

Inference

к.ф.-м.н. Тихомиров М.М.

НИВЦ МГУ имени М. В. Ломоносова

Can i run it LLM?

Can you run it? LLM version

Information

- GPU information comes from [TechPowerUp GPU Specs](#)
- Mainly based on [Model Memory Calculator by hf-accelerate](#) using `transformers` library
- Inference is calculated following [EleutherAI Transformer Math 101](#), where is estimated as

$$\text{Memory}_{\text{Inference}} \approx \text{Model Size} \times 1.2$$

- For LoRa Fine-tuning, I'm assuming a **16-bit** dtype of trainable parameters. The formula (in terms of GB) is

$$\text{Memory}_{\text{LoRa}} \approx \left(\text{Model Size} + \# \text{ trainable Params}_{\text{Billions}} \times \frac{16}{8} \times 4 \right) \times 1.2$$

[NousResearch/Meta-Llama-3-8B-Instruct](#) (7.6B)

int4 int8 float16/bfloat16 float32

int4 refers to models in GPTQ-4bit, AWQ-4bit or Q4_0 GGUF/GGML

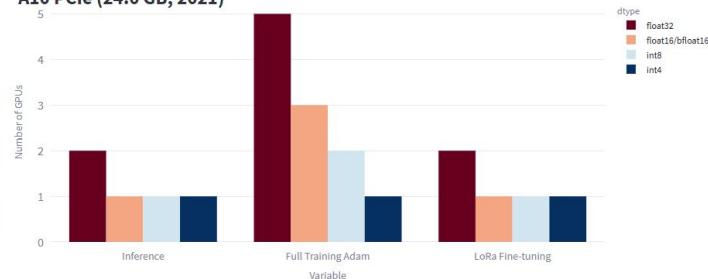
✓ You require 1 GPUs for Inference

✓ You require 1 GPUs for Full Training Adam

✓ You require 1 GPUs for LoRa Fine-tuning (2.0%)

	float32	float16/bfloat16	int8	int4
Total Size (GB)	27.96	13.98	6.99	3.49
Inference (GB)	33.55	16.77	8.39	4.19
Training using Adam (GB)	111.83	55.92	27.96	13.98
LoRA Fine-Tuning (GB)	34.99	18.22	9.83	5.63

Number of GPUs required for A10 PCIe (24.0 GB, 2021)



Через что обычно “генерируют” с LLM

- Huggingface + transformers + pytorch
 - “Нативное” решение, низкая эффективность
- VLLM
 - Один из лучших фреймворков для инференса с нацеленностью на высокую пропускную способность
- llama.cpp (или ее клоны)
 - llama
 - aphrodite

Какие характеристики оценивают

- Занимаемая память
- Токен/секунда
- Пропускная способность при “нагрузке”
 - Обработка батчами (Batch inference)
 - Токен/секунда или RPS (запрос в секунду)
- Время до первого токена
- Поддерживаемые квантизации
- Возможность multigpu/multinode
- Возможность cpu offload

HF + transformers: инициализация

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer, GenerationConfig

MODEL_NAME = "RefalMachine/RuadaptQwen2.5-1.5B-instruct"
model = AutoModelForCausalLM.from_pretrained(
    MODEL_NAME,
    load_in_8bit=False,
    torch_dtype=torch.bfloat16,
    device_map="cuda:0",
    attn_implementation='flash_attention_2'
)
model.eval()

tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
generation_config = GenerationConfig.from_pretrained(MODEL_NAME)
generation_config.max_new_tokens = 512
generation_config.temperature = 0.3
generation_config.repetition_penalty = 1.05
```

HF + transformers: генерация

```
import time
inputs = ["Напиши функцию для генерации ответа через LLM, используя transformers."]
for query in inputs:
    prompt = tokenizer.apply_chat_template([
        {
            "role": "user",
            "content": query
        }
    ], tokenize=False, add_generation_prompt=True)

    data = tokenizer(prompt, return_tensors="pt", add_special_tokens=False)
    data = {k: v.to(model.device) for k, v in data.items()}

    s = time.time()
    output_ids = model.generate(**data, generation_config=generation_config)[0]
    output_ids = output_ids[len(data["input_ids"][0]):]
    output = tokenizer.decode(output_ids, skip_special_tokens=True).strip()
    gen_time = time.time() - s
    print(len(output) / gen_time)
    print(output)
```

HF + transformers: пример результата

167.16134602604677

Для генерации ответа через LLM (Language Model) с использованием библиотеки `transformers` из PyTorch, вам нужно выполнить следующие шаги:

```
1. **Установите необходимые библиотеки**:  
```bash  
pip install transformers torch
```  
  
2. **Создайте функцию генерации ответа**:  
```python  
from transformers import AutoModelForCausalLM, AutoTokenizer

def generate_response(prompt):
 # Устанавливаем модель и токенайзер
 model_name = "EleutherAI/gpt-neo-125M"
 tokenizer = AutoTokenizer.from_pretrained(model_name)
 model = AutoModelForCausalLM.from_pretrained(model_name)

 # Предлагаемый запрос
 input_text = prompt

 # Преобразуем текст в токены
 inputs = tokenizer.encode(input_text, return_tensors='pt')

 # Генерируем ответ
 with torch.no_grad():
 outputs = model.generate(inputs, max_length=50, num_return_sequences=1)

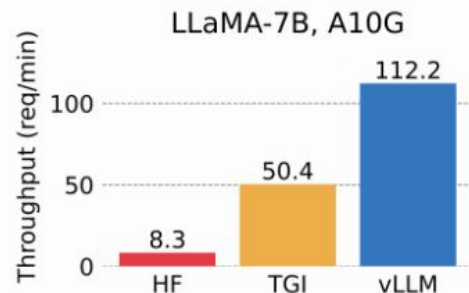
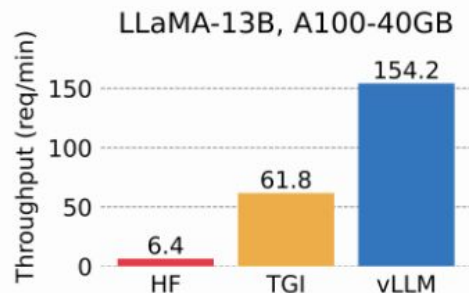
 # Переводим ответ обратно в текст
 response_text = tokenizer.decode(outputs[0], skip_special_tokens=True)

 return response_text

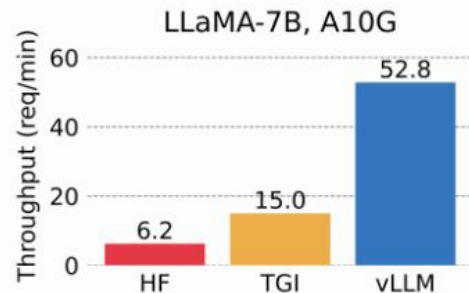
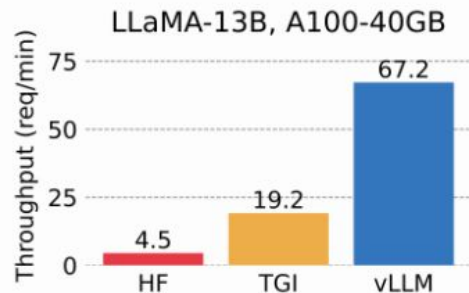
Пример использования
prompt = "Какой будет следующий шаг в разработке проекта?"
print(generate_response(prompt))
```
```

VLLM

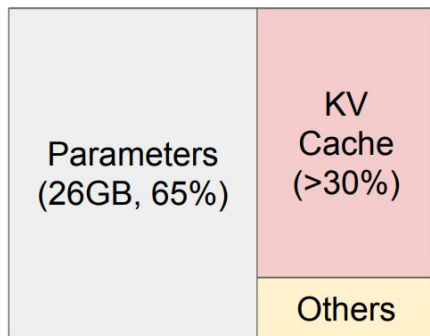
- Одно из главных узких мест инференса LLM - память
- В частности классический attention + kv cache потребляет много (1.7GB for a single sequence in LLaMA-13B)
