electronics

MDPI

Review

Edge Intelligence: A Review of Deep Neural Network Inference in Resource-Limited Environments

Dat Ngo [1], Hyun-Cheol Park [1] and Bongsoon Kang [2,*]

[1] Department of Computer Engineering, Korea National University of Transportation, Chungju 27469, Republic of Korea; datngo@ut.ac.kr (D.N.); hc.park@ut.ac.kr (H.-C.P.)
[2] Department of Electronics Engineering, Dong-A University, Busan 49315, Republic of Korea
* Correspondence: bongsoon@dau.ac.kr; Tel.: +82-51-200-7703

Abstract: Deploying deep neural networks (DNNs) in resource-limited environments—such as smartwatches, IoT nodes, and intelligent sensors—poses significant challenges due to constraints in memory, computing power, and energy budgets. This paper presents a comprehensive review of recent advances in accelerating DNN inference on edge platforms, with a focus on model compression, compiler optimizations, and hardware–software co-design. We analyze the trade-offs between latency, energy, and accuracy across various techniques, highlighting practical deployment strategies on real-world devices. In particular, we categorize existing frameworks based on their architectural targets and adaptation mechanisms and discuss open challenges such as runtime adaptability and hardware-aware scheduling. This review aims to guide the development of efficient and scalable edge intelligence solutions.

Keywords: deep neural network; edge intelligence; edge inference; AI accelerator

Academic Editor: Hubert Zarzycki

Received: 15 May 2025
Revised: 15 June 2025
Accepted: 18 June 2025
Published: 19 June 2025

Citation: Ngo, D.; Park, H.-C.; Kang, B. Edge Intelligence: A Review of Deep Neural Network Inference in Resource-Limited Environments. *Electronics* **2025**, *14*, 2495. https://doi.org/10.3390/electronics14122495

1. Introduction

Artificial intelligence (AI) research encompasses a broad range of techniques aimed at enabling machines to perform intelligent tasks. A key subset of AI is machine learning (ML), which focuses on developing algorithms that allow systems to learn from data, or, in other words, to automatically improve through experience. Within ML, there exists the brain-inspired approach that seeks to mimic biological processes and includes spiking neural networks (SNNs) and artificial neural networks (ANNs). A crucial subset of ANNs is deep learning (DL), which leverages multi-layered neural networks to model complex data patterns and achieve state-of-the-art performance in various AI applications [1].

At the core of a deep neural network (DNN) is the artificial neuron (Figure 1a), which is inspired by biological neurons. In this analogy, the inputs represent signals received by the neuron's dendrites, the weights correspond to the synaptic strength that determines the importance of each input, and the output is transmitted through the axon to other neurons. A neural network (Figure 1b) consists of multiple neurons organized into layers: the input layer, which receives raw data; one or more hidden layers, where computations and transformations occur; and the output layer, which generates predictions. A DNN (Figure 1c) extends this structure by incorporating multiple hidden layers, allowing it to learn complex hierarchical features and capture intricate patterns in data, making it particularly powerful for tasks such as image recognition, natural language processing, and autonomous systems.

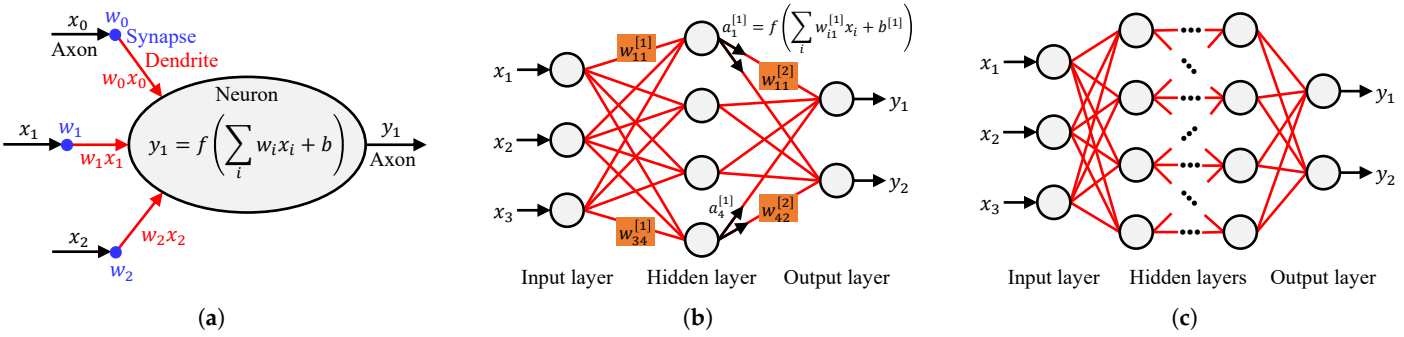

Figure 1. Deep learning fundamentals: (**a**) Neuron. (**b**) Neural network. (**c**) Deep neural network. Notations: x_i = input data, $w_{ij}^{[k]}$ = weights associated with synapse from neuron i to neuron j in layer k, $b^{[k]}$ = bias in layer k, $f(\cdot)$ = activation function, $a_i^{[k]}$ = activation of neuron i in layer k, and y_i = predicted output.

As illustrated in in Figure 2, deep learning consists of two key phases: training and inference. Training is the process where a neural network learns patterns from data by adjusting its weights using optimization techniques such as Adam [2] or stochastic gradient descent (SGD) [3]. During training, large labeled datasets are used, and backpropagation is applied to minimize the error between predictions and actual labels. This phase is computationally intensive, often requiring powerful hardware like GPUs or TPUs. In contrast, inference is the deployment phase, where the trained model is used to make predictions on new, unseen data. Unlike training, inference does not involve updating weights but instead performs forward propagation to generate outputs. As inference requires significantly fewer computations, it can run on edge devices or cloud-based systems, making deep learning models practical for real-world applications.

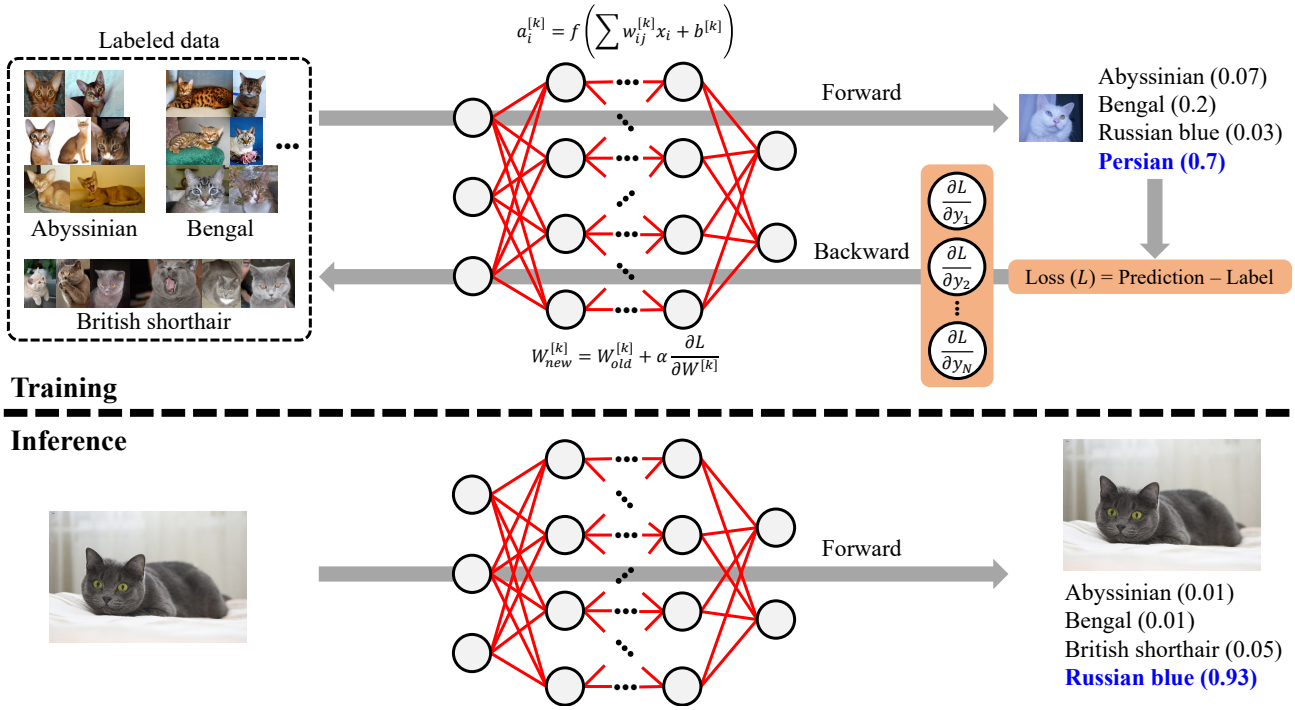

Figure 2. Illustration of training and inference workflow. Notations: $W_{old}^{[k]}$ = current set of weights in layer k, $W_{new}^{[k]}$ = updated set of weights in layer k, $\partial L/\partial W^{[k]}$ = gradient of loss L, and α = learning rate.

Edge inference (EI) occurs directly on local devices such as smartphones, IoT devices, or embedded systems, enabling real-time processing with low latency and reduced

dependency on network connectivity. This approach enhances privacy and security by keeping data on the device but may be constrained by computational resources and power consumption. In contrast, cloud inference leverages powerful servers to process data remotely, allowing for more complex and larger models that require substantial computational resources. Cloud inference can handle multiple requests simultaneously and is ideal for large-scale applications, but it depends on network connectivity and may introduce latency. The focus of this paper is EI due to its numerous practical applications across various industries.

In autonomous vehicles, EI allows for rapid processing of sensor data, such as camera feeds and LiDAR scans, to detect obstacles and make driving decisions instantly. In healthcare, wearable devices use EI to monitor vital signs and detect anomalies, such as irregular heartbeats, without needing continuous internet access. Smart surveillance systems leverage edge AI to analyze video footage in real time, detecting suspicious activities while preserving privacy by keeping sensitive data on the device. In industrial automation, EI enhances predictive maintenance by analyzing sensor data to detect potential equipment failures before they occur. Additionally, in consumer electronics, voice assistants and smart home devices use EI to process commands quickly and respond without delay. These applications highlight the benefits of EI, including low latency, enhanced privacy, and reduced dependency on network infrastructure.

The evolving trend of performing inference on edge devices introduces several critical design challenges that must be addressed to support the rapid advancements in DL algorithms. One major challenge is programmability, as hardware must be adaptable to accommodate frequent updates and changes in DL models. Additionally, real-time performance is a key requirement, especially for edge applications that demand immediate processing and decision-making within strict latency constraints. Another significant concern is power consumption, as edge devices often operate with limited energy resources, making efficiency a top priority. While customized, application-specific hardware can significantly enhance energy efficiency, it often comes at the cost of reduced flexibility and programmability, limiting its ability to adapt to new algorithms. Consequently, edge device design involves a careful balancing act, optimizing trade-offs between performance, power efficiency, latency, energy consumption, cost, and form factor to achieve the best possible results for real-world applications.

The growing demand for intelligent services anywhere, anytime, has driven AI research toward edge computing. Additionally, the widespread adoption of connected gadgets, portable processing platforms, embedded sensing units, and IoT systems generates vast amounts of edge data. By 2030, an estimated 5.8 billion IoT devices will be connected to cyberspace [4]. However, transferring this massive data to the cloud presents challenges in network capacity, bandwidth, and security. Edge computing addresses these issues by processing data closer to its source, enabling real-time, efficient, and secure computing while facilitating EI implementation. Several surveys explore different aspects of DL and EI:

- Studies on DL architectures, algorithms, and applications exist but lack discussions on latest compression techniques for edge intelligence [5–7].
- Research on compression methods, such as low-rank approximation, pruning, and weight sharing, includes hardware acceleration but excludes software frameworks and broader hardware platforms [8–10].
- Reviews on convolutional neural network (CNN) hardware implementation cover ASICs and FPGAs but neglect software–hardware co-design and EI pipelines [11,12].
- ML frameworks and compilers have been reviewed but without considering edge-specific compression and optimization [13].

Despite extensive research, no single survey provides a comprehensive review of DNN inference on resource-limited edge environments, addressing model development, model size reduction, integrated design, and programming environments, targeting hardware architectures, performance benchmarking, and challenges. This paper aims to bridge these gaps in existing surveys by focusing on lightweight models, compression techniques, and software–hardware co-design. Major contributions include

- A comprehensive discussion on EI techniques, including compression, hardware accelerators, and software–hardware co-design, along with a classification of compression strategies, practical applications, benefits, and limitations;
- A detailed review of software tools and hardware platforms with a comparative analysis of available options;
- Highlighting design hurdles and proposing avenues for future investigation in efficient edge intelligence, outlining unresolved problems and potential advancements.

2. Edge Inference of Deep Learning Models

Edge inference of DL models enables real-time processing by reducing latency and bandwidth usage compared to cloud-based inference. It can be categorized into three main strategies: inference on edge servers, on-device inference at edge nodes, and cooperative inference that distributes workloads between edge nodes and edge servers, each offering distinct benefits and trade-offs:

- Inference on edge servers involves deploying DL models on local computing nodes near data sources, such as micro data centers or 5G base stations. This approach offers higher computational power than standalone edge devices, allowing for complex model inference with lower latency compared to cloud solutions. However, it still requires network connectivity, which can introduce variable delays, and may have limited scalability compared to large cloud infrastructures.
- On-device inference on edge nodes runs DL models directly on resource-constrained hardware, such as smartphones, IoT devices, and embedded systems. This method ensures real-time responsiveness, enhances privacy by keeping data local, and reduces reliance on external networks. However, it faces challenges related to limited computational resources, power constraints, and the need for model compression and optimization techniques to fit within hardware capabilities.
- Cooperative inference splits computation between edge nodes and edge servers, where lightweight processing occurs on the node while computationally intensive tasks are offloaded to the edge server. This hybrid approach balances efficiency, latency, and power consumption while leveraging strengths of both local and remote computing. It is particularly beneficial for applications like autonomous vehicles and smart surveillance. However, it introduces additional complexity in workload partitioning and requires reliable network communication between edge node and server.

Figure 3 illustrates the EI paradigm in autonomous vehicles, providing a clearer depiction of the roles of edge devices, edge servers, and the cloud. Autonomous cars perform real-time inference on-board using optimized DL models for tasks such as object detection, lane keeping, and collision avoidance, ensuring rapid decision-making with minimal latency. However, for more computationally intensive tasks like high-resolution map updates, predictive traffic analysis, and vehicle-to-everything (V2X) communication, the vehicle offloads data to nearby edge servers. These edge servers aggregate and process data from multiple vehicles, enabling intelligent driving decisions based on shared situational awareness. The edge servers also facilitate dynamic updates to high-definition maps by integrating sensor data from multiple sources in real time. Finally, the edge servers connect to the cloud, where large-scale data analytics, deep model retraining, and long-term traffic

pattern analysis are conducted. This hierarchical computing framework balances real-time performance, computational efficiency, and data bandwidth while ensuring autonomous vehicles operate safely and intelligently in diverse driving environments.

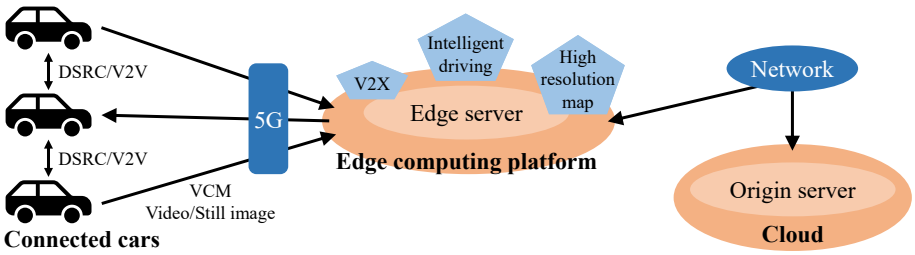

Figure 3. Example of edge inference paradigm in autonomous vehicles. Abbreviations: DSRC = dedicated short-range communications, V2V = vehicle to vehicle, VCM = video coding for machines, and V2X = vehicle to everything.

This paper concentrates especially on inference directly on edge devices. The main strategies for running DL models on resource-limited edge platforms include designing efficient models, applying compression techniques, and utilizing hardware acceleration. As DL models are typically designed for GPUs and TPUs, adapting them into smaller, lightweight architectures or compression pre-trained networks poses significant challenges for effective edge intelligence. Model design can be performed manually (based on experience) or automatically through neural architecture search (NAS). After designing the model, compression techniques are applied to reduce its size in terms of parameters and computation while maintaining acceptable performance. Inference is then executed through a software–hardware co-design approach, where computationally intensive tasks are offloaded to hardware accelerators. This overall design process for inference on edge devices is summarized in Figure 4.

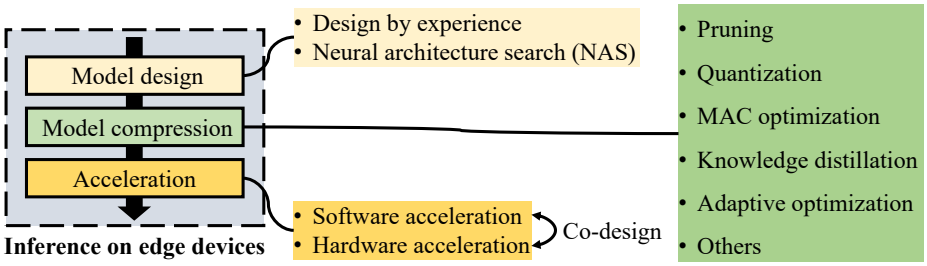

Figure 4. The typical design process for inference on edge devices. Deep learning architectures may be developed either through expert-driven design or automated neural architecture search. These models undergo compression via standalone or combined techniques. Data processed at edge devices is handled through both software and hardware accelerations. MAC stands for multiply–accumulate operation.

2.1. Model Design for Edge Inference

2.1.1. Design by Experience

In CNNs, the majority of computational resources are consumed by convolution operations, making inference speed heavily dependent on how efficiently these convolutions are performed. Designing compact and efficient architectures with fewer parameters and operations is essential for improving performance on resource-limited edge environments.

One common strategy involves replacing large convolution filters with multiple smaller filters applied sequentially, which reduces computational complexity without sacrificing accuracy. For example, MobileNet [14] and Xception [15] simplify convolutions by extensively using 1×1 pointwise convolutions. MobileNet introduces two primary optimizations: reducing input size and employing depthwise separable convolutions—a

combination of depthwise and pointwise convolutions—followed by average pooling and a configurable dense layer. MobileNet's depth multiplier allows for adjusting the number of filters per layer. Depending on the variant, MobileNet configurations range from 0.5 to 4.2 million parameters and 41 to 559 million multiply–accumulate operations (MACs), achieving 50∼70% accuracy.

MobileNetV2 [16] improves on this design by incorporating residual connections and intermediate feature representations via a bottleneck residual block containing one 1×1 convolution and two depthwise separable convolutions. This approach reduces parameters by 30% and computation by 50% relative to MobileNet, while increasing accuracy.

Similarly, SqueezeNet [17] compresses CNN channels using 1×1 convolutions and introduces the "fire module", which first squeezes inputs with 1×1 filters and then expands with parallel 1×1 and 3×3 convolutions. SqueezeNet achieves a $50\times$ reduction in parameters compared to AlexNet [18] while maintaining comparable accuracy, though it tends to consume more energy. It successor, SqueezeNext [19], replaces fire modules with two-stage bottlenecks and separable convolutions, reaching $112\times$ fewer parameters than AlexNet with improved accuracy.

Other approaches focus on channel grouping and shuffling to reduce computation. ShuffleNet [20] uses group convolutions combined with channel shuffling to improve information flow, enhancing accuracy by 3% over MobileNet with similar complexity. Its variant ShuffleNet Sx adds stride-2 convolutions, channel concatenation, and a scaling hyperparameter for channel width. CondenseNet [21] extends this concept by learning filter groups during training and pruning redundant parameters, achieving comparable accuracy to ShuffleNet but with half the parameters. ANTNet [22] further advances group convolution models by surpassing MobileNetV2 in accuracy by 0.8% while reducing parameters by 6%, operations by 10%, and inference latency by 20% on mobile platforms.

From an algorithmic optimization standpoint, the Winograd algorithm [23] effectively accelerates small-kernel convolutions and mini-batch processing by minimizing floating-point and integer operations up to $2.25\times$ [24] and $3.13\times$ [25], respectively. It achieves this through arithmetic transformations that leverage addition and bit-shift operations, which are highly hardware-efficient. FPGA implementations of Winograd convolutions [26] utilize line buffer caching, pipelining, and parallel processing to further speed up computation.

The UniWig architecture [27] integrates Winograd and GEMM-based operations within a unified processing element structure to optimize hardware resource utilization. Additionally, weight-stationary CNNs benefit from reconfigurable convolution kernels [28], which reuse weights to optimize LUT usage on FPGAs and reduce overall computational demands.

Design by experience has proven effective in developing lightweight architectures optimized for edge deployment. Although this approach often involves trade-offs among complexity, energy efficiency, and accuracy, it provides practical solutions for real-time inference on resource-limited devices. Collectively, hand-crafted designs form a solid foundation for efficient edge intelligence. Figure 5 illustrates various convolution techniques discussed above for compact CNN design, while Table 1 offers a summarized comparison for quick reference.

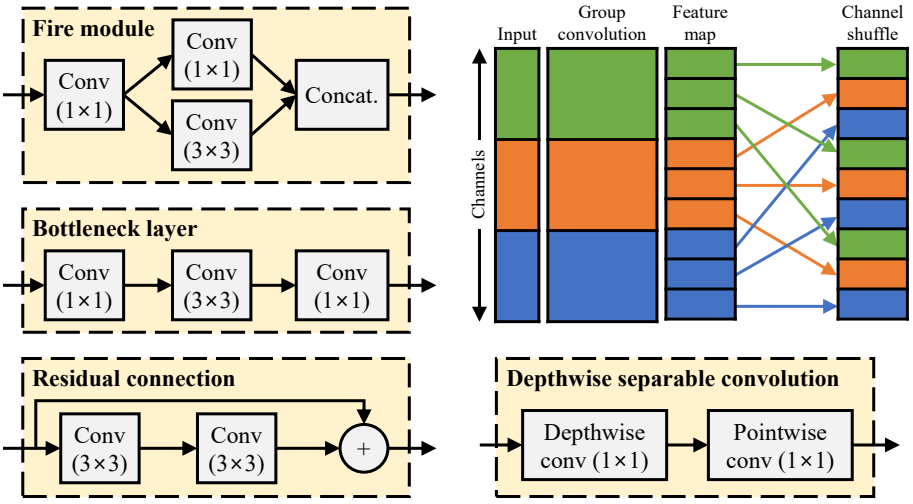

Figure 5. Illustration of various convolution types, including fire modules, bottleneck layers, residual connections, channel shuffling, and depthwise separable convolutions. Abbreviations: Conv = convolution, Concat. = concatenation.

Table 1. Summary of manually designed techniques for efficient edge deployment.

Technique	Effectiveness	Limitations	Trade-Offs	Edge Deployment
MobileNet series	Compact and efficient; accuracy 50∼70% for MobileNet; improved in V2	Limited for very-high-accuracy tasks	Model size vs. accuracy	Highly suitable; widely adopted
Xception	Efficient convolutions with depthwise separable layers	Complex architecture design	Efficiency vs. design complexity	Suitable for edge with careful tuning
SqueezeNet series	Extreme parameter reduction (50× to 112× smaller)	Higher energy consumption	Model size vs. energy usage	Suitable with energy-aware hardware
ShuffleNet/ CondenseNet/ ANTNet	Improved accuracy and speed through group convolutions	More complex training; specialized implementation	Accuracy vs. latency	Highly suitable
Winograd algorithm	Up to 3.13× reduction in computation	Limited to small kernels; complex hardware logic	Hardware complexity vs. speed	Highly suitable for FPGA-/ASIC-based accelerators
UniWig architecture	Combines Winograd and GEMM for better utilization	Requires advanced hardware support	Hardware resource efficiency vs. complexity	Suitable for specialized FPGA/ASIC
Reconfigurable kernels	Efficient weight reuse; reduced LUT usage	FPGA-specific; complex implementation	Hardware flexibility vs. design complexity	Suitable for FPGA-based accelerators

2.1.2. Neural Architecture Search

Neural architecture search (NAS) plays a crucial role in designing efficient DL models, particularly for EI, where computational resources are limited. NAS automates the process of discovering optimal neural network architectures, significantly reducing the reliance on manual design and expert knowledge. The primary components of NAS include the design domain, exploration strategy, and evaluation method. The design domain specifies the range of permissible architectural components—such as layer types, connectivity patterns, and hyperparameters. The exploration strategy determines how candidate architectures are explored within the design domain, using methods such as reinforcement learning, the evolutionary algorithm, or gradient-based optimization. Finally, the evaluation method assesses the performance of candidate architectures, often using accuracy, latency, energy consumption, or a multi-objective metric. Efficient NAS techniques aim to strike a balance

between accuracy and computational efficiency, making them essential for deploying DL models on resource-constrained edge devices.

MobileNetV3 [29], available in both large and small variants, was developed using NAS and the NetAdapt [30] algorithm. As illustrated in Figure 6, NetAdapt is an adaptive method that generates multiple candidate networks based on specific quality metrics. These candidates are tested directly on the target hardware, and the one with the best accuracy is selected. Compared to its predecessor, MobileNetV2, the large version of MobileNetV3 achieves a 25% performance boost without sacrificing accuracy, while the small version improves accuracy by 6.6% with comparable model size and latency.

