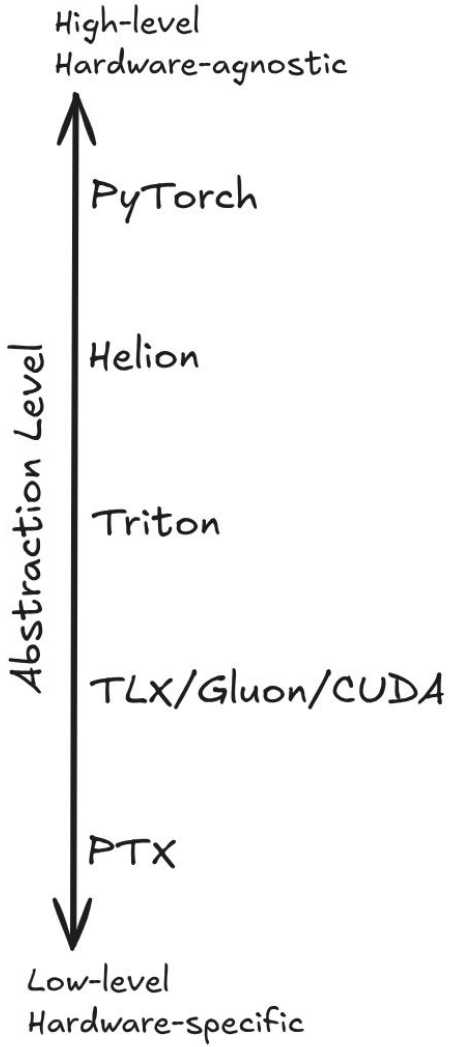
Helion:
A high-level DSL for ML kernels

Jason Ansel
PyTorch Compilers @ Meta
2025-08-05

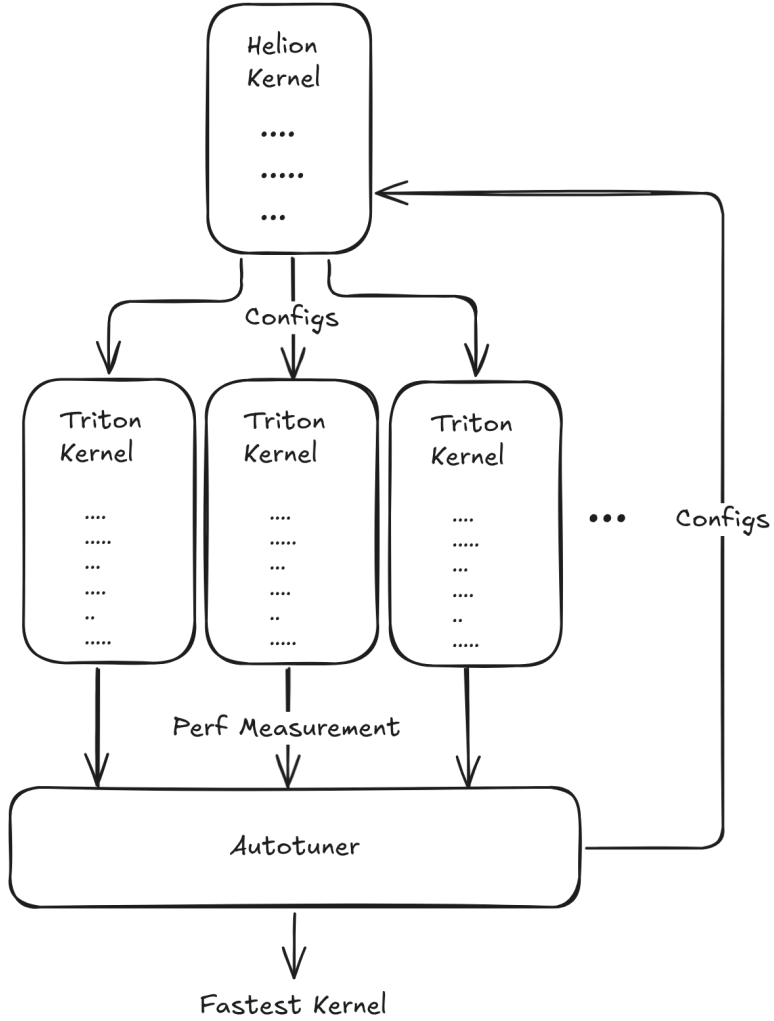
Motivation: Why a new DSL?

