<https://developer.nvidia.com/cuda-training>

Programmation parrallèle

La loi de Moore dit que les transistors doublent tous les 18 mois. Mais pas la vitesse ! Ceci est dû à la dissipation thermique qui s’empire au fur et à mesure que les transistors diminuent de taille. Si on néglige la dissipation thermique (pas de pâte thermique sous notre CPU, vitesse de l’horloge trop élevée) ça crâme ! On ne peut donc pas pousser trop loin la vitesse.

L’idée de base de la puissance : On veut remplir une piscine, on dispose de sceaux et d’une fontaine à quelques mètres, comment remplir la piscine le plus rapidement possible :

* On rapproche la piscine de la fontaine : moins d’allers-retours donc elle se remplit plus facilement. C’est compliqué à organiser car l faut déplacer la piscine, problème de place… *Dans le CPU il faut diminuer la distance des bus, pour que la communication entre les différents périphériques se fasse le plus rapidement possible (intérêt du cache). Compliqué car nécessite de modifier le hardware, non accessible au programmeur…*
* Marcher de plus en plus vite : Le temps pour faire un aller-retour est diminué. Mais on est vite limité par notre endurance. *On augmente donc la vitesse d’horloge du CPU. Mais il possède des limitations physiques qui l’empêchent d’aller trop vite (dissipation thermique)*.
* Appeler des amis pour nous aider : Plusieurs travailleurs en parallèle. *C’est aussi très compliqué à gérer. Le ILP (instruction level parralelism) comme le pipeling, branch perdictor augmenter de façon quadratique la complexité des CPU (grosse partie juste pour le contrôle) pour un gain linéaire en vitesse.*
* Porter plus de sceaux : Avec un aller-retour on a mis plus d’eau dans la piscine. Mais on ne possède pas assez de force pour porter beaucoup de sceaux. *Le CPU n’est pas assez puissant, et n’a qu’une poignée de cœurs à disposition. Il utilise donc le GPU qui est à disposition dans n’importe quel PC (en oubliant les chipsets graphiques pour PC portables). Qui avant n’étaient utilisés que pour le traitement vidéo !*

Les 3 choses qui bloquent la vitesse des CPU :

Les limitations techniques (dissipation thermique)

La vitesse des CPUs a trop dépassé la vitesse des mémoires (le plus lent dans un PC)

Le ILP est trop compliqué à gérer pour un gain pas si bon en terme de vitesse

<https://www.youtube.com/watch?v=KfGnLltyRH4>

D’où vient l’intérêt des GPU et leurs explosions en terme technique, des gamers !





Des millions de pixels sur chaque image. La couleur de chacun des pixels est calculée pour avoir la meilleure représentation graphique de notre monde. Génération de fumée, explosions, rendus des visages. Des milliards de pixels traités 60 fois par seconde.

Les CPUs sont très bon pour rendre des tâches « single thread » rapides. Dans les GPUs on ne s’intéresse pas à la vitesse d’un seul « thread » mais de tous. On s’intéresse à la vitesse de traitement de pixels, pas de pixel.

On se sert du GPU pour des tâches simples, et qui nécessitent du temps (boucles d’additions, de multiplications). Le CPU continue de gérer le reste plus compliqué (les instructions spécifiques, l’accès à la mémoire, la communication…).

Unité pour mesurer la performance d’un GPU

**FLOPS : FL**oating-point **O**perations **P**er **S**econd (vitesse de calcul)

BPS : Bytes Per Seconds (capacité de transfert)

En Computed Tomography : Création d’un volume 3D en scannant la personne. 28 000 personnes par an ont un cancer suite à CT.

CPU – 2h CUDA -2min

Réduction du temps de scan du patient, et donc diminution du risque de cancer du patient.

3 moyens d’accélérer ses applications :

OpenACC (simple d’utilisation): Permet aux programmeurs d’identifier les endroits de leur code qui nécessite une accélaration (qui est possible par CUDA). Il va donner des « hint » au CPU, on a pas besoin de modifier le code et le compilateur s’occupe tout seul (directives de préprocesseurs).

Librairies (accélérations « drop-in ») : Ne pas réinventer la roue. Utilisation de librairies existantes, notamment en traitement d’images.

<https://developer.nvidia.com/gpu-accelerated-libraries>

Programmation (flexibilité): Programmer ses propres Kernels perso.

