

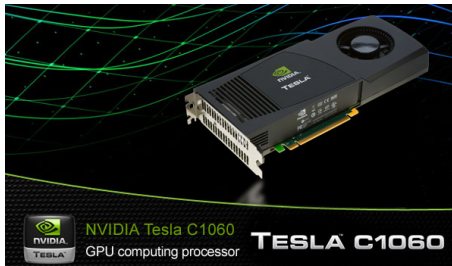
Initiation au calcul parallèle

programmation sur processeurs graphiques (GPU)

Nicolas GAC - MCF Université Paris Sud



Cours S4 TI - IUT de Cachan



- 1 Architecture des processeurs graphiques
 - GPU : Graphic Processing Unit
 - Avant CUDA ☹
 - Après CUDA ☺

- 2 Programmation sous CUDA
 - Principes généraux
 - Multiplication de matrices

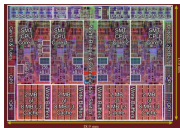
- 3 Exemples du SDK de CUDA

- 1 Architecture des processeurs graphiques
 - GPU : Graphic Processing Unit
 - Avant CUDA ☹
 - Après CUDA ☺
- 2 Programmation sous CUDA
- 3 Exemples du SDK de CUDA

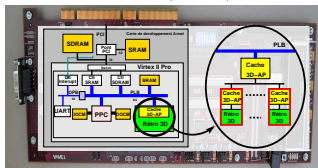
Calcul haute performance

High Performance Computing (HPC)

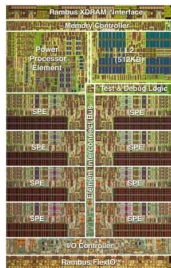
- Parallélisation sur machines multi-processeurs
 ➔ Efficace sur machine à mémoire distribuée
- Noeuds de calculs performants
 ➔ processeurs multi-core, many-core ou FPGA/ASIC



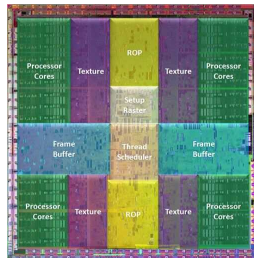
Intel Nehalem (4 coeurs)



SoPC (prototype)



IBM Cell (8+1 coeurs)

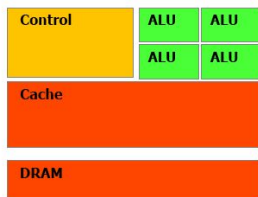


Nvidia GTX 200 (240 coeurs)

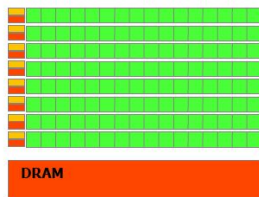
GPU : Graphic Processing Unit

Evolution vers une architecture *many core*

- A l'origine, architecture dédiée pour le rendu de volume
 ➤ Pipeline graphique (prog. en OpenGL/Cg)
- Depuis 2006, architecture adaptée à la parallélisation de divers calculs scientifiques
 ➤ CUDA : Common Unified Device Architecture (prog. en C)

