Wave programming language

Ivan Butygin

AMD
together we advance_

Motivational example: tiled and optimized GEMM

```
constraints = [tkw.WorkgroupConstraint(M, BLOCK_M, 0)]
constraints += [tkw.WorkgroupConstraint(N, BLOCK_N, 1)]
constraints += [tkw.TilingConstraint(K, BLOCK_K)]
constraints += [tkw.WaveConstraint(M, BLOCK_M / 2)]
constraints += [tkw.WaveConstraint(N, BLOCK_N / 2)]

constraints += [tkw.HardwareConstraint(threads_per_wave=64, mma_type=F32_16x16x16_F16)]

if dynamic_dims:
    constraints += [tkw.Assumption(K > BLOCK_K * 4)]

@tkw.wave(constraints)
def gemm(
    a: tkl.Memory[M, K, ADDRESS_SPACE, tkl.f16],
    b: tkl.Memory[N, K, ADDRESS_SPACE, tkl.f16],
    c: tkl.Memory[M, N, GLOBAL_ADDRESS_SPACE, tkl.f32],
):
    c_reg = tkl.Register[M, N, tkl.f32](0.0)
    @tkw.reduction(K, init_args=[c_reg])
    def repeat(acc: tkl.Register[M, N, tkl.f32]) -> tkl.Register[M, N, tkl.f32]:
        a_reg = tkw.read(a)
        b_reg = tkw.read(b)
        acc = tkw.mma(a_reg, b_reg, acc)
        return acc

    tkw.write(repeat, c)
```

What is Wave

- Symbolic domain specific language for high performance machine learning.
- Targeting GPU (AMDGPU only currently)
- Python syntax + sympy symbolic expressions
- Explicit separating between kernel logic and distribution strategy
- MLIR for codegen, mostly upstream dialects
- IREE as last-mile optimizer and runtime

Motivation: why new language

- HW matmul intrinsics are required for the competitive perf on modern GPUs
  - All GPU vendors have them in some form
  - Direct CUDA/HIP programming them is too time consuming and error prone
    - They are usually a collective operation across many threads, doesn't fit nicely to SIMT model
    - Matrix elements may need a nontrivial layout in registers/shared memory
    - Tiling/scheduling intertwined with the kernel logic
  - Need a more convenient way to experiment with various intrinsics/tile sizes/distribution
- Wave
  - High level kernel description operating on whole tensors level
  - Tiling and distribution strategy is spelled explicitly and separated from the kernel description
  - Tiling operating on block/wave level
  - Symbolic data types for tensor shapes and distribution patterns
  - Memory access pattern per block/thread is decided by the compiler transparently
  - Automatic masking for unaligned shapes
  - Easy way to test new data access patterns

Basic syntax: elementwise copy kernel

- Constraints to describe how to tile/distribute computation across GPU WGs/Threads
  - Splits work shape into workgroups and than each WG into waves

```
constraints = [
    tkw.HardwareConstraint(threads_per_wave=64)
]
constraints += [tkw.WorkgroupConstraint(M, BLOCK_M, 1)]
constraints += [tkw.WorkgroupConstraint(N, BLOCK_N, 0)]
constraints += [tkw.WaveConstraint(M, BLOCK_M)]
constraints += [tkw.WaveConstraint(N, BLOCK_N)]
```

- All tensor shapes are symbolic
  - Tensor shapes and constraints are connected symbolically
- Kernel logically operating on the entire tensors

```
@tkw.wave(constraints)
def test(
    a: tkl.Memory[M, N, ADDRESS_SPACE, tkl.f16],
    b: tkl.Memory[M, N, ADDRESS_SPACE, tkl.f16],
):
    res = tkw.read(a)
    tkw.write(res, b)
```

- Setting symbol values
  - Global work size not need to be divisible on tile sizes, masking ops are inserted automatically for unaligned shapes

```
options = WaveCompileOptions(
    subs={
        M: shape[0],
        N: shape[1],
        ADDRESS_SPACE: GLOBAL_MEMORY,
    },
)

test = wave_compile(options, test)
test(a, b)
```

