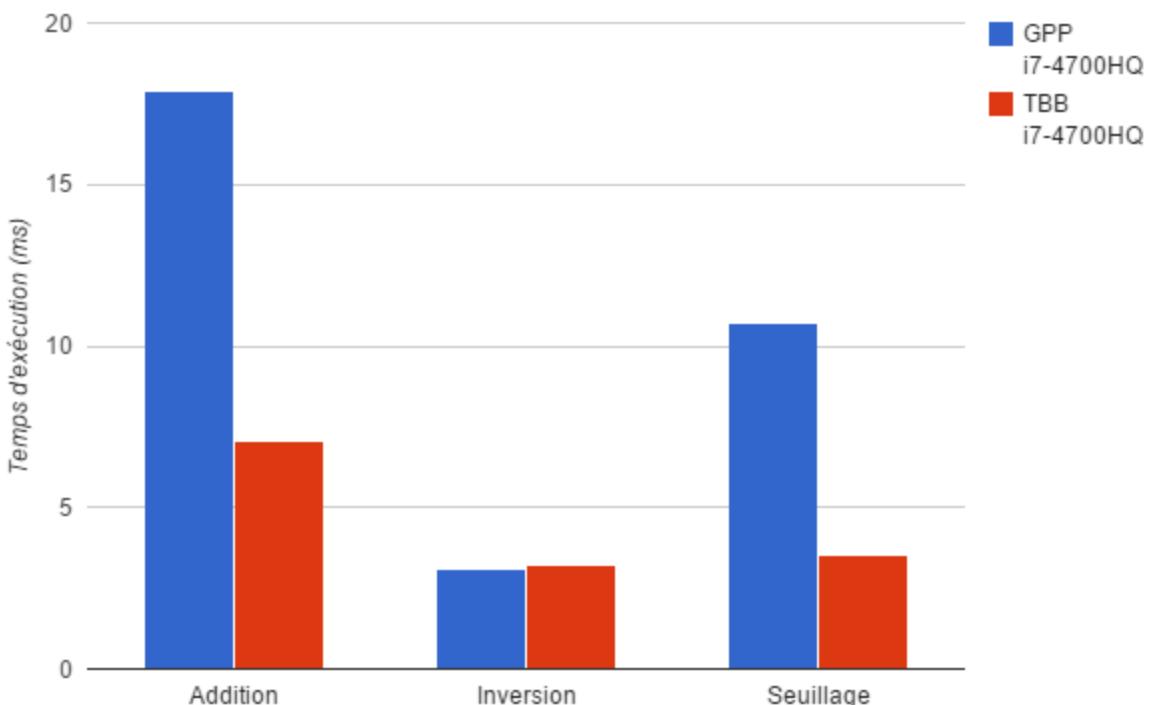
Nous obtenons ainsi les résultats suivant pour les 5 implémentations sur nos GPP des algorithmes vectorisés et parallélisés :



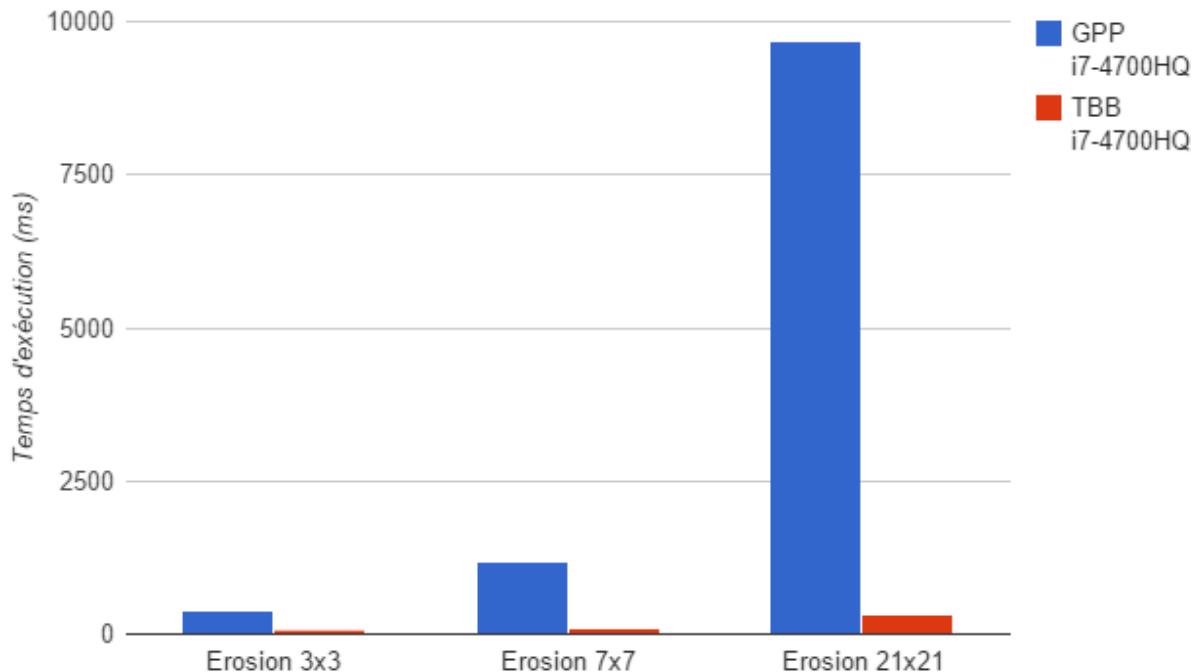
Les algorithmes étant très courts, le i5 réussit à avoir les meilleures performances dues notamment à une fréquence plus élevée malgré un nombre de cœur logiques de 4 (Le i7 a 4 coeurs physiques et 8 coeurs logiques en comparaison). Néanmoins, on remarque que la différence entre l'implémentation SSE et celle TBB avec SSE est très faible. Avec même certains filtres très simple comme l'inversion, les performances sont meilleures sans l'utilisation d'Intel TBB. On remarque donc que le temps d'exécution actuel des threads est si court que nous n'arrivons pas à tirer partie de la parallélisation de ces filtres, SSE faisant déjà l'essentiel du travail. Mais ceci ne sera pas le cas pour des filtres plus complexes.



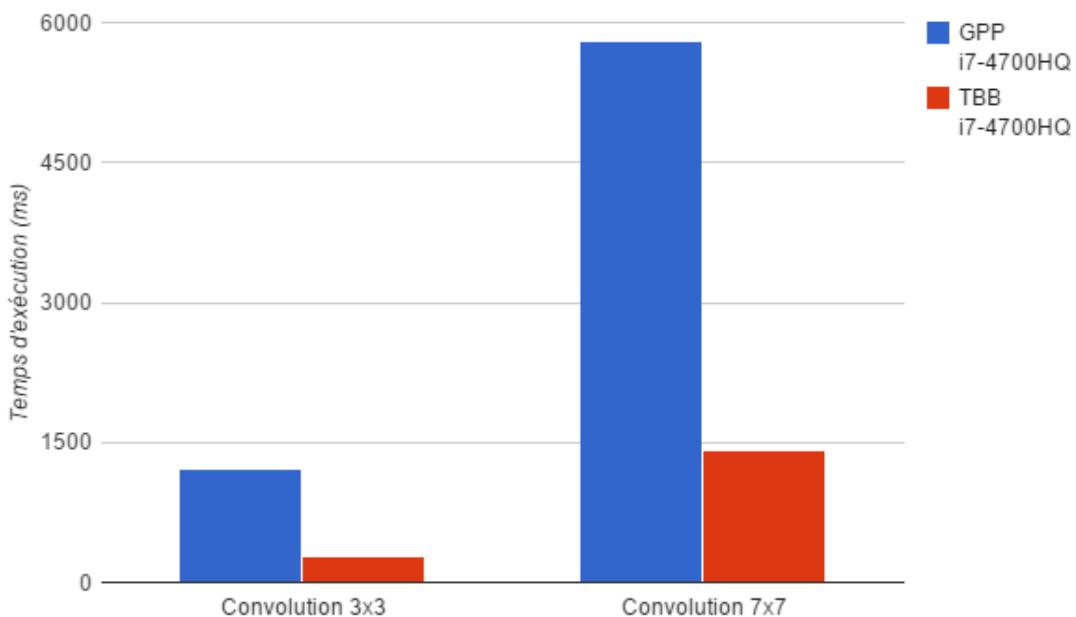
Pour les filtres plus complexe, on remarque que le Core i5 est plus lent que le i7. Ceci était prévisible à la vue de la complexité du filtre ajouté au nombre de coeurs logiques que chacun des CPU disposent. Le i7 ayant 2 fois plus de coeurs logiques est alors plus rapide que le i5 et ce, malgré une fréquence plus faible de 1.1 GHZ.



Le coût de création d'un thread est, pour les filtres simples, non négligeable par rapport au temps d'exécution du thread. C'est sur ce point que les différences de performances entre les filtres simples et les filtres plus complexe sont observables.

