The Transformative Impact of MLIR: Key Developments in AI Compilation and Hardware Co-Design (2022-2025)

1. Introduction: MLIR in the Era Beyond Moore's Law

The Compiler Challenge

The landscape of computing is undergoing a fundamental transformation. The relentless pace of Moore's Law, which dictated hardware scaling for decades, is demonstrably slowing, altering the trajectory of performance improvements.[1] Simultaneously, the complexity and scale of Artificial Intelligence (AI) models, particularly in areas like large language models (LLMs) and generative AI, are exploding. This confluence of factors has driven the proliferation of diverse and specialized hardware accelerators – Graphics Processing Units (GPUs), Tensor Processing Units (TPUs), Neural Processing Units (NPUs), Intelligence Processing Units (IPUs), AMD's AI Engines (AIEs), Field-Programmable Gate Arrays (FPGAs), and custom Application-Specific Integrated Circuits (ASICs) – each designed to tackle specific computational bottlenecks.[1] This heterogeneity presents a profound challenge for compiler technology. Traditional monolithic compiler infrastructures, such as LLVM and GCC, while powerful for conventional CPU targets, struggle to effectively manage the complexity and diversity of these modern hardware targets and the high-level abstractions used in AI frameworks.[1] The result was often a fragmented ecosystem, with different frameworks (TensorFlow, PyTorch, JAX) and hardware vendors developing bespoke, often incompatible, compiler stacks, leading to duplicated effort and hindering portability.[3]

Enter MLIR

Multi-Level Intermediate Representation (MLIR) emerged as a direct response to these challenges.[1] Conceived within Google and now part of the LLVM project, MLIR represents a paradigm shift in compiler infrastructure design.[1] It is not merely another Intermediate Representation (IR), but a flexible and extensible *framework* for building compilers. Its core design principles – modularity, extensibility, and reusability – are manifest in its unique ability to represent and manipulate code at multiple levels of abstraction simultaneously.[1] This is achieved through the central concept of **dialects**: self-contained units that define domain-specific operations, types, and attributes.[1] These dialects allow MLIR to bridge the gap between high-level programming models used in AI frameworks and the low-level details of target hardware, facilitating progressive lowering and optimization across different abstraction layers.[6] MLIR's Static Single Assignment (SSA)-based structure provides a robust foundation for

analysis and transformation.[7]

Report Scope and Purpose

This report synthesizes the most significant developments, breakthroughs, and impactful applications within the MLIR ecosystem over the approximately 2022-2025 timeframe. It provides an expert-level analysis focusing on MLIR's evolving role in AI software optimization, including the critical area of data processing and pipeline management, its integration with mainstream AI frameworks, its transformative influence on AI hardware acceleration and co-design, and the burgeoning research landscape surrounding it. Particular attention is given to the innovative 'Transform' dialect and its implications for compiler control. The objective is to present a comprehensive technical assessment of MLIR's impact and trajectory for an audience of technical leaders, engineers, and researchers engaged with AI systems and compiler technology. The analysis underscores that MLIR's rise is not merely incidental but a necessary evolutionary step in compiler technology, driven by the fundamental shifts in hardware capabilities and the computational demands of modern AI.

2. The Evolving MLIR Ecosystem: Growth, Principles, and Identity

2.1 Core Infrastructure Growth and Community Expansion

The period from 2022 to 2025 witnessed substantial growth and activity within the MLIR project, reflecting its increasing importance and adoption. As an integral part of the broader LLVM ecosystem, MLIR benefited from the significant development velocity of LLVM itself. In 2024 alone, the main LLVM repository saw nearly 37,500 commits, adding over 9.3 million lines of code, roughly in line with activity in 2022 and 2023.[15] While these figures represent the entire LLVM project, they indicate a healthy and active development environment within which MLIR resides.

More telling is the dramatic expansion of the contributor base. The number of unique authors contributing to LLVM surged from 1,573 in 2022 and 1,932 in 2023 to an unprecedented 2,138 in 2024.[15] This represents a more than six-fold increase compared to the 336 authors a decade prior in 2014.[15] This rapid growth strongly correlates with the emergence and maturation of MLIR, signaling widespread industry and academic investment. The increasing number of contributors from diverse organizations signifies that MLIR has transcended its origins as a Google-internal project [3] and is becoming a de facto standard infrastructure for compiler development, particularly in the AI domain.

Key catalysts for this growth were the open-sourcing of MLIR and its contribution to

the LLVM Foundation, which made the technology accessible to the entire industry and fostered collaborative development.[3] The community actively engages through platforms like the LLVM Discourse forums, which replaced the older mailing list system to provide better structure, searchability, and integration (e.g., with GitHub accounts).[16] Furthermore, events like the biannual LLVM Developers' Meetings, which explicitly include MLIR content and dedicated MLIR Workshops, serve as crucial hubs for knowledge sharing, discussion, and networking among developers, researchers, and users.[17]

2.2 Foundational Concepts: Dialects, Operations, Extensibility

MLIR's power and flexibility stem from a set of well-defined core concepts.[7] At its heart, MLIR represents programs as a graph-like structure of **Operations**, which consume and produce **Values**. Each Value has a **Type** known at compile time, and Operations can possess **Attributes** representing compile-time constant information.[8] Operations are organized within **Blocks**, which in turn reside within **Regions**, allowing for nested structures essential for representing control flow or scoped computations.[8] MLIR utilizes an SSA (Static Single Assignment) form, simplifying dataflow analysis.[7]

The most fundamental concept enabling MLIR's modularity and extensibility is the **Dialect**.[1] Unlike traditional compiler infrastructures often centered around a single, monolithic IR [4], MLIR allows developers to define custom dialects. Each dialect encapsulates a set of operations, types, and attributes relevant to a specific domain or abstraction level. This allows MLIR to represent diverse concepts, from high-level framework operations (like TensorFlow or PyTorch ops) and mathematical abstractions (like linear algebra on tensors) down to hardware-specific instructions and control flow constructs, all within a unified infrastructure.[1]

Traditional IRs, operating at a single abstraction level, often face a dilemma when compiling complex domains like AI. Representing high-level constructs directly can make the IR unwieldy, while lowering too early ("premature lowering" [7]) discards valuable semantic information needed for effective optimization. MLIR's dialects circumvent this by enabling a multi-level representation.[8] Compilers can start with high-level dialects preserving domain semantics and progressively lower the IR through a series of intermediate dialects, applying optimizations at the most appropriate level.[6] This modularity significantly simplifies the construction of complex compilers, particularly for heterogeneous hardware targets.[7]

Dialects are typically defined using **TableGen**, a configuration description language within LLVM, which generates much of the C++ boilerplate code for operations, types, and their interfaces.[14] The MLIR project includes a rich set of standard dialects that

serve as building blocks, including func (functions), arith (standard arithmetic), scf (structured control flow), affine (polyhedral abstractions), linalg (linear algebra on tensors/memrefs), tensor, memref (memory buffers), vector, gpu (GPU abstractions), and llvm (for lowering to LLVM IR).[8] This dialect-based architecture represents a fundamental shift from monolithic IRs towards composable, multi-level abstractions, allowing compiler developers to tackle complexity by isolating concerns at appropriate levels.

2.3 MLIR's Identity Crisis: General Infrastructure vs. AI Solution

MLIR originated within Google Brain specifically to address the fragmentation and complexity of AI compiler stacks.[3] The initial goal was to create a unified, reusable infrastructure to replace the myriad of incompatible graph technologies and compilers used across projects like TensorFlow, XLA, and TensorFlow Lite, particularly for targeting hardware like TPUs.[3]

However, MLIR was intentionally designed as a *general-purpose* compiler framework, not strictly limited to machine learning.[3] Its powerful abstractions and extensibility quickly attracted interest from other domains, leading to its adoption in areas like quantum computing (e.g., NVIDIA's CUDA Quantum [27]), hardware design and high-level synthesis (HLS) through projects like CIRCT [3], homomorphic encryption (HEIR [29]), and potentially even database query compilation (Substrait-MLIR [13]).

This broad success, fueled by its open-sourcing and contribution to the LLVM Foundation [3], created an "identity crisis".[3] While MLIR excels as domain-agnostic, reusable infrastructure, the AI community simultaneously pushed for it to become an end-to-end AI compiler solution.[3] This led to a "dialect explosion," where numerous AI-specific dialects (representing framework operations, intermediate optimizations, etc.) were added to the upstream MLIR project, sometimes with limited governance.[3]

This situation conflates MLIR's core, general-purpose infrastructure with the specific AI solutions built upon it. It raises questions about what "MLIR" as a project truly encompasses and guarantees.[3] While MLIR forms the foundation for major AI compiler projects like OpenXLA and Triton, and is even used within parts of NVIDIA's CUDA stack [3], the ambiguity between its role as a general framework and a specific AI solution persists. This internal complexity risks creating fragmentation *within* the MLIR ecosystem, potentially mirroring the external fragmentation it was designed to solve. Recent efforts towards improved governance, such as establishing distinct Area Teams for MLIR Core and specific dialects, aim to address this challenge and clarify the project's structure and identity.[3]

3. MLIR for AI Data Pipelines: Pre-compilation, Structuring, and Wrangling

3.1 The Critical Role of Data Pipelines in AI

Data is the fundamental input for machine learning models, and the process of preparing this data – often termed the Extract, Transform, Load (ETL) pipeline – is critical for successful model training and deployment.[30] These pipelines extract raw data from diverse sources (databases, files, APIs), transform it through cleaning, normalization, feature engineering, and structuring, and finally load it into a format suitable for consumption by ML frameworks.[30] Ensuring data quality, consistency, accuracy, and efficient processing is paramount, as the performance and reliability of ML models are directly dependent on the data they are trained on.[30]

Input data pipelines often represent a significant performance bottleneck, consuming substantial compute resources and potentially starving hardware accelerators like GPUs and TPUs, which can process training steps much faster than data can be supplied.[32] Efficient ETL involves complex operations like handling large data volumes, applying intricate transformations, managing data shuffling and batching, and overlapping communication with computation.[32] Common preprocessing steps include data cleaning (handling missing values, outliers, inconsistencies), integration from multiple sources, data reduction (aggregation, feature selection), data transformation (normalization, scaling, encoding categorical features), and discretization.[36]

3.2 MLIR Approaches to Data Optimization

MLIR presents a compelling infrastructure for optimizing data pipelines due to its ability to represent diverse computations and data structures within a unified framework.[8] While traditional ML workflows often use separate tools for ETL (e.g., Spark, Pandas, Airflow [19]) and model training, MLIR offers the potential to represent and optimize both stages cohesively. This co-location enables cross-stage optimizations that are difficult to achieve otherwise.

