Faculté des Sciences & Techniques — Université de Limoges

Master 1ère année

Dév. GPGPU

Corrections TD n°1

Programmation GPGPU & CUDA

Les notions de «*threads*», «*blocks*» et de «*grille*»

1 – Soit le source suivant :

a. À quoi sert `blockIdx.x` ? Comment est-il défini ?
Il sert à identifier le bloc dans la grille suivant la dimension x de la grille : chaque bloc est numéroté de 0 *à* $N-1$. *cette variable est définie automatiquement par le «runtime», c-à-d au moment de l'exécution de la thread.*

b. À quoi sert la ligne 5 `if (tid < N)` ?
Il sert à faire du contrôle d'erreur pour éviter qu'une thread ne travaille sur une case de tableau non définie.
Cela pourrait arriver si on lance plus de threads qu'il n'y a de cases à traiter.

c. Que fait le programme ?
- *un «kernel» est défini de la ligne 3 à la ligne 7 :*
  - *il correspond au travail réalisé par une seule thread :*
    - *additionne le contenu de deux cases d'indice correspondant au numéro du bloc auquel appartient la thread*
    - *range le résultat dans la case de même indice d'un* $3^{ème}$ *tableau ;*
  - *le programmeur a choisi*
    - *d'associer une thread par case des tableaux à traiter ;*
    - *de répartir chacune de ces threads dans un bloc différent.*
- *ligne 25 : on définit une grille suivant une seule dimension de N blocs. Chacun de ces blocs contient une seule thread ;*
- ligne 13 à 15 : on réserve de l'espace sur la carte graphique répartis en trois zones et pour chacune d'elles, on récupère une référence, c-à-d l'adresse de la zone allouée ;
- ligne 17 à 21 : initialisation des valeurs des tableaux source ;
- ligne 23 à 24 : on copie le contenu des tableaux sources vers les zones mémoires réservées sur la carte graphique ;
- ligne 25 : on lance l'exécution du kernel sur une grille définie comme un vecteur de N blocs, chaque bloc contenant une thread ;
- ligne 28 : on copie le contenu du tableau résultat de la carte graphique dans le tableau résultat de l'hôte ;
- ligne 30 à 32 : on affiche le contenu du tableau résultat ;
- ligne 34 à 36 : on libère la mémoire allouée sur la carte graphique.

d. Que se passe-t-il si on lance le *kernel* avec l'instruction suivante :

```
25    add<<<1,N>>>( dev_a, dev_b, dev_c );
```

On définit maintenant une grille contenant un seul bloc de N threads.

Est-ce qu'il faut modifier le code du *kernel* ?
Oui, il faut modifier l'association «case de tableau» ⟺ «thread» en modifiant l'identifiant utilisé par une thread pour non plus utiliser le numéro du bloc (il n'y en a qu'un), mais utiliser le numéro de la thread dans le bloc :

```
4    int tid = threadIdx.x ;
```

Est-ce qu'il y a des limitations au nombre de threads par block ? *On peut lancer au plus 512 threads par bloc suivant l'architecture des cartes disponibles dans les salles de TP.*

2 – Questions :

a. Que se passe-t-il si on veut faire la somme de vecteurs dont la taille est > 512 ? > 65535 ? *Comme on ne peut allouer plus de 512 threads par bloc, il faut créer plus d'un bloc.*
Ces différents blocs sont combinés en une grille où chaque dimension < 65535.

b. Soit la formule :
$add \lll (N + 127)/128, 128 \ggg (dev\_a, dev\_b, dev\_c)$;
Que permet-elle de faire ? *Elle permet de faire une division entière avec arrondi à la valeur supérieure sans utiliser la fonction d'arrondi, ce qui est util lorsque l'on veut définir dynamiquement la taille de la grille en fonction de la taille des données à traiter.*

c. À quoi correspond l'expression : `int tid = threadIdx.x + blockIdx.x * blockDim.x;` ?
À calculer le numéro global de la thread dans la grille.
Par exemple, si « `blockIdx.x` *» vaut 2 et «* `threadIdx.x` *» vaut 3 avec une taille de bloc, «* `blockDim.x` *», de 10 threads, alors la thread courante est celle de numéro 23 en partant de zéro (le numéro de bloc, comme celui des threads commence à zéro).*

d. et l'expression : `blockDim.x * gridDim.x` ?
Le nombre de threads/bloc par le nombre de bloc : le nombre total de threads définies dans la grille.

```
 1 #define N 32768
 2 __global__ void add(int *a,int *b,int *c)
 3 { int tid = threadIdx.x
 4           + blockIdx.x * blockDim.x;
 5 while (tid < N) {
 6   c[tid] = a[tid] + b[tid];
 7   tid += blockDim.x * gridDim.x; }
 8 }
 9 int main( void )
10 {
11    ...
```

```
24 add<<<128,128>>>( dev_a, dev_b, dev_c );
25 /* copie du tableau c du GPU sur le CPU */
26 cudaMemcpy(c,dev_c,N*sizeof(int),
27               cudaMemcpyDeviceToHost);
28 for (int i=0; i<N; i++) {
29   printf("%d + %d = %d\n",a[i],b[i],c[i]);
30 }
31 /* liberer la memoire allouee sur GPU */
32 cudaFree( dev_a );
33 cudaFree( dev_b );
34 cudaFree( dev_c );
35 return 0;
36 }
```

e. Que fait le programme ?
Il fait la somme de deux tableaux a et b dans le tableau c.

