Sorbonne Université

Rapport

PLDAC : Enhancing SPLADE with Large Vocabularies in Neural Information Retrieval

Étudiants :
Yuan LIU
Herve NGUYEN

Encadré par :
Benjamin PIWOWARSKI

14 mai 2024

Résumé

SPLADE est un modèle neuronal pour la recherche d'information, qui se sert d'une représentation sparse pour avoir plusieurs avantage par rapport aux modèles denses tel que :

- La possibilité d'utiliser un dictionnaire inversé de manière efficace.
- Avoir une correspondance lexicale explicite entre la requête et les résultats.
- Une meilleur interprétabilité.

Notre objectif ici est d'optimiser la partie backpropagation en tirant partie de la représentation sparse des données d'entraînement.

Le project consiste d'abord à determiner des expressions du backward de l'apprentissage de SPLADE et utiliser ces expressions pour optimiser la backpropagation avec des exécutions en parallèle sur GPU.

Repository GitHub : `https://github.com/Mosakana/PLDAC_Splade_GPU_Calculate`

Table des matières

1 Définitions

1.1 Tenseurs et dimensions

Pour aborder le sujet, nous devons d'abord définir les tenseurs de base que nous allons manipuler.

- Le tenseur X qui contient les données d'entraînement, $X \in \mathbb{R}^{B \times L \times D}$. Avec B la taille du batch, L la longueur maximum d'une séquence et D la dimension d'un emdedding d'un mot.
- Le tenseur W de poids, $W \in \mathbb{R}^{D \times V}$. Avec D la dimension d'un embedding et V la taille du vocabulaire.
- Le vecteur b du biais, $b \in \mathbb{R}^{V}$. Avec V la taille du vocabulaire.
- Le masque M, $M \in \mathbb{R}^{B \times L}$ dont les éléments prennent soit la valeur $-\inf$ ou 0. Avec B la taille du batch, V la taille du vocabulaire.
- $XW + b + M$ est donc de dimension $B \times L \times D$.

1.2 Forward

Le forward de SPLADE prends l'expression suivante :

$$f(X, W, b) = ReLU(max(XW + b + M)) \tag{1}$$

Avec max ici le max sur la dimension L, sachant que $XW + b + M$ est de dimension $B \times L \times V$. Alors $max(XW + b + M)$ et sa $ReLU$ est donc de dimension $B \times V$.

2 Expressions des gradients

Afin de pouvoir essayer d'améliorer le backward de SPLADE avec du code parallélisé, nous avons besoin de l'expression des gradients de W, X, b par rapport à la loss.

Soit $\nabla \in B \times V$ le gradient de la loss par rapport à $f(X, W, b)$, ∇_{st} est un élément de ∇ à la position respective.

2.1 Gradient de W

On a d'abord, avec la règle de la chaîne :

$$\begin{aligned}
\frac{\partial L}{\partial W_{dv}} &= \sum\nolimits_{b \in \{1,\ldots,B\},\, v' \in \{1,\ldots,V\}} \frac{\partial L}{\partial f_{b'v'}} \frac{\partial f_{b'v'}}{\partial W_{dv}} && (2)\\
&= \sum\nolimits_{b' \in \{1,\ldots,B\},\, v' \in \{1,\ldots,V\}} \nabla_{b'v'} \frac{\partial f_{bv'}}{\partial W_{dv}} && (3)
\end{aligned}$$

Le gradient de W_{dv} par rapport à f_{st} prend la forme :

$$\begin{aligned}
\frac{\partial f_{b'v'}}{\partial W_{dv}} &= \frac{\partial ReLU(max(XW + b + M))_{b'v'}}{\partial W_{dv}} && (4)\\
&= \frac{\partial ReLU(max(X_{b'\bullet\bullet} W_{\bullet v'} + b_{v'} + M_{b'}))}{\partial W_{dv}} && (5)
\end{aligned}$$

Soit $l^* \in \mathbb{N}^{B\times V}$ la matrice des indices sur la dimension L de $XW+b+M$, où $l^*_{b,v}$ est donc le l sur L où on obtient la valeur du max dans $max(XW+b+M)$ pour un couple (b,v) donné.

