MVAPICH

MPI, PGAS and Hybrid MPI+PGAS Library

HiBD

High-Performance Big Data

HiDL

High-Performance Deep Learning

Flover: A Temporal Fusion Framework for Efficient Autoregressive Model Parallel Inference

Follow us on

https://twitter.com/mvapich

Jinghan Yao, Nawras Alnaasan, Tian Chen,

Aamir Shafi, Hari Subramoni, D.K. Panda

The Ohio State University

E-mail: shafi.16@osu.edu

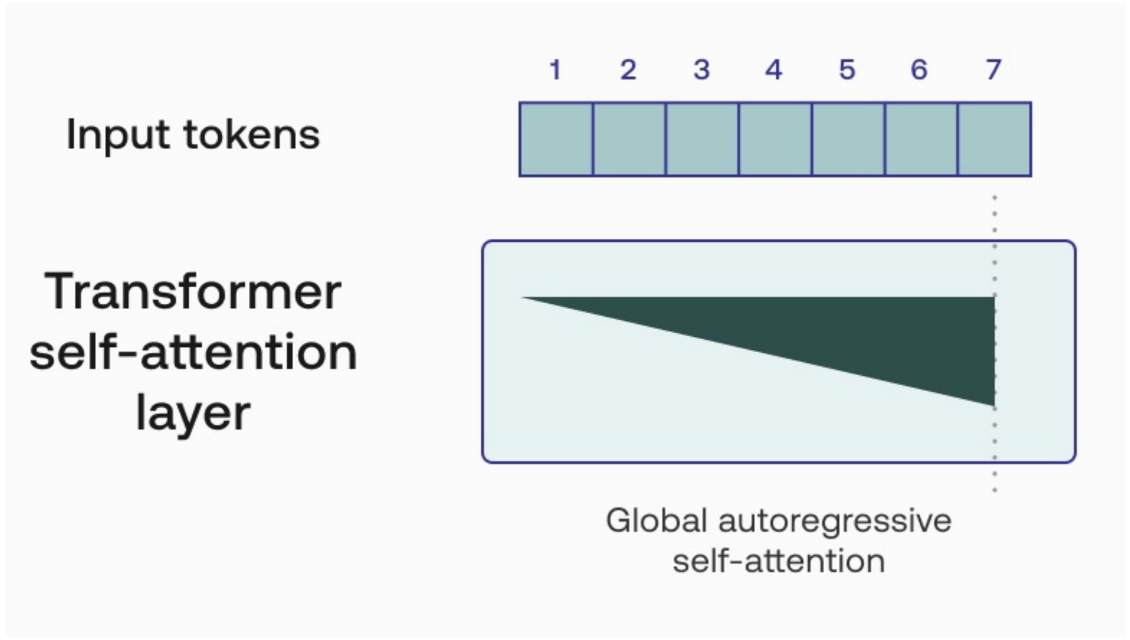
Presentation Outline

- Introduction to Autoregressive models
  - Deployment scenarios of inference Generative LLMs
  - Existing parallel inference methods
- Designs, implementations, and experiments
  - Temporal fusion of multiple random requests
  - Adaptive memory shuffle for creating contiguous memory
- Demo
- Summary

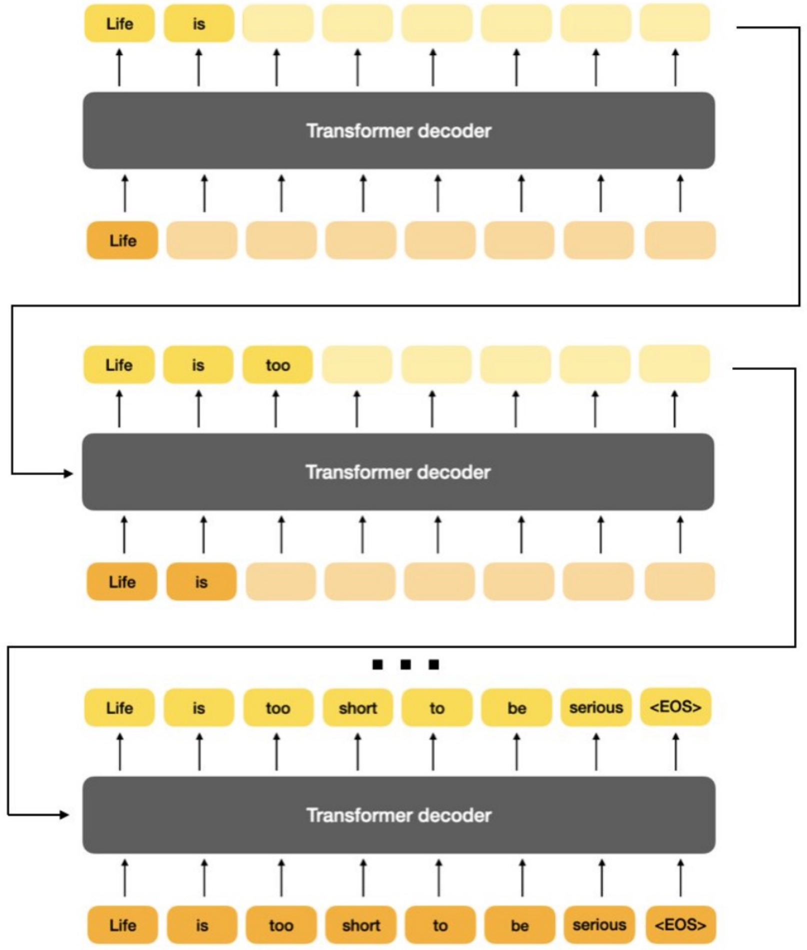
Introduction to Autoregressive Models

- An autoregressive model is a type of time series model that uses observations from previous time points to predict future values.

Generative LLMs are all autoregressive models that follow a time dependency in generating new tokens.



1
2
3
4
5
6
7
Input tokens
Transformer self-attention layer
Global autoregressive self-attention



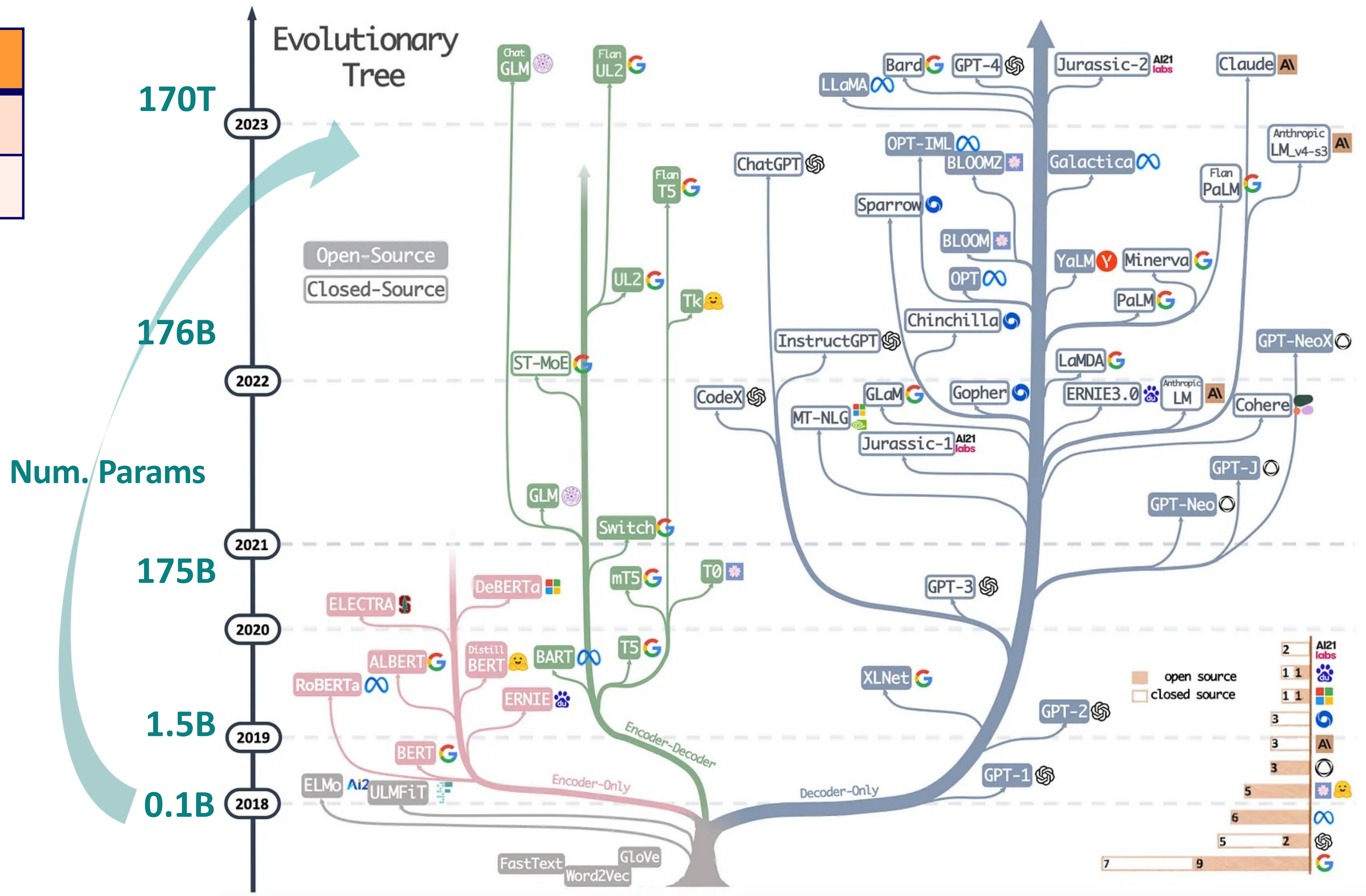
Parallel Inference on large language models? Overview

