Punica

Serving multiple LoRA fine-tuned LLMs at the cost of one

[MLSys'24]
Lequn Chen* (UW), Zihao Ye* (UW), Yongji Wu (Duke),
Danyang Zhuo (Duke), Luis Ceze (UW), Arvind Krishnamurthy (UW)

UNIVERSITY of WASHINGTON
Duke UNIVERSITY

Adapting Pre-trained LLMs to Tasks

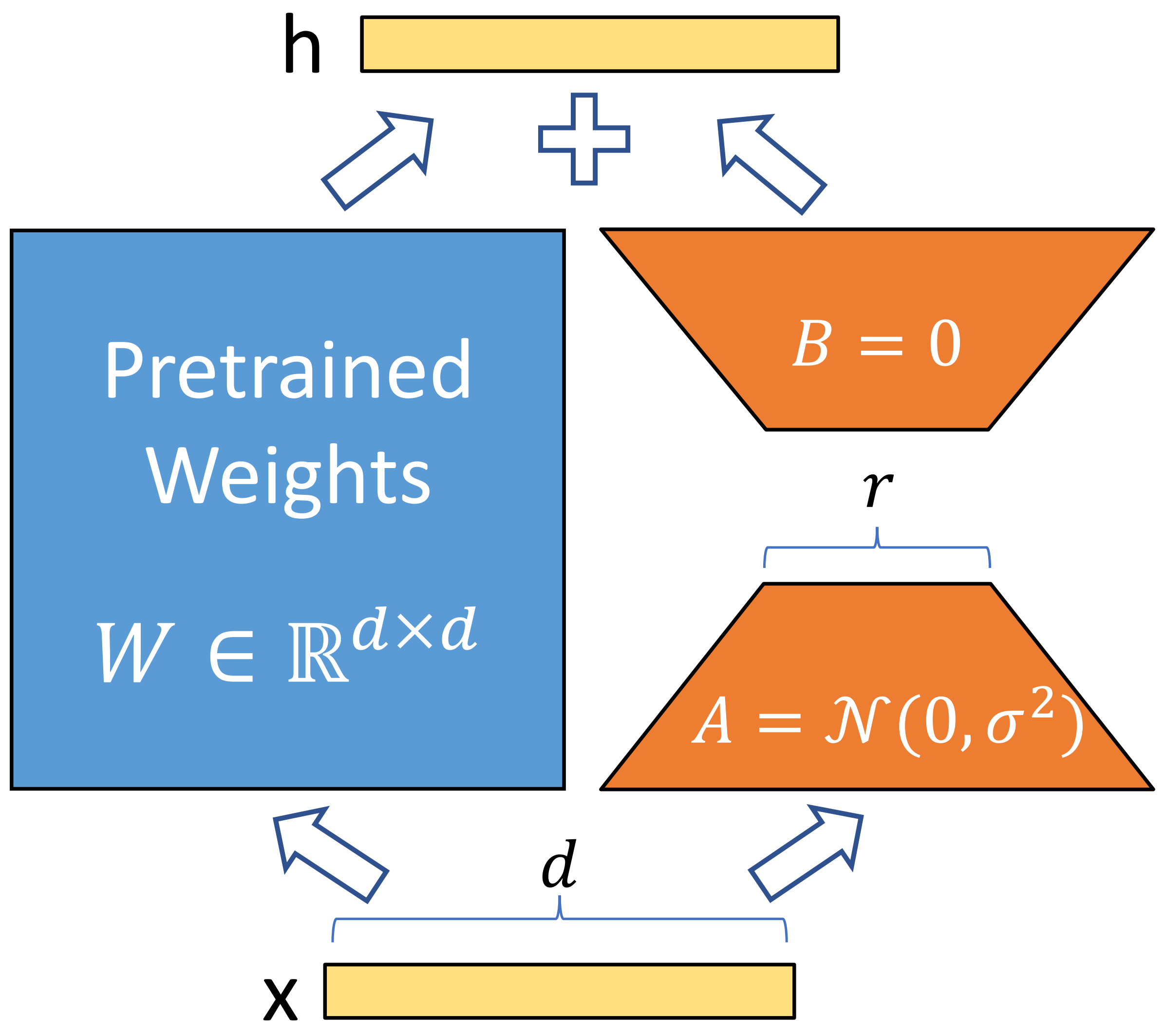
Finetuning

- Follow instructions
- Human alignments
- Adapt to task input/output format
- Add new documents, domain knowledge
- Personalize

LoRA: Low-Rank Adaptation of LLMs

Parameter Efficient Fine-Tuning

- Adding <1% parameter (e.g., r=16, h=4096)
  - W' = W + AB
  - W: [h1, h2], A: [h1, r], B: [r, h2]
  - xW' = x(W+AB) = xW + xAB
- Advantage:
  - Faster training, Lower memory usage
  - Low storage overhead
- How to serve LoRA models?



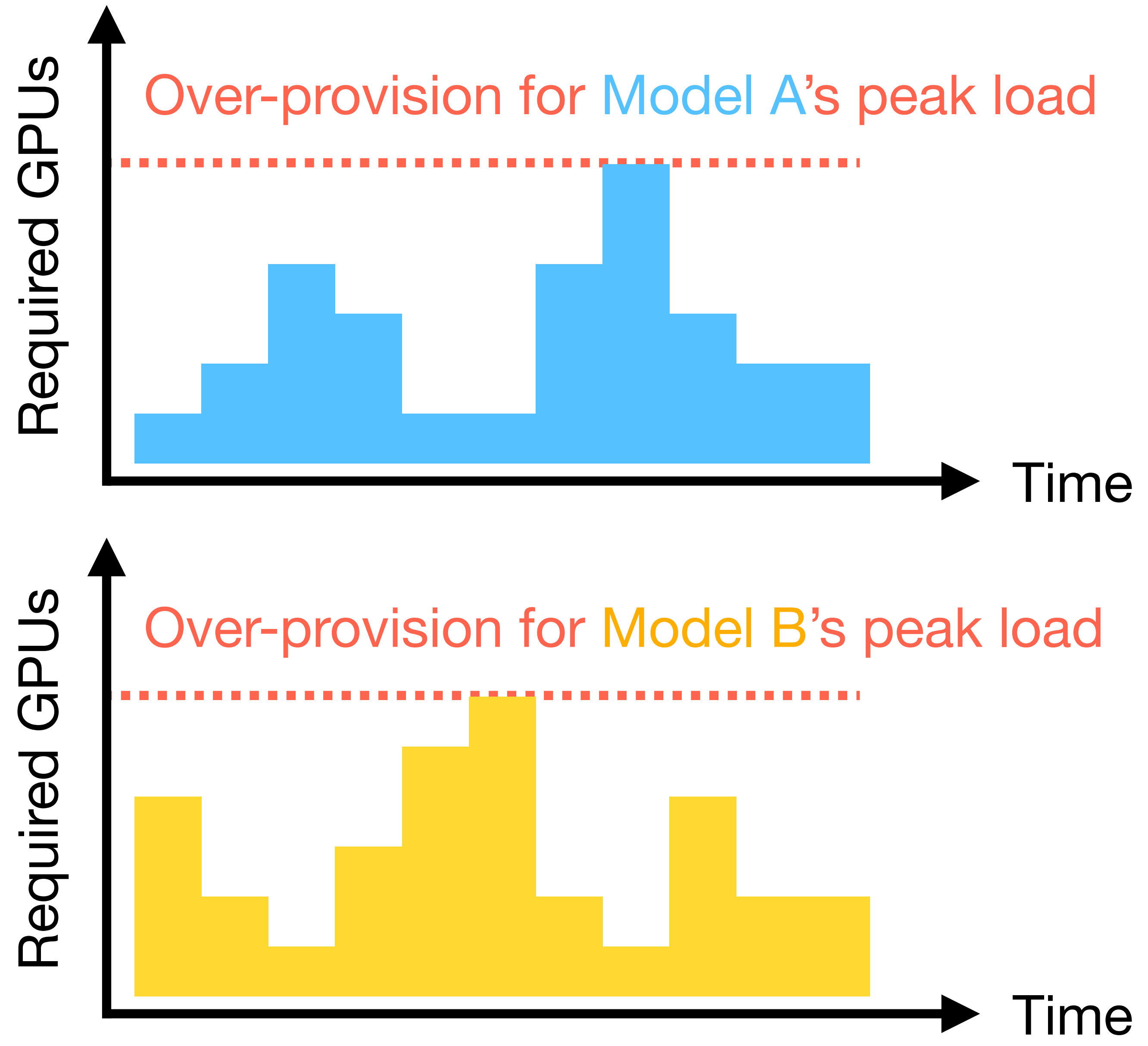
2022, LoRA: Low-Rank Adaptation of Large Language Models

