# Custom PyTorch Kernels with IREE and Turbine

Kunwar Grover

# Introduction

PyTorch

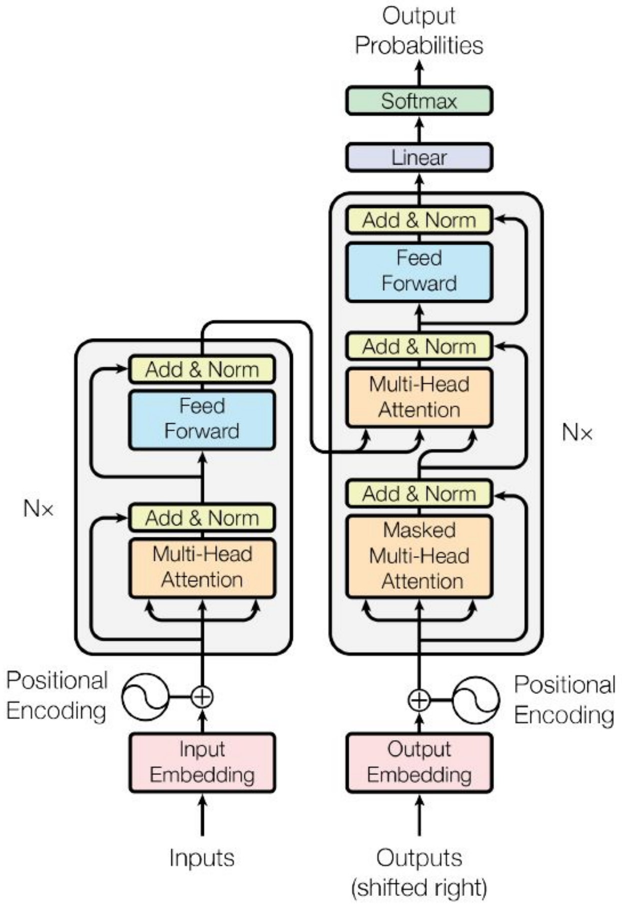
IREE

MLIR-based end-to-end compiler and runtime for Machine Learning

IREE: https://github.com/openxla/iree

# Writing a Neural Network

PyTorch



# Profiling

| | | | | |
|---|---|---|---|---|
| main$async_dispatch_220_attention_40x1024x64xf16 | main$async_dispatch_220_attention_40x1024x64xf16 | 210.93 ms (6.66%) | 960 | 219.72 μs |
| main$async_dispatch_214_matmul_transpose_b_2048x1280x1280_f16 | main$async_dispatch_214_matmul_transpose_b_2048x1280x1280_f16 | 154.56 ms (4.88%) | 3,840 | 40.25 μs |
| main$async_dispatch_236_matmul_transpose_b_2048x10240x1280_f16xf16xf32 | main$async_dispatch_236_matmul_transpose_b_2048x10240x1280_f16xf16xf32 | 143.72 ms (4.54%) | 960 | 149.71 μs |
| main$async_dispatch_70_attention_20x4096x64xf16 | main$async_dispatch_70_attention_20x4096x64xf16 | 127.02 ms (4.01%) | 160 | 793.86 μs |
| main$async_dispatch_238_matmul_transpose_b_2048x1280x5120_f16xf16xf32 | main$async_dispatch_238_matmul_transpose_b_2048x1280x5120_f16xf16xf32 | 114.78 ms (3.63%) | 960 | 119.57 μs |
| main$async_dispatch_217_transpose_2x20x1024x64_f16 | main$async_dispatch_217_transpose_2x20x1024x64_f16 | 89.22 ms (2.82%) | 3,840 | 23.23 μs |
| main$async_dispatch_205_conv_2d_nhwc_hwcf_2x32x32x1280x3x3x1280_f16 | main$async_dispatch_205_conv_2d_nhwc_hwcf_2x32x32x1280x3x3x1280_f16 | 82.35 ms (2.60%) | 160 | 514.72 μs |
| main$async_dispatch_222_matmul_transpose_b_2048x1280x1280_f16xf16xf32 | main$async_dispatch_222_matmul_transpose_b_2048x1280x1280_f16xf16xf32 | 78.52 ms (2.48%) | 1,968 | 39.9 μs |
| main$async_dispatch_226_matmul_transpose_b_128x1280x2048_f16 | main$async_dispatch_226_matmul_transpose_b_128x1280x2048_f16 | 69.19 ms (2.19%) | 1,920 | 36.04 μs |
| main$async_dispatch_213_generic_2048x1280_f16xf32xf32xf16xf16xf16 | main$async_dispatch_213_generic_2048x1280_f16xf32xf32xf16xf16xf16 | 49.59 ms (1.57%) | 2,880 | 17.22 μs |
| main$async_dispatch_221_transpose_2x1024x20x64_f16 | main$async_dispatch_221_transpose_2x1024x20x64_f16 | 39.87 ms (1.26%) | 1,920 | 20.77 μs |
| main$async_dispatch_212_generic_2048x1280_f32 | main$async_dispatch_212_generic_2048x1280_f32 | 34.92 ms (1.10%) | 2,880 | 12.13 μs |