- LLM training & inference

	Phase	Sensitivity
Training	Model-learning	Throughput
Inference	**User-facing**	**Latency**

- Inference: Latency-sensitive
  - Final phase of deep learning
  - The closest end to users
- Smaller batch size in the workflow
  - Less efficient GPU utilization
- Users' requests arrive randomly
  - Very hard to parallelize
- Performed by various hardware
  - Single/multi GPUs, edge devices
- Response time is the most crucial
  - Long waiting is not acceptable

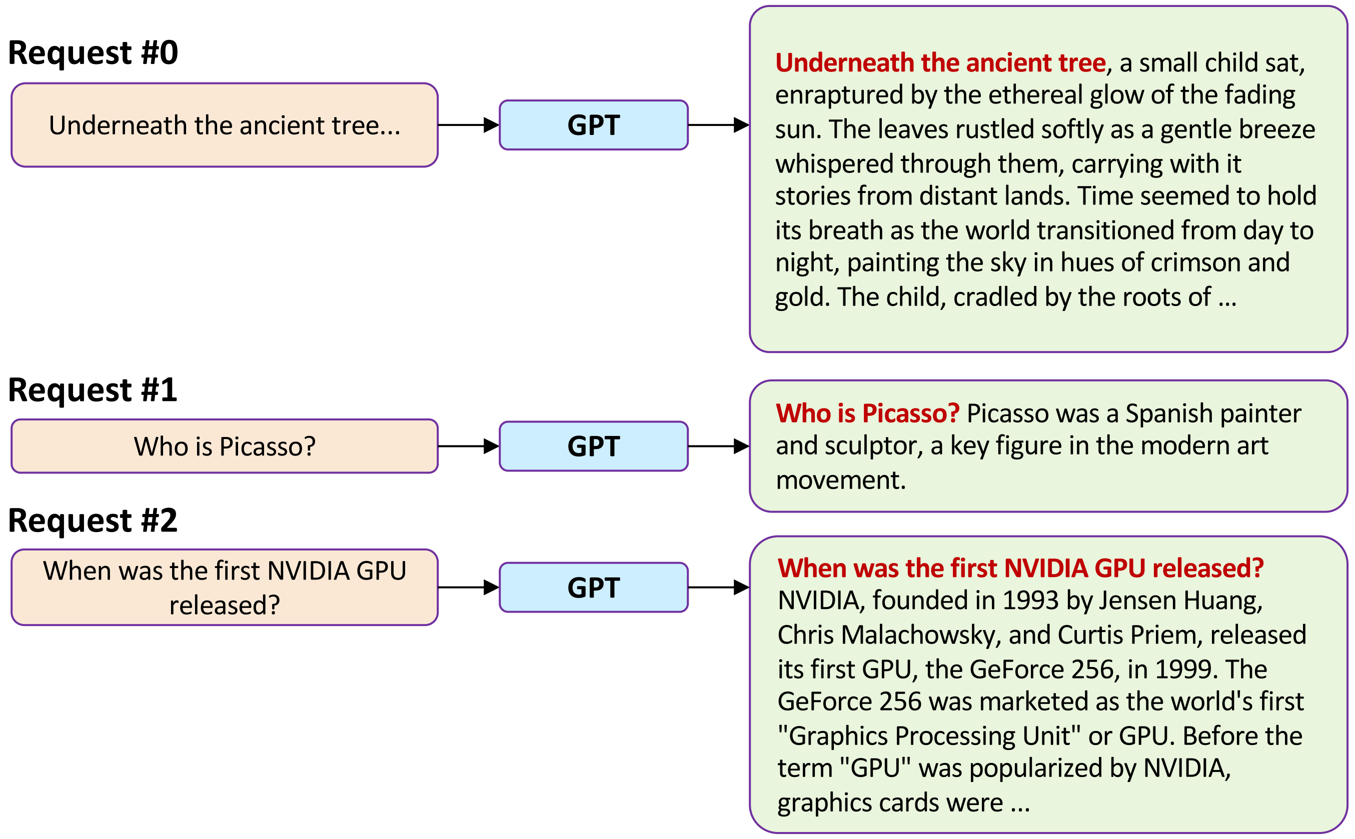
Large Language Models till now



Courtesy: Yang, Jingfeng, et al. "Harnessing the power of llms in practice: A survey on chatgpt and beyond." *arXiv preprint arXiv:2304.13712* (2023).

Enhancing Real-time Parallel Inference on Generative LLM

- GPT models generate responses sequentially, creating answers one word at a time, not in one go. We refer as an autoregressive process.
- Inference requests arrive randomly on the server side.
- Each request only handles a very small batch size.
- Inference on GPT models usually has highly variable lengths of answer.



Request #0

Underneath the ancient tree... → **GPT** →

Underneath the ancient tree, a small child sat, enraptured by the ethereal glow of the fading sun. The leaves rustled softly as a gentle breeze whispered through them, carrying with it stories from distant lands. Time seemed to hold its breath as the world transitioned from day to night, painting the sky in hues of crimson and gold. The child, cradled by the roots of ...

Request #1

Who is Picasso? → **GPT** →

Who is Picasso? Picasso was a Spanish painter and sculptor, a key figure in the modern art movement.

Request #2

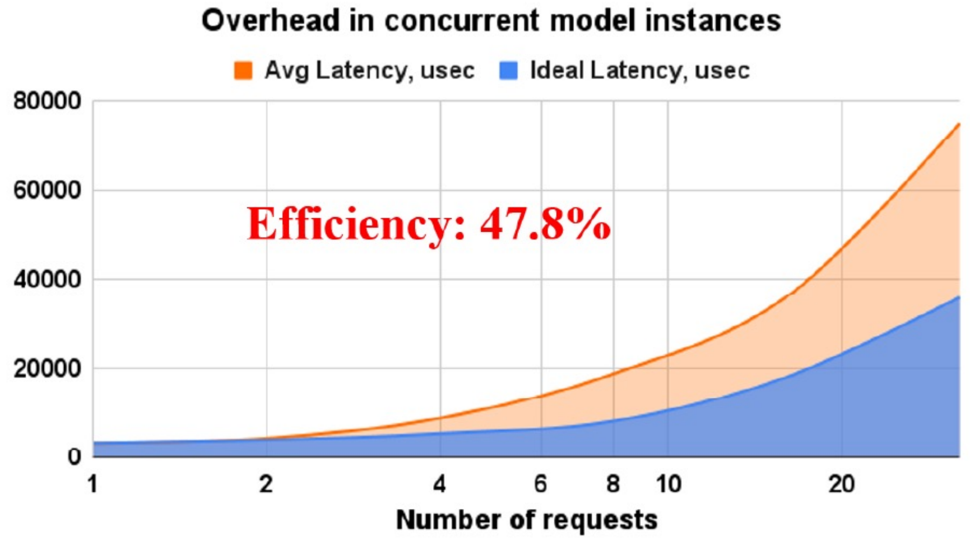
When was the first NVIDIA GPU released? → **GPT** →

When was the first NVIDIA GPU released? NVIDIA, founded in 1993 by Jensen Huang, Chris Malachowsky, and Curtis Priem, released its first GPU, the GeForce 256, in 1999. The GeForce 256 was marketed as the world's first "Graphics Processing Unit" or GPU. Before the term "GPU" was popularized by NVIDIA, graphics cards were ...

Examples

- Request #0 arrives at +1s, will take 8s to finish
- Request #1 arrives at +2.8s, will take 4s to finish
- Request #2 arrives at +6s, will take 11s to finish

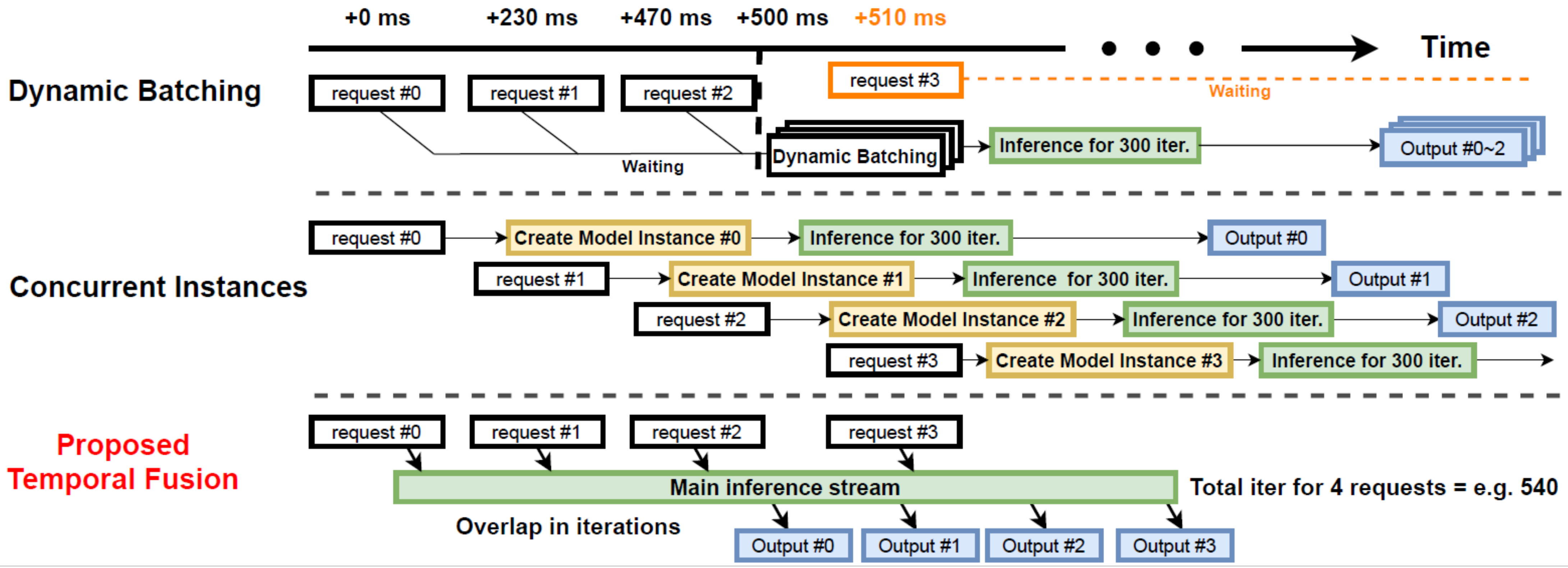
What is the best strategy to parallel inference these requests?



