UNIVERSITÉ Grenoble Alpes

POLYTECH GRENOBLE

Architecture des processeurs

Nvidia TEGRA X1

Partie GPU

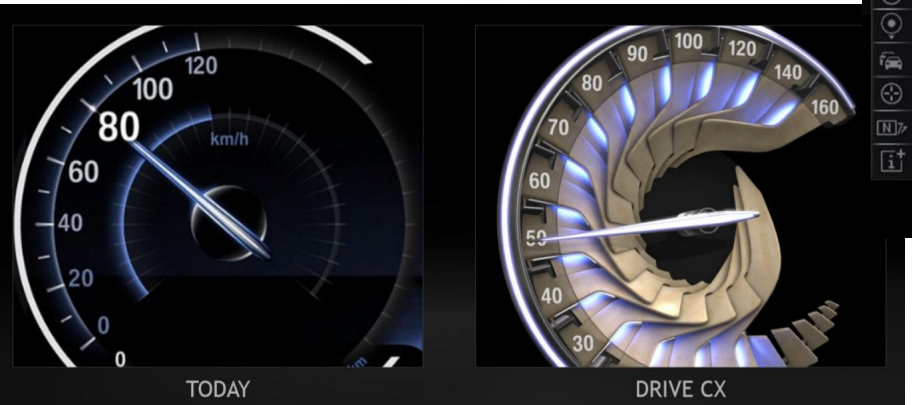
Florian Di Palma, Andy Robin, Rodolphe Fleury, Justine Lachenaud

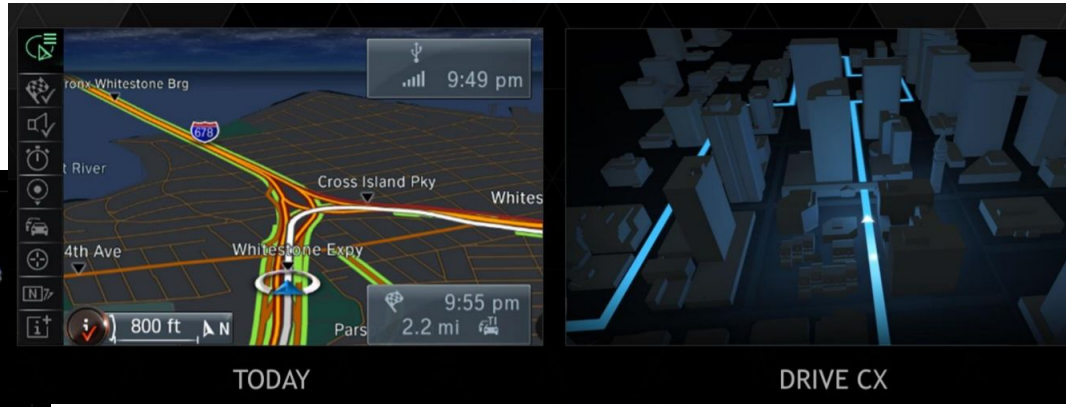
Sommaire

1. Présentation générale
2. Architecture Maxwell
3. Le Warp Scheduler
4. Exemple : Coalesced Access
5. Les mémoires
6. Les coeurs
7. Codage du GPU
8. Comparaison CPU/GPU au travers d'un exemple

Présentation Générale

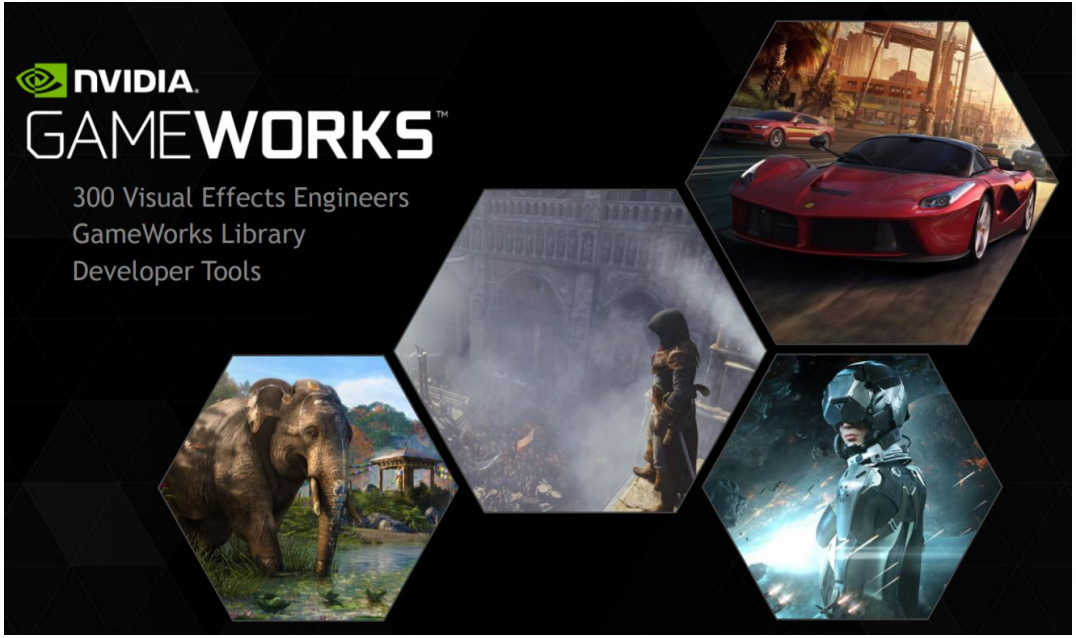
- Présent dans les nouvelles technologies embarquées : automobile, robotique, vision par ordinateur.
- En particulier : aide au conducteur, vision par ordinateur, deep-learning.





