西北工業大學
NORTHWESTERN POLYTECHNICAL UNIVERSITY

kvcached

Prism: Unleashing GPU Sharing for Cost-Efficient Multi-LLM Serving

[1]UCLA [2]UC Berkeley [3]Rice University [4]Harvard University [5]CMU [6]Intel [7]Stanford University [8]LMSYS

Presented by Mingxuan Liu, Northwestern Polytechnical University

January 6, 2026

arXiv > cs > arXiv:2505.04021

Background: GPU Multiplexing

- **(Traditional) GPU Multiplexing**
  - NVIDIA: Time slicing, MPS, stream Priority, and MIG
  - Papers: REEF@OSDI22, TGS@NSDI23, Orion@Eurosys24, LithOS@SOSP25
  - However, they just focus on compute resouce sharing.
- **Systems serving multiple LLMs (Serverless Style)**
  - Optimized model loading: ServerlessLLM, HydraServe@NSDI26
  - DRAM preloading: DeepServe@ATC25
  - Fine-grained autoscaling: BlitzScale@OSDI25, Aegaeon@SOSP25
  - However, the goal is to reduce cold start latency.

Background: CUDA VMM for LLM

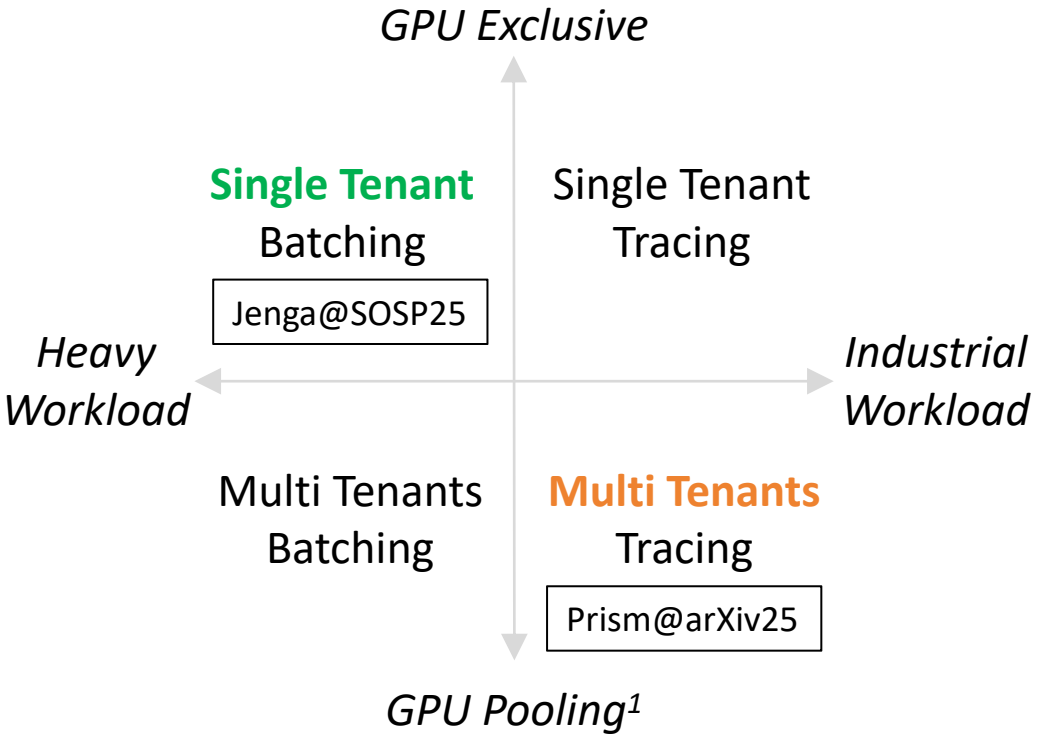
西北工業大學 NORTHWESTERN POLYTECHNICAL UNIVERSITY

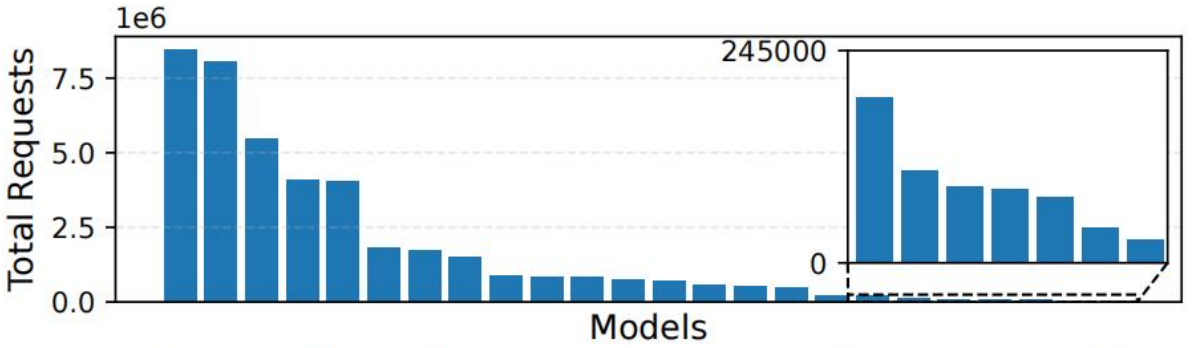
- **Virtual Memory in OS Kernel**
  - Isolation, mem. expansion like *mmap*, simple programming, IPC like *shm*, etc.
- **CUDA VMM API[1] (CUDA 10.2, 2019)**
  - Core API: *cuMemAddressReserve*, *cuMemCreate* and *cuMemMap*, etc.
  - Example: vectorAddMMAP[2]
    - mem_ptr is distributed across multiple GPU devices
- **CUDA VMM-based LLM**
  - PyTorch Expandable Segments[3] (2023)
  - vAttention@arXiv24, vTensor@arXiv24/GMLake@ASPLOS24
    - However, just improve kernel efficiency for serving a single model
  - vLLM sleep mode: Offload model weights to CPU Mem.
    - sleep()
    - wake_up()

