

Calcul sur GPU

Paul-Etienne Rétaux et Jiangbo Wang, 31 janvier 2024

Introduction

Cette séance de quatre heures a pour vocation de nous familiariser avec la surcouche CUDA, proposée par NVIDIA pour le calcul sur GPU. Nos applications se feront sur du calcul matriciel pour le traitement d'image. Le matériel utilisé est une carte de type Jetson, exploitée sous Ubuntu.

Introduction	1
0.Faire fonctionner votre GPU	1
1. Obfuscation	3
1.1 Premiers pas avec CUDA	3
1.2 Images RGB	4
2. Flou et convolutions	5
Etape 0	5
Etape 1	6
Conclusion	6

0.Faire fonctionner votre GPU

Dans cette première partie, nous dressons la liste des performances de l'unité de traitement graphique présente sur notre Jetson. Grâce à un script fourni par notre encadrant, M. Bordat, que nous compilons, nous pouvons ensuite exécuter le binaire "deviceQuery", qui donne les résultats suivants :

```

CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "NVIDIA Tegra X2"
  CUDA Driver Version / Runtime Version      10.0 / 10.0
  CUDA Capability Major/Minor version number: 6.2
  Total amount of global memory:             7860 MBytes (8241844224 bytes)
  ( 2 ) Multiprocessors, (128) CUDA Cores/MP: 256 CUDA Cores
  GPU Max Clock rate:                       1020 Mhz (1.02 GHz)
  Memory Clock rate:                        1300 Mhz
  Memory Bus Width:                         128-bit
  L2 Cache Size:                            524288 bytes
  Maximum Texture Dimension Size (x,y,z)    1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers 1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers 2D=(32768, 32768), 2048 layers
  Total amount of constant memory:           65536 bytes
  Total amount of shared memory per block:   49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                32
  Maximum number of threads per multiprocessor: 2048
  Maximum number of threads per block:       1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                      2147483647 bytes
  Texture alignment:                         512 bytes
  Concurrent copy and kernel execution:      Yes with 1 copy engine(s)
  Run time limit on kernels:                  No
  Integrated GPU sharing Host Memory:         Yes
  Support host page-locked memory mapping:   Yes
  Alignment requirement for Surfaces:         Yes
  Device has ECC support:                     Disabled
  Device supports Unified Addressing (UVA):   Yes
  Device PCI Domain ID / Bus ID / location ID: 0 / 0 / 0
  Compute Mode:
    < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 10.0, CUDA Runtime Version = 10.0, NumDevs = 1, Device0 = NVIDIA Tegra X2
Result = PASS

```

Fig. 1 : Résultats de l'exécution du binaire "deviceQuery", qui donne toutes les caractéristiques de notre GPU

Nous en déduisons ainsi les caractéristiques suivantes :

- 1) Il y a 256 ALUs (nom donné aux CUDA Cores).
- 2) Pour obtenir, le nombre d'opérations flottantes par seconde que peut effectuer notre GPU, nous effectuons le calcul suivant :

$$\text{FLOPS} = N_{\text{CUDA cores}} \times \text{GPU_Max_Clock_Rate} \times \text{FLOPs/cycle} = 256 * 1.02 * 10^9 * 3$$
 soit **7.8336.10¹¹ FLOPS**
- 3) La bande passante théorique du GPU est donnée par : Memory Bandwidth =

$$\text{Memory clock rate} * \text{bus width} = 1300 * 10^6 * 128 \text{ soit une bande passante de } 2.08.10^4 \text{ Mo/s.}$$
- 4) Après avoir exécuté la commande *lscpu*, nous obtenons les résultats suivants :

```

Architecture: aarch64
Byte Order: Little Endian
CPU(s): 6
On-line CPU(s) list: 0-5
Thread(s) per core: 1
Core(s) per socket: 3
Socket(s): 2
Vendor ID: ARM
Model: 3
Model name: Cortex-A57
Stepping: r1p3
CPU max MHz: 2035,2000
CPU min MHz: 345,6000
BogoMIPS: 62.50
L1d cache: 32K
L1i cache: 48K
L2 cache: 2048K
Flags: fp asimd evtstrm aes pmull sha1 sha2 crc32

```

Fig. 2 : Résultats de l'exécution de la commande lscpu

Le CPU a donc une fréquence maximale de 2,0352 GHz contre 1,02 GHz dans le cas du GPU. Cependant, dans le cas du calcul matriciel sur des matrices de taille dépassant la centaine, il reste beaucoup plus judicieux d'effectuer du calcul matriciel sur un GPU qui divise les traitements en différents blocs de threads qu'un CPU qui traite séquentiellement tous les éléments d'une matrice.

