TP4
OpenCL OpenACC

GIF 4104
Programmation parallèle et distribuée

Pier-Luc Auger (PLAUG2 - 910 098 011)
Mathieu Garon (MAGAR220 - 111 005 164)
Colosse USER46

Université Laval

Rapport TP4

Mise en contexte

Pour ce troisième TP, il est demandé d'écrire le code séquentiel fourni lors du TP2 pour filtrer une image de manière parallèle. Cette fois-ci, le but était d'utiliser la programmation massivement parallèle et de tirer profit de la puissance des grappes de calcul munies de GPU sur Helios. Dans les prochaines sections, des implémentations utilisant des outils tel que OpenCL et OpenACC vous seront présentés.

Méthode utilisée

OpenCL

Le choix de OpenCL comparativement a CUDA provient du fait que OpenCL à l'avantage de supporter une grande partie des GPU et CPU sur le marché. Son désavantage principal étant le nombre moins élevé de librairies à porter de main, cela n'affecte en aucun point le déroulement du projet présenté.

En premier lieu, les concepts de OpenCL sont comparables à ceux de OpenMPI. Il est à noter que OpenCL demande une grande quantité d'initialisation qui ne sera pas couverte en détail dans le cadre de ce rapport. Lorsque le tout est initialisé et paramétré, une fonction appelée "Kernel" doit être implantée. Il s'agit de la fonction qui va être exécuter par tous les fils du programme.

Initialisation

Plusieurs étapes devront être faites pour exécuter le Kernel, et celle-ci doit être paramétrée correctement pour assurer la portabilité du code et sa performance. Dans le cas du laboratoire présent, une seule "device" est utilisée, il s'agira du premier GPU de la liste, et si aucun GPU n'est présent, un CPU pourra être utilisé pour effectuer les calculs. L'étape suivante consiste à compiler le kernel. Pour le TP, des options de compilations sont ajoutées pour faciliter les modifications dans le programme. Par exemple, BLOCK_SIZE est défini en fonction du nombre de groupes utilisé dans le programme principale. Ensuite, le kernel est préparé pour être exécuté, trois buffers cl_mem sont utilisés en paramètre pour l'image d'entrée, le filtre et l'image de sortie, les deux autres paramètres sont des informations nécessaires comme la largeur de l'image et la taille du filtre. Finalement, l'envoi du kernel dans la liste de commandes doit être paramétré avec une quantité de groupe local et le nombre de fils global. Par défaut, le programme utilise des tailles de 32 x 32 pour les groupes locaux (pour des raisons de performances) et le nombre de fils total est déterminé par la taille de l'image : il s'agit du multiple de 32 le plus haut contenant au moins tous les pixels de l'image. Nécessairement, des fils seront calculés pour rien, mais en terme d'efficacité le gain est plus important.

Optimisations

Pour avoir un gain significatif sur le temps d'exécution du programme par un GPU, la mémoire se doit d'être mieux gérée. La première optimisation est de ne pas utiliser la mémoire global pour tous les paramètres de la fonction. En effet, le filtre par exemple est simplement utilisé en lecture par la totalité des fils. Puisque le fil ne risque pas d'avoir une taille significative, il peut être placé en mémoire constante pour éviter des accès vers la mémoire globale.