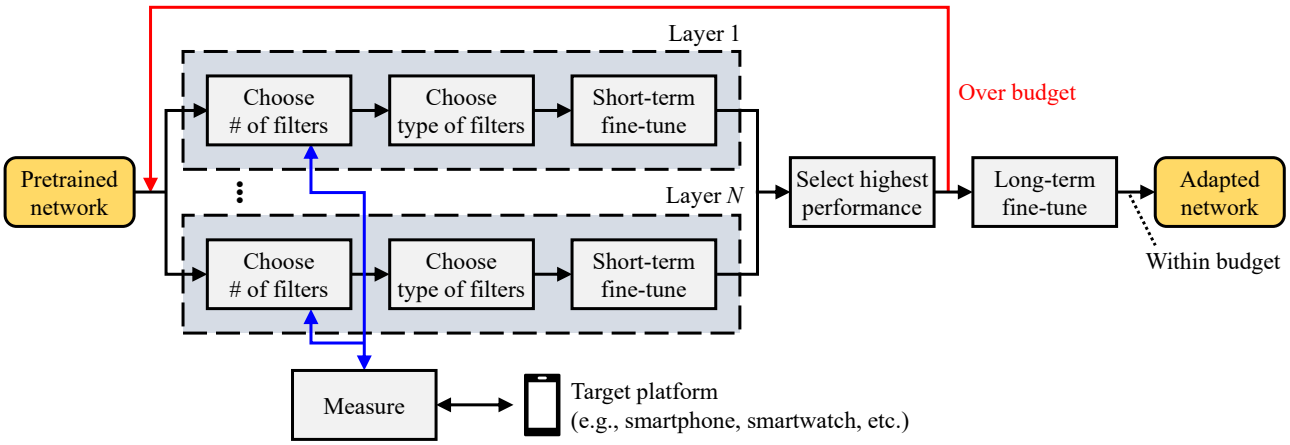

Figure 6. An illustration of the NetAdapt algorithm. At each step, it reduces resource usage by pruning filters layer by layer, selects the variant with the highest accuracy, and fine-tunes the final model once the target constraint is satisfied.

In NetAdapt, hardware costs associated with target platforms are treated as "hard" constraints, meaning that the method solely maximizes model accuracy and, therefore, cannot yield Pareto-optimal solutions. To address this limitation, a hardware-aware NAS (HW-NAS) approach presented in [31] formulates a multi-objective optimization problem considering both model accuracy and computational complexity (measured in FLOPs). This optimization is solved using reinforcement learning (RL), where a neural network predicts the performance of candidate architectures. The resulting solutions—represented as two-objective vectors—are then evaluated against the true Pareto-optimal front to identify the best network.

However, as shown in [32], FLOPs often poorly correlate with real-world latency, as models with identical FLOPs can exhibit different latencies across platforms. To overcome this limitation, MNASNet [33] incorporates measured inference latency—obtained by evaluating candidate networks on actual mobile phones—into the objective function. The reward function combines target latency, measured latency, and model accuracy using a customized weighted product method. As a result, MNASNet achieves image classification with only 78 ms latency, which is 1.8× faster than MobileNetV2 and 2.3× faster than NASNet, while also improving accuracy by 0.8% and 1.2%, respectively.

In general, hardware constraints such as FLOPs and latency are not differentiable with respect to model parameters, hindering the application of gradient-based search methods. This constraint has led to the widespread use of RL-based searches in HW-NAS. FBNet [34] addresses this by employing the Gumbel-Softmax function, making the joint accuracy–latency objective function continuous and thus amenable to gradient-based optimization. In FBNet, the latency of each candidate architecture is computed as the sum of its component latencies, which are pre-stored in a look-up table. To reduce training overhead, candidate models are trained on proxy datasets such as MNIST and CIFAR-10.

While training on proxy datasets can reduce costs, it may not generalize well to large-scale tasks. ProxylessNAS [35] mitigates this by enabling architecture search directly on large-scale datasets and target hardware platforms. It adopts binarized paths to activate and update only a single path during training, thereby minimizing memory usage. Furthermore, by modeling inference latency as a continuous function of architectural parameters, ProxylessNAS facilitates efficient gradient-based search with rapid convergence.

Beyond gradient- and RL-based strategies, recent research [36] explores the use of genetic algorithms for NAS. In this approach, the fitness function combines performance estimation and hardware cost penalties. Performance is approximated by the similarity between hidden layer representations of a reference model and those of candidate networks, while hardware cost is represented by either FLOPs or latency. Experimental results demonstrate significant improvements in efficiency, achieving a 40× reduction in search time and a 54× decrease in CO_2 emission compared to ProxylessNAS on the ImageNet dataset.

The integration of hardware constraints into NAS frameworks addresses the accuracy degradation often observed when NAS-generated models are deployed across heterogeneous edge devices. In [37], this issue is tackled by the once-for-all (OFA) network. During inference, only a selected sub-network within the OFA model is activated to process input data. The training strategy optimizes the performance of all potential sub-networks by dynamically sampling different architectural paths. A subset of these paths is then used to train both accuracy and latency predictors. Based on a given hardware target and latency constraint, RL-based architecture search is applied to identify an optimal sub-network. The OFA model is structured into five sequential units with progressively decreasing feature map sizes and increasing channel widths. Each unit supports configurations of two, three, or four layers; each layer may contain three, four, or six channels and uses kernels of size three, five, or seven. This configuration enables the generation of approximately 2×10^{19} sub-networks, all sharing the same weights (about 7.7 million parameters).

In summary, the evolution of HW-NAS techniques reflects a growing emphasis on jointly optimizing model accuracy and deployment efficiency across diverse edge platforms. Early approaches such as NetAdapt and MNASNet highlight the shift from theoretical metrics such as FLOPs to real-world latency considerations. Subsequent methods—FBNet, ProxylessNAS, and OFA—further advance this trend by integrating gradient-based optimization, large-scale training, and flexible sub-network deployment. These developments demonstrate that effective HW-NAS design must account for both hardware variability and scalability, enabling practical and efficient AI deployment in resource-constrained environments.

2.2. Model Compression for Edge Inference

2.2.1. Pruning

Modern DL models are often heavily overparameterized. Pruning is an effective model compression strategy that eliminates less critical weights or filters from a pre-trained network. This process not only reduces memory requirements but also enhances inference speed.

Pruning methods are typically categorized into unstructured and structured types [38]. Unstructured pruning removes individual weights while keeping neurons active as long as they maintain at least one remaining connection. For example, a sparsification method for LSTM networks implemented on FPGAs [39] divides matrices into banks with an identical number of non-zero elements, retaining larger weights to maintain accuracy. However, unstructured pruning often leads to irregular memory access and computation patterns, which can hinder hardware parallelism [39,40].

In contrast, structured pruning [41] removes entire groups of weights, such as adjacent weights, sections of a filter, or even complete filters [42]. A structured pruning method for CNNs [43] selectively removes filters to boost network efficiency without significant

accuracy degradation. This technique trains an agent to prune layer by layer, retraining after each pruning step before proceeding to the next layer. Structured pruning better supports hardware acceleration by aligning with data-parallel architectures [44], improving both compression rates and reducing storage demands.

Hybrid pruning combines both structured and unstructured approaches for optimized hardware-friendly models. For example, one approach gradually removes a small portion of the least significant weights from each layer, guided by a predefined threshold, and follows up with retraining. Neurons that lose all connections are subsequently removed. Weight pruning effectively compresses shallow networks without notable accuracy loss. An iterative vectorwise sparsification strategy for CNNs and RNNs [40] shows faster responses with negligible accuracy degradation, achieving 75% sparsity. However, filter pruning in deeper networks tends to cause more significant accuracy drops.

Figure 7 illustrates various pruning methods and their impact on network structures. Currently, there are three main strategies for creating sparse networks through pruning.

A. Magnitude-Based Approach: The contribution of each weight in a network is proportional to its absolute value. To induce sparsity, weights below a predefined threshold are removed. Magnitude-based sparsity not only reduces model size but also accelerates convergence. In [45], a pruning strategy is applied incrementally across layers, where low-magnitude weights are pruned and the model is retrained to recover lost accuracy. Over several iterations, the remaining weights are fine-tuned. Using this approach, AlexNet achieves a 9× reduction in weight count and a 3× reduction in MAC operations. The pruning effect is more pronounced in fully connected layers—with reductions as high as 9.9×—compared to 2.7× in convolutional layers. Experimental results with AlexNet and VGGNet [46] show that about 80% of weights can be pruned with fine-tuning and around 50% without it.

Dynamic pruning methods have also been explored. In [47], a dynamic pruning approach reduces inference latency for transformer models on edge hardware. This method decreases both MAC operations and tensor size while preserving up to 98.4% accuracy in keyword spotting tasks. When tolerating up to 4% accuracy loss, the technique achieves up to 94% reduction in operations and delivers up to 16× faster multihead self-attention inference compared to a standard keyword transformer. Another example is ABERT [48], a pruning scheme tailored for BERT models, which automatically discovers optimal sub-networks under pruning ratio constraints. The deployment of the pruned network on a Xilinx Alveo U200 FPGA yields a 1.83× performance gain over the baseline BERT model.

One key limitation of magnitude-based pruning lies in selecting an appropriate threshold. Using a single threshold across all layers often leads to suboptimal results, as parameter magnitudes vary between layers. While setting different thresholds per layer offers a solution, it is difficult to implement. Differentiable pruning [49] addresses this limitation by introducing a learnable pruning process. Additionally, ref. [50] introduces a Taylor series-based criterion to evaluate neuron importance in CNNs, formulating pruning as a joint optimization task over both network weights and pruning decisions. Gate Decorator [51], a global structured pruning method, scales channels in CNNs and removes filters when their scaling factors drop to zero.

B. Regularization-Based Approach: These techniques incorporate a regularization term into the loss function to enforce the desired level of sparsity [52]. As the model trains, it gradually evolves toward a sparse sub-network that meets the targeted sparsity ratio while maintaining high accuracy. A primary benefit of regularization-based approaches is that they preserve the integrity of the weights, resulting in better overall accuracy. However, a key drawback is that these methods often require a large number of iterations to fully converge.

C. Energy-Based Approach: The technique introduced in [53] prunes weights based on estimated energy consumption, factoring in the number of MAC operations, data sparsity, and memory hierarchy movement. It achieves a 1.74× improvement in energy efficiency on AlexNet when compared to traditional magnitude-based pruning.

A significant challenge with unstructured pruning is that it results in sparse weight matrices, which complicates efficient hardware deployment. While this form of pruning is generally more effective in fully connected layers, it does not always translate into reduced latency. The irregular pattern of removed weights can hinder matrix multiplication performance. Other limitations include limited transferability across model architectures, inefficient on-chip memory access, a dependency on fine-tuning to recover accuracy, and imbalanced computational workloads that reduce parallelism.

Zero-skipping techniques [54] attempt to bypass unnecessary multiplications involving zeros created by pruning. However, skipping zeros in weights and activations can lower overall performance efficiency [55]. These methods also demand large on-chip memory to fully leverage hardware parallelism. To address this, weights can be stored in a dense format [56], reducing memory overhead. Map batching [57] further improves efficiency by reusing kernels in dense layers, avoiding frequent transfers to external memory.

Despite their benefits, pruning entire filters or channels in CNNs often results in substantial accuracy degradation. Therefore, CNNs are commonly compressed using unstructured pruning, which creates sparsely populated weight tensors while preserving the original network architecture.

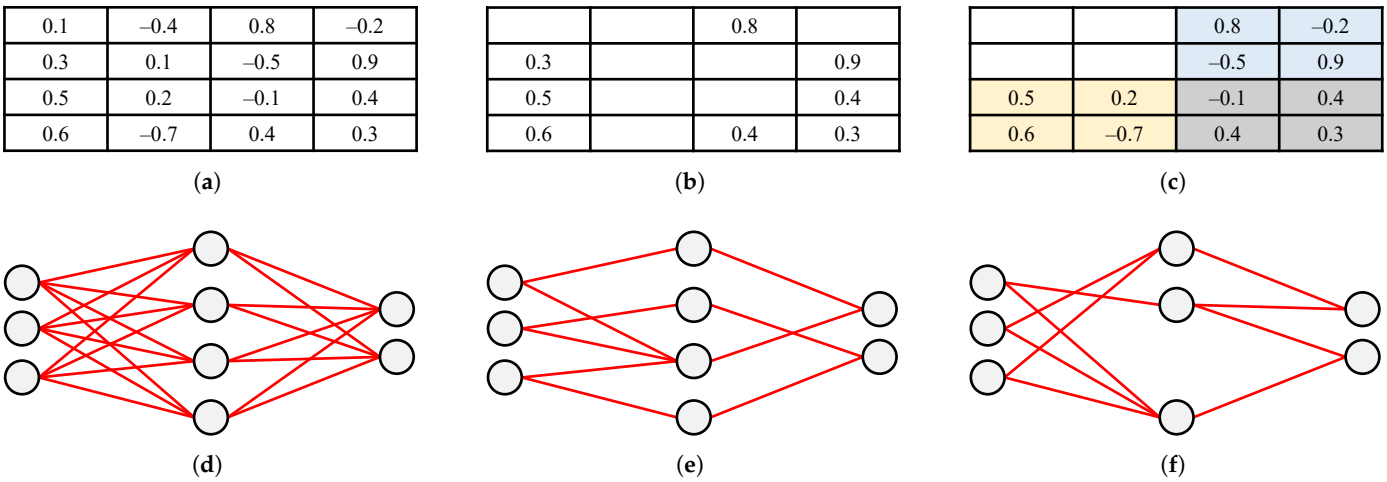

Figure 7. Visualization of different pruning methods: (**a**) Original set of weights. (**b**) Unstructured pruning applied to lower half based on magnitude. (**c**) Structured pruning by removing 2 × 2 group with smallest average. (**d**) Basic neural network before pruning. (**e**) Unstructured pruning of (**d**) by removing 50% of weights. (**f**) Neuron pruning applied to network in (**d**).

D. Hardware Support for Structured and Unstructured Pruning: While pruning effectively reduces model size and computation, its real-world impact on EI performance depends on the underlying hardware's ability to exploit sparsity. Edge accelerators such as NPUs, FPGAs, and ASICs vary widely in their support for different sparsity patterns.

Structured pruning is typically more hardware-friendly because it preserves regularity in memory access and computation. Removing entire channels or filters allows compilers to simplify tensor shapes and generate optimized dense matrix operations without requiring major changes in the hardware pipeline. In contrast, unstructured pruning, which removes individual weights, often results in irregular memory access and control flow. Exploiting such sparsity demands dedicated support, such as sparse matrix storage, zero-skipping

hardware, or specialized sparse compute units. These features are not widely available in most edge devices.

Table 2 summarizes how structured and unstructured sparsity are supported across representative edge accelerators. Given these trends, recent work emphasizes hardware-aware pruning, which aligns pruning granularity with accelerator architecture. Co-design efforts such as SCNN [58], EyerissV2 [59], and Cambricon-S [60] demonstrate how custom accelerators can natively support block- or vector-level sparsity, balancing flexibility and performance. However, these designs are yet to be widely adopted in commercial edge hardware.

Table 2. Comparison of hardware support for sparsity patterns on edge accelerators. NA stands for not available.

Sparsity Type	Accelerator	Hardware Support	Latency Benefit	Energy Efficiency	Remarks
Structured	NVIDIA Jetson	TensorRT [61] with channel pruning	Moderate	Moderate	Works well with fused layers and dense GEMMs
	Google Edge TPU	Filter pruning support	High	High	Prefers regular conv [a] shapes, no dynamic sparsity
	ARM Ethos-U	Native pattern support	High	High	Integrates well with CMSIS-NN pruning methods [62]
	Xilinx FPGA	Manual scheduling	High	Very high	Custom dataflow optimizations possible
Unstructured	NVIDIA Jetson	Sparse GEMM via cuSPARSE [63]	Limited	Slight	Irregular compute, limited speedup without 2:4 support
	Google Edge TPU	No direct support	Negligible	Negligible	Requires retraining for structured sparsity
	ARM Ethos-U	Unsupported	NA	NA	Compilation fails without dense fallback
	Xilinx FPGA	Sparse matrix engines	Moderate	Moderate	Requires high memory bandwidth, controller overhead

[a] Convolutional layers.

2.2.2. Quantization

Most DL models rely on floating-point representations for weights and activations, which preserve detailed information but significantly slow down processing. However, as DNNs are designed to capture key features and are inherently robust to small variations, they can tolerate minor errors introduced by quantization. Quantization reduces the precision of model parameters and activations, minimizing the demand for high-cost floating-point operations. By decreasing the bit width, quantization enables faster and more efficient integer-based computations in hardware.

Quantization provides three main advantages: (i) it reduces memory consumption by representing data with fewer bits; (ii) it simplifies computations, leading to faster inference times; and (iii) it boosts energy efficiency. Data can be quantized either linearly or nonlinearly, and the bit width used for quantization can be fixed across the model or vary between layers, filters, weights, and activations. In fixed-bit quantization, a uniform bit width is applied across the network, whereas variable quantization allows different parts of the network to use different precisions.

Replacing 32-bit single-precision floating-point (FP32) operations with lower-precision formats—such as half-precision (FP16 [64]), reduced-precision (FP8 [65]), or fixed-point representations [66]—greatly simplifies DL computations. For example, 8-bit integer (INT8) addition and multiplication can be 3.3× and 15.5× more energy-efficient than 32-bit equivalents [67]. As a result, AlexNet can achieve under 1% accuracy loss [68] when reduced to 4∼9 bits. Prior work has shown that using 8-bit activations and weights can deliver a 3∼4× memory reduction and a 10× speedup without accuracy degradation [69,70].

Table 3 summarizes representative quantization methods, categorized by precision type—fixed-point, floating-point, variable, reduced, and nonlinear. Fixed-point methods such as weight fine-tuning [71] and contrastive-calibration [72] balance accuracy and simplicity, showing only 0.6∼2.3% top-1 accuracy loss. Floating-point methods (for example, FP8 [73]) outperform the traditional INT8 quantization with less than 1% accuracy drop while supporting a wider range of quantized operations. Variable precision schemes such as Stripes [74] and Bit Fusion [75] provide bit-level flexibility but lack clear reports on accuracy metrics. Reduced precision approaches, including BinaryNet [76] and XNOR-

Net [77], drastically lower bit widths to 1∼2 bits, gaining speedups at the cost of significant accuracy loss (up to 18.1%). Nonlinear methods such as Incremental Network Quantization (INQ) [78] and Deep Compression [79] use weight clustering and pruning strategies to achieve compression with minimal accuracy degradation.

Table 3. A summary of representative studies on the quantization of deep learning models. FP32 denotes the 32-bit floating point representation, NA stands for not available, QAT is the abbreviation for quantization-aware training, and PTQ represents post-training quantization.

Quantization Technique	Model/Method	Bit Width		Top-1 Accuracy Drop on ImageNet [a]	Calibration Complexity
		Weights	Activations		
Fixed-point precision	Weight fine-tuning [71]	8	8	0.6∼2.3%	QAT
	Contrastive-calibration [72]	2, 4	2, 4	1.6%	PTQ
Floating-point precision	FP8 [73]	8	8	−0.15∼0.11%	PTQ
Variable precision	Stripes [74]	16	1∼16	NA	NA
	UNPU [80]	1∼16	16	NA	NA
	Loom [81]	1∼16	1∼16	NA	NA
	Bit Fusion [75]	1, 2, 4, 8, 16	1, 2, 4, 8, 16	NA	NA
	BitBlade [82]	1, 2, 4, 8, 16	1, 2, 4, 8, 16	NA	NA
Reduced precision	BinaryConnect [83]	1	FP32	NA	QAT
	Ternary Weight Network [84]	2 [b]	FP32	2.3%	QAT
	Trained Ternary Quantization [85]	2 [b]	FP32	−0.3∼1%	QAT
	XNOR-Net [77]	1 [b]	1 [b]	18.1%	QAT
	BinaryNet [76]	1	1	NA	QAT
	DoReFa-Net [86]	1 [b]	2 [b]	3.3%	QAT
	Quantized Neural Network [87]	1	2 [b]	5.2∼5.57%	NA
	HWGQ-Net [88]	1 [b]	2 [b]	5.8∼8.4%	QAT
	SmoothQuant [89]	8	8	NA	PTQ
	PD-Quant [90]	2, 4	2, 4	1.29%	PTQ
Nonlinear quantization	Incremental Network Quantization [78]	5	FP32	−1.59∼ − 0.13%	QAT
	Deep Compression [79]	8 (conv [c]), 4 (dense [d])	16	0.01%	NA
		4 (conv), 2 (dense)	16	1.99%	NA

[a] The quantized model's performance is compared with that of the FP32 implementation. [b] FP32 for the first and last layers. [c] Convolutional layers. [d] Fully connected layers.

Binary networks represent an extreme form of quantization, replacing multiplications with 1-bit XNOR operations. Techniques such as BinaryConnect [83] and BinaryNet [76] compress model size up to 32× and enable fast inference but suffer from large accuracy drops on complex datasets. To mitigate this, DoReFa-Net [86], QNN [87], and HWGQ-Net [88] use hybrid precision strategies, maintaining 1-bit weights with multi-bit activations. Ternary quantization methods such as the Ternary Weight Network [84] and Trained Ternary Quantization [85] introduce a zero weight level to reduce information loss, achieving accuracy drops ranging from −0.3% to 2.3%, where the negative value denotes an improvement over the full-precision baseline.

The reduced precision methods discussed above follow the quantization-aware training (QAT) strategy, which requires retraining the quantized model and is therefore time-consuming. An alternative approach is post-training quantization (PTQ), as demonstrated by SmoothQuant [89] and PD-Quant [90]. PTQ methods eliminate the need for retraining; instead, they rely on calibration using a small set of data samples. SmoothQuant enhances INT8 quantization for large language models by smoothing weight and activation distributions, achieving negligible accuracy loss without fine-tuning. PD-Quant introduces a prediction difference-based calibration approach tailored to ultra-low precision (2-bit) quantization, yielding a top-1 accuracy loss of only 1.29% for extreme bit widths.

Hardware-friendly designs such as Stripes [74] and Loom [81] use bit-serial operations to reduce power by substituting multipliers with AND gates and adders. UNPU [80] lever-

ages 1∼16-bit weights with fixed activations, while Loom applies serial quantization for both trading latency for hardware efficiency. Bit Fusion [75] dynamically fuses processing elements to match operand bit widths, while BitBlade [82] removes shift–add overhead using direct bitwise summation.

Nonlinear quantization techniques exploit weight and activation distributions to minimize quantization error, often using logarithmic spacing or lookup tables. INQ [78] applies iterative quantization with retraining to preserve performance, while Deep Compression [79] clusters and prunes weights for minimal accuracy degradation. These methods are especially effective for edge deployment due to their low calibration needs and robust accuracy.

While binarization suits small datasets or neuromorphic chips, it remains challenging for large-scale tasks due to accuracy loss. Accuracy improvements often rely on keeping the first and last layers in higher precision or combining multiple quantization strategies.

Quantization has also been extended to RNNs and Transformers. SpeechTransformer [91] and vision transformers [92] benefit from integer-only quantization, achieving 4× reductions in parameter count and 4.11× inference speedups. Despite these advantages, quantization introduces trade-offs including precision loss, architectural constraints, and gradient computation challenges. Moreover, quantized models have been shown to be more vulnerable to distribution shifts compared to their full-precision counterparts [93]. This vulnerability, however, can be mitigated through carefully designed post-training quantization strategies.

In summary, quantization remains a powerful tool for model optimization, with various techniques offering distinct trade-offs. Table 3, along with the accompanying comparison and deployment analysis, offers a roadmap for selecting suitable methods based on accuracy, hardware efficiency, and deployment constraints.

2.2.3. Multiply-and-Accumulate Optimization

Multiplication operations account for a substantial portion of the computational burden in DL algorithms, particularly during inference. Therefore, techniques that reduce or eliminate multiplications can dramatically improve efficiency—especially in resource-limited edge environments.

One effective approach is DiracDeltaNet [94], which replaces spatial convolutions with lightweight shift operations and depthwise convolutions followed by efficient 1×1 convolutions. This strategy reduces the parameter count by a factor of 48× and the number of operations by 65× for the VGG-16 model, while maintaining competitive accuracy on the ImageNet benchmark. Its structured pruning and minimal reliance on multiplications make it particularly appealing for FPGA-based deployments.

A more radical strategy is proposed in [95], where all multiplications and nonlinear functions are precomputed and stored in LUTs. During inference, the model relies solely on fixed-point additions and table indexing. Activation functions are replaced with quantized approximations. This technique eliminates floating-point operations entirely, achieving extremely low computational overhead. However, it comes at the cost of potentially large memory requirements due to LUT size, which may limit scalability for more complex models.

In contrast, the Alphabet Set Multiplier (ASM) introduced in [96] reduces multiplication cost by reusing partial products. It precomputes a compact set of "alphabets" (small multiplication results), which are then dynamically combined using shifters, adders, and selection units to reconstruct full multiplications. ASM is particularly effective for fixed-weight models, where computation can be shared across multiple operations. This method strikes a balance between efficiency and accuracy, making it a strong candidate for embedded systems and ASIC-based deployments.

For edge deployment, DiracDeltaNet [94] offers the best overall balance of performance, hardware simplicity, and model accuracy. However, for applications requiring extremely low power consumption and where memory is not a bottleneck, the LUT-based approach [95] may be preferable. ASM [96] is ideal in scenarios where custom hardware design is feasible and shared computation can be exploited.

2.2.4. Knowledge Distillation

Knowledge distillation (KD) is a widely used model compression strategy that enables a lightweight DL model to emulate the behavior and performance of a larger, more accurate model [97]. This typical setup includes a teacher–student paradigm, where a large, high-capacity teacher network guides the training of a smaller student model designed for efficient inference on constrained hardware platforms. An overview of this process is shown in Figure 8.

