Parallel computing on GPU
Part 1 : CUDA programming

Nicolas GAC

Maître de conférences à l'université Paris Saclay
Laboratoire des Signaux et Systèmes (L2S) - Groupe Problèmes Inverses (GPI)

M2 SETI - C4 GPU/FPGA et A4 GPU

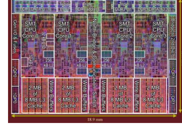
1. GPU (re)designed as a many core architecure
   - GPU computing
   - Multithreading (SIMT)
   - Co-processing
2. Programming in CUDA
   - Compilation
   - Parallelisation
3. A toy example : acceleration of matrix multiplication
   - GPU code
   - CPU code
   - Performance

1. GPU (re)designed as a many core architecure
   - GPU computing
   - Multithreading (SIMT)
   - Co-processing
2. Programming in CUDA
   - Compilation
   - Parallelisation
3. A toy example : acceleration of matrix multiplication
   - GPU code
   - CPU code
   - Performance

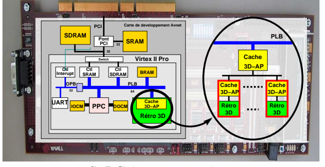
Calcul haute performance

High Performance Computing (HPC)

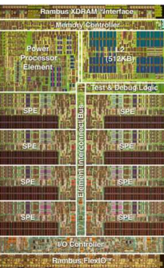
- Parallélisation sur machines multi-processeurs
  ➲ Efficace sur machine à mémoire distribuée
- Noeuds de calculs performants
  ➲ processeurs multi-core, many-core ou FPGA/ASIC



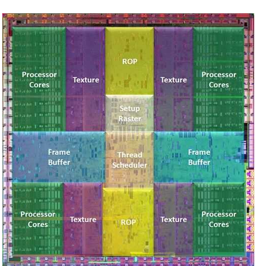
Intel Nehalem (4 coeurs)



SoPC (prototypage)



IBM Cell (8+1 coeurs)

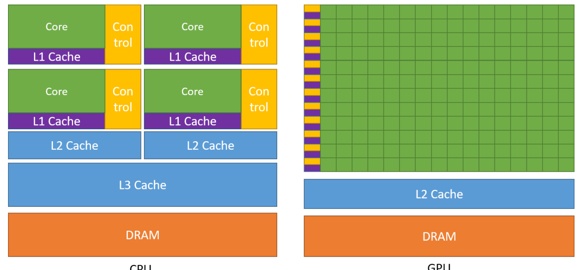


Nvidia GTX 200 (240 coeurs)

