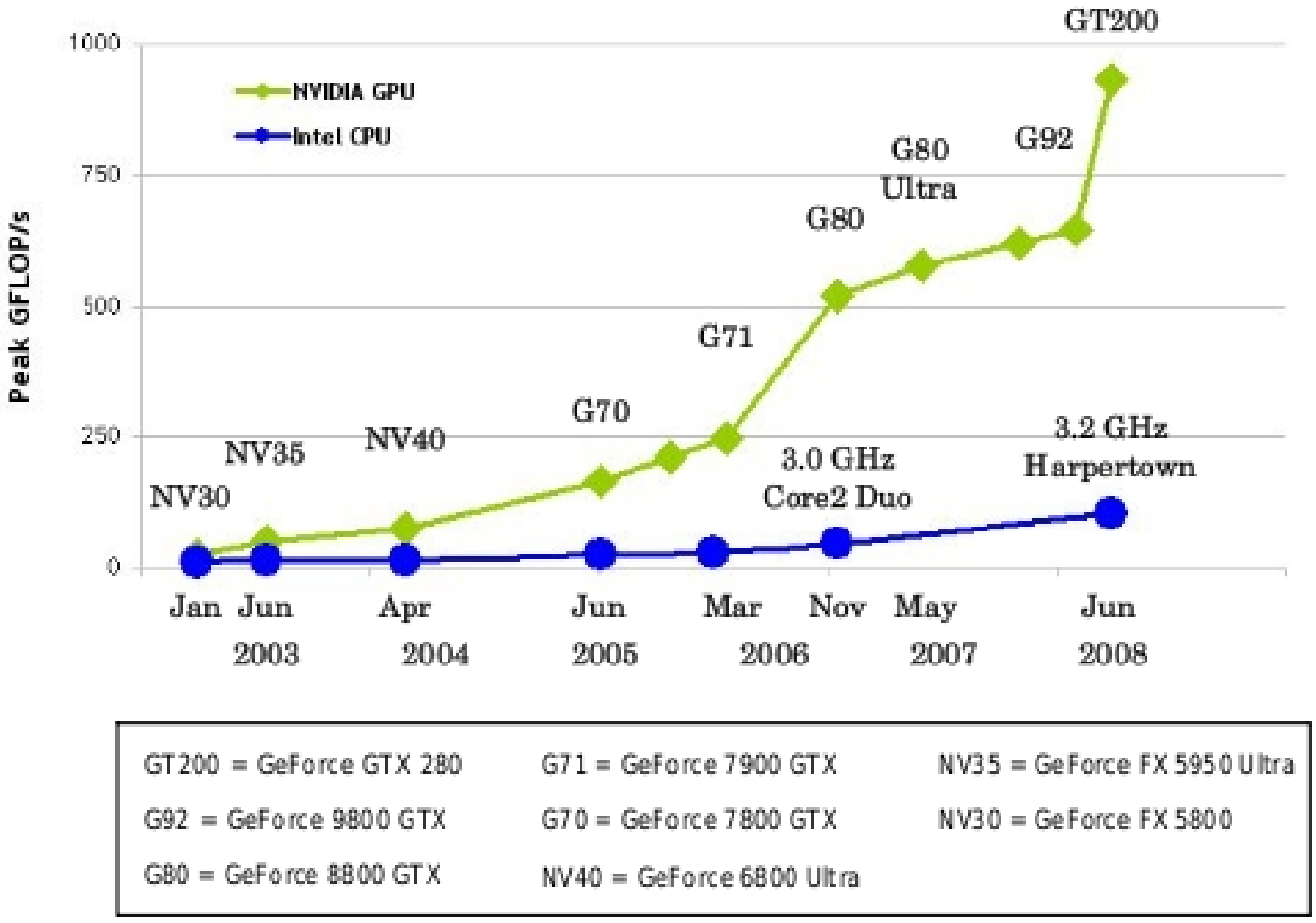
# Introduction à la programmation GPU

P. Fortin – UPMC / LIP6
ANR TaMaDi
27/10/2010

# Les processeurs graphiques (GPU)

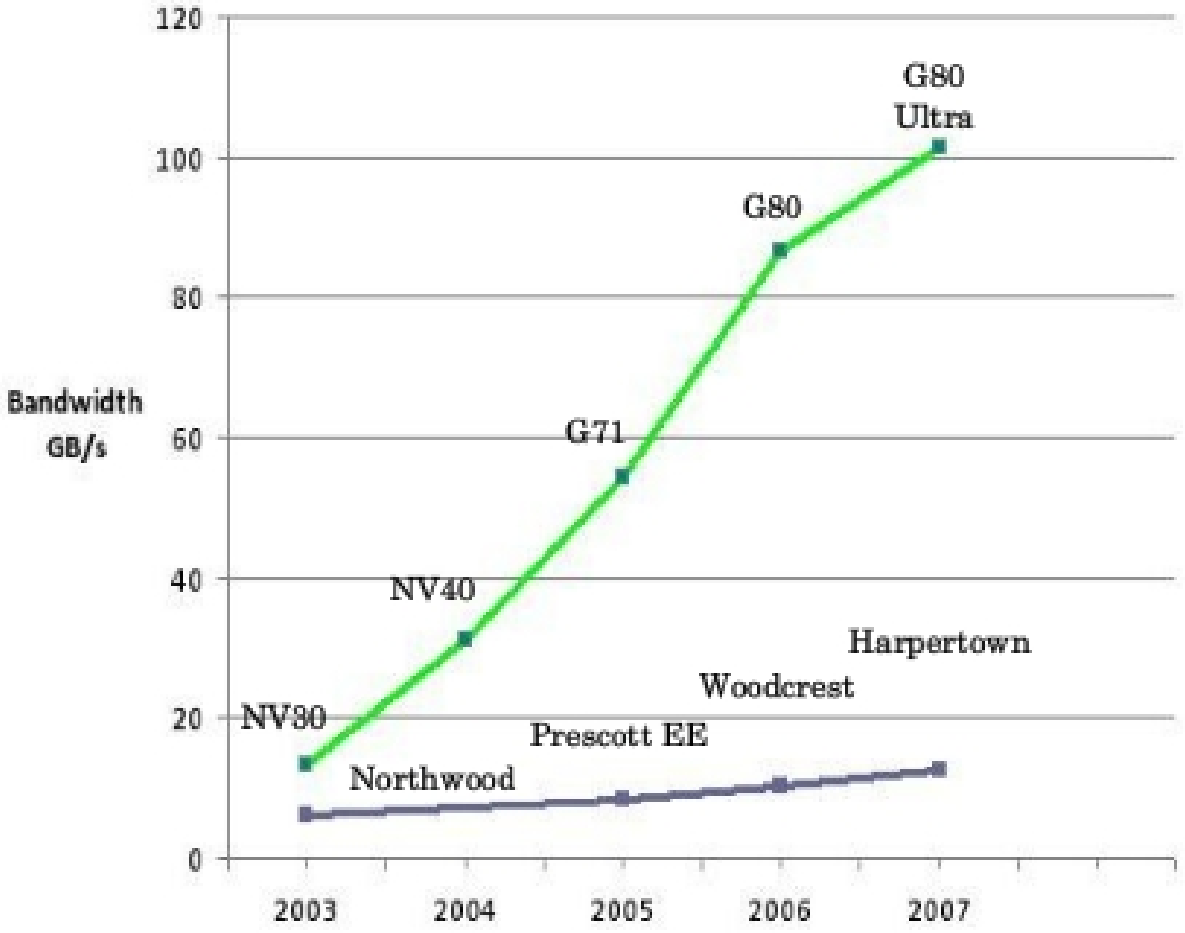
- GPU : *Graphics Processing Unit*
- GPGPU : *General-Purpose computation on Graphics Processing Unit* (ou *GPU Computing*)
- Architectures massivement parallèles conçues pour décharger le CPU des traitements graphiques
  - Introduction progressive des nombres flottants, de la simple et de la double précision, du respect de la norme IEEE et des fonctions mathématiques élémentaires (cos, sin, sqrt...)
  - Architecture « *manycore* »

# Comparaison CPU-GPU : performance crête




(CUDA Programming Guide 2.3)