1. https://developer.nvidia.com/blog/introducing-low-level-gpu-virtual-memory-management/
2. https://github.com/NVIDIA/cuda-samples/tree/master/Samples/0_Introduction/vectorAddMMAP
3. Expandable blocks in allocator. https://github.com/pytorch/pytorch/pull/96995

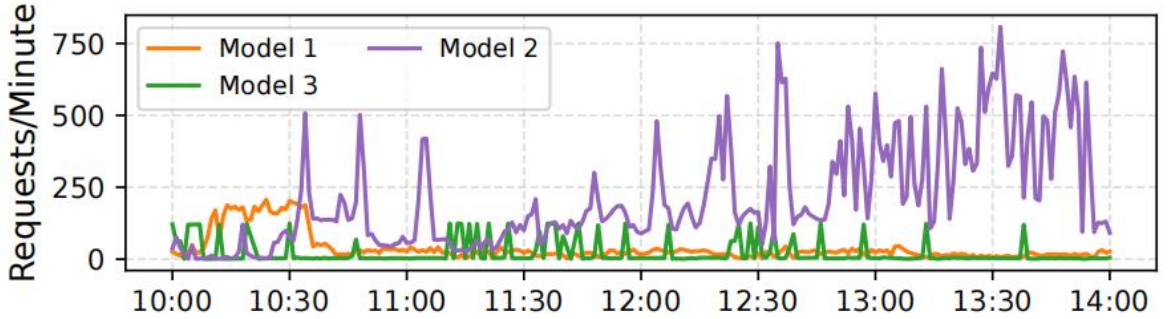
Background: Metrics of LLM Serving

- **SLO Attainment %: TTFT and TPOT**
- **Goal**: **Max{*Thpt* w/ or w/o SLO Attainment} + Min{*Mem wasting*}**





(a) Long-tail model popularity over a 4-month period.



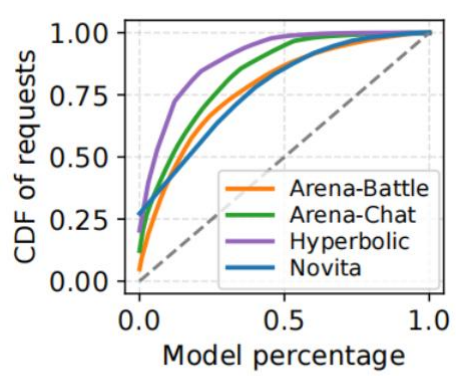
(b) Request rates over a 4-hour period in a day.

1. GPU Pooling can be divided into two types roughly: GPU Multiplexing or GPU Auto-scaling (Aegaeon@SOSP25)

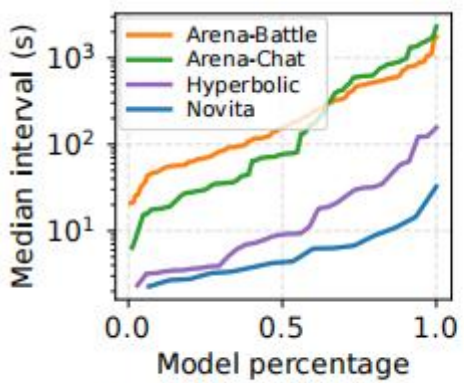
Observation & Insights

- I#1: Diverse latency SLOs
- I#2: Multiple low-demand models can space share GPUs
- I#3: No fixed sharing policies work well across Industrial workload.
- I#4: The frequent long idle periods allow time sharing:
  - evicting idle models from GPUs and reloading upon new request arrivals.

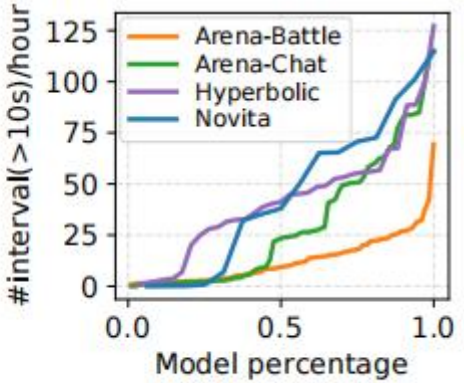
Trace name	Service provider	# models	Time span
Hyperbolic	Hyperbolic [26]	24	4 months
Novita	Novita AI [4]	16	1 month
Arena-Battle	Chatbot Arena [14]	129	16 months
Arena-Chat	Chatbot Arena [14]	84	11 days



(a) Long-tail model popularity.



