

INSA-5IF

Calcul parallèle et GPU

Séance 1 : architecture des GPU

Christian Wolf
INSA-Lyon, Dép. IF

Planning



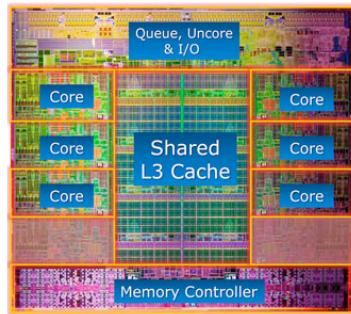
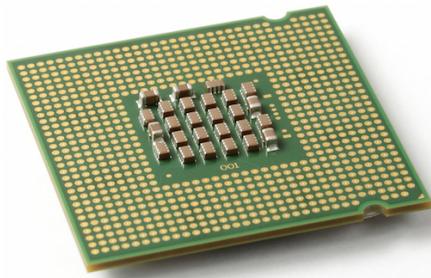
Christian W.	Cours	Introduction : architecture d'un GPU
Jonathan R.C.	Cours	Introduction : calcul en parallèle (1/2)
Jonathan R.C.	Cours	Introduction : calcul en parallèle (2/2)
Jonathan R.C.	Cours	Open-MP
Jonathan R.C.	TP	Open-MP
Christian W.	Cours + TP	Programmation en CUDA : introduction
Jonathan R.C.	TP	Open-MP
Christian W. + Lionel M.	Cours + TP	Programmation en CUDA : mémoires
Jonathan R.C.	TP	MPI
Jonathan R.C.	TP	MPI
Jonathan R.C.	TP	Runtime
Christian W.	Cours + TP	Débogage et optimisation sur GPU
Christian W.	Cours + TP	GPU Design patterns
Lionel M.	Cours + TP	Open-ACC
Lionel M.	Cours + TP	Open-ACC

CPU : Performances

Augmenter la performance d'un CPU est devenu difficile, car les voies classiques sont saturées :

- Augmentation de la fréquence
- Rendre plus rapide l'accès mémoire (Caches L1, L2, L3, L4)
- Paralléliser l'exécution des étapes de chaque instruction (pipeline)
- Exécution super scalaire
- Instructions vectoriels (MMX, SSE etc.)

Réponse : paralléliser



CPU multi-cœurs



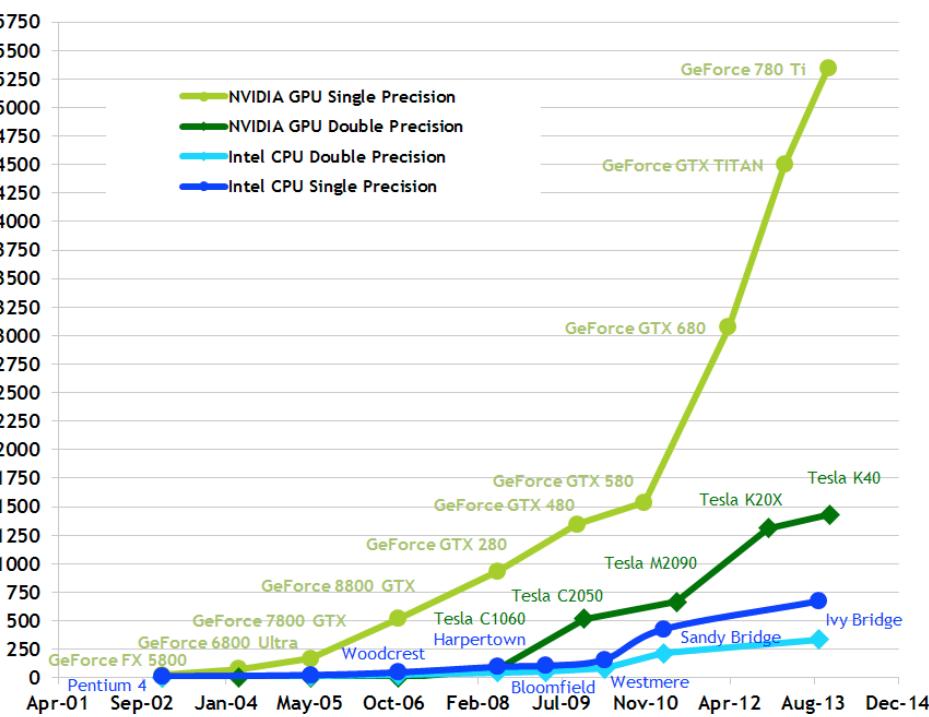
GPU



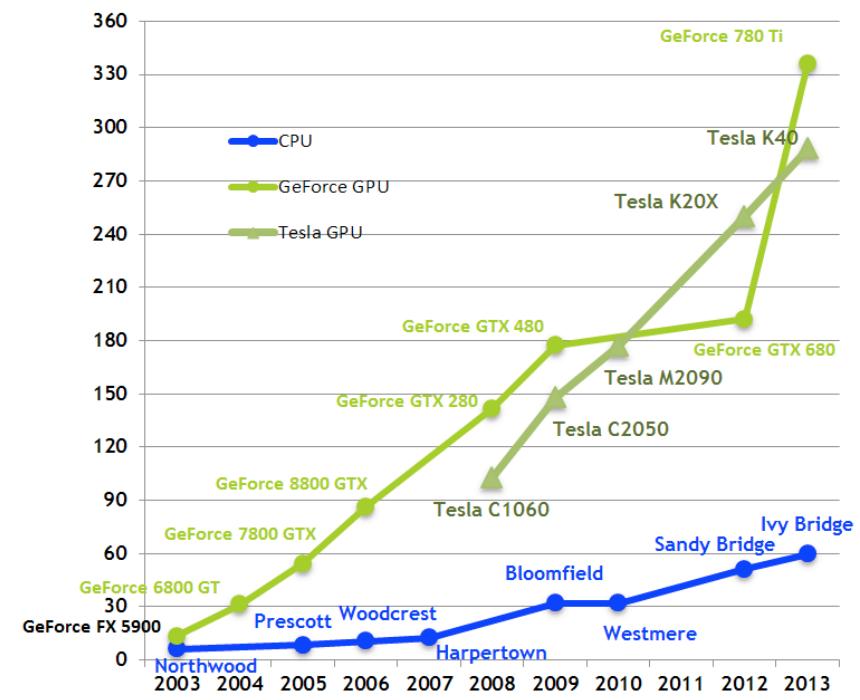
Grappes de calculs (cluster)

CPU vs. GPU : performances

Theoretical GFLOP/s

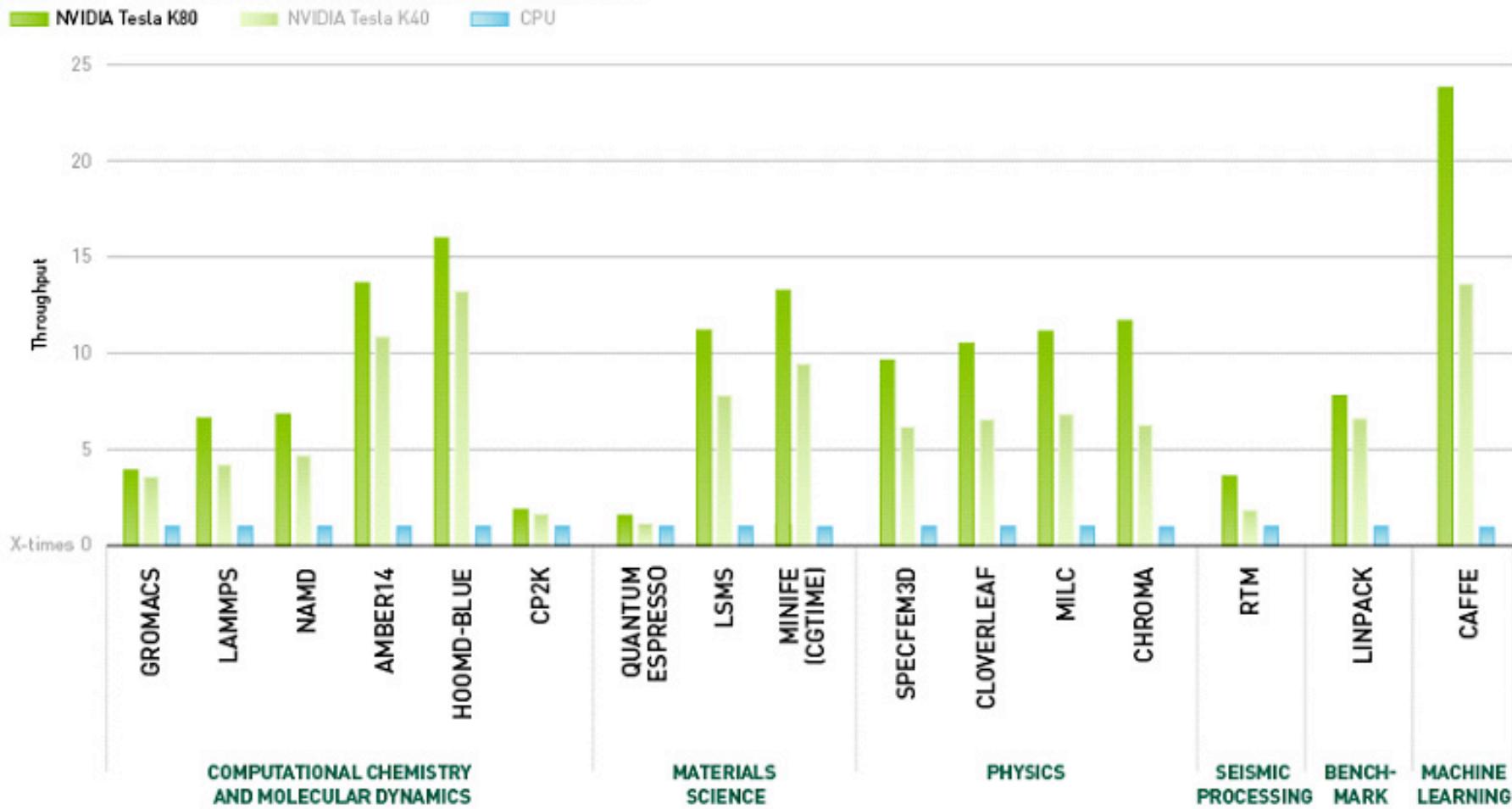


Theoretical GB/s



Calcul scientifique sur GPU

NVIDIA® TESLA® ACCELERATOR PERFORMANCE



GPU vs. Super-ordinateurs

Pour comparaison, les meilleurs clusters (« grappes ») de calcul mondiaux.

Octobre 2015 : #1 mondial = Tianhe-2, Chine :

- 3 120 000 cores (Intel Xeon E5-2692v2 12C 2.2GHz)
- 1 024 000 GB RAM
- Consommation 17.6 MW (centrale nucléaire : 860 MW)
- Linux bien sur! Programmation en Open-MP
- 54Pflops/s théoriquement, 34PFlops Linpack



A Lyon : CNRS/IN2P3



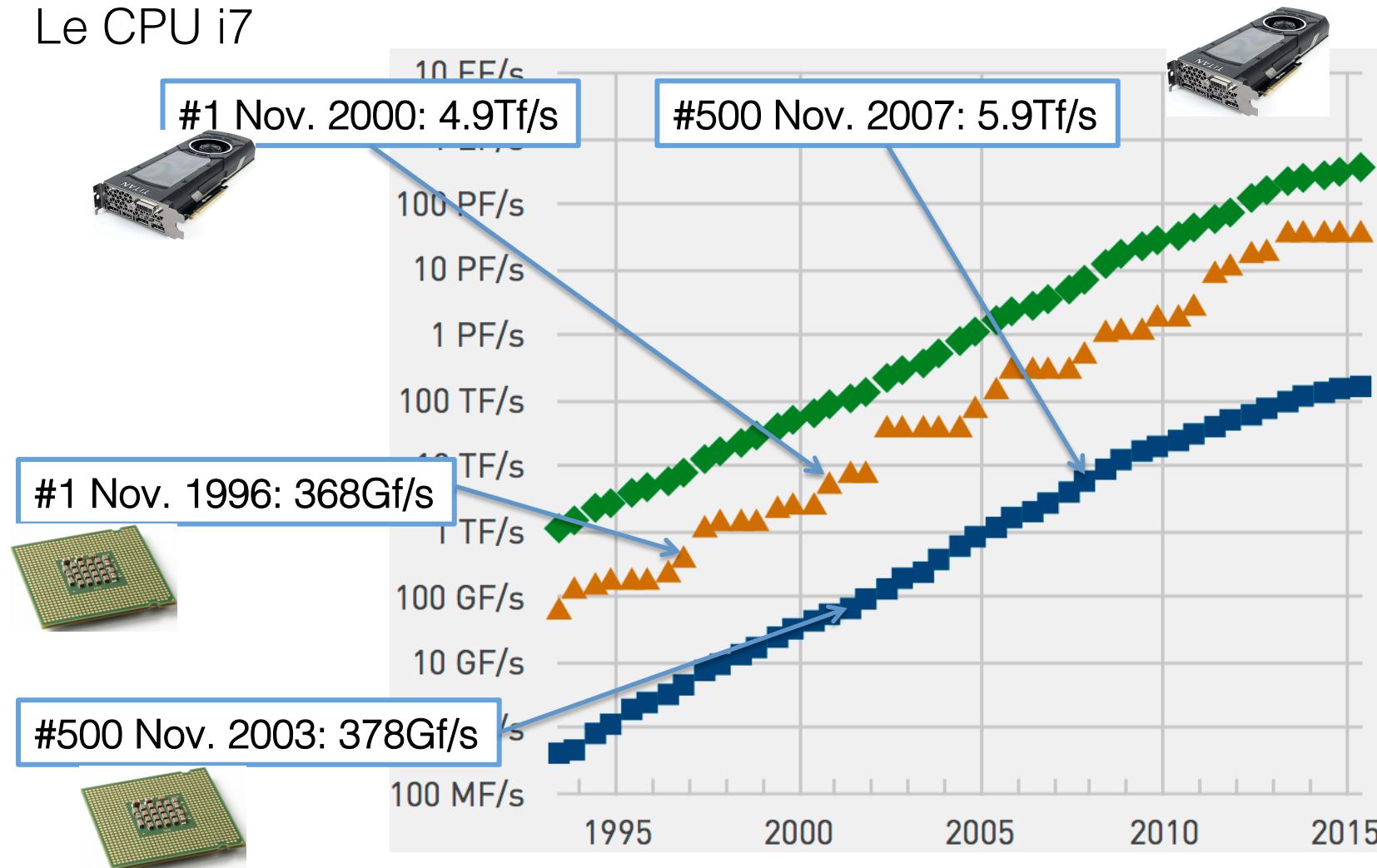
12000 cores (24 000 Hyperthreading)
0.5 Mwatt (calcul et refroidissement)

CPU vs. GPU vs. Super-ordinateur

Comparé avec la liste des 500 meilleurs clusters mondiaux :

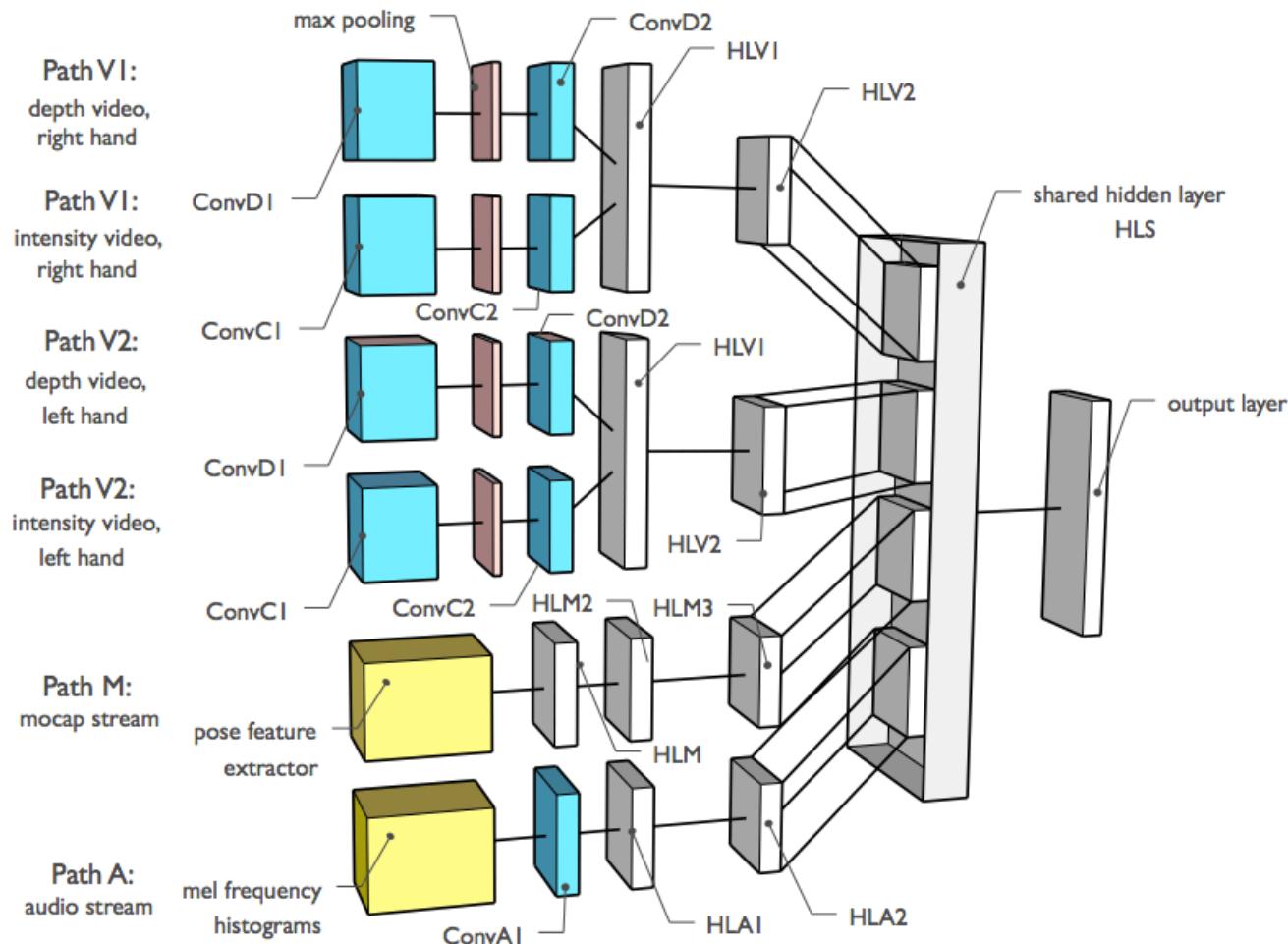
Le GPU Nvidia Titan X (Maxwell) : 6600 Gflops/s = 6.6Tflops/s

Le CPU i7



Application : reconnaissance de gestes

Réseau
de neurones
profond





NVIDIA® JETSON™ EMBEDDED PLATFORM
DEEP LEARNING FOR NEXT-GENERATION INTELLIGENT, AUTONOMOUS MACHINES



Le G du GPU : graphisme 3D

Problème d'origine : affichage d'un maillage 3D texturé à 60-100 frames par seconde.