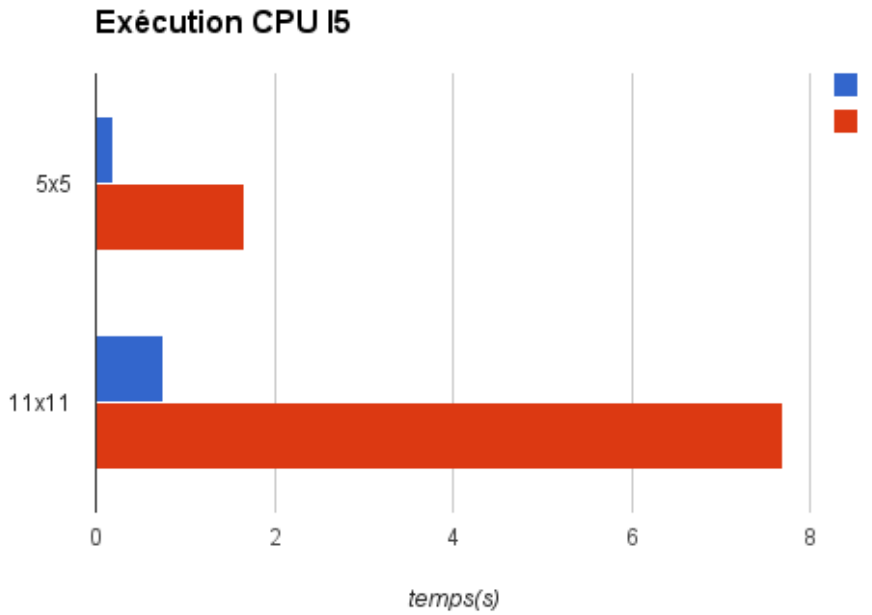
Ensuite, l'accès aux données de l'image en mémoire globale doit être minimisé. Pour ce faire, des parties de l'image doit être déplacé dans la mémoire local qui est plus rapide. Dans le cas d'une convolution, les accès à l'image sont connus donc il est facile de déplacer les données. La procédure utilisée est que chacun des fils contribue à placer au moins une donné dans le tableau local. Puisque la convolution demande d'utiliser les données autour du pixel traité en fonction de la taille du filtre, des données supplémentaires devront être gardées en mémoire. Des conditions (voir le code du kernel) sont vérifié pour s'assurer de placer les données dans le tableau pour permettre la convolution. Il est important d'utiliser des barrières locales pour assurer que tous les fils soient synchronisés lors des calculs.
Finalement, l'utilisation d'une variable int4 ainsi que de la fonction mad() (cette fonction optimise la fonction a*b+c en vitesse en perdant de la précision) permet d'optimiser le nombre d'instructions utilisé pour l'exécution des mathématiques relié à la convolution.

Améliorations possibles

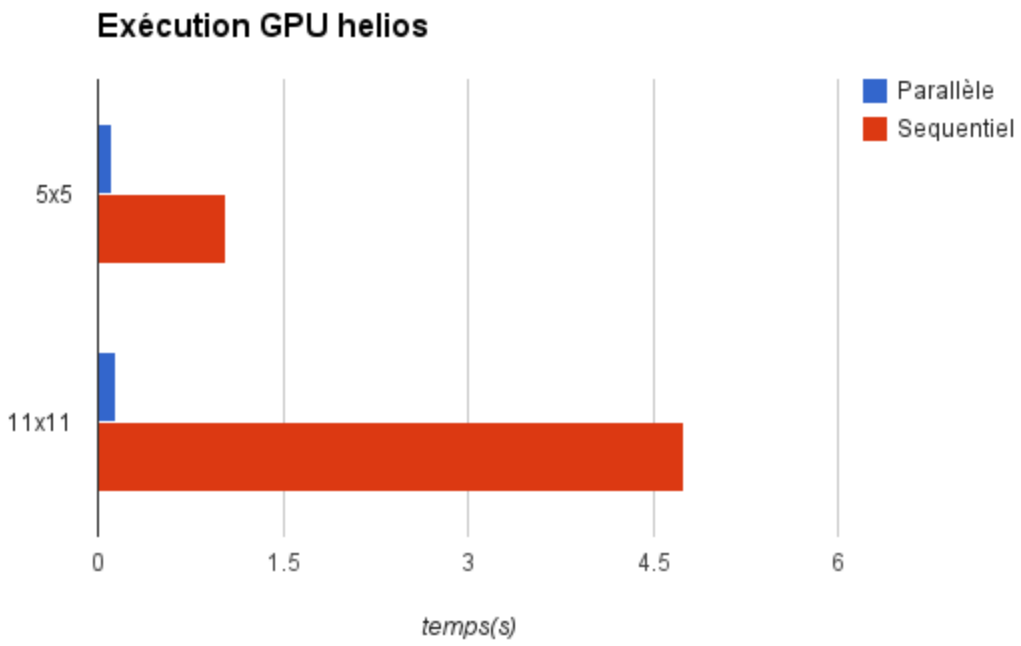
Finalement, il serait possible d'optimiser le code en modifiant l'architecture des données de l'image. En effet, le traitement des trois canaux avec des float4 par exemple pourrait minimiser le nombre d'instructions machines à exécuter.

