S61691

Accelerating End-to-End Large Language Models System using a Unified Inference Architecture and FP8

Xiaonan Nie
Baichuan-AI Inc. & Peking University
xiaonan.nie@pku.edu.cn

Jack Chen
NVIDIA
jackch@nvidia.com

2024.03.19

Presenter

Xiaonan Nie, Leader of the MLSys Group at Baichuan-AI Inc. Xiaonan Nie concentrates on developing the infrastructure that enables training and serving kinds of Baichuan-series models across thousands of GPUs.

Jack Chen, NVIDIA DevTech Team Manager.
Jack Chen mainly focuses on inference acceleration of NLP and CV models, one of the key developers of FasterTransformer and participant in TensorRT-LLM development to help enable quantization features.

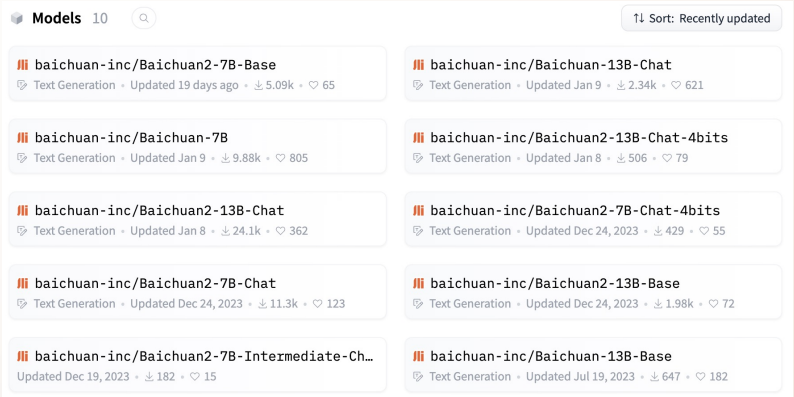
Introducing Baichuan Intelligent Technology

Baichuan-AI Inc., established on April 10, 2023, has emerged as the most rapidly evolving AGI startup in China, and has open sourced models, technical report and training checkpoints.

- **State-of-the-art Infrastructure:** Releasing one significant LLM model every 28 days.
- **Advanced Technology:** Our open-source models outperform similar-sized models, e.g., LLaMA2; Our proprietary models achieve the top performance in handling hallucinations among all models in China.
- **Pioneering commercial open-source in Chinese:** Filling the gap in the domestic market for large models that are open-source, free, and commercially viable, achieving over 5 million downloads.

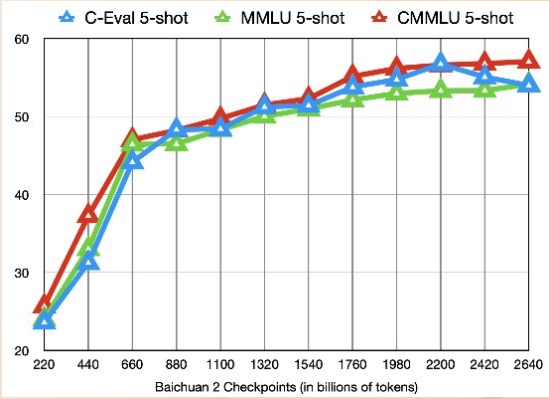
Introducing Baichuan Intelligent Technology

If interested, please access to our *open-sourced models* at https://huggingface.co/baichuan-inc , or access to our *proprietary models* through our chat platform and API at https://www.baichuan-ai.com .



Baichuan Models

Technical Report



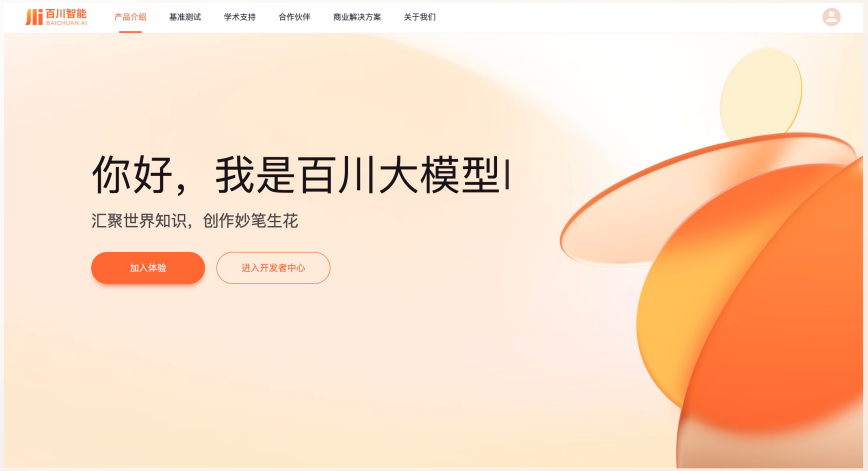
Training Checkpoints

Outline

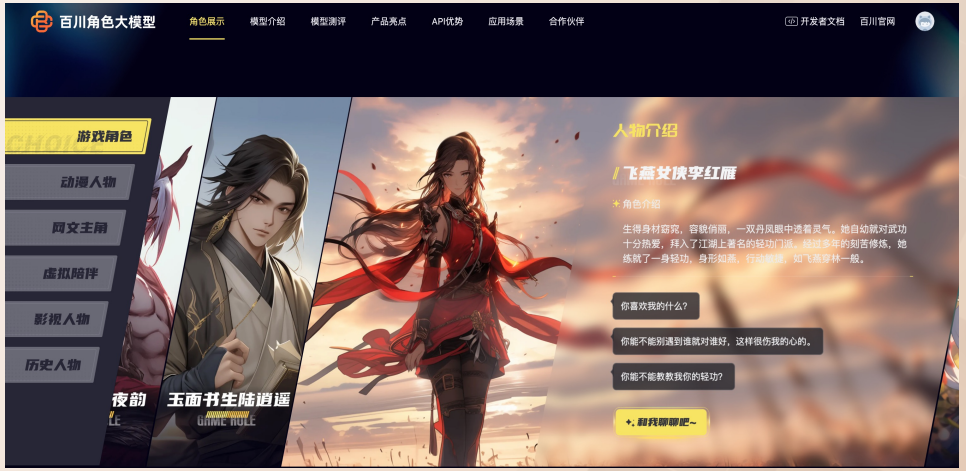
1. **Background and Challenges**
2. The design of our inference system
3. Typical workloads & our best practice
4. Summary and future work

Large language models bring new applications

- Large language models (LLMs) have the remarkable ability to comprehend human intentions, promoting interactions that are increasingly intuitive and effective.
  - ➢ Enhancing production efficiency through **Chat Model**.
  - ➢ Entertainment and engagement with **Character Model**.



Chat Model
baichuan-ai.com

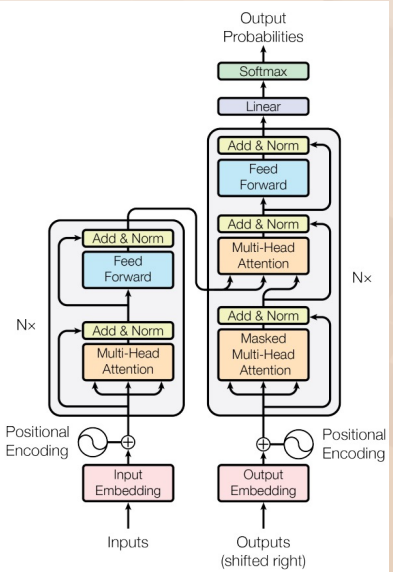


Character Model
npc.baichuan-ai.com

Backbone of LLMs: Transformer

- Most LLMs are built on the Transformer architecture[1].

