

Подходы к оптимизации LLM

Максим Желнин

Оптимизация инференса LLM
Оптимизация обучения LLM
Квантизация
Mixture of Experts

KV-Cache
Paged Attention
Flash Attention

Gradient Checkpointing
Gradient Accumulation
Mixed Precision
Model/Data Parallelism
DeepSpeed/FSDP

GPTQ
LLM.int8()
QLoRA

Sparse Mixture of
Experts / Mixtral

>>> Оптимизация инференса
LLM

KV-Cache

Для входной последовательности токенов $(x_1, \dots, x_n) \in \mathbb{R}^{n \times d}$ трансформер предсказывает вероятность следующего токена авторегрессионным способом, т.е. в зависимости от предыдущих токенов:

$$P(x) = P(x_1) \cdot P(x_2 | x_1) \cdots P(x_n | x_1, \dots, x_{n-1})$$

Таким образом в процессе генерации на основе промпта (входной последовательности токенов) (x_1, \dots, x_n) языковая модель последовательно, токен за токеном, генерирует ответ $(x_{n+1}, \dots, x_{n+T})$

Сначала вычисляются линейные проекции q_i, k_i, v_i , затем attention score a_{ij} путем умножения q_i на все предыдущие k_j , и наконец выход o_i как взвешенная сумма a_{ij} на все предыдущие v_j

$$q_i = W_q x_i, \quad k_i = W_k x_i, \quad v_i = W_v x_i \quad a_{ij} = \frac{\exp(q_i^\top k_j / \sqrt{d})}{\sum_{t=1}^i \exp(q_i^\top k_t / \sqrt{d})}, \quad o_i = \sum_{j=1}^i a_{ij} v_j$$

KV - cache - память, требуемая для хранения *key* и *value* для всей (входящей и сгенерированной) последовательности токенов

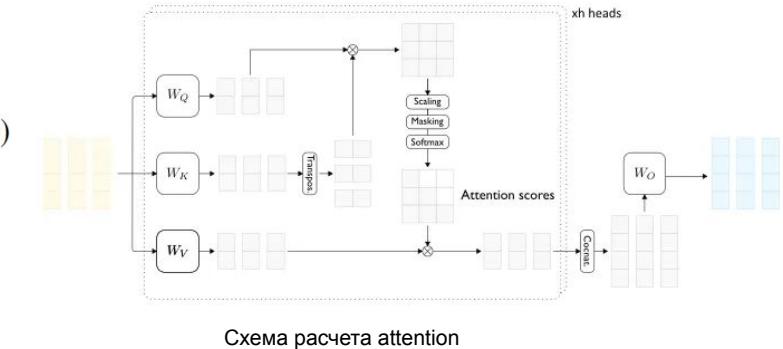
Генерации текста делится на два этапа:

The prompt phase - генерация первого токена $P(x_{n+1} | x_1, \dots, x_n)$ по входной последовательности токенов (x_1, \dots, x_n)

The autoregressive generation phase - авторегрессионное генерация следующих токенов. Данный этап заканчивается, когда достигнута максимальная длина последовательности или возникнет *<eos>* токен. При этом *key*, *value* кэшируются $k_1, \dots, k_{n+1}, v_1, \dots, v_{n+1}$

Расчеты могут быть проведены параллельно поскольку все входные токены известны.

Поскольку возникает зависимость от предыдущего токена, распараллеливание на этапе генерации затруднено, а также возникает большое количество операций умножения вектора на матрицу.



vLLM: Paged Attention

Paged Attention разбивает KV - cache для каждой последовательности токенов на KV - блоки.
Каждый блок хранит key, value для фиксированного числа токенов B (KV-block size).

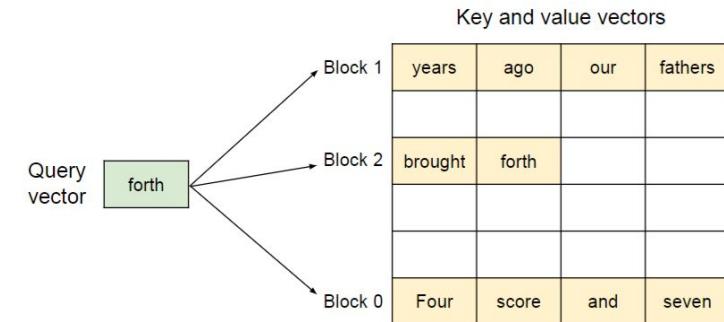
Выход o_i вычисляется отдельно для каждого j -ого KV блока

$$K_j = (k_{(j-1)B+1}, \dots, k_{jB}) \quad V_j = (v_{(j-1)B+1}, \dots, v_{jB})$$

$$A_{ij} = \frac{\exp(q_i^\top K_j / \sqrt{d})}{\sum_{t=1}^{\lceil i/B \rceil} \exp(q_i^\top K_t / \sqrt{d})}, \quad o_i = \sum_{j=1}^{\lceil i/B \rceil} V_j A_{ij}^\top,$$

где $A_{ij} = (a_{i,(j-1)B+1}, \dots, a_{i,jB})$ - attention score для j -ого KV блока

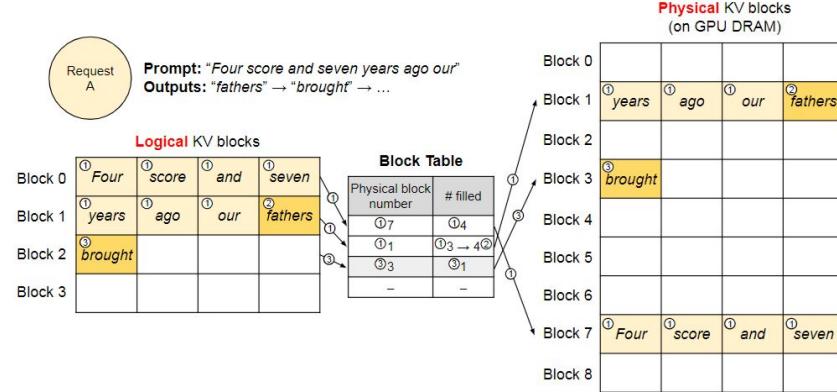
Благодаря такому подходу, Key и Value могут храниться в не последовательных (non-contiguous) ячейках памяти, что повышает эффективность использования памяти.



Key и Value хранятся в различных блоках, которые не идут непрерывно друг за другом. В каждый момент времени Query токен 'forth' умножается на блок Key токенов 'Four score and seven', чтобы вычислить attention score A_{ij} , а затем умножив строку A_{ij} на блок Value токенов получить выход o_j .

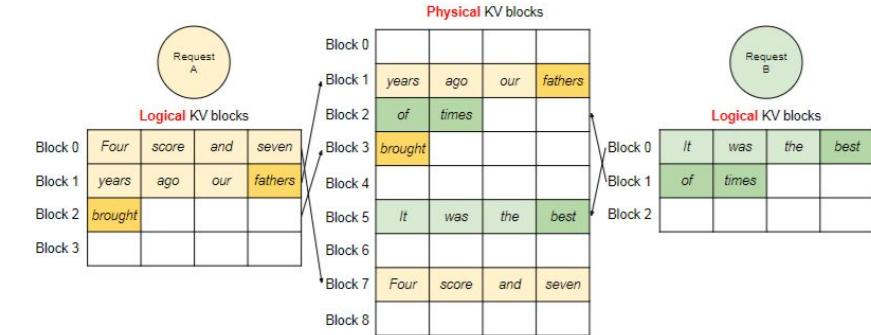
Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., ... & Stoica, I. (2023, October). Efficient memory management for large language model serving with paged attention. In Proceedings of the 29th Symposium on Operating Systems Principles (pp. 611-626).

<https://training.continuumlabs.ai/inference/why-is-inference-important/paged-attention-and-vllm>
<https://blog.vllm.ai/2023/06/20/vllm.html>

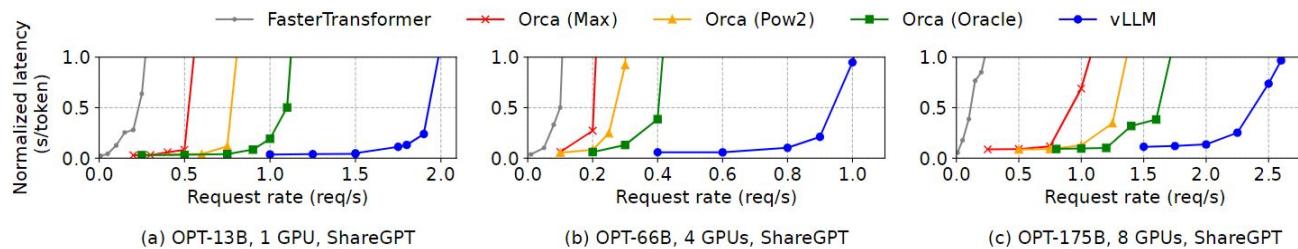


Управление KV - блоками в памяти выполняется посредством *Block Table*, в которой записано соответствие между логическим порядком следования KV-блоков и блоками в памяти GPU, где они хранятся.

При поступлении запроса к LLM резервируются блоки в памяти для хранения будущих генерированных данных и соответствующих им Key, Value. Затем входные и генерированные токены объединяются в одну последовательность и передаются в LLM. Расчет attention для блоков выполняется параллельно.



Поблочное хранение данных позволяет одновременно выполнять расчеты по нескольким запросам.



vLLM показала наименьшее
увеличение задержки ответа от скорости
запросов по сравнению с
существующими на тот момент
решениями

DeepSpeed: FastGen

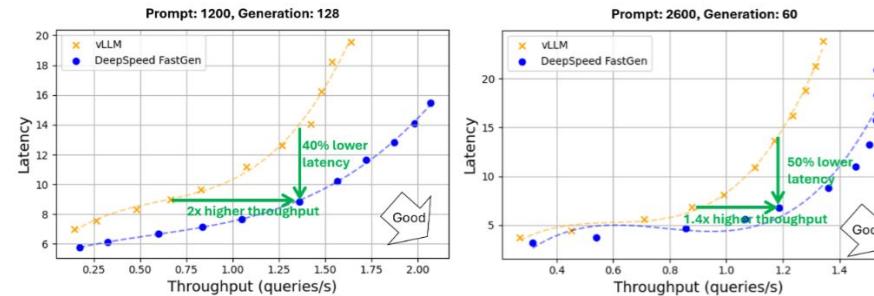
Dynamic SplitFuse

1. Длинные промпты (входные последовательности) разбиваются на части, каждая из которых обрабатывается параллельно. Вычисления выполняются с максимальным батчем входных данных, а веса линейных слоев, загруженные для расчетов с входными токенами переиспользуются для генерации.

