

# Pilote Nvidia, LLaMA et ChatGPT

LLaMA (Large Language Model Meta AI) est une famille de modèles de langage de grande taille (LLMs), publiée par Meta AI à partir de février 2023.

J'ai récemment assemblé mon ordinateur avec une carte graphique Nvidia. Vous pouvez consulter ici comment assembler un ordinateur : <https://lzwjava.github.io/computer>.

Après cela, j'ai commencé à exécuter le projet LLaMA. L'URL GitHub du projet LLaMA est <https://github.com/facebookresearch/llama>.

## Installer le pilote Nvidia

Lorsque vous exécutez la commande,

```
torchrun --nproc_per_node 1 example_text_completion.py \
    --ckpt_dir llama-2-7b/ \
    --tokenizer_path tokenizer.model \
    --max_seq_len 128 --max_batch_size 4
```

Cela affiche l'erreur, "RuntimeError: Distributed package doesn't have NCCL built in". Apprenons-en plus sur NCCL.

La bibliothèque de communication collective NVIDIA (NCCL) implémente des primitives de communication multi-GPU et multi-nœuds optimisées pour les GPU NVIDIA et le réseau. Je me réfère aux sites web ci-dessous pour installer les pilotes NVIDIA.

- Téléchargements du CUDA Toolkit 12.2 Update 1, <https://developer.nvidia.com/cuda-downloads>
- NVIDIA NCCL, <https://developer.nvidia.com/nccl>
- Documentation NVIDIA Deep Learning NCCL, <https://docs.nvidia.com/deeplearning/nccl/install-guide/index.html>
- Guide d'installation de NVIDIA CUDA pour Linux, <https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html>
- Après l'installation d'Ubuntu, vous rencontrez la gestion MOK, <https://www.cnblogs.com/yutian-blogs/p/13019226.html>
- Ubuntu 22.04 pour le Deep Learning, <https://gist.github.com/amir-saniyan/b3d8e06145a8569c0d0e030a>
- Notes sur Ubuntu 22.04, <https://github.com/kmcminn/thinkpad/tree/main/extreme3g>

Lorsque nous installons avec succès le pilote NVIDIA pour notre carte graphique, puis que nous utilisons la commande `nvidia-smi` pour afficher ses détails, elle peut afficher les informations suivantes.

```
(base) lzw@lzw-MS-7E01:~$ nvidia-smi
```

```
Jeu 17 Août 04:15:43 2023
```

```
+-----+
| NVIDIA-SMI 535.86.10              Version du pilote : 535.86.10    Version CUDA : 12.2 |
+-----+-----+-----+
| GPU   Nom                        Persistence-M | Bus-Id          Disp.A | ECC Volatile Non Corrigé |
| Ventilateur  Temp   Perf          Pwr:Usage/Cap |                 Utilisation de la mémoire | Utilisation GPU
|                                     |                                     | Mode MIG |
+=====+=====+=====+
|   0   NVIDIA GeForce RTX 4070          On | 00000000:01:00.0  On |                 N/A |
| 0%    34C    P8                     9W / 215W |    666MiB / 12282MiB |    15%    Par défaut |
|                                     |                                     | N/A |
+-----+-----+-----+
```

```
+-----+
| Processus :                               |
| GPU   GI   CI          PID   Type   Nom du processus                               Utilisation de la mémoire
|       ID   ID                                     Utilisation
+=====+
|   0   N/A  N/A       1926    G    /usr/lib/xorg/Xorg                               381MiB |
|   0   N/A  N/A       2065    G    /usr/bin/gnome-shell                             120MiB |
|   0   N/A  N/A       3482    G    gnome-control-center                             2MiB |
|   0   N/A  N/A       3803    G    ...irefox/2987/usr/lib/firefox/firefox          149MiB |
+-----+
```

En réalité, il est difficile d'atteindre cette phase. Veuillez consulter attentivement le lien suivant, Ubuntu 22.04 Notes, <https://github.com/kmcminn/thinkpad/tree/main/extreme3g>.

## Apprendre LLaMA

Après avoir téléchargé les modèles et essayé d'exécuter la commande, nous rencontrerons l'erreur suivante :

torch.cuda.OutOfMemoryError: CUDA à court de mémoire. Tentative d'allocation de 86.00 MiB (GPU 0; capacité totale de 11.69 GiB; 9.70 GiB déjà alloués; 64.81 MiB libres; 9.70 GiB réservés au total par PyTorch). Si la mémoire réservée est » à la mémoire allouée, essayez de définir max\_split\_size\_mb pour éviter la fragmentation.

Comme la mémoire de notre carte graphique n'est que de 12 Go, et que la taille du modèle llama-2-7b est d'environ 13 Go, nous ne pouvons pas le faire fonctionner avec notre carte graphique.

Nous essayons d'utiliser l'autre projet, open-llama-3b, [https://huggingface.co/openlm-research/open\\_llama\\_3b](https://huggingface.co/openlm-research/open_llama_3b).

Nous rencontrons l'erreur suivante.