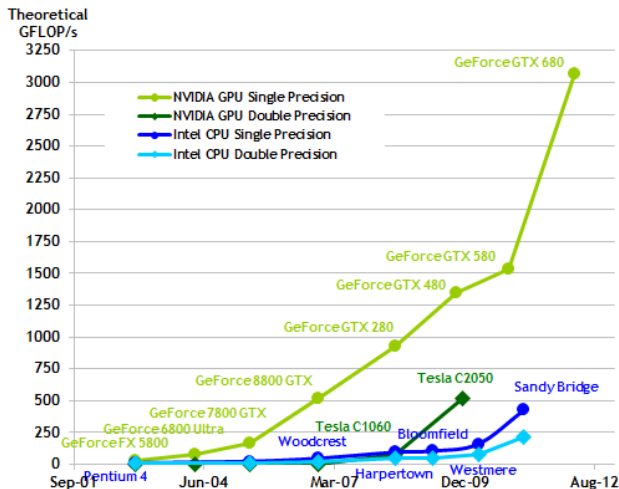


CPU



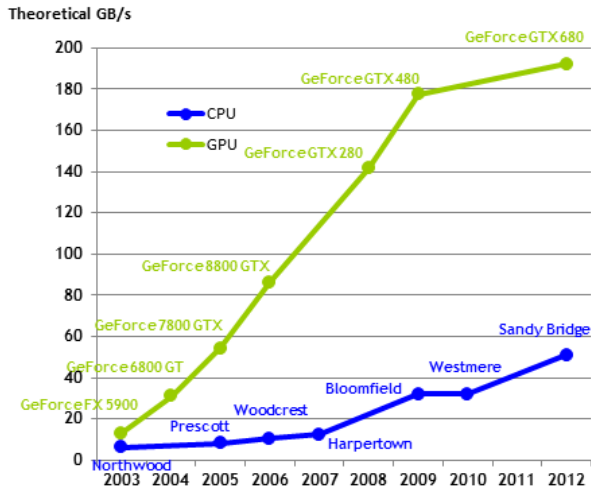
GPU

Puissance de calcul



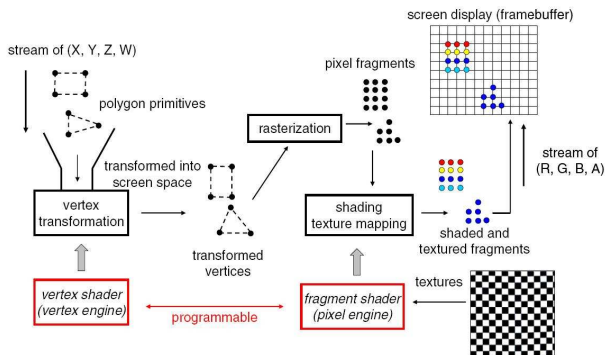
source : [programming guide 12]

Débit mémoire



source : [programming guide 12]

Avant CUDA : pipeline graphique



source : [Xu 07]

Vertex
Shader

Transformation
géométrique

Rasterization

Polygon \rightarrow
Fragments

Fragment
Shader

Calcul sur les
Pixels

Après CUDA : plein de threads !

CUDA : Common Unified Device Architecture

Vertex shader + Fragment shader = Streams Processors (SPs)

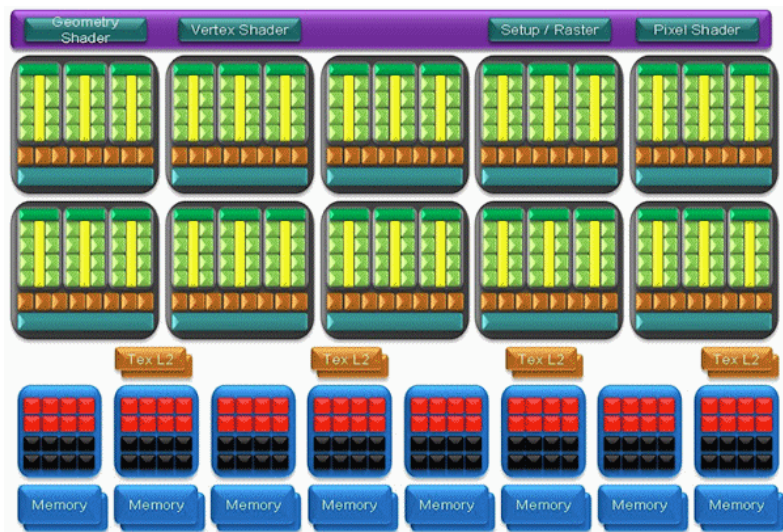
⇒ Le GPU devient un processeur “many cores”

Jusqu'à 240 Stream Processors (Chip GT200 - Juin 2008)

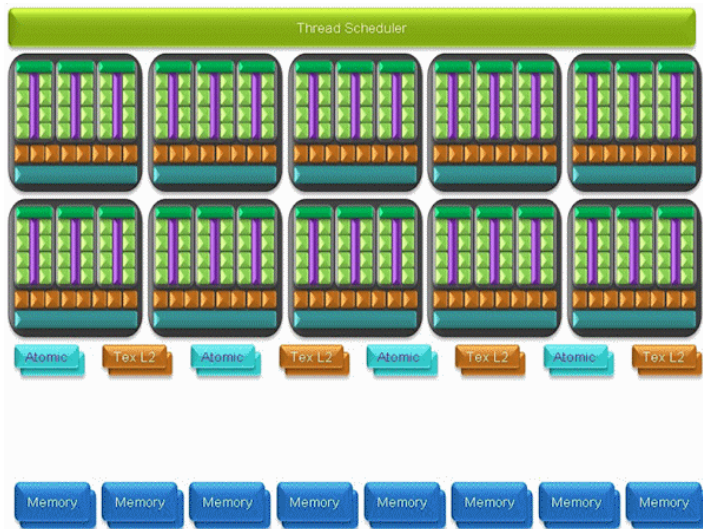
- 30 coeurs SIMT (Single instructions Multiple Threads)
- 8 SPs (Stream Processors) par coeur SIMT
- 3 flop (MADD + MUL) par SP