Applications : jeux vidéo, logiciels CAO

- Grand nombre de triangles (résolution du maillage)
 $> 10\ 000\ 000 \dots$
- Grand nombre de pixels (résolution de l'écran)
 $4K = 3840 \times 2160 = 8\ 294\ 400$

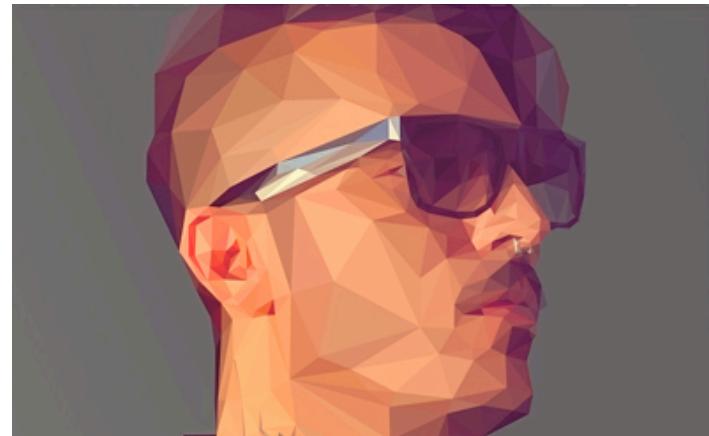


Figure : www.ukietech.com

Pipeline classique de rendu

Une pipeline simplifiée :

1. *Vertex Shading, Transform and Lighting*

Transformer les sommets du maillage, leur affecteur des données « intermédiaires » (couleur / ombrage / normales).

2. *Rastering*

Déterminer la couleur de chaque pixel d'un triangle à partir de la texture associée.

3. *Shading*

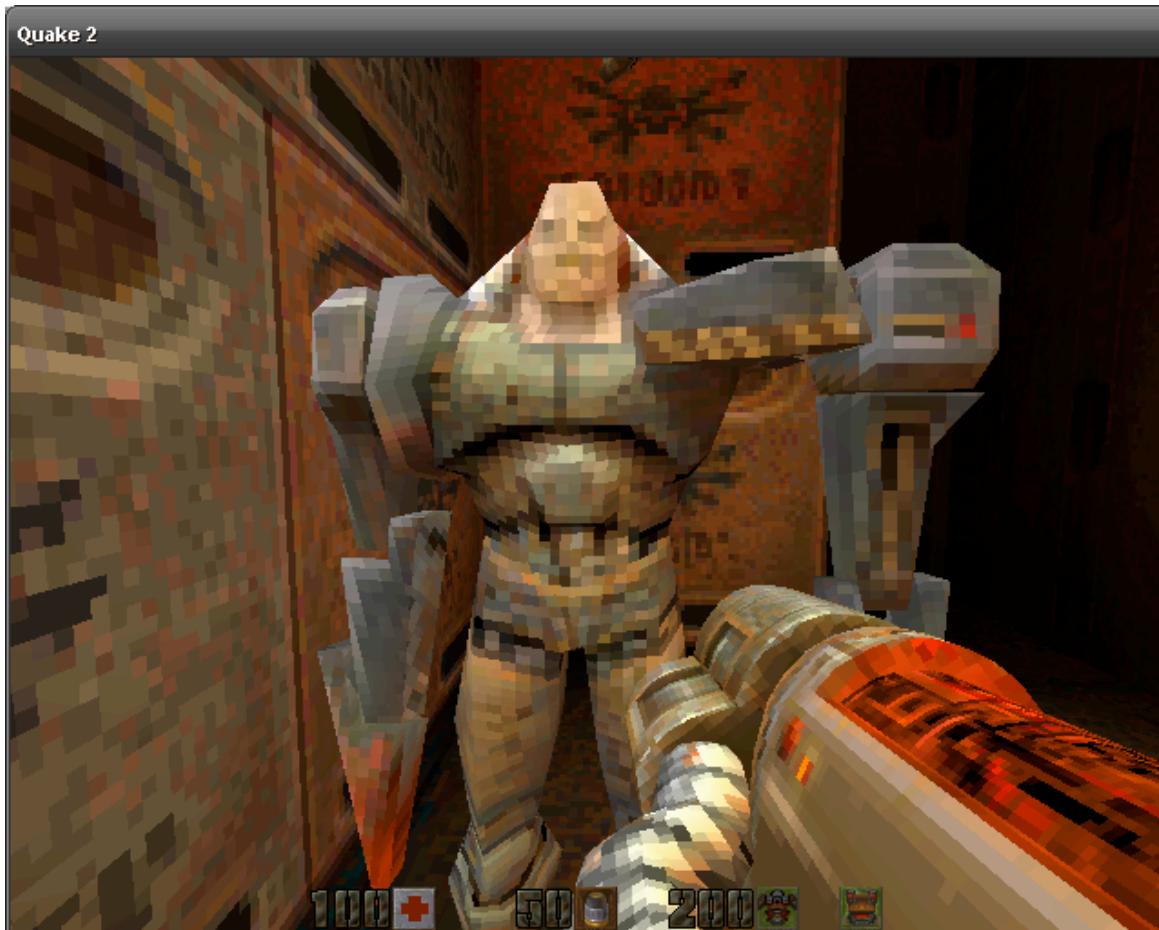
Déterminer la couleur finale en intégrant d'autres facteurs (ombrage)

4. *Frame buffering*

Ecriture du pixel en gérant la visibilité à l'aide d'un z-buffer

Histoire des GPU : (1) le néant

Avant ~1995 : graphisme 3d avant l'ère du GPU. Tout est calculé entièrement en logiciel et donc sur le CPU.



(Quake, ID Software)

Histoire des GPU : (2) 3Dfx

1996 : Introduction par *Diamond* de l'accélérateur 3D « *3Dfx Voodoo* ».

Accélération de certaines parties du pipeline.

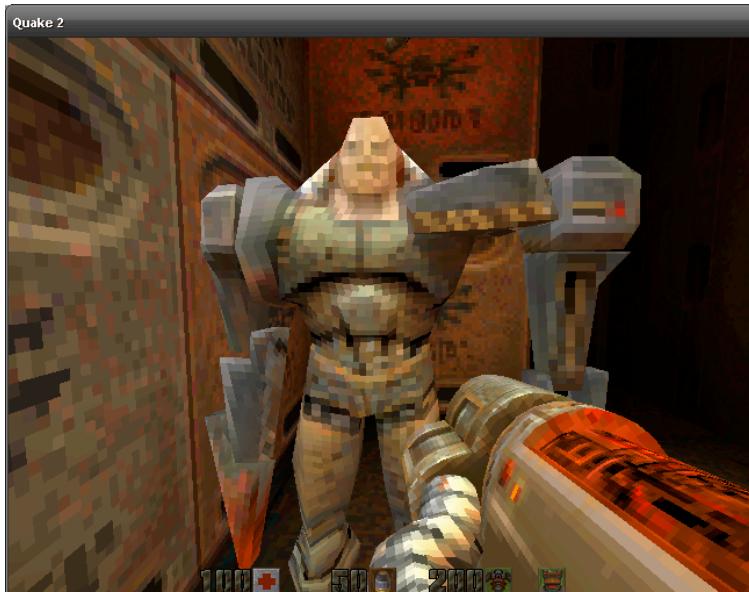
Il ne s'agit pas de la première carte. En revanche, elle prend rapidement 50%-80% du marché, tellement elle domine la concurrence en terme de performance.



Histoire des GPU : (3) GPU

1999 : introduction par Nvidia des cartes graphiques 3D dites « GPU ». Le pipeline de rendu (complet!) est entièrement réalisé en matériel et donc non-programmable (Nvidia GeForce 256).

- Augmentation de la résolution (de l'écran, des textures)
- Augmentation de la complexité des modèles 3D (nombre de triangles)



Quake 1 (CPU)



Quake 2 (GPU)

Histoire des GPU : (4) GPGPU

Pipeline programmable introduite avec Nvidia GeForce 3.
Pour calculer la valeur exacte d'un pixel (d'un vertex), on peut faire appel à une fonction programmable nommée « *shader* » (**pixel shader**, **vertex shader**).

Début de l'ère du « GPGPU » : *General Purpose Computing on Graphical Processing Units* ».

Premières utilisations pour le calcul scientifique (non prévues). Programmation très complexe.

Nécessite la conversion des données au format « *pixel* » ou « *vertex* ».

Histoire du calcul GPU : (5) CUDA

En Nov. 2006 Nvidia introduit la GeForce 8800 pour réagir sur la demande de la communauté HPC :

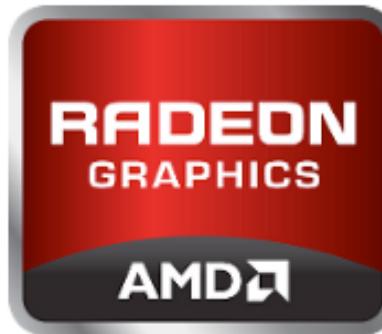
- « *GPU computing* » au lieu de « GPGPU ».
- API dédié au calcul scientifique à coté de l'API graphique
- Utilisation de C pour la programmation

Méthodologie : SIMT (*Single instruction multiple thread*)

GPU : les deux vendeurs



VS



- Programmation en CUDA (langage conçu par Nvidia)
 - Open-CL possible (pas optimal?)
 - Plus utilisé pour le calcul scientifique
 - Utilisé dans ce séminaire!
- CUDA non disponible
 - Programmation en Open-CL

GPU Nvidia : gammes

- **GeForce** : cartes graphiques pour usage général : jeux, infographie, calcul parallèle
- **Quadro** : cartes graphiques dédiées aux professionnels : création de contenu digital, CAO, DAO
- **Tesla** : cartes graphiques dédiées uniquement au calcul scientifique
- **Tegra** : liées aux systèmes mobiles (téléphones, tablettes)

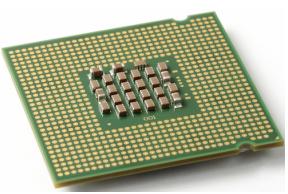
Cartes « Tesla »

- Spécifiquement conçues pour le calcul scientifique
- Pas de sortie vidéo! bonne chance pour vos jeux ...
- Double précision
- Plus de mémoire embarquée (12Go pour certains)
- Possibilité de communication « *Infiniband* » pour connecter des GPU embarqués sur différents ordinateurs.



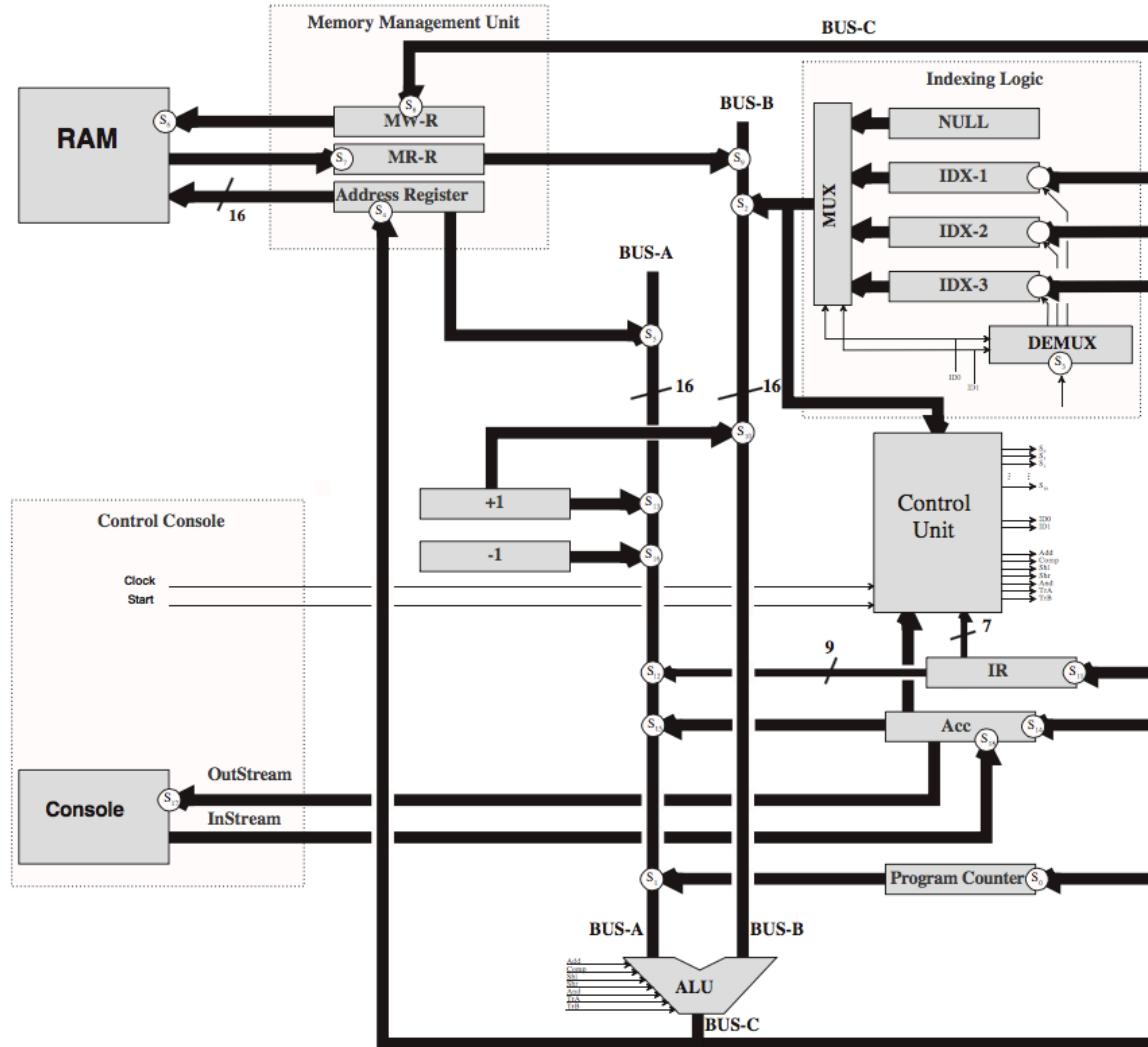
Générations de GPU Nvidia

- G80/Tesla (à ne pas confondre avec les cartes Tesla)
- Fermi (2010-)
- Kepler (2012-)
- Maxwell (2014-)
 - GeForce Titan X,
 - GeForce 980Ti
 - GeForce 750 (utilisé en TP)
 - Tesla M40 (et variantes)
- Pascal (depuis juillet 2016)
 - GeForce Titan X Pascal
 - GeForce 1080
 - Tesla P40 (et variantes)