GEMM

```
constraints = [tkw.WorkgroupConstraint(M, BLOCK_M, 0)]
constraints += [tkw.WorkgroupConstraint(N, BLOCK_N, 1)]
constraints += [tkw.TilingConstraint(K, BLOCK_K)]
constraints += [tkw.WaveConstraint(M, BLOCK_M / 2)]
constraints += [tkw.WaveConstraint(N, BLOCK_N / 2)]

constraints += [tkw.HardwareConstraint(threads_per_wave=64, mma_type=F32_16x16x16_F16)]

if dynamic_dims:
    constraints += [tkw.Assumption(K > BLOCK_K * 4)]

@tkw.wave(constraints)
def gemm(
    a: tkl.Memory[M, K, ADDRESS_SPACE, tkl.f16],
    b: tkl.Memory[N, K, ADDRESS_SPACE, tkl.f16],
    c: tkl.Memory[M, N, GLOBAL_ADDRESS_SPACE, tkl.f32],
):
    c_reg = tkl.Register[M, N, tkl.f32](0.0)
    @tkw.reduction(K, init_args=[c_reg])
    def repeat(acc: tkl.Register[M, N, tkl.f32]) -> tkl.Register[M, N, tkl.f32]:
        a_reg = tkw.read(a)
        b_reg = tkw.read(b)
        acc = tkw.mma(a_reg, b_reg, acc)
        return acc

    tkw.write(repeat, c)
```

GEMM

- Wave size and matmul intrinsic to use (alternatively, can be set per individual mma op)

```
constraints += [tkw.HardwareConstraint(threads_per_wave=64, mma_type=F32_16x16x16_F16)]
```

- Assumptions for dynamically sized dimensions, used later in compilation pipeline

```
if dynamic_dims:
    constraints += [tkw.Assumption(K > BLOCK_K * 4)]
```

- ADDRESS_SPACE controls if input array should be promoted to shared mem

```
    a: tkl.Memory[M, K, ADDRESS_SPACE, tkl.f16],
    b: tkl.Memory[N, K, ADDRESS_SPACE, tkl.f16],
```

- Allocate temp storage for accumulator

```
    c_reg = tkl.Register[M, N, tkl.f32](0.0)
```

- Reduction loop across K dimension as this dimension is tiled

```
    @tkw.reduction(K, init_args=[c_reg])
    def repeat(acc: tkl.Register[M, N, tkl.f32]) -> tkl.Register[M, N, tkl.f32]:
        a_reg = tkw.read(a)
        b_reg = tkw.read(b)
```

- mma is mapped to the hw matmul instructions

```
        acc = tkw.mma(a_reg, b_reg, acc)
        return acc

    tkw.write(repeat, c)
```

conv2d/igemm

```
x_mapping = tkw.IndexMapping(
    num_iterators=2,
    inputs={
        N: i // SZ_OUT,
        C: j % C,
        H: (i % SZ_OUT) % W_OUT * stride + (j // C) % WF,
        W: (i % SZ_OUT) // W_OUT * stride + (j // C) // WF,
    },
    outputs={M: i, K: j},
)
w_mapping = ...
out_mapping = ...

@tkw.wave(constraints)
def conv(
    x: tkl.Memory[N, H, W, C, ADDRESS_SPACE, tkl.f16],
    we: tkl.Memory[HF, WF, C, NF, ADDRESS_SPACE, tkl.f16],
    out: tkl.Memory[N, H_OUT, W_OUT, NF, GLOBAL_ADDRESS_SPACE, tkl.f32],
):
    c_reg = tkl.Register[M, NF, tkl.f32](0.0)
    @tkw.reduction(K, init_args=[c_reg])
    def repeat(acc: tkl.Register[M, NF, tkl.f32]) -> tkl.Register[M, NF, tkl.f32]:
        a_reg = tkw.read(x, mapping=x_mapping)
        b_reg = tkw.read(we, mapping=w_mapping)
        acc = tkw.mma(a_reg, b_reg, acc)
        return acc

    tkw.write(repeat, out, mapping=out_mapping)
```