2. Короткие промпты, полученные от нескольких пользователей, группируются в батчи одинакового размера, что позволяет оптимизировать вычисления при генерации.

Batching Scheme	Operation(s)		Total Time	Per-token Time	
	Linear	Attn		Prefill	Decode
Prefill-only	224.8	10	234.8	0.229	-
Decode-only	44.28	5.68	49.96	-	12.49
Decode-maximal	223.2	15.2	238.4	0.229	1.2

Объединение генерации с обработкой входных токенов позволяет оптимизировать процесс расчета forward pass, поскольку большинство времени уходит на выполнение линейных преобразований.



При большом количестве запросов такой подход показывает лучшую эффективность по сравнению с Paged Attention

>>> Оптимизация обучения LLM

Flash Attention

Иерархия памяти GPU:

SRAM (Статическая память с произвольным доступом) - быстрая кэш память маленького размера.
HBM (Высокопропускная память) - относительно медленная память большого размера.

Flash Attention - метод для ускорения расчета Attention в моделях трансформерах за счет уменьшения обмена данными между HBM и SRAM путем проведения большинства расчетов только с использованием SRAM. При этом вычисление Attention выполняется точно.

Стандартный способ вычисления Attention:

$Q, K, V \in \mathbb{R}^{N \times d}$ N - длина входной последовательности, d - количество голов

$S = QK^T \in \mathbb{R}^{N \times N}$, $P = \text{softmax}(S) \in \mathbb{R}^{N \times N}$, $O = PV \in \mathbb{R}^{N \times d}$,

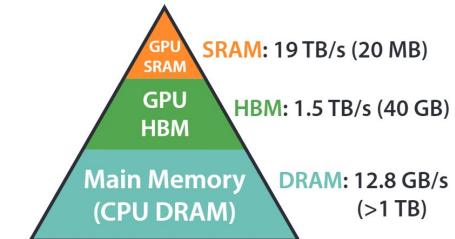
где функция softmax применяется поэлементно к S , деленному на корень из количества голов d .

Алгоритм стандартного способа:

На входе: Q, K, V в HBM

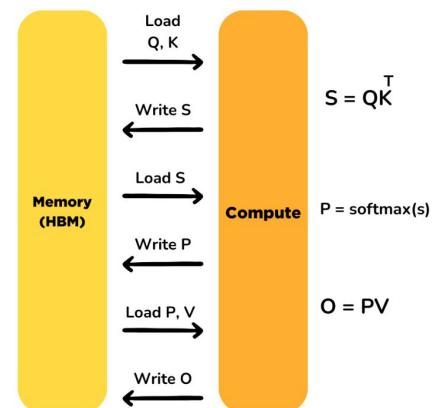
1. Загружаем Q, K поблочно из HBM в SRAM. Вычисляем матрицу S и записываем ее в HBM.
2. Загружаем S из HBM, вычисляем P и записываем ее в HBM.
3. Загружаем поблочно P, V из HBM, вычисляем O , записываем ее в HBM.

Основное время при вычислении Attention уходит не на вычисления, а на обмен данными, возникающих в ходе вычислений, между HBM и SRAM.



Иерархия памяти, используемая для работы и обучения моделей

Standard Attention Implementation



В стандартном способе: $O(N^2)$ обращений к памяти

Dao, T., Fu, D., Ermon, S., Rudra, A., & Ré, C. (2022). Flash Attention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35, 16344-16359.

Tiling - вычисление Attention по блокам с выполнением всех операций только в SRAM.

В стандартном подходе значения элементов в строке P_i матрицы \mathbf{P} вычисляются по формуле
Таким образом, чтобы вычислить softmax, необходимо сначала вычислить матрицу \mathbf{S}

$$P_i = \text{softmax}(S_i) = \frac{1}{\sqrt{d}} \frac{e^{S_i}}{\sum_k e^{S_k}}$$

Побочное вычисление Attention для вектора $x \in \mathbb{R}^B$

1) Вычисление softmax с нормирующим множителем $e^{-m(x)}$, чтобы предотвратить появление больших значений при взятии экспоненты, которые могут вызвать переполнение числового типа данных

$$m(x) := \max_i x_i, \quad f(x) := [e^{x_1-m(x)} \quad \dots \quad e^{x_B-m(x)}], \quad \ell(x) := \sum_i f(x)_i, \quad \text{softmax}(x) := \frac{f(x)}{\ell(x)}.$$

2) Объединение Attention, вычисленных для двух векторов $x = [x^{(1)} \ x^{(2)}] \in \mathbb{R}^{2B}$, где $x^{(1)}, x^{(2)} \in \mathbb{R}^B$

$$m(x) = m([x^{(1)} \ x^{(2)}]) = \max(m(x^{(1)}), m(x^{(2)})), \quad f(x) = [e^{m(x^{(1)})-m(x)} f(x^{(1)}) \quad e^{m(x^{(2)})-m(x)} f(x^{(2)})],$$

$$\ell(x) = \ell([x^{(1)} \ x^{(2)}]) = e^{m(x^{(1)})-m(x)} \ell(x^{(1)}) + e^{m(x^{(2)})-m(x)} \ell(x^{(2)}), \quad \text{softmax}(x) = \frac{f(x)}{\ell(x)}.$$

$\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ N - длина входной последовательности, d - количество голов

Алгоритм Flash Attention.

На входе: $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ в HBM

1. Задаем размеры будущих блоков, которые поместятся в чип SRAM памяти M:

$$B_c = \lceil \frac{M}{4d} \rceil, B_r = \min(\lceil \frac{M}{4d} \rceil, d)$$

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

3. Разбиваем \mathbf{Q} на T_r блоков размером $B_r \times d$; разбиваем \mathbf{K}, \mathbf{V} на T_c блоков размером $B_c \times d$.

4. Разбиваем \mathbf{O} на T_r блоков размером $B_r \times d$; разбиваем ℓ, m на T_c блоков размером $B_c \times d$.

5. **for** $1 \leq j \leq T_c$ **do**

6. Считываем $\mathbf{K}_j, \mathbf{V}_j$ из HBM на чип SRAM.

7. **for** $1 \leq i \leq T_r$ **do**

8. Считываем $\mathbf{Q}_i, \mathbf{O}_i, l_i, m_i$ из HBM в чип SRAM.

9. На чипе вычисляем скалярное произведение: $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$

10. На чипе вычисляем статистики:

$$\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}, \tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}, \tilde{l}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$$

11. Обновляем глобальные статистики для матриц:

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

12. Записываем attention в HBM:

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

13. Записываем статистики в HBM:

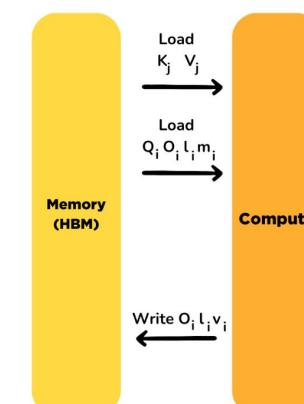
$$\ell_i \leftarrow \ell_i^{\text{new}}, m_i \leftarrow m_i^{\text{new}}$$

14. **end for**

15. **end for**

16. **return** \mathbf{O} .

Flash Attention



Kernel operations fused together, reducing reads & writes

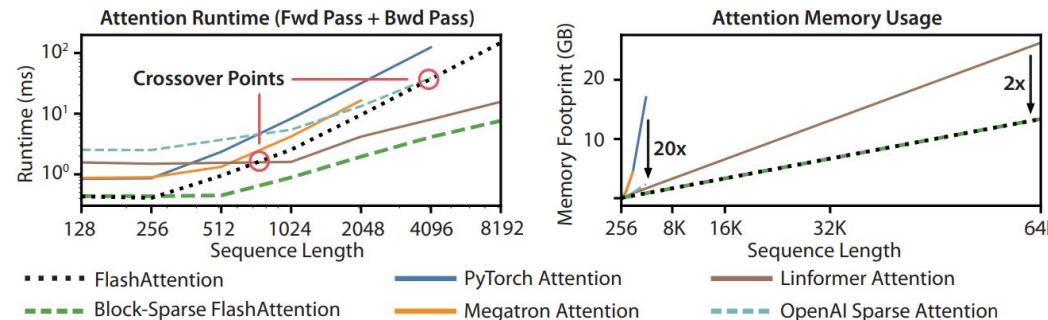
$$\begin{aligned} S_{ij} &= \mathbf{Q}_i \mathbf{K}_j^T \\ m &= \text{rowmax of } S \\ P &= \exp(s - m) \\ L &= \text{rowsum of } P \\ m &= \max(\tilde{m}_{ij}, m) \\ \text{calculate } O &\text{ from } L \& m \end{aligned}$$

Initialize O, l and m matrices with zeroes. m and l are used to calculate cumulative softmax. Divide Q, K, V into blocks (due to SRAM's memory limits) and iterate over them, for i is row & j is column.

Flash Attention: $O(N)$ обращений к памяти

Flash Attention имеет следующие преимущества:

1. *Tiling*. Вычисление Attention по блокам с использованием только SRAM.
2. *Recomputation*. Для вычисления градиентов по \mathbf{Q} , \mathbf{K} , \mathbf{V} требуется хранить в памяти матрицы \mathbf{S} , \mathbf{P} . Для того чтобы уменьшить размер памяти можно хранить только матрицу \mathbf{O} и некоторые нормирующие статистики, что на backward pass позволит восстановить матрицы \mathbf{S} , \mathbf{P} из матриц \mathbf{Q} , \mathbf{K} , \mathbf{V} путем выполнения вычислений только в памяти SRAM. В результате количество вычислений увеличивается, но благодаря тому, что все вычисления выполняются на SRAM, это не приводит к значимому увеличению времени, необходимого для backward pass.
3. *Kernel Fusion*. Вычисление Attention поблочно позволяет выполнять все операции с матрицами (умножение, softmax, dropout) с использованием только одного CUDA kernel. Как результат удается исключить медленные операции считывание и записи матриц в HBM.



По сравнению с стандартным подходом и другими методами к расчету Attention, метод Flash Attention оказывается быстрее и экономнее по памяти.