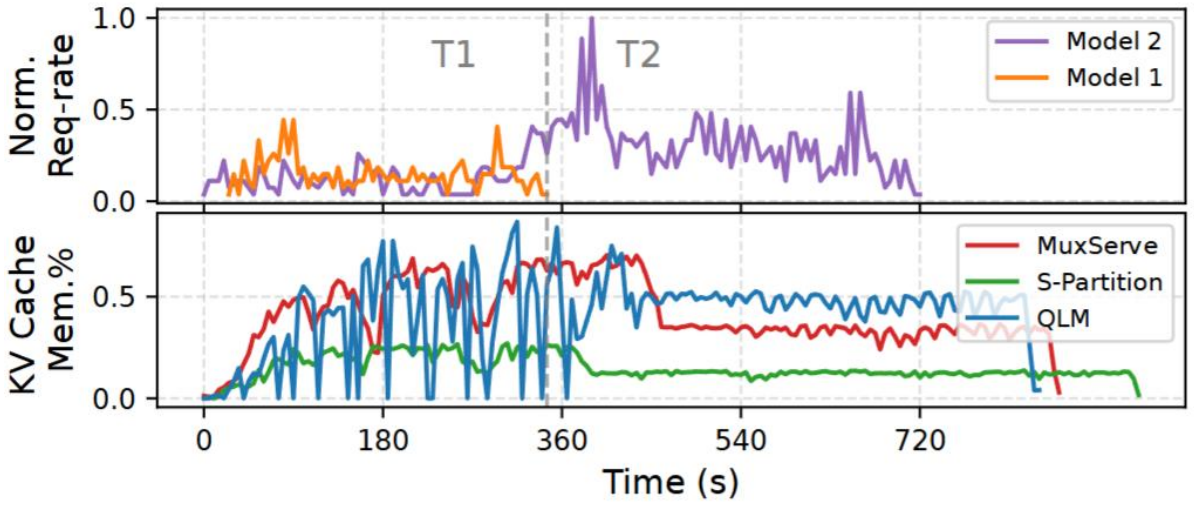
(b) Median request interval.



(c) Number of intervals per hour.

A detailed analysis of four production multi-LLM serving traces.

Limitations of Existing Approaches



	Dedicated	S-Pa.	Muxs.	QLM	Ours
Space sharing by model colocation	✗	✓	✓	✗	✓
Time sharing by model swapping	✗	✗	✗	✓	✓
Runtime sharing policy adaptation	✗	✗	✗	✗	✓
Cost saving & meeting SLOs	✗	✗	✗	✗	✓
# GPUs needed for 8 models	8	7	5	8	2
SLO attainment with 2 GPUs	-	39%	51%	45%	99%

Existing systems **allocate memory statically** or **use fixed sharing policies (Space-sharing or Time-sharing)**, lacking the flexibility to adapt to workload fluctuations at runtime.

- **S-Partition (MIG[1])**: Hardware-based space sharing, casuing the lowest mem. %
- **QLM**: Time sharing by model swapping, but unstable
- **MuxServe**: based on offline profiling, cannot adjust based on workload changes.
  - Lack model evication: when a model becomes idle, it continues occupying memory.

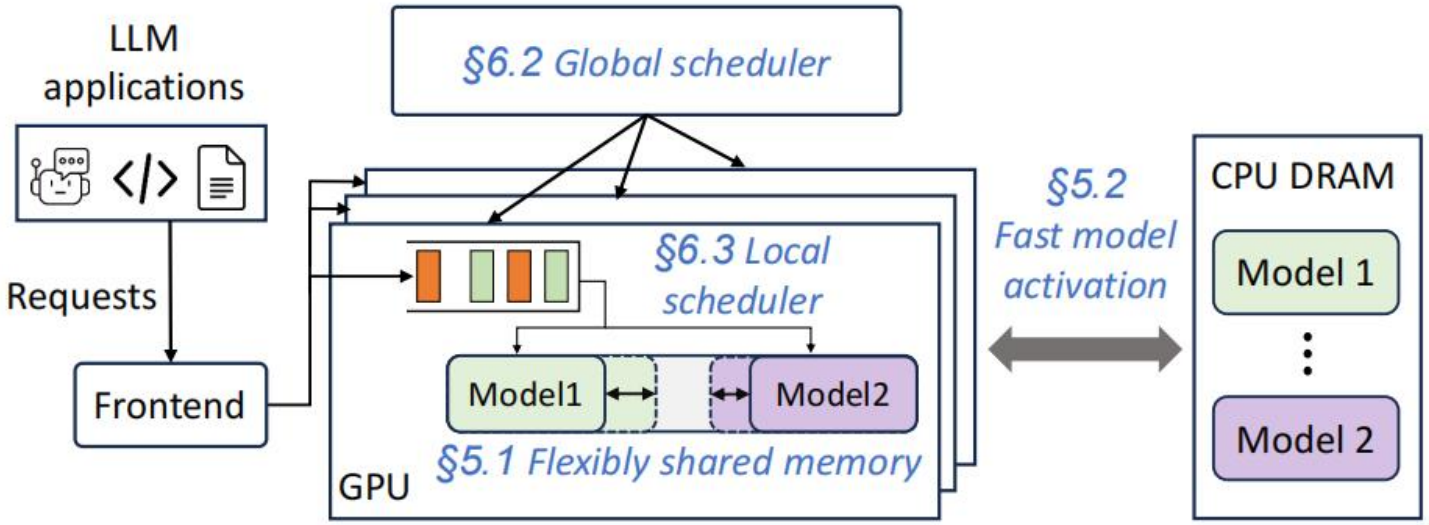
1. I guess.