- Models deployed in Baichuan can be roughly categorized into two aspects:
  - ➢ Auto-regressive decoder models, e.g., GPT.
    - *Dialog Model*: Powers conversational agents by generating human-like responses.
    - *Intentional Understanding Model*: Interprets the user's intent, enabling more accurate responses and actions by understanding the context and purpose of queries.
  - ➢ Non-auto-regressive encoder models, e.g., BERT.
    - *Safety Model*: Identifies and filters inappropriate or harmful content.
    - *Quality Model*: Assesses and ensures the quality of generated content, focusing on relevance, accuracy, and coherence to meet high standards.

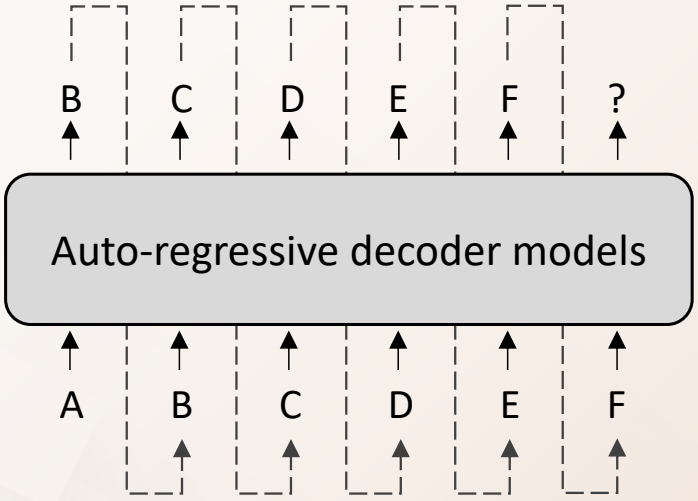


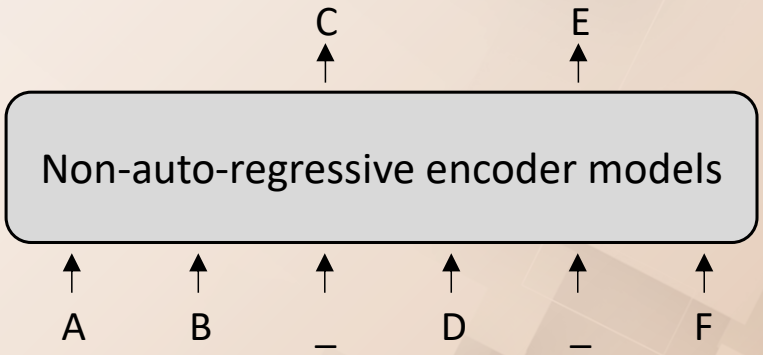
Transformer[1]

[1] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need[J]. NeurIPS 2017

Backbone of LLMs: Transformer

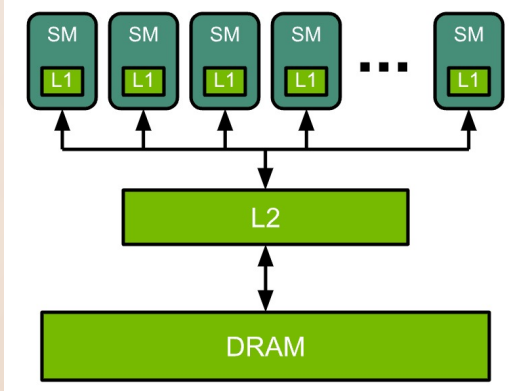
- Models deployed in Baichuan can be roughly categorized into two aspects:
  - ➢ Auto-regressive decoder models, e.g., GPT.
    - These models generate text one word at a time, predicting the next word based on the previous ones.
  - ➢ Non-Auto-regressive encoder models, e.g., BERT.
    - Generate outputs in a parallel fashion, improving efficiency for certain tasks..





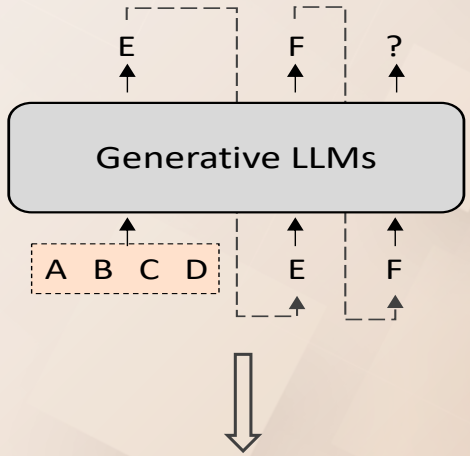
GPU: The underlying hardware for deep learning

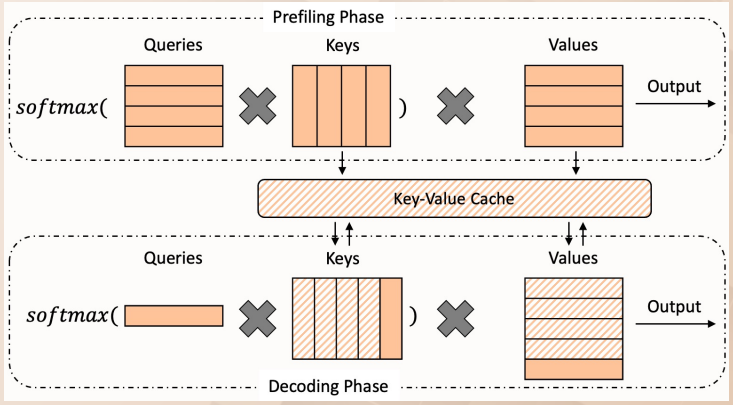
- GPUs have become indispensable in deep learning, providing the parallel processing capabilities required to train and inference LLMs efficiently.
  - ➢ Streaming Multiprocessors (SMs):
    - Enable fast parallel processing of the matrix multiplications and other linear algebra operations that are fundamental to LLMs computations.
  - ➢ L1 Cache (~KB):
    - Enable the fastest data access of each SM.
  - ➢ L2 Cache (~MB):
    - Enable faster data access of all SMs.
  - ➢ High-Bandwidth Memory (~GB):
    - Enable fast data transfer to the L1/L2 cache than traditional GDDR memory.



How do LLMs perform generation?

- Generative LLMs produce text sequentially, generating one output token at a time, and respond to a user query in two phases:
  - Prefilling Phase: Time to first token, TTFT
    - Process *the entire tokens* within the input prompt/context in one iteration and generate the first output token. (*one iteration*)
    - Users receive the first token of the response from LLMs.
  - Decoding Phase: Time per output token, TPOT
    - Based on the prompt/context and the previous generated tokens, generate the subsequent output tokens one at a time until finished. (*multiple iterations*)
    - Users receive the complete response from LLMs sequentially, one token at a time.





Details of prefiling and decoding phase

Why is it hard to accelerate LLM generation?

