Programmation GPU

P. Fortin
Programmation parallèle avancée (PPA)
Master 2 Informatique SAR-STL, UPMC

Programmation parallèle avancée (PPA)

- Plan :
  - Programmation GPU
  - Introduction au calcul haute performance
  - Programmation des architectures multicoeurs et optimisation de code
- Evaluation :
  - Projet (en binôme) : 40 %
  - Examen : 60 %

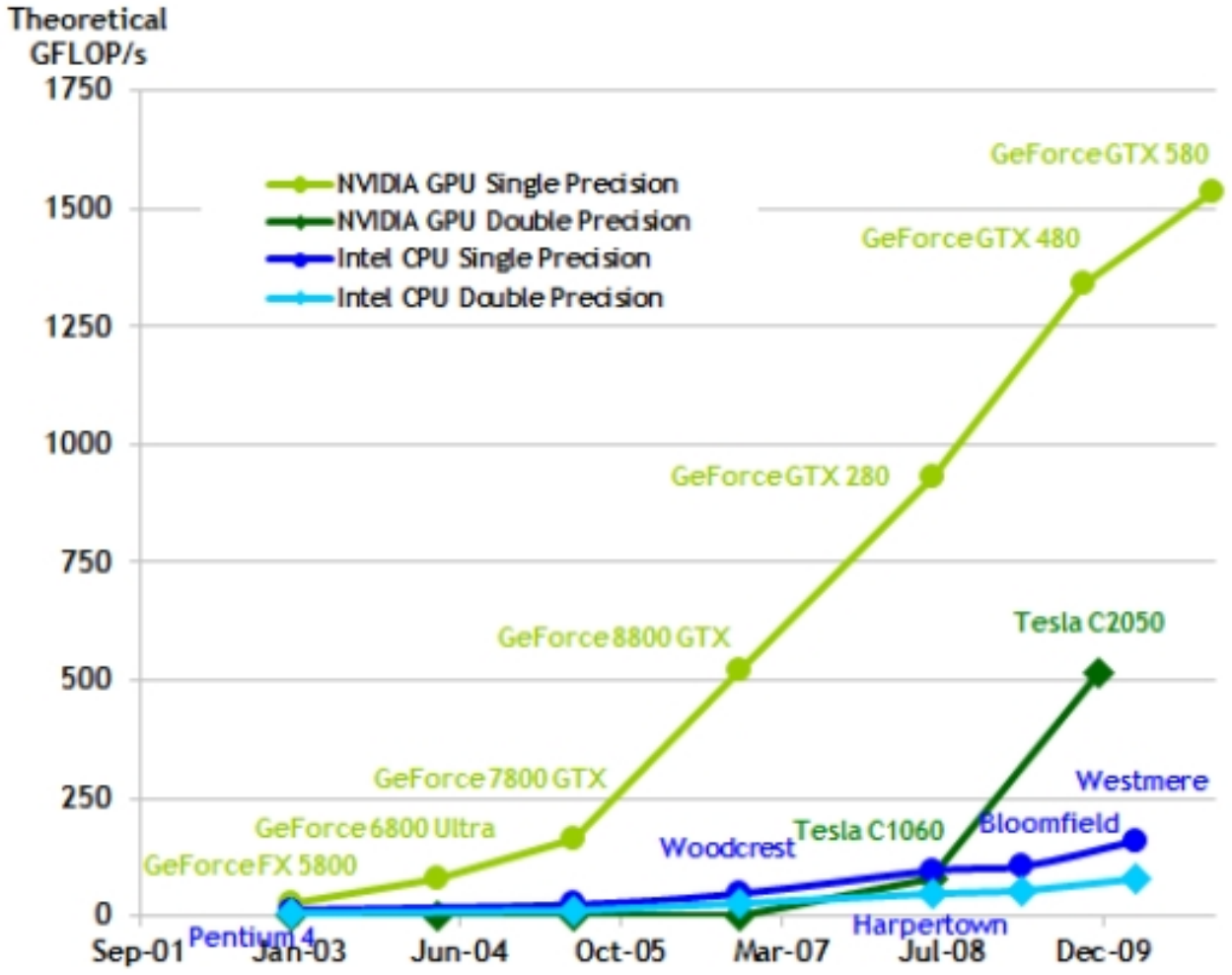
Programmation GPU : références

- Documentation CUDA :
  http://docs.nvidia.com/cuda/index.html
  Notamment :
  - CUDA C Programming Guide
  - CUDA C Best Practices Guide
  - Exemples du CUDA SDK : réduction, transposition de matrice, produit de matrices, …
- Cours et tutoriels CUDA :
  https://developer.nvidia.com/cuda-training
  Par ex. : ECE 408 (W. W. Hwu and D. Kirk, Univ. Of Illinois, https://ece408.hwu.crhc.illinois.edu/)
- Remerciements
  - J.L. Lamotte, *Programmation GPU*, UPMC

Les processeurs graphiques (GPU)

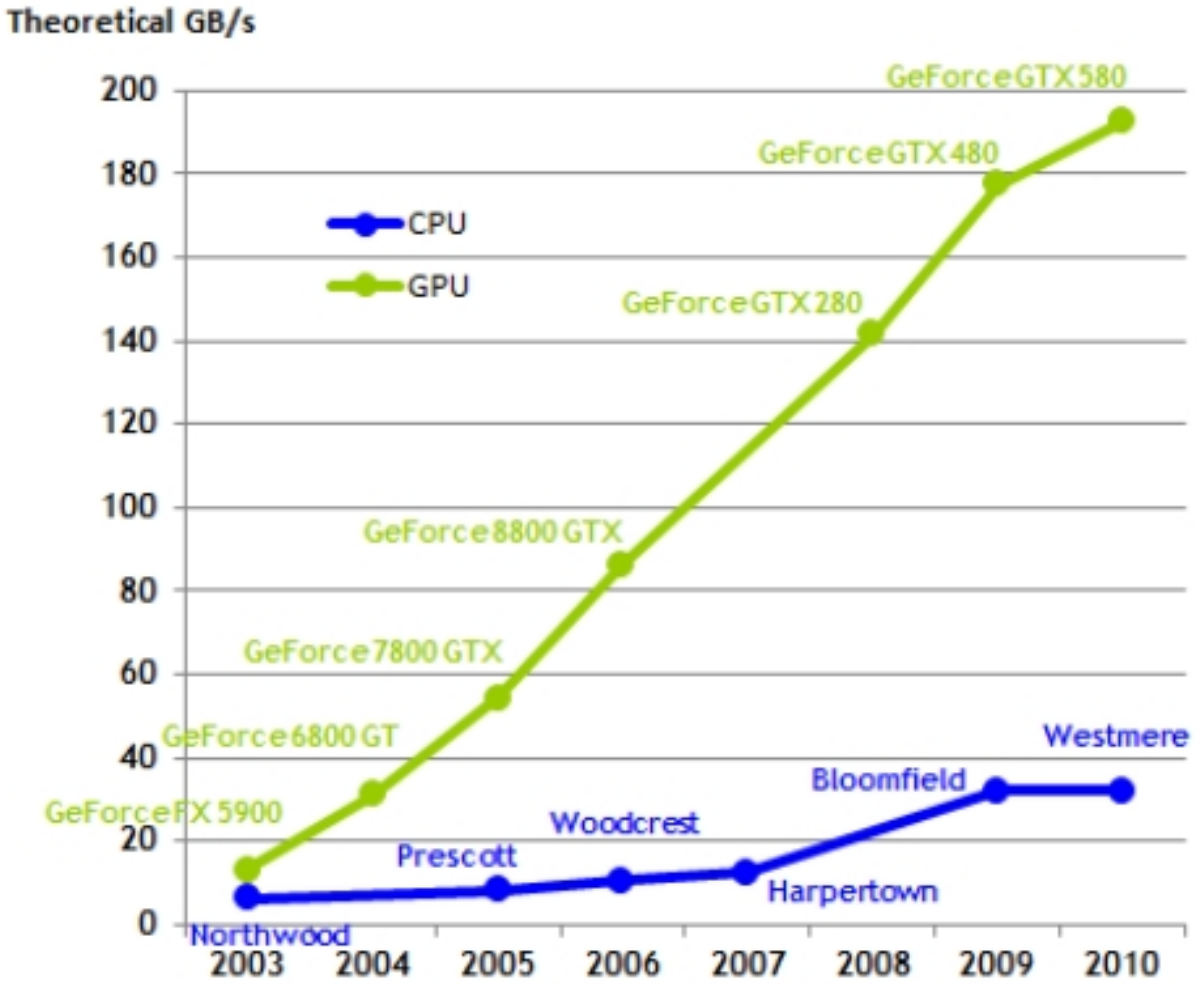
- GPU : *Graphics Processing Unit*
- GPGPU : *General-Purpose computation on Graphics Processing Unit* (ou *GPU Computing*)
  - Programmation CUDA (2007) : GPU NVIDIA
  - Programmation OpenCL (2008) : GPU NVIDIA, GPU AMD, CPU multicœur, FPGA ...
- Architectures massivement parallèles conçues pour décharger le CPU des traitements graphiques
  - Introduction progressive des nombres flottants, de la simple et de la double précision, du respect de la norme IEEE et des fonctions mathématiques élémentaires (cos, sin, sqrt...)
  - Architectures « *manycore* »

Comparaison CPU-GPU : performance crête



(CUDA Programming Guide 4.0)

Comparaison GPU-GPU : bande passante mémoire



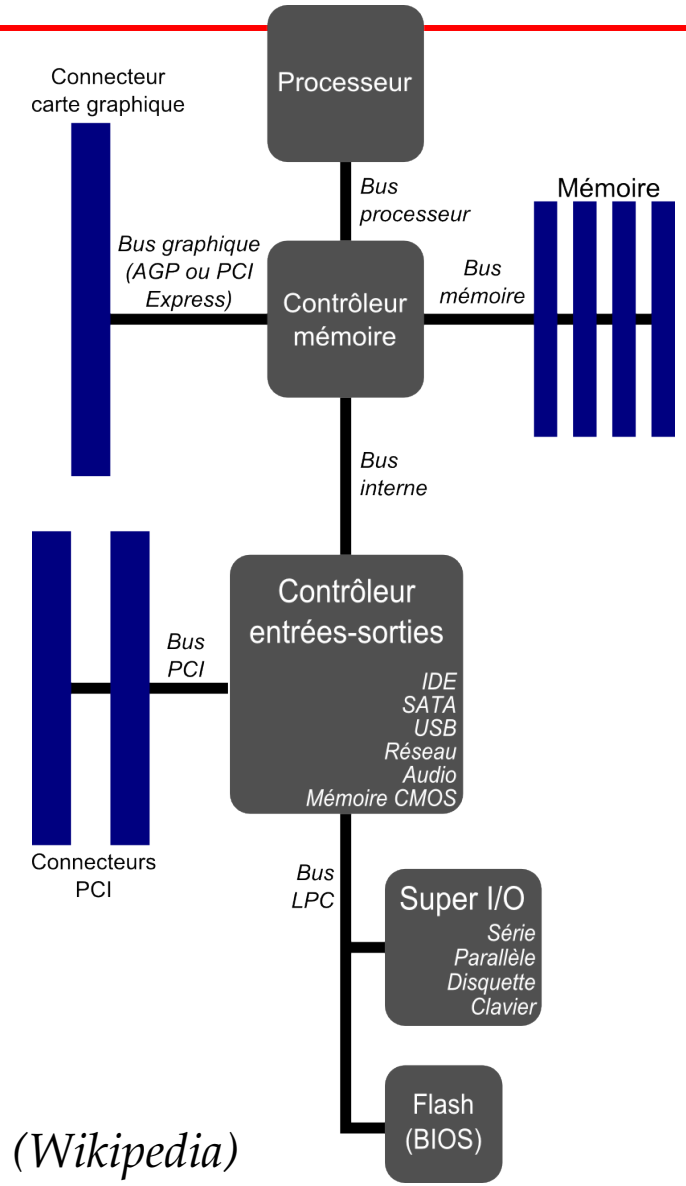
(CUDA Programming Guide 4.0)

Comparaison des architectures CPU-GPU

- Réduction de la place prise par les caches et par les unités de contrôle de l'exécution → augmentation du nombre d'unités de calcul
- GPU adapté à du **parallélisme de données massif et régulier**

(CUDA Programming Guide 2.3)