Pour débugger : NVIDIA Parallel Nsight

<http://www.nvidia.com/object/nsight.html>

GTX 560M : <http://www.geforce.com/hardware/notebook-gpus/geforce-gtx-560m/specifications>

<http://www.notebookcheck.net/NVIDIA-GeForce-GTX-560M.48313.0.html>

CUDA architecture (Computed Unified Devide Architecture) : Permet la parrallélisation GPU. Permet d’exposer le GPU pour une application plus générale.

CUDA C/C++ : Une des façons d’utiliser les GPU. Basé sur le standard de l’industrie.

Extensions qui permettent la programmation

API (application programming interface) pour gérer des devices, mémoire etc…

<http://on-demand.gputechconf.com/gtc/2012/video/S0624-Monday-Introduction-to-CUDA-C.mp4>

Conférences :

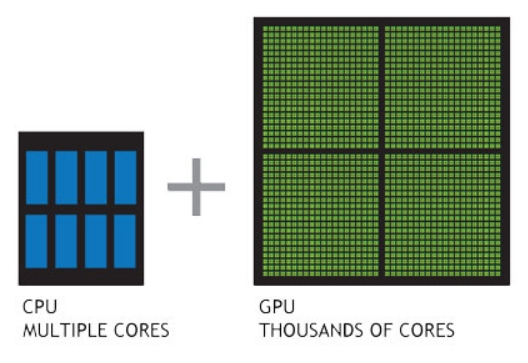
<http://on-demand-gtc.gputechconf.com/gtcnew/on-demand-gtc.php>

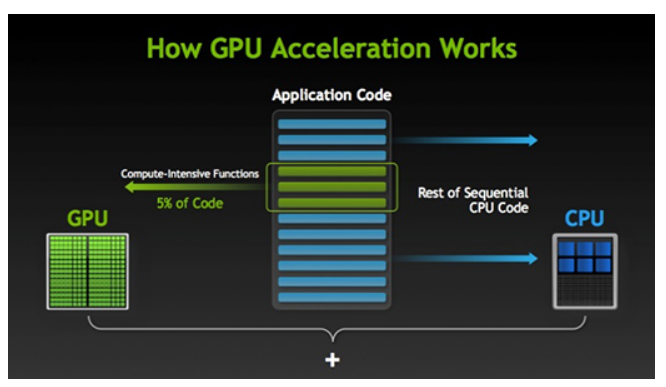
Concepts : Programmation hétérogène, Blocks, Threads, Indexing

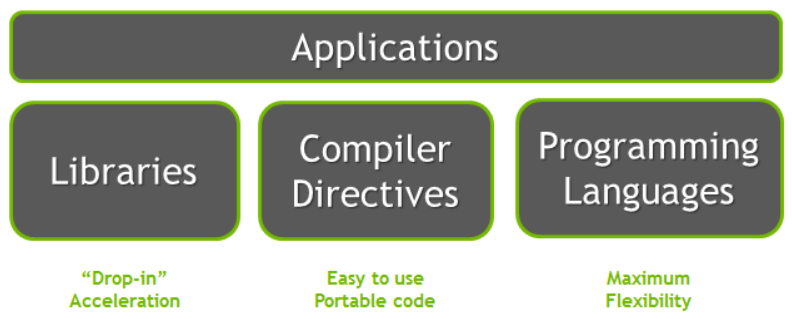
Programmation hétérogène :

