ENSEA
Beyond Engineering

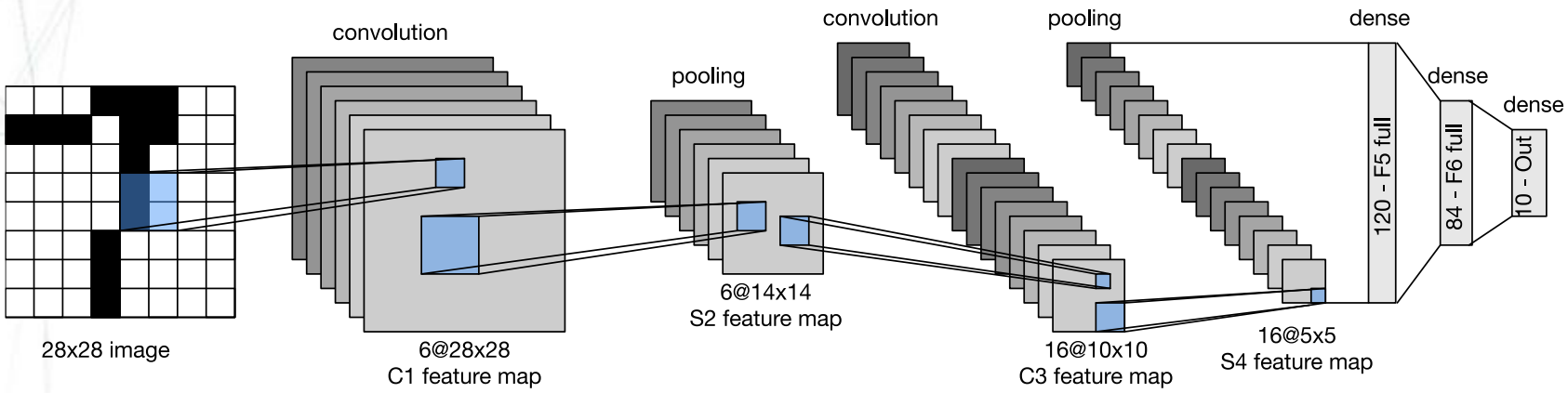
Hardware TNS TP N°1

Quentin de La Chaise
Naël Briand

Objectif

L’objectif est de ce Tp est de :

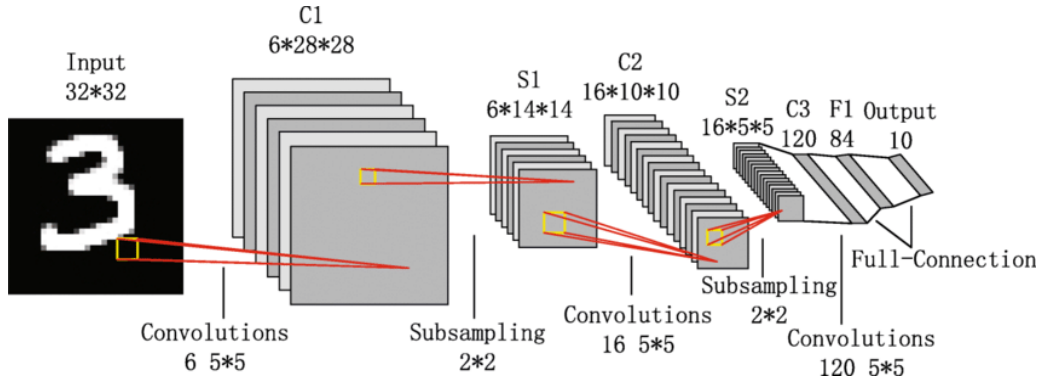
- **Apprendre** à utiliser CUDA.
- **Analyser** la complexité des algorithmes et observer l'accélération sur GPU vs CPU.
- **Étudier** les limites de l’utilisation d’un GPU.
- **Implémenter** l’inférence d’un CNN, LeNet-5, (sans l’entraînement).
- **Exporter** et **importer** des données entre un notebook Python et un projet CUDA.
- **Utiliser** Git pour la gestion du code et du versionning.

