MLIR

A Compiler Infrastructure for the End of Moore's Law

Chris Lattner* · Google (SiFive at pub.)
Mehdi Amini · Google
Uday Bondhugula · IISc Bangalore
Albert Cohen · Google
Andy Davis · Google
Jacques Pienaar · Google
River Riddle · Google
Tatiana Shpeisman · Google
Nicolas Vasilache · Google
Oleksandr Zinenko · Google

Presented by: Abhinandan Sharma

CSE 240D: System Design for Machine Intelligence · May 2026

LLVM: A Transformative Success — With Real Limits

Designed for CPUs in 2004 — its fixed abstractions are now structural bottlenecks.

✓ What LLVM Got Right

Single SSA-based IR

One set of passes (DCE, CSE, inlining) works across C, Rust, Swift, Fortran

Mature backend targets

x86, ARM, RISC-V, WebAssembly — decades of production code generation

Modular pass pipeline

Clean function and module pass manager — straightforward to extend

Massive industry adoption

Clang, Swift, Rust, Julia, NVIDIA CUDA, Apple — the de facto standard

✗ Where LLVM Falls Short

IR fixed at "C with vectors"

Too low-level for ML graphs, polyhedral loops, or high-level type systems

No nested regions

CFGs are flat — loop trees and closures need lossy workarounds

Whole-module use-def chains

Functions are tightly coupled — cannot be compiled in parallel

Heterogeneous targets are bolt-ons

GPU / TPU / FPGA backends added via intrinsics, not first-class citizens

The Core Problem: One IR Can't Serve Every Master

Machine Learning

The fragmentation

TF, XLA, TensorRT, Core ML, TFLite all speak different languages.

The cost

No shared infrastructure → poor errors, unpredictable perf, hard to port to new hardware.

High-Level Languages

The fragmentation

Swift, Rust, Julia, Fortran each built their own mid-level IR from scratch.

The cost

Same infrastructure reimplemented N times. No sharing, no reuse.

Heterogeneous Hardware

The fragmentation

GPU / TPU / FPGA backends bolted on through intrinsics — not first-class.

The cost

Fragile, hard to maintain, and blocks cross-target optimization.

The field needed one shared foundation — not N bespoke compilers.

What's Broken

P1	**"One size fits all" IRs are too coarse** LLVM IR ('C with vectors') and JVM ('OO with GC') offer a single fixed level. High-level analysis becomes impossible once code is lowered.
P2	**Every language reinvents its own mid-level IR** Swift (SIL), Rust (MIR), Julia IR, Fortran — each rebuilt the same infrastructure from scratch. No sharing, no reuse.
P3	**High cost, low quality** Building bespoke IRs is expensive and quality suffers — slow compile times, buggy implementations, poor diagnostics.
P4	**ML stacks are deeply fragmented** TensorFlow alone spans XLA, TensorRT, Core ML, TF Lite, TPU IR, LLVM IR — none sharing infrastructure or design principles.
P5	**No solution for cross-language / heterogeneous targets** Sharing OpenMP or C calling conventions across languages, or targeting custom accelerators, had no clean solution.

2

Design Principles

Seven guiding ideas behind MLIR's architecture

Seven Design Principles at a Glance

1 · Multi-Level Abstraction

- Dialects as custom vocabularies
- High-level semantics preserved
- No forced early lowering

2 · SSA + Nested Regions

- First-class structural containers
- Loops as physical parents
- No CFG spaghetti

3 · Progressive Lowering

- Incremental step-by-step descent
- Per-operation granularity
- Shatters phase ordering

Seven Design Principles (continued)

4 · Heterogeneous Targets

- Custom hardware dialects
- GPU / TPU / NPU first-class
- Silicon primitives explicit

5 · Parallel Compilation

- Isolated-from-above regions
- Broken global use-def webs
- Multi-core safe by design

6 & 7 · Declarative Rewrites + Traceability

- TableGen DRR vs ad-hoc C++
- Paper trail on every Op
- WYSINWYX prevention

What Is a Dialect? (Deep Dive)

- A Dialect is a custom vocabulary — a namespace grouping Ops, types & attributes at one abstraction tier
- Represents code at different levels: TF-Graph → Loop → Vector → Hardware instruction
- **Key breakthrough: dialects from different tiers can live in the same file simultaneously**
    - e.g., scf.for (software loop) + dma_load (bare-metal) share one block with zero wrappers
- Under the hood every dialect Op is the same mlir::Operation C++ base class — dialects are a namespace illusion
    - Variables between dialect Ops are mlir::Value pointers — zero-copy, zero-conversion data flow

🦻 *Clarification: A dialect is not a compiler stage — it is a vocabulary that can mix freely with any other dialect at any time.*

Nested Regions vs. Flat CFG (Deep Dive)

❌ Flat CFG (LLVM)

- Loops are an illusion of branch arrows
- break/continue/exceptions create spaghetti paths that dissolve boundaries
- Mutation passes split blocks — hierarchy must be recomputed from scratch repeatedly
- Heavy pre-analysis required before every loop transformation
- No native nesting = no safe parallel compilation across regions

✅ Nested Regions (MLIR)

- Loops are physical containers — Russian nesting dolls in memory
- Hierarchy is a literal tree structure, never breaks under mutations
- Outer-scope values visible inside via simple lexical nesting rules
- isolated-from-above regions enable safe multi-core compilation
- Structured control flow preserved until explicitly lowered to CFG

3

IR Design Details

Operations, Attributes, Regions, Dialects & the Type System

Operations (Ops) — The Universal Semantic Unit

- **Everything in MLIR is an Op: instruction, function, module — uniform architecture**
- No fixed opcode set — users define custom Ops; passes treat unknowns conservatively (never crash)
- Ops carry: opcode | operands & results (SSA) | attributes | regions | location info
- Semantic Traits: tag Ops with properties like NoSideEffect, SameOperandsAndResultType
  - Generic text form: "dialect.opname"(%arg0, %arg1) {attr="val"} : (type) -> type
- Value bundles: %-identifiers can be packs; : gives count, # extracts one value from pack
- Optimization interfaces let passes query Ops for inlining legality, cost models, etc.

🦻 *Passes can query any Op through traits & interfaces — no hardcoded instruction lists needed.*

Attributes · Location Info · Symbols

Attributes

- Typed compile-time static constants
- Per-Op open key-value dict
- Placed in {braces} between operands and type
- used to store loop bounds, tensor shapes, or even massive arrays of raw binary weights

Location Info

- Every Op carries loc(...) metadata
- Preserves original source stack trace
- Backs debug info & diagnostic output
- Extensible: AST nodes, DWARF, file:line
- Enables WYSINWYX prevention

Symbols & Symbol Tables

- String-keyed names (@func, @global)
- Can be referenced before definition
- Enables recursive functions
- Acts as a string reference rather than a direct dataflow pointer - safely referenced anywhere in the module
- Ops can attach nested symbol tables
- Global state without SSA tangles

Regions & Blocks — Structural Nesting

- **Hierarchy: Op → (list of) Regions → (list of) Blocks → (list of) Ops [recursive]**
- Single-block region: clean bounded scope for loop bodies (affine.for, scf.for)
- Multi-block region: forms a local CFG with terminator Ops (cf.cond_br, switch, return)
- **Functional SSA — No Phi (ϕ) Nodes**
  - Instead: blocks declare typed Block Arguments; terminators pass values to successor blocks explicitly
  - e.g.: cf.br ^exit(%result : i32) — data handed off like a function call argument
- Entry block argument of affine.for = the loop induction variable (%arg4)
- Value dominance: outer-region values visible inside child regions via lexical nesting

🦻 *Phi nodes encode implicit data flow; block arguments make it explicit — easier to analyze and transform.*

Dialects & Type System

- Dialect = logical namespace for related Ops, types & attributes (prefix: affine., tf., fir.)
  - Prevents name collisions; encourages modular reuse across compilation layers
- Multiple dialects coexist in the same block → enables progressive lowering without forced conversion
- **Type System rules:**
  - Every value has exactly one type; strict equality checking — zero implicit conversions
  - User-extensible: can reference foreign types (llvm::Type, clang::Type)
  - Standard built-ins: arbitrary-precision integers, floats, tuples, tensors, memref (ND arrays)
- Functions & Modules are just Ops in the builtin dialect — no special hardcoded status

Strict typing + no implicit conversions = bugs surface at IR validation time, not silent runtime errors.

How Cross-Dialect Interaction Works (Deep Dive)

- Step 1 — Declare via TableGen (.td): define namespace, Ops (ins/outs/traits), auto-generate C++ with mlir-tblgen
- Step 2 — Register: load custom dialect alongside standard dialects in the MLIRContext driver
- **Step 3 — Seamless coexistence example:**
  - %sum = arith.addi %a, %b : i32 ← standard arithmetic dialect
  - %safe = secure.obfuscate %sum : i32 ← custom crypto dialect consumes same value
- Zero-overhead pipe: mlir::Value is a direct C++ pointer — no copies, no wrappers
- Type uniquing: MLIRContext singletons allow pointer-equality type checks across all dialects
- Dialects are a namespace illusion — all Ops share one mlir::Operation base class at runtime

🦻 *The dialect boundary exists only for humans; the compiler sees a single uniform Op tree.*

4 IR Infrastructure

ODS, DRR, Pass Manager, Textual IR & Verifiers

Operation Definition Spec (ODS) — Declarative Ops

- Problem: writing C++ parsing/printing/verification boilerplate for every Op is slow & error-prone
- **ODS is a DSL (Domain Specific Language) embedded in TableGen (.td) — declare the blueprint, generate the C++**
- Each Op definition specifies:
  - name · trait list · named arguments (ins) with type constraints · named results (outs)
  - Human-readable summary & description → auto-generates dialect documentation
- mlir-tblgen compiles .td → C++ classes with named accessors (op.getAlpha()), verifiers, parsers
- Escape hatch: inject raw C++ into builder/printer/verifier clauses when ODS is too limiting
- Type constraints are extensible: 'tensor with float element', 'vector of rank 2', etc.

🦻 *Same .td file drives both generated C++ code and documentation — single source of truth.*

Declarative Rewrite Rules (DRR) — Transformations as Patterns

- **Treats code is a DAG (Directed Acyclic Graph) of SSA use-def edges — perfect for pattern matching**
  - DAG = nodes (Ops) + directed arrows (data flow) + acyclic (always flows definition → use)
- DRR: also TableGen-embedded DSL — declare source pattern + equivalent target pattern
  - Example: LeakyRelu(x, α) → Select(CmpF(x, 0), x, Const(α)) — declared in ~10 lines
- Supports dynamic constraints, benefit scores (pattern priority), multi-result replacements
- DRR compiles to C++ graph-traversal code — can mix with manual C++ rewrite patterns freely
- Subsumes LLVM InstCombine, DAGCombine, PeepholeOptimizer as a single unified extensible system

🦻 *DRR + getCanonicalizationPatterns() = one universal Canonicalization pass replaces dozens of ad-hoc passes.*

Pass Manager & Parallel Compilation

- Op-agnostic: pass manager works on any Op at any nesting depth — not locked to module/function granularity
- **Parallel Compilation Breakthrough:**
  - isolated-from-above Ops (e.g., builtin.func) are strict data firewalls
  - SSA use-def chains cannot cross isolation barriers → independent subtrees
  - Pass manager safely dispatches different functions to different CPU cores simultaneously
- No whole-module use-def chains (unlike LLVM): globals use symbol tables, constants are Ops with attributes
- Use-def tracks: each SSA definition holds a list of uses; empty list = dead code, safe to delete

🦻 *Isolated regions are not just a semantic feature — they are the mechanism that makes multi-core compilation correct.*

Round-Trippable IR · Verifiers · Documentation

Round-Trippable Textual IR

- Text = exact mirror of in-memory C++ state
- RAM → text → RAM with zero loss
- Stop compiler mid-pipeline, edit text, inject back
- Each pass testable in isolation with .mlir files
- Generic verbose form + custom pretty-print form

Verifiers (Immune System)

- Phase 1 – Structural: types match, SSA dominance (defined before use), terminators present
- Phase 2 – Semantic: per-Op rules (binary Op needs exactly 2 operands)
- Dialect attributes can restrict which Ops they attach to
- Violation = abort compilation (invariant, not warning)

Auto-Generated Documentation

- ODS .td → markdown docs automatically
- Same source → code + docs always in sync

5

Applications & Adoption

TensorFlow · Polyhedral · Fortran · Domain-Specific Compilers

Community Adoption & Growth Metrics

16

Universities
at HPC workshops

4

National labs
across 4 countries

14

Multinational
companies endorsing

100+

Industry devs
at LLVM roundtables

26+

Active dialects
in public/private repos

7

Companies replacing
custom infra with MLIR

TensorFlow Graphs — DFG vs. CFG (Deep Dive)

- Dataflow Graph (DFG): Ops execute the moment input data arrives — massive async parallelism
- **LLVM's problem: built for sequential CFG (instruction pointer model) – struggles with DFG**
  - Mapping a DFG to LLVM shatters graph structure into thread pools + mutexes → lost optimization context
- **MLIR TF dialect solution:**
  - Control Tokens (!tf.control): SSA values that carry ordering info as data flow
  - Operations output both a math result AND a control token signal
  - Feeding token as input to next Op mathematically enforces execution order in the DAG
- Result: strict control flow rules enforced without breaking the unified dataflow graph abstraction

🦻 *Control tokens turn an ordering constraint (control flow) into a data dependency — keeping the graph a single DAG.*

Polyhedral Code Generation — The Affine Dialect

- Affine dialect = simplified polyhedral IR inside MLIR's SSA framework
- **Similarities to classical polyhedral tools:**
  - memref type: injective N-D arrays — no pointer aliasing by construction. Supports layout maps.
  - affine.for / affine.if: loops & conditionals with static affine bounds as attributes
  - affine.load / affine.store enforce indexing to affine functions of loop iterators
- **Key differences vs. ISL/Pluto:**
  - Rich types: layout maps embedded in memref isolate data layout from loop math
  - Mix of abstractions: loop bodies expose typed SSA values — scalar passes interleave freely
  - Smaller representation gap: loops preserved in IR → no polyhedron scanning (eliminates $O(2^n)$ cost)
  - Production speed: graph rewrites (using DRR) replace exponential ILP solvers

Fortran IR (FIR) — HPC & First-Class Concepts

- Fortran superpower: no pointer aliasing by default → aggressive vectorization & parallelization
- flang (LLVM Fortran frontend) uses MLIR to support high-level Fortran-specific optimizations:
  - Array copy elimination: fuse A = B + C + D into one loop, compute entirely in registers
  - Advanced loop optimizations, call specialization, devirtualization
- **First-class Vtable modeling:**
  - LLVM: Vtable smashed into raw pointer arithmetic — context lost, inlining impossible
  - FIR: fir.dispatch_table is an explicit IR operation → Devirtualization Pass reads & eliminates overhead
- Dialect reuse: FIR composes with OpenMP & OpenACC dialects — shared passes across language barriers

🦻 *'First-class' = the IR natively understands the concept, so optimization passes can reason about it directly.*

Domain-Specific Compilers — Rapid Prototyping at Scale

- MLIR dramatically lowers the engineering cost of hyper-focused compilation pipelines
- **Case 1 — FSM Rewrite Optimizer (Meta-Compiler):**
  - Hardware vendors needed to add runtime-extensible optimization patterns in device drivers
  - Solution: model MLIR rewrite patterns as an MLIR dialect — compile the optimizer with MLIR itself
  - Generates highly efficient Finite State Machine matchers on the fly
- **Case 2 — Lattice Regression Compiler:**
  - Millions of users impacted; bottlenecked by monolithic C++ template metaprogramming
  - **3 engineers · 3 person-months → 8× performance improvement on production model**
  - Vastly improved compile-time transparency vs. opaque C++ templates

🦻 *MLIR enables 'compiler as a product feature' — small teams can ship domain compilers in months, not years.*

Mixing Dialects — Cross-Layer Infrastructure Reuse

- **Most profound MLIR capability: operations from different dialects coexist in the same block**
- Example pipeline in a single .mlir file:
  - affine dialect: controls loop tiling & scheduling (polyhedral math)
  - arith dialect: standard scalar arithmetic inside the loop body
  - nvidia_gpu dialect: hardware acceleration calls in the innermost loop
- Benefit: generic polyhedral tiling pass can tile & parallelize loops whose bodies are completely proprietary hardware Ops
- OpenMP dialect reused across both Fortran (flang) and C (Clang) frontends — shared optimization passes
- **This is a class of reuse not seen in any prior compiler system**

🦻 *Traditional: convert entire program to one representation, then optimize. MLIR: optimize at each level simultaneously.*

Open Challenges & Future Horizons

- **The Unopinionated Design Challenge:**
  - MLIR removes all guardrails — 'art' of abstraction design (Op vs. Attribute vs. Type?) is not yet codified
  - Field must develop best practices for composable, non-conflicting abstractions
- **Active research frontiers:**
  - Out-of-tree dialects: managing explosion of third-party proprietary/open-source dialects
  - Replacing front-end ASTs: MLIR dialects inside language frontends for early semantic optimization
  - Non-code data: MLIR dialects for JSON / Protocol Buffer parsing, validation, routing pipelines
- Compiler education: MLIR's textual IR as a teaching tool for IR design — largely unexplored

MLIR — Key Takeaways

One IR, Many Levels

Dialects let high-level ML graphs and bare-metal instructions coexist in the same file.

Structure Preserved

Nested Regions keep loop hierarchy as physical containers — not a CFG illusion.

Declare, Don't Repeat

ODS + DRR replace thousands of lines of C++ boilerplate with declarative specifications.

Parallel by Design

isolated-from-above regions make multi-core compilation correct by construction.

Reuse Without Coupling

Traits, hooks & interfaces let generic passes work across arbitrary dialects.

Trace Everything

Every Op carries a provenance paper trail — WYSINWYX becomes auditable.

Lattner et al., arXiv:2002.11054 · MLIR: A Compiler Infrastructure for the End of Moore's Law

Thank you for your attention!