Современные подходы оптимизации LLMs

Козлова Оксана, Железин Михаил

План исследования

В этой работе будут исследованы несколько подходов для уменьшения времени инференса нейронной сети:
- Оптимизация вычислительного графа
- Использование kv-caching для тансформеров
- Tensor parallelism

Для анализа существующих решений взяты следующие статьи:

Оптимизация графа вычислений:
- Tri Dao, Daniel Y. Fu**. FlashAttention**: Fast and Memory-Efficient Exact Attention with IO-Awareness
- https://huggingface.co/blog/optimize-llm
- FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning
- Zhihao Jia, Todd Warszawski . **TASO:** Optimizing Deep Learning Computation with Automatic Generation of Graph.

kv-caching:
- https://huggingface.co/blog/optimize-llm, https://medium.com/@joaolages/kv-caching-explained-276520203249
- Efficient Memory Management for Large Language Model Serving with PagedAttention, https://arxiv.org/pdf/2309.06180.pdf

Tensor parallelism:
- https://www.deepspeed.ai/tutorials/automatic-tensor-parallelism/#example-script
- https://blog.gopenai.com/understanding-tensor-parallelism-to-fit-larger-models-on-multiple-devices-d4da1821d41b

Анализ статей и Эксперименты

FlashAttention

В этой статье предлагается изменять вычисление слоя **Attention** для уменьшения количества обращений к HBM при его вычислении.

Стандартный Attention вычисляется следующим образом:

Algorithm 0 Standard Attention Implementation

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.

1: Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \mathrm{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
4: Return $\mathbf{O}$.

Происходит много загрузок из HBM в SRAM и обратно

В методе **Flash-attention** мы берем предобученные веса для self-attention и изменяем порядок вычислений, разбивая матрицы на блоки и загружаем их в SRAM и делая все вычисления над этим кусочком не выгружая его в HBM и обратно не загружая. Этот способ не уменьшает количество самих вычислений, уменьшает только количество обращений к памяти, А работаем мы с исходными весами не переобучая сеть заново, просто изменяя граф вычислений.

Основная идея заключается в том, что мы разбиваем входные данные Q, K, V на блоки, загружаем их из медленного HBM в быстрый SRAM, затем вычисляем выходные данные внимания относительно этих блоков. Масштабируя выходные данные каждого блока на правильный коэффициент нормализации перед их суммированием, мы получаем правильный результат в конце.

Algorithm 1 FlashAttention

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size M.

Set block sizes $B_c = \lceil \frac{M}{4d} \rceil$, $B_r = \min\left(\lceil \frac{M}{4d} \rceil, d\right)$.

Initialize $\mathbf{O} = (0)_{N \times d} \in \mathbb{R}^{N \times d}$, $\ell = (0)_N \in \mathbb{R}^N$, $m = (-\infty)_N \in \mathbb{R}^N$ in HBM.

Divide $\mathbf{Q}$ into $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ blocks $\mathbf{Q}_1, \ldots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \left\lceil \frac{N}{B_c} \right\rceil$ blocks $\mathbf{K}_1, \ldots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \ldots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.

Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_i, \ldots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide ℓ into T_r blocks $\ell_i, \ldots, \ell_{T_r}$ of size B_r each, divide m into T_r blocks $m_1, \ldots, m_{T_r}$ of size B_r each.

for $1 \le j \le T_c$ **do**

  Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.

  for $1 \le i \le T_r$ **do**

    Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.

    On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.

    On chip, compute $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.

    On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.

    Write $\mathbf{O}_i \leftarrow \text{diag}(\ell_i^{\text{new}})^{-1}(\text{diag}(\ell_i) e^{m_i - m_i^{\text{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.

    Write $\ell_i \leftarrow \ell_i^{\text{new}}$, $m_i \leftarrow m_i^{\text{new}}$ to HBM.

  end for

end for

Return $\mathbf{O}$.

Практическая часть

Этот метод оптимизации широко применяется для LLM. Мы решили провести первые эксперименты с этим методом с **GPT-2** из-за ограниченных мощностей, далее проведем эксперименты над более тяжеловесными моделями.