We analyzed LLM generation workloads and summarized the following challenges:

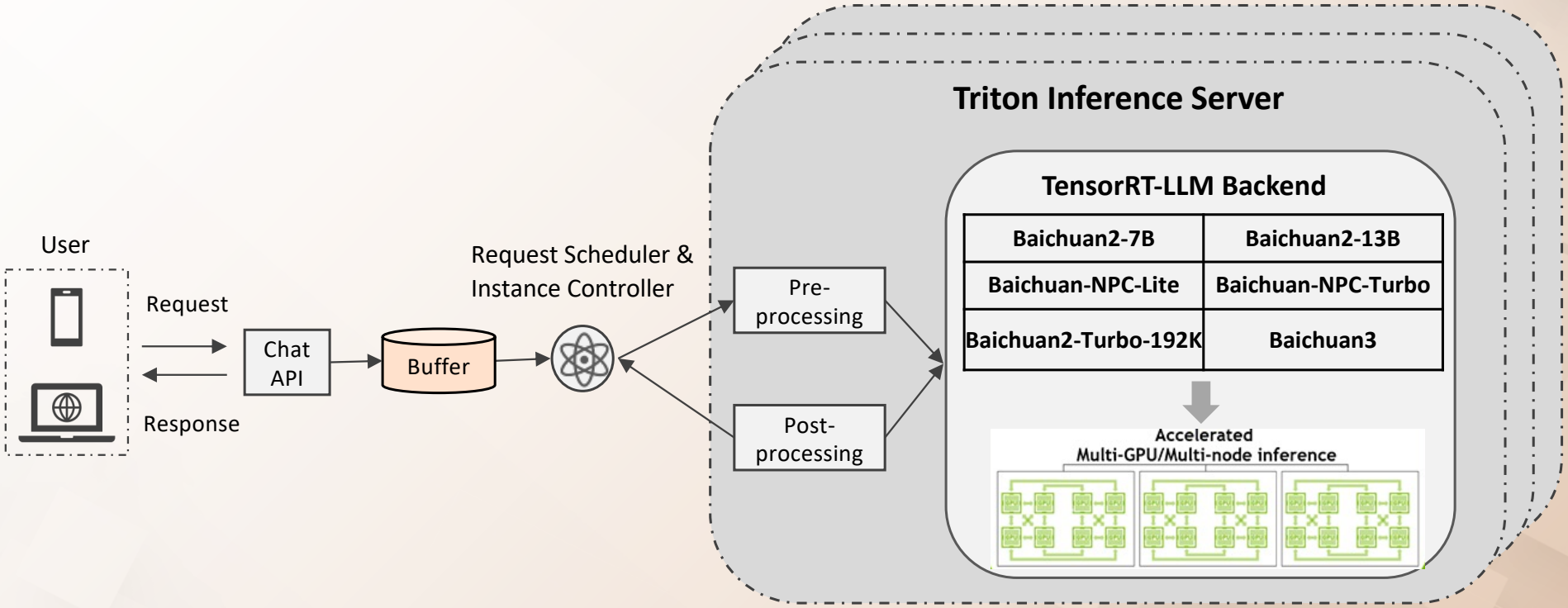
1. **Decoding efficiency**: For one-token-at-a-time generation, it is usually limited by GPU memory bandwidth and underutilize the GPU computational ability.
2. **Attention mechanism dependency**: it needs to store all previous key-value caches in GPU memory until finished, which accounts for most of available GPU memory.
3. **Different characteristics of prefilling and decoding**: general matrix multiply (GEMM) for prefilling and general matrix-vector multiply (GEMV) for decoding when batch size = 1.
4. **Unpredictable prompt(input) lengths**: the generation latency is affected by the longest prompt within the input batch; different lengths will cause the memory fragments of KV-cache because of frequent allocation & release.

Outline

1. Background and challenges
2. **The design of our inference system**
3. Typical workloads & our best practice
4. Summary and future work

The design of our inference system

We build our inference system based on the software ecosystem of NVIDIA, e.g., Triton Inference Server and TensorRT-LLM.



Strategies in our inference system

1. General acceleration strategies:
    a. **Paged KV Cache of Attention** for efficient memory management
    b. **Tensor Parallelism & Pipeline Parallelism** for distributed inference
    c. **Fused Kernels**, e.g., Flash-attention, for efficient computation
2. Model-specific strategies:
    a. **Quantization**, e.g.,INT4/INT8/FP8 for weight, KV-Cache and activation, for reducing memory footprints and the size of memory access
    b. **Group-query Attention** for reducing the size of KV-Cache
3. Phase-specific strategies:
    a. **Speculative decoding** for accelerating decoding of huge models
    b. **Flash-decoding** for accelerating decoding of long-context models

Outline

1. Background and Challenges
2. The design of our inference system
3. **Typical workloads & our best practice**
   - Baichuan2-7B/13B: commonly used model size
   - Baichuan2-turbo-192K: long context model
   - Baichuan3: over 100B parameters
4. Summary and future work

Baichuan2-7B/13B

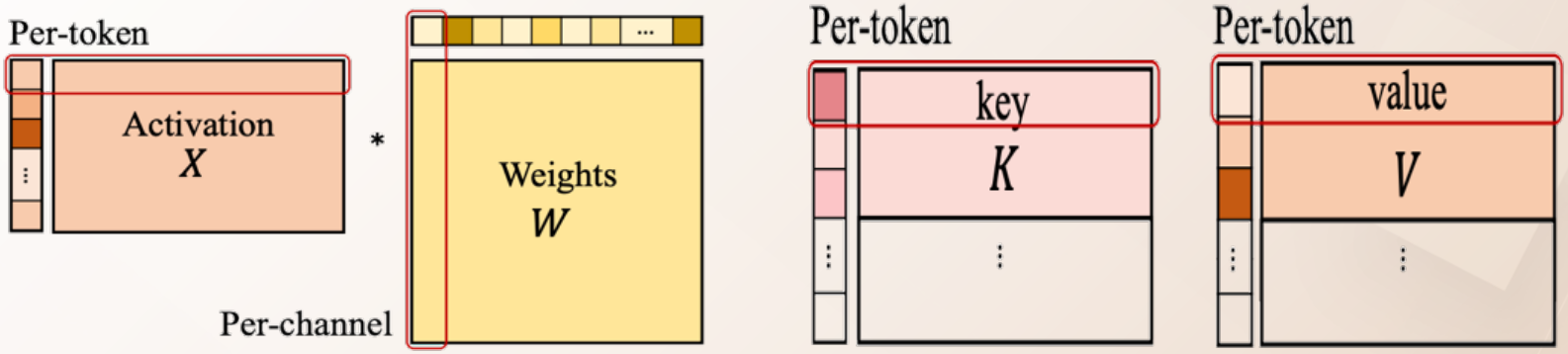
On September 9th, we released Baichuan2-7B/13B, which were trained on a 2.6 trillion tokens, and outperformed models of similar size in various benchmarks.

	C-Eval	MMLU	CMMLU	Gaokao	AGIEval	BBH
	5-shot	5-shot	5-shot	5-shot	5-shot	3-shot
GPT-4	68.40	83.93	70.33	66.15	63.27	75.12
GPT-3.5 Turbo	51.10	68.54	54.06	47.07	46.13	61.59
LLaMA-13B	28.50	46.30	31.15	28.23	28.22	37.89
LLaMA2-13B	35.80	55.09	37.99	30.83	32.29	46.98
Vicuna-13B	32.80	52.00	36.28	30.11	31.55	43.04
Chinese-Alpaca-Plus-13B	38.80	43.90	33.43	34.78	35.46	28.94
XVERSE-13B	53.70	55.21	58.44	44.69	42.54	38.06
Baichuan-13B-Base	52.40	51.60	55.30	49.69	43.20	43.01
Baichuan2-13B-Base	58.10	59.17	61.97	54.33	48.17	48.78

	JEC-QA	CEval-MMLU-CMMLU	MedQA-USMLE	MedQA-MCMLE	MedMCQA
	5-shot	5-shot	5-shot	5-shot	5-shot
GPT-4	59.32	77.16	80.28	74.58	72.51
GPT-3.5 Turbo	42.31	61.17	53.81	52.92	56.25
LLaMA-13B	27.54	35.14	28.83	23.38	39.52
LLaMA2-13B	34.08	47.42	35.04	29.74	42.12
Vicuna-13B	28.38	40.99	34.80	27.67	40.66
Chinese-Alpaca-Plus-13B	35.32	46.31	27.49	32.66	35.87
XVERSE-13B	46.42	58.08	32.99	58.76	41.34
Baichuan-13B-Base	41.34	51.77	29.07	43.67	39.60
Baichuan2-13B-Base	47.40	59.33	40.38	61.62	42.86

Baichuan2-7B/13B (W8KV8A16)

- By compressing weights and KV cache to 8-bit, we found that only little model quality drop is introduced in Baichuan2-7B/13B.