⇒ soit ~ **1 Teraflops** pour la carte Geforce GTX 285 (SP @1,5 GhZ)

GT200 utilisé en mode graphique



GT200 utilisé en mode CUDA



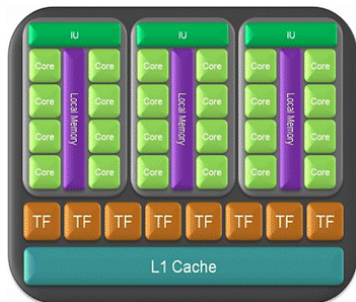
Coeur SIMT : Single Instruction Multiple Instruction

Hardware

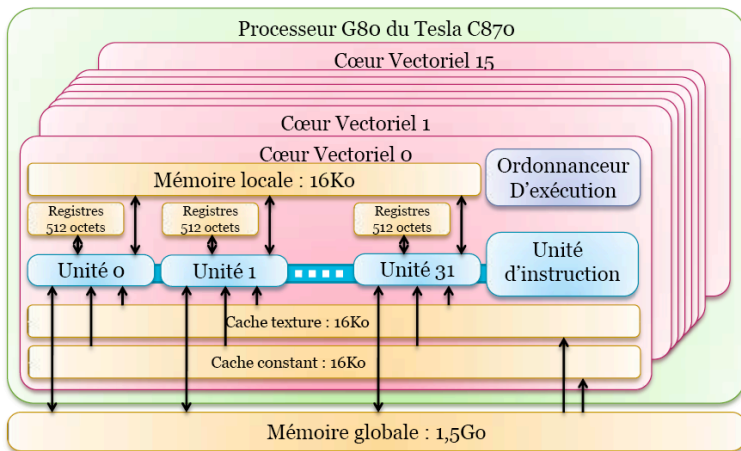
- 8 unités scalaires (Stream Processors)
- 2 Specific Processor Units (SFUs) :
cos(), sin(), exp()...
- 16 Ko de registres (2 Ko par SP)
- 16 ko de mémoire locale (shared memory)
- Accès rapide aux textures : cache 2D
+ Texture Filtering (TF)

Ordonnancement des threads

Execution par groupes de 32 threads
appelés **warps**


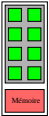
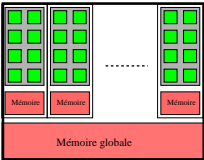


Accès aux mémoire(s)

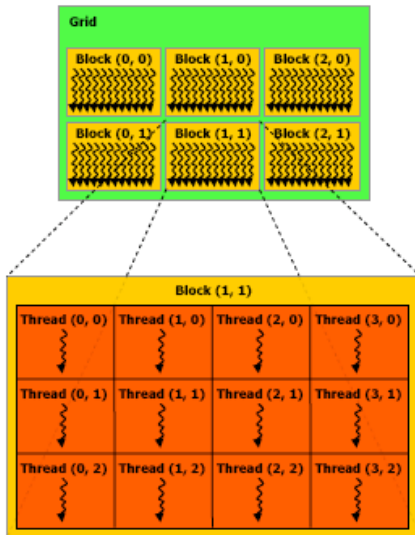


source : [Kirschenmann 08]