Opportunity: Space-sharing + Time-sharing

- **Goal: Dynamically adjust memory allocation facing various workload**
- **Flexibly combine space and time sharing**
  - low-demand models share a GPU during steady periods (space sharing)
  - When one becomes idle, it can be swapping (time sharing)
- **Challenges**:
  - **C#1: How to enable flexible cross-model memory coordination?**
    - PagedAttention manages KV Cache within **pre-allocated memory**.
    - Requiring dynamic memory redistribution across multiple models.
  - **C#2: How to coordinate memory allocation to maximize SLO attainment?**
    - Scheduling Problem

Overview of Prism

- **Flexible Cross-Model Memory Coordination**
  - kvcaced: a shim layer for on-demand memory allocation
  - Fast model activation
- **Memory Sharing to max SLO Att.: a two-level resource scheduling**
  - Global Model Placement Scheduling (Alg. 1)
  - GPU-Local Request Scheduling (Alg. 2)



Cross-Model Memory Coordination

西北工業大學 NORTHWESTERN POLYTECHNICAL UNIVERSITY

- **PagedAttention**: Static memory allocation & cannot support multiple models
- **KVCached**: A shim layer for on-demand cross-model memory allocation
  - Only **virtual memory** is reserved when LLM Inference Engine startups.
  - **Physical memory** is allocated and mapped on demand
  - Implemented as a **runtime library**
    - Provides APIs like alloc_kvcache and free_kvcachefor KV cache management
    - With minimal code changes (e.g., ~20 lines in SGLang)

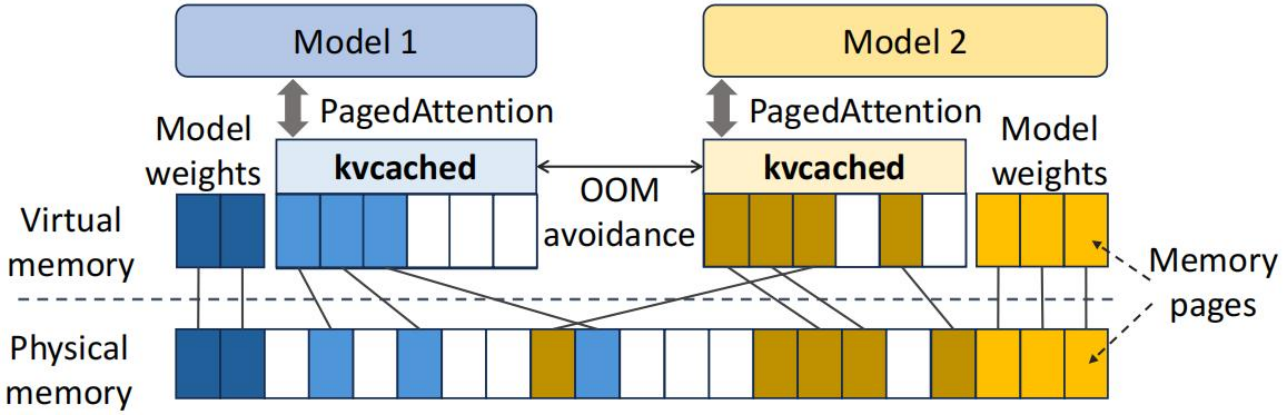


Figure 6: Flexible memory sharing in Prism.

APIs for inference engines	Corresponding kvcached functions
alloc_kvcache (size, shape)	alloc_virtual_tensor (size, shape)
free_kvcache (size, shape)	free_virtual_tensor (size, shape)
alloc_kv (num_tokens)	map (tensor, offset)
free_kv ([ids_to_free])	unmap (tensor, offset)

Table 3: APIs and functions provided by Prism's kvcached.

