Helion: A High-level DSL for Kernel Authoring

Oguz Ulgen
Meta Superintelligence Labs

From Meta to the PyTorch Foundation

Oct 2025 - Beta
PyTorch Conference SF

First interactions
with the ecosystem
> Triton Dev Conference
> PyTorch Conference 2025

Triton backend

Q1 2026

Close collaboration
with PyTorch partners

1st Helion Hackathon
San Francisco - March 14th
350+ applicants, 50 teams

Demo at **NVIDIA GTC** 2026

April 2026
PyTorch Conference Europe

Joining
the PyTorch Foundation

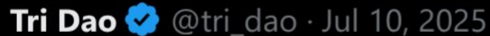
Helion 1.0

More backends to come

What’s Helion?

- Helion lets you author tile-based kernels using PyTorch
    - No more passing tensor sizes & strides
- It lowers to Triton (soon: Pallas, later: CuTe DSL and Metal)
    - One helion kernel corresponds to many triton kernels
- It comes with a powerful autotuner
    - no more sprinkling eviction hints or stage counts through your code

- In effect, this makes it easier to write kernels than Triton
- With potentially better performance due to better tuning
- Removes tuning knobs as a “free variable,” and systematically tests instead



Tri Dao @tri_dao · Jul 10, 2025

I really like Phil Tillet's framing of different tools having different tradeoffs in productivity and performance: torch compile, triton, CUDA, PTX. It's still early but CuTe-DSL and similar Python-based DSL might bend this curve. And soon we can probably get LLMs to generate these kernels!

Phil's talk here: semianalysis.com/wp-content/upl...

productivity

Torch

Triton

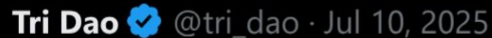
CuTe

DSL

CUDA

PTX

performance



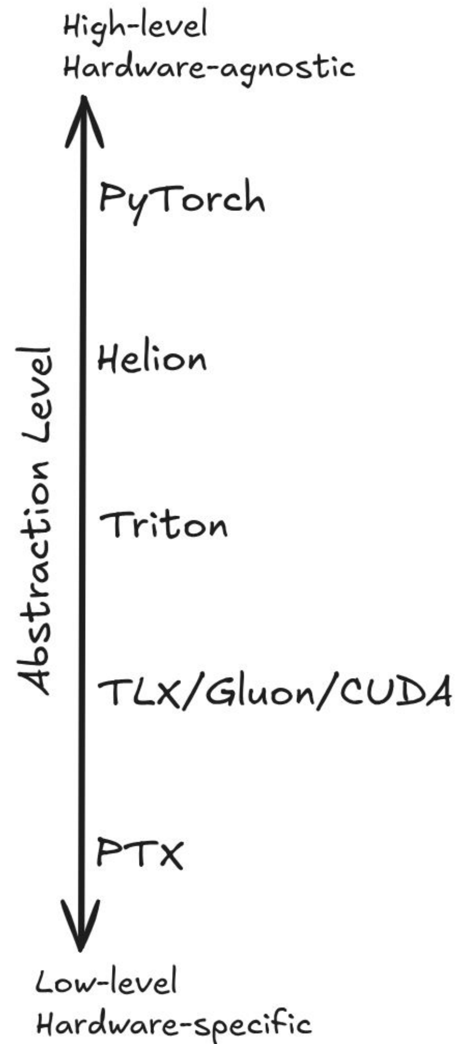
Tri Dao @tri_dao · Jul 10, 2025

I really like Phil Tillet's framing of different tools having different tradeoffs in productivity and performance: torch compile, triton, CUDA, PTX. It's still early but CuTe-DSL and similar Python-based DSL might bend this curve. And soon we can probably get LLMs to generate these kernels!

Phil's talk here: semianalysis.com/wp-content/upl...

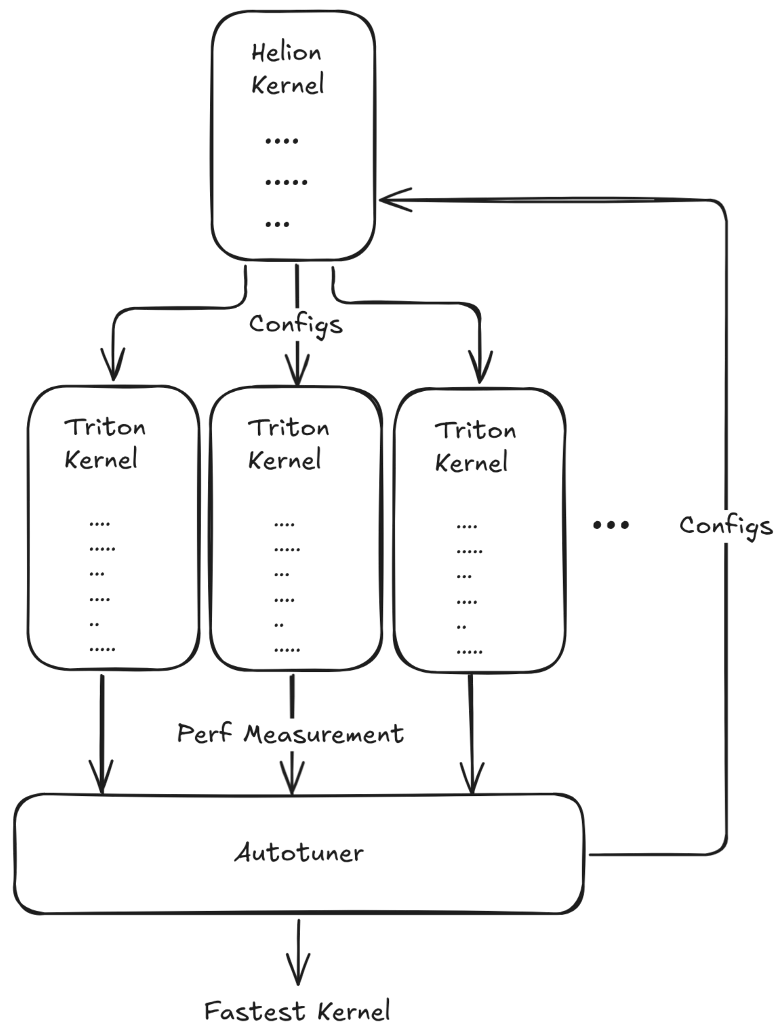
Motivation: Why a new DSL?

