INF 560
Calcul Parallèle et Distribué
Cours 4

Eric Goubault et Sylvie Putot

Ecole Polytechnique

12 janvier 2015

Plan du cours

- Retour sur la mémoire
  - mémoire partagée
  - exemple de la transposée
- Flot de contrôle et synchronisation en CUDA
  - flot de contrôle
  - synchronisation intra-block, device/host
  - opérations atomiques en CUDA
  - application au calcul non typiquement SIMT et au calcul de π, cf. TD
- Quelques fonctionnalités avancées (pistes pour les projets...)

Hiérarchie mémoire

- Rappel: accès à la mémoire globale couteux
  - Essayer d'avoir au maximum des accès mémoire amalgamés (threads d'un warp accèdent à des éléments consécutifs de la mémoire)
  - En général, meilleur de passer par la mémoire partagée pour des calculs: copie de bouts de donnée de la mémoire globale vers la mémoire partagée de chaque blcos
- Mémoire partagée (par blocs, qualificatif `__shared`)
  - Taille limitée
  - Optimisation de l'accès à la mémoire partagée: éviter les "bank conflicts"

Bancs mémoire

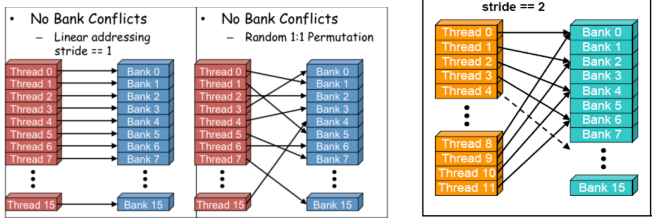
La mémoire d'une machine parallèle à mémoire partagée est généralement partitionnée en bancs (32 pour archi Fermi ou Kepler) sur chacun desquels un seul `read` ou un seul `write` est effectué à chaque instant

```
Bank    |      1      |      2      |      3      |...
Address |  0  1  2  3 |  4  5  6  7 |  8  9 10 11 |...
Address | 64 65 66 67 | 68 69 70 71 | 72 73 74 75 |...
...
```

Cartes Fermi/Kepler: 32 banks de mots de 32/64 bits contigus

Bank conflict?

- Cartes Fermi/Kepler: 32 banks de mots de 32/64 bits contigus
- Cas particulier ou pas de conflit: tous les threads d'un warp lisent la même adresse en même temps (broadcast)
- Sinon perte de performance: sérialisation des accès mémoire



Exemple: transposition de matrice

Version naive: accès non amalgamé en mémoire globale

```
__global__ void transpose_naive(float *out, float *in, int w, int
  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];
  } }
```

Optimisation en passant par la mémoire partagée

- La matrice est partitionnée en sous-blocs carrés
- Un bloc de la matrice est associé à un bloc de threads:
  - Charger le bloc 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 threads
- Ecrire le résultat dans la mémoire globale, par blocs

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;
  }
  __syncthreads();
  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 );
```

Remarques

- En TD: produit de matrices en utilisant la mémoire partagée:
  - version naive serait de charger une ligne et une colonne entière en mémoire partagée par thread. Potentiellement trop gros pour la mémoire partagée, avec lecture non amalgamée en mémoire globale
  - suggestion: version par blocs
- Mémoire partagée allouée statiquement comme dans l'exemple de la transposée; ou dynamiquement au lancement du kernel (cf après)
- Synchronisation et flot de contrôle ?

Flot de contrôle: "warp divergence"

- Les threads sont exécutés par warps de 32 threads, où tous les threads exécutent la même instruction en même temps
- Que se passe-t'il si les différents threads d'un warp ne doivent pas tous effectuer la même chose
  - cf exemple précédent

