SparOA: Sparse and Operator-aware Hybrid Scheduling for Edge DNN Inference

Ziyang Zhang
Politecnico di Milano
Milan, Italy
ziyang.zhang@polimi.it

Jie Liu
Harbin Institute of Technology
Shenzhen, China
jieliu@hit.edu.cn

Luca Mottola
Politecnico di Milano
Milan, Italy
luca.mottola@polimi.it

ABSTRACT

The resource demands of deep neural network (DNN) models introduce significant performance challenges, especially when deployed on resource-constrained edge devices. Existing solutions like model compression often sacrifice accuracy, while specialized hardware remains costly and inflexible. Hybrid inference methods, however, typically overlook how operator characteristics impact performance. In this work, we present SparOA, a CPU-GPU hybrid inference framework, which leverages both sparsity and computational intensity to optimize operator scheduling. SparOA embraces aforementioned challenges through three key components: (1) a threshold predictor that accurately determines optimal sparsity and computational intensity thresholds; (2) a reinforcement learning-based scheduler that dynamically optimizes resource allocation based on real-time hardware states; and (3) a hybrid inference engine that enhances efficiency through asynchronous execution and batch size optimization. Extensive results show that SparOA achieves an average speedup of 1.22×∼1.31× compared to all baselines, and outperforms the CPU-Only by up to 50.7×. Also, SparOA achieves optimal energy-per-inference, consuming 7%∼16% less energy than the SOTA co-execution baseline.

ACM Reference Format:
Ziyang Zhang, Jie Liu, and Luca Mottola. 2025. SparOA: Sparse and Operator-aware Hybrid Scheduling for Edge DNN Inference. In *ACM Conference (Conference'17), July 2017, Washington, DC, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

An increasing number of intelligent applications rely on deep neural network (DNN) models for real-time inference at the network edge [42]. These applications span various fields, including but not limited to autonomous driving [46], industrial automation [10], and medical diagnostics [13]. However, edge devices often have limited computational power, storage capacity, and energy supply, making it highly challenging to deploy and run complex DNN models on these devices.

To address this challenge, software and hardware optimization methods exist in the literature. For instance, model compression [14] reduces model size and computational complexity through pruning, quantization, and knowledge distillation, making the models more suitable for deployment on resource-constrained devices. However, such methods often come at the cost of reduced inference accuracy and may fail to meet real-time requirements in certain scenarios. Leveraging specialized hardware, such as TPUs and FPGAs, may accelerate the inference process. While this hardware can significantly enhance performance for specific types of tasks, their high cost and limited flexibility make them difficult to gain wide adoption for general-purpose edge devices.

Hybrid inference. Traditional solutions targeting efficient DNN inference also typically focus on a single hardware component, such as the CPU or GPU. However, these techniques often miss out on the potential gains due to employing multiple components. Hybrid inference [20, 30, 49] techniques dynamically schedule operators within a DNN across different processors. For instance, CoDL [21] dynamically schedules operator execution by evaluating its affinity for heterogeneous processors, thereby optimizing latency.

Most existing hybrid inference methods overlook the impact of operator characteristics, such as sparsity and computational intensity. Significant differences in the computational characteristics and resource requirements of each operator indeed exist. For instance, convolution and fully connected operators typically involve dense matrix multiplication or convolution operations, requiring substantial computational resources. In contrast, activation and normalization operators demand less computation.

Key insight. We simultaneously consider *sparsity* and *computational intensity* as metrics to quantify the resource requirements of operators. First, sparsity is an important indicator of the distribution of computational load in DNN

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

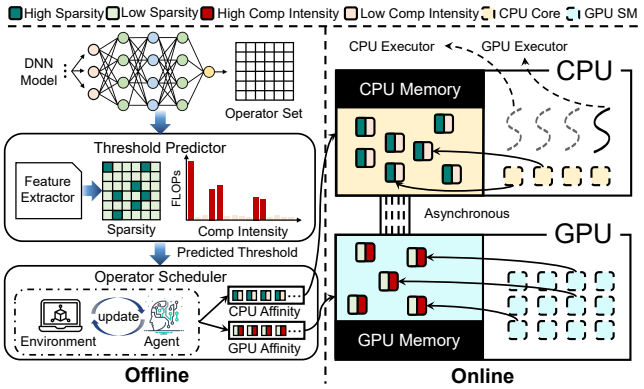


Figure 1: Overview of SparOA.

models. Operators with high sparsity imply that most activation values are zero, allowing these operations to be skipped to reduce computational overhead, yet without affecting the output. Computational intensity refers to the number of floating-point operations (FLOPs) required per unit of input data, serving as a measure of the computational complexity and resource demands. Operators with high computational intensity typically involve dense matrix multiplications or convolutions and require higher computational resources. Convolution and fully connected operators, due to their compute-intensive nature, are better suited for execution on a GPU, while normalization and activation operators are more efficiently processed on a CPU.