- Thousands of custom kernels used at large companies
  - Custom kernels turn into tech debt quickly
  - Hard to port to new hardware
  - Heterogeneous hardware is a new trend
- Higher level: automate things that are manual in Triton:
  - Tensor indexing (strides, block_ptr/pointer/TMA)
  - Wrapper/grid/block sizes definitions
  - Defining search spaces
  - Arg management
  - Templating
- Provides more control than *torch.compile*
- Better performance portability
- Autotuning (see next slide)



Autotuning

- One Helion kernels maps to many Triton kernels, defined by a config
- Search spaces, are created implicitly
  - E.g. hl.tile([m, n]) implies block sizes, iteration order, flattening, swizzling, etc choices
- The autotuner can search thousands of configurations (each corresponding to a Triton kernel)
- Autotuning is done once ahead-of-time, and resulting tuned configuration are deployed



Hardware Heterogeneity

Helion currently supports various hardwares via Triton backend:

- Nvidia A100, H100, B200, GB200+
- AMD MI300X+
- CPU (via Triton CPU)
- Accelerators that can consume Triton

We are currently working on:

- Cute DSL for Nvidia GPUs
- Pallas backend for Google TPUs
- Metal backend for Apple devices

Matrix Multiply Example

```python
import torch, helion, helion.language as hl

@helion.kernel()
def matmul(x: torch.Tensor, y: torch.Tensor) -> torch.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]):
        acc = hl.zeros([tile_m, tile_n], dtype=torch.float32)
        for tile_k in hl.tile(k):
            acc = torch.addmm(acc, x[tile_m, tile_k], y[tile_k, tile_n])
        out[tile_m, tile_n] = acc

    return out
```

Language Constructs

- hl.tile(sizes) subdivides iteration space into tiles
  - Tile sizes, iteration order, flattening, swizzling is autotuned
  - Outermost hl.tile becomes GPU launch grid
  - Inner hl.tile become loops on device
- Standard PyTorch ops can be used
  - Familiarity with PyTorch means you already know most of Helion
  - Pointwise ops (add, sigmoid, etc): lowered with TorchInductor
  - Reduction ops (sum, softmax, etc): lowered with TorchInductor
  - Matmul ops: become tl.dot
  - View ops: become tl.* ops
- Control flow
  - Supported in the top level kernel (becomes multiple FX graphs in Helion IR)
  - Function calls are traced with make_fx

Configuration Space

Indexing Choices

Helion

```
v = x[tile0, tile1]
```

Output Triton (Helion config.indexing = “pointer”)

```
indices_0 = pid_0 * _BLOCK_SIZE_0 + tl.arange(0, _BLOCK_SIZE_0)
mask_0 = indices_0 < x_size_0
…
v = tl.load(x + indices_0[:, None] * x_stride_0
              + indices_1[None, :] * x_stride_1,
            mask_0[:, None] & mask_1[None, :], other=0)
```

Output Triton (Helion config.indexing = “block_ptr”)

```
v = tl.load(tl.make_block_ptr(x,
   [x_size_0, x_size_1], [x_stride_0, x_stride_1], [offset_0, offset_1],
   [_BLOCK_SIZE_0, _BLOCK_SIZE_1], [1, 0]),
   boundary_check=[0, 1], padding_option='zero'))
```

Output Triton (Helion config.indexing = “tensor_descriptor”)

```
x_desc = tl.make_tensor_descriptor(x, [x_size_0, x_size_1],
[x_stride_0, x_stride_1], [_BLOCK_SIZE_0, _BLOCK_SIZE_1])
…
v = x_desc.load([offset_0, offset_1])
```

Implicit Block Sizes

Helion

Output Triton (Helion config.block_sizes = [32, 32])

```
@triton.jit
def _add_kernel(...,
                _BLOCK_SIZE_0: tl.constexpr,
                _BLOCK_SIZE_1: tl.constexpr):
    num_blocks_0 = tl.cdiv(x_size_0, _BLOCK_SIZE_0)
    pid_0 = tl.program_id(0) % num_blocks_0
    pid_1 = tl.program_id(0) // num_blocks_0
    offset_0 = pid_0 * _BLOCK_SIZE_0
    indices_0 = (offset_0 + tl.arange(0, _BLOCK_SIZE_0)).to(tl.int32)
    mask_0 = indices_0 < x_size_0
    offset_1 = pid_1 * _BLOCK_SIZE_1
    indices_1 = (offset_1 + tl.arange(0, _BLOCK_SIZE_1)).to(tl.int32)
    …

# Call the kernel:
_BLOCK_SIZE_0 = 32
_BLOCK_SIZE_1 = 32
_add_kernel[triton.cdiv(x.size(0), _BLOCK_SIZE_0) *
            triton.cdiv(x.size(1), _BLOCK_SIZE_1)](
…_BLOCK_SIZE_0, _BLOCK_SIZE_1, …)
```

Loop Flattening

Helion

for tile0, tile1 in hl.tile([x, y]):