Several MLIR-based approaches target data pipeline optimization:

- **Data Tiling and Packing:** Specialized hardware often requires data to be arranged in specific layouts or processed in tiles for optimal performance. MLIR can be used to model and optimize these data arrangements. For instance, work on targeting AMD's Ryzen AI NPUs uses MLIR-based techniques to derive optimal data tiling and packing strategies, managing data flow through the processor array and leveraging low-level DMA control for efficient data movement.[42]
- **Data Structuring and Wrangling:** MLIR dialects can effectively model various

data structures beyond simple tensors, such as multi-dimensional arrays (memrefs) [8] and potentially even dataframes or semi-structured data like JSON.[13] Standard operations within dialects like tensor, memref, and arith, along with custom operations, can represent common data transformations like reshaping, transposing, type conversions, and element-wise operations.[12]

- **Optimization Techniques:** MLIR's pass infrastructure allows applying various optimizations to data manipulation code. Canonicalization passes can simplify redundant operations (e.g., transpose(transpose(x)) -> x, or chains of reshapes).[40] Common subexpression elimination (CSE) can remove repeated calculations.[44] Pattern rewriting frameworks (both C++ based and declarative DRR) enable targeted optimizations like constant folding through reshape operations.[40] Furthermore, MLIR's ability to represent higher-level constructs facilitates advanced Data Layout Optimizations (DLO) potentially applied via Link-Time Optimization (LTO). Because MLIR can preserve information about structures (like C structs) across modules, LTO passes can perform optimizations like instance interleaving (rearranging fields of structs in an array for better cache locality) and dead field elimination (removing unused struct fields), which are simpler to implement in MLIR than in lower-level IRs like LLVM IR.[43] Bufferization passes manage the explicit allocation and deallocation of memory buffers [18], and specialized primitives like reuse_at and buffer_at (demonstrated in HeteroCL) can explicitly manage on-chip memory hierarchies and reuse buffers to minimize off-chip memory access.[41]

The ability to represent both data preparation steps and model computation within the same MLIR framework opens possibilities for holistic optimizations, such as fusing data preprocessing operations directly into compute kernels or optimizing data layouts based on how the subsequent model consumes the data.[43]

3.3 Case Studies and Projects

Several research projects and tools demonstrate MLIR's application to data-centric tasks:

- **DAPHNE:** This system explicitly targets integrated data analysis pipelines encompassing ETL, ML, and HPC.[34] It uses a custom MLIR dialect, DaphneIR, to represent operations on frames and matrices, along with standard dialects like SCF for control flow. DAPHNE performs multi-level optimizations within MLIR before lowering to LLVM, aiming to optimize the entire workflow holistically.[34]
- **HeteroCL:** Originally based on Halide IR, HeteroCL migrated to MLIR for improved scalability and extensibility in defining hardware customizations.[41] Its HCL dialect allows specifying compute, data type, and memory customizations. Notably, it

includes primitives like .reuse_at() and .buffer_at() to explicitly generate and manage reuse buffers and write buffers within custom memory hierarchies, crucial for optimizing data movement in data-intensive applications.[41]

- **Substrait-MLIR:** This ongoing project aims to create an MLIR dialect for Substrait, a cross-language format for database query plans.[13] By representing database operations within MLIR, it seeks to provide a common infrastructure for query optimization and potentially bridge the gap between traditional data processing systems and MLIR's AI/HPC capabilities.[13]
- **Noisy Arithmetic Example:** While focused on homomorphic encryption (FHE) [29], the example of tracking "noise" through integer arithmetic using an MLIR analysis pass demonstrates a relevant capability.[45] Similar analyses could track data quality metrics, distributions, or other properties through complex ETL transformations within MLIR.

These projects illustrate that MLIR's flexibility extends beyond tensor computations. Its application to data management, memory hierarchy optimization, and query plan representation signals a trend towards using MLIR as a unifying infrastructure for data engineering and AI engineering tasks.

3.4 Comparison and Challenges

While MLIR offers significant potential for optimizing the *transformation* and *loading* stages of ETL, particularly when tightly coupled with ML model execution, it currently faces challenges compared to mature, dedicated ETL frameworks and libraries.

MLIR's strengths lie in its potential for deep hardware optimization, fusion of data preparation with compute kernels, and unified representation. However, established ETL tools like Apache Spark, Apache Airflow, Pandas, or commercial platforms [19] possess rich ecosystems with extensive connectors for diverse data sources (the 'Extract' stage), sophisticated orchestration and scheduling features, mature monitoring capabilities, and high-level APIs optimized for data manipulation productivity.

Representing complex data cleaning logic (e.g., intricate validation rules, fuzzy matching) or stateful transformations might be less straightforward or efficient in MLIR's current dialects compared to specialized Python libraries like Pandas or data quality tools.[36] Therefore, MLIR is unlikely to replace the entire data engineering stack in the near term. Its most promising role appears to be in accelerating the computationally intensive *transformation* steps within data pipelines and enabling tighter integration and co-optimization with downstream ML model execution, rather

than managing the entire end-to-end ETL process.

4. The Transform Dialect: Unleashing Fine-Grained Compiler Control

4.1 Motivation: Beyond Monolithic Passes

Traditional compiler optimization flows rely heavily on sequences of pre-defined passes (pass pipelines), often configured via command-line flags.[46] While effective for general-purpose optimization, this coarse-grained approach often lacks the precision required to optimize specific, critical sections of code for the diverse and specialized hardware prevalent today.[46] Source-level annotations or pragmas offer finer control but are typically limited to specific transformations anticipated by compiler developers and require invasive, non-modular compiler changes to implement.[47]

A significant limitation of this model is that much of the powerful transformation logic implemented within compiler helper functions (e.g., routines for tiling, unrolling, vectorizing specific loops) remains hidden or inaccessible to the end-user unless they are willing to write custom compiler passes in C++ and rebuild the compiler – a task requiring deep compiler expertise.[46] Domain-specific scheduling languages like Halide and TVM address this by separating the algorithm from its optimization schedule, but they typically require reimplementing optimizations within their own frameworks and do not easily integrate with existing general-purpose compiler infrastructure.[46] The MLIR Transform dialect was conceived to bridge this gap, providing a mechanism to expose and compose existing compiler capabilities with fine-grained precision directly within the MLIR framework.[46]

4.2 Core Concepts and Implementation

The Transform dialect is, itself, an MLIR dialect, but its purpose is meta-compilational: it defines operations that manipulate and transform *other* MLIR code (the "payload" IR).[46] Instead of the compiler executing a fixed pipeline, it interprets a Transform dialect script provided by the user, which explicitly directs the optimization process.[46]

Key concepts include:

- **Handles:** Transform operations operate on handles, which are standard MLIR SSA values. These handles represent lists of operations within the payload IR that are targeted by the transformation.[49] Handles can be produced by matching operations (e.g., match.op) or as results of other transform operations.
- **Transform Operations:** These are operations defined within the Transform dialect (e.g., loop.tile, loop.unroll, bufferization.eliminate_alloc_tensor). Each

transform operation typically takes one or more handles as input, applies a specific compiler transformation (often leveraging existing internal compiler functions) to the associated payload operations, and produces new handles corresponding to the newly created or modified payload operations.[46]

- **Payload IR:** The actual program code (e.g., user functions containing loops and computations) that is being optimized.
- **Transform IR:** The MLIR code written using the Transform dialect that specifies the sequence of optimizations to apply to the payload IR.

