INF 560
Calcul Parallèle et Distribué
Cours 3

Eric Goubault et Sylvie Putot

Ecole Polytechnique

5 janvier 2015

Plan du cours

- Vision matériel: architecture cartes graphiques NVIDIA
- Vision logiciel: l'abstraction logique de l'architecture proposée par le langage CUDA (Compute Unified Device Architecture)
- Programmation CUDA
  - un peu de C ...
  - API CUDA
  - Un exemple: addition de matrices
- Revenons à l'architecture...pour optimiser...
  - obtenir un exécutable correct est raisonnablement facile
  - mais pour une bonne performance, il faut comprendre comment le code CUDA est exécuté sur les GPUS
  - threads: abstraction logique en grille et blocs
  - hiérarchie mémoire

Calcul sur GPU

Tirer parti pour des applications scientifiques de la puissance de calcul des cartes graphiques:

- Puissance de calcul importante (GFlops/s)
- Bande mémoire importante (GB/s)
- Disponibilité importante (des centaines de millions de cartes GPU compatibles NVIDIA dans le monde)

Principes de calcul sur GPU vs CPU

- CPU: minimiser la latence d'UN thread: grosses mémoires caches, unité de contrôle sophistiquée
- GPU: maximiser le débit de calcul
  - multithreading massif (création de threads très légère)
  - multithreading massif pour cacher la latence: pas de gros cache
  - nombre de threads sur une puce limité par les ressources (registres par threads, accès mémoire etc)
  - la logique de contrôle est partagée par des groupes de threads

Architecture: Fermi (2010-12), Kepler (12-?)

- Grand public:
  - Quadro K2000 (salles 31, 32, 33, 34): archi Kepler, 384 coeurs (2 SMX, multiprocesseurs de 192 coeurs)
  - GeForce GT 430 (salles 30, 35, 36): archi Fermi, 96 (2 SM, multiproc de 48 thread processors)
- Orientées HPC: Tesla K-20m (LIX): archi Kepler 2496 coeurs (13 SMX de 192 coeurs)

La brique de base est le "Streaming Multiprocessor (SMX)"

SMX: 192 single-precision CUDA cores, 64 double-precision units, 32 special function units (SFU), and 32 load/store units (LD/ST).

Architecture

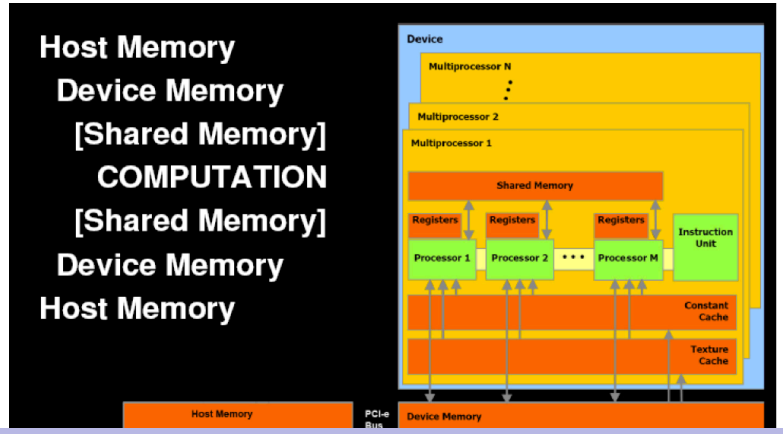
Streaming Multiprocessor (SMX)

- Ici (architecture Kepler): 192 coeurs par SMX
- Des groupes de 32 coeurs (appelés Warps) s'exécutent simultanément (mode SIMT) sur chaque SMX
- L'exécution alterne entre les warps actifs et inactifs
- Chaque thread a ses propres registres, ce qui limite le nombre de threads actifs en même temps
- Mémoire partagée rapide par SMX (ici 64KB): permet la communication entre threads
- Accès à la mémoire globale: latence de 400-600 cycles

Ordres de grandeur typiques pour de bonnes performances: des centaines de coeurs physiques, des dizaines de milliers de threads parallèles

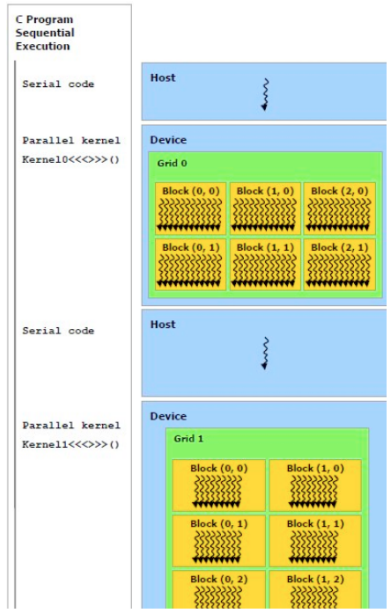
Architecture physique

- Organisés en multiprocesseurs (SM/SMX)
- Registres 32 bits par thread
- Mémoire partagée rapide uniquement par SM/SMX
- Mémoire ("constante") à lecture seule (ainsi qu'un cache de textures à lecture seule)



Une exécution typique

- Initialisation de la carte
- Allocation de la mémoire sur l'hôte (ou CPU) et sur la carte/device (ou GPU)
- Copie des données depuis l'hôte sur la carte
- Lancement de multiples instances parallèles ("kernels") sur la carte
- Utilisation éventuelle de mémoire partagée plus rapide
- Copie des résultats entre la carte et le CPU
- Désallocation de la mémoire et fin