Fast Model Activation

- **Existing model swapping**: often within just a few seconds, too slow!
- **Reusable engine pools**
  - Maintains an engine pool on each GPU
  - Engines are pre-initialized with virtual address space and distributed contexts
  - Model activation: selects an available engine from the pool
  - Model eviction: physical memory is released, virtual address space is returned to the engine pool
- **Parallel model weight loading**
  - Chunking model weights into smaller segments
  - Loading them in parallel across multiple GPUs on the same node
  - Aggregating them to the target GPU via high-speed NVLink interconnects

Two-Level Demand-Aware Scheduling

- **How to coordinate memory allocation to maximize SLO attainment?**
  - The Scheduling Problem
- **Level 1 Global Model Placement Scheduling**
  - KV pressure ratio (KVPR): the degree of KV cache pressure on a GPU
  - Assign the model to the GPU with the lowest KVPR
- **Level 2 GPU-Local Request Scheduling**
  - GPU-level shared request queues
    - replace independent queues for each engine
    - avoid memory contention between co-located models
  - Priority-based admission control
    - employs the Moore-Hodgson algorithm to sort requests by deadline
    - prioritizing high-priority requests (such as those with urgent TTFT SLOs)

Implementation

- **The backend for LLM Serving**
  - **kvcached library**: provides standard KV cache allocation APIs
  - Extended SGLang (only 22 LOCs)
  - Configured MPS to 100% per model

- **The frontend**
  - a Redis queue to cache requests
  - GPU-Local scheduler (a **Python** process)
    - For TP Models, runs only on 1st rank
  - Global scheduler (a **Python** process)
    - Communicates with engines by ZeroMQ

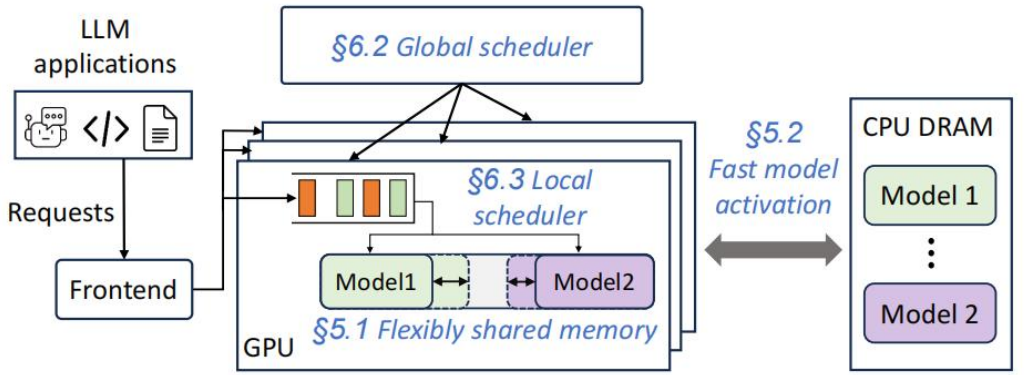


Figure 5: The system architecture and design overview of Prism.

Evaluation: Setup

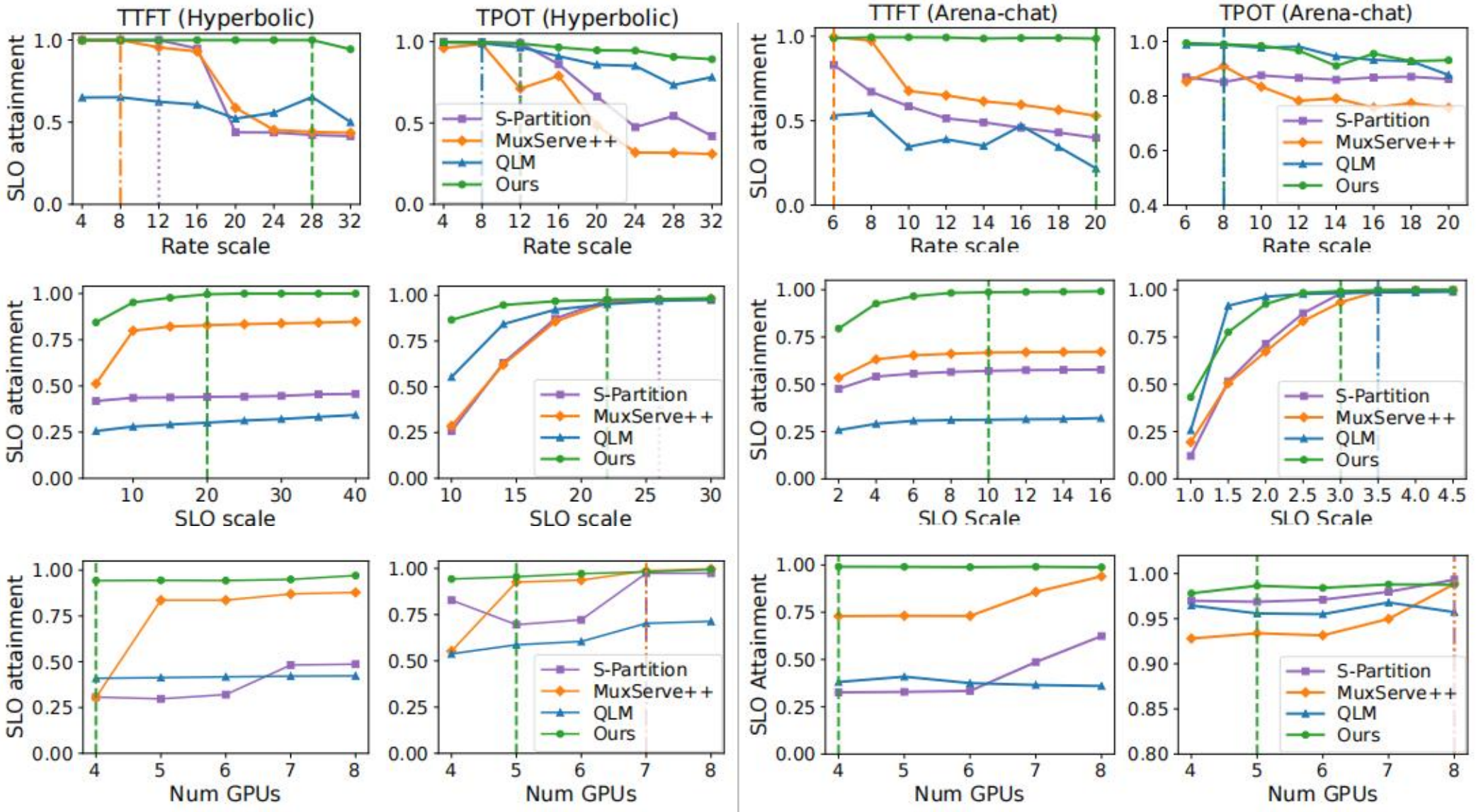
- **Test bed:** 4 GPU servers with {8 H100-80GB GPUs/NVLINK} connected with 100Gbps Ethernet network
- **Traces and models:** Hyperbolic and Arena-Chat, which also be scaled to simulate a variety of scenarios.