Output Triton (Helion config.flatten_loops = [True])

```
@triton.jit
def _add_kernel(..., _BLOCK_SIZE_0_1: tl.constexpr):
    offsets_0_1 = tl.program_id(0) * _BLOCK_SIZE_0_1 + tl.arange(
            0, _BLOCK_SIZE_0_1)
    indices_1 = offsets_0_1 % x_size_1
    indices_0 = offsets_0_1 // x_size_1
    mask_0_1 = offsets_0_1 < x_size_0 * x_size_1
    …
```

```
# Call the kernel
_BLOCK_SIZE_0_1 = 1024
_add_kernel[triton.cdiv(x.size(0) * x.size(1), _BLOCK_SIZE_0_1)](
    …, _BLOCK_SIZE_0_1, …)
```

Reduction Rolling

Helion

Output Triton (Helion config.reduction_loops = None)

```
load = tl.load(x, …)  # loads entire row
v = tl.sum(load, 1)
```

Output Triton (Helion config.reduction_loops = [32])

```
acc = tl.full([_BLOCK_SIZE_0, _REDUCTION_BLOCK_1], 0)
for roffset_1 in tl.range(0, x_size_1, _REDUCTION_BLOCK_1):
    rindex_1 = roffset_1 + tl.arange(0, _REDUCTION_BLOCK_1)
    mask_1 = rindex_1 < x_size_1
    load = tl.load(x + …)
    acc += load
v = tl.sum(sum_1_acc, 1)
```

PID Type (Persistent Kernels)

Output Triton (Helion config.pid_type = "flat")

```
@triton.jit
def kernel(...):
    …

kernel[triton.cdiv(x.size(0), _BLOCK_SIZE_0) *
       triton.cdiv(x.size(1), _BLOCK_SIZE_1)](...)
```

Output Triton (Helion config.pid_type = "xyz")

```
@triton.jit
def kernel(...):
    …

kernel[triton.cdiv(x.size(0), _BLOCK_SIZE_0),
       triton.cdiv(x.size(1), _BLOCK_SIZE_1)](...)
```

Output Triton (Helion config.pid_type = "persistent_interleaved")

```
@triton.jit
def kernel(...):
    total_pids = (tl.cdiv(x_size_0, _BLOCK_SIZE_0) *
                  tl.cdiv(x_size_1, _BLOCK_SIZE_1))
    for virtual_pid in tl.range(tl.program_id(0), total_pids, _NUM_SM):
        …

kernel[_NUM_SM](...)
```

Output Triton (Helion config.pid_type = "persistent_blocked")

```
@triton.jit
def kernel(...):
    total_pids = (tl.cdiv(x_size_0, _BLOCK_SIZE_0) *
                  tl.cdiv(x_size_1, _BLOCK_SIZE_1))
    block_size = tl.cdiv(total_pids, _NUM_SM)
    start_pid = tl.program_id(0) * block_size
    end_pid = tl.minimum(start_pid + block_size, total_pids)
    for virtual_pid in tl.range(start_pid, end_pid):
        …

kernel[_NUM_SM](...)
```

L2 Grouping / Loop Reordering

Output Triton (Helion config.l2_grouping = None)

```
num_blocks_0 = tl.cdiv(x_size_0, _BLOCK_SIZE_0)
pid_0 = tl.program_id(0) % num_blocks_0
pid_1 = tl.program_id(0) // num_blocks_0
```

Output Triton (Helion config.loop_orders = [[1, 0]])

```
num_blocks_0 = tl.cdiv(x_size_1, _BLOCK_SIZE_1)
pid_0 = tl.program_id(0) % num_blocks_1
pid_1 = tl.program_id(0) // num_blocks_1

… (usage of pid_0/pid_1 swapped throughout code)
```

Output Triton (Helion config.l2_grouping = 32)

```
num_pid_m = tl.cdiv(x_size_0, _BLOCK_SIZE_0)
num_pid_n = tl.cdiv(x_size_1, _BLOCK_SIZE_1)
inner_2d_pid = tl.program_id(0)
num_pid_in_group = 32 * num_pid_n
group_id = inner_2d_pid // num_pid_in_group
first_pid_m = group_id * 32
group_size_m = min(num_pid_m - first_pid_m, 32)
pid_0 = (first_pid_m + inner_2d_pid %
            num_pid_in_group % group_size_m)
pid_1 = inner_2d_pid % num_pid_in_group // group_size_m
```

Triton Tunables

```
config.range_unroll_factors = [2]
config.range_warp_specializes = [True]
config.range_num_stages = [3]
config.range_multi_buffers = [False]
config.range_flattens = [True]
```

Output Triton

```
for offset_1 in tl.range(
    0, x_size_1.to(tl.int32), _BLOCK_SIZE_1,
    loop_unroll_factor=2,
    warp_specialize=True,
    num_stages=3,
    disallow_acc_multi_buffer=True,
    flatten=True,
):
```

```
config.num_warps = 16
config.num_stages = 4
```

```
kernel[...](..., num_warps=16, num_stages=4)
```

Existing Triton tunables are automatically explored by Helion autotuner

Automatic Masking

Helion

v = (x+1).sum(1)

Output Triton (Helion settings.static_shapes = False)

```
tmp0 = x + 1
tmp1 = tl.where(mask_0[:, None] & mask_1[None, :], tmp0, 0)
v = tl.sum(tmp1, 1)
```

Output Triton (Helion settings.static_shapes = True + divisible sizes)

```
tmp0 = x + 1
v = tl.sum(tmp0, 1)
```

`x+1` turns zero masked elements coming from the load into ones, so Helion automatically inserts an extra `tl.where` mask.
This is easy to get wrong in Triton.