Programmation CUDA

- la carte graphique="GPU" ou "device" est utilisé comme "co-processeur" de calcul pour l'hôte ou "CPU'
  - la mémoire du CPU est distincte de celle du GPU
  - mais on peut faire des transferts de l'un vers l'autre (couteux)
- le kernel crée sur le GPU un ensemble de threads, organisé de façon logique en une grille
- cette grille est une abstraction, qui est mappée physiquement sur la carte à l'exécution

Ce nest PAS un multiprocesseur plat comme PRAM

- la synchronisation globale est chère
- l'accès à la mémoire globale est cher

Modèle de parallélisme CUDA

Une hierarchie de threads concurrents

- les kernels sont composés de threads, qui exécutent chacun un clône (instance) du même programme séquentiel
- les threads sont groupés en blocs, les blocs en grille
- ils sont ordonnancés sur la carte sans préemption
- les threads d'un meme bloc peuvent coopérer
- les threads et blocs ont un unique identifiant: les threads connaissent ainsi leur position dans la grille, qui leur permet d'instancier la fonction sur leur données

Un modèle de programmation parallèle adaptable

- thread: processeur scalaire virtualisé (avec registres, etc)
- bloc: multiprocesseur virtualisé (avec threads, shared memory)
- la virtualisation permet l'adaptation lors d'un changement de carte: mapping d'une structure virtuelle (grille de blocs de threads) sur les multiprocesseurs physiques

Un peu de C: pointeurs/allocation C

- Notion explicite d'adresse mémoire

```
float *x,y; // x est un pointeur sur un float
x=&y; // x est l'adresse de y (allocation statique)
x=(float*) malloc(sizeof(float)); // alloc dynamique
```

- Tableaux et matrices

```
// Vecteur à N entrées (bloc de N mots de 32 bits)
float *x = (float *) malloc(N*sizeof(float));
// Matrice de dim N*M: y(i,j) est donné par y[i*M+j]
float *y = (float *) malloc(N*M*sizeof(float));
// Ou sinon: N pointeurs sur N vecteurs de taille M
float **z = (float **) malloc(N*sizeof(float *));
for (i=0; i<N; i++)
  z[i] = (float *) malloc(M*sizeof(float));
```

- Ou encore tableau C++:

```
float *x = new float[N];
```

Remarque: en C si `float *x;` alors `x[i]` est équivalent à `*(x+i)`

Un exemple: addition de matrices 2D

```
void add_matrix(float *a, float *b, float *c, int N) {
  for (int i=0; i<N; i++)
    for (int j=0; j<N; j++)
      c[i*N+j]=a[i*N+j]+b[i*N+j]; }

int main(int argc, char **argv) {
  float *x, *y, *z;
  int N=16, i=3, j=5;
  x=(float *) malloc(N*N*sizeof(float));
  y=(float *) malloc(N*N*sizeof(float));
  z=(float *) malloc(N*N*sizeof(float));
  (...)
  add_matrix(x, y, z, N);
  // affichage: %d pour les entiers, %f pour les float
  printf("z[%d,%d] vaut: %f\n",i,j,z[i*N+j]);
}
```

Composantes de la distribution CUDA (version 5.0 en salles machines)

- driver
  - contrôle bas niveau de la carte graphique
- toolkit
  - nvcc: le compilateur CUDA
  - des outils de profiling et debugging
  - des bibliothèques
- SDK (Software Developement Kit)
  - des exemples
  - des utilitaires (vérifications de messages d'erreur etc)
  - peu de documentation...
  - disponible en salles machines sous `/usr/local/cuda/samples`

API sur le GPU

- Qualificateurs de types de fonctions:

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

  - `__global__ ...`: les kernels, doivent retourner `void`
  - pas de récursion pour les fonctions exécutées sur le device
- Qualificateurs de types de variables:
  - `__device__ int x;`: `x` est un entier en mémoire globale
  - `__constant__ int x=5;`: `x` est un entier en mémoire constante
  - `__shared__ int x;`: `x` est un entier en mémoire partagée

API sur le GPU

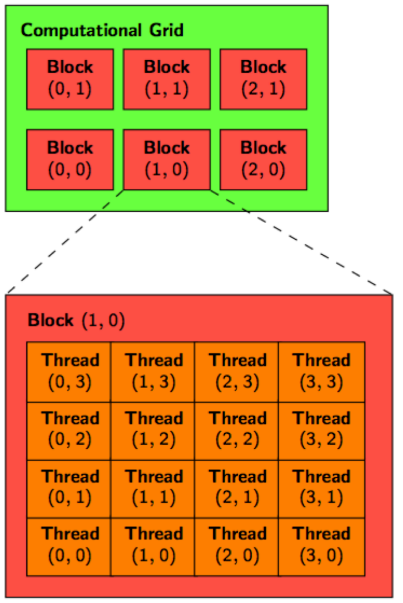
- Lancement depuis l'hôte du kernel `f` et configuration d'exécution (mapping logique sur la grille):

  `f<<< dimgrid, dimblock >>>(...)`

  où `dimgrid` et `dimblock` sont de type `dim3`
- Un appel d'un kernel est asynchrone (non bloquant)
- Chaque instance (thread) du kernel sait où il est exécuté par:
  - `blockIdx`: un `uint3` indiquant le bloc courant
  - `threadIdx`: un `uint3` indiquant le thread du bloc
  - `blockDim`: un `dim3` donnant les dimensions d'un bloc
  - `gridDim`: un `dim3` donnant les dimensions de la grille
- Si `A` est de type `dim3` ou `uint3`, `A.x`, `A.y` et `A.z` renvoient des `int` donnant les 3 coordonnées
  - La différence entre `dim3` et `uint3` est que les composantes non-initialisées d'un `dim3` valent 1

