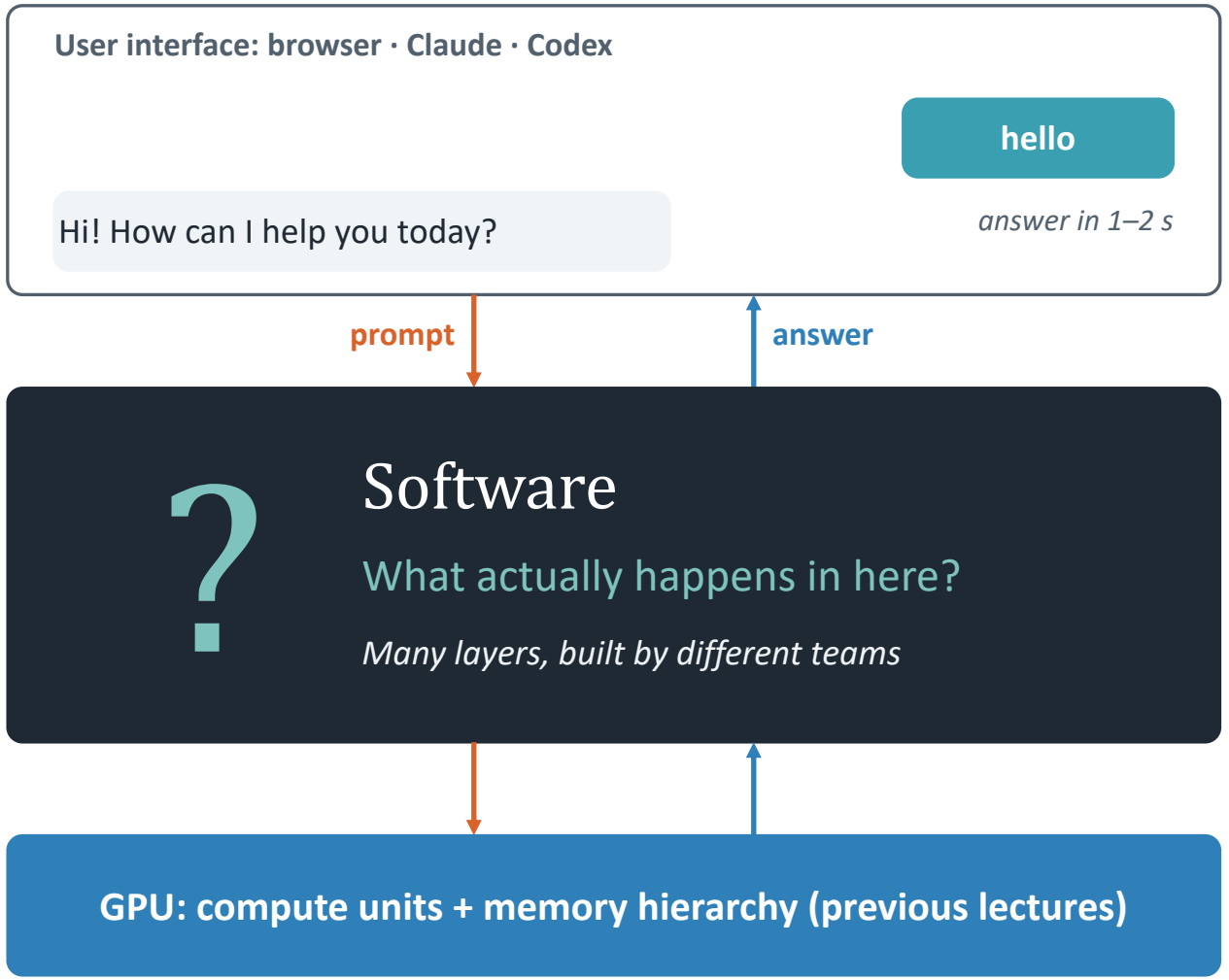
The AI Software Stack as an Optimization Problem

Cliffs are constraints; slopes are mechanisms

Lecture 6 · AI Infrastructure

Li Shang

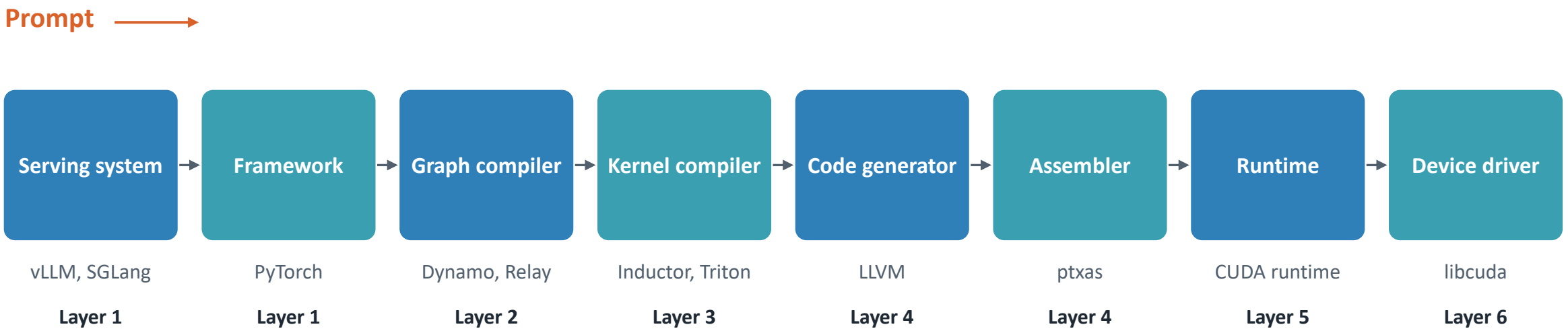
From prompt to GPU: what is in the black box?



Today: open the black box, a bit

How many layers of software touch one prompt?

Approximately eight pieces of software, grouped into six layers, each built by a different team in silo mostly.



Below the driver: GPU hardware (covered in earlier lectures)

What each piece does, in one sentence

Serving system	**Layer 1**	Accepts requests from many users and decides which run together at each step
Framework	**Layer 1**	Holds the model as code (PyTorch modules) and dispatches its operators
Graph compiler	**Layer 2**	Captures the model as a graph of operators (computation/dataflow graph)
Kernel compiler	**Layer 3**	Turns each fused operator into a loop nest and picks its tiling, order, parallelism
Code generator	**Layer 4**	Lowers the loop nest to a hardware-level representation: LLVM IR, then PTX
Assembler	**Layer 4**	Translates PTX into machine code (SASS) for one specific GPU and allocates its registers
Runtime	**Layer 5**	Allocates GPU memory, launches kernels in order and overlaps independent work
Device driver	**Layer 6**	Gives the process a GPU context, manages GPU memory, hands launches to the GPU

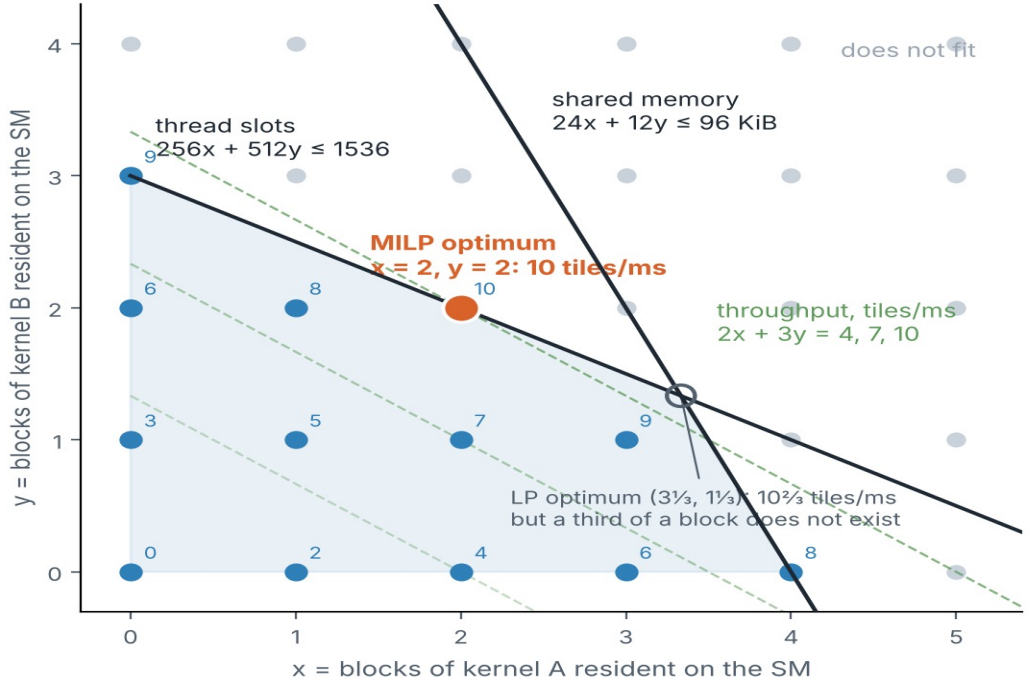
What each piece optimizes, in one sentence

Serving system	**Layer 1**	Keep the GPU busy: batch requests and use KV-cache memory efficiently
Framework	**Layer 1**	Dispatch operators with little overhead and pick the fastest kernel for each
Graph compiler	**Layer 2**	Cut HBM traffic and kernel count by fusing and rewriting operators
Kernel compiler	**Layer 3**	Maximize data reuse in fast memory and keep every SM busy
Code generator	**Layer 4**	Emit instructions that use tensor cores and wide vector loads
Assembler	**Layer 4**	Balance registers against occupancy and schedule instructions to hide latency
Runtime	**Layer 5**	Hide launch and copy overhead so the GPU never waits on the CPU
Device driver	**Layer 6**	Keep setup, transfer and JIT costs between process and GPU small

A MILP example: how many thread blocks fit on one SM?

One SM, two kernels resident at once. How many blocks of each?

	Shared memory	Threads	Throughput per block
Block of kernel A (x)	24 KiB	256	2 tiles / ms
Block of kernel B (y)	12 KiB	512	3 tiles / ms
The SM offers	96 KiB	1,536 slots	maximize total



maximize $2x + 3y$ **tiles / ms**

subject to $24x + 12y \le 96$ (shared memory)

$256x + 512y \le 1536$ (thread slots)

$x, y \in \{0, 1, 2, \ldots\}$ (whole blocks)

Decision variables x, y

What the solver chooses. Whole numbers, so the candidates are the dots, not the plane.

Constraints (two lines)

Each resource limit is a line that cuts the plane in half; the dots below both lines are the feasible set, 13 of 30.

Objective 2x + 3y tiles/ms

Linear, so its contours are parallel lines and the best point of the polygon is a corner: (3⅓, 1⅓). Not a real design.

Mixed-integer: the solver must return the best dot, (2, 2) = 10 tiles/ms, and a proof that no feasible dot beats it. Our template: 6 variables, 3 constraints, minimize predicted time; one-hot menus keep it linear.

One idea runs through the whole lecture

Part 1 · Every layer of the stack is a discrete optimization problem

Part 2

Search

Compilers search the space ever more aggressively: templates, cost models, reinforcement learning

Part 3

Formulate

A language model writes the problem as mathematics, and a solver optimizes it with bounds

Part 3

Verify

A verifier proves that every design the formulation admits is correct

Search harder, or formulate and prove?

