Neural Magic | Red Hat | vLLM

Low-Precision Inference in vLLM

Tyler Michael Smith

Member of Technical Staff, Red Hat
vLLM Committer

Lucas Wilkinson

Principal Software Engineer, Red Hat
vLLM Committer

Outline

- vLLM Introduction
- Linear Layers and Quantization
- CUTLASS and Epilogues
- Machete Deep Dive

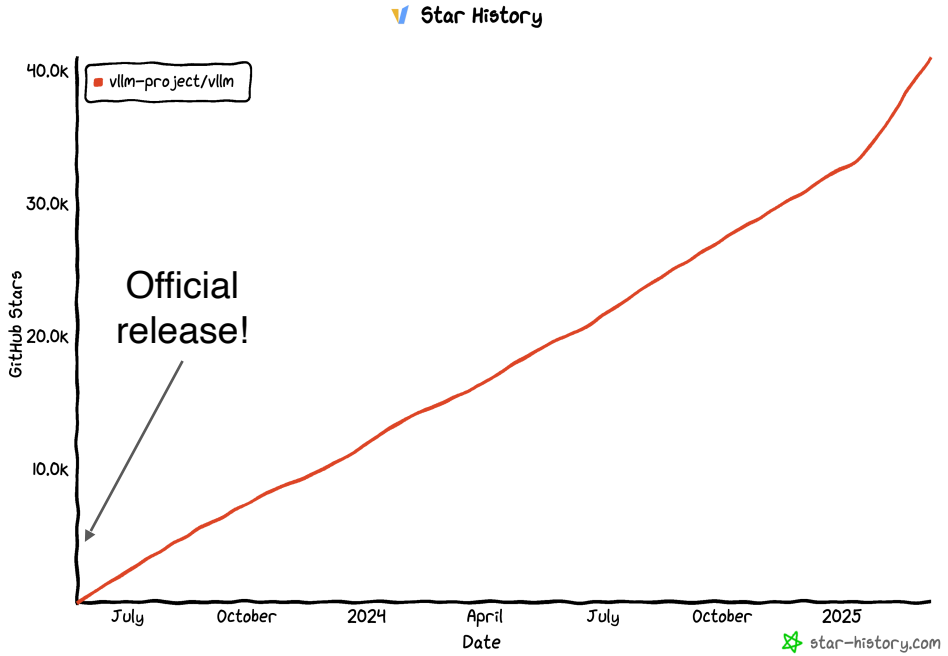
What is vLLM

vLLM Today

https://github.com/vllm-project/vllm

```
$ pip install vllm
```

41.9K Stars



vLLM API (1): `LLM` class

A Python interface for offline batched inference

```python
from vllm import LLM

# Example prompts.
prompts = ["Hello, my name is", "The capital of France is"]
# Create an LLM with HF model name.
llm = LLM(model="meta-llama/Meta-Llama-3.1-8B")
# Generate texts from the prompts.
outputs = llm.generate(prompts) # also llm.chat(messages)
```

vLLM API (2): OpenAI-compatible server

A FastAPI-based server for online serving

Server

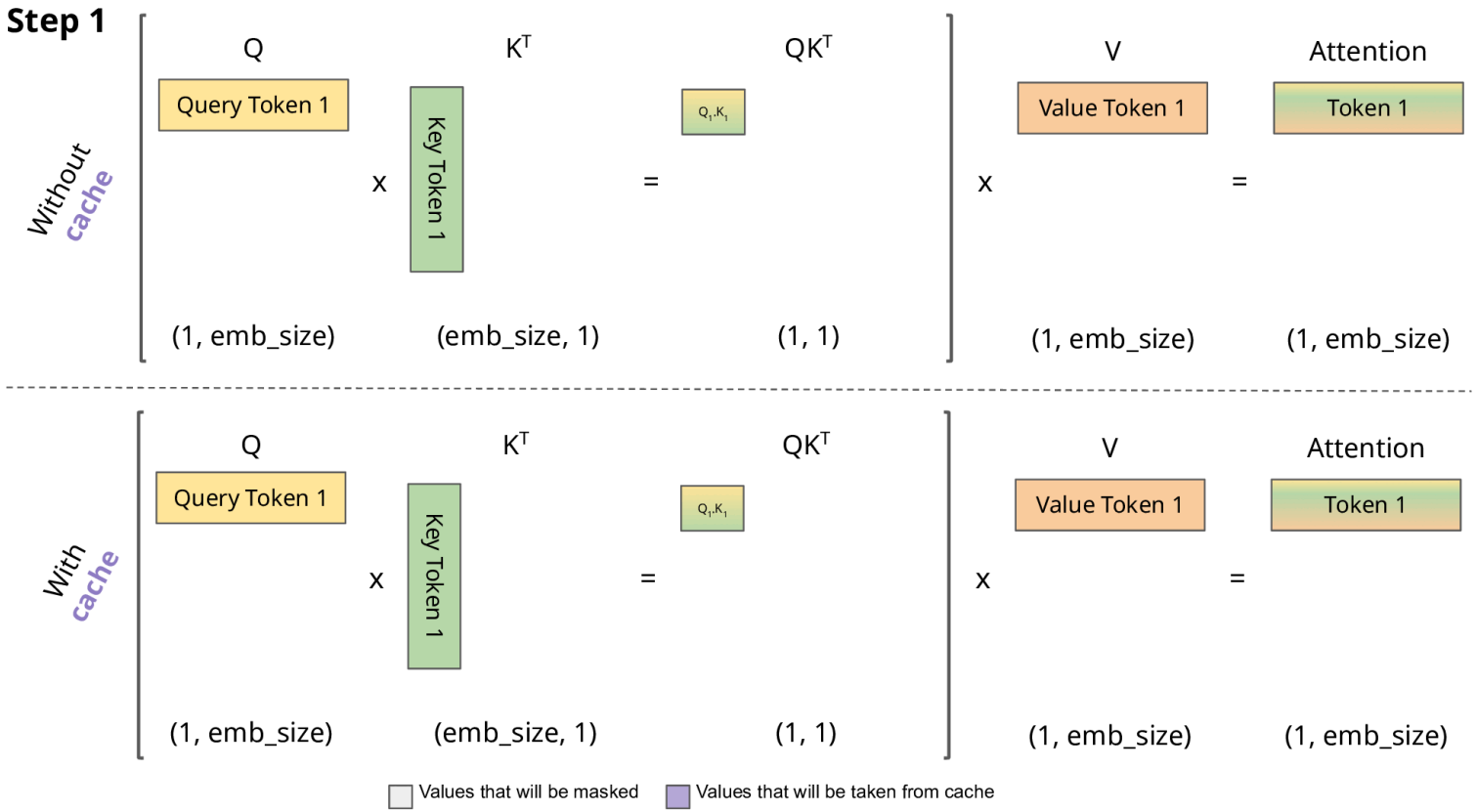
```
$ vllm serve meta-llama/Meta-Llama-3.1-8B
```

Client

```
$ curl http://localhost:8000/v1/completions \
    -H "Content-Type: application/json" \
    -d '{
        "model": "meta-llama/Meta-Llama-3.1-8B",
        "prompt": "San Francisco is a",
        "max_tokens": 7,
        "temperature": 0
    }'
```

KV Cache

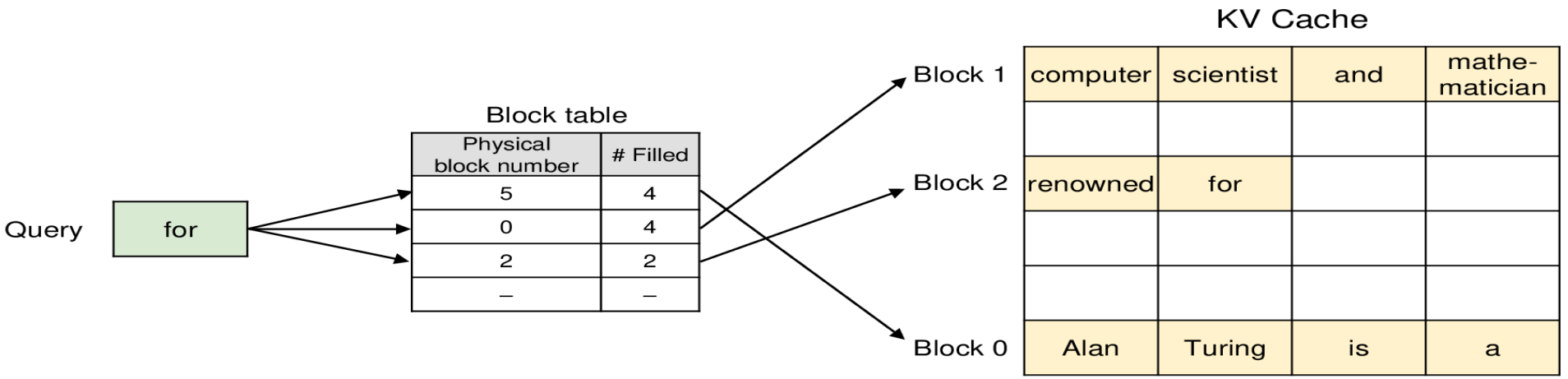
Caching Key and Value vectors in self-attention saves redundant computation and accelerates decoding



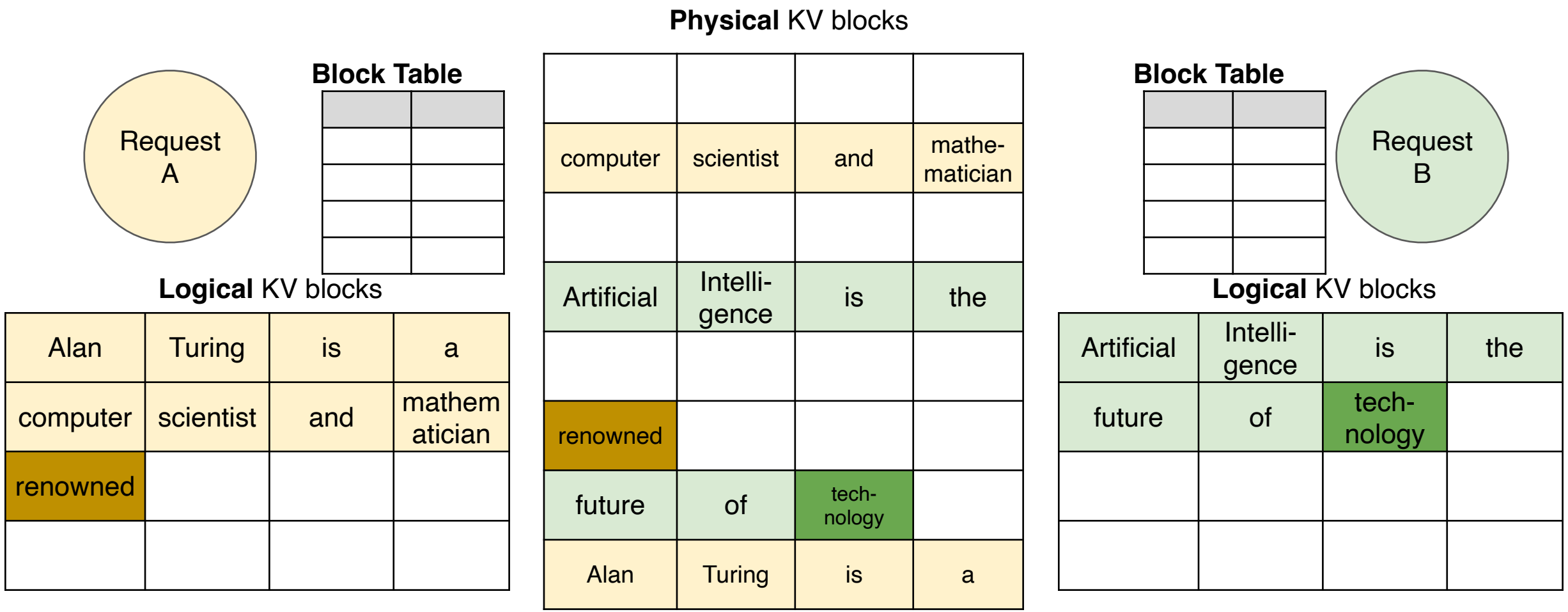
https://medium.com/@joaolages/kv-caching-explained-276520203249

PagedAttention

An attention algorithm that allows for storing continuous keys and values in non-contiguous memory space.



Manage KV cache like OS virtual memory



Linear Layers in vLLM

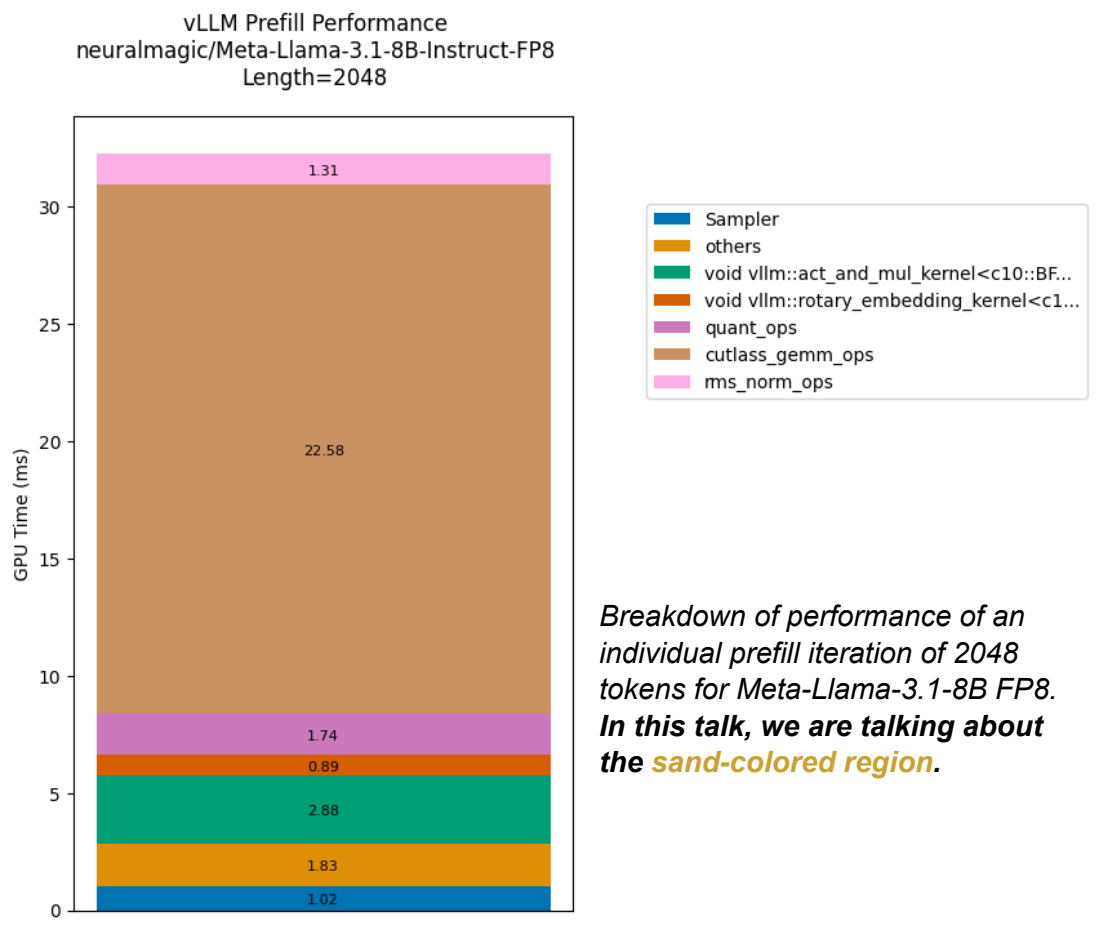
Why Quantize Your Model?

