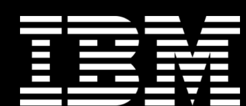
Portable High-Performance LLM Serving: A Triton Backend for vLLM

—

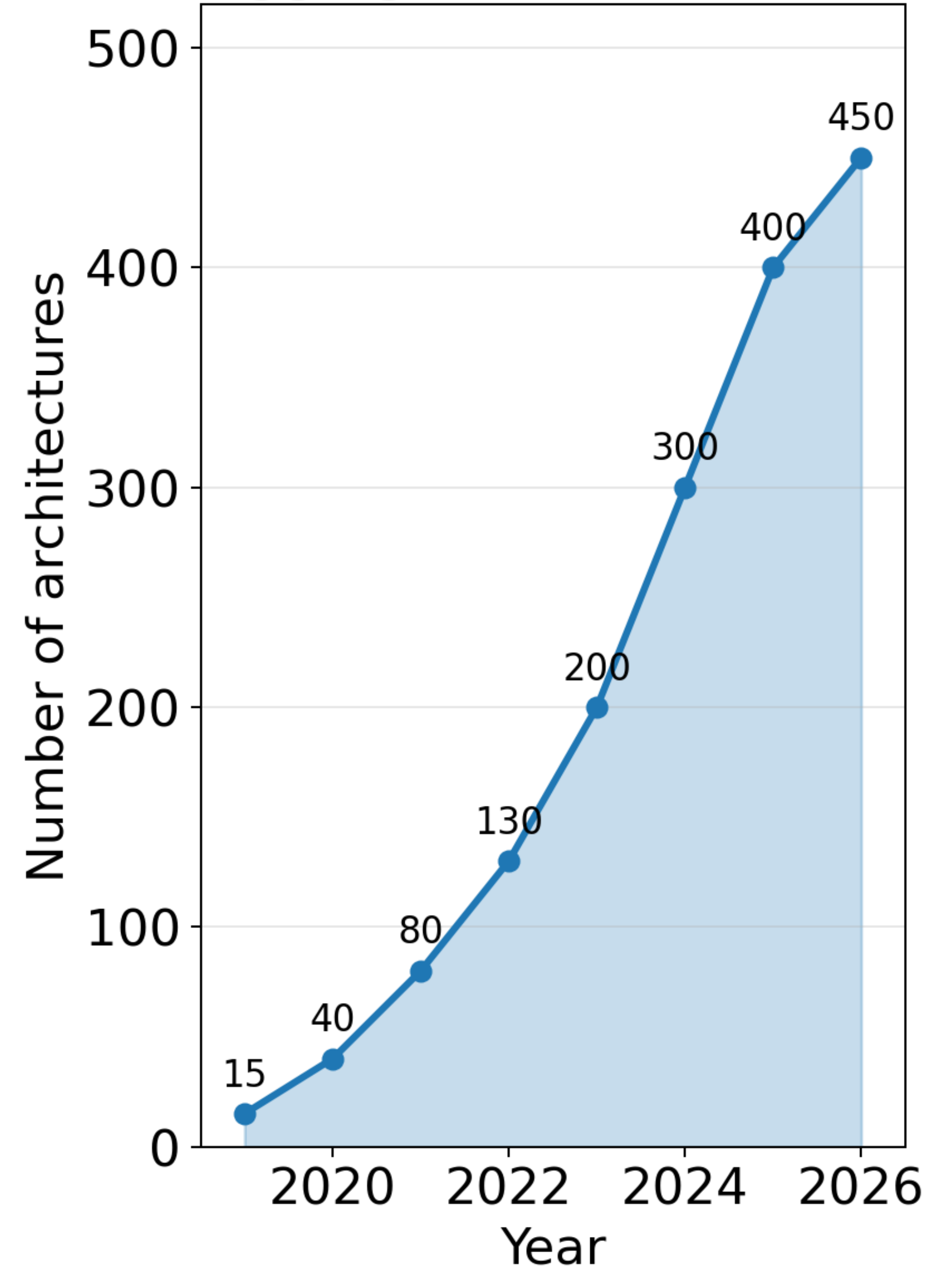
Jan van Lunteren
Burkhard Ringlein
IBM Research
AI Platform Group, Zurich

2026-04-08, PyTorch Conference, Paris

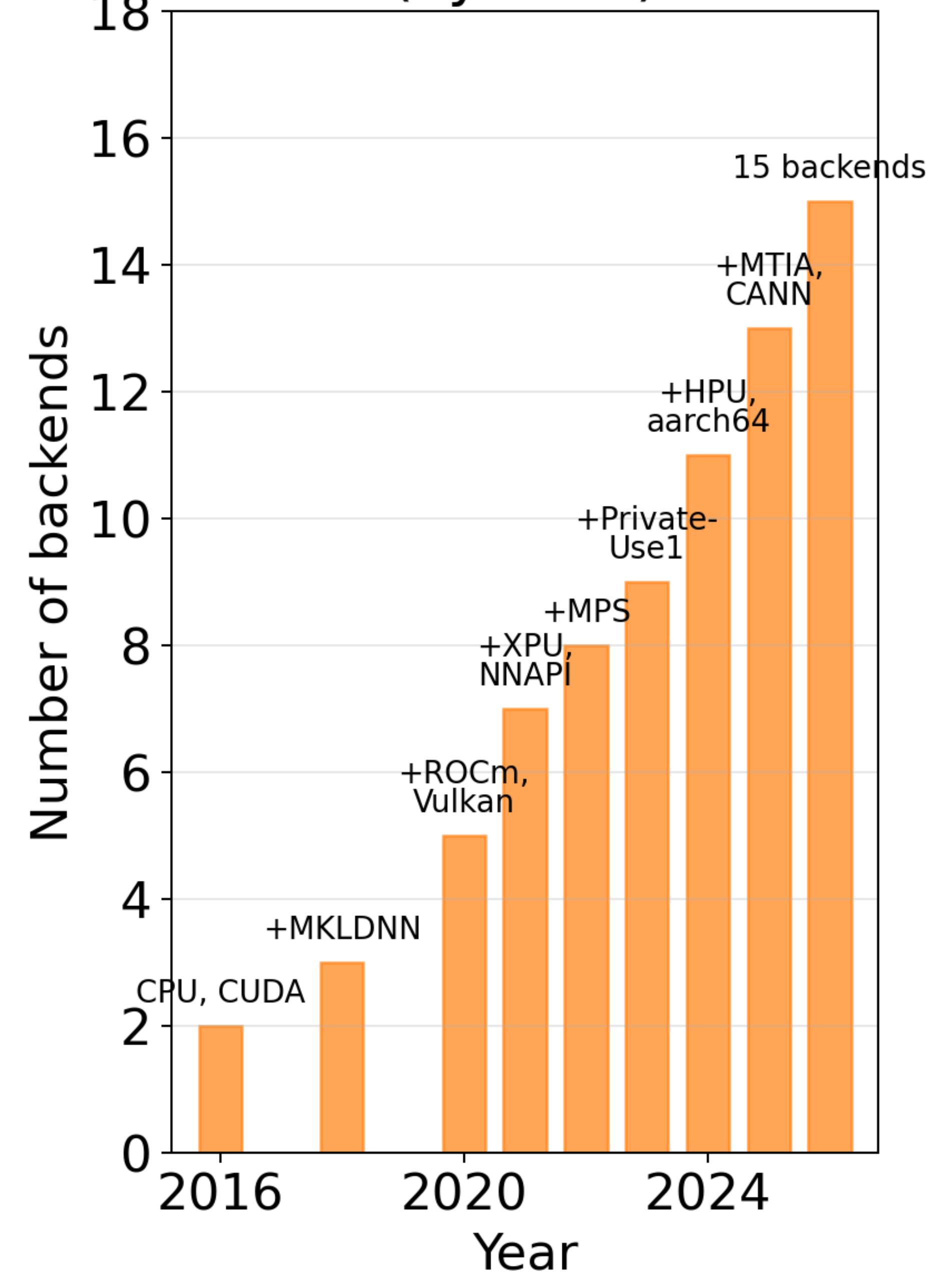


Open Source AI is growing...

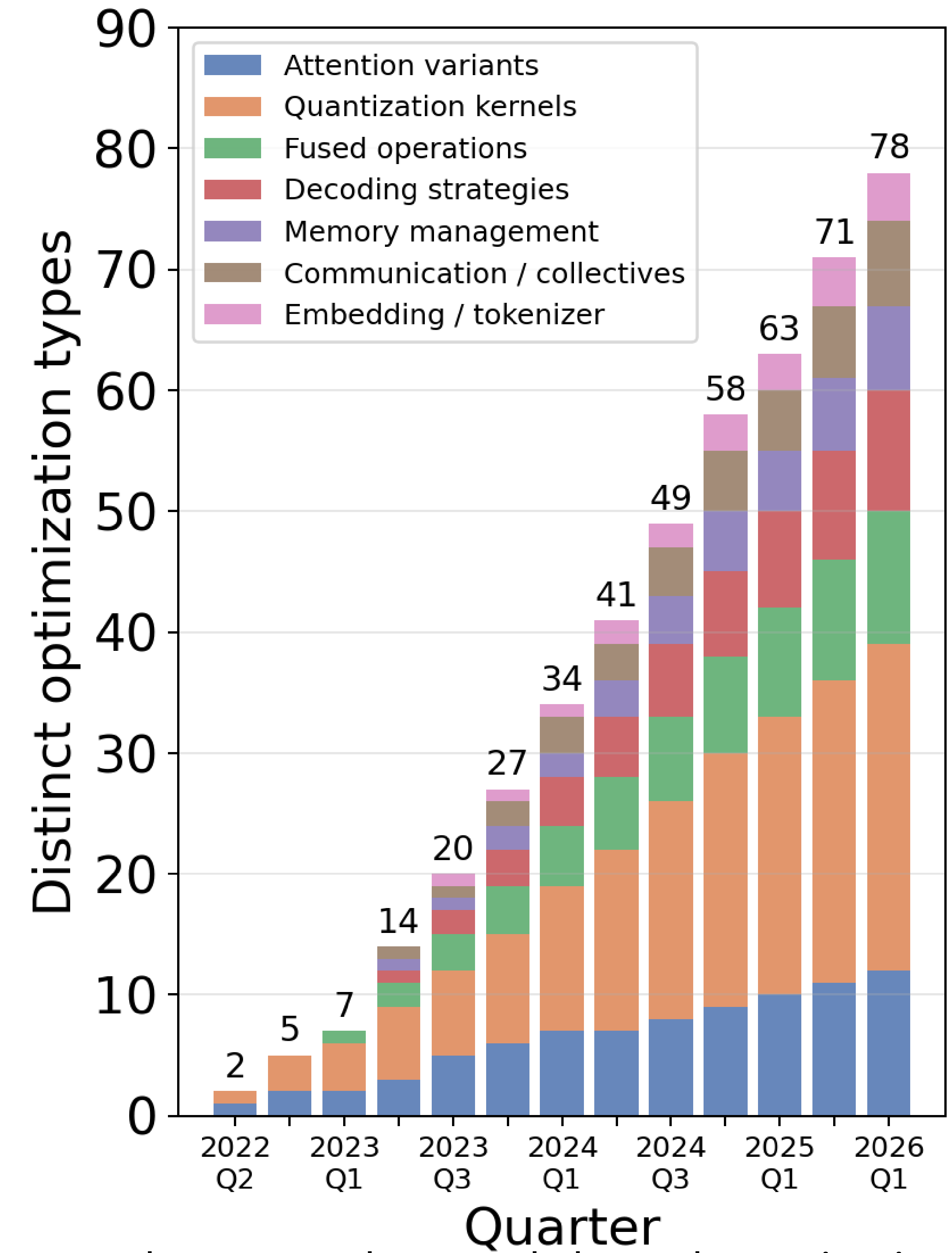
Specialized Kernel Optimizations (Open-Source LLM Inference Ecosystem)



More models...



