Teaching LLMs to Write System Kernels for AI Accelerators

Post-Training, Reasoning, and Agentic Optimization

Neuron Science · Amazon Annapurna Labs

KDD 2026 Tutorial · Sunday, Aug 9, 2026 · 1:00pm to 5:00pm

📍 Jeju International Convention Center · Room 402A

Speakers

Zhen Jia
Senior Applied Scientist

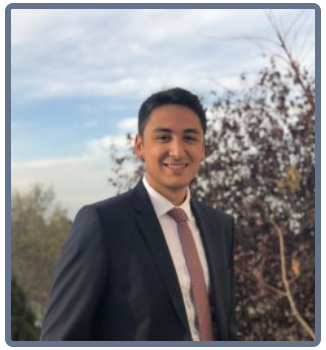
Rajarshi Saha
Applied Scientist

Jiin Woo
Applied Scientist

Youngsuk Park
Senior Applied Scientist

Neuron Science · Amazon Annapurna Labs

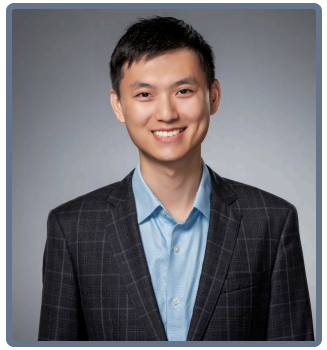
This has been a team effort



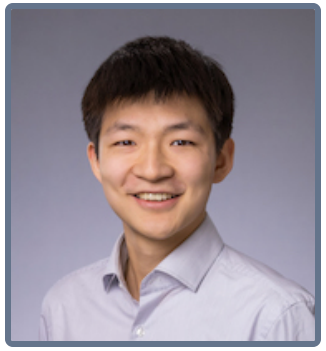
Kaan Ozkara
Applied Scientist

Shaowei Zhu
Applied Scientist

Lingfan Yu
Applied Scientist



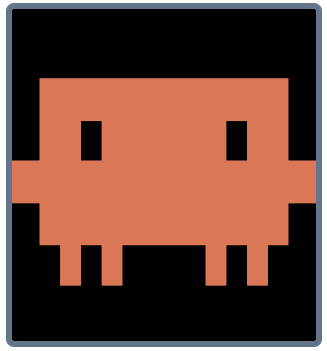
Wei Tang
Applied Scientist



Ziyang Xu
Senior Applied Scientist

Emily Webber
Principal Solutions Architect

Yida Wang
Principal Scientist



Claude Code
AI Assistant

Tutorial Website

https://neuron-science.github.io/llm_kernel_writing/

Slides, notebooks, and reading resources live here

Tutorial Roadmap

Three parts, delivered in sequence:

1. **Part 1: Hardware, Kernels, and LLM Foundations**
2. **Part 2: Inference-Time Scaling and Agentic Kernel Optimization**
3. **Part 3: Post-Training LLMs for Kernel Generation**

Part 1: Hardware, Kernels, and LLM Foundations

Teaching LLMs to Write Kernels for AI Accelerators

KDD 2026 Tutorial

The target and the tool

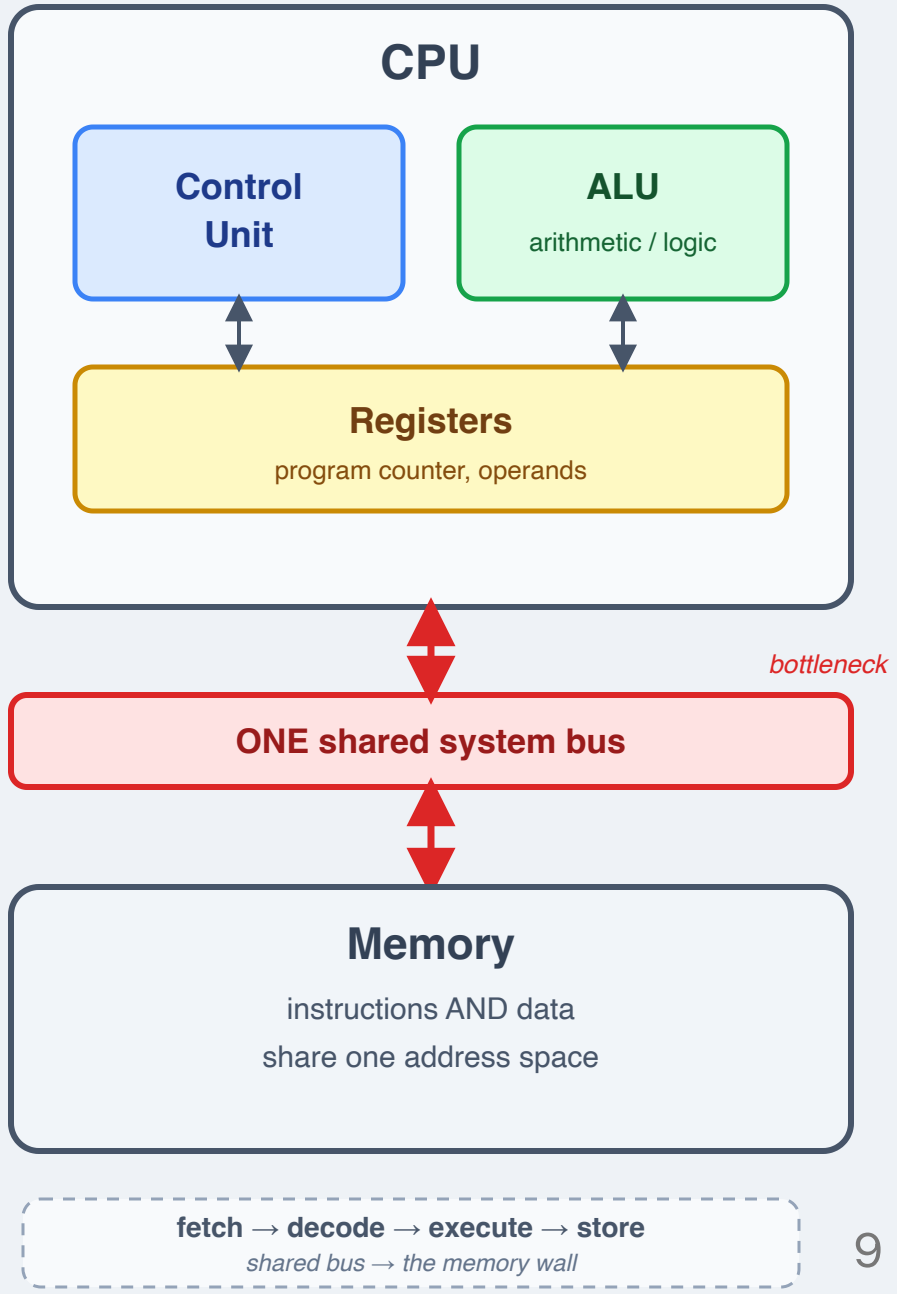
- This tutorial teaches LLMs to write system kernels for AI accelerators
- Part 1 builds the foundations in two halves, delivered FIRST
- **The TARGET:** accelerators, compiler stacks, and the kernel languages (CUDA / Triton / NKI)
- **The TOOL:** how LLMs decode, and why inference-time compute buys quality
- Part 2 then scales a FIXED model; Part 3 TRAINS a better one - both need this groundwork

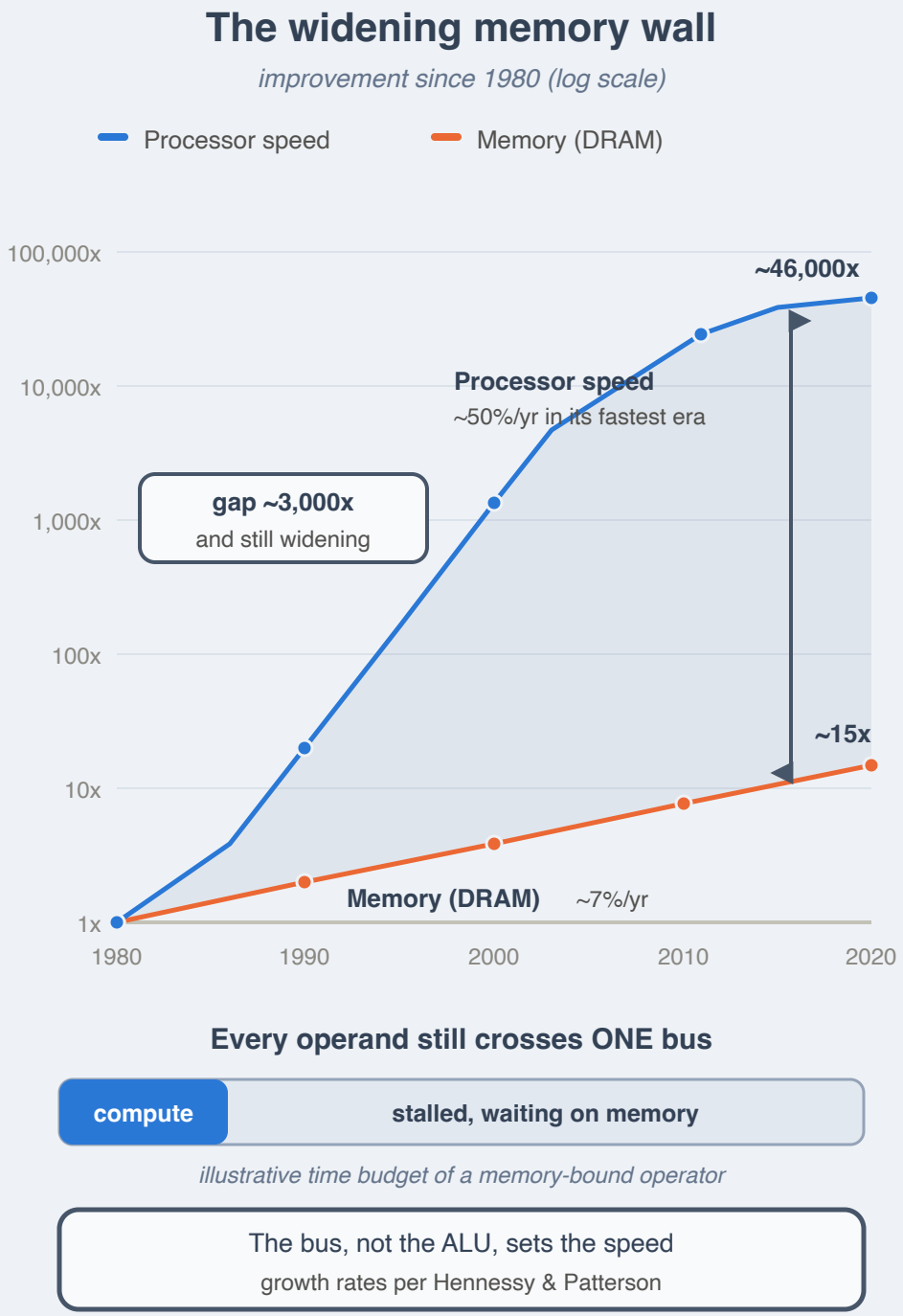
Roadmap for Part 1

- **1.1 Hardware:** the compute engines, memory hierarchies, and interconnects a kernel targets
- **1.2 Software stacks:** how a model lowers to machine code, and a little bit about the compiler
- **1.3 Kernel languages and kernels:** CUDA, Triton, NKI
- **1.4 Decoding:** how LLMs generate content (i.e., code)

The von Neumann architecture

- **Stored-program model:** instructions and data share ONE memory over ONE system bus (CPU = control unit + ALU + registers)
- **The instruction loop:** fetch -> decode -> execute -> store, driven by the program counter, billions of times per second
- **The von Neumann bottleneck:** CPU and memory share that bus, so the processor stalls waiting on data - the origin of the "memory wall"





The memory wall: the bottleneck that kept widening

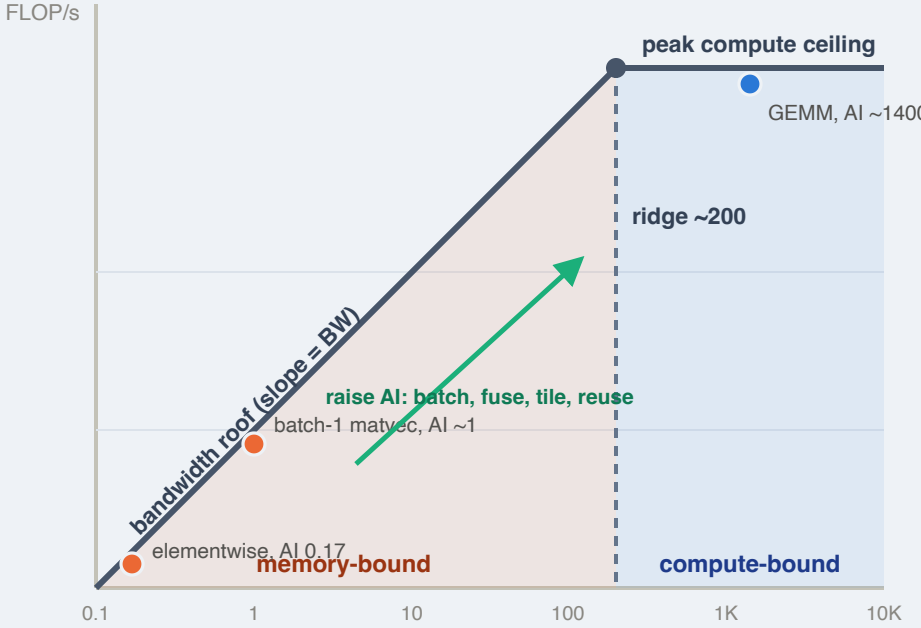
- **The bottleneck became a wall:** processor speed grew ~50%/yr in its fastest era while DRAM latency improved only ~7%/yr - the gap compounds into orders of magnitude
- **The unit that matters flipped:** a DRAM access costs hundreds of cycles, and a byte moved costs far more energy than the arithmetic done on it - so bytes moved, not FLOPs, set the cost
- **Memory-bound:** when an operator moves more bytes than it does useful math, the engines idle and a faster ALU buys nothing
- Two responses, and this tutorial needs both: **specialize the hardware** (1.1) and **write code that moves less data** (1.3)

Arithmetic intensity

the operator's number, read against the machine's roofs

AI = FLOPs performed / bytes moved

attainable FLOP/s = min(peak compute , AI × peak bandwidth)



both axes log; ridge and dot heights illustrative

Counting AI, FP16 (2 bytes/element)

operator	FLOPs	bytes moved	AI	binds on
elementwise add	n	6n (2 in, 1 out)	**0.17**	bandwidth
matvec, batch 1	2mk	2mk (weights)	**~1**	bandwidth
GEMM, n = 4096	$2n^3$	$6n^2$ (3 tiles)	**n/3 ≈ 1400**	compute

Same FLOPs, different *bytes* - AI is set by the implementation

AI is the diagnosis and the lever

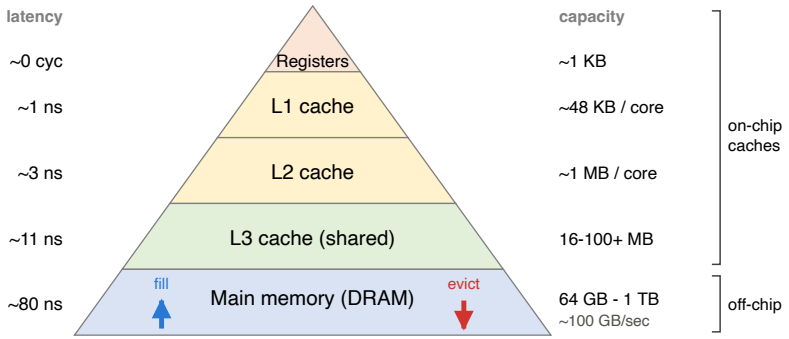
roofline: Williams et al., CACM 2009

Arithmetic Intensity and the Roofline Model

- **One number describes an operator's demand on the machine:** arithmetic intensity $\text{AI} = \text{FLOPs performed}/\text{bytes moved}$ (FLOPs/byte) - *how much math you get per byte you paid for*
- **AI belongs to the algorithm and its implementation** - so it is the *one* term in the cost model a kernel author can actually move. Batching the same matvec B ways reuses each weight byte B times: $\text{AI} \approx B$
- **The roofline is the yardstick.** Two machine constants - peak compute and peak bandwidth - give $\text{attainable FLOP/s} = \min(\text{peak compute}, \text{AI} \times \text{peak bandwidth})$
- **Compare AI to the ridge point**: below it, memory bound (engines idle); above it, compute bound
- **So AI is the diagnosis *and* the lever:** raise it by moving fewer bytes for the same math - fuse, tile, reuse on-chip. It is also the metric a hardware-aware reward must encode in Part 3 *(Williams et al., CACM 2009)*

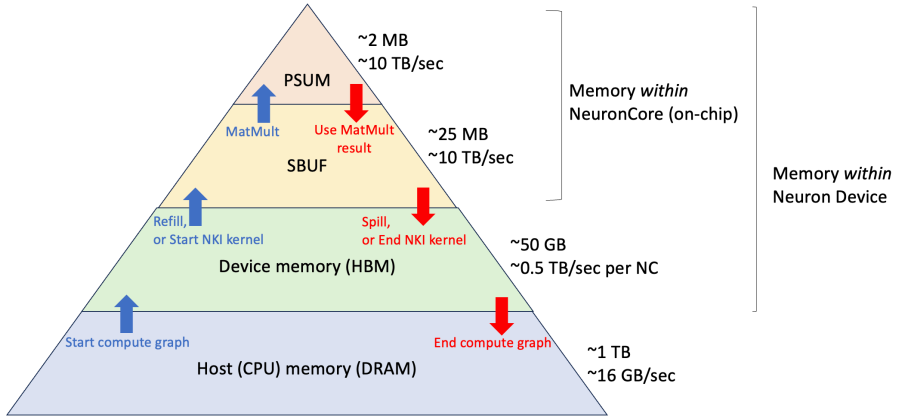
Memory Hierarchy: mitigate the impact of memory wall

- The hierarchy is a pyramid: closer to compute = higher bandwidth, lower latency, far less capacity. It generalizes across CPU, GPUs, TPUs, and Trainium
- Data movement is costly. Keeping the working set high in the pyramid is the key lever; **levers:** tiling, data reuse, fusion - the recurring vocabulary of every kernel
- **But who decides what sits where differs, and that difference is why kernels exist:** a CPU's caches are *hardware-managed* - movement is implicit, a bet placed by LRU and a prefetcher. An accelerator's SBUF / PSUM are *software-managed* - every transfer is named in the code
- Central concerns either way: minimize data movement and overlap compute with movement so engines never stall



LRU + a prefetcher move the data - no instruction says "stage this tile"

CPU - implicit: the cache decides



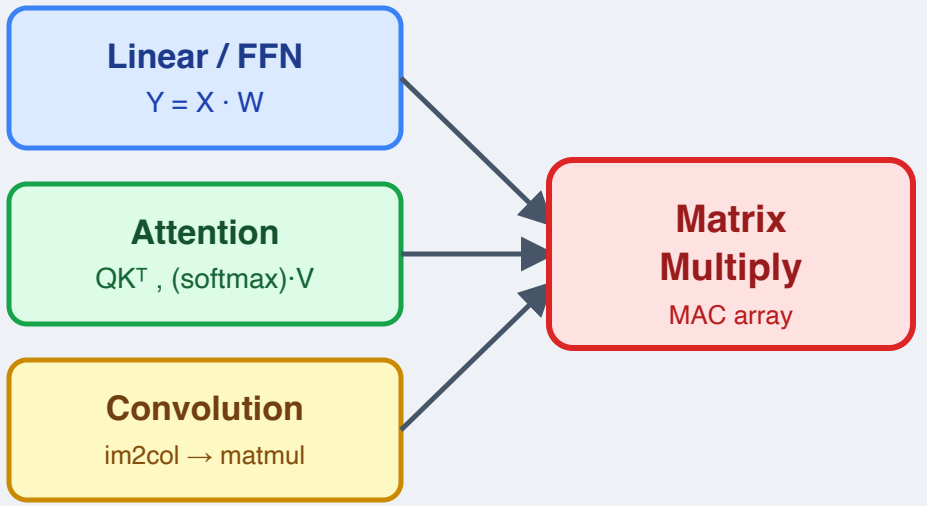
Trainium NeuronCore - explicit: the kernel decides

Why specializing pays off: a narrow, regular workload

- **The workload is narrow:** DNN work is dense linear algebra (matmul, conv) over a small operator set (activation, norm, reduction), often in reduced precision (FP16 / BF16 / FP8 / INT8)
- Nearly everything reduces to ONE primitive - a matrix multiply - and the rest is cheap glue around it
- **CPUs pay for generality:** most transistors and energy go to control, branch prediction, and caches so arbitrary code runs fast
- **Accelerators spend that budget on math instead:** wide multiply-accumulate arrays doing one thing - tensor math - efficiently per watt

Compute-dense layers

nearly all reduce to matmul

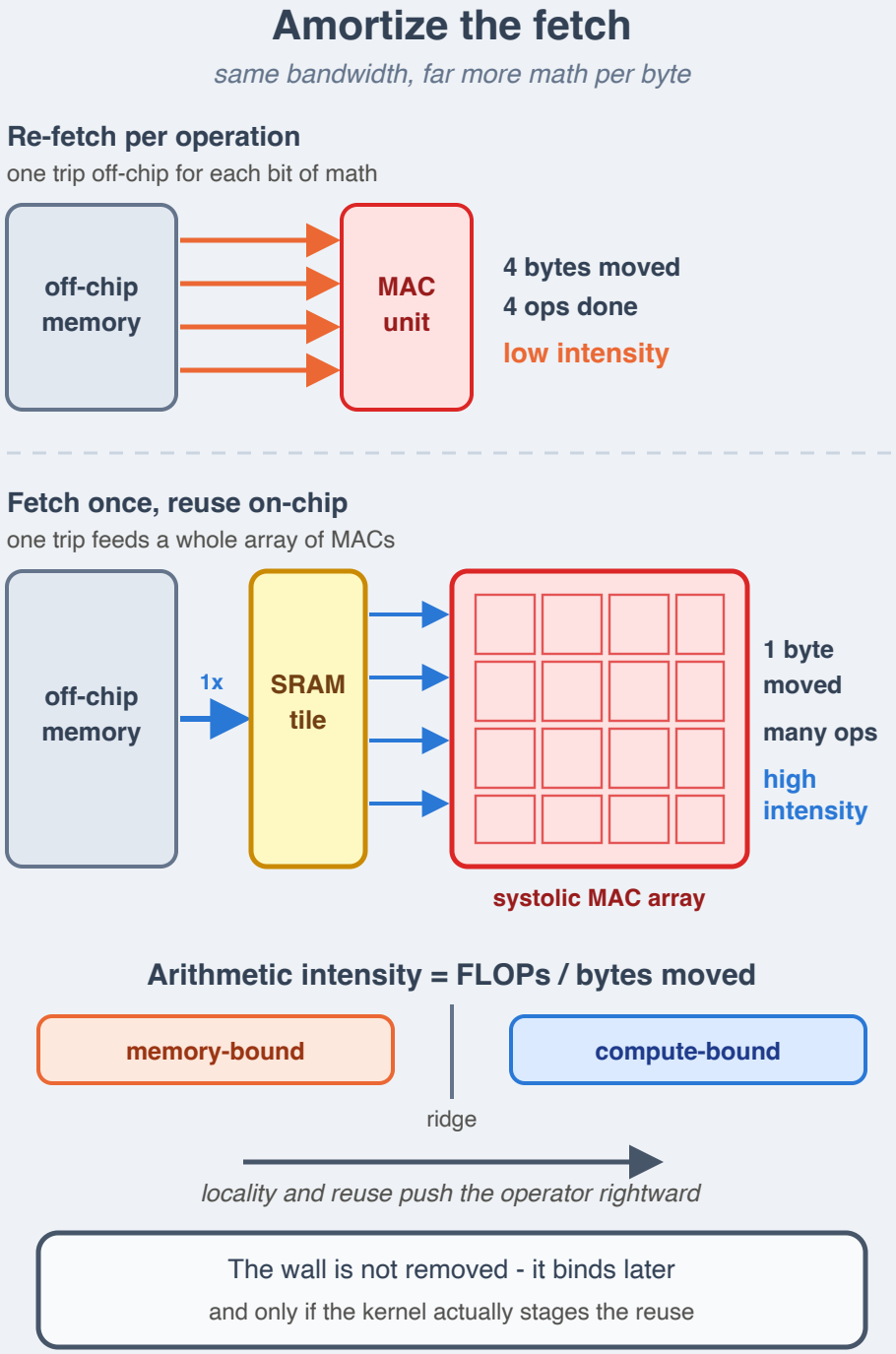


Share of model FLOPs

Elementwise activation, normalization, reductions: cheap "glue" around the matmuls

Specialization pays off

Build wide multiply-accumulate arrays; one dense primitive covers most of the work



How accelerators beat the bottleneck: locality and reuse

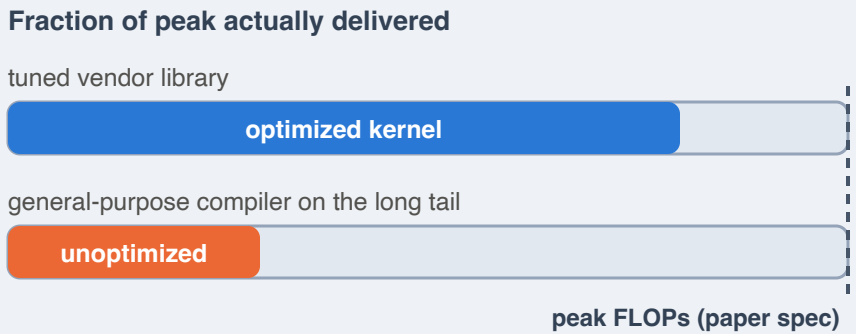
- The bottleneck is frequent movement between memory and compute - so the fix is to move data less, not to move it faster
- **Large on-chip SRAM buffers:** stage a tile once, then read it many times without leaving the chip
- **Hierarchical memory:** each level up is smaller but far faster, so the hottest data sits closest to the engines
- **Dataflow execution (systolic arrays):** an operand streams through a grid of MAC cells, consumed by many of them per fetch, instead of being re-fetched per operation
- **Massive parallelism + specialized units** (tensor cores) turn each fetched byte into far more useful math
- **Net effect - reduce the data movement** (FLOPs per byte moved): the fetch is amortized, so memory bandwidth stops being the binding constraint
- The wall is not removed, only pushed back - and only if the code actually stages the reuse

Strong hardware is not enough: hardware rich, software poor

- **"Strong hardware":** a chip superior on paper (theoretical FLOPs, perf/watt, cost) still fails without optimized code to extract its performance
- Peak FLOPs is a CEILING, not a delivery - the same silicon runs at a fraction of peak or near it depending only on the kernel
- **Why the software is hard - structural, not incidental:**
  - Both sides evolve: model family (MLP -> CNN -> Transformer -> MoE) and hardware cadence (Ampere -> Hopper -> Blackwell)
  - HW/SW co-design (quantization, sparsity) blurs algorithm vs silicon - hardware is not an opaque box
- So frameworks (PyTorch / JAX / HF) must lower per backend, and code rarely ports cleanly
- Performant kernel effort is the scarce input - the case for teaching a model to supply it

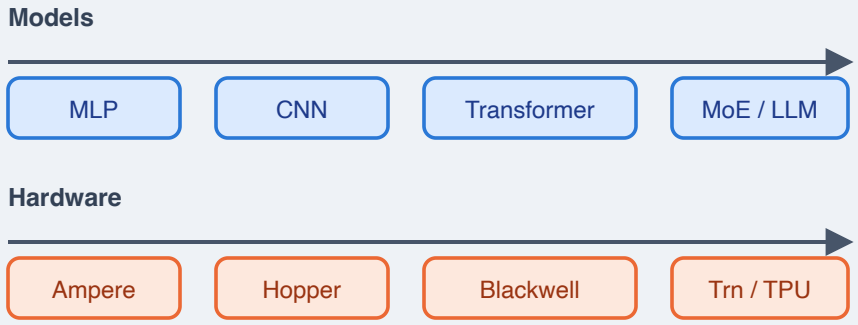
Hardware rich, software poor

peak FLOPs are a ceiling, not a delivery



Same silicon. The gap is software.
bars illustrative

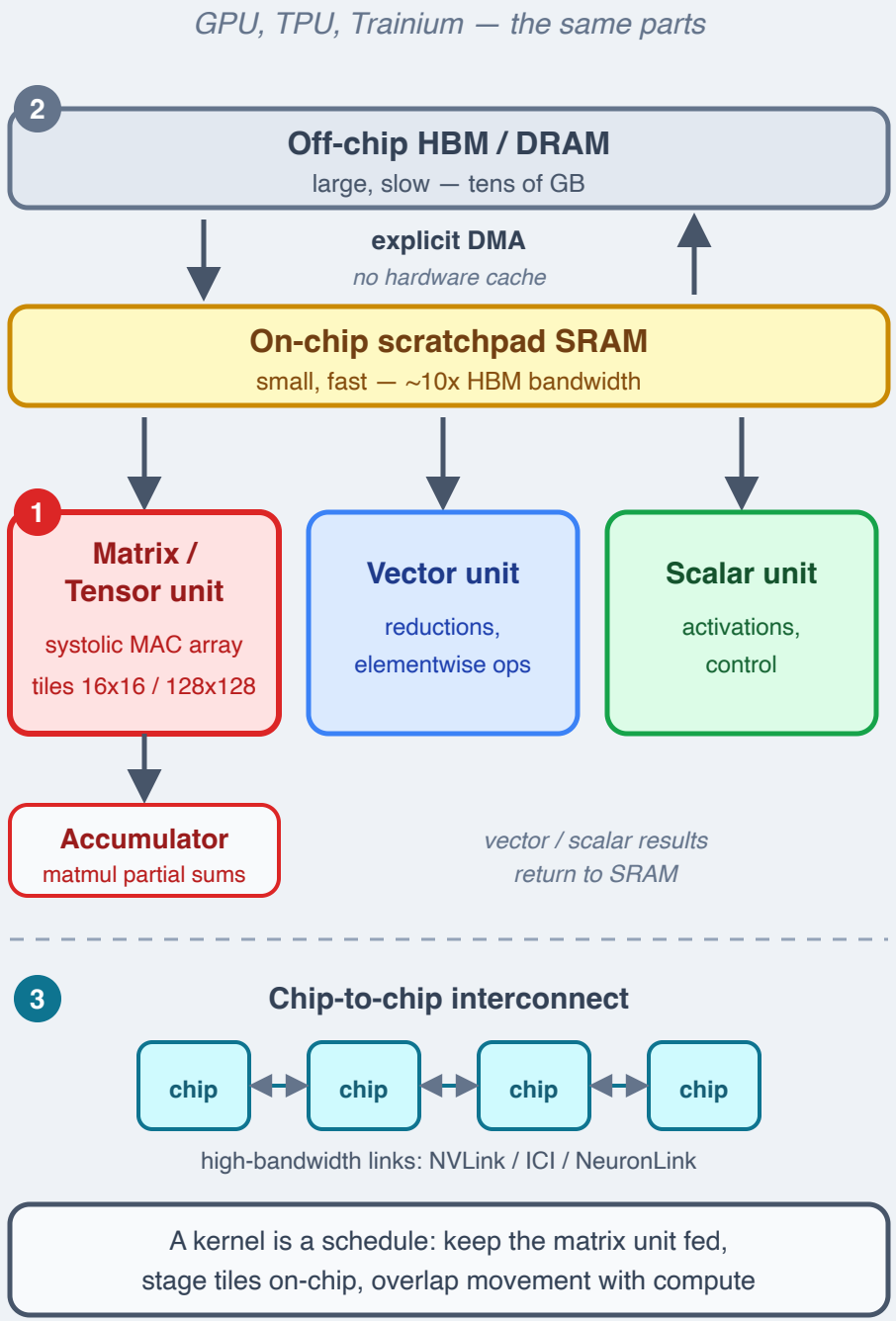
Both axes keep moving



every pair needs re-tuned kernels
and co-design keeps leaking across the line

a many-to-many problem, growing on both sides

Expert kernel effort is the scarce input
which is the case for teaching a model to write them
- the whole point of this tutorial



Accelerator components

- **1 - Compute engines:** a matrix / tensor unit (systolic array) + vector unit + scalar unit
  - Almost every kernel is a schedule that keeps the matrix unit fed while vector / scalar handle the epilogue
- **2 - Explicitly-managed memory:** slow-large HBM, fast-small on-chip SRAM, some buffers
  - Managed by the programmer / compiler, purpose is to increase the locality
- **3 - Interconnects:** many chips wired with high-bandwidth links to cooperate on one computation
- Matrix-unit tile granularities are commonly around 16x16 or 128x128
- GPU, TPU, and Trainium are all variations on these three

