Introduction à la programmation des GPU

Architecture massivement *multithread*
CUDA

Pierre Kestener

CEA-Saclay, IRFU, SEDI, France
Service d'Electronique, Informatique et Détecteurs

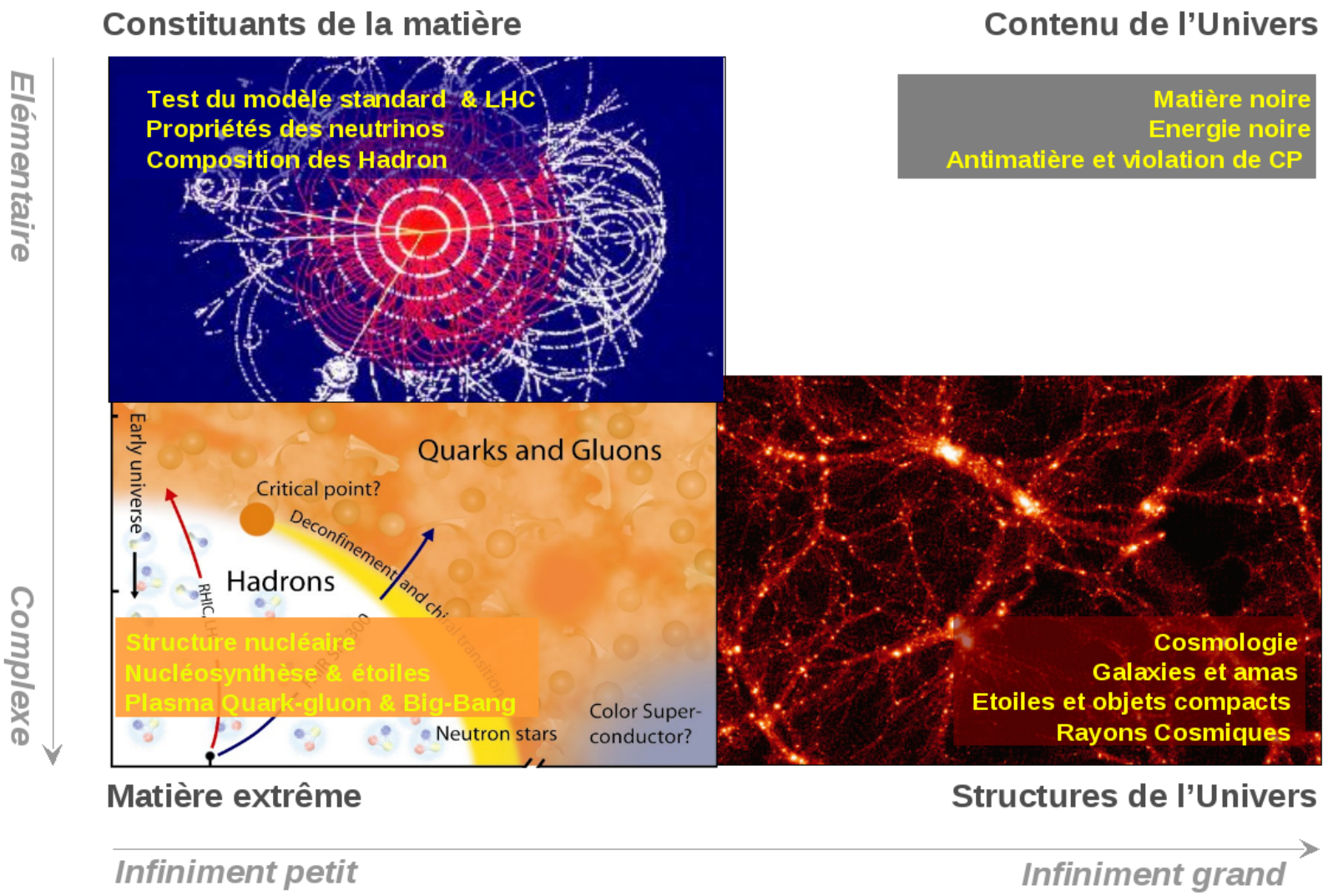
ENSTA, IN203 - Parallélisme - Février-Mars 2010

Au menu...

- Présentation générale
- Exemple en calcul scientifique : équations d'Euler
- Historique GPU
  - des processeurs vectoriels vers les GPU
  - migration des fonctionalités hardware/software
- Introduction au *GPU Computing*, calcul scientifique sur processeur graphique (NVIDIA)
  - architecture matérielle CUDA (architecture calcul / mémoire)
  - modèle de programmation/compilation/execution et language CUDA
  - outils de développement (compilateur, emulateur, debog, etc...)
  - *parallel programming patterns*
- Bibliographie web
  - liens vers des resources en ligne (cours, codes sources, etc...)

Sommaire

IRFU (Institut de recherche sur les lois fondamentales de l'Univers)



COAST (COmputational ASTrophysics at Saclay)

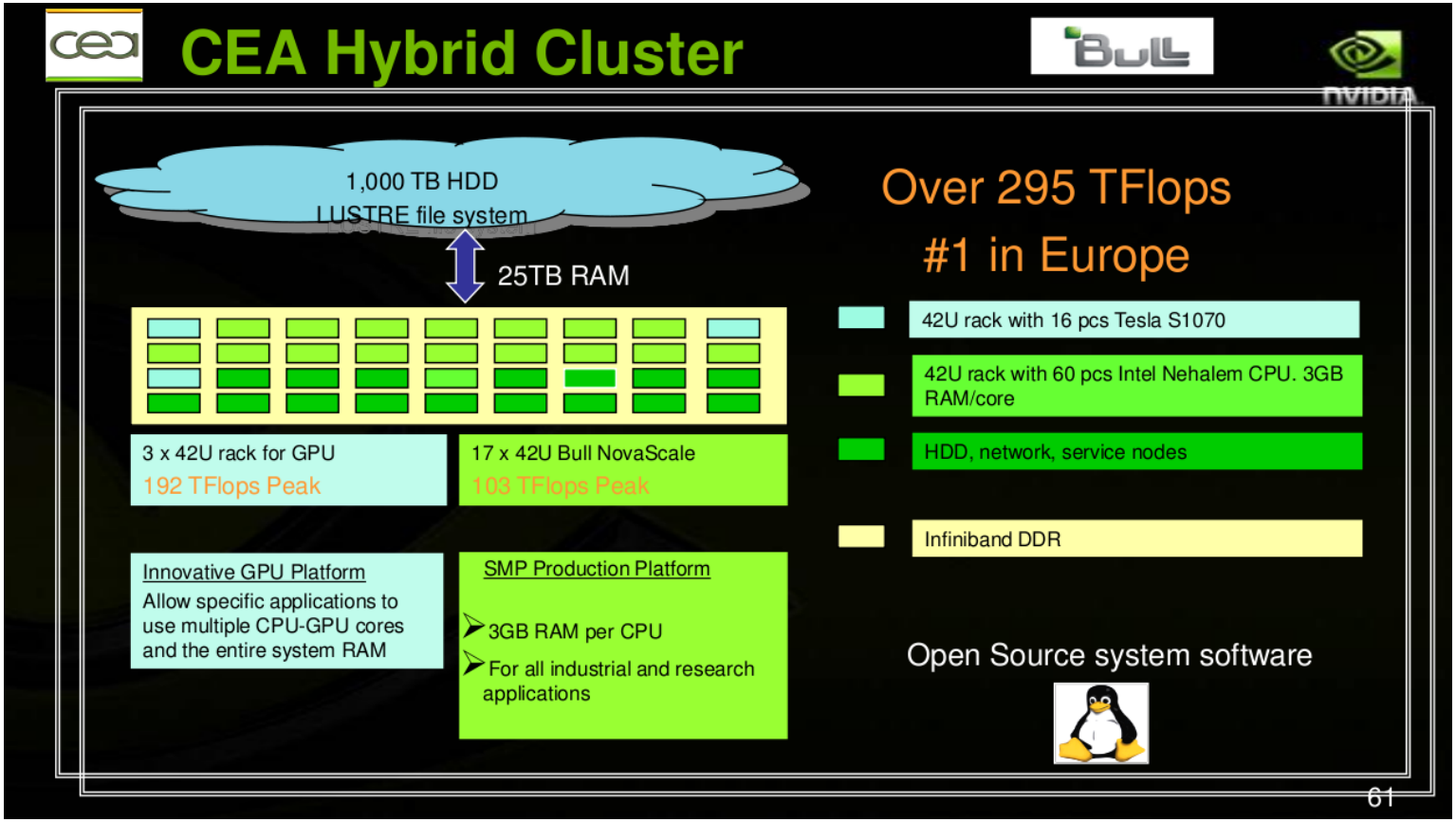
Projet de simulations numériques massivement parallèles pour l'astrophysique (resp. scientifique et technique : R. Teyssier et B. Thooris)
http://irfu.cea.fr/Projets/COAST/

- développement de **méthodes numériques**
- **implantation logicielle, optimisation et parallélisation**
- développement d'**outils de visualisation** et **bases de données**

Moyens de calcul :

- locaux : cluster 256 processeurs Opteron, mémoire de 8GBytes/coeur
- nationaux : PLATINE, CEA/CCRT, 10000 coeurs Itanimum2@1.6GHz, 50 TFlops
- nationaux : BABEL, CNRS/IDRIS, BlueGene, 40000 coeurs PPC, 110 TFlops
- 2009 : CEA, GENCI-CINES, **combinaison CPU/GPU**, 10000 Xeon-quad et 48 boites Tesla S1070 (4 GPU) : 128+200 TFlops

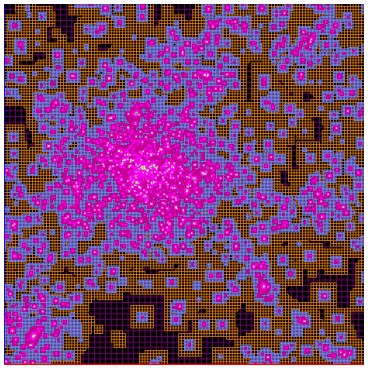
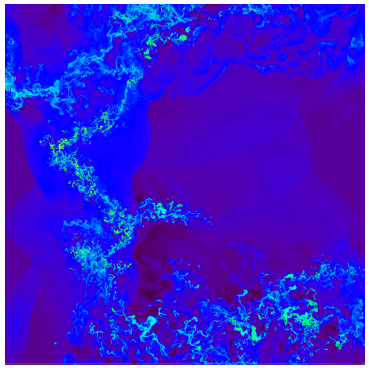
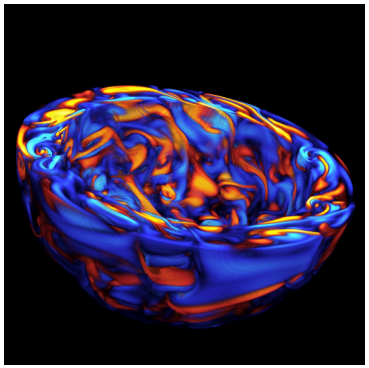
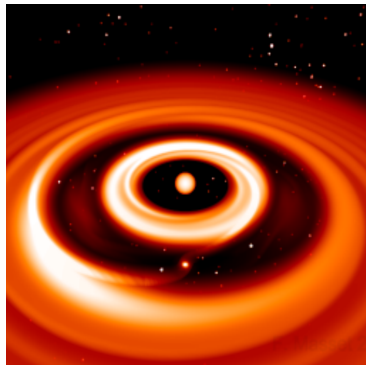
CEA Cluster hybride CPU/GPU, début 2009



http://www-ccrt.cea.fr/fr/moyen_de_calcul/titane.htm

COAST (COmputational ASTrophysics at Saclay)

Quelques codes développés à Saclay :

- RAMSES, *R. Teyssier et al.* : simulation de fluides autogravitants (hydrodynamique3D+MHD+N-corps, raffinement de grille adaptatif) pour l'étude des structures aux grandes échelles et la formation des galaxies, **record de simulation N-corps en 2007 (70 milliards de particules)**
- HERACLES, *E. Audit et al.* : étude du milieu interstellaire (hydrodynamique et transfert radiatif)
- ASH, *A. S. Brun et al.* : code de magnétohydrodynamique en géométrie sphérique (intérieur du Soleil)
- JUPITER, *F. Massey et al.* : étude des disques protoplanétaires et forces de marées disque/planète (hydrodynamique 3D, grille adaptative)

Grand challenge GPU / GENCI

http://www-ccrt.cea.fr/fr/le_ccrt/pdf/Programme_JS_CCRT_0909.pdf

Accélération GPU du transfert radiatif en cosmologie, par R. Teyssier et Dominique Aubert.

Example d'application : équations d'Euler 2D

- Formulation conservative des équations d'Euler 2D :

$$\mathbf{U}_t + \mathbf{F}(\mathbf{U})_x + \mathbf{G}(\mathbf{U})_y = 0$$

- Variables conservatives et termes de flux:

$$U = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ E \end{bmatrix}, F = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ u(E+p) \end{bmatrix}, G = \begin{bmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ v(E+p) \end{bmatrix}$$

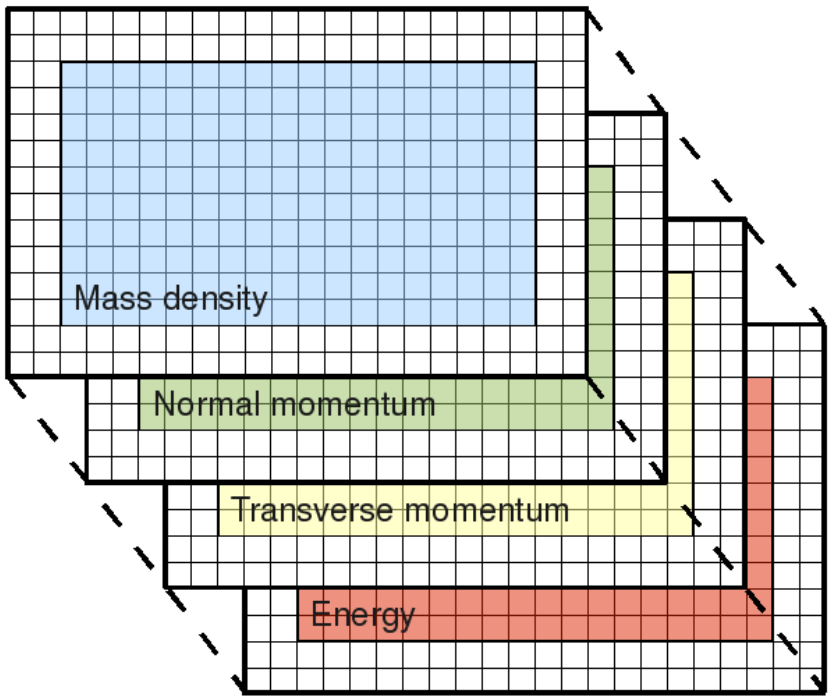
- $E = \rho(\frac{1}{2}\mathbf{V}^2 + e)$ Energie volumique totale
- Equation d'état des gaz parfaits pour l'énergie interne : $e = \frac{p}{\rho(\gamma-1)}$, avec $\gamma = 1.4$ (air à $T = 20^o C$)
- passage aux variables non-conservatives: $U \Rightarrow W$ avec ${}^T W = [\rho, u, v, p]$

Schémas numériques

- Méthode de Godunov du 1er ordre :
  $\mathbf{U}_i^{n+1} = \mathbf{U}_i^n + \frac{\Delta t}{\Delta x}(\mathbf{F}_{i-\frac{1}{2}} - \mathbf{F}_{i+\frac{1}{2}})$
- implantation du schéma MUSCL-Hancock (Monotone Upstream-centered Schemes for Conservation Laws)
- schéma Godunov 2ème ordre où les $\mathbf{U}_i^n$ sont remplacés par des fonctions linéaires par morceaux.
- MUSCL-Hancock en 3 étapes :
  - calcul des pentes et des valeurs de $\mathbf{U_i}$ aux bords de la cellule
  - évolution d'un 1/2 pas de temps
  - résolution du problème de Riemann pour avoir les nouveaux flux $\mathbf{F_{i+\frac{1}{2}}}$ et mise à jour des $\mathbf{U_i}$
- résolution 2D par séparation des directions

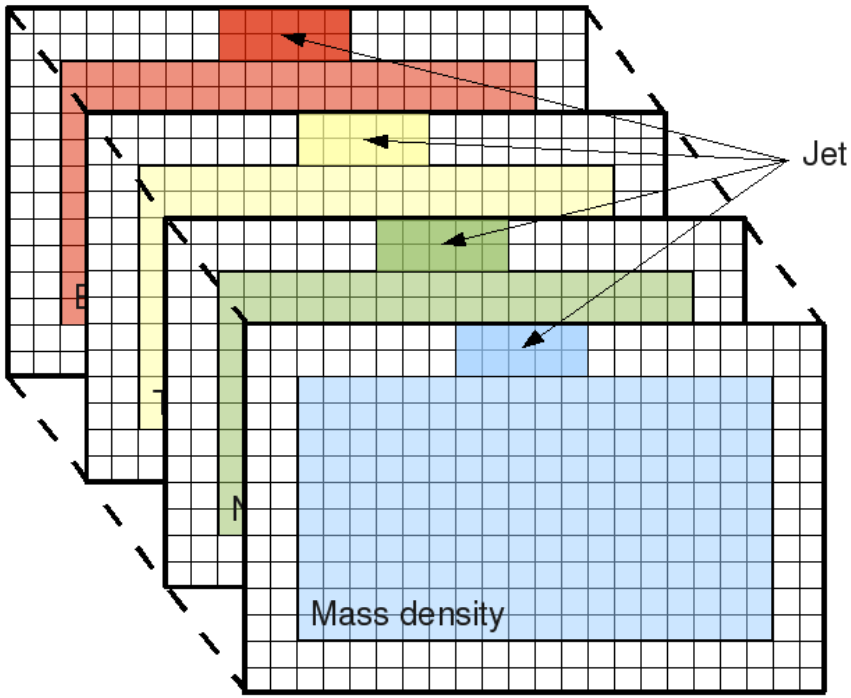
Structures de données

- 4 grilles 2D:
  - 1 par variable conservative
  - discrétisation de l'espace 2D ou évolue le fluide
  - conditions limites: bordure de 2 éléments, plusieurs types:
    - fermée
    - ouverte sur l'infini
    - ouverte sur la bordure opposée



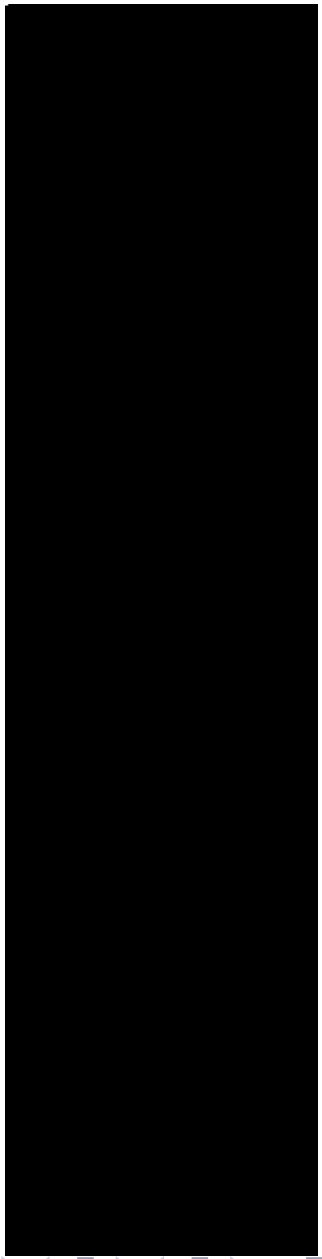
Simulation

- simulation d'un jet fluide injecté dans le domaine
- paramètres de la simulation
  - paramètres du run (temps simulation, fréquence output)
  - paramètres géométriques (largeur, longueur, Δx)
  - types de bordures
  - paramètres du schéma
  - paramètres du jet