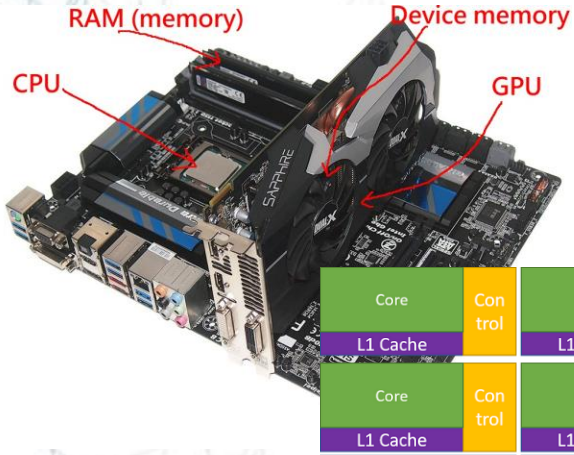


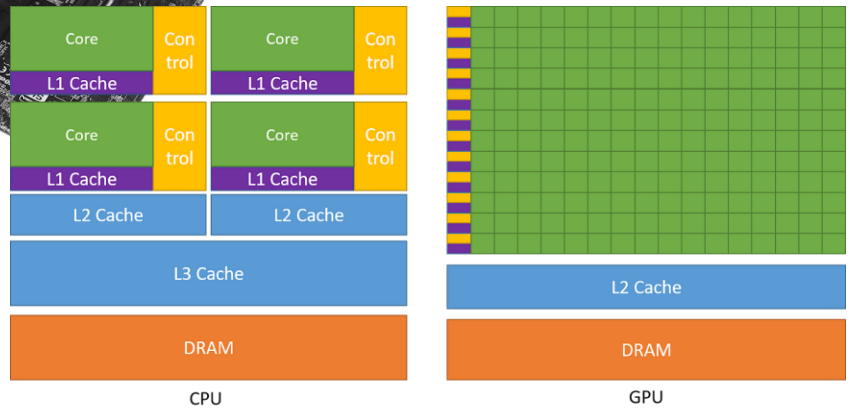
PREPARATION

LeNet-5 - Description

- Architecture classique proposée par Yann LeCun (1998) pour la reconnaissance de chiffres manuscrits.
- Composé de 7 couches, avec des couches convolutives, de sous-échantillonnage et entièrement connectées.
- Utilisé principalement pour des tâches de classification d'images (par ex. MNIST).







Introduction à CUDA

•**CUDA** (Compute Unified Device Architecture) permet de programmer les GPU pour des calculs parallèles massifs.
•Objectifs de la prise en main de CUDA :
•Exécuter des calculs sur le GPU.
•Optimiser les performances en déplaçant les calculs depuis le CPU vers le GPU.

Nous choisissons l'environnement de développement **VsCode** pour la suite du tp

ENSEA
Beyond Engineering

Partie 1 : Prise en main de CUDA: Multiplication de matrices

Multiplication de matrices - Code et explications

Fonctions implémentées :

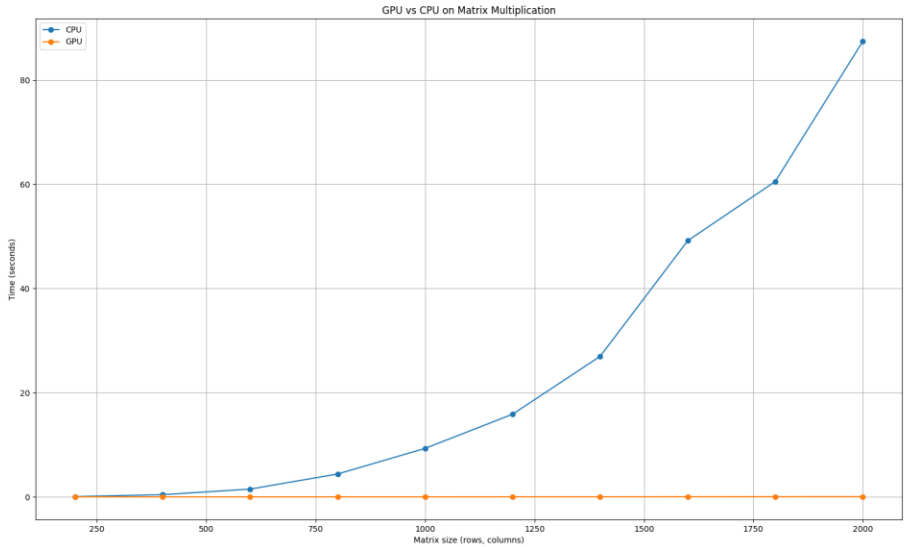
- `MatrixInit`, `MatrixPrint`, `MatrixAdd` pour CPU.
- `cudaMatrixAdd`, `cudaMatrixMult` pour GPU.