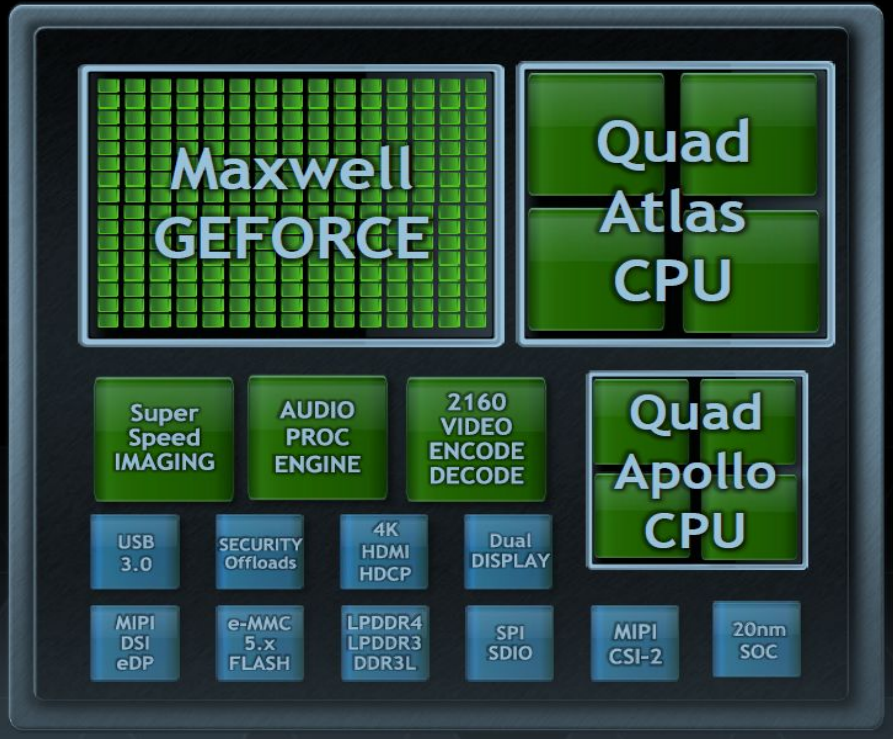
Présentation Générale

- Jeux vidéos, Nintendo Switch



Présentation Générale

TEGRA X1 – OVERVIEW



GRAPHICS	Maxwell GeForce - World's Fastest GPU *2 x SMM units, DX-next, OpenGL 4.4*
CPU	Octo-Core 64b ARM v8 CPU Complex *4xCA57 Atlas/2MB L2; 4xCA53 Apollo/512KB L2*
MEMORY	64b / Dual-Quad Channel Memory Interface *LPDDR4-3200, LPDDR3E-1866, DDR3L-1866*
VIDEO	4K x 2K Encode and Decode *H.264, H.265, VP8, VP9 (dec-only)*
POWER	Low Power *20nm, HW Offloads, Isolated Pwr Rails, PRISM*
DISPLAY	4K x 2K 24b @60Hz, 1080p @120Hz *DSI 2x4, eDP, High Speed HDMI 2.0, DP*
IMAGING	Full Quad Camera imaging, Dual ISP 650Mp/s *Maxwell 16fp imaging GPGPU, HW AO-HDR*
Mobile I/O	Designed for mobile *e.MMC5.x, USB3.0/2.0/HSIC, SD/SDIO 3.0, CSI-2*

Architecture Maxwell

Architecture Maxwell

Différences entre CPU et GPU

CPU

- Optimiser la performance “séquentielle”
- Diminuer / maîtriser les latences d’accès à la mémoire (cache)
- Logique sophistiquée pour résoudre des dépendances (prédiction de branchements, exécution out of order etc.)