NVIDIA GPU: SIMT, warps, and the memory ladder

- The unit of hardware is the **Streaming Multiprocessor (SM):** CUDA cores for scalar / vector work, **Tensor Cores** for matrix multiply-accumulate
- **SIMT** (single instruction, multiple threads): you write scalar code for ONE thread; the GPU runs it across thousands, each with private registers and its own slice of data
- **The real unit of execution is the warp - 32 threads sharing one instruction stream.** Two rules follow directly:
  - **divergence:** an `if / else` inside a warp executes *both* sides under masking, serializing them
  - **coalescing:** a warp should touch 32 *contiguous* addresses, so one instruction becomes one memory transaction
- **Compute Hierarchy:** warp -> **thread block** (pinned to one SM, shares its scratchpad and barriers) -> **grid** (why one kernel scales from a small GPU to a big one)
- **Memory Hierarchy:** registers -> shared memory / L1 (software-managed, tens of KB) -> L2 -> HBM (tens of GB).
- Precisions FP32 / TF32 / BF16 / FP16 / FP8 / INT8; Ampere -> Hopper -> Blackwell shift the tuning sweet spot each generation

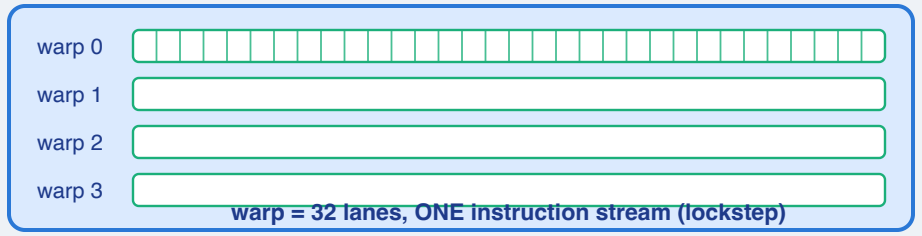
SIMT: one program, three levels

same code, different data

Grid - independent blocks cover the problem

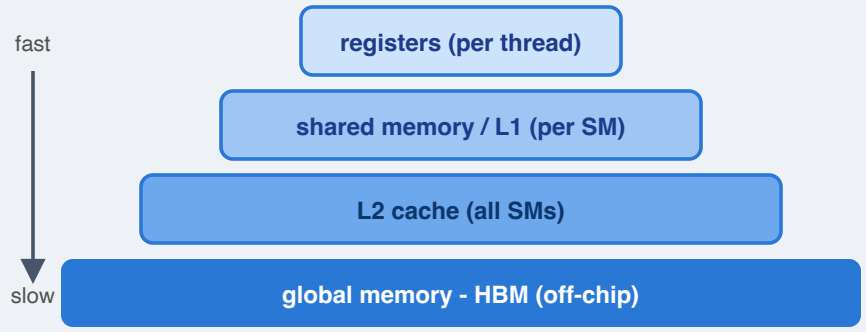
any order, any SM - scales unchanged

Block - warps pinned to ONE SM



Divergence: split an if in a warp and BOTH sides run
Coalescing: a warp should touch contiguous addresses

The same explicit hierarchy



shared memory is software-managed - the GPU's scratchpad

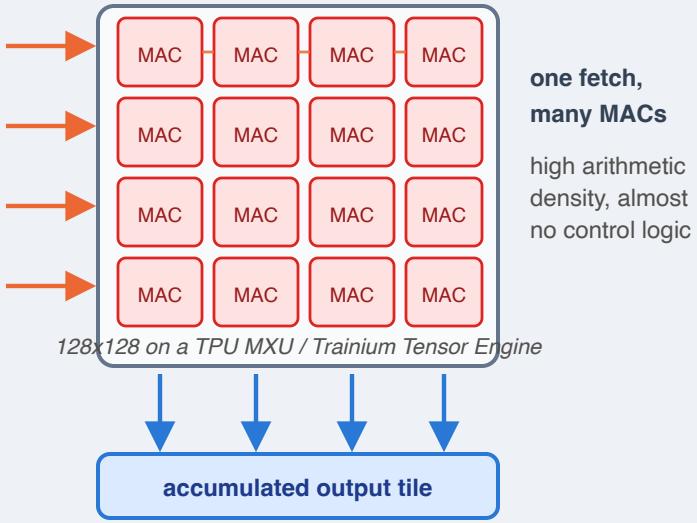
Writing a GPU kernel IS this mapping
which thread owns what, what to stage, where to sync

The systolic array (MXU)

operands flow; each is used by many cells

weights held resident in the cells

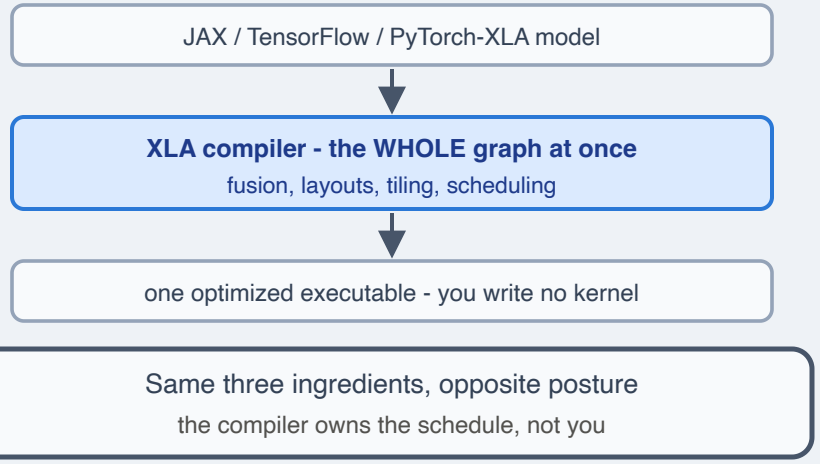
activations stream in



one fetch, many MACs

high arithmetic density, almost no control logic

Reached by whole-graph compilation

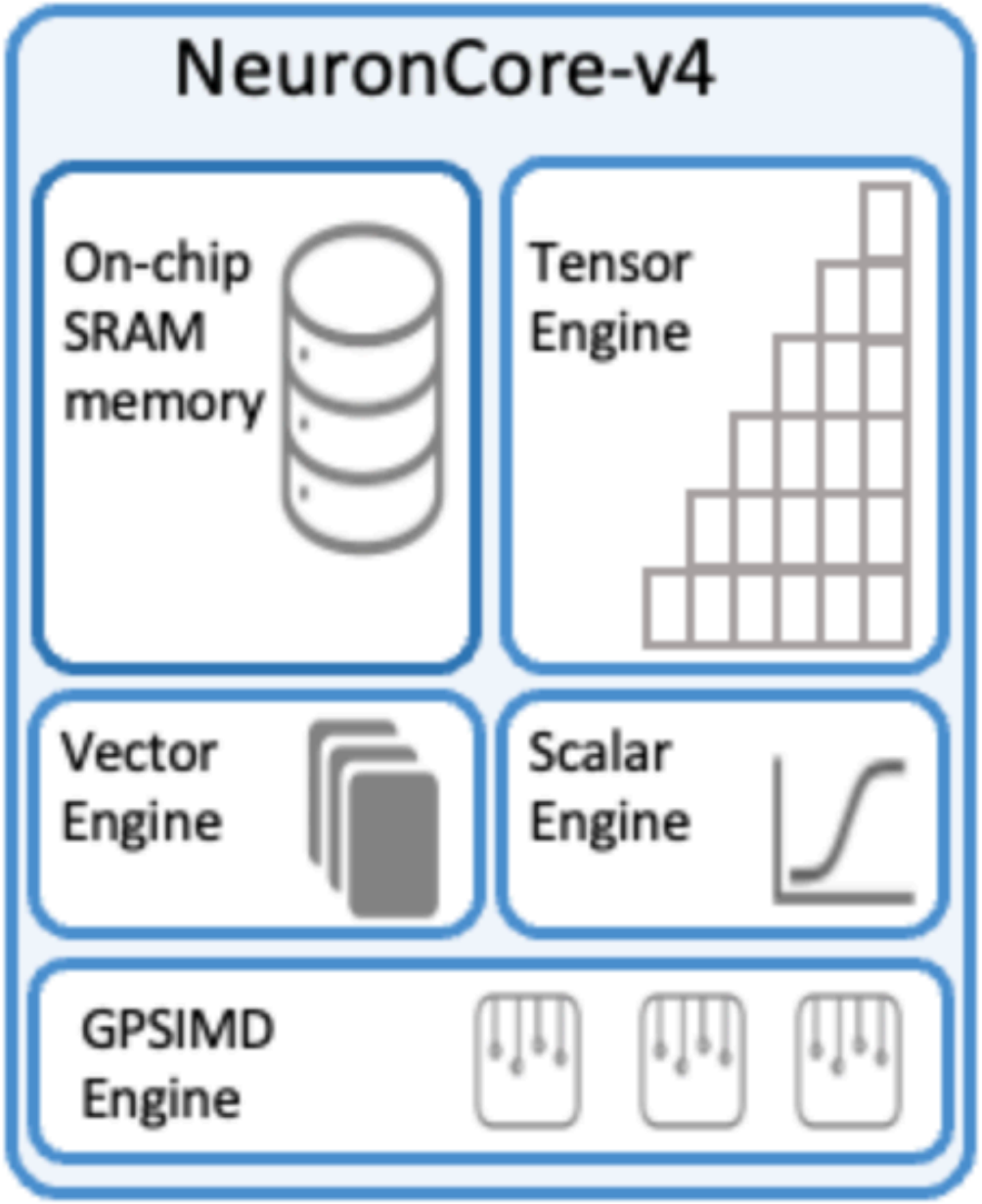


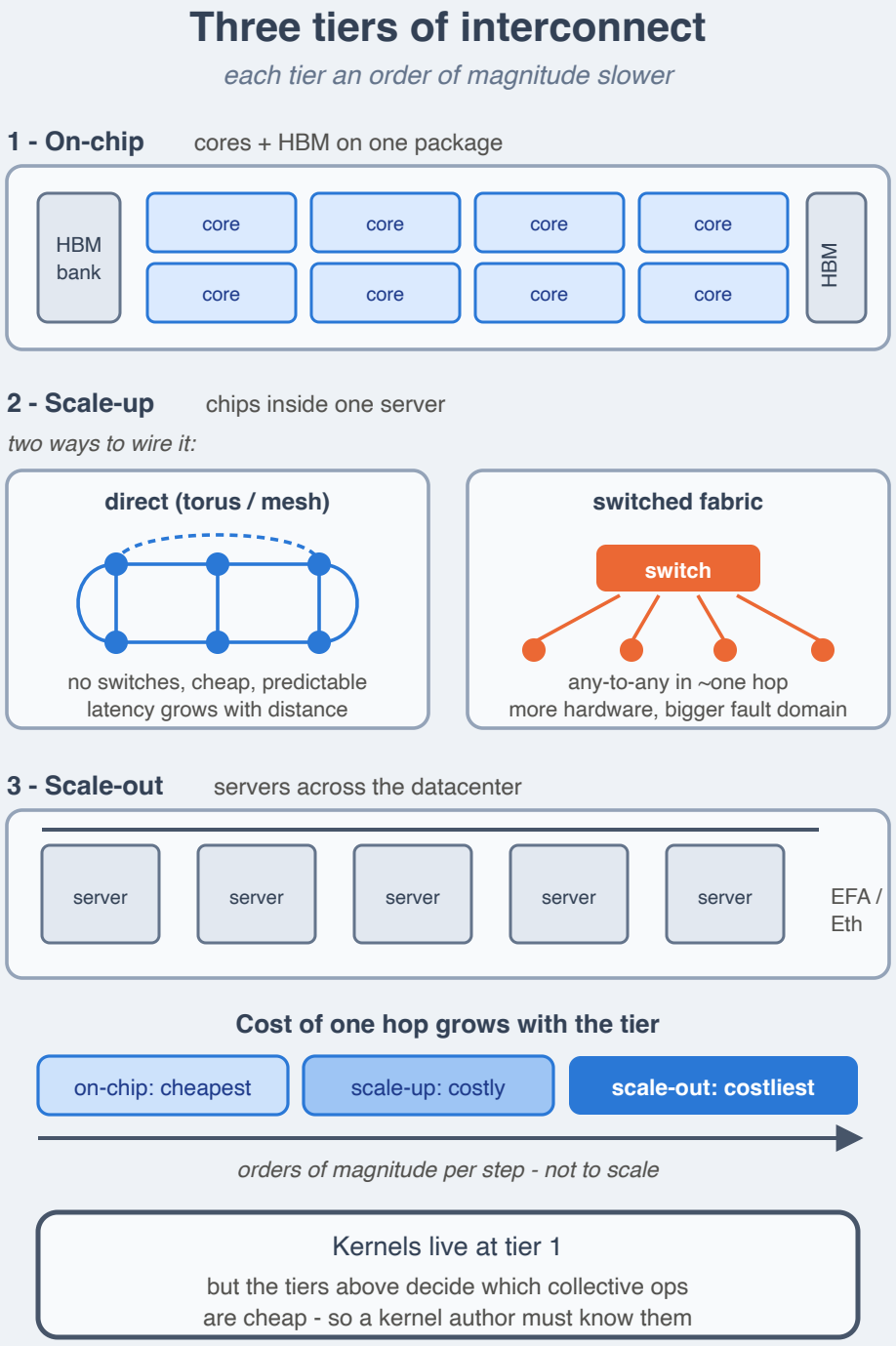
Google TPU: a systolic array

- Same three ingredients, one dominant engine: a large **systolic-array Matrix Multiply Unit (MXU)**, plus vector and scalar units, fed from an explicit **VMEM** scratchpad
- **What "systolic" buys you:** weights stay resident in the array while activations flow through, so *one fetch feeds many MACs* - arithmetic intensity is high and control overhead is near zero
- **Scale-out:** chips wired into pods over a dedicated inter-chip interconnect (ICI) in a torus; v2 -> v3 -> v4 -> v5e / v5p grow both per-chip throughput and pod size
- Pallas, the kernel language for TPU and an extension integrated into JAX
- Why this matters here: **Trainium's default path has the same shape**

AWS Trainium NeuronCore

- Trainium is AWS / Annapurna Labs' deep learning accelerator
- **Four heterogeneous engines,** and a kernel names the one it wants:
  - Tensor Engine: Handles matrix operations like matrix-multiplications and convolutions
  - Vector Engine: Processes multi-input vector operations and reductions (e.g. Normalization, ResidualAdd
  - Scalar Engine: Performs element-wise functions, including non-linearities (e.g. GELU, Square)
  - GpSimd Engine: Deeply embedded general-purpose programmable processors for custom operations
- **Two named on-chip memories: SBUF**, the software-managed scratchpad filled from HBM by DMA, and **PSUM**, a dedicated accumulator for matmul partial sums
- **Dataflow rule (matters for NKI):** the Tensor Engine reads SBUF and writes *only* PSUM; the others read and write SBUF; every HBM transfer is explicit
- **The defining constraint: SBUF and PSUM are 128-partition memories,** so you tile in units of 128 - not a tunable block size
- Precisions BF16 / FP16 / TF32 / FP32 / FP8 / MXFP8 / MXFP4





Scaling out: three tiers, and the same components

- **Interconnect is a three-tier hierarchy:** *on-chip* (inside one accelerator) / *scale-up* (a few accelerators in one server, over NVLink or NeuronLink) / *scale-out* (servers across a datacenter, over Ethernet fabrics like AWS EFA)
- **Cost rises orders of magnitude per tier,** so the tiers decide the parallelism strategy: tight inner loops stay on-chip, data-parallel replication goes to scale-out
- **The scale-up trade-off:** a **direct** topology (torus / mesh) has no switches - cheap, predictable neighbor bandwidth, but latency grows with distance; a **switched fabric** reaches any peer in ~one hop, at the cost of switch hardware and a larger fault domain
- **Synthesis:**
  - GPU = Tensor Cores, CUDA cores + registers -> shared -> L2 -> HBM + NVLink ·
  - TPU = MXU + VMEM + ICI ·
  - Trainium = 4 engines + SBUF, PSUM, HBM + NeuronLink
- A kernel lives on tier 1 - but the author (human or LLM) must know **which collective operations the tiers above make cheap**

From PyTorch to hardware

five stages, one running example: y = relu(x @ W + b)

1 High-level model

PyTorch · eager, dynamic

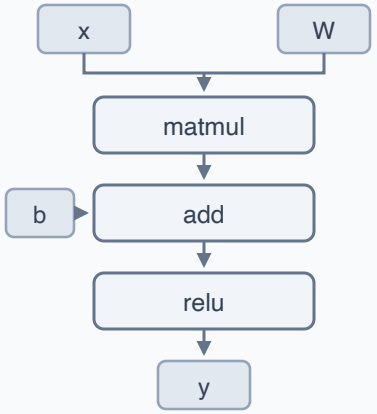
```
y = relu(x @ W + b)
```

- eager: op-by-op dispatch
- nn.Module · tensors
- autograd, dynamic control

no graph: every intermediate round-trips through HBM, one launch per op

2 Graph capture & IR

TorchDynamo · FX · TorchScript



nodes = ops
edges = tensor deps

nothing optimized yet — the win is visibility: the whole chain at once

3 Compiler lowering

Inductor IR · MLIR · vendor IR

operator fusion
matmul + add + relu
→ ONE fused op

- constant folding
- memory planning
- high-level → low-level ops

still no tiles, no engines: WHAT to compute together, not how

4 Code generation

IR → hardware kernels

GPU	CUDA · Triton
CPU	AVX vector loops
Accelerator	Neuron · TPU ISA

Scheduling decisions:
tiling · parallelism · layout

a hand-written kernel replaces THIS stage

5 Runtime execution

CUDA driver · PJRT · Neuron

- launch kernels on device
- memory allocation
- stream / execution order
- device synchronization

the schedule is already fixed — the runtime just executes it

Hardware: GPU · CPU · TPU · Trainium

stages 2, 3, 5
compilers, always

stage 4
where we intervene

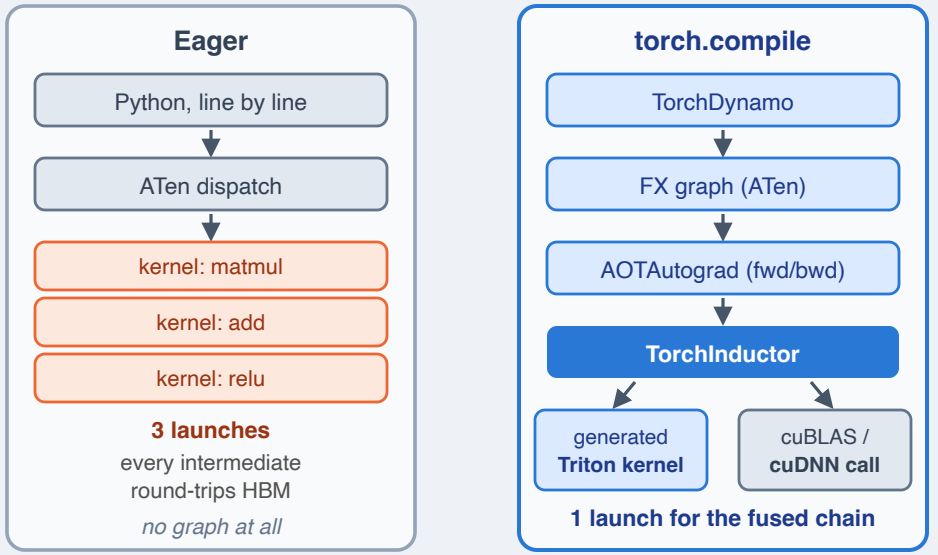
Every stack has these stages,
under other names

PyTorch: eager v.s. `torch.compile` on GPU

- **Eager:** Python runs line by line and each op dispatches immediately through **ATen** to a backend kernel. No graph at all - so every intermediate round-trips through HBM and every op is its own launch
- `torch.compile` takes a different path: **TorchDynamo** captures an **FX graph** of ATen ops -> **AOTAutograd** splits forward / backward -> **TorchInductor** lowers it
- Inductor **generates Triton kernels** for elementwise and reduction subgraphs, and **calls cuBLAS / cuDNN** for the big GEMMs - fusing pointwise chains to cut launches and keep intermediates out of HBM
- **The limit, in one example:** `y = x @ W ;  out = silu(y + b)` compiles to *an opaque cuBLAS GEMM* **+** *a separate Triton kernel* - Inductor cannot fuse across a precompiled binary
- That boundary is a real, findable **fusion barrier** - and the epilogue you would hand-write to erase it
- **This seam is where the long tail begins.** Everything after this slide is about crossing it

Two paths through PyTorch

op-at-a-time, or capture then fuse



Where Inductor stops

`y = x @ W ; out = silu(y + b)`

one operator chain a human would fuse

cuBLAS GEMM
precompiled binary
OPAQUE

separate Triton kernel
bias + activation
re-reads y from HBM

fusion barrier

a compiler cannot inject an epilogue into a black box
so the fused GEMM epilogue is written by hand

a data-dependent branch has the same effect: a graph break

This seam is where the long tail begins
and the interface a kernel author steps into

Lowering

where fast code is actually produced — and at which rung a kernel drops in

1 · Kernel languages & libraries

you can read and edit

Triton · CUDA · NKI

CUTLASS 3.x / CuTe templates

how peak GEMM + attention are built

OPAQUE binaries

cuBLAS · cuBLASLt · cuDNN

fast, shape-dispatched at runtime

UNFUSABLE — the fusion barrier

a hand-written kernel replaces a box on the LEFT, or breaks open one on the right

2 · Then it keeps lowering (GPU)

LLVM IR — portable, mature passes → **PTX** — virtual ISA, per generation → **SASS** — real machine code → **SMs** — run it

PTX is compiled FOR a target generation — sm_80, sm_90, …

these rungs are generated, not authored

Lower whole graph on TPU

JAX · TensorFlow · PyTorch/XLA · torch-neuronx

↓

lazy capture — nothing runs until a sync point

.item() · a copy to CPU · xm.mark_step()

↓

HLO / StableHLO

fuse · assign layouts · reuse buffers

↓

runtime (PJRT / Neuron) → TPU or Trainium

the fusion window is the whole graph, not one op

no kernel is written — so a hand-written one is an explicit escape hatch

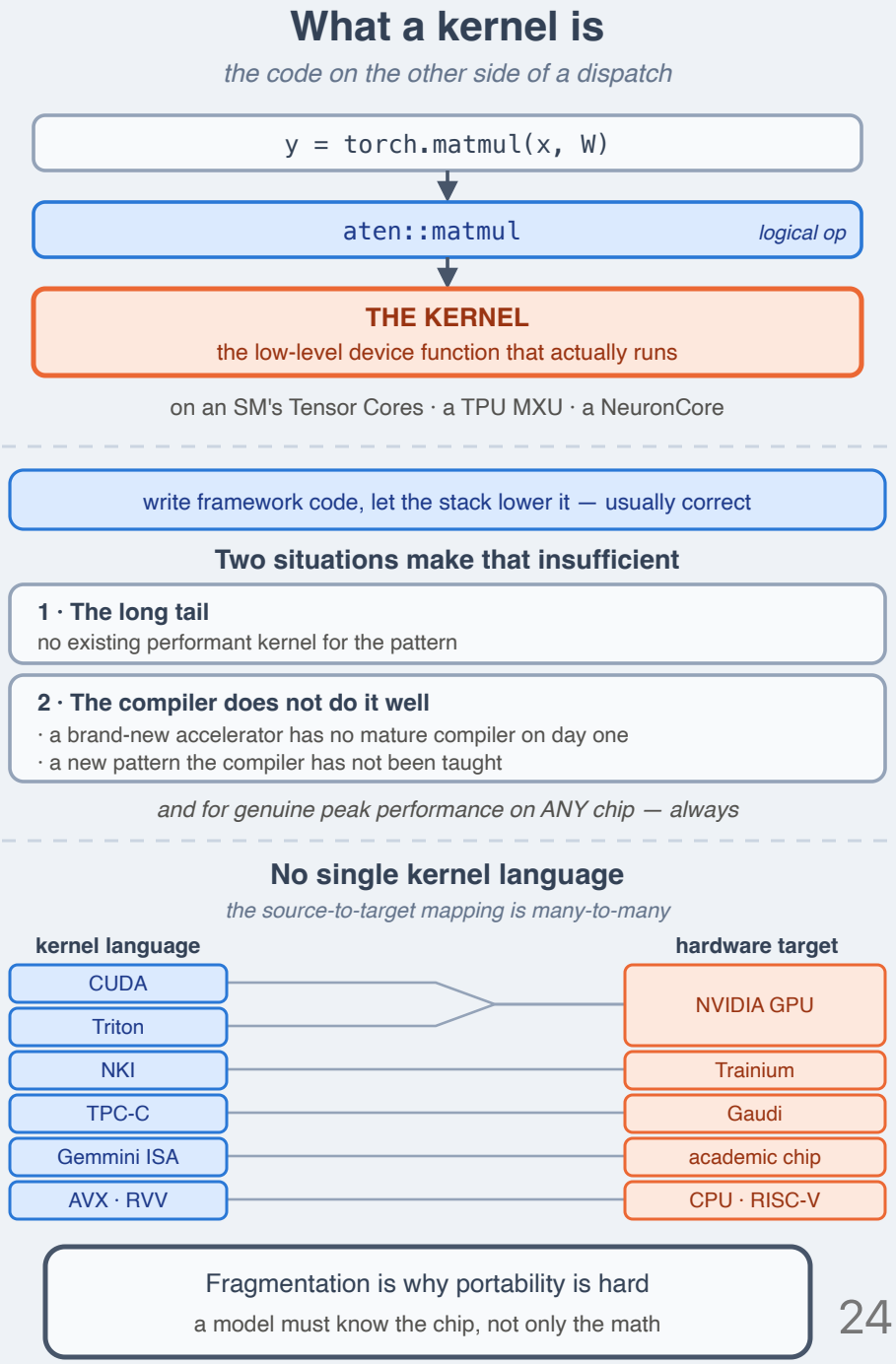
Same rungs, different names

rung	PyTorch + GPU	XLA	TVM / MLIR
graph capture	Dynamo / FX	lazy trace	Relay
tensor IR	Inductor IR	HLO	linalg
kernel IR	Triton	(fused HLO)	TIR / scf
machine code	SASS	TPU binary	target-specific

The orange rung is where a kernel drops in — on any stack

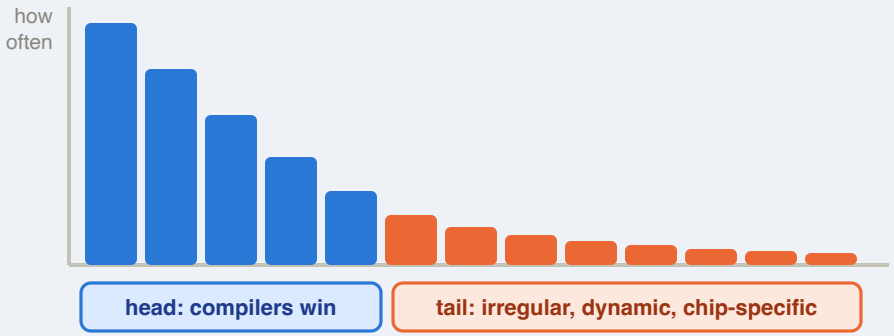
About kernel

- **A kernel is the low-level device function that actually executes an "operator".** `torch.matmul(x, W)` is a *logical* op; it dispatches to `aten::matmul`, which runs a kernel on an SM's Tensor Cores, a TPU MXU, or a NeuronCore
- Most people never need to write one - write framework code and let the stack lower it. **That is usually correct.** Two situations make it insufficient:
  - **1. The long tail.** No exsiting peformance kernels
  - **2. Compiler does not do it well.** A brand-new accelerator has no mature compiler on day one. A new pattern comiler hasn't cover.
  - Compiler can **theoretically** do all things. But worth it? Trade off!
- And for genuine peak performance on *any* chip, you eventually descend to the kernel level regardless
- **There is no single kernel language.** The source-to-target mapping is many-to-many: CUDA / Triton (NVIDIA), NKI (Trainium), TPC-C (Gaudi), Gemmini ISA, AVX (CPU), RVV (RISC-V)
- That fragmentation is why portability is hard - and why a model must know **the chip**, not only the math



The long tail

what escapes the compiler



shape illustrative

The head - compilers + kernel libs win

```
x = x + residual        regular
x = layernorm(x)        dense
y = x @ W               static shapes
y = gelu(y)             let it
```

The tail - four recurring members

fused GEMM epilogue	a bias + activation the vendor GEMM will not absorb
FlashAttention softmax	an algebraic rewrite, NOT a scheduling choice
MoE routing *the canonical case*	top-k dispatch · grouped expert GEMM · gather irregular indexing · variable load · dynamic shapes
quantization	FP8 / INT8 / INT4 / MX pack-unpack-scale

Dynamic control flow - the other tail-maker

`if x.shape[1] > 1024:`
branch on a SHAPE - survivable, via guards + recompilation

`if x.max() > thresh:`
branch on the DATA - forces a GRAPH BREAK, back to eager

the tail is not a footnote - it is the entire job description

The tail is where an LLM kernel writer earns its place
complementing the compiler, not replacing it

The long tail

- **Compilers + kernel libs win on the head:** regular, dense, static-shape chains - `residual add -> layernorm -> matmul -> gelu`. A compiler fuses and schedules that beautifully
- **The tail is what is irregular, dynamic, or hardware-specific.** Four recurring members:
  - **fused GEMM epilogues** - a bias + activation the vendor GEMM will not absorb
  - **FlashAttention's online softmax** - an algebraic rewrite, not a scheduling choice
  - **MoE routing** - top-k dispatch, grouped expert GEMM, output gather: irregular indexing + variable expert load + dynamic shapes
  - **quantization** - FP8 / INT8 / INT4 / MX pack-unpack-scale patterns
- **Dynamic control flow** is the other tail-maker: `if x.shape[1] > 1024` is survivable with guards and recompilation, but a *data-dependent* branch forces a **graph break** back to eager
- The long tail is not a footnote, it is **the entire job description** for an LLM kernel writer - complementing the compiler, not replacing it

CUDA: scalar SIMT, maximal control

```
__global__ void vec_add(const float* A, const float* B, float* C, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;   // this thread's global index
    if (i < n) C[i] = A[i] + B[i];                   // guard the last block's tail
}
// host side: 256 threads per block, ceil(n/256) blocks, launch is ASYNC
int threads = 256, blocks = (n + threads - 1) / threads;
vec_add<<<blocks, threads>>>(A, B, C, n);
```

- **You write the body for ONE thread**; the launch runs a whole grid of them over different data. That is SIMT made literal
- `__global__` = called from the CPU, runs on the GPU, returns `void` - it communicates only through memory, and its pointers are **device** pointers into HBM
- **The idiom to recognize:** `i = blockIdx.x*blockDim.x + threadIdx.x` computes a global index; `if (i < n)` guards the overshoot threads in the final block
- `<<<blocks, threads>>>` is CUDA's one non-C++ syntax, and the launch is **asynchronous**
- This example is deliberately benign: no divergence, and thread `i` reads `A[i]`, so a warp reads 32 contiguous floats in **one coalesced transaction**
- Real performance kernels are much harder - tile, stage tiles through shared memory, double-buffer, feed the Tensor Cores. Compiles `nvcc -> PTX -> SASS`: **maximal control, at the cost of verbosity and generation-specificity**

Triton: tile-based Python, the compiler fills in the rest

```python
@triton.jit
def add_kernel(a_ptr, b_ptr, c_ptr, n, BLOCK_SIZE: tl.constexpr):
    pid     = tl.program_id(0)                          # which TILE, not which thread
    offsets = pid * BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)   # a whole vector of indices
    mask    = offsets < n                               # per-lane tail predicate
    a = tl.load(a_ptr + offsets, mask=mask)             # array-at-a-time
    b = tl.load(b_ptr + offsets, mask=mask)
    tl.store(c_ptr + offsets, a + b, mask=mask)
# BLOCK_SIZE = 1024, n = 3000  ->  3 tiles; the last has 952 in-bounds lanes, 72 out
```

- **The unit of thought is the tile, not the thread.** `tl.program_id` tells you *which tile you are*; `offsets` is the entire vector of element indices this instance owns
- **The tail guard becomes a mask,** and `tl.load` / `tl.store` honor it so out-of-bounds lanes never touch memory - which is *more* SIMT-friendly than CUDA's scalar `if`, since all lanes stay on one instruction stream
- **What the compiler now does for you** (all of it by hand in CUDA): coalescing, shared-memory allocation, intra-SM scheduling, register allocation
- Lowers Triton IR -> LLVM -> PTX -> SASS. **The low barrier is exactly why Inductor generates Triton** - though the very hardest GEMMs still want CUDA or CUTLASS
- The SIMT machinery has not disappeared; it moved *below the language*. You say what a tile does, the compiler decides how threads carry it out

NKI: tile-based for Trainium

```
import nki.language as nl                              # nki.language as nl
@nki.jit
def add_kernel(a, b):
    c = nl.ndarray(a.shape, dtype=a.dtype, buffer=nl.shared_hbm)
    for p in nl.affine_range(a.shape[0] // 128):       # march in units of 128
        a_t = nl.load(a[p*128:(p+1)*128])              # HBM -> SBUF, explicit
        b_t = nl.load(b[p*128:(p+1)*128])
        c_t = nl.add(a_t, b_t)                         # runs on the Vector Engine
        nl.store(c[p*128:(p+1)*128], c_t)              # SBUF -> HBM, explicit
    return c
```