	Activation	Weight	Q	K	V	CEval	MMLU	CMMLU
Baichuan2-Chat-7B	FP16	INT8	FP16	INT8	INT8	0.001	-0.0007	-0.0006
Baichuan2-Chat-13B	FP16	INT8	FP16	INT8	INT8	-0.0074	-0.0017	-0.0021
Baichuan2-Base-7B	FP16	INT8	FP16	INT8	INT8	0.002	-0.001	-0.0003
Baichuan2-Base-13B	FP16	INT8	FP16	INT8	INT8	-0.0022	-0.0013	0

Accuracy on different benchmarks compared with FP16 activation and FP16 weight

Baichuan2-7B/13B (W8KV8A16)

- By compressing weights and KV cache to 8-bit, we successfully increased the average generation throughput (tokens/s) to $1.40\times$ and $1.15\times$, respectively.

A100*, TP = 1 1000 sequences from alpaca_gpt4_data_zh.json benchmark with gptManagerBenchmark TokensPerSecond = (total_output_tokens/bacth_size)/total_latency			
batch	FP16 inflight batching (IFB) TokensPerSecond	int8 weight only + int8 kv cache inflight batching (IFB) TokensPerSecond	ratio
4	24.80	41.67	1.68
8	23.49	35.97	1.53
12	22.38	31.00	1.39
16	21.77	30.07	1.38
20	20.12	28.45	1.41
24	19.47	27.00	1.39
28	18.09	25.54	1.41
32	17.08	23.09	1.35
36	15.71	21.60	1.37
40	14.85	20.46	1.38
44	14.08	18.96	1.35
48	12.96	17.95	1.38
52	12.30	16.78	1.36
56	11.51	15.84	1.38
60	11.66	14.86	1.27
64	10.47	14.34	1.37

*for technical reference & discussion only

A100*, TP = 2 1000 sequences from alpaca_gpt4_data_zh.json benchmark with gptManagerBenchmark TokensPerSecond = (total_output_tokens/bacth_size)/total_latency			
batch	FP16 inflight batching (IFB) TokensPerSecond	int8 weight only + int8 kv cache inflight batching (IFB) TokensPerSecond	ratio
4	37.17	51.64	1.39
8	30.95	38.63	1.25
12	29.41	33.39	1.14
16	28.32	32.00	1.13
20	25.81	29.34	1.14
24	24.41	27.93	1.14
28	22.88	26.47	1.16
32	21.48	24.42	1.14
36	20.25	23.49	1.16
40	19.50	22.60	1.16
44	18.35	21.50	1.17
48	17.01	20.08	1.18
52	16.03	18.86	1.18
56	15.27	18.09	1.18
60	14.67	17.23	1.18
64	14.68	17.21	1.17

*for technical reference & discussion only

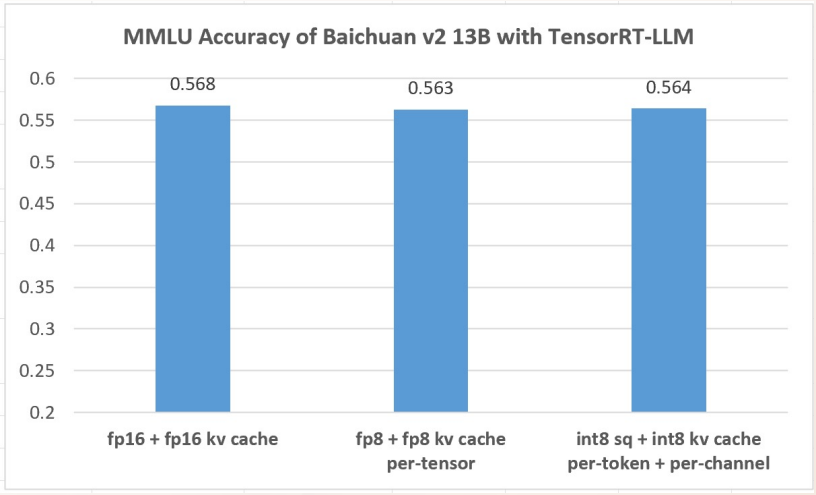
Advancements in Hopper and Ada Series: FP8

- Enhanced Computational Precision
  - The Hopper and Ada Lovelace GPUs introduce FP8 computational precision.
  - Theoretically, this advancement enables calculations at twice the speed of the previous BF16 standard.
- Optimization of Memory Utilization
  - Significant reduction in the size of activations and weights memory.
  - This leads to more efficient use of GPU memory resources and improved overall performance.
- Improved Throughput
  - Enhanced processing capabilities result in better throughput.
  - Allows for faster data processing and analysis, optimizing workload management.
- Enhanced First Token Latency
  - Noticeable improvement in the latency for processing the first token.
  - Contributes to faster startup times and more responsive system performance.

The Fundamentals of FP8 Computation

- FP8

Baichuan v2 13B with TensorRT-LLM on Hopper gpu input seqlen/output seqlen = 1024/128					
	tokens/s				
batch	fp8 + fp8 kv cache	fp16 + fp16 kv cache	speedup fp8 vs fp16	int8 sq + int8 kv cache	speedup fp8 vs int8
1	82.59	57.15	1.45	77.8	1.06
8	586.12	328.84	1.78	531.72	1.10
16	1054.44	686.36	1.54	939.62	1.12
32	1777.73	1175.74	1.51	1538.71	1.16



- FP8 has 1.45x~1.78x speedup compared to FP16.
- FP8 has 1.06x~1.16 speedup compared to INT8, thanks to a). better hopper style fp8 gemm; b). simpler quantization strategy and better kernel fusion.
- FP8 achieves 99.1% accuracy of fp16 on Baichuan v2 13B.
- FP8 uses a much simpler quantization strategy to achieve similar accuracy compared to INT8.

