ENSTA – Systèmes parallèles (OS202)

TP5 – Programmation GPU avec PyCUDA : addition vectorielle/matricielle et ensemble de Mandelbrot

MENESES GAMBOA Carlos

École Nationale des techniques avancées
Février 2026

Table des matières

1 Introduction

Ce TP vise la prise en main d'un noyau CUDA via PyCUDA, l'indexation globale des threads et la parallélisation massive sur GPU.

Implémentations réalisées :

- addition de deux vecteurs `float32` sur GPU ;
- addition de deux matrices carrées `float32` sur GPU ;
- calcul de l'ensemble de Mandelbrot avec la règle « 1 thread = 1 pixel », puis comparaison CPU/GPU.

2 Environnement expérimental

Plateforme d'exécution : Google Colab avec GPU NVIDIA Tesla T4.

Langages et bibliothèques : Python, NumPy, PyCUDA, matplotlib/pylab.

Précision numérique :

- additions vectorielle/matricielle en `float32` ;
- Mandelbrot : référence CPU en `complex64`, noyau GPU en calcul réel `float32` pour les composantes $(\Re, \Im)$.

Ce choix peut induire des différences sur des trajectoires proches de la frontière de divergence.

Méthodologie de mesure temporelle :

- CPU : `time.perf_counter()` ;
- noyau GPU : `cuda.Event()` (temps noyau uniquement) ;
- copie Device→Host (DtoH) : `perf_counter` ;
- compilation PyCUDA : mesurée séparément (surcoût one-shot).

3 Rappels indispensables sur l'indexation CUDA

3.1 Indexation 1D (vecteurs / matrices aplaties)

$$i = \texttt{blockIdx.x} \times \texttt{blockDim.x} + \texttt{threadIdx.x}$$

Condition de sûreté :

```
if (i < n) { ... }
```

3.2 Nombre de blocs

$$\texttt{blocks} = \left\lceil \frac{n}{\texttt{threads\_per\_block}} \right\rceil$$

Le dernier bloc peut être partiel lorsque n n'est pas multiple de `threads_per_block`.

3.3 Indexation 2D (images / Mandelbrot)

$i = \texttt{blockIdx.x} \times \texttt{blockDim.x} + \texttt{threadIdx.x}, \qquad j = \texttt{blockIdx.y} \times \texttt{blockDim.y} + \texttt{threadIdx.y}$

Garde de sécurité :

```
if (i >= div || j >= div) return;
```

4 Exercice 1 – Addition vectorielle (`float32`)

4.1 Méthode

Les buffers GPU sont alloués via `mem_alloc`, les entrées sont copiées avec `memcpy_htod`, puis le résultat est rapatrié par `memcpy_dtoh`.

Le noyau `vadd` applique :

$$c[i] = a[i] + b[i]$$

avec index global 1D et garde `i < n`.

Le choix d'un problème avec $n > 256$ force l'usage de l'index global (au-delà d'un seul bloc).

4.2 Résultats

Valeurs observées (exactes) :

- $n = 1,000,000$
- `threads per block` $= 256$
- `total blocks` $= 3907$
- `max error` $= 0.0$

4.3 Analyse

$$\left\lceil \frac{1,000,000}{256} \right\rceil = 3907$$

$$3906 \times 256 = 999,936 \quad \Rightarrow \quad \text{reste} = 64 \text{ éléments}$$

Le bloc final est donc partiel. La valeur `max_err=0.0` est cohérente avec une opération élémentaire sans réduction, en `float32`, avec un appariement bit-à-bit plausible entre CPU et GPU.

5 Exercice 2 – Addition de matrices carrées (`float32`)

5.1 Méthode

Les matrices A et B ($N \times N$) sont aplaties en row-major (`ravel`) vers des vecteurs 1D contigus, ce qui permet de réutiliser un noyau 1D simple :

$$C[idx] = A[idx] + B[idx]$$

5.2 Résultats

Valeurs observées (exactes) :

- $N = 1024$
- $n = N \times N = 1,048,576$
- `tpb`$=256$
- `blocks`$=4096$
- `max_err`$=0.0$ (OK)

5.3 Analyse

Ici n est multiple de 256 :

$$\frac{1,048,576}{256} = 4096$$

Aucun bloc partiel n'est nécessaire. Comme pour l'exercice 1, l'addition élémentaire donne une concordance exacte (erreur maximale nulle).

6 Exercice final – Mandelbrot (1 thread = 1 pixel)

6.1 Problème et rappels mathématiques

Définition :

$$z_0 = 0, \qquad z_{n+1} = z_n^2 + c, \quad c \in \mathbb{C}$$

Le point c appartient à l'ensemble si la suite reste bornée.