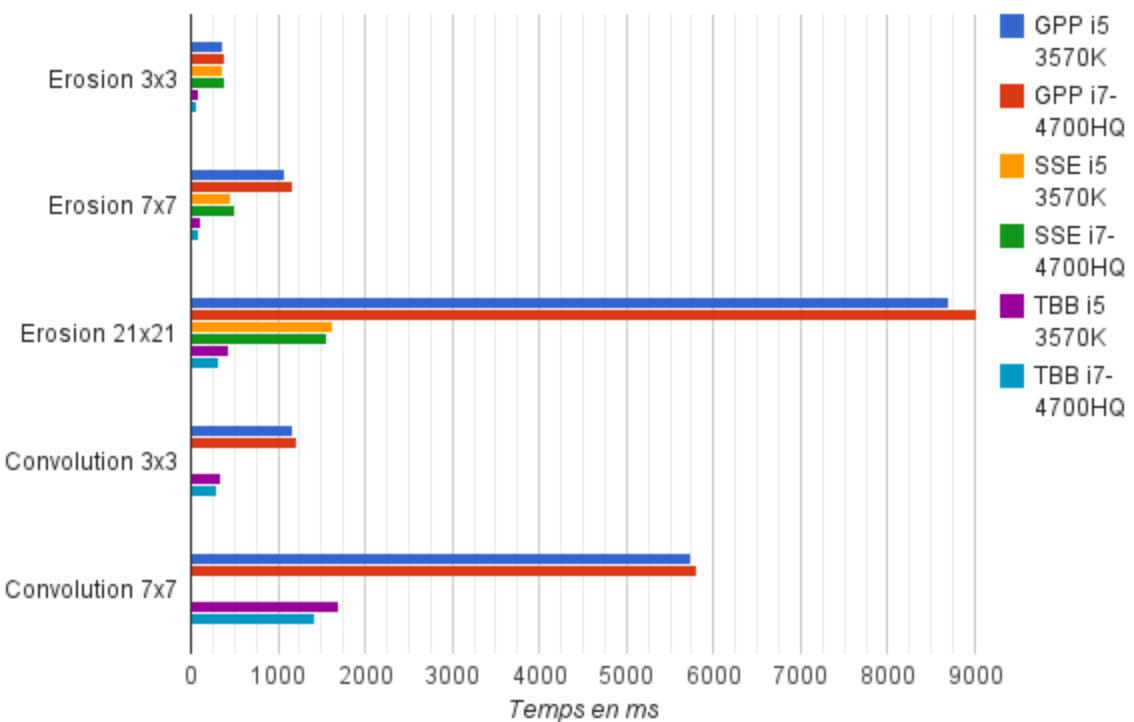
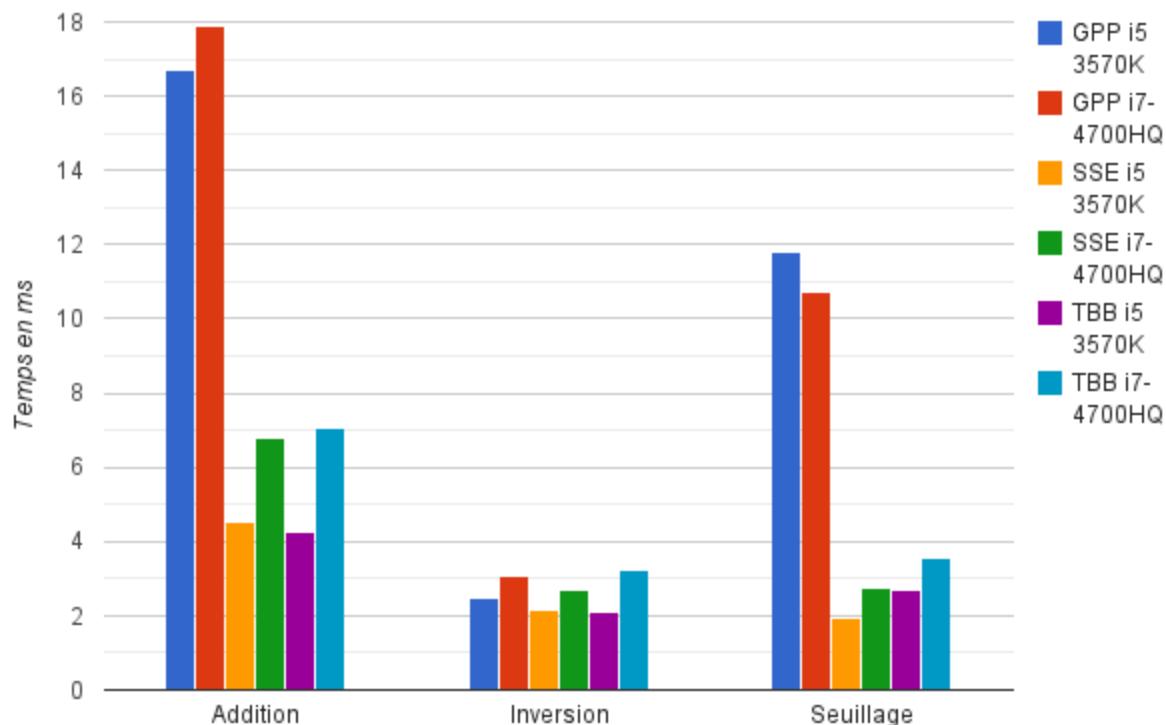


Sur l'érosion, le gain va du facteur 5 au facteur 30, y compris lorsqu'il n'y en avait pas avec l'implémentation utilisant uniquement SSE comme pour l'érosion 3 par 3. La parallélisation est alors ici un excellent complément à la vectorisation.



Des gains allant du facteur 4 au facteur 5 sont aussi à noter pour la convolution, avec uniquement de la parallélisation.

### 3.6. Conclusion sur les implémentations sur GPP



Comme nous l'avons vu précédemment, une implémentation sur GPP basique peut être optimisée de plusieurs manières. Tout d'abord par les instructions SIMD qui nous ont permis de gagner plus qu'un facteur 5 (Seuillage et érosion 21x21). Néanmoins, pour l'érosion de faible rayon (3x3 par exemple), nous obtenons tout juste les mêmes performances, tout comme pour l'inversion. Cela est dû aux temps d'accès à la mémoire et à l'implémentation (uniquement dans le cas de l'érosion) qui requiert d'avoir un rayon proche de 7 (donc 15x15) pour utiliser au mieux les instructions SIMD comme expliqué précédemment. Via la parallélisation de l'implémentation avec les extensions SSE, nous avons à nouveau réussi à gagner un facteur 5 en moyenne sur les filtres plus complexes.

Au final, avant de se poser la question d'implémenter son filtre sur GPU, il est tout à fait intéressant de se pencher sur les différentes optimisations possibles sur GPP. D'une implémentation simple à une implémentation optimisée, le facteur gagné peut être de plus de 30 (Cas de notre érosion 21x21). Ainsi, tous ces facteurs peuvent rendre une implémentation sur GPU inutile si les filtres sont simples. Néanmoins, une implémentation sur GPP sera toujours limité niveau parallélisation, alors que les filtres de traitement d'image peuvent être parallélisés. Nous allons donc maintenant étudier nos implémentations sur GPU.

### 3.7. Implémentation sur GPU avec CUDA

L'API CUDA permet la programmation générique sur processeurs graphiques NVidia. La lecture de la partie lui étant dédiée dans le rapport sur les architectures et marchés des GPU est très fortement recommandée avant d'aborder cette partie.

Deux versions de l'addition ont été réalisées. Une première version effectue les additions de façon classique en quatre étapes : récupération des valeurs des deux images, additions des valeurs, saturation du résultat et placement du résultat dans le pixel de sorti correspondant. En première approche, chaque thread effectuait ces quatre étapes pour un pixel à la fois. Nous avons ensuite fait traiter plusieurs pixels par chaque thread. Les meilleurs résultats étaient avec 4 pixels par thread. Ensuite, au lieu d'effectuer ces quatre étapes de façon intercalées, nous les avons séparées en quatre boucles séparées. Ainsi le kernel commence par récupérer toutes les données, puis additionne les quatre valeurs, puis sature les quatre, et enfin retourne les quatre résultats. C'est le principe de l'ILP (Instruction Level Parallelism) qui permet d'utiliser la pipeline liée aux instructions pour lancer des traitements similaires tous à la suite, sans attendre les résultats d'opérations précédentes. Cette modification nous a fait gagné environ 10% de performances. Enfin, nous avons utilisé la directive `#pragma unroll` afin de dérouler les boucles, et d'améliorer l'efficacité de l'ILP.

#### Exemple de boucle déroulée.

```
#pragma unroll
for (int i = 0; i < 4; ++i) {
    a[i] = b[i] + c[i];
}
```

Une deuxième approche pour l'addition a été d'utiliser l'instruction intrinsèque SIMD `__vaddus4` qui permet l'addition saturée de quatre entiers non signés sur 8 bits.

```
__global__ void addition(uint32_t *entree1, uint32_t *entree2, uint32_t *sortie) {
    int thread = (gridDim.x * blockIdx.y + blockIdx.x) * blockDim.x + threadIdx.x;
    sortie[thread] = __vaddus4(entree1[thread], entree2[thread]);
}

int threadsParBlock = 256; // 32 * 32 / 4
dim3 blocks(entree1->largeur / 32, entree1->hauteur / 32, 1);
addition<<<blocks, threadsPerBlock>>>((uint32_t *) entree1->data,
                                             (uint32_t *) entree2->data,
                                             (uint32_t *) sortie->data);
```

L'implémentation de l'inversion a été très similaire à celle de l'addition : une première version a utilisé les mêmes méthode pour prendre partie de l'ILP et une deuxième utilise l'intrinsèque `__vsubss4` pour effectuer l'inversion de quatre entiers non signés sur 8 bits.

L'implémentation du seuillage a également bénéficié de l'utilisation d'un intrinsèque : `__vcmpgeu4` pour la comparaison de quatre entiers non signés sur 8 bits. Cependant, la deuxième implémentation est bien plus simple que les précédentes. En effet, l'utilisation du principe l'ILP était bien moins efficace qu'une implémentation naïve, probablement à cause de l'utilisation de tests dans le kernel.

L'érosion a été implémentée de deux manières. Une première implémentation très naïve se contente de parcourir l'élément structurant pour en trouver le minimum. La deuxième implémentation prend en compte le fait que le calcul de pixels proches les uns des autres auront besoin des mêmes données. Ainsi, au lieu d'aller chercher en mémoire globale les données à chaque fois que l'on en a besoin, ce qui aurait pour effet d'aller chercher de nombreuses fois la même donnée pour plusieurs pixels, nous exécutons l'érosion sur une région de l'image au sein d'un même block afin que les différents threads (un par pixel) puissent se partager via la mémoire partagée les données des pixels voisins. Ainsi, en début de thread, chaque pixel du block va chercher certains pixels dont il aura besoin et les place dans un segment de mémoire partagée partagé entre tous les threads du block. À la fin de cette étape, ce segment contient toutes les données qui seront nécessaires au calcul de l'érosion sur tous les pixels calculée par le block. Cela présente trois avantages : l'accès à la mémoire partagée est plus rapide que l'accès à la mémoire globale, les threads se partagent les données à aller chercher en mémoire et chaque donnée n'est récupérée qu'une seule fois depuis la mémoire globale, et les lignes de la région stockée en mémoire partagée étant plus petites que la largeur de l'image, les données en mémoire partagée sont plus compactées et donc grâce à des mécanismes de cache, plus rapides à accéder que si elles étaient plus dispersées. Le calcul de l'érosion se fait ensuite de la même manière que pour l'implémentation naïve, mais utiliser les données en mémoire partagée qui sont donc plus rapides à

l'exécution. La seule précaution à prendre est qu'il faut attendre que tous les threads aient fini de récupérer leurs données avant de lancer le calcul de l'érosion grâce à la directive `__syncthreads`.