Базовая модель до экспериментов

Время

```
ime

start_time = time.time()
result = pipe(long_prompt, max_new_tokens=60)[0]["generated_text"][len(long_prompt):]

print(f"Generated in {time.time() - start_time} seconds.")
result
```

Python

```
Generated in 1.307373285293579 seconds.
```

Память

```
bytes_to_giga_bytes(torch.cuda.max_memory_allocated())
```

```
0.3001108169555664
```

После применения Flash-attention

Время

```
= time.time()
.backends.cuda.sdp_kernel(enable_flash=True, enable_math=False, enable_mem_efficient=True):
    result = pipe(long_prompt, max_new_tokens=60)[0]["generated_text"][len(long_prompt):]

print(f"Generated in {time.time() - start_time} seconds.")
result
```

Python

```
Generated in 1.127269983291626 seconds.
```

Память

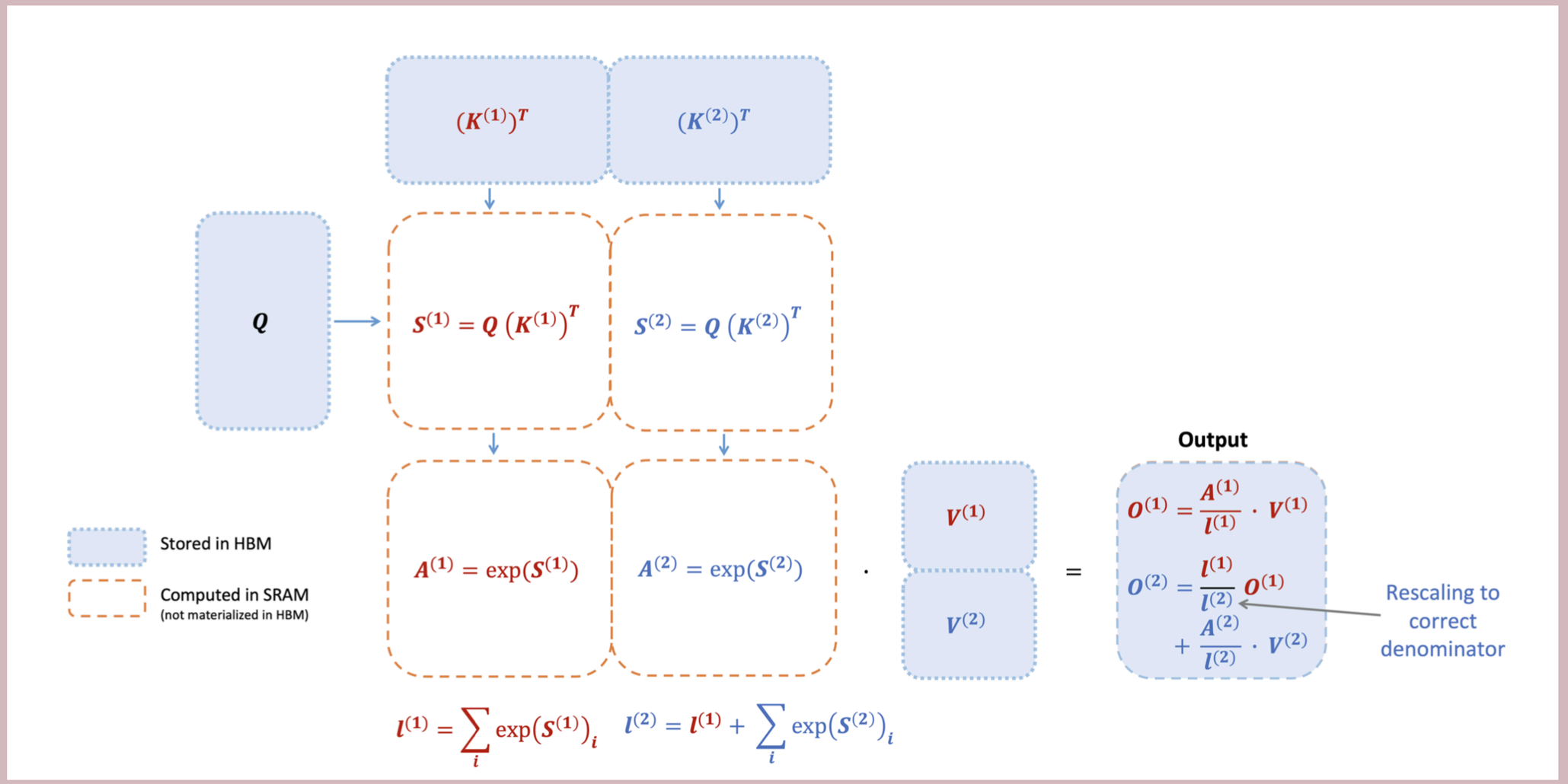
```
bytes_to_giga_bytes(torch.cuda.max_memory_allocated())
```

```
0.33203601837158203
```