- 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
- Better PyTorch 2 interoperability
- 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)
- Once autotuning is done, user can copy configuration into their code to skip autotuning



Matrix Multiply Example

```
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

Templating via Closures

```
@helion.kernel
def matmul(x: torch.Tensor, y: torch.Tensor, epilogue):
    n, k = x.size()
    k, m = y.size()
    out = torch.empty([n, m], dtype=x.dtype)
    for n_tile, m_tile in hl.tile((n, m)):
        acc = hl.zeros([n_tile, m_tile], dtype=x.dtype)
        for k_tile in hl.tile(k):
            acc += torch.matmul(x[n_tile, k_tile], y[k_tile, m_tile])
        out[n_tile, m_tile] = epilogue(acc, (n_tile, m_tile))
    return out

bias = torch.randn(....)  # bias is captured by the epilogue lambda closure
result = matmul(x, y, lambda acc, tile: torch.sigmoid(acc + bias[tile]))
```

Autotuner Usage

- Takes about 10 minutes to search 1500 candidate Triton kernels
- Today:
  - Differential Evolution
  - Finite search
  - Random search
- In the future:
  - LLM-guided search
  - Reinforcement learning
  - Shared performance databases
- Autotuning ends telling user how to hardcode config:

```
[593s] Autotuning complete in 593.1s after searching 1520 configs.
One can hardcode the best config and skip autotuning with:
    @helion.kernel(config=helion.Config(block_sizes=[[64, 128], [16]], loop_orders=[[1,
```

Update your kernel to skip autotuning

```
@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:
```

Can also specify a list of configs and Helion will pick the fastest.

Attention Kernel

Attention code size

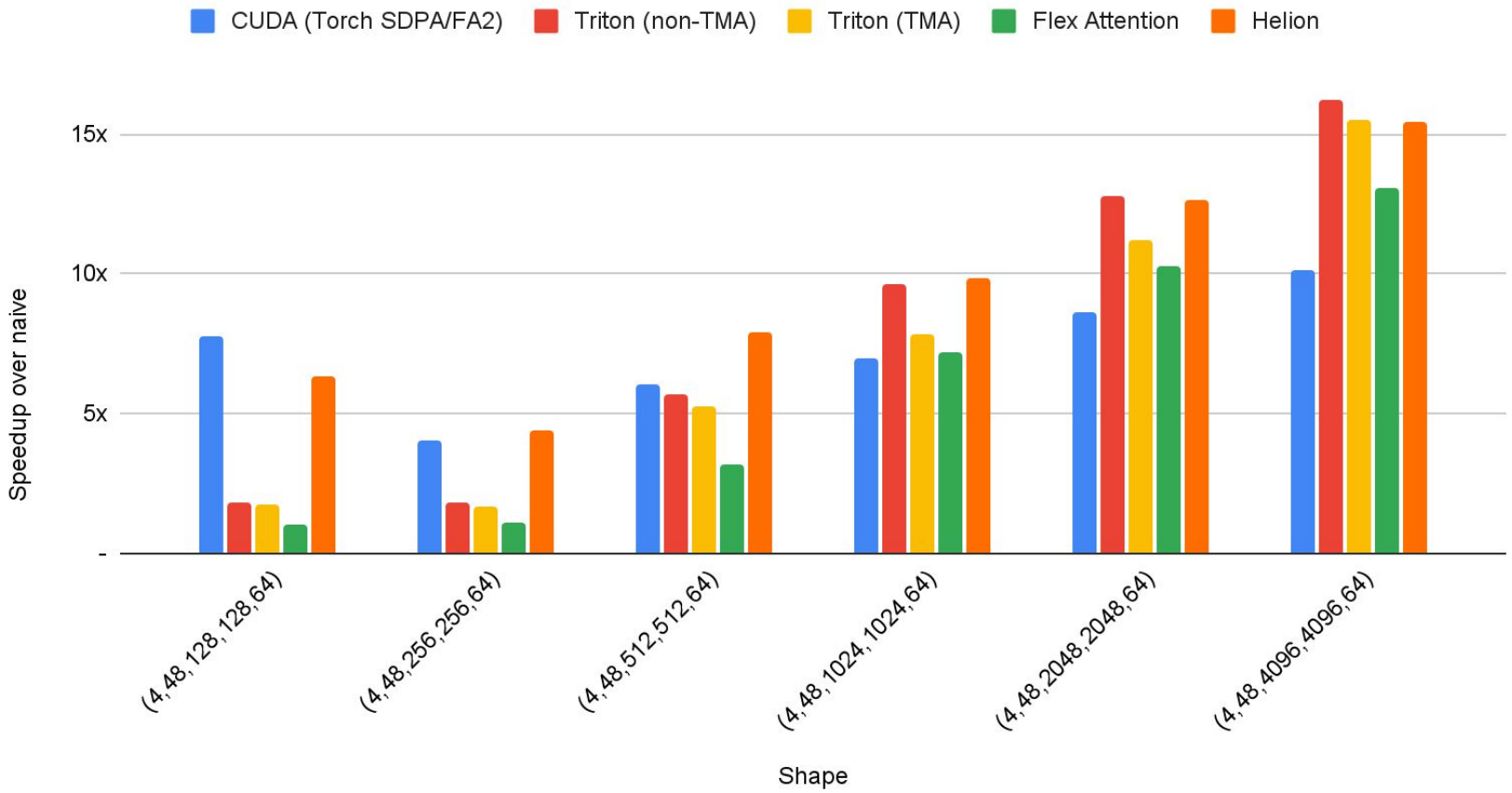
- 1 line in PyTorch
- 30 lines in Helion
- 120 lines in Triton
- 1000+ of lines in CUDA

Lower level languages specialize

- Triton/CUDA versions hard-code assumptions about tensor layouts
- CUDA versions hard-code assumptions about hardware (different code for V100/H100/B200)