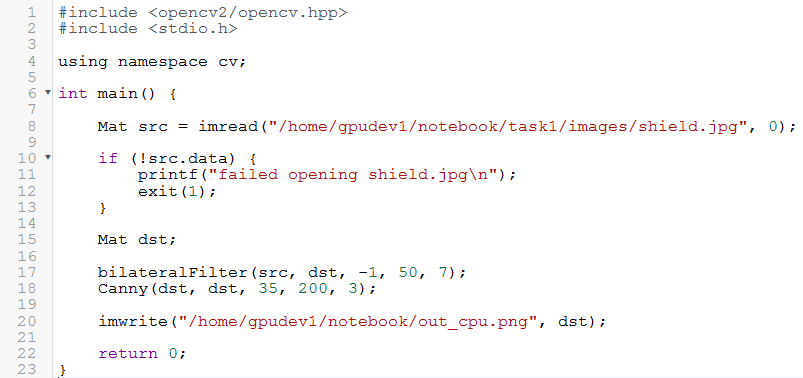
Hôte : CPU et sa mémoire

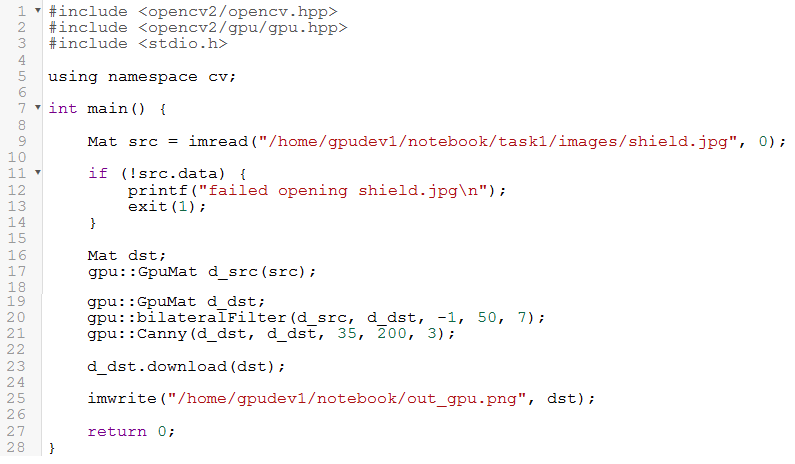
Dispositif : GPU et sa mémoire











Différence entre grids, blocs et threads.

<https://www.youtube.com/watch?v=KM-zbhyz9f4#t=93>

<https://www.youtube.com/watch?v=kzXjRFL-gjo>

Filtre Sobel :

<http://stackoverflow.com/questions/19838795/sobel-filter-in-cuda-cant-show-full-image>

Instal OpenCV :

<http://stackoverflow.com/questions/10901905/installing-opencv-2-4-3-in-visual-c-2010-express>

<https://www.youtube.com/watch?v=jCJ-S7hHgUw>

Pour la config 64 bits : <https://www.youtube.com/watch?v=e_TQ9c3n_d8>

Séparer le noyau du kernell :

<http://stackoverflow.com/questions/2090974/how-to-separate-cuda-code-into-multiple-files>

Pour filtrage par GPU :

Voir le projet « boxFilterNPP » situé dans :

C:\ProgramData\NVIDIA Corporation\CUDA Samples\v6.5\7\_CUDALibraries\boxFilterNPP

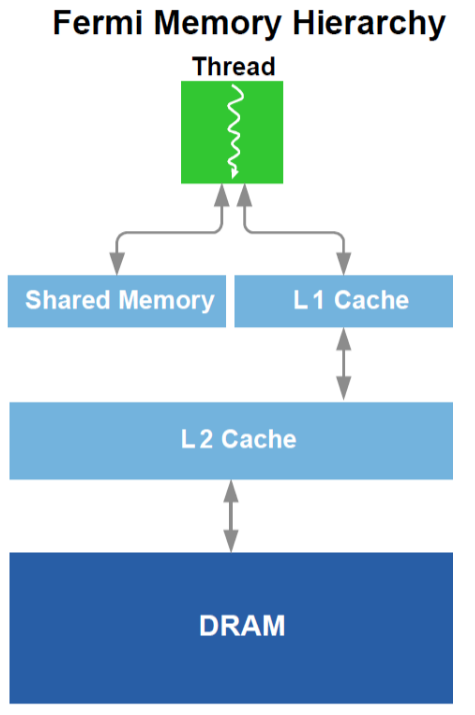
Les résultats sont dans :