Inefficiency of existing solutions, severe overhead

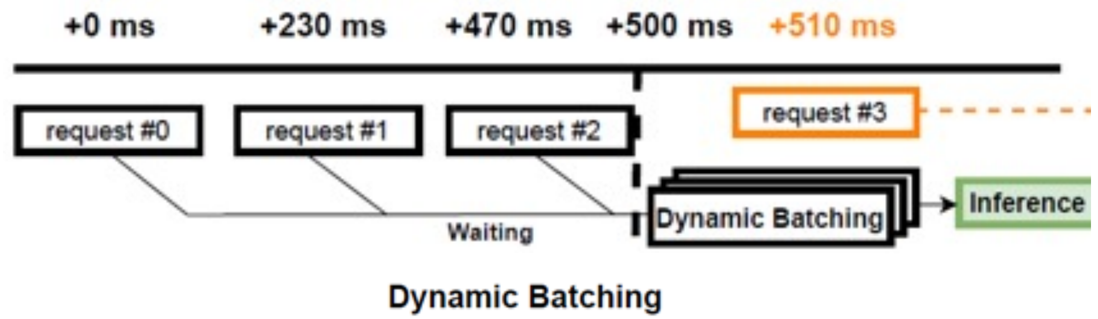
Existing Parallel Inference solutions

- Dynamic batching allows the server to wait within a time window (e.g. 500ms), requests that arrive within the time window will be packed together. When the time window is reached or the maximum requests are presented, the packed batch will be passed into the inference model for efficient processing.
- Concurrent instances allows the immediate launching of a new inference model instance once a request arrives, and the instance will only infer this request. For e.g., 32 parallel inference requests will require 32 model instances launched simultaneously.

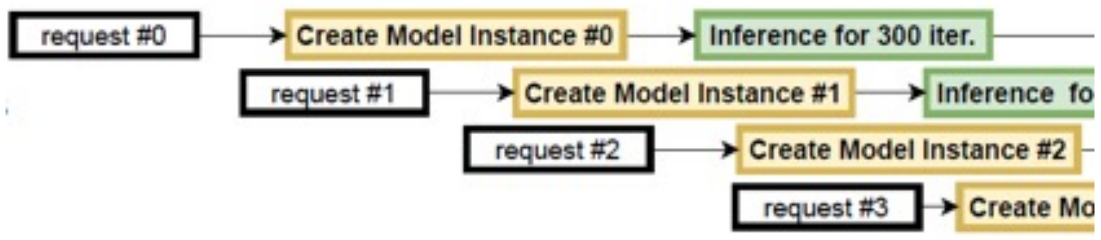


Existing Parallel Inference solutions -- Bottlenecks

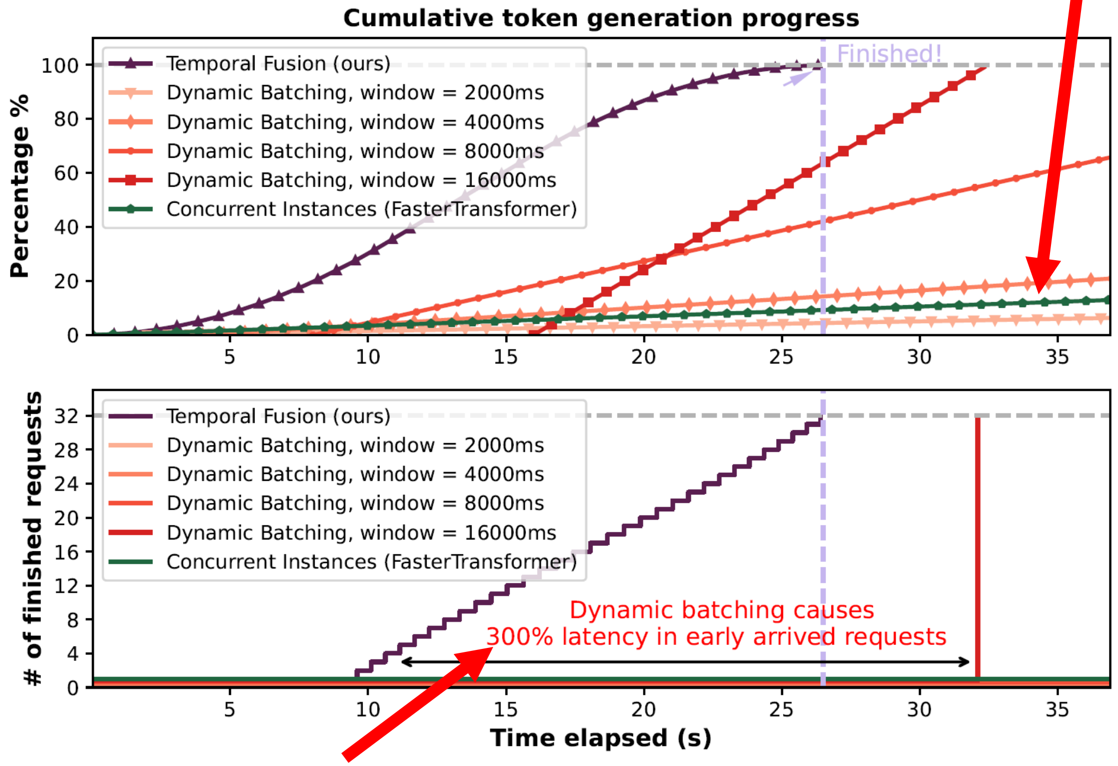
- Dynamic batching: Determining the time window can be heuristic and exhibits no pattern. For e.g., earlier requests will have to wait for the whole window until it can be processed. This significantly increases latency and prevents possible overlap of computation.
- Concurrent instance allows the immediate launching of a new inference model instance once a request arrives, and the instance will only infer this request. For e.g., 32 parallel inference requests will require 32 model instances launched simultaneously, causing severe resource/bandwidth contention and redundant kernel launching.



Dynamic Batching



Concurrent Instances

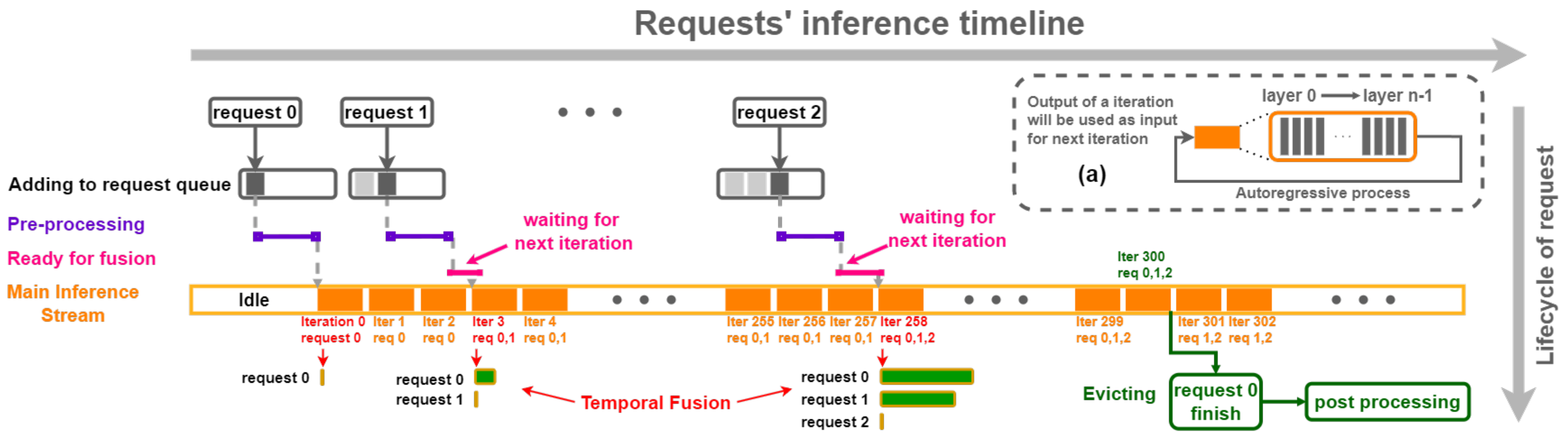


Presentation Outline

- Introduction to Autoregressive models
  - Deployment scenarios of inference Generative LLMs
  - Existing parallel inference methods
- Designs, implementations, and experiments
  - Temporal fusion of multiple random requests
  - Adaptive memory shuffle for creating contiguous memory
- Demo
- Summary

