

# Calcul GPU – Cours 1: Introduction

Jonathan Rouzaud-Cornabas

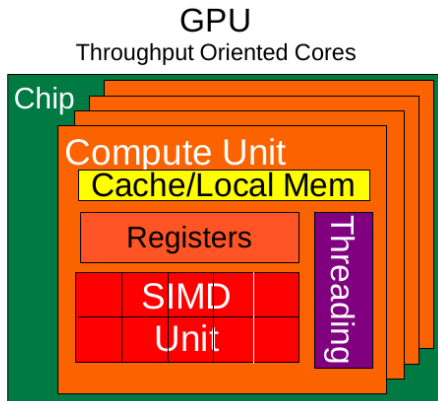
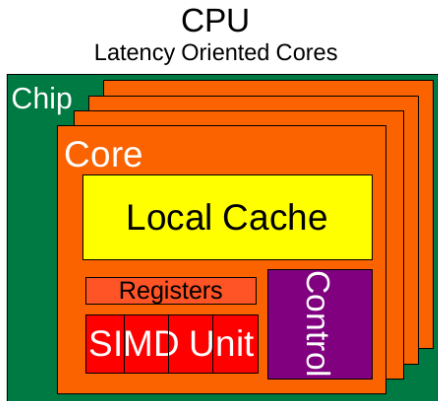
LIRIS / Insa de Lyon – Inria Beagle

Cours inspiré de ceux de Prof Wen-mei Hwu  
(University of Illinois at Urbana–Champaign)

# Références

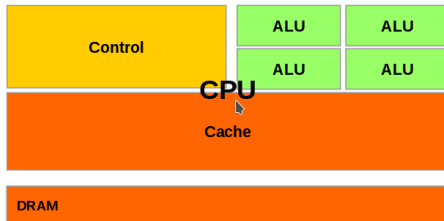
- D. Kirk and W. Hwu, “Programming Massively Parallel Processors – A Hands-on Approach,” Morgan Kaufman Publisher, 3rd edition
- NVIDIA, NVidia CUDA C Programming Guide, version 8.0, NVidia, 2016 (reference book)

# CPU: Conception orienté latence



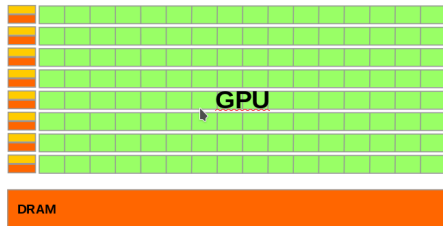
# CPU: Conception orienté latence

- Haute fréquence d'horloge
- Caches de grandes tailles
  - Converti les accès à haute latence dans la mémoire en accès à basse latence dans le cache
- Système de contrôle sophistiqué
  - Prédiction de branche pour réduire la latence dû aux branches
  - Chargement de données pour réduire la latence dû à l'accès aux données
- Puissante unité arithmétique et logique (ALU)
  - Réduction de la latence des opérations



## GPU: Conception orienté débit

- Fréquence d'horloge modérée
- Caches de petites tailles
  - Pour maximiser le débit mémoire
- Contrôle simple
  - Pas de prédiction de branches
  - Pas de chargement de données
- Unité arithmétique et logique (ALU) à faible consommation
  - Nombreuses, à haute latence mais fortement pipeliner pour de fort débit
- Nécessite un très grand nombre de threads pour que la latence soit tolérable



# Les applications peuvent bénéficier des deux

- CPUs pour les parties séquentielles où la latence est critique
  - CPUs peuvent être  $10+X$  plus rapide que les GPUs pour le code séquentiel
- GPUs pour les parties parallèles où le débit est critique
  - GPUs peuvent être  $10+X$  plus rapide que les CPUs pour le code parallèle

# La stratégie gagnante est d'utiliser les deux

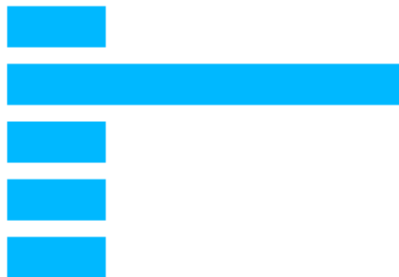
- CPUs pour les parties séquentielles où la latence est critique
  - CPUs peuvent être  $10+X$  plus rapide que les GPUs pour le code séquentiel
- GPUs pour les parties parallèles où le débit est critique
  - GPUs peuvent être  $10+X$  plus rapide que les CPUs pour le code parallèle

## Répartition de charges

Le temps total d'exécution d'une application parallèle est limitée par le thread qui met le plus de temps à finir



good



bad!



# Bande passante mémoire globale

**Ideal**



**Reality**

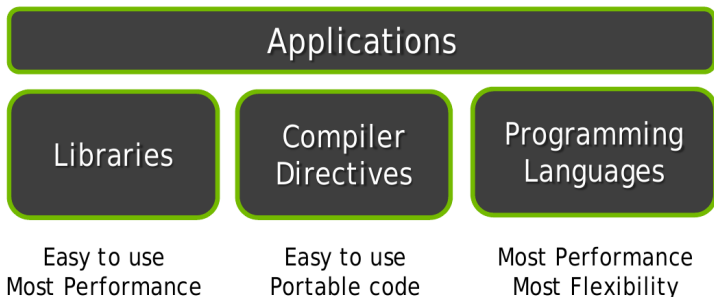


## Conflit des accès aux données: sérialisation et délais

- Les applications massivement parallèles ne peuvent pas se permettre la sérialisation
- La contention provoquée par un accès concurrent à une ressource critique provoque la sérialisation



## 3 méthodes d'accélération du code



## Bibliothèques: Facile à utiliser et rapide

- **Facile à utiliser** : les bibliothèques facilitent l'utilisation des GPUs sans nécessité une connaissance en profondeur de leur programmation
- **Facilement interchangeable** : La plupart des bibliothèques accélérées GPU suivent des APIs standards et donc diminuent la quantité de code à modifier
- **Qualité** : Les bibliothèques offrent une haute qualité d'implémentations pour des fonctions courantes

# Exemple de bibliothèques accélérées GPU

Linear Algebra  
FFT, BLAS,  
SPARSE, Matrix



CUDA|tools



CUSP

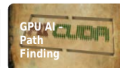
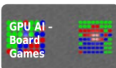
Numerical & Math  
RAND, Statistics



ArrayFire



Data Struct. & AI  
Sort, Scan, Zero Sum



Visual Processing  
Image & Video



NVIDIA  
Video  
Encode



## Addition de vecteurs en Thrust

```
thrust::device_vector<float> thrust::device_vector<float> thrust::  
    device_vector<float> deviceInput1(inputLength);  
  
deviceInput2(inputLength);  
  
deviceOutput(inputLength);  
  
thrust::copy(hostInput1, hostInput1 + inputLength,  
             deviceInput1.begin());  
  
thrust::copy(hostInput2, hostInput2 + inputLength,  
             deviceInput2.begin());  
  
thrust::transform(deviceInput1.begin(), deviceInput1.end(),  
                 deviceInput2.begin(), deviceOutput.begin(),  
                 thrust::plus<float>());
```

## Directives de compilation: Facile et portable

- **Facile à utiliser** : le compilateur prend en charge les détails comme la gestion du parallélismes et les déplacements mémoires
- **Portable** : Le code est générique, pas spécifique à un matériel
- **Instable** : La performance du code est très liée au compilateur (voir la version du compilateur)