On remarque donc que $\frac{\partial f_{b'v'}}{\partial W_{dv}}$ peut prendre différentes valeurs possible selon le cas.

$$\frac{\partial f_{b'v'}}{\partial W_{dv}} = \begin{cases} 0 & \text{si } v' \neq v \\ 0 & \text{si } l \neq l^*_{b'v}, \forall l \in [0, L[\\ \text{sinon} & (6) \end{cases} \tag{6}$$

Si nous ne sommes pas les deux premiers cas, nous avons donc (le masque ici étant à 0) :

$$\frac{\partial f_{b'v'}}{\partial W_{dv}} = \frac{\partial (X_{b'l^*_{b'v}\bullet} W_{\bullet v} + b_w)}{\partial W_{dv}} \tag{7}$$

$$= X_{b'l^*_{b'v}d} \tag{8}$$

Par conséquent selon (3), (6), (8) :

$$\frac{\partial L}{\partial W_{dv}} = \sum\nolimits_{b'\in\{1,\dots,B\}} \nabla_{b'v} X_{b'l^*_{b'v}d} \tag{9}$$

Et pour une colonne de W, l'expression est :

$$\frac{\partial L}{\partial W_{\bullet v}} = \sum\nolimits_{b'\in\{1,\dots,B\}} \nabla_{b'v} X_{b'l^*_{b'v}\bullet} \tag{10}$$

2.2 Gradient de X

Comme avec W, avec la règle de la chaîne :

$$\frac{\partial L}{\partial X_{bld}} = \sum\nolimits_{b'\in\{1,\dots,B\},\, v'\in\{1,\dots,V\}} \frac{\partial L}{\partial f_{b'v'}} \frac{\partial f_{b'v'}}{\partial X_{bld}} \tag{11}$$

$$= \sum\nolimits_{b'\in\{1,\dots,B\},\, v'\in\{1,\dots,V\}} \nabla_{b'v'} \frac{\partial f_{b'v'}}{\partial X_{bld}} \tag{12}$$

Le gradient de X_{bld} par rapport à f_{st} prend la forme :

$$\frac{\partial f_{b'v'}}{\partial X_{bld}} = \frac{\partial ReLU(max(XW+b+M))_{b'v'}}{\partial X_{bld}} \tag{13}$$

$$= \frac{\partial ReLU(max(X_{b'\bullet\bullet} W_{\bullet v'} + b_{v'} + M_{b'}))}{\partial X_{bld}} \tag{14}$$

Donc :

$$\frac{\partial f_{b'v'}}{\partial X_{bld}} = \begin{cases} 0 & \text{si } b' \neq b \\ 0 & \text{si } l \neq l^*_{bv'} \\ \text{sinon} & (15) \end{cases} \tag{15}$$

Si $b = b$ et $l = l^*_{bv'}$, nous avons doc :

$$\begin{aligned}\frac{\partial f_{b'v'}}{\partial X_{bld}} &= \frac{X_{bl^*\bullet}W_{\bullet v'} + b_{v'}}{X_{bld}} && (16)\\ &= W_{dv'} && (17)\end{aligned}$$

Par conséquent, selon (12), (16), (17) :

$$\frac{\partial L}{\partial X_{bld}} = \sum\nolimits_{v' \in \{1,\ldots,V\} \backslash l = l^*_{bv'}} \nabla_{bv'} W_{dv'} \tag{18}$$

Pour notre projet on utilisera donc :

$$\frac{\partial L}{\partial X_{bl\bullet}} = \sum\nolimits_{v' \in \{1,\ldots,V\} \backslash l = l^*_{bv'}} \nabla_{bv'} W_{\bullet v'} \tag{19}$$

2.3 Gradient de b

Avec la règle de la chaîne :

$$\begin{aligned}\frac{\partial L}{\partial b_v} &= \sum\nolimits_{b' \in \{1,\ldots,B\},\, v' \in \{1,\ldots,V\}} \frac{\partial L}{\partial f_{b'v'}} \frac{\partial f_{b'v'}}{\partial b_v} && (20)\\ &= \sum\nolimits_{b' \in \{1,\ldots,B\},\, v' \in \{1,\ldots,V\}} \nabla_{b'v'} \frac{\partial f_{b'v'}}{\partial b_v} && (21)\end{aligned}$$

