INF 560
Calcul Parallèle et Distribué
Cours 3

Eric Goubault

CEA, LIST & Ecole Polytechnique

28 janvier 2013

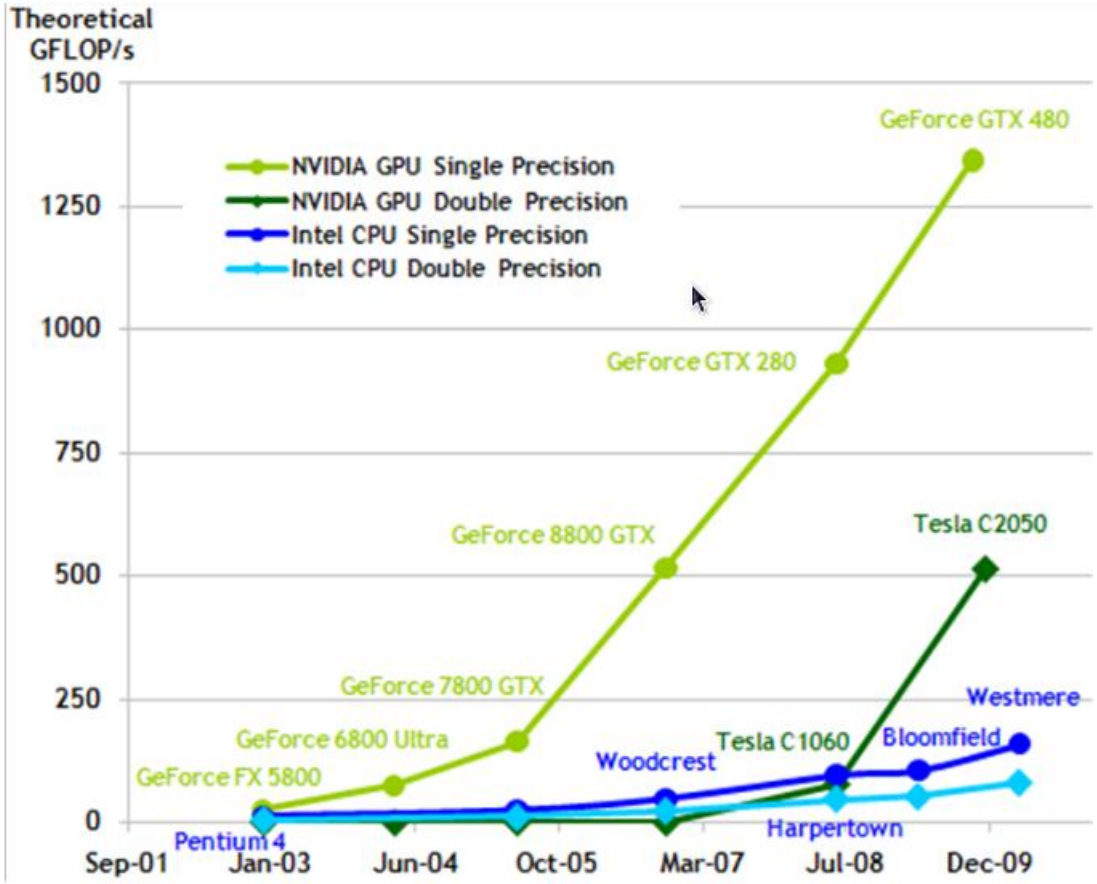
Plan du cours

- CUDA et architecture NVIDIA
- L'abstraction logique de l'architecture proposée par CUDA
- C versus JAVA...
- API CUDA
- Un exemple: addition de matrices
- Revenons à l'architecture...pour optimiser...
- Un exemple: transposition de matrices
- Pour aller plus loin...

ÉCOLE POLYTECHNIQUE

CUDA: un modèle quasi PRAM?

- "Compute Unified Device Architecture"
- Programmation massivement parallèle en C sur cartes NVIDIA, typiquement mémoire partagée
- Tirer parti de la puissance des cartes graphiques:



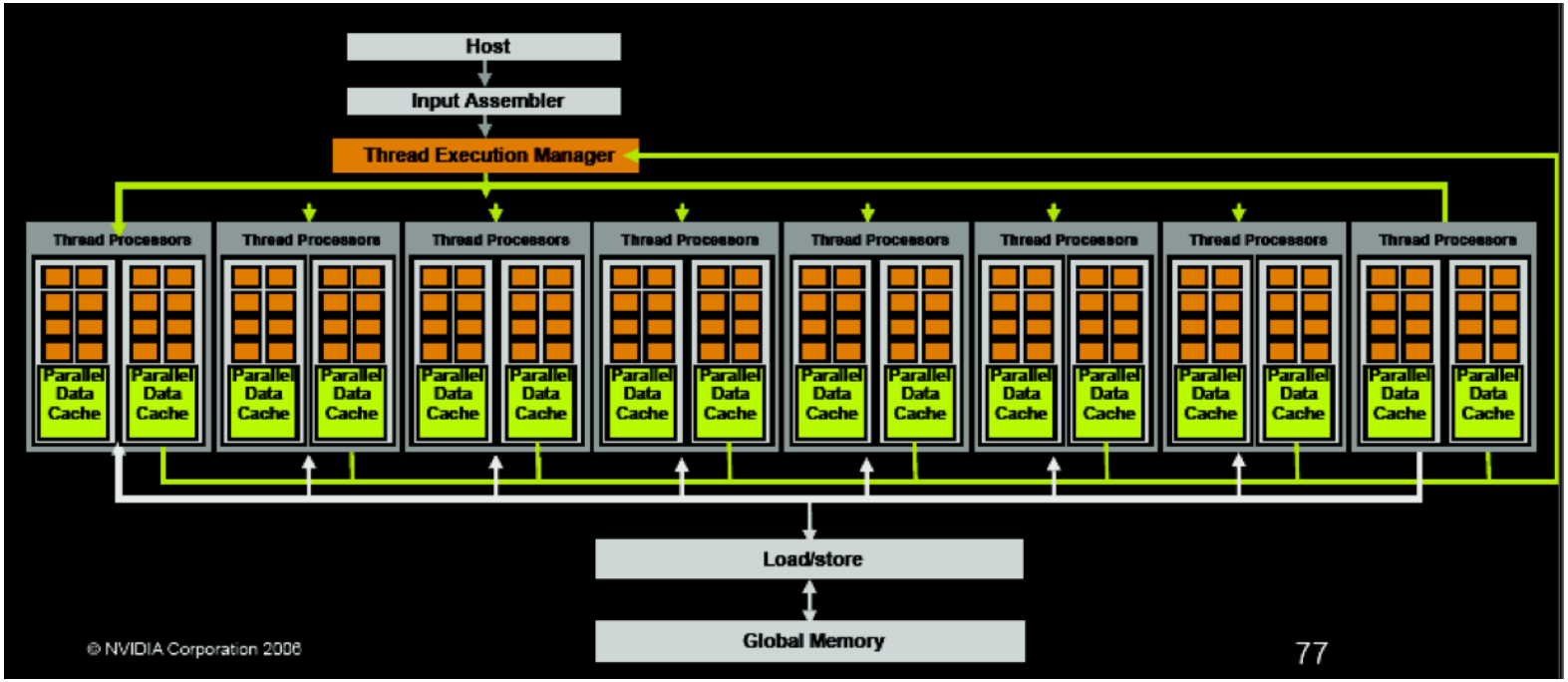
ÉCOLE POLYTECHNIQUE

CUDA: un modèle quasi PRAM?

- Peut être programmé pratiquement comme une PRAM CREW, à la différence près que le coût mémoire(s!) est variable et indispensable à gérer (prochain cours)
- On revient à la technique de saut de pointeur (`scan`, fournie dans la SDK CUDA!); avant cela quelques notions sur CUDA...

ÉCOLE POLYTECHNIQUE

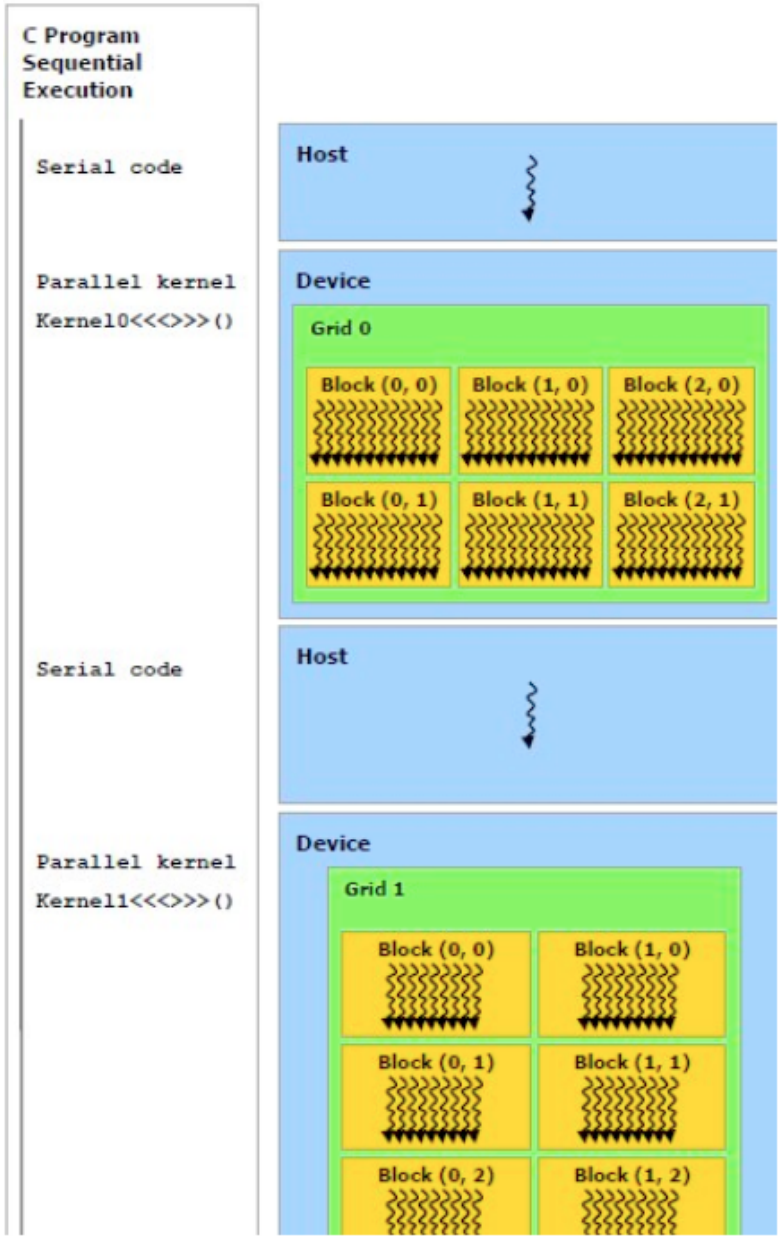
Architecture physique



(ici 128 threads procs. pour 8 multi-procs de 16 coeurs)

- L'hôte peut charger des données sur le GPU, et calculer en parallèle avec le GPU
- Thread processor $\sim$ processus PRAM mais...