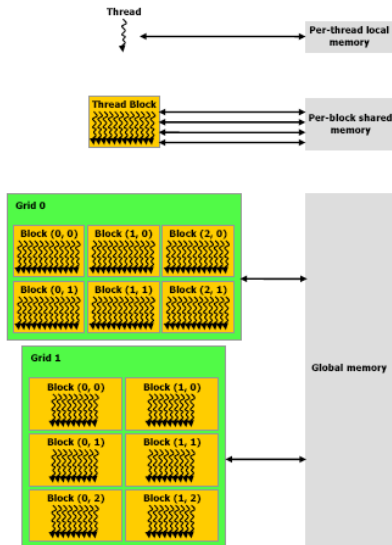
Découpage en threads

Matériel	Logiciel	Exécution	
 un Stream Processor (SP)	un thread	séquentielle	(a)
 un coeur SIMT	un bloc de threads (plusieurs warps)	parallèle (SIMT)	(b)
 une carte GPU (device)	une grille de threads (kernel)	parallèle (MIMD) mémoire centralisée	(c)

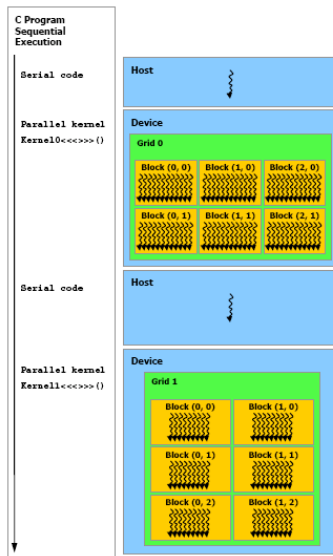
Un **id** par thread et un **id** par bloc de threads



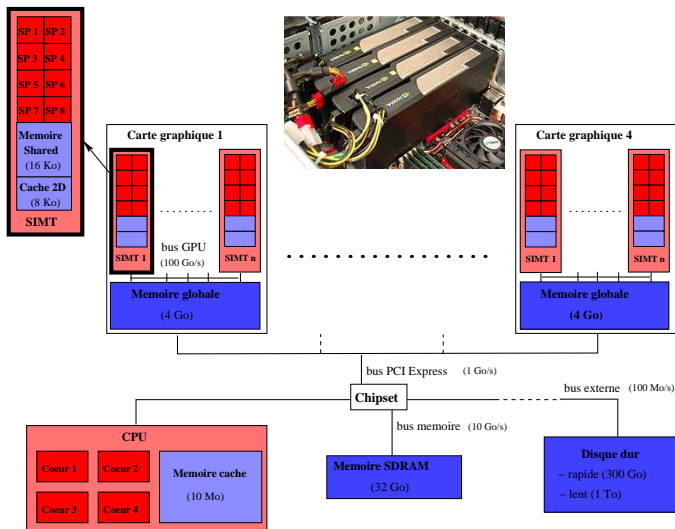
Hiérarchie mémoire





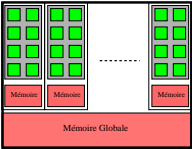
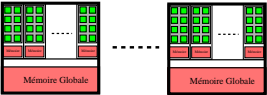
PC hôte et carte graphique



Supercalculateur personnel

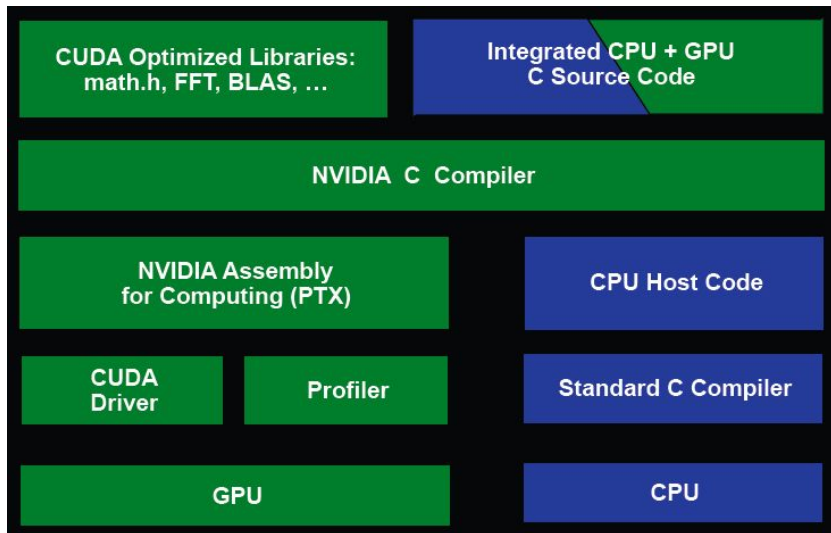


Découpage en threads et en grilles (kernels)