**Exemple d'utilisation de la synchronisation de threads et de la mémoire partagée** (ici on additionne un terme avec le terme suivant) :

```
#define N 256
__global__ void additionSuivant(float *entree, float *sortie) {
    __shared__ float[N + 1] cache;
    int thread = threadIdx.x;
    int decalage = blockDim.x * blockDim.x; // blockDim.x = N

    cache[thread] = entree[decalage + thread];

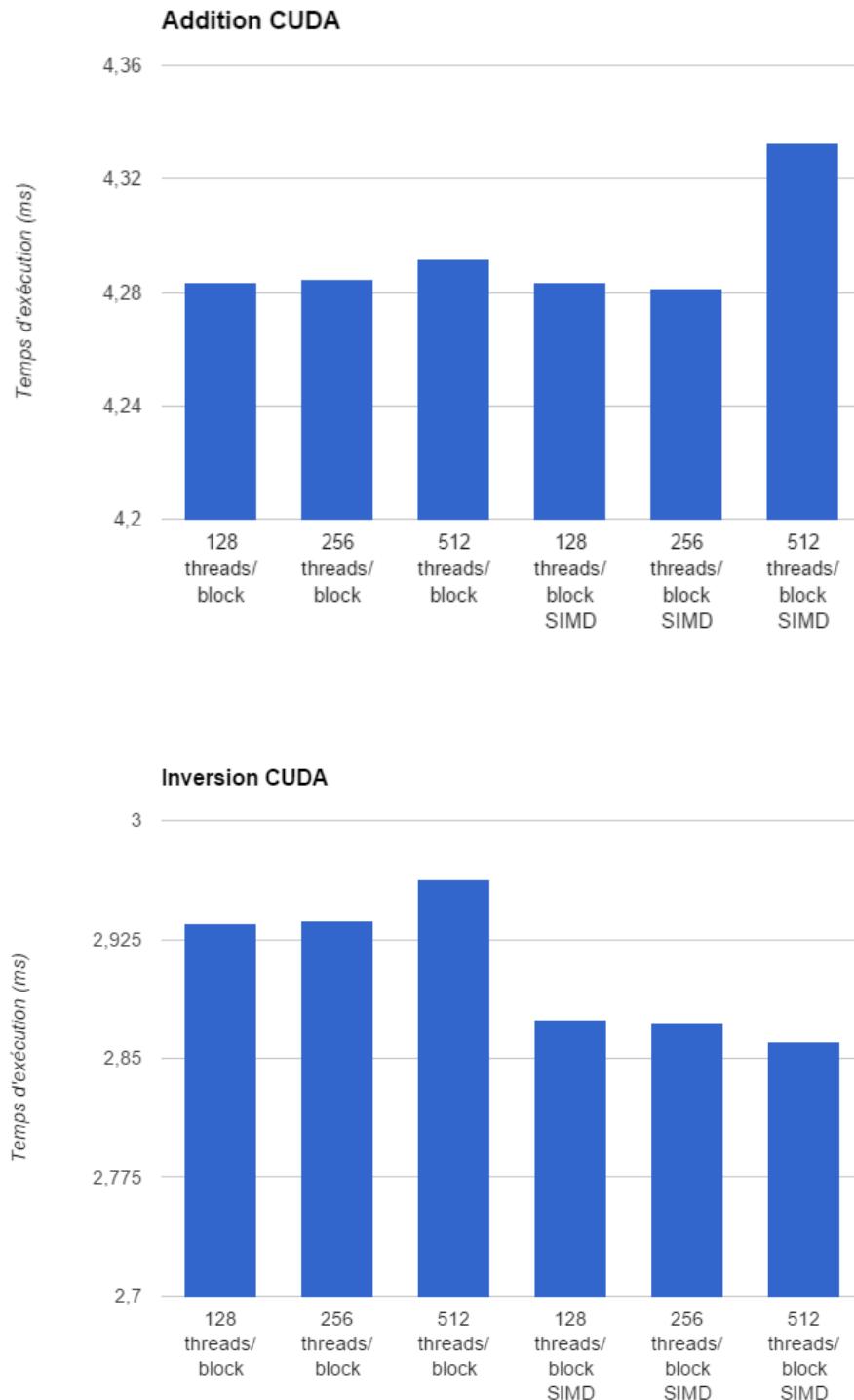
    if (thread == N - 1) // Dernier thread => on va chercher la valeur suivante
        cache[thread + 1] = entree[decalage + thread + 1];

    __syncthread();
    sortie[decalage + thread] = cache[thread] + cache[thread + 1];
}
```

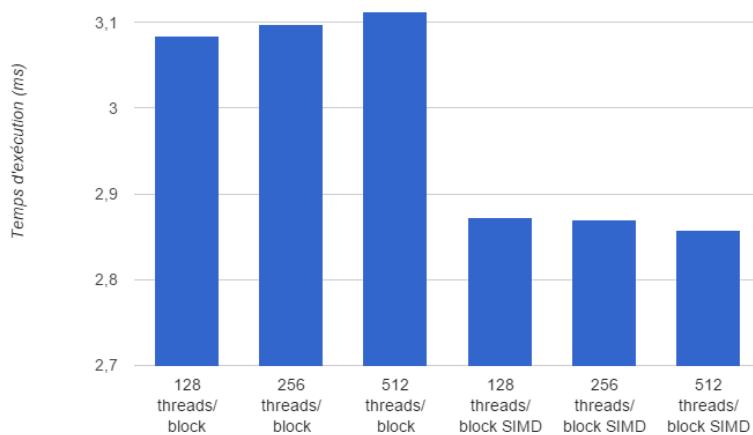
La convolution a été implémentée de façon canonique uniquement. Nous aurions pu imaginer implémenter la même stratégie de gestion de la mémoire que pour l'érosion, mais nous nous serions attendu à des résultats similaires que pour l'érosion, et pour des raisons de temps, nous avons décidé de ne pas le faire. La seule optimisation qui a été faite a été d'utiliser l'instruction `fmaf` qui permet de faire une opération de type  $a \times b + c$  en un cycle.

## Résultats de l'implémentation avec CUDA

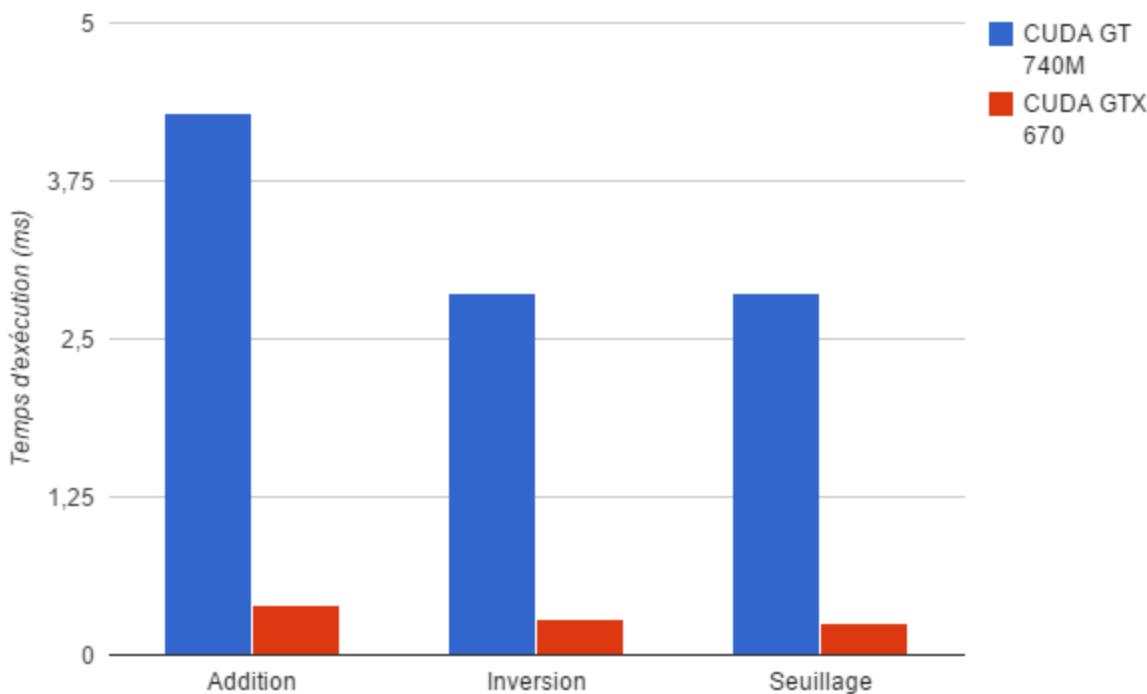
Un facteur jouant sur le temps d'exécution est le nombre de threads exécutés par block. Nous avons donc testé les différentes versions des trois algorithmes simples avec différents nombres de threads par block.



