TP : Implémentation d'un CNN - LeNet-5 sur GPU

Objectifs & Méthodes de travail

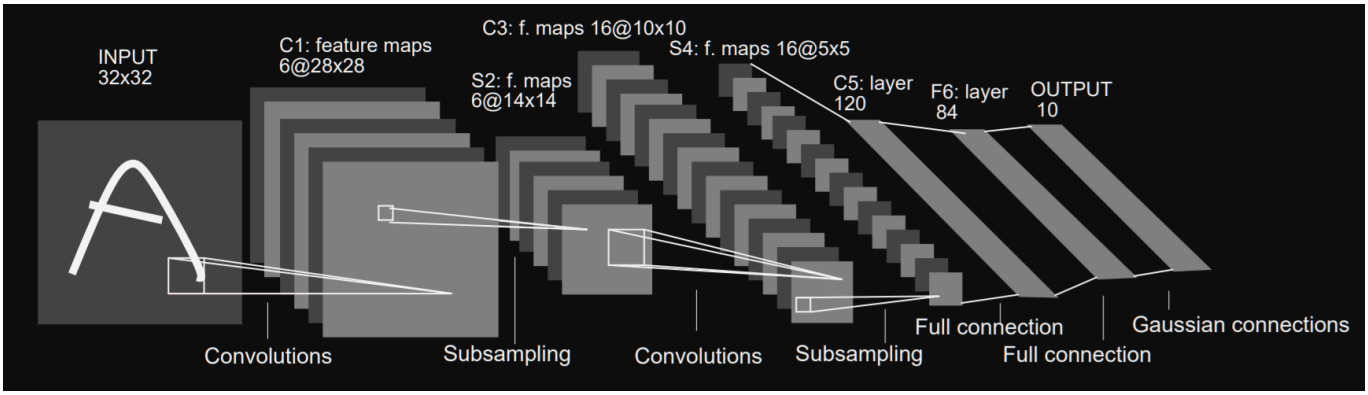
Objectifs Les objectif de ces 4 séances de TP de HSP sont :

- Apprendre à utiliser CUDA,
- Etudier la complexité de vos algorithmes et l'accélération obtenue sur GPU par rapport à une éxécution sur CPU,
- Observer les limites de l'utilisation d'un GPU,
- Implémenter "from scratch" un CNN : juste la partie inférence et non l'entrainement,
- Exporter des données depuis un notebook python et les réimporter dans un projet cuda,
- Faire un suivi de votre projet et du versionning à l'outil git,

Implémentation d'un CNN

L'objectif à terme de ces 4 séances est d'implémenter l'inférence dun CNN très claissque : LeNet-5 proposé par Yann LeCun et al. en 1998 pour la reconnaissance de chiffres manuscrits.

La lecture de cet article vous apportera les informations nécessaires pour comprendre ce réseau de neurone.



https://www.datasciencecentral.com/profiles/blogs/lenet-5-a-classic-cnn-architecture

Layer 3 Attention, contraitement à ce qui est décrit dans l'article, la 3eme couche du CNN prendra en compte tous les features pour chaque sortie

Suivi du projet - Git

Comme tous projets à l'Ensea, vous allez écrire du code, partagez vos réalisations et surtout faire du versioning.

Si vous n'avez pas encore de compte github (ou autre plateforme git), je vous laisse suivre ce petit tuto et créer votre compte : Allez directement à la partie "create your identity" du tutorial suivant : https://www.unixmen.com/use-git-commands-linux-terminal/

Vous avez 5 commandes à connaitre : Pour télécharger un projet git depuis Github : "git clone " Pour rajouter des fichiers à synchroniser : "git add ." depuis votre dossier de travail Pour préparer votre dossier à envoyer

sur github en ajoutant un commentaire sur les mises à jour : "git commit -m "" " Pour synchroniser votre git avec github : "git push" Pour mettre à jour vote repository sur votre PC depuis github : "git pull"

Pour partager votre projet, deux possibilités s'offrent à vous : Rendre votre dépot public et juste partager le lien. Rendre votre dépot privé et ajouter les personnes avec qui vous souhaitez partager votre projet.

