



CONFERENCE

—NORTH AMERICA 2026—

Helion: CuteDSL and TPU Backends
for Heterogeneous Hardware,
and Why It Suits Agents

Oguz Ulgen · Dunfan Lu · Jason Ansel

PYTORCH CONFERENCE 2026 • SAN JOSE

Helion: CuteDSL and TPU Backends for Heterogeneous Hardware, and Why It Suits Agents

Oguz Ulgen • Dunfan Lu • Jason Ansel

Meta Superintelligence Labs

October 20, 2026

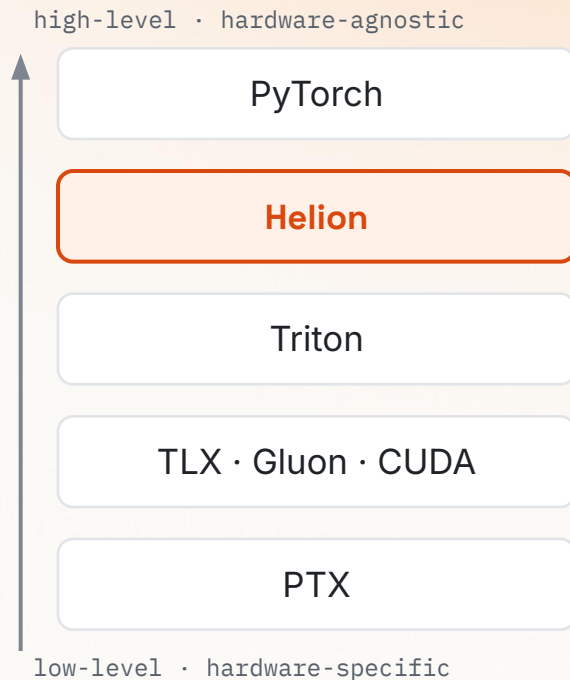


Helion: PyTorch with tiles

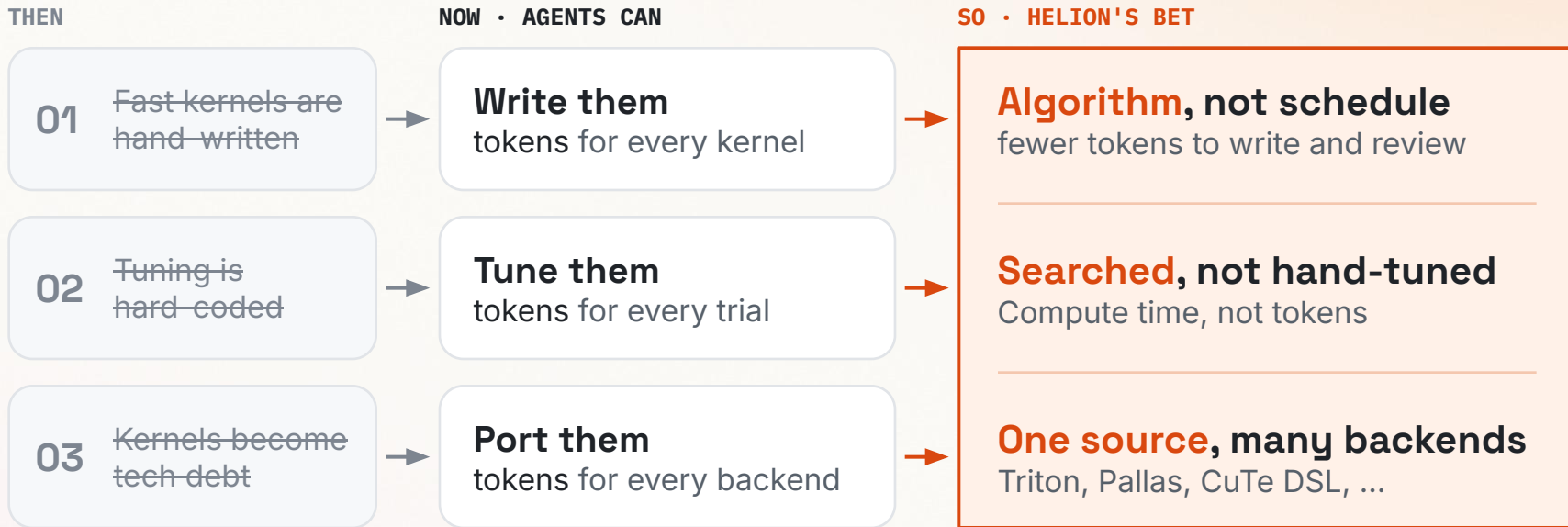
- A Python-embedded DSL for ML kernels
- Kernels are PyTorch ops on tiles
- A compiler and an autotuner generate the device code
- Open source · PyTorch Foundation project since April 2026

You write the algorithm.

Helion owns the schedule and the hardware.



Why a new DSL, **when agents write kernels?**

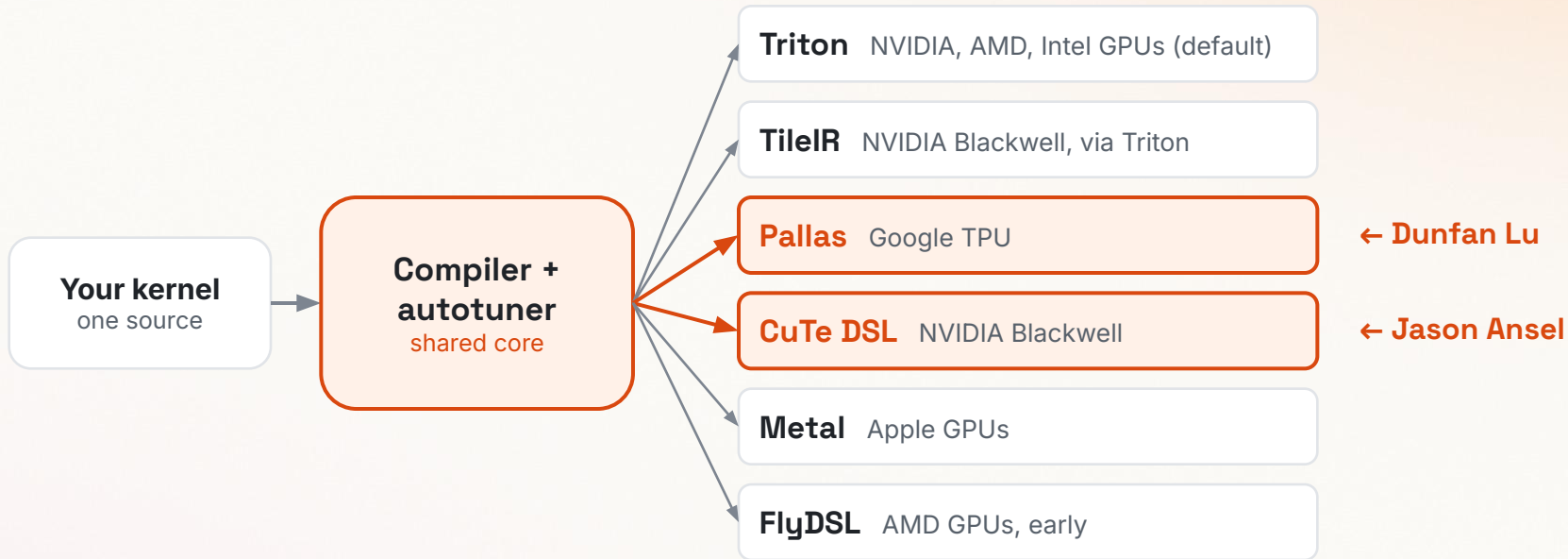


Tokens are the new engineer-hours. Helion kernels need fewer of them.



