Whole-Model Tuner

For the IREE ML Compiler

Jakub Kuderski*, Bangtian Liu, Amily Wu, Max Dawkins

*jakub.kuderski@amd.com

2025-11-10, CDP 2025, Toronto, Canada

Agenda

- Background
  - Traditional compilers and ML compilers
  - The IREE / MLIR software stack
  - Code Generation in IREE
- SHARK Tuner Architecture
  - Basic idea
  - Whole-model tuning
  - Optimizing the tuning loop
- Results and Future Work

Background: ML Models

Rapid advancements in AI model architecture

- Very high-level source language(s)
  - Typically Python-based Domain Specific Languages (PyTorch, JAX, TensorFlow)

```
xq = xq.transpose(1, 2)  # (bs, n_local_heads, seqlen, head_dim)
keys = keys.transpose(1, 2)
values = values.transpose(1, 2)
scores = torch.matmul(xq, keys.transpose(2, 3)) / math.sqrt(self.head_dim)
if mask is not None:
        scores = scores + mask  # (bs, n_local_heads, seqlen, cache_len + seqlen)
scores = F.softmax(scores.float(), dim=-1).type_as(xq)
```

Background: Diverse Target Hardware

Rapid advancements in AI hardware

- CPU – lowest barrier of entry, high programmability
  - Low throughput
- GPU – good programmability, great performance
  - Broad range of implementations: from mobile chips to fancy data center cards
- Custom Accelerators
  - NPUs, TPUs, AIE, etc.

Background: Programming Models

Traditional C-like languages

- Code is written from a perspective of a single thread
  - Vendor extension expose SIMD instructions
- Support for branching, indirection (pointers), dynamic memory allocation

```
__m512i acc[16];
for (int i = 0; i < M0; ++i) {
  acc[i] = _mm512_loadu_si512((__m512i*)(out_ptr + i * 16));
}

for (int k = 0; k < K; ++k) {
  __m512i rhs = _mm512_cvtepi8_epi16(_mm256_loadu_si256((const __m256i*)rhs_ptr));
  rhs_ptr += 32;
  for (int i = 0; i < M0; ++i) {
    acc[i] = _mm512_dpwssd_epi32(acc[i], rhs, _mm512_cvtepi8_epi16(_mm256_set1_epi16(*(const int16_t*)(lhs_ptr))));
    lhs_ptr += 2;
  }
}
```