Résultats

Pour la même image, deux filtres ont été utilisés, soit un filtre 5x5 et un filtre 11x11. Cela a pour effet d'augmenter le nombre de traitements numérique à faire dans chaque fils et donc de caché la latence de communication. De plus, le programme a été testé sur un GPU et sur un CPU afin de voir les gains apportés par le même algorithme.



Dans le cas présent, le speedup du filtre 5x5 est de 8.22 et celui du filtre 11x11 est de 10.1. Les valeurs obtenues prouvent l'importance de la latence en fonction du nombre de traitements. De plus, OpenCL offre une performance non négligeable pour l'utilisation des CPU.



Finalement, les performances pour le GPU sont comparables au CPU lors de l'utilisation d'un filtre de 5x5, par contre, l'avantage de celui-ci se reflète lorsqu'on augmente le nombre d'opérations à calculer pour les fils. En effet, le speedup passe de 9.5 à 31.1. Encore une fois, cela est dû au fait que le matériel est optimisé pour exécuter les opérations en parallèle et la latence de communication devient dans ce cas un goulot d'étranglement important.

OpenACC

Avec OpenACC, les directives #pragma interprétables à la compilation par pgc++ se sont avérées utiles. Pour être efficace, il faut gérer efficacement les transferts d'informations entre la mémoire CPU et la DRAM des cartes graphiques. Ayant ceci en tête, l'essentiel de la logique est la suivante:

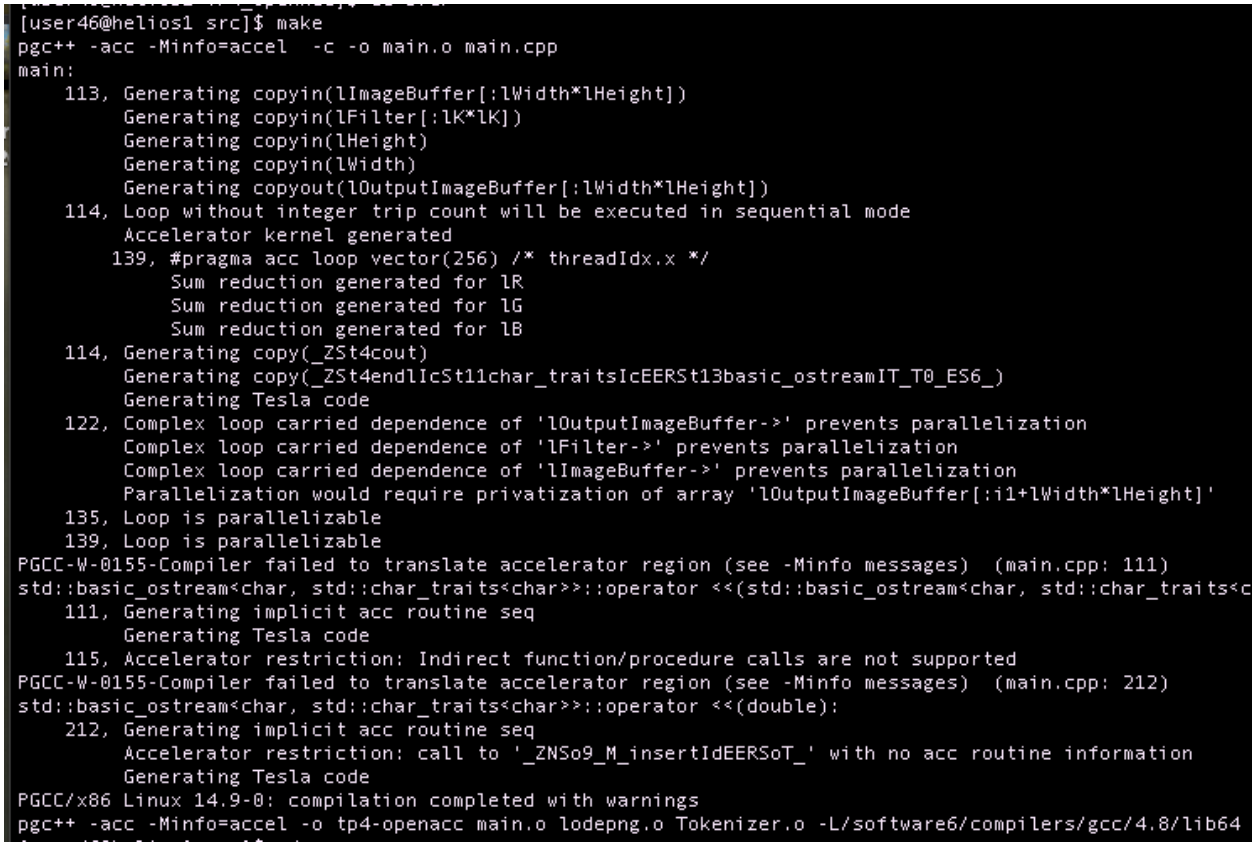
- Utilisation de `copyin` pour copier les valeurs qui seront seulement lues
- Utilisation de `copyout` pour les valeurs qui serviront de buffer de sortie

