M2 E3A - SETI

TP GPU

Saša Radosavljevic

27 février 2023

Table des matières

1 | TP1 Seuillage

1.1 - Contexte

Lors de ce TP, nous allons programmer un algorithme de seuillage couleur à l'aide de trois langages de programmation. Nous analyserons les résultats relevant des temps d'exécution notamment entre différentes optimisations CPU sous matlab. Nous verrons alors l'exécution de ce même seuillage sur une cible Nvidia Jetson Nano. Nous pourrons alors mesurer les temps d'exécutions entre les versions CPU et CUDA afin de voir les gains que l'on peut tirer d'une utilisation d'un GP-GPU pour le traitement parallèle.

1.2 - Matlab

La première partie consiste à programmer l'algorithme de seuillage et de changement de couleur sur Matlab. Matlab a la particularité de pouvoir traiter des matrices dans ses opérations. De ce fait, les différentes boucles itératives permettant de parcourir les pixels d'une image peuvent êtres enlevées afin de tirer un maximum des avantages offerts par Matlab.

L'exécution sera faite sur un PC fixe muni d'un CPU Ryzen 9 3900x de fréquence de base à 3,8 Ghz avec un boost à 4,6 Ghz. Il est composé de 12 coeurs et d'un cache L3 de 64 Mo. La mémoire RAM est composée de 2 barrettes de 16 Go à 3200 MHz C16 en dual channel.

Le code exécuté est donné ci-dessous :

```
%tic toc pour mesurer le temps de calcul
disp("Seuil :");

tic;
ima_out=ima;
nr = ima_r./sqrt(ima_r.^2 + ima_b.^2 + ima_g.^2);
ima_seuil = ima.*(nr>0.7);
toc;
figure('name','RGB out','numbertitle','off');image(ima_seuil);

disp("Jaune :")
```

```
tic;
ima_jaune = ima_seuil;
ima_jaune(:,:,2) = ima_seuil(:,:,1);
toc;
figure('name','RGB out','numbertitle','off');image(ima_jaune);

disp("Reinsertion dans l'image :")
tic;
ima_out = ima - ima_seuil + ima_jaune;
figure('name','RGB out','numbertitle','off');image(ima_out);
toc;
```

On obtient les résultats suivants :

- Seuil : Elapsed time is 0.006431 seconds.
- Jaune : Elapsed time is 0.006348 seconds.
- Réinsertion dans l'image : Elapsed time is 0.003195 seconds..
- Exécution totale : 0.015394 seconds.

Si l'on compare ce résultat avec l'exécution du code avec boucles, c'est-à-dire boucle sur i et j avec calcul de l'intensité de rouge dans les boucles, puis en dehors des boucles et pour finir le calcul sans boucles, on obtient les résultats suivants :

- VERSION 1 : Elapsed time is 0.078048 seconds.
- VERSION 2 : Elapsed time is 0.020645 seconds.
- VERSION 3 : Elapsed time is 0.013962 seconds.

On observe bien une amélioration d'un facteur 6 lorsqu'on enlève les boucles. On obtient un résultat similaire entre les deux versions (codée et solution) sans boucles.

(a) Image d'entrée

(b) Image après traitement

FIGURE 1.1 – Exécution du programme de changement de couleur

1.3 - Jetson nano CPU

La deuxième partie s'effectue sur une carte Nvidia Jetson Nano dotée d'un CPU Quad-core ARM A57 @ 1.43 GHz.

On exécutera sur la cible le code suivant :

```
void seuillage_C(float image_out[][SIZE_J][SIZE_I], float
 ↪ image_in[][SIZE_J][SIZE_I])
{
    float r, g, b;
    for(int j=0; j<SIZE_J; j++){
        for(int i=0; i<SIZE_I; i++){
            r = image_in[0][j][i];
            g = image_in[1][j][i];
            b = image_in[2][j][i];

            if(r/sqrt(r*r+g*g+b*b) > 0.7){
                image_out[0][j][i] = r;
                image_out[1][j][i] = g;
                image_out[2][j][i] = b;
            }
        }
    }
}
```

En observant l'impact d'inversion des indices i et j sur l'optimisation des calculs, on obtient :

- Boucles bon ordre pour le cache : 95 ms.
- Boucles mauvais ordre pour le cache : 171 ms.

