CUDA

Les fichiers sources sont disponibles à cette adresse : http://matthieu-2807.github.io/CUDA/

Lab0 - Hello World

On compile le Hello World avec la commande *nvcc cuda_ex_1.c –o cuda_ex_1*. On exécute ensuite la commande *./cuda_ex_1* pour tester le fonctionnement.

```
etudiant1@pc-gi-572:~/lab0$ nvcc cuda_ex_1.c -o cuda_ex_1
etudiant1@pc-gi-572:~/lab0$ ls
cuda_ex_1  cuda_ex_1.c  cuda_ex_1.cu
etudiant1@pc-gi-572:~/lab0$ ./cuda_ex_1
Hello World!
```

Lab1 - Vector Add

Le fichier de base donne le résultat suivant :

```
etudiant1@pc-gi-572:~/lab1$ make cpu
gcc -O3 vecadd.c -o vecadd_cpu
etudiant1@pc-gi-572:~/lab1$ ./vecadd_cpu
Running CPU vecAdd for 720896 elements
time=0.0018 seconds, MFLOPS=406.8
C[0]=2147483648.00
C[1]=2147483648.00
C[2]=2147483648.00
C[3]=2147483648.00
C[4]=2147483648.00
C[5]=2147483648.00
C[6]=2147483648.00
C[7]=2147483648.00
C[8]=2147483648.00
C[9]=2147483648.00
```

On insère les lignes suivantes pour faire une allocation de mémoire dans le GPU :

```
1  cudaMalloc((void**)&dev_a, N * sizeof(float));
2  cudaMalloc((void**)&dev_b, N * sizeof(float));
3  cudaMalloc((void**)&dev_c, N * sizeof(float));
```

Pour la libérer on utilise les lignes suivantes :

```
1 cudaFree(dev_a);
2 cudaFree(dev_b);
3 cudaFree(dev_c);
```

Ensuite pour échanger les données entre l'host et le GPU, nous utilisons les commandes suivantes :

```
// Copie les données HOST -> GPU
 cudaMemcpy(dev_a, A, N * sizeof(float), cudaMemcpyHostToDevice);
 cudaMemcpy(dev_b, B, N * sizeof(float), cudaMemcpyHostToDevice);
// Copie les données de GPU -> HOST
cudaMemcpy(C, dev_c, N * sizeof(float), cudaMemcpyDeviceToHost);
```

Pour s'assurer qu'il n'y a pas d'erreur lors de l'allocation de mémoire GPU, nous mettons en place un système de lecture d'erreur pour chaque allocation.

```
cudaError_t err;
err = cudaMalloc((void**)&dev_a, N * sizeof(float));
if (err != cudaSuccess)
{
    fprintf(stderr, "cudaMalloc ERROR : , %s.\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}
```

Pour finir nous écrivons un système de timer permettant de connaitre le temps d'exécution.

```
/* On crée les timer et on lance begin */
cudaEventCreate(&begin);
cudaEventCreate(&stop);
cudaEventRecord(begin, 0);

//////////////////////////////
/* Le code à chronométrer */
//////////////////////////////

/* On arrete le chrono et on compare begin et stop */
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&rt, begin, stop);  // Résultat en milliseconde
rt /= 1E3; // On converti en seconde
printf("time=%.4f seconds, MFLOPS=%.1f\n", rt, (float)N/rt/1E6);
/* On supprime les timers */
cudaEventDestroy(begin);
cudaEventDestroy(stop);
```

Après modification du fichier makefile pour l'adapter au nom de notre fichier, nous avons le résultat suivant après exécution.

```
etudiant1@pc-gi-572:~/lab1$ make gpu
nvcc vecadd1.cu -o vecadd_gpu
etudiant1@pc-gi-572:~/lab1$ ./vecadd_gpu
Running GPU vecAdd for 720896 elements
time=0.0021 seconds, MFLOPS=337.3
C[0]=2147483648.00
C[1]=2147483648.00
C[2]=2147483648.00
```

```
C[3]=2147483648.00
C[4]=2147483648.00
C[5]=2147483648.00
C[6]=2147483648.00
C[7]=2147483648.00
C[8]=2147483648.00
C[9]=2147483648.00
```

Lab2 - Device management

Vérification de la compatibilité CUDA

On souhaite que le code s'exécute uniquement sur les devices qui ont une version égale ou supérieure à 1.3. De plus nous souhaitons séparer le code CUDA du code Main. A la fin de cette modification, nous devons modifier le makefile pour que tout compile correctement avec les nouveaux fichiers.

