



CONFERENCE

—NORTH AMERICA 2026—

**Composable Tensor Programming Across
Compiler Stages: PyTorch, TCL and vISA**

Sol Kim, FuriosaAI

RNGD ships today with a full software stack

Furiosa RNGD PCIe Card

Energy-efficient inference



512 TFLOPS

64 TFLOPS (FP8)
× 8 processing elements

48 GB

HBM3 capacity

256 MB SRAM

384 TB/s
on-chip bandwidth

1.5 TB/s

Memory bandwidth

2 × HBM3

CoWoS-S

180 W TDP

For air-cooled data
centers

Partners



Furiosa NXT RNGD Server

4 PFLOPS, 384 GB HBM3, 3 kW



4 PFLOPS

512 TFLOPS × 8 cards

384 GB

HBM3 capacity

2,048 MB SRAM

256 MB × 8 cards

12 TB/s

Memory bandwidth

16 × HBM3

CoWoS-S

3 kW

Power consumption

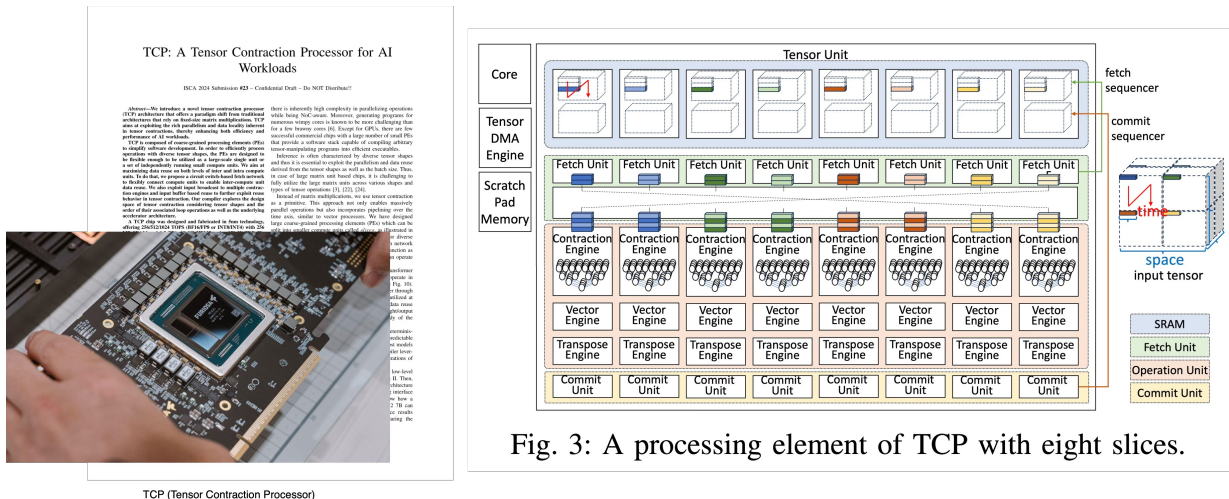
Partners



CONFERENCE

—NORTH AMERICA 2026—

One tensor-contraction model spans RNGD to PyTorch



Contraction is the hardware primitive, so the tensor programming model stacks on top of it