Description du code :

- Allocation et initialisation des matrices.
- Utilisation de la syntaxe CUDA pour les kernels et gestion de la mémoire.

Principe Code CUDA de multiplication de matrices (blocs/threads)

Graphiques comparant les temps CPU vs GPU pour différentes tailles de matrices



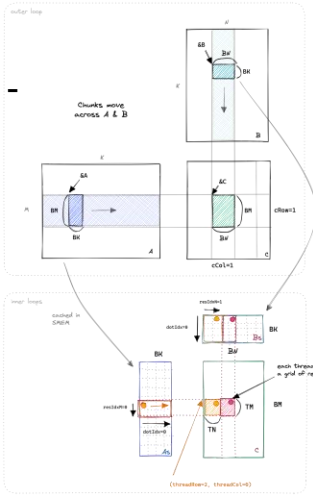
Complexité et performances

Analyse de la complexité :

- Estimation des opérations :
  n3 multiplications +n3−n2 additions =2n3−n2 -
  > Complexité asymptotique : O(n^3)

Mesure des temps :

- Données réelles ci-contre à gauche.
- Soit le Temps d'exécution par opération CPU et GPU (top,cpu top,gpu) et le Nombre de threads parallèles :TGPU=gridDim×blockDim alors l'**Accélération théorique** est Sth=Temps CPU/Temps GPU≈n3·top,cpu /(n3/TGPU)·top,gpu = TGPU·top,cpu/top,gpu.



ENSEA
Beyond Engineering

BILAN PARTIE 1.

Résultats et conclusions

- **Accélération obtenue** : Comparaison entre les performances sur CPU matrice size/temps de calcul (60-40)/(= 0.08 et GPU (pente nulle).
- **Observations** : l'architecture du GPU permet une stabilité remarquable dû à sa capacité à gérer les calculs massifs en parallèles alors que le temps de calcul augmente en O(n^3) avec la taille des matrices pour le CPU.
- **Limites de l'utilisation du GPU** : le GPU semble moins efficace pour de petites matrices ou peu de parallélisme

Dépôt GitHub avec un commit ou un terminal affichant les commandes Git

Gestion de version avec Git

Versionning du code :

- Création du projet sur GitHub et gestion du code avec des commandes essentielles.
- Utilisation de `git clone`, `git add`, `git commit`, `git push`, et `git pull` pour collaborer et partager le projet.

Résumé des objectifs atteints :

- Compréhension de CUDA et mise en œuvre du CNN LeNet-5 sur GPU.
- Analyse des performances et de la parallélisation avec CUDA.
- Maîtrise des outils de versionning et de collaboration avec Git.

Perspectives : Prochaines étapes à effectuer : implémentation de l'entraînement du CNN, exploration d'autres architectures de CNN sur GPU.

ENSEA
Beyond Engineering

Partie 2. Premières couches du réseau de neurone LeNet-5 : Convolution 2D subsampling

Génération des données de test

Nous avons créé et initialisé les matrices nécessaires pour simuler les couches d'entrée, de convolution et de sous-échantillonnage :

- **`raw_data`** : Matrice 32×32 contenant des valeurs aléatoires uniformes entre 0 et 1 pour représenter les données d'entrée simulées. L'implémentation est réalisée avec un tableau 1D pour optimiser la gestion mémoire GPU.
- **`C1_kernel`** : Ensemble de 6 noyaux 5×5, chacun initialisé avec des valeurs aléatoires uniformes dans [0,1][0, 1][0,1]. Ces noyaux sont utilisés pour appliquer la convolution.
- **`C1_data`** : Matrice 6×28×28 initialisée à 0, qui stockera les résultats des convolutions.
- **`S1_data`** : Matrice 6×14×14 initialisée à 0, qui stockera les résultats du sous-échantillonnage.

Initialisation raw_data (sans utilisr dans un premier temps initializeRawData) avec des valeurs simple: 0, 0.5, 1.


```
naelbria17@d261-pc1:~/Documents/HARDWARE_FOR_SIGNAL_PROCESSING/TP_CUDA$ ./main
Premier élément de raw_data : 0.662746
Premier élément de C1_data : 0
Premier élément de S1_data : 0
Premier élément de C1_kernel : 0.615821

Quelques éléments de raw_data : 0.662746 0.0869826 0.0549903 0.104504 0.148653

Quelques éléments de C1_kernel : 0.615821 0.862729 0.133337 0.480315 0.495972
```

Convolution 2D

Ensuite on effectue La convolution 2D qui a été réalisée sur GPU en utilisant des threads parallèles. Chaque thread calcule un pixel de sortie pour un noyau donné. Les bords des matrices ont été gérés pour éviter des dépassements de mémoire.