| main$async_dispatch_19_conv_2d_nhwc_hwcf_2x128x128x320x3x3x320_f16 | main$async_dispatch_19_conv_2d_nhwc_hwcf_2x128x128x320x3x3x320_f16 | 31.53 ms (1.00%) | 96 | 328.43 μs |
| main$async_dispatch_55_conv_2d_nhwc_hwcf_2x64x64x640x3x3x640_f16 | main$async_dispatch_55_conv_2d_nhwc_hwcf_2x64x64x640x3x3x640_f16 | 30.88 ms (0.98%) | 96 | 321.71 μs |
| main$async_dispatch_1083_conv_2d_nhwc_hwcf_2x32x32x1280x3x3x2560_f16 | main$async_dispatch_1083_conv_2d_nhwc_hwcf_2x32x32x1280x3x3x2560_f16 | 30.48 ms (0.96%) | 32 | 952.44 μs |
| main$async_dispatch_237_generic_2x1024x5120_f16 | main$async_dispatch_237_generic_2x1024x5120_f16 | 27.6 ms (0.87%) | 960 | 28.75 μs |
| main$async_dispatch_64_matmul_transpose_b_8192x640x640_f16 | main$async_dispatch_64_matmul_transpose_b_8192x640x640_f16 | 25.08 ms (0.79%) | 640 | 39.18 μs |
| main$async_dispatch_86_matmul_transpose_b_8192x5120x640_f16xf16xf32 | main$async_dispatch_86_matmul_transpose_b_8192x5120x640_f16xf16xf32 | 22.76 ms (0.72%) | 160 | 142.26 μs |
| main$async_dispatch_21_matmul_transpose_b_2x320x1280_f16xf16xf32 | main$async_dispatch_21_matmul_transpose_b_2x320x1280_f16xf16xf32 | 20.65 ms (0.65%) | 80 | 258.18 μs |
| main$async_dispatch_229_transpose_2x20x64x64_f16 | main$async_dispatch_229_transpose_2x20x64x64_f16 | 20.42 ms (0.65%) | 1,920 | 10.64 μs |
| main$async_dispatch_67_transpose_2x10x4096x64_f16 | main$async_dispatch_67_transpose_2x10x4096x64_f16 | 20.24 ms (0.64%) | 640 | 31.63 μs |
| main$async_dispatch_2250_conv_2d_nhwc_hwcf_2x128x128x4x3x3x320_f16 | main$async_dispatch_2250_conv_2d_nhwc_hwcf_2x128x128x4x3x3x320_f16 | 19.44 ms (0.61%) | 16 | 1.22 ms |
| main$async_dispatch_88_matmul_transpose_b_8192x640x2560_f16xf16xf32 | main$async_dispatch_88_matmul_transpose_b_8192x640x2560_f16xf16xf32 | 18.87 ms (0.60%) | 160 | 117.97 μs |
| main$async_dispatch_231_attention_40x1024x64xf16 | main$async_dispatch_231_attention_40x1024x64xf16 | 17.93 ms (0.57%) | 960 | 18.68 μs |
| main$async_dispatch_2218_conv_2d_nhwc_hwcf_2x128x128x320x3x3x640_f16 | main$async_dispatch_2218_conv_2d_nhwc_hwcf_2x128x128x320x3x3x640_f16 | 17.41 ms (0.55%) | 32 | 544.14 μs |
| main$async_dispatch_1958_conv_2d_nhwc_hwcf_2x64x64x1280x3x3x1280_f16 | main$async_dispatch_1958_conv_2d_nhwc_hwcf_2x64x64x1280x3x3x1280_f16 | 16.91 ms (0.53%) | 16 | 1.06 ms |