Flash Attention 2

Улучшения FlashAttention2:

- Уменьшение количества не матричных операций. Для этого деление на $I^{(new)}$ выполняется не при каждом обновлении \mathbf{O} для нового блока, а после прохода всех блоков T_r :

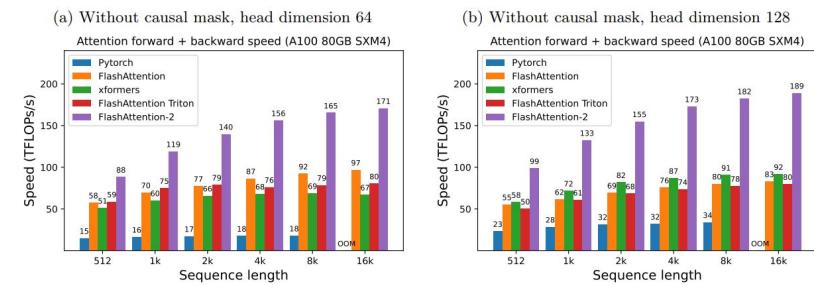
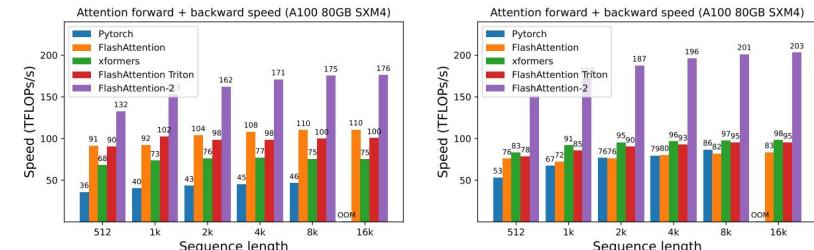
$$\begin{aligned}\mathbf{O}^{(2)} &= \text{diag}(\ell^{(1)}/\ell^{(2)})^{-1} \mathbf{O}^{(1)} + \text{diag}(\ell^{(2)})^{-1} e^{\mathbf{S}^{(2)} - \mathbf{m}^{(2)}} \mathbf{V}^{(2)} \\ \tilde{\mathbf{O}}^{(2)} &= \text{diag}(\ell^{(1)})^{-1} \mathbf{O}^{(1)} + e^{\mathbf{S}^{(2)} - \mathbf{m}^{(2)}} \mathbf{V}^{(2)} \\ \mathbf{O}_i &= \text{diag}(l^{T_c})^{-1} \tilde{\mathbf{O}}^{T_c}\end{aligned}$$

- Оптимизация памяти путем иного расчета backward pass, в котором вместо величин m, l используется только одна величина

$$L_i = m_i^{(T_c)} + \log(\ell_i^{(T_c)})$$

- Оптимизация расхода памяти GPU для длинных входных последовательностей на forward backward pass путем распараллеливания по длине последовательности.

- Уменьшение количества чтения и записи памяти при операциях с блоками при вычислении Attention



Flash Attention 2 показывает прирост скорости на forward и backward pass по сравнению с FlashAttention

Gradient Checkpointing

Backpropagation требует хранения всех промежуточных активаций для расчета градиентов. Это приводит к расходу $O(n)$ дополнительной памяти, где n глубина сети. Для того чтобы уменьшить объем памяти, необходимый для обучения модели, можно сохранить только часть активаций, а остальные активации пересчитать на этапе backward pass.

Gradient Checkpointing - метод для уменьшения объема памяти, требуемого для обучения модели, за счет того, что на этапе forward pass сохраняется только часть активаций, а недостающие активации пересчитываются во время backward pass.

Рассмотрим простой пример: трехслойная сеть $n = 3$

Forward pass: $\mathbf{Y} = f^3(\mathbf{W}^3 f^2(\mathbf{W}^2 f^1(\mathbf{W}^1 \mathbf{X})))$ Activations: $\mathbf{A}^0 = \mathbf{X}$, $\mathbf{A}^i = f^i(\mathbf{S}^i)$, $\mathbf{S}^i = \mathbf{W}^i \mathbf{A}^{i-1}$
 Loss: $L = l(\mathbf{Y}, \mathbf{Y}_{target})$

Стандартная схема расчета backward pass:

Входные данные: активации \mathbf{A}^i , вычисленные на Forward Pass

Последовательно от последнего слоя к первому:

1) Вычисляем производную от Loss функции по весам \mathbf{W}^i -го слоя

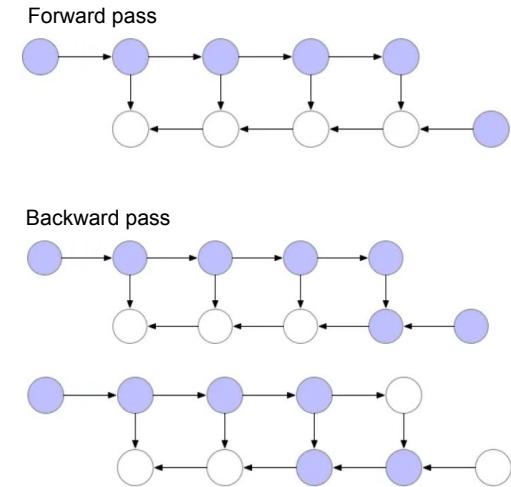
2) Обновляем веса \mathbf{W}^i , исходя из используемого оптимизатора

3) Удаляем активации \mathbf{A}^i и производные предыдущего слоя из памяти.

$$\frac{\partial L}{\partial W_{ij}^3} = \frac{\partial l}{\partial A_k^3} \frac{\partial f_k^3}{\partial S_m^3} \frac{\partial S_m^3}{\partial W_{ij}^3} = \frac{\partial l}{\partial A_k^3} \frac{\partial f_k^3}{\partial S_i^3} A_j^2 = B_i^3 A_j^3$$

$$\frac{\partial L}{\partial W_{ij}^2} = \frac{\partial l}{\partial A_k^3} \frac{\partial f_k^3}{\partial S_m^3} \frac{\partial S_m^3}{\partial A_n^2} \frac{\partial f_n^2}{\partial S_p^2} \frac{\partial S_p^2}{\partial W_{ij}^2} = B_m^3 W_{mn} \frac{\partial f_n^2}{\partial S_i^2} A_j^1 = B_m^3 B_{mi}^2 A_j^1$$

$$\frac{\partial L}{\partial W_{ij}^1} = B_m^3 W_{mn} \frac{\partial f_n^2}{\partial S_l^2} \frac{\partial S_l^2}{\partial A_k^2} \frac{\partial f_k^1}{\partial S_p^1} \frac{\partial S_p^1}{\partial W_{ij}^1} = B_m^3 B_{ml}^2 W_{lk} \frac{\partial f_k^1}{\partial S_i^1} A_j^0 = B_m^3 B_{ml}^2 B_{li}^1 X_j$$



*Иллюстрации приведены для четырехслойной сети $n = 4$

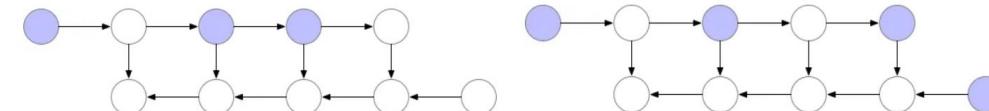
Расчета backward pass с использованием Gradient Checkpointing:

Входные данные: $\text{sqrt}(n)$ активации A^i , вычисленных на Forward Pass

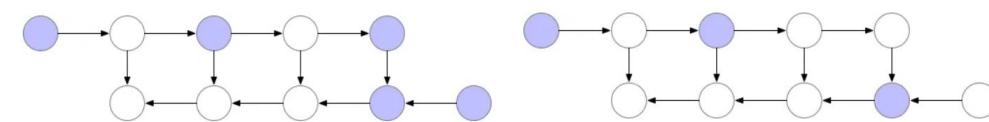
- 1) Вычисляем производную от Loss функции по весам W^i i -ого слоя. При этом если активации A^i нет в памяти, то вычисляем ее
- 2) Обновляем веса W^i , исходя из используемого оптимизатора
- 3) Удаляем активации A^i и производные предыдущего слоя из памяти.

Forward pass

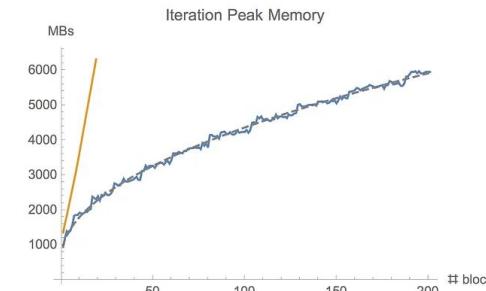
Вычисляются все активации, но в памяти хранится только часть активаций



Backward pass



Недостающие активации вычисляются заново, а затем удаляются



При использовании Gradient Checkpointing требуется $\text{sqrt}(n)$ памяти для обучения сети, где n - глубина сети.

Memory Requirement $O(n)$
 Compute Requirement $O(n)$
 Forward calcs per node 1

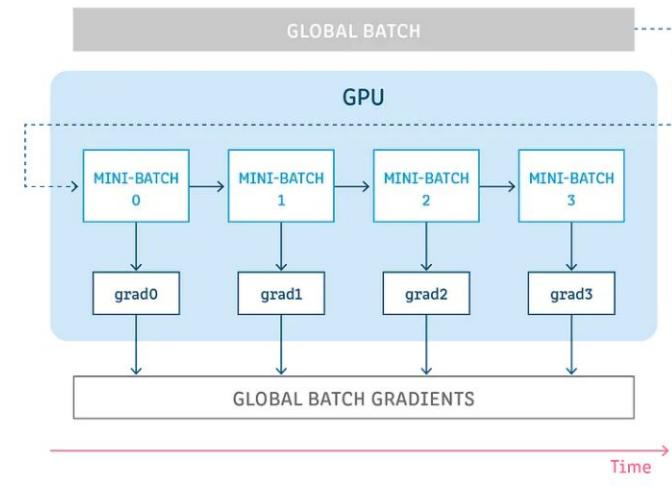
Memory Requirement $O(\sqrt{n})$
 Compute Requirement $O(n)$
 Forward calcs per node 1 to 2

Gradient Accumulation

Gradient Accumulation - метод накопления градиентов для весов моделей по различным объектам в течение нескольких итераций, что позволяет обучать модели с батчем большого размера без существенного увеличения требований к памяти GPU.

Обычно в ходе backpropagation градиенты модели вычисляются для каждого батча. Однако, в случае больших моделей и батчей большого размера - это приводит к требованию больших ресурсов памяти GPU.

Для решения этой проблемы был предложен метод **Gradient Accumulation**, в котором большой батч накапливается путем расчета и сохранения градиентов для малых батчей составленных из различных объектов из обучающего датасета. Как только накопленное количество объектов, для которых вычислены градиенты по малым батчам, удовлетворит размеру большого батча, выполняется шаг оптимизатора и обновление весом модели.