Existing work [9, 44] primarily focuses on optimizing inference latency based on sparsity while neglecting computational intensity. For instance, even if an operator exhibits high sparsity, scheduling it to run on a CPU when it also has high computational intensity may result in suboptimal performance. Conversely, parallel computing enabled by GPUs is better suited for such tasks. Existing methods also often employ static scheduling strategies, determining the allocation of operators beforehand. This method cannot adapt to dynamically changing hardware states and input data, leading to suboptimal performance. For instance, in some cases, a GPU may fail to efficiently handle tasks with high computational intensity due to insufficient memory, and the task should then switch to the CPU for execution.

Contribution. We present SparOA, an adaptive operator scheduling strategy based on sparsity and computational intensity. By accurately analyzing the sparsity and computational intensity characteristics of each operator, SparOA can automatically and dynamically optimize the allocation strategy of operators to heterogeneous processors.

Figure 1 shows an overview of SparOA, comprising both offline and online components. The offline phase consists of two modules, i.e., the *threshold predictor* and the *operator scheduler*. The light-weight, yet accurate threshold predictor estimates the optimal thresholds for sparsity and computational intensity of each DNN operator, guiding operator scheduling in the online phase. For instance, operators with high sparsity but low computational intensity are prioritized for the CPU, while operators with low sparsity and high computational intensity are preferentially assigned to the GPU. Based on the predicted sparsity and computational intensity threshold, the operator scheduler optimizes the scheduling strategy for the input DNN model, using reinforcement learning. For the online phase, the *hybrid inference* engine coordinates the synchronous execution of operators on heterogeneous processors according to the optimal scheduling strategy, while using asynchronous and dynamic batching to further improve resource utilization efficiency.

We implement SparOA using commercial off-the-shelf (COTS) edge devices, including NVIDIA Jetson AGX Orin and Orin Nano. We report in Sec. 6 on experiments with popular DNN models including convolutional neural network (CNN)-based and attention-based architectures. The results show that SparOA consistently outperforms state-of-the-art (SOTA) methods, achieving an average 1.22×~1.31× speedup over SOTA compilers (e.g., TensorRT [36], TVM [4]) and co-execution frameworks (e.g., CoDL [21]) , and up to 50.7× speedup over CPU-Only.

Overall, we make the following contributions:

1) we preset a threshold predictor based on sparsity and computational intensity, which accurately predicts the resource requirements of operators;
2) we conceive an operator scheduling strategy based on reinforcement learning, which dynamically optimizes the scheduling strategy according to real-time hardware status and input data;
3) we implement an efficient hybrid inference engine that significantly improves CPU-GPU hybrid inference efficiency through asynchronous execution and dynamic batching optimization.
4) the results show that SparOA achieves optimal latency and energy efficiency compared to the SOTA baselines.

2 BACKGROUND AND MOTIVATION

2.1 Sparsity and Computational Intensity

Sparsity [9, 44] refers to the proportion of zero elements in the activation values and is often used to measure the computational load distribution of a DNN model. High sparsity in an operator means that most of the activation values are zero, allowing these zero-value operations to be skipped.

Evaluating sparsity entails capturing the activation value distribution characteristics of each DNN operator. For instance, the ReLU activation function typically produces high

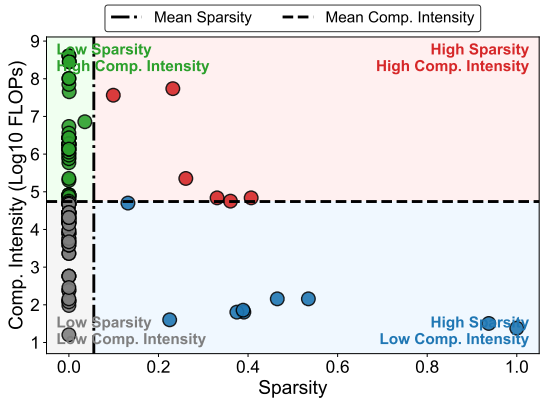


Figure 2: The distribution of sparsity and computational intensity for each operator in MobileNetV3-small, measured on NVIDIA Jetson AGX Orin. We set the batch size to 1.

sparsity because negative values are directly set to zero. Operators with high sparsity, due to their reduced computational workload and lower data transfer latency, are more suitable for execution on the CPU. Conversely, operators with low sparsity require more computational resources and are therefore better suited for GPU processing.

