MLIR: Building Better Compilers for Modern Computing

MLIR is a new way to build compilers that solves a fundamental problem: modern computing needs compilers that can work at many different levels of abstraction, from high-level machine learning operations down to hardware-specific instructions, but traditional compilers like LLVM only work well at one level (close to machine code).[1] [2]

The Core Problem

Traditional compiler infrastructure forces developers to choose between working at a high level (where you can reason about complex operations) or a low level (where you can optimize for specific hardware). When you have diverse hardware like CPUs, GPUs, and specialized AI accelerators, this creates **fragmentation**—every team builds their own custom compiler from scratch, wasting effort and making it hard for different tools to work together.[3] [4] [1]

The MLIR Solution: Dialects and Progressive Lowering

MLIR introduces **dialects**, which are like specialized vocabularies for different levels of abstraction or computing domains. Think of dialects as LEGO sets—each set has pieces designed for a specific purpose (space, castles, city buildings), but all the pieces can connect together. A machine learning dialect can represent neural network operations, while a polyhedral dialect handles loop optimizations, and an LLVM dialect connects to traditional low-level code generation.[2] [5] [6] [7] [8] [9]

The key innovation is **progressive lowering**—you gradually translate from high-level representations to low-level ones in steps, keeping useful information as long as possible. For example, a tensor operation might lower to loop-based code, then to vectorized instructions, then to hardware-specific assembly, with optimization opportunities at each stage.[5] [9] [10] [2]

Why This Matters

MLIR dramatically reduces the cost of building domain-specific compilers because you reuse infrastructure instead of building everything from scratch. Multiple dialects can coexist in the same program, allowing operations at different abstraction levels to work together seamlessly. This is especially valuable for heterogeneous computing where a single program might run parts on CPUs, GPUs, and specialized accelerators.[6] [9] [1] [3] [5]

The infrastructure is extensible and modular—you define custom operations, types, and transformations for your specific domain using declarative specifications rather than hardcoding everything. This makes MLIR both more flexible than traditional compilers and easier to extend for new use cases.[4] [11] [1] [5]

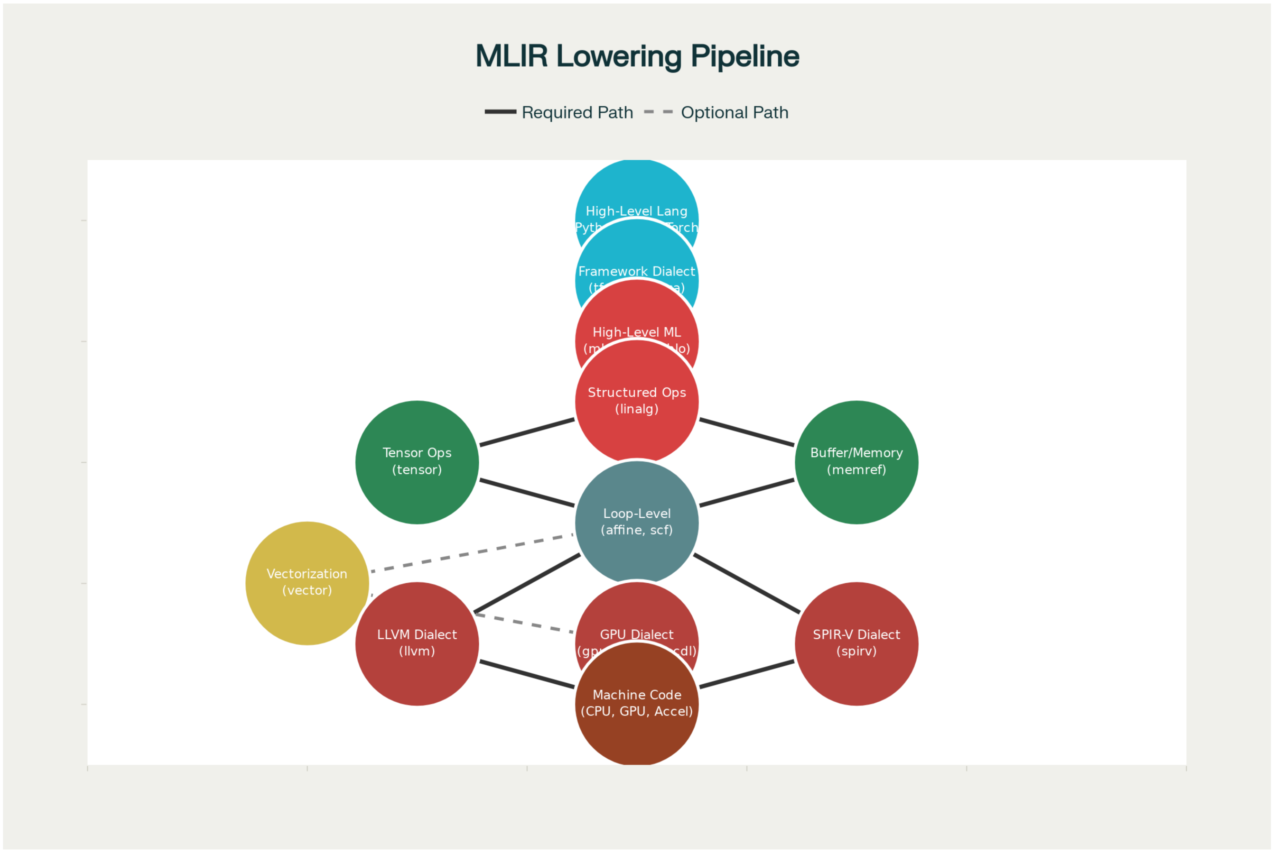
Real-World Impact

Major projects like TensorFlow, XLA, and IREE use MLIR as their compiler foundation, demonstrating its effectiveness for machine learning workloads. The Mojo programming language also builds on MLIR infrastructure. By providing a common intermediate representation that works across abstraction levels, MLIR enables better compiler interoperability and reduces duplicated engineering effort across the industry.[8] [12] [1] [3] [2]

