AVIGNON UNIVERSITÉ

TP3

ILSEN-Alt
LUO Yingqi

16 octobre 2022

Master d'Informatique
Ingénierie Logiciel et Sécurité Numérique
UE Programmation Parallèle
UCE

Responsables
ROUVIER Mickael,
LABRAK Yanis

UFR
SCIENCES
TECHNOLOGIES
SANTÉ

AVIGNON UNIVERSITÉ
CERI

CENTRE
D'**ENSEIGNEMENT**
ET DE **RECHERCHE**
EN **INFORMATIQUE**
ceri.univ-avignon.fr

Sommaire

1 La fonction main

- réserver l'espace sur CPU, et initisaliser les matrice.

```
int *a = (int *)malloc(size);
int *b = (int *)malloc(size);
int *c = (int *)malloc(size);
```

```
// initialiser les vectors a et b pour calculer leur addition après:
for (int i = 0; i < vectorSize; ++i)
{
    a[i] = i;
    b[i] = i;
    c[i] = 0;
}
```

```
// initialiser les vectors a et b pour calculer leur multiplication après:
for (int i = 0; i < nbRow * 8; ++i)
{
    a[i] = i;
    b[i] = i;
}

for (int i = 0; i < rsltVectorSize; ++i)
{
    c[i] = 0;
}
```

- Distribuer autant d'espace sur GPU.

```
int *da, *db, *dc;

cudaMallocManaged(&da, size);
cudaMallocManaged(&db, size);
cudaMallocManaged(&dc, size);
```

- Transférer les opérandes au GPU.

```
cudaMemcpy(da, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(db, b, size, cudaMemcpyHostToDevice);
cudaMemcpy(dc, c, size, cudaMemcpyHostToDevice);
```

- Calcule dans GPU.

```
/** Fonction kernel */
```

- Récupère les données du GPU vers le CPU.

```
cudaMemcpy(c, dc, size, cudaMemcpyDeviceToHost);
```

2 Addition de deux vecteurs

Pour calculer l'addition de deux vectors, je distribue chaque thread pour calculer chaque élément du résultat de vector.

2.1 Fonction kernel

```
add<<<numberOfBlock, threadsPerBlock>>>(da, db, dc);
```

```
__global__ void add(int *a, int *b, int *c)
{
    int index = blockDim.x * blockIdx.x + threadIdx.x;
    c[index] = a[index] + b[index];
}
```

2.2 Temps d'exécution avec les différents nombre de block et de thread

- nombre de block : 32
  nombre de thread par block : 8

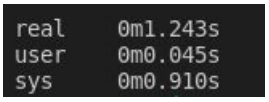

Figure 1

- nombre de block : 126
  nombre de thread par block : 100

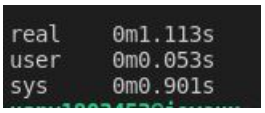

Figure 2

- nombre de block : 126
  nombre de thread par block : 512

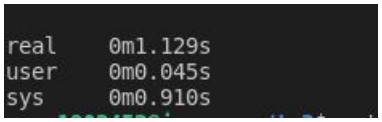

Figure 3

- nombre de block : 512
  nombre de thread par block : 126

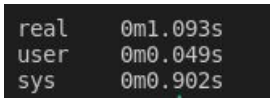

Figure 4

3 Multiplication de deux matrices

Pour la calcul de multiplication, un thread calcule un élément de la matrice de résultat. Le size de la vector résultat est équal à nombre de block(nombre de lignes du matrice résultat) * nombre de thread par block(nombre de collones du matrice résultat).

Pour le nombre de lignes de matrice a et le nombre de collones de matrice b est égal à racine carré du size totale de la vector résultat.

3.1 Fonction kernel

```
multiply<<<numberOfBlock, threadsPerBlock>>>(da, db, dc, vectorSize, nbRow);
```

```
__global__ void multiply(int *a, int *b, int *c, int vectorSize, int nbRow)
{

    int index = blockDim.x * blockIdx.x + threadIdx.x;
    int row = index / nbRow;
    int col = index % nbRow;

    int rslt = 0;
    for (int i = 0; i < nbRow; i++)
    {
        rslt += a[row * nbRow + i] * b[i * nbRow + col];
    }

    c[index] = rslt;
}
```

3.2 Temps d'exécution avec les différents nombre de block et de thread

- nombre de block : 32
  nombre de thread par block : 8

```
real    0m1.134s
user    0m0.033s
sys     0m0.919s
```

Figure 5

- nombre de block : 126
  nombre de thread par block : 100

```
real    0m1.078s
user    0m0.041s
sys     0m0.896s
```

Figure 6

- nombre de block : 126
  nombre de thread par block : 512

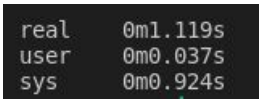

Figure 7

- nombre de block : 512
  nombre de thread par block : 126

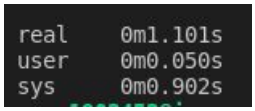

Figure 8

4 Conclusion

Après observation, des temps d'exécution sont très similaire, cela prouve bien l'utilité de faire du parallélisme