Reduce Memory Footprint

- *Weights are most of the VRAM usage for reasonable context lengths!*

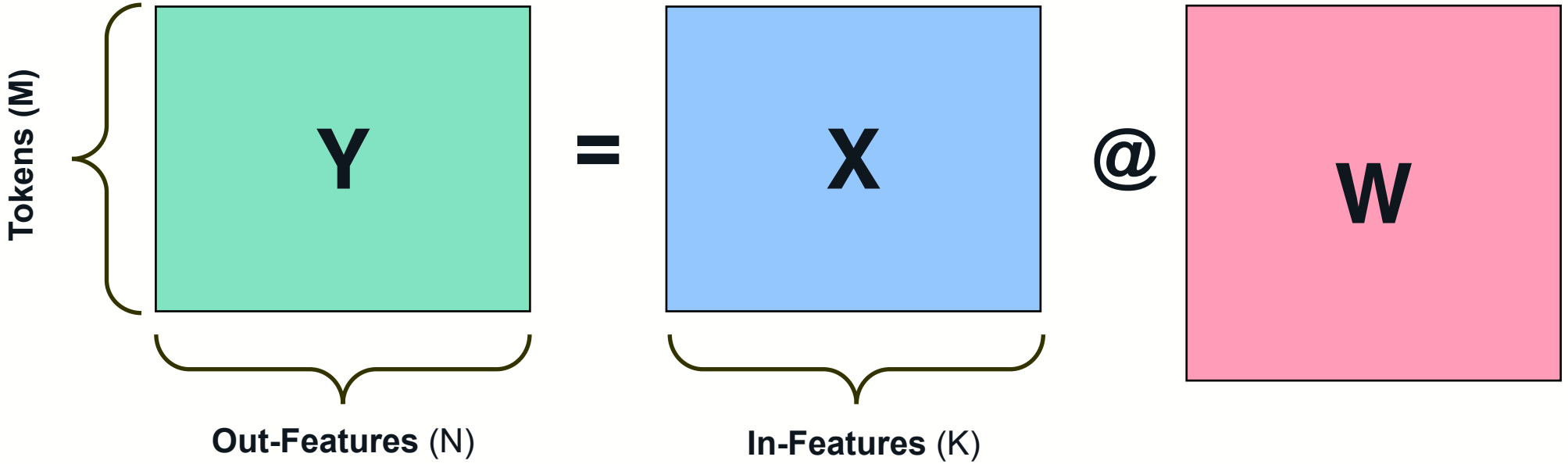
Speed up the linear layers, where we spent the majority of the time

- *Caveats: Attention becomes more important at long context lengths!*



Breakdown of performance of an individual prefill iteration of 2048 tokens for Meta-Llama-3.1-8B FP8. ***In this talk, we are talking about the sand-colored region.***

Background - Linear Layers

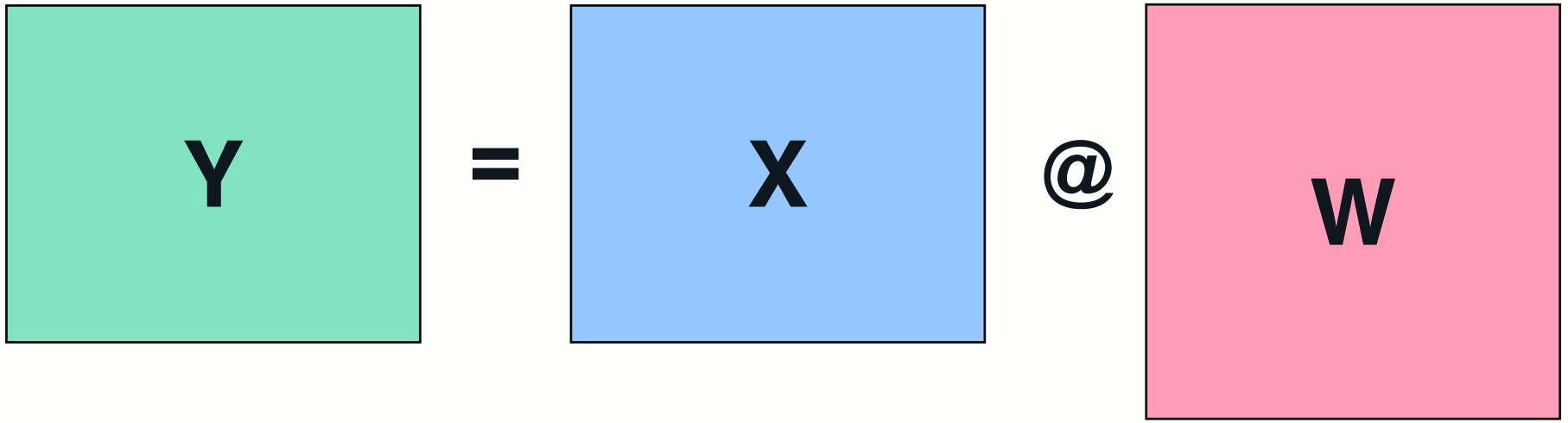


Total FLOPs = 2 * M * K * N

Total Bytes = 2 * (M * N + M * K + K * N)

- O(N^3) Compute
- O(N^2) Data movement
- What do we need to optimize?
- Depends on the size of the problem!

Background - Prefill



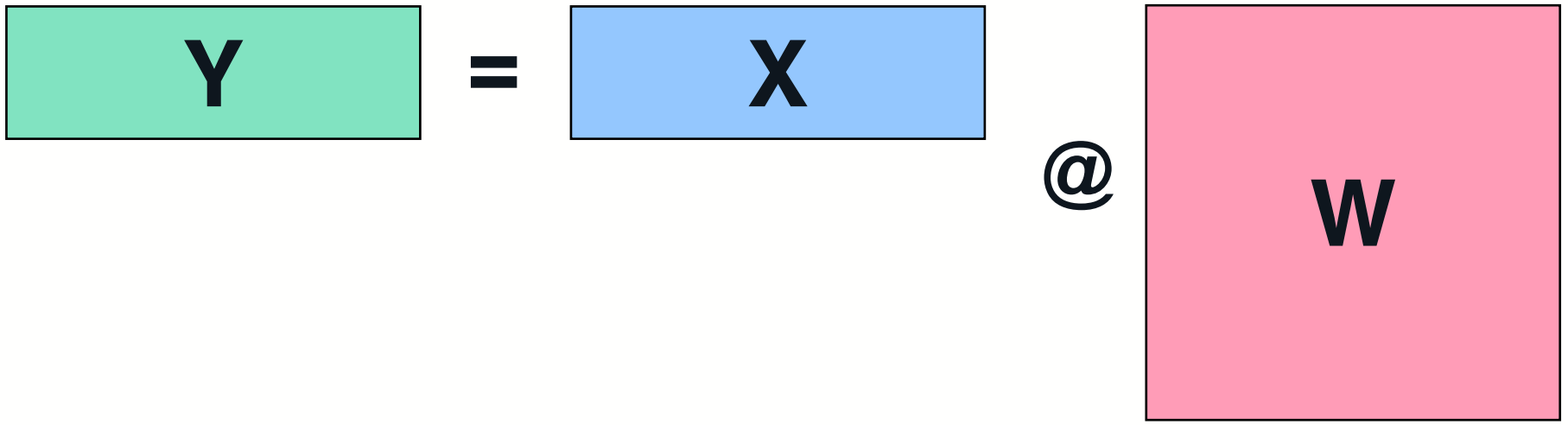
Total FLOPs = 2 * M * K * N

Total Bytes = 2 * (M * N + M * K + K * N)

Prefill GEMMs:

- M is large
- Flops ≫ Bytes read
- Focus on rate of computation!

Background - Decode



Total FLOPs = 2 * M * K * N

Total Bytes = 2 * (M * N + M * K + K * N)

Decode GEMMs:

- M is small
- Flops ≈ Bytes read
- Focus on reducing data movement!

Ecosystem Compatible

- Various quantization formats (AutoAWQ, AutoGPTQ, GGUF) used by the OSS ecosystem

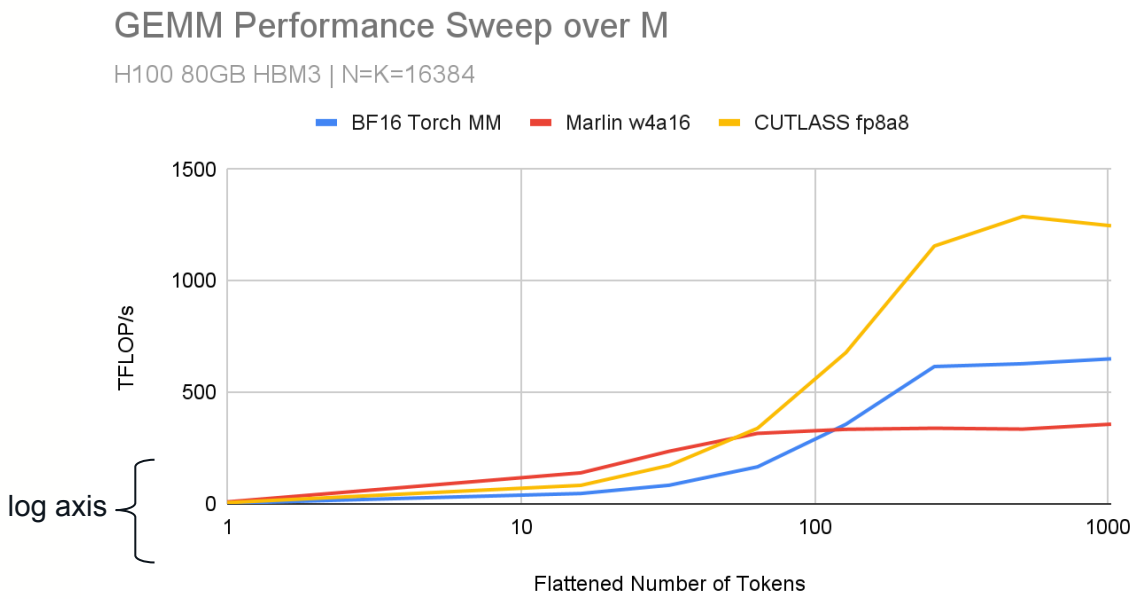
- Most popular formats are integrated in vLLM can be natively loaded and run with our most optimized kernels

Implementation	Volta	Turing	Ampere	Ada	Hopper
AWQ	✗	✅	✅	✅	✅
GPTQ	✅	✅	✅	✅	✅
Marlin (GPTQ/AWQ/FP8)	✗	✗	✅	✅	✅
INT8 (W8A8)	✗	✅	✅	✅	✅
FP8 (W8A8)	✗	✗	✗	✅	✅
AQLM	✅	✅	✅	✅	✅
bitsandbytes	✅	✅	✅	✅	✅
DeepSpeedFP	✅	✅	✅	✅	✅
GGUF	✅	✅	✅	✅	✅

Why Quantize Weights and Activations?

- Quantizing weights reduces memory footprint and data movement.
- Quantizing activations gives compute speedups.

	H100 SXM
BFLOAT16 Tensor Core*	1,979 teraFLOPS
FP16 Tensor Core*	1,979 teraFLOPS
FP8 Tensor Core*	3,958 teraFLOPS
INT8 Tensor Core*	3,958 TOPS



As M sweeps from small to large, the problem transitions to bandwidth to compute bound, where the best performing algorithm maximizes compute speed.

Why Quantize Weights and Activations?

Simple Model – 3 Parameters

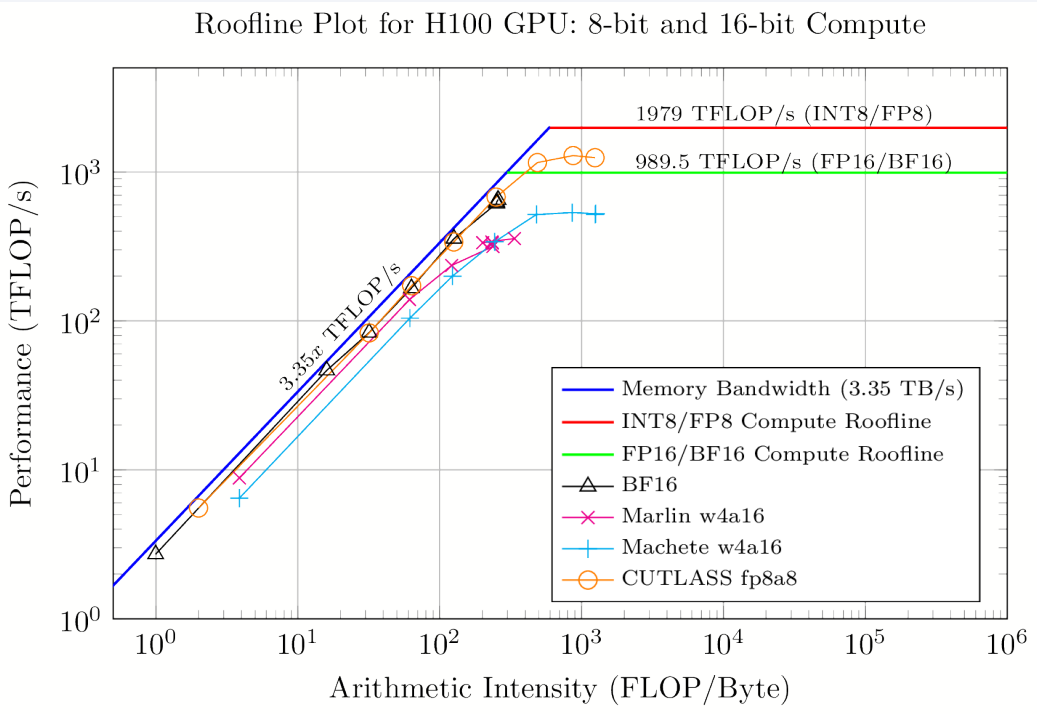
- Rate of computation - TFlop/s
- Rate of data movement - TB/s
- Arithmetic Intensity - Flop/Byte

Simple Graph – 2 Regimes

- Low arithmetic intensity = Bandwidth limited
- High arithmetic intensity = Compute limited

