Wave

A Symbolic Python DSL and Compiler for High-Performance Machine Learning

Harsh Menon · Oleksandr Zinenko · Gaurav Verma · Stanley Winata · Ivan Butygin
Nithin Meganathan · Sanket Pandit · William Hatch · Surya Jasper · Megan Kuo
Sahil Faizal · Ashay Rane · Aurore De Spirlet · Martin Paul Lücke

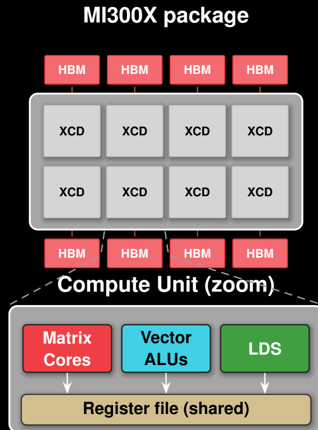
AMD · *MLSys 2026*

github.com/iree-org/wave

Motivation

AMD Instinct GPUs: where the performance lives

- Each CU mixes **Matrix Cores** (MFMA), **Vector ALUs** (VALU), and **LDS** (shared memory) — all sharing the same register file.
- MFMA instructions are wave-cooperative: a **16**×**16**×**16** `bf16` MMA needs 4 elements/thread across 64 threads, stride 16.
- Miss the MFMAs ⇒ order-of-magnitude perf penalty.
- The catch: these layouts change per MMA variant, tile size, and generation.



AMD

Why Wave: existing DSLs don't match matrix cores well

System	Granularity	What they still ask of you
CUDA / HIP	Thread	Everything (layouts, barriers, pointer math)
Triton, Helion	Block (workgroup)	Pointer/index arithmetic, shared-memory mgmt
TileLang, cuTile	Tile	Explicit tiling + schedule choices
Halide / TVM	Algorithm + Schedule	User writes the schedule per HW
Exo	Rewrites	Manual, correctness-preserving rewrites
Wave	Wave / subgroup	Compiler derives address math from constraints

Why Wave is better

- **Matches hardware granularity.** MFMA executes at the wave level; a wave-level DSL aligns 1:1 with the instruction and removes the "who owns which element" reasoning from the user.
- **Decouples *what* from *where*.** Same kernel source → new tile size, new MMA variant, new GPU: no kernel rewrite, only a new constraint list.
- **Symbolic across the boundary.** Shapes stay symbolic through MLIR, so one source supports prefill *and* decode attention, many shapes per model.

AMD

The hidden cost: addresses you'd otherwise write by hand

For a single `read` of A feeding a $16\times16\times16$ `mma` on AMD CDNA3:

$$M \to (W_x \cdot \texttt{BLOCK_M} + \lfloor T_x/64 \rfloor \cdot (\texttt{BLOCK_M}/2) + T_x \bmod 16,\ 1,\ 1)$$

$$K \to (i_K \cdot \texttt{BLOCK_K} + 4 \cdot \lfloor (T_x \bmod 64)/16 \rfloor,\ 4,\ 1)$$

In Triton/CUDA, this shows up as explicit pointer arithmetic:

```
offs_m = pid_m * BLOCK_M + tl.arange(0, BLOCK_M)          # per-block M
offs_k = tl.arange(0, BLOCK_K)                            # per-block K
a_ptrs = a_ptr + (offs_m[:, None] * stride_am             # re-derive
                  + offs_k[None, :] * stride_ak)          #   everything
#  ... and again for B, and again when you change the MMA variant,
#  ... and again for swizzled shared-memory tiles, and again for BSHD, ...
```

- Every tile size, MMA variant, or layout tweak touches these formulas.
- Wave's compiler derives the index math automatically from a few constraints — the explicit pointer arithmetic disappears from user code.

AMD

The Wave DSL

AMD

Wave: two design principles

1. Implicit Indexing

Address computation is derived by the compiler from hardware characteristics and a small set of distribution constraints.