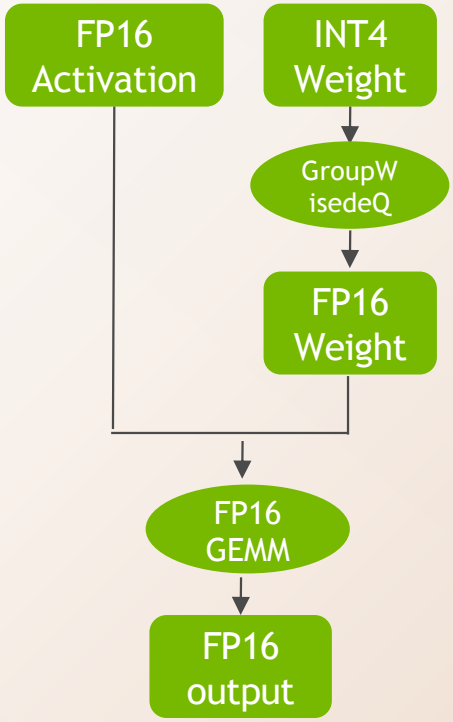
The Fundamentals of FP8 Computation

- W4A(FP)8

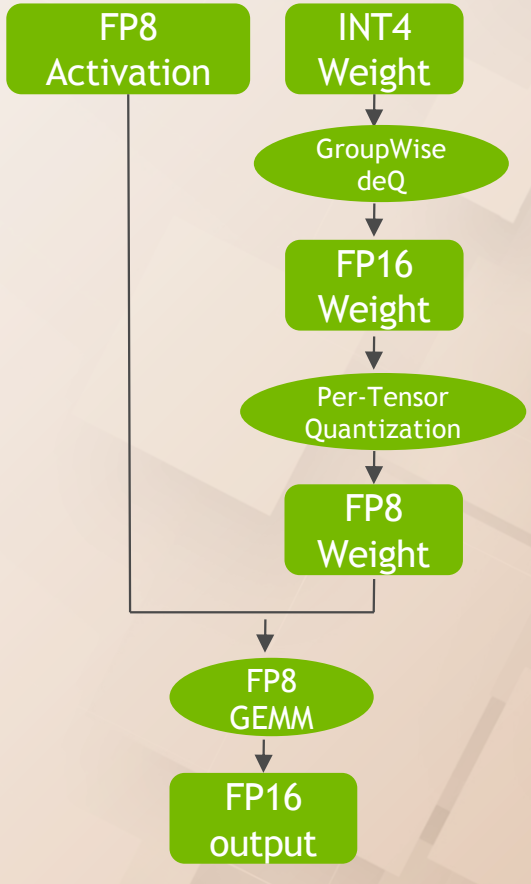
- w4a* with int4 weight can reduce the device memory cost of weight. ¼ of FP16 weight & ½ of FP8 weight.

- w4a* with int4 weight can accelerate LLM inference with small batch size * beam search, which is memory bound cases.

- with fp8 tensor core, w4a(fp)8 GEMM has better performance than w4a16 in compute heavy cases.



INT4 weight * FP16 activation (w4a16) GEMM



INT4 weight * FP8 activation (w4a(fp)8) GEMM

The Fundamentals of FP8 Computation

- W4A(FP)8 implementation with cutlass

- A brief introduction of the open-sourced cutlass example : 55_hopper_mixed_dtype_gemm.
- These kernels are used in TensorRT-LLM with some modification. We use these modified kernel to generation the following test results.

Step 1: Define a mma op used in main loop

```
// We specify scale + zero elements to indicate that we require both. Scales and biases have the same format.
using CollectiveMainloopScaleWithZeroPoint = typename cutlass::gemm::collective::CollectiveBuilder<
    ArchTag, OperatorClass,
    cute::tuple<ElementB, ElementScale, ElementZero>, LayoutB_Transpose, AlignmentB,
    ElementA, LayoutA_Transpose, AlignmentA,
    ElementAccumulator,
    TileShape, ClusterShape,
    cutlass::gemm::collective::StageCountAutoCarveout<
      static_cast<int>(sizeof(typename CollectiveEpilogue::SharedStorage))
    >,
    KernelSchedule
  >::CollectiveOp;
```

- B should be int4 weight with groupwise scales and zeros.
- A should be fp8 activation.

Step 2: Define an epilogue op

```
using CollectiveEpilogue = typename cutlass::epilogue::collective::CollectiveBuilder<
    cutlass::arch::Sm90, cutlass::arch::OpClassTensorOp,
    TileShape, ClusterShape,
    EpilogueTileType,
    ElementAccumulator, ElementAccumulator,
    // Transpose layout of D here since we use explicit swap + transpose
    // the void type for C tells the builder to allocate 0 smem for the C matrix.
    // We can enable this if beta == 0 by changing ElementC to void below.
    ElementC, typename cutlass::layout::LayoutTranspose<LayoutC>::type, AlignmentC,
    ElementD, typename cutlass::layout::LayoutTranspose<LayoutD>::type, AlignmentD,
    EpilogueSchedule // This is the only epi supporting the required swap + transpose.
  >::CollectiveOp;
```

The Fundamentals of FP8 Computation

- W4A(FP)8 implementation with cutlass

Step 3: Define a gemm kernel with mma op and epilogue op

```
using GemmKernelScaleWithZeroPoint = cutlass::gemm::kernel::GemmUniversal<
    Shape<int,int,int,int>, // Indicates ProblemShape
    CollectiveMainloopScaleWithZeroPoint,
    CollectiveEpilogue
>;
```

Step 4: Define an adapter as fp8*int4 gemm type

```
using GemmScaleWithZeroPoint = cutlass::gemm::device::GemmUniversalAdapter<GemmKernelScaleWithZeroPoint>;
```

Step 5: Define an instance of fp8*int4 gemm type

```
GemmScaleWithZeroPoint gemm;
```

Step 6: Check the argument and initialize

```
// Check if the problem size is supported or not
CUTLASS_CHECK(gemm.can_implement(arguments));

// Initialize CUTLASS kernel with arguments and workspace pointer
CUTLASS_CHECK(gemm.initialize(arguments, workspace.get()));
```

Step 7: Run

```
CUTLASS_CHECK(gemm.run());
```

- A, B, scales & zeros should be placed in arguments

Advantages of W4A(FP)8 on Baichuan2-13B

- Speedup the computation of GEMM kernels

Baichuan v2 13B with TensorRT-LLM on Hopper gpu input seqlen/output seqlen = 1024/128				
batch	**type**	**w4a(fp)8 (ms)**	**w4a16 (ms)**	**speedup w4a(fp)8 vs w4a16**
1	QKV gemm in context phase	0.580	1.162	2.003
1	MLP FC1 in context phase	0.511	1.021	1.998
1	QKV gemm in generation phase	0.024	0.024	1.000
1	MLP FC1 in generation phase	0.027	0.027	1.000
32	QKV gemm in context phase	18.345	32.680	1.781
32	MLP FC1 in context phase	16.347	36.814	2.252
32	QKV gemm in generation phase	0.041	0.055	1.341
32	MLP FC1 in generation phase	0.043	0.055	1.279

- The **generation phase** of LLM is usually memory bound, especially when batch size is small.
  - Batch size = 1, w4a(fp)8 has the same performance of w4a16.
  - Batch size = 32, less memory bound with more compute, w4a(fp)8 has 1.27x~1.34x speedup compared to w4a16.
- The **context phase** of LLM is usually compute bound.
  - w4a(fp)8 has 1.78x~2.25x speedup compared to w4a16. The speedup increases with the shape of gemm.