User-define autotuner knobs

```
my_flag = hl.register_tunable("my_flag", EnumFragment(choices=(0,1,2)))
...
for tile in hl.tile(x.size()):
    ...
    if my_flag == 0:
        ...
    elif my_flag == 1:
        ...
    else:
        ...
```

Supports: Enum, Boolean, Integer, PowerOfTwo, Lists, and user-defined parameter types

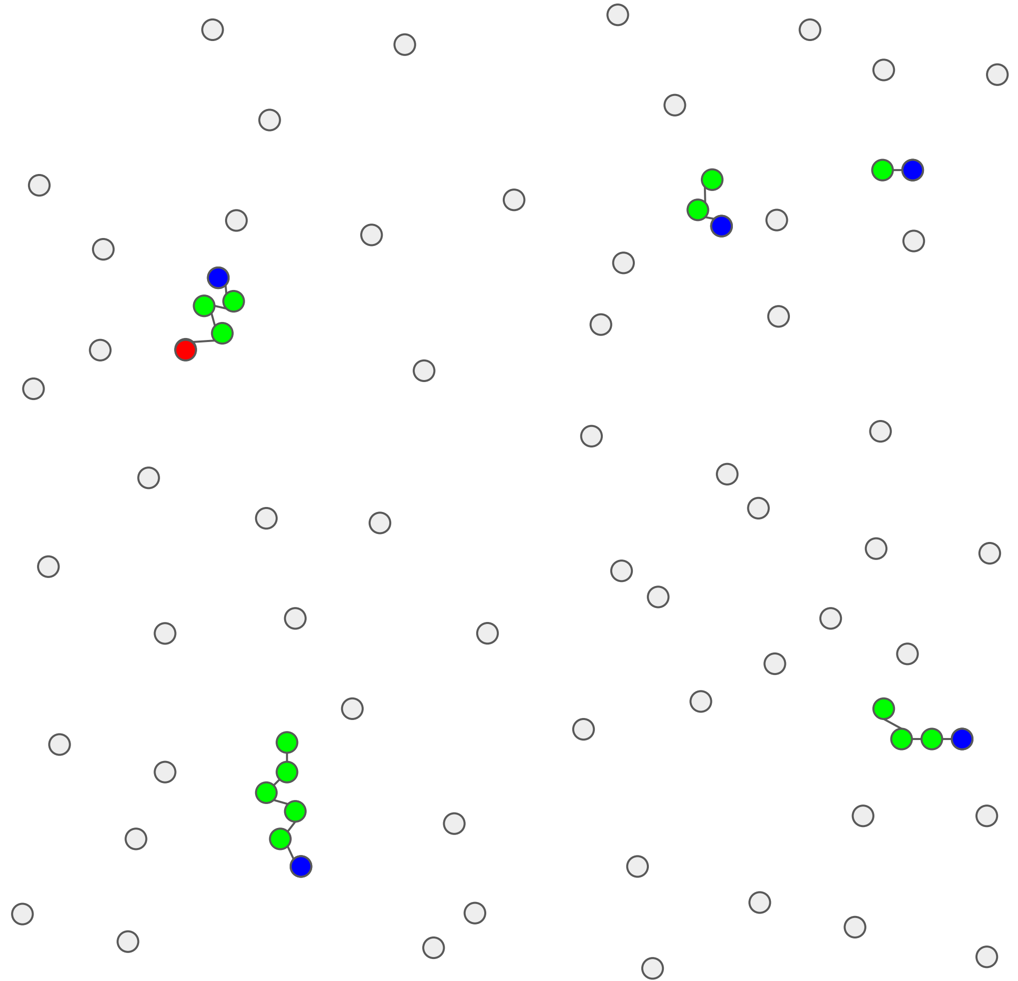
Autotuning

Autotuning Algorithms

- Takes about 10 minutes to search 1200 candidate Triton kernels
- Today:
    - Pattern search
    - Differential evolution
    - Finite search
    - DE-Surrogate hybrid
    - (default) LFBO pattern search
- In the future:
    - LLM-guided search
    - Reinforcement learning
    - Shared performance databases
    - Other search techniques