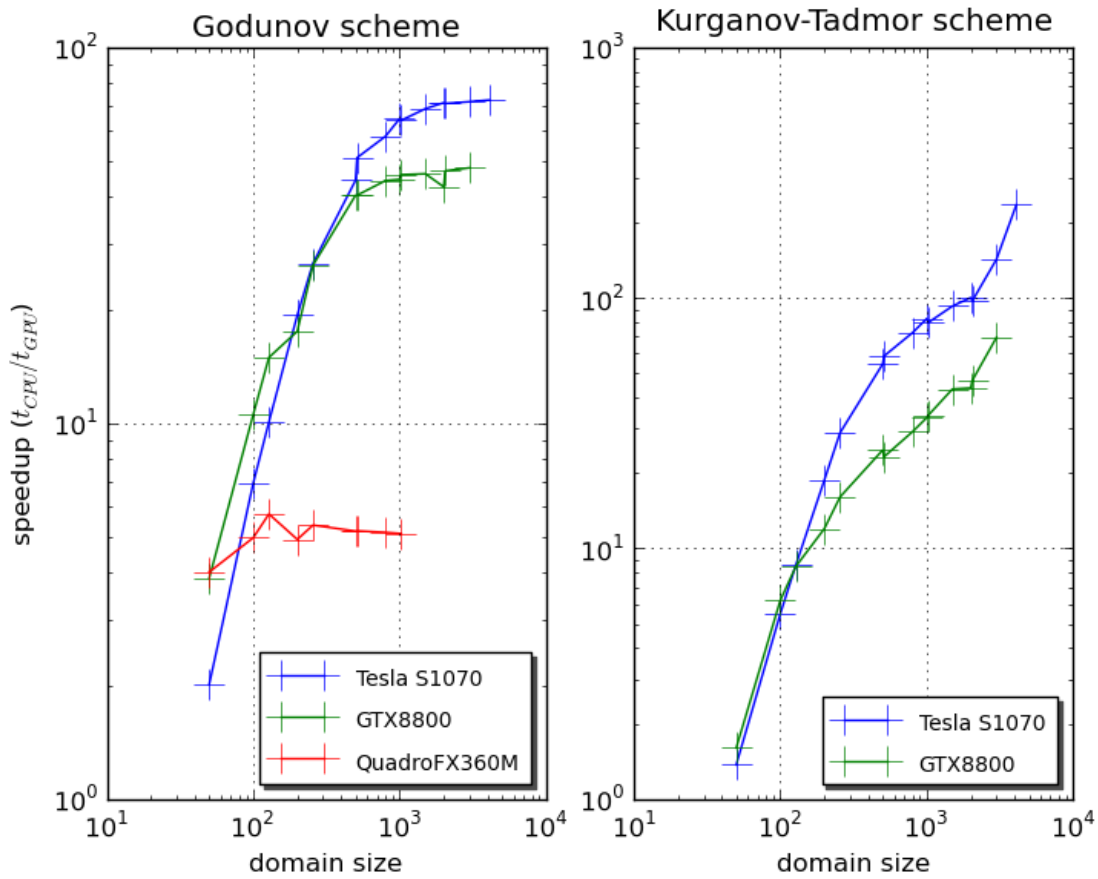


Exemple de simulation

- run: $t_{end} = 1s$, 1 output tous les 50 pas de temps
- géométrie: grille rectangulaire 200×800, $\Delta x = 0.05m$
- bordures: fermées des 4 cotés
- schéma: 10 itérations pour Riemann, facteur de courant 0.8
- jet: central, largeur 10 éléments, $v = 300m.s^{-1}$
- conditions initiales : ${}^T W = [\rho = 1, u = 0, v = 0, p = 1]$ à l'intérieur et ${}^T W = 0$ sur les bords



Performance



Référence : P. Kestener *et al.*, HPCTA 2010, Busan.

Sommaire

Des processeurs vectoriels aux GPU

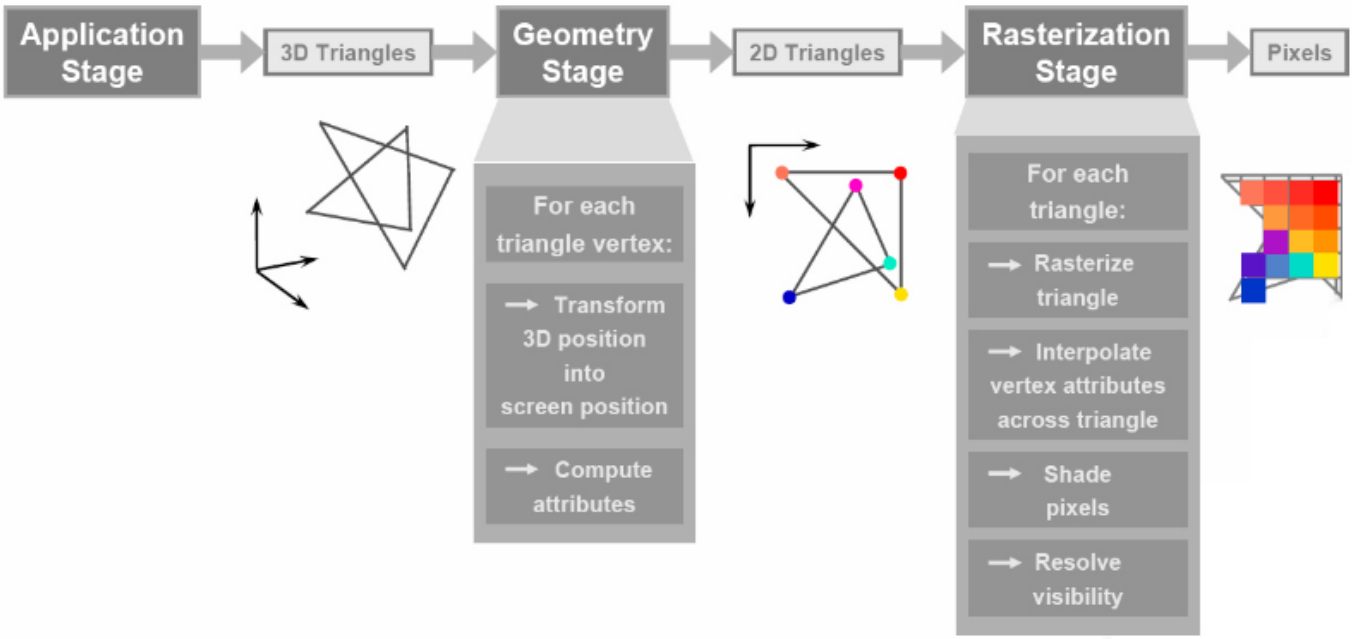
http://www.irisa.fr/archi09/defour.pdf

- Processeur vectoriel : exécuter la même instruction sur chacune des données d'un tableau dans un même coup d'horloge
- la plupart des super-calculateurs des années 80-90 étaient vectoriels
- CRAY-1: 1976, 80MHz, 64-bit/data, 24-bit/addresse, vecteur de registres, 160 MIPS, 250 MFLOPS, 8MB RAM, 5.5 tonnes, ~ 200-kW (refroidissement compris)
- les processeurs scalaires actuels possèdent des jeux d'instructions vectorielles (SIMD) : SSE, Altivec qui utilisent des registres dédiés du processeur (128 bits).
- chute des processeurs vectoriels (fin 80 - début 90) : utilisation techno CMOS; marché grand public du micro; microprocesseur monopuce / processeur vectoriel multi-puce; augmentation des tailles de cache sur les processeurs scalaires; complexité de la programmation des proc vectoriels.

Evolution des GPU : architecture et programmation

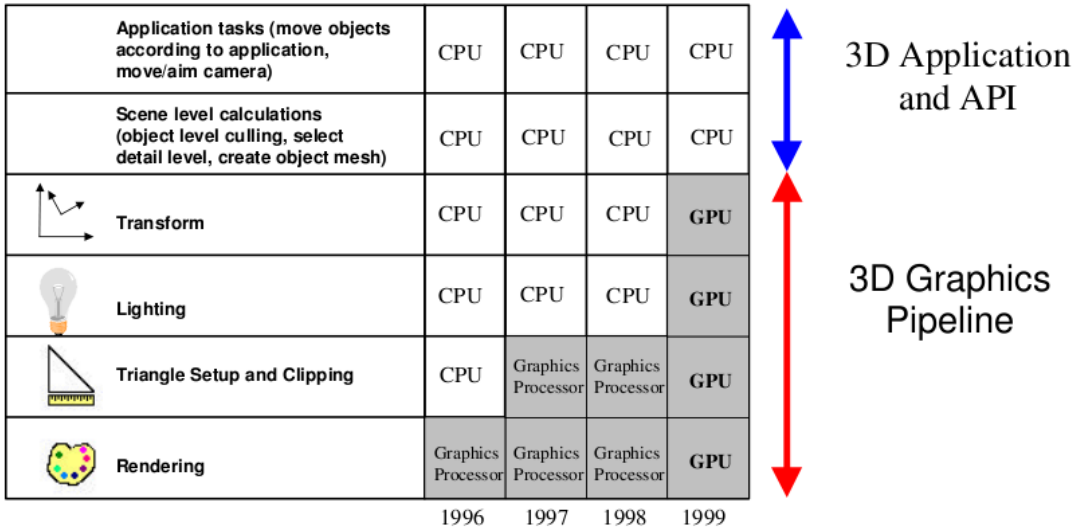
- notion de *pipeline graphique*
- notion de *shaders* (fonctionalité programmable dans le *pipeline* graphique)
- aperçu des architectures de GPU
- apparition du GPGPU pré-CUDA (avant 2004)

Aperçu des architectures de GPU : 1995 à 1999



- pipeline graphique : schéma fonctionnel
- fonction initiale : **décharger le CPU de toutes les tâches graphiques**
- matériel dédié pour des tâches spécialisées
- *pipeline graphique* 2D puis 3D
- à partir de 1995 : rendu 3D, fonction de rasterisation (image vectorielle vers image matricielle) et rendu plus complexe

Mini historique GPU : 1995 à 1999

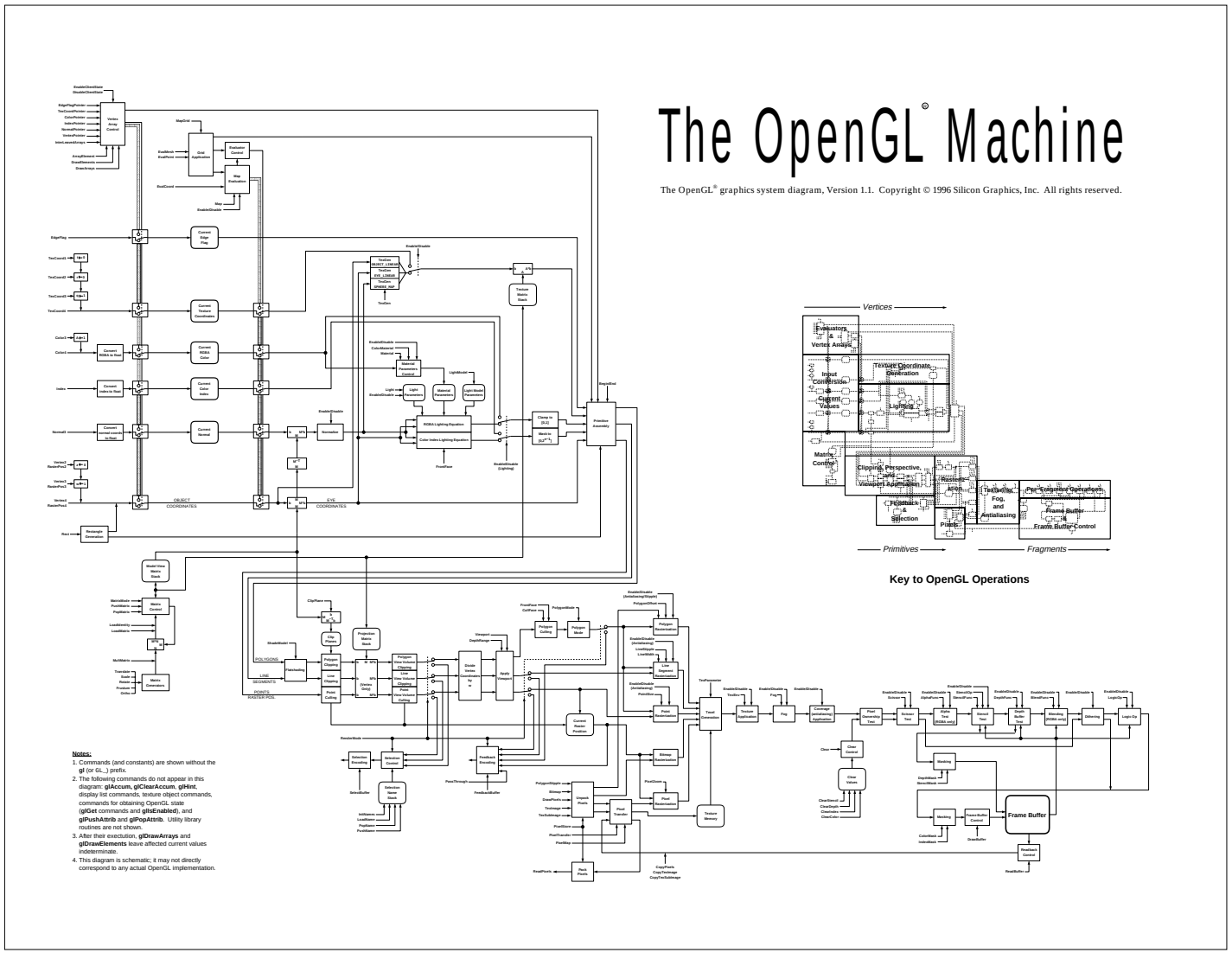


http://www.nvidia.com/object/Technical_Brief_TandL.html

- spécification de standards d'interface de programmation (API, modèle de programmation trans-vendeur, abstraction du matériel) pour les *pipelines graphiques* : OpenGL, DirectX
- essor du 3D grand public : *Toy Story I / II* chez Pixar (RenderMan, rendu *off-line*, lancer de rayon), consoles de jeux
- 1999 : transfert vers les GPU des opérations de transformation (multiplication matrice 4x4) et éclairage

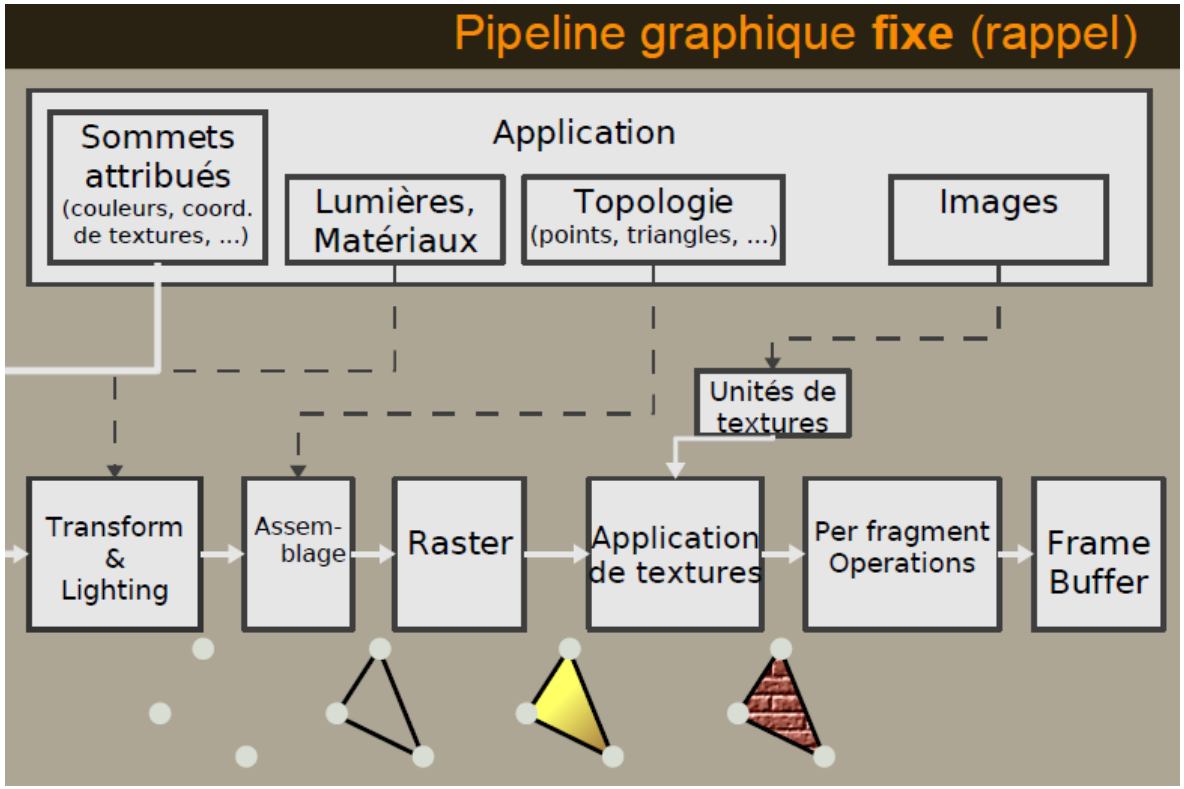
OpenGL StateMachine

http://www.opengl.org/documentation/specs/version1.1/state.pdf



Mini historique GPU : 2000 et après

- 2001 : Nvidia GeForce3
- **pipeline graphique programmable** : les *pixel* et *vertex-shaders* écrit en **langage bas-niveau**, avantage : plus grande flexibilité et permettre au matériel de suivre les évolutions rapides des standards, programmation plus facile et plus rapide.