Kernel authors never write offsets, thread IDs, or pointer math.

2. Symbolic Mapping

Shapes, tile sizes, and mapping parameters stay symbolic (`sympy`) deep into compilation.

Hardware and shape specialization happens late.

Programming model sits between SIMT and block-level: the **wave / subgroup** is the unit of authoring. **One kernel source** $\rightarrow$ many shapes, many MMA variants, multiple GPU families.

AMD

Wave at a glance — GEMM kernel

Mixed-precision $C = A \cdot B^\top$, $M \times N \times K$, all shapes symbolic.

```
@wave(constraints)
def mm(
    a: Memory[M, K, ADDRESS_SPACE_0, f16],
    b: Memory[N, K, ADDRESS_SPACE_0, f16],
    c: Memory[M, N, ADDRESS_SPACE_1, f32]):
```


AMD

Wave at a glance — GEMM kernel

Mixed-precision $C = A \cdot B^\top$, $M \times N \times K$, all shapes symbolic.

```
@wave(constraints)
def mm(
    a: Memory[M, K, ADDRESS_SPACE_0, f16],
    b: Memory[N, K, ADDRESS_SPACE_0, f16],
    c: Memory[M, N, ADDRESS_SPACE_1, f32]):
```

```
  c_reg = Register[M, N, f32](0.0)
```


AMD

Wave at a glance — GEMM kernel

Mixed-precision $C = A \cdot B^{\top}$, $M \times N \times K$, all shapes symbolic.

```
@wave(constraints)
def mm(
    a: Memory[M, K, ADDRESS_SPACE_0, f16],
    b: Memory[N, K, ADDRESS_SPACE_0, f16],
    c: Memory[M, N, ADDRESS_SPACE_1, f32]):
```

```
  c_reg = Register[M, N, f32](0.0)
```

```
  @iterate(K, init_args=[c_reg])
  def loop(acc: Register[M, N, f32]) -> Register[M, N, f32]:
      a_reg = read(a)
      b_reg = read(b)
      acc   = mma(a_reg, b_reg, acc)    # invokes matrix cores
      return acc
```


AMD

Wave at a glance — GEMM kernel

Mixed-precision $C = A \cdot B^\top$, $M \times N \times K$, all shapes symbolic.

```
@wave(constraints)
def mm(
    a: Memory[M, K, ADDRESS_SPACE_0, f16],
    b: Memory[N, K, ADDRESS_SPACE_0, f16],
    c: Memory[M, N, ADDRESS_SPACE_1, f32]):
```

```
  c_reg = Register[M, N, f32](0.0)
```

```
  @iterate(K, init_args=[c_reg])
  def loop(acc: Register[M, N, f32]) -> Register[M, N, f32]:
      a_reg = read(a)
      b_reg = read(b)
      acc   = mma(a_reg, b_reg, acc)    # invokes matrix cores
      return acc
```

```
  write(loop, c)
```

- Symbolic types: M, N, K are `sympy` symbols; `ADDRESS_SPACE` too.
- Sequential reduction dim K introduced via `iterate`.
- No pointer arithmetic, no `offs_m`/`offs_n`, no thread IDs.

AMD

Wave at a glance — distribution via constraints

The *same* kernel, mapped to the device by a separate constraint list:

```
constraints = [
  # iteration dims M, N --> workgroup idx x, y
  WorkgroupConstraint(M, BLOCK_M, 0),
  WorkgroupConstraint(N, BLOCK_N, 1),
```


AMD

Wave at a glance — distribution via constraints

The *same* kernel, mapped to the device by a separate constraint list:

```
constraints = [
  # iteration dims M, N --> workgroup idx x, y
  WorkgroupConstraint(M, BLOCK_M, 0),
  WorkgroupConstraint(N, BLOCK_N, 1),

  # further split M, N across waves inside a workgroup
  WaveConstraint(M, BLOCK_M / 2),
  WaveConstraint(N, BLOCK_N / 2),
```

