

Controlador de Nvidia, LLaMA y ChatGPT

LLaMA (Large Language Model Meta AI) es una familia de modelos de lenguaje grande (LLMs, por sus siglas en inglés), lanzada por Meta AI a partir de febrero de 2023.

Recientemente construí mi computadora con una GPU Nvidia. Puedes ver cómo hacerlo aquí: Cómo Construir una Computadora, <https://lzwjava.github.io/computer>.

Después de eso, comencé a ejecutar el proyecto LLaMA. La URL del proyecto LLaMA en GitHub es <https://github.com/facebookresearch/llama>.

Instalar el controlador de Nvidia

Cuando ejecutas el comando,

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

Muestra el error: “RuntimeError: Distributed package doesn’t have NCCL built in”. Aprendamos sobre NCCL.

La Biblioteca de Comunicación Colectiva de NVIDIA (NCCL) implementa primitivas de comunicación multi-GPU y multi-nodo optimizadas para las GPU y redes de NVIDIA. Me refiero a los siguientes sitios web para instalar los controladores de NVIDIA.

- Descargas de CUDA Toolkit 12.2 Update 1, <https://developer.nvidia.com/cuda-downloads>
- NVIDIA NCCL, <https://developer.nvidia.com/nccl>
- Documentación de NVIDIA Deep Learning NCCL, <https://docs.nvidia.com/deeplearning/nccl/install-guide/index.html>
- Guía de instalación de NVIDIA CUDA para Linux, <https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html>
- Después de instalar Ubuntu, te encuentras con “Perform MOK Management”, <https://www.cnblogs.com/yutian-blogs/p/13019226.html>
- Ubuntu 22.04 para Deep Learning, <https://gist.github.com/amir-saniyan/b3d8e06145a8569c0d0e030af6>
- Notas de Ubuntu 22.04, <https://github.com/kmcminn/thinkpad/tree/main/extreme3g>

Cuando instalamos exitosamente el controlador de NVIDIA para nuestra tarjeta gráfica y luego usamos el comando `nvidia-smi` para mostrar sus detalles, puede mostrar la siguiente información.

```
(base) lzw@lzw-MS-7E01:~$ nvidia-smi
Jue Ago 17 04:15:43 2023
+-----+
| NVIDIA-SMI 535.86.10      Versión del controlador: 535.86.10  Versión CUDA: 12.2  |
|-----+-----+-----+
| GPU  Nombre          Persistencia-M | Bus-Id      Disp.A | Volatile Uncorr. ECC | | | |
| Ventilador Temp   Rendimiento      Pwr:Uso/Cap |           Uso de memoria | 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 |
+-----+-----+-----+-----+-----+-----+
+-----+
| Procesos:
| GPU  GI  CI      PID  Tipo  Nombre del proceso          Uso de memoria GPU |
|       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 |
+-----+
```

En realidad, es difícil llegar a esta fase. Por favor, consulta cuidadosamente el enlace aquí, Notas de Ubuntu 22.04, <https://github.com/kmcminn/thinkpad/tree/main/extreme3g>.

Aprende LLaMA

Después de descargar los modelos e intentar ejecutar el comando, nos encontraremos con el siguiente error:

`torch.cuda.OutOfMemoryError: CUDA sin memoria. Se intentó asignar 86.00 MiB (GPU 0; capacidad total de 11.69 GiB; 9.70 GiB ya asignados; 64.81 MiB libres; 9.70 GiB reservados en total por PyTorch). Si la memoria reservada es > que la memoria asignada, intenta configurar max_split_size_mb para evitar la fragmentación.`

Como la memoria de nuestra tarjeta gráfica es de solo 12 GB, y el tamaño del modelo llama-2-7b es de aproximadamente 13 GB, no podemos hacerlo funcionar con nuestra tarjeta gráfica.

Intentamos utilizar el otro proyecto, open-llama-3b, https://huggingface.co/openlm-research/open_llama_3b.

Encontramos el siguiente error.

`RuntimeError: Se esperaba que todos los tensores estuvieran en el mismo dispositivo, pero se encontraron al menos dos dispositivos, cuda:0 y cpu! (al verificar el argumento para el argumento index en el método wrapper_CUDA_index_select)`

Y le preguntamos a ChatGPT sobre esto.

El ChatGPT nos proporciona una solución muy elegante. Necesitamos agregar el siguiente código.