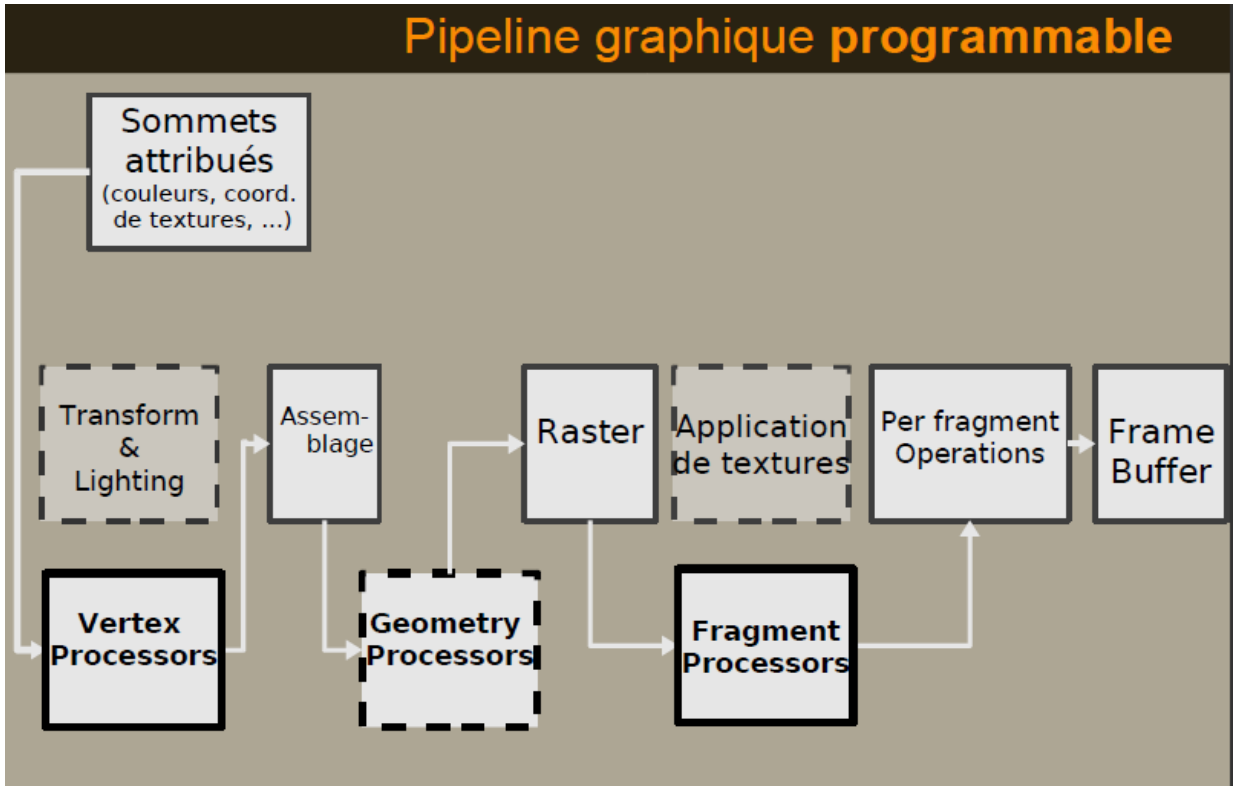


1

[1] www.irit.fr/~Vincent.Forest/teaching/SeminaireGPU.pdf

Mini historique GPU : 2000 et après

- 2001 : Nvidia GeForce3
- **pipeline graphique programmable** : les *pixel* et *vertex-shaders* écrit en **langage bas-niveau**, avantage : plus grande flexibilité et permettre au matériel de suivre les évolutions rapides des standards, programmation plus facile et plus rapide.



[1]

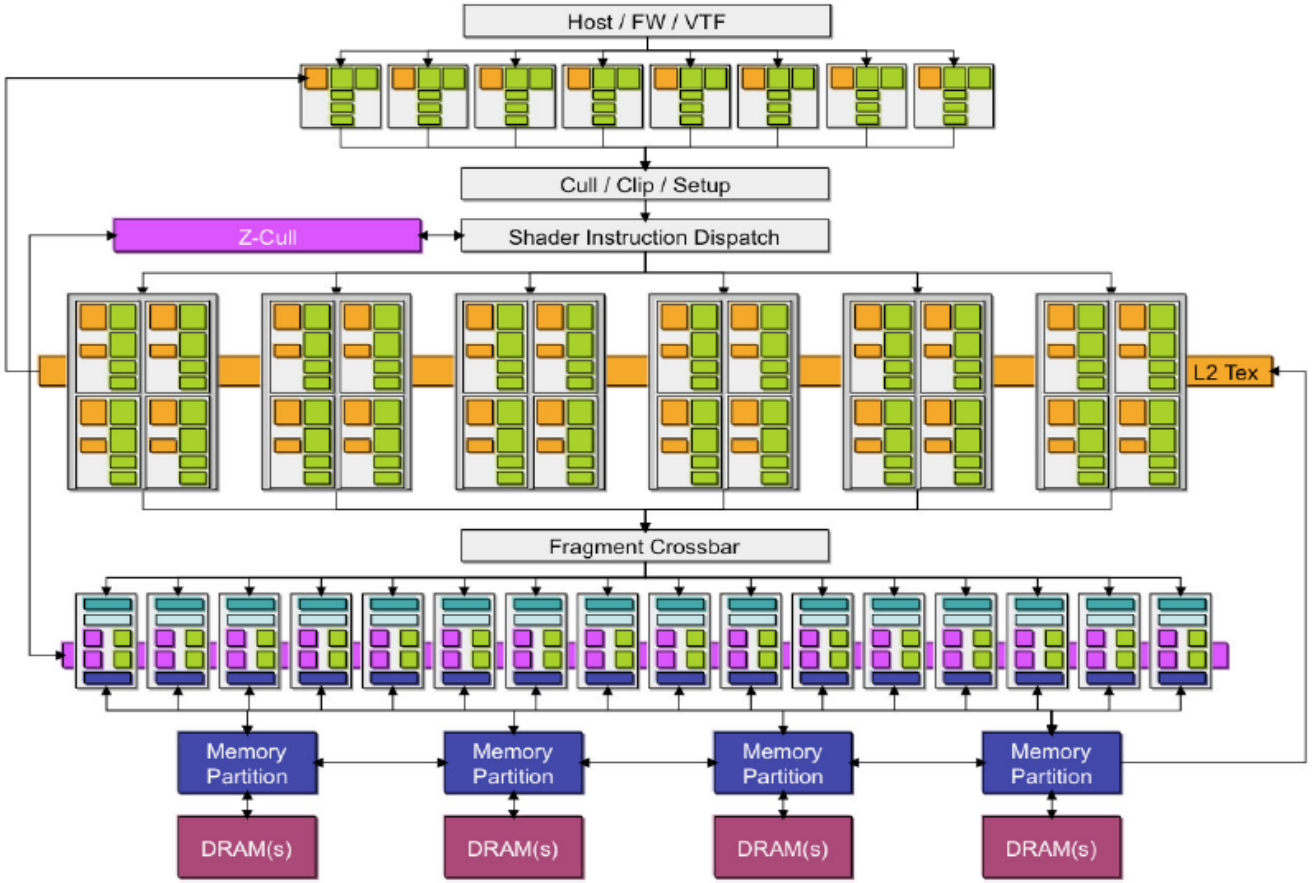
[1] www.irit.fr/~Vincent.Forest/teaching/SeminaireGPU.pdf

Mini historique GPU : 2000 et après

- 2001 : Nvidia GeForce3
- **pipeline graphique programmable** : les *pixel* et *vertex-shaders* écrit en **langage bas-niveau**, avantage : plus grande flexibilité et permettre au matériel de suivre les évolutions rapides des standards, programmation plus facile et plus rapide.
- GPU : *Vertex Processors* (MIMD), *Fragment Processors* (SIMD), float 32bits
- introduction de **langages haut-niveau** : Cg (Nvidia, compatible OpenGL/DirectX), HLSL (Microsoft, compatible API DirectX uniquement)
- abstraction du matériel, le programmeur n'a besoin de connaître que "peu" d'information sur le matériel

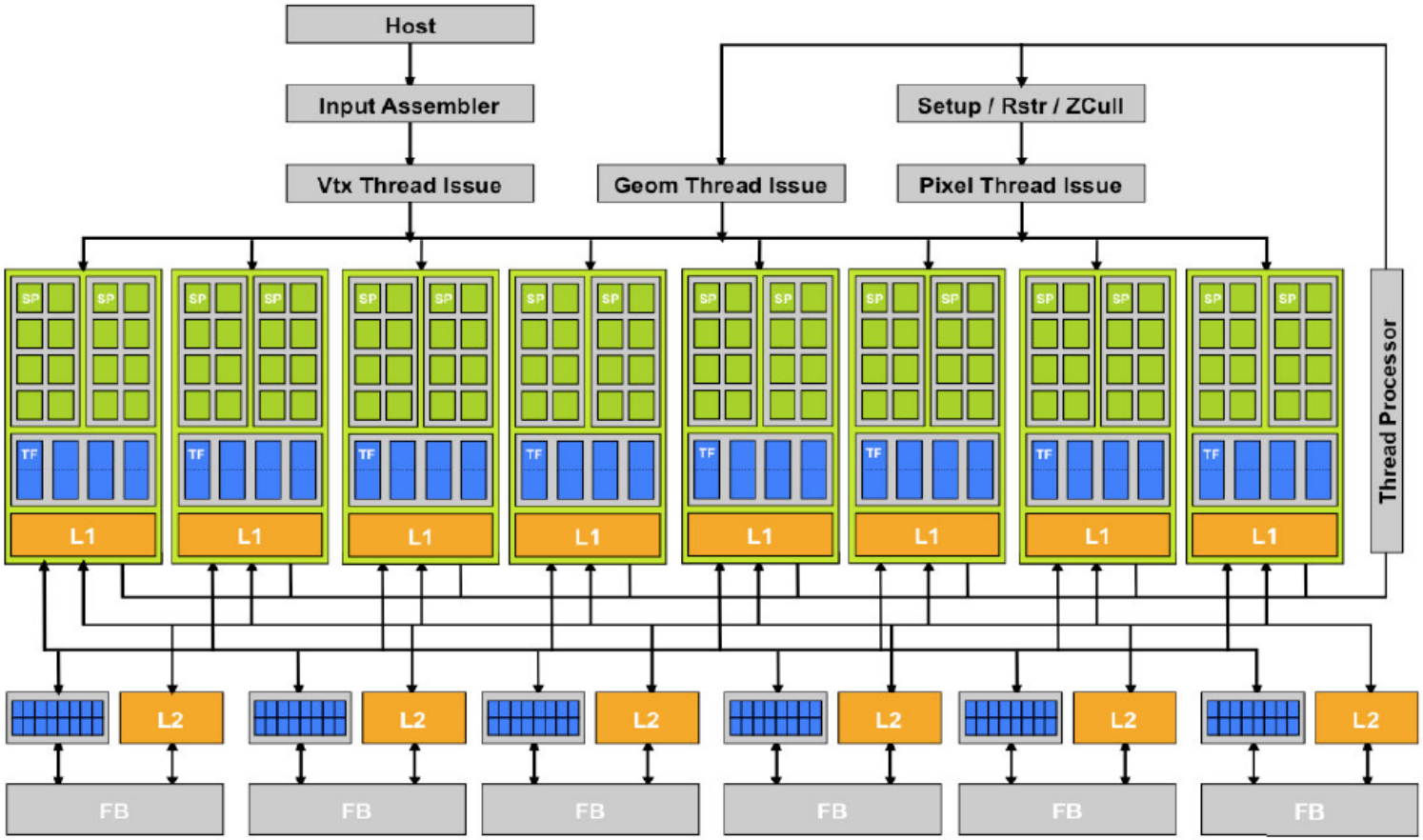
Transition GPGPU - CUDA - architecture unifiée - 2007

Nvidia GeForce 7800



Transition GPGPU - CUDA - architecture unifiée - 2007

Nvidia GeForce 8000



GPU computing - avant CUDA

GPGPU : General Purpose computions on GPU

- on détourne les concepts graphiques des languages Cg ou HLSL pour faire du calcul
- Tutorial OpenGL pour le GPGPU (2005)
  http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html
  voir aussi le chapitre 31 de GPUGems2, *Mapping Computational concepts to GPUs*
- GPGPU concept 1 : arrays = texture
  - frame buffer / off-screen rendering (récupérer les résultats de calcul et éviter les seuillages)
  - mémoire : tableau → GL_TEXTURE_2D / (read-only or write-only), attacher une texture à un FBO
  - malloc → 10 lignes autour de *glTexImage2D*
  - indexes (entier) → texture coordinates (float $\in [0,1]$)
  - data type : float → GL_LUMINANCE or GL_RGBA
  - taille de tableau : puissance de 2 (?)
  - maîtrise du passage texel vers pixel
- code exemple : addition de 2 tableaux saxpy_cg.cpp

GPU computing - avant CUDA

GPGPU : General Purpose computions on GPU

- on détourne les concepts graphiques des languages Cg ou HLSL pour faire du calcul
- Tutorial OpenGL pour le GPGPU (2005)
  http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html
  voir aussi le chapitre 31 de GPUGems2, *Mapping Computational concepts to GPUs*
- GPGPU concept 2 : kernel programs = shaders
- GPGPU concept 3 : computing = drawing
- code exemple : addition de 2 tableaux saxpy_cg.cpp

GPU computing - CUDA

- Nvidia Geforce8800, 2006, vers une **architecture unifiée** (on fusionne les différents type de *shaders processors*)
- CUDA : **Compute Unified Device Architecture**
- CUDA SDK (compilateur+librairies) pour le GPGPU, language *C-like*

Applications à l'étude :

- météo : http://www.mmm.ucar.edu/wrf/WG2/GPU/
- CFD : http://www.nvidia.com/object/computational_fluid_dynamics.html
- chimie quantique : http://pubs.acs.org/doi/abs/10.1021/jp0776762
- dynamique moléculaire (problèmes à N corps): http://www.ks.uiuc.edu/Research/gpu/
- analyse finançière : http://www.oneye.com.au/downloads/OnEye-Options-v1.pdf
- traitement du signal et de l'image : GPU-CV https://picoforge.int-evry.fr/cgi-bin/twiki/view/Gpucv/Web/
- géophysique

et l'embarqué...

- OpenGL ES (Open Graphics Library for Embedded System)
  - OpenGL ES 1.1.X (2004-2008): *fixed function hardware*, choisi comme API 3D dans Symbian , Android, iPhone SDK, ...
  - OpenGL ES 2.0.X (2007-...): *programmable hardware*, spécification d'un *shading language* haut-niveau, utilisé par les nouveaux iPod Touch, Nokia N900,

- nouveaux *systèmes sur puce* (CPU+GPU sur une seule puce), marché des tablet-PC, *entertainment center*, ...
  http://tegradeveloper.nvidia.com/tegra/

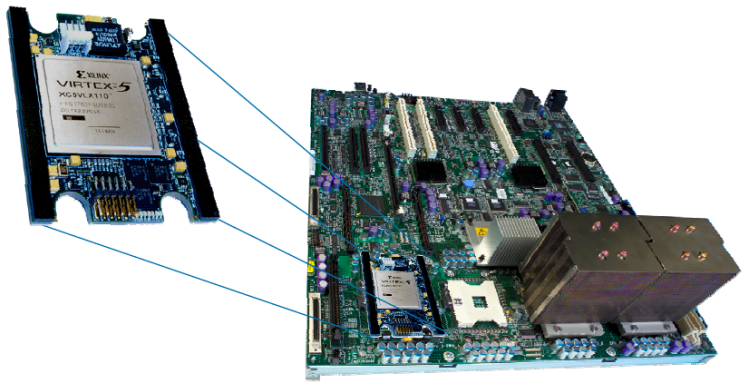
FPGA : reconfigurable computing

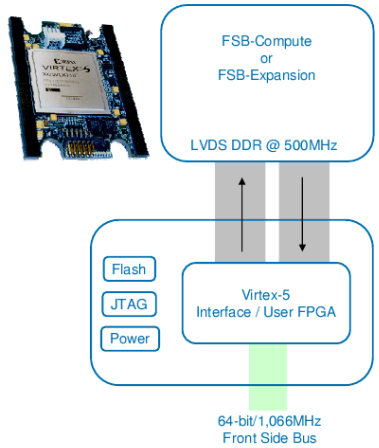
- **FPGA** : circuits intégrés *génériques* (matrices de blocs logiques reprogrammables, routage labile; inventés vers 1985 pour l'embarqué
- programmation en HDL (*Hardware description language*) très bas niveau, on décrit la fonctionnalité au niveau cycle d'horloge
- apparition de bibliothèque d'IP (*Intellectual Properties*)
- de plus en plus complexe : intégration de bloc de communication (Ethernet, Gigabit Tranceiver, etc) et de calcul (processeur PPC, bloc DSP)
- *reconfigurable computing* : utilisation des FPGA pour le calcul scientifique (dynamique moléculaire, bio-informatique, ...)
- introduction de languages haut-niveau (dérivé du C) pour programmer les FPGA et intégration dans des systèmes standards (PC, super-calculateurs, ...)
- outils haut-niveau qui masque/abstrait le flot de conception bas-niveau (synthèse logique vers RTL, placement routage, ...) pour ce concentrer sur les algorithmes scientifiques

FPGA : reconfigurable computing / accelerateur

voir http://www.nallatech.com/

- Intel S7000FC4UR
  http://www.intel.com/design/servers/platforms/S7000FC4UR/index.htm





FPGA : reconfigurable computing

Quelques liens :

- référence biblio : Reconfigurable Computing, *Accelerating Computation with FPGA*, by M. Gokhale and P.S. Graham, Springer, 2005.
- wikipedia, article Reconfigurable computing
- workshop HPRCTA09

- En perte d'intérêt **apparente** depuis le *buzz* `GPU Computing` et le regain d'intéret des architecture multi-cœurs (possible convergence des outils CPU/GPU dans un futur proche...)

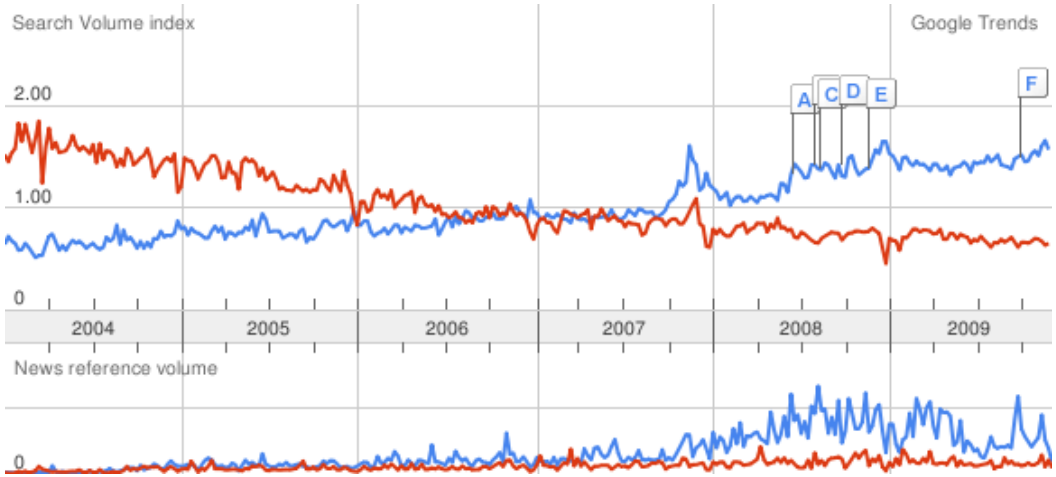
FPGA : reconfigurable computing

Applications :

- applications de type calcul : SDR (*Software Defined Radio*, télécommunication, radio-astrophysique, militaire)
- applications de type réseau haut-débit/stockage : faible latence (finance, *trading*), compression, ...
- acteurs inductriels : Nallatech, DRC Computer, XtremeData
- outils haut-niveau commerciaux (*C-to-RTL*) : Mitrion-C, ImpulseC, ...