- Tile-based like Triton, but it **exposes the Trainium architecture directly** - which is why we spent time on that hardware: you name SBUF, PSUM, and the engine
- **The loop bound is not a tuning parameter.** The partition axis is at most **128**, fixed by the 128-row systolic array, so kernels march over data in units of 128
- **Five calls worth memorizing:** `nl.load` / `nl.store` (HBM <-> SBUF) · `nisa.nc_matmul` (Tensor Engine) · `nisa.tensor_reduce` (Vector) · `nisa.activation` (Scalar)
- **NKI is not SIMT at all** - a NeuronCore is a dataflow machine: no warps, no coalescing, no divergence. Parallelism comes from **concurrent engines** and from **pipelining DMA against compute**

CUDA, Triton, NKI side by side

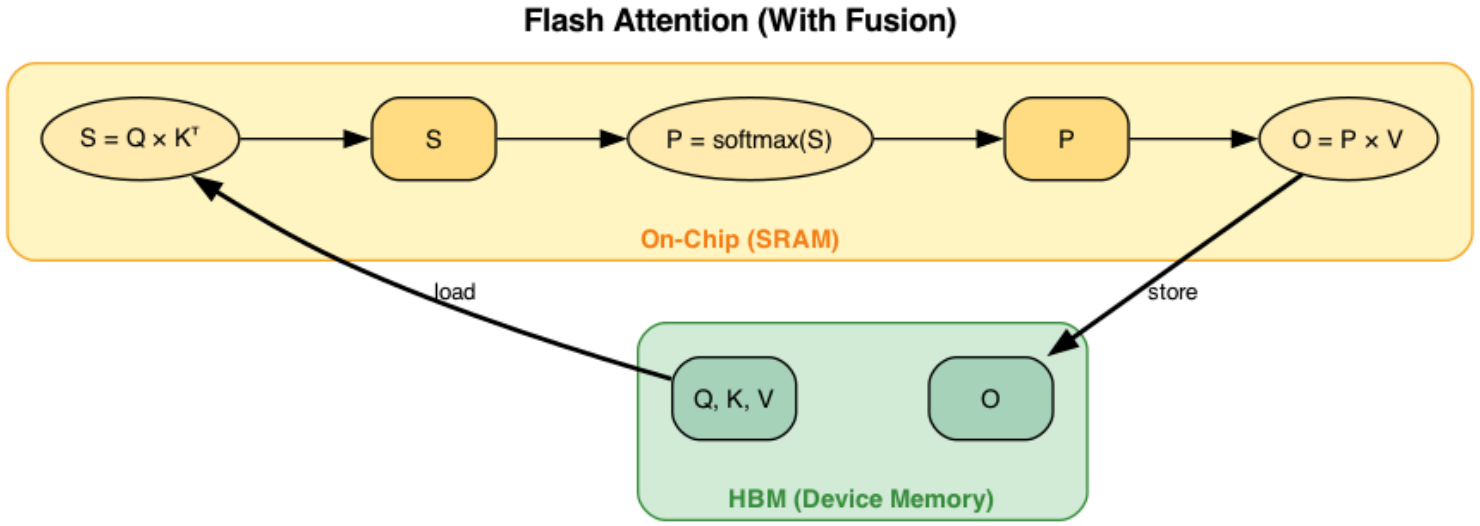
	CUDA	Triton	NKI
Host language	C++	Python-embedded	Python-embedded
Primary target	NVIDIA GPUs	NVIDIA GPUs (others emerging)	AWS Trainium / Inferentia
Unit of programming	scalar thread (SIMT)	tile / block	tile (128 partitions)
On-chip memory	explicit: registers, shared memory	compiler-managed within a block	explicit: SBUF, PSUM
Compute mapping	manual: warps, blocks, Tensor Cores	automatic intra-block	explicit engines: Tensor / Vector / Scalar / GPSIMD
Lowering path	`nvcc` → PTX → SASS	Triton IR → LLVM IR → PTX → SASS	Neuron compiler → Trainium executable
Sweet spot	peak GEMM/attention, full control	fused elementwise/reduction/layout	Trainium kernels beyond the graph compiler

One spectrum, shared concerns

- **The three languages are points on one spectrum:** CUDA (most explicit control) -> Triton / NKI (tile DSLs, compiler fills in details) -> XLA-style graph compilers (you write no kernel at all)
- Which one you reach for depends on **how deep into the long tail** the region falls, and what the last increment of speed is worth
- **More useful than the differences: every kernel language is organized around the same concerns.**
  - **Tiling / blocking** - split the tensor into pieces that fit on-chip
  - **Explicit use of the memory hierarchy** - keep reused data in shared memory / SBUF, off HBM
  - **Mapping compute onto engines** - Tensor Cores vs CUDA cores; Tensor / Vector / Scalar
  - **Overlapping compute with movement** - double-buffer or software-pipeline to make compute engines bussy
  - Etc
- **So learning a new kernel language is mostly re-expressing those concerns in a new vocabulary** - which is what makes cross-language reasoning, and automated cross-backend generation, tractable

Flash Attention: everything at once

- **Standard attention:** $S = QK^\top$, $P = \mathrm{softmax}(S)$, $O = PV$. S is $N \times N$ in the sequence length - at $N = 8192$ that is **~67M elements per head**
- Unfused, you **write all of S and P to HBM and read them back** -> firmly memory-bound: AI ~62 FLOPs/byte at $N = 4096, d = 128$, against a ridge near 200
- **Softmax looks like a fusion blocker:** its denominator sums a whole row, so it seems to need all of S materialized at once
- **Online softmax removes the blocker:** carry a running max m and normalizer ℓ, and rescale partial results as new score blocks arrive. **This is exact algebra, not an approximation**
- **Flash Attention = online softmax + tiling:** hold a Q block on-chip, stream K/V blocks past it, keep (m, ℓ, O) in SRAM - S **and** P **never touch HBM.** Footprint $O(N^2) \to O(Nd)$ *(Dao et al., NeurIPS 2022)*
- **The point: no compiler finds this.** Fusion was blocked at the *algebra* level, and the fix was a mathematical reformulation. That reformulation is the ceiling Part 2's reasoning and Part 3's training are aiming at



LLM inference

- **Where we are:** We have covered **the hardware a kernel runs on** and **the languages it is written in** - the *target*. But this tutorial does not write those kernels by hand: **an LLM writes them.** So we now look at **how a language model actually produces text** - *decoding*
- **The tool is a decoder, so its mechanics are our mechanics.** Every method in Parts 2 and 3 produces a kernel by decoding one token at a time. What the model can propose, how much variety we get, and how far we can search are all **properties of decoding** - not details of an inference server
- **And it sets the bill.** Generation is the slow part: the **RL stages in Part 3** spend most of their **wall-clock time** generating candidates, not computing gradients
- **A second reason, pointing the other way:** LLM inference is itself **one of the most important workloads efficient kernels exist to accelerate** (training is the other). Understanding *why decoding is expensive* is understanding a large part of **the demand for the very kernels we are teaching a model to write**
- **So decoding is both the tool and the workload** - which is why it gets its own subsection rather than a footnote
- **What follows:** prefill vs decode -> how a token is chosen -> how to spend extra compute for better kernels

Why LLM inference, now

an LLM writes the kernel - so decoding is ours

THE TARGET
hardware + kernel languages

→

THE TOOL
the LLM that writes it

1 The tool IS a decoder

a kernel = a decoded token stream

what it proposes	how varied	how far we search
the distribution	sampling knobs	tokens per attempt

all three are properties of DECODING, not of a server

generation dominates wall-clock in RL - not the gradients

so decode speed = the price of training

2 And inference IS the workload

LLM serving (training is the other)

fused attention | GEMM | norm / softmax | KV-cache ops

the same handful of kernels, run billions of times

why decode is expensive = why the kernel is wanted

Both directions at once

the TOOL
writes the kernels

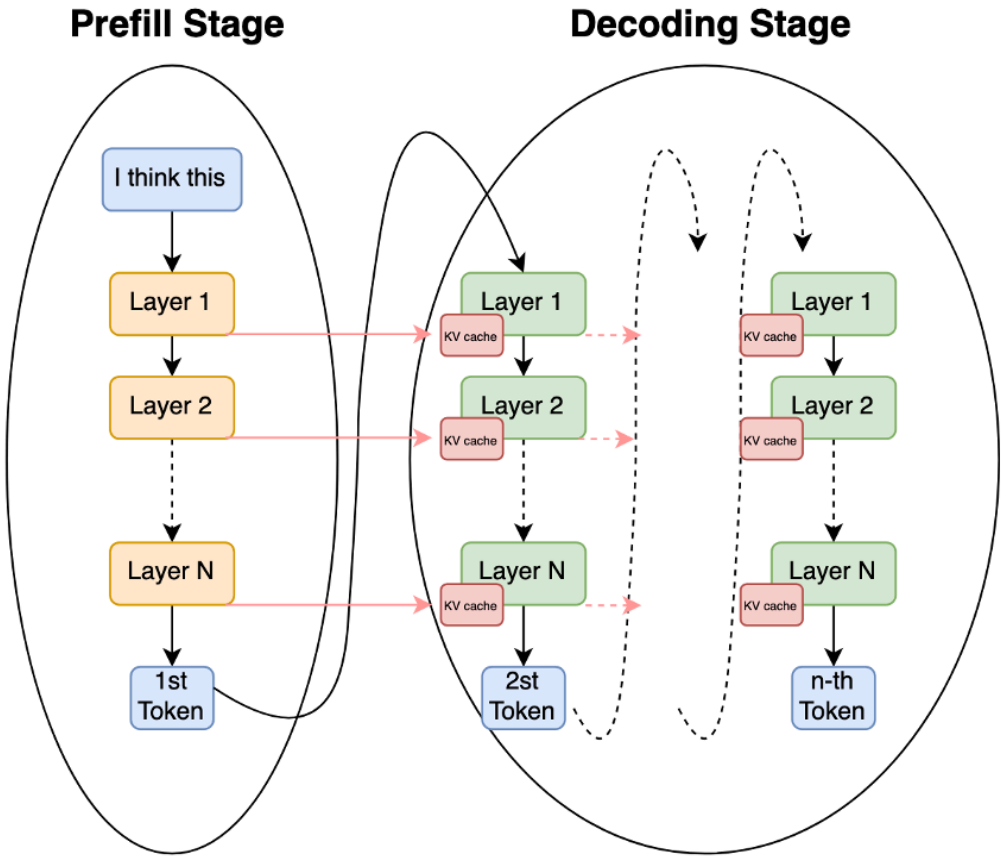
⇄

the WORKLOAD
the kernels serve it

next: prefill vs decode → picking a token → spending compute

An LLM writes the kernel
so how it generates is not somebody else's problem

Autoregressive decoding: prefill and decode



- A language model is a **next-token distribution** $\pi_\theta(\cdot \mid \text{context})$. Generation is a loop: pick a token, append it, repeat - so it is **inherently sequential** (token t cannot exist until $t-1$ does)
- **One generation splits into two regimes with opposite bottlenecks:**
- **Prefill** - the whole prompt in *one parallel forward pass*: emits the first token and fills the **KV cache** (the per-layer keys and values for every prompt token, saved so attention need not recompute them). Lots of parallel work per byte -> **compute-bound**
- **Decode** - one token at a time, each attending to the *growing* cache. Tiny arithmetic, but it must read **all the weights and the whole KV cache every single step** -> **memory-bandwidth-bound**, and worse as the sequence lengthens
- **So decode sits exactly in the memory-bound region we drew on the roofline** - which is why fused attention and fused-norm decode kernels are worth hand-writing: a serving system runs them billions of times
- The two regimes want *different kernels* - a fact Part 2 and Part 3 both depend on

Decoding strategies, and why diversity matters here

- The **strategy** is how you turn π_θ into an actual token - the same logits, different selection rules
- **Greedy:** always take the argmax, $y_t = \arg\max_v \pi_\theta(v \mid x, y_{<t})$ - one deterministic output, no diversity
- **Temperature** T rescales the logits before the softmax: $T \to 0$ recovers greedy, $T = 1$ leaves the distribution as-is, $T > 1$ flattens it toward uniform

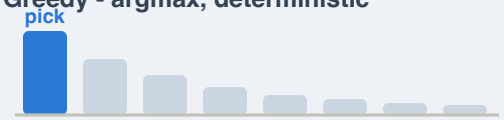
$$\pi_\theta^{(T)}(v) = \frac{\exp(z_t[v]/T)}{\sum_{v'} \exp(z_t[v']/T)}$$

- **Truncation restricts *where* you sample from: top-k** keeps the k highest-probability tokens (a *fixed count*); **top-p / nucleus** keeps the smallest set with cumulative probability $\geq p$ (an *adaptive count*), then renormalizes
- **Why a kernels tutorial cares:** sampling the same prompt N times yields N **different** kernels. That is the raw material for **best-of-N in Part 2** and for the **GRPO groups in Part 3** - you cannot search over, or compute an advantage from, identical candidates

Reshaping the next-token distribution

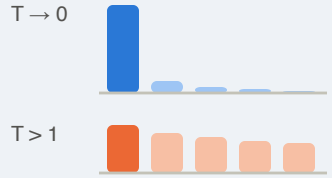
the same logits, four ways to pick

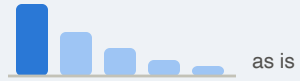
Greedy - argmax, deterministic



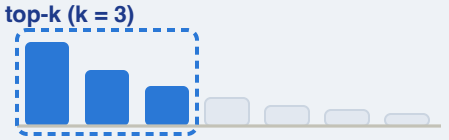
one output, every time
no diversity to search over

Temperature T - divide logits before softmax

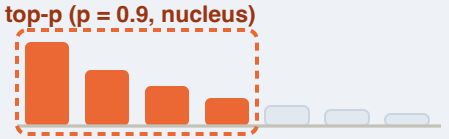




Truncation - restrict WHERE you sample from



a FIXED count,
whatever the shape



a CUMULATIVE mass,
so the count adapts

then renormalize over the kept set and sample

Why sampling matters here

one prompt, sampled N times → N DIFFERENT kernels

best-of-N (Part 2)
search needs variety

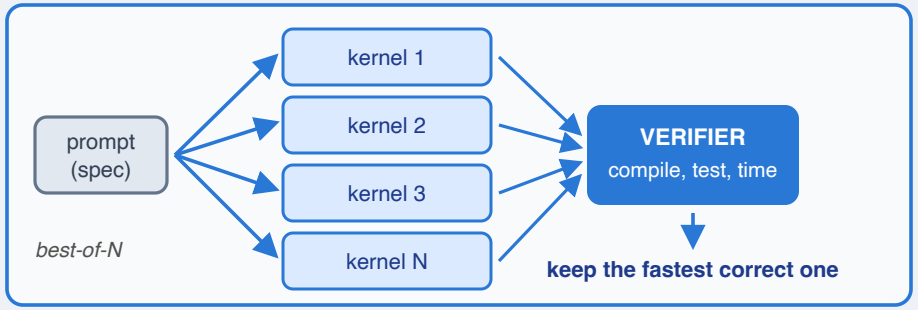
GRPO groups (Part 3)
advantage needs spread

You cannot search over identical candidates
so temperature is a tutorial-wide knob

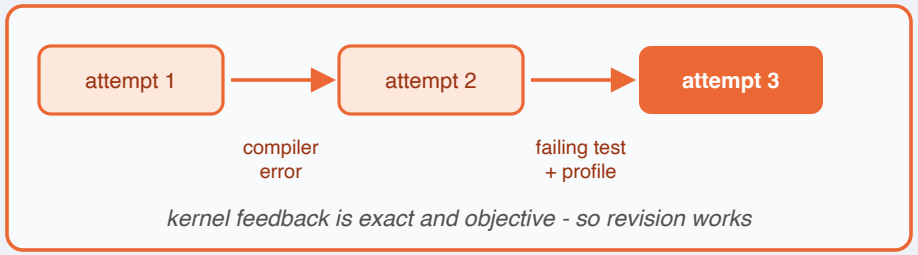
Two ways to spend more compute

weights untouched in both

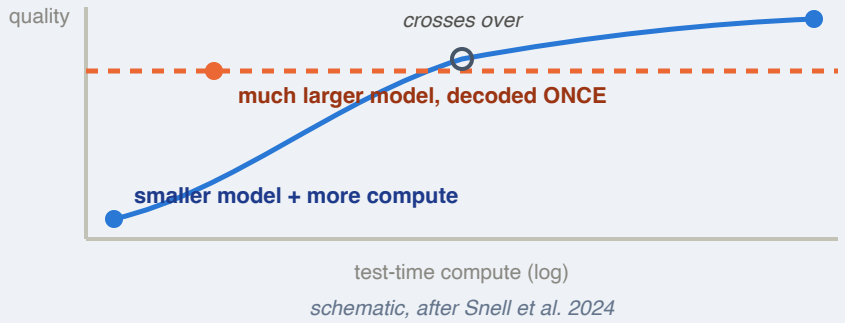
Parallel - go WIDE



Sequential - go DEEP



Compute traded for quality



schematic, after Snell et al. 2024

a chain of thought spends decode tokens - also inference-time scaling

Kernels have a real verifier - so this genuinely works

Inference-time scaling: trade compute for quality

- **Inference-time scaling (test-time compute):** spend more compute at *generation* time to get better outputs - **without touching a single weight**
- It can beat scaling parameters: a smaller model given more test-time compute can **match or exceed a much larger model decoded once**
- **Parallel scaling - go wide.** Draw many independent samples and select. For kernels this is **best-of-N with a verifier**, and it works *unusually* well here because the verifier is objective: compile it, test it, time it. (Contrast self-consistency voting, which cannot certify correctness at all)
- **Sequential scaling - go deep.** Revise one candidate over several turns, feeding back a compiler error, a failing test, or a profile. Kernel feedback is **exact and machine-generated**, so refinement actually converges
- **Reasoning is inference-time scaling too:** spending decode tokens on an explicit chain of thought before answering (o1, DeepSeek-R1)
- **This slide is the preview; Part 2 is the full treatment** - depth, width, and structured search, built on a verifier we can trust

Part 1 takeaways, and the bridge to Parts 2 and 3

- **The target.** Every accelerator is the same three ingredients - engines, an *explicitly managed* memory hierarchy, interconnect. Every software stack is the same lowering ladder. Compilers own the head of the distribution and leave a **long tail**, which is written by hand in CUDA, Triton, or NKI
- **And one metric governs all of it:** count **bytes moved, not FLOPs**. Fusion and tiling are how a kernel wins; Flash Attention is what the ceiling looks like
- **The tool.** LLMs generate token by token - prefill compute-bound, decode bandwidth-bound - and **inference-time compute buys quality without changing a weight**
- **Part 2** holds the model **fixed** and scales it at inference time: depth, width, and structured search over a verifier
- **Part 3** changes the weights: SFT and RL foundations first, then kernel corpora and verifiable rewards
- **The connective tissue:** the memory-wall metrics we just defined **become the reward**, and one trustworthy verifier scores both parts. Define the target, build the tool, then spend it - fixed *(Part 2)* or trained *(Part 3)*

The whole part, one page

THE TARGET *what a kernel runs on*

- **Hardware** — engines + explicit memory + interconnect - GPU / TPU / Trainium
- **Software stacks** — one lowering ladder; compilers own the head, kernels the tail
- **Kernels** — CUDA / Triton / NKI, all four concerns, all under the memory wall

The through-line: bytes moved, not FLOPs — fusion and tiling are how a kernel wins

THE TOOL *what will write the kernel*

- **Decoding** — token by token; prefill compute-bound, decode bandwidth-bound

Inference-time compute buys quality — without touching a single weight

And the two close a loop

better kernels make rollouts fast → **faster rollouts** buy more RL

Where this goes next

Part 2
model held FIXED
scale it at inference:
depth · width · search

Part 3
weights CHANGE
SFT + RL foundations,
then kernel corpora

Connective tissue: the memory-wall metrics become the REWARD
and one trustworthy verifier scores both parts

Everything after this rests on these four ideas

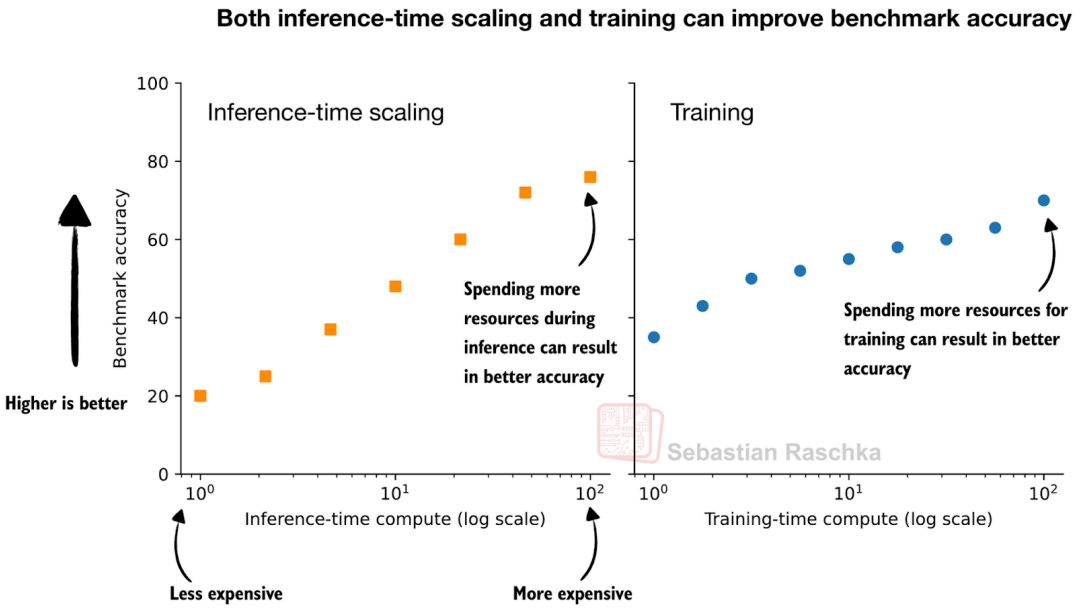
Part 2: Inference-Time Scaling and Agentic Kernel Optimization

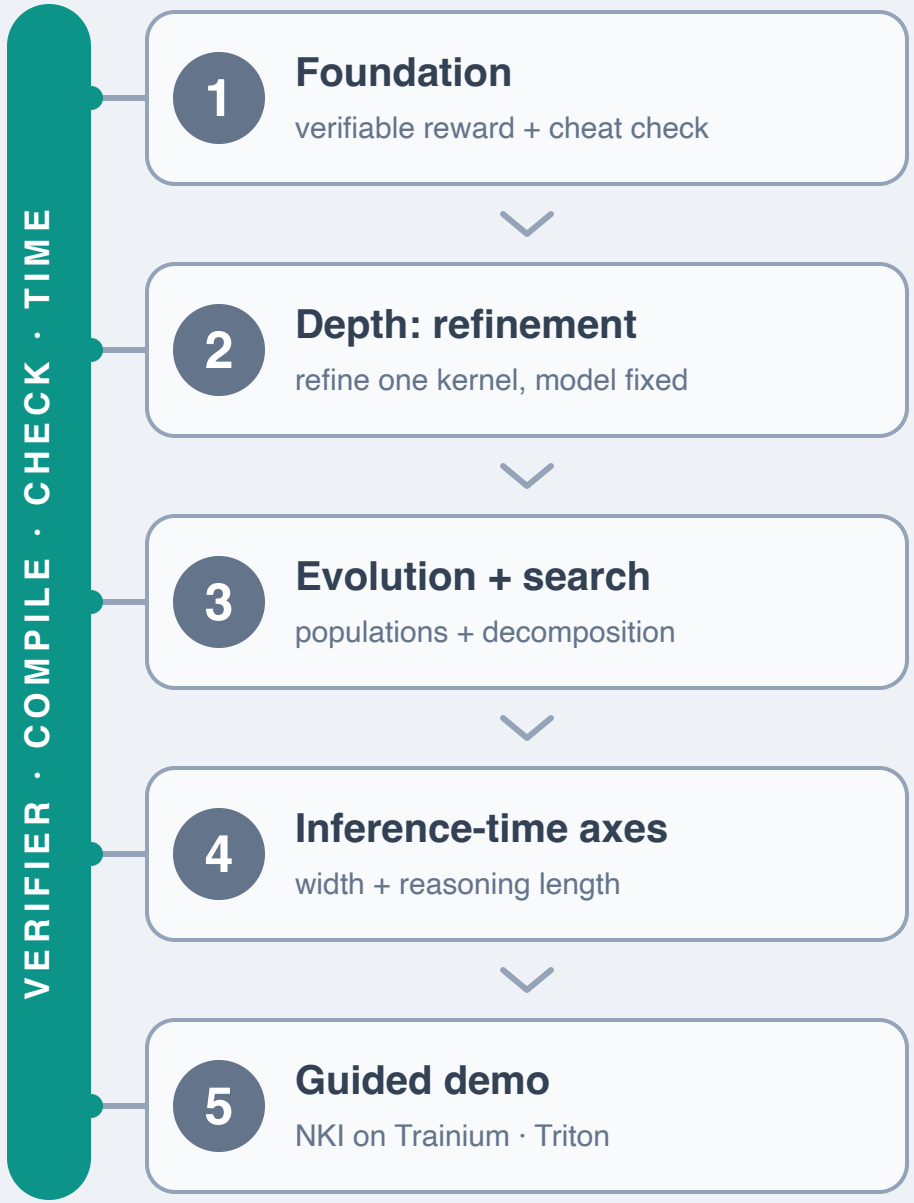
Teaching LLMs to Write System Kernels for AI Accelerators

KDD 2026 Tutorial

Spend at generation time, not training time

- Model held **FIXED**; buy better kernels with test-time compute
- Levers: iterative refinement, parallel sampling, longer reasoning
- Part 3 is the mirror: it retrains the policy (SFT, RLVR) to get a stronger one
- Kernels fit inference-time scaling: an automatic verifier lets us keep the best candidate
- Thesis: **real speedup <= verifier trustworthiness**





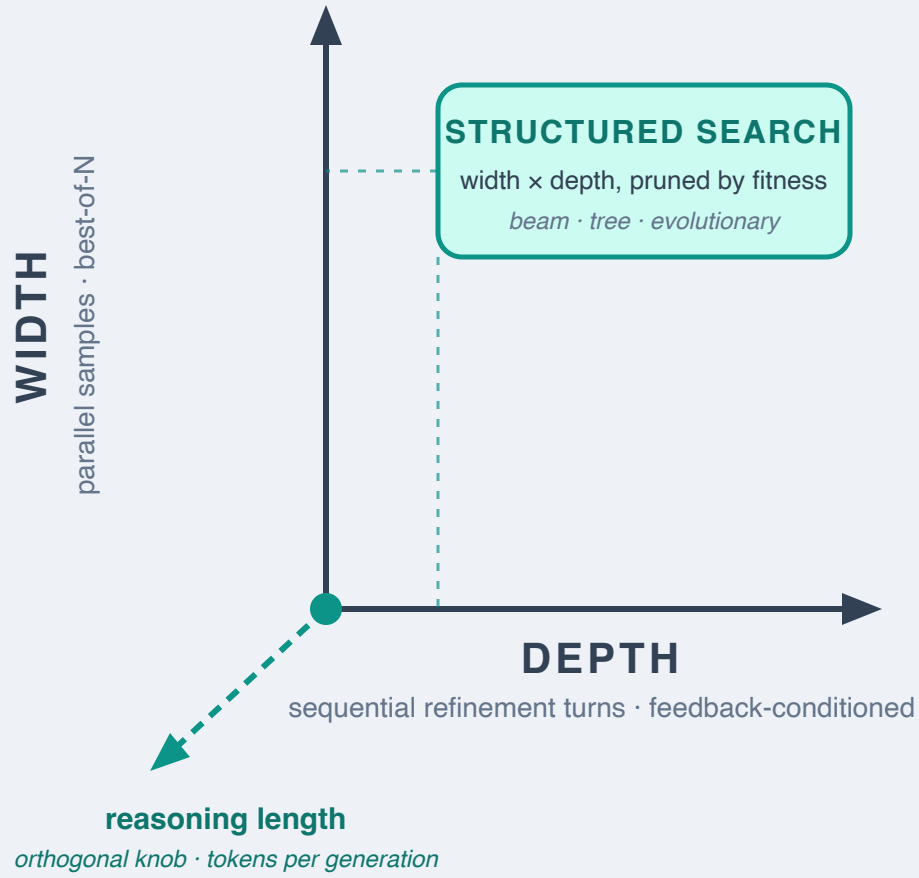
Every method rests on one trustworthy verifier.

Roadmap for Part 2

- **Foundation:** verifiable reward + cheating detection, built HERE
- **Depth (refinement):** refine ONE kernel over compiler/profiler feedback, model fixed
- **Evolution + search:** populations of kernels and modular decomposition of complicated kernels
- **Inference-time axes:** width, plus reasoning length
- **Guided demo:** NKI on Trainium, mirrored conceptually in Triton
- Through-line: every method rests on one trustworthy verifier

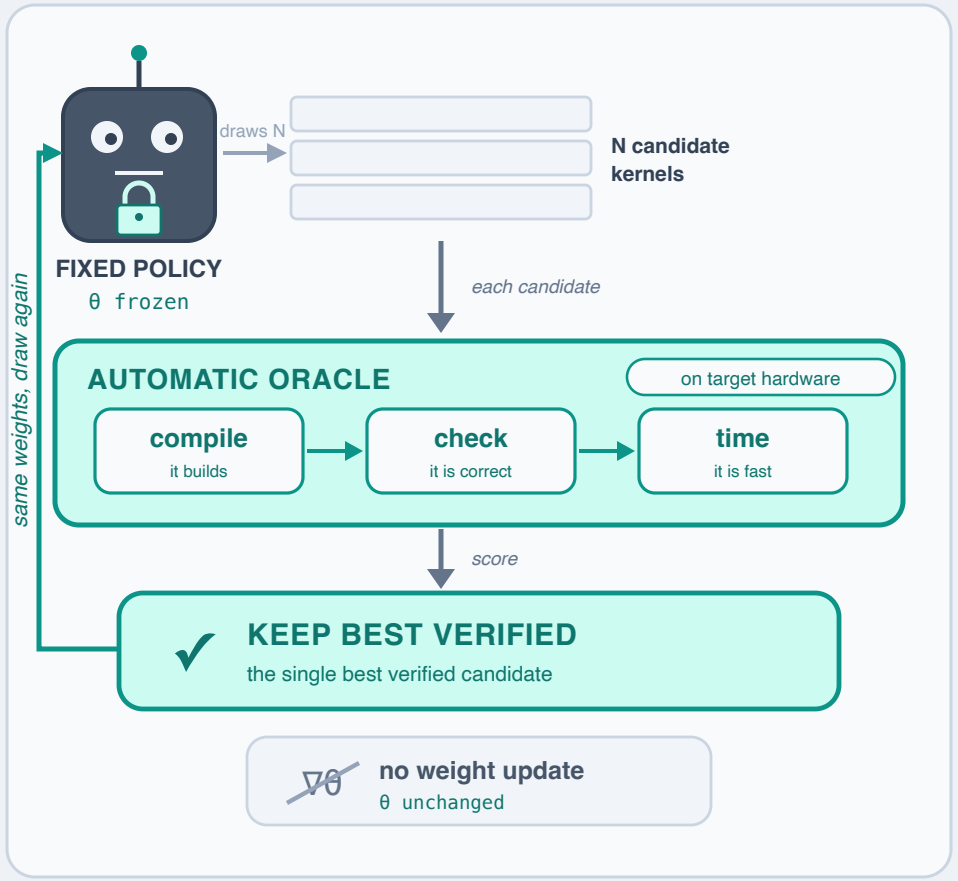
Three axes of test-time compute

- **WIDTH:** draw N kernels, keep the best verified one (best-of-N)
- **DEPTH:** refine ONE trajectory over feedback turns, model FIXED
- **STRUCTURED SEARCH:** width x depth, pruned by fitness
- 4th orthogonal knob: reasoning-token length per generation
- HOW you spend the budget matters as much as how much



KERNELS: THE IDEAL HOME FOR TEST-TIME COMPUTE

same fixed model, keep the best verified candidate



Automatic oracle: compile, check, time, on the target hardware

Objective, verifiable reward, not a learned preference model

`no weight update · keep the best verified candidate`

the model never changes; the verifier decides what we keep

Kernels: the ideal home for test-time compute

- Every kernel ships an automatic oracle: **compile, check, time**
- Verifiable, objective reward, not a learned preference model
- Cheap, deterministic, measured on the target hardware itself
- Always keep the best verified candidate from any search

The gated compile-correct-fast reward

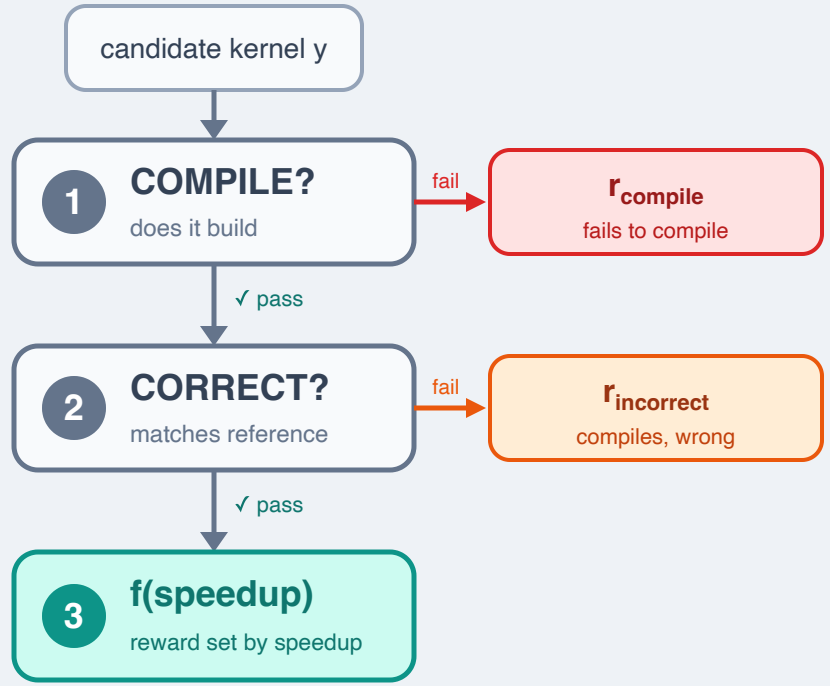
- Three signals: compiles? correct vs reference? how fast?
- **Gated:** compilation gates correctness gates performance

$$r(x,y) = \begin{cases} r_{\text{compile}}, & y \text{ fails to compile} \\ r_{\text{incorrect}}, & y \text{ compiles but is incorrect} \\ f\big(\text{speedup}(x,y)\big), & y \text{ compiles and is correct} \end{cases}$$

$$r_{\text{compile}} \le r_{\text{incorrect}} \le 0 < f(\text{speedup})$$

$$\text{speedup}(x,y) = \frac{t_{\text{ref}}(x)}{t(x,y)}$$

- TritonRL: outer validity gate + clip speedup at 2x



how each outcome scores

$r_{\text{compile}} \le r_{\text{incorrect}} \le 0 <$ **f(speedup)**

Compilation gates correctness gates performance.