En effet, ces traitements sont incontournables, puisque autrement, les overheads sont beaucoup trop importants et le programme est trop lent puisque tout est recopié de la mémoire CPU vers les GPU. Non seulement y a-t-il perte de performance, mais le coût en espace mémoire est considérable. En utilisant copyin et copyout, le programme est plus efficacement découpé. En d'autres mots, c'est la manière de réduire les communications inutiles entre le "host" et les "devices".

Par contre, avec ces directives seulement, le code s'avère plus lent que le code séquentiel. En effet, il n'y a aucune parallélisation d'effectuée mais seulement des "overheads" engendrés par la communication entre le CPU et les GPU. En effet, il a fallut utiliser les directives de parallélisation pour accélérer l'exécution du programme:

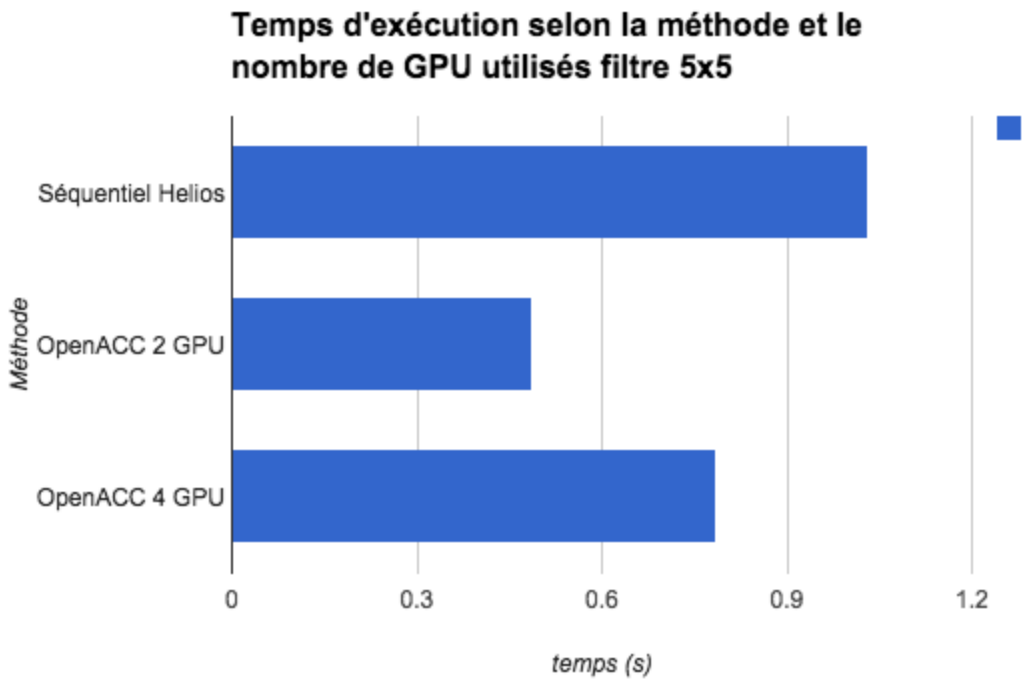
- Déclaration d'une région parallèle avec `acc region`
- Déclaration d'une boucle à paralléliser avec `acc loop independent vector`
- Déclaration d'une boucle interne à paralléliser avec `acc loop gang`
- Réduction avec `acc loop reduction(+:lR,+:lG,+:lB)`