# Flash Attention

CUDA/HIP? > 1000 lines

Fused Attention → Flash Attention 2

```cpp
inline __device__ void compute_attn_1rowblock(const Params &params, const int bidb, const int bidh, const int m_block) {

    using Element = typename Kernel_traits::Element;
    using ElementAccum = typename Kernel_traits::ElementAccum;
    using index_t = typename Kernel_traits::index_t;

    // Shared memory.
    extern __shared__ char smem_[];

    // The thread index.
    const int tidx = threadIdx.x;

    constexpr int kBlockM = Kernel_traits::kBlockM;
    constexpr int kBlockN = Kernel_traits::kBlockN;
    constexpr int kHeadDim = Kernel_traits::kHeadDim;
    constexpr int kNWarps = Kernel_traits::kNWarps;

    auto seed_offset = at::cuda::philox::unpack(params.philox_args);
    flash::Dropout dropout(std::get<0>(seed_offset), std::get<1>(seed_offset), params.p_dropout_in_uint8_t,
                           bidb, bidh, tidx, params.h);

    // Save seed and offset for backward, before any early exiting. Otherwise the 0-th thread block might
    // exit early and no one saves the rng states.
    if (Is_dropout && blockIdx.x == 0 && blockIdx.y == 0 && blockIdx.z == 0 && tidx == 0) {
        params.rng_state[0] = std::get<0>(seed_offset);
        params.rng_state[1] = std::get<1>(seed_offset);
    }

    const BlockInfo</*Varlen=*/!Is_even_MN> binfo(params, bidb);
    if (m_block * kBlockM >= binfo.actual_seqlen_q) return;

    const int n_block_min = !Is_local ? 0 : std::max(0, (m_block * kBlockM + binfo.actual_seqlen_k - binfo.actual_seqlen_q - params.window_size_left) / kBlockN);
    int n_block_max = cute::ceil_div(binfo.actual_seqlen_k, kBlockN);
    if (Is_causal || Is_local) {
        n_block_max = std::min(n_block_max,
                               cute::ceil_div((m_block + 1) * kBlockM + binfo.actual_seqlen_k - binfo.actual_seqlen_q + params.window_size_right, kBlockN));
        // if (threadIdx.x == 0 && blockIdx.y == 0 && blockIdx.z == 0) {
        //     printf("m_block = %d, n_block_max = %d\n", m_block, n_block_max);
        // }
    }
    // We exit early and write 0 to gO and gLSE. This also covers the case where actual_seqlen_k == 0.
    // Otherwise we might read OOB elements from gK and gV.
    if ((Is_causal || Is_local || !Is_even_MN) && n_block_max <= n_block_min) {
        const index_t row_offset_o = binfo.q_offset(params.o_batch_stride, params.o_row_stride, bidb)
            + m_block * kBlockM * params.o_row_stride + bidh * params.o_head_stride;
        const index_t row_offset_lse = (bidb * params.h + bidh) * params.seqlen_q + m_block * kBlockM;
        Tensor gO = make_tensor(make_gmem_ptr(reinterpret_cast<Element *>(params.o_ptr) + row_offset_o),
                                Shape<Int<kBlockM>, Int<kHeadDim>>{},
```

# Flash Attention

Block DSLs < 50 lines

Fused Attention → Flash Attention 2

```
@triton.jit
def _attn_fwd_inner(acc, l_i, m_i, q,  #
                    K_block_ptr, V_block_ptr,  #
                    start_m, qk_scale,  #
                    BLOCK_M: tl.constexpr, BLOCK_DMODEL: tl.constexpr, BLOCK_N: tl.constexpr,
                    STAGE: tl.constexpr, offs_m: tl.constexpr, offs_n: tl.constexpr,  #
                    N_CTX: tl.constexpr):
    # range of values handled by this stage
    if STAGE == 1:
        lo, hi = 0, start_m * BLOCK_M
    elif STAGE == 2:
        lo, hi = start_m * BLOCK_M, (start_m + 1) * BLOCK_M
        lo = tl.multiple_of(lo, BLOCK_M)
    # causal = False
    else:
        lo, hi = 0, N_CTX
    K_block_ptr = tl.advance(K_block_ptr, (0, lo))
    V_block_ptr = tl.advance(V_block_ptr, (lo, 0))
    # loop over k, v and update accumulator
    for start_n in range(lo, hi, BLOCK_N):
        start_n = tl.multiple_of(start_n, BLOCK_N)
        # -- compute qk ----
        k = tl.load(K_block_ptr)
        qk = tl.zeros([BLOCK_M, BLOCK_N], dtype=tl.float32)
        qk += tl.dot(q, k)
        if STAGE == 2:
            mask = offs_m[:, None] >= (start_n + offs_n[None, :])
            qk = qk * qk_scale + tl.where(mask, 0, -1.0e6)
            m_ij = tl.maximum(m_i, tl.max(qk, 1))
            qk -= m_ij[:, None]
        else:
            m_ij = tl.maximum(m_i, tl.max(qk, 1) * qk_scale)
            qk = qk * qk_scale - m_ij[:, None]
        p = tl.math.exp2(qk)
        l_ij = tl.sum(p, 1)
        # -- update m_i and l_i
        alpha = tl.math.exp2(m_i - m_ij)
        l_i = l_i * alpha + l_ij
        # -- update output accumulator --
        acc = acc * alpha[:, None]
        # update acc
        v = tl.load(V_block_ptr)
        acc += tl.dot(p.to(tl.float16), v)
        # update m_i and l_i
        m_i = m_ij
        V_block_ptr = tl.advance(V_block_ptr, (BLOCK_N, 0))
        K_block_ptr = tl.advance(K_block_ptr, (0, BLOCK_N))
    return acc, l_i, m_i
```

# Choosing the Right Abstraction

Tensor Level DSLs

Block Level DSLs

CUDA/HIP

Assembly?