Sparsity may also be used for compressing and optimizing DNN models [9, 59]. For instance, pruning operators with high sparsity may further reduce the computational complexity and storage requirements of a DNN. Further, sparsity analysis may help identify redundant operations within the DNN model, supporting operator fusion.

Computational intensity measures the computational demand of each operator in a DNN, expressed in floating-point operations per second (FLOPs). Operators with high computational intensity may involve dense matrix multiplications, convolution operations, or complex nonlinear transformations. In DNN inference optimization, computational intensity analysis is essential.

For instance, a GPU with parallel computing capabilities is apt to handle tasks with high computational intensity, such as convolutions and fully connected operators. These operators may involve large amounts of matrix multiplications and additions, which can be efficiently completed on the GPU. In contrast, a CPU excels in single-threaded performance, making it more suitable for tasks with low computational intensity, such as normalization and activation functions.

2.2 Motivation

When scheduling operators, it is necessary to consider computational intensity to determine the hardware component for each task. Relying solely on sparsity for scheduling while ignoring the impact of computational intensity may lead to suboptimal performance. Certain operators, despite having high sparsity, may also exhibit high computational intensity, and incorrectly assigning them to the CPU for execution may result in increased latency and resource waste.

Independence of Sparsity and Computational Intensity. Existing hybrid scheduling methods [9, 44] assume a strong negative correlation between sparsity and computational intensity. However, our profiling reveals that these two metrics are, in fact, orthogonal dimensions. Figure 2 illustrates the operator distribution of MobileNetV3-Small across the sparsity-intensity spectrum on the NVIDIA Jetson platform. The analysis uncovers four distinct quadrants:

- *Quadrant II (High Sparsity & High Intensity):* as shown in the top-right of Figure 2, certain operators (e.g., *Conv2d* operator) exhibit significant sparsity (> 0.4) due to accumulated *ReLU* activations in prior layers. However, due to their large channel dimensions, their computational intensity remains extremely high ($> 10^8$ FLOPs). Assigning these operators to the CPU solely based on their high sparsity would cause severe computational bottlenecks, as the CPU lacks the parallelism to handle the massive workload despite the zeros.
- *Quadrant III (Low Sparsity & Low Intensity):* conversely, the bottom-left quadrant contains operators like *BatchNorm2d*. These operators are dense (sparsity ≈ 0) but possess low arithmetic intensity. They are memory-bound rather than compute-bound. Scheduling them to the GPU based solely on their density would incur unnecessary kernel launch and data transfer overheads that outweigh the execution benefits.
- *Quadrants I & IV:* these represent the intuitive cases: dense, heavy operators in quadrant 2 ideal for GPU, and sparse, light operators in quadrant 4 ideal for CPU.

This distribution reveals that neither sparsity nor computational intensity alone is a sufficient proxy for CPU-GPU hybrid operator scheduling. We addresses this by treating them as joint state features, enabling the scheduler to distinguish these complex scenarios. For instance, advisably offloading high-sparsity/high-intensity operators to the GPU while keeping low-sparsity/low-intensity operators on the CPU to minimize end-to-end latency.

3 THRESHOLD PREDICTOR

Existing works usually rely on hand-designed rules to guide the scheduling strategy of operators, which may simply assign all operators with sparsity above a fixed threshold to the CPU and other operators to the GPU. However, these methods show several shortcomings. First, the fixed threshold is difficult to adapt to the diversity of different DNN models

and input data. Second, it ignores the impact of computational intensity on inference performance, which may lead to suboptimal scheduling decisions.

As shown in Figure 3, we leverage the advantages of the Transformer architecture in capturing long-range dependencies and contextual information, and design a Transformer-based threshold predictor to dynamically adjust the sparsity threshold and computational intensity threshold of each operator through automatic learning, and use it as the basis for scheduling decisions.