Получилось **ускорить инференс нейронной сети на 14%** за счет применения алгоритма, По результатам видно что немного увеличился размер задействовано памяти, но это можно объяснить тем, что у модели GPT-2 слой attention отличается от более крупных LLM. Будем продолжать эксперименты

FlashAttention

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{N\times N}, \quad \mathbf{P} = \mathrm{softmax}(\mathbf{S}) \in \mathbb{R}^{N\times N}, \quad \mathbf{O} = \mathbf{P}\mathbf{V} \in \mathbb{R}^{N\times d},$$



FlashAttention-2

FlashAttention-2 уменьшает количество non-matrix multiplication.

Уменьшение количества non-matrix операций связано с тем, как они обновляют и масштабируют данные в процессе работы.

Вместо того, чтобы постоянно изменять размер этих данных на каждом шаге, они решили делать это только в самом конце процесса.

Flash-attention

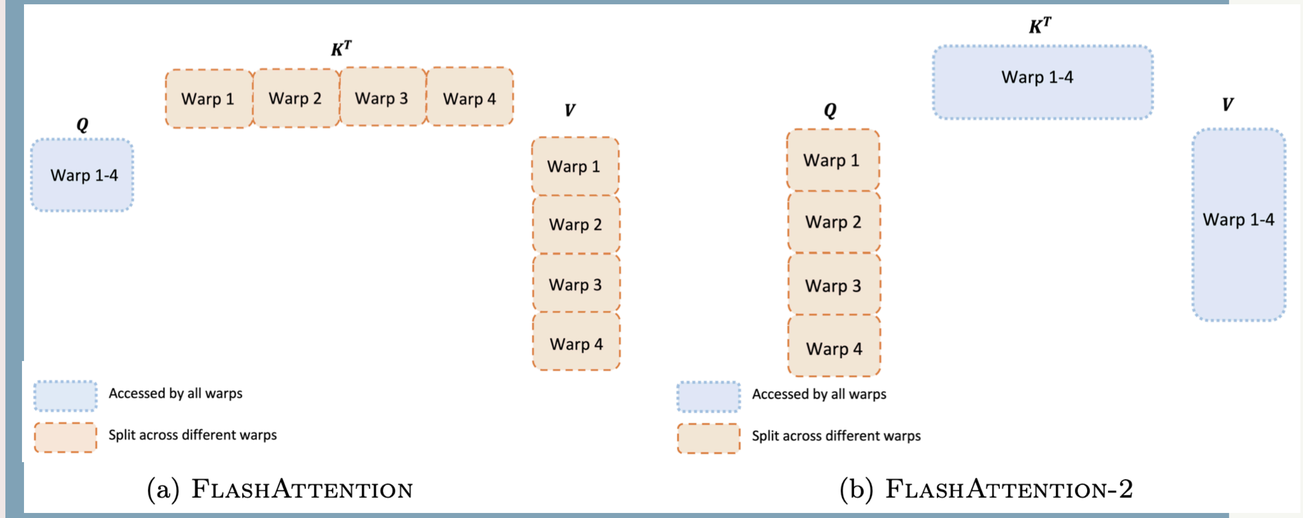
We do not have to rescale both terms of the output update by $\mathrm{diag}(\ell^{(2)})^{-1}$:

$$\mathbf{O}^{(2)} = \mathrm{diag}(\ell^{(1)}/\ell^{(2)})^{-1}\mathbf{O}^{(1)} + \mathrm{diag}(\ell^{(2)})^{-1}e^{\mathbf{S}^{(2)}-m^{(2)}}\mathbf{V}^{(2)}.$$