Wave at a glance — distribution via constraints

The *same* kernel, mapped to the device by a separate constraint list:

```
constraints = [
  # iteration dims M, N --> workgroup idx x, y
  WorkgroupConstraint(M, BLOCK_M, 0),
  WorkgroupConstraint(N, BLOCK_N, 1),

  # further split M, N across waves inside a workgroup
  WaveConstraint(M, BLOCK_M / 2),
  WaveConstraint(N, BLOCK_N / 2),

  # sequential reduction dim K is tiled
  TilingConstraint(K, BLOCK_K),
```


AMD

Wave at a glance — distribution via constraints

The *same* kernel, mapped to the device by a separate constraint list:

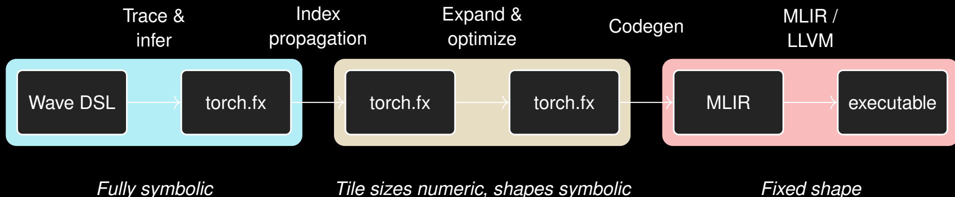
```
constraints = [
  # iteration dims M, N --> workgroup idx x, y
  WorkgroupConstraint(M, BLOCK_M, 0),
  WorkgroupConstraint(N, BLOCK_N, 1),

  # further split M, N across waves inside a workgroup
  WaveConstraint(M, BLOCK_M / 2),
  WaveConstraint(N, BLOCK_N / 2),

  # sequential reduction dim K is tiled
  TilingConstraint(K, BLOCK_K),

  # pick the MMA variant and subgroup size
  HardwareConstraint(threads_per_wave=64,
                     mma_type=MMAType.F32_16x16x16_F16),
]
```

- **Workgroup** level: which CU runs which output tile.
- **Wave** level: split the workgroup tile across waves.
- **Register** level is *inferred* from the MMA variant.
- Changing tile sizes or MMA variant never touches the kernel.

AMD

Wave compilation pipeline

Wave compilation pipeline



- **torch.fx** is the substrate: get visualization, debug tooling, graph rewrites for free.
- **MLIR** handles low-level lowering: `vector`, `scf`, `amdgpu` dialects → LLVM → binary.
- Key insight: stay symbolic *across* the boundary, specialize *once* late.

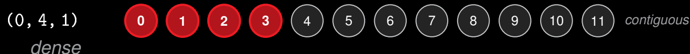
AMD

Index sequences: what each thread accesses

A triple (O, N, D) describes one strided run: **start** O, **count** N, **stride** D.

`(0, 4, 1)`

dense



`(0, 4, 3)`

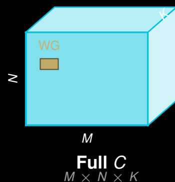
strided

- Advantage: simple (just three numbers) and expressive — a composition also captures **block-strided** MMA patterns.
- All entries are `sympy` expressions $\Rightarrow$ same representation for many shapes.

AMD

Worked example: GEMM tile hierarchy

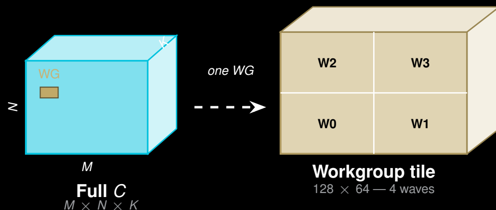
`@wave` GEMM, `BLOCK_M=128`, `BLOCK_N=64`; CDNA3 16×16×16 MMA.



AMD

Worked example: GEMM tile hierarchy

`@wave` GEMM, `BLOCK_M=128`, `BLOCK_N=64`; CDNA3 16×16×16 MMA.