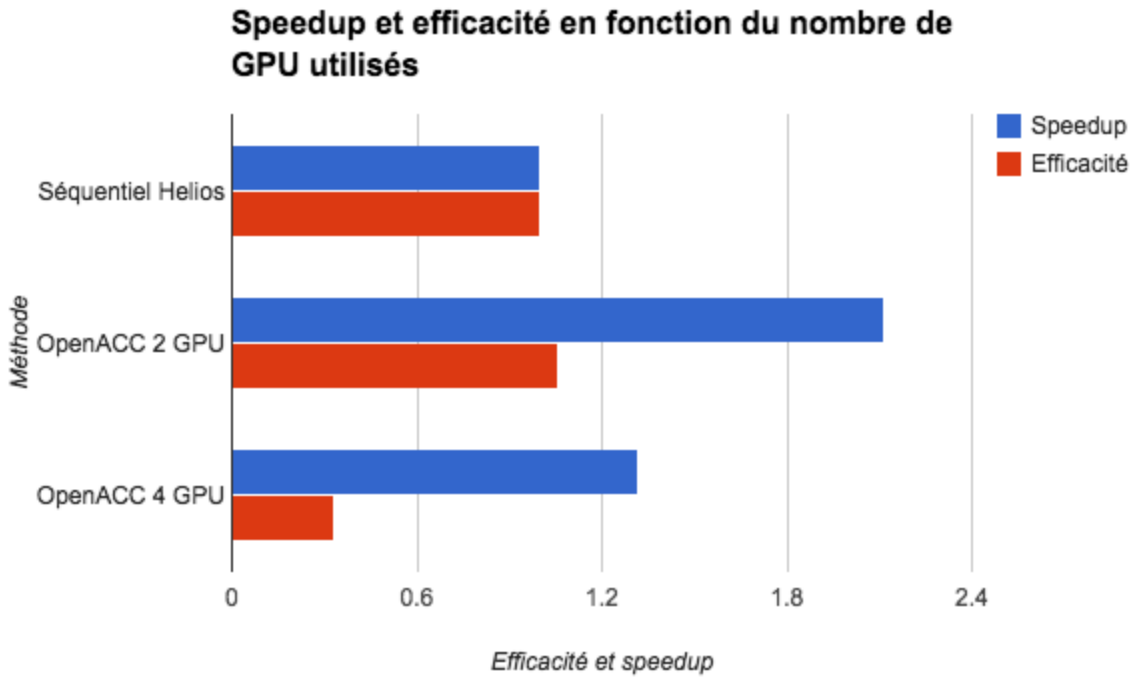
Pour arriver à ce résultat, l'utilisation du drapeau -Minfo=accel fut utile pour que le compilateur nous indique les améliorations faites et celles envisageables. En effet, ce genre de sortie fut très utile:

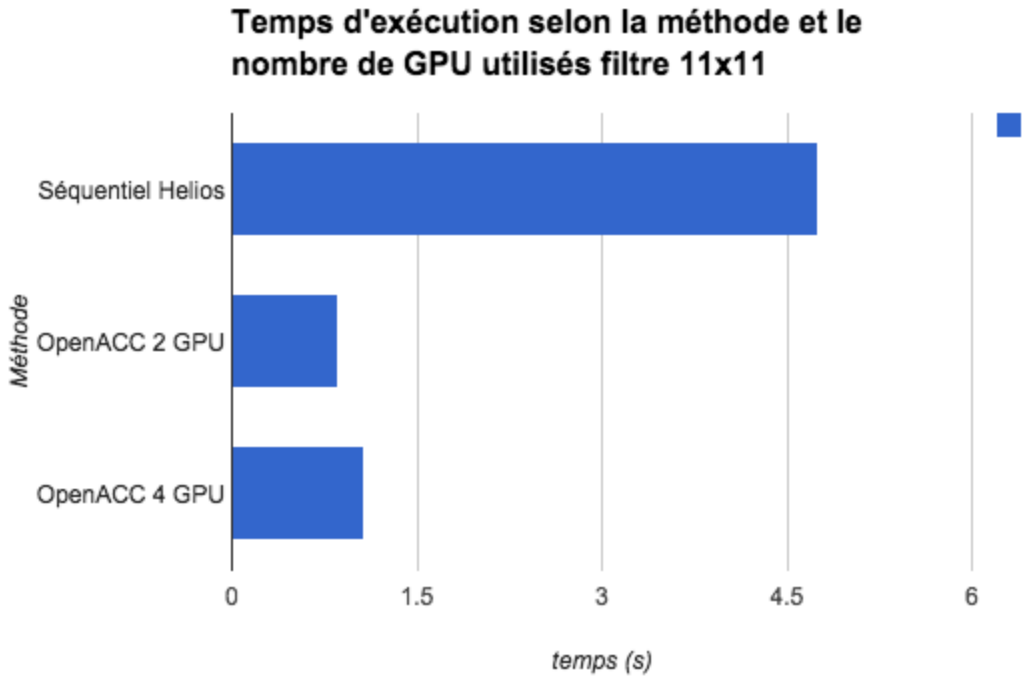


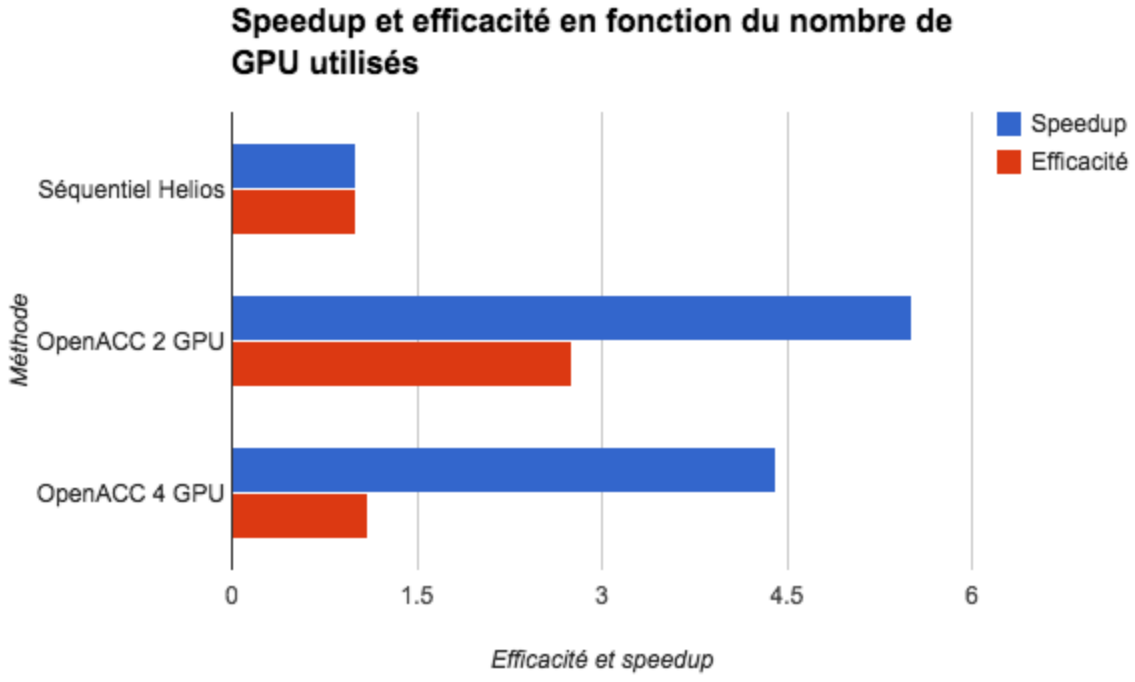
Pour le débogage, OpenACC s'est montré complexe à gérer. En effet, il était difficile de modifier le programme et de suivre les bogues autrement qu'en utilisant la sortie standard cout. Combiné au délai d'exécution sur Hélios, le développement s'est avéré fastidieux.

