AMAL - TP 11
Programmation GPU (CUDA)

Nicolas Baskiotis - Benjamin Piwowarski - Laure Soulier

2020-2021

1 But

Beaucoup d'opérateurs existent aujourd'hui dans PyTorch (ou autres APIs). Toutefois, il peut être utile de savoir étendre PyTorch pour des fonctions spécifiques qui peuvent être implémentées de manière plus intelligente (= moins de mémoire et/ou moins de temps) si on les programme directement en C++, en utilisant le parallélisme du CPU ou, encore mieux, du GPU.

Fonction à implémenter La fonction que nous cherchons à utiliser est celle de la distance entre deux ensemble de vecteurs. Étant donné N_1 vecteurs $\mathbf{x}_i$ et N_2 vecteurs $\mathbf{y}_j$, calculer l'ensemble des distances

$$d_{ij}^{\epsilon} = d_p^{\epsilon}(\mathbf{x}_i, \mathbf{y}_j) = \left(\sum_k |x_{ik} - y_{jk} + \epsilon|^p \right)^{1/p}$$

où $\mathbf{x} \in \mathbb{R}^{N_1 \times d}$, $\mathbf{y} \in \mathbb{R}^{N_2 \times d}$, $d_{ij}^{\epsilon} \in \mathbb{R}$, et où $p \in \mathbb{R}^{+*}$ et ϵ sont des paramètres (ϵ permet d'éviter dans une certaine mesure les valeurs nulles, et est utilisé pour le calcul des distances dans PyTorch).

2 Calculs préliminaires (sur papier)

Question 1

Vous calculerez les dérivées partielles $\frac{\partial L}{\partial x_{ik}}$ et $\frac{\partial L}{\partial y_{jk}}$ en fonction de la dérivée partielle $\Delta_{ij} = \frac{\partial L}{\partial d_{ij}^{\epsilon}}$ de l'erreur par rapport aux sorties.

3 Squelette de code

Vous téléchargerez une archive qui contient une grande partie du code nécessaire pour tester vos implémentations. La version naïve de l'implémentation en `pytorch` vous est donnée dans le fichier `distances/naive.py`. Ce répertoire contient deux squelettes qui serviront à l'implémentation des deux versions améliorées que vous coderez : l'une en `torchscript` qui compilera à l'aide de `jit` votre code en `C++` - `cpp.py` ; l'autre qui donnera une implémentation `GPU` (en utilisant *numba*) - `cuda.py`.

Vous trouverez un script `checks.py` à la racine qui vous permettra de tester votre code :

- `python3 checks.py check METHOD DIRECTION` pour tester si vos résultats correspondent à la version python de votre implémentation où `METHOD` est `cpp` ou `cuda` et `DIRECTION` est `forward` ou `backward`
- `python3 checks.py benchmark METHOD` où `METHOD` est `py`, `cpp`, ou `cuda` pour tester la vitesse de votre implémentation, vous utiliserez
- `--help` pour connaitre tous les paramètres que vous pouvez modifier. Par exemple, `python3 checks.py -1 500 -2 500 benchmark cuda` permet de tester avec $N_1 = N_2 = 500$.
- `python3 curves.py` pour calculer les temps de calcul CPU/GPU et tracer les courbes correspondantes

Le code Python dans `naive.py` utilise la fonction de distance pairwise qui permet de calculer la distance entre des paires de vecteurs - il faut donc faire une boucle pour répéter l'opération N_1 (ou N_2) fois.

Dans tous les cas, si vous modifiez du code, il sera recompilé à la demande. Le backward de la version naïve est calculé automatiquement par PyTorch en utilisant le chaînage des opérateurs. Cette fonction sert de référence pour calculer la fonction ou son gradient (implémentations CPU et GPU).

4 Implémentation CPU

Dans cette section, il s'agit de faire une implémentation CPU en utilisant TorchScript qui permet de "compiler" votre code Python en un programme qui sera exécuté directement par le code C++ de PyTorch ; ceci a pour conséquence que seule une sous-partie de Python peut être utilisée (cf. la documentation). Par ailleurs, `TorchScript` permet de s'affranchir du calcul de graphe dynamique de PyTorch en compilant statiquement les fonctions. Vous pouvez donc faire des fonctions `forward` et `backward` optimisées.

Pour l'implémentation CPU, vous utiliserez un forward similaire au code Python fourni. Vous ferez attention à optimiser vos fonctions : par exemple, il vaut mieux se passer d'opérateurs tel que `stack`) et d'affecter le résultats dans un tenseur déjà créé.

Vous implémenterez votre fonction backward en utilisant des méthodes de haut niveau (vectorielles) ; ceci vous permettra de vérifier vos formules avant le passage au GPU et d'avoir un code relativement rapide en CPU.

Question 2

Implémentez le forward et backward dans le fichier `cpp.py`.

5 GPU

Le point le plus important de ce TP est l'implémentation GPU. Afin de vous faciliter la tâche, vous utiliserez numba CUDA qui permet de générer du code pour GPU à partir de code Python. Il y a beaucoup plus de restrictions sur le sous-ensemble de Python que vous pouvez utiliser, aussi bien au niveau du langage (ex. pas de compréhension de liste) ou des librairies Python. Le prix à payer est un coût constant important lors de l'appel des fonctions (par rapport à un code C++/CUDA).

Avant de vous lancer dans la programmation GPU, il faut en comprendre les mécanisme de bases. Ceux-ci sont décrits avec détail dans la documentation CUDA disponible sur le site de Nvidia. Nous résumons les points importants ci-dessous.

Très brièvement, dans un GPU CUDA, un programme multi-thread est décomposé en blocs de threads. Chaque bloc est exécuté de manière indépendante (et dans n'importe quel ordre). Un ensemble de threads d'un bloc ont la garantie d'être exécutés en même temps : plus précisément, chaque thread exécute *exactement* le même code au même moment mais avec un registre local (au niveau du processeur GPU), ce qui permet d'effectuer des calculs en parallèle.

Dans chaque bloc, les threads peuvent coopérer. Dans ce TP, nous n'utiliserons pas ce mécanisme de coopération (ce qui ne pose pas de problème vu la nature du calcul à implémenter), mais nous découperons tout de même le problème en faisant en sorte que le nombre de threads par groupe corresponde au nombre de threads supporté par l'architecture ; pour connaître ce nombre, vous pouvez utiliser `nvidia-smi` et repérer le nom de la carte, avant de vous référer au tableau. Dans ce TP, vous pouvez régler dynamiquement le nombre de threads en utilisant l'options –threads afin d'observer la différence (il n'y en aura pas beaucoup). La plupart des GPUs sont limités à 1024 threads.

Dans le cadre de ce TP, nous utiliserons une structure de blocs et de threads en 2D qui permet de faciliter le calcul sur des matrices car les blocs et threads correspondent directement à une sortie de la matrice (dans notre cas), et plus généralement à une zone à calculer. La figure 5 illustre la structure de blocs et threads en 2D (il y a l'équivalent en 1D et 3D).

CUDA étend le C++ en permettant de définir des noyaux (*kernels*) qui sont des fonctions C++ spéciales et exécutées en parallèle (sur GPU). Numba (Cuda) reproduit ce mécanisme en Python : un noyau CUDA est appelé avec la syntaxe suivante depuis le code

`cudakernel[[numBlocks, threadsPerBlock]]`(paramètres)

où `cudakernel` est une fonction décorée par texttt@cuda.jit() et compilée à la volée

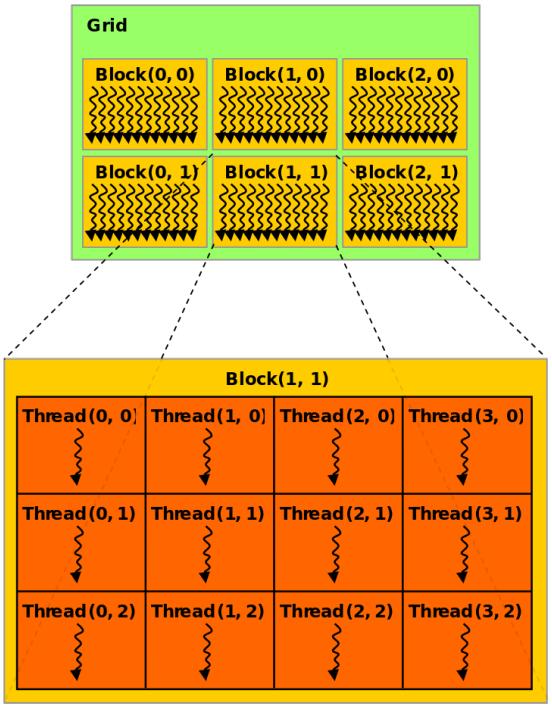

FIGURE 1 – (Wikipedia) Structure bloc / thread 2D. Le nombre de blocs est 3×2. Chaque bloc correspond à 4×3 threads.

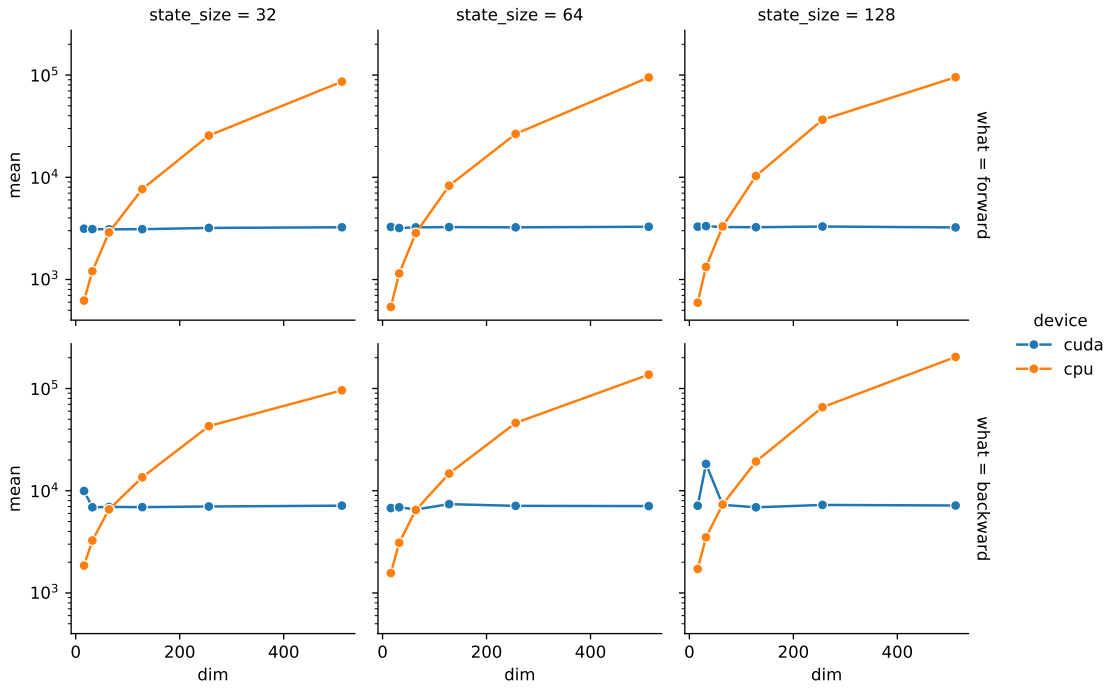

FIGURE 2 – Performance GPU vs CPU (µs) - implémentations TorchScript pour le CPU et Numba (CUDA) pour le GPU

par Numba. Dans cette fonction, en plus des arguments, vous avez accès aux informations suivantes sur la thread/bloc en cours d'exécutions :

Index du bloc `numba.cuda.blockIdx.x` et `numba.cuda.blockIdx.y`

Nombre de blocs `numba.cuda.blockDim.x` et `numba.cuda.blockDim.y`

Index du thread `numba.cuda.threadIdx.x` et `numba.cuda.threadIdx.y`

Question 3

Il vous reste à compléter la fonction noyau `cuda.py` où chaque thread calcule un d_{ij}^{ϵ}. Pour le backward, il vous faudra compléter l'appel du noyau et le noyau lui même.

6 Résultats attendus

Les résultats peuvent varier selon votre implémentation et le matériel, mais vous devriez obtenir un type de résultat illustré figure 6.

Aller plus loin

Pour optimiser encore plus le code, il est possible de coder directement en C++. Voici une liste de pointeurs

Documentation PyTorch (extensions C++).

C++ API avec en particulier une introduction à l'utilisation des tenseurs.