Quel IDE utiliser ?

Quel IDE pour faire du CUDA ? Peu d'IDE propose une gestion de CUDA propre mais une coloration synthaxique du langage C conviendra très bien. Vous êtes libre dans le choix de l'IDE pour développer en CUDA mais la compilation et l'éxécution se feront via la console. Un simple éditeur de texte fera donc l'affaire. Cependant pour des questions d'ergonomie, je vous déconseille de coder directement dans la console avec "nano" ou "vim". Vim est très puissant pour ceux qui savent s'en servir 😉 mais je ne suis pas assez expert pour être à l'aise avec.

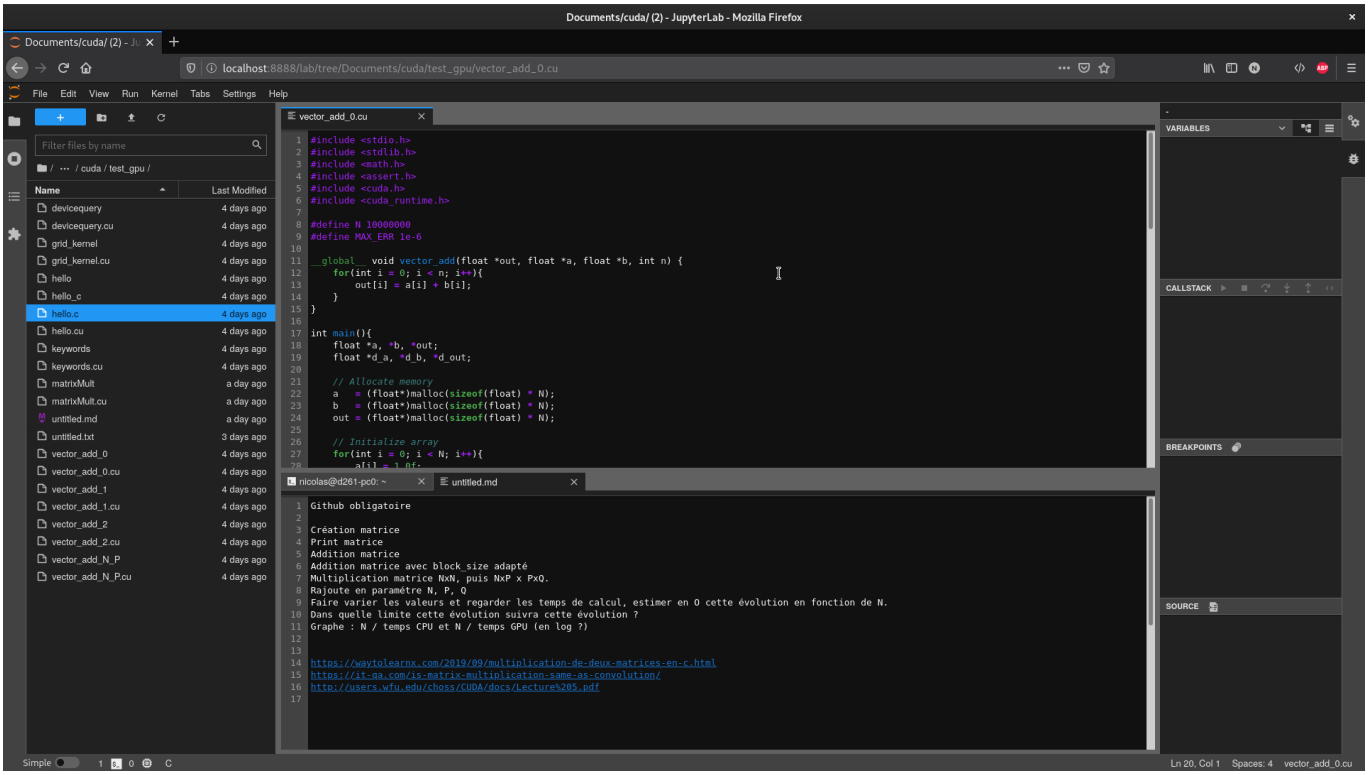
VsCode VsCode est déjà installé sur les machines et propose nativement une compilation avec nvcc pour éxécuter des codes .cu