- Реальное ускорение не такое большое, как на графике



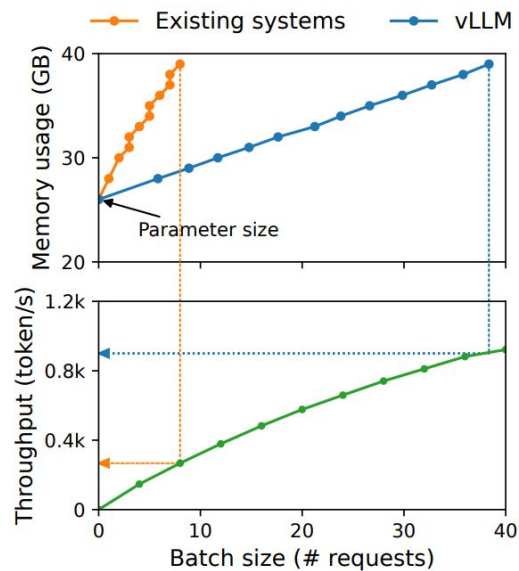
Serving throughput when each request asks for *one output completion*. vLLM achieves 14x - 24x higher throughput than HF and 2.2x - 2.5x higher throughput than TGI.



VLLM



NVIDIA A100 40GB



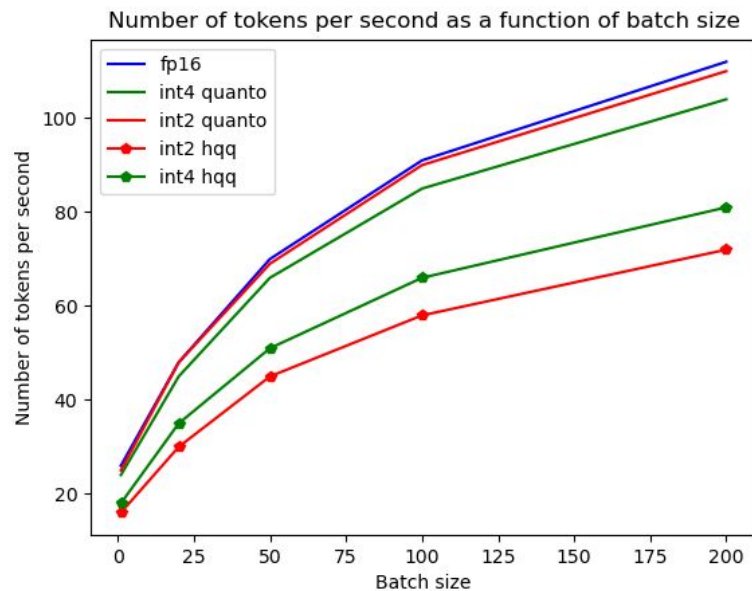
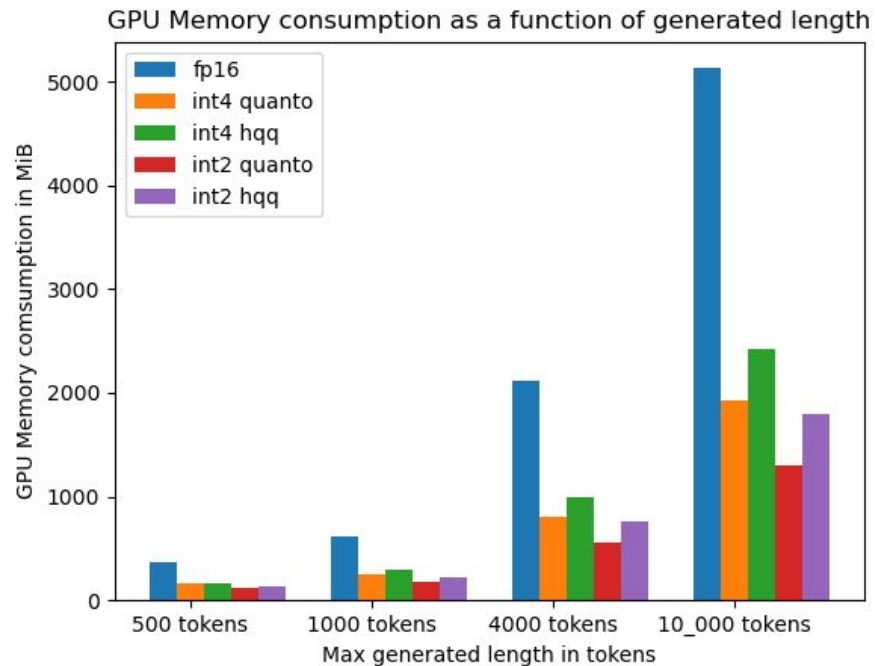
KV cache

- Мы генерируем токен за токеном, высчитывая на каждом слое $QKV \dots$
- K и V не меняются для будущих токенов, так как зависят только от прошлых, и они нужны для расчета будущих токенов
- Их можно закешировать!

Для одного токена требуется байт (в fp16)

$$2 \cdot 2 \cdot n_{\text{layers}} \cdot n_{\text{heads}} \cdot d_{\text{head}}$$

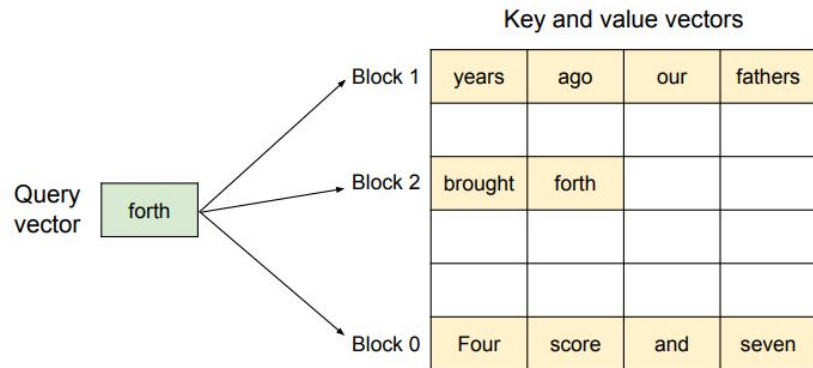
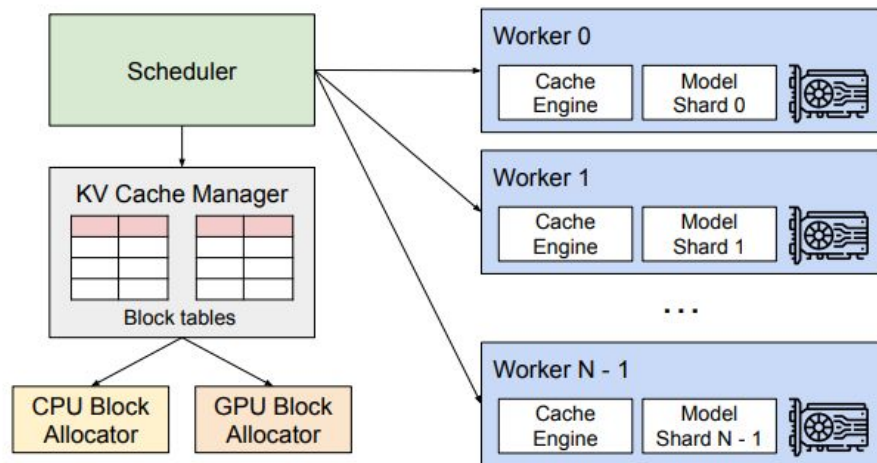
KV cache: квантизированный



Paged Attention

- Одна из основных фич vLLM
- Разбивает kv-cache на части, эффективно их хранит и обращается по аналогии с страничной организацией памяти в ОС
- При одинаковых префиксах (промптах) и множественном декодировании, для разных последовательностей общая часть хранится только в едином экземпляре
- Во время расчета внимания, PagedAttention определяет и извлекает разные блоки KV-кеша по отдельности.

Paged Attention



VLLM: инициализация