Gradient Accumulation позволяет проводить обучение моделей с батчами большого размера на GPU с малой памяти, но приводит к увеличению времени обучения.

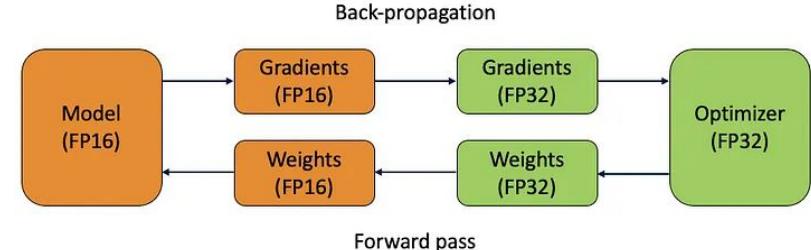
Mixed Precision Training

Mixed Precision Training - это метод обучения, при котором forward pass и расчет градиентов выполняется в fp16/bf16, а параметры оптимизатора, приращение и обновление весов делаются в fp32.

Алгоритм Mixed Precision Training

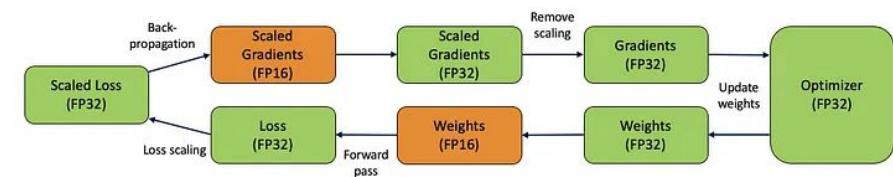
На входе имеется модель в fp16/bf16

1. Для батча обучающих данных вычисляется forward pass в fp16/bf16. Вычисленные активации для каждого слоя хранятся в fp16/bf16.
2. Расчет Loss функции. Если вычисления выполнялись в fp16, то Loss умножается на масштабный коэффициент S
3. На этапе backward pass вычисляются градиенты для весов каждого слоя в fp16/bf16. Если градиенты вычислялись в fp16, то градиенты умножаются на коэффициент $1/S$
4. Расчет параметров оптимизатора в fp32
5. Обновление весов модели в fp32
6. Конвертация fp32 весов в fp16 для следующей итерации. При этом fp32 веса сохраняются для расчета следующих обновленных весов.



Использование half-precision (fp16 / bf16) для обучения модели требует хранение fp32 копий весов.

Однако дополнительные затраты памяти невелики из-за возможности взять больший батч (активации занимают меньше памяти), а также более быстрого расчета forward pass и backward pass.



Вычисление градиентов в fp16 формате, может привести к тому, что большинство градиентов будет равно 0. Для решения этой проблемы Loss, вычисленный для объектов батча, умножается на масштабный коэффициент S . Вследствие этого значение градиентов увеличивается (уменьшается количество знаков после запятой) и backward pass можно выполнить более точно. Перед обновлением весов градиенты умножаются на $1/S$.

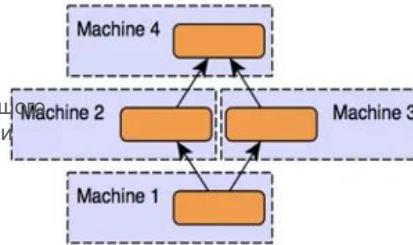
Проведение расчетов в формате bf16 не требует использование масштабных коэффициентов.

Стандартные подходы к распределенному обучению

Model Parallelism - параллельное обучение модели, которое предполагает вертикальное размещение слоев модели на отдельных GPU. Расчет forward pass и backward pass выполняется последовательно для каждой части модели, размещенной на одной GPU, с передачей полученного результата в следующую часть модели.

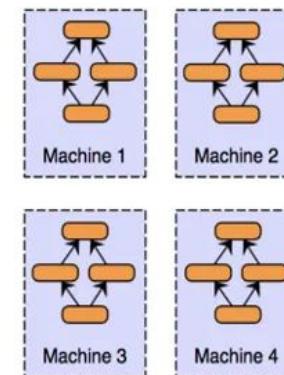
Model Parallelism

Model Parallelism имеет хорошую эффективность по памяти, но требует большого объема обмена данными между различными GPU.



Data Parallelism - параллельное обучение модели, при котором она полностью загружается на каждую GPU, а входящие данные распределяются между GPU, исходя из оставшейся памяти. После выполнения forward и backward pass, полученные градиенты для каждой копии модели усредняются и используются для обновления весов каждой копии модели.

Data Parallelism



Data Parallelism требует малого обмена данными между GPU, но имеет плохую эффективность по памяти.

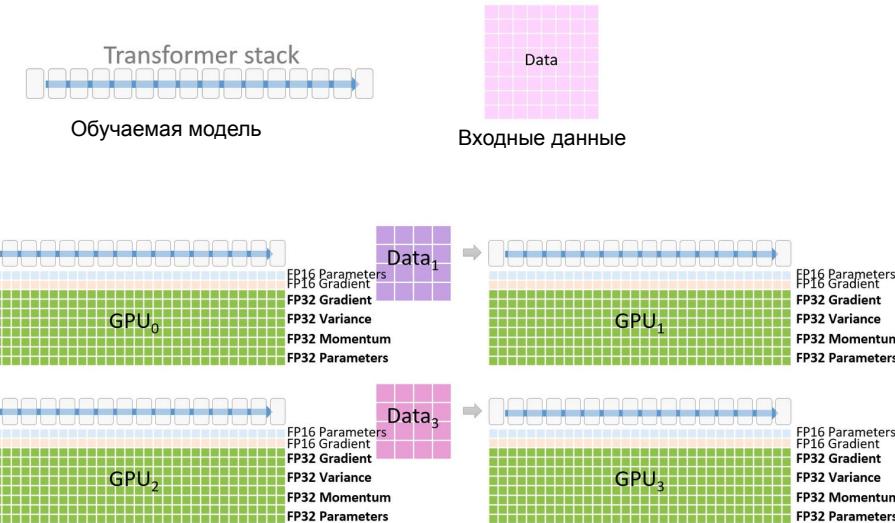
DeepSpeed - ZeRO

DeepSpeed ZeRO - библиотека для параллельного обучения LLM, основанная на Zero Redundancy Optimizer (ZeRO) подходе, который заключается в оптимизации расхода памяти путем эффективного распределения параметров оптимизатора, градиентов и весов модели между имеющимися GPU.

В ходе обучения модели расходуемая память может быть разделена на **Memory State Memory** (большая часть) и **Residual State Memory** (меньшая часть).

Model State Memory - память, которая расходуется на хранение параметров оптимизатора (momentum, variance для Adam), значений весов слоев и их градиентов.

Residual State Memory - память, которая используется для хранения активаций, промежуточных переменных, а также фрагментированные блоки памяти, которые не могут быть использованы для хранения данных.



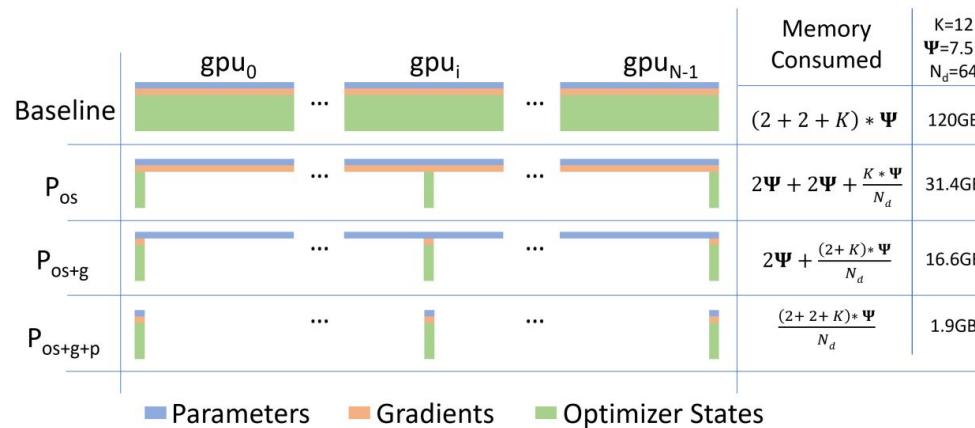
Rajbhandari, S., Rasley, J., Ruwase, O., & He, Y. (2020, November). Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis* (pp. 1-16). IEEE.
<https://www.cs.cmu.edu/~zhihaoj2/15-849/slides/13-zero-redundancy.pdf>

ZeRO - DP: Optimizing Model State Memory

- 1) Разделенное хранение состояний оптимизатора (P_{os}) - 4x уменьшение памяти
- 2) Разделенное хранение градиентов (P_{os+g}) - 8x уменьшение памяти
- 3) Разделенное хранение параметров модели (P_{os+g+p}) - линейно относительно степени разбиения N_d . Разделение на 64 GPU ($N_d = 64$) дает уменьшение в 64x уменьшение памяти.

ZeRO - R: Optimizing Residual State Memory

- 1) Разделенное хранение активаций. При необходимости хранение активаций на RAM
- 2) Подбор размера памяти для хранение промежуточных переменных, чтобы достичь баланса между объемом памяти и вычислительной эффективностью.
- 3) ZERO-R предотвращает фрагментацию памяти посредством последовательного заполнения блоков памяти.



K - мультипликатор объема памяти, требующегося для хранения параметров оптимизатора. K = 12, соответствует Adam
 Ψ - количество параметров модели
 N_d - степень распараллеливания

Оптимизация расхода Model State Memory при обучении с применением ZeRO
(Baseline - Data Parallelism)

ZeRO - DP Optimizing Model State Memory

ZeRO Stage 1: Optimizer State Partitioning P_{os}

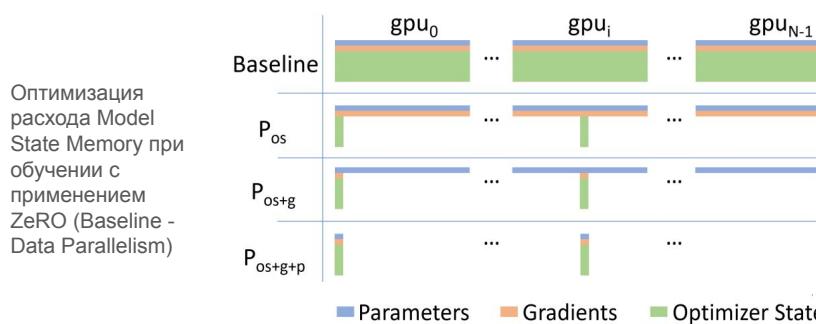
Для распараллеливания обучения на N_d GPU параметры состояния оптимизатора разделяются на N_d равных частей так, что i -ое GPU отвечает только за обновление i -ой части параметров. Вследствие этого на одной GPU необходимо хранить и обновлять только $1/N_d$ обновленных параметров модели и оптимизатора.