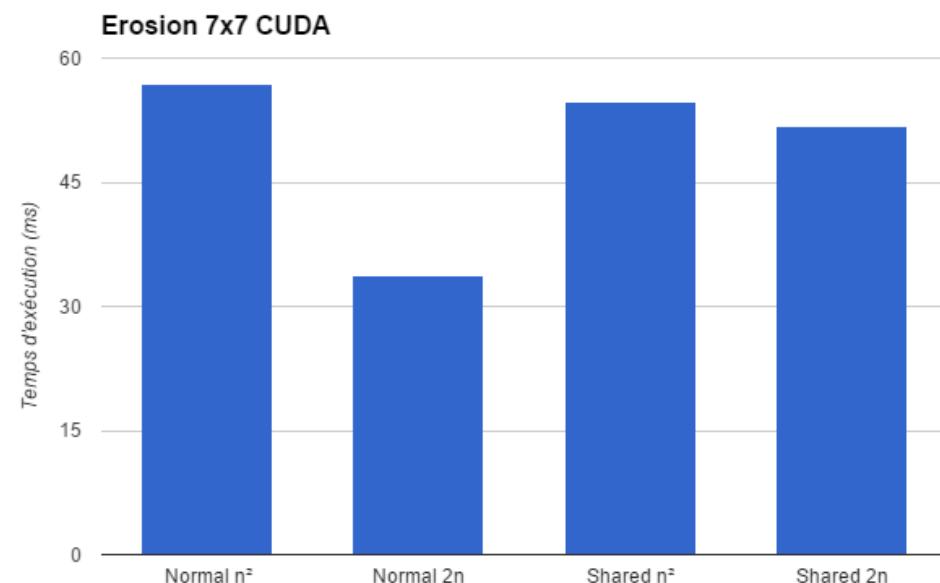
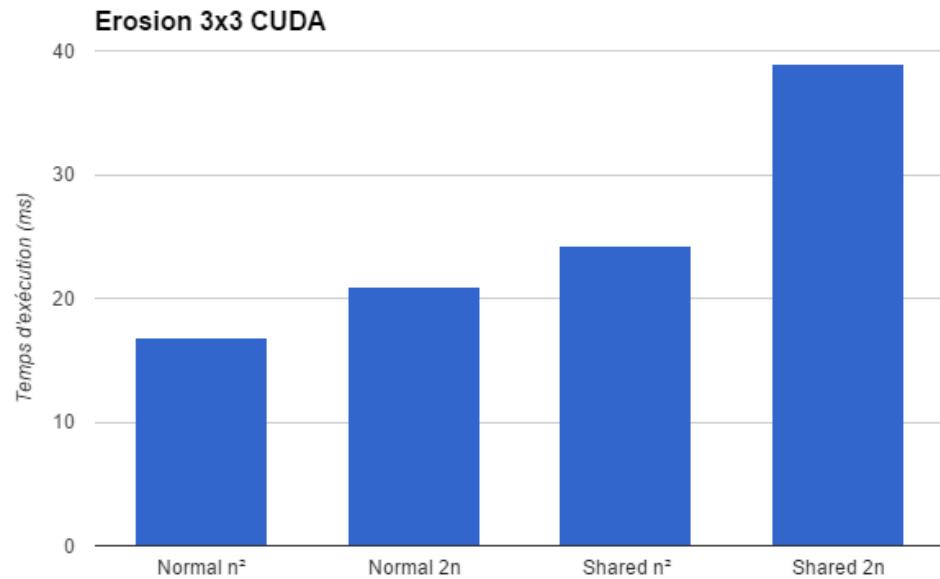
### Seuillage CUDA

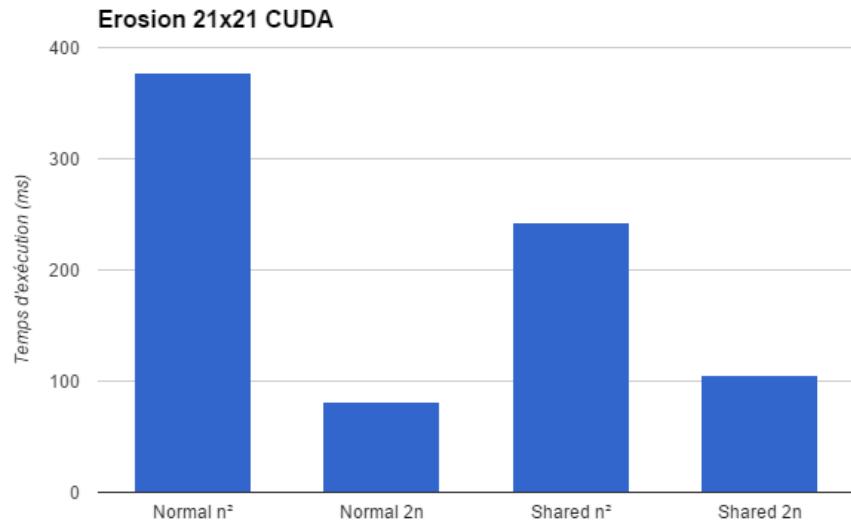


les valeurs de threads par block testées sont 128, 256 et 512. Ce sont les trois valeurs qui sont susceptibles de donner les meilleures performances. Un indice pour trouver les meilleures valeurs est la taille d'un warp : 32. Afin d'optimiser au mieux l'occupation des coeurs du GPU, le nombre de threads par block doit être multiple de cette taille.



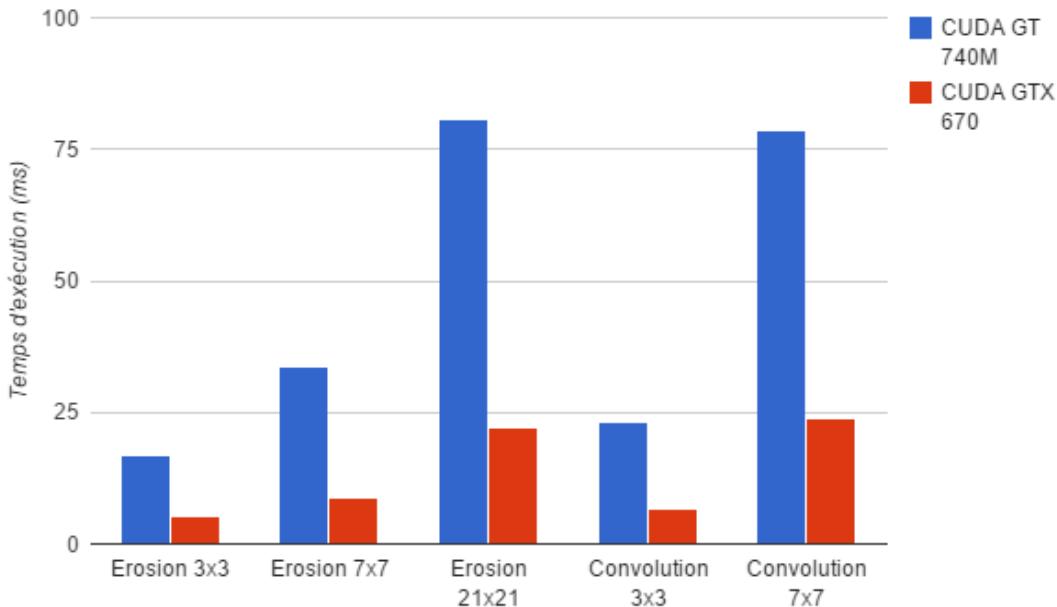
La différence entre les deux cartes graphiques est très conséquente. La GT 740M est une carte graphique de PC portable de moyenne gamme alors que la GTX 670 est une carte graphique haut de gamme destinée à un public type gamer et possède donc une puissance de calcul bien plus élevée, malgré que les deux GPU de ces cartes graphiques soient basées sur l'architecture Kepler (la GTX 670 possède tout de même 1344 coeurs à 915 MHz contre 384 coeurs à 810 MHz pour la GT 740M).





Ces graphiques montrent les temps d'exécution de l'érosion pour différentes tailles d'élément structurant en version normale et en version utilisant la mémoire partagée (notée *shared*) en lançant l'algorithme soit sur tout l'élément structurant (noté  $n^2$ ) soit en lançant une fois en ligne et une fois en colonne (noté  $2n$ ) (ce qui est possible car l'élément structurant est rectangle).

On remarque que la version  $2n$  est plus performante que la version  $n^2$  quand l'élément structurant est suffisamment grand. En effet, lorsqu'il est trop petit, le coût de lancer deux fois une érosion est trop conséquent par rapport au temps d'exécution du kernel. De plus la version qui place les données en mémoire partagée est également plus efficace quand la taille de l'élément structurant est grand pour des raisons similaires : le coût de placement des données en mémoire partagée est trop élevé par rapport au temps de calcul de l'érosion.



### 3.8. Implémentation sur GPU avec OpenCL

L'implémentation OpenCL se devait d'être multi-plateforme. C'est en effet l'avantage d'OpenCL qui est une librairie qui permet d'harmoniser la manière d'interagir avec le GPU entre tous les différents constructeurs de ces périphériques. Ainsi, nous avons développé une classe principale permettant l'initialisation d'OpenCL suivant le GPU (Ici, Intel ou NVidia). Nous avons forcé l'utilisation d'OpenCL 1.2 car la version OpenCL 2.0 d'Intel n'est encore qu'en beta. Cette classe contient aussi la possibilité d'effectuer un benchmark de chaque kernel envoyé. Cela nous permet donc d'obtenir le temps précis d'exécution côté GPU ainsi que le temps de transfert des données.

**Exemple de chargement des données pour le kernel addition :**

```
data1 = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
                      mem_size, output_image->data[c]);
data2 = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                      mem_size, input_image2->data[c]);
```

**Exemple de définition des arguments du kernel addition :**

```
clSetKernelArg(add_kernel, 0, sizeof(cl_mem), &data1);
clSetKernelArg(add_kernel, 1, sizeof(cl_mem), &data2);
clSetKernelArg(add_kernel, 2, sizeof(size_t), &size);
```

**Exemple du lancement de l'exécution du kernel addition, et de la récupération des données :**

```
clEnqueueNDRangeKernel(queue, add_kernel, 1, NULL, &global_ws, &local_ws,
                      0, NULL, &event);
```

```
clEnqueueReadBuffer(queue, data1, CL_TRUE, 0, mem_size, output_image->data[c],
0, NULL, &event);
```

*Local\_ws* représente le nombre de work-items local à un work-group.  
*global\_ws* représente le nombre de work-items global.

Pour les trois premiers filtres (addition, inversion et seuillage), l'optimisation se joue sur le groupement de données sur lequel on va exécuter chaque instance de notre kernel. La taille maximale des vecteurs sous OpenCL étant 16, chaque instance du kernel va effectuer l'addition, l'inversion ou le seuillage via une instruction prenant un vecteur de 16 données en entrée.

De plus, pour l'addition, OpenCL disposait d'une fonction permettant d'additionner et saturer dans le même temps l'addition en un cycle.