```
from vllm import LLM, SamplingParams
from transformers import AutoTokenizer

MODEL_NAME = "RefalMachine/RuadaptQwen2.5-1.5B-instruct"
sampling_params = SamplingParams(
    temperature=0.01,
    top_p=0.9,
    top_k=40,
    max_tokens=1000,
    repetition_penalty=1.0,
)
llm = LLM(
    model=MODEL_NAME,
    max_seq_len_to_capture=4096,
    max_model_len=4096
)
tokenizer = llm.get_tokenizer()
```

VLLM: генерация

```
import time
inputs = ["Напиши функцию для генерации ответа через LLM, используя transformers."]
for query in inputs:
    messages = [{
        "role": "user",
        "content": query
    }]
    prompt = tokenizer.apply_chat_template(
        messages, tokenize=True, add_generation_prompt=True
    )
    s = time.time()
    outputs = llm.generate(prompt_token_ids=[prompt], sampling_params=sampling_params)
    gen_time = time.time() - s
    for output in outputs:
        generated_text = output.outputs[0].text
        print(len(generated_text) / gen_time)
        print(generated_text)
```

VLLM: пример результата

866.654579678111

Для генерации ответа через LLM (Language Model) с использованием библиотеки `transformers` из PyTorch, вам нужно выполнить следующие шаги:

1. ****Установите библиотеки**:**