Modèle d'exécution: "coprocesseur PRAM"

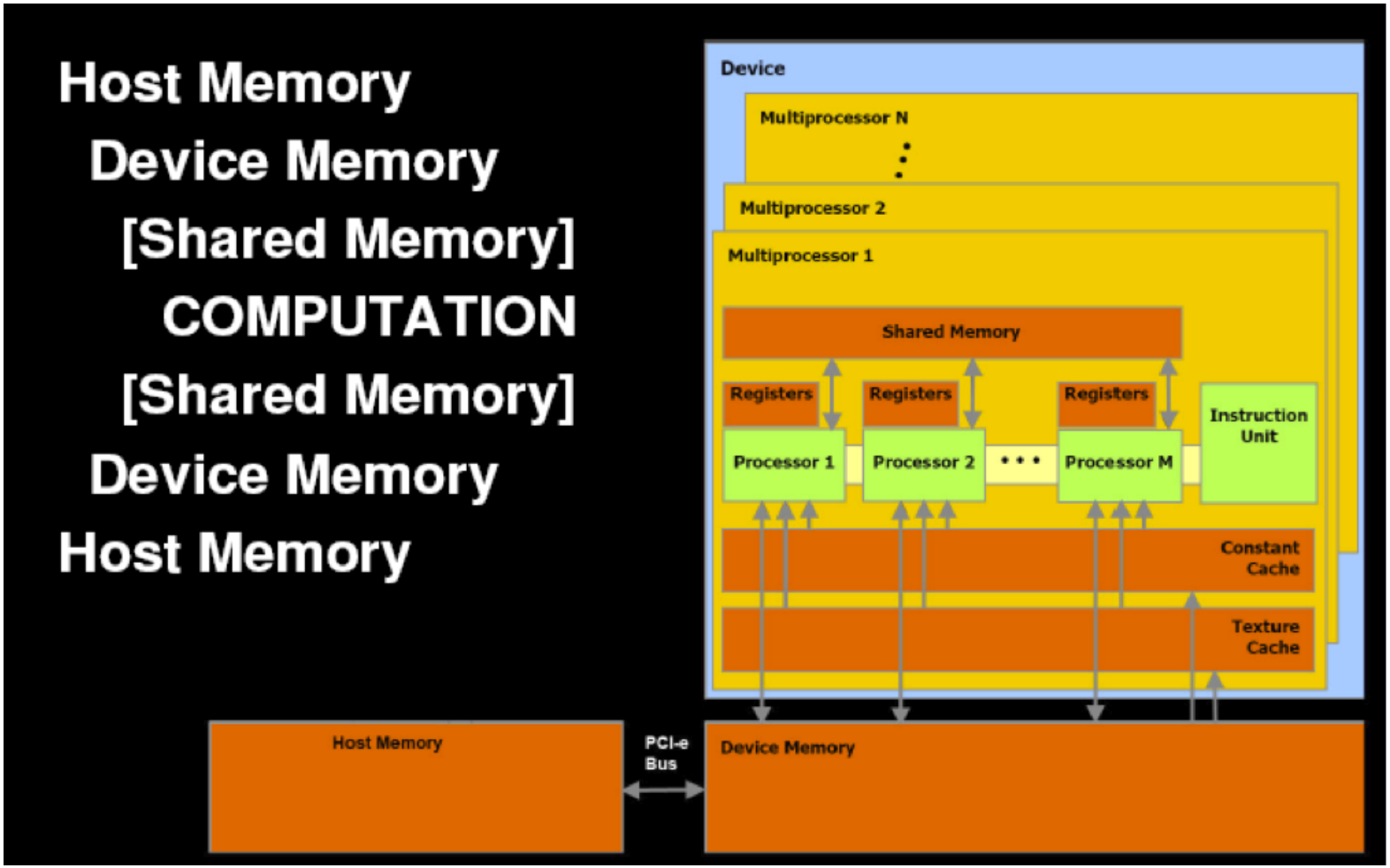


Architecture physique

- Organisés en multiprocesseurs (ex. GeForce GT 430 des salles 32, 36: 2 multiproc de 48 coeurs=96 coeurs, à 1.4GHz)
- registres 32 bits par multi-proc.
- **mémoire partagées rapide uniquement par multi-proc.!**
- une mémoire ("constante") à lecture seule (ainsi qu'un cache de textures à lecture seule)

ÉCOLE POLYTECHNIQUE

Architecture physique



Modèle de programmation - un peu de vocabulaire

- la carte graphique="GPU" ou "device" est utilisé comme "co-processeur" de calcul pour le processeur de la machine hôte, le PC typiquement ou "host" ou "CPU'
  - la mémoire du CPU est distincte de celle du GPU
  - mais on peut faire des recopies de l'un vers l'autre (couteux)
- une fonction calculée sur le device est appelée "kernel" (noyau)
- le kernel est dupliqué sur le GPU comme un ensemble de threads - cet ensemble de threads est organisé de façon logique en une "grid"
- chaque clône du kernel connaît sa position dans la grid et peut calculer la fonction définie par le kernel sur différentes données
- cette grid est mappée physiquement sur l'architecture de la carte au "runtime"

ÉCOLE POLYTECHNIQUE

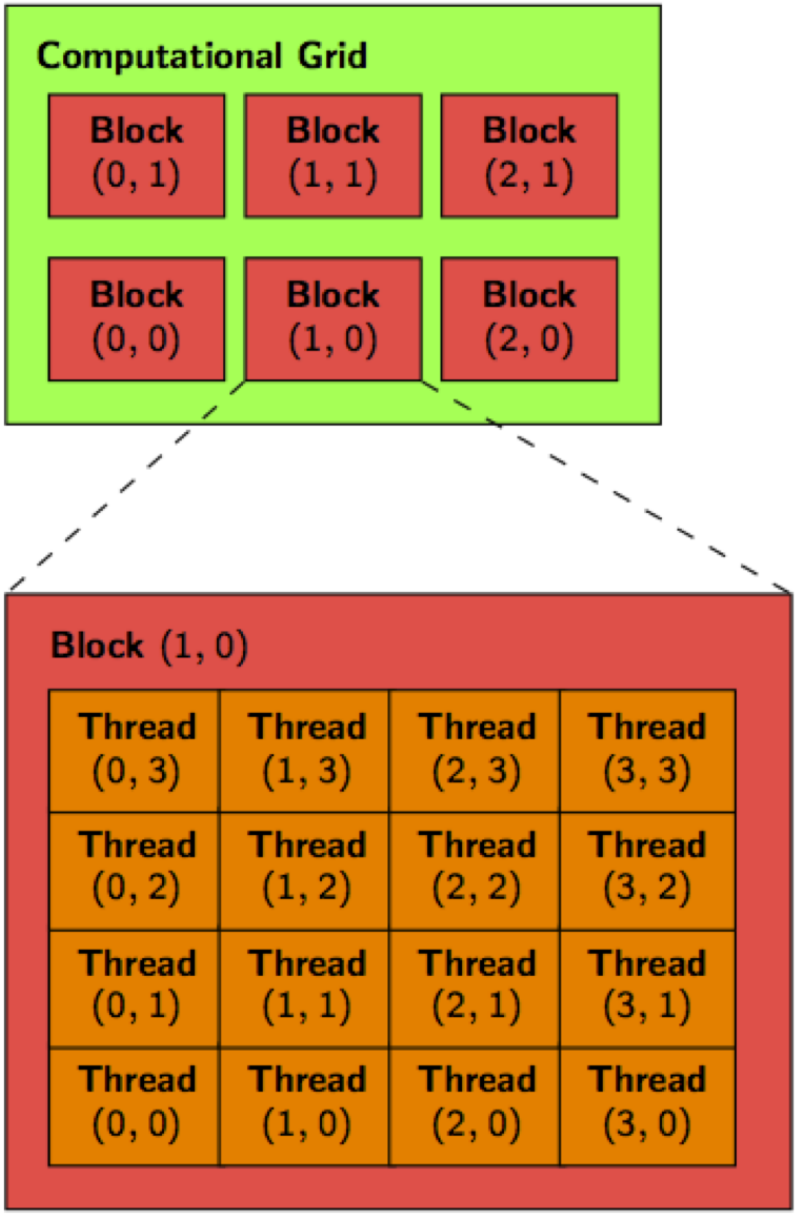
Abstraction logique - grid

- une grid est un tableau 1D, 2D ou 3D de "thread blocks" - au maximum 65536 blocks par dimension (en pratique, 2D...)
- chaque thread block est un tableau 1D, 2D ou 3D de "threads", chacun exécutant un clône (instance) du kernel - au maximum 512 threads par block (en général)
- chaque block a un unique `blockId`
- chaque thread a un unique `threadId` (dans un block donné)

Attention

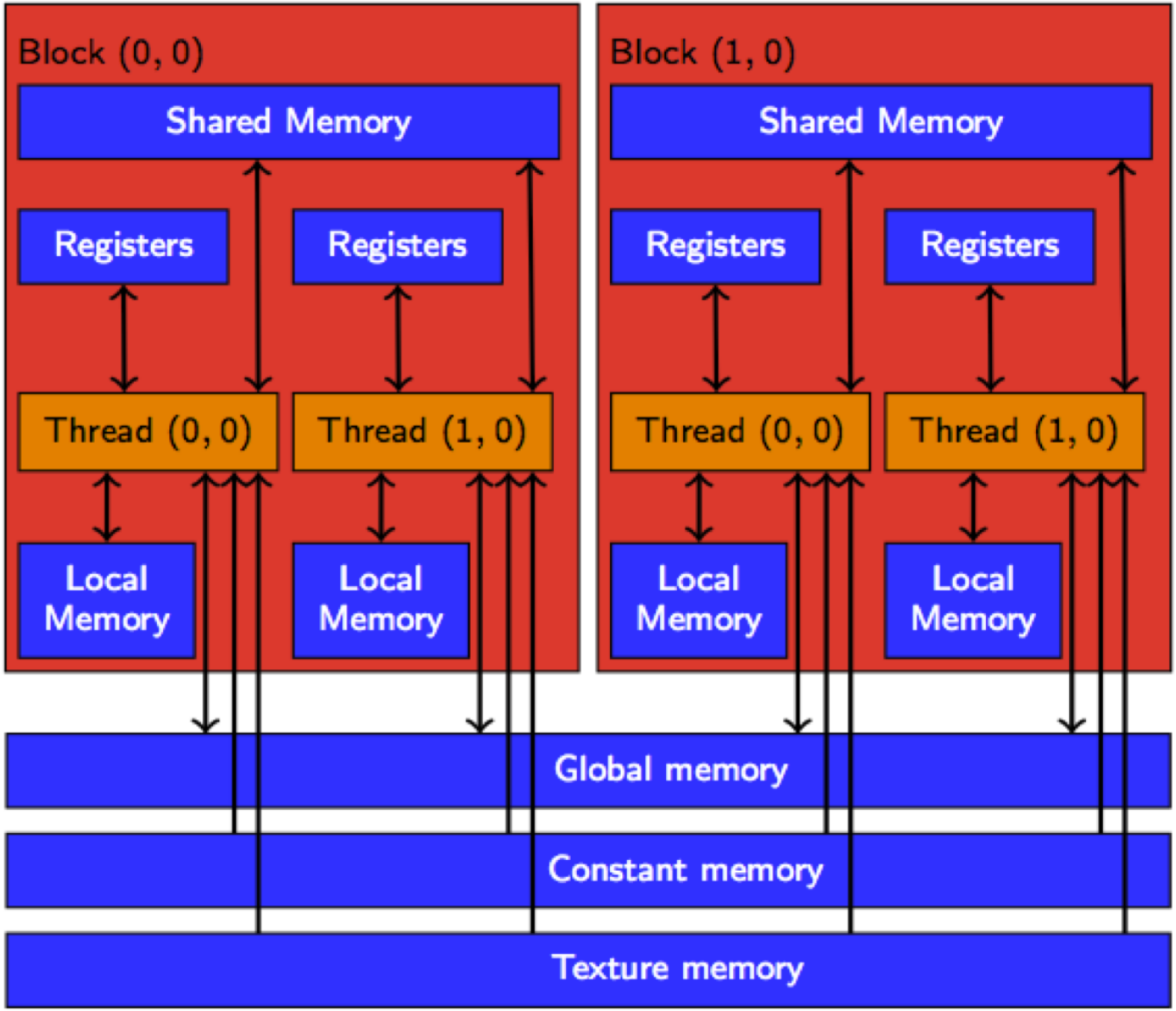
Pour toutes ces limitations (nombre de "blocks" etc.), ceci dépend précisément de l'architecture de la carte, à voir en exécutant `deviceQuery` (à importer depuis la SDK et à compiler depuis `nsight`)

