arm

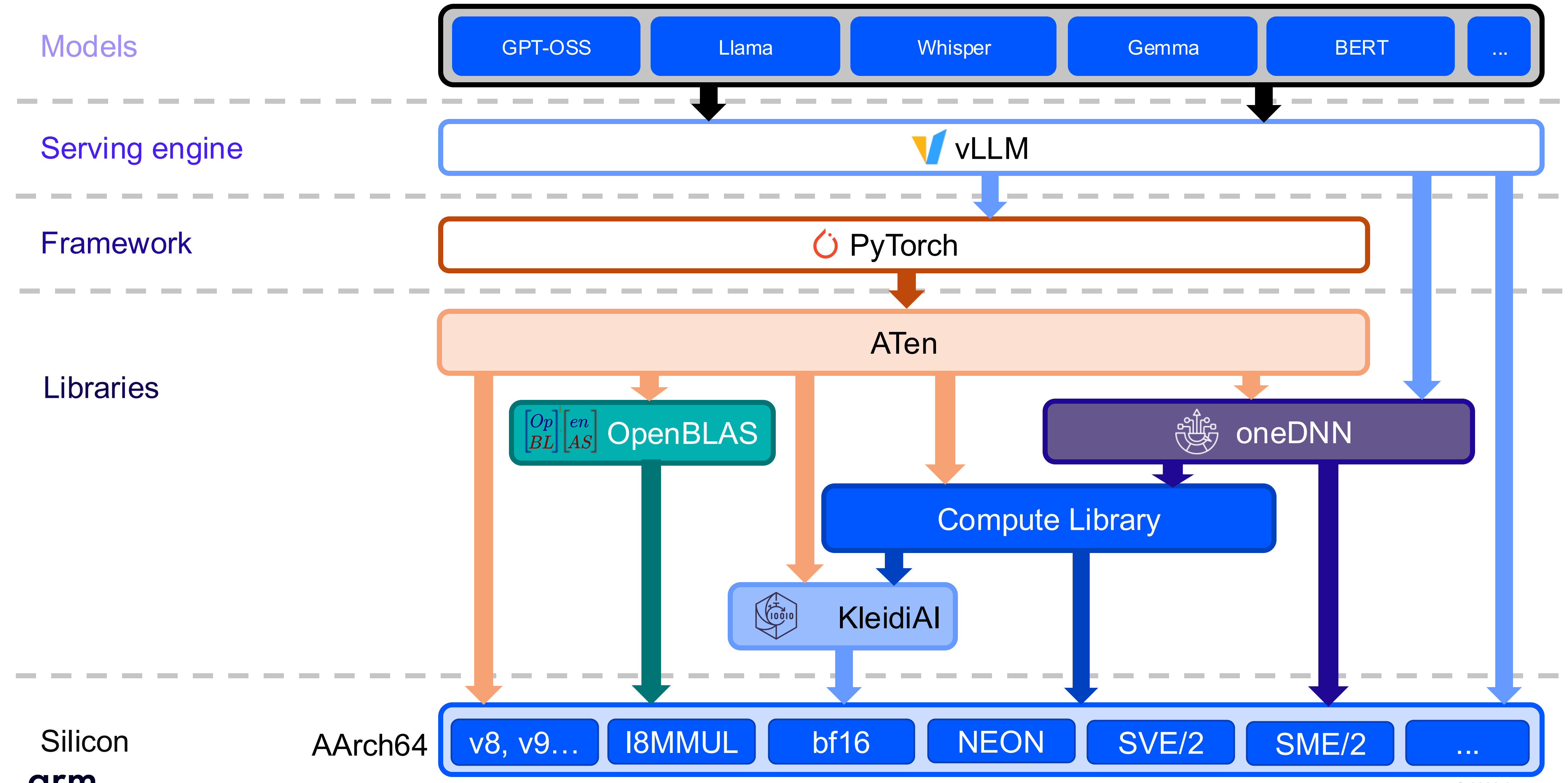
# Optimizing LLM Inference on CPU in the PyTorch Ecosystem

# Lessons from vLLM

Crefeda Rodrigues & Fadi Arafeh

© 2026 Arm

# vLLM Software Stack on Arm



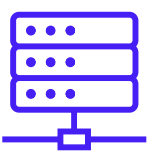

arm

© 2026 Arm

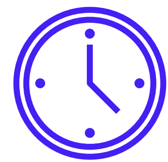
# Practical Deployments for CPU Inference

**AI head node**: Leading the orchestration and control of AI systems, including coordinating accelerators

**Hybrid workload processing:** CPU can be used for mixed workloads including general purpose computing and AI specific workloads



**Leverages existing CPU infrastructure:** uses standard CPU hardware in server deployments



**Off-peak batching**: use idle CPU capacity for offline inference



© 2026 Arm

arm

# State of vLLM on AArch64 CPU before

# Build / CI / Reliability

[Feature]: CI workflow for building non-CUDA Aarch64 wheels #26017

[Bug]: AArch64 build fails due to old torch version in requirements/cpu-build.txt #25578

[Bug]: CPU Backend: vllm / torch.compile issue with torch > 2.8 #28982

© 2026 Arm

# Performance issues and missing support for vLLM V1 features

[Bug]: chunked prefill disabled & max batched tokens not compatible with max model length on non-X86 CPU Backend #28981

[CPU Backend] [Bug]: Default VLLM_CPU_KVCACHE_SPACE is too small for CPU Backend #29233

[Bug]: Very low CPU utilization on AArch64 CPUs when VLLM_CPU_OMP_THREADS_BIND is set #27369

[Bug]: Incorrect outputs with batch size > 1 on AArch64 CPU #27034

© 2026 Arm

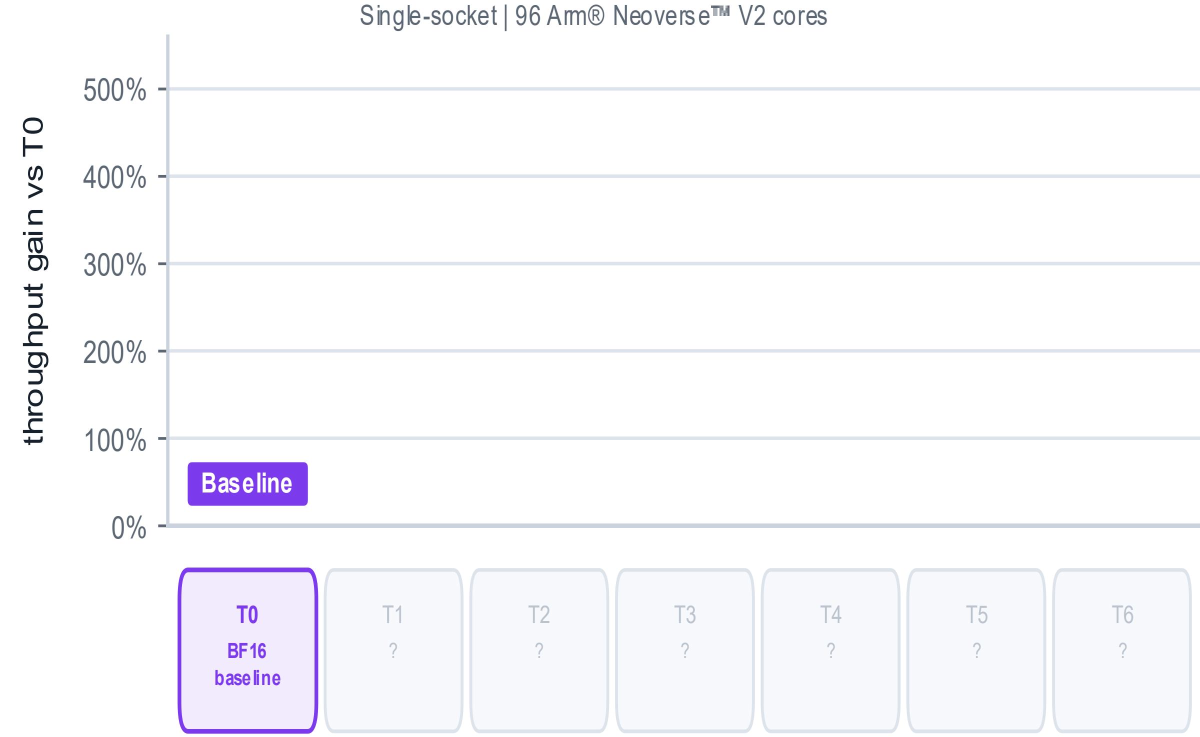
arm

# Non-Kernel Optimization

© 2026 Arm

# Non-Kernel Optimization

- Once issues were fixed, it was time to look at performance
- Throughput benchmark:
  - Llama 3.1 8b, BF16
  - 256 input / 256 output tokens
- The throughput was much lower than expected
- > 80% of the time was spent in layers dispatched to highly optimized GEMM implementations
- The standalone GEMM kernels run within 80-90% of the speed-of-light
- We need to optimize at the system level