What we will cover today

Part 1

The optimization stack

Seven classic kernel optimizations, then one LLM inference request walked down six software layers

Part 2

Four generations of ML compilers

TVM, Triton, TileLang and CUDA-L2: who defines the search space, and who searches it

Part 3

Formulate and verify

A language model writes the optimization problem, a solver solves it, a verifier proves it correct

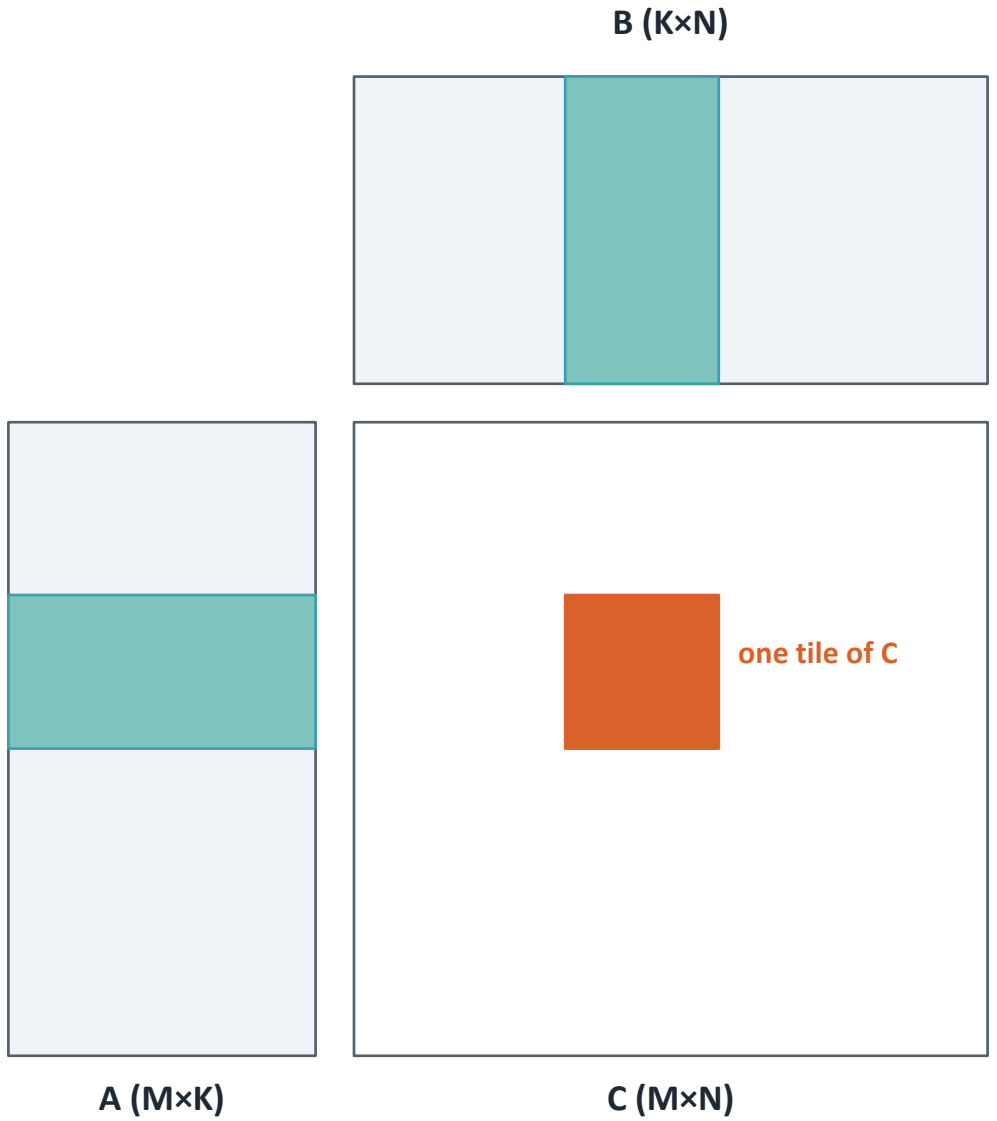
Part 1 · 35 min

The optimization stack

Running example: C = A × B

```
for (m = 0; m < M; m++)
  for (n = 0; n < N; n++) {
    float acc = 0;
    for (k = 0; k < K; k++)
      acc += A[m][k] * B[k][n];
    C[m][n] = acc;
  }
```

A is M×K, B is K×N, C is M×N. Three loops, 2·M·N·K floating-point operations.



Seven knobs (1–4): where data lives and how loops are cut

1 Tiling

Split the iteration space into blocks

Knob: tile size along M, N, K

Example: 128×128 tiles of C that fit in shared memory

2 Memory placement

Choose where each buffer lives

Knob: global, shared or registers

Example: stage tiles of A and B in shared memory for reuse

3 Loop permutation

Reorder the loop nest

Knob: one of 6 orders of m, n, k

Example: k outermost reuses A[m][k] across a row of C

4 Fusion / fission

Merge or split operations

Knob: where the boundaries go

Example: GEMM + bias + GELU, intermediate stays in registers

Seven knobs (5–7): instructions, threads and time

5 Unrolling

Replicate the loop body

Knob: unroll factor

Trade-off: more instruction-level parallelism vs. register pressure

6 Thread / block mapping

Decide which outputs each block, warp and thread computes

Knob: how many outputs per block, per warp, per thread

Example: block → 128×128 tile of C, warp → 32×64, thread → 8×8 kept in registers

7 Synchronization

Coordinate threads in time

Knob: barrier placement, async copies, pipeline depth

Example: load tile k+1 while computing tile k (software pipelining)

Every knob takes a value from a small discrete set, and the knobs interact.

A kernel is one point in a discrete space

Knob	Fixes this part of the mapping
Tiling	Spatial decomposition and parallelism
Memory placement	Where data resides
Loop permutation	Order of data flow
Fusion / fission	Granularity of computation
Unrolling	The instruction stream
Thread / block mapping	Execution binding
Synchronization	Timing and coordination

One GEMM, crude choices

$$5^3 \times 6 \times 4 \times 3 \times \ldots$$

tile sizes × loop orders × unroll factors × pipeline depths × block shapes

Tens of thousands

of points for one operator. A model has hundreds of operators, and the choices interact.

The space is rugged: small changes, big cliffs

Real performance depends on all of these at once

Occupancy	Register pressure
Shared-memory bank conflicts	Memory coalescing
Warp divergence	Instruction mix
Tensor-core eligibility	Launch geometry

A tile slightly too big does not get slightly slower.

It spills registers to local memory, which lives in HBM, and performance falls off a cliff.

Neighboring points in the space can differ by large factors.

Cliffs are constraints; slopes are mechanisms. Capacity, registers and races are boundaries a solver can be told to respect. The last 40% between a textbook kernel and cuBLAS is a dozen mechanisms of 1–7% each (pipelining depth, barrier trimming, cluster multicast, tile order) that a cost model has to learn. Part 3 returns to this.

Six software layers sit between a prompt and the GPU

Layer	Examples	Discrete decisions it makes
1 Model and serving Modules, request batches	vLLM, SGLang, TensorRT-LLM	Batching, KV-cache paging, quantization, parallelism plan
2 Graph IR Operators joined by tensors	Dynamo FX graph, TVM Relay / Relax	Fusion boundaries, layouts, pattern replacement
3 Tensor IR Loops and index expressions	TensorIR, Inductor, Triton; libraries beside it	Tile sizes, loop order, placement, unrolling, binding, pipelining
4 Hardware IR LLVM IR, PTX, SASS	LLVM NVPTX, ptxas, ROCm	Registers per thread, occupancy, instruction schedule
5 Runtime Launches, streams, memory	CUDA runtime, CUDA Graphs	Launch grouping, copy-compute overlap, buffer reuse
6 Device driver Contexts, modules, queues	libcuda, kernel-mode driver	Pinned vs. unified memory, PTX JIT, GPU sharing

The workload: serve Llama-3-8B on one H100

16 GB

FP16 weights of an 8-billion-parameter model

3.35 TB/s

H100 (SXM) HBM bandwidth

≈ 5 ms

per decode token, before any arithmetic: 16 GB ÷ 3.35 TB/s

Prefill: compute-bound

Process the whole prompt at once

Many tokens share each weight read

Limited by tensor-core throughput

Decode: memory-bound

Generate one token per step

Every weight is read again for each token

Limited by HBM bandwidth

Layer 1 · Model and serving: change the model

The model as written (a PyTorch module), wrapped in a serving engine (vLLM, SGLang) that decides what runs each step

Grouped-query attention

Several query heads share one key-value head

Smaller KV cache

Quantization

Store weights in 8 or 4 bits instead of 16

Lowers the 5 ms decode floor directly

Pruning and distillation

Remove weights, or train a smaller model to imitate

Fewer bytes and FLOPs per token

Largest payoff in the stack, but it changes the function: it needs retraining or accuracy checks (end of semester)

Layer 1 · Schedule requests around the KV cache

≈ 128 KB

of KV cache per token (Llama-3-8B, FP16)

≈ 1 GB

for one 8,000-token conversation

Continuous batching

New requests join the batch at every step

PagedAttention

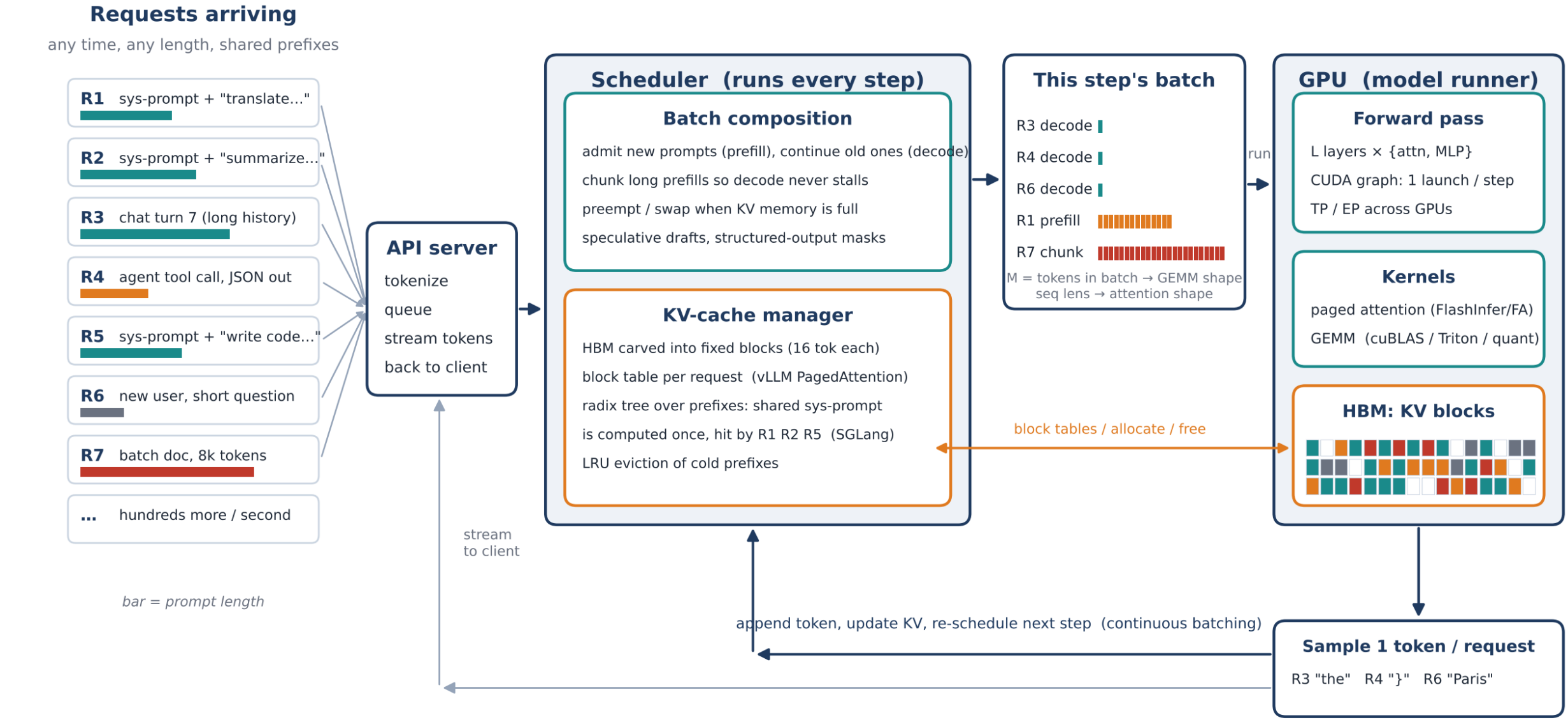
KV cache stored in fixed-size blocks, so memory does not fragment

Prefix sharing

Requests with the same prompt prefix share its KV cache

Prefill / decode interleaving

Mix compute-bound and memory-bound work on one GPU



Serving layer = KV-cache management + batch composition, repeated every step. It fixes the problem instance (M, sequence lengths, memory budget) that the compiler layers below optimize.