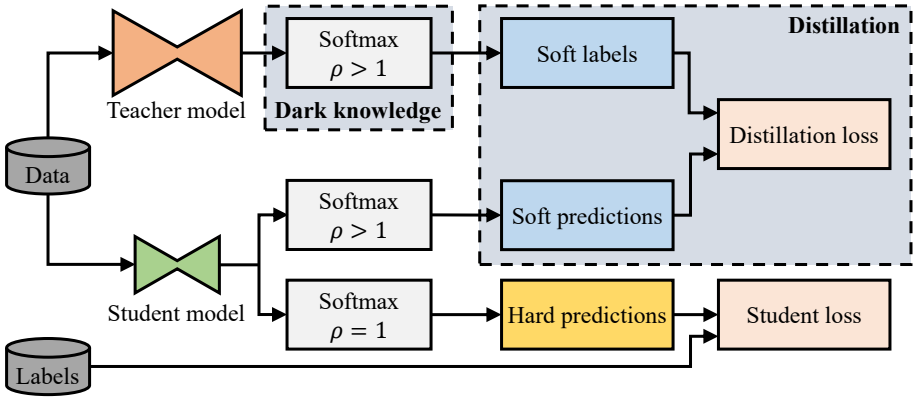

Figure 8. Illustration of knowledge distillation. ρ denotes the softmax temperature.

The essence of KD lies in transferring learned representations from teacher to student via a carefully designed loss function. Originally proposed in [98] and extended in [99], the approach often involves the student learning to match the teacher's softmax output probabilities, which provide richer supervision than hard labels. For example, in speech recognition tasks, KD has yielded a 2% increase in accuracy, enabling the student to approach the performance of a large ensemble of teacher models. Beyond soft labels, KD techniques have evolved to incorporate internal features—such as intermediate activations [100], neuron outputs [101], or feature maps [102]—which help the student to gain deeper insights from the teacher's representations.

Numerous strategies have emerged to tailor student model architecture for edge deployment. For example, DeepRebirth [103] compresses networks by merging convolutional layers with adjacent non-parametric layers (such as pooling and normalization), followed by layerwise fine-tuning, resulting in a 3× speedup and 2.5× reduction in memory usage. In contrast, Rocket Launching [104] synchronously trains teacher and student networks to accelerate convergence, reducing training overhead.

In natural language processing (NLP) for edge environments, KD has proven highly effective. For example, TinyBERT [105], a distilled version of BERT, retains 96.8% of the original accuracy while shrinking the model size by 7.5× and speeding up inference by 9.4×. Similarly, sentence-level distillation techniques [106] have enabled the creation of lightweight sentence transformers that maintain competitive performance.

Student model design can also trade off width for depth, as in FitNets [102], where a deeper, narrower student learns from intermediate hints provided by the teacher. This setup matches the teacher's performance on CIFAR-10 while benefiting from reduced model size and increased inference speed.

Further developments include mutual learning [107] (students co-learn), teacher–assistant frameworks [108] (a multi-stage distillation process), and self-distillation [109] (models teach themselves using their own past outputs). KD has even been extended to dataset distillation, where small synthetic datasets replicate the training behavior of full datasets, significantly reducing training cost and data storage needs [110,111].

Among the KD-based methods, TinyBERT [105] stands out as the most suitable for edge deployment in NLP tasks, offering a compelling balance between size, speed, and accuracy. In vision tasks, DeepRebirth [103] and FitNets [102] provide efficient alternatives with strong inference performance and low overhead. Nevertheless, care must be taken when deploying KD-trained models across edge environments due to their sensitivity to distribution shifts and reliance on teacher-specific behavior. However, KD remains a powerful and versatile compression technique, particularly when inference efficiency and model compactness are paramount.

2.2.5. Adaptive Optimization

Adaptive DL compression techniques dynamically adjust computational workloads during inference in response to the complexity of each input, enhancing efficiency while preserving performance. Unlike static compression methods such as quantization, pruning, or knowledge distillation—which apply uniform reductions regardless of input difficulty—adaptive approaches dynamically adjust computation to suit varying inference demands. For example, processing a simple background frame in a surveillance video requires far less computation than detecting objects in a crowded scene. Similarly, basic sentence comprehension in NLP may not demand full model depth compared to syntactically ambiguous inputs.

One such technique is frame skipping in video analysis. NoScope [112] employs a lightweight difference detector to identify significant changes between consecutive frames, allowing the system to bypass redundant computations when frames are similar. This approach has achieved up to 15.5× speedup in binary classification tasks over fixed-angle webcam and surveillance video while maintaining accuracy within 1∼5% of state-of-the-art networks.

Another method involves using dual-model systems comprising a “small” and a “large” network [113]. The small model processes inputs first, and if its confidence in the prediction is high, the result is accepted. If not, the input is passed to the larger, more complex model for further processing. This strategy can lead to significant energy savings, up to 94.1% in digit recognition tasks, provided the larger model is infrequently activated.

Early exiting (illustrated in Figure 9a) is a technique where intermediate classifiers are added to a DNN, allowing the model to terminate inference early if a confident prediction is made at an earlier stage. BranchyNet [114] is an implementation of this concept, adding multiple exit points to the base network and using entropy of the softmax output to decide whether to exit early or continue processing. This method reduces latency and computational load, especially when deeper layers are seldom needed.

Extending early exiting to distributed systems, DDNNs [115] incorporate exit points at various levels, such as cloud, edge server, and end-device. This hierarchical approach allows for efficient processing by exiting at the most appropriate level based on the input’s complexity. Similarly, DeepIns [116] applies this concept in industrial settings, collecting data via edge sensors and determining processing at the edge or cloud level based on confidence scores.

SkipNet [117] (illustrated in Figure 9b) introduces skippable tiny NNs, or gates, within the architecture, enabling the model to bypass certain layers during inference based on input characteristics. This dynamic routing conserves computational resources with minimal

impact on accuracy. In practical applications, such as smart assistants and wearable devices, hierarchical inference is employed where a lightweight model handles initial processing, and more complex analysis is offloaded to cloud-based models as needed [118,119].

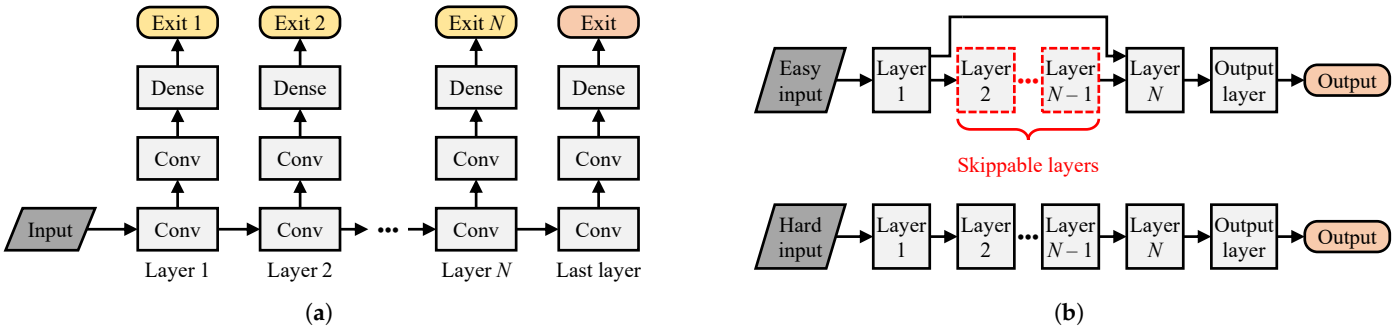

Figure 9. An illustration of the early exiting technique (**a**) and the SkipNet architecture (**b**).

Table 4 summarizes representative adaptive model compression methods, highlighting their design objectives, underlying architectures, compression techniques, performance outcomes (lossless, lossy, or enhanced), and experimental results. Table 5 lists popular lightweight DL models and compares them in terms of parameter count and computational cost, most of which stem from applying such compression methods to large-scale DL models.

Adaptive compression methods offer a nuanced and intelligent approach to optimizing inference workloads, with each method presenting a distinct balance between computational efficiency and accuracy retention:

- Most energy-efficient: Dual-model systems [113] and early exiting methods [114,115] stand out, offering substantial energy savings with little accuracy compromise.
- Best for latency-sensitive edge deployment: SkipNet [117] and BranchyNet [114] reduce unnecessary computation in real time, making them well-suited for wearable devices and real-time analytics.
- Most suitable for dynamic environments: DDNNs [115] offer scalable, hierarchical processing ideal for industrial and IoT scenarios, albeit with increased system complexity.

Overall, early exiting such as in BranchyNet [114] emerges as a balanced choice for edge deployment, combining low latency, energy savings, and minimal accuracy loss, especially when computation budgets vary with input complexity.

2.3. Sparsity Handling for Efficient Edge Inference

Pruning has been shown to eliminate over 90% of weights in dense layers and about 60% in convolutional layers of typical models, leading to a substantial weight sparsity [45]. Even compact models such as MobileNetV2 [16] and EfficientNet-B0 [120] benefit substantially, achieving 50∼93% sparsity, translating into 2.5× to 4.2× reductions in MACs [121]. In natural language processing, pruning large-scale models such as Transformer [122] and BERT [123] yields 80% [124] and 93% [125] sparsity, respectively, enabling 4.8× to 12.3× reductions in MACs. For example, SpAtten [126] exploits structural sparsity in attention mechanisms to reduce both computation and memory access by 3.8× and 1.1×, respectively.

Recurrent models such as GRU [127] and LSTM [128] also respond well to pruning, tolerating over 80% weight sparsity with negligible accuracy loss. Techniques such as MASR [129] introduce activation sparsity of approximately 60% by modifying normalization schemes, while DASNet [130] leverages dynamic sparsification to reduce MACs by 27% in AlexNet and 12% in MobileNet, incurring less than 1% accuracy loss.

Table 4. Summary of representative model compression techniques. ↑ and ↓ denote "increase" and "decrease", respectively.

Type	Technique	Base Model	Objective	Results	Ref.
Lossy	Binarization	AlexNet	Speedup, model size ↓	58× faster, 32× smaller	[77]
	Pruning	AlexNet	Energy ↓	3.7× energy ↓	[53]
	Pruning	VGG-16	Speedup	10× faster	[50]
	Pruning	NIN	Speedup	1.24× faster	[131]
	Low-rank approximation	VGG-S	Energy ↓	4.26× energy ↓	[132]
	Low-rank approximation	CNN	Speedup	2× faster	[133]
	Low-rank approximation	CNN	Speedup	4.5× faster	[134]
	Knowledge distillation	GooLeNet	Speedup, memory ↓	3× faster, 2.5× memory ↓	[103]
	Memory access optimization	VGG	Power, performance trade-off	83% accuracy	[135]
Lossless	Pruning, quantization, Huffman coding	VGG-16	Model size ↓	49× smaller	[79]
	Pruning	VGG	Computation ↓	5× computation ↓	[136]
	Pruning	AlexNet	Memory ↓	13× memory ↓	[45]
	Compact network design	AlexNet	Model size ↓	50× smaller	[17]
	Kernel separation, low-rank approximation	AlexNet, VGG	Speedup, memory ↓	13.3× faster, 11.3× memory ↓	[137]
	Low-rank approximation	VGG-16	Computation ↓	9× computation ↓	[138]
	Knowledge distillation	WideResNet	Memory ↓	40% memory ↓	[139]
	Adaptive optimization	MobileNet	Speedup	1.7× faster	[140]
Enhanced	Quantization	Faster R-CNN	Model size ↓	4.16× smaller	[70]
	Pruning	ResNet	Speedup	20× faster	[141]
	Compact network design	SqueezeNet	Model size ↓	5.17× smaller	[142]
	Compact network design	YOLOv2	Speedup, model size ↓	34% faster, 15.1× smaller	[143]
	Knowledge distillation	ResNet	Accuracy ↑	1.1% ↑	[144]
	Knowledge distillation, regularization	NIN	Memory ↓	32.28× memory ↓	[145]
	Dataflow optimization	DNN	Speedup, energy ↓	58× faster, 104× energy ↓	[146]

Table 5. Summary of representative lightweight models, sorted by number of operations.

Model	Weight Count (millions)	Operation Count (billions)
MobileNetV3-Small [29]	2.9	0.11
MNASNet-Small [33]	2.0	0.14
CondenseNet [21]	2.9	0.55
MobileNetV2 [16]	3.4	0.60
ANTNets [22]	3.7	0.64
NASNet-B [147]	5.3	0.98
MobileNetV1 [14]	4.2	1.15
PNASNet [140]	5.1	1.18
SqueezeNext [19]	3.2	1.42
SqueezeNet [17]	1.25	1.72

In CNNs, sparsity is also induced during both training and inference stages. For example, MobileNetV2 [16] can skip 20% of MACs due to activation sparsity. Overall, pruning can lead to 40∼50% MAC reductions and significant energy savings, especially when paired with sparsity-aware hardware [148].

2.3.1. Hardware Implications: Static versus Dynamic Sparsity

Pruning methods require hardware accelerators that can effectively exploit sparsity, which can be structured or unstructured and processed using static or dynamic techniques:

- Static sparsity: Devices such as Cambricon-X [149] pre-identify zero weights during training and store their locations alongside indices, enabling efficient skipping of useless computations. However, these devices often require complex memory indexing that limits their real-time adaptability.
- Dynamic sparsity: Accelerators such as ZENA [150], SNAP [151], and EyerissV2 [59] detect and process non-zero elements at runtime. While more flexible, they require sophisticated control logic, increasing hardware complexity and potentially raising latency if no optimized properly.

Structured sparsity, such as block sparsity [152] or vectorwise block sparsity [153], alleviates indexing challenges by grouping non-zero elements under shared indices. This improves processing element (PE) utilization and memory bandwidth, offering a better trade-off between compression and hardware simplicity.

2.3.2. Neuron-Level Sparsity and Selective Execution

Neuron-level pruning focuses on reducing unnecessary computation by skipping the evaluation of less important neurons. Techniques presented in [154,155] identify negative pre-activations (rendered null by ReLU) early in the pipeline or use predictors to compute only selective neuron outputs. Coupled with bit-serial processing, these techniques allow adaptive precision and reduce power consumption per operation.

Some systems integrate dual sparsity, leveraging both weight and neuron-level sparsity simultaneously for maximum reduction in workload, as demonstrated in [58,60,156].

2.3.3. Sparse Weight Compression and Memory Efficiency

Efficient coding of sparse parameters is critical for minimizing memory access overhead [79]:

- Compressed Sparse Row (CSR) and fn (CSC) formats are common. CSC, as used in EyerissV2 [59] and SCNN [58], offers better memory coalescing and access patterns than CSR [157].
- EIE [156] and SCNN [58] handle compressed dense and convolutional layers, respectively, using address-based MAC skipping and interconnection meshes for partial sum aggregation.
- Cnvlutin [158] and NullHop [159] focus on activation compression, using formats such as Compressed Image Size (CIS) and Run-Length Coding (RLC) to reduce both computation and data movement.

The UCNN accelerator [160] generalizes sparsity beyond zeros and reusing repeated weights, not only reducing memory usage but also minimizing redundant computation, which is an essential trait for edge-oriented accelerators.

2.3.4. Conclusions

Pruning, in all its forms, offers substantial gains in computation and energy savings. However, structured pruning with hardware-aware block sparsity and sparse weight compression using a CSC format provide the best balance between efficiency and deployability on edge platforms. These methods reduce memory usage and computational cost while simplifying accelerator design, unlike unstructured pruning, which poses challenges in memory access and load balancing.

For edge deployment, structured block sparsity combined with lightweight hardware accelerators—such as EyerissV2 [59] and SCNN [58]—is the most suitable approach, delivering consistent performance gains with minimal hardware complexity and excellent energy efficiency.

3. Software Tools and Hardware Platforms for Edge Inference

3.1. Software Tools

3.1.1. Compression–Compiler Co-Design

Compression-compiler co-design has recently gained momentum as a promising approach to boost EI performance on general-purpose edge devices. This method synergistically combines DL model compression with compiler-level optimization, enabling more efficient deployment by reducing model footprint and accelerating inference. By translating high-level DL constructs into hardware-efficient instructions, compilers play a pivotal role in optimizing runtime performance.

Although this co-design paradigm remained underexplored for some time, recent advancements have demonstrated its potential to significantly improve energy efficiency and reduce latency. Notable examples include PCONV [161], PatDNN [162], and CoCoPIE [163], all of which jointly explore model compression and compiler design. In the compression phase, structured pruning is used to produce hardware-friendly weight patterns. During compilation, instruction-level and thread-level parallelism are harnessed to generate optimized executable code.

MCUNet [164] exemplifies this integration by combining compact model architectures with compiler optimizations that eliminate redundant instructions and memory accesses. Similarly, CoCoPIE introduces a specialized software framework that co-optimizes pruning and quantization in tandem with compiler behavior. As shown in Figure 10, CoCoPIE introduces a custom intermediate representation (IR) that explicitly encodes compression patterns—particularly structured sparsity—during the compilation phase. This IR captures both data-level and compute-level sparsity patterns and feeds them into downstream scheduling and code generation stages.

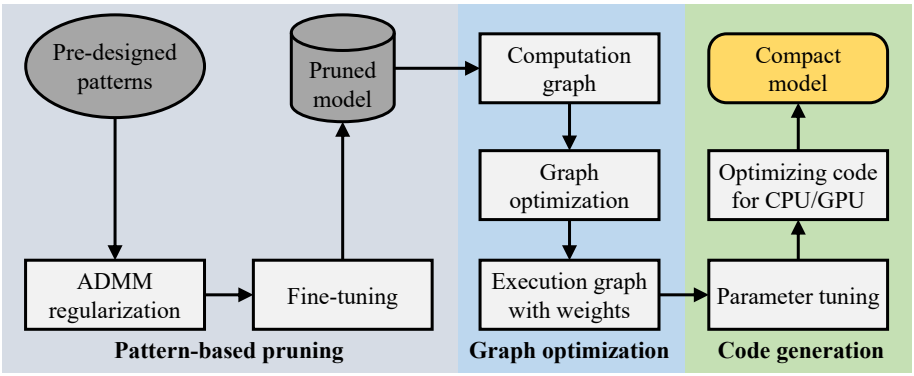

Figure 10. Illustration of compression–compiler co-design framework used in CoCoPIE [163].

CoCoPIE's IR is designed to preserve and exploit structured pruning decisions made during model compression. Each convolutional or matrix multiplication layer is associated with metadata that encodes

- Pruning masks or block patterns, such as 2D pruning windows;
- Retained kernel/channel indices for efficient memory access;
- Execution hints that guide instruction selection and tiling strategies.

During compilation, these annotations inform a custom scheduling engine that avoids loading or executing pruned weights, eliminating redundant memory fetches and arithmetic operations. The IR supports composability, allowing the compiler to treat a CNN as a sequence of reusable, pruned building blocks. This representation facilitates hierarchical compression and enables pattern-matching passes for aggressive optimization, such as fusing pruned layers or collapsing zero-weight regions during register allocation.

The CoCoPIE framework consists of two key modules:

- CoCo-Gen performs structured pruning and generates pruning-aware IR with meta-data on weight masks and instruction patterns. It emits executable code tailored to the micro-architecture of the target device.
- CoCo-Tune accelerates pruning by leveraging the composable IR to identify reusable, sparsity-inducing patterns. It performs coarse-to-fine optimization using building blocks drawn from pre-trained models, significantly reducing retraining cost.

Results reported for a commercial platform (Snapdragon 858) have demonstrated the superiority of CoCoPIE over baseline compilers such as TVM [165], TensorFlow-Lite, and MNN [166] across representative models, including VGG-16, ResNet-50, and MobileNetV2. Regarding execution time on a CPU, CoCoPIE exhibits 2.3∼8.1× speedup over TVM, 12∼44.5× over TensorFlow-Lite, and 1.9∼15.5× over MNN, respectively. On a GPU, CoCoPIE is 4.1∼14.4×, 2.5∼20×, and 2.5∼6.2× faster than TVM, TensorFlow-Lite, and MNN, respectively. In terms of energy efficiency, CoCoPIE is 6.29× more efficient than Eyeriss [167].

In summary, compression–compiler co-design is emerging as a powerful approach to achieve energy-efficient, high-performance DL inference on everyday edge devices.

3.1.2. Algorithm–Hardware Co-Design

The hardware-based execution of DNN algorithms for efficient inference continues to draw significant research interest. This growing focus on customized hardware stems from two main factors: (i) the performance of conventional hardware is approaching its limits, and (ii) integrating compression techniques into hardware offers a promising path to handle the rising computational demands. At the core of this shift lies the concept of algorithm–hardware co-design. A general overview of this co-design approach is depicted in Figure 11.

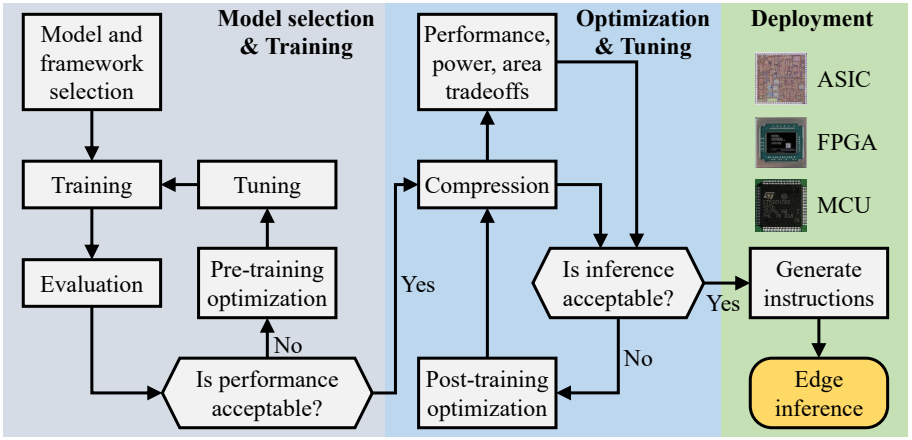

Figure 11. An overview of the algorithm–hardware co-design approach. Model selection and training are performed using any machine learning framework. The trained model is subsequently refined through an edge-optimized intermediate software layer before being deployed on the designated edge device.

The process begins with developing the algorithm within standard ML frameworks. Once optimal models are trained, an edge-specific intermediate software layer optimizes them for deployment. These optimized models are then executed on the target hardware to accelerate inference. Algorithm-level optimizations guide the hardware design, while hardware implementation provides valuable feedback to refine algorithm development further. This bidirectional interaction defines algorithm–hardware co-design as a promising direction for creating application-specific DNN accelerators tailored for edge environments.

3.1.3. Efficient Parallelism

Parallelization techniques are key to accelerating DL inference by efficiently utilizing underlying hardware resources. This strategy distributes workloads across multiple processing units, improving both latency and throughput. CNNs and RNNs, due to their inherently repetitive computations, particularly benefit from such parallel execution on platforms such as CPUs, GPUs, and DSPs [168].

For example, CNNdroid [169], a GPU-based deep CNN inference engine tailored for Android devices, delivers up to 60× speedup and 130× energy savings, highlighting the performance gains of mobile GPU acceleration. However, GPU-based parallelization, while effective, poses challenges in energy-limited environments and often demands complex hardware-specific tuning, especially when deploying on heterogeneous systems with tight thermal budgets.

CUDA, NVIDIA's parallel computing platform, enables fine-grained control over GPU threads and memory hierarchy, making it ideal for performance-critical applications such as real-time object detection and robotic navigation [170]. However, applying CUDA directly to mobile GPUs often results in suboptimal efficiency due to architectural limitations and limited thermal headroom [171]. As a more portable alternative, RenderScript [172]—a high-level framework for Android—automatically maps data-parallel tasks to available GPU cores. Using this abstraction, SqueezeNet achieves over 310× speedup on a Nexus 5 smartphone, illustrating the software simplicity and deployment ease of RenderScript at the cost of reduced hardware-level optimization flexibility compared to CUDA.

In contrast, Cappuccino [173] focuses on automated model deployment for EI. It breaks down neural operations across kernels, filters, and output channels, promoting parallel computation with minimal manual intervention. While this approach enhances developer productivity and ensures reasonable energy efficiency, it may introduce slight numerical deviations due to approximation during parallel execution.

DeepSense [174] represents another GPU-accelerated inference solution designed for mobile environments, assigning heavy computation layers to the GPU while the CPU handles lighter tasks such as padding and memory operations. However, mobile GPUs' limited memory bandwidth and storage often restrict support for larger CNN architectures [175]. A practical workaround, demonstrated using YOLO on the Jetson TK1, involves splitting workloads between CPU and GPU. This hybrid setup can still yield up to 60× inference acceleration, though it requires careful synchronization and resource management [176].

To further reduce computational cost, DeepX [177] applies singular value decomposition to compress weight matrices and shrink neural layers. This not only reduces model size but also facilitates parallel processing of compressed layers, improving both execution speed and energy efficiency. The DeepX Toolkit (DXTK) [178] generalizes this concept, enabling broad compatibility across model types and making it a robust solution for scalable, resource-limited edge platforms.

Among the evaluated methods, DeepX and DXTK stand out as the most edge-friendly solutions due to their balanced design: they offer significant energy savings, require minimal manual tuning, and support parallel inference across compact networks. Their compatibility with compressed models further makes them well-suited for battery-powered and thermally constrained devices. In contrast, while CUDA-based implementations offer granular performance control, their higher hardware and development complexity make them less ideal for general-purpose edge deployment without dedicated optimization efforts.

3.1.4. Memory Optimization

Deep learning inference relies heavily on computation-intensive MAC operation, each necessitating three memory reads (weights, input activations, and partial sums) and one

write to update results. This frequent memory access becomes a critical performance bottleneck, particularly for edge devices where energy efficiency and low latency are paramount. To address this challenge, most DNN accelerators adopt a hierarchical memory architecture (illustrated in Figure 12), consisting of

- Off-chip DRAM, which stores the bulk of model parameters and intermediate data;
- On-chip SRAM-based global buffers (GLBs), which cache activations and weights temporarily;
- Register files (RFs) embedded within each PE, providing the fastest and lowest-power memory access.

While DRAM offers large capacity, it is the most energy-intensive component. In contrast, on-chip memories provide faster access at significantly lower power, but their capacity is limited. As such, optimizing memory access patterns to minimize DRAM usage and maximize on-chip reuse is essential for EI.