Background: The SIMT Programming Model

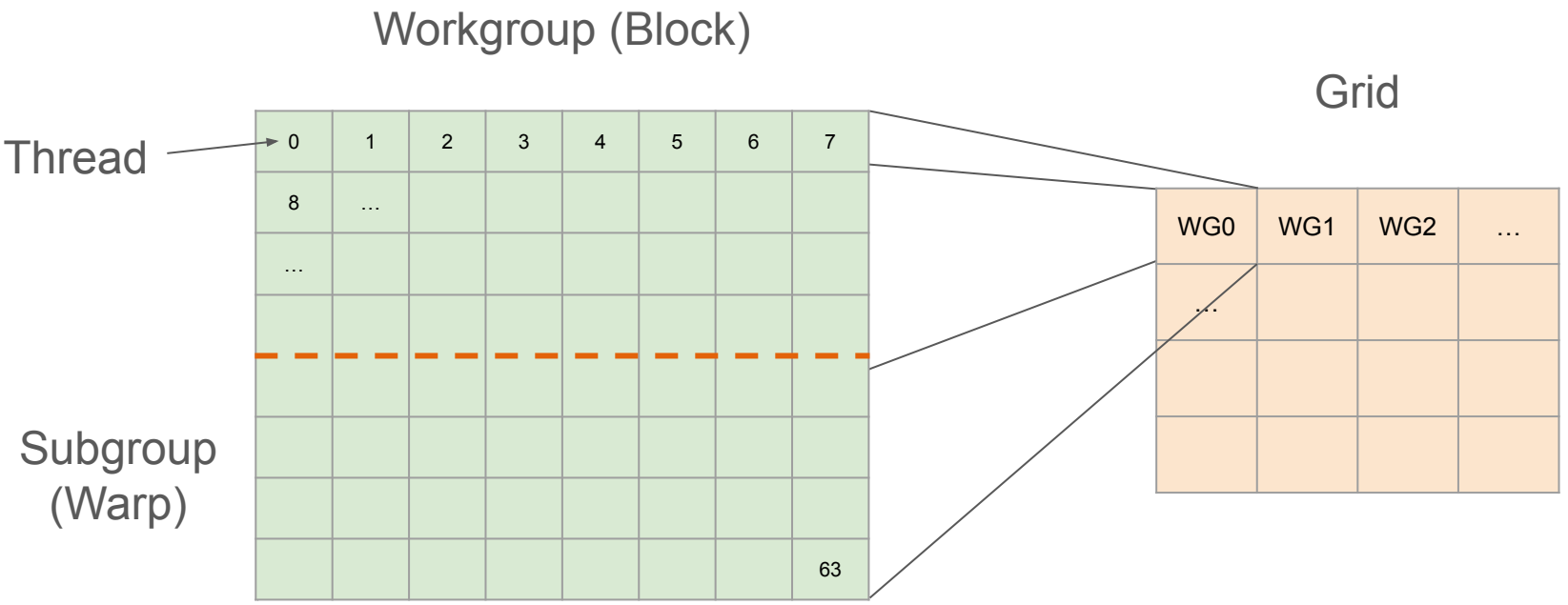
Single Instruction Multiple Threads

CUDA / HIP/ OpenCL / GLSL / HLSL / etc.

- Code is written from a perspective of a single thread
- Limited indirection and branching
- Threading managed by the hardware

```
void main() {
  const uvec2 wgID = gl_WorkGroupID.xy;
  const uvec2 localID = gl_LocalInvocationID.xy;
  const uint threadID = gl_SubgroupInvocationID;
  const uint startRow = wgID.x * WG_ROWS;

  for (uint y = 0; y < N0; ++y) {
    uint r = startRow + y * WG_Y + localID.y;
    int32_t laneResult = 0;
    i32vec4 tileA[K0_VEC];
    i32vec4 tileB[K0_VEC];
    ...
  }
```