© 2026 Arm

# Memory Allocation

- LLMs create intense multi-threaded memory pressure on CPUs
- With glibc **malloc**, large tensor allocations are not reused well
  → high rate of page faults
  → noisier benchmarks
- **mimalloc** improves large-allocation reuse and reduces thread contention
- We made mimalloc the default PyTorch allocator on Arm
- **133% higher throughput**

© 2026 Arm

# Improving Synchronization at high thread counts

- In some cases, more threads regresses performance
- Paged attention perf report – **74%** on `gomp_iter_dynamic_next`!

```
--97.94%--gomp_thread_start
    |
    |--90.08%--paged_attention_v1_impl<float,…>
        |
        |--74.07%--gomp_iter_dynamic_next
        |--7.00%--reduceValueBlock<float,…>::λ(int)
```

- **Applies to other ops across the stack**
- Slow LL/SC loop in the hot path

```
bool gomp_iter_dynamic_next(long *pstart, long *pend) {

  ws = current_thread()->work_share;
  chunk = ws->chunk_size;
  start = atomic_fetch_add(&ws->next, chunk);
  ...
}
```

```
// Read Modify Write

  for (;;) {
    long old = LDXR(p);          // load & reserve
    long newv = old + delta;
    int fail = STLXR(p, newv); // store if not dirty
    if (fail == 0) {
      DMB_ISH();
      return old;
    }
    // otherwise retry
  }
```



© 2026 Arm

# Improving Synchronization at high thread counts

- In some cases, more threads regresses performance
- Paged attention perf report – **74%** on `gomp_iter_dynamic_next`!

```
--97.94%--gomp_thread_start
        |
        |--90.08%--paged_attention_v1_impl<float,…>
                |
                |--1.22%--gomp_iter_dynamic_next
                |--29.0%--reduceValueBlock<float,…>::λ(int)
```

- Applies to other ops across the stack
- Slow LL/SC loop in the hot path
- **Arm's LSE's atomic RMW is up to 100x faster**

```
bool gomp_iter_dynamic_next(long *pstart, long *pend) {

  ws = current_thread()->work_share;
  chunk = ws->chunk_size;
  start = atomic_fetch_add(&ws->next, chunk);
  ...
}
```

```
// Read Modify Write

  for (;;) {
    long old = LDXR(p);          // load & reserve
 // atomically: old = *p; *p += delta; return old
LDADDAL(p, delta);
      DMB_ISH();
      return old;
    }
    // otherwise retry
  }
```

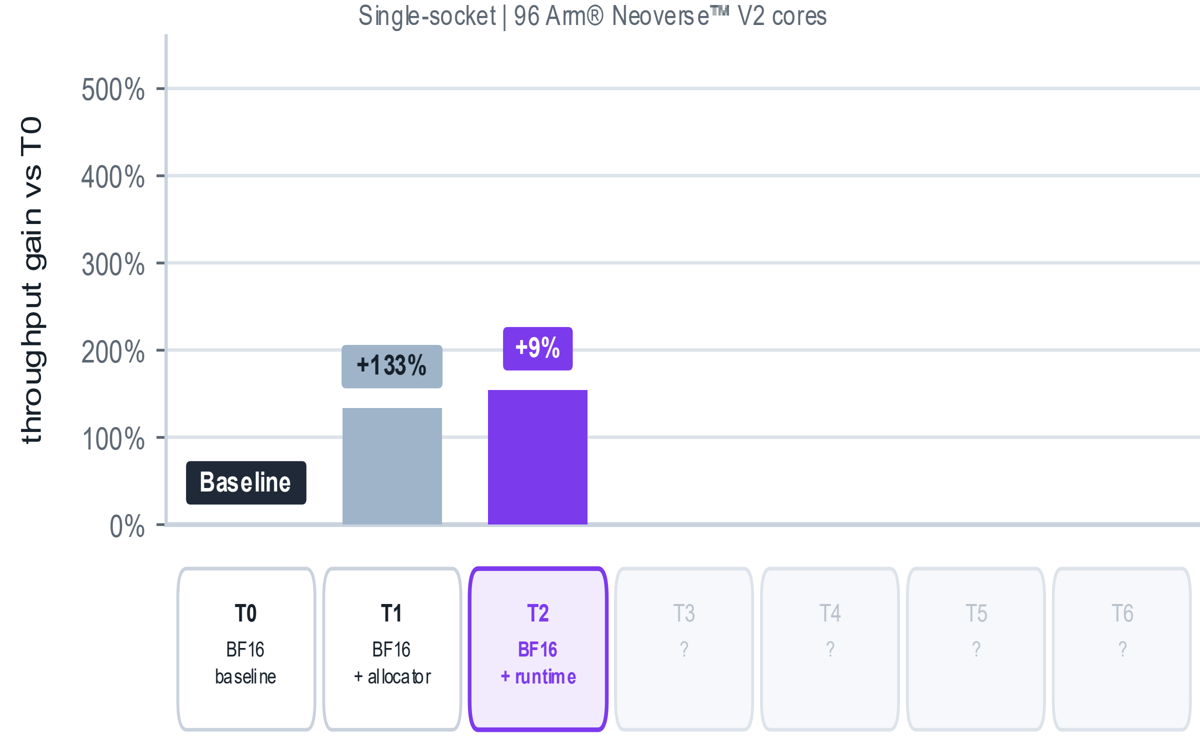

© 2026 Arm

# Improving Synchronization at high thread counts

- In some cases, more threads regresses performance
- Paged attention perf report – **74%** on `gomp_iter_dynamic_next`!

```
--97.94%--gomp_thread_start
    |
    |--90.08%--paged_attention_v1_impl<float,…>
        |
        |--1.22%--gomp_iter_dynamic_next
        |--29.0%--reduceValueBlock<float,…>::λ(int)
```

- Applies to other ops across the stack
- Slow LL/SC loop in the hot path
- Arm's LSE's atomic RMA is up to 100x faster
- We now build a performant runtime that supports Arm's LSE extension in PyTorch
- **9% higher throughput!**

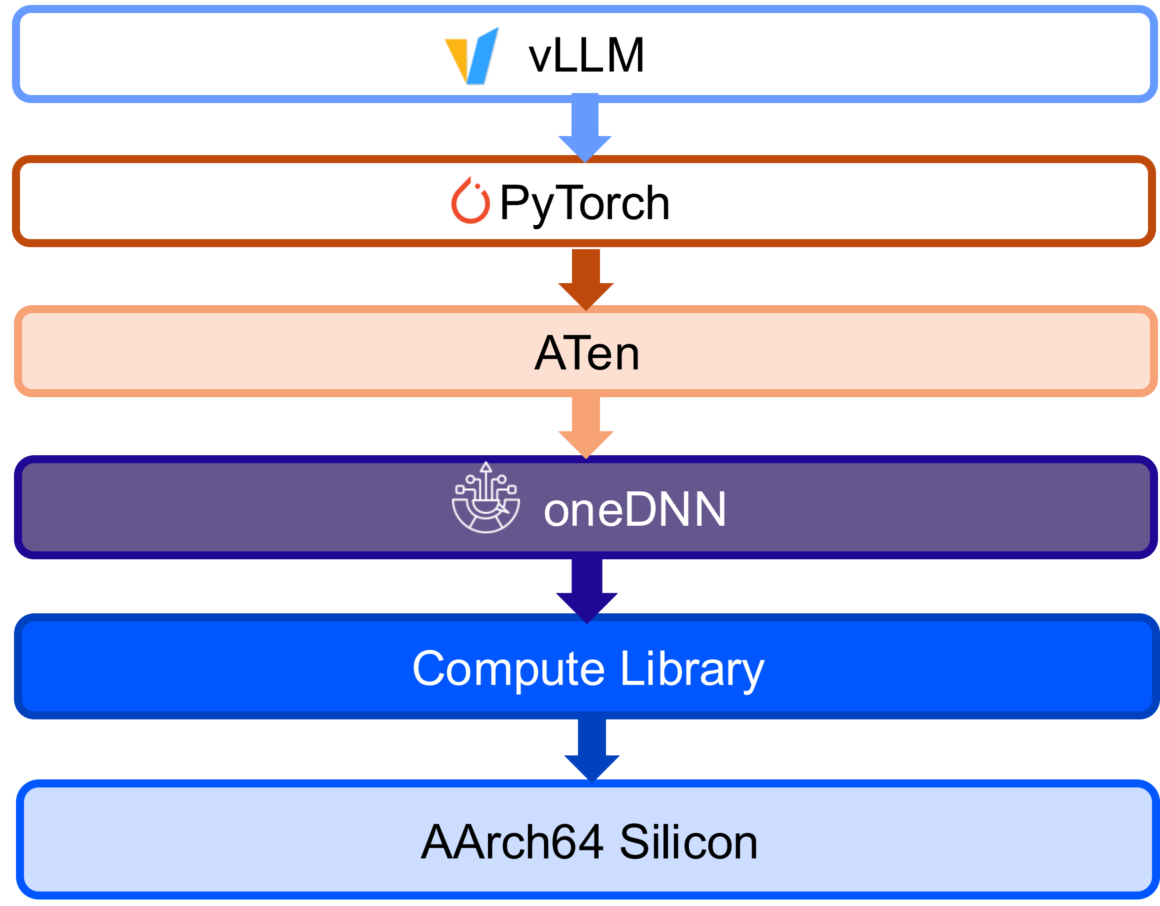



arm

© 2026 Arm

# Integration Overhead

- Matmuls go:
  **`vLLM → PyTorch → oneDNN → ACL`**
- New features take time to propagate across the stack
- Libraries are opinionated:
  deeper stack → higher API friction
- PyTorch stores weights as plain tensors and re-packs them on every call
- Packing is not free!
- Solvable with torch.compile + graph freezing
- We still needed a fast eager-mode path