Compression wins in the bandwidth-limited regime

Compute-limited prefills need compute speedup

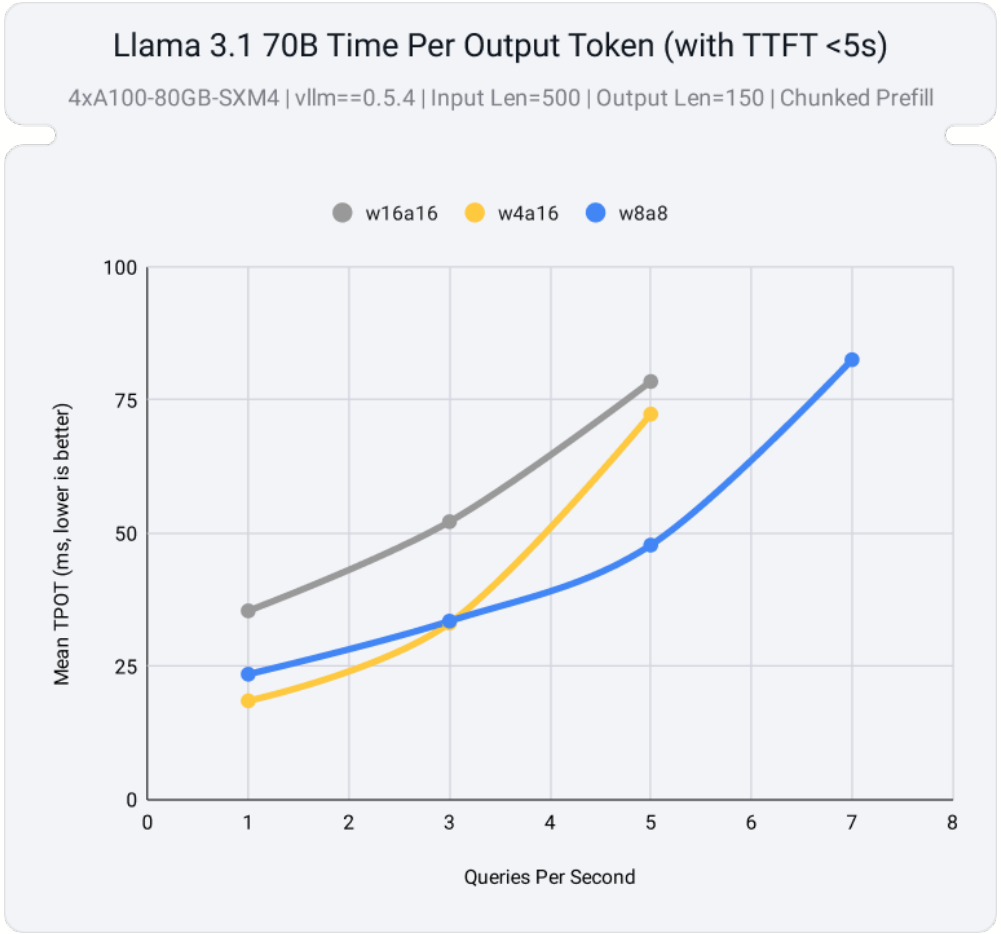


The same numbers from the previous slide, showing *why* the FP8 computation wins at larger M sizes.

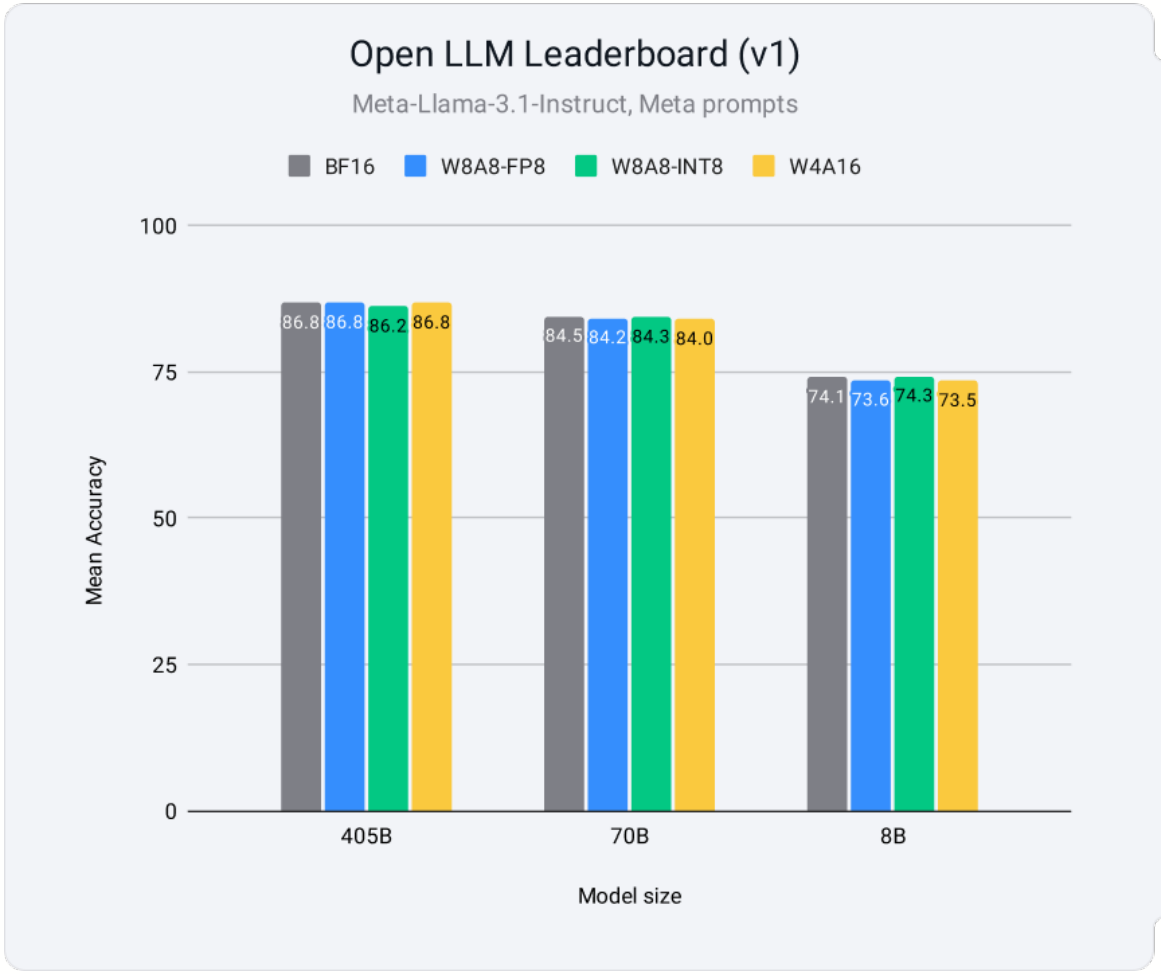
Motivation - E2E Speedup!

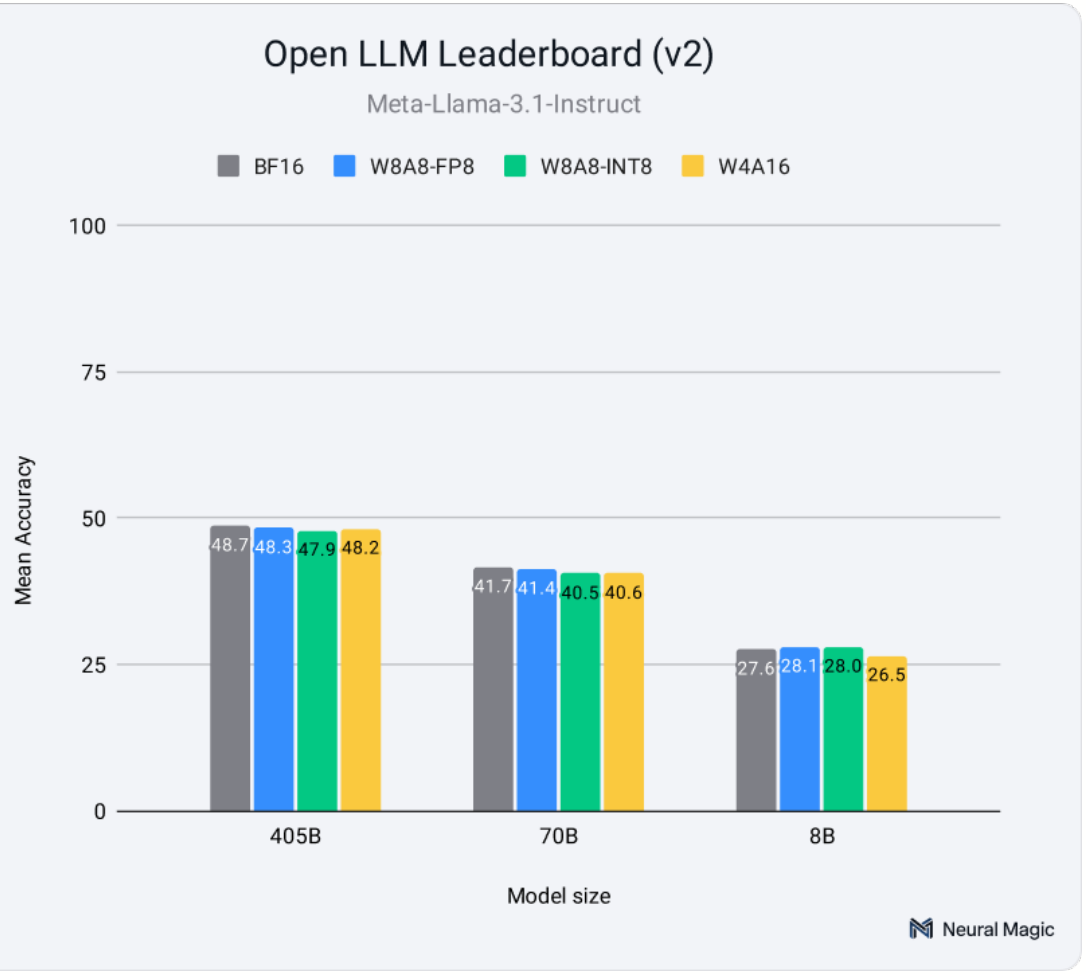
Low QPS:
- Small number of tokens
- Need to optimize data movement

High QPS:
- Larger number of tokens
- Need to optimize compute



… with near-zero accuracy loss





https://neuralmagic.com/blog/we-ran-over-half-a-million-evaluations-on-quantized-llms-heres-what-we-found/

How to Quantize Weights and Activations?

vllm-project/llm-compressor

Step 1: Quantize with llm-compressor

```python
from llmcompressor.transformers import oneshot
from llmcompressor.modifiers.quantization import QuantizationModifier

# Configure the quantization algorithm to run.
recipe = QuantizationModifier(targets="Linear", scheme="FP8", ignore=["lm_head"])

# Apply quantization.
oneshot(
    model=model,
    dataset=ds,
    recipe=recipe,
    max_seq_length=MAX_SEQUENCE_LENGTH,
    num_calibration_samples=NUM_CALIBRATION_SAMPLES,
)

# Save to disk compressed.
SAVE_DIR = MODEL_ID.split("/")[1] + "-W8A8-FP8"
model.save_pretrained(SAVE_DIR, save_compressed=True)
tokenizer.save_pretrained(SAVE_DIR)
```

Step 2: Evaluate with lm-evaluation-harness

Run the following to test accuracy on GSM-8K:

```
lm_eval --model vllm \
  --model_args pretrained="llama-compressed-quickstart",add_bos_token=true \
  --tasks gsm8k \
  --num_fewshot 5 \
  --limit 250 \
  --batch_size 'auto'
```

Step 3: Deploy with vLLM

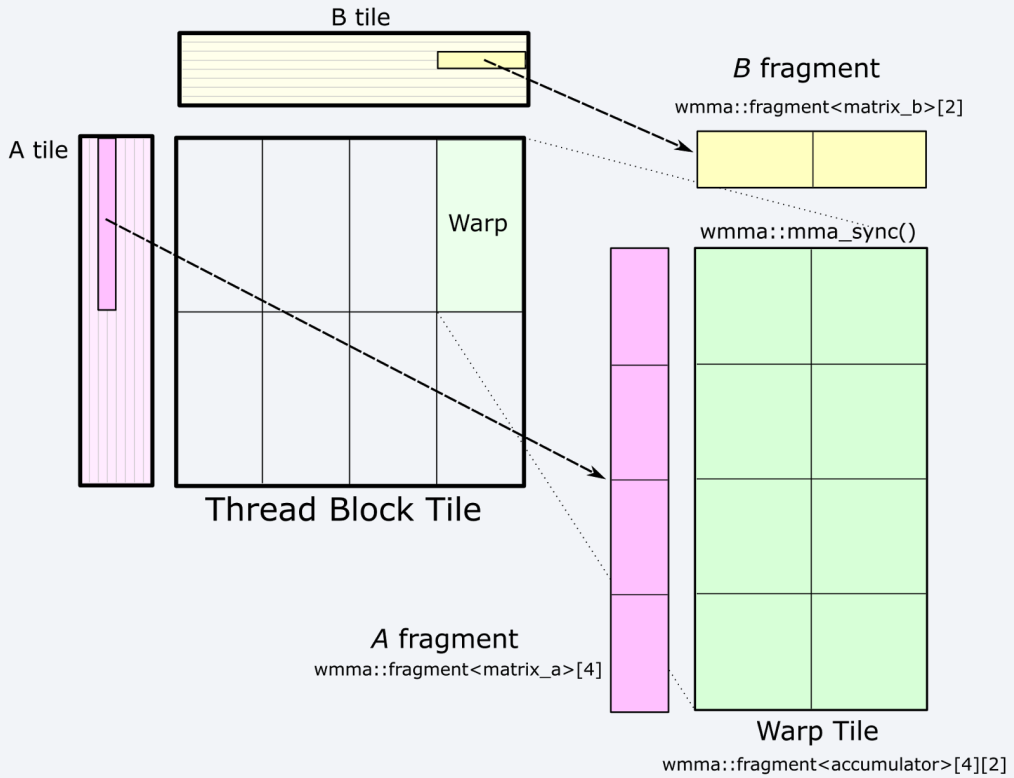
```python
from vllm import LLM

model = LLM("llama-compressed-quickstart")
output = model.generate("I love 4 bit models because")
```

CUTLASS Kernels in vLLM

What is CUTLASS?

- **A “collection of CUDA C++ template abstractions for implementing high-performance linear algebra operations"**
  - Not a library like cuBLAS. It lets users build libraries from modular pieces that can be used to construct specific kernels.
  - CuTe layout algebra formalizes the layout of high-dimensional tensors in linear memory.
  - For activation quantization in vLLM, we chose CUTLASS specifically for epilogue fusion.