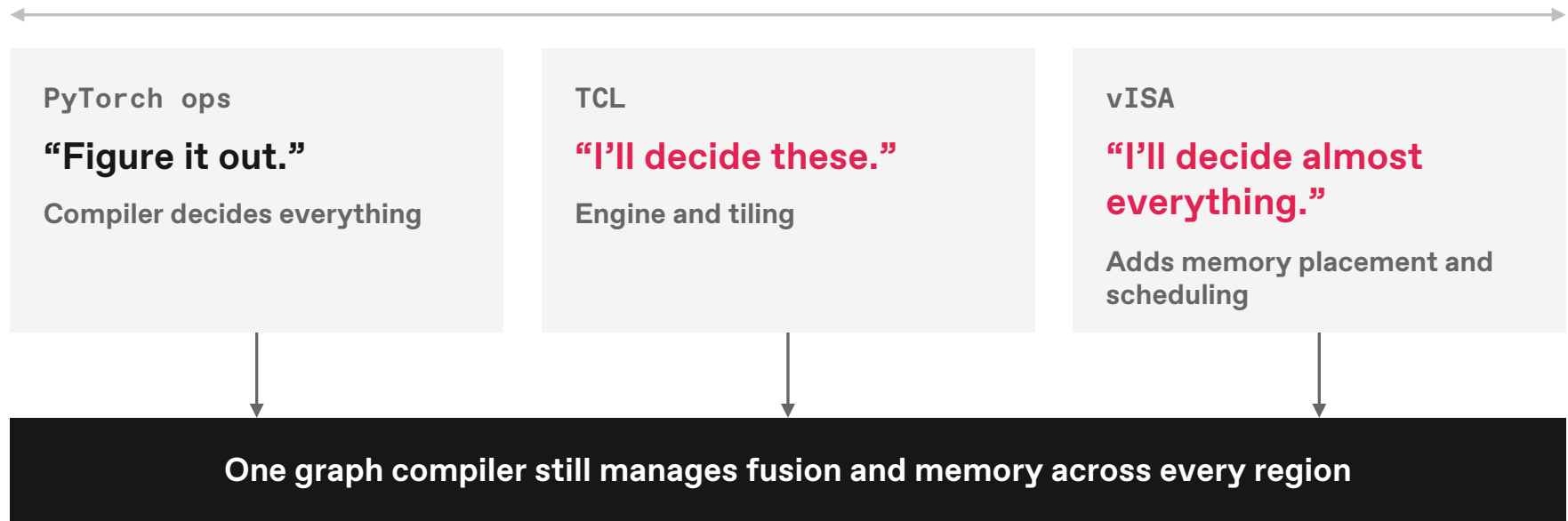
RNGD → vISA → TCL → PyTorch

Source: Kim et al., "TCP: A Tensor Contraction Processor for AI Workloads," ISCA 2024.

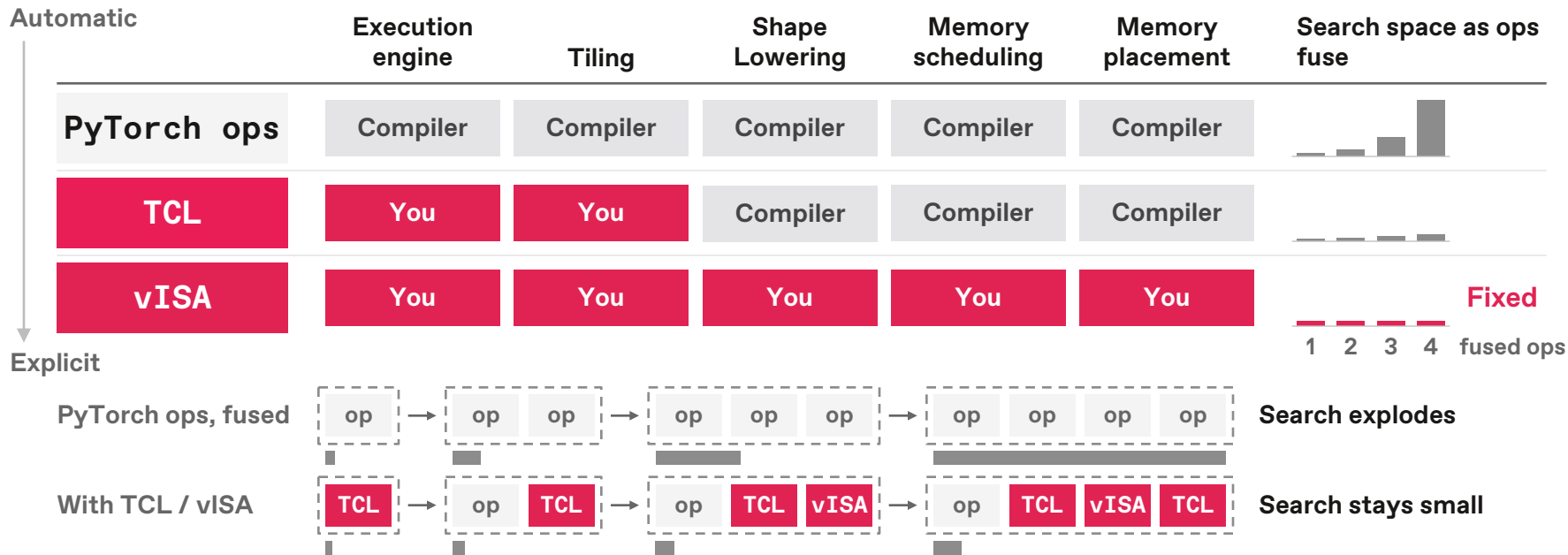
Only bottlenecks need human or AI tuning