A. Dataflow Strategies for Memory Reuse: Efficient data reuse strategies can drastically reduce memory traffic and energy consumption. Four common dataflow schemes, as summarized in Table 6, are widely employed in DL accelerators:

- Weight Stationary (WS): This strategy keeps weights fixed in RFs while streaming activations across PEs, significantly reducing weight-fetch energy. WS is well-suited for convolutional layers with high weight reuse. For example, NeuFlow [179] applies WS for efficient vision processing. However, frequent movement of activations and partial sums may still incur moderate energy costs.
- No Local Reuse (NLR): In this scheme, neither inputs nor weights are retained in local RFs; instead, all data are fetched from GLBs. While this simplifies hardware design and supports flexible scheduling, the high frequency of memory access limits its energy efficiency. DianNao [180] follows this approach but mitigates energy consumption by reusing partial sums in local registers.
- Output Stationary (OS): OS retains partial sums within PEs, allowing output activations to accumulate locally. This reduces external data movement and is particularly beneficial for CNNs. ShiDianNao [181] adopts this strategy and embeds its accelerator directly into image sensors, eliminating DRAM access and achieving up to 60× energy savings over DianNao [180]. OS is especially well-suited for vision-centric edge devices requiring real-time inference.
- Row Stationary (RS): RS maximizes reuse at all levels—inputs, weights, and partial sums—within RFs. It loops data within PEs, significantly minimizing memory traffic. MAERI [182] is a representative architecture that implements RS using flexible interconnects and tree-based data routing. Despite increased hardware complexity, RS offers excellent performance-to-energy trade-offs and is highly scalable across DL models.

Table 6. Summary of four dataflow strategies.

Strategy	Memory Access Frequency	Energy Efficiency	Hardware Design Complexity	Edge Suitability
Weight stationary	Moderate	Good	Low	Moderate
No local reuse	High	Poor	Low	Limited
Output stationary	Low	Very good	Moderate	High
Row stationary	Very low	Excellent	High	Very high

B. Reducing Memory Load Through Compression and Sparsity: Beyond dataflow optimizations, compression and sparsity techniques further alleviate memory bottlenecks:

- Run-Length Coding (RLC): This method compresses consecutive zeros, reducing external bandwidth by up to 1.5× for both weights and activations [167].

- Zero-Skipping: This approach skips MAC operations involving zero values, reducing unnecessary memory access and saving up to 45% of energy [167].
- Weight Reuse Beyond Zeros: This scheme generalizes sparsity by recognizing and reusing repeated weights, not limited to zeros, further lowering memory access and computation requirements [160].

These strategies reduce both the volume and frequency of memory access, allowing edge accelerators to achieve greater energy efficiency without compromising performance.

C. Suitability for Edge Deployment: Considering both memory usage and access patterns, RS and OS strategies are the most suitable for edge deployment:

- RS achieves the best overall efficiency by minimizing access at all hierarchy levels, though it introduces higher design complexity. It is preferred in scenarios where flexibility and scalability are needed.
- OS offers a more straightforward design and strong efficiency in vision-centric models, making it highly appropriate for lightweight, task-specific edge devices.

When coupled with compression methods such as RLC and zero-skipping, these dataflows become even more advantageous, enabling ultra-low-power, high-throughput edge AI deployments.

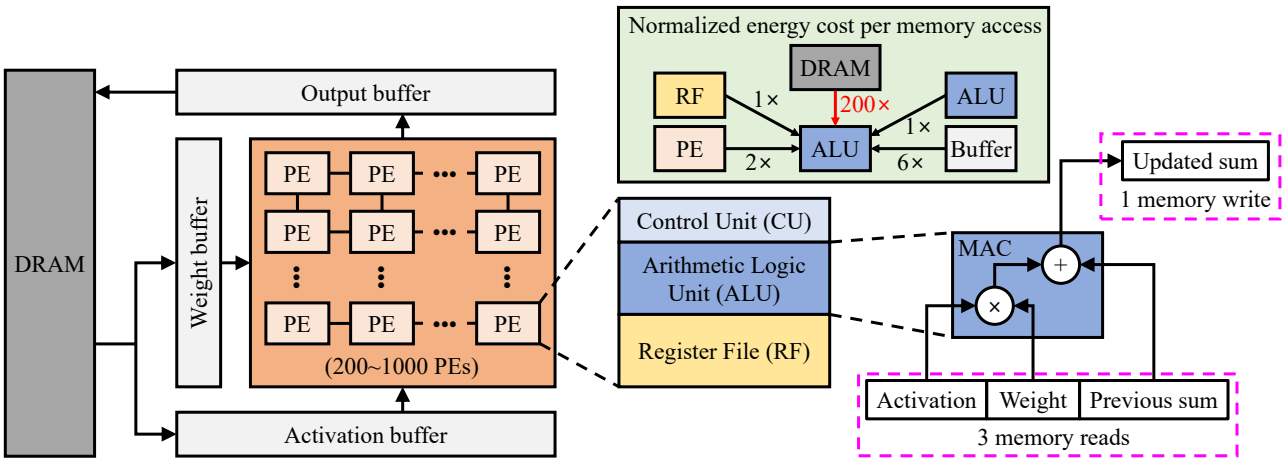

Figure 12. An illustration of a typical DNN accelerator. At the heart of the ALU lies the MAC unit, which typically requires up to four memory accesses per operation. The normalized energy cost associated with each type of memory access is also illustrated.

3.1.5. FPGA-Centric Deployment via Deep Learning HDL Toolbox

The Deep Learning HDL Toolbox by MathWorks offers a comprehensive framework for deploying DL models on Xilinx and Intel FPGA/SoC platforms, making it a practical solution for EI [183]. The toolbox enables users to develop DL models natively in MATLAB (currently supported from version R2021a onward) or import pre-trained models from external sources, such as Keras and ONNX, with seamless integration into a hardware deployment workflow.

A distinguishing feature of the toolbox is its automated hardware compilation flow, where users can generate synthesizable Verilog or VHDL using HDL Coder and Simulink, targeting custom edge hardware without writing low-level HDL code manually. This high level of abstraction dramatically simplifies the hardware design process, making it well-suited for engineers with limited hardware experience. Deployment can be conducted via standard Ethernet or JTAG interfaces, allowing real-time model instruction loading and performance monitoring.

To optimize performance on resource-limited edge devices, the toolbox supports 8-bit integer quantization, significantly reducing memory footprint and improving com-

putational throughput. Quantization minimizes the demand on off-chip memory (for example, DDR), enabling better use of on-chip block RAM and reducing overall energy consumption—a key consideration in edge AI scenarios.

Figure 13 presents the architecture of the MATLAB-generated HDL processor IP core, which is platform-agnostic and scalable. The core integrates

- AXI4 master interfaces for DDR communication;
- Dedicated convolution and fully connected (FC) processors;
- Activation normalization modules supporting ReLU, max-pooling, and normalization functions;
- Controllers and scheduling logic to coordinate dataflow and computation.

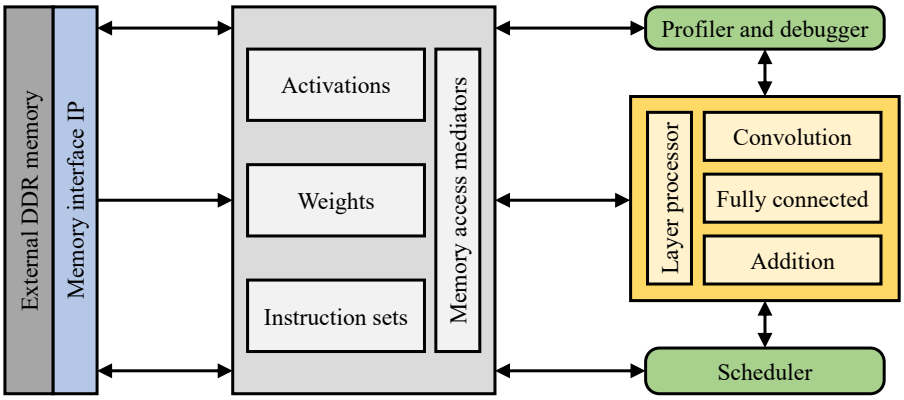

Figure 13. An overview of the MATLAB HDL DL Processor IP that features four AXI4 master interfaces for data transfer, external DDR memory, a dedicated layer processing unit, and a scheduling module.

Input activations and weights are transferred from DDR to block RAM via AXI4 and then processed by the convolution or FC units. Intermediate results remain on-chip, reducing redundant off-chip memory access, which is costly in terms of both latency and energy. The IP core also supports output feedback to MATLAB, facilitating layerwise latency profiling and bottleneck analysis.

Compared to conventional SoC or GPU-based solutions, the toolbox's FPGA-based deployment achieves a more balanced trade-off between performance and energy efficiency, particularly in scenarios requiring deterministic latency and low-power operation—making it highly suitable for edge AI applications such as embedded vision, robotics, and autonomous sensing.

Moreover, the toolbox includes built-in hardware support for essential DL layers, including 2D and depthwise convolutions, FC layers, various activation functions, normalization, dropout, and pooling. It supports the deployment of popular models such as AlexNet, VGG, DarkNet, YOLOv2, MobileNetV2, and GoogLeNet to FPGA development boards such as Xilinx Zynq-7000 ZC706 FPGA, Zynq UltraScale+ MPSoC ZCU102, and Intel Arria 10 SoC. This wide compatibility makes the toolbox ideal for rapid prototyping and deployment of DL models on energy- and resource-constrained edge platforms, combining customizability, efficiency, and portability in a single development flow.

3.1.6. AMD Vitis AI

AMD Vitis AI is an integrated development framework engineered to streamline and accelerate DL inference on AMD's adaptable computing platforms [184]. It encompasses a full toolchain—ranging from IP cores and development libraries to pre-trained models and compiler toolkits—designed to simplify and optimize AI deployment. The

core inference engine in this environment is the Deep Learning Processing Unit (DPU), a reconfigurable hardware block tailored for executing popular DL models such as ResNet, YOLO, GoogLeNet, SSD, MobileNet, and VGG.

The DPU is supported across a broad spectrum of AMD hardware platforms, including Zynq UltraScale+ MPSoCs, the Kria KV260 Vision AI Starter Kit, Versal adaptive SoCs, and Alveo accelerator cards. For embedded deployments, Vitis AI enables DPU integration via both the Vitis unified software platform and the Vivado hardware design suite. Figure 14 illustrates the complete Vitis AI toolchain, highlighting the DPU, compiler, runtime, and profiling components tailored for edge deployment. Notably, developers can utilize prebuilt board support packages (BSPs) for rapid deployment without needing in-depth hardware design skills, making Vitis AI an accessible yet powerful solution for AI at the edge.

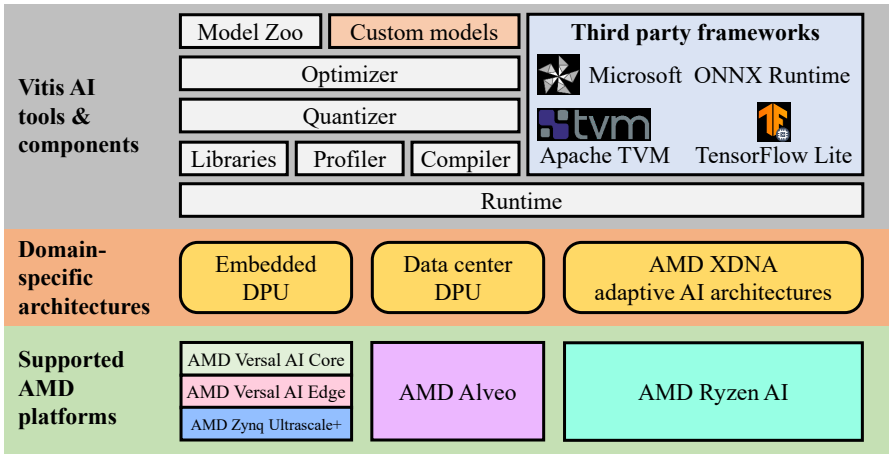

Figure 14. Overview of AMD Vitis AI. It includes tools, libraries, models, and optimized IPs for accelerating DL inference on edge platforms.

From a memory and energy constraint, Vitis AI is well-optimized for low-power inference. The DPU architecture is designed to maximize data locality, significantly reducing the frequency of off-chip DRAM access—one of the primary contributors to power consumption. Quantization plays a key role in this: the Vitis AI Quantizer converts floating-point models to INT8, which not only lowers memory usage but also improves energy efficiency and inference speed with negligible accuracy loss. Compared to standard floating-point models, INT8 quantization can reduce memory footprint by over 4× and inference latency by up to 3×, making it especially well-suited for resource-limited edge devices.

Model development is further streamlined through an ecosystem of tools. The Model Zoo offers pre-trained networks for diverse edge applications such as surveillance, robotics, and ADAS. The Model Inspector ensures compatibility with DPU-supported layers, while the Optimizer employs structural pruning techniques to minimize redundant computations, enhancing both performance and energy efficiency.

Once optimized and quantized, models are compiled using the Vitis AI Compiler, which maps them to the DPU's native instruction set. This stage performs advanced graph-level optimizations, improving data reuse and enabling parallel execution across the DPU's computing elements.

Deployment is facilitated through the Vitis AI Runtime (VART), a lightweight runtime environment supporting both C++ and Python APIs (Python version 3). It allows asynchronous and efficient DPU control within embedded Linux or cloud-hosted applications. Building on VART, the Vitis AI Library provides high-level APIs and pre-/post-processing functions to reduced development time.

To fine-tune performance, the Vitis AI Profiler enables detailed analysis of inference bottlenecks across CPU, memory, and DPU domains—offering visibility into system-wide behavior without requiring source code changes.

Compared to general-purpose AI toolchains, Vitis AI offers a tightly coupled hardware–software co-design that excels in both performance-per-watt and flexibility. The ability to perform quantized inference using dedicated DPU hardware results in a highly energy-efficient solution—far surpassing CPU-based implementations and often outperforming GPU-based alternatives in power-sensitive environments. In contrast to platforms such as NVIDIA's Jetson series, which rely on fixed hardware accelerators, AMD's programmable fabric allows developers to tailor DPU configurations to match specific model workloads, offering an edge in customization and scalability.

3.1.7. NVIDIA TensorRT

NVIDIA TensorRT is a highly optimized DL inference engine tailored for high-throughput, low-latency deployment on embedded and automotive platforms [61]. Designed atop the CUDA parallel computing architecture, TensorRT converts pre-trained DL models into compact, high-speed runtime engines, making it well-suited for power- and performance-constrained edge environments. Leveraging NVIDIA's specialized sparse tensor cores, TensorRT applies a suite of six key optimizations to maximize execution efficiency:

- Mixed-precision quantization, which transforms FP32 weights and activations to FP16 or INT8, reducing computation time and memory usage.
- Layer and tensor fusion, which merges multiple operations into a single CUDA kernel to cut memory access overhead.
- Kernel auto-tuning, which selects optimal algorithms and batch sizes tailored to the hardware's capabilities.
- Dynamic memory allocation, ensuring tensors occupy memory only when needed, minimizing memory usage.
- Multi-stream execution, enabling concurrent inference across multiple data streams for improved throughput.
- Time-step fusion for RNNs, which collapses sequential operations over time into fused kernels to enhance execution parallelism.

Compared to traditional CPU inference, TensorRT can deliver speedups of up to 40×, with significantly lower latency and memory footprint. Its quantization pipeline—particularly KL-divergence-guided INT8 calibration—preserves accuracy while achieving notable compression, making it ideal for real-time AI on devices such as the NVIDIA Jetson series.

In terms of memory efficiency, TensorRT excels through fusion and dynamic management, substantially reducing memory access frequency and temporary buffer usage. This makes it competitive against other edge-oriented toolkits such as AMD Vitis AI [184] and MATLAB HDL Toolbox [183]. Unlike toolflows requiring hardware synthesis or FPGA IP integration, TensorRT offers immediate deployment on CUDA-enabled edge GPUs without requiring hardware design expertise, improving development agility.

Therefore, TensorRT is particularly well-suited for edge AI applications demanding high inference speed, moderate-to-low memory usage, and plug-and-play deployment on embedded GPU platforms. Its primary limitation lies in its reliance on NVIDIA GPUs, whereas alternatives such as Vitis AI or MATLAB HDL Toolbox support FPGAs and SoCs, potentially offering better energy efficiency in ultra-low-power settings. However, for vision, robotics, or autonomous driving applications with available CUDA hardware, TensorRT remains a top-tier choice.

3.1.8. Qualcomm Neural Processing SDK

The Qualcomm Neural Processing SDK (NPS) is a comprehensive toolkit that facilitates the deployment of DL models on Snapdragon-based mobile platforms [185]. Compatible with models trained in popular frameworks such as TensorFlow, Caffe, and ONNX, the SDK enables efficient, low-latency inference by exploiting Snapdragon's heterogeneous architecture, which integrates Kryo CPUs, Adreno GPUs, and Hexagon DSPs. This flexibility allows developers to offload specific tasks to the most appropriate processing unit, maximizing performance without relying on cloud infrastructure.

As shown in Figure 15, the development pipeline begins with model training and export from a supported framework. The trained model is then converted into a Deep Learning Container (.dlc) file, which is optimized for execution on the Snapdragon Neural Processing Engines (NPEs). Additional optimizations such as quantization and compression can be applied at this stage to reduce memory usage and boost execution speed, further tailoring the model for edge operation.

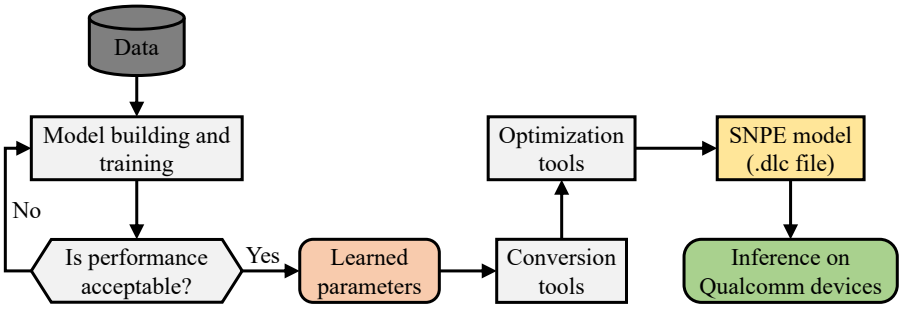

Figure 15. Workflow overview of Qualcomm Neural Processing SDK.

The SDK provides multiple interfaces, including C++/Java APIs and GStreamer plugins, enabling developers to integrate these optimized models directly into Android applications. After deployment, the NPS toolkit offers profiling and debugging tools to analyze latency, memory usage, and performance bottlenecks, facilitating iterative refinement and accuracy tuning.

From a comparative standpoint, Qualcomm NPS stands out in energy efficiency and native integration with Android devices, making it ideal for smartphones, wearables, and embedded IoT systems. Unlike platforms such as NVIDIA TensorRT [61]—which require CUDA-capable GPUs—or MATLAB HDL Toolbox [183],which targets FPGA-based implementations, NPS is optimized for commercial off-the-shelf Snapdragon SoCs, removing hardware design barriers and offering seamless software-level integration.

Furthermore, compared to AMD Vitis AI [184], which targets more power-hungry FPGA and SoC platforms, the NPS SDK delivers a lighter-weight deployment path tailored to mobile edge devices, where thermal and energy constraints are critical. Its support for custom layers also allows for specialized DL applications beyond standard vision models.

In conclusion, the Qualcomm NPS is particularly well-suited for edge AI deployments on mobile platforms, offering an efficient, scalable, and developer-friendly solution for executing DL inference entirely on-device—balancing computational power, memory usage, and power efficiency.

3.1.9. Cadence Stratus HLS

Stratus HLS, developed by Cadence Design Systems Inc., is a high-level synthesis (HLS) tool that translates DL models into register-transfer level (RTL) hardware implementations [186]. Designed for performance-driven hardware development, Stratus HLS

allows engineers to analyze critical metrics such as power consumption, silicon area, and inference throughput, making it a powerful option for designing DL accelerators with precise hardware constraints.

As illustrated in Figure 16, the development pipeline starts with training the DL model in conventional frameworks such as TensorFlow or Caffe. After training, the learned parameters—weights and biases—are exported and reused for RTL-based deployment. The model architecture itself must be manually reimplemented in SystemC, where developers can define parameterized data types, control numerical precision, and specify hardware-specific constraints. This SystemC representation forms the basis for HLS compilation.

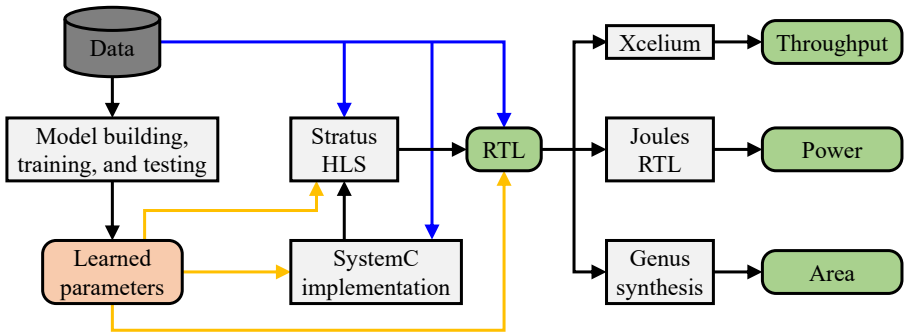

Figure 16. Workflow overview of Cadence Stratus HLS for RTL synthesis and inference.

Stratus HLS supports fixed-point quantization, typically ranging from 16-bit down to 12-bit, allowing for reduced memory usage and computational complexity—both crucial for edge applications. Once the SystemC model is verified, Status HLS compiles it into synthesizable Verilog RTL code. This RTL design can then be evaluated using other Cadence tools: Joules RTL Power Solution for energy profiling, Genus Synthesis Solution for estimating chip area, and Xcelium Simulator for simulating functional performance.

From a comparative standpoint, Status HLS offers a high degree of design flexibility and fine-grained hardware control, distinguishing it from automated deployment tools such as TensorRT [61], Vitis AI [184], or MATLAB HDL Toolbox [183]. However, this flexibility comes at the cost of a steeper learning curve and longer development times, as manual hardware modeling in SystemC is required.

In terms of edge deployment, Stratus HLS is particularly well-suited for custom ASIC or low-power FPGA implementations, where power efficiency, chip footprint, and deterministic latency are top priorities. Unlike platform-dependent SDKs such as Qualcomm NPS [185] or NVIDIA TensorRT—designed for existing SoC platforms—Stratus HLS empowers designers to build tailor-made inference engines optimized for their specific edge use cases. This makes it ideal for mission-critical or resource-constrained environments, such as wearable devices, industrial sensors, or ultra-low-power embedded systems.

In summary, Stratus HLS provides a robust path for deploying DL models in highly customized edge hardware, enabling precise optimization of silicon, power, and performance. While it requires more hardware expertise than other toolchains, it offers unparalleled control for engineers targeting high-efficiency edge AI solutions.

3.2. Hardware Platforms

3.2.1. CPU and GPU

General-purpose CPUs possess billions of transistors and are architected to handle a broad range of computational tasks, including DL training and inference. Their powerful cores and expansive memory systems enable them to execute complex workloads with high flexibility. However, despite their versatility, CPUs are often suboptimal for EI due to several drawbacks: elevated power consumption, limited parallelism, low computational

density, and inefficiencies in silicon utilization. The large chip area—partly due to control-intensive components and general-purpose logic—further contributes to higher fabrication costs and thermal overhead. Moreover, CPUs are inherently sequential machines and struggle to exploit the massive parallelism characteristic of DL workloads, which reduces their overall efficiency in low-latency inference tasks.

GPUs, by contrast, are optimized for data-parallel operations and have become a cornerstone of high-performance computing (HPC) and DL acceleration. Featuring hundreds to thousands of cores, GPUs are designed to process large volumes of simple arithmetic operations in parallel, making them far more effective than CPUs for executing CNNs and matrix multiplications. Like CPUs, GPUs are programmable, supporting major DL frameworks through tools such as CUDA, cuDNN, and PyTorch (all versions). However, this performance comes at a cost: high energy consumption, substantial thermal output, and data transfer latency—especially when interfacing with off-chip memory or external peripherals.

In the context of edge AI, these characteristics impose critical limitations. Power-constrained devices—such as drones, mobile devices, wearables, or IoT sensors—cannot afford the power and thermal budget that GPUs typically require. Although modern embedded GPU solutions attempt to address these limitations, their deployment feasibility depends heavily on use-case demands and device constraints.

For example, NVIDIA's Jetson TX2 integrates an embedded GPU capable of running DL inference tasks at approximately 15 W, balancing performance and power for intermediate edge applications such as smart cameras or mobile robots [187]. A more constrained alternative is the Jetson Nano, which combines a 128-core Maxwell GPU and a quad-core ARM Cortex-A57 CPU to deliver DL capabilities within a 5∼10 W power envelope [188]. The JetPack SDK [189] provides a software stack supporting major DL frameworks, enabling developers to prototype and deploy inference workloads with relative ease.

In conclusion, while both CPUs and GPUs can perform DL inference, neither is inherently ideal for ultra-constrained edge applications. CPUs are constrained by sequential execution and power inefficiency, while GPUs, though parallel, are still energy-intensive. For power- and cost-sensitive edge environments, dedicated hardware accelerators (such as NPUs, DSPs, or custom ASICs) typically offer better performance-per-watt. However, CPUs and GPUs remain highly valuable for rapid development, prototyping, and edge applications where flexibility and software compatibility are prioritized over energy efficiency.