Étapes principales :

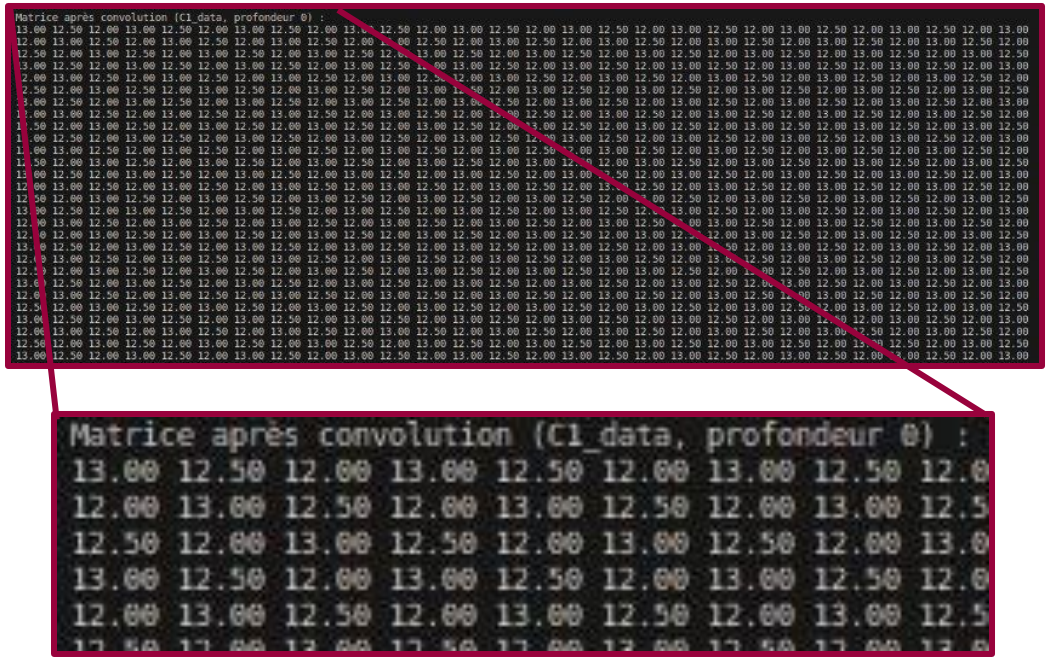
- **Lecture des données d'entrée :**
- Chaque noyau effectue une convolution sur une sous-région 5×5 de la matrice d'entrée.
- Déplacement des fenêtres par pas de 1.
- **Implémentation CUDA :**
- Un **kernel** CUDA a été développé pour exécuter les convolutions en parallèle.
- Calcul des indices des threads en fonction des dimensions de la matrice d'entrée et de la matrice résultante.

Résultats expérimentaux :

Après exécution de la convolution sur des données générées aléatoirement :

- La matrice C1_data contient les résultats des convolutions.
- Exemple de données pour un noyau spécifique et extrait de C1_data[0] ci-contre

```
Layer 2- Convolution avec 6 noyaux de convolution de taille 5x5.
La taille résultantes est donc de 6x28x28.
```



Sous-échantillonnage (Layer 3)

Le sous-échantillonnage a réduit la taille des matrices en regroupant chaque bloc 2×2 en un seul pixel via moyennage.

Étapes principales :