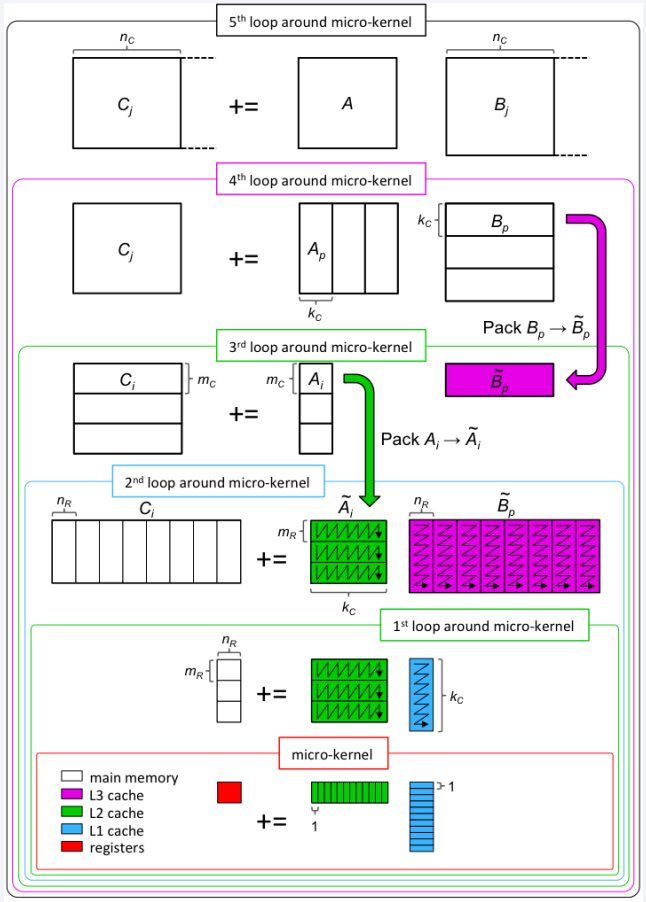


Opportunities for Fusion in GEMM

Fusion is a technique to reduce data movement by applying element-wise operations to operands that are already in fast memory.

Epilogue fusion applies elementwise operations to the output right before writing it to GPU memory.

We use CUTLASS ***epilogue fusion*** to handle scales and zero points and future-looking flexibility.



The BLIS figure GEMM on CPUs
- Robert van de Geijn
Originally appeared in Van Zee and Smith (2017)

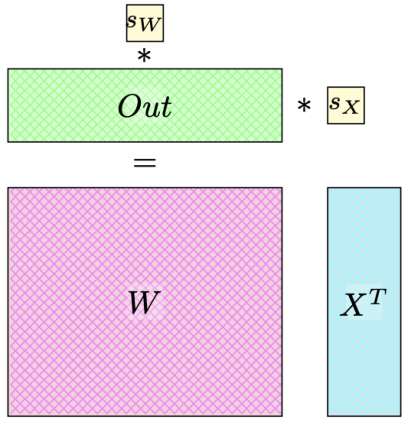
Epilogue Fusion - Symmetric Quantization

Quantize by *scaling* floating point vectors, rounding them, and storing as a low-precision int.

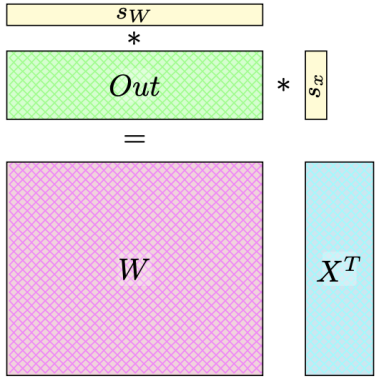
Example:
- Find absmax of X
- Multiply each element by 128 / absmax(X)
- Store as a vector of int8s

Goal: Compute with quantized inputs
Solution:
- Factor out the scales and apply them to the output after the GEMM.
- Epilogue fusion efficiently applies scales

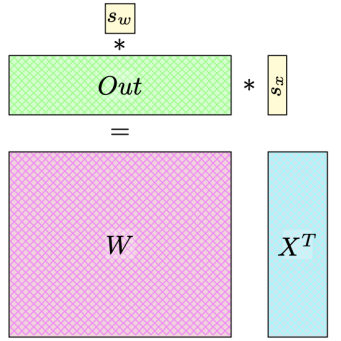
Scales can be per-tensor:



Or per-token/channel:



Or any other combination:



Epilogue Fusion - Zero Points

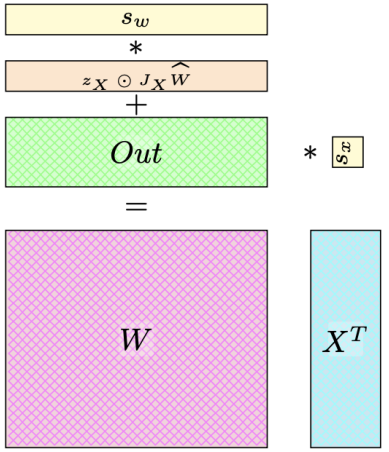
Asymmetric quantization adds a *zero point.*
- More effectively quantizes skewed distributions
- Up to one more bit of precision
- Zero no longer maps to zero

Epilogues now must apply correction terms that depend on the weight and activation matrices.

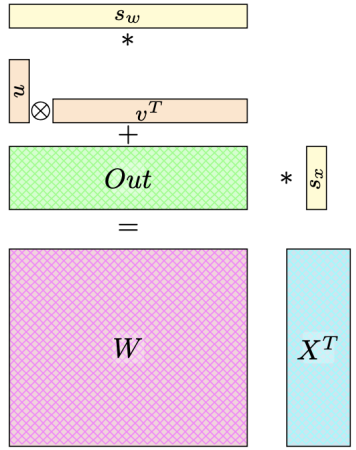
Static per-tensor:
- Compute a correction term offline
- Apply it like a bias

Dynamic per-token:
- Correction term computed during the forward pass.
- Correction term is now the size of the output matrix!
- Compute an outer product during and add it to the output during the epilogue.

Static per-tensor:



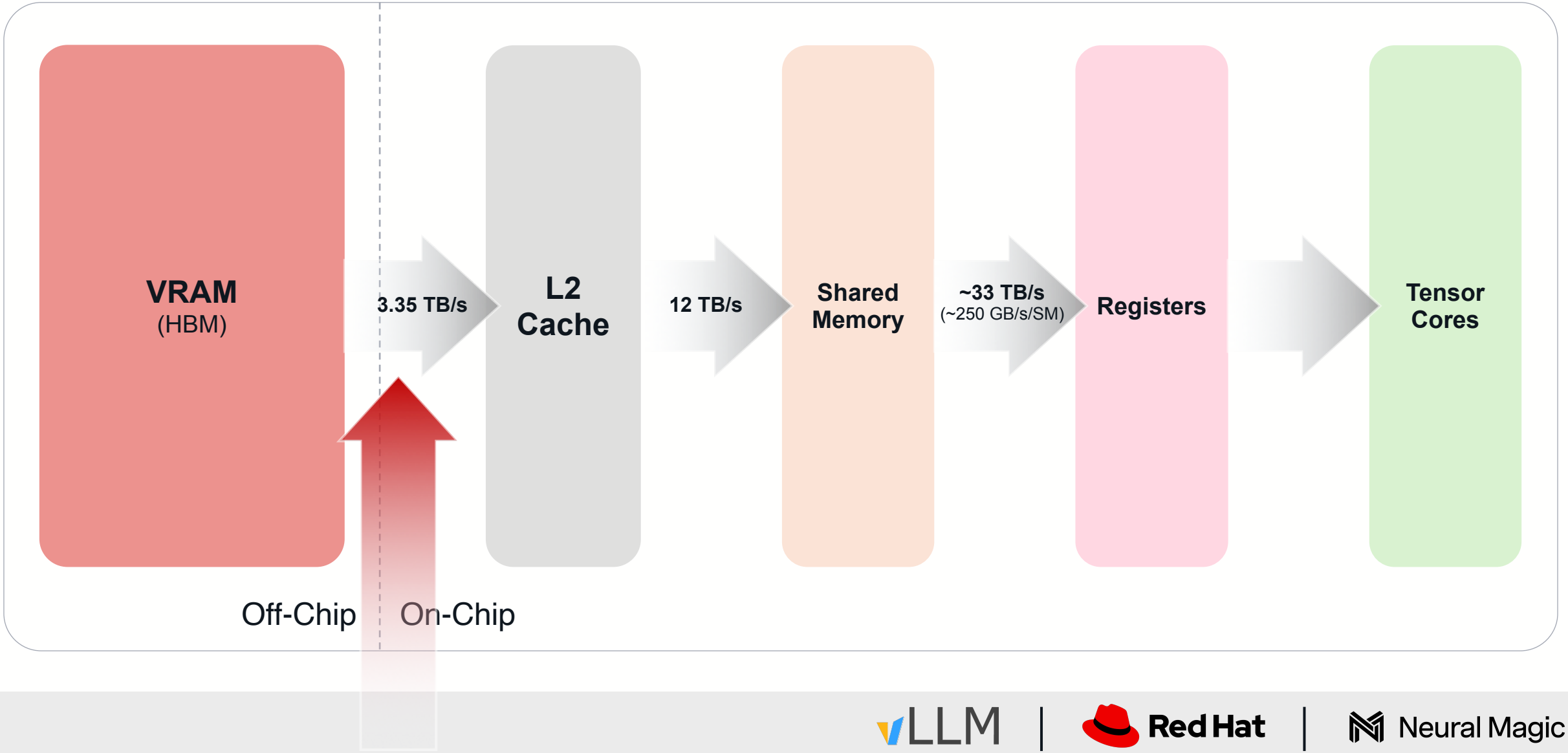
Dynamic per-token:



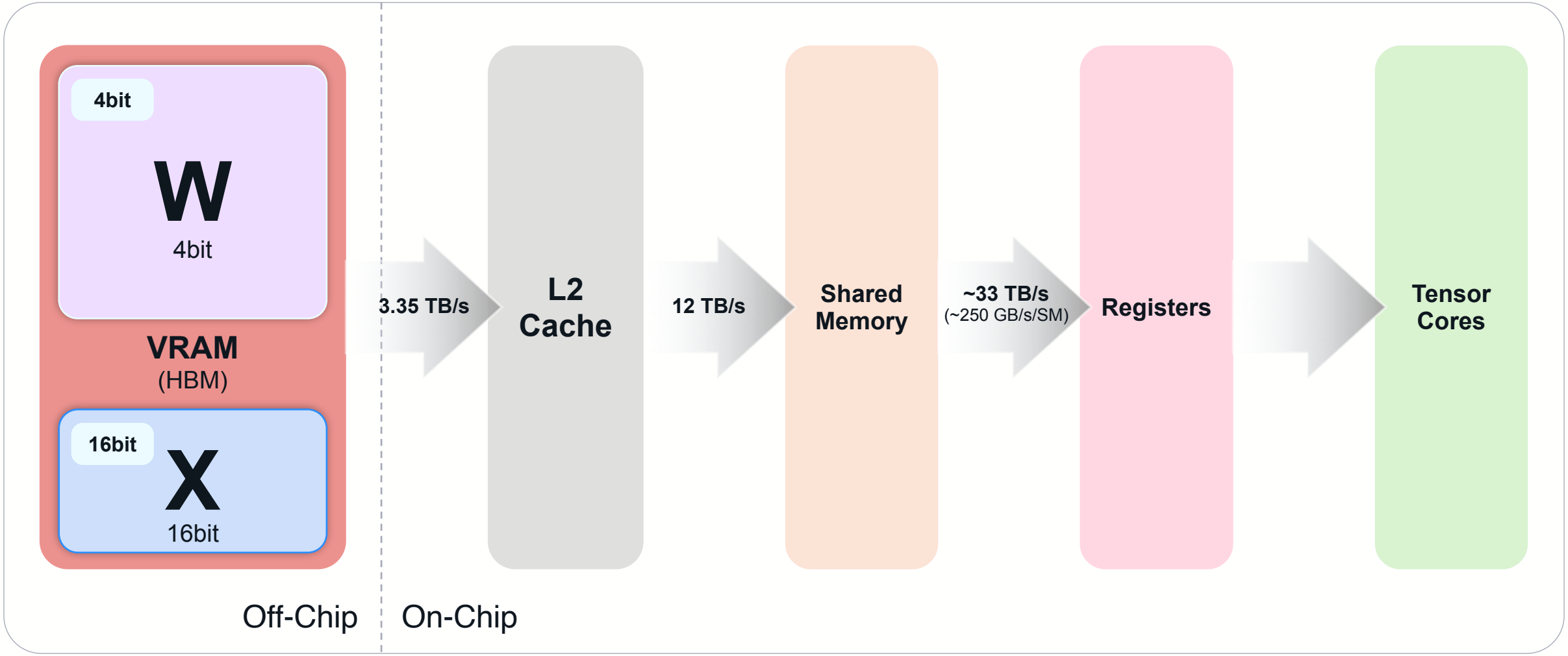
Machete Deep Dive

w4a16

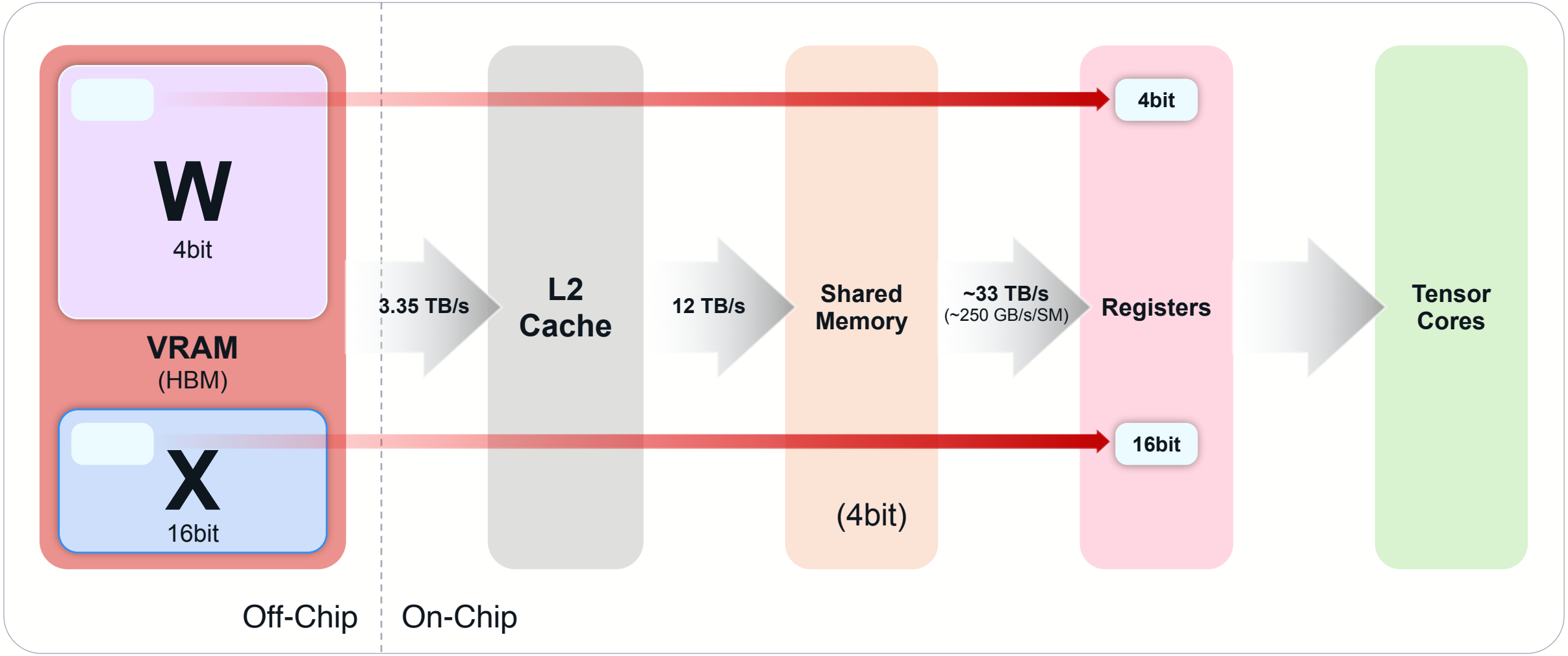
Background - Mixed-Input Linear Layers



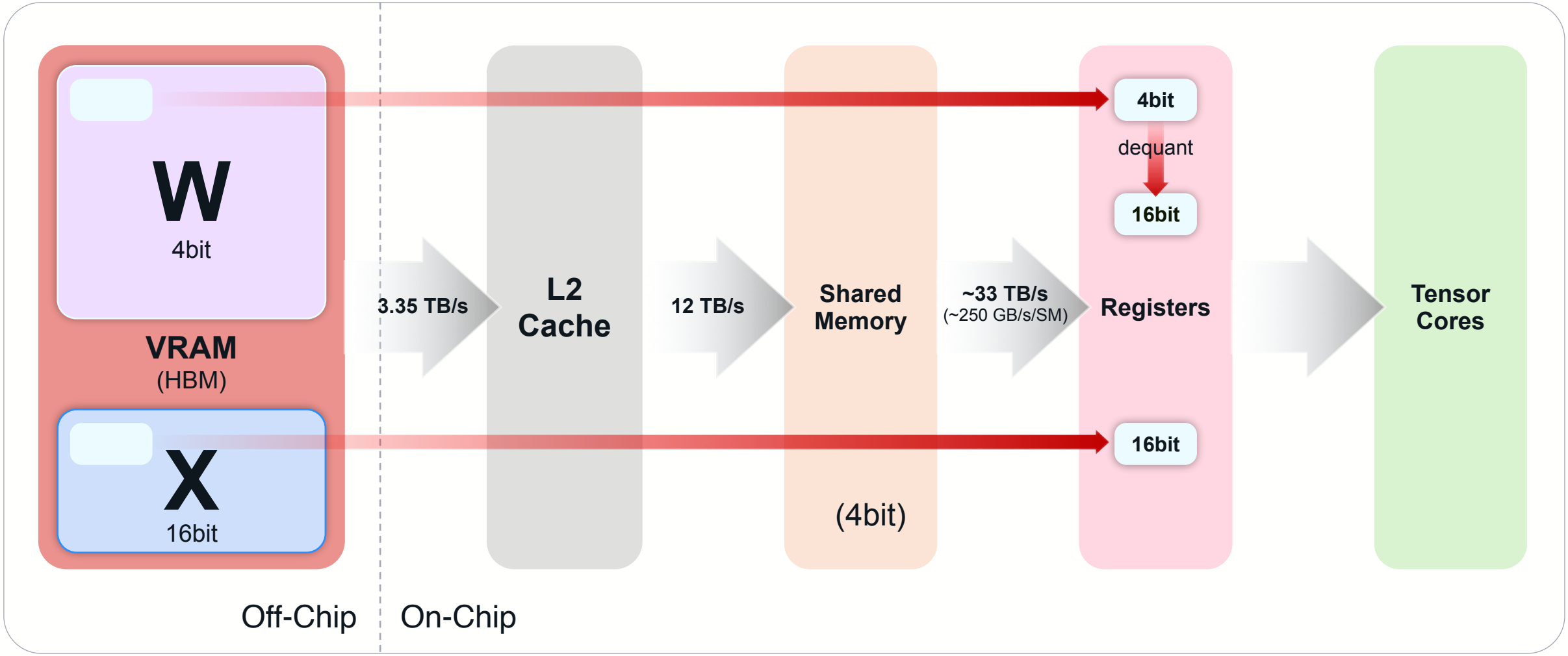
Background - Mixed-Input Linear Layers



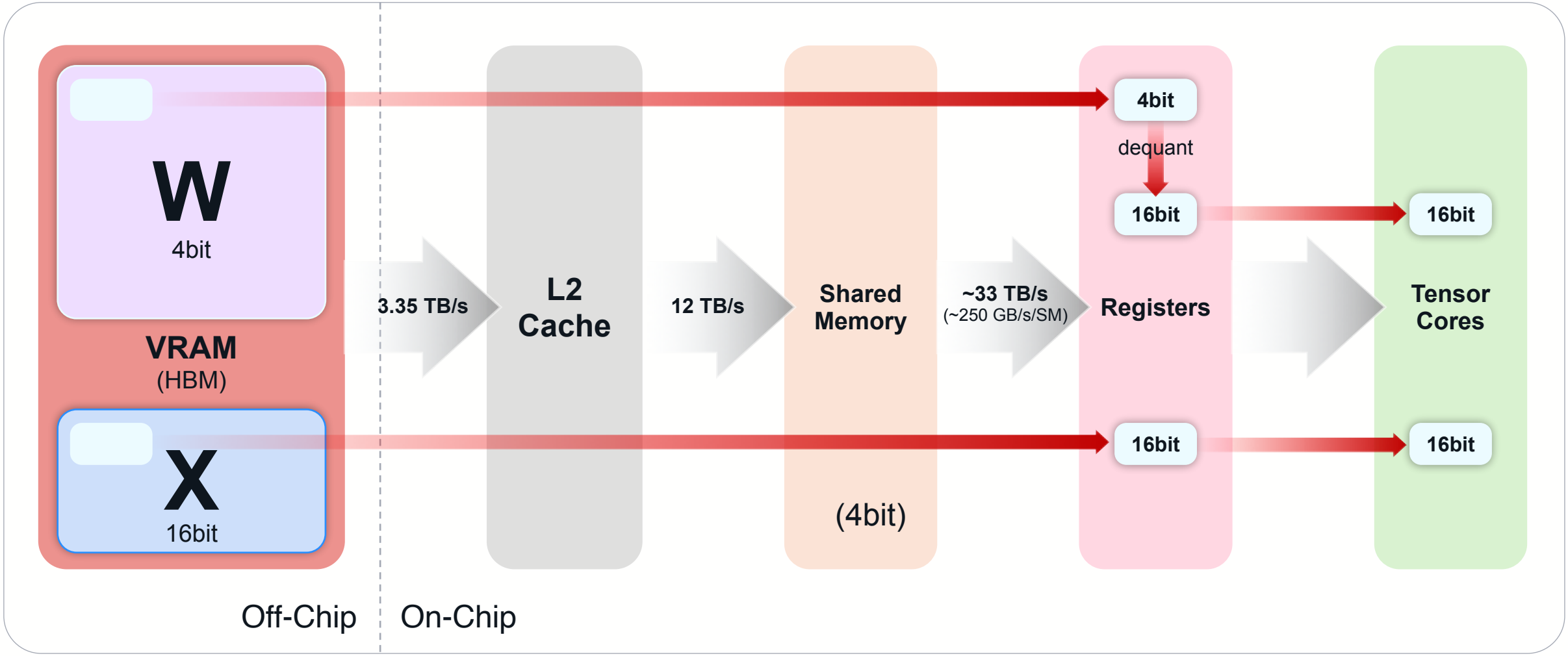
Background - Mixed-Input Linear Layers



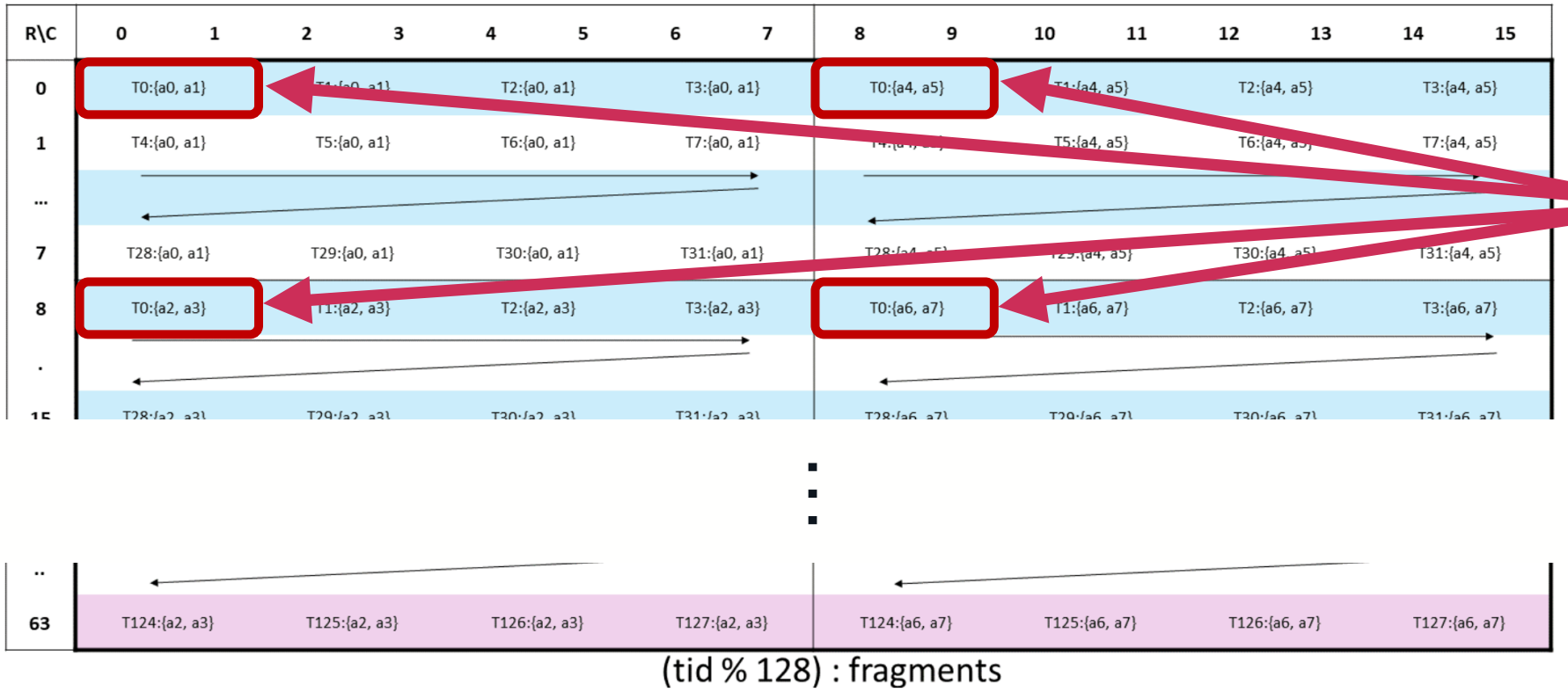
Background - Mixed-Input Linear Layers



Background - Mixed-Input Linear Layers



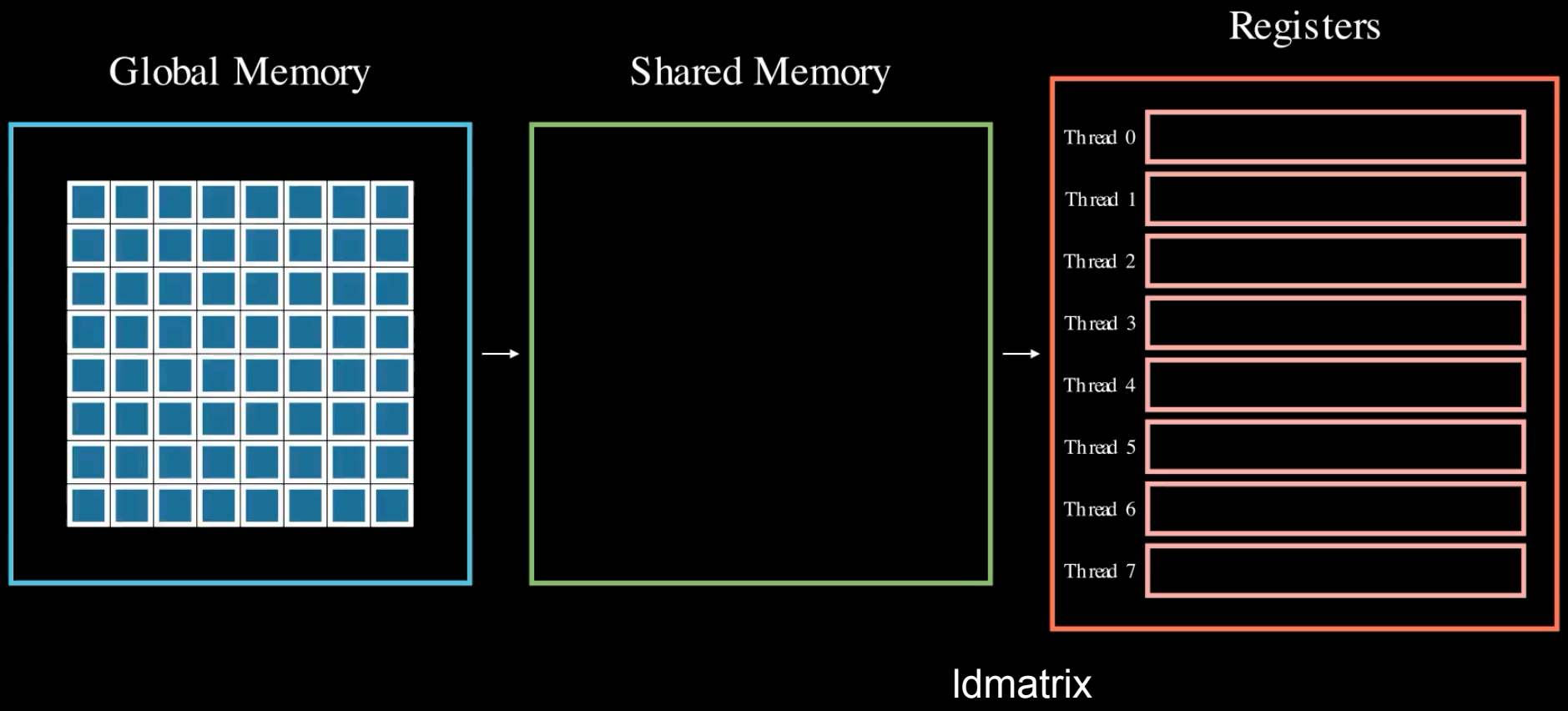
Weight Pre-shuffling | Tensor Core Layouts