ZeRO Stage 2: Gradient Partitioning P_g

Поскольку для обновления части параметров, хранящихся на одной GPU, требуются только градиенты, соответствующие этим параметрам, то хранение градиентов также может быть распределено между GPU. Такой подход приводит к уменьшению памяти с 2Ψ до $2\Psi / N_d$.

ZeRO Stage 3: Parameter Partitioning P_p

Распределенное хранение в памяти параметров состояния оптимизатора и градиентов позволяет также разделить между GPU веса слоев, соответствующие этим данным. Когда для расчета forward pass или backward pass требуются параметры из соседнего слоя, они берутся из соответствующего параллельного процесса. Благодаря правильно организованному процессу обмена данных, это приводит увеличению количества операций обмена данных только в 1,5 раз.



ZeRO - R Optimizing Residual State Memory

Partitioned Activation Checkpointing: P_a

На каждой GPU хранятся только активации, соответствующие параметрам модели, находящейся на этой GPU. В дополнение к этому активации хранятся только для некоторых шагов forward pass по аналогии с Gradient Checkpointing. В случае нехватки памяти, контрольные точки для активаций модели могут быть выгружены в DRAM.

Constant Size Buffers: C_B

Для хранения временных и промежуточных переменных на всех устройствах резервируется объем памяти постоянного размера.

Memory Defragmentation M_D

Оптимизация памяти для выполнения forward pass и backward pass операций приводит к возникновению тензоров с различным временем существования. В случае forward pass одна часть активаций сохраняется длительное время в памяти для вычисления, а другая быстро удаляется.

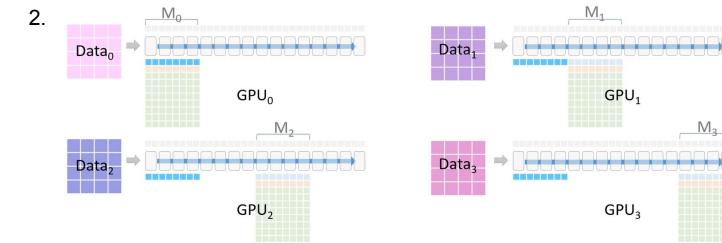
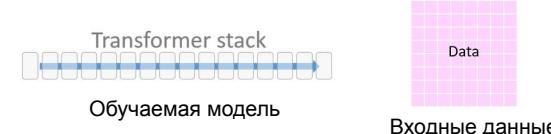
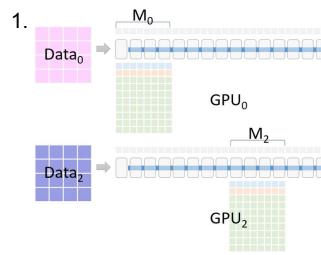
В случае backward pass сохраняются градиенты по весам слоев модели, а градиенты по активациям и другие промежуточные переменные быстро удаляются.

Использование тензоров с различным временем существования приводит к фрагментации памяти.

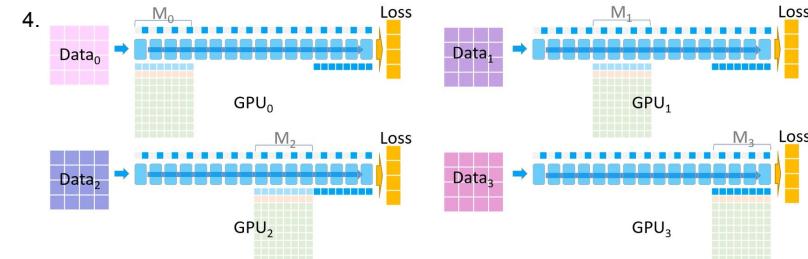
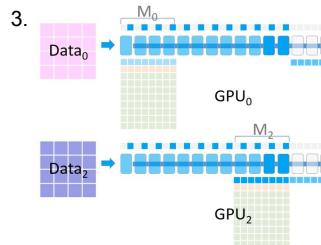
При обучении больших моделей фрагментация памяти приводит к неблагоприятным последствиям:

- 1) OOM (переполнение памяти) вследствие отсутствия достаточного количества непрерывно расположенных блоков;
- 2) снижение эффективности, поскольку требуется значительное количество времени для поиска непрерывно расположенных блоков.

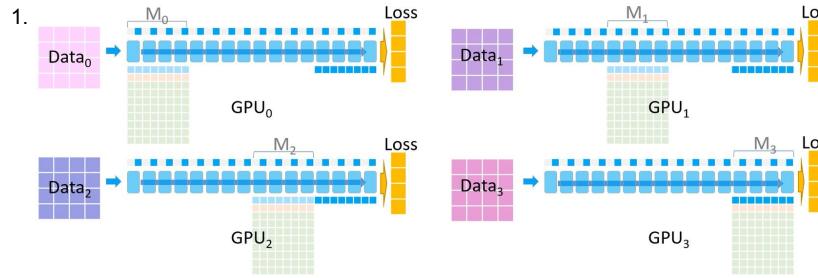
Для решения этой проблемы ZeRO выполняет дефрагментацию памяти on-the-fly, сохраняя активации и градиенты в соседние фрагменты памяти.

Расчет Forward Pass P_{os+g+p} (ZeRO 1+2+3 stages)

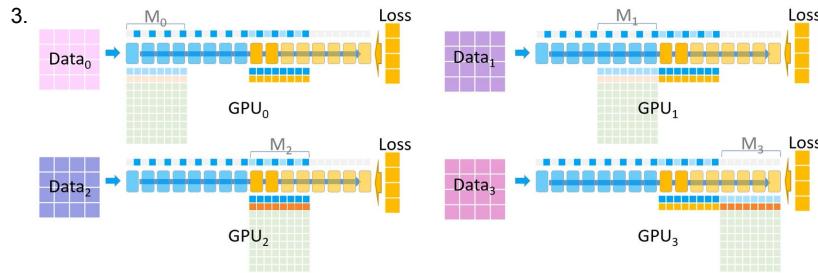
Вычисление forward pass для части параметров модели M_0 . Расчет выполняется для каждой части данных $Data_i$ на всех GPU. Для этого параметры M_0 передаются с GPU₀ на остальные GPU. После выполнения расчета параметры M_0 удаляются с GPU_{1,2,3}. Вычисленные активации сохраняются на каждой GPU



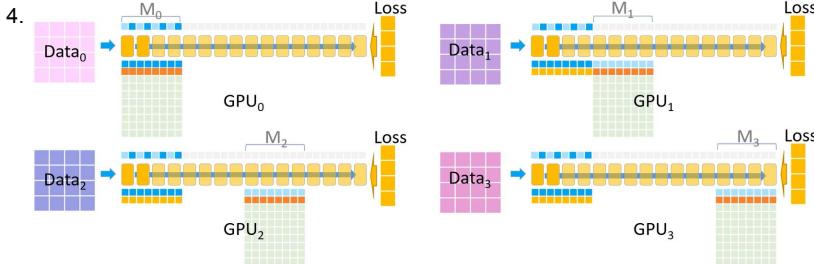
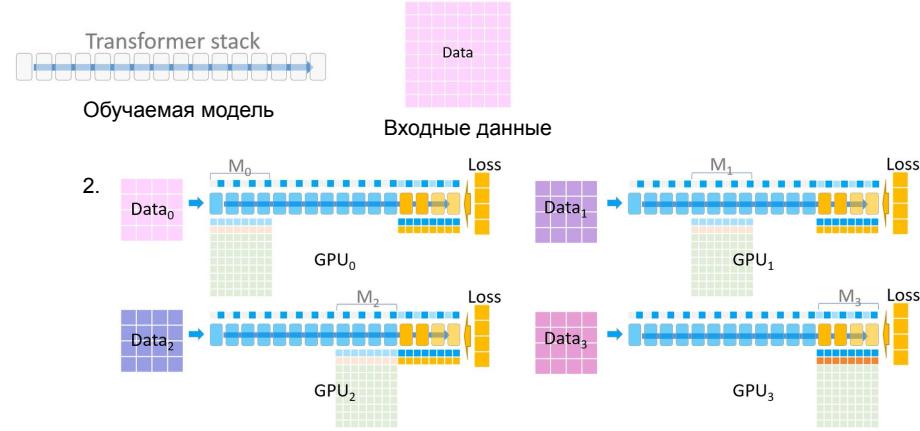
Аналогично расчет forward pass последовательно выполняется для параметров $M_{1,2,3}$. При этом следуя Gradient Checkpointing, на каждой GPU сохраняется только часть активаций, соответствующая параметрам M_i

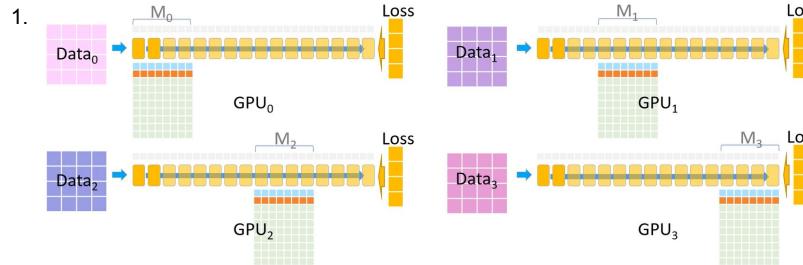
Расчет Backward Pass P_{os+g+p} (ZeRO 1+2+3 stages)

Каждой GPU_i соответствует Loss, вычисленный для Data_i, часть активаций, вычисленная для параметров M_i, и параметры M₃, полученные с GPU₃.