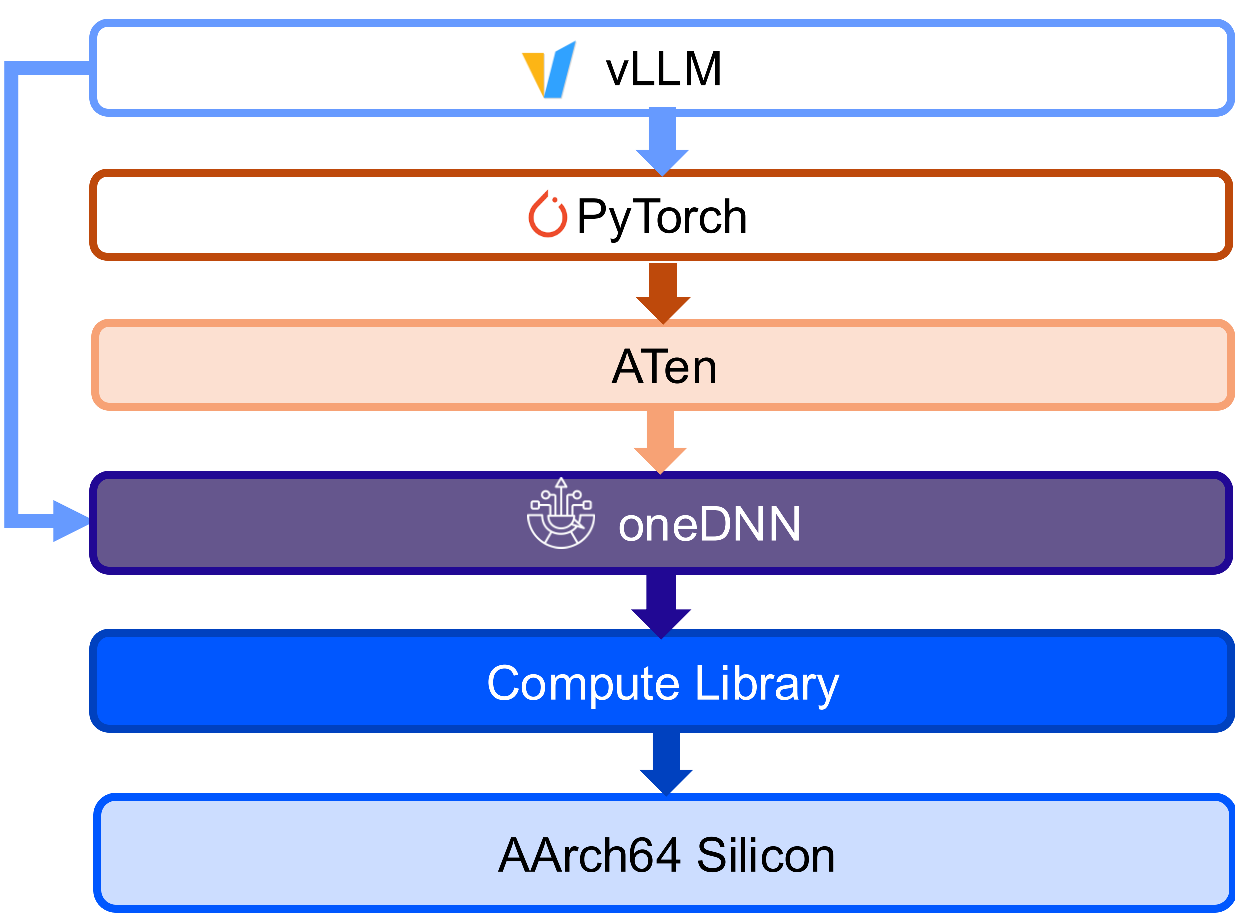




© 2026 Arm

# Integration Overhead

- PyTorch stores weights as plain tensors and re-packs them on every call
- Packing is not free!
- Solvable with torch.compile + graph freezing
- We still needed a fast eager-mode path
- oneDNN was already available in vLLM
- Extend that to support BF16 MatMuls on Arm
- **Call oneDNN/ACL directly from vLLM**
- **pre-pack weights once at model load**.
- Eliminates redundant re-packs & reduces overhead from the deep stack

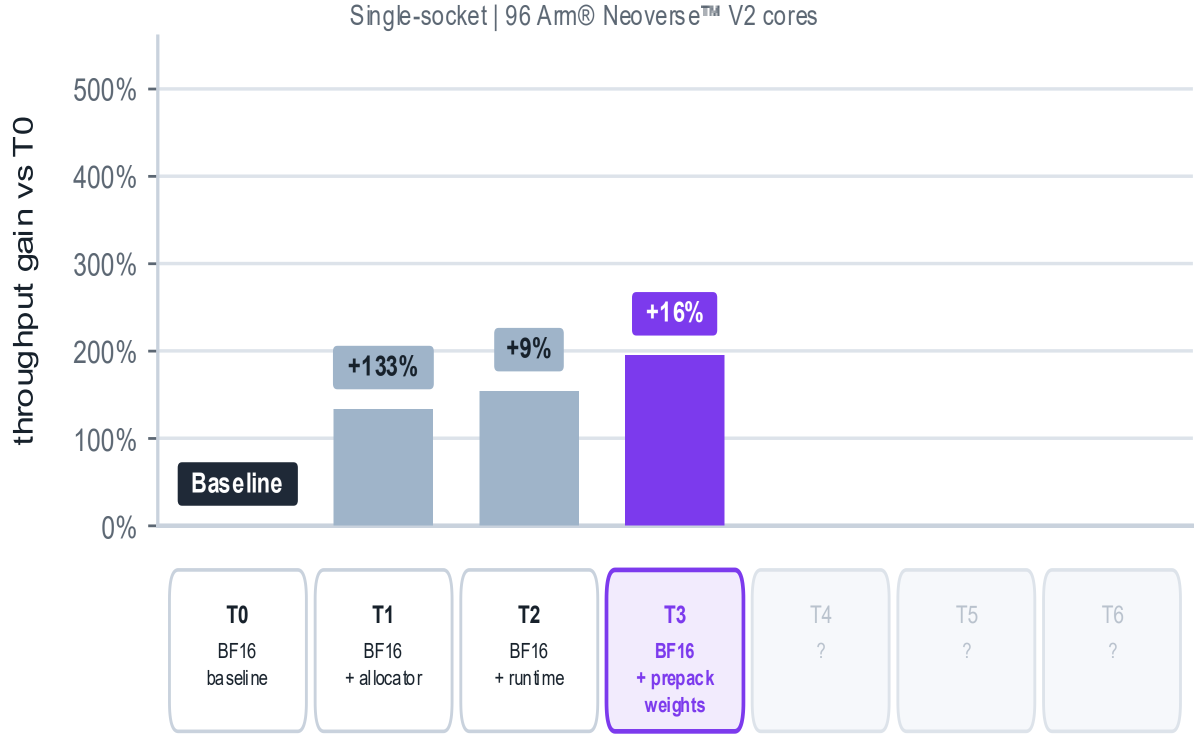




© 2026 Arm

# Integration Overhead

- PyTorch stores weights as plain tensors and re-packs them on every call
- Packing is not free!
- Solvable with torch.compile + graph freezing
- We still needed a fast eager-mode path
- oneDNN was already available in vLLM
- Extend that to support BF16 MatMuls on Arm
- **Call oneDNN/ACL directly from vLLM**
- **pre-pack weights once at model load**.
- Eliminates redundant re-packs & reduces overhead from the deep stack
- **16% higher throughput!**