```python
@helion.kernel
def attention(q_in: torch.Tensor, k_in: torch.Tensor, v_in: torch.Tensor):
    m_dim, n_dim, head_dim = q_in.size(-2), k_in.size(-2), q_in.size(-1)
    q_view = q_in.reshape([-1, m_dim, head_dim])
    v_view = v_in.reshape([-1, n_dim, head_dim])
    k_view = k_in.reshape([-1, n_dim, head_dim]).transpose(1, 2)
    out = torch.empty_like(q_view)
    qk_scale = 1.0 / math.sqrt(head_dim) * 1.44269504  # 1/log(2)
    for tile_b, tile_m in hl.tile([q_view.size(0), m_dim], block_size=[1, None]):
        m_i = hl.full([tile_b, tile_m], float("-inf"), dtype=torch.float32)
        l_i = torch.full_like(m_i, 1.0)
        acc = hl.zeros([tile_b, tile_m, head_dim], dtype=torch.float32)
        q = q_view[tile_b, tile_m, :]
        for tile_n in hl.tile(v_view.size(1)):
            k = k_view[tile_b, :, tile_n]
            qk = torch.bmm(q, k)
            m_ij = torch.maximum(m_i, torch.amax(qk, -1) * qk_scale)
            qk = qk * qk_scale - m_ij[:, :, None]
            p = torch.exp2(qk)
            l_ij = torch.sum(p, -1)
            alpha = torch.exp2(m_i - m_ij)
            l_i = l_i * alpha + l_ij
            acc = acc * alpha[:, :, None]
            v = v_view[tile_b, tile_n, :]
            p = p.to(v.dtype)
            acc = torch.baddbmm(acc, p, v)
            m_i = m_ij
        m_i += torch.log2(l_i)
        acc = acc / l_i[:, :, None]
        out[tile_b, tile_m, :] = acc.to(out.dtype)
    return out.view(q_in.size())
```

NVIDIA B200 Attention Speedups



NVIDIA B200 Average Speedups Over Eager

Kernel	**PyTorch Eager**	**Hand-written Triton**	**Helion**
rms_norm (3 shapes)	1x	1.9x	**4.0x**
layer_lorm (30 shapes)	1x	1.3x	**1.9x**
softmax (74 shapes)	1x	5.3x	**6.1x**
cross_entropy (6 shapes)	1x	0.9x	**3.1x**
sum (3 shapes)	1x	0.8x	**1.2x**
jagged_mean (4 shapes)	1x	44.4x	**113.6x**
embedding (12 shapes)	1x	1.1x	**1.3x**

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'))
```

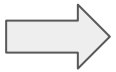
Output Triton (Helion config.indexing = "tensor_descriptor")

```
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

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



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

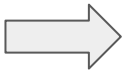
```
@triton.jit
def _add_kernel(...,
                _BLOCK_SIZE_0: tl.constexpr,
                _BLOCK_SIZE_1: tl.constexpr):
    num_blocks_0 = tl.cdiv(x_size_0, _BLOCK_SIZE_0)
    pid_0 = tl.program_id(0) % num_blocks_0
    pid_1 = tl.program_id(0) // num_blocks_0
    offset_0 = pid_0 * _BLOCK_SIZE_0
    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

```
v = x[tile0, :].sum(1)
```