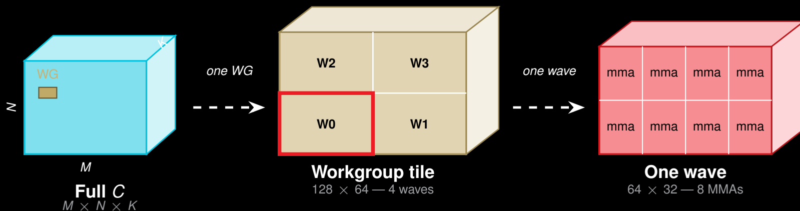


- One workgroup tile is split across **4 waves** (2×2); each wave computes a 64×32 sub-tile.

AMD

Worked example: GEMM tile hierarchy

`@wave` GEMM, `BLOCK_M=128`, `BLOCK_N=64`; CDNA3 $16 \times 16 \times 16$ MMA.

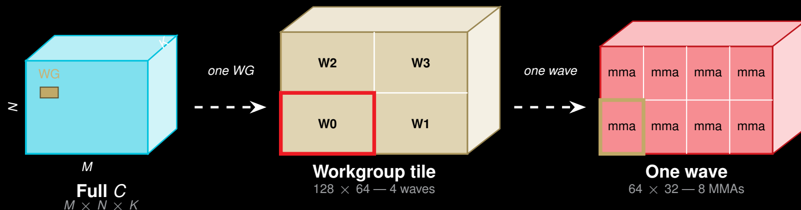


- One workgroup tile is split across **4 waves** (2×2); each wave computes a 64×32 sub-tile.
- One wave issues **8** 16×16 MMAs (dim-scaling: $M=4$, $N=2$).

AMD

Worked example: GEMM tile hierarchy

`@wave` GEMM, `BLOCK_M=128`, `BLOCK_N=64`; CDNA3 16×16×16 MMA.

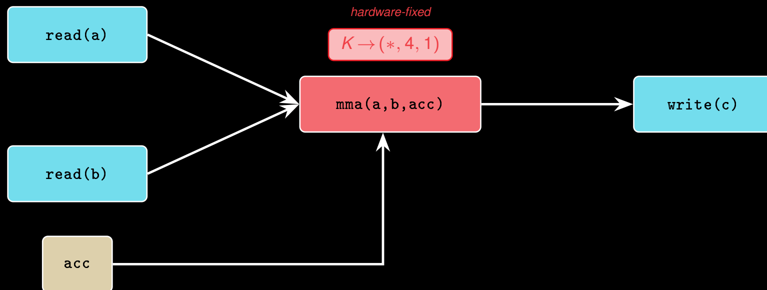


- One workgroup tile is split across **4 waves** (2×2); each wave computes a 64×32 sub-tile.
- One wave issues **8** 16×16 MMAs (dim-scaling: $M=4$, $N=2$).
- Each MMA is **wave-cooperative**: 64 threads, 4 K-elements per thread.
- None of this appears in the kernel; the compiler derives it from constraints.

AMD

Inferring index sequences via sparse dataflow

Each op carries an index triple (O, N, D). `mma`'s triple is **hardware-fixed** — connected memory ops conform.

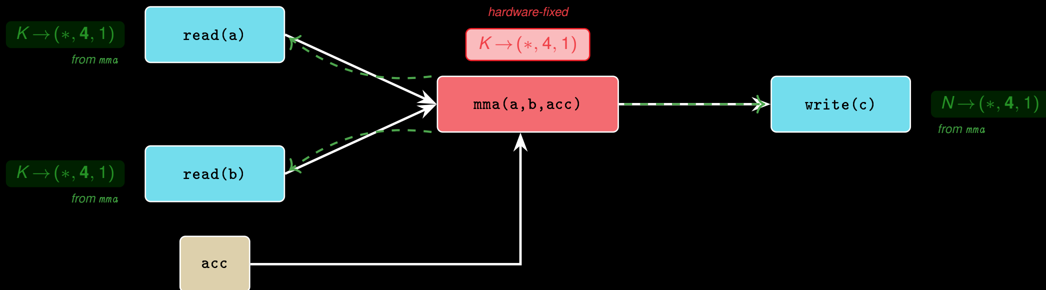


- `mma`'s triple is fixed by the MMA hardware spec (MFMA $16 \times 16 \times 16$: 4 K-elements per thread).