Abstraction logique - grid

- Une grille est un tableau 1D, 2D (ou 3D depuis Cuda 4.0) de blocs de threads
- Chaque bloc est un tableau 1D, 2D ou 3D de threads



Le Hello World de CUDA: addition de vecteurs (vecteur.cu)

```
// sur le GPU
// Calcul de C=A+B: chaque thread fait une addition
__global__ void vecAdd(float* a, float* b, float* c) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  c[i] = a[i] + b[i];
}

// sur le CPU
int main() {
  // initialisation mémoire, transfert des données sur la carte, e
  [...]
  // N/256 blocs de 256 threads
  vecAdd<<<N/256, 256>>>(d_a, d_b, d_c);
  [...]
}
```


Tableaux et gestion mémoire sur le GPU

Mémoire linéaire

- Allocation: cudaMalloc

```
int size = N*N*sizeof(float);
// tableau de N*N float en mémoire globale du GPU
float* d_A; cudaMalloc(&d_A, size);
float* h_A = (float*)malloc(size); // sur l'hôte
```

- Copie synchrone (bloquante) de size octets entre hôte et GPU

```
// copie de size octets de h_A vers d_A (hote vers device)
cudaMemcpy(d_A,h_A,size,cudaMemcpyHostToDevice);
cudaMemcpy(h_A,d_A,size,cudaMemcpyDeviceToHost);
// ou encore cudaMemcpyHostToHost, cudaMemcpyDeviceToDevice
```

- Libération:

```
cudaFree(d_A); free(h_A);
```

Tableaux 2D/3D: voir guides CUDA

- `cudaMallocPitch()`, `cudaMalloc3D()`
- A utiliser avec `cudaMemcpy2D()` et `cudaMemcpy3D()`

Compilation/exécution de cette version minimale

- Les kernels CUDA sont écrits dans des fichiers `.cu` (du C avec extensions Cuda, pas de récursion)
- Les fonctions et main sur le CPU, dans des fichiers `.c` ou dans le même `.cu` (du C ou C++)
- Compilateur NVIDIA `nvcc` compile les `.c` en utilisant le compilateur C sous-jacent (gcc etc.) et les `.cu`

Exemple (spécifique salle TD):

```
// variables d'environnement (ou dans .bashrc ou .bash_profile etc
[pontiac ~]$ export PATH=/usr/local/cuda-5.0/bin:${PATH}
[pontiac ~]$ export LD_LIBRARY_PATH=/usr/local/cuda-5.0/lib:$LD_LIE
// compilation, exécutable dans vecteur
[pontiac ~]$ nvcc -I/usr/local/cuda/include -L/usr/local/cuda/lib
vecteur.cu -o vecteur
// exécution
[pontiac ~]$ ./vecteur
```


Aide au débug

Utilisation de printf dans les kernels

- Possible pour les compute capability >= 2: (compiler avec option `-arch=sm_20` ou plus, cf doc). Par exemple

```
[orchis]$ /usr/local/cuda-5.0/bin/nvcc -arch=sm_30 -I/usr/local/cuda/include
-I/usr/local/cuda/samples/common/inc -L/usr/local/cuda/lib matrix2c.cu -o matrix2
```

- Les sorties vont dans un buffer (de taille limitée...), imprimé avec délai (possiblement bcp);
- `cudaDeviceSynchronize();` ou `cudaDeviceReset();` pour vider ce buffer à la fin

Vérification des aspects mémoire: cuda-memcheck

- compiler l'exécutable avec l'option `-g -G`
- puis lancer `cuda-memcheck mon_prgm_cuda`

Ou encore le debugger mais il faut 2 cartes ... cf TDs

Utilitaires de la SDK Cuda

```
#include<helper_cuda.h>  // récup messages d'erreur
#include<helper_timer.h> // timers

// checkCudaErrors récupère et affiche les erreurs
checkCudaErrors(cudaMalloc(&d_A, size));
checkCudaErrors(cudaMemcpy(d_A,h_A,size,cudaMemcpyHostToDevice));
checkCudaErrors(cudaFree(d_A));

// création/utilisation d'un timer
StopWatchInterface timer = 0; sdkCreateTimer(&timer );
sdkStartTimer(&timer );
// [...] code dont on veut mesurer le temps d'exécution
sdkStopTimer(&timer );
printf(Processing time: %f (ms)\n,sdkGetTimerValue(&timer));
sdkDeleteTimer(&timer );

// getLastCudaError teste si le kernel a produit une erreur
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);
getLastCudaError("Kernel execution failed");
```

Ajouter `-I/usr/local/cuda/samples/common/inc` dans la ligne de compilation:

```
nvcc -I/usr/local/cuda/include -L/usr/local/cuda/lib
-I/usr/local/cuda/samples/common/inc  vecteur.cu -o vecteur
```


Sinon pour compiler...