3.2.2. SoC and ASIC

Researchers are increasingly adopting ASIC and SoC solutions for DL tasks, driven by their unmatched performance and energy efficiency. A prominent example is ShiDianNao [181], a custom ASIC designed to optimize memory access patterns, thereby minimizing latency and reducing power usage. It belongs to the DianNao family [180], a series of DL accelerators tailored for embedded environments. Another advanced design, UNPU [80], is implemented using a 65 nm CMOS technology and supports inference across convolutional, recurrent, and fully connected layers. UNPU enables flexible weight precision ranging from 1 to 16 bits and adopts an LUT serial processing scheme to maximize energy efficiency during MAC operations. Depending on the precision used, UNPU delivers peak performance from 345.6 to 7372 GOPS, achieving up to 3080 GOPS per Watt (GOPS/W) in energy efficiency.

Table 7 provides a comparative summary of representative ASICs and SoCs for EI. Among them, Envision [118], built on a 28 nm CMOS process, targets always-on vision tasks in wearable devices. It repurposes inactive arithmetic units for lower-precision computations, delivering energy efficiency of up to 10,000 GOPS/W—one of the highest

among surveyed designs. Similarly, SNAP [151], fabricated using a 16 nm process, exploits unstructured sparsity to optimize convolutional workloads. Despite its compact 2.4 mm^2 area, SNAP achieves an impressive energy efficiency of 21,550 GOPS/W and maintains low power operation at 348 mW when running pruned ResNet-50 inference.

Table 7. Overview of notable ASIC and SoC solutions designed for edge inference. NA stands for not available.

ASIC	Process (nm)	Area (mm^2)	Performance (GOPS)	Power Consumption (mW)	Normalized Power Consumption (mW) [a]	Energy Efficiency (GOPS/W)	Supported DL Models	Target Applications
DNA 100 [190]	16	NA	5780	850	NA	6800	All neural network layers	IoT, AR/VR, autonomous driving, surveillance, mobile devices
Stripes [74]	65	122.1	168,768.6 [b]	17,886.4 [b]	3.6	9435.6	CNN	Image classification
Eyeriss [167]	65	12.3	29.7∼42	255∼332	2685.4∼2916.8	126.5	CNN	Image analysis
Bit Fusion [75]	16	5.9	318.9 [c]	895	1969.8	335.3	CNN, RNN	Image classification, object detection, language modeling, optical character recognition
DesignWare [191]	16	NA	512	NA	NA	NA	CNN, R-CNN, YOLO	Object detection, image classification, segmentation
UNPU [80]	65	16	9.9∼914.8	3.2∼297	84.4	3080	CNN, RNN, FNN	Image classification
Envision [118]	28	1.9	2∼3000	7.5∼300	222.5∼8559.4	260∼10,000	CNN	Always-on vision, face recognition
PDFA [192]	65	5.8	129.4∼221.8	168	547.3∼938.3	770∼1320	CNN, FCN	Object tracking
SNAP [151]	16	2.4	351.3∼7844.2	16.3∼364	80.5	21,550	CNN, FCN	Image classification
Cambricon-X [149]	65	6.4	544	954	1143.9	570.2	CNN	Image classification
Scalable processor [68]	40	2.4	7.5∼748.8	25∼288	666.9∼5780	300∼2600	CNN	Image classification
Sticker [193]	65	7.8	8.2∼15,425.6	20.5∼248.4	8.6∼1333.8	400∼62,100	CNN, FCN	Object detection
EIE [156] [d]	45	40.8	102	600	600	170	CNN, RNN, LSTM	Image classification, object detection, automatic image captioning
FlexFlow [194]	65	3.9	380∼500	1000	2139.6∼2815.3	380∼500	CNN	Image analysis
SqueezeFlow [195]	65	4.8	652.8∼761.6 [e]	536.1	610.3∼712	1217.7∼1420.7	Sparse CNN	Image analysis

[a] Power consumption values are normalized with respect to both area and performance. [b] These values are derived based on Stripes's performance relative to DaDianNao [196]. [c] This value is derived based on Bit Fusion's performance relative to NVIDIA Titan Xp. [d] EIE is used as a reference for normalizing power consumption. [e] These values are derived based on SqueezeFlow's performance relative to Cambricon-X.

Sticker [193] offers another high-efficiency design, delivering up to 62,000 GOPS/W in a 7.7 mm^2 footprint. It incorporates a reconfigurable circuit that dynamically adapts across nine sparsity and memory access modes to maximize energy utilization. Other compact ASICs, such as FlexFlow [194] and SqueezeFlow [195], provide energy efficiency ranging from 380 to 1420.7 GOPS/W by supporting flexible dataflows and exploiting sparsity-aware execution, respectively. Bit Fusion [75], though implemented in 16 nm technology, achieves 335.3 GOPS/W while demonstrating competitive normalized power consumption at 1969.8 mW.

On the other end of the spectrum, devices such as Eyeriss [167] and Cambricon-X [149], despite offering modest energy efficiency (126.5 and 570.2 GOPS/W, respectively), are valuable for benchmarking due to their detailed open-source performance data. EIE [156], optimized for sparse matrix-vector multiplication, is used as a normalization reference in this study, offering 170 GOPS/W under a 600 mW power budget.

In terms of normalized power consumption—which accounts for both area and throughput—Stripes [74] shows excellent scaling with a normalized power value of 3.6 mW, making it one of the most power-efficient designs relative to its performance. Similarly, UNPU [80] and Sticker [193] also exhibit low normalized power consumption (84.4 and 8.6∼1333.8 mW, respectively), reflecting their optimization for energy-aware workloads.

Major commercial efforts include Cadence DNA 100 [190], a 16 nm IP core designed for scalable inference acceleration. Supporting frameworks such as TensorFlow and Caffe, DNA 100 enables deployment across a wide range of applications—from wearables and AR/VR to autonomous systems—while delivering up to 6800 GOPS/W. Likewise, Synopsys DesignWare EV7x [191], a 16 nm FinFET-based IP core, integrates a CNN accelerator and 512-bit vector DSP, achieving up to 512 GOPS in performance. Ceva's NeuPro-P [197], another scalable IP solution, provides 4000∼200,000 GOPS per core, supporting various quantization levels and targeting applications such as ADAS and smart cameras.

Google's Coral Edge TPU [198] focuses on low-power inference with high-speed execution and supports lightweight transfer learning. Compatible with TensorFlow-Lite and models such as YOLO and R-CNN, the Edge TPU enables efficient offline processing and is ideal for EI applications.

Among all hardware options, ASICs clearly offer the highest energy efficiency for edge inference, as demonstrated by the superior GOPS/W figures in Table 7. However, ASICs typically hardwire computations, limiting their adaptability to evolving DL models and algorithms. This lack of flexibility can lead to higher development costs and reduced scalability. Attempts to enhance flexibility often result in increased chip area and power consumption. Therefore, both coarse- and fine-grained reconfigurable architectures continue to gain attention as alternative platforms for rapid prototyping and deployment in EI systems.

3.2.3. FPGA and Reconfigurable Architectures

The spatial organization of hardware accelerators for DL models generally falls into two main categories: fine-grained and coarse-grained architectures. FPGAs are examples of fine-grained reconfigurable platforms, where logic elements are programmed at a low level using hardware description languages (HDLs). In contrast, Coarse-Grained Reconfigurable Architectures (CGRAs) have emerged as a promising alternative for edge applications. CGRAs offer several advantages over traditional fine-grained designs, including higher clock speeds, faster computational throughput, and reduced compilation and reconfiguration times. These features make CGRAs particularly effective for the kind of tensor-heavy operations found in DL inference. Their architecture strikes a balance between flexibility and efficiency, positioning them as an attractive solution for high-performance yet adaptable edge computing deployments.

A. FPGA: FPGAs provide a versatile and reconfigurable hardware platform that offers faster deployment compared to ASICs, making them well-suited for evolving DL model requirements. Comprising programmable logic blocks interconnected via configurable routing resources, FPGAs enable customized hardware implementations using HDLs such as Verilog and VHDL. Their reprogrammability and support for rapid prototyping have made them increasingly attractive for DL inference, especially in edge applications where power efficiency and flexibility are crucial.

The growing need for efficient edge AI solutions has driven the development of compact, high-performance FPGAs. Devices such as the Xilinx Zynq-7000 series, particularly the ZYNQ-7020, are widely adopted for implementing CNN inference. Notably, the 8-bit fixed-point version of the VGG-16 model achieved up to 84 GOPS on this platform [199], while the 16-bit version reached 13 GOPS [200]. Similarly, Lite-CNN [201], a lightweight INT8-quantized CNN deployed on the ZYNQ XC7Z020, has demonstrated peak performance of 408 GOPS with energy efficiency of 33 GOPS/W, highlighting the potential of FPGAs for high-throughput, low-power edge inference.

Despite these strengths, FPGA-based DL accelerators face limitations. Mapping trained models onto FPGA hardware requires additional design effort and toolchain fa-

miliarity. Moreover, the compilation and synthesis processes can be time-consuming, and reconfiguration is relatively slow compared to loading models onto CPUs or GPUs. FPGAs also typically operate at lower clock frequencies than ASICs or GPUs, which can limit performance in certain use cases.

Nevertheless, for edge deployment, FPGAs offer a compelling trade-off between performance, energy efficiency, and adaptability. Their reusability across different models and support for reduced-precision inference make them particularly advantageous for real-time, resource-constrained environments such as IoT devices and autonomous systems. Table 8 summarizes key FPGA-based DL implementations, comparing model types, throughput, and energy efficiency to provide insight into their practical suitability for edge AI solutions.

Table 8. Summary of representative FPGA-based implementations for DL inference, sorted by energy efficiency.

FPGA Platform	DL Model	Peak Performance (GOPS)	Energy Efficiency (GOPS/W)
Xilinx Virtex-7	C-LSTM [202]	131.1	6.0
Zynq ZU3EG	Synetgy [94]	47.1	8.6
Arria 10GX1150	BBS-LSTM [39]	304.1	15.9
Xilinx XC7Z100	DeltaRNN [203]	192.0	26.3
Xilinx ZC702	XNOR-Net [77]	207.8	44.2
Xilinx ZC702	DoReFa-Net [86]	410.2	181.5

B. Reconfigurable Architectures: DL inference accelerators are typically composed of a structured array of PEs, RFs, and either shared or scratchpad memory. These components work together to maximize performance by enabling efficient dataflow, low-latency computation, and high energy efficiency. The architectural design—especially when optimized for data reuse, memory locality, and compute parallelism—can significantly reduce the dependency on power-hungry off-chip memory access, which is crucial for real-time, edge-based inference.

In particular, CGRAs have emerged as a favorable alternative to general-purpose CPUs and GPUs for DL workloads, especially at the edge [167,204,205]. Unlike traditional processors that rely heavily on software-level parallelism and external memory bandwidth, CGRAs incorporate tightly coupled PEs with local memory, which enhances temporal and spatial reuse. This results in improved throughput and lower energy consumption—key metrics for deploying inference on power- and thermal-constrained edge devices. Table 9 highlights several state-of-the-art reconfigurable architectures, comparing supported model types, peak throughput, and energy efficiency.

Table 9. Overview of notable reconfigurable architectures for DL inference, sorted by energy efficiency. NA stands for not available.

Reconfigurable Architecture	Supported DL Model	Peak Performance (GOPS)	Energy Efficiency (GOPS/W)
Lite-CNN [201]	CNN	363	33
SDT-CGRA [206]	CNN, SVM, etc.	92.3	50.8
Eyeriss [167]	CNN	46.2	166
EyerissV2 [59]	CNN	153.6	962.9
Thinker chip [207]	CNN, RNN	409.6	1060
NullHop [159]	CNN	450	3000
DNPU [208]	CNN, RNN	1214	8100
DRP [209]	CNN, DNN	960	NA

Eyeriss [167], for example, is a pioneering CGRA tailored for convolutional workloads. It integrates 168 PEs interconnected via a network-on-chip (NoC) and employs data sparsification and compression to reduce off-chip memory access. With 16-bit fixed-point quantization, it achieves 166 GOPS/W at just 278 mW while running AlexNet, making it an excellent candidate for mobile and embedded platforms. Its successor, EyerissV2 [59], adopts a hierarchical buffer and PE structure, further reducing interconnect overhead and improving scalability for larger DL models—an advantage in real-time inference at the edge.

Thinker [207], another CGRA chip, provides three levels of reconfigurability, including variable-precision multipliers (8-bit/16-bit), support for zero-skipping, and dynamic on-chip memory control. These features allow it to sustain high utilization rates of its PEs and achieve up to 590 GOPS/W at 335 mW, making it highly suitable for edge devices with tight power envelopes.

SCNN [58] goes a step further by introducing sparsity-aware design in both weights and activations, using a reconfigurable multiplier array and accumulator registers. Its architecture includes dual 8-bit multipliers per PE (configurable to 16-bit), and it supports multiple DL layer types including pooling and activation. A related ASIC version fabricated on a 65 nm process demonstrates a high throughput of 409.6 GOPS and energy efficiency of 5090 GOPS/W—ideal for battery-operated AI systems [210].

DNPU [208] supports both CNNs and RNNs and includes dedicated logic for dense layers, pooling, and activation. It supports variable integer precision (4-, 8-, and 16-bit) to trade off accuracy and energy consumption. In 4-bit mode, DNPU reaches 1200 GOPS and 3900 GOPS/W, a performance level well-suited for time-sensitive and power-aware edge inference.

SDT-CGRA [206] introduces programmable delays to handle temporally aligned data streams, using three MAC instructions per PE and a VLIW control architecture. This level of control allows for efficient handling of variable-length sequences and attention-based models, enhancing its applicability for edge NLP and time-series applications.

Architectures such as Tianjic [211] and Simba [212] implement decentralized weight storage and inter-PE weight exchange, minimizing data movement and improving scalability [196,205]. These designs are especially beneficial for edge systems that prioritize low-latency processing across multiple cores or modules.

The Dynamically Reconfigurable Processor (DRP) [209] integrates a CGRA core that can adapt its topology in real time, supporting various precision formats from binary to 16-bit fixed/floating-point. This reconfigurability makes it an appealing option for heterogeneous edge workloads that require quick switching between DL tasks with minimal overhead.

In the reinforcement learning domain, the CGRA introduced in [213] provides stationary-function PEs, configurable activation units, and internal memory addressing, which enable efficient policy updates and action-value computations at the edge.

Microsoft's NPU, designed under Project Brainwave [214], supports up to 96k MAC operations with an SIMD instruction set optimized for vectorized operations. When mapped to Intel's Stratix 10 FPGA, it can achieve between 10,000 and 35,000 GOPS for real-time RNN inference, balancing high throughput with FPGA-level adaptability—well-matched for industrial or embedded AI edge scenarios.

Across these platforms, performance is heavily influenced by how well memory access patterns are managed. Neural accelerators often rely on three primary data streams: weights, input activations, and intermediate outputs. NoCs allow direct inter-PE communication to reuse data locally, reducing the need for costly DRAM access. Moreover, GLBs minimize memory traffic, while double buffering enables simultaneous computation and data prefetching, increasing pipeline throughput [207].

In comparison to general-purpose CPUs and even GPUs, CGRA-based DL accelerators offer superior energy efficiency, deterministic latency, and model-specific customization, making them especially valuable for EI. While GPUs deliver high raw performance, their high power draw and memory bottlenecks limit their viability in power-constrained environments. FPGAs offer reconfigurability but suffer from long compilation times and toolchain complexity. CGRAs bridge this gap by offering hardware efficiency closer to ASICs while retaining enough flexibility to adapt to different DL models—an essential trait for edge AI systems that require balance between agility and performance.

3.2.4. Vision Processing Unit

Vision Processing Units (VPUs) represent a class of specialized microprocessors optimized for executing DL inference in machine vision tasks at the edge. Designed for low-power environments, VPUs enable real-time inference on pre-trained CNNs and other vision-centric DL models, balancing performance with energy efficiency. Their architecture makes them ideal for integration into edge-centric domains such as surveillance systems, robotics, smart home devices, UAVs, AR/VR wearables, IoT nodes, and mobile devices—where constraints on power, size, and latency are critical.

One prominent VPU example is the Movidius Myriad X [215], a vision-oriented SoC supporting 8- and 16-bit integer as well as 16-bit floating-point operations. It incorporates a 4K image processing pipeline and handles input from up to eight HD sensors. With up to 4000 GOPS peak performance, it excels in vision tasks including image enhancement, object detection, and DL inference. Its efficient parallel dataflow and compatibility with the Myriad Development Kit and Intel's OpenVINO toolkit [216] enable easy model deployment without requiring extensive hardware customization—making it highly suitable for energy-constrained edge environments.

Built around the Myriad X chip, the Intel Neural Compute Stick 2 (NCS 2) [217] offers plug-and-play AI acceleration in a compact USB form factor. It supports a wide array of DL frameworks such as TensorFlow, Caffe, ONNX, PyTorch, and Apache MXNet, allowing seamless migration of existing models. Its ability to connect directly to platforms such as Raspberry Pi and Intel NUC makes it a go-to solution for portable edge inference scenarios, including image classification, speech processing, and lightweight translation tasks.

NeuFlow [179], implemented on a Xilinx Virtex-6 FPGA, is designed for high-throughput, real-time vision tasks such as object detection and semantic segmentation. Despite being developed on older FPGA technology, NeuFlow achieves 12 fps while consuming only 10 W, demonstrating a 100$\times$ speedup over conventional CPUs for specific urban street scene segmentation tasks. While not as power-efficient as modern VPUs, its flexible FPGA base allows for customization, which is valuable in research and prototyping contexts.

Google's Pixel Visual Core (PVC) [218] is another VPU tailored for mobile environments. Built around ARM Cortex-A53 cores and equipped with custom image processors, PVC offers up to 3280 GOPS in FP32 precision at only 6$\sim$8 W, maintaining low idle power draw with bursts of high-speed processing when needed. Its 256-PE ALUs organized in 2D arrays enable dense computation for tasks like HDR+ image processing and on-device ML inference. PVC's integration in Google Pixel devices highlights its effectiveness in delivering real-time vision inference in battery-operated platforms.

The Mobileye EyeQ6 SoC family [219] represents high-performance VPUs tailored for automotive-grade edge computing. Manufactured using a 7 nm FinFET process, EyeQ6 combines a VLIW SIMD architecture with scalable multi-core configurations, enabling up to 34,000 GOPS in INT8 precision. This architecture is built to handle the complex, safety-critical workloads in autonomous driving systems, demonstrating real-time decision-

making capabilities with low latency and tight power envelopes—a vital trait for automotive edge inference.

Compared to general-purpose CPUs and GPUs, VPUs offer a much more power-efficient and application-specific alternative for EI. While they generally lack the programmability and compute versatility of GPUs, their optimized architectures deliver high throughput with minimal energy draw—ideal for edge scenarios requiring sustained performance in thermally constrained environments. Devices such as Myriad X and EyeQ6 clearly illustrate this balance, with compact form factors and efficient DL handling tailored to embedded and real-time workloads.

3.2.5. Microcontroller Unit

Microcontrollers (MCUs) have long served as the cornerstone of embedded systems, powering a wide range of applications including industrial control, consumer electronics, automotive systems, healthcare devices, and vision-based solutions. Their integration into virtually every sector is driven by attributes such as ultra-low power consumption, affordability, and ease of integration. As the demand for intelligent edge computing grows, incorporating DL capabilities into MCUs has opened the door to enabling real-time, low-power AI in billions of edge devices.

Traditionally optimized for control-oriented tasks rather than compute-intensive workloads, MCUs were not designed with DL inference in mind. However, modern MCUs have evolved significantly, now featuring higher clock frequencies, increased cache sizes, and more advanced control units. These improvements, coupled with efficient model compression techniques—such as fixed-point quantization, pruning, and operator fusion—have enabled the deployment of lightweight DL models even within the tight constraints of MCU hardware.

DL inference on MCUs is typically limited to models that have been highly compressed and optimized. Many contemporary MCUs support 8- or 16-bit integer operations and some feature-limited forms of parallelism such as basic pipelining or SIMD-like operations. Despite these improvements, key limitations persist, including restricted on-chip memory, lack of hardware acceleration for matrix operations, and limited instruction-level parallelism. These factors limit the complexity and depth of DL models that can be effectively deployed.

Several software frameworks have been developed to bridge the gap between DL models and MCU hardware. For example, CMSIS-NN [62], tailored for ARM Cortex-M devices, provides a set of optimized neural network kernels and supports quantized model deployment through post-training transformation. Similarly, the PULP-NN library [220] for GAP8 processors offers a memory-efficient execution model using software-controlled scratchpad memory, ideal for scenarios where external DRAM is unavailable.

However, many MCUs still lack hardware support for sub-word operations, which forces 8-bit or smaller computations to be serialized within 32-bit registers, reducing computational throughput. To address this, some researchers have introduced RISC-V ISA extensions for sub-byte quantization support [221]. While this can significantly enhance compression and efficiency, its adoption remains limited due to the relative scarcity and higher cost of commercial RISC-V MCUs.

Real-world deployment examples such as Hello Edge [222] demonstrate the feasibility of using DL models—including CNNs, RNNs, and hybrid architectures—on ARM Cortex-M MCUs for applications such as voice recognition in always-on devices. These systems require a delicate balance between ultra-low power consumption and responsiveness, as they often operate on batteries and demand immediate reactions to stimuli (for example, voice commands or gestures).

TensorFlow-Lite for MCUs (TFLM) further extends DL support to MCUs with minimal resources, such as the ARM Cortex-M3 with solely 16 kB of RAM [216]. TFLM supports various 32-bit MCU platforms including ESP32, STM32, and NXP MCUs, allowing developers to port small-scale models for tasks such as keyword spotting, sensor anomaly detection, and environmental monitoring.

Another framework, MicroAI [223], is specifically designed for 32-bit MCUs and offers a complete toolchain for training, quantizing, and deploying DL models. It has been evaluated on platforms such as Ambiq Apollo3 and STM32L452RE, showing competitive performance in tasks such as image classification and activity monitoring under tight memory and power budgets.

When compared to other hardware options such as VPUs, FPGAs, and ASICs, MCUs stand out for their unmatched power efficiency and cost-effectiveness, making them especially suitable for ultra-constrained edge deployments. While they cannot match the computational capabilities or throughput of VPUs or specialized accelerators, their simplicity, ubiquity, and real-time responsiveness make them ideal for deploying highly optimized DL models in environments where power, space, and cost are paramount. In other words, MCUs are best suited for simple inference tasks rather than complex models or high-throughput applications. They excel in scenarios requiring always-on, low-latency, and privacy-sensitive processing, such as keyword spotting, sensor data analysis, or basic gesture recognition. For more demanding DL tasks—such as real-time object detection or video analysis—alternative edge accelerators with greater compute capabilities and parallelism would be more appropriate.

4. Edge Inference Evaluation Metrics

4.1. Model Size and Memory

Model size is a key factor in assessing the effectiveness of EI. For edge deployments, lightweight network architectures are generally favored. Several hyperparameters—including the number of layers, the size of each layer, the choice of activation functions, the computational complexity per layer, and the density of inter-layer connections—have a direct impact on inference speed and efficiency. As a result, both the overall model architecture and its hyperparameter configuration must be carefully designed. Additionally, the total count of parameters (such as weights and biases) determines the memory footprint, making it another critical performance indicator.

For ease of explanation, let us assume that all model parameters share the same bit width. Under this assumption, the model size (S) can be defined as the product of the total number of parameters (C) and the bit width (B), as shown in Equation (1).

In convolutional layers, let f_i and f_o denote the number of input and output feature maps, respectively. Given a kernel size of $n \times m$, the total number of weights is $n \cdot m \cdot f_i \cdot f_o$, and the number of biases is f_o. Thus, the parameter count for a convolutional layer is $(n \cdot m \cdot f_i + 1) \cdot f_o$.

For dense layers, let d_i and d_o represent the number of input and output nodes. The number of weights is $d_i \cdot d_o$, and the number of biases is d_o, resulting in a parameter count of $(d_i + 1) \cdot d_o$. The value of d_i is derived from the dimensionality of the output feature map from the preceding layer.

Other layers, such as input and pooling layers, typically do not introduce additional parameters. Therefore, if the model contains N_{conv} convolutional layers and N_{dense} dense layers, the total parameter count is given by Equation (2).

$$S = C \cdot B, \tag{1}$$

$$C = \sum_{N_{\text{conv}}} (n \cdot m \cdot f_i + 1) \cdot f_o + \sum_{N_{\text{dense}}} (d_i + 1) \cdot d_o. \quad (2)$$

Table 10 illustrates an example of model size calculation. The input size is $1 \times 30 \times 30$ (channel $\times$ height $\times$ width). The first and second convolutional layers use kernels of size 7×7 and 3×3, respectively, producing 32 and 16 output feature maps. No boundary padding is applied, and the convolutional kernel strides across the input with a step size of 1 pixel. Max pooling layers are configured with 2×2 kernels.

Table 10. Example of DL model size calculation.

Layer	Output Dimensions	Parameters
Input	$1 \times 30 \times 30$	0
1st conv.	$32 \times 24 \times 24$	$(7 \cdot 7 \cdot 1 + 1) \cdot 32 = 1600$
1st max pooling	$32 \times 12 \times 12$	0
2nd conv.	$16 \times 10 \times 10$	$(3 \cdot 3 \cdot 32 + 1) \cdot 16 = 4624$
2nd max pooling	$16 \times 5 \times 5$	0
Dense	256	$(16 \cdot 5 \cdot 5 + 1) \cdot 256 = 102,656$
Output	10	$(256 + 1) \cdot 10 = 2570$
Total parameter count (C)		111,450
Bit width (B)		16 bits
Model size (S)		1,783,200 bits = 222.9 KB

4.2. Accuracy

Accuracy measures whether the inference engine can reliably complete its intended task. It is defined as the ratio of correct predictions to the total number of predictions, as shown in Equation (3), where TP, TN, FP, and FN represent true positives, true negatives, false positives, and false negatives, respectively. In EI applications, where computations are predominantly implemented using fixed-point arithmetic, it is crucial to examine the impact of bit precision on model accuracy. Experimental findings in [224] indicate that INT8, INT16, and INT32 representations yield accuracy comparable to that of FP32. However, reducing the precision further to INT4 incurs a noticeable degradation, with accuracy dropping by 3.44%.

$$\text{ACC} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}. \quad (3)$$

Inference accuracy can sometimes be influenced by the speed at which input data is delivered. For example, in video stream processing, rapid frame input may lead to frame drops due to hardware limitations, causing a decline in accuracy [225]. Beyond accuracy, robustness is a vital aspect of inference performance. Models that perform well on clean, ideal datasets in controlled environments may struggle in real-world conditions. As EI aims to deliver DL capabilities directly to users, robustness becomes crucial. Evaluating performance across multiple, well-established datasets helps gauge how well a model generalizes. Additionally, if specific preprocessing steps enhance accuracy, they must be consistently applied during inference to maintain performance.