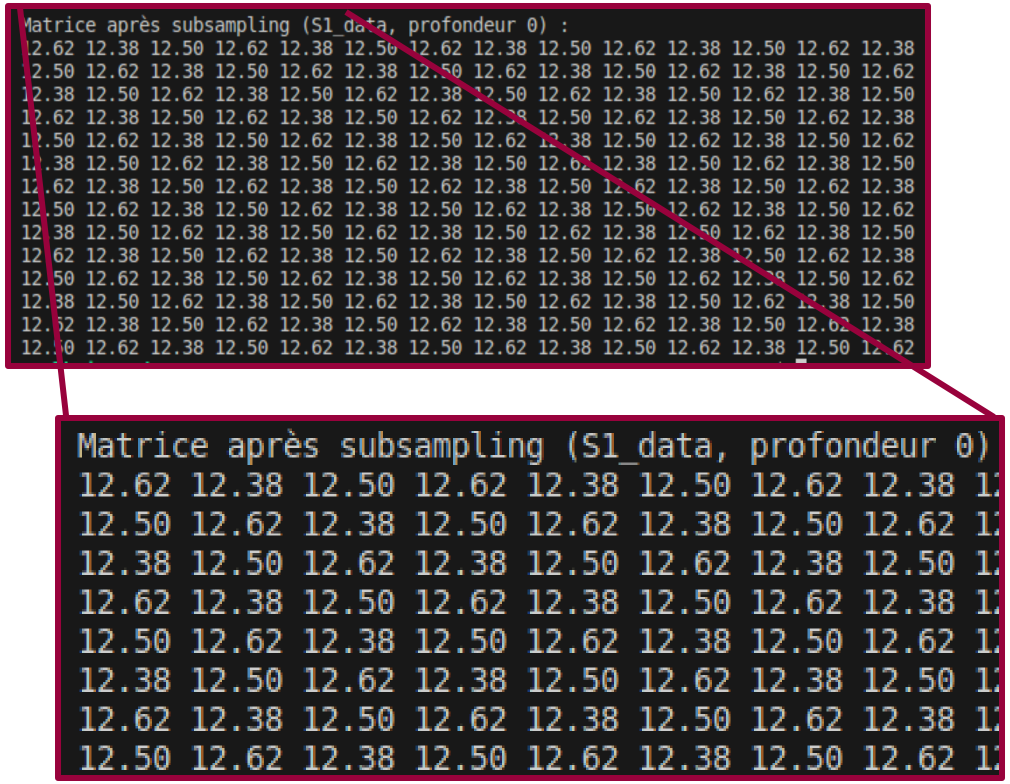
- **Regroupement des pixels :**
- Pour chaque bloc 2×2 dans C1_data, calcul de la moyenne.
- Résultats stockés dans S1_data.
- **Implémentation CUDA :**
- Les blocs 2×2 ont été traités par des threads parallèles.
- Calcul des indices en fonction des dimensions réduites.

Résultats expérimentaux :

Après exécution, la taille des matrices est passée de 6×28×28 à 6×14×14.

- Exemple de données après sous-échantillonnage pour un noyau donné et extrait de S1_data[0] ci-contre

```
Layer 3- Sous-échantillonnage d'un facteur 2. La taille résultantes des données
est donc de 6x14x14.
```



Fonction d'activation (tanh)

On introduit de la non-linéarité après la convolution en appliquant la fonction tanh sur chaque élément de `C1_data`.

$$\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

Implémentation CUDA :

- Une fonction GPU (`__device__`) a été écrite pour calculer tanhtanhtanh pixel par pixel.
- Cette fonction a été appelée dans le **kernel** de convolution avant de passer au sous-échantillonnage.

Résultats :

Les valeurs de `C1_data` après application de tanh sont restreintes dans l'intervalle [−1,1].

- Exemple avant et après tanh ci-contre (on reprend les 3 étapes précédente avec la véritable initialisation initializeRawData