AMD
together we advance_

conv2d/igemm

- GEMM convolution: im2col -> GEMM -> col2im
- Implicit as it doesn't require allocation temp tensors
- Same kernel as GEMM
- In/out tensors are 4d
- Reads/writes has the custom mapping

```
@tkw.wave(constraints)
def conv(
    x: tkl.Memory[N, H, W, C, ADDRESS_SPACE, tkl.f16],
    we: tkl.Memory[HF, WF, C, NF, ADDRESS_SPACE, tkl.f16],
    out: tkl.Memory[N, H_OUT, W_OUT, NF, GLOBAL_ADDRESS_SPACE, tkl.f32],
):
    c_reg = tkl.Register[M, NF, tkl.f32](0.0)

    @tkw.reduction(K, init_args=[c_reg])
    def repeat(acc: tkl.Register[M, NF, tkl.f32]) -> tkl.Register[M, NF, tkl.f32]:
        a_reg = tkw.read(x, mapping=x_mapping)
        b_reg = tkw.read(we, mapping=w_mapping)
        acc = tkw.mma(a_reg, b_reg, acc)
        return acc

    tkw.write(repeat, out, mapping=out_mapping)
```

conv2d/igemm

- Custom mapping for read/write ops
  - Maps 4d input into 2d tiles mma expects
  - Iterators describe iteration shape and symbolic exprs maping to the input/output tensors elements
  - Semantics similar to MLIR linalg.generic

```
i = tkw.IndexMapping.iterator(0)
j = tkw.IndexMapping.iterator(1)

x_mapping = tkw.IndexMapping(
    num_iterators=2,
    inputs={
        N: i // SZ_OUT,
        C: j % C,
        H: (i % SZ_OUT) % W_OUT * stride + (j // C) % WF,
        W: (i % SZ_OUT) // W_OUT * stride + (j // C) // WF,
    },
    outputs={M: i, K: j},
)
w_mapping = tkw.IndexMapping(
    num_iterators=2,
    inputs={NF: i % NF, C: j % C, HF: (j // C) % WF, WF: (j // C) // WF},
    outputs={NF: i, K: j},
)
```

AMD
together we advance_

Attention

- 2 mmas inside reduction loop with different intrinsics
- Reductions across WG
- Direct iteration index access
- And more…

```
@tkw.reduction(K2, init_args=[init_max, init_sum, c_reg])
def repeat(
    partial_max: tkl.Register[B, M, tkl.f32],
    partial_sum: tkl.Register[B, M, tkl.f32],
    acc: tkl.Register[B, N, M, tkl.f32],
):
    imm_reg = tkl.Register[B, K2, M, tkl.f32](0.0)
    q_reg = tkw.read(q)
    k_reg = tkw.read(k)
    inner_acc = tkw.mma(k_reg, q_reg, imm_reg, mfma_variant[0])
    x_j = tkw.permute(inner_acc, target_shape=[B, M, K2])
    k2_index = tkw.self_index(K2, tkl.i64)
    mask = tkw.apply_expr(k2_index, lambda x: x < K2)
    mask = tkw.broadcast(mask, target_shape=[M, K2])
    mask = tkw.cast(mask, tkw.i1)
    bias = tkw.select(mask, ZEROF, MIN_INF)
    x_j = x_j + bias
    m_j = tkw.max(x_j, partial_max, dim=K2)
    e_delta_max = tkw.exp2(partial_max - m_j)
    e_delta = tkw.exp2(x_j - m_j)
    e_init = partial_sum * e_delta_max
    d_j = tkw.sum(e_delta, e_init, dim=K2)
    imm_f16 = tkw.cast(e_delta, tkl.f16)
    v_reg = tkw.read(v, mapping=v_mapping)
    new_acc = acc * e_delta_max
    acc = tkw.mma(v_reg, imm_f16, new_acc)
    return m_j, d_j, acc
```

Architecture

- Frontend
  - Torch.fx tracing
- Middle end
  - We are using torch.fx + sympy as IR
  - Index sequence analysis
  - Tiling/Expansion
  - Some optimizations
- Backend
  - MLIR vector, arith, scf, amdgpu dialects
  - IREE as last-mile optimizer
  - IREE as runtime
    - Standalone runtime WIP