⁂

MLIR Lowering Pipeline: From High-Level Code to Machine Instructions

The transformation from high-level programming languages to machine code in MLIR follows a tree-like structure with multiple branches and convergence points, where each level represents a progressive reduction in abstraction.[22] [23]



MLIR Progressive Lowering Pipeline: The transformation path from high-level programming languages through various dialect levels down to machine code

Level 1: Framework-Specific Entry Point

When you write code in Python, TensorFlow, or PyTorch, it first gets converted into a **framework-specific dialect** that closely mirrors the original operations. For example, a TensorFlow convolution operation becomes `tf.Conv2D`, preserving the high-level semantics of the source language. This allows MLIR to perform framework-specific optimizations while maintaining familiarity with the original code structure.[24] [23] [25] [26] [27]

Level 2: High-Level ML Operations

The framework dialect lowers to **high-level ML dialects** like MHLO (Meta High-Level Optimizer) or StableHLO, which provide a standardized representation of machine learning operations independent of any specific framework. This convergence point allows different frontends (TensorFlow, PyTorch, ONNX) to share the same subsequent optimization passes.[22] [23] [28]

Level 3: Structured Operations

Operations then lower to the **Linalg dialect**, which represents computations as structured mathematical operations on multi-dimensional arrays. Linalg expresses operations like matrix multiplications and convolutions in terms of their mathematical structure, enabling powerful loop-level transformations and optimizations.[24] [22] [23] [29]

Level 4: The Bufferization Split

At this point, the pipeline **branches** into two parallel paths :[22] [23]

Tensor path: Operations remain in the tensor dialect, working with immutable, value-semantic tensors that are easier to reason about for high-level transformations.[23] [26]

Memory path: Through a process called bufferization, tensors convert to the memref (memory reference) dialect, which represents explicit memory buffers with addresses and layouts. This is necessary for actual code generation since hardware works with memory, not abstract tensors.[30] [24] [22] [23]

Level 5: Loop-Level Representations

Both paths **converge** at loop-level dialects like Affine (for perfectly nested loops with compile-time analyzable bounds) or SCF (Structured Control Flow for general loops). At this level, a matrix multiplication becomes explicit nested loops iterating over array indices. This is where polyhedral optimizations like loop fusion, tiling, and parallelization happen.[24] [22] [23] [31]

Level 6: Optional Vectorization

The pipeline can optionally branch through the **vector dialect**, which represents SIMD (Single Instruction, Multiple Data) operations that process multiple values simultaneously. This step is crucial for leveraging modern CPU vector instructions (AVX, NEON) or GPU parallelism.[22] [23] [32]

Level 7: Backend-Specific Targets

The pipeline **diverges** into multiple backend-specific dialects depending on the target hardware :[23] [31]

LLVM dialect: For CPU code generation, operations lower to LLVM IR, which LLVM's backend can compile to x86, ARM, or other CPU architectures.[30] [31]

GPU dialects: For GPU targets, code lowers to gpu dialect, then further to NVVM (NVIDIA) or ROCDL (AMD) for specific GPU architectures.[22] [23]

SPIR-V dialect: For cross-platform GPU/accelerator code using the Vulkan or OpenCL ecosystems.[22] [23]

Level 8: Machine Code Generation

Finally, all paths **converge** at actual machine code—CPU instructions, GPU kernels, or accelerator-specific binaries that the hardware can execute. The LLVM backend handles final register allocation, instruction selection, and machine-specific optimizations.[30] [23] [31]

Key Characteristics of This Tree Structure

The tree isn't strictly linear—dialects can **coexist** in the same program, with some operations at higher levels while others have already lowered. The order of transformations can vary; vectorization might happen before or after loop generation depending on the optimization strategy. This flexibility allows MLIR to choose the most efficient lowering path for each specific use case while maintaining correctness guarantees through the dialect conversion framework. [24] [33] [23] [29] [26] [32]

⁂

Polyhedral Code Generation: Turning Math into Loops

Polyhedral code generation is the process of converting a mathematical representation of loops (as geometric shapes) back into actual executable code with for-loops, bounds, and conditionals. Think of it like having architectural blueprints (the polyhedron) and needing to give construction workers step-by-step instructions (the actual code).[42] [43] [44] [45]

The Mathematical Representation

In the polyhedral model, loop iterations are represented as **integer points inside a polyhedron** —a multidimensional geometric shape defined by linear inequalities. For example, a nested loop that iterates `i` from 0 to n and `j` from 1 to `i` forms a triangular region when visualized, where each point `(i, j)` is one execution of the loop body. This is expressed mathematically as $ \mathcal{D} = { x \in \mathbb{Z}^n \mid Ax + b \geq \mathbf{0} } $, where the matrix A and vector b encode all the loop bounds and constraints.[43] [44] [46] [45]

