Chapitre 9 : Exercice 1

Versions :

1. Version 1 – naïve thread mapping :

```
__global__ void dotNaive(float *a, float *b, float *res, int n) {
   __shared__ float partialSum[2*BLOCK_SIZE];
   unsigned int t = threadIdx.x;
   unsigned int start = 2*blockIdx.x*blockDim.x;

   partialSum[t] = a[start + t] * b[start + t];
   partialSum[blockDim.x+t] = a[start + blockDim.x+t] * b[start + blockDim.x+t];

   for (unsigned int stride = 1;stride <= blockDim.x; stride *= 2) {
         __syncthreads();
         if (t % stride == 0)
               partialSum[2*t] += partialSum[2*t+stride];
   }
   res[0] = partialSum[0];
}
```

2. Version 2 – gère toutes tailles de vecteur :

```
#define BLOCK_SIZE 1024
__global__ void dotNaive(float *a, float *b, float *res, int n) {
   __shared__ float partialSum[2*BLOCK_SIZE];
   unsigned int t = threadIdx.x;
   unsigned int start = 2*blockIdx.x*blockDim.x;

   if (start + t < n)
         partialSum[t] = a[start + t] * b[start + t];
   else
         partialSum[t] = 0.0;
   if (start + blockDim.x+t < n)
         partialSum[blockDim.x+t] = a[start + blockDim.x+t] * b[start + blockDim.x+t];
    else
         partialSum[blockDim.x+t] = 0.0;
   for (unsigned int stride = blockDim.x; stride > 0; stride /= 2) {
         __syncthreads();
         if (t < stride)
               partialSum[t] += partialSum[t+stride];
   }
   res[blockIdx.x] = partialSum[0];
}

int block_nb = (n - 1)/(2*BLOCK_SIZE)+1;
```

3. Version 3 – meilleure réduction de kernel :

```
for (unsigned int stride = blockDim.x; stride > 0; stride /= 2) {
        __syncthreads();
        if (t < stride)
                partialSum[t] += partialSum[t+stride];
 }
 res[blockIdx.x] = partialSum[0];
```

Questions

How many floating operations are being performed in your reduction kernel ?

Pour commencer, afin d'implémenter nos éléments dans la mémoire partagée, nous multiplions deux éléments pour chaque threads dans chaque block. Dans nous avons 2*block_size*block_nb.

Avec :

- Block_size : taille des blocks
- Block_nb : nombre de block

Concernant, l'addition, il y en a autant que notre stride est divisible par deux.

Mathématique nous pouvons tel quel :

$$\sum_{i=1} \frac{block\_size}{2^i}$$

Avec i le nombre de fois que stride est divisible par 2.

On peut donc avoir un nombre d'opérations flottantes :

$$2 * block\_size * block\_nb + \sum_{i=1} \frac{block\_size}{2^i}$$

How many gobal memory reads are being performed by your kernel ?

Chaque threads doit charger 4 fois les données. De ce fait, pour l'accès en lecture, nous avons :

Al = thread*4*n

Avec :

- N la taille du vecteur

How many global memory writes are being performed by your kernel ?

Chaque block va récupérer qu'une seule sortie de notre mémoire partagé (partialSum). Puis on fait somme de notre résultat en fonction du nombre de block présent. De ce fait, l'accès en écriture est tel que :

Ar : block_nb = (n-1)/(2*BLOCK_SIZE)+1

Avec :

- N taille du vecteur

- BLOCK_SIZE taille des blocks (1024 threads)

How many times does a single thread block synchronize to reduce its portion of the array to a single value ?

Pour chaque block, nous synchronisons un thread autant de fois que de pas (stride) de calcul à faire. C'est-à-dire le nombre de fois jusqu'à que notre foulé sera inférieur à 0, en sachant que chaque foulé est divisé par 2 à chaque pas (car on fait un calcul entre deux éléments).

On divise donc notre pas autant de fois que notre taille du block est divisible par deux.

Mathématiquement, cela viendra à dire :

$$\frac{\log(blocksize)}{\log(2)}$$

Dans ma situation où il y a 1024 threads, cela voudrait dire qu'un thread se synchronise 10 fois. Car il y a aura 10 fois la boucle for qui va s'exécuter jusqu'à que notre nombre de threads soit négative quand on le divise par 2.

Explain why the third version better than the second one.

Cette version est plus efficace car nous avons des opérations flottantes à faire en moins. En effet, on ne multiplie plus par 2 notre t (id du thread) pour avoir l'emplacement de nos éléments dans la mémoire partagée. Et le changement effectué dans la boucle for n'a pas d'impact : le nombre d'opération flottante reste la même.

Explain how to get rid of the data transfer or partial results and the host-side final reduction.

Afin de ne plus avoir de calcul à faire sur le côté host, nous pouvons tout faire sur le côté device et ainsi se débarrasser des résultats partiels. Pour ce faire, nous pouvons créer un deuxième kernel qui appellerait notre kernel principale, ferait la somme entre tous nos résultats pour chaque block et ainsi on ne transfèrera qu'un seul résultat à notre host.

Describe what changes are needed to handle vector lenght larger than the maximum thread number in the CUDA grid.

On cherche à savoir ce qu'il est possible de faire dans cette situation : $n > block\_size$. Or dans mon programme, cette situation est déjà résolu.

Is it possible that the result changes from an exécution to an other ?

D'après le cours, l'ordre des opérations n'a pas d'importance. De ce fait, je ne pense pas qu'il y aura un changement dans le résultat.