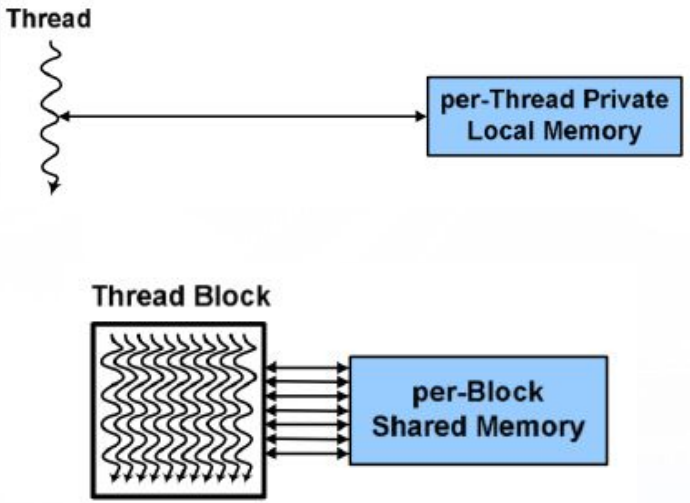
GPU

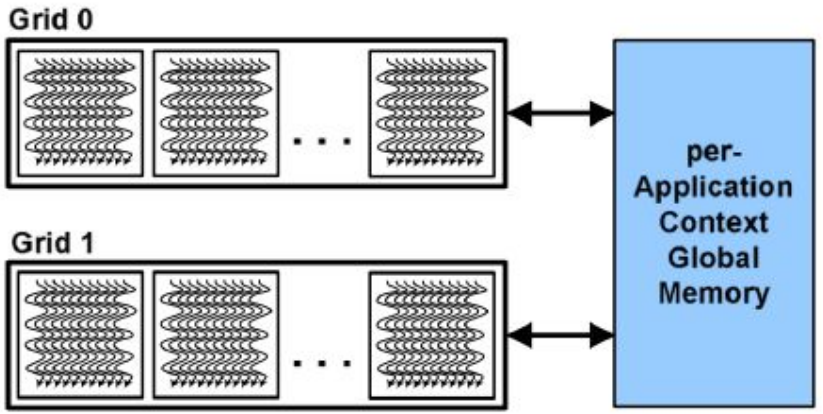
- Existence d’un très grand nombre de thread
- Cacher des latences d’accès à la mémoire (exécuter les instructions pour lesquelles les données sont disponibles)