Rappel : architecture d'un CPU

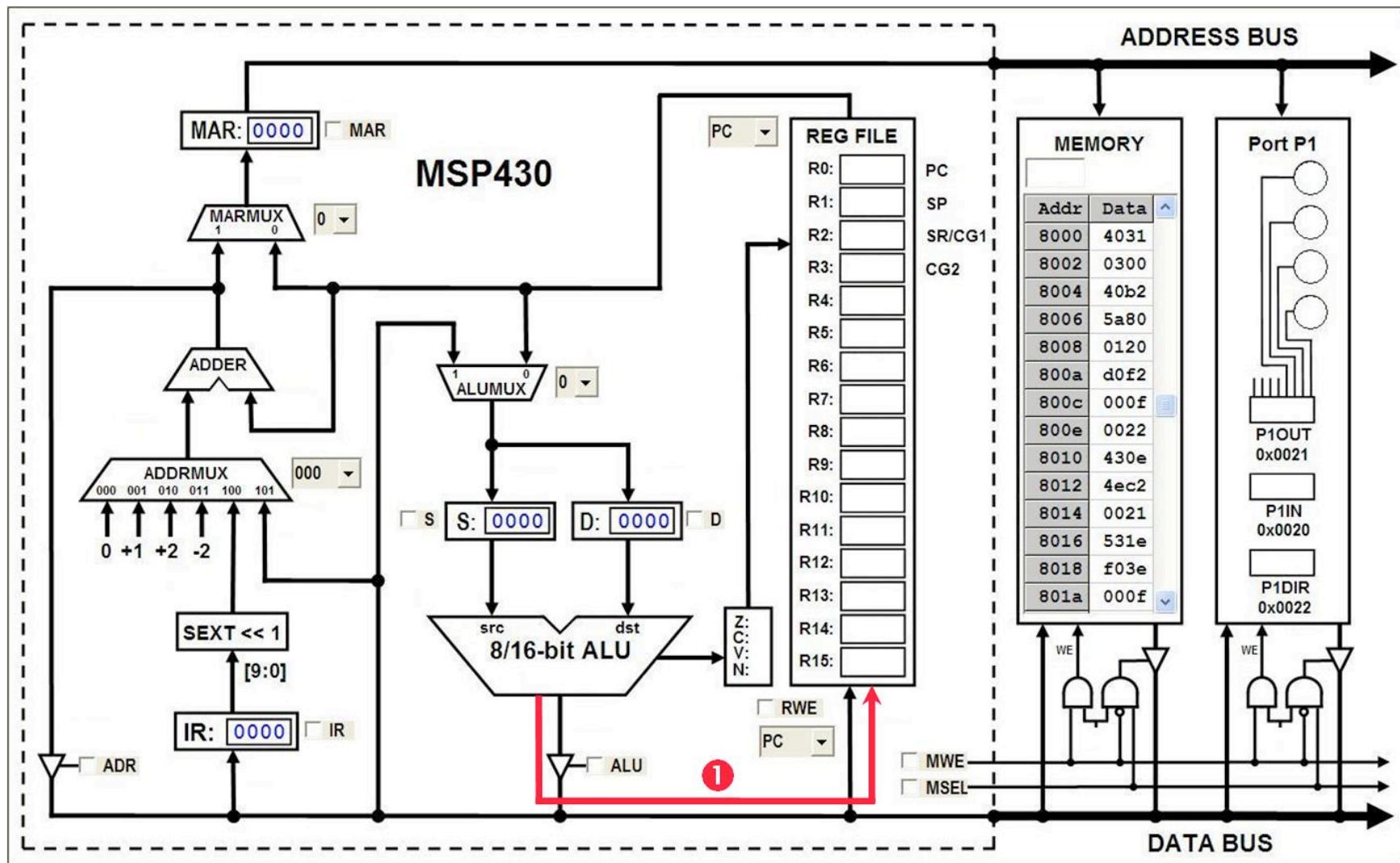
Rappel : micromachine, INSA-3IF





Rappel : architecture d'un CPU

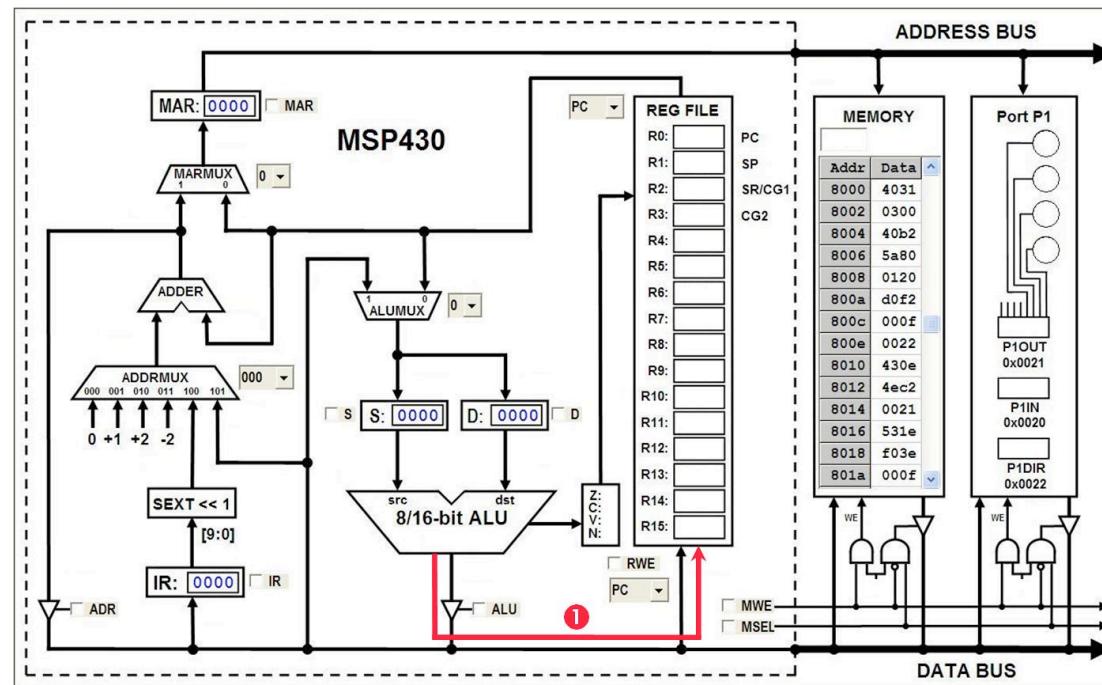
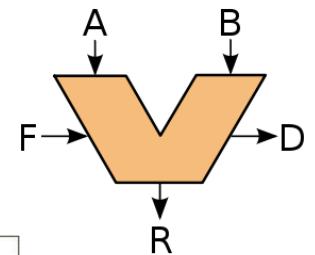
Intel MSP 430 (vu en 3IF)





CPU : Le cycle de von Neuman

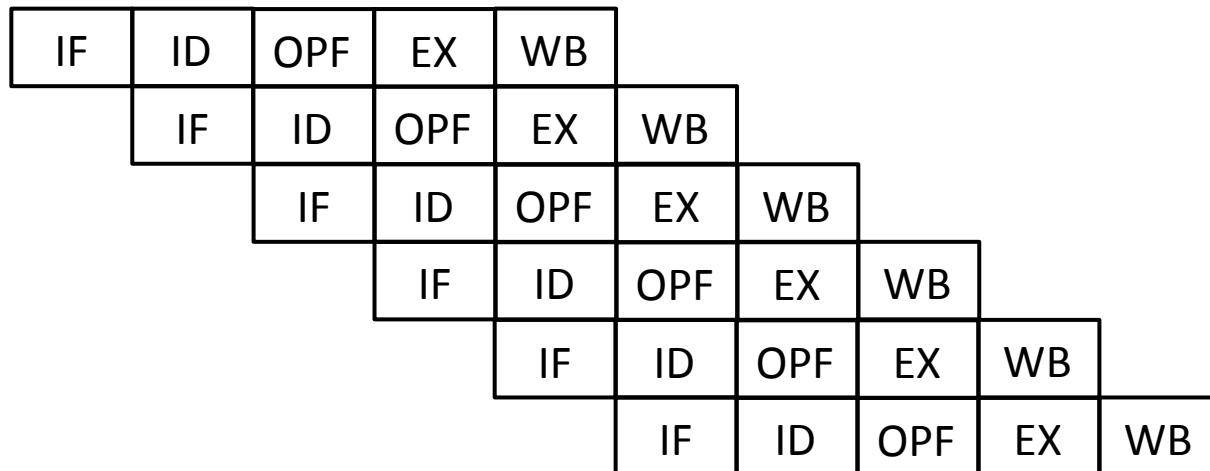
- Etapes du cycle de von Neumann :
 - IF instruction fetch
 - ID instruction decode
 - OPF Operand fetch
 - EX execute
 - WB result write back

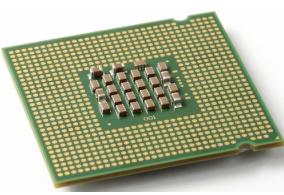




Pipeline d'un CPU

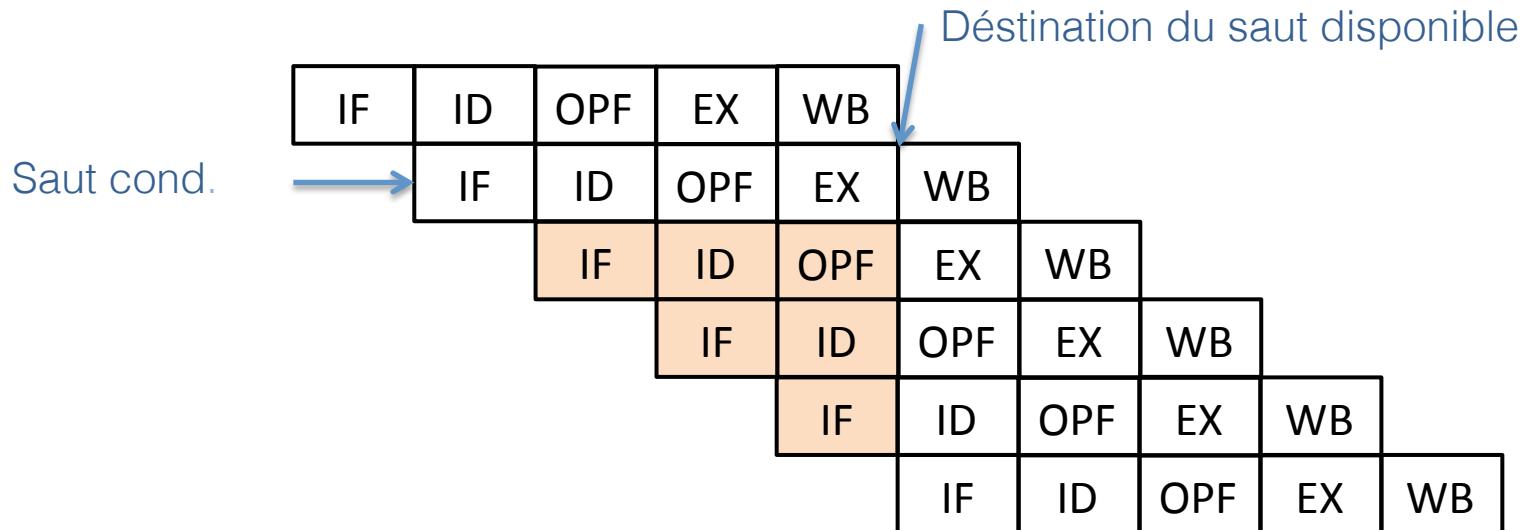
- Paralléliser l'exécution des cycles de von Neumann
 - Permet de terminer une nouvelle instruction à chaque tic d'horloge, si
 - Pas de dépendances entre instructions consécutives
 - Opérandes disponibles
 - Sinon : attentes (« bulles »)



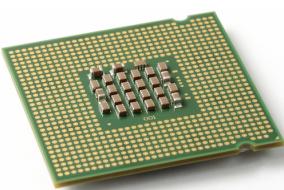


Prédiction de branchement

- Pour les instructions de type saut conditionnel, le CPU essaie de deviner le résultat du saut (« *branch prediction* »)
- Les instructions suivant l'instruction en cours sont exécutées de manière spéculative
- Si la prédiction est fausse, les résultats sont supprimés et la pipeline vidée

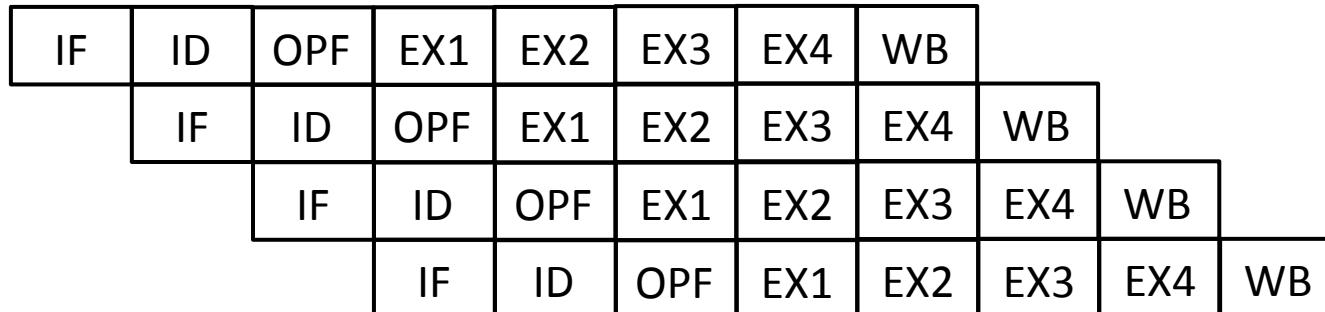


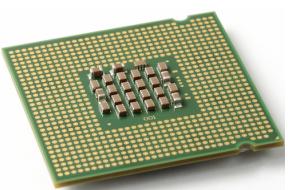
Pas réalisé sur les GPU!!



Fréquence de l'horloge

- Augmenter la fréquence de l'horloge peut augmenter la performance
- Nécessite une maîtrise de la complexité de chaque étape
- Création de *pipelines* très longues (20 cycles et plus)
- Toujours une instruction par *clock* et donc une meilleure performance ... dans le meilleur cas!
- Problème : consommation

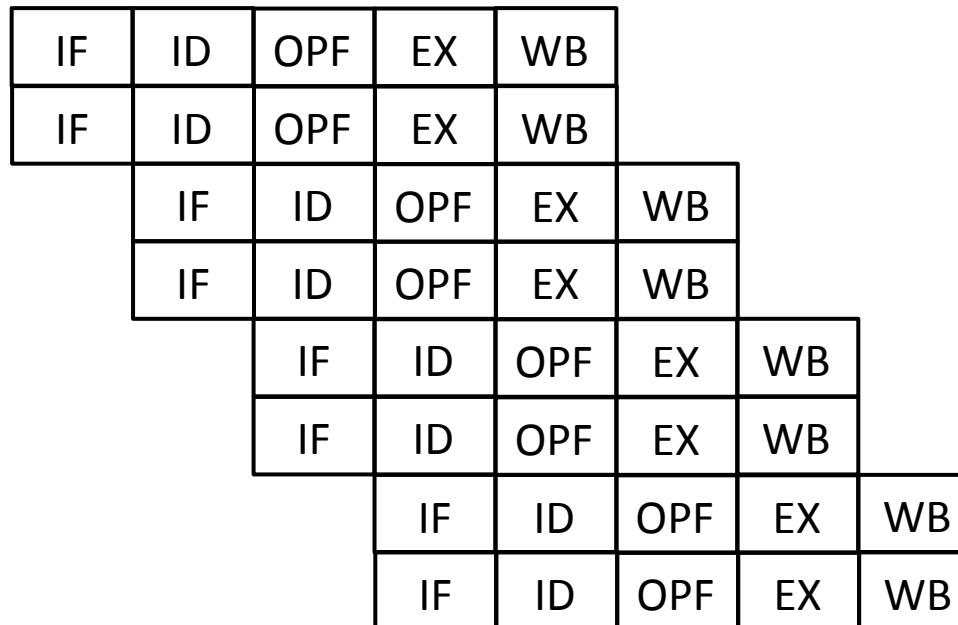


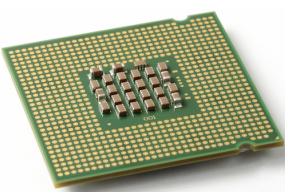


CPU : architecture super-scalaire

Dans les meilleures conditions, permet de terminer plus d'une instruction à chaque tic d'horloge.

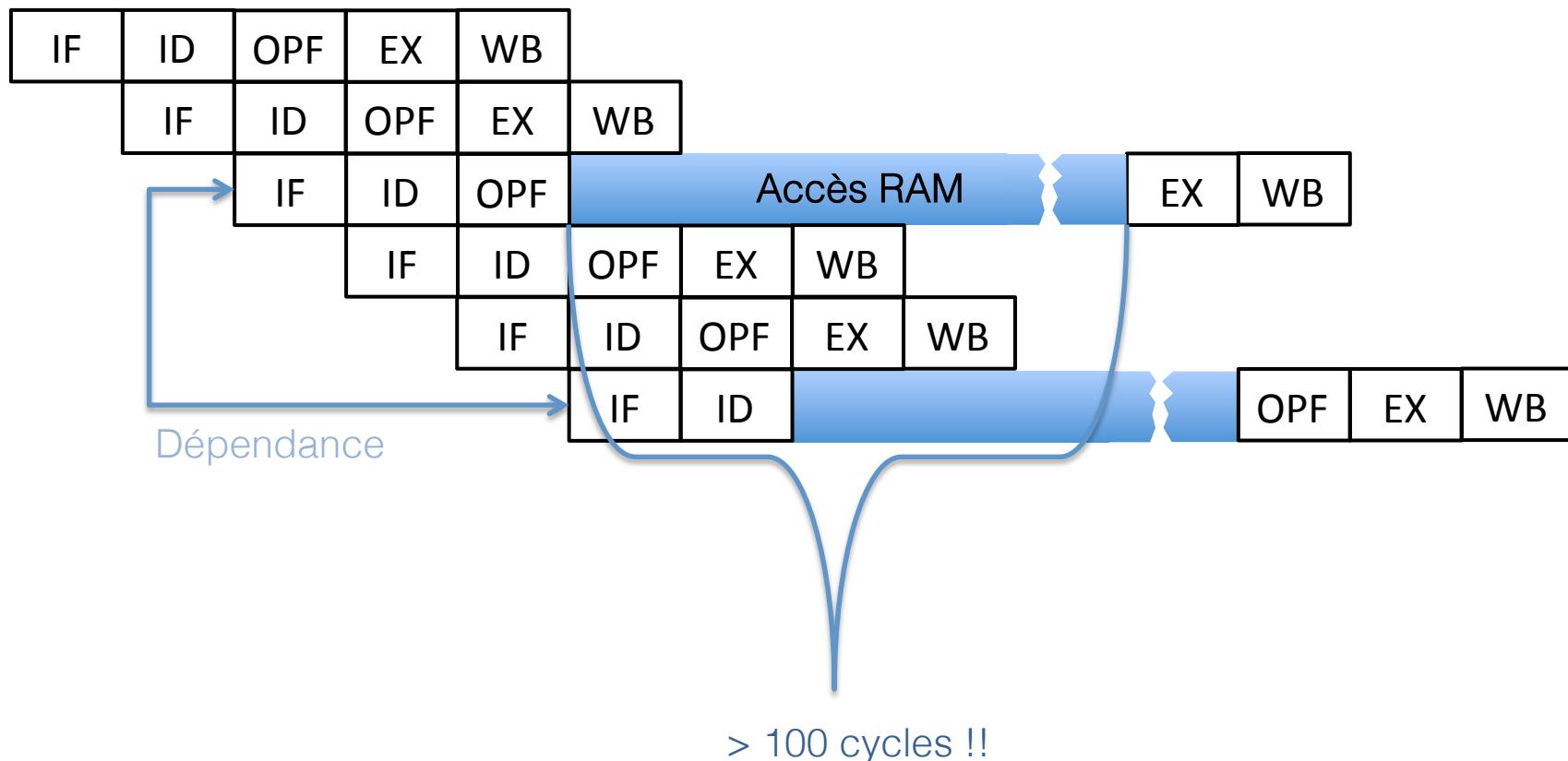
Duplication des unités de calcul au sein du CPU.