# Comparaison GPU-GPU : bande passante mémoire




*(CUDA Programming Guide 2.3)*

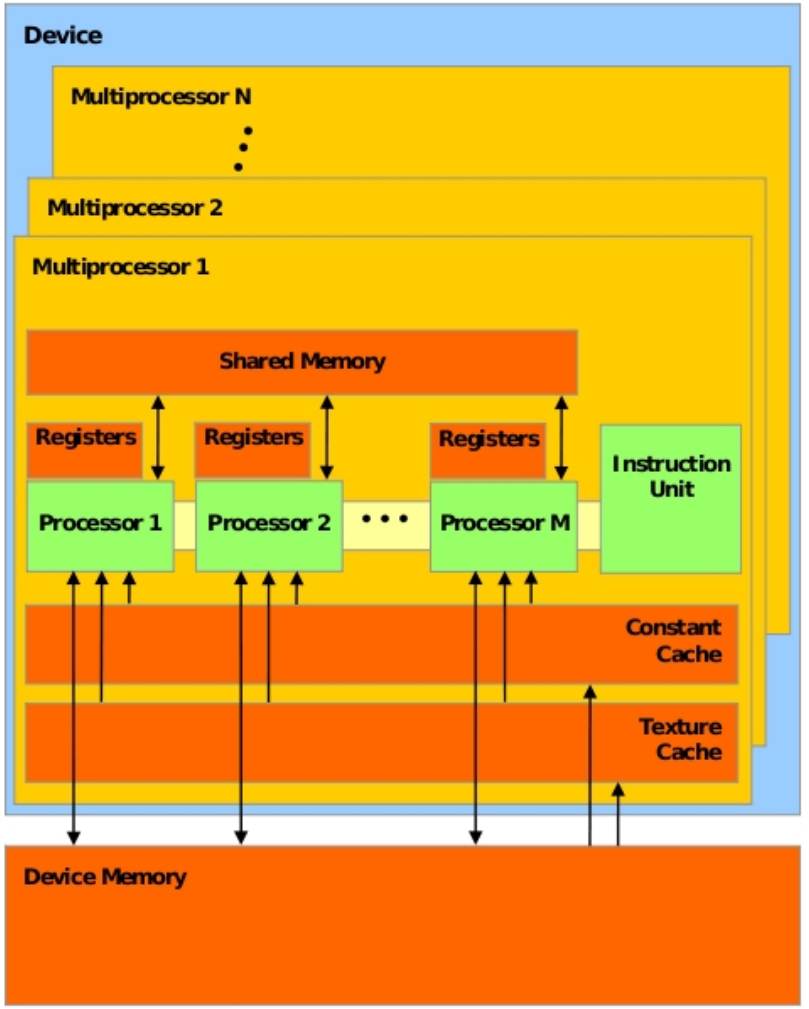
# Architecture des GPU

- Réduction de la place prise par les caches et par les unités de contrôle de l'exécution
- Augmentation des unités de calcul


*(CUDA Programming Guide 2.3)*

# Architecture des GPU (2)

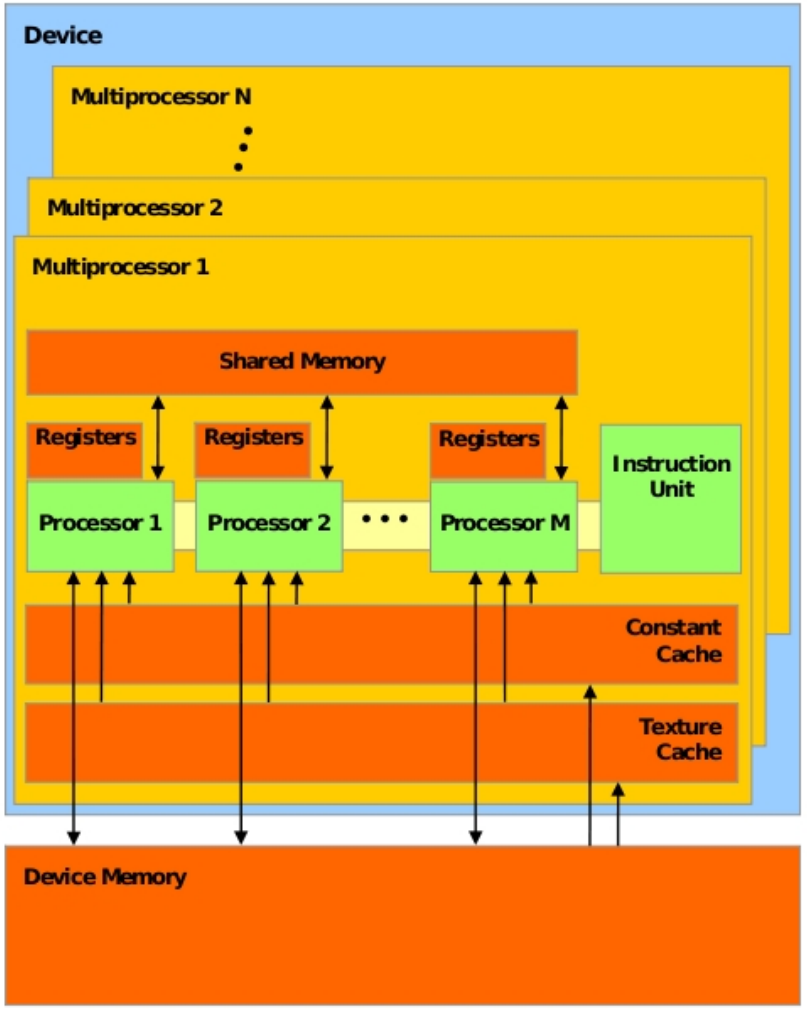
- 1 Multiprocesseur (MP) contient 8 *Scalar Processor* (SP) « coeurs »
- Exemple Tesla C1060 :
  - 30 Multiprocesseurs → 240 « coeurs » @ 1296 MHz
  - Ou : 30 coeurs 8-*way* SIMD
  - Performance crête
    - Simple précision : 933 Gflop/s
    - Double précision : 78 Gflop/s
  - Mémoire globale (*Device Mem*) : 4 Go
  - Bande passante mémoire : 102 Go/s
  - Bande passante CPU-GPU (bus PCI Express 2.0 16x) : 8 Go/s
  - Mémoire partagée : 16 Ko