=> Optimise la performance par watt

Architecture Maxwell

Exécution des threads





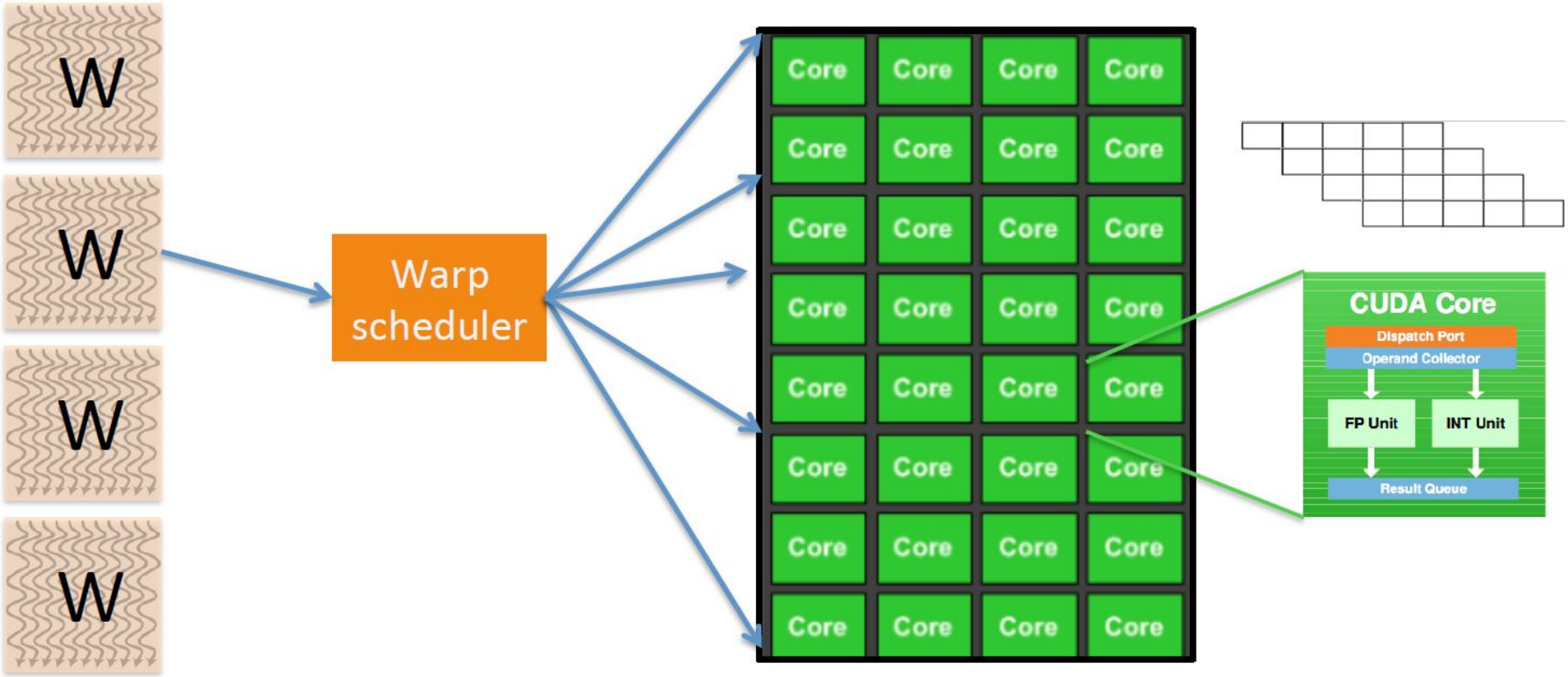
Le Warp Scheduler

Problèmes :

- Comment fournir les données aux threads si la bande passante est limitée ?
- Comment fournir 256 instructions différentes par cycle ?

```
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 }
```

