Faculté des Sciences & Techniques — Université de Limoges

Master 1ère année

Dév. GPGPU

TD n°1

Programmation GPGPU & CUDA

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

1 – a. Expliquez comment passer d'un tableau à deux dimensions à un tableau à une dimension ?

b. Soit un tableau de 100 éléments.
Si chaque *thread* CUDA accède à une case différente de ce tableau dans le code du « *kernel* » exécuté, comment l'accès à la mémoire va-t-il être fait ?

2 – Soit le source suivant :

```
 1 #define N 10
 2
 3 __global__ void add( int *a, int *b, int *c ) {
 4    int tid = blockIdx.x; /* handle the data at this index */
 5 if (tid < N)
 6    c[tid] = a[tid] + b[tid];
 7 }
 8
 9 int main( void ) {
10      int a[N], b[N], c[N];
11      int *dev_a, *dev_b, *dev_c;
12      /* allocation de la memoire sur le GPU */
13      cudaMalloc( (void**)&dev_a, N * sizeof(int) ) ;
14      cudaMalloc( (void**)&dev_b, N * sizeof(int) ) ;
15      cudaMalloc( (void**)&dev_c, N * sizeof(int) ) ;
16      /* fill the arrays 'a' and 'b' on the CPU */
17      for (int i=0; i<N; i++)
18      {
19              a[i] = -i;
20              b[i] = i * i;
21      }
22      /* copie des tableaux a et b sur le GPU */
23      cudaMemcpy( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice );
24      cudaMemcpy( dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice );
25      add<<<N,1>>>( dev_a, dev_b, dev_c );
26
27      /* copie du tableau c depuis le GPU sur le CPU */
28      cudaMemcpy( c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost );
29      /* display the results */
30      for (int i=0; i<N; i++) {
31              printf( "%d + %d = %d\n", a[i], b[i], c[i] );
32      }
33      /* liberer la memoire allouee sur le GPU */
34      cudaFree( dev_a );
35      cudaFree( dev_b );
36      cudaFree( dev_c );
37      return 0;
38 }
```

a. À quoi sert `blockIdx.x` ? Comment est-il défini ?

b. À quoi sert le ligne 5 `if (tid < N)` ?

c. Que fait le programme ? *Décrivez le travail en terme de threads, de blocks et de grilles.*

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 );
```

Est-ce qu'il faut modifier le code du *kernel* ?
Est-ce qu'il y a des limitations au nombre de threads par block ?

3 – Questions :

a. Que se passe-t-il si on veut faire la somme de vecteurs dont la taille est > 512 ? > 65535 ?

b. Soit la formule :
$add \lll (N + 127)/128, 128 \ggg (dev\_a, dev\_b, dev\_c)$;
Que permet-elle de faire ?

c. À quoi correspond l'expression : `int tid = threadIdx.x + blockIdx.x * blockDim.x;` ?

d. et l'expression : `blockDim.x * gridDim.x`

```
 1 #define N 32768
 2 __global__ void add( int *a, int *b, int *c ) {
 3    int tid = threadIdx.x + blockIdx.x * blockDim.x;
 4    while (tid < N) {
 5                c[tid] = a[tid] + b[tid];
 6                tid += blockDim.x * gridDim.x;
 7          }
 8 }
 9 int main( void )
10 {
11    int a[N], b[N], c[N];
12    int *dev_a, *dev_b, *dev_c;
13     /* allocation de la memoire sur le GPU */
14    cudaMalloc( (void**)&dev_a, N * sizeof(int) );
15    cudaMalloc( (void**)&dev_b, N * sizeof(int) );
16    cudaMalloc( (void**)&dev_c, N * sizeof(int) );
17     /* remplissage des tableaux a et b sur le CPU */
18    for (int i=0; i<N; i++)
19    {    a[i] = i;
20       b[i] = i * i; }
21     /* copie des tableaux a et b sur le GPU */
22    cudaMemcpy( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice );
23    cudaMemcpy( dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice );
24     add<<<128,128>>>( dev_a, dev_b, dev_c );
25     /* copie du tableau c depuis le GPU sur le CPU */
26     cudaMemcpy( c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost );
27
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 le GPU */
32    cudaFree( dev_a );
33    cudaFree( dev_b );
34    cudaFree( dev_c );
35    return 0;
36 }
```

d. Que fait le programme ?

e. À quoi sert la ligne 6 ?

f. Comment va se dérouler l'exécution suivant la grille définie en ligne 24 ?

▬▬▬ Mémoire partagée et synchronisation

4 – Soit le produit scalaire de deux vecteurs :

$$(x_1, x_2, x_3, x_4).(y_1, y_2, y_3, y_4) = x_1y_1 + x_2y_2 + x_3y_3 + x_4y_4$$

a. Donnez la taille de la grille pour une taille de données de $33 * 1024$ et une taille de block de 256 threads.

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 + blockIdx.x * blockDim.x;
3    while (tid < N) {
4                c[tid] = a[tid] * b[tid];
5                tid += blockDim.x * gridDim.x;
6          }
7 }
```