# OpenACC

```
#pragma acc parallel loop
    copyin(input1[0:inputLength],
           input2[0:inputLength]),
    copyout(output[0:inputLength])
for(i = 0; i < inputLength; ++i) {
    output[i] = input1[i] + input2[i];
}
```



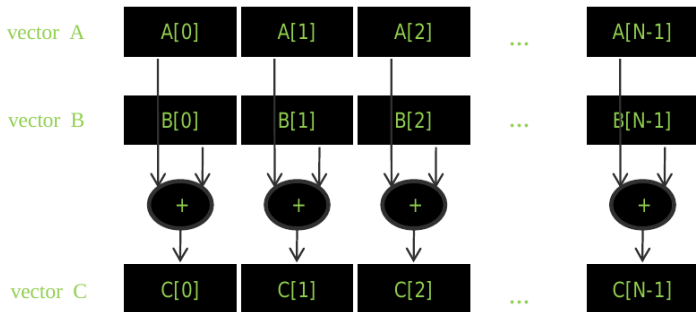
# OpenMP

```
#pragma omp target data map(to:input1)
#pragma omp target data map(to:input2)
#pragma omp target data map(out:output)
#pragma omp teams distribute parallel for
for(i = 0; i < inputLength; ++i) {
    output[i] = input1[i] + input2[i];
}
```

# Langages de programmation: le plus performant et le plus flexible

- **Performance** : Le programmeur a le meilleur contrôle sur le parallélisme et les mouvements de données
- **Flexible** : Pas besoin de se conformer à des interfaces de bibliothèques limitées ou de directives
- **Verbeux** : Le programmeur a souvent besoin de décrire tous les détails d'implémentation

# Parallélisme de données: Addition de vecteurs

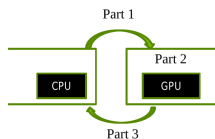


## Addition de vecteurs en C

```
// Compute vector sum C = A + B
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int i;
    for (i = 0; i < n; i++) h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N elements
    ...
    vecAdd(h_A, h_B, h_C, N);
}
```

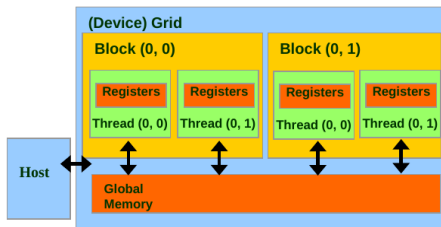
# Addition de vecteurs en C



```
// Compute vector sum C = A + B
void vecAdd(float *h_A, float *h_B,
            float *h_C, int n)
{
    int i;
    for (i = 0; i < n; i++) h_C[i] = h_A[i]
                                + h_B[i];
}

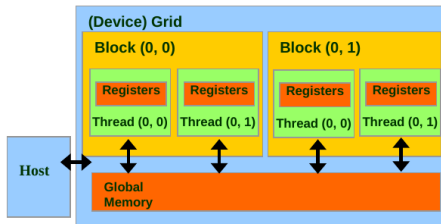
int main()
{
    // Memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N elements
    ...
    vecAdd(h_A, h_B, h_C, N);
}
```

# Les mémoires/caches de base en CUDA



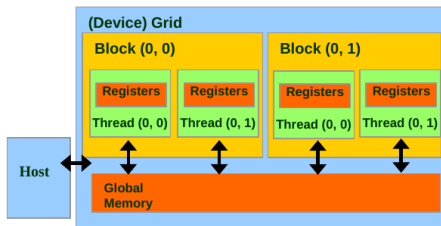
- Le code device peut :
  - lire/écrire les registres (par threads)
  - lire/écrire l'ensemble de la mémoire globale
- Le code hôte peut
  - Transférer des données depuis / vers la mémoire globale du GPU

# API de la gestion de la globale mémoire en CUDA



- `cudaMalloc()`
  - Allouer un objet dans la mémoire globale du GPU
  - Deux paramètres
    - Adresse du pointeur où sera allouée l'objet
    - Taille (en bytes) de l'objet à allouer
- `cudaFree()`
  - Libère l'objet dans la mémoire globale
  - Un paramètre
    - Le pointeur vers l'objet à libérer

# API de transfer mémoire *CPU* <> *GPU* en CUDA



## cudaMemcpy()

- Transfert de données vers/depuis la mémoire GPU
- Nécessite 4 paramètres
  - Pointeur de la destination
  - Pointeur de la source
  - Nombre de bytes à copier
  - Type/Direction du transfert
- ATTENTION : Le transfert est asynchrone



## Code hôte : Addition de vecteurs en CUDA

```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n * sizeof(float); float *d_A, *d_B, *d_C;
    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_C, size);
    // Kernel invocation code      to be shown later
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

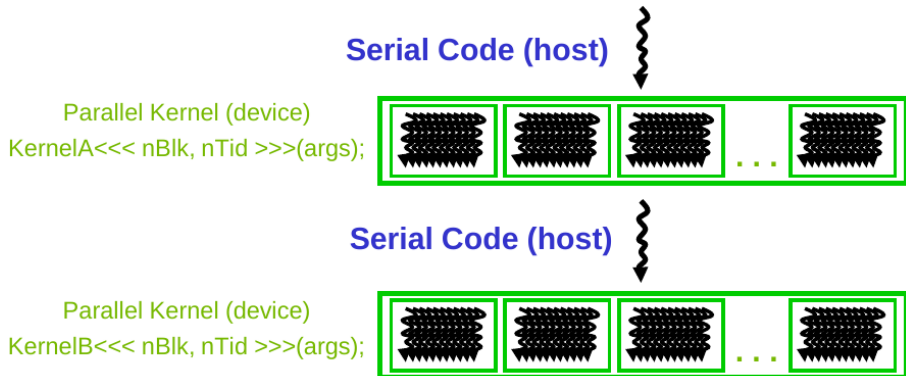
## Penser à vérifier les codes de retour d'erreurs