```
[...]
 if ( xIndex < width && yIndex < height ) {
  out[index_out]=block[index_transpose];
}
```

  - ou encore: conditions aux bords de la grille pour un schéma différences finies
- Appelé "warp divergence" (divergence de flot dans un warp)

Divergence de flot dans un warp

Pas de problème a priori mais attention:

- Le coût de l'exécution sera la somme des 2 branches, comme si tous les threads exécutaient les 2 branches
  - Perte de performance potentiellement importante: facteur $\times 32$ dans le pire cas si un thread fait un traitement coûteux tandis que les autres du warp ne font rien
  - Exemple typique: EDP avec des conditions aux limites couteuses à calculer
  - Dans ce cas, utiliser 2 kernels: un pour les points intérieurs, l'autre pour les conditions aux limites
- Pas de barrière de synchronisation `__syncthreads()` dans une conditionnelle: risque d'interblocage (les threads attendent que tous les threads du bloc aient atteint la barrière pour continuer)

En TD, vous avez vu...

Un calcul de π inefficace sur GPU:

- Coût de communication CPU$\leftrightarrow$GPU prohibitif par rapport au calcul. Des solutions possibles:
  - Plus de calcul (augmenter n)
  - Plus de calcul en parallèle et moins de communications: faire des sommes partielles sur le GPU en utilisant saut de pointeur / SCAN
  - Les nouvelles opérations de type `shuffle`
  - Communiquer de façon asynchrone avec le calcul (`cudaMemcpyasync`)
- Partie séquentielle non négligeable restant sur le CPU, cf. loi d'Amdahl, plus tard...
- Il ne suffit pas d'avoir beaucoup de processeurs pour faire quelque chose d'efficace...

Rappel: saut de pointeur

```
for d:=1 to log2(n) do
  forall k:=1 to n in parallel do
    offset := pow(2,d-1);
    if k >= offset then
      x[out][k] := x[in][k-offset]+x[in][k];
    else
      x[out][k] := x[in][k];
  done
  assign(x[out],x[in]);
```

Version basique de saut de pointeur:

- 1 thread par instance k de boucle, dans chaque bloc d'une grille (256,1,1)
- Chaque bloc est limité à des tableaux de 1024 éléments (nombre de threads maxi sur un multi-processeur) - cf. `scan` de la SDK (un peu plus compliqué...)
- Somme des sommes partielles faite ensuite sur le CPU

scan: implémentation CUDA (naive)

(scan_kernel.cu)

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

  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[blockIdx.x*blockDim.x+thid] = temp[pout*n+thid];
}
```


Remarques

Barrière de synchronisation `__syncthreads()`:

- Permet ici de s'assurer que tous les threads ont écrit leur calcul en mémoire partagée avant de l'utiliser
- Puisque les threads ont besoin d'un calcul écrit par un autre thread *du même bloc*

Mémoire partagée allouée dynamiquement:

- 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[];` (`extern` obligatoire)
- Il faut passer, au moment de l'appel au kernel, la taille à allouer en mémoire partagée par multiprocesseur

  `kernel<<<griddim,blockdim,sharedmemsize>>>`

- On n'alloue qu'un bloc mémoire contigu, à charge du programmeur de le découper en les données dont il a besoin...

Implémentation CUDA

(scan.cu)

```
[...]
// initialize the input data on the host
for( unsigned int i = 0; i < num_elements; ++i)
  h_data[i] = floorf(1000*(rand()/(float)RAND_MAX));

// allocate device memory input and output arrays
float* d_idata; float* d_odata;
cudaMalloc( (void**) &d_idata, mem_size);
cudaMalloc( (void**) &d_odata, mem_size);
cudaMemcpy( d_idata, h_data, mem_size, cudaMemcpyHostToDevice);

shared_mem_size = 2*num_threads*sizeof(float);
dim3  grid(256, 1, 1);
dim3  threads(num_threads, 1, 1);
scan<<< grid, threads, shared_mem_size >>> (d_odata, d_idata);
[...] // transfert sur le CPU et somme des sommes partielles
```

Plus généralement: synchronisation en CUDA

- Comme en JAVA, à cause de la mémoire partagée
- Pas de sémaphores/moniteurs ici, mais des fonctions de synchronisation:
  - intra-bloc: `__syncthreads();`
  - globale en passant par le CPU (très inefficace car attend la fin de l'appel au kernel): `cudaThreadSynchronize();` (mais aussi les `cudaMemcpy();` par effet de bord)
  - les opérations "atomiques" (ininterruptibles)
- Les opérations atomiques font des opérations "read-modify-write" sans interférence par d'autre threads
- S'appliquent à la mémoire globale et à la mémoire partagée

Opérations atomiques: exemple

- Supposons que les threads incrémentent un compteur en mémoire partagée

```
__shared__ int compteur;
[...]
if (...) compteur++;
```

- Problème: plusieurs threads, notamment dans le warp, vont vouloir mettre à jour en même temps:

thread 0	thread 1	...	thread 32
read	read	...	read
add	add	...	add
write	write	...	write

- Avec des instructions atomiques, read-add-write devient une instruction unique, et sur une variable partagée elles seront exécutées les unes après les autres par les threads.

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)

Existent aussi sur les float.