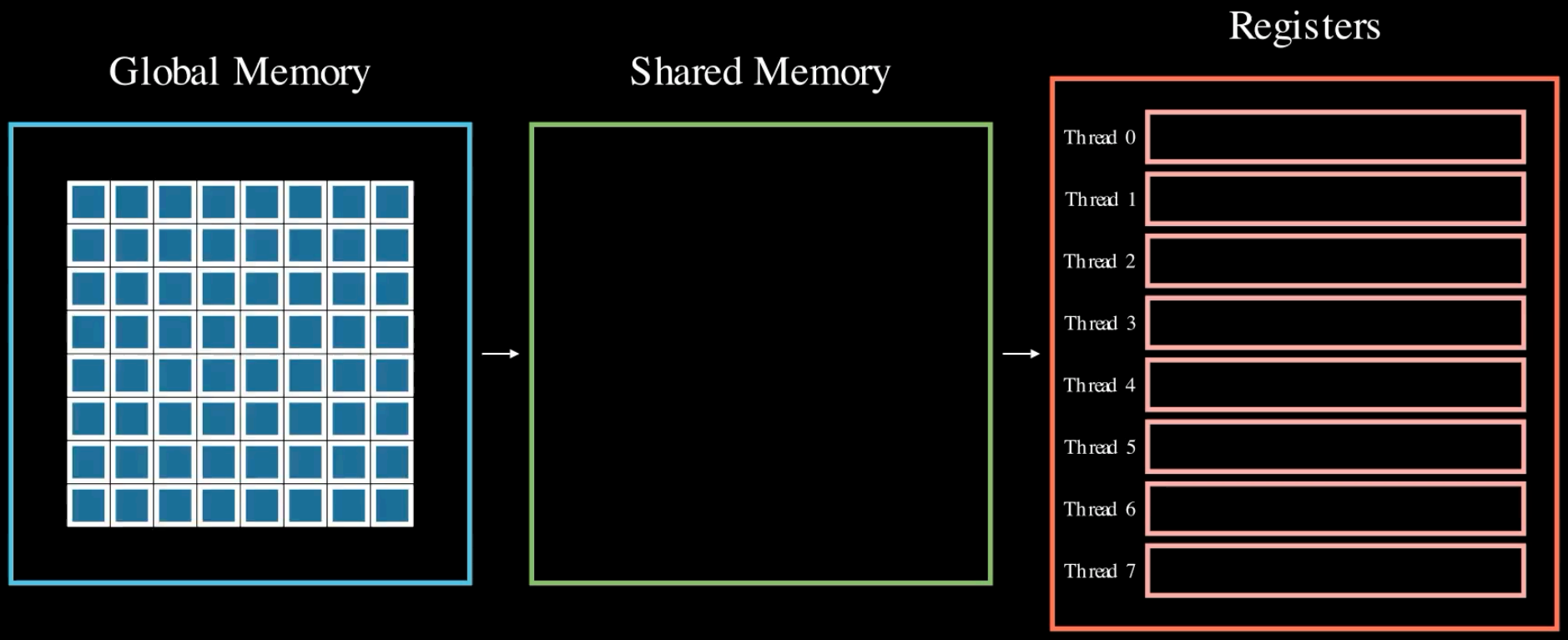
Data for Thread 0
Non-contiguous

Figure 122: *WGMMA .m64nNk16 register fragment layout for matrix A.*

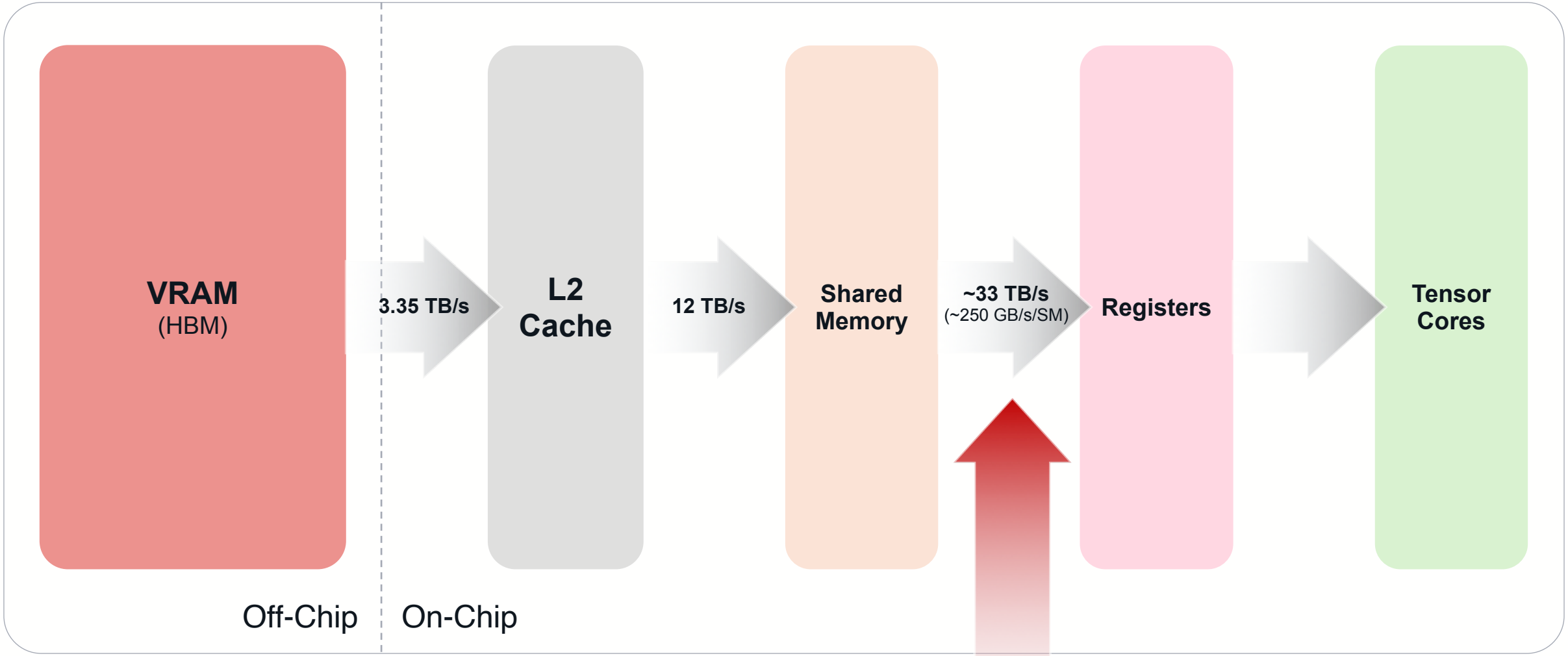
Weight Pre-shuffling | Weight Loading (ldmatrix)



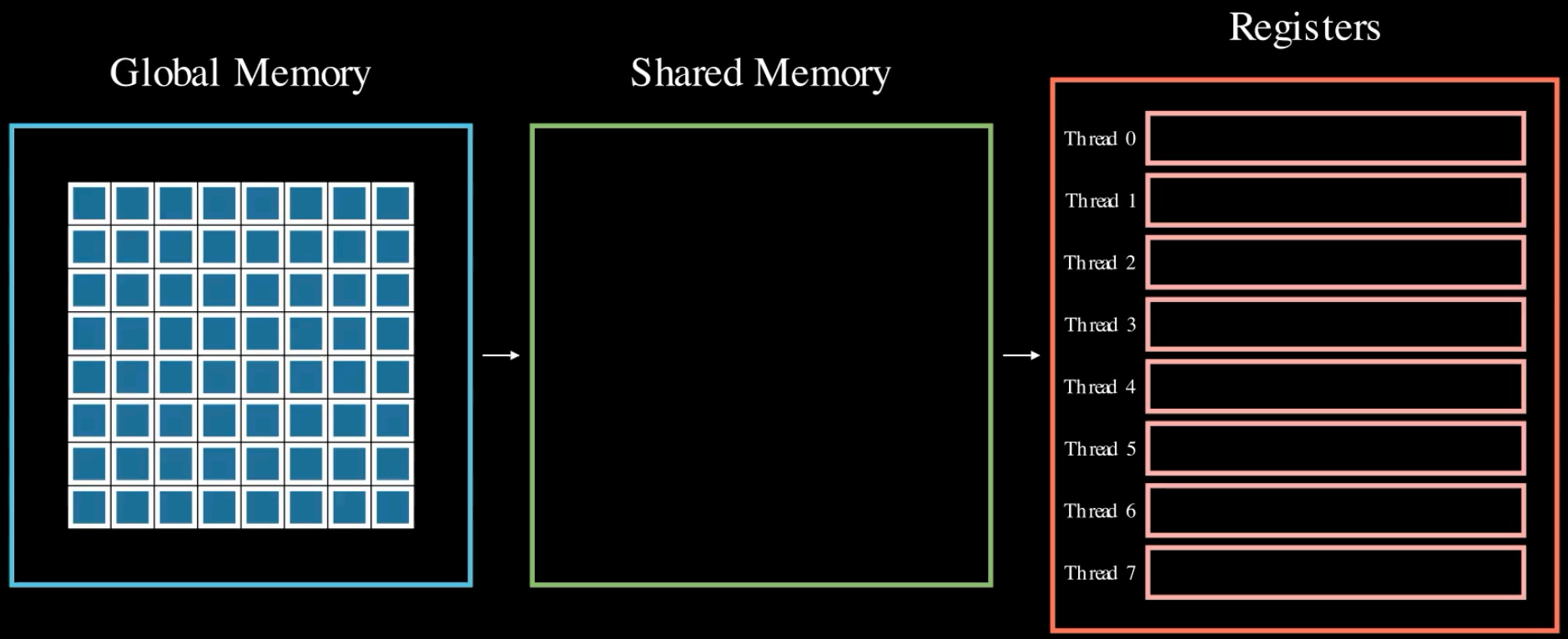
Weight Pre-shuffling | Weight Loading (4bit)



Weight Pre-shuffling | Weight Loading Bottleneck

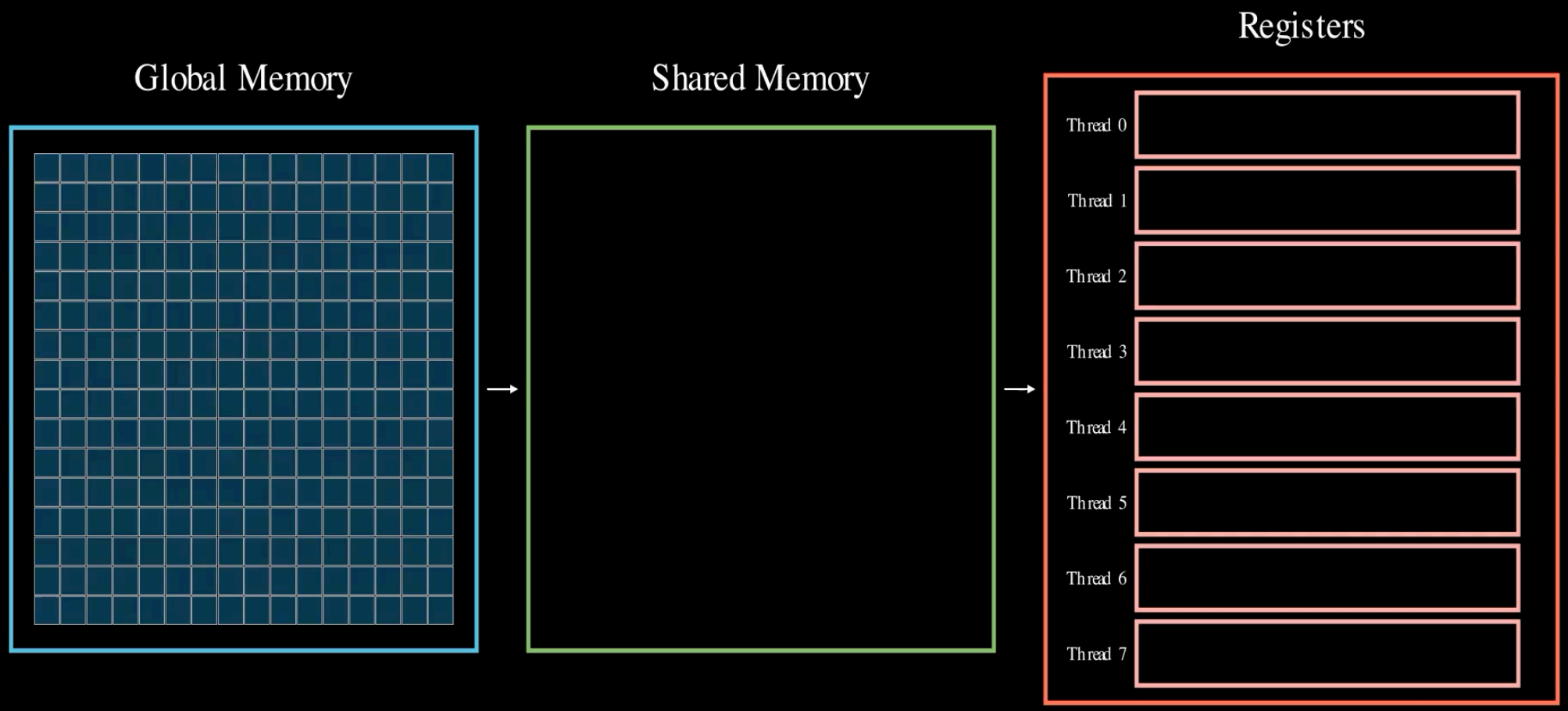


Weight Pre-shuffling | Weight Loading (32bit)



32bit loads

Weight Pre-shuffling | Weight Loading (128bit)



128bit loads

Weight Pre-shuffling | What is CuTe

- a way of **describing multidimensional hierarchical layouts**
- includes a **formal layout algebra**
- **implemented** as a collection of C++ templates **in CUTLASS**

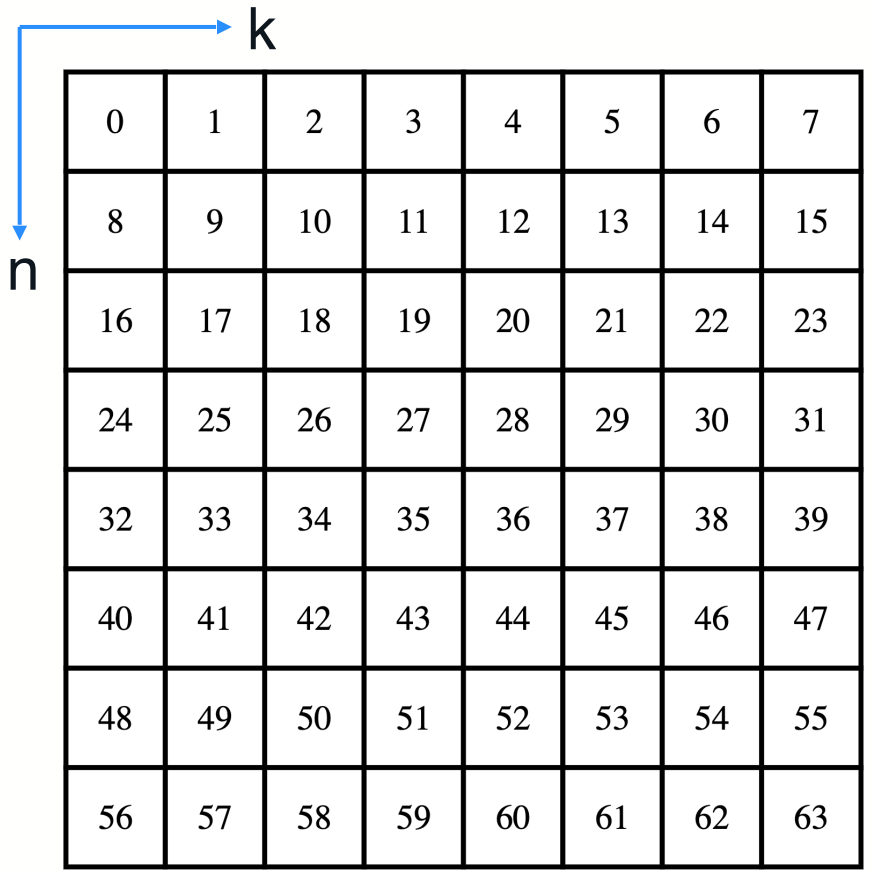
Docs: CuTe Docs Colfax (Jay Shah) Blog

Weight Pre-shuffling | CuTe Layouts

$$Shape : Stride$$

(8, 8) : (_, _)

n (rows) k (cols)