f. À quoi sert la ligne 6 ?
Elle permet de traiter des tableaux dont la dimension est supérieure à celle de la grille :
- *la grille est définie suivant une seule direction, x, comme le tableau sur lequel elle travaille ;*
- *chaque thread s'occupe que d'une case par «front» de threads, c-à-d par dimension de grille : par exemple, si la grille définie un «front» de 100 threads et le tableau fait 300 cases, alors chaque thread traitera 3 cases et la thread 0 traitera la case 0, puis la case 100, et enfin la case 200.*

g. Comment va se dérouler l'exécution suivant la grille définie en ligne 24 ?
La grille définie un «front» de threads de $128 * 128 = 16384$, *chaque thread traitera donc 2 cases du tableau de 32768 cases.*

Mémoire partagée et synchronisation

3 – a. Donnez la taille de la grille pour $N = 33 * 1024$ et une taille de bloc de 256 threads.
On peut utiliser la formule de la division entière avec arrondi à la valeur supérieure :
$taille\_grille = (33 * 1024 + 255)/256 = 132$ *blocs.*

b. Voici une première version du *kernel* pour faire l'opération :

```
1 __global__ void mult(int *a,int *b,int *c)
2 { int tid = threadIdx.x
3             + blockIdx.x * blockDim.x;
4   while (tid < N) {
5     c[tid] = a[tid] * b[tid];
6     tid += blockDim.x * gridDim.x;
7   }
8 }
```

Que reste-t-il à faire ?
Il reste à faire la somme de toutes les cases du tableau c.
Est-il possible de le faire sur le GPU ?
Oui.
Comment s'y prendre de manière efficace ?
Il faut trouver une manière de répartir le travail entre différentes threads en faisant attention :
- *aux accès concurrents aux tableaux a, b et c ;*
- *à déterminer correctement la correspondance entre l'indice d'accès aux cases des tableaux et l'identifiant de la thread.*

Une première proposition «naïve» pourrait être de faire travailler une seule thread sur toutes les cases du tableau : plus d'utilisation du parallèlisme.

Une seconde proposition serait de faire travailler une thread par bloc sur une sous-somme et ensuite de faire la somme globale de toutes ces sous-sommes éventuellement sur le CPU. Pour améliorer les performances, le calcul des « produits » nécessaires au calcul de la sous-somme pourrait être fait dans la mémoire partagée associée au bloc, ce qui éliminerait les accès à la mémoire globale contenant le tableau c pour le stockage de chaque produit calculé.

c. Voici une seconde proposition :

```
1  __global__ void dot(float *a,float *b,float *c)
2  { __shared__ float cache[threadsPerBlock];
3    int tid = threadIdx.x
4                + blockIdx.x * blockDim.x;
5    int cacheIndex = threadIdx.x;
6    float temp = 0; while (tid < N) {
7      temp += a[tid] * b[tid];
8      tid += blockDim.x * gridDim.x;
9    }
10   // set the cache values
11   cache[cacheIndex] = temp;
12 }
```

Que reste-t-il à réaliser ?
Ici, on utilise un tableau appelé « `cache` » qui utilise la mémoire partagée utilisable dans un bloc (cette mémoire est d'accès partagée uniquement à l'ensemble des threads du même bloc). Dans ce cache, chaque thread réalise la somme des valeurs qu'elle doit traiter lorsque le tableau de données est de taille supérieure à celle du « front » de threads défini par la grille.
Dans cette proposition, on s'arrête on ayant la somme intermédaire uniquement dans le tableau « `cache` » dans la mémoire partagée du bloc.
Il reste à sauvegarder le résultat obtenu par chaque thread de la mémoire partagée du bloc vers la mémoire de la carte graphique.

Comment le faire ? Dans le GPU ?
Il faudrait utiliser certaines des threads afin d'obtenir des résultats intermédiaires : faire le calcul de la sous-somme par bloc. Ensuite, on peut choisir de faire la somme de toutes les sous-sommes soit dans le CPU, soit dans le GPU. Enfin, il faut récupérer le ou les résultats du GPU dans le CPU.