Flash-attention-2

$$\tilde{\mathbf{O}}^{(2)} = \mathrm{diag}(\ell^{(1)})^{-1}\mathbf{O}^{(1)} + e^{\mathbf{S}^{(2)}-m^{(2)}}\mathbf{V}^{(2)}.$$

- В FlashAttention ключи (K) и значения (V) распределялись по разным warps, в то время как запросы (Q) были доступны всем warps. Это требовало записи промежуточных результатов в общую память и последующей их синхронизации
- В FlashAttention-2 подход был изменен: теперь Q распределяется между варпами, а K и V доступны всем warps. Это позволяет каждому warp выполнять умножение матриц для получения своего сегмента результатов, не требуя дополнительной коммуникации между варпами.



(a) FlashAttention (b) FlashAttention-2

Запуск без flash-attention, время инференса

```
import time

start_time = time.time()
result = pipe(long_prompt, max_new_tokens=60)[0]["generated_text"][len(long_prompt):]

print(f"Generated in {time.time() - start_time} seconds.")
result
```

```
Generated in 1.2615265846252441 seconds.
```

Память

```
bytes_to_giga_bytes(torch.cuda.max_memory_allocated())
```

```
1.9718608856201172
```

На 60% уменьшили время инференса

И на 35% уменьшили затраченную память

Запуск с flash-attention-2, время инференса

```
import time

start_time = time.time()
with torch.backends.cuda.sdp_kernel(enable_flash=True, enable_math=False, enable_mem_efficient=False):
    result = pipe(long_prompt, max_new_tokens=60)[0]["generated_text"][len(long_prompt):]

print(f"Generated in {time.time() - start_time} seconds.")
result
```

```
Generated in 0.50432014465332 seconds.
```

Память

```
bytes_to_giga_bytes(torch.cuda.max_memory_allocated())
```

```
1.309723377227783
```

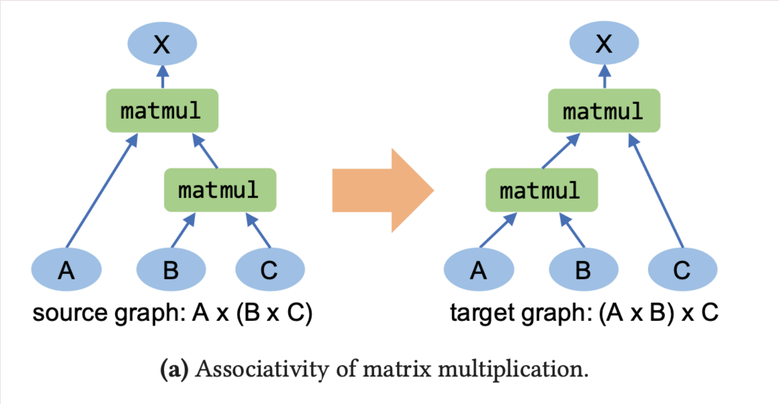
TASO

TASO - это оптимизатор графов вычислений нейронных сетей, который автоматически генерирует разные вариации графов вычислений.

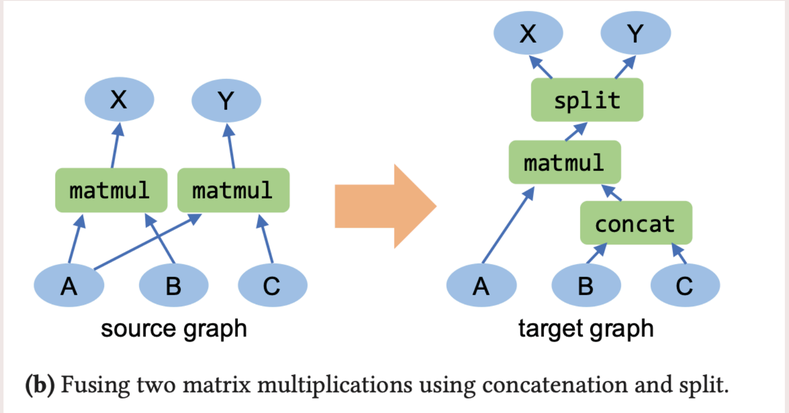
Для разных архитектур оптимальный граф вычислений будет отличаться. Например data layout может оказать большое влияние на производительность во время выполнения. Наилучшая компоновка зависит как от оператора, так и от оборудования. Например, на GPU P100 свертка лучше всего выполняется при row-major layout.