One source, many backends

Every accelerator has its own kernel language; Helion compiles one source to each



```
HELION_BACKEND=cute python my_kernel.py      # or per kernel: @helion.kernel(backend="pallas")
```



A complete kernel: softmax

```
softmax.py

import torch
import helion
import helion.language as hl

@helion.kernel()
def softmax(x: torch.Tensor) -> torch.Tensor:
    n, _m = x.size()
    out = torch.empty_like(x)
    for tile_n in hl.tile(n):
        row = x[tile_n, :]
        amax = torch.amax(row, dim=1, keepdim=True)
        exp = torch.exp(row - amax)
        out[tile_n, :] = exp / exp.sum(dim=1, keepdim=True)
    return out
```

YOU WRITE

The math, as PyTorch ops
Which loops run over tiles

HELION OWNS

Tile sizes and launch grid
Indices, strides, masks
Loop order, pipelining, warps
Code for your backend

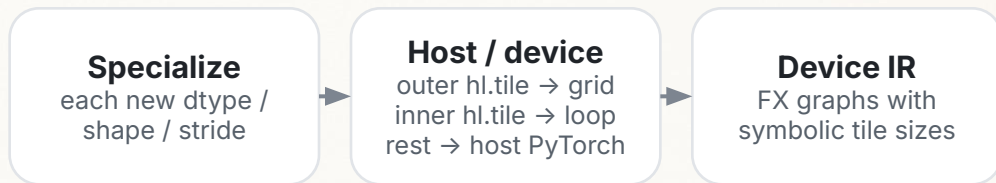
host: plain PyTorch

device: one kernel

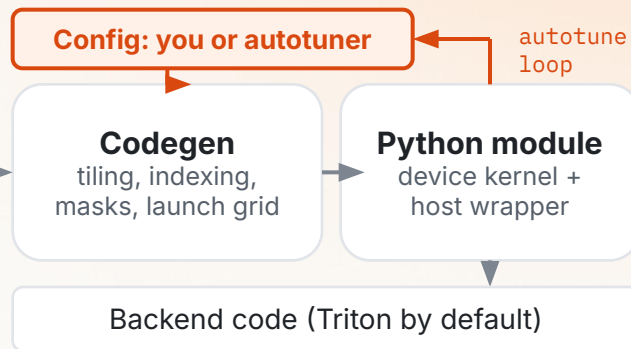


How it works

ONCE PER SHAPE



ONCE PER CONFIG



Device IR for a matmul's inner loop: FX graphs with symbolic tile sizes (u0, u1, u2), shared by every config

```

● ● ● device IR (FX)

def for_loop_0(arg0_1: "f32[u0, u1]"):
    x: "f16[1024, 1024]" = _host_tensor('x')
    ...
    block_size_2: "Sym(u2)" = _get_symnode('block_size_2')
    load: "f16[u0, u2]" = load(x, [sym_size_int, block_size_2])
    load_1: "f16[u2, u1]" = load(y, [block_size_2, sym_size_int_1])
    acc: "f32[u0, u1]" = torch.ops.aten.addmm.default(_new_var, load, load_1)
    return [acc]
  
```

LOWERING · FIRST RULE WINS

hl ops

→ backend hook

→ Helion aten rules (matmul, views)

→ TorchInductor (pointwise, reductions)



Matmul with a fused epilogue

```

matmul.py

@helion.kernel()
def matmul(x: Tensor, y: Tensor, epilogue=lambda acc, tile: acc) -> Tensor:
    m, k = x.size()
    k, n = y.size()
    out = torch.empty([m, n], dtype=x.dtype, device=x.device)
    for tile_m, tile_n in hl.tile([m, n]): # ①
        acc = hl.zeros([tile_m, tile_n], dtype=torch.float32)
        for tile_k in hl.tile(k): # ②
            acc = torch.addmm(acc, x[tile_m, tile_k], y[tile_k, tile_n])
        out[tile_m, tile_n] = epilogue(acc, (tile_m, tile_n)) # ③
    return out

# fused bias + relu:
out = matmul(x, y, lambda acc, tile: torch.relu(acc + bias[tile[1]])) # ③

```

① Outer tile → launch grid

② Inner tile → device loop; addmm → tl.dot

③ Lambda epilogue fused; captured bias becomes a kernel arg

No strides, masks or grid math in the source.

Flash attention in 18 lines

attention.py

```
@helion.kernel()
def attention(q: Tensor, k: Tensor, v: Tensor) -> Tensor:
    b, m, d = q.size() # [batch*heads, seq_len, head_dim]
    out = torch.empty_like(q)
    for tb, tm in hl.tile([b, m]):
        m_i = hl.full([tb, tm], float("-inf"), dtype=torch.float32)
        l_i = hl.zeros([tb, tm], dtype=torch.float32)
        acc = hl.zeros([tb, tm, d], dtype=torch.float32)
        for tn in hl.tile(k.size(1)):
            s = torch.bmm(q[tb, tm, :], k[tb, tn, :].mT, torch.float32) * d**-0.5
            m_new = torch.maximum(m_i, s.amax(-1))
            p = torch.exp(s - m_new[:, :, None])
            alpha = torch.exp(m_i - m_new)
            l_i = l_i * alpha + p.sum(-1)
            acc = torch.baddbmm(acc * alpha[:, :, None], p.to(v.dtype), v[tb, tn, :])
            m_i = m_new
        out[tb, tm, :] = (acc / l_i[:, :, None]).to(out.dtype)
    return out
```

- Online softmax
- Two batched matmuls
- fp32 running max, sum and accumulator
- No masks, strides or pointer math
- Ragged lengths like 777 just work
- Matches eager SDPA within bf16 tolerance



Readable output and automatic masking

```

generated Triton · softmax example · fp16 [4096, 2560], 2560 columns padded to 4096 lanes

mask_1 = indices_1 < 2560
# src[softmax.py:10]: row = x[tile_n, :]
row = tl.load(x + (indices_0[:, None] * 2560 + (0 + indices_1)[None, :] * 1),
              mask_1[None, :], other=0)
# src[softmax.py:11]: amax = torch.amax(row, dim=1, keepdim=True)
_mask_to = tl.where(tl.broadcast_to(mask_1[None, :], [_BLOCK_SIZE_0, _RDIM_SIZE_1]),
                    row, tl.full([], float('-inf'), tl.float16))
...
# src[softmax.py:13]: out[tile_n, :] = exp / exp.sum(dim=1, keepdim=True)
_mask_to_1 = tl.where(tl.broadcast_to(mask_1[None, :], [_BLOCK_SIZE_0, _RDIM_SIZE_1]),
                      v_3, tl.full([], 0, tl.float16))
sum_1 = tl.cast(tl.reshape(tl.sum(_mask_to_1.to(tl.float32), 1), [_BLOCK_SIZE_0, 1]), tl.float16)

