

Nvidia Driver, LLaMA and ChatGPT

LLaMA (Large Language Model Meta AI) is a family of large language models (LLMs), released by Meta AI starting in February 2023.

I recently built my computer with Nvidia GPU. You may check out here, How to Build a Computer, <https://lzwjava.github.io/computer>.

After that, I started to run up the LLaMA project. LLaMa project's GitHub URL is <https://github.com/facebookresearch/llama>.

Install Nvidia Driver

When you run the command,

```
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
```

It shows the error, “RuntimeError: Distributed package doesn't have NCCL built in”. Let's learn about NCCL.

The NVIDIA Collective Communication Library (NCCL) implements multi-GPU and multi-node communication primitives optimized for NVIDIA GPUs and Networking. I refer to the below websites to install NVIDIA drivers.

- CUDA Toolkit 12.2 Update 1 Downloads, <https://developer.nvidia.com/cuda-downloads>
- NVIDIA NCCL, <https://developer.nvidia.com/nccl>
- NVIDIA Deep Learning NCCL Documentation, <https://docs.nvidia.com/deeplearning/nccl/install-guide/index.html>
- NVIDIA CUDA Installation Guide for Linux, <https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html>
- After Installing Ubuntu, You Encounter Perform MOK Management, <https://www.cnblogs.com/yutian-blogs/p/13019226.html>
- Ubuntu 22.04 for Deep Learning, <https://gist.github.com/amir-saniyan/b3d8e06145a8569c0d0e030af6d60bea>
- Ubuntu 22.04 Notes, <https://github.com/kmcminn/thinkpad/tree/main/extreme3g>

When we successfully install the NVIDIA driver for our graphic card, and then we use `nvidia-smi` command to show its detail, it can show the below info.

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

Thu Aug 17 04:15:43 2023

NVIDIA-SMI 535.86.10			Driver 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	Memory-Usage	GPU-Util	Compute 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

Processes:						
GPU	GI	CI	PID	Type	Process name	GPU Memory
	ID	ID				Usage
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

Actually, it is hard to reach this phase. Please refer to the link here carefully, Ubuntu 22.04 Notes, <https://github.com/kmcminn/thinkpad/tree/main/extreme3g>.

Learn LLaMA

After downloading the models, and trying to run the command, we will encounter the below error,

```
torch.cuda.OutOfMemoryError: CUDA out of memory. Tried to allocate 86.00 MiB (GPU 0; 11.69 GiB total capacity; 9.70 GiB already allocated; 64.81 MiB free; 9.70 GiB reserved in total by PyTorch) If reserved memory is » allocated memory try setting max_split_size_mb to avoid fragmentation.
```

As the memory of our graphic card is only 12 GB, and the size of the llama-2-7b model is around 13GB, so we can't get it to run with our graphic card.

We try to use the other project, open-llama-3b, https://huggingface.co/openlm-research/open_llama_3b.

We encounter the below error.

```
RuntimeError: Expected all tensors to be on the same device, but found at least two devices, cuda:0 and cpu! (when checking argument for argument index in method wrapper_CUDA__index_select)
```

And we ask ChatGPT about this.

The ChatGPT gives us a very beautiful fix. We need to add the below code.

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

Finally, we can run it up.

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

```
Q: What is the largest animal?
```

```
A: The blue whale.
```

```
Q: What is the largest animal?
```

```
A: The blue whale. It is the largest animal on Earth. It is also the
```

We also try the project, llama2.c, <https://github.com/karpathy/llama2.c>.

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

```
One day, a big lion and a little rabb
```

```
it were in the woods. The big lion said, "Let's race to the river and drink some water." The little rab
```

```
The lion was fast, but he was careful not to hurt the little rabbit. The rabbit was getting close to the
```

```
The little rabbit saw the lion in the water and thought, "I should tell the lion not to drink." He said
```

```
The little rabbit tried to help the lion, but he couldn't. The lion drank too much and his tummy started
```

```
achieved tok/s: 185.527329
```

Understand Code by ChatGPT

Zhiwei: Could you explain the below 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]]]]:
    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: Sure, here's an explanation of the provided code:

```

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]]]]:

```

This function is named `generate` and accepts several parameters:

- `prompt_tokens` is a list containing sequences of tokens for prompts.
- `max_gen_len` is the maximum length of the generated text.
- `temperature` is a parameter used for controlling randomness during generation.
- `top_p` is a probability threshold between 0 and 1 used for controlling diversity during sampling.
- `logprobs` is a boolean indicating whether to record the log probabilities of the generation process.
- `echo` is a boolean indicating whether to echo the input during generation.

```

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

```

Here, the function gets the model parameters and calculates the batch size of the prompts. It ensures that the batch size is not greater than the maximum batch size specified in the model parameters.

```

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)

```

These lines calculate the minimum and maximum lengths of the prompts and ensure that the maximum prompt length does not exceed the maximum sequence length specified in the model parameters. The total length is calculated based on the maximum generated length and the maximum prompt length.

```

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")

```

This section initializes the `tokens` tensor with padding tokens and fills it with the token sequences from the prompts.

```

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

```

If `logprobs` is True, a tensor `token_logprobs` is created to store log probabilities.

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

Variables `prev_pos` (previous position), `eos_reached` (whether end-of-sequence tokens are encountered), and `input_text_mask` (positions not filled with padding) are initialized.

The following loop generates tokens starting from the minimum prompt length up to the total length.

```
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
```

Inside this loop, the model generates a probability distribution `logits` for the next token. If `logprobs` is

True, it computes the log probabilities using cross-entropy. Depending on the `temperature`, it uses softmax or argmax to select the next token. It then updates the `tokens` tensor and checks for end-of-sequence tokens.

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

If `logprobs` is True, the log probabilities tensor is converted to a Python list.

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

In this loop, for each input sample, it generates the output token sequence and associated log probabilities.

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

Finally, the function returns the generated token sequences and their corresponding log probabilities (if requested).