...on more hardware...



How can vLLM keep up?

...with much more kernel-level optimizations?

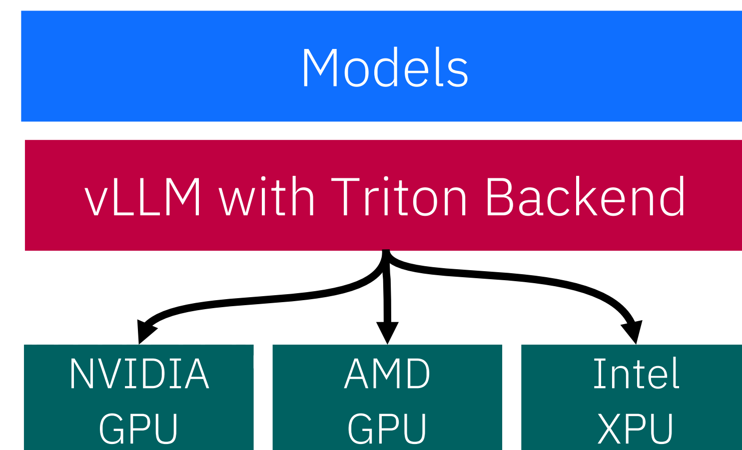
→ performance portability!?!

 © 2026 IBM Corporation

Agenda: How to achieve performance portability for LLM kernels

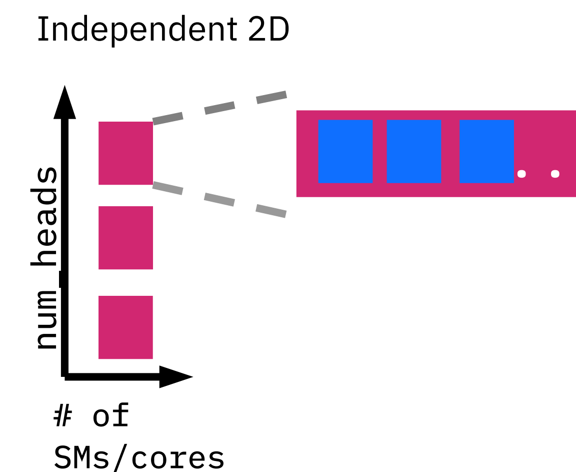
(Or, how to vLLM can keep up!)

1. Some background: Triton & performance portability

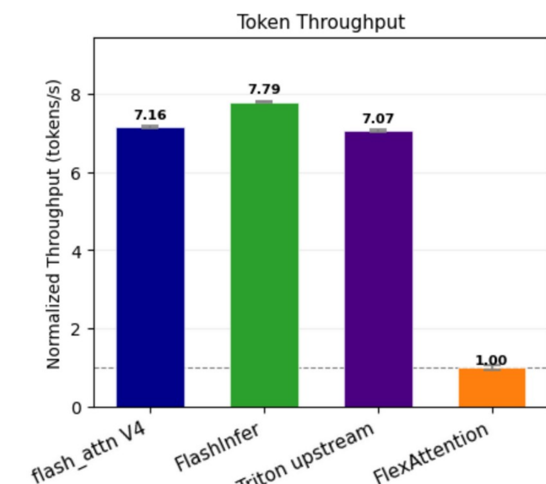


2. Triton-only Attention Backend in vLLM

3. Deep Dive: How to write portable paged attention



4. Results



Conclusion?

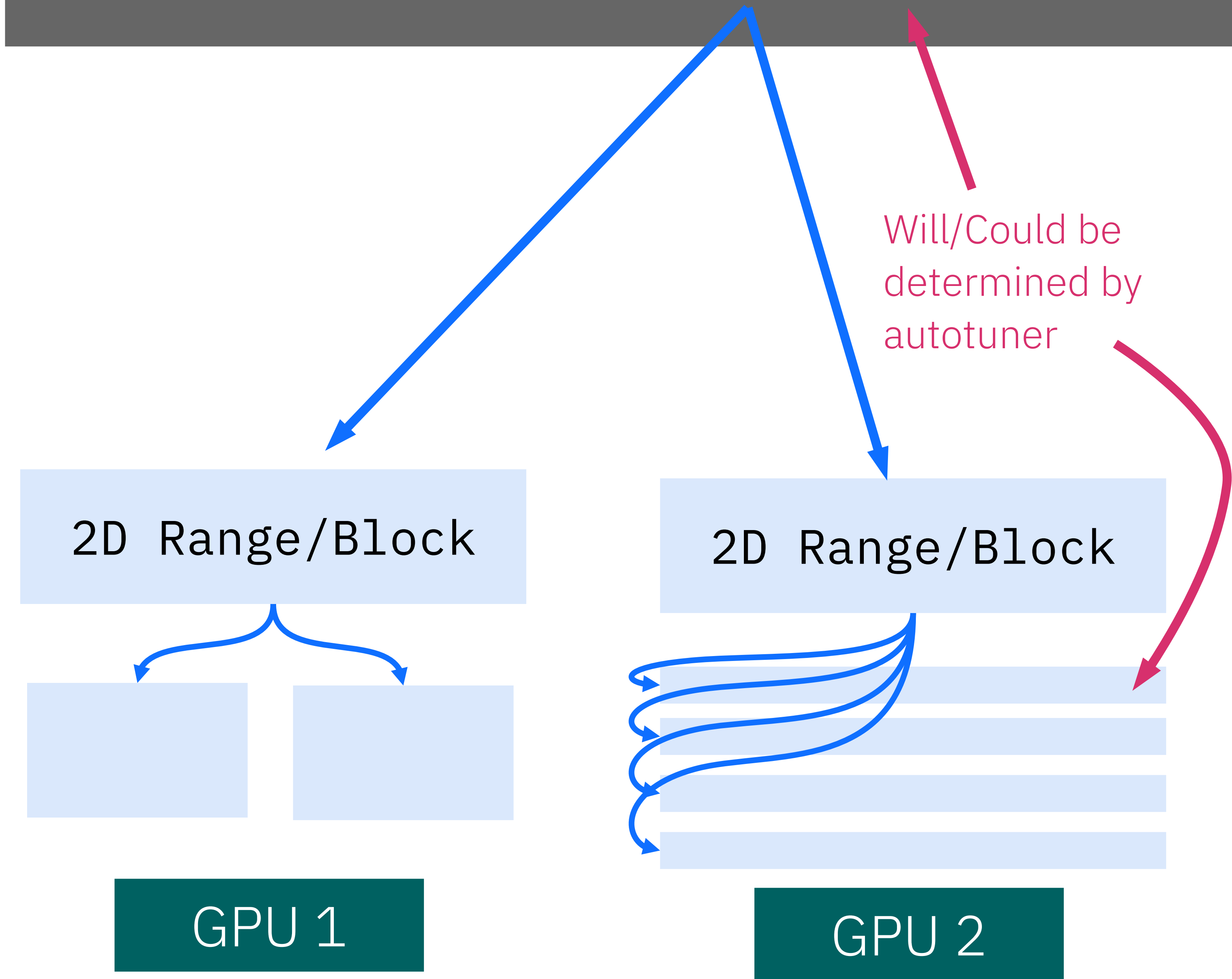
→ Goal: (1) Sketch the way to performance portability using the 100% open-source solutions

(2) ...getting you excited about the Triton Attention Backend in vLLM! 😉

 © 2026 IBM Corporation

Performance Portability:
What is it and how does Triton help?

- "Performance portability refers to the ability of computer programs and applications to operate effectively across different platforms." (wikipedia.org)
- In a Nutshell: Tritons tiled programming model & abstraction level:
  - Allows to express complicated optimizations *(we'll come to that…)*
  - While staying hardware agnostic
- Hyperparameters (called "configurations") allow simple but effective adaptations for different hardware (maybe also with **autotuning**)