Flover -- a Temporal Fusion framework for LLM inference

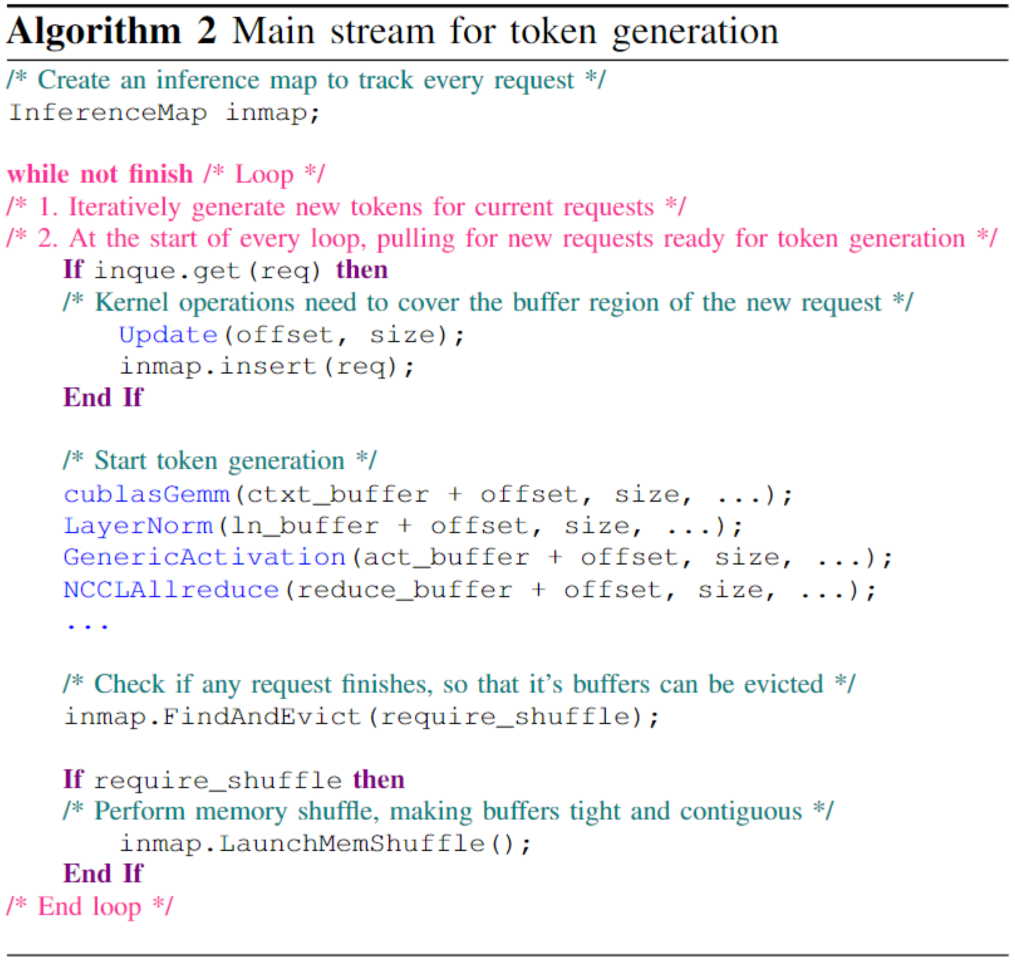
- Main contributions:
  - Promptly processes incoming requests eliminating the need for any batching or time window allocation
  - Avoids launching redundant model instances or kernel calls



Flover runs a main inference instance throughout the whole runtime, which can adaptively generate new tokens for any number of requests.

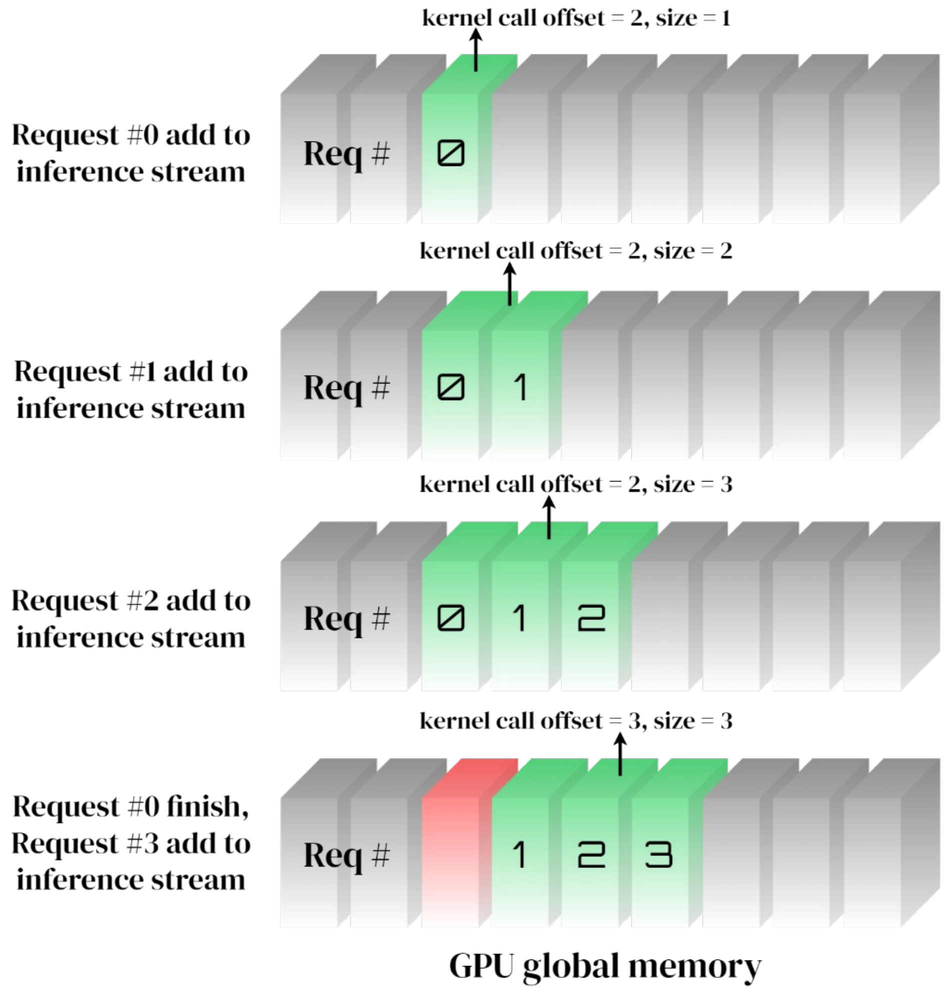
Designs – Combined kernel launch

- Generating tokens for different requests follows identical procedures, we do not need separate launching of kernels.
- We can simply update the buffer offset and size, at the beginning of every iteration when new requests are added to inference stream, then we initialize single GPU kernel calls to operate on the entire buffer.



Algorithm 2 Main stream for token generation

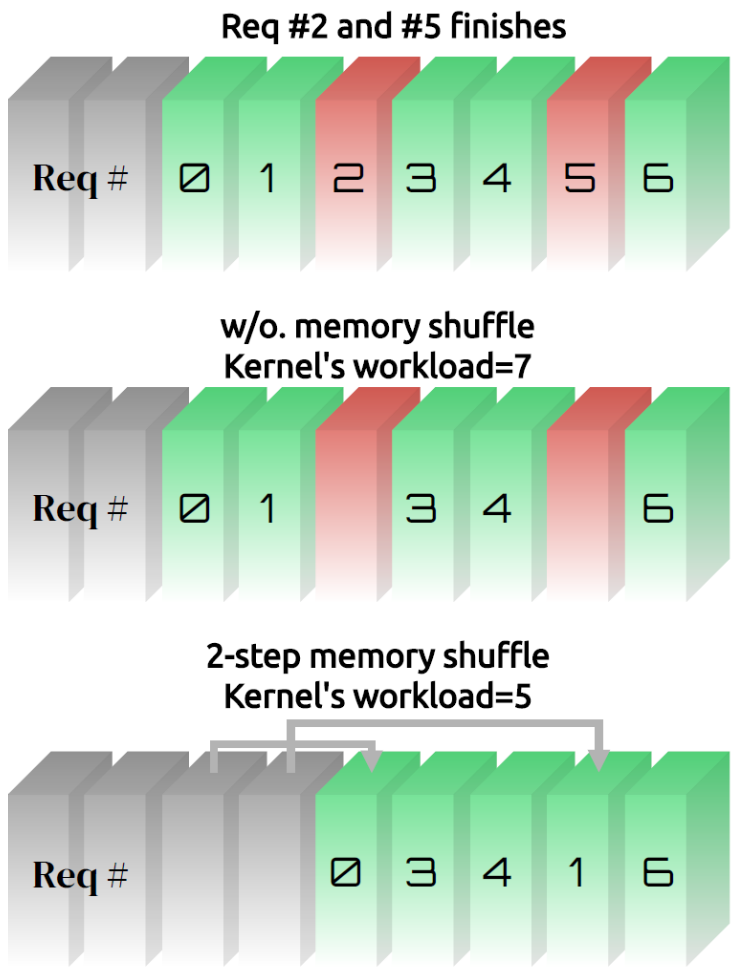
```
/* Create an inference map to track every request */
InferenceMap inmap;

while not finish /* Loop */
/* 1. Iteratively generate new tokens for current requests */
/* 2. At the start of every loop, pulling for new requests ready for token generation */
    If inque.get(req) then
    /* Kernel operations need to cover the buffer region of the new request */
        Update(offset, size);
        inmap.insert(req);
    End If

    /* Start token generation */
    cublasGemm(ctxt_buffer + offset, size, ...);
    LayerNorm(ln_buffer + offset, size, ...);
    GenericActivation(act_buffer + offset, size, ...);
    NCCLAllreduce(reduce_buffer + offset, size, ...);
    ...

    /* Check if any request finishes, so that it's buffers can be evicted */
    inmap.FindAndEvict(require_shuffle);

    If require_shuffle then
    /* Perform memory shuffle, making buffers tight and contiguous */
        inmap.LaunchMemShuffle();
    End If
/* End loop */
```


kernel call offset = 2, size = 1
Request #0 add to inference stream
Req #
0
kernel call offset = 2, size = 2
Request #1 add to inference stream
Req #
0
1
kernel call offset = 2, size = 3
Request #2 add to inference stream
Req #
0
1
2
kernel call offset = 3, size = 3
Request #0 finish, Request #3 add to inference stream
Req #
1
2
3
GPU global memory