TASO необходимо подать на вход допустимые операторы и оно сгенерит доступные графы вычислений и проверит их корректность.

Приведем примеры возможных изменений в вычислении графа:



(a) Associativity of matmatrix multiplication.



(b) Fusing two matrix multiplications using concatenation and split.

На рисунке **a** показан пример графа с использованием ассоциативности умножения матриц. На рисунке **b** два матричных умножения объединяются в одно с использованием конкатенации и разделения по размеру строки.

Algorithm 1 Graph substitution generation algorithm.

```
1:  Input: A set of operators 𝒫, and a set of input tensors ℐ.
2:  Output: Candidate graph substitutions 𝒮.
3:
4:  // Step 1: enumerating potential graphs.
5:  𝒟 = {}  // 𝒟 is a graph hash table indexed by their fingerprints.
6:  Build(1, ∅, ℐ)
7:  function Build(n, 𝒢, ℐ)
8:      if 𝒢 contains duplicated computation then
9:          return
10:     𝒟 = 𝒟 + (FingerPrint(𝒢), 𝒢)
11:     if n < threshold then
12:         for op ∈ 𝒫 do
13:             for i ∈ ℐ and i is a valid input to op do
14:                 Add operator op into graph 𝒢.
15:                 Add the output tensors of op into ℐ.
16:                 Build(n + 1, 𝒢, ℐ)
17:                 Remove operator op from 𝒢.
18:                 Remove the output tensors of op from ℐ.
19:
20: // Step 2: testing graphs with identical fingerprint.
21: 𝒮 = {}
22: for 𝒢1, 𝒢2 ∈ 𝒟 with the same FingerPrint(·) do
23:     if 𝒢1 and 𝒢2 are equivalent for all test cases then
24:         𝒮 = 𝒮 + (𝒢1, 𝒢2)
25: return 𝒮
```

TASO сначала генерирует все потенциальные графы, используя заданный набор операторов. Входными тензорами могут быть начальные входные тензоры или выходные тензоры предыдущих операторов.

Строки 7-18 показывает алгоритм поиска в глубину для построения всех графов ациклических вычислений, которые не содержат дублирующихся вычислений. Генератор игнорирует операторы выполняющие одно и то же вычисление с одними и теми же входными тензорами.

Для операторов с одинаковым хэшем (FingerPrint) дополнительно проверяет их эквивалентность на наборе рандомных тестовых примеров.

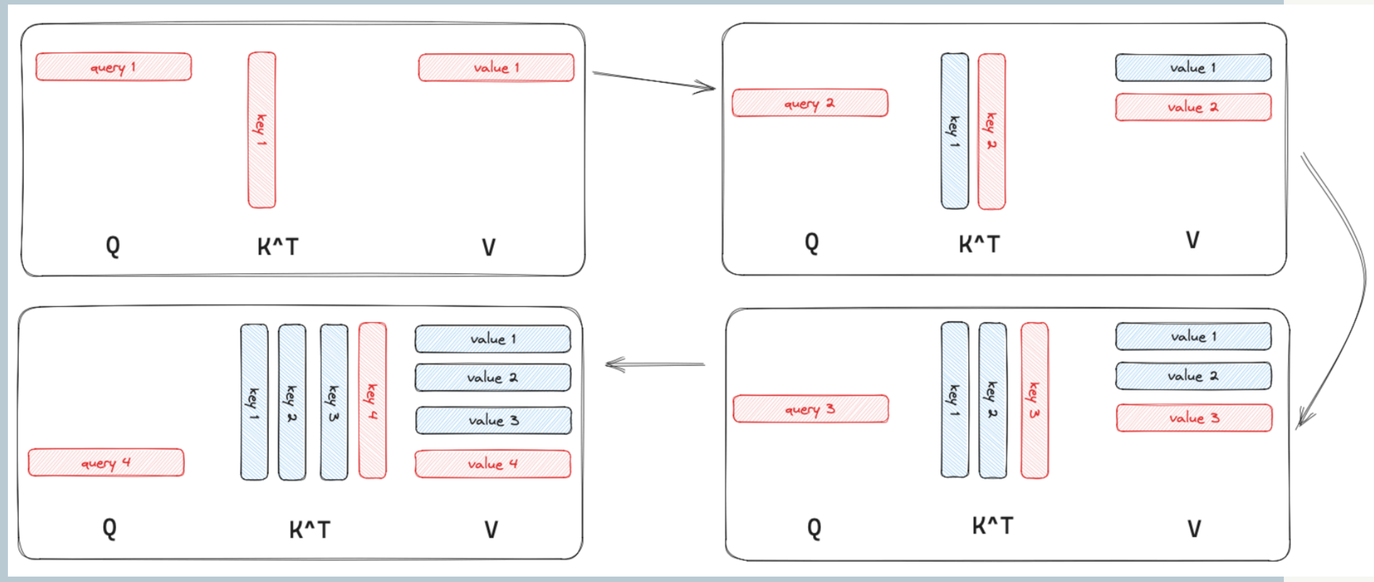
Все возможные графовые подстановки затем проходят проверку их корректности