AMD

Inferring index sequences via sparse dataflow

Each op carries an index triple (O, N, D). `mma`'s triple is **hardware-fixed** — connected memory ops conform.

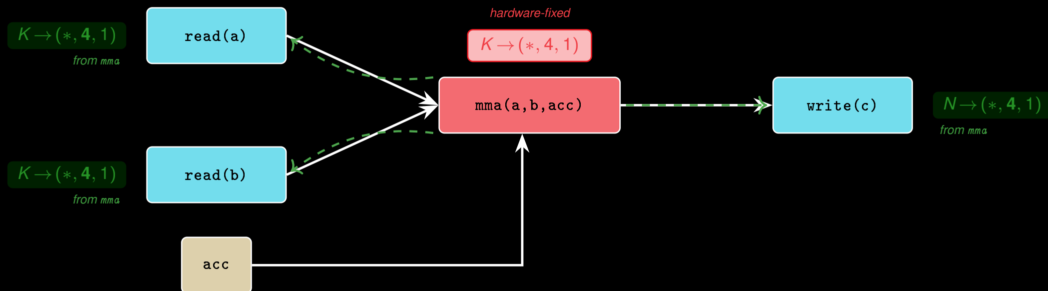


- `mma`'s triple is fixed by the MMA hardware spec (MFMA $16 \times 16 \times 16$: 4 K-elements per thread).
- Memory ops inherit the MMA layout: reads match $D_K = 4$, write matches $D_N = 4$.

AMD

Inferring index sequences via sparse dataflow

Each op carries an index triple (O, N, D). `mma`'s triple is **hardware-fixed** — connected memory ops conform.

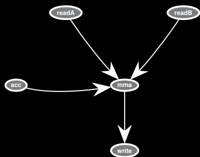


- `mma`'s triple is fixed by the MMA hardware spec (MFMA $16 \times 16 \times 16$: 4 K-elements per thread).
- Memory ops inherit the MMA layout: reads match $D_K = 4$, write matches $D_N = 4$.
- **Priority**: MMA $>$ reduction $>$ memory. Misalignment $\Rightarrow$ $10\times$ penalty.

AMD

Expansion: SIMW $\rightarrow$ SIMT

Wave expands the traced GEMM graph into fully-unrolled per-thread ops:

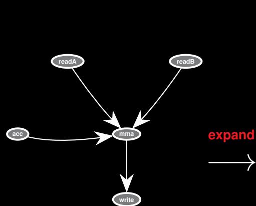


(a) traced GEMM

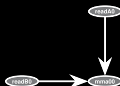
AMD

Expansion: SIMW → SIMT

Wave expands the traced GEMM graph into fully-unrolled per-thread ops:



expand
⟶



(a) traced GEMM

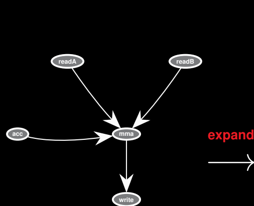
(b) expanded (M=4, N=2 to match wave tile) — base *acc/write per MMA omitted*

- Output tile (M=0, N=0): one A-read, one B-read, one MMA.