```
```bash
pip install transformers torch
```
```

2. ****Создайте функцию генерации ответа**:**

```
```python
from transformers import AutoModelForCausalLM, AutoTokenizer
import torch

def generate_response(prompt, model_name='qwen', max_length=50):
 # Загрузка модели и токенизатора
 model = AutoModelForCausalLM.from_pretrained(model_name)
 tokenizer = AutoTokenizer.from_pretrained(model_name)

 # Предварительная подготовка данных
 prompt = prompt.strip()
 inputs = tokenizer(prompt, return_tensors='pt')

 # Генерация ответа
 with torch.no_grad():
 outputs = model.generate(**inputs, max_length=max_length, num_return_sequences=1)

 # Преобразование ответа в текст
 response = tokenizer.decode(outputs[0], skip_special_tokens=True)

 return response

Пример использования
prompt = "Какой будет следующий сезон сериала 'Звездные войны'?"
response = generate_response(prompt)
```



# VLLM: openai service

```
from openai import OpenAI

oclient = OpenAI(api_key='123', base_url=url)

myprompt = '''**Instruction**
Ответь на вопрос, но строго следуй формату:

<thoughts>
Твои внутренние размышления на русском (пользователь их не видит)
</thoughts>
<answer>
Ответ для пользователя
</answer>

Question
Сколько букв а в слове балалаайка?'''
res = oclient.chat.completions.create(
 model=model_name,
 messages=[{"role": "system", "content": "You are a helpful and harmless assistant. You should think step-by-step."},
 {"role": "user", "content": myprompt}],
 temperature=0.01,
 top_p=0.9,
 max_tokens=1000,
 extra_body={
 "repetition_penalty": 1.0,
 "add_generation_prompt": True,
 }
)
if res.choices is not None:
 res = res.choices[0].message.content.strip()
print(res)
```

```
python -u -m vllm.entrypoints.openai.api_server \
--model /workdir/data/models/qwen/QwQ-32B-Preview \
--gpu-memory-utilization 0.9 \
--max_model_len 8000 \
--max_num_seqs 100 \
--port 8888
```

# VLLM: openai service: пример ответа

<thoughts>

Сколько букв "а" в слове "балалайка"? Хм, нужно посчитать количество букв "а" в этом слове. Слово "балалайка" - это название музыкального инструмента, но мне нужно сосредоточиться на счете букв.

Давай разобьем слово на отдельные буквы, чтобы легче было посчитать. Б - А - Л - А - Л - А - А - Ы - К - А. Поставлю их в ряд:

Б А Л А Л А А Ы К А

Теперь пересчитаем количество "а". Смотрим каждую букву:

1. Б - не а

2. А - это а (1)

3. Л - не а

4. А - это а (2)

5. Л - не а

6. А - это а (3)

7. А - это а (4)

8. Ы - не а

9. К - не а

10. А - это а (5)

Так, получается пять букв "а". Но лучше перепроверить, может быть, я ошиблась.

Повторяю счет:

Б А Л А Л А А Ы К А

А - 1, А - 2, А - 3, А - 4, А - 5. Да, действительно пять букв "а".

А может быть, в слове "балалайка" меньше "а"? Нет, слово именно "балалайка", с двумя "а" подряд в середине. Да, пять "а".

Уверена в ответе.

</thoughts>

<answer>

В слове "балалайка" пять букв "а".

</answer>

# VLLM: openai service: пример ответа

<thoughts>

Пользователь спрашивает, сколько букв "й" в слове "балалаайка". Сначала нужно понять, как правильно написано это слово. Обычно слово "балалайка" пишется без extra "й", но возможно здесь опечатка или особенное написание. Проверим правописание слова "балалайка". Да, стандартное написание – балалайка, с одной "й". Если пользователь написал "балалаайка", то возможно, это опечатка, и ли он хочет специально вставить extra "й". Нужно уточнить, но сначала можно посчитать количество "й" в слове "балалаайка", как написано пользователем.

Слово "балалаайка" состоит из следующих букв: б, а, л, а, л, а, а, й, к, а. Давайте перечислим все "й": видим только одну "й" перед "к". Поэтому, в этом слове одна буква "й".

Но, возможно, пользователь имел в виду стандартное слово "балалайка". В этом случае, слово "балалайка" содержит две буквы "й": первая "й" между "л" и "а", вторая "й" перед "к". Но в данном случае пользователь написал "балалаайка", что может быть опечаткой.