Transformers KV Caching

При вычислении self-attention мы вычисляем скалярное произведение вектора q на векторы k_i каждого другого токена перед ним во входной последовательности. При добавлении нового элемента в последовательность нам приходится заново вычислять взвешенные суммы для всех предыдущих элементов. Это неэффективно, особенно когда длина последовательности увеличивается.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

KV-caching – это ***сохранение уже вычисленных пар ключ-значение*** для предыдущих элементов последовательности. Когда добавляется новый элемент, вместо того чтобы заново проходиться по всей последовательности, мы используем сохраненные значения. Это значительно ускоряет процесс.

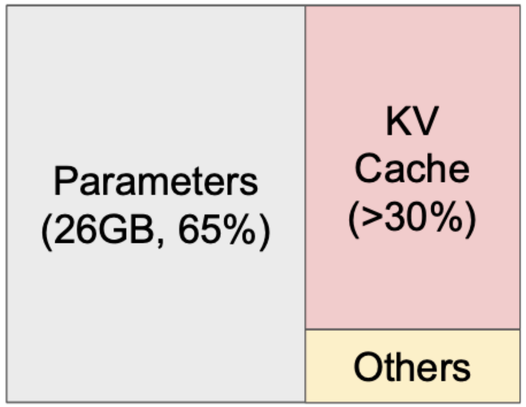


Запустили инференс GPT-2 на коллабе с использованием KV-caching и без него и замерили время работы:

with KV caching: 12.26 +- 1.367 seconds

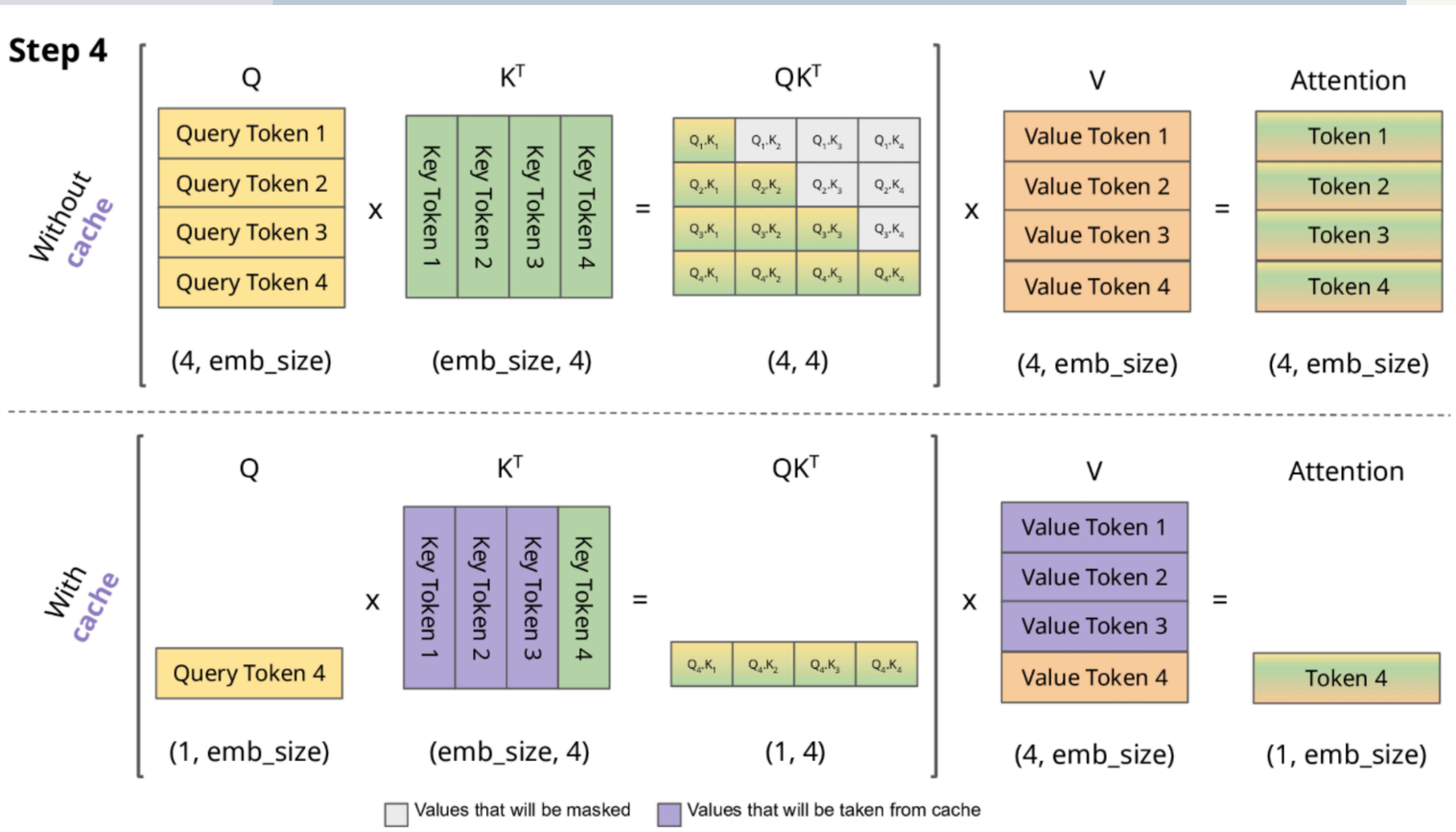
without KV caching: 54.873 +- 0.393 seconds

На 78% уменьшили время инференса



Затраченную память при добавлении kv-caching самостоятельно померить пока не успели.

Опираясь на статью это около 30% от веса параметров.