Prefill steps are compute-bound (large M); decode steps are memory-bound (M = number of requests). vLLM and SGLang share this structure; they differ in scheduling policy and prefix-cache design.

Layer 1 · Training adds a parallelism plan

Data parallel

Replicate the model, split the batch

Tensor parallel

Split each layer's matrices across GPUs

Pipeline parallel

Split the layers into stages on different GPUs

FSDP / ZeRO sharding

Shard parameters, gradients and optimizer state

Activation checkpointing

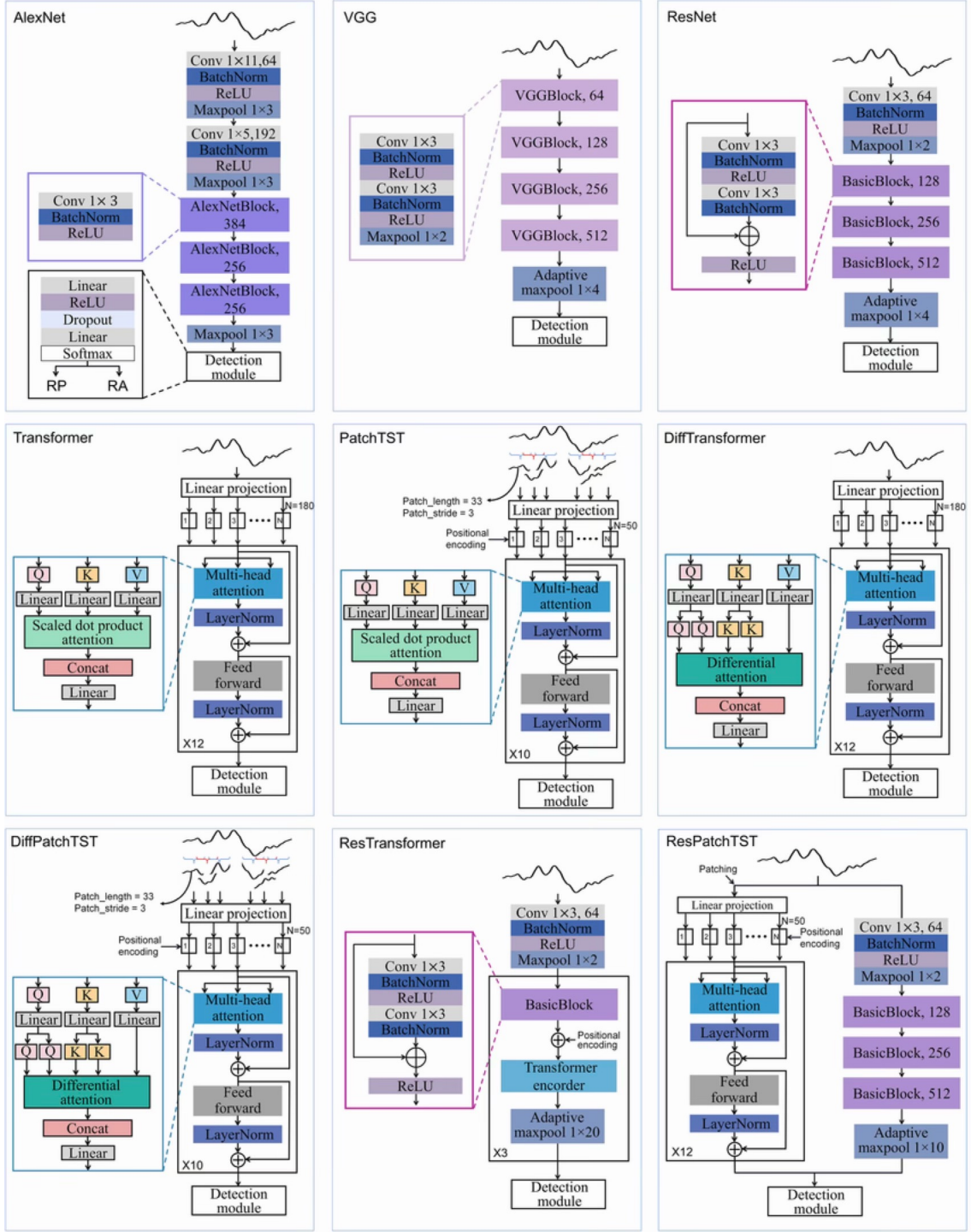
Recompute activations in backward instead of storing them

Every one is a discrete choice: how many GPUs per dimension, what to shard, what to recompute.

Layer 2 · Graph IR: the first global view of the computation

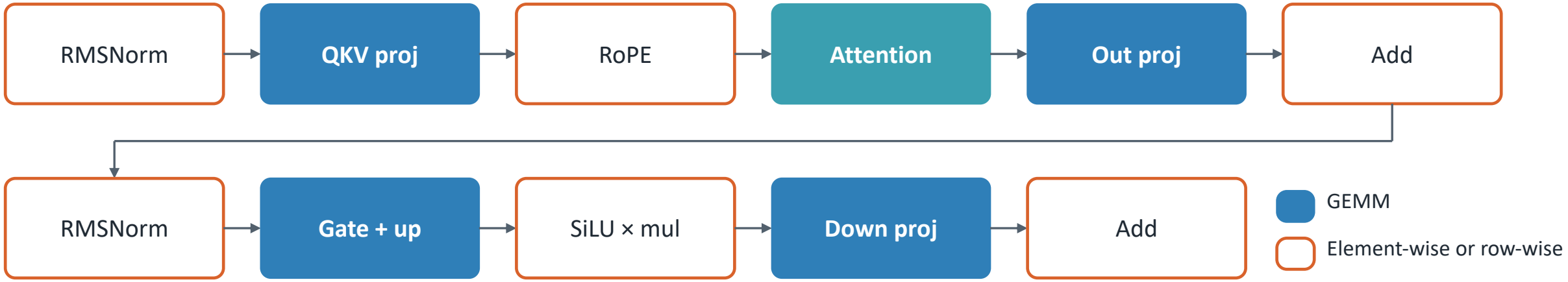
- High-level neural network definitions
- Describes function, not execution
- Organized as layers or modules
- Independent of hardware details

ALL models are computation/dataflow graph



Layer 2 · Graph IR: the first global view of the computation

One decoder layer of Llama as a graph: nodes are operators, edges are tensors

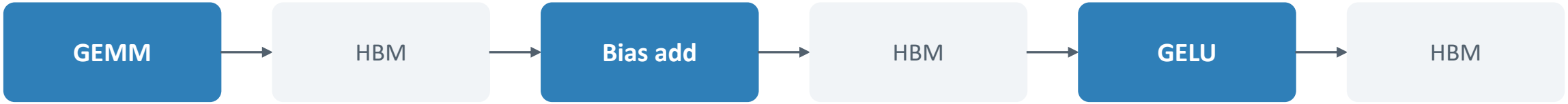


How you get it: torch.compile (Dynamo) in PyTorch 2, Relay / Relax in TVM. For training, AOTAutograd adds the backward graph.

The first layer where the compiler sees the whole computation at once

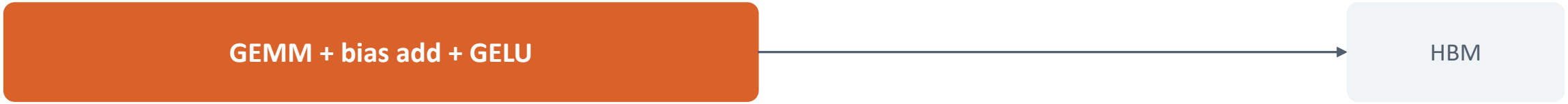
Graph IR · Fusion: keep intermediates on chip

Without fusion: every intermediate makes a round trip through HBM



3 kernels, 2 extra HBM writes and reads

With fusion: one kernel, the intermediate stays in registers



Horizontal fusion

Q, K, V projections share an input: one bigger GEMM

Pattern replacement

softmax(QK^T)V becomes one FlashAttention call

Graph IR · Classic compiler passes on the whole graph

Constant folding

Precompute whatever does not depend on the input

Dead-code elimination

Remove operators whose results are never used

Layout selection

Choose which dimension is contiguous (e.g. NCHW vs. NHWC)

Memory planning

Buffers with non-overlapping lifetimes share storage

Graph partitioning

Split the graph across devices

All discrete decisions. Choosing fusion boundaries alone is a graph-partitioning problem.

Layer 3 · Tensor IR: each operator becomes a loop nest

```
# GEMM after tiling i and j by 32
for io in parallel(ceil(M/32)):     # blocks
  for jo in range(ceil(N/32)):
    for ii in range(32):
      for ji in vectorized(32):     # SIMD
        acc = 0
        for kk in range(K):         # reduction
          acc += A[io*32+ii, kk] * B[kk, jo*32+ji]
        C[io*32+ii, jo*32+ji] = acc
```

What becomes explicit

Loop bounds and shapes

Reductions

Every memory access and its index

Where you see it

TVM TensorIR

Inductor's loop IR, emitted as Triton

MLIR linalg and affine

Same mathematics, different loop nests: the schedule picks the point in the space.