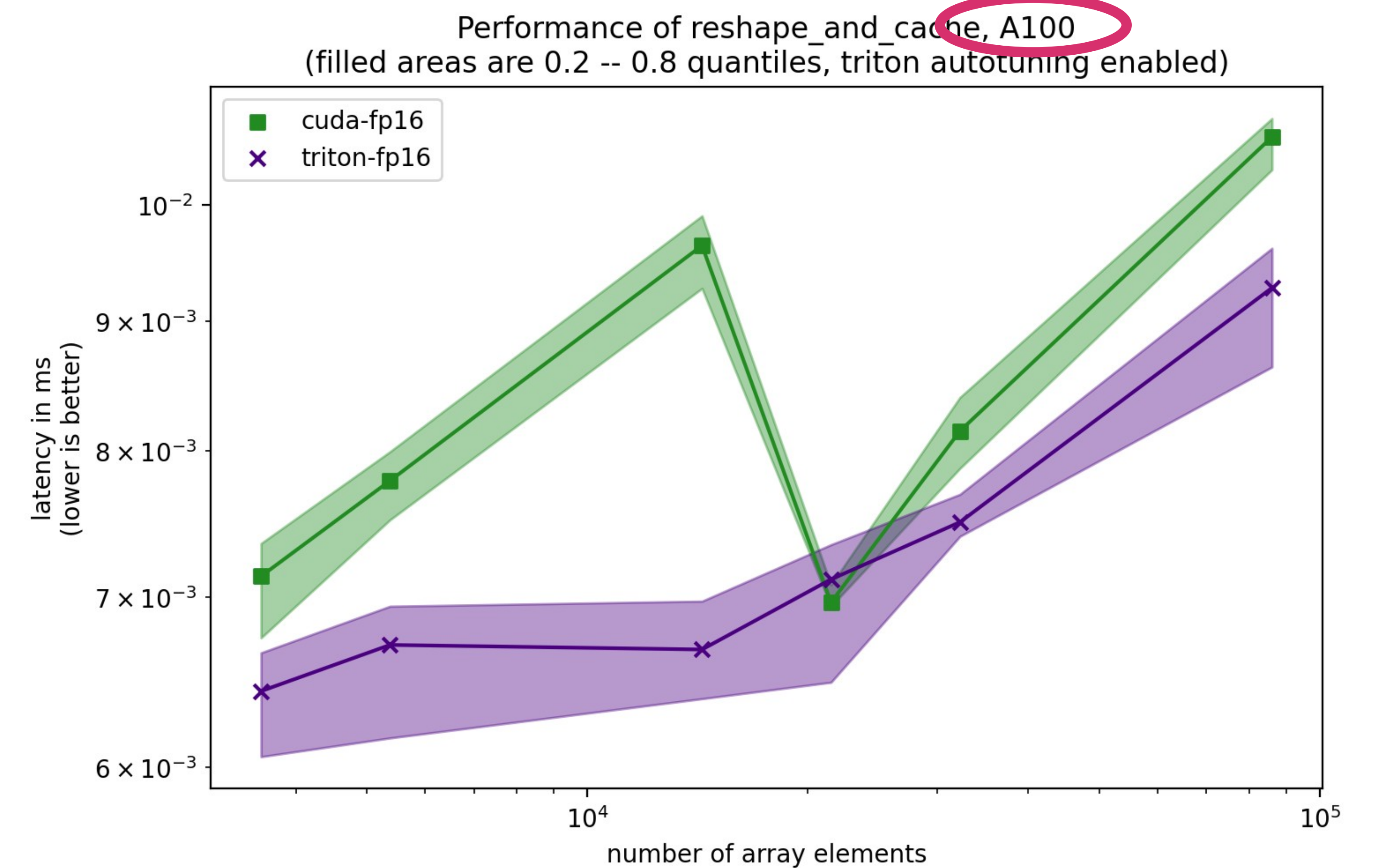


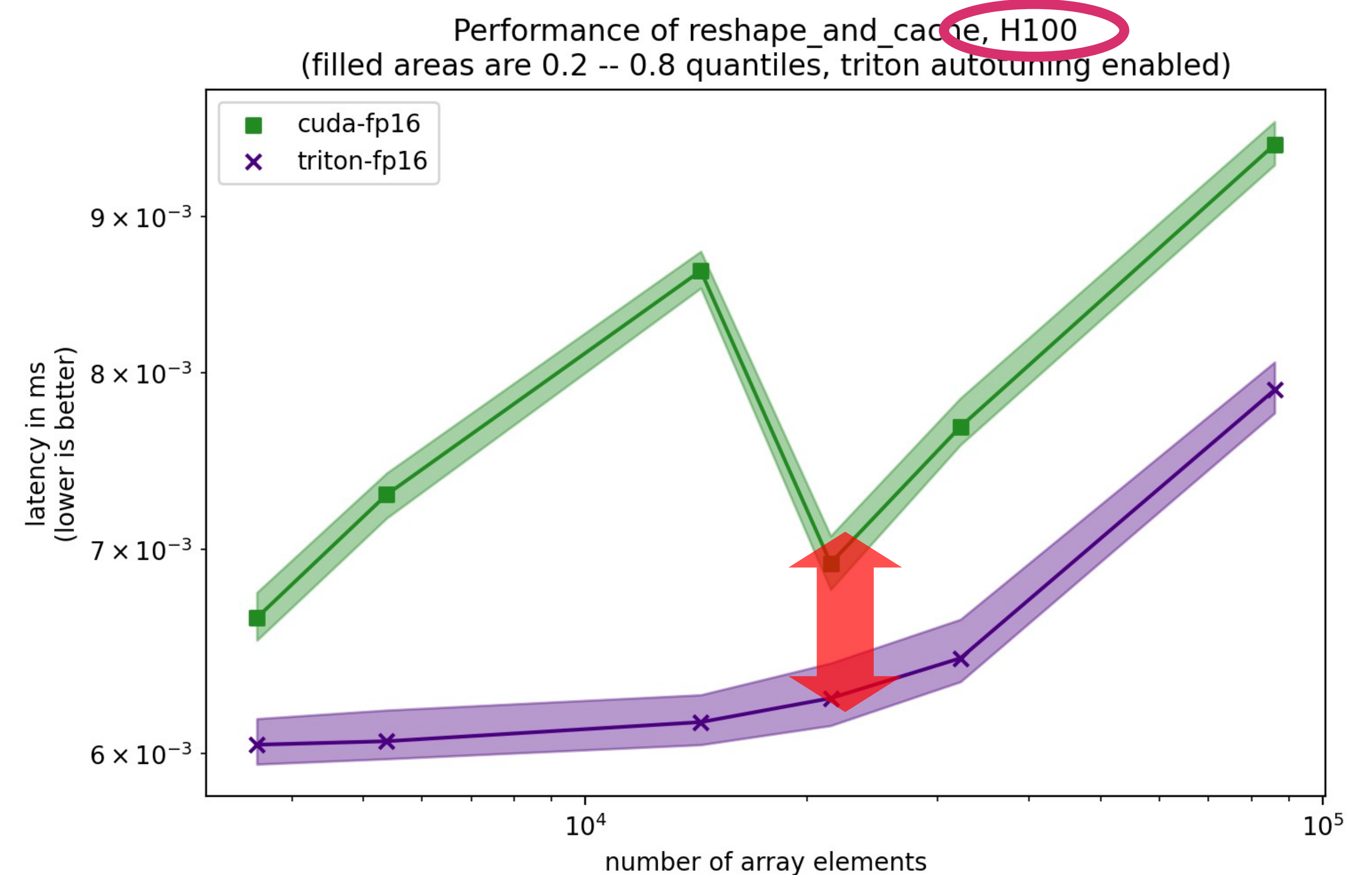
Limited performance portability? → one example

- Reshape and cache: simple vLLM kernel to populate the paged cache after linear layers
- Even within GPUs of the same vendor, CUDA kernels cannot achieve peak performance, without adaption
  - Here: Triton outperfomed CUDA by 16% (due to inferior blocking)

➔ Because writing high-performing code requires knowledge of the hardware (e.g. tensor core size, cache size, access patterns)

- ▸ Hardware changes → performance of same low-level code degrades
- ▸ Besides Just-in-Time compilation and autotuning can take more runtime information into account

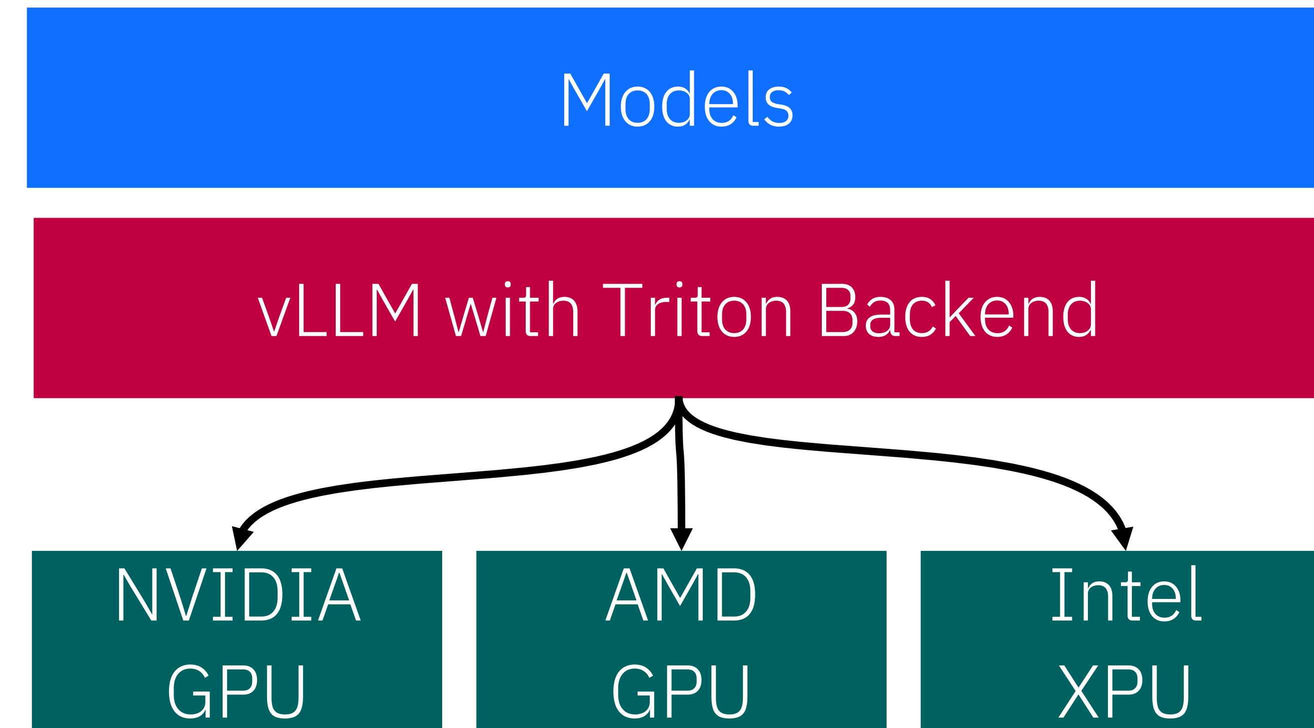




 © 2026 IBM Corporation

Introducing: `vllm-triton-backend`

- We (IBM Research & RedHat) built a production-ready vLLM attention backend written entirely in Triton
  - Contains multiple kernels for prefill & decoding (paged attention)
  - Packaged together with heuristics to adapt for different scenarios
- (a Backend in vLLM encapsulates different attention-implementations, usually wrappers for external libraries)
- → First (and only) triton-only backend in vLLM
- Runs on NVIDIA, AMD & Intel
  - "its always there"
- Delivering state-of-the-art performance
  - Is the default for AMD-based deployments, or GTP-OSS on pre-Hopper GPUs, etc.
  - (PRs of latest changes still pending)
- In the meantime: *a lot of help from the community!*
  (Thanks a lot!)