Background: The SIMT Programming Model



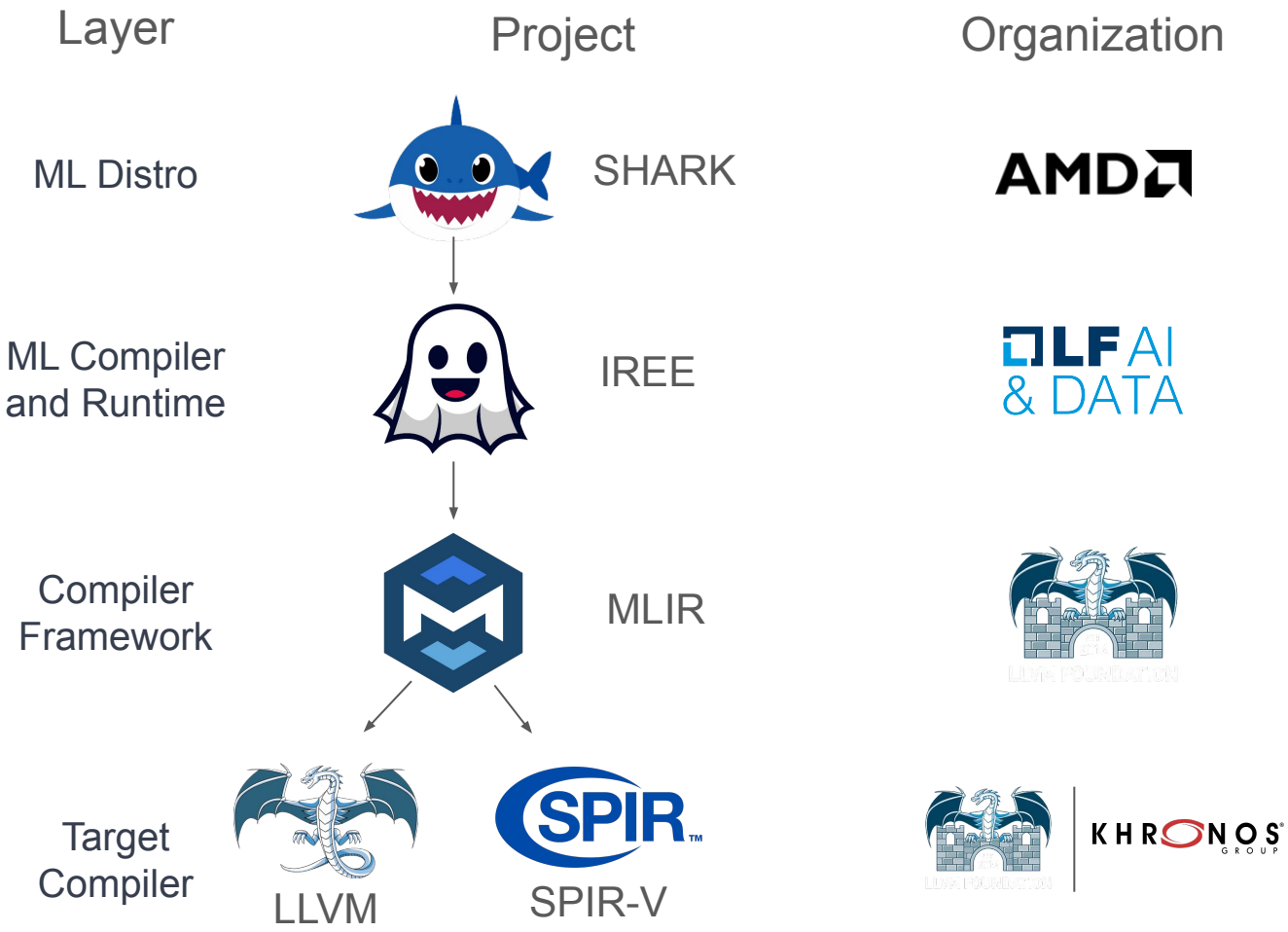
IREE

Intermediate Representation Execution Environment

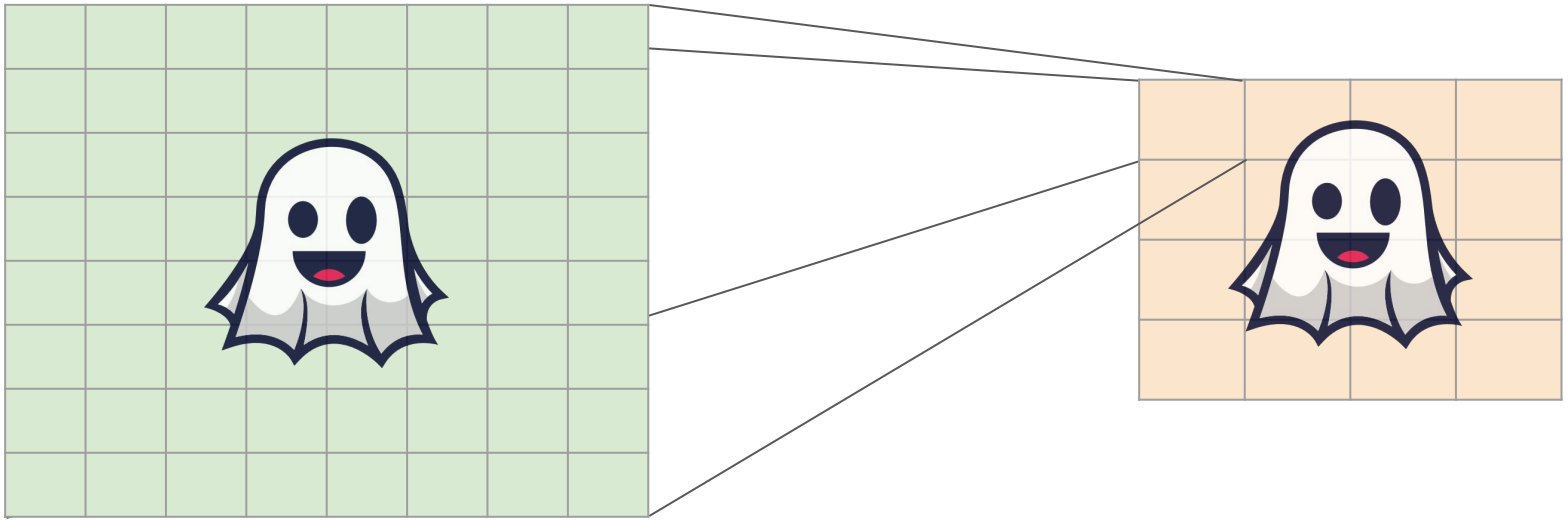
Pronounced ‘eerie’

- Build a suite of extensible, composable, re-targetable open source tools for efficiently deploying popular and emerging ML models
- Unlock access to the hardware ecosystem from any framework
- Increase the speed of ML innovation for all ML practitioners
- Project under the Linux Foundation (AI & Data Science)

Software Stack



The IREE Programming Model



The IREE Programming Model

IREE responsible for deciding the compilation strategy:

- Dispatch (kernel) formation
- Scheduling of work (command buffers)
- Distribution to workgroups, subgroups, and threads
  - Memory access pattern
- Shared memory management

Sample PyTorch Input Program

```
import torch
import torch.nn as nn

class ToyModel(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, arg0: torch.Tensor, arg1: torch.Tensor, arg2: torch.Tensor) -> torch.Tensor:
        mat_b = torch.transpose(arg1, 0, 1)
        mm = torch.matmul(arg0, mat_b)
        matvec = torch.mv(mm.to(torch.float32), arg2)
        res = nn.functional.relu(matvec)
        return res

model = ToyModel()
example_inputs = (torch.randn(1024, 512, dtype=torch.float16),
                  torch.randn(2048, 512, dtype=torch.float16),
                  torch.randn(2048, dtype=torch.float32))
```