© 2026 Arm

arm

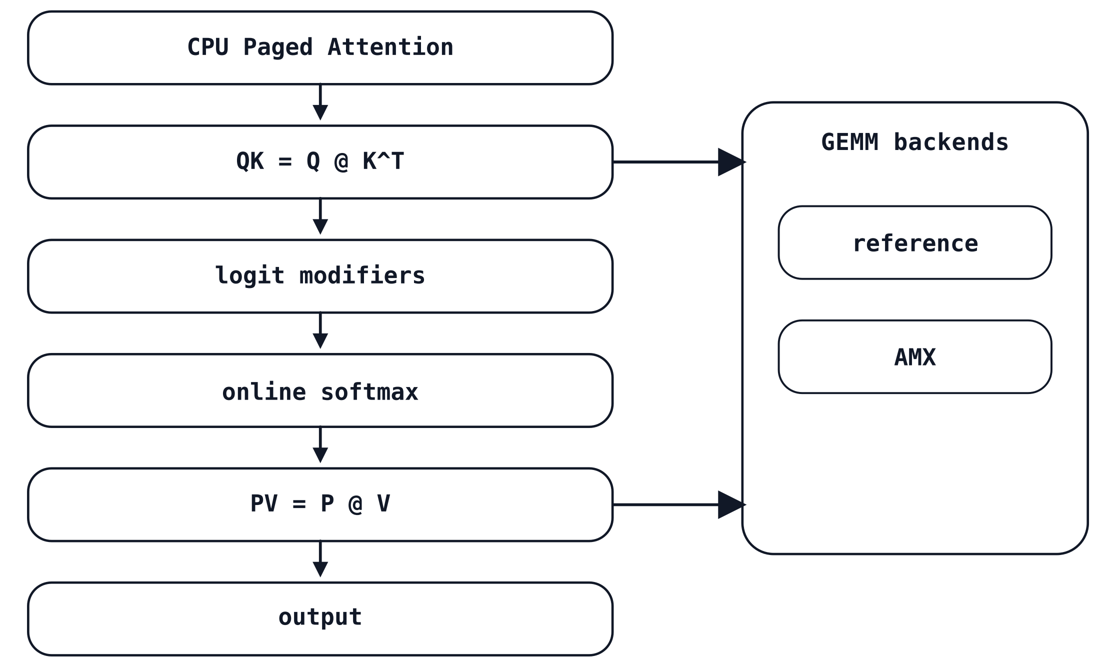
# Paged Attention

© 2026 Arm 

# Paged Attention

## In vLLM's CPU Backend

- x86-optimized implementation contributed by the ecosystem
- Supports chunked-prefill, Sliding window attention, sinks, alibi slops, etc
- Not optimized for Arm
- Let's optimize the hotspots!



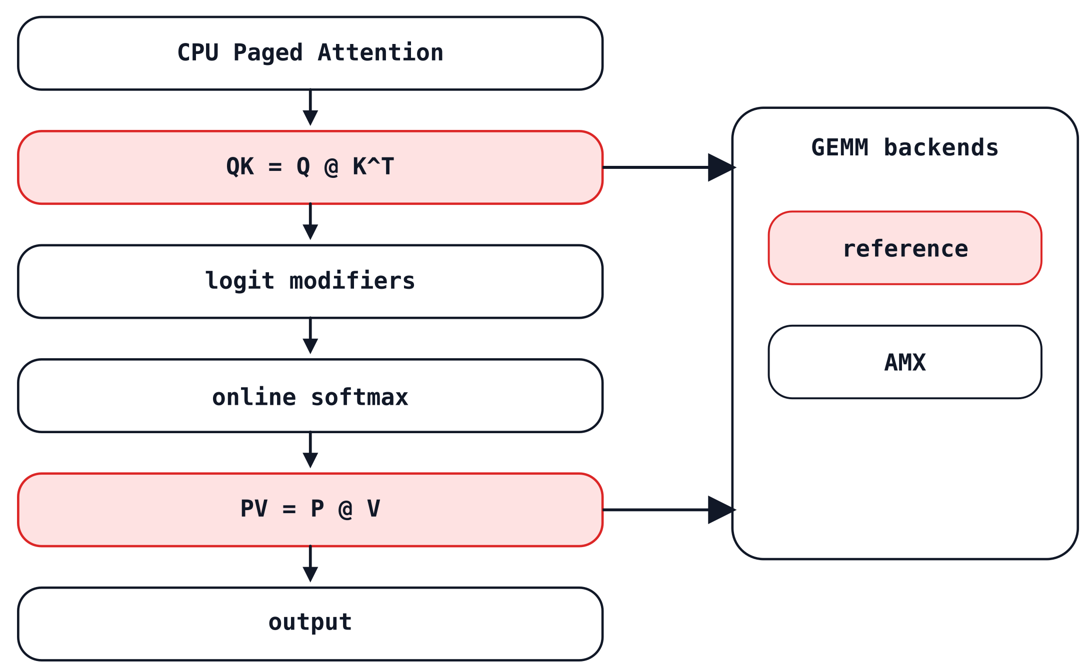


© 2026 Arm

# Paged Attention

## Optimization for Arm

- The Q-K, P-V GEMMs are an obvious starting point



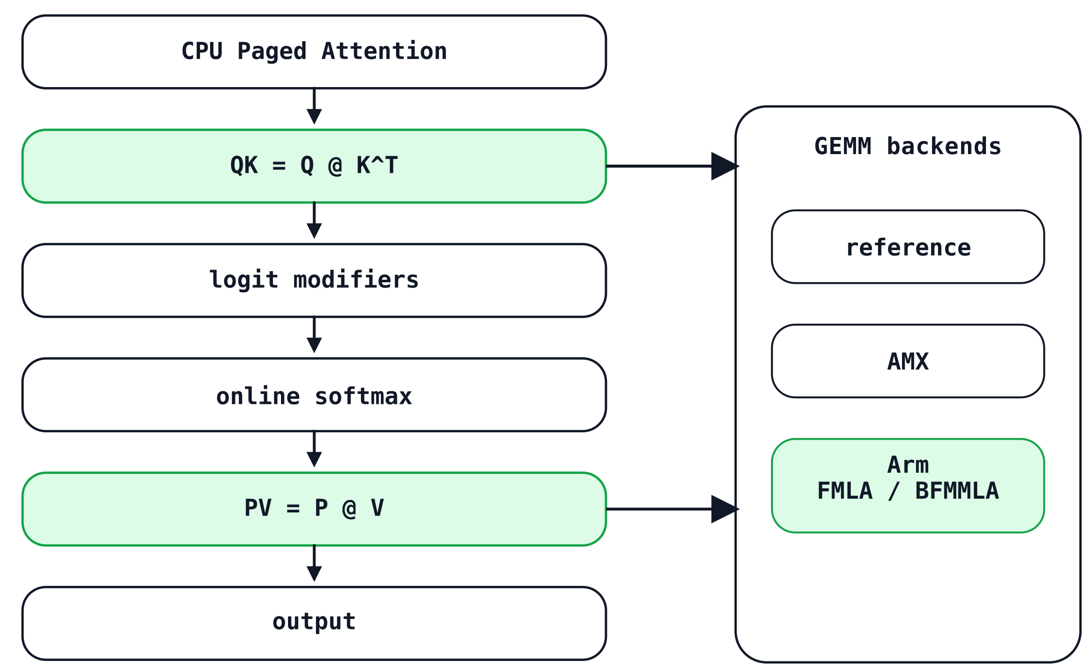


© 2026 Arm

# Paged Attention

## Optimization for Arm

- The Q-K, P-V GEMMs are an obvious starting point
- We accelerate them with custom FMLA & BFMMLA micro-kernels



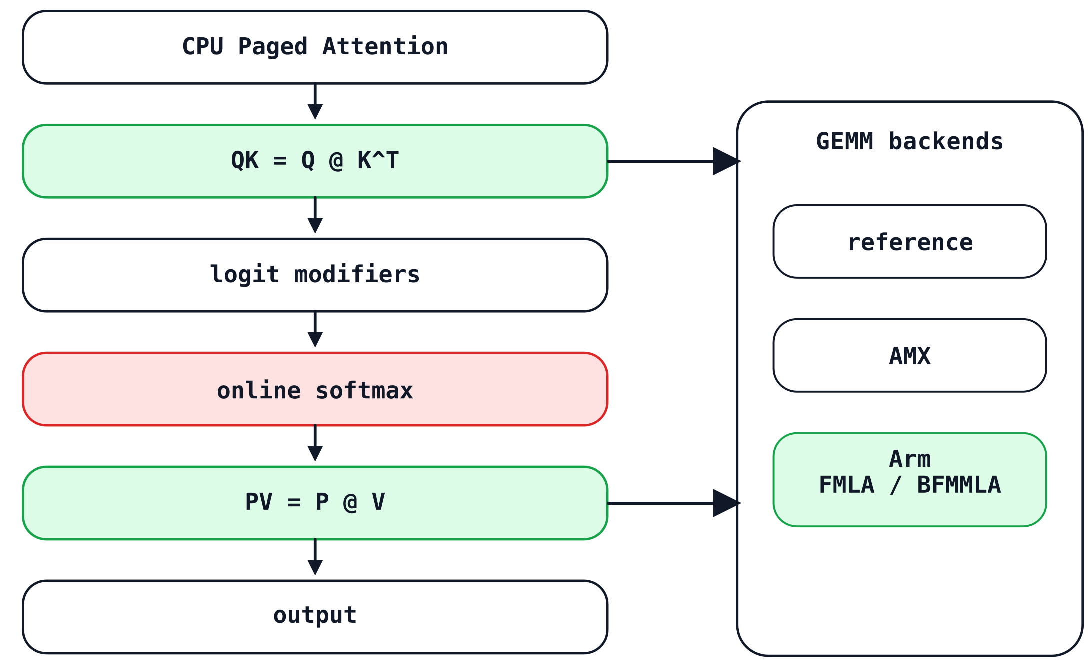



© 2026 Arm

# Paged Attention

## Optimization for Arm

- The Q-K, P-V GEMMs are an obvious starting point
- We accelerate them with custom FMLA & BFMMLA micro-kernels
- Now, the **softmax** is ~ **30%** of runtime