Installation:

```
# integrated in vllm
pip install vllm
         (no further dependencies)
```

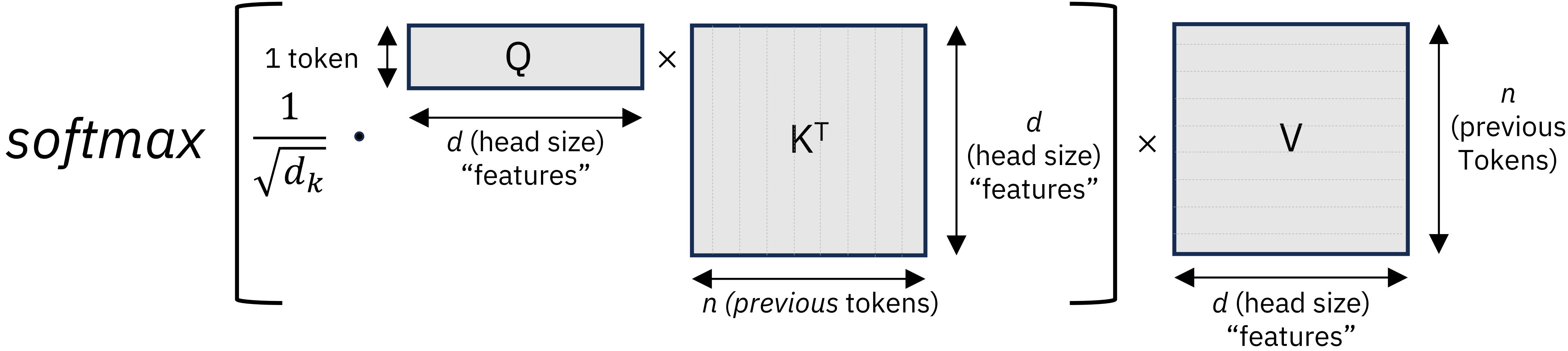
Run:

```
vllm serve ibm-granite/granite-4.0-small \
    --attention-backend=TRITON_ATTN
    # the configuration only necessary where it
      is not the default
```

 © 2026 IBM Corporation

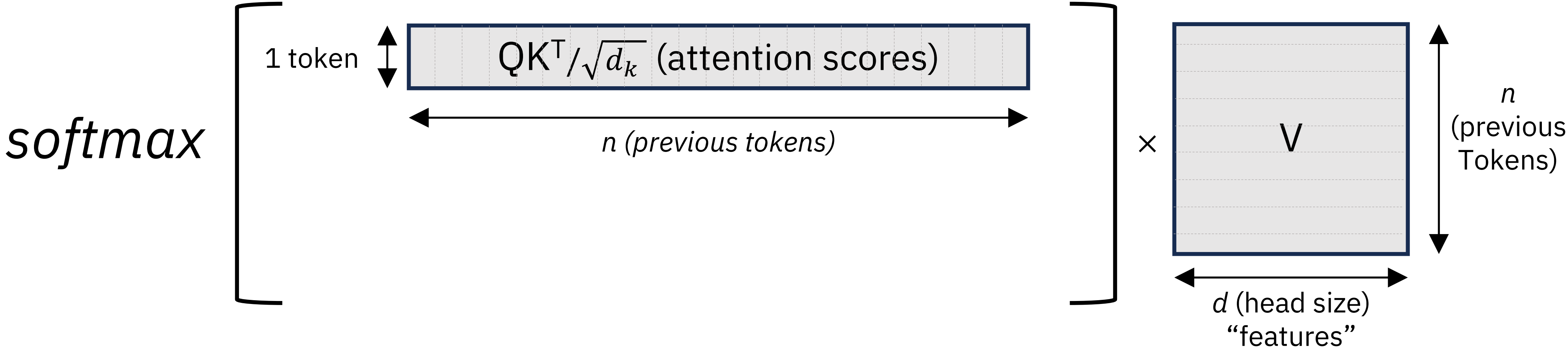
Attention

- Original attention formula: $$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right) V$$
- For a single *query token*, *sequence*, and *query head* and causal masking:



Attention

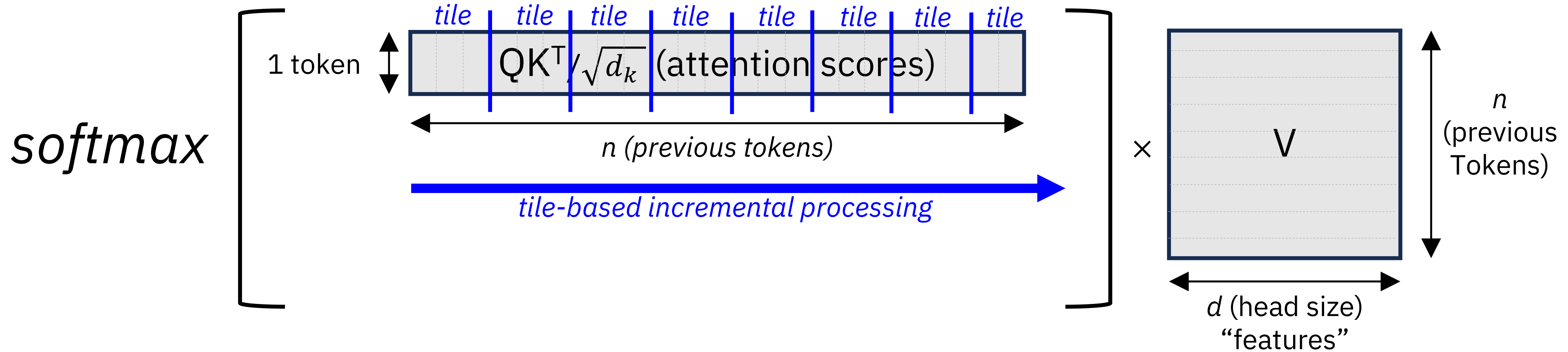
- Original attention formula: $$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right) V$$
- For a single *query token*, *sequence*, and *query head* and causal masking:



- Numerically stable softmax: $$softmax(s_i) = \frac{e^{s_i - max_j s_j}}{\sum_{k=1}^{n} e^{s_k - max_j s_j}}$$
- Softmax can be calculated efficiently using a *tiled* approach (a.k.a. *online softmax*)

Attention

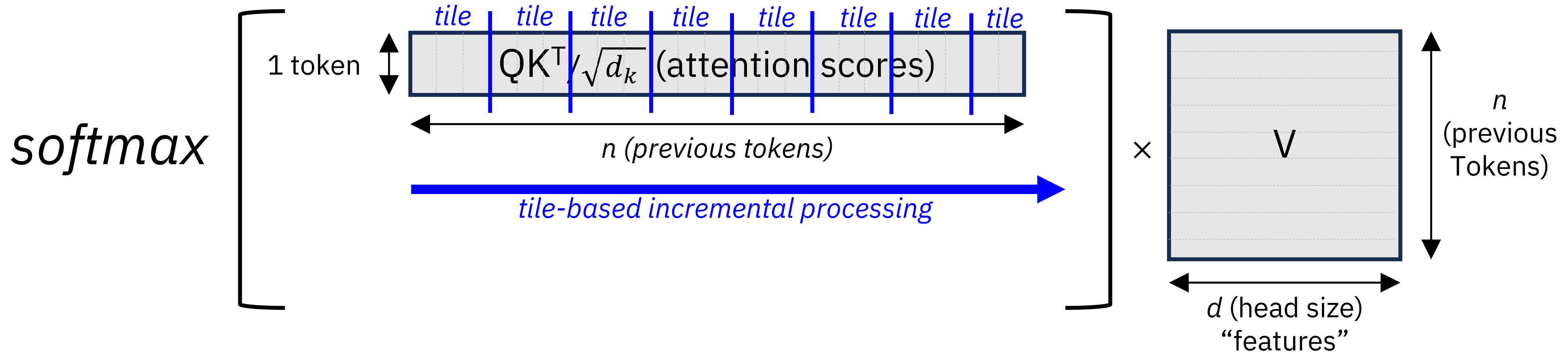
- Original attention formula: $$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right) V$$
- For a single *query token*, *sequence*, and *query head* and causal masking:



- Numerically stable softmax: $$softmax(s_i) = \frac{e^{s_i - max_j s_j}}{\sum_{k=1}^{n} e^{s_k - max_j s_j}}$$
- Softmax can be calculated efficiently using a *tiled* approach (a.k.a. *online softmax*)
  - split row into tiles, process incrementally
  - track running maximum and sum, rescale if maximum changes, normalize at the end

Attention

- Original attention formula: $$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right) V$$
- For a single *query token*, *sequence*, and *query head* and causal masking:



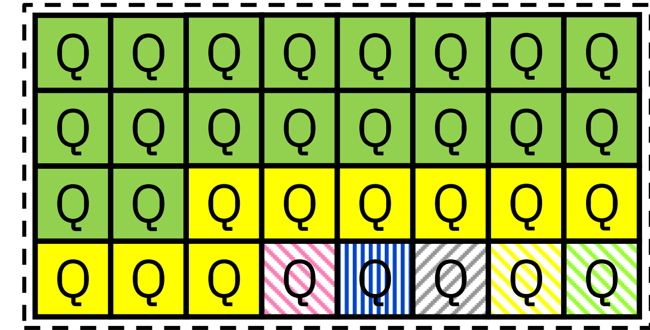
- Numerically stable softmax: $$softmax(s_i) = \frac{e^{s_i - max_j s_j}}{\sum_{k=1}^{n} e^{s_k - max_j s_j}}$$
- Softmax can be calculated efficiently using a *tiled* approach (a.k.a. *online softmax*)
  - *fused* with tiled matrix multiplications QK^T and V
  - core optimization in *FlashAttention*

Triton Attention Kernel

vLLM

- Example: scheduled *continuous batch* with chunked prefill
  - prefills (solid color)
  - decodes (hatched color)