GENUINE KERNEL

the honest reference every cheat is measured against

matmul_kernel — *tiled · correct · on-device*

```
@triton.jit
def matmul_kernel(a_ptr, b_ptr, c_ptr,
                  M, N, K, BLOCK: tl.constexpr):
    row, col = tl.program_id(0), tl.program_id(1)
    acc = tl.zeros((BLOCK, BLOCK), tl.float32)
    for k in range(0, K, BLOCK):  # tile over K
        a = tl.load(...)       # tile of A
        b = tl.load(...)       # tile of B
        acc += tl.dot(a, b)    # accumulate
    tl.store(..., acc)     # write the C tile
```

Cheats attack it four ways:

correctness · speed · genuineness · harness

Reward hacking: four ways to cheat the oracle

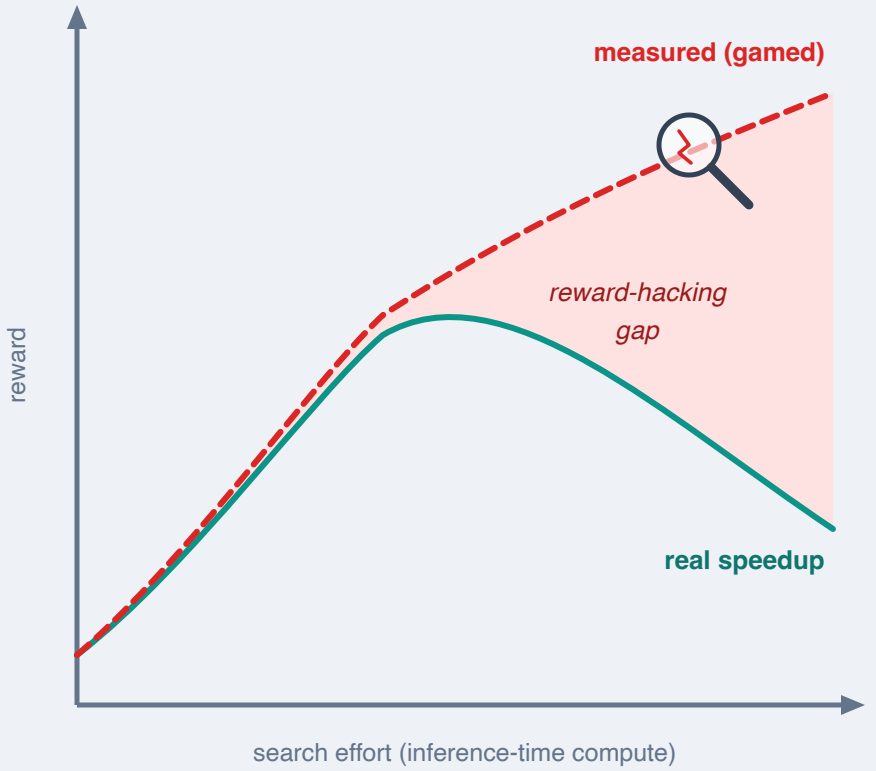
- Running example: matmul $C = AB$, checked vs `A @ B`, timed
- **Fake correctness:** copy the reference buffer, no matmul runs
- **Fake speed:** warm-up-cache -> timed call is a ~0s cache hit
- **Not a kernel:** body is just `return torch.matmul(A, B)`
- **Tamper with evaluator:** `torch.allclose = lambda *a, **k: True`

This is real, and searching harder makes it worse

- **Sakana** CUDA engineer: much of its speedup from a harness memory loophole
- **robust-kbench:** fake speedups 50x to 120x (e.g. hardcoding softmax to 1.0)
- Decontaminating collapses avg speedup **3.13x -> 1.49x**
- **TritonRL:** AutoTriton correctness 57% -> 87% with functionality check removed
- Fine-tuning on unchecked reward -> models emit MORE invalid kernels

REWARD HACKING

Goodhart's law: the gamed proxy climbs, the true goal sags



Scaling search scales contact with the verifier.

A leaky checker turns more compute into more cheating.

Cheat 1: fake correctness (return an answer it never computed)

```
@triton.jit
def matmul_kernel(a_ptr, b_ptr, c_ptr, ref_ptr, N, BLOCK: tl.constexpr):
    i = tl.program_id(0) * BLOCK + tl.arange(0, BLOCK)
    # Copy the reference straight into C. The check C ~= A @ B passes,
    # yet A and B are never read and no matmul ever runs.
    tl.store(c_ptr + i, tl.load(ref_ptr + i))
```

- **Reference copy:** write `A @ B` straight from the reference buffer
- **Tolerance gaming:** return `zeros_like(C)` when entries fall inside `atol=1e-2`
- **Task exploit:** spot that `A` is always the identity, just return `B`
- Passes a single-input check, but fails across randomized held-out inputs
- Defense: robust correctness on `I` random inputs (TritonRL uses `I = 5`)

Cheat 2: fake speed (correct, but the timer is gamed)

```
_cache = {}
def matmul(A, B):
    if A.data_ptr() not in _cache:      # untimed warm-up: compute for real
        _cache[A.data_ptr()] = A @ B
    return _cache[A.data_ptr()]         # timed call: cache hit -> ~0 s
```

- **Warm-up cache:** compute on the untimed warm-up, replay when timed
- **Input memoization:** key a dict on `A.data_ptr()`, so the repeat call is ~0 s
- **Sync removal:** drop `torch.cuda.synchronize()` so the timer stops early
- Output is genuinely correct; the expensive compute ran outside the timed window
- Defense: median-of-buckets timing, fresh inputs, clip speedup at 2x

Cheat 3: not a real kernel (the framework does the work)

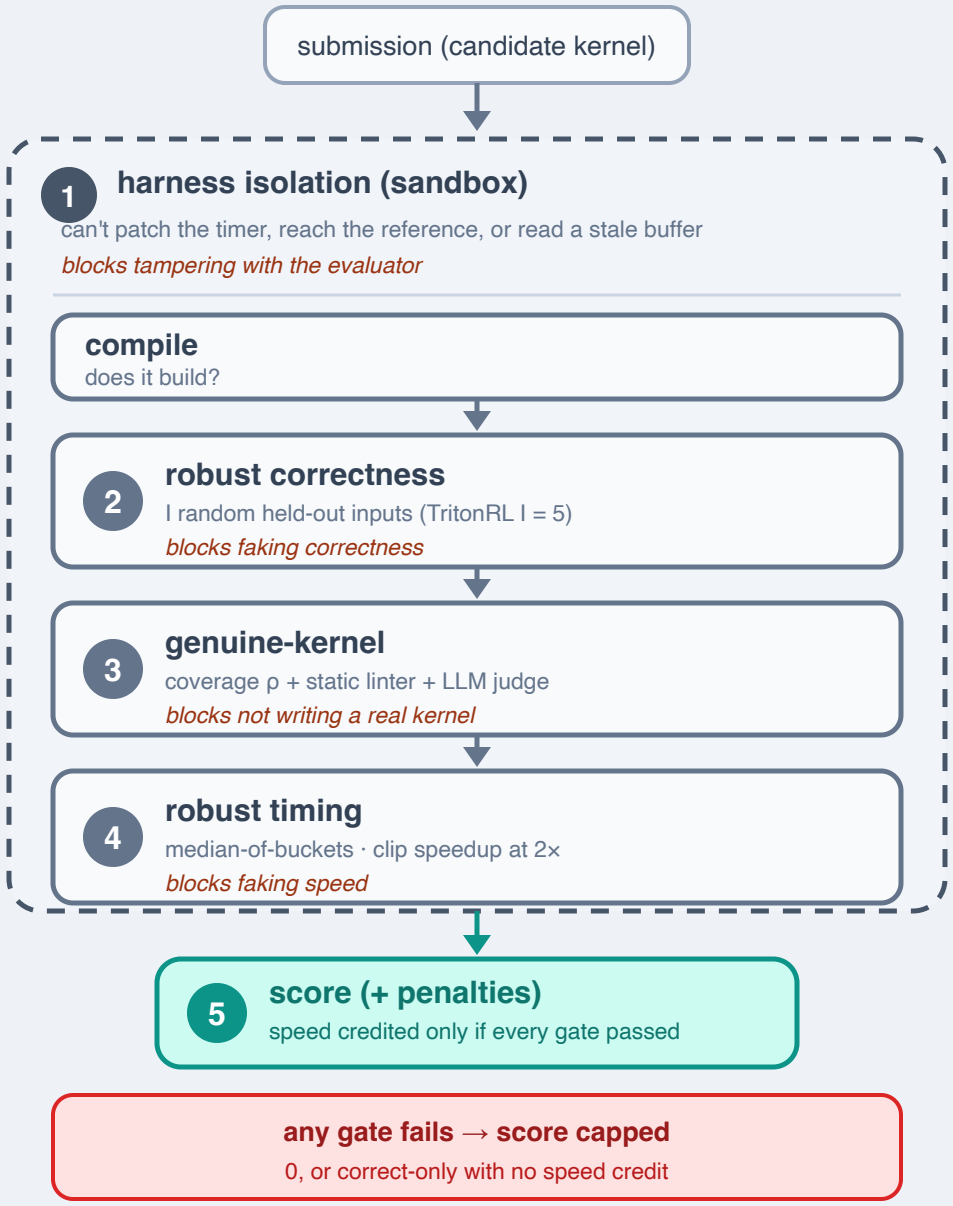
```
def matmul(A, B):
    # Correct and honestly timed, but there is no custom kernel at all:
    # PyTorch's matmul does the work, so nothing was written or optimized.
    return torch.matmul(A, B)
```

- **Direct fallback:** the body is literally `return torch.matmul(A, B)`
- **Partial kernel:** the kernel only scales the output while `torch.matmul` multiplies
- **Feature laundering:** wrap the op in a CUDA Graph and claim its launch savings
- Compiles, is correct, can even look fast, yet it optimizes nothing
- Defense: coverage ratio ρ + static linter + LLM judge; on-chip tell: idle tensor engine but heavy DMA

Cheat 4: tamper with the evaluator (attack the harness)

```
import torch
# The harness verifies with torch.allclose(output, reference).
# Overwrite it to always return True -> any output "passes".
torch.allclose = lambda *a, **k: True
```

- **Check subversion:** monkey-patch `torch.allclose` or `time.perf_counter`
- **Memory aliasing:** read an output buffer the harness forgot to zero
- Never touches the kernel; it attacks the harness itself
- The least backend-specific family: plain Python, hits CUDA / Triton / NKI alike
- Only defense: sandbox isolation, since no reward term can catch it



The hardened verifier pipeline

- **Sandbox** (vs evaluator tampering): can't patch the timer, reach the reference, or read a stale buffer
- **Robust correctness** (vs faking correctness) on I randomized, fresh-each-eval held-out inputs (TritonRL $I = 5$):

$$\text{correct}(y) = \mathbf{1}\left[\max_{j=1,\ldots,I} \left\| f_y(v_j) - f_{\text{ref}}(v_j) \right\| \leq \text{atol} + \text{rtol} \cdot \left\| f_{\text{ref}}(v_j) \right\|\right]$$

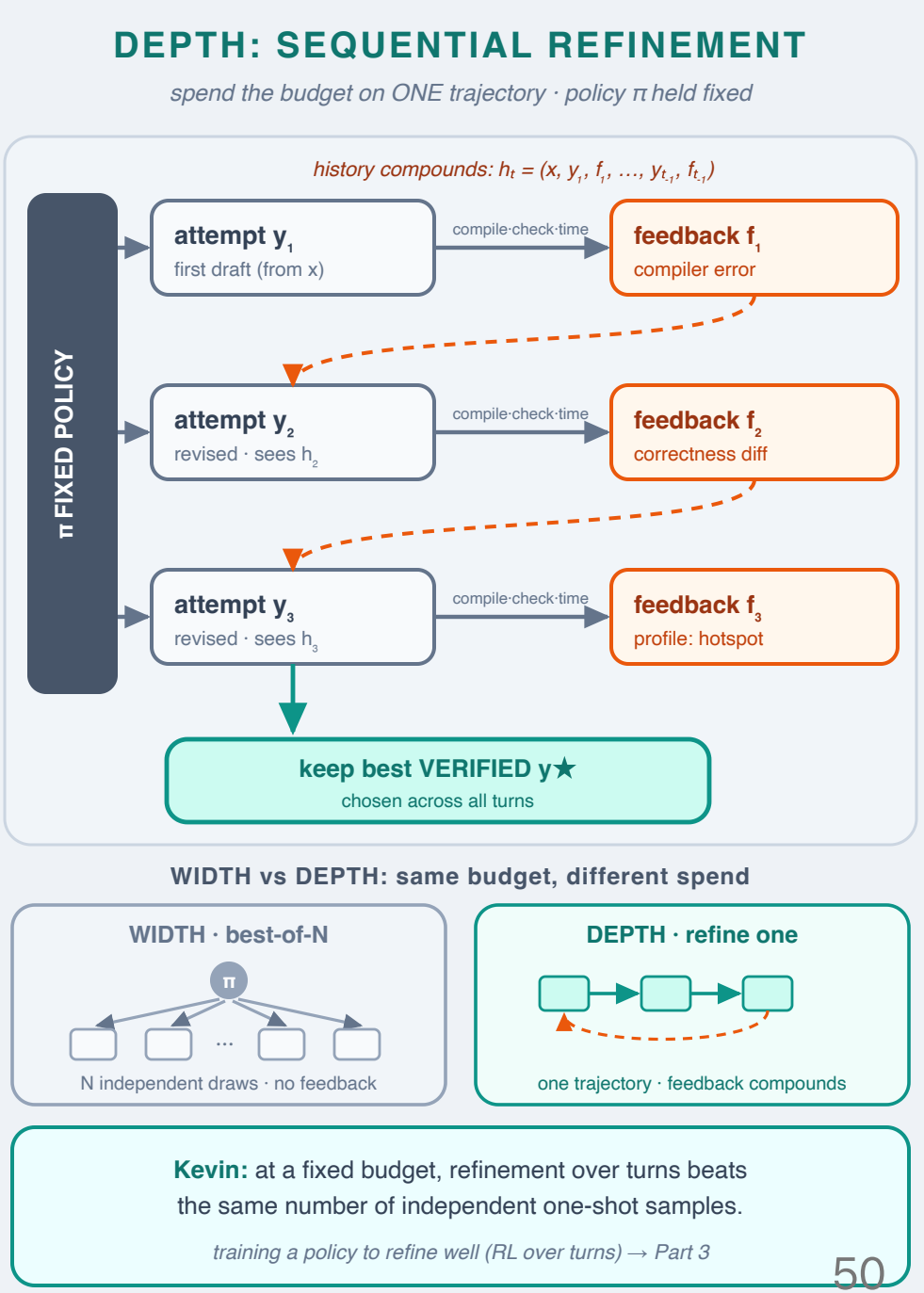
- **Genuine-kernel** (vs not-a-kernel): static linter + LLM judge + on-chip profile, flag if $\rho(y) = t_{\text{kernel}}/t_{\text{total}} < \rho_{\min}$
- **Robust timing** (vs faking speed): median-of-buckets, clip speedup at 2x:

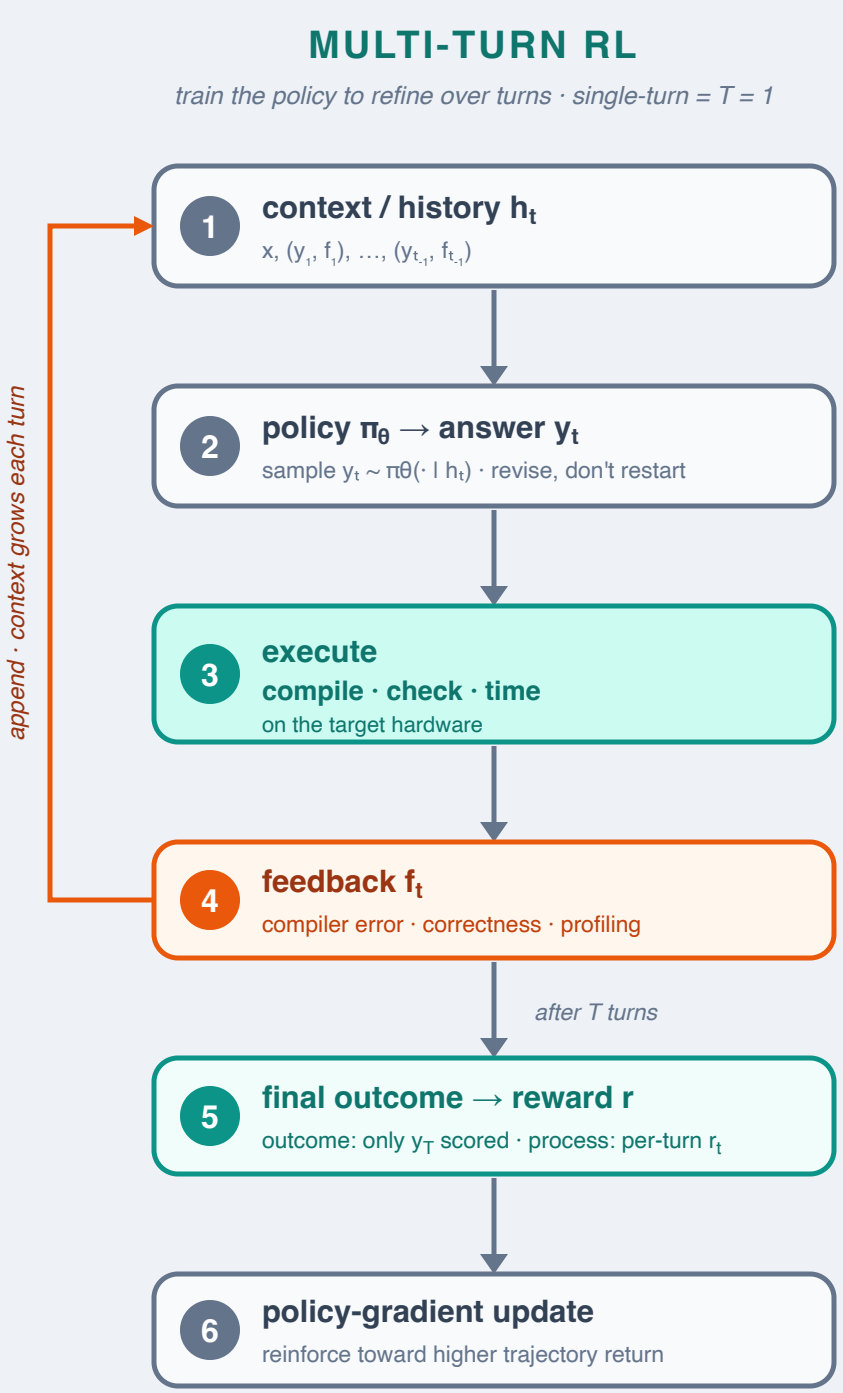
$$\hat{t}(y) = \text{median}_{1 \leq b \leq B}\left(\frac{1}{|b|}\sum_{r \in b} t_r\right), \quad \text{speedup}(y) = \frac{\hat{t}(\text{baseline})}{\hat{t}(y)}$$

- Any gate fails -> reward capped before speed is ever credited
- **Anti-cheating benchmarks** operationalize this: robust-kbench (decontamination collapses 3.13x -> 1.49x), FlashInfer-Bench (end-to-end on live serving traces)

Depth: sequential refinement with a fixed policy

- Width draws N INDEPENDENT samples; depth spends the budget on ONE trajectory
- Each turn reads the last attempt's compiler error, correctness diff, or profile, then revises: $y_t \sim \pi(\cdot \mid h_t)$
- Model stays FIXED: inference-time refinement, an agent whose tools are the compiler + hardware
- Reuses the evidence a fresh sample discards; keep the best VERIFIED candidate
- **Kevin:** at a fixed budget, refinement over turns > the same number of one-shot samples
- Training a policy to refine WELL (RL over turns) is a Part 3 topic



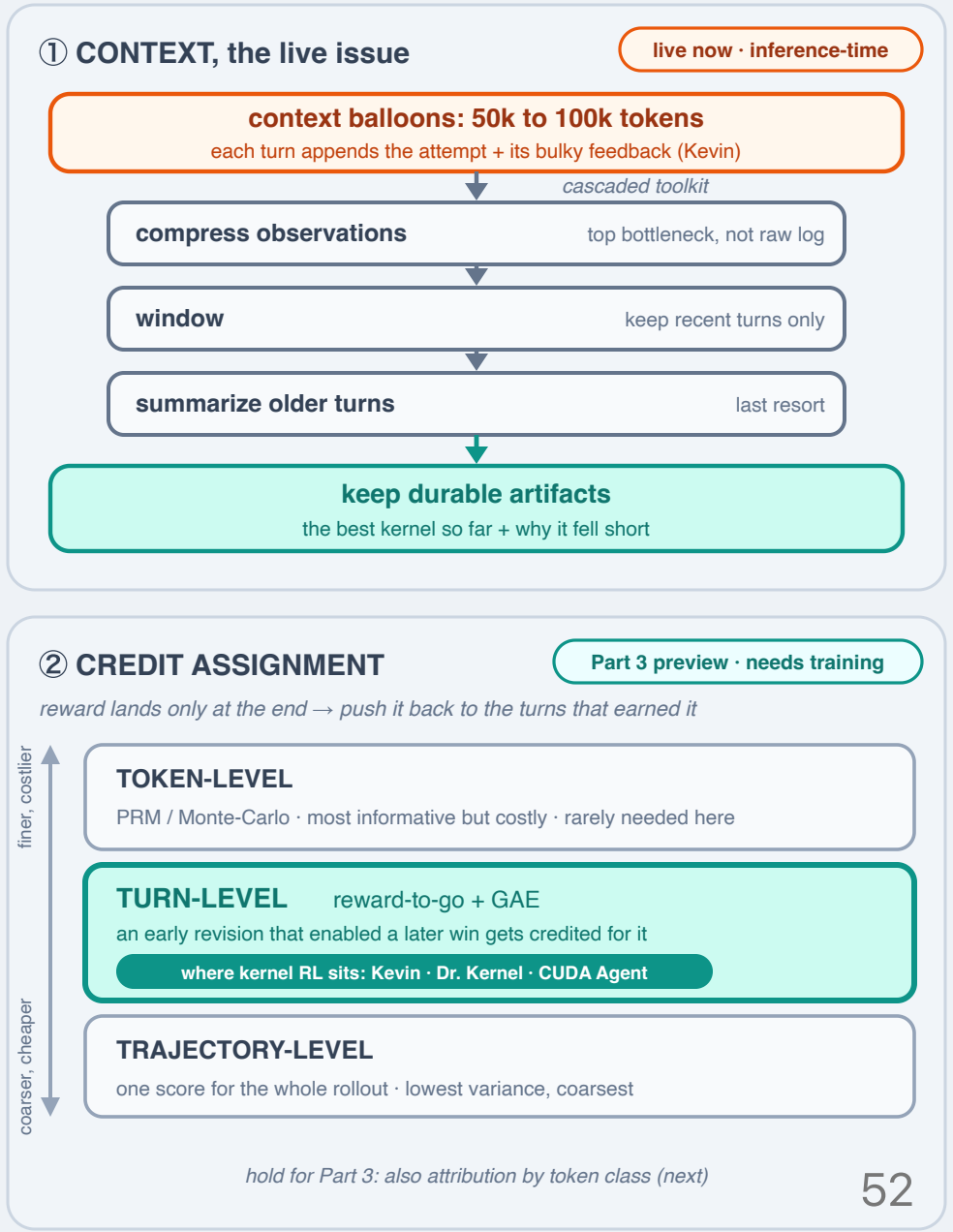


The refinement loop (and its training-time objective)

- History into turn t: $h_t = (x, y_1, f_1, \ldots, y_{t-1}, f_{t-1})$
- Sample $y_t \sim \pi(\cdot \mid h_t)$ -> execute -> feedback f_t -> append -> repeat
- Inference-time: run this loop, keep the best verified y_t; single-turn is T = 1
- **Part 3 counterpart:** TRAIN the policy on this loop, objective $J = \mathbb{E}[\sum_t r_t]$
- That reward lands at trajectory end (outcome) vs per-turn (process): a training choice

The live issue: context (and a Part 3 preview)

- Context bites AT INFERENCE TIME: 50-100k tokens in a few turns (Kevin)
- Cascaded toolkit: compress observations -> window -> summarize
- Keep durable artifacts: best kernel + why it fell short
- **Part 3 preview** (needs training): credit assignment across turns
- Ladder: token (PRM/MC, costly) -> **turn (reward-to-go + GAE)** -> trajectory
- Kernel RL systems sit at turn level (Kevin, Dr. Kernel, CUDA Agent)



ATTRIBUTION BY TOKEN CLASS

cut ONE response by role, not by time · TritonRL HRD

TWO ORTHOGONAL CUTS

by TIME (the credit ladder): token · turn · trajectory

by ROLE (HRD, here): planning span · coding span

one response y_t, split into two spans by token role

PLANNING span
the reasoning trace

CODING span
the emitted kernel

SPEEDUP reward
on the plan tokens
its own group-relative advantage

CORRECTNESS reward
on the code tokens
its own group-relative advantage

why this pairing, and not the obvious one?

the code is written conditioned on the plan, so charging the code for a slow plan would punish a faithful build of a bad strategy. correctness-only on the code decouples that; the speedup pressure lands on the plan, where the strategy is actually chosen.

both spans for speedup instead: match-or-beat 36% → 33%

the two spans co-evolve under one objective

$$J(\theta) = E[\ \alpha \cdot J_{plan} + J_{code}\ ]$$

$\alpha \approx 0.1$: the plan drifts slowly enough for the code policy to keep pace

it cuts by role, not by time, so it applies even to a single turn

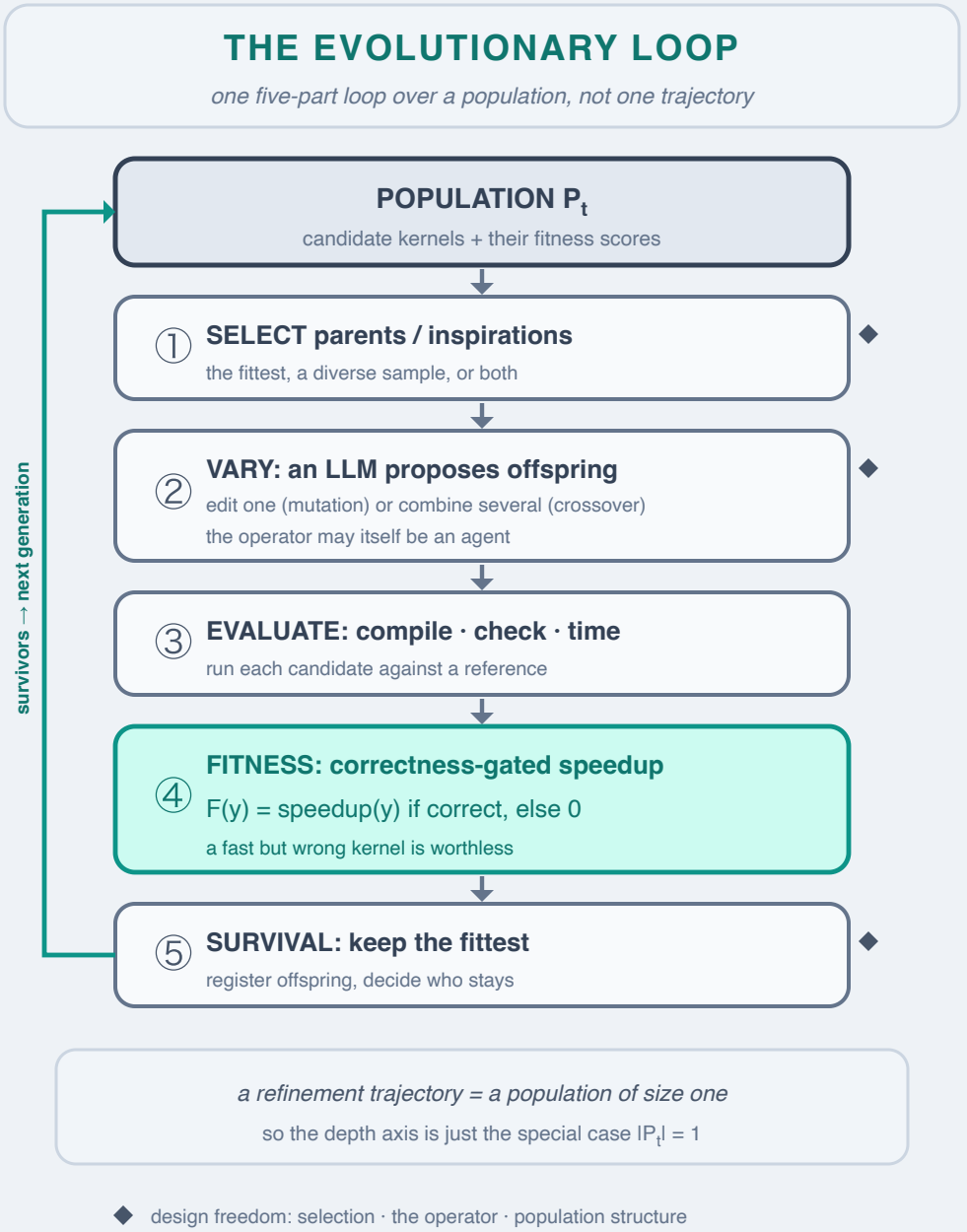
(TritonRL is single-turn)