```
cudaError_t err = cudaMalloc((void **) &d_A, size);
if (err != cudaSuccess) {
    printf(" %s in %s at line %d\n",
           cudaGetErrorString(err), __FILE__, __LINE__);
    exit(EXIT_FAILURE);
}
```

# Modèle d'exécution CUDA

Application s'exécutant sur une machine hétérogène: hôte (CPU) + device (GPU)

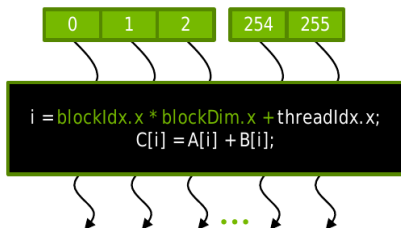
- Les parties séquentielles s'exécutent sur l'hôte
- Les parties parallèles s'exécutent sur le device



# Tableaux de threads parallèle

Un noyau CUDA est exécuté par une grille (tableau) de threads

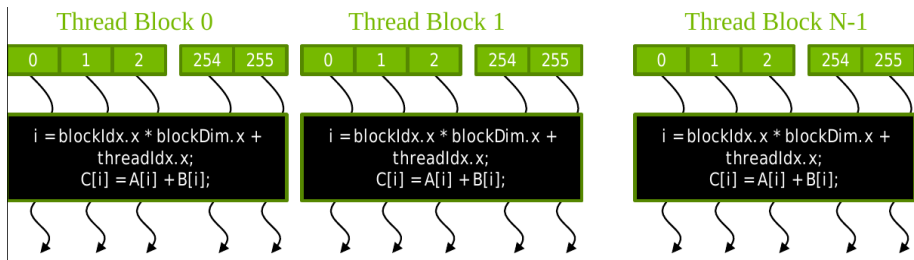
- Tous les threads d'une grille exécute le même noyau (SPMD : Single Program Multiple Data)
- Chaque thread a des indexes qui sont utilisé pour calculer des adresses mémoires et prendre des décisions de contrôle



# Bloc de threads : Coopération

Diviser le tableau de threads en plusieurs blocs

- Les threads au sein d'un bloc coopèrent via la mémoire partagée, les opérations atomiques et les barrières de synchronisation
- Les threads de différents blocs ne peuvent pas interagir



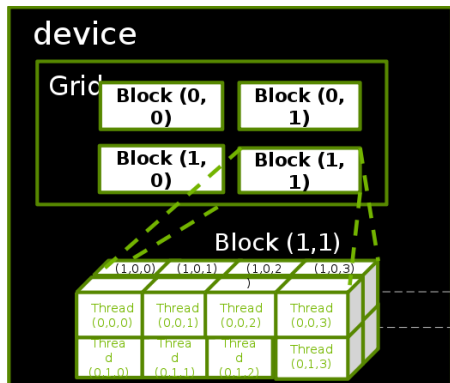
## blockIdx et threadIdx

Chaque thread utilise des indexes pour décider quelle donnée sur laquelle travailler

- blockIdx: 1D, 2D, or 3D (CUDA 4.0)
- threadIdx: 1D, 2D, or 3D

Cela simplifie l'adressage de la mémoire quand des données multidimensionnelles sont traitées

- Image processing
- Résoudre des PDEs sur des volumes
- ...



# Compilateur spécifique NVCC

- NVIDIA fournit un compilateur spécifique CUDA-C/C++
- NVCC compile le code device (GPU) et transmet le code hôte (CPU) au compilateur de l'hôte (gcc/clang/...)
- NVCC se base sur LLVM

# Exemple de code CUDA 1 : Hello World

```
--global-- void mykernel(void) {  
}
```

```
int main(void) {  
    mykernel<<<1,1>>>();  
    printf(" Hello World!\n" );  
    return 0;  
}
```

```
$ nvcc main.cu
```

```
$ ./a.out
```

```
Hello World!
```



# Les outils de debugging

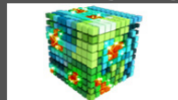
NSIGHT



CUDA-GDB



CUDA MEMCHECK



NVIDIA Provided

allinea  
DDT

TotalView®

3<sup>rd</sup> Party

<http://on-demand.gputechconf.com/gtc/2014/presentations/S4578-cuda-debugging-command-line-tools.pdf>

# printf

- Même format que le printf C
- Sortie bufferisé : Flush seulement aux points de synchronisation explicite
- Sortie non ordonnée : Comme pour un programme multi-threadé
- Possible de changer la taille du buffer :  
`cudaDeviceSetLimit(cudaLimitPrintFifoSize, size_t size);`

# Flags de compilation

- Attention, il y a 2 compilateurs d'utiliser
  - NVCC: code du device
  - Compilateur de l'hôte : code C/C++
- NVCC supporte certains des flags de compilation de l'hôte
  - Mais si le flag n'est pas supporté, il est possible de le transférer au compilateur de l'hôte via `-Xcompiler`
  - Exemple : `-Xcompiler -fopenmp`
- Flags de debugging
  - `-g`: Inclut les symboles de debugging pour le code hôte
  - `-G`: Inclut les symboles de debugging pour le code device (désactive les optimisations)
  - `-lineinfo`: Inclut les informations sur la ligne (dans le code) dans les symboles (pas d'impact sur l'optimisation)

# cuda-memcheck : valgrind memory tools pour CUDA

- Outils de débugging de la mémoire
  - Ne nécessite pas de recompilation : `cuda-memcheck ./exe`
- Peut détecter les erreurs suivantes
  - Fuites mémoires
  - Erreurs de mémoire (Out Of Bounds, misaligned access, illegal instruction, etc)
  - Situation de concurrence
  - Barrières illégales
  - Mémoire non initialisée
- Pour activer l'affichage de la ligne où se trouve l'erreur :
  - `-Xcompiler -rdynamic -lineinfo`

## cuda-memcheck: Exemple 1

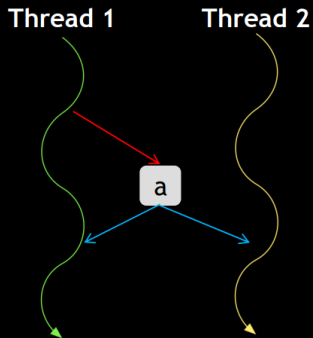
Par défaut, détection les accès mémoires mal alignés ou out of bounds

```
Invalid __global__ read of size 4
  at 0x000000b8 in basic.cu:27:kernel2
  by thread (5,0,0) in block (3,0,0)
  Address 0x05500015 is misaligned
```

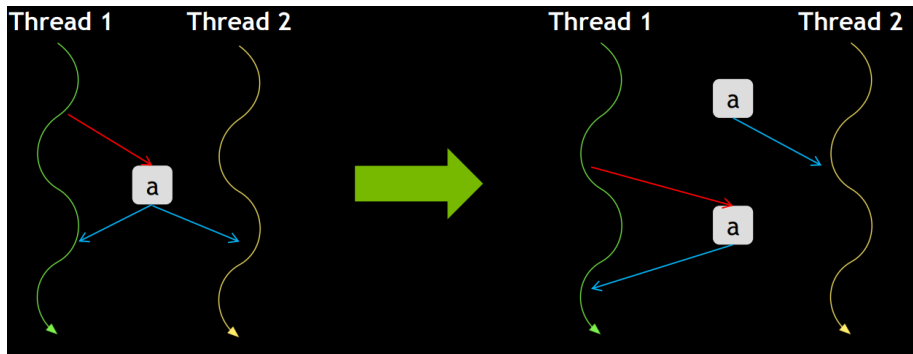
```
Malloc/Free error encountered : Double free
  at 0x0002de18
  by thread (1,0,0) in block (0,0,0)
  Address 0x50c8b99a0
```

# cuda-memcheck: Exemple de situation de concurrence

```
__global__ int bcast(void) {  
    int x;  
    __shared__ int a;  
    if (threadIdx.x == WRITER)  
        a = threadIdx.x;  
    x = a;  
    // do some work  
}
```



## cuda-memcheck: Partage de données entre 2 threads



- Le thread 2 peut lire avant ou après modification la variable *a*
- Nécessite un ordre
- `cuda-memcheck --tool racecheck --racecheck-report analysis`

## cuda gdb : gdb pour CUDA

- cuda-gdb est une extension pour CUDA à GDB
  - Permet de déboguer en même temps le code CUDA et CPU
- Fonctionne sous Linux et Mac OSX
- Pour Windows, il faut utiliser NSIGHT Visual Studio Edition