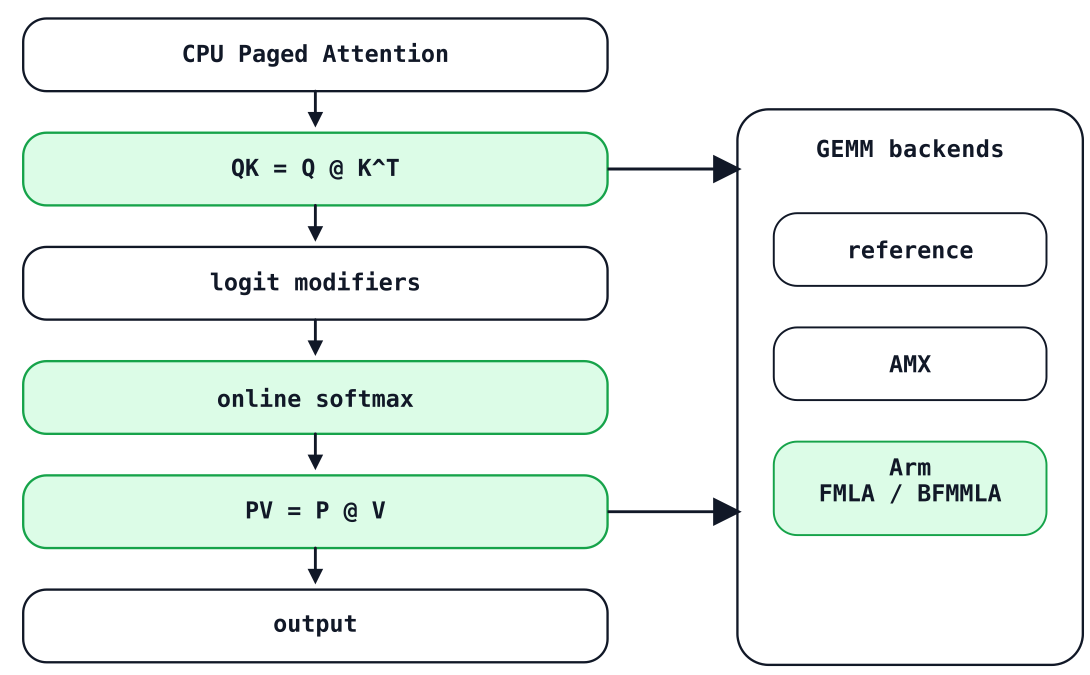


© 2026 Arm

# Paged Attention

## Optimization for Arm

- The Q-K, P-V GEMMs are an obvious starting point
- We accelerate them with custom FMLA & BFMMLA micro-kernels
- Now, the **softmax** is ~ **30%** of runtime
- We accelerate **exp** with a vectorized 3rd degree polynomial approximation
- Up to **4x faster** paged attention on Arm!



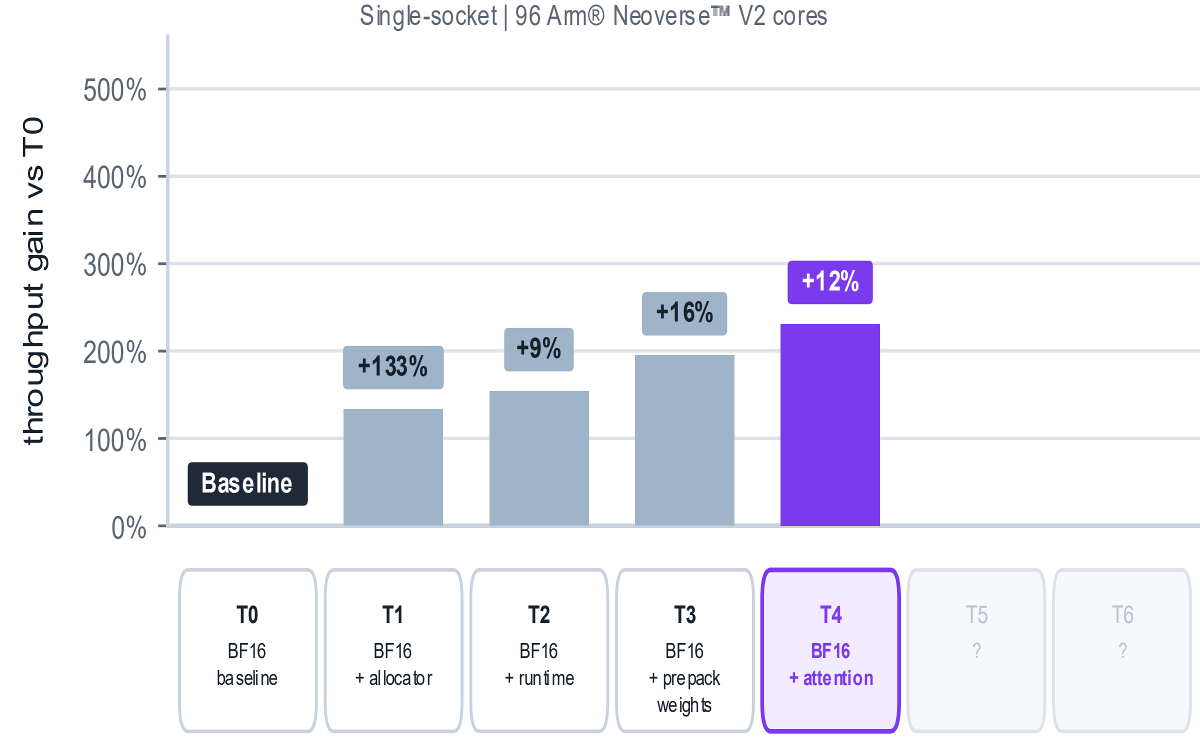


© 2026 Arm

# Paged Attention

## Optimization for Arm

- The Q-K, P-V GEMMs are an obvious starting point
- We accelerate them with custom FMLA& BFMMLA micro-kernels
- Now, the **softmax** is ~ **30%** of runtime
- We accelerate **exp** with a vectorized 3rd degree polynomial approximation
- Up to **4x faster** paged attention on Arm!
- **12% higher throughput!**