En comparaison avec le temps obtenu avec la version 1 de Matlab on est à un facteur 1,21, ce qui est assez proche au vu du processeur utilisé précedemment. Le même code exécuté sur le ryzen obtient un temps d'exécution de 6,5 ms. On a cette fois un vrai écart d'un facteur 14,6 au vu de la différence de puissance de calcul.

1.4 - Jetson Nano CUDA

En dernier lieu, on souhaite utiliser le GPU de la Jetson Nano pour observer les bénéfices de l'utilisation de CUDA. On peut tout d'abord regarder les caractéristiques du GPU avec l'exécutable `deviceQuery` disponible dans le répertoire */usr/local/cuda/samples/bin*.

deviceQuery :

- Multiprocesseur : 1
- CUDA cores : 128
- Max Fclock = 922Mhz
- Mémoire : 2 Go
- Memory clock : 13 Mhz
- L2 : 262 Ko
- Warp size : 12
- Maximum number of threads per block : 1024

La RTX 3060 ti dispose de 4864 coeurs Cuda @ 1,6Ghz

Disposant de 1024 threads par block, une des solutions serait d'attribuer une ligne par block soit 960 pixels -> 960 threads.

Soit le programme à exécuter,

```
__global__ void seuillage_kernel(float d_image_in[][SIZE_J][SIZE_I],
 ↪  float d_image_out[][SIZE_J][SIZE_I])
{
        // A VOUS DE CODER
        int i, j;
        float r, g, b;

        j = blockIdx.x;
        i = threadIdx.x;

        r = d_image_in[0][j][i];
        g = d_image_in[1][j][i];
        b = d_image_in[2][j][i];

        if ((r / sqrt(r * r + g * g + b * b)) > 0.7) {
                d_image_out[0][j][i] = r;
                d_image_out[1][j][i] = g;
                d_image_out[2][j][i] = b;
        }
}
```

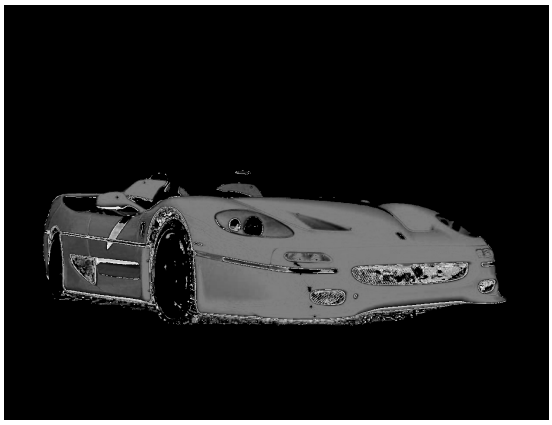
On obtient les résultats suivants :

Cible	CPU (ms)	GPU (ms)	Mem Manag (%)	GPU brut (ms)	R CPU	R Cuda
Nano	84	53	90	5,3	1	1
PC Matlab	13.9				6	
PC C/Cuda	6,5	8,71	98,6	0,12	12,9	44,6

La Nvidia A100 ayant 1,27x plus de coeurs cuda, on peut dire que le temps de traitement brut serait de 0,1 ms et augmentant encore un peu le memory management à 99%. On

peut retomber sur ce résultat en comparant le nombre de coeurs de la Nano, soit un rapport de 48,35, $5,3/48,35 = 0,109$ ms.

Et les images en sortie :