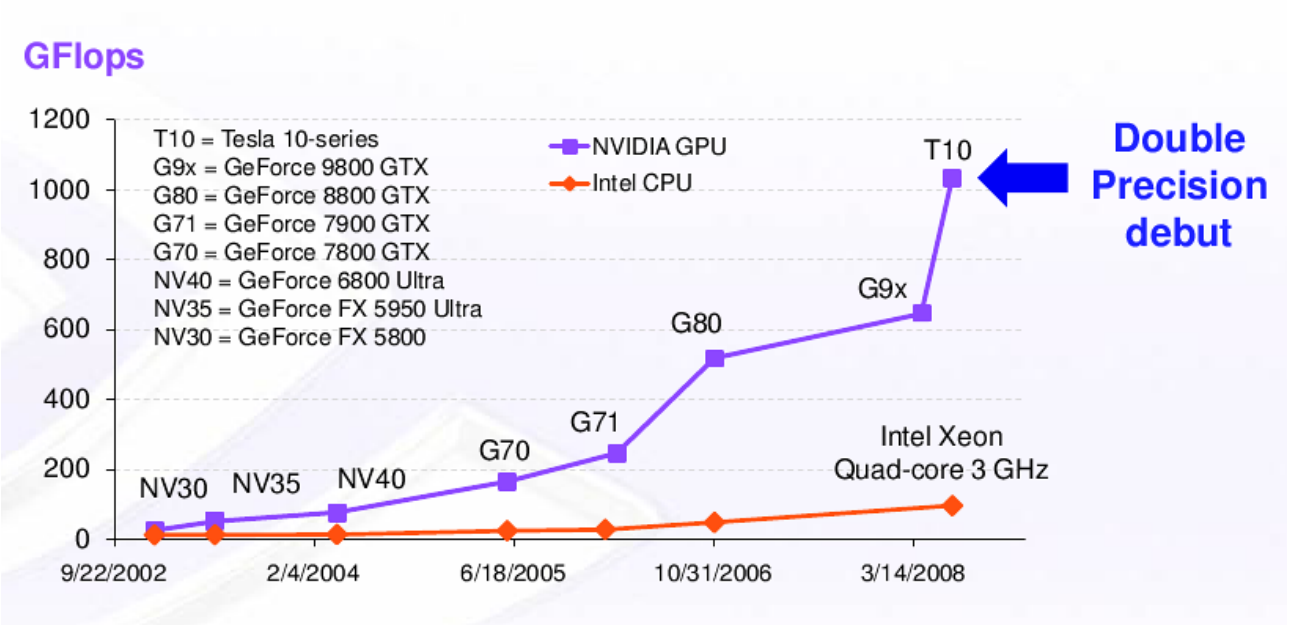
les problèmes :

- les FPGA sont chers (≠ GPU, marché du jeux vidéo)
- la complexité des outils FPGA pour des programmeurs
- support des fabricants de puces pour le HPC (≠ NVIDIA)



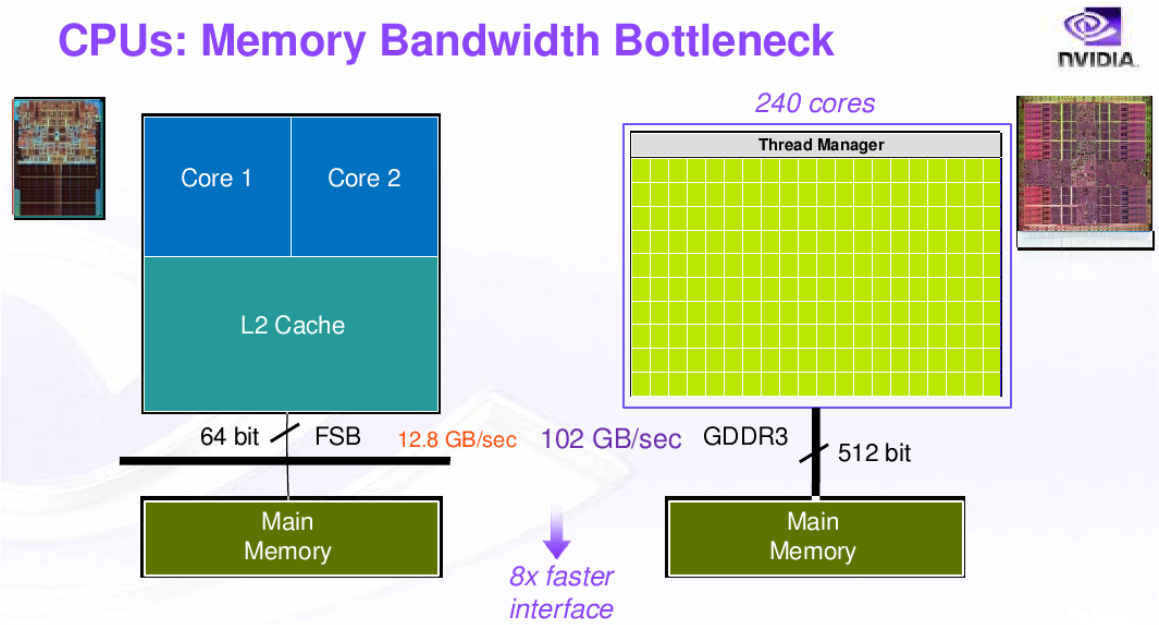
Pourquoi utiliser les GPU ?

- performance brute (GFLOPS); faible coût du GFLOPS; architecture massivement *multi-thread* (**matériel**)



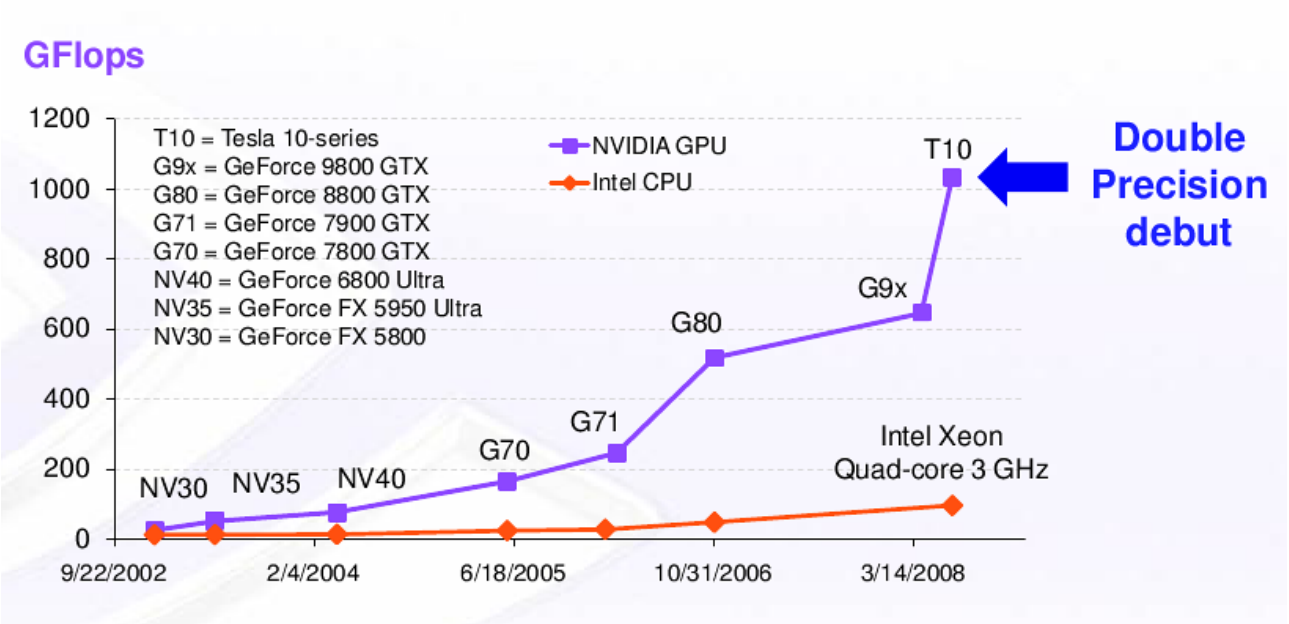
Pourquoi utiliser les GPU ?

- comparaison CPU/GPU : beaucoup plus de transistors dédiés au calcul sur les GPU, moins au contrôle



Pourquoi utiliser les GPU ?

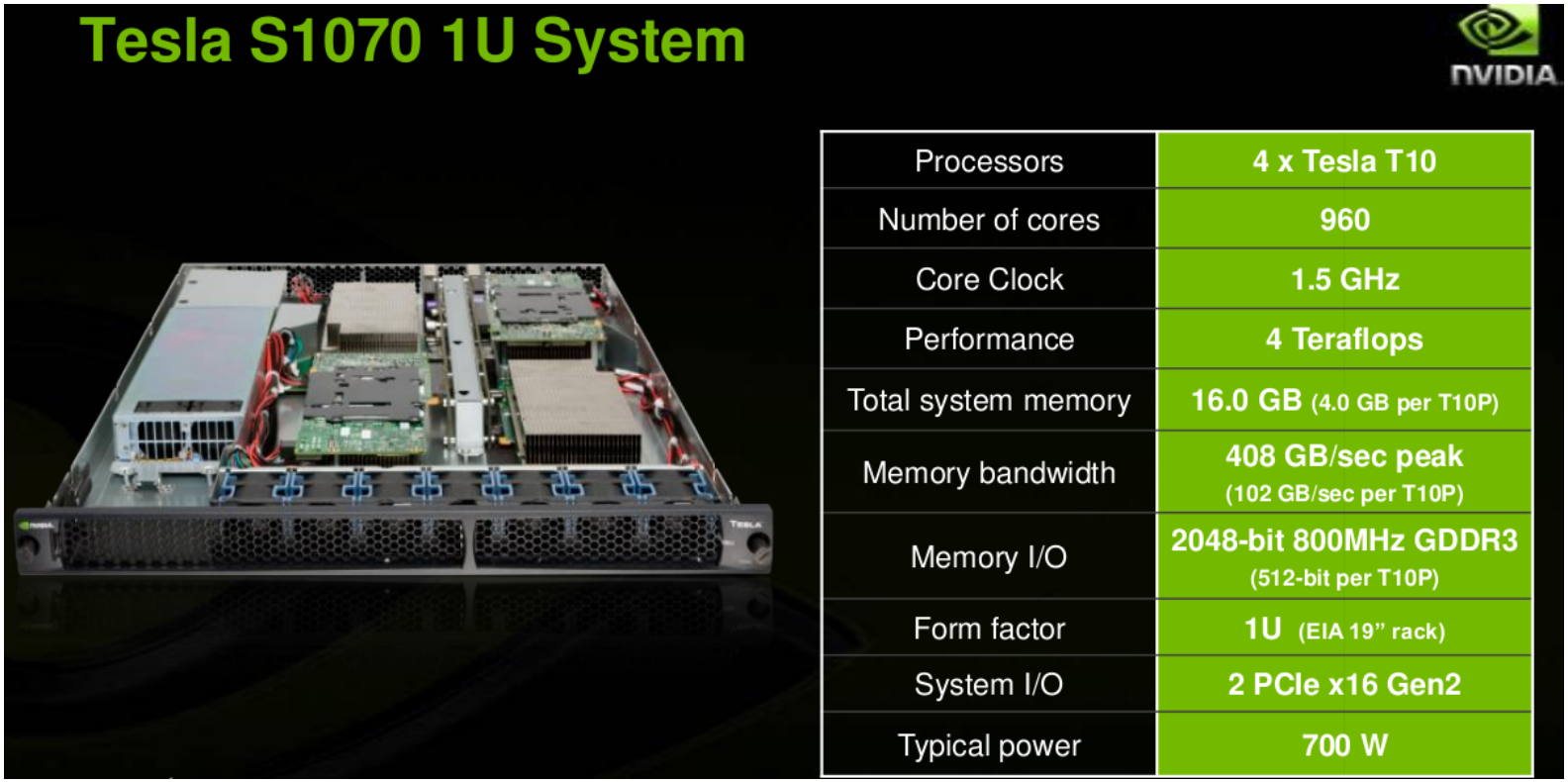
- produit de masse (donc modèle pérenne ?); GPU = super-calculateur *domestique*



Pourquoi utiliser les GPU ?

- accessibilité : modèle de programmation ***simple*** :
  - l'utilisateur lance un ensemble **structuré** de *thread*
  - modèle mémoire **CRCW**
  - extension du langage C
  - interopérabilité avec l'API graphique (OpenGL); (faciliter le développement d'un GUI sans avoir à transférer intermédiairement les données sur la mémoire carte mère par exemple)
- CUDA : Compute Unified Device Architecture
  Unification du matériel et du logiciel pour avoir une architecture pour le calcul parallèle

Serveur dédié au GPU Computing



Processors	4 x Tesla T10
Number of cores	960
Core Clock	1.5 GHz
Performance	4 Teraflops
Total system memory	16.0 GB (4.0 GB per T10P)
Memory bandwidth	408 GB/sec peak (102 GB/sec per T10P)
Memory I/O	2048-bit 800MHz GDDR3 (512-bit per T10P)
Form factor	1U (EIA 19" rack)
System I/O	2 PCIe x16 Gen2
Typical power	700 W

- gain en performance
- gain en consommation électrique
- fiabilité ???
  http://gpgpu.univ-perp.fr/images/3/36/GPU_Reliability_2008.pdf

GPU computing : double precision

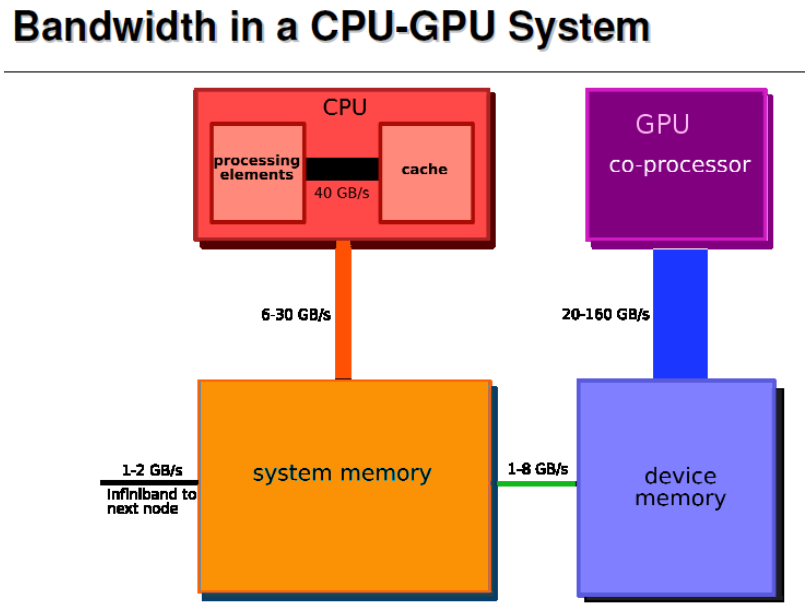
Double précision

Double Precision Floating Point

	NVIDIA GPU	SSE2	Cell SPE
Precision	IEEE 754	IEEE 754	IEEE 754
Rounding modes for FADD and FMUL	All 4 IEEE, round to nearest, zero, inf, -inf	All 4 IEEE, round to nearest, zero, inf, -inf	Round to zero/truncate only
Denormal handling	Full speed	Supported, costs 1000's of cycles	Flush to zero
NaN support	Yes	Yes	No
Overflow and Infinity support	Yes	Yes	No infinity, clamps to max norm
Flags	No	Yes	Some
FMA	Yes	No	Yes
Square root	Software with low-latency FMA-based convergence	Hardware	Software only
Division	Software with low-latency FMA-based convergence	Hardware	Software only
Reciprocal estimate accuracy	24 bit	12 bit	12 bit
Reciprocal sqrt estimate accuracy	23 bit	12 bit	12 bit
log2(x) and 2^x estimates accuracy	23 bit	No	No

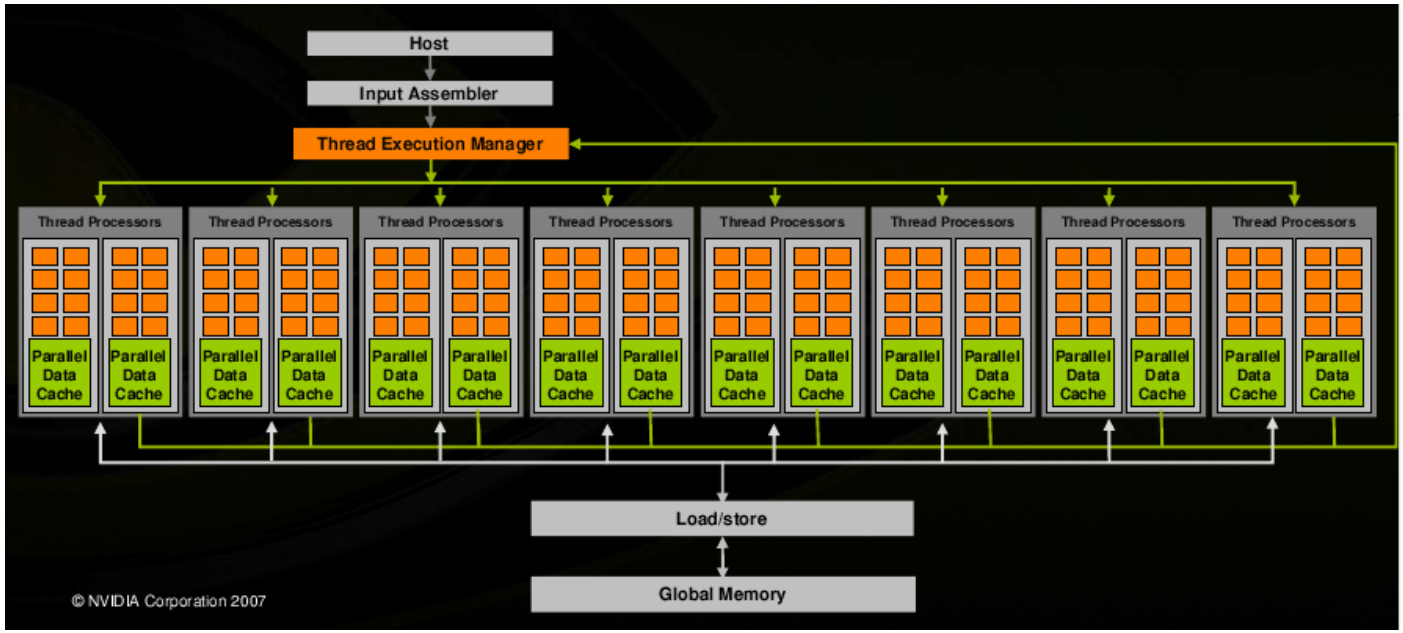
CPU/GPU : bande-passante mémoire théorique

- **lien CPU-GPU**, bus Pci-express x16, Gen2 :
  $BP = 16 * 2 * 250\text{MBytes/s} =$ **8 GBytes/s**
- **CPU-local** : mémoire DDR2 ($f = 266\text{MHz}$)
  BP= $266 * 10^6$*4 (transferts/cycle) * 8 (bytes) = **8.5 GBytes/s**
- **GPU-local (carte GTX280)** : bus 512 bit, DDR@$f = 1100\text{MHz}$,
  BP= $2 * 1100 * 10^6$(transferts/s) * 512/8 (bytes/transfert)= **140 GBytes/s**



