International Journal of Science and Research Archive

eISSN: 2582-8185

Cross Ref DOI: 10.30574/ijsra

Journal homepage: https://ijsra.net/

(REVIEW ARTICLE)

Check for updates

Enhancing Compiler Design for Machine Learning Workflows with MLIR

Ankush Jitendrakumar Tyagi *

University of Texas at Arlington, Texas, USA.

International Journal of Science and Research Archive, 2025, 16(02), 1397-1405

Publication history: Received on 15 July 2025; revised on 23 August; accepted on 26 August 2025

Article DOI: https://doi.org/10.30574/ijsra.2025.16.2.2463

Abstract

The rapidly changing world of machine learning (ML) workloads has added a tremendous burden on our conventional compiler infrastructures, and these systems frequently lack the flexibility, scalability, and optimization options needed to address the emerging, heterogeneous computing landscape. As ML frameworks continue to expand their support to a wide variety of hardware platforms (such as GPUs, TPUs, or FPGAs), it is critical that the corresponding compiler infrastructures accommodate the now-requisite levels of both high-level algorithmic abstractions and low-level hardware-specific optimizations. Google has developed the Multi-Level Intermediate Representation, the Multi-Level, which solves these problems by delivering a modular, extensible compiler infrastructure, optimized to the modern ML development workflow. MLIR facilitates the specification of domain-specific intermediate representations (IRs) and thus allows optimizations at many abstraction levels, including tensor algebra to hardware-specific instruction sets. This architecture enables control of code transformation at the finest grain, enables custom dialects, and has encouraged interoperability among popular ML frameworks (TensorFlow, PyTorch, JAX, and ONNX). Using MLIR as a compiler design aspect, developers gain both better portable compilation and higher optimization granularity, as well as time-reducing development. In addition, MLIR progresses in the realization of AI-specific compiler optimizations, including quantization, operator fusion, and parallel execution scheduling. Finally, MLIR is an important step in compiler design that will not only deliver more performance on ML applications, but also a more maintainable and extensible ecosystem for future AI-driven computing systems. This paper discusses the new paradigm offered by MLIR, its ability to redefine compiler infrastructure to satisfy the sophisticated needs of present ML workloads, and the avenue to more intelligent, flexible, and resource-efficient software-hardware integration.

Keywords: MLIR (Multi-Level Intermediate Representation); Compiler Optimization; Machine Learning Workflows; Heterogeneous Hardware Compilation; Domain-Specific Dialects

1. Introduction

With the increasing popularity of industries being revolutionized by the use of machine learning (ML) in healthcare, financial, autonomous systems, and natural language processing, an ever-increasing amount of complexity has been put into the software and hardware that support these processes. At the heart of these systems is the compiler, the layer that performs the conversion of high-level texts to optimally distributed, machine-specific instructions that execute on any of many computing systems. Historically, general-purpose programming-based compilers were created, which have limited capabilities to accommodate the multidimensional and dynamic nature of ML workloads [1-3].

As the hardware became more and more heterogeneous (PCPUs, GPUs, TPUs, and FPGAs), ML developers now also have to face the challenge of having to write performant code to a widening number of platforms. This is further complicated by the fact that ML pipelines are multilayered, covered by such concerns as tensor operations and algebraic transformations, memory buffers, and instruction-level parallelism. These complex workloads are frequently beyond the flexibility, extensibility, and modularity capabilities of traditional compiler systems [4-5].

* Corresponding author: Ankush Jitendrakumar Tyagi

Aiming at filling this gap, Google proposed a new semantics-rich compiler infrastructure, Multi-Level Intermediate Representation (MLIR), that has as its goal the unification and extension of all code definitions, transformations, and optimizations at any level of abstraction. MLIR also promises to be a transformative technology in the ML computing sector when used to compile ML programs, requiring modularity coupled with extensibility in compiler design in order to optimize to contemporary hardware as well as the higher-level abstractions desired by system ML frameworks.

This paper describes MLIR and its benefits to compiler design to enable modern machine learning pipelines. It explains the weaknesses of classical compiler stacks, gives an overview of the MLIR architecture at length, outlines the advantages of modular intermediate representations, and looks at use cases of integrating it with popular ML frameworks and hardware back-ends.

2. Challenges in Traditional Compiler Design for Machine Learning

LLVM and GCC are typical of traditional compilers that have been designed to be used in general-purpose programming and appear on CPU-based architectures. Although they provide mature toolchains and strong optimizations of scalar computations, they do not include built-in mechanisms to address the tensor operations and data parallelism or heterogeneous execution model typical of ML workloads [1]. In ML applications, computation graphs can involve chaining together multiple high-level operators, such as convolutions, matrix multiplications, and activations, that are vastly different than the flat control-flow graphs used in usual compiler representations. Consequently, these compilers will need to use external graph optimizers, and this adds complexity to the toolchain and makes the process inefficient because the same transformation is repeatedly performed and serialized between representations [2]. Table 1 highlights the key limitations of traditional compiler design when applied to machine learning workloads. It summarizes how classical approaches fall short in handling ML-specific challenges such as hardware diversity, dynamic computation graphs, and large-scale parallelism.

Besides, there is little optimization in different layers of abstraction. The traditional compilers lack a single representation that can be used to transform at both the high-level tensor algebra and the low-level hardware instructions. Such a segmentation results in duplication of efforts, more time to compile, and not-optimal functioning because of the lack of cross-layer optimizations [3]. The adaptability of the old-fashioned compilers is also a burden to implement in new hardware. As an example, the ability to target a custom accelerator, such as TPU or ASIC, usually needs large backend engineering, so it can be challenging to rapidly prototype and deploy. This is of concern, especially in ML, where hardware progress is faster than software foundations [4].

Such limitations have caused a new direction to be necessary, a push towards multi-level representations, modular optimization passes, and domain- or hardware-specific dialects. It was exactly in order to meet these needs that there was developed of MLIR.

Table 1 Challenges in Traditional Compiler Design for Machine Learning

Challenge	Description	Why It Matters in ML
Complexity of ML Workloads	Traditional compilers were designed for general-purpose programs, not large-scale tensor computations.	ML models involve massive parallelism and irregular operations that conventional compilers struggle to optimize.
Hardware Diversity	Classical compilers target CPUs primarily, with limited GPU/TPU/FPGA support.	ML requires heterogeneous hardware utilization; lack of support limits performance portability.
Dynamic Computation Graphs	Traditional compilers assume static control flows and memory allocation.	Frameworks like PyTorch use dynamic graphs, which require runtime flexibility beyond static compilation.
Optimization Scalability	Conventional optimization passes (e.g., loop unrolling, SSA form) don't scale well for huge tensor operations.	Inefficient optimizations increase training time and inference latency in ML pipelines.
Data Movement Overheads	Compilers traditionally optimize compute, not memory bandwidth and communication.	ML workloads are often memory-bound; data locality and movement dominate execution time.

Lack of Domain-Specific Abstractions	Generic IRs (Intermediate Representations) fail to capture high-level ML operators.	MLIR, XLA, and TVM emerged to provide ML-focused abstractions missing in classical compilers.
Parallelism Management	Compilers for general-purpose code don't handle distributed or large-scale parallel training efficiently.	ML requires scaling across clusters, GPUs, and accelerators with minimal overhead.
Energy Efficiency	Traditional compilers prioritize execution speed without considering power constraints.	In ML, especially edge computing, optimizing for energy efficiency is critical.
Debugging and Profiling Limitations	Legacy tools focus on sequential programs, not large ML models.	Developers need insight into performance bottlenecks across layers, operators, and devices.

3. Overview of MLIR: Architecture and Key Features

MLIR is a revolutionary step towards compiler infrastructure: a Swiss army knife of compilers that provides superior extensibility and modularity and reinvents the notion of intermediate representation (IR) in the context of modern machine learning (ML). Basically, MLIR introduces a framework of flexible elements, namely operations, types, and attributes, which could be combined together into reusable composable dialects. Each of the dialects contains domain-specific semantics over a common infrastructure, thereby facilitating modularity, interoperability, and integration within different domains and across different layers of compilers [5]. The multi-level IR design is one of the groundbreaking elements of MLIR because it allows representations at different levels of abstraction. The design also allows developers to specify high-level tensor-centric operations, which are sequentially lowered to intermediate and low-level representations optimized to be executed on specific hardware targets. As an example, a complicated task like a matrix multiplication can first be described and represented in the higher-level tensor dialect and later transformed into instruction sequences for hardware specifications (e.g., GPUs, TPUs, or CPUs) by lowering passes [6]. Figure 1 illustrates the MLIR (Multi-Level Intermediate Representation) framework, connecting high-level frontends like TensorFlow and PyTorch through core dialects and transformations to backend hardware targets such as CPUs, GPUs, and TPUs. Key features like extensibility, hardware support, and optimizations are annotated as benefits of this architecture.

The framework facilitates a few important attributes that make the framework flexible and efficient. First, MLIR allows the developer of compilers to create custom dialects where it is possible to extend to certain workloads or hardware targets, depending on what is required. ML languages designed to support accurate modeling of the computations that the framework executes include the tensor Tensor, Linalg, and GPU. Second, MLIR has a dynamic lowering process, in which operations are lowered by proceeding through more concrete representations, through a series of lowering passes. It is then possible to make optimizations at suitable levels of abstraction to achieve maximum performance and still retain semantic fidelity. Third, MLIR adds a modular pass infrastructure that accommodates configuration, reusability, and composition of transformation passes to suit the needs of diverse compilation pipelines. Last, MLIR is patterned after the Static Single Assignment (SSA) form, a long-standing, compiler-friendly representation that facilitates data flow analysis and optimization by assuring each variable is used single-assigned [4-7].

Notably, the architecture of MLIR is meant to supplement and expand the current compilation technologies. It is compatible with existing compilers like LLVM and offers an LLVM dialect to allow an easy move of the MLIR IRs into executable machine code. In addition, MLIR is a good intermediate layer to vendors of ML frameworks because they can directly export to MLIR representations of their frontend computation graphs (e.g., TensorFlow). This decreases the requirement to have duplicate serialization passes and allows more intensive end-to-end optimizations in one unified pipeline. Consequently, MLIR allows not only to improve performance and maintainability but also to make the creation of hardware-optimized ML applications much speedier.

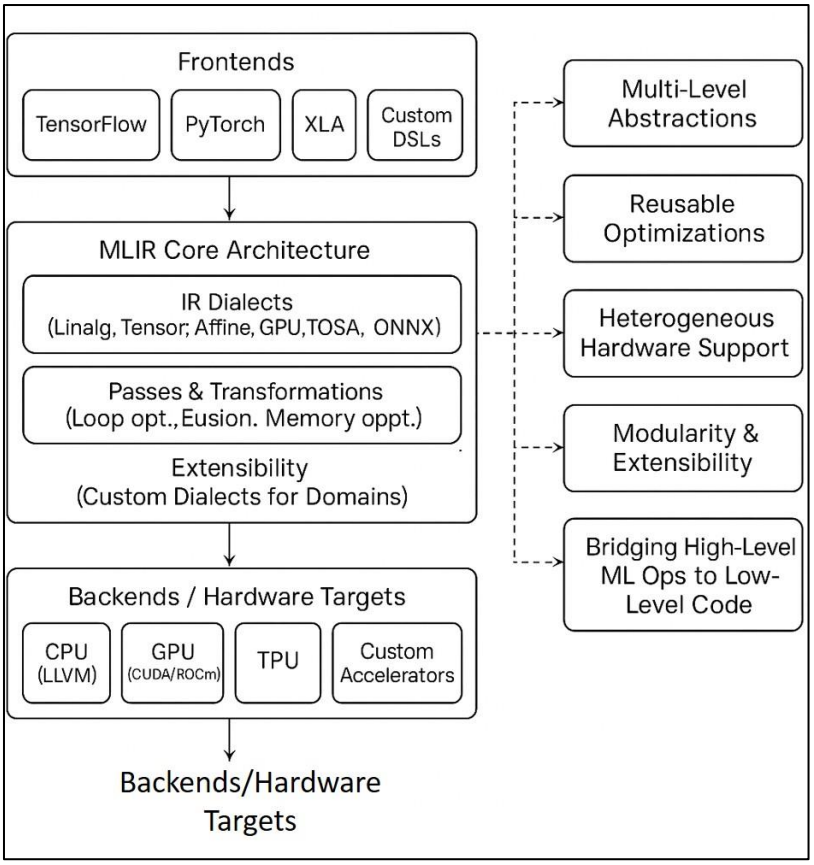

Figure 1 MLIR Architecture Flowchart

4. Modular Intermediate Representations for Multi-Level Optimization

One distinctive feature of MLIR is its model of multi-level abstraction, allowing compiler developers the opportunity to represent, analyze, and optimize machine learning (ML) workloads at multiple levels of semantic looseness in a single infrastructure. Unlike existing intermediate representations (IRs) working at a specific abstraction level, MLIR enables transformation between high-level algebraic operations and low-level hardware-targeted instructions and forms an extensible and flexible optimization pipeline [8]. MLIR also defines dialects (at the highest level), including Tensor, Linalg, and Affine, which abstract over tensor operations and loop constructs in a hardware-agnostic fashion. These dialects are aligned well with the constructs of ML frameworks, such as TensorFlow or Pytorch, and this alignment simplifies other transformations, including reshaping tensors, tiling them, or simplifying them algebraically, but without being prematurely bound to hardware specifics. A structured representation of linear algebra computation is readily available, for instance, using the Linalg dialect, and the compiler can use mathematical properties such as commutativity and associativity to perform more optimising transformations [9][10]. Table 2 outlines how modular intermediate representations enable multi-level optimization across high-level, mid-level, and low-level abstractions in machine learning compilers. By structuring optimizations at different layers, compilers can bridge semantic-rich ML operations with efficient hardware execution, ensuring scalability, portability, and performance.

Passing the computation levels down the compilation pipeline, MLIR switches to an intermediate representation in the form of dialects such as the Structured Control Flow (SCF) and MemRef dialects. Such representations preserve enough semantic richness to benefit targeted optimizations and cause the IR to become less difficult to constrain by a hardware back-end. This is where the compiler applies loop unrolling, pipeline optimizations on the software, and optimization of the layouts on memory, among others, which would be important transformations to minimize the execution time and also to boost memory efficiency. Domain-specific operations, including layout transformations and operator fusion that are important in performance-critical ML pipelines, are also possible with mid-level IRs [11].

The end of the transformation process is to be reduced to lower-level dialects, including the LLVM dialect, GPU dialect, or other lower-level hardware-specific targets. These dialects reveal low-level transformations required to generate backend code, such as instruction selection, register allocation, and control flow transformation operations, which are

close to the typical compiler backend operations [12]. MLIR has the ability to define and inject custom dialects at any level of the hierarchy and can thereby provide the flexibility to the hardware developer or compiler engineers to optimize and customize the code generation path to meet the requirements of new architectures.

Importantly, the modular pass manager in MLIR permits an on-demand composition and sequence of transformation passes over these abstraction levels. This attribute allows the developers to coordinate optimization mechanisms in a manner that both general-purpose methods and hardware-specific tuning can be supported. As an example, to target the TPUs on Google, a compiler designer can use quantization passes, operator fusions, and XLA-specific optimizations before targeting the LLVM dialect, guaranteeing portability and optimum performance [13]. The fact that MLIR enables optimizations at all levels of the computational stack, i.e., not just on a level of ML abstractions but also on a level of low-level instructions, makes it both more flexible than monolithic compilers and more powerful in the development of scalable, maintainable, and high-performance ML systems.

Table 2 Modular Intermediate Representations for Multi-Level Optimization

IR Level	Examples of IR / Dialects	Optimization Focus	Key Benefits
High-Level IR	ONNX, TOSA, Graph IR, Domain-Specific DSLs	Operator fusion, layout/shape rewrites, quantization	Captures ML semantics; framework interoperability; enables domain-specific optimizations
Mid-Level IR	MLIR Dialects (Linalg, Tensor, Affine), Scheduling IR	Loop transformations, tiling, fusion, bufferization, vectorization	Exposes dataflow & parallelism; enables hardware-agnostic optimizations; bridges high-level ops to low-level code
Low-Level IR	LLVM IR, SPIR-V, PTX	Register allocation, instruction scheduling, target-specific lowering, inlining & DCE	Hardware-level optimizations; ensures efficient code generation for CPUs, GPUs, TPUs, NPUs, and accelerators
Cross-Level Feedback	Profiling Data, Cost Models, Auto-Tuning Signals	Guides transformation parameters across levels	Ensures adaptive optimizations based on runtime performance and device-specific behavior
Partitioning / Mapping	Device Placement (CPU, GPU, FPGA, TPU, NPU)	Kernel mapping, heterogeneous execution planning	Efficient utilization of heterogeneous hardware; portability across diverse compute backends

5. Integrating MLIR with ML Frameworks and Hardware Backends

The complete potential of MLIR is best demonstrated when it can act as an intermediary between high-level machine learning (ML) frameworks on the one hand, and on-hardware, machine-specific layers of execution, on the other. The modular and extensible nature of MLIR enables it to fit well with any variety of ML ecosystems, providing a coherent intermediate language to be collected and a bridge to facilitate optimization and cross-framework deployment across a variety of hardware. An outstanding example of this integration is with TensorFlow, whose MLIR support is very deeply integrated into the compiler stack with dialects like TensorFlow Graph and TFExecutor. These dialects express the dataflow characteristic of TensorFlow computation graphs and can be used to translate them to lower-level dialects such as TFLite and XLA HLO and then, in turn, compile to LLVM IR or accelerator-specific dialects. This nested transformation pipeline has the benefit of providing optimizations such as common subexpression elimination, buffer reuse, and shape inference in early stages, but remaining able to leverage backend-specific optimizations [14].

Similar efforts are being made to transfer such abilities to PyTorch through the Torch-MLIR project, aiming at having TorchScript programs represented in MLIR. Through this, PyTorch can access MLIR's modular pass infrastructure, which would allow cross-layer passes and backend support as that of TensorFlow MLIR integration, or even beyond [15]. Also, the MLIR ecosystem has recently been expanded so as to support the ONNX (Open Neural Network Exchange) model format, which is an intermediate model format to support portability. The ONNX-MLIR project allows lowering ONNX models to MLIR dialects, thereby providing the same optimization toolchain with backend targets spanning a wide variety of hardware platforms [16].

The ability to tailor the hardware targets where MLIR can execute is one of the defining strengths of MLIR. The definition of bespoke dialects and transformation passes enables hardware vendors to craft custom compilation pipelines for new accelerators with little engineering effort. Several start-ups like Cerebras and Graphcore have heavily invested in MLIR to allow them to more closely align and match up their novel chip designs to their tensor-heavy workload [17, 18]. In addition to this, the GPU dialect in MLIR allows the compilation to CUDA and ROCm and consequently ensures that execution on NVIDIA and AMD GPUs has no impedance. In much the same way, dialects over TPUs enable seamless coupling with Google-owned tensor processing ASICs, and MLIR's reconfigurability can give FPGA vendors the opportunity to be compatible with their dynamically reprogrammable execution environments.

Such close coupling with the layers it is built on (software-side and hardware it supports) firmly establishes MLIR as a universal compiler infrastructure, which can handle the complexities of the ML pipeline in a consistent and extensible framework, between model definition and hardware execution.

6. Optimisation Techniques with MLIR Dialects

MLIR also enables its users, such as developers, to optimise using a large and expressive toolbox of domain-specific optimizations that are tailored to the needs of the machine learning (ML) workloads. Among the most important optimizations it enables is the operator fusion, where sequences of operations on tensors, like a sequence of matrices multiplied by activations, are subsequently unified into a single computational kernel. This greatly reduces access to memory, reduces cache misses, and enhances throughput. This step can be automated and modularized with MLIR and its Linalg dialect, along with custom fusion passes to create fused loop nests with improvements to both latency and memory efficiency [19]. The critical optimisation is quantisation, consisting of decreasing the mathematical accuracy of matrix arithmetic, typically floating-point 32-bit (FP32), down to 8-bit integers (INT8), thus lowering the computing burden and memory demand. MLIR Quant dialect enables quantized operations and types to readily and selectively insert quantization-aware transformations at an early stage and at a fine-grain in the compilation pipeline [20].

Another big factor that influences ML model performance on a contemporary platform is efficient data layout. MLIR enables the rearranging of tensors into dependent layouts that match which part of the memory they access per instruction [21] through transformations involving the MemRef and Affine dialects. Another area that MLIR is very good at is parallelization, as is required when tapping into multi-core and many-core architectures like GPUs. The compiler infrastructure will be capable of automatically finding parallel loops with the help of a compiler dialect such as SCF (Structured Control Flow) and another dialect, GPU, to map the loops to thread hierarchies (blocks and warps in a CUDA or ROCm system). The assignment of optimization passes (loop tiling, vectorization, fusion, and thread mapping) can be made selectively based on whether the target hardware has shared or distributed memory [22].

In addition to these general-purpose transformations, MLIR has a mechanism for defining custom dialects that allows target-specific optimizations to be built. Specific examples can be seen in the case of the TPU backend in Google, where specialized dialects are used to allow optimizations such as strip-mining, software pipeline scheduling, and data prefetching, which have been fine-tuned to make use of the TPU interconnect topology and hierarchical memory system [23]. This is because this ability to define hardware-aware dialects guarantees that MLIR is able to support state-of-the-art accelerators, but leaves flexibility to innovate quickly. In sum, the dialect-based design of MLIR offers a coherent infrastructure in which advanced, performance-sensitive optimizations can be applied in the ML software stack, simplifying the process of implementing these optimizations, and minimizing the design and engineering effort that would otherwise be required to implement new high-performance compilers, to the simultaneous benefit of eliminating runtime performance bottlenecks in the ML software stack.

7. Future Directions in Compiler Infrastructure for AI Systems

The future of compiler infrastructure in Machine Learning (ML) is soon to change immensely in comparison to the current development in the software and hardware ecosystems. The extensible, modular architecture presented by MLIR provides a solid starting position; it is on a number of fronts that strategic development will yield the most potential. Unifying the existing patchwork of AI compiler landscapes is one of such possible directions. Although currently MLIR, XLA, TVM, and Glow have overlapping features, the fact that they are designed around a unified representation makes MLIR a candidate tool that could bring unity between frontend model representation and backend, hardware-specific compilation. Examples of such convergence can be seen in such projects as Torch-MLIR and ONNX-MLIR, with the latter potentially serving as an inter-framework common intermediate representation, standardizing optimization passes, and making life easier when sharing and interoperating across platforms [24][25].

Automation of compiler pass generation is the next important frontier to go alongside ecosystem unification. Because ML models are becoming more complex and more diverse, it is becoming less viable to design efficient compiler passes by hand. Combining meta-compilation strategies like reinforcement learning-based autotuning and empirical performance modeling with MLIR has the potential to produce compilers that can automatically optimize transformations on a per-workload basis and customized target hardware [26]. Besides, the hardware innovation is not becoming stationary. As new architecture paradigms such as neuromorphic processors, quantum accelerators, and optical chips begin to appear, MLIR needs to keep adding support to new hardware. The use of a dialect-based abstraction system that it presents is also very suited to this task, allowing both researchers and hardware vendors to prototype new IRs and reducing the execution patterns specific to their domain, and to their domain-specific execution models, with significantly reduced risk. Confirmed by early work on developing dialects to support analog computing and spiking neural networks, this flexibility makes the platform ideal to support their enhancement [27]. Moreover, as chiplet-based designs and heterogeneous systems-on-chip (SoCs) become more common, future compiler designs will also have to support distributed compute units and be tuned to such considerations as interconnect latency and energy efficiency, as well as shared memory limitations [28].

The second essential direction is linking the compiler infrastructures to the wider machine learning lifecycle tooling. ML compilation marks just the first step of the ML pipeline that involves training and evaluating the model, and deployment alongside real-time monitoring. Future versions of MLIR can also be fully integrated with MLflow, Kubeflow, and TensorBoard, so that compiler optimization becomes a first-class aspect of experiment tracking and deployment [29]. Lastly, community engagement and transparent governance will be fundamental to the long-term development and future of MLIR. With the onset of contributions and involvement of tech giants, such as Google, NVIDIA, and Intel, MLIR is expected to evolve into a transparent, RFC-based governance model that enables cooperation, innovation, and standardization both in the academic, industrial, and open-source realms [30].

8. Conclusion

With application complexity and performance requirements of machine learning applications increasing at an alarming rate, the current conventional compiler architecture is not sufficient in offering the flexibility and scalability, along with optimization capabilities required to service modern-day ML workloads. MLIR stands out as a revolutionary response to these problems, by providing a multi-level extensible compiler infrastructure at all abstraction levels- allowing integration with ML frameworks, effective optimization to all forms of heterogeneous hardware, and enabling new forms of innovation through custom dialects.

With the help of modular passes, domain-specific optimizations, and a wide range of compatibility in the ecosystem, MLIR helps to increase developer productivity, and a major improvement is the performance and maintainability of ML applications. Its use in compilation pipelines to TensorFlow, PyTorch, and ONNX demonstrates its applicability as a pattern that can standardise AI infrastructure. The future of the MLIR Roadmap is more ambitious still, consisting of machine learning in the compiler itself, a wider range of hardware, and connecting to entire ML pipelines.

Effectively, MLIR represents not only an advancement in compiler infrastructure but also a paradigm shifts in how software is constructed and optimized for AI systems ensuring the sustained relevance of compiler tools in the rapidly evolving landscape of machine learning.

References

[1] Muchnick S. Advanced compiler design implementation. San Francisco: Morgan Kaufmann; 1997.

[2] Cooper KD, Torczon L. Engineering a compiler. San Francisco: Morgan Kaufmann; 2022.

[3] Martínez PA, Peccerillo B, Bartolini S, García JM, Bernabé G. Applying Intel's oneAPI to a machine learning case study. Concurrency Comput Pract Exp. 2022;34(13):e6917.

[4] Hennessy JL, Patterson DA. Computer architecture: a quantitative approach. Amsterdam: Elsevier; 2011.

[5] Lattner C, Amini M, Bondhugula U, Cohen A, Davis A, Pienaar J, et al. MLIR: Scaling compiler infrastructure for domain specific computation. In: Proc IEEE/ACM Int Symp Code Gen Optim (CGO). IEEE; 2021. p. 2–14.

[6] Agostini NB, Curzel S, Amatya V, Tan C, Minutoli M, Castellana VG, et al. An MLIR-based compiler flow for system-level design and hardware acceleration. In: Proc IEEE/ACM Int Conf Comput-Aided Des (ICCAD). 2022. p. 1–9.

[7] Cytron R, Ferrante J, Rosen BK, Wegman MN, Zadeck FK. Efficiently computing static single assignment form and the control dependence graph. ACM Trans Program Lang Syst. 1991;13(4):451–90.

[8] Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, et al. PyTorch: An imperative style, high-performance deep learning library. In: Adv Neural Inf Process Syst (NeurIPS). 2019;32.

[9] Drescher F, Engelke A. Fast template-based code generation for MLIR. In: Proc ACM SIGPLAN Int Conf Compiler Constr. 2024. p. 1–12.

[10] Mithul C, Abdulla DM, Virinchi MH, Sathvik M, Belwal M. Exploring compiler optimization: a survey of ml, dl and rl techniques. In2024 8th International Conference on Computational System and Information Technology for Sustainable Solutions (CSITSS) 2024 Nov 7 (pp. 1-6). IEEE.

[11] Cao S, Yang M, Liu Y, Li Y, Zhao B, Chen X, et al. A heterogeneous CNN compilation framework for RISC-V CPU and NPU integration based on ONNX-MLIR. IEEE Trans Comput-Aided Des Integr Circuits Syst. 2025.

[12] Kumar A, Khedkar A, So H, Kuo M, Gurjar A, Biswas P, et al. DSP-MLIR: A domain-specific language and MLIR dialect for digital signal processing. In: Proc ACM SIGPLAN/SIGBED Int Conf Lang Comp Tools Embedded Syst. 2025. p. 146–57.

[13] Phothilimthana M, Abu-El-Haija S, Cao K, Fatemi B, Burrows M, Mendis C, et al. Tpugraphs: A performance prediction dataset on large tensor computational graphs. In: Adv Neural Inf Process Syst (NeurIPS). 2023;36:70355–75.

[14] Boehm M, Interlandi M, Jermaine C. Optimizing tensor computations: From applications to compilation and runtime techniques. In: Companion Proc Int Conf Manage Data (SIGMOD). 2023. p. 53–9.

[15] Bao G, Shi H, Cui C, Zhang Y, Yao J. UFront: Toward a unified MLIR frontend for deep learning. In: Proc IEEE/ACM Int Conf Autom Softw Eng (ASE). 2024. p. 255–67.

[16] Lange K, Fontana F, Rossi F, Varile M, Apruzzese G. Machine learning in space: Surveying the robustness of on-board ML models to radiation. In: Proc IEEE Space Comput Conf (SCC). 2024. p. 51–64.

[17] Panchumarthy R, Benala TR. An overview of AI workload optimization techniques. In: Boosting Software Development Using Machine Learning. 2025. p. 269–99.

[18] Pavlidakis M, Kitching C, Tomlinson N, Søndergaard M. Cross-vendor GPU programming: Extending CUDA beyond NVIDIA. In: Proc Workshop Heterog Composable Disaggregated Syst. 2025. p. 45–51.

[19] Di Z, Wang L, Ma Z, Shao E, Zhao J, Ren Z, et al. Accelerating parallel structures in DNNs via parallel fusion and operator co-optimization. ACM Trans Archit Code Optim.

[20] Zhang C, Cheng J, Yu Z, Zhao Y. MASE: An efficient representation for software-defined ML hardware system exploration. In: NeurIPS Workshop Mach Learn Syst (MLSys). 2023.

[21] Zhang H, Xing M, Wu Y, Zhao C. Compiler technologies in deep learning co-design: A survey. Intell Comput. 2023;2:0040.

[22] Bisbas G, Lydike A, Bauer E, Brown N, Fehr M, Mitchell L, et al. A shared compilation stack for distributed-memory parallelism in stencil DSLs. In: Proc ACM Int Conf Archit Support Program Lang Oper Syst (ASPLOS). 2024. p. 38–56.

[23] Martínez PA, Bernabé G, García JM. HDNN: A cross-platform MLIR dialect for deep neural networks. J Supercomput. 2022;78(11):13814–30.

[24] Chen T, Moreau T, Jiang Z, Zheng L, Yan E, Shen H, et al. TVM: An automated end-to-end optimizing compiler for deep learning. In: Proc USENIX Symp Oper Syst Design Implement (OSDI). 2018. p. 578–94.

[25] Pai S, Pingali K. A compiler for throughput optimization of graph algorithms on GPUs. In: Proc ACM SIGPLAN Int Conf Object-Oriented Program Syst Lang Appl (OOPSLA). 2016. p. 1–19.

[26] Ashouri AH, Killian W, Cavazos J, Palermo G, Silvano C. A survey on compiler autotuning using machine learning. ACM Comput Surv. 2018;51(5):1–42.

[27] Davies M, Wild A, Orchard G, Sandamirskaya Y, Guerra GAF, Joshi P, et al. Advancing neuromorphic computing with Loihi: A survey of results and outlook. Proc IEEE. 2021;109(5):911–34.

[28] Lee KJ, Lee J, Choi S, Yoo HJ. The development of silicon for AI: Different design approaches. IEEE Trans Circuits Syst I Regul Pap. 2020;67(12):4719–32.

[29] Zaharia M, Chen A, Davidson A, Ghodsi A, Hong SA, Konwinski A, et al. Accelerating the machine learning lifecycle with MLflow. IEEE Data Eng Bull. 2018;41(4):39–45.

[30] Suo C, Chen J, Liu S, Jiang J, Zhao Y, Wang J. Fuzzing MLIR compiler infrastructure via operation dependency analysis. In: Proc ACM SIGSOFT Int Symp Softw Test Anal (ISSTA). 2024. p. 1287–99.