4.3. Area

Chip area refers to the surface area of the silicon wafer on which the DL inference implementations are fabricated. It is typically measured in square millimeters (mm^2) or square micrometers (μm^2). The area requirement is influenced by several factors, including the fabrication technology node, system architecture, gate count, and on-chip memory size. Due to the intricacies involved, accurately calculating chip area is time-consuming and often impractical during early design stages. To address this, designers rely on estimation

tools such as the Cadence InCyte Chip Estimator [226]. This tool employs a utilization-based methodology, leveraging standard cell and I/O libraries provided by IP vendors and foundries to estimate area.

As chip cost is closely tied to its area, considerations such as memory type and capacity, as well as the complexity of control logic, play a major role in determining overall expense. Inference engines generally offer on-chip memory in the range of a few hundred kilobytes. Therefore, both the chip area and memory requirements must be carefully evaluated during design. One effective strategy is to employ fixed-point arithmetic, which reduces the bit width of model parameters. This not only lowers the complexity of arithmetic units but also decreases the memory footprint required for parameter storage. As demonstrated in [224], using INT16 and INT8 representations can reduce chip area by 2.43× and 4.98×, respectively, compared to conventional FP32 implementations.

4.4. Energy Efficiency

Power consumption, typically measured in milliwatts (mW) or microwatts (μW), represents the total energy required to perform inference tasks. This measurement should account not only for the computation but also for the energy spent on memory access. While emerging energy-harvesting technologies offer alternative power sources for edge devices, minimizing inference power consumption remains a critical priority. The overall power draw is heavily influenced by the number of computations performed and the frequency with which data is transferred to and from off-chip memory. Memory access often dominates energy consumption, making it a key optimization target. In fact, the power cost associated with memory can often be estimated by analyzing the volume of data read and written during each inference cycle.

As defined in [227], the total power consumption of a DL model is the sum of the power consumption by all its layers. The power consumption for each layer can be expressed as

$$P_{\text{layer}} = P_{\text{data}} + P_{\text{comp}}, \tag{4}$$

where P_{data} denotes the energy required for reading, writing, and transferring data—including weights, biases, and input/output feature maps—and P_{comp} represents the energy consumed in performing MACs.

Given the significant difference in energy costs for on-chip and off-chip memory access, it is essential to account for memory hierarchy when estimating P_{data}. This involves optimizing memory access patterns and calculating the number of bits accessed at each memory level. Based on this analysis, the normalized energy cost per bit for each memory tier can be determined. Similarly, accurate estimation of P_{comp} requires computing the total number of MACs involved and multiplying by the energy cost per MAC operation.

Energy efficiency evaluates how effectively a device performs operations relative to its power consumption. As edge devices typically rely on battery power, achieving ultra-low power consumption along with high computational efficiency is crucial. Energy efficiency is a core metric in assessing DL inference performance and is commonly expressed in giga operations per second per watt (GOPS/W) or tera operations per second per watt (TOPS/W), indicating the number of operations executed per watt of energy used. In systems where floating-point computations dominate, the FLOPS per watt (FLOPS/W) metric is also employed. The efficiency of energy use is affected by the model size, computational complexity, and memory access frequency.

4.5. Latency and Throughput

Latency refers to the amount of time an inference engine takes to complete a single inference task. It is a key factor for achieving real-time performance, especially in time-

critical applications. Typically measured in milliseconds (ms) or microseconds (µs), latency represents the time gap between receiving an input and producing the corresponding output. In EI, minimizing latency is highly desirable. For example, real-time applications such as AR/VR and robotic vision often demand response times within a few milliseconds.

Several factors influence latency, including the hardware specifications of the edge device, resource availability, and the execution strategy of the inference process. Optimizing these aspects is essential to ensure fast and efficient inference performance in real-world environments. Similar to chip area estimation, determining the exact inference latency of DL models is often impractical. Instead, a recent work [228] proposes an estimation framework that (i) converts the model into a directed acyclic graph, (ii) encodes the graph using adjacency and feature matrices, (iii) applies Graph Neural Networks (GNNs) and Gate Recurrent Units (GRUs) to learn graph representations, and (iv) predicts latency based on the extracted features.

Throughput in DL refers to the maximum number of inputs a network can handle within a specific time frame. It is typically measured by the number of operations an inference accelerator can execute per second. Higher throughput reflects superior processing performance. Common units for measuring throughput include giga operations per second (GOPS) and tera operations per second (TOPS). As most DL computations involve MAC operations, throughput is also often represented as giga MACs per second (GMACS) or tera MACs per second (TMACS). Each MAC comprises a multiplication followed by an addition—equating two operations—so, the TOPS-to-TMACS ratio is 2:1.

Using the same notations introduced for convolutional and dense layers in the model size calculation, the total number of floating-point operations can be expressed as

$$T = \sum_{N_{\text{conv}}} (n \cdot m \cdot f_i \cdot f_o \cdot H \cdot W) + \sum_{N_{\text{dense}}} (d_i \cdot d_o), \tag{5}$$

where H and W denote the height and width of the output feature map. Given the total operation count T, throughput can be computed by dividing T by the estimated latency. To convert throughput from GOPS or TOPS to GMACS or TMACS, simply multiply by two.

Achieving high throughput is essential for enabling real-time performance across a wide range of applications such as autonomous driving, medical imaging, security systems, and industrial automation. Real-time inference requires both low latency and high throughput. The throughput capability of a system is closely tied to the number of processing cores on the chip, making core count a critical factor when evaluating inference hardware.

5. Challenges and Opportunities

5.1. Automatic Hardware Mapping

FPGAs and CGRAs offer flexibility to address some of the limitations of ASICs by allowing for the reconfiguration of DL operations, functional blocks, or entire network architectures. However, translating trained DL models into hardware-compatible forms for inference remains a complex, time-consuming, and largely manual process. A notable gap exists in the development of automated tools that can efficiently map DNNs onto hardware platforms. Current solutions for automating DL model deployment on FPGAs or CGRAs are still suboptimal, highlighting the need for more capable and streamlined mapping frameworks.

ML frameworks play a vital role in enabling rapid model prototyping across diverse applications. Expanding these frameworks with additional functions and libraries can significantly ease the integration of compression techniques. For example, NVIDIA's cuSPARSE [63] and Intel's oneMKL [229] libraries facilitate sparse matrix operations, while

TensorFlow supports both training-time and post-training quantization. Xilinx also contributes with FINN-R [230], a framework that enables quantized inference on FPGAs. To advance the deployment of efficient DL models on reconfigurable hardware, the development of more sophisticated training and compression-support libraries is essential.

5.2. Cross-Layer Mapping and Runtime Adaptation for Edge Deployment

While automatic hardware mapping plays a vital role in optimizing DNN inference for specific platforms, static mapping alone is insufficient in dynamic edge environments. Edge devices frequently experience fluctuations in workload, power availability, temperature, and communication latency. These factors necessitate adaptive deployment strategies that can adjust inference behavior at runtime to maintain efficiency and responsiveness.

Recent frameworks [231,232] have started integrating runtime information into deployment decisions. Apollo [231] provides real-time, cycle-accurate power modeling through on-chip power meters that track key RTL signal transitions. This allows edge systems to introspect and adapt inference tasks based on live power consumption, enabling runtime control strategies such as throttling, migration, or selective model scaling. AdaptiveNet [232] further extends this concept by offering a comprehensive online optimization and deployment adaptation framework. It combines system-level feedback (energy or latency) with model parameters to dynamically select optimal deployment configurations on edge devices.

These cross-layer frameworks highlight the importance of moving beyond static mappings and incorporating runtime feedback loops into the deployment pipeline. Compilers and runtime environments must evolve to support flexible scheduling, latency-aware kernel fusion, memory-aware operator placement, and dynamic precision tuning. By integrating model structure, platform, constraints, and runtime context, such systems can better navigate the trade-offs between accuracy, latency, and energy efficiency in real-world edge scenarios.

5.3. Automatic and Edge-Aware Compression

Most current compression models rely heavily on manual tuning and expert knowledge—for example, selecting quantization bit widths, determining rank values in matrix decomposition, or setting layer-specific sparsity levels. To maximize memory efficiency and minimize computation while maintaining acceptable accuracy, compression-related hyperparameters—such as pruning ratios, quantization precision, pruning patterns across layers, and training epochs—should be automatically optimized. As a result, developing automated compression strategies presents a compelling area for future research.

Although many compression techniques are applied after training, integrating them into the training phase remains complex due to the iterative nature of the learning process. However, automating compression during training could lead to better compression rates and more efficient hardware deployment. This opens up exciting research opportunities for designing compression-aware training workflows that accelerate convergence and reduce computational demands.

Despite the growing variety of DNN compression methods, the field lacks standardized evaluation metrics to enable fair and consistent comparisons. Establishing robust benchmarking criteria for compression performance remains an open and valuable direction for further investigation.

5.4. Neural Architecture Search for Edge Inference

NAS has the potential to deliver highly accurate and efficient models. However, one of its primary challenges lies in the substantial computational cost and time required to explore the vast search space. This opens promising research directions for developing

more efficient NAS methods capable of generating lightweight DL architectures suitable for edge deployment.

Given the diversity of edge hardware platforms and specialized neural accelerators, each with unique performance characteristics, incorporating hardware-awareness into NAS is crucial. A hardware-aware NAS approach can use real-time performance feedback from target devices to tailor architectures specifically optimized for those platforms. For example, ProxylessNAS [35] demonstrates the ability to search for architectures directly compatible with hardware constraints. Nevertheless, the increasing variety of edge devices, particularly in the IoT field, further expands the NAS search space and adds complexity to the optimization process.

5.5. Edge Training

DL model training is typically conducted in the cloud, where the process is static and isolated, with no built-in mechanism to retain and apply learned knowledge over time. This approach not only demands extensive datasets but also limits the scope of application. Enabling on-device retraining and transfer learning on the edge could unlock numerous possibilities, especially in scenarios where data is dynamic and continuous learning from new inputs is essential.

Although edge training accelerators such as ScaleDeep [233] and HyPar [234] have been introduced, they often overlook critical aspects such as data sparsity and the use of variable precision in gradients, parameters, and activations. This leaves ample room for enhancement by incorporating layer-wise optimizations and supporting mixed-precision computations to increase performance and energy efficiency. Given the intensive computation requirements of training compared to inference, these challenges become even more pronounced on edge devices.

Most edge training approaches rely heavily on supervised learning, which assumes access to abundant, high-quality labeled data—a condition rarely met in real-world settings, where data is typically sparse and unlabeled. Active learning [235] offers partial relief by enabling models to query the most informative data for labeling, but it still depends on human annotation and works best with smaller datasets. Federated learning provides a promising alternative by decentralizing training: models are trained locally on edge devices using private data, and only model parameters are shared to collaboratively build a global model without centralizing the data.

To further address data limitations, strategies such as incremental learning and data augmentation can be employed. Moreover, Lifelong Machine Learning (LML) [236] offers a forward-looking solution by mimicking human-like adaptability—continuously learning from previous experiences and applying that knowledge to new tasks. While LML is currently designed for high-performance computing systems, adapting it for edge environments presents a compelling research frontier. Achieving this would enable edge devices to better cope with changing conditions and data scarcity in real-world applications.

6. Conclusions

This review contributes a structured and up-to-date synthesis of deep learning (DL) inference strategies tailored for edge intelligence, with several distinguishing features that set it apart from existing surveys. Unlike prior works that emphasize DL architectures or hardware implementations in isolation, this paper offers a holistic view that integrates edge-specific model compression techniques, embedded AI hardware platforms, and memory optimization strategies—areas that are often underrepresented or treated separately in the literature.

A key novelty of this survey lies in its comprehensive taxonomy that bridges the gap between algorithmic methods and system-level deployment challenges, particularly highlighting recent advancements in model compression (for example, pruning, quantization, knowledge distillation) within the context of both hardware and software co-design for edge inference. We also distinguish our work by systematically evaluating software frameworks and compilers in conjunction with hardware accelerators—something previous reviews have not fully addressed.

This paper identifies several gaps in the current body of work, including the lack of end-to-end analysis of software—hardware co-design pipelines, limited support for adaptive optimization in resource-limited environments, and the insufficient integration of energy-aware design across the DL stack. These gaps indicate opportunities for future research aimed at making DL models more portable, scalable, and efficient at the edge.

For practitioners and engineers, we recommend adopting lightweight DL models that are co-optimized with deployment platforms, using quantization-aware training, structured pruning, and memory-efficient operations as standard practices. Additionally, we emphasize the importance of choosing compilers and runtime environments that align with the target edge hardware to maximize performance and minimize power consumption.

The relevance of this topic continues to grow as edge AI becomes central to applications in smart cities, autonomous systems, and real-time sensing. Addressing the unique constraints of edge environments—such as limited energy, computing power, and memory—will be essential to unlocking the next generation of intelligent, responsive, and sustainable AI-powered systems.

Author Contributions: Conceptualization, B.K.; methodology, D.N.; formal analysis, D.N.; investigation, D.N. and H.-C.P.; resources, B.K.; writing—original draft preparation, D.N.; writing—review and editing, D.N., H.-C.P. and B.K.; visualization, D.N. and H.-C.P.; supervision, B.K. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2023R1A2C1004592).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The raw data supporting the conclusions of this article will be made available by the authors on request.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Sze, V.; Chen, Y.H.; Yang, T.J.; Emer, J.S. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proc. IEEE* **2017**, *105*, 2295–2329. [CrossRef]
2. Kingma, D.; Ba, J. Adam: A Method for Stochastic Optimization. *arXiv* **2014**, arXiv:1412.6980. [CrossRef]
3. Leon, B. *Online Learning and Neural Networks*; Cambridge University Press: Cambridge, UK, 1998; Chapter Online Algorithms and Stochastic Approximations, pp. 9–42.
4. Statista. Number of Licensed Cellular Internet of Things (IoT) Connections Worldwide from 2021 to 2030. Available online: https://www.statista.com/statistics/1403316/global-licensed-cellular-iot-connections/ (accessed on 17 March 2025).
5. Pouyanfar, S.; Sadiq, S.; Yan, Y.; Tian, H.; Tao, Y.; Reyes, M.P.; Shyu, M.L.; Chen, S.C.; Iyengar, S.S. A Survey on Deep Learning: Algorithms, Techniques, and Applications. *ACM Comput. Surv.* **2018**, *51*, 1–36. [CrossRef]
6. Alom, M.Z.; Taha, T.M.; Yakopcic, C.; Westberg, S.; Sidike, P.; Nasrin, M.S.; Hasan, M.; Van Essen, B.C.; Awwal, A.A.S.; Asari, V.K. A State-of-the-Art Survey on Deep Learning Theory and Architectures. *Electronics* **2019**, *8*, 292. [CrossRef]
7. Cheng, Y.; Wang, D.; Zhou, P.; Zhang, T. Model Compression and Acceleration for Deep Neural Networks: The Principles, Progress, and Challenges. *IEEE Signal Process. Mag.* **2018**, *35*, 126–136. [CrossRef]

8. Wang, E.; Davis, J.J.; Zhao, R.; Ng, H.C.; Niu, X.; Luk, W.; Cheung, P.Y.K.; Constantinides, G.A. Deep Neural Network Approximation for Custom Hardware: Where We've Been, Where We're Going. *ACM Comput. Surv.* **2019**, *52*, 1–39. [CrossRef]
9. Capra, M.; Bussolino, B.; Marchisio, A.; Shafique, M.; Masera, G.; Martina, M. An Updated Survey of Efficient Hardware Architectures for Accelerating Deep Convolutional Neural Networks. *Future Internet* **2020**, *12*, 113. [CrossRef]
10. Moolchandani, D.; Kumar, A.; Sarangi, S. Accelerating CNN Inference on ASICs: A Survey. *J. Syst. Archit.* **2021**, *113*, 101887. [CrossRef]
11. Shawahna, A.; Sait, S.M.; El-Maleh, A. FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review. *IEEE Access* **2019**, *7*, 7823–7859. [CrossRef]
12. Mittal, S. A survey of FPGA-based accelerators for convolutional neural networks. *Neural Comput. Appl.* **2020**, *32*, 1109–1139. [CrossRef]
13. Li, M.; Liu, Y.; Liu, X.; Sun, Q.; You, X.; Yang, H.; Luan, Z.; Gan, L.; Yang, G.; Qian, D. The Deep Learning Compiler: A Comprehensive Survey. *IEEE Trans. Parallel Distrib. Syst.* **2021**, *32*, 708–727. [CrossRef]
14. Howard, A.; Zhu, M.; Chen, B.; Kalenichenko, D.; Wang, W.; Weyand, T.; Andreetto, M.; Adam, H. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv* **2017**, arXiv:1704.04861. [CrossRef]
15. Chollet, F. Xception: Deep Learning with Depthwise Separable Convolutions. In Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Honolulu, HI, USA, 21–26 July 2017; pp. 1800–1807. [CrossRef]
16. Sandler, M.; Howard, A.; Zhu, M.; Zhmoginov, A.; Chen, L.C. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In Proceedings of the 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Salt Lake City, UT, USA, 18–23 June 2018; pp. 4510–4520. [CrossRef]
17. Iandola, F.; Han, S.; Moskewicz, M.; Ashraf, K.; Dally, W.; Keutzer, K. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. *arXiv* **2016**, arXiv:1602.07360. [CrossRef]
18. Krizhevsky, A.; Sutskever, I.; Hinton, G. ImageNet classification with deep convolutional neural networks. *Commun. ACM* **2017**, *60*, 84–90. [CrossRef]
19. Gholami, A.; Kwon, K.; Wu, B.; Tai, Z.; Yue, X.; Jin, P.; Zhao, S.; Keutzer, K. SqueezeNext: Hardware-Aware Neural Network Design. In Proceedings of the 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), Salt Lake City, UT, USA, 18–22 June 2018; pp. 1719–171909. [CrossRef]
20. Zhang, X.; Zhou, X.; Lin, M.; Sun, J. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. In Proceedings of the 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Salt Lake City, UT, USA, 18–23 June 2018; pp. 6848–6856. [CrossRef]
21. Huang, G.; Liu, S.; Maaten, L.; Weinberger, K. CondenseNet: An Efficient DenseNet Using Learned Group Convolutions. In Proceedings of the 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Salt Lake City, UT, USA, 18–23 June 2018; pp. 2752–2761. [CrossRef]
22. Xiong, Y.; Kim, H.; Hedau, V. ANTNets: Mobile Convolutional Neural Networks for Resource Efficient Image Classification. *arXiv* **2019**, arXiv:1904.03775. [CrossRef]
23. Winograd, S. On multiplication of 2 × 2 matrices. *Linear Algebra Its Appl.* **1971**, *4*, 381–388. [CrossRef]
24. Lavin, A.; Gray, S. Fast Algorithms for Convolutional Neural Networks. In Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, 27–30 June 2016; pp. 4013–4021. [CrossRef]
25. Meng, L.; Brothers, J. Efficient Winograd Convolution via Integer Arithmetic. *arXiv* **2019**, arXiv:1901.01965. [CrossRef]
26. Lu, L.; Liang, Y.; Xiao, Q.; Yan, S. Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs. In Proceedings of the 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Napa, CA, USA, 30 April–2 May 2017; pp. 101–108. [CrossRef]
27. Kala, S.; Mathew, J.; Jose, B.R.; Nalesh, S. Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs. In Proceedings of the 2019 32nd International Conference on VLSI Design and 2019 18th International Conference on Embedded Systems (VLSID), Delhi, India, 5–9 January 2019; pp. 209–214. [CrossRef]
28. Hardieck, M.; Kumm, M.; Moller, K.; Zipf, P. Reconfigurable Convolutional Kernels for Neural Networks on FPGAs. In Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside, CA, USA, 24–26 February 2019; pp. 43–52. [CrossRef]
29. Howard, A.; Sandler, M.; Chu, G.; Chen, L.C.; Chen, B.; Tan, M.; Wang, W.; Zhu, Y.; Pang, R.; Vansudevan, V.; et al. Searching for MobileNetV3. *arXiv* **2019**, arXiv:1905.02244. [CrossRef]
30. Yang, T.J.; Howard, A.; Chen, B.; Zhang, X.; Go, A.; Sandler, M.; Sze, V.; Adam, H. NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications. In Proceedings of the Computer Vision–ECCV 2018: 15th European Conference, Munich, Germany, 8–14 September 2018, pp. 289–304. [CrossRef]
31. Smithson, S.; Yang, G.; Gross, W.; Meyer, B. Neural networks designing neural networks: Multi-objective hyper-parameter optimization. In Proceedings of the 35th International Conference on Computer-Aided Design (ICCAD), Austin, TX, USA, 7–10 November 2016; pp. 1–8. [CrossRef]

32. Zhang, L.; Yang, Y.; Jiang, Y.; Zhu, W.; Liu, Y. Fast Hardware-Aware Neural Architecture Search. In Proceedings of the 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), Seattle, WA, USA, 14–19 June 2020; pp. 2959–2967. [CrossRef]
33. Tan, M.; Chen, B.; Pang, R.; Vasudevan, V.; Sandler, M.; Howard, A.; Le, Q. MnasNet: Platform-Aware Neural Architecture Search for Mobile. In Proceedings of the 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Long Beach, CA, USA, 15–20 June 2019; pp. 2815–2823. [CrossRef]
34. Wu, B.; Dai, X.; Zhang, P.; Wang, Y.; Sun, F.; Wu, Y.; Tian, Y.; Vajda, P.; Jia, Y.; Keutzer, K. FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search. In Proceedings of the 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Long Beach, CA, USA, 15–20 June 2019; pp. 10726–10734. [CrossRef]
35. Cai, H.; Zhu, L.; Han, S. ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware. In Proceedings of the 7th International Conference on Learning Representations (ICLR), New Orleans, LA, USA, 6–9 May 2019.
36. Sinha, N.; Shabayek, A.; Kacem, A.; Rostami, P.; Shneider, C.; Aouada, D. Hardware Aware Evolutionary Neural Architecture Search using Representation Similarity Metric. In Proceedings of the 2024 IEEE/CVF Winter Conference on Applications of Computer Vision (WACV), Waikoloa, HI, USA, 3–8 January 2024; pp. 2616–2625. [CrossRef]
37. Han, C.; Chuang, G.; Tianzhe, W.; Zhekai, Z.; Song, H. Once-for-All: Train One Network and Specialize it for Efficient Deployment. In Proceedings of the 8th International Conference on Learning Representations (ICLR), Addis Ababa, Ethiopia, 26–30 April 2020.
38. Dai, X.; Yin, H.; Jha, N. NeST: A Neural Network Synthesis Tool Based on a Grow-and-Prune Paradigm. *IEEE Trans. Comput.* **2019**, *68*, 1487–1497. [CrossRef]
39. Cao, S.; Zhang, C.; Yao, Z.; Xiao, W.; Nie, L.; Zhan, D.; Liu, Y.; Wu, M.; Zhang, L. Efficient and Effective Sparse LSTM on FPGA with Bank-Balanced Sparsity. In Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside, CA, USA, 24–26 February 2019; pp. 63–72. [CrossRef]
40. Zhu, M.; Zhang, T.; Gu, Z.; Xie, Y. Sparse Tensor Core: Algorithm and Hardware Co-Design for Vector-wise Sparse Neural Networks on Modern GPUs. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, Columbus, OH, USA, 12–16 October 2019; pp. 359–371. [CrossRef]
41. Wen, W.; Wu, C.; Wang, Y.; Chen, Y.; Li, H. Learning structured sparsity in deep neural networks. In Proceedings of the 30th International Conference on Neural Information Processing Systems, Barcelona, Spain, 5–10 December 2016; pp. 2082–2090.
42. Mao, H.; Han, S.; Pool, J.; Li, W.; Liu, X.; Wang, Y.; Dally, W.J. Exploring the Granularity of Sparsity in Convolutional Neural Networks. In Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), Honolulu, HI, USA, 21–26 July 2017; pp. 1927–1934. [CrossRef]
43. Huang, Q.; Zhou, K.; You, S.; Neumann, U. Learning to Prune Filters in Convolutional Neural Networks. In Proceedings of the 2018 IEEE Winter Conference on Applications of Computer Vision (WACV), Lake Tahoe, NV, USA, 12–15 March 2018; pp. 709–718. [CrossRef]
44. Yu, J.; Lukefahr, A.; Palframan, D.; Dasika, G.; Das, R.; Mahlke, S. Scalpel: Customizing DNN pruning to the underlying hardware parallelism. In Proceedings of the 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), Toronto, ON, Canada, 24–28 June 2017; pp. 548–560. [CrossRef]
45. Han, S.; Pool, J.; Tran, J.; Dally, W. Learning both weights and connections for efficient neural networks. In Proceedings of the 29th International Conference on Neural Information Processing Systems-Volume 1, Montreal, QC, Canada, 7–12 December 2015; pp. 1135–1143.
46. Simonyan, K.; Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv* **2015**, arXiv:1409.1556. [CrossRef]
47. Jelcicova, Z.; Verhelst, M. Delta Keyword Transformer: Bringing Transformers to the Edge through Dynamically Pruned Multi-Head Self-Attention. *arXiv* **2022**, arXiv:2204.03479. [CrossRef]
48. Huang, S.; Liu, N.; Liang, Y.; Peng, H.; Li, H.; Xu, D.; Xie, M.; Ding, C. An Automatic and Efficient BERT Pruning for Edge AI Systems. In Proceedings of the 23rd International Symposium on Quality Electronic Design (ISQED), Santa Clara, CA, USA, 6–7 April 2022; pp. 1–6. [CrossRef]
49. Manessi, F.; Rozza, A.; Bianco, S.; Napoletano, P.; Schettini, R. Automated Pruning for Deep Neural Network Compression. In Proceedings of the 24th International Conference on Pattern Recognition (ICPR), Beijing, China, 20–24 August 2018; pp. 657–664. [CrossRef]
50. Molchanov, P.; Tyree, S.; Karras, T.; Aila, T.; Kautz, J. Pruning Convolutional Neural Networks for Resource Efficient Inference. *arXiv* **2017**, arXiv:1611.06440. [CrossRef]
51. You, Z.; Yan, K.; Ye, J.; Ma, M.; Wang, P. Gate decorator: Global filter pruning method for accelerating deep convolutional neural networks. In Proceedings of the 33rd International Conference on Neural Information Processing Systems, Vancouver, QC, Canada, 8–14 December 2019; pp. 2133–2144.