Le Warp Scheduler

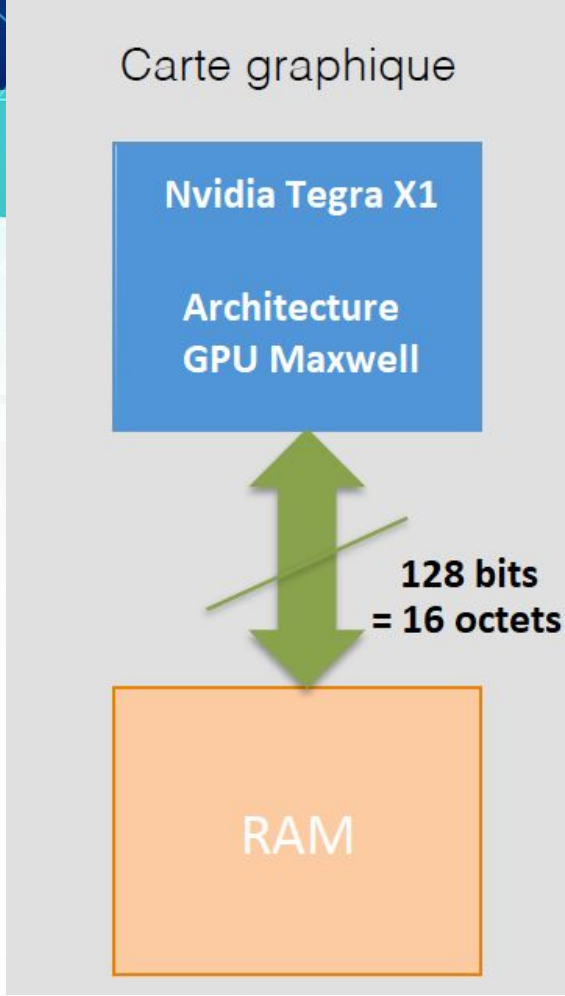


La mémoire

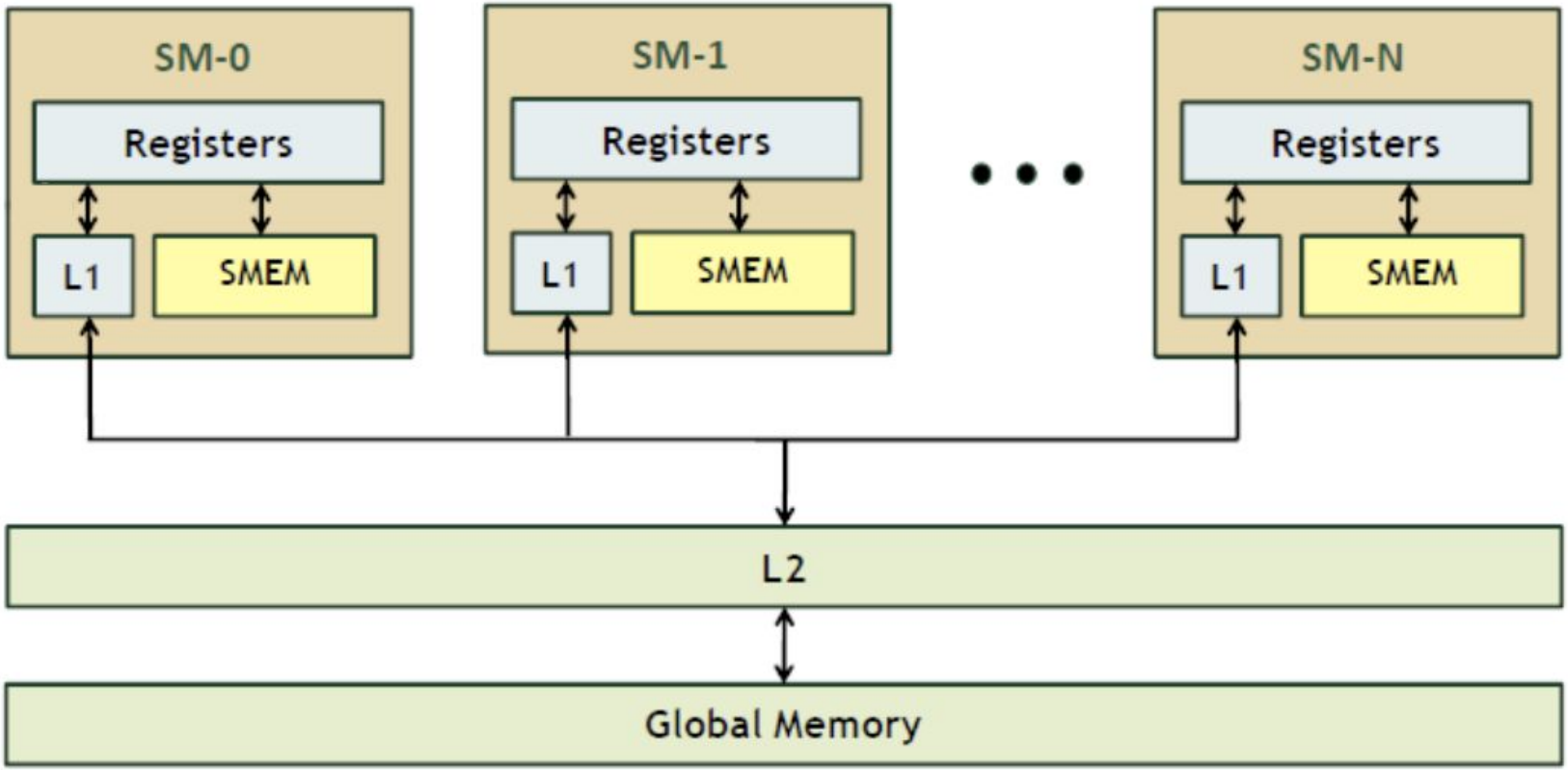
- Interface mémoire 128 bits = 16 octets
- Bande passante de 25.6 Go/s

LPDDR4-3200 MHz (mémoire externe partagée avec le CPU)