© 2026 Arm

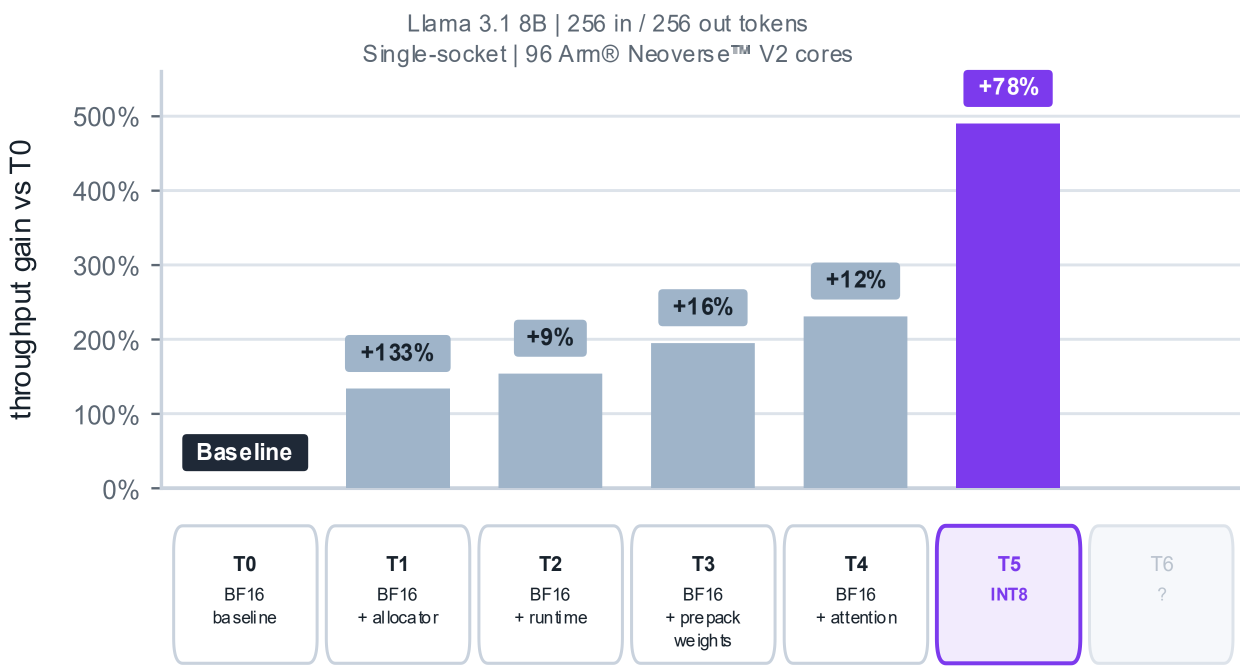
arm

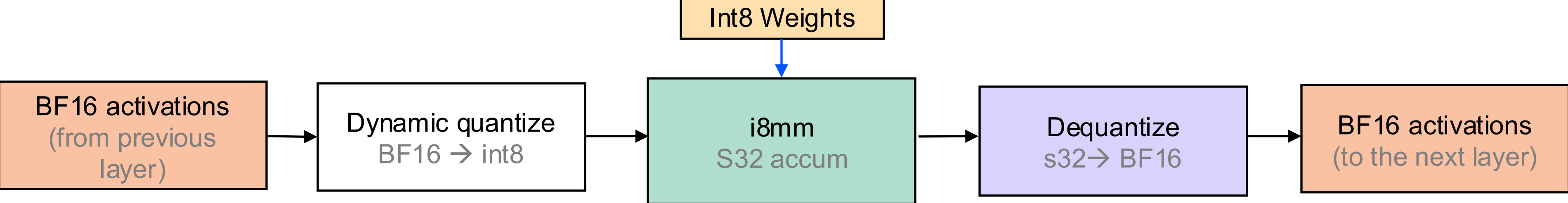
# Quantization

© 2026 Arm

# INT8 W8A8

- Memory-bandwidth savings + fast INT8 i8mm instructions ~2x faster to BF16.
- Accelerated through **oneDNN**
- The oneDNN ecosystem effect
  - Fujitsu contributed the SVE256 int8 kernels
  - Intel did the plumbing work to enable these in vLLM
  - We contributed the SVE128 int8 kernels
- Reaches 99% BF16 accuracy on MLPerf's Llama 3.1 8b benchmarks

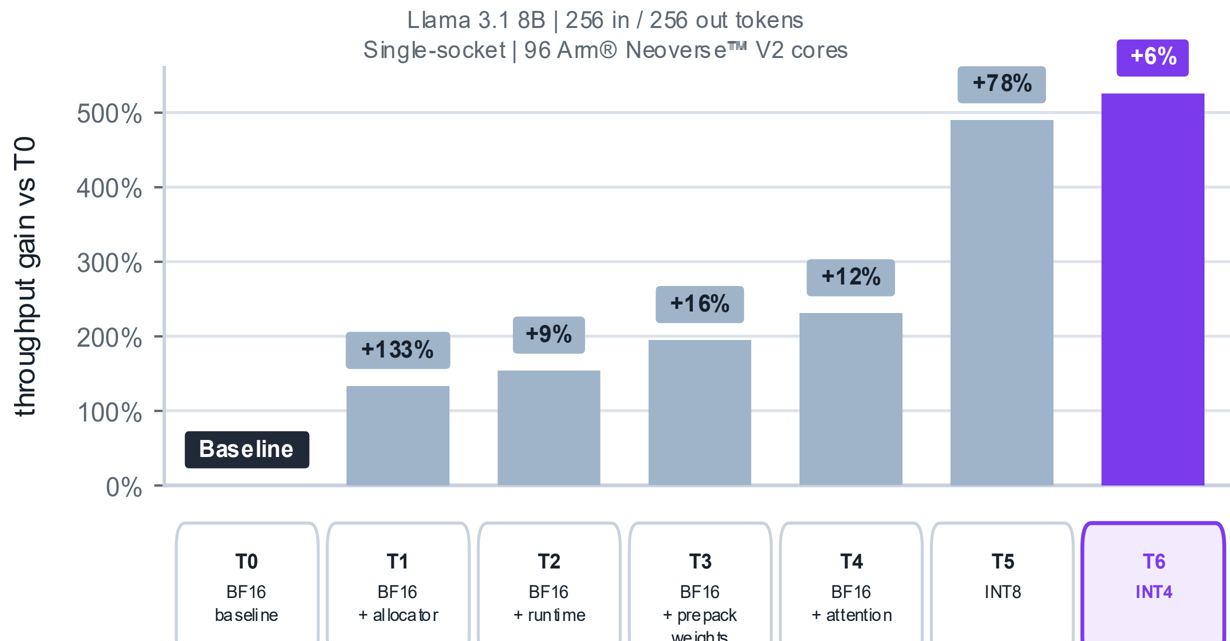


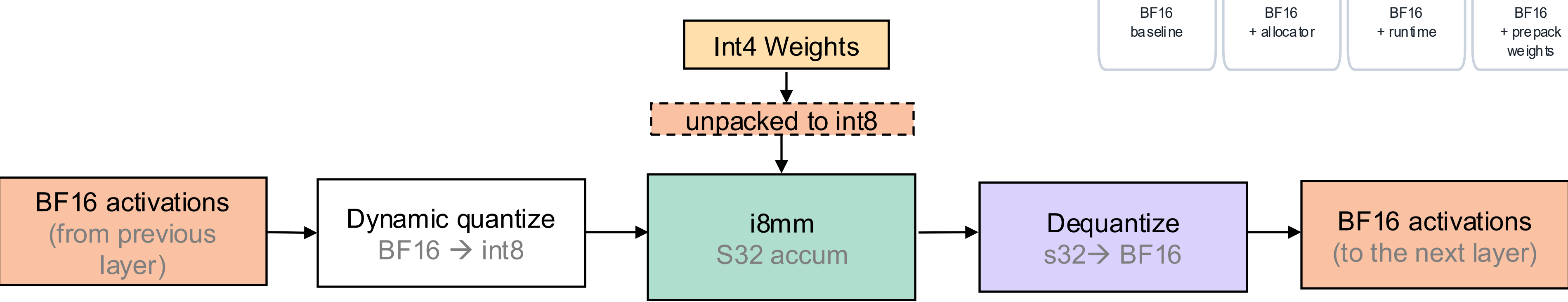



© 2026 Arm

# INT4 W4A8

- INT4 halves memory compared to INT8, fits larger models on CPU.
- **KleidiAI** int4 kernels accelerate linear layers implemented using **i8mm** instructions.
- W4A8 channelwise Llama-3.1-8B (GPTQ) reaches 99% BF16 accuracy on MLPerf's Llama 3.1 8b benchmarks