- `int atomicExch(int* address, int val);` stocke val dans address
- `int atomicMin[Max](int* address, int val);`
- `unsigned int atomicInc[Dec](unsigned int* address, unsigned int val);` incrémente address de 1 si la valeur stockée précédemment est strictement inférieure à val, sinon met 0 dans address
- `int atomicAnd[Or/Xor](int* address, int val);`

Toutes ces fonctions retournent l'ancienne valeur de address

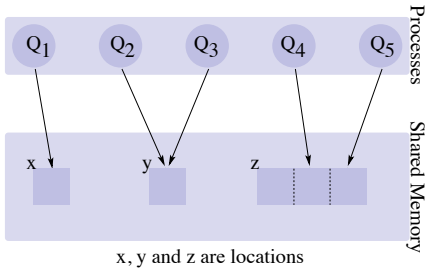
Compare and Swap

Fonction utile: `int atomicCAS(int* address, int compare, int val);`

- Si l'ancienne valeur stockée dans address est égale à compare, alors écrit val à cette adresse: retourne l'ancienne valeur dans tous les cas
- Permet par exemple de coder des sémaphores

```
// variable globale: 0 unlocked, 1 locked
__device__ int lock=0;
__global__ void kernel(...) {
  ...
if (threadIdx.x==0) {
  // prend le verrou
  do {} while(atomicCAS(&lock,0,1));
  ...
  // libère le verrou
  lock = 0;
  }
}
```

Cas particulier de primitives d'exclusion mutuelle



x, y and z are locations

Primitives classiques (Dijkstra 1968...)

- Sémaphore binaires ou mutex, ou verrou: permettant l'accès exclusif à des ressources partagées.
- Généralisation: sémaphores à compteur (verrou pouvant être détenu par jusqu'à n processus mais pas $n+1$).
- Moniteurs: mécanismes permettant notamment de bloquer et reveiller des processus

Une subtilité: visibilité des lectures et écritures

Quels sont les résultats possibles du code suivant?

```
__device__ int x = 0, y = 0;
__device__ void thread1() {
    x = 1; a = y;
}
__device__ void thread2() {
    y = 1;   b = x;
}
```

- Avec le modèle mémoire habituel, *à cohérence séquentielle*:
  - a=0 et b=1, si le thread 1 s'exécute avant le thread 2
  - a=1 et b=0, si le thread 2 s'exécute avant le thread 1
  - a=1 et b=1, si x et y sont tous deux écrits avant d'être lus
- Avec le modèle mémoire "faible" (multi-coeurs, GPUs...), on peut en plus avoir a=0 et b=0: écriture de x et y non bloquants si pas de dépendance avec suite du thread
- Solution: `__threadfence()`: forcer la visibilité de l'écriture

Memory fences

```
__device__ int x = 0, y = 0;
__device__ void thread1() {
    x = 1; __threadfence(); a = y;
}
__device__ void thread2() {
    y = 1;  __threadfence(); b = x;
}
```

- `a = y` ne sera exécuté qu'après que l'écriture `x = 1` sera visible en mémoire à tous
- De même pour le thread 2
- Au moins une écriture sera donc visible avant les affectations `a = y` et `b = x`: on ne peut plus avoir a=0 et b=0:

Memory Fences

- `void __threadfence_block();` attend que toutes les écritures en mémoire globale et partagée soient visibles à tous les threads du bloc
- `void __threadfence();` attend que toutes les écritures en mémoire partagée soient visibles à tous les threads du bloc; et que toutes les écritures en mémoire globale soient visibles à tous les threads
- `void __threadfence_system();` si plusieurs cartes

Memory Fences vs `__syncthreads()`

- `__syncthreads()` ne synchronise que dans un bloc, tandis qu'appliqué à une variable en mémoire globale, `__threadfence()` est global
- dans un bloc, `__threadfence()` est plus faible que `__syncthreads()`: il ne synchronise pas tous les threads, et il n'est pas nécessaire que tous les threads atteignent l'instruction (contrairement à `__syncthreads()` qui ne peut donc pas être utilisé à l'intérieur d'une branche d'un `if`)

Retour à l'exemple du sémaphore

Attendre la visibilité des écritures avant de relâcher le verrou

```
// variable globale: 0 unlocked, 1 locked
__device__ int lock=0;
__global__ void kernel(...) {
  ...
if (threadIdx.x==0) {
  // prend le verrou
  do {} while(atomicCAS(&lock,0,1));
  ...
  __threadfence(); // attendre fin des écritures

  // libère le verrou
  lock = 0;
  }
}
```