Model size	1B-3B	4B-8B	9B-30B	31B-70B
#LLMs	43	8	3	4

Total 58 Models used in evaluation.

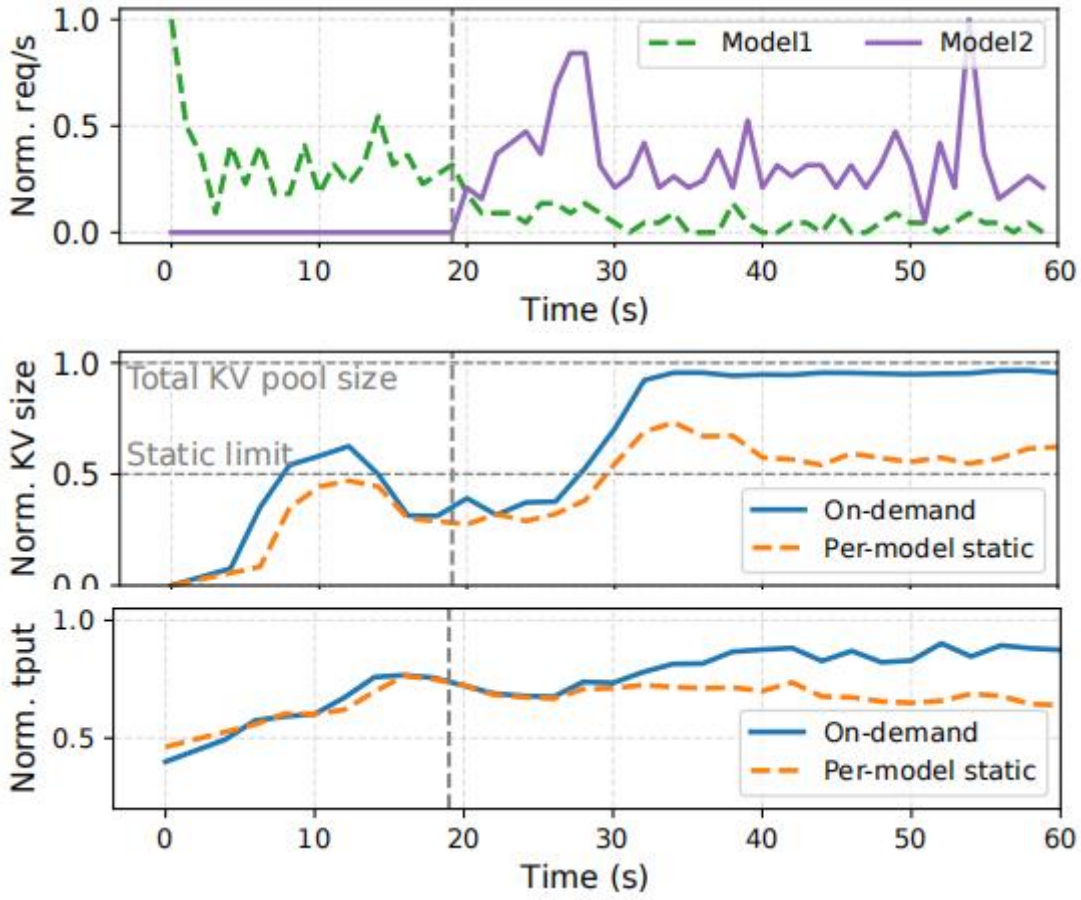
- **Metric:** TTFT/TPOT SLO Attainment
- **Baseline:**
  - **Static Partition** - NVIDIA MIG
  - **MuxServe++** - ported MuxServe from vLLM to SGLang
  - **QLM** - Model swapping-based time sharing