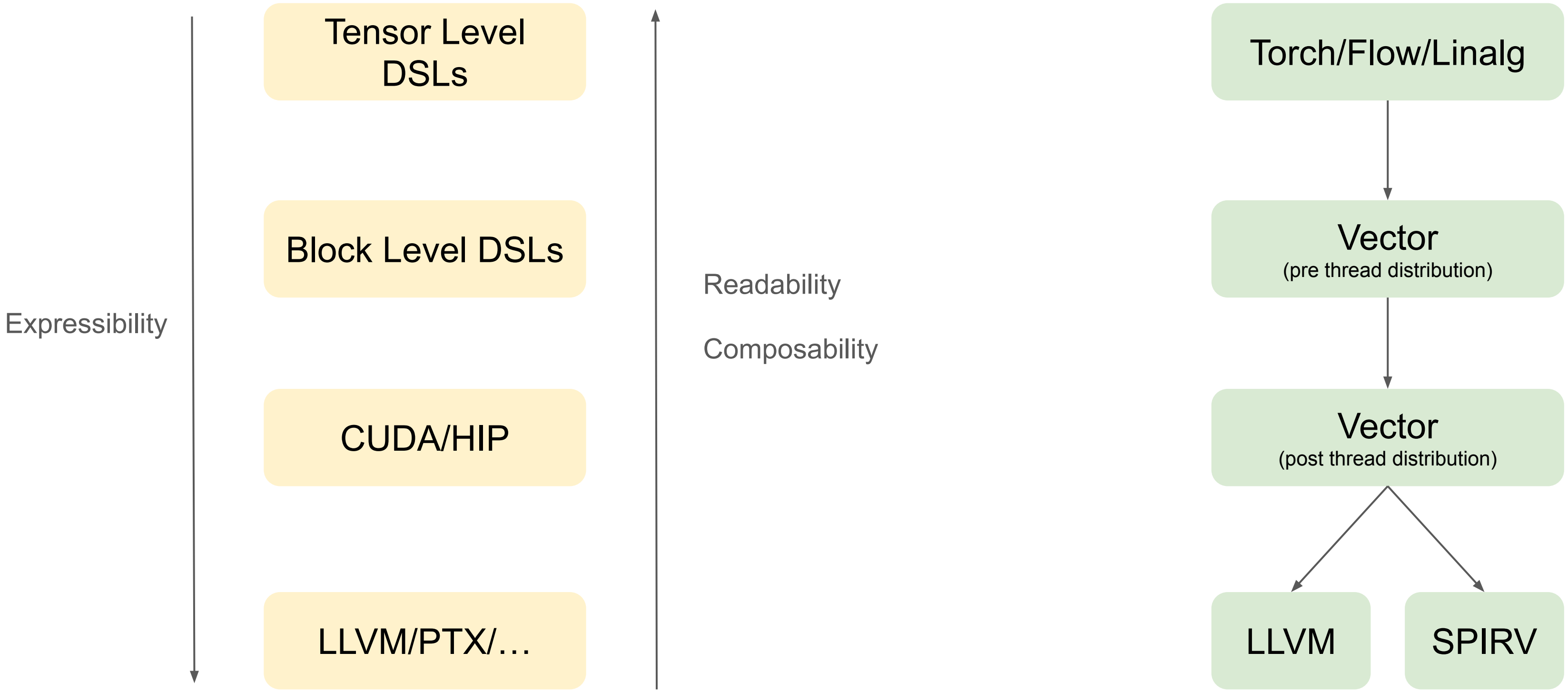
Expressibility

Readability

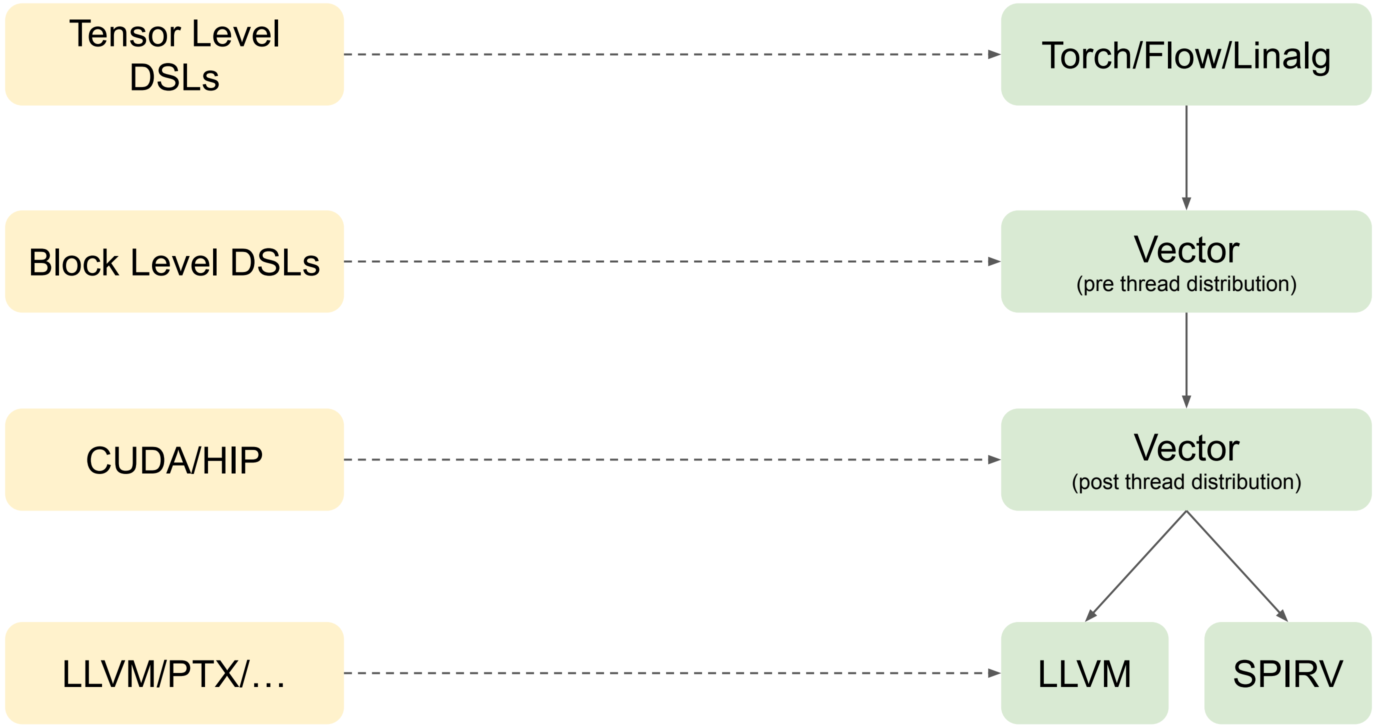
Composability

# Choosing the Right Abstraction

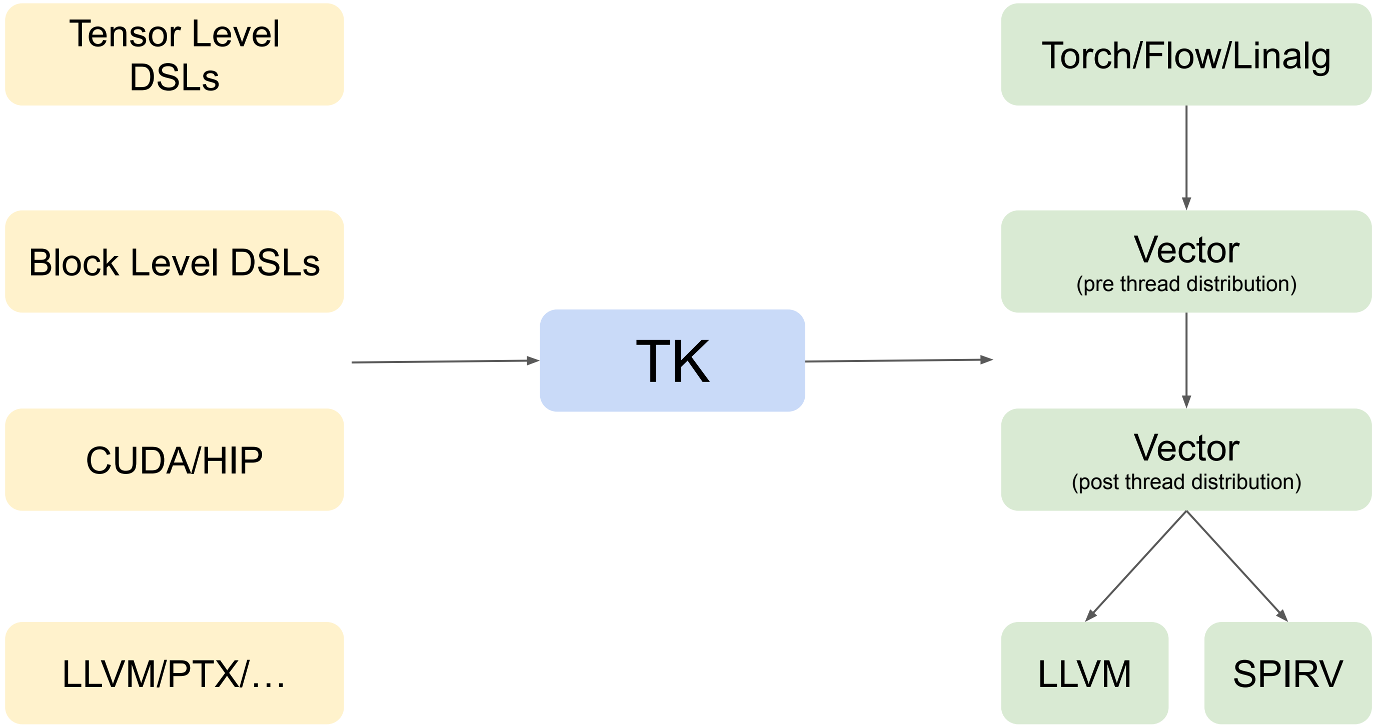
# Choosing the Right Abstraction



# Choosing the Right Abstraction

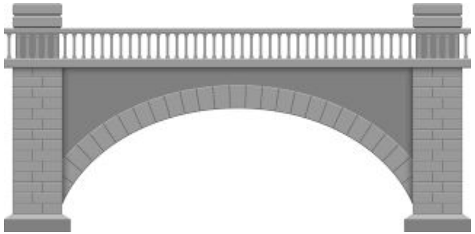


# Turbine Kernels



# Methodology

PyTorch



IREE

SHARK-Turbine

# SHARK-Turbine

torch.compile

torch.export

eager execution

SHARK-Turbine

# Need for a Kernel DSL

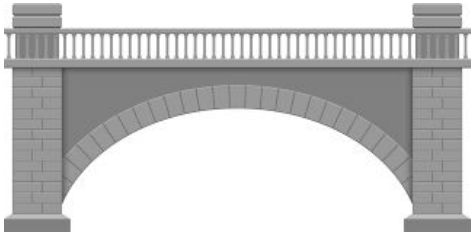
- Handwritten / Python bindings to try out new kernel variants