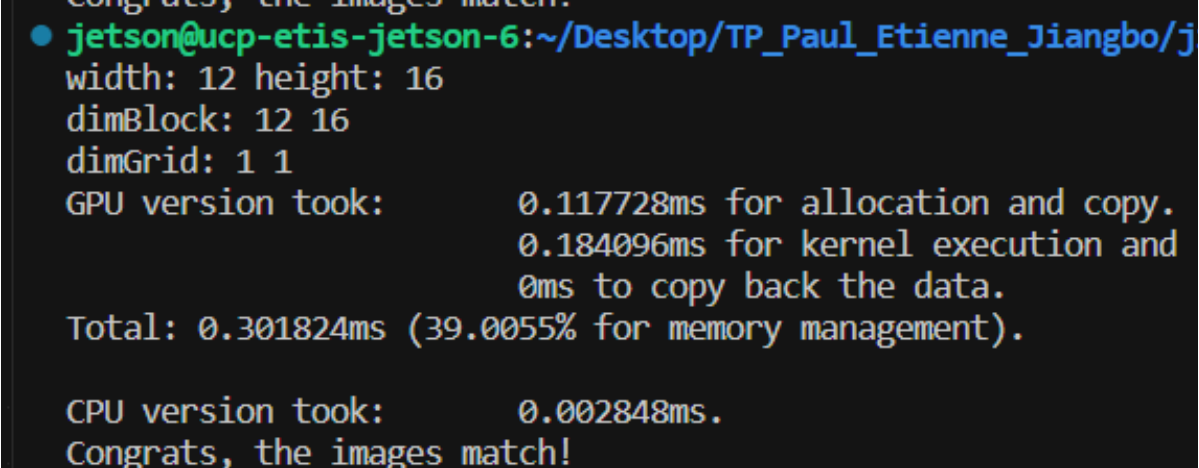
1. Obfuscation

1.1 Premiers pas avec CUDA

L'obfuscation est l'art de cacher les choses dans des signaux. Ici nous avons deux images, qui, prises séparément, ne représentent que du bruit. Pour en tirer leur secret, il va falloir les sommer, pixel par pixel.

Pour ce projet, nous utilisons cmake afin de générer le makefile dans un dossier build. La compilation est lancée via la commande classique make.

Après avoir implémenté la structure de base d'un programme CUDA qui fait fonctionner notre kernel sur un seul block, nous pouvons exécuter notre programme sur les images small_frag_1 et small_frag_2.



```
● jetson@ucp-etis-jetson-6:~/Desktop/TP_Paul_Etienne_Jiangbo/j
width: 12 height: 16
dimBlock: 12 16
dimGrid: 1 1
GPU version took:      0.117728ms for allocation and copy.
                    0.184096ms for kernel execution and
                    0ms to copy back the data.
Total: 0.301824ms (39.0055% for memory management).

CPU version took:      0.002848ms.
Congrats, the images match!
```

Fig. 3 : Résultats de l'exécution du code d'obfuscation sur les images de taille 12x16

- 1) Notre implémentation CUDA prend donc 0.301 ms dont 0.118 ms pour la gestion mémoire (39.01 %).
- 2) L'implémentation CPU prend 0.003 ms (facteur 100 entre les deux durées). En effet, le GPU ne procède qu'avec un seul bloc de threads et ne tire pas profit du traitement simultané de plusieurs blocs. Le traitement d'un bloc étant plus lent que l'exécution séquentielle sur CPU, cela explique les résultats observés.

1.2 Images RGB

Cette fois, nous travaillons avec des images RGB sur 32 bits qui sont trop importantes pour être traitées par un seul bloc de threads. Nous créons donc 25x25 blocs de taille 10x10.

- 1) L'implémentation CUDA prend 2.98 ms (91.1 % du temps pris par la gestion mémoire; 39.01 % dans le cas précédent).

```
jetson@ucp-etis-jetson-6:~/Desktop/TP_Paul_Etienne_Jiangbo/pau
width: 250 height: 250
dimBlock: 10 10
dimGrid: 25 25
GPU version took:      1.64618ms for allocation and copy.
                     0.26464ms for kernel execution and
                     1.06525ms to copy back the data.
Total: 2.97606ms (91.1077% for memory management).

CPU version took:      5.37587ms.
Congrats, the images match!
```