RuntimeError : Tous les tenseurs devraient être sur le même appareil, mais au moins deux appareils ont été trouvés, cuda:0 et cpu ! (lors de la vérification de l'argument pour l'argument index dans la méthode wrapper\_CUDA\_\_index\_select)

Et nous demandons à ChatGPT à ce sujet.

Le ChatGPT nous propose une solution très élégante. Nous devons ajouter le code suivant.

```
input_ids = input_ids.to(model.device)
```

Enfin, nous pouvons le lancer.

```
(llama) lzw@lzw-MS-7E01:~/Projects/open_llama_3b$ python run.py
```

```
Q: Quel est le plus grand animal ?
```

```
A: La baleine bleue.
```

```
Q: Quel est le plus grand animal ?
```

```
A: La baleine bleue. C'est le plus grand animal sur Terre. Elle est également
```

Nous avons également testé le projet llama2.c, <https://github.com/karpathy/llama2.c>.

```
(base) lzw@lzw-MS-7E01:~/Projects/llama2.c$ ./run stories15M.bin
```

```
Un jour, un grand lion et un petit lapin étaient dans les bois. Le grand lion dit : "Faisons la course .  
Le lion était rapide, mais il faisait attention à ne pas blesser le petit lapin. Le lapin se rapprochai  
Le petit lapin vit le lion dans l'eau et pensa : "Je devrais dire au lion de ne pas boire." Il dit : "L  
Le petit lapin essaya d'aider le lion, mais il ne put rien faire. Le lion avait bu trop d'eau et son ver  
tok/s atteint : 185.527329
```

## Comprendre le code avec ChatGPT

Zhiwei : Pourrais-tu expliquer les codes ci-dessous ?

```
def generate(
    self,
    prompt_tokens: List[List[int]],
    max_gen_len: int,
    temperature: float = 0.6,
    top_p: float = 0.9,
    logprobs: bool = False,
    echo: bool = False,
) -> Tuple[List[List[int]], Optional[List[List[float]]]]:
    params = self.model.params
    bsz = len(prompt_tokens)
    assert bsz <= params.max_batch_size, (bsz, params.max_batch_size)

    min_prompt_len = min(len(t) for t in prompt_tokens)
    max_prompt_len = max(len(t) for t in prompt_tokens)
    assert max_prompt_len <= params.max_seq_len
    total_len = min(params.max_seq_len, max_gen_len + max_prompt_len)

    pad_id = self.tokenizer.pad_id
    tokens = torch.full((bsz, total_len), pad_id, dtype=torch.long, device="cuda")
    for k, t in enumerate(prompt_tokens):
        tokens[k, : len(t)] = torch.tensor(t, dtype=torch.long, device="cuda")
    if logprobs:
        token_logprobs = torch.zeros_like(tokens, dtype=torch.float)

    prev_pos = 0
    eos_reached = torch.tensor([False] * bsz, device="cuda")
    input_text_mask = tokens != pad_id
    for cur_pos in range(min_prompt_len, total_len):
        logits = self.model.forward(tokens[:, prev_pos:cur_pos], prev_pos)
        if logprobs:
            token_logprobs[:, prev_pos + 1 : cur_pos + 1] = -F.cross_entropy(
                input=logits.transpose(1, 2),
```

```

        target=tokens[:, prev_pos + 1 : cur_pos + 1],
        reduction="none",
        ignore_index=pad_id,
    )

    if temperature > 0:
        probs = torch.softmax(logits[:, -1] / temperature, dim=-1)
        next_token = sample_top_p(probs, top_p)
    else:
        next_token = torch.argmax(logits[:, -1], dim=-1)

next_token = next_token.reshape(-1)
# ne remplacer le token que si le prompt a déjà été généré
next_token = torch.where(
    input_text_mask[:, cur_pos], tokens[:, cur_pos], next_token
)
tokens[:, cur_pos] = next_token
eos_reached |= (~input_text_mask[:, cur_pos]) & (
    next_token == self.tokenizer.eos_id
)
prev_pos = cur_pos
if all(eos_reached):
    break

    if logprobs:
        token_logprobs = token_logprobs.tolist()
        out_tokens, out_logprobs = [], []
        for i, toks in enumerate(tokens.tolist()):
            # couper à la longueur maximale de génération
            start = 0 if echo else len(prompt_tokens[i])
            toks = toks[start : len(prompt_tokens[i]) + max_gen_len]
            probs = None
            if logprobs:
                probs = token_logprobs[i][start : len(prompt_tokens[i]) + max_gen_len]
            # couper au token eos si présent
            if self.tokenizer.eos_id in toks:
                eos_idx = toks.index(self.tokenizer.eos_id)
                toks = toks[:eos_idx]

```

```

        probs = probs[:eos_idx] if logprobs else None
        out_tokens.append(toks)
        out_logprobs.append(probs)
    return (out_tokens, out_logprobs if logprobs else None)