Чтобы быть точным, нужно уточнить, но сначала можно ответить на основании того, как написано.

</thoughts>

<answer>

В слове "балалаайка", как вы его написали, буква "й" встречается один раз. Однако, если вы имели в виду стандартное слово "балалайка", то там две буквы "й". Пожалуйста, уточните, если нужно.

</answer>

## Offline Batched inference: сравнение

- Случай с длинной генерацией - огромное ускорение
- В случае генерации 1 токена, ускорения подобного нет

```
import time
prompt_token_ids = [tokenizer(d, add_special_tokens=False)['input_ids'] for d in dataset]
s = time.time()
outputs = llm.generate(prompt_token_ids=prompt_token_ids, sampling_params=sampling_params)
gen_time = time.time() - s
total_len = 0
total_tokens = 0
for output in outputs:
 generated_text = output.outputs[0].text
 total_len += len(generated_text)
 total_tokens += len(output.outputs[0].token_ids)
print(total_len / gen_time)
print(total_tokens / gen_time)
```

```
Processed prompts: 100% ██████████ 500/500 [00:31<00:00, 16.13it/s, est. speed input: 459.52 toks/s, output: 5921.70 toks/s]
```

24127.652771122684  
5912.201165616972

```
import time
from tqdm import tqdm
bs = 16

total_len = 0
s = time.time()
for i in tqdm(range(0, len(dataset[:100]), bs)):
 res = model.generate_batch(dataset[i:i+bs])
 batch_char_len = sum([len(t) for t in res[1]])
 total_len += batch_char_len
total_time = time.time() - s

print(total_len / total_time)
```

100% | 7/7 [02:14<00:00, 19.22s/it

1259.603208250194

# Multi-gpu/Multi-node

vLLM позволяет очень легко поднять модель в multi-gpu с использованием tensor parallel или pipeline parallel, передав флаги, например:

```
--tensor-parallel-size 4
--pipeline-parallel-size 2
```

Предусмотрена возможность multi-node.

vLLM поддерживает квантизации: GPTQ, AWQ,

# llamacpp

- Другой популярный фреймворк для инференса LLM
- Фокус не на пропускную способность, но на возможность запуска в целом:
  - Свои типы для квантизации GGUF
  - Возможность выгрузки слоев на CPU (cpu offload)
  - Множество форков и успешных “клонов” (ollama и др)

# llamacpp

```
root@e735b069dcc2:/workdir/projects/llama.cpp# CUDA_VISIBLE_DEVICES=0 ./llama-cli -m RuadapTQwen2.5-32B-instruct-GGUF/Q4_K_M.gguf -n 512 -cnv -p "You are Qwen, created by Alibaba Cloud. You are a helpful assi
```

0	NVIDIA A100-SXM4-80GB	Off	00000000:07:00.0 Off	0
N/A	26C P0	74W / 400W	7766MiB / 81920MiB	0% Default Disabled

```
system
```

```
You are Qwen, created by Alibaba Cloud. You are a helpful assistant.
```

```
> Привет, как запустить llama-cpp cli с offload половины слоев на CPU?
```

```
Привет! Для запуска Llama.cpp с offload (переключением нагрузки) половины слоев на CPU, нужно воспользоваться следующими параметрами при запуске CLI версии Llama.cpp. Важно помнить, что точные параметры и поддержка могут зависеть от версии Llama.cpp, которую вы используете. Вот пример команды для запуска с offloading половины слоев на CPU:
```

```
```bash
```