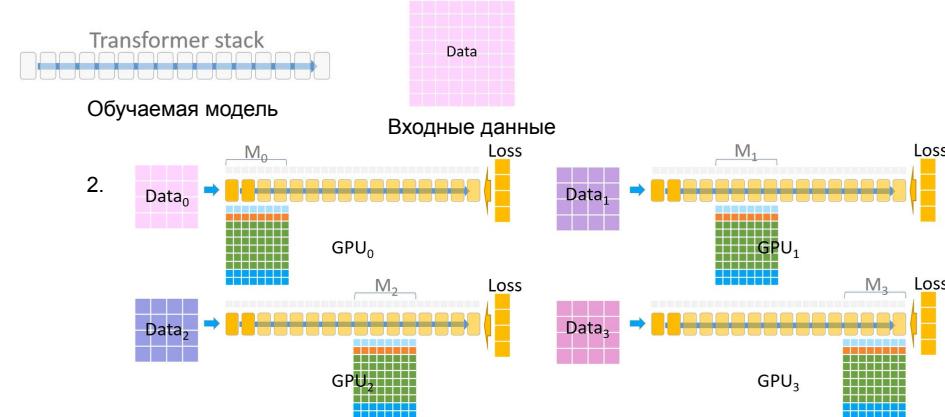


Для расчета градиентов по параметрам M₂ они передаются с GPU₂ на остальные GPU. На каждой GPU_i восстанавливаются пропущенные активации и находятся градиенты для данных Data_i. Найденные градиенты передаются на GPU₂, где они усредняются, чтобы найти градиенты для всех входных данных.

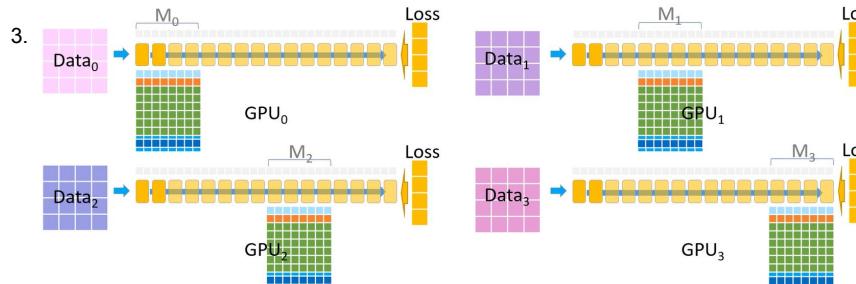


Расчет Backward Pass P_{os+g+p} (ZeRO 1+2+3 stages)

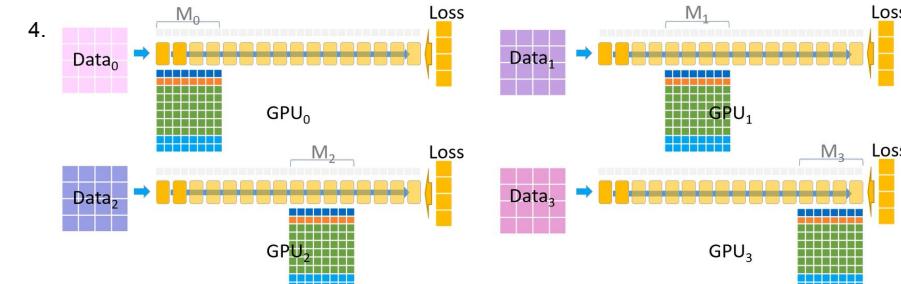
На каждой GPU_i хранятся градиенты, вычисленные по всем входным данным, для части параметров M_i .



На каждой GPU_i выполняются шаги оптимизатора в fp32 для расчета обновленных параметров M_i .



Обновленные параметры M_i конвертируются из fp32 в fp16.

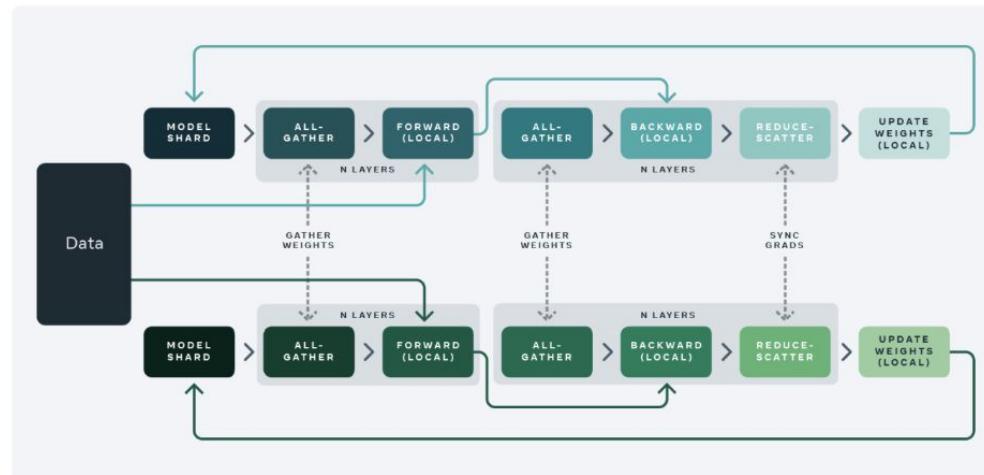


Исходные параметры заменяются на обновленные параметры M_i в fp16, которые затем используется на следующей итерации.

FSDP

Альтернативная реализация ZeRO подхода к распределенному обучению

Fully sharded data parallel training



All - Gather Weight - распределение весов модели, хранящихся на одной GPU, среди всех GPU, участвующих в ее обучении.

Reduce - Scatter Grads - накопление каждой GPU градиентов для хранящихся на ней весов модели.

Zhao, Y., Gu, A., Varma, R., Luo, L., Huang, C. C., Xu, M., ... & Li, S. (2023). Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*.

<https://engineering.fb.com/2021/07/15/open-source/fsdp/>

<https://openmmlab.medium.com/its-2023-is-pytorch-s-fsdp-the-best-choice-for-training-large-models-fe8d2848832f>

<https://docs.nvidia.com/deeplearning/ncc1/user-guide/docs/usage/collectives.html#allreduce>

>>> Квантизация LLM

OBS & GPTQ

OBS (Optimal Brain Surgeon) - метод оптимизации, основанный на аналитических формулах для расчета влияние весов модели на Loss функцию и восстановления ошибки, вызванной квантизацией (прунингом).

GPTQ - метод квантизации LLM, основанный на **OBS**.

После квантизации возникает ошибка, вызванная тем, что деквантизованные веса отличаются от оригинальных. Можно ли уменьшить ошибку квантизации путем калибровки весов в процессе квантизации без дообучения модели?

Решение данной задачи может быть получено с помощью метода Optimal Brain Surgeon, который предоставляет аналитические формулы, позволяющие оценить влияние весов на Loss функцию и последовательно перераспределять ошибку, вызванной квантизацией (прунингом) весов, на еще не квантованные (не запрунинные) веса.

Метод квантизации GPTQ - вычислительно эффективное применение метода Optimal Brain Surgeon к квантизации LLM

LeCun, Y., Denker, J., & Solla, S. (1989). Optimal brain damage. *Advances in neural information processing systems*, 2.

Hassibi, B., & Stork, D. (1992). Second order derivatives for network pruning: Optimal brain surgeon. *Advances in neural information processing systems*, 5.

Frantar, E., & Alistarh, D. (2022). Optimal brain compression: A framework for accurate post-training quantization and pruning. *Advances in Neural Information Processing Systems*, 35, 4475-4488.

Frantar, E., Ashkboos, S., Hoefler, T., & Alistarh, D. (2022). Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*.

Рассмотрим целевую функцию

$$E = E(f(\mathbf{W}, \mathbf{X}), \mathbf{Y}),$$

где f -- нейросетевая модель, \mathbf{W} - матрица весов, \mathbf{X} -- батч входных данных, \mathbf{Y} -- выходные данные.

Зададим приращение $\delta\mathbf{W}$ для весов \mathbf{W} модели f .

Тогда записывая матрицы $\mathbf{W}, \delta\mathbf{W}$ в виде векторов $\mathbf{w} = (W_{11}, \dots, W_{1d_{in}}, \dots, W_{d_{out}d_{in}})$, $\delta\mathbf{w} = (\delta W_{11}, \dots, \delta W_{1d_{in}}, \dots, \delta W_{d_{out}d_{in}})$, по формуле Тейлора получим следующее выражение:

$$\delta E = \left(\frac{\partial E}{\partial \mathbf{w}} \right)^T \cdot \delta\mathbf{w} + \frac{1}{2} \delta\mathbf{w}^T \cdot \left(\frac{\partial^2 E}{\partial \mathbf{w}^2} \right) \cdot \delta\mathbf{w} + O(\|\delta\mathbf{w}\|^3)$$

Если модель f была обучена и достигла локального минимума, то первая производная равна ноль:

$$\frac{\partial E}{\partial \mathbf{w}} = \mathbf{0}$$

Вследствие чего, пренебрегая слагаемыми третьего и более высокого порядка, получим:

$$\delta E \approx \frac{1}{2} \delta\mathbf{w}^T \cdot \mathbf{H} \cdot \delta\mathbf{w},$$

где $\mathbf{H} = \frac{\partial^2 E}{\partial \mathbf{w}^2}$ - матрица Гессе.

Найдем приращение $\delta\mathbf{w}$, при котором приращение целевой функции будет минимальной $\delta E \rightarrow \min$, а ошибка при квантизации веса w_q будет равна нулю:

$$\mathbf{e}_q^T \cdot \delta\mathbf{w} = \Delta, \quad \Delta = \text{quant}(w_q) - w_q,$$

где \mathbf{e}_q^T - базисный вектор для веса (координаты) w_q , Δ - ошибка квантизации.

Для решения поставленной задачи оптимизации с ограничением в виде равенства воспользуемся методом множителей Лагранжа.
Функция Лагранжа:

$$L(\delta\mathbf{w}, \lambda) = \frac{1}{2} \delta\mathbf{w}^T \cdot \mathbf{H} \cdot \delta\mathbf{w} + \lambda [\mathbf{e}_q^T \cdot \delta\mathbf{w} - \Delta],$$

где λ - множитель Лагранжа.

Учитывая, что матрица H симметрична, частные производные функции L имеют вид:

$$\frac{\partial L}{\partial \delta\mathbf{w}} = \mathbf{H} \cdot \delta\mathbf{w} + \lambda \mathbf{e}_q = 0, \quad \frac{\partial L}{\partial \lambda} = \mathbf{e}_q^T \cdot \delta\mathbf{w} - \Delta = 0$$

Решение системы уравнений имеет вид:

$$\delta\mathbf{w} = \frac{\Delta}{[\mathbf{H}^{-1}]_{qq}} \mathbf{H}^{-1} \cdot \mathbf{e}_q, \quad L = \frac{1}{2} \frac{\Delta^2}{[\mathbf{H}^{-1}]_{qq}}$$

Таким образом, после квантизации веса w_q , полученная ошибка квантизации Δ может быть скомпенсирована следующим приращением весов:

$$\mathbf{w}_{update} = \mathbf{w} + \delta\mathbf{w} = \mathbf{w} + \frac{(\text{quant}(w_q) - w_q)}{[\mathbf{H}^{-1}]_{qq}} \mathbf{H}_{:,q}^{-1},$$

где $\mathbf{H}_{:,q}^{-1}$ - столбец q матрицы \mathbf{H}^{-1} .