Exported MLIR

```
import iree.turbine.aot as aot
exported = aot.export(model, *example_inputs)
```

```
func.func @main(%arg0: !torch.vtensor<[1024,512],f16>, %arg1: !torch.vtensor<[2048,512],f16>,
                %arg2: !torch.vtensor<[2048],f32>) -> !torch.vtensor<[1024],f32> {
  %int0 = torch.constant.int 0
  %int1 = torch.constant.int 1
  %0 = torch.aten.transpose.int %arg1, %int0, %int1 : !torch.vtensor<[2048,512],f16>, !torch.int, !torch.int -> !torch.vtensor<[512,2048],f16>
  %1 = torch.aten.mm %arg0, %0 : !torch.vtensor<[1024,512],f16>, !torch.vtensor<[512,2048],f16> -> !torch.vtensor<[1024,2048],f16>
  %int6 = torch.constant.int 6
  %2 = torch.prims.convert_element_type %1, %int6 : !torch.vtensor<[1024,2048],f16>, !torch.int -> !torch.vtensor<[1024,2048],f32>
  %3 = torch.aten.mv %2, %arg2 : !torch.vtensor<[1024,2048],f32>, !torch.vtensor<[2048],f32> -> !torch.vtensor<[1024],f32>
  %4 = torch.aten.relu %3 : !torch.vtensor<[1024],f32> -> !torch.vtensor<[1024],f32>
  return %4 : !torch.vtensor<[1024],f32>
}
```

Global Optimization and Dispatch Formation

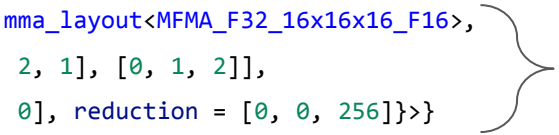
```
%3 = flow.dispatch.workgroups(%0, %1) : (tensor<1024x512xf16>, tensor<2048x512xf16>) -> tensor<1024x2048xf16> =
  (%arg5: !flow.dispatch.tensor<readonly:tensor<1024x512xf16>>,
   %arg6: !flow.dispatch.tensor<readonly:tensor<2048x512xf16>>,
   %arg7: !flow.dispatch.tensor<writeonly:tensor<1024x2048xf16>>) {
  %cst = arith.constant 0.000000e+00 : f32
  %7 = flow.dispatch.tensor.load %arg5, offsets = [0, 0], sizes = [1024, 512], strides = [1, 1]
    : !flow.dispatch.tensor<readonly:tensor<1024x512xf16>> -> tensor<1024x512xf16>
  %8 = flow.dispatch.tensor.load %arg6, offsets = [0, 0], sizes = [2048, 512], strides = [1, 1]
    : !flow.dispatch.tensor<readonly:tensor<2048x512xf16>> -> tensor<2048x512xf16>
  %9 = tensor.empty() : tensor<1024x2048xf16>
  %10 = tensor.empty() : tensor<1024x2048xf32>
  %11 = linalg.fill ins(%cst : f32) outs(%10 : tensor<1024x2048xf32>) -> tensor<1024x2048xf32>
  %12 = linalg.matmul ins(%7, %8 : tensor<1024x512xf16>, tensor<2048x512xf16>) -> tensor<1024x2048xf32>
  %13 = linalg.truncf %12 : tensor<1024x2048xf32> -> tensor<1024x2048xf16>
  flow.dispatch.tensor.store %13, %arg7, offsets = [0, 0], sizes = [1024, 2048], strides = [1, 1]
    : tensor<1024x2048xf16> -> !flow.dispatch.tensor<writeonly:tensor<1024x2048xf16>>
  flow.return
}
```

The Problem

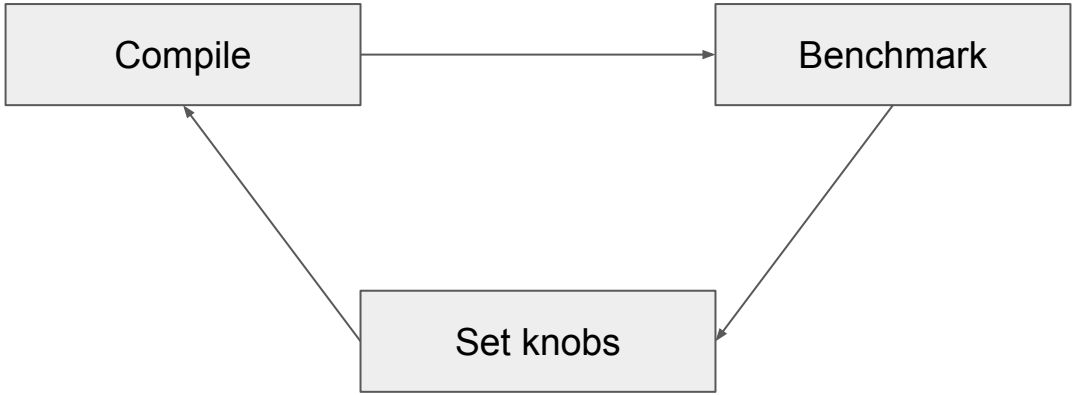
- Real world models dived into up to thousands of unique dispatches
  - Typically only a small number dominate the model running time
- The IREE compiler has to decide how to configure each dispatch
  - Multiple levels of tiling, shared memory, which intrinsics to use, etc.
- Heuristics are imperfect
  - Very hard to account for downstream effects like register allocation or energy consumption