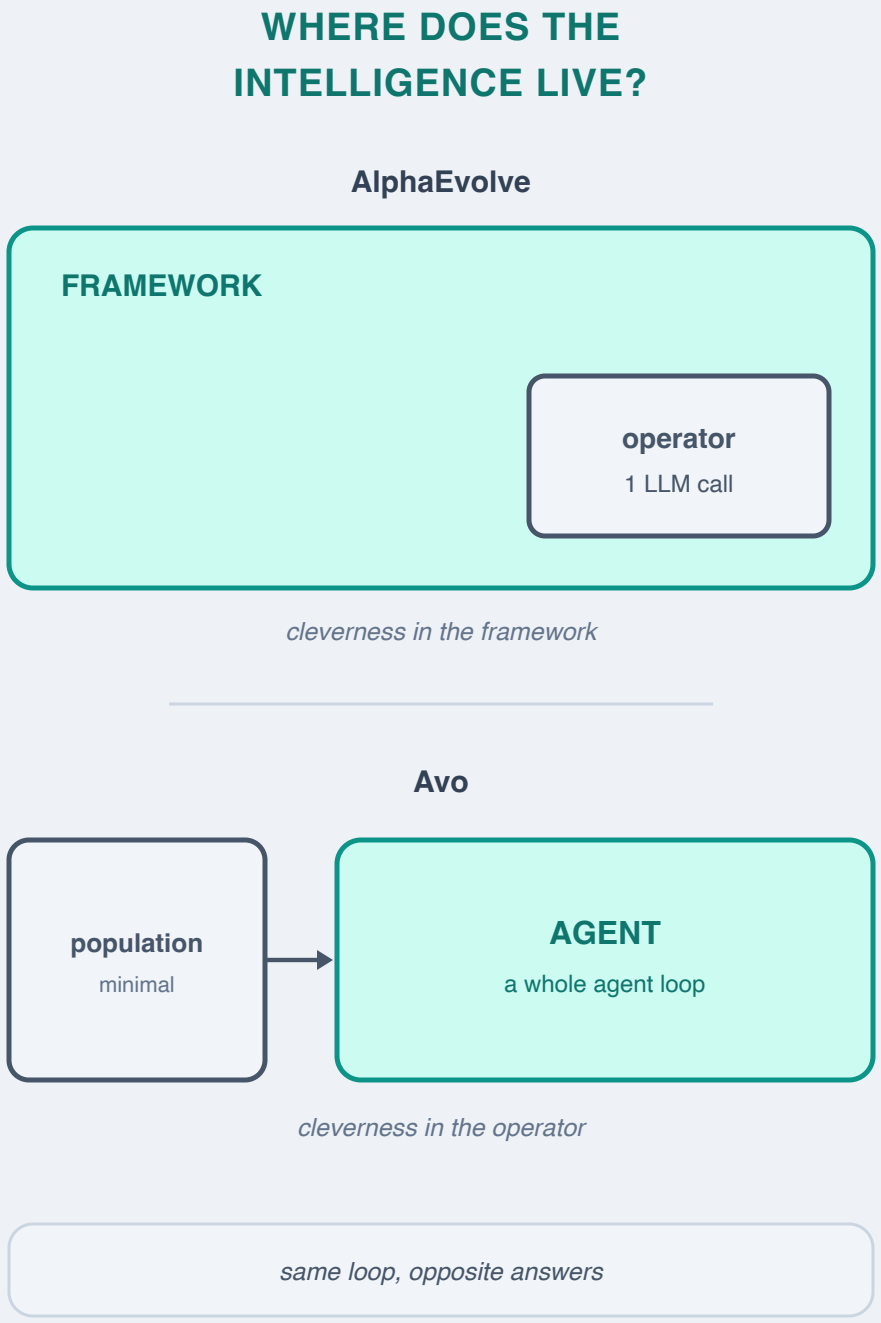
Part 3 preview: attribution by token class

- A second attribution axis, also a TRAINING-time idea (previewed here, built in Part 3)
- Prev slide's ladder split a trajectory BY TIME (token/turn/trajectory); this splits ONE response BY ROLE of its tokens
- Planning span (reasoning) + coding span (kernel), each its own advantage
- **Asymmetric:** plan -> speedup reward, code -> correctness reward
- Code is conditioned on plan; charging code for a slow plan punishes faithful code
- Both spans for speedup instead lowers match-or-beat (Part 3 quantifies); $J = \mathbb{E}[\alpha J_{\text{plan}} + J_{\text{code}}], \alpha \approx 0.1$

The evolutionary loop: a population, not one trajectory

- Depth refined ONE trajectory; evolution improves a **POPULATION** P_t over generations
- Five-part loop: select -> vary (LLM offspring) -> evaluate -> fitness -> survive
- Fitness = correctness-gated speedup: $F(y) = \text{speedup}(y)$ if correct, else 0
- A refinement trajectory = a population of size one
- Design freedom: selection, the operator, population structure





Where does the intelligence live? AlphaEvolve vs Avo

- Axis 1: how much work counts as one mutation = where cleverness sits
- **AlphaEvolve:** rich framework (program DB, MAP-Elites + islands, diffs) + 1-call operator
  - 48 scalar mults (4x4 complex matmul); 23% Pallas; 32% FlashAttention (XLA IR)
- **Avo:** minimal population, the operator IS an autonomous agent
  - beats FlashAttention-4 by 5.0-10.5% (causal attn, B200); MHA -> GQA in ~30 min

Beam / tree / evolutionary: one loop, two knobs

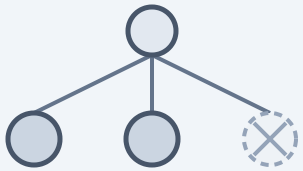
- Axis 2 (orthogonal to Axis 1): parents per operator + how selection prunes
- **Beam:** unary (mutation-only), strict top-B, no diversity
- **Tree (MCTS/UCT):** unary, adaptive node choice + explicit exploration bonus
- **Evolutionary:** n-ary (mutation AND crossover), diversity-preserving selection

SAME LOOP, TWO KNOBS

operator: parents per mutation · selection: how it prunes

BEAM SEARCH

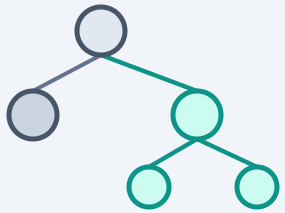
unary (1 parent → child) · strict top-B, no diversity



keep the best B

TREE SEARCH (MCTS / UCT)

unary · adaptive node choice + exploration bonus

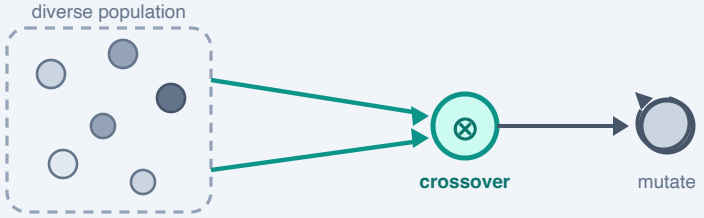


budget flows to the promising subtree

the under-tried branch is not pruned

EVOLUTIONARY

n-ary: mutation AND crossover · diversity-preserving



each row relaxes a constraint of the one above

one population loop, two knobs set differently

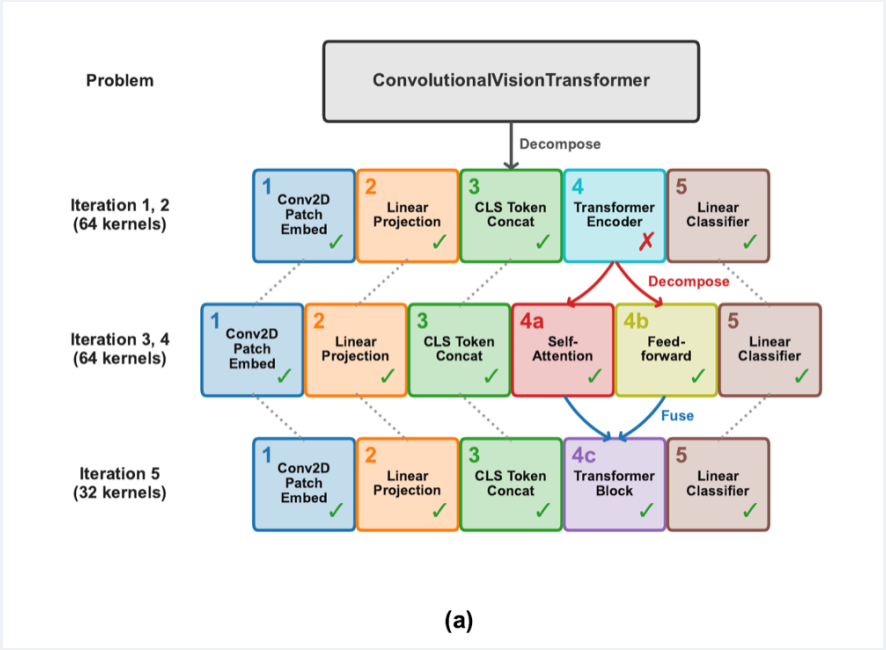
Kernel-specific systems, read through the two axes

- **Autocomp:** beam ($B = 6$) + editable optimization menu; menu-dropout for diversity; 5.6x over Gemmini, 1.9x over expert Trainium
- **AccelOpt:** beam + self-improving memory (NKI); distills each round's rewrites (even negatives) into reusable strategies; 49% -> 61% peak Trn1; matched Claude Sonnet 4 at 26x lower cost
- **Astra:** role-decomposed agents (test / profile / plan / code) over a shared log; 1.32x vs 1.08x for the monolith (SGLang CUDA)
- **KernelEvolve @ Meta:** formalizes the whole family as a tuple $(F, \pi_{\text{sel}}, O, \tau)$
  - correctness-gated fitness $F(v) = t_{\text{pytorch}}/t_{\text{triton}}$, with $F(v) = 0$ if v fails to compile or is incorrect
  - selection π_{sel} instantiates greedy / MCTS-UCT / evolutionary; RAG-steered universal operator O; termination τ

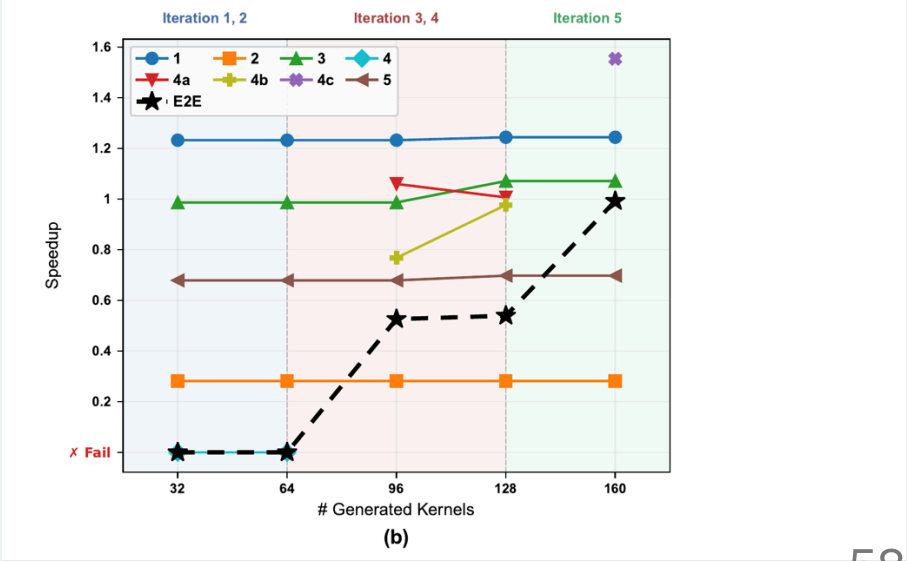
Search over the decomposition, not just the code

- Monolithic kernels: inefficient (re-optimize everything) + opaque (which part is wrong?)
- Decompose -> generate -> compose: **KernelFalcon** established the recipe
- **MKEvolve:** evolve the decomposition topology + programmatic composition
- BF16: a correct kernel's rounding ~1e-2 can exceed a bug's ~1e-3
- Per-operator calibrated tolerance catches subtle BF16 bugs; 15-35% fewer tokens; 10.7x-44.3x over torch.compile

MKEVOLVE ON CONV-VIT (KernelBench L3)



(a) decomposition topology evolves: subproblem 4 splits, then fuses



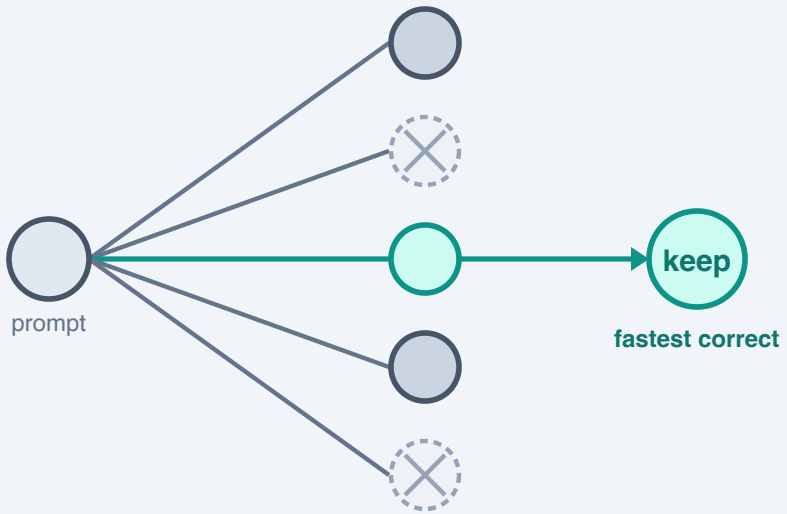
(b) E2E speedup climbs as the structure is refined

PARALLEL SAMPLING, BEST-OF-N

draw N kernels, keep the fastest correct one

BEST-OF-N

one prompt → N independent draws
Pass@N asks: did any of them pass?



failures and slow draws are discarded;
only the fastest passing kernel survives

Width: parallel sampling and best-of-N

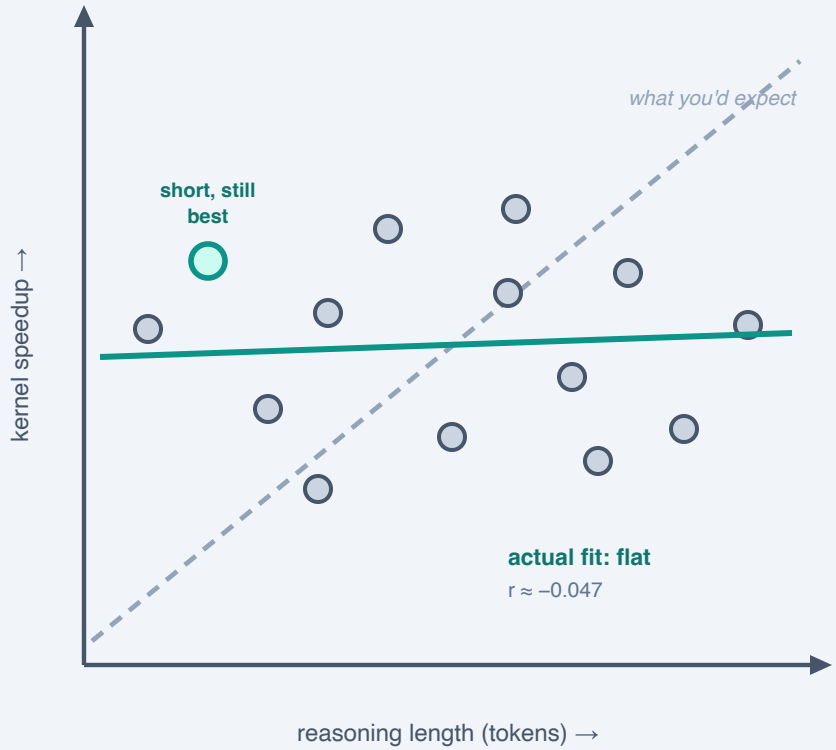
- Best-of-N: draw N kernels, keep fastest correct one
- **MultiKernelBench:** greedy 164/855 -> N=5 lifts Sonnet 4 to 200 (CUDA Pass@5 55.8%)
- **ConCuR on KernelBench:** Pass@1 -> Pass@10 = 58% -> 91% executable
- Uneven returns: AscendC / Pallas stay single-digit however sampled
- Width raises coverage, not the ceiling -> why depth and search exist

The reasoning-length surprise

- Longer chain-of-thought does NOT mean better kernels
- **ConCuR:** shorter correct traces -> higher accuracy
- Speedup ~uncorrelated with length (Pearson $r \approx -0.047$)
- 4,892 shortest-correct-and-fast examples rival far larger models
- Length as a difficulty signal: Easy <4000, Medium 4000-8500, Hard >8500 tokens
- Compute-optimal: widen and deepen only where difficulty says it pays

LONGER REASONING ≠ BETTER KERNELS

speedup is ~uncorrelated with chain-of-thought length

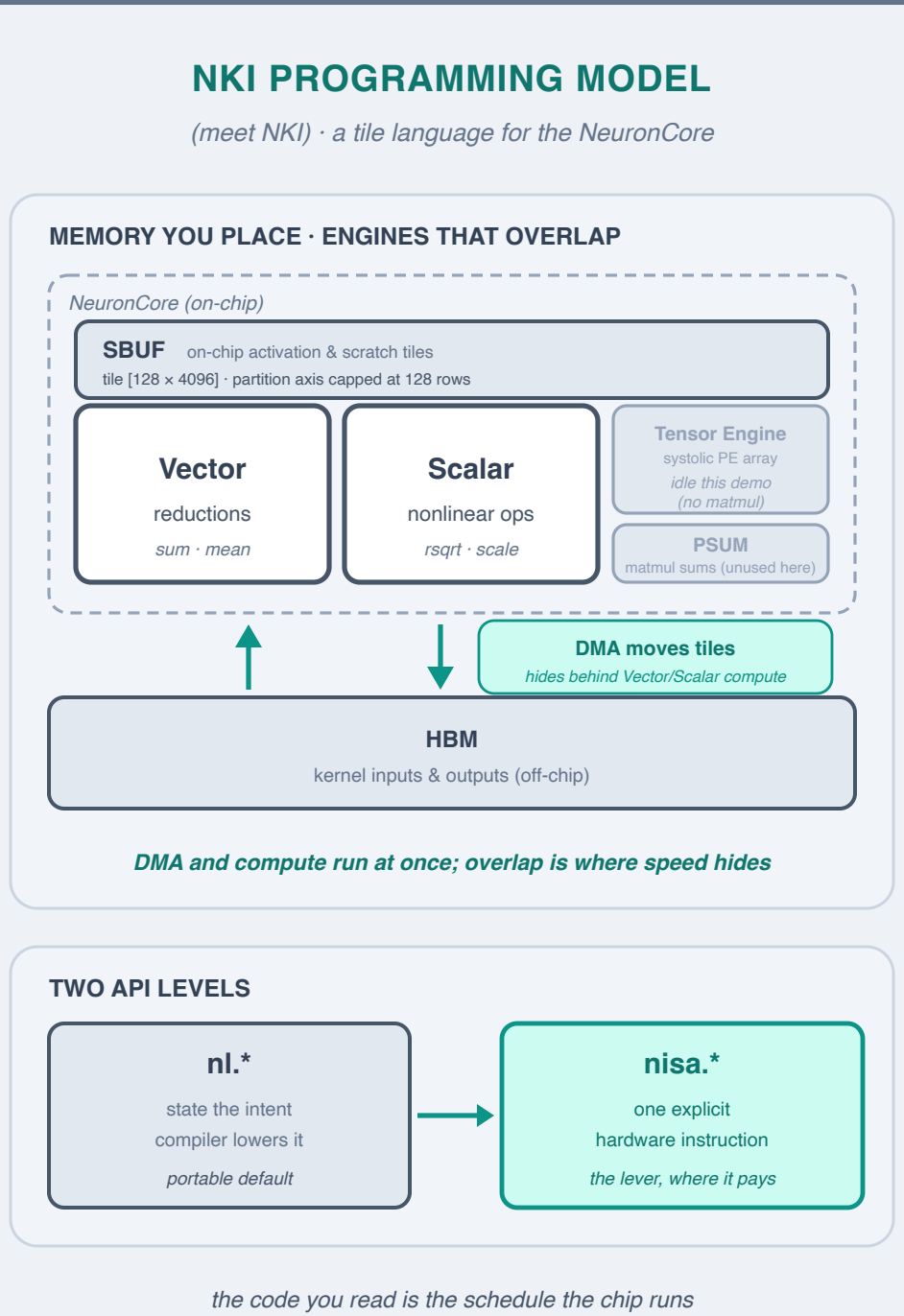


thinking longer does not make the kernel faster

length is a difficulty signal, not a quality lever

Verification: the enabler and the bottleneck

- Every axis rests on a verifier where best-of-N / fittest really IS best
- Search scales contact with the verifier -> any gap is found and exploited faster
- Reward hacking is the dominant failure: strip the functionality check and AutoTriton "correctness" jumps 57% -> 87%
- **KernelBench:** core protocol, correctness within tolerance + Fast_p over `torch.compile`
- **MultiKernelBench:** 285 tasks / CUDA, AscendC, Pallas -> CUDA gains do not transfer
- **robust-kbench:** closes evaluator loopholes -> avg speedup 3.13x -> 1.49x once exploits removed
- **FlashInfer-Bench:** real serving traces, live SGLang / vLLM substitution



From the verifier to a real kernel: meet NKI

- **NKI**, a tile-oriented language for the NeuronCore: you place memory by hand and map work to concurrent engines

Resource	Role in this demo
HBM	Kernel inputs and returned outputs (off-chip)
SBUF	Activation, weight, and scratch tiles (on-chip)
PSUM	Accumulations for matmul
DMA	Moves tiles between HBM and on-chip memory
Vector, Scalar	Reductions, tensor ops, nonlinear ops

- Engines overlap, so how much **DMA hides behind compute** is the whole game; partition axis caps at **128 rows** -> `[1024, 4096]` streams as **8 tiles**
- Two API levels (right): `nl.*` states intent (portable default); `nisa.*` = one explicit instruction, the lever the profile motivates

Setup: the op and the context to supply

- Demo op: **BF16 residual-add + weighted RMSNorm** over `[1024, 4096]`, one Trn2 NeuronCore (steady-state)

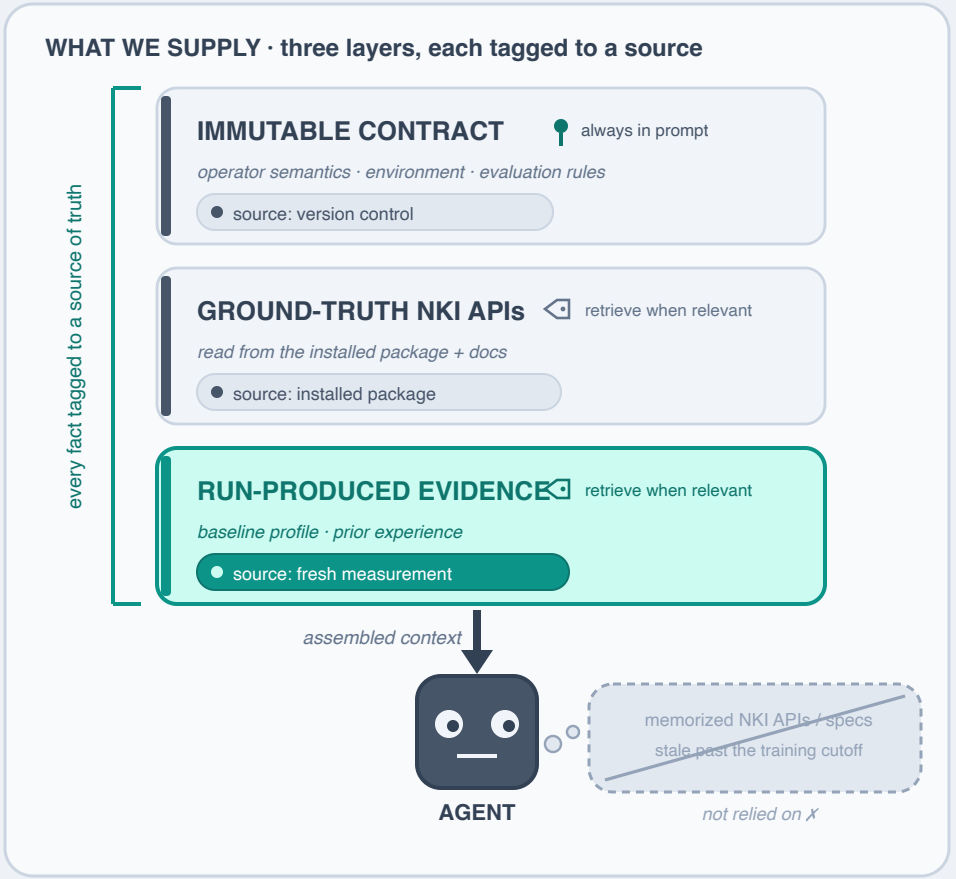
$$z_{r,h} = x_{r,h} + a_{r,h}$$

$$y_{r,h} = \mathrm{BF16}\left(\frac{z_{r,h}}{\sqrt{\frac{1}{H}\sum_j z_{r,j}^2 + 10^{-6}}}\, w_h \right)$$

- The agent reads your repo, but its memorized **NKI APIs, Trainium specs, and signatures are stale** past its cutoff
- Fix: a context package where **every fact is tagged to a source of truth** (version control or a fresh measurement), never recall
- That package = immutable contract (**semantics, environment, evaluation**) + ground-truth APIs + run-produced evidence
- Keep the non-negotiables always visible; let it **retrieve an API page or trace slice only when relevant**

THE CONTEXT PACKAGE

every fact tagged to a source of truth, never recall



Contract (semantics, environment, evaluation) + ground-truth APIs + run-produced evidence

Non-negotiables always visible; retrieve an API page or trace slice only when relevant

`version control · installed package · fresh measurement`

supply what the model cannot trust from memory, tagged to where it came from

The executable reference: closes every fork

Runnable Torch = unambiguous spec + oracle: the same function that documents the op also generates the ground truth the harness checks against.

```
def residual_rms_norm_reference(x, residual, weight):
    z = x.float() + residual.float()
    inverse_rms = torch.rsqrt(
        z.square().mean(dim=1, keepdim=True) + 1e-6
    )
    return (z * inverse_rms * weight.float()).to(torch.bfloat16)
```

$$z_{r,h} = \mathrm{FP32}(x_{r,h}) + \mathrm{FP32}(a_{r,h})$$
$$m_r = \tfrac{1}{H}\sum_j z_{r,j}^2, \quad s_r = (m_r + 10^{-6})^{-1/2}$$
$$y_{r,h} = \mathrm{BF16}(z_{r,h}\, s_r\ \mathrm{FP32}(w_h))$$

Pins the math **and** the interface; leaves tiling, buffering, and scheduling free.

Property	Fixed value
Inputs	activations `x`, residual `a`, weight `w`
Output	one tensor `y`; no aliases / side effects
Shapes	`x,a,y`: `[1024,4096]`; `w`: `[4096]`
Layout	contiguous row-major HBM
Dtypes	BF16 I/O; FP32 arithmetic path
Reduction	hidden axis, independently per row
Epsilon	`1e-6`, inside the reciprocal sqrt

THE READABLE nl.* BASELINE

correct and auditable, before any tuning

BASELINE nl.* KERNEL · 8-TILE STREAM

```
@nki.jit
def residual_rms_norm_kernel(x, residual, weight):
    assert x.shape == residual.shape == (TOKENS, HIDDEN)
    assert x.dtype == nl.bfloat16 # BF16 in / out
    out = nl.ndarray(x.shape, dtype=x.dtype, buffer=nl.shared_hbm)
    weight_row = nl.load(weight.reshape((1, HIDDEN)))
    weight_tile = nl.broadcast_to(weight_row, (TILE_TOKENS, HIDDEN))
    for tile_idx in range(NUM_TILES): # 8 tiles of [128, 4096]
        start = tile_idx * TILE_TOKENS
        stop = start + TILE_TOKENS
        x_tile = nl.load(x[start:stop, :])
        res_tile = nl.load(residual[start:stop, :])
        summed = nl.add(x_tile, res_tile, dtype=nl.float32) # FP32
        mean_sq = nl.mean(nl.square(summed), axis=1, keepdims=True)
        inv_rms = nl.rsqrt(nl.add(mean_sq, EPS))
        inv_rms = nl.broadcast_to(inv_rms, (TILE_TOKENS, HIDDEN))
        normalized = nl.multiply(summed, inv_rms, dtype=nl.float32)
        y_tile = nl.multiply(normalized, weight_tile, dtype=x.dtype)
        nl.store(out[start:stop, :], value=y_tile)
    return out
```

TOKENS 1024 · HIDDEN 4096 · TILE_TOKENS 128 · NUM_TILES 8

8 tiles of [128, 4096] stream through SBUF; FP32 math, BF16 I/O

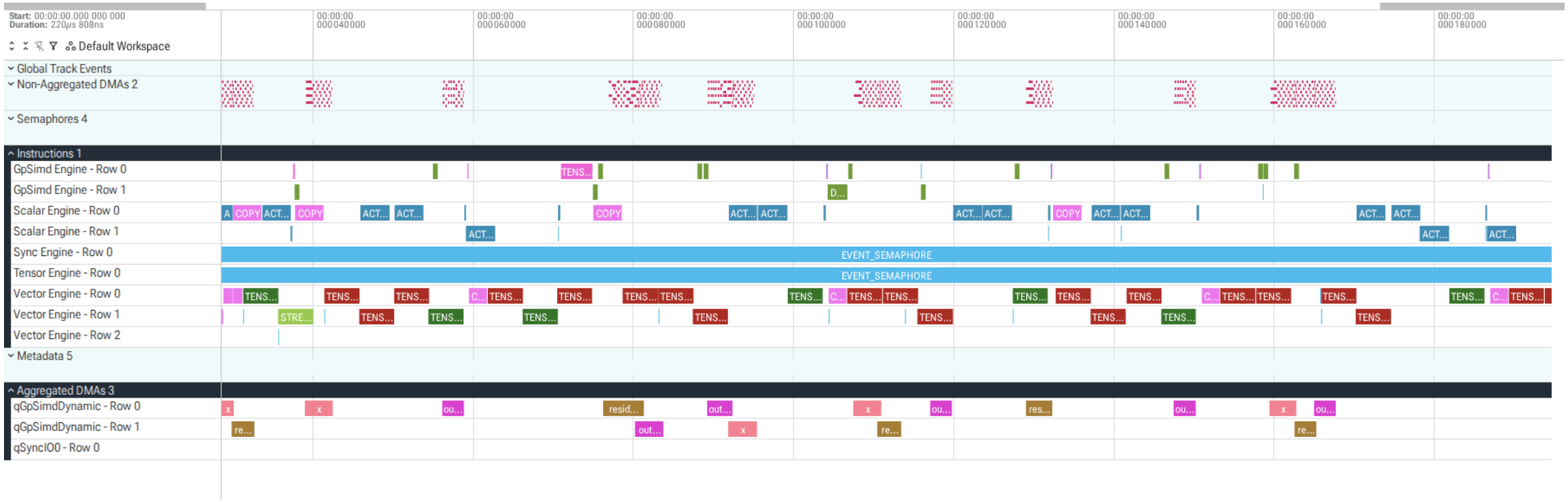
assertions fail loudly; every intermediate stays on the device

accept a correct, auditable baseline before tuning anything

Stage 1: generate correct, gate before profiling

- Device boundary = correctness (whole op in-kernel)
- Host-compute + copy-out = the reward-hack (cf. Module 2)
- Torch reference = ambiguity resolver + oracle
- Accept simple `nl.*` baseline; don't pre-tune unmeasured bottleneck
- Gate BEFORE profiling: structural + allclose(2^{-7}, 1e-2) + max-abs + mean-abs + cosine 0.9999
- All checks run on adversarial inputs

Stage 2a: profile baseline, totals hide structure



- Steady state: profile the **2nd execution**; the warm run pays one-time load and setup
- Read against **218.666 us** latency: Vector active **162.230 us** dominates (DMA 76.154, Scalar 78.340)
- But totals hide structure: one engine stacks overlapping slices into rows, DMA overlaps compute, and long **EVENT_SEMAPHORE** bars are idle waits, not work

Stage 2b: roofline analysis, the operator is memory-bound

Classify from the **fixed interface**, not the profile ($N = RH = 4{,}194{,}304$ elements; BF16 = 2 B):

$$Q_{\min} = 6N + 2H = 25{,}174{,}016 \text{ bytes}, \qquad F_{\text{useful}} = 5N + R = 20{,}972{,}544 \text{ FLOPs}$$

$$I = \frac{F_{\text{useful}}}{Q_{\min}} \approx 0.833 \text{ FLOP/byte}, \qquad I_{\text{ridge}} = \frac{1.0 \text{ TFLOP/s}}{368 \text{ GB/s}} \approx 2.72 \text{ FLOP/byte}$$

$$I \approx 0.83 \;<\; I_{\text{ridge}} \approx 2.72 \implies \text{memory leg of the roofline}$$

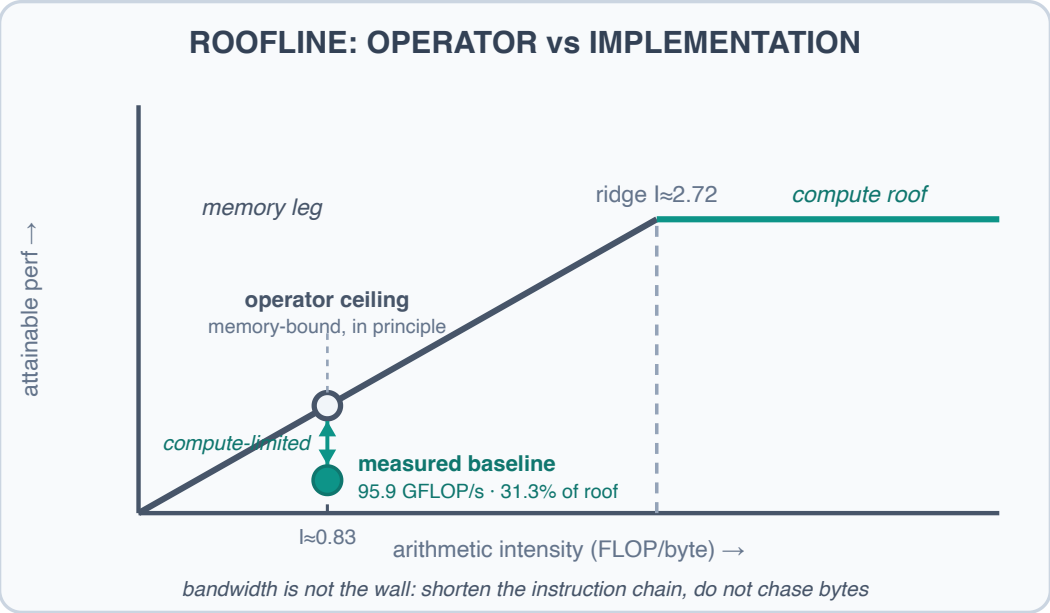
- $Q_{\min}$: the HBM bytes the ABI **forces** (read x + residual, load weight once, write y); no kernel moves fewer
- F_{useful}: FLOPs counted straight off the operator (add, square, row-reduce, scale, weight); `rsqrt` omitted, ~0.1%, cannot flip the class
- I = useful FLOPs per byte moved, a **property of the operator** no rewrite can change
- I_{ridge} = peak compute / peak bandwidth (1.0 TFLOP/s over 16 DMA engines x 23 B/ns); the memory-to-compute crossover
- $I < I_{\text{ridge}}$ -> best possible implementation is **memory-bound**: a claim about the operator, not yet about our code

Stage 2c: memory-bound operator, compute-bound implementation

$$B_{\mathrm{DMA,max}} = 16 \times 23\,\mathrm{B/ns} = 368\,\mathrm{GB/s},\quad P_{\mathrm{roof}}(I) = I\,B_{\mathrm{DMA,max}} \approx 306.6\,\mathrm{GFLOP/s},\quad t_{\mathrm{HBM,min}} = \frac{Q_{\min}}{B_{\mathrm{DMA,max}}} \approx 68.41\,\mu\mathrm{s}$$

These ground the table: roof fraction = 95.9 / 306.6 ≈ 31.3%; DMA efficiency = 330.6 / 368 ≈ 89.8%.