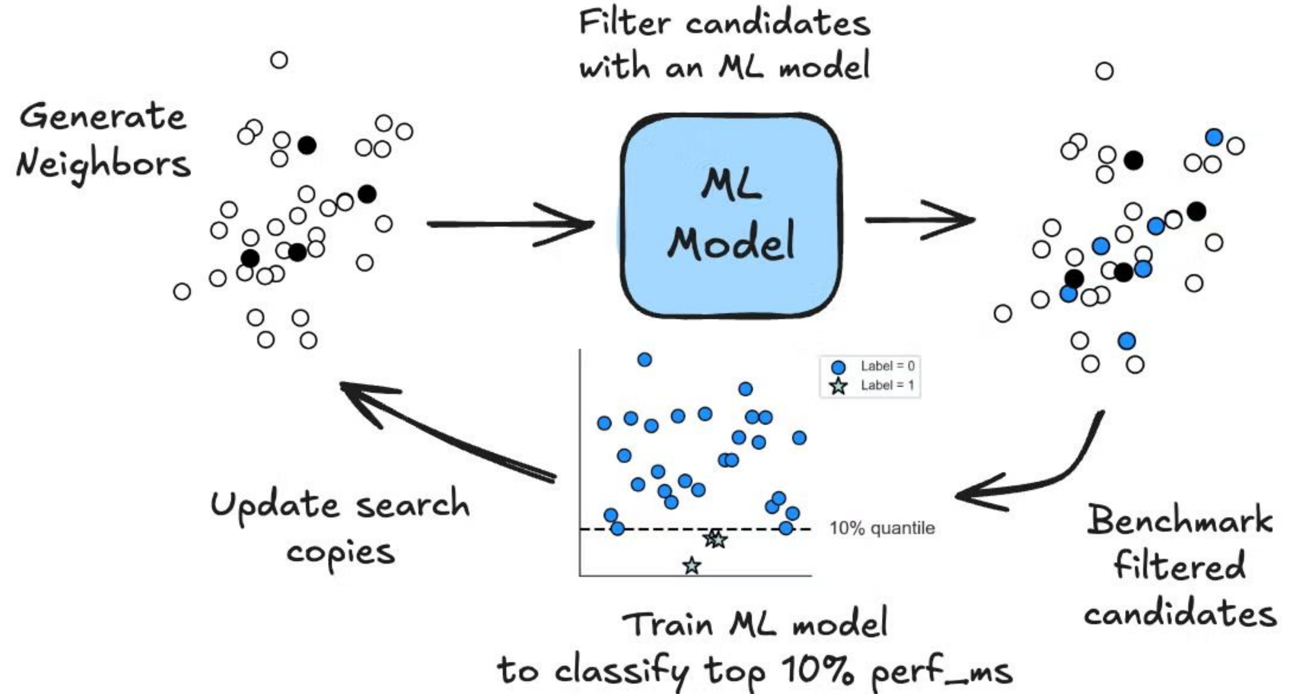
Pattern search

1. Initial population of 100 random configs
2. Select top 5 fastest
3. Hill-climb to local minimums
4. Select fastest

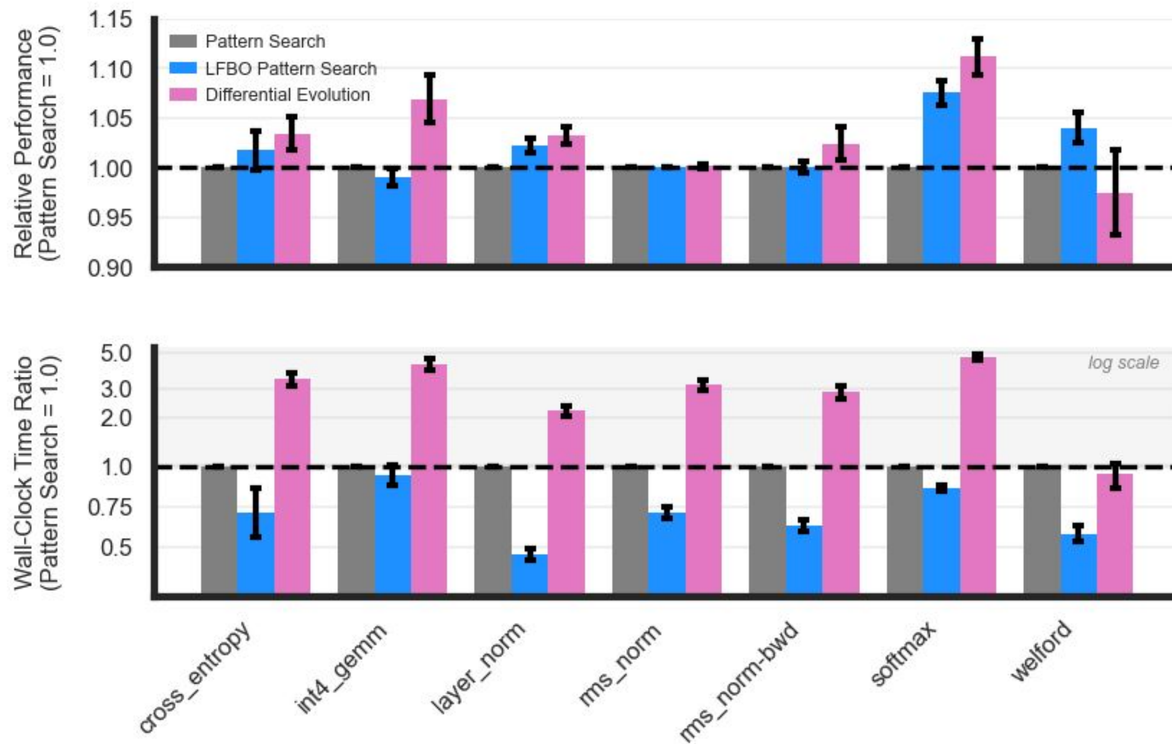


LFBO Pattern Search (Default)

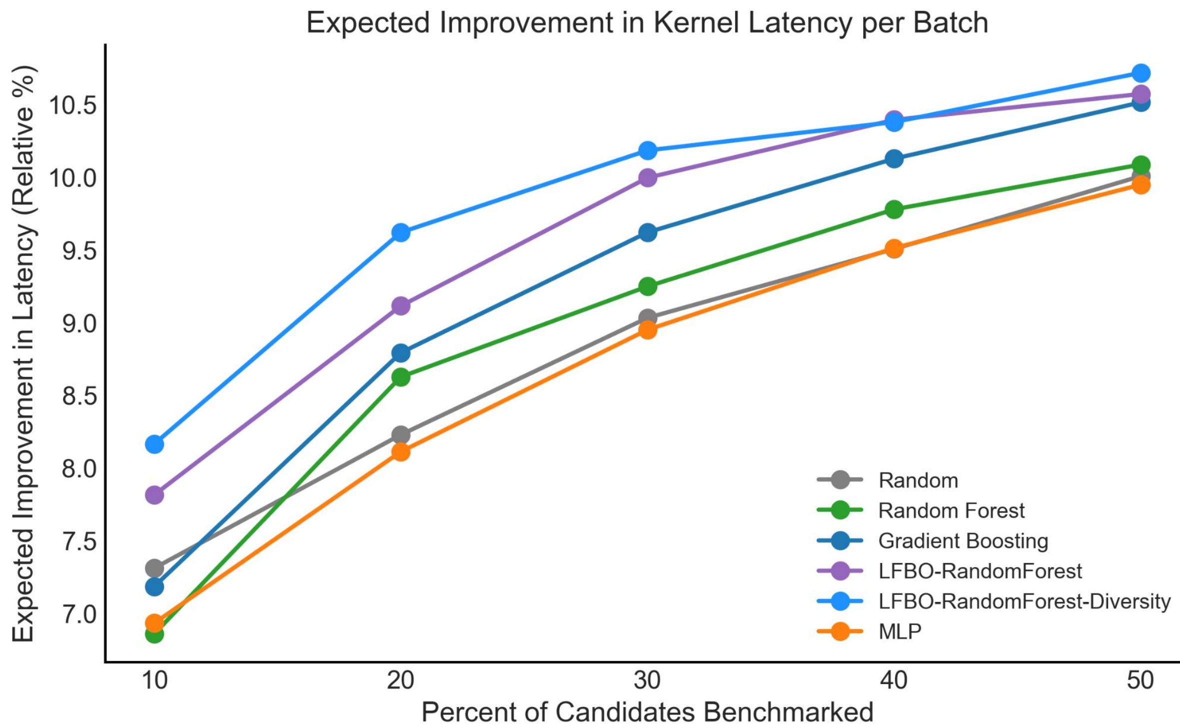
- Likelihood-Free Bayesian Optimization (LFBO) Pattern Search
- Random Forest classifier as a surrogate model to select which configurations to benchmark