Designs – Memory Shuffle for creating contiguous buffer

1. Inference requests for generative models differ in the maximum output lengths:
    - This can lead to memory bubbles leading to inefficient design
2. Flover tackles this by explicitly managing memory and performing an efficient memory shuffle:
    - The figure shows when request #2 and #5 finish, there will be bubbles in memory
    - Using an explicit 2-step shuffling design, Flover ensures that each kernel always works on a continuous buffer avoiding the overhead of page loading each chunk

```
/* Start token generation */
cublasGemm(ctxt_buffer + offset, size, ...);
LayerNorm(ln_buffer + offset, size, ...);
GenericActivation(act_buffer + offset, size, ...);
NCCLAllreduce(reduce_buffer + offset, size, ...);
...
```



```
Running requests
Req #0, mem_id = 2
Req #1, mem_id = 3
Req #2, mem_id = 4
Req #3, mem_id = 5
Req #4, mem_id = 6
Req #5, mem_id = 7
Req #6, mem_id = 8

Req #0, mem_id = 2
Req #1, mem_id = 3
Req #3, mem_id = 5
Req #4, mem_id = 6
Req #6, mem_id = 8
 buffer offset = 2
 buffer   size = 7
```

Shuffled to contiguous buffer

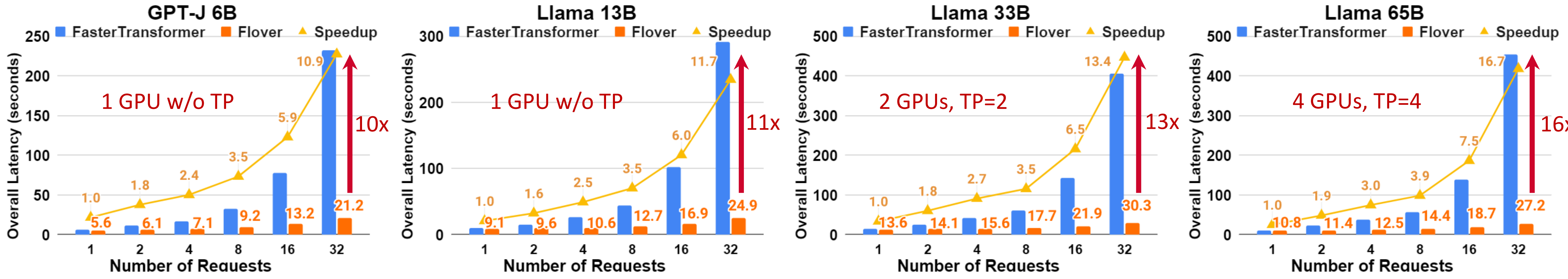
```
Req #0, mem_id = 4
Req #3, mem_id = 5
Req #4, mem_id = 6
Req #1, mem_id = 7
Req #6, mem_id = 8
 buffer offset = 4
 buffer   size = 5
```

Flover Implementation

- Software:
  - Based on NVIDIA FasterTransformer C++ codebase, which is one of the most widely used Triton backends and large language model (LLM) solutions.
  - For the following experiments, we use several famous language models --- GPT-J 6B, Llama 7B, Llama 13B, Llama 33B, and Llama 65B.
    - GPT-J 6B is officially supported in FasterTransformer.
    - Llama variants are provided and validated under https://github.com/NVIDIA/FasterTransformer/pull/575.
    - We use modular design that enables fast implementations of new GPT models.
  - Tensor parallelism is leveraged across GPUs.
  - MVAPICH2-GDR 2.3.7 is used for controlling and synchronizing different ranks.
  - NCCL 2.14.3 is used for collective communications.
- Hardware:
  - We conduct all experiments on NVIDIA A100-SXM 80GB GPUs with AMD EPYC 7763 64-Core Processor. Each computing node has 2 CPU sockets and 4 GPUs connected by NVLINK.

Experiment 1. Overall Latency of inference requests

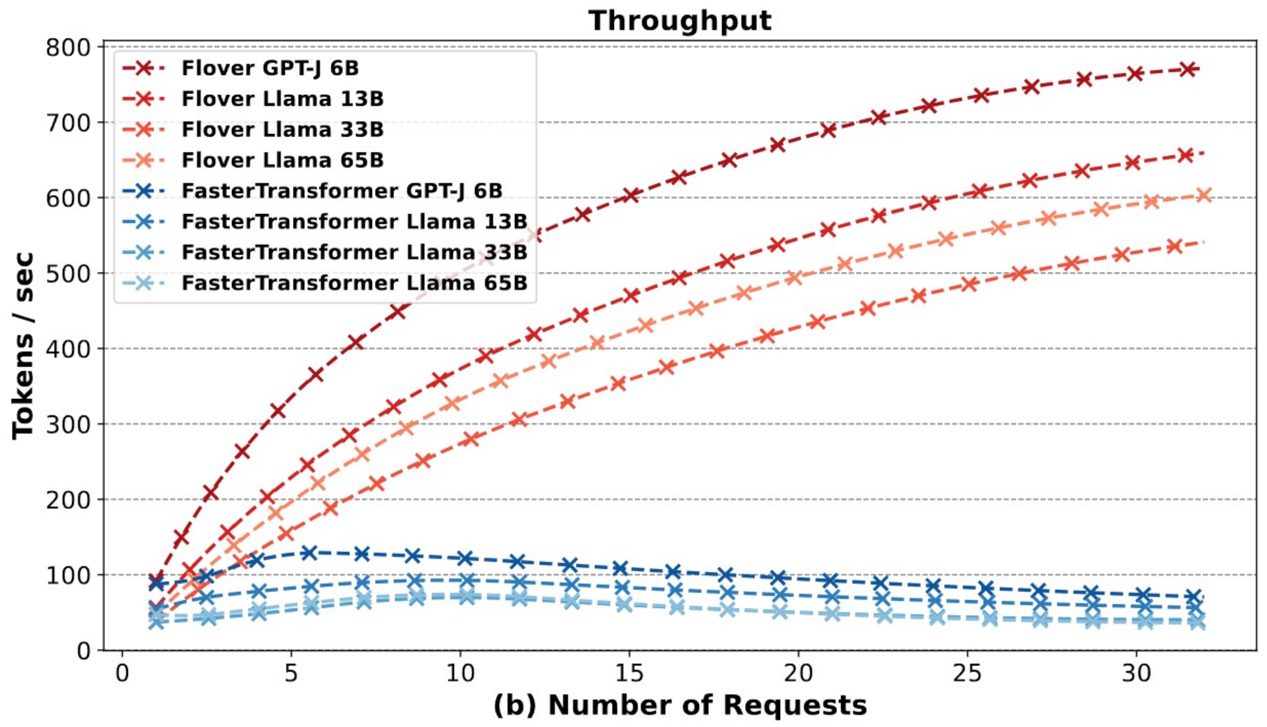
- This experiment quantifies the performance of Flover, while using temporal fusion, to process multiple requests in parallel:
  - We use a constant time interval of 500ms to study the parallel efficiency.
- Notice that for all models, the average inference latency for a single request is >> 500ms, therefore it leaves great potential for parallel acceleration.
- Flover achieves up to 16.7x speedup in latency compared to FasterTransformer.



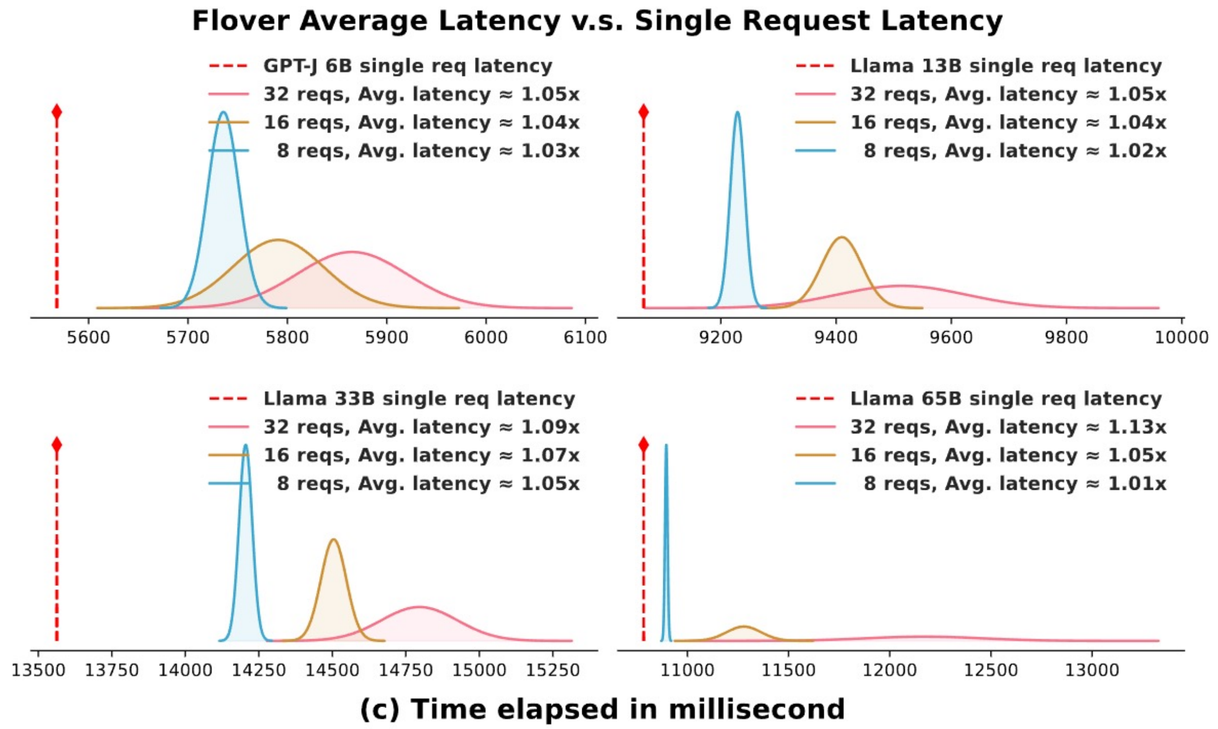
(a). Parallel inference on different models. We measure the overall time spent on parallel inference **1, 2, 4, 8, 16, 32** requests.