Configured Root Operation

```
%mm = linalg.matmul indexing_maps = [affine_map<(d0, d1, d2) -> (d0, d2)>,
                                     affine_map<(d0, d1, d2) -> (d1, d2)>,
                                     affine_map<(d0, d1, d2) -> (d0, d1)>]
    {lowering_config = #iree_gpu.lowering_config<{mma_kind = #iree_gpu.mma_layout<MFMA_F32_16x16x16_F16>,
                                                   subgroup_basis = [[2, 2, 1], [0, 1, 2]],
                                                   workgroup = [64, 128, 0], reduction = [0, 0, 256]}>}
    ins(%3, %4 : tensor<2048x5120xf16>, tensor<1280x5120xf16>)
    outs(%6 : tensor<2048x120xf32>) -> tensor<2048x1280xf32>
```



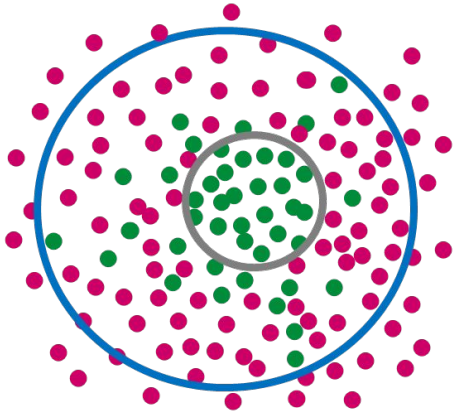
Autotuning Loop



SHARK Tuner Architecture

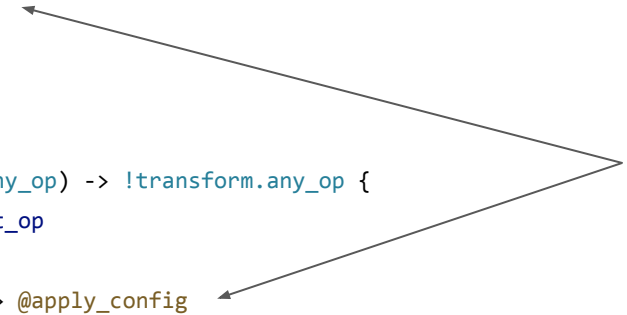
Tuning overrides compiler heuristics in search for the fastest configurations.

- Tuner implemented as a Python project outside of the compiler
- Uses the Z3 theorem prover to enumerate candidates
  - Over-approximate the search space of valid configurations
- Materialized as MLIR Transform Dialect scripts
  - Executed during compilation by an interpreter



Sample Tuning Spec File

```
transform.named_sequence @apply_config(%op: !transform.any_op,
                                       %config: !transform.any_param) {
 transform.annotate %op "compilation_info" = %config : !transform.any_op, !transform.any_param
 transform.yield
}

transform.named_sequence
@__kernel_config(%variant_op: !transform.any_op) -> !transform.any_op {
 %res = transform.foreach_match in %variant_op
   // Expected speedup: 1.22x.
   @match_mmt_2048x1280x5120_f16_f16_f32 -> @apply_config
   : (!transform.any_op) -> !transform.any_op
 transform.yield %res : !transform.any_op
}
```