```

- Generated code is annotated with the Helion source line it came from: `# src[file:line]`
- Padded lanes load 0; Helion re-masks with `-inf` before `amax` and 0 before `sum`. When tiles divide the shape, the masks disappear.
- `HELION_PRINT_OUTPUT_CODE=1` prints the generated code
- `HELION_INTERPRET=1` runs the kernel eagerly, so `print()` and `breakpoint()` work inside `hl.tile` loops



One Helion kernel → many device kernels

● ● ● config A: indexing=pointer · pid_type=flat

masks · pointer math

```
pid_0 = tl.program_id(0) % num_blocks_0
mask_0 = indices_0 < 2000
for offset_2 in tl.range(0, 1000, _BLOCK_SIZE_2):
    mask_2 = indices_2 < 1000
    load = tl.load(x + (indices_0[:, None] * 1000 + indices_2[None, :] * 1),
                  mask_0[:, None] & mask_2[None, :], other=0)
    acc = tl.dot(..., acc=acc_copy_0, ...)
# host: _launcher(_helion_matmul, (2016,), x, y, out, num_warps=4, num_stages=3)
```

● ● ● config B: indexing=tensor_descriptor · pid_type=persistent_interleaved · l2_groupings=[8]

no masks · descriptors

```
x_desc = tl.make_tensor_descriptor(x, [2000, 1000], [1000, 1], [_BLOCK_SIZE_0, _BLOCK_SIZE_2])
for virtual_pid in tl.range(tl.program_id(0), total_pids, _NUM_SM):
    first_pid_m = group_id * 8
    ...
    for offset_2 in tl.range(0, 1000, _BLOCK_SIZE_2):
        load = x_desc.load([offset_0, offset_2])
        acc = tl.dot(..., acc=acc_copy_0, ...)
# host: _launcher(_helion_matmul, (_NUM_SM,), x, y, out, _NUM_SM, num_warps=8, num_stages=4)
```

A config changes the **schedule**, not the algorithm.



Triton config knobs, and your own

block_sizes tile size per hl.tile dim

loop_orders tile iteration order

flatten_loops fold an N-D tile into 1-D

l2_groupings reorder program ids for L2 reuse

reduction_loops whole row vs. looped reduction

indexing per-access pointer / tensor_descriptor

pid_type flat · xyz · persistent (blocked / interleaved)

num_warps, num_stages warps and pipeline stages

range_* per-loop unroll, stages, warp specialization

split_k.py · a user-defined knob

```
from helion.autotuner import PowerOfTwoFragment

out = torch.zeros([m, n], dtype=x.dtype, device=x.device) # atomic_add accumulates
split_k = hl.register_tunable("split_k", PowerOfTwoFragment(1, 256))
k_block = helion.next_power_of_2(helion.cdiv(k, split_k))
for tile_m, tile_n, outer_k in hl.tile([m, n, k], block_size=[None, None, k_block]):
    acc = hl.zeros([tile_m, tile_n], dtype=torch.float32)
    for inner_k in hl.tile(outer_k.begin, outer_k.end):
        acc = torch.addmm(acc, x[tile_m, inner_k], y[inner_k, tile_n])
    hl.atomic_add(out, [tile_m, tile_n], acc)
```

Your own knobs: `hl.register_tunable(name, fragment)` with Enum (numeric), Boolean, Integer, PowerOfTwo fragments

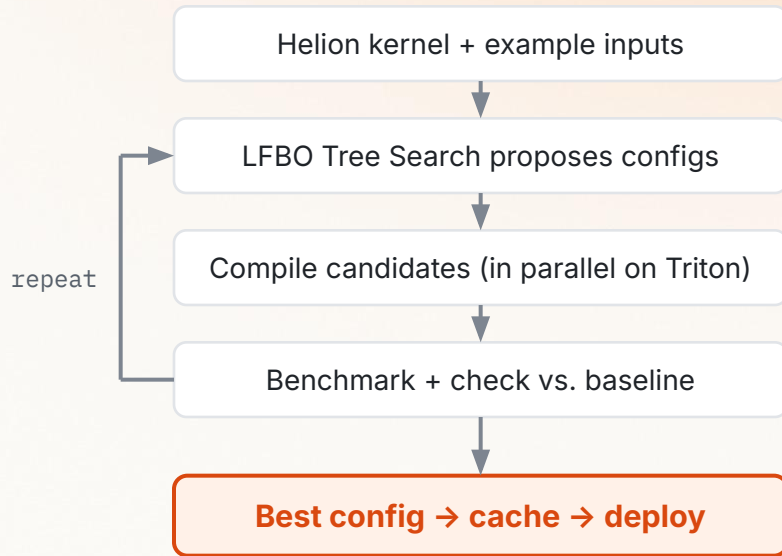


Your code defines the search space

- Your `hl.tile` loops imply the search space

$10^{15}+$ configs for a plain matmul on Triton
(shape-dependent)

- Default search: LFBO Tree Search, a Bayesian optimizer
- Every candidate is checked against a baseline



Deep dive today ***"Smarter Autotuning for Kernels"*** · Jongsok Choi & Ethan Che · 2:15 PM, Room 210B

Deploying tuned kernels

```
tune.py → kernels.py

# 1) tune.py: tune ahead of time, save
for tag, args in datasets.items():
    config = add.autotune(args)          # -> helion.Config
    config.save(f"configs/add_{tag}.json")

# 2) kernels.py: ship the tuned configs
@helion.kernel(configs=[
    helion.Config.load("configs/add_small.json"),
    helion.Config.load("configs/add_large.json"),
])
def add(x: torch.Tensor, y: torch.Tensor) -> torch.Tensor:
    ...
```

Scripts that autotune need an `if __name__ == "__main__":` guard

config=

one fixed config, no tuning at run time (partial configs are filled in)

configs=[...]

benchmarks only these, once per new shape

autotune_multi

one config across several shapes

@helion.aot_kernel

distills tuning into a decision-tree heuristic file you commit

allow_helion_deps=False

standalone export: imports only torch + the backend DSL; not every kernel supports it yet