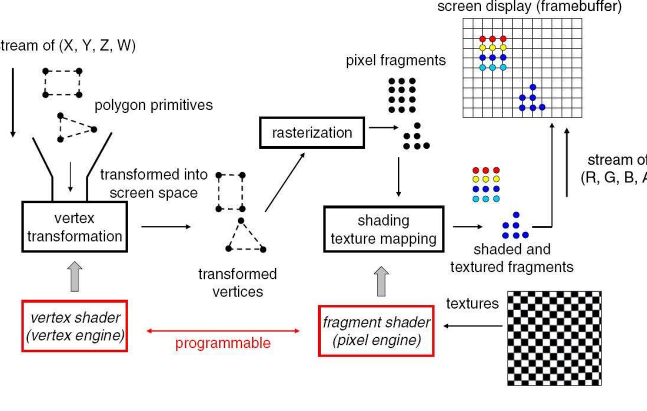
GPU : Graphic Processing Unit

Evolution vers une architecture *many core*

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



Avant CUDA : pipeline graphique



Vertex Shader

Transformation géométrique

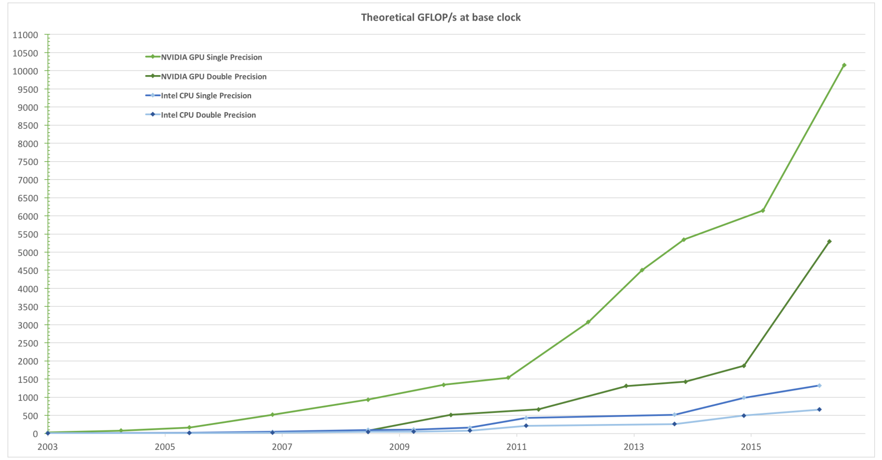
Rasterization

Polygon − > Fragments

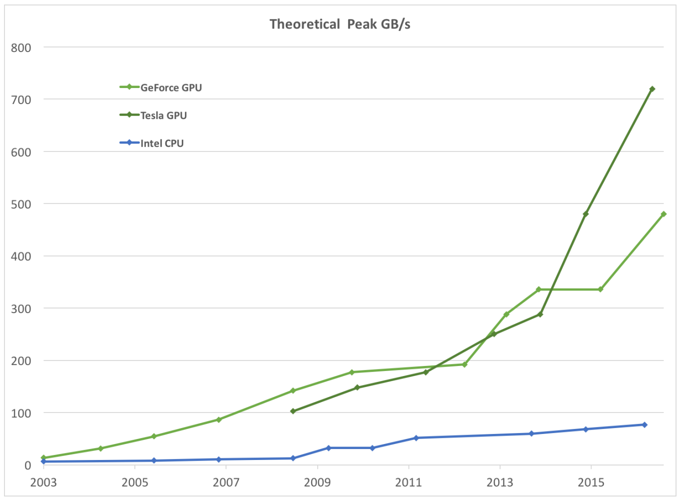
Fragment Shader

Calcul sur les Pixels

Puissance de calcul



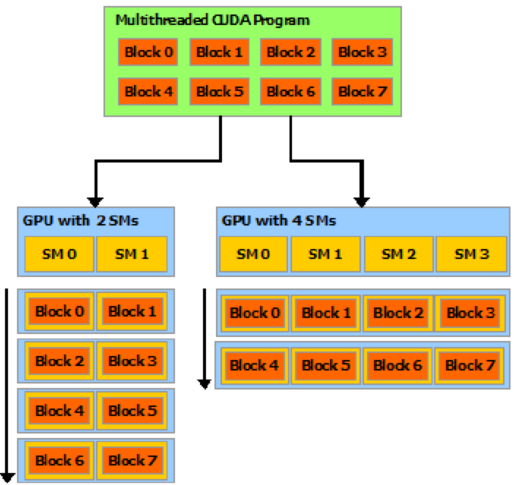
Débit mémoire



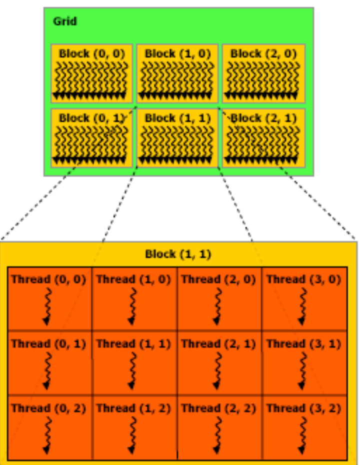
Découpage en threads

	Matériel	Logiciel	Exécution	
	un Streaming Processor (SP)	**un thread**	séquentielle	**(a)**
Mémoire	un Streaming MultiProcessor (SM)	**un bloc de threads** (plusieurs warps)	parallèle (SIMT)	**(b)**
Mémoire Mémoire Mémoire Mémoire globale	une carte GPU (device)	**une grille de threads** (kernel)	parallèle (MIMT)	**(c)**

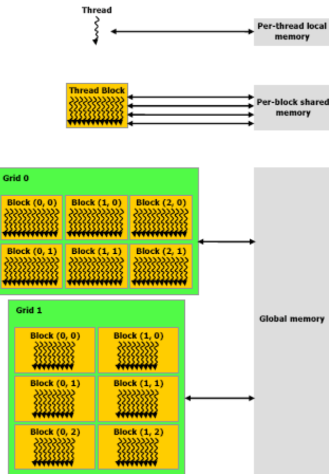
Ordonnancement des blocs de threads



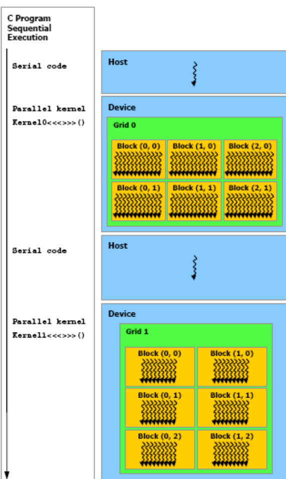
Un **id** par thread et un **id** par bloc de threads



