TP1: Programmation CUDA
“CUDA devices and threading”

P. Bakowski

Analyse d'un circuit/device nVidia

Dans le premier exercice nous allons analyser les fonctionnalités offertes par la carte graphique nVidia installée dans votre ordinateur.

Il s'agit d'écrire un programme qui sollicite l'environnement « runtime » de CUDA et lui demander les paramètres du circuit graphique – GPU.

Ces paramètres sont communiqués par le biais de la structure:

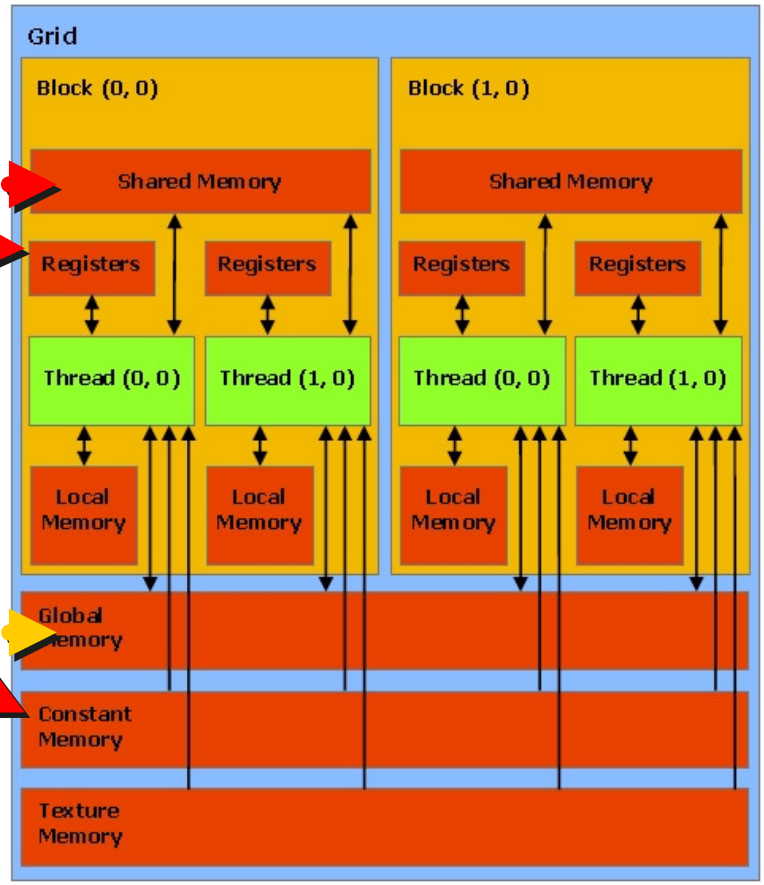
```
struct cudaDeviceProp
```

et la fonction:

```
cudaGetDeviceProperties(&deviceProp,device);
```

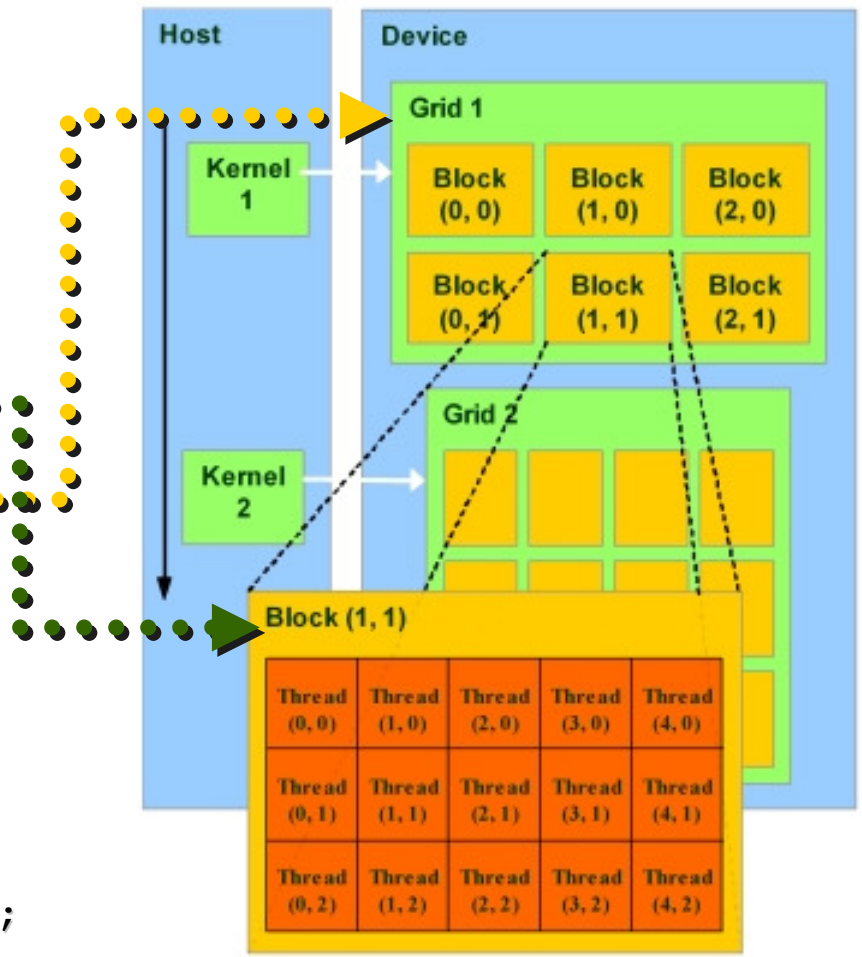
Analyse d'un circuit/device nVidia

```
struct cudaDeviceProp
{
    char name[256];
    size_t totalGlobalMem;
    size_t sharedMemPerBlock;
    int regsPerBlock;
    int warpSize;
    size_t memPitch;
    int maxThreadsPerBlock;
    int maxThreadsDim[3];
    int maxGridSize[3];
    size_t totalConstMem;
    int major;
    int minor;
    int clockRate;
    size_t textureAlignment;
    int deviceOverlap;
    int multiProcessorCount;
    int kernelExecTimeoutEnabled;
}
```



Analyse d'un circuit/device nVidia

```
struct cudaDeviceProp
{    .
     char name[256];
     size_t totalGlobalMem;
     size_t sharedMemPerBlock;
     int regsPerBlock;
     int warpSize;
     size_t memPitch;
     int maxThreadsPerBlock;
     int maxThreadsDim[3];
     int maxGridSize[3];
     size_t totalConstMem;
     int major;
     int minor;
     int clockRate;
     size_t textureAlignment;
     int deviceOverlap;
     int multiProcessorCount;
     int kernelExecTimeoutEnabled;
}
```