Architecture Matérielle Hiérarchique : Nvidia CUDA : G80 - G200 - Fermi

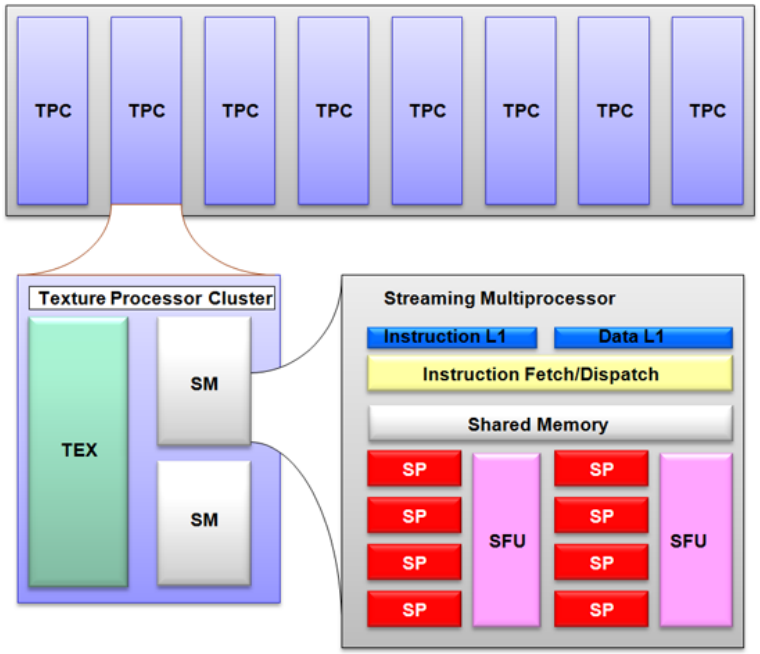
cours CUDA :
http://courses.ece.uiuc.edu/ece498/al/Syllabus.html



- 8 TPC (*Thread Processor Cluster*) - 2 SM (*Streaming Multiprocessor*) - 8 SP (*Streaming Processor*, SIMD)
- 8 TPC × 2 SM × 8 SP-cores = 128 cores

Architecture Matérielle Hiérarchique : Nvidia CUDA : G80 - G200 - Fermi

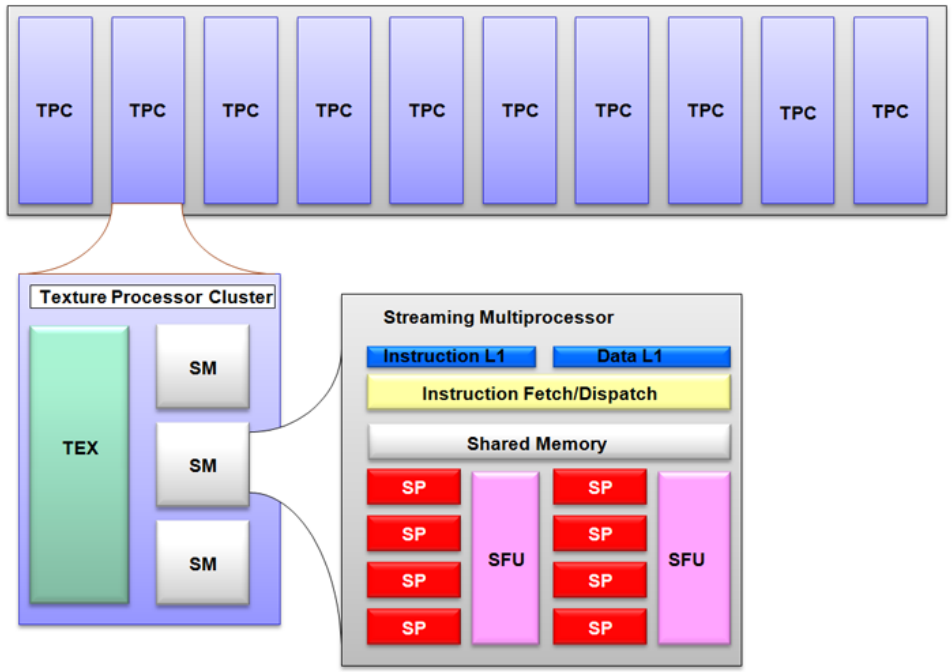
CUDA : G80 (fin 2006), hardware capability 1.0



- 8 TPC (*Thread Processor Cluster*) - 2 SM (*Streaming Multiprocessor*) - 8 SP (*Streaming Processor*, SIMD)
- 8 TPC × 2 SM × 8 SP-cores = 128 cores

Architecture Matérielle Hiérarchique : Nvidia CUDA : G80 - G200 - Fermi

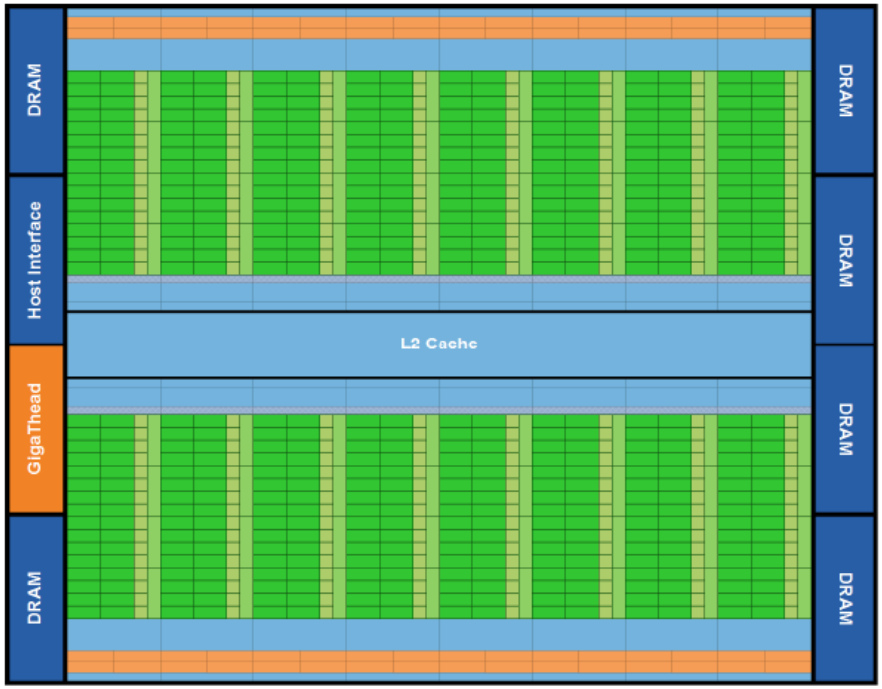
CUDA : G200 (mi 2008), hardware capability 1.3



- 10 TPC - 3 SM - 8 SP-cores
- 10 TPC × 3 SM × 8 SP-cores = 240 cores

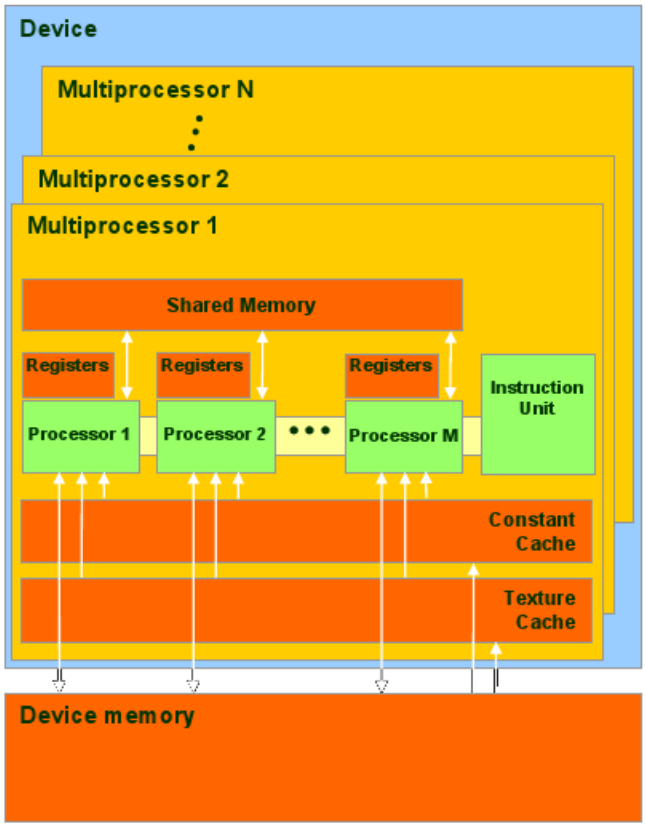
Architecture Matérielle Hiérarchique : Nvidia CUDA : G80 - G200 - Fermi

CUDA : Fermi (debut 2010), hardware capability 1.x



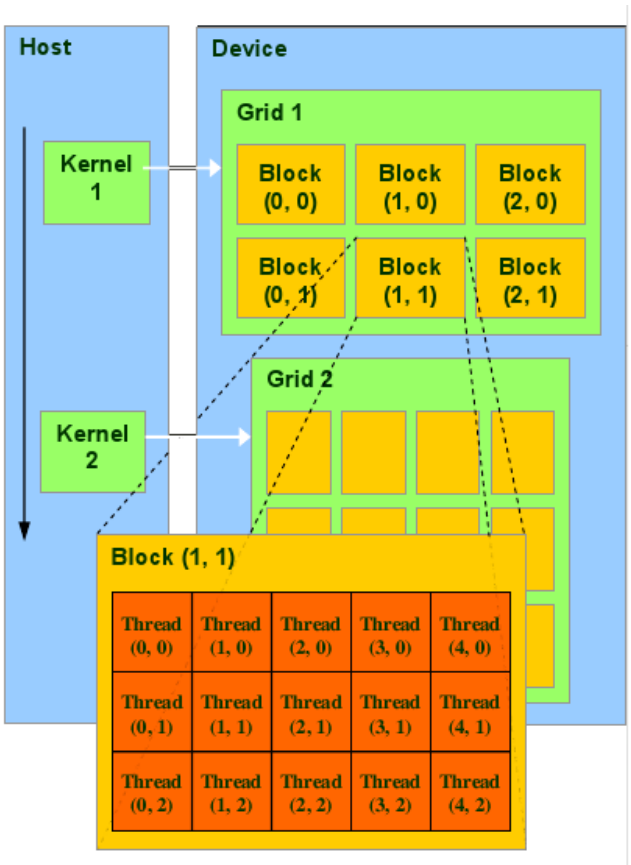
- 16 SM - 32 SP-cores
- 16 SM × 32 SP-cores = 512 cores

Architecture Mémoire Nvidia CUDA : G80/GT200



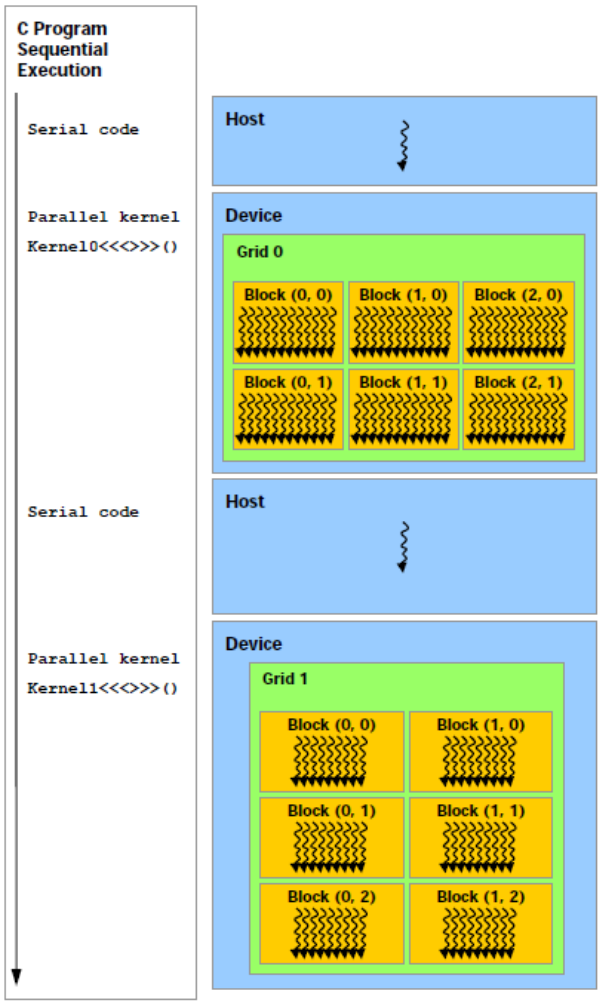
- espaces mémoire *off-chip* :
  - **mémoire globale** : RW, jusqu'à qq GBytes, lente (qq 100 cycles d'horloge)
- espaces mémoire *on-chip* :
  - **texture** : RO, physiquement en mémoire globale mais cachée
  - **constant** : RO, 64kB/puce, très rapide (1-4 cycles)
  - **partagée** : RW, très rapide (si conflits de banques évités), 16kB/multiprocesseur
  - **registre** : RW, très rapide, 8-16kB/multiprocesseur

CUDA : Modèle de programmation



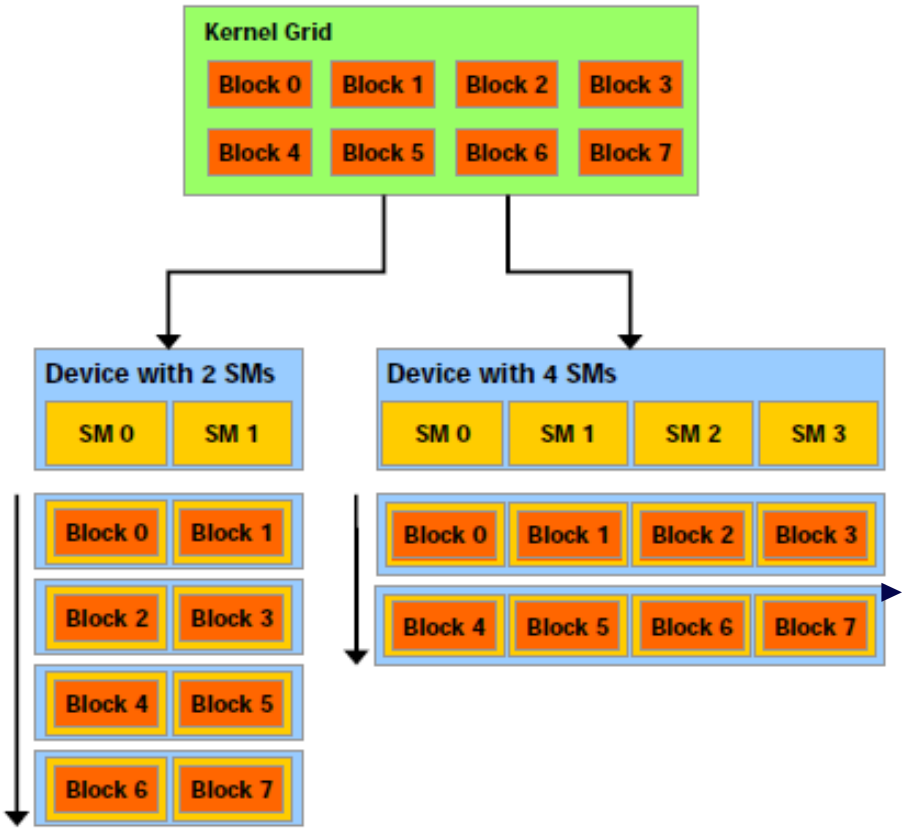
- écrire un programme pour un ~~pixel~~ *thread*
- instantier ce programme sur beaucoup de *threads* en parallèle
- grille (**logique**) indexée de blocs de *threads* : *threadIdx* et *blockIdx* (*variables intrinsèques*)
- les *threads* d'un même bloc ont accès à un espace de mémoire partagée
- les blocs sont indépendants, pas de synchronisation, execution dans un ordre indéfini

CUDA : Modèle de programmation



- systèmes hétérogènes : CPU et GPU ont des espaces mémoires séparés (*host* et *device*)
- l'utilisateur se concentre sur la parallélisation d'un algorithme (niveau logique) et pas sur la mécanique de séquencement des blocs de *threads* sur les multiprocesseurs.

CUDA : Modèle de programmation



- les blocs de *threads* peuvent être executé dans n'importe quel ordre
  - concurrent (sur des multiprocesseurs différents)
  - séquentiellement (sur le même multiprocesseur)
- l'indépendance des blocs assure une extensibilité (*scalability*) par rapport au nombre de multiprocesseurs.

CUDA : Modèle d'execution - notion de *warp*

`/usr/local/cuda23/doc/ptx_isa_1.4.pdf`

- Les *kernels* sont associés à une grille de bloc de threads
  - un kernel executé à la fois (hardware ≤ 1.3)
  - plusieurs kernels executé simultanément (à partir de Fermi, mars 2010)
- **l'unité de contrôle des multiprocesseurs** crée, gère et planifie le séquencement (*scheduling*) des *threads* et leur exécution sous forme de **warp** (groupe de *threads* d'indexes consécutifs)
- *branch divergence* Un *warp* exécute une instruction à la fois. Si dans le code un même *warp*, on a un *branch* (if-then-else) qui dépend des données (du *threadIdx* par exemple), alors les 2 branches sont exécutées séquentiellement (les *threads* qui suivent la première branche du *if* doivent attendre ceux du *else* pour se retrouver de nouveau à la même instruction). Perte en performance.
- les *threads* GPU sont très légers (création et changement de contexte ne coûtent presque rien).