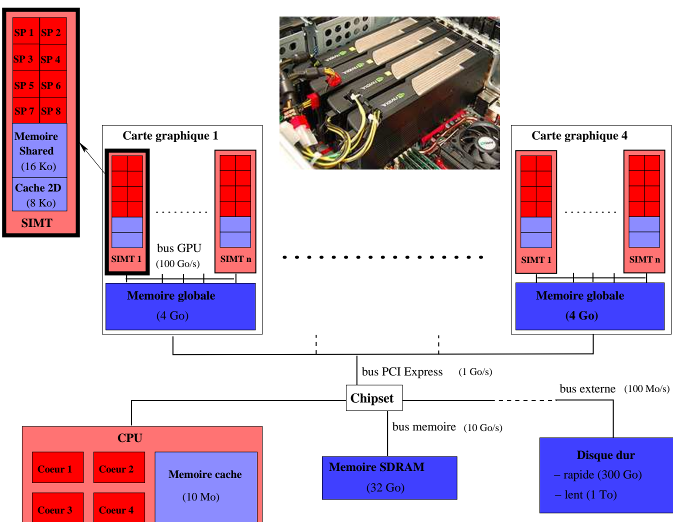
Hiérarchie mémoire



PC hote et carte graphique



Supercalculateur personnel

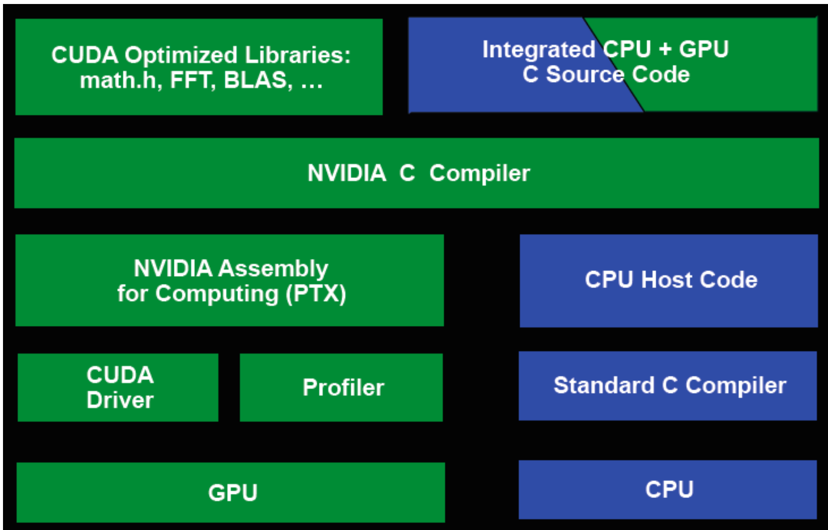


Découpage en threads et en **grilles (kernels)**

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

1. GPU (re)designed as a many core architecure
   - GPU computing
   - Multithreading (SIMT)
   - Co-processing
2. Programming in CUDA
   - Compilation
   - Parallelisation
3. A toy example : acceleration of matrix multiplication
   - GPU code
   - CPU code
   - Performance

Flot de développement logiciel



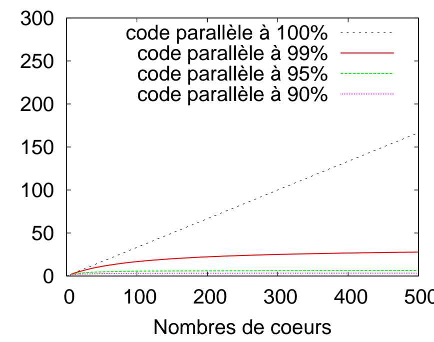
Programmation GPU

1) 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'execution sera soit *memory bound* soit *computation bound* (ex : calcul X^k [?])

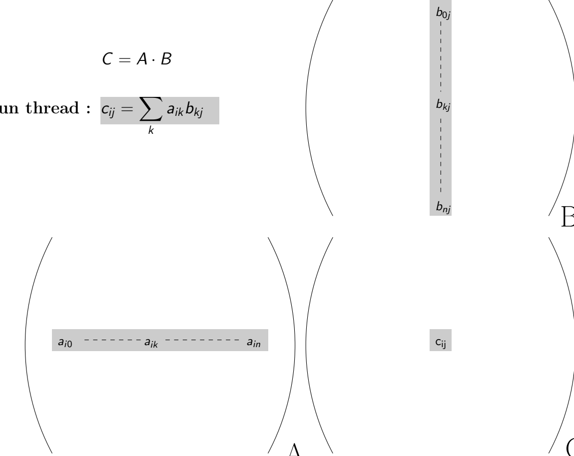
➲ optimisation du code portera alors soit sur les **accès mémoire** ou soit sur la **complexité arithmétique**

