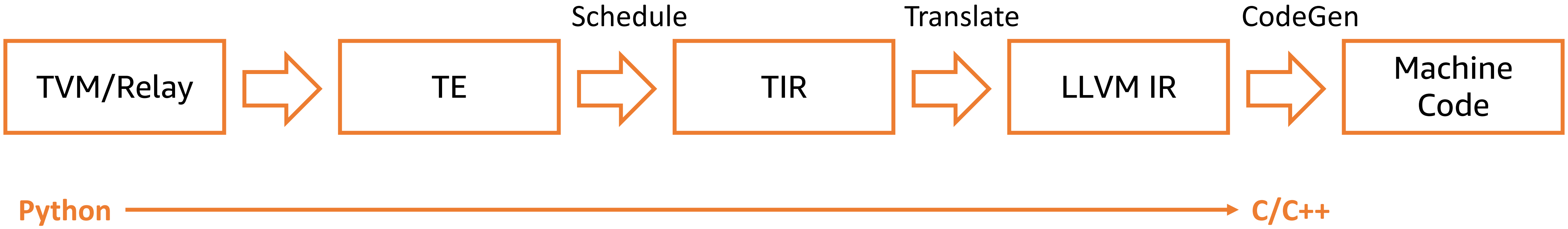
Tensorization with MLIR

Liangfu Chen, SDE@AWS

Compilation Flow



TVM/Relay → TE → Schedule → TIR → Translate → LLVM IR → CodeGen → Machine Code

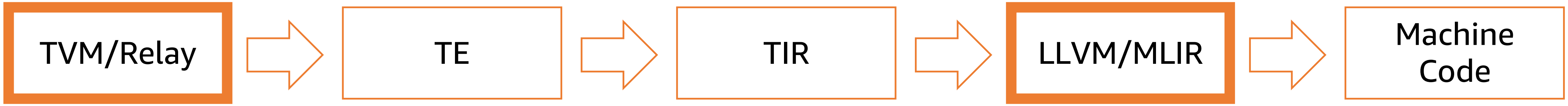
Python → C/C++

Easier to control workflow

Faster execution

Why tensorize in MLIR?

Idea: Use the right tool to do the right job.



Graph level optimizations

```
AlterOpLayout
BackwardFoldScaleAxis
Conv2dToSparse
DenseToSparse
FoldConstant
PartitionGraph
SimplifyInference
...
```

TIR transformations

```
BF16Legalize
BackwardFoldScaleAxis
LoopPartition
LowerIntrin
Simplify
UnrollLoop
VectorizeLoop
...
```

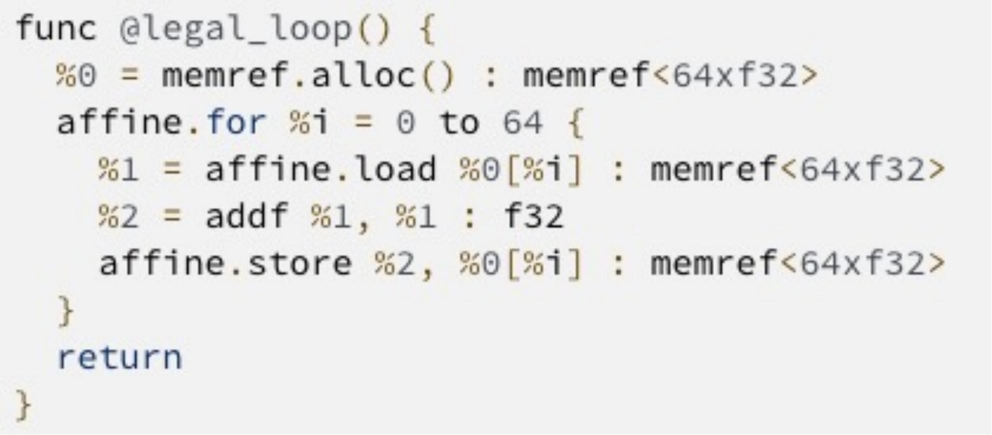
Affine loop transformations

```
-affine-loop-fusion
-affine-loop-invariant-code-motion
-affine-loop-tile
-affine-loop-unroll
-affine-super-vectorize
-cse: Eliminate common sub-expressions
-normalize-memrefs: Normalize memrefs
...
```