```
(cuda-gdb) b main //set break point at main
(cuda-gdb) r //run application
(cuda-gdb) l //print line context
(cuda-gdb) b foo //break at kernel foo
(cuda-gdb) c //continue
(cuda-gdb) cuda thread //print current thread
(cuda-gdb) cuda thread 10 //switch to thread 10
(cuda-gdb) cuda block //print current block
(cuda-gdb) cuda block 1 //switch to block 1
(cuda-gdb) d //delete all break points
(cuda-gdb) set cuda memcheck on //turn on cuda memcheck
(cuda-gdb) r //run from the beginning
```

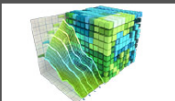


# Les outils de profiling

## NSIGHT



## NVVP



## NVPROF

```

--25614 Profiling results:
+-----+-----+-----+-----+-----+-----+
+ Host | Device | Timing | generate | Function | details | P111 |
+-----+-----+-----+-----+-----+-----+
19.80s  866.40ms  100FLOP  1.7270s  1.568ms  2.816ms  world sh
17.82s  756.40ms  200  1.4000s  1.200ms  1.7530s  kerCmp
19.23s  446.85ms  15000  1.7440s  1.536ms  2.208ms  world sh
+-----+-----+-----+-----+-----+-----+
1. 0.000s  1.000s  1.0000s  generate_function | details | P111 | P1
17.82s  756.40ms  200  1.4000s  1.200ms  1.7530s  kerCmp
2. 0.00s  11.819ms  200  219.09ms  240.33us  266.83ms  kerMain
1.150s  58.577ms  101  68.263ms  920ns  17.97ms  CGEMV
0.937s  14.146ms  200  88.881ms  71.888ms  64.711ms  kerCall
0.279s  12.816ms  400  81.080ms  141.20ms  54.030ms  CGEMV
0.440s  12.872ms  200  66.374ms  91.080ms  61.080ms  kerMain
0.033s  10.973ms  200  36.303ms  31.080ms  36.108ms  kerMain
0.265s  1.1242ms  400  17.293ms  22.320ms  31.320ms  CGEMV
0.123s  2.1342ms  1  2.1342ms  2.1342ms  2.1342ms  world sh
  
```

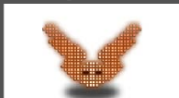
NVIDIA Provided

## TAU



Tuning and Analysis Utilities

## VampirTrace



3<sup>rd</sup> Party

# nvprof

## Un profileur en ligne de commande

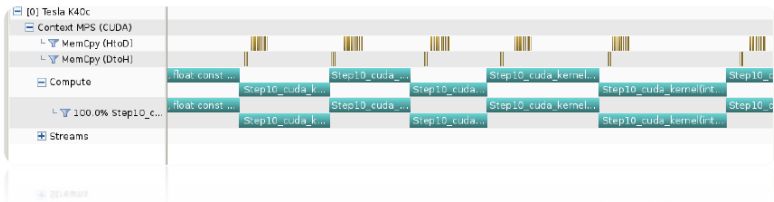
- Calcul le temps d'exécution de chaque noyau
- Calcul le temps de transfert depuis/vers la mémoire device
- Collecte des métriques et des événements
- Supporte des hiérarchies de processus complex
- Collecte les profiles nécessaires pour NVIDIA Visual Profiler
- Ne nécessite pas de recompilation

## Exemple d'utilisation de nvprof

- Information de base : `nvprof`
- Lister les métriques disponible : `nvprof --query-metrics`
- Efficacité des accès aux données :  
`nvprof --metrics gld_efficiency , gst_efficiency`
- Créer une timeline qui pourra être charger par NVVP :  
`nvprof -o profile . timeline`
- Créer une trace des métriques d'analyse pour lire dans NVVP :  
`nvprof -o profile . metrics --analysis-metrics`

# NVIDIA Visual Profiler (NVVP)

## Timeline



## Guided System

### 1. CUDA Application Analysis

### 2. Performance-Critical Kernels

### 3. Compute, Bandwidth, or Latency Bound

The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results at right indicate that the performance of kernel "Step10\_cuda\_kernel" is most likely limited by compute.

#### Perform Compute Analysis

The next likely bottleneck to performance for this kernel is compute so you should first perform compute analysis to determine how it is limiting performance.

#### Perform Latency Analysis

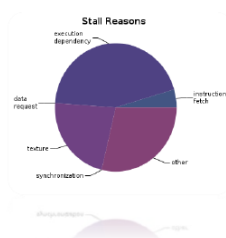
#### Perform Memory Bandwidth Analysis

Instruction and memory latency and memory bandwidth are likely not the primary performance bottleneck for this kernel, but you may still want to perform those analyses.

#### Repeat Analysis

If you modify the kernel you need to rerun your application to update this analysis.

## Analysis

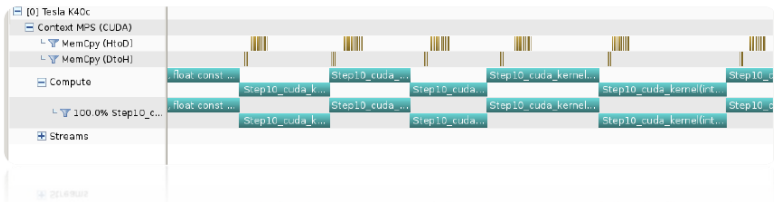


# NVTX

- Permet d'augmenter les traces d'exécution
- La bibliothèque NVTX permet d'ajouter des annotations
  - Il faut ajouter : `#include <nvToolsExt.h>`
  - Et ajouter le lien à la compilation : `-lnvToolsExt`
- Pour marquer le début d'une annotation :  
`nvtxRangePushA( description );`
- Pour marquer la fin d'une annotation : `nvtxRangePop();`
- Possibilité de faire chevaucher les annotations

# NVTX Profile

## Timeline



## Guided System

### 1. CUDA Application Analysis

### 2. Performance-Critical Kernels

### 3. Compute, Bandwidth, or Latency Bound

The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results at right indicate that the performance of kernel "Step10\_cuda\_kernel" is most likely limited by compute.

#### Perform Compute Analysis

The most likely bottleneck to performance for this kernel is compute so you should first perform compute analysis to determine how it is limiting performance.

#### Perform Latency Analysis

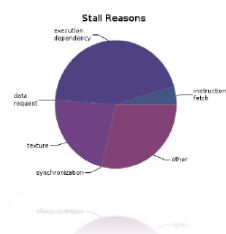
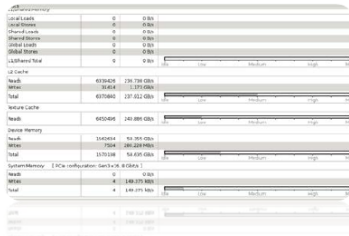
#### Perform Memory Bandwidth Analysis

Instruction and memory latency, and memory bandwidth are likely not the primary performance bottleneck for this kernel, but you may still want to perform those analyses.

