

# Programmation sur Processeur Graphique – GPGPU

## TP : réseau de neurones

Centrale Nantes

P.-E. Hladik, pehladik@ec-nantes.fr

—

Version bêta (11 janvier 2023)

## 1 Travail à réaliser

La partie pratique va vous permettre d'appliquer les concepts de parallélisme des données et de programmation en CUDA à un code plus important que ceux des travaux dirigés. L'objectif final est de démontrer une maîtrise de la programmation sur GPU et de la conception pour des données parallèles.

Pour cela, vous allez programmer de zéro (un code séquentiel vous est fourni comme exemple) un réseau de neurones artificiels et chercher à optimiser sa vitesse de calcul sur GPU. Pour réaliser cela, vous devrez :

- déterminez quelle partie de l'application prend du temps,
- déterminez une ou plusieurs approches pour prendre en considération le parallélisme,
- réalisez plusieurs implémentations de l'approche,
- mesurez les performances d'exécution des implémentations pour divers paramètres,
- reliez les résultats aux concepts du cours.

A la fin des séances de TP vous devrez rendre votre code ainsi qu'un rapport comprenant les éléments suivants :

- une explication de la structure du code séquentiel,
- une explication et justification pour chacune des optimisations que vous avez réalisées,
- une évaluation des performances de vos optimisations.

La notation prendra en considération les éléments suivants :

- Démonstration des performances obtenues
  - Produit des résultats corrects
  - Réalise une bonne accélération par rapport au code de base
  - Compare plusieurs optimisations
- Rapport
  - Couvre bien toutes les sections requises
  - Est techniquement solide
  - Est clair, organisé et bien écrit
  - Propose une analyse approfondie du problème, de l'approche et des résultats
- Qualité du code
  - Le style de codage est soigné
  - Le code est bien documenté

Bien que le rapport ne représente qu'une partie de votre note, il est essentiel qu'il soit bien fait pour valider la partie réalisation. Par exemple, si vous avez appliqué une optimisation intéressante, il est essentiel que vous l'expliquiez bien dans votre rapport.

## 2 Cas d'étude : réseau de neurones artificiels

Un réseau de neurones artificiel est constitué de différentes couches (layers) de neurones. On note  $L$  le nombre de couches avec 0 pour désigner la couche d'entrée.

Les neurones d'une couche sont reliés à la totalité des neurones des couches adjacentes. Ces liaisons sont soumises à un poids altérant l'effet de l'information sur le neurone de destination.

Les sorties de chaque neurone d'une couche sont envoyées à la couche suivante, appelée couche cachée, où elles sont combinées et transformées par des fonctions d'activation non linéaires. Cette couche cachée peut avoir un nombre quelconque de neurones et peut être composée de plusieurs sous-couches si nécessaire.

Un réseau de neurones peut être représenté par une liste de matrices contenant les poids des liaisons entre les couches. Nous noterons  $w^{(l)}$  la matrice qui représente les poids entre la couche  $l \geq 1$  et  $l-1$  (la couche d'entrée est noté 0). En notant  $n^{(l)}$  le nombre de neurones dans la couche  $l$ , la matrice  $w^{(l)}$  est de taille  $n^{(l)} \times n^{(l-1)}$ .

Un biais est ajouté pour chaque couche permettant d'activer plus ou moins facilement un neurone. Les biais seront explicitement modélisés (au lieu de supposer que nous avons un neurone supplémentaire par couche avec une activation qui est toujours 1). Nous avons donc pour chaque couche  $l \geq 1$  un vecteur colonne  $b^{(l)}$  de taille  $n^{(l)}$ .

**Algorithme de propagation :** En notant  $a^{(l)}$  le vecteur représentant l'activation des neurones de la couche  $l$  (avec  $a^{(0)}$  la couche d'entrée) et  $z^{(l)}$  les entrées pondérées de la couche  $l$ , on a

$$z^{(l)} = w^{(l)} \times a^{(l-1)} + b^{(l)}$$

et

$$a^{(l)} = f\left(z^{(l)}\right)$$

avec  $f$  la fonction d'activation.

**Algorithme de rétro-propagation du gradient :** En notant  $y$  le vecteur colonne avec les valeurs attendues sur la couche de sortie, la fonction d'erreur peut simplement se calculer par

$$e^{(L)} = a^{(L)} - y$$

et le gradient avec

$$\delta^{(L)} = (a^{(L)} - y) \circ f'(z^{(L)})$$

On peut ainsi calculer pour chaque couche le gradient de la fonction de coût

$$\delta^{(l-1)} = (w^{(l)})^T \times \delta^{(l)} \circ f'(z^{(l-1)})$$

avec  $\circ$  le produit matriciel de Hadamard.