Utiliser le template CUDA et son makefile

Recopier `/users/profs/info/goubaul1/CUDA5.0` (la même chose que dans `/usr/local/cuda/samples` mais épuré de nombreux exemple pour pouvoir être recopié sur vos comptes):

```
[pontiac template]$ make
/usr/local/cuda-5.0/bin/nvcc -m32  -gencode arch=compute_10,code=sm_10 -gencode
arch=compute_20,code=sm_20 -gencode arch=compute_30,code=sm_30
-gencode arch=compute_35,code=sm_35 -I/usr/local/cuda-5.0/include -I. -I..
-I../../common/inc -o template.o -c template.cu
g++ -m32  -I/usr/local/cuda-5.0/include -I. -I.. -I../../common/inc -o
template_cpu.o -c template_cpu.cpp
g++ -m32 -o template template.o template_cpu.o -L/usr/local/cuda-5.0/lib -lcudart
mkdir -p ../../bin/linux/release
cp template ../../bin/linux/release
[pontiac template]$ ../../bin/linux/release/template
../../bin/linux/release/template Starting...
GPU Device 0: "GeForce GT 430" with compute capability 2.1
Processing time: 92.646004 (ms)
```

Utiliser nsight (Eclipse)

Reprendre exactement la manip sur la page TD 3!

Exemple modifié: somme de matrices

```
#include <helper_cuda.h>
#include <helper_functions.h>
#include <stdlib.h>
#include <stdio.h>

const int N = 1024;
const int blocksize = 16;
const int MAX = 100;

__host__ void add_matrix_cpu(float* a, float *b, float *c, int N)
  int i, j;
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      c[i*N+j]=a[i*N+j]+b[i*N+j];
}

__global__ void add_matrix(float* a, float *b, float *c, int N ) {
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int index = col + row*N;
  if ( col < N && row < N )
    c[index] = a[index] + b[index];
}
```


Somme de matrices (suite)

```
int main() {
  int k;
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];
  for ( int i = 0; i < N*N; ++i ) {
    a[i] = 1.0f; b[i] = 3.5f; }
  float *ad, *bd, *cd; const int size = N*N*sizeof(float);
  checkCudaErrors(cudaMalloc( (void**)&ad, size ));
  checkCudaErrors(cudaMalloc( (void**)&bd, size ));
  checkCudaErrors(cudaMalloc( (void**)&cd, size ));

  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );

  StopWatchInterface *timer = 0;  sdkCreateTimer(&timer);
  sdkStartTimer(&timer);
  for (k=1;k<=MAX;k++) {
    checkCudaErrors(cudaMemcpy( ad, a, size, cudaMemcpyHostToDevic
    checkCudaErrors(cudaMemcpy( bd, b, size, cudaMemcpyHostToDevic
    add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );
    getLastCudaError("Kernel execution failed");
    checkCudaErrors(cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHos
  }
```

Somme de matrices (fin)

```
sdkStopTimer(&timer);
printf("Processing time on GPU: %f (ms)\n",sdkGetTimerValue(&tim
sdkDeleteTimer(&timer);

checkCudaErrors(cudaFree( ad ));
checkCudaErrors(cudaFree( bd ));
checkCudaErrors(cudaFree( cd ));

StopWatchInterface *timer2 = 0;
sdkCreateTimer(&timer2);
sdkStartTimer(&timer2);
for (k=1;k<=MAX;k++)
  add_matrix_cpu(a,b,c, N);
sdkStopTimer(&timer2);
printf("Processing time on CPU: %f (ms)\n",sdkGetTimerValue(&tim
sdkDeleteTimer(&timer2);
delete[] a;
delete[] b;
delete[] c;
return EXIT_SUCCESS;
}
```


Exécution

```
[orchis ~]$ nvcc -I/usr/local/cuda/include -I/usr/local/cuda/sampl
[orchis ~]$ ./matrix2
Processing time on GPU: 2.688060 (ms)
Processing time on CPU: 3.229910 (ms)
```

- Codes sur GPU et CPU du même ordre de grandeur ...
- Pourquoi ? Beaucoup de transfert mémoire CPU - GPU comparé au calcul
- Par exemple si on accumule maintenant `ad := ad + bd` sans Memcpy intermédiaire: facteur 10

```
Processing time on GPU: 0.352680 (ms)
Processing time on CPU: 3.169960 (ms)
```

- Attention, si vous augmentez trop `N` ou `blocksize` vous aurez une erreur (dépassement du nombre de blocs ou du nombre de threads maxi par bloc). Les dimensions maxi (propres à la carte) sont données par deviceQuery, en salle machine:

`/users/profs/info/goubaul1/CUDA5.0/0_Simple/deviceQuery/deviceQuery`

Le résultat de deviceQuery en Salle 32 (Quadro K200)

```
Device 0: "Quadro K2000"
  CUDA Driver Version / Runtime Version          5.0 / 5.0
  CUDA Capability Major/Minor version number:    3.0
  Total amount of global memory:                 2047 MBytes (2146762752 bytes)
  ( 2) Multiprocessors x (192) CUDA Cores/MP:    384 CUDA Cores
  GPU Clock rate:                                954 MHz (0.95 GHz)
  Memory Clock rate:                             2000 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 262144 bytes
  Max Texture Dimension Size (x,y,z)             1D=(65536), 2D=(65536,65536), 3D=(4096,
  Max Layered Texture Size (dim) x layers        1D=(16384) x 2048, 2D=(16384,16384) x 2
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     2147483647 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      No
  Device PCI Bus ID / PCI location ID:           1 / 0
deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 5.0, CUDA Runtime Version = 5.0
```


Synchronisation – cf prochain cours

- Les threads à l'intérieur d'un bloc peuvent se synchroniser par barrière (point de rendez-vous de tous les threads):

```
[...]
__syncthreads();
[...]
```

- Les blocs peuvent coordonner par des opérations atomiques (par ex `atomicInc()`)
- Synchronisation par le CPU: barrière soit explicite (`cudaThreadSynchronize()`) soit implicite entre des kernels dépendants