Grid



ÉCOLE POLYTECHNIQUE

Modèle mémoire



Modèle mémoire

Suit la hiérarchie (logique) de la grid:

- Mémoire globale (du device): la plus lente (400 à 600 cycles de latence!), accessible (lecture/écriture) à toute la grid
  - Possibilité d'optimiser cela en "amalgamant" les accès
- Mémoire partagée: rapide mais limitée (16Ko par multiprocesseur), accessible (lecture/écriture) à tout un block - qualificatif `__shared__`

Modèle mémoire

- Registres (16384 par multiprocesseur): rapide mais très limitée, accessible (lecture/écriture) à un thread
- Mémoire locale: lente (200 a 300 cycles!) et limitée, accessible (lecture/écriture) - gérée automatiquement lors de la compilation (quand structures ou tableaux trop gros pour être en registre)

En plus de cela (quasi pas traité ici), mémoire constante et texture: rapide, accessible (en lecture uniquement depuis le GPU, lecture/écriture depuis le CPU) à toute la grid. Mémoire constante très petite ($\sim$8 à 64 K)

API sur le GPU

- Qualificateurs de types de fonctions:
  - `__device__ f(...)`: la fonction `f` uniquement appelable/exécutable sur le GPU (device)
  - `__host__ g(...)`: la fonction `g` uniquement appelable/exécutable sur le CPU (host)
  - `__global__ h(...)`: la fonction `h` exécutable sur le GPU et appelable depuis le CPU (tous les kernels)
- Qualificateurs de types de variables:
  - `__device__ int x;`: `x` est un entier en mémoire globale
  - `__constant__ int x=5;`: `x` est un entier en mémoire constante
  - `__shared__ int x;`: `x` est un entier en mémoire partagée

ÉCOLE POLYTECHNIQUE

API sur le GPU

- Exécution du kernel `f` et mapping logique sur la grid:
  - `f<<<GridDim, BlockDim>>>(...)`
  - où `GridDim` est de type `dim3`
  - où `BlockDim` est de type `dim3`
- Chaque instance du kernel sait où il est exécuté par:
  - `blockIdx` renvoie un `uint3` indiquant dans quel block on est
  - `threadIdx` renvoie un `uint3` indiquant dans quel thread du block on est
- Dans chaque block, tous les threads exécutant le kernel peuvent se synchroniser (se mettre en attente à un point de rendez-vous) en faisant `__syncthreads();`

ÉCOLE POLYTECHNIQUE

Type dim3

- Si `A` est de type `dim3` ou `uint3`
- La différence entre `dim3` et `uint3` est que les composantes non-initialisées d'un `dim3` sont par défaut égales à 1
- Il existe d'autre types vecteurs...
- `A.x`, `A.y` et `A.z` renvoient des `int` donnant les 3 coordonnées
- Il est alors facile de mapper tout tableau n-dimensionnel sur une grille par une simple formule impliquant `blockIdx.x`, `blockIdx.y`, `blockIdx.z` et `threadIdx.x`, `threadIdx.y` et `threadIdx.z`

Premier exemple CUDA: somme de matrices

```
const int N = 1024;
const int blocksize = 16;
const int MAX = 100;

__host__ void add_matrix_cpu(float* a, float *b, float *c, int N) {
  int i, j;
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      c[i*N+j]=a[i*N+j]+b[i*N+j];
}

__global__ void add_matrix(float* a, float *b, float *c, int N ) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if ( i < N && j < N )
    c[index] = a[index] + b[index];
}
```

Premier exemple CUDA: somme de matrices

```
int main() {
  int k;
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  StopWatchInterface *timer = 0;
  sdkCreateTimer(&timer);

  for ( int i = 0; i < N*N; ++i ) {
    a[i] = 1.0f; b[i] = 3.5f; }

  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&bd, size );
  cudaMalloc( (void**)&cd, size );
  cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );
  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );

  sdkStartTimer(&timer);
  for (k=1;k<=MAX;k++) {
    add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );
    cudaThreadSynchronize();
  }
```

Premier exemple CUDA: somme de matrices

```
    sdkStopTimer(&timer);
    printf("Processing time on GPU: %f (ms)\n",sdkGetTimerValue(&timer)/MAX);
    sdkDeleteTimer(&timer);
    cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );
    cudaFree( ad );
    cudaFree( bd );
    cudaFree( cd );

    StopWatchInterface *timer2 = 0;
    sdkCreateTimer(&timer2);
    sdkStartTimer(&timer2);
    for (k=1;k<=MAX;k++)
      add_matrix_cpu(a,b,c, N);
    sdkStopTimer(&timer2);
    printf("Processing time on CPU: %f (ms)\n",sdkGetTimerValue(&timer2)/MAX);
    sdkDeleteTimer(&timer2);
    delete[] a;
    delete[] b;
    delete[] c;
    return EXIT_SUCCESS;
}
```

Un peu de C d'abord...:

```
float *c; /* result */
void add_matrix(float *a, float *b, int N) {
  for (int i=0; i<N; i++)
    for (int j=0; j<N; j++)
      c[i+j*N]=a[i+j*N]+b[i+j*N]; }
int main(int argc, char **argv) {
  float *x, *y;
  int N=16;
  x=(float *) malloc(N*sizeof(float));
  y=(float *) malloc(N*sizeof(float));
  c=(float *) malloc(N*sizeof(float));
  (...)
  add_matrix(a, b, N); }
```

Gestion mémoire (JAVA)

```
public class matrix {
  float c[]; /* result */

  public void add_matrix(...) ...

  public void main(String args) {
    float x[], y[], z[];
    int N=16;
    x=new float[N];
    y=new float[N];
    c=new float[N];
    (...)
    add_matrix(x, y, N);
}
```

Gestion mémoire en C sur le CPU/GPU

- Pointeurs...
- Fonctions d'allocation C `malloc`/`free`; ainsi que `new`/`delete` avec `nvcc`
- tableaux 1D, 2D, 3D... arithmétique des pointeurs en C
- Fonctions spécifiques d'allocation mémoire sous CUDA: `cudaMalloc`, `cudaFree`, fonctions de recopie synchrones (i.e. bloquantes) `cudaMemcpy` mais aussi fonctions sur des tableaux `cudaMallocPitch` (2D), `cudaMalloc3D` (3D) et `cudaMemcpy2D`, `cudaMemcpy3D`

Pointeurs/allocation C

- Notion d'adresse mémoire
- Déclaration `float *x;` "pointeur" sur `x`
- `float y, *x;` puis `x=&y;` (allocation statique)
- `float *x;` puis `x=(float *) malloc(sizeof(float));` (allocation dynamique)

ÉCOLE POLYTECHNIQUE

Tableaux C

- Vecteur: `float x = (float *) malloc(N*sizeof(float));` vecteur à `N` entrées (bloc contigu de `N` mots de 32 bits)
- Matrice: `float *x = (float *) malloc(N*M*sizeof(float));` matrice de dimension `N*M`
  - on récupère $x_{i,j}$ par `x[i*M+j]`
- Autre méthode: `float **x = (float **) malloc(N*sizeof(float *));` puis:

```
for (i=0; i<N; i++)
  x[i] = (float *) malloc(M*sizeof(float));
```

  Le tableau est ainsi implémenté comme `N` pointeurs sur `N` vecteurs de taille `M` (chacun un bloc contigu de mémoire)
- Ainsi de suite en dimension supérieure...

Remarque: en C si `float *x;` alors `x[i]` est équivalent à `*(x+i)`

Quelques différences syntaxiques C/JAVA

- `System.out.println("Le resultat est:  "+x)` : `printf("Le resultat est:  %d\n",x)` (si `x` est de type `int`, sinon `%f` si de type `float` etc.)
- Voir la définition de `main`
- ...

Tableaux CUDA

- `cudaMalloc(void **x,int y);` alloue `y` octets dans la mémoire du GPU et renvoie un pointeur sur l'adresse ainsi allouée
- Exemple: allocation d'un tableau de 256 éléments flottants
  `float* devPtr; cudaMalloc( (void * *) & devPtr, 256 * sizeof(float) );`

ÉCOLE POLYTECHNIQUE

Tableaux CUDA

- Egalement: `cudaMallocPitch` pour un tableau 2D
- Exemple: allocation et l'utilisation d'un tableau 2D.

```
float* devPtr; int pitch;
cudaMallocPitch((void**)&devPtr,&pitch,&width,&height);
```

- On accède alors aux éléments du tableau 2D correspondant par:

```
for (int r = 0 ; r < height ; ++r) {
  float* row = (float *) ( (char *) devPtr + r * pitch);
  for (int c = 0 ; c < width ; ++c) {
    float element = row[c]; } } }
```

(similaire pour un tableau 3D)

- Permet d'améliorer les performances en respectant certaines contraintes d'"alignement" (adresses mémoire multiples de 16/64 bits...)

ÉCOLE POLYTECHNIQUE

Copie de tableaux (CUDA)

- `cudaMemcpy(void *dst,const void *src,size_t count,enum cudaMemcpyKind kind)`
  - Copie de l'adresse `src` vers l'adresse `dst`
  - `count` octets
  - dans la direction `cudaMemcpyHostToHost`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost` ou `cudaMemcpyDeviceToDevice`
- `cudaMemcpy` peut renvoyer une erreur
  - `cudaSuccess`
  - `cudaErrorInvalidValue`
  - `cudaErrorInvalidDevicePointer`
  - `cudaErrorInvalidMemcpyDirection`

ÉCOLE POLYTECHNIQUE

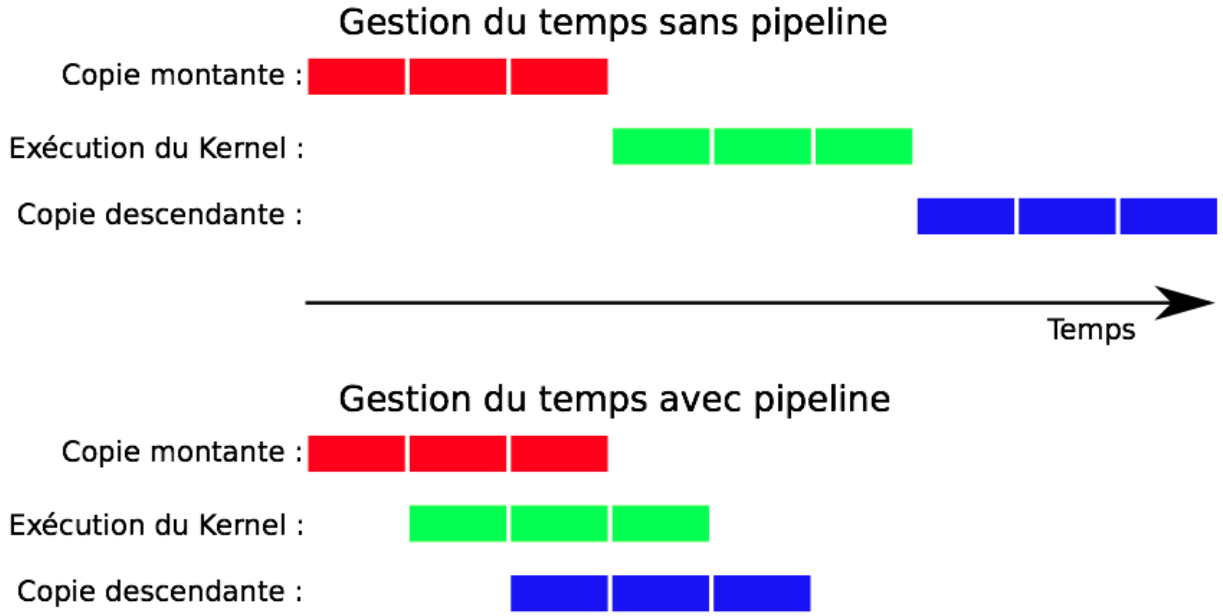
Pour aller plus loin

- `cudaMemcpyToArray(struct cudaArray * dstArray,size_t dstX,size_t dstY,const void * src,size_t count,enum cudaMemcpyKind kind)`
- `cudaMemcpy2D(void * dst,size_t dpitch,const void * src,size_t spitch,size_t width,size_t height,enum cudaMemcpyKind kind)`
- Depuis la carte pour allouer dans la mémoire du CPU: `cudaMallocHost(void **pointer, size_t size)` , `cudaFreeHost`

ÉCOLE POLYTECHNIQUE

Pour aller plus loin

- `cudaMemcpyAsync` pour des copies asynchrones (potentiellement, c.-à-d. non bloquantes):
  - mais il faut verrouiller des bouts de mémoire, et les déverouiller explicitement (`HostRegister`, `HostUnregister` sur l'hôte, à partir de la version 4 de CUDA)
  - ou alors associer les `cudaMemcpy` à des "CUDA streams": plusieurs "copy engines" (si l'architecture le permet) permettent de faire plusieurs transferts mémoires en parallèle, et en parallèle de l'exécution. On verra au cours 6/7 l'intérêt algorithmique de ces "recouvrements" communication/calcul.