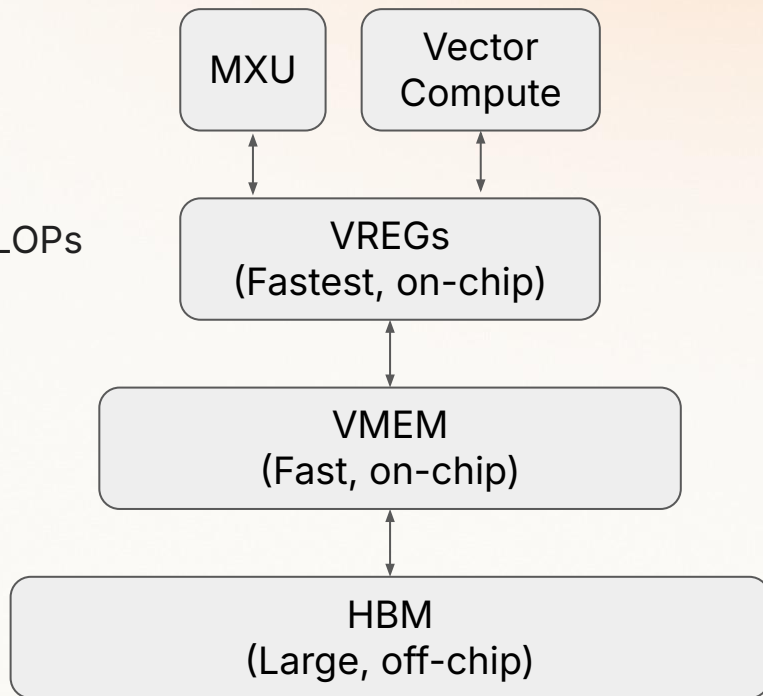
TPU Codegen Case Study

Dunfan Lu



TPU Hardware & Pallas Programming Model

- Explicit memory hierarchy with async copies
- Mostly sequential, few large workers
- MXU: matrix multiplication unit with significant TFLOPs



Helion TPU Codegen

Maximise software pipelining

- Outer loop: Pipelined tile fetches and compute (emit_pipeline)
- Inner loop: autotuned between
 - Pipelined tile fetches and compute (emit_pipeline)
 - Pre-fetching all tiles into VMEM, if possible (unroll)
- Auto-tuned pipeline buffer sizes

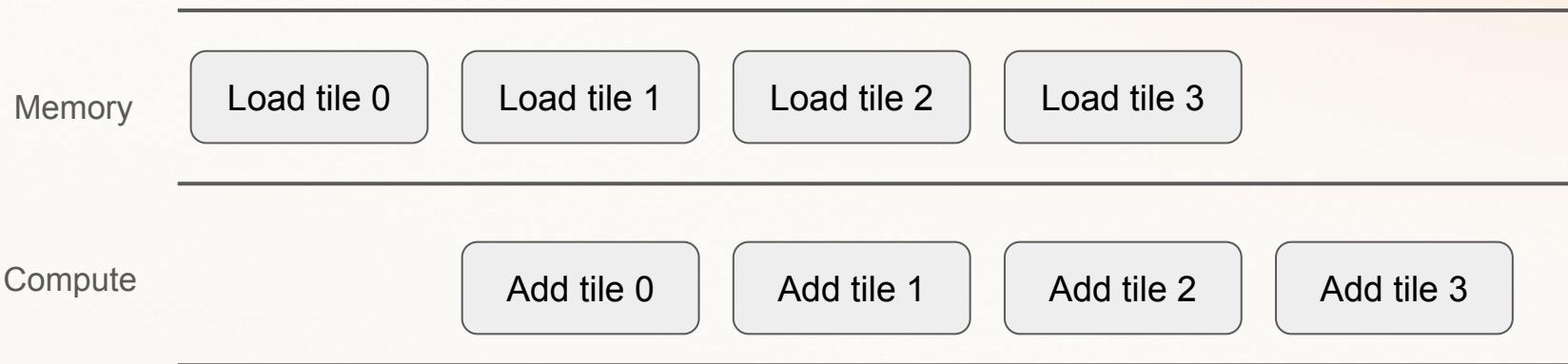


add: Helion source

```
@helion.kernel
def add(x: torch.Tensor, y: torch.Tensor) -> torch.Tensor:
    out = torch.empty_like(x)
    for tile in hl.tile(out.size()):
        out[tile] = x[tile] + y[tile]
    return out
```



add: Pipelining



add: Generated Pallas Code Overview

pseudocode

```
def _loop_body(x_tile, y_tile, out_tile):  
    out_tile[:] = x_tile[:] + y_tile[:]
```

← Device code
operates on VMEM ref

```
def _kernel_body(x, y, out):  
    # [x tiling, y tiling, out tiling]  
    block_specs = [VMEM(BLOCK_SIZE), VMEM(BLOCK_SIZE), VMEM(BLOCK_SIZE)]  
    pallas.tpu.emit_pipeline(_loop_body, block_specs, ...)(x, y, out)
```

← Device code
operates on HBM refs
Tiled invocation of loop body

```
def add(x, y):  
    out = pallas.kernel(_kernel_body, ...)(x, y)  
    return out
```

← Host-side code



attention: Helion source

```
@helion.kernel
def attention(q: torch.Tensor, k: torch.Tensor, v: torch.Tensor) -> torch.Tensor:
    out = torch.empty(...)
    for tile_b, tile_q in hl.tile(B * H, S):
        this_q = q[tile_b, tile_q, :]
        acc = ...
        for tile_kv in hl.tile(S):
            this_k = k[tile_b, tile_kv, :]
            this_v = v[tile_b, tile_kv, :]
            <matmul, online softmax, matmul, update acc>
        out[tile_b, tile_q] = acc
    return out
```



attention: Generated Pallas Code Structure

```
def _outer_loop_body(q_VMEM, k_HBM, v_HBM, out_VMEM):
    acc = ...
    this_q = q_VMEM[:, :, :]
    def _inner_loop_body(k_VMEM, v_VMEM):
        this_k = k_VMEM[:, :, :]
        this_v = v_VMEM[:, :, :]
        <matmul, online softmax, matmul, update acc>
    # [K tile size, V tile size]
    block_specs = [VMEM(BLOCK_SIZE_1), VMEM(BLOCK_SIZE_1)]
    pallas.tpu.emit_pipeline(_inner_loop_body, block_specs)(k_HBM, v_HBM)
    out_VMEM[...] = acc
```

Inner loop tiles K and V



```
def _kernel_body(q_HBM, k_HBM, v_HBM, out_HBM):
    # [Q tile size, K tile size, V tile size, out tile size]
    block_specs = [VMEM(BLOCK_SIZE_0), HBM, HBM, VMEM(BLOCK_SIZE_0)]
    pallas.tpu.emit_pipeline(_outer_loop_body, ...)(q_HBM, k_HBM, v_HBM, out_HBM)
```