E1 SLO attainment



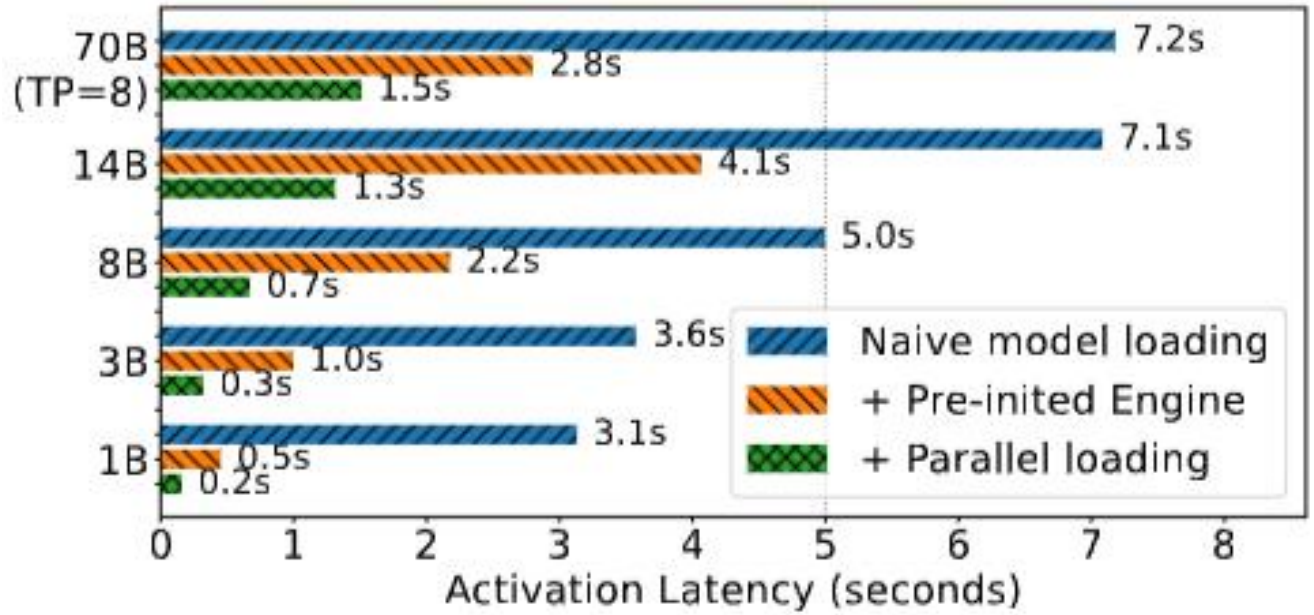
E2-1 Cross-model memory coordination

- a simplified **two-model trace** extracted from Arena-Chat
- **Per-model Static**: each model use 50% of total memory



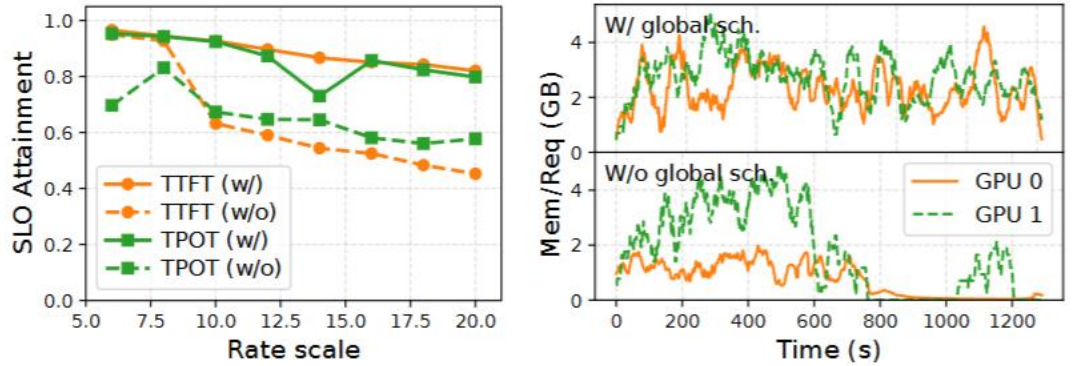
E2-2 Model activation speed

- The activation time for models with different sizes.
- Data is measured on H100 GPUs.



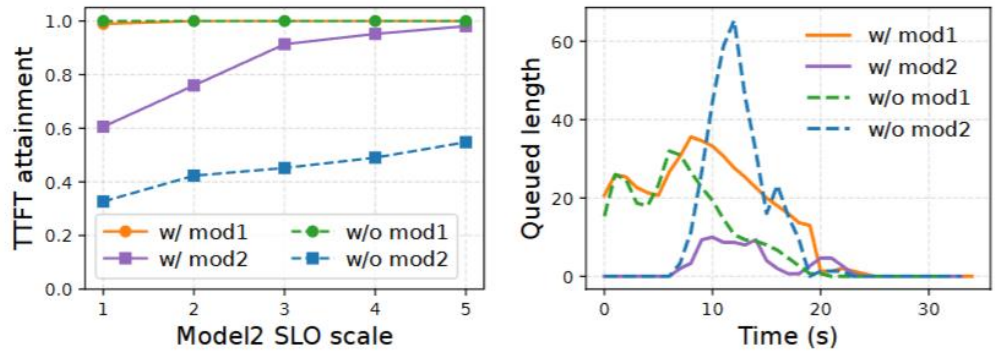
E2-3 Scheduling

- Global model placement scheduling
  - Used two GPUs to serve eight models
- GPU local request scheduling
  - Fix the SLO scale of Model1 to eight and vary the SLO scales of Model2
  - Model1 consistently maintains high attainment
  - enabling our GPU-local scheduling improves the SLO attainment of Model2 by more than 40%