Применение данного алгоритма для LLM имеет два существенных ограничения:

- 1) Расчет матрицы \mathbf{H}^{-1} для всей LLM не может быть выполнен на современных GPU.
- 2) После квантизации отдельно взятого веса и обновления остальных весов модели требуется пересчет матрицы \mathbf{H}^{-1} .

Решение поставленных проблем:

- 1) Вводим отдельную целевую функцию E для каждого линейного слоя.
- 2) Выполняем квантизацию не отдельно взятого веса из матрицы весов \mathbf{W} , а всего столбца.

Запишем ошибку квантизации для каждого слоя относительно некоторого батча \mathbf{X} :

$$E = \|\mathbf{W} \cdot \mathbf{X} - quant(\mathbf{W}) \cdot \mathbf{X}\|_2^2,$$

Найдем приращение $\delta\mathbf{W}$, которое минимизирует приращение ошибки квантизации δE .

Запишем приращение δE .

$$\begin{aligned}\delta E &= 2([\mathbf{W} \cdot \mathbf{X} - quant(\mathbf{W}) \cdot \mathbf{X}] \cdot \mathbf{X}^T) : \delta\mathbf{W} + \delta\mathbf{W} \cdot \mathbf{X} \cdot \mathbf{X}^T : \delta\mathbf{W} = \\ &= 2\text{tr}(\mathbf{X} \cdot \mathbf{X}^T \cdot [\mathbf{W} - quant(\mathbf{W})]^T \cdot \delta\mathbf{W}) + \text{tr}(\mathbf{X} \cdot \mathbf{X}^T \cdot \delta\mathbf{W}^T \cdot \delta\mathbf{W})\end{aligned}$$

Условие того, что приращение $\delta\mathbf{W}$ минимизирует приращение ошибки квантизации δE ,

вызванное квантизацией столбца q записывается, как

$$\delta\mathbf{W} \cdot \mathbf{e}_q - \Delta = \mathbf{0},$$

где $\Delta = quant(\mathbf{W})_{:,q} - \mathbf{W}_{:,q}$ - ошибка квантизации.

Повторяя рассуждения, сделанные при квантизации одного веса, получим, что в случае квантизации одновременно всего столбца матрицы весов оптимальное приращение $\delta\mathbf{W}$ и значений функции Лагранжа L может быть записано как:

$$\delta\mathbf{W} = \frac{\Delta \cdot \mathbf{H}_{q,:}^{-1}}{[\mathbf{H}^{-1}]_{qq}}, \quad L = \frac{1}{2} \frac{\|quant(\mathbf{W})_{:,q} - \mathbf{W}_{:,q}\|_2^2}{[\mathbf{H}^{-1}]_{qq}}$$

где $\mathbf{H} = 2\mathbf{X} \cdot \mathbf{X}^T$ - матрица Гессе.

Выполнение квантизации по столбцам значительно уменьшает вычислительные ресурсы требуемые для квантизации модели. Однако даже в этом случае процесс квантизации будет занимать достаточно продолжительное время, потому что для больших матриц необходимо будет обновлять все неквантизованные веса.

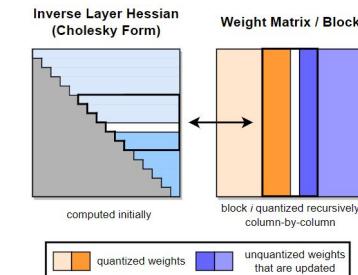
Для того чтобы получить дополнительный прирост к скорости разделим исходную матрицу весов на блоки, состоящие из B столбцов (в методе GPTQ берется 128 столбцов).

Тогда квантизацию внутри блока будем выполнять последовательно от столбца к столбцу, перераспределяя при этом ошибку квантизации на оставшиеся не квантизованные столбцы. После того как блок квантизован перераспределим накопленную ошибку на оставшуюся не квантизованную матрицу.

Для быстрого расчета обратной матрицы Гессе \mathbf{H}^{-1} используется разложение Холецкого.

При этом для повышения устойчивости расчета к диагональным элементам добавляется регуляризующее слагаемое λ :

$$\lambda = 0.01 * \text{mean}(\text{diag}(\mathbf{H}))$$



Algorithm 1 Quantize \mathbf{W} given inverse Hessian $\mathbf{H}^{-1} = (\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I})^{-1}$ and blocksize B .

```

 $\mathbf{Q} \leftarrow \mathbf{0}_{d_{\text{row}} \times d_{\text{col}}}$  // quantized output
 $\mathbf{E} \leftarrow \mathbf{0}_{d_{\text{row}} \times B}$  // block quantization errors
 $\mathbf{H}^{-1} \leftarrow \text{Cholesky}(\mathbf{H}^{-1})^\top$  // Hessian inverse information
for  $i = 0, B, 2B, \dots$  do
    for  $j = i, \dots, i + B - 1$  do
         $\mathbf{Q}_{:,j} \leftarrow \text{quant}(\mathbf{W}_{:,j})$  // quantize column
         $\mathbf{E}_{:,j-i} \leftarrow (\mathbf{W}_{:,j} - \mathbf{Q}_{:,j}) / [\mathbf{H}^{-1}]_{jj}$  // quantization error
         $\mathbf{W}_{:,j:(i+B)} \leftarrow \mathbf{W}_{:,j:(i+B)} - \mathbf{E}_{:,j-i} \cdot \mathbf{H}_{j,j:(i+B)}^{-1}$  // update weights in block
    end for
     $\mathbf{W}_{:, (i+B):} \leftarrow \mathbf{W}_{:, (i+B):} - \mathbf{E} \cdot \mathbf{H}_{i:(i+B), (i+B)}^{-1}$  // update all remaining weights
end for

```

OPT	Bits	125M	350M	1.3B	2.7B	6.7B	13B	30B	66B	175B
full	16	27.65	22.00	14.63	12.47	10.86	10.13	9.56	9.34	8.34
RTN	4	37.28	25.94	48.17	16.92	12.10	11.32	10.98	110	10.54
GPTQ	4	31.12	24.24	15.47	12.87	11.39	10.31	9.63	9.55	8.37
RTN	3	1.3e3	64.57	1.3e4	1.6e4	5.8e3	3.4e3	1.6e3	6.1e3	7.3e3
GPTQ	3	53.85	33.79	20.97	16.88	14.86	11.61	10.27	14.16	8.68

Значение переплексии на Wiki2 для модели OPT

Bits&Bytes

bitsandbytes - библиотека для квантизации трансформеров в int8, fp4, nf4

Библиотека включает в себя два метода LLM.int8() и поблочная квантизация в fp4/nf4 с возможностью дополнительной квантизации параметров масштабирования.



Dettmers, T., Lewis, M., Belkada, Y., & Zettlemoyer, L. Llm. int8 (): 8-bit matrix multiplication for transformers at scale, 2022. *CoRR abs/2208.07339*.

Dettmers, T., Pagnoni, A., Holtzman, A., & Zettlemoyer, L. (2024). QLoRA: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems*, 36.

<https://huggingface.co/blog/hf-bitsandbytes-integration>

<https://huggingface.co/blog/4bit-transformers-bitsandbytes>

LLM.int8()

LLM.int8() - метод квантизации весов и активаций LLM в int8 с возможностью повышения точности за счет выделения outliers.

Схема LLM.int8():

1. Квантизация активаций X и весов W

1.1 Для каждой строки матрицы X и каждого столбца* матрицы W определяются параметры масштабирования c_x и c_w . Коэффициенты масштабирования хранятся в fp16/bf16.

$$c_x(i) = \max(|X_{[i, :]}|), \quad c_w(j) = \max(|W_{[:, j]}|)$$

$$q_{\max} = 2^8/2 - 1 = 127$$

2. Forward pass

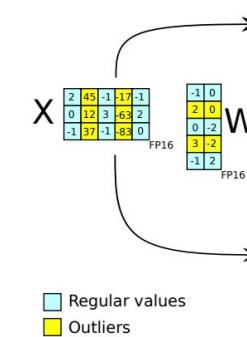
2.1 Вычисляем в int32 произведение целочисленных матриц:

$$\text{Out}_{ij}^{int32} = X_{ik}^{int8} W_{kj}^{int8}$$

Опционально для повышения точности можно отдельно выделить столбцы в активациях X , где есть хотя бы один элемент с очень большим значением (outliers), и соответствующие им столбцы из матрицы весов W в отдельную группу O . Обычно в качестве outliers рассматриваются элементы с абсолютным значением больше 6.0. Данная группа столбцов O из X не квантизуется, а произведение с соответствующей группой столбцов из W вычисляется в fp16. Полученный результат складывается с деквантизованным выходом.

$$\text{Out}_{ij}^{fp16} = \sum_{k \in O} X_{ik}^{fp16} W_{kj}^{int8} + \frac{c_x(i)c_w(j)}{127 * 127} \sum_{l \notin O} X_{il}^{fp16} W_{lj}^{int8}$$

LLM.int8()



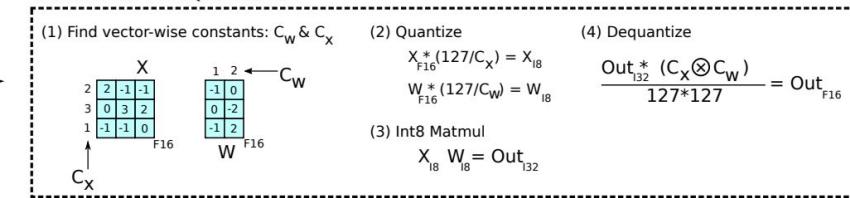
1.2 Вычисляются квантизованные веса:

$$X_{ij}^{int8} = \text{round} \left[\frac{127}{c_x(i)} X_{ij}^{fp16} \right], \quad W_{ij}^{int8} = \text{round} \left[\frac{127}{c_w(j)} W_{ij}^{fp16} \right]$$

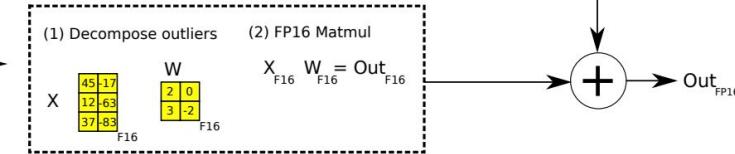
2.2 Деквантезируем из int32 в fp16

$$\text{Out}_{ij}^{fp16} = \frac{c_x(i)c_w(j)}{127 * 127} \text{Out}_{ij}^{int32}$$

8-bit Vector-wise Quantization



16-bit Decomposition



Bits&Bytes fp4/nf4

bitsandbytes fp4 / nf4 - метод квантизации весов LLM в fp4/nf4 с возможностью дополнительной квантизации коэффициентов масштабирования.