Outer loop tiles Q and Out

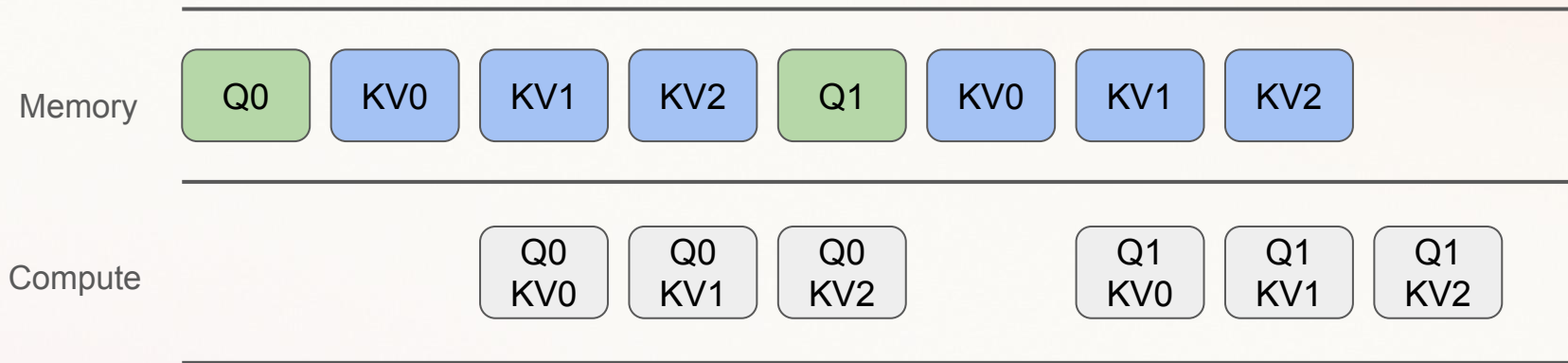


```
def attention(q, k, v):
    out = pallas.kernel(_kernel_body, ...)(q, k, v)
    return out
```



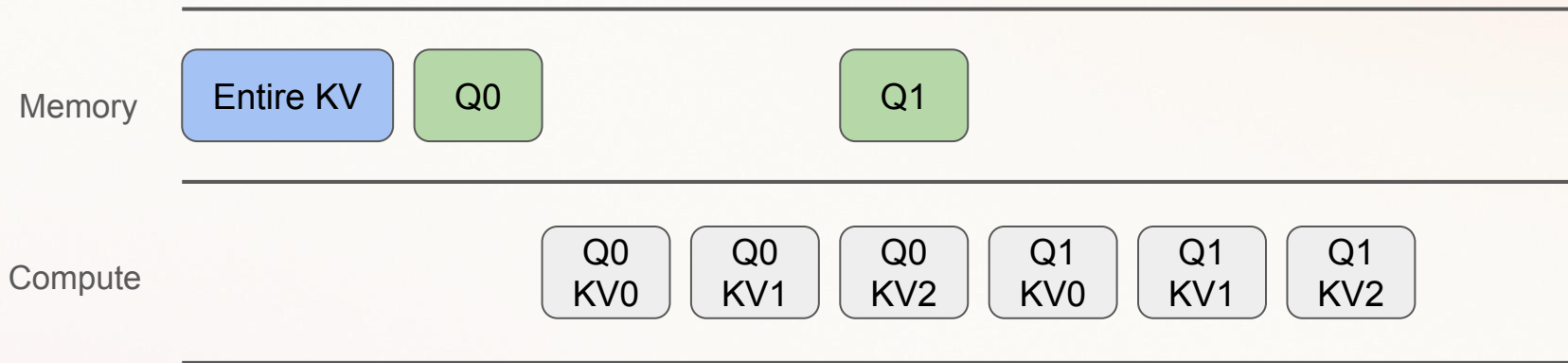
attention: Pipelined KV

```
pallas_loop_type="emit_pipeline"
```



attention: Prefetched KV

```
pallas_loop_type="unroll"
```



attention: Generated Pallas Code Structure

```
def _outer_loop_body(q_VMEM, k_VMEM_FULL, v_VMEM_FULL, out_VMEM):
```

```
    acc = ...
```

```
    this_q = q_VMEM[:, :, :]
```

```
    for offset in range(0, k_VMEM_FULL.size(1), BLOCK_SIZE_1):
```

```
        this_k = k_VMEM_FULL[:, pallas.ds.slice(offset, BLOCK_SIZE_1), :]
```

```
        this_v = v_VMEM_FULL[:, pallas.ds.slice(offset, BLOCK_SIZE_1), :]
```

```
        <matmul, online softmax, matmul, update acc>
```

```
    out_VMEM[...] = acc
```

← Manually slicing K and V

```
def _kernel_body(q_HBM, k_HBM, v_HBM, out_HBM):
```

```
    # [Q tile size, K tile size, V tile size, out tile size]
```

```
    block_specs = [VMEM(BLOCK_SIZE_0), VMEM(len(k_HBM)), VMEM(len(v_HBM)), VMEM(BLOCK_SIZE_0)]
```

```
    pallas.tpu.emit_pipeline(_outer_loop_body, ...)(q_HBM, k_HBM, v_HBM, out_HBM)
```

↗ Load K and V fully in VMEM

```
def attention(q, k, v):
```

```
    out = pallas.kernel(_kernel_body, ...)(q, k, v)
```

```
    return out
```



Perf implications of inner loop type

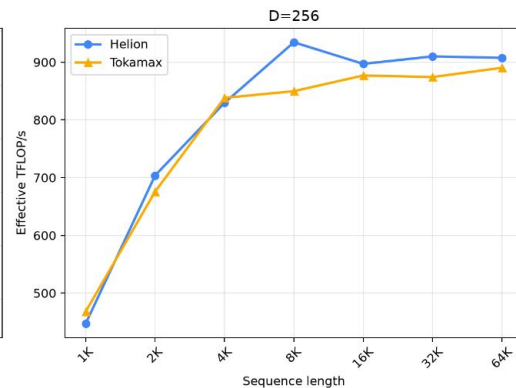
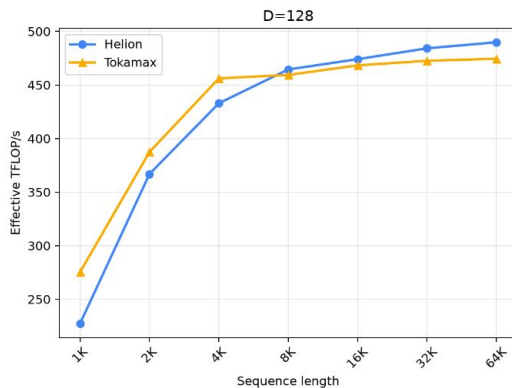
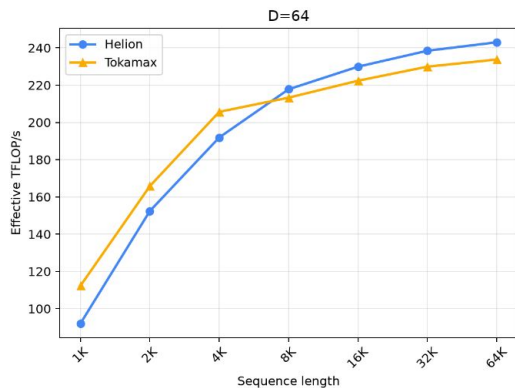
- **emit_pipeline:**
 - Has bubbles in compute
 - VMEM usage is $O(\text{tile size})$
- **unroll:**
 - Less bubbles in compute
 - VMEM usage is $O(\text{input size})$
 - Entire KV needs to fit within VMEM, not always possible
- **Helion's Autotuner automatically finds the preferred loop type**