Comparison of autotuners on NVIDIA B200



Comparison of expected improvements across surrogate models on Nvidia B200



Deployment

Deploying a single autotuned configuration

```python
@helion.kernel(config=helion.Config(
    block_sizes=[64, 64, 64],
    loop_orders=[[0, 1]],
    l2_groupings=[4],
    range_unroll_factors=[0, 1],
    range_warp_specializes=[None, False],
    range_num_stages=[0, 3],
    range_multi_buffers=[None, False],
    range_flattens=[None, None],
    num_warps=8,
    num_stages=6,
    indexing='block_ptr',
    pid_type='flat'
))
def matmul(x: torch.Tensor, y: torch.Tensor) -> torch.Tensor:
```

Autotuning should be done ahead of time in a development/staging environment

Autotuning and deploying multiple configurations

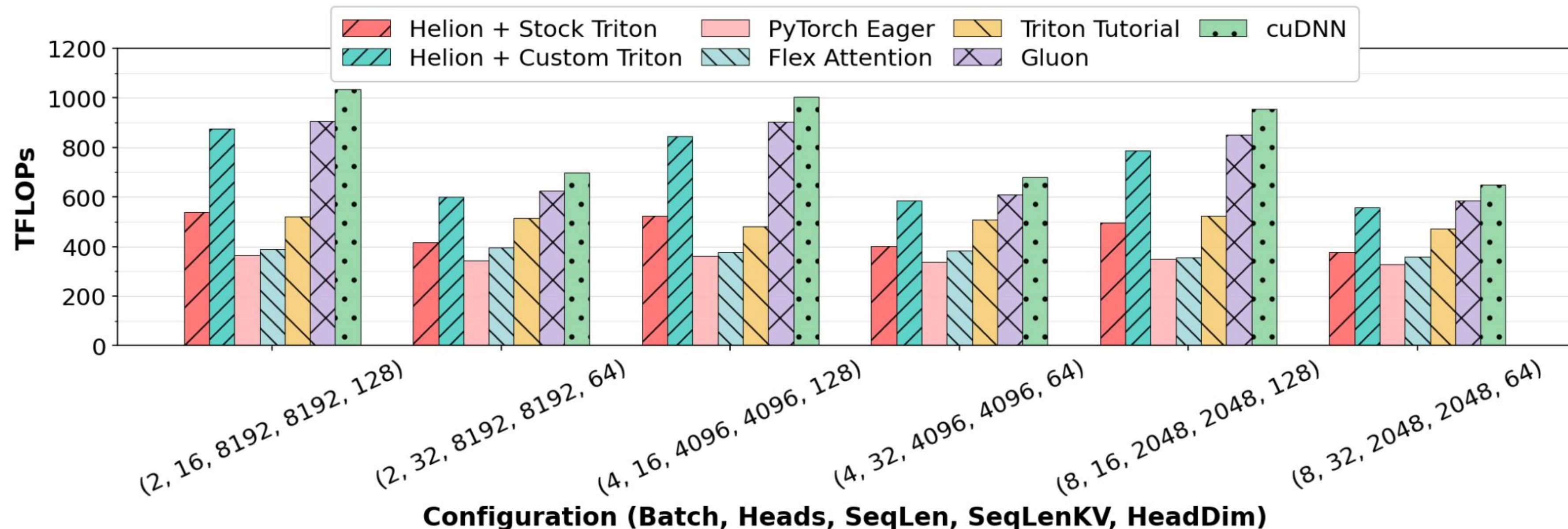
```
datasets = {
  "small": (
      torch.randn(2**16, device="cuda"),
      torch.randn(2**16, device="cuda"),
  ),
  "large": (
      torch.randn(2**24, device="cuda"),
      torch.randn(2**24, device="cuda"),
  ),
}

for tag, args in datasets.items():
  config = my_kernel.autotune(args)
  config.save(f"my_kernel_{tag}.json")
```

```
@helion.kernel(configs=[
    helion.Config.load("my_kernel_small.json"),
    helion.Config.load("my_kernel_large.json"),
])
def my_kernel(x, y):
    ...
```

Will try each configuration for each unique shape and pick the fastest.