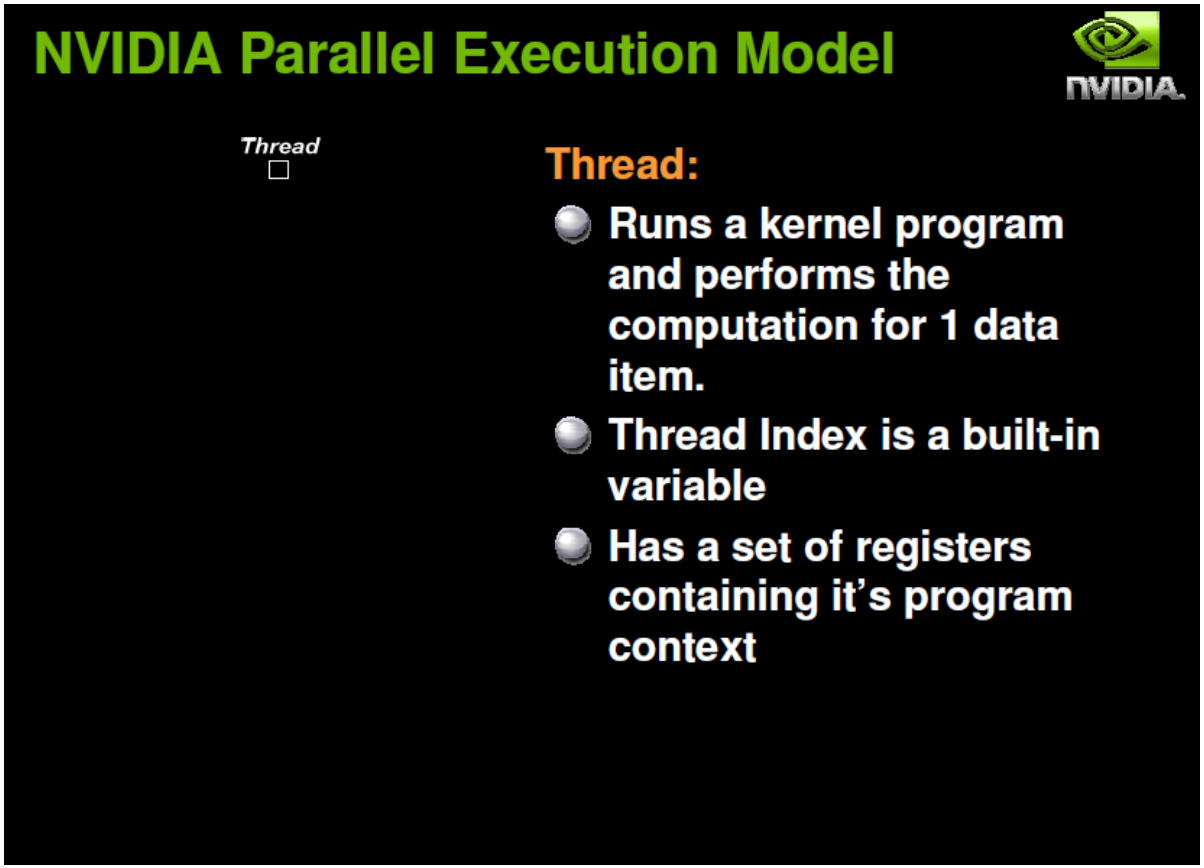
CUDA : Modèle d'execution - notion de *warp*

`/usr/local/cuda23/doc/ptx_isa_1.4.pdf`

- un bloc est exécuté sur un seul multi-processeur (pas de migration).
- plusieurs bloc peuvent être executés par un multiprocesseur
  - maximun 8 blocs par multiprocesseur
  - maximun 768 threads par multiprocesseur
- Avantages de CUDA / GPGPU d'avant 2006
  - modèle mémoire (accès aléatoire; mémoire partagée par bloc)
  - apprentissage beaucoup plus rapide; seulement quelques extensions en C; aucunes connaissance du *pipeline* graphique et de son API (OpenGL) requise

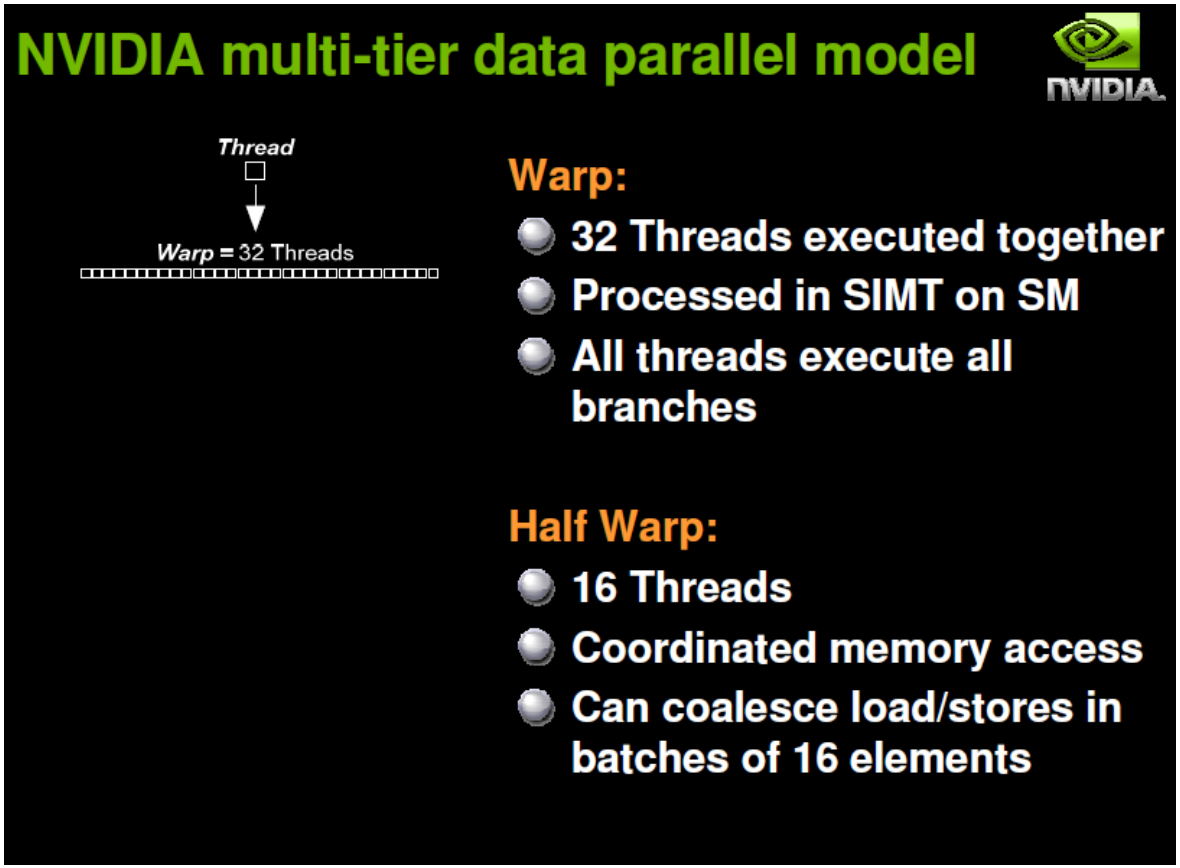
CUDA : SIMT Multithread Execution

Basé sur les slides de Brent Oster (NVIDIA)



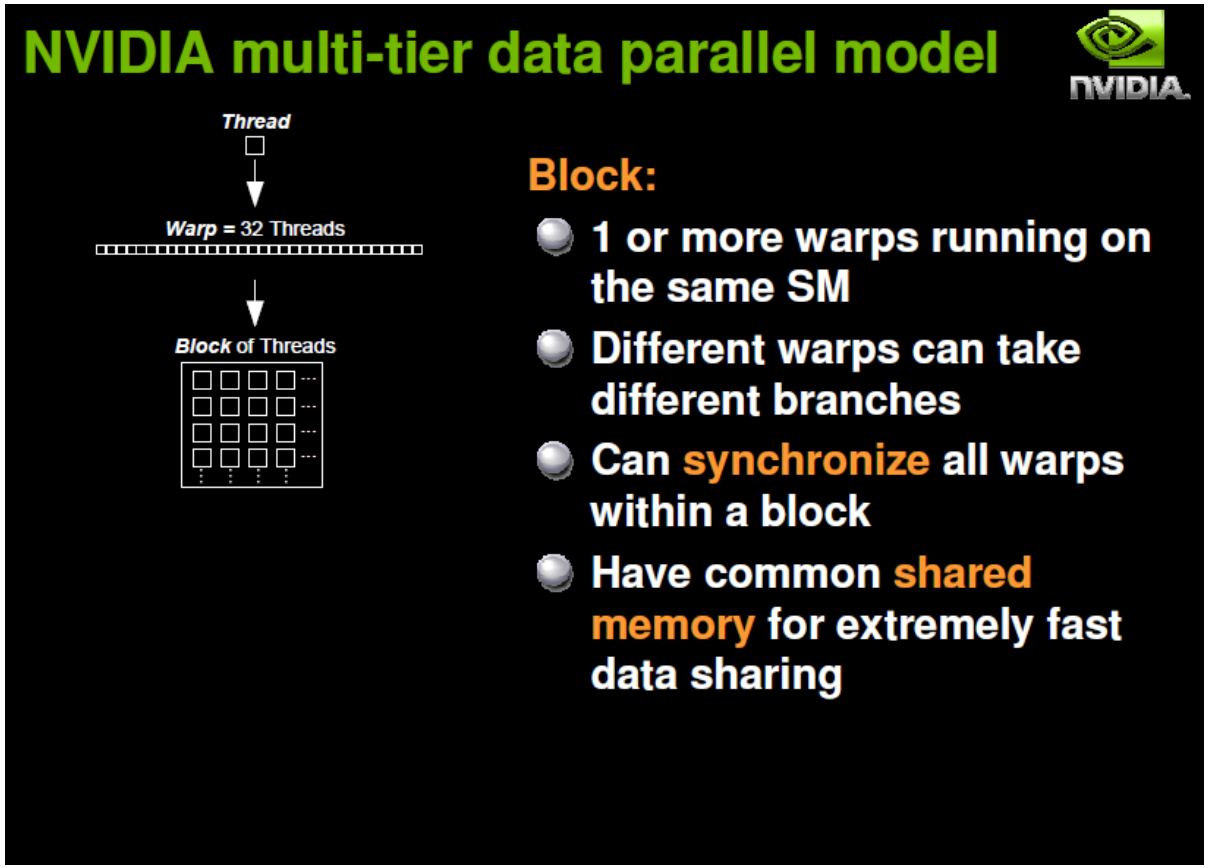
CUDA : SIMT Multithread Execution

Basé sur les slides de Brent Oster (NVIDIA)



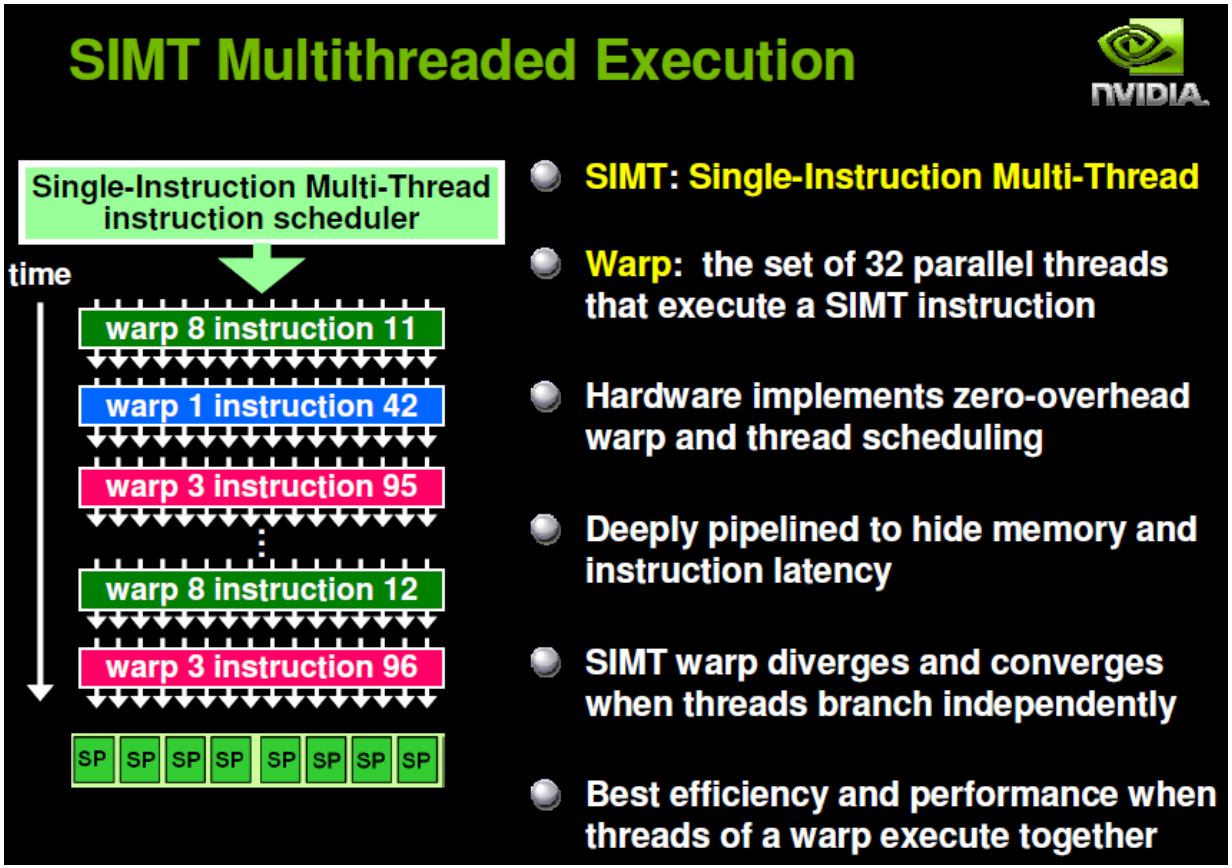
CUDA : SIMT Multithread Execution

Basé sur les slides de Brent Oster (NVIDIA)



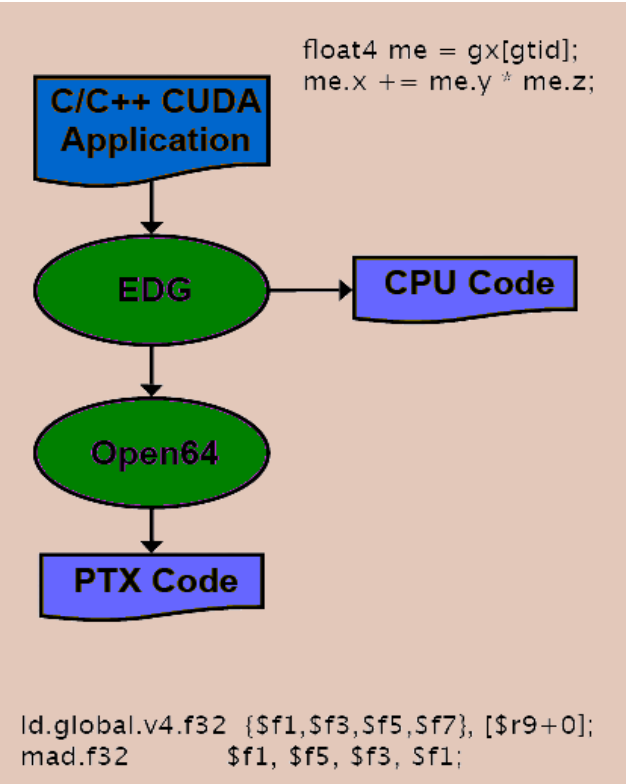
CUDA : SIMT Multithread Execution

Basé sur les slides de Brent Oster (NVIDIA)



CUDA : flot de compilation

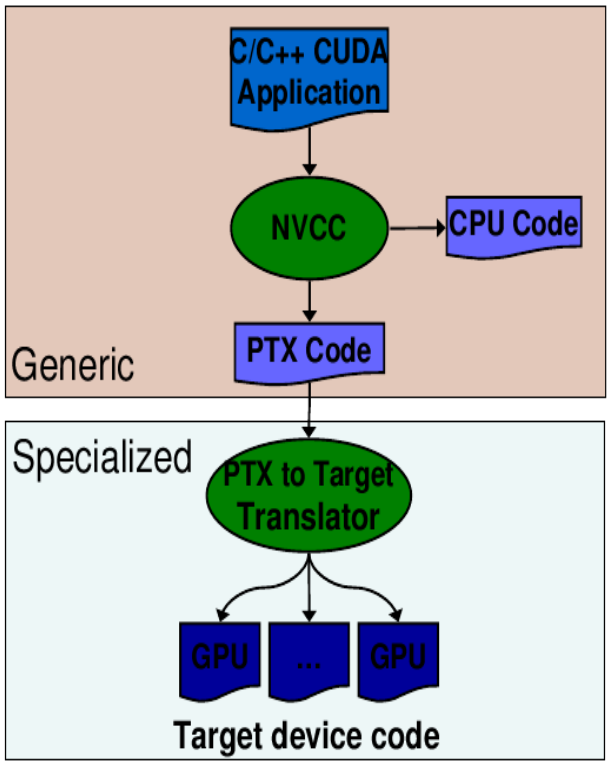
http://www.nvidia.com/docs/IO/55972/220401_Reprint.pdf



- NVCC : *compiler driver* : appelle en sous-main nvopencc (open64), gcc/g++, ...)
- PTX : *Parallel Thread eXecution*
- PTX définit un ISA (*Instruction Set Architecture*) et une architecture de machine virtuelle qui abstrait le matériel (portabilité et extensibilité)
- le pilote de la carte graphique transforme le binaire PTX en instructions machine bas-niveau et les transfère via le bus PCI-e directement au GPU.

CUDA : flot de compilation

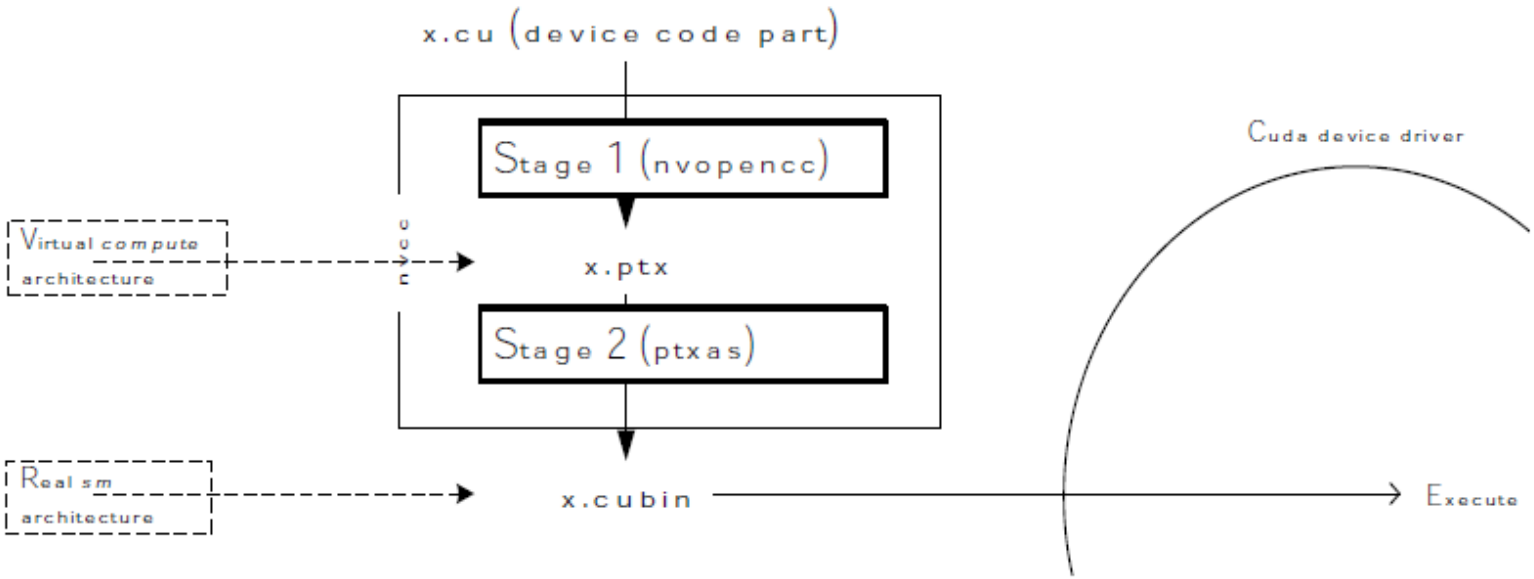
http://www.nvidia.com/docs/IO/55972/220401_Reprint.pdf



- la 2ème étape utilise l'outil ptxas : assembleur PTX vers cubin (instruction machine bas-niveau)

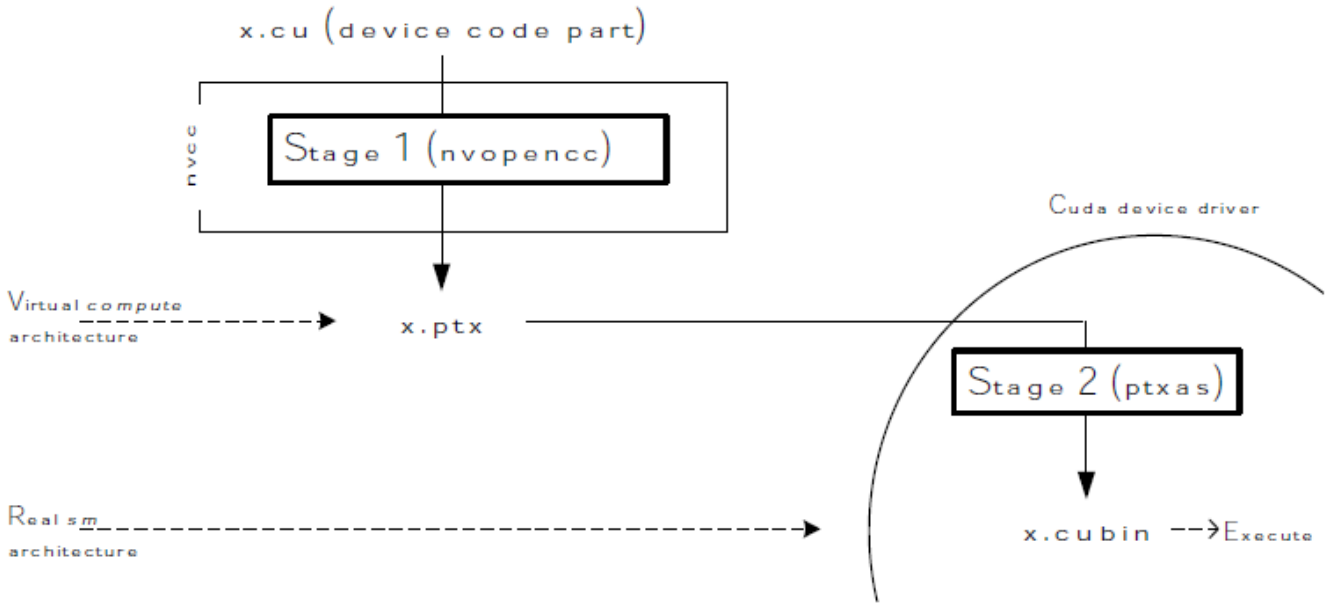
CUDA : flot de compilation

documentation nvcc : nvcc_2.3.pdf



CUDA : flot de compilation

documentation nvcc : nvcc_2.3.pdf



CUDA : Programmation en C et API

- modificateurs de déclaration

```
__global__ void KernelFunc(...); // kernel callable from host
__device__ void DeviceFunc(...); // function callable on device
__device__ int  GlobalVar;       // variable in device memory
__shared__ int  SharedVar;       // shared in PDC by thread block
__host__   void HostFunc(...);   // function callable on host
```

- variables intrinséques : indexes *threadIdx* et *blockDim*, *blockIdx* et *gridDim* (*read-only registers*)
- invocation de fonction en spécifiant les nombre de bloc et *threads*/bloc

```
KernelFunc<<<500, 128>>>(...); // launch 500 blocks w/ 128 threads each
```

- synchronisation des *threads* d'un même bloc

```
__syncthreads(); // barrier synchronization within kernel
```

- routines spécifiques inspirées de la *libc* (ex: allocation mémoire, transfert de données entre les mémoires globales du CPU et du GPU, ...)