Pour le seuillage, nous avons dû effectuer une conversion en `float16`, c'est à dire un vecteur de 16 `float`, des données d'entrées (`uchar16`). En effet, l'instruction permettant la comparaison `isgreaterequal` n'acceptait pas le type `uchar` en entrée. Néanmoins, le résultat retourné étant soit 0 soit 1, la conversion en `uchar16` convertie automatiquement ces valeurs soit à 0 et soit à 255. Il est aussi bon de noter que les conversions sont presque négligeable en terme d'exécution. D'où le faible impact sur les performances d'exécution.

#### Exemple d'un kernel addition avec OpenCL :

```
__kernel void add(__global uchar16* datai1, __global uchar16* datai2, const int size)
{
    const int idx = get_local_size(0)*get_group_id(0) + get_local_id(0);

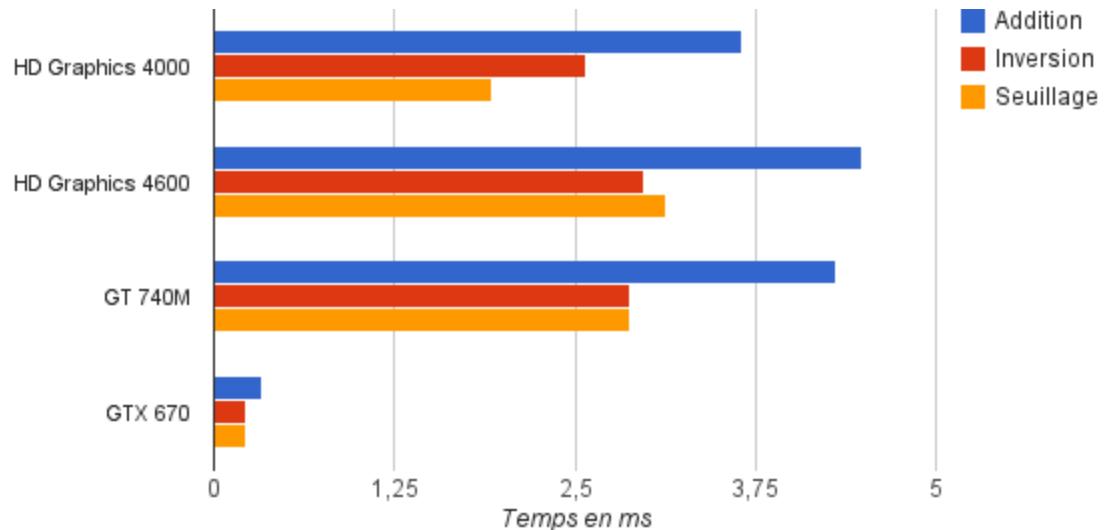
    if (idx < size) {
        datai1[idx] = add_sat(datai1[idx], datai2[idx]);
    }
}
```

Pour l'érosion et la convolution, contrairement aux 3 premières implémentations où nous avons nos données en une dimension (vecteur découpé en partie de 16 pixels), nous avons ici fait le choix de découper nos données en 2 dimensions pour le kernel. En effet, pour l'érosion et la convolution, chaque pixel dépend des pixels qui l'entourent. Ainsi, l'accès à la mémoire des blocks est optimisé.

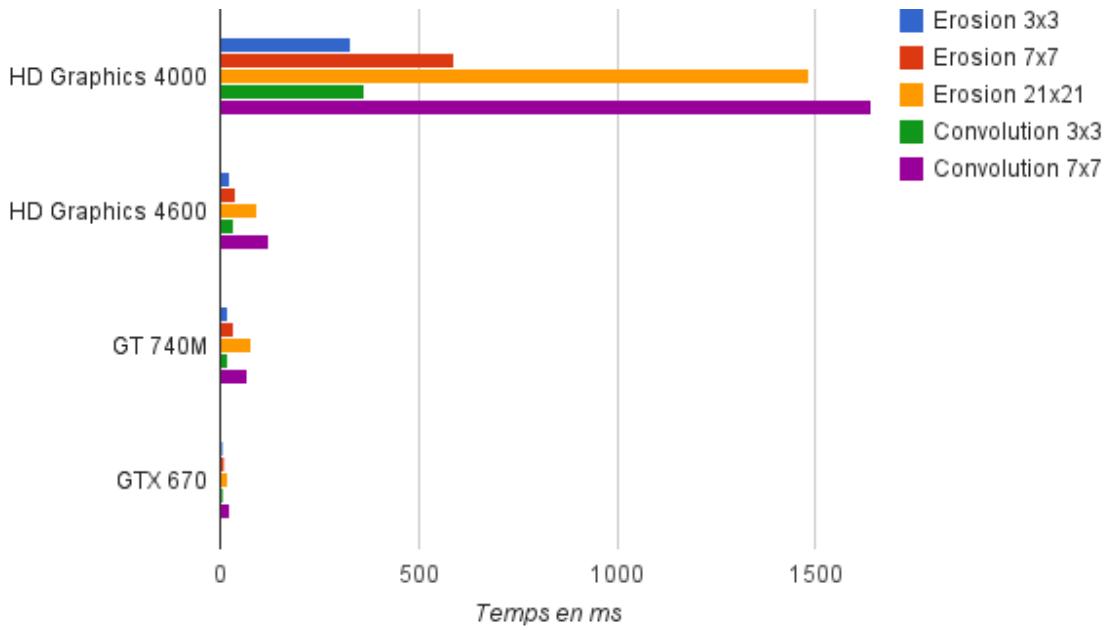
Dans le cas de l'érosion, plusieurs versions ont été implémentées. Une version qui traite tout l'élément structurant d'un coup (dite version  $n^2$ ) ainsi qu'une version effectuant d'abord une érosion en ligne puis en colonne (possible puisque l'élément structurant est rectangle) appelée version  $2n$ . Et pour chacune, une version en mémoire partagée, et une version sans mémoire partagée. La version mémoire partagée ne fonctionne pas sur les GPU Intel Graphics pour des raisons de limitations de nombre de threads synchronisés (16 dans ce cas).

## Résultats de l'implémentation avec OpenCL

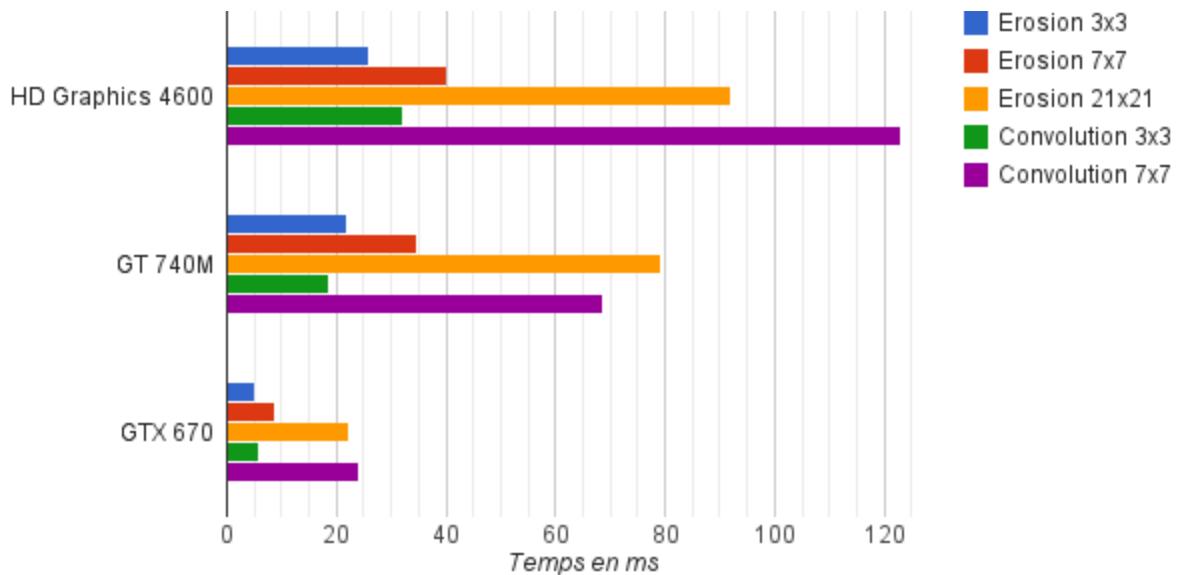
Voici donc les résultats que nous obtenons pour nos implémentations OpenCL. Ces benchmarks ont été réalisés sur deux GPU différents d'NVidia et deux GPU différents d'Intel. Ils tiennent compte uniquement du temps d'exécution. Tout d'abord les 3 premiers filtres :



Pour les filtres simples, la différence de performance entre deux HD Graphics est très faible. Néanmoins, le CPU accompagnant la HD Graphics 4000 (Core i5-3570k) étant plus performant que le CPU accompagnant la HD Graphics 4600 (Core i7-4700HQ), les performances sont meilleures sur la HD Graphics 4000. Côté NVidia, les deux GPU étant vraiment différents (un GPU de PC Portable (GT 740M) et un GPU de PC fixe (GTX 670)), les performances sont très largement meilleures sur la GTX 670. On gagne ainsi un facteur 13 en moyenne entre ces deux architectures. Nous pouvons remarquer que les temps d'exécution sur GT 740M et sur HD Graphics 4600 sont relativement équivalents.



Pour l'érosion et la convolution, des filtres plus complexe et donc comportant davantage d'instructions coté kernel, les performances changent beaucoup de ce qu'on pouvait avoir entre 2 GPU Intel sur les 3 premiers filtres. En effet, il y a un facteur au minimum de 12 en terme de performances entre les deux HD Graphics. Ceci s'explique par l'architecture du HD Graphics 4000 par rapport à celle du HD Graphics 4600. La grande différence peut s'expliquer par un nombre d'unités d'exécution différents (respectivement 16 et 20), ainsi que leurs architectures qui s'appuient pour l'un sur la famille Ivy Bridge et pour la HD Graphics 4600 sur la famille Haswell. Plus de détails sont apportés dans le rapport sur les architectures des GPU.