Résultats obtenus






Temps d'exécution selon la méthode et le nombre de GPU utilisés filtre 11x11
Séquentiel Helios
OpenACC 2 GPU
OpenACC 4 GPU
Méthode
0
1.5
3
4.5
6
temps (s)

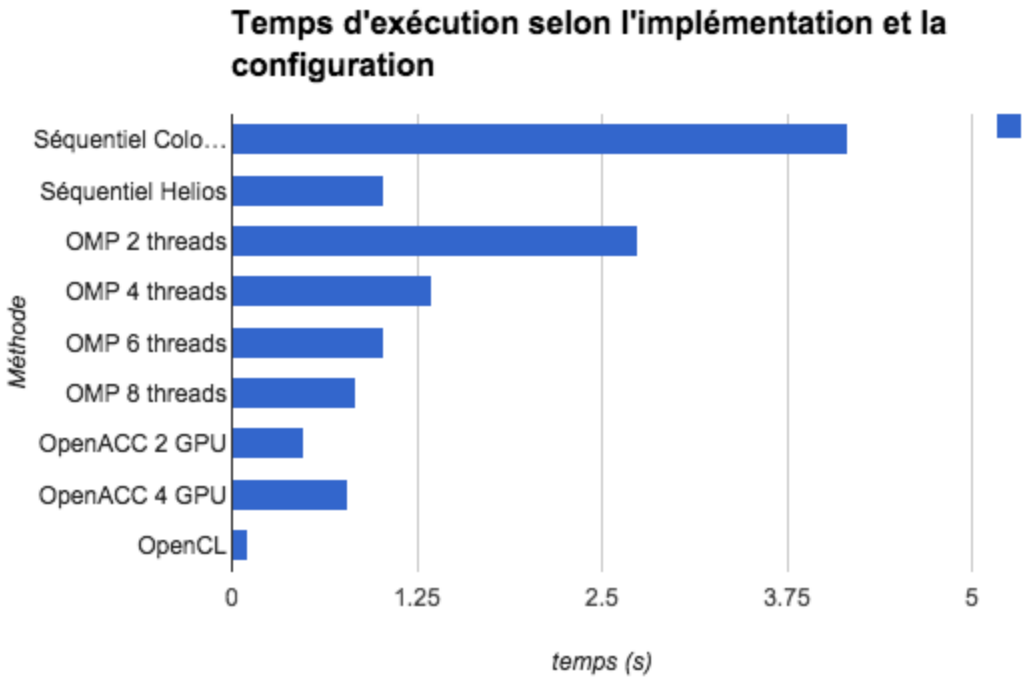

Speedup et efficacité en fonction du nombre de GPU utilisés
Speedup
Efficacité
Séquentiel Helios
OpenACC 2 GPU
OpenACC 4 GPU
Méthode
0
1.5
3
4.5
6
Efficacité et speedup