- Transform dialect to experiment with new kernels

- Needed a Kernel Language to target existing phases of our compiler

# Methodology

PyTorch



SHARK-Turbine
+ TK

IREE

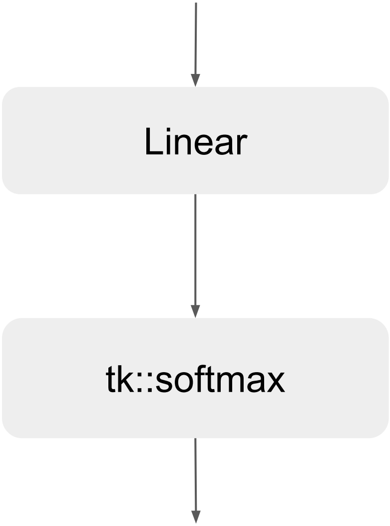
Already existing compiler

# Turbine Kernels

Named after TK, the cat

# Turbine Kernels

```python
import shark_turbine.kernel as tk

@tk.gen.kernel(...)
def softmax(...):
  ...


class NN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = torch.nn.Linear(64, 64)

    def forward(self, x):
        x = self.linear(x)
        x = softmax(x)
        return x
```



# Turbine Kernels

```
util.func public @main(%arg0: !hal.buffer_view,
                       %arg1: !hal.fence,
                       %arg2: !hal.fence) -> !hal.buffer_view {
  %cst = arith.constant 0.000000e+00 : f32
  %0 = hal.tensor.import %arg1 => %arg0 : !hal.buffer_view -> tensor<64x64xf32>
  %weight = util.global.load @weight : tensor<64x64xf32>

  // Linear
  %1 = tensor.empty() : tensor<64x64xf32>
  %2 = linalg.copy %weight
  %3 = linalg.fill ins(%cst : f32) outs(%1 : tensor<64x64xf32>)
  %4 = linalg.matmul ins(%0, %2 : tensor<64x64xf32>, tensor<64x64xf32>)
                     outs(%3 : tensor<64x64xf32>)
  %bias = util.global.load @bias : tensor<64xf32>
  %5 = linalg.add ins(%4, %bias : tensor<64x64xf32>, tensor<64xf32>)
                  outs(%1 : tensor<64x64xf32>)

  // TK Kernel Softmax
  %6 = flow.dispatch @tk_kernel_softmax::@tk_kernel_softmax(%5)
                     : (tensor<64x64xf32>) -> tensor<64x64xf32>

  %7 = hal.tensor.barrier join(%6 : tensor<64x64xf32>) => %arg2 : !hal.fence
  %8 = hal.tensor.export %7 : tensor<64x64xf32> -> !hal.buffer_view
  util.return %8 : !hal.buffer_view
}
```

# The Turbine Kernel Language

```python
import shark_turbine.kernel as tk
import shark_turbine.kernel.lang as tkl


M = tkl.sym.M
N = tkl.sym.K

@tk.gen.kernel(M)
def softmax(
    input: tkl.InputBuffer[M, N, tkl.f32],
    output: tkl.OutputBuffer[M, tkl.f32]
):
    row_index = tkl.program_id(0)
    row = tkl.load(input, (row_index, 0), (1, N))
    row_minus_max = row - tkl.max(row)
    numerator = tkl.exp2(row_minus_max)
    denominator = tkl.sum(numerator)
    softmax_output = numerator / denominator
    tkl.store(output, (row_index), softmax_output)
```

# The Turbine Kernel Language: Traced

```
import shark_turbine.kernel as tk
import shark_turbine.kernel.lang as tkl


M = tkl.sym.M
N = tkl.sym.K

@tk.gen.kernel(M)
def softmax(
    input: tkl.InputBuffer[M, N, tkl.f32],
    output: tkl.OutputBuffer[M, tkl.f32]
):
    row_index = tkl.program_id(0)
    row = tkl.load(input, (row_index, 0), (1, N))
    row_minus_max = row - tkl.max(row)
    numerator = tkl.exp2(row_minus_max)
    denominator = tkl.sum(numerator)
    softmax_output = numerator / denominator
    tkl.store(output, (row_index), softmax_output)
```

TK is symbolically traced

# The Turbine Kernel Language: Traced

```
region_0:graph():
    %input_1 : InputBuffer[N, M] = placeholder[target=input]
    %output : Output[N] = placeholder[target=output]
    %thread_program_id : SymIndex∈[0, M) = call_function[target=thread_program_id](args = (0,))
    %kernel_buffer_load = call_function[target=kernel_buffer_load](args = (%input_1, (%thread_program_id, 0), (1, K)))
    %vector_max = call_function[target=vector_max](args = (%kernel_buffer_load,))
    %sub = call_function[target=sub](args = (%kernel_buffer_load, %vector_max))
    %exp2 = call_function[target=exp2](args = (%sub,), kwargs = {})
    %vector_sum = call_function[target=vector_sum](args = (%exp2,))
    %truediv = call_function[target=truediv](args = (%exp2, %vector_sum))
    %kernel_buffer_store = call_function[target=kernel_buffer_store](args = (%output, (%thread_program_id, 0), %truediv))
    return None
```