CPU : Cache! Cache! Cache!

Un *cache miss* met en attente toute la pipeline

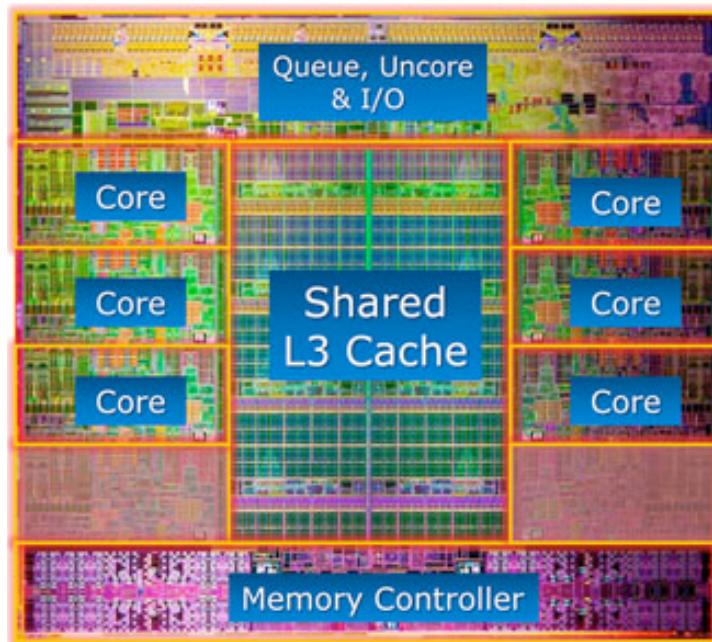




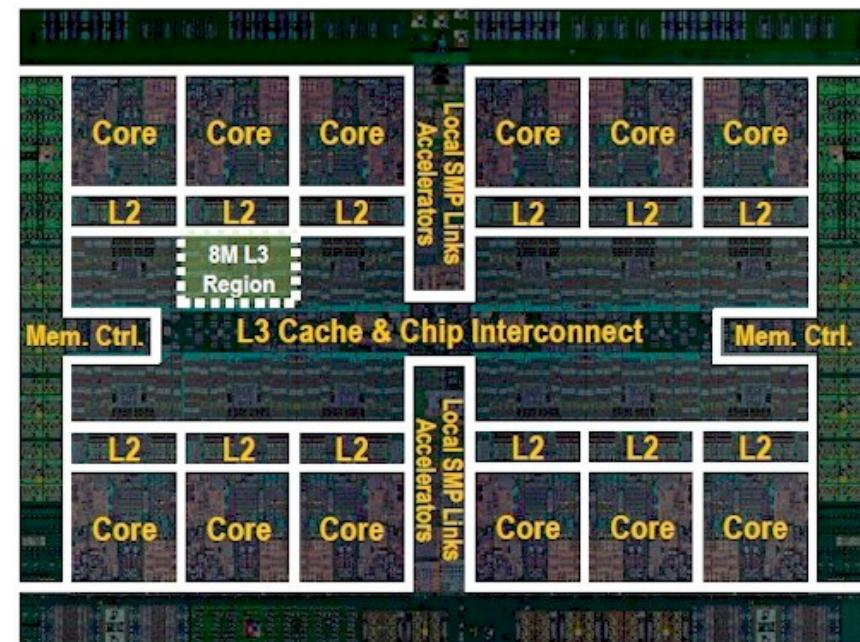
CPU : Cache! Cache! Cache!

Une très très grande partie des transistors d'un CPU moderne est consacrée aux caches (L1, L2, L3, L4?)

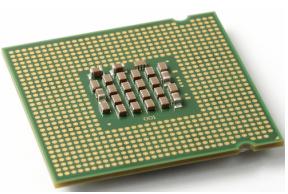
Raison : optimisation pure de la performance d'un seul *thread*



Intel i7



IBM Power 8 (2014)



CPU : Cache! Cache! Cache!

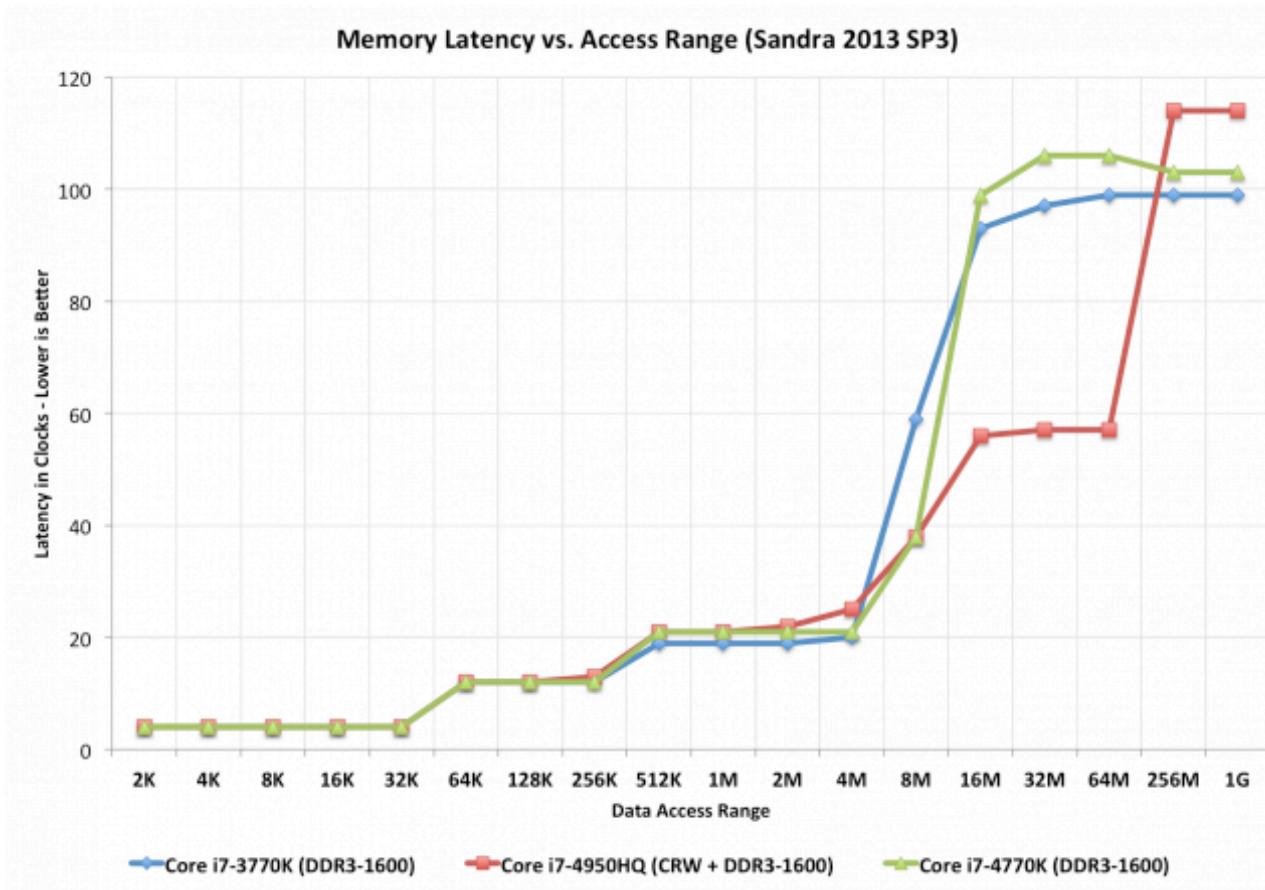
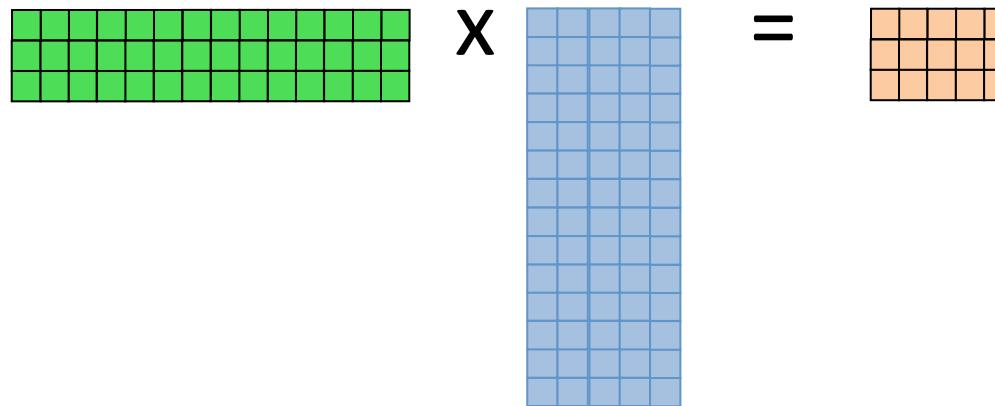


Figure : AnandTech

Stratégies CPU vs. GPU

- CPU
 - Optimiser la performance « séquentielle » pure
 - Diminuer / maîtriser les latences d'accès à la mémoire (Cache!!)
 - Logique sophistiquée pour résoudre des dépendances (prédiction de branchements, exécution *out of order* etc.)
- GPU :
 - On suppose l'existence d'un TRES grand nombre de threads
 - Cacher des latences d'accès à la mémoire (exécuter les instructions pour lesquelles les données sont disponibles)
 - Optimiser la performance par watt

Exemple : multiplication de matrices



La solution séquentielle classique :

```
1   for( int r = 0; r < h; r++)
2   {
3       for( int c = 0; c < w; c++)
4       {
5           m1xm2[r*w + c] = 0;
6
7           for( int k = 0; k < w; k++)
8               m1xm2[r*w + c] += m1[r*w + k] * m2[k*w + c];
9       }
10 }
```

Solution parallèle

Exécution en parallèle d'une fonction ayant comme paramètre l'indice du résultat à calculer :

```
1 void mult_kernel_simple(int c, int r)
2 {
3     output[r*mxWidth + c] = 0.0f;
4     for( int k = 0; k < mxWidth; k++)
5         output[r*mxWidth + c] += mx1[r*mxWidth + k] * mx2[k*mxWidth + c];
6 }
```



Appeler en parallèle pour tous c, r

Sur CPU avec un seul cœur et en cas de *threads* multiples, est-ce que les latences dues aux accès RAM seront cachées?



CPU : changement de contexte

Réponse : NON

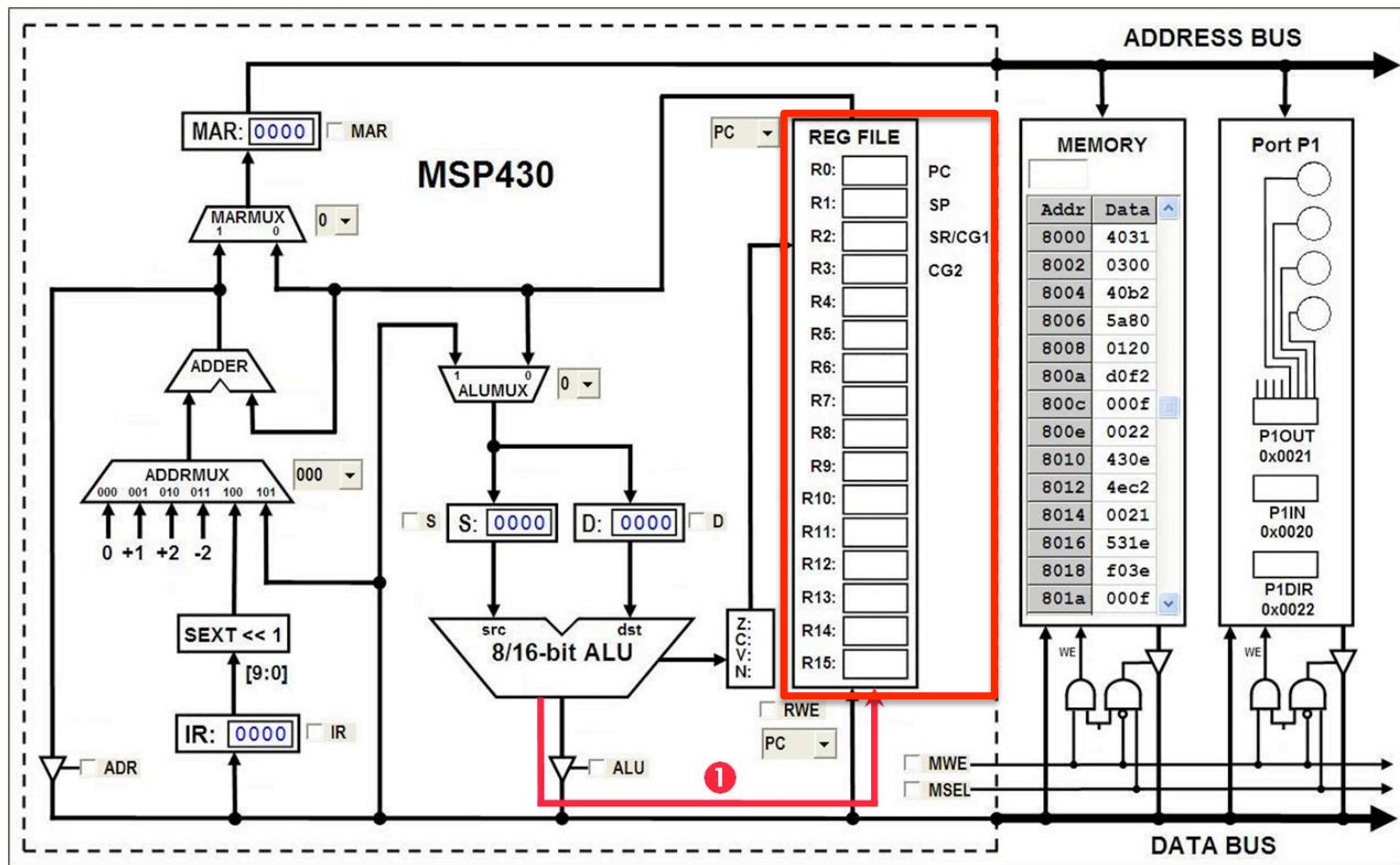
- Un changement de thread sur un CPU demande un changement de « contexte », une opération chère :
- Les registres sont disponibles une seule fois dans le processeur :
 - Program counter (PC)
 - Instruction register (IR)
 - Stack pointer (SP)
 - Status register (SR)
 - Registres de calcul (par exemple AX, BX, CX ... pour x86).
- Basculer d'un processus ou *thread* à un autre nécessite :
 - Stockage des registres de l'ancien *thread*
 - Chargement des registres du nouveau *thread*

La solution GPU

- Stockage d'un très grand nombre de *threads* en parallèle directement sur le processeur
- Le GPU peut choisir un *thread* sans aucun délai (0 cycles!)
- A chaque tic d'horloge une instruction est choisie dont toutes les dépendances ont été satisfaites (disponibilité des opérandes etc.)
- Pour chaque registre (PC, IR, SP ...), une **copie (!!)** est réalisée et cablée pour chaque *thread* (et non pas par unité de calcul!)
- Génération Maxwell (e.g. Titan X, GTX980) :
 - 2048 threads par « streaming-multiprocessor »
 - 16 SM sur un GPU : 32768 réalisations de chaque registre !!

Register file

Rappel : les registres du CPU Intel MSP430 sont numérotés et organisés sous forme d'un « *register file* »



Register file

Registerfile d'un GPU : copies multiples de chaque registre.
Une copie pour chaque thread « démarré », dont un grand nombre qui ne sont pas exécutés en un instant donné!

| REG FILE |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| R0: |
| R1: |
| R2: |
| R3: |
| R4: |
| R5: |
| R6: |
| R7: |
| R8: |
| R9: |
| R10: |
| R11: |
| R12: |
| R13: |
| R14: |
| R15: |

Nvidia : allocation dynamique du nombre de registres par thread (jusqu'à 63) en fonction de l'occupation du GPU.

