INSA-5IF

Programmation massivement parallèle sur GPU

Séance 2 : introduction à CUDA

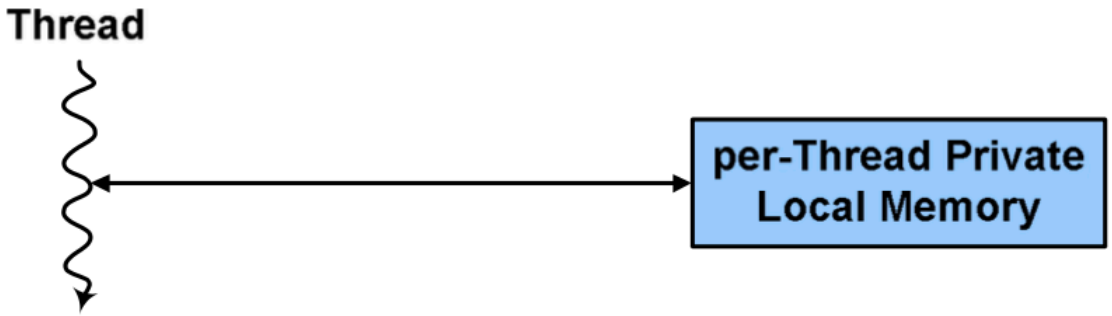
Christian Wolf
INSA-Lyon, Dép. IF, LIRIS

Planning

Christian W.	Cours	Introduction : architecture d'un GPU
Jonathan R.C.	Cours	Introduction : calcul en parallèle (1/2)
Jonathan R.C.	Cours	Introduction : calcul en parallèle (2/2)
Jonathan R.C.	Cours	Open-MP
Jonathan R.C.	TP	Open-MP
Christian W.	Cours + TP	Programmation en CUDA : introduction
Jonathan R.C.	TP	Open-MP
Christian W. + Lionel M.	Cours + TP	Programmation en CUDA : mémoires
Jonathan R.C.	TP	MPI
Jonathan R.C.	TP	MPI
Jonathan R.C.	TP	Runtime
Christian W.	Cours + TP	Débogage et optimisation sur GPU
Christian W.	Cours + TP	GPU Design patterns
Lionel M.	Cours + TP	Open-ACC
Lionel M.	Cours + TP	Open-ACC

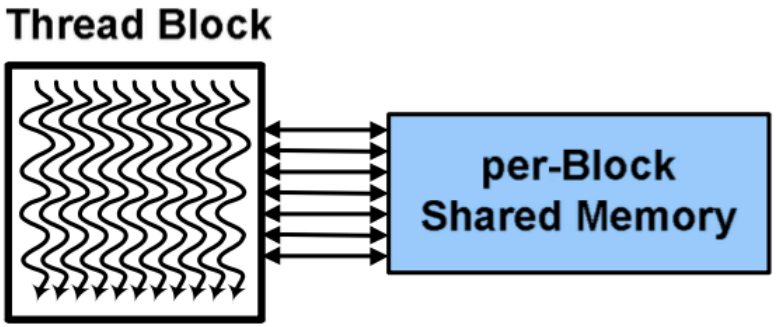
Rappel : *thread*

- Une instance d'un *kernel*
- Dispose d'un PC (*program counter*), registres, mémoire privée, entrées et sorties
- Une ID propre dans un *block*