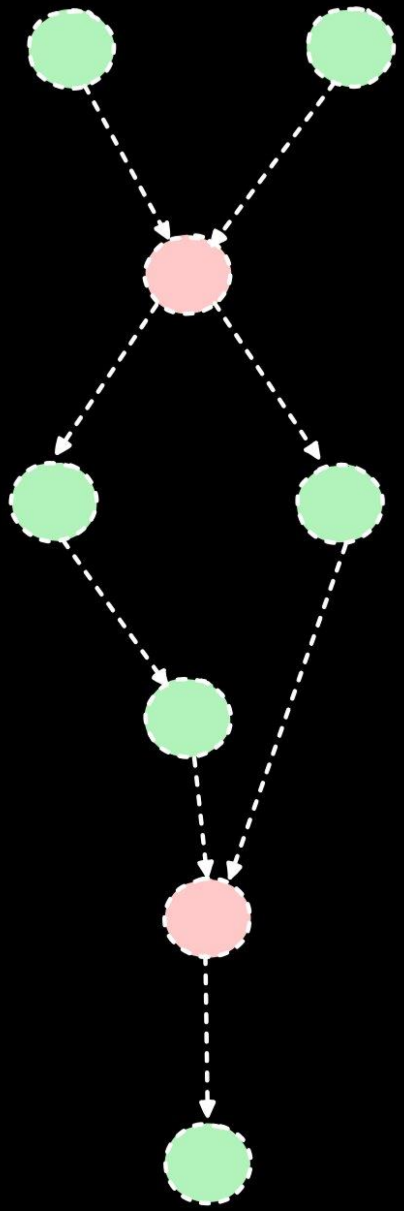
Frontend - tracing

- **Tracing**
  - Uses torch.fx to trace the kernel
  - Torch.fx calls the kernel function and traces the kernel using special Proxy objects that act as placeholders for actual values
  - Avoids the need to implement custom parser and leverages the flexibility of Python
  - Need special handling for control flow ops

- **Intermediate Representation**
  - Builds on torch.fx intermediate representation (python-based IR)
  - SSA
  - Adds wave-specific operators and types
  - Adds symbolic types
  - Leverages torch.fx infrastructure for graph rewrites

Middle end - transformations

- **Index Sequence Analysis & Thread Shape Analysis**
  - Given a computation graph G with nodes N, and a specific set of nodes V with requirements on memory access patterns (like MMA), determine the memory access patterns for every node in the graph
  - Nodes providing requirements can be thought of as “sources” and the rest as “sinks”
  - Need to propagate information from the “sources” to the “sinks”
  - Also need to handle “conflicts” when thread shapes don’t agree
    - In the best case, this requires just a broadcast
    - In the worst case, shuffles or trips to shared memory



AMD
together we advance_

Middle end - transformations

- **Expansion**
  - While the kernel distribution is authored from the perspective of a single wave, the compiler needs to generate code for a single thread
  - To do that, we need to expand the kernel according to the input sizes and the constraints specified by the kernel author