CUDA :
modèle de programmation

Différents niveau d'abstraction

Programmation de haut niveau, spécialisée. Exemple :
Tensorflow, de Google (mathématiques, *machine learning*)

```
with tf.device('/gpu:0'):
    #Sigmoidal layer (9x9) 4096->4096
    h_conv8 = cf.conv2d_bias(h_conv7, [9,9,2048,1])
    #Softmax + CE
    # We first reshape
    h_vect = tf.reshape(h_conv8, [-1, 120*160*1])

    keep_prob = tf.placeholder(tf.float32, name='keep_prob')
    h_drop = tf.nn.dropout(h_vect, keep_prob)

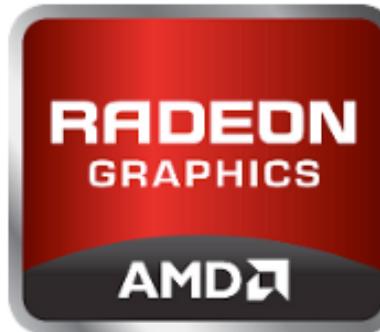
    v = tf.reshape(tf.nn.softmax(h_drop), [-1, 120, 160, 1], name='y')
with tf.device('/cpu:0'):
    error = cr.calculate_error_tf(y, y_, BATCH_SIZE, name='error')
with tf.device('/gpu:1'):
    gt_flat = tf.reshape(y_, [-1, 120*160*1])
```

Programmation de bas niveau. Accès direct à l'API Nvidia.
Exemples : CUDA (traité en séminaire), Open-CL

Programmation bas niveau



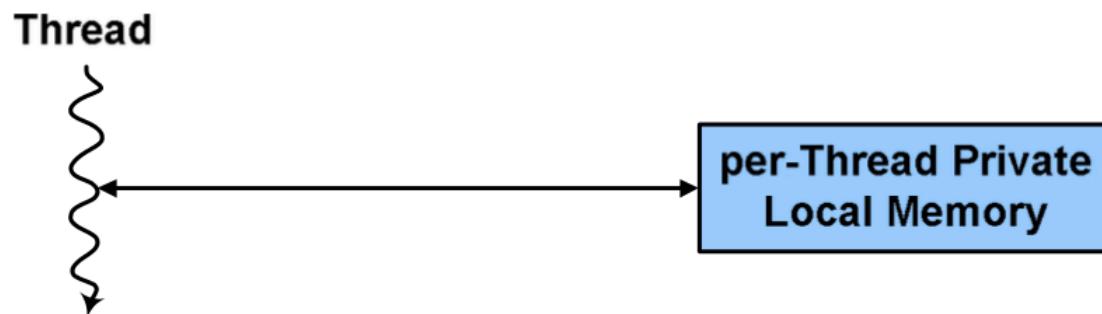
VS



- Programmation en CUDA
(langage conçu par Nvidia)
 - Open-CL possible (pas optimal?)
 - Plus utilisé pour le calcul scientifique
 - Utilisé dans ce séminaire!
- CUDA non disponible
 - Programmation en Open-CL

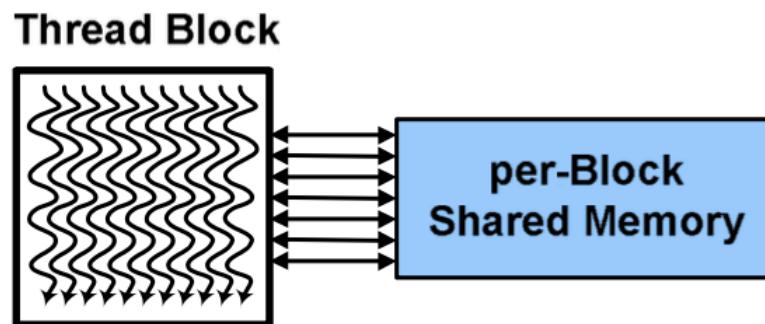
Hiérarchie « logicielle » : thread

- Une instance d'un *kernel*
- Si « actif », dispose d'un PC (*program counter*), registres, mémoire privée, entrées et sorties
- Une ID propre dans un *block*



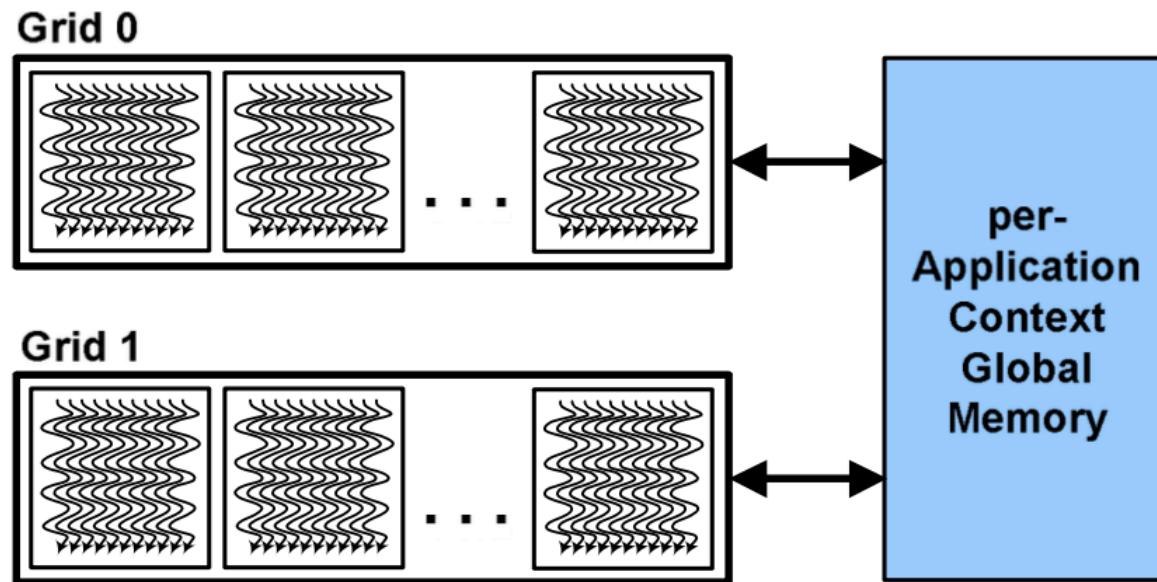
Hiérarchie « logicielle » : bloc

- Un ensemble de *threads* pouvant coopérer :
 - Synchronisation
 - Mémoire partagée
 - Une ID propre dans une *grid*



Hiérarchie « logicielle » : *grid*

- Tableau de *block* exécutant le même *kernel*
- Peuvent accéder à la mémoire globale du GPU
- Synchronisation uniquement en terminant le *kernel* actuel et puis en démarrant un nouveau *kernel*.

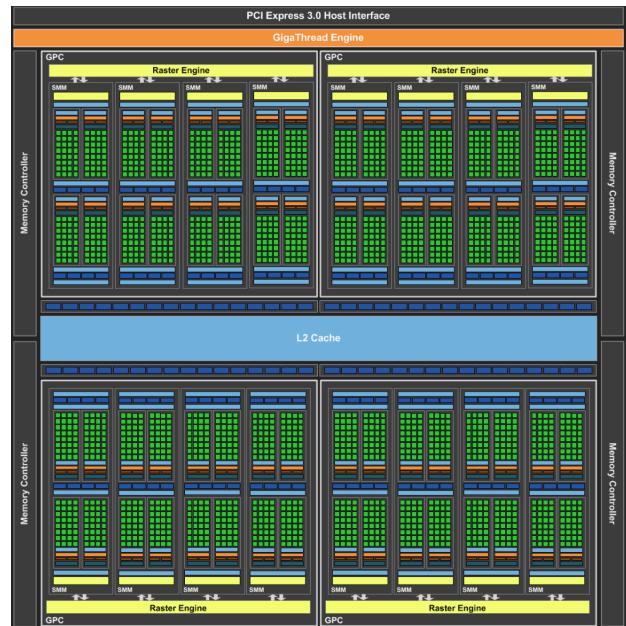
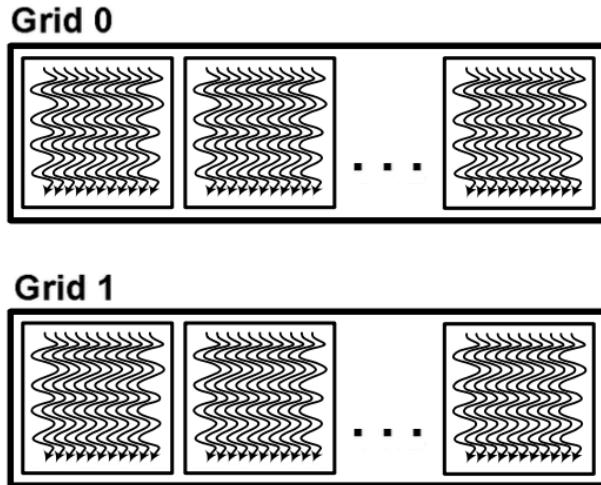


CUDA vs. GPU

mapping logiciel - matériel

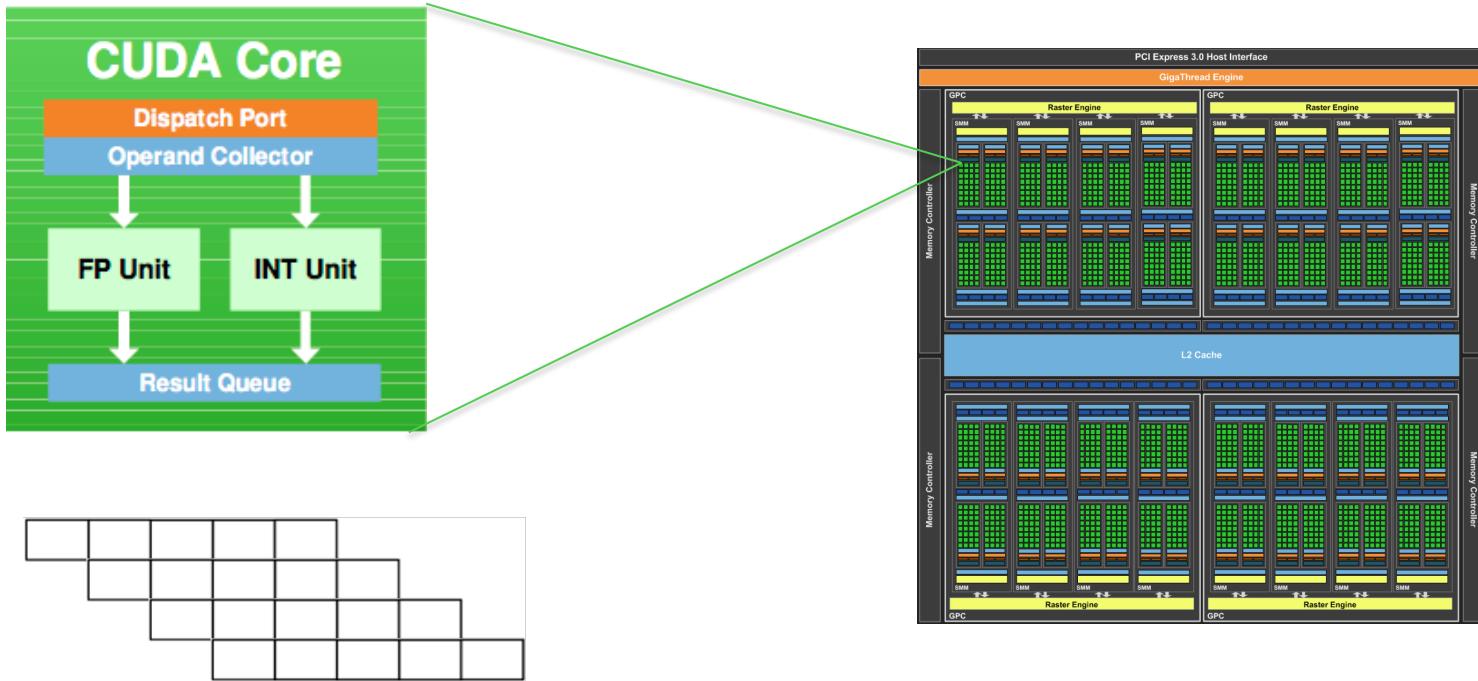
Mapping logiciel - matériel

La structuration *thread/block/grid* est propre au modèle de programmation « CUDA ». Elle est mappé sur une hiérarchie « matériel » sur le GPU. Le mapping est transparent au programmeur! De manière générale, le nombre de *threads* n'est **pas** égal au nombre de cœurs!



Un « cuda core »

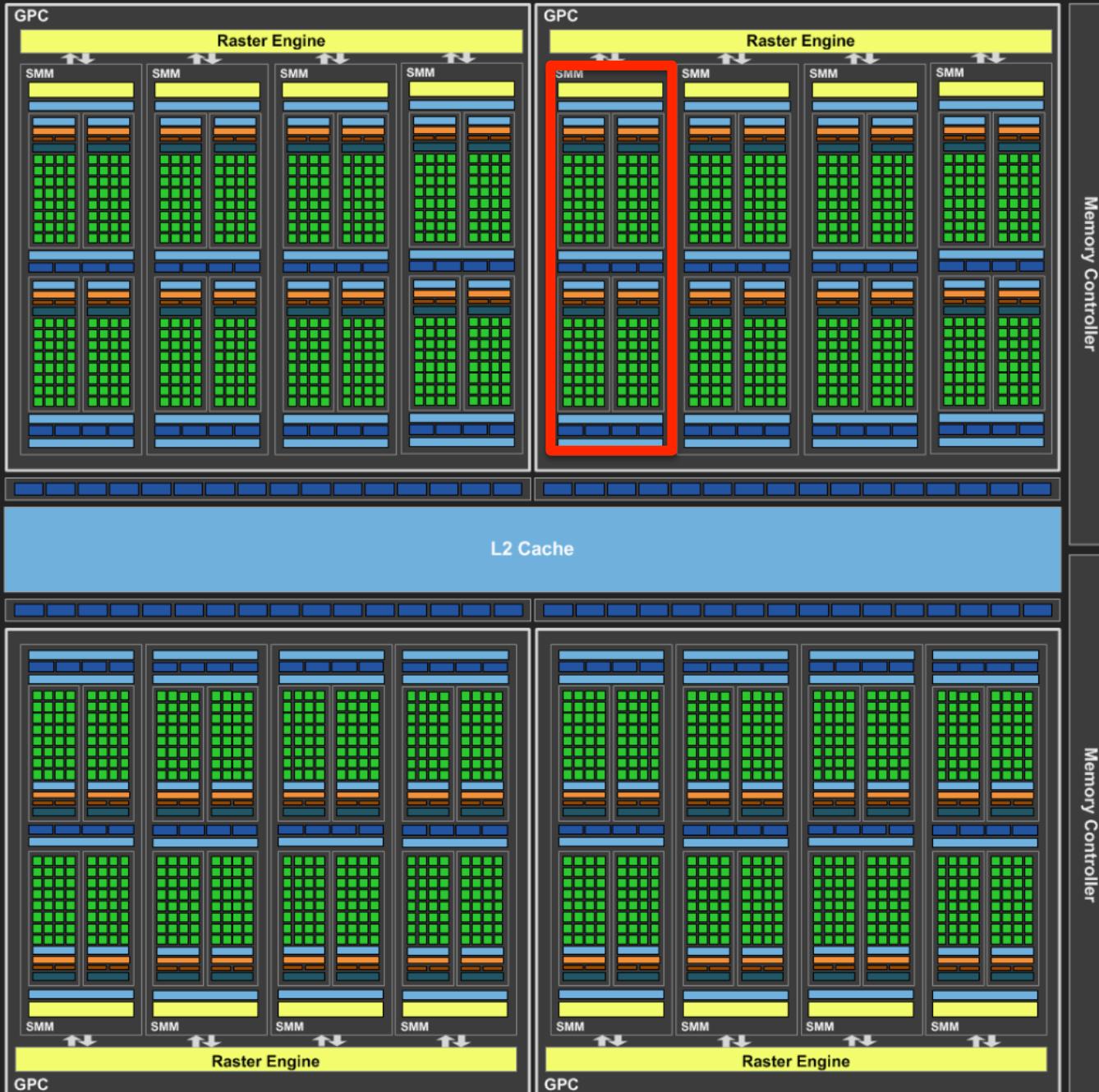
- Une ALU « integer » + une ALU « float »
- Chacune est équipée d'une *pipeline* (sans la partie initiale IF, ID ...)



Longeur du pipeline? ... On ne sait pas.

Nvidia : « *The latency of read-after-write dependencies is approximately 24 cycles, but this latency is completely hidden on multiprocessors that have sufficient warps of threads concurrent per multiprocessor.* »

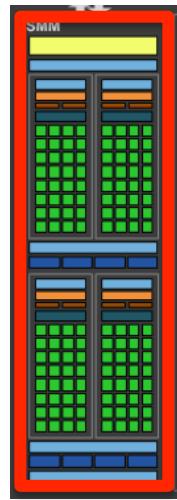
GigaThread Engine



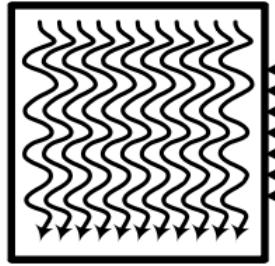
Maxwell (GTX 980)

16 « Streaming multi-processors » (SM)

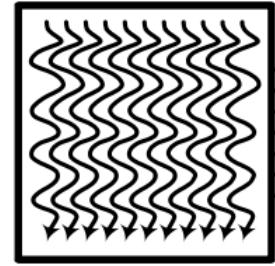
Les SM sont assez indépendants.