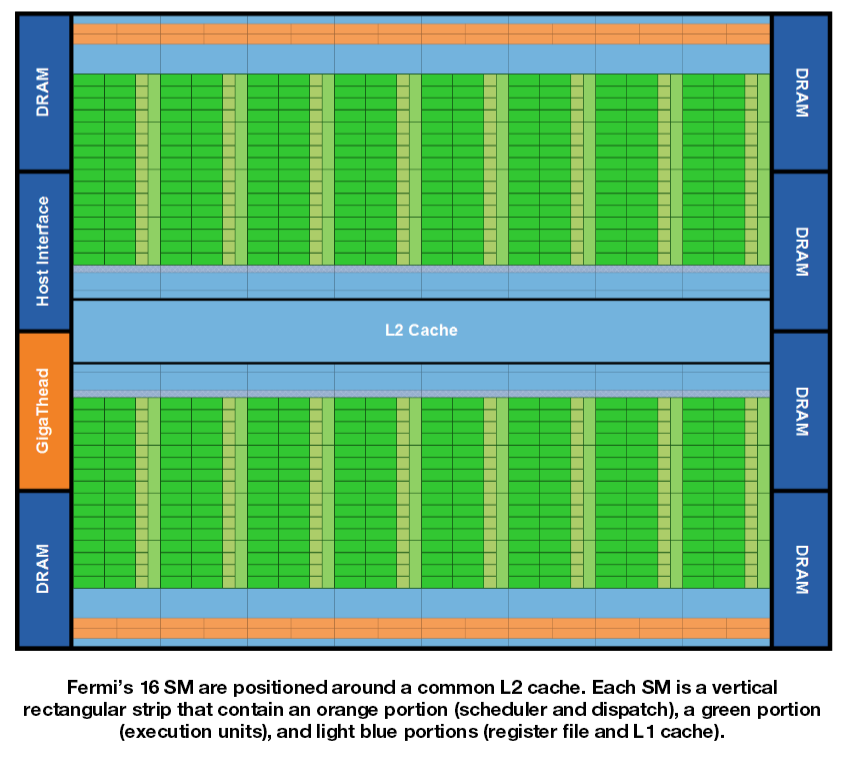
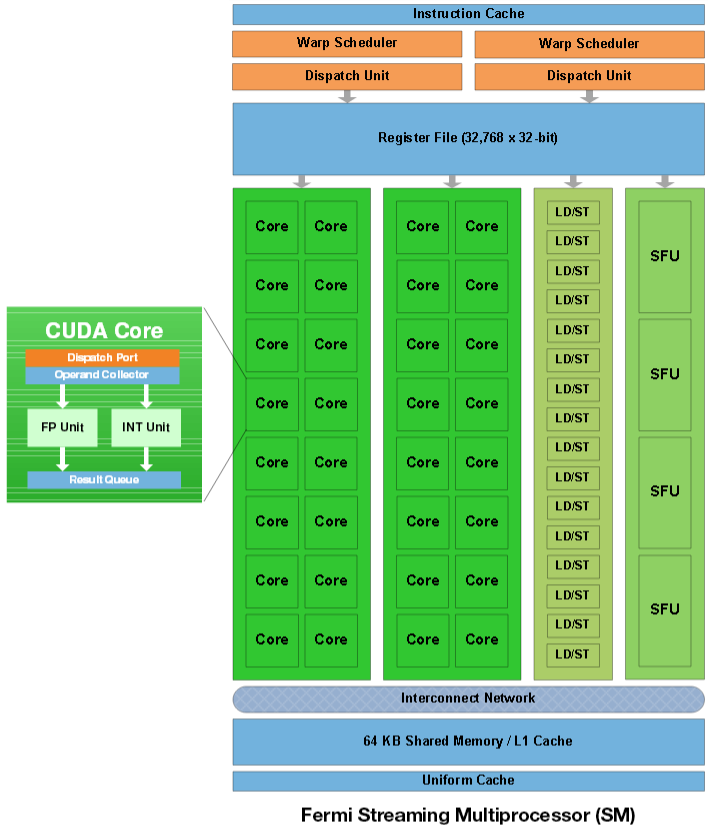
C:\ProgramData\NVIDIA Corporation\CUDA Samples\v6.5\common\data

Documentation sur la librairie NPP :  
<http://docs.nvidia.com/cuda/pdf/NPP_Library.pdf>

Architecture FERMI :

<http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf>





Le code est exécuté par groupe de 32 threads (1 warp) car depuis l’architecture Keppler, un SM contient 32 coeurs. 1 bloc correspond en gros à un SM. Si on a défini <<<1,128>>> alors le SM sera appelé 4 fois pour exécuter la fonction.

<https://www.pgroup.com/lit/articles/insider/v2n1a5.htm>

<http://www.tomshardware.fr/articles/CUDA-CPU-GPU,2-503-7.html>

Multiplication matricielle avec mémoire partagée :

<http://docs.nvidia.com/cuda/cuda-c-programming-guide/#shared-memory>

Idée pour traiter l’image : On divise l’image en 16 patchs. Chaque patch va dans la shared memory d’un SM, on fait le calcul de la convolution sur ce patch et on retourne le résultat.