CUDA : Programmation en C et API

- type de données spécifiques, voir l'entête vector_types.h : exemple *dim3*, propriété d'alignement mémoire

	Memory	Scope	Lifetime
`__shared__ int SharedVar;`	shared	thread block	thread block
`__device__ int GlobalVar;`	global	grid	application
`__constant__ int ConstantVar;`	constant	grid	application

- les variables sans qualificatifs sont *a priori* allouées dans les registres (sauf si une grande quantité est demandée, allocation en mémoire globale)
- la quantité de mémoire partagée par bloc peut être connue à la compilation ou au *runtime* (spécifié dans les paramètres de lancement d'un *kernel*)

CUDA : Programmation en C et API

`file:///usr/local/cuda23/doc/html/modules.html`

Main Page | Modules | Data Structures

Here is a list of all modules:

- CUDA Runtime API
  - Thread Management
  - Error Handling
  - Device Management
  - Stream Management
  - Event Management
  - Execution Control
  - Memory Management
  - OpenGL Interoperability
  - Direct3D 9 Interoperability
  - Direct3D 10 Interoperability
  - Texture Reference Management
  - Version Management
  - C++ API Routines
  - Data types used by CUDA Runtime
- CUDA Driver API
  - Initialization
  - Device Management
  - Version Management
  - Context Management
  - Module Management
  - Stream Management
  - Event Management
  - Execution Control
  - Memory Management
  - Texture Reference Management
  - OpenGL Interoperability
  - Direct3D 9 Interoperability
  - Direct3D 10 Interoperability
  - Data types used by CUDA driver

- *Runtime API* : haut-niveau, *device emulation*
- *Driver API* : bas-niveau, contrôle fin des GPU, un thread CPU peut contrôler plusieurs GPU, optimisation PTX JIT (Just-In-Time), code plus verbeux
- on ne peut pas utiliser les 2 API en même temps.

CUDA Programmation en C : exemple

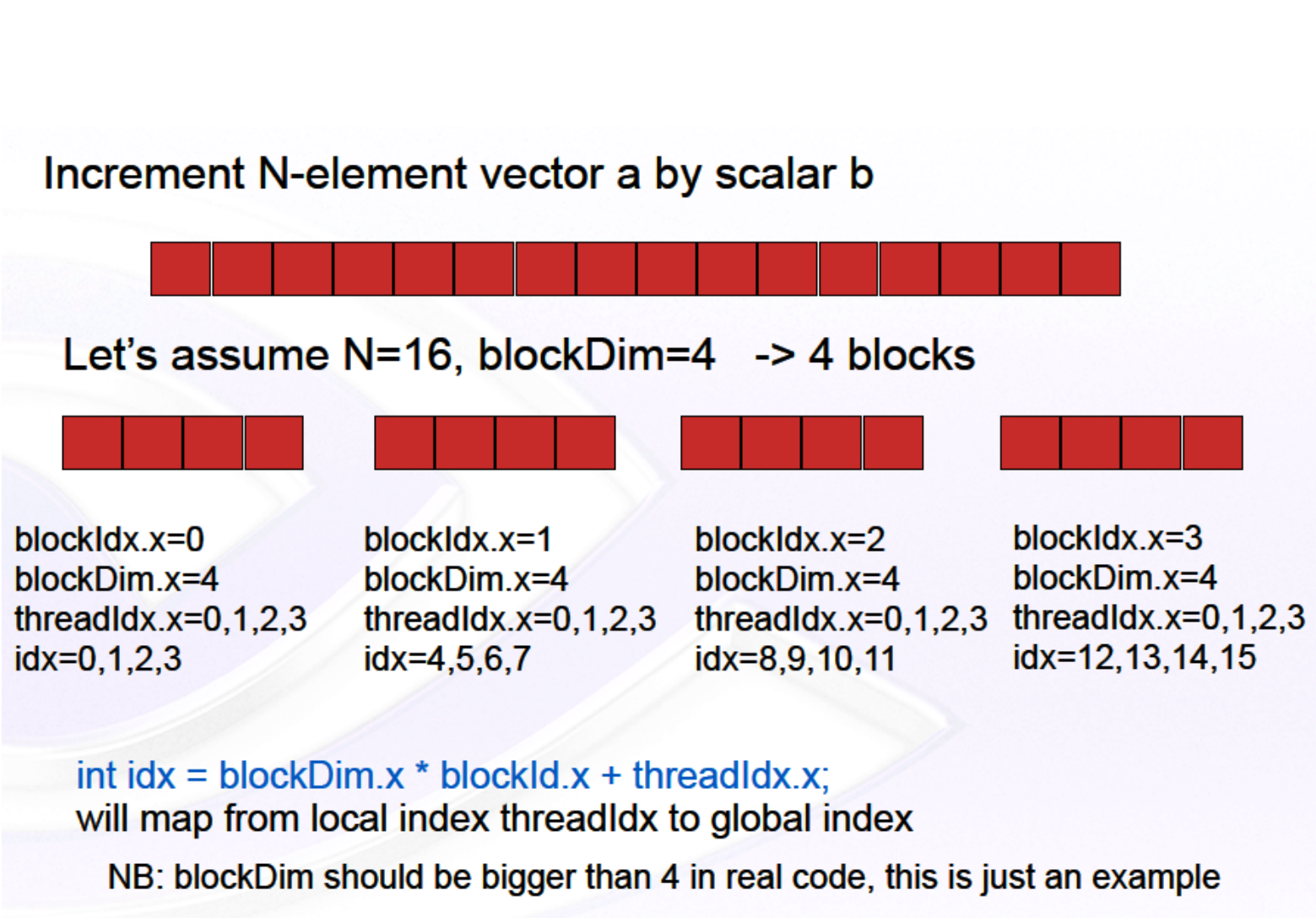
CPU program

```
void increment_cpu(float *a, float b, int N)
{
    for (int idx = 0; idx<N; idx++)
        a[idx] = a[idx] + b;
}


void main()
{
  .....
    increment_cpu(a,b,N);
}
```

CUDA program

```
__global__ void increment_gpu(float *a, float b)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    a[idx] = a[idx] + b;
}


void main()
{
  .....
    dim3 dimBlock (blocksize);
    dim3 dimGrid (N/blocksize);
    increment_gpu<<<dimGrid, dimBlock>>>(a,b);
}
```

CUDA Programmation en C : exemple



CUDA Programmation en C : exemple

CPU program

```
void increment_cpu(float *a, float b, int N)
{
    for (int idx = 0; idx<N; idx++)
        a[idx] = a[idx] + b;
}


void main()
{
  .....
    increment_cpu(a,b,N);
}
```

CUDA program

```
__global__ void increment_gpu(float *a, float b, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if( idx < N)
        a[idx] = a[idx] + b;
}


void main()
{
  .....
    dim3 dimBlock (blocksize);
    dim3 dimGrid (ceil(N / (float)blocksize));
    increment_gpu<<<dimGrid, dimBlock>>>(a,b,N);
}
```