Exemple de stream

```
// Create Streams
cudaStream_t streamA, streamB;
cudaStreamCreate(&streamA);
cudaStreamCreate(&streamB);

// Create Event
cudaEvent_t sync;
cudaEventCreate(&sync);

// Uploads
cudaMemcopy(pDeviceA, pHostA, sizeInByteA, cudaMemcpyHostToDevice, streamA);
cudaMemcopy(pDeviceA, pHostA, sizeInByteA, cudaMemcpyHostToDevice, streamA);

// Kernels
kernelA<<<BpG, TpB, 0, streamA>>>(pDeviceA);
kernelB<<<BpG, TpB, 0, streamB>>>(pDeviceB);

// Downloads
cudaMemcopy(pHostA, pDeviceA, sizeInByteA, cudaMemcpyDevicetoHost, streamA);
cudaMemcopy(pHostB, pDeviceB, sizeInByteB, cudaMemcpyDevicetoHost, streamB);
```

CUDA event

- Attaché à un CUDA stream
- Intérêt: opération de synchronisation permettant de faire attendre le CPU jusqu'à ce que l'évènement (event) ait été exécuté par le stream.
- cudaEventCreate, cudaEvenRecord pour l'attacher à un stream, cudaEventSynchronize pour synchroniser
- Peut être une synchronisation globale: il faut l'attacher au contexte d'exécution CUDA qui est le stream 0

```
// Synchronize on CUDA context
cudaEventRecord(sync, 0);
cudaEventSynchronize(sync);

// Destroy Event
cudaEventDestroy(sync);

// Destroy Streams
cudaStreamDestroy(streamA);
cudaStreamDestroy(streamB);
```

Compilation

- Les kernels CUDA sont écrits dans des fichiers `.cu`
- Les fonctions et main sur le CPU, dans des fichiers `.c` ou dans le même `.cu`
- Compilateur NVIDIA `nvcc` qui compile les `.c` en utilisant le compilateur C sous-jacent (`gcc` etc.) et les `.cu`

Exemple (spécifique salle TD):

```
[volvo ~]$ export PATH=/usr/local/cuda-5.0/bin:${PATH}
[volvo ~]$ export LD_LIBRARY_PATH=/usr/local/cuda-5.0/lib:$LD_LIBRARY_PATH
[volvo ~]$ nvcc -I/usr/local/cuda-5.0/include/
-I/users/profs/info/goubaul1/CUDA5.0/common/inc/
-L/users/profs/info/goubaul1/CUDA5.0/common/lib/ matrix.cu
```

Sinon...

Utiliser le template CUDA et son makefile

Sous /users/profs/info/goubaul1/CUDA5.0 (à recopier chez soi):

```
[volvo template]$ make
/usr/local/cuda-5.0/bin/nvcc -m32  -gencode arch=compute_10,code=sm_10 -gencode
arch=compute_20,code=sm_20 -gencode arch=compute_30,code=sm_30
-gencode arch=compute_35,code=sm_35 -I/usr/local/cuda-5.0/include -I. -I..
-I../../common/inc -o template.o -c template.cu
g++ -m32  -I/usr/local/cuda-5.0/include -I. -I.. -I../../common/inc -o
template_cpu.o -c template_cpu.cpp
g++ -m32 -o template template.o template_cpu.o -L/usr/local/cuda-5.0/lib -lcudart
mkdir -p ../../bin/linux/release
cp template ../../bin/linux/release
[volvo template]$ ls ../../bin/linux/release/template
../../bin/linux/release/template
[volvo template]$ ../../bin/linux/release/template
../../bin/linux/release/template Starting...

GPU Device 0: "GeForce GT 430" with compute capability 2.1

Processing time: 58.962002 (ms)
```

Utiliser nsight (Eclipse)

Reprendre exactement la manip sur la page TD 3!

Exécution

```
[volvo ~]$ nvcc -I/usr/local/cuda/include ...
[volvo ~]$ ./matrix
Processing time on GPU: 1.346160 (ms)
Processing time on CPU: 3.132520 (ms)
```

(en utilisant les fonctions de `helper....h`)
Code sur GPU presque 2.5 fois plus rapide que sur CPU. (avec les cartes l'année dernière, 51 fois plus rapide!)

ÉCOLE POLYTECHNIQUE

Synchronisation en CUDA

- Même problème qu'en JAVA, à cause de la mémoire partagée...
- Pas de sémaphores, mais quelques fonctions de synchronisation:
  - intra-bloc: `__syncthreads();`
  - entre le GPU et le CPU (très inefficace: attend la fin de l'appel au kernel): `cudaThreadSynchronize();` (mais aussi les `cudaMemcpy();` par effet de bord)
  - les fonctions "atomiques" (ininterruptibles)
- Ces fonctions atomiques font des opérations "read-modify-write" sans interférence par d'autre threads
- S'appliquent à la mémoire globale et à la mémoire partagée

ÉCOLE POLYTECHNIQUE

Quelques fonctions atomiques CUDA

- `int atomicAdd(int *address, int val);` (ajoute val à la valeur pointée à l'adresse `address`)
- `int atomicSub(int *address, int val);` (retire val à la valeur pointée à l'adresse `address`)

Ces deux fonctions retournent les valeurs avant modification.

Autres fonctions atomiques CUDA

- `int atomicExch(int* address, int val);` stocke `val` dans `address`
- `int atomicMin(int* address, int val);`
- `int atomicMax(int *address, int val);`
- `unsigned int atomicInc(unsigned int* address, unsigned int val);` incrémente `address` de 1 si la valeur stockée précédemment est inférieure ou égale à `val`

ÉCOLE POLYTECHNIQUE

Autres fonctions atomiques CUDA

- `unsigned int atomicDec(unsigned int* address, unsigned int val);`
- `int atomicAnd(int* address, int val);`
- `int atomicOr(int* address, int val);`
- `int atomicXor(int* address, int val);`

Compare and Swap

Fonction très utile:

- `int atomicCAS(int* address, int compare, int val);`
- compare la valeur à l'adresse `address` à `compare` et écrit `val` à l'adresse `address` seulement si cette valeur est égale à `compare`
- permet de coder les sémaphores binaires (et ainsi de suite, les sémaphores à compteur etc.)

Memory Fences

Attention, les écritures dans un threads peuvent ne pas être terminées et "visibles" d'autres threads, dans un bloc, ou de façon globale.

- `void __threadfence();`
- `void __threadfence_system();`
- `void __threadfence_block();`
- ces instructions forcent l'écriture, et rendent visibles les écritures au niveau de la carte, du système et du bloc respectivement

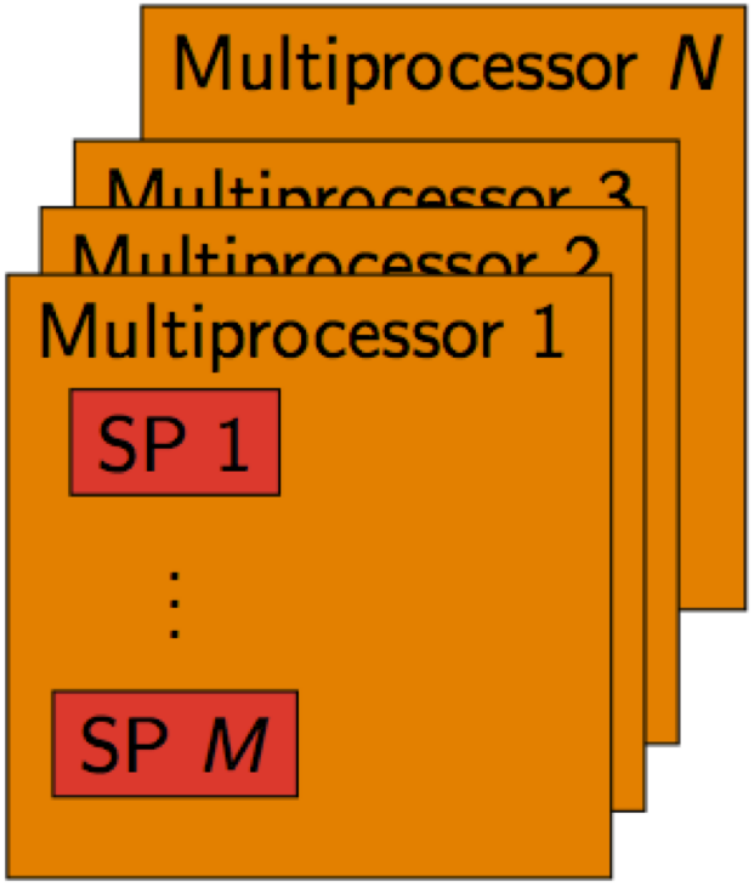
ÉCOLE POLYTECHNIQUE

Sémaphores binaires en CUDA

```
__device__ int lock=0;

__global__void kernel(...) {
  ...
  // P(lock)
  do { } while (atomicCAS(&lock, 0, 1));
  ...
  __threadfence();
  // V(lock)
  lock=0;
  ... }
```

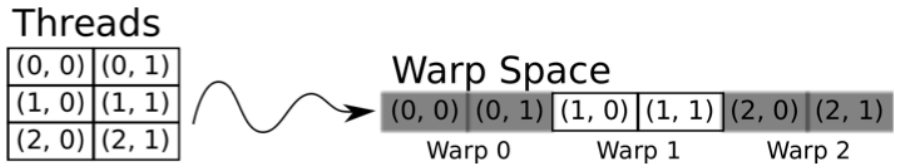
Marche si le nombre de blocs est inférieur ou égal au nombre de threads.