Automatic

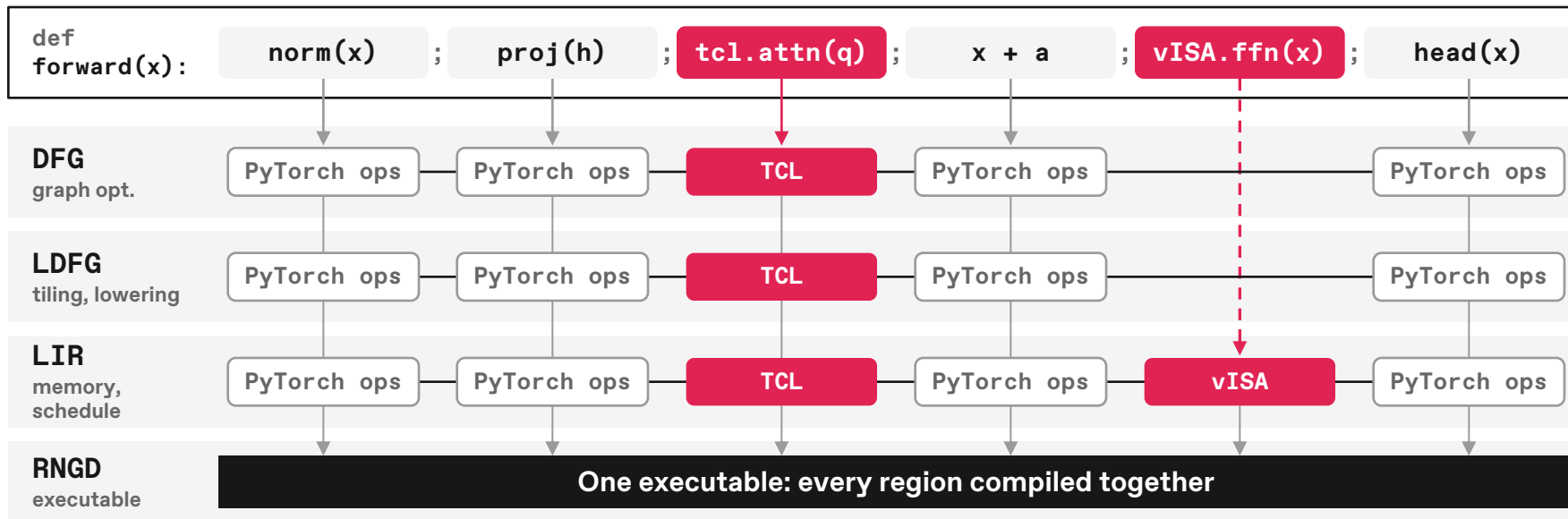
Explicit



More explicit levels trade compiler search for programmer control



Each region joins at the stage that preserves its intent



One program, three levels: each call joins the graph at its own stage, then compiles as one