Première étape, s'assurer que le device est compatible.

```
1  void cudaDeviceInit(int major, int minor)
2  {
3      int devCount, device;
4      cudaGetDeviceCount(&devCount);
5  // Aucun GPU pouvant utiliser CUDA n'a été détecté
6      if (devCount == 0)
7      {
8          printf("No CUDA capable devices detected.\n");
9          exit(EXIT_FAILURE);
10     }
11 // On regarde les propriétés du GPU
12     for (device=0; device < devCount; device++)
13     {
14         struct cudaDeviceProp props;
15         cudaGetDeviceProperties(&props, device);
16 /* If a device of compute capability >= major.minor is found,
   use it */
17         if (props.major > major || (props.major == major &&
   props.minor >= minor)) break;
18     }
19 // Si les propriétés sont inférieures à ce qui est demandé, on
   quitte le code, sinon on poursuit le code
20     if (device == devCount)
21     {
22         printf("No device with %d.%d compute capability or
   above is detected.\n", major, minor);
23         exit(EXIT_FAILURE);
24     }
25     else
26         cudaSetDevice(device);
27 }
```

Séparation du code CPU/GPU

Pour séparer le code, nous faisons un fichier main.c contenant uniquement du code C et un fichier vecadd.cu contenant du code CUDA. La partie précédente se trouve bien entendu dans le fichier vecadd.cu. Dans notre main.c nous mettons deux lignes pour utiliser les méthodes de CUDA.

La première ligne est : `cudaDeviceInit(1,3);`

Cette ligne permet de vérifier la compatibilité du device. Ensuite nous la ligne qui permet de lancer le calcul : `rt = vecAdd(A, B, C, N);`

Le reste du fichier main.c est le traitement des données retourné par le GPU.

Changement du makefile

Nous modifions le makefile pour permettre de compiler tous les fichiers modifiés et créés via la commande **make myapp**

```
CC        = nvcc
CCFLAGS   = -O3 -I/usr/local/cuda/include -Dreal=double
NVCC      = nvcc
CUFLAGS   = -arch sm_13
LINK      = nvcc
LINKFLAGS = -L/usr/local/cuda/lib64 -lcuda
all: clean myapp
myapp: main.o utils.o vecadd.o
$(LINK) $(LINKFLAGS) utils.o main.o vecadd.o -o $@
clean:
rm -rf myapp vecadd.linkinfo *~ *.o
main.o: main.c
$(CC) $(CCFLAGS) -c $<
utils.o: utils.c
$(CC) $(CCFLAGS) -c $<
vecadd.o: vecadd.cu
$(NVCC) $(CUFLAGS) $(CCFLAGS) -c $<
```

Lab3 - Sum / Dot product

La somme (ou produit scalaire = "dot product") consiste à prendre des groupes de 2 nombres (entiers, décimaux, …) et de faire la somme des produits de ces 2 nombres.

Dans un premier temps, nous allouons nos variables et leur donnons une valeur aléatoire (random). Ensuite nous faisant appel à la méthode produit scalaire qui s'écrit comme suit, puis nous affichons le résultat.

```
1  #define THREADS_PER_BLOCK 512
2  __global__ void dot(int *a, int *b, int *c)
3  {
4       __shared__ int temp[THREADS_PER_BLOCK];
5       int index = threadIdx.x + blockIdx.x * blockDim.x;
6       temp[threadIdx.x] = a[index]*b[index];
7       __syncthreads();
8       if(0 == threadIdx.x){
9           int sum = 0;
10          for(int i = 0; i < THREADS_PER_BLOCK;i++)
11              sum += temp[i];
12          atomicAdd(c,sum);
13      }
14 }
```

Le principe de cette méthode consiste à récupérer chaque produit (a1*b1, a2*b2, …) calculé dans des threads différents. On attend ensuite la fin des calculs dans chaque thread, c'est-ce qu'on appelle la

synchronisation. Enfin, on effectue le "regroupement" des valeurs afin de calculer la somme de ces produits.

Lab4 - Matrix-Matrix multiplication

Les jeux vidéo utilisent en permanence des matrices afin de mettre à jour les données à afficher sur votre écran. Cela peut être l'affichage d'un personnage ou le déplacement de la caméra. Nous allons voir ici un petit exemple d'utilisation de CUDA pour exécuter des traitements parallèles sur de très grandes matrices.

Dans l'exemple présenté, nous utilisons CUDA pour récupérer le résultat d'une multiplication entre 2 matrices. Par défaut, nous aurions tendance à boucler sur les différents éléments des 2 matrices pour effectuer nos calculs. Grâce au traitement, nous pouvons réduire ces boucles (3) à une seule boucle qui parcourt chaque élément de la matrice finale.

```
1  __global__ void mmult(float *a, float *b, float *c, int N)
2  {
3        int row = blockIdx.y;
4        int col = blockIdx.x*32 + threadIdx.x;
5        float sum = 0.0f;
6        for (int n = 0; n < N; ++n)
7        {
8              sum += a[row*N+n]*b[n*N+col];
9        }
10       c[row*N+col] = sum;
11 }
```

Dans cette méthode, on calcule chaque élément de la matrice finale ("c") en suivant le principe de la multiplication de matrices. On stocke ensuite la valeur de l'élément dans une variable ("sum"), variable assignée à la valeur de l'élément à la fin de la méthode.