Serving LoRA fine-tuned LLMs

Challenge: Resource over-provision

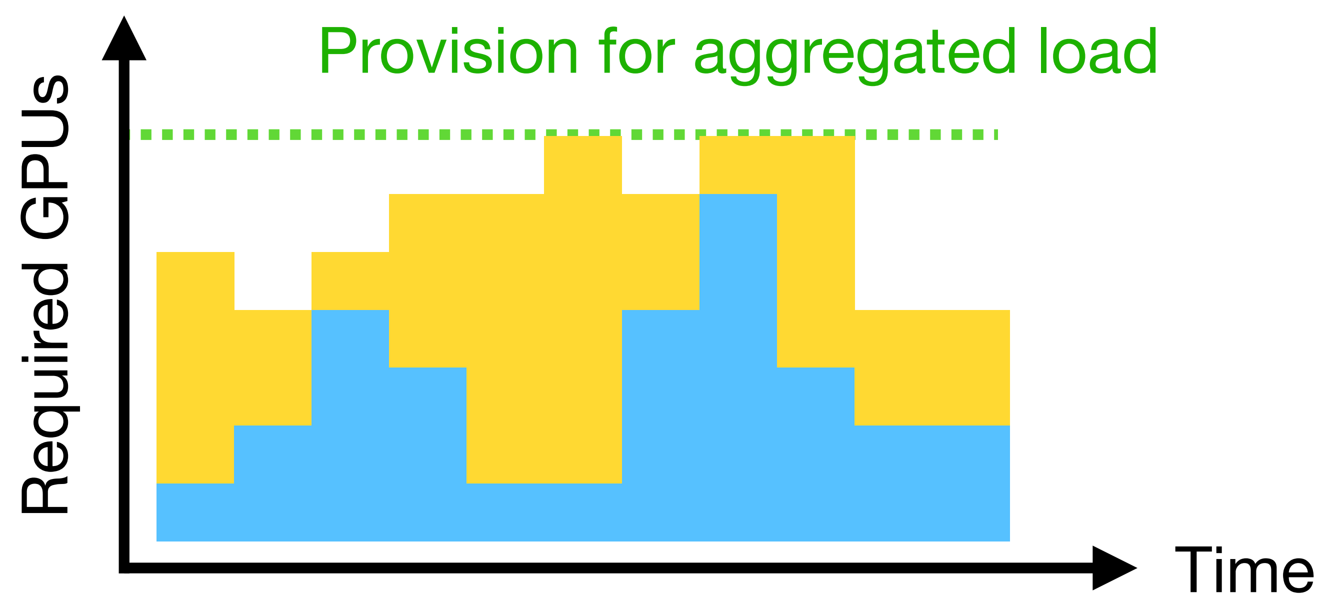
😟 Serving LoRA adapters individually

- Each adapter requires 10 GPUs
- Need 10*5 GPUs for 5 adapters
- Wastes GPU memory for backbone LLM



😃 Multi-tenant LoRA serving

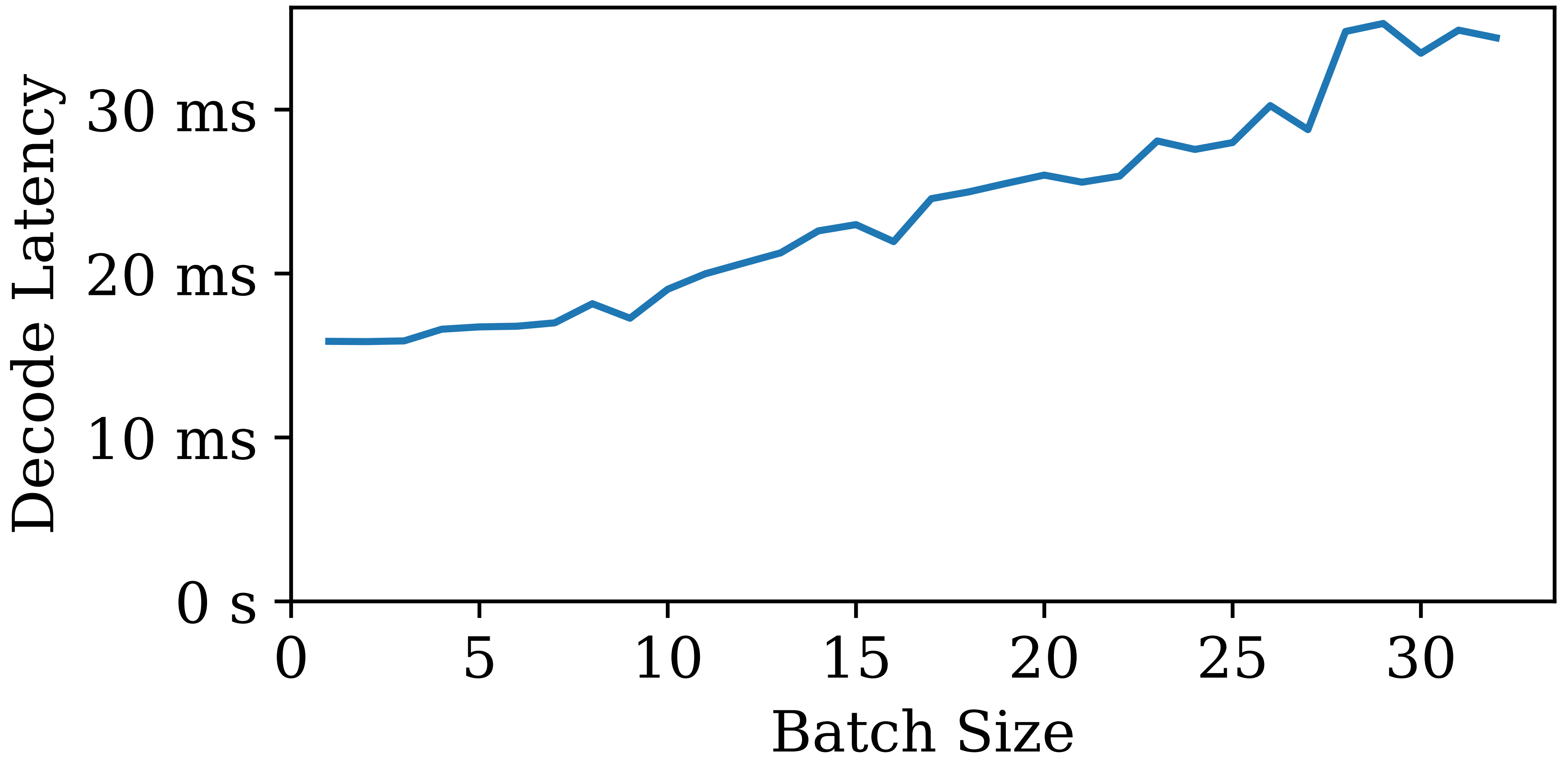
- Pool all GPUs to serve all LoRA adapters
- Smooth out loads
- Much less over-provision
- Share backbone LLM
- (But how?)



Serving LoRA fine-tuned LLMs

Challenge: Reduced batch efficiency

- Straw-man approach for serving multi-LoRA:
  - Group requests by LoRA adapter
  - Swap LoRA weight
- Reduced batch efficiency
- LLM has strong batching effect
  - latency(b=6) is close to latency(b=1)
  - Especially for Dense layers
    - (LoRA is applied to Dense layers)
- Example: AABCCC
  - ✅ Desired: b=6
  - ❌ Reality: b=2, b=1, b=3



How to enable batching for LoRA?

Serving LoRA fine-tuned LLMs

A closer look

- Identical adapter:
  n Requests, 1 Adapter

$$Y := XW + XAB$$

- Distinct adapters:
  n Requests, n Adapters

$$\begin{pmatrix} \vec{y_1} \\ \vdots \\ \vec{y_n} \end{pmatrix} := \begin{pmatrix} \vec{x_1} \\ \vdots \\ \vec{x_n} \end{pmatrix} W + \begin{pmatrix} \vec{x_1} A_1 B_1 \\ \vdots \\ \vec{x_n} A_n B_n \end{pmatrix}$$

- Mixed adapters:
  n Requests, <n Adapters

$$\begin{pmatrix} \begin{pmatrix} \overrightarrow{y_{1,1}} \\ \vdots \\ \overrightarrow{y_{1,b_1}} \end{pmatrix} \\ \vdots \\ \begin{pmatrix} \overrightarrow{y_{n,1}} \\ \vdots \\ \overrightarrow{y_{n,b_n}} \end{pmatrix} \end{pmatrix} := \begin{pmatrix} \begin{pmatrix} \overrightarrow{x_{1,1}} \\ \vdots \\ \overrightarrow{x_{1,b_1}} \end{pmatrix} \\ \vdots \\ \begin{pmatrix} \overrightarrow{x_{n,1}} \\ \vdots \\ \overrightarrow{x_{n,b_n}} \end{pmatrix} \end{pmatrix} W + \begin{pmatrix} \begin{pmatrix} \overrightarrow{x_{1,1}} \\ \vdots \\ \overrightarrow{x_{1,b_1}} \end{pmatrix} A_1 B_1 \\ \vdots \\ \begin{pmatrix} \overrightarrow{x_{n,1}} \\ \vdots \\ \overrightarrow{x_{n,b_n}} \end{pmatrix} A_n B_n \end{pmatrix}$$