Remarque: noter le `__threadfence()` dans une conditionnelle ici

Exemple d'utilisation dans le calcul de π

```
__device__ unsigned int count = 0;
__device__ float calculatePartialSum(const float *array) {
  [..] // saut de pointeur dans le bloc (sur blockDim.x thread)
}
__device__ float calculateTotalSum(float *result) {
  [...] // saut de pointeur sur BLOCKSIZE<blockDim.x thread
}
__global__ void sum(const float* array, float* result) {
  __shared__ bool isLastBlockDone;
  // Chaque bloc calcule une somme partielle de array
  float partialSum = calculatePartialSum(array, blockDim.x);
  if (threadIdx.x == 0) {
    // Thread 0 de chaque bloc stocke la somme partielle en mem gl
    result[blockIdx.x] = partialSum;
    __threadfence(); // rendre visible à tous les blocs
    // Thread 0 de chaque bloc signale qu'il a effectue le calcul
    unsigned int value = atomicInc(&count, gridDim.x);
    // Thread 0 de chaque bloc determine si son bloc est le dernie
    isLastBlockDone = (value == (gridDim.x - 1));
  }
```

(rappel: `atomicInc(address,val)` incrémente `address` de 1 si la valeur stockée précédemment est inférieure à `val`, sinon met 0 dans `address`, et retourne l'ancienne valeur dans `address`)

Exemple d'utilisation: suite

```
  // Pour que chaque thread lise la valeur correcte de isLastBlockD
  __syncthreads();
  if (isLastBlockDone) {
    // Dernier bloc somme les sommes dans result[0 .. gridDim.x-1]
    float totalSum = calculateTotalSum(result);
    if (threadIdx.x == 0) {
      // Thread 0 du dernier bloc stocke la somme globale en mem g
      // et reinitialie count pour une nouvelle execution du noyau
      result[0] = totalSum;
      count = 0;
    }
  }
}
```

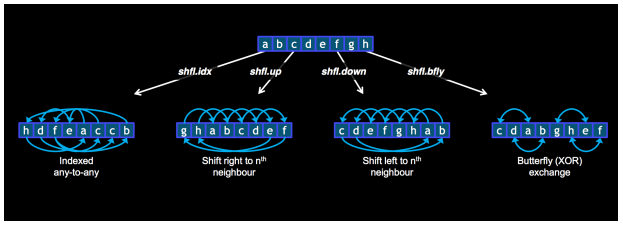
Pour aller plus loin: warp shuffle (compute cap >= 3.x)

- Nouvelle opération introduite par l'architecture Kepler
- Pouvoir échanger des données entre threads d'un warp sans passer par la mémoire locale (ou globale)
- Les threads lisent des données (de 32 bits, float ou int) d'un autre thread seulement explicitement défini comme participant au shuffle (sinon comportement indéfini)

La position des threads d'un warp est donnée par un laneID entre 0 et warpSize-1 (`threadIdx.x%32` pour des blocs 1D). Quatre variantes: `__shfl()`, `__shfl_up()`, `__shfl_down()`, `__shfl_xor()`

Warp shuffles (cf documentation aussi)

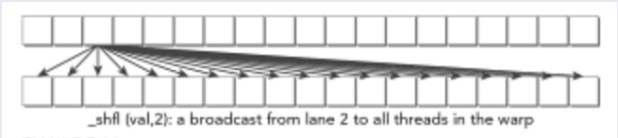
- `int __shfl(int var, int srcLane);` retourne la valeur du registre local `var` du thread de laneID `srcLane`
- `int __shfl_up(int var, unsigned int delta);` retourne la valeur de var du thread `delta` threads à droite du courant (retourne sa valeur locale si ce thread n'existe pas)
- `int __shfl_down(int var, unsigned int delta);` idem
- `int __shfl_xor(int var, int laneMask);` XOR bit à bit entre laneMask et le laneID du thread courant donne la lane dont on copiera la valeur



Exemples d'utilisation des warp shuffles

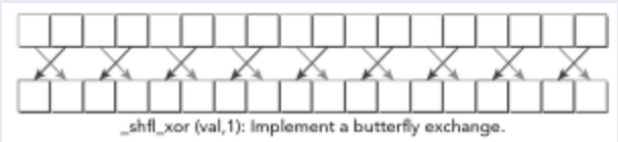
Broadcast d'une valeur

```
val = __shfl(val, 2); // retourne "val" de la lane 2
```