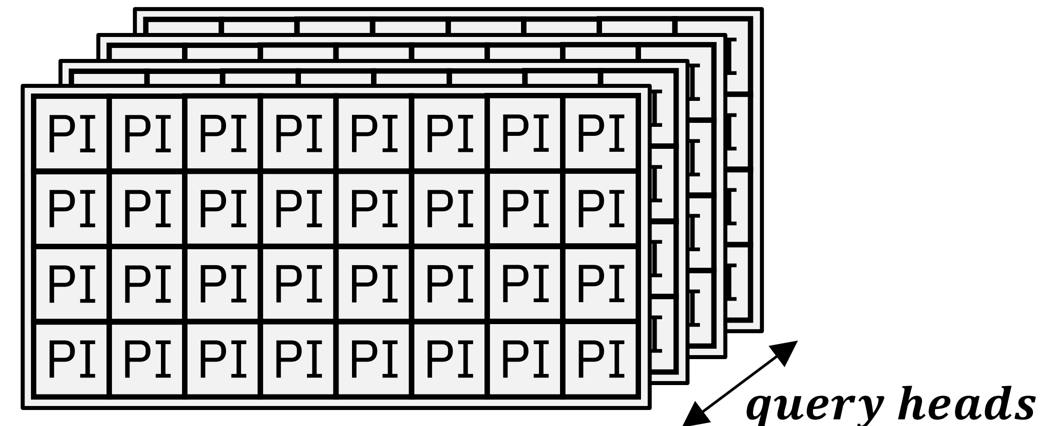
→ belong to different sequences with *separate KV contexts*



Naïve Approach

- Launch Triton attention kernel with grid: $\textit{query tokens} \times \textit{query heads}$
- Each *program instance* (PI)
  - handles a *single* query token and query head
  - attends over the KV context of its sequence

→ *redundant* KV reads across PIs
  - for query tokens from the same prompt
  - for query heads mapped to the same KV head (in case of Grouped Query Attention, *GQA*)

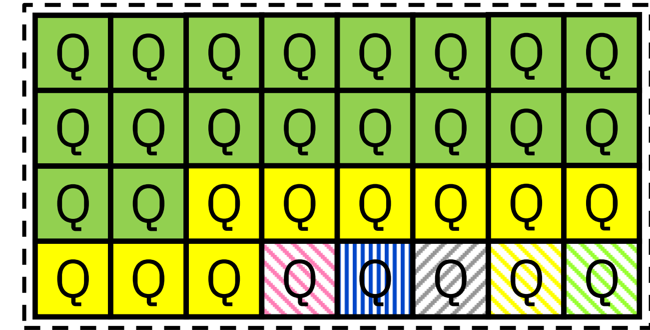


Triton Attention Kernel

vLLM

- Example: scheduled *continuous batch* with chunked prefill
  - prefills (solid color)
  - decodes (hatched color)

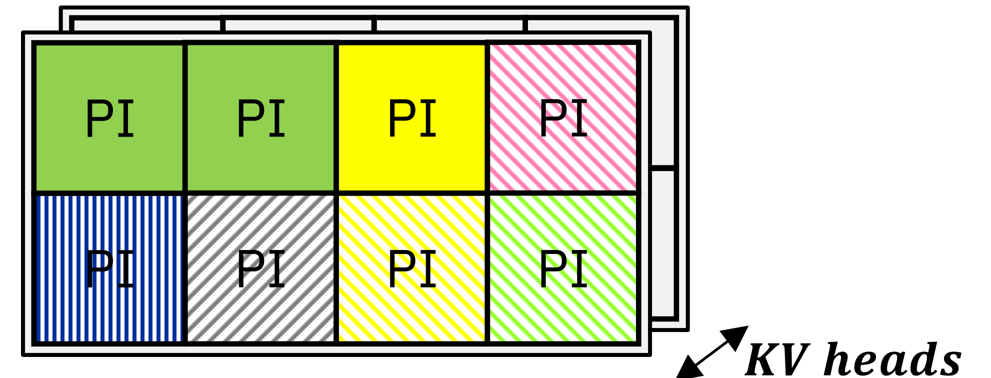
→ belong to different sequences with *separate KV contexts*



Q Block Optimization

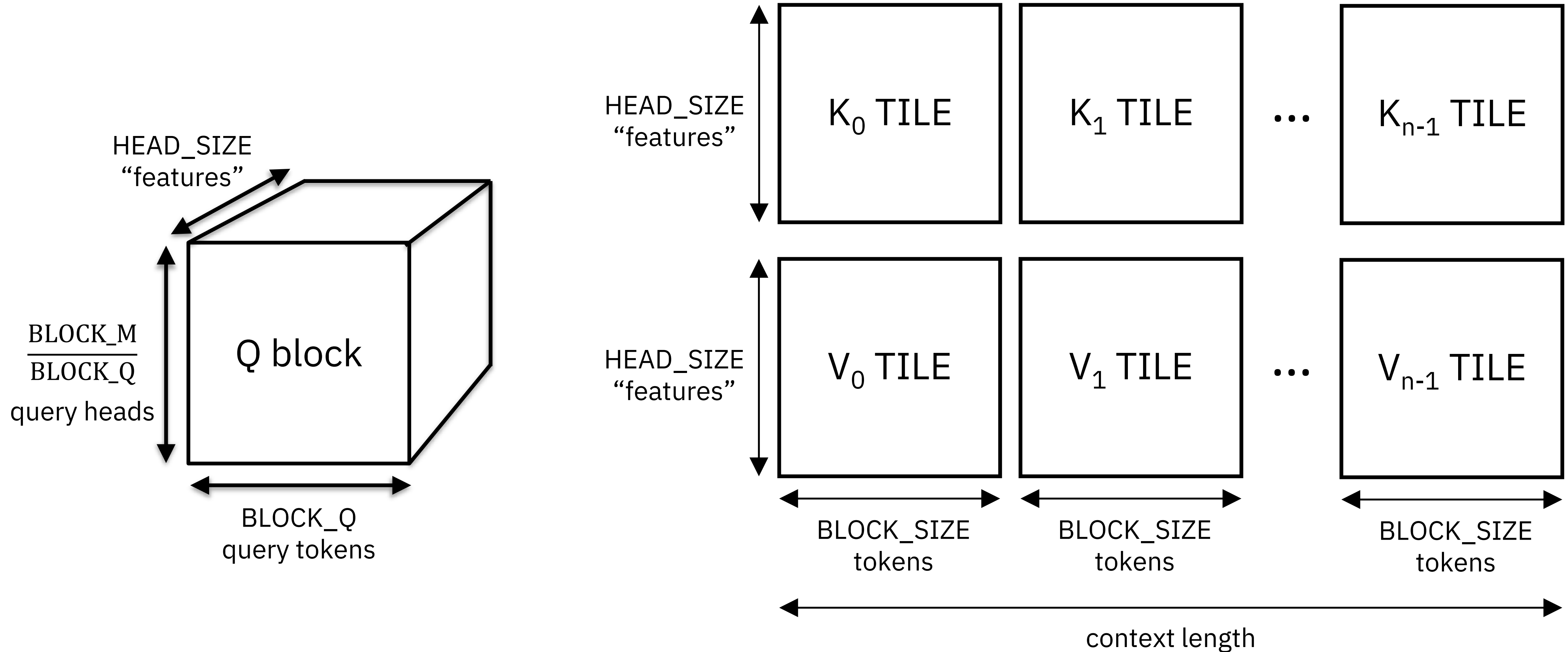
- Launch Triton attention kernel with grid: $\boldsymbol{Q\ blocks} \times \boldsymbol{KV\ heads}$
- Each *program instance* (PI) handles a *Q block*
  - multiple (prefill) or one (decode) query token(s)
  - multiple query heads mapped to the same KV head (*GQA*)
  - attends over the KV context of its sequence

→ better *KV reuse*, lower memory bandwidth



Triton Attention Kernel

- Each *program instance* (PI) handles one Q block of size $BLOCK_M$, covering
  - $BLOCK_Q$ query tokens (prefill) or 1 query token (decode)
  - $\frac{BLOCK_M}{BLOCK_Q}$ query heads mapped to the same KV head (GQA)



Triton Attention Kernel

Challenge: Few Q blocks Lead to Small Launch Grid $\boldsymbol{Q\ Blocks \times KV\ heads}$

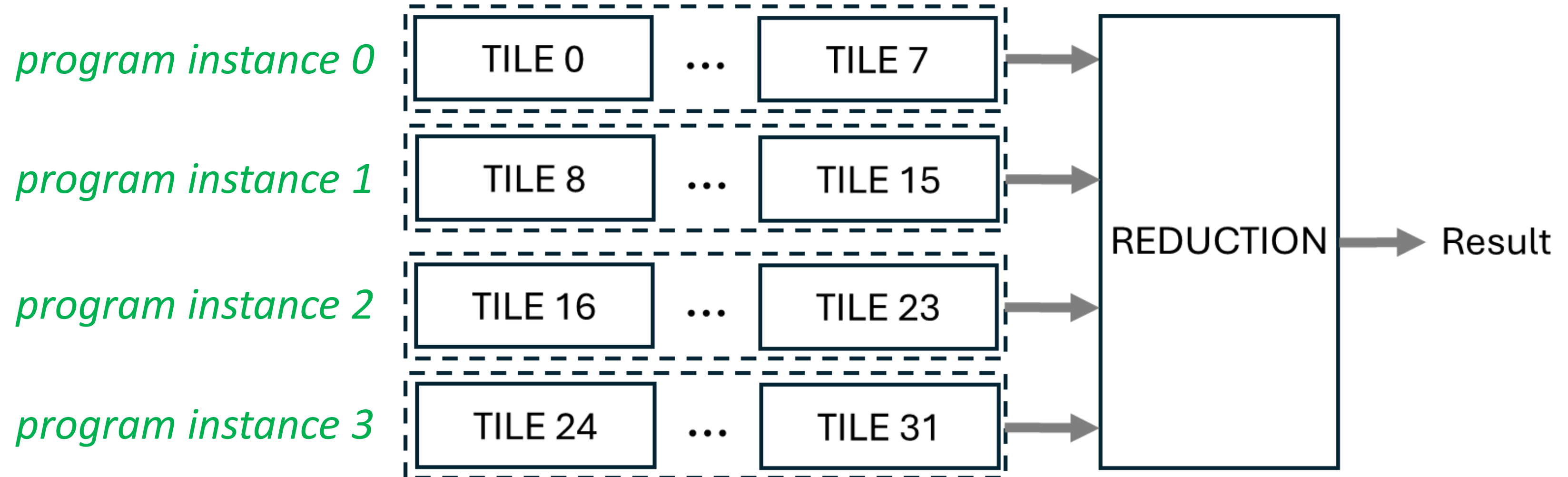
- Example: single-sequence decode involves one Q block per KV head
- Small *decode* batches → few program instances → *GPU underutilization*