Maxwell :

<http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf>

Pour l’analyse de performance :

*Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., ... & Dubey, P. (2010, June). Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In ACM SIGARCH Computer Architecture News (Vol. 38, No. 3, pp. 451-460). ACM.*

Temps execution filtrage de sobel horizontal 8bit non signé (en mode debug)

GPU :

Lena 256x256 diffms 0.107299998 float

Lena 512x512 diffms 0.107299998 float

Lena 1024x1024 diffms 0.107299998 float

Lena 2048x2048 diffms 0.380400002 float

Lena 4096x4096 diffms 1.45780003 float

Lena 8192x8192 diffms 5.74630022 float

C:\ProgramData\NVIDIA Corporation\CUDA Samples\v6.5\common\data

CPU :

Lena 256x256 diffms 4.88000011 float

Lena 512x512 diffms 18.3099995 float

Lena 1024x1024 diffms 69.7900009 float

Lena 2048x2048 diffms 275.600006 float

Lena 4096x4096 diffms 1039.00000 float

Lena 8192x8192 diffms 4149.60010 float

Résultats :

C:\Users\Lo\Documents\VisualStudio2013\Projects\Filtrage\_Sobel\_GPU\Filtrage\_Sobel\_GPU

Normalement il faut changer le noyau de convolution pour traiter une image de plus grande résolution. Pour les tests on s’est limité à garder le même noyau pour avoir une même base de comparaison, mais ce qu’il faut comprendre c’est que dans la réalité il faut utiliser un noyau plus gros et donc le temps de calcul augmente exponentiellement !

Stress test

Effectué avec une tablette de refroidissement ce qui ralentit l’augmentation de la température et améliore la dissipation thermique à vide du système. Avec CPUID Hardware Monitor, pendant 5min.

GPU :

Début à 43°C (0.820V) jusqu’à 73°C (1V) pour redescendre à 50°C.

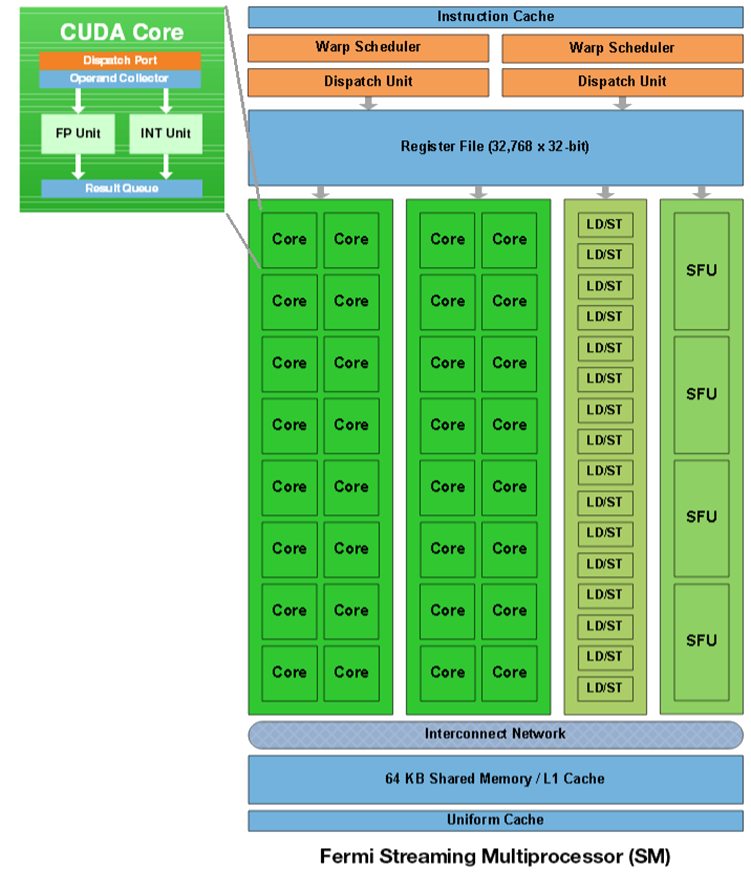
CPU :