Matériel	Logiciel	Exécution
 un Stream Processor (SP)	un thread	séquentielle (a)
 un processeur SIMT	un bloc de threads	parallèle (SIMT) (b)
 une carte GPU (device)	une grille de threads (kernel)	parallèle (MIMD) mémoire centralisée (c)
 PC multi-carte	threads du PC hôte via librairie pthread (un thread CPU = un kernel GPU)	parallèle (MIMD) mémoire distribuée (d)

- 1 Architecture des processeurs graphiques
- 2 **Programmation sous CUDA**
 - Principes généraux
 - Multiplication de matrices
- 3 Exemples du SDK de CUDA

Flot de développement logiciel



Programmation GPU

❶ Parallélisation de l'algorithme

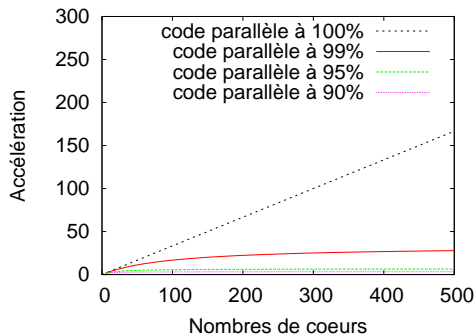
➤ nourrir en threads (plus ou moins indépendants) le GPU

n coeurs (1 Ghz)

vs

1 coeur (3 Ghz)

taux de parallélisation	accélération GTX 200 (240 coeurs)
100 %	80
99 %	24
95 %	6
90 %	3



Programmation GPU

① Parallélisation de l'algorithme

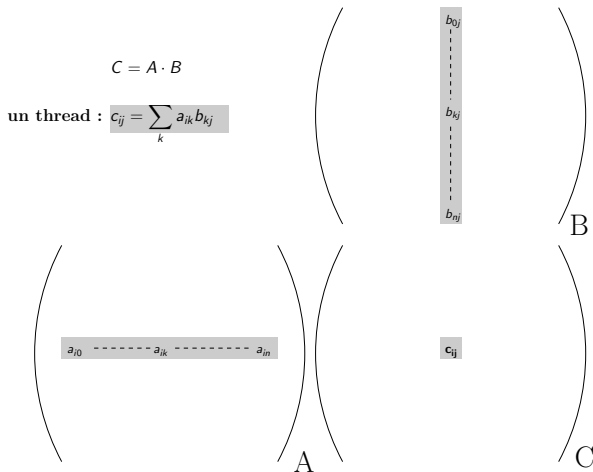
➤ nourrir en threads (plus ou moins indépendants) le GPU

② Implémentation GPU

Selon l'intensité arithmétique du code (puissance de calcul exploitée / débit des données), l'exécution sera soit *memory bound* soit *computation bound* (ex : calcul X^k [Kirschenmann 08])

➤ optimisation du code portera alors soit sur les **accès mémoire** ou soit sur la **complexité arithmétique**

Parallélisation du calcul matriciel

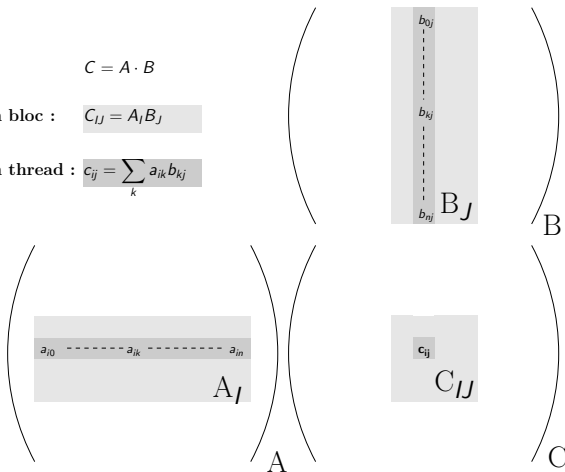


Découpage en blocs de threads

$$C = A \cdot B$$

un bloc : $C_{IJ} = A_I B_J$

un thread : $c_{ij} = \sum_k a_{ik} b_{kj}$



kernel = code des threads exécutés sur le GPU

```
__global__ void matrixMul_kernel( float* C, float* A, float* B,int matrix_size) {  
    float Csum;  
    int ifirst,jfirst;  
    int i,j;  
  
    ifirst=blockIdx.x*BLOCK_SIZE;  
    jfirst=blockIdx.y*BLOCK_SIZE;  
  
    i=ifirst+threadIdx.x;  
    j=jfirst+threadIdx.y;  
  
    for (k = 0; k < matrix_size; k++)  
        Csum += A[i][k] * B[k][j];  
  
    C[i][j] = Csum;  
}
```