(2*32 bits * 3200 MHz) / 8 = 25.6 Go/s



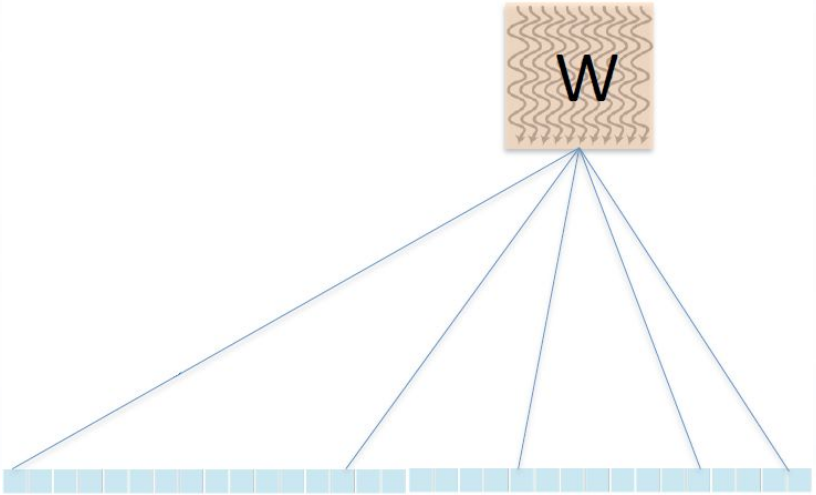
La mémoire - représentation globale



Exemple : Coalesced access

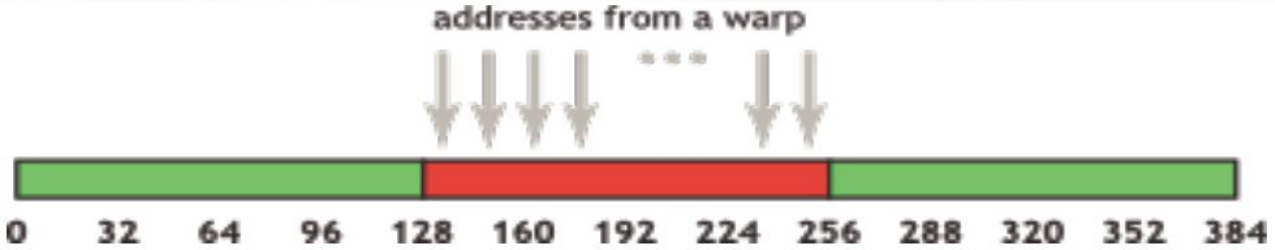
- Objectif : minimiser le nombre de transactions mémoire causée par les accès mémoire des 32 threads d'un Warp

- Tous les threads d'un warp exécute la même instruction
  => Potentiellement 32 adresses différentes



Coalesced access

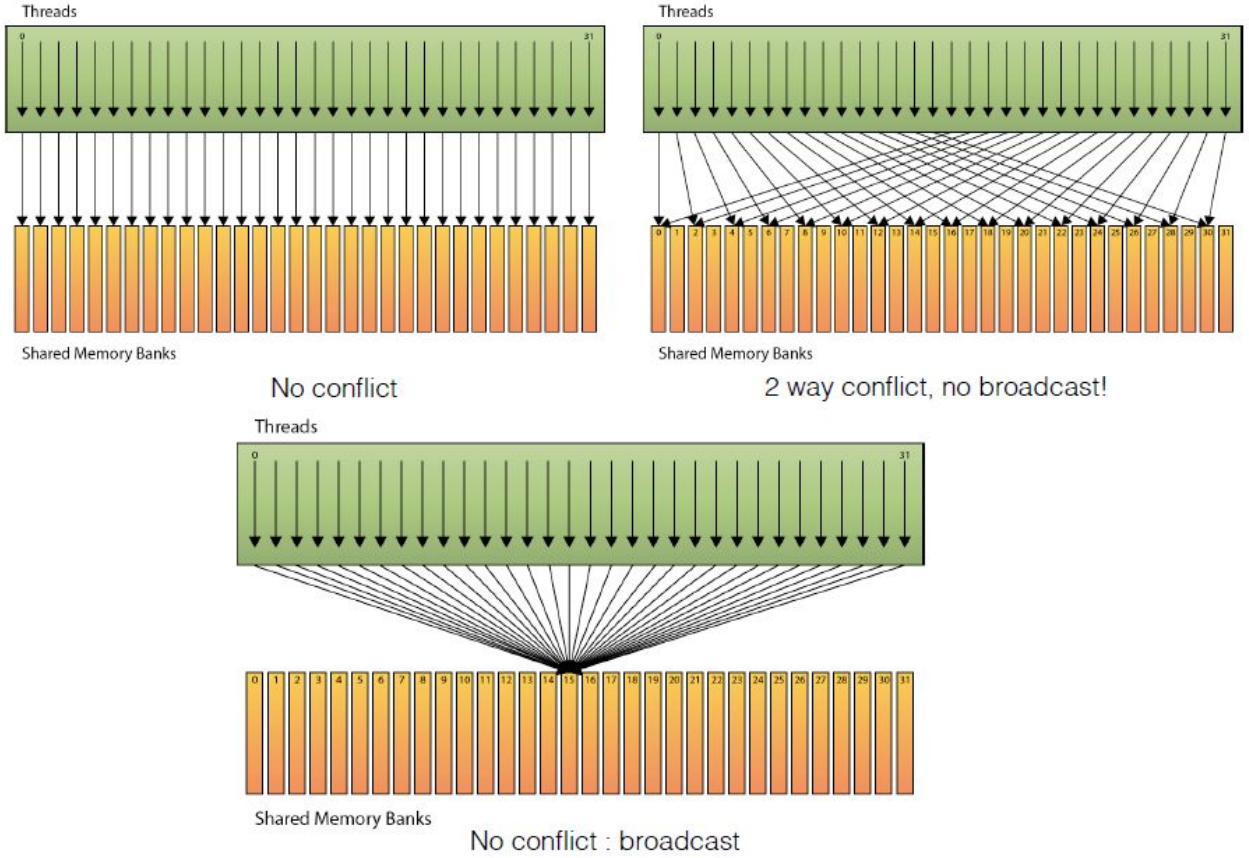
- 32 threads d'un warp participent (4 warp sur chaque SM)
- Chacun des threads accède à un word de 32 bits = 32*4=128 bits
- Cela représente une transaction de 128 bits avec le cache L1