Layer 3 · Tensor IR: all seven knobs act on the loop nest

Knob	How it shows up in the loop nest
Tiling	Split i into io (tile index) and ii (within the tile)
Loop permutation	The order of io, jo, ii, ji, kk
Memory placement	Stage tiles of A and B in shared memory before the kk loop
Fusion	Bias add and GELU move into the loop that writes C
Unrolling	Replicate the body of the kk loop
Thread / block mapping	io → blockIdx, ji → threads or SIMD lanes
Synchronization	A barrier or async copy between consecutive kk tiles

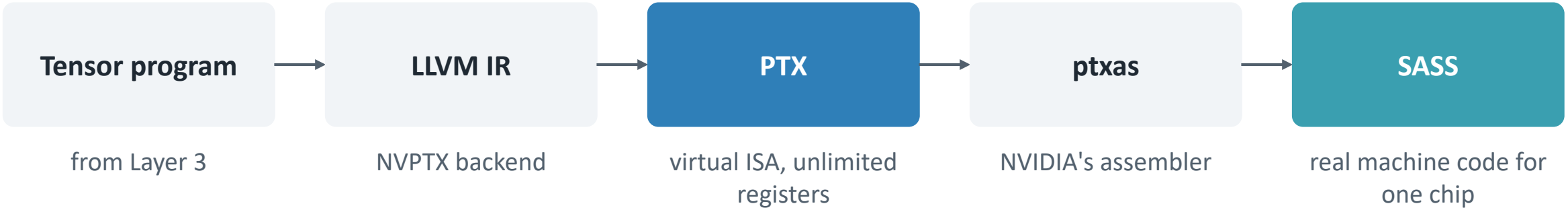
Most of the compiler search in Part 2 happens on this layer.

Layer 3 · Vendor libraries sit beside the compiler

cuBLAS / CUTLASS	cuDNN	FlashAttention	NCCL
GEMM	Convolution	Attention	Communication across GPUs (training)

Call a hand-tuned library, or generate code for this operator? That choice is a decision too.

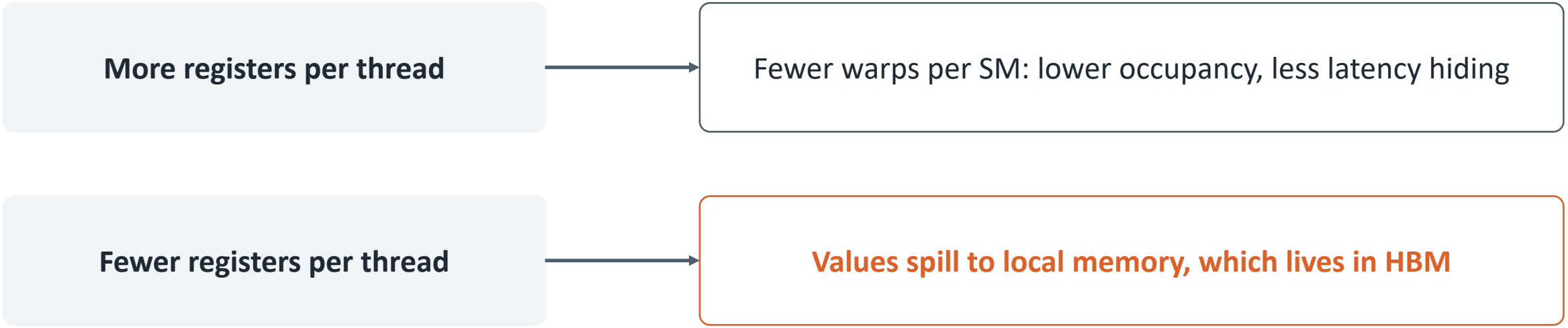
Layer 4 · Hardware IR: lowering to machine code



Triton takes a similar path through MLIR dialects. AMD uses the same structure with ROCm.

From here on, optimization is about physical resources, not algebra

Layer 4 · Hardware IR: registers per thread decide occupancy



The code generator also schedules instructions, picks tensor-core instructions and vector widths, and avoids shared-memory bank conflicts.

Programmer levers: launch bounds, maximum register count, unroll hints, inline PTX

Layer 5 · Runtime: launching hundreds of kernels per token

One decode step ≈ 10 kernels × 32 layers = several hundred launches, each costing a few µs on the CPU

Launch one by one



GPU idles while the CPU issues each launch

With CUDA Graphs



Record once, replay with a single launch

Other runtime knobs

Overlap copies with compute on streams

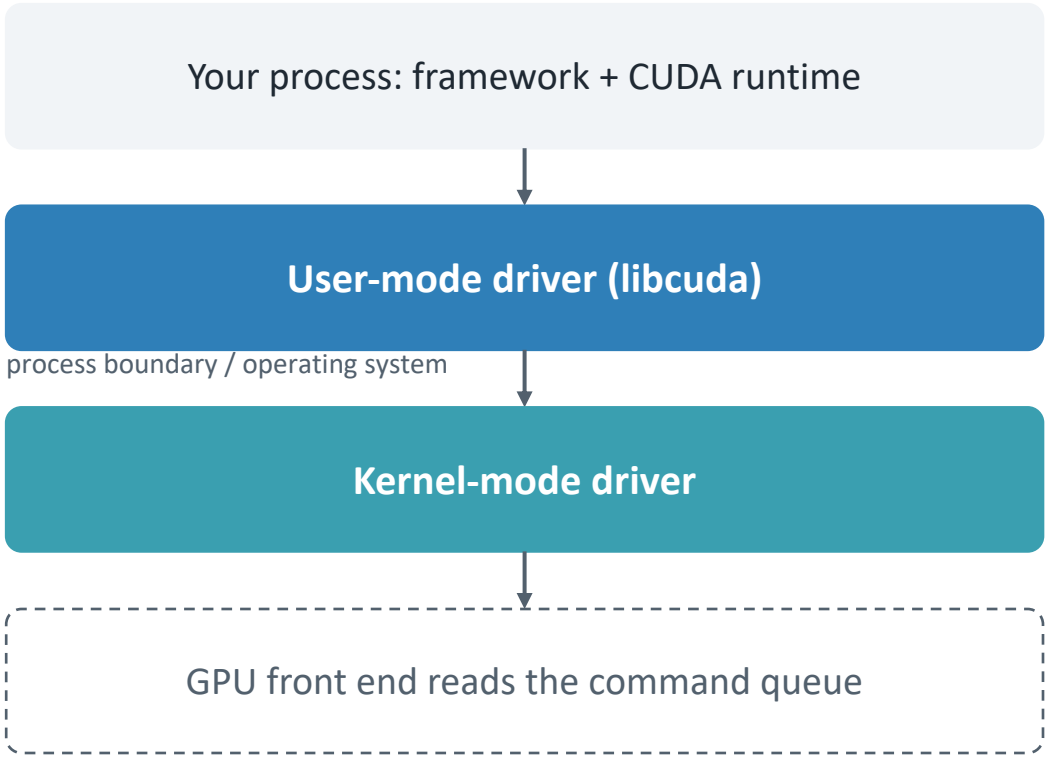
Reuse buffers across steps

Fewer CPU–GPU synchronizations

Training: overlap all-reduce with backward

Illustration, not to scale

Layer 6 · Device driver: the last software before silicon



What the driver does

Creates a context per process

Loads compiled modules onto the GPU

Manages GPU virtual memory and page tables

JIT-compiles PTX when no matching SASS exists

Turns launches into queued commands

Knobs that still matter

Pinned host memory for faster transfers

Unified memory vs. explicit copies

MPS or MIG to share one GPU; cache JIT results

Recap: every layer is a search problem

Layer	Representation	Discrete decisions
1 Model and serving	Module, request batches	Batch size, KV block size, admission, parallel degrees
2 Graph IR	Operator graph	Fusion boundaries, layouts, replacements
3 Tensor IR	Loop nest	Tile sizes, loop order, placement, binding, pipelining
4 Hardware IR	PTX, SASS	Register budget, instruction schedule, vector width
5 Runtime	Launch sequence	Launch grouping, streams, buffer reuse
6 Device driver	Command queue	Memory mode, JIT caching, GPU sharing

An optimization is only possible where its structure is visible.

Who makes these decisions, and how?

Experts

write libraries by hand

cuBLAS, cuDNN

→

Compilers that search

templates, cost models, RL

Part 2

→

Machines that formulate and prove

LLM + solver + verifier

Part 3

Part 2 · 30 min

Four generations of ML compilers

Two eras, two different questions

Before LLMs (2017–2020)

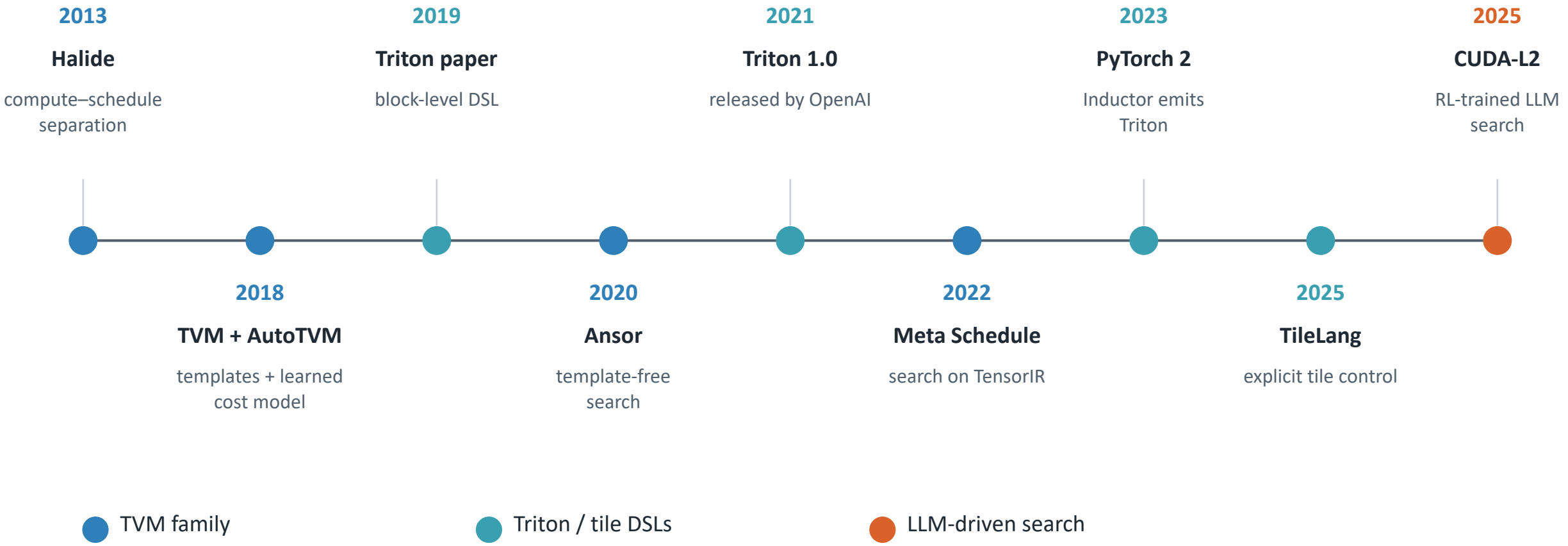
Models: CNNs, RNNs, GANs, early Transformers
Hardware: NVIDIA, TPU, ARM, FPGAs, Ascend, Graphcore
Question: "Write once, run fast everywhere?"
Answer: TVM: portable, compiler-driven search

The LLM era (2020–)

Models: converged on the Transformer
Hardware: NVIDIA CUDA, Tensor Cores, NVLink became standard
Question: "The last 20% on one chip?"
Answer: Triton, TileLang, then LLM-driven search

The tools changed because the question changed.