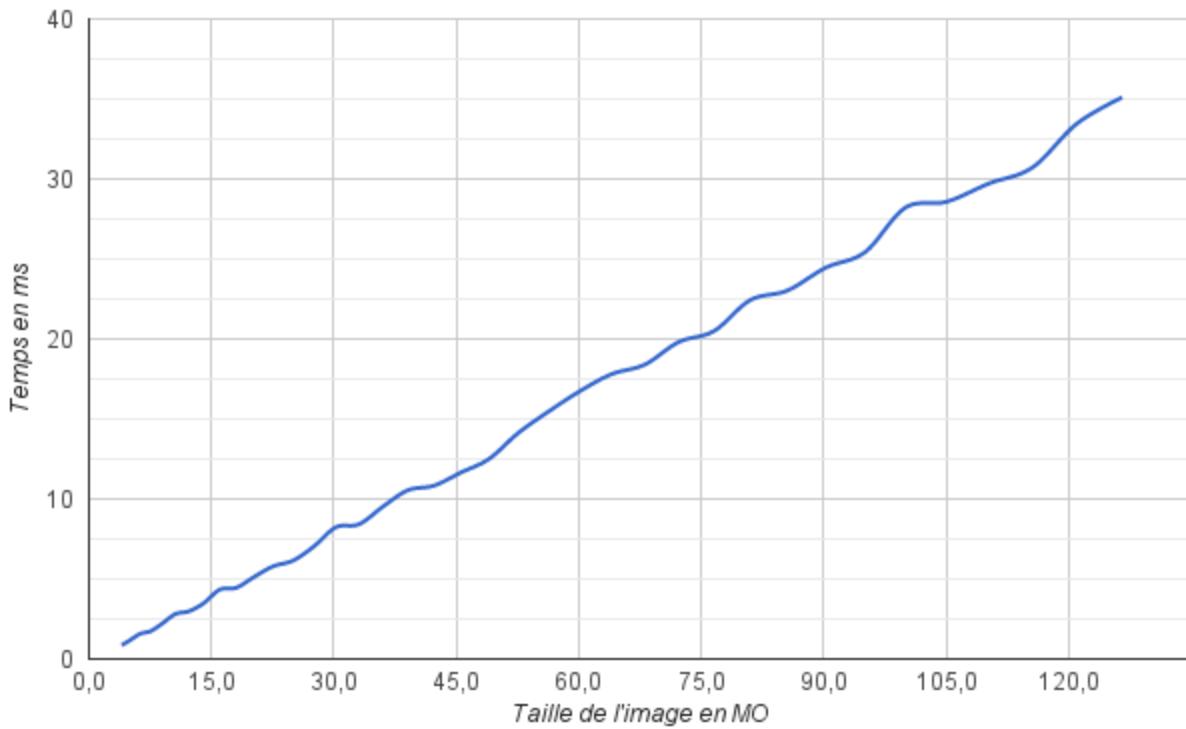
Voici ainsi une version sans la HD Graphics 4000 pour se rendre compte des importantes différences entre les différents GPU. Côté NVidia, la différence constatée s'explique par les différences de performances entre la GT 740M et la GTX 670.

Pour l'érosion, nous pouvons remarquer que multiplier par 2 le rayon ne multiplie pas par 2 le temps d'exécution mais presque par 1.5. Et pour un rayon 7 fois plus grand, on remarque que le temps d'exécution n'est multiplié que par 3.5.

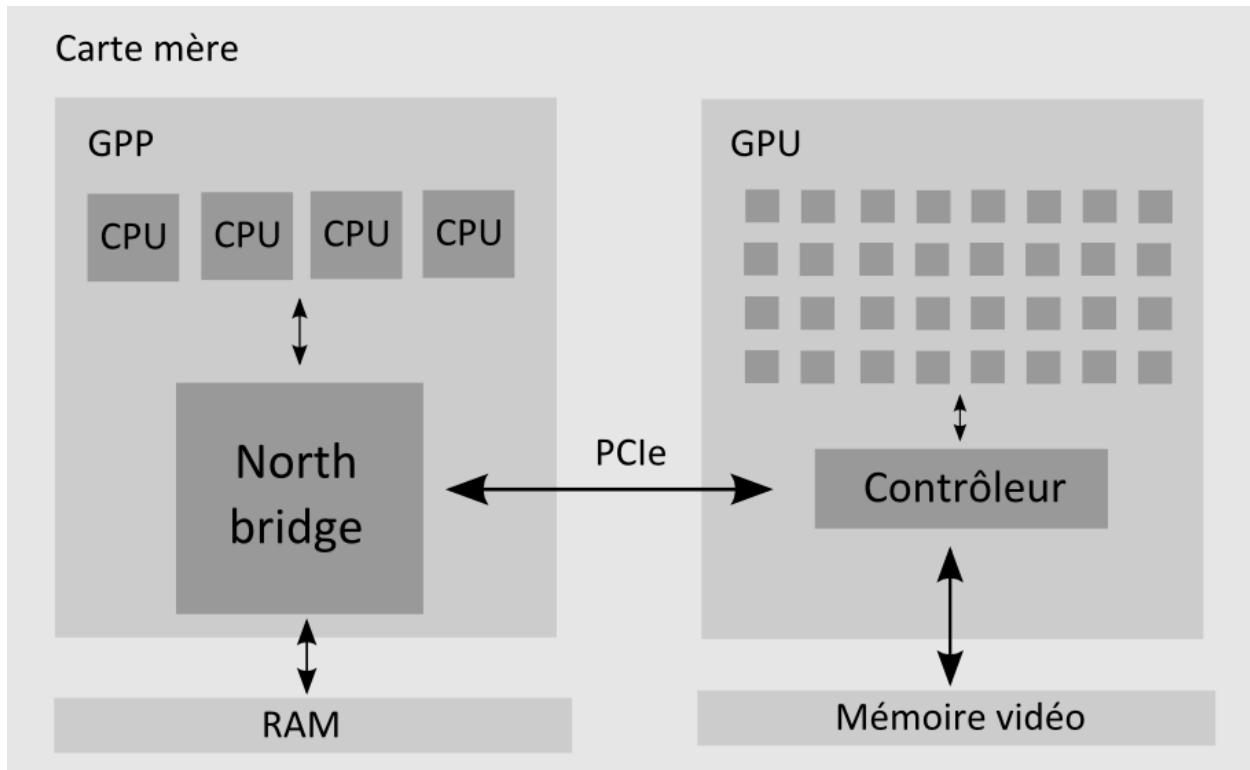
Côté convolution, la même observation ne peut être faite. En effet, dès que le rayon augmente, les calculs du kernel augmentent beaucoup et cela se justifie par l'algorithme même de la convolution malgré l'optimisation effectuée en exécutant une multiplication et addition dans le même cycle.

### 3.9. Conclusion sur les implémentations sur GPU

En conclusion de la partie GPU, il est important de revenir sur un point qui n'a que très peu été indiqué : le temps de transfert. En effet, ce point est certainement un des plus importants lorsque l'on compare performances GPU avec GPP. Les temps de transfert étant moindres sur GPP comparés à ceux sur GPU, si le gain n'est pas suffisamment conséquent sur GPU, il se peut qu'en ajoutant le temps de transfert, il soit plus intéressant de rester sur GPP. C'est ce que nous allons présenter par la suite. Ci-dessous, un graphique représentant le temps de transfert aller-retour d'une image de taille variable.