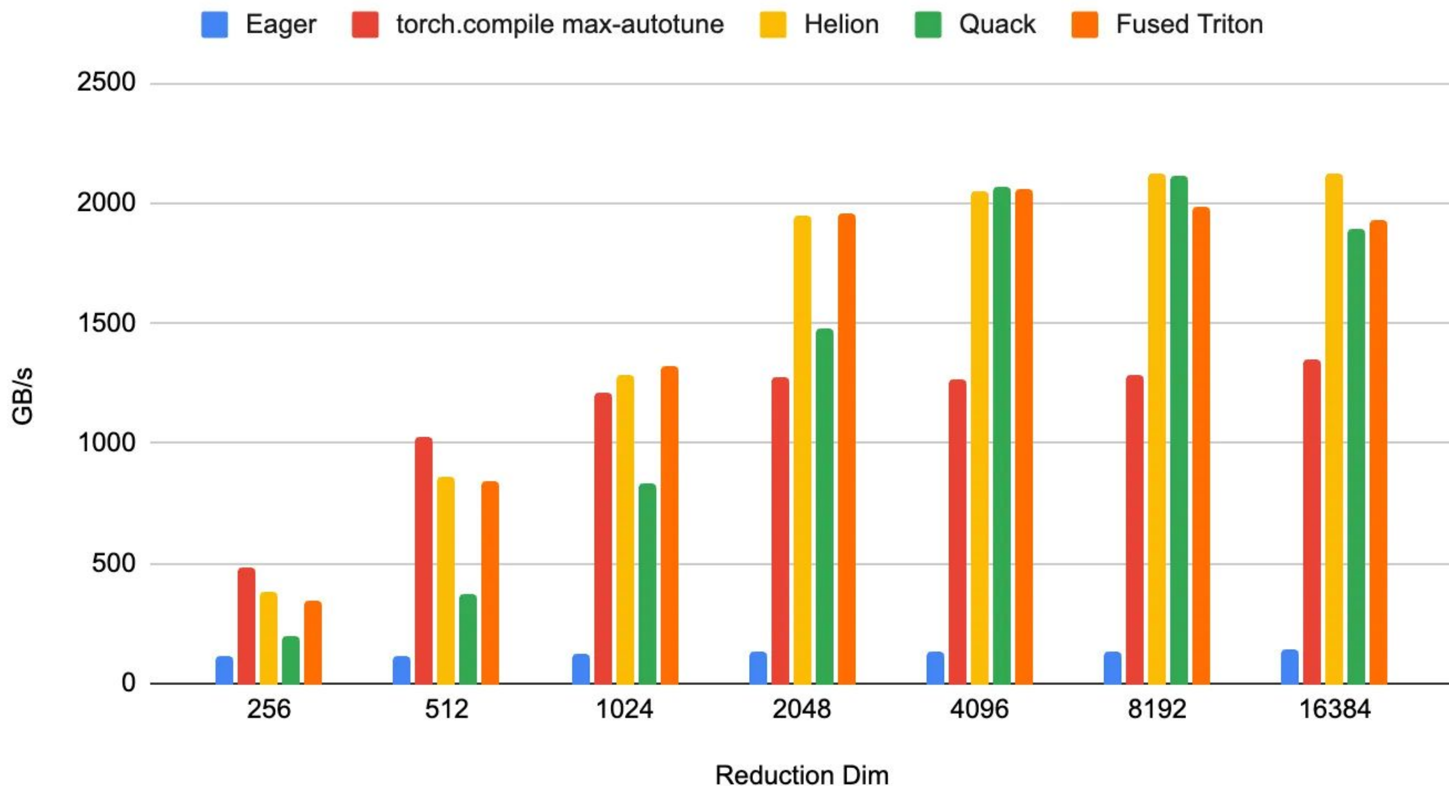
Performance

Case Study 1: Attention Performance Comparison (NVIDIA B200 - 750W, 1965Mhz)

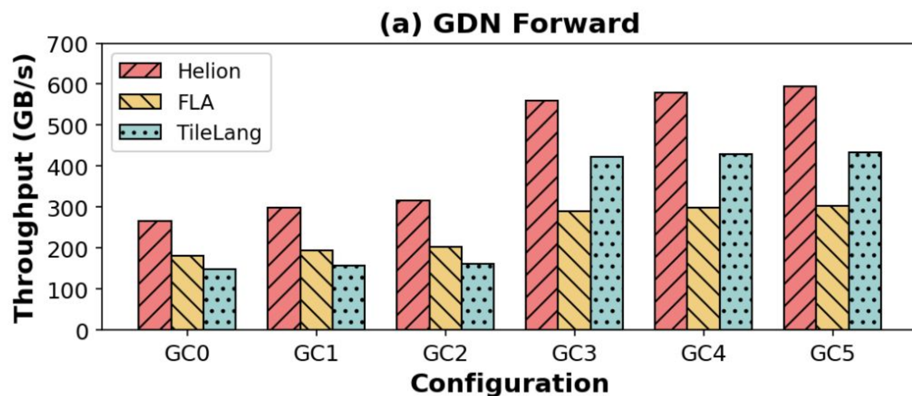


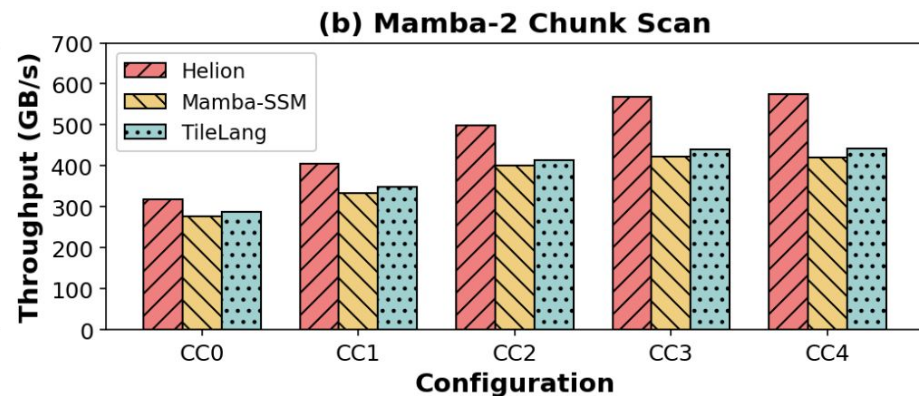
Custom Triton = FB Triton + NVIDIA CompileIQ
FB Triton branch: https://github.com/facebookexperimental/triton/tree/ws-3.5

Case Study 2: Comparison to CuTeDSL/Quack
For RMSNorm backwards (M=32768, NVIDIA H100 - 500W, 1620 MHz)