Architecture matérielle

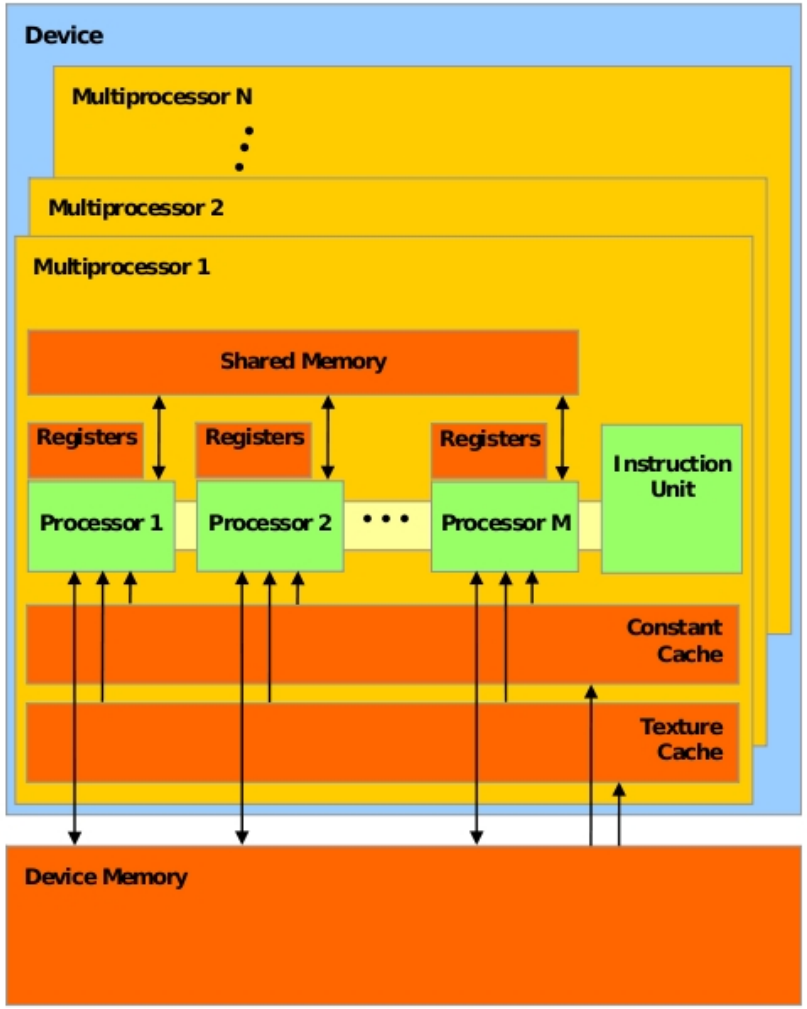


(Wikipedia)

- Bus processeur :
  - QPI (Intel) : 25.6 Go/s
  - HyperTransport (AMD) : 51.2 Go/s
- Bus mémoire : par ex. DDR3-1600 → 12,8 Go/s
- PCI Express :
  - 2.0 16x : 8 Go/s dans chaque direction
  - 3.0 16x : 16 Go/s dans chaque direction
- Bande passante mémoire GPU : 177.4 Go/s (GTX 480)

Modèle de l'architecture : unités de calcul

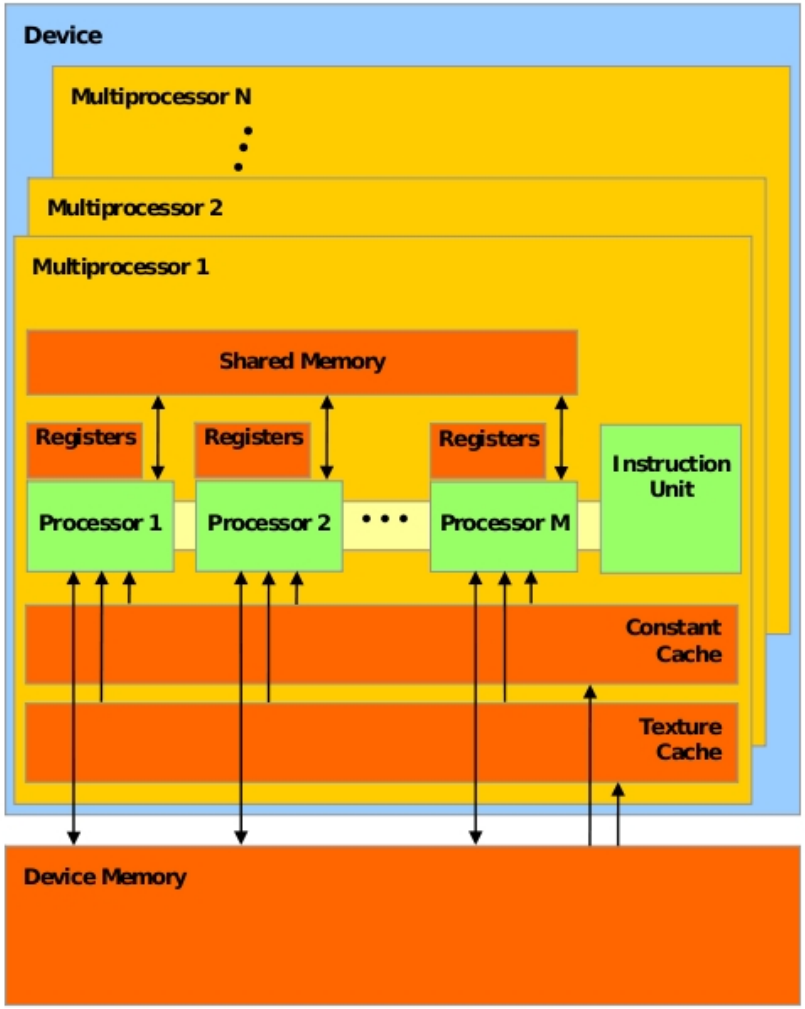
- GPU composé de plusieurs *multiprocesseurs* (MP)
  → architecture MIMD au niveau des MP
- Chaque multiprocesseur est composé de plusieurs *scalar processors* (SP) ou *cœurs CUDA* :
  - ➤ 32 SP par MP sur architecture Fermi
  - ➤ **Exécution SIMD** des SP dans un MP
- 1 MP GPU ≈ 1 cœur CPU avec unités SIMD



(CUDA Programming Guide 2.3)

Modèle de l'architecture : niveaux mémoire

- *Registres* :
  - 32768 par MP (Fermi)
  - Accès instantané
- *Mémoire partagée* :
  - Entre 16 Ko et 48 Ko par MP (Fermi)
  - Latence : ~4 cycles
- *Cache L1* (depuis Fermi) :
  - Taille cache + taille mémoire partagée = 64 Ko
  - Au même niveau que la mémoire partagée
  - Et cache L2 (depuis Fermi) commun à tous les MP (768 ko)
- *Mémoire globale :*
  - Accessible par tous les MP
  - Latence : entre 400 et 800 cycles



(CUDA Programming Guide 2.3)

Compute capability CUDA

- Classement des architectures GPU CUDA
- *Compute capability* :
  - <1.3 : pas de double précision pour les nombres à virgule flottante
  - 1.3 : double précision pour les nombres à virgules flottante
  - 2.0 et 2.1 : architecture Fermi (exécution concurrente de kernels, caches...)
  - 3.0 : architecture Kepler
  - 3.5 : architecture Kepler (dont Dynamic parallelism)

Architecture des GPU : exemple (1)

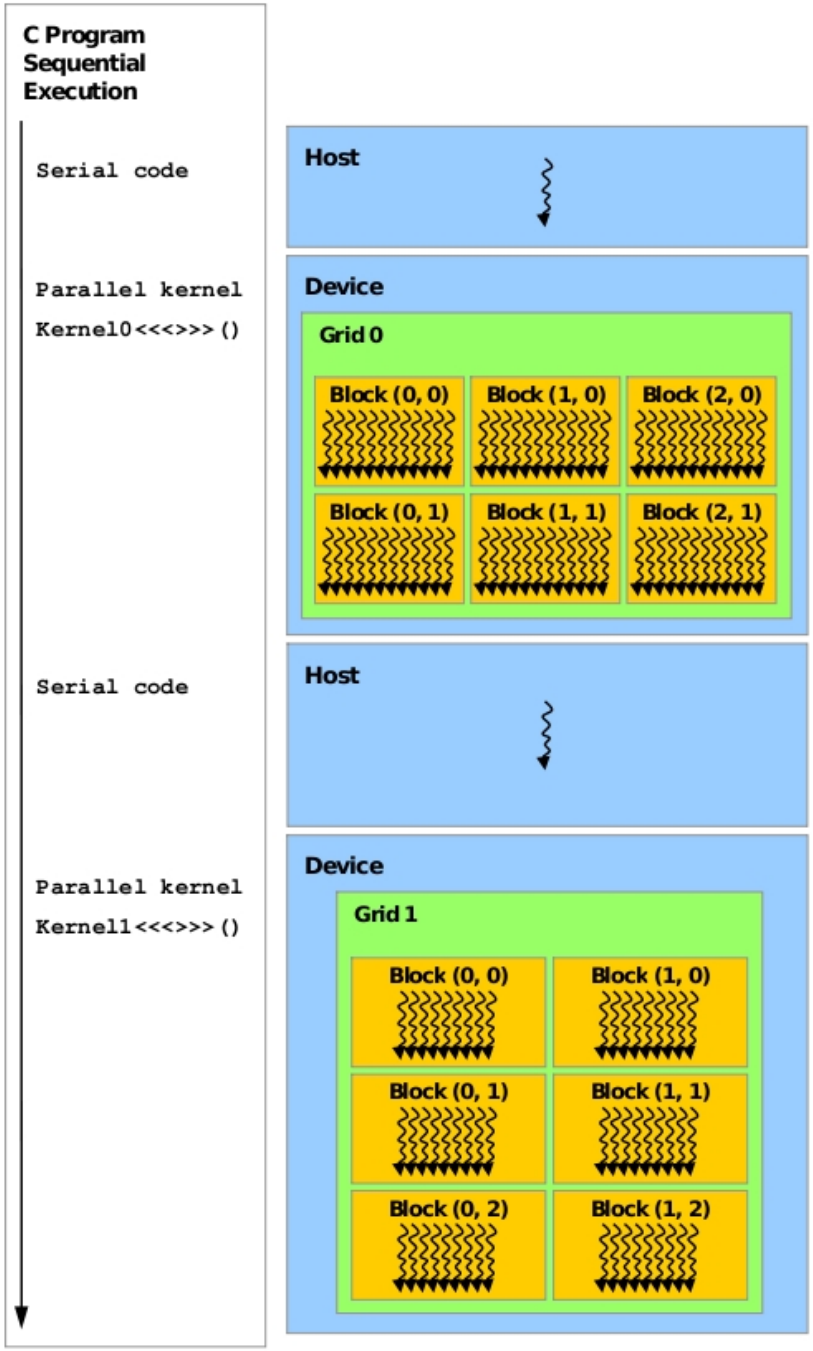
- Exemple NVIDIA GTX 480 :
  - Architecture Fermi
  - 15 Multiprocesseurs (MP) avec 32 *Scalar Processors* (SP) chacun
    → 480 SP @ 1.401 GHz
  - Performance crête :
    - Simple précision : 1,345 Tflop/s
    - Double précision : 168,120 Gflop/s
  - Mémoire globale (*Device Mem*) : 1536 MB (GDDR5)
  - Bande passante mémoire : 177,4 Go/s
  - *Compute capability* CUDA : 2.0
  - Jusqu'à 16 kernels concurrents
  - Cache L1 + mémoire partagée = 64 Ko
  - Cache L2 (768 Ko)

Architecture des GPU : exemple (2)

- Exemple NVIDIA Fermi C2050/C2070 :
  - 14 Multiprocesseurs (MP) avec 32 *Scalar Processors* (SP) chacun
    → 448 SP @ 1.15 GHz
  - Performance crête :
    - Simple précision : 1,03 Tflop/s
    - Double précision : 515 Gflop/s
  - Mémoire globale (*Device Mem*) : 3 Go pour C2050, 6 Go pour C2070
  - Bande passante mémoire : 144 Go/s
  - *Compute capability* CUDA : 2.0
  - Jusqu'à 16 kernels concurrents
  - Cache L1 + mémoire partagée = 64 Ko
  - Cache L2 (768 Ko)
  - Protection mémoire ECC

Principes généraux

- Code C/C++ séquentiel sur CPU
- *kernel* = code parallèle CUDA sur GPU
- Lancement du programme sur CPU (= *host* / *hôte*)
  - Transfert de données du CPU vers le GPU
  - Lancement **asynchrone** du *kernel* de calcul basé sur une *grille* 3D de *blocs* 3D de *threads* CUDA
    → Programmation « thread centrée » (**SPMD**) du kernel CUDA
  - Transfert des résultats du GPU vers le CPU



(CUDA Programming Guide 2.3)

Stream Computing

- Paradigme de programmation parallèle sur lesquels sont basés les GPU
- 1 ou plusieurs « flots » de données (*streams*) en entrée (E1, E2, E3) et en sortie (S1, S2)
- On applique la même fonction K (*kernels*) à chaque élément :

  $$K(E1(i), E2(i), E3(i)) \rightarrow (S1(i), S2(i))$$

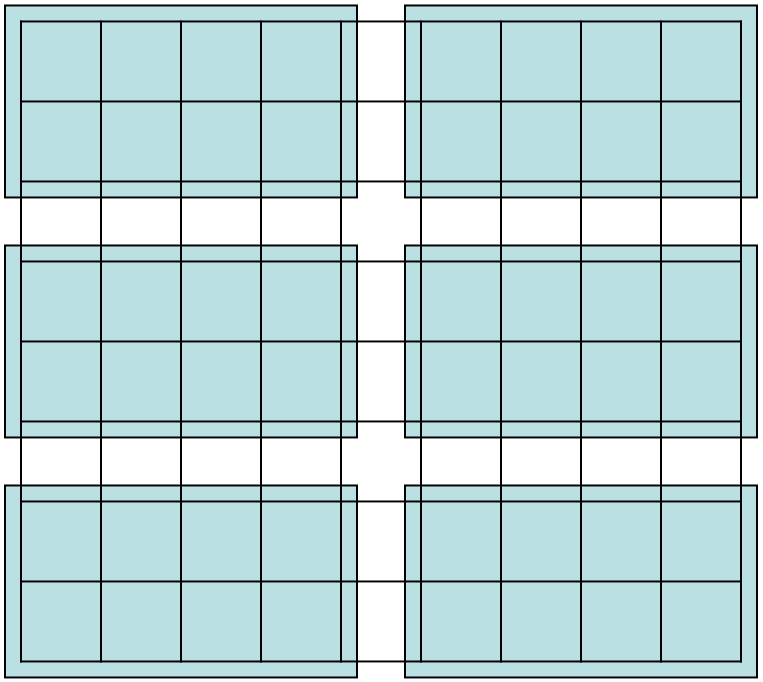
- Facilite
  - ➤ Remplissage des pipelines
  - ➤ Recouvrement des latences mémoires et arithmétiques
- Initialement conçu pour traitement d'images et de vidéos, traitement du signal ...
- Modèle « assoupli » pour calcul généraliste sur GPU