Advantages of W4A(FP)8 on Baichuan2-13B

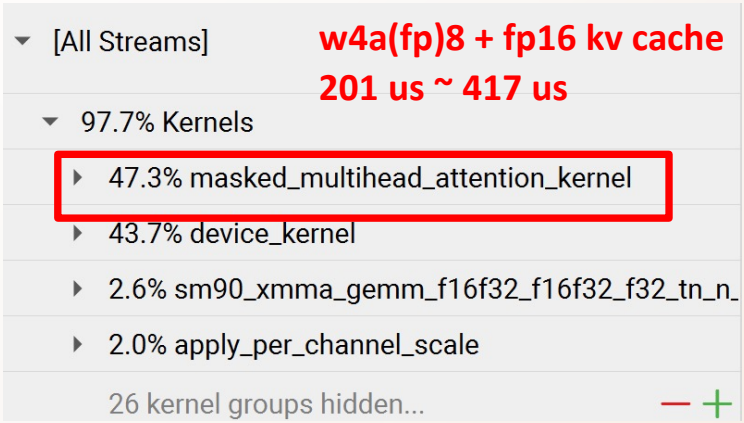
- Speedup the first token latency

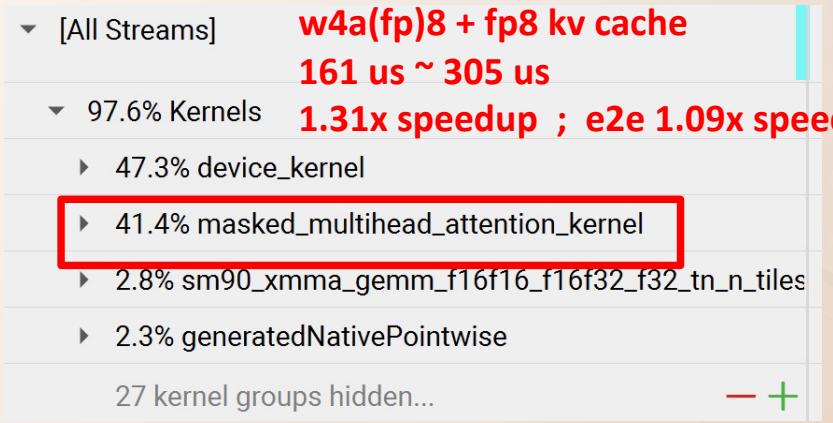
Baichuan v2 13B with TensorRT-LLM on Hopper gpu batch size = 1 first token latency			
input seqlen	**w4a16 (ms)**	**w4a(fp)8 (ms)**	**speedup**
1k	203	116.21	1.75
16k	3988.75	2552.48	1.56
32k	9857.51	7071.87	1.39
64k	26848.51	21237.88	1.26

- w4a(fp)8 is 1.26x~1.75x faster than w4a16 in first token latency.
- When the input seqlen increases, the first token latency speedup of w4a(fp)8 becomes smaller because we use FP16 FMHA in both case. We are working on FP8 FMHA.

Advantages of W4A(FP8) on Baichuan2-13B

- Speedup of the FP8 KV Cache
  - FP8 kv cache speedup on MMHA (batch = 32, input/output seqlen = 1024/1024, TensorRT-LLM):





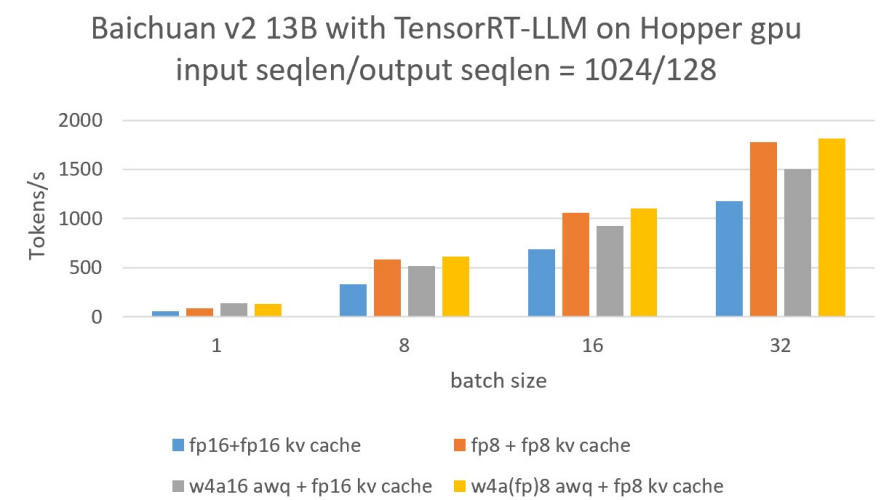
  - FP8 kv cache with larger batch size increases throughput:

Baichuan v2 13B with TensorRT-LLM on Hopper gpu input/output seqlen = 1024/128			
batch size	**w4a16 (tokens/s)**	**w4a(fp)8 (tokens/s)**	**speedup**
64	1818.37	2525.5	1.39
120	OOM	2733.4	1.50

Advantages of W4A(FP)8 on Baichuan2-13B

- E2E Performance of w4a(fp)8 on Baichuan v2 13B

Baichuan v2 13B with TensorRT-LLM on Hopper gpu input seqlen/output seqlen = 1024/128				
	tokens/s			
batch	fp16+fp16 kv cache	fp8 + fp8 kv cache	w4a16 awq + fp16 kv cache	w4a(fp)8 awq + fp8 kv cache
1	57.15	82.59	134.04	127.18
8	328.84	586.12	514.77	610.87
16	686.36	1054.44	921.27	1100.44
32	1175.74	1777.73	1500.58	1811.69

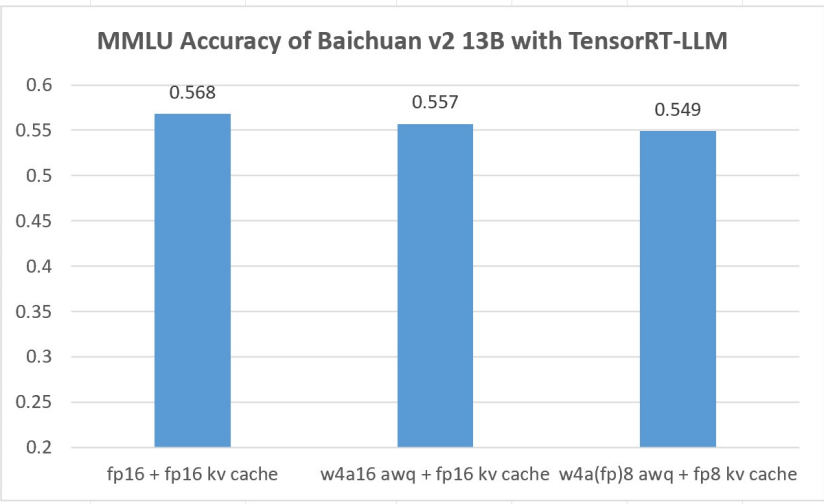


- For small batches, LLM is memory-bound, where w4a16 and w4a(fp)8 decrease loading latency, performing similarly, but w4a(fp)8 outperforms fp8 & fp16.
- With large batches, LLM becomes compute-bound; w4a16 leverages fp16 tensor cores, and w4a(fp)8 uses fp8 tensor cores, making w4a(fp)8 faster. w4a(fp8) matches fp8's performance.