Mapping physique block→multiprocesseurs



Runtime

- Un block est exécuté par un seul multiprocesseur
- Chaque block est divisé en groupes de threads (“physiques”) appelés “warps”
- Un warp (en général 32 threads) est exécuté physiquement en parallèle
- Un warp est constitué de threads de `threadIdx` consécutifs et croissants
- L’ordonnanceur de la carte alterne entre les warps

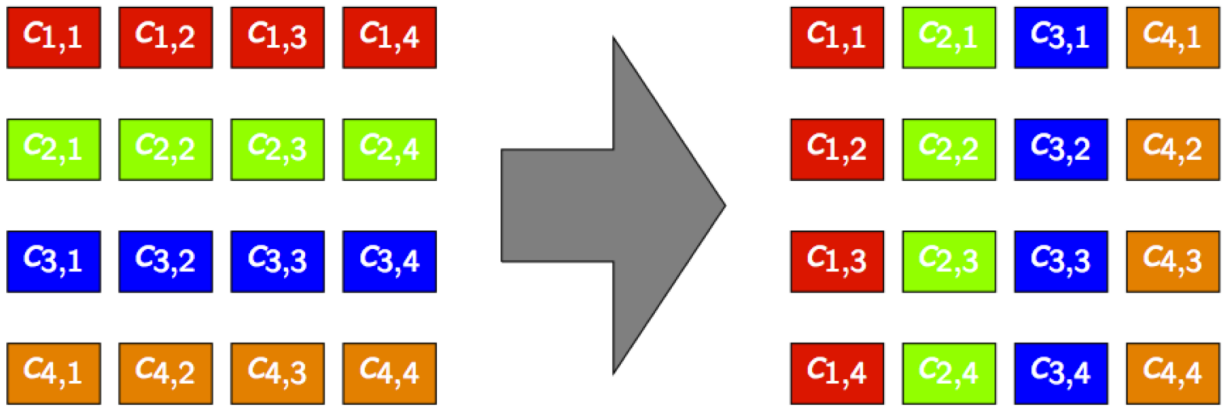


Efficacité/sécurité

- Pour être efficace (pour que l'ordonnanceur ait toujours quelque chose à ordonnancer), il faut essayer de définir un nombre de blocks de 2 à 100 fois égal au nombre de multiprocesseurs.
- De même, on essaie de définir plusieurs warps par multiprocesseurs (tirer parti du recouvrement potentiel calcul/accès mémoire)
- Ne pas oublier `__syncthreads()` et `cudaThreadSynchronize()` (au niveau du CPU) pour assurer les fonctions de barrière de synchronisation (resp. attente qu'un kernel soit terminé)

ÉCOLE POLYTECHNIQUE

Optimisation - ex.: transposition



Optimisation - mémoire globale "amalgamée"

```
__global__ void transpose_naive
   (float *out, float *in, int w, int h ) {
  unsigned int xIdx = blockDim.x * blockIdx.x + threadIdx.x;
  unsigned int yIdx = blockDim.y * blockIdx.y + threadIdx.y;
  if ( xIdx < w && yIdx < h ) {
    unsigned int idx_in = xIdx + w * yIdx;
    unsigned int idx_out = yIdx + h * xIdx;
    out[idx_out] = in[idx_in];
  }
}
```

Pas rapide...

```
[volvo ~]$ nvcc -I/usr/local/cuda/include ...
[volvo ~]$ ./transpose
Processing time (naive) on GPU: 1.006980 (ms)
Processing time on CPU: 10.051590 (ms)
```

Accélération de 10 fois avec l'approche naive...(9 fois, l'année dernière)

ÉCOLE POLYTECHNIQUE

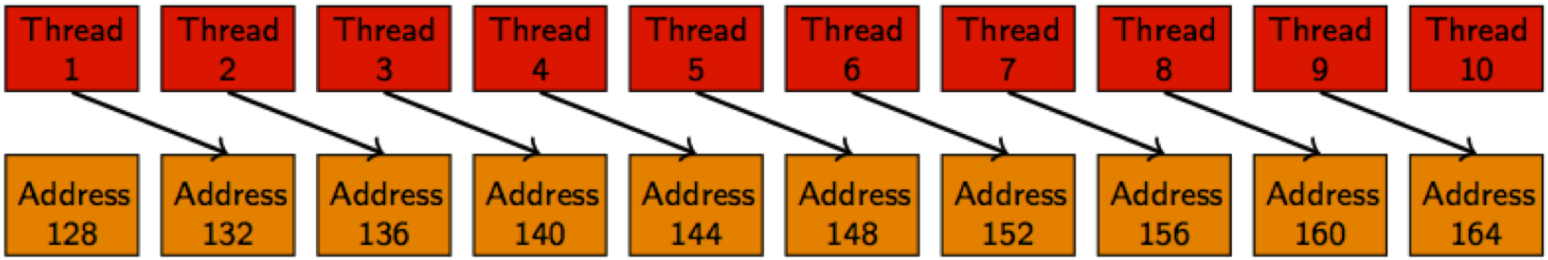
Mémoire partagée et accès à la mémoire globale “amalgamés”:

- En général, il est bien meilleur de passer par la mémoire partagée pour des calculs intensifs
- Dans ce cas, on alloue les données initiales dans la mémoire globale du GPU, on recopie les données depuis le CPU...
- Puis on recopie des bouts de ces données dans la mémoire partagée de chaque block (à la fin on recopiera les résultats de la mémoire partagée à la mémoire globale, puis au CPU)

Optimisation

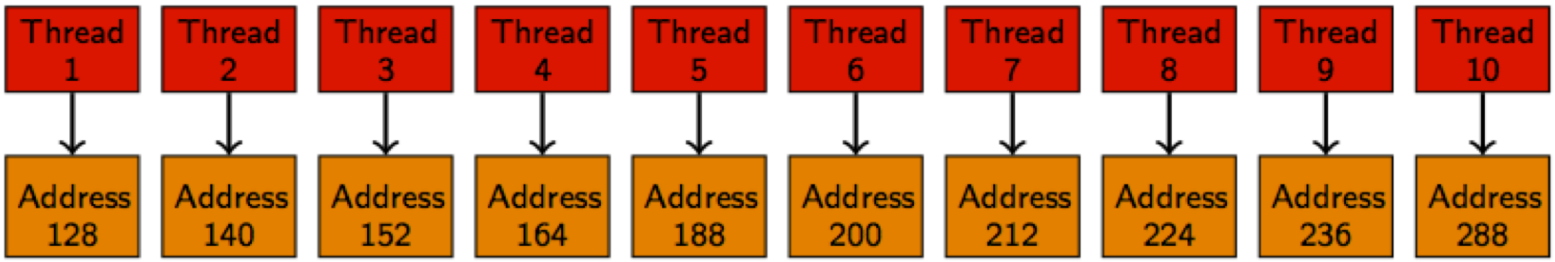
- Point important pour la bonne utilisation de la bande passante mémoire globale vers mémoire partagée: “amalgamation”...
- Il faut faire attention à deux choses dans l'utilisation de la mémoire partagée:
  - Bien utiliser `__syncthreads()`: permet d'attendre que tous les threads d'un même block ont bien lu ou écrit leurs données en mémoire partagée ou globale par exemple, avant de continuer un calcul...
  - Bank conflict...

Explication: accès mémoire non-amalgamé



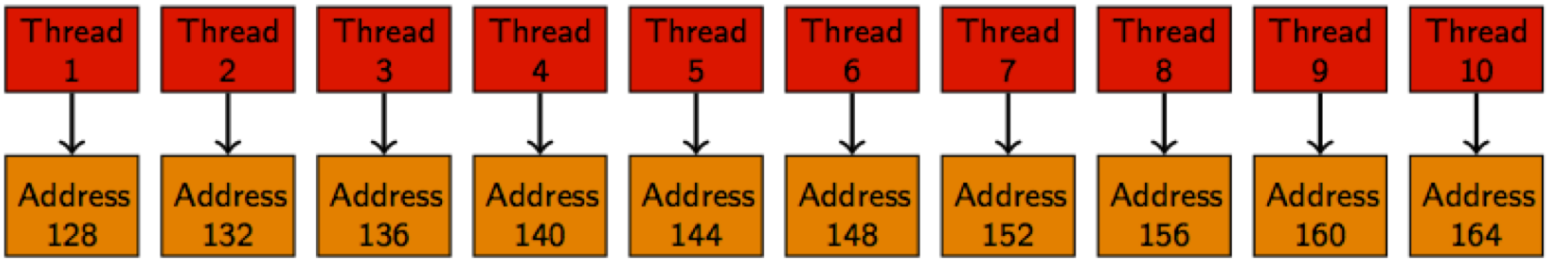
Accès non aligné modulo 16

Explication: accès mémoire non-amalgamée



Adresses non “connexes” dans un bloc

Ce qu'il faut faire: accès mémoire amalgamée

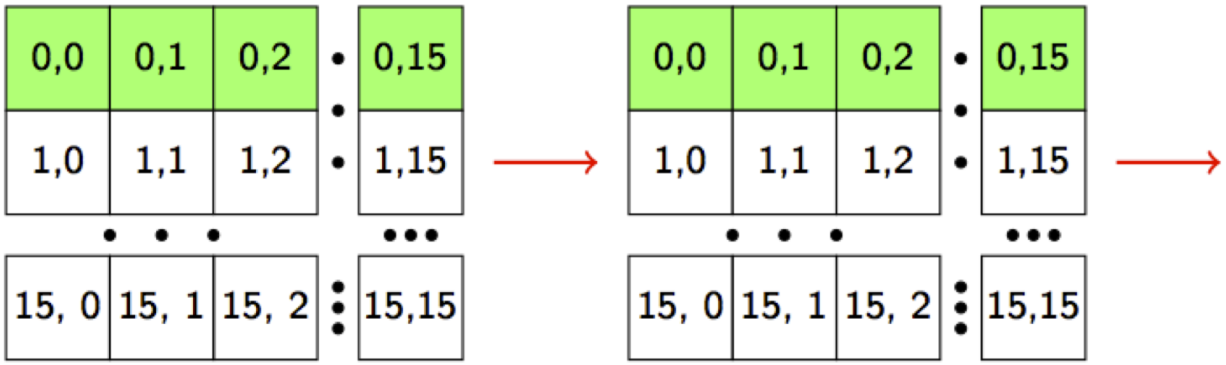


On doit accéder à la mémoire par des accès 8, 16, 32/64 128 bits consécutifs (dans l'ordre des threads pour avant 1.2 - ce n'est pas notre cas ici), dans un bloc mémoire de 32, 64 ou 128 octets; adresse de départ alignée modulo 16

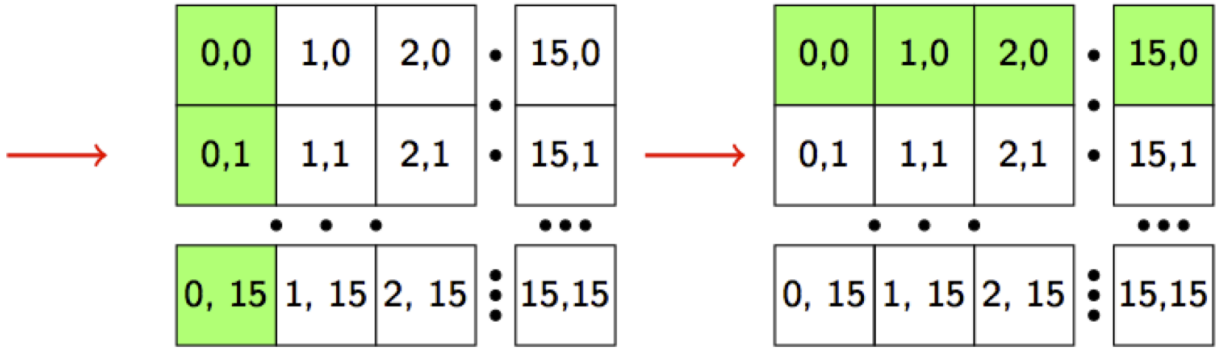
Version amalgamée

- La matrice est partitionnée en sous-blocs carrés
- Un bloc carré est associé à un block (bx,by):
  - Charger le bloc (bx,by) de la mémoire globale vers la mémoire partagée
  - Faire la transposition en mémoire partagée (pas de problème d'amalgamation, juste les "bank conflicts...") en parallèle sur tous les thread processor
- Ecrire le résultat dans la mémoire globale, par blocs contigus

Ce qu'il faut faire pour la transposition:



Lecture de la mém. globale ; Ecriture en mémoire partagée



Lire les adresses transposées en SMEM ; Ecrire dans la mém. globale

Version amalgamée

```
__global__ void transpose
  (float *out, float *in, int width, int height ) {
  __shared__ float block[BLOCK_DIM*BLOCK_DIM];
  unsigned int xBlock = blockDim.x * blockIdx.x;
  unsigned int yBlock = blockDim.y * blockIdx.y;
  unsigned int xIndex = xBlock + threadIdx.x;
  unsigned int yIndex = yBlock + threadIdx.y;
  unsigned int index_out, index_transpose;

 if ( xIndex < width && yIndex < height ) {
  unsigned int index_in=width*yIndex+ xIndex;
  unsigned int index_block=threadIdx.y*BLOCK_DIM+threadIdx.x;
  block[index_block]=in[index_in];
  index_transpose=threadIdx.x*BLOCK_DIM+threadIdx.y;
  index_out=height*(xBlock+threadIdx.y)+yBlock+threadIdx.x;
 }
 __synchthreads();
 if ( xIndex < width && yIndex < height ) {
  out[index_out]=block[index_transpose];}
}
```

Dans le main

```
N = 1024
blocksize = BLOCK_DIM = 16;

dim3 dimBlock( blocksize, blocksize );
dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );

transpose<<<dimGrid, dimBlock>>>( ad, bd, N, N );
```

Compilation et exécution

```
[finlande ~]$ nvcc -I/usr/local/cuda/include ...
[finlande ~]$ ./transpose
Processing time (naive) on GPU: 1.006980 (ms)
Processing time (coalesced) on GPU: 0.918700 (ms)
Processing time on CPU: 10.051590 (ms)
```

Accélération de 10 fois avec l'approche naive...de 12 fois avec l'approche améliorée (30 fois l'année dernière)

```
is148098:Cours03 Eric$ ./transpose2
Processing time (naive) on GPU: 1.978900 (ms)
Processing time (coalesced) on GPU: 0.943500 (ms)
Processing time on CPU: 11.288430 (ms)
```

Amélioration de plus de deux fois sur mon Mac (par rapport à la version naïve)...dépend de l'architecture du nombre de threads, de blocks déclarés etc.

