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.*