AMD

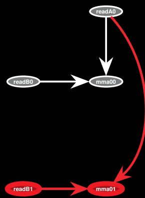
Expansion: SIMW → SIMT

Wave expands the traced GEMM graph into fully-unrolled per-thread ops:



(a) traced GEMM

expand →



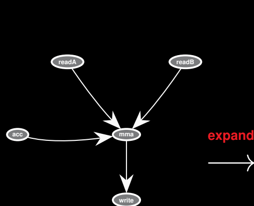
(b) expanded (M=4, N=2 to match wave tile) — base, $+N$ *acc/write per MMA omitted*

- Output tile (M=0, N=0): one A-read, one B-read, one MMA.
- Expand along N (red): new `readB1` + `mma01`; `readA0` reused.

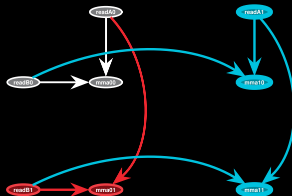
AMD

Expansion: SIMW → SIMT

Wave expands the traced GEMM graph into fully-unrolled per-thread ops:



expand →



(a) traced GEMM

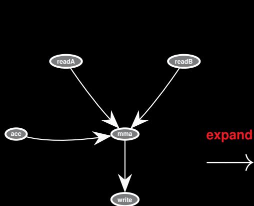
(b) expanded (M=4, N=2 to match wave tile) — base, $+N$, $+M=1$ *acc/write per MMA omitted*

- Output tile ($M=0$, $N=0$): one A-read, one B-read, one MMA.
- Expand along N (red): new `readB1` + `mma01`; `readA0` reused.
- Expand to $M=1$ (cyan): new `readA1` + 2 MMAs; both B-reads reused.

AMD

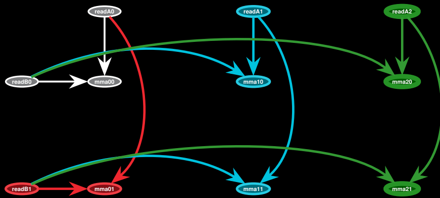
Expansion: SIMW $\rightarrow$ SIMT

Wave expands the traced GEMM graph into fully-unrolled per-thread ops:



(a) traced GEMM

expand $\longrightarrow$



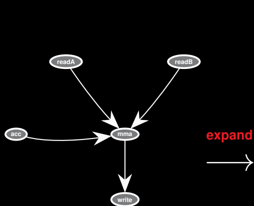
(b) expanded (M=4, N=2 to match wave tile) — base, $+N$, $+M$=1, $+M$=2 *acc/write per MMA omitted*

- Output tile (M=0, N=0): one A-read, one B-read, one MMA.
- Expand along N (red): new `readB1` + `mma01`; `readA0` reused.
- Expand to M=1 (cyan): new `readA1` + 2 MMAs; both B-reads reused.
- Continue to M=2 (green): same pattern — 1 new A-read, 2 new MMAs; B-reads reused again.

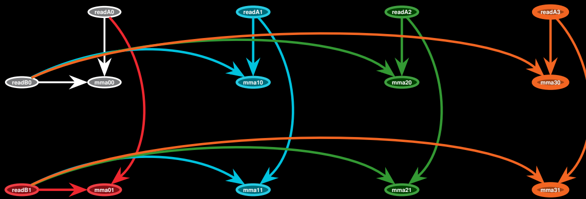
AMD

Expansion: SIMW → SIMT

Wave expands the traced GEMM graph into fully-unrolled per-thread ops:



expand →



(a) traced GEMM

(b) expanded ($M=4$, $N=2$ to match wave tile) — base, $+N$, $+M=1$, $+M=2$, $+M=3$ *acc/write per MMA omitted*

- Output tile ($M=0$, $N=0$): one A-read, one B-read, one MMA.
- Expand along N (red): new `readB1` + `mma01`; `readA0` reused.
- Expand to $M=1$ (cyan): new `readA1` + 2 MMAs; both B-reads reused.
- Continue to $M=2$ (green): same pattern — 1 new A-read, 2 new MMAs; B-reads reused again.
- Finish with $M=3$ (orange): **8 MMAs** from **6 reads** (naively 16). Same mechanism unrolls K.