Adapter 1

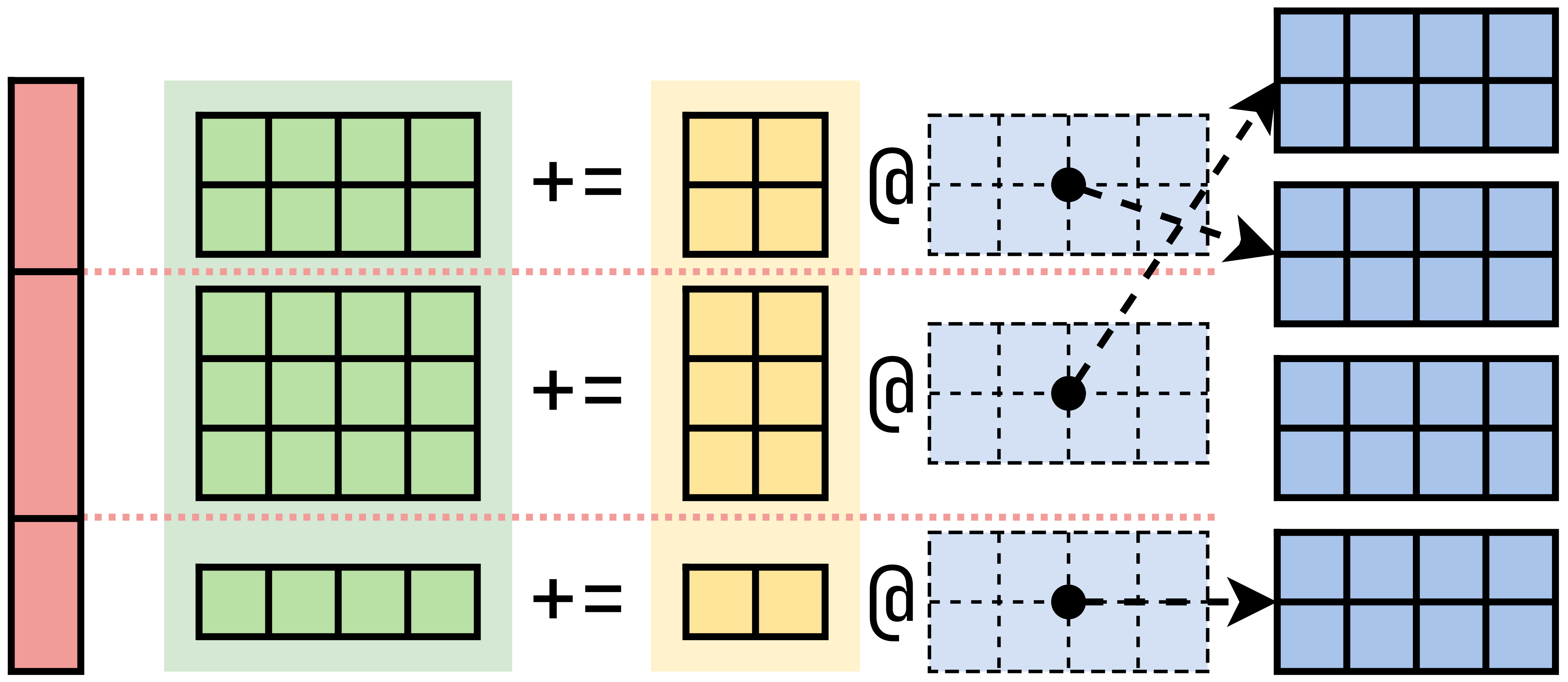
Adapter n

Base model

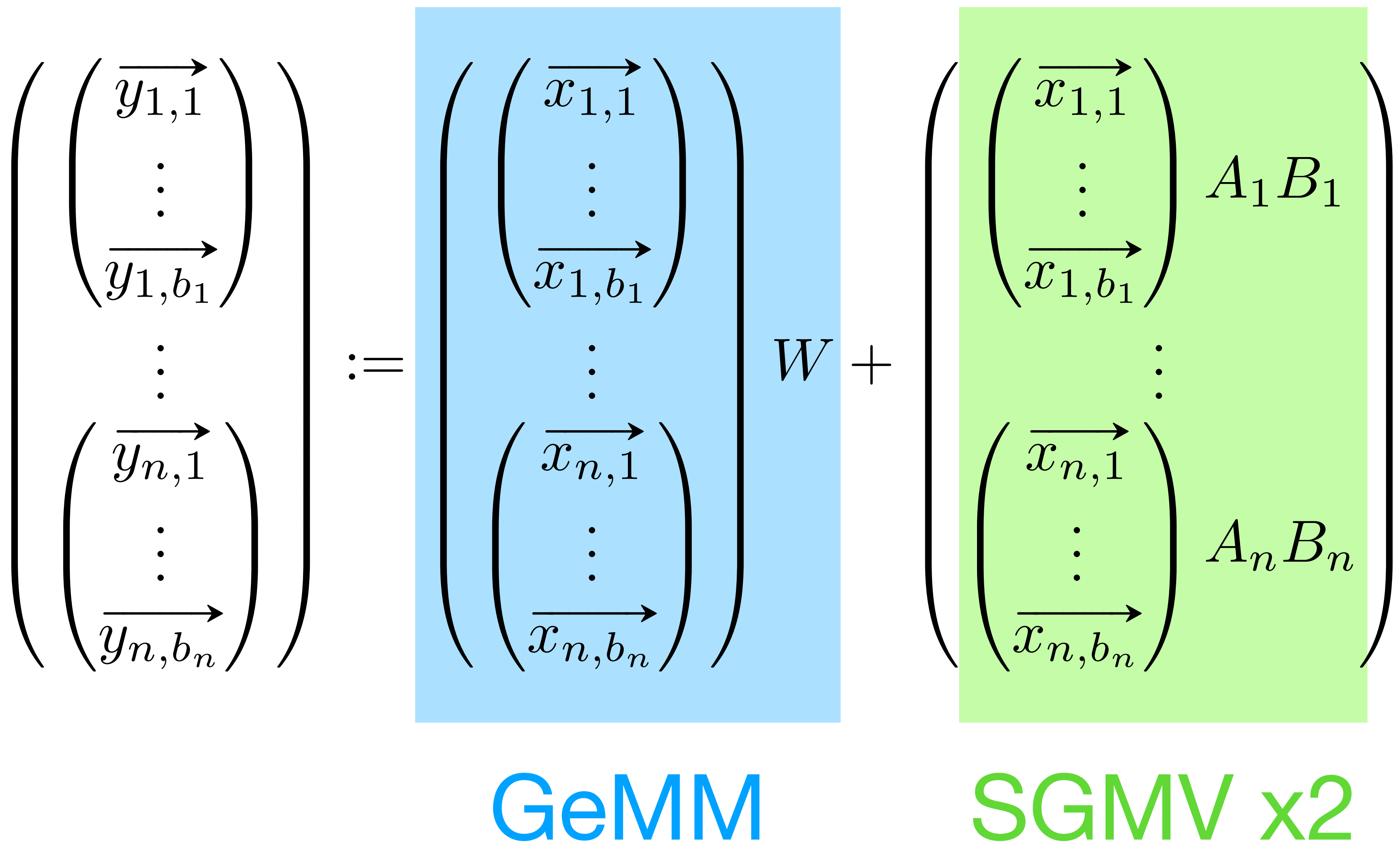
Punica: Serving multiple LoRA LLMs at the cost of one

We made a custom CUDA kernel, called SGMV

Segmented Gather Matrix-Vector Multiplication



`Y[s[i]:s[i+1]] += X[s[i]:s[i+1]] @ W[i]`



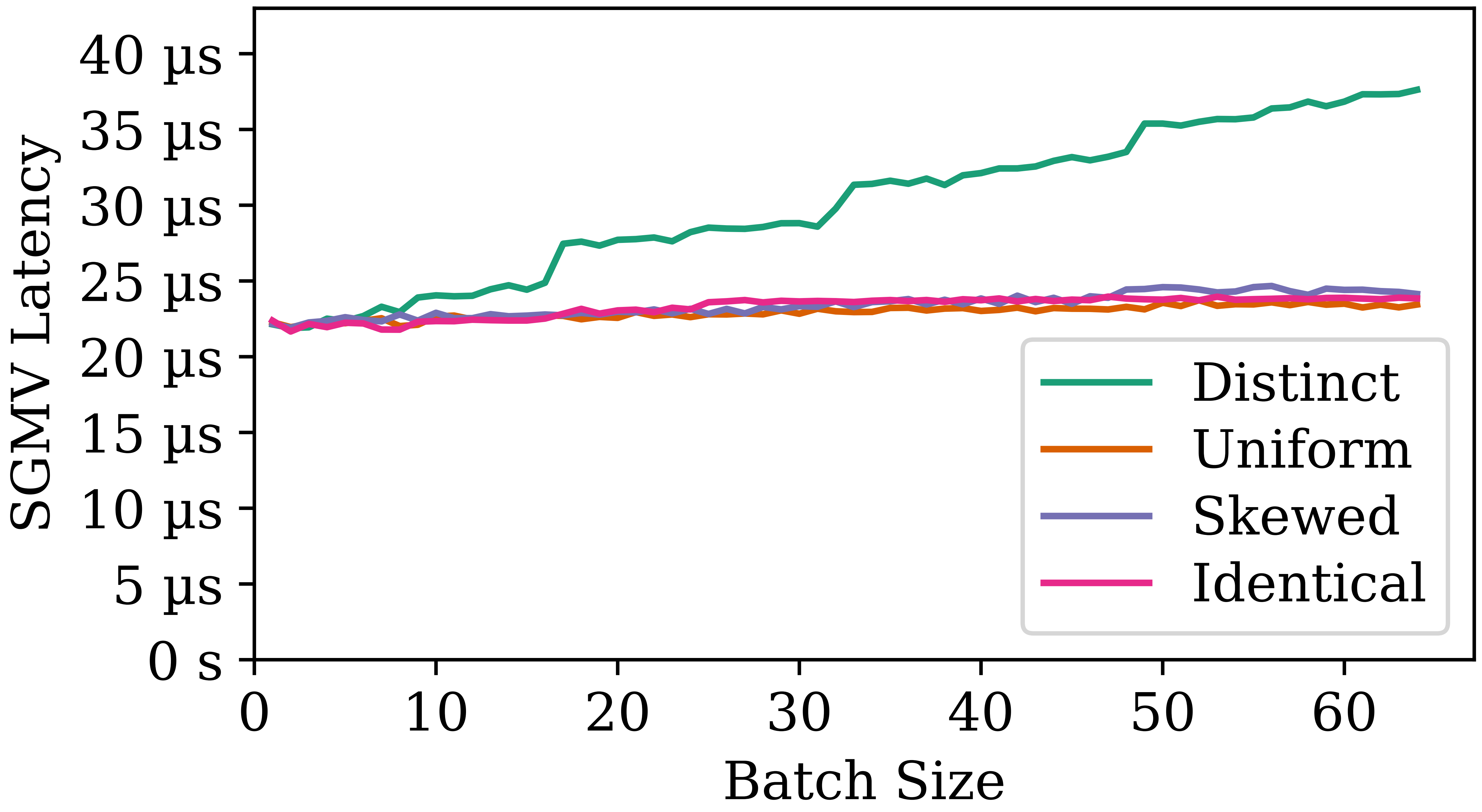
$$\begin{pmatrix} \begin{pmatrix} \overrightarrow{y_{1,1}} \\ \vdots \\ \overrightarrow{y_{1,b_1}} \end{pmatrix} \\ \vdots \\ \begin{pmatrix} \overrightarrow{y_{n,1}} \\ \vdots \\ \overrightarrow{y_{n,b_n}} \end{pmatrix} \end{pmatrix} := \begin{pmatrix} \begin{pmatrix} \overrightarrow{x_{1,1}} \\ \vdots \\ \overrightarrow{x_{1,b_1}} \end{pmatrix} \\ \vdots \\ \begin{pmatrix} \overrightarrow{x_{n,1}} \\ \vdots \\ \overrightarrow{x_{n,b_n}} \end{pmatrix} \end{pmatrix} W + \begin{pmatrix} \begin{pmatrix} \overrightarrow{x_{1,1}} \\ \vdots \\ \overrightarrow{x_{1,b_1}} \end{pmatrix} A_1 B_1 \\ \vdots \\ \begin{pmatrix} \overrightarrow{x_{n,1}} \\ \vdots \\ \overrightarrow{x_{n,b_n}} \end{pmatrix} A_n B_n \end{pmatrix}$$

