Advance in Efficient Large Language Models Serving Systems

Yunpeng Tai

Nov 12, 2024

School of Data Science
Chinese University of Hong Kong, Shenzhen

Outline

Outline

Success of LLMs

Generative large language models (LLMs) have become a driving force behind significant advancements in artificial intelligence (AI) and have demonstrated exceptional performance across a wide range of language-related tasks.

- Text Translation
- Text Paraphrasing
- Code Assistance
- ...

Cost of LLMs

Challenge

However, the pricing of LLMs prevents their widespread deployment in real-world applications.

Company	Model	1M input($)	1M output($)
OpenAI	o1-preview	15	60
OpenAI	GPT-4o	2.5	10
Anthropic	Claude 3 Opus	15	75
Anthropic	Claude 3.5 Sonnet	3	15
Google	Gemini 1.5 pro	1.25	5

Cost of LLMs

Challenge

However, the pricing of LLMs prevents their widespread deployment in real-world applications.

Company	Model	1M input($)	1M output($)
OpenAI	o1-preview	15	60
OpenAI	GPT-4o	2.5	10
Anthropic	Claude 3 Opus	15	75
Anthropic	Claude 3.5 Sonnet	3	15
Google	Gemini 1.5 pro	1.25	5

- The large model size and complexity of LLMs lead to the expensive computational requirements during deployment.

Research Question [1]

Question

What is an efficient large language models serving system?

Research Question [1]

Question

What is an efficient large language models serving system?

1. Low latency and fast response time.

Research Question [1]

Question

What is an efficient large language models serving system?

1. Low latency and fast response time.
2. Small memory consumption on devices.

Research Question [1]

Question

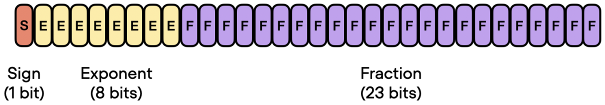
What is an efficient large language models serving system?

1. Low latency and fast response time.
2. Small memory consumption on devices.
3. High throughput to simultaneous requests.

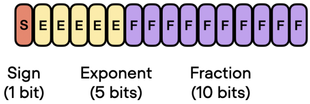
Outline

Numerical Precision

float 32



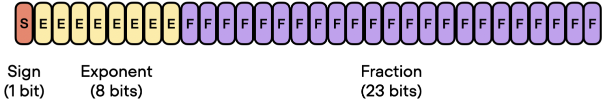
float 16 ("half" precision)



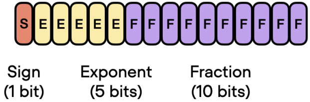
Float32 $\rightarrow$ Float16 leads to lower memory consumption and faster computing speed.

Numerical Precision

float 32



float 16 ("half" precision)



Float32 → Float16 leads to lower memory consumption and faster computing speed.

However, the numerical precision has decreased primarily due to fewer bits allocated for the exponent.

Computational Errors

```
>>> torch.tensor(10**6, dtype=torch.float32)
tensor(1000000.)
>>> torch.tensor(10**6, dtype=torch.float16)
tensor(inf, dtype=torch.float16)
```

Low numerical precision is likely to cause some computational errors such as inf and nan.

Computational Errors

```
>>> torch.tensor(10**6, dtype=torch.float32)
tensor(1000000.)
>>> torch.tensor(10**6, dtype=torch.float16)
tensor(inf, dtype=torch.float16)
```

Low numerical precision is likely to cause some computational errors such as inf and nan.

- How to use reduced bits while maintain the performance?

Which to Quantize

For LLMs at and beyond 6.7B parameters, the feed-forward and attention layers and their matrix multiplication account largely for the memory and the computation complexity [2].

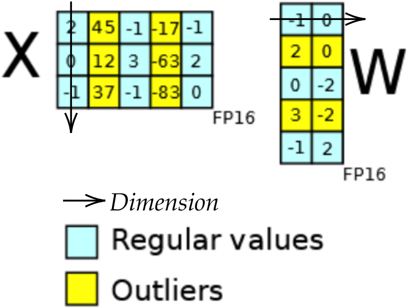
Which to Quantize

For LLMs at and beyond 6.7B parameters, the feed-forward and attention layers and their matrix multiplication account largely for the memory and the computation complexity [2].