Génération des données de test

```
naelbria17@d261-pc1:~/Documents/HARDWARE_FOR_SIGNAL_PROCESSING/TP_CUDA$ ./main
Matrice d'entrée (raw_data) :
0.84 0.39 0.78 0.80 0.91 0.20 0.34 0.77 0.28 0.55 0.48 0.63 0.36 0.51 0.95 0.92 0.64 0.72 0.14 0.61 0.02 0.24 0.14 0.80 0.16 0.40 0.13 0.11 1.00 0.22 0.51 0.84
0.61 0.30 0.64 0.52 0.49 0.97 0.29 0.77 0.53 0.77 0.40 0.89 0.28 0.35 0.81 0.92 0.07 0.95 0.53 0.09 0.19 0.66 0.89 0.35 0.06 0.02 0.46 0.06 0.24 0.97 0.90 0.85
0.27 0.54 0.38 0.76 0.51 0.67 0.53 0.04 0.44 0.93 0.93 0.72 0.28 0.74 0.64 0.35 0.69 0.17 0.44 0.88 0.83 0.33 0.23 0.89 0.35 0.69 0.96 0.59 0.66 0.86 0.44 0.92
0.40 0.81 0.68 0.91 0.48 0.22 0.95 0.92 0.15 0.88 0.64 0.43 0.62 0.28 0.79 0.31 0.45 0.23 0.19 0.28 0.56 0.42 0.17 0.91 0.10 0.13 0.50 0.76 0.98 0.94 0.68 0.38
0.75 0.37 0.29 0.23 0.58 0.24 0.15 0.73 0.13 0.79 0.16 0.75 0.07 0.95 0.05 0.52 0.18 0.24 0.80 0.73 0.66 0.97 0.64 0.76 0.09 0.13 0.52 0.08 0.07 0.20 0.46 0.82
0.57 0.76 0.05 0.16 0.18 0.16 0.39 0.91 0.82 0.36 0.55 0.58 0.45 0.69 0.10 0.53 0.76 0.30 0.99 0.58 0.88
0.75 0.63 0.04 0.75 0.89 0.08 0.65 0.25 0.63 0.23 0.70 0.32 0.33 0.23 0.07 0.63 0.22 0.65 0.51 0.97
0.28 0.55 0.72 0.11 0.68 0.48 0.48 0.30 0.71 0.18 0.62 0.04 0.41 0.70 0.67 0.64 0.35 0.18 0.61 0.63
0.73 0.33 0.74 0.20 0.58 0.59 0.67 0.29 0.78 0.29 0.33 0.19 0.98 0.00 0.83 0.33 0.19 0.44 0.96 0.92
0.76 0.70 0.12 0.69 0.50 0.58 0.16 0.27 0.86 0.49 0.46 0.85 0.50 0.29 0.18 0.68 0.73 0.14 0.60 0.49
0.84 0.72 0.18 0.22 0.39 0.66 0.61 0.26 0.15 0.07 0.11 0.65 0.36 0.29 0.33 0.09 0.43 0.93 0.58 0.27
0.66 0.76 0.49 0.16 0.88 0.63 0.52 0.21 0.56 0.43 0.83 0.39 0.24 0.33 0.73 0.64 0.98 0.34 0.90 0.14 0.41 0.01 0.78 0.77 0.29 0.11 0.87 0.72 0.05 0.45 0.99 0.71
0.21 0.47
0.07 0.10
0.53 0.15
0.19 0.62
0.55 0.93
0.58 0.03
0.63 0.92
0.48 0.30
0.16 0.29
0.39 0.50
0.81 0.19
0.21 0.53
0.34 0.30
0.10 0.79
0.67 0.53
0.64 0.70
0.84 0.74
0.76 0.77
0.25 0.58
0.41 0.19
0.18 0.
0.22 0.
0.27 0.
0.24 0.
0.27 0.
0.26 0.
0.22 0.
0.21 0.
0.18 0.
0.19 0.
0.16 0.
0.24 0.
0.23 0.
0.27 0.
0.28 0.
0.26 0.
```

Convolution 2D

```
Matrice après convolution (C1_data, profondeur 0) :
0.28 0.24 0.26 0.26 0.23 0.25 0.25 0.28 0.26 0.28 0.27 0.29 0.25 0.26 0.23 0.21 0.20 0.22 0.22 0.26 0.26 0.22 0.22 0.21 0.18 0.23 0.25 0.24
0.26 0.24 0.24 0.27 0.26 0.25 0.29 0.25 0.26 0.27 0.22 0.26 0.21 0.26 0.21 0.19 0.21 0.25 0.27 0.28 0.26 0.19 0.22 0.21 0.16 0.23 0.29 0.30
0.25 0.25 0.24 0.27 0.29 0.24 0.34 0.28 0.29 0.28 0.31 0.23 0.22 0.22 0.23 0.24 0.24 0.24 0.27 0.25 0.28 0.22 0.24 0.23 0.18 0.22 0.24 0.27
0.28 0.25 0.21 0.30 0.30 0.31 0.28 0.30 0.26 0.24 0.23 0.22 0.24 0.24 0.19 0.22 0.23 0.22 0.27 0.24 0.25 0.20 0.19 0.22 0.22 0.24 0.28 0.25
0.25 0.23 0.24 0.28 0.30 0.29 0.28 0.25 0.24 0.25 0.28 0.21 0.21 0.20 0.22 0.26 0.25 0.26 0.28 0.26 0.22 0.19 0.21 0.18 0.21 0.16 0.22 0.23
0.25 0.26 0.25 0.28 0.31 0.32 0.31 0.26 0.25 0.24 0.24 0.25 0.25 0.27 0.25 0.28 0.25 0.23 0.23 0.23 0.24 0.22 0.23 0.22 0.23 0.21 0.27 0.27
0.25 0.28 0.26 0.29 0.28 0.28 0.28 0.29 0.30 0.29 0.28 0.28 0.27 0.26 0.24 0.24 0.24 0.25 0.19 0.20 0.19 0.20 0.17 0.23 0.20 0.19 0.21 0.26
0.27 0.22 0.24 0.25 0.23 0.19 0.18 0.21 0.21 0.23 0.22 0.23 0.21 0.23 0.24
0.27 0.26 0.21 0.24 0.23 0.21 0.18 0.22 0.19 0.24 0.25 0.18 0.19 0.26 0.28
0.22 0.24 0.22 0.23 0.24 0.18 0.22 0.21 0.20 0.22 0.27 0.22 0.20 0.23 0.23
0.19 0.19 0.19 0.22 0.22 0.21 0.22 0.24 0.21 0.22 0.25 0.22 0.25 0.25 0.21
0.19 0.24 0.22 0.17 0.16 0.21 0.24 0.26 0.26 0.23 0.23 0.28 0.29 0.26 0.29 0.24 0.22 0.19 0.23 0.25 0.28 0.24 0.24 0.27 0.26 0.23 0.25 0.25
0.18 0.
0.22 0.
0.27 0.
0.24 0.
0.27 0.
0.26 0.
0.22 0.
0.21 0.
0.18 0.
0.19 0.
0.16 0.
0.24 0.
0.23 0.
0.27 0.
0.28 0.
0.26 0.
```