Lancement du kernel depuis le PC hôte

```
#define BLOCK_SIZE 16

void matrixMul_host(int N) {
...
//setup execution parameters
dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid(N /BLOCK_SIZE , N /BLOCK_SIZE );
//execute the kernel
matrixMul_kernel<<< grid, threads >>>(C_device, A_device, B_device, N);
...
}
```

Gestion de la mémoire GPU via le PC hôte

```
#define BLOCK_SIZE 16

void matrixMul_host(int N) {

    // allocate host memory int mem_size=N2*sizeof(float);
    float* A_host = (float*) malloc(mem_size);
    float* B_host = (float*) malloc(mem_size);
    float* C_host = (float*) malloc(mem_size);

    // allocate device memory
    float* A_device,B_device,C_device;
    cudaMalloc((void**) &A_device, mem_size);
    cudaMalloc((void**) &B_device, mem_size);
    cudaMalloc((void**) &C_device, mem_size);

    // copy host memory to device cudaMemcpy(A_device, A_host, mem_size,cudaMemcpyHostToDevice);
    cudaMemcpy(B_device, B_host, mem_size,cudaMemcpyHostToDevice);

    //setup execution parameters
    dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
    dim3 grid(N /BLOCK_SIZE , N /BLOCK_SIZE );

    // execute the kernel
    matrixMul_kernel<<< grid, threads >>>(C_device, A_device, B_device, N);

    // copy result from device to host
    cudaMemcpy(C_host, C_device, mem_size,cudaMemcpyDeviceToHost) ;

}
```

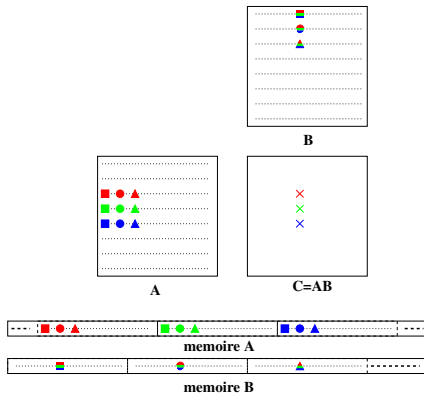
Temps GPU

Matrices de taille 1024·1024

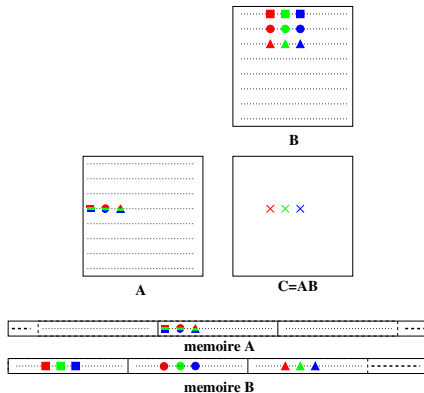
	Processeur	Temps d'exécution	Transfert mémoire
C non optimisé	Xeon Quad core 2.7 Ghz	9.35 s	
Cuda	Testla C1060 240 PE @1,3 Ghz	1.35 s (*6,9)	< 1%

Accès séquentiels à la mémoire globale

Acces non sequentiels en memoire globale



Acces sequentiels en memoire globale



Acces par le thread 1 : ■ ● ▲
 Acces par le thread 2 : ■ ● ▲
 Acces par le thread 3 : ■ ● ▲

→ temps

Temps GPU avec accès mémoire séquentiels

Matrices de taille 1024·1024

	Processeur	Temps d'exécution	Transfert mémoire
C non optimisé	Xeon Quad core 2.7 Ghz	9.35 s	
Cuda	Testla C1060 240 PE @1,3 Ghz	1.35 s (*6,9)	< 1%
Cuda acces seq.	Testla C1060 240 PE @1,3 Ghz	124 m s (*10,9)	5%