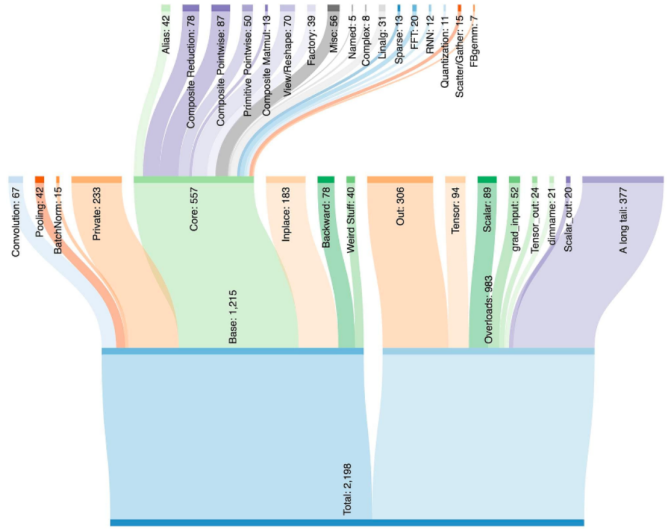
A decade of ML compilers at a glance



TVM: the problem it solved

	NVIDIA	TPU	ARM	FPGA	Ascend
Conv	■			■	
GEMM			■		
Attention		■			■
ReLU	■			■	
Norm			■		
Add		■			■

Each cell: a kernel tuned by hand, then maintained as models change



Hundreds × tens

operators × hardware backends

TVM's promise

Write each operator once

Let the compiler generate fast code for every target

University of Washington, OSDI 2018

Compute–schedule separation: what vs. how

What to compute

```
C = te.compute((M, N),
  lambda i, j: te.sum(A[i, k] * B[k, j],
                      axis=k))
```

How to compute (the schedule)

```
s = te.create_schedule(C.op)
i, j = s[C].op.axis
io, ii = s[C].split(i, factor=32)
jo, ji = s[C].split(j, factor=32)
s[C].reorder(io, jo, ii, ji)
s[C].parallel(io)
s[C].vectorize(ji)
```

The loop nest it produces

```
for io in parallel(M/32):
  for jo in range(N/32):
    for ii in range(32):
      for ji in vectorized(32):
        acc = 0
        for k in range(K):
          acc += A[..] * B[..]
        C[..] = acc
```

The compute fixes the meaning; a legal schedule changes only the speed

A schedule is a point in the Part 1 space, written as code

Schedule primitive	Knob from Part 1	Example
split	Tiling	split(i, factor=32)
reorder	Loop permutation	reorder(io, jo, ii, ji)
cache_read / compute_at	Memory placement, fusion	stage A, B tiles in shared memory
unroll	Unrolling	unroll(ki)
bind / parallel	Thread / block mapping	bind(io, blockIdx.x)
vectorize	Vector width	vectorize(ji)

A schedule assigns a value to every knob, so a program can search over schedules.

Relay: TVM's graph IR, with one worked example

Relay runs the Layer 2 passes: fusion, layout transformation, memory planning, constant folding

Example: folding BatchNorm into the convolution (inference)



$$BN(y) = \gamma\,(y - \mu) / \sqrt{(\sigma^2 + \varepsilon)} + \beta = s \cdot y + t, \quad \text{with } s = \gamma / \sqrt{(\sigma^2 + \varepsilon)},\ t = \beta - \mu \cdot s \text{ fixed per channel}$$

Fold into the convolution: $W' = s \cdot W,\ b' = s \cdot b + t$

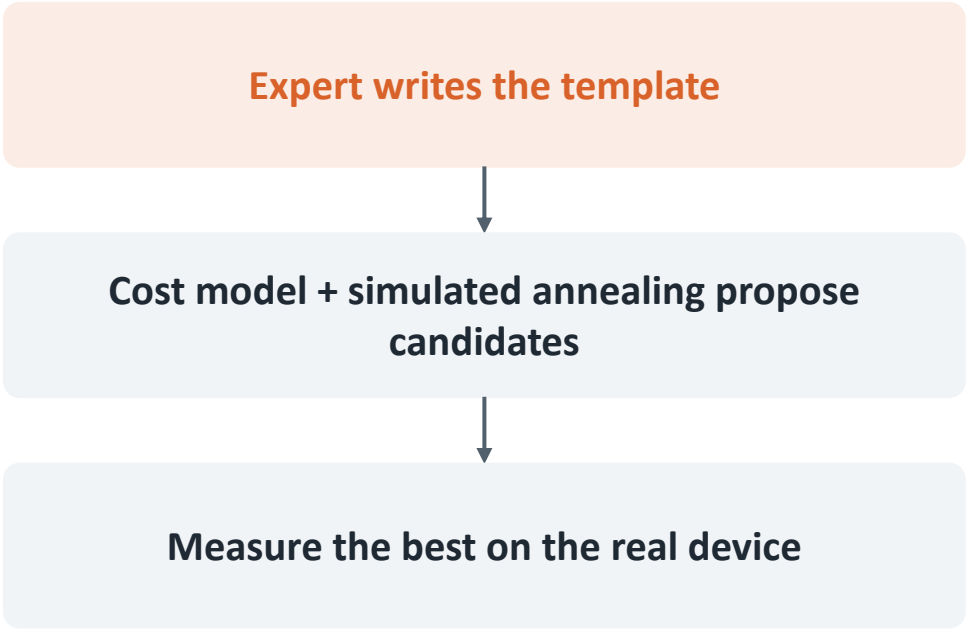
Result: one convolution with a fused ReLU. No BatchNorm kernel runs at all.

AutoTVM: humans define the space, the machine searches

A template with holes, written by an expert

```
# AutoTVM template (sketch)
cfg.define_split("tile_i", i, num_outputs=2)
cfg.define_split("tile_j", j, num_outputs=2)
cfg.define_knob("unroll", [0, 1])
io, ii = cfg["tile_i"].apply(s, C, i)
jo, ji = cfg["tile_j"].apply(s, C, j)
```

Each define_split / define_knob is a hole the machine fills



Human defines the space. Machine searches it.

Ansor and Meta Schedule: the machine defines the space too

Ansor (2020)

Space: rules generate program sketches; no templates
Search: evolutionary search + learned cost model

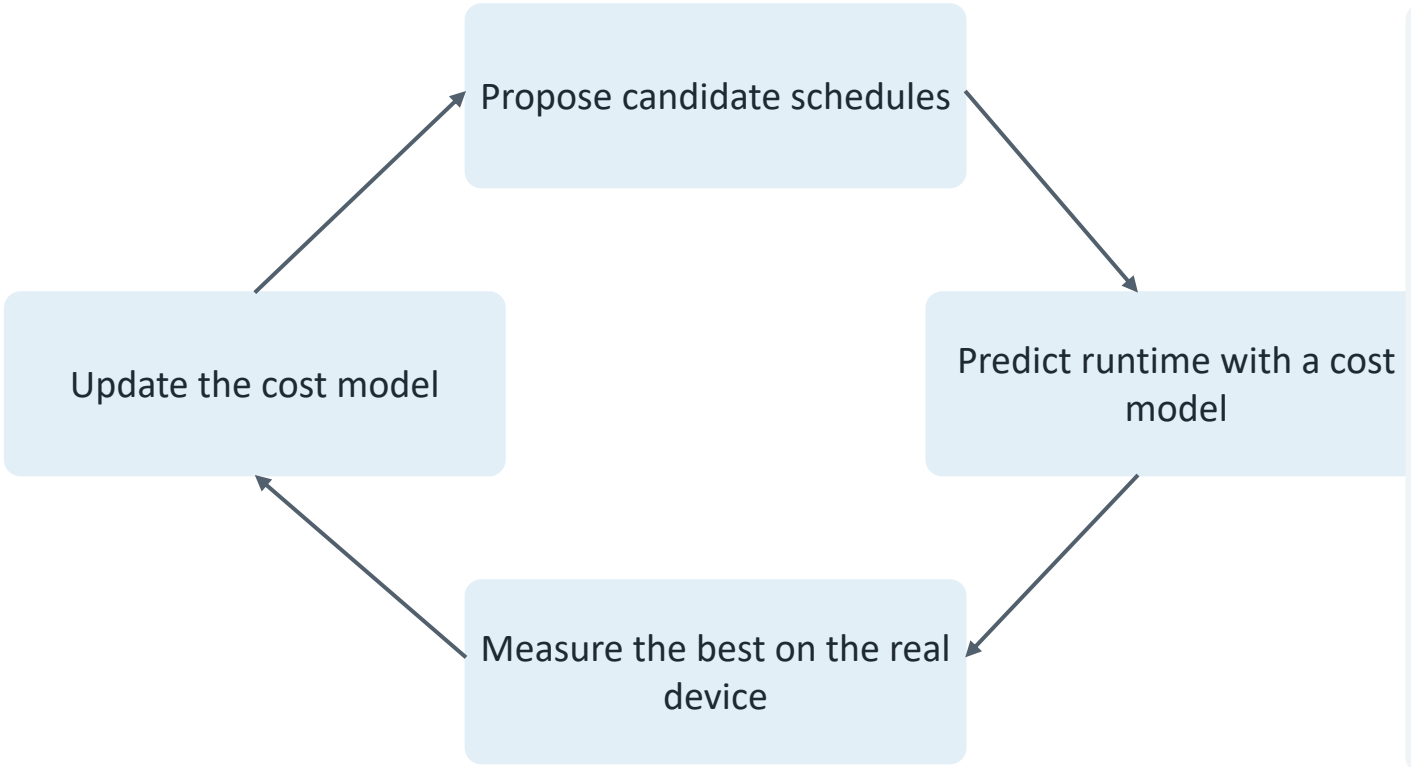
Meta Schedule (2022)

Space: a probabilistic program over TensorIR transformations
Search: evolutionary search + learned cost model, validated on hardware

Who defines the space: expert template → generation rules → probabilistic program

Not reinforcement learning: evolutionary search guided by a learned cost model.

The search loop: sample, predict, measure



What you get

Objective: measured runtime
Method: sample, predict, measure
Cost: hours of tuning per model per device
Guarantee: best found within the budget
A black box: no reason why, no bound on how far from best

Triton: the bottleneck became writing kernels, not porting them

	CUDA	Triton
Memory (global / shared / registers)	Manual	Automatic
Parallelism	Blocks, warps, threads	Mostly blocks
Tensor cores	Manual	Automatic
Vectorization (.8 to .128)	Manual	Automatic

"One Triton program is roughly one CUDA block."

The deal Triton offers

You write: one program instance per block of data

The compiler handles: threads, shared memory, coalescing, tensor cores, pipelining

Python syntax, CUDA-level performance for many kernels

The same vector add in CUDA and in Triton

CUDA

```
__global__ void vec4_add(float* a,
                         float* b,
                         float* c) {
  int idx = (threadIdx.x +
             blockIdx.x * blockDim.x) * 4;
  float4 ra = FETCH_FLOAT4(a[idx]);
  float4 rb = FETCH_FLOAT4(b[idx]);
  float4 rc;
  rc.x = ra.x + rb.x;  rc.y = ra.y + rb.y;
  rc.z = ra.z + rb.z;  rc.w = ra.w + rb.w;
  FETCH_FLOAT4(c[idx]) = rc;
}
```

Triton

```
@triton.jit
def add(x_ptr, y_ptr, z_ptr, N,
        BLOCK: tl.constexpr):
  pid = tl.program_id(0)
  offs = pid * BLOCK + tl.arange(0, BLOCK)
  mask = offs < N
  x = tl.load(x_ptr + offs, mask=mask)
  y = tl.load(y_ptr + offs, mask=mask)
  tl.store(z_ptr + offs, x + y, mask=mask)

grid = (triton.cdiv(N, 1024),)
add[grid](x, y, z, N, BLOCK=1024)
```

CUDA: thread index, vector width, memory access and registers by hand. Triton: one program per block; threads are hidden.