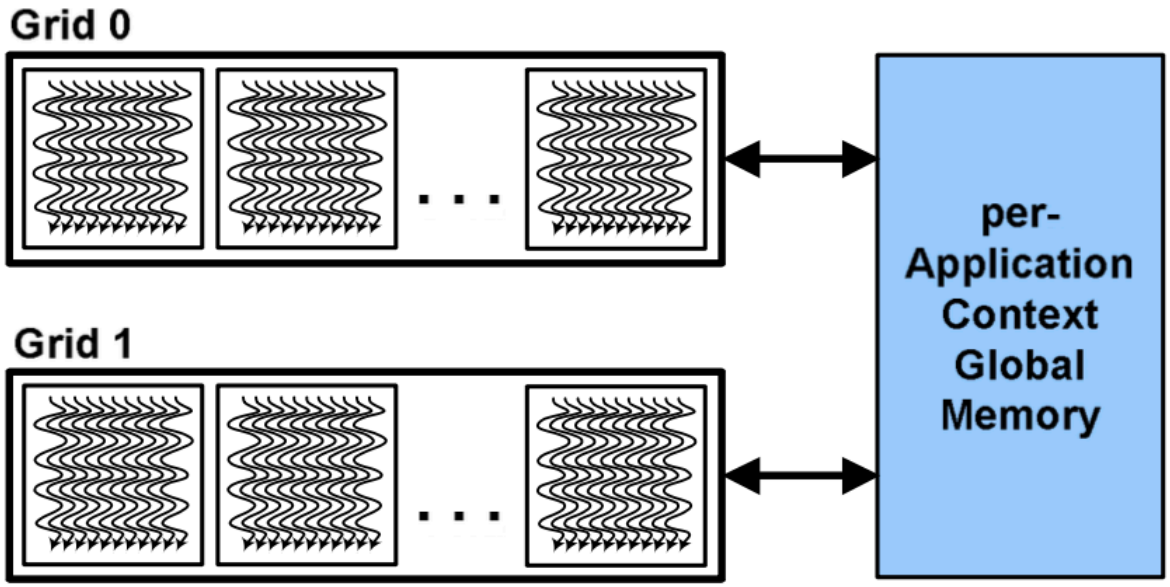
Rappel : *block*

- Un ensemble de *threads* pouvant coopérer :
  - Synchronisation
  - Mémoire partagée
  - Une ID propre dans une *grid*

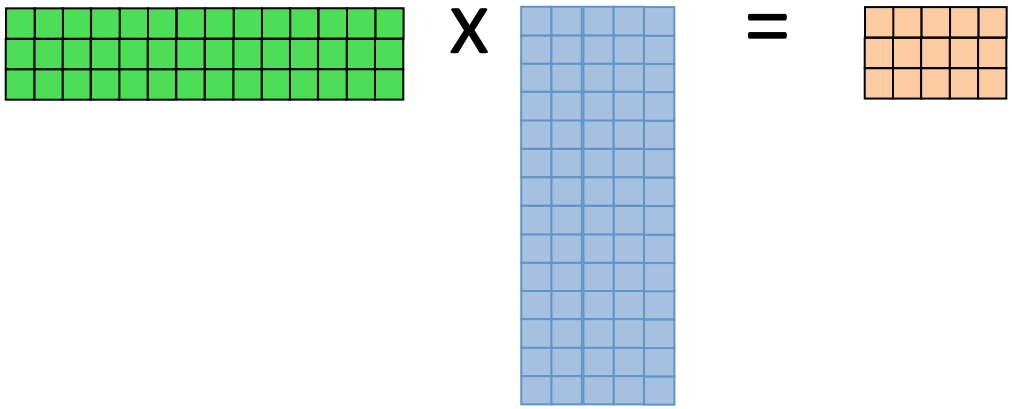


Rappel : *grid*

- Tableau de *block* exécutant le même *kernel*
- Peuvent accéder à la mémoire globale du GPU
- Synchronisation uniquement en terminant le *kernel* actuel et puis en démarrant un nouveau *kernel.*



Exemple : multiplication de matrices



La solution séquentielle classique :

```
1     for( int r = 0; r < h; r++)
2     {
3         for( int c = 0; c < w; c++)
4         {
5             m1xm2[r*w + c] = 0;
6
7             for( int k = 0; k < w; k++)
8                 m1xm2[r*w + c] += m1[r*w + k] * m2[k*w + c];
9         }
10    }
```

Solution parallèle

Exécution en parallèle d'une fonction ayant comme paramètre l'indice du résultat à calculer :

```
1 void mult_kernel_simple(int c, int r)
2 {
3     output[r*mxWidth + c] = 0.0f;
4     for( int k = 0; k < mxWidth; k++)
5         output[r*mxWidth + c] += mx1[r*mxWidth + k] * mx2[k*mxWidth + c];
6 }
```