Que reste-t-il à faire ?
Est-il possible de le faire sur le GPU ?
Comment s'y prendre de manière efficace ?

c. Voici une seconde proposition :

```
__global__ void dot( float *a, float *b, float *c ) {
   __shared__ float cache[threadsPerBlock];
   int tid = threadIdx.x + blockIdx.x * blockDim.x;
   int cacheIndex = threadIdx.x;
   float temp = 0;
   while (tid < N) {
            temp += a[tid] * b[tid];
            tid += blockDim.x * gridDim.x;
       }
       // set the cache values
       cache[cacheIndex] = temp;
   }
```

Que reste-t-il à réaliser ?
Comment le faire ? Dans le GPU ?
Doit-on prendre des précautions ?
Cette opération ressemble-t-elle à une opération « courante » du parallélisme ?

d. Donnez une version complète de la proposition 2 la plus efficace possible ?
Quelle est la complexité de ce travail ?
Est-il possible de finaliser tout le traitement dans le GPU et pourquoi ?

e. Soient le code suivant :

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

et le code :

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

Quelle(s) différence(s) ?
Les deux versions sont-elles correctes ?

```
const int N = 33 * 1024;
const int threadsPerBlock = 256;
const int blocksPerGrid = (N+threadsPerBlock-1) / threadsPerBlock;
__global__ void dot( float *a, float *b, float *c ) {
   __shared__ float cache[threadsPerBlock];
   int tid = threadIdx.x + blockIdx.x * blockDim.x;
   int cacheIndex = threadIdx.x;
   float temp = 0; while (tid < N) {
            temp += a[tid] * b[tid];
            tid += blockDim.x * gridDim.x;
       }
       // set the cache values
       cache[cacheIndex] = temp;
       // synchronize threads in this block
      __syncthreads();
       // for reductions, threadsPerBlock must be a power of 2 because of the following code
       int i = blockDim.x/2;
       while (i != 0) {
          if (cacheIndex < i)
             cache[cacheIndex] += cache[cacheIndex + i];
          __syncthreads();
          i /= 2;
       }
       if (cacheIndex == 0) c[blockIdx.x] = cache[0];
   }
int main( void ) {
   float *a, *b, c, *partial_c;
   float *dev_a, *dev_b, *dev_partial_c;
   // allocate memory on the CPU side
   a = (float*)malloc( N*sizeof(float) );
   b = (float*)malloc( N*sizeof(float) );
   partial_c = (float*)malloc( blocksPerGrid*sizeof(float) );
   // allocate the memory on the GPU
   cudaMalloc( (void**)&dev_a, N*sizeof(float) );
   cudaMalloc( (void**)&dev_b, N*sizeof(float) );
   cudaMalloc( (void**)&dev_partial_c, blocksPerGrid*sizeof(float) );
   // fill in the host memory with data
   for (int i=0; i<N; i++) {
      a[i] = i;
      b[i] = i*2; }
   // copy the arrays 'a' and 'b' to the GPU
   cudaMemcpy( dev_a, a, N*sizeof(float), cudaMemcpyHostToDevice );
   cudaMemcpy( dev_b, b, N*sizeof(float), cudaMemcpyHostToDevice );

   dot<<<blocksPerGrid,threadsPerBlock>>>( dev_a, dev_b, dev_partial_c );

   // copy the array 'c' back from the GPU to the CPU
   cudaMemcpy( partial_c, dev_partial_c, blocksPerGrid*sizeof(float), cudaMemcpyDeviceToHost );
   // finish up on the CPU side
   c = 0;
   for (int i=0; i<blocksPerGrid; i++) {
               c += partial_c[i];
         }
   printf("Result\,: %f\n", c);
   // free memory on the GPU side
   cudaFree( dev_a );
   cudaFree( dev_b );
   cudaFree( dev_partial_c );
   // free memory on the CPU side
   free( a );
   free( b );
   free( partial_c );
}
```