1. GPU (re)designed as a many core architecure
   - GPU computing
   - Multithreading (SIMT)
   - Co-processing
2. Programming in CUDA
   - Compilation
   - Parallelisation
3. A toy example : acceleration of matrix multiplication
   - GPU code
   - CPU code
   - Performance

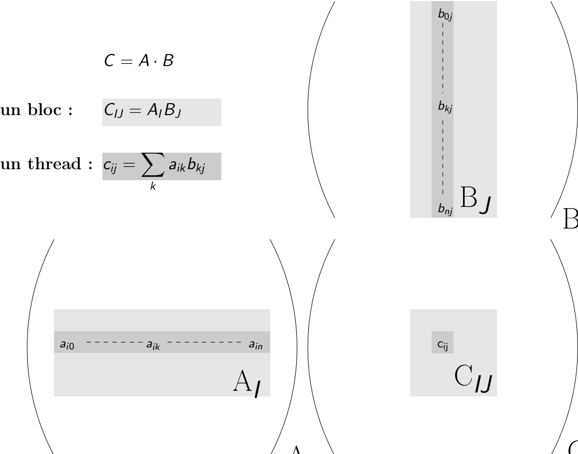
Parallélisation du calcul matriciel

$$C = A \cdot B$$

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



Découpage en blocs de threads



kernel = code des threads executés sur le GPU

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

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

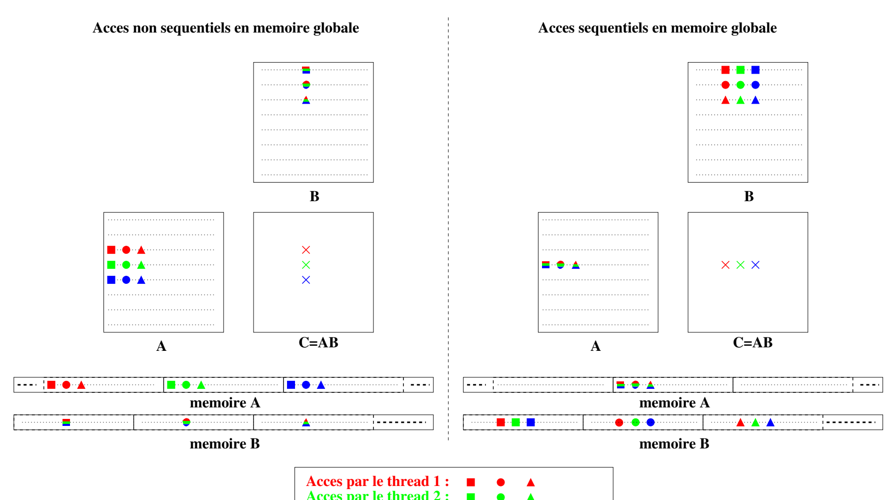
```
#define BLOCK_SIZE 16

void matrixMul_host(int N) {

// allocate host memory int mem_size=N^2*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

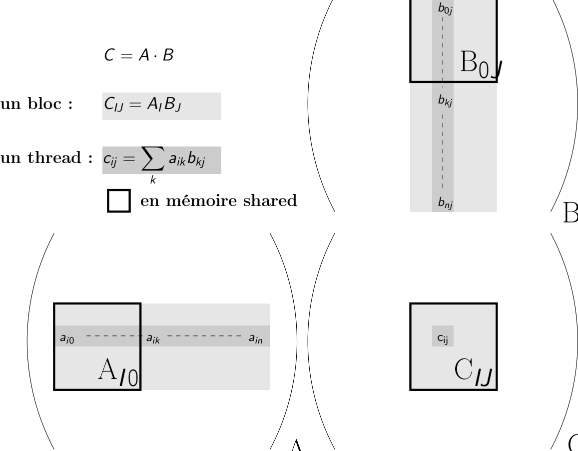


Temps GPU avec accès mémoire sequentiels

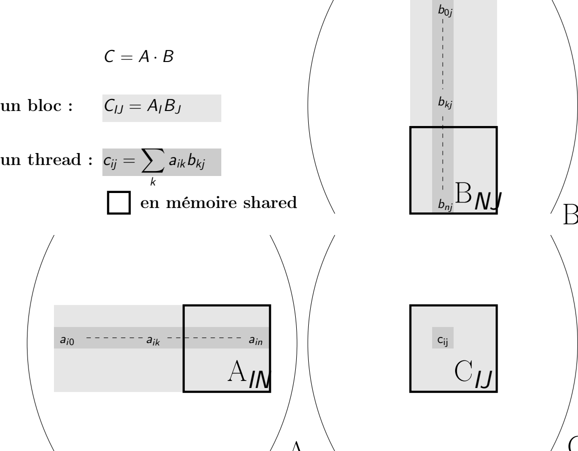
Matrices de taille 1024·1024

	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



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
cublasSgemm('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%