Threads: abstraction logique vue de plus près

- Un bloc est exécuté par un seul multiprocesseur (SMX)
- Chaque bloc est exécuté par groupes de threads ("physiques") appelés "warps"
- Un warp (en général 32 threads) est exécuté physiquement en parallèle - parallélisme "SIMT" (SIMD, mais synchrone avec seulement une divergence possible de flot de contrôle)
- Un warp est constitué de threads de `threadIdx` consécutifs et croissants
- L'ordonnanceur de la carte alterne entre les warps

Abstraction logique vue de plus près

- Si le nombre de blocs excède le nombre de SMXs, alors plusieurs blocs (jusqu'à 16 pour Kepler en compute cap 3.0) seront exécutés (par warps, jusqu'à 64 par SMX) simultanément sur chaque SMX s'il y a suffisamment de registres et de mémoire partagée; les autres sont en attente et exécutés plus tard
- Tous les threads d'un blocs ont accès à la même mémoire partagée mais ne voient pas ce que font les autres blocs (même s'ils sont exécutés sur le même SMX)
- Les blocs d'un même kernel doivent être indépendants: il n'y a pas de garantie sur l'ordre dans lequel ils s'exécutent; le programme doit être valide pour n'importe quel entrelacement
- Des kernels indépendants peuvent s'exécuter sur différents flots de données

Spécifications techniques des cartes

Table 12. Technical Specifications per Compute Capability

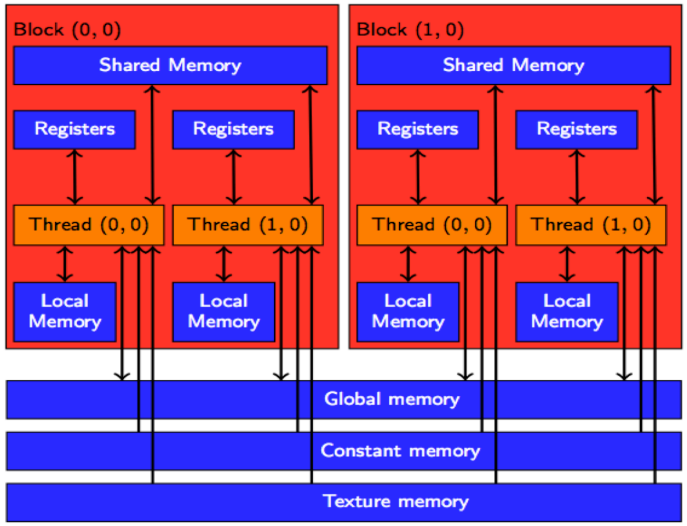
	Compute Capability							
Technical Specifications	**1.1**	**1.2**	**1.3**	**2.x**	**3.0**	**3.5**	**5.0**	**5.2**
Maximum dimensionality of grid of thread blocks	2			3				
Maximum x-dimension of a grid of thread blocks	65535			$2^{31}-1$				
Maximum y- or z-dimension of a grid of thread blocks	65535							
Maximum dimensionality of thread block	3							
Maximum x- or y-dimension of a block	512			1024				
Maximum z-dimension of a block	64							
Maximum number of threads per block	512			1024				
Warp size	32							
Maximum number of resident blocks per multiprocessor	8				16		32	
Maximum number of resident warps per multiprocessor	24	32		48	64			
Maximum number of resident threads per multiprocessor	768	1024		1536	2048			
Number of 32-bit registers per multiprocessor	8 K	16 K		32 K	64 K			
Maximum number of 32-bit registers per thread	128			63		255		
Maximum amount of shared memory per multiprocessor	16 KB			48 KB			64 KB	96 KB
Maximum amount of shared memory per thread block	16 KB			48 KB				
Number of shared memory banks	16			32				
Amount of local memory per thread	16 KB			512 KB				
Constant memory size	64 KB							
Cache working set per multiprocessor for constant memory	8 KB						10 KB	

Conséquences en terme d'efficacité/sécurité

- Pour être efficace (pour que l'ordonnanceur ait toujours quelque chose à ordonnancer), il faut essayer d'avoir suffisamment de blocs et de warps pour occuper tous les multiprocesseurs
  - minimum 16 blocs / 64 warps de 32 threads en même temps par multiprocesseur (cf spécification des cartes)
  - mieux d'en avoir davantage pour tirer parti du recouvrement potentiel calcul/accès mémoire
- Ne pas oublier `__syncthreads()` (blocs) et `cudaThreadSynchronize()` (synchro globale au niveau du CPU) pour assurer les fonctions de barrière de synchronisation (resp. attente qu'un kernel soit terminé)
  - le programme doit être valide pour n'importe quel entrelacement d'exécution des blocs

Modèle mémoire

Les accès mémoire: un point clé



Modèle mémoire

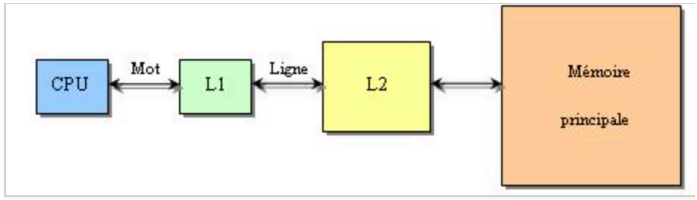
Suit la hiérarchie (logique) de la grid:

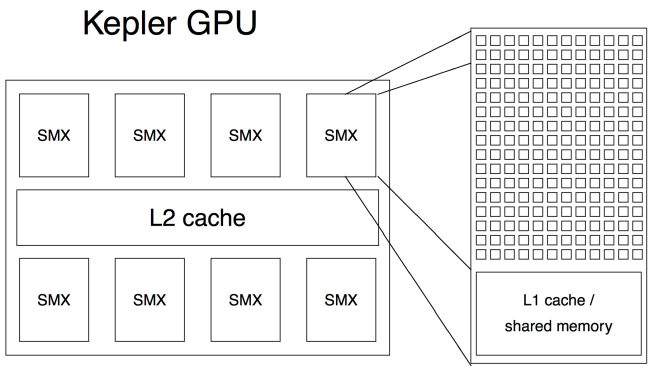
- Mémoire globale (du device): la plus lente (400 à 600 cycles de latence!), accessible (lecture/écriture) à toute la grille
- Mémoire partagée: rapide mais limitée (48KB par multiprocesseur), accessible (lecture/écriture) à tout un bloc - qualificatif `__shared__`
- Registres (64000 par multiprocesseur, 63 par thread): rapide mais très limitée, accessible (lecture/écriture) à un thread
- Mémoire locale (512KB): plus lente que les registres, accessible (lecture/écriture) - gérée automatiquement lors de la compilation (quand structures ou tableaux trop gros pour être en registre)
- Mémoire constante (64KB) et texture: rapide, accessible en lecture uniquement depuis le GPU, lecture/écriture depuis le CPU.

Caractéristiques mémoire

mémoire	accès	portée	durée de vie
registre	R/W	thread	thread
locale	R/W	thread	thread
partagée	R/W	bloc	bloc
globale	R/W	tous les threads + hôte	application
constante	R	tous les threads + hôte	application
texture	R	tous les threads + hôte	application

Modèle mémoire (partiel) CPU / Kepler





Mémoire cache

Mémoire cache: exploiter la localité des données

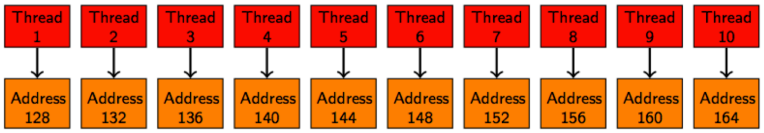
- localité temporelle: une donnée juste accédée risque d'être ré-utilisée prochainement, on la garde dans le cache
- localité spatiale: des données proches risquent d'être également utilisées prochainement, on les charge également dans le cache

L'unité de base de transfert de données est la ligne de cache. Typiquement 64 bytes (ou 16 registres 32 bits),

Kepler

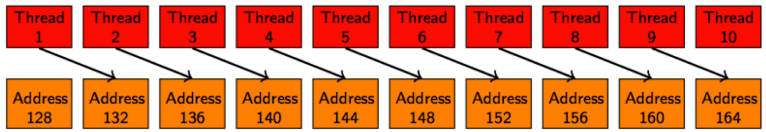
- ligne de cache 128 bytes
- 1.5MB de cache L2 depuis la mémoire globale
- 64kB de cache L1 par SMX depuis la mémoire partagée

Si possible: accès mémoire amalgamés

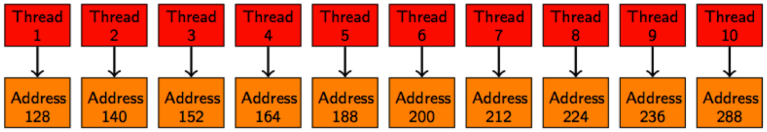


Les threads d'un warp accédent à la mémoire par des accès à 8, 16, 32/64, 128 bits consécutifs dans un bloc mémoire de 32, 64 ou 128 octets; adresse de départ alignée modulo 16

A éviter: accès mémoire non-amalgamés



Accès non aligné modulo 16



Adresses non "connexes" dans un bloc

Exemple d'accès amalgamé

```
__global__ void assign_matrix1(float* a, int N ) {
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int threadId = col + row*N;
  int index = col + row*N;
  a[index] = threadId;
}
```

- Les threads d'un warp accèdent des éléments consécutifs du tableau a
- Si les données sont bien alignées (a[0] est au début d'une ligne de cache), alors toutes les données pour un warp sont sur la même ligne de cache: transfert amalgamé

Exemple à éviter

```
__global__ void assign_matrix2(float* a, int N ) {
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int threadId = col + row*N;
  int index = row + col*N;
  a[index] = threadId;
}
```

- Les threads d'un warp accèdent des éléments espacés du tableau a
- Chaque accès nécessite le chargement d'une ligne de cache: mauvais, bande mémoire gâchée

Performances pour l'affectation (cf essai_amalgame.cu)

```
const int N = 1024;
const int blocksize = 16;
[...] assign_matrix1<<<dimGrid, dimBlock>>>( ad, N ); [...]
printf("Processing time (amalgamee) on GPU: %f (ms)\n",sdkGetTim
[...] assign_matrix2<<<dimGrid, dimBlock>>>( ad, N ); [...]
printf("Processing time (non amalgamee) on GPU: %f (ms)\n",sdkGe
```

```
[pontiac]$ ./essai_amalgame
Processing time (amalgamee) on GPU: 0.458600 (ms)
Processing time (non amalgamee) on GPU: 0.726260 (ms)
sylvie.putot@tesla:~$ ./essai_amalgame
Processing time (amalgamee) on GPU: 0.037260 (ms)
Processing time (non amalgamee) on GPU: 0.079340 (ms)
// En augmentant les dimensions: N=4096, blocksize = 32
[pontiac]$ ./essai_amalgame
Processing time (amalgamee) on GPU: 7.379540 (ms)
Processing time (non amalgamee) on GPU: 28.894851 (ms)
```