Appeler en parallèle pour tous c, r

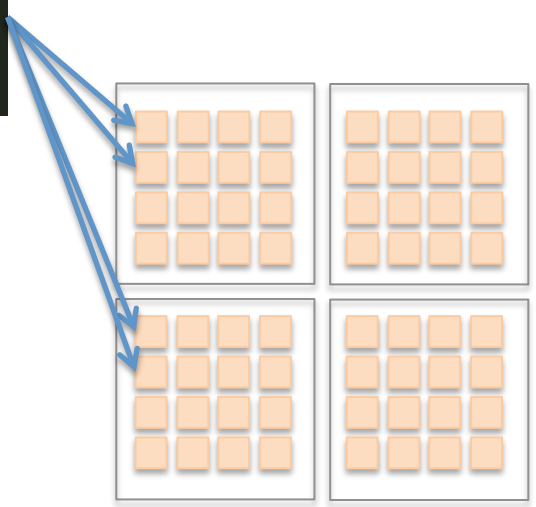
En CUDA cette fonction est appelée un « noyeau » (*kernel).*

Chaque paire (c,r) correspond à un *thread.*

Organisation en *blocks*

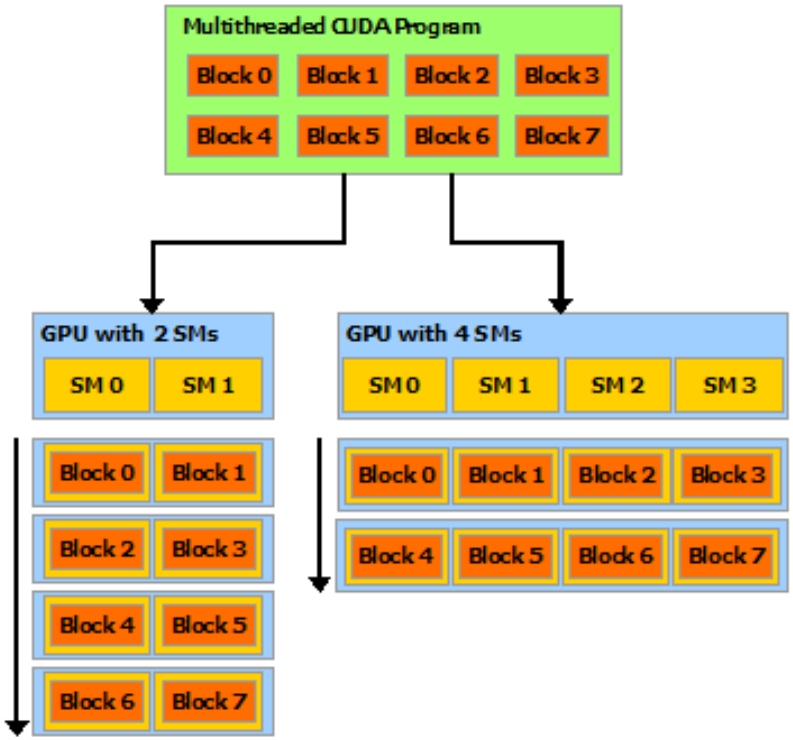
- Les noyaux seront organisés en *blocks*
- Dans un premier temps, notre découpage sera quelconque (à optimiser ultérieurement)
- Partage de mémoire possible parmi les *threads* d'un *block* (non traité dans l'exemple)

```
1 void mult_kernel_simple(int c, int r)
2 {
3     output[r*mxWidth + c] = 0.0f;
4     for( int k = 0; k < mxWidth; k++)
5         output[r*mxWidth + c] += mx1[r*mxWidth + k] * mx2[k*mxWidth + c];
6 }
```