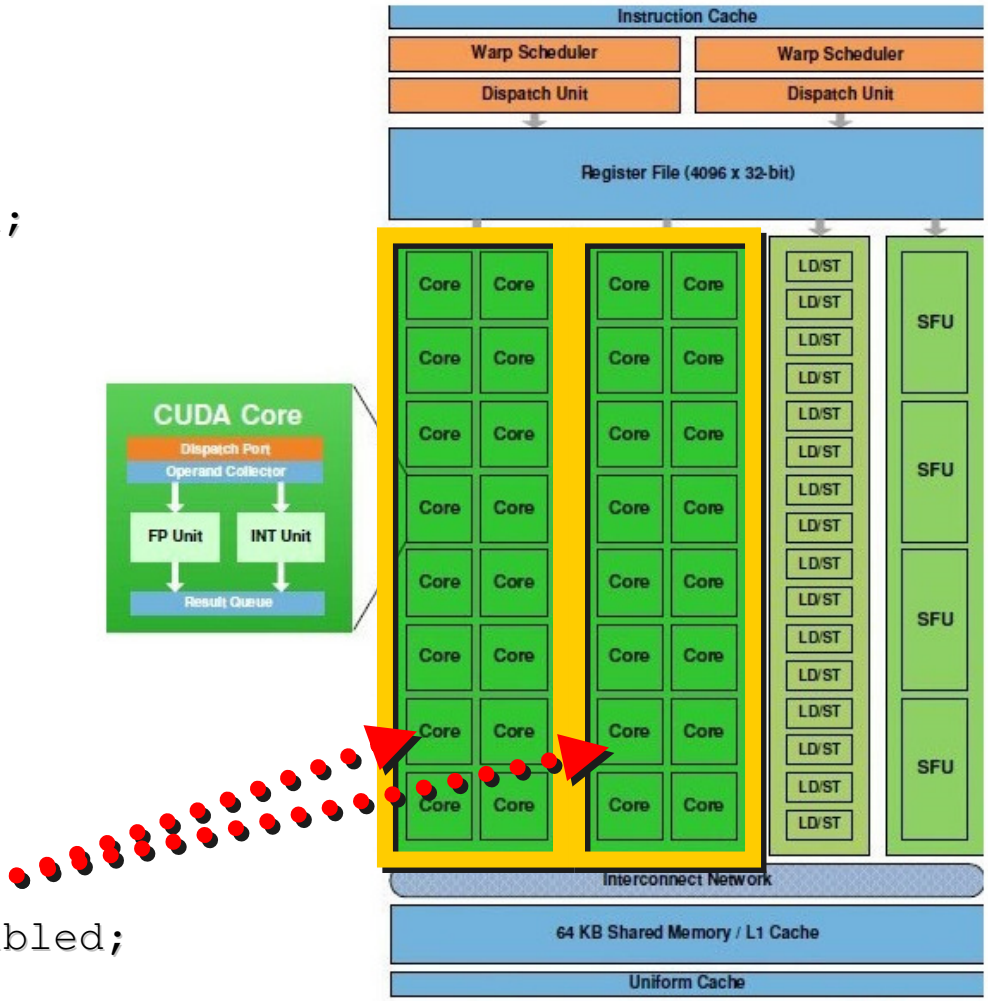
Analyse d'un circuit/device nVidia

```
struct cudaDeviceProp
{
    char name[256];
    size_t totalGlobalMem;
    size_t sharedMemPerBlock;
    int regsPerBlock;
    int warpSize;
    size_t memPitch;
    int maxThreadsPerBlock;
    int maxThreadsDim[3];
    int maxGridSize[3];
    size_t totalConstMem;
    int major;
    int minor;
    int clockRate;  // in KHz
    size_t textureAlignment;
    int deviceOverlap;
    int multiProcessorCount;
    int kernelExecTimeoutEnabled;
}
```

Compute capability (version)	GPUs
1.0	G80
1.1	G86, G84, G98, G96, G96b, G94, G94b, G92, G92b
1.2	GT218, GT216, GT215
1.3	GT200, GT200b
2.0	GF100
2.1	GF104, GF106, GF108

Analyse d'un circuit/device nVidia

```
struct cudaDeviceProp
{    .
     char name[256];
     size_t totalGlobalMem;
     size_t sharedMemPerBlock;
     int regsPerBlock;
     int warpSize;
     size_t memPitch;
     int maxThreadsPerBlock;
     int maxThreadsDim[3];
     int maxGridSize[3];
     size_t totalConstMem;
     int major;
     int minor;
     int clockRate;
     size_t textureAlignment;
     int deviceOverlap;
     int multiProcessorCount;
     int kernelExecTimeoutEnabled;
}
```



Analyse d'un circuit/device nVidia

	Multiprocessors	cores	Compute Capability
GeForce GTX 480	15	480	2.0
GeForce GTX 470	14	448	2.0
GeForce GTX 465	11	352	2.0
GeForce GTX 295	2×30	2×240	1.3
GeForce GTX 280/GTX	285 30	240	1.3
GeForce GTX 260	24	192	1.3
GeForce 210	2	16	1.2
GeForce GT 240	12	96	1.2
GeForce GT 220	6	48	1.2
GeForce GT 130	12	96	1.1
GeForce GT 120	4	32	1.1
GeForce GTS 250	16	128	1.1
GeForce 9800 GX2	2×16	2×128	1.1
GeForce 9800 GTX	16	128	1.1
GeForce 8800 GT	14	112	1.1
GeForce 8800 GTS	12	96	1.0
GeForce 8600 GT/GTS	4	32	1.1