Consider this illustrative example adapted from [49]:

MLIR

```
// Transform IR Script
transform.named_sequence @optimize_loops(%payload_func :!transform.any_op) {
  // Find the first 'scf.for' loop inside the payload function
  %outer_loop = transform.structured.match ops{["scf.for"]} in %payload_func
    ->!transform.op<"scf.for">

  // Define tile sizes
  %c8 = transform.param.constant 8 : index

  // Tile the outer loop with size 8
  %tiled_loops:2 = transform.structured.tile_using_for %outer_loop tile_sizes [%c8]
    interchange
    -> (!transform.op<"scf.for">,!transform.op<"scf.for">)

  // Unroll the inner tiled loop (handle: %tiled_loops#1) completely
  %unrolled_inner = transform.loop.unroll %tiled_loops#1 { factor = 0 } // factor=0
means full unroll
    ->!transform.any_op

  transform.yield
}
```

This script finds a loop in the payload, tiles it, and then unrolls the inner loop resulting from the tiling. The Transform dialect is implemented within MLIR and features an

extensible design, allowing new transform operations to be added easily.[46] An interface mechanism allows existing C++ helper functions within the compiler to be exposed as Transform dialect operations.[47]

4.3 Key Capabilities

The Transform dialect provides several powerful capabilities:

- **Composition:** Simple, atomic transform operations can be chained together, using the handles produced by one transform as input to the next, allowing the construction of arbitrarily complex optimization pipelines.[46]
- **Extensibility:** New transformations can be exposed by defining new Transform dialect operations and associating them with the corresponding C++ implementation, without altering the core dialect or requiring users to rebuild the entire compiler for every new optimization strategy.[46]
- **Static Verification:** A crucial feature is the system of pre- and post-conditions associated with transform operations.[46] Handle types (e.g., !transform.op<"scf.for">) specify the expected type of payload operation a transform can be applied to. Attributes can add further constraints. This allows the MLIR infrastructure to statically verify the Transform script, catching errors like applying a loop transformation to a non-loop operation or applying a destructive transform twice to the same handle *before* executing the potentially expensive compilation.[46] The transform.cast operation allows explicit type checking between transforms.[51]
- **Parameterization:** Transformations can be configured using parameters, which can be compile-time constants (like tile sizes or unroll factors) or even values derived dynamically from the payload IR itself, enabling more adaptive optimization strategies.[49]

By representing the optimization strategy itself as MLIR IR, the Transform dialect elevates compiler control from a simple configuration task to a programmable one. This "compiler programming" paradigm opens the door to analyzing, verifying, and potentially even automatically generating or optimizing the compilation strategy itself, integrating naturally with techniques like autotuning.

4.4 Applications and Impact (Case Studies from CGO 2025 Paper)

The practical utility and impact of the Transform dialect were demonstrated through five case studies presented in the CGO 2025 paper [46]:

1. **Expressing Pass Pipelines:** This study showed that existing, coarse-grained pass pipelines can be faithfully replicated using Transform dialect scripts with

negligible compilation time overhead, confirming its efficiency as a control mechanism.[46]

2. **Robust Lowering:** This case study focused on lowering IR containing a complex mix of dialects. It highlighted the critical role of the Transform dialect's static pre- and post-conditions in building robust and reliable lowering sequences, preventing errors that might occur in less explicitly controlled pass pipelines.[46]
3. **Debugging Performance:** The Transform dialect proved effective in diagnosing performance regressions. By enabling fine-grained control over which optimizations were applied where, developers could quickly isolate and disable counter-productive transformation patterns that were hurting performance.[46]
4. **Fine-Grained Optimization:** This study demonstrated the power of precise control. By meticulously applying loop tiling, vectorization, and importantly, integrating calls to specialized, highly optimized microkernel library functions (exposed via custom transform ops), significant performance improvements were achieved on relevant benchmarks, surpassing what standard pass pipelines could deliver.[49]
5. **Autotuning Integration:** The final case study showed the ease with which the Transform dialect integrates with state-of-the-art autotuning frameworks. The parameterized nature of the Transform script allowed search algorithms to effectively explore the optimization space (e.g., different tile sizes, unroll factors) to find high-performing configurations automatically.[48]

These case studies collectively validate the Transform dialect's practical value across the compiler development and performance engineering lifecycle. They provide concrete evidence that it delivers on its promise of fine-grained control, reusability of compiler internals, improved robustness, and seamless integration with automated tuning methods, offering tangible advantages over traditional compiler control mechanisms.

5. MLIR Integration in Mainstream AI Frameworks: Bridging the Gap

5.1 The Need for Framework Integration

Modern AI frameworks like TensorFlow, PyTorch, and JAX provide high-level, productive interfaces for defining complex machine learning models. However, translating these high-level descriptions into efficient code that runs optimally across a diverse landscape of hardware accelerators (CPUs, GPUs, TPUs, etc.) is a major challenge.[1] This necessitates sophisticated compiler backends capable of understanding both the semantics of the ML framework and the intricacies of the

target hardware. MLIR, along with related projects like XLA and IREE, has emerged as a critical technology for building these compiler backends, enabling performance optimization, hardware targeting, and improved portability.

5.2 TensorFlow & OpenXLA

TensorFlow has long utilized XLA (Accelerated Linear Algebra) as a compiler backend to optimize performance, particularly on Google's TPUs and also for GPUs and CPUs.[52] XLA aims to improve execution speed by fusing operations and specializing code, enhance memory usage via buffer analysis, and reduce reliance on custom ops by optimizing fused low-level ops automatically.[52]

XLA's architecture heavily involves MLIR.[52] While historically using its own HLO (High Level Operations) representation, the modern XLA pipeline increasingly relies on MLIR dialects. The **StableHLO** dialect now serves as the primary, versioned interface layer between ML frameworks (including TensorFlow, PyTorch via Torch-MLIR, and JAX) and MLIR-based compilers like XLA and IREE.[52] Models are lowered from the framework's representation to StableHLO, which is then consumed by the compiler backend. XLA performs target-independent optimizations on StableHLO/HLO (like CSE, fusion) before invoking target-specific backends (e.g., GPU, CPU) for further optimization and code generation, often via the MLIR LLVM dialect.[52]

The **TOSA** (Tensor Operator Set Architecture) dialect is another MLIR dialect relevant to TensorFlow, particularly for TensorFlow Lite (TFLite) inference.[61] However, the incremental upgrade of TOSA to v1.0 exposed significant compatibility challenges between TensorFlow/TFLite (which generated the older TOSA version) and downstream compilers like IREE (which adopted the newer v1.0).[61] This breakage, occurring around late 2024 / early 2025, necessitated users pinning to older versions of TensorFlow, IREE, and associated tooling to maintain compatibility, highlighting the critical need for careful versioning and coordination across the decoupled components of the MLIR ecosystem.[61]

To simplify the integration of diverse hardware backends with frameworks like TensorFlow and JAX, the **PJRT** (Plugin-based Runtime) interface was developed and open-sourced as part of OpenXLA.[58] PJRT provides a standardized API for frameworks to discover, load, and interact with different compiler runtimes and hardware devices dynamically. This allows hardware vendors to provide PJRT plugins for their devices, enabling framework support without requiring deep integration into the framework's core codebase.[59] Intel, for example, uses PJRT to provide its GPU backend for TensorFlow and JAX.[59]

The **OpenXLA** project represents a collaborative effort by Google and numerous industry partners (including AMD, NVIDIA, Intel, Arm, Meta, AWS) to develop an open-source ecosystem of ML compiler technologies, with XLA, StableHLO, IREE, and PJRT as key components, all leveraging MLIR.[52] This initiative aims to standardize interfaces, promote portability, and reduce the N*M integration complexity between frameworks and hardware targets.

5.3 PyTorch

PyTorch, known for its dynamic nature and Pythonic interface, presents different challenges for compiler integration compared to TensorFlow or JAX. The **Torch-MLIR** project serves as the primary bridge connecting the PyTorch ecosystem to MLIR-based backends.[62] It is designed as core infrastructure *for building* end-to-end compilation flows, rather than being a complete compiler itself.[62]

Torch-MLIR's architecture features a frontend and a backend.[62] The frontend ingests various PyTorch program representations (primarily via PyTorch's JIT IR, which can be produced by TorchScript, TorchDynamo/torch.compile, torch.fx, etc.) and lowers them to the MLIR torch dialect. This dialect mirrors many PyTorch concepts, including its type system and operators.[62]

A critical stage in the frontend is lowering the torch dialect representation to conform to the **"backend contract"**. This contract defines a subset of the torch dialect with specific properties required by downstream MLIR backends: tensors must have value semantics (be immutable and non-aliased), and tensors must have known ranks (number of dimensions) and data types (dtypes), ideally with fully known shapes.[62] Achieving this contract, especially when starting from TorchScript (which represents stateful nn.Module hierarchies and lacks static shape information), requires significant transformations handled by pipelines like torchscript-module-to-torch-backend-pipeline. These include functionalization (converting stateful modules to functional code), shape and dtype inference (often requiring user hints), and simplification of Pythonic constructs.[62] The impedance mismatch between PyTorch's dynamic, object-oriented nature and the typically static, functional nature expected by MLIR compiler backends necessitates this dedicated bridging infrastructure.

Once the IR conforms to the backend contract, Torch-MLIR's backend can lower it to various target MLIR dialects, including Linalg (for CPU/GPU codegen via LLVM), TOSA, and StableHLO (for integration with XLA/IREE).[62] This modular design allows different compiler backends to consume PyTorch models via the standardized backend

contract provided by Torch-MLIR.

5.4 JAX

JAX leverages a functional programming paradigm combined with transformations like jax.jit (just-in-time compilation), jax.grad (automatic differentiation), and jax.vmap (auto-vectorization).[65] For its JIT compilation capabilities, JAX relies heavily on the XLA compiler.[65]

The integration between JAX and MLIR-based backends like XLA and IREE is facilitated primarily through StableHLO and the PJRT runtime interface.[52] When jax.jit is invoked, the JAX function is traced and converted into JAX IR, which is then lowered to StableHLO.[67] This StableHLO representation is passed via the PJRT interface to the selected backend (e.g., XLA compiler for GPU/TPU, IREE compiler, or potentially other PJRT plugins) for optimization and code generation.[52]

JAX's functional nature generally maps more cleanly onto compiler IRs like StableHLO compared to the complexities of handling PyTorch's stateful modules. This relatively direct mapping simplifies the compiler integration task and likely contributes to JAX's strong performance and adoption on accelerators via XLA and IREE.[54] Research efforts also explore extending JAX's capabilities using MLIR backends, such as the experimental work on providing MLIR-based sparse tensor support for JAX.[68] JAX, combined with XLA's capabilities for automatic parallelization (GSPMD) [54] and PJRT's multi-device support, is widely used for large-scale model training.[54]

5.5 Framework Integration Summary

The integration of MLIR into major AI frameworks is a dynamic and evolving process, aiming to provide portability and performance across diverse hardware. The move towards standardized interfaces like StableHLO and PJRT within the OpenXLA ecosystem represents a significant effort to create a more modular and interoperable landscape. However, challenges related to dialect versioning, maintaining performance parity, and bridging the gap between dynamic framework features and static compiler requirements remain active areas of development.

Table 1: MLIR Integration in Major AI Frameworks (ca. 2022-2025)

Framework	Key MLIR Integration Project(s)	Core Input Dialect(s) to Compiler	Integration Interface/Layer	Notable Successes/ Capabilities	Key Challenges/ Recent

					Issues
TensorFlow	OpenXLA (XLA, IREE), TensorFlow Lite	StableHLO, TOSA	PJRT, TF C API	Strong TPU/GPU/CPU support via XLA, TFLite inference ecosystem	TOSA v1.0 compatibility issues [61], Historical complexity
PyTorch	Torch-MLIR, OpenXLA (IREE, XLA)	torch -> StableHLO, TOSA, Linalg	torch.compile, PJRT	Growing backend support (IREE, XLA), Modular bridge design	Lowering complexity (state, dynamic shapes) [62], Performance tuning
JAX	OpenXLA (XLA, IREE)	StableHLO	jax.jit, PJRT	High performance on accelerators, Clean functional mapping	Reliance on XLA/IREE backend maturity, Custom op handling [63]

6. MLIR Reshaping Hardware: AI Accelerators and Co-Design

6.1 The Imperative for Hardware-Specific Compilation

The proliferation of specialized AI accelerators is a direct consequence of the need to overcome the limitations of general-purpose processors for demanding AI workloads.[1] Achieving peak performance on these diverse architectures—ranging from massively parallel GPUs to dataflow-oriented TPUs/IPUs and VLIW-based AIEs—requires compilers that can understand and exploit their unique features.[2] Generic compilation strategies are often insufficient.[4] MLIR's extensible dialect system provides a powerful mechanism for hardware designers and compiler engineers to create domain-specific compilers that effectively map high-level AI models onto specialized hardware, significantly reducing the cost and complexity compared to building compilers from scratch.[1]

6.2 Tenstorrent

Tenstorrent provides a compelling example of deep, native MLIR adoption for

targeting specialized AI hardware. Their core compiler is **TT-Forge**, explicitly built upon MLIR.[58] The associated **TT-MLIR** open-source project defines a hierarchy of custom MLIR dialects to represent computations targeting Tenstorrent accelerators [58]:

- **TTIR (Tenstorrent Intermediate Representation):** A primary IR level for Tenstorrent hardware.
- **TTNN (Tenstorrent Neural Network):** Likely represents higher-level neural network constructs or fused operations suitable for their architecture.
- **TTKernel:** Represents lower-level kernel details.
- (Future dialects like .ttm, .ttnn mentioned in TT-Explorer roadmap [73])

TT-Forge supports multiple frontends to ingest models from standard frameworks, leveraging open standards: tt-torch (using PyTorch 2.X/torch-mlir, outputting StableHLO), tt-forge-fe (using TVM to handle PyTorch, ONNX, TF), and tt-xla (using PJRT to ingest JAX via StableHLO).[58]

A particularly innovative aspect of Tenstorrent's toolchain is **TT-Explorer**, a graphical tool designed for "Human-In-Loop" compilation.[58] TT-Explorer allows users to visualize the TTIR graph, inspect operation attributes, view performance and accuracy metrics overlaid on the graph, edit parameters via an "Overrides" mechanism, trigger re-compilation, and observe the results.[73] Its roadmap includes support for more dialects, visualizing graph transformations, and integration with other tools.[73] This interactive approach, enabled by MLIR's structured IR, empowers developers to directly tune and optimize models for Tenstorrent hardware. Tenstorrent's strategy showcases a full commitment to the MLIR philosophy, building a comprehensive, MLIR-native toolchain with custom dialects and novel interactive tooling, while also embracing interoperability through standards like StableHLO and PJRT.

6.3 NVIDIA

NVIDIA's CUDA platform remains the dominant ecosystem for GPU computing. While CUDA itself predates MLIR, NVIDIA is actively integrating MLIR into its compiler stack, leveraging its capabilities while building upon its existing, mature infrastructure.[3] NVIDIA contributes significantly to the LLVM project, upon which its CUDA Compiler (NVCC) is based.[74]

MLIR's integration appears primarily as intermediate layers bridging high-level representations to NVIDIA's established backend:

- **NVVM IR:** This is NVIDIA's internal, LLVM IR-based representation for GPU kernels, featuring specific conventions, address spaces (global, shared, constant), and intrinsic functions.[74] NVCC compiles source languages or

higher-level IRs down to NVVM IR, which is then optimized and translated to PTX (Parallel Thread Execution) assembly.[74] NVVM IR has its own versioning and debug metadata specifications.[75]

- **MLIR gpu Dialect:** This standard MLIR dialect provides target-agnostic abstractions for common GPU programming concepts, such as kernel launches (gpu.launch), kernel functions (gpu.func), thread and block IDs (gpu.thread_id, gpu.block_id), barriers, and memory spaces (global, workgroup/shared).[26] A typical compilation pipeline involves outlining the body of a gpu.launch into a separate gpu.func kernel, attaching target-specific information (like SM architecture via nvvm.attach_target), and then lowering the gpu dialect operations to the nvvm dialect using passes like convert-gpu-to-nvvm.[26]
- **MLIR nvgpu Dialect:** This dialect serves as a bridge between the target-agnostic gpu and vector dialects and the target-specific nvvm dialect.[76] It represents NVIDIA-specific hardware features and PTX-level operations directly in MLIR, such as asynchronous data copies between global and shared memory (nvgpu.device_async_copy, managed via groups), memory barriers (nvgpu.mbarrier.*), matrix load operations (nvgpu.ldmatrix), Tensor Memory Accelerator (TMA) operations for efficient memory access (nvgpu.tma.*), and Matrix Multiply-Accumulate (MMA) instructions, including support for sparse MMA and warpgroup-level operations targeting newer architectures.[76] This allows optimizations related to these specific hardware features to be expressed and performed within MLIR before the final lowering to NVVM/PTX.
- **CUDA Quantum:** For its quantum computing platform, NVIDIA adopted a more MLIR-native approach from the outset.[27] The nvq++ compiler uses Clang to parse C++ code and then leverages custom MLIR dialects (**Quake** for quantum operations, **CC** for classical control) to represent the quantum kernels.[28] Tools like cudaq-quake perform the C++ AST to MLIR conversion, and cudaq-opt applies MLIR passes for optimization.[28] The platform even allows users to register and run their own custom MLIR passes on the Quake IR.[27]

Overall, NVIDIA's strategy appears evolutionary, integrating MLIR into its highly optimized CUDA/NVCC/LLVM toolchain primarily as intermediate abstraction layers (gpu, nvgpu) rather than replacing the entire backend. This leverages MLIR's strengths in handling higher-level structures while retaining the mature and performant NVVM/PTX code generation infrastructure. Newer initiatives like CUDA Quantum demonstrate a deeper, ground-up MLIR adoption. Public details on future MLIR roadmap specifics beyond existing dialects are limited, though GTC presentations hint at ongoing work, potentially around runtime compilation or

enhanced Python integration.[77]

6.4 AMD

AMD is actively utilizing and contributing to the LLVM/MLIR ecosystem to support its diverse range of hardware, including CPUs, ROCm-based GPUs, Ryzen AI NPUs, and Versal AI Engines (AIEs).

- **Ryzen AI NPUs:** For its NPUs based on XDNA architecture (found in Ryzen AI processors like Phoenix, Hawk Point, and the upcoming Strix Point with XDNA2), AMD open-sourced **"Peano"**.[15] Peano is an LLVM compiler backend designed specifically for these AI engines, enabling compilation for this specialized hardware within the standard LLVM/MLIR framework.[15] Complementing this, work presented at FOSDEM 2025 focuses on using MLIR dialects and passes for optimizing data tiling and packing specifically for Ryzen AI NPUs, aiming to efficiently manage data movement and utilize DMA capabilities.[42]
- **ROCm & GPUs:** AMD continues to enhance its ROCm platform for GPU computing. Research efforts showcased include running standard, unmodified C/C++ code directly on AMD GPUs via LLVM/ROCm, bypassing the need for specific GPU languages.[15] The porting of the classic game DOOM to run almost entirely on the GPU using ROCm and LLVM libc serves as a demonstration of this capability.[15] Frameworks like IREE utilize MLIR to compile models (e.g., from PyTorch) for execution on AMD GPUs (often via SPIR-V or ROCm backends), offering an alternative to lower-level programming models like OpenCL or HIP.[69]
- **AI Engines (AIEs):** Targeting the complex, heterogeneous Versal ACAP devices containing AIE arrays requires sophisticated compilation flows. The **ARIES** project, developed at Cornell and collaborators, provides an MLIR-based compilation flow specifically for AIE architectures.[2] It addresses limitations of previous AIE programming frameworks by introducing a novel tile-based programming model in Python that allows users to explicitly map tasks and exploit task-level, tile-level, and instruction-level parallelism (via primitives like .to(), .pipeline(), .vectorize()).[2] ARIES uses a unified MLIR representation, leveraging the existing AIEVec dialect for core-level intrinsics and introducing a new **ADF (Adaptive Data Flow)** dialect to model the inter-tile parallelism and dataflow connections within the AIE array. It performs multi-level optimizations (global: broadcast detection, data forwarding; local: DMA-to-IO conversion, core placement, vectorization, buffer management) before generating executable code (AIE intrinsics, ADF APIs, HLS C++, XRT host code).[2] ARIES demonstrates a deep integration of MLIR, using custom dialects to effectively manage the complexity and parallelism of the AIE architecture.

AMD's strategy involves leveraging MLIR across its hardware portfolio, developing targeted compiler solutions (Peano, ARIES) and contributing backends to the open-source ecosystem. This approach allows them to tailor compilation strategies to the specific needs of their NPUs, GPUs, and AIEs within a common infrastructure framework.

6.5 IPUs (Graphcore)

Graphcore's Intelligence Processing Unit (IPU) features a unique massively parallel architecture with numerous independent cores, each with fast local memory.[71] The software stack for the IPU is the **Poplar SDK**, which was co-designed with the hardware.[79] Poplar provides a C++ graph programming framework and libraries (PopLibs [82]), along with integrations for standard ML frameworks like TensorFlow, PyTorch, and ONNX.[81]

Poplar's relationship with LLVM and MLIR is that of using them as *components* within its larger, bespoke graph compilation system:

- **LLVM:** The Poplar graph compiler uses LLVM as a backend to generate code for the individual IPU cores.[79]
- **MLIR:** Poplar utilizes MLIR for *some high-level optimizations* within its graph compiler.[71] The specific nature and extent of these MLIR-based optimizations are internal details of the Poplar compiler.

The Poplar compiler itself manages the complex tasks of scheduling the computation graph across the IPU's parallel cores, partitioning work, managing data movement between tiles using the IPU's interconnect, and optimizing memory allocation.[71] Poplar's programming model is centered around computational graphs composed of fine-grained tasks (vertices).[71]

Unlike Tenstorrent's TT-Forge or AMD's ARIES, Poplar is not fundamentally an MLIR-based compiler. Instead, it incorporates MLIR technology for specific optimization tasks within its own established graph compilation framework, which ultimately relies on LLVM for final code generation for the IPU cores. No public, specific "IPU dialect" for MLIR is documented as part of the Poplar SDK, suggesting MLIR's role is more internal compared to other accelerator vendors who expose MLIR dialects as primary interfaces.

6.6 Other Accelerator Projects & Trends

The use of MLIR for targeting AI accelerators extends beyond the major players:

- **TPU-MLIR:** An open-source project specifically targeting Sophgo's TPUs.[87] It provides a full toolchain, converting models from ONNX, PyTorch, TFLite, and Caffe into an MLIR representation using a high-level TOP (Tensor Operation) dialect, which is then lowered to a device-specific TPU dialect. The toolchain includes quantization capabilities (F16, INT8 with calibration) and generates a final executable bmodel file.[11]
- **ONNX-MLIR:** This project focuses on providing a direct compilation path from ONNX models using an ONNX dialect within MLIR.[88] It supports code generation for generic CPUs and IBM's Telum AI accelerator, offering compiler interfaces and a runtime environment.[88]
- **Intel Graph Compiler:** Intel is developing an MLIR-based graph compiler designed to optimize deep learning workloads.[89] It accepts MLIR (primarily linalg on tensors) as input, applies optimizations, and generates code for Intel CPUs and GPUs (requiring OpenCL runtime).[89]
- **Hardware/Software Co-design:** MLIR's multi-level nature inherently facilitates hardware/software co-design.[2] By allowing hardware features, constraints, and specialized instructions to be represented in dedicated dialects early in the compilation flow (as seen in ARIES [2] or the nvgpu dialect [76]), MLIR enables tighter integration between software compilation strategies and hardware capabilities. This allows optimizations to be aware of hardware specifics much earlier than in traditional flows that only target hardware late in the process via low-level IR like LLVM IR.

The widespread development of MLIR-based compilers for a variety of accelerators (Sophgo TPU, IBM Telum, Intel GPU, AMD NPU/AIE, Tenstorrent IPU) underscores MLIR's success as a foundational framework. It significantly lowers the barrier for hardware vendors and researchers to build specialized, high-performance compiler toolchains, enabling faster support for standard ML frameworks on new and existing hardware compared to developing entirely new compiler infrastructures.

6.7 Hardware Acceleration Summary

MLIR has become a central technology in the development of compilers for diverse AI hardware. Different vendors adopt varying strategies, from deep MLIR-native toolchains to using MLIR as a component within larger systems. The ability to define custom dialects is key to targeting specialized accelerator features effectively.

Table 2: MLIR Adoption in Hardware Acceleration (ca. 2022-2025)

Vendor/Proj	Key	Relevant	Primary Use	Integration	Key

ect	Compiler/Project(s)	MLIR Dialects (Standard & Custom)	Case	Approach	Frameworks Supported (via Compiler)
NVIDIA	NVCC, CUDA Quantum	gpu, nvgpu, nvvm (target), Quake, CC (Quantum)	Mid/Low-level Opt & Codegen	Component in LLVM/CUDA stack	CUDA ecosystem, C++ (Quantum)
AMD (GPU/ROCm)	ROCm Compiler, IREE	gpu, rocdl (target), amdgpu, StableHLO (via IREE)	Backend Codegen, Framework Target	LLVM Backend, IREE Integration	HIP, OpenCL, PyTorch, TF, JAX (via IREE)
AMD (Ryzen AI NPU)	Peano	Custom (Peano backend), linalg, vector?	Backend Codegen	LLVM Backend (Open Source)	C/C++, Frameworks via higher layers?
AMD (AIE/Versal)	ARIES	AIEVec, ADF (custom), affine, scf, memref	Full Heterogeneous Compilation	MLIR-native Flow	Python (Custom API)
Tenstorrent	TT-Forge (TT-MLIR)	TTIR, TTNN, TTKernel (custom), StableHLO (input)	Full Compilation Toolchain	MLIR-native Flow	PyTorch, JAX, TF, ONNX (via TVM)
Graphcore	Poplar SDK	Standard MLIR dialects (internal use)	High-Level Graph Opt.	Component in Poplar stack	PyTorch, TF, ONNX (via Poplar)
Sophgo (TPU-MLIR)	TPU-MLIR	TOP, TPU (custom)	Full Compilation Toolchain	MLIR-native Flow (Open Source)	PyTorch, ONNX, TFLite, Caffe

Intel (Graph Comp.)	Intel Graph Compiler	linalg, vector, gpu, spirv? (target)	Graph Optimization & Codegen	MLIR-based Compiler	Frameworks emitting linalg?
IBM (ONNX-MLIR)	ONNX-MLIR	ONNX (custom)	ONNX Model Compilation	MLIR-based Compiler (Open Source)	ONNX

7. Key Research Trends and Open Source Impact

7.1 Influential Research Papers & Themes (Last 2-3 Years)

The academic and research communities have actively embraced MLIR, pushing its capabilities and exploring new application domains. Several key themes and influential papers have emerged in the 2022-2025 timeframe:

- **Explicit Compiler Control (Transform Dialect):** The work culminating in the CGO 2025 paper by Lücke, Zinenko, Moses, Steuwer, and Cohen formally introduced the Transform dialect.[46] This research provides a foundational mechanism for fine-grained, programmable control over the compilation process, moving beyond static pass pipelines.
- **Heterogeneous System Compilation:** Addressing the complexity of modern systems with multiple, diverse processing units is a major focus. The ARIES paper (Zhuang et al., FPGA'25) presented a comprehensive MLIR-based flow for AMD's AIEs, demonstrating custom dialects for managing parallelism and memory hierarchies.[2] Similarly, the HETOCompiler work (arXiv:2407.09333) introduced a generic hyper dialect within MLIR to abstract data management and parallel computation for general heterogeneous platforms.[6]
- **Integrated Data Analysis Pipelines:** The DAPHNE project (Damme et al., CIDR'22) pioneered the use of MLIR to build a unified system for pipelines combining ETL, ML, and HPC tasks, showcasing MLIR's potential to bridge data management and high-performance computation.[34]
- **Modular Compiler Construction and Optimization:** Research continues on leveraging MLIR for building more modular and reusable compiler components. The work by Vasilache et al. (LCPC'22/arXiv'22) focused on composable and modular code generation techniques within MLIR, particularly for tensor compilers.[18] Performance studies, such as achieving near-peak theoretical performance for DGEMM using MLIR-based code generation, demonstrate the effectiveness of these approaches.[60]

- **Compiler Robustness and Testing:** As MLIR's complexity grows, ensuring its correctness becomes crucial. Recent research has focused on developing specialized fuzzing and testing techniques tailored for MLIR's multi-dialect structure. Projects like MLIRSmith, MLIRod, and DESIL aim to automatically generate or mutate MLIR code to uncover bugs, including challenging "silent bugs" (incorrect results without crashes) and undefined behavior (UB) arising from dialect interactions or lowering processes.[5]
- **Hardware Synthesis:** MLIR, particularly through the CIRCT project, is being explored for high-level synthesis (HLS), translating high-level languages like Julia directly into hardware description languages like Verilog.[22]

This research activity indicates a maturing MLIR ecosystem. While early work focused on establishing the core infrastructure and basic AI compilation, recent efforts are tackling more advanced challenges: managing heterogeneity, integrating data processing, enhancing compiler programmability and robustness, and extending MLIR's reach into adjacent domains like hardware design.

7.2 Notable Open Source Projects & Libraries

MLIR's success is intrinsically linked to its vibrant open-source ecosystem. Key projects and libraries leveraging MLIR include:

- **Core Infrastructure:** The upstream LLVM/MLIR project itself remains the central hub.[15]
- **Framework Integration:**
  - **Torch-MLIR:** Provides the bridge for lowering PyTorch models to MLIR dialects.[62]
  - **OpenXLA:** An ecosystem encompassing XLA (compiler), IREE (compiler+runtime), and StableHLO (portability dialect), heavily utilizing MLIR for compiling TensorFlow, PyTorch, and JAX.[52]
  - **ONNX-MLIR:** A dedicated project for compiling ONNX models via an MLIR ONNX dialect.[88]
- **Hardware Backends & Toolchains:**
  - **TPU-MLIR:** Open-source compiler for Sophgo TPUs.[87]
  - **tt-mlir:** Tenstorrent's open-source MLIR compiler components.[58]
  - **Peano:** AMD's open-source LLVM backend for Ryzen AI NPUs.[15]
  - **CIRCT:** A sub-project focused on MLIR dialects and tools for circuit design and HLS.[22]
- **Specialized Domains:**
  - **HEIR:** Developing MLIR dialects and tools for compiling Homomorphic Encryption computations.[29]

- **Substrait-MLIR:** Building an MLIR dialect for the Substrait database query plan representation.[13]

The diversity of these projects, spanning framework integration, hardware enablement, and specialized computational domains, validates MLIR's role as a versatile and powerful foundational technology. It provides the essential building blocks [1] that enable various communities and companies to construct tailored compiler solutions, fulfilling its promise as a reusable and extensible infrastructure.[1]

7.3 Community Engagement

A thriving community is essential for the continued development and adoption of an open-source project like MLIR. Key engagement mechanisms include:

- **LLVM Developers' Meetings:** These biannual conferences are major events for the entire LLVM community, including MLIR. They feature technical talks, tutorials, workshops (often with dedicated MLIR tracks), panels, and networking opportunities.[17] Presentations cover topics ranging from core MLIR features like bufferization [18] and pattern rewriting [18] to specific applications and dialect developments.
- **Open Design Meetings:** Historically, regular online Open Design Meetings provided a forum for discussing MLIR's evolution and design proposals, fostering collaboration between Google's initial team and external contributors.[3]
- **LLVM Discourse:** The primary platform for online discussion, questions, proposals (RFCs), and announcements related to MLIR and LLVM.[16] This forum replaced older mailing lists, offering better organization and features.
- **Tutorials and Documentation:** While the official MLIR documentation provides language references and some tutorials (e.g., Toy language, mlir-opt usage, dialect creation) [8], the rapid pace of development means documentation and introductory materials can sometimes lag.[92] Community members and projects like HEIR often contribute additional tutorials and talks.[29]

These avenues facilitate knowledge sharing, collaborative design, and the growth of the MLIR user and developer base.

8. Comparative Analysis and Future Outlook

8.1 Comparing MLIR-based Approaches

The flexibility inherent in MLIR means that there isn't a single, monolithic "MLIR approach." Instead, different projects and vendors leverage the infrastructure in diverse ways, leading to varied architectural patterns:

- **MLIR vs. Precursors/Alternatives (TVM, Glow):** Projects like Apache TVM and Facebook's Glow were early pioneers in ML compilation, addressing the need for optimizing framework graphs for diverse hardware.[55] TVM, in particular, introduced influential concepts like the separation of algorithm and schedule [46] and employed techniques like autotuning extensively.[56] However, TVM faced challenges in keeping pace with rapidly evolving hardware (especially specialized units like Tensor Cores), suffered from fragmentation as vendors created incompatible forks, and its development slowed relative to framework evolution.[56] MLIR, emerging slightly later, focused heavily on providing a robust, multi-level *infrastructure* with dialects, aiming for greater modularity and extensibility from the outset.[7] While TVM and Glow were initially more focused on being end-to-end solutions, MLIR positioned itself as a framework *for building* such solutions.[94] There is potential for interoperability, perhaps by defining TVM dialects within MLIR or translating between their respective IRs.[94] Recent research also suggests that MLIR-based autotuning approaches (potentially leveraging the Transform dialect) might achieve comparable results with significantly fewer samples than TVM's methods.[95]
- **Hardware Backend Strategies:** Hardware vendors exhibit different MLIR adoption strategies. Tenstorrent represents a deep, MLIR-native approach, building its entire TT-Forge compiler around custom MLIR dialects.[58] NVIDIA integrates MLIR more incrementally, using standard (gpu) and custom (nvgpu) dialects as intermediate layers above its existing, mature NVVM IR and PTX generation backend.[26] Graphcore appears to use MLIR as a component for specific high-level optimizations within its broader, C++-based Poplar graph compiler framework, which relies on LLVM for core-level code generation.[71] AMD employs MLIR strategically across different product lines, developing specific backends (Peano for NPUs [15]) and full compilation flows (ARIES for AIEs [2]).
- **Data Pipeline Strategies:** For data processing, the DAPHNE project exemplifies an ambitious approach, using MLIR to build an integrated system covering ETL, ML, and HPC.[34] A more common, perhaps pragmatic, approach involves using MLIR to optimize specific compute-intensive kernels *within* a larger, traditional ETL workflow managed by tools like Spark or Airflow.

This diversity demonstrates MLIR's adaptability. It functions as a versatile toolkit rather than a prescriptive solution. Different users select and combine MLIR's components (dialects, passes, infrastructure) based on their specific needs, legacy systems, and target domains, leading to varied integration depths and architectural choices.

8.2 Synthesizing Major Trends and Breakthroughs (Last 2-3 Years)

Analyzing the developments from 2022-2025 reveals several significant trends and breakthroughs shaping the MLIR landscape:

- **Trend 1: Standardization via Interfaces:** A clear trend is the push towards standardizing the interfaces between ML frameworks and MLIR-based compilers. **StableHLO** is emerging as the de facto standard *input dialect* for compilers like XLA and IREE, promoting framework portability.[52] Simultaneously, **PJRT** is gaining traction as the standard *runtime interface*, allowing frameworks to dynamically load and interact with different hardware backends in a plug-and-play manner.[59]
- **Trend 2: Proliferation of Hardware-Specific Dialects:** As more hardware vendors adopt MLIR, there is a corresponding increase in the creation of custom, vendor-specific dialects (e.g., nvgpu, amdgpu, TTIR, TPU, ADF) designed to expose unique hardware features and enable targeted optimizations within the MLIR framework.[2]
- **Trend 3: Rise of Explicit Compiler Control:** The development and application of the **Transform dialect** represent a significant shift towards giving performance engineers direct, programmable control over the compilation process.[46] Its successful use in debugging, fine-grained optimization, and autotuning integration indicates growing adoption.
- **Trend 4: Broadening Scope:** MLIR's application space is expanding considerably beyond its initial focus on core ML model compilation. Active research and development are applying MLIR to **data analysis pipelines** (DAPHNE [34]), **hardware design and synthesis** (CIRCT [22]), **quantum computing** (CUDA Quantum [27]), and **homomorphic encryption** (HEIR [29]).
- **Breakthrough 1: Achieving Critical Mass:** MLIR has firmly established itself as the foundational compiler infrastructure underpinning major industry efforts in AI compilation, including OpenXLA, Torch-MLIR, and numerous vendor-specific toolchains. Its adoption by key players across the hardware and software spectrum signifies it has reached critical mass.
- **Breakthrough 2: Demonstrating Performance:** MLIR-based compilation techniques have proven capable of generating highly optimized code, achieving performance close to theoretical hardware peaks for critical computational kernels like GEMM, demonstrating its viability for high-performance computing tasks.[60]

8.3 Future Directions and Challenges

Despite its successes, MLIR faces ongoing challenges and has clear areas for future development:

- **Addressing Fragmentation and Identity:** The "dialect explosion" and the

ambiguity between MLIR as core infrastructure versus an AI solution require careful management.[3] Continued efforts in community governance, potentially through mechanisms like the LLVM Area Teams, are needed to ensure coherence, manage dialect contributions effectively, and perhaps clarify the boundaries between the domain-agnostic core and domain-specific extensions.[3] Robust mechanisms for dialect versioning and ensuring compatibility between different MLIR components (framework frontends, dialects, backends) are crucial to avoid issues like the TOSA v1.0 breakage.[61]

- **Improving Usability and Accessibility:** While powerful, MLIR currently requires significant compiler expertise.[47] Making the infrastructure more accessible to domain experts (e.g., ML researchers, data scientists) who are not compiler specialists is important for broader adoption. This could involve developing higher-level abstractions, improving tooling, enhancing documentation and tutorials, or further developing programmable interfaces like the Transform dialect.[47]
- **Maturing Data Pipeline Integration:** While projects like DAPHNE and Substrait-MLIR show promise, MLIR's capabilities for handling the full spectrum of ETL tasks (especially data extraction, complex cleaning, orchestration) need further development to compete with dedicated data engineering frameworks.[13] Defining more comprehensive dialects or libraries for common data processing tasks could be beneficial.
- **Enhancing End-to-End Optimization:** Realizing the full potential of MLIR requires enabling more holistic optimizations that span across different dialects, abstraction levels, and pipeline stages (e.g., co-optimizing data layout based on compute patterns, fusing data transformations with model layers). This requires sophisticated analyses and transformation capabilities that can operate across dialect boundaries.
- **Debugging and Verification:** As compilation flows become more complex, involving multiple dialects and intricate lowering paths, robust tools and techniques for debugging transformations and verifying the correctness of the generated code are essential.[5] Continued research in areas like MLIR fuzzing and formal verification is needed.

MLIR has successfully established a powerful and flexible foundation. The next phase of its evolution will likely focus on refining the ecosystem built upon this foundation, improving the developer experience, enhancing its capabilities in adjacent domains like data processing, and tackling the complexities arising from its own success to fully realize its potential as a unifying force in compilation technology.

9. Conclusion

Over the past three years, Multi-Level Intermediate Representation (MLIR) has rapidly transitioned from a promising research project to a cornerstone technology underpinning the modern AI compilation landscape. Its emergence was driven by the fundamental need for a more flexible, extensible, and modular compiler infrastructure capable of handling the growing complexity of AI models and the increasing diversity of hardware accelerators in the post-Moore's Law era.

MLIR's core contribution lies in its dialect-based architecture, which enables the representation of computation at multiple levels of abstraction within a single, unified framework. This has proven instrumental in bridging the gap between high-level AI frameworks (TensorFlow, PyTorch, JAX) and the specifics of hardware targets ranging from NVIDIA and AMD GPUs to specialized accelerators like Tenstorrent IPUs, AMD NPUs and AIEs, and Sophgo TPUs. Vendors are increasingly leveraging MLIR to build specialized compilers, often defining custom dialects to expose unique hardware capabilities, thereby accelerating the enablement of standard ML frameworks on their platforms.

Key trends during this period include a concerted effort towards standardization through common interfaces like StableHLO and PJRT, aiming to decouple frameworks from backends and enhance portability. Concurrently, the proliferation of hardware-specific dialects highlights MLIR's role in enabling hardware innovation. The development and application of the Transform dialect mark a significant advancement, offering performance engineers unprecedented fine-grained, programmable control over the compilation process itself. Furthermore, MLIR's scope has demonstrably broadened beyond core ML compilation, with active research and development extending its application to data analysis pipelines, hardware design, quantum computing, and cryptography.

While MLIR's foundational role appears secure, challenges remain. Managing the complexity and potential fragmentation arising from its own extensibility, improving usability for a wider range of developers, deepening its integration with data processing workflows, and enhancing end-to-end optimization capabilities are critical areas for future work. Nonetheless, MLIR has fundamentally reshaped the compiler landscape for AI and heterogeneous computing. Its trajectory indicates it will continue to be a driving force behind innovation, enabling the efficient deployment of increasingly sophisticated AI models on current and future generations of computing hardware.

Works cited

1. MLIR: A Compiler Infrastructure for the End of Moore's Law - AI Resources - Modular, accessed April 15, 2025, https://www.modular.com/ai-resources/mlir-a-compiler-infrastructure-for-the-end-of-moore-s-law
2. www.csl.cornell.edu, accessed April 15, 2025, https://www.csl.cornell.edu/~zhiruz/pdfs/aries-fpga2025.pdf
3. Democratizing AI Compute, Part 8: What about the MLIR compiler infrastructure? - Modular, accessed April 15, 2025, https://www.modular.com/blog/democratizing-ai-compute-part-8-what-about-the-mlir-compiler-infrastructure
4. MLIR Part 1 - Introduction to MLIR and Modern Compilers - Stephen Diehl, accessed April 15, 2025, https://www.stephendiehl.com/posts/mlir_introduction/
5. MLIR generic representation for polynomial multiplication using affine... - ResearchGate, accessed April 15, 2025, https://www.researchgate.net/figure/MLIR-generic-representation-for-polynomial-multiplication-using-affine-and-std-dialects_fig2_349993972
6. A Method for Efficient Heterogeneous Parallel Compilation: A Cryptography Case Study, accessed April 15, 2025, https://arxiv.org/html/2407.09333v2
7. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation - Reliable Computer Systems - University of Waterloo, accessed April 15, 2025, https://rcs.uwaterloo.ca/~ali/cs842-s23/papers/mlir.pdf
8. MLIR Language Reference, accessed April 15, 2025, https://mlir.llvm.org/docs/LangRef/
9. MLIR - TensorFlow, accessed April 15, 2025, https://www.tensorflow.org/mlir
10. DESIL: Detecting Silent Bugs in MLIR Compiler Infrastructure - arXiv, accessed April 15, 2025, https://arxiv.org/html/2504.01379v1
11. [2210.15016] TPU-MLIR: A Compiler For TPU Using MLIR - ar5iv - arXiv, accessed April 15, 2025, https://ar5iv.labs.arxiv.org/html/2210.15016
12. MLIR CodeGen Dialects for Machine Learning Compilers - Lei.Chat(), accessed April 15, 2025, https://www.lei.chat/posts/mlir-codegen-dialects-for-machine-learning-compilers/
13. substrait-io/substrait-mlir-contrib - GitHub, accessed April 15, 2025, https://github.com/substrait-io/substrait-mlir-contrib
14. Defining Dialects - MLIR, accessed April 15, 2025, https://mlir.llvm.org/docs/DefiningDialects/
15. LLVM Had Another Exciting Year With More Than 37k Commits, 35.5 Million Lines, accessed April 15, 2025, https://www.phoronix.com/news/LLVM-Code-Activity-2024
16. Discourse Migration Guide — LLVM 19.0.0git documentation, accessed April 15, 2025, https://rocm.docs.amd.com/projects/llvm-project/en/docs-6.4.0/LLVM/llvm/html/DiscourseMigrationGuide.html

17. 2025 European LLVM Developers' Meeting - Swoogo, accessed April 15, 2025, https://llvm.swoogo.com/2025eurollvm/
18. Matthias Springer's Homepage, accessed April 15, 2025, https://m-sp.org/
19. Building a JSONiq Query Optimizer using MLIR - Research Collection, accessed April 15, 2025, https://www.research-collection.ethz.ch/bitstream/handle/20.500.11850/460014/thesis-mfiebig.pdf
20. Dialects - MLIR, accessed April 15, 2025, https://mlir.llvm.org/docs/Dialects/
21. Should Julia use MLIR in the future? - Internals & Design, accessed April 15, 2025, https://discourse.julialang.org/t/should-julia-use-mlir-in-the-future/110459
22. Hardware.jl — An MLIR-based Julia HLS Flow (Work in Progress) - arXiv, accessed April 15, 2025, https://arxiv.org/html/2503.09463v1
23. Hardware.jl — An MLIR-based Julia HLS Flow (Work in Progress) - Capra, accessed April 15, 2025, https://capra.cs.cornell.edu/latte25/paper/5.pdf
24. Quickstart tutorial to adding MLIR graph rewrite, accessed April 15, 2025, https://mlir.llvm.org/docs/Tutorials/QuickstartRewrites/
25. Creating a Dialect - MLIR, accessed April 15, 2025, https://mlir.llvm.org/docs/Tutorials/CreatingADialect/
26. 'gpu' Dialect - MLIR, accessed April 15, 2025, https://mlir.llvm.org/docs/Dialects/GPU/
27. Create your Own MLIR Pass — NVIDIA CUDA Quantum documentation - GitHub Pages, accessed April 15, 2025, https://nvidia.github.io/cuda-quantum/0.4.0/using/advanced/mlir_pass.html
28. cuda-quantum/Overview.md at main - GitHub, accessed April 15, 2025, https://github.com/NVIDIA/cuda-quantum/blob/main/Overview.md
29. Tutorials and Talks - HEIR, accessed April 15, 2025, https://heir.dev/docs/tutorials/
30. Optimizing ETL Pipelines: Best Practices, Tools & Architecture for Efficient Data Workflow, accessed April 15, 2025, https://www.acceldata.io/blog/etl-pipelines-key-concepts-components-and-best-practices
31. How to Build ETL Data Pipeline in ML - Neptune.ai, accessed April 15, 2025, https://neptune.ai/blog/build-etl-data-pipeline-in-ml
32. tf.data: A Machine Learning Data Processing Framework - VLDB Endowment, accessed April 15, 2025, https://vldb.org/pvldb/vol14/p2945-klimovic.pdf
33. How to Optimize Your ETL Pipeline for Maximum Efficiency - DEV Community, accessed April 15, 2025, https://dev.to/chainguns/how-to-optimize-your-etl-pipeline-for-maximum-efficiency-3b56
34. www.cidrdb.org, accessed April 15, 2025, https://www.cidrdb.org/cidr2022/papers/p4-damme.pdf
35. [2101.12127] tf.data: A Machine Learning Data Processing Framework, accessed April 15, 2025, https://ar5iv.labs.arxiv.org/html/2101.12127
36. Data Preprocessing and Data Cleaning. | by Prabesh Sharma | Medium, accessed April 15, 2025, https://medium.com/@sharmaprabesh027/data-preprocessing-and-data-cleanin

g-de318cb7b1b5
37. Multilingual Information Retrieval | PDF - Scribd, accessed April 15, 2025, https://www.scribd.com/document/689704273/Multilingual-Information-Retrieval
38. Data Cleaning and Preprocessing in Machine Learning - CodeSignal, accessed April 15, 2025, https://codesignal.com/learn/courses/data-cleaning-and-preprocessing-in-machine-learning
39. 9 Preprocessing – Applied Machine Learning Using mlr3 in R, accessed April 15, 2025, https://mlr3book.mlr-org.com/chapters/chapter9/preprocessing.html
40. Chapter 3: High-level Language-Specific Analysis and Transformation - MLIR, accessed April 15, 2025, https://mlir.llvm.org/docs/Tutorials/Toy/Ch-3/
41. Memory Optimization and Profiling for MLIR-Based HeteroCL - CS@Cornell, accessed April 15, 2025, https://www.cs.cornell.edu/courses/cs6120/2022sp/blog/hcl-mlir/
42. MLIR-based Data Tiling and Packing for Ryzen AI NPU - FOSDEM 2025, accessed April 15, 2025, https://fosdem.org/2025/schedule/event/fosdem-2025-6641-mlir-based-data-tiling-and-packing-for-ryzen-ai-npu/
43. LTO and Data Layout Optimizations in MLIR - LLVM.org, accessed April 15, 2025, https://llvm.org/devmtg/2021-02-28/slides/Prashantha-MLIR-LTO.pdf
44. Using `mlir-opt`, accessed April 15, 2025, https://mlir.llvm.org/docs/Tutorials/MlirOpt/
45. MLIR — A Global Optimization and Dataflow Analysis - Math ∩ Programming, accessed April 15, 2025, https://www.jeremykun.com/2023/11/15/mlir-a-global-optimization-and-dataflow-analysis/
46. The MLIR Transform Dialect - arXiv, accessed April 15, 2025, https://arxiv.org/html/2409.03864v2
47. The MLIR Transform Dialect: Your Compiler Is More Powerful Than You Think - Michel Steuwer, accessed April 15, 2025, https://michel.steuwer.info/files/publications/2025/CGO-2025-2.pdf
48. The MLIR Transform Dialect - Your compiler is more powerful than you think - CGO 2025, accessed April 15, 2025, https://2025.cgo.org/details/cgo-2025-papers/7/The-MLIR-Transform-Dialect-Your-compiler-is-more-powerful-than-you-think
49. www.arxiv.org, accessed April 15, 2025, https://www.arxiv.org/pdf/2409.03864v2
50. [2409.03864] The MLIR Transform Dialect. Your compiler is more powerful than you think, accessed April 15, 2025, https://arxiv.org/abs/2409.03864
51. 2023 EuroLLVM - Tutorial: Controllable Transformations in MLIR - YouTube, accessed April 15, 2025, https://www.youtube.com/watch?v=P4gUj3QtH_Y
52. XLA architecture - OpenXLA Project, accessed April 15, 2025, https://openxla.org/xla/architecture
53. XLA - OpenXLA Project, accessed April 15, 2025, https://openxla.org/xla
54. OpenXLA is available now to accelerate and simplify machine learning | Google Open Source Blog, accessed April 15, 2025,

https://opensource.googleblog.com/2023/03/openxla-is-ready-to-accelerate-and-simplify-ml-development.html
55. Accelerating ML through Compilation: Building an ML Compiler that Works | HTEC, accessed April 15, 2025, https://htec.com/insights/accelerating-ml-through-compilation-building-ml-compiler-that-works/
56. Democratizing AI Compute, Part 6: What about AI compilers (TVM and XLA)? - Modular, accessed April 15, 2025, https://www.modular.com/blog/democratizing-ai-compute-part-6-what-about-ai-compilers
57. openxla/xla: A machine learning compiler for GPUs, CPUs, and ML accelerators - GitHub, accessed April 15, 2025, https://github.com/openxla/xla
58. TT-Forge™ - Tenstorrent, accessed April 15, 2025, https://tenstorrent.com/en/software/tt-forge
59. PJRT: Simplifying ML Hardware and Framework Integration | Google Open Source Blog, accessed April 15, 2025, https://opensource.googleblog.com/2023/05/pjrt-simplifying-ml-hardware-and-framework-integration.html
60. MLIR-based Code Generation for High-Performance Machine Learning on AArch64 - Lund University Publications, accessed April 15, 2025, https://lup.lub.lu.se/student-papers/record/9146373/file/9146374.pdf
61. Support for LiteRT (TensorFlow Lite, .tflite) with TOSA 1.0 · Issue ..., accessed April 15, 2025, https://github.com/iree-org/iree/issues/19777
62. torch-mlir/docs/architecture.md at main · llvm/torch-mlir · GitHub, accessed April 15, 2025, https://github.com/llvm/torch-mlir/blob/main/docs/architecture.md
63. JAX Integration Completeness Milestone - GitHub, accessed April 15, 2025, https://github.com/openxla/iree/milestone/33
64. An introduction to Torch-MLIR - FOSDEM 2025, accessed April 15, 2025, https://fosdem.org/2025/schedule/event/fosdem-2025-6643-an-introduction-to-torch-mlir/
65. Quickstart - JAX documentation, accessed April 15, 2025, https://docs.jax.dev/en/latest/quickstart.html
66. JAX and OpenXLA Part 1: Run Process and Underlying Logic - Intel, accessed April 15, 2025, https://www.intel.com/content/www/us/en/developer/articles/technical/jax-openxla-running-process-and-underlying-logic-1.html
67. JAX and OpenXLA Part 1: Run Process and Underlying Logic - Intel, accessed April 15, 2025, https://www.intel.com/content/www/us/en/developer/articles/technical/jax-and-openxla-run-process-and-underlying-logic-1.html
68. MLIR Sparsifier - MPACT Research Group | Google for Developers, accessed April 15, 2025, https://developers.google.com/mlir-sparsifier/colabs/Sparse_JAX_CPU_Benchmark_Colab
69. AMD Talks Up IREE/MLIR Programming For Ryzen AI NPUs - Reddit, accessed April

15, 2025, https://www.reddit.com/r/Amd/comments/1ij56pd/amd_talks_up_ireemlir_programming_for_ryzen_ai/
70. Democratizing AI Compute, Part 4: CUDA is the incumbent, but is it any good? - Modular, accessed April 15, 2025, https://www.modular.com/blog/democratizing-ai-compute-part-4-cuda-is-the-incumbent-but-is-it-any-good
71. C4ML 2021, accessed April 15, 2025, https://www.c4ml.org/c4ml-2021
72. tenstorrent/tt-mlir - GitHub, accessed April 15, 2025, https://github.com/tenstorrent/tt-mlir
73. Roadmap - tt-mlir documentation, accessed April 15, 2025, https://docs.tenstorrent.com/tt-mlir/tt-explorer-roadmap.html
74. CUDA LLVM Compiler - NVIDIA Developer, accessed April 15, 2025, https://developer.nvidia.com/cuda-llvm-compiler
75. 1. Introduction — NVVM IR Specification 12.8 documentation - NVIDIA Docs, accessed April 15, 2025, https://docs.nvidia.com/cuda/nvvm-ir-spec/
76. 'nvgpu' Dialect - MLIR, accessed April 15, 2025, https://mlir.llvm.org/docs/Dialects/NVGPU/
77. Highlights - NVIDIA, accessed April 15, 2025, https://images.nvidia.com/nvimages/gtc/pdf/GTC2025_Highlights.pdf
78. Nvidia adds native Python support to CUDA - Hacker News, accessed April 15, 2025, https://news.ycombinator.com/item?id=43581584
79. LLVM DISTRIBUTORS CONFERENCE 2021 - GitHub, accessed April 15, 2025, https://raw.githubusercontent.com/ClangBuiltLinux/llvm-distributors-conf-2021/main/slides/graphcore.pdf
80. IPU Processors - Graphcore, accessed April 15, 2025, https://www.graphcore.ai/products/ipu
81. 1. Introduction — Poplar SDK Overview - Graphcore Documents, accessed April 15, 2025, https://docs.graphcore.ai/projects/sdk-overview/en/latest/overview.html
82. Poplar® Software - Graphcore, accessed April 15, 2025, https://www.graphcore.ai/products/poplar
83. POPLAR OVERVIEW - Graphcore, accessed April 15, 2025, https://www.graphcore.ai/hubfs/assets/Poplar%C2%81%20technical%20overview%20NEW%20BRAND.pdf
84. graphcore/poplibs: Poplar libraries - GitHub, accessed April 15, 2025, https://github.com/graphcore/poplibs
85. 5.1. Poplar Tutorial 1: Programs and Variables - Graphcore Documents, accessed April 15, 2025, https://docs.graphcore.ai/projects/tutorials/en/latest/poplar/tut1_variables/README.html
86. 1. Introduction — Poplar and PopLibs User Guide - Graphcore Documents, accessed April 15, 2025, https://docs.graphcore.ai/projects/poplar-user-guide/en/latest/introduction.html
87. Machine learning compiler based on MLIR for Sophgo TPU. - GitHub, accessed April 15, 2025, https://github.com/sophgo/tpu-mlir

88. onnx/onnx-mlir: Representation and Reference Lowering of ... - GitHub, accessed April 15, 2025, https://github.com/onnx/onnx-mlir
89. intel/graph-compiler: MLIR-based toolkit targeting intel heterogeneous hardware - GitHub, accessed April 15, 2025, https://github.com/intel/graph-compiler
90. HETOCompiler: An MLIR-based crypTOgraphic Compilation Framework for HEterogeneous Devices - arXiv, accessed April 15, 2025, https://arxiv.org/html/2407.09333v1
91. High Performance Code Generation in MLIR: An early case study with GEMM - YouTube, accessed April 15, 2025, https://www.youtube.com/watch?v=boXI7rmaasU
92. MLIR — Getting Started - Math ∩ Programming, accessed April 15, 2025, https://www.jeremykun.com/2023/08/10/mlir-getting-started/
93. Compilers: Talking to The Hardware - Unify AI, accessed April 15, 2025, https://unify.ai/blog/deep-learning-compilers
94. Google lasted work: MLIR Primer - Development - Apache TVM Discuss, accessed April 15, 2025, https://discuss.tvm.apache.org/t/google-lasted-work-mlir-primer/1721
95. ML2Tuner: Efficient Code Tuning via Multi-Level Machine Learning Models - arXiv, accessed April 15, 2025, https://arxiv.org/html/2411.10764v1