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)

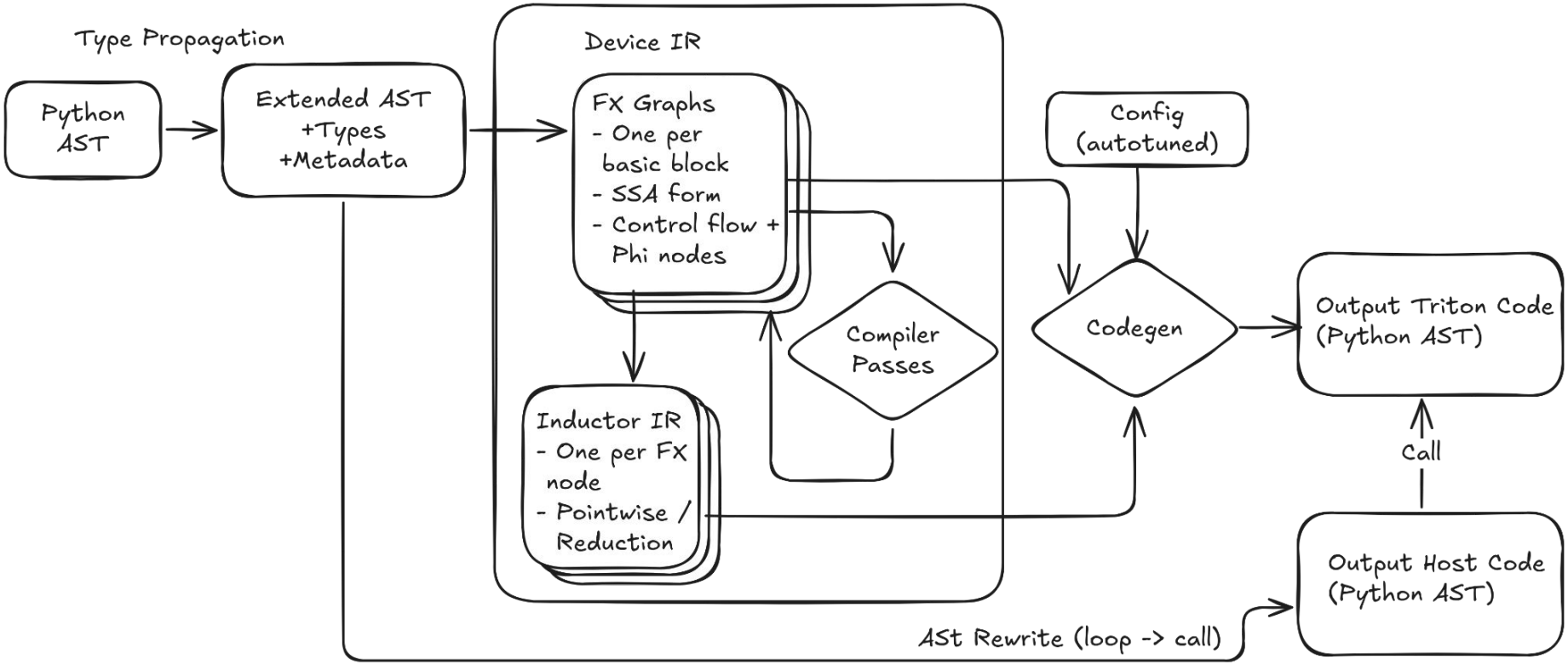
```
tmp0 = x + 1
tmp1 = tl.where(mask_0[:, None] & mask_1[None, :], tmp0, 0)
v = tl.sum(tmp1, 1)
```

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.

Compiler Internals



Current/Future Work

- New language features
  - Close expressability gaps
- New autotuning approaches
  - Performance model driven search
  - Reinforcement learning / LLMs
  - Additional empirical search algorithms
- Improving caching for autotuning
- New hardware backends
- Mixed communication/compute kernels
- Better performance
  - Collaborating with Triton teams on exposing more knobs to autotuning
- More docs, tutorials, and user testing

Try Helion Today!

- **Early Development Warning:** Helion is currently in an experimental stage. You should expect bugs, incomplete features, and APIs that may change in future versions.
- Planning beta release at PyTorch Conference (Oct 22-23, 2025)
- Helion is usable today, and we would love for you try it out and tell us what you think.
- Early feedback and bug reports will help shape the language.
- Open source contributions welcome!

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

Docs: https://helionlang.com