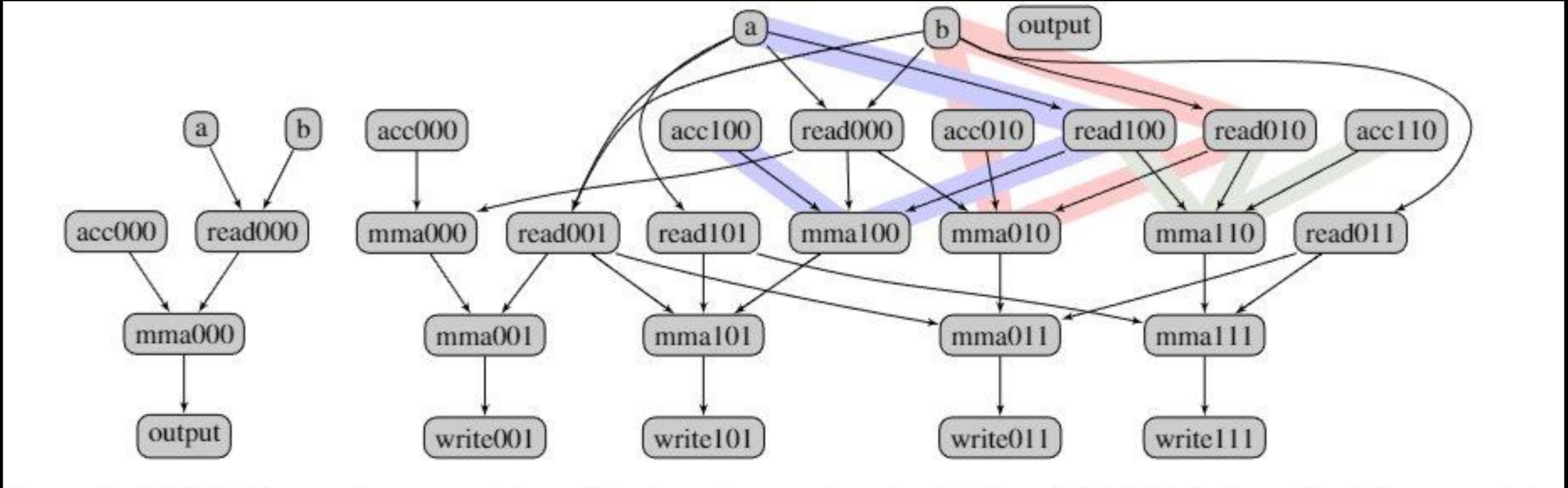


Figure 2: a) (left) The graph representation of the traced gemm kernel of Listing 1. b) (right) shows the fully expanded gemm kernel. We highlight the added nodes in subsequent expansion steps in different colors: 1. Expansion in (0, 1, 0) red, 2. Expansion in (1, 0, 0) blue, 3. Expansion in (1, 1, 0)? green.

Middle end - optimizations

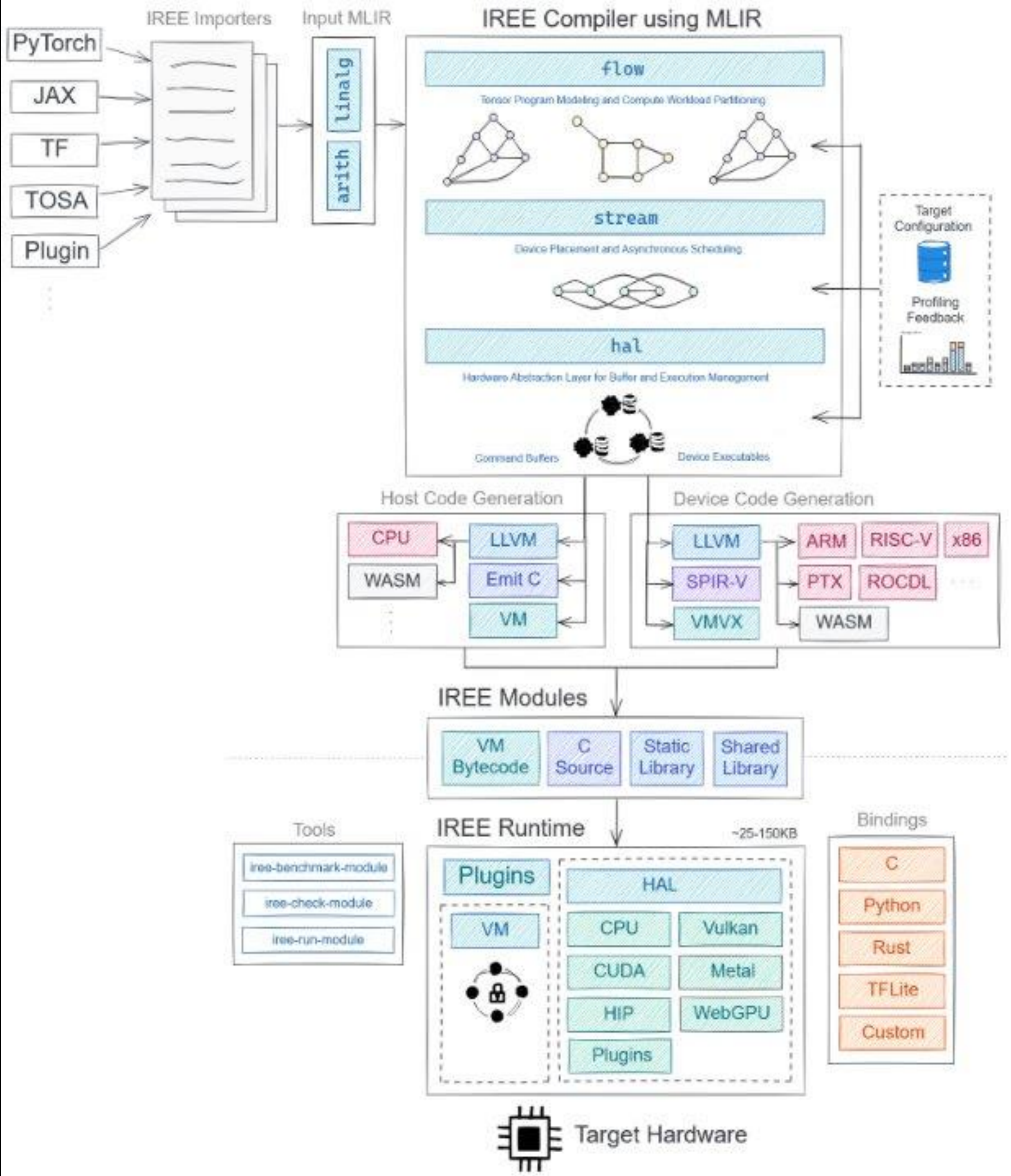
- Promotion to Shared Memory
- Global Load Optimization
- FP8 Virtual GEMM Optimizations
- Partitioning Strided Operators
- Barrier Insertion
- Instruction Scheduling
- Hoisting loop invariant Wave ops
- Contiguous Load Detection
- …