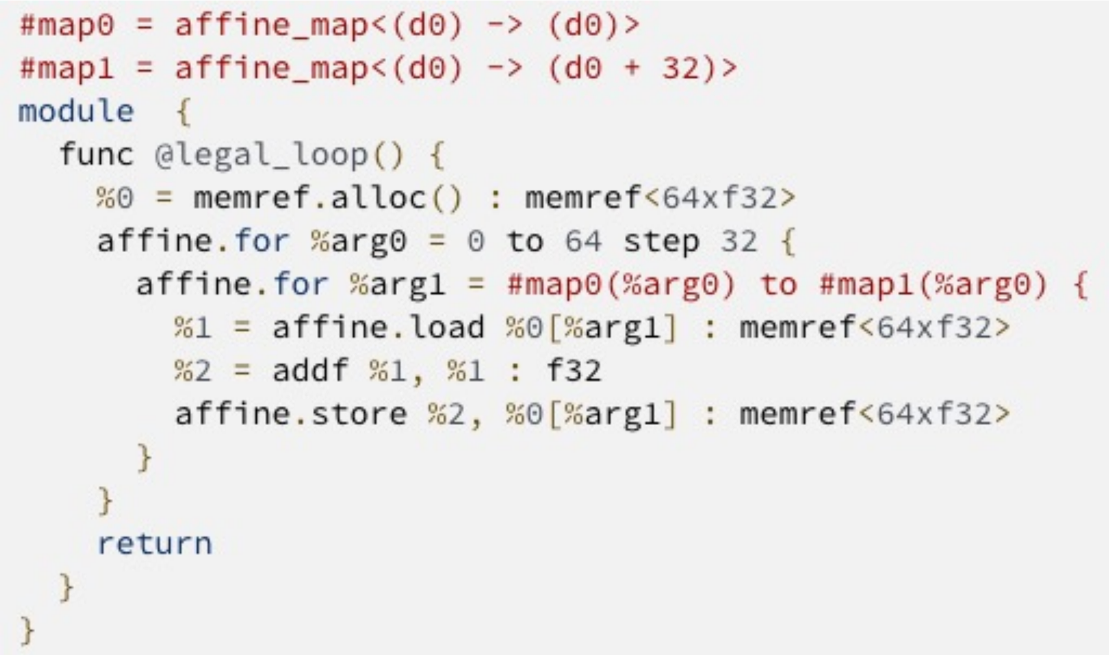
How to tensorize in MLIR?

Idea: Loop tiling and pattern matching.

- Loop tiling (`-affine-loop-tile="tile-size=32"`)



```
func @legal_loop() {
  %0 = memref.alloc() : memref<64xf32>
  affine.for %i = 0 to 64 {
    %1 = affine.load %0[%i] : memref<64xf32>
    %2 = addf %1, %1 : f32
    affine.store %2, %0[%i] : memref<64xf32>
  }
  return
}
```