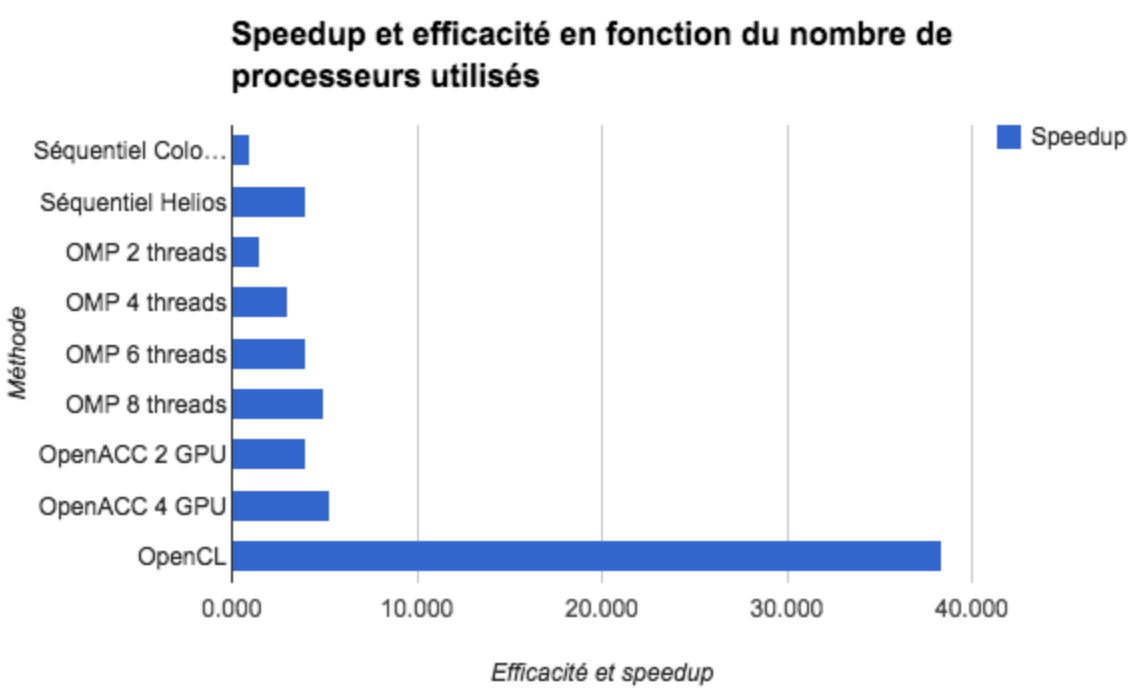
Analyse des résultats

En somme, on remarque bien que la programmation massivement parallèle avec GPU est très efficace pour la résolution de problèmes de ce genre. Les calculs de masse sur des matrice sont efficacement parallélisable et on s'en rend bien compte quand on prend compte les speedup obtenus, spécialement avec OpenCL dans notre cas.

Dans le cas d'OpenMP, les gains étaient considérables tout de même, mais OpenCL a livré des résultats d'un ordre de performance significativement plus grand.

Voir la page suivant pour des comparaisons graphique avec OpenMP pour la même image et un filtre 5x5.


Temps d'exécution selon l'implémentation et la configuration
Séquentiel Colo…
Séquentiel Helios
OMP 2 threads
OMP 4 threads
OMP 6 threads
OMP 8 threads
OpenACC 2 GPU
OpenACC 4 GPU
OpenCL
Méthode
0
1.25
2.5
3.75
5
temps (s)


Speedup et efficacité en fonction du nombre de processeurs utilisés
Speedup
Séquentiel Colo…
Séquentiel Helios
OMP 2 threads
OMP 4 threads
OMP 6 threads
OMP 8 threads
OpenACC 2 GPU
OpenACC 4 GPU
OpenCL
Méthode
0.000
10.000
20.000
30.000
40.000
Efficacité et speedup

Lancement de l’exécutable

OpenACC

Pour lancer le programme utilisant OpenACC, il faut aller dans le répertoire `/home/user46/GIF-4104/Remise/TP4/OpenACC` et exécuter la commande `make`. Un makefile est dans le répertoire. Les scripts dans le répertoire permettent de lancer le programme via Moab sur Helios.

OpenCL

Pour utiliser le programme, il s’agit d’utiliser `cmake` pour générer le makefile. Le CMakeList est fourni avec le code. Cela permettra ensuite d’utiliser la commande `make` pour compiler le programme.

Sur colosse, le code est déjà disponible et exécutable. Des scripts Moab sont dans le répertoire `/home/user46/GIF-4104/Remise/TP4/OpenCL`. Les scripts dans le répertoire permettent de lancer le programme.