3.1 Data Preprocessing and Feature Extraction

The input to the threshold predictor comes from statistical data of each operator in the DNN model, including sparsity, computational intensity and input features. The feature extraction process is as follows:

- *Sparsity:* for each activation operator, we calculate sparsity by counting the proportion of non-zero elements. Let the output tensor of a certain layer be $O \in \mathbb{R}^{H \times W \times C}$, where H, W, C represent height, width, and number of channels, respectively. Sparsity ρ can be expressed as

$$\rho = 1 - \frac{c(O)}{n(O)} \tag{1}$$

  where $c(O)$ represents the number of non-zero elements in the tensor, and $n(O)$ represents the total number of elements in the tensor.
- *Computational intensity:* for each compute-intensive operator, we estimate computational intensity by calculating the number of floating-point operations (FLOPs). For instance, the computational intensity of convolution operator I can be expressed as:

$$I = K_h \cdot K_w \cdot C_{in} \cdot C_{out} \cdot H \cdot W \tag{2}$$

  where K_h, K_w are the height and width of the convolution kernel, and C_{in}, C_{out} are the number of input and output channels, respectively.

Instead, the *input features* include batch size B, number of input channels C_{in}, input height H, and width W, extracted directly from the input tensor.

3.2 Predictor Architecture

This section details the architectural design of the threshold predictor. The primary motivation for this architecture is to capture both global dependencies between input features (sparsity, computational intensity, batch size, and tensor dimensions) and temporal correlation across sequential operators. Conventional methods, such as fixed threshold rules, fail to adapt to the diversity of DNN models and dynamic input data, resulting in low prediction accuracy and suboptimal resource utilization. To address this, our objective is to enhance prediction precision while supporting real-time inference optimization through a parameter-efficient yet compact neural network architecture.

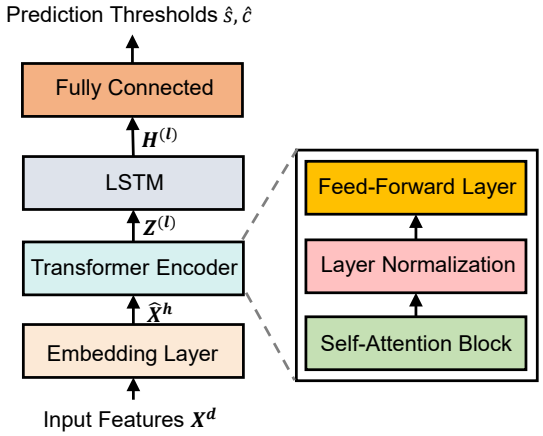


Figure 3: The architecture of threshold predictor.

Our design combines three key components: a Transformer encoder to model long-range feature interactions, an LSTM module to capture sequential operator patterns, and a lightweight output layer for threshold prediction. This hybrid design enables the predictor to: *i)* comprehensively analyze operator-processor affinity through multi-dimensional feature embedding, *ii)* maintain temporal consistency in scheduling decisions for sequential DNN layers, and *iii)* generate precise sparsity/computational intensity thresholds that directly optimize hybrid execution performance metrics.

The design of each component is described in detail below.

- *Embedding layer:* maps the input vector $X = [\rho, I, B, C_{in}, H, W], X \in \mathbb{R}^d$ into a high-dimensional space to capture complex relationships between features. The output dimension of the embedding layer is h (hidden layer dimension).
- *Transformer encoder:* the Transformer encoder consists of multiple self-attention mechanisms and feed-forward networks, used to capture global dependencies between features. The output of each encoder can be represented as:

$$Z^{(l)} = \text{FFN}(\text{LN}(X^{(l-1)} + \text{MHSA}^{(l-1)})) \tag{3}$$

  where l represents the number of Transformer encoder layers, MHSA represents multi-head self-attention, FFN represents feed-forward network, and LN represents normalization.
- *LSTM:* to capture temporal dependencies in sequential features, the output of the Transformer is fed into a bidirectional LSTM (Long Short-Term Memory [17]), the output can be represented as:

$$H_t^{(l)} = \text{LSTM}(Z_t^{(l)}) \tag{4}$$

where $Z_t^{(l)}$ is the output sequence of the Transformer at time step t, and $H_t^{(l)}$ is the hidden state of the LSTM at time step t.