One matvec, three levels of control

$y[H] = W[H, X] \cdot x[X]$ · $X = 800$ (input), $H = 256$ (output), bf16

PYTORCH · PYTHON

Compiler decides

```
import furiosa.torch as ftorch

@ftorch.compile
def matvec(x, w):
    return w @ x

npu = torch.device("furiosa:0")
y = matvec.to(npu) (
    x.to(npu), w.to(npu))
```

Write the intent

TCL · PYTHON

Choose engine and tensor axes

```
@tcl.tensor_operation # DPE
def einsum_dpe(lhs, rhs, axes):
    return tcl.operation.contract(
        lhs, rhs, to=axes)

@tcl.kernel
def matvec(x: bf16[X],
           w: bf16[H, X]) -> bf16[H]:
    y: bf16[H] = einsum_dpe(w, x, [H])
    return y
```

Structure the kernel

vISA · RUST

Place tensors and schedule instructions

```
axes![X = 800, H = 256];

#[device(chip = 1, pe = 4)]
pub fn matvec(ctx, input, weight) {
    // Placement: HBM → DM / TRF
    let w_dm: DmTensor<..., m![H], m![X]> =
        weight.to_dm(&mut ctx.tdma);
    let x_trf = input.to_trf(...); // reused

    // Schedule: fetch → collect → contract
    let out = ctx.main.begin(w_dm.view())
        .fetch().collect()
        .contract_outer(&x_trf)
        .commit();

    out.to_hbm(&mut ctx.tdma)
}
```

Control data movement + execution

Move down the stack only when the extra control is worth the code.

One model can mix PyTorch, TCL, and vISA

Decorators select the level per function; compile builds the module together

```
@kernel(tcl=matmul_tcl)
def matmul(x: torch.Tensor, w: torch.Tensor) -> torch.Tensor: ...

@kernel(visa_source=VISA_MATVEC) # W[256, 800] @ x[800], shapes fixed in Rust
def matvec(input: torch.Tensor, weight: torch.Tensor) -> torch.Tensor: ...

@compile(global_scheduling=True) # ATen + TCL scheduled with the parent
def head(h, w):
    return matmul(torch.relu(h).reshape(1, -1), w) * 2 + 1

@compile
class Model(torch.nn.Module):
    def __init__(self, w1, w2):
        ...
        self.matvec = matvec
        self.head = head
    def forward(self, x):
        h = self.matvec(x, self.w1)
        return self.head(h, self.w2)
```

TCL kernel

`@kernel(tcl=...)`

Makes a TCL kernel a torch op

vISA kernel

`@kernel(visa_source=...)`

Wraps a fixed-shape vISA kernel

Mixed region

`@compile(global_scheduling=True)`

ATen + TCL scheduled with the parent

Whole model

`@compile`

Compiles every level into one binary

Pick the level per function; the compiler still sees one model.

TCL on the DPE: one contract call is a matmul

$C[M, N] = A[M, K] \cdot B[K, N]$ · Dot Product Engine (DPE), f32

```
@tcl.tensor_operation
def matmul_dpe_op(a, b, to):
    # DPE: one Contraction Engine pass. K is shared and absent from
    # `to`, so it's reduced away; M and N pass through untouched.
    t = operation.contract(a, b, to=to)
    return commit.typecast(t, tcl.f32)

@tcl.kernel
def matmul_dpe(
    a: tcl.Tensor[tcl.f32, [M, K]],
    b: tcl.Tensor[tcl.f32, [K, N]],
) -> tcl.Tensor[tcl.f32, [M, N]]:
    return matmul_dpe_op(a, b, [M, N])
```

Tensor operation

@tcl.tensor_operation

Defines the op one engine runs

Contraction

operation.contract(a, b, to=...)

K is not in the output, so it is summed

Commit

commit.typecast(t, tcl.f32)

Writes the result back as f32

Kernel

@tcl.kernel

Typed shapes: $[M, K] \times [K, N] \rightarrow [M, N]$

Name the output axes; the shared axis K is reduced in a single DPE pass.

