

Nvidia ドライバー、LLaMA、そして ChatGPT

LLaMA (Large Language Model Meta AI) は、2023年2月から Meta AI によってリリースされた大規模言語モデル (LLM) のファミリーです。

最近、Nvidia GPU を搭載したコンピュータを組み立てました。詳細はこちらをご覧ください、コンピュータの組み立て方、<https://lzwjava.github.io/computer>。

その後、LLaMA プロジェクトを実行し始めました。LLaMA プロジェクトの GitHub URL は <https://github.com/facebookresearch/llama> です。

Nvidia ドライバのインストール

コマンドを実行すると、

```
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
```

このコードは、`torchrun` を使用して1つのプロセスで `example_text_completion.py` スクリプトを実行するコマンドです。以下のオプションが指定されています：

- `--nproc_per_node 1`: 1つのノードで1つのプロセスを実行します。
- `--ckpt_dir llama-2-7b/`: チェックポイントディレクトリとして `llama-2-7b/` を指定します。
- `--tokenizer_path tokenizer.model`: トークナイザーモデルのパスとして `tokenizer.model` を指定します。
- `--max_seq_len 128`: 最大シーケンス長を 128 に設定します。
- `--max_batch_size 4`: 最大バッチサイズを 4 に設定します。

エラーが表示されます：「RuntimeError: Distributed package doesn't have NCCL built in」。NCCL について学びましょう。

NVIDIA Collective Communication Library (NCCL) は、NVIDIA GPU とネットワーク向けに最適化されたマルチ GPU およびマルチノード通信プリミティブを実装しています。以下のウェブサイトを参照して、NVIDIA ドライバーをインストールしました。

- CUDA Toolkit 12.2 Update 1 ダウンロード, <https://developer.nvidia.com/cuda-downloads>

- NVIDIA NCCL, <https://developer.nvidia.com/nccl>
- NVIDIA Deep Learning NCCL ドキュメント, <https://docs.nvidia.com/deeplearning/nccl/install-guide/index.html>
- NVIDIA CUDA Linux インストールガイド, <https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html>
- Ubuntu インストール後に Perform MOK Management が表示される場合, <https://www.cnblogs.com/yutian-blogs/p/13019226.html>
- ディープラーニング用 Ubuntu 22.04, <https://gist.github.com/amir-saniyan/b3d8e06145a8569c0d0e030af6d60>
- Ubuntu 22.04 メモ, <https://github.com/kmcminn/thinkpad/tree/main/extreme3g>

NVIDIA ドライバーをグラフィックカードに正常にインストールした後、`nvidia-smi` コマンドを使用してその詳細を表示すると、以下の情報が表示されます。

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

```
2023年8月17日 04:15:43
```

```

+-----+
| NVIDIA-SMI 535.86.10           ドライバーバージョン: 535.86.10   CUDAバージョン: 12.2       |
+-----+-----+-----+-----+
| GPU 名前                      持続性-M | バス-ID          ディスプレイ.A | 揮発性未修正ECC | |
| ファン  温度   パフォーマンス     電力:使用量/上限 |      メモリ使用量 | GPU使用率  計算モード |
|                               |                               |                               |      MIGモード |
+=====+=====+=====+=====+
|   0  NVIDIA GeForce RTX 4070      オン | 00000000:01:00.0 オン |                               | N/A |
| 0%   34C    P8                   9W / 215W |   666MiB / 12282MiB |   15%   デフォルト |
|                               |                               |                               |      N/A |
+-----+-----+-----+-----+

```

```

+-----+
| プロセス:                                                                |
| GPU  GI  CI      PID   タイプ   プロセス名                                     GPUメモリ |
|      ID  ID                                     使用量   |
+=====+=====+=====+=====+
|   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 |
+-----+

```

実際、この段階に到達するのは難しいです。こちらのリンク、Ubuntu 22.04 Notes、<https://github.com/kmcminn/thinkpad/tree/main/extreme3g> を注意深く参照してください。

LLaMA を学ぶ

モデルをダウンロードし、コマンドを実行しようとする、以下のエラーが発生します。

```
torch.cuda.OutOfMemoryError: CUDA のメモリ不足です。86.00 MiB の割り当てを試みま
した (GPU 0; 合計容量 11.69 GiB; 9.70 GiB が既に割り当て済み; 64.81 MiB が空き; PyTorch
によって合計 9.70 GiB が予約されています)。予約メモリが割り当てメモリを大幅に
上回る場合、フラグメンテーションを防ぐために max_split_size_mb を設定してみ
てください。
```