- *Fully connected:* the output of the LSTM at the last time step T, finally, is mapped to sparsity and computational intensity thresholds through a fully connected layer. The output can be represented as:

$$(\hat{s}_t, \hat{c}_t) = \sigma(\omega \cdot H_T^{(l)} + b) \tag{5}$$

where $\hat{s}_t$ and $\hat{c}_t$ represent the predicted sparsity and computational intensity thresholds, respectively. ω and b are weight and bias parameters, and σ is the activation function (e.g., Sigmoid or ReLU).

3.3 Training

The training of the threshold predictor adopts a supervised learning method. Let the training dataset be $\{(X_i, y_i)\}_{i=1}^N$, where X_i is the input vector, and $y_i = (s_i, c_i)$ are the actual sparsity and computational intensity thresholds. The loss function is defined as

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} \left(|s_i - \hat{s}_i|^2 + |c_i - \hat{c}_i|^2 \right) \tag{6}$$

where N is the number of samples, s_i and c_i represent the true sparsity and computational intensity thresholds.

Ground Truth Acquisition. To obtain the ground truth labels $y_i = (s_i, c_i)$ for supervised training, we conduct a one-time, offline exhaustive search on the target hardware. For each operator type in our dataset, we measure its execution latency across a comprehensive grid of sparsity levels and input sizes (which correlate to computational intensity) on both the CPU and GPU. The true optimal thresholds (s_i, c_i) are identified as the boundary points where the optimal execution device switches (e.g., from CPU to GPU) to minimize latency. Summarily, we collected approximately 2,000 samples from the five DNN models listed in Table 2 on both Jetson Orin Nano and AGX Orin.

4 OPERATOR SCHEDULER

The primary goal of the learning-based operator scheduler in SparOA is to minimize the total inference time by intelligently assigning operators to heterogeneous hardware components. This involves considering multiple factors, including sparsity, computational intensity, memory constraints, and overheads of switching between different processors. We employ the Soft Actor-Critic (SAC) algorithm, a state-of-the-art RL method, to learn the optimal policy for operator scheduling. The algorithm is well-suited for dynamic, high-dimensional search spaces, making it applicable to the complex operator scheduling decisions at stake.

Learning vs. Rules. Note that SparOA does not rely on fixed heuristic rules (e.g., always assign high-intensity operators to GPU). Instead, sparsity (ρ) and computational intensity (I) serve merely as state features input to the SAC agent. The agent autonomously learns the complex, non-linear mapping between these features and the optimal device assignment through trial-and-error, guided by a reward function that penalizes latency. Consequently, the agent retains the flexibility to make counter-intuitive but optimal decisions—for instance, scheduling a high-intensity operator on the CPU if the GPU memory is contended or if the data transfer overhead outweighs the acceleration benefit. This learning-based approach allows SparOA to adapt to specific operator behaviors (e.g., sparse vs. dense kernels) beyond simple intensity metrics.

4.1 Modeling

We model the operator scheduling problem as a Markov decision process (MDP). We defined next its components including the state space, the action space, as well as the reward and transition probabilities.

State space. We model the current state of reasoning process and used that to provide a decision-making basis for agents in RL. The state space includes the following features:

- *Sparsity:* operators with high sparsity can usually reduce the amount of computation by skipping zero-value operations.
- *Computational intensity:* indicates the number of FLOPs required per unit of input data.
- *Input size:* the size of the input tensor for the current operator.
- *Output size:* the size of the output tensor for the current operator, which is used to evaluate the computational requirements of subsequent operators.
- *Hardware state:* includes GPU memory usage, CPU load level, and switching overhead between the heterogeneous processors.

Therefore, the state space $\mathcal{S}$ in RL is defined as

$$\mathcal{S} = \{\rho, I, N_{in}, N_{out}, M_{gpu}, M_{cpu}, O_{switch}\} \tag{7}$$

where ρ is the sparsity, I is the computational intensity, N_{in} and N_{out} are the input and output sizes respectively. M_{gpu} is the GPU memory usage, M_{cpu} is the CPU load level, and O_{switch} is the heterogeneous processor switching overhead.

Action space. The action space in SparOA is a continuous range that represents the resource allocation ratio between the CPU and the GPU. Specifically, the action $\mathcal{A} \in [0, 1]$ represents the proportion allocated to the GPU, whereas the remainder is allocated to the CPU. For instance, $\mathcal{A} = 1$ means full use of the GPU and $\mathcal{A} = 0$ means full use of the CPU.