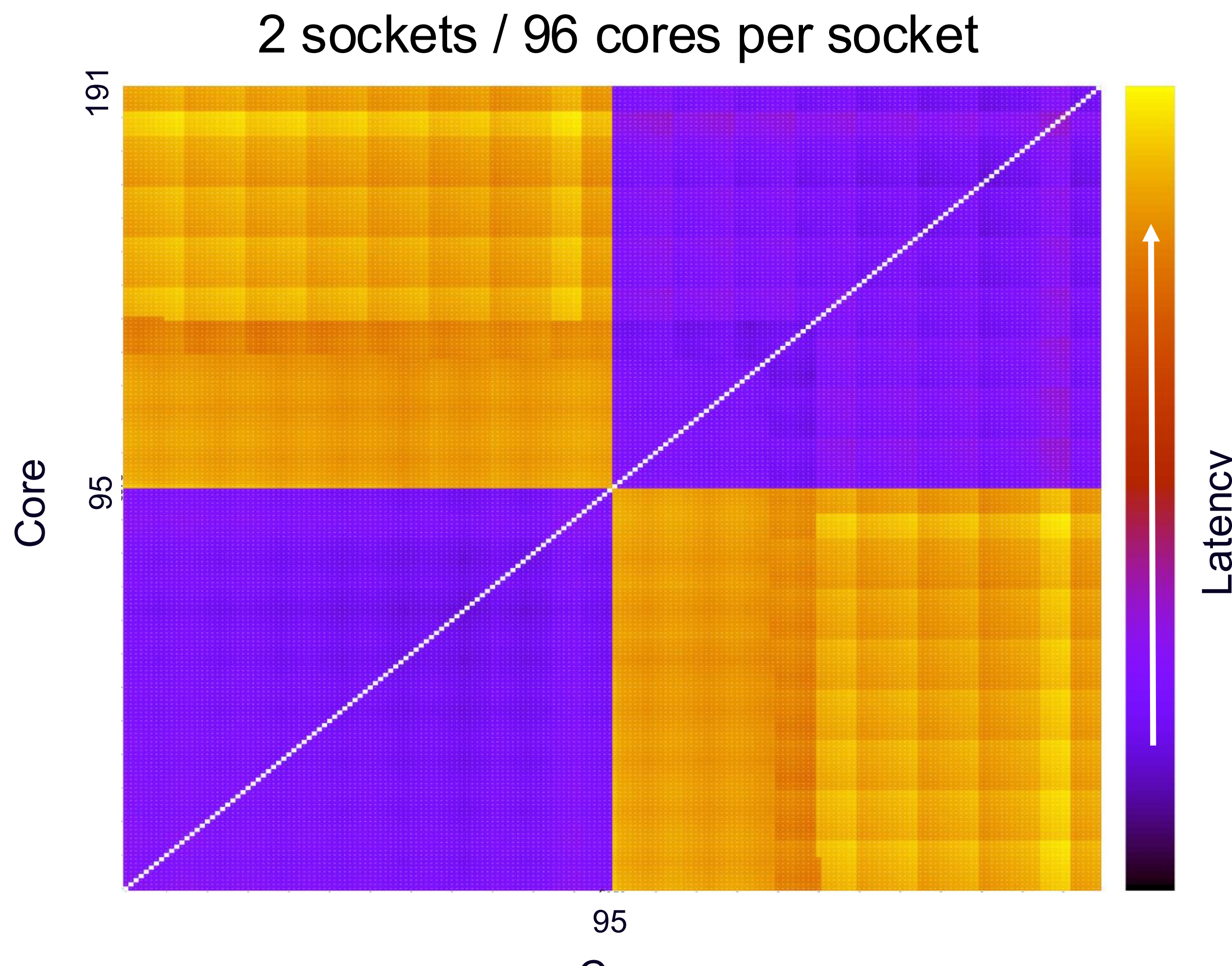
© 2026 Arm

arm

# Multi-NUMA Inference

# More Cores, More Complex Topology

- Modern servers give us **lots of cores**
- Higher core counts often mean **multiple sockets and/or NUMA domains**
- The machine is no longer a **uniform memory system**
- Cross-socket / remote-NUMA access adds more overhead





© 2026 Arm

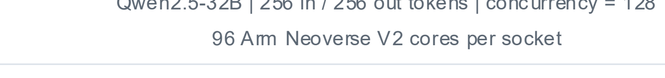
# More Cores, More Complex Topology

- Modern servers give us **lots of cores**
- Higher core counts often mean **multiple sockets and/or NUMA domains**
- The machine is no longer a **uniform memory system**
- Cross-socket / remote-NUMA access adds more overhead

© 2026 Arm

# More Cores, More Topology

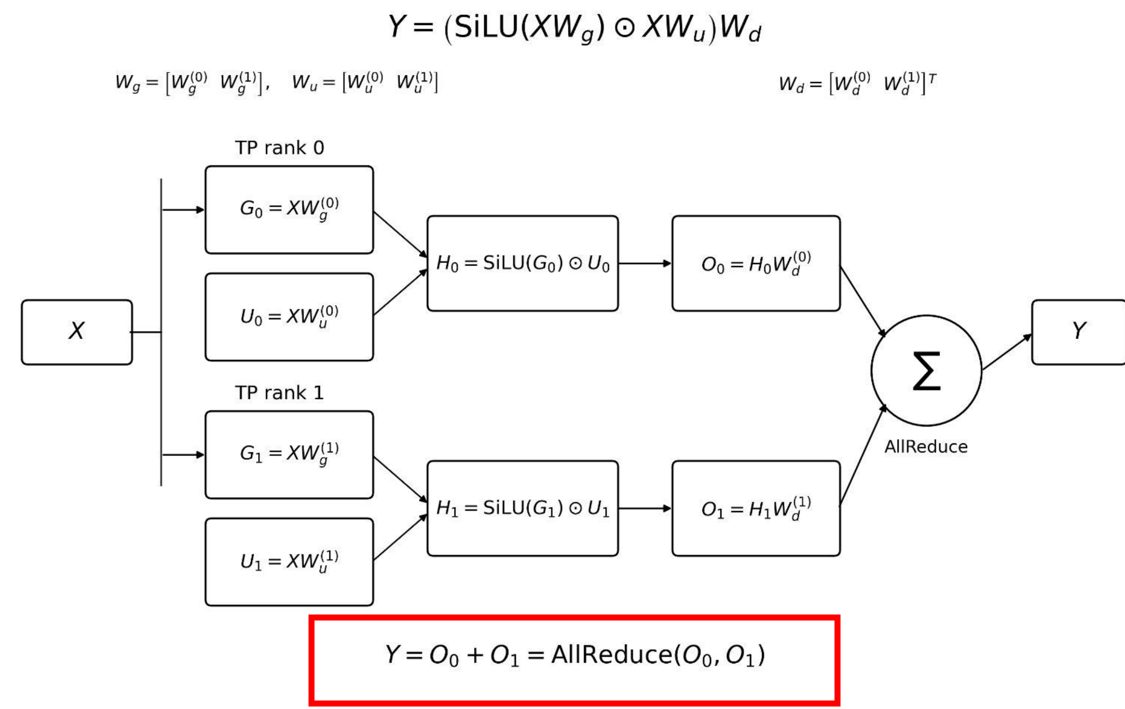
- Modern servers give us **lots of cores**
- Higher core counts often mean **multiple sockets and/or NUMA domains**
- The machine is no longer a **uniform memory system**
- Cross-socket / remote-NUMA access adds more overhead
- **Spanning requests across 2 sockets yields worse latency than single socket**



# Scaling Across Sockets / NUMAs

- Keep execution **local** whenever possible
- For throughput: run a **model replica** on each socket
- For latency: you may need more compute and memory bandwidth than one socket can provide
- **Partition the model across sockets**
- **Tensor parallelism!**
- **Each NUMA region (rank) operates on its weight shard**
- **Ranks communicate to share results**

tensor-parallel in LLama style MLP



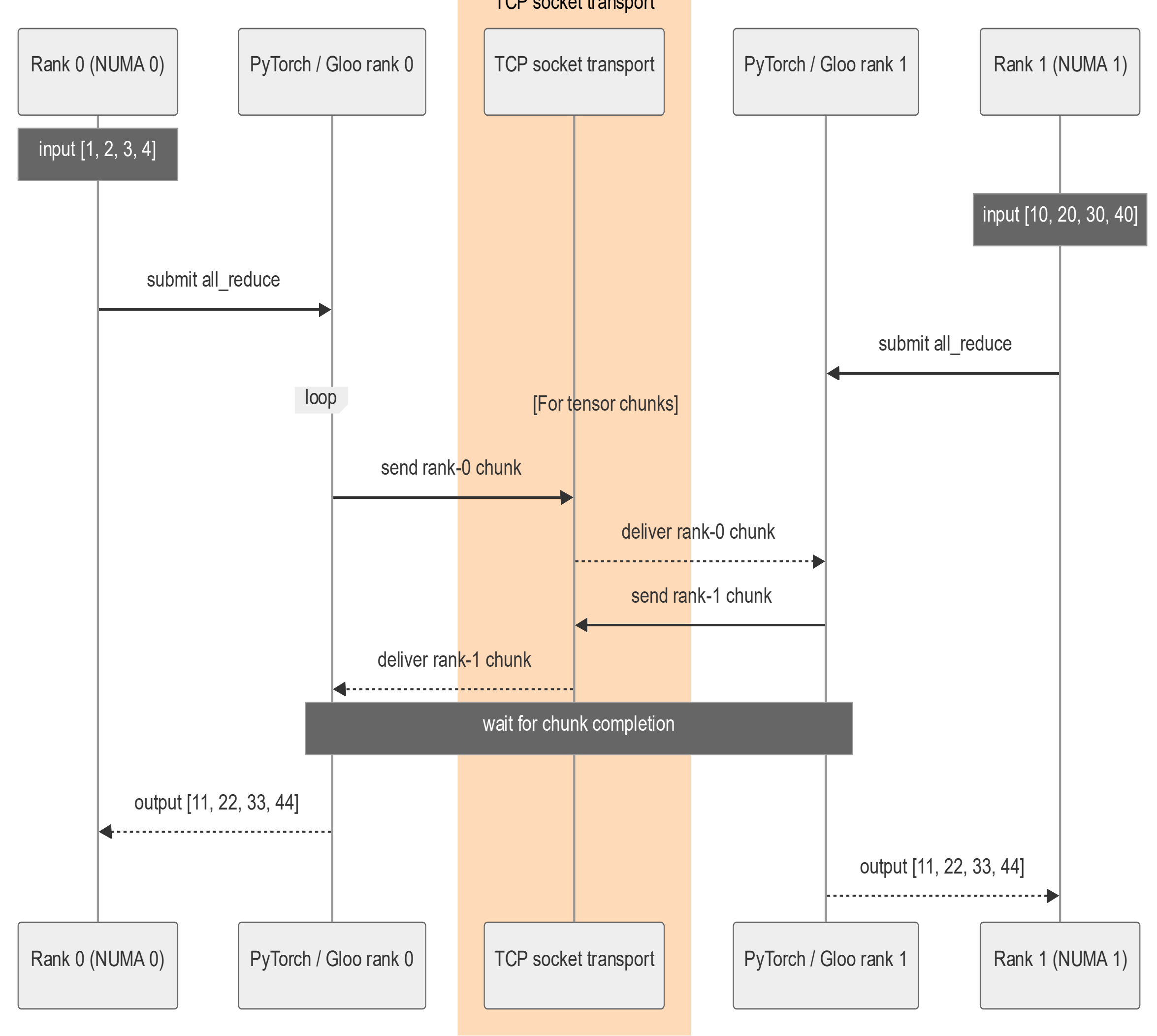


© 2026 Arm

# all-reduce with 2 sockets

## torch.distributed

- Communication through torch.distributed → Gloo
- Generic Message Passing through TCP sockets
- We have 2 ranks on the same machine
- … but paying overhead as if the ranks were on different machines
- > 50% of inference time waiting