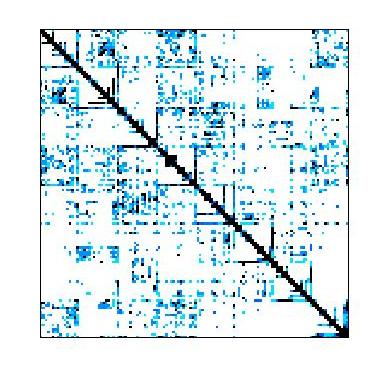
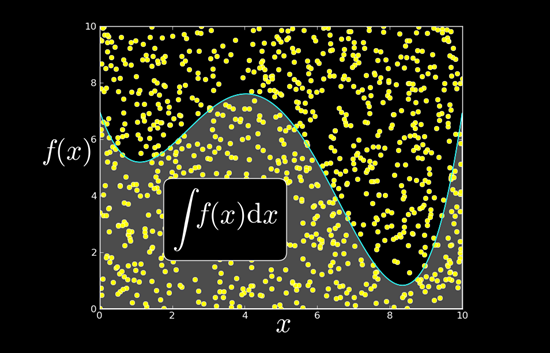
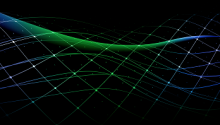
Début à 51°C (0.781V) jusqu’à 64°C (1.211V) pour redescendre à 51°C

Ouverture de la question sur le FPGA, plus appliqué dans un contexte embarqué (consomme moins) et est beaucoup plus versatile (avec FPGA, une patte peut être n’importe quoi). CPU/GPU plus dans un contexte « fixe », tout le monde a accès au GPU sans devoir créer sa propre carte (contrairement à FPGA). La carte graphique est versatile dans la programmation et est plus accessible, le FPGA sera plutôt utilisé dans les systèmes embarqués où lorsque l’on a programmé on y touche presque plus (à part quelques modifications mineures).

Infos sur GPU computing :

<http://www.info.univ-angers.fr/~richer/cuda_crs4.php>



[](http://code.google.com/p/thrust/downloads/list)

NVIDIA cuRAND

C++ STL Features for CUDA

NVIDIA cuBLAS

NVIDIA cuSPARSE

NVIDIA NPP

NVIDIA cuFFT

NVIDIA Math

Librairie « math.h »