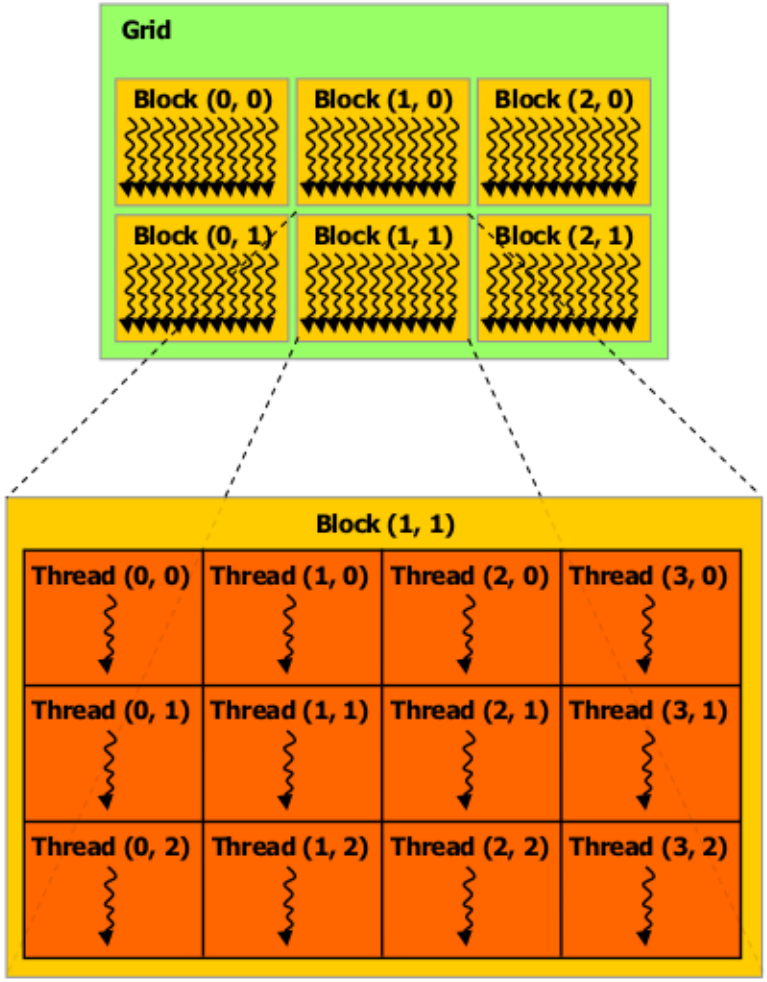
CUDA

- CUDA : *Compute Unified Device Architecture*
  = Langage de programmation GPU + runtime
- Langage CUDA = C++ avec extensions CUDA
  - ➤ Toutes les fonctionnalités C++ ne sont pas supportées (voir CUDA C Programming Guide)
  - ➤ Templates C++ supportées → intérêt pour le HPC
- Différentes versions :
  - ➤ CUDA 1.X et 2.X
  - ➤ CUDA 3.X (3.0, 3.1, 3.2) : pour architecture Fermi
  - ➤ CUDA 4.X (4.0, 4.1, 4.2)
  - ➤ CUDA 5.0 et 5.5 : pour architecture Kepler

Indexation des threads et des blocs

Exemple avec des données 2D :
→ grille 2D de blocs 2D :
bloc (x,y) / thread (x,y)





(CUDA Programming Guide)

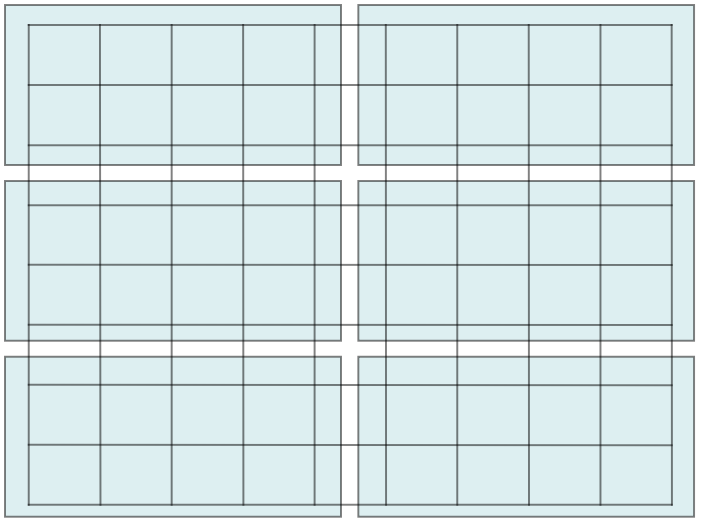
Indexation des threads et des blocs (2)

gridDim.x, gridDim.y, gridDim.z :
nombre de blocs dans chaque dimension de la grille

blockDim.x, blockDim.y, blockDim.z :
nombre de threads dans chaque dimension du bloc

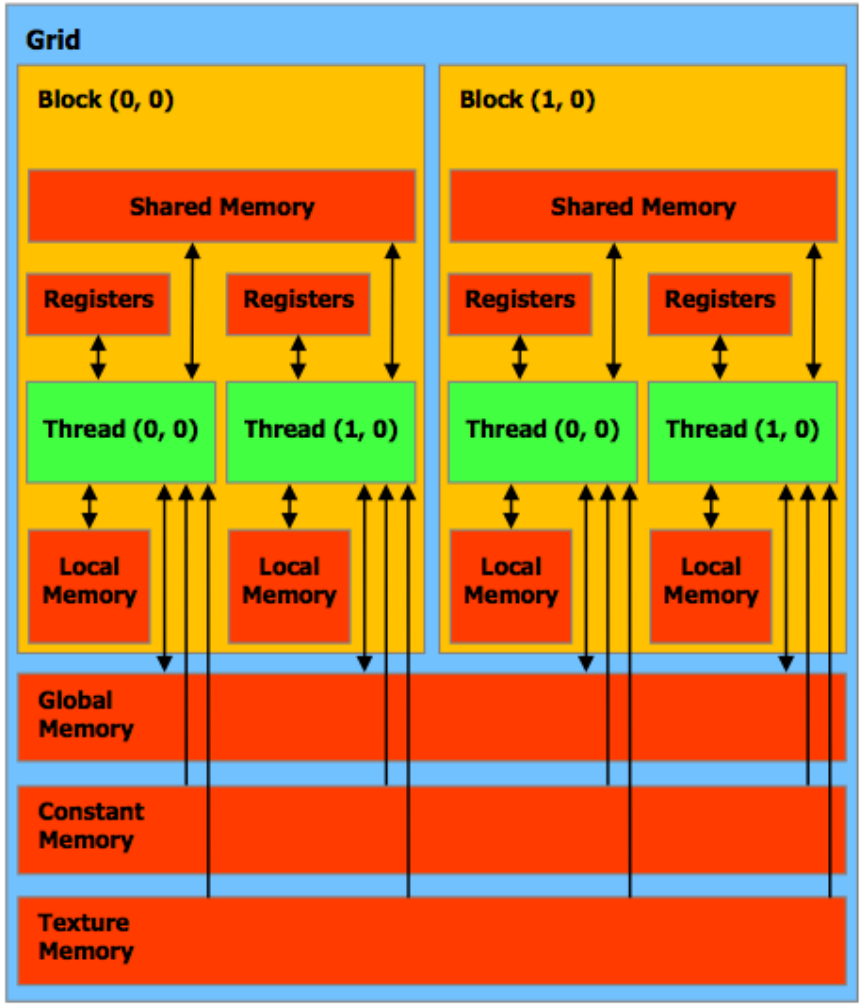
blockIdx.x, blockIdx.y, blockIdx.z :
indice du bloc dans chaque dimension de la grille, avec
$0 \leq$ blockIdx.x < gridDim.x

threadIdx.x, threadIdx.y, threadIdx.z :
indice du thread dans chaque dimension de son bloc, avec
$0 \leq$ threadIdx.x < blockDim.x



gridDim.x = 2 blockDim.x = 5
gridDim.y = 3 blockDim.y = 2
gridDim.z = 1 blockDim.z = 1

Déploiement des blocs et threads CUDA sur l'architecture GPU



(NVIDIA)

- Chaque bloc de threads est placé sur 1 MP (plusieurs blocs par MP possible)
- Aucune garantie sur l'ordre d'exécution des blocs par le runtime CUDA
  → pas (ou très peu) de dépendances possibles entre les blocs
  → pas de barrière de synchronisation entre les blocs de la grille
- Tous les threads ont accès à la mémoire globale
- Mémoires globale et constante : persistantes entre les appels de kernels d'une même application

Déploiement des blocs et threads CUDA sur l'architecture GPU (2)

- Les threads du blocs s'exécutent sur les 32 SP du MP (Fermi)
- Les threads d'un même bloc peuvent partager des données via la mémoire partagée
  → coopération possible

  → barrières de synchronisation au sein d'un bloc de thread
- Exécution **SIMD** par *warp* de 32 threads → alors que chaque thread est programmé pour effectuer un calcul **scalaire**
- Aucune garantie sur l'ordre d'exécution des warps au sein d'un bloc par le runtime CUDA
- Constitution des warps : regroupement des threads par groupe de 32 en suivant d'abord la dimension 'x', puis la dimension 'y', puis la dimension 'z'

Exemple : kernel CUDA de multiplication de matrices

```
__global__ void matmulKernel(float* d_A,
                             float* d_B,
                             float* d_C,
                             int n) {
  unsigned int j = blockDim.x*blockIdx.x+threadIdx.x;
  unsigned int i = blockDim.y*blockIdx.y+threadIdx.y;

  if (i < n && j < n){
    float temp=0;

    for(int k=0; k<n; k++)
      temp = temp + d_A[i * n + k] * d_B[k * n + j];

    d_C[i * n + j] = temp;
  }
}
```

Compilation

- Compilateur pour CUDA : nvcc
- Fichier CUDA : .cu ou .cuh
- Plusieurs stratégies :
  - Code source : *.cu *.cuh :
    - Tous les fichiers sont compilés avec nvcc
    - Pas applicable à des applications complexes
  - Code source : *.{c,cpp}, *.h, *.cu, *.cuh :
    - Fichiers *.{c,cpp}, *.h → compilateur gcc/g++, icc/icpc …
    - Fichiers *.cu, *.cuh → compilateur nvcc
    - Attention à l'édition de lien entre C et CUDA : nommage des fonctions « à la C++ » par nvcc → extern "C "

Extensions de langage CUDA (1)

- Pour les fonctions :

__global__ : appel sur CPU, exécution sur GPU (kernel)

__host__ : appel sur CPU, exécution sur CPU (défaut)

__device__ : appel sur GPU, exécution sur GPU

- Attention :
  - Un kernel a toujours “void” comme type de retour
  - Récursivité
    - __global__ : non supportée (mais voir *Dynamic parallelism*)
    - __device__ : possible à partir de compute capability 2.0
  - Pas de déclaration de variable *static* dans une fonction GPU
  - Pas de fonction avec un nombre d’arguments variable
  - __device__ et __host__ ne sont pas incompatibles

Extensions de langage CUDA (2) : déclarations de variables

- Variable automatique sans qualificateur déclarée dans le code GPU → généralement placée en **registres**
- Qualificateur **__device__**
  - Déclaration d'une variable sur le GPU depuis le code hôte
  - Sans __shared__ et __constant__ : variable en mémoire globale, et accessible en lecture/écriture par le GPU et le CPU (cudaMemcpy{To/From}Symbol(), ...)
- Qualificateur **__shared__**
  - Déclaration d'une variable dans la mémoire partagée d'un bloc de threads (durée de vie de la variable = durée de vie du bloc)
  - Déclaration dans le code GPU ou dans le code hôte
  - Accessible uniquement par les threads du bloc
  - Mémoire partagée = « Cache » géré par le programmeur
- Qualificateur **__constant__**
  - Déclaration d'une variable dans la mémoire constante du GPU
  - Déclaration dans le code GPU ou dans le code hôte
  - Accessible par le CPU en lecture/écriture (cudaMemcpy{To/From}Symbol(), ...) et par le GPU en lecture