#### Repeat Analysis

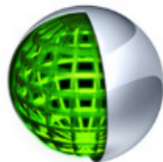
If you modify the kernel you need to rerun your application to update this analysis.

## Analysis

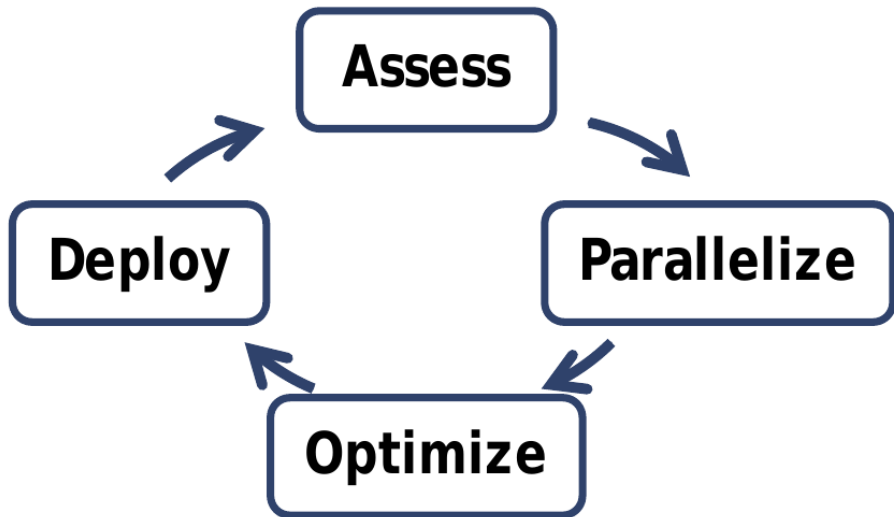


# NSIGHT

- IDE spécifique pour CUDA
  - Edition de code source : colorisation syntaxique, code refactoring, etc.
  - Chaîne de compilation
  - Visual Debugger
  - Visual Profiler
- Linux/Macintosh
  - L'éditeur est Eclipse
  - Le débogueur est cuda-gdb avec une interface graphique
  - Le profileur est NVVP
- Windows
  - S'intègre directement dans Visual Studio
  - Le profileur est NSIGHT VSE

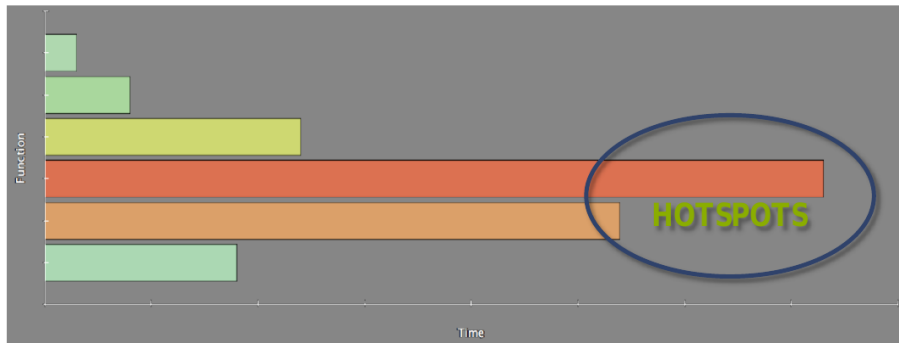


# CUDA Optimization





# CUDA Optimization: Evaluation



- Profile votre code et trouver le(s) point(s) chaud
- Se concentrer sur les points qui auront le plus de bénéfice

# CUDA Optimization: Parallélisation

Applications

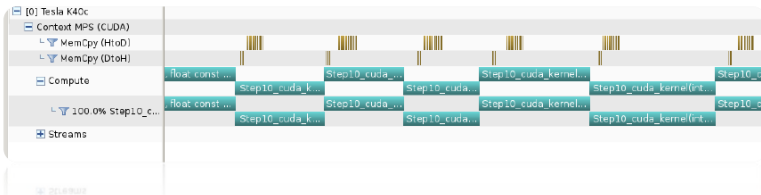
Libraries

Compiler  
Directives

Programming  
Languages

# CUDA Optimization: Optimization

## Timeline



## Guided System

### 1. CUDA Application Analysis

### 2. Performance-Critical Kernels

### 3. Compute, Bandwidth, or Latency Bound

The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results at right indicate that the performance of kernel "Step10\_cuda\_kernel" is most likely limited by compute.

#### Perform Compute Analysis

The most likely bottleneck to performance for this kernel is compute as you should first perform compute analysis to determine how it is limiting performance.

#### Perform Latency Analysis

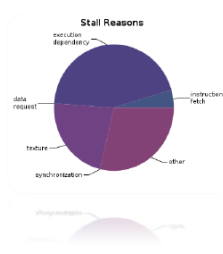
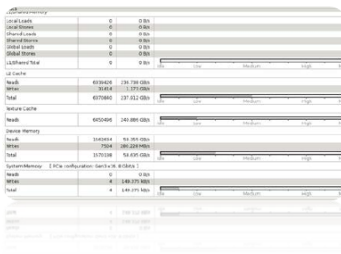
Instruction and memory latency and memory bandwidth are likely not the primary performance bottleneck for this kernel, but you may still want to perform these analyses.

#### Run Analysis

If you modify the kernel you need to rerun your application to update this analysis.

© NVIDIA Corporation

## Analysis



# CUDA Optimization: Analyse des goulots d'étranglement

- Don't assume an optimization was wrong
- Verify if it was wrong with the profiler

129 GB/s ➔ 84 GB/s

## LI/Shared Memory

Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	2097152	3,351,979 GB/s	
Shared Stores	131072	84,459 GB/s	
Global Loads	131072	42,249 GB/s	
Global Stores	131072	42,249 GB/s	
Atomic	0	0 B/s	
<b>LI/Shared Total</b>	<b>2490368</b>	<b>3,520,977 GB/s</b>	

gpuTranspose_kernel(int, int, float const *, float*)	
Start	547.303 ms (
End	547.716 ms (
Duration	413.872 $\mu$ s
Grid Size	[ 64,64,1 ]
Block Size	[ 32,32,1 ]
Registers/Thread	10
Shared Memory/Block	4 KiB
▼ Efficiency	
Global Load Efficiency	100%
Global Store Efficiency	100%
Shared Efficiency	▲ 5.9%
Warp Execution efficiency	100%
Non-Predicated Warp Execution Efficiency	97.1%
▼ Occupancy	
Achieved	86.7%
Theoretical	100%
▼ Shared Memory Configuration	
Shared Memory Requested	48 KiB
Shared Memory Executed	48 KiB

## 🔔 Shared Memory Alignment and Access Pattern

Memory bandwidth is used most efficiently when each shared memory load and store has proper alignment and access pattern.

*Optimization:* Select each entry below to open the source code to a shared load or store within the kernel with an inefficient alignment or access pattern. For each access pattern of the memory access.

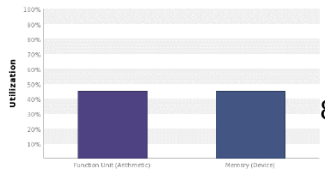
▼ Line / File | main.cu - /home/jlutjens/code/CudaHandsOn/Example19

49 | Shared Load Transactions/Access = 16, Ideal Transactions/Access = 1 [ 2097152 transactions for 131072 total executions ]

# CUDA Optimization: Analyse des performances

## gpuTranspose\_kernel(int, int, float const \*, float

Start	770.067
End	770.324
Duration	256.714
Grid Size	[ 64,64,1
Block Size	[ 32,32,1
Registers/Thread	10
Shared Memory/Block	4.125 KiB
Efficiency	
Global Load Efficiency	100%
Global Store Efficiency	100%
Shared Efficiency	50%
Warp Execution Efficiency	100%
Non-Predicated Warp Execution Efficiency	97.1%
Occupancy	
Achieved	87.7%
Theoretical	100%
Shared Memory Configuration	
Shared Memory Requested	48 KiB
Shared Memory Executed	48 KiB



84 GB/s → 137 GB/s

L1 Shared Memory		
Local Loads	0	0 B/s
Local Stores	0	0 B/s
Shared Loads	131072	138.433 GB/s
Shared Stores	131072	138.118 GB/s
Global Loads	131072	69.217 GB/s
Global Stores	131072	69.217 GB/s
Atomic	0	0 B/s
L1 Shared Total	524960	415.984 GB/s
L2 Cache		
L1 Reads	524288	69.217 GB/s
L1 Writes	524288	69.217 GB/s
Texture Reads	0	0 B/s
Atomic	0	0 B/s
Noncoherent Reads	0	0 B/s
Total	1048576	138.433 GB/s
Texture Cache		
Reads	0	0 B/s
Device Memory		
Reads	524960	69.306 GB/s
Writes	524288	69.217 GB/s
Total	1049257	138.523 GB/s

## Exemple : Code device de l'addition de vecteur

- Calcul la somme de 2 vecteurs  $C = A + B$
- Chaque thread effectue une addition de pair