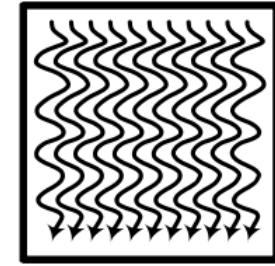
Thread Block



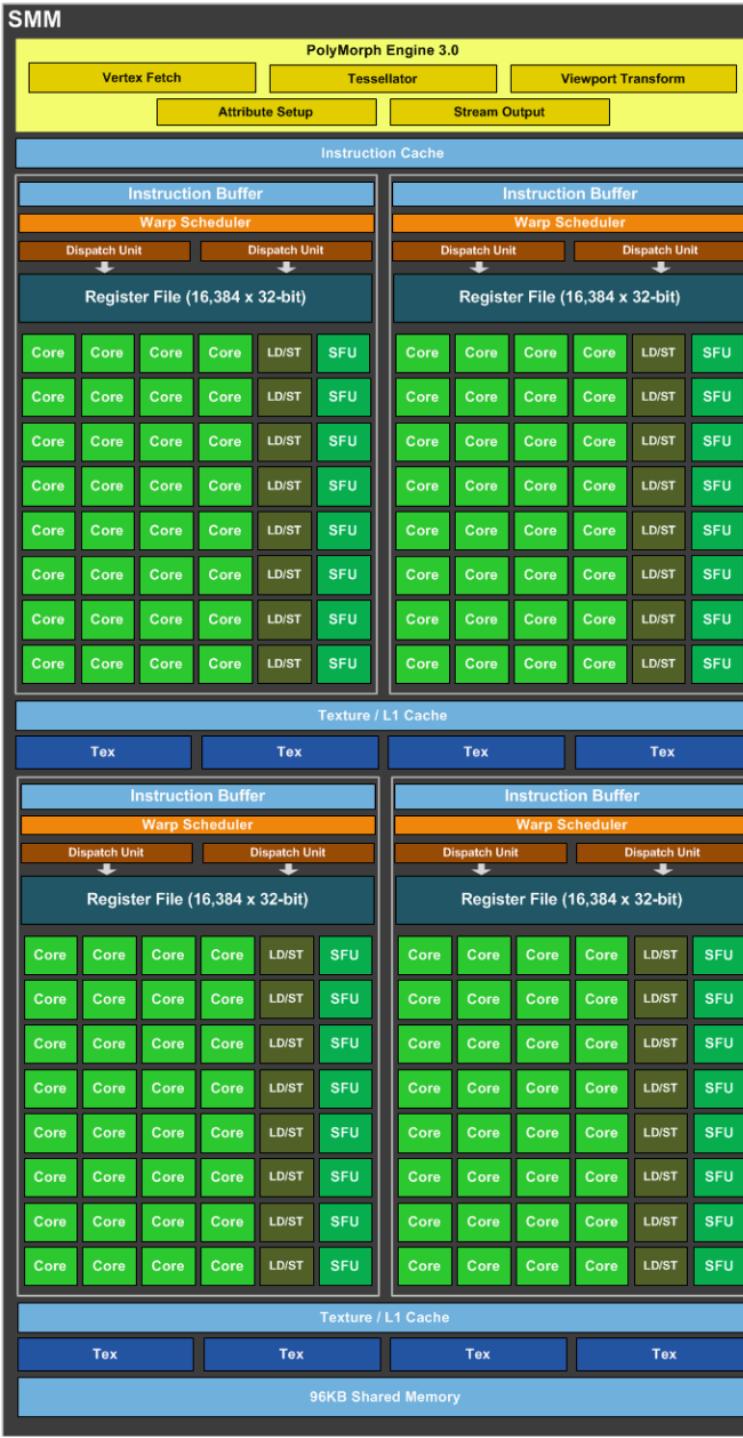
Thread Block



Thread Block



Un *thread block* est affecté à un SM, et il ne le quittera plus
Plusieurs *blocks* peuvent être affectés au même SM

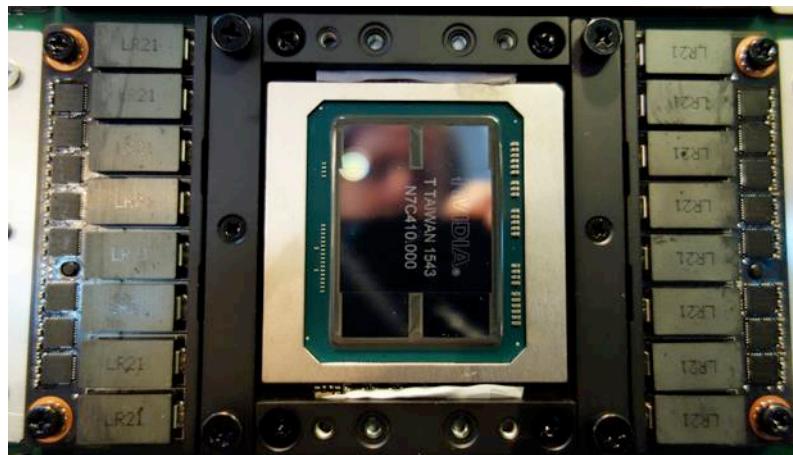


Première Génération « Maxwell » :

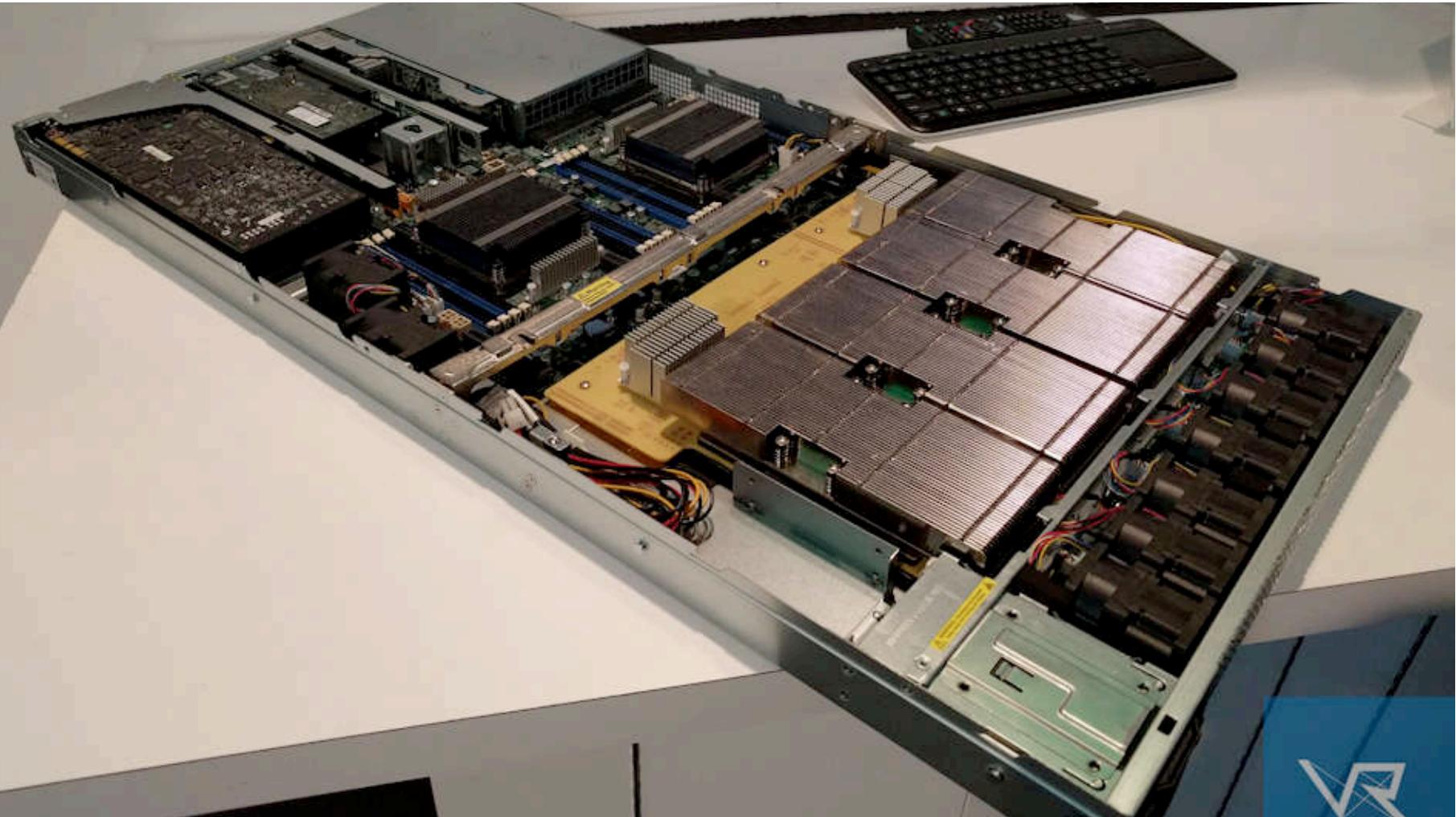
- 128 cœurs par SM, groupés en 4 blocs de 32
 - Avec 16 SM = 2048 cœurs max par GPU
 - Chaque cœur contient une pipeline complète

Deuxième génération « Maxwell » :
> 3000 cœurs ...

Dernière génération : Pascal



NVIDIA DGX-1



Warp drive

- Avec ~4000 Cuda cores (génération Pascal), on pourrait avoir ~4000 threads exécutés à chaque cycle.
- Problème : comment fournir les données aux threads, si la bande passante est limitée?
- Comment fournir ~ 4000 *instructions différentes* par cycle?
- Réponse : on regroupe les threads par *warp* exécutant la même instruction.

Warp drive

- Chaque SM organise les *thread* en groups de 32 appelés « *warp* » (*warp* <> *block* !!)
- Les *threads* d'un *warp* sont exécutés en parallèle
- La même instruction est exécutée par tous les threads, généralement sur des données différentes
- SIMT = *Single Instruction Multiple Data*
- Exemple de la multiplication de matrices :

```
1 void mult_kernel_simple(int c, int r)
2 {
3     output[r*mxWidth + c] = 0.0f;
4     for( int k = 0; k < mxWidth; k++)
5         output[r*mxWidth + c] += mx1[r*mxWidth + k] * mx2[k*mxWidth + c];
6 }
```

- Chaque thread du warp exécute une multiplication
- Les opérandes sont différents

Divergence des branches

- En cas de branchements, les threads d'un *warp* peuvent prendre des chemins différents
- Tous les chemins seront exécutés séquentiellement ensemble par le *warp* entier
- Les threads du *warp* n'ayant pas pris le chemin en cours sont désactivés

```
1 if (a>0)
2 {
3     instruction-a;
4     if (b>0)
5         instruction-b;
6     else
7         instruction-c;
8 }
```

- Cela concerne uniquement les *threads* du même *warp*.

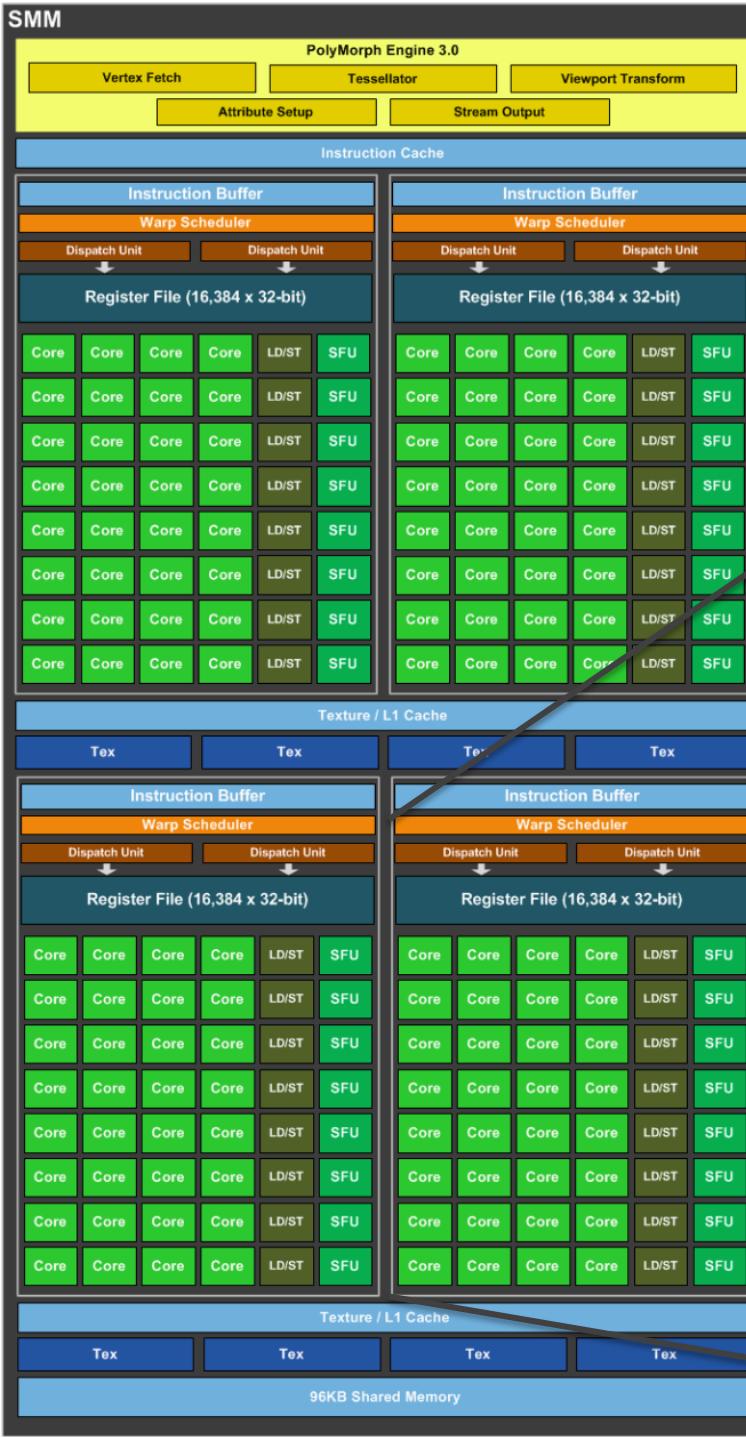
Divergence des branches

Dans le pire des cas, 32 chemins différents seront exécutés séquentiellement et ensemble par les 32 *threads* du *warp*.

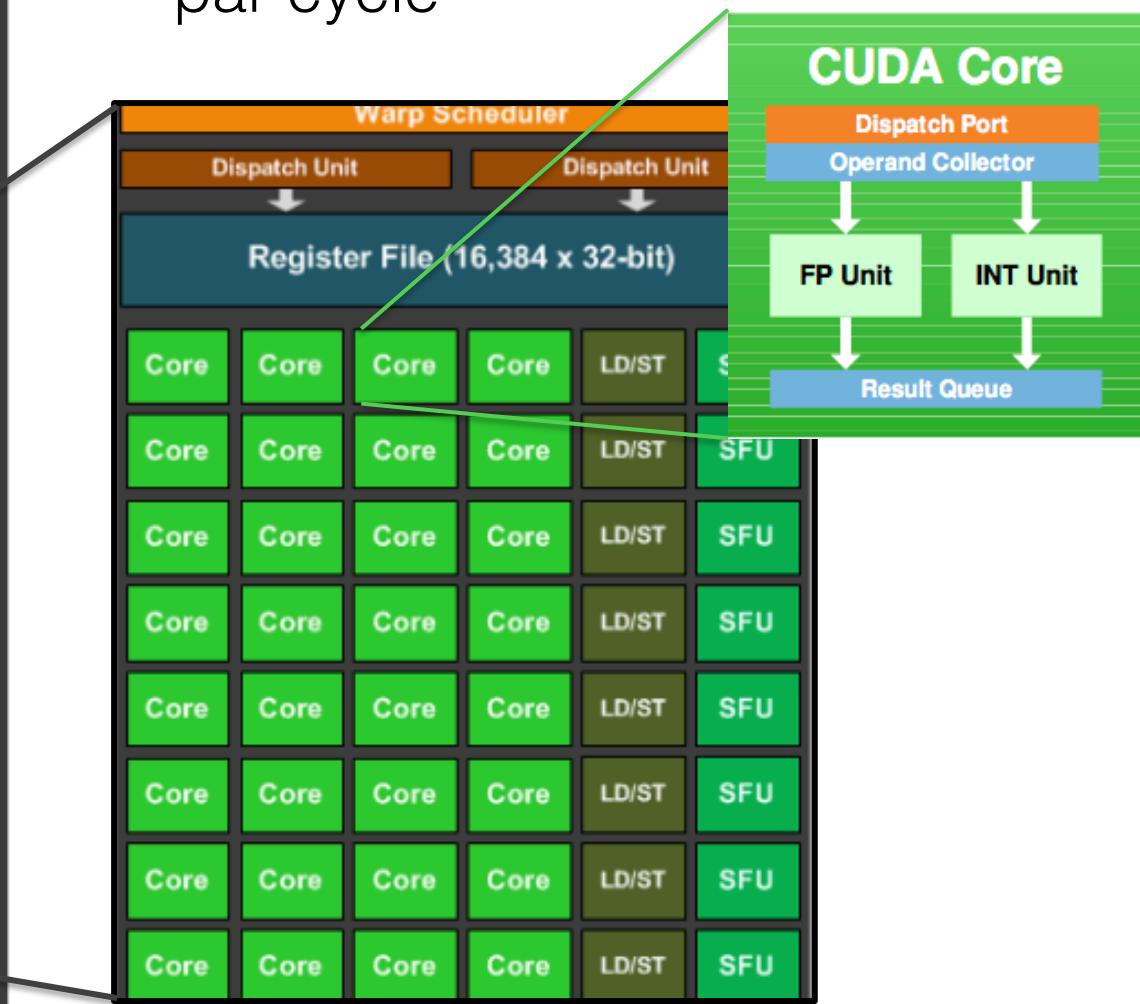
```
1 switch (c) {  
2     case 0: instruction-a; break;  
3     case 1: instruction-b; break;  
4     case 2: instruction-c; break;  
5     case 3: instruction-d; break;  
6     case 4: instruction-e; break;  
7     case 5: instruction-f; break;  
8     ...  
9     ...  
10 }
```

SIMD or not SIMD?

