

Nvidia-Treiber, LLaMA und ChatGPT

LLaMA (Large Language Model Meta AI) ist eine Familie von großen Sprachmodellen (LLMs), die ab Februar 2023 von Meta AI veröffentlicht wurde.

Ich habe kürzlich meinen Computer mit einer Nvidia-GPU zusammengebaut. Sie können hier nachlesen, wie man einen Computer baut: <https://lzwjava.github.io/computer>.

Danach begann ich, das LLaMA-Projekt auszuführen. Die GitHub-URL des LLaMA-Projekts ist <https://github.com/facebookresearch/llama>.

Nvidia-Treiber installieren

Wenn Sie den Befehl ausführen,

```
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
```

Hinweis: Der Code wurde nicht übersetzt, da es sich um eine technische Anweisung handelt, die in der Regel in der Originalsprache belassen wird.

Es zeigt den Fehler: “RuntimeError: Distributed package doesn’t have NCCL built in”. Lassen Sie uns etwas über NCCL lernen.

Die NVIDIA Collective Communication Library (NCCL) implementiert Multi-GPU- und Multi-Node-Kommunikationsprimitive, die für NVIDIA-GPUs und Netzwerke optimiert sind. Ich verweise auf die folgenden Websites, um NVIDIA-Treiber zu installieren.

- CUDA Toolkit 12.2 Update 1 Downloads, <https://developer.nvidia.com/cuda-downloads>
- NVIDIA NCCL, <https://developer.nvidia.com/nccl>
- NVIDIA Deep Learning NCCL Dokumentation, <https://docs.nvidia.com/deeplearning/nccl/install-guide/index.html>
- NVIDIA CUDA Installationshandbuch für Linux, <https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html>
- Nach der Installation von Ubuntu stoßen Sie auf “Perform MOK Management”, <https://www.cnblogs.com/yutian-blogs/p/13019226.html>
- Ubuntu 22.04 für Deep Learning, <https://gist.github.com/amir-saniyan/b3d8e06145a8569c0d0e030af6d>

- Ubuntu 22.04 Notizen, <https://github.com/kmcminn/thinkpad/tree/main/extreme3g>

Wenn wir den NVIDIA-Treiber für unsere Grafikkarte erfolgreich installiert haben und dann den Befehl `nvidia-smi` verwenden, um die Details anzuzeigen, kann er die folgenden Informationen anzeigen.

```
(base) lzw@lzw-MS-7E01:~$ nvidia-smi
Do Aug 17 04:15:43 2023
+-----+
| NVIDIA-SMI 535.86.10          Treiber-Version: 535.86.10    CUDA-Version: 12.2      |
|-----+-----+-----+
| GPU  Name                  Persistence-M | Bus-Id      Disp.A | Volatile Uncorr. ECC | | |
| Fan   Temp     Perf          Pwr:Usage/Cap |          Speicherverbrauch | GPU-Util Compute M. |
|           |                               |          |          |          MIG M. |
|-----+-----+-----+-----+-----+-----+
|  0  NVIDIA GeForce RTX 4070        On  | 00000000:01:00.0  On |          N/A | | |
|  0%   34C     P8              9W / 215W |  666MiB / 12282MiB |  15%    Default |
|           |                               |          |          |          N/A |
+-----+-----+-----+
+-----+
| Prozesse:
| GPU  GI  CI          PID  Typ  Prozessname          GPU-Speicher |
|           ID  ID
|-----+-----+-----+-----+-----+-----+-----+
|  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 |
+-----+
```

Tatsächlich ist es schwierig, diese Phase zu erreichen. Bitte beziehen Sie sich sorgfältig auf den Link hier, Ubuntu 22.04 Notes, <https://github.com/kmcminn/thinkpad/tree/main/extreme3g>.

Lerne LLaMA

Nachdem wir die Modelle heruntergeladen haben und versuchen, den Befehl auszuführen, werden wir auf den folgenden Fehler stoßen:

```
torch.cuda.OutOfMemoryError: CUDA-Speicher erschöpft. Es wurde versucht,  
86.00 MiB zuzuweisen (GPU 0; 11.69 GiB Gesamtkapazität; 9.70 GiB bereits  
zugewiesen; 64.81 MiB frei; 9.70 GiB insgesamt von PyTorch reserviert). Wenn der  
reservierte Speicher deutlich größer ist als der zugewiesene Speicher, versuchen  
Sie, max_split_size_mb zu setzen, um Fragmentierung zu vermeiden.
```

Da der Speicher unserer Grafikkarte nur 12 GB beträgt und die Größe des llama-2-7b-Modells etwa 13 GB umfasst, können wir es nicht mit unserer Grafikkarte ausführen.

Wir versuchen, das andere Projekt, open-llama-3b, zu verwenden, https://huggingface.co/openlm-research/open_llama_3b.

Wir stoßen auf den folgenden Fehler.

```
RuntimeError: Es wurde erwartet, dass alle Tensoren auf demselben Gerät  
liegen, aber es wurden mindestens zwei Geräte gefunden, cuda:0 und cpu!  
(beim Überprüfen des Arguments für das Argument index in der Methode wrap-  
per_CUDA_index_select)
```

Und wir fragen ChatGPT danach.