Echange d'une valeur (butterfly)

```
val = __shfl_xor(val, 1); // retourne "val" de la lane (cur XOR 1)
```



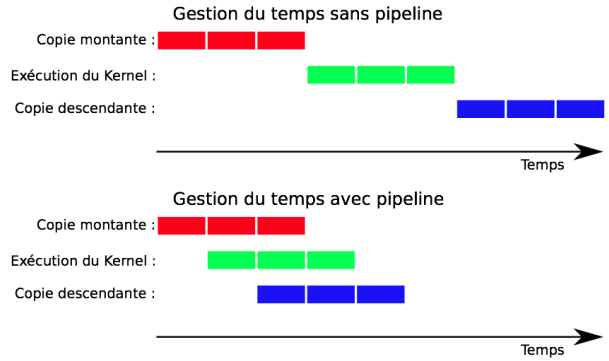
Utilisation pour optimiser la somme des sommes partielles (PI)

Deux façons de faire la somme de tous les éléments d'un warp:

- ```
  for (int i=1; i<=16; i=i*2) {
    value += __shfl_up(value, i);
     if (laneId >= i)
       value += n;
  }
  ```

- ```
  for (int i=1; i<=16; i*=2)
    value += __shfl_xor(value, i);
  ```

Communication asynchrone hôte-carte



- Le GPU peut opérer les transferts mémoire et le calcul de façon concurrente.
- Implémentés par les streams (flux): permettent de s'assurer de l'ordre dans lequel on lancera la copie des données, le lancement du kernel, et la recopie vers l'hôte.

(à compiler avec `nvcc -arch sm_30`, "CUDA capability 3.0")

Transferts de données hôte-carte

- Jusqu'ici les transferts entre hôte et carte utilisant `cudaMemcpy()` étaient bloquants
- Il y a une version non bloquante `cudaMemcpyAsync` (qui rend la main avant d'avoir fini):

```
cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, 0);
kernel<<<grid,block>>>(a_d);
cpuFunction();
```

  - le CPU n'attend pas la fin de la copie de mémoire ni du kernel pour exécuter sa fonction `cpuFunction()`
  - par contre, le kernel n'est lancé qu'après fin de la copie de la mémoire

Remarques:

- La mémoire sur l'hôte doit être allouée par `cudaHostAlloc()`, et l'appel `cudaMemcpyAsync` contient un argument supplémentaire, un `stream ID` qui est 0 ici
- Ici pas encore de recouvrement entre communication et calcul sur le GPU (juste entre CPU et GPU)

Recouvrement transfert mémoire avec une exécution sur le GPU

- Recouvrement transfert mémoire / calcul possible quand le transfert mémoire se fait sur une partie de mémoire non utilisée par le kernel
- Obstacle: sur le device, une opération ou appel de fonction n'est pas exécutée avant que le précédent soit terminé
- Remède: les CUDA streams permettent d'introduire de la concurrence
  - un CUDA stream = une queue d'opérations à effectuer sur le GPU
  - ces opérations seront efffectuées dans l'ordre ou elles sont introduites dans le stream (et chacune attend terminaison de la précédente)
  - l'hôte peut définir plusieurs streams
  - les streams peuvent être synchronisés par des barrières de synchronisation dans le code CPU, mais pas de synchronisation possible inter-streams depuis l'intérieur des kernels

Execution concurrente de kernels

En résumé, les streams permettent:

- Le recouvrement transfert mémoire / calcul
- Mais aussi jusqu'à 32 (pour Kepler) streams / kernels indépendants exécutés en parallèle sur la carte
- Une façon pour le GPU de se rapprocher d'un modèle MIMD

Exemple utilisant les streams

```
// créer les streams
cudaStream_t stream[2];
for (int i = 0; i <= 1; i++)
  cudaStreamCreate(&stream[i]);

// allocation mémoire hôte
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);

for (int i = 0; i <= 1; i++) {
  cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
        size, cudaMemcpyHostToDevice, stream[i]);
  MyKernel<<<100, 512, 0, stream[i]>>>(outputDevPtr + i * size,
        inputDevPtr + i * size, size);
  cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
        size, cudaMemcpyDeviceToHost, stream[i]);
}

// Libérer
cudaStreamDestroy(stream[0]);
cudaStreamDestroy(stream[1]);
```


Quelques fonctions de synchronisation sur les streams

Depuis le CPU:

- `cudaDeviceSynchronize()`: attend que toutes les instructions précédentes de tous les streams soient terminées
- `cudaStreamSynchronize()`: attend que toutes les sinstructions précédentes du stream en paramètre soient terminés (les autres streams continuent sur la carte)
- `cudaStreamWaitEvent()`: prend un stream et un évênement (event) en paramètres et fait attendre toutes les instructions ajoutées au stream après cet appel que l'évênement ait été exécuté. Si le stream est 0, la synchronisation est globale à tous les streams

(voir la documentation)

CUDA 5.0 and later - CUDA capability 3.5

Sur les cartes les plus récentes (Tesla), on a aussi:

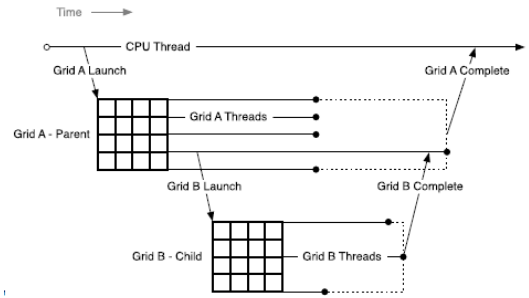
- Communication directe entre la mémoire de plusieurs device (et espace d'adressage virtuel pour les devices, et le host)
- Parallélisme dynamique: création dynamique de processus, avec des grilles propres (par exemple raffinement adaptatif de maillage etc)

```
__global__ void child_launch(int *data) {
   data[threadIdx.x] = data[threadIdx.x]+1;
}

__global__ void parent_launch(int *data) {
   data[threadIdx.x] = threadIdx.x;
   __syncthreads();
   if (threadIdx.x == 0) {
       child_launch<<< 1, 256 >>>(data);
       cudaDeviceSynchronize(); }
   __syncthreads(); }

void host_launch(int *data) {
    parent_launch<<< 1, 256 >>>(data); }
```


CUDA 5.5/6.0 - CUDA capability 3.5



Pour expérimenter pour les projets en CUDA:

- Machine du LIX: 2 device chacun de 2496 coeurs
- Accessible depuis les salles TD à l'IP 129.104.252.237
- M'envoyer un mail pour avoir accès (ensuite ce sera login et mot de passe LDAP)
- Mais faire la mise au point en salle machine

Profiling et debug (cf documentation en ligne)

- NVIDIA visual profiler: intégré sous `nsight`, ou le visual profiler seul `nvpp`, ou encore en ligne de commande `nvprof`: collecte des infos sur
  - transferts mémoire
  - occupation des processeurs (kernels), efficacité du calcul et des transferts mémoire
- `cuda-memcheck`: compiler avec options `nvcc -g -G` pour obtenir l'exécutable `mon_exec_cuda`, puis lancer `cuda-memcheck mon_exec_cuda`, ou encore `cuda-memcheck -tool racecheck mon_exec_cuda` pour afficher aussi les éventuels conflits d'accès en mémoire partagée.

Quelques librairies basées sur CUDA

- CUBLAS: algèbre linéaire (Basic Linear Algebra Subroutines) pour des matrices denses
- CUFFT: Fast Fourier Transform
- CUSPARSE: routines pour des matrices creuses
- NPP: imagerie
- CURAND: génération de nombres aléatoires
- MAGMA: LAPACK pour GPU (construit sur CUBLAS)
- Thrust: librairie C++ haut-niveau qui abstrait 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

Les projets: commencer à réfléchir au sujet

- Notation du cours sur projets, en monomes on binomes
- Programmation au choix Cuda ou Java/RMI
- Des propositions de sujets (cf page web du cours pour des description des suggestions)
  - Simulation de modèles atmosphériques sur GPU (cf présentation T Dubos en TD)
  - Campagne sismique (vieux sujet, propagation d'ondes et éventuellement problème inverse)
  - Calcul de distance d'édition entre des séquences ADN
  - Résolution du problème SAT
- Ou un sujet de votre choix (à valider avec moi; possibilité éventuelle de parallélisation d'un projet réalisé par ailleurs)
  - machine learning (suggestions de F. Nielsen): transport optimal sur Cuda, ou k plus proches voisins (plus classique), etc
  - simulation: systèmes n-corps etc
  - applications pour le graphique
  - jeux
  - algèbre linéaire (des cas pas traités par les librairies existantes?)
  - etc

La suite

- TD3 Salle Info 32 à 10h15: présentation du projet par T. Dubos (LMD) "Simulation de modèles atmosphériques" + suite et fin du TD3-4 Cuda
- Cours 5 en Amphi Lagarrigue lundi 19 janvier: synchronisation Java et RMI