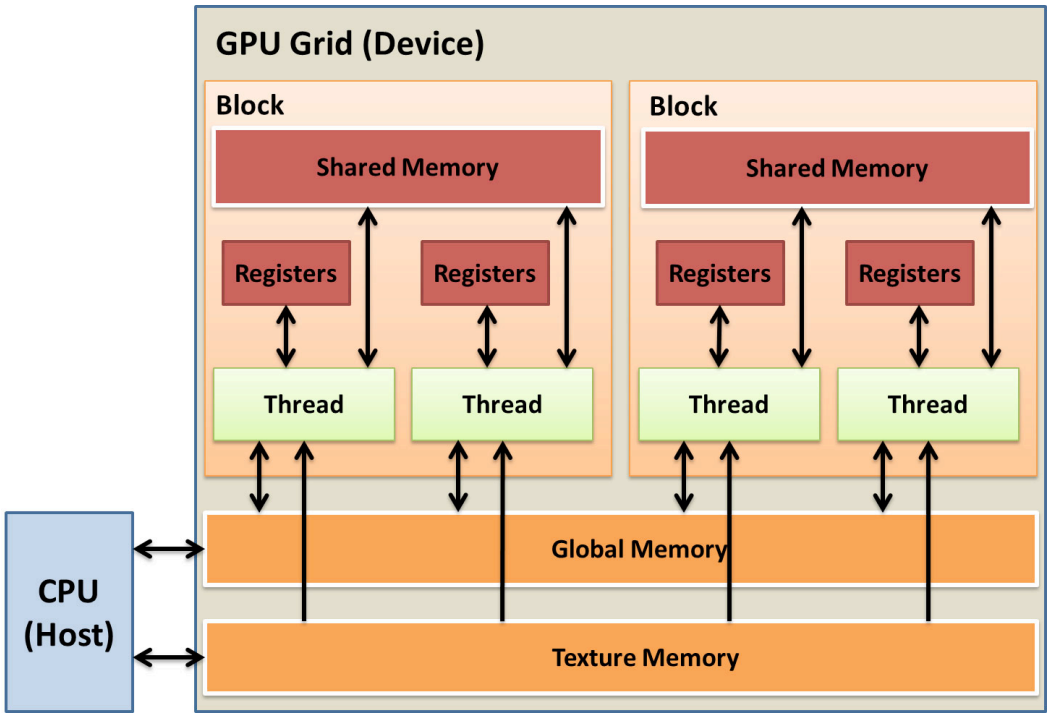
SM

Un block sera associé entièrement à un SM (*streaming multi-processeur)*



Séquence classique de code CUDA

- Le CPU alloue de la mémoire sur le GPU
- Le CPU copie les données dans la mémoire globale du GPU
- Le CPU lance l'exécution du *kernel* sur le GPU
- Le GPU exécute le *kernel*
- Le CPU copie les résultats depuis la mémoire globale du GPU



La version « CUDA » du noyau

Mot clé déclarant un *kernel*

```
1  __global__
2  void mult_kernel_simple(int mxWidth, float *mx1, float *mx2, float *output)
3  {
4      int c = blockIdx.x*blockDim.x + threadIdx.x;
5      int r = blockIdx.y*blockDim.y + threadIdx.y;
6
7      output[r*mxWidth + c] = 0.0;
8      for( int k = 0; k < mxWidth; k++)
9          output[r*mxWidth + c] += mx1[r*mxWidth + k] * mx2[k*mxWidth + c];
10 }
```

Indice du *block* au sein du *grid*

Indice du *thread* au sein du *block*

Appel du noyau

```
1  int main(void)
2  {
3      ...
4      // allocate memory on GPU
5      cudaMalloc( (void**) &gpuMatrix1, matrixSizeInBytes);
6      ...
7
8      // copy data from CPU memory to GPU memory
9      cudaMemcpy(gpuMatrix1, matrix1, matrixSizeInBytes, cudaMemcpyHostToDevice);
10     ...
11
12     // Set grid and block size
13     dim3 dimBlock(32, 32);
14     dim3 dimGrid(matrixWidth/dimBlock.x, matrixWidth/dimBlock.y);
15
16     // run kernel
17     mult_kernel_simple<<<dimGrid, dimBlock>>>( matrixWidth, gpuMatrix1, gpuMatrix2, gpuO
18
19     // copy back results from GPU memory to CPU memory
20     cudaMemcpy( outputData, gpuOutput, matrixSizeInBytes, cudaMemcpyDeviceToHost);
21     ...
22 }
```