Tensor parallelism

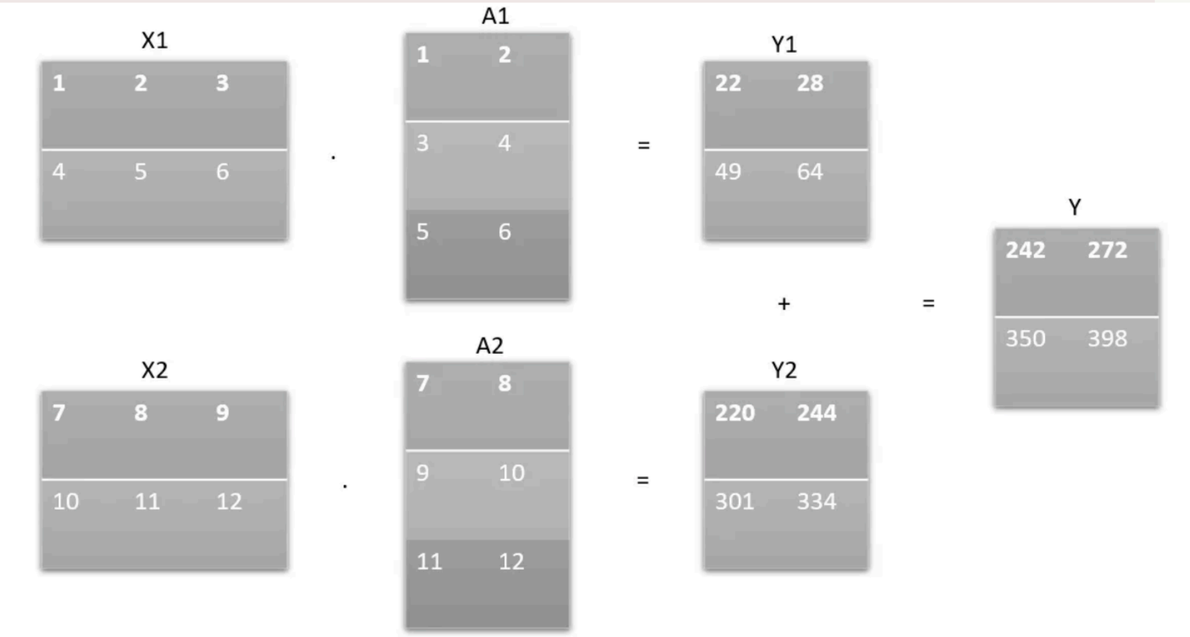
```
TransformerBlock(
   (input_layernorm): LayerNorm((4096,), eps=1e-05,
elementwise_affine=True)
   (self_attention): Attention(
     (q_proj): Linear(in_features=4096, out_features=4096, bias=False)
     (k_proj): Linear(in_features=4096, out_features=4096, bias=False)
     (v_proj): Linear(in_features=4096, out_features=4096, bias=False)
     (o_proj): Linear(in_features=4096, out_features=4096, bias=False)
   )
   (post_attention_layernorm): LayerNorm((4096,), eps=1e-05,
elementwise_affine=True)
   (mlp): MLP(
     (dense_h_to_4h): Linear(in_features=4096, out_features=16384,
bias=False)
     (dense_4h_to_h): Linear(in_features=16384, out_features=4096,
bias=False)
   )
)
```

Пример слоя для оптимизации

Для параллельного вычисления линейных слоёв на нескольких устройствах нужно разделить веса на столько частей, сколько есть устройств, и выполняем перемножение подматриц отдельно на каждом из устройств. В конце необходимо объединить все выходы вместе.

Разделение весов для примера выше

- h to 4h column parallel linear->(1, 10, 4096) . (4096, 8192) = (1, 10, 8192)
- 4h to h row parallel linear->(1, 10, 8192) . (8192, 4096) = (1, 10, 4096)
- All reduce operation



Consider X1 and X2 as dense_h_to_4h outputs on the 2 devices; A1 and A2 as dense_4h_to_h divided weights between the 2 devices; all reduce Y1 and Y2 to get final Y output;

Запуск без параллелизма

```
pipe = transformers.pipeline(task="text2text-generation", model="google/t5-v1_1-small", device=0)
```

Время инференса

```
[39]: import time

start_time = time.time()
result = pipe('Input String')

print(f"Generated in {time.time() - start_time} seconds.")
result
```

```
Generated in 0.13912627792358398 seconds.
```

Получили около 16% уменьшение время инференса

Запуск с TensorParallelizm

```
import os
import torch
import transformers
import deepspeed
local_rank = 0
world_size = -1
# create the model pipeline
pipe = transformers.pipeline(task="text2text-generation", model="google/t5-v1_1-small", device=local_rank)

pipe.model = deepspeed.init_inference(
    pipe.model,
    mp_size=world_size,
    dtype=torch.float
)
```

Время инференса

```
import time

start_time = time.time()
result = pipe(('Input String'))

print(f"Generated in {time.time() - start_time} seconds.")
result
```

```
Generated in 0.1156313419342041 seconds.
```

Results

Были изучены и воспроизведены на практике следующие методы ускорения инференса нейронных сетей:

- Flash-attention
- Flash-attention-2
- kv-caching
- Tensor-parallelism

Были выбраны два наиболее эффективных метода - это **flash-attention-2** и **kv-caching**, именно они были применены в итоговом приложении

Для демонстрации работы методов была выбрана нейронная сеть ChatGLM2-6B и было написано приложение на Flask.

GitHub: https://github.com/ksana-kozlova/LLM_inference