Mémoire globale

Tableaux en mémoire globale

- Ce qu'on a vu pour le moment
- Stockés sur la mémoire de la carte
- Alloués par l'hôte; pointeurs passés en paramètre aux kernels

Variables en mémoire globale

```
__device__ int ma_globvar;
__global__ void mon_kernel(...) { ... }
```

- définie dans le fichier du kernel; peut être lue/modifiée par les kernels
- peut être lue/modifiée par l'hote par les fonctions `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`
- durée de vie = l'application

Mémoire constante

Variables constantes ressemblent aux variables globales mais ne peuvent pas être modifiées par les kernels

```
__constant__ int ma_const;
```

- définie dans le fichier du kernel; peut être lue par les kernels, durée de vie = l'application
- initialisée par l'hote par les fonctions `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`
- 64KB de mémoire constante, plus chaque SMX à 8KB de cache
- Si tous les threads lisent la même constante, quasi aussi rapide que les registres

Registres et mémoire locale

Registres: rapides mais limités

- Les variables locales dans les kernels sont mises par défaut dans des registres
- 64K registres 32 bits par SMX, jusqu'à 63 registres par thread (255 pour K20)
- Si on veut le max de threads = 2048 threads par SMX (1024 par bloc): dans ce cas 32 registres maxi par thread
- Attention: si l'application a besoin de plus de registres, elle va utiliser la mémoire globale (ça coute!)

Mémoire "locale": tableaux et structures

- Petit tableau local: sera converti par le compilateur en registres scalaires
- Sinon mis dans le cache L1 ou la mémoire globale (mais 'considéré 'tableau local" car une copie par thread)

Mémoire partagée

- Dans un kernel, une variable (scalaire, ou tableau) déclarée avec le préfixe `__shared__`
- Partagée (lecture/écriture) par tous les threads d'un bloc: utile pour la communication
- Peut être utilisée en alternative aux registres ou tableaux locaux
- Attention à la synchronisation: barrières `__syncthreads();` par bloc (sinon SIMD seulement à l'intérieur d'un WARP)
- Mémoire partagée peut aussi être allouée dynamiquement; argument supplémentaire lors de l'appel d'un kernel
- 64KB pour archi Kepler; divisée 16/48 ou 32/32ou 48/16 enter cache L1 et mémoire partagée (par défaut 48KB, se règle par `cudaDeviceSetCacheConfig`)

En résumé

- La performance dépend naturellement du nombre de blocs de threads et de threads qui peuvent être exécutés sur un multiprocesseur
- Les ressources en mémoire coutent en performance
  - beaucoup de registres par thread => moins de blocs de threads
  - beaucoup de mémoire partagée => moins de blocs de threads
- Essayer de ne pas utiliser trop de registres (32 registres par thread max pour Kepler pour une occupation optimale)

Template /usr/local/cuda/samples/0_Simple/template/template.cu

```
// includes, system
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

// includes CUDA
...
// includes, project
...

////////////////////////////////////////////////////////////////////////////////
// declaration, forward
void runTest(int argc, char **argv);

extern "C"
void computeGold(float *reference, float *idata,
const unsigned int len);
```

```
//! Simple test kernel for device functionality
//! @param g_idata  input data in global memory
//! @param g_odata  output data in global memory
__global__ void testKernel(float *g_idata, float *g_odata)
{
    // shared memory
    // the size is determined by the host application
    extern  __shared__  float sdata[];

    // access thread id
    const unsigned int tid = threadIdx.x;
    // access number of threads in this block
    const unsigned int num_threads = blockDim.x;

    // read in input data from global memory
    sdata[tid] = g_idata[tid];
    __syncthreads();
    // perform some computations
    sdata[tid] = (float) num_threads * sdata[tid];
    __syncthreads();
    // write data to global memory
    g_odata[tid] = sdata[tid];
}
```



```
// Program main
int main(int argc, char **argv)
{
    runTest(argc, argv);
}

//! Run a simple test for CUDA
void runTest(int argc, char **argv)
{
    bool bTestResult = true;
    printf("%s Starting...\n\n", argv[0]);
    // use command-line specified CUDA device,
    // otherwise use device with highest Gflops/s
    int devID = findCudaDevice(argc, (const char **)argv);

    StopWatchInterface *timer = 0;
    sdkCreateTimer(&timer);
    sdkStartTimer(&timer);
    unsigned int num_threads = 32;
    unsigned int mem_size = sizeof(float) * num_threads;

    // allocate host memory
    float *h_idata = (float *) malloc(mem_size);
```



```
    // initialize the memory
    for (unsigned int i = 0; i < num_threads; ++i)
    {     h_idata[i] = (float) i;     }
    // allocate device memory
    float *d_idata;
    checkCudaErrors(cudaMalloc((void **) &d_idata, mem_size));
    // copy host memory to device
    checkCudaErrors(cudaMemcpy(d_idata, h_idata, mem_size,
                               cudaMemcpyHostToDevice));
    // allocate device memory for result
    float *d_odata;
    checkCudaErrors(cudaMalloc((void **) &d_odata, mem_size));
    // setup execution parameters
    dim3  grid(1, 1, 1);
    dim3  threads(num_threads, 1, 1);
    // execute the kernel
    testKernel<<< grid, threads, mem_size >>>(d_idata, d_odata);
    // check if kernel execution generated and error
    getLastCudaError("Kernel execution failed");
    // allocate mem for the result on host side
    float *h_odata = (float *) malloc(mem_size);
    // copy result from device to host
    checkCudaErrors(cudaMemcpy(h_odata, d_odata, sizeof(float) * n
                               cudaMemcpyDeviceToHost));
```



```
    sdkStopTimer(&timer);
    printf("Processing time: %f (ms)\n", sdkGetTimerValue(&timer))
    sdkDeleteTimer(&timer);
    // compute reference solution
    float *reference = (float *) malloc(mem_size);
    computeGold(reference, h_idata, num_threads);

    // check result and write file for regression test
    if (checkCmdLineFlag(argc, (const char **) argv, "regression")
        sdkWriteFile("./data/regression.dat", h_odata, num_threads
    else
    {
        // custom output handling when no regression test running
        // in this case check if the result is equivalent to the e
        bTestResult = compareData(reference, h_odata, num_threads,
    }

    // cleanup memory
    free(h_idata); free(h_odata); free(reference);
    checkCudaErrors(cudaFree(d_idata));
    checkCudaErrors(cudaFree(d_odata));
    cudaDeviceReset();
    exit(bTestResult ? EXIT_SUCCESS : EXIT_FAILURE);
}
```