Appel du *kernel* en parallèle pour chaque *thread*

Vérification des erreurs (1)

Obligation!

```
ok=cudaMemcpy (resarr, _resarr, size, cudaMemcpyDeviceToHost);
if (ok!=cudaSuccess)
{
    std::cerr << "*** Could not transfer data back from GPU!\n";
    exit (1);
}

ok=cudaFree(_imarr);
if (ok!=cudaSuccess)
{
    std::cerr << "*** Could not free GPU memory!\n";
    exit (1);
}
ok=cudaFree(_resarr);
```

Vérification des erreurs (2)

```
// kernel call: tree prediction in parallel, on the GPU
treeKernel<<<dimGrid,dimBlock>>>(gpuFeatures, gpuFeaturesIntegral, gpuResult,
    w, h, w_integral, h_integral,
    gpuU8, gpuI8, gpuI16, gpuU32, gpuF,
    nodeCount, gpuxLDim, gpuyLDim, numLabels);

// make the host block until the device is finished with foo
cudaThreadSynchronize();

// check for error
ok = cudaGetLastError();
std::cerr << "CUDA status: " << cudaGetErrorString(ok) << "\n";
if (ok!=cudaSuccess)
{
    std::cerr << "*** Error at kernel launch!\n";
    exit(1);
}
```

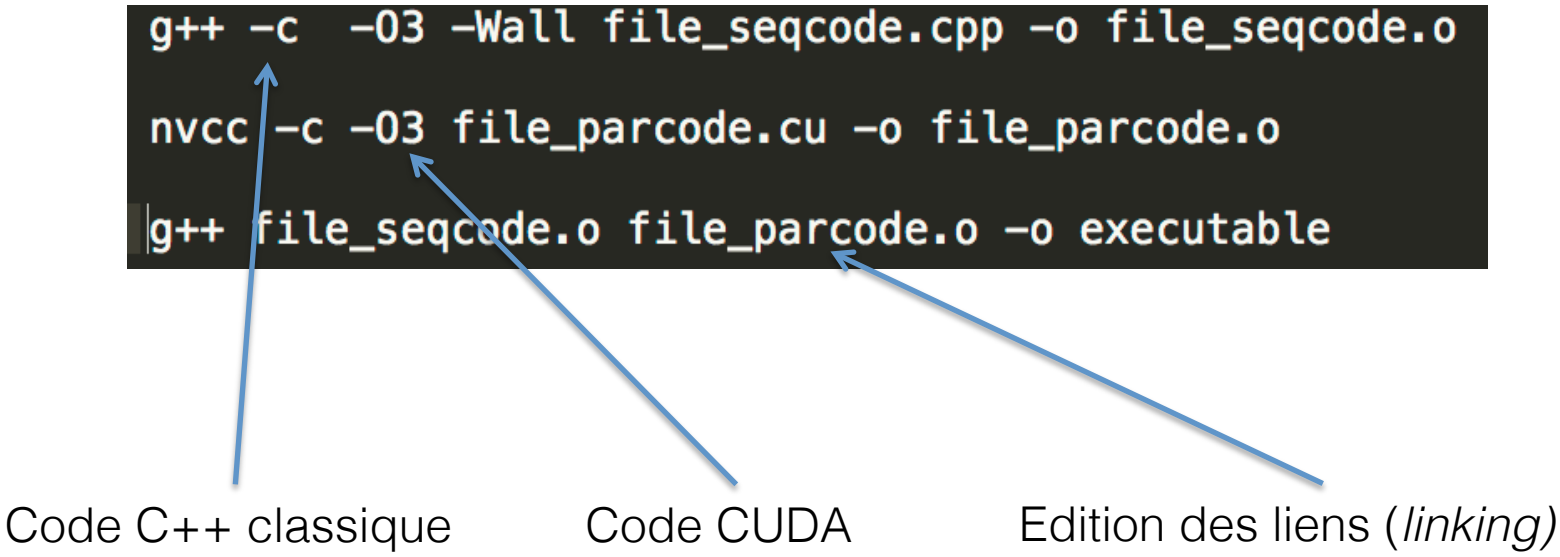
Info mémoire utilisée

```
size_t f, tot;
cudaMemGetInfo(&f, &tot);
std::cerr << "Free: " << f/(1024*2024) << " Total: " << tot/(1024*2024) << "\n";
```