ÉCOLE POLYTECHNIQUE

Retour sur la PRAM: algorithme (naif) scan CUDA

- Limitée à des tableaux de 512 éléments (nombre de threads maxi sur un multi-processeur) - cf. `scan` de la SDK (un peu plus compliqué...)

```
for d := 1 to log2n do
    forall k in parallel do
        if k ≥ 2^d then
            x[out][k] := x[in][k − 2^(d-1)] + x[in][k]
        else
            x[out][k] := x[in][k]
    swap(in, out)
```

- Ici, 1 thread processor par instance de boucle, dans une grid (1,1,1), de blocs unidimensionnels
- En fait, il faut passer à des plus gros tableaux et à une optimisation plus fine pour avoir de bonnes performances...

Retour sur scan: implémentation CUDA (naive)

(scan_kernel.cu)

```
__global__ void scan(float *g_odata, float *g_idata, int n)
{
// Dynamically allocated shared memory for scan kernels
extern __shared__ float temp[];
int thid = threadIdx.x;
int pout = 0;
int pin = 1;
// Cache the computational window in shared memory
temp[pout*n + thid] = (thid > 0) ? g_idata[thid-1] : 0;
```

Subtilité

- Ce code nous permet de voir une nouveauté en CUDA...
- Jusqu'ici nous n'avons alloué la mémoire partagée statiquement: `__shared__ float x[100];`
- Quand on ne connait pas avant l'exécution la taille souhaitée: `extern __shared float x[];`: le `extern` est obligatoire...
- ...et il y a une subtilité à l'appel du kernel...(plus loin)

ÉCOLE POLYTECHNIQUE

Implémentation CUDA

```
for (int offset = 1; offset < n; offset *= 2)
{
  pout = 1 - pout;
  pin  = 1 - pout;
  __syncthreads();
  temp[pout*n+thid] = temp[pin*n+thid];
  if (thid >= offset)
    temp[pout*n+thid] += temp[pin*n+thid - offset];
  }
  __syncthreads();
  g_odata[thid] = temp[pout*n+thid];
}
```

ÉCOLE POLYTECHNIQUE

Implémentation CUDA

(scan.cu)

```
int main( int argc, char** argv)
{    runTest( argc, argv);
     return 1; }
void runTest( int argc, char** argv)
{
// initialize the input data on the host to be integer values
// between 0 and 1000
for( unsigned int i = 0; i < num_elements; ++i)
  h_data[i] = floorf(1000*(rand()/(float)RAND_MAX));
// compute reference solution
float* reference = (float*) malloc( mem_size);
computeGold( reference, h_data, num_elements);
```

Implémentation CUDA

```
// allocate device memory input and output arrays
float* d_idata;
float* d_odata[3];
cudaMalloc( (void**) &d_idata, mem_size);
cudaMalloc( (void**) &(d_odata[0]), mem_size);
cudaMalloc( (void**) &(d_odata[1]), mem_size);
cudaMalloc( (void**) &(d_odata[2]), mem_size);
// copy host memory to device input array
cudaMemcpy( d_idata, h_data, mem_size, cudaMemcpyHostToDevice);
```

ÉCOLE POLYTECHNIQUE

Implémentation CUDA

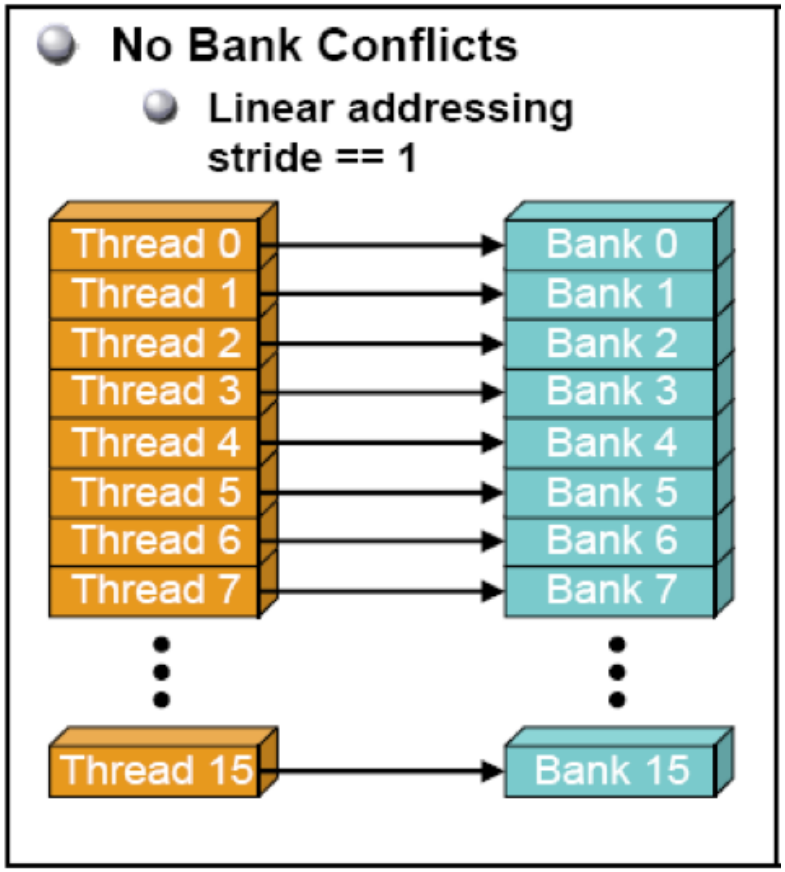
```
// setup execution parameters
// Note that these scans only support a single thread-block worth of data,
// but we invoke them here on many blocks so that we can accurately compare
// performance
    dim3  grid(256, 1, 1);
    dim3  threads(num_threads*2, 1, 1);

unsigned int numIterations = 100;
for (unsigned int i = 0; i < numIterations; ++i)
  {
  scan<<< grid, threads, 2 * shared_mem_size >>>
            (d_odata[0], d_idata, num_elements);
  }
cudaThreadSynchronize();
...
```

Allocation de la mémoire partagée

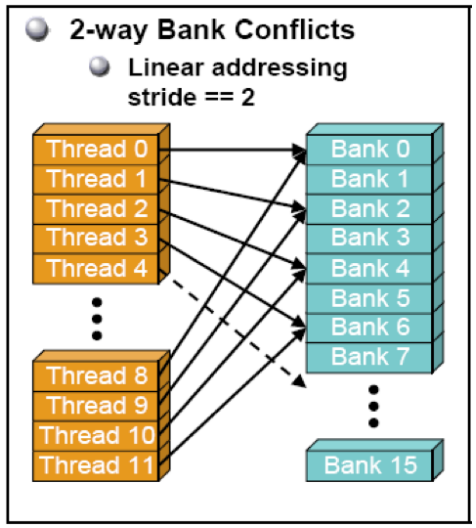
- Il faut passer, au moment de l'appel au kernel, la taille à allouer en mémoire partagée par multiprocesseur
- `kernel<<<griddim, blockdim, sharedmemsize>>>`
- Attention, on n'alloue qu'un bloc mémoire contigu comme cela, charge au programmeur de le découper en les données dont il a besoin...

Pour aller plus loin: "Bank conflict"?

- La mémoire d'une machine parallèle à mémoire partagée est généralement partitionnée en "bank" sur lesquelles un seul `read` ou un seul `write` (de mots 32 bits ici) est effectué à tout instant (2 cycles en général)
- Cf. PRAM EREW...



"Bank conflict"?



- Cas de conflit:
- Perte de performance: sérialisation des accès mémoire...
- Cas particulier: tous les threads d'un warp lisent la même adresse en même temps...pas de conflit!
- Ex. les cartes G80:
  - ont 16 banks, faits de mots 32 bits contigus
  - Donc # bank=adresse % 16
  - attention à l'alignement des données! (`__ALIGN(X)`)

Exemple de "bank conflict"

Code standard:

```
__global__ void test(int *gpA)
{
    __shared__ int sa[16];
    sa[0]=3; // bank conflict if blocksize > 1
    gpA[0]=sa[0]; // bank conflict again
}
```

Template /usr/local/cuda-5.0/samples/0_Simple/template/template.cu

```
// includes, system
include <stdlib.h>include <stdio.h>
include <string.h>include <math.h>

// includes CUDA
...
// includes, project
...

////////////////////////////////////////////////////////////////////////////////
// declaration, forward
void runTest(int argc, char **argv);

extern "C"
void computeGold(float *reference, float *idata, const unsigned int len);
```

TEMPLATE.CU

```
////////////////////////////////////////////////////////////////////////////////
//! Simple test kernel for device functionality
//! @param g_idata  input data in global memory
//! @param g_odata  output data in global memory
////////////////////////////////////////////////////////////////////////////////
__global__ void
testKernel(float *g_idata, float *g_odata)
{
    // shared memory
    // the size is determined by the host application
    extern  __shared__  float sdata[];

    // access thread id
    const unsigned int tid = threadIdx.x;
    // access number of threads in this block
    const unsigned int num_threads = blockDim.x;

    // read in input data from global memory
    sdata[tid] = g_idata[tid];
    __syncthreads();

    // perform some computations
    sdata[tid] = (float) num_threads * sdata[tid];
    __syncthreads();

    // write data to global memory
    g_odata[tid] = sdata[tid];
}
```

TEMPLATE.CU

```
////////////////////////////////////////////////////////////////////////////////
// Program main
////////////////////////////////////////////////////////////////////////////////
int
main(int argc, char **argv)
{
    runTest(argc, argv);
}

////////////////////////////////////////////////////////////////////////////////
//! Run a simple test for CUDA
////////////////////////////////////////////////////////////////////////////////
void
runTest(int argc, char **argv)
{
    bool bTestResult = true;

    printf("%s Starting...\n\n", argv[0]);

    // use command-line specified CUDA device, otherwise use device with highest Gflops/
    int devID = findCudaDevice(argc, (const char **)argv);

    StopWatchInterface *timer = 0;
    sdkCreateTimer(&timer);
    sdkStartTimer(&timer);

    unsigned int num_threads = 32;
    unsigned int mem_size = sizeof(float) * num_threads;

    // allocate host memory
    float *h_idata = (float *) malloc(mem_size);
```