Le gradient de b_v par rapport à f_st s'exprime de la manière suivante :

$$\begin{aligned}\frac{\partial f_{b'v'}}{\partial b_v} &= \frac{\partial ReLU(max(XW + b + M))_{b'v'}}{\partial b_v} && (22)\\ &= \frac{\partial ReLU(max(X_{b'..}W_{.v'} + b_{v'} + M_{b'}))}{\partial b_v} && (23)\\ &= \begin{cases} 0 & \text{si } v \neq v' \\ 0 & \text{si } M_{b'} = -\infty \\ 1 & \text{sinon} \end{cases} && (24)\end{aligned}$$

D'après (21) et (24) :

$$\begin{aligned}\frac{\partial L}{\partial b_v} &= \sum\nolimits_{b' \in \{1,\ldots,B\} \backslash M_{b'} = 0} \nabla_{b'v} && (25)\\ \frac{\partial L}{\partial b\bullet} &= \sum\nolimits_{b' \in \{1,\ldots,B\} \backslash M_{b'} = 0} \nabla_{b'\bullet} && (26)\end{aligned}$$

3 Concept du calcul GPU

3.1 L'optimisation sur le calcul du gradient de W

Selon la formule (10) du gradient de W :

$$\frac{\partial L}{\partial W_{\bullet v}} = \sum_{b'} \nabla_{b'v} X_{b' l^*_{b'v} \bullet}$$

Il est possible de calculer cette formule en parallèle par rapport à la dimension D. Chaque processus s'occupe un d et parcourt la liste qui contient des triplets sous la forme (b, v, l^*_{bv}) en accumulant la valeur de $\nabla_{bv} X_{bl^*_{bv}d}$ à la position (d, v) de la matrice du gradient de W.

Il existe une autre approche. La tâche de calcul peut être divisé par la dimension V. Cependant, pour cette approche, la liste de triplet doit être transformée à une liste 2D qui regroupe les triplets de la même valeur de v dans une sous-liste. Chaque processus itère une sous-liste de triplets de v correspondant. À chaque itération, les valeurs de $\nabla_{bv} X_{bl^*_{bv}\bullet}$ par rapport au triplet pris (b, v, l^*_{bv}) sera sauvegardé à la postion de $(\bullet, v)$ de la matrice du gradient de W.

3.2 L'optimisation sur le calcul du gradient de X

La formule du gradient de X (cf. (19)) :

$$\frac{\partial L}{\partial X_{bl\bullet}} = \sum_{v' \backslash l = l^*_{b,v'}} \nabla_{bv'} W_{\bullet v'}$$

Ce calcul peut aussi être divisé par la dimension D. Les processus parallèles chacun travaille sur un d différent, accumule le produit de $\nabla_{bv} W_{dv}$ à l'adresse (b, l^*_{bv}, d) de la matrice du gradient de X par rapport au triplet (b, v, l^*_{bv}) récupéré.

Cependant, cette tâche ne peut pas être coupée par la dimension V. Si deux processus avec v différent, mais ils ont les même valeur de b et l^* par une coïncidence. Lorqu'ils exécutent les calculs, il est possible de se produit un conflit de I/O, celui-ci provoque possiblement des erreurs en resultat.

Pour le gradient de X, il est possible de découper le calcul par la dimension de B. Pourtant, généralement, les donées d'entraînement ont une taille de B qui est plus petite que celle de V ou D. Donc il n'est pas très efficace de diviser la tâche par rapport à la dimension B.

3.3 L'optimisation sur le calcul du gradient de b

Ici, il y a finalement peu d'intérêt d'implémenter le calcul sous Triton car *scatter_ add_* de Pytorch déjà optimisé existe pour cette opération.

4 Implémentation au niveau du code

4.1 Vérification des résultats

Premièrement, pour vérifier nos résultats. Nous avons écris un module test nommé check_grad_splade qui est exécutable pour tester nos expressions par rapport à ce que torch obtient. (avec torch.gradcheck)

4.2 Récupération des indices à élément non nuls

Dû à la nature sparse de X, de la ReLU et du masque, il est pertinent de vouloir seulement manipuler les éléments non-nuls parmis de nombreux éléments nuls.