Advantages of W4A(FP)8 on Baichuan2-13B

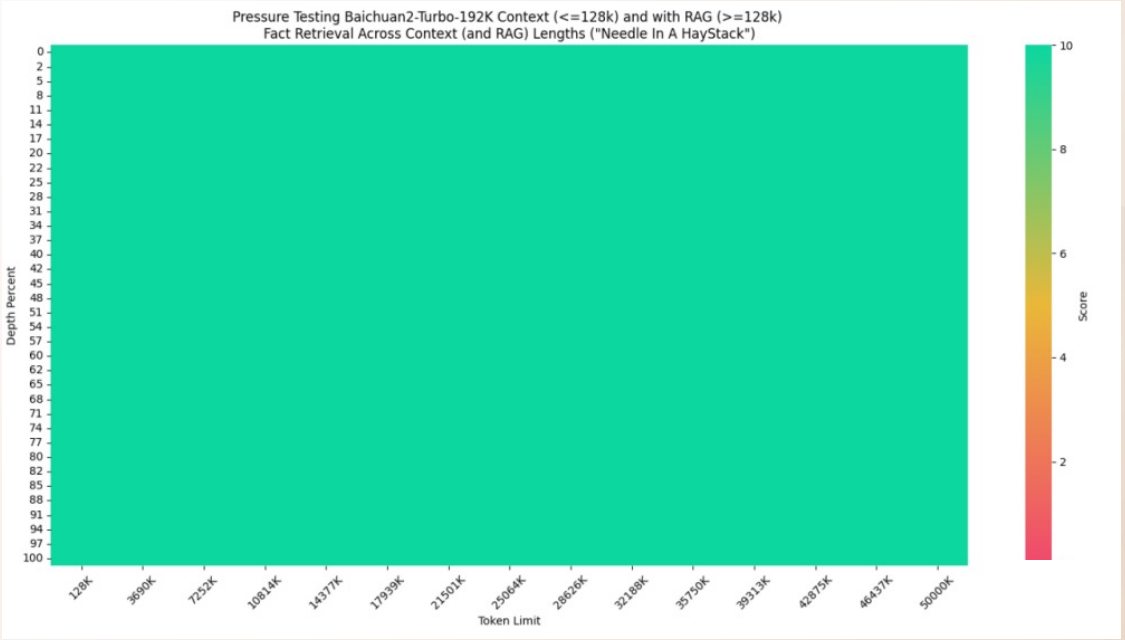
- Model performance on downstream tasks
  - ➢w4a(fp)8 awq achieves 96.6% accuracy of fp16 on Baichuan v2 13B.

MMLU Accuracy of Baichuan v2 13B with TensorRT-LLM	
dataType	**avg score**
fp16 + fp16 kv cache	0.568
w4a16 awq + fp16 kv cache	0.557
w4a(fp)8 awq + fp8 kv cache	0.549



Baichuan2-Turbo-192K: Long-Context Model

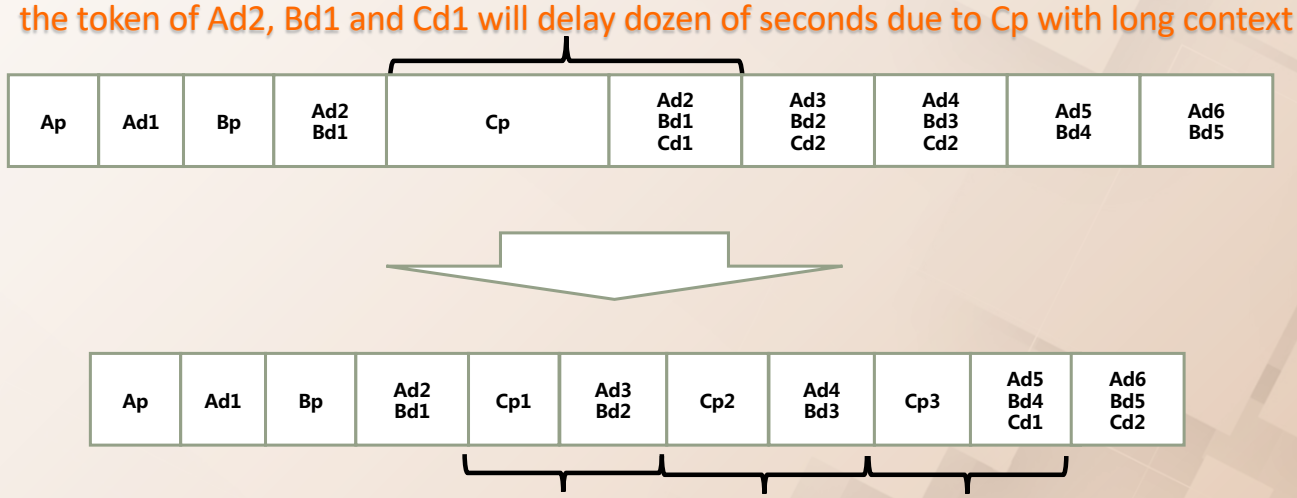
- On December 19th, we released the API of our search-enhanced Baichuan2-Turbo-192K model, which not only supports an ultra-long context window of 192K but also integrate an advanced search-enhanced knowledge base, further extending its capabilities.



"Needle in a Haystack" Evaluation(Green in the best), where we assess our model across as many as 50000K tokens.

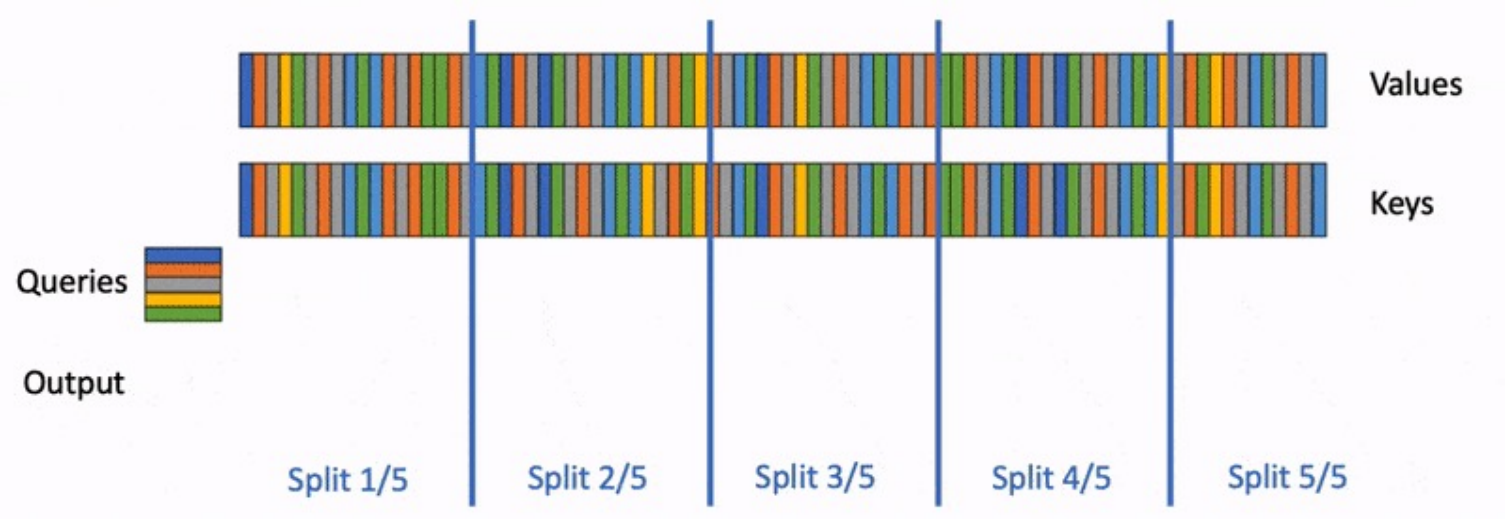
Baichuan2-Turbo-192K: Long-Context Model

- Challenges of deploying Baichuan2-Turbo-192K:
  - ➢ The computation in the prefilling phase for long contexts can block the decoding phase of other contexts.
  - ➢Solutions: Long contexts are partitioned into much smaller chunks and scheduled across multiple forward passes.