Example : Matrice 2D

On y accède par => *adresse_debut* + *largeur* * *y* + *x*

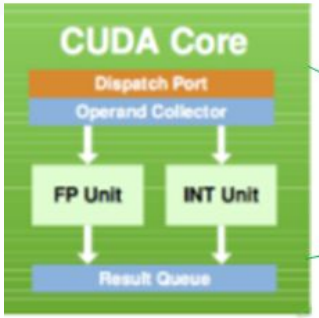
Accès à la mémoire partagée



Les coeurs

Les CUDA cores:

- 256 cœurs: 128 par SMM
- 2 ALU: « int » et « float »
- Un pipeline sur chaque ALU
- Pas d'étape IF et DE



Les coeurs

FMA (Fused Multiply-Add):

- Intégré aux CUDA core
- Fusion de 2 opération dans une instruction
- Opération des floats sur 16 bits

SFU (Super Function Unit):

- Opérations plus complexes
- Plus rapide, moins précis

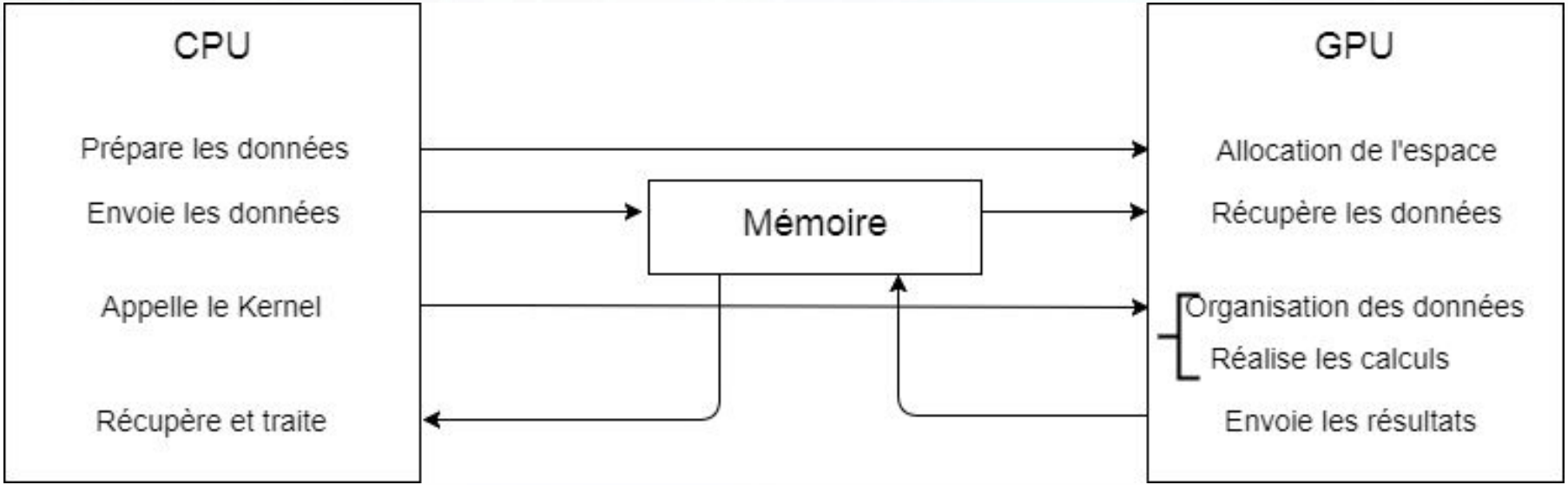
Codage du GPU

Environnement CUDA

- Architecture de traitement parallèle
- Calcul intensif
- Utilisation des « CUDA cores »
- Langage C/C++, fortran