ChatGPT bietet uns eine sehr elegante Lösung. Wir müssen den folgenden Code hinzufügen:

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

Schließlich können wir es hochfahren.

```
(llama) lzw@lzw-MS-7E01:~/Projects/open_llama_3b$ python run.py  
F: Was ist das größte Tier?  
A: Der Blauwal.  
F: Was ist das größte Tier?  
A: Der Blauwal. Es ist das größte Tier auf der Erde. Es ist auch das
```

Wir haben auch das Projekt llama2.c ausprobiert, <https://github.com/karpathy/llama2.c>.

```
(base) lzw@lzw-MS-7E01:~/Projects/llama2.c$ ./run stories15M.bin
Eines Tages waren ein großer Löwe und ein kleiner Hase im Wald. Der große Löwe sagte: "Lass uns zum Flu...
Der Löwe war schnell, aber er war vorsichtig, den kleinen Hasen nicht zu verletzen. Der Hase kam dem Flu...
Der kleine Hase sah den Löwen im Wasser und dachte: "Ich sollte dem Löwen sagen, dass er nicht trinken ...
Der kleine Hase versuchte, dem Löwen zu helfen, aber er konnte es nicht. Der Löwe trank zu viel und sei...
erreichte tok/s: 185.527329
```

Code verstehen mit ChatGPT

Zhiwei: Könnten Sie den folgenden Code erklären?

```
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)
    # Token nur ersetzen, wenn der Prompt bereits generiert wurde
    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()):
            # auf maximale Generierungslänge kürzen
            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]
        # auf EOS-Token kürzen, falls vorhanden
        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: Sicher, hier ist eine Erklärung des bereitgestellten Codes:

```

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]]]]:

```

Diese Funktion heißt `generate` und akzeptiert mehrere Parameter:

- `prompt_tokens` ist eine Liste, die Sequenzen von Tokens für Eingabeaufforderungen enthält.
- `max_gen_len` ist die maximale Länge des generierten Textes.
- `temperature` ist ein Parameter, der zur Steuerung der Zufälligkeit während der Generierung verwendet wird.
- `top_p` ist ein Wahrscheinlichkeitsschwellenwert zwischen 0 und 1, der zur Steuerung der Vielfalt während des Samplings verwendet wird.
- `logprobs` ist ein boolescher Wert, der angibt, ob die Log-Wahrscheinlichkeiten des Generierungsprozesses aufgezeichnet werden sollen.
- `echo` ist ein boolescher Wert, der angibt, ob die Eingabe während der Generierung wiedergegeben werden soll.

```
params = self.model.params
```

```

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

```

(Der Code bleibt unverändert, da es sich um eine technische Implementierung handelt, die nicht übersetzt werden muss.)

Hier erhält die Funktion die Modellparameter und berechnet die Batch-Größe der Eingabeauforderungen. Sie stellt sicher, dass die Batch-Größe nicht größer ist als die im Modell angegebene maximale Batch-Größe.

```

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)

```

Diese Zeilen berechnen die minimalen und maximalen Längen der Prompts und stellen sicher, dass die maximale Prompt-Länge die im Modell angegebene maximale Sequenzlänge nicht überschreitet. Die Gesamtlänge wird basierend auf der maximalen generierten Länge und der maximalen Prompt-Länge berechnet.

```

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")

```

Hinweis: Der Code wurde nicht übersetzt, da es sich um eine Programmiersprache handelt, die in der Regel nicht übersetzt wird.

Dieser Abschnitt initialisiert den `tokens`-Tensor mit Padding-Tokens und füllt ihn mit den Token-Sequenzen aus den Prompts.

```

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

```

Wenn `logprobs` True ist, wird ein Tensor `token_logprobs` erstellt, um die Log-Wahrscheinlichkeiten zu speichern.

```

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

```

Hinweis: Der Code wurde nicht übersetzt, da es sich um eine Programmiersprache handelt, die in der Regel nicht übersetzt wird.

Die Variablen `prev_pos` (vorherige Position), `eos_reached` (ob End-of-Sequence-Token erkannt wurden) und `input_text_mask` (Positionen, die nicht mit Padding gefüllt sind) werden initialisiert.

Die folgende Schleife erzeugt Token, beginnend bei der minimalen Eingabelänge bis zur Gesamtlänge.

```
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
```

Innerhalb dieser Schleife generiert das Modell eine Wahrscheinlichkeitsverteilung `logits` für das nächste Token. Wenn `logprobs` auf True gesetzt ist, werden die logarithmierten Wahrscheinlichkeiten mithilfe der Kreuzentropie berechnet. Abhängig von der `temperature` wird entweder

Softmax oder Argmax verwendet, um das nächste Token auszuwählen. Anschließend wird der `tokens`-Tensor aktualisiert und auf End-of-Sequence-Tokens überprüft.

```
if logprobs:  
    token_logprobs = token_logprobs.tolist()
```

Hinweis: Der Code bleibt unverändert, da es sich um eine Programmiersprache handelt und keine Übersetzung erforderlich ist.

Wenn `logprobs` True ist, wird der Log-Wahrscheinlichkeiten-Tensor in eine Python-Liste umgewandelt.

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

In dieser Schleife wird für jede Eingabeprobe die Ausgabe-Token-Sequenz und die zugehörigen Log-Wahrscheinlichkeiten generiert.

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

Schließlich gibt die Funktion die generierten Token-Sequenzen und ihre entsprechenden Log-Wahrscheinlichkeiten zurück (falls angefordert).