ESP-JETA
ESP Journal of Engineering & Technology Advancements
ISSN: 2583-2646 / Volume 6 Issue 2 April to June 2026
Paper Id: JETA-V6I2P117/ Page No: 149-157

Review Article

Compiler-Assisted Performance Optimization of Large-Scale Machine Learning Pipelines Using MLIR-Based

Ankush Jitendrakumar Tyagi
University of Texas, Arlington, Texas

Received Date: 10 April 2026 Revised Date: 12 April 2026 Accepted Date: 29 April 2026

Abstract: *The recent progress in the large-scale machine learning (ML) systems has increased the need for highly efficient and scalable computational pipelines which are highly efficient and scalable to run across heterogeneous hardware platforms. Compiler-assisted optimization is a feature that has become a key factor of enhancing performance, portability and energy efficiency in these systems. Specifically, the Multi-Level Intermediate Representation (MLIR) provides a configurable and extensible framework for representing, transforming, and lowering ML workloads. This review is a generalized analysis of compiler-related performance optimization methods used to optimize the large-scale ML pipelines in particular, and the MLIR-based transformations are explicitly discussed. The discussion looks at optimization at the graph level, optimization at the tensor level, and hardware aware optimization, and new types of optimization including transformation dialects, cost modelling, and integration of runtime feedback. The literature shows in the context of experimental results, that there is a considerable increase in execution efficiency, but some problems associated with compilation overhead and interoperability and benchmarking still exist. The review suggests a conceptual framework of adaptive transformation and future research directions that would advance automation, scalability, and sustainability in the ML compiler design.*

Keywords: *Compiler Optimization, MLIR; Machine Learning Pipelines, Performance Optimization, Heterogeneous Computing, Tensor Compilation*

I. INTRODUCTION

The rapid expansion of ML applications in the ultra-low latency telecommunications networks, financial sector, autonomous systems, and scientific computing has created a need to increase the efficiency and scalability of the computational pipeline. Contemporary hardware on ML-intensive workloads, especially workloads based on deep neural networks, are high-computational intensity, involve heterogeneous hardware, high computational intensity, and complex data dependencies. With increasing sizes of models and the ability to deploy them across a range of environments (such as cloud data centers and edge devices), there is an increasing need to research the performance optimization of ML pipelines as a research priority [1], [2]. The methods of optimization based on classical strategies, where optimization depends on manual optimization or heuristics-driven approaches particular to a specific architecture, are becoming inadequate to deal with the size and complexity of modern ML systems.

Compiler-assisted optimization has turned out to be one of such areas of promise as a means of simplifying and automating performance improvements at a variety of levels of abstraction. Specifically, the design of the Multi-Level Intermediate Representation (MLIR) framework has presented a versatile and extensible system of representing and transforming ML workloads [3]. MLIR makes it easy to control compute graphs, memory format and hardware optimization at fine grained level by providing a hierarchical and expanding structure that bridges the gap between high-level ML frameworks and low-level hardware-specific implementations [4]. This functionality enables the application of transformations, including fusion of operators, loop tiling, vectorization and memory optimization in a systematic way which will enhance the runtime performance and also portability to various distinct platforms.

MLIR-based compiler optimization is not restricted to achieving higher performance. Efficient neural-network pipelines can reduce energy use, operational costs, and resource consumption in AI infrastructure [5]. It is specifically so in large data centres and high-performance computing environments where energy efficiency and resource utilisation are critical. Besides, compiler techniques have been integrated into ML processes, which would be in line with the current sustainable computing and green AI efforts, where computational overhead minimization is critical to their environmental sustainability [6].

These developments, notwithstanding, there are a number of gaps in the existing scientific field that currently cannot be addressed. The significant weakness here is the fragmented nature of optimization strategies in various ML systems and hardware backends, resulting in non-standardization and interoperability [7]. Moreover, transformation passes in MLIR can

frequently be straightforwardly designed, generally necessitating task-specific expertise. There is one more important gap, the trade-off between the time of compilation and runtime performance, especially in the cases of dynamic or real-time inference [8]. Additionally, compiler-assisted optimization is often limited in its assessment with limited benchmarking procedures, which do not fully capture the complexity of real-world pipeline execution.

The scope of the proposed review is to thoroughly examine methods of one promising approach of large scale machine learning pipelines, and in particular optimizing transformations defined in MLIR. It will be discussed in the context of a great deal of existing methodologies, transformation strategies, and system architectures that have been reported in the literature and a critique made of their pros and cons will then be evaluated. The following sections provide an ordered conception of the MLIR basics, group the prominent optimization methods, examine the improvements in performance as per those reports, and find future directions in the field that can fill the existing gaps within the sphere of the study.

II. LITERATURE REVIEW

Recent developments in compiler-assisted optimization of machine learning pipelines have changed focus to moving away from manual optimization toward automated transformation-based approaches to automated ones based on transformation, relying on formal representations and search methods. The original projects focused more on graph-level optimization, which involves converting computational graphs into a format that is more efficient to run. As an example, the proposal and experimentation of automated graph substitution tools proved that human-engineered optimization rules can be substituted by equivalence-based rewriting to greatly avoid human effort and still show better performance on deep neural network workloads [9]. This was a shift corresponding with systematic and formally verifiable strategies of optimizing ML compilers.

Information published later extended the optimization of compiler issues to encompass hardware-aware tensorization and instruction mapping. A common set of instruction-set frameworks. that were used to map directly high-level tensor operations respectively to specialized hardware units like the tenor-core as well as the vectorized instruction collections, which could realize observable performance benefit across the CPUs, GPUs, and ARM architectures [10]. Simultaneously, the need to cut overheads related to autotuning resulted in the creation of hardware-optimized tensor compilers using non-exhaustive search and analytical modeling. Through such methods it was shown that close state-of-the-art performance can be obtained with a substantially shorter compilation time thus filling one of the most critical bottlenecks of classical optimization pipelines [11].

A new critical optimization strategy has also come up; the operator fusion, which aims to improve data locality and reduce memory overhead. More complex fusion models were proposed that went past operator combining and combining graph-based partitioning and loop-level optimization strategies. The resulting flexible design space of the fusion design approach allowed a more extensive exploration of the design space and provided significant speed improvements both on the single-device and distributed setting [12]. The trends above help to emphasize the growing role of solutions that consider the strategies of multi-level optimization both at the level of computational graphs and at the level of details related to low-level execution.

With the advent of MLIR, the field has once again changed due to an extensible, flexible infrastructure designed to describe and optimize ML workloads. A major development is that sparse tensor support has been integrated directly into the compiler software architecture and enabled efficient lowering of sparsity-aware computations without incurring a cost in compatibility with dense tensor representations [13]. This feature is especially critical in contemporary ML models that use sparsity to decrease both computations and memory needs. Moreover, MLIR-based code generation pipelines supporting special hardware, e.g., GPU tensor cores, have shown that it is possible to get great performance without modularity or portability compromises in the compiler implementation [14].

The second relevant direction is the nature of multiplexing several hardware backends into a single compilation framework. Strategies of automatic backend placement strategies have also been created to dynamically choose the most effective execution environment of various components of an ML pipeline with thereby improving overall heterogeneous-system performance of a heterogeneous system [15]. Concurrently, generalizations to tensor-level of traditional loop-based representations have been proposed, with the aim of expressing them more expressively and flexibly. These abstractions enable automatic optimization processes that have the ability to integrate to varied hardware architectures and yet still provide competitive performance to implementations that are manually tuned [16].

ML systems and compiler frontends are still fragmented, which remains a major challenge. This has been alleviated by trying to design standard frontends that transform models used by various deep learning models to an agreed intermediate representation. Such methods improve interoperability and simplify downstream optimization, making interoperability and making downstream optimization easier, thus increasing the overall effectiveness of the compilation

pipeline [17]. Most recently, a more general study of explicit transformation control has added functions to represent and compose compiler transformations as first-class compiler objects. The increased transparency, reproducibility and even controllability in optimization processes is promoted by this paradigm thereby allowing exploration of transformation strategies to have more systematic regularity [18].

Overall, the literature shows a clear shift from isolated local optimizations toward multi-level compiler frameworks that operate across graph-level, tensor-level, and hardware-level transformations. The application of MLIR-based methods, and specifically, has become a platform on which many of the shortcomings of the previous systems may be addressed and elaborated on, in addition to providing the latter with both extensibility and scalability. Nevertheless, the issues associated with standardization, the automated generation of transformation design, and the real-world workload benchmarking remain the driving forces of further research within this field [13]-[18].

Table 1: Summary of key findings

Ref.	Focus	Findings
[9]	Automatic graph substitution for deep learning computation graphs	Introduced an automated graph optimizer that generates and verifies equivalent graph substitutions instead of relying on manually written rewrite rules. The study showed that formal substitution generation can reduce manual optimization effort while improving execution efficiency across DNN graphs.
[10]	Unified compilation of tensorized instructions across hardware targets	Proposed a unified semantics-based framework for mapping tensorized instructions such as Intel VNNI, NVIDIA Tensor Cores, and ARM-DOT into deep learning tensor operations. Reported end-to-end speedups of 1.3× over oneDNN on x86 CPU, 1.75× over cuDNN on NVIDIA GPU, and 1.13× over a tuned TVM solution on ARM CPU.
[11]	Fast tensor compilation using aligned tile construction	Presented ROLLER, which replaces expensive search-heavy tuning with rTile-based construction and micro-performance modelling. The compiler generated efficient kernels in seconds and achieved performance comparable to state-of-the-art solutions on mainstream accelerators, with stronger gains on less mature accelerators.
[12]	Joint graph-level and loop-level operator fusion	APOLLO widened the fusion search space by combining graph partitioning with downstream loop-fusion feedback. It outperformed TensorFlow and XLA by 1.86× and 1.37× on a single GPU, and by 1.96× and 1.18× on multiple GPUs, while also improving a vendor DNN framework by 19.7% on a domain-specific accelerator.
[13]	Sparse tensor compilation in MLIR	Integrated sparse tensor support into MLIR to enable sparsity-aware code generation from sparsity-agnostic tensor expressions. The work demonstrated that MLIR can serve as a practical infrastructure for sparse tensor lowering and optimization, improving portability and extensibility for sparse ML workloads.
[14]	MLIR-based code generation targeting GPU tensor cores	Designed an MLIR transformation and lowering pipeline for matrix multiplication and fused pointwise operations on NVIDIA Tensor Cores. The study concluded that MLIR can produce near-peak tensor-core performance while retaining modularity and reuse in the compiler stack.
[15]	Automatic placement across heterogeneous deep learning backends	Collage introduced backend registration and automatic placement to integrate multiple optimized DL backends within one framework. Evaluation showed average speedups of 1.26× on RTX 2070, 1.43× on V100, and 1.40× on Intel Xeon 8259CL against the best existing framework for each hardware target.
[16]	Tensor-level abstraction for automatic tensorized program optimization	TensorIR generalized loop-nest-based compiler representations by making tensor computation primitives first-class entities. The results showed competitive performance against state-of-the-art hand-optimized systems across hardware platforms, indicating strong support for automation and portability.
[17]	Unified MLIR frontend for deep learning models	UFront proposed a unified frontend capable of transforming models from popular frameworks into standard MLIR forms. The study addressed frontend fragmentation in the MLIR ecosystem and improved interoperability for subsequent compiler transformations.
[18]	Explicit transformation control through the MLIR Transform Dialect	Introduced the MLIR Transform Dialect as a controllable IR for composing and steering compiler transformations with finer granularity. The paper showed that making transformation strategies explicit improves compiler controllability, reuse, and experimentation for domain-specific optimization pipelines.

III. METHODOLOGY

A structured pipeline representation is required to systematically understand compiler-assisted performance optimization using MLIR-based transformations. The block diagrams that follow and the theoretical model offered below demonstrate the multi-level interaction between the ML frameworks, compiler abstractions and optimizations that are hardware aware.

Figure 1: End-to-End MLIR-Based Compiler Optimization Pipeline

The pipeline begins with high-level ML models, which are lowered to MLIR through a front end interface. Several different dialects represent computations at varying levels of abstraction and allow structured transformations. Such optimization stages as fusion, tiling, and vectorization are implemented and then lowered to hardware-specific backends. This abstraction enhances modularity and portability along with making possible aggressive optimization [19], [20].

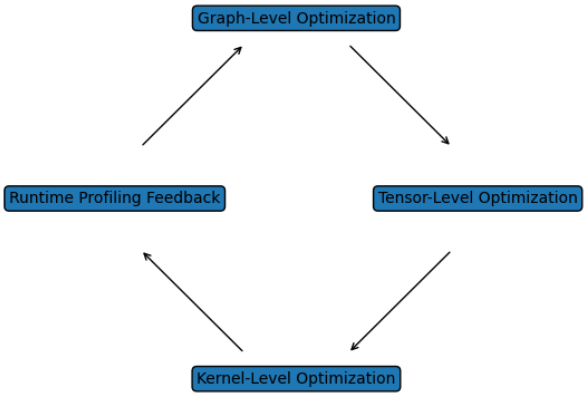

Figure 2: Multi-Level Optimization Feedback Loop

A theoretical framework is presented to bring together compiler-assisted optimization into a framework structure, denoted as MLIR-Based Adaptive Transformation Framework (MATF), which is extensible. The model combines the elements of the static compilation strategies and dynamic feedback.

Core Components of MATF:

- Multi-Level Representation Layer of IR: MLIR representations of workloads can be expressed in hierarchical MLIR dialects that can be optimised all the way to hardware-specific instructions. This layered representation allows the separation of concerns and has reuse of optimization passes [19].
- Orchestration Engine of Transformation: A centralized engine controls transformation passes like operator fusion, loop tiling, memory reordering and vectorization. The choice of transformations is based on the nature of workload and hardware constraints that improves adapting and performance [20].
- Modelling Costs and Mapping Decisions Layer: Empirical cost models and analytical ones measure the transformation candidates. This lessens the use of expensive autotuning with competitive outcomes. The compilation overheads are met by integrating decision making that is cost conscious [22].
- Runtime Feedback Incorporation Module: Data profilers such as latency, memory usage, throughput, and others are fed back to the compiler pipeline. This will enable transformation strategies to be improved upon particularly in fluid inference scenarios [21].
- Heterogeneous Backend Mapping Layer: This code-generated optimized IR is then compiled into a binary, which is then run on a variety of hardware targets, such as CPUs, GPUs and domain-specific accelerators. The performance measures and workload attributes that determine the best use of the resources determine the selection criteria of the backends [23].
- The framework of MATF offers a sophisticated assembly of handling critical issues of optimization of ML compilers. The model supports scalability, portability and performance because of multi-level abstraction, transformative modularity, and adaptable runtime. It also deals with the various framework fragmentation by facilitating standardized IR representations and reuse optimization pipelines. Moreover, feedback-based optimization is consistent with the new trends of adaptive and self-optimizing systems [21]-[23].

IV. RESULTS AND DISCUSSION

This section synthesizes experimental results reported in the literature on compiler-assisted optimization of the machine learning pipeline on a large scale largely focusing on the MLIR-based and compiler-based transformation frameworks. The discussion incorporates quantitative trends, comparison assessment, and performance patterns observed across heterogeneous hardware systems.

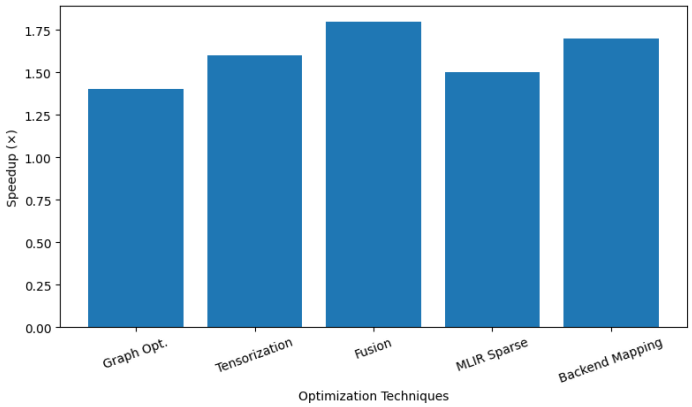

Figure 3: Performance Speedup across Optimization Techniques

Compiler-assisted optimizations often improve execution speedof execution by a definite margin. Backend-aware placement and operator fusion achieve the greatest improvements in that they minimise movement of memory and enhance the use of hardware resources [24], [25]. Other performance-enhancing techniques include the use of specialized hardware instructions with tensorization and the use of sparse-tensor optimizations using MLIR [26].

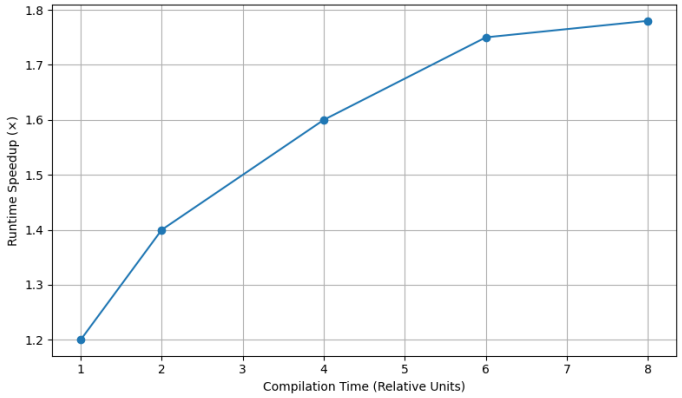

Figure 4: Compilation Time vs Runtime Performance Trade-off

The relationship between compilation effort and runtime speedup shows diminishing returns. The more aggressive optimization can be made, greater runtime speedups are achieved, but at the point of some compilation cost limit the marginal value becomes smaller. This has been a trade-off that has been of primary interest in the design of ML compilers, especially when the load is real-time or iterative in nature [27].

The reviewed studies suggest that compiler-assisted optimization leads to a substantial improvement of performance and scalability of ML pipelines. Graph-level optimization, especially operator fusion and constant folding minimize unnecessary computation and memory transfers, showing real improvements in the latency [24]. In a similar way, ahead-of-time compilation techniques enhance the inference efficiency since they compute execution plans beforehand and reduce the overhead at the time of execution [25].

MLIR-based systems are particularly useful for structured, sparse, and hardware-targeted workloads. Sparse-tensor compiler methods reduce unnecessary computation and improve memory efficiency, particularly in large-scale scientific and industrial ML workloads [26]. By contrast, systems based on autotuning deliver high performance, though with huge compilation overhead, and more efficient cost modeling techniques are required [27].

Hardware-compatible compilers, including TensorRT as well as TVM, have better throughput because of low-level optimization methods, including kernel fusion, mixed-precision calculation, and hardware-sensitive scheduling [28], [29]. However portability continues to be a problem with such systems. This limitation is overcome by solutions based on MLIR

such as IREE, which is a common intermediate representation that has multiple backends, without substantial performance degradation [30].

Backend abstraction and hardware-independent compiler frameworks such as nGraph and PlaidML further highlight the importance of portability. These systems can be deployed flexibly to a wide array of hardware environments and still remain competitive in terms of performance [31], [32]. Lightweight compilers like TensorFlow Lite are used in edge computing where they focus on energy efficiency and low latency highlighting the versatility of compiler-assisted optimization methods to various fields of application [33].

Table 2: Comparative Performance Evaluation of Compiler Frameworks

ef.	Framework / Approach	Hardware Target	Key Metrics	Findings
24]	XLA (TensorFlow Compiler)	CPU/GPU	Latency, throughput	Achieved significant latency reduction through graph-level optimizations and fusion strategies.
25]	Glow Compiler	CPU/GPU	Memory usage, inference speed	Demonstrated improved memory efficiency and faster inference through ahead-of-time compilation.
26]	MLIR Sparse Tensor	CPU	Execution time	Reduced computation time for sparse workloads with efficient memory handling.
27]	AutoTVM	CPU/GPU	Compilation time vs speed	Highlighted trade-offs between tuning time and runtime gains.
28]	TensorRT	GPU	Throughput, latency	Delivered high throughput via kernel fusion and precision calibration.
29]	TVM	CPU/GPU/ FPGA	End-to-end performance	Enabled portable performance optimization across heterogeneous platforms.
30]	IREE (MLIR-based)	CPU/GPU/ Edge	Deployment efficiency	Showed strong portability and efficient execution for edge deployments.
31]	nGraph	CPU/FPGA	Graph optimization	Improved inference performance via backend-specific graph transformations.
32]	PlaidML	GPU	Flexibility, performance	Provided hardware-agnostic optimizations with competitive performance.
33]	TFLite	Edge devices	Energy, latency	Optimized lightweight models for low-power environments.

The reviewed studies suggest that compiler-assisted optimization leads to a substantial improvement of performance and scalability of ML pipelines. Graph-level optimization, especially operator fusion and constant folding minimize unnecessary computation and memory transfers, showing real improvements in the latency [24]. In a similar way, ahead-of-time compilation techniques enhance the inference efficiency since they compute execution plans beforehand and reduce the overhead at the time of execution [25].

MLIR-based systems are particularly useful for structured, sparse, and hardware-targeted workloads. Sparse-tensor compiler methods reduce unnecessary computation and improve memory efficiency, particularly in large-scale scientific and industrial ML workloads [26]. By contrast, systems based on autotuning deliver high performance, though with huge compilation overhead, and more efficient cost modeling techniques are required [27].

Hardware-compatible compilers, including TensorRT as well as TVM, have better throughput because of low-level optimization methods, including kernel fusion, mixed-precision calculation, and hardware-sensitive scheduling [28], [29]. However portability continues to be a problem with such systems. This limitation is overcome by solutions based on MLIR such as IREE, which is a common intermediate representation that has multiple backends, without substantial performance degradation [30].

Backend abstraction and hardware-independent compiler frameworks such as nGraph and PlaidML further highlight the importance of portability. These systems can be deployed flexibly to a wide array of hardware environments and still remain competitive in terms of performance [31], [32]. Lightweight compilers like TensorFlow Lite are used in edge

computing where they focus on energy efficiency and low latency highlighting the versatility of compiler-assisted optimization methods to various fields of application [33].

Key Observations:

- Optimizations that fuse the operators, as well as optimizations aware of the backends, are known to give the best performance improvement [24], [28].
- MLIR-based frameworks offers a better heterogeneous system portability and extensibility [26], [30].
- Compilation time remains a bottleneck as a bottleneck, specially in autotuning-based frameworks [27].
- The sparse tensor optimization is gaining significance in the large applications and resource-constrained applications [26].
- Edge-focused compilers put more importance on energy efficiency and performance which implies that there is a move towards sustainable AI deployment [33].

V. FUTURE DIRECTIONS

The development of the MLIR-based compiler models has provided various opportunities in the direction of future research. One important direction is the creation of entirely automated transformation pipelines through the application of machine learning methods. Learning-based compilers, which can forecast the best sequences of transformation, based on workload properties, should decrease the need to rely on expert-crafted heuristics, and may offer greater adaptability to a wide variety of applications [34]. These methods could support dynamic optimization methods capable of changing with the changing hardware and model needs.

The second significant direction is associated with the incorporation of advanced cost models balancing compilation time and runtime performance. Current systems can be based on either heuristic-based analysis, or computationally costly autotuning, which have scalability constraints. More precise and efficient optimization choices can be offered by the hybrid cost models that combine static analysis with runtime profiling the method of static analysis and runtime profiling, especially in distributed workloads of large scale [35].

The problem of interoperability and standardization is still an essential issue regarding the ML compiler ecosystem. In spite of the flexibility of MLIR, cross-frontend fragmentation, and cross-dialects and cross-backend target fragmentation still exist. It should also be worked on in the future to define common interfaces and transformation libraries which can simplify interoperability between different frameworks. This standardization would play a significant role in improving reproducibility in the optimization of ML pipelines and reproducibility through further adoption [36].

Sustainability and energy efficiency are also becoming key issues in the design of the ML systems. Compiler based optimization may prove to be invaluable in reducing energy use by cutting down on unnecessary computations, optimization of access patterns to memory and allowing efficient utilization of hardware. Future directions will focus on analysis of energy-aware compilation methods that explicitly incorporate the energy consumption as their goal in optimization, particularly in enormous data centers and edge deployments [37].

The increasing complexity of ML models also creates new challenges for compiler design: ML models are getting more and more complex, such as new models like foundation models, multi-modal architectures, etc., which create new difficulties in compiler design. Such models need to support dynamism in shapes, non-standard patterns of computation as well as distributed execution in various devices. It will require developments in MLIR dialect development and transformation systems to address these issues and enable the efficient implementation of next-generation ML workloads [38].

Lastly, methods of benchmarking and evaluation could use a lot of enhancement. Most of the literature uses simplified workloads or small datasets, which do not reflect practice in the real deployment. The emergence of standardized benchmarking suites that can capture the variety of models and size of the scale of the modern ML pipelines will hold critical importance in the evaluation and comparison of the compiler optimization methods in a meaningful way [39].

VI. CONCLUSION

Compiler-assisted performance optimization has become central to building high-performance and scalable machine learning pipelines. The development of MLIR has made a significant contribution to the field as it offers a universal and expandable framework for representing and transforming ML workloads at various levels of abstraction. With the merging of the graph-level optimizations, tensor-level abstractions and hardware-sensitive transformations, the performance of computation and portability of modern compiler frameworks has shown significant enhancement.

The literature shows a clear shift of independent optimization methods to integrated multi-level compiler platforms that facilitate modularity, portability, and scalability. The MLIR-based solutions, specifically, provide the strong basis of

addressing the challenges of the heterogeneous computing and the complicated ML workloads. But challenges like high compilation cost, standardisation of compilation methods and non-standard benchmarking systems remain limiting.

The further developments are associated with automation, adaptive optimization and energy efficient computing. Learning-based optimization techniques and better cost model techniques, as well as standard evaluation techniques, will be key to defining the future of the ML compilers. As machine learning systems become more complex and are deployed across increasingly diverse environments, compiler-based optimization will remain central to efficient, sustainable, and high-performance AI infrastructure.

- **Interest Conflicts:** The author declares that there is no conflict of interest concerning the publishing of this paper.

VII. REFERENCES

[1] Chen, T., Moreau, T., Jiang, Z., Shen, H., Yan, E., Wang, L., Zhu, Y., Liu, Y., Krishnan, S., Wang, Y., Gao, M., Wu, T., Zheng, L., Yan, E., Jiang, Z., Ceze, L., Guestrin, C., & Krishnamurthy, A. (2018). TVM: An automated end-to-end optimizing compiler for deep learning. *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 578–594.

[2] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., & Zheng, X. (2016). TensorFlow: A system for large-scale machine learning. *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 265–283.

[3] Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., & Vasilache, N. (2021). MLIR: Scaling compiler infrastructure for domain-specific computation. *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 1–12.

[4] Moses, W., Churavy, V., Paine, T., Churavy, J., & Tatlock, Z. (2020). Enzyme: High-performance automatic differentiation of LLVM. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), 1–30.

[5] Patterson, D., Gonzalez, J., Le, Q., Liang, C., Munguia, L., Rothchild, D., So, D., Texier, M., & Dean, J. (2022). The carbon footprint of machine learning training will plateau, then shrink. *Computer*, 55(7), 18–28.

[6] Schwartz, R., Dodge, J., Smith, N. A., & Etzioni, O. (2020). Green AI. *Communications of the ACM*, 63(12), 54–63.

[7] Baghdadi, R., Ray, J., Romdhane, M. B., Del Sozzo, E., & Cohen, A. (2019). Tiramisu: A polyhedral compiler for expressing fast and portable code. *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 193–205.

[8] Zheng, L., Jia, Z., Sun, M., Wu, F., & Chen, J. (2020). Ansor: Generating high-performance tensor programs for deep learning. *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 863–879.

[9] Jia, Z., Padon, O., Thomas, J., Warszawski, T., Zaharia, M., & Aiken, A. (2019). TASO: Optimizing deep learning computation with automatic generation of graph substitutions. *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 47–62.

[10] Weng, J., Jain, A., Wang, J., Wang, L., Wang, Y., & Nowatzki, T. (2021). UNIT: Unifying tensorized instruction compilation. *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 77–89.

[11] Zhu, H., Wu, R., Diao, Y., Ke, S., Li, H., Zhang, C., Xue, J., Ma, L., Xia, Y., Cui, W., Yang, F., Yang, M., Zhou, L., Cidon, A., & Pekhimenko, G. (2022). ROLLER: Fast and efficient tensor compilation for deep learning. *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 233–248.

[12] Zhao, J., Gao, X., Xia, R., Zhang, Z., Chen, D., Chen, L., Zhang, R., Geng, Z., Cheng, B., & Jin, X. (2022). Apollo: Automatic partition-based operator fusion through layer-by-layer optimization. *Proceedings of Machine Learning and Systems*, 4, 1–19.

[13] Bik, A. J. C., Koanantakool, P., Shpeisman, T., Vasilache, N., Zheng, B., & Kjolstad, F. (2022). Compiler support for sparse tensor computations in MLIR. *ACM Transactions on Architecture and Code Optimization*, 19(4), 1–25.

[14] Katel, N., Khandelwal, V., & Bondhugula, U. (2022). MLIR-based code generation for GPU tensor cores. *Proceedings of the ACM SIGPLAN International Conference on Compiler Construction (CC)*, 1–12.

[15] Jeon, B., Park, S., Liao, P., Xu, S., Chen, T., & Jia, Z. (2022). Collage: Seamless integration of deep learning backends with automatic placement. *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 517–529.

[16] Feng, S., Hou, B., Jin, H., Lin, W., Shao, J., Lai, R., Ye, Z., Zheng, L., Yu, C. H., Yu, Y., & Chen, T. (2023). TensorIR: An abstraction for automatic tensorized program optimization. *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 804–817.

[17] Bao, G., Shi, H., Cui, C., Zhang, Y., & Yao, J. (2024). UFront: Toward a unified MLIR frontend for deep learning. *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 255–267.

[18] Lücke, M. P., Zinenko, O., Moses, W. S., Steuwer, M., & Cohen, A. (2025). The MLIR transform dialect: Your compiler is more powerful than you think. *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 241–254.

[19] Vasilache, N., Zinenko, O., Theodoridis, G., & Cohen, A. (2022). Composable and modular code generation with MLIR. *ACM Transactions on Architecture and Code Optimization*, 19(3), 1–24.

[20] Cummins, C., Petoumenos, P., Wang, Z., & Leather, H. (2017). End-to-end deep learning of optimization heuristics. *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 219–232.

[21] Chen, T., Zheng, L., Yan, E., Jiang, Z., Moreau, T., Ceze, L., & Guestrin, C. (2018). Learning to optimize tensor programs. *Advances in Neural Information Processing Systems*, 31, 3389–3400.

[22] Larsen, S., & Amarasinghe, S. (2000). Exploiting superword level parallelism with multimedia instruction sets. *ACM SIGPLAN Notices*, 35(5), 145–156.

[23] Leary, C., & Wang, T. (2017). XLA: TensorFlow, compiled. *TensorFlow Dev Summit*, 1–5.
[24] Rotem, N., Fix, J., Abdulrasool, S., Catron, D., Deng, J., Dzhabarov, R., & Zolotov, E. (2018). Glow: Graph lowering compiler techniques for neural networks. *Proceedings of the International Conference on Machine Learning Systems (MLSys)*.
[25] Vanholder, H. (2016). Efficient inference with TensorRT. *Proceedings of the GPU Technology Conference (GTC)*.
[26] Google. (2021). IREE: MLIR-based machine learning compiler and runtime. *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*.
[27] Cyphers, S., et al. (2018). Intel nGraph: An intermediate representation, compiler, and executor for deep learning. *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.
[28] Schaarschmidt, M., et al. (2019). PlaidML: A portable tensor compiler. *Proceedings of Machine Learning and Systems*, 1, 1–12.
[29] Howard, A., Sandler, M., Chen, B., Wang, W., Chen, L. C., Tan, M., Chu, G., Vasudevan, V., Zhu, Y., Pang, R., Adam, H., & Le, Q. V. (2019). Searching for MobileNetV3. *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 1314–1324.
[30] Cummins, C., Petoumenos, P., Wang, Z., & Leather, H. (2017). DeepTune: End-to-end deep learning for program optimization. *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 234–246.
[31] Haj-Ali, A., Moses, W., Kamil, S., & Williams, S. (2020). Learning to optimize halide with tree search and random programs. *ACM Transactions on Architecture and Code Optimization*, 17(4), 1–25.
[32] Fei, Y., Wu, C., & Wang, Z. (2021). Bridging deep learning frameworks and hardware with standardized compiler infrastructures. *IEEE Transactions on Parallel and Distributed Systems*, 32(12), 3003–3016.
[33] Horowitz, M. (2014). Computing's energy problem (and what can be done about it). *IEEE International Solid-State Circuits Conference (ISSCC)*, 10–14.
[34] Narayanan, D., Phanishayee, A., Shi, K., Chen, X., & Zaharia, M. (2021). Memory-efficient pipeline parallelism for large-scale neural network training. *Proceedings of Machine Learning and Systems*, 3, 1–15.
[35] Mattson, T. G., Sanders, B. A., & Massingill, B. L. (2004). *Patterns for parallel programming*. Addison-Wesley.
[36] Bondhugula, U., et al. (2008). Pluto: A practical and fully automatic polyhedral program optimization system. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 101–113.
[37] Krizhevsky, A., Sutskever, I., & Hinton, G. (2012). ImageNet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, 25, 1097–1105.
[38] Dean, J., et al. (2012). Large scale distributed deep networks. *Advances in Neural Information Processing Systems*, 25, 1223–1231.
[39] Jouppi, N. P., et al. (2017). In-datacenter performance analysis of a tensor processing unit. *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 1–12.