

# Nvidia 驱动, LLaMA 和 ChatGPT

ChatGPT 帮助翻译。

---

LLaMA (Large Language Model Meta AI) 是由 Meta AI 从 2023 年 2 月开始发布的一系列大型语言模型 (LLMs)。

我最近使用了 Nvidia GPU 组装了自己的电脑。你可以在这里查看，文章标题是《如何组装一台电脑》，链接为<https://lzwjava.github.io/computer>。

在那之后，我开始运行 LLaMA 项目。LLaMa 项目的 GitHub 链接是 <https://github.com/facebookresearch/llama>。

## 安装 Nvidia 驱动

当你运行以下命令时，

```
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
```

它会显示错误信息：“RuntimeError: Distributed package doesn’t have NCCL built in”。让我们了解一下 NCCL。

NVIDIA 集体通信库 (NCCL) 实现了针对 NVIDIA GPU 和网络进行优化的多 GPU 和多节点通信基元。

我参考了以下网站来安装 NVIDIA 驱动程序。

- CUDA Toolkit 12.2 Update 1 下载链接，<https://developer.nvidia.com/cuda-downloads>
- NVIDIA NCCL，<https://developer.nvidia.com/nccl>
- NVIDIA 深度学习 NCCL 文档，<https://docs.nvidia.com/deeplearning/nccl/install-guide/index.html>
- NVIDIA CUDA Linux 安装指南，<https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html>
- 在安装 Ubuntu 后，遇到 MOK 管理问题，<https://www.cnblogs.com/yutian-blogs/p/13019226.html>
- 针对深度学习的 Ubuntu 22.04 配置，<https://gist.github.com/amir-saniyan/b3d8e06145a8569c0d0e030af6d60bea>
- Ubuntu 22.04 笔记，<https://github.com/kmcminn/thinkpad/tree/main/extreme3g>

当我们成功为显卡安装了 NVIDIA 驱动程序后，可以使用 `nvidia-smi` 命令查看其详细信息，显示如下：

```
(base) lzw@lzw-MS-7E01:~$ nvidia-smi
Thu Aug 17 04:15:43 2023
+-----+
| NVIDIA-SMI 535.86.10      驱动版本: 535.86.10    CUDA 版本: 12.2      |
|-----+-----+
| GPU 名称          持续运行时长 | 总线 ID      显示运行 | 易失性 ECC      | | |
| 风扇 温度 性能      功率: 使用/上限 |           内存使用 | GPU 利用率 计算模式 |
|           |           |           |           |           MIG 模式 |
+=====+=====+=====+=====+=====+=====+
| 0 NVIDIA GeForce RTX 4070     运行中 | 00000000:01:00.0 运行中 |           N/A | | |
| 0% 34C P8             9W / 215W | 666MiB / 12282MiB | 15%   默认模式 |
|           |           |           |           |           N/A |
+-----+-----+-----+-----+
+-----+
| 进程:
| GPU GI CI      PID  类型  进程名称          GPU 内存 |
|       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 |
+-----+
```

实际上，达到这个阶段是有一定难度的。请仔细参考这里的链接，Ubuntu 22.04 笔记，<https://github.com/kmcminn/thinkpad/tree/main/extreme3g>。

## 运行 LLaMA

在下载了模型后，尝试运行命令时，我们会遇到以下错误：

```
torch.cuda.OutOfMemoryError: CUDA 内存不足。尝试分配 86.00 MiB (GPU 0; 总容量
11.69 GiB; 已分配 9.70 GiB; 空闲 64.81 MiB; PyTorch 已总共保留了 9.70 GiB) 如果保
留的内存 > 已分配的内存，请尝试将 max_split_size_mb 设置得更大，以避免碎片化。
```

由于我们的显卡内存只有 12 GB，而 llama-2-7b 模型的大小约为 13GB，所以我们无法在我们的显卡上运行它。

我们尝试使用另一个项目，open-llama-3b，[https://huggingface.co/openlm-research/open\\_llama\\_3b](https://huggingface.co/openlm-research/open_llama_3b)。

我们遇到了以下错误：

RuntimeError: 期望所有张量都在相同的设备上，但发现至少两个设备，cuda:0 和 cpu! (在  
检查方法 wrapper\_CUDA\_index\_select 的参数索引时)

然后我们向 ChatGPT 请教了这个问题。

ChatGPT 给出了一个非常好的修复方法。我们需要添加以下代码：

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

最终，我们可以运行它。

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

Q: 什么是最大的动物?

A: 蓝鲸。

Q: 什么是最大的动物?

A: 蓝鲸。它是地球上最大的动物。它还是

我们还尝试了 llama2.c 项目，<https://github.com/karpathy/llama2.c>。

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

一天，一只大狮子和一只小兔子在树林里。大狮子说：“我们比赛跑到河边喝点水。”小兔子同意了，他们开始比赛。

狮子跑得很快，但他小心不伤害小兔子。兔子靠近河边，但大狮子没有看见他。当狮子穿过河时，他滑倒掉进了河里。

小兔子看到狮子掉进了水里，想：“我应该告诉狮子不要喝水。”他说：“狮子，不要从河里喝水。不安全。”但狮子没有听，喝

小兔子试图帮助狮子，但他做不到。狮子喝得太多，肚子疼了。大狮子很伤心，说：“我告诉过你不要从河里喝水。”小兔子哭着

达到的 tok/s 速度：185.527329

## 通过 ChatGPT 理解代码

Zhiwei: 解释代码

```
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)
    # only replace token if prompt has already been generated

```

```

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()):
    # cut to max gen len
    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]
    # cut to eos tok if any
    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: 以下是提供的代码的解释：

```

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]]]]:

```

这个函数名为 `generate`, 接受了几个参数:

- `prompt_tokens` 是一个包含提示标记序列的列表。
- `max_gen_len` 是生成的最大文本长度。
- `temperature` 是用于在生成过程中控制随机性的参数。
- `top_p` 是一个介于 0 到 1 之间的概率阈值, 用于控制采样的多样性。
- `logprobs` 是一个布尔值, 指示是否记录生成过程的对数概率。
- `echo` 是一个布尔值, 指示在生成过程中是否回显输入。

```

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

```

在这里, 函数获取了模型参数, 并计算了提示的批次大小。它确保批次大小不大于模型参数中指定的最大批次大小。

```

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)

```

这些行计算了提示的最小和最大长度, 并确保最大提示长度不超过模型参数中指定的最大序列长度。总长度基于最大生成长度和最大提示长度计算而来。

```

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")

```

这部分初始化了填充标记的 `tokens` 张量, 并使用提示中的标记序列填充它。

```

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

```

如果 `logprobs` 为真, 创建了一个与 `tokens` 张量形状相同的张量 `token_logprobs`, 用于存储对数概率。

```
prev_pos = 0
```

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

变量 `prev_pos`（前一个位置）、`eos_reached`（是否遇到了序列结束标记）和 `input_text_mask`（未填充标记的位置）被初始化。

下面的循环从最小提示长度开始逐步生成标记，直到总长度。

```
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
```

在这个循环内，模型生成下一个标记的概率分布 `logits`。如果需要，通过交叉熵计算对数概率。根据温度参数 `temperature`，使用 `softmax` 函数或 `argmax` 策略选择下一个标记。然后，更新 `tokens` 张量，并检查是否遇到了序列结束标记。

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

如果 `logprobs` 为真，则将对数概率张量转换为 Python 列表。

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

在这个循环内，为每个输入样本生成输出的标记序列和相关的对数概率。

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

最后，函数返回生成的标记序列和它们对应的对数概率（如果需要）。