- Le modèle SIMT (*Single Instruction Multiple Thread*) d'un GPU est bien différent du modèle SIMD (*Single Instruction Multiple Data*) classique employé dans les architectures vectorielles
- Le programmeur peut ignorer ce mode de fonctionnement et implémenter un code classique.
- La performance est optimale si le code est adapté (éviter les divergences des branches)

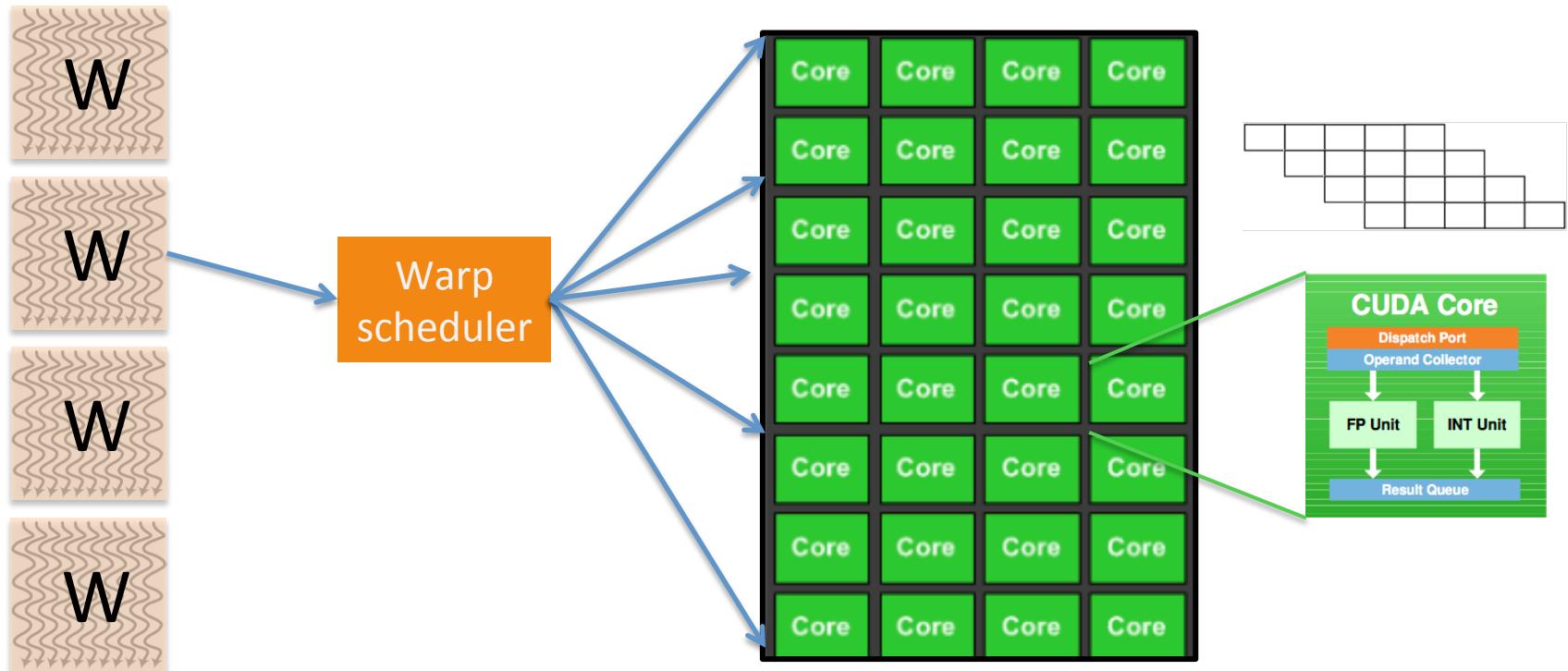


- 4 warp scheduler par SM
 - Chaque warp scheduler peut dispatcher deux instructions par cycle



Warp scheduling

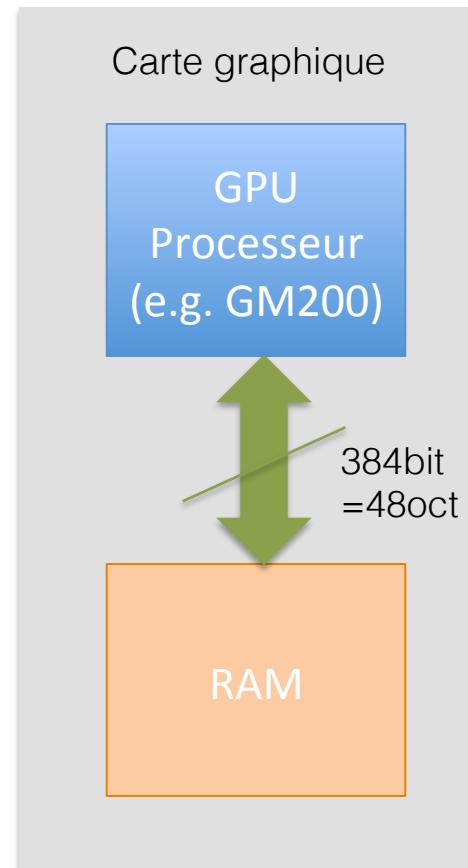
- Chaque *warp scheduler* peut envoyer deux instructions différentes aux cœurs du SM
- Sélection uniquement des warps ayant toutes les dépendances satisfaites
- Les premières étapes des *pipelines* sont faits dans les SM et non pas dans le cœurs (pour le calcul des dépendances)



GPU et mémoire

L'accès à la mémoire GPU est géré par une interface particulièrement large :

Titan X Max	384bit	336Go/s	12Go
GTX 980	256bit	224Go/s	4Go
GTX 750	128bit	80Go/s	1Go
CPU i7	64-128bit	20-37Go/s	



Mémoire et caches

L'accès au cache L1 est plus large encore (1024bit)

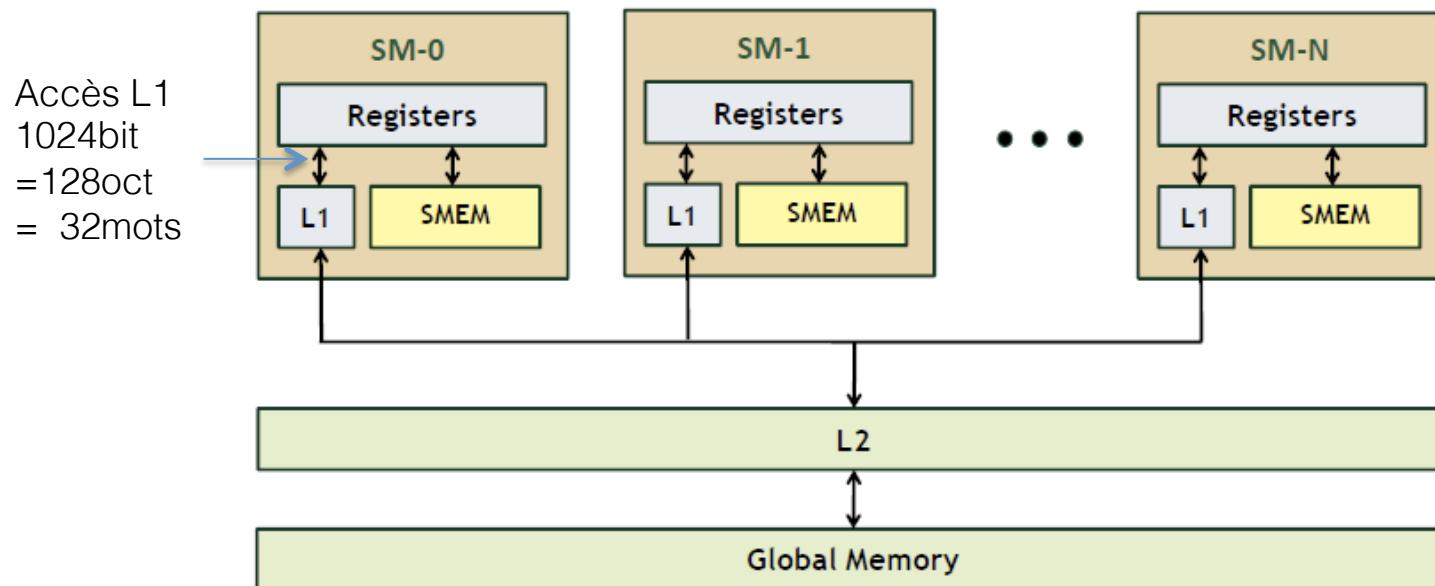


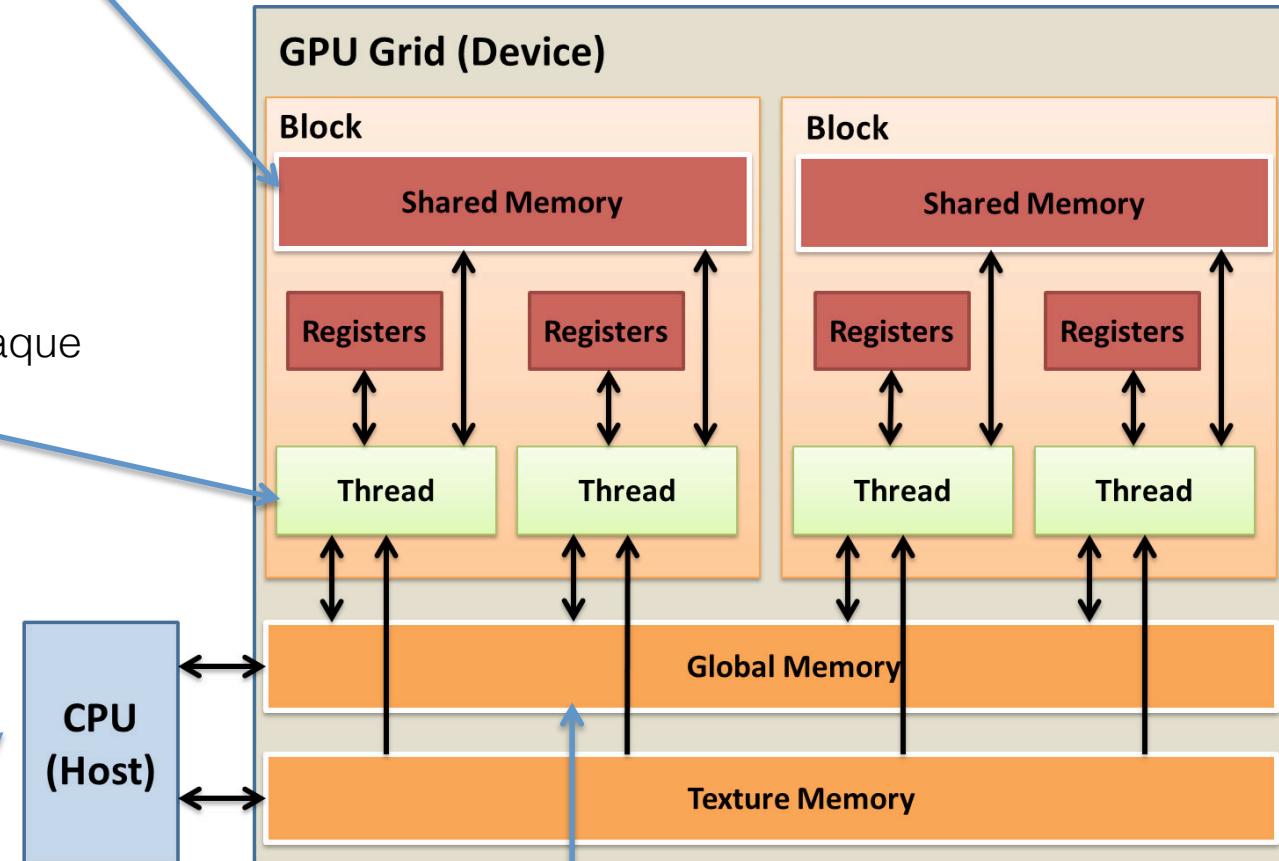
Figure: Orange Owl Solutions

Mémoire partagée
de chaque bloc
(« *Shared* »)

Registres de chaque
thread

Mémoire vive du PC

Mémoire privée du *thread* (« *Local* »)
Mémoire global du GPU (« *Global* »)



Mémoire : propriétés

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	Yes††	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

† Cached in L1 and L2 by default on devices of compute capability 2.x; cached only in L2 by default on devices of higher compute capabilities, though some allow opt-in to caching in L1 as well via compilation flags.

†† Cached in L1 and L2 by default on devices of compute capability 2.x and 3.x; devices of compute capability 5.x cache locals only in L2.

Mémoire partagée = L1 Cache!

La mémoire partagé et le cache L1 sont réalisés par la même mémoire physique : choix à faire par le programmeur.

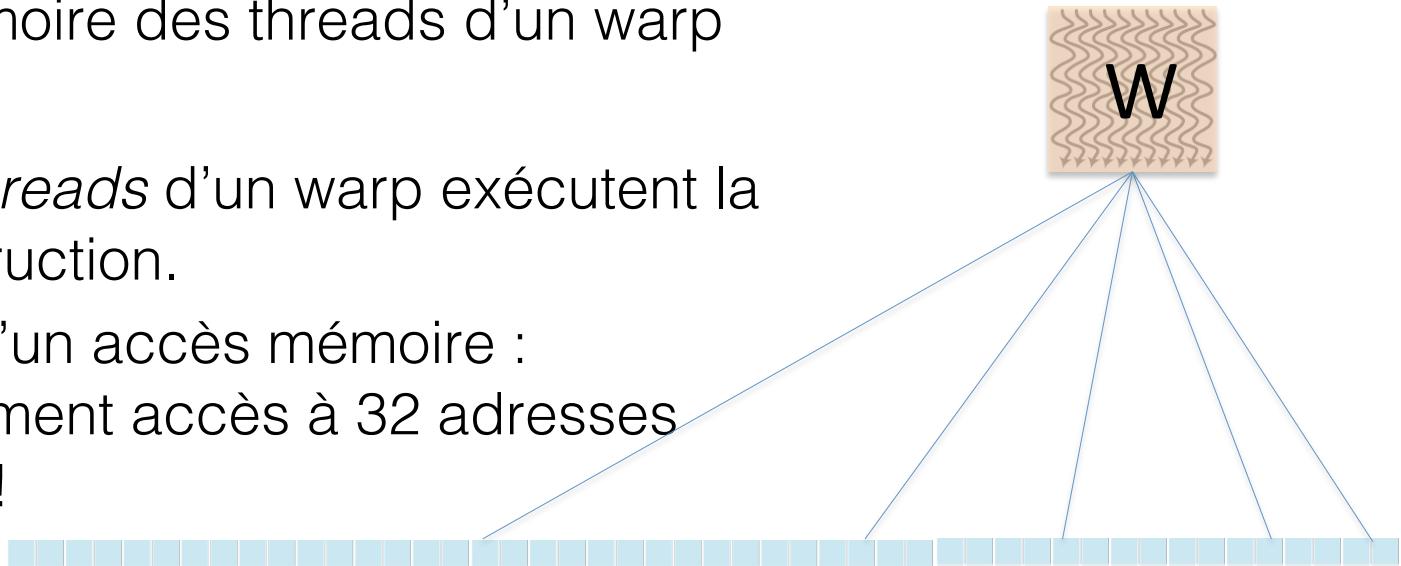


Alignement de la mémoire

- L'adresse d'un type doit être un multiple de la taille du type.
- Exemple pour « int » alignement aux multiples de 4
- Sinon : génération de plusieurs accès mémoire, perte de performance
- Un pointeur renvoyé par *cudaMalloc()* est toujours aligné à un multiple ≥ 256 .

« Coalesced access »

- Objectif : minimiser le nombre de transactions mémoires causées par les accès mémoire des threads d'un warp
- Tous les *threads* d'un warp exécutent la même instruction.
- S'il s'agit d'un accès mémoire : potentiellement accès à 32 adresses différentes!
- Sans aucune stratégie, le système sera surchargé



Coalesced acces : motiver simple

- 32 *threads* d'un *warp* participent
- Chacun accède à un mot de 32 bit = $32 \times 4 = 128$ octets
- Cela correspond à une seule transaction de 128-octets avec le cache L1

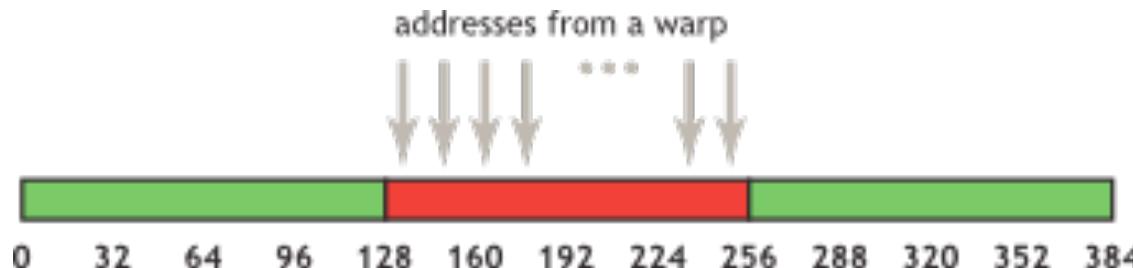


Figure: Nvidia

- Les données pour les *threads* non actifs (divergence!) sont cherchées quand même
- Pas de perte de performance en cas de permutations à l'intérieur de la ligne

Accès non aligné (L1)

Si l'accès du *warp* est séquentiel (mots consécutifs) mais non alignés sur 128bit, deux lignes sont cherchées du cache L1.

