A Template-Based Code Generation Approach for MLIR

Florian Drescher

School of Computation, Information and Technology
Technical University of Munich

Supervisor: Alexis Engelke

2023-05-11

Copy-and-Patch Compilation[1]

- ▶ Template-based code generation

- ✔ Very fast compilation
- ✔ Still good code

- ✘ High implementation effort
- ✘ Does not integrate with LLVM

$$\begin{bmatrix} a_{11} & \cdots \\ \vdots & \ddots \end{bmatrix} \times \begin{bmatrix} b_{11} & \cdots \\ \vdots & \ddots \end{bmatrix} + \begin{bmatrix} c_{11} & \cdots \\ \vdots & \ddots \end{bmatrix}$$

$$\Downarrow$$

```
func @foo(%A: mat, %B: mat, %C: mat) -> mat {
    %X = matmul mat %A, %B
    %Y = matadd mat %X, %C
    ret mat %Y
}
```

$$\Downarrow$$

Native Code

[1] H Xu and F Kjolstad. "Copy-and-Patch Compilation: A Fast Compilation Algorithm for High-Level Languages and Bytecode". In: *Proc. ACM Program. Lang.* 5.OOPSLA (Oct. 2021). DOI: 10.1145/3485513.

MLIR - Multi-Level IR Compiler Framework[2]

```
func.func @increment(%arg0 : i64) -> i64 {
    %a = arith.constant 1 : i64
    %b = arith.addi %a, %arg0 : i64
    func.return %b : i64
}
```

- ➡ Custom dialects and instructions with lowering to LLVM IR
- ➡ Derive templates automatically from defined lowerings

- ✔ Extendable for custom DSLs
- ✔ Adopted in real world use-cases
- ✔ Part of LLVM project

- ✘ Opaque instruction semantics – only defined as lowerings

[2] C Lattner et al. "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation". In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308.

Automatic Template Generation

out = arith.addi(in : i64, in : i64)

Extract semantics for:

- Input and outputs
- Regions
- Block arguments
- Terminator operands

Limitations:

? Derive relation between attribute values and code

? Lowering relying on other values (blocks, scopes . . .)

Example – Automatic Template Generation

Derived Template for addi Instruction

```
declare void @next(ptr)
@off0 = external global i8, align 1
@off1 = external global i8, align 1
@off2 = external global i8, align 1
define void @add(ptr %mem) {
    %ptr1 = getelementptr i64, ptr %mem, i64 ptrtoint (ptr @off0 to i64)
    %op1 = load i64, ptr %ptr1, align 8
    %ptr2 = getelementptr i64, ptr %mem, i64 ptrtoint (ptr @off1 to i64)
    %op2 = load i64, ptr %ptr2, align 8
    %res = add i64 %op1, %op2
    %ptr3 = getelementptr i64, ptr %mem, i64 ptrtoint (ptr @off2 to i64)
    store i64 %res, ptr %ptr3, align 8
    musttail call void @next(ptr %mem)
    ret void
}
```

Optimizations

Implemented:

- Using registers to pass variables between templates
- Evaluation of constant instructions during template generation

Future:

- Inline region template into a parent instruction
- Use native `%rsp` instead of explicit first argument
- Propagate constants inside hot loops

Applications

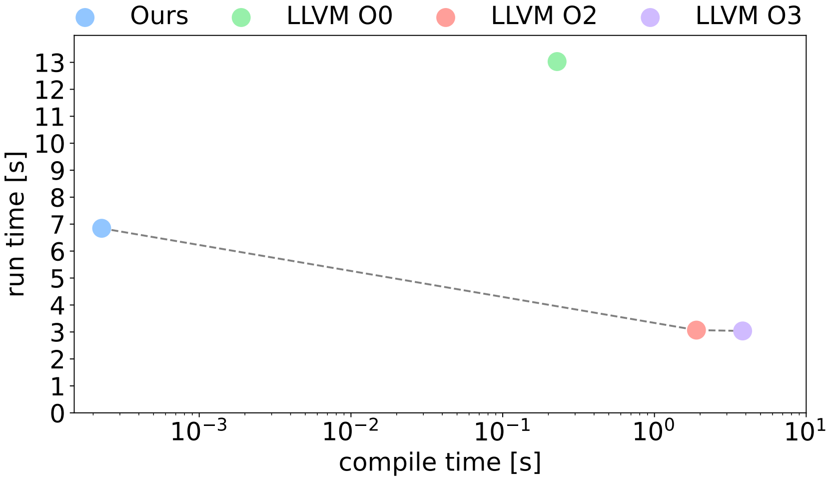
ONNX MLIR

- ONNX machine learning model to native code with MLIR
- Replace entire lowering pipeline with our approach
- Evaluated on ResNetv43