TCL on the Vector Engine: multiply, then reduce

Same matmul and kernel signature · Vector Engine (VE), f32

```
@tcl.tensor_operation
def matmul_ve_op(a, b, reduce_axes):
    # VE, step 1: broadcast-multiply. a:[M,K] has no N axis, b:[K,N]
    # has no M axis -- both broadcast at once into product:[M,K,N].
    product = operation.mulf(a, b)
    # VE, step 2: reduce K only, leaving M and N standing -> [M,N].
    total = operation.reduce(product, axes=reduce_axes, op=tcl.Addf)
    return commit.typecast(total, tcl.f32)

@tcl.kernel
def matmul_ve(
    a: tcl.Tensor[tcl.f32, [M, K]],
    b: tcl.Tensor[tcl.f32, [K, N]],
) -> tcl.Tensor[tcl.f32, [M, N]]:
    return matmul_ve_op(a, b, [K])
```

Broadcast multiply

```
operation.mulf(a, b)
```

$[M, K] \times [K, N] \rightarrow \text{product } [M, K, N]$

Reduce

```
operation.reduce(..., op=tcl.Addf)
```

Sums K away, leaving $[M, N]$

Commit

```
commit.typecast(total, tcl.f32)
```

Writes the result back as f32

Kernel call

```
matmul_ve_op(a, b, [K])
```

Passes the axis to reduce, not the output

Same result as the DPE version, built from two explicit Vector Engine steps.

vlSA places every tensor and stage by hand

$C[I, J] = A[I, K] \cdot B[K, J]$ · $I = J = 256, K = 64, \text{bf16}, 1 \text{ chip}, 1 \text{ PE}$

```
axes![I = 256, J = 256, K = 64];
type Chip = m![1]; type Cluster = m![1 # 1]; type Lane = m![J % 8];
type Slice = m![I / 32, J / 32]; // 8 × 8 = 64 slices, one output tile each
#[device(chip = 1, pe = 1)]
pub fn matmul(
    ctx: &mut Context,
    a: &DmTensor<bf16, Chip, Cluster, Slice, m![I % 32, K]>,
    b: &TrfTensor<bf16, Chip, Cluster, Slice, Lane, m![J / 8 % 4, K]>,
) -> HbmTensor<bf16, Chip, m![I, J]> {
    let result: DmTensor<bf16, Chip, Cluster, Slice, m![I % 32, J % 32]> =
        ctx.main.begin(a.view())
        .fetch::<m![I % 32, J / 8 % 4], m![K]>()
        .collect::<m![I % 32, J / 8 % 4, K / 16], m![K % 16]>()
        .contract_outer::<m![I % 32, J / 8 % 4, K / 32], m![K % 32], _, _, _>(b)
        .contract_packet::<m![1]>()
        .contract_time::<m![I % 32, J / 8 % 4]>()
        .contract_lane::<m![I % 32, J / 8 % 4], m![J % 8]>(LaneMode::Interleaved)
        .cast::<bf16, m![J % 8 # 16]>()
        .commit_trim::<m![J % 8]>()
        .commit();
    result.to_hbm(&mut ctx.tdma) // DM → HBM
}
```

Contraction

Hardware mapping

`type Slice = m![I/32, J/32]`

64 slices, one 32 × 32 output tile each

Memory placement

`&DmTensor · &TrfTensor`

A in DM, B in TRF, result to HBM

Fetch

`.fetch::<..., m![K]>()`

Streams A tiles from DM, K innermost

Write back

`commit() · to_hbm()`

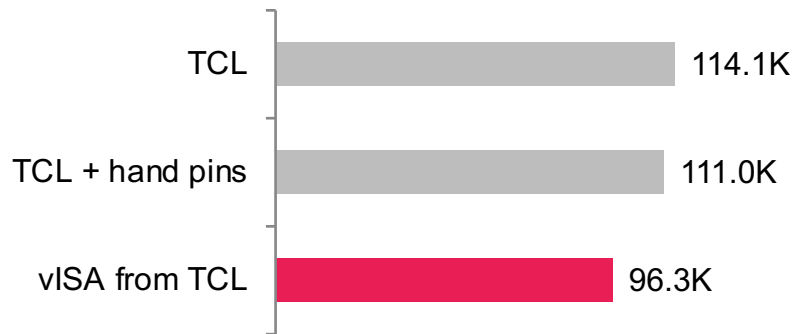
Commit to DM, then DMA to HBM

Tiling, memory placement, and instruction order all live in the code, not the compiler.