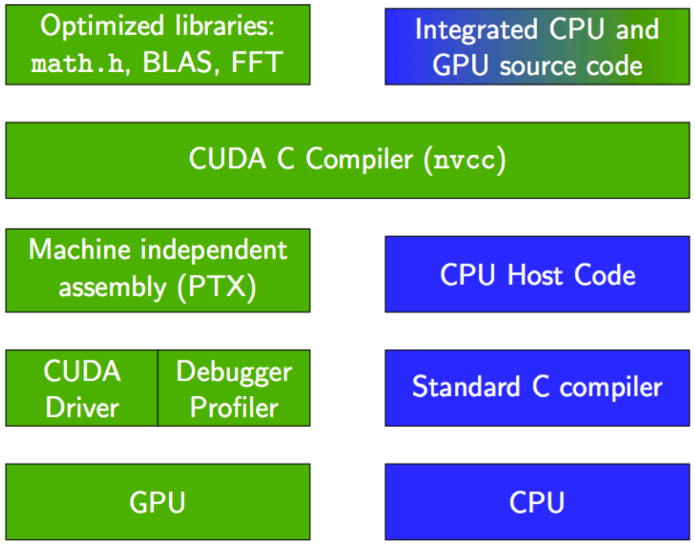
template_cpu.cpp

```
////////////////////////////////////////////////////////////////////////////////
// export C interface
extern "C"
void computeGold(float *reference, float *idata, const unsigned in

////////////////////////////////////////////////////////////////////////////////
//! Compute reference data set
//! Each element is multiplied with the number of threads / array
//! @param reference  reference data, computed but preallocated
//! @param idata      input data as provided to device
//! @param len        number of elements in reference / idata
////////////////////////////////////////////////////////////////////////////////
void
computeGold(float *reference, float *idata, const unsigned int len
{
    const float f_len = static_cast<float>(len);

    for (unsigned int i = 0; i < len; ++i)
    {
        reference[i] = idata[i] * f_len;
    }
}
```

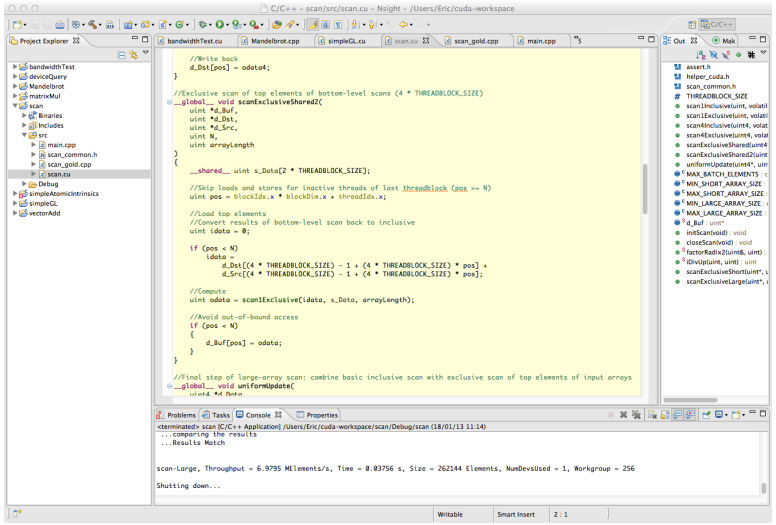

Remarques... sur les performances

- Ne vous laissez pas décourager par de piètres performances pour une première version de vos programmes
- Essayez de comprendre les raisons:
  - bank conflict (voir prochain cours)
  - transferts de données trop importants entre CPU et GPU pour un calcul trop court
  - trop de passage par la mémoire globale du GPU, et pas assez par la mémoire partagée au niveau des multi-processeurs
  - problèmes d'alignement des données
- Choisir suffisamment de blocs et de threads (multiple du nombre de threads par warp: 32...) pour cacher la latence d'accès à la mémoire
  - Typiquement 128 à 256 (min: 64, max: 512 en général)
  - Mais plus il y a des threads dans un block...plus cela peut être lent quand on fait `__syncthreads()`...

CUDA SDK



La version de la SDK 5.0



La suite

- TD3 Salle Info 32 à 10h15: première séance Cuda (TD sur 2 séances)
- Cours 4 en Amphi Lagarrigue lundi 12 janvier: programmation Cuda, synchronisation