SGMV Kernel Performance

Under different popularity distribution

- **Distinct**: n Requests, n Adapters
- **Identical**: n Requests, 1 Adapter
- **Uniform**, **Skewed**: in between

- Latency
  - Distinct: increases only slightly
  - Other cases: “free lunch”



Where does the free lunch come from?

Computation for LoRA

Very narrow vector-matrix multiplication

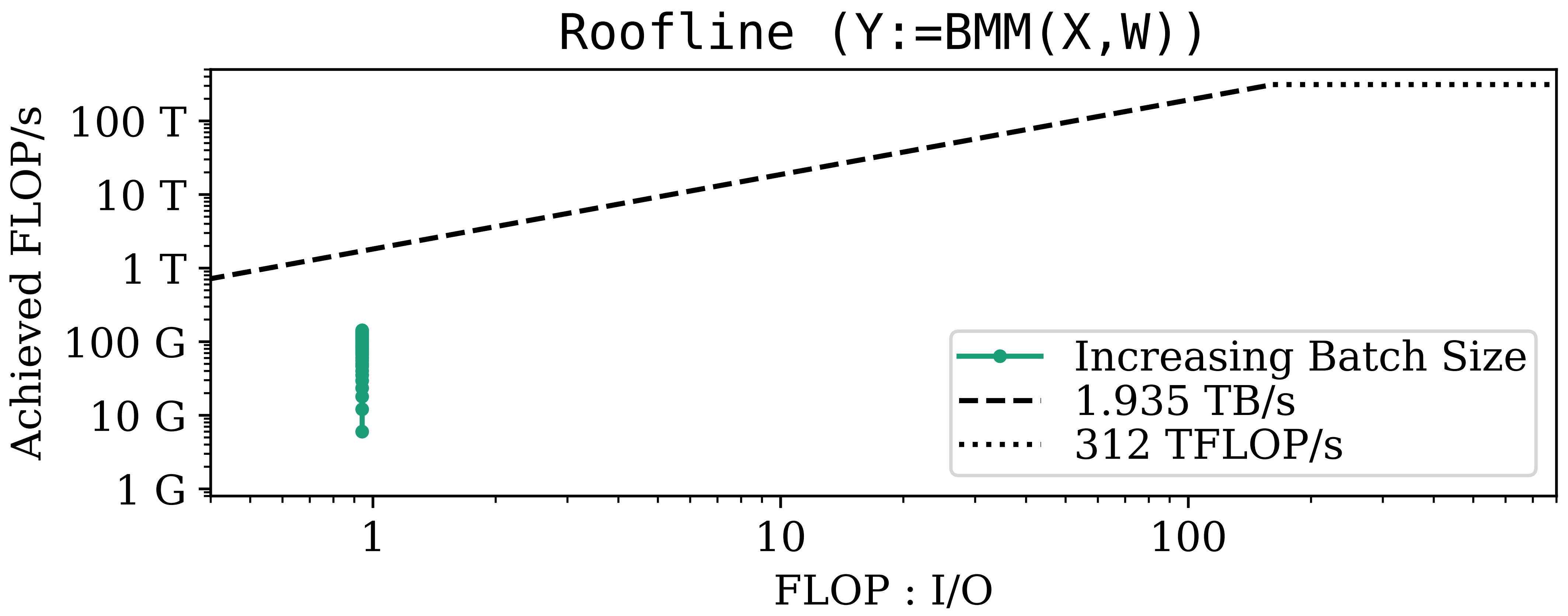
$$\begin{pmatrix} \overrightarrow{x_1} A_1 B_1 \\ \vdots \\ \overrightarrow{x_n} A_n B_n \end{pmatrix}$$

$$\overrightarrow{v} := \overrightarrow{x} A$$

`(1,16) := (1,4096) @ (4096,16)`

$$\overrightarrow{y} := \overrightarrow{v} B$$

`(1,4096) := (1,16) @ (16,4096)`



- Problem: Only utilize a small portion of GPU compute units
- **Batching: Increase degree of parallelism**
- Y := BMM(X, W)
  Yi := Xi @ Wi
  - X: [B, 1, H], W: [B, H, R], Y: [B, 1, R]
- Arithmetic Intensity
  - FLOP: BHR
  - I/O: BH + BR + BHR ≈ BHR
  - Intensity: FLOP/IO ≈ O(1)

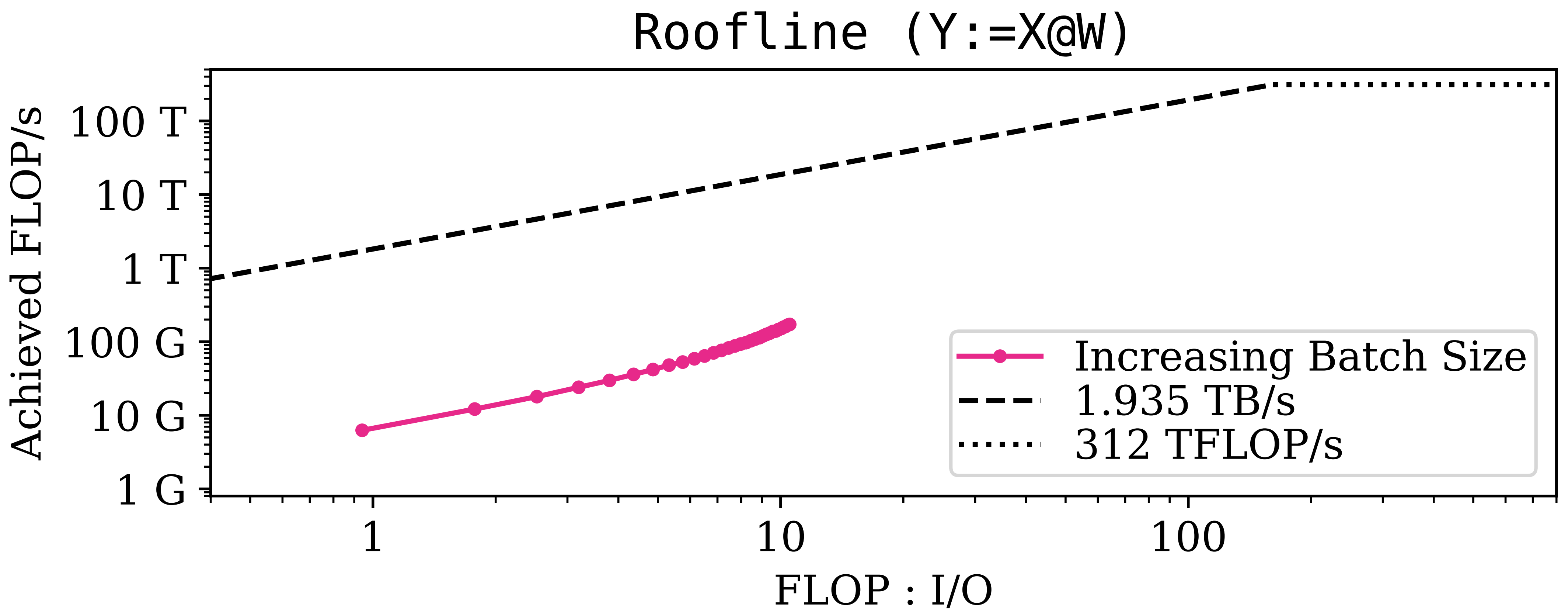
Computation for LoRA

Weight-sharing

$$\begin{pmatrix} \overrightarrow{x_{1,1}} \\ \vdots \\ \overrightarrow{x_{1,b_1}} \end{pmatrix} A_1 B_1 \\ \vdots \\ \begin{pmatrix} \overrightarrow{x_{n,1}} \\ \vdots \\ \overrightarrow{x_{n,b_n}} \end{pmatrix} A_n B_n$$

$$\begin{pmatrix} \overrightarrow{v_1} \\ \vdots \\ \overrightarrow{v_n} \end{pmatrix} := \begin{pmatrix} \overrightarrow{x_1} \\ \vdots \\ \overrightarrow{x_n} \end{pmatrix} A$$

```
(13,16) := (13,4096) @ (4096,16)
```