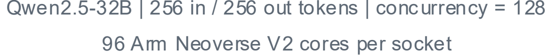
© 2026 Arm

# tensor-parallel with 2 sockets

## torch.distributed

- Communication through torch.distributed → Gloo
- Generic Message Passing through TCP sockets
- We have 2 ranks on the same machine
- … but paying overhead as if the ranks were on different machines
- > 50% of inference time waiting
- **Latency with tp=2 and this communication path is worse that that of a single socket**

Latency with Multi-Socket Inference

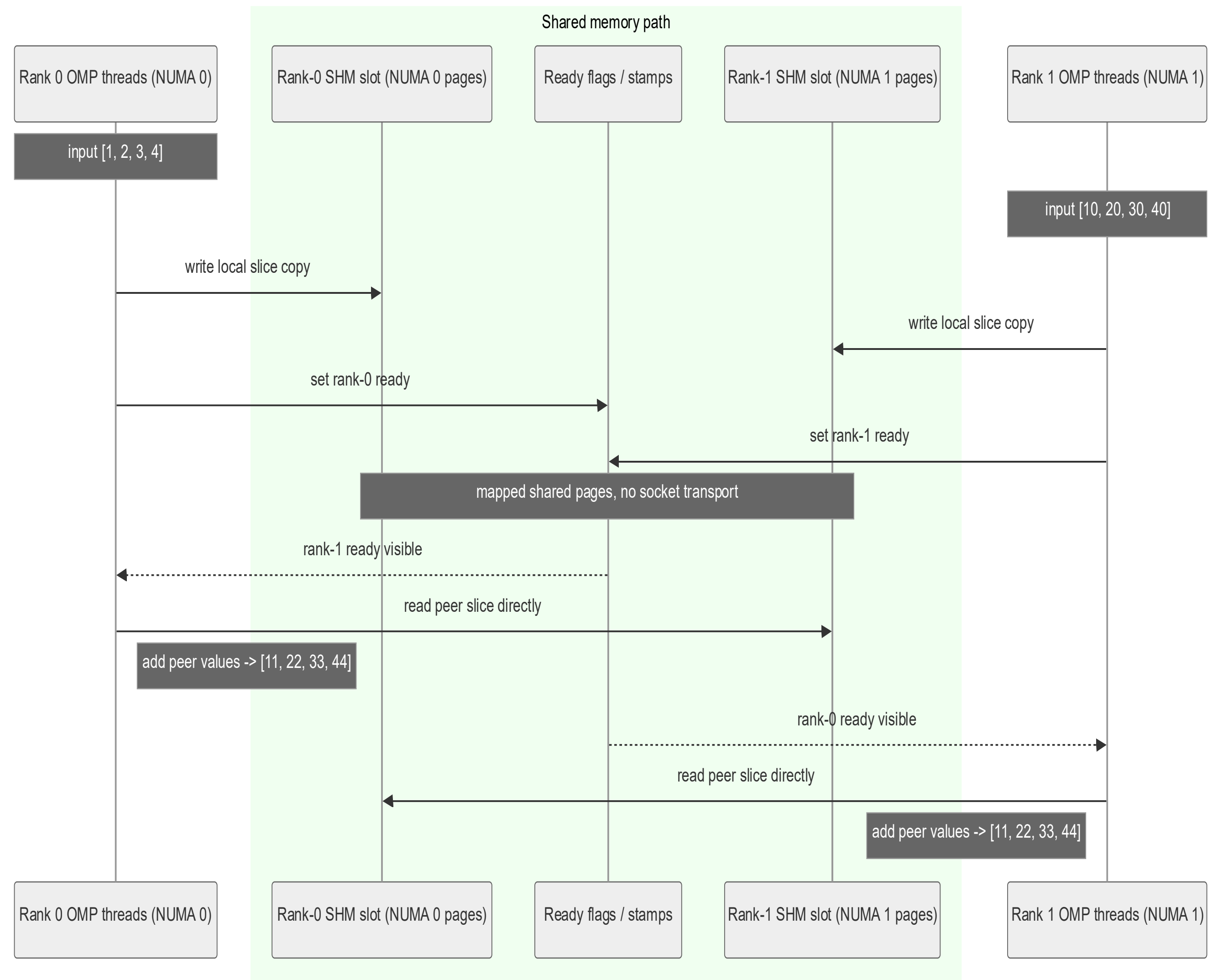


© 2026 Arm

# all-reduce with 2 sockets

## vLLM's custom SHM communicator

- x86-oriented implementation contributed by community
- Now, enabled for Arm!
- Communicate through **mapped shared memory**
- Minimal cross NUMA traffic
- Lightweight synchronization
- Communication path matches the system topology



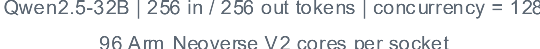

arm

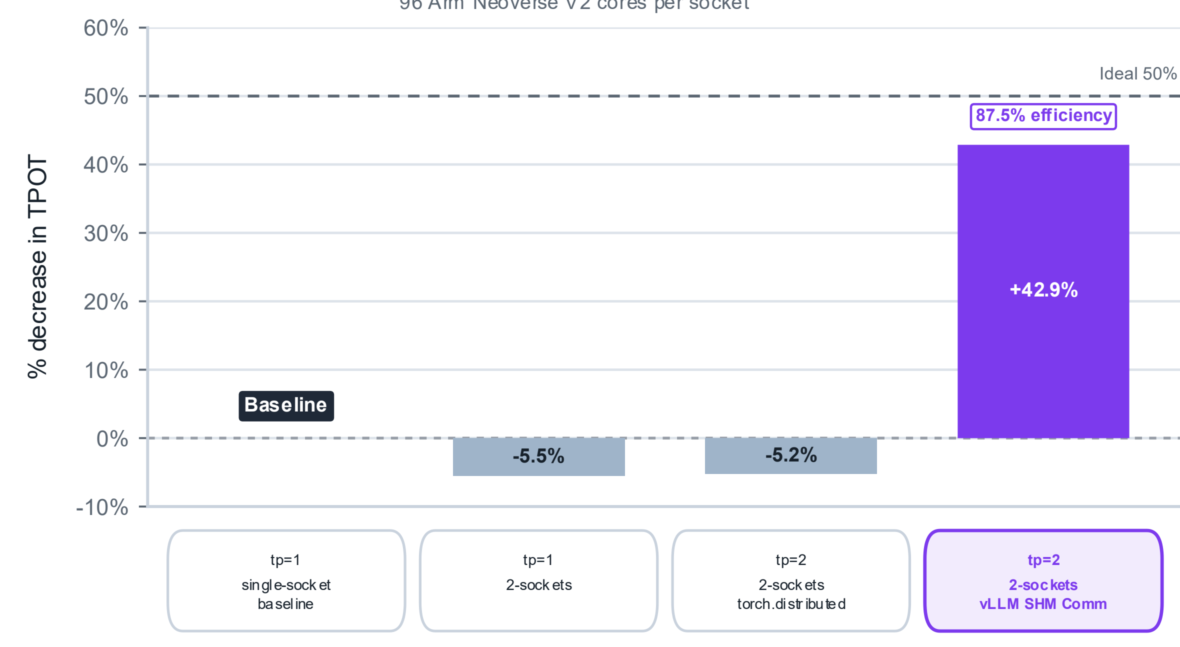
© 2026 Arm

# tensor-parallel with 2 sockets

## vLLM's custom SHM communicator

- x86-oriented implementation contributed by community
- Communicate through **mapped shared memory**
- Minimal cross NUMA traffic
- Lightweight synchronization
- Communication path matches the system topology
- **Latency decreases by 43%**





© 2026 Arm

# Recap

- Our journey accelerating vLLM on Arm CPUs
- Started with correctness issues, crashes, and poor CPU utilization
- Now supports out-of-the-box low-precision inference across sockets
- Achieved through ecosystem collaboration
- And optimizations at the kernel, system, and integration layers
- Across Arm Compute Library, **oneDNN**, **KleidiAI**, **OpenBLAS**, **PyTorch**, and **vLLM**
- And **it will only get better**!

arm COMPUTE LIBRARY

arm KleidiAI

$\begin{bmatrix} Op \\ BL \end{bmatrix}^\top \times \begin{bmatrix} en \\ AS \end{bmatrix}$

© 2026 Arm

arm

Tack
ಧನ್ಯವಾದಗಳು
Merci
Danke
Gracias
Grazie
谢谢
ありがとう
Asante
Thank you
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה
ధన్యవాదములు
Köszönöm

The Arm trademarks featured in this presentation are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners.

www.arm.com/company/policies/trademarks