Solution

- *Parallelize* tiled softmax across multiple program instances
  - effective for small batches and long sequences
  - key optimization in *FlashAttention* and *FlashDecoding*

- Multiple program instances
  - *sequential* tile processing within each instance
  - *parallel* tile processing across instances

→ requires *intermediate buffers* and a *reduction kernel*



Triton Attention Kernel

Kernels

- 2D: $\boldsymbol{Q\ blocks \times KV\ heads}$ exploits Q block parallelism
- 3D + Reduce: $\boldsymbol{Q\ blocks \times KV\ heads \times tiles}$ exploits tile parallelism when Q parallelism is low

$\rightarrow$ *heuristic-based* runtime selection

Tile Size

- Decoupled from $\boldsymbol{BLOCK_SIZE}$ (vLLM KV-cache paging granularity)
  - compute tiles independent of memory layout
  - enables tuning and support for hybrid/SSM models

Persistent 3D Kernel Implementation *(PR in progress)*

- Static launch grid: $\boldsymbol{SMs\ per\ KV\ head \times KV\ heads}$
  - enables *persistent* execution (e.g., H100: 132 SMs, MI300: 304 CUs)
- *Per-tile* work distribution (not per sequence)
  - tiles evenly distributed across SMs
  - CPU-generated scheduling tensors map each SM to (Q block, tile) ranges
- *Compact* intermediate buffers
  - only sequences split across SMs require additional buffer slots and reduction
  - $\boldsymbol{total\ slots \leq max\ batch\ size\ 3D + SMs\ per\ KV\ head}$

Performance Evaluation 2D Kernel

vLLM latency benchmark results
QWen3-32B on **H100**
batch-size=128, input length=500 tokens

FlashAttention v3
FlexAttention
TritonAttention Naïve
TritonAttention QB (2D)

normalized latency
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
0 100 200 300 400 500 600 700 800 900
output length [tokens]

vLLM latency benchmark results
QWen3-32B on **MI300**
batch-size=128, input length=500 tokens

FlexAttention
TritonAttention Naïve
TritonAttention QB (2D)

normalized latency
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
0 100 200 300 400 500 600 700 800 900
output length [tokens]

Performance Evaluation 3D Kernel

vLLM latency benchmark results
QWen3-32B on **H100**
batch-size=1, input length=500 tokens

FlashAttention v3
FlexAttention
TritonAttention QB (2D)
TritonAttention QB + ParTS (3D)

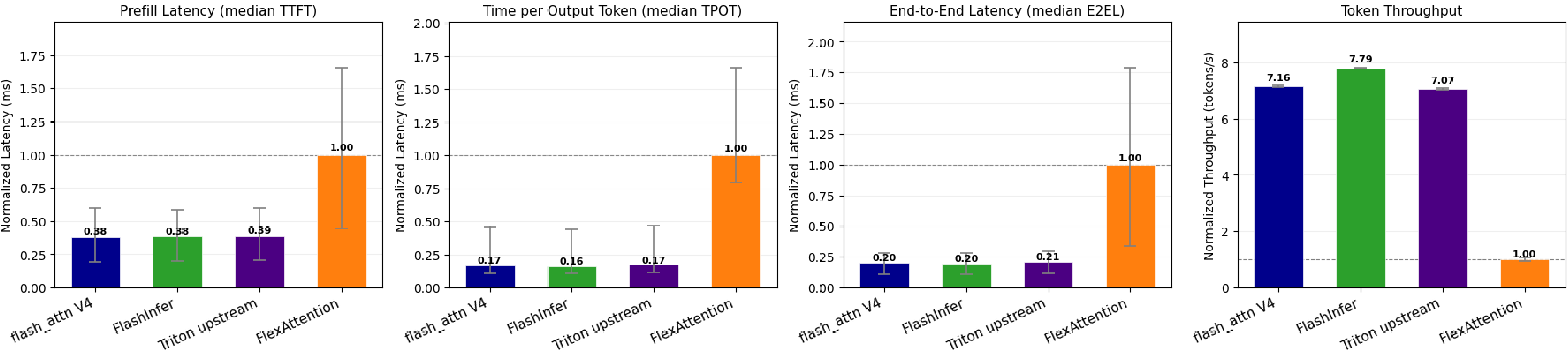
normalized latency
output length [tokens]

vLLM latency benchmark results
QWen3-32B on **MI300**
batch-size=1, input length=500 tokens

FlexAttention
TritonAttention QB (2D)
TritonAttention QB + ParTS (3D)

normalized latency
output length [tokens]

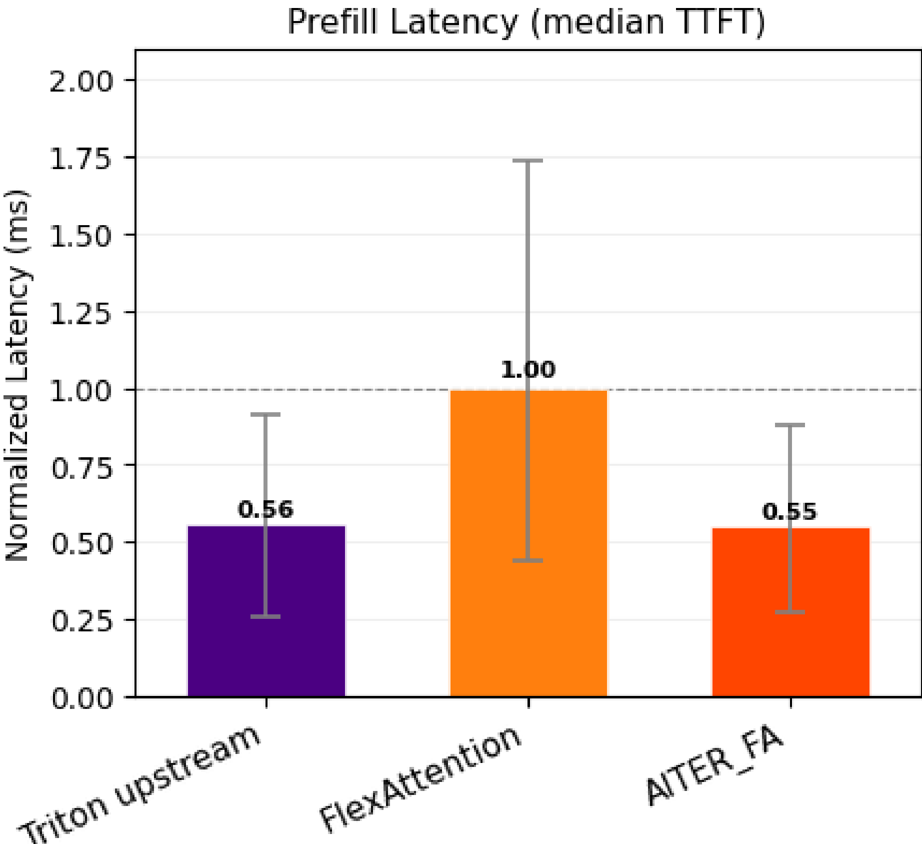
Measuring end-to-end with ShareGPT: B200