- Data movement
  - Weight: GPU memory → GPU register
  - Applied to N inputs. Amortized cost
- **Batching: Increase arithmetic intensity**
- Y := X @ W
  - X, Y: [B, H], W: [H, H], H >> B
- Arithmetic Intensity
  - FLOP: BH^2
  - I/O: 2BH + H^2 ≈ H^2
  - Intensity: FLOP/IO ≈ O(B)

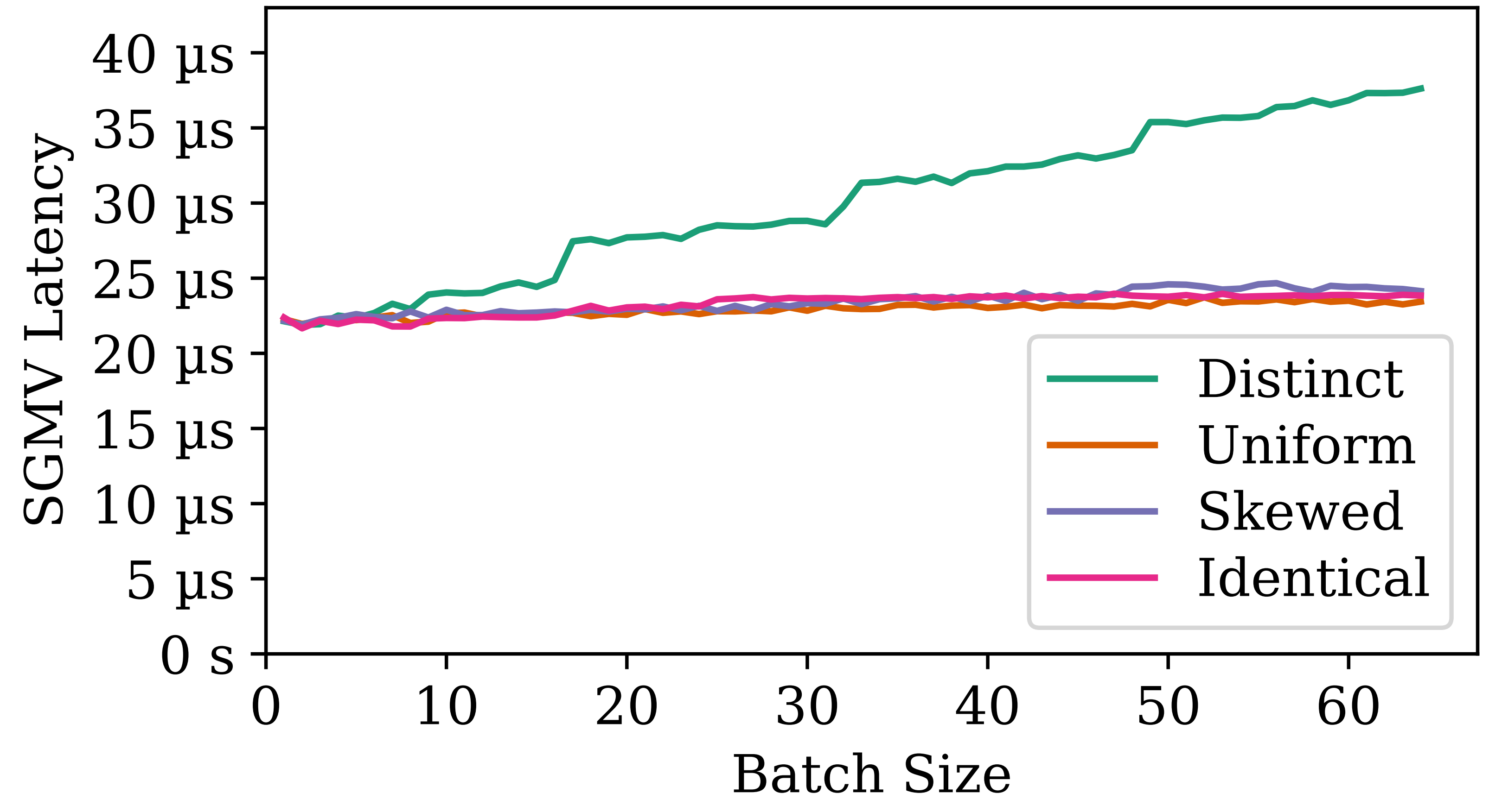
SGMV Kernel Performance

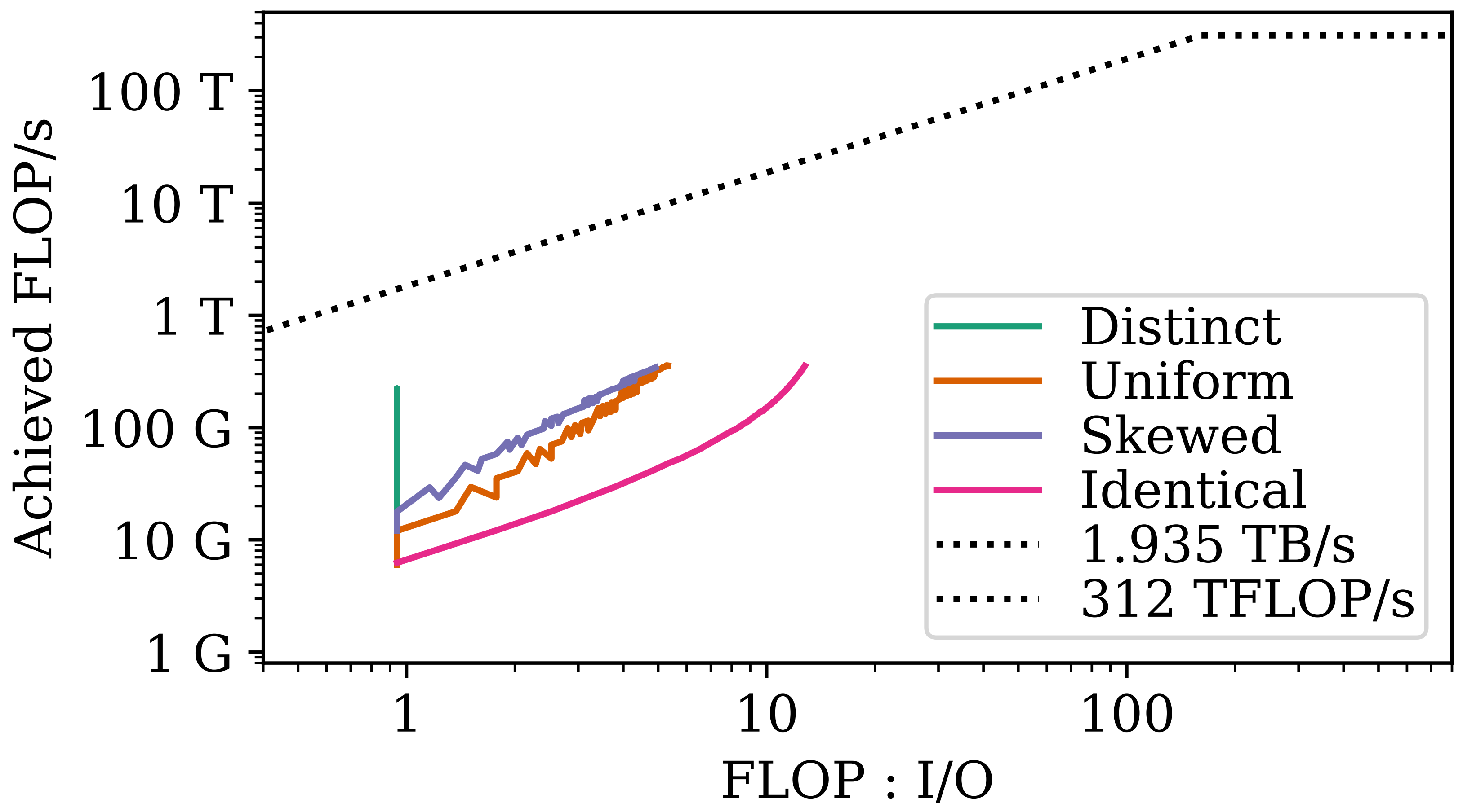
Under different popularity distribution

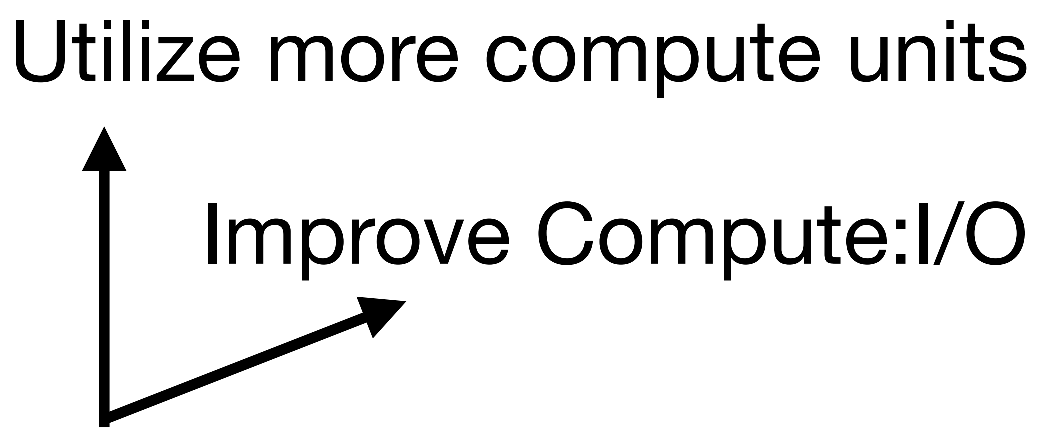
- **Distinct**: n Requests, n Adapters
- **Identical**: n Requests, 1 Adapter
- **Uniform, Skewed**: in between