Les différentes mémoires et accès en CUDA

Type de mémoire	*Localisation*	*Cache*	*Type d'accès*	*Accès*	*Durée de vie*
Register	On chip		RW	Par un thread	thread
Shared	On chip		RW	Tous les threads d'un bloc	bloc
Local	Off-chip	Yes (Fermi)	RW	Par un thread	thread
Global	Off-chip	Yes (Fermi)	RW	Tous les threads + CPU	Programme
Constante	Off-chip	Yes	R	Tous les threads + CPU	Programme
Texture	Off-chip	Yes	R	Tous les threads + CPU	programme

Extensions de langage CUDA (3)

- Variables prédéfinies :
  - *threadIdx* : coordonnées d'un thread
  - *blockIdx* : coordonnées d'un bloc
  - *blockDim* : dimensions d'un bloc
  - *gridDim* : dimensions d'une grille
  - Toutes de type *dim3*
    - 3 champs : x,y,z
    - Par défaut, chaque champ est initialisé à 1.
    - Exemple :
      - « dim3 nbthead(256); » → (256,1,1)
      - « dim3 nbbloc(128,128); » → (128,128,1)
        → 4 millions de threads

Extensions de langage CUDA (4)

- Fonctions « *intrinsics* » :
  - `__syncthreads()` → barrière de synchronisation pour tous les threads d'un bloc
- API :
  - Allocation mémoire, transferts de données, synchronisations hôte ↔ GPU, …
  - Exemple : `cudaDeviceSynchronize()` → attente depuis l'hôte de la fin de toutes les opérations lancées sur le GPU
- Lancement d'un kernel depuis l'hôte :

```
matmulKernel<<<tailleGrille,threadsParBloc>>>(d_A,d_B,d_C,N);
```

Allocation dynamique de mémoire sur GPU et transferts de données CPU ↔ GPU

- 2 modes de gestion dynamique de mémoire sur GPU :
  - *CUDA arrays* → mémoire de texture
  - *Linear memory* → allocation dynamique de mémoire sur le GPU (en mémoire globale)
    - `cudaError_t cudaMalloc (void **devPtr, size_t size)`
    - `cudaError_t cudaFree (void *devPtr)`

  → Transferts de données (synchrones) CPU ↔ GPU :
    - `cudaError_t cudaMemcpy (void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)`
    - Valeurs possibles pour `kind` :

  `cudaMemcpyHostToDevice / cudaMemcpyHostToHost / cudaMemcpyDeviceToHost / cudaMemcpyDeviceToDevice`

Transferts de données CPU ↔ GPU (suite)

- Pour les variables déclarées statiquement sur le GPU :
  - Copie CPU -> GPU

```
cudaError_t cudaMemcpyToSymbol (const char *symbol,
  const void *src, size_t count, size_t offset = 0,
  enum cudaMemcpyKind kind = cudaMemcpyHostToDevice)
```

    Autre possibilité : `cudaMemcpyDeviceToDevice`

  - Copie GPU -> CPU

```
cudaError_t cudaMemcpyFromSymbol (void *dst,
    const char *symbol, size_t count, size_t offset = 0,
    enum cudaMemcpyKind kind = cudaMemcpyDeviceToHost)
```

    Autre possibilité : `cudaMemcpyDeviceToDevice`

  - Voir aussi : `cudaGetSymbolAddress()`

Multiplication de matrices (suite)

```
#define N 1024
#define TAILLE_BLOC_X 16
#define TAILLE_BLOC_Y 16

int main(){
  float *A, *B, *C, *C_ref_CPU, *d_A, *d_B, *d_C;
  int taille_totale=N*N*sizeof(float), i,j;

  /* Allocation CPU : */
  A=(float*)malloc(taille_totale); B=(float*)malloc(taille_totale);
  C=(float*)malloc(taille_totale); C_ref_CPU=(float*)malloc(taille_totale);
  /* Initialisation : */
  fillMatriceFloat(A, N);  fillMatriceFloat(B, N);

  /* Allocation GPU : */
  cudaMalloc((void **) &d_A, taille_totale);  cudaMalloc((void **) &d_B, taille_totale);
  cudaMalloc((void **) &d_C, taille_totale);

  /* Transferts CPU → GPU (synchrones) : */
  cudaMemcpy(d_A, A, taille_totale, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, B, taille_totale, cudaMemcpyHostToDevice);
```

Multiplication de matrices (suite)

```
/* Lancement de kernel (asynchrone) : */
 dim3 threadsParBloc(TAILLE_BLOC_X, TAILLE_BLOC_Y);
 dim3 tailleGrille(ceil(N/(float) TAILLE_BLOC_X), ceil(N/(float) TAILLE_BLOC_Y));
 matmulKernel<<< tailleGrille, threadsParBloc>>>(d_A, d_B, d_C, N);

 /* Transfert GPU → CPU (synchrone) : */
 cudaMemcpy(C, d_C, taille_totale, cudaMemcpyDeviceToHost));

 /* Vérification : */
 matmul_CPU(A, B, C_ref_CPU, N);
 for(i=0;i<N;i++)
      for(j=0;j<N;j++)
        if (fabsf(C[i*N+j] – C_ref_CPU[i*N+j]) > 0.001)
          printf("%4d %4d  h %le d %le\n", i, j, C_ref_CPU[i*DIM+j], C[i*DIM+j]);

 /* Libération mémoire GPU et CPU : */
 cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
 free(A); free(B); free(C); free(C_ref_CPU);
}
```

Gestion des erreurs

- Toutes les fonctions CUDA exécutées depuis l'hôte retourne une erreur :
  - Type : `cudaError_t` → si ok : `cudaSuccess`
- Pour les opérations asynchrones :
  - Le code d'erreur retourné ↔ uniquement lancement de l'opération
  - CUDA maintient une variable erreur :
    - Initialisée à `cudaSuccess`, puis modifiée à chaque fois qu'une erreur se produit
    - `cudaGetLastError()` retourne la valeur cette variable (et remise à `cudaSuccess`)
    - Vérification d'une opération asynchrone donnée :
      - Synchronisation nécessaire juste après l'appel
      - 2 modes : « debug » et « production »
    - Obtenir le message d'erreur à partir de la variable `err` de type `cudaError_t : cudaGetErrorString(err)`
- Voir macros CUDA_ERROR et CUDA_SYNC_ERROR en TP

Performances et optimisations GPU

- **Ordonnancement des threads sans surcoût sur GPU :**
  - Toutes les données de chaque thread stockées dans les registres du MP
  - « Surcharger » chaque MP de threads pour
    - Recouvrir les accès (latences) mémoire :
      - mémoire partagée (~4 cycles)
      - mémoire globale (entre 400 et 800 cycles)
    - Recouvrir les latences arithmétiques : ~24 cycles pour la longueur du pipeline sur GPU

→ besoin de (plusieurs dizaines de) milliers de threads sur GPU

Occupancy

- ***Occupancy*** = *ratio of the number of* ***active warps*** *per multiprocessor to the maximum number of possible active warps (CUDA C Best Practices Guide)*

  → Il faut un niveau d'*occupancy* suffisant (~ 50%)

  → Au dessus de ce niveau, une meilleure *occupancy* n'implique pas forcément de meilleures performances
- Limite au nombre de warps par MP :
  - ➤ Pas assez de warps dans la grille → problème trop « petit » pour le GPU
  - ➤ Nombre de registres par thread → à minimiser
  - ➤ Taille de mémoire partagée consommée (par bloc) –> à minimiser

Choix du nombre de threads par bloc

- Nombre de threads par bloc
  - ➤ Doit être de préférence un multiple de la taille du warp
    - Exécution SIMD implicite pour le programmeur
    - Changement (futur) de taille de warp potentiellement transparent pour l'utilisateur
  - ➤ Augmentation du nombre de threads par bloc
    - Peut améliorer les performances (mutualisation de certaines données/ressources au sein du bloc)
    - Ou pas : limites matérielles (nombre de threads par bloc, nombre de registres et taille de la mémoire partagée du MP), *occupancy*, __syncthreads() ...
- Nombre optimal de threads par bloc : recherche exhaustive (→ *auto-tuning*) ou *CUDA GPU Occupancy Calculator*
- Taille de la grille → dépend (généralement) des données une fois la taille des blocs fixée + avoir au moins 1 bloc par MP

Limites matérielles pour les tailles des blocs et de la grille

	Compute capability			
	1.0 et 1.1	1.2 et 1.3	2.x	3.0 et 3.5
Dimension maxi x d'une grille	65535	65535	65535	$2^{31}-1$
Dimension maxi y ou z d'une grille	65535			
Nombre maxi de thread par bloc	512		1024	
Dimension maxi x ou y d'un bloc	512		1024	
Dimension maxi z d'un bloc	64			
Taille d'un warp	32			
Nombre maxi de blocs résident / MP	8			16
Nombre maxi de warp résident / MP	24	32	48	64
Nombre maxi de threads / MP	768	1024	1536	2048
Nombre de registres 32-bit / MP	8192	16384	32768	65536
Taille maxi de la mémoire partagée	16 Ko		48 Ko	

Divergence de calcul

- GPU → exécution partiellement SIMD
  - ➤ Implicite pour le programmeur, mais attention à l'impact de la divergence de calcul entre les threads d'un même warp :
    - Branchement conditionnel *if … then … else*
      → **sérialisation de l'exécution des branches *then* et *else***
    - Boucles avec un nombre d'itérations différent par thread
      → à la sortie de la boucle : **attente des threads qui sont encore en train d'exécuter la boucle**
  - ➤ Remarque : sur GPU, exécution des instructions dans l'ordre (*in-order*), et pas de prédiction de branchement
  - ➤ Mécanisme de *prédication* (uniquement si quelques instructions) : pipelines non vidés grâce à un prédicat (= masque)
- Alternatives possibles :
  - ➤ Si divergence connue statiquement : réorganiser le flot d'instructions et/ou les structures de données
  - ➤ Sinon : modifier l'algorithme...

Divergence mémoire

- Accès « **amalgamé** » (*coalesced memory access*) en mémoire globale :
  - Le warp accède à des données **contiguës et alignées**
  - Tous les accès mémoire du warp sont alors effectués en 1 seule transaction mémoire
- Sur architecture Fermi : cache pour mémoire globale
  - Lignes de cache de 128 octets (= 32 x 4 octets)
  - Lignes de cache alignées sur 128 octets
  - Si accès non amalgamés : chargement d'autant de lignes de cache que nécessaires
- Remarque : avec cudaMalloc() ou __device__ → adresses alignées sur au moins 256 octets

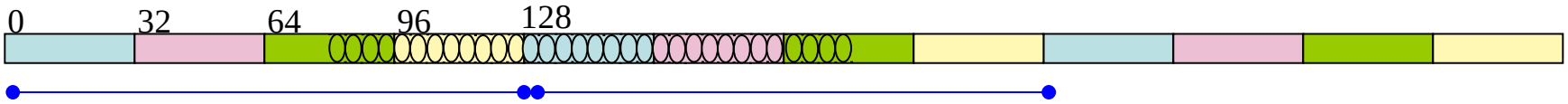
Gestion des accès mémoire sur Fermi

Accès mémoire (4 octets) par un thread d'un warp

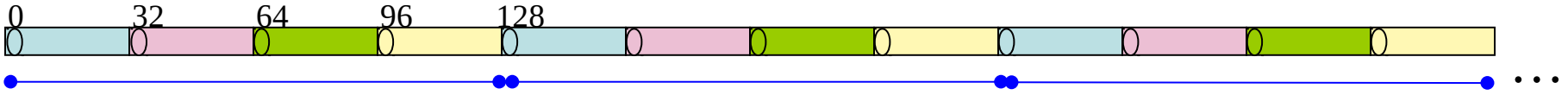
Ligne(s) de cache effectivement chargée(s)

Données contiguës et alignées : 1 accès de 128 octets

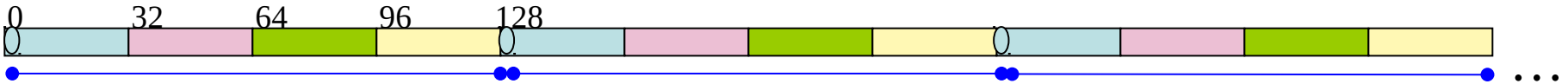
Données contiguës non alignées : 2 accès de 128 octets



Données espacées de 32 octets et alignées : 8 accès de 128 octets



Données espacées de 128 octets : 32 accès de 128 octets



Etude du comportement d'un noyau de calcul

Noyau de calcul : un programme qui élève à la puissance *K* les éléments d'un tableau 1D (d'après un exemple de W. Kirschenmann).