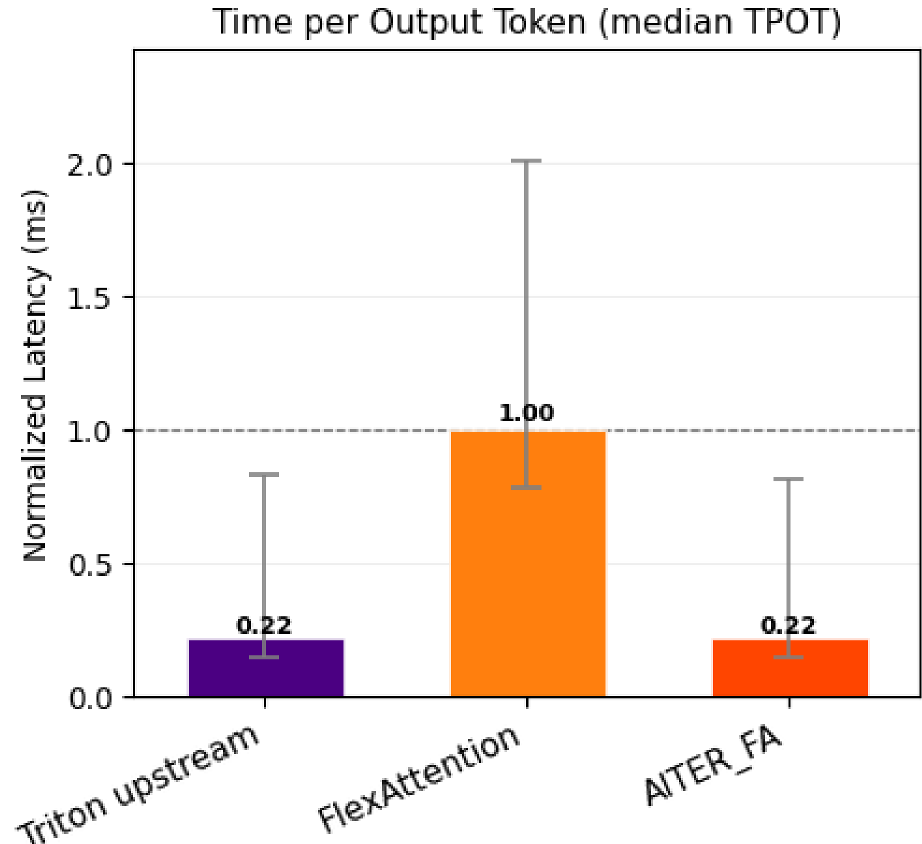


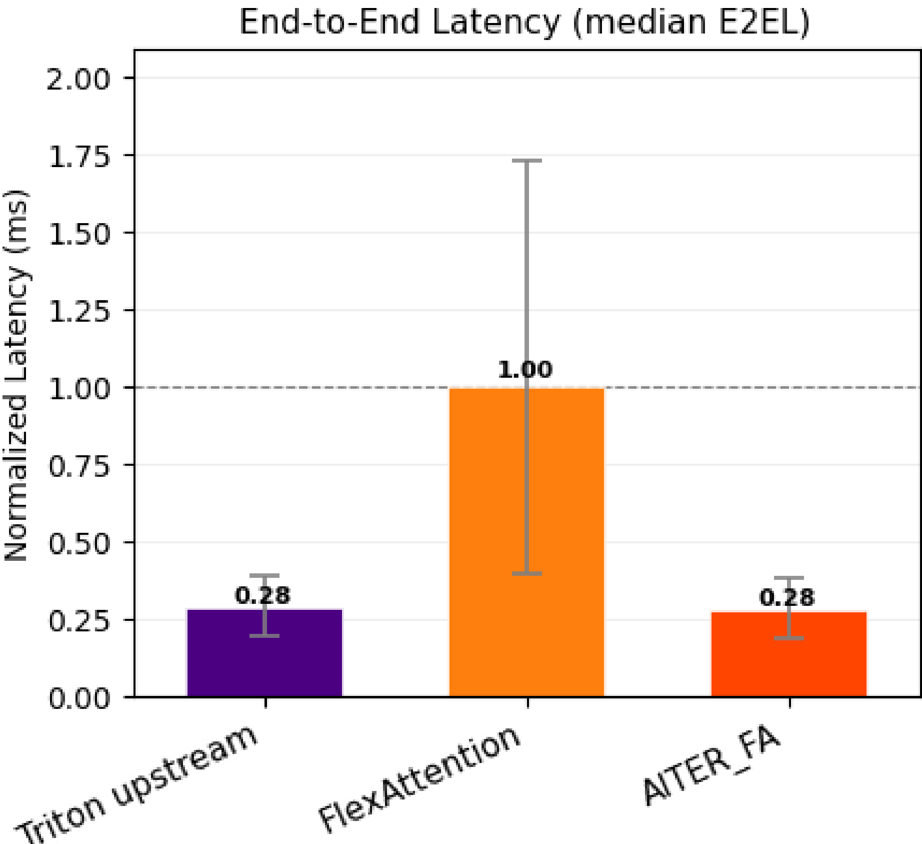
 © 2026 IBM Corporation

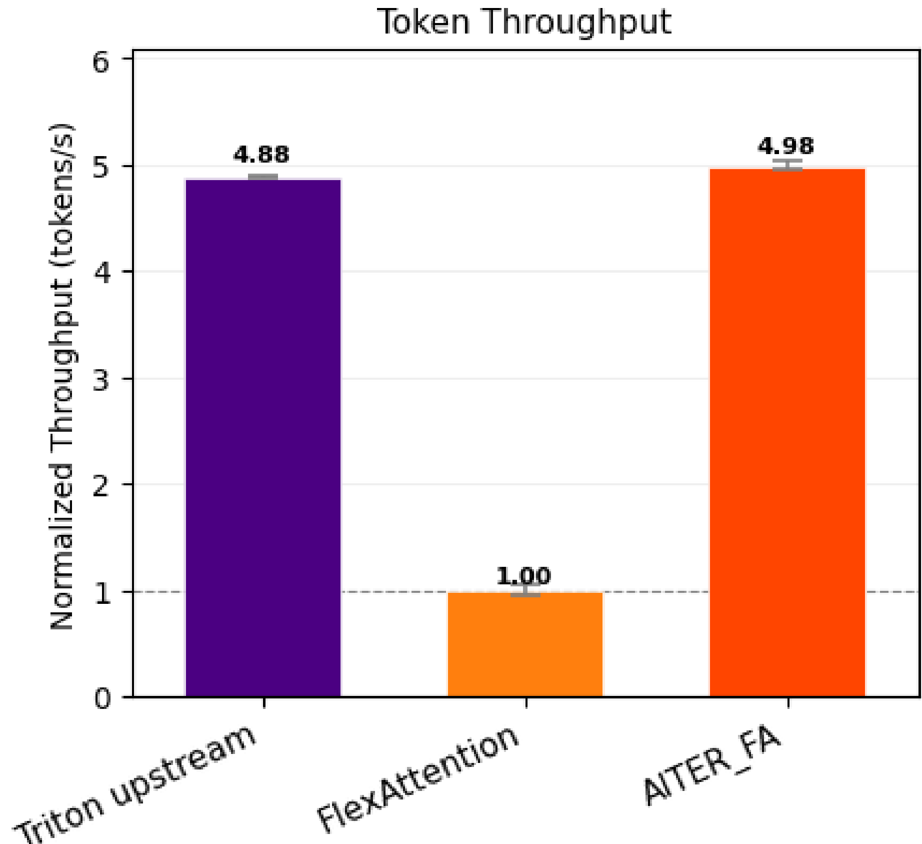
Measuring end-to-end with ShareGPT: MI300X

vllm bench serve with ShareGPT dataset — Qwen3-32B on MI300X
(normalized to FlexAttention)
(end2end benchmarking of vLLM, no prefix caching)









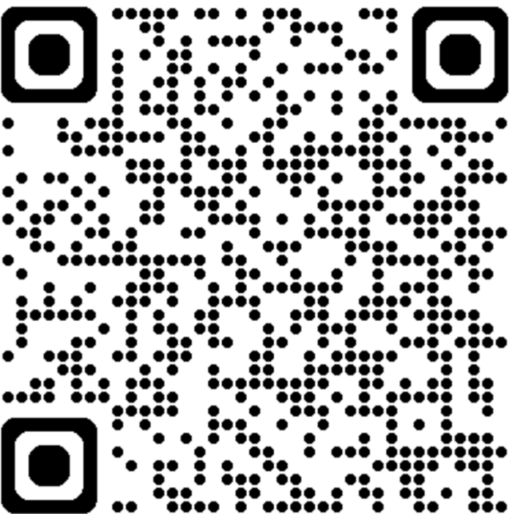
- errorbars are p20/p80 for latencies, min/max for throughput
- all values are average of 3 measurements

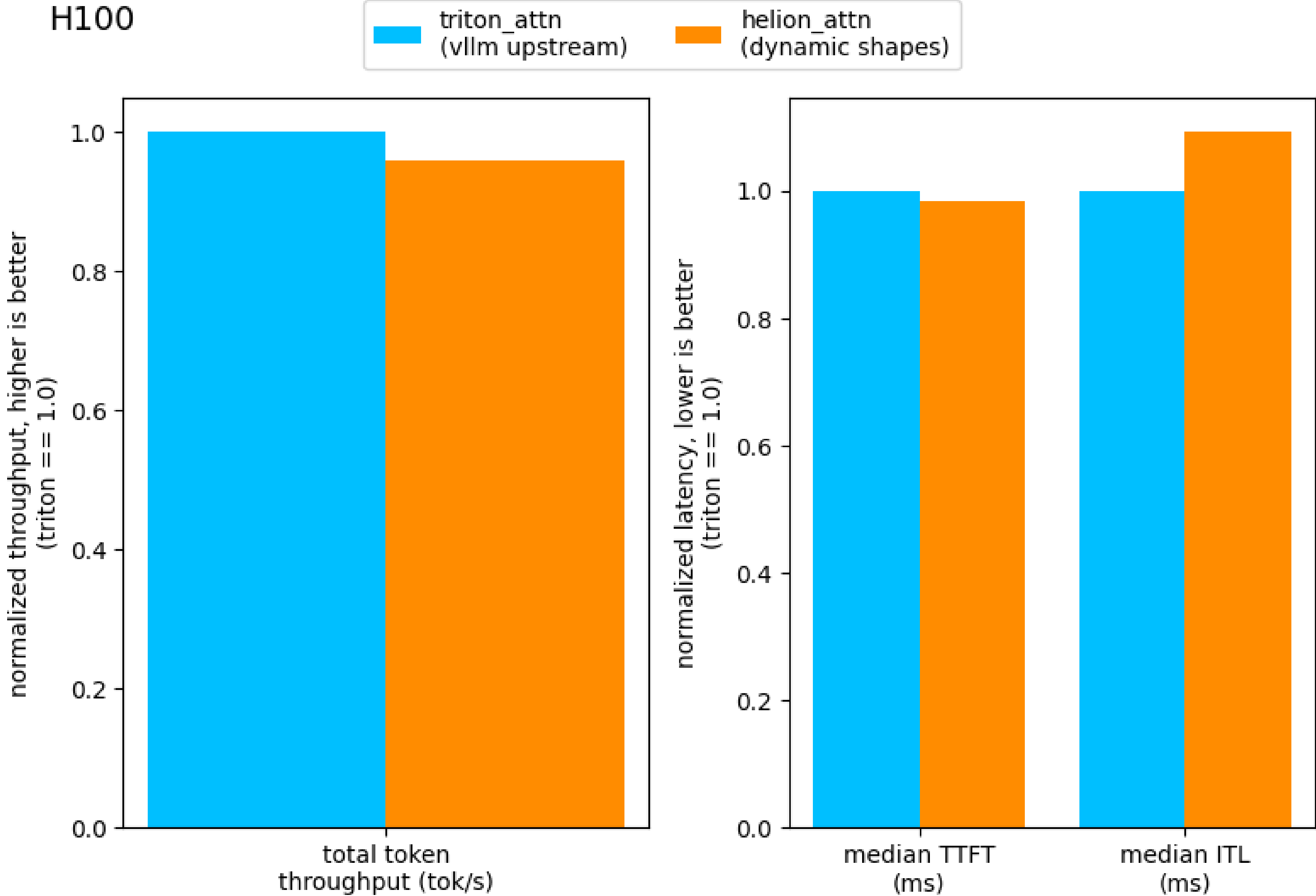
warmup: 2 iterations

 © 2026 IBM Corporation

Outlook: Helion, the new kid on the block?

- Helion as new DSL from pytorch
- “Tiled pytorch” or “higher-level Triton”
- We implemented a (simple) paged attention kernel in Helion
  - → https://pytorch.org/blog/portable-paged-attention-in-helion/
  - Already getting close....
- Code as vLLM draft PR#27293





 © 2026 IBM Corporation

Conclusion?

- We developed a triton-only feature-complete attention backend for vLLM
  - that achieves state-of-the-art performance on AMD & NVIDIA (and also runs on Intel)
  - with a single-source kernel
- Community-maintained kernels in an open-source high-level language help to "catch the train" in LLM kernel development by enabling performance portability

```
pip install vllm
# no further
#  dependencies
```

- Big thanks to the team: Thomas Parnell, Chih-Chieh Yang, Sara Kokkila Schumacher, Sage Moore, Lukas Wilkinson, Luka Govedič

Want to know more?