```
./main -m your_model.bin -ngl 1
```

```
```
```

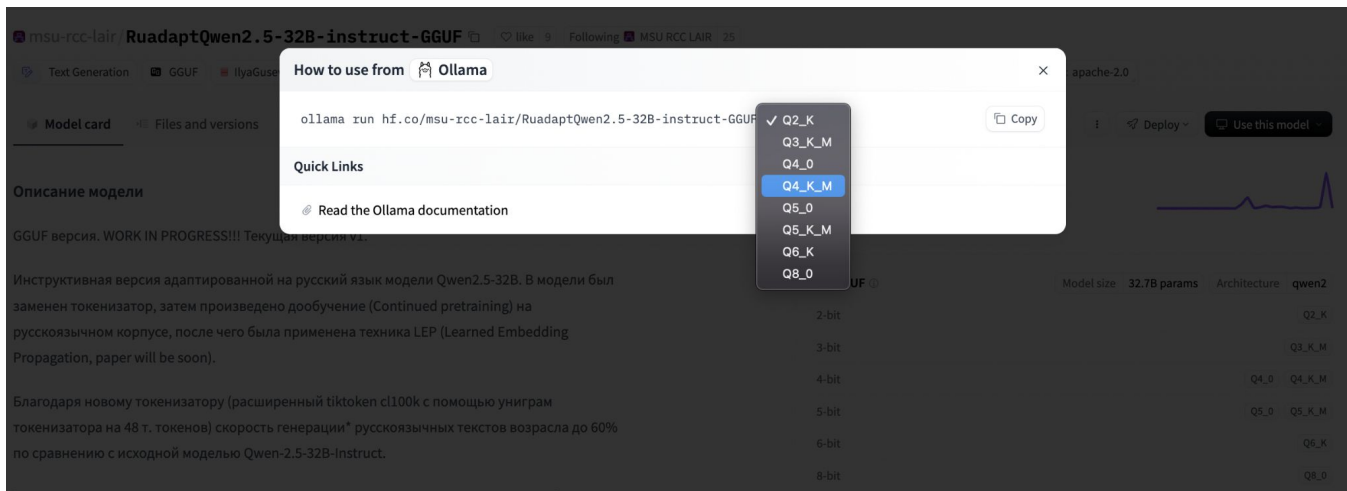
```
В этом примере:
```

```
- `-m your_model.bin` - указывает путь к вашему моделируемому файлу.
```

```
- `-ngl 1` (Number of GPU Layers) - указывает количество слоев, которые остаются на GPU, при этом остальные слои будут автоматически offload'нуты на CPU.
```

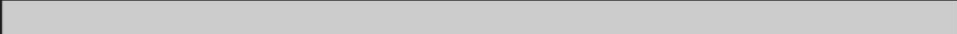

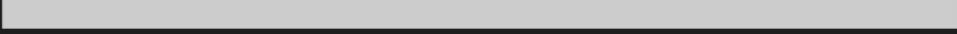
# ollama

- Удобный и популярный клон Llamasrpp
- Можно тянуть GGUF модели прямо с HF
- 





# ollama

```
root@e735b069dcc2:/workdir/projects/llama.cpp# ollama run hf.co/msu-rcc-lair/RuadaptQwen2.5-32B-instruct-GGUF:Q4_K_M
pulling manifest
pulling 5425034a1864... 100%  19 GB
pulling e94a8ecb9327... 100%  1.6 KB
pulling 562b337b9498... 100%  193 B
verifying sha256 digest
writing manifest
success
```

# ollama

>>> Чем ollama отличается от llama-cpp?

**\*\*Ollama\*\*** и **\*\*LLaMA-CPP\*\*** – это две разные реализации, но обе работают с моделями семейства LLaMA.

### Ollama:

- Это более новый проект.
- Разработанный как замена для `llama.cpp`, он стремится упростить работу с модели LLM (Large Language Model).
- Ориентирован на удобство и производительность: включает в себя дополнительные функции, такие как автоматическое управление моделью, возможность быстрого переключения между различными моделями LLaMA и поддержка других моделей.
- Написан на Rust и C++, что делает его относительно более быстрым и безопасным с точки зрения памяти.
- Позволяет работать через простой API или командную строку.

### LLaMA-CPP:

- Это первая известная реализация, поддерживающая моделирование модели LLaMA с использованием только CPU (без поддержки GPU).
- Написано на чистом C++ под руководством команды разработчиков от Facebook Research и сообщества open-source.
- Предоставляет низкоуровневый доступ к функциям модели, что делает его мощным для тех, кто хочет кастомизировать или интегрировать модель в свои собственные приложения напрямую.

В общем, если вы ищете более современный подход с дополнительной функциональностью и удобством использования (особенно через API), Ollama может быть предпочтительнее. Однако, для разработчиков, которым нужна полная гибкость на низкоуровневом уровне или те, кому важен полностью открытый исходный код без дополнительных абстракций, `LLaMA-CPP` всё ещё остаётся отличным выбором.

### Вывод:

- **\*\*Ollama\*\***: Простота использования, дополнительные функции и производительность.
- **\*\*LLaMA-CPP\*\***: Низкоуровневая гибкость, полное управление.