(a) Attainment with rate scales (b) GPU load status

The effectiveness of global model placement scheduling



(a) Attainment with SLO scales (b) Queue length with time

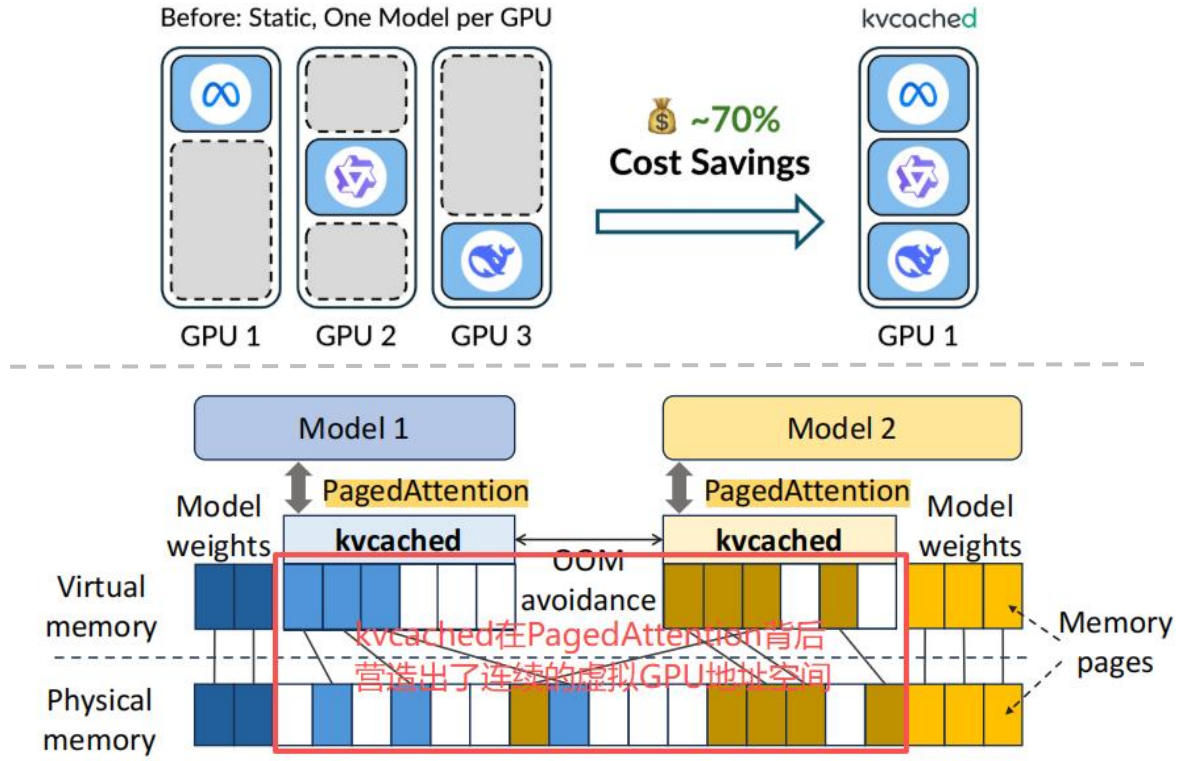
The effectiveness of GPU local request scheduling.

Summary

- **Goal**: Max Tenants Density
- CUDA VMM API
  - cuMemAddressReserve
  - cuMemCreate
  - cuMemMap
  - Example: vectorAddMMAP
- kvcached: support virtual Tensor
- Prism = kvcached + Scheduling

APIs for inference engines	Corresponding `kvcached` functions
alloc_kvcache (size, shape)	alloc_virtual_tensor (size, shape)
free_kvcache (size, shape)	free_virtual_tensor (size, shape)
alloc_kv (num_tokens)	map (tensor, offset)
free_kv ([ids_to_free])	unmap (tensor, offset)

Table 3: APIs and functions provided by Prism's `kvcached`.



We implemented a prototype of Prism with ~10,400 lines of Python and 774 lines of C++ code. As the serving backend, we used SGLang [64], a widely adopted open-source inference engine, and extended it with our `kvcached` library to support on-demand memory allocation. `kvcached` is implemented using CUDA VMM APIs [36] and provides standard KV cache allocation APIs (Table 3) accessible through Python

西北工業大學
NORTHWESTERN POLYTECHNICAL UNIVERSITY

Thanks for Listening

Mingxuan Liu

PhD student at Northwestern Polytechnical University

January 6, 2026