私たちのグラフィックカードのメモリは 12GB しかなく、llama-2-7b モデルのサイズは約 13GB であるため、このグラフィックカードでは実行できません。

私たちは、別のプロジェクトである open-llama-3b、https://huggingface.co/openlm-research/open_llama_3b を使用してみます。

以下のエラーが発生しました。

```
RuntimeError: すべてのテンソルが同じデバイス上にあることが期待されました
が、少なくとも2つのデバイス、cuda:0 と cpu が見つかりました！ (メソッド wrap
per_CUDA__index_select の引数 index をチェックしている際に発生)
```

そして、これについて ChatGPT に尋ねます。

ChatGPT は非常に美しい修正を提供してくれました。以下のコードを追加する必要があります。

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

最後に、それを実行することができます。

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

Q: 最も大きな動物は何ですか？

A: シロナガスクジラです。

Q: 最も大きな動物は何ですか？

A: シロナガスクジラです。地球上で最も大きな動物です。また、

また、プロジェクト「llama2.c」も試してみました。<https://github.com/karpathy/llama2.c>


```

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)
        # プロンプトがすでに生成されている場合のみトークンを置換
        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()):
    # 最大生成長に切り詰める
    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]
        # もし eos トークンがあればそこで切り詰める
    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: もちろん、以下に提供されたコードの説明をします:

```

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]]]]:

```

この関数は、指定されたプロンプトトークンに基づいてテキストを生成するためのものです。以下に各パラメータの説明を示します:

- `prompt_tokens`: 生成の基となるプロンプトのトークンリスト。リストのリストとして渡されます。
- `max_gen_len`: 生成するテキストの最大長。
- `temperature`: 生成時のサンプリングの温度。値が高いほどランダム性が高くなります。
- `top_p`: トップ p サンプリングのパラメータ。確率の累積がこの値に達するまでのトークンのみを考慮します。
- `logprobs`: 生成されたトークンの対数確率を返すかどうか。
- `echo`: プロンプトを出力に含めるかどうか。

この関数は、生成されたトークンのリストと、必要に応じて対数確率のリストを返します。

この関数は `generate` という名前です、いくつかのパラメータを受け取ります:

- `prompt_tokens` は、プロンプトのトークンシーケンスを含むリストです。
- `max_gen_len` は、生成されるテキストの最大長です。
- `temperature` は、生成中のランダム性を制御するためのパラメータです。
- `top_p` は、サンプリング中の多様性を制御するために使用される 0 から 1 の間の確率閾値です。
- `logprobs` は、生成プロセスの対数確率を記録するかどうかを示すブール値です。
- `echo` は、生成中に入力をエコーするかどうかを示すブール値です。

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

このコードスニペットでは、`params` にモデルのパラメータを代入し、`bsz` にプロンプトトークンの数を代入しています。その後、`assert` 文を使用して、バッチサイズ (`bsz`) がモデルの最大バッチサイズ (`params.max_batch_size`) を超えていないことを確認しています。もし超えていた場合、エラーが発生します。

ここで、関数はモデルのパラメータを取得し、プロンプトのバッチサイズを計算します。この関数は、バッチサイズがモデルパラメータで指定された最大バッチサイズを超えないようにします。

```
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)
```

このコードスニペットでは、以下の処理を行っています：

1. `min_prompt_len` に `prompt_tokens` 内の最小のトークン長を代入します。
2. `max_prompt_len` に `prompt_tokens` 内の最大のトークン長を代入します。
3. `max_prompt_len` が `params.max_seq_len` 以下であることを確認します (アサーション)。
4. `total_len` に `params.max_seq_len` と `max_gen_len + max_prompt_len` のうち小さい方を代入します。

このコードは、プロンプトのトークン長を確認し、生成するシーケンスの長さを制限するために使用されます。

これらの行は、プロンプトの最小および最大長を計算し、プロンプトの最大長がモデルパラメータで指定された最大シーケンス長を超えないようにします。総長は、生成される最大長とプロンプトの最大長に基づいて計算されます。