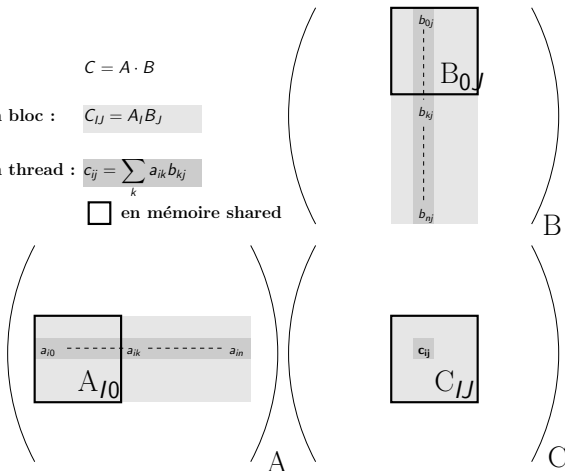
Optimisation des accès mémoire

$$C = A \cdot B$$

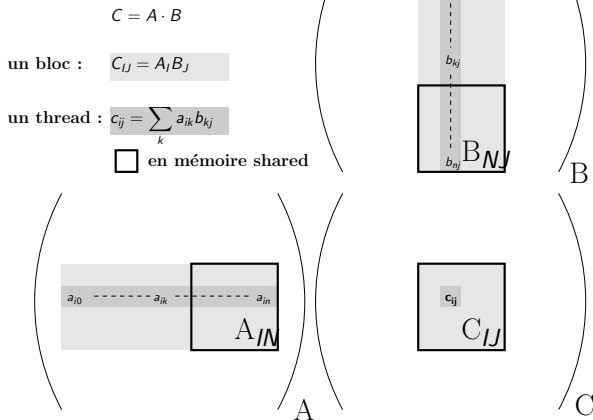
un bloc : $C_{IJ} = A_I B_J$

un thread : $c_{ij} = \sum_k a_{ik} b_{kj}$

☐ en mémoire shared



Optimisation des accès mémoire



Variable type qualifiers

`__device__`

- en mémoire globale
- durée de vie de l'application
- accessible par tous les threads de la grille et par le hôte via la librairie runtime

`__constant__`

- en mémoire globale (accès via cache constante)
- durée de vie de l'application
- accessible par tous les threads de la grille et par le hôte via la librairie runtime

`__shared__`

- en mémoire shared (locale à un coeur SIMT)
- durée de vie du bloc de threads
- **seulement accessible par les threads d'un même bloc**

Temps GPU optimisé

Matrices de taille 1024·1024

	Processeur	Temps d'exécution	Transfert mémoire
C non optimisé	Xeon Quad core 2.7 Ghz	9.35 s	
Cuda	Testla C1060 240 PE @1,3 Ghz	1.35 s (*6,9)	< 1%
Cuda acces seq.	Testla C1060 240 PE @1,3 Ghz	124 m s (*10,9)	5%
Cuda shared mem.	Testla C1060 240 PE @1,3 Ghz	17,5 m s (*7,1)	34%

Librairie CUBLAS : CUda Basic Linear Algebra Subprograms

```
#include <cublas.h>
#include <cutil.h>

int main(void) {
    float alpha = 1.0f, beta = 0.0f;
    int N = 1024;
    int mem_size = 1024*1024*sizeof(float);

    // Allocate host memory
    float* A_host = (float*) malloc(mem_size);
    float* B_host = (float*) malloc(mem_size);
    float* C_host = (float*) malloc(mem_size);

    cublasInit();

    //Allocate device memory
    float* A_device,B_device,C_device;
    cublasAlloc(N*N, sizeof(float), (void **)&A_device);
    cublasAlloc(N*N, sizeof(float), (void **)&B_device);
    cublasAlloc(N*N, sizeof(float), (void **)&C_device);

    // copy host memory to device
    cublasSetMatrix(N,N, sizeof(float), A_host, N, A_device, N);
    cublasSetMatrix(N,N, sizeof(float), B_host, N, B_device, N);

    //Calcul matriciel sur le GPU
    cublasSgemv('n', 'n', N, N, N, alpha, A_device, N,B_device, N, beta, C_device, N);

    //Récupération du résultat sur le PC hôte
    cublasGetMatrix(N,N, sizeof(float), C_device,N, C_host, N);
}
```