```

ChatGPT : Bien sûr, voici une explication du code fourni :

```

def generate(
    self,
    prompt_tokens: List[List[int]],
    max_gen_len: int,
    temperature: float = 0.6,
    top_p: float = 0.9,
    logprobs: bool = False,
    echo: bool = False,
) -> Tuple[List[List[int]], Optional[List[List[float]]]]:

```

Cette fonction est nommée `generate` et accepte plusieurs paramètres :

- `prompt_tokens` est une liste contenant des séquences de tokens pour les invites.
- `max_gen_len` est la longueur maximale du texte généré.
- `temperature` est un paramètre utilisé pour contrôler le caractère aléatoire lors de la génération.
- `top_p` est un seuil de probabilité entre 0 et 1 utilisé pour contrôler la diversité lors de l'échantillonnage.
- `logprobs` est un booléen indiquant s'il faut enregistrer les log-probabilités du processus de génération.
- `echo` est un booléen indiquant s'il faut répéter l'entrée lors de la génération.

```

params = self.model.params
bsz = len(prompt_tokens)
assert bsz <= params.max_batch_size, (bsz, params.max_batch_size)

```

Ici, la fonction récupère les paramètres du modèle et calcule la taille du lot (batch size) des invites. Elle s'assure que la taille du lot ne dépasse pas la taille maximale spécifiée dans les paramètres du modèle.

```

min_prompt_len = min(len(t) for t in prompt_tokens)
max_prompt_len = max(len(t) for t in prompt_tokens)
assert max_prompt_len <= params.max_seq_len
total_len = min(params.max_seq_len, max_gen_len + max_prompt_len)

```

Ces lignes calculent les longueurs minimales et maximales des invites et s'assurent que la longueur maximale de l'invite ne dépasse pas la longueur maximale de séquence spécifiée dans les paramètres du modèle. La longueur totale est calculée en fonction de la longueur maximale générée et de la longueur maximale de l'invite.

```

pad_id = self.tokenizer.pad_id
tokens = torch.full((bsz, total_len), pad_id, dtype=torch.long, device="cuda")
for k, t in enumerate(prompt_tokens):
    tokens[k, : len(t)] = torch.tensor(t, dtype=torch.long, device="cuda")

```

Cette section initialise le tenseur `tokens` avec des tokens de remplissage et le remplit avec les séquences de tokens provenant des prompts.

```

if logprobs:
    token_logprobs = torch.zeros_like(tokens, dtype=torch.float)

```

Si `logprobs` est `True`, un tenseur `token_logprobs` est créé pour stocker les log-probabilités.

```

prev_pos = 0
eos_reached = torch.tensor([False] * bsz, device="cuda")
input_text_mask = tokens != pad_id

```

Les variables `prev_pos` (position précédente), `eos_reached` (indiquant si des tokens de fin de séquence ont été rencontrés), et `input_text_mask` (positions non remplies par du remplissage) sont initialisées.

La boucle suivante génère des tokens à partir de la longueur minimale du prompt jusqu'à la longueur totale.

```

for cur_pos in range(min_prompt_len, total_len):
    logits = self.model.forward(tokens[:, prev_pos:cur_pos], prev_pos)
    if logprobs:
        token_logprobs[:, prev_pos + 1 : cur_pos + 1] = -F.cross_entropy(

```

```

        input=logits.transpose(1, 2),
        target=tokens[:, prev_pos + 1 : cur_pos + 1],
        reduction="none",
        ignore_index=pad_id,
    )
    if temperature > 0:
        probs = torch.softmax(logits[:, -1] / temperature, dim=-1)
        next_token = sample_top_p(probs, top_p)
    else:
        next_token = torch.argmax(logits[:, -1], dim=-1)

    next_token = next_token.reshape(-1)
    next_token = torch.where(
        input_text_mask[:, cur_pos], tokens[:, cur_pos], next_token
    )
    tokens[:, cur_pos] = next_token
    eos_reached |= (~input_text_mask[:, cur_pos]) & (
        next_token == self.tokenizer.eos_id
    )
    prev_pos = cur_pos
    if all(eos_reached):
        break

```

Dans cette boucle, le modèle génère une distribution de probabilité `logits` pour le prochain token. Si `logprobs` est `True`, il calcule les log-probabilités en utilisant l'entropie croisée. En fonction de la température, il utilise `softmax` ou `argmax` pour sélectionner le prochain token. Il met ensuite à jour le tenseur `tokens` et vérifie la présence de tokens de fin de séquence.

```

if logprobs:
    token_logprobs = token_logprobs.tolist()

```

Si `logprobs` est `True`, le tenseur des log-probabilités est converti en une liste Python.

```

out_tokens, out_logprobs = [], []
for i, toks in enumerate(tokens.tolist()):
    ...

```



Dans cette boucle, pour chaque échantillon d'entrée, elle génère la séquence de tokens de sortie ainsi que les probabilités logarithmiques associées.

```
return (out_tokens, out_logprobs if logprobs else None)
```

Enfin, la fonction retourne les séquences de tokens générées ainsi que leurs probabilités logarithmiques correspondantes (si demandées).