AMAL 2019-20 - TP 11 - Programmation GPU

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).

Calculs préliminaires (sur papier)

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

Code

Vous téléchargerez une archive qui contient une grande partie du code nécessaire pour tester vos implémentations. En particulier, en vous plaçant à la racine du répertoire, il faudra que vous modifiiez trois fichiers

- `code/distances_python.py` pour le code Python
- `code/distances/cpp/distances.cpp` pour le code C++/CPU
- `code/distances/cuda/distances_cuda_kernel.cu` pour le code C++/GPU

Pour tester si vos résultats correspondent à la version python de votre implémentation, vous utiliserez `python3 checks.py check METHOD` où `METHOD` est `cpp`, ou `cuda`.

Pour tester la vitesse de votre implémentation, vous utiliserez `python3 checks.py benchmark METHOD` où `METHOD` est `py`, `cpp`, ou `cuda`.

Utilisez `--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 $N_1 = N_2 = 500$.

Dans tous les cas, si vous modifiez du code, il sera recompilé à la demande.

Implémentation Python

Vous implémenterez de façon naïve le code, en utilisant la fonction torch.nn.functional.pairwise_distance 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.

Le backward sera calculé automatiquement par PyTorch en utilisant le chaînage des opérateurs.

Implémentation CPU

Pour l'implémentation CPU, vous utiliserez un forward similaire au Python; 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.

La plupart des opérations sur les tenseurs disponibles en Python le sont aussi en C++. Par exemple, `torch.nn.functional.pairwise_distance` correspond à `at::pairwise_distance`. Les opérateurs `view` et `expand` existent également : avec un tenseur `x`, `x.view({-1, 1})` en C++ correspond à `x.view((-1, 1))` en python.

La documentation de l'API C++ est disponible ici, avec en particulier une introduction à l'utilisation des tenseurs et la liste des opérations `at`.

GPU

La documentation CUDA est disponible sur le site de Nvidia.

Très brièvement, dans un GPU CUDA, un programme multi-thread est décomposé en blocs de threads. Chaque bloc est executé 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. Dans chaque bloc, les threads peuvent coopérer. Dans ce TP, nous n'utiliserons pas ce mécanisme, mais nous découperons tout

de même le problème en faisant en sorte que le nombre de thread par groupe corresponde au nombre de thread supporté par l'architecture; pour connaitre ce nombre, vous pouvez utiliser `nvidia-smi` et repérer le nom de la carte, avant de vous référer au tableau.

CUDA C++ étend le C++ en permettant de définir des noyaux (*kernels*) qui sont des fonctions C++ spéciales et qui seront exécutées en parallèle. Un noyau CUDA est appelé avec la syntaxe suivante depuis le code

```
cudakernel<<<numBlocks, threadsPerBlock>>>
```

Vous trouverez dans `code/distances/cuda/distances_cuda_kernel.cu` une fonction `distances_cuda_forward` qui est appelée depuis Python (à quelques détails près) et qui permet d'appeler un noyau CUDA `distances_cuda_forward_kernel` pour différent types de réels (`float32`, `float64`) avec des tenseurs facilement accessibles (voir l'introduction à l'utilisation des tenseurs). Il vous reste à compléter la fonction noyau `distances_cuda_forward_kernel` où chaque thread calcule un d_{ij}^{ϵ}.

Pour le backward, il vous faudra compléter l'appel du noyau et le noyau lui même.

Informations utiles

Documentation PyTorch sur les extensions

Documentation PyTorch

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

Résultats attendus

Les résultats peuvent varier selon votre implémentation et le matériel, mais vous devriez avoir une vitesse similaire pour le code python et C++ (sauf si vous utilisez du parallélisme, e.g. OpenMP). Avec $N_1 = 100, N_2 = 101$ et $d = 256$, le code GPU est environ 50 fois plus rapide (forward 170µs/backward 520µs avec l'implémentation CUDA, et 56000µs/23000µs pour Python) et utilise nettement moins de mémoire.