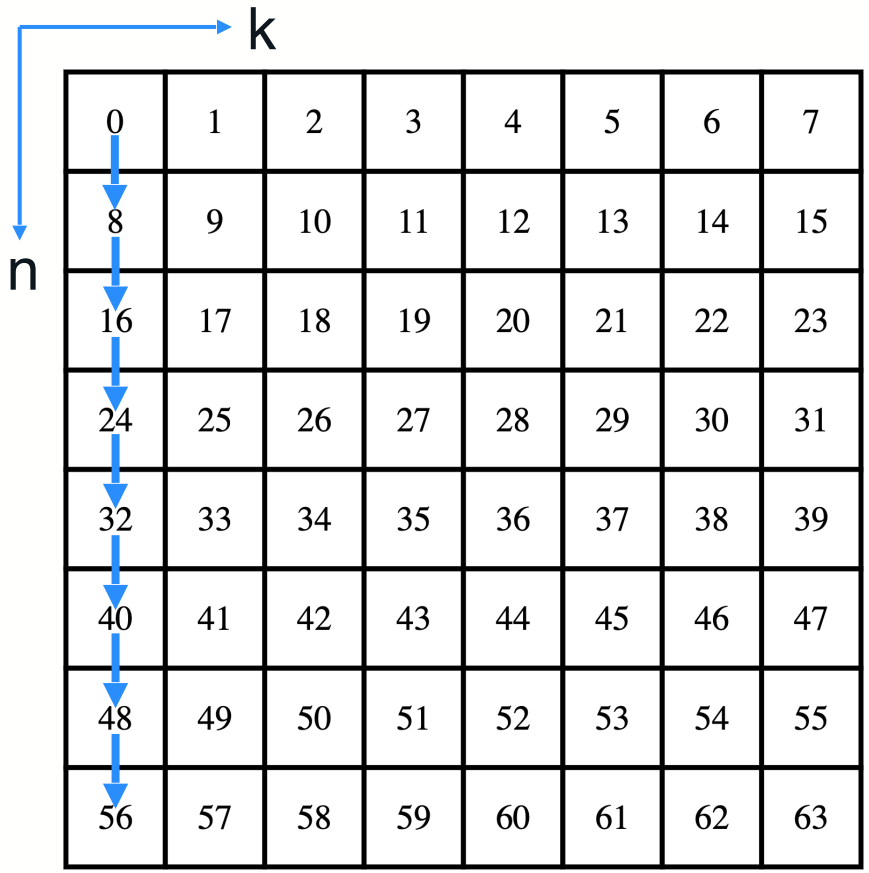


Weight Pre-shuffling | CuTe Layouts

$$Shape : Stride$$

(8, 8) : (8, _)

n (rows) k (cols)



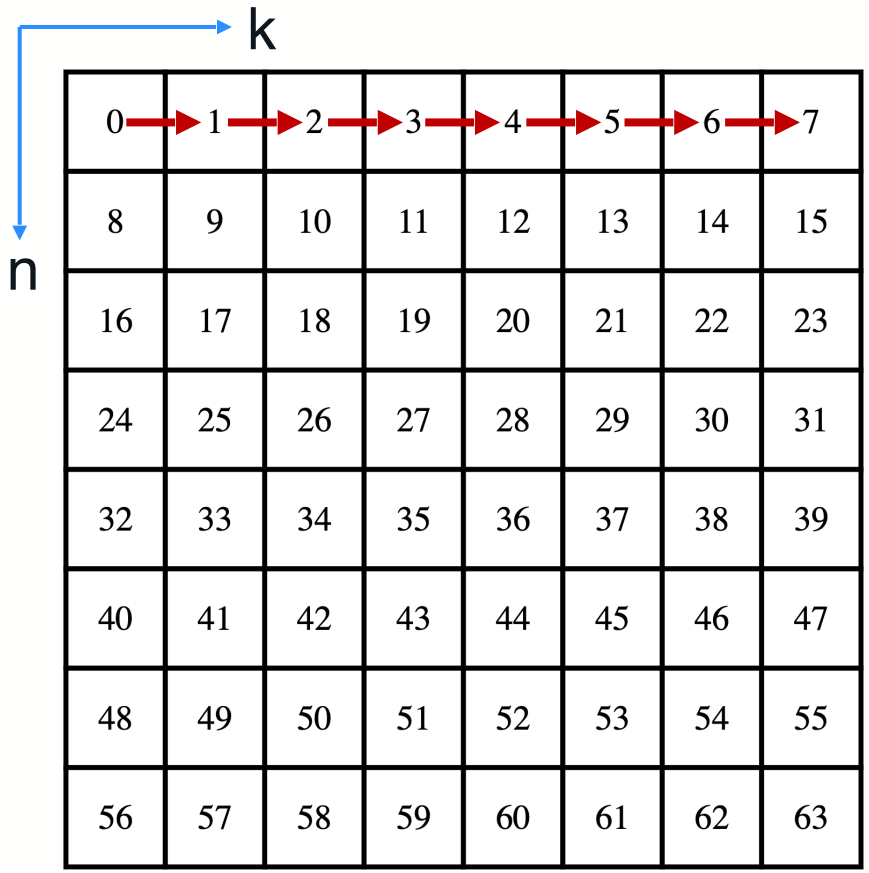
Weight Pre-shuffling | CuTe Layouts

$$Shape : Stride$$

(8, 8) : (8, 1)

n (rows)

k (cols)

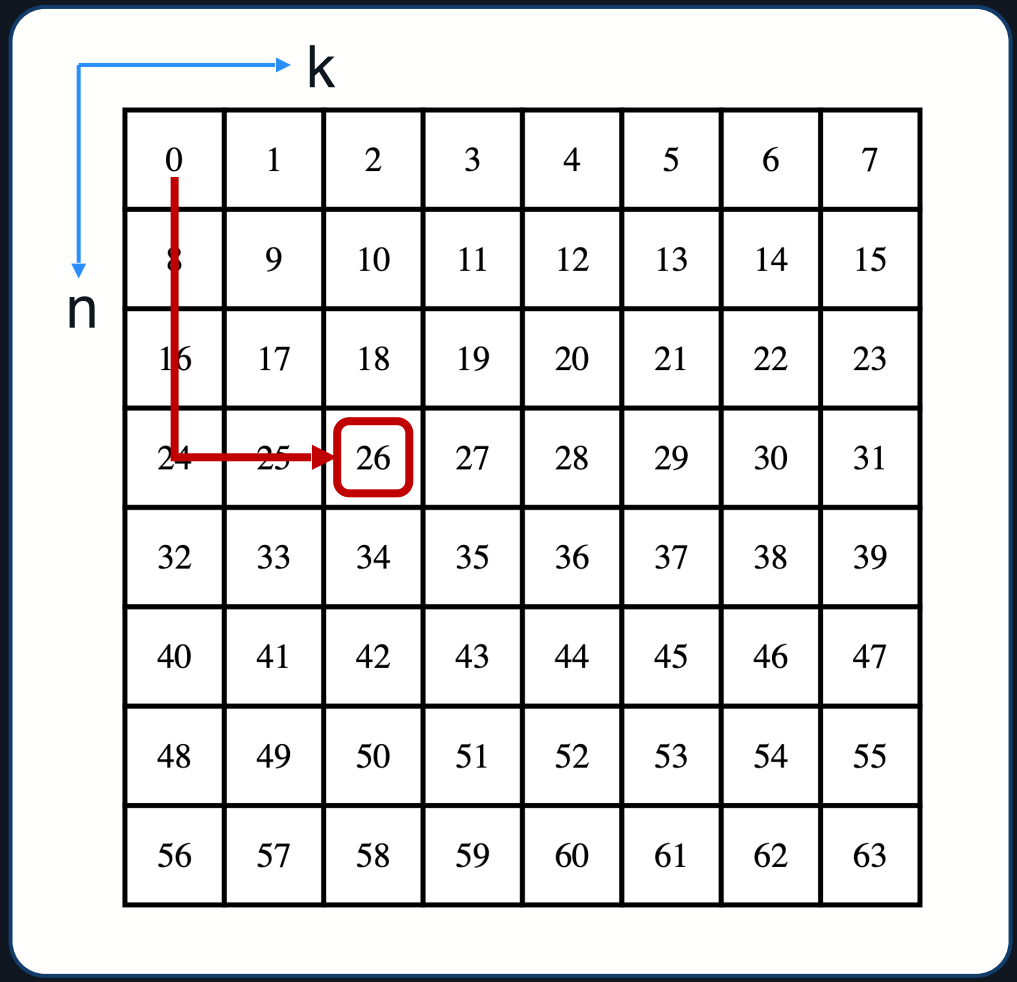


Weight Pre-shuffling | CuTe Layouts

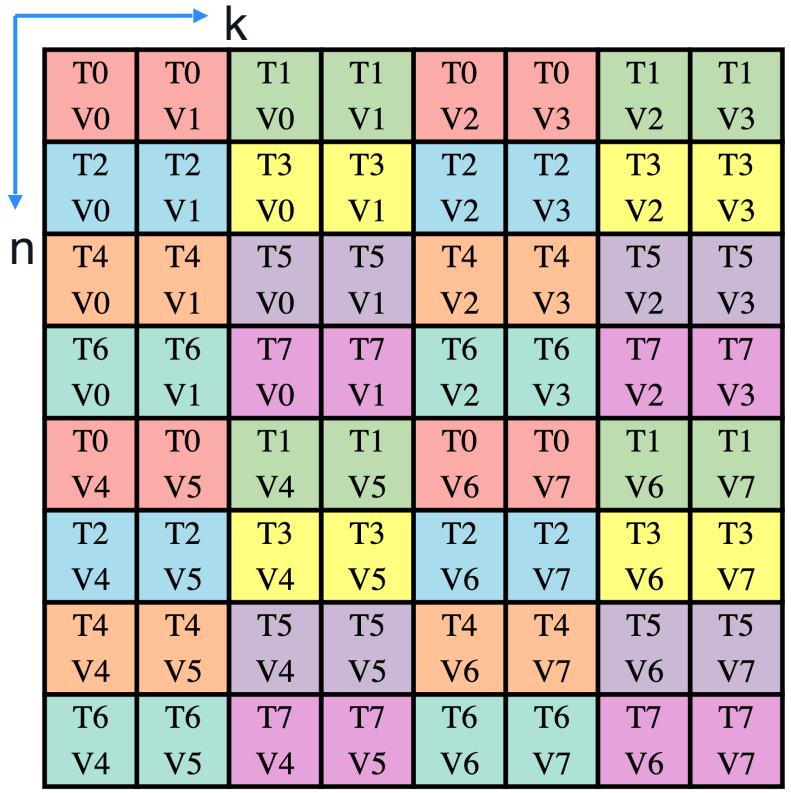
$$Shape : Stride$$

$$f = (8, 8) : (8, 1)$$

$$f(3, 2) = 3 * 8 + 2 * 1 = 26$$



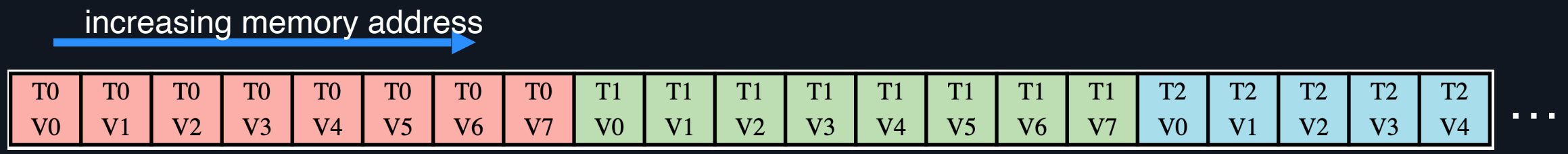
Weight Pre-shuffling | Tensor Core Layout



$f(n, k) \rightarrow (t, v)$

$f = ((4, 2), (2, 2, 2)) : ((2, 32), (8, 1, 16))$

Weight Pre-shuffling | Desired Layout



$$m(\mathrm{t}, \mathrm{v}) \longrightarrow \text{mem\_offset}$$

$$m = \ (8, 8) : (8, 1)$$

Weight Pre-shuffling | Desired Layout

Tensor Core Layout
(Static, provided by CuTE)

$$f(\text{n, k}) \rightarrow (\text{t, v})$$

Thread/Value to desired
mem-location

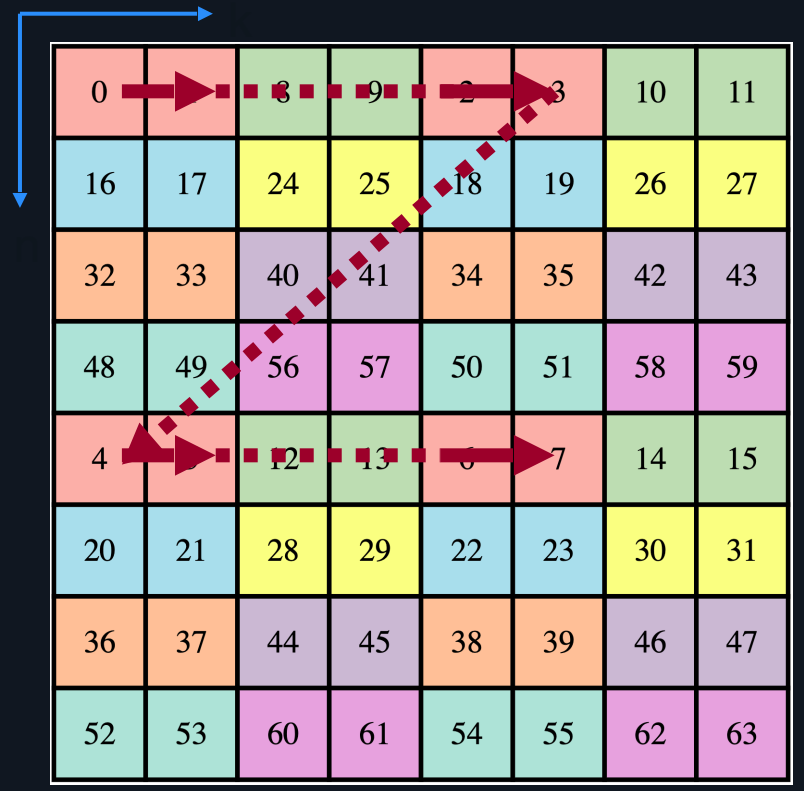
$$m(\text{t, v}) \rightarrow \text{mem} = (\text{t, v}) : (\text{v, 1})$$