Doit-on prendre des précautions ?
Il faut faire attention aux accès concurrents et aux contraintes de l'utilisation de la mémoire associée à chaque bloc : il faut s'assurer que toutes les threads d'un bloc ont fini avant d'exploiter le résultat, c-à-d utiliser la ***synchronisation****.*

Synchronisation en CUDA

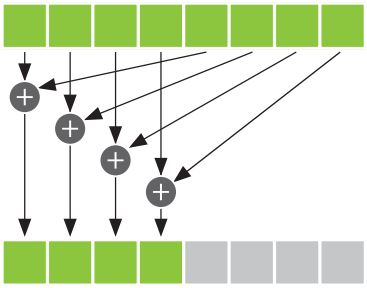
En CUDA, la synchronisation n'est possible qu'au niveau d'un bloc.
Explication : le traitement SPMT, « **Simple Program Multiple Threads** *», se fait par « WARP » de 32 threads matérielles et le GPU doit ordonnancer l'exécution de toutes les warps d'un même bloc (scheduling et multi-tâche). Il faut donc attendre que toutes les threads d'un même bloc aient terminé, avant de passer à un traitement utlisant leur résultat global.*

Cette opération ressemble-t-elle à une opération « courante » du parallélisme ?
Oui, à l'opération de ***réduction****.*

```
1  #define threadsPerBlock 256
2  #define N  33792
3  __global__ void dot( float *a, float *b, float *c ) {
4     __shared__ float cache[threadsPerBlock];
5     int tid = threadIdx.x + blockIdx.x * blockDim.x;
6     int cacheIndex = threadIdx.x;
7     float temp = 0; while (tid < N) {
8       temp += a[tid] * b[tid];
9       tid += blockDim.x * gridDim.x;  }
10    cache[cacheIndex] = temp;        // set the cache values
11    __syncthreads();
12    if (threadIdx.x == 0)
13    { temp = 0;
14       for(int i=0;i<threadsPerBlock;i++)
15          temp += cache[i];
16       c[blockIdx.x] = temp;  }
17 }
```

Ici, seul la thread$_0$ de chaque bloc travaille $\Longrightarrow$ pas de parallélisme !

d. Donnez une version complète de la proposition 2 la plus efficace possible ?



On pourrait faire la sous-somme d'un bloc en essayant de maximiser à chaque étape le nombre de threads qui travaillent :

- *chaque thread fait la somme de deux valeurs et seulement la moitié des threads travaillent à l'étape 1 ;*
- *puis on recommence à l'étape 2, etc.*
- *jusqu'à n'avoir qu'une seule thread qui travaille et stocke dans le tableau c la sous-somme associée au bloc.*

L'avantage est que chaque thread d'un même warp demande une case de la moitié gauche, puis une case de la moitié droite : le warp demande un ensemble de cases contigües pour la partie gauche puis pour la partie droite, ce qui améliore les échanges mémoire sur le bus de données qui est de taille multiple de la taille d'une case : plus de vitesse au lieu de threads en attente d'accès mémoire irrégulier.

Quelle est la complexité de ce travail ?

Elle est en $log_2(n)$.

Est-il possible de finaliser tout le traitement dans le GPU et pourquoi ?

Non il faut faire la somme des valeurs de blocks sur le CPU, ce sera plus rapide car on ne pourrait faire travailler qu'une seule thread.

e. Soient le code suivant :

```
1 int i = blockDim.x/2;
2 while (i != 0) {
3    if (cacheIndex < i)
4       cache[cacheIndex] += cache[cacheIndex + i];
5    __syncthreads();
6    i /= 2; }
```

et le code :

```
1 int i = blockDim.x/2;
2 while (i != 0) {
3    if (cacheIndex < i) {
4       cache[cacheIndex] += cache[cacheIndex + i];
5       __syncthreads();
6       }
7    i /= 2; }
```

Quelle(s) différence(s) ?

Dans la version 1, l'opération de synchronisation « `__syncthreads()` » est réalisée par toutes les threads alors que dans la version 2 elle n'est réalisée que par les threads qui travaillent.

Les deux versions sont-elles correctes ?

*Non, la deuxième version est **incorrecte** : on est devant un problème de « **divergence** » :*

- *toutes les threads ne font pas la même chose ce qui est **grave** dans le cas de la synchronisation : pour terminer la « barrière de synchronisation », il faut que **toutes** les threads d'un même bloc l'atteignent, c-à-d exécutent le « `__syncthreads()` ».*
- *ici, ce n'est pas le cas à cause de la condition : le travail du kernel se bloque (ou pas, mais le résultat est indéfini).*

Cuda dispose de mécanisme d'arrêt automatique de l'exécution de kernel qui prend trop de temps à s'exécuter sans tenir compte du résultat : il sera faux et imprévisible.

```
1  __global__ void dot( float *a, float *b, float *c ) {
2      __shared__ float cache[threadsPerBlock];
3      int tid = threadIdx.x + blockIdx.x * blockDim.x;
4      int cacheIndex = threadIdx.x;
5      float temp = 0; while (tid < N) {
6         temp += a[tid] * b[tid];
7         tid += blockDim.x * gridDim.x;
8     }
9     cache[cacheIndex] = temp;
10     __syncthreads();
11     int i = blockDim.x/2;
12     while (i != 0) { if (cacheIndex < i)
13                         cache[cacheIndex] += cache[cacheIndex + i];
14                      __syncthreads();
15                      i /= 2; }
16     if (threadIdx.x == 0)  c[blockIdx.x] = cache[threadIdx.x];
17 }
```

une seule thread par bloc