Quelques restrictions :

N : nombre d'éléments du tableau $N=2^n$

K : une puissance de 2

Cœur de calcul en CUDA :

```
__device__ __host__ inline void
  myPow(float *x, const unsigned int & K){
  float result=1;
  for (int i = 0 ; i < K ; i++)
    result *= *x;
  *x = result;
}
```

Kernel CUDA

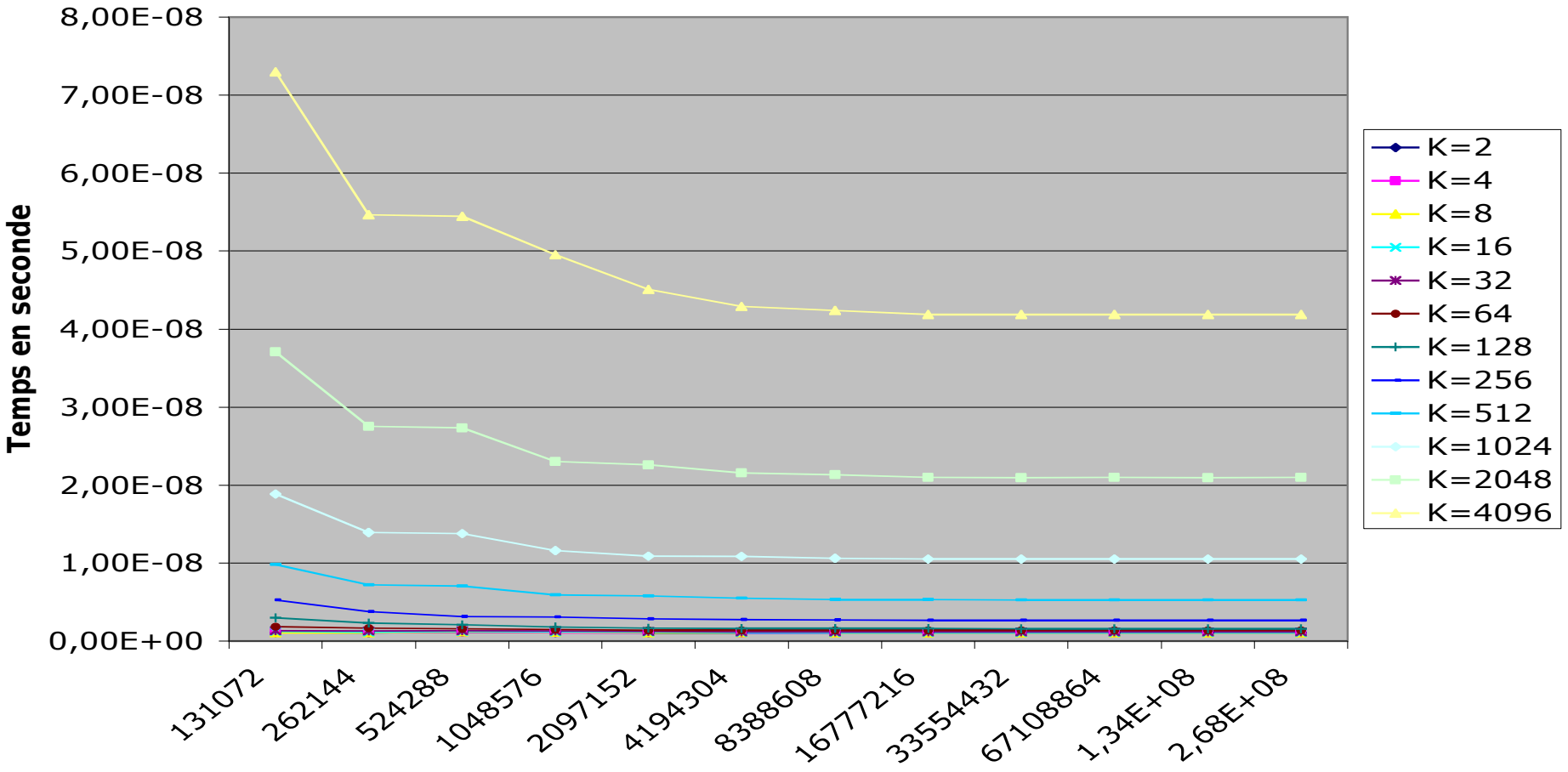
Chaque thread calcule 16 éléments du tableau :

```
_global__ void puissanceKernel(float* X,
                    const unsigned int K,
                    const unsigned int nbElts){
  const int begin =
   16*(threadIdx.x+blockIdx.x*blockDim.x);

  myPow(&X[begin], K);
  myPow(&X[begin+1], K);
  myPow(&X[begin+2], K);
   …
   myPow(&X[begin+14], K);
  myPow(&X[begin+15], K);
}
```

Le programme principal

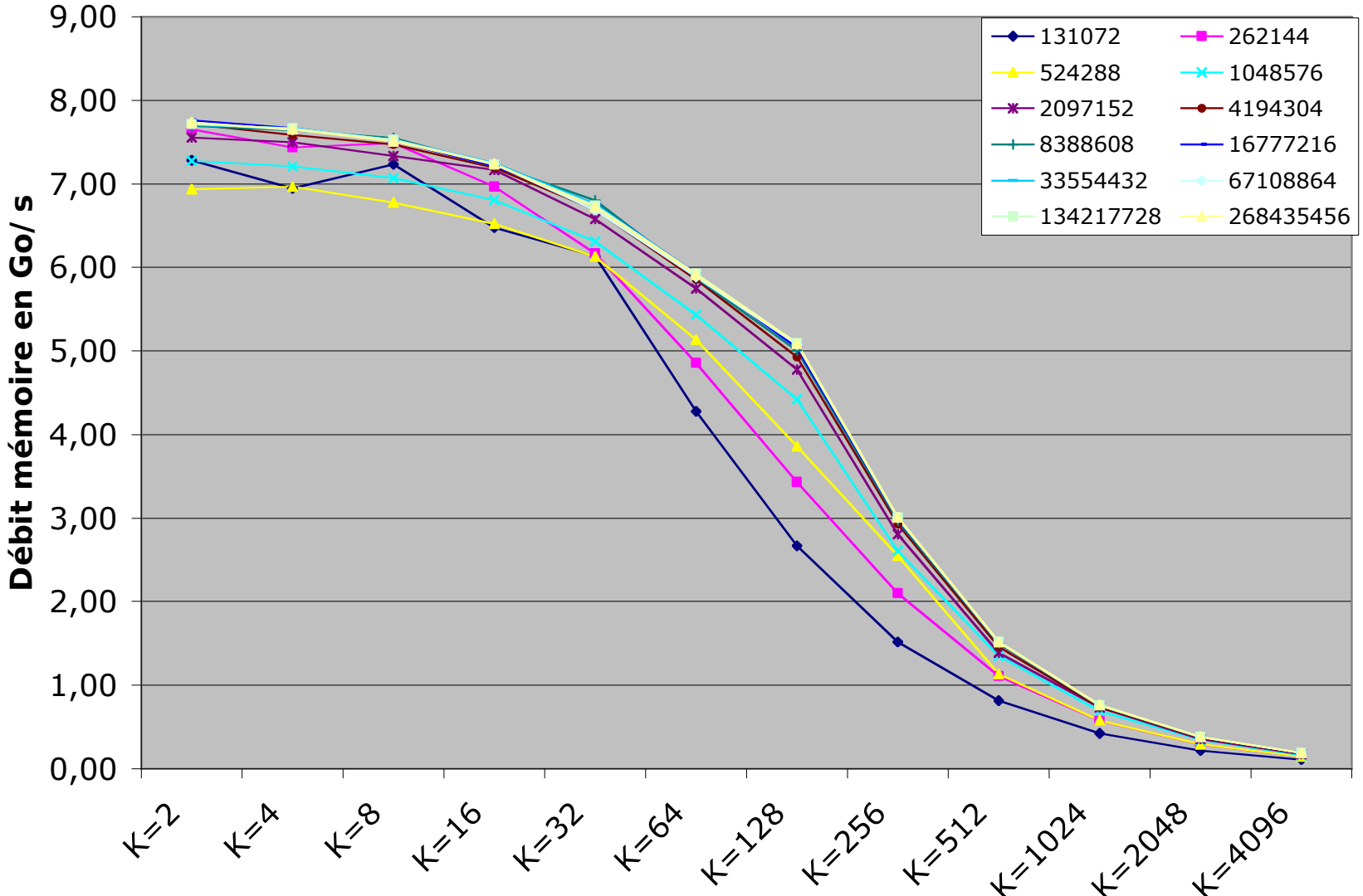
```
…
vec=(float*)malloc(taille_totale);
res =(float*)malloc(taille_totale);
fillVectorFloat(vec, N);
cudaMalloc((void **) &d_vec, taille_totale);
cudaMemcpy(d_vec,vec,taille_totale,cudaMemcpyHostToDevice);

dim3 threadsParBloc(512,1,1);
dim3 tailleGrille((N/512)/16,1,1);

my_gettimeofday(&tic);
puissanceKernel<<<tailleGrille,threadsParBloc>>>(d_vec, K,
  N);
cudaDeviceSynchronize();
my_gettimeofday(&toc);

cudaMemcpy(res,d_vec,taille_totale,cudaMemcpyDeviceToHost));
…
```

Temps de calcul par élément (en fonction de N)