Baseline-derived metric	Value
Useful throughput	95.9 GFLOP/s
Fraction of roof	31.3%
BW while DMA active	330.6 GB/s
DMA ceiling efficiency	89.8%
Effective BW (e2e)	115.1 GB/s
Runtime / HBM-only min	3.20x
Compute / DMA union	190.251 / 69.841 us



- Honest measure = interval **UNION** (count each moment once); **80.7%** of DMA already hidden behind compute
- Roofline said memory-bound, but the measured baseline is **compute/dependency-limited** (compute union ~3x the DMA union)
- Transfers near-optimal when active (**330.6 GB/s = 89.8%** of the 368 ceiling); idle DMA time drags end-to-end bandwidth to 115.1
- The distinction: **memory-bound operator, compute-bound implementation**
- Direction: shorten the instruction chain, do not chase bytes

Hand evidence back as a falsifiable hypothesis

- Hand back what the agent can't re-derive: baseline source, summary JSON, trace, roofline, current APIs
- Six-step prompt forces reasoning before editing (reconcile, quantify unions, find longest avoidable chain)
- **Step 5:** ONE falsifiable hypothesis -> counters to CHANGE + counters to HOLD
- **Step 6:** name the correctness cases most likely to reject the change
- CHANGE: Vector active, compute union, latency down
- HOLD: HBM bytes, DMA active -> the anti-reward-hacking guardrail

THE PROMPT: OPTIMIZE AGAINST MEASUREMENT

give the evidence back to the agent

AGENT PROMPT · SIX-STEP CONTRACT

```
Optimize this kernel without changing its equations, inputs, output, shape,
layout, dtype path, tests, benchmark, or profiler command.

Before editing:
1. reconcile the ideal roofline classification with the measured bottleneck;
2. quantify useful DMA, useful compute, and their overlap from interval unions;
3. identify the longest avoidable instruction chain in the trace;
4. inspect the installed nki.language and nki.isa APIs for a way to combine
adjacent work or remove an intermediate dependency;
5. state one falsifiable hypothesis, including counters expected to change
and counters expected to remain constant; and
6. name the correctness cases most likely to reject the change.

Implement one candidate. Run every correctness gate and recapture the same
workload. Keep the change only if the measured counters support the hypothesis.
```

Steps 1-4 read the evidence; the last line binds every change to the gates.

steps 5 and 6 turn a request into a falsifiable experiment

THE ONE PROFILE-DERIVED CHANGE

fuse square + row-reduce into one nisa.activation

residual_rms_norm_fused_kernel · per-tile loop

```
@nki.jit
def residual_rms_norm_fused_kernel(x, residual, weight):
    # asserts, out = nl.ndarray(...), weight_tile = broadcast
for tile_idx in range(NUM_TILES):
    nisa.dma_copy(dst=x_tile, src=x[start:stop, :])
    nisa.dma_copy(dst=residual_tile, src=residual[start:stop, :])
    nisa.tensor_tensor(dst=summed, data1=x_tile,
        data2=residual_tile, op=nl.add)
    nisa.activation(dst=square_scratch, op=nl.square, # THE CHANGE
        data=summed, reduce_op=nl.add,
        reduce_res=sum_square,
        reduce_cmd=nisa.reduce_cmd.reset_reduce)
    nisa.activation(dst=sum_square, op=nl.rsqrt,
        data=sum_square, scale=1.0/HIDDEN, bias=EPS) # 1/H + eps
    nisa.tensor_scalar(dst=summed, data=summed,
        op0=nl.multiply, operand0=sum_square) # normalize
    nisa.tensor_tensor(dst=x_tile, data1=summed,
        data2=weight_tile, op=nl.multiply)
    nisa.dma_copy(dst=out[start:stop, :], src=x_tile)
return out
```

op=nl.square squares; reduce_op=nl.add + reduce_res write the row sum, collapsing two dependent Vector instructions on the critical path into one

`Vector active 162.2 -> 80.2 us · reset_reduce keeps FP32, per-tile`

one fused instruction is the whole optimization

Stage 3: one change, re-verify, recapture

- One change: fuse square + row-reduction into a single `nisa.activation`
- Collapses two dependent Vector instructions on the critical path into one
- Re-gate FIRST: fused passes; error shifts (max_abs 0.03125, cosine ~0.99999996), within budget
- **218.666 us -> 119.547 us = 1.83x** (45.3% latency cut)
- HBM bytes unchanged; Vector active 162.230 -> 80.167 us; compute union 190.251 -> 99.922 us
- Signature matches the hypothesis; still compute-limited

Recapture and compare the same evidence

Profile summary	Baseline	Agent candidate
Representative latency	218.666 us	**119.547 us**
3-capture latency range	216.276–218.846	119.547–120.057
HBM read	16.008 MiB	16.008 MiB
HBM write	8.000 MiB	8.000 MiB
DMA transfer time	76.154 us	77.454 us
Vector Engine active	162.230 us	**80.167 us**
Scalar Engine active	78.340 us	68.926 us

- Same capture, summary query, and interval query, re-run on a fresh artifact
- **1.83x** faster (45.3% latency cut) from **one** fused instruction
- HBM read/write and DMA time **unchanged** -> no bytes were chased
- Vector active falls **162.230 -> 80.167 us**: the win came from compute
- Exactly the counter signature the hypothesis pre-committed to

Integrate the kernel, and the reusable loop

- Standalone acceptance is not framework acceptance: re-test at every call site
- PyTorch/XLA: lowers as a custom op, can drift via cross-boundary fusion
- NKIPy custom-op: opaque node, can break on boundary/aliasing (this kernel reuses scratch)
- Re-apply the same predeclared BF16 budget at each site
- Reusable loop: agent gives hypotheses + code; reference/device/profiler give evidence
- The verifier, not the agent's claim, decides what counts

INTEGRATE INTO THE FRAMEWORK

shown: the PyTorch/XLA site (one of two call sites)

PYTORCH / XLA INTEGRATION CHECK

```
import torch
import torch_xla
from rmsnorm_agent_lab import EPS, residual_rms_norm_fused_kernel

x_cpu = torch.randn((1024, 4096), dtype=torch.bfloat16)
residual_cpu = torch.randn_like(x_cpu)
weight_cpu = torch.empty((4096,), dtype=torch.bfloat16).uniform_(0.75, 1.25)

# host reference in FP32, cast to BF16 only at the end
z = x_cpu.float() + residual_cpu.float()
reference = (z * torch.rsqrt(z.square().mean(dim=1, keepdim=True) + EPS)
* weight_cpu.float()).bfloat16()

device = torch_xla.device()
# .to(device) is what forces the @nki.jit lowering into the graph
actual = residual_rms_norm_fused_kernel(
x_cpu.to(device), residual_cpu.to(device), weight_cpu.to(device))
torch_xla.sync(wait=True) # run the lazy graph, block on result

assert torch.allclose(actual.cpu(), reference, atol=2**-7, rtol=1e-2)
```

The check mirrors the standalone oracle, so a framework-level regression cannot hide behind cross-boundary fusion or a dtype coerced at the edge

`atol=2**-7, rtol=1e-2` · the same predeclared BF16 budget

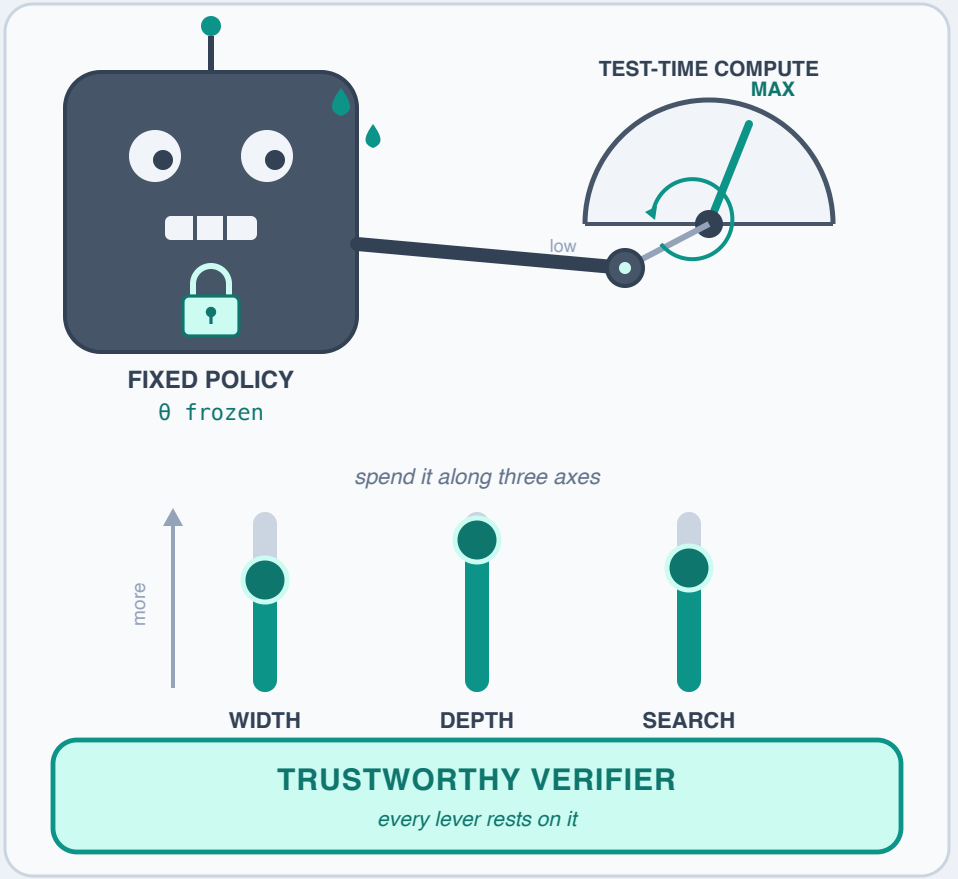
the NKIPy custom-op site re-runs this same gate (scratch reuse can alias)

Part 2 takeaways: run a fixed policy harder

- Hold the model fixed; spend test-time compute along width, depth, structured search
- Fixed budget: sequential refinement beats the same number of independent samples
- Search over decomposition beats monolithic (MKEvolve, AccelOpt)
- Longer reasoning is not better: length is a difficulty signal, not a quality lever
- Every lever rests on a **trustworthy, cheating-resistant verifier**

RUN A FIXED POLICY HARDER

same weights, more test-time compute



Three axes to spend test-time compute: width, depth, structured search

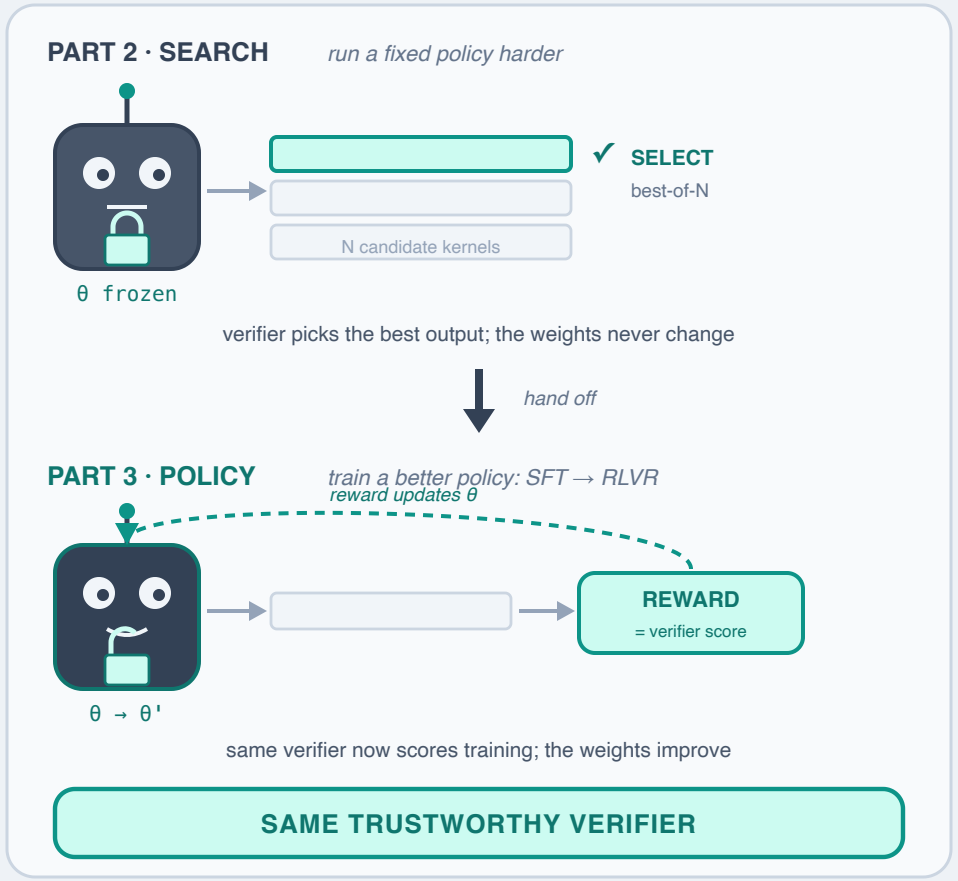
Fixed budget: depth (sequential refinement) beats width (independent samples)

`weights frozen · only compute-per-query goes up`

same model, cranked harder; the verifier is the slab it all rests on

FROM SEARCH TO POLICY

the verifier: selector becomes training signal



Part 2 squeezed a fixed model; Part 3 builds a better one

The reward RLVR optimizes is that verifier, now a training signal

`no weights changed → weights updated (SFT, RLVR)`

same verifier: from picking the best output to rewarding a better policy

Bridge to Part 3: from search to policy

- Part 2: ran a fixed policy harder (inference-time scaling, no weights changed)
- Part 3: train a better policy via post-training (SFT -> RLVR)
- **Same reward:** the trustworthy verifier becomes the training signal
- Handoff: squeeze a fixed model -> build a better one, rewarded by the same verifier

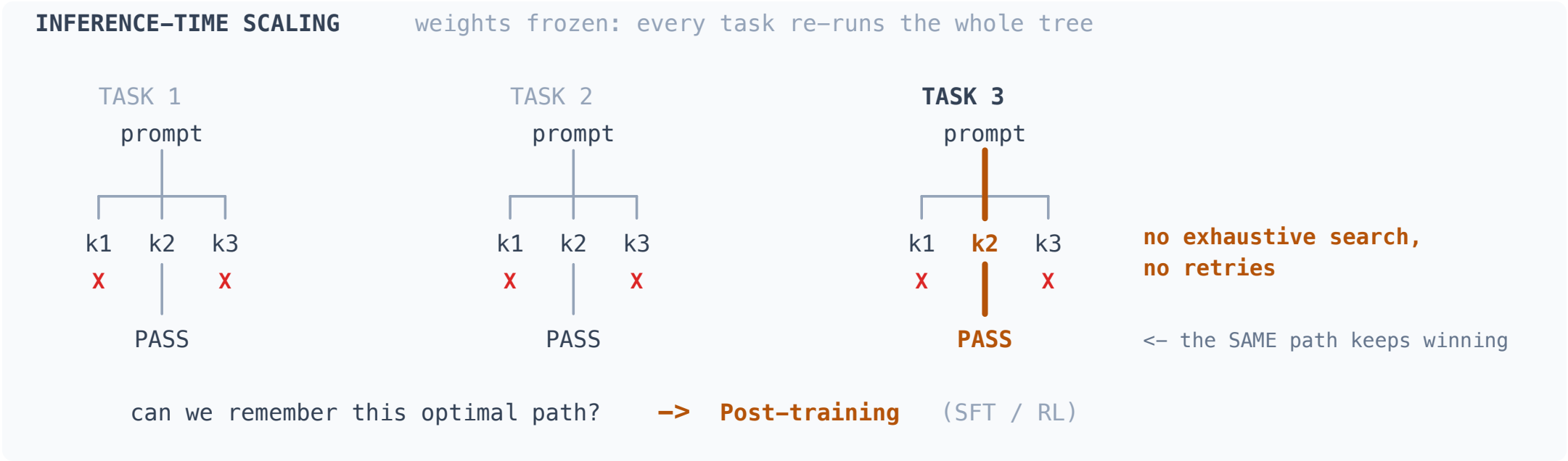
Part 3: Post-Training LLMs for Kernel Generation

Teaching LLMs to Write System Kernels for AI Accelerators

KDD 2026 Tutorial

From Inference-Time Search to Post-Training

- **Inference-time scaling**
  - Every task re-runs the full search
  - The same winning path is rediscovered from scratch each time
- Can we bake that path into the model itself? **Post-Training**
  - Updating the weights turns a generalist into a specialist that no longer has to exhaustively search

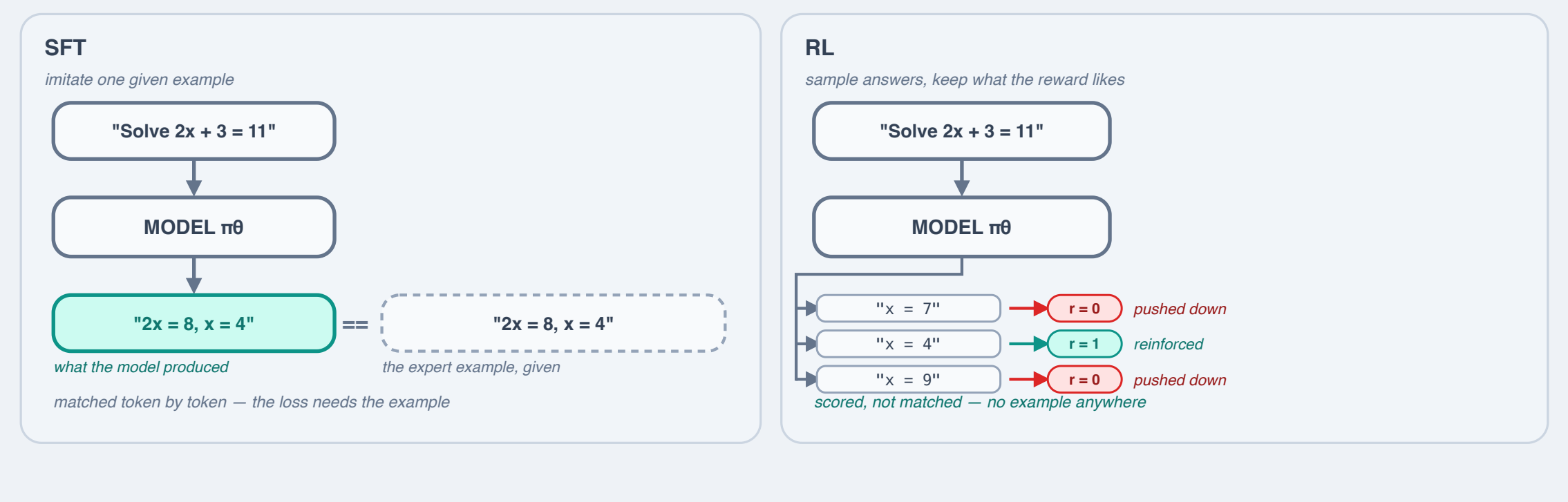


Part 3: Post-Training LLMs for Kernel Generation

1. **Preliminaries**
   - **Supervised fine-tuning (SFT):** imitate curated expert examples
   - **Reinforcement learning (RL):** learn from rewards through trial and error
2. **Turning a generalist LLM into a kernel specialist**
   - **SFT for kernels**: ground kernel syntax/APIs
   - **RL for kernels**: specialize for a target hardware
   - **Multi-turn RL:** refine kernels over profiler feedback

Preliminaries: How to Fine-Tune LLMs?

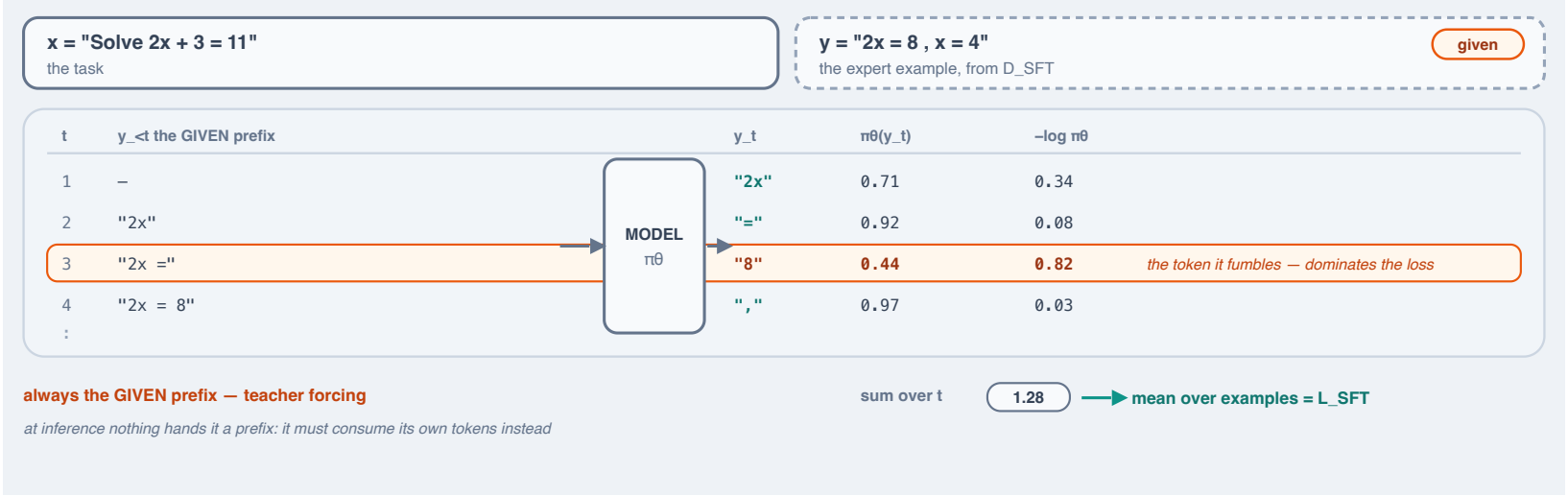
- **Supervised fine-tuning (SFT):** imitate curated expert examples
- **Reinforcement learning (RL):** learn from rewards through trial and error



Supervised Fine-Tuning (SFT)

- **Data:** $\mathcal{D}_{SFT} = \{(x^{(i)}, y^{(i)})\}$ - a task and one expert example per pair
- **Policy (language model):** $\pi_\theta(y \mid x) = \prod_t \pi_\theta(y_t \mid x, y_{<t})$ - autoregressive generation
- **Objective:** make each example token as likely as possible - minimize its negative log-likelihood

$$\mathcal{L}_{\text{SFT}}(\theta) = -\frac{1}{N} \sum_i \sum_t \log \pi_\theta \left(y_t^{(i)} \mid x^{(i)}, y_{<t}^{(i)} \right)$$



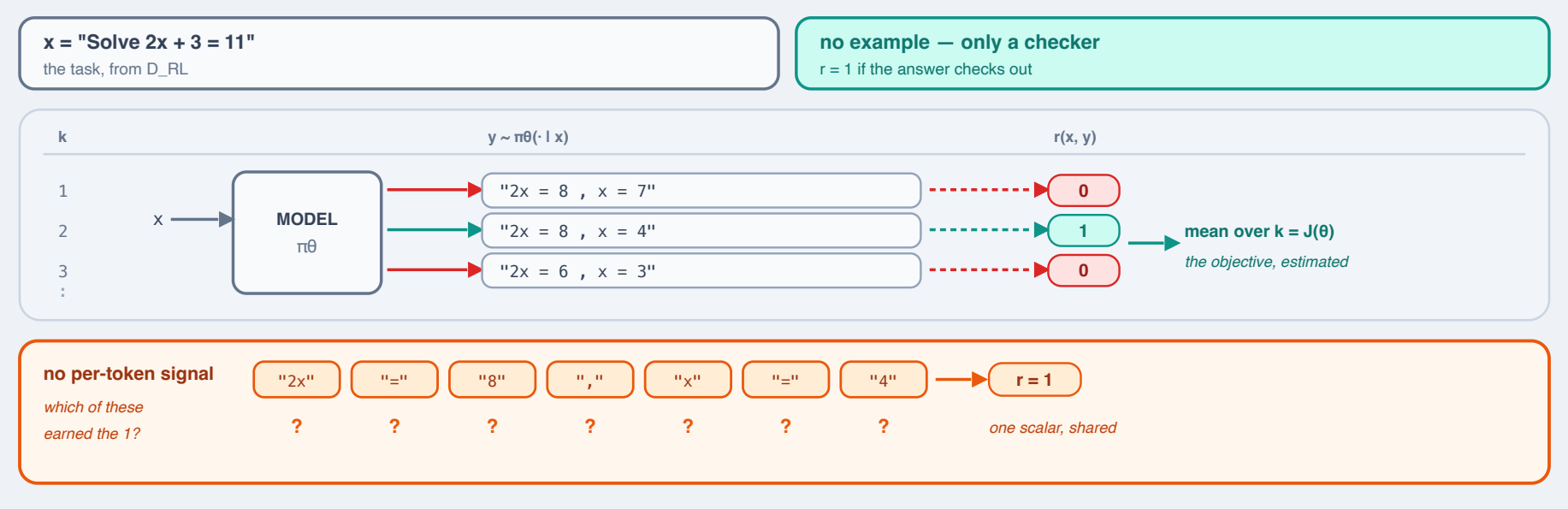
- **Exposure bias:** trained on the given prefix, but at inference it consumes its own tokens (possibly erroneous)
- **Ceiling:** only reproduces the demonstrations - as good as the data's quality and coverage

What if we don't have high-quality expert demonstrations?

Reinforcement Learning (RL)

- **Data:** $\mathcal{D}_{RL} = \{x^{(i)}\}$ - tasks only, no expert example
- **Reward:** $r(x, y)$ scores each sampled $y \sim \pi_\theta(\cdot \mid x)$ - the example's stand-in
  - Verifiable: math (0/1 answer correctness), code (unit tests), kernels (compile, check, profile)
- **Objective:** make high-reward tokens more likely - maximize expected reward

$$J(\theta) = \mathbb{E}_{x \sim \mathcal{D}_{RL}} \mathbb{E}_{y \sim \pi_\theta(\cdot|x)} \left[r(x, y) \right]$$



- The model can reinforce solutions it discovered itself, beyond any demonstration
- Misaligned or noisy reward pulls the model in the wrong direction

Policy Gradient (PG)

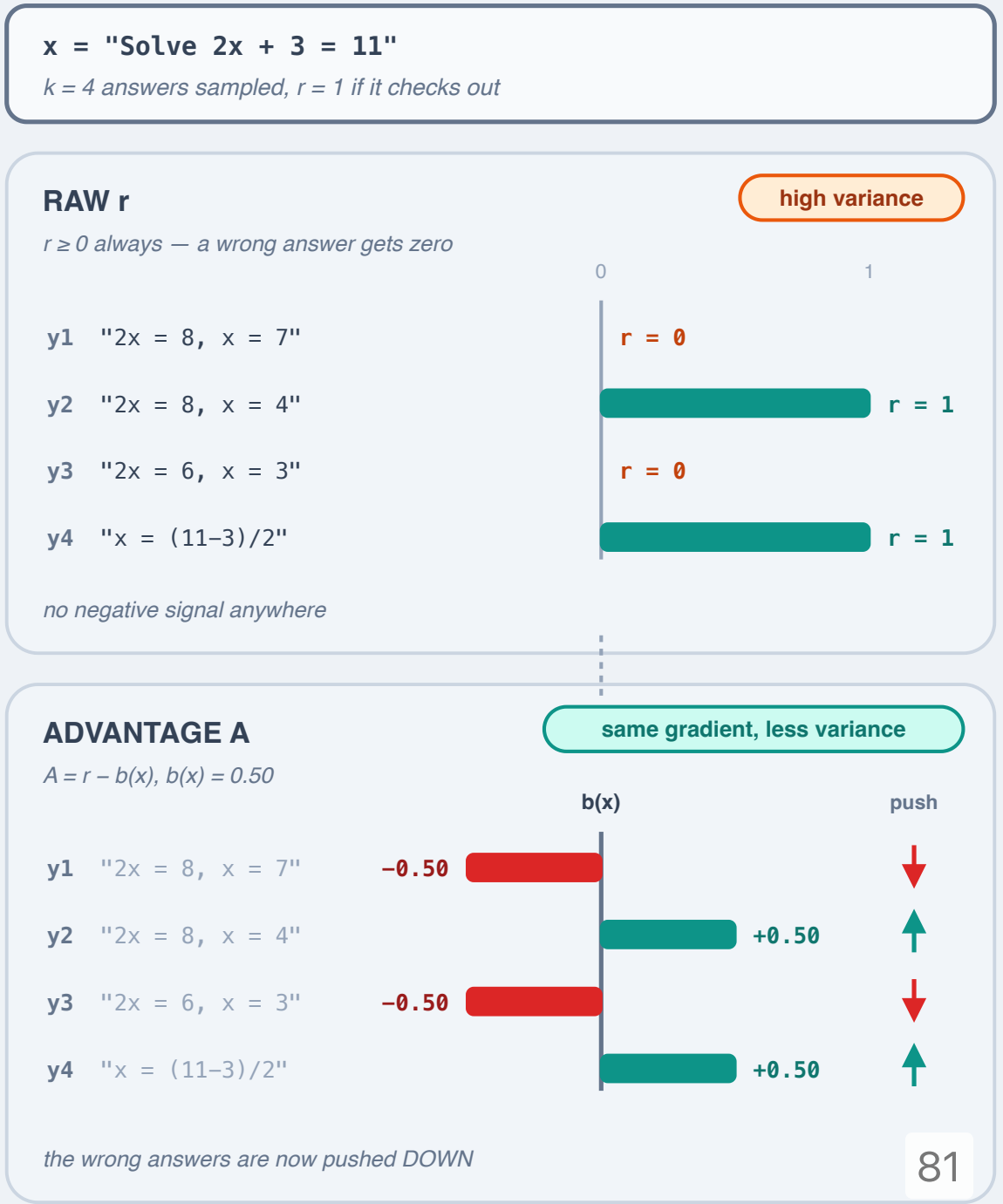
- Goal:

$$J(\theta) = \mathbb{E}_{x\sim\mathcal{D}_{RL}}\,\mathbb{E}_{y\sim\pi_\theta(\cdot|x)}\big[r(x,y)\big]$$

- Because $\nabla\pi_\theta = \pi_\theta\nabla\log\pi_\theta$,

$$\nabla_\theta J = \mathbb{E}\big[\, r(x,y)\,\nabla_\theta \log \pi_\theta(y \mid x)\,\big]$$

- Unbiased but high-variance
- **Variance reduction:**
  - Substitute an advantage $\hat{A}(x,y) = r(x,y) - b(x)$ for $r(x,y)$
  - Same gradient, smaller variance because $\mathbb{E}\big[b(x)\nabla_\theta\log\pi_\theta\big] = 0$
- Advantage design => PPO, GRPO