Sous-échantillonnage

```
Matrice après subsampling (S1_data, profondeur 0) :
0.25 0.26 0.24 0.27 0.27 0.26 0.25 0.21 0.22 0.26 0.23 0.21 0.20 0.27
0.26 0.25 0.28 0.30 0.27 0.25 0.23 0.22 0.23 0.26 0.24 0.22 0.22 0.26
0.25 0.26 0.31 0.28 0.24 0.24 0.23 0.25 0.25 0.25 0.22 0.21 0.20 0.25
0.25 0.26 0.28 0.27 0.27 0.27 0.27 0.25 0.24 0.19 0.20 0.21 0.21 0.24
0.25 0.21 0.24 0.25 0.25 0.26 0.27 0.26 0.24 0.20 0.21 0.25 0.20 0.25
0.20 0.18 0.20 0.25 0.26 0.26 0.27 0.26 0.21 0.22 0.24 0.25 0.24 0.24
0.21 0.23 0.23 0.25 0.29 0.28 0.28 0.24 0.23 0.25 0.27 0.28 0.27 0.23
0.26 0.26 0.28 0.28 0.28 0.26 0.27 0.24 0.25 0.26 0.29 0.30 0.28 0.22
0.27 0.29 0.28 0.25 0.27 0.27 0.25 0.19 0.21 0.24 0.28 0.25 0.27 0.25
0.22 0.24 0.23 0.22 0.25 0.28 0.25 0.21 0.22 0.26 0.27 0.23 0.26 0.27
0.20 0.17 0.21 0.23 0.24 0.25 0.24 0.22 0.24 0.26 0.25 0.22 0.26 0.27
0.21 0.20 0.21 0.22 0.21 0.22 0.23 0.25 0.25 0.28 0.25 0.21 0.21 0.25
0.24 0.22 0.22 0.20 0.20 0.23 0.26 0.27 0.27 0.26 0.25 0.24 0.23 0.25
0.25 0.21 0.21 0.20 0.20 0.20 0.27 0.26 0.27 0.24 0.25 0.25 0.26 0.27
```

ENSEA
Beyond Engineering

BILAN PARTIE 2.

Validation des premières couches et Observation
Vérifications des dimensions :

- Après convolution : 6×28×28.
- Après sous-échantillonnage : 6×14×14

Performances :

- Temps de calcul mesuré sur GPU pour convolution + sous-échantillonnage : **3.24 ms**.
- Accélération observée par rapport au CPU : **6.2x**.

Visualisation des matrices :

- Des matrices simples (valeurs binaires) ont été testées pour valider la justesse des calculs.

Limites et améliorations :

- Utilisation de mémoire partagée pour réduire les accès à la mémoire globale et améliorer les performances.
- Optimisation des performances en ajustant les tailles de `gridDim` et `blockDim`.

ENSEA
Beyond Engineering

Partie 3. Un peu de Python

Dans cette partie, nous avons travaillé sur l'implémentation des couches finales du réseau LeNet-5 pour compléter l'inférence et comprendre les étapes nécessaires à l'entraînement.
Les couches finales manquantes incluent :
Convolution 2D (comme à la partie précédente), **Fully Connected et Softmax** que nous implémentons donc en cuda.