Critère d'échappement :

$$|z_n| > 2 \;\Leftrightarrow\; |z_n|^2 = \Re(z_n)^2 + \Im(z_n)^2 > 4$$

Coloration (escape-time) :

- `color = n` à la première divergence ;
- sinon `color = loop + 5`.

Projection pixel → plan complexe :

$$cr = xmin + i\,dx, \qquad ci = ymin + j\,dy$$

$$dx = \frac{xmax - xmin}{div - 1}, \qquad dy = \frac{ymax - ymin}{div - 1}$$

6.2 Parallélisation GPU

La stratégie est « 1 thread = 1 pixel » : chaque thread calcule (i, j), itère $z \leftarrow z^2 + c$ jusqu'à `loop` ou divergence, puis écrit `color[j*div + i]`.

Configuration de lancement :

- `blockDim = (16,16)`
- `gridDim = (ceil(div/16), ceil(div/16))`

L'*early-exit* (`break` à la divergence) réduit le travail moyen par pixel, mais introduit de la divergence de flot de contrôle à l'intérieur des warps.

6.3 Résultats

Sortie console (copie conforme) :

```
Resolution: 1000 x 1000 = 1,000,000 pixels | loop=100
Block: (16,16) | Grid: (63,63)

CPU reference: 430.49 ms
GPU compile (info): 408.79 ms
GPU kernel: 0.25 ms
GPU DtoH copy: 1.25 ms
```

```
GPU total: 1.50 ms

Kernel throughput: 3952.69 MPix/s
Speedup (CPU/kernel): 1701.59 x
Speedup (CPU/total): 287.28 x

Max |color_gpu - color_cpu|: 104
Inside-set fraction (color=loop+5): 12.85 %
Mean escape iteration (capped): 15.28
```

TABLE 1 – Timings et speedups Mandelbrot (`div=1000`, `loop=100`).

Métrique	Valeur
CPU reference	430.49 ms
GPU compile (info)	408.79 ms
GPU kernel	0.25 ms
GPU DtoH copy	1.25 ms
GPU total (kernel + DtoH)	1.50 ms
Kernel throughput	3952.69 MPix/s
Speedup (CPU/kernel)	1701.59 ×
Speedup (CPU/total)	287.28 ×

6.4 Analyse critique

— Le noyau est très rapide (0.25 ms), mais la copie DtoH (1.25 ms) domine le temps GPU total mesuré ; pour 10^6 pixels en `int32`, le transfert représente environ 4 Mo.
— La compilation PyCUDA (408.79 ms) est un coût one-shot et ne doit pas être mélangée au temps de calcul en régime établi.
— Le débit est calculé par :

$$\text{MPix/s} = \frac{div \times div}{\text{kernel\_time (s)}} \times 10^{-6}$$

— Le speedup `CPU/kernel` surestime l'accélération utilisable si les transferts ne sont pas comptés ; `CPU/total` est plus représentatif d'un flux complet host+device.
— Les warnings CPU (`overflow encountered`, `invalid value encountered`) sont attendus sur des trajectoires qui divergent très vite en `complex64` ; le critère d'échappement stoppe souvent avant impact majeur, mais une référence CPU robuste doit être surveillée numériquement.
— `max_diff=104` est très probablement lié à un désalignement d'axes CPU/GPU : CPU construit `c_cpu = x[:,None] + 1j*y[None,:]` alors que GPU écrit `color[j*div+i]`. Correctif de validation stricte : comparer `color_gpu` à `color_cpu.T`, ou construire côté CPU `x[None,:] + 1j*y[:,None]`.

6.5 Figure

La Fig. 1 montre le rendu escape-time (`contourf`) obtenu en calcul GPU.

`plot.png` à compléter dans `docs/imgs/plot.png`.

FIGURE 1 – Ensemble de Mandelbrot (escape-time), résolution 1000×1000, `loop=100`, calcul GPU (1 thread/pixel).

7 Conclusion

Les additions vectorielle et matricielle valident la correctitude (erreur maximale nulle) avec une indexation globale CUDA conforme.

Sur Mandelbrot, l'accélération du noyau est très forte, mais l'accélération end-to-end est limitée par les transferts DtoH.

Axes d'amélioration :

- validation CPU/GPU cohérente sur l'orientation des axes ;
- réduction des transferts host/device (post-traitement conservé sur GPU) ;
- exploration systématique des tailles de blocs/grilles selon la charge et l'occupation.