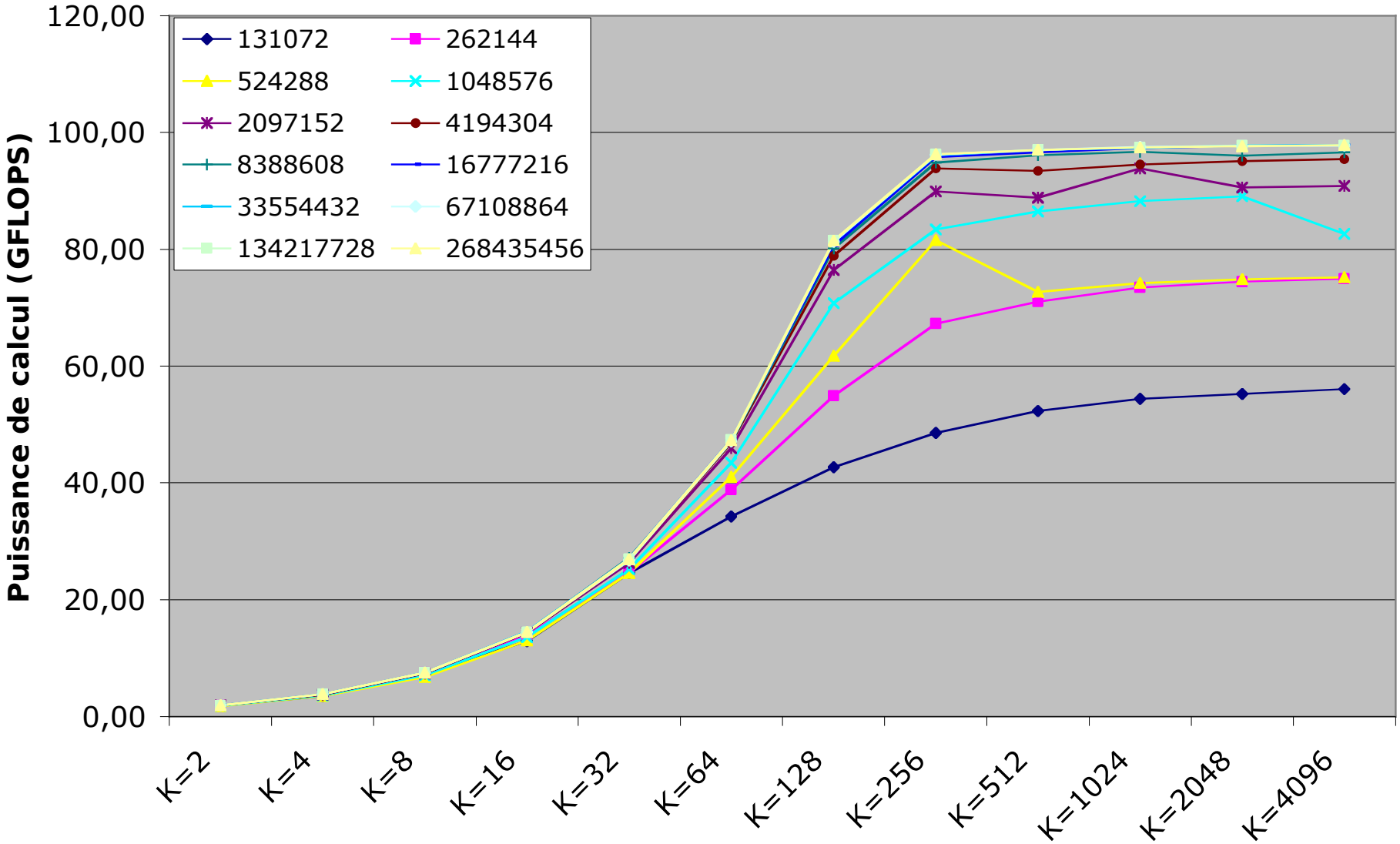


Débit mémoire soutenu de notre application

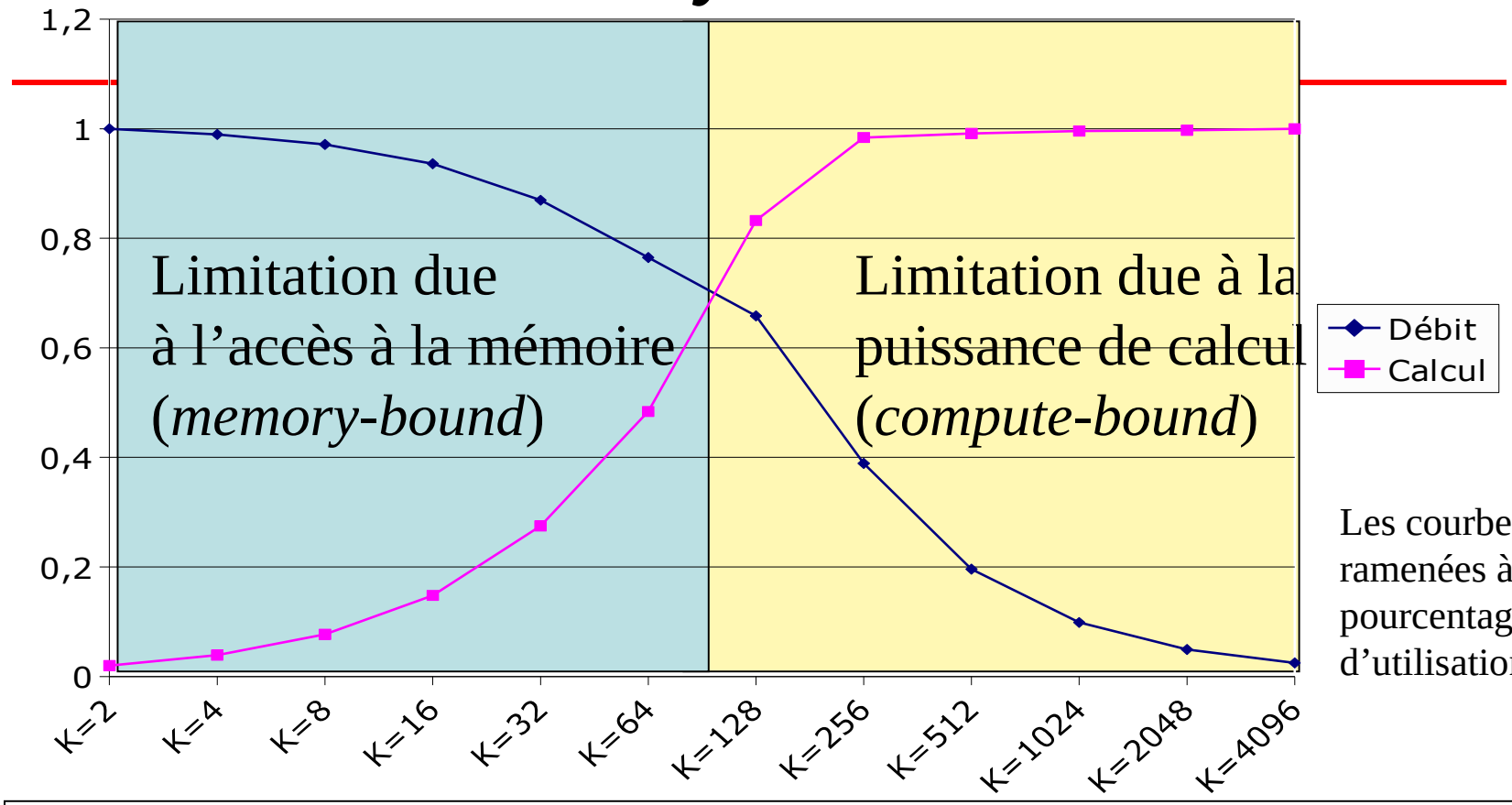
- Nombre d'accès mémoire (*loads* et *stores*)/ temps d'execution (en Go/s)



Puissance de calcul soutenue de notre application (en Gflop/s)



La synthèse



Les courbes sont ramenées à des pourcentages d'utilisation

Définition de l'**intensité arithmétique** (ou intensité de calcul) :
I= nombre d'opérations / nombre d'accès mémoire

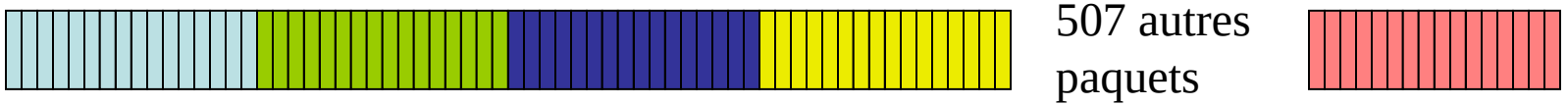
→ A comparer avec les caractéristiques du GPU : permet de savoir s'il faut optimiser en priorité les accès mémoire ou les calculs

→ Pour notre programme de test : I=K/8

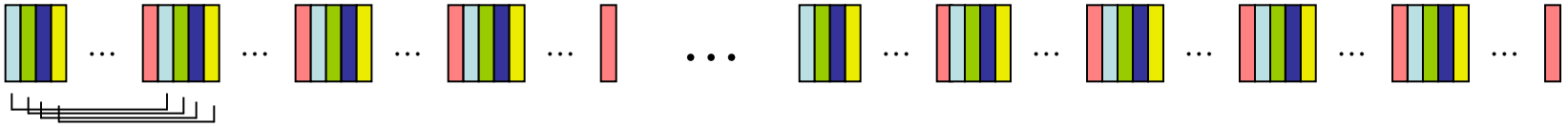
Réorganisation des accès mémoires

- Exemple avec 262144 éléments = 512*16*32
- Exécution par blocs de 512 threads → 32 blocs de threads
- Chaque thread traite un « paquet » de 16 éléments

Version 1 16 éléments x 512 paquets



Version 2



Saut de 512 éléments

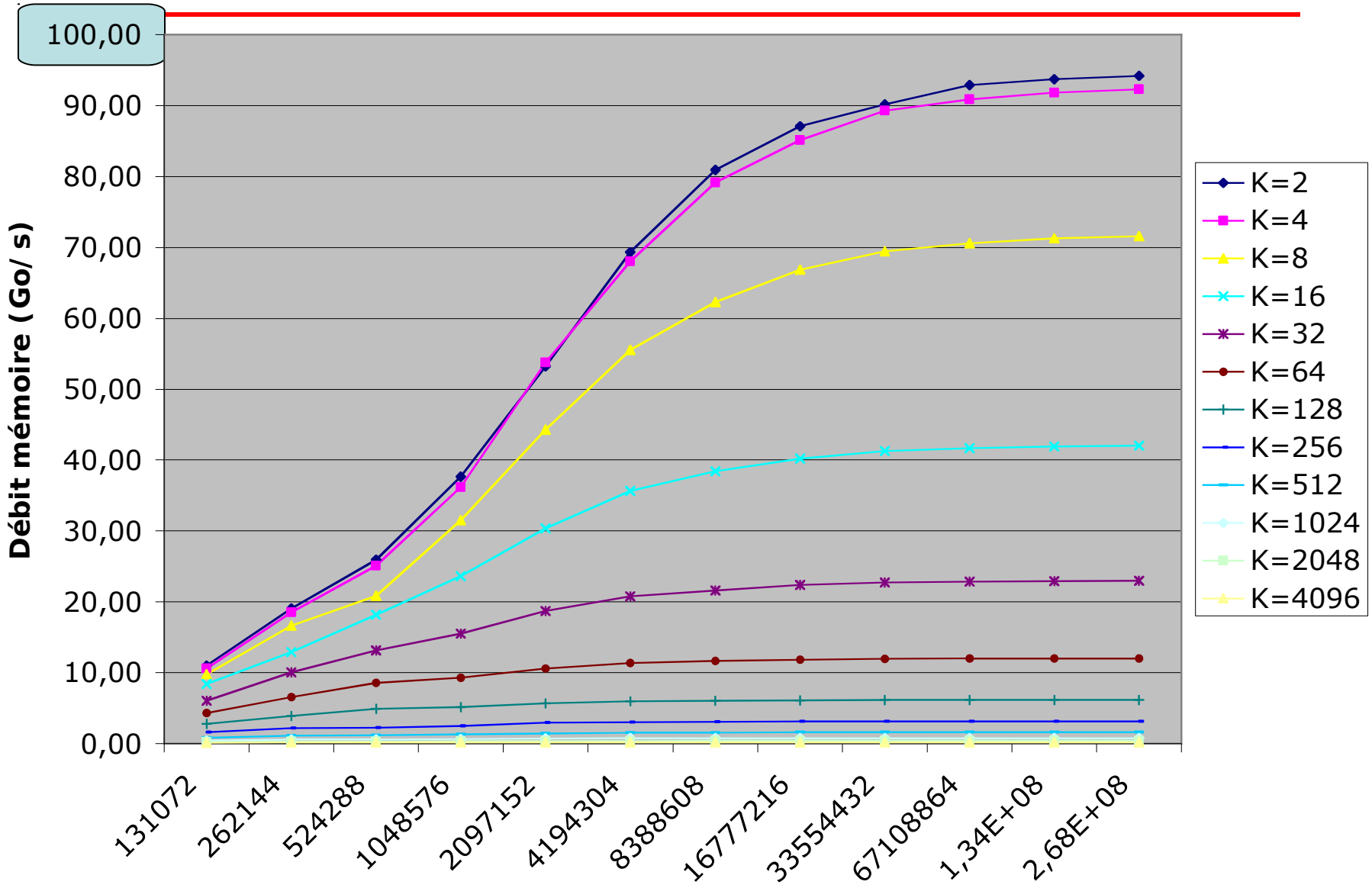
La couleur indique quel thread exécute le calcul d'un élément du tableau

Optimisation des accès mémoire (V2)

```
/*** V1 ***/
_global__ void puissanceKernel(
   float* X,
   const unsigned int K,
   const unsigned int nbElts){
  const int begin=16*(threadIdx.x+
        blockIdx.x*blockDim.x);

  myPow(&X[begin], K);
  myPow(&X[begin+1], K);
  myPow(&X[begin+2], K);
  myPow(&X[begin+3], K);
   …
  myPow(&X[begin+12], K);
  myPow(&X[begin+13], K);
  myPow(&X[begin+14], K);
  myPow(&X[begin+15], K);
}
```

```
/*** V2 ***/
__global__ void puissanceKernel(
   float* X,
   const unsigned int K,
   const unsigned int nbElts){
  const int begin = threadIdx.x+16*
                blockIdx.x*blockDim.x;

  myPow(&X[begin], K);
  myPow(&X[begin+1*blockDim.x], K);
  myPow(&X[begin+2*blockDim.x], K);
  myPow(&X[begin+3*blockDim.x], K);
   …
  myPow(&X[begin+12*blockDim.x], K);
  myPow(&X[begin+13*blockDim.x], K);
  myPow(&X[begin+14*blockDim.x], K);
  myPow(&X[begin+15*blockDim.x], K);
}
```

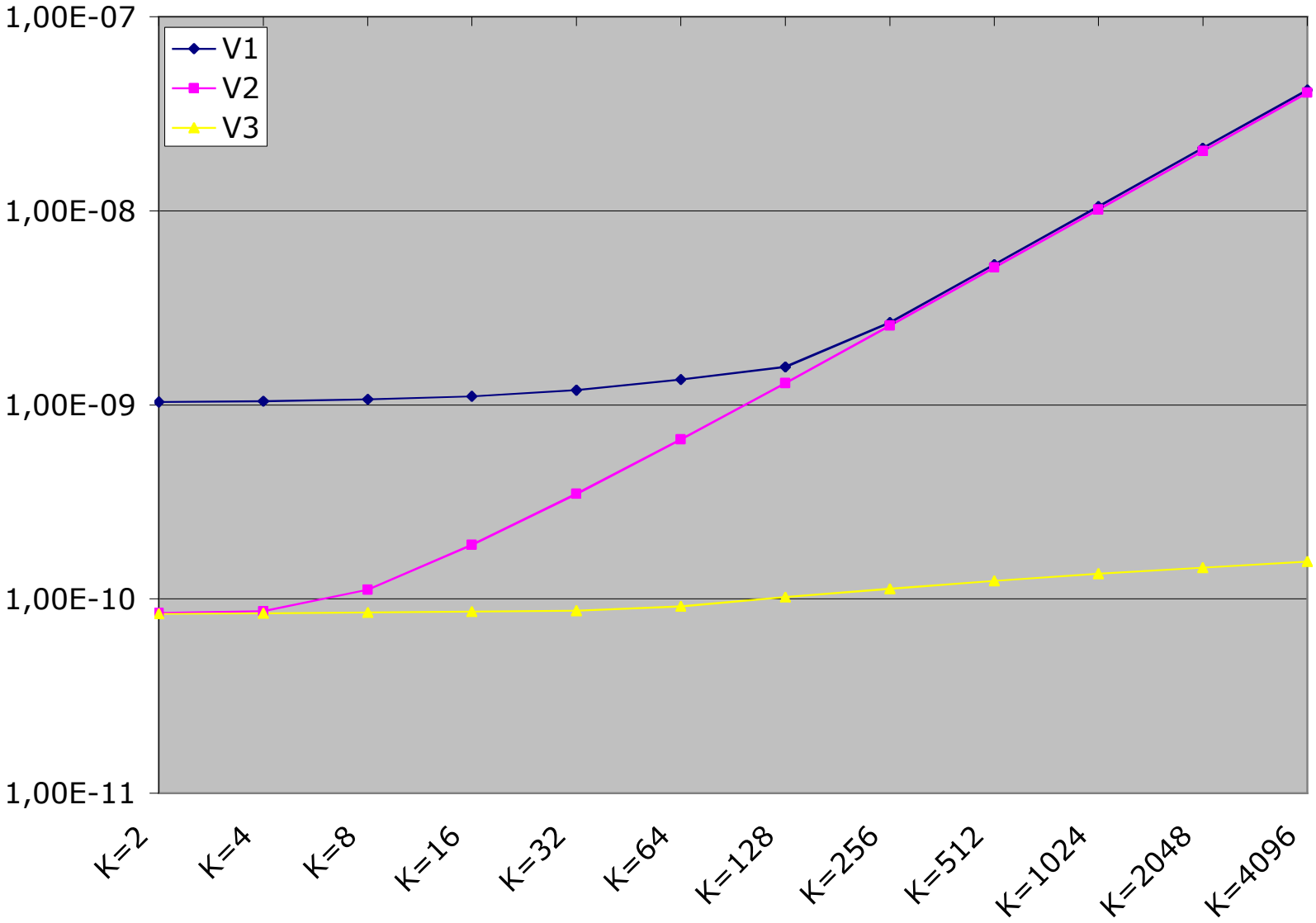
Débit mémoire soutenu (V2)