*(CUDA Programming Guide 2.3)*

# Architecture des GPU (3)

- Exemple Fermi C2050 :
  - 14 Multiprocesseurs avec 32 SP chacun
    → 448 SP (« coeurs ») @ 1.15 GHz
  - Ou : 14 coeurs 32-*way* SIMD
  - Performance crête :
    - Simple précision : 1,03 Tflop/s
    - Double précision : 515 Gflop/s
  - Mémoire globale (*Device Mem*) : 3 Go
  - Bande passante mémoire : 144 Go/s
  - Cache L1 (configurable : de 16Ko à 48 Ko → complète mémoire partagée)
    + cache L2 (768 Ko)
  - Protection mémoire ECC

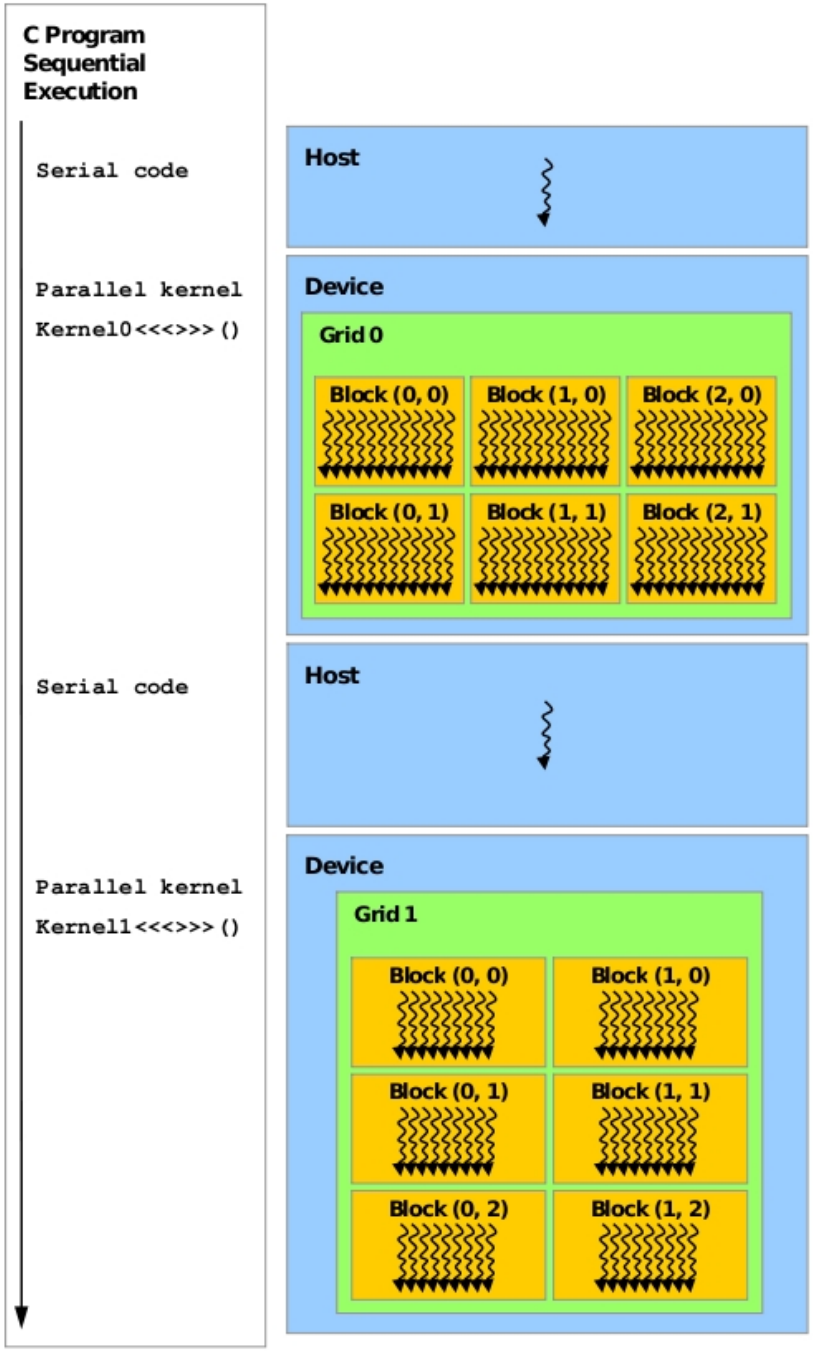



*(CUDA Programming Guide 2.3)*

# Programmation des GPU

NVIDIA : CUDA

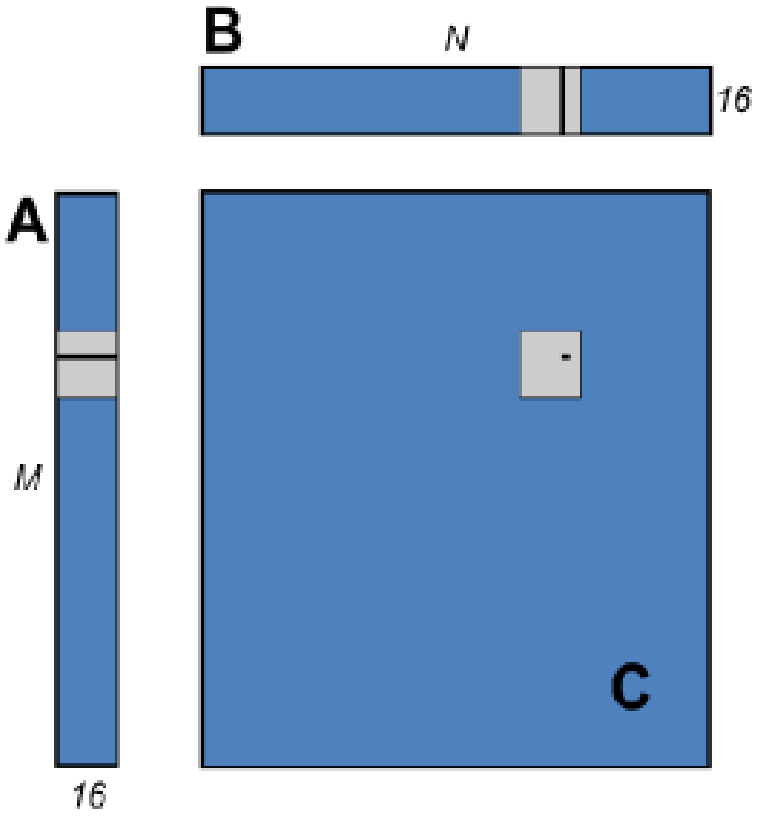
- Code C/C++ séquentiel sur CPU et code parallèle CUDA (C/C++ avec extensions CUDA) sur GPU
- Grille 2D de blocs 3D de threads
  - Chaque bloc de threads est placé sur 1 des multiprocesseurs
  - Les threads du blocs s'exécutent sur les SP du multiprocesseur
- Programmation « thread centrée »