CLion Idem que VsCode, CLion comprend parfaitement le Cuda mais n'est pas installé dans votre

Jupyter-Lab Jupyter-lab est une option très satisfaisante, vous pouvez suivre le protocole d'installation sur la page de ce cours (lien) en l'adaptant au langage étudié. Les avantages sont multiples :

- Vous pouvez reprendre votre projet d'où vous voulez,
- Vous pouvez transférer vos fichiers facilement,
- Vous pouvez y intégrer une console pour la compilation et l'éxécution,
- Vous pouvez coder dans d'autres langages facilement (matlab, octave, python avec/sans GPU : vous pouvez même importer votre notebook, etc...)
- La listes des noyaux disponibles : https://github.com/jupyter/jupyter/wiki/Jupyter-kernels

Votre interface de travail pourrait ressembler facilement à cela.



Ressources

Pour ce TP, vous aurez besoin (Moodle) de :

- **LeNet5.ipynb**: le code python de l'implémentation du CNN LeNet-5.
- **train-images.zip**: les images d'entrainement du CNN.
- **printMNIST.cu**: un programme cuda qui permet d'afficher une image du dataset.

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

Dans toute cette partie, ne faites qu'un seul fichier .cu

Multiplication de matrices

Afin de prendre en main toutes les subtilités de CUDA, vous allez devoir mettre en place des fonctions d'opération sur des matrices. Paramètres :

- n : nombre de lignes de la matrice,
- p : nombre de colonnes de la matrice si n différent de p,
- M : pointeur de la matrice

Allocation de mémoire

L'allocation de la mémoire (`malloc`) se fera dans votre fonction principale main.

Création d'une matrice sur CPU

Cette fonction initialise une matrice de taille n x p Initialisez les valeurs de la matrice de façon aléatoire entre -1 et 1.

```
void MatrixInit(float *M, int n, int p)
```

Affichage d'une matrice sur CPU

Cette fonction affiche une matrice de taille n x p

```
void MatrixPrint(float *M, int n, int p)
```

Addition de deux matrices sur CPU

Cette fonction additionne deux matrices M1 et M2 de même taille n x p

```
void MatrixAdd(float *M1, float *M2, float *Mout, int n, int p)
```

Addition de deux matrices sur GPU

Cette fonction additionne deux matrices M1 et M2 de même taille n x p Vous pouvez considérer les dimensions des matrices comme les paramètres gridDim et blockDim : les lignes correspondent aux blocks, les colonnes correspondent aux threads

```
__global__ void cudaMatrixAdd(float *M1, float *M2, float *Mout, int n, int p)
```

Multiplication de deux matrices NxN sur CPU

Cette fonction multiplie 2 matrices M1 et M2 de taillle n x n

```
void MatrixMult(float *M1, float *M2, float *Mout, int n)
```

Ne pas dépasser N=1000 sinon les calculs vont commencer à être très long.

Multiplication de deux matrices NxN sur GPU

Cette fonction multiplie 2 matrices M1 et M2 de taillle n x n Vous pouvez considérer les dimensions des matrices comme les paramètres gridDim et blockDim : les lignes correspondent aux blocks, les colonnes correspondent aux threads

```
__global__ void cudaMatrixAdd(float *M1, float *M2, float *Mout, int n)
```

Ici N = 10k est réalisable.

Compléxité et temps de calcul

Pour chaque opération (addition et multiplication) :

- Estimer la compléxité des opérations (ie le nombre d'opération nécessaire pour faire le calcul),
- L'accélération théorique obtenue en effectuant ces opérations sur GPU au lieu de CPU,
    - Mesurer le temps CPU (cf <time.h>),
    - Mesurer le temps GPU (avec <time.h> ou nvprof),
- En mesurant l'accélération réellement obtenue :
    - Confronter les résultats avec les caractéristiques du GPU utilisé (cf cours),
    - Faites varier les tailles et dimension des grid et block (par exemple gridDim = 1 et blockDim = (nbr ligne x nbr colonne),

Paramétrage de votre programme

Transformer votre programme pour que la dimension des matrices soient prises en argument du programme et non fixée à la compilation.