Et ainsi mettre à jour les poids avec

$$w^{(l)} \leftarrow w^{(l)} - \alpha \cdot \delta^{(l)} \times (a^{(l-1)})^T$$

et les biais

$$b^{(l)} \leftarrow b^{(l)} - \alpha \cdot \delta^{(l)}$$

avec  $\alpha$  la vitesse d'apprentissage.

**Initialisation des poids** Les biais du réseau peuvent être initialisés à 0. Les poids dans  $w^{(l)}$  ne doivent cependant pas l'être et sont initialisés en utilisant une gaussienne centrée sur 0 et avec un écart type de  $\frac{1}{\sqrt{n^{(l-1)}}}$ .

**Minibatch** La descente de gradient par minibatch est une variante de l'algorithme de descente de gradient qui divise l'ensemble de données d'apprentissage en petits lots qui sont utilisés pour calculer l'erreur du modèle et mettre à jour les coefficients du modèle.

Notons  $m$  la taille d'un minibatch. Les vecteurs  $a^{(l)}$  et  $z^{(l)}$  deviennent alors des matrices de taille  $n^{(l)} \times m$  où les colonnes représentent les valeurs pour une entrée.

Les équations de propagation et rétro-propagation deviennent :

$$\begin{aligned} z^{(l)} &= w^{(l)} \times a^{(l-1)} + b^{(l)} \times \mathbf{1} \\ a^{(l)} &= f(z^{(l)}) \\ \delta^{(L)} &= (a^{(L)} - y) \circ f'(z^{(L)}) \\ \delta^{(l-1)} &= (w^{(l)})^T \times \delta^{(l)} \circ f'(z^{(l-1)}) \\ w^{(l)} &\leftarrow w^{(l)} - \frac{\alpha}{m} \delta^{(l)} \times (a^{(l-1)})^T \\ b^{(l)} &\leftarrow b^{(l)} - \frac{\alpha}{m} \delta^{(l)} \times \mathbf{1}^T \end{aligned}$$

avec  $\mathbf{1}$  un vecteur ligne de taille  $1 \times m$  et  $y$  une matrice  $n^{(L)} \times m$  avec les sorties attendues pour chaque entrée.

**Epoch** L'entraînement du réseau est effectué sur des epoch. Au début de chaque epoch, les données d'entrée sont mélangées de manière aléatoire et divisées en minibatches.

La façon la plus simple de procéder est de générer un vecteur de nombres de 0 à  $n$ , avec  $n$  le nombre de données d'entrée, représentant les indices des données dans les vecteurs d'entrée, de mélanger aléatoirement ce vecteur d'indices (par exemple en effectuant  $n$  permutations aléatoires) et de prendre les  $m$  éléments dans l'ordre donné par ce vecteur.

En utilisant ceci, nous pouvons calculer la matrice d'entrée  $a^{(0)}$  pour le calcul de propagation et la matrice de résultat attendu  $y$  utiliser pour la rétropropagation.

À la fin d'une epoch, nous devons tester la précision du réseau formé. Pour chaque exemple des données de test, calculez la réponse du réseau en appelant le calcul de propagation puis en prenant l'indice du neurone avec la plus grande activation dans la couche de sortie. En comparant cette réponse à l'étiquette attendue donnée par les données, on peut compter le nombre d'erreurs ou de réussites.

### 3 Dataset MNIST

Pour valider le programme, vous utiliserez un exemple classique de reconnaissance de digits dont les dataset sont disponibles sur <http://yann.lecun.com/exdb/mnist/> et sur hippocampus.

Le dataset comprend 60 000 données étiquetées. Une donnée est une image de  $28 \times 28$  pixels et les étiquettes sont un digit (0 à 9) associé à la donnée. Un ensemble de 10 000 données de validation sont aussi fournies dans le même format.

Les données ne sont pas normalisées, et il faut les ramener leur valeur entre 0,0 (noir) et 1,0 (blanc), plutôt qu'entre 0 et 255.

## 4 Code fourni

Afin de vous aider vous trouverez sur hippocampus les jeux de données étiquetées ainsi qu'un code C séquentiel réalisant l'ensemble des traitements pour l'exemple MNIST. Vous pouvez partir de ce code pour optimiser les calculs et commencer à écrire les parties à porter sur le GPU.

L'algorithme est testé avec le dataset de MNIST en considérant que :

- la couche d'entrée est de taille 784 ( $28 \times 28$ )
- il n'y a qu'une seule couche cachée de 30 neurones
- la couche de sorties comprend 10 neurones (un par digit entre 0 et 9) ainsi la valeur 2 sera codée par  $y = (0, 0, 1, 0, 0, 0, 0, 0, 0, 0)^T$
- la fonction d'activation est la fonction sigmoïde :  $f(x) = \frac{1}{1+e^{-x}}$  et  $f'(x) = f(x)(1 - f(x))$
- la vitesse d'apprentissage  $\alpha$  est de 0.05