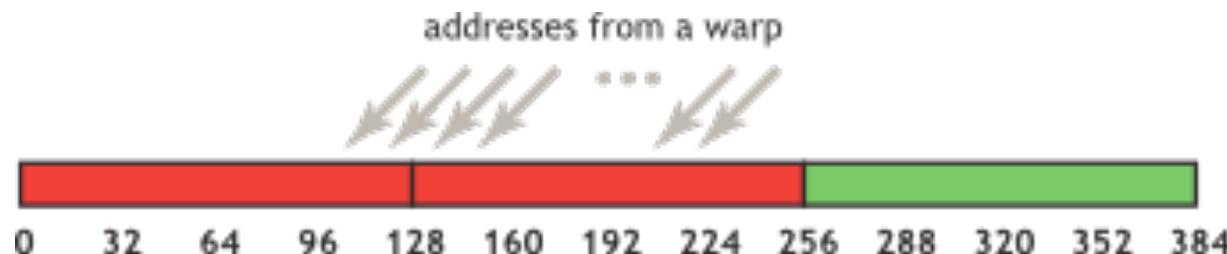


Figure: Nvidia

Accès non aligné (L2)

Cas similaire en cas de *cache miss* L1 avec accès en L2 :

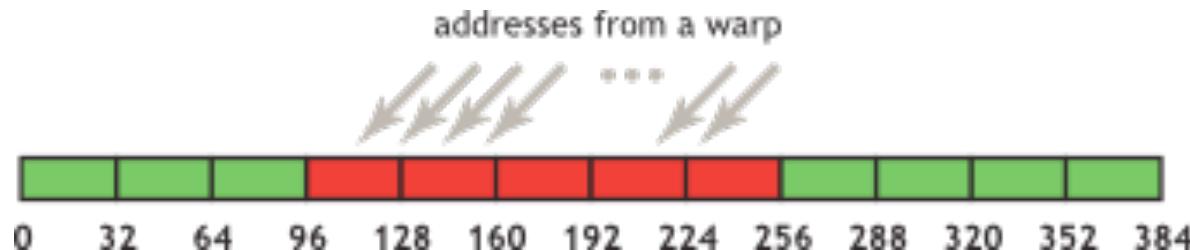


Figure: Nvidia

Accès avec *stride*

Accès à des éléments étant des multiples d'un entier, par exemple de deux :

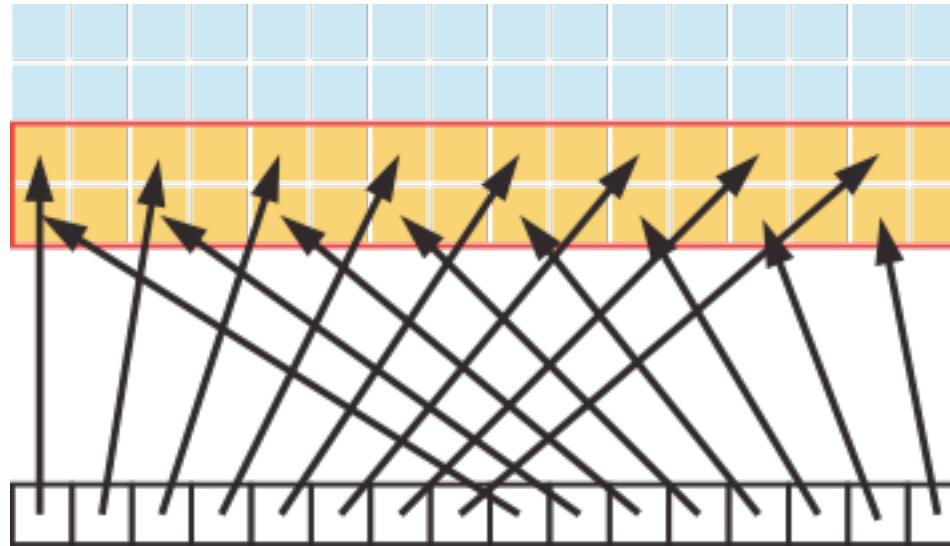


Figure: Nvidia

Chargement de deux lignes de cache L1 (si alignement!)

Accès avec *stride*

Perte de performance avec des accès de moins en moins dense. Au pire de cas : lecture d'une ligne de cache par thread du *warp*!

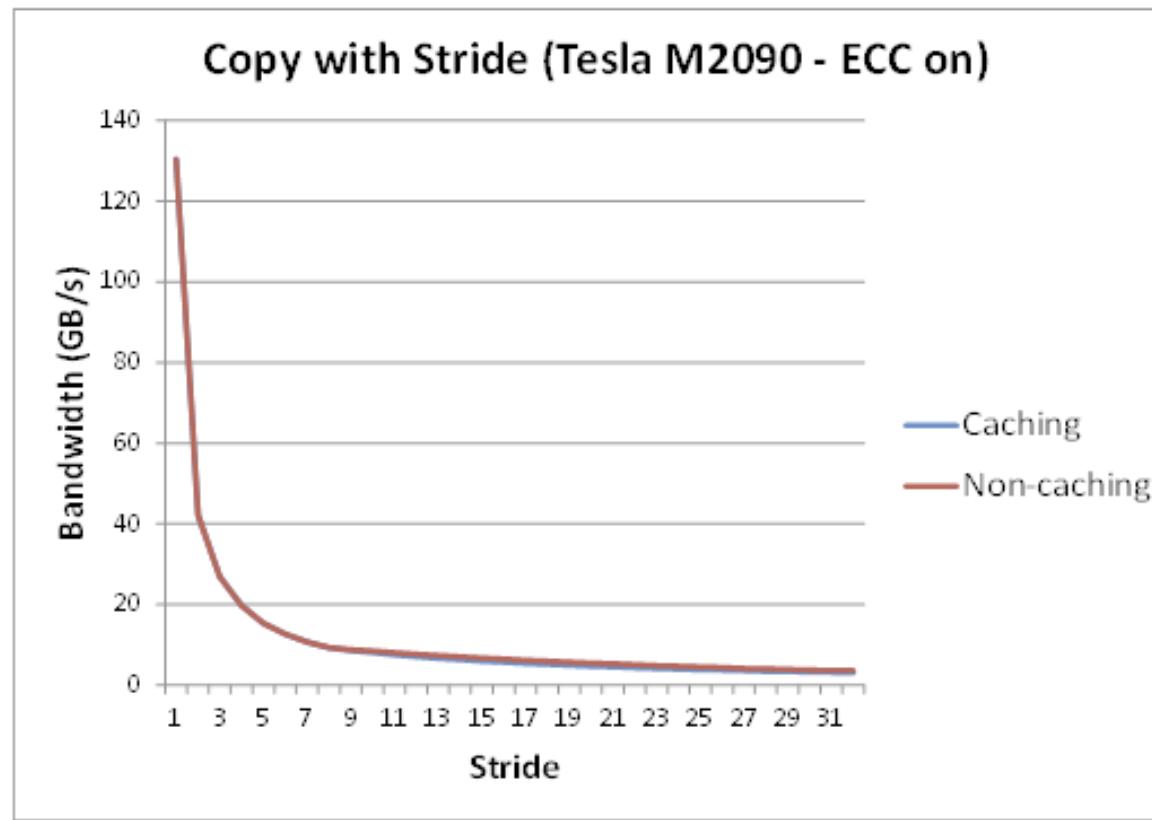


Figure: Nvidia

Accès à la mémoire partagée

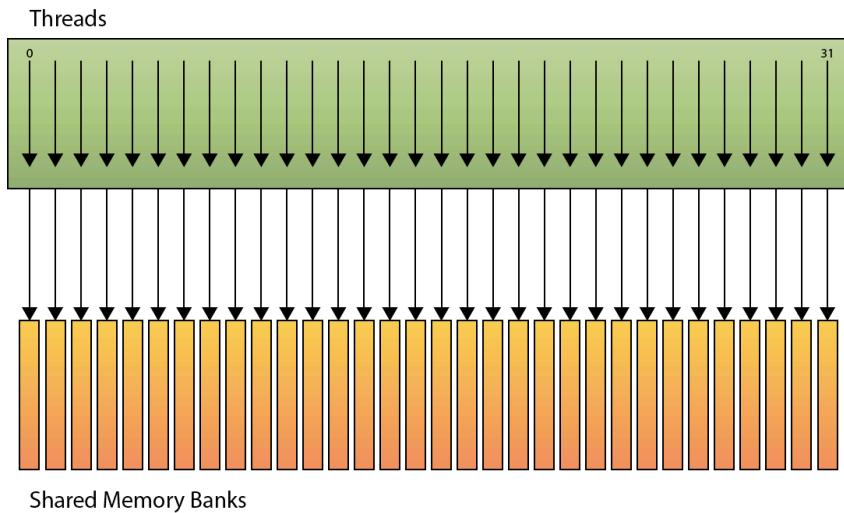
La mémoire partagée est organisée en bancs. Les accès aux bancs différents peuvent être réalisés en parallèle.

Chaque banc peut fournir 64 bits chaque cycle.

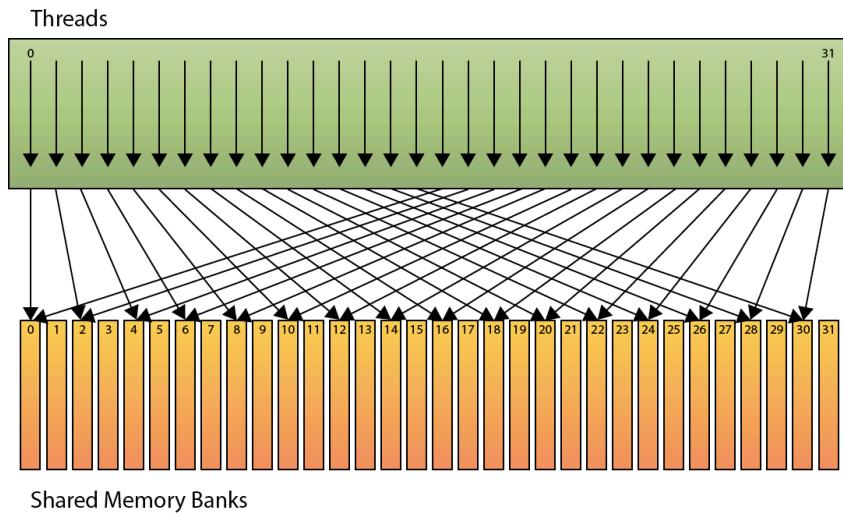
Deux modes possibles :

- Changement de banc après 32bit
- Changement de banc après 64bit

Accès à la mémoire partagée



No conflict



2 way conflict, no broadcast!

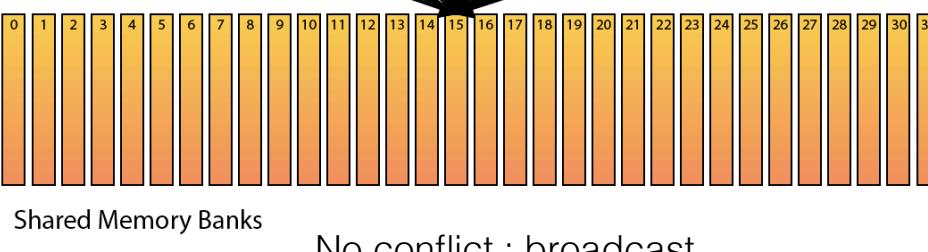
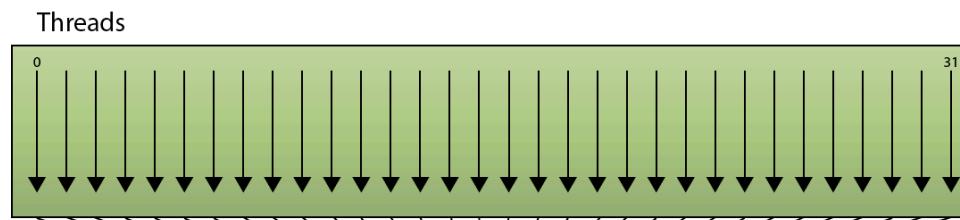


Figure:
3dgep.com

Accès aux matrices 2D

Les matrices 2D sont souvent accédées de manière suivante :

```
1 | adresse_debut + largeur * y + x
```

Pour que l'accès soit « *coalesced* », il faut configurer comme multiple de 32 (taille du *warp*) :

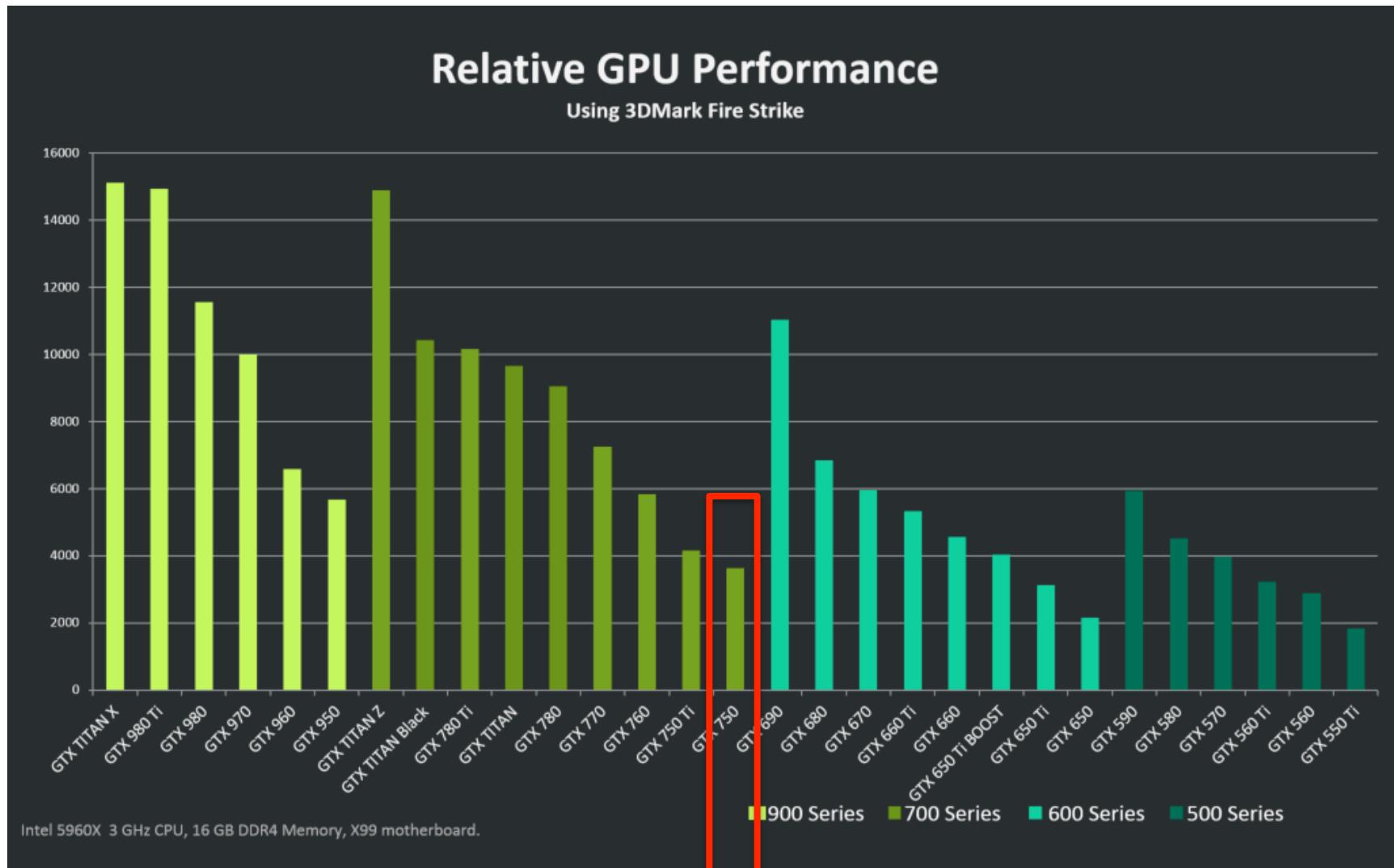
- Largeur du tableau
- Taille du *thread block*

Compute capabilities

GeForce GTX 750 :
installées en salle TP

Technical Specifications	Compute Capability							
	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3
Maximum number of resident grids per device (Concurrent Kernel Execution)	16	4		32			16	
Maximum dimensionality of grid of thread blocks			3					
Maximum x-dimension of a grid of thread blocks	65535			$2^{31}-1$				
Maximum y- or z-dimension of a grid of thread blocks			65535					
Maximum dimensionality of thread block			3					
Maximum x- or y-dimension of a block			1024					
Maximum z-dimension of a block			64					
Maximum number of threads per block			1024					
Warp size			32					
Maximum number of resident blocks per multiprocessor	8		16			32		
Maximum number of resident warps per multiprocessor	48			64				
Maximum number of resident threads per multiprocessor	1536			2048				
Number of 32-bit registers per multiprocessor	32 K		64 K	128 K		64 K		
Maximum number of 32-bit registers per thread block	32 K		64 K			32 K		
Maximum number of 32-bit registers per thread	63			255				
Maximum amount of shared memory per multiprocessor		48 KB		112 KB	64 KB	96 KB	64 KB	
Maximum amount of shared memory per thread block			48 KB					
Number of shared memory banks			32					
Amount of local memory per thread			512 KB					
Constant memory size			64 KB					
Cache working set per multiprocessor for constant memory		8 KB			10 KB			
Cache working set per multiprocessor for texture memory	12 KB		Between 12 KB and 48 KB					

Performances 3D



GTX 750 : installées en salle TP

En salle de TP : Nvidia GeForce GTX 750

Modèles	GeForce GTX 750	GeForce GTX 750 Ti	GeForce GTX Titan X
Finesse de gravure des processeurs			
Code de la puce	GM107	GM200	
Surface de la puce	148 mm ²	601 mm ²	
Transistors	1.9 G	8,1 G	
Fréquence 3D	1010 MHz	1000 Mhz	
Fréquence Turbo	≈ 1100 MHz	≈ 1100 MHz	≈ 1100 Mhz
Température maximale avec Turbo		80	83 °C
Nombre de ROP	16		96
TMU	32	40	192
cœurs Cuda	512	640	3072
SFU	?		
GPC	1		6
SMM	4	5	24
Enveloppe thermique	60 W	65 W	250 W
Type de mémoire			
Capacité possible	1 Go	2 Go	12 Go
Vitesse de la mémoire	1250 MHz	1350 MHz	1753 Mhz
Largeur du bus mémoire	128 bits		384 bits
Débit mémoire	74,5 Go/s	80,5 Go/s	336 Go/s
Pixels Fillrate	17,3 Gpixel/s		103 Gpixel/s
Textures Fillrate	34 Gtexel/s	43 Gtexel/s	
Filtrage Géométrique	1445 Mtriangle/s	1807 Mtriangle/s	
Calcul Simple Précision	1111 Gflops	1388 Gflops	6156 Gflops
Calcul Double Précision	34 Gflops	43 Gflops	206 Gflops
Date	18/02/2014		04/2015



Conclusion

- Un GPU est conçu pour les cas où une très grande quantité de données est traitée avec les mêmes instructions (SIMT)
- Le modèle de programmation est très générale. Le modèle SIMT est caché du programmeur
- Un très grand nombre d'unités de calcul est disponible
- Les latences dues aux accès à la mémoire sont cachées par l'exécution très selective des instructions « prêtes »