Match the op and annotate with the returned attribute

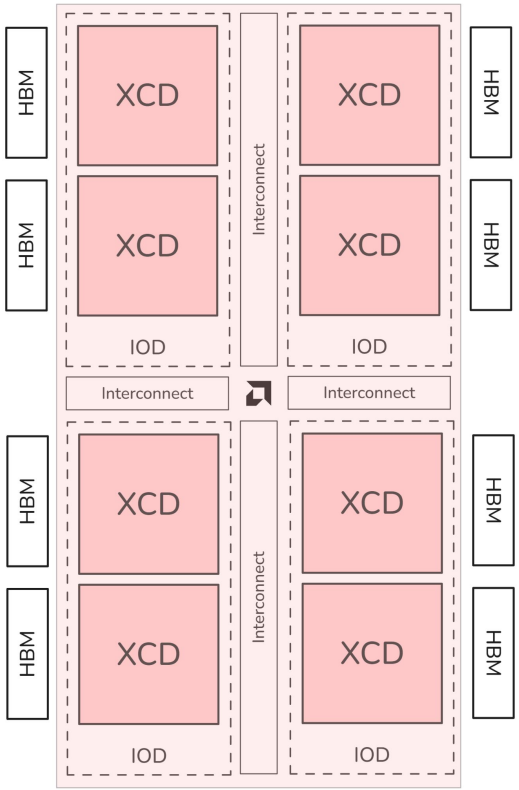
Sample Tuning Spec File

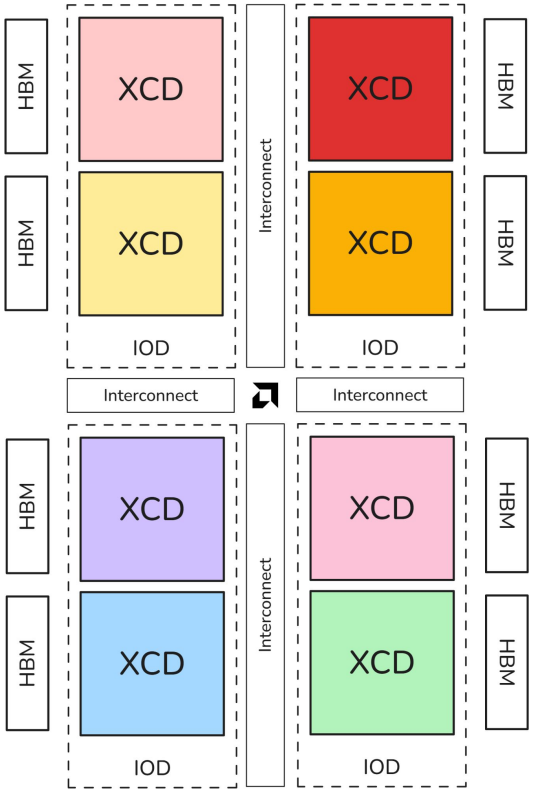
```
transform.named_sequence
@match_mmt_2048x1280x5120_f16_f16_f32(%matmul: !transform.any_op) -> (!transform.any_op, !transform.any_param) {
 %batch, %m, %n, %k = transform.iree.match.contraction %matmul,
   lhs_type = f16, rhs_type = f16, output_type = f32,
   indexing_maps = [affine_map<(d0, d1, d2) -> (d0, d2)>,
                    affine_map<(d0, d1, d2) -> (d1, d2)>,
                    affine_map<(d0, d1, d2) -> (d0, d1)>] : !transform.any_op -> !transform.param<i64>
 transform.iree.match.dims_equal %m, [2048] : !transform.param<i64>
 transform.iree.match.dims_equal %n, [1280] : !transform.param<i64>
 transform.iree.match.dims_equal %k, [5120] : !transform.param<i64>

 %config = transform.param.constant #iree_codegen.compilation_info<
   lowering_config = #iree_gpu.lowering_config<{promote_operands = [0, 1],
                                                 mma_kind = #iree_gpu.mma_layout<MFMA_F32_16x16x16_F16>,
                                                 subgroup_basis = [[2, 2, 1], [0, 1, 2]],
                                                 workgroup = [64, 128, 0], reduction = [0, 0, 64]}>,
   translation_info = #iree_codegen.translation_info<pipeline = LLVMGPUVectorDistribute
     workgroup_size = [256, 1, 1] subgroup_size = 64,
     {gpu_pipeline_options = #iree_gpu.pipeline_options<prefetch_shared_memory = true>}>
 > -> !transform.any_param
 transform.yield %matmul, %config : !transform.any_op, !transform.any_param
}
```