AMD

Compiler optimizations

Now we move from how Wave *represents* a kernel to what the compiler *does* with it. Symbolic shapes and strides make a wide range of optimizations mechanical — three of the most impactful:

- **Load coarsening.** Detect consecutive reads ($O_2 = O_1 + D_1$) and fuse them into wider vector loads, restaged through LDS. Specialized **gather-to-shared** and **swizzled** variants on CDNA3/4.
- **Software pipelining.** Issue reads for iteration $i+k$ while MMA and writes work on iteration i, hiding HBM latency behind compute.
- **Workgroup reordering.** Remap physical workgroup IDs for better L2 locality — purely a re-indexing pass.

Full details, ablations, and additional passes in the paper.

AMD

Evaluation

AMD

Evaluation setup

Hardware (all ROCm 7.0)

- AMD Instinct **MI300X** (CDNA3, 64 threads/wave)
- AMD Instinct **MI325X** (CDNA3)
- AMD Radeon **RX9070XT** (RDNA4, 32 threads/wave)

Baselines

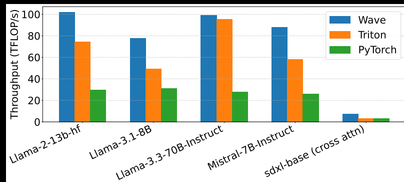
- PyTorch 2.8.0 (eager & compile)
- Triton 3.4.0 (via AITER `gemm_a16w16`)
- TileLang `0.1.6`
- TVM 0.19 with Ansor and MetaSchedule

Workloads

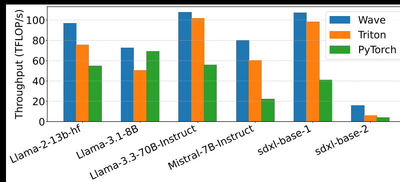
- **Attention**: Llama-2-13B, Llama-3.1-8B, Llama-3.3-70B, Mistral-7B, SDXL cross-attention.
- **GEMM**: 573 shapes from Llama 2/3, Mistral, and GPT-OSS-20B, plus an 8192^3 square.

AMD

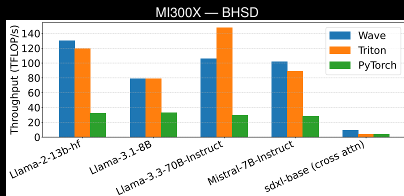
Attention throughput — MI300X & MI325X



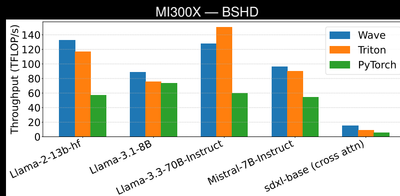
MI300X — BHSD



MI300X — BSHD



MI325X — BHSD



MI325X — BSHD

MI300X

- **2.9**× PyTorch
- **2.5**× Triton
- HM: 2.0× / 1.4×

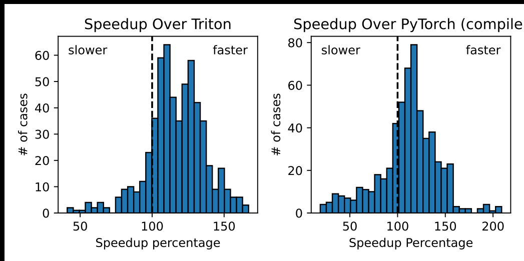
MI325X

- **2.7**× PyTorch
- **1.7**× Triton
- HM: 1.9× / 1.1×

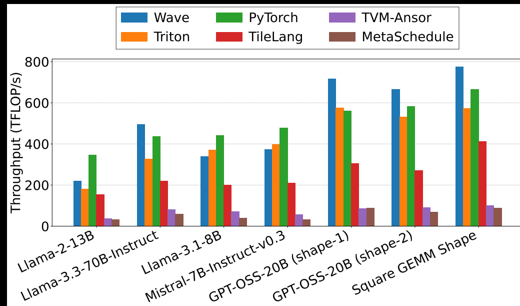
Wave trades occupancy for **34 KB more LDS** ⇒ higher throughput.