Example

```
[...]
%2 = onnx.constant dense<1.0> : tensor<...>
%3 = onnx.constant dense<2.0> : tensor<...>
%4 = onnx.Conv(%arg0, %2, %3) {...}
    : (tensor<...>, ...) -> tensor<...>
%5 = onnx.Relu(%4)
    : (tensor<...>) -> tensor<...>
%6 = onnx.MaxPoolSingleOut(%5) {...}
    : (tensor<...>) -> tensor<...>
[...]
```

Results – ONNX MLIR



+ 1000x faster compilation than O0
+ 7500x faster compilation than O2

+ 2x faster execution than LLVM O0
– 2x slower execution than LLVM O2

Applications

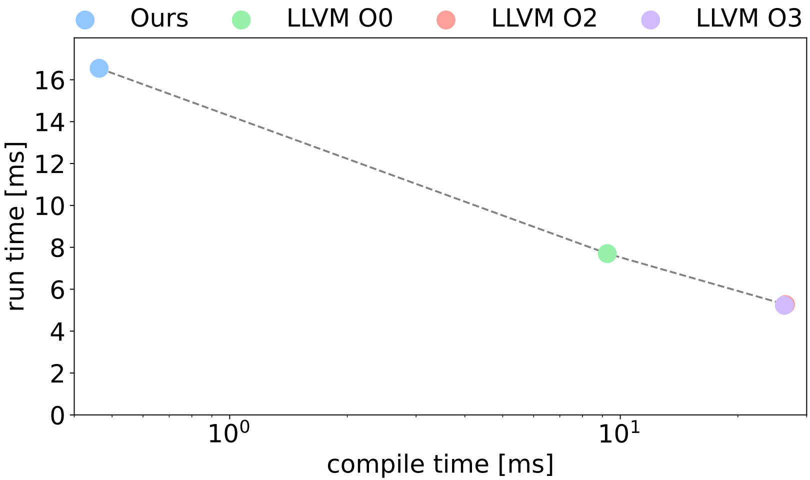
LingoDB

- Database query execution engine based on MLIR
- Replace final lowering to native code with our approach
- Evaluated on TPC-H queries

Example

```
[...]
func.func @nextRow(!util.ref<i8>)
func.func @addInt(!util.ref<i8>, i1, i32)
[...]
%c1 = arith.constant 1 : i64
%elem = util.load %row[%idx] : <i64> -> i64
%match = arith.cmpi eq, %elem, %c1 : i64
scf.if %match {
    func.call @addInt(%out, true, %elem)
        : (!util.ref<i8>, i1, i32) -> ()
    func.call @nextRow(%output)
        : (!util.ref<i8>) -> ()
}
[...]
```

Results – LingoDB

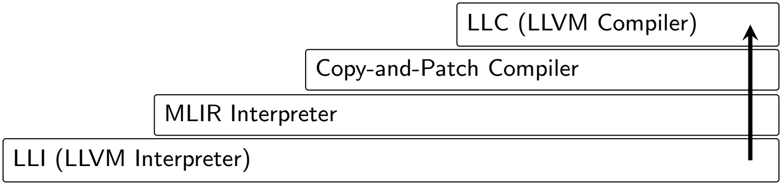


+ 20x faster compilation than O0
+ 60x faster compilation than O2

– 2x slower execution than LLVM O0
– 3x slower execution than LLVM O2

Vision – Which Templates to Generate?

- Precompile all possible template configurations
  - ✔ Demonstrated in "Copy-and-Patch Compilation" for two use-cases
  - ✘ Not always possible/desirable
- On-demand compilation
  - Enables fine-grained caching of code
  - Use in adaptive compilation as additional tier



Summary

- Template-based compilation allows for very fast compilation
- Deriving templates automatically from MLIR to LLVM IR lowering avoids high implementation effort
- Good trade-off between compilation and execution time
  - in LingoDB (60x faster compilation vs. 3x slower execution)
  - in ONNX MLIR (7400x fast compilation vs. 2x slower execution)
- Deeper integrate template-based compilation into adaptive optimization

➥ Establish template-based compilation as code generation approach for MLIR