TEMPLATE.CU

```
// initialize the memory
for (unsigned int i = 0; i < num_threads; ++i)
{     h_idata[i] = (float) i;     }
// allocate device memory
float *d_idata;
checkCudaErrors(cudaMalloc((void **) &d_idata, mem_size));
// copy host memory to device
checkCudaErrors(cudaMemcpy(d_idata, h_idata, mem_size,
                           cudaMemcpyHostToDevice));

// allocate device memory for result
float *d_odata;
checkCudaErrors(cudaMalloc((void **) &d_odata, mem_size));

// setup execution parameters
dim3  grid(1, 1, 1);
dim3  threads(num_threads, 1, 1);
// execute the kernel
testKernel<<< grid, threads, mem_size >>>(d_idata, d_odata);
// check if kernel execution generated and error
getLastCudaError("Kernel execution failed");

// allocate mem for the result on host side
float *h_odata = (float *) malloc(mem_size);
// copy result from device to host
checkCudaErrors(cudaMemcpy(h_odata, d_odata, sizeof(float) * num_threads,
                           cudaMemcpyDeviceToHost));

sdkStopTimer(&timer);
printf("Processing time: %f (ms)\n", sdkGetTimerValue(&timer));
sdkDeleteTimer(&timer);
```

TEMPLATE.CU

```
    // compute reference solution
    float *reference = (float *) malloc(mem_size);
    computeGold(reference, h_idata, num_threads);

    // check result
    if (checkCmdLineFlag(argc, (const char **) argv, "regression"))
    {
        // write file for regression test
        sdkWriteFile("./data/regression.dat", h_odata, num_threads, 0.0f, false);
    }
    else
    {
        // custom output handling when no regression test running
        // in this case check if the result is equivalent to the expected soluion
        bTestResult = compareData(reference, h_odata, num_threads, 0.0f, 0.0f);
    }

    // cleanup memory
    free(h_idata);
    free(h_odata);
    free(reference);
    checkCudaErrors(cudaFree(d_idata));
    checkCudaErrors(cudaFree(d_odata));

    cudaDeviceReset();
    exit(bTestResult ? EXIT_SUCCESS : EXIT_FAILURE);
}
```

TEMPLATE_CPU.CPP

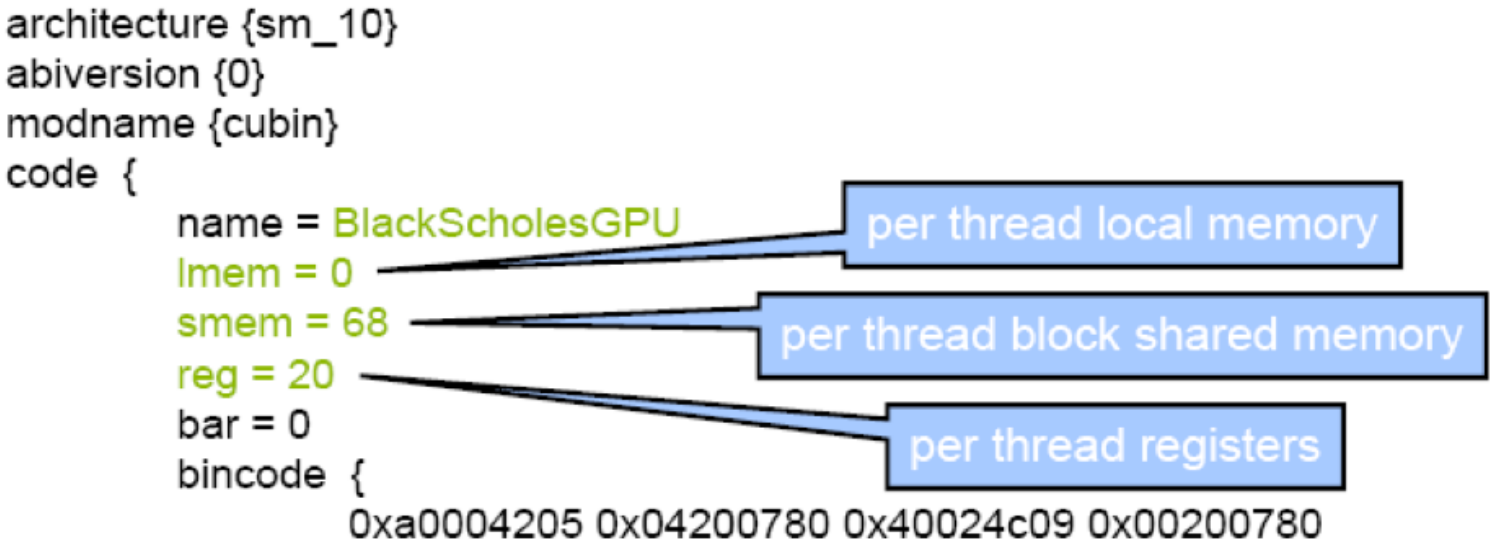
```
////////////////////////////////////////////////////////////////////////////////
// export C interface
extern "C"
void computeGold(float *reference, float *idata, const unsigned int len);

////////////////////////////////////////////////////////////////////////////////
//! Compute reference data set
//! Each element is multiplied with the number of threads / array length
//! @param reference  reference data, computed but preallocated
//! @param idata      input data as provided to device
//! @param len        number of elements in reference / idata
////////////////////////////////////////////////////////////////////////////////
void
computeGold(float *reference, float *idata, const unsigned int len)
{
    const float f_len = static_cast<float>(len);

    for (unsigned int i = 0; i < len; ++i)
    {
        reference[i] = idata[i] * f_len;
    }
}
```

Remarques... sur les performances

- Ne vous laissez pas décourager par de piètres performances pour une première version de vos programmes (de plus les nouvelles cartes sont bien moins performantes que celles de l'année dernière)
- Essayez de comprendre les raisons:
  - bank conflict
  - transferts de données trop importants entre CPU et GPU pour un calcul trop court
  - trop de passage par la mémoire globale du GPU, et pas assez par la mémoire partagée au niveau des multi-processeurs
  - pour les experts: problèmes d'alignement des données (cf. présentation "optimisation CUDA" sur page web)
- Utilisez le "occupancy calculator" (feuille excel - cf. page `nvidia.com/cuda`) et éventuellement le profiler sous `nsight`

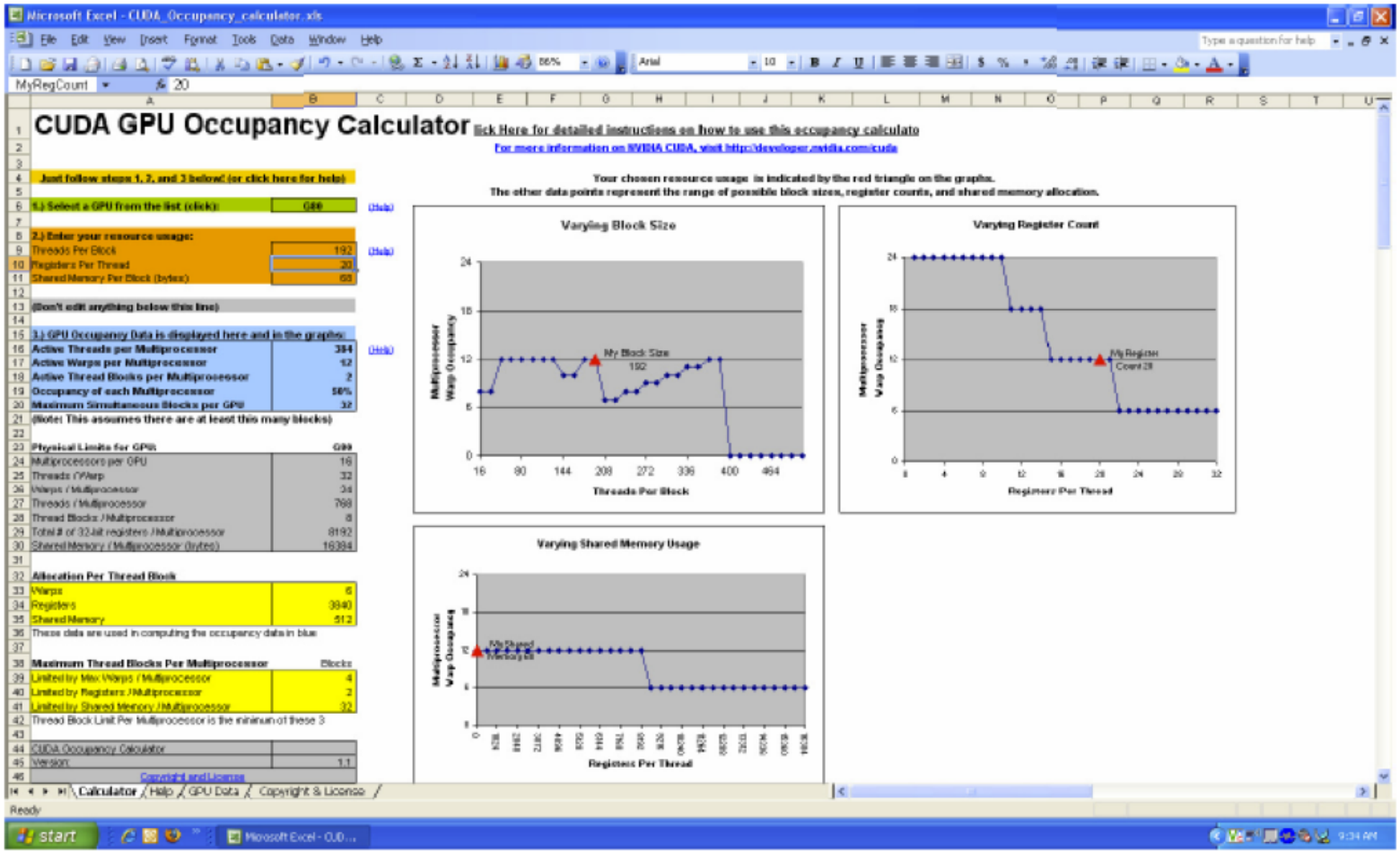
Cubin et "Occupancy Calculator"

Compiler avec `nvcc -cubin`, ouvrir le fichier `...cubin`:



Cubin et "Occupancy Calculator"