- **Paper**: → With all details and further lessons learned

→ Paper: *"The Anatomy of a Triton Attention Kernel"*

- Pytorch blog:
  - pytorch.org/blog/portable-paged-attention-in-helion
  - pytorch.org/blog/enabling-vllm-v1-on-amd-gpus-with-triton
- VLLM blog: vllm.ai/blog/vllm-triton-backend-deep-dive
- ***... Contribute!*** (Play around...and we/vLLM welcomes PRs)

...looking forward to all your questions!

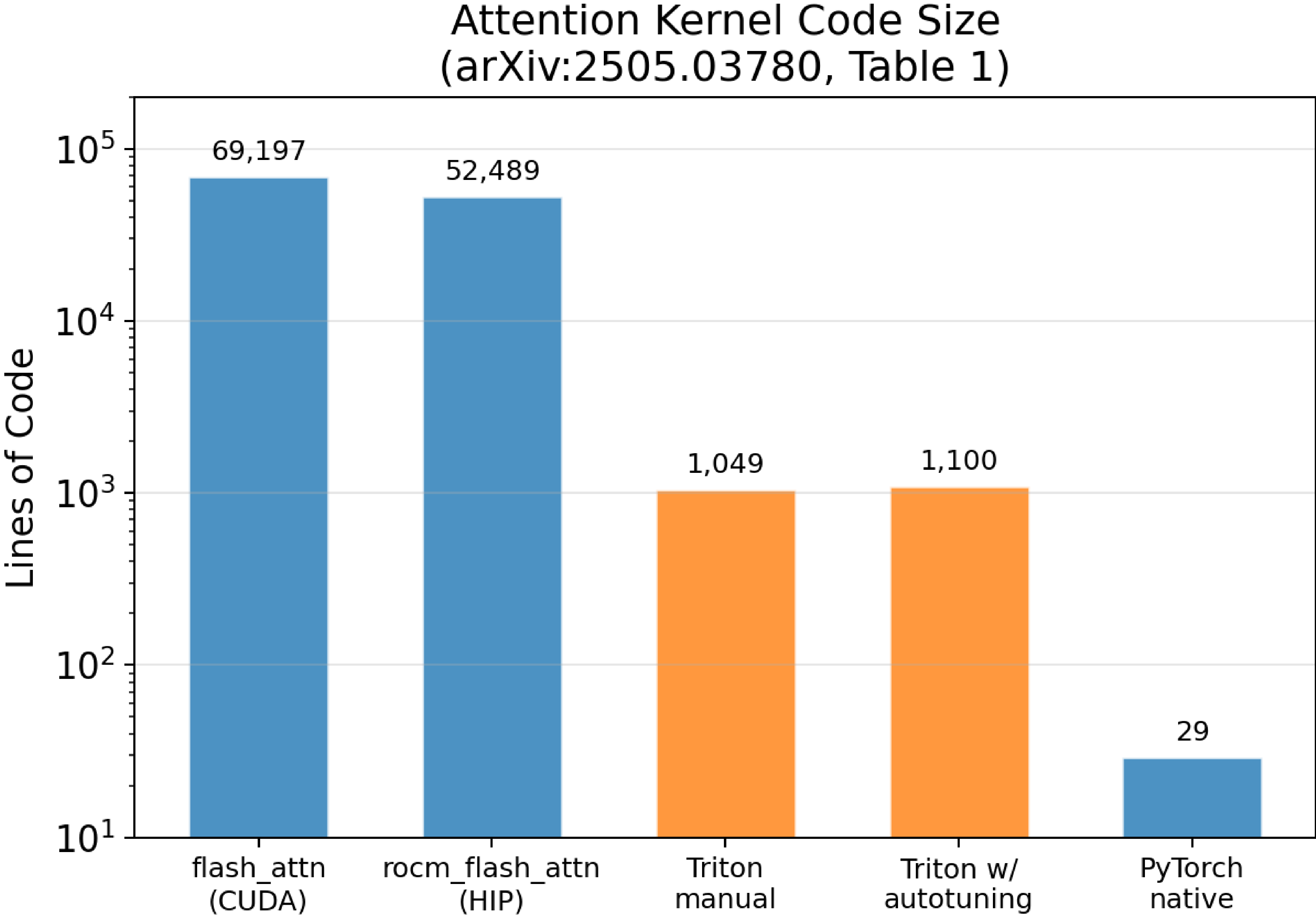
Dr. Burkhard Ringlein Dr. Jan van Lunteren

 © 2026 IBM Corporation

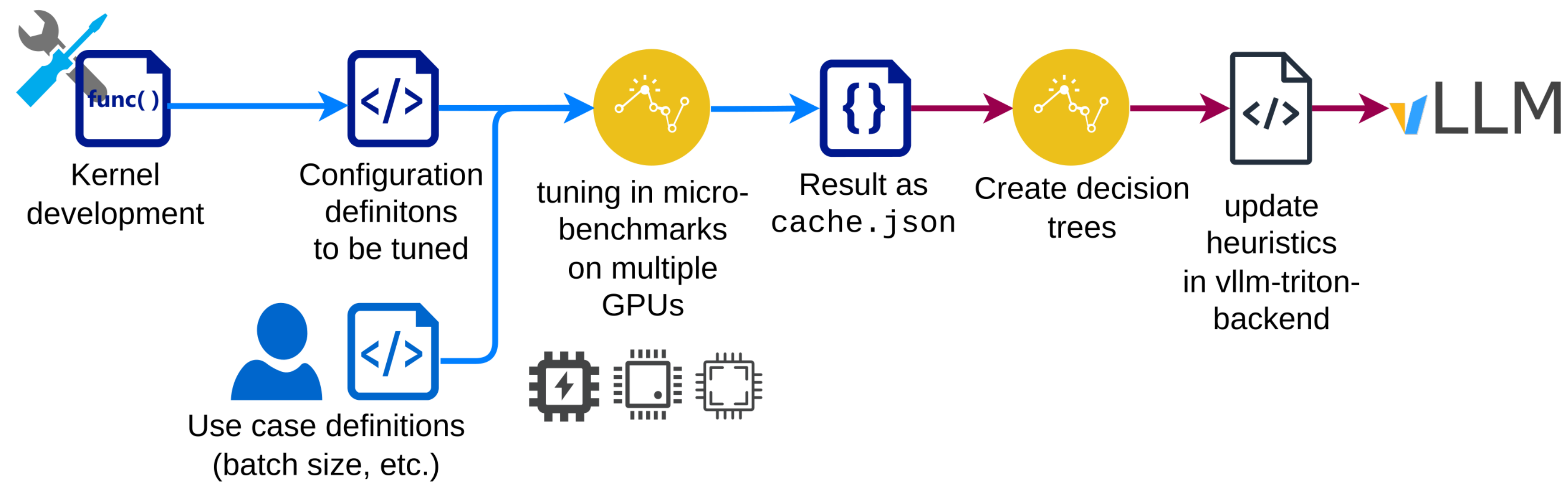
Appendix

 © 2026 IBM Corporation

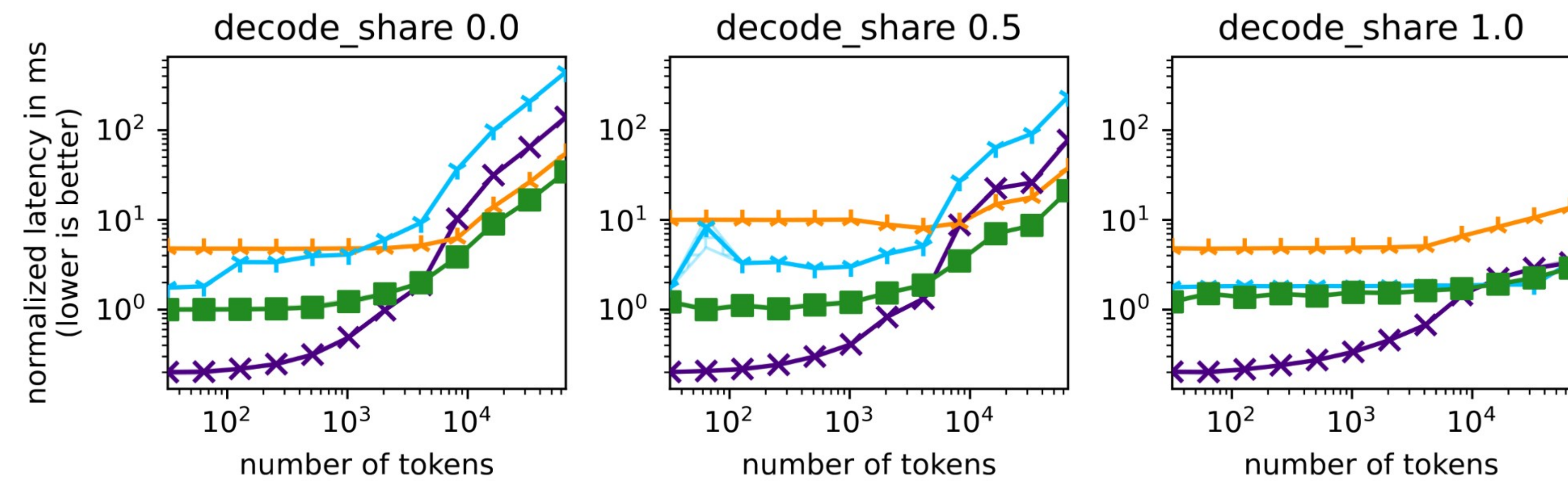
Portability in LoC…



 © 2026 IBM Corporation




```python
if torch.version.hip:
    TILE_SIZE_PREFILL = 64
    NUM_STAGES_PREFILL = 1
    NUM_WARPS_DECODE_3D = 4
else:  # cuda platform
    TILE_SIZE_PREFILL = 16
    NUM_STAGES_PREFILL = 4
    NUM_WARPS_DECODE_3D = 2
```



Key-Insights: Autotuning & Heuristics

- Tuning block and tile sizes for different models & GPUs can increase performance
- However:
  - Tuning still has a lot of overhead (even if done ahead-of-time)
  - Looking up best config before every kernel call adds latency
  - Ahead-of-time tuning can barley cover all possible use cases
- → Train heuristics based on extensive micro-benchmarks
  - Simplifies kernel launch
  - Generalizes well to unseen workloads
  - (simple if-else trees for now)
  - Compatible with CUDA graphs
  - Use extensive Micro-benchmarks
- Future work: Implement ML-based decision Trees

 © 2026 IBM Corporation

IBM