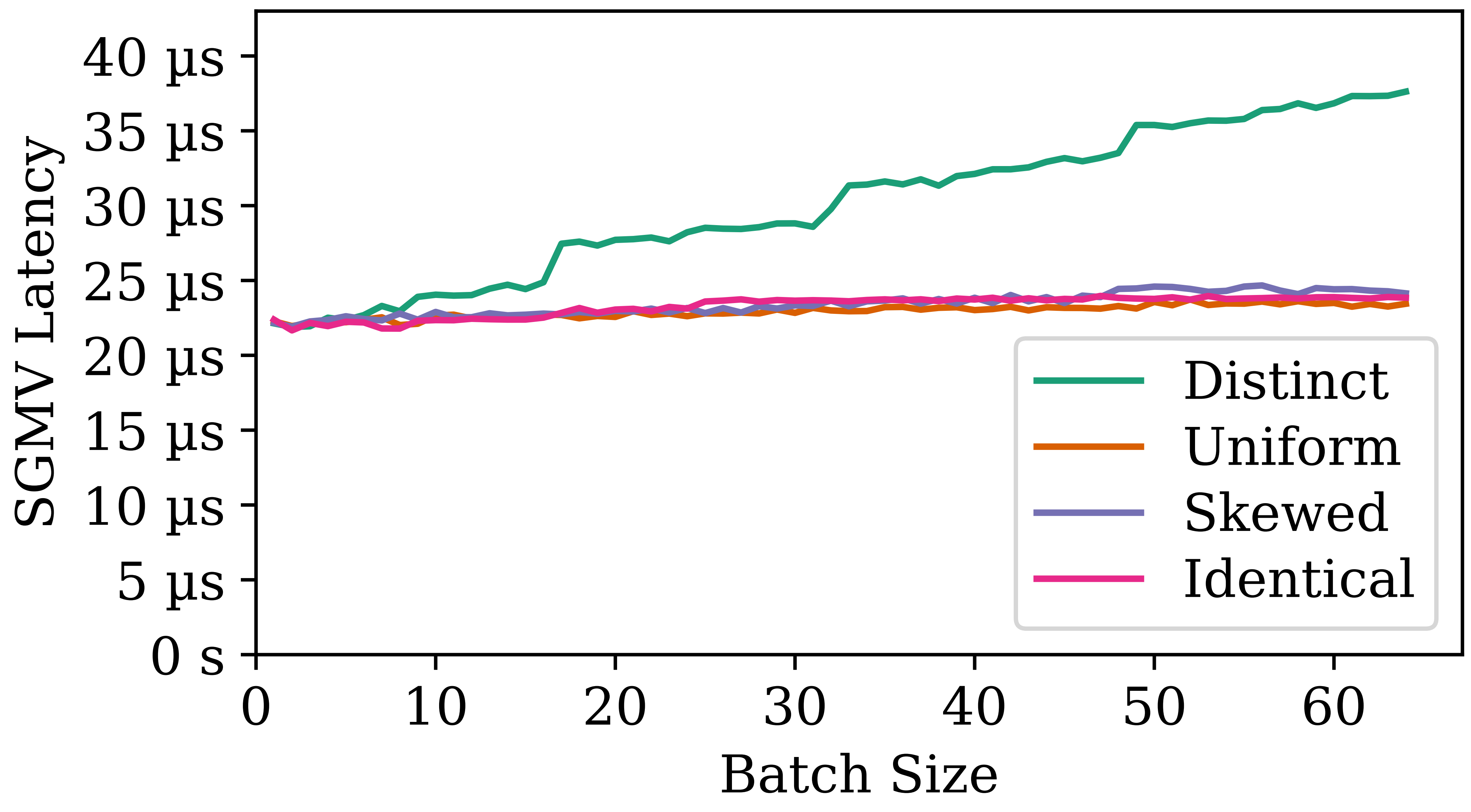
- Latency
  - Distinct: latency gradually increases
  - Others: in “free lunch” range

- Batching effect
  - Improve arithmetic intensity
  - Improve degree of parallelism







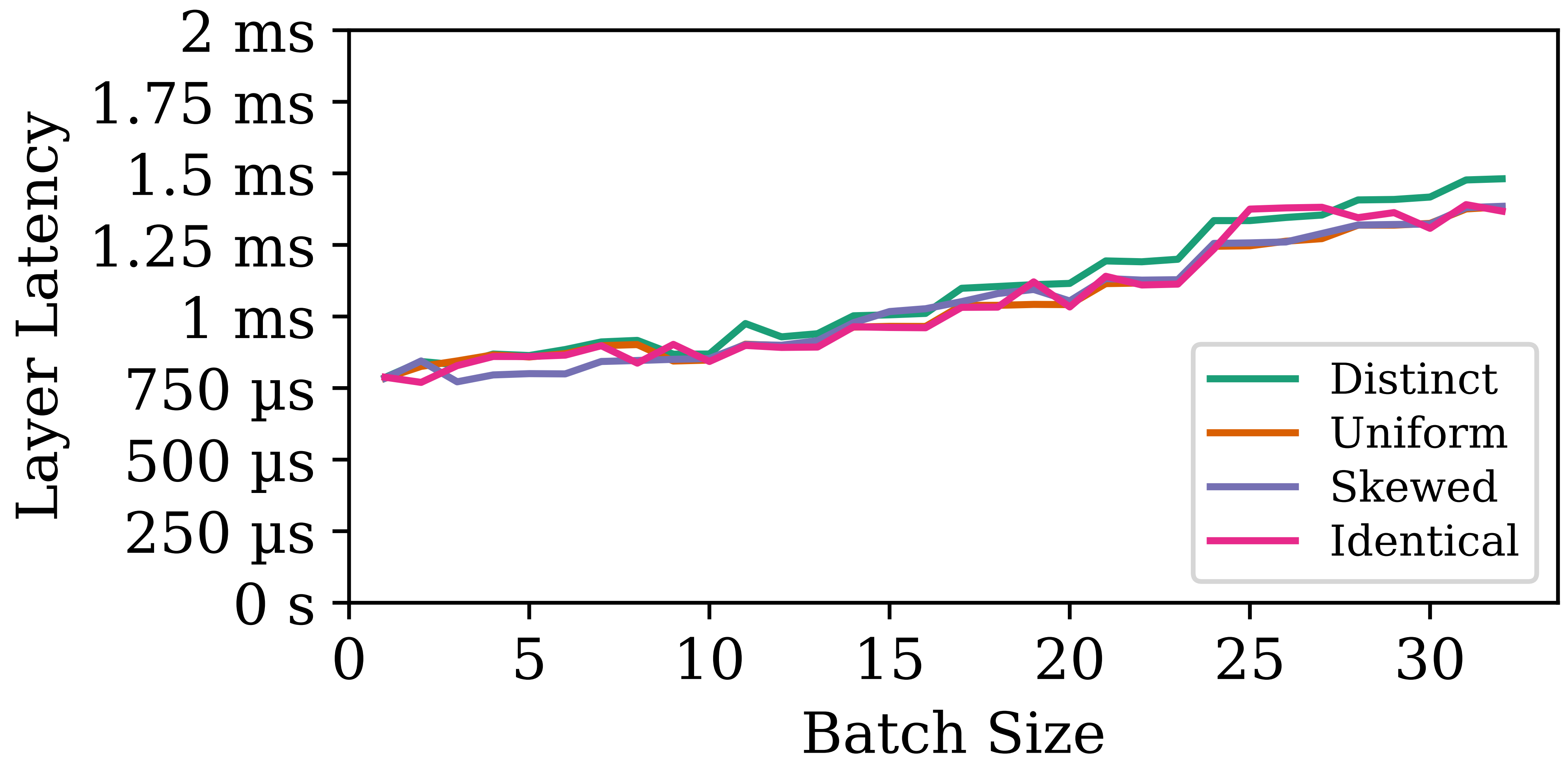


How to handle popularity difference of LoRA models? (1 adapter vs N adapters)

Transformer Layer Latency

Negligible difference across popularity

- Distinct (N adapters) vs Identical (1 adapter): very close
- Negligible difference!
- Popularity difference is hidden e2e
  - Self-Attention is slower than Dense
  - Base model GeMM is slower than LoRA SGMV
  - LoRA adds only about 10% latency