The Code Generation Algorithm

The core algorithm works by **progressively scanning the polyhedron dimension by dimension**, much like how you'd read a book—left to right, then move down a line, then turn the page. Here's the step-by-step process :[47] [42] [43]

Projection: For the outermost dimension, project the polyhedron onto that axis to find the range of valid values. In the triangle example, projecting onto the `i` axis gives you the range from 0 to n.[47] [43]

Loop generation: Create a for-loop with bounds determined by the projection. Constraints that only involve the loop variable become the lower and upper bounds, while constraints independent of it become if-statements outside the loop.[43] [47]

Recursion: Fix the current dimension and repeat the process for inner dimensions. When you fix `i`, the valid range for `j` becomes 1 to `i`, which generates the inner loop.[47] [43]

Separation: When multiple statements exist, compute which polyhedra overlap and generate code that interleaves their execution in the correct order. This involves computing set differences and intersections to create disjoint regions.[44] [43] [47]

Handling Transformations

The power of polyhedral code generation is that **loop transformations are just geometric transformations** of the polyhedron. Loop tiling becomes slicing the polyhedron into smaller blocks, loop interchange becomes rotating the coordinate system, and loop fusion combines multiple polyhedra. After applying these transformations mathematically, you run the code generation algorithm on the transformed polyhedron to produce optimized code.[48] [45] [49] [44] [43]

MLIR's Affine Dialect Implementation

In MLIR, the **affine dialect** implements polyhedral concepts with restrictions that loops must have bounds expressible as affine functions (linear combinations) of surrounding loop variables and parameters. The `affine.for` operation represents perfectly nested loops, and `affine.if` represents conditions that are affine expressions. MLIR provides optimization passes like `affine-loop-tile`, `affine-loop-fusion`, and `affine-parallelize` that perform polyhedral transformations, then automatically generate code from the transformed representation.[50] [51] [52] [48]

The Challenge: Redundancy Elimination

Raw code generation often produces **redundant guards and conditions**—unnecessary if-statements that are always true given the surrounding loop context. For example, after generating nested loops, there might be a condition checking `m < n` inside an inner loop, even though the outer loop already guarantees this. A critical optimization pass called **redundancy elimination** simplifies the code by using contextual information from enclosing loops to remove these unnecessary checks.[43] [47]

Why This Matters

Polyhedral code generation enables **automatic parallelization and locality optimization** because the geometric representation makes dependences and reuse patterns explicit. Once you know which iterations touch which array elements (also represented as affine functions), you can safely reorder, tile, or parallelize loops while preserving correctness. The generated code leverages cache hierarchies and vector instructions without manual tuning.[45] [53] [49] [54] [48]

⁂

General-Purpose Languages Targeting MLIR

Beyond ML frameworks, several **general-purpose** and **domain-specific languages** compile to MLIR, leveraging its multi-level infrastructure for performance and hardware targeting.[62] [63]

Mojo: Python-Compatible Systems Language

Mojo is the most prominent general-purpose language built entirely on MLIR. Designed to combine Python's usability with C++-level performance, Mojo acts as "syntactic sugar for MLIR," meaning its syntax directly maps to MLIR operations. The language compiles to MLIR Core using a custom compiler framework called KGEN, which enables metaprogramming and explicit parametric types before instantiation. Mojo can target CPUs, GPUs, TPUs, and ASICs through MLIR's multi-backend infrastructure, making it particularly suitable for AI and accelerator programming.[64] [65] [66]

Julia: Emerging MLIR Frontend

Julia has active research into using it as an MLIR frontend through extensible compiler integration. Recent work (2025) demonstrates a framework where Julia's multiple dispatch and compiler hooks generate MLIR code dynamically. Package developers can define intrinsic functions that map Julia operations to MLIR dialect operations—for example, mapping `+(a::f32, b::f32)` to the `arith.addf` operation. This approach supports developing domain-specific languages within Julia that emit MLIR for GPU programming, hardware synthesis, and custom transformations.[67] [68] [69] [70]

C/C++ via Translation Paths

C and C++ can reach MLIR through indirect compilation paths. The standard approach compiles C/C++ to LLVM IR using Clang, then imports it into MLIR using `mlir-translate --import-llvm`. While this loses high-level semantic information, it allows existing C/C++ codebases to enter the MLIR ecosystem. A CIL (C Intermediate Language) dialect was mentioned in 2020 for direct C++ compilation, though public availability remains limited.[71] [72]

Domain-Specific Languages

The MLIR ecosystem hosts numerous **DSLs** targeting specific problem domains :[63] [62]

Hardware description: Languages for hardware synthesis and FPGA programming compile to MLIR dialects like CIRCT (Circuit IR Compilers and Tools).[70] [62]

Quantum computing: Quantum programming languages use MLIR's extensibility to represent quantum circuits and operations.[62] [63]

Probabilistic programming: Languages for Bayesian inference and statistical modeling explore MLIR as a compilation target.[73] [62]

The Toy Language Tutorial

MLIR's official documentation includes **Toy**, a simple tensor-based language used to teach MLIR concepts. While not a production language, Toy demonstrates how to define custom dialects, implement optimizations, and progressively lower through MLIR's infrastructure. It serves as a template for building new language frontends.[74] [75] [76]

Why Languages Target MLIR

Languages choose MLIR over directly targeting LLVM because MLIR provides **higher-level abstractions** that preserve semantic information longer in the compilation pipeline. This enables domain-specific optimizations (loop transformations, kernel fusion, memory layout) that would be difficult or impossible at LLVM's low level. Additionally, MLIR's dialect system allows languages to gradually lower through domain-appropriate intermediate representations, accessing optimizations tailored to each abstraction level.[66] [64] [63]

⁂

Key Advantages of MLIR

MLIR provides several fundamental benefits that address limitations in traditional compiler infrastructure, particularly for modern computing workloads involving AI and heterogeneous hardware.[83] [84]

Multi-Level Abstraction Preservation

MLIR's **dialect system** allows code to exist at multiple abstraction levels simultaneously, preserving high-level semantic information throughout compilation. Traditional compilers like LLVM force immediate lowering to near-machine-level IR, which destroys information about tensors, loop nests, and domain-specific operations that could enable powerful optimizations. By maintaining representations from high-level ML operations down to hardware instructions in a unified framework, MLIR enables optimizations at each appropriate level—tensor algebra optimizations at the top, polyhedral loop transformations in the middle, and register allocation at the bottom.[85] [86] [87] [88]

Heterogeneous Hardware Support

MLIR excels at **compilation for diverse hardware targets**, including CPUs, GPUs, TPUs, FPGAs, and custom accelerators. The infrastructure provides native support for representing parallel constructs, memory hierarchies, and hardware-specific operations through specialized dialects. A single MLIR program can contain operations targeting different backends, which the compiler progressively lowers to appropriate hardware-specific dialects (NVVM for NVIDIA GPUs, ROCDL for AMD, ARM NEON for ARM processors). This contrasts sharply with LLVM, which was designed primarily for CPUs and requires awkward workarounds for accelerators.[86] [89] [90] [91] [85] [83]

Reduced Compiler Development Cost

MLIR dramatically **lowers the barrier to building domain-specific compilers** by providing reusable infrastructure components. Instead of writing entire compiler stacks from scratch (as teams did with XLA for TensorFlow, Glow for PyTorch, etc.), developers define custom dialects and transformations that integrate with existing MLIR passes. The declarative specification of operations, types, and constraints through TableGen reduces boilerplate code and enables automatic generation of parsers, printers, and verifiers. This makes it feasible for small teams to create production-quality compilers for specialized domains like quantum computing, cryptography, or hardware synthesis.[92] [88] [93] [94] [95] [96] [86] [83]

Compiler Interoperability

MLIR **connects existing compilers together** rather than replacing them. Since multiple dialects coexist in the same IR, you can import code from different sources—TensorFlow graphs, PyTorch operations, hand-written C++—and compile them jointly with shared optimizations. MLIR acts as a "lingua franca" where dialects for different domains (ML frameworks, parallel programming models, hardware targets) can interact. The infrastructure also provides paths to existing backends: LLVM dialect converts to LLVM IR for CPU code generation, GPU dialects target CUDA/ROCm, and SPIR-V dialect reaches Vulkan and OpenCL.[89] [90] [94] [97] [92] [83]

First-Class Polyhedral Support

Unlike LLVM's Polly (which operates as an isolated optimization pass), MLIR **integrates polyhedral abstractions** directly into the IR through the affine dialect. Loop nests, multidimensional arrays, and affine transformations are first-class concepts that persist across compilation stages. This enables aggressive loop optimizations—tiling, fusion, interchange, parallelization—with formal correctness guarantees at higher abstraction levels where the structure is still visible. The affine dialect serves as a bridge between high-level tensor operations and low-level loop code, allowing optimizations impossible in traditional IRs.[87] [91] [98] [99] [100]

Progressive Lowering with Mixed Abstractions

MLIR supports **gradual transformation** where not all code needs to be at the same level simultaneously. You can optimize the hot path of a program at low level while keeping cold paths at high level, or lower different functions through different dialect paths based on their characteristics. This flexibility allows compilers to apply the right tool to each part of the code—polyhedral optimization for regular loops, dataflow optimization for neural network graphs, traditional CFG optimization for control-heavy code.[88] [101] [102] [85] [87]

Graph and Operator-Level Optimizations

For ML workloads, MLIR enables **high-level graph optimizations** like operator fusion, constant folding, and layout transformation before lowering to loops. Operations like "batch normalization followed by ReLU activation" can be fused into single kernels that avoid intermediate memory traffic—optimizations that are nearly impossible once code is lowered to LLVM IR. The tensor type system preserves shape information and data layout throughout compilation, enabling data-flow analyses that improve cache utilization and memory bandwidth.[91] [85] [89]

⁂

Common MLIR Exit Points

MLIR provides multiple **backend targets** where compiled code exits the MLIR ecosystem and transforms into executable machine code or intermediate formats consumable by other toolchains.[109] [110]

LLVM IR Backend (Primary CPU Path)

The **LLVM dialect** serves as the primary exit point for CPU code generation. After progressive lowering through various MLIR dialects, operations convert to the LLVM dialect, which closely mirrors LLVM IR instructions. The `mlir-translate` tool then performs a straightforward translation from LLVM dialect to actual LLVM IR bitcode, which LLVM's backend compiles to machine code for x86, ARM, RISC-V, and other CPU architectures. This two-stage flow (MLIR→LLVM dialect→LLVM IR→machine code) keeps non-trivial transformations within MLIR while leveraging LLVM's mature instruction selection and register allocation.[111] [112] [113] [114] [109]

GPU Backends

For **GPU targets**, MLIR provides several specialized exit paths :[110] [115]

NVVM dialect: Targets NVIDIA GPUs by lowering to NVVM IR (NVIDIA's LLVM-based representation), which the CUDA compiler driver (`ptxas`) compiles to PTX assembly and eventually GPU machine code.[115] [110]

ROCDL dialect: Targets AMD GPUs through ROCm's compiler stack, producing AMDGPU machine code via LLVM's AMDGPU backend.[116] [115]

GPU dialect: Provides a higher-level, vendor-neutral representation of GPU operations (kernels, memory transfers, synchronization) that can lower to either NVVM, ROCDL, or SPIR-V depending on the target.[117] [115]

SPIR-V Backend (Cross-Platform GPU/Accelerator)

The **SPIR-V dialect** exits to SPIR-V binary format, a portable intermediate representation for GPUs and accelerators. SPIR-V targets Vulkan graphics pipelines, OpenCL compute kernels, and various vendor-specific accelerators. The `mlir-translate --mlir-to-spirv` command serializes MLIR's SPIR-V dialect operations into the standardized SPIR-V binary encoding, which GPU drivers consume at runtime. This path is particularly important for cross-vendor portability where code needs to run on Intel, AMD, NVIDIA, ARM Mali, or Qualcomm Adreno GPUs.[118] [119] [120] [110]

Native Machine Code Generation

Experimental work explores **direct code generation** from MLIR to machine code without going through LLVM IR. This includes JIT-focused x86 code generators that emit assembly or object code directly from MLIR dialects. The MIR (Machine IR) dialect wraps LLVM's Machine Code layer to represent target-specific assembly instructions as MLIR operations, enabling direct lowering to x86, ARM, or other architectures. While not yet production-ready, this approach reduces compilation latency for JIT scenarios and provides finer control over instruction selection.[121] [122] [111]

Hardware Synthesis Backends

For **FPGA and ASIC targets**, MLIR exits through hardware description formats :[123] [116]

CIRCT (Circuit IR): A collection of MLIR dialects for hardware design that compile to Verilog or VHDL for FPGA synthesis.[124] [123]

Vendor-specific flows: Custom dialects for TPUs, systolic arrays, or domain-specific accelerators that generate firmware or configuration bitstreams.[125] [116]

C/C++ Code Generation

Some workflows use MLIR to **generate C or C++ source code** as an exit point, particularly for embedded systems or when interfacing with existing C++ libraries. The EmitC dialect produces C code from MLIR operations, allowing downstream compilation with standard C compilers. This path is useful for targeting platforms where LLVM backend support is limited or when human-readable generated code is required for debugging.[126] [122] [127]

Binary Object Files and Executables

The final exit points are **linkable object files** (`.o`, `.obj`) or **executable binaries**. After MLIR translates to LLVM IR, the LLVM backend generates object files through standard compilation flows (`llc` or JIT execution engines). For GPU code, PTX or AMDGPU object files get linked with host code to produce heterogeneous executables. JIT compilation paths can generate machine code directly into memory for immediate execution without filesystem artifacts.[128] [129] [111] [121] [109] [115] [117]

Multiple Simultaneous Exits

A distinguishing feature of MLIR is support for **heterogeneous compilation** where a single program exits through multiple backends. For example, CPU-targeted code lowers through LLVM dialect while GPU kernels exit via NVVM or SPIR-V, with coordination operations managing data transfer and synchronization between execution contexts. This enables efficient compilation for distributed heterogeneous systems where different code regions target CPUs, GPUs, TPUs, and FPGAs simultaneously.[130] [116] [117]

⁂

Common Entry Points to MLIR

MLIR accepts code from multiple sources and frameworks, providing various **frontend entry points** that convert high-level programming languages and frameworks into MLIR dialects.[137] [138]

Machine Learning Framework Frontends

The most common entry points come from **ML frameworks** converting their computational graphs to MLIR :[139] [140]

TensorFlow: TensorFlow graphs enter MLIR through the TensorFlow (tf) dialect, which represents TensorFlow operations directly in MLIR. The TensorFlow compiler infrastructure uses MLIR as its core, converting tf.Graph objects into tf dialect operations.[138] [141]

PyTorch/Torch: The Torch-MLIR project provides first-class support for converting PyTorch models to MLIR. Two primary paths exist: through TorchScript (capturing eager execution) and via FX graphs (capturing computation graphs), both lowering to the torch dialect.[142] [139] [138]

JAX: JAX computations enter through StableHLO or MHLO dialects, representing high-level array operations and transformations.[141] [143]

ONNX: The ONNX-MLIR project provides an entry point for ONNX (Open Neural Network Exchange) models, converting standardized ML model formats into MLIR dialects. This enables interoperability between different ML frameworks.[144] [139]

Programming Language Frontends

Several **general-purpose languages** have direct entry points to MLIR :[138] [142]

Fortran (Flang): The LLVM Fortran compiler (Flang) uses FIR (Fortran IR) as its entry point to MLIR. The compiler parses Fortran source, builds a parse tree and symbol table, then lowers to FIR dialect before transforming through MLIR passes.[145] [137] [138]

Mojo: Mojo source code compiles directly to MLIR through the KGEN compiler framework, treating Mojo as syntactic sugar over MLIR operations.[146] [147]

Julia: Experimental Julia frontends leverage Julia's compiler hooks to generate MLIR dialects dynamically, creating an entry point for Julia programs.[148] [142]

C/C++: While less direct, C/C++ can enter MLIR by first compiling to LLVM IR with Clang, then using `mlir-translate --import-llvm` to convert LLVM IR into MLIR's LLVM dialect.[149] [150]

Domain-Specific Language Entry Points

Hardware description languages: Enter through CIRCT dialects for circuit representation and hardware synthesis.[151] [138]

Quantum languages: Quantum computing frameworks have custom dialects as entry points for quantum circuit representation.[152] [138]

Linear algebra DSLs: Mathematical computation languages enter through the Linalg dialect, representing structured operations on arrays.[143] [141]

Direct MLIR Text and Bytecode

Textual MLIR: Hand-written or tool-generated MLIR code in textual format (`.mlir` files) can be directly loaded via `mlir-opt`. This is the most direct entry point, commonly used for testing, debugging, and educational purposes.[153] [154] [155]

Bytecode format: MLIR supports a binary bytecode format for efficient serialization and deserialization, providing a compact entry point for stored or transmitted MLIR modules.[156] [153]

API-Based Entry Points

Builder APIs: Programs can programmatically construct MLIR using C++ or Python builder APIs, directly creating operations, types, and attributes in memory without parsing textual IR. This is common when embedding MLIR in larger compiler frameworks or JIT systems.[155] [137]

Custom dialect parsers: Projects define custom assembly formats for their dialects, allowing domain-specific syntax to serve as an entry point that parses directly into MLIR operations.[156] [137]

The flexibility of these entry points reflects MLIR's design as a **unifying compiler infrastructure** that connects diverse programming models, frameworks, and domains into a common compilation pipeline.[157] [137]

⁂

1. https://arxiv.org/abs/2002.11054
2. https://www.stephendiehl.com/posts/mlir_introduction/
3. https://www.modular.com/blog/democratizing-ai-compute-part-8-what-about-the-mlir-compiler-infrastructure
4. https://www.cs.cornell.edu/courses/cs6120/2023fa/blog/mlir/
5. https://www.emergentmind.com/topics/mlir-based-compiler
6. https://docs.pennylane.ai/projects/catalyst/en/stable/dev/dialects.html
7. https://www.answeroverflow.com/m/1208151797575721001
8. https://en.wikipedia.org/wiki/MLIR_(software)
9. https://users.cs.utah.edu/~mhall/mlir4hpc/pienaar-MLIR-Tutorial.pdf
10. https://mlir.llvm.org/docs/Tutorials/Toy/Ch-5/
11. https://llvm.org/devmtg/2019-04/slides/Keynote-ShpeismanLattner-MLIR.pdf
12. https://www.youtube.com/shorts/8QJsJZMbBNo
13. https://mlir.llvm.org
14. https://llvm.org/devmtg/2020-09/slides/MLIR_Tutorial.pdf
15. https://news.ycombinator.com/item?id=35791960
16. https://rcs.uwaterloo.ca/~ali/cs842-s23/papers/mlir.pdf

17. https://mlir.llvm.org/docs/Tutorials/Toy/Ch-4/
18. https://www.lei.chat/posts/compilers-and-irs-llvm-ir-spirv-and-mlir/
19. https://arxiv.org/pdf/2002.11054.pdf
20. https://discourse.julialang.org/t/should-julia-use-mlir-in-the-future/110459
21. https://www.cs.cornell.edu/courses/cs6120/2025sp/blog/bril-mlir/
22. https://www.lei.chat/posts/mlir-codegen-dialects-for-machine-learning-compilers/
23. https://apxml.com/courses/compiler-runtime-optimization-ml/chapter-2-advanced-ml-intermediate-representations/mlir-lowering-paths
24. https://mlir.llvm.org/docs/Tutorials/Toy/Ch-5/
25. https://mlir.llvm.org/docs/Tutorials/Toy/Ch-3/
26. https://www.jeremykun.com/2023/10/23/mlir-dialect-conversion/
27. http://lastweek.io/notes/MLIR/
28. https://docs.pennylane.ai/projects/catalyst/en/stable/dev/dialects.html
29. https://www.jeremykun.com/2023/08/10/mlir-running-and-testing-a-lowering/
30. https://mlir.llvm.org/docs/Tutorials/Toy/Ch-6/
31. https://www.jeremykun.com/2023/11/01/mlir-lowering-through-llvm/
32. https://mlir.llvm.org/docs/Dialects/Vector/
33. https://mlir.llvm.org/docs/DialectConversion/
34. https://circt.llvm.org/docs/Dialects/Pipeline/
35. https://repo.fib.upc.edu/damaso-pierre.de.la.cruz/LiftingLoops/-/blob/main/mlir/docs/Tutorials/Toy/Ch-6.md
36. https://www.stephendiehl.com/posts/mlir_egraphs/
37. https://www.reddit.com/r/Compilers/comments/1crvnpt/is_there_any_tool_that_converts_python_to_mlir/
38. https://www.arxiv.org/pdf/2409.03864v2.pdf
39. https://arxiv.org/html/2409.03864v2
40. https://rcs.uwaterloo.ca/~ali/cs842-s23/papers/mlir.pdf
41. https://www.stephendiehl.com/posts/mlir_introduction/
42. https://www.youtube.com/watch?v=C04zi95Qo5w
43. https://icps.u-strasbg.fr/~bastoul/research/papers/Bas04-PACT.pdf
44. https://www.slideshare.net/slideshow/introduction-to-polyhedral-compilation/70482946
45. https://www.cs.cornell.edu/courses/cs6120/2023fa/blog/polyhedral/
46. http://misailo.web.engr.illinois.edu/courses/526-sp17/lec15.pdf
47. https://www.youtube.com/watch?v=MJ_NjnIqM38
48. https://www.stephendiehl.com/posts/mlir_affine/
49. https://pliss2019.github.io/albert_cohen_slides.pdf
50. https://users.cs.utah.edu/~mhall/mlir4hpc/cohen-MLIR-loop-overview.pdf
51. https://llvm.org/devmtg/2021-02-28/slides/Vinay-MLIR-loopnest.pdf
52. https://mlir.llvm.org/docs/Dialects/Affine/
53. http://polyhedral.info

54. https://en.wikipedia.org/wiki/Polytope_model
55. https://xavierleroy.org/publi/polyhedral-codegen.pdf
56. https://dl.acm.org/doi/10.1145/3434321
57. https://mlir.llvm.org/docs/Rationale/RationaleSimplifiedPolyhedralForm/
58. https://www.youtube.com/watch?v=mt6pIpt5Wk0
59. https://www.youtube.com/watch?v=iAF-orse4hE
60. https://xilinx.github.io/mlir-air/AIRTransformPasses.html
61. https://en.wikipedia.org/wiki/Frameworks_supporting_the_polyhedral_model
62. https://mlir.llvm.org/users/
63. https://www.lei.chat/posts/compilers-and-irs-llvm-ir-spirv-and-mlir/
64. https://docs.modular.com/mojo/vision/
65. https://www.modular.com/mojo
66. https://en.wikipedia.org/wiki/Mojo_(programming_language)
67. https://arxiv.org/abs/2503.04771
68. https://arxiv.org/pdf/2503.04771.pdf
69. https://libstore.ugent.be/fulltxt/RUG01/003/212/846/RUG01-003212846_2024_0001_AC.pdf
70. https://ism.engineer/content/papers/2025_LATTE_JuliaHLS-MLIR.pdf
71. https://discourse.llvm.org/t/how-to-translate-c-c-code-to-mlir-directly/83222
72. https://stackoverflow.com/questions/62564895/frontend-support-for-mlir
73. https://discourse.mc-stan.org/t/an-mlir-based-ir-target-for-probabilistic-programming-languages/37298
74. https://www.stephendiehl.com/posts/mlir_introduction/
75. https://mlir.llvm.org/docs/Tutorials/Toy/Ch-2/
76. https://mlir.llvm.org/docs/Tutorials/Toy/Ch-3/
77. https://www.reddit.com/r/ProgrammingBuddies/comments/1m5y9xq/im_designing_a_language_from_scratch_with_a_mlir/
78. https://discourse.julialang.org/t/should-julia-use-mlir-in-the-future/110459
79. https://dl.acm.org/doi/10.1145/3721848.3721854
80. https://github.com/coderonion/awesome-mojo-max-mlir
81. https://discourse.julialang.org/t/should-julia-use-mlir-in-the-future/110459/20
82. https://www.reddit.com/r/MachineLearning/comments/1fji12z/discussion_mojo_modular_has_anyone_used_it_in_a/
83. https://mlir.llvm.org
84. https://arxiv.org/abs/2002.11054
85. https://www.answeroverflow.com/m/1208151797575721001
86. https://www.stephendiehl.com/posts/mlir_introduction/
87. https://mlir.llvm.org/docs/Rationale/Rationale/
88. https://www.cs.cornell.edu/courses/cs6120/2023fa/blog/mlir/
89. https://ai-academy.training/2024/10/04/developing-ml-compilers-for-heterogeneous-hardware/

90. https://arxiv.org/html/2407.09333v2
91. https://h2rc.cse.sc.edu/2022/slides/7_Neuendorffer.pdf
92. https://www.lei.chat/posts/compilers-and-irs-llvm-ir-spirv-and-mlir/
93. https://discourse.mc-stan.org/t/an-mlir-based-ir-target-for-probabilistic-programming-languages/37298
94. https://www.modular.com/blog/democratizing-ai-compute-part-8-what-about-the-mlir-compiler-infrastructure
95. https://sumble.com/tech/mlir
96. https://rcs.uwaterloo.ca/~ali/cs842-s23/papers/mlir.pdf
97. https://pldi25.sigplan.org/details/pldi-2025-src/3/An-MLIR-Dialect-for-Distributed-Heterogeneous-Computing
98. https://mlir.llvm.org/docs/Dialects/Affine/
99. https://llvm.org/devmtg/2021-02-28/slides/Vinay-MLIR-loopnest.pdf
100. https://www.stephendiehl.com/posts/mlir_affine/
101. https://mlir.llvm.org/docs/Tutorials/Toy/Ch-5/
102. https://www.jeremykun.com/2023/08/10/mlir-running-and-testing-a-lowering/
103. https://discourse.julialang.org/t/should-julia-use-mlir-in-the-future/110459
104. https://www.reddit.com/r/ProgrammingLanguages/comments/ygs8t1/compilers_and_irs_llvm_ir_spirv_and_mlir/
105. https://www.cs.cornell.edu/courses/cs6120/2022sp/blog/hcl-mlir/
106. https://www.reddit.com/r/ProgrammingLanguages/comments/at0alm/mlir_primer_a_compiler_infrastructure_for_the_end/
107. https://arxiv.org/html/2407.09333v1
108. https://discourse.llvm.org/t/idea-mlir-for-networking-dsp-a-vision-for-a-new-infrastructure-starting-with-a-simulator-poc/88031
109. https://mlir.llvm.org/docs/TargetLLVMIR/
110. https://www.lei.chat/posts/compilers-and-irs-llvm-ir-spirv-and-mlir/
111. https://llvm.org/devmtg/2021-02-28/slides/Vinay-MLIR-codegen.pdf
112. https://mlir.llvm.org/docs/Dialects/LLVM/
113. https://mlir.llvm.org/docs/Tutorials/Toy/Ch-6/
114. https://www.jeremykun.com/2023/11/01/mlir-lowering-through-llvm/
115. https://apxml.com/courses/compiler-runtime-optimization-ml/chapter-2-advanced-ml-intermediate-representations/mlir-lowering-paths
116. https://h2rc.cse.sc.edu/2022/slides/7_Neuendorffer.pdf
117. https://arxiv.org/html/2407.09333v2
118. https://groups.google.com/g/llvm-dev/c/n0vU71iHNis
119. https://mlir.llvm.org/docs/Dialects/SPIR-V/
120. https://llvm.org/devmtg/2022-11/slides/TechTalk16-SPIR-V-Backend-in-LLVM.pdf
121. https://discourse.llvm.org/t/mlir-codegen-native-jit-focused-x86-code-generation-for-mlir-without-the-llvm-dialect/72870

122. https://arxiv.org/pdf/2202.03293.pdf
123. https://mlir.llvm.org/users/
124. https://dl.acm.org/doi/10.1145/3721848.3721854
125. https://www.reddit.com/r/Compilers/comments/1hmy886/backend_codegenoptimizations_for_tpus/
126. https://lup.lub.lu.se/luur/download?func=downloadFile&recordOId=9146373&fileOId=9146374
127. https://discourse.mc-stan.org/t/an-mlir-based-ir-target-for-probabilistic-programming-languages/37298
128. https://www.jeremykun.com/2023/08/10/mlir-getting-started/
129. https://discourse.llvm.org/t/shameless-advertisement-for-a-beginner-friendly-llvm-backend-book/88194
130. https://pldi25.sigplan.org/details/pldi-2025-src/3/An-MLIR-Dialect-for-Distributed-Heterogeneous-Computing
131. https://mlir.llvm.org/docs/DialectConversion/
132. https://www.jeremykun.com/2023/10/23/mlir-dialect-conversion/
133. https://code.ornl.gov/llvm-doe/llvm-project/-/blob/bolt/main/mlir/docs/DialectConversion.md
134. https://www.stephendiehl.com/posts/mlir_introduction/
135. https://www.reddit.com/r/vulkan/comments/evcnbn/is_there_a_reliable_method_to_compiler_opencl_c/
136. https://gricad-gitlab.univ-grenoble-alpes.fr/violetf/llvm-project/-/blob/8b264613260cb5c881b3b5634189bdceae3f93d8/mlir/docs/ConversionToLLVMDialect.md
137. https://docs.tenstorrent.com/tt-mlir/
138. https://mlir.llvm.org/users/
139. https://github.com/llvm/torch-mlir
140. https://www.modular.com/blog/democratizing-ai-compute-part-8-what-about-the-mlir-compiler-infrastructure
141. https://apxml.com/courses/compiler-runtime-optimization-ml/chapter-2-advanced-ml-intermediate-representations/mlir-lowering-paths
142. https://arxiv.org/abs/2503.04771
143. https://www.lei.chat/posts/mlir-codegen-dialects-for-machine-learning-compilers/
144. https://onnx.ai/onnx-mlir/doxygen_html/OnnxMlirRuntime/
145. https://flang.llvm.org/docs/Overview.html
146. https://docs.modular.com/mojo/vision/
147. https://en.wikipedia.org/wiki/Mojo_(programming_language)
148. https://libstore.ugent.be/fulltxt/RUG01/003/212/846/RUG01-003212846_2024_0001_AC.pdf
149. https://stackoverflow.com/questions/62564895/frontend-support-for-mlir
150. https://discourse.llvm.org/t/how-to-translate-c-c-code-to-mlir-directly/83222
151. https://ism.engineer/content/papers/2025_LATTE_JuliaHLS-MLIR.pdf
152. https://discourse.mc-stan.org/t/an-mlir-based-ir-target-for-probabilistic-programming-languages/37298
153. https://mlir.llvm.org/docs/Tutorials/MlirOpt/
154. https://www.stephendiehl.com/posts/mlir_introduction/

155. https://www.jeremykun.com/2023/08/10/mlir-getting-started/
156. https://mlir.llvm.org/docs/LangRef/
157. https://mlir.llvm.org
158. http://lastweek.io/notes/MLIR/
159. https://www.reddit.com/r/ProgrammingLanguages/comments/lnqr8a/compiler_ir_well_il_design_question_syntax_for/