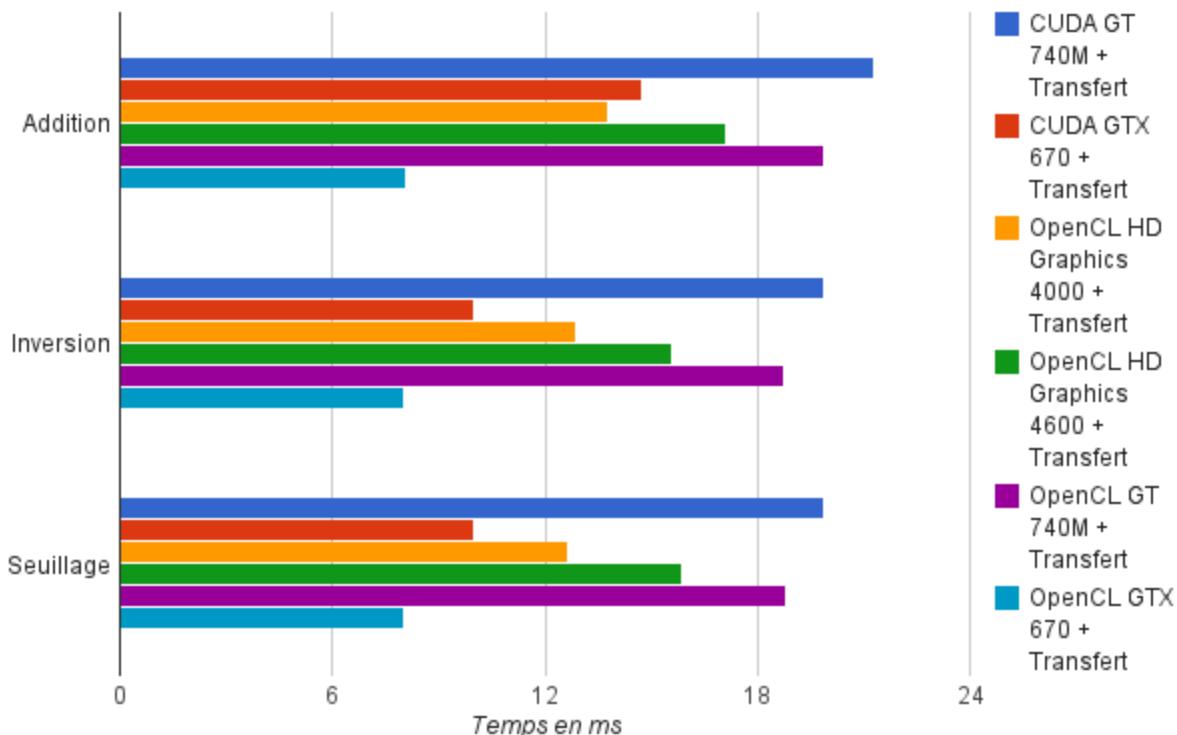
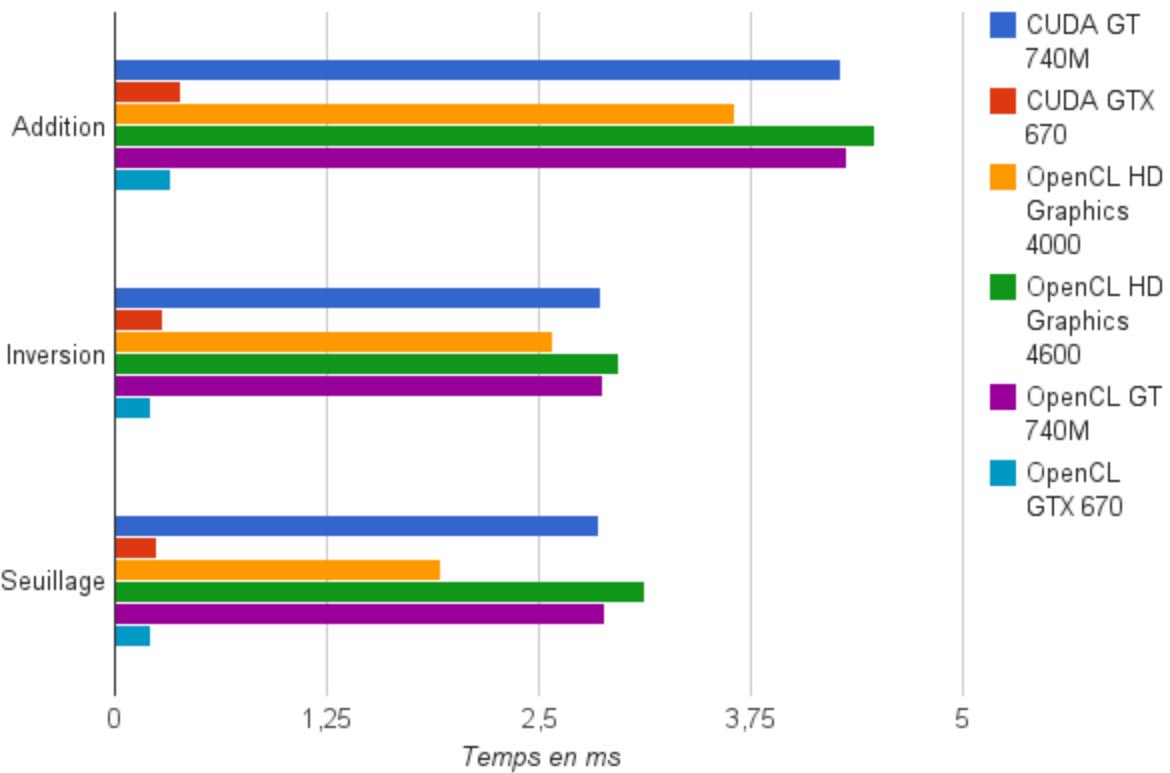


Le temps de transfert d'une image varie en fonction de l'architecture interne de l'ordinateur. Les transferts se font au travers du north bridge (aujourd'hui le plus souvent inclus au GPP) via PCIe (Peripheral Component Interconnect Express).

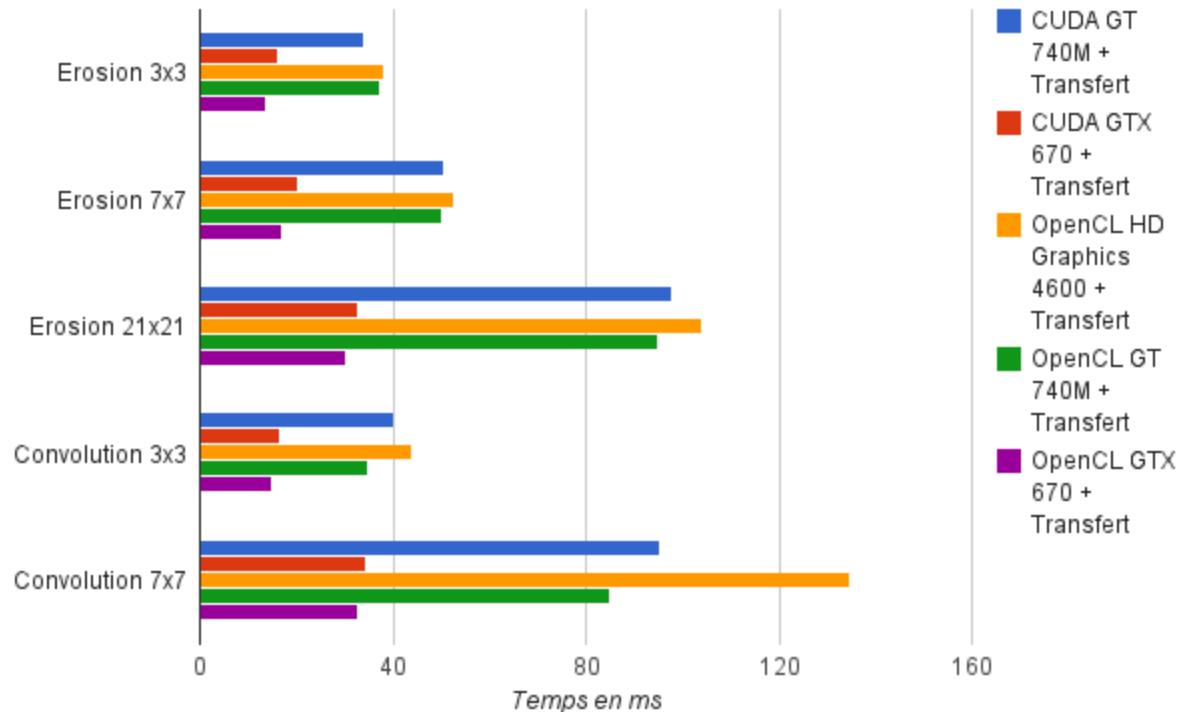
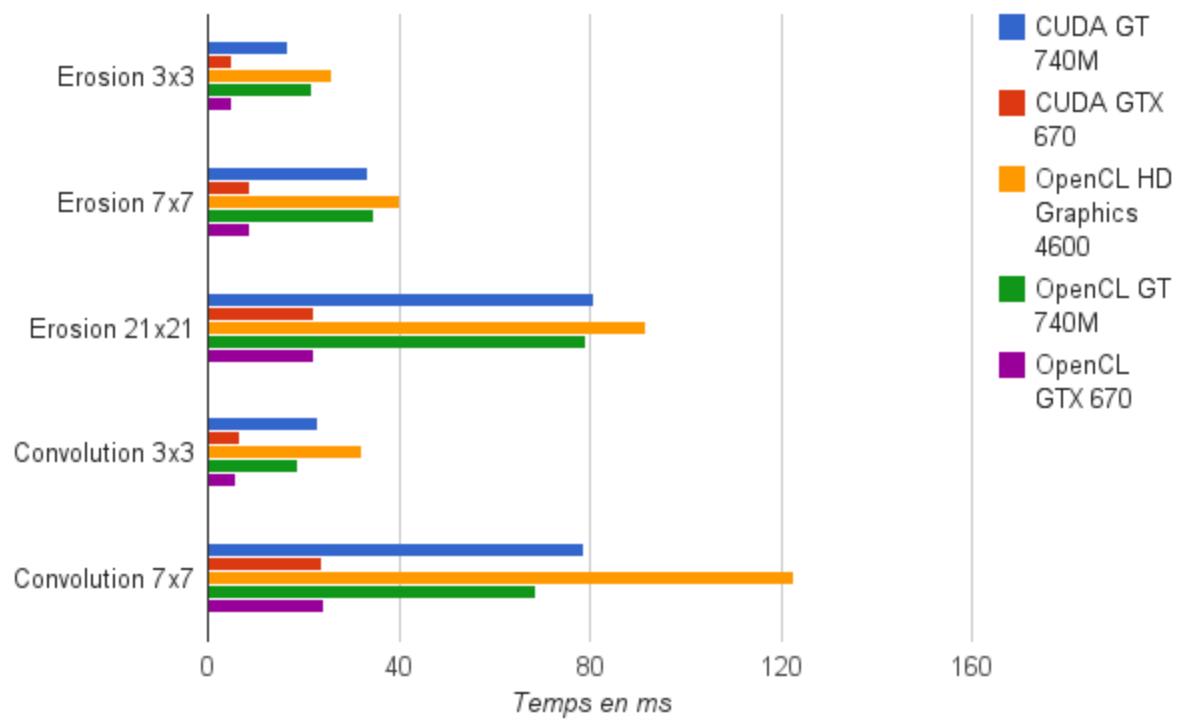


Pour le cas de l'ordinateur contenant la GTX 670 avec la HD Graphics 4000, le temps de transfert moyen était de 2,2ms pour l'aller et de même pour le retour. Ce temps était quadruplé pour l'ordinateur portable contenant une GT 740M avec une HD Graphics 4600. Ainsi en fonction des utilisateurs, les algorithmes peuvent prendre plus ou moins de temps. De plus, les temps de transferts pour un GPU Intel intégré et les temps de transferts pour un GPU dédié Nvidia sont différents. En effet, le temps de transfert pour l'ordinateur comportant la HD Graphics 4000 était en moyenne de 5ms aller retour, soit 1 ms plus lent que pour une exécution sur GPU NVidia malgré la proximité du GPU Intel du CPU. Toutefois, le temps de transfert pour l'ordinateur portable contenant la HD Graphics 4600 était de 12ms aller retour. Ainsi, le temps de transfert est plus faible de 4ms par rapport à une exécution sur le GPU NVidia dédié 740M. On conclut donc que la proximité physique des composants importe peu sur les temps de transferts malgré le fait que l'on pourrait être amené à penser que les transferts sont très court sur les GPU on-board, c'est à dire les Intel HD Graphics.

Comme indiqué précédemment, les benchmarks suivant comporteront deux versions, une avec les temps d'exécution ajoutés au temps de transfert de chaque périphériques et une sans les temps de transfert. Nous ne tiendrons pas compte ici des performances de la HD Graphics 4000 qui sont, comme montrées précédemment, très mauvaises dans le cadre de l'érosion et de la convolution.