Analyse du GPU (Exo0)

Exercice 0:

Ecrire un programme qui permet d'afficher:

- Le nom du circuit
- La taille de la mémoire globale
- La taille de la mémoire partagée (dans un bloc)
- Le nombre de registres par bloc
- Le nombre de *threads* par bloc
- Version: majeure, mineure
- Fréquence horloge (en KHz)
- Le nombre de multiprocesseurs

Analyse d'un circuit/device nVidia

Quelques éléments du programme:

```
#include <stdio.h> #include <stdlib.h>
#include <cuda.h>  #include <cuda_runtime.h>

int main(){
  cudaDeviceProp deviceProp;
  int device=0;
  cudaGetDeviceProperties(&deviceProp, device );
//
```

Attention:
Le code source doit être enregistré avec extension .cu;
la compilation nécessite l'utilisation du script nvcc.

```
%nvcc -o exo0 exo0.cu
```

Addition des vecteurs, premier kernel

Dans cet exercice nous allons écrire notre premier kernel – fonction exécutée par le GPU.
Le programme est composé de plusieurs parties:

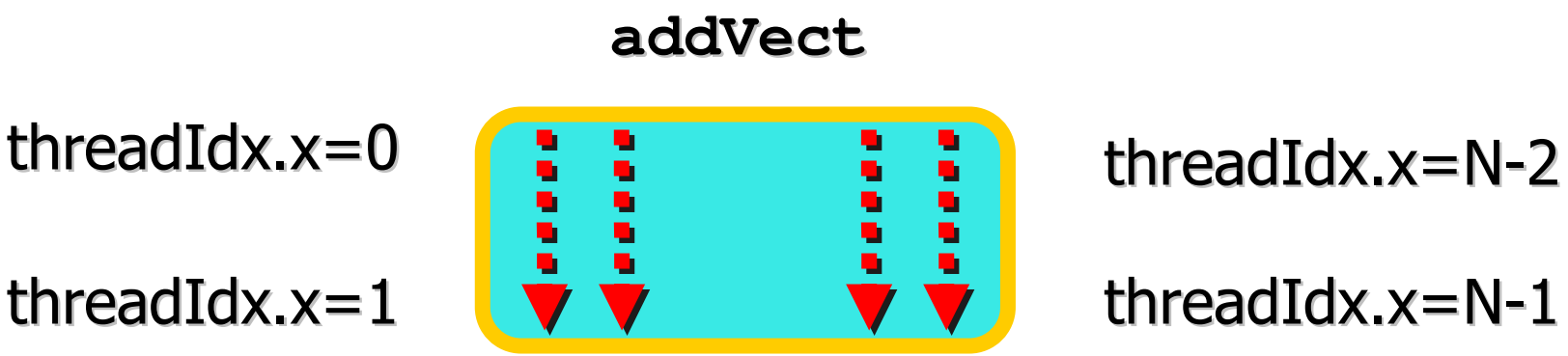
- Le kernel évoqué comme fonction globale:
  __global__ addVect(..)

- La fonction principale main() avec:
  - allocation de la mémoire globale du GPU
  - transfert des arguments dans cette mémoire
  - appel du kernel avec un triple chevron
    <<<..>>>
  - transfert des résultats de la mémoire GPU

