

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     {
20         a[i] = i;
21         b[i] = i * i;
22     }
23     /* copie des tableaux a et b sur le GPU */
24     cudaMemcpy( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice );
25     cudaMemcpy( dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice );
26     add<<<128,128>>>( dev_a, dev_b, dev_c );
27     /* copie du tableau c depuis le GPU sur le CPU */
28     cudaMemcpy( c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost );
29     for (int i=0; i<N; i++) {
30         printf( "%d + %d = %d\n", a[i], b[i], c[i] );
31     }
32     /* liberer la memoire allouee sur le GPU */
33     cudaFree( dev_a );
34     cudaFree( dev_b );
35     cudaFree( dev_c );
36     return 0;
37 }
```

- 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 :

$$(x_1, x_2, x_3, x_4) \cdot (y_1, y_2, y_3, y_4) = x_1 y_1 + x_2 y_2 + x_3 y_3 + x_4 y_4$$

- 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 :

```
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;
6     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 ?

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 :

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

et le code :

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

Quelle(s) différence(s) ?

Les deux versions sont-elles correctes ?

```

1 const int N = 33 * 1024;
2 const int threadsPerBlock = 256;
3 const int blocksPerGrid = (N+threadsPerBlock-1) / threadsPerBlock;
4 __global__ void dot( float *a, float *b, float *c ) {
5     __shared__ float cache[threadsPerBlock];
6     int tid = threadIdx.x + blockIdx.x * blockDim.x;
7     int cacheIndex = threadIdx.x;
8     float temp = 0; while (tid < N) {
9         temp += a[tid] * b[tid];
10        tid += blockDim.x * gridDim.x;
11    }
12    // set the cache values
13    cache[cacheIndex] = temp;
14    // synchronize threads in this block
15    __syncthreads();
16    // for reductions, threadsPerBlock must be a power of 2 because of
    the following code
17    int i = blockDim.x/2;
18    while (i != 0) {
19        if (cacheIndex < i)
20            cache[cacheIndex] += cache[cacheIndex + i];
21        __syncthreads();
22        i /= 2;
23    }
24    if (cacheIndex == 0) c[blockIdx.x] = cache[0];
25 }
26 int main( void ) {
27     float *a, *b, c, *partial_c;
28     float *dev_a, *dev_b, *dev_partial_c;
29     // allocate memory on the CPU side
30     a = (float*)malloc( N*sizeof(float) );
31     b = (float*)malloc( N*sizeof(float) );
32     partial_c = (float*)malloc( blocksPerGrid*sizeof(float) );
33     // allocate the memory on the GPU
34     cudaMalloc( (void**)&dev_a, N*sizeof(float) );
35     cudaMalloc( (void**)&dev_b, N*sizeof(float) );
36     cudaMalloc( (void**)&dev_partial_c, blocksPerGrid*sizeof(float) );
37     // fill in the host memory with data
38     for (int i=0; i<N; i++) {
39         a[i] = i;
40         b[i] = i*2; }
41     // copy the arrays 'a' and 'b' to the GPU
42     cudaMemcpy( dev_a, a, N*sizeof(float), cudaMemcpyHostToDevice );
43     cudaMemcpy( dev_b, b, N*sizeof(float), cudaMemcpyHostToDevice );
44
45     dot<<<blocksPerGrid,threadsPerBlock>>>( dev_a, dev_b, dev_partial_c );
46
47     // copy the array 'c' back from the GPU to the CPU
48     cudaMemcpy( partial_c, dev_partial_c, blocksPerGrid*sizeof(float), cu
    daMemcpyDeviceToHost );
49     // finish up on the CPU side
50     c = 0;
51     for (int i=0; i<blocksPerGrid; i++) {
52         c += partial_c[i];
53     }
54     printf("Result\\,: %f\\n", c);
55     // free memory on the GPU side
56     cudaFree( dev_a );
57     cudaFree( dev_b );
58     cudaFree( dev_partial_c );
59     // free memory on the CPU side
60     free( a );
61     free( b );
62     free( partial_c );
63 }

```