Key Insights

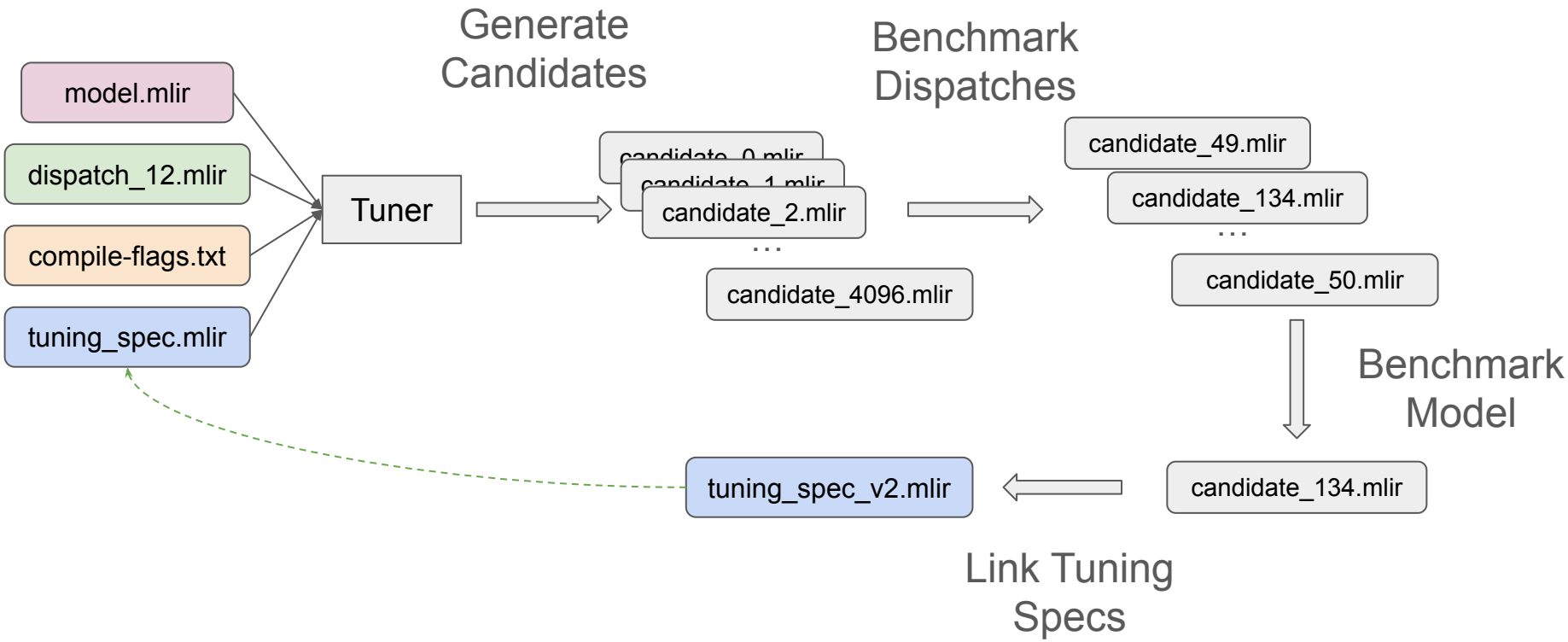
- Improving a single dispatch may sometimes regress the model
  - Observed with power-limited GPUs
- Chips within the same SKU have different baseline performance
  - Normalize results based on the physical device used for benchmarking
- Compiler design should account for the tuner
  - Keep the search space small
- Benchmarking time dominates tuning time

MI325X Compute Topologies





Whole-Model Tuning

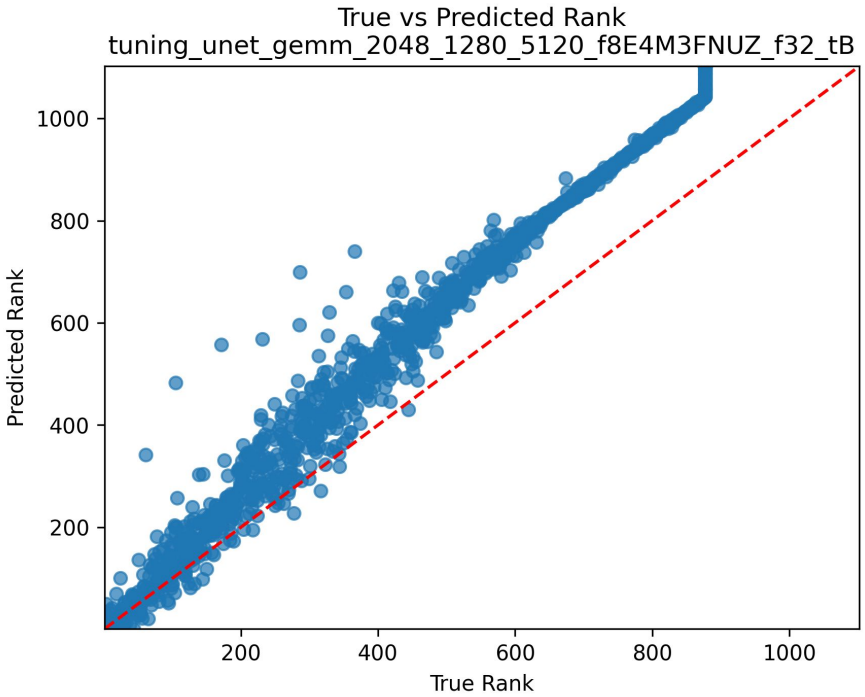


Deciding Candidate Evaluation Order

- Benchmarking thousands of candidates takes long
  - Prioritize those likely to perform well
  - Only benchmark the first N candidates