Codage du GPU

Organisation d’un code CUDA:



Les kernels

- Fonction exécutée sur le GPU
- 3 Différents types :
  1. __global__ : correspond à un kernel exécuté sur le GPU mais appelé par le CPU
  2. __device__ : correspond à un kernel exécuté et appelé par le GPU
  3. __host__ : correspond au mode de fonctionnement par défaut

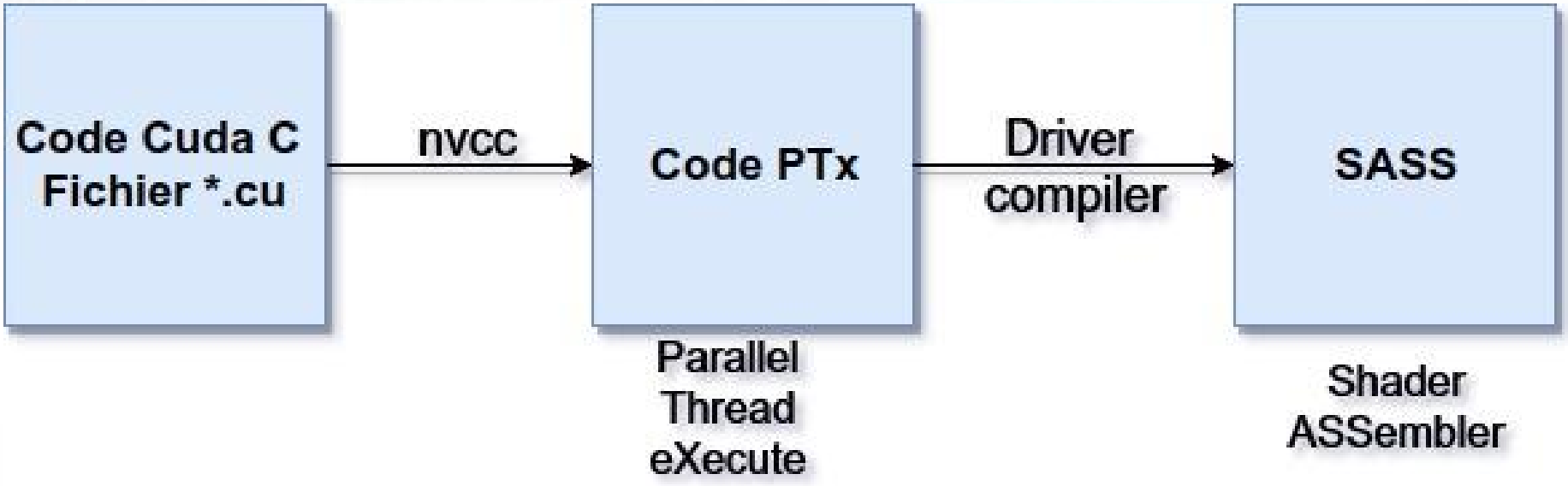
Exemple d'un kernel

```
__global__ void vecAdd(float * A,
float * B, float * C){
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
…
int main(){
  // utilisation du kernel
  vecAdd<<<N, 1>>>(A, B, C);
// kernel <<<threadsParBloc, nBlocs
>>> (arguments);
}
```

Variables implicites en lecture seule

1. **`blockIdx`** : index du bloc dans la grille,
2. **`threadIdx`** : index du thread dans le bloc,
3. **`blockDim`** : nombre de threads par bloc (valeur de *threadsParBloc* du paramétrage du kernel).

Compilation




Conclusion

Sources :

https://www.overclockingmadeinfrance.com/quelles-sont-les-unites-de-base-dun-gpu/

http://tcuvelier.developpez.com/tutoriels/gpgpu/cuda/introduction/

http://tcuvelier.developpez.com/tutoriels/gpgpu/cuda/approfondi/

http://jeux.developpez.com/faq/gpgpu/

http://www.stackoverflow.com

http://liris.cnrs.fr/christian.wolf/teaching/gpu/ : Cours CNRS

https://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf : Whitepaper

Documentation technique Nvidia

Documentation du CUDA Toolkit