```
__global__  
void vecAddKernel(float* A, float* B, float* C, int n)  
{  
    int i = threadIdx.x+blockDim.x*blockIdx.x;  
    if(i<n) C[i] = A[i] + B[i];  
}
```

## Exemple : Code hôte de l'addition de vecteur

- La fonction *ceil* permet d'être sûr qu'il a assez de threads pour toutes les éléments des vecteurs

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    // d_A, d_B, d_C allocations and copies omitted
    // Run ceil(n/256.0) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0),256>>>(d_A, d_B, d_C, n);
}
```

## Exemple : Code hôte de l'addition de vecteur

- Une autre méthode pour appeler le noyau

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    dim3 DimGrid((n-1)/256 + 1, 1, 1);
    dim3 DimBlock(256, 1, 1);
    vecAddKernel<<<DimGrid, DimBlock>>>(d_A, d_B, d_C, n);
}
```



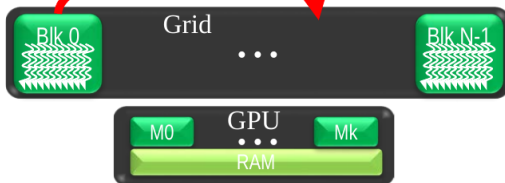
# Execution du noyau en bref

```
__host__
void vecAdd(...)
```

```
{
  dim3 DimGrid(ceil(n/256.0),1,1);
  dim3 DimBlock(256,1,1);
  vecAddKernel<<<DimGrid,DimBlock>>>(d_A,d_B
,d_C,n);
}
```

```
__global__
```

```
void vecAddKernel(float *A,
  float *B, float *C, int n)
{
  int i = blockIdx.x * blockDim.x
    + threadIdx.x;
  if( i<n ) C[i] = A[i]+B[i];
}
```

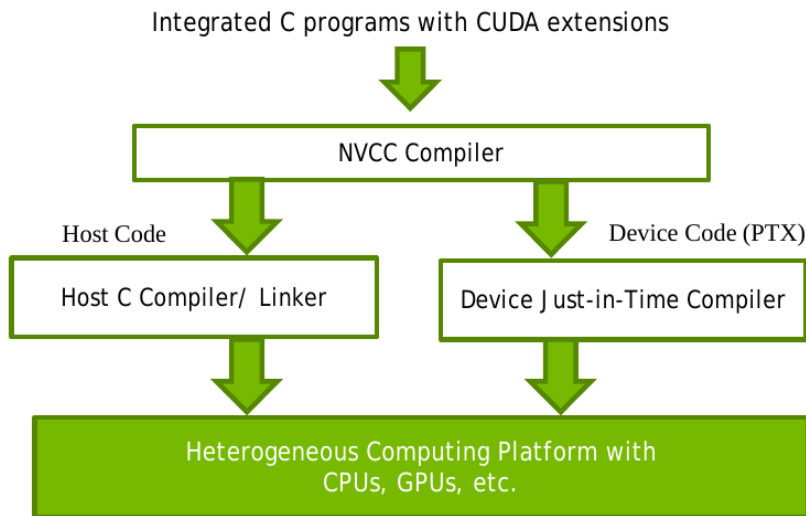


# Déclarations de fonctions CUDA

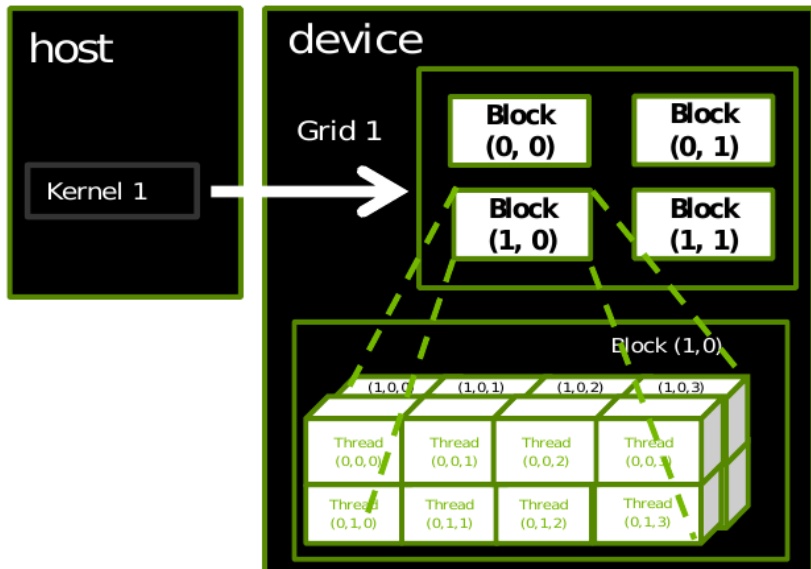
	s'exécute sur	s'appelle depuis
<code>__device__ float DeviceFunc()</code>	device	device
<code>__global__ void KernelFunc()</code>	device	host
<code>__host__ float HostFunc()</code>	host	host

- `__global__` définit un noyau, doit forcément retourner void
- `__device__` et `__host__` peuvent être utilisés ensemble
- `__host__` est optionnel si utilisé seul

# Chaîne de compilation CUDA



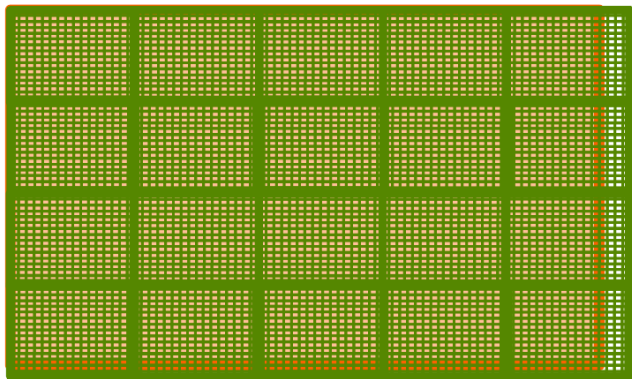
# Exemple d'une grille multi-dimension



# Parcours d'une image avec une grille 2D

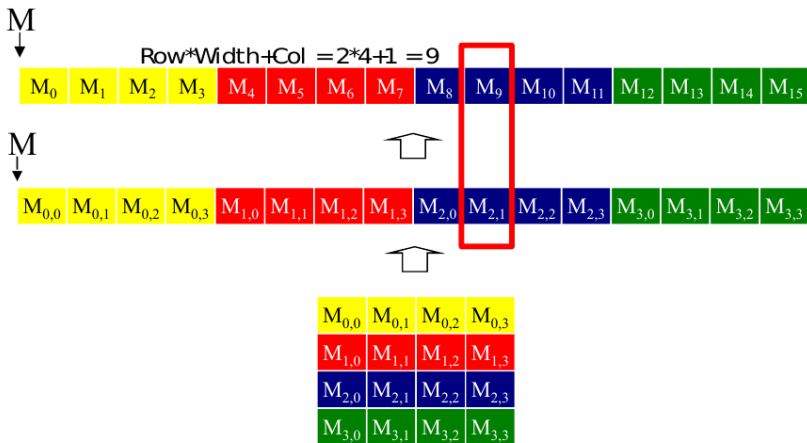


16x16 blocks



62x76 picture

# D'une image à un indice



## Exemple d'un noyau multi-dimension

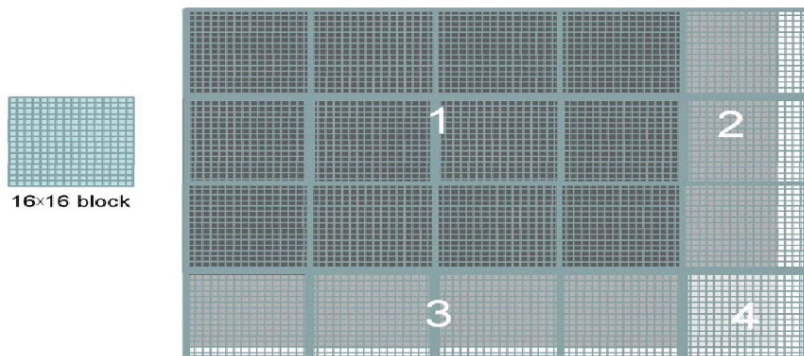
```
--global-- void PictureKernel(float* d_Pin, float* d_Pout,
                             int height, int width)
{
    // Calculate the row # of the d_Pin and d_Pout element
    int Row = blockIdx.y*blockDim.y + threadIdx.y;
    // Calculate the column # of the d_Pin and d_Pout element
    int Col = blockIdx.x*blockDim.x + threadIdx.x;
    // each thread computes one element of d_Pout if in range
    if ((Row < height) && (Col < width)) {
        d_Pout[Row*width+Col] = 2.0*d_Pin[Row*width+Col];
    }
}
```

## Code hôte d'un noyau multi-dimension