52. Wang, H.; Zhang, Q.; Wang, Y.; Yu, L.; Hu, H. Structured Pruning for Efficient ConvNets via Incremental Regularization. In Proceedings of the 2019 International Joint Conference on Neural Networks (IJCNN), Budapest, Hungary, 14–19 July 2019; pp. 1–8. [CrossRef]
53. Yang, T.J.; Chen, Y.H.; Sze, V. Designing Energy-Efficient Convolutional Neural Networks Using Energy-Aware Pruning. In Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Honolulu, HI, USA, 21–26 July 2017; pp. 6071–6079. [CrossRef]
54. Wong, T.T. Performance evaluation of classification algorithms by k-fold and leave-one-out cross validation. *Pattern Recognit.* **2015**, *48*, 2839–2846. [CrossRef]
55. Struharik, R.; Vukobratovic, B.; Erdeljan, A.; Rakanovic, D. CoNNa–Hardware accelerator for compressed convolutional neural networks. *Microprocess. Microsystems* **2020**, *73*, 102991. [CrossRef]
56. Nurvitadhi, E.; Venkatesh, G.; Sim, J.; Marr, D.; Huang, R.; Hock, J.; Liew, Y.; Srivatsan, K.; Moss, D.; Subhaschandra, S.; et al. Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks? In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2017; pp. 5–14. [CrossRef]
57. Shen, Y.; Ferdman, M.; Milder, P. Escher: A CNN Accelerator with Flexible Buffering to Minimize Off-Chip Transfer. In Proceedings of the 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Napa, CA, USA, 30 April–2 May 2017; pp. 93–100. [CrossRef]
58. Han, S.; Liu, X.; Mao, H.; Pu, J.; Pedram, A.; Horowitz, M.A.; Dally, W.J. SCNN: An accelerator for compressed-sparse convolutional neural networks. In Proceedings of the 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), Toronto, ON, Canada, 24–28 June 2017; pp. 27–40. [CrossRef]
59. Chen, Y.H.; Yang, T.J.; Emer, J.; Sze, V. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *IEEE J. Emerg. Sel. Top. Circuits Syst.* **2019**, *9*, 292–308. [CrossRef]
60. Zhou, X.; Du, Z.; Guo, Q.; Liu, S.; Liu, C.; Wang, C.; Zhou, X.; Li, L.; Chen, T.; Chen, Y. Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach. In Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Fukuoka, Japan, 20–24 October 2018; pp. 15–28. [CrossRef]
61. NVIDIA. TensorRT Documentation. Available online: https://docs.nvidia.com/deeplearning/tensorrt/latest/index.html (accessed on 15 April 2025).
62. Lai, L.; Suda, N.; Chandra, V. CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs. *arXiv* **2018**, arXiv:1801.06601. [CrossRef]
63. NVIDIA. cuSPARSE. Available online: https://docs.nvidia.com/cuda/cusparse/ (accessed on 9 May 2025).
64. Micikevicius, P.; Narang, S.; Alben, J.; Diamos, G.; Else, E.; Garcia, D.; Ginsburg, B.; Houston, M.; Kuchaiev, O.; Venkatesh, G.; et al. Mixed Precision Training. *arXiv* **2018**, arXiv:1710.03740. [CrossRef]
65. Wang, N.; Choi, J.; Brand, D.; Chen, C.Y.; Gopalakrishnan, K. Training deep neural networks with 8-bit floating point numbers. In Proceedings of the 32nd International Conference on Neural Information Processing Systems, Montreal, QC, Canada, 3–8 December 2018; pp. 7686–7695.
66. Armeniakos, G.; Zervakis, G.; Soudris, D.; Henkel, J. Hardware Approximate Techniques for Deep Neural Network Accelerators: A Survey. *ACM Comput. Surv.* **2022**, *55*, 1–36. [CrossRef]
67. Horowitz, M. Computing's energy problem (and what we can do about it). In Proceedings of the 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC), San Francisco, CA, USA, 9–13 February 2014; pp. 10–14. [CrossRef]
68. Moons, B.; Verhelst, M. A 0.3–2.6 TOPS/W precision-scalable processor for real-time large-scale ConvNets. In Proceedings of the 2016 IEEE Symposium on VLSI Circuits (VLSI-Circuits), Honolulu, HI, USA, 15–17 June 2016, pp. 1–2. [CrossRef]
69. Vanhoucke, V.; Senior, A.; Mao, M. Improving the speed of neural networks on CPUs. In Proceedings of the Deep Learning and Unsupervised Feature Learning NIPS Workshop, Granada, Spain, 12–17 December 2011; p. 4.
70. Nasution, M.A.; Chahyati, D.; Fanany, M.I. Faster R-CNN with structured sparsity learning and Ristretto for mobile environment. In Proceedings of the 2017 International Conference on Advanced Computer Science and Information Systems (ICACSIS), Bali, Indonesia, 28–29 October 2017; pp. 309–314. [CrossRef]
71. Gysel, P.; Motamedi, M.; Ghiasi, S. Hardware-oriented Approximation of Convolutional Neural Networks. *arXiv* **2016**, arXiv:1604.03168. [CrossRef]
72. Shang, Y.; Liu, G.; Kompella, R.R.; Yan, Y. Enhancing Post-training Quantization Calibration through Contrastive Learning. In Proceedings of the 2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Seattle, WA, USA, 16–22 June 2024; pp. 15921–15930. [CrossRef]
73. Shen, H.; Mellempudi, N.; He, X.; Gao, Q.; Wang, C.; Wang, M. Efficient Post-training Quantization with FP8 Formats. In Proceedings of the 7th Machine Learning and Systems (MLSys), Santa Clara, CA, USA, 13–16 May 2024, pp. 483–498.
74. Judd, P.; Albericio, J.; Moshovos, A. Stripes: Bit-Serial Deep Neural Network Computing. *IEEE Comput. Archit. Lett.* **2017**, *16*, 80–83. [CrossRef]

75. Sharma, H.; Park, J.; Suda, N.; Lai, L.; Chau, B.; Kim, J.K.; Chandra, V.; Esmaeilzadeh, H. Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Network. In Proceedings of the 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), Los Angeles, CA, USA, 1–6 June 2018; pp. 764–775. [CrossRef]
76. Courbariaux, M.; Hubara, I.; Soudry, D.; El-Yahniv, R.; Bengio, Y. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or −1. *arXiv* **2016**, arXiv:1602.02830. [CrossRef]
77. Rastegari, M.; Ordonez, V.; Redmon, J.; Farhadi, A. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. In Proceedings of the Computer Vision–ECCV 2016, Amsterdam, The Netherlands, 11–14 October 2016; pp. 525–542. [CrossRef]
78. Zhou, A.; Yao, A.; Guo, Y.; Xu, L.; Chen, Y. Incremental Network Quantization: Towards Lossless CNNs with Low-Precision Weights. *arXiv* **2017**, arXiv:1702.03044. [CrossRef]
79. Han, S.; Mao, H.; Dally, W. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *arXiv* **2016**, arXiv:1510.00149. [CrossRef]
80. Lee, J.; Kim, C.; Kang, S.; Shin, D.; Kim, S.; Yoo, H.J. UNPU: A 50.6TOPS/W unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision. In Proceedings of the 2018 IEEE International Solid-State Circuits Conference-(ISSCC), San Francisco, CA, USA, 11–15 February 2018; pp. 218–220. [CrossRef]
81. Sharify, S.; Lascorz, A.D.; Siu, K.; Judd, P.; Moshovos, A. Loom: Exploiting Weight and Activation Precisions to Accelerate Convolutional Neural Networks. In Proceedings of the 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 24–28 June 2018; pp. 1–6. [CrossRef]
82. Ryu, S.; Kim, H.; Yi, W.; Kim, J.J. BitBlade: Area and Energy-Efficient Precision-Scalable Neural Network Accelerator with Bitwise Summation. In Proceedings of the 2019 56th ACM/IEEE Design Automation Conference (DAC), Las Vegas, NV, USA, 2–6 June 2019; pp. 1–6.
83. Courbariaux, M.; Bengio, Y.; David, J.P. BinaryConnect: Training deep neural networks with binary weights during propagations. In Proceedings of the 29th International Conference on Neural Information Processing Systems-Volume 2, Montreal, QC, Canada, 7–12 December 2015; pp. 3123–3131.
84. Liu, B.; Li, F.; Wang, X.; Zhang, B.; Yan, J. Ternary Weight Networks. In Proceedings of the 2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Rhodes Island, Greece, 4–10 June 2023; pp. 1–5. [CrossRef]
85. Zhu, C.; Han, S.; Mao, H.; Dally, W. Trained Ternary Quantization. *arXiv* **2017**, arXiv:1612.01064. [CrossRef]
86. Zhou, S.; Wu, Y.; Ni, Z.; Zhou, X.; Wen, H.; Zou, Y. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. *arXiv* **2018**, arXiv:1606.06160. [CrossRef]
87. Hubara, I.; Courbariaux, M.; Soudry, D.; El-Yaniv, R.; Bengio, Y. Quantized neural networks: Training neural networks with low precision weights and activations. *J. Mach. Learn. Res.* **2017**, *18*, 6869–6898.
88. Cai, Z.; He, X.; Sun, J.; Vasconcelos, N. Deep Learning with Low Precision by Half-Wave Gaussian Quantization. In Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Honolulu, HI, USA, 21–26 July 2017; pp. 5406–5414. [CrossRef]
89. Xiao, G.; Lin, J.; Seznec, M.; Wu, H.; Demouth, J.; Han, S. SmoothQuant: Accurate and efficient post-training quantization for large language models. In Proceedings of the 40th International Conference on Machine Learning (ICML), Honolulu, HI, USA, 23–29 July 2023; pp. 38087–38099.
90. Liu, J.; Niu, L.; Yuan, Z.; Yang, D.; Wang, X.; Liu, W. PD-Quant: Post-Training Quantization Based on Prediction Difference Metric. In Proceedings of the 2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Vancouver, BC, Canada, 17–23 June 2023; pp. 24427–24437. [CrossRef]
91. Bie, A.; Venkitesh, B.; Monteiro, J.; Haidar, M.; Rezagholizadeh, M. A Simplified Fully Quantized Transformer for End-to-end Speech Recognition. *arXiv* **2020**, arXiv:1911.03604. [CrossRef]
92. Li, Z.; Gu, Q. I-ViT: Integer-only Quantization for Efficient Vision Transformer Inference. In Proceedings of the 2023 IEEE/CVF International Conference on Computer Vision (ICCV), Paris, France, 1–6 October 2023, pp. 17019–17029. [CrossRef]
93. Shen, L.; Edalati, A.; Meyer, B.; Gross, W.; Clark, J. Robustness to distribution shifts of compressed networks for edge devices. *arXiv* **2024**, arXiv:2401.12014. [CrossRef]
94. Yang, Y.; Huang, Q.; Wu, B.; Zhang, T.; Ma, L.; Gambardella, G.; Blott, M.; Lavagno, L.; Vissers, K.; Wawrzynek, J.; et al. Synetgy: Algorithm-hardware Co-design for ConvNet Accelerators on Embedded FPGAs. In Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside, CA, USA, 24–26 February 2019; pp. 23–32. [CrossRef]
95. Baluja, S.; Marwood, D.; Covell, M.; Johnston, N. No Multiplication? No Floating Point? No Problem! Training Networks for Efficient Inference. *arXiv* **2018**, arXiv:1809.09244. [CrossRef]
96. Sarwar, S.S.; Venkataramani, S.; Raghunathan, A.; Roy, K. Multiplier-less Artificial Neurons exploiting error resiliency for energy-efficient neural computing. In Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, Germany, 14–18 March 2016; pp. 145–150.

97. Wang, L.; Yoon, K.J. Knowledge Distillation and Student-Teacher Learning for Visual Intelligence: A Review and New Outlooks. *IEEE Trans. Pattern Anal. Mach. Intell.* **2022**, *44*, 3048–3068. [CrossRef]
98. Bucilua, C.; Caruana, R.; Niculescu-Mizil, A. Model compression. In Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, 20–23 August 2006; pp. 535–541. [CrossRef]
99. Hinton, G.; Vinyals, O.; Dean, J. Distilling the Knowledge in a Neural Network. *arXiv* **2015**, arXiv:1503.02531. [CrossRef]
100. Heo, B.; Lee, M.; Yun, S.; Choi, J. Knowledge transfer via distillation of activation boundaries formed by hidden neurons. In Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, Honolulu, HI, USA, 27 January–1 February 2019; pp. 3779–3787. [CrossRef]
101. Huang, Z.; Wang, N. Like What You Like: Knowledge Distill via Neuron Selectivity Transfer. *arXiv* **2017**, arXiv:1707.01219. [CrossRef]
102. Romero, A.; Ballas, N.; Kahou, S.; Chassang, A.; Gatta, C.; Bengio, Y. FitNets: Hints for Thin Deep Nets. *arXiv* **2015**, arXiv:1412.6550. [CrossRef]
103. Li, D.; Wang, X.; Kong, D. DeepRebirth: Accelerating deep neural network execution on mobile devices. In Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence, New Orleans, LA, USA, 2–7 February 2018; pp. 2322–2330.
104. Zhou, G.; Fan, Y.; Cui, R.; Bian, W.; Zhu, X.; Gai, K. Rocket launching: A universal and efficient framework for training well-performing light net. In Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence, New Orleans, LA, USA, 2–7 February 2018; pp. 4580–4587.
105. Jiao, X.; Yin, Y.; Shang, L.; Jiang, X.; Chen, X.; Li, L.; Wang, F.; Liu, Q. TinyBERT: Distilling BERT for Natural Language Understanding. *arXiv* **2020**, arXiv:1909.10351. [CrossRef]
106. Chaulwar, A.; Malik, L.; Krajewski, M.; Reichel, F.; Lundbæk, L.N.; Huth, M.; Matejczyk, B. Extreme compression of sentence-transformer ranker models: Faster inference, longer battery life, and less storage on edge devices. *arXiv* **2022**, arXiv:2207.12852. [CrossRef]
107. Zhang, Y.; Xiang, T.; Hospedales, T.; Lu, H. Deep Mutual Learning. In Proceedings of the 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Salt Lake City, UT, USA, 18–23 June 2018; pp. 4320–4328. [CrossRef]
108. Mirzadeh, S.I.; Farajtabar, M.; Li, A.; Levine, N.; Matsukawa, A.; Ghasemzadeh, H. Improved Knowledge Distillation via Teacher Assistant. *arXiv* **2019**, arXiv:1902.03393. [CrossRef]
109. Yuan, L.; Tay, F.; Li, G.; Wang, T.; Feng, J. Revisiting Knowledge Distillation via Label Smoothing Regularization. In Proceedings of the 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Seattle, WA, USA, 13–19 June 2020; pp. 3902–3910. [CrossRef]
110. Wang, T.; Zhu, J.Y.; Torralba, A.; Efros, A. Dataset Distillation. *arXiv* **2020**, arXiv:1811.10959. [CrossRef]
111. Bohdal, O.; Yang, Y.; Hospedales, T. Flexible Dataset Distillation: Learn Labels Instead of Images. *arXiv* **2020**, arXiv:2006.08572. [CrossRef]
112. Kang, D.; Emmons, J.; Abuzaid, F.; Bailis, P.; Zaharia, M. Flexible Dataset Distillation: Learn Labels Instead of Images. *Proc. VLDB Endow.* **2017**, *10*, 1586–1597. [CrossRef]
113. Park, E.; Kim, D.; Kim, S.; Kim, Y.D.; Kim, G.; Yoon, S.; Yoo, S. Big/little deep neural network for ultra low power inference. In Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis, Amsterdam, The Netherlands, 4–9 October 2015; pp. 124–132.
114. Teerapittayanon, S.; McDanel, B.; Kung, H. BranchyNet: Fast inference via early exiting from deep neural networks. In Proceedings of the 23rd International Conference on Pattern Recognition (ICPR), Cancun, Mexico, 4–8 December 2016; pp. 2464–2469. [CrossRef]
115. Teerapittayanon, S.; McDanel, B.; Kung, H. Distributed Deep Neural Networks Over the Cloud, the Edge and End Devices. In Proceedings of the 37th International Conference on Distributed Computing Systems (ICDCS), Atlanta, GA, USA, 5–8 June 2017; pp. 328–339. [CrossRef]
116. Li, L.; Ota, K.; Dong, M. Deep Learning for Smart Industry: Efficient Manufacture Inspection System with Fog Computing. *IEEE Trans. Ind. Inform.* **2018**, *14*, 4665–4673. [CrossRef]
117. Wang, X.; Yu, F.; Dou, Z.Y.; Darrell, T.; Gonzalez, J. SkipNet: Learning Dynamic Routing in Convolutional Networks. In Proceedings of the European Conference on Computer Vision (ECCV), Munich, Germany, 8–14 September 2018; pp. 409–424.
118. Moons, B.; Uytterhoeven, R.; Dehaene, W.; Verhelst, M. 14.5 Envision: A 0.26-to-10TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable Convolutional Neural Network processor in 28nm FDSOI. In Proceedings of the 2017 IEEE International Solid-State Circuits Conference (ISSCC), San Francisco, CA, USA, 5–9 February 2017; pp. 246–247. [CrossRef]

119. Parsa, M.; Panda, P.; Sen, S.; Roy, K. Staged Inference using Conditional Deep Learning for energy efficient real-time smart diagnosis. In Proceedings of the 39th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC), Jeju, Republic of Korea, 11–15 July 2017; pp. 78–81. [CrossRef]
120. Tan, M.; Le, Q. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. *arXiv* **2020**, arXiv:1905.11946. [CrossRef]
121. Elsen, E.; Dukhan, M.; Gale, T.; Simonyan, K. Fast Sparse ConvNets. In Proceedings of the 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Seattle, WA, USA, 13–19 June 2020; pp. 14617–14626. [CrossRef]
122. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.; Kaiser, L.; Polosukhin, I. Attention is All you Need. In Proceedings of the Advances in Neural Information Processing Systems 30 (NIPS 2017), Long Beach, CA, USA, 4–9 December 2017.
123. Devlin, J.; Chang, M.W.; Lee, K.; Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv* **2019**, arXiv:1810.04805. [CrossRef]
124. Gale, T.; Elsen, E.; Hooker, S. The State of Sparsity in Deep Neural Networks. *arXiv* **2019**, arXiv:1902.09574. [CrossRef]
125. Wolf, T.; Debut, L.; Sanh, V.; Chaumond, J.; Delangue, C.; Moi, A.; Cistac, P.; Rault, T.; Louf, R.; Funtowicz, M.; et al. HuggingFace's Transformers: State-of-the-art Natural Language Processing. *arXiv* **2020**, arXiv:1910.03771. [CrossRef]
126. Wang, H.; Zhang, Z.; Han, S. SpAtten: Efficient Sparse Attention Architecture with Cascade Token and Head Pruning. In Proceedings of the 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA), Seoul, Republic of Korea, 27 February–3 March 2021; pp. 97–110. [CrossRef]
127. Han, S.; Kang, J.; Mao, H.; Hu, Y.; Li, X.; Li, Y.; Xie, D.; Luo, H.; Yao, S.; Wang, Y.; et al. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA. In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2017; pp. 75–84. [CrossRef]
128. Zhu, M.; Gupta, S. To prune, or not to prune: Exploring the efficacy of pruning for model compression. *arXiv* **2017**, arXiv:1710.01878. [CrossRef]
129. Gupta, U.; Reagen, B.; Pentecost, L.; Donato, M.; Tambe, T.; Rush, A.; Wei, G.; Brooks, D. MASR: A Modular Accelerator for Sparse RNNs. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, Seattle, WA, USA, 23–26 September 2019; pp. 1–14. [CrossRef]
130. Yang, Q.; Mao, J.; Wang, Z.; Li, H. DASNet: Dynamic Activation Sparsity for Neural Network Efficiency Improvement. In Proceedings of the 31st International Conference on Tools with Artificial Intelligence (ICTAI), Portland, OR, USA, 4–6 November 2019; pp. 1401–1405. [CrossRef]
131. Guo, J.; Potkonjak, M. Pruning Filters and Classes: Towards On-Device Customization of Convolutional Neural Networks. In Proceedings of the 1st International Workshop on Deep Learning for Mobile Systems and Applications, Niagara Falls, NY, USA, 23 June 2017; pp. 13–17. [CrossRef]
132. Kim, Y.D.; Park, E.; Yoo, S.; Choi, T.; Yang, L.; Shin, D. Compression of Deep Convolutional Neural Networks for Fast and Low Power Mobile Applications. *arXiv* **2016**, arXiv:1511.06530. [CrossRef]
133. Denton, E.; Zaremba, W.; Bruna, J.; LeCun, Y.; Fergus, R. Exploiting linear structure within convolutional networks for efficient evaluation. In Proceedings of the 28th International Conference on Neural Information Processing Systems-Volume 1, Montreal, QC, Canada, 8–13 December 2014; pp. 1269–1277.
134. Jaderberg, M.; Vedaldi, A.; Zisserman, A. Speeding up Convolutional Neural Networks with Low Rank Expansions. *arXiv* **2014**, arXiv:1405.3866. [CrossRef]
135. Han, S.; Shen, H.; Philipose, M.; Agarwal, S.; Wolman, A.; Krishnamurthy, A. MCDNN: An Approximation-Based Execution Framework for Deep Stream Processing Under Resource Constraints. In Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services, Singapore, 26–30 June 2016; pp. 123–136. [CrossRef]
136. Liu, Z.; Li, J.; Shen, Z.; Huang, G.; Yan, S.; Zhang, C. Learning Efficient Convolutional Networks through Network Slimming. In Proceedings of the 2017 IEEE International Conference on Computer Vision (ICCV), Venice, Italy, 22–29 October 2017; pp. 2755–2763. [CrossRef]
137. Bhattacharya, S.; Lane, N. Sparsification and Separation of Deep Learning Layers for Constrained Resource Inference on Wearables. In Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM, Stanford, CA, USA, 14–16 November 2016; pp. 176–189. [CrossRef]
138. Maji, P.; Bates, D.; Chadwick, A.; Mullins, R. ADaPT: Optimizing CNN inference on IoT and mobile devices using approximately separable 1-D kernels. In Proceedings of the 1st International Conference on Internet of Things and Machine Learning, Liverpool, UK, 17–18 October 2017; pp. 1–12. [CrossRef]
139. Crowley, E.; Gray, G.; Storkey, A. Moonshine: Distilling with Cheap Convolutions. In Proceedings of the Advances in Neural Information Processing Systems 31 (NeurIPS 2018), Montreal, QC, Canada, 2–8 December 2018.