Optimisation du calcul (V3)

```
/*** V2 ***/
__device__ __host__ inline void myPow(float *x,
                                      const unsigned int & K){
  float result=1;
  for (int i = 0 ; i < K ; i++)
    result *= *x;
  *x = result;
}

 /*** V3 (rappel : K est une puissance de 2) ***/
 __device__ __host__ inline void myPow(float *x,
                                       const unsigned int & K){
   float result=*x;
   for (int i = 1 ; i < K ; i*=2)
     result *= result;
   *x=result
 }
```

Temps de calcul pour V1/V2/V3



Utilisation de la mémoire partagée : produit matriciel

- Version sans mémoire partagée (présentée précédemment)
  - ➤ Chaque thread calcule un élément de la matrice C sur le GPU (d_C), indépendamment des autres threads de son bloc
  - ➤ Hypothèses : matrices carrées (n,n), et blocs de threads carrés de dimensions (TAILLE_BLOC_DIM, TAILLE_BLOC_DIM) (TAILLE_BLOC_DIM fixée statiquement, avec 'n' multiple de TAILLE_BLOC_DIM)

```
#define TAILLE_BLOC_DIM 16
__global__ void matmulKernel(float* d_A, float* d_B, float* d_C, int n) {
  unsigned int j = TAILLE_BLOC_DIM*blockIdx.x+threadIdx.x;
  unsigned int i = TAILLE_BLOC_DIM*blockIdx.y+threadIdx.y;
  float temp=0;

  for(int k=0; k<n; k++)
    temp = temp + d_A[i * n + k] * d_B[k * n + j];

  d_C[i * n + j] = temp;
}
```

Utilisation de la mémoire partagée : produit matriciel (suite)

- Version avec mémoire partagée
  - Chaque bloc de thread calcule un sous-bloc de la matrice C via la mémoire partagée : les threads du bloc coopèrent pour le chargement des sous-blocs de A et de B en mémoire partagée
  - Mêmes hypothèses

```
__global__ void matmulKernel2(float* d_A, float* d_B, float* d_C, int n) {
  // Numéro de ligne ('ib') et de colonne ('jb') du sous-bloc C(ib, jb) :
   int ib = blockIdx.y;
   int jb = blockIdx.x;

  // Indice de ligne ('i') et de colonne ('j') dans le sous-bloc C(ib, jb) :
  int i = threadIdx.y;
  int j = threadIdx.x;

  float tmp = 0;
```

Utilisation de la mémoire partagée : produit matriciel (suite)

```
// Boucle sur les sous-blocs de A et B
for (int m = 0; m < (n / TAILLE_BLOC_DIM); ++m) {
  // Adresse du premier élément du sous-bloc A(ib, m) et du sous-bloc B(m, jb) :
  float *p_d_A = d_A + (n*TAILLE_BLOC_DIM*ib + TAILLE_BLOC_DIM*m);
  float *p_d_B = d_B  + (n*TAILLE_BLOC_DIM*m + TAILLE_BLOC_DIM*jb);

  // Mémoire partagée pour stocker les sous-blocs de A et B :
  __shared__ float As[TAILLE_BLOC_DIM][TAILLE_BLOC_DIM];
  __shared__ float Bs[TAILLE_BLOC_DIM][TAILLE_BLOC_DIM];

  // Chaque thread charge un élément de A et de B (globale → partagée) :
  As[i][j] = p_d_A[n * i + j];
  Bs[i][j] = p_d_B[n * i + j] ;

  // Synchronisation avant calcul nécessaire entre les threads du bloc :
  __syncthreads();
```

Utilisation de la mémoire partagée : produit matriciel (suite)

```
    // Multiplication des sous-blocs A(ib, m) et B(m, jb) :
    for (int e=0; e<TAILLE_BLOC_DIM; ++e)
            tmp += As[i][e] * Bs[e][j];

    // Synchronisation pour garantir que le calcul est terminé
    // pour tous les threads du bloc avant de ré-écrire dans As et Bs
    // (à l'itération suivante) :
    __syncthreads();
    }

  // Chaque thread écrit son élément de C en mémoire globale :
  d_C[n * (TAILLE_BLOC_DIM * ib + i) + TAILLE_BLOC_DIM * jb + j) = tmp;
}
```

Utilisation de la mémoire partagée

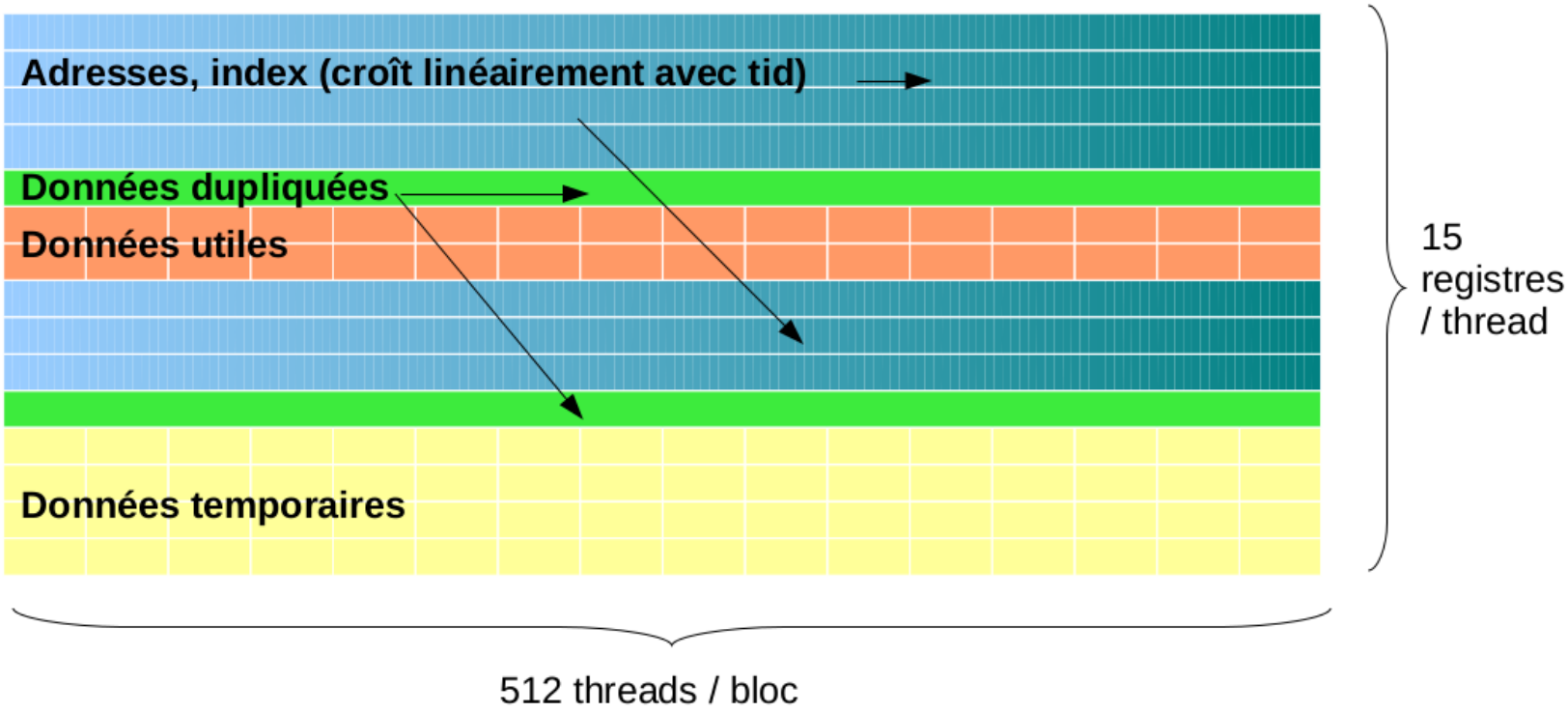
- Comparaison
  - Version sans mémoire partagée : chaque élément de A et de B est lu 'n' fois depuis la mémoire globale.
  - Version avec mémoire partagée : chaque élément de A et de B n'est lu que (n / TAILLE_BLOC_DIM) fois depuis la mémoire globale.
- Possible de fixer la taille de la mémoire partagée à l'exécution :
  - Dans kernel CUDA : extern __shared__ float buf[];
  - Appel du kernel :
    my_kernel<<<tailleGrille, threadsParBloc, tailleMemPartagee>>>(... args ...);
    où tailleMemPartagee = taille totale de mémoire partagée pour un bloc de thread
- Sur Fermi :
  - 32 bancs en mémoire partagée
  - Bancs entrelacés par mot mémoire de 32 bits
- Attention aux conflits entre bancs mémoire : voir documentation CUDA en fonction de la *compute capability*

ILP et GPU
(d'après S. Collange et V. Volkov)

- Quel grain de calcul associé à chaque thread CUDA ?
  - ➤ Le plus petit grain de calcul possible ?
    - Avantages : utilisation du parallélisme de données pour recouvrir les latences (arithmétiques et mémoire)
    - Inconvénients : surcoût mémoire (stockage des données privées à chaque thread) et surcoût de gestion (initialisation, calculs d'adresse, contrôle, opérations redondantes...)
  - ➤ Grain « un peu plus gros » :
    - Parallélisme d'instructions (ou *Instruction Level Parallelism – ILP*) : parallélisme disponible au sein du flot d'instruction de chaque thread
    - Permet également de masquer les latences (remplissage des pipelines)
    - ILP + déroulage de boucle → à utiliser avec modération : attention à l'augmentation du nombre de registres par thread et au *register spilling* en *local memory* (= mémoire globale)

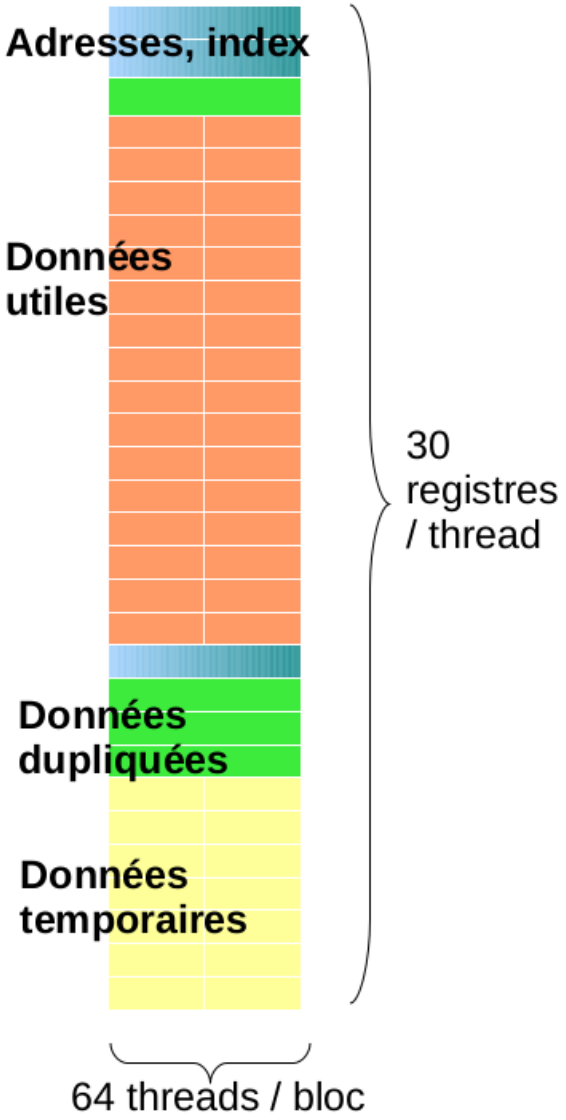
Exemple : produit matriciel
(d'après S. Collange et V. Volkov)

- D'après V. Volkov : « Programming inverse memory hierarchy : case of stencils on GPUs », ParCFD, 2010.
- Implémentation produit matriciel simple précision (SGEMM) dans CUBLAS 1.1 :
  - 512 threads / bloc, 15 registres / thread
  - 9 registres / 15 : données redondantes → seuls 2 registres réellement utiles



Moins de threads, plus d'ILP
(d'après S. Collange et V. Volkov)

- SGEMM Volkov
  - 8 éléments calculés / thread
  - Déroulage de boucle
  - Moins d'accès à la mémoire partagée,
    plus d'accès aux registres
    (meilleure bande passante)
  - 1920 registres contre 7680 pour calcul identique
  - Idem pour les opérations redondantes
- Bilan :
  - Gain de +60% en performance par rapport à CUBLAS 1.1
  - Intégré dans CUBLAS 2.0



