Learn Compiler Design Through xDSL: A Practical Course for AI/ML Engineers

Djordje Todorovic

August 2025

Contents

1 Introduction: Compilers as Representation and Transformation Engines

1.1 Welcome to the World of Compilers

If you're coming from an AI/ML background, you already understand the power of transforming data through layers of abstraction. Compilers are remarkably similar: they take programs written in one representation and systematically transform them into another, more optimized or more executable form.

Think of a compiler as a sophisticated pipeline that:

1. **Represents** programs as structured data (Abstract Syntax Trees, Intermediate Representations)
2. **Transforms** these representations through a series of optimization passes
3. **Analyzes** code to understand dependencies, patterns, and optimization opportunities
4. **Generates** efficient target code for specific hardware

1.2 Why xDSL?

xDSL is a Python-native compiler framework that makes compiler concepts accessible to Python programmers. Unlike traditional compiler frameworks written in C++ (like LLVM), xDSL allows you to:

- Build compilers using familiar Python syntax
- Prototype and experiment quickly
- Understand compiler internals without low-level complexity
- Leverage the entire Python ecosystem for analysis and visualization

1.3 Course Philosophy: Learn by Building

This course takes a hands-on approach. Instead of starting with theory, we'll build small compilers and optimization passes from day one. Each concept will be introduced through practical examples that you can run, modify, and experiment with.

1.4 What You'll Learn

By the end of this course, you'll understand:

- How compilers represent programs internally (IR - Intermediate Representations)
- What operations and operators mean in a compiler context
- How to write analysis passes that extract information from code
- How to implement transformations that optimize programs
- The connection between compiler optimizations and ML model optimization

2 Chapter 1: Understanding Intermediate Representations (IR)

2.1 What is an IR?

An Intermediate Representation (IR) is the compiler's internal language for representing programs. Just as neural networks use tensors to represent data, compilers use IRs to represent code structure and semantics.

Key properties of IRs:

- **Abstract**: Higher-level than machine code, but more structured than source code
- **Explicit**: Makes implicit operations visible (e.g., memory allocations, type conversions)
- **Analyzable**: Designed for easy pattern matching and transformation
- **Hierarchical**: Can represent nested structures (functions, loops, blocks)

2.2 The Compiler Pipeline: From Source to Machine Code

Before diving into xDSL specifics, let's understand how compilers work. A compiler is traditionally divided into three main phases, each with specific responsibilities:

2.2.1 Frontend: Understanding Your Code

The frontend translates source code into an intermediate representation. It consists of:

1. Lexical Analysis (Lexer/Tokenizer)

- Breaks source code into tokens (keywords, identifiers, operators)
- Like splitting a sentence into words
- Example: `"x = 42 + y"` becomes tokens: [`ID:x`, `ASSIGN`, `NUM:42`, `PLUS`, `ID:y`]

2. Syntax Analysis (Parser)

- Builds an Abstract Syntax Tree (AST) from tokens
- Checks grammatical structure
- Like diagramming a sentence to understand its structure

```
# Source: x = 42 + y
# AST representation:
Assignment(
    left=Variable("x"),
    right=BinaryOp(
        op="+",
        left=Number(42),
        right=Variable("y")
    )
)
```

Listing 1: Simple AST Example

3. Semantic Analysis

- Type checking: Is this operation valid?
- Symbol resolution: What does this variable refer to?
- Example: Can't add a string to a number, variables must be declared

2.2.2 Middle-end: Optimization Central

The middle-end works on the IR to optimize code:

- **Platform-independent**: Optimizations work for any target
- **Examples**: Dead code elimination, constant folding, loop optimization
- **Multiple passes**: Each optimization is a separate pass over the IR

2.2.3 Backend: Generating Machine Code

The backend translates optimized IR to target-specific code:

- **Instruction selection**: Choose machine instructions
- **Register allocation**: Assign variables to CPU registers
- **Instruction scheduling**: Order operations for performance

2.3 Learning from Giants: LLVM and MLIR

2.3.1 LLVM: The Industry Standard

LLVM (Low Level Virtual Machine) is the most widely used compiler infrastructure:

- Powers Clang (C/C++), Swift, Rust, and many other languages
- Provides a well-defined IR (LLVM IR) that many frontends target
- Excellent tutorial: Kaleidoscope Tutorial
- Shows how to build a complete compiler for a simple language

```
; LLVM IR for: int add(int a, int b) { return a + b; }
define i32 @add(i32 %a, i32 %b) {
entry:
  %sum = add i32 %a, %b
  ret i32 %sum
}
```

Listing 2: LLVM IR Example

2.3.2 MLIR: Multi-Level IR

MLIR (Multi-Level Intermediate Representation) extends LLVM's concepts:

- Developed by Google, now part of LLVM project
- Supports multiple IRs (dialects) in the same infrastructure
- Used by TensorFlow, PyTorch, and other ML frameworks
- Tutorial: Toy Language Tutorial
- xDSL is heavily inspired by MLIR but implemented in Python

```
// High-level ML operation
%result = "tf.MatMul"(%a, %b) : (tensor<2x3xf32>, tensor<3x4xf32>)
                                -> tensor<2x4xf32>

// After lowering to linalg dialect
%result = linalg.matmul ins(%a, %b : memref<2x3xf32>, memref<3x4xf32>)
                       outs(%c : memref<2x4xf32>)

// After lowering to loops
scf.for %i = 0 to 2 {
  scf.for %j = 0 to 4 {
    scf.for %k = 0 to 3 {
      // actual computation
    }
  }
}
```

Listing 3: MLIR Example - Multiple Abstraction Levels

2.4 Why xDSL Uses IR Instead of AST

While traditional compilers start with AST, xDSL (like MLIR) works directly with IR because:

- **Uniformity**: All transformations work on the same structure
- **Composability**: Easy to combine different languages/dialects
- **Reusability**: Optimizations work across different source languages
- **Analysis-friendly**: SSA form makes many analyses trivial

Think of it this way:

- **AST**: Like a sentence diagram - shows structure
- **IR**: Like assembly with types - shows computation

2.5 SSA Form: Single Static Assignment

xDSL uses SSA (Single Static Assignment) form, where each variable is assigned exactly once. This is similar to functional programming and makes many analyses simpler.

```
# Traditional form
x = 5
x = x + 1
y = x * 2

# SSA form (conceptual)
x_0 = 5
x_1 = x_0 + 1
y_0 = x_1 * 2
```

Listing 4: Traditional vs SSA Form

Why SSA matters:

- **No ambiguity**: Each value has exactly one definition
- **Easy analysis**: Use-def chains are explicit
- **Better optimization**: Many optimizations become simpler
- **Parallel-friendly**: Dependencies are clear

2.6 Your First xDSL Program

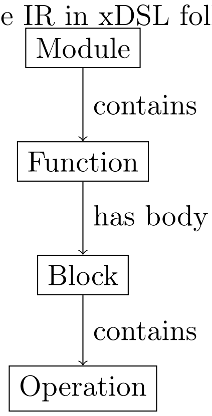
Let's create a simple IR program using xDSL:

```python
from xdsl.context import Context
from xdsl.dialects import builtin, arith, func
from xdsl.ir import Operation, Block, Region
from xdsl.builder import Builder

# Create a context (like a namespace for dialects)
ctx = Context()
ctx.load_dialect(builtin.Builtin)
ctx.load_dialect(arith.Arith)
ctx.load_dialect(func.Func)

# Build a simple function that adds two numbers
@Builder.implicit_region
def create_add_function():
    # Create function signature: (i32, i32) -> i32
    func_type = func.FunctionType.from_lists([builtin.i32, builtin.i32], [builtin.i32])

    with func.FuncOp("add", func_type, visibility="public").body:
        # Get function arguments
        arg0 = Block.current().args[0]
        arg1 = Block.current().args[1]

        # Create add operation
        result = arith.Addi(arg0, arg1).result

        # Return the result
        func.Return(result)

module = builtin.ModuleOp([create_add_function()])
print(module)
```

Listing 5: Creating IR in xDSL

2.7 Understanding IR Structure

The IR in xDSL follows a hierarchical structure:



2.8 Dialects: Modular IR Design

xDSL organizes operations into *dialects* - collections of related operations and types. This is similar to how PyTorch has different modules (nn, optim, etc.).

Common dialects in xDSL:

- `arith`: Arithmetic operations (add, multiply, divide)
- `func`: Function definitions and calls
- `memref`: Memory operations (like NumPy arrays)
- `scf`: Structured control flow (loops, conditionals)
- `linalg`: Linear algebra operations (matrix multiply, convolution)

3 Chapter 2: Operations and Operators

3.1 What is an Operation?

An operation in xDSL is the fundamental unit of computation. Each operation:

- Has a name (e.g., `arith.addi` for integer addition)
- Takes input values (operands)
- Produces output values (results)
- May have attributes (compile-time constants)
- Can contain regions (nested blocks of operations)

3.2 Creating Custom Operations

Let's define a custom operation for matrix operations:

```python
from xdsl.irdl import (
    IRDLOperation,
    OpResult,
    Operand,
    OperandDef,
    ResultDef,
    op_def,
    result_def
)
from xdsl.dialects.builtin import TensorType

@op_def
class MatMulOp(IRDLOperation):
    """Matrix multiplication operation"""
    name = "my_dialect.matmul"

    # Define operands (inputs)
    lhs = operand_def(TensorType)
    rhs = operand_def(TensorType)

    # Define results (outputs)
    result = result_def(TensorType)

    def __init__(self, lhs: Value, rhs: Value):
        super().__init__(operands=[lhs, rhs],
                        result_types=[infer_matmul_type(lhs, rhs)])

    def verify(self):
        """Verify that the operation is well-formed"""
        # Check that dimensions are compatible
        lhs_shape = self.lhs.type.shape
        rhs_shape = self.rhs.type.shape

        if lhs_shape[-1] != rhs_shape[0]:
            raise ValueError("Incompatible matrix dimensions")
```

Listing 6: Defining Custom Operations

3.3 Operation Semantics

Every operation has well-defined semantics - what it means to execute that operation:

```python
# Arithmetic operations
add_op = arith.Addi(a, b)  # result = a + b
mul_op = arith.Muli(a, b)  # result = a * b

# Memory operations
alloc_op = memref.Alloc.get(shape=[10, 20], element_type=f32)  # Allocate 10x20 array
load_op = memref.Load.get(memref_value, indices)  # Load from memory
store_op = memref.Store.get(value, memref_value, indices)  # Store to memory

# Control flow operations
for_op = scf.For(lower_bound, upper_bound, step) # for loop
if_op = scf.If(condition, has_else=True)  # if-then-else
```

Listing 7: Operation Semantics Example

3.4 Attributes vs Operands

Understanding the difference between attributes and operands is crucial:

- **Attributes**: Compile-time constants (shapes, types, flags)
- **Operands**: Runtime values (variables, intermediate results)

```python
# Constant has an attribute (the value)
const = arith.Constant.from_int_and_width(42, 32)  # 42 is an attribute

# Add has operands (runtime values)
result = arith.Addi(x, y)  # x and y are operands

# Alloc has attributes (shape) but no operands
mem = memref.Alloc.get(shape=[100], element_type=f32)  # shape is attribute
```

Listing 8: Attributes vs Operands

4 Chapter 3: Analysis Passes

4.1 What is Analysis?

Analysis passes extract information from IR without modifying it. They answer questions like:

- Which variables are used where? (Use-Def Analysis)
- Which operations can run in parallel? (Dependency Analysis)
- What are the loop bounds? (Range Analysis)
- Which operations are dead code? (Liveness Analysis)

4.2 Writing a Simple Analysis Pass

Let's write an analysis that counts operations by type:

```python
from xdsl.passes import Pass
from xdsl.ir import Operation, OpResult
from collections import defaultdict

class OperationCounterPass(Pass):
    """Count operations by type in the IR"""

    name = "count-ops"

    def apply(self, module: builtin.ModuleOp) -> None:
        op_counts = defaultdict(int)

        # Walk through all operations in the module
        for op in module.walk():
            op_counts[op.name] += 1

        # Print analysis results
        print("Operation counts:")
        for op_name, count in sorted(op_counts.items()):
            print(f"  {op_name}: {count}")

        # Store results for use by other passes
        self.op_counts = op_counts
```

Listing 9: Operation Counter Analysis

4.3 Use-Def Chains

Use-Def chains track where values are defined and used:

```python
class UseDefAnalysis(Pass):
    """Analyze use-def relationships"""

    def apply(self, module: builtin.ModuleOp) -> None:
        # Build use-def chains
        for op in module.walk():
            for result in op.results:
                print(f"Value {result} defined by {op.name}")
                print(f"  Used by: {[use.operation.name for use in 
    result.uses]}")

    def find_unused_values(self, module: builtin.ModuleOp) -> list[
    OpResult]:
        """Find values that are never used"""
        unused = []
        for op in module.walk():
            for result in op.results:
                if not result.uses:
                    unused.append(result)
        return unused
```

Listing 10: Use-Def Analysis

4.4 Dominance Analysis

Dominance tells us which operations must execute before others:

```
from xdsl.ir import Block
from xdsl.irdl.dominance import DominanceInfo

class DominanceAnalysis(Pass):
    """Analyze control flow dominance"""

    def apply(self, module: builtin.ModuleOp) -> None:
        # For each function in the module
        for func_op in module.body.ops:
            if isinstance(func_op, func.FuncOp):
                dom_info = DominanceInfo(func_op.body)

                # Check dominance relationships
                for block in func_op.body.blocks:
                    dominators = dom_info.get_dominators(block)
                    print(f"Block {block} dominated by: {dominators}")
```

Listing 11: Dominance Analysis

4.5 Dataflow Analysis

Dataflow analysis propagates information through the program:

```
class ConstantPropagationAnalysis(Pass):
    """Analyze which values are compile-time constants"""

    def apply(self, module: builtin.ModuleOp) -> None:
        known_constants = {}

        # First pass: identify constants
        for op in module.walk():
            if isinstance(op, arith.Constant):
                known_constants[op.result] = op.value.value

        # Iteratively propagate constants
        changed = True
        while changed:
            changed = False
            for op in module.walk():
                if isinstance(op, arith.Addi):
                    # If both operands are known constants
                    if (op.lhs in known_constants and
                        op.rhs in known_constants):
                        result_value = (known_constants[op.lhs] +
                                      known_constants[op.rhs])
                        if op.result not in known_constants:
                            known_constants[op.result] = result_value
                            changed = True

        return known_constants
```

Listing 12: Constant Propagation Analysis

5 Chapter 4: Transformations and Optimizations

5.1 What is a Transformation?

Transformations modify the IR to improve performance, reduce code size, or prepare for further optimizations. Unlike analysis passes, transformations actually change the program structure.

Key principles:

- **Correctness**: Must preserve program semantics
- **Profitability**: Should improve some metric (speed, size, power)
- **Composability**: Should work well with other transformations

5.2 Pattern-Based Rewriting

xDSL provides powerful pattern matching for transformations:

```
from xdsl.pattern_rewriter import (
    PatternRewriter,
    RewritePattern,
    op_type_rewrite_pattern
)

class FoldAddZero(RewritePattern):
    """Fold x + 0 -> x"""

    @op_type_rewrite_pattern
    def match_and_rewrite(self, op: arith.Addi,
                          rewriter: PatternRewriter) -> None:
        # Check if right operand is zero
        if isinstance(op.rhs.owner, arith.Constant):
            if op.rhs.owner.value.value == 0:
                # Replace all uses of the add with the left operand
                rewriter.replace_op(op, [op.lhs])
                return

        # Check if left operand is zero
        if isinstance(op.lhs.owner, arith.Constant):
            if op.lhs.owner.value.value == 0:
                rewriter.replace_op(op, [op.rhs])
```

Listing 13: Simple Algebraic Optimization

5.3 Common Optimizations

5.3.1 Dead Code Elimination

Remove operations whose results are never used:

```
class DeadCodeElimination(Pass):
    """Remove unused operations"""

    def apply(self, module: builtin.ModuleOp) -> None:
```

```python
        ops_to_remove = []

        for op in module.walk():
            # Skip operations with side effects
            if self.has_side_effects(op):
                continue

            # Check if any results are used
            all_dead = all(not result.uses for result in op.results)

            if all_dead:
                ops_to_remove.append(op)

        # Remove dead operations
        for op in ops_to_remove:
            op.erase()

    def has_side_effects(self, op: Operation) -> bool:
        """Check if operation has side effects"""
        # Memory writes, function calls, etc.
        return isinstance(op, (memref.Store, func.Call))
```

Listing 14: Dead Code Elimination

5.3.2 Common Subexpression Elimination

Identify and eliminate redundant computations:

```python
class CommonSubexpressionElimination(Pass):
    """Eliminate redundant computations"""

    def apply(self, module: builtin.ModuleOp) -> None:
        # Map from (op_type, operands) to result
        expression_map = {}

        for op in module.walk():
            if self.is_pure(op):
                # Create a key for this expression
                key = self.get_expression_key(op)

                if key in expression_map:
                    # Found duplicate - replace with existing
                    existing_result = expression_map[key]
                    op.results[0].replace_by(existing_result)
                    op.erase()
                else:
                    # First occurrence - remember it
                    expression_map[key] = op.results[0]

    def get_expression_key(self, op: Operation):
        """Create hashable key for expression"""
        return (op.name, tuple(op.operands))
```

Listing 15: Common Subexpression Elimination

5.3.3 Loop Optimizations

Optimize loop structures for better performance:

```python
class LoopUnrolling(Pass):
    """Unroll small loops"""

    def apply(self, module: builtin.ModuleOp) -> None:
        for op in module.walk():
            if isinstance(op, scf.For):
                if self.should_unroll(op):
                    self.unroll_loop(op)

    def should_unroll(self, loop: scf.For) -> bool:
        """Decide if loop should be unrolled"""
        # Get loop bounds if constant
        if (isinstance(loop.lb.owner, arith.Constant) and
            isinstance(loop.ub.owner, arith.Constant)):
            lb = loop.lb.owner.value.value
            ub = loop.ub.owner.value.value
            iterations = ub - lb

            # Unroll small loops
            return iterations <= 4
        return False

    def unroll_loop(self, loop: scf.For):
        """Completely unroll a loop"""
        # Clone loop body for each iteration
        # Update induction variable references
        # Remove original loop
        pass  # Implementation details omitted
```

Listing 16: Loop Unrolling

5.4 Lowering Transformations

Lowering transforms high-level operations into simpler ones:

```python
class LowerMatrixOps(Pass):
    """Lower matrix operations to loops"""

    def apply(self, module: builtin.ModuleOp) -> None:
        for op in module.walk():
            if isinstance(op, MatMulOp):
                self.lower_matmul(op)

    def lower_matmul(self, matmul: MatMulOp):
        """Lower matmul to three nested loops"""
        builder = Builder.before(matmul)

        # Get dimensions
        m, k = matmul.lhs.type.shape
        _, n = matmul.rhs.type.shape

        # Allocate result
        result = builder.insert(memref.Alloc.get([m, n], f32))

        # Generate three nested loops
        with builder.insert(scf.For(0, m, 1)) as i:
            with builder.insert(scf.For(0, n, 1)) as j:
                # Initialize accumulator
                zero = builder.insert(arith.Constant.from_float(0.0, f32))

                with builder.insert(scf.For(0, k, 1, [zero])) as (k_idx, acc):
                    # Load elements
                    a_elem = builder.insert(memref.Load(matmul.lhs, [i, k_idx]))
                    b_elem = builder.insert(memref.Load(matmul.rhs, [k_idx, j]))

                    # Multiply and accumulate
                    prod = builder.insert(arith.Mulf(a_elem, b_elem))
                    new_acc = builder.insert(arith.Addf(acc, prod))

                    scf.Yield(new_acc)

                # Store result
                builder.insert(memref.Store(acc, result, [i, j]))

        # Replace matmul with lowered result
        matmul.results[0].replace_by(result)
        matmul.erase()
```

Listing 17: Lowering High-Level Operations

6 Chapter 5: Practical Examples

6.1 Building a Simple Expression Compiler

Let’s build a complete compiler for arithmetic expressions:

```
from dataclasses import dataclass
from typing import Union

# Define AST nodes
@dataclass
class Expr:
    pass

@dataclass
class Number(Expr):
    value: int

@dataclass
class BinaryOp(Expr):
    left: Expr
    op: str
    right: Expr

# Parser (simplified)
def parse(text: str) -> Expr:
    """Parse expression like ’2 + 3 * 4’"""
    # Implementation omitted for brevity
    pass

# IR Generator
class ExprIRGen:
```

```python
    def __init__(self):
        self.builder = Builder()

    def compile(self, expr: Expr) -> OpResult:
        """Compile expression to xDSL IR"""
        if isinstance(expr, Number):
            return arith.Constant.from_int_and_width(
                expr.value, 32).result

        elif isinstance(expr, BinaryOp):
            left = self.compile(expr.left)
            right = self.compile(expr.right)

            if expr.op == '+':
                return arith.Addi(left, right).result
            elif expr.op == '*':
                return arith.Muli(left, right).result
            # ... other operations

# Optimizer
def optimize(module: builtin.ModuleOp):
    """Apply optimization passes"""
    passes = [
        ConstantFolding(),
        CommonSubexpressionElimination(),
        DeadCodeElimination(),
    ]

    for pass_instance in passes:
        pass_instance.apply(module)

# Complete pipeline
def compile_expression(text: str):
    # Parse
    ast = parse(text)

    # Generate IR
    ir_gen = ExprIRGen()
    result = ir_gen.compile(ast)

    # Wrap in module
    module = builtin.ModuleOp([
        func.FuncOp("compute",
                   func.FunctionType.from_lists([], [builtin.i32]),
                   body=[func.Return(result)])
    ])

    # Optimize
    optimize(module)

    return module
```

Listing 18: Expression Compiler

6.2 Implementing a Domain-Specific Optimization

Let’s create an optimization for neural network operations:

```python
class FuseBatchNormIntoConv(Pass):
    """Fuse batch normalization into preceding convolution"""

    def apply(self, module: builtin.ModuleOp) -> None:
        for conv in module.walk():
            if isinstance(conv, nn.Conv2d):
                # Look for batch norm following conv
                for use in conv.result.uses:
                    if isinstance(use.operation, nn.BatchNorm2d):
                        self.fuse(conv, use.operation)

    def fuse(self, conv: nn.Conv2d, bn: nn.BatchNorm2d):
        """Fuse batch norm parameters into convolution"""
        # Extract batch norm parameters
        gamma = bn.gamma  # scale
        beta = bn.beta    # shift
        mean = bn.running_mean
        var = bn.running_var
        eps = bn.eps

        # Compute fused weights and bias
        std = sqrt(var + eps)
        fused_weight = conv.weight * (gamma / std)
        fused_bias = gamma * (conv.bias - mean) / std + beta

        # Create new convolution with fused parameters
        fused_conv = nn.Conv2d(
            weight=fused_weight,
            bias=fused_bias,
            # ... other conv parameters
        )

        # Replace conv -> bn with fused conv
        bn.result.replace_by(fused_conv.result)
        conv.erase()
        bn.erase()
```

Listing 19: Neural Network Optimization

6.3 Building a Mini ML Compiler

Let’s create a compiler for a subset of ML operations:

```python
# Define ML dialect
@dialect_def
class MLDialect(Dialect):
    name = "ml"

@op_def
class LinearOp(IRDLOperation):
    """Linear layer: y = Wx + b"""
    name = "ml.linear"

    input = operand_def(TensorType)
    weight = operand_def(TensorType)
    bias = operand_def(TensorType)

    output = result_def(TensorType)
```

```python

@op_def
class ReluOp(IRDLOperation):
    """ReLU activation"""
    name = "ml.relu"

    input = operand_def(TensorType)
    output = result_def(TensorType)

# Optimization: Fuse Linear + ReLU
class FuseLinearRelu(RewritePattern):
    @op_type_rewrite_pattern
    def match_and_rewrite(self, relu: ReluOp,
                          rewriter: PatternRewriter):
        # Check if input is from linear
        if isinstance(relu.input.owner, LinearOp):
            linear = relu.input.owner

            # Create fused operation
            fused = LinearReluOp(
                linear.input,
                linear.weight,
                linear.bias
            )

            # Replace both ops with fused version
            rewriter.replace_op(relu, [fused.output])
            rewriter.erase_op(linear)

# Lower to hardware operations
class LowerMLToHardware(Pass):
    """Lower ML ops to target hardware"""

    def apply(self, module: builtin.ModuleOp):
        target = self.get_target()  # CPU, GPU, TPU, etc.

        for op in module.walk():
            if isinstance(op, LinearOp):
                if target == "gpu":
                    self.lower_linear_to_cublas(op)
                elif target == "cpu":
                    self.lower_linear_to_loops(op)
```

Listing 20: Mini ML Compiler

7 Chapter 6: Advanced Topics

7.1 Multi-Level IR and Progressive Lowering

Real compilers use multiple levels of abstraction:

```python
class CompilationPipeline:
    """Progressive lowering through multiple IR levels"""

    def compile(self, source_code: str) -> str:
        # Level 1: High-level ML operations
        ml_ir = parse_to_ml_dialect(source_code)
        ml_ir = optimize_ml_level(ml_ir)

        # Level 2: Linear algebra operations
        linalg_ir = lower_ml_to_linalg(ml_ir)
        linalg_ir = optimize_linalg_level(linalg_ir)

        # Level 3: Loops and memory operations
        loop_ir = lower_linalg_to_loops(linalg_ir)
        loop_ir = optimize_loops(loop_ir)

        # Level 4: Target-specific operations
        target_ir = lower_to_target(loop_ir)
        target_ir = optimize_target_specific(target_ir)

        # Code generation
        return generate_code(target_ir)
```

Listing 21: Multi-Level Compilation Pipeline

7.2 Cost Models and Autotuning

Choosing the best optimization requires understanding costs:

```python
class CostModel:
    """Estimate operation costs"""

    def estimate_cost(self, op: Operation) -> float:
        if isinstance(op, arith.Mulf):
            return 1.0  # FP multiply cost
        elif isinstance(op, memref.Load):
            return self.memory_latency(op)
        elif isinstance(op, scf.For):
            iterations = self.analyze_trip_count(op)
            body_cost = sum(self.estimate_cost(body_op)
                          for body_op in op.body.walk())
            return iterations * body_cost

class AutotuningPass(Pass):
    """Try different optimization strategies"""

    def apply(self, module: builtin.ModuleOp):
        strategies = [
            self.try_vectorization,
            self.try_loop_tiling,
            self.try_parallelization,
        ]

        best_module = module
        best_cost = self.cost_model.estimate_cost(module)

        for strategy in strategies:
            candidate = strategy(module.clone())
            cost = self.cost_model.estimate_cost(candidate)

            if cost < best_cost:
                best_module = candidate
                best_cost = cost

        return best_module
```

Listing 22: Cost-Based Optimization

7.3 Connection to ML Compilation

Modern ML frameworks are essentially compilers:

- **PyTorch JIT**: Traces Python code to TorchScript IR
- **TensorFlow XLA**: Compiles TF graphs to optimized kernels
- **JAX**: Uses XLA for JIT compilation
- **Apache TVM**: Optimizes ML models for various hardware

The concepts you've learned apply directly:

- **Graph optimization** = IR transformation
- **Kernel fusion** = Operation merging
- **Memory planning** = Register allocation
- **Quantization** = Type lowering

8 Chapter 7: Hands-On Exercises

8.1 Exercise 1: Build Your Own Optimizer

Implement an optimization that eliminates redundant memory operations:

```
class StoreLoadForwarding(Pass):
    """Forward stored values to subsequent loads"""

    def apply(self, module: builtin.ModuleOp):
        # TODO: Your implementation here
        # Hint: Track recent stores to each memory location
        # Replace loads with stored values when possible
        pass
```

Listing 23: Exercise: Store-Load Forwarding

8.2 Exercise 2: Create a Custom Dialect

Design a dialect for quantum computing operations:

```
@dialect_def
class QuantumDialect(Dialect):
    name = "quantum"

# TODO: Define operations like:
# - Hadamard gate
# - CNOT gate
# - Measurement
# - Quantum circuit optimization passes
```

Listing 24: Exercise: Quantum Dialect

8.3 Exercise 3: Implement Vectorization

Transform scalar operations to vector operations:

```
class Vectorization(Pass):
    """Automatically vectorize loops"""

    def apply(self, module: builtin.ModuleOp):
        # TODO: Find vectorizable loops
        # Transform scalar ops to vector ops
        # Handle loop remainder
        pass
```

Listing 25: Exercise: Auto-Vectorization

8.4 Exercise 4: Build a Simple JIT Compiler

Create a JIT compiler for mathematical expressions:

```
class ExpressionJIT:
    """JIT compile and execute expressions"""

    def compile(self, expr_string: str):
        # Parse expression
        # Generate IR
        # Optimize
        # Generate machine code (or Python bytecode)
        # Return callable function
        pass

# Usage:
jit = ExpressionJIT()
f = jit.compile("x * 2 + y")
result = f(x=5, y=3)  # Should return 13
```

Listing 26: Exercise: JIT Compiler

9 Chapter 8: Real-World Applications

9.1 Case Study 1: Optimizing Neural Network Inference

Let's optimize a real neural network for deployment:

```
def optimize_nn_for_deployment(model: NeuralNetwork):
    """Optimize neural network for inference"""

    # Convert to IR
    ir = convert_nn_to_ir(model)

    # Apply optimizations
    optimizations = [
        # Structural optimizations
        FuseLayers(),            # Conv+BN, Linear+ReLU
        RemoveDropout(),         # Not needed for inference

        # Quantization
        QuantizeWeights(bits=8),
        QuantizeActivations(bits=8),

        # Memory optimizations
        InplaceOperations(),     # Reuse buffers
        MemoryPlanning(),        # Minimize allocation

        # Target-specific
        UseTargetIntrinsics(),  # AVX, NEON, etc.
        AutoTuneKernels(),       # Platform-specific tuning
    ]

    for opt in optimizations:
        ir = opt.apply(ir)

    return ir
```

Listing 27: Neural Network Optimization Pipeline

9.2 Case Study 2: Domain-Specific Language Compiler

Build a compiler for financial computations:

```
# Define financial operations dialect
@dialect_def
class FinanceDialect(Dialect):
    name = "finance"

    @op_def
    class BlackScholesOp(IRDLOperation):
        """Black-Scholes option pricing"""
        name = "finance.black_scholes"

        spot = operand_def(f64)
        strike = operand_def(f64)
        time = operand_def(f64)
        rate = operand_def(f64)
        volatility = operand_def(f64)

        call_price = result_def(f64)
        put_price = result_def(f64)

# Optimize for numerical stability
class NumericalStabilityPass(Pass):
    """Ensure numerical stability in financial calculations"""

    def apply(self, module: builtin.ModuleOp):
        for op in module.walk():
            if isinstance(op, math.ExpOp):
                # Prevent overflow/underflow
                self.add_range_check(op)
            elif isinstance(op, arith.DivfOp):
                # Prevent division by zero
                self.add_zero_check(op)
```

Listing 28: Financial DSL Compiler

9.3 Case Study 3: Compiler for Distributed Computing

Compile programs for distributed execution:

```
@dialect_def
class DistributedDialect(Dialect):
    name = "dist"

    @op_def
    class AllReduceOp(IRDLOperation):
        """All-reduce across distributed nodes"""
        name = "dist.allreduce"

        input = operand_def(TensorType)
        output = result_def(TensorType)
        reduction = attr_def(str)  # "sum", "max", etc.

class DistributionPass(Pass):
    """Distribute computation across nodes"""

    def apply(self, module: builtin.ModuleOp):
        # Analyze data dependencies
        deps = self.analyze_dependencies(module)

        # Partition computation
        partitions = self.partition_computation(module, deps)

        # Insert communication operations
        for partition in partitions:
            self.insert_communication_ops(partition)

        # Generate distributed code
        return self.generate_distributed_code(partitions)
```

Listing 29: Distributed Computing Compiler

10 Conclusion: Your Journey Forward

10.1 What You've Learned

Congratulations! You now understand:

- How compilers represent programs as structured data (IR)
- The role of operations, attributes, and types
- How to write analysis passes to understand code
- How to implement transformations that optimize programs
- The connection between compiler techniques and ML optimization

10.2 Next Steps

Your compiler journey doesn't end here:

1. **Explore Real Compilers**: Study LLVM, GCC, or V8
2. **Contribute to xDSL**: Add new dialects or optimizations
3. **Apply to Your Domain**: Build domain-specific compilers
4. **Learn Advanced Topics**:

- Polyhedral compilation
- Verified compilation
- Quantum compilation
- Neuromorphic compilation

10.3 Resources for Further Learning

- **xDSL Documentation**: `https://xdsl.dev`
- **MLIR Tutorial**: `https://mlir.llvm.org/docs/Tutorials/`
- **Engineering a Compiler** (Cooper & Torczon)
- **Modern Compiler Implementation** (Appel)
- **The SSA Book**: `http://ssabook.gforge.inria.fr/`

10.4 Final Thoughts

Compilers are the bridge between human intent and machine execution. As AI/ML systems become more complex and heterogeneous hardware becomes more prevalent, understanding compilation becomes crucial. The techniques you've learned - representation, analysis, and transformation - are fundamental to building efficient, scalable systems.

Remember: Every optimization you write makes programs faster, every analysis you implement makes systems more reliable, and every transformation you create pushes the boundaries of what's computationally possible.

Happy compiling!

A Appendix A: xDSL Quick Reference

A.1 Common Operations

```
# Arithmetic
arith.Constant.from_int_and_width(42, 32)  # Integer constant
arith.Constant.from_float(3.14, f32)       # Float constant
arith.Addi(a, b)  # Integer addition
arith.Muli(a, b)  # Integer multiplication
arith.Addf(x, y)  # Float addition
arith.Mulf(x, y)  # Float multiplication

# Memory
memref.Alloc.get(shape=[10, 20], element_type=f32)
memref.Load(mem, indices)
memref.Store(value, mem, indices)

# Control Flow
scf.For(start, end, step, init_values)
scf.If(condition, has_else=True)
scf.While(init_values)

# Functions
func.FuncOp(name, func_type, body)
func.Call(func_name, args)
func.Return(values)
```

Listing 30: xDSL Operations Quick Reference

A.2 Common Patterns

```python
# Walking operations
for op in module.walk():
    process(op)

# Pattern matching
if isinstance(op, arith.Addi):
    handle_add(op)

# Building IR
with Builder.implicit_region():
    result = arith.Addi(a, b).result

# Replacing operations
old_op.results[0].replace_by(new_value)
old_op.erase()
```

Listing 31: Common xDSL Patterns

B Appendix B: Setting Up Your Environment

B.1 Installation

```bash
# Install xDSL
pip install xdsl

# For development
git clone https://github.com/xdslproject/xdsl.git
cd xdsl
pip install -e ".[dev]"

# Run tests
pytest tests/
```

Listing 32: Installing xDSL

B.2 Your First xDSL Program

```python
#!/usr/bin/env python3

from xdsl.context import Context
from xdsl.dialects import builtin, arith, func
from xdsl.printer import Printer

# Create context and load dialects
ctx = Context()
ctx.load_dialect(builtin.Builtin)
ctx.load_dialect(arith.Arith)
ctx.load_dialect(func.Func)

# Build a simple function
func_type = func.FunctionType.from_lists([builtin.i32], [builtin.i32])

with func.FuncOp("double", func_type).body as func_body:
    arg = func_body.block.args[0]
    two = arith.Constant.from_int_and_width(2, 32)
    result = arith.Muli(arg, two)
    func.Return(result)

# Create module
module = builtin.ModuleOp([func_body.owner])

# Print the IR
printer = Printer()
printer.print_op(module)
```

Listing 33: hello$_{c}$*ompiler.py*

C Appendix C: Glossary

AST Abstract Syntax Tree - Tree representation of source code structure

Basic Block Sequence of operations with single entry and exit

CFG Control Flow Graph - Graph showing possible execution paths

Dataflow Analysis tracking how data moves through program

Dialect Collection of related operations and types in xDSL

Dominance Relationship where one operation must execute before another

IR Intermediate Representation - Compiler's internal program representation

Lowering Transforming high-level operations to simpler ones

Operation Fundamental unit of computation in xDSL

Operand Input value to an operation

Pass Transformation or analysis applied to IR

Pattern Template for matching and replacing IR structures

Region Container for blocks in xDSL (e.g., function body, loop body)

Result Output value from an operation

SSA Single Static Assignment - Form where each variable assigned once

Type Description of data format and constraints

Use-Def Chain Links between value definitions and uses

Value Data flowing between operations (operands and results)

Verification Checking that IR is well-formed and valid