AMD

GEMM results — distribution and selected shapes



573 LLM-relevant shapes



Selected MI300X shapes

- Wave is predominantly faster than Triton, and faster than PyTorch (compile) in $\sim$**60%** of LLM-relevant cases.
- Selected shapes: **20–30%** faster than Triton/PyTorch on large models; about 2$\times$ TileLang.
- Portability: same kernel source retargets to Radeon RX9070XT with two parameter changes and one tuning round.

AMD

Recap

Wave in one slide

The problem

Writing fast GPU kernels for AMD matrix cores couples algorithm with hardware: the author has to hand-write per-thread offsets, distribution, and layout for every MMA variant.

What Wave does

A Python DSL where the author writes only the computation (no offsets, no thread IDs). Distribution is a separate constraint list. The compiler synthesizes per-thread address math via index sequences and keeps everything symbolic until late.

Highlights

- Attention up to **2.9**$\times$ PyTorch, **2.5**$\times$ Triton (MI300X / MI325X).
- GEMM: beats Triton/PyTorch on $\sim$**60%** of 573 LLM shapes.
- One source $\rightarrow$ MI300X / MI325X / RX9070XT with two parameter changes.

AMD

Thank you!

Questions?

`github.com/iree-org/wave`

Open-source project — contributions welcome!

Integrated as an attention backend in SGLang.

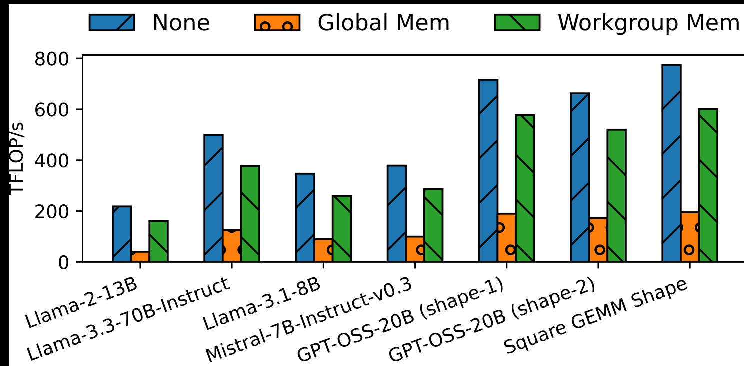
harsh.menon@amd.com

AMD

Backup

AMD

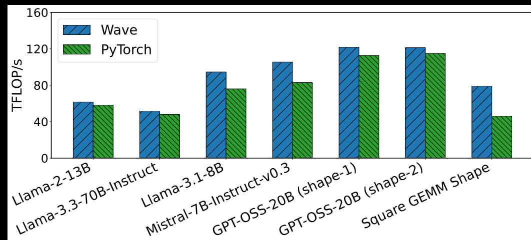
Backup — Ablation: what each optimization buys



- Disable **global memory opts** ⇒ **2.7–4.4× slowdown** (harmonic mean **3.0**×).
- Disable **schedule reordering** ⇒ **24–35% slowdown** (harmonic mean **30**%).
- Smaller kernels hurt more: less parallelism to hide latency.

AMD

Backup — Portable performance (RX9070XT)



GEMM throughput on RX9070XT

Portability result

- `threads_per_wave=32` (was 64)
- `mma_type = RDNA4_F32_16x16x16_F16`

Plus one round of tuning. *No changes to the kernel source.*

- GEMM: Wave is **1.04–1.47**× PyTorch (harmonic mean **1.14**×).
- Attention on RX9070XT: up to 19.3× PyTorch (BHSD) and 9.3× PyTorch (BSHD).
- Why attention improves: fewer barriers (**6 vs 9**) and `ds.bpermute` in-register reductions.

AMD