vISA pins memory to cut the last 7–16% of cycles

Gemma4-12B decode FFN

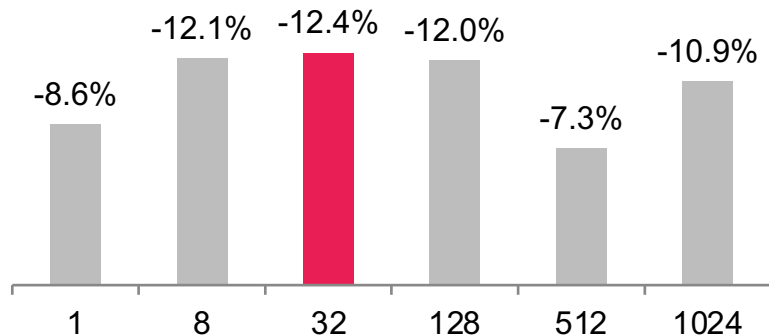
T=1, 8 PE



Weight DMAs drop 15 → 6, removing ~9K DRAM-bridge cycles.

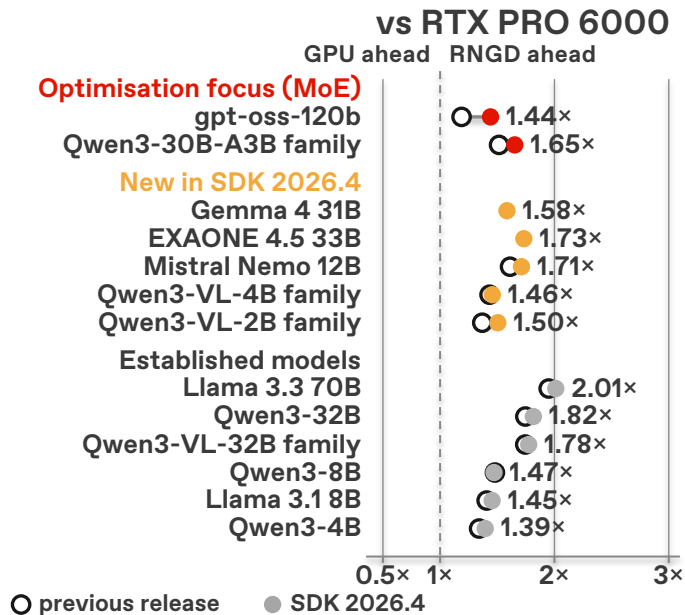
Qwen3.6-27B MLP vs. TCL

Tokenwise, 8 PE × 4 chips

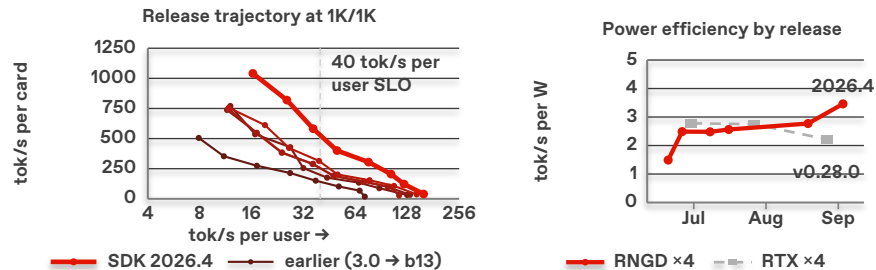


All 11 buckets improve (6 shown); geomean is 1.12×. At ≥512 tokens, vISA handles up/gate; TCL handles down.