Compilation

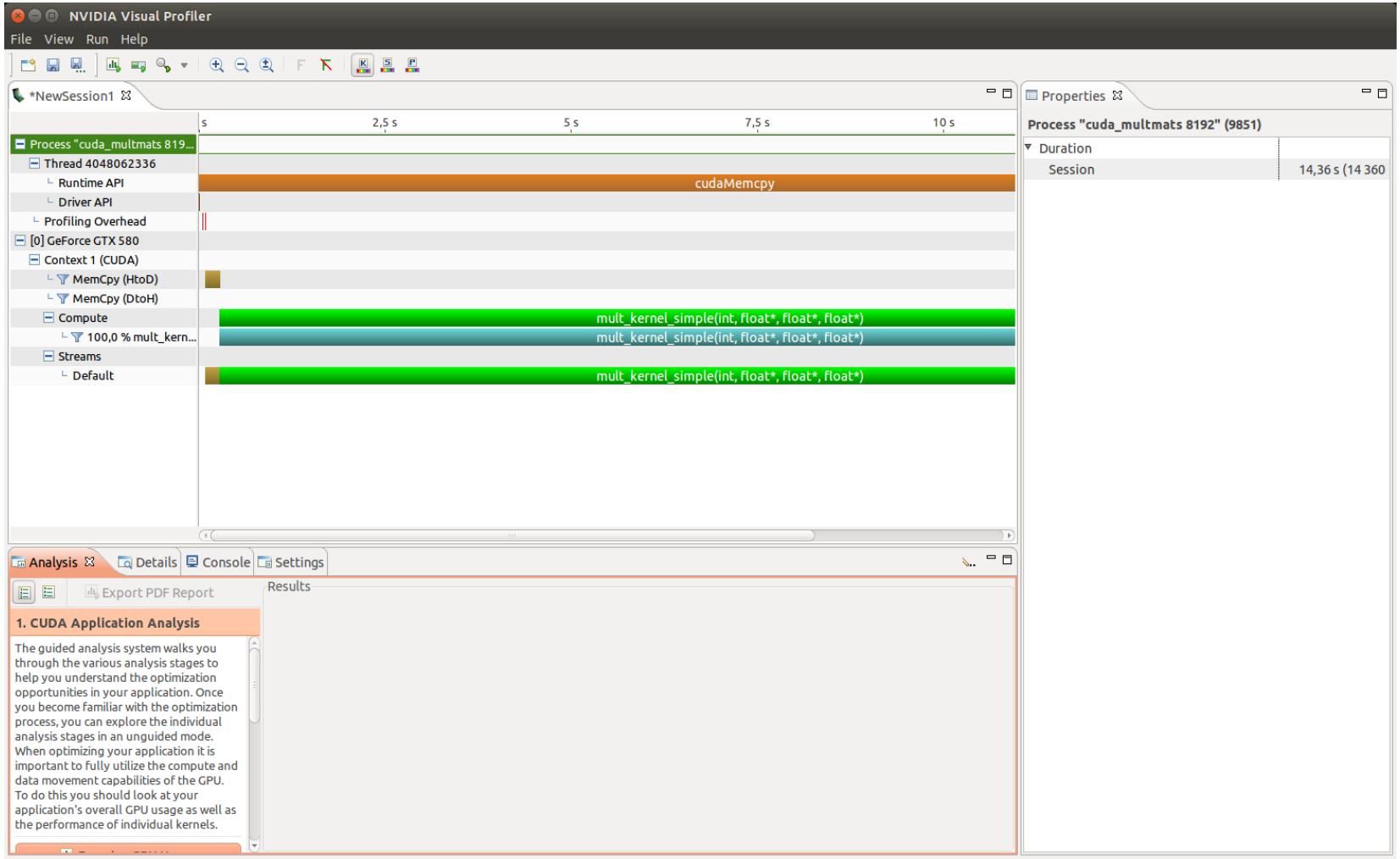
- CUDA utilise un compilateur spécifique, qui fait appel à un compilateur classique C++ (gcc)
- Généralement on sépare le code CUDA du code classique (non obligatoire)

```
g++ -c  -O3 -Wall file_seqcode.cpp -o file_seqcode.o

nvcc -c -O3 file_parcode.cu -o file_parcode.o

g++ file_seqcode.o file_parcode.o -o executable
```