Desired Layout
(Pre-shuffled weight Layout)

$$g(\text{n, k}) = m \circ f$$

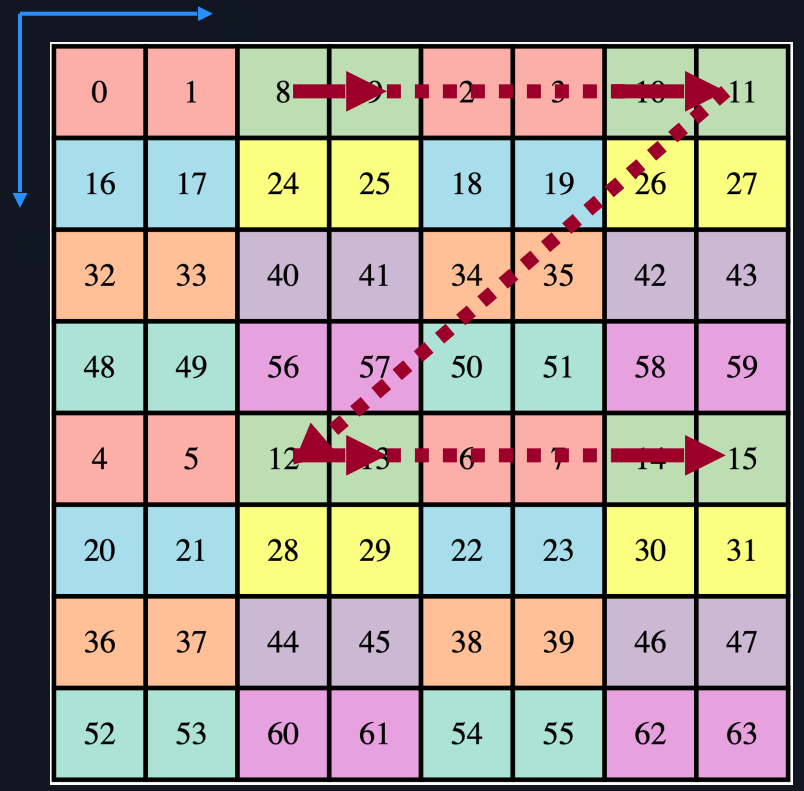
Layout Composition
(CuTe)

Weight Pre-shuffling | Desired Layout



$$g = m \circ f = ((4, 2), (2, 2, 2)) : ((16, 4), (1, 8, 2))$$

Weight Pre-shuffling | Desired Layout



$$g = m \circ f = ((4, 2), (2, 2, 2)) : ((16, 4), (1, 8, 2))$$

Weight Pre-shuffling | Pre-shuffle Kernel

Layout Right Inverse (CuTe)

Layout Composition (CuTe)

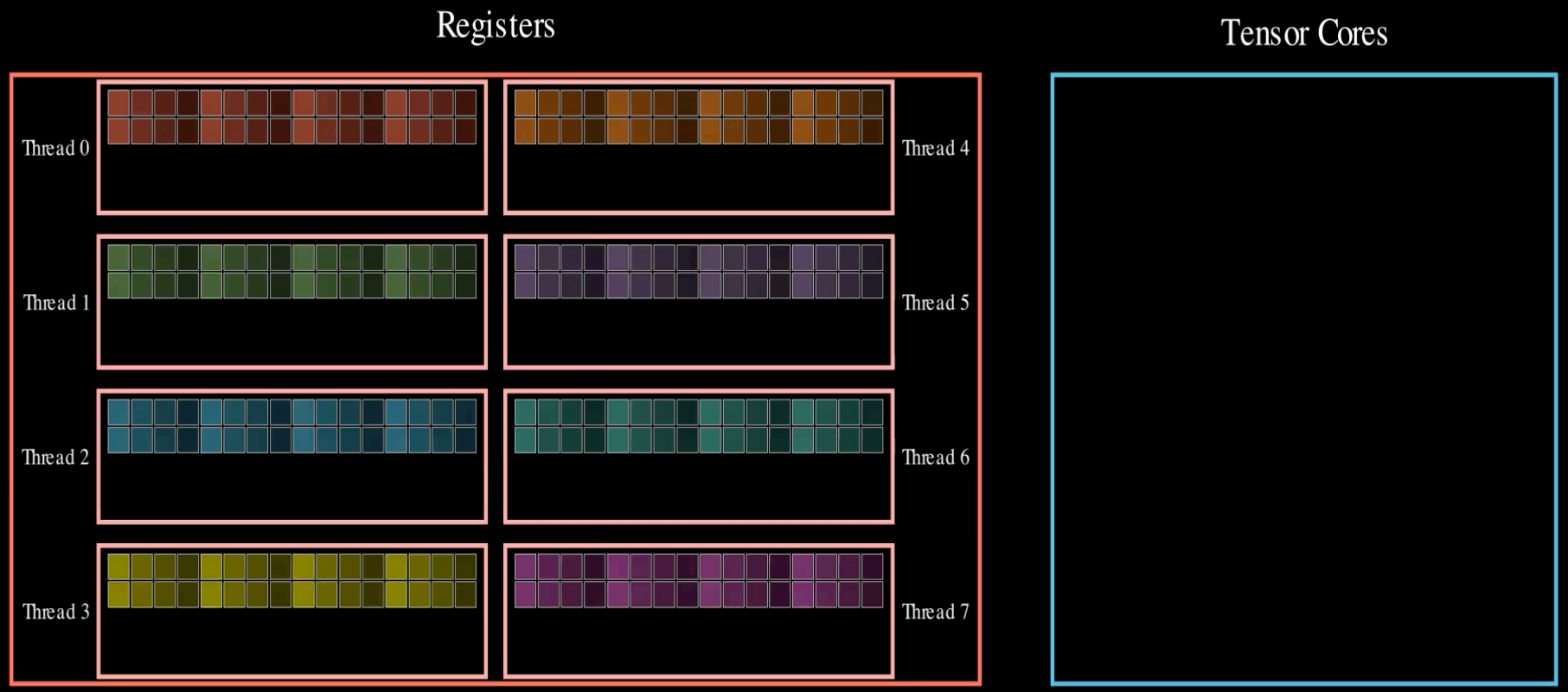
$$p = s \circ g^{-1}$$

Algebraically determine pre-shuffle kernel

Source Layout (e.g. col-major, …)

Desired Layout

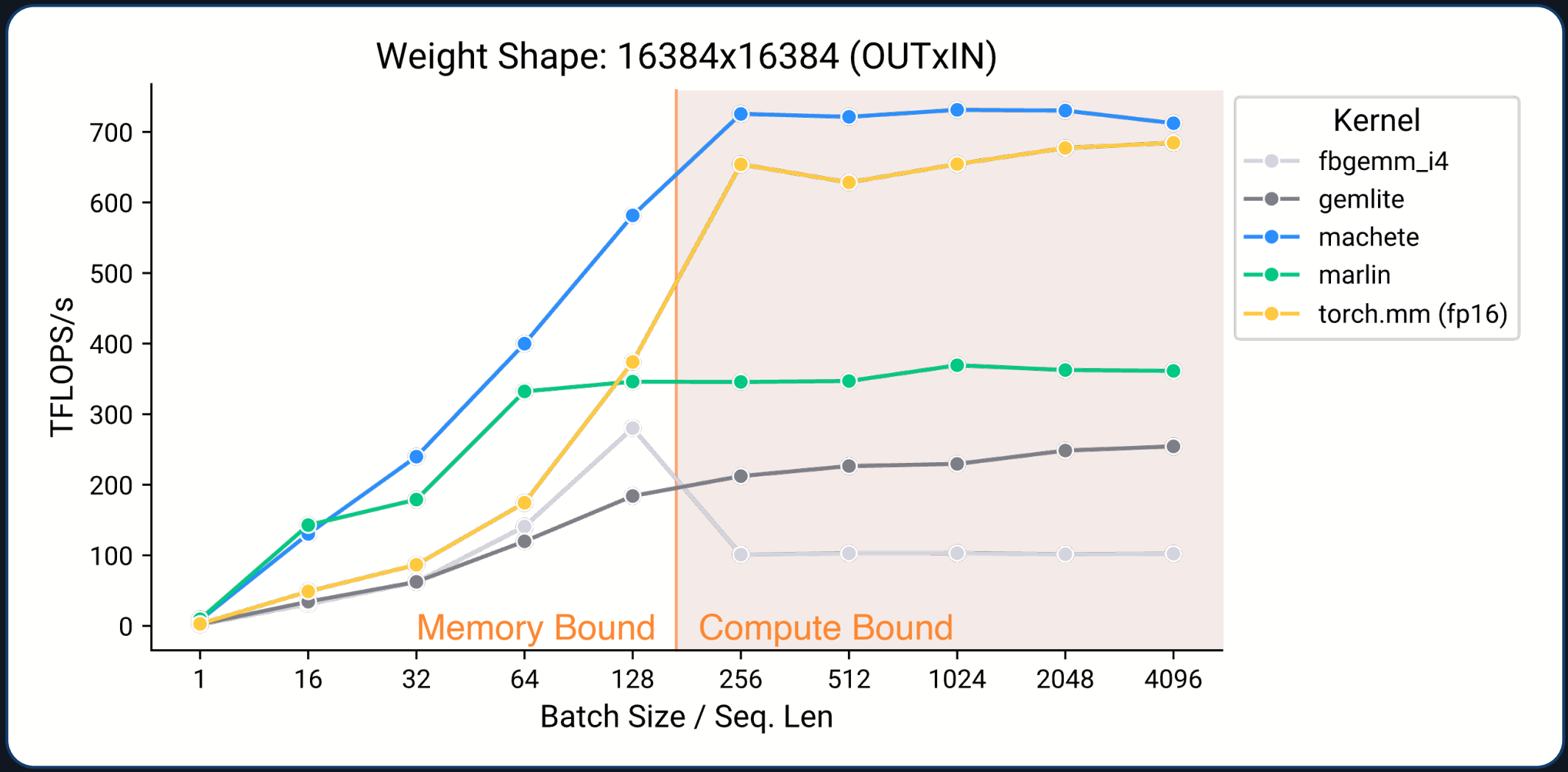
Interleaved Upconverts



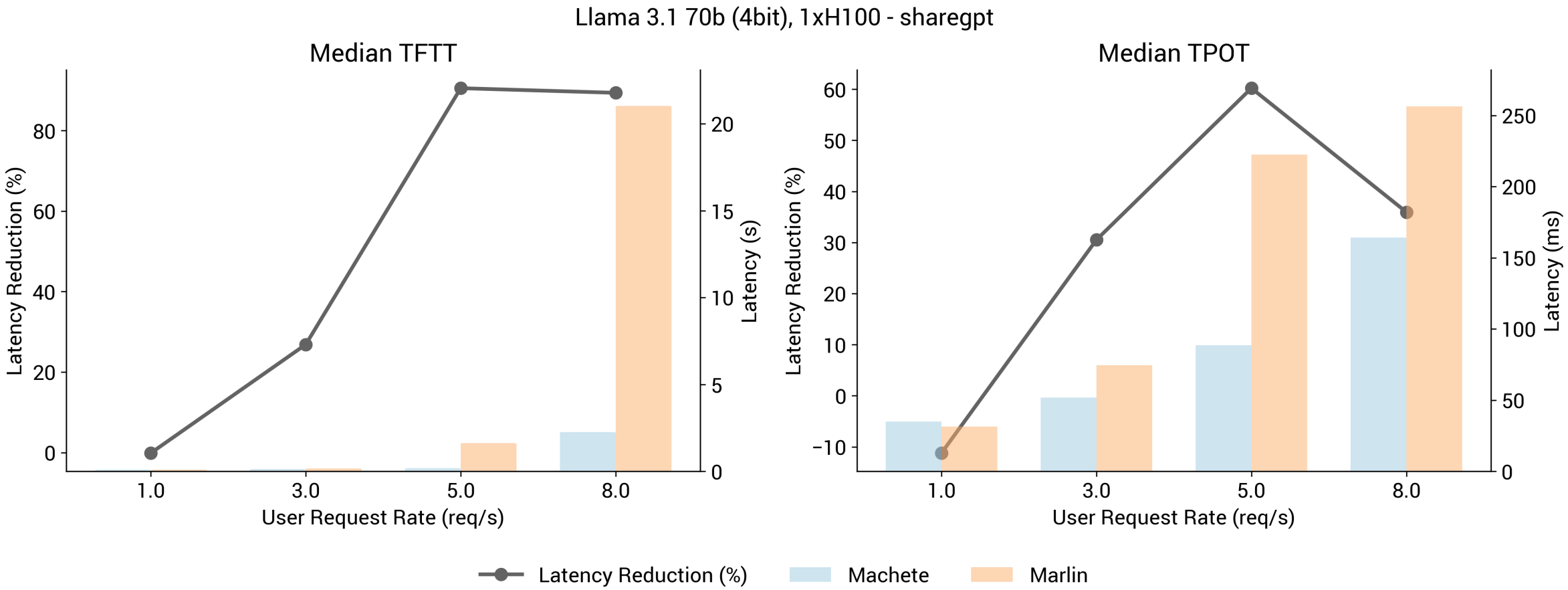
Machete | Additional-Optimizations

- Pipelining modified CUTLASS
- Stream-K via CUTLASS' PersistentTileSchedulerSm90StreamK
- Use New Hopper Hardware Instructions
  - Tensor Memory Accelerator (TMA)
  - wgmma modified CUTLASS
- Warp-specialization, CUTLASS

Machete I Results



Machete | Results



Other Forms of Quantization and Compression

In vLLM today

- Block quantization
  - Used in DeepSeek V3
- Quantized KV caches
  - Reduced memory footprint
  - Critical for long contexts
- 2:4 Sparsity
  - Compress weights and speed up computation
  - Combines with quantization

Future Work / In-Progress

- Blackwell FP4 and FP6
  - Hardware support for block-quantization
- Spin Quant
  - Hadamard transforms reduce outliners
- Quantized collective ops

Get Involved With vLLM

Try it out!

- pip install vllm
- vllm serve neuralmagic/Meta-Llama-3.1-8B-Instruct-FP8

Give Us Feedback

Respond and tell us what we are doing right and what we can do better with vLLM.

Join the vLLM Developer Slack

Contribute to key vLLM features

- Comment and review PRs that are interesting to you
- Join the discussion on RFCs
- Check out “good first issue” tags
- Build examples and demos with other tools

Join Neural Magic’s Mission

Neural Magic wants to bring open-source LLMs and vLLM to every enterprise on the planet.

We are looking for **vLLM Engineers** to help us accomplish our mission.

https://www.redhat.com/en/jobs

vLLM | Red Hat | Neural Magic

vLLM

Building the fastest and easiest-to-use open-source LLM inference & serving engine!

https://github.com/vllm-project/vllm

https://slack.vllm.ai

https://twitter.com/vllm_project

https://opencollective.com/vllm