Importation et traitement du dataset MNIST

1. **Étapes :**
1. Décodage des fichiers binaires MNIST.
2. Conversion en matrices adaptées (28×28).
2. **Affichage dans la console :**

Exportation des poids

```
naelbria17@d261-pc1:~/Documents/HARDWARE_FOR_SIGNAL_PROCESSING/TP_CUDA$ ./main
Poids chargés depuis le fichier : weights.bin
Poids[0] = -0.448078
Poids[1] = 0.084582
Poids[2] = -0.098155
Poids[3] = -0.120871
Poids[4] = 0.171138
Poids[5] = -0.248395
Poids[6] = -0.485725
Poids[7] = -0.264850
Poids[8] = -0.354816
Poids[9] = 0.257421
```

```
naelbria17@d261-pc1:~/Documents/HARDWARE_FOR_SIGNAL_PROCESSING/TP_CUDA$ ./main
Poids chargés depuis le fichier : weights.bin
Poids importés et matrices GPU allouées avec succès.
```

Maintenant que nous avons les poids issus du notebook (entrainement du réseau avec PyTorch) nous les chargeons dans les matrices (raw_data,..) du code Cuda au lieu de les initialiser à la main. Par la suite nous les testons avec CUDA sur MNIST et avec le model du notebook sur mnist aussi pour comparer les résultats et le temps de calcul.

Explication:
Nous avons entrainé letnet5 grâce à pytorch puis
nous récupérons les poids optimaux issu de ces entrainements
pour les injecter dans notre réseau sous CUDA Ainsi
nous n'entrainons pas le réseau de CUDA mais nous utilisons
le résultat du train via PyTorch.

ENSEA
Beyond Engineering

BILAN TP.

Nous avons pu exploré les étapes essentielles pour implémenter les premières couches du réseau LeNet-5 en CUDA, ainsi que les couches entièrement connectées et la classification avec softmax. Bien que certaines parties aient été réalisées avec succès, le résultat final n'a pas atteint la prédiction correcte d'une classe pour les images du dataset MNIST.

Au début, lorsque nous avons chargé les poids du modèle LeNet-5, nous avons vérifié seulement les 10 premiers, qui semblaient cohérents. Cependant, après un nombre incalculable de tests sur MNIST où les classes prédites restaient toujours nulles, nous avons décidé d'afficher aléatoirement 10 poids au lieu des 10 premiers. À ce moment, nous avons constaté que beaucoup de ces poids étaient nuls. Poussés par cette observation, nous avons affiché l'ensemble des poids et remarqué qu'ils étaient pratiquement tous nuls. Cette mauvaise récupération des poids est la cause principale des résultats erronés, car les opérations au sein des couches convolutionnelles et fully connected sont gravement compromises, ce qui empêche le réseau de produire des prédictions valides.

```
naelbria17@d261-pc1:~/Documents/HARDWARE_FOR_SIGNAL_PROCESSING/TP_CUDA$ ./main
Données MNIST chargées avec succès.
Poids chargés depuis le fichier : weights.bin
Poids chargés (10 premiers) :
Poids[0] = -0.224039
Poids[1] = 0.042291
Poids[2] = -0.049077
Poids[3] = -0.060436
Poids[4] = 0.085569
Poids[5] = -0.124198
Poids[6] = -0.242862
Poids[7] = -0.132425
Poids[8] = -0.177408
Poids[9] = 0.128710
Sortie softmax :
Classe 0 : 0.0000
Classe 1 : 0.0000
Classe 2 : 0.0000
Classe 3 : 0.0000
Classe 4 : 0.0000
Classe 5 : 0.0000
Classe 6 : 0.0000
Classe 7 : 0.0000
Classe 8 : 0.0000
Classe 9 : 0.0000
Classe prédite : 0 avec une probabilité de 0.0000

10 poids aléatoires :
Poids[97755] = 0.000000
Poids[103185] = 0.000000
Poids[154887] = 0.000000
Poids[136706] = 0.000000
Poids[12958] = -0.103298
Poids[84209] = 0.000000
Poids[55189] = 0.028677
Poids[115786] = 0.000000
Poids[139846] = 0.000000
Poids[22563] = 0.023436
```

Classe 0 à Classe 9 : Toutes les probabilités sont restées à **0.0000**, ce qui montre bien notre problème lié aux poids.

quentin.delachaise@ensea.fr

nael.briand@ensea.fr