```
// assume that the picture is m n,  
// m pixels in y dimension and n pixels in x dimension  
// input d_Pin has been allocated on and copied to device  
// output d_Pout has been allocated on device  
...  
dim3 DimGrid((n-1)/16 + 1, (m-1)/16+1, 1);  
dim3 DimBlock(16, 16, 1);  
PictureKernel<<<DimGrid, DimBlock>>>(d_Pin, d_Pout, m, n);  
...
```

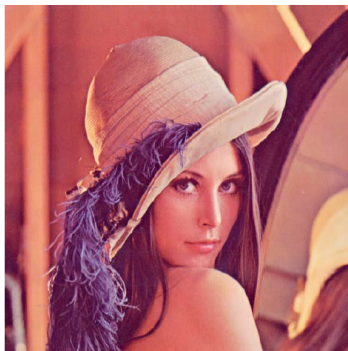


# Parcourir une image 62x76 avec des blocs 16x16



Tous les threads d'un bloc ne suivront pas le même chemin dans le flux de contrôle

# Conversion d'une image RGB vers des niveaux de gris

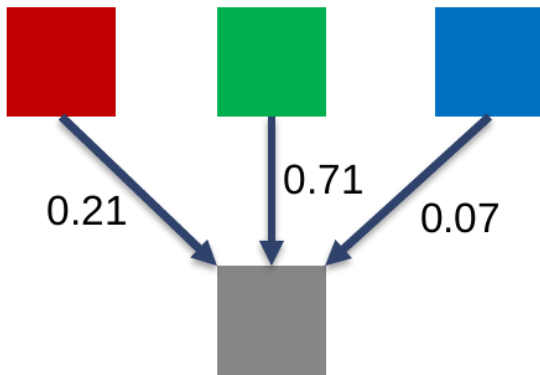


Une image en niveaux de gris est une image qui contient seulement l'intensité d'une valeur pour chaque pixel

## Formule de conversion

Pour chaque pixel  $(r,g,b)$  à l'index  $(i,j)$  :

$$\text{grayPixel}[i,j] = 0.21 * r + 0.71 * g + 0.07 * b$$



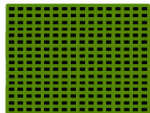
## Code de la conversion

```
#define CHANNELS 3 // we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__ void colorConvert(unsigned char * grayImage,
    unsigned char * rgbImage, int width, int height) {
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    if (x < width && y < height) {
        // get 1D coordinate for the grayscale image
        int grayOffset = y*width + x;
        // one can think of the RGB image having
        // CHANNEL times columns than the gray scale image
        int rgbOffset = grayOffset*CHANNELS;
        unsigned char r = rgbImage[rgbOffset]; // red value for pixel
        unsigned char g = rgbImage[rgbOffset + 2]; // green value for pixel
        unsigned char b = rgbImage[rgbOffset + 3]; // blue value for pixel
        // perform the rescaling and store it
        // We multiply by floating point constants
        grayImage[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
    }
}
```

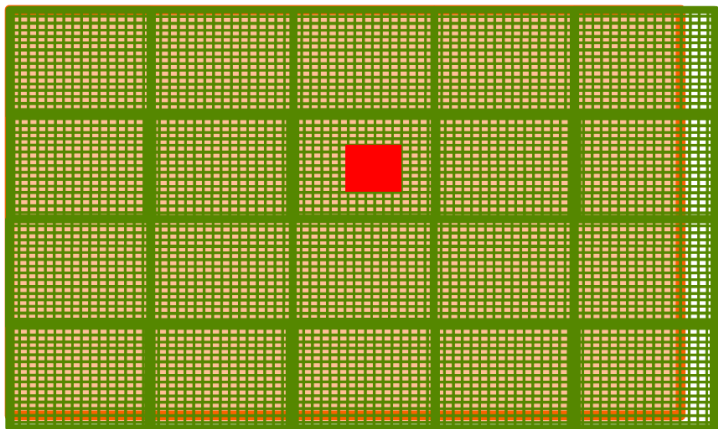
# Floutage d'image



# Boîte de floutage



Pixels  
processed  
by a thread  
block



# Code de floutage

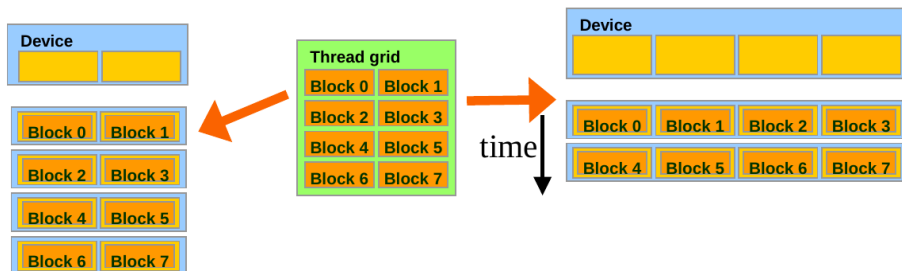
```
--global--  
void blurKernel(unsigned char * in, unsigned char * out, int w,  
{  
    int Col = blockIdx.x * blockDim.x + threadIdx.x;  
    int Row = blockIdx.y * blockDim.y + threadIdx.y;  
    if (Col < w && Row < h) {  
        ... // Rest of our kernel  
    }  
}
```

## Code de floutage