$\downarrow$

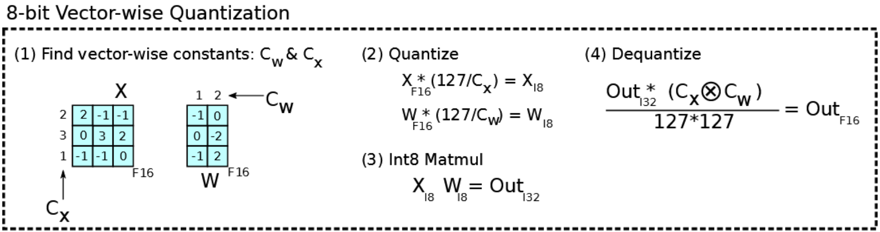
Quantize the parameters of the feed-forward and attention layers and perform their matrix multiplication in less bits format such as INT8 or INT4 during inference process.

LLM.int8() [3]



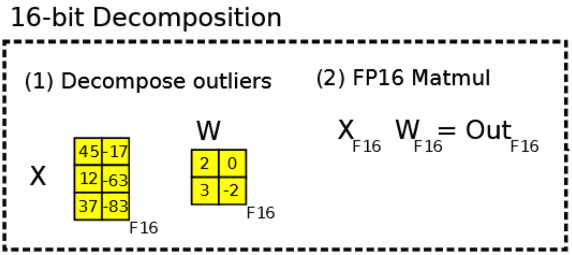
1. We identify the regular values and the outliers that exhibit a magnitude significantly larger than that of the other values across other dimensions.

LLM.int8() [3]



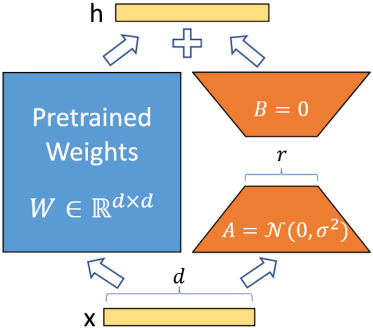
2. To process regular values, first quantize the matrices to INT8 format, then perform the multiplication. Finally, convert the resulting product into FP16 format.

LLM.int8() [3]



3. To address the outliers, perform the multiplication in FP16 format, as they hurt the performance when using INT8 format.

LoRA [4]



- Freeze the original model parameters.
- Initialize a small set of trainable parameters for certain components W of the model.
- Decompose the update in a low-rank manner.
  $W = W + \Delta W \rightarrow W = W + BA$

QLoRA [5]

1. Quantize the pretrained model in 4-bit NormalFloat format which yields better results than INT4 and FP4 for normally distributed data.

QLoRA [5]

2. Introduce the Double Quantization which is a method that quantizes the quantization constants resulting from quantizing the model from FP32 format to FP8 format.

Outline

Why Parallel

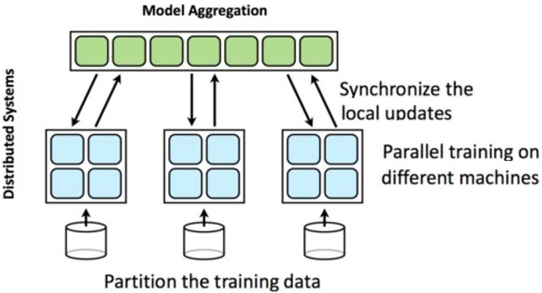
1. Parallel computation enhances the efficiency of the training process, particularly when dealing with large datasets and complex models.

Why Parallel

1. Parallel computation enhances the efficiency of the training process, particularly when dealing with large datasets and complex models.

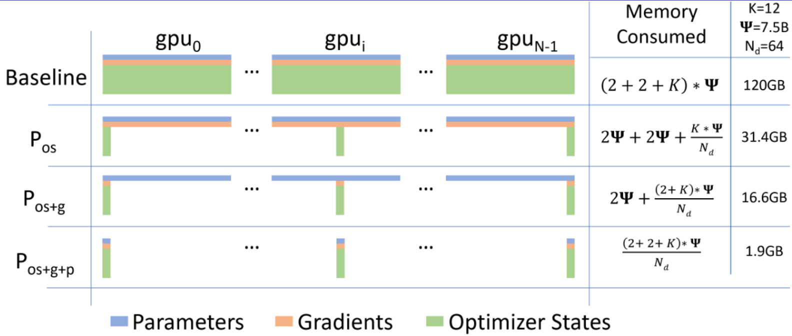
2. Parallel computation enhances convergence stability by allowing multiple instances to be learned simultaneously, leading to improved performance.