Case Study 3: Comparison to TileLang and FLA For GDN Forward and Mamba-2-chunk-scan kernel (NVIDIA B200 - 750 W, 1965 MHz)





TileLang: https://github.com/tile-ai/tilelang
FLA: https://github.com/fla-org/flash-linear-attention

NVIDIA B200 Speedups Over Eager PyTorch

Kernel	Helion Speedup	Torch Compile Speedup	Triton Speedup
cross_entropy	3.52	2.18	1.09
gemm	0.85	0.69	0.70
int4_gemm	7.53	8.89	3.23
jsd	8.89	8.99	1.43
kl_div	3.80	3.25	3.76
layer_norm	1.92	1.69	1.47
layer_norm-bwd	1.54	1.12	0.70
rms_norm	5.78	5.73	5.07
rms_norm-bwd	4.97	4.17	3.28
softmax	3.49	1.53	2.84
welford	2.00	2.04	0.79
geomean	**3.27**	**2.70**	**1.76**

AMD MI350X Speedups Over Eager PyTorch

Kernel	Helion Speedup	Torch Compile Speedup	Triton Speedup
geglu	1.01	1.01	1.01
gemm (fp16)	1.17	1.27	1.21
int4_gemm	7.58	5.98	1.68
jsd	12.07	12.06	2.75
kl_div	3.69	3.63	3.44
layer_norm	1.18	1.11	1.10
rms_norm	3.11	3.00	3.18
softmax	1.58	1.32	1.30
swiglu	1.04	1.04	1.04
geomean	**2.37**	**2.26**	**1.65**

April 2026:
Joining the PyTorch Foundation!

Multi-org maintainers

New experts contributing alongside Meta

First one from THINKING MACHINES

Community governed roadmap

New **Technical Steering Committee** under the PyTorch Foundation to set priorities openly

For the ecosystem

More **multi-hardware** support with new backends coming

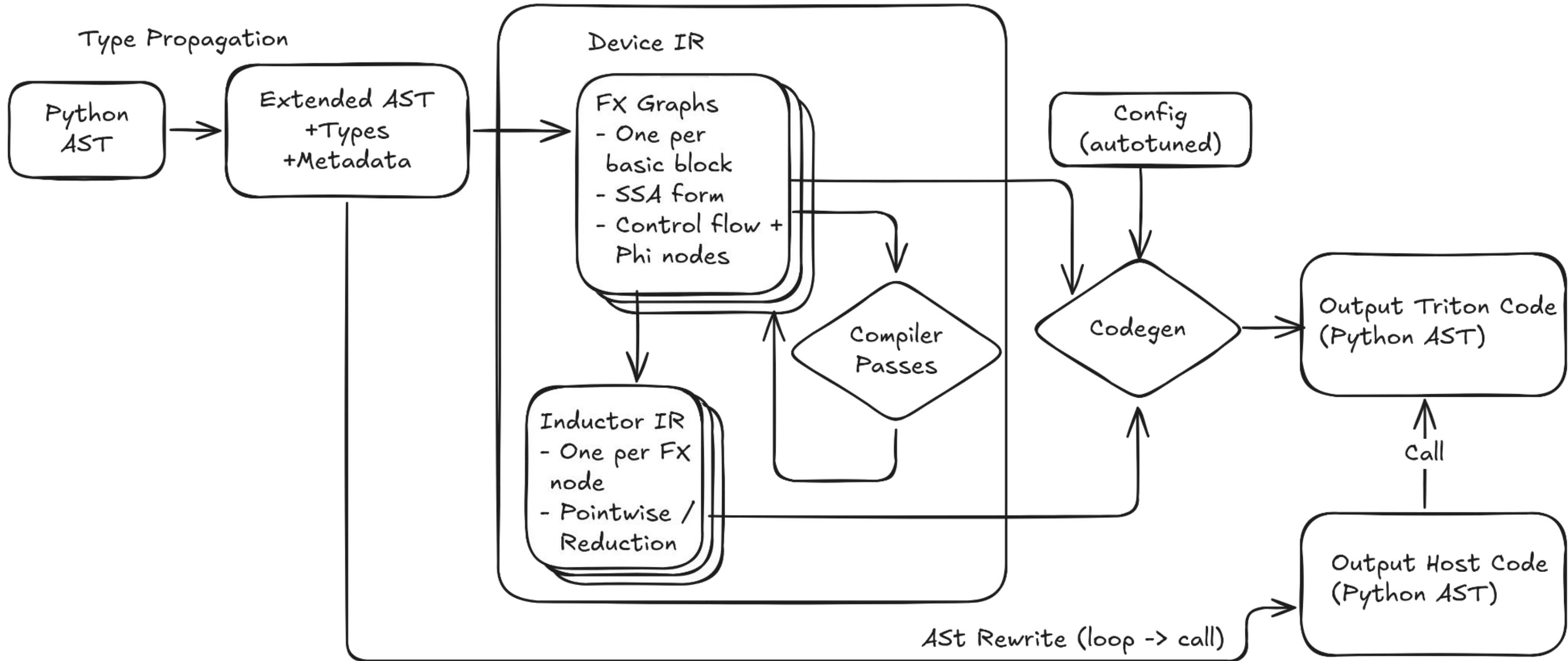
Vendor-neutral, community-first

All contributions welcome!

What’s happening in 2026

- Technical Roadmap
  - Distributed support
  - Automatically generate backward kernels
  - Prologue and epilogue fusions
  - New backends (Pallas, CuteDSL and Metal)
  - vLLM x Helion collaboration
- Helion Tutorial @ PDLI 2026 (June 2026)
- Stay tuned for more events!

Compiler Internals



Try Helion Today!

Open source contributions are welcome!
Connect with us on **GPU MODE Discord** in the #helion channel

Code: https://github.com/pytorch/helion

Docs: https://helionlang.com

Blog: https://pytorch.org/blog/helion/