CUDA Programmation en C : exemple

- exemple : addition de vecteur

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

- invocation

```
void main() {
  // allocate device (GPU) memory
  float* d_A, d_B, d_C;
  cudaMalloc( (void**) &d_A, N * sizeof(float));
  cudaMalloc( (void**) &d_B, N * sizeof(float));
  cudaMalloc( (void**) &d_C, N * sizeof(float));

  // copy host memory to device
  cudaMemcpy( d_A, h_A, N * sizeof(float),cudaMemcpyHostToDevice);
  cudaMemcpy( d_B, h_B, N * sizeof(float),cudaMemcpyHostToDevice);

  // Execute on N/256 blocks of 256 threads each
  vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Optimisations CUDA

Un programme CUDA doit prendre en compte les contraintes suivantes

- Accès coalescent en mémoire globale : des *threads* d'indexes consécutifs doivent accéder à des cases mémoires d'addresse consécutives, problème d'alignement
- Utiliser la mémoire partagée (bien meilleure bande passante que la mémoire globale)
- Utiliser efficacement le parallélisme
  Garder le GPU occupé un maximun de temps
  avoir un rapport calcul / accès mémoire élevé
  Utiliser au mieux la hiérarchie de threads.
- Tenir compte des conflits de banc mémoire

Algorithmes parallèles de base

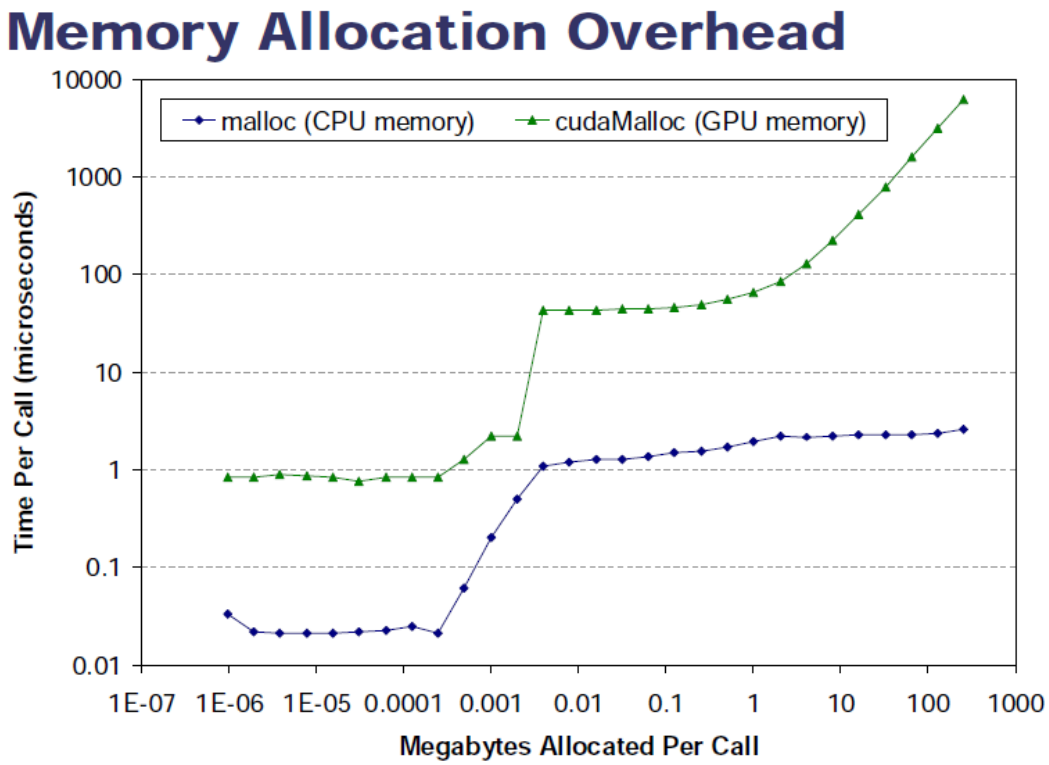
parallel-prefix-sum ("scan"), parallel sort and parallel reduction

- Thrust : http://code.google.com/p/thrust
- cudpp : http://gpgpu.org/developer/cudpp
- comparaison Thrust/CUDPP : http://code.google.com/p/thrust/wiki/ThrustAndCUDPP
- exemple reduction dans le SDK Cuda

CUDA : Allocation mémoire et temps de transfert

Basé sur les slides de K. Skadron (University of Virginia)

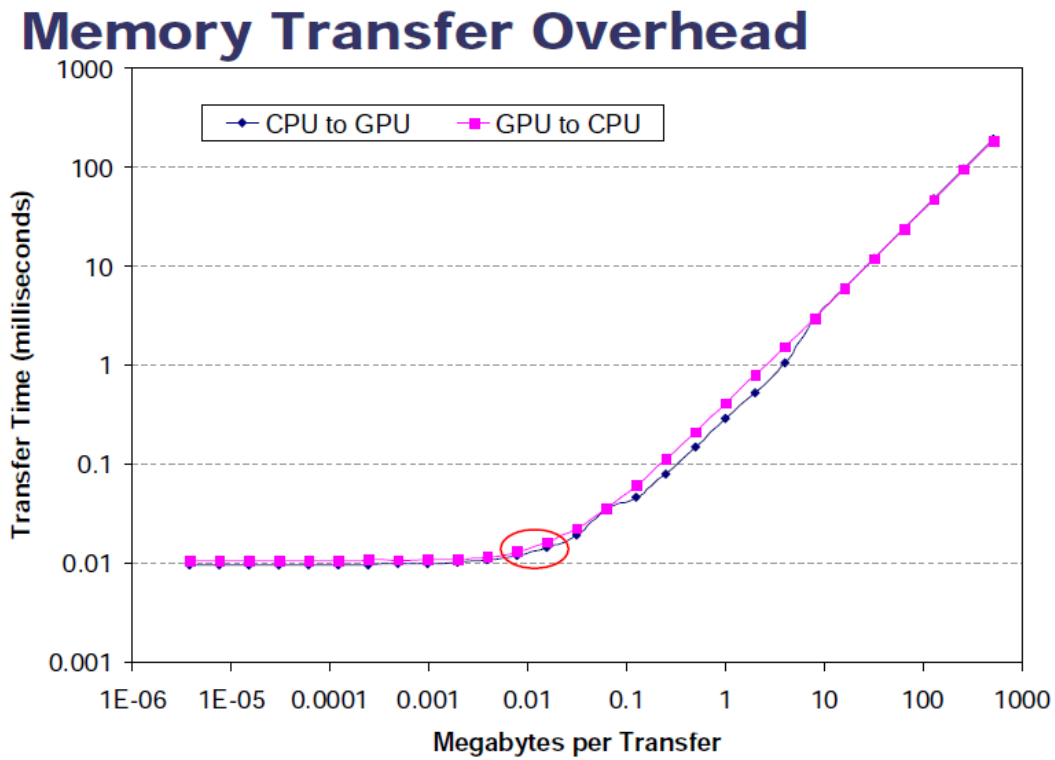
- allocation en mémoire globale du GPU coû teuse



CUDA : Allocation mémoire et temps de transfert

Basé sur les slides de K. Skadron (University of Virginia)

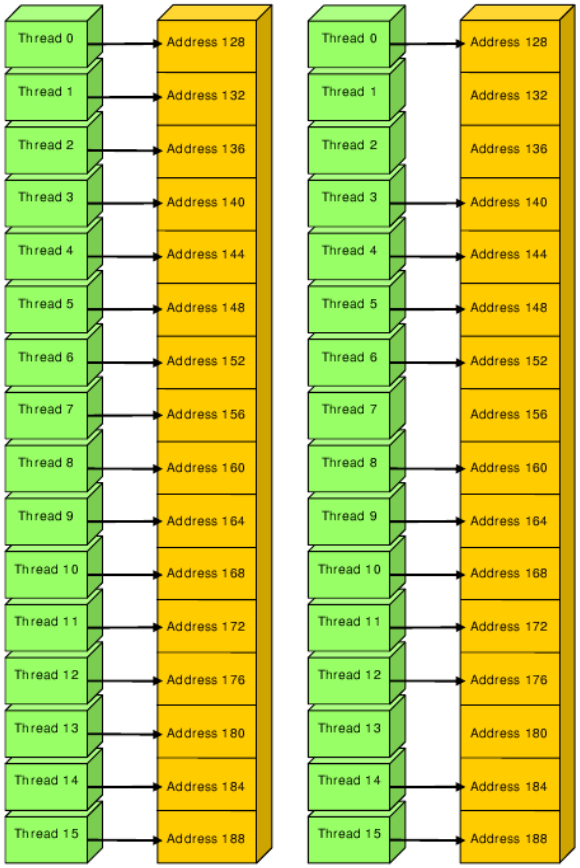
- temps de transfert CPU ←→ non-négligeable (exemple de calcul d'une FFT2D)



CUDA optimisation

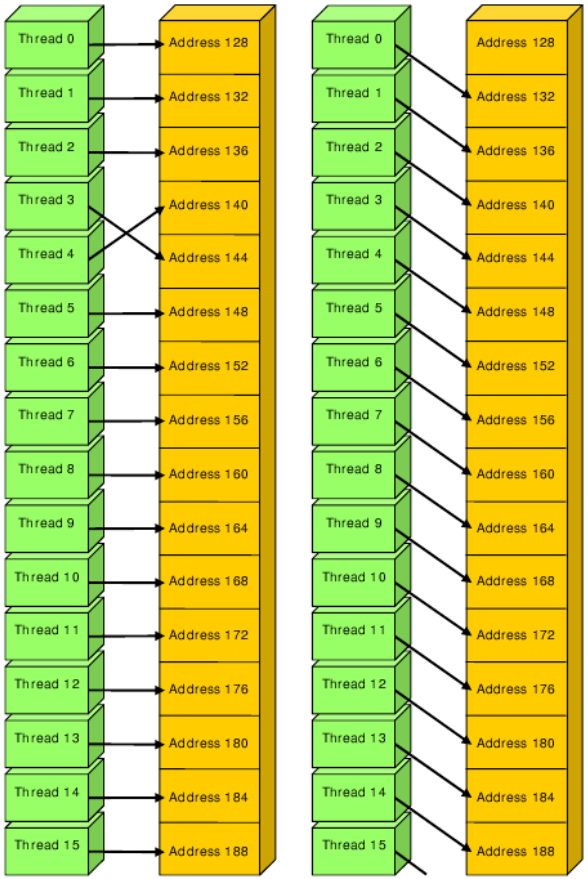
- voir les slides de isc2009 : CUDA Optimisation
- voir les slides http://moss.csc.ncsu.edu/~mueller/cluster/nvidia/GPU+CUDA.pdf exemple de code de transposition de matrice (illustration des concept de coalescence, conflit de banc memoire, ...) à partir du slide 142.

CUDA optimisation



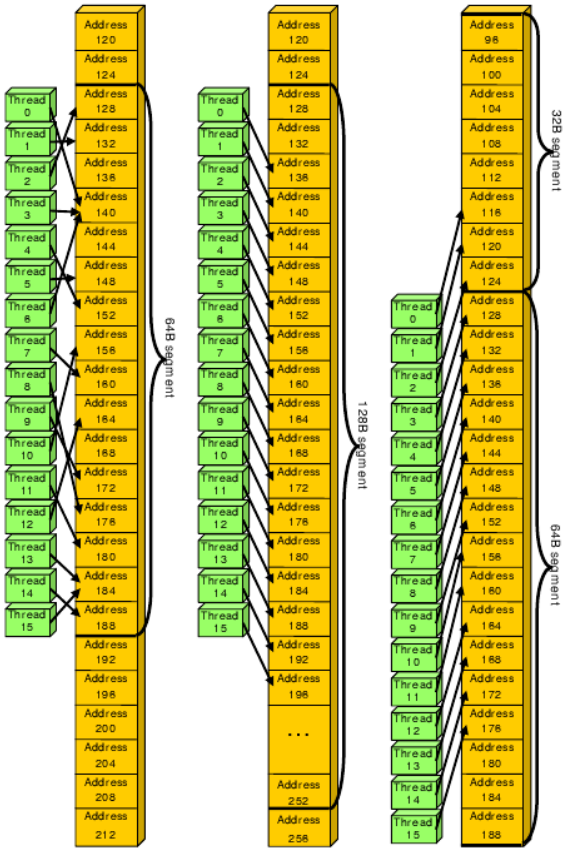
Left: coalesced **`float`** memory access, resulting in a single memory transaction.
Right: coalesced **`float`** memory access (divergent warp), resulting in a single memory transaction.

CUDA optimisation



Left: non-sequential **float** memory access, resulting in 16 memory transactions.
Right: access with a misaligned starting address, resulting in 16 memory transactions.

CUDA optimisation



Left: random **float** memory access within a 64B segment, resulting in one memory transaction.
Center: misaligned **float** memory access, resulting in one transaction.
Right: misaligned **float** memory access, resulting in two transactions.

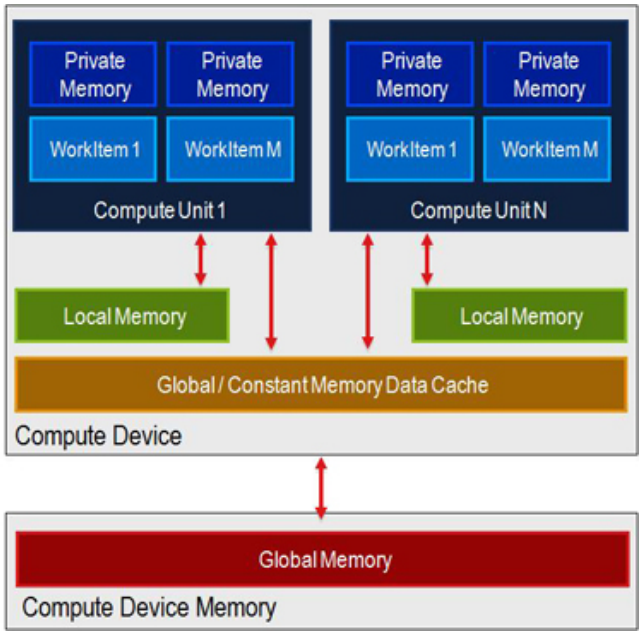
CUDA : profiler / terminologie

- gld_incoherent: Number of non-coalesced global memory loads
- gld_coherent: Number of coalesced global memory loads
- gst_incoherent: Number of non-coalesced global memory stores
- gst_coherent: Number of coalesced global memory stores
- local_load: Number of local memory loads
- local_store: Number of local memory stores
- branch: Number of branch events taken by threads
- divergent_branch: Number of divergent branches within a warp
- instructions: instruction count
- warp_serialize: Number of threads in a warp that serialize based on address conflicts to shared or constant memory
- cta_launched: executed thread blocks

GPU computing - CUDA

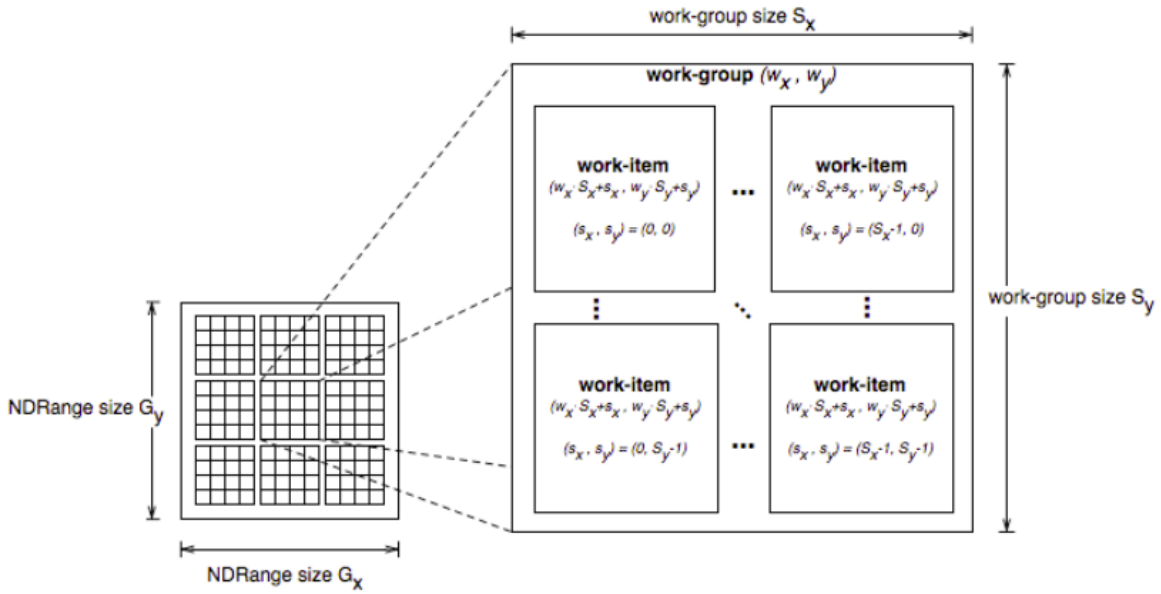
- Nivia Performance Primitives
- NVIDIA NPP is a library of functions for performing CUDA accelerated processing
- The initial set of functionality in the library focuses on imaging and video processing

GPU computing - OpenCL



- OpenCL sur Wikipedia // Introduction OpenCL
- standard du groupe de travail `http://www.khronos.org`, version 1.0 (12/2008)
- programmer les GPU (Nvidia/ATI), les CPU multi-cœurs et les CELL dans un même language/modèle de programation : *Data and task parallel compute model*
- OpenCL emprunte de nombreuses caractéristiques à CUDA

GPU computing - OpenCL



- OpenCL sur Wikipedia // Introduction OpenCL
- standard du groupe de travail `http://www.khronos.org`, version 1.0 (12/2008)
- programmer les GPU (Nvidia/ATI), les CPU multi-cœurs et les CELL dans un même language/modèle de programation : *Data and task parallel compute model*
- OpenCL emprunte de nombreuses caractéristiques à CUDA

GPU computing - OpenCL

	OpenCL	C for CUDA
Driver-style API	Yes	Yes, Optional
Language Integration	No	Yes
C-like kernels	Yes	Yes
Full pointer support	No	Yes
C++ Language Features	No	Yes
Context management	Explicit	Implicit
Asynchronous execution	Context mode	API call dependent
Synchronization	Sync objects	Ordered operation containers
Multi-device sync	Yes	No
Profiling API	Yes	Through Events
Memory management	Objects	Pointers
Cross-device data sharing	Implicit or Explicit	Explicit
Source Level JIT	Yes	No
Device Independent Deployment	Yes	Partial (only between GPUs)

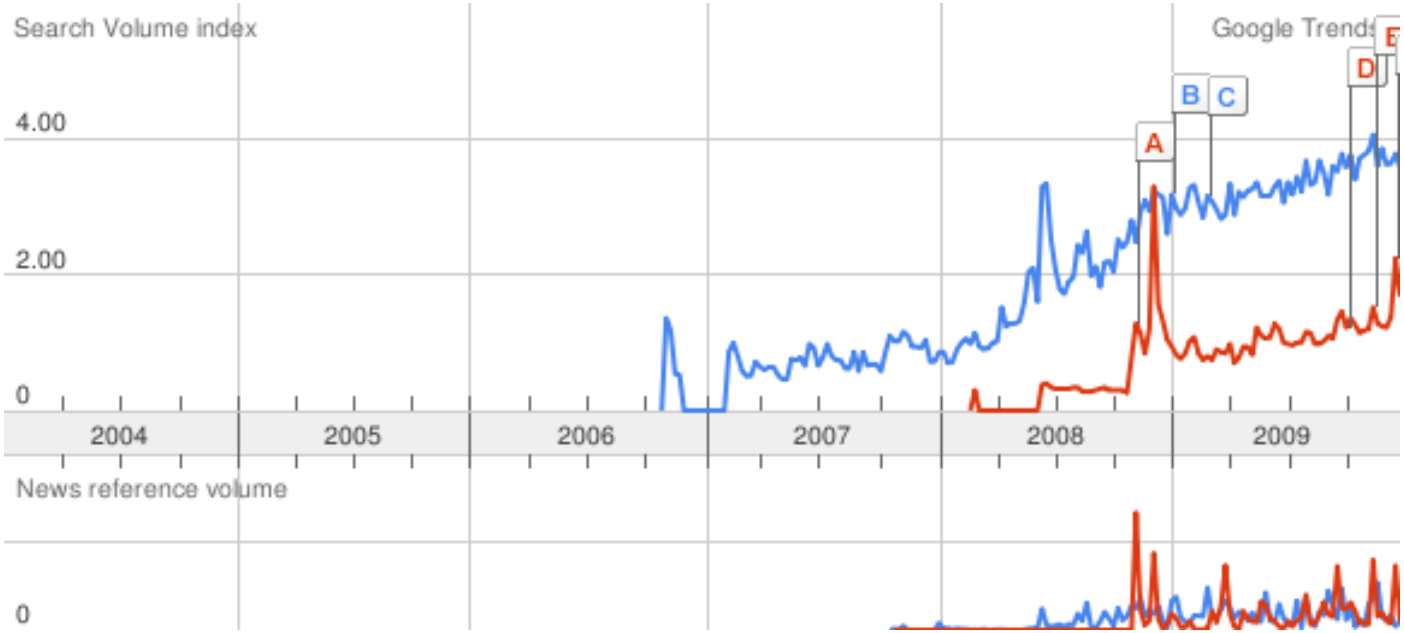
- transfert CUDA vers OpenCL :

  http://developer.amd.com/documentation/articles/pages/OpenCL-and-the-ATI-Stream-v2.0-Beta.aspx#four

- CUDA en avance de phase sur OpenCL; CUDA 3.0 (fin 2009)

GPU computing - OpenCL

Google Trends :



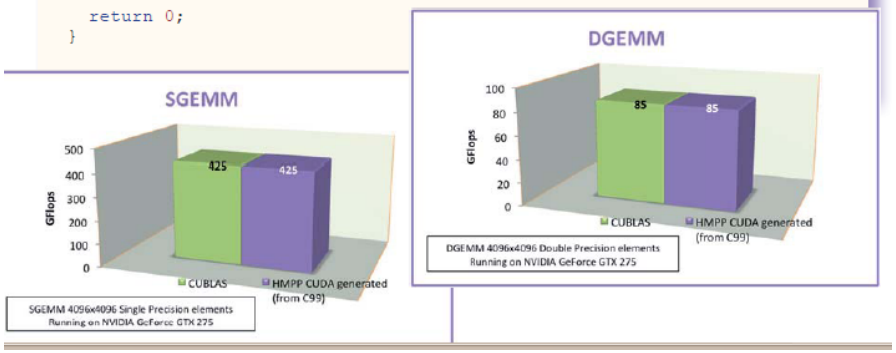
GPU computing - OpenCL

Google Trends :

GPU computing - HMPP (Caps)

- programmation par directive *à la OpenMP*: CAPS et l'outil HMPP (Hybrid Manycores Parallel Programing)

```
#pragma hmpp sgemm codelet, args[m;n;k;alpha;beta;a;b].io=in, &
#pragma hmpp sgemm args[c].io=inout, target=CUDA:BROOK
void sgemm(int m, int n, int k,
                    float alpha, float a[m][k], float b[k][n],
                    float beta, float c[m][n]);

int main(int argc, char **argv) {

...

/* Allocate device and memory */
#pragma hmpp sgemm allocate, args[c].size={M,N}
/* Prefetch all data, alpha and beta are constant */
#pragma hmpp sgemm advancedload, args[alpha;beta;a;b;c], &
#pragma hmpp sgemm args[m;n;k;n;alpha;beta].const=true
...

#pragma hmpp sgemm callsite, args[alpha;beta;a;b;c].advancedload=true, &
#pragma hmpp sgemm asynchronous
  sgemm( M, N, K, alpha, t1, t2, beta, t3 );

/* asynchronous execution barrier */
#pragma hmpp sgemm synchronize
/* retrieve c output data  */
#pragma hmpp sgemm delegatedstore, args[c]
/* release the device */
#pragma hmpp sgemm release
...

  return 0;
}
```



Installer CUDA

- Avoir un GPU compatible CUDA : G80 ou plus (≥ 2007)
- système exemple : OS Linux - x86_64 (Novembre 2009) - CUDA 2.3 [1]
- Driver Nvidia noyau : 190.18 - `cudadriver_2.3_linux_64_190.18.run`
- chaîne de compilation (*toolkit*): `cudatoolkit_2.3_linux_64_ubuntu9.04.run`
- SDK : `cudasdk_2.3_linux.run`
  - utilitaires : passage des arguments de la ligne de commande, mesure de temps de calcul, *read/write* PGM, GLUT ...
  - CUDPP (*CUDA Data Parallel Primitives Library*)
  - ~ 70 exemples d'applications : reduction, produit scalaire, traitement d'images, finance, ...
- Mode emulation : le driver n'est pas nécessaire

[1] !!! CUDA 2.3 pas compatible avec gcc 4.4

Avoir un GPU compatible CUDA

- exécuter *deviceQuery*
- header `cuda_runtime_api.h` : cudaGetDeviceCount, cudaGetDeviceProperties
- connaître la version du driver installé : `cat /proc/driver/nvidia/version`

Quadro FX4600

```
Fichier Édition Affichage Terminal Onglets Aide
pkestene@i... pkestene@i... pkestene@i... pkestene@i... pkestene@i...
CUDA Device Query (Runtime API) version (CUDART static linking)
There is 1 device supporting CUDA

Device 0: "Quadro FX 4600"
  CUDA Driver Version:                           2.30
  CUDA Runtime Version:                          2.30
  CUDA Capability Major revision number:         1
  CUDA Capability Minor revision number:         0
  Total amount of global memory:                 804585472 bytes
  Number of multiprocessors:                     12
  Number of cores:                               96
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 8192
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          262144 bytes
  Texture alignment:                             256 bytes
  Clock rate:                                    1.19 GHz
  Concurrent copy and execution:                 No
  Run time limit on kernels:                     Yes
  Integrated:                                    No
  Support host page-locked memory mapping:       No
  Compute mode:                                  Default (multiple host
threads can use this device simultaneously)

Test PASSED

Press ENTER to exit...
```

Tesla C1060

```
kestener@uchu1: ~/install/nvidia/sdk/2.3/C
Fichier Édition Affichage Terminal Onglets Aide
kestener@uchu1: ~/install/nvidia/sdk/2.... pkestene@irfupcj92: ~
CUDA Device Query (Runtime API) version (CUDART static linking)
There are 2 devices supporting CUDA

Device 0: "Tesla C1060"
  CUDA Driver Version:                           2.30
  CUDA Runtime Version:                          2.30
  CUDA Capability Major revision number:         1
  CUDA Capability Minor revision number:         3
  Total amount of global memory:                 4294705152 bytes
  Number of multiprocessors:                     30
  Number of cores:                               240
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 16384
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          262144 bytes
  Texture alignment:                             256 bytes
  Clock rate:                                    1.30 GHz
  Concurrent copy and execution:                 Yes
  Run time limit on kernels:                     No
  Integrated:                                    No
  Support host page-locked memory mapping:       Yes
  Compute mode:                                  Default (multiple host
threads can use this device simultaneously)

Device 1: "Tesla C1060"
  CUDA Driver Version:                           2.30
  CUDA Runtime Version:                          2.30
  CUDA Capability Major revision number:         1
```

SDK CUDA

- les exemples du SDK fonctionnent suivant un même schéma (voir le projet *template*), en général 3 fichiers sources :
  - `nom_du_projet.cu` : contient le *main* et donc le point d'entrée vers l'exécution de code sur le GPU
  - `nom_du_projet_kernel.cu` : version CUDA d'un algorithme
  - `nom_du_projet_gold.c` : vers CPU native, pour comparaison fonctionnelle des résultats et éventuellement *benchmark* des performances
- Quelques exemples plus importants *pédagogiquement*
  - `transpose` : notion de coalescence des accès mémoire
  - `reduction` : exemple de calcul de la somme des éléments d'un tableau (les algorithmes sous-jacents ont été transformé en librairie (C++ avec *template*) : voir Thrust et CUPP)
- On peut mettre du code CUDA dans un fichier C/CPP standard, à condition de le protéger par la macro `__CUDACC__`

Cours/Tutoriels en ligne sur CUDA

- multimédia : http://www.nvidia.com/object/cuda_education.html
- CUDA Education http://www.nvidia.com/object/cuda_university_courses.html
- un des 1er cours sur CUDA Université Illinois
- Workshop ASPLOS2008 : http://gpgpu.org/asplos2008
- École thématique ARCHI09 : http://www.irisa.fr/archi09/defour.pdf (historique processeur vectoriel / précurseur GPU).
- Parallel Programing (John Hopkins University) : cs420
- NCSA, 1er avril 2009
- http://sites.google.com/site/cudaiap2009/home
- CUDA wiki at DAAC
- CUDA Tutorial at ISC2009
- un bon résumé : http://www.caam.rice.edu/~timwar/NUDG/RMMC/CUDA.html

Bibliographie

- doc CUDA : CudaReferenceManual.pdf, nvcc_2.3.pdf, NVIDIA_CUDA_ProgrammingGuide_2.3.pdf, NVIDIA_CUDA_BestPracticesGuide_2.3.pdf, ptx_isa_1.4.pdf

S'informer en ligne sur le GPU Computing

- les flux RSS / News:
  - sur Twitter http://twitter.com/nvidiadeveloper et le flux RSS http://twitter.com/statuses/user_timeline/18691870.rss
  - les News Nvidia http://news.developer.nvidia.com
  - GPGPU.org Rss http://gpgpu.org/feed
- le forum Nvidia pour poser des questions sur CUDA : http://forums.nvidia.com/index.php?showforum=62
- le coin des développeurs Nvidia : http://developer.nvidia.com/page/home.html
- un blog intéressant : http://gpumodeling.blogspot.com/