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

L'architecture du réseau LeNet-5 est composé de plusieurs couches :

- **Layer 1** - Couche d'entrée de taille 32x32 correspondant à la taille des images de la base de donnée MNIST
- **Layer 2** - Convolution avec 6 noyaux de convolution de taille 5x5. La taille résultantes est donc de 6x28x28.
- **Layer 3** - Sous-échantillonnage d'un facteur 2. La taille résultantes des données est donc de 6x14x14.

Layer 1 - Génération des données de test

Nous n'allons pas tout de suite travailler sur les données de la base de donnée MNIST.

Pour l'instant, vous devez générez des matrices correspondant à ce que l'on a besoin :

- Une matrice float raw_data de taille 32x32 initialisé avec des valeurs comprises entre 0 et 1, correspondant à nos données d'entrée.
- Une matrice float C1_data de taille 6x28x28 initialisé à 0 qui prendra les valeurs de sortie de la convolution 2D. C1 correspond aux données après la première Convolution.
- Une matrice float S1_data de taille 6x14x14 intialisé à 0 qui prendra les valeurs de sortie du sous-échantillonnage. S1 correspond aux données après le premier Sous-échantillonnage.
- Une matrice float C1_kernel de taille 6x5x5 initialisé à des valeurs comprises entre 0 et 1 correspondant à nos premiers noyaux de convolution.

Vous avez le choix pour créer ces vecteurs :

- Soit vous creez des tableaux à 1 dimension N=32x32, 6x28x28, 6x14x14 et 6x5x5 respectivement, donc chaque case correspond à un élément,
- Soit vous creez des tableaux à 2 ou 3 dimensions, mais dans ce cas vous devrez gérer des pointeurs de pointeurs de pointeurs.

Créez ces matrices et les fonctions d'initialisation adéquates.

Layer 2 - Convolution 2D

Réaliser la première convolution 2D :

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

Layer 3 - Sous-échantillonnage

Réalisez le sous-échantillonnage 2D : Le sous-échantillonnage se fera par moyennage de 2x2 pixels vers 1 pixel.

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

Tests

Fournissez à vos premières couches du CNN des valeurs "simple" et vérifiez les valeurs obtenues. Utilisez la fonction `MatrixPrint` pour affichez les différentes valeurs entre chaques couches de votre CNN.

Fonctions d'activation

Dans notre réseau de neurone, il manque dans fonctions d'activation, notamment en sortie de la première convolution.

Rajouter une fonction d'activation de type tanh.

Cette fonction peut être appelé par chaque kernel de votre GPU, le prototype sera donc :

```
__device__ float activation_tanh(float M)
```

Une fonction cette fonction créée, testez à nouveau vos premières couches.

Partie 3. Un peu de Python

Dans cette partie vous allez faire l'entrainement de votre réseau de neurone et comprendre les dernières couches associés à mettre en place dans votre programme CUDA.

Notebook jupyter

Exécutez le notebook LeNet5.ipynb.

Faites une synthèse des différents layers et listez ce qu'il vous manque.

Création des fonctions manquantes

Faites une synthèse des différents layers et listez ce qu'il vous manque.

Lesquelles sont facilement parallélisable ?

Creez toutes les fonctions manquantes.

Importation du dataset MNIST et affichage des données en console

Le dataset **MNIST** est disponible sous la forme d'un fichier `train-images.zip`.
Les images sont sauvegardées dans un fichier binaire à l'intérieur de ce fichier zip.

Un script en C/CUDA, nommé `printMNIST.cu`, est disponible pour l'ouverture de ce fichier et l'affichage d'une image.

Inspirez-vous de ce script pour charger les données à appliquer dans votre propre programme.

Export des poids dans un fichier

Afin de faire fonctionner notre réseau de neurones sur des données représentant des objets réels (et non plus des valeurs aléatoires), il est nécessaire d'importer les poids dans un programme écrit en C/CUDA.

1. Exportez les poids dans un format qui sera facile à importer ultérieurement dans votre programme.

2. Créez, dans votre programme CUDA, l'ensemble des matrices nécessaires.
3. Importez les poids dans vos différentes matrices.