How Triton compiles, and how little it searches



@triton.jit: specialize on types and constants, lower through MLIR, cache the binary, launch

The whole search: a short list a human writes

```
@triton.autotune(configs=[
  triton.Config({"BLOCK": 128}, num_warps=4),
  triton.Config({"BLOCK": 256}, num_warps=8),
  triton.Config({"BLOCK": 128}, num_stages=3),
], key=["N"])
```

Where Triton lands

Good at: element-wise ops, norms, softmax, FlashAttention-style kernels
Reach: PyTorch 2 Inductor emits Triton by default
Struggles with: exotic layouts, deep pipelines, warp specialization

TileLang: give the expert the knobs back

TileLang construct	Knob it exposes
T.alloc_shared, T.alloc_fragment	Memory placement
T.copy	Data movement between levels
T.gemm	Tensor-core tile computation
T.Pipelined(num_stages=…)	Synchronization, pipeline depth
T.Parallel, kernel launch shape	Thread / block mapping
Layout annotations (swizzle, pad)	Placement without bank conflicts

Why go back to explicit control?

Complex kernels: multi-head latent attention, INT4 / INT3 GEMMs, Hopper warp specialization

The hidden decisions are where the remaining performance lives

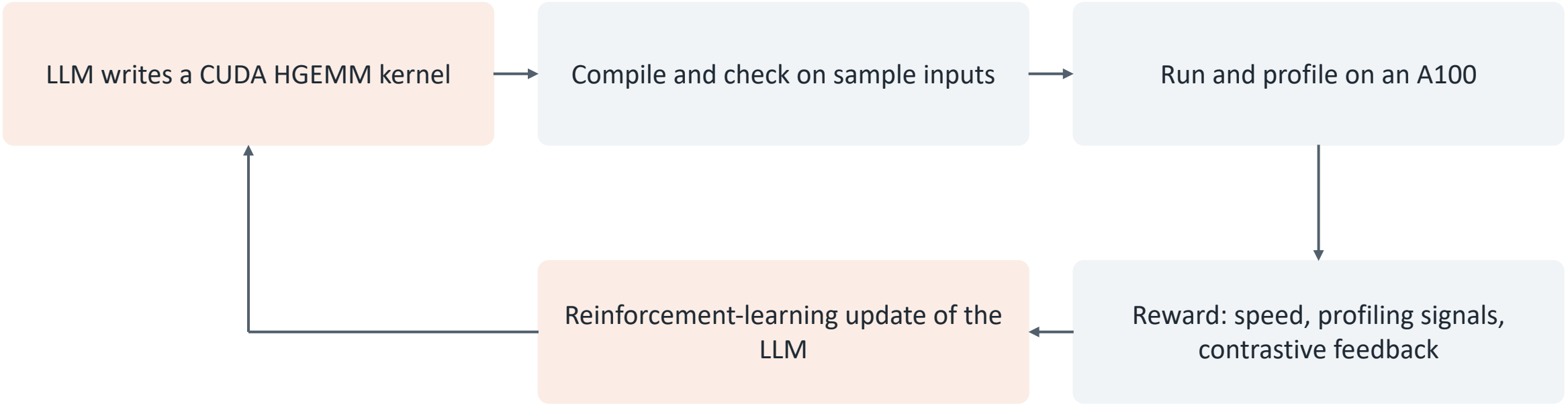
TileLang moves back toward the human: the opposite direction from Meta Schedule

Triton vs. TileLang: productivity vs. a higher ceiling

	Triton	TileLang
Memory layout	Compiler decides	Explicit tiles, swizzle, padding, alignment
Pipelining	User sets num_stages only	Explicit depth, buffers, load/compute order
Thread binding	One program ≈ one CUDA block	Explicit warps; warp specialization
Debugging	Transformations obscure the PTX	Decisions are visible in the code
Best for	Standard ops, fast prototyping	Complex attention, low-bit kernels

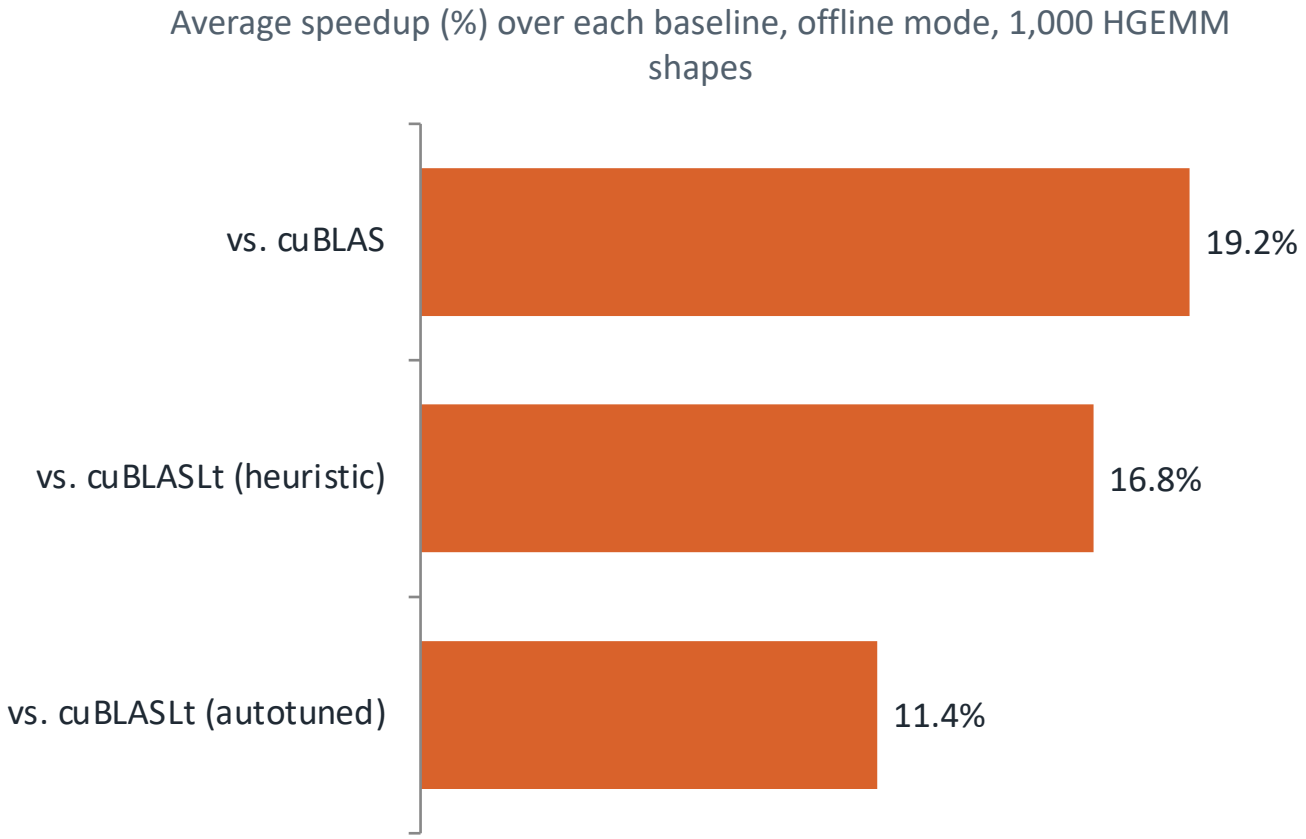
Experts want the knobs back: a sign the space is too hard to search blindly.

CUDA-L2: let a language model search



Search space: all CUDA programs. Searcher: the model itself. Evaluated on 1,000 (M, N, K) shapes.

CUDA-L2 beats NVIDIA's own libraries on half-precision GEMM



22–28%

faster than torch.matmul

1.3–1.4×

on small and mid-sized shapes; near parity on large shapes that saturate the GPU

Source: CUDA-L2 (arXiv:2512.02551), 1,000 M×N×K shapes; 22% offline and 28% server mode over torch.matmul

What it discovered: known knobs, values no human tried

Pad to unlock a better tile

M = 8192 → 8320 rows (+1.6% FLOPs) so a 160-row tile fits; about 15% faster than autotuned cuBLASLt

↓

Knob 1: tiling

Prefetch four tiles ahead

K+4 lookahead instead of the usual K+1, hiding A100's async-copy latency

↓

Knob 7: synchronization, pipeline depth

Swizzle stride per shape

No swizzle for small GEMMs, small strides for medium, large strides for large

↓

Knob 6: thread / block mapping

Known knobs set to values humans never tried.

The honest limits of searching harder

Tested, or proved one kernel afterwards

Correctness is checked on sample inputs, and the check can be gamed: a tensor of zeros passes 4 of 60 KernelBench level-1 tasks. The 2026 verifiers (ProofWright, Volta, Kuiper) prove one finished kernel after search, and the kernels that can be proved run slower.

A black box that does not transfer

It cannot say why a choice is right or how far it is from the best. Agents that explain every step beat the reference on 87% of tasks on the GPU they tuned on, and on 25% when the same kernels run on another chip (KernelGenBench).

Expensive and narrow

About 5 million tokens per operator. On transformers, 80–86% of runtime is already cuBLAS or FlashAttention, so the realistic end-to-end gain from better generated kernels is 1.3–2.5%.

An unstructured space

"All CUDA programs." The model rediscovers tiles, stages and swizzles that we can already write down, and pays for the rediscovery on every new shape and chip.

The field: 30+ LLM-for-kernels systems in six clusters

Post-trained kernel models

AutoTriton, CUDA-L1, CUDA-L2, Dr. Kernel

Agentic generate → profile → refine

CUDA-LLM, AutoKernel, StitchCUDA, KernelFalcon

Profiling-aware optimization

SwizzlePerf, PRAGMA, KernelBand

Evolutionary, population-based

GPU Kernel Scientist, cuPilot, K-Search, KernelFoundry

Beyond CUDA

KernelEvolve, KernelCraft, CuTeGen, QiMeng

Benchmarks

KernelBench and newer agent settings

Who defines the space, and who searches it?

Generation	Defines the space	Searches it	Correctness	Optimality
Libraries (cuBLAS, cuDNN)	Expert	Expert	Expert testing	Nothing stated
AutoTVM (2018)	Expert template	Cost model + annealing	By construction	Best in budget
Ansor / Meta Schedule	Generation rules	Evolutionary + cost model	By construction	Best in budget
Triton (2019–21)	Programmer + compiler	Short autotune list	Programmer's job	Best of the list
Helion (2025)	Programmer: tiled loops	Evolutionary search, ~1,500 configs	By construction	Best found
TileLang (2025), Gluon / CuTile (2026)	Expert, explicit knobs	Expert + autotune	Programmer's job	Best of the list
CUDA-L2 (2025)	LLM: all CUDA programs	RL, runtime reward	Tested on samples	Best found
Twill (NVIDIA/Stanford, 2025)	Expert: ops, units, latencies	Exact SMT over pipelining and warp roles	Schedule legal by construction	Optimal in the model
Ours (Part 3)	**LLM writes the formulation**	**Exact solver (MILP)**	**Family formally verified**	**Optimal in the model, with a bound**

The more powerful the search, the less we know about correctness and distance from the best. Twill is the closest prior work: an exact formulation, written by hand, that rediscovers the FlashAttention-4 schedule within 1%.