Pour les trois premiers filtres dit simples, les performances entre l'implémentation CUDA et l'implémentation OpenCL est légère. Néanmoins, temps de transfert cumulés, les différences entre une implémentation OpenCL et une implémentation CUDA dépendent alors de l'architecture de l'ordinateur sur lequel est exécuté ces filtres.



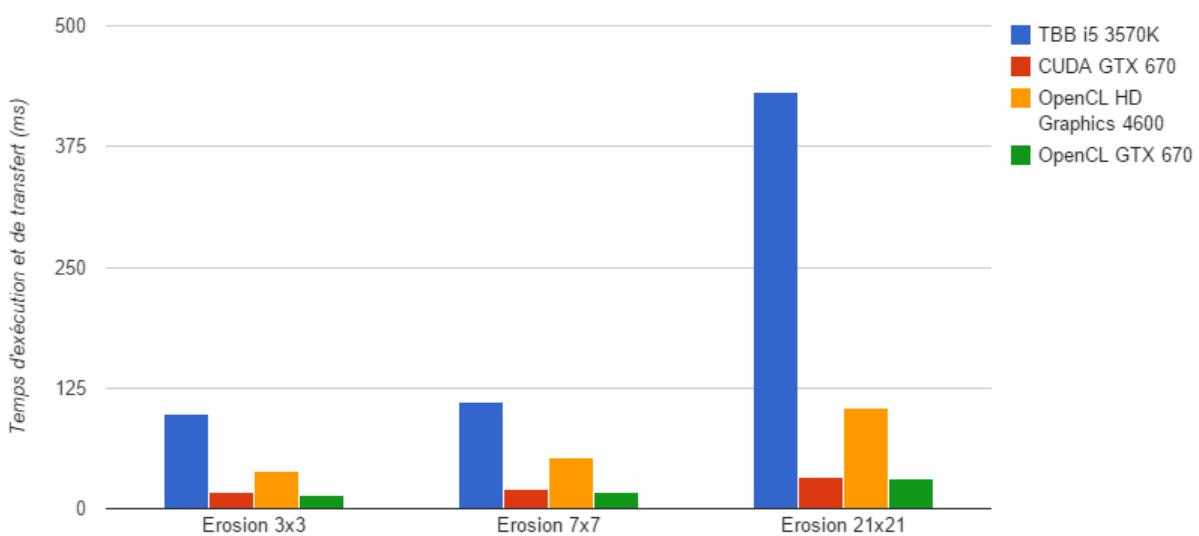
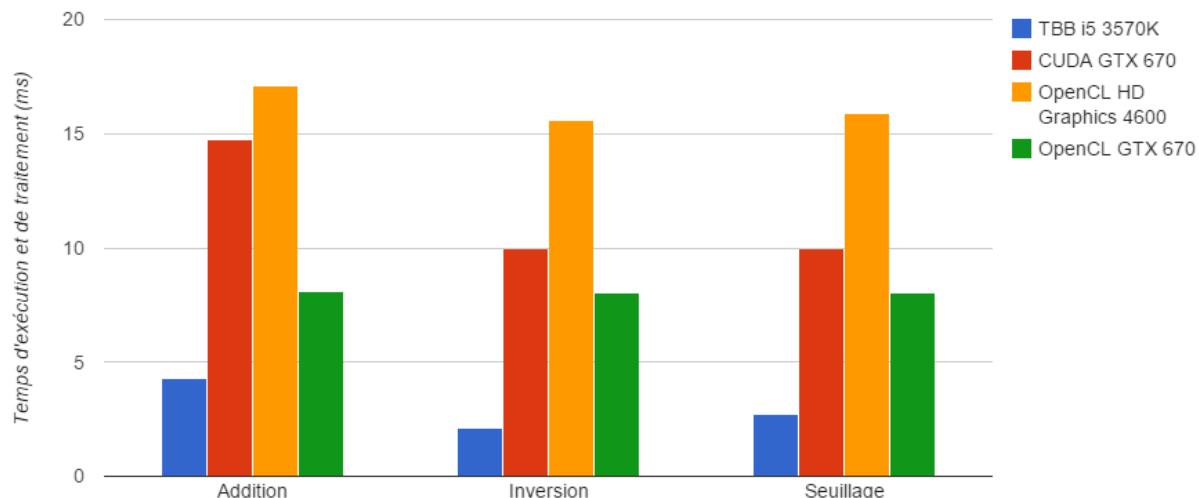
Le même syndrome se déroule pour les filtres plus complexes tels que la convolution et l'érosion. Les temps d'exécution des implémentations OpenCL et CUDA sur GPU NVidia sont à nouveau similaire. On peut remarquer une écart plus important entre l'implémentation OpenCL et NVidia ayant pour GPU la

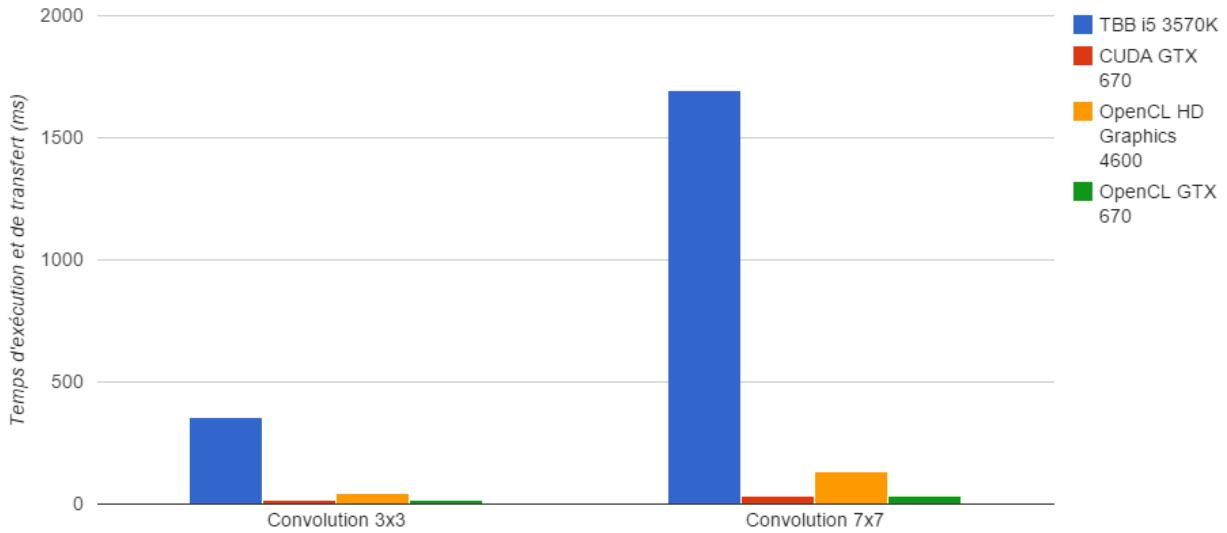
GT 740M. Là encore, les temps de transferts viennent montrer une équivalence de l'implémentation sur GT 740M avec une implémentation sur Intel HD Graphics 4000 / 4600. Ainsi, il semble nécessaire d'opter pour une implémentation sur GPU dédiée seulement si le GPU est bien plus puissant que le GPU intégré au CPU Intel et ce, pour une question de temps de transfert pour la version sur ordinateur portable.

Globalement, les implémentations OpenCL et CUDA ne varient que peu en performances. Rappelons l'avantage d'OpenCL qui est sa portabilité et permet d'être exécuté sur tout type de GPU, contrairement à CUDA qui est spécifique à NVidia et permet d'optimiser spécifiquement pour un GPU NVidia.

## 4. Conclusion

Voici un récapitulatif des meilleurs temps obtenus sur GPP (avec vectorisation et parallélisation), avec CUDA sur GPU NVidia et avec OpenCL sur GPU NVidia et Intel.





Il apparaît alors évident que le choix de l'architecture sur laquelle un algorithme doit être implémenté n'est pas anodin. Le principal facteur qui va entrer en jeu est le temps de transfert. En effet, pour des programmes s'exécutant rapidement sur GPU, les temps de transfert peuvent rendre le temps total de traitement supérieur au temps qu'aurait pris une exécution optimisée sur GPP (C'est ici le cas pour les filtres simples tels que l'addition, l'inversion et le seuillage).

Le choix des API de programmation est aussi important. On peut parier sur de la portabilité avec OpenCL ou être spécifique à une architecture (comme avec CUDA) ce qui permet parfois d'optimiser plus spécifiquement et donc plus efficacement un programme.

De façon générale, les algorithmes plus complexes comme l'érosion et la convolution seront toujours plus rapides lorsqu'ils sont implémentés sur GPU, que ce soit sur une carte graphique dédiée ou même sur un IGP. Ces derniers sont d'ailleurs, comme il est dit dans le rapport sur les marchés et architectures des GPU, de plus en plus courant, voir même omniprésents dans les PC actuels.

## 5. Références

- *Investiguer et évaluer la technologie GPGPU*, M. Colliot, Projet ENSICAEN 3A SATE;
- *Better Performance at Lower Occupancy*, Vasily Volkov, UC Berkeley, September 22, 2010.
- *How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures*, Gabriele Paoloni, Intel Corporation
- *NVIDIA's Next Generation CUDA™ Compute Architecture: Fermi™*
- *NVIDIA's Next Generation CUDA™ Compute Architecture: Kepler™ GK110*

## 6. Lexique

**ALU** = Arithmetic Logic Unit  
**API** = Application Programming Interface  
**APU** = Accelerated Processing Unit  
**CPU** = Central Processing Unit  
**EU** = Execution Unit  
**GPGPU** = General Purpose Graphics Processing Unit  
**GPP** = General Purpose Processor  
**GPU** = Graphics Processing Unit  
**IGP** = Integrated Graphics Processor  
**ILP** = Instruction Level Parallelism  
**PCIe** = Peripheral Component Interconnect Express  
**RAM** = Random Access Memory  
**SIMD** = Single Instruction Multiple Data  
**SSE** = Streaming SIMD Extensions  
**TBB** = Threading Building Blocks