Backend - lowering

- Lower symbolic access patterns to affine.apply + sequence of MLIR operations
  - Prior to lowering, we simplify the symbolic expressions using sympy to reduce the number of instructions emitted / type of instructions emitted
  - Most operators are then lower to the vector dialect
    - Read -> vector.read / vector.gather / vector.maskedload
    - Write -> vector.store / vector.scatter / vector.maskedstore
    - MMA -> amdgpu.mfma
    - Reduce -> gpu.shuffle
    - Special Intrinsics (Instruction Scheduling Barriers) -> llvm.intrinsics
  - Eventually lowered to LLVM IR for device

Backend - runtime

- **IREE Runtime**
  - Leverages IREE Runtime to launch kernels
  - Encapsulates kernels within the stream/flow dialect for easy integration into models compiled by IREE
  - Can leverage existing tools like iree-run-module and iree-benchmark-module to run and benchmark kernels
  - Integrates efficiently with Pytorch
    - Zero-copy of tensors to-and-from Pytorch (still some issues being worked on)
    - Kernels can be called from Pytorch code and are cached to avoid recompilation on every call



Backend optimizations

- General vector/arith dialect canonicalizations
- MLIR CSE
- Integer range analysis (IREE)
  - Reduce int<->index casts count
  - Remove selects after affine.apply/ceildiv lowering
  - Divisibility (simplify divs/mods when work size is divisible by tile size)

Some takeaways

- Python as primary language
  - We are using upstream MLIR python bindings
  - Slow kernel compile times (partially offset by caching)
  - Not all MLIR things are exposed to Python
- torch.fx is minimal IR
  - Quickly enabling e2e flow
  - Had to implement our own common utilities (CSE, DCE) on top of it
- No good upstream symbolic type dialect
  - Using affine exprs + affine.apply for represent index calculations
  - Have converter from subset of sympy to affine expr + arith
  - Would be nice to have one upstrem
- IREE runtime is good at quickly enabling e2e flow
  - With just few lines of IR and few API calls we have a runnable e2e kernel and pytorch arrays integration
  - Would be nice to have this in MLIR upstream too

Fin

- Project repo: https://github.com/iree-org/iree-turbine/tree/main/iree/turbine/kernel/wave
- Discussions: IREE discord Discord
- We are hiring!

AMD