*(CUDA Programming Guide 2.3)*

# Exemple : produit matriciel
## (d'après *CUDA 3.1 Best Practices Guide*)

```
__global__ void sharedABMultiply(float *a, float* b,
float *c, int N){
  __shared__ float aTile[16][16], bTile[16][16];
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  float sum = 0.0f;
  aTile[threadIdx.y][threadIdx.x]=
        a[row*16+threadIdx.x];
  bTile[threadIdx.y][threadIdx.x] =
        b[threadIdx.y*N+col];
  __syncthreads();
  for (int i = 0; i < 16; i++) {
    sum += aTile[threadIdx.y][i]* bTile[i][threadIdx.x];
  }
  c[row*N+col] = sum ;
}
```



# Programmation des GPU

- Architecture SIMT (Single-Instruction, Multiple-Thread)
  - Threads scalaires → masque l'exécution SIMD (mais celle-ci doit être prise en compte pour obtenir de meilleures performances)
  - Exécution synchrone par *warp* de 32 threads
    - Eviter tout branchement divergent entre les threads d'un même warp ! (→ *predication* sur Fermi)
    - Distribution déterministe des warps sur les blocs de threads
    - Nombre maximum de warps par multiprocesseur : 48 (Fermi), soit 48 x 32 (threads) x 14 (multiprocesseurs) = 21504 threads
  - Ordonnancement des threads sans surcoût (nombreux registres : 32768 par multiprocesseur sur C2050)
    → recouvrement des accès mémoires en « surchargeant » le multiprocesseur de threads
    → mais : taille de la mémoire partagée limitée, et attention au *register spilling*

# Programmation des GPU (2)

- Hiérarchie mémoire exposée :
  - mémoire partagée → *user-managed cache*
  - Et aussi : mémoire constante et mémoire de textures
- *Coalesced memory accesses* : regroupement de plusieurs accès mémoire (depuis la mémoire globale) en 1 seule opération
  - Sur Fermi C2050 : 1 ligne de cache = 128 octets
    → 1 *coalesced memory access* si les 32 threads d'1 warp accèdent à 32 floats alignés (*128-byte aligned*) en mémoire globale
- Opérations atomiques (ex : *atomicAdd()* )
- Sur Fermi : exécution concurrente de plusieurs kernels
  - Avec CUDA 3.1 : jusqu'à 16 kernels exécutés en même temps

# Tuning de code CUDA (d'après F. Bodin)

1) Déterminer le nombre de threads par bloc
   → degré de vectorisation
2) Etudier les « coalesced memory accesses »
3) Déterminer le nombre de blocs dans la grille
4) Prendre en compte la mémoire partagée
5) Minimiser le nombre de registres

# Bibliothèques optimisées

- CUBLAS (extension de l'interface BLAS pour spécifier indépendamment les transferts CPU ↔ GPU) :
  - Produit matriciel réel simple précision (SGEMM) :
    - sur C1060 = 374 Gflop/s max environ
    - sur C2050 (version de MAGMA) = 645 Gflop/s (63% de la performance crête)

      → limité par bande passant mémoire: *memory bound* (*computation bound* sur CPU)
- Et aussi pour LAPACK : MAGMA et CULA
- CUFFT

# Autres langages de programmation des GPU

- AMD :
  - Brook+ (haut niveau, stream computing), CAL, CTM (Close To The Metal, bas niveau)
  - GPU AMD : instructions vectorielles
  - AMD = AMD+ATI → AMD Fusion (2011) : CPU+GPU=APU (Accelerated Processing Unit)
- OpenGL (Open Graphics Library)→ OpenCL (Open Compute Library)
  - Portable sur tous les GPU (et aussi multicoeurs, Cell...)
  - Instructions vectorielles (GPU ATI, SSE...)
  - Actuellement supporté par Apple (Mac OS X v10.6), NVIDIA, AMD-ATI
  - CUDA conserve certaines spécificités (ex : *Page-Locked Host Memory* pour accélérer les transferts mémoire CPU-GPU)

# Points forts et points faibles

- Bande passante mémoire GPU meilleure que bande passante mémoire CPU
- Programmation relativement aisée (CUDA)
- Respect norme IEEE 754 (simple et double) sur les GPU les plus récents
- Exploitation directe des nouveaux GPU (mais attention aux optimisations spécifiques à un GPU...)

- Requiert un parallélisme de données massif : milliers de threads
  + 32-*way* SIMD : régularité des calculs entre threads
- Requiert des applications avec une forte intensité de calcul
- Optimisation avancée du code non triviale
- Faible bande passante du bus PCI (→ goulet d'étranglement)
- Pas de partage des ressources matérielles (mémoire partagée, registres...)