Can be interpreted to check for correctness

# The Turbine Kernel Language: Traced

```
stream.executable private @tk_kernel_softmax__1 {
  stream.executable.export public @tk_kernel_softmax__1 workgroups() -> (index, index, index) {
    %c64 = arith.constant 64 : index
    %c1 = arith.constant 1 : index
    stream.return %c64, %c1, %c1 : index, index, index
  }
  module {
    func.func @tk_kernel_softmax__1(%arg0: !stream.binding, %arg1: !stream.binding) {
      %cst = arith.constant 0.000000e+00 : f32
      %c0 = arith.constant 0 : index
      %workgroup_id_0 = stream.dispatch.workgroup.id[0] : index
      %0 = stream.binding.subspan %arg0[%c0] : !stream.binding -> memref<64x64xf32>
      %1 = vector.transfer_read %0[%workgroup_id_0, %c0], %cst : vector<1x64xf32>
      %2 = vector.multi_reduction <maxnumf>, %1, %cst [0, 1] : vector<1x64xf32> to f32
      %3 = vector.broadcast %2 : f32 to vector<1x64xf32>
      %4 = arith.subf %1, %3 : vector<1x64xf32>
      %5 = math.exp2 %4 : vector<1x64xf32>
      %6 = vector.multi_reduction <add>, %5, %cst [0, 1] : vector<1x64xf32> to f32
      %7 = vector.broadcast %6 : f32 to vector<1x64xf32>
      %8 = arith.divf %5, %7 : vector<1x64xf32>
      %9 = stream.binding.subspan %arg1[%c0] : !stream.binding -> memref<64x64xf32>
      vector.transfer_write %8, %9[%workgroup_id_0, %c0] : memref<64x64xf32>
      return
    }
  }
}
```

# The Turbine Kernel Language: Control Flow

```python
# N, M, K, BLOCK_M, BLOCK_N, BLOCK_K --> tk.sym

@tk.gen.kernel(M // BLOCK_M, N // BLOCK_N)
def gemm(
    A: tkl.InputBuffer[M, K, tkl.f16],
    B: tkl.InputBuffer[N, K, tkl.f16],
    output: tkl.OutputBuffer[M, N, tkl.f16],
):
    grid_n = tkl.program_id(0)
    grid_m = tkl.program_id(1)

    acc = tkl.constant((BLOCK_M, BLOCK_N), tkl.f32, 0.0)

    @tkl.for_loop(0, K // BLOCK_K, init_args=[acc])
    def body(i, c):
        a = tkl.load(A, (grid_n, i * BLOCK_M), (BLOCK_M, BLOCK_K))
        b = tkl.load(B, (i * BLOCK_N, grid_m), (BLOCK_N, BLOCK_K))
        b = tkl.transpose(b, (1, 0))
        return (tkl.dot(a, b, c),)

    tkl.store(output, (grid_n, grid_m), body[0])
```

FP Inspired Control Flow

# The Turbine Kernel Language: Dependently Typed

```
import shark_turbine.kernel as tk
import shark_turbine.kernel.lang as tkl


M = tkl.sym.M
N = tkl.sym.K

@tk.gen.kernel(M)
def softmax(
    input: tkl.InputBuffer[M, N, tkl.f32],
    output: tkl.OutputBuffer[M, tkl.f32]
):
    row_index = tkl.program_id(0)
    row = tkl.load(input, (row_index, 0), (1, N))
    row_minus_max = row - tkl.max(row)
    numerator = tkl.exp2(row_minus_max)
    denominator = tkl.sum(numerator)
    softmax_output = numerator / denominator
    tkl.store(output, (row_index), softmax_output)
```

Symbolic Indexing with sympy

Dependent Types

# The Turbine Kernel Language: Symbolic Grid

```python
# N, M, K, BLOCK_M, BLOCK_N, BLOCK_K --> tk.sym

@tk.gen.kernel(M // BLOCK_M, N // BLOCK_N)
def gemm(
    A: tkl.InputBuffer[M, K, tkl.f16],
    B: tkl.InputBuffer[N, K, tkl.f16],
    output: tkl.OutputBuffer[M, N, tkl.f16],
):
    grid_n = tkl.program_id(0)
    grid_m = tkl.program_id(1)

    acc = tkl.constant((BLOCK_M, BLOCK_N), tkl.f32, 0.0)

    @tkl.for_loop(0, K // BLOCK_K, init_args=[acc])
    def body(i, c):
        a = tkl.load(A, (grid_n, i * BLOCK_M), (BLOCK_M, BLOCK_K))
        b = tkl.load(B, (i * BLOCK_N, grid_m), (BLOCK_N, BLOCK_K))
        b = tkl.transpose(b, (1, 0))
        return (tkl.dot(a, b, c),)

    tkl.store(output, (grid_n, grid_m), body[0])
```