Tests

Testez votre programme avec les données MNIST.

A ce stade, vous devriez avoir un programme capable de lire les données MNIST, de les traiter à travers les différentes couches de votre réseau de neurone, et de vous donner une prédiction.

Vous pouvez comparer vos résultats avec ceux obtenus dans le notebook jupyter.

Si vous avez des différences, essayez de les expliquer. Autrement, Félicitations ! Vous avez implémenté l'inférence d'un réseau de neurones convolutif en C/CUDA.

Pour aller plus loin

Dans ce TP, nous avons implémenté l'inférence d'un réseau de neurones convolutif en C/CUDA. Voici quelques pistes pour aller plus loin.

Entraînement du réseau de neurones

Bien que l'inférence soit la partie la plus importante pour la plupart des applications, l'entraînement du réseau de neurones est souvent plus complexe et nécessite plus de ressources. Pour entraîner un réseau de neurones, il faut calculer le gradient de l'erreur par rapport aux poids du réseau, et mettre à jour ces poids en fonction du gradient. Cela nécessite de calculer les dérivées des fonctions d'activation et de coût, et de les propager à travers le réseau.

En pratique, cela peut être fait en utilisant des bibliothèques comme PyTorch ou TensorFlow, qui fournissent des outils pour calculer automatiquement les gradients et mettre à jour les poids du réseau.

Si l'on souhaite implémenter l'entraînement du réseau de neurones en C/CUDA, deux approches sont possibles :

- Implémenter la rétropropagation du gradient et la descente de gradient stochastique à la main (plus simple mais spécifique au réseau de neurones)
- Implémenter un algorithme de différentiation automatique (autograd) comme celui de PyTorch (plus complexe mais plus général et réutilisable).

Pour la première approche, vous pouvez consulter l'implementation de la rétropropagation du gradient en C/CUDA d'un llm de type GPT-2 ici.

Pour la deuxième approche, vous pouvez consulter karpathy/micrograd qui est une implémentation simple de l'autograd en Python, le port en C/CUDA est disponible ici.

Approche hybride

Il est aussi possible d'utiliser une approche hybride CPU/GPU pour accélérer l'entraînement du réseau de neurones. A la place de recoder l'ensemble du programme en C/CUDA, on peut utiliser des librairies comme PyTorch ou TensorFlow en Python pour définir le réseau de neurones et identifier la partie qui prend le plus de

temps à s'exécuter. On peut ensuite coder cette partie en C/CUDA pour accélérer l'exécution et l'appeler depuis Python.

Le cas de l'Attention

Le mécanisme d'attention, également appelé couche d'attention, est au cœur des Transformers, une architecture utilisée dans les réseaux de neurones modernes pour le traitement du langage naturel (LLM) tels que GPT-2, BERT, etc. Bien que très efficace, le mécanisme d'attention est très coûteux en termes de calculs, car sa complexité est quadratique par rapport à la taille de la séquence d'entrée.

Pour accélérer l'attention, plusieurs approches ont été proposées, dont Flash Attention, qui optimise le calcul en réduisant l'utilisation de mémoire et en accélérant les opérations. Contrairement à l'attention classique qui stocke toutes les activations intermédiaires en mémoire, Flash Attention traite les entrées par blocs, en utilisant des techniques comme le tuilage et la fusion de noyaux pour maximiser l'utilisation des mémoires rapides des GPU (comme la SRAM). Cette méthode permet de calculer l'attention de manière efficace, même pour de longues séquences, en évitant les goulots d'étranglement liés à la mémoire et en réduisant le coût quadratique.

Cet algorithme optimisé peut être implémenté sous forme de kernels en C/CUDA pour accélérer l'entraînement du réseau de neurones, comme illustré dans ce projet. Ces kernels peuvent ensuite être appelés depuis Python pour améliorer les performances de cette partie du réseau de neurones.

Cet algorithme optimisé peut être ensuite implémenté en kernels C/CUDA pour accélérer l'entraînement du réseau de neurones comme fait dans ce projet. Ces kernels peuvent être ensuite appelés depuis Python pour accélérer cette partie du réseau de neurones.