Request Scheduling

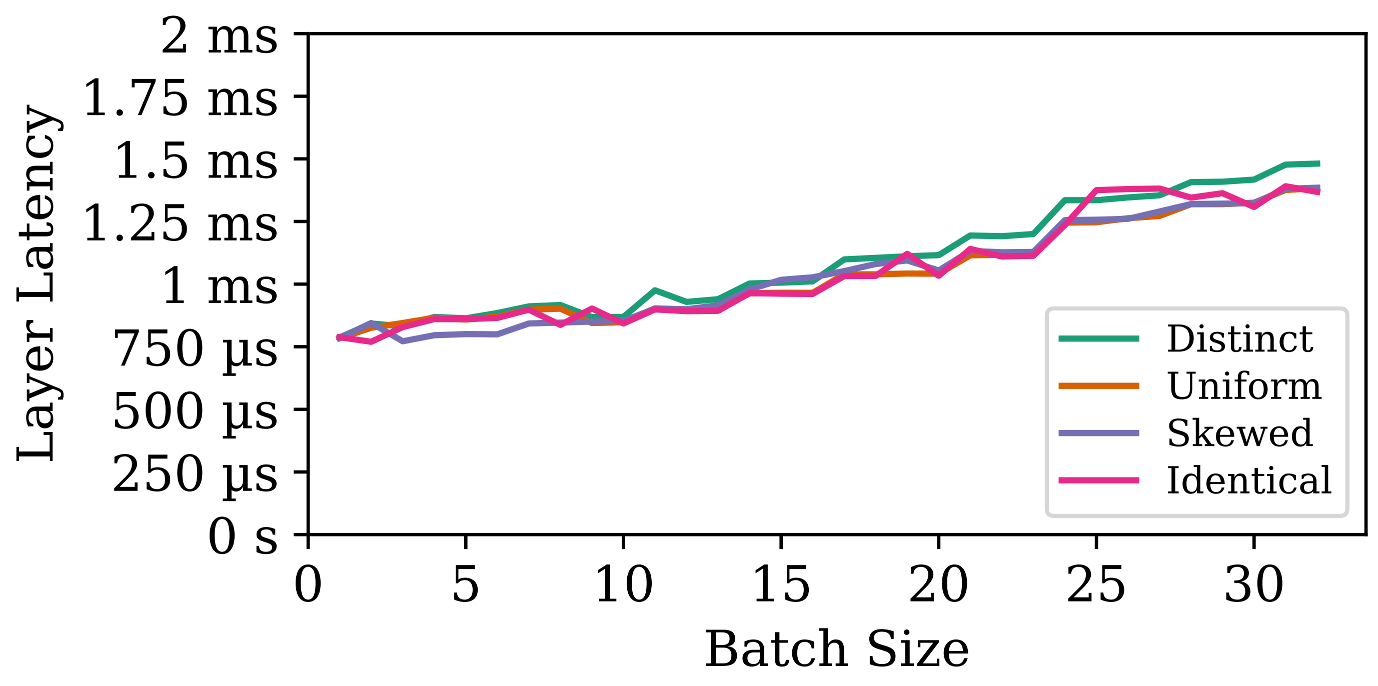
Simple & Effective Solution

- How?
  - Dispatch to **busiest** available GPU
    - Subject to GPU memory size limit for KvCache
  - On-demand loading of LoRA adapters (2ms)
    - This does not block the computation of the existing batch
- Why?
  - Batch size is the most important thing
  - Hundreds of decode steps (30ms per step) + affinity
  - Consolidate GPU usage, Auto-scaling

Punica: Serving multiple LoRA LLMs at the cost of one

Simplified system design

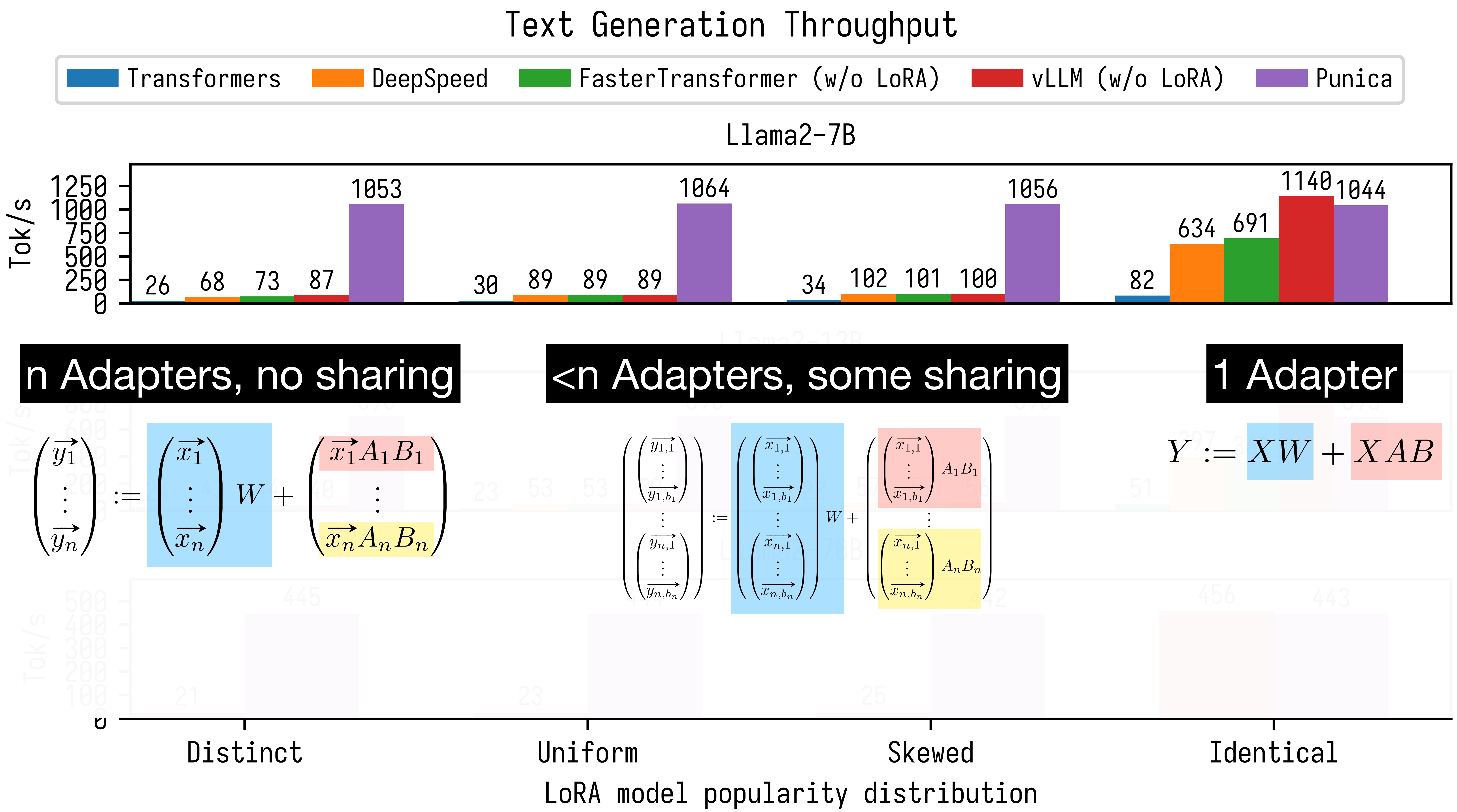
- 😆 Serve N models == Serve 1 model
- ✅ Share base model weight
- ✅ Batching efficiency
- ✅ Resource provision
- ✅ Amortize request rate fluctuation



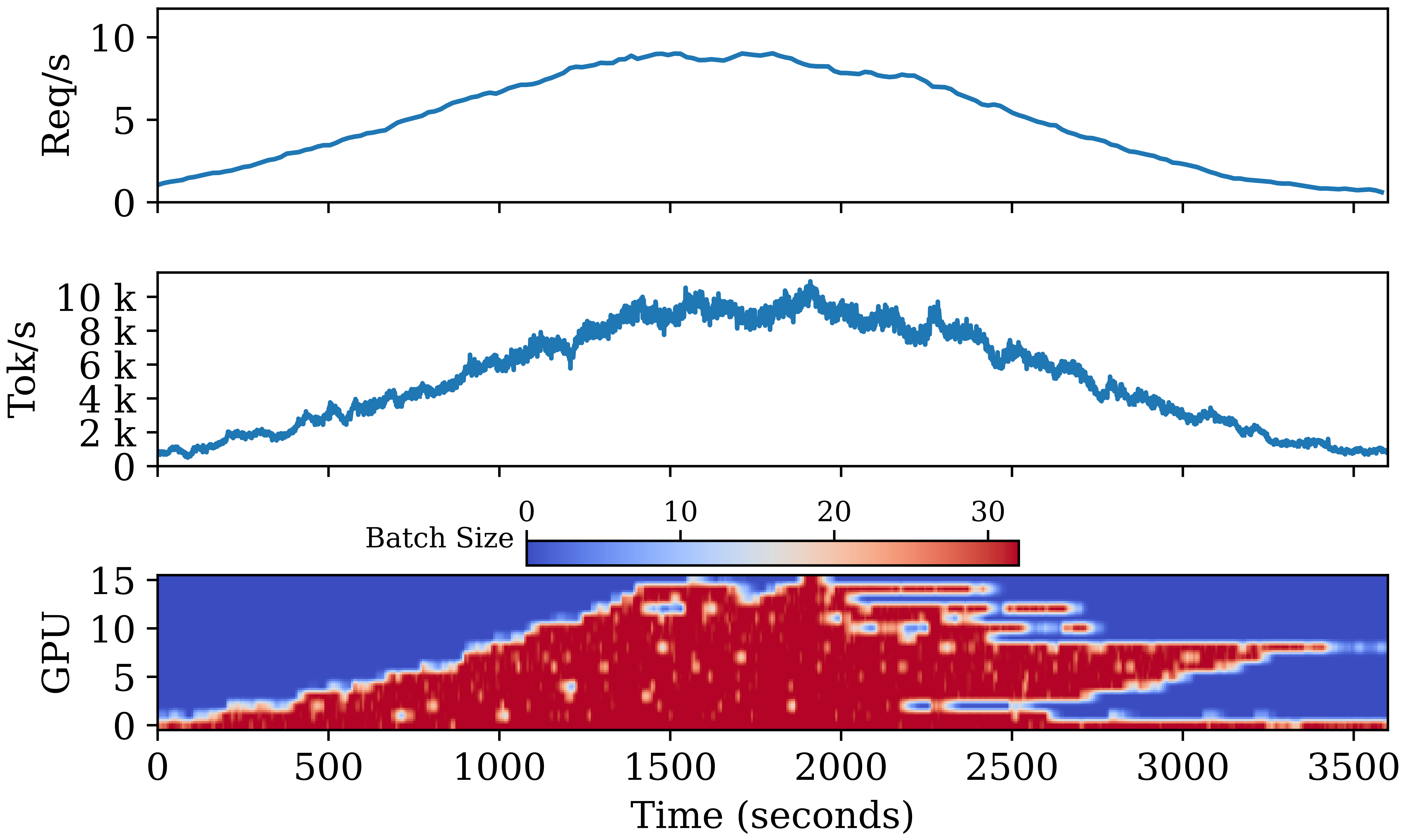
- ✅ Apply common LLM optimizations
  - Continuous batching
  - Request migration
  - Weight quantization
  - FlashInfer (github.com/flashinfer-ai/flashinfer)
    - PagedAttention
    - FlashAttention
    - Batch decoding
    - Ragged input
    - Share-prefix decoding
    - INT4/FP8 KVCache quantization
    - Optimized Group Query Attention
  - …