Opérations atomiques

- Opérations atomiques possibles sur entiers 32-bit ou 64-bit en mémoire globale ou en mémoire partagée
  - Opérations arithmétiques : atomicAdd(), atomicSub(), atomicExch(), atomicMin(), atomicMax(), atomicCAS() ...
  - Opérations logiques : atomicAnd(), atomicOr() …
  - Permet une coordination (limitée) entre blocs de threads via la mémoire globale

Conseils pour l'optimisation des kernels

1) Déterminer le grain de calcul par thread ($\rightarrow$ ILP)
2) Déterminer le nombre de dimensions des blocs et de la grille (sans fixer statiquement, si possible, le nombre de threads par bloc)
3) Etudier les accès mémoire amalgamés
4) Prendre en compte la mémoire partagée
5) Minimiser le nombre de registres

Optimisation des transferts de données CPU ↔ GPU

- Utilisation de la p*age-locked memory* aussi appelée *pinned memory*
  - Fonctions d'allocation et de libération mémoire :
    - `cudaMallocHost()` (C API) / `cudaHostAlloc()` et `cudaFreeHost()`
  - Le GPU accède directement à la zone mémoire → meilleure bande passante entre CPU et GPU
- *Page-locked memory* allouée en mode *write-combining* :
  - Disponible via option de `cudaHostAlloc()`
  - Les caches CPU ne « gèrent » plus cette zone mémoire
  - Bande passante CPU ↔ GPU encore meilleure
  - Mais lectures depuis CPU très lentes

Recouvrement des transferts CPU ↔ GPU

- Fonctions asynchrones préfixées par `Async` pour les copies mémoires :
  - Par ex. : `cudaMemcpyAsync()`
- Recouvrement des transferts CPU ↔ GPU
  - Par du calcul sur CPU : naturel !
  - Par du calcul sur GPU : à partir de la compute capability 1.1, exécution concurrente possible entre un kernel sur GPU et des transferts de données entre *page-locked memory* (sur CPU) et mémoire globale du GPU
    → besoin des *CUDA streams*
  - Par un transfert CPU ↔ GPU dans le sens inverse : à partir de la compute capability 2.x, possible d'effectuer concurremment un transfert « *page-locked host memory* → GPU » avec un transfert « GPU → *page-locked host memory* »

Exécution concurrente de kernels sur GPU

- Remarque : exécution concurrente CPU et GPU naturelle !
- A partir de la compute capability 2.x : possible d'exécuter concurremment jusqu'à 16 kernels sur GPU (32 à partir de 3.5)
  - ➤ Permet de mieux exploiter la puissance de calcul du GPU pour des kernels disposant d'un nombre de warps insuffisant
  - ➤ Besoin des *CUDA streams* :
    - *Stream* = séquence d'opérations qui s'exécutent dans l'ordre
    - Par défaut, toutes les opérations sont lancées (et exécutées dans l'ordre sur GPU) dans le stream 0
    - Besoin de créer des streams supplémentaires pour obtenir de la concurrence : type `cudaStream_t`, `cudaStreamCreate()`, `cudaStreamDestroy()`, `cudaStreamSynchronize()`...
  - ➤ Attention : nombreuses contraintes sur l'exécution concurrente des streams et synchronisations implicites (voir CUDA C Programming Guide et « CUDA C/C++ Streams and Concurrency » de S. Rennich)
    → gains modérés en pratique pour l'exécution concurrente de kernels

Exemple : calcul de la puissance K des éléments d'un tableau

```
#define CHUNK 131072
 float *vec, *res, *d_vec0, *d_vec1;
 int i, K=...;   /* elevation à la puissance K */
 int N=...;
 int taille_totale=N*sizeof(float);
 cudaStream_t stream0, stream1;
 cudaStreamCreate(&stream0);cudaStreamCreate(&stream1);

 cudaHostAlloc(&vec, taille_totale, cudaHostAllocDefault);
 cudaHostAlloc(&res, taille_totale, cudaHostAllocDefault);
 fillVectorFloat(vec, N);

 cudaMalloc((void **) &d_vec0, CHUNK*sizeof(float));
 cudaMalloc((void **) &d_vec1, CHUNK*sizeof(float));
```

Exemple : calcul de la puissance K des éléments d'un tableau (suite)

```
 dim3 threadsParBloc(512,1,1);
 dim3 tailleGrille((CHUNK/512)/16,1,1);

 for (int j=0; j<N; j+=CHUNK*2){
    cudaMemcpyAsync(d_vec0,vec+j,
         CHUNK*sizeof(float),cudaMemcpyHostToDevice, stream0);
    cudaMemcpyAsync(d_vec1,vec+(j+CHUNK),
         CHUNK*sizeof(float),cudaMemcpyHostToDevice, stream1);

    puissanceKernel<<<tailleGrille, threadsParBloc,
      0 /* mémoire partagée */, stream0>>>(d_vec0, K, CHUNK);
    puissanceKernel<<<tailleGrille, threadsParBloc,
      0 /* mémoire partagée */,stream1>>>(d_vec1, K, CHUNK);

    cudaMemcpyAsync(res+j,d_vec0,
         CHUNK*sizeof(float),cudaMemcpyDeviceToHost, stream0));
    cudaMemcpyAsync(res+(j+CHUNK),d_vec1,
         CHUNK*sizeof(float),cudaMemcpyDeviceToHost, stream1));
 } /* for j */
 cudaStreamSynchronize(stream0); cudaStreamSynchronize(stream1);

 cudaFree(d_vec0); cudaFree(d_vec1);
 cudaFreeHost(vec); cudaFreeHost(res);
}
```

Exemple : calcul de la puissance K des éléments d'un tableau (suite)

- Recouvrements :
  - Les 2 transferts CPU → GPU ne se recouvrent pas
  - Recouvrement du transfert CPU → GPU du stream1 avec le calcul sur GPU du stream0
  - Recouvrement des calculs sur GPU de stream0 et de stream1
  - Recouvrement du transfert GPU → CPU du stream0 uniquement avec les derniers blocs de threads du kernel du stream1 (voir synchronisations implicites dans documentation CUDA)
  - Recouvrement du transfert GPU → CPU du stream1 avec le transfert CPU → GPU du stream0 de l'itération suivante
- Performances :
  - Code sans streams (N = 268435456 K=4096) :

    temps d'exécution : 2.464448 → temps/élément : 9.180784e-09
  - Code avec streams (N = 268435456 K=4096) :

    temps d'exécution : 1.294222 → temps/éléments : 4.821352e-09
  - Soit un gain d'un facteur ~1,90

Bibliothèques et outils CUDA

- Bibliothèques :
  - CUBLAS : algèbre linéaire
    - Extension de l'interface BLAS pour spécifier indépendamment les transferts CPU ↔ GPU
    - Et aussi : MAGMA (LAPACK)
  - CUFFT : *Fast Fourier Transform*
  - CURAND : génération de nombres aléatoires
  - ...
- Outils CUDA : CUDA-GDB (*débuggeur*) , CUDA Profiler, CUDA-MEMCHECK …
- 2 niveaux d'API CUDA côté hôte :
  - *Runtime API* : programmation « haut niveau » (présentée dans ce cours)
  - *Driver API* : programmation « bas niveau »

Architecture des GPU
NVIDIA Kepler

- Exemple NVIDIA Kepler K20 (GK110) :
  - 13 Multiprocesseurs (MP) avec 32x6=192 *Scalar Processors* (SP) chacun
    → 2496 SP @ 0,706 GHz

    → warp toujours de 32 threads
  - Performance crête :
    - Simple précision : 3,52 Tflop/s
    - Double précision : 1,17 Tflop/s
  - Mémoire globale (*Device Mem*) : 5 Go
  - Bande passante mémoire : 208 Go/s
  - Bande passante CPU-GPU : 8 Go/s (PCI Express 2.0) ou 16 Go/s (PCI Express 3.0)
  - *Compute capability* CUDA : 3.5
  - CUDA 5.X

Spécificités de l'architecture Kepler et de CUDA 5.X

- *Hyper-Q* : exécution concurrente (kernels et transfers)
  - Jusqu'à 32 kernels concurrents sur K20
  - Permet d'éviter les synchronisations implicites entre streams sur Fermi
  - Facilite l'utilisation de CUDA + MPI (programmation et performances)
- *Dynamic parallelism :*
  - Lancement de kernels directement depuis le code d'un kernel CUDA sur GPU via <<< … >>>
  - Algorithmes récursifs
  - Attention au grain de calcul et au degré de parallélisme des kernels lancés depuis le GPU ...
- Voir : http://docs.nvidia.com/cuda/kepler-tuning-guide/

OpenCL

OpenGL (Open Graphics Library)→ OpenCL (Open Compute Library)

- Portable sur tous les GPU : NVIDIA, AMD, Intel
  - ➤ et aussi : CPU multicoeur, Cell, FPGA...
- Actuellement supporté par Apple, NVIDIA, AMD-ATI, Intel, IBM...
- Documentation : http://www.khronos.org/opencl/ et sites spécifiques à chaque implémentation
- Historique :
  - ➤ OpenCL 1.0 (2008, Mac OS X Snow Leopard)
  - ➤ OpenCL 1.1 (2010)
  - ➤ OpenCL 1.2 (2011) : *device partitioning*...
  - ➤ OpenCL 2.0 (2013) : *dynamic parallelism, C11 atomics*...

OpenCL (suite)

- Programmation OpenCL pour GPU :
  - ➤ Programmation « bas niveau » côté hôte (verbeux) : équivalent à CUDA Driver API
  - ➤ Programmation (et optimisation) des kernels : très proche de CUDA
    - Numérotation des work-items « locale » au work-group et « globale » (sur l'ensemble des work-items)
    - Instructions vectorielles explicites (GPU AMD, SSE, AVX...)
- CUDA conserve généralement certaines spécificités pour la programmation des GPU

CUDA	OpenCL
multiprocesseur	*compute unit*
thread	*work-item*
thread block	*work-group*
warp	*wave-front (AMD GPU)*
grid	*NDRange*
shared memory	*local memory*
global memory	*global memory*

GPU AMD

- Programmation des GPU AMD :
  - Historiquement : Brook+ (haut niveau, stream computing), CAL, CTM (*Close To The Metal*, bas niveau)
  - Désormais : OpenCL (et HSA...)
- GPU AMD : instructions vectorielles, mais architecture scalaire sur les derniers GPU AMD (par ex. HD 7970)
- Exemple de l'AMD Radeon HD 7970 « Tahiti » :
  - 32 processeurs @925 Mhz
  - 2048 « unités de calcul » au total
    → 3,79 Tflop/s (947 Gflop/s) en simple (double) précision (crête)
  - bande passante mémoire interne max de 264 Go/s

Programmation des GPU par directives de compilation

- HMPP (Hybrid Multicore Parallel Programming workbench)
  - CAPS Entreprise (www.caps-entreprise.com)
  - Directives « à la OpenMP » → préserve le code C ou Fortran original
  - Partie du code à accélérer sur HWA = *codelet*
    - Code de la *codelet* écrit à la main par l'utilisateur ou généré automatiquement par HMPP *codelet generator* (CUDA, OpenCL, SSE)
    - HMPP runtime : utiliser le HWA (GPU, FPGA) si présent et disponible, sinon exécution de la version native (C ou Fortran)
    - Optimisations : préchargement des données, communications asynchrones (→ recouvrement communications/calcul) …
- OpenACC : nouveau standard (2011) de programmation par directives de compilation (CAPS Enterprise, CRAY, PGI, NVIDIA : http://www.openacc-standard.org)
- OpenMP 4.0 (2013) : directives supplémentaires pour accélérateurs, code SIMD...

HMPP : exemple

```
#pragma hmpp matmul codelet, target=CUDA:BROOK:SSE, args[...]...=...
void matmul(args...){
   … code...;
}

#pragma hmpp matmul callsite [, options...]
matmul( ... );
```

(d'après « *HMPP™ datasheet* » et "*HMPP™: A Hybrid Multi-core Parallel Programming Environment*", Workshop on General Processing Using GPUs, Boston)

Quelques conclusions et remarques finales

- Puissance de calcul et bande passante mémoire GPU supérieure (~ un ordre de grandeur) à celles d'un CPU multicoeur
- GPU adapté à un parallélisme de données
  - ➤ Massif : besoin de pouvoir « extraire » plusieurs centaines/milliers de threads de l'application
  - ➤ Régulier : attention aux divergences (calcul et mémoire)
- Faible bande passante du bus PCI → goulet d'étranglement possible
- Programmation « relativement » aisée en CUDA (proche C/C++) mais
  - ➤ Besoin de réécrire tout ou partie de l'application
  - ➤ Changement algorithmiques nécessaires pour exploiter pleinement le GPU
  - ➤ Optimisation avancée du code non triviale (pas de contrôle sur le runtime CUDA → profiling)

Quelques conclusions et remarques finales (2)

- Exploitation directe des nouveaux GPU
  - ➤ Parallélisme de données → passage à l'échelle
  - ➤ Exécution SIMD masquée
  - ➤ Mais : auto-tuning nécessaire, et attention aux optimisations spécifiques à une compute capability
- Respect norme IEEE 754 en simple et double précisions (à quelques exceptions près) sur les GPU (compute capability > 1.3), mais attention : aux changements dans l'ordre des opérations, aux fonctions mathématiques...
- Pas de partage des ressources matérielles (mémoire partagée, registres...)