Experiment 2. Throughput/Latency in Token Generation

- Flover efficiently utilizes GPUs, as shown in the Fig (b) for throughput, by enabling individual kernels to operate on larger contiguous buffer pieces. This is similar to increasing the batch size from 1 to 32 leading to near linear speedups.
- Does processing requests in parallel slowdown the average inference latency for each request? This is important to understand as this directly affects the user experience of their request. Fig (c) shows that with 8 and 32 parallel requests, the slowdown is limited to 3% and 8% respectively.



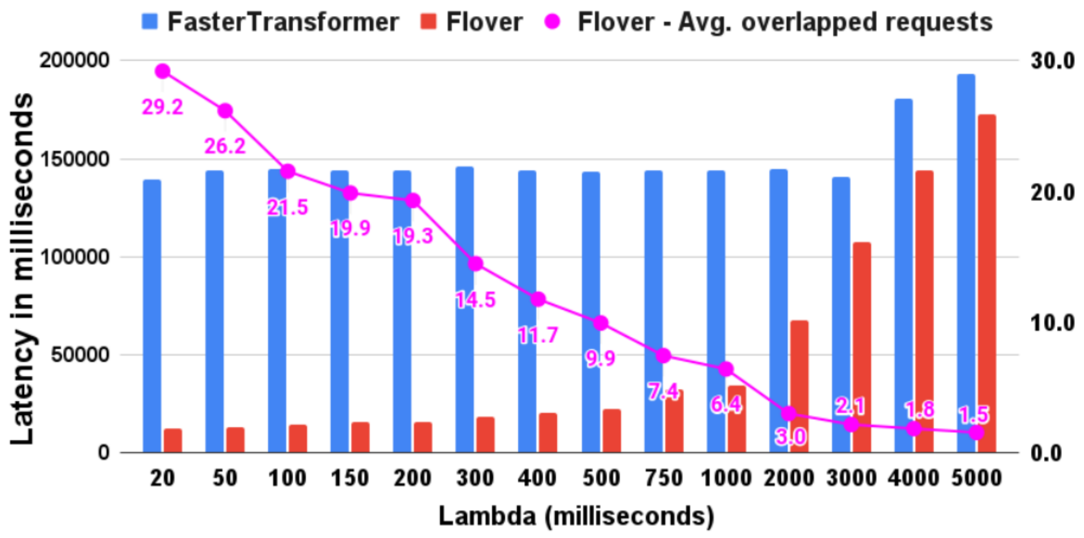
(b) Number of Requests



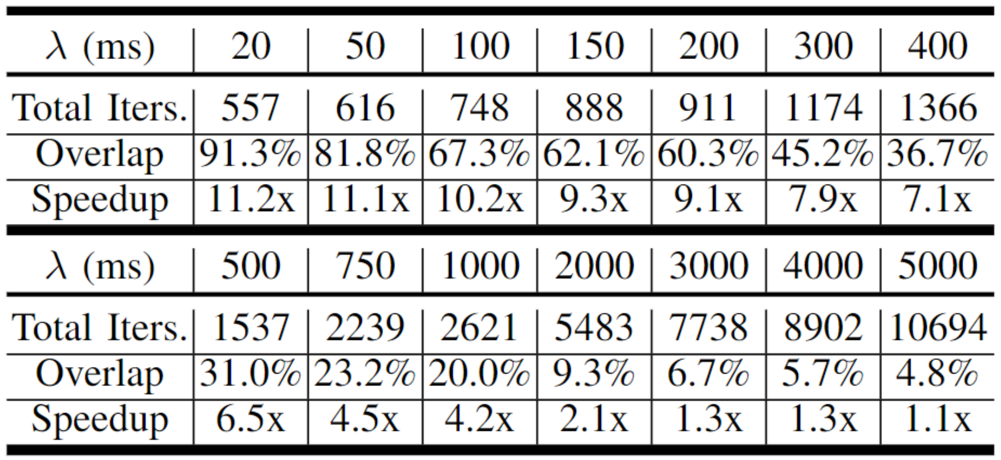
(c) Time elapsed in millisecond

Experiment 3. Random Arrival of Requests

- To depict a realistic setting, this experiment models the arrival of inference requests as a Poisson process for the inference requests.
  - Here we use 32 requests. Time intervals between requests are randomly sampled from the exponential distribution with different λ.
  - A single request requires 512 iterations, which take 5800ms on the inference server.
  - With relatively small λ, Flover achieves 11.2x speedup against FasterTransformer. As 91.3% of all requests are overlapped and hence processed in parallel with single kernel call.
  - When increasing λ, the temporal overlapped requests are fewer, therefore, less space for optimizing the parallel



(a)



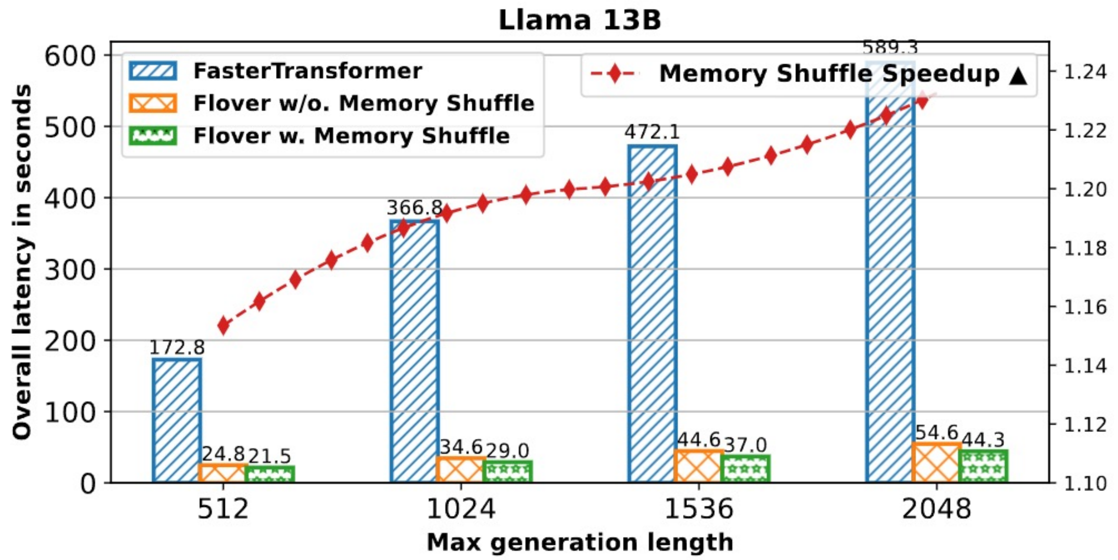
λ (ms)	20	50	100	150	200	300	400
Total Iters.	557	616	748	888	911	1174	1366
Overlap	91.3%	81.8%	67.3%	62.1%	60.3%	45.2%	36.7%
Speedup	11.2x	11.1x	10.2x	9.3x	9.1x	7.9x	7.1x

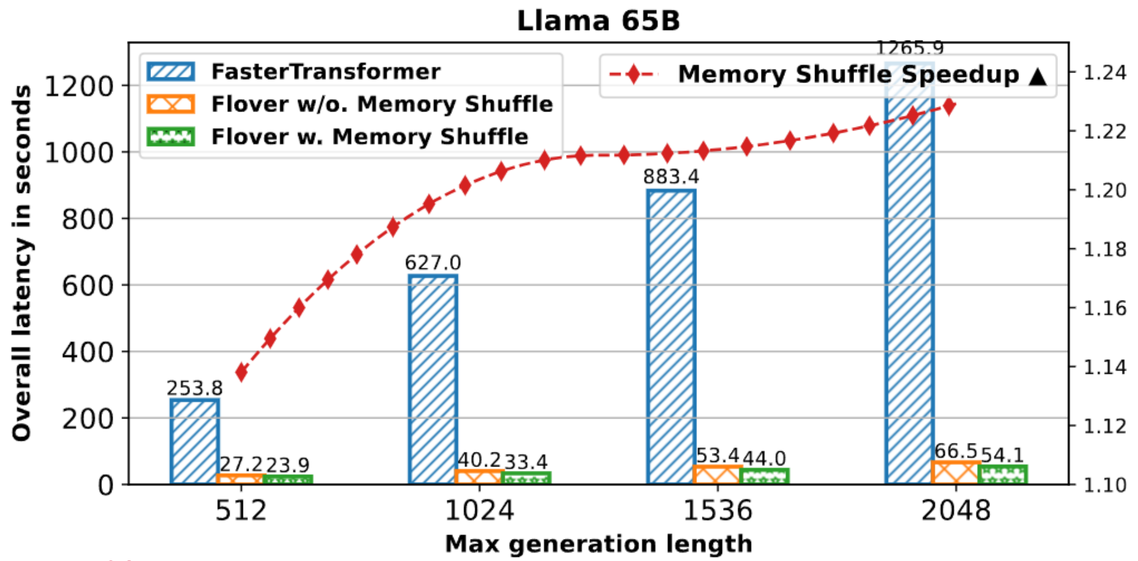
λ (ms)	500	750	1000	2000	3000	4000	5000
Total Iters.	1537	2239	2621	5483	7738	8902	10694
Overlap	31.0%	23.2%	20.0%	9.3%	6.7%	5.7%	4.8%
Speedup	6.5x	4.5x	4.2x	2.1x	1.3x	1.3x	1.1x

(b)

Experiment 4. Memory Shuffle for Non-uniform Requests

- In real-world scenarios, requests from different users vary drastically in the total number of iterations (another random variable):
  - This randomness can cause variability in the length of the generated sequences, making it hard to fit a simple distribution.
  - To evaluate this in the worst-case, we adopt a uniform distribution $U(a, b)$ to model total iterations:
    - Lower bound a is set to 128, and we vary upper bound $b = \{512, 1024, 1536, 2048\}$.
    - If a request finishes, we evict it from the memory, and perform memory shuffle to reorganize the requests buffer, guaranteeing a contiguous memory region for GPU kernel to operate on.