Part 3 · 20 min

Formulate and verify, layer by layer

Why not just write one giant optimization problem?

Today: one layer at a time

Today's stack (Part 1)

✓ Solvable

✗ Loses the cross-layer interactions that matter most

One monolithic MILP

Every choice at once

✓ Keeps every interaction

✗ Too large and too weakly structured to solve

A discovered formulation

What we want

✓ Keeps the interactions that matter

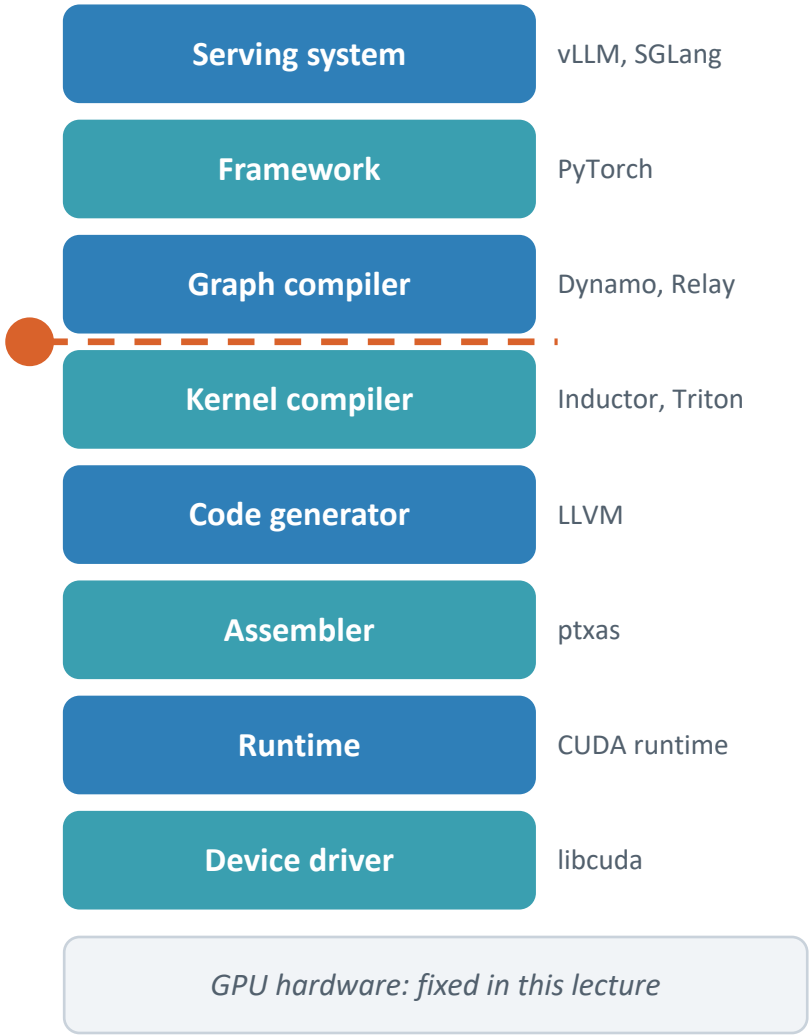
✓ Small enough to solve

The hard part is not solving the problem. It is finding a formulation worth solving.

That is a reasoning task, and reasoning is what language models are good at.

The stack of Part 1, seen as refinement steps

The software stack (Part 1)



Today: one layer at a time

Each layer is optimized on its own, by its own team, against a frozen interface to the layers above and below.

It commits its decisions before the layers below have made theirs: a tile is chosen before the registers are allocated.

The decomposition itself was chosen by people, once, in advance. Cross-layer choices are never on the table.

Ours: one step per boundary

LLM writes a template for the decisions this boundary opens
Solver optimizes inside the template, with bounds
Verifier proves every admitted design still computes what the layer above demanded
Nothing is discarded between layers: a decision is reopened when a lower layer shows it was wrong.

Next slide: GEMM tiling is this step, from the graph compiler's fused operator to the kernel compiler's tiles in shared memory. The same contract applies at every other boundary.

Anatomy of a template: six parts

1 Scope Γ

The fixed context of this step: the workload (computation, sizes, precision), the target hardware (resource limits and behavior), and the refinement the template performs.

Here: FP32 GEMM via shared-memory tiles

read by: agent

2 Decisions X

The variables the template sets, each with a finite set of allowed values. Searched ones are chosen by the solver; frozen ones are fixed by the template.

Here: Tm, Tn, Tk, S, P, β

read by: solver

3 Constraints C

The conditions a choice must satisfy to be legal: resource limits and safety rules. The legal designs form the feasible set.

Here: fits on the chip; $P \le S - 2 + \beta$

read by: solver

4 Cost Ĵ

A function of the decisions and the scope that predicts how good a legal design is. The solver minimizes it over the feasible set.

Here: predicted run time T

read by: solver

5 Correctness

The properties every legal design must have to compute exactly the workload, the conditions each relies on, and how each is proved.

Here: every design computes $C = A \cdot B$

read by: verifier

6 Contract K1–K3

Three conditions linking search and proof: constraints plus frozen choices imply the correctness conditions (K1), which guarantee correctness (K2); a legal design exists (K3).

Here: every pick is proven correct

read by: all

The language model writes all six parts; the solver and the verifier are bound by them.

1 · Scope Γ: what is given

```
refinement              what this step does
  GEMM → GEMM tiled through shared memory:
  a block computes one Tm×Tn tile of C in KT phases;
  each phase stages a Tm×Tk tile of A and a Tk×Tn
  tile of B in one shared-memory slot
workload                 defines "correct"
  C[M,N] += A[M,K]·B[K,N]    any M,N,K ≥ 1
  FP32, e = 4 bytes per element, FP32 accumulation
chip limits    illustrative, not measured
  N_SM    = 128          streaming multiprocessors (SMs)
  M_blk   = 96 KiB       shared memory one block may use
  R_SM    = 65,536       registers per SM
  r_max   = 255          registers per thread
  F_SM    = 625 GFLOP/s  FP32 rate of one SM
  B_DRAM = 1.8 TB/s      DRAM bandwidth
  t_bar   = 40 cycles    cost of one block-wide barrier
hardware semantics    assumed by the proofs
  barrier: no thread passes until all arrive;
           shared-memory writes before it are visible
  load:    completes before the next barrier
```

Definition

The scope is the fixed context of this step: the workload to compute, the hardware it runs on, and the refinement this template performs. If the solver could choose it, it is a decision, not scope.

Refinement	says what this step adds
Workload	decides what is correct
Chip limits	decide which designs exist and how fast they are
Semantics	are what the correctness proof stands on

Scope is relative to the step: in a co-design step, shared memory or registers move from scope into decisions.

2 · Decisions X: what this template chooses

Definition: the decisions are the variables this template sets, each with a finite set of allowed values. Searched ones are chosen by the solver, frozen ones are fixed by the template, derived ones follow from the others.

Searched: the solver picks one value from each menu

Variable	Menu	Meaning	Part 1 knob
Tm, Tn	16, 32, 64, 128, 256	Rows and columns of C that one thread block computes	Tiling
Tk	8, 16, 32, 64	Columns of A and rows of B consumed in one phase	Tiling
S	1, 2, 3	Shared-memory slots per block (each holds one A tile and one B tile)	Memory placement
P	0, 1, 2	Prefetch distance: phases loaded ahead of the one being computed	Synchronization
β	0, 1	Trailing barrier after each compute phase (1 = kept)	Synchronization

5 × 5 × 4 × 3 × 3 × 2 = 1,800 combinations

Frozen ≠ given: a later template may unfreeze them, and some correctness properties depend on them.

Frozen: chosen by the template, not searched

```
n_t   = 256   threads per block
a_max = 128   accumulator registers/thread
grid  = ⌈M/Tm⌉ × ⌈N/Tn⌉ blocks
tile stride = tile size (no overlap)
guard: skip any index past the
       matrix edge
leading barrier before each compute
```

Derived: follow from the above

$a = T_m T_n / n_t$ outputs each thread accumulates
$KT = \lceil K/T_k \rceil$ phases of the K loop
$G_m \times G_n = \lceil M/T_m \rceil \times \lceil N/T_n \rceil$ blocks in the grid

3 · Constraints C: what the solver must respect

Definition: the constraints are the conditions a choice must satisfy to be legal. C1 and C2 are resource limits (the design fits on the chip); C3 is the only correctness constraint. Feasible set: 722 of 1,800.

C1 $S \cdot e \cdot (T_m T_k + T_k T_n) \le M_{blk}$ resource limit: slots fit in shared memory
C2 $1 \le T_m T_n / n_t \le a_{max}$ resource limit: 1 to 128 outputs per thread
C3 $P \le S - 2 + \beta$ correctness: protects lifetime
register checks, true for every choice: a_{max} + address and loop registers ≤ r_{max};
$n_t \cdot r_{max} = 65{,}280 \le R_{SM}$, so one block always fits in the register file

Slots S	β = 1	β = 0
1	P = 0	none
2	P ≤ 1	P = 0
3	P ≤ 2	P ≤ 1

Why C3: count the slots in the ring (S = 3, no trailing barrier)

slot 1: tile p
computing now

slot 2: tile p − 1
a slow warp may still read it

slot 3: tile p + 1
free: load ahead (P = 1)

A trailing barrier (β = 1) forces every warp to finish tile p − 1, freeing that slot: then P can be 2.

Check: 1000³ GEMM design

Tm = Tn = 128, Tk = 32, S = 3, P = 1, β = 0

C1: 3 × 4 × 8,192 B = 96 KiB ≤ 96 KiB (tight)

C2: 16,384 / 256 = 64 outputs per thread

C3: 1 ≤ 3 − 2 + 0 (tight)

4 · Cost Ĵ: what "best" means

Definition: the cost is a function of the decisions and the scope that predicts how good a legal design is. Here it predicts run time T, and the solver minimizes it over the feasible set.

minimize T the predicted run time of the whole GEMM

$W = \lceil G_m G_n / N_{SM} \rceil$

waves: the grid has $G_m G_n$ blocks and one block runs per SM at a time, so the chip needs W rounds of blocks

$T_{comp} = W \cdot (2T_m T_n K / F_{SM} + KT \cdot (1+\beta) \cdot t_{bar})$

compute time: one block's $2 \cdot T_m T_n K$ flops at one SM's rate, plus KT phases × (1+β) barriers × t_{bar}, times W waves

$T_{mem} = e \cdot (G_m G_n K(T_m + T_n) + MN) / B_{DRAM}$

memory time: every block reads $K(T_m+T_n)$ elements of A and B, so all blocks read $G_m G_n K(T_m+T_n)$; C is written once, MN; × e bytes ÷ bandwidth

$T = \max(T_{comp}, T_{mem})$ if $P \geq 1$; $T = T_{comp} + T_{mem}$ if $P = 0$

with prefetch, loads overlap compute and the slower side sets the time; without prefetch (P = 0) they add

Model assumptions: they steer the search, they are never proved

One block per SM at a time · no L2 reuse (every block reads its A and B strips from DRAM) · edge tiles charged as full tiles

Solver form: each menu becomes one-hot binary variables and the products are linearized exactly → MILP with 93 variables, 248 constraints; HiGHS finds the optimum in < 0.2 s, and brute force over the 722 legal designs agrees.

Symbols: Tm, Tn, Tk, S, P, β are the searched decisions (slide 2); Gm, Gn, KT are derived from them; N_SM, F_SM, B_DRAM, e, t_bar are the chip limits (slide 1); M, N, K are the workload. Illustrative chip; times are modeled, not measured.