	S = 16k	S = 32k
emit_pipeline TFLOPs	811	910
unroll TFLOPs	897	OOM

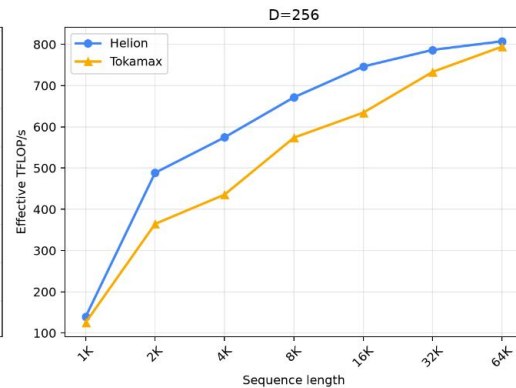
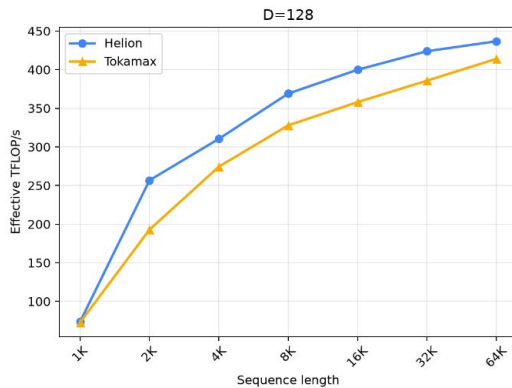
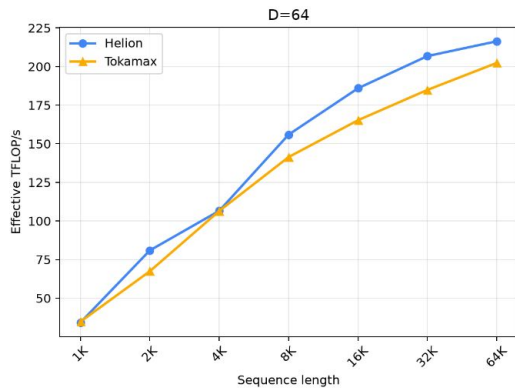


attention benchmarks vs Tokamax

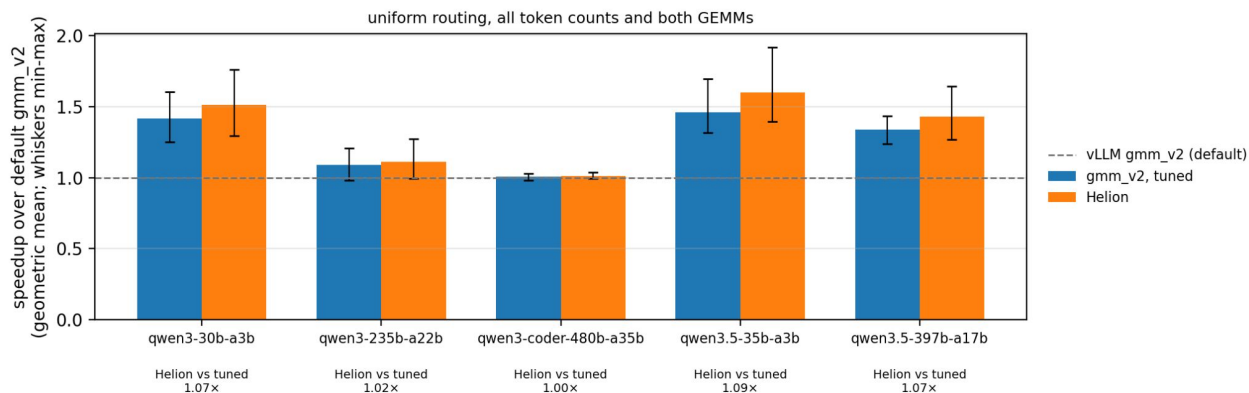
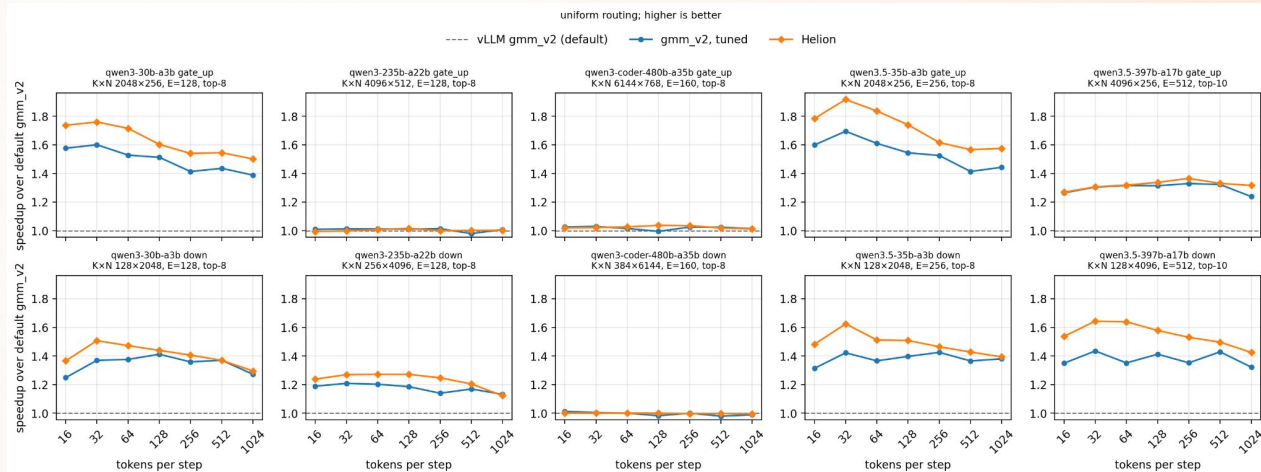
TPU v7 BF16 Dense Attention: Helion vs Tokamax