Ainsi, au niveau du forward on récupère les indices du résultat à la sortie du forward des éléments non nuls.

```
relu = ReLU()
maximum, max_indice = torch.max(output, 1)
result = relu(maximum)

indice_not_zero = torch.nonzero(result, as_tuple=True)

effective_indice = []
for b, v in zip(*indice_not_zero):
    effective_indice.append((b, v, max_indice[b, v]))
```

Listing 1 – Récupération des indices effectives

4.3 Utilisation de la librairie triton

Pour implémenter les équations de calcul directement avec le GPU, nous avons utilisé la bibliothèque Triton de OpenAI qui permet d'écrire et compiler des kernel GPU en python sans passer par le code CUDA.

Le flag "@triton.jit" permet de déclarer à l'interpréteur python de considérer la fonction en dessous en tant que fonction kernel à éxécuter dans le GPU pour un processus donné.

À l'intérieur de la fonction on se retrouve donc à manipuler des pointeurs dans une plage de données, dans un point de vue très bas niveau.

Ci dessous le processus correspondant va itérer sur une rangée pour calculer une partie locale du gradient.

```python
@triton.jit
def gradient_w_kernel(index_ptr, grad_ptr, delta_ptr, x_ptr,
                      D, N_INDEX,
                      stride_index_dim0, stride_index_dim1,
                      stride_grad_d, stride_grad_v,
                      stride_delta_b, stride_delta_v,
                      stride_x_b, stride_x_l, stride_x_d,
                      BLOCK_SIZE_D: tl.constexpr):

    pid = tl.program_id(axis=0)
    start_point = pid * BLOCK_SIZE_D
    offsets = start_point + tl.arange(0, BLOCK_SIZE_D)
    mask = offsets < D

    for i in range(N_INDEX):
        b = tl.load(index_ptr + (i * stride_index_dim0) + 0 *
           stride_index_dim1)
        v = tl.load(index_ptr + (i * stride_index_dim0) + 1 *
           stride_index_dim1)
        lmax = tl.load(index_ptr + (i * stride_index_dim0) + 2 *
           stride_index_dim1)

        delta = tl.load(delta_ptr + (b * stride_delta_b + v *
           stride_delta_v))
        x = tl.load(x_ptr + (b * stride_x_b + lmax * stride_x_l +
           offsets * stride_x_d), mask=mask)
        grad = tl.load(grad_ptr + (offsets * stride_grad_d + v *
           stride_grad_v), mask=mask)

        grad += delta * x

        tl.store(grad_ptr + (offsets * stride_grad_d + v *
           stride_grad_v), grad, mask=mask)
```

Listing 2 – Fonction à compiler en tant que kernel pour calculer le gradient de W

Pour assigner utiliser des processus du GPU et calculer une partie du gradient on fait donc appel à une fonction wrapper qui appeler la fonction kernel correspondante. Dans l'exemple ci-dessous on lance donc les kernels GPU sur l'ensemble des processus dans la grille de thread (grid).

```python
def compute_gradient_w(index, delta, x, grad_weight):
    N_INDEX = index.shape[0]
    D = x.shape[2]

    BLOCK_SIZE_D = triton.next_power_of_2(D)
    num_warps = 4
    if BLOCK_SIZE_D >= 2048:
        num_warps = 8
    if BLOCK_SIZE_D >= 4096:
        num_warps = 16

    grid = lambda meta: (triton.cdiv(D, meta['BLOCK_SIZE_D']), )
    gradient_w_kernel[grid](index, grad_weight, delta, x,
                            D, N_INDEX,
                            index.stride(0), index.stride(1),
                            grad_weight.stride(0), grad_weight.stride(1),
                            delta.stride(0), delta.stride(1),
                            x.stride(0), x.stride(1), x.stride(2),
                            num_warps=num_warps,
                            BLOCK_SIZE_D=BLOCK_SIZE_D)

    return grad_weight.clone().detach().requires_grad_(True)
```