Схема квантизации в fp4/nf4:

1. Квантизация весов \mathbf{W}

- 1) Матрица \mathbf{W} преобразуется в одномерный массив \mathbf{w} , который разделяется на блоки \mathbf{w}_b в количестве $n = (h \times o) / B$, где h - количество входных признаков, o - количество выходящих признаков, B - размер блока. Размер блока фиксированный $B = 64$.
- 2) Для каждого блока \mathbf{w}_b находится свой коэффициент масштабирования c_b , равный значению максимального элемента в блоке. Параметры масштабирования хранятся в fp32.
- 3) В каждом интервале значения весов \mathbf{w}_b масштабируются в промежуток от [-1, 1] путем деления на коэффициент c_b .
- 4) Отмасштабированные значения конвертируются в 4 битный вещественнозначимый формат fp4 или nf4.

2. Дополнительная квантизация параметров масштабирования c_b .

С целью уменьшения памяти, требуемой для хранения параметров масштабирования c_b , они могут быть также квантованы.

- 1) Массив параметров разбивается на блоки размером 256 и для каждого блока выбирается выбирается коэффициент масштабирования c_{bb} .
- 2) Параметры c_b квантуются в fp8.

Forward Pass

1. Выполняется деквантизация коэффициентов масштабирования весов c_u

$$c_{b,i}^{fp32} = c_{b,i}^{fp8} c_{bb,i}^{fp32}, \quad c_{b,i}^{fp32} = c_{b,i}^{fp32} + \text{mean}(c_{b,i}^{fp32})$$

2. Выполняется деквантизация весов

$$\tilde{\mathbf{w}}_{b,i}^{fp16} = (c_{b,i}^{fp32} \mathbf{w}_{b,i}^{fp4/nf4})_{fp16}$$

$$c_{b,i}^{fp32} = \max(|\mathbf{w}_{b,i}^{fp16}|), i = 1, \dots, n; \quad \mathbf{w}_{b,i}^{fp4/nf4} = \text{convert} \left(\frac{\mathbf{w}_{b,i}^{fp16}}{c_{b,i}^{fp32}} \right)_{fp16 \rightarrow fp4/nf4}$$

В fp4 все вещественнозначимые числа описываются формулой:

$$X_{fp4} = (-1)^s \left(\frac{d_1}{2} + \frac{d_2}{2^2} + \frac{d_3}{2^3} \right), \quad s = \{0, 1\}, d_i = \{0, 1\}$$

nf4 - асимметричный формат нормально распределенных вещественных чисел

[-1.0, -0.6961928009986877, -0.5250730514526367, -0.39491748809814453, -0.28444138169288635, -0.1847734302282334, -0.09105003625154495, 0.0, 0.07958029955625534, 0.16093020141124725, 0.24611230194568634, 0.33791524171829224, 0.44070982933044434, 0.5626170039176941, 0.7229568362236023, 1.0]

$$c_{bb,i}^{fp32} = \max(c_{b,i}^{fp32}), i = 1, \dots, m; \quad c_{b,i}^{fp8} = \text{convert} \left(\frac{c_{b,i}^{fp32} - \text{mean}(c_{b,i}^{fp32})}{c_{bb,i}} \right)_{fp32 \rightarrow fp8}$$

3. Деквантизованные веса умножаются на активации:

$$\mathbf{Out}^{fp16} = \mathbf{X}^{fp16} \tilde{\mathbf{W}}^{fp16}$$

LoRA

Low-Rank Adaptation (LoRA) - метод дообучения LLM, в котором к весам оригинальной модели добавляются обучаемые адаптеры, состоящие из двух низкоранговых матриц. При этом оригинальные веса модели замораживаются.

QLoRA - квантизации LLM с помощью *bitsandbytes fp4 / nf4* и дообучение квантизированной модели с помощью **LoRA**.

$$\mathbf{W}_{orig} \in \mathbb{R}^{o \times h}, \mathbf{B} \in \mathbb{R}^{o \times r}, \mathbf{A} \in \mathbb{R}^{r \times h} \quad \mathbf{W}_{train} = \mathbf{W}_{orig} + \mathbf{BA}$$

Благодаря тому, что размерность r матриц \mathbf{A} , \mathbf{B} много меньше, чем размерности h и o матрицы \mathbf{W} , удается существенно сократить количество обучаемых параметров.

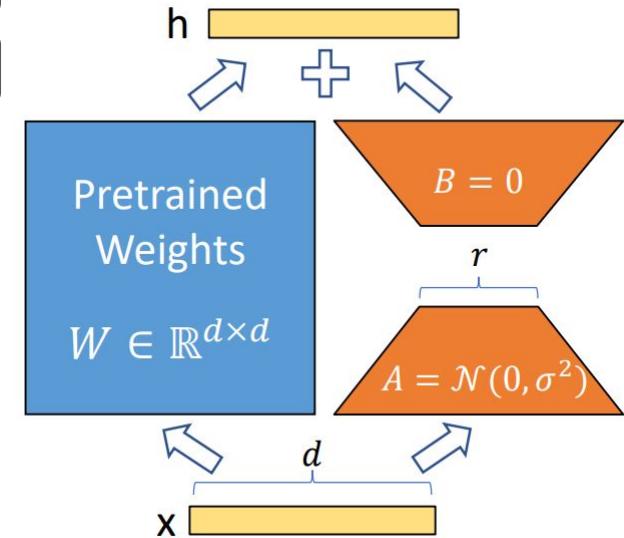
$$r \ll \min(h, o)$$

Матрица \mathbf{A} инициализируется случайными значениями из нормального распределения, Матрица \mathbf{B} принимается равной нулевой матрице.

В качестве предобученных весов \mathbf{W} модели могут быть взяты квантизованные веса $\text{quant}(\mathbf{W})$. Благодаря этому можно значительно уменьшить требования к объему памяти, необходимого для дообучения модели.

$$\text{Forward Pass} \quad \mathbf{Out} = \mathbf{XW}_{train}^T = \mathbf{XW}_{orig}^T + \mathbf{XA}^T \mathbf{B}^T$$

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., ... & Chen, W. (2021). Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.



Активации \mathbf{X} умножаются на веса \mathbf{W} модели и LoRA адаптеры \mathbf{A} , \mathbf{B} . Полученные матрицы складываются.

>>> Mixture of Experts

MoE: Mixture of Experts

Mixture of Experts - подход к решению общей задачи путем ее разделения между группой экспертов, каждый из которых специализируется на решении своей подзадачи.

Mixtral 8x7b применение подхода Mixture of Experts к трансформерам.

В качестве экспертов используются линейные слои.

Каждый линейный слой в блоке декодера GPT-трансформера заменяется на 8 линейных слоев экспертов.

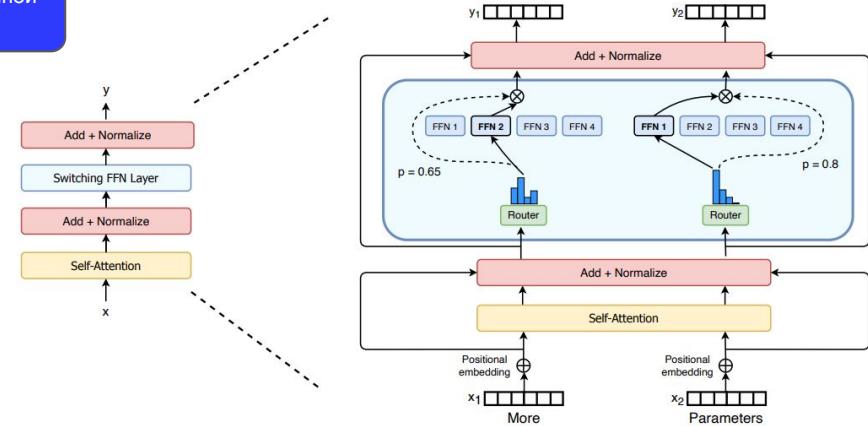
Обработка каждого токена выполняется 2-мя экспертами, которые выбираются Router слоем, в качестве которого используется линейный слой. К выходу каждого из отобранных экспертов применяется функция активации SwiGLU после чего результат складывается.

$$y = \sum_{i=0}^{n-1} \text{Softmax}(\text{Top2}(x \cdot W_g))_i \cdot \text{SwiGLU}_i(x)$$

Model	Active Params	MMLU	HellaS	WinoG	PIQA	Arc-e	Arc-c	NQ	TriQA	HumanE	MBPP	Math	GSM8K
LLaMA 2 7B	7B	44.4%	77.1%	69.5%	77.9%	68.7%	43.2%	17.5%	56.6%	11.6%	26.1%	3.9%	16.0%
LLaMA 2 13B	13B	55.6%	80.7%	72.9%	80.8%	75.2%	48.8%	16.7%	64.0%	18.9%	35.4%	6.0%	34.3%
LLaMA 1 33B	33B	56.8%	83.7%	76.2%	82.2%	79.6%	54.4%	24.1%	68.5%	25.0%	40.9%	8.4%	44.1%
LLaMA 2 70B	70B	69.9%	85.4%	80.4%	82.6%	79.9%	56.5%	25.4%	73.0%	29.3%	49.8%	13.8%	69.6%
Mistral 7B	7B	62.5%	81.0%	74.2%	82.2%	80.5%	54.9%	23.2%	62.5%	26.2%	50.2%	12.7%	50.0%
Mixtral 8x7B	13B	70.6%	84.4%	77.2%	83.6%	83.1%	59.7%	30.6%	71.5%	40.2%	60.7%	28.4%	74.4%

<https://huggingface.co/blog/moe#:~:text=So%2C%20what%20exactly%20is%20a.expert%20is%20a%20neural%20network.>

Jiang, A. Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., ... & Sayed, W. E. (2024). Mixtral of experts. arXiv preprint arXiv:2401.04088.



Архитектура Sparse MoE слоев

Семинар: pipeline для QLoRA с использование русского
инструктивного датасета.

https://github.com/On-Point-RND/Efficient-DL-models-Seminars/blob/master/Week%204/Seminars/LLM_quantization/QLoRA_seminar.ipynb

Домашнее задание: анализ падения точности
OPT-350m при использовании различных методов
квантизации и спарсификации.

<https://github.com/On-Point-RND/Efficient-DL-models-Seminars/tree/master/Home%20Work/HW%201>