(a) Image après seuil CPU

(b) Image après seuil GPU et canal rouge

FIGURE 1.2 – Exécution du programme de seuil

1.5 - Kmeans Python

Le premier TP de l'UE d'outils pour le Machine Learning (D3), a pour objectif d'utiliser l'algorithme des Kmeans pour transformer les pixels d'une image vers le centre du cluster auquel il appartient. Pour ce faire, nous utilisons différents outils disponibles sur Python et scikit-learn. Malheureusement, ce script est assez gourmand et son temps d'exécution est élevé. On propose avec le code suivant d'observer le gain de performance apporté par CUDA.

```
1  import cupy as cp
2  import numpy as np
3  from sklearn.cluster import KMeans
4  from skimage import io
5  import time
6
7  # Load the image using skimage
8  image = io.imread('fruits.jpg')
9
10 # Convert the image to a CuPy array
11 image_cp = cp.asarray(image)
12
13 # Flatten the image into a 2D array of pixels
14 image_flat = image_cp.get().reshape(image_cp.shape[0] *
    ↪  image_cp.shape[1], image_cp.shape[2])
15
16 def Kmeans_cuda(K=1):
```

```
    # Use KMeans to cluster the pixels into a specified number of clusters
    kmeans = KMeans(n_clusters=K, random_state=0).fit(image_flat)

    # Predict the cluster for each pixel
    clusters = kmeans.predict(image_flat)

    # Create a new CuPy array to hold the modified image
    new_image_cp = cp.empty_like(image_cp)

    # Iterate over each pixel and assign its value to the corresponding cluster center
    for i, cluster in enumerate(clusters):
        new_image_cp[i // image_cp.shape[1], i % image_cp.shape[1]] = cp.asarray(kmeans.cluster_centers_[cluster])

    # Convert the CuPy array back to a NumPy array
    new_image = cp.asnumpy(new_image_cp)

    # Save the modified image using skimage
    io.imsave("fruits" + "_%d" % K + "_cuda.jpg", new_image)


for K in range(1,256):
    start_time = time.time()
    Kmeans_cuda(K=K)
    end_time = time.time()
    print(f"It took {end_time-start_time:.2f} seconds to compute for K =",K)
```

Et le résultat est assez flagrant :

Pour K = 20 clusters, on a un rapport de temps d'exécution de 325 entre CPU et CUDA.

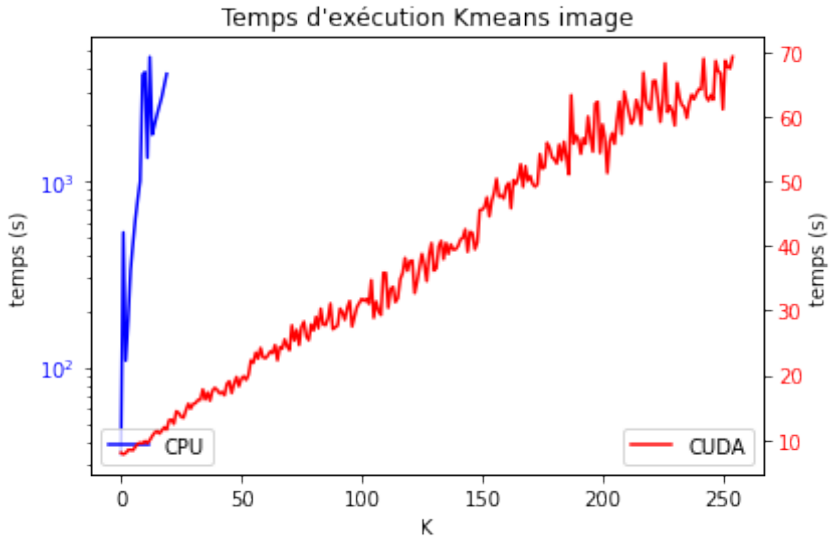


FIGURE 1.3 – Temps d'exécution Kmeans sur CPU et CUDA d'une image de 103 230 pixels

1.6 - Conclusion

Au cours de ce TP, nous avons appris à utiliser les outils CUDA en passant par une première analyse sous Matlab et en C. Nous avons pu constater la puissance que peut offrir CUDA pour le traitement d'un ensemble très large de données. La majeur problème est l'accès aux données qui peuvent ralentir toute l'exécution du problème, notamment lorsque l'on possède un GPU très puissant créant un bottleneck sur le transport de données.

2 | TP2 Réduction sommation

2.1 - Contexte

Au cours de ce TP, nous travaillerons soit sur Nvidia Jetson Nano, soit une carte Nvidia RTX 3060 Ti personnelle. (Voir TP Seuillage pour les specs de la Nano)

RTX 3060 ti :

- 38 MP
- 4864 Cuda Cores
- Max clock rate = 1665 MHz - Max threads/block = 1024 - Max threads/multiprocessor = 1536 - Memory 8192 MB

On souhaite mesurer les temps exécutions que peuvent proposer les différentes solutions proposées à notre problème de sommation sur des variables dépendantes. L'objectif est d'essayer de profiter de toutes les capacités offerte par un GPU pour la parallélisation de ce type d'algorithme.