```
--global--
void blurKernel(unsigned char * in, unsigned char * out,
               int w, int h) {
    int Col = blockIdx.x * blockDim.x + threadIdx.x;
    int Row = blockIdx.y * blockDim.y + threadIdx.y;
    if (Col < w && Row < h) {
        int pixVal = 0;
        int pixels = 0;
        for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
            for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol)
                int curRow = Row + blurRow;
                int curCol = Col + blurCol;
                if (curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
                    pixVal += in[curRow * w + curCol];
                    pixels++;
                }
            }
        }
        out[Row * w + Col] = (unsigned char)(pixVal / pixels);
    }
}
```



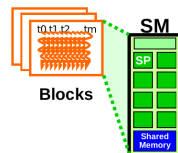
# Passage à l'échelle transparent



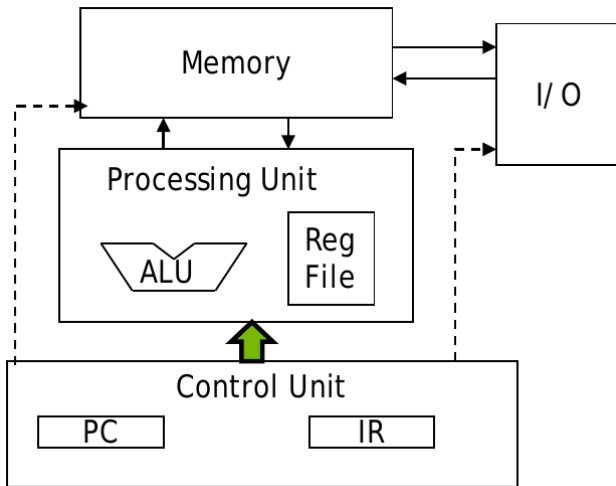
- Les blocs peuvent être exécutés dans n'importe quel ordre
- Le matériel est libre d'assigner un bloc à n'importe lequel des unités de calcul

# Exécuter des blocs de threads

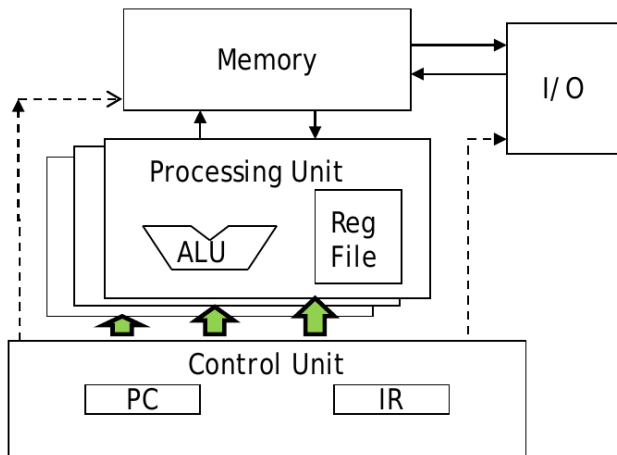
- Les threads sont assignés à un Streaming Multiprocessors (SM) à la granularité d'un bloc
  - Jusqu'à 8 blocs peuvent être alloués par SM
  - Fermi SM peuvent gérer jusqu'à 1,536 threads (nb threads par warp x Max warp par SM), 2048 à partir de Kepler
    - Par exemple  $256 \text{ (threads/bloc)} * 6 \text{ blocs}$
    - Ou  $512 \text{ (threads/bloc)} * 3 \text{ blocs}$ , etc.
  - SM est en charge de gérer les index de threads/blocs
  - SM ordonnance et gère l'exécution des threads



# Modèle Von-Neumann Model



# Modèle Von-Neumann Model avec SIMD



Single Instruction Multiple Data  
(SIMD)

# Warp : l'unité d'ordonnancement

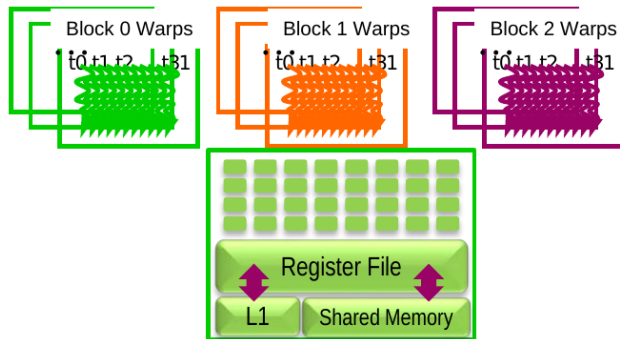
Chaque bloc est exécuté par des warps de 32 threads

- C'est une décision d'implémentation et ne fait pas du tout parti du modèle de programmation CUDA
- Les warps sont des unités d'ordonnancement dans les SMs
- Les threads au sein d'un warp sont exécuté sous la forme de SIMD
- Le nombre de threads au sein d'un warp peut changer suivant les générations (toujours 32 pour l'instant)

## Les warps en pratique

Si 3 blocs sont assignés à un SM et que chaque bloc a 256 threads, combien de warps y a-t-il dans un SM ?

- Chaque bloc est découpé en  $256/32 = 8$  warps
- Il y a donc  $8 * 3 = 24$  warps



# Ordonnancement de threads

Les SMs implémentent un ordonnancement à coût nul pour les warps

- Les warps dont la prochaine instruction et les variables sont disponibles est éligible pour l'exécution
- Les warps éligibles sont sélectionnés pour l'exécution en utilisant une politique d'ordonnancement avec priorité
- Tous les threads dans un warp exécutent la même instruction quand sélectionnés

## Quelques réflexions sur la granularité des blocs

Pour une multiplication de matrice en utilisant plusieurs blocs, dois-je choisir des blocs de  $8 \times 8$ ,  $16 \times 16$  ou  $32 \times 32$  ?

- Pour  $8 \times 8$ , nous avons 64 threads par bloc. Puisque chaque SM peut s'occuper de jusqu'à 1,536 threads ce qui se traduit par 24 blocs. Mais puisque chaque SM ne peut prendre que jusqu'à 8 blocs, il n'y aura que 512 threads par SM.
- Pour  $16 \times 16$ , nous avons 256 threads par bloc. Puisque chaque SM peut prendre jusqu'à 1,536 threads, il peut se charger de 6 blocs et donc atteindre sa capacité maximal.
- Pour  $32 \times 32$ , nous avons 1024 threads par bloc. Seulement un bloc peut tenir dans un SM. Il y a donc uniquement les  $2/3$  des capacités d'un SM qui sont utilisées.