The Procedure



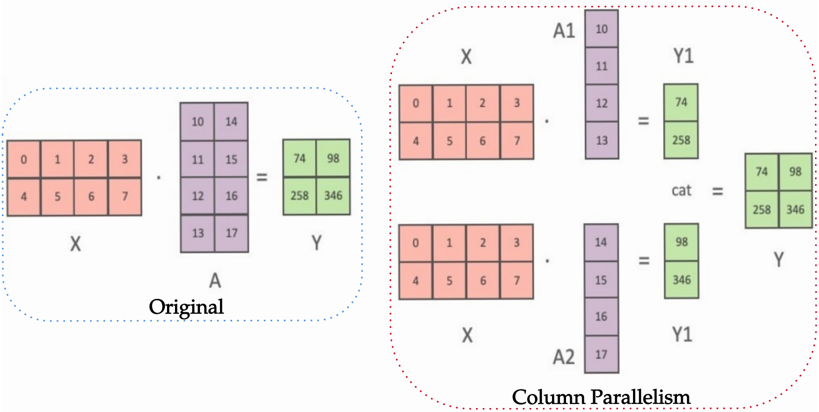
1. Partition the training data.
2. Parallel training on multiple machines.
3. Synchronize the updates from multiple machines.
4. Update the model and forward the updates to machines. Repeat from step 2.

ZeRO [6]



1. P_{os} : shards the optimizer states (save 73.8%).
2. P_{os+g} : shards the optimizer states and gradients (save 86.2%).
3. P_{os+g+p} : shards the optimizer states, gradients, and parameters (save 98.4%)

Tensor Parallelism [7]



- Split the matrix by its rows or columns, perform the independent multiplication on multiple devices, and accumulate the multiplication results.

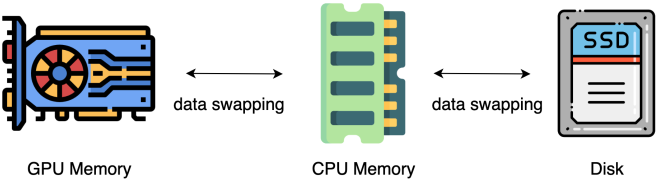
Pipeline Parallelism [8]

```
===================  ===================
|  0 | 1 | 2 | 3  |  |  4 | 5 | 6 | 7  |
===================  ===================
        GPU0                 GPU1
```

- Given that one single GPU fails to fit a whole model, we can put the layers of the model on multiple devices.
- For instance, the model executes the computations for the first four layers on one GPU, while the remaining four layers are processed on a second GPU.

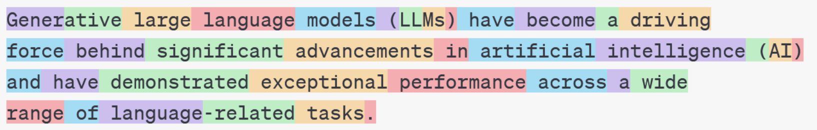
Outline

The Procedure [9]



When GPU memory is exhausted, it is possible to transfer data to CPU memory or even to disk for temporary storage.

Text → Tokens



The original input text is converted into "tokens," represented as distinct color blocks in the image below (e.g., Generative → Gener & ative).

https://platform.openai.com/tokenizer

Text Generation

Algorithm 1: Auto-Regressive Decoding for LLM Inference

Initialize the input sequence X_0 with a given context or start token
for $t = 1$ *to* T **do**
    Predict the next token $y_t = \text{argmax}_y P(y|X_{t-1})$
    Update the input sequence $X_t = X_{t-1} \oplus y_t$
    if y_t *is EOS* **then**
        break
    end
end

- $P(y|X_{t-1})$ represents the probability of the next token y given the current sequence X_{t-1}, and $\oplus$ denotes the concatenation operation.
- The argmax function is used to select the most probable next token at each step.

Computational Redundancy

Algorithm 1: Auto-Regressive Decoding for LLM Inference

```
Initialize the input sequence X_0 with a given context or start token
for t = 1 to T do
    Predict the next token y_t = argmax_y P(y|X_{t-1})
    Update the input sequence X_t = X_{t-1} ⊕ y_t
    if y_t is EOS then
        break
    end
end
```

1. When predicting a new token at position t, the model needs to walk through the previous context $(1, \cdots, t-1)$.

Computational Redundancy

Algorithm 1: Auto-Regressive Decoding for LLM Inference

Initialize the input sequence X_0 with a given context or start token
for $t = 1$ *to* T **do**
    Predict the next token $y_t = \text{argmax}_y P(y|X_{t-1})$
    Update the input sequence $X_t = X_{t-1} \oplus y_t$
    if y_t *is* EOS **then**
        break
    end
end

1. When predicting a new token at position t, the model needs to walk through the previous context $(1, \cdots, t-1)$.
2. However, the previous context $(1, \cdots, t-1)$ exhibits significant overlap with the context for predicting a new token at position $t-1$.