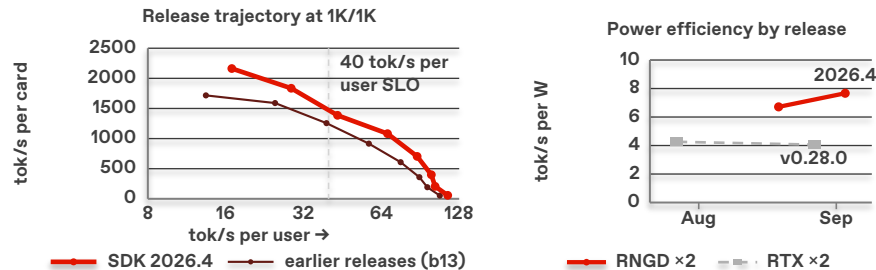
Right kernel per bottleneck: 1.39–2.01×



gpt-oss-120b 1.44× vs RTX · power efficiency ×2.38 in 75 days

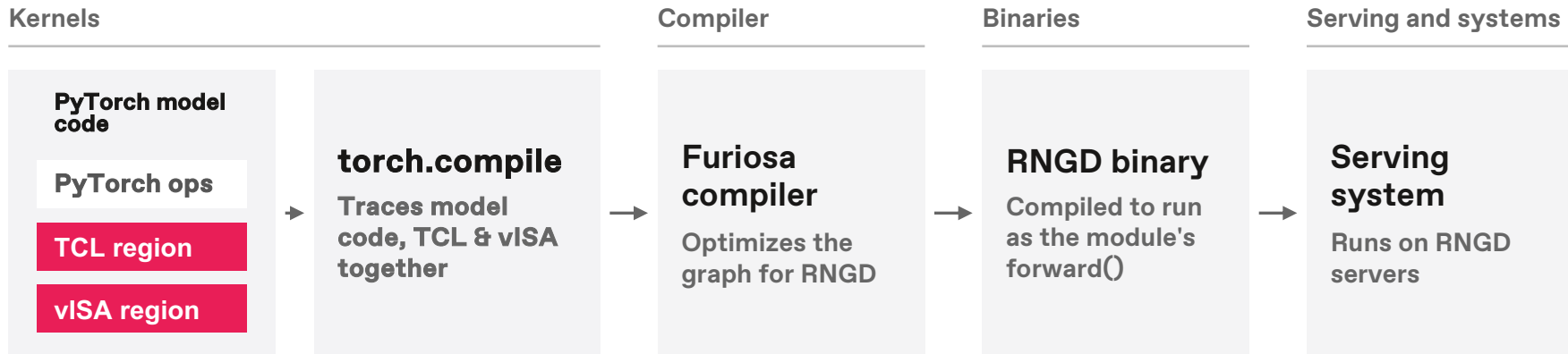


Mistral-NeMo-12B 1.71× vs RTX from first enablement



SDK 2026.4 (furiosa-llm 2026.4.0b15) vs vLLM 0.28.0 · power efficiency = output tok/s ÷ accelerator W, mean over scenarios run on both systems · 1K/1K

The compiler composes every region into one RNGD binary



One PyTorch model can mix regular model code with TCL and vISA regions.

torch.compile captures the complete model consistently; the Furiosa compiler optimizes and compiles every region together.

TCL and furiosa-torch open in Q4 2026; vISA is public now

Planned public release

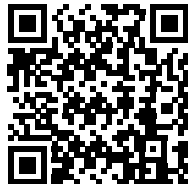
Q4 2026

furiosa-torch and TCL are planned for public release in the fourth quarter of 2026.

Start with the vISA docs

Already public today

A practical entry point to understand RNGD's programming model.



developer.furiosa.ai/furiosa-opt/book/

Scan the QR code to explore the programming model before the full software stack is released.

See TCL in action at Furiosa Booth 2, to the left of the main entrance.