2.2 - Analyse du code

Nous avons 5 voire 6 (solution) méthodes pour essayer de réduire le temps de calcul au maximum.

Calcul sur CPU :

```
int sum = 0;
for( int i_test = 0 ; i_test < NUM_TESTS ; ++i_test )
{
    sum = 0;
    for( int i = 0 ; i < N ; ++i )
        sum += a_host[i];
}
```

Calcul sur CPU + OpenMP :

```
omp_sum = 0;
#pragma omp parallel shared(omp_sum){
    #pragma omp for reduction(+:omp_sum)
    for(int i=0; i<N; i++){
        omp_sum += a_host[i];
    }
}
```

Calcul sur GPU avec Thrust :

```
int thrust_sum = 0;
for( int i_test = 0 ; i_test < NUM_TESTS ; ++i_test )
{
    thrust_sum = thrust::reduce( thrust::device_ptr<int>(a_device),
    ↪ thrust::device_ptr<int>(a_device+N) );
}
```

Calcul sur GPU :

```
int gpu_sum = 0;
for( int i_test = 0 ; i_test < NUM_TESTS ; ++i_test )
{
gpu_sum = reduce_on_gpu( N, a_device );
}
```

Calcul sur GPU optimisé :

```
int optim_gpu_sum = 0;
for( int i_test = 0 ; i_test < NUM_TESTS ; ++i_test )
{
    optim_gpu_sum = reduce_on_gpu_optimized<BLOCK_DIM>( N, a_device );
}
```

2.3 - Réduction GPU

La première solution non optimisée consiste à utiliser les blocs de threads disponibles sur le GPU.

2.3.1 - Méthode non parallélisée

```
// Compute the sum of my elements.
for (int i = threadIdx.x + range.x; i < range.y; i += blockDim.x) {
    my_sum += in_buffer[i];
}

// Copy my sum in shared memory.
s_mem[threadIdx.x] = my_sum;

// Make sure all the threads have copied their value in shared memory.
__syncthreads();

// Compute the sum inside the block.
// The first thread of the block stores its result.
if (threadIdx.x == 0) {
    float sum2 = 0;
    for (int i = 0; i < blockDim.x; i++) {
        sum2 += s_mem[i];
    }
    out_buffer[blockIdx.x] = sum2;
}
```

2.3.2 - Méthode par réduction

```
// Compute the sum of my elements.
int my_sum = 0;
for (int idx = range.x + threadIdx.x; idx < range.y; idx += blockDim.x)
    my_sum += in_buffer[idx];

// Copy my sum in shared memory.
s_mem[threadIdx.x] = my_sum;

// Make sure all the threads have copied their value in shared memory.
__syncthreads();

int offset;
// Compute the sum inside the block.
for (offset = blockDim.x / 2; offset > 16; offset /= 2)
{
    if (threadIdx.x < offset)
        s_mem[threadIdx.x] += s_mem[threadIdx.x + offset];
    __syncthreads();
}

```

```
//INSIDE WARP 0 SYNC NOT NECESSARY
for (; offset > 0; offset /= 2)
{
    if (threadIdx.x < offset)
        s_mem[threadIdx.x] += s_mem[threadIdx.x + offset];
    //__syncthreads( );
}


// The first thread of the block stores its result.
if (threadIdx.x == 0)
    out_buffer[blockIdx.x] = s_mem[0];
```

2.4 - Réduction optimisée

Cette fois-ci, on va chercher à utiliser les plus petits blocs de threads pouvant être utilisés, aussi appelés des warps. Ils contiennent chacun 32 threads.