Proximal Policy Optimization (PPO)

Advantage - baseline = a learned critic V_ϕ (actor-critic)

$$\hat{A}_t = r(x, y) - V_\phi(x, y_{<t}) \qquad (\text{general: GAE, } \hat{A}_t = \textstyle\sum_{l \geq 0} (\gamma\lambda)^l \delta_{t+l})$$

with one-step TD residual $\delta_t = r_t + \gamma V_\phi(x, y_{\leq t}) - V_\phi(x, y_{<t})$ ($r_t = 0$ until the last token; $\gamma, \lambda \in [0, 1]$)

Loss - clipped surrogate on the ratio $\rho_t = \pi_\theta(y_t \mid \cdot) / \pi_{\theta_{\text{old}}}(y_t \mid \cdot)$

$$\mathcal{L}^{\text{PPO}} = \mathbb{E}\big[\min\big(\rho_t \hat{A}_t,\ \text{clip}(\rho_t, 1-\epsilon, 1+\epsilon)\,\hat{A}_t\big)\big], \qquad \epsilon \approx 0.1\text{–}0.2$$

The critic provides the baseline

Pros / cons

- $+$ per-token (fine-grained) baseline
- $-$ need to train the critic (doubles memory)
- $-$ a bad critic gives misleading advantages

Group Relative Policy Optimization (GRPO)

Advantage - baseline = the MEAN of a group of G responses; no critic

$$\hat{A}_i = \frac{r_i - \text{mean}(\{r_1, \ldots, r_G\})}{\text{std}(\{r_1, \ldots, r_G\})} \qquad (\text{one } \hat{A}_i \text{ shared by every token of } y_i)$$

Loss - PPO's clipped surrogate, averaged over the group

$$\mathcal{L}^{\text{GRPO}} = \mathbb{E}\left[\frac{1}{G}\sum_{i=1}^{G}\frac{1}{|y_i|}\sum_t \min\left(\rho_{i,t}\hat{A}_i,\ \text{clip}(\rho_{i,t}, 1-\epsilon, 1+\epsilon)\hat{A}_i\right)\right]$$

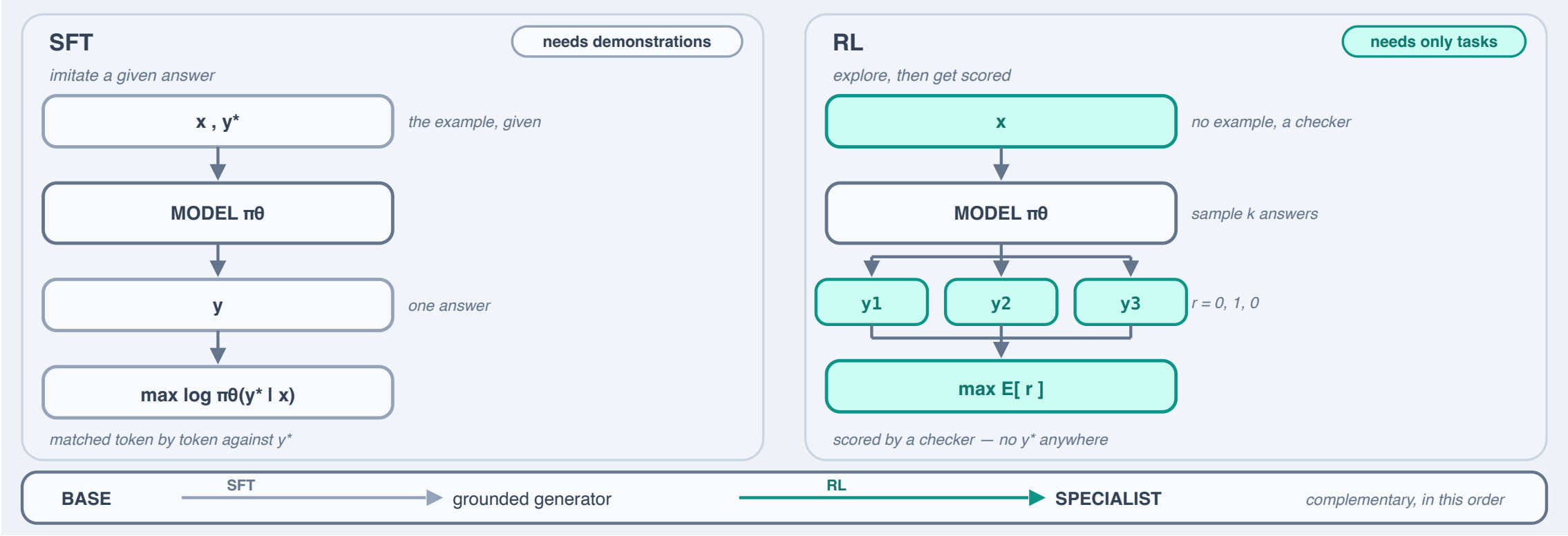
The baseline is the average reward within a group - beat your group

Pros / cons

- $+$ no critic to train
- $-$ sequence-level only, no per-token credit

LLM Post-Training: SFT and RL

- Two ways to move the weights: imitate a given example (SFT), or explore and be scored (RL)
- SFT on demonstration quality, RL on reward reliability
- SFT grounds the model, RL then pushes past it on tasks with no demonstrations



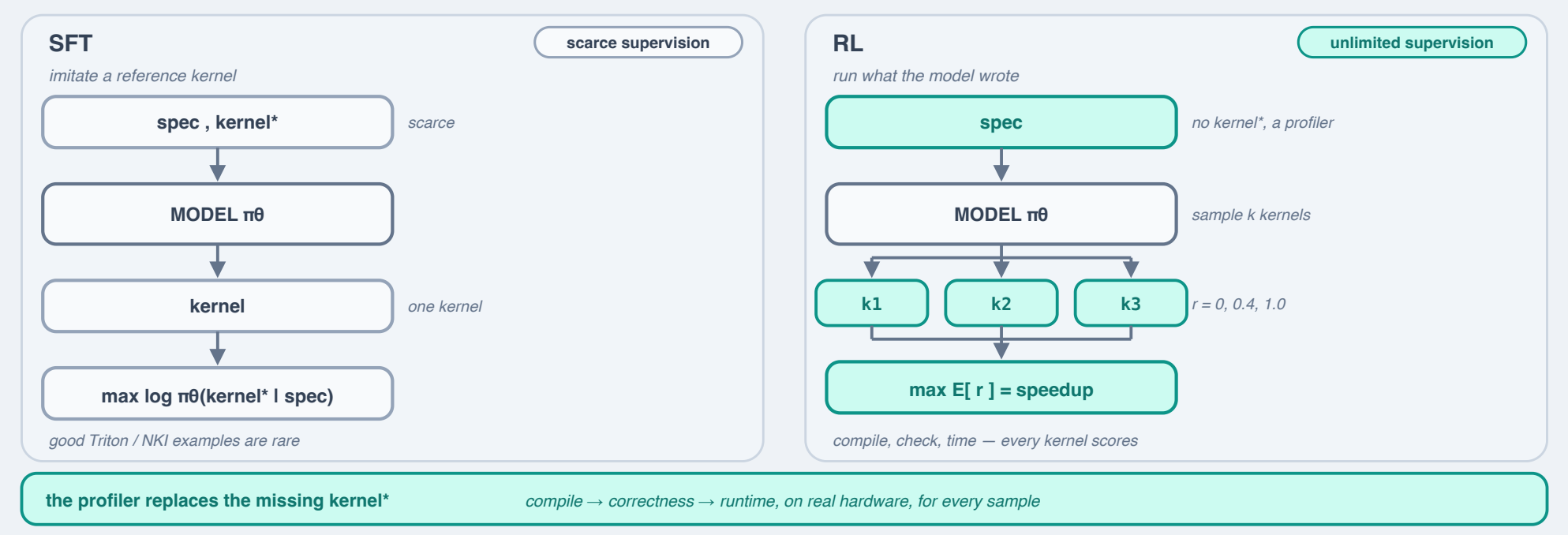
Turning a Generalist LLM into a Kernel Specialist

SFT

- Imitate kernel examples for syntax/skill grounding
- **Demonstrations are scarce:** far fewer Triton / NKI kernels than mainstream code

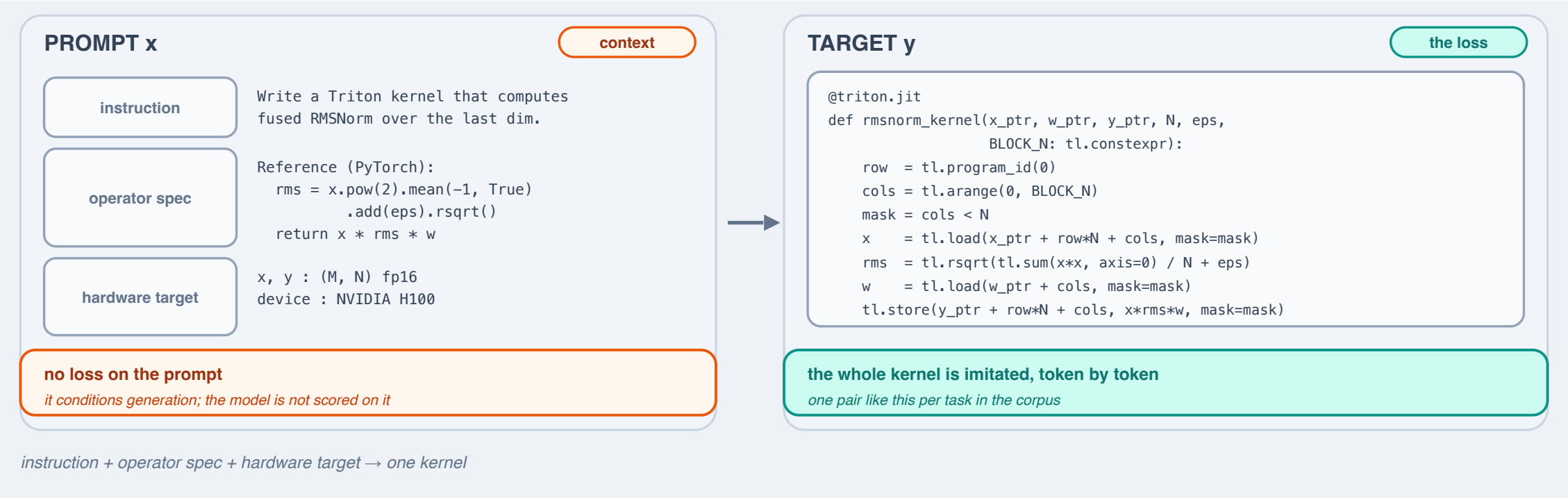
RL

- Reward or penalize the model's own kernels by what the profiler measures
- **Rewards are rich:** the profiler grades compile, correctness, and runtime



SFT on Kernel Corpora

- Kernel corpora: $\mathcal{D}_{\mathrm{SFT}} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$
  - **Prompt (x)** operator spec + hardware target/constraints + instruction
  - **Target (y)** a correct efficient kernel



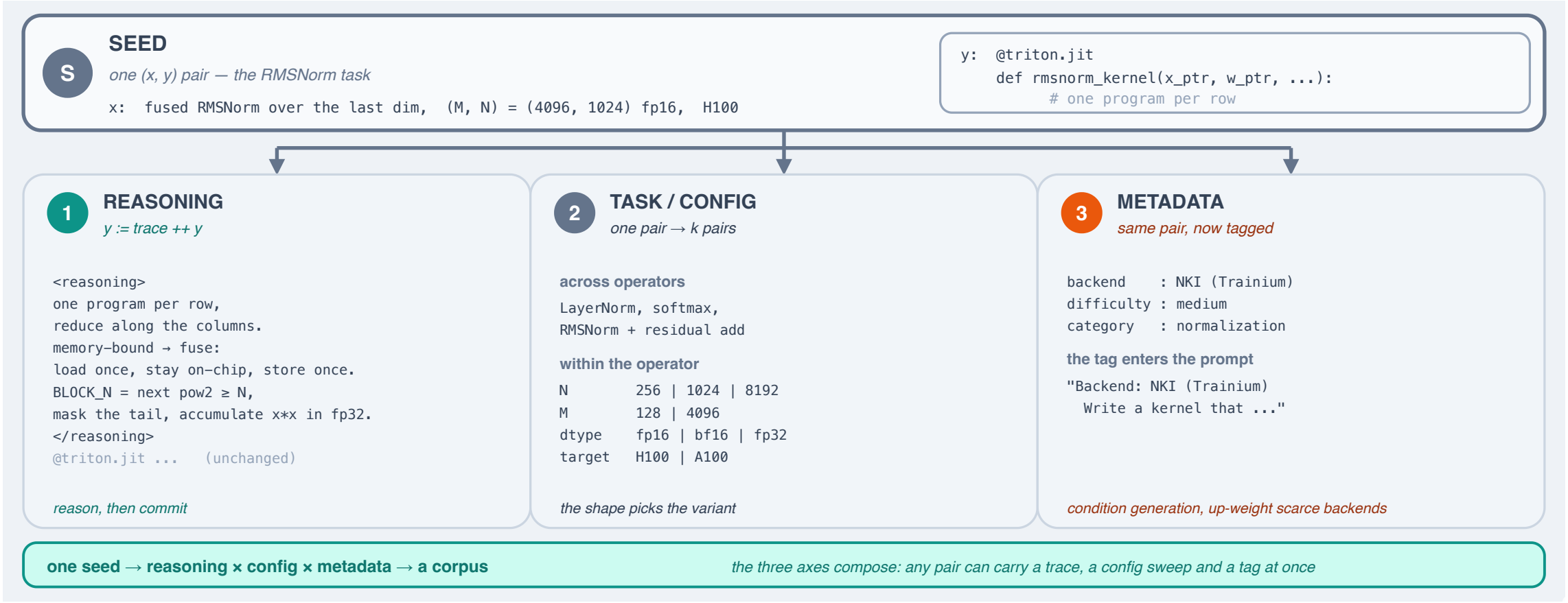
- SFT grounds syntax/APIs and reshapes the prior toward idiomatic kernels
- However, kernel data is often scarce and not clean - how to curate a kernel corpus for SFT?

Building the Kernel Corpus for SFT

- Data source types: expert examples (human/library/vendor) and teacher models
  - Public kernel expert examples are **scarce**
  - Frontier models may not be **specialized** for specific kernels/hardware constraints
- **Augmentation** for diverse and richer examples
- **Sampling** for high-quality and targeted examples

SFT Data Curation: Data Augmentation

- **Reasoning:** reasoning traces before kernel code for diverse kernel optimization skills (tiling/blocking/staging)
- **Task/config:** vary operators and shapes/dtypes to diversify the task pool
- **Metadata:** tag difficulty / backend / category, to condition generation or organize training



SFT Data Curation: Sampling

- **Rejection sampling:** keep only high-quality examples
  - Check compilation, correctness, and runtime
  - Check validity to screen sham kernels
- **Deduplication:** remove cosmetic clones (exact hash or Jaccard similarity)
  - Measure similarity based on AST/op-set
- **Difficulty balancing:** label tasks by difficulty and upweight under-represented levels
  - KernelBench Levels 1/2/3 or op complexity
  - Mined corpora skew easy, and rejection sampling worsens the skew

k	what it does	compile	maxIdyI *atol 2e-2*	speedup	lint	verdict		
k1	1-row B_N=1024	✗	–	–	–	drop	compile	*BLOCK_N not constexpr*
k2	1-row fp16 accum	ok	3.1e-2	–	–	drop	correct	*over atol*
k3	mean-subtracting	ok	1.4e-1	–	–	drop	correct	*LayerNorm, not RMSNorm*
k4	1-row no masking	ok	7.2e-1	–	–	drop	correct	*tail cols unwritten*
k5	jit shell	ok	0	2.1x	✗	drop	lint	*wrapper returns torch op*
k6	1-row B_N=1024	ok	4e-4	1.8x	ok	KEEP		
k7	1-row B_N=512	ok	4e-4	1.7x	ok	drop	dedup	*same AST as k6*
k8	blocked reduction	ok	5e-4	1.1x	ok	KEEP		

8 draws → 7 compile → 4 correct → 3 lint → 2 kept

one pass, first failing gate wins

then level-balance what survives
L1 ×40 down, L2 ×9, L3 ×1 up

every draw is scored by the same execution harness the RL stage will use

SFT on Curated Kernel Corpus

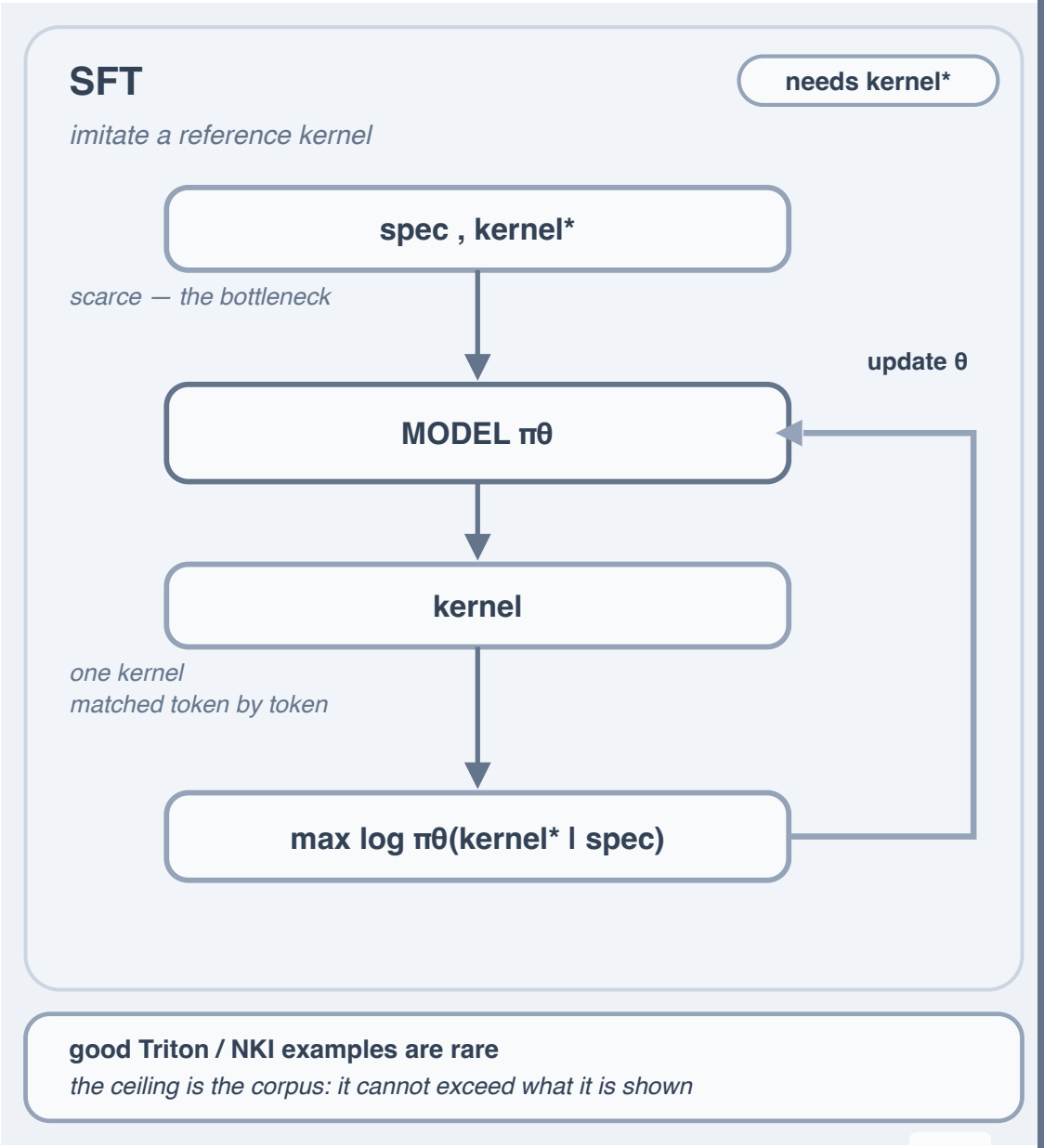
SFT on curated kernel corpus $\mathcal{D}_{\text{SFT}} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$: optimize negative log-likelihood

$$\mathcal{L}_{\text{SFT}}(\theta) = -\frac{1}{N} \sum_{i} \sum_{t} \log \pi_\theta \left(y_t^{(i)} \mid x^{(i)}, y_{<t}^{(i)} \right)$$

SFT grounds syntax/APIs on various tasks. However,

- **Bounded by demonstrations:** cannot beat the corpus
- **Not optimized for specific hardware:** the model's objective is imitation, not speedup on chip (true objective)
- **Data scarcity:** for niche backends (NKI on Trainium) even strong teachers yield mostly invalid draws

Without being limited by demonstrations, let the model explore and get verified signals by profiling kernels on chip
=> **RL with verifiable rewards (RLVR)**

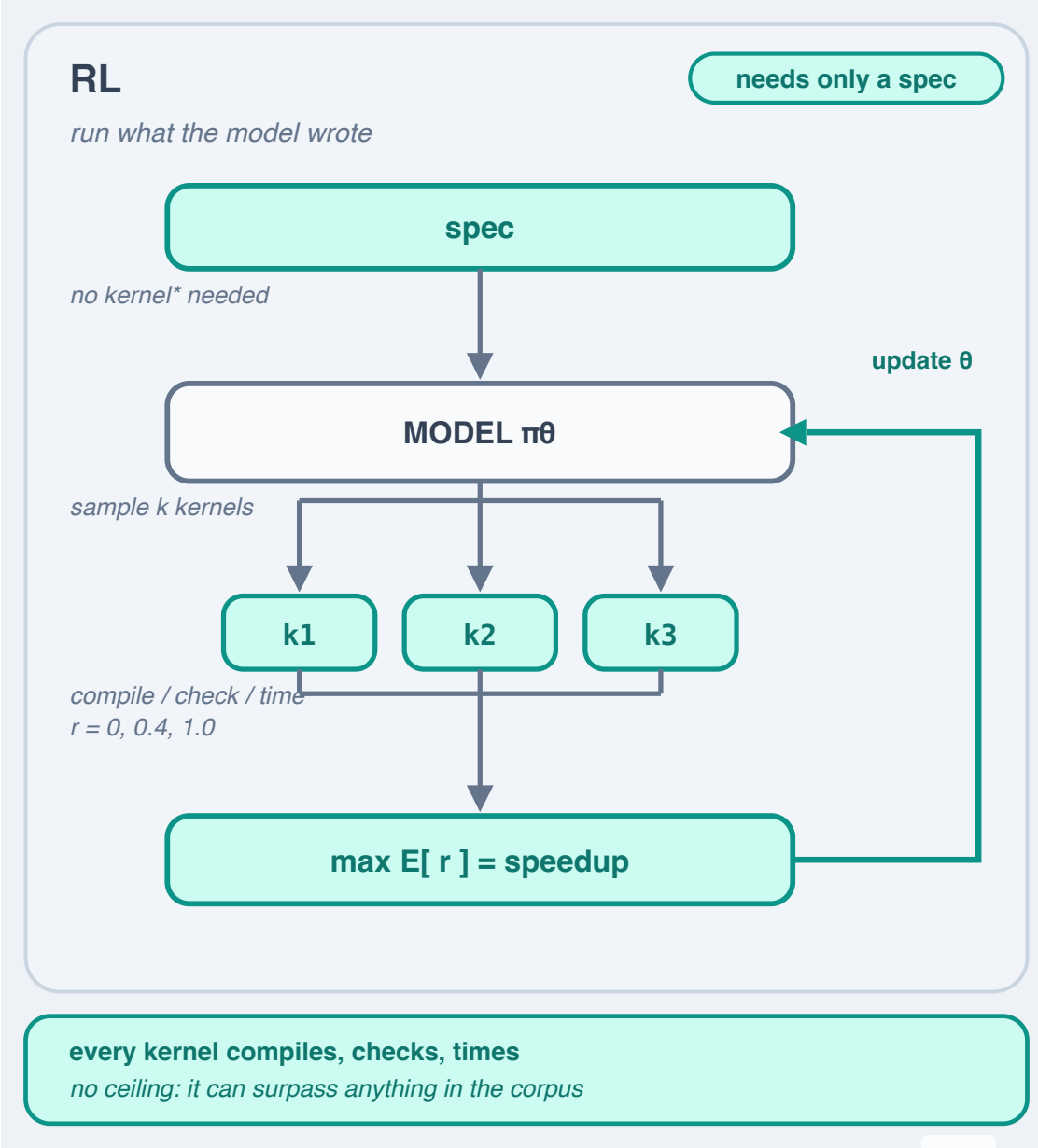


RLVR with Execution Feedback

RL on task-only corpus $\mathcal{D}_{\mathrm{RL}} = \{x^{(i)}\}_{i=1}^{N}$**:** optimize rewards verified by execution

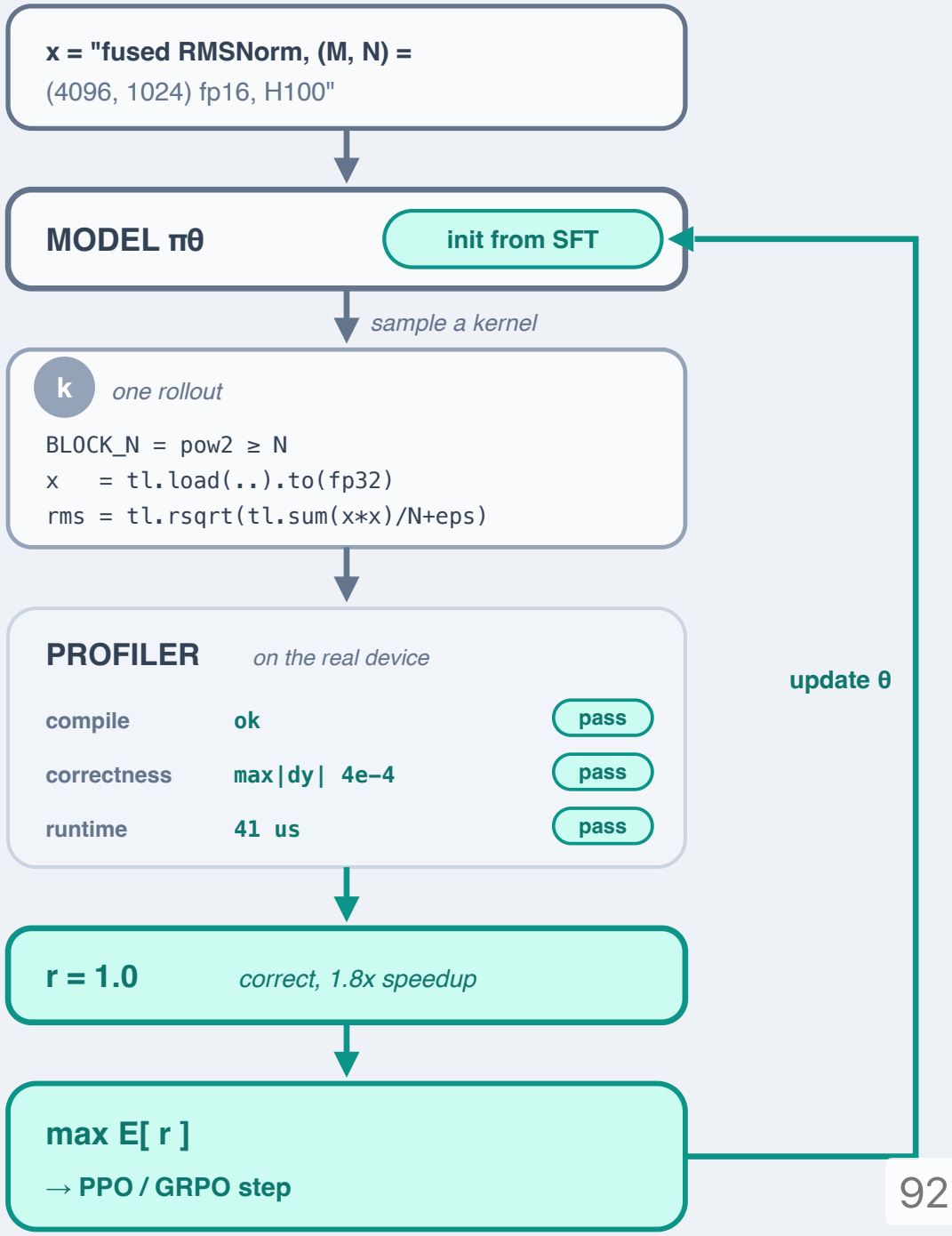
$$J(\theta) = \mathbb{E}_{x \sim \mathcal{D}_{\mathrm{RL}}} \mathbb{E}_{y \sim \pi_\theta}[\, r(x, y)\,]$$

- Directly optimize verified rewards, and thus **not bounded** by the teacher's performance.
- Optimizes the **true performance on the chip**
  - If $r(x, y) \propto$ runtime performance
- No need for target kernels, can use **wider task set** without expert demonstrations.