Symbolically Shaped Grid

# The Turbine Kernel Language: Symbolic Grid

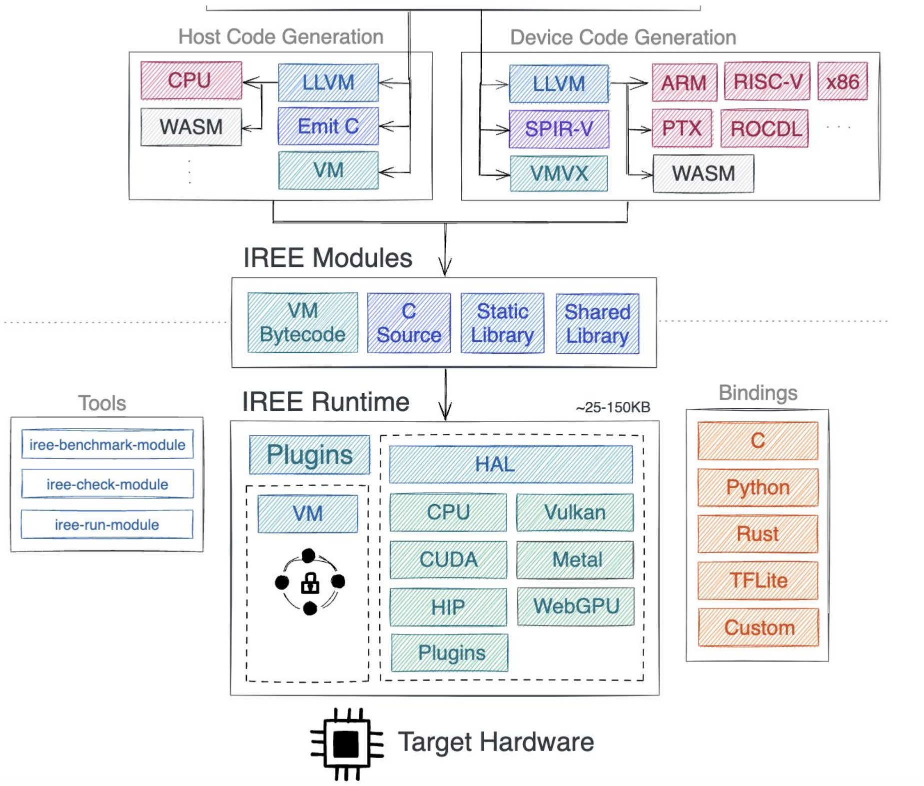
```
stream.executable private @tk_kernel_matmul__1 {

  stream.executable.export public @tk_kernel_matmul__1 workgroups()
                                               -> (index, index, index) {
    %c4 = arith.constant 4 : index
    %c4_0 = arith.constant 4 : index
    %c1 = arith.constant 1 : index
    stream.return %c4, %c4_0, %c1 : index, index, index
  }

  builtin.module {
    func.func @tk_kernel_matmul__1(%arg0: !stream.binding, %arg1: !stream.binding, %arg2:
!stream.binding) {
      %cst = arith.constant 0.000000e+00 : f16
      …
  }
```

# Integration with SHARK-Turbine

```python
import torch
from shark_turbine.aot import export
import shark_turbine.kernel as tk
import shark_turbine.kernel.lang as tkl


M = tkl.sym.M
N = tkl.sym.K

@tk.gen.kernel(M)
def softmax(
    input: tkl.InputBuffer[M, N, tkl.f32], output: tkl.OutputBuffer[M, N, tkl.f32]
):
    row_index = tkl.program_id(0)
    row = tkl.load(input, (row_index, 0), (1, N))
    row_minus_max = row - tkl.max(row)
    numerator = tkl.exp2(row_minus_max)
    denominator = tkl.sum(numerator)
    softmax_output = numerator / denominator
    tkl.store(output, (row_index, 0), softmax_output)


class NN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = torch.nn.Linear(64, 64, dtype=torch.float32)

    def forward(self, x):
        x = self.linear(x)
        x = softmax(x)
        return x
```

```python
model = NN()
a = torch.ones(64, 64, dtype=torch.float32)
exported = export(model, a)

# See torch IR
exported.print_readable()

# See internal linalg and async IR.
exported.import_to("iree_internal")
exported.print_readable()

# Compile and Run
compiled = exported.compile(save_to="softmax_aot.vmfb", target_backends="rocm")

# Eager execution
eager_results = model.forward(a)
print(eager_results)
```

# Compiler Backends available to target




Also, plugin support!

Plugin Example:

https://github.com/nod-ai/iree-amd-aie

# Shoutouts

Triton: https://github.com/openai/triton

Pallas: https://jax.readthedocs.io/en/latest/pallas/

# Where to find this work?

Try it out: `pip install shark-turbine`

SHARK-Turbine: https://github.com/nod-ai/SHARK-Turbine

IREE: https://github.com/openxla/iree

Examples: https://github.com/Groverkss/fa2-tk

Get to know more about TK on IREE Discord (Cat, DSL, or both):

https://discord.com/invite/26P4xW4

All our development is open source!

# Thank you

Key takeaways:

- A DSL to target existing IREE compiler phases
- Supported with torch.compile, torch.export and eager execution
- Enters at graph level, allowing graph optimizations

```python
import shark_turbine.kernel as tk

@tk.gen.kernel(...)
def softmax(...):
  ...


class NN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = torch.nn.Linear(64, 64)

    def forward(self, x):
        x = self.linear(x)
        x = softmax(x)
        return x
```

https://github.com/nod-ai/SHARK-Turbine

https://discord.com/invite/26P4xW4