```

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")

```

このコードは、PyTorch を使用して、与えられたプロンプトトークンをパディングされたテンソルに変換する処理を行っています。以下に各ステップの説明を日本語で示します。

1. `pad_id = self.tokenizer.pad_id:`
 - `pad_id` は、トークナイザーが定義するパディングトークンの ID を取得します。パディングトークンは、シーケンスの長さを揃えるために使用されます。
2. `tokens = torch.full((bsz, total_len), pad_id, dtype=torch.long, device="cuda"):`
 - `tokens` は、バッチサイズ `bsz` とシーケンスの総長 `total_len` を持つテンソルを作成します。このテンソルは、すべての要素が `pad_id` で初期化され、データ型は `torch.long`、デバイスは `cuda` (GPU) に設定されます。
3. `for k, t in enumerate(prompt_tokens)::`
 - `prompt_tokens` は、各プロンプトのトークンリストを含むリストです。このループでは、各プロンプトトークンリスト `t` とそのインデックス `k` を取得します。
4. `tokens[k, : len(t)] = torch.tensor(t, dtype=torch.long, device="cuda"):`
 - 各プロンプトトークンリスト `t` を `torch.tensor` に変換し、`tokens` テンソルの対応する行にコピーします。このとき、`t` の長さに応じて、`tokens` テンソルの該当部分が更新されます。

このコードは、バッチ内の各プロンプトトークンをパディングされたテンソルに変換し、GPU 上で処理できるようにするためのものです。

このセクションでは、`tokens` テンソルをパディングトークンで初期化し、プロンプトからのトークンシーケンスで埋めます。

```

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

```

`logprobs` が `True` の場合、対数確率を格納するためのテンソル `token_logprobs` が作成されます。

```

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

```

このコードスニペットでは、以下の変数が定義されています：

- `prev_pos`: 前の位置を保持するための変数で、初期値は0です。
- `eos_reached`: バッチサイズ (`bsz`) に基づいて、各シーケンスが終端記号 (EOS) に到達したかどうかを示すブール値のテンソルです。初期状態ではすべての要素が `False` に設定されています。
- `input_text_mask`: トークンがパディング ID (`pad_id`) でないかどうかを示すマスクです。このマスクは、入力テキストの有効な部分を識別するために使用されます。

変数 `prev_pos` (前の位置)、`eos_reached` (シーケンス終端トークンが検出されたかどうか)、および `input_text_mask` (パディングが埋められていない位置) が初期化されます。

次のループは、最小のプロンプト長から始めて、トータルの長さに達するまでトークンを生成します。

```
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)
```

このコードは、指定された範囲内でトークンを生成するループです。`logits` はモデルの出力で、`logprobs` が有効な場合、トークンの対数確率を計算します。`temperature` が0より大きい場合、確率分布をソフトマックス関数で計算し、`sample_top_p` 関数を使って次のトークンをサンプリングします。`temperature` が0以下の場合、最も確率の高いトークンを選択します。

```
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

```

このループ内で、モデルは次のトークンに対する確率分布 `logits` を生成します。`logprobs` が `True` の場合、クロスエントロピーを使用して対数確率を計算します。`temperature` に応じて、次のトークンを選択するためにソフトマックスまたはアグマックスを使用します。その後、`tokens` テンソルを更新し、シーケンス終了トークンをチェックします。

```

if logprobs:
    token_logprobs = token_logprobs.tolist()

```

このコードは、`logprobs` が真 (`True`) の場合に、`token_logprobs` をリスト形式に変換しています。`tolist()` メソッドは、NumPy 配列や PyTorch テンソルなどを Python のネイティブなリストに変換するために使用されます。

`logprobs` が `True` の場合、対数確率テンソルは Python のリストに変換されます。

```

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

```

このループでは、各入力サンプルに対して、出力トークンシーケンスとそれに関連する対数確率を生成します。

```

return (out_tokens, out_logprobs if logprobs else None)

```

このコードは、`out_tokens` と、`logprobs` が `True` の場合に `out_logprobs` を返し、そうでない場合は `None` を返すことを示しています。コード自体は変更せず、そのままにしておきます。

最後に、この関数は生成されたトークンシーケンスとそれに対応する対数確率（要求された場合）を返します。