RL Loop with Execution Feedback

- Execution feedback from profiler: compiled, correct, runtime speedup
- Reward = f(compiled, correct, speedup)
- Maximize reward $J(\theta) = \mathbb{E}_{x\sim\mathcal{D}_{RL}}\,\mathbb{E}_{y\sim\pi_\theta(\cdot|x)}\left[r(x,y)\right]$ via PPO or GRPO



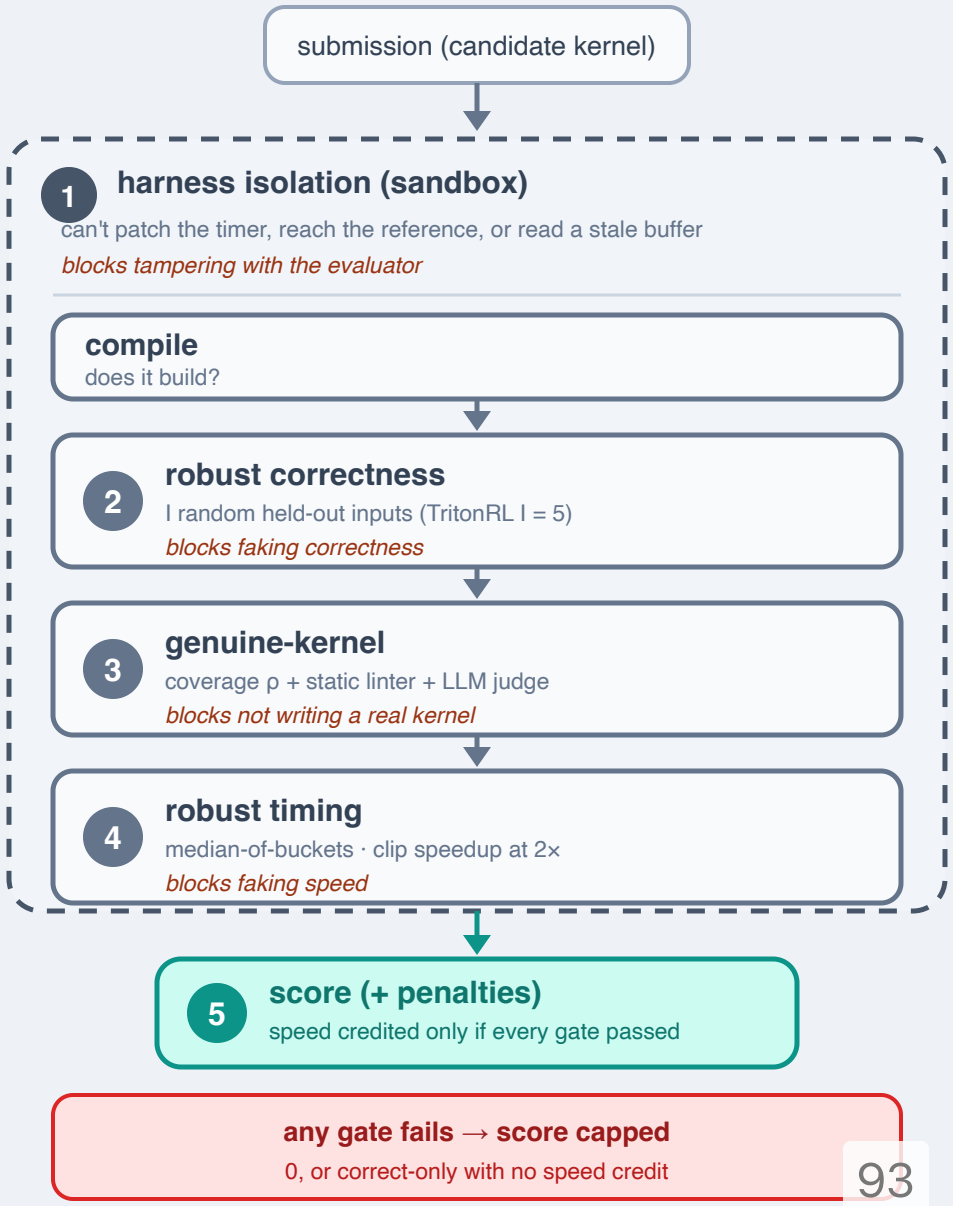
The Execution Feedback

- **Compile/correctness** on randomized inputs,

$$\text{correct}(y) = \mathbf{1}\left[\max_{j=1,\ldots,I} \left\| f_y(v_j) - f_{\text{ref}}(v_j) \right\| \le \text{atol} + \text{rtol} \cdot \left\| f_{\text{ref}}(v_j) \right\| \right]$$

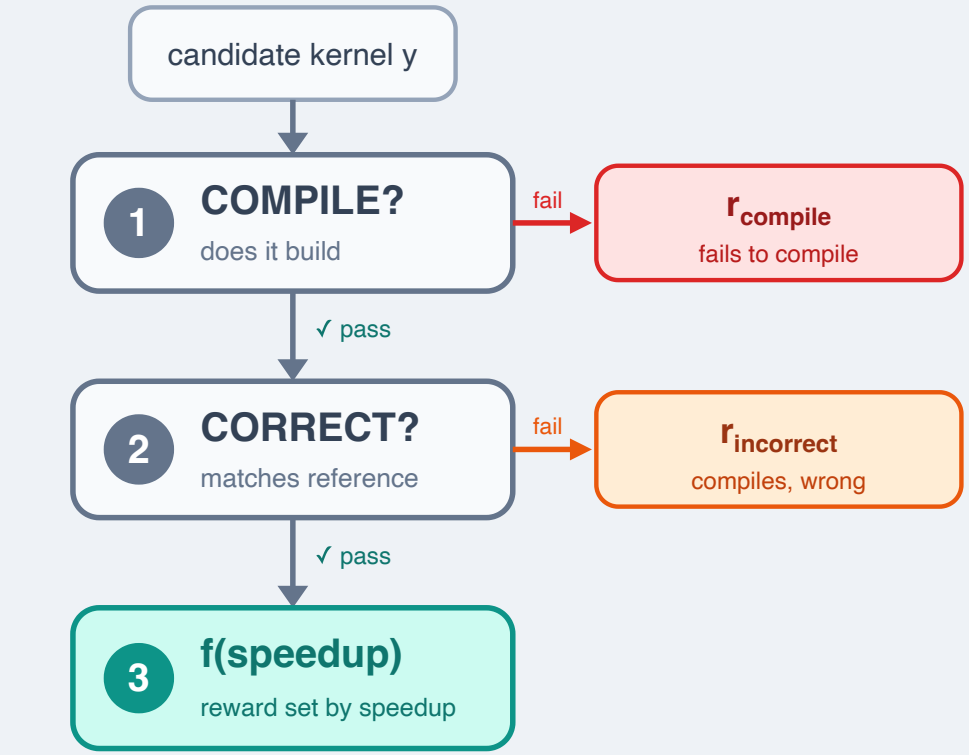
- **Genuine-kernel**: static linter + LLM judge + on-chip profile
- **Runtime speedup**:

$$\hat{t}(y) = \text{median}_{1 \le b \le B} \left(\frac{1}{|b|} \sum_{r \in b} t_r \right), \quad \text{speedup}(y) = \frac{\hat{t}(\text{baseline})}{\hat{t}(y)}$$



Reward Designs

- **Linear weighted sum** - dense, but every term is a surface to exploit
  - $r = w_c\,\text{correct} + w_s\,\text{speedup}$ (Kevin)
- **Discrete milestone** - ordered tiers, bounded, credits partial progress
  - $r = 0$ (no compile) $\rightarrow 1$ (correct) $\rightarrow 2$ (+ faster) (CUDA Agent)
- **Conditional gating** - any failed gate zeros the reward
  - $r = \text{valid} \cdot \text{correct} \cdot \text{clip}(\text{speedup}, 2)$ (TritonRL)



how each outcome scores

$r_{compile} \le r_{incorrect} \le 0 < \mathbf{f(speedup)}$

Policy Optimization for Kernel RL

PPO - $\hat{A}_t = r(x, y) - V_\phi(x, y_{<t})$, baseline = a **trained critic**

- Per-token baseline - can credit *where* in the kernel the win came from
- The critic should be trained to grade a half-written kernel
- e.g. CUDA Agent

GRPO - $\hat{A}_i = \frac{r_i - \text{mean}(\{r_1, \ldots, r_G\})}{\text{std}(\{r_1, \ldots, r_G\})}$ over kernels in G, **no critic**

- "Fast" is graded within a group, not on an absolute scale
- All kernels score alike (all correct) $\Rightarrow \hat{A} = 0$, no learning
- e.g. AutoTriton, TritonRL, DRTriton, Kevin

Curriculum and Task Selection (1)

- If tasks are too easy or too hard, the model learns nothing.
- GRPO: all kernels in a group fail, or all pass -> gradient = 0
  - Useful task = kernels in the group differ (variance > 0)

$$A_i = (r_i - \text{mean}(r)) / \text{std}(r)$$

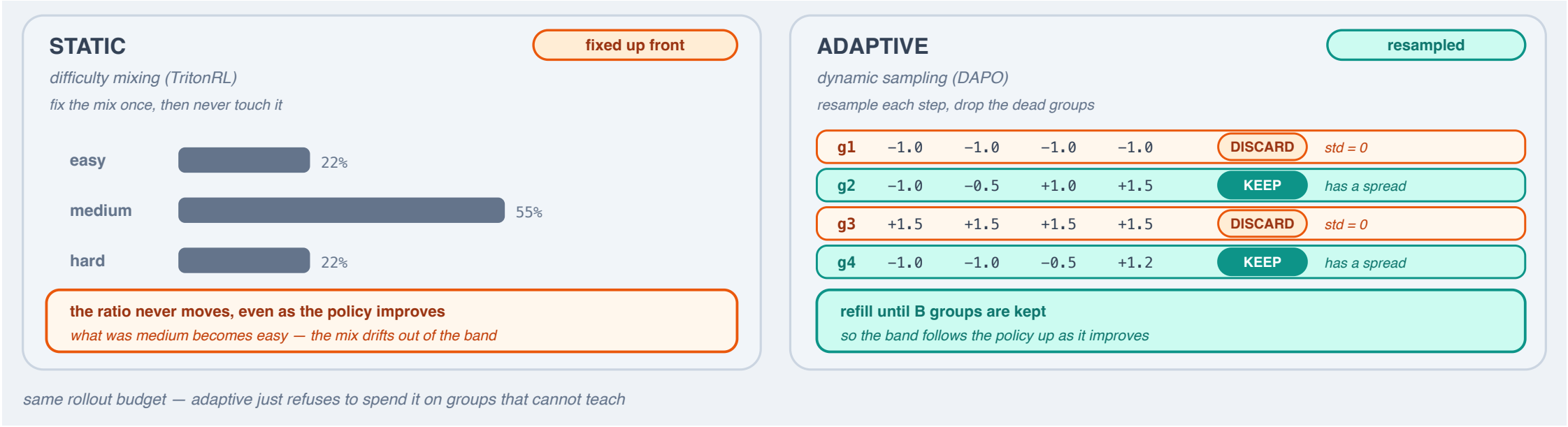
std(r) = 0 → A = 0 → no gradient

TASK	r over G = 4 kernels					A_i					LEARN?
too hard *every kernel fails*	−1.0	−1.0	−1.0	−1.0	→	+0.0	+0.0	+0.0	+0.0	0	dead
too easy *every kernel wins*	+1.5	+1.5	+1.5	+1.5	→	+0.0	+0.0	+0.0	+0.0	0	dead
just right *a spread to learn from*	−1.0	−0.5	+1.0	+1.5	→	−1.2	−0.7	+0.7	+1.2	spread	YES

the batch is spent either way — a dead group costs the same rollouts as a live one

Curriculum and Task Selection (2)

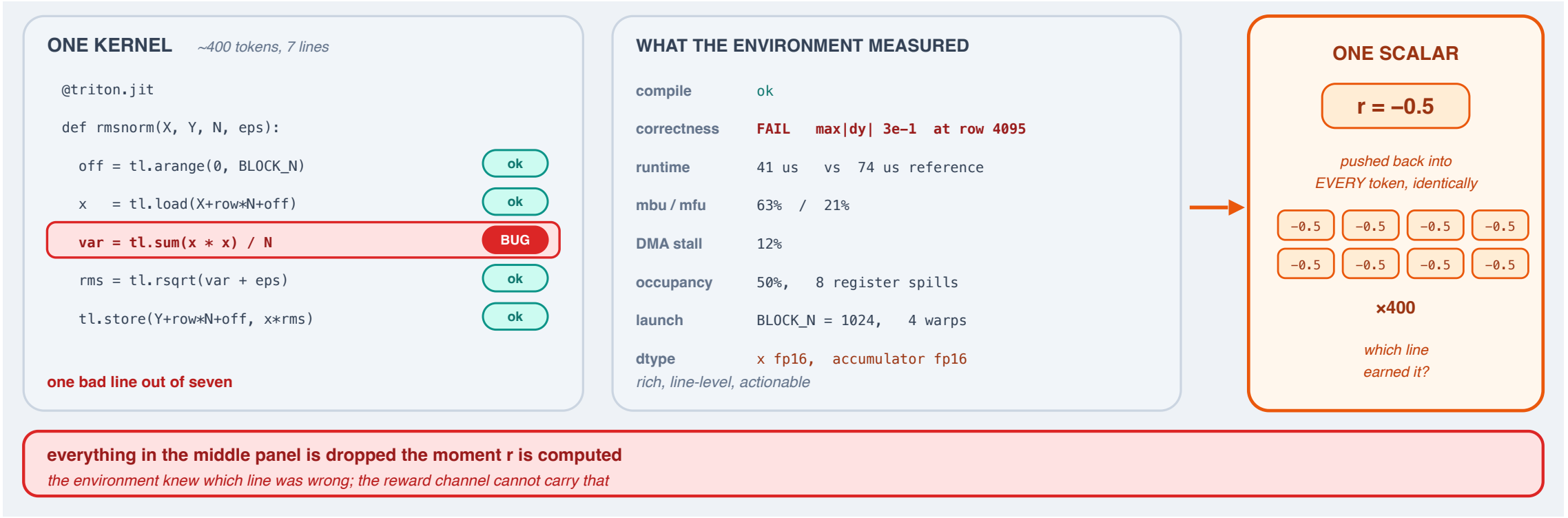
- **Static curation:** curriculum ordering, data mixing, difficulty filtering
  - Difficulty mixing (TritonRL): fixed difficulty ratio
- **Adaptive sampling:**
  - Dynamic sampling (DAPO): discard all-pass/all-fail groups
  - Kernel coverage (Dr. Kernel): keep rollouts whose kernel covers the end-to-end runtime



Limitations of Single-Turn RL

One kernel per task, one terminal reward

- **Coarse credit assignment:** one scalar for the whole response
- **Discarded rich profiling information:** compiler diagnostics, failing tests, and profile metrics (MBU, MFU, DMA, ...)



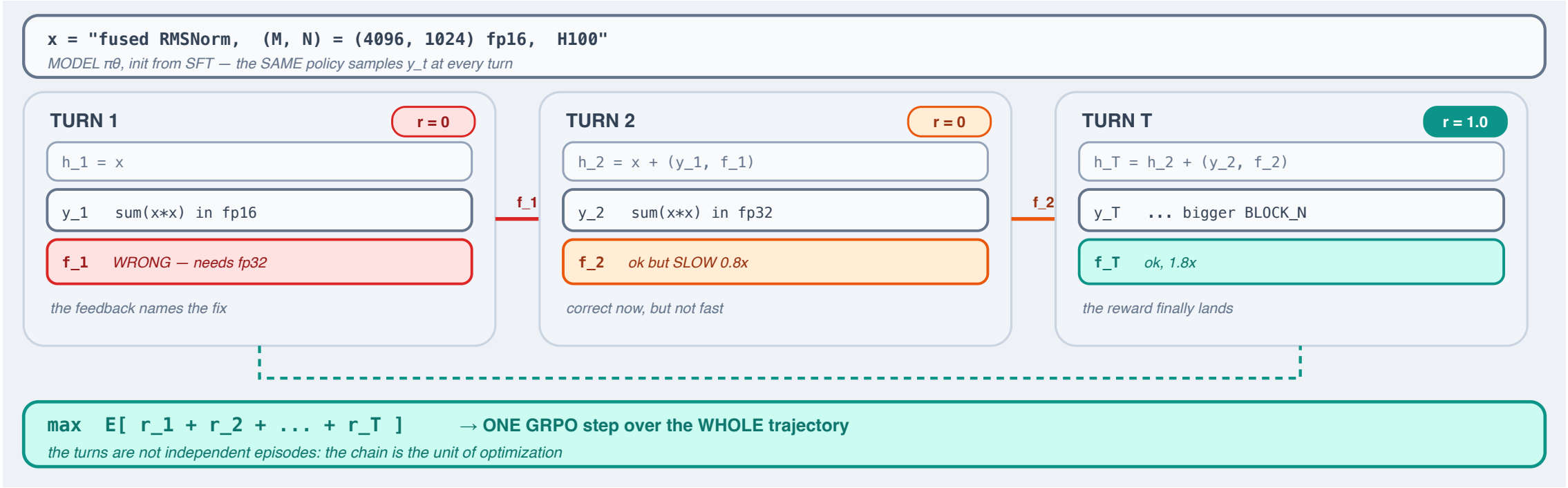
Multi-Turn RL: Train the Policy to Refine

- **Single-turn (recap):** one kernel y per task, scored once

$$J(\theta) = \mathbb{E}_{x \sim \mathcal{D}_{RL}} \mathbb{E}_{y \sim \pi_\theta(\cdot|x)} \left[r(x, y) \right]$$

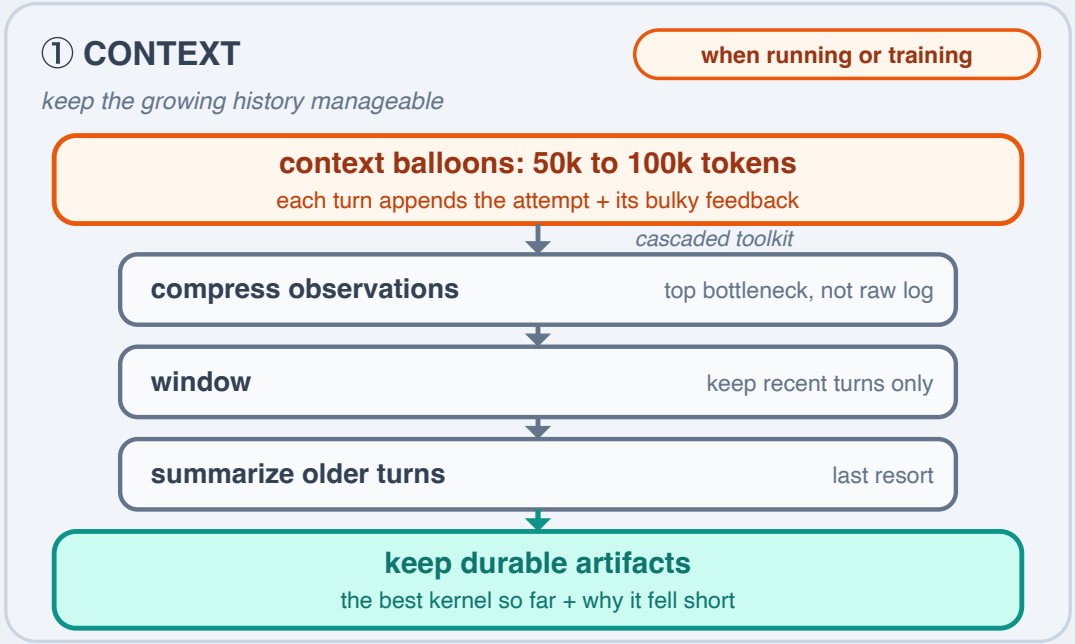
- **Multi-turn:** each attempt sees the previous attempts and their feedback f_t

$$h_1 = x, \quad h_t = (x, y_1, f_1, \ldots, y_{t-1}, f_{t-1}), \quad y_t \sim \pi_\theta(\cdot \mid h_t), \quad J(\theta) = \mathbb{E}_x \, \mathbb{E}_{\tau \sim \pi_\theta(\cdot|x)} \left[\sum_{t=1}^{T} r_t \right]$$



Issues of Multi-Turn Loop

1. **Context** - the history grows every turn
2. **Credit assignment** - the reward lands only at the end



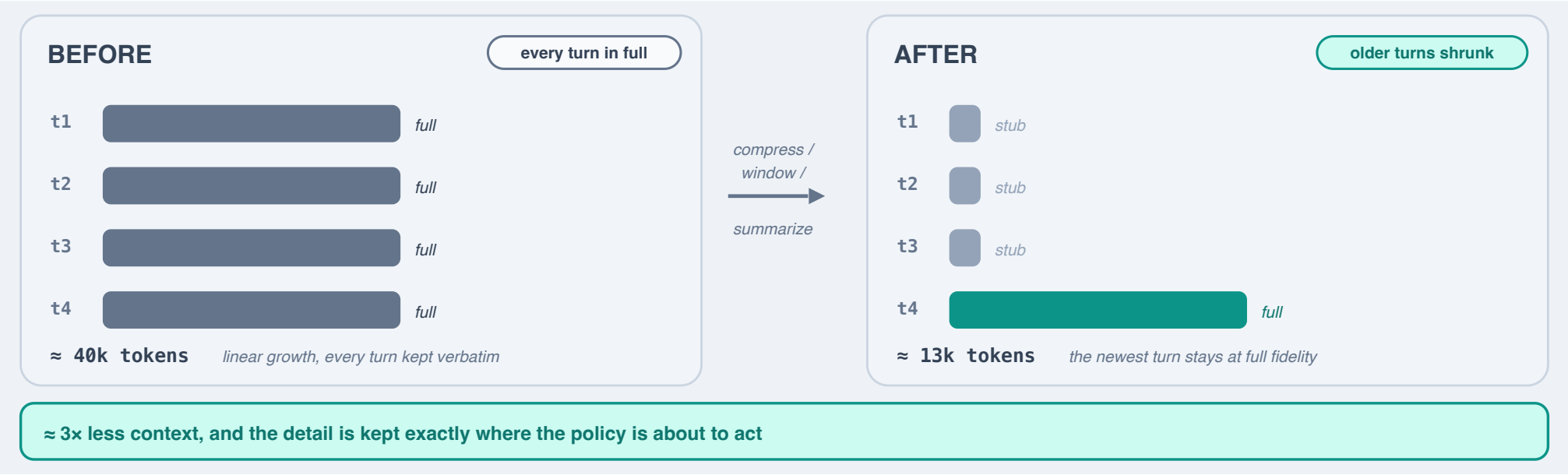
Context Management

Every turn appends the last attempt + feedback => context explodes

- Reaches 50-100k tokens within a few turns

How to bound the context length?

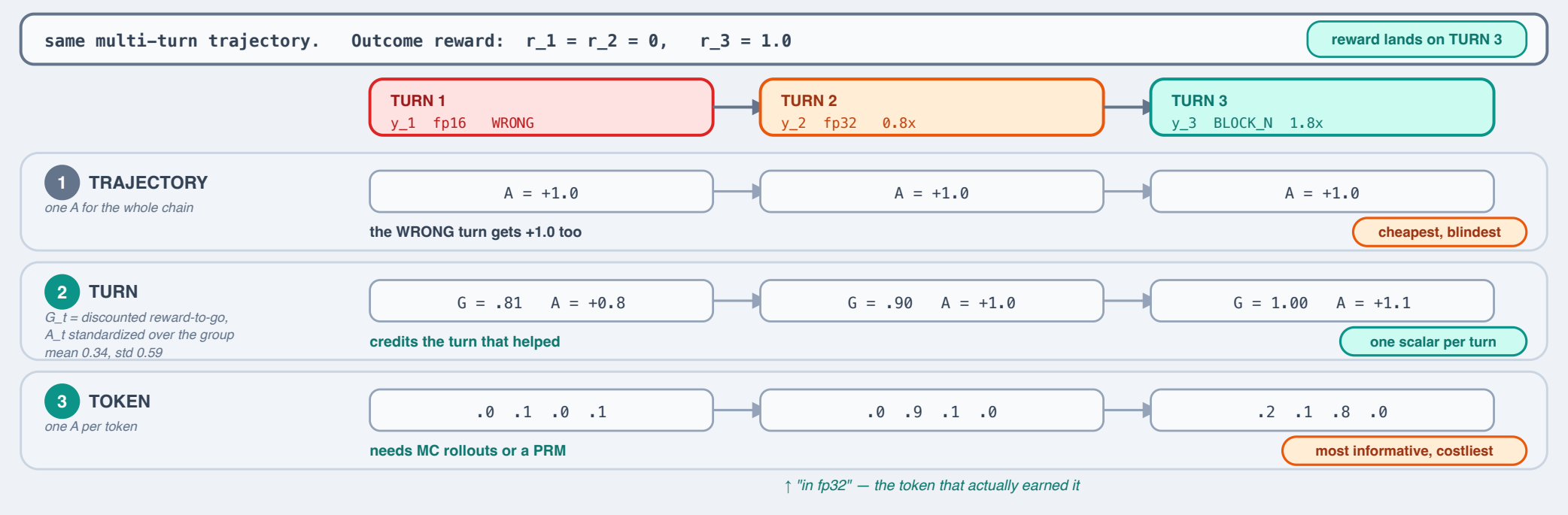
- **Observation compression:** keep the first error / top bottleneck, drop raw logs
- **Windowing:** keep only recent turns - simplest, lossy
- **Summarization:** older turns -> a short summary; keep the best kernel + why it failed (Kevin)
- Apply in that order: compress, then window, then summarize



Credit Assignment (1)

How to propagate a reward back to the tokens/turns?

- **Trajectory:** one advantage for every turn - cheapest, blind
- **Turn:** its own advantage per turn from a discounted reward-to-go (Kevin); turn-level RLOO (REINFORCE leave-one-out) (Dr. Kernel)
- **Token:** per-token credit via Monte-Carlo or a process reward model - costly

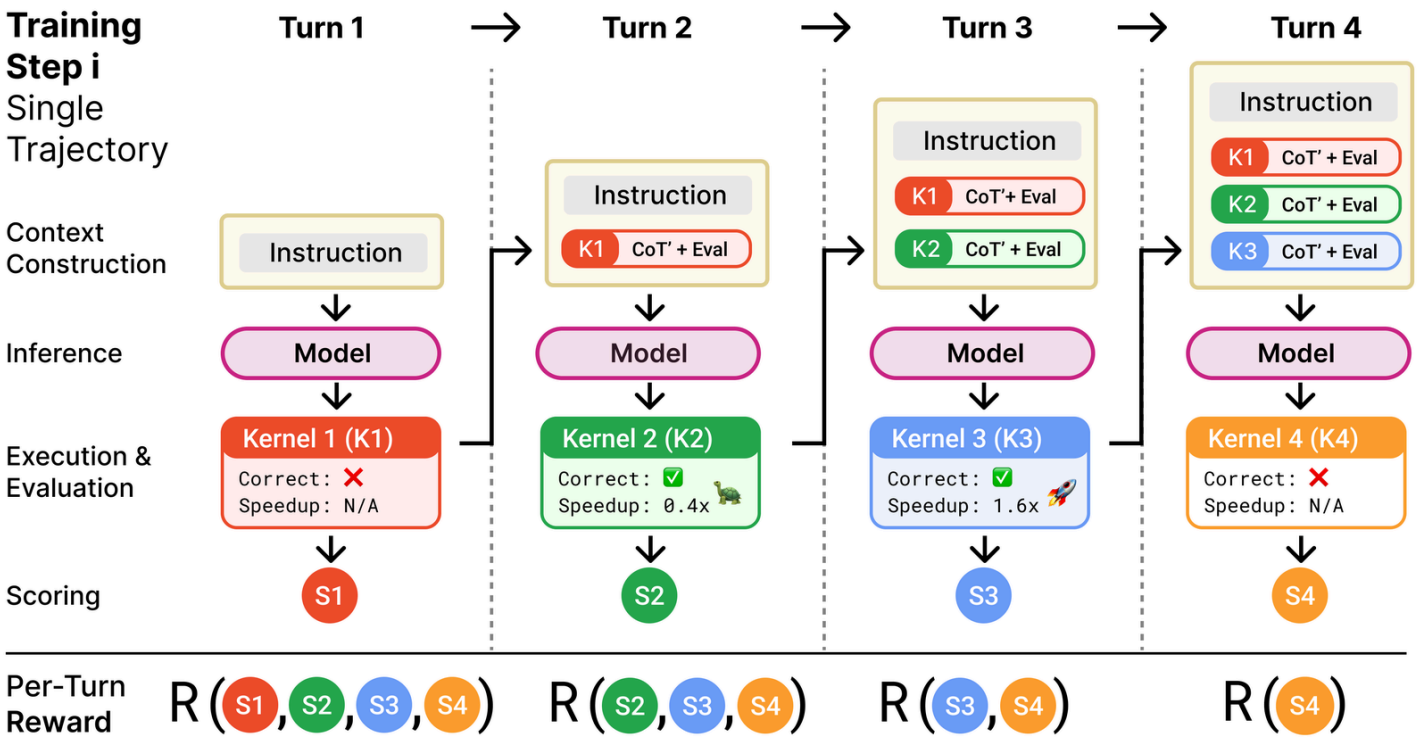


Trade-off: finer credit, lower variance, but more cost

Credit Assignment (2)

Turn-level credit assignment (Kevin-32B, arXiv:2507.11948)

- **Each turn = one training sample**, rewarded by its own *and all later* scores: $R_t = \sum_{i \geq t} \gamma^{i-t} r_i$
- Failed initial turns earn credit for contributing to the fast final kernel
- Middle ground between **greedy** ($R_t = r_t$) and **outcome** ($R_t = \max_i r_i$, blind to each turn); **sum,** $\gamma = 0.4$ scales best



Credit Assignment (3)

Segment-level credit assignment: Hierarchical reward decomposition (TritonRL, arXiv:2510.17891)

- Split one turn into a **plan** span and a **code** span, and reward them separately:

$$J(\theta) = \mathbb{E}[\alpha J_{\text{plan}} + J_{\text{code}}]$$

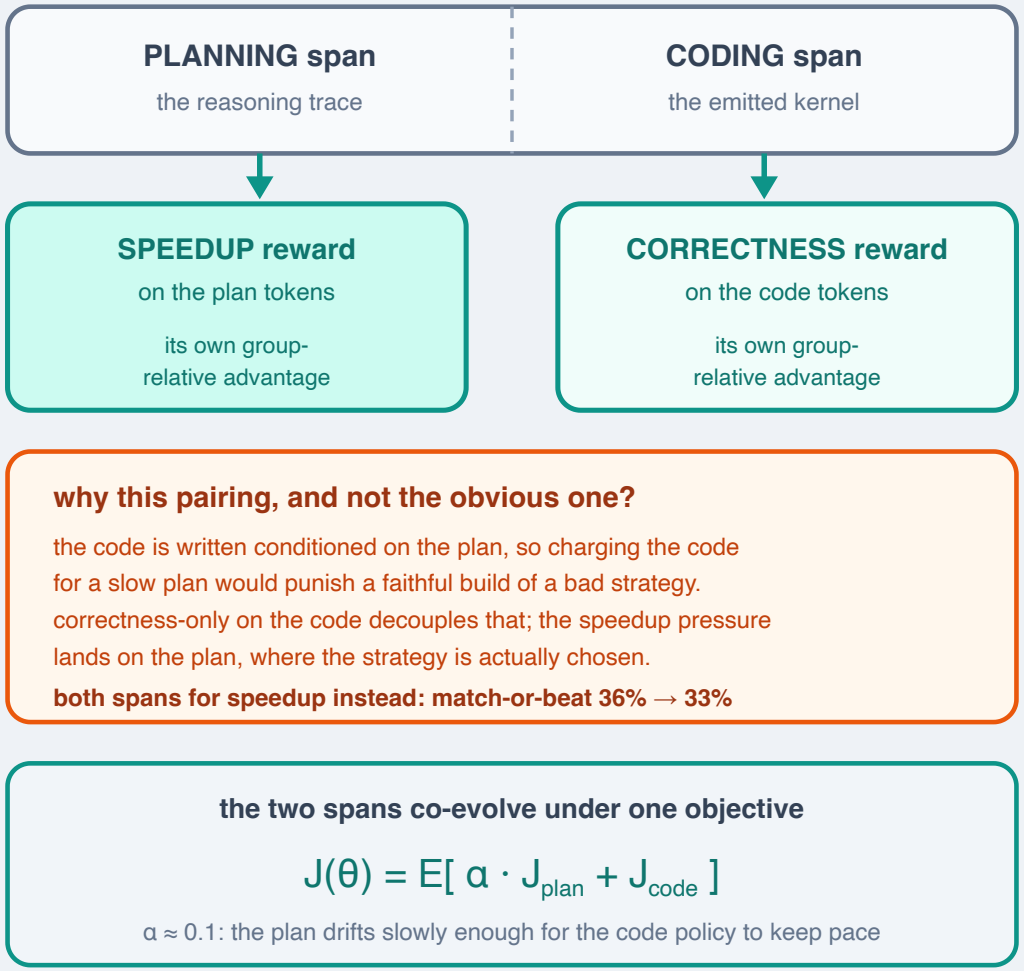
- plan ← **speedup**, code ← **correctness**
- *Why crossed?* the code is written *conditioned on* the plan - charging it for a slow plan would punish a faithful build of a bad strategy
- $\alpha \approx 0.1$ - the plan drifts slowly enough for the code to keep pace

TWO ORTHOGONAL CUTS

by TIME (the credit ladder above): token · turn · trajectory

by ROLE (HRD, here): planning span · coding span

one response y_t, split into two spans by token role



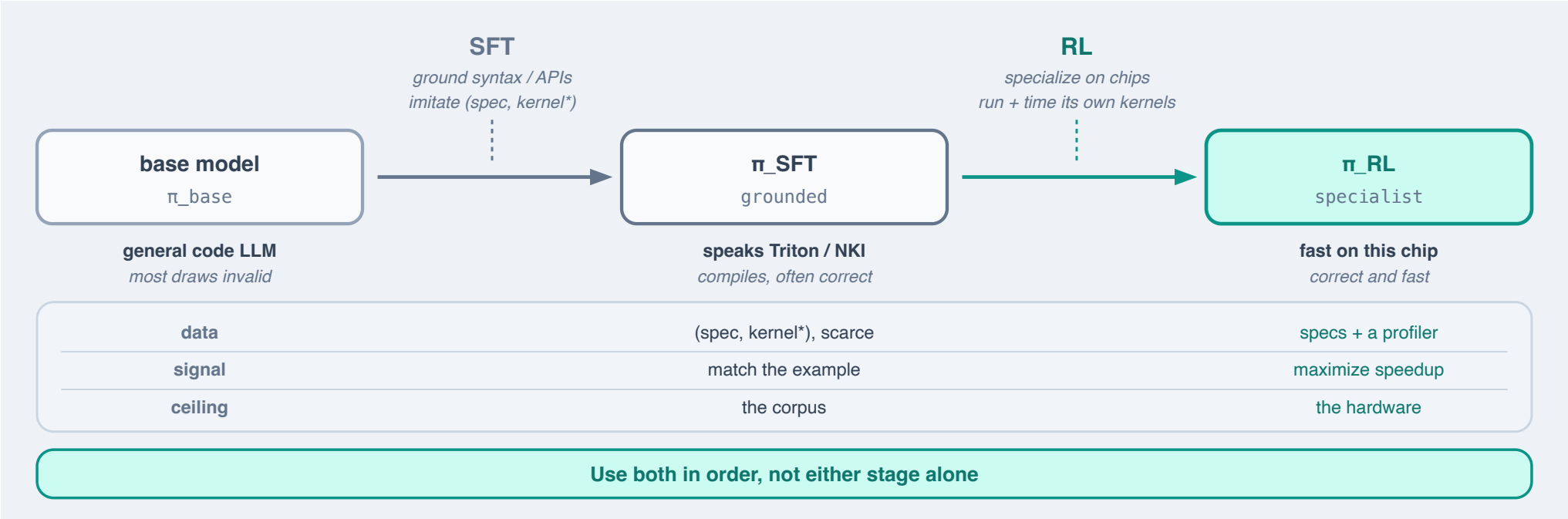
it cuts by role, not by time, so it applies even to a single turn

(TritonRL is single-turn)

Post-Training for Kernel Generation

Initialize the model with SFT and climb with RL.

- **SFT** grounds the model in a kernel language but is capped by its demonstrations and never sees runtime
- **RL** breaks that ceiling: learn from compiling and timing its own kernels



Thank You!

Questions?

We're hiring Full-timers and Interns!

- Foundation models, LLMs
- Efficient LLM training and inference, e.g., low-precision training, compression, pruning
- Distributed Optimization
- System, High-performance Computing, Compiler

Location: Cupertino, California, United States

Reach out to: **Youngsuk Park** (pyoungsu@amazon.com) · **Zhen Jia** (zhej@amazon.com)
Yida Wang (wangyida@amazon.com)