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

Finalmente, podemos ejecutarlo.

```
(llama) lzw@lzw-MS-7E01:~/Projects/open_llama_3b$ python run.py
P: ¿Cuál es el animal más grande?
R: La ballena azul.
P: ¿Cuál es el animal más grande?
R: La ballena azul. Es el animal más grande en la Tierra. También es el
```

También probamos el proyecto llama2.c, <https://github.com/karpathy/llama2.c>.

```
(base) lzw@lzw-MS-7E01:~/Projects/llama2.c$ ./run stories15M.bin
Un día, un gran león y un pequeño conejo estaban en el bosque. El gran león dijo: "Hagamos una carrera!
El león era rápido, pero tuvo cuidado de no lastimar al pequeño conejo. El conejo se estaba acercando a...
El pequeño conejo vio al león en el agua y pensó: "Debería decirle al león que no beba". Dijo: "León, no...
El pequeño conejo intentó ayudar al león, pero no pudo. El león bebió demasiado y su barriga comenzó a...
tokens por segundo alcanzados: 185.527329
```

Entender el código con ChatGPT

Zhiwei: ¿Podrías explicar los siguientes códigos?

```
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)

    # solo reemplazar el token si el prompt ya ha sido generado
    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()):
                # cortar a la longitud máxima de generación
                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]
                # cortar al token eos si existe
                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: Claro, aquí tienes una explicación del código proporcionado:

```

def generar(
    self,
    tokens_de_prompt: List[List[int]],
    longitud_max_generacion: int,
    temperatura: float = 0.6,
    top_p: float = 0.9,
    logprobs: bool = False,
    echo: bool = False,
) -> Tuple[List[List[int]], Optional[List[List[float]]]]:

```

Esta función se llama `generate` y acepta varios parámetros:

- `prompt_tokens` es una lista que contiene secuencias de tokens para los prompts.
- `max_gen_len` es la longitud máxima del texto generado.
- `temperature` es un parámetro utilizado para controlar la aleatoriedad durante la generación.
- `top_p` es un umbral de probabilidad entre 0 y 1 utilizado para controlar la diversidad durante el muestreo.
- `logprobs` es un booleano que indica si se deben registrar las probabilidades logarítmicas del proceso de generación.
- `echo` es un booleano que indica si se debe repetir la entrada durante la generación.

```

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

```

En este fragmento de código, se accede a los parámetros del modelo (`params`), se calcula el tamaño del lote (`bsz`) basado en la longitud de los tokens de entrada (`prompt_tokens`), y se verifica que el tamaño del lote no exceda el tamaño máximo permitido por el modelo (`params.max_batch_size`). Si el tamaño del lote es mayor, se lanza una excepción con los valores de `bsz` y `params.max_batch_size`.

Aquí, la función obtiene los parámetros del modelo y calcula el tamaño del lote (batch size) de las indicaciones (prompts). Se asegura de que el tamaño del lote no sea mayor que el tamaño máximo de lote especificado en los parámetros del modelo.

```
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)
```

Estas líneas calculan las longitudes mínima y máxima de los prompts y aseguran que la longitud máxima del prompt no exceda la longitud máxima de secuencia especificada en los parámetros del modelo. La longitud total se calcula en función de la longitud máxima generada y la longitud máxima del prompt.

```
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")
```

Esta sección inicializa el tensor `tokens` con tokens de relleno y lo llena con las secuencias de tokens de los prompts.

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

Si `logprobs` es `True`, se crea un tensor `token_logprobs` para almacenar las probabilidades logarítmicas.

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

Se inicializan las variables `prev_pos` (posición anterior), `eos_reached` (si se han encontrado tokens de fin de secuencia) y `input_text_mask` (posiciones no llenadas con padding).

El siguiente bucle genera tokens comenzando desde la longitud mínima del prompt hasta la longitud total.

```

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

```

Dentro de este bucle, el modelo genera una distribución de probabilidad `logits` para el siguiente token. Si `logprobs` es True, calcula las probabilidades logarítmicas utilizando la entropía cruzada. Dependiendo de la `temperature`, utiliza `softmax` o `argmax` para seleccionar el siguiente token. Luego, actualiza el tensor `tokens` y verifica la presencia de tokens de fin de secuencia.

```

if logprobs:
    token_logprobs = token_logprobs.tolist()

```

Si `logprobs` es True, el tensor de probabilidades logarítmicas se convierte en una lista de Python.

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

```

En este bucle, para cada muestra de entrada, genera la secuencia de tokens de salida y las probabilidades logarítmicas asociadas.

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

Nota: El código no necesita traducción, ya que es un fragmento de código en Python y los nombres de las variables y funciones deben mantenerse en inglés.

Finalmente, la función devuelve las secuencias de tokens generadas y sus correspondientes probabilidades logarítmicas (si se solicita).