Fig. 4 : Résultats de l'obfuscation pour une grille de taille 25x25

- 2) L'implémentation sur CPU prend 5.38 ms et est donc 1,5 fois plus lente que l'exécution sur GPU : le traitement simultané de plusieurs blocs a pris l'avantage sur l'exécution séquentielle.
- 3) Si nous prenons une taille de blocs de 15x15 au lieu des 10x10 précédents, nous constatons que l'implémentation est moins efficace (3.29 ms contre les 2.98 ms précédentes). En effet, auparavant, nous avions `dimBlock.dimGrid = 250`. Maintenant, nous avons `dimBlock.dimGrid = 255`. Pour éviter les dépassements d'indice, nous avons implémenté une boucle IF qui vérifie que les index obtenus à partir des "threadids" ne sont pas supérieurs à 249. La sortie de la boucle IF à chaque fois que cette condition ne sera pas satisfaite va entraîner des traitements supplémentaires et donc un temps de traitement global plus important.

```
jetson@ucp-etis-jetson-6:~/Desktop/TP_Paul_Etienne_Jiangbo/pau
width: 250 height: 250
dimBlock: 15 15
dimGrid: 17 17
GPU version took:      1.39939ms for allocation and copy.
                     0.449792ms for kernel execution and
                     1.44288ms to copy back the data.
Total: 3.29206ms (86.3371% for memory management).

CPU version took:      5.01152ms.
Congrats, the images match!
```

Fig. 5 : Résultats de l'obfuscation pour une grille de taille 17x17

2. Flou et convolutions

Cette étape consiste à flouter une image. Pour ce faire, nous avons un tableau représentant les valeurs des pixels. Pour chaque pixel de l'image, on superpose un tableau de poids centré sur le pixel courant. Pour calculer la valeur du pixel flouté, on multiplie chaque pixel

avec le poids "au dessus". Pour finir, on somme tous les résultats des multiplications et on donne cette valeur au pixel en cours. On répète ce procédé pour tous les pixels de l'image.

Etape 0

Dans cette première étape, nous séparons les différents canaux de la matrice (R,G,B) pour avoir trois tableaux distincts contenant chacun les composantes R, G et B de l'image.

Nous commençons par un test de notre code sur l'image "cinque_terre_small.tiff".

```
→ build ./convolution 0 ../data/cinque_terre_small.tiff 1 2
numRows: 313, numCols: 557
Your code ran in: 25.245600 msecs.
PASS: Images match, congrats!
```

Fig. 6 : Validation du floutage sur l'image cinque_terre_small.tiff

- 1) Sur l'image cinque_terre_large.tiff, nous obtenons un temps de calcul de 480 ms pour des blocs de taille 10x10. Si en revanche nous construisons des blocs de taille 32x32, c'est-à-dire avec le nombre maximal de threads (1024), nous obtenons un temps de traitement légèrement plus faible (453 ms contre 477).

```
→ build ./convolution 0 ../data/cinque_terre_large.tiff 1 2
numRows: 3495, numCols: 4369
blockSize: 10, 10
gridSize: 437, 350
Your code ran in: 476.993713 msecs.
PASS: Images match, congrats!
Global error: 383
```

```
→ build ./convolution 0 ../data/cinque_terre_large.tiff 1 2
numRows: 3495, numCols: 4369
blockSize: 32, 32
gridSize: 137, 110
Your code ran in: 453.373108 msecs.
PASS: Images match, congrats!
Global error: 383
```

Fig. 7 : Comparaison des temps de traitement de cinque_terre_large.tiff pour différentes tailles de blocs

Etape 1

En utilisant la mémoire constante, qui est plus rapide que la mémoire globale, nous pouvons obtenir un speedup d'exécution de $(\text{Ancien_tps_de_calcul} - \text{Nouveau_tps_de_calcul}) / \text{Nouveau_tps_de_calcul} = (453-409)/409$ soit un **speedup de 11 %**.

```
→ build ./convolution 1 ../data/cinque_terre_large.tiff 1 400
numRows: 3495, numCols: 4369
blockSize: 32, 32
gridSize: 137, 110
Your code ran in: 409.023407 msecs.
PASS: Images match, congrats!
Global error: 383
```

Fig. 8 : Résultats du traitement avec l'utilisation de la mémoire constante

Conclusion

Ainsi, cette séance nous a permis d'apprendre à extraire les performances intéressantes de notre GPU (Bande passante mémoire, nombre d'opérations flottantes par seconde etc.). Nous avons également pu écrire des programmes complets CUDA de sommation puis de floutage par convolution d'images. A chaque fois, nous nous sommes penchés sur l'optimisation du temps de calcul en modifiant la taille des blocs pour éviter les "sauts" de boucle IF tout en exploitant le nombre maximum de threads par bloc. Nous n'avons malheureusement pas eu le temps de réaliser l'Etape n°2 de la partie "Convolution" par manque de temps qui utilise la mémoire partagée.