Baichuan2-Turbo-192K: Long-Context Model

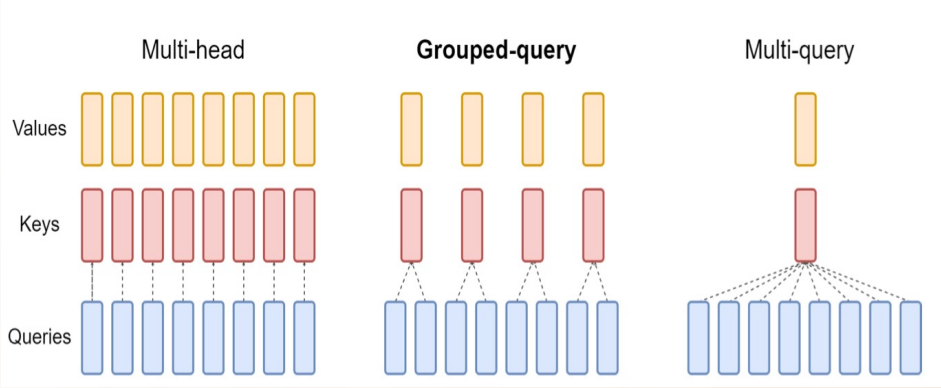
- Challenges of deploying Baichuan2-Turbo-192K:
  - ➢ The operator handling long contexts cannot fully utilize GPU resources.
  - ➢ Solution: Flash-decoding[1] parallelizes the computation along the sequence dimension, and introduces an reduction step to get the final results.



[1] https://crfm.stanford.edu/2023/10/12/flashdecoding.html

Baichuan2-Turbo-192K: Long-Context Model

- Beyond: Group-Query Attention:
  - ➢ Long context' KV cache occupies a large amount of GPU memory.
  - ➢ GQA greatly reduces the size of KV cache by sharing Key and Value cache among different attention heads.



Group Query Attention[1]

Batch Size	Seq Length	Hidden size	Layer	**#Head**	**#KV Head**	MHA KV cache memory (FP16)	GQA KV cache memory (FP16)	GQA KV cache memory (Int8)
1	1024	8192	80	**64**	**8**	2.5	0.3125	0.15625
16	1024	8192	80	**64**	**8**	40	5	2.5
32	1024	8192	80	**64**	**8**	80	10	5
64	1024	8192	80	**64**	**8**	160	20	10

Total KV memory = batch_size * seq_len *(d_model / n_heads) * n_layers * 2 (K + V) * 2 * n_KV_heads

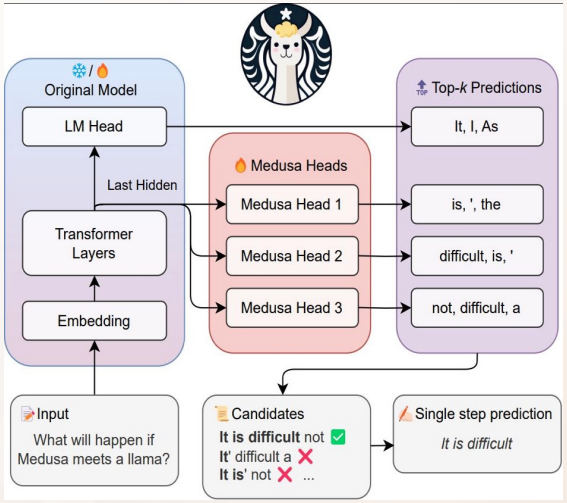
[1] Ainslie, Joshua, et al. "Gqa: Training generalized multi-query transformer models from multi-head checkpoints." *arXiv* 2023.

Baichuan3: >100B parameters

- Challenges of deploying Baichuan3 Model
  - ➢ Enormous GPU memory requirements; for instance, GPT3-175B requires 175GB of memory just to store its 8-bit parameters, which is equal to 7.3 times an A10 GPU(24GB).
  - ➢ How can we efficiently serve models with over one hundred billion parameters on cost-effective GPUs, such as the A10?
    1. Cost-effective GPUs support only limited batch sizes, leading to reduced throughput when serving large models on these GPU servers.
    2. Large models require distributed inference, but these GPUs lack NVLink and rely on communication through limited-bandwidth PCIe.

Baichuan3: >100B parameters

- Cost-effective GPUs support only limited batch sizes, leading to reduced throughput when serving large models on these GPU servers.
  - ➢ Solution: **Medusa based speculative decoding**. Compared to traditional schemes for speculative decoding involving both large and small models, Medusa eliminates the need to consider data interaction and resource allocation among multiple model services, simplifying deployment.



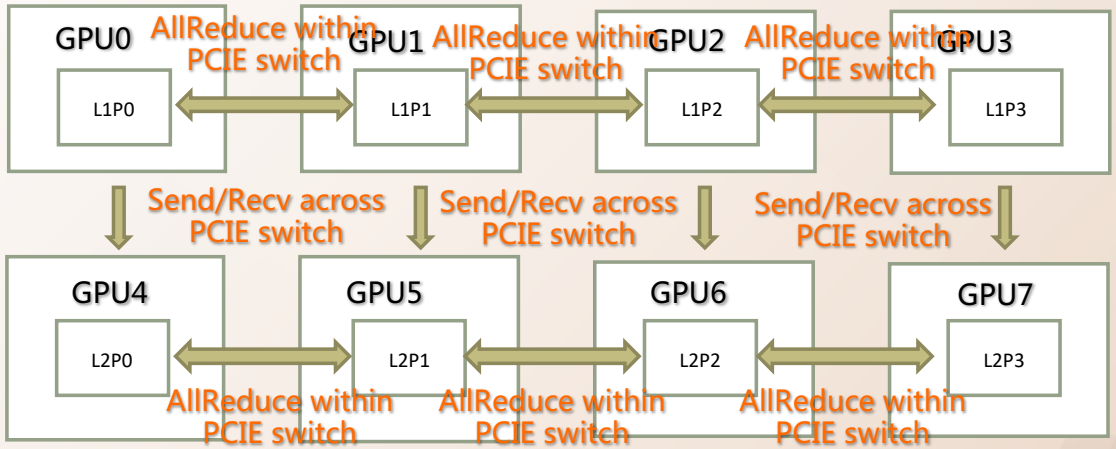
Overall throughput improvement

input/output	number of concurrent requests						
	1	2	4	8	16	32	64
1k/512	65.8%	63.3%	51.4%	33.8%	18.6%	2.9%	0.3%
4k/512	62.4%	60.1%	53.2%	37.5%	21.7%	16.0%	
8k/512	59.3%	56.2%	50.3%	32.0%	28.2%		
16k/512	57.0%	53.1%	48.8%	30.1%			

Cai, Tianle, et al. "Medusa: Simple llm inference acceleration framework with multiple decoding heads." *arXiv* (2024).

Baichuan3: >100B parameters

- Large models require distributed inference(TP+PP), but these GPUs lack NVLink and rely on communication through limited-bandwidth PCIe.
  - ➢ Solution: Modify the mapping of model partitions according to the topology of GPU devices and implement efficient asynchronous communication operations.



Summary and Future Work

- In this talk, we illustrated the challenges of LLM (Large Language Model) inference systems, along with the design of our unified inference system and its comprehensive optimization strategies.
- Then, we introduced our best practices for deploying the Baichuan series models, which include billion-parameter models (Baichuan2-7B/13B), a long-context model (Baichuan2-Turbo-192K), and models with over a hundred billion parameters (Baichuan3).
- In the future, we will focus on the following directions:
  - Integrating training and inference to ensure the timeliness of models.
  - Training better small models based on the knowledge of large models.
  - Accelerating the inference of MoE (Mixture of Experts) models and multi-modal models.
  - …

Thanks for listening!

We're open to collaboration and look forward to joining us!

Please contact xiaonan.nie@pku.edu.cn.