12x

Text Generation Throughput (Single Instance)



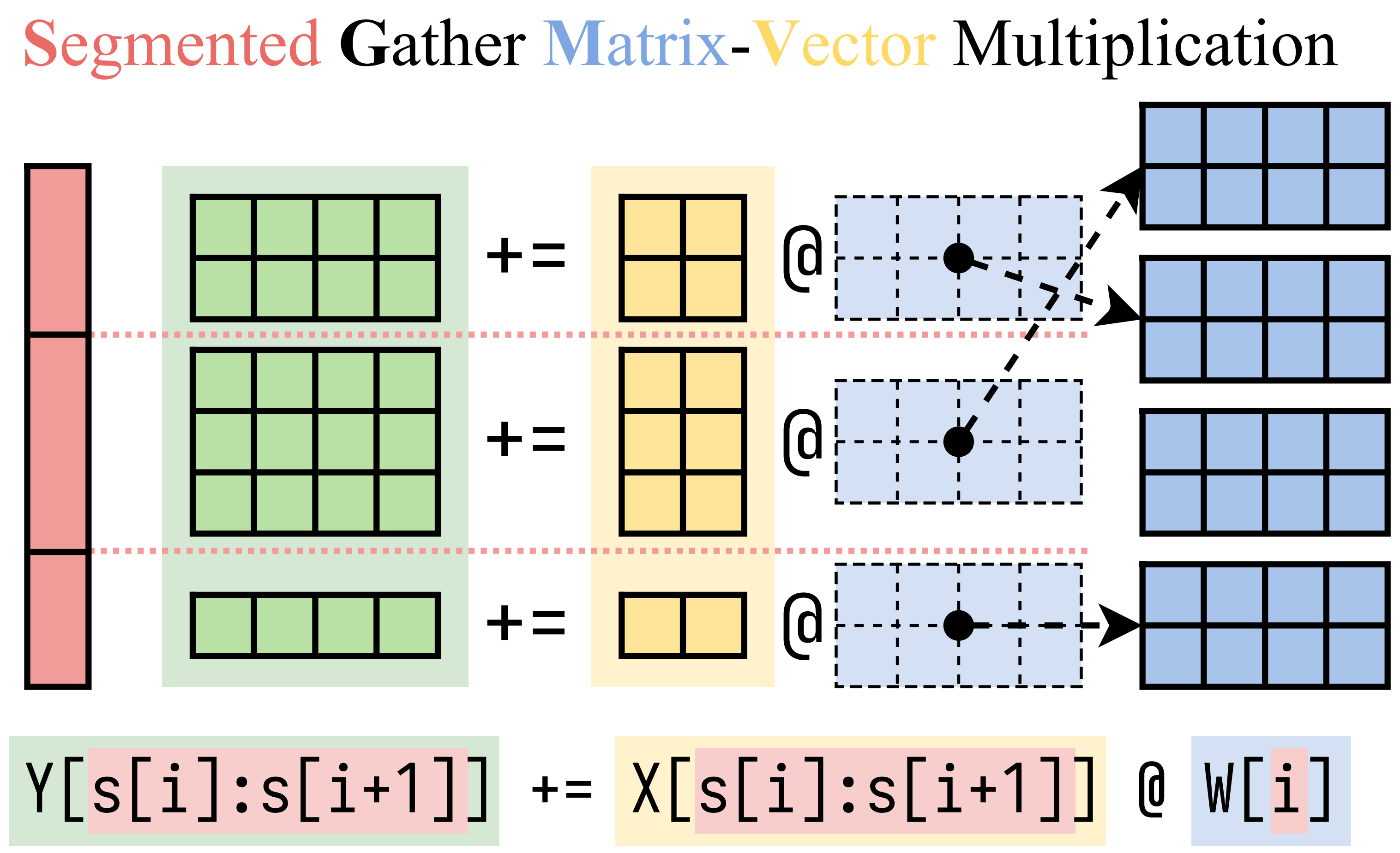
Consolidate Cluster-wide GPU Usage

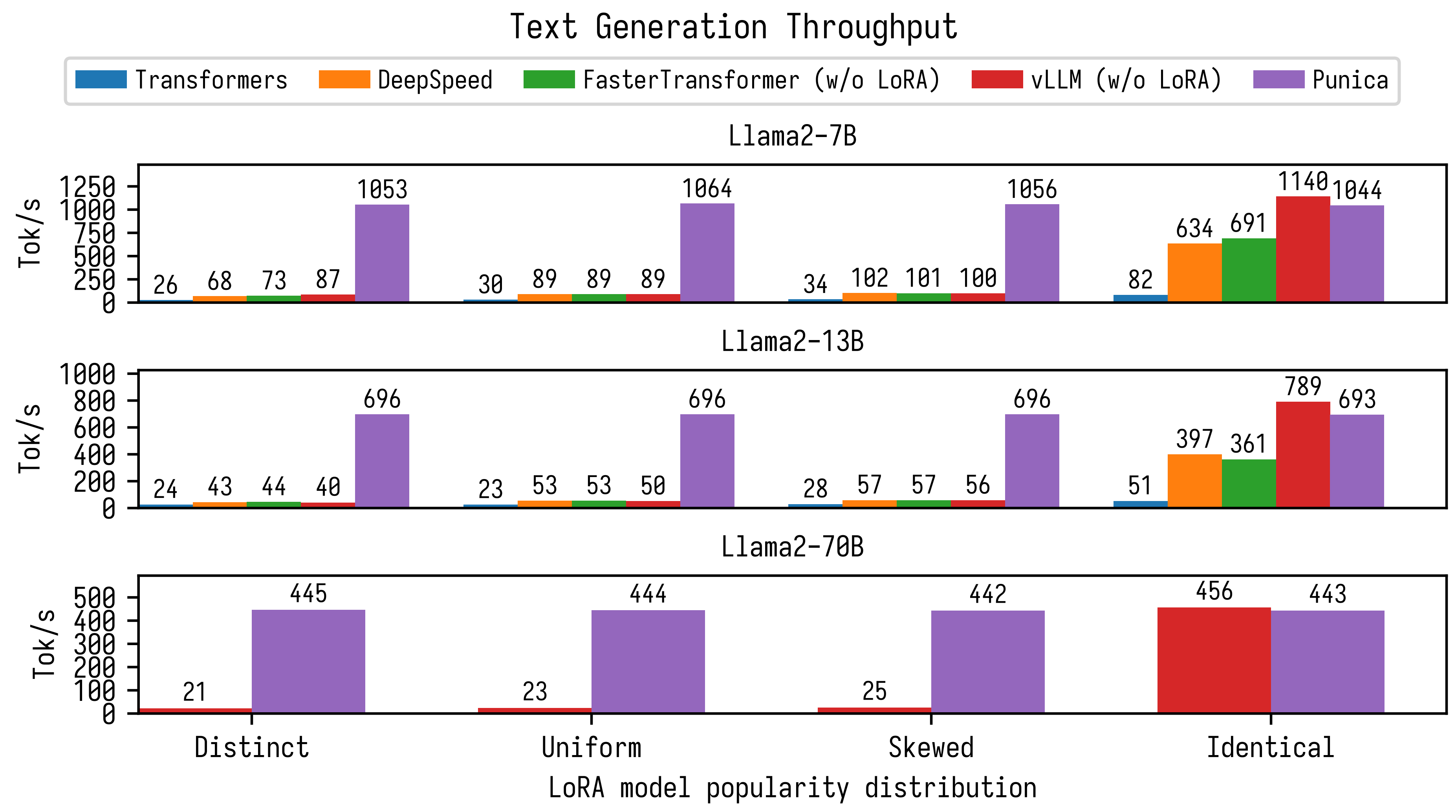


Punica: Serving multiple LoRA LLMs at the cost of one

- SGMV kernel: efficiently batch different LoRA models
- Simplify multi-model scheduling as single-model scheduling
- Consolidate GPU usage by prioritizing batch size
- 12x throughput

https://github.com/punica-ai/punica





Backup Slides

Comparison with S-LoRA [MHni]

- Please note that S-LoRA is arxived on Nov 6. We didn't have an opportunity to do a quantitative comparison before the MLSys deadline. Here are the differences based on reading the S-LoRA paper.
- *S-LoRA is built upon the open-source code of an earlier version of Punica*, in particular, the BGMV kernel. BGMV assumes different LoRA models for each input in the batch. It suffers in the following two cases:
- (1) Prefill. In the prefill stage, thousands of tokens may map to the same LoRA weight. S-LoRA addresses this issue by writing a kernel for prefill (MBGMM).
- (2) Shared LoRA weight across requests. S-LoRA does not address this problem.
- We solve both problems efficiently with SGMV. SGMV's semantics cover both.
- S-LoRA extends the BGMV kernel to support different ranks. As discussed in the previous section, we can easily add this support to SGMV.
- S-LoRA's Unified Paging is an extension to PagedAttention, fitting LoRA weights to the memory pool layout. We rely on PyTorch's cached memory allocator for memory management and have no such constraints.
- S-LoRA implemented prefetching and overlapping for loading LoRA weights. Our paper discusses this option and opts to use on-demand loading (Section 5.2).
- S-LoRA's tensor parallel scheme shards the computation of LoRA but adds communication. Our tensor parallel scheme replicates one side of LoRA, thus avoiding extra communication, as discussed in the previous section.