TPU v7 BF16 Causal Attention: Helion vs Tokamax



group_gemm benchmarks vs Tokamax



Credit to Emilio Cota

Helion's new CuteDSL backend

Jason Ansel



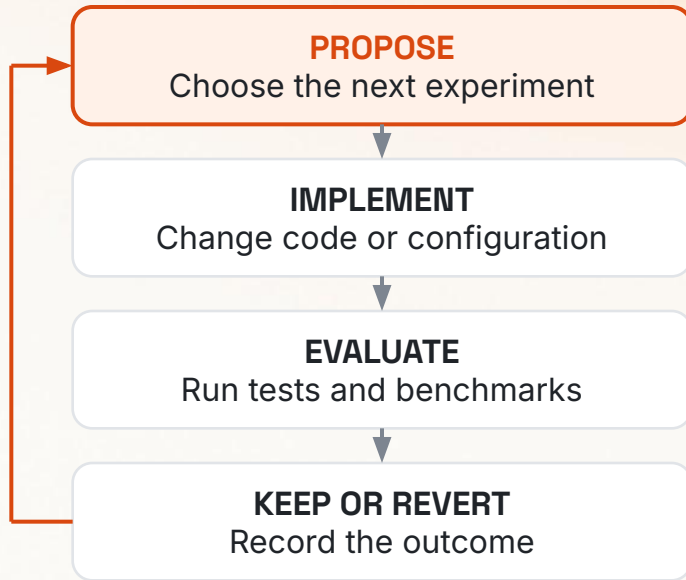
Agentic optimization loops

Give the agent

1. An objective and constraints
2. Tests and benchmarks
3. A budget and experiment history

Measured results guide
the next experiment

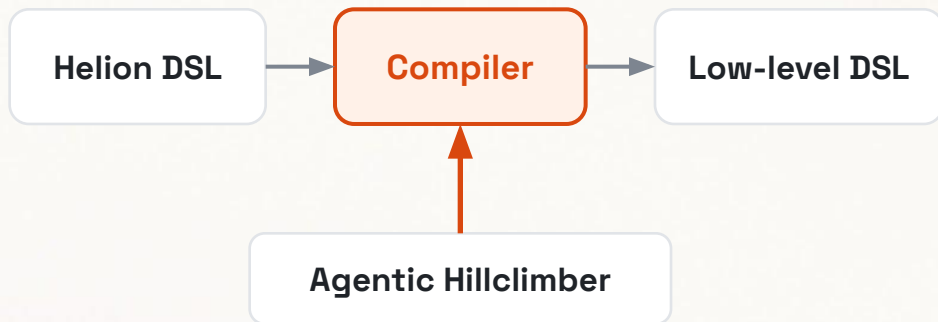
Next
experiment



Repeat until the goal is met or the budget runs out

Hillclimb the compiler vs the kernel

Hillclimbing the compiler



Hillclimbing the kernel



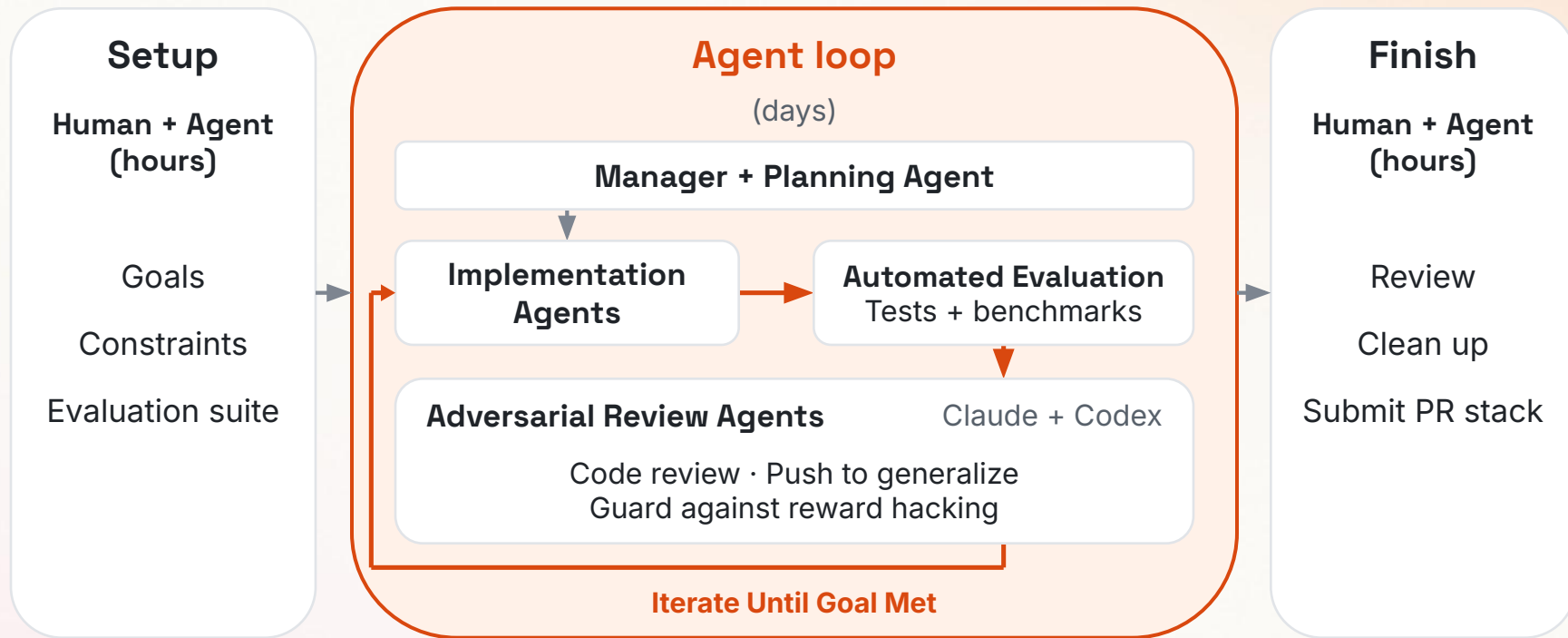
Advantages of hillclimbing the compiler

Amortize tokens, time, and checks across thousands of kernels

Higher-level code is easier for agents to edit reliably

Reproduce results and change models more easily

Agentic performance hillclimbing



Why Helion suits agentic hill climbing

Trade-offs become explicit choices

Expose discrete choices for tile sizes, scheduling, staging, and parallelism. The autotuner searches them instead of relying on a single heuristic for every kernel.

LLM + autotuner search more efficiently

LLMs seed promising starting points and propose source-level algorithmic changes. The autotuner systematically searches local configurations, saving time and tokens.

A structured search space provides guardrails

Helion's structured DSL and compiler constraints narrow the search space, help preserve correctness, and reduce invalid edits, making optimization more reliable.



Attention Variants (NVIDIA B200; power-capped)

Achieved TFLOPS per implementation, and Helion (CuTe) speedup vs the best non-Helion implementation

bf16; B=4, H=16, D=64, 2K / 4K / 8K tokens unless noted:

gqa-causal: H=32, H_kv=8, D=128 · mla-causal: D=192, D_v=128 · document: B=1, 4K / 8K / 16K tokens · gemma2: H_kv=8, D=128 · svg-spatial / svg-temporal: 128 prompt + 4 / 8 / 16 frames of 512 tokens · shared-prefix: B=1, 4K / 8K / 16K tokens · flamingo: 2K / 4K / 8K text queries over half as many image keys · jetspec-tree: 64 tree queries over a 2K / 4K / 8K prefix ·

jetspec-training: 128 queries over a 2K / 4K / 8K prefix · graphormer-spatial / graphormer-edge: 512 / 1K / 2K nodes

TFLOPS cells = arithmetic mean over the 3 shapes; speedup column = geomean over the 3 shapes of the per-shape ratios