- Comparing possible ordering heuristics
  - Metric: how many iterations until the first nearly optimal candidate found
  - Baseline: random shuffle

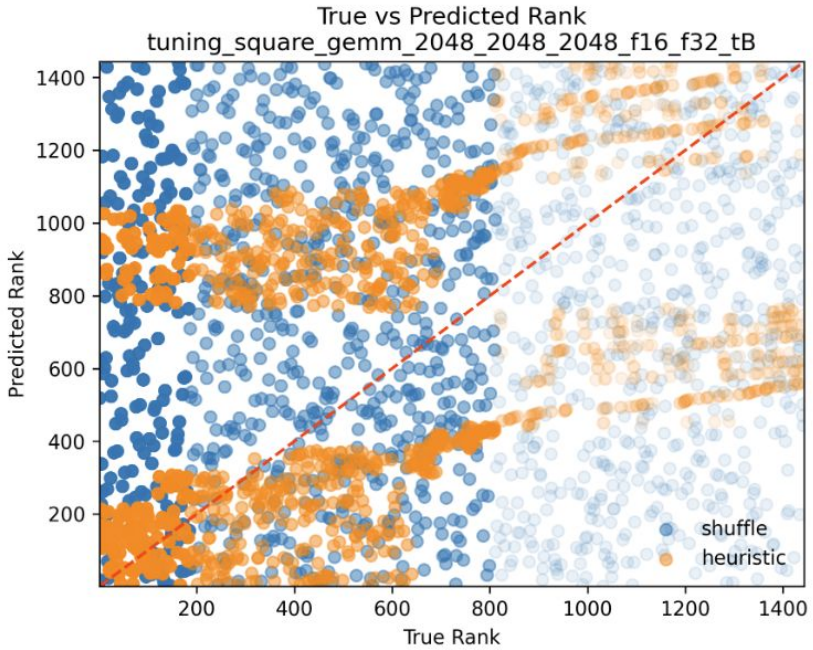
Random Forest Predictor



Accurate but high-cost

- Long training time
- Not interpretable
- 400 MB blob per GPU

Handwritten Heuristic



Inaccurate but maintainable

- Derived from random forest features and expert intuition

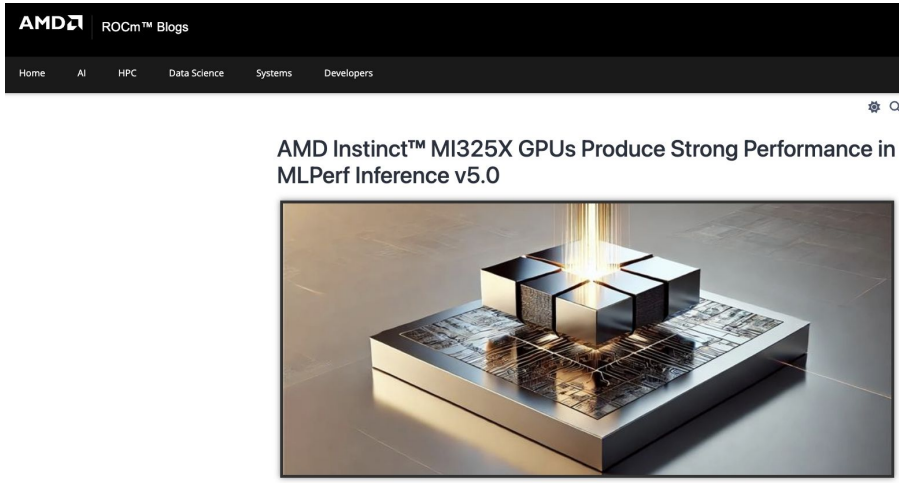
Key features selected:

- Power-of-two tile K
- Intrinsic area to volume ratio
- Quantization inefficiency

Results: ~2.5x tuning time reduction

Results

- 15% end-to-end improvement on SDXL in MLPerf Inference v5.0
- Used for both kernel and whole-model tuning across data-center and desktop AMD GPUs
- Vehicle for finding better compiler heuristics
- Pretty good compiler fuzzer



Ongoing Work

- Support for more ML operators
  - Currently only matmul, convolution, and attention
- Moving constraint generation to the compiler via the SMT dialect
  - Also automatically synthesize compile time verification logic
- Integration with hipDNN
- Support for non-GPU targets

Engagement

IREE

- https://iree.dev
- GitHub: https://github.com/iree-org/iree
- Active Discord Server: https://discord.gg/wEWh6Z9nMU

Questions?