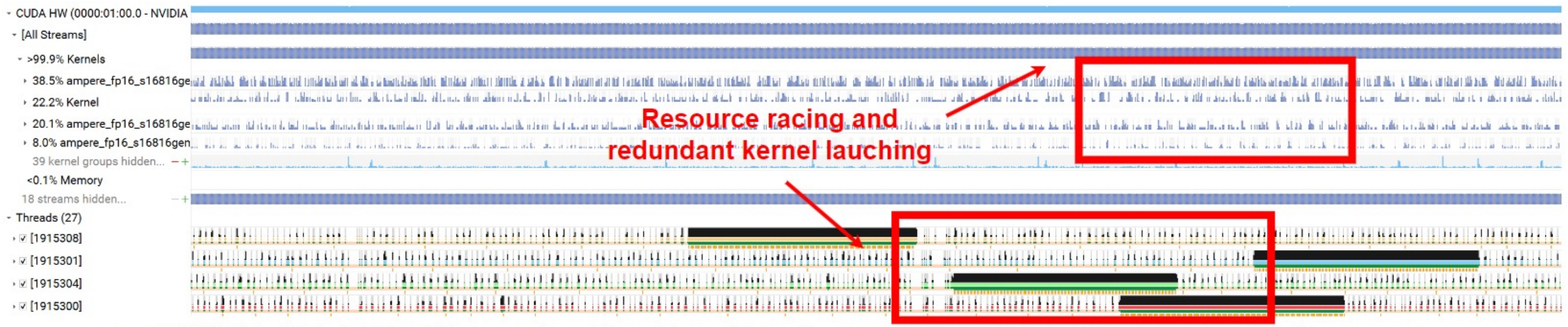




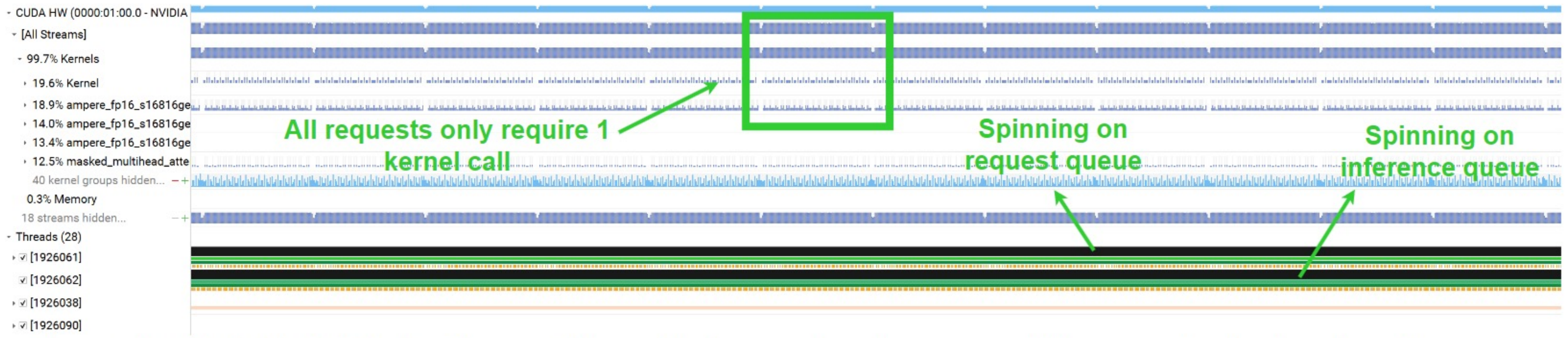
Memory shuffle further brings a 20% speedup!

Experiment 5. Hardware Profiling

- The CUDA profiling shows uniform resource usage for Flover due to 1 thread and 1 instance issuing kernel calls
  - Each segment in Fig (b) represents one iteration in inference


CUDA HW (0000:01:00.0 - NVIDIA
[All Streams]
>99.9% Kernels
38.5% ampere_fp16_s16816ge
22.2% Kernel
20.1% ampere_fp16_s16816ge
8.0% ampere_fp16_s16816gen
39 kernel groups hidden...
<0.1% Memory
18 streams hidden...
Threads (27)
[1915308]
[1915301]
[1915304]
[1915300]
Resource racing and redundant kernel lauching

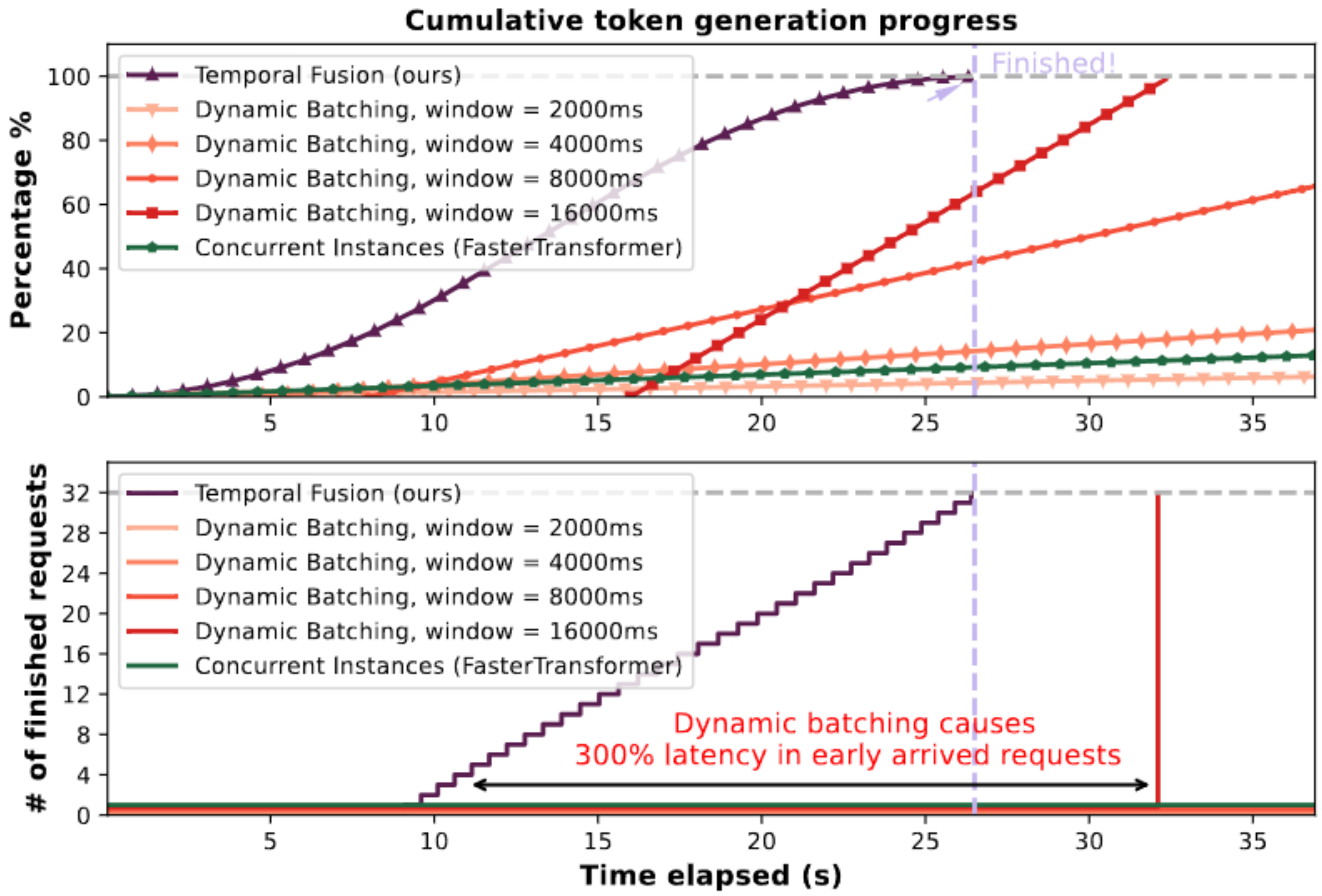
(a) NVIDIA Nsight Profiling on baseline FasterTransformer. All model instances compete for resources.


CUDA HW (0000:01:00.0 - NVIDIA
[All Streams]
99.7% Kernels
19.6% Kernel
18.9% ampere_fp16_s16816ge
14.0% ampere_fp16_s16816ge
13.4% ampere_fp16_s16816ge
12.5% masked_multihead_atte
40 kernel groups hidden...
0.3% Memory
18 streams hidden...
Threads (28)
[1926061]
[1926062]
[1926038]
[1926090]
All requests only require 1 kernel call
Spinning on request queue
Spinning on inference queue

(b) NVIDIA Nsight Profiling on Flover. Each green box denotes parallel generating 1 token for all requests.