Computational Redundancy

Algorithm 1: Auto-Regressive Decoding for LLM Inference

Initialize the input sequence X_0 with a given context or start token
for $t = 1$ *to* T **do**
    Predict the next token $y_t = \text{argmax}_y P(y|X_{t-1})$
    Update the input sequence $X_t = X_{t-1} \oplus y_t$
    if y_t *is EOS* **then**
        break
    end
end

1. When predicting a new token at position t, the model needs to walk through the previous context $(1, \cdots, t-1)$.
2. However, the previous context $(1, \cdots, t-1)$ exhibits significant overlap with the context for predicting a new token at position $t-1$.
3. Each time the model predicts a new token, it must re-calculate previously computed results, thus leading to computational redundancy.

KV Cache

1. Store the previous computation results into a cache.
2. Avoid computational redundancy by retrieving the information from the cache instead of re-computation.
3. The inference process is then accelerated by utilizing a cache.

Challenges of KV Cache

The naive implementation of KV cache is to pre-allocate a contiguous memory with a maximum sequence length assumption.

Challenges of KV Cache

The naive implementation of KV cache is to pre-allocate a contiguous memory with a maximum sequence length assumption.

↓

1. Requests of various output lengths take up the same memory.
2. The total memory can not be fully utilized due to the memory fragmentation.

Improved Implementation of KV Cache

1. vLLM proposes paged attention that partitions the KV cache into non-contiguous memory blocks and significantly improves the batch size as well as throughput [10].

Improved Implementation of KV Cache

1. vLLM proposes paged attention that partitions the KV cache into non-contiguous memory blocks and significantly improves the batch size as well as throughput [10].

2. SpecInfer proposes tree attention and depth-first tree traversal to eliminate redundant KV cache allocation for multiple output sequences sharing the same prefix [11].

Improved Implementation of KV Cache

1. vLLM proposes paged attention that partitions the KV cache into non-contiguous memory blocks and significantly improves the batch size as well as throughput [10].
2. SpecInfer proposes tree attention and depth-first tree traversal to eliminate redundant KV cache allocation for multiple output sequences sharing the same prefix [11].
3. LightLLM uses token-level memory management mechanism to reduce memory usage. [12]

Outline

Conclusion

How to build efficient large language models serving systems?

1. Low-Bit Quantization
2. Parallel Computation
3. Memory Management

The above frameworks makes huge progress in achieving low latency, small memory consumption, and high throughput.

Future

1. For low-bit quantization, there may be more stable quantization methods for broad scales of LLMs which also aligns with the scaling law of LLMs.

Future

1. For low-bit quantization, there may be more stable quantization methods for broad scales of LLMs which also aligns with the scaling law of LLMs.

2. For parallel computation, the latency introduced by the communication may be better handled to further speed up the computation.

Future

1. For low-bit quantization, there may be more stable quantization methods for broad scales of LLMs which also aligns with the scaling law of LLMs.

2. For parallel computation, the latency introduced by the communication may be better handled to further speed up the computation.

3. For memory management, the performance degradation caused by the fine-grained memory strategies may be improved without losing the memory efficiency.

Outline

[1] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Hongyi Jin, Tianqi Chen, and Zhihao Jia.
Towards efficient generative large language model serving: A survey from algorithms to systems.
arXiv preprint arXiv:2312.15234, 2023.

[2] Gabriel Ilharco, Cesar Ilharco, Iulia Turc, Tim Dettmers, Felipe Ferreira, and Kenton Lee.
High performance natural language processing.
In Aline Villavicencio and Benjamin Van Durme, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Tutorial Abstracts*, pages 24–27, Online, November 2020. Association for Computational Linguistics.

[3] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer.
Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale.
Advances in Neural Information Processing Systems, 35:30318–30332, 2022.

[4] Edward J Hu, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al.
Lora: Low-rank adaptation of large language models.
In *International Conference on Learning Representations*.

[5] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer.
Qlora: Efficient finetuning of quantized llms.
Advances in Neural Information Processing Systems, 36, 2024.

[6] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He.
Zero: memory optimizations toward training trillion parameter models.
In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16, 2020.

[7] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro.
Megatron-lm: Training multi-billion parameter language models using model parallelism.
arXiv preprint arXiv:1909.08053, 2019.

[8] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia.
Memory-efficient pipeline-parallel dnn training.
In *International Conference on Machine Learning*, pages 7937–7947. PMLR, 2021.

[9] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He.
{Zero-offload}: Democratizing {billion-scale} model training.
In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564, 2021.

[10] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica.
Efficient memory management for large language model serving with pagedattention.
In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

[11] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, et al.
Specinfer: Accelerating large language model serving with tree-based speculative inference and verification.
In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 932–949, 2024.

[12] Lightllm; commit: 789637f.
https://github.com/ModelTC/lightllm.
Accessed on: 2024-10-24.

Any Question?