Le kernel

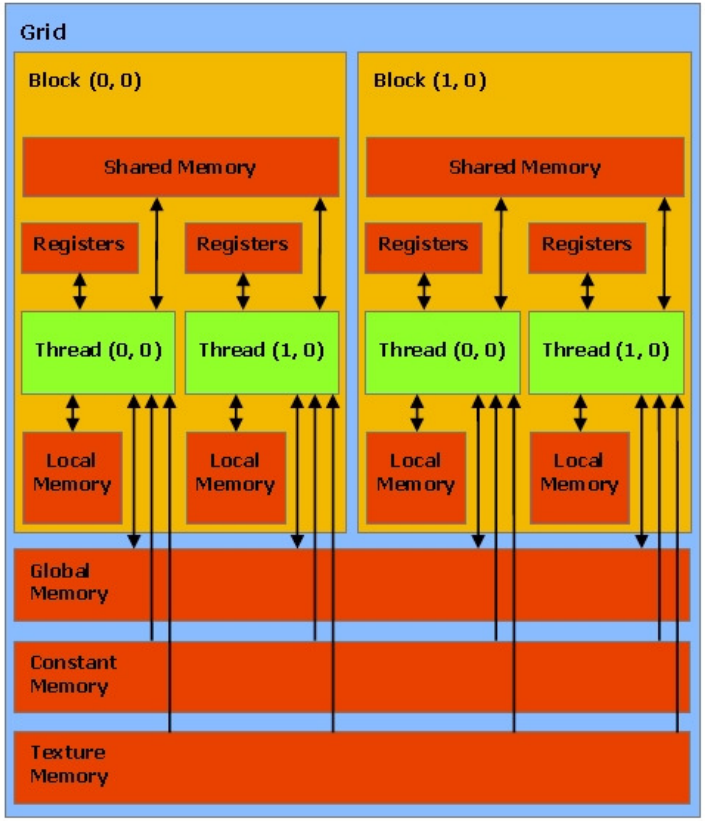
```
__global__ void addVect(float* in1,float* in2,float* out)
{
int i = threadIdx.x;
out[i] = in1[i] + in2[i];
}
```

La variable threadIdx.x est une variable automatique fournie par le compilateur; elle donne l'index du thread exécutant la fonction.



Allocation de la mémoire GPU

```
float* Cv1;
cudaMalloc((void **)&Cv1,memsize);
float* Cv2;
cudaMalloc((void **)&Cv2,memsize);
float* Cres;
cudaMalloc((void **)&Cres,memsize);
```



Attention:

A la fin d'exécution la mémoire GPU doit être libérée par:

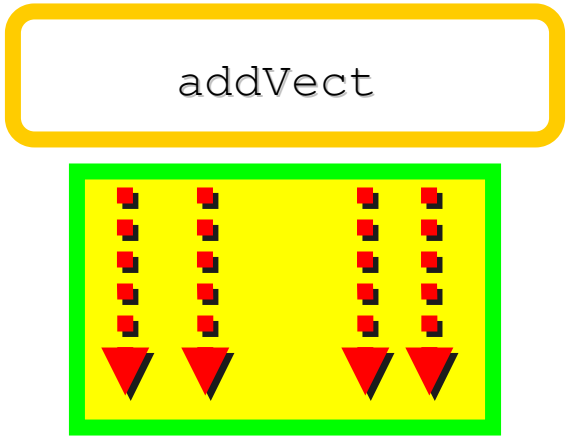
```
cudaFree(Cv1); cudaFree(Cv2);
cudaFree(Cres);
```

Transfert des données et appel du kernel

```
cudaMemcpy(Cv1,v1,memsize,cudaMemcpyHostToDevice);
cudaMemcpy(Cv2,v2,memsize,cudaMemcpyHostToDevice);

addVect<<<1,vsize>>>(Cv1,Cv2,Cres);

cudaMemcpy(res,Cres,memsize,cudaMemcpyDeviceToHost);
```



`<<<1,vsize>>>`

1 bloc, vsize kernels

Addition des vecteurs (Exo1a)

Exercice 1a:
Ecrire une application complète avec le kernel qui additionne deux vecteurs.
Les valeurs des vecteurs peuvent être préparées sous la forme de constantes:
Par exemple:

```
float v1[]={1,2,3,4,5,6,7,8,9,10};
float
v2[]={10.9,11.8,12.7,13.6,14.5,15.4,16.3,17.2,18.1,19.0};
```

Après le retour du kernel afficher les résultats.