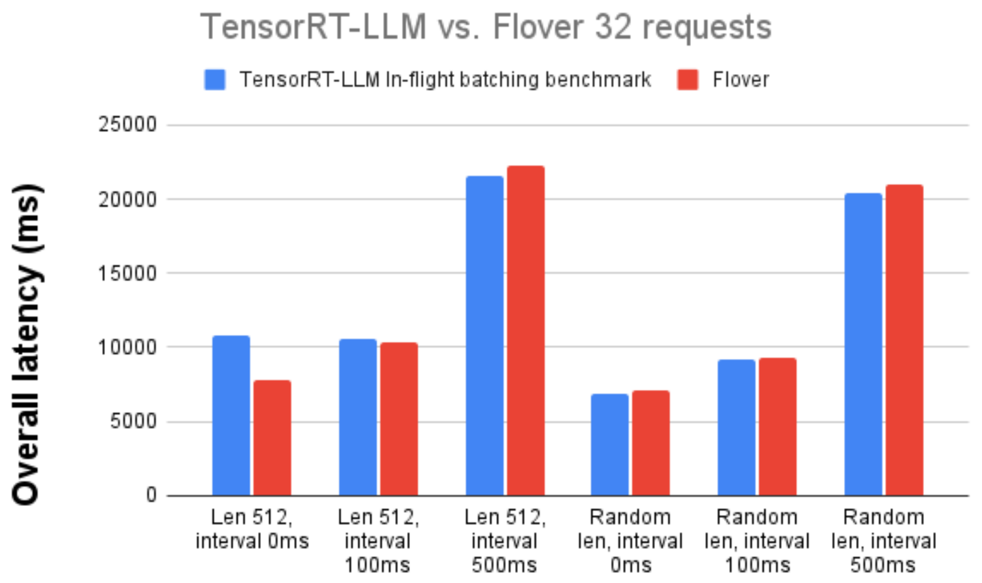
Experiment 6. A Comprehensive Comparison

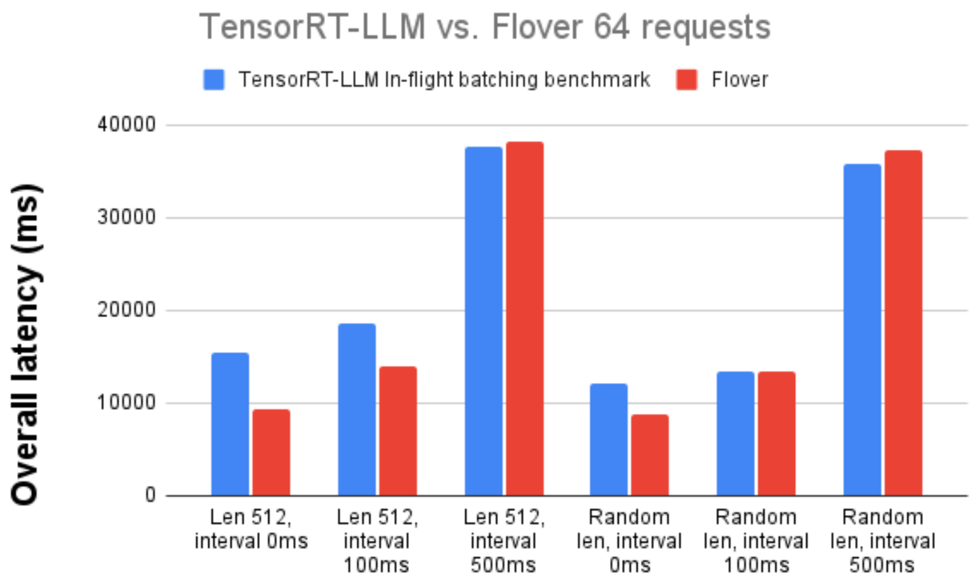
- Flover maintains both scalability and responsiveness while dealing with heavy load scenarios.
  - The figure shows the cumulative token generation progress of processing 32 requests in parallel.



Experiment 7. Compare to the latest TensorRT-LLM

- We compare the latest TensorRT-LLM commit `d8ebeee` with the released Flover.
  - We use the in-flight batching benchmark to evaluate the performance of TensorRT-LLM
- We test the following combinations on LlaMA 7B:
  - Number of requests: a) 32, b) 64
  - Generation length of each request: a) 512, b) random sample from a uniform distribution of $U(128, 512)$.
  - Interval between requests:
    - 0 ms (this is the default setting in TensorRT-LLM in-flight batching benchmark)
    - 100 ms and 500 ms, we modify the gptManagerBenchmark.cpp by adding the following after Line.512:
      - `std::this_thread::sleep_for(std::chrono::milliseconds(100)); // 100, 500, customized intervals`

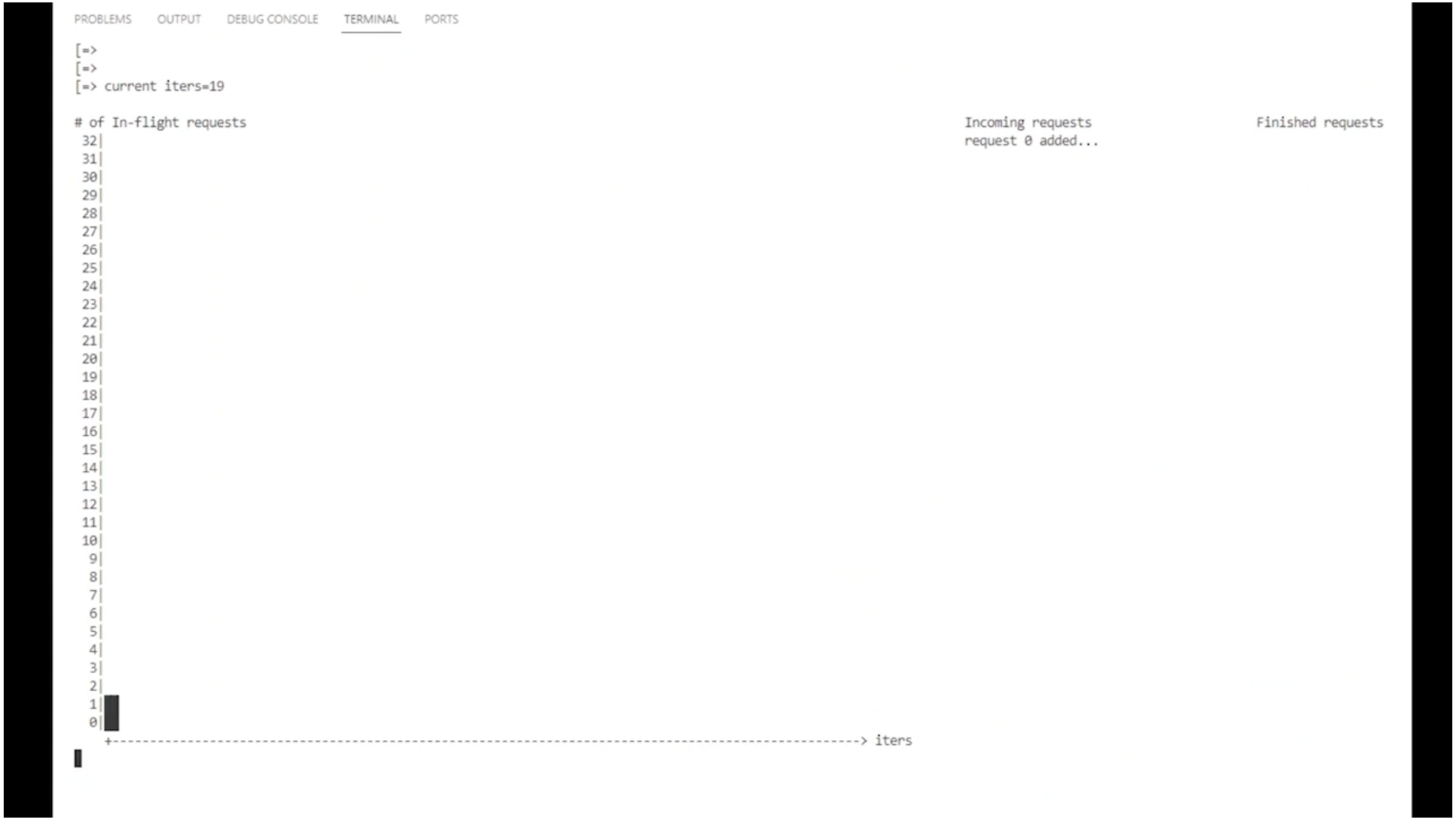




Presentation Outline

- Introduction to Autoregressive models
  - Deployment scenarios of inference Generative LLMs
  - Existing parallel inference methods
- Designs, implementations, and experiments
  - Temporal fusion of multiple random requests
  - Adaptive memory shuffle for creating contiguous memory
- Demo
- Summary

Demo: Parallel Inference 32 requests with random generation lengths on LLaMA 7B



```
[=>
[=>
[=> current iters=19

# of In-flight requests                                                                    Incoming requests          Finished requests
 32|                                                                                       request 0 added...
 31|
 30|
 29|
 28|
 27|
 26|
 25|
 24|
 23|
 22|
 21|
 20|
 19|
 18|
 17|
 16|
 15|
 14|
 13|
 12|
 11|
 10|
  9|
  8|
  7|
  6|
  5|
  4|
  3|
  2|
  1|█
  0|█
    +------------------------------------------------------------------------------------> iters
```

Summary

- This talk presented Flover that leverages temporal parallelism of autoregressive models:
  - Promptly processes requests avoiding the need for any batching or time window allocation
  - Avoids launching redundant model instances or kernel calls utilizing efficient memory management
- Flover achieves optimal performance on single GPU and distributed inference scenarios, ensuring robustness and scalability in diverse autoregressive model inference:
  - The talk presented performance numbers compared to FasterTransformer and TensorRT-LLM
- Flover is available as an open-source software:
  - https://github.com/OSU-Nowlab/Flover
  - Comments and contributions are welcome
- In the future, we plan to enhance Flover to support emerging language model architectures

Thank You!

{shafi.16}@osu.edu

X *Follow us on*

https://twitter.com/mvapich

Network-Based Computing Laboratory

http://nowlab.cse.ohio-state.edu/

MVAPICH MPI, PGAS and Hybrid MPI+PGAS Library

The High-Performance MPI/PGAS Project

http://mvapich.cse.ohio-state.edu/

HiBD High-Performance Big Data

The High-Performance Big Data Project

http://hibd.cse.ohio-state.edu/

HiDL *High-Performance Deep Learning*

The High-Performance Deep Learning Project

http://hidl.cse.ohio-state.edu/