The action space can be formalized as follows:

$$a \in \mathcal{A}, \mathcal{A} \in [0, 1] \tag{8}$$

Reward. We formulate the as an evaluation of the quality of the agent's behavior in RL. In SparOA, the reward takes into account the following factors:

- *Inference latency:* encourages reducing inference time.
- *Memory usage:* avoids exceeding memory limits.
- *Switching overhead:* reduces processor switching.

The reward is formalized as

$$r = -(\lambda_1 \cdot L + \lambda_2 \cdot (M_{gpu} + M_{cpu}) + \lambda_3 \cdot O_{switch}) \tag{9}$$

where L is the inference latency and λ is the weight coefficient used to balance the importance of different goals.

Transition probabilities. It mathematically formalizes how the environment's state evolves when an action is taken. Specifically, it defines the probability distribution $P(s'|s, a)$, representing the likelihood of transitioning to state s' from state s after applying action a. We derive this probability distribution by combining two key factors:

- *DNN architectural constraints:* the dependencies between operators (e.g., layer execution order) and their inherent computational properties (e.g., sparsity and computational intensity) impose deterministic transitions.
- *Hardware dynamic:* probabilistic variations arise from hardware-specific factors like GPU memory contention (e.g., when multiple operators compete for limited memory bandwidth) or CPU core availability (e.g., due to background processes).

Consider a scenario where the current state s includes a convolutional operator with sparsity $\rho = 0.2$, computational intensity $I = 1.8 \times 10^{11}$ FLOPs, and GPU memory usage $M_{\text{gpu}} = 75\%$. The agent selects action $a = 1$ (allocate to GPU). The transition to s' is influenced by: 1) The convolution's output tensor size N_{out} updates the subsequent operator's input size. 2) GPU memory usage increases by ΔM_{gpu}, calculated from the operator's weight and activation storage requirements.

4.2 SAC-based Operator Scheduling

SAC is an RL algorithm based on the maximum entropy framework, which achieves a balance between exploration and exploitation by introducing an entropy regularization term in the policy optimization process. The core idea is to maximize not only the expected return but also the entropy of the policy, thereby encouraging the agent to maintain a certain degree of randomness when making decisions. This feature makes it particularly suitable for solving complex continuous action space problems, such as tasks that dynamically allocate computing resources [45], that is, precisely our case. The key components of SAC include:

- *Policy network:* $\pi_\theta(a|s)$ maps states s to actions a. It is parameterized by θ and outputs a probability distribution over continuous actions.
- *Q-networks:* Q_{ϕ_1} and Q_{ϕ_2} estimate the expected return for taking action a in state s. These two networks aim to mitigate overestimation, which can be updated using the Bellman equation:

$$Q(s, a) \leftarrow r + \gamma \mathbb{E}_{s' \sim p, a' \sim \pi}[Q(s', a') - \alpha \log \pi(a'|s')] \tag{10}$$

  where γ is the discount factor, and α is the temperature parameter controlling the trade-off between exploration and exploitation.
- *Entropy regularization:* to maximize the expected return while maintaining high entropy in the policy:

$$J(\pi) = \mathbb{E}_{s \sim p, a \sim \pi}[Q(s, a)] + \alpha \mathcal{H}(\pi(\cdot|s)) \tag{11}$$

  where $\mathcal{H}(\pi(\cdot|s)) = -\mathbb{E}_{a_t \sim \pi}[\log \pi(a_t, s_t)]$ is the entropy of the policy.
- *Target networks:* to stabilize training, the parameters of the target networks are updated slowly using a moving average.

$$\phi_i' \leftarrow \tau \phi_i + (1 - \tau)\phi_i' \tag{12}$$

  where τ is the smoothing coefficient.
- *Temperature parameter α:* to dynamically adjust the strength of entropy regularization, SAC introduces a learnable temperature parameter α, and the optimization objective becomes

$$J(\alpha) = \mathbb{E}_{a \sim \pi}[-\alpha \log \pi(a_t, s_t) + \bar{\mathcal{H}}] \tag{13}$$

  where $\bar{\mathcal{H}}$ is the target entropy, usually set to the negative action space dimension: $\bar{\mathcal{H}} = -dim(\mathcal{A})$.

Algorithm 1 illustrates the operator scheduling optimization process. The algorithm operