Proceedings of CONF-CDS 2026 Symposium: Data-Centric AI Security: Securing Models, Learning Agents, and Autonomous Systems
DOI: 10.54254/2755-2721/2026.CH35364

AI Compilation and Hardware-Software Co-Design for Resource-Constrained Edge Devices: A Brief Review

Chenhe Zhu

Maynooth International Engineering College, Fuzhou University, Fuzhou, China
17705013350@163.com

Abstract. Edge artificial intelligence (Edge AI) has emerged as a promising solution to provide real-time and privacy-aware intelligent services at proximity to data sources. However, it is difficult to implement deep learning on edge devices with limited resources, such as microcontrollers, embedded devices, and IoT terminals with low power consumption, because they have limited computing power, memory capacity, and energy supply. This paper introduces a concise overview on AI compilation and hardware co-design methodologies for efficient inference execution on resource-constrained edge devices. This paper provides an overview of the key challenges in deploying deep learning models on resource-constrained devices and presents illustrative solutions in terms of compression techniques and hardware implementation. The discussion includes graph optimization, operator fusion, quantization, memory-aware inference, hardware-specific code generation, accelerator-based inference, latency prediction, and runtime optimization. The reviewed work suggests that high-performance efficient edge AI requires cross-layer rather than isolated model-level compression. AI compilation enables more efficient execution by converting neural networks to optimized hardware-aware code, and hardware-co-design further improves latency, memory, and energy efficiency. These results imply that next-generation edge AI systems should focus on portable compiler toolchains, memory-aware optimization, energy-aware design, and standardized benchmarking methodologies.

Keywords: Edge AI, Resource-Constrained Devices, AI Compilation, Hardware Co-Design, TinyML

1. Introduction

Edge computing with artificial intelligence has become a common way to deploy intelligent services close to data sources. The growth of Internet of Things devices, smart sensors, embedded systems, and wearable terminals has generated massive data at the network edge. Although cloud computing provides powerful processing capability, it may suffer from high latency, bandwidth bottlenecks, privacy risks, and transmission costs [1]. In contrast, Edge AI performs computation near users or devices, enabling lower latency, reduced bandwidth consumption, improved privacy protection, and timely local decision-making [2].

However, deploying deep learning models on resource-constrained edge devices remains difficult. Microcontrollers and low-power embedded systems usually have limited computation

© 2026 The Authors. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (https://creativecommons.org/licenses/by/4.0/).

DOI: 10.54254/2755-2721/2026.CH35364

capacity, memory size, energy budgets, and their hardware platforms are often fragmented. TinyML brings machine learning to extremely low-profile devices, but these platforms still face severe storage, memory, and processing limitations [3]. Lightweight deep learning research further shows that reducing parameters or floating-point operations does not always guarantee faster inference, because execution efficiency is also affected by memory access, data movement, operator implementation, and hardware characteristics [4].

Existing studies have discussed Edge AI, TinyML, lightweight deep learning, and deep learning compilers from different perspectives. However, efficient deployment on resource-constrained edge devices requires cross-layer optimization involving model compression, memory-aware inference, AI compilation, runtime support, and hardware execution. Deep learning compilers transform high-level models into optimized code through intermediate representation, graph optimization, operator fusion, and hardware-specific code generation [5], while TensorFlow Lite Micro shows the importance of portable runtime support for fragmented embedded platforms [6]. Therefore, this paper reviews AI compilation and hardware-software co-design techniques for efficient inference on resource-constrained edge devices.

2. Background and deployment challenges

2.1. Edge AI and TinyML

Edge computing shifts data processing and computation away from centralized cloud data centers to terminal-side network edges, alleviating the risk of transmission delay, bandwidth strain, and privacy leak [1]. Edge AI also implements AI models nearer users or data sources for local inference and decision making on sensors, smart cameras, wearable devices, and industrial endpoints [2].

TinyML drives edge intelligence to the world of microcontrollers and other ultra-low power embedded systems. It stresses low latency, low power, and low bandwidth features and is applicable to ever-on sensing operations like keyword spotting, activity recognition, environmental monitoring, and predictive maintenance [3]. However, these advantages hinge on models being able to fit into devices with extremely constrained computation, memory, and power resources.

2.2. Resource constraints in edge devices

Limitations of computing, memory, energy consumption, and hardware heterogeneity are the primary challenges in humanized cognitive edge devices. Microcontrollers and embedded processors are orders of magnitude less powerful than cloud servers or desktop GPUs, so it is difficult to run sophisticated neural networks on them directly.Memory constraints is also a big challenge since model parameters and intermediate activation tensors need to be stored in limited Flash and SRAM memory. TensorFlow Lite Micro emphasizes that embedded devices may have as little as a couple of megabytes, or a few hundred kilobytes of memory, or even less, so memory planning, tensor buffer reuse, and runtime memory management become important [6]. Moreover, many edge devices run on batteries, and local inference is required to meet the tight power budgets [3]. Hardware heterogeneity complicates deployment even more as portability is limited by varying processors, accelerators, memory hierarchies, and toolchains, and these factors add to the engineering expense [6].

In additional, the reduction of parameter counts or floating-point operations does not always infer the faster inference because the execution efficiency is also constrained by memory access,

DOI: 10.54254/2755-2721/2026.CH35364

data movement, operator implementation, and hardware features [4]. Thus, it is necessary to perform cross-layer optimization for effective Edge AI deployment that integrates quantization, memory-aware inference, AI compiler, runtime scheduler, memory planner, and hardware-aware runtime. Figure 1 exemplifies this framework.

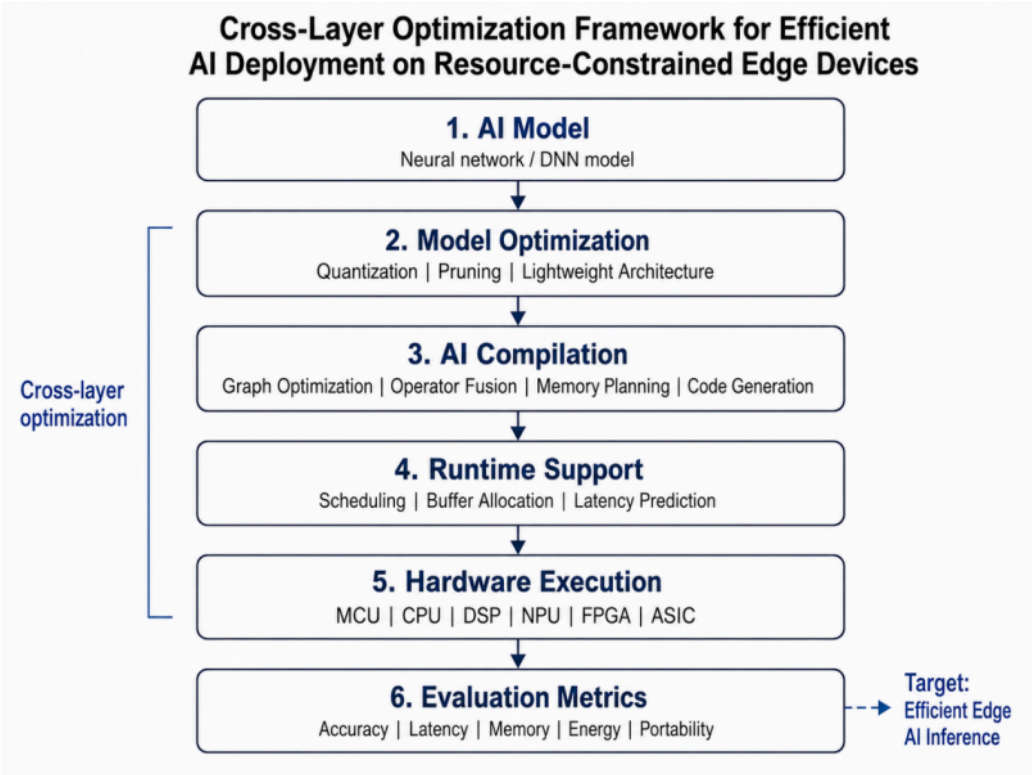

Figure 1. Cross-layer optimization framework for efficient AI deployment on resource-constrained edge devices

3. AI compilation techniques

As shown in Figure 1, AI compilation connects model-level optimization and hardware execution by transforming high-level neural network models into efficient code for target platforms. A deep learning compiler usually includes frontend processing, intermediate representation, optimization passes, and backend code generation. Typical optimization techniques include graph optimization, operator fusion, hardware-specific optimization, auto-tuning, and optimized kernel generation [5].

3.1. Graph optimization and tensor program optimization

Graph optimization is often performed at an early stage of AI compilation. In most deployment frameworks, a neural network is represented as a computational graph, where operators are modeled as nodes and tensors are treated as edges between them. Based on this graph structure, the compiler can apply a series of transformation passes, such as removing redundant operators, folding constants, eliminating unnecessary data copies, and merging suitable neighboring operations. Among these techniques, operator fusion is particularly valuable for edge inference, since it can reduce the generation of intermediate tensors, lower memory access frequency, and improve execution latency. In addition to graph-level optimization, tensor-level intermediate representation also plays an important role in hardware-aware compilation. TensorIR provides a structured

DOI: 10.54254/2755-2721/2026.CH35364

abstraction for describing tensor computation, multi-dimensional memory access patterns, and hardware-related constraints [7].

Optimization at the operator level is equally important. ROLLER points out that traditional tensor compilers often rely on costly search procedures to determine loop transformations and scheduling strategies. To reduce this overhead, it introduces the rTile abstraction, which is designed according to hardware characteristics and can generate efficient kernels within seconds [8]. This capability is especially meaningful for edge AI scenarios, because edge devices are highly diverse and require compilation methods that are both efficient and repeatable.

3.2. Memory planning and hardware-specific code generation

Memory planning is a critical step in edge AI compilation, as embedded devices usually provide only limited SRAM and Flash resources. After graph transformation and operator scheduling, the compiler still needs to decide how tensors are allocated, reused, and released during execution. Well-designed buffer reuse strategies can reduce peak memory consumption and make it possible to deploy neural networks on devices with strict memory constraints.

Ahead-of-time compilation is another important technique for edge deployment. For bare-metal Internet of Things devices, dynamically interpreting a model graph may introduce unnecessary runtime overhead and increase system complexity. MicroTVM addresses this problem by translating pre-trained models into C source libraries for bare-metal deployment, thereby reducing runtime dependencies and supporting platforms such as ARM Cortex-M devices [9]. It can also offload compute-intensive operators to dedicated accelerators when such hardware is available. However, hardware-specific code generation becomes more difficult when edge platforms are heterogeneous. Many MCU-class systems now integrate general-purpose processors together with specialized DNN accelerators. MATCH tackles this issue through a model-aware TVM-based framework, which maps DNN operators to appropriate processors or accelerators while maintaining retargetability across different heterogeneous MCU platforms [10].

More generally, compiler technology has become an essential part of deep learning hardware–software co-design. It links model-level optimization, workload mapping, memory scheduling, hardware interfacing, and multi-backend code generation into a unified deployment workflow [11].

Table 1. Representative AI compilation techniques for edge devices

Technique	Main Purpose	Representative Work	Benefit for Edge AI
Graph optimization	Simplify computational graphs and remove redundant operations	Deep Learning Compiler Survey [5]	Reduces execution overhead
Operator fusion	Combine adjacent operators into a single execution unit	Deep Learning Compiler Survey [5]	Reduces memory access and latency
Tensor-level IR	Represent tensor computation and hardware tensor primitives	TensorIR [7]	Enables hardware-aware optimization
Tensor compilation	Generate efficient operator kernels	ROLLER [8]	Reduces compilation time and improves kernel efficiency
AOT compilation	Generate executable code before deployment	MicroTVM [9]	Supports bare-metal edge deployment
Hardware-aware mapping	Map operators to CPUs or accelerators	MATCH [10]	Improves performance on heterogeneous edge devices

DOI: 10.54254/2755-2721/2026.CH35364

Table 1. (continued)

Compiler-assisted co-design	Connect model, compiler, runtime, and hardware layers	Compiler Technologies in Deep Learning Co-Design [11]	Supports cross-layer optimization

Table 1 shows that AI compilation techniques improve edge inference through graph-level simplification, tensor-level optimization, ahead-of-time code generation, and hardware-aware mapping.

4. Hardware co-design and optimization

AI compilation improves neural network running performance through graph, tensor, memory, and code-generation optimization. However, software-level compilation alone is not enough for resource-constrained edge devices. Edge inference also depends on compute units, memory hierarchy, dataflow, interconnection, accelerator architecture, and power budget. Therefore, hardware co-design treats edge inference as a system-level optimization problem involving model design, model optimization, algorithm–hardware co-design, and accelerator design [12].

4.1. Hardware-aware model and accelerator design

Hardware-aware optimization should consider the interaction among model architecture, accelerator architecture, and compiler mapping, rather than treating them as independent stages. NAAS is a typical example of this idea, as it searches neural network structures, accelerator architectures, and compiler mappings within a unified optimization loop [13]. Through this joint search, the neural workload can be better matched with accelerator resources, which helps improve compute-array utilization, memory efficiency, and energy-delay performance [13].

Hardware-aware model design aims to generate neural networks that can maintain good accuracy while running efficiently on the target hardware. Different from conventional NAS methods, hardware-aware NAS introduces practical deployment metrics, including latency, energy consumption, memory footprint, and hardware utilization, into the search objective [14]. This is especially important for edge devices, where the most suitable model is not necessarily the most accurate one, but the one that achieves a reasonable balance among accuracy, latency, memory usage, and energy efficiency.

This form of co-design is highly relevant to resource-constrained edge scenarios because it reduces the gap between model requirements and hardware capabilities. It also connects model-level optimization with low-level hardware execution, which is consistent with the cross-layer framework illustrated in Figure 1.

4.2. Dataflow and accelerator-level optimization

Dataflow optimization is a key problem in hardware co-design, because tensor computation depends not only on arithmetic operations but also on how data is moved and reused. For spatial architectures, hardware dataflow determines how computation is mapped to processing elements, in what order operations are executed, how tensors are reused, and how data is transferred across the memory hierarchy. TENET shows that accurate dataflow modeling is necessary for estimating reuse, bandwidth demand, latency, and energy consumption on tensor accelerators [15].

DOI: 10.54254/2755-2721/2026.CH35364

Memory optimization is closely connected with dataflow design, since edge devices usually have limited on-chip memory and costly off-chip memory access. A well-designed dataflow can increase data reuse, reduce memory bandwidth pressure, and improve accelerator utilization. Therefore, tiling, loop ordering, buffer allocation, and data movement scheduling are all important factors for efficient edge inference.

FPGA-based acceleration provides a concrete example of hardware/software co-design in edge AI deployment. SECDA combines SystemC simulation with hardware execution to reduce the exploration cost caused by repeated FPGA synthesis [16]. It also demonstrates that accelerator performance is not determined only by the hardware architecture itself, but also by CPU-side drivers, data transfer mechanisms, tiling strategies, and overall system integration [16].

4.3. Model compression, memory, and runtime optimization

Model compression and runtime optimization are closely associated with hardware co-design, because they directly influence memory footprint, arithmetic cost, and inference latency. Quantization lowers numerical precision, which can reduce model size, memory traffic, and computational complexity on low-power edge devices [17].

Memory-aware inference is also essential for microcontroller-class platforms. MCUNetV2 shows that peak activation memory can become a major limitation when visual deep learning models are deployed on tiny devices. Its patch-based inference strategy reduces memory pressure by processing image regions in a more memory-efficient manner [18]. This suggests that optimizing activation memory can be as important as reducing the number of model parameters.

Runtime-level optimization further improves deployment efficiency across different edge platforms. Latency prediction methods such as nn-Meter can estimate inference latency on various hardware devices, allowing developers to compare models and deployment configurations before conducting costly on-device tests [19]. These methods show that efficient edge AI deployment requires coordination across the model, compiler, runtime, and hardware layers.

Table 2. Comparison of optimization layers for resource-constrained edge AI

Optimization Layer	Representative Techniques	Main Objective	Key Constraint	Related References
Model level	Quantization, pruning, lightweight architecture, hardware-aware NAS	Reduce model size and computation	Accuracy degradation	[4, 12, 14, 17]
Compiler level	Graph optimization, operator fusion, AOT compilation, hardware-aware mapping	Improve execution efficiency and portability	Backend dependence	[5, 7-11]
Memory level	Patch-based inference, tensor buffer reuse, memory planning	Reduce peak memory usage	Model structure and tensor lifetime	[6, 9, 18]
Runtime level	Latency prediction, scheduling, deployment configuration selection	Reduce latency and improve deployment selection	Runtime overhead and platform diversity	[10, 19]
Hardware level	Neural accelerators, FPGA acceleration, dataflow optimization, PE mapping	Improve energy efficiency and throughput	Design complexity and hardware cost	[12, 13, 15, 16]
Co-design level	Joint model–compiler–hardware search, DSE, hardware/software co-design	Balance accuracy, latency, memory, and energy	Large design space	[11, 13, 16]

DOI: 10.54254/2755-2721/2026.CH35364

Table 2 shows that efficient edge inference depends on coordination across model, compiler, memory, runtime, hardware, and co-design layers.

5. Conclusion

Efficient AI deployment on resource-constrained edge devices is a cross-layer optimization problem involving models, compilers, runtime systems, memory management, and hardware platforms. This paper reviewed AI compilation and hardware co-design techniques for improving neural network inference under limited computation, memory, and energy budgets. AI compilation improves execution efficiency through graph optimization, tensor-level representation, memory planning, ahead-of-time compilation, and hardware-aware code generation, while hardware co-design enhances latency, memory efficiency, and energy consumption through accelerator-aware optimization.

In addition, quantization, memory-aware inference, and latency prediction work with compiler and hardware co-design by reducing model size, peak memory usage, and deployment uncertainty. Future edge AI systems should emphasize reusable compiler abstractions, automated design space exploration, accurate latency and energy modeling, and portable toolchains. Such cross-layer approaches can support more efficient and energy-aware AI inference on resource-constrained edge devices.

References

[1] Hua, H., Li, Y., Wang, T., Dong, N., Li, W., & Cao, J. (2023). Edge computing with artificial intelligence: A machine learning perspective. *ACM Computing Surveys, 55*(9), Article 184.
[2] Singh, R., & Gill, S. S. (2023). Edge AI: A survey. *Internet of Things and Cyber-Physical Systems, 3*, 71–92.
[3] Abadade, Y., Temouden, A., Bamoumen, H., Benamar, N., Chtouki, Y., & Hafid, A. S. (2023). A comprehensive survey on TinyML. *IEEE Access, 11*, 96892–96922.
[4] Liu, H.-I., Galindo, M., Xie, H., Wong, L.-K., Shuai, H.-H., Li, Y.-H., & Cheng, W.-H. (2024). Lightweight deep learning for resource-constrained environments: A survey. *ACM Computing Surveys, 56*(10), Article 267.
[5] Li, M., Liu, Y., Liu, X., Sun, Q., You, X., Yang, H., Luan, Z., Gan, L., Yang, G., & Qian, D. (2020). The deep learning compiler: A comprehensive survey. arXiv preprint arXiv:2002.03794.
[6] David, R., Duke, J., Jain, A., Reddi, V. J., Jeffries, N., Li, J., Kreeger, N., Nappier, I., Natraj, M., Regev, S., Rhodes, R., Wang, T., & Warden, P. (2021). TensorFlow Lite micro: Embedded machine learning on TinyML systems. *Proceedings of Machine Learning and Systems*.
[7] Feng, S., Hou, B., Jin, H., Lin, W., Shao, J., Lai, R., Ye, Z., Zheng, L., Yu, C. H., Yu, Y., & Chen, T. (2022). TensorIR: An abstraction for automatic tensorized program optimization. arXiv preprint arXiv:2207.04296.
[8] 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*, 233–248.
[9] Liu, C., Jobst, M., Guo, L., Shi, X., Partzsch, J., & Mayr, C. (2023). Deploying machine learning models to ahead-of-time runtime on edge using MicroTVM. arXiv preprint arXiv:2304.04842.
[10] Hamdi, M. A., Daghero, F., Sarda, G. M., Van Delm, J., Symons, A., Benini, L., Verhelst, M., Pagliari, D. J., & Burrello, A. (2024). MATCH: Model-aware TVM-based compilation for heterogeneous edge devices. arXiv preprint arXiv:2410.08855.
[11] Zhang, H., Xing, M., Wu, Y., & Zhao, C. (2023). Compiler technologies in deep learning co-design: A survey. *Intelligent Computing, 2*, Article 0040.
[12] Shuvo, M. M. H., Islam, S. K., Cheng, J., & Morshed, B. I. (2023). Efficient acceleration of deep learning inference on resource-constrained edge devices: A review. *Proceedings of the IEEE, 111*(1), 42–91.
[13] Lin, Y., Yang, M., & Han, S. (2021). NAAS: Neural accelerator architecture search. arXiv preprint arXiv:2105.13258.
[14] Benmeziane, H., El Maghraoui, K., Ouarnoughi, H., Niar, S., Wistuba, M., & Wang, N. (2021). A comprehensive survey on hardware-aware neural architecture search. arXiv preprint arXiv:2101.09336.

DOI: 10.54254/2755-2721/2026.CH35364

[15] Lu, L., Guan, N., Wang, Y., Jia, L., Luo, Z., Yin, J., Cong, J., & Liang, Y. (2021). TENET: A framework for modeling tensor dataflow based on relation-centric notation. arXiv preprint arXiv:2105.01892.
[16] Haris, J., Gibson, P., Cano, J., Agostini, N. B., & Kaeli, D. (2021). SECDA: Efficient hardware/software co-design of FPGA-based DNN accelerators for edge inference. *arXiv preprint arXiv: 2110.00478*.
[17] Gholami, A., Kim, S., Dong, Z., Yao, Z., Mahoney, M. W., & Keutzer, K. (2021). A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv: 2103.13630*.
[18] Lin, J., Chen, W.-M., Lin, Y., Gan, C., Han, S., et al. (2021). MCUNetV2: Memory-efficient patch-based inference for tiny deep learning. *arXiv preprint arXiv: 2110.15352*.
[19] Zhang, L., Chen, S., Ma, L., Zhang, Y., Wang, Y., & Liu, Y. (2021). nn-Meter: Towards accurate latency prediction of deep-learning model inference on diverse edge devices. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*.