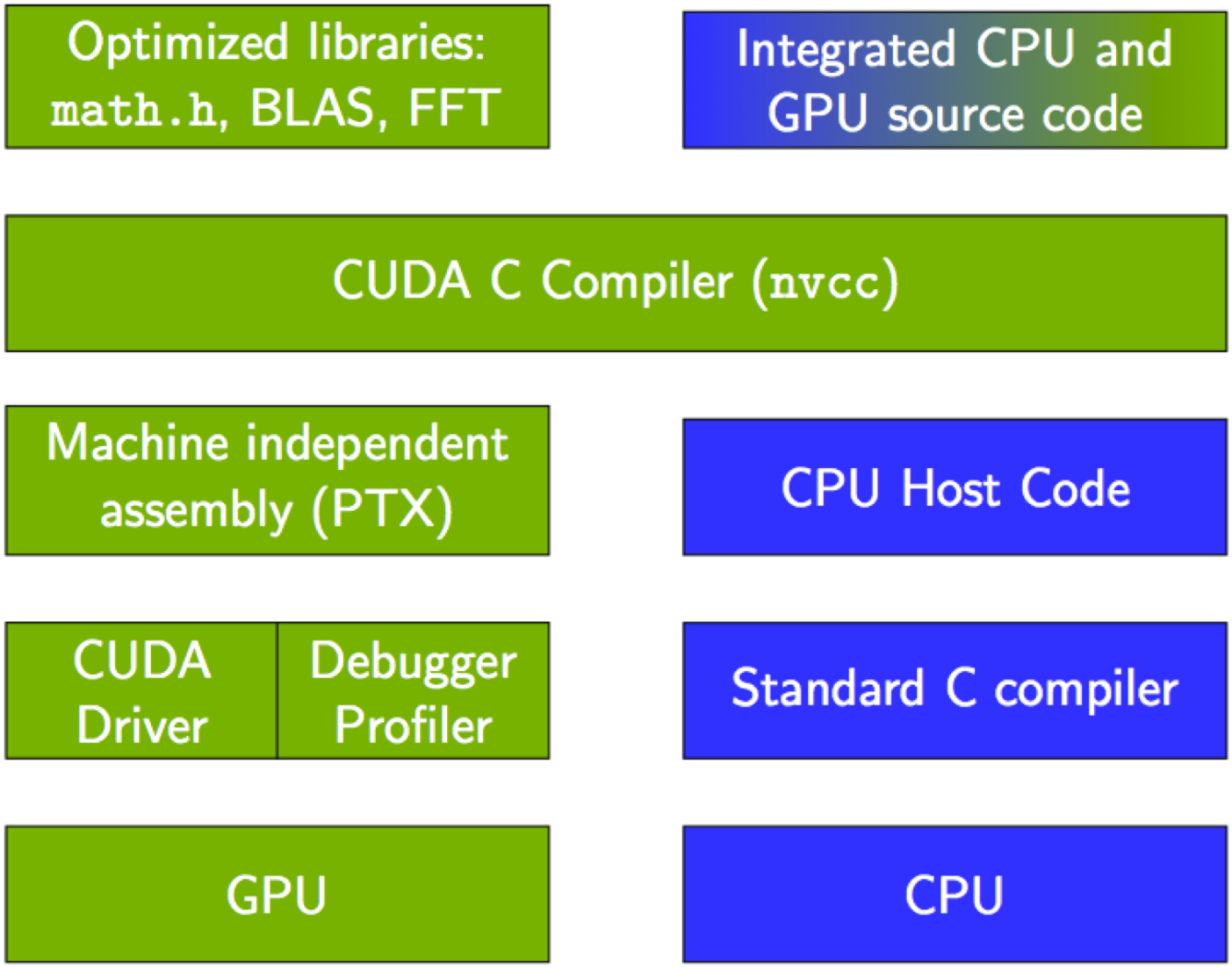
Feuille excel: permet de déterminer au mieux le nombre de threads et block:



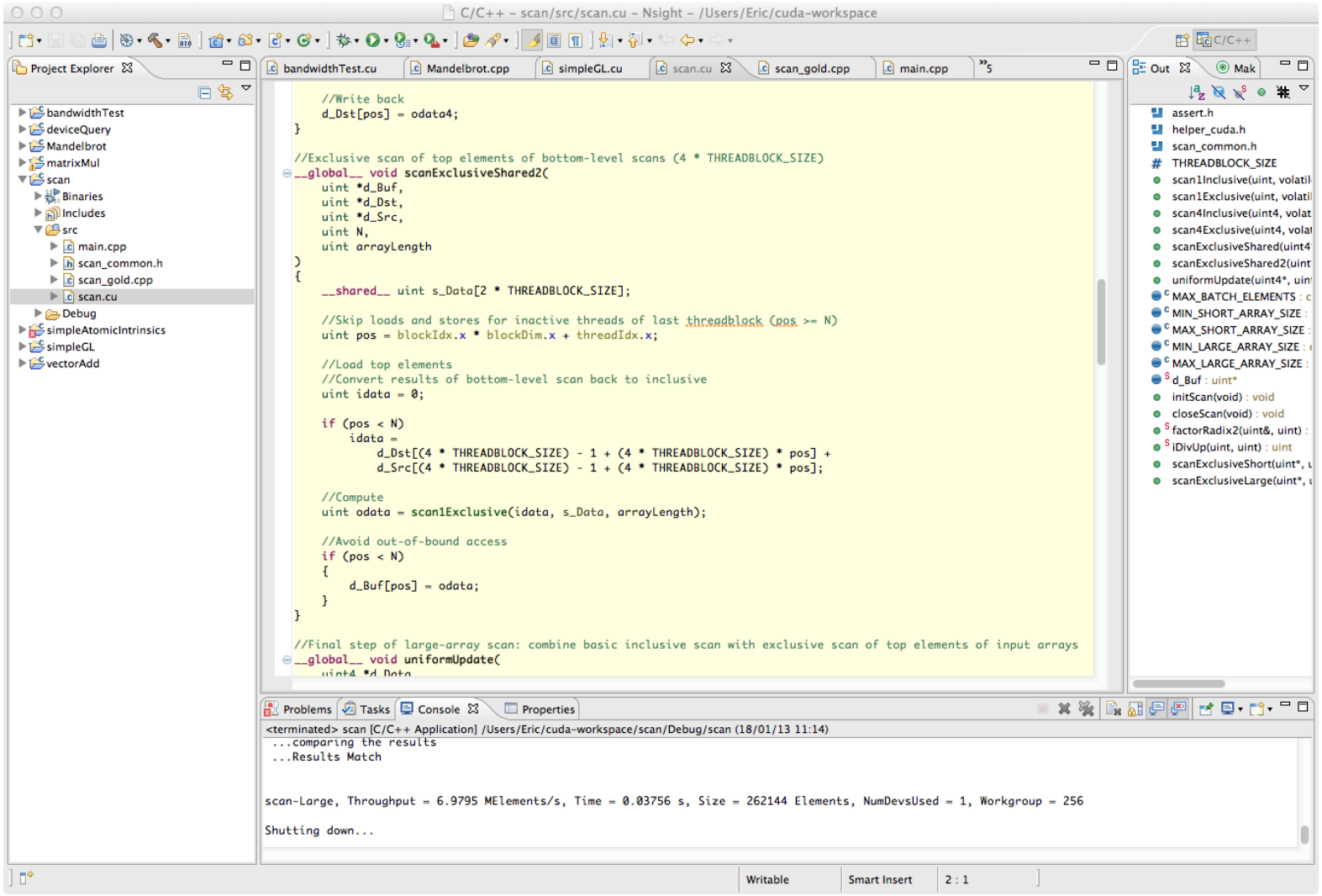
Performances...suite

- Choisir un nombre de threads grand (multiple du nombre de threads par warp: 32...) pour cacher la latence d'accès à la mémoire
  - Typiquement 128 à 256 (min: 64, max: 512 en général)
  - Mais plus il y a des threads dans un block...plus cela peut être lent quand on fait `__syncthreads()`...
- Choisir un nombre de blocks important (au moins double du nombre de multiprocesseurs - : $\geq$ 10 typiquement)

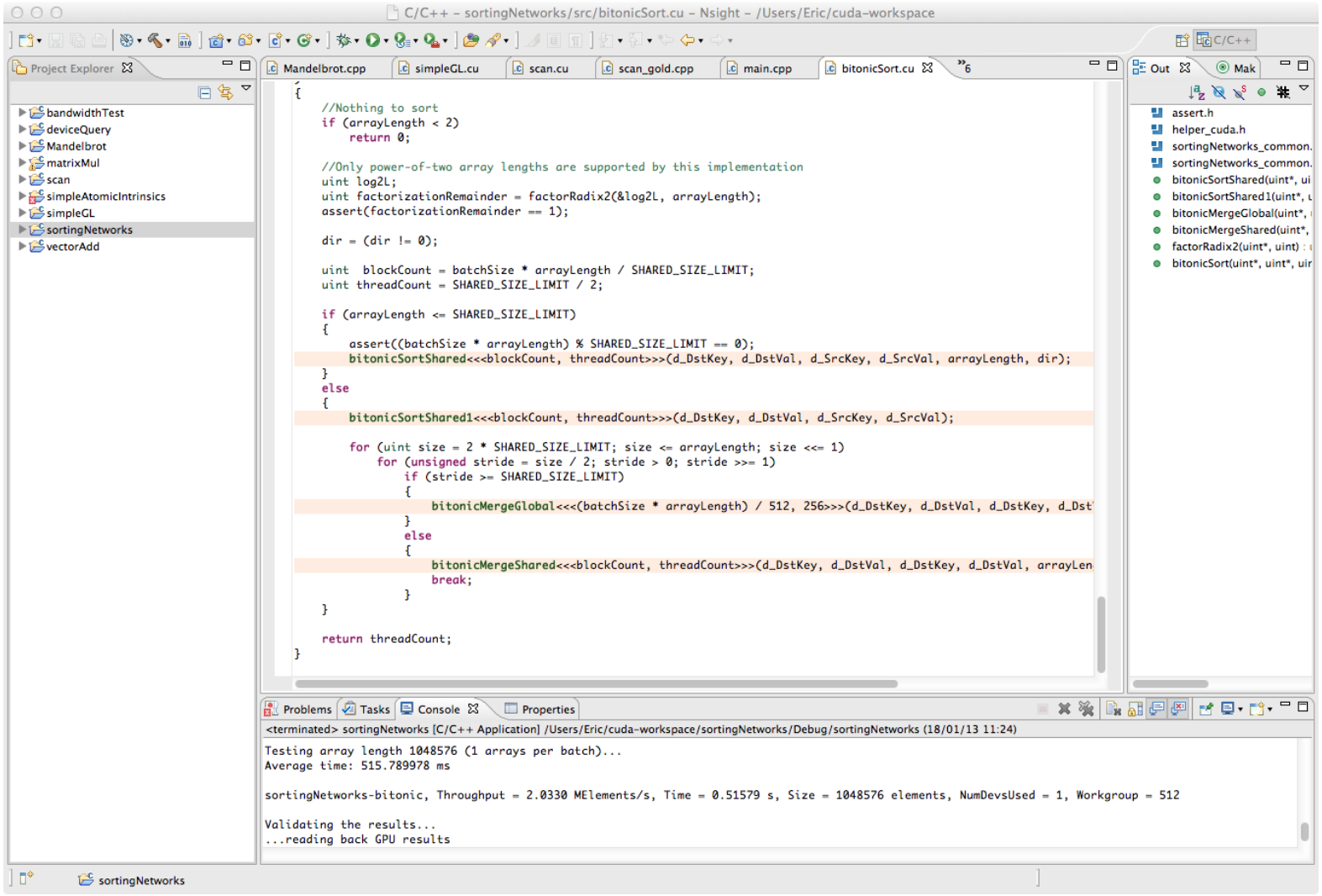
CUDA SDK


Optimized libraries:
math.h, BLAS, FFT
Integrated CPU and
GPU source code
CUDA C Compiler (nvcc)
Machine independent
assembly (PTX)
CPU Host Code
CUDA
Driver
Debugger
Profiler
Standard C compiler
GPU
CPU

La version de la SDK 5.0



Remarque: exemple de réseau de tri en CUDA



Pour aller plus loin (OpenCL)

- Multi-plateforme, beaucoup plus verbeux
- Le code GPU est compilé à l'exécution du code host.
- Vocabulaire un peu différent...:

CUDA	OpenCL
Thread	Work item
Block	Work group
Grid	NDRange
Shared memory	Local memory
Registers	Private memory

Exemple de programme OpenCL

```
  cl_program program[1];
  cl_kernel kernel[2];
  cl_command_queue cmd_queue;
  cl_context   context;
  cl_device_id cpu = NULL, device = NULL;
  cl_int err = 0;
  size_t returned_size = 0;
  size_t buffer_size;
  cl_mem a_mem, b_mem, ans_mem;

  // Find the CPU CL device, as a fallback
  err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1, &cpu, NULL);

  // Now create a context to perform our calculation with the
  // specified device
  context = clCreateContext(0, 1, &device, NULL, NULL, &err);
  // And also a command queue for the context
  cmd_queue = clCreateCommandQueue(context, device, 0, NULL);

  // Load the program source from disk
  const char * filename = "example.cl";
  char *program_source = load_program_source(filename);
  program[0] = clCreateProgramWithSource(context, 1, (const char**)&program_source,
  NULL, &err);
  // Now create the kernel "objects" that we want to use in the example file
  kernel[0] = clCreateKernel(program[0], "add", &err);
...
  // example.cl
  __kernel void
  add(__global float *a, __global float *b, __global float *answer)
  {     int gid = get_global_id(0);
        answer[gid] = a[gid] + b[gid];     }
```