		SDPA (cuDNN)	Flash- Attention 4	FlexAttention (Triton)	FlexAttention (Triton, max-autotune)	FlexAttention (CuTe)	Helion (CuTe)	Helion (CuTe) speedup vs best other	
dense	noop	696	583	451	451	684	688	0.99x	TFLOPS (useful work / time)
	causal	578	545	334	318	445	565	0.98x	
	gqa-causal	873	879	468	468	850	882	1.00x	
	mla-causal	830	835	328	329	769	856	1.03x	
static mask	sliding-window	45 dense mask	408	256	261	342	429	1.04x	
	prefix-lm	104 dense mask	no kernel	352	337	479	585	1.25x	
	dilated-sw	25 dense mask	no kernel	97	107	120	193	1.59x	
	block-diffusion	51 dense mask	no kernel	294	248	157	453	1.55x	
	natten2d	9 dense mask	no kernel	21	30	19	83	2.74x	
	sta2d	99 dense mask	no kernel	373	343	506	597	1.20x	
	sta3d	38 dense mask	no kernel	283	279	34	447	1.56x	
	batchify	14 dense mask	no kernel	106	136	77	236	1.73x	
	svg-spatial	91 dense mask	no kernel	342	326	346	458	1.32x	
	svg-temporal	91 dense mask	no kernel	160	195	89	299	1.56x	
gather mask	document	17 dense mask	323	38	110	92	338	1.05x	
	shared-prefix	14 dense mask	no kernel	196	200	154	324	1.55x	
	global-sw	53 dense mask	no kernel	147	147	46	200	1.36x	
	flamingo	44 dense mask	no kernel	279	279	289	468	1.43x	
	vsa	8 dense mask, 2K/4K	no kernel	348	348	no kernel	387	1.12x	
speculative	jetspec-tree	18 dense mask	no kernel	30	44	28	68	1.54x	
	jetspec-training	31 dense mask	no kernel	109	187	41	229	1.21x	
score mod	alibi	81 dense mask	no kernel	242	286	336	494	1.47x	
	rel-bias	wrong output	no kernel	294	367	493	549	1.11x	
	softcap	no kernel	107	136	179	68	281	1.58x	
	sandwich	wrong output	no kernel	35	150	62	359	2.41x	
	sigmoid-act	no kernel	no kernel	137	152	46	217	1.43x	
	graphormer-spatial	60 dense mask	no kernel	56	98	76	198	2.03x	
	graphormer-edge	38 dense mask	no kernel	37	51	12	72	1.40x	
combined	gemma2	no kernel	183	173	179	125	368	2.01x	
geomean over all 87 shapes								1.41x	

TFLOPS = useful FLOPS / median time: 2·(d_qk + d_v) per unmasked (q, kv) pair, the same count for every implementation (no credit for computing masked blocks); on masked / biased rows SDPA runs cuDNN over a materialised dense mask or bias, built inside its timed region on the rows marked † (the block-sparse variants) and precomputed on the others. Black outline / bold = within 2% of the row's best. Speedup column = geomean over the shapes of best non-Helion time / Helion (CuTe) time; darker orange = larger, pink = below parity; 'no kernel' = not expressible; 'wrong output' = outside tolerance; '2K/4K only' = shapes with a result.

Attention Variants, BACKWARD pass (NVIDIA B200; power-capped)

Backward (dQ, dK, dV) achieved TFLOPS per implementation, and Helion (CuTe) speedup vs the best non-Helion implementation

bf16; B=4, H=16, D=64, 2K / 4K / 8K tokens unless noted:

gqa-causal: H=32, H_kv=8, D=128 · mla-causal: D=192, D_v=128 · document: B=1, 4K / 8K / 16K tokens · gemma2: H_kv=8, D=128 · svg-spatial / svg-temporal: 128 prompt + 4 / 8 / 16 frames of 512 tokens · shared-prefix: B=1, 4K / 8K / 16K tokens · flamingo: 2K / 4K / 8K text queries over half as many image keys · jetspec-tree: 64 tree queries over a 2K / 4K / 8K prefix ·

jetspec-training: 128 queries over a 2K / 4K / 8K prefix · graphormer-spatial / graphormer-edge: 512 / 1K / 2K nodes

TFLOPS cells = arithmetic mean over the 3 shapes; speedup column = geomean over the 3 shapes of the per-shape ratios

		SDPA (cuDNN)	Flash- Attention 4	FlexAttention (Triton)	FlexAttention (Triton, max-autotune)	FlexAttention (CuTe)	Helion (CuTe)	Helion (CuTe) speedup vs best other
dense	noop	626	556	345	344	563	706	1.13x
	causal	570	473	297	297	398	615	1.08x
	gqa-causal	696	663	365	344	552	709	1.02x
	mla-causal	653	611	188	248	no kernel	658	1.01x
static mask	sliding-window	80	380	248	248	294	484	1.27x
	prefix-lm	131	no kernel	305	305	421	634	1.51x
	dilated-sw	44	no kernel	104	105	119	243	2.04x
	block-diffusion	89	no kernel	273	274	304	550	1.79x
	natten2d	16	no kernel	28	28	28	95	3.37x
	sta2d	172	no kernel	312	313	462	635	1.37x
	sta3d	69	no kernel	265	265	199	498	1.87x
	batchify	25	no kernel	129	130	140	239	1.70x
	svg-spatial	160	no kernel	295	295	387	615	1.59x
	svg-temporal	159	no kernel	180	180	166	447	2.56x
	document	29	298	142	144	118	354	1.17x
	shared-prefix	26	no kernel	154	166	181	341	1.82x
gather mask	global-sw	97	no kernel	154	155	115	314	2.04x
	flamingo	72	no kernel	251	243	290	403	1.21x
	vsa	25	no kernel	284	284	no kernel	523	1.82x
	jetspec-tree	52	no kernel	46	49	46	99	1.73x
speculative	jetspec-training	72	no kernel	122	111	101	203	1.51x
	alibi	121	no kernel	275	276	426	601	1.41x
score mod	rel-bias	wrong output	no kernel	323	325	572	749	1.31x
	softcap	no kernel	259	194	188	172	427	1.65x
	sandwich	wrong output	no kernel	76	184	230	503	2.19x
	sigmoid-act	no kernel	no kernel	139	143	114	285	2.00x
	graphormer-spatial	117	no kernel	61	125	106	280	1.90x
	graphormer-edge	77	no kernel	21	47	28	139	1.73x
combined	gemma2	no kernel	232	99	204	189	424	1.83x
geomean over all 87 shapes								1.61x

TFLOPS (useful work / time)

TFLOPS = useful backward FLOPS / median time: 2·(3·d_qk + 2·d_v) per unmasked (q, kv) pair (five GEMMs; the same count for every implementation, no credit for computing masked blocks); every implementation is timed on the backward only (forward run once outside the timed region); on masked / biased rows SDPA's timed region also materialises the dense mask or bias from the mods (torch.compile), the forward figure's protocol. mla-causal (D=192, D_v=128): no Helion CuTe backward kernel (TMEM budget of the 1-CTA family); baselines only, not in the geomean. Black outline / bold = within 2% of the row's best. Speedup column = geomean over the shapes of best non-Helion time / Helion (CuTe) time; darker orange = larger, pink = below parity; 'no kernel' = not expressible; 'wrong output' = outside tolerance; '2K/4K only' = shapes with a result.

NVIDIA B200 (power-capped, ~740 W), bf16, CUDA-event do_bench medians with the same timer for every implementation; Helion numbers are cold full autotunes. Oct 2026.

What matters as performance converges

Long-term bet: stronger models and enough search
bring more systems close to hardware limits.

Cost of reaching performance

Token efficiency, accelerator time, and human effort determine how quickly and cheaply we reach the target.

Quality of the resulting software

Maintainability, development velocity, stability, and reproducibility matter when performance is comparable.

Reuse across kernels and hardware

Compiler improvements amortize optimization effort. High-level source carries algorithms forward as backends evolve.