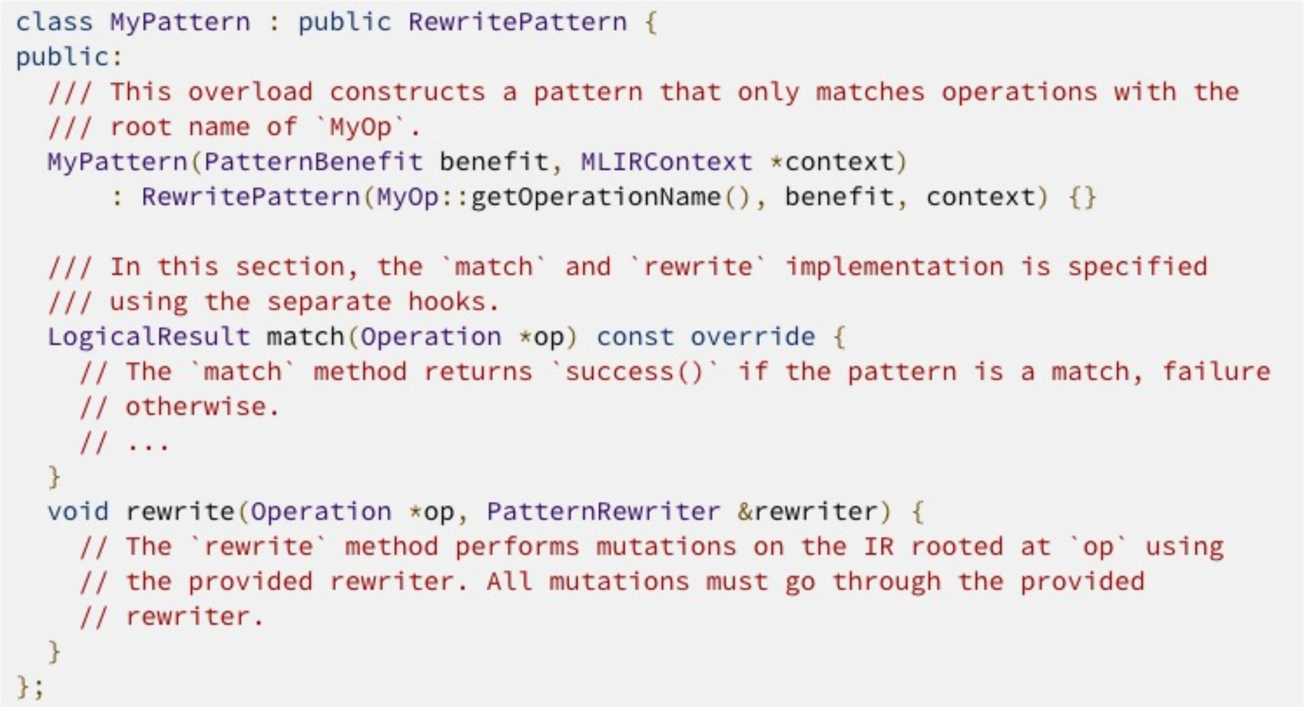


```
#map0 = affine_map<(d0) -> (d0)>
#map1 = affine_map<(d0) -> (d0 + 32)>
module  {
  func @legal_loop() {
    %0 = memref.alloc() : memref<64xf32>
    affine.for %arg0 = 0 to 64 step 32 {
      affine.for %arg1 = #map0(%arg0) to #map1(%arg0) {
        %1 = affine.load %0[%arg1] : memref<64xf32>
        %2 = addf %1, %1 : f32
        affine.store %2, %0[%arg1] : memref<64xf32>
      }
    }
    return
  }
}
```

How to tensorize in MLIR?

Idea: Loop tiling and pattern matching.

- Pattern matching



```
class MyPattern : public RewritePattern {
public:
  /// This overload constructs a pattern that only matches operations with the
  /// root name of `MyOp`.
  MyPattern(PatternBenefit benefit, MLIRContext *context)
      : RewritePattern(MyOp::getOperationName(), benefit, context) {}

  /// In this section, the `match` and `rewrite` implementation is specified
  /// using the separate hooks.
  LogicalResult match(Operation *op) const override {
    // The `match` method returns `success()` if the pattern is a match, failure
    // otherwise.
    // ...
  }
  void rewrite(Operation *op, PatternRewriter &rewriter) {
    // The `rewrite` method performs mutations on the IR rooted at `op` using
    // the provided rewriter. All mutations must go through the provided
    // rewriter.
  }
};
```

Conclusion

- Transform Relay to MLIR for tensorization
  - Transform Relay -> TIR -> MLIR to get tensor expressions
- Leverage MLIR to perform tensorize operations
  - Perform loop tiling and pattern matching
  - Polyhedral analysis and optimizations
- Build whole graph into a single function
  - Scanning whole graph helps management of scratchpad in compiler
- Perform data rate matching between loop nests
  - Improves temporal locality