140. Liu, C.; Zoph, B.; Neumann, M.; Shlens, J.; Hua, W.; Li, L.J.; Li, F.F.; Yuille, A.; Huang, J.; Murphy, K. Progressive Neural Architecture Search. In Proceedings of the Computer Vision–ECCV 2018: 15th European Conference, Munich, Germany, 8–14 September 2018; pp. 19–35. [CrossRef]
141. Gordon, A.; Eban, E.; Nachum, O.; Chen, B.; Wu, H.; Yang, T.J.; Choi, E. MorphNet: Fast & Simple Resource-Constrained Structure Learning of Deep Networks. In Proceedings of the 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Salt Lake City, UT, USA, 18–23 June 2018; pp. 1586–1595. [CrossRef]
142. Shafiee, M.; Li, F.; Chwyl, B.; Wong, A. SquishedNets: Squishing SqueezeNet further for edge device scenarios via deep evolutionary synthesis. *arXiv* **2017**, arXiv:1711.07459. [CrossRef]
143. Wong, A.; Famuori, M.; Shafiee, M.; Li, F.; Chwyl, B.; Chung, J. YOLO Nano: A Highly Compact You Only Look Once Convolutional Neural Network for Object Detection. In Proceedings of the 2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition (EMC2-NIPS), Vancouver, BC, Canada, 13 December 2019; pp. 22–25. [CrossRef]
144. Zagoruyko, S.; Komodakis, N. Paying More Attention to Attention: Improving the Performance of Convolutional Neural Networks via Attention Transfer. *arXiv* **2017**, arXiv:1612.03928. [CrossRef]
145. Sau, B.; Balasubramanian, V. Deep Model Compression: Distilling Knowledge from Noisy Teachers. *arXiv* **2016**, arXiv:1610.09650. [CrossRef]
146. Tsung, P.K.; Tsai, S.F.; Pai, A.; Lai, S.J.; Lu, C. High performance deep neural network on low cost mobile GPU. In Proceedings of the 2016 IEEE International Conference on Consumer Electronics (ICCE), Las Vegas, NV, USA, 7–11 January 2016; pp. 69–70. [CrossRef]
147. Zoph, B.; Vasudevan, V.; Shlens, J.; Le, Q. Learning Transferable Architectures for Scalable Image Recognition. *arXiv* **2018**, arXiv:1707.07012. [CrossRef]
148. Lee, J.; Lee, J.; Han, D.; Lee, J.; Park, G.; Yoo, H.J. 7.7 LNPU: A 25.3TFLOPS/W Sparse Deep-Neural-Network Learning Processor with Fine-Grained Mixed Precision of FP8-FP16. In Proceedings of the 2019 IEEE International Solid-State Circuits Conference-(ISSCC), San Francisco, CA, USA, 17–21 February 2019; pp. 142–144. [CrossRef]
149. Zhang, S.; Du, Z.; Zhang, L.; Lan, H.; Liu, S.; Li, L.; Guo, Q.; Chen, T.; Chen, Y. Cambricon-X: An accelerator for sparse neural networks. In Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Taipei, Taiwan, 15–19 October 2016; pp. 1–12. [CrossRef]
150. Kim, D.; Ahn, J.; Yoo, S. ZeNA: Zero-Aware Neural Network Accelerator. *IEEE Des. Test* **2018**, *35*, 39–46. [CrossRef]
151. Kung, H.T.; McDanel, B.; Zhang, S.Q. Adaptive Tiling: Applying Fixed-size Systolic Arrays To Sparse Convolutional Neural Networks. In Proceedings of the 24th International Conference on Pattern Recognition (ICPR), Beijing, China, 20–24 August 2018; pp. 1006–1011. [CrossRef]
152. Deng, C.; Liao, S.; Xie, Y.; Parhi, K.; Qian, X.; Yuan, B. PermDNN: Efficient compressed DNN architecture with permuted diagonal matrices. In Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture, Fukuoka, Japan, 20–24 October 2018; pp. 189–202. [CrossRef]
153. Kung, H.; McDanel, B.; Zhang, S. Packing Sparse Convolutional Neural Networks for Efficient Systolic Array Implementations: Column Combining Under Joint Optimization. In Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems, Providence, RI, USA, 13–17 April 2019; pp. 821–834. [CrossRef]
154. Akhlaghi, V.; Yazdanbakhsh, A.; Samadi, K.; Gupta, R.K.; Esmaeilzadeh, H. SnaPEA: Predictive Early Activation for Reducing Computation in Deep Convolutional Neural Networks. In Proceedings of the 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), Los Angeles, CA, USA, 1–6 June 2018; pp. 662–673. [CrossRef]
155. Song, M.; Zhao, J.; Hu, Y.; Zhang, J.; Li, T. Prediction Based Execution on Deep Neural Networks. In Proceedings of the 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), Los Angeles, CA, USA, 1–6 June 2018; pp. 752–763. [CrossRef]
156. Han, S.; Liu, X.; Mao, H.; Pu, J.; Pedram, A.; Horowitz, M.A.; Dally, W.J. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In Proceedings of the 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), Seoul, Republic of Korea, 18–22 June 2016; pp. 243–254. [CrossRef]
157. Dorrance, R.; Ren, F.; Markovic, D. A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on FPGAs. In Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 26–28 February 2014; pp. 161–170. [CrossRef]
158. Albericio, J.; Judd, P.; Hetherington, T.; Aamodt, T.; Jerger, N.E.; Moshovos, A. Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing. In Proceedings of the 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), Seoul, Republic of Korea, 18–22 June 2016; pp. 1–13. [CrossRef]
159. Aimar, A.; Mostafa, H.; Calabrese, E.; Rios-Navarro, A.; Tapiador-Morales, R.; Lungu, I.A.; Milde, M.B.; Corradi, F.; Linares-Barranco, A.; Liu, S.C.; et al. NullHop: A Flexible Convolutional Neural Network Accelerator Based on Sparse Representations of Feature Maps. *IEEE Trans. Neural Networks Learn. Syst.* **2019**, *30*, 644–656. [CrossRef] [PubMed]

160. Hegde, K.; Yu, J.; Agrawal, R.; Yan, M.; Pellauer, M.; Fletcher, C. UCNN: Exploiting computational reuse in deep neural networks via weight repetition. In Proceedings of the 45th Annual International Symposium on Computer Architecture, Los Angeles, CA, USA, 2–6 June 2018; pp. 674–687. [CrossRef]
161. Ma, X.; Guo, F.M.; Niu, W.; Lin, X.; Tang, J.; Ma, K.; Ren, B.; Wang, Y. PCONV: The Missing but Desirable Sparsity in DNN Weight Pruning for Real-Time Execution on Mobile Devices. In Proceedings of the AAAI Conference on Artificial Intelligence, New York, NY, USA, 7–12 February 2020; pp. 5117–5124. [CrossRef]
162. Niu, W.; Ma, X.; Lin, S.; Wang, S.; Qian, X.; Lin, X.; Wang, Y.; Ren, B. PatDNN: Achieving Real-Time DNN Execution on Mobile Devices with Pattern-based Weight Pruning. In Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 16–20 March 2020; pp. 907–922. [CrossRef]
163. Guan, H.; Liu, S.; Ma, X.; Niu, W.; Ren, B.; Shen, X.; Wang, Y.; Zhao, P. CoCoPIE: Enabling real-time AI on off-the-shelf mobile devices via compression-compilation co-design. *Commun. ACM* **2021**, *64*, 62–68. [CrossRef]
164. Lin, J.; Chen, W.M.; Lin, Y.; Cohn, J.; Gan, C.; Han, S. MCUNet: Tiny deep learning on IoT devices. In Proceedings of the 34th International Conference on Neural Information Processing Systems, Vancouver, BC, Canada, 6–12 December 2020; pp. 11711–11722.
165. Chen, T.; Moreau, T.; Jiang, Z.; Zheng, L.; Yan, E.; Cowan, M.; Shen, H.; Wang, L.; Hu, Y.; Ceze, L.; et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. *arXiv* **2018**, arXiv:1802.04799. [CrossRef]
166. Jiang, X.; Wang, H.; Chen, Y.; Wu, Z.; Wang, L.; Zou, B.; Yang, Y.; Cui, Z.; Cai, Y.; Yu, T.; et al. MNN: A Universal and Efficient Inference Engine. *arXiv* **2020**, arXiv:2002.12418. [CrossRef]
167. Chen, Y.H.; Krishna, T.; Emer, J.S.; Sze, V. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE J. Solid-State Circuits* **2016**, *52*, 127–138. [CrossRef]
168. Alzantot, M.; Wang, Y.; Ren, Z.; Srivastava, M. RSTensorFlow: GPU Enabled TensorFlow for Deep Learning on Commodity Android Devices. In Proceedings of the 1st International Workshop on Deep Learning for Mobile Systems and Applications, Niagara Falls, NY, USA, 23 June 2017; pp. 7–12. [CrossRef]
169. Oskouei, S.; Golestani, H.; Hashemi, M.; Ghiasi, S. CNNdroid: GPU-Accelerated Execution of Trained Deep Convolutional Neural Networks on Android. In Proceedings of the 24th ACM International Conference on Multimedia, Amsterdam, The Netherlands, 15–19 October 2016; pp. 1201–1205. [CrossRef]
170. Rizvi, S.; Cabodi, G.; Patti, D.; Francini, G. GPGPU Accelerated Deep Object Classification on a Heterogeneous Mobile Platform. *Electronics* **2016**, *5*, 88. [CrossRef]
171. Cao, Q.; Balasubramanian, N.; Balasubramanian, A. MobiRNN: Efficient Recurrent Neural Network Execution on Mobile GPU. In Proceedings of the 1st International Workshop on Deep Learning for Mobile Systems and Applications, Niagara Falls, NY, USA, 23 June 2017; pp. 1–6. [CrossRef]
172. Marchetti, A.; Marchetti, G. *RenderScript: Parallel Computing on Android, the Easy Way*; Alberto Marchetti: Rome, Italy, 2016; Chapter What Is RenderScript; pp. 9–13.
173. Motamedi, M.; Fong, D.; Ghiasi, S. Cappuccino: Efficient CNN Inference Software Synthesis for Mobile System-on-Chips. *IEEE Embed. Syst. Lett.* **2019**, *11*, 9–12. [CrossRef]
174. Huynh, L.; Balan, R.; Lee, Y. DeepSense: A GPU-based Deep Convolutional Neural Network Framework on Commodity Mobile Devices. In Proceedings of the 2016 Workshop on Wearable Systems and Applications, Singapore, 30 June 2016; pp. 25–30. [CrossRef]
175. Rallapalli, S.; Qiu, H.; Bency, A.; Karthikeyan, S.; Govindan, R.; Manjunath, B.; Urgaonkar, R. *Are Very Deep Neural Networks Feasible on Mobile Devices?* University of Southern California Technical Report; University of Southern California: Los Angeles, CA, USA, 2016; pp. 916–965.
176. Redmon, J.; Divvala, S.; Girshick, R.; Farhadi, A. You Only Look Once: Unified, Real-Time Object Detection. In Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, 27–30 June 2016; pp. 779–788. [CrossRef]
177. Lane, N.D.; Bhattacharya, S.; Georgiev, P.; Forlivesi, C.; Jiao, L.; Qendro, L.; Kawsar, F. DeepX: A Software Accelerator for Low-Power Deep Learning Inference on Mobile Devices. In Proceedings of the 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN), Vienna, Austria, 11–14 April 2016; pp. 1–12. [CrossRef]
178. Lane, N.; Bhattacharya, S.; Mathur, A.; Forlivesi, C.; Kawsar, F. DXTK: Enabling Resource-efficient Deep Learning on Mobile and Embedded Devices with the DeepX Toolkit. In Proceedings of the 8th EAI International Conference on Mobile Computing, Applications and Services, Cambridge, UK, 30 November–1 December 2016; pp. 98–107. [CrossRef]
179. Farabet, C.; Martini, B.; Corda, B.; Akselrod, P.; Culurciello, E.; LeCun, Y. NeuFlow: A runtime reconfigurable dataflow processor for vision. In Proceedings of the CVPR 2011 WORKSHOPS, Colorado Springs, CO, USA, 20–25 June 2011; pp. 109–116. [CrossRef]
180. Chen, Y.; Chen, T.; Xu, Z.; Sun, N.; Temam, O. DianNao family: Energy-efficient hardware accelerators for machine learning. *Commun. ACM* **2016**, *59*, 105–112. [CrossRef]

181. Du, Z.; Fasthuber, R.; Chen, T.; Ienne, P.; Li, L.; Luo, T.; Feng, X.; Chen, Y.; Temam, O. ShiDianNao: Shifting vision processing closer to the sensor. In Proceedings of the 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA), Portland, OR, USA, 13–17 June 2015; pp. 92–104. [CrossRef]
182. Kwon, H.; Samajdar, A.; Krishna, T. MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects. In Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems, Williamsburg, VA, USA, 24–28 March 2018; pp. 461–475. [CrossRef]
183. MATLAB. Deep Learning HDL Toolbox. Available online: https://www.mathworks.com/help/deep-learning-hdl/ (accessed on 15 April 2025).
184. AMD. Vitis AI. Available online: https://xilinx.github.io/Vitis-AI/3.5/html/index.html (accessed on 15 April 2025).
185. Qualcomm. Developing Apps with the Qualcomm Neural Processing SDK for AI. Available online: https://docs.qualcomm.com/bundle/publicresource/topics/80-63442-4/developing-apps-qualcomm-neural-processing-sdk.html?product=1601111740010412 (accessed on 15 April 2025).
186. Roane, J. Automated HW/SW Co-Design of DSP Systems Composed of Processors and Hardware Accelerators. Available online: https://www.cadence.com/en_US/home/resources/white-papers/automated-hw-sw-co-design-of-dsp-systems-composed-of-processors-and-hardware-accelerators-wp.html (accessed on 15 April 2025).
187. NVIDIA. NVIDIA Jetson TX2 NX System-on-Module. Available online: https://developer.nvidia.com/downloads/jetson-tx2-nx-system-module-data-sheet (accessed on 7 May 2025).
188. NVIDIA. NVIDIA Jetson Nano. Available online: https://developer.nvidia.com/embedded/jetson-nano (accessed on 7 May 2025).
189. NVIDIA. JetPack SDK. Available online: https://developer.nvidia.com/embedded/jetpack (accessed on 8 May 2025).
190. Cadence. Cadence Tensilica DNA 100 Processor. Available online: https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/company/Events/CDNLive/Secured/Proceedings/IL/2018/ip/IP02.pdf (accessed on 1 May 2025).
191. Synopsys. Synopsys EV7x Vision Processors. Available online: https://www.synopsys.com/dw/ipdir.php?ds=ev7x-vision-processors (accessed on 1 May 2025).
192. Han, D.; Lee, J.; Lee, J.; Yoo, H.J. A 1.32 TOPS/W Energy Efficient Deep Neural Network Learning Processor with Direct Feedback Alignment based Heterogeneous Core Architecture. In Proceedings of the 2019 Symposium on VLSI Circuits, Kyoto, Japan, 9–14 June 2019; pp. C304–C305. [CrossRef]
193. Yuan, Z.; Yue, J.; Yang, H.; Wang, Z.; Li, J.; Yang, Y.; Guo, Q.; Li, X.; Chang, M.F.; Yang, H.; et al. Sticker: A 0.41-62.1 TOPS/W 8Bit Neural Network Processor with Multi-Sparsity Compatible Convolution Arrays and Online Tuning Acceleration for Fully Connected Layers. In Proceedings of the 2018 IEEE Symposium on VLSI Circuits, Honolulu, HI, USA, 18–22 June 2018; pp. 33–34. [CrossRef]
194. Lu, W.; Yan, G.; Li, J.; Gong, S.; Han, Y.; Li, X. FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks. In Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), Austin, TX, USA, 4–8 February 2017; pp. 553–564. [CrossRef]
195. Li, J.; Jiang, S.; Gong, S.; Wu, J.; Yan, J.; Yan, G.; Li, X. SqueezeFlow: A Sparse CNN Accelerator Exploiting Concise Convolution Rules. *IEEE Trans. Comput.* **2019**, *68*, 1663–1677. [CrossRef]
196. Chen, Y.; Luo, T.; Liu, S.; Zhang, S.; He, L.; Wang, J.; Li, L.; Chen, T.; Xu, Z.; Sun, N.; et al. DaDianNao: A Machine-Learning Supercomputer. In Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, Cambridge, UK, 13–17 December 2014; pp. 609–622. [CrossRef]
197. Ceva. Scalable Edge NPU IP for Generative AI. Available online: https://www.ceva-ip.com/product/ceva-neupro-m/ (accessed on 1 May 2025).
198. Coral. Edge TPU Inferencing Overview. Available online: https://coral.ai/docs/edgetpu/inference/ (accessed on 1 May 2025).
199. Guo, K.; Sui, L.; Qiu, J.; Yu, J.; Wang, J.; Yao, S.; Han, S.; Wang, Y.; Yang, H. Angel-Eye: A Complete Design Flow for Mapping CNN Onto Embedded FPGA. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2018**, *37*, 35–47. [CrossRef]
200. Venieris, S.I.; Bouganis, C.S. fpgaConvNet: Mapping Regular and Irregular Convolutional Neural Networks on FPGAs. *IEEE Trans. Neural Netw. Learn. Syst.* **2019**, *30*, 326–342. [CrossRef]
201. Véstias, M.; Policarpo Duarte, R.; T. de Sousa, J.; Neto, H. Lite-CNN: A High-Performance Architecture to Execute CNNs in Low Density FPGAs. In Proceedings of the 28th International Conference on Field Programmable Logic and Applications (FPL), Dublin, Ireland, 27–31 August 2018; pp. 399–3993. [CrossRef]
202. Wang, S.; Li, Z.; Ding, C.; Yuan, B.; Qiu, Q.; Wang, Y.; Lang, Y. C-LSTM: Enabling Efficient LSTM using Structured Compression Techniques on FPGAs. In Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 25–27 February 2018; pp. 11–20. [CrossRef]
203. Gao, C.; Neil, D.; Ceolini, E.; Liu, S.C.; Delbruck, T. DeltaRNN: A Power-efficient Recurrent Neural Network Accelerator. In Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 25–27 February 2018; pp. 21–30. [CrossRef]

204. Fleischer, B.; Shukla, S.; Ziegler, M.; Silberman, J.; Oh, J.; Srinivasan, V.; Choi, J.; Mueller, S.; Agrawal, A.; Babinsky, T.; et al. A Scalable Multi-TeraOPS Deep Learning Processor Core for AI Trainina and Inference. In Proceedings of the 2018 IEEE Symposium on VLSI Circuits, Honolulu, HI, USA, 18–22 June 2018; pp. 35–36. [CrossRef]
205. Jouppi, N.P.; Young, C.; Patil, N.; Patterson, D.; Agrawal, G.; Bajwa, R.; Bates, S.; Bhatia, S.; Boden, N.; Borchers, A.; et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. In Proceedings of the 44th Annual International Symposium on Computer Architecture, Toronto, ON, Canada, 24–28 June 2017; pp. 1–12. [CrossRef]
206. Fan, X.; Wu, D.; Cao, W.; Luk, W.; Wang, L. Stream Processing Dual-Track CGRA for Object Inference. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2018**, *26*, 1098–1111. [CrossRef]
207. Yin, S.; Ouyang, P.; Tang, S.; Tu, F.; Li, X.; Zheng, S.; Lu, T.; Gu, J.; Liu, L.; Wei, S. A High Energy Efficient Reconfigurable Hybrid Neural Network Processor for Deep Learning Applications. *IEEE J. Solid-State Circuits* **2018**, *53*, 968–982. [CrossRef]
208. Shin, D.; Lee, J.; Lee, J.; Lee, J.; Yoo, H.J. DNPU: An Energy-Efficient Deep-Learning Processor with Heterogeneous Multi-Core Architecture. *IEEE Micro* **2018**, *38*, 85–93. [CrossRef]
209. Fujii, T.; Toi, T.; Tanaka, T.; Togawa, K.; Kitaoka, T.; Nishino, K.; Nakamura, N.; Nakahara, H.; Motomura, M. New Generation Dynamically Reconfigurable Processor Technology for Accelerating Embedded AI Applications. In Proceedings of the 2018 IEEE Symposium on VLSI Circuits, Montreal, QC, Canada, 7–12 December 2015; pp. 41–42. [CrossRef]
210. Yin, S.; Ouyang, P.; Tang, S.; Tu, F.; Li, X.; Liu, L.; Wei, S. A 1.06-to-5.09 TOPS/W reconfigurable hybrid-neural-network processor for deep learning applications. In Proceedings of the 2017 Symposium on VLSI Circuits, Kyoto, Japan, 5–8 June 2017; pp. C26–C27. [CrossRef]
211. Pei, J.; Deng, L.; Song, S.; Zhao, M.; Zhang, Y.; Wu, S.; Wang, G.; Zou, Z.; Wu, Z.; He, W.; et al. Towards artificial general intelligence with hybrid Tianjic chip architecture. *Nature* **2019**, *572*, 106–111. [CrossRef]
212. Zimmer, B.; Venkatesan, R.; Shao, Y.S.; Clemons, J.; Fojtik, M.; Jiang, N.; Keller, B.; Klinefelter, A.; Pinckney, N.; Raina, P.; et al. A 0.11 pJ/Op, 0.32-128 TOPS, Scalable Multi-Chip-Module-based Deep Neural Network Accelerator with Ground-Reference Signaling in 16nm. In Proceedings of the 2019 Symposium on VLSI Circuits, Kyoto, Japan, 9–14 June 2019; pp. C300–C301. [CrossRef]
213. Liang, M.; Chen, M.; Wang, Z.; Sun, J. A CGRA based Neural Network Inference Engine for Deep Reinforcement Learning. In Proceedings of the 2018 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS), Chengdu, China, 26–30 October 2018; pp. 540–543. [CrossRef]
214. Fowers, J.; Ovtcharov, K.; Papamichael, M.; Massengill, T.; Liu, M.; Lo, D.; Alkalay, S.; Haselman, M.; Adams, L.; Ghandi, M.; et al. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA), Los Angeles, CA, USA, 1–6 June 2018; pp. 1–14. [CrossRef]
215. Intel. Intel Movidius Myriad X Vision Processing Unit 4GB. Available online: https://www.intel.com/content/www/us/en/products/sku/125926/intel-movidius-myriad-x-vision-processing-unit-4gb/specifications.html (accessed on 15 May 2025).
216. Intel. Intel Distribution of OpenVINO Toolkit. Available online: https://www.intel.com/content/www/us/en/developer/tools/openvino-toolkit/overview.html (accessed on 2 May 2025).
217. Intel. Intel Neural Compute Stick 2. Available online: https://cdrdv2-public.intel.com/749742/neural-compute-stick2-product-brief.pdf (accessed on 15 May 2025).
218. Shacham, O.; Reynders, M. Pixel Visual Core: Image Processing and Machine Learning on Pixel 2. Available online: https://blog.google/products/pixel/pixel-visual-core-image-processing-and-machine-learning-pixel-2/ (accessed on 2 May 2025).
219. Mobileye. The Evolution of EyeQ. Available online: https://www.mobileye.com/technology/eyeq-chip/ (accessed on 2 May 2025).
220. Rossi, D.; Loi, I.; Conti, F.; Tagliavini, G.; Pullini, A.; Marongiu, A. Energy efficient parallel computing on the PULP platform with support for OpenMP. In Proceedings of the 28th Convention of Electrical & Electronics Engineers in Israel (IEEEI), Eilat, Israel, 3–5 December 2014; pp. 1–5. [CrossRef]
221. Garofalo, A.; Tagliavini, G.; Conti, F.; Rossi, D.; Benini, L. XpulpNN: Accelerating Quantized Neural Networks on RISC-V Processors Through ISA Extensions. In Proceedings of the 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 9–13 March 2020; pp. 186–191. [CrossRef]
222. Zhang, Y.; Suda, N.; Lai, L.; Chandra, V. Hello Edge: Keyword Spotting on Microcontrollers. *arXiv* **2018**, arXiv:1711.07128. [CrossRef]
223. Novac, P.E.; Hacene, G.; Pegatoquet, A.; Miramond, B.; Gripon, V. Quantization and Deployment of Deep Neural Networks on Microcontrollers. *Sensors* **2021**, *21*, 2984. [CrossRef]
224. Hashemi, S.; Anthony, N.; Tann, H.; Bahar, R.; Reda, S. Understanding the impact of precision quantization on the accuracy and energy of neural networks. In Proceedings of the Conference on Design, Automation & Test in Europe (DATE), Lausanne, Switzerland, 27–31 March 2017; pp. 1478–1483.
225. Lee, J.; Hwang, K. YOLO with adaptive frame control for real-time object detection applications. *Multimed. Tools Appl.* **2022**, *81*, 36375–36396. [CrossRef]

226. Cadence. Cadence InCyte Chip Estimator: Fast and Accurate Estimation of IC Size, Power, Performance, and Cost. Available online: http://pdf2.solecsy.com/567/201d55e2-371a-4d6a-b7a5-00461f050f65.pdf (accessed on 7 June 2025).
227. Yang, T.J.; Chen, Y.H.; Emer, J.; Sze, V. A method to estimate the energy consumption of deep neural networks. In Proceedings of the 51st Asilomar Conference on Signals, Systems, and Computers, Pacific Grove, CA, USA, 29 October–1 November 2017; pp. 1916–1920. [CrossRef]
228. Zhao, W.; Wang, L.; Duan, F. ELPG: End-to-End Latency Prediction for Deep-Learning Model Inference on Edge Devices. In Proceedings of the 10th International Conference on Computer and Communications (ICCC), Chengdu, China, 13–16 December 2024; pp. 2160–2165. [CrossRef]
229. Intel. Intel oneAPI Math Kernel Library (oneMKL). Available online: https://www.intel.com/content/www/us/en/docs/oneapi/programming-guide/2024-1/intel-oneapi-math-kernel-library-onemkl.html (accessed on 9 May 2025).
230. Blott, M.; Preuber, T.; Fraser, N.; Gambardella, G.; Obrien, K.; Umuroglu, Y.; Leeser, M.; Vissers, K. FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks. *ACM Trans. Reconfigurable Technol. Syst. (TRETS)* **2018**, *11*, 1–23. [CrossRef]
231. Xie, Z.; Xu, X.; Walker, M.; Knebel, J.; Palaniswamy, K.; Hebert, N.; Hu, J.; Yang, H.; Chen, Y.; Das, S. APOLLO: An Automated Power Modeling Framework for Runtime Power Introspection in High-Volume Commercial Microprocessors. In Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Virtual, 18–22 October 2021; pp. 1–14. [CrossRef]
232. Wen, H.; Li, Y.; Zhang, Z.; Jiang, S.; Ye, X.; Quyang, Y.; Zhang, Y.; Liu, Y. AdaptiveNet: Post-deployment Neural Architecture Adaptation for Diverse Edge Environments. In Proceedings of the 29th Annual International Conference on Mobile Computing and Networking, Madrid, Spain, 2–6 October 2023; pp. 1–17. [CrossRef]
233. Venkataramani, S.; Ranjan, A.; Banerjee, S.; Das, D.; Avancha, S.; Jagannathan, A.; Durg, A.; Nagaraj, D.; Kaul, B.; Dubey, P.; et al. SCALEDEEP: A scalable compute architecture for learning and evaluating deep networks. In Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA), Toronto, ON, Canada, 24–28 June 2017; pp. 13–26. [CrossRef]
234. Song, L.; Mao, J.; Zhuo, Y.; Qian, X.; Li, H.; Chen, Y. HyPar: Towards Hybrid Parallelism for Deep Learning Accelerator Array. In Proceedings of the 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), Washington, DC, USA, 16–20 February 2019; pp. 56–68. [CrossRef]
235. Ren, P.; Xian, Y.; Chang, X.; Huang, P.Y.; Li, Z.; Gupta, B.; Chen, X.; Wang, X. A Survey of Deep Active Learning. *ACM Comput. Surv. (CSUR)* **2021**, *54*, 1–40. [CrossRef]
236. Chen, Z.; Liu, B. *Lifelong Machine Learning*; Springer: Cham, Switzerland, 2018; pp. 1–207.