2.4.1 - Méthode non parallélisée

```
// Compute the sum of my elements.
for (int i = threadIdx.x + range.x; i < range.y; i += blockDim.x) {
    my_sum += in_buffer[i];
}
// Copy my sum in shared memory.
s_mem[threadIdx.x] = my_sum;

// Compute the sum inside each warp.
__syncthreads();

// Each warp leader stores the result for the warp.
if (lane_id == 0) {
    float sum2 = 0;
    for (int i = threadIdx.x; i < threadIdx.x + WARP_SIZE; i++) {
        sum2 += s_mem[i];
    }
    s_mem[threadIdx.x] = sum2;
}
__syncthreads();

if (warp_id == 0)
{
    // Read my value from shared memory and store it in a register.
    // Sum the results of the warps.
```

```
    if (threadIdx.x == 0) {
        float sum3 = 0;
        for (int i = threadIdx.x; i < threadIdx.x + blockDim.x; i += WARP_SIZE) {
            sum3 += s_mem[i];
        }

        out_buffer[blockIdx.x] = sum3;
    }
}
```

2.4.2 - Méthode optimisée

```
// Compute the sum of my elements.
int my_sum = 0;
for (int idx = range.x + threadIdx.x; idx < range.y; idx += BLOCK_DIM)
    my_sum += in_buffer[idx];

// Copy my sum in shared memory.
s_mem[threadIdx.x] = my_sum;

// Compute the sum inside each warp.
#pragma unroll
for (int offset = 16; offset > 1; offset >>= 1)
    if (lane_id < offset)
        s_mem[threadIdx.x] = my_sum += s_mem[threadIdx.x + offset];

__syncthreads();

// Each warp leader stores the result for the warp.
if (lane_id == 0)
    s_mem[warp_id] = my_sum += s_mem[threadIdx.x + 1];

__syncthreads();

if (warp_id == 0)
{
    // Read my value from shared memory and store it in a register.
    my_sum = s_mem[lane_id];

    // Sum the results of the warps.
#pragma unroll
    for (int offset = NUM_WARPS / 2; offset > 1; offset >>= 1)
        if (threadIdx.x < offset)
            s_mem[threadIdx.x] = my_sum += s_mem[threadIdx.x + offset];
}
```

```
34
35  // The 1st thread stores the result of the block.
36  if (threadIdx.x == 0)
37      out_buffer[blockIdx.x] = my_sum += s_mem[1];
```

2.4.3 - Méthode optimisée sans mysum+=

```
// Compute the sum of my elements.
int my_sum = 0;
for (int idx = range.x + threadIdx.x; idx < range.y; idx += BLOCK_DIM)
    my_sum += in_buffer[idx];

// Copy my sum in shared memory.
s_mem[threadIdx.x] = my_sum;

// Compute the sum inside each warp.
#pragma unroll
for (int offset = 16; offset > 0; offset >>= 1)
    if (lane_id < offset)
        s_mem[threadIdx.x] += s_mem[threadIdx.x + offset];

__syncthreads();

// Each warp leader stores the result for the warp.
if (lane_id == 0)
    s_mem[warp_id] = s_mem[threadIdx.x];

__syncthreads();

if (warp_id == 0)
{
    // Sum the results of the warps.
#pragma unroll
    for (int offset = NUM_WARPS / 2; offset > 0; offset >>= 1)
        if (threadIdx.x < offset)
            s_mem[threadIdx.x] += s_mem[threadIdx.x + offset];
}

// The 1st thread stores the result of the block.
if (threadIdx.x == 0)
    out_buffer[blockIdx.x] = s_mem[0];
```

2.5 - Résultats

Les codes présentés précédemment sont exécutés sur un Desktop composé d'un AMD Ryzen 9 3900x et d'une Nvidia 3060ti.

	Non parallélisé (ms)	Parallélisé (ms)
CPU 1 Thread	103	103
CPU 24 Threads	8.7	8.7
GPU Thrust	124	124
GPU	2.32	1.99
GPU opti	2.12	1.98
GPU mysum		1.93

TABLEAU 2.1 – Temps d'exécution des différentes versions de la sommation

Il semblerait que le PC ai du mal avec Thrust, mais il n'y a pas d'erreur lors de la compilation sur visual studio.

2.6 - Conclusion

Nous avons pu apprendre à utiliser les threads en blocs et en warps pour paralléliser au maximum un algorithme de sommation sur une variable dépendante à l'aide de la méthode par réduction. La méthode de réduction est disponible par défaut sur openMP et avec Thrust nous permettant en une simple instruction de réaliser le code de réduction qui, cependant, obtient de meilleure performances lorsqu'il est réécrit pour l'application en ayant connaissance de l'architecture de l'algorithme et d'un GPU.