Addition des vecteurs (Exo1b)

Exercice 1b:

Ecrire la même application en mettant en œuvre le mécanisme de blocs: Exemple de modifications à faire:

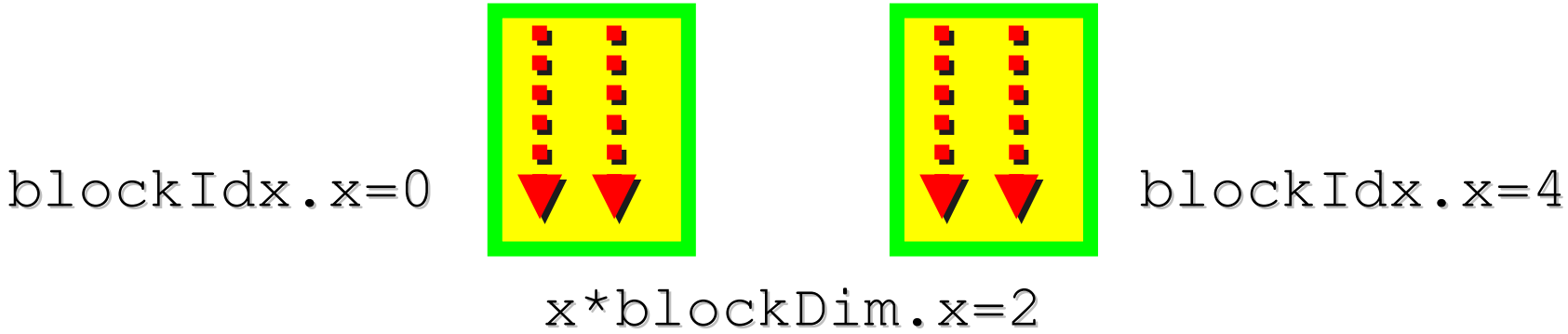
```
#define BLN 5
```

Appel du kernel:

```
addVect<<<BLN,vsize/BLN>>>(Cv1,Cv2,Cres);
```

Calcul de l'indexe dans le kernel:

```
int i = threadIdx.x + blockIdx.x*blockDim.x;
```

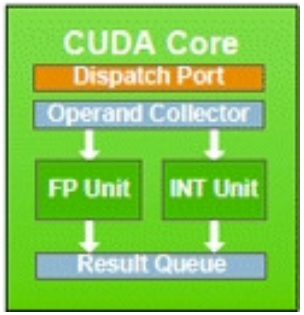


Multiplication des matrices

La multiplication des matrices, nécessitant des opérations de la multiplication et de l'addition sur plusieurs éléments indépendant, permet d'exploiter pleinement la puissance des circuits GPU.

i j 0*1 + 1*3 = 3

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 3 & 4 \\ 0 & 0 \end{bmatrix}$$



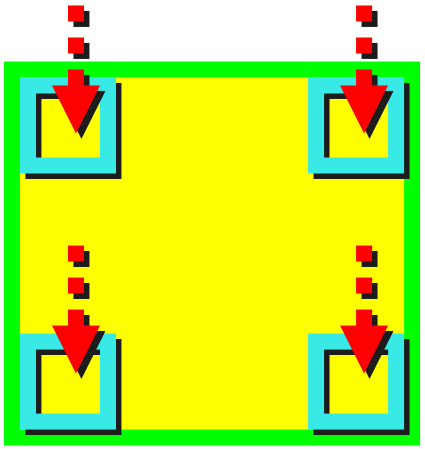
$$\mathbf{C}_{ik} = (\mathbf{A}\,\mathbf{B})_{ik} = \sum_{j=1}^{N} A_{ij} B_{jk}$$

Multiplication des matrices

L'organisation de threads en deux dimensions se prête très bien pour calculer les points dans la matrice des résultats. Nous allons donc utiliser les structures prédéfinies de type dim3

```
dim3 dimBlock(DIM,DIM);
dim3 dimGrid(1,1);
```

Chaque case de la matrice est traitée par un thread.
Chaque thread à deux identificateurs:
threadIdx.x et threadIdx.y



Multiplication des matrices - kernel

La fonction du kernel récupère les arguments suivantes:

```
// allocation on GPU
cudaMalloc((void **)&dev_A,DIM*DIM*sizeof(float));
cudaMalloc((void **)&dev_B,DIM*DIM*sizeof(float));
cudaMalloc((void **)&dev_C,DIM*DIM*sizeof(float));

// data transfer to GPU
int sizemat= DIM*DIM*sizeof(float),
cudaMemcpy(dev_A,buffA, sizemat, cudaMemcpyHostToDevice);
cudaMemcpy(dev_B,buffB, sizemat, cudaMemcpyHostToDevice);
```

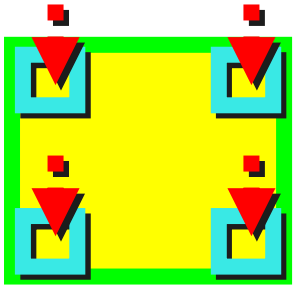
Multiplication des matrices - kernel

Elle est appelée par:

```
dim3 dimBlock(DIM,DIM);  dim3 dimGrid(1,1);
matrix_mul<<<dimGrid,dimBlock>>>(dev_A,dev_B,dev_C,DIM);
```

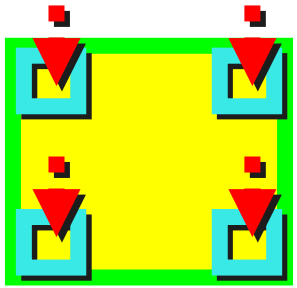
Son code est:

```
__global__ void matrix_mul(float* dev_A,float* dev_B,
float* dev_C, int Width)
  {
  int tx = threadIdx.x; int ty = threadIdx.y;
  float Pvalue =0;
  for(int k=0;k<Width;++k)
  {
    float Ael=dev_A[ty*Width + k];
    float Bel=dev_B[k*Width +tx];
    Pvalue += Ael*Bel;
  } dev_C[ty*Width+tx]= Pvalue;
}
```



Multiplication des matrices - CPU

Ci-dessous le code de multiplication des matrices réalisé pour le CPU.
Ici le traitement des cases sont réalisés séquentiellement par 2 boucles for, une pour la dimension verticale (i) et une pour la dimension horizontale (j).



```
for(int i=0;i<DIM;i++)
  for(int j=0;j<DIM;j++)
    for(int k=0;k<DIM;k++)
      buffC[j + i*DIM] += buffA[k+j*DIM]*buffB[j+k*DIM];
```

Analyse des performances

Avec CUDA l'analyse de performances peut être réalisée par les mécanismes d'événements.
Les événements peuvent être initialisés par:

```
float elapsedTime;
cudaEvent_t start,stop;    // declaration
cudaEventCreate(&start);   // creation
cudaEventCreate(&stop);
cudaEventRecord(start,0); // activation
// execution
cudaEventRecord(stop,0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsedTime,start,stop);
// display elapsedTime
```

Sujet: comparaison des performances

Sujet:

Ecrire une application de multiplication des matrices avec 2 modes de fonctionnement: mode GPU et mode CPU

Comparer le temps d'exécution à l'aide des événements.

Pour obtenir les résultats probants utiliser les matrices carrées de taille (DIM) minimum de 64*64. Le contenu des matrices peut être initialisé par la fonction rand():

```
float* random_block(int size)
{
float  *ptr;
ptr = (float *)malloc(size*sizeof(float));
for (int i=0;i<size;i++)
  ptr[i] = rand();
return ptr;
}
```

Résumé

- Dans ce TP nous avons étudié les principales caractéristiques fonctionnels des unités GPU.
- Nous avons étudié et écrit quelques exemples simples de programmation CUDA incluant:
  - analyse d'un « device » CUDA
  - écriture d'un kernel simple
  - écriture d'un kernel à deux dimensions de traitement
  - analyse des performances basée sur les événements CUDA