pour le GPU

Traitement de données parallèle

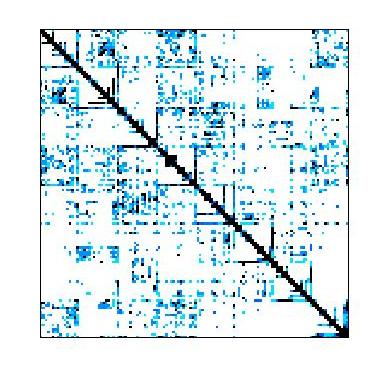
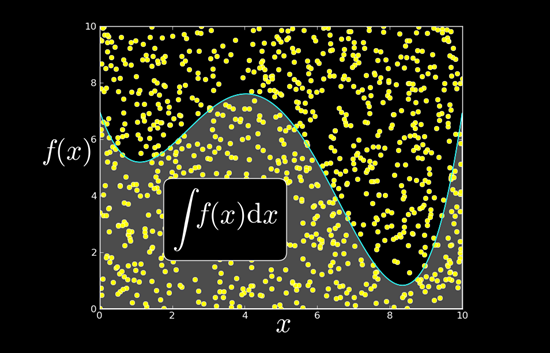
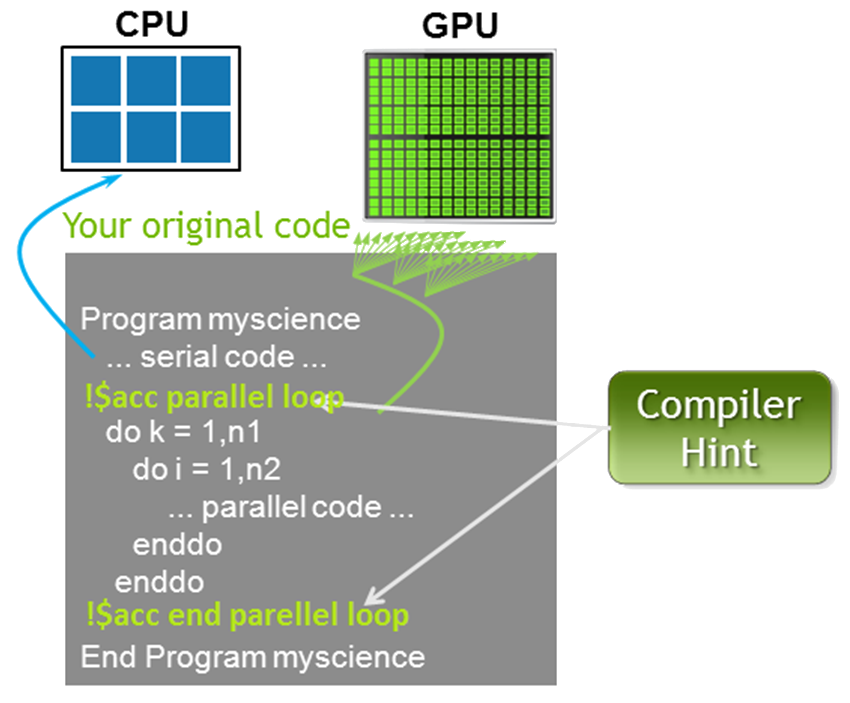
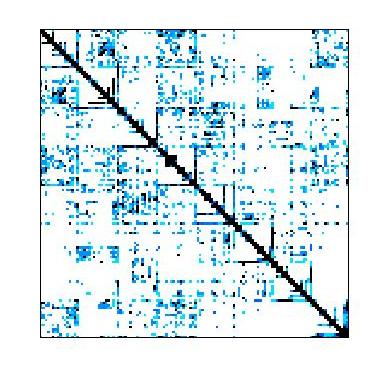
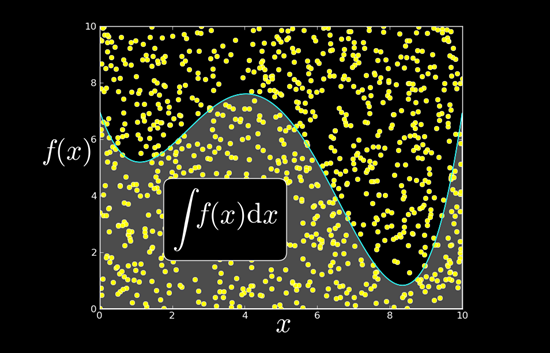
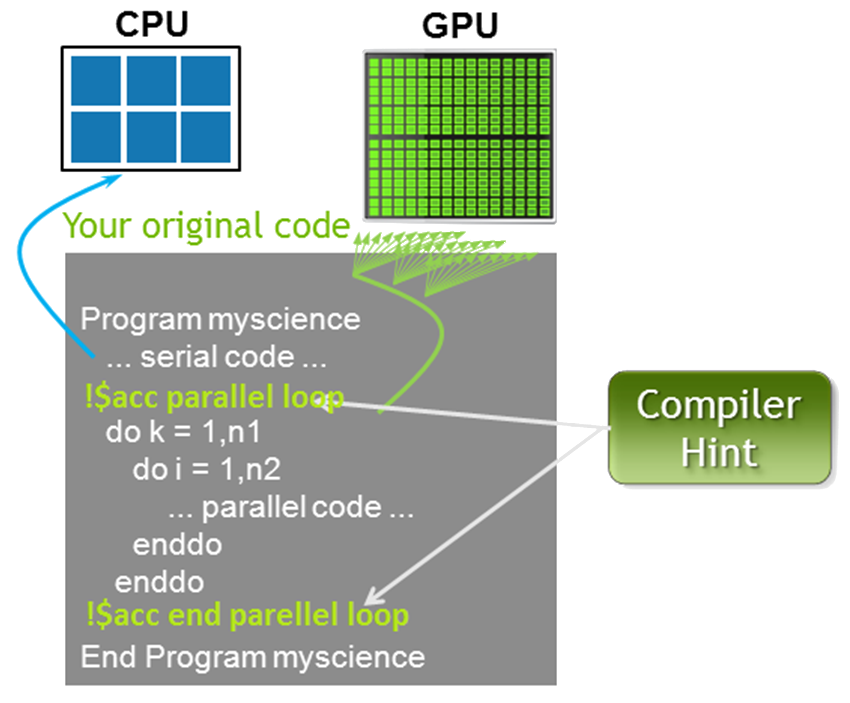
Transformation de Fourrier rapide

Algèbre linéaire creuse

Traitement d’images et de vidéo

Génération de nombres aléatoires

Algèbre linéaire classique

[](http://code.google.com/p/thrust/downloads/list)[](http://code.google.com/p/thrust/downloads/list)

C++ STL Features for CUDA

NVIDIA cuRAND

Génération de nombres aléatoires

NVIDIA cuSPARSE

Traitement de données parallèle

Algèbre linéaire creuse

NVIDIA cuRAND

C++ STL Features for CUDA

NVIDIA cuSPARSE

Traitement de données parallèle

Algèbre linéaire creuse