1000³ GEMM: T = 53.1 µs, compute-bound

128×128 tiles → 8×8 = 64 blocks ≤ 128 SMs, so 1 wave; KT = 32 phases; T_comp = 52.4 + 0.7 (barriers) = 53.1 µs, T_mem = 38.6 µs

Alternative	T (µs)	Why it loses
64×128 tiles	56.8	More re-reads of A, B
Keep barrier	53.9	2× barriers (+1.34%)
Tk = 16	53.8	2× phases
Tk = 64	91.8	Only S = 1 fits: no prefetch
128×256 tiles	106.3	Only 32 blocks

The winner leaves half the SMs idle: the model ignores L2. The solver is exact; the model may not be.

5 · Correctness: what must hold for every design

Definition: correctness is the set of properties every legal design must have to compute exactly the workload, the condition each relies on, and who guarantees that condition.

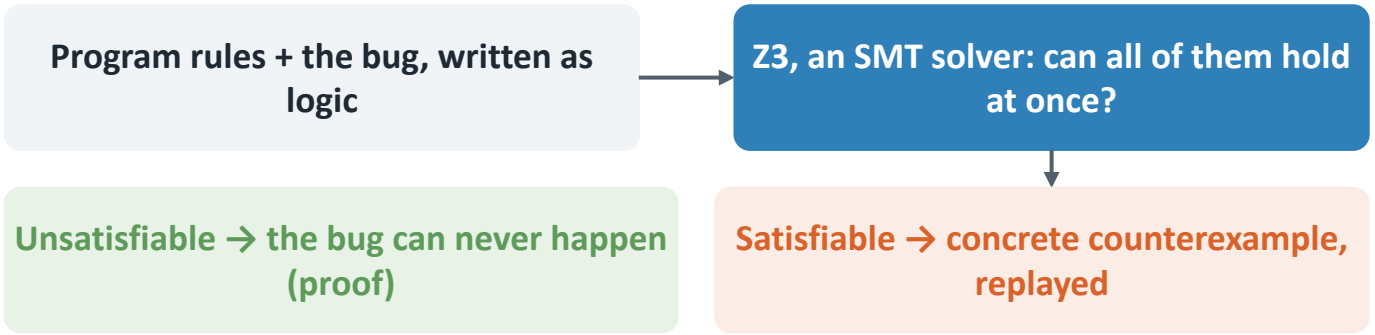
Property	Means	Relies on	Guaranteed by
Lifetime	No slot overwritten before its last read	$P \le S - 2 + \beta$	Constraint C3
Readiness	No slot read before its load completes	Leading barrier	Frozen
Coverage	Every output element computed	Grid rounded up	Frozen
Exact-once	Each product added exactly once	Tiles do not overlap	Frozen
No-phantom	No access past a matrix edge	Guard on every index	Frozen
Agreement	Result equals C = A·B	All of the above	Proof (next slide)

Only lifetime depends on searched decisions (S, P, β), so only it needs a constraint (C3).
Readiness, coverage, exact-once and no-phantom are guaranteed by frozen choices; agreement follows from all of them.

```
program  (meaning of a design)
for each block (i, j):
  for u in 0..P-1: load tile u → slot u%S
  for p in 0..KT-1:
    if p+P < KT: load tile p+P → (p+P)%S
    barrier                    # leading
    C[i,j] += slot p%S         # guarded
    if β: barrier              # trailing
backward map π
  forget grid, slots, schedule ⇒ C = A·B
certificate Σ
  R  relation: slot u%S holds tile u from
     its load completion to its last read
  ρ  rank: tiles not yet consumed; it falls
     every phase, so the program finishes
```

6 · How the verifier proves it: search for a bug, find none

Definition: to verify a property is to prove that no legal design can violate it, for every matrix size and every timing in the model, by asking a solver to find a violation.



Example: no-phantom, for every matrix size E

variables: $E \geq 1$, block $b \geq 0$, offset $0 \leq o < \text{tile}$

program: $\text{index} = b \cdot \text{tile} + o$, guard: $\text{index} < E$

bug: $\text{index} \geq E$

guard and bug contradict for every E → unsatisfiable → proved

Schedule properties (lifetime, readiness): each load, barrier and compute gets a time variable, ordered only by program order and the barrier rule; Z3 searches every interleaving of a model with 2 warps and 5 phases.

If the LLM writes a weaker C3

$C3'$: $P \leq S - 1$

Solver picks S = 2, P = 1, β = 0: same predicted time, less shared memory

Z3 finds a timing: a fast warp loads tile 4 into slot 0 while a slow warp still needs tile 2 from it

Replay: C[0,1] gets A[0,2]·B[4,1] instead of A[0,2]·B[2,1]

Template rejected; the counterexample goes back to the agent.

7 · Trust: one proof per template, a cheap check per solution

Definition: the contract is three conditions, checked once per template, that link search and proof. Then each solution needs only four cheap checks.

K1 **Constraints + frozen choices ⇒ preconditions**

C3 gives the lifetime precondition; the frozen choices give the other four

K2 **Preconditions ⇒ correct**

All six properties proved for the whole family; C3 is exact on all 24 checked (S, P, β) combinations (P up to 3), so it is the weakest sound rule

K3 **Not empty**

Tm = Tn = 16, Tk = 8, S = 1, P = 0, β = 1 is legal for every GEMM shape in scope

Don't trust the solver or the compiler. Trust a small checker.

Per solution: four cheap checks

✓ **In scope?**

FP32 GEMM; chip limits and barrier behavior match

✓ **Decodes cleanly?**

Exactly one value per menu: 128, 128, 32, 3, 1, 0

✓ **Constraints hold?**

C1 96 ≤ 96 KiB · C2 64 outputs/thread · C3 1 ≤ 1

✓ **Frozen choices kept?**

Generated code has leading barrier, ceil grid, guard, disjoint tiles

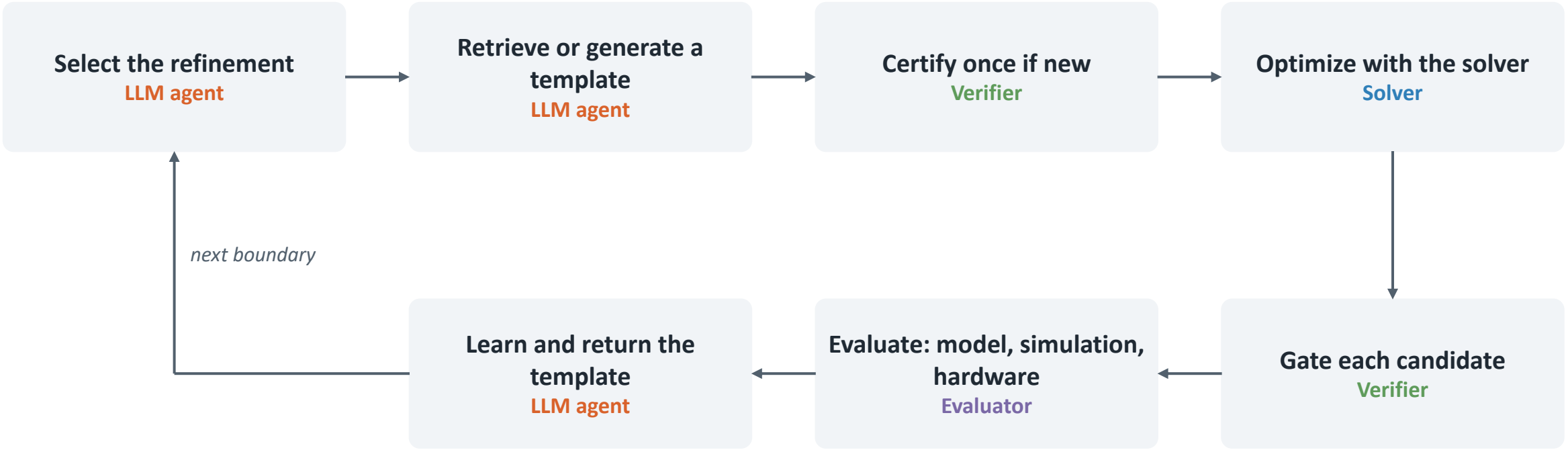
Why still check? Solver tolerances and big-M encodings, translation bugs, and code generation can each break the link to the proof.

Progressive refinement: one small step per layer

Refinement step	New object the LLM formulates	Solver decides	Verifier proves
Graph (Layer 2)	Fusion groups, layouts	Partition of the operator graph	Same function as the unfused graph
Tensor program (Layer 3)	Tiles in on-chip memory	Tile shapes, loop order, placement	Coverage, exact-once, no-phantom
Data movement (3–4)	Async transfers, buffer slots	Pipeline depth, prefetch distance	Lifetime: no live slot overwritten; no deadlock
Execution binding (4)	Warps, registers, schedule	Mapping, register budget	Readiness: operands ready before use; agreement
Chip resources (research version: the chip is a decision too)	Banks, ports, queue depths	Capacities, jointly with software	Requests served without conflict

Each step adds a little structure, so both search and verification stay tractable.

The loop: formulate, certify, optimize, gate, evaluate, learn



Why verification stays cheap

The expensive proof is made once per template family. Each candidate is gated only against instance conditions, and unaffected proofs are reused.

Preliminary results

Shape	Timing	TileLang default	TileLang autotuned	cuBLASLt	MIP T1	MIP T1+T2	MIP T1+T2+T3	MIP T1–T4	cuBLASLt + graph	MIP T1–T3 vs best of TileLang autotuned and cuBLASLt	MIP T1–T4 vs cuBLASLt + graph
1024^3	warm	45.3	33.6	32.8	32.9	**29.0**	29.0	**27.6**	30.4	**1.13x**	**1.10x**
2048^3	warm	250.0	190.6	178.4	189.1	189.1	189.1	186.9	178.0	0.94x	0.95x
4096^3	warm	3,650.6	1,525.5	1,508.0	1,523.9	**1,469.0**	1,469.0	1,469.0	1,486.8	**1.03x**	**1.01x**
8192^3	warm	37,464.8	11,499.6	12,016.9	**11,448.9**	11,448.9	11,448.9	11,448.9	11,986.1	**1.00x**	**1.05x**
4096×4096×1024	warm	436.1	414.0	391.2	381.6	**375.5**	375.5	**373.2**	393.2	**1.04x**	**1.05x**
decode M=32	cold	217.8	164.8	140.8	**118.0**	118.0	118.0	118.0	140.6	**1.19x**	**1.19x**
decode M=128	cold	195.2	171.3	159.7	**124.6**	124.6	124.6	124.6	156.3	**1.28x**	**1.25x**

From searching harder to formulating and proving

Search

Templates	AutoTVM
Cost models	Ansor, Meta Schedule
Programmer DSLs	Triton, TileLang
RL on an LLM	CUDA-L2



Formulate and verify

LLM	writes the problem as mathematics
Solver	searches it with bounds
Verifier	proves every admitted design correct
Loop	measures, learns, refines the next layer

Three takeaways

1

The stack is six layers deep

From the serving system to the driver, every layer makes discrete decisions: a search problem at each level.

2

Compilers handed the search to machines

Templates, then learned cost models, then reinforcement learning on language models.

3

Next: formulate and prove

An LLM writes the problem, a solver optimizes it with bounds, a verifier proves the result correct.

Questions?