Listing 3 – Fonction wrapper qui permet d'appeler les fonction kernels

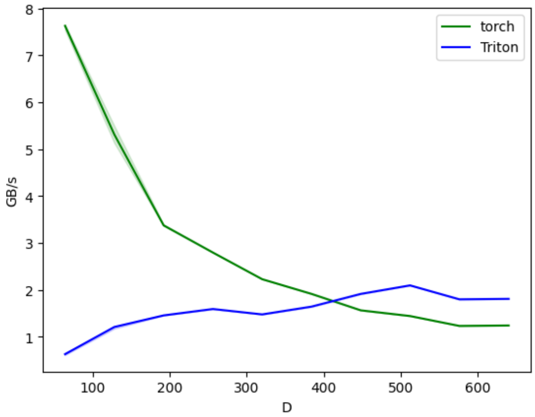
4.4 Évaluation des performances

Pour les performances nous avons tenté de comparer la version avec la bibliothèque torch et notre améliorée version sur Triton.

Pour l'optimisation de GPU calcul, 2 versions est fait pour une comparaison d'efficace parmis les choix de la dimension coupée. La permière version est de la coupe en dimension D sur W et la coupe en dimension D sur X. La deuxième version est de la coupe en dimension V sur W et la coupe en dimension D sur X.

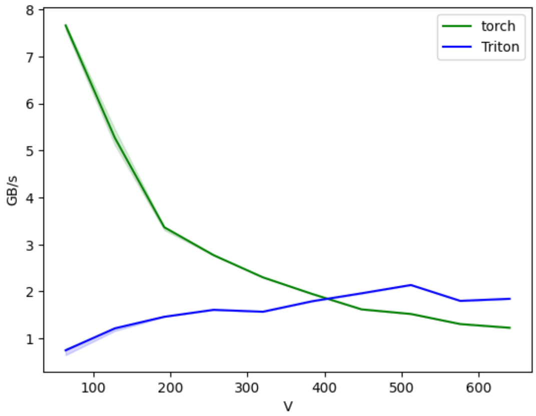
Les tenseurs sont initialisés de manière aléatoire avec torch.randn.

La métrique mesurée ici est le débit de calcul en Gigaoctet par seconde.



D	B	L	V	Torch	Triton
64	64	64	64	7.628	0.630
128	128	128	128	5.312	1.210
192	192	192	192	3.376	1.456
256	256	256	256	2.796	1.592
320	320	320	320	2.228	1.477
384	384	384	384	1.916	1.643
448	448	448	448	1.564	1.915
512	512	512	512	1.443	2.097
576	576	576	576	1.232	1.799
640	640	640	640	1.241	1.809

(a) coupe en D sur W



D	B	L	V	Torch	Triton
64	64	64	64	7.659	0.746
128	128	128	128	5.262	1.214
192	192	192	192	3.365	1.460
256	256	256	256	2.770	1.607
320	320	320	320	2.298	1.567
384	384	384	384	1.944	1.790
448	448	448	448	1.617	1.959
512	512	512	512	1.519	2.135
576	576	576	576	1.305	1.798
640	640	640	640	1.226	1.842

(b) coupe en V sur W

FIGURE 1 – La comparaison de performance entre différente coupe en dimension

Nous avons alors mesuré de débit de traitement / de calcul entre les deux versions. Les performances de Triton sont inférieures jusqu'à 400 éléments dans un batch, mais surpassent celles de Torch au-delà.

Parmis les deux versions, les tendance des variations de performance sont similaires. Mais, la performance de la coupe en V sur W est légèrement plus haute que celle de la coupe en D sur W.

En raison de la limite de l'environement d'exécution (RTX 4070 12Go), nous ne pouvons pas allons plus loin que la taille de 640 de chaque dimension. Il existe aussi une possibilité que la différence miniscule de performance entre ces deux version élargisse en un environement d'une capacité de calcul plus élevée. (eg. Nvidia A100)

5 Perspectives

Jusqu'à maintenant, nous n'avons que tenté d'optimiser le backward de SPLADE. Cependant, il y a aussi de l'intérêt à prendre avantage de la nature parsimonieuses des donnée pour également optimiser le forward.

D'ailleurs, les tests effectués ici sont des tests isolés à petite échelle. Il aurait été aussi intérêssant de voir si nos gains de performance se répliquent lorsque l'on entraine SPLADE sur le vrai dataset MSMARCO, ce qui n'est pas possible avec les équipements que nous avons à disposition.