Temps CUBLAS

Matrices de taille 1024·1024

	Processeur	Temps d'exécution	Transfert mémoire
C non optimisé	Xeon Quad core 2.7 Ghz	9.35 s	
Cuda	Testla C1060 240 PE @1,3 Ghz	1.35 s (*6,9)	< 1%
Cuda acces seq.	Testla C1060 240 PE @1,3 Ghz	124 m s (*10,9)	5%
Cuda shared mem.	Testla C1060 240 PE @1,3 Ghz	17,5 m s (*7,1)	34%
CUBLAS	Testla C1060 240 PE @1,3 Ghz	12,8 m s (*1,4)	43%

Interface Matlab (1/2)

```
void mexFunction( int nlhs, mxArray *plhs[],int nrhs, const mxArray *prhs[] ){
    unsigned int N = mxGetM(prhs[0]);
    plhs[0] = mxCreateDoubleMatrix(N,N, mxREAL);

    // Assign pointers to each input and output.
    double *A_double_h,*B_double_h,*C_double_h;
    A_double_h = mxGetPr(prhs[0]);
    B_double_h = mxGetPr(prhs[1]);
    C_double_h = mxGetPr(plhs[0]);

    //Conversion de A et B en float
    float *A_float_h,*B_float_h,*C_float_h;
    A_float_h=(float *)mxMalloc(sizeof(float)*N*N);
    B_float_h=(float *)mxMalloc(sizeof(float)*N*N);
    C_float_h=(float *)mxMalloc(sizeof(float)*N*N);
    for (int i=0;i<N*N;i++){
        A_float_h[i]=(float) A_double_h[i];
        B_float_h[i]=(float) B_double_h[i];}

    // allocate device memory
    float* A_d,B_d,C_d;
    cudaMalloc((void**) &A_d, N*N*sizeof(float));
    cudaMalloc((void**) &B_d, N*N*sizeof(float));
    cudaMalloc((void**) &C_d, N*N*sizeof(float));

    // copy host memory to device
    cudaMemcpy(A_d, A_float_h,N*N*sizeof(float) ,cudaMemcpyHostToDevice) ;

    cudaMemcpy(B_d, B_float_h,N*N*sizeof(float) ,cudaMemcpyHostToDevice) ;
```

Interface Matlab (2/2)

```
// setup execution parameters
dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid(N /BLOCK_SIZE , N /BLOCK_SIZE );

// execute the kernel
matrixMul<<< grid, threads >>>(C_d, A_d, B_d, N, N);

// copy result from device to host
cudaMemcpy(C_float_h, C_d,N*N*sizeof(float) ,cudaMemcpyDeviceToHost );

//Conversion de C en double
for (i=0;i<N*N;i++){
    C_double_h[i]=(double) C_float_h[i];
}
```

- 1 Architecture des processeurs graphiques
- 2 Programmation sous CUDA
- 3 Exemples du SDK de CUDA**

Exemples du SDK de CUDA [Zone 12] (1/2)

Algèbre linéaire

- Multiplication matrice
- Calcul des valeurs propres

Traitement d'image

- Convolution d'image (noyau separable)
- Convolution d'image par FFT
- Filtre de Sobel
- DCT
- Débruitage d'image
- Histogram

Exemples du SDK de CUDA [Zone 12] (2/2)

Monte carlo

- Quasi random generator
- Simulation Montecarlo pour la finance
- Mersenne Twister (générateur de nombre aléatoire)

Manipulation de données

- Réduction parallèle
- Scan (utilisé pour les algorithmes de tri)

Références



Wilfried Kirschenmann.

Parallélisation d'un solveur de neutronique sur GPU.

In 2ième Journée Thème Émergeant GPGPU (GDR ASR), December 2008.



NVIDIA Cuda programming guide.

<http://docs.nvidia.com/cuda/index.html>, 5.0 edition, 2012.



Fang Xu & Klaus Mueller.

Real-time 3D computed tomographic reconstruction using commodity graphics hardware.

Physics in Medicine and Biology, vol. 52, no. 12, pages 3405–3419, 2007.



CUDA Zone.

<https://developer.nvidia.com/category/zone/cuda-zone>, 2012.