Outils de débogage

nvvp est livré avec CUDA



Bonne pratiques

GPU Optimization Fundamentals

- Find ways to parallelize sequential code
- Adjust kernel launch configuration to maximize device utilization
- Ensure global memory accesses are coalesced
- Minimize redundant accesses to global memory
- Avoid different execution paths within the same warp
- Minimize data transfers between the host and the device

http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/

Quelques pièges

- Valider le code GPU par un résultat de référence obtenu sur CPU
- Le *kernel* sur GPU peut échouer plus ou moins silencieusement
- Attention à la persistance des résultats dans la mémoire du GPU
- Arrondi des *floats* différent entre CPU et certains GPU

Partie « TP » : filtrage d'images

Nous implémenterons une méthode permettant de traiter des images numériques et d'appliquer des filtres, tels que le lissage ou la détection de contours. L'exemple suivant illustre un résultat typique se servant de cette fonctionnalité :

Input Image

res_smoothed.png

res_edges.png

Une image

Ici nous traitons des images dites « de niveaux de gris « où chaque pixel peut prendre des valeurs entre 0 (noir) et 255 (blanc). Une image est représentée par la classe **`cv::Mat`** de la bibliothèque « open-cv ».

Lecture à partir d'un fichier :

```
cv::Mat im = cv::imread(inputfname,-1);
if (!im.data)
{
    std::cerr << "*** Cannot load image: " << inputfname << "\n";
    exit(1);
}
```

Accès aux pixels :

```
im.at<unsigned char>(y,x)
```

Les convolutions

Le principe du filtrage linéaire (a.k.a. « convolution ») est simple : la valeur de chaque pixel est remplacée par une somme pondérée impliquant sa valeur d'origine et les valeurs de ses voisins. Les poids w de cette combinaison linéaire sont les paramètres de ce filtre.

$$result = \left(\sum_{i=1}^{9} w_i I_i \right)$$

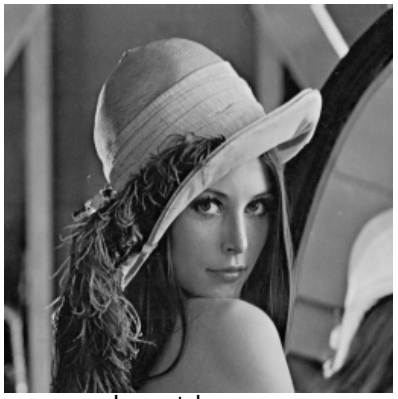
Exemple d'un filtre

Le filtre Gaussien est un cas spécifique d'un filtre linéaire dont les poids sont donnés par le masque suivant :

1	2	1
2	4	2
1	2	1

* 1/16

Effet de lissage :



Input Image

res_smoothed.png

Problème

- Implémentation d'une version CPU de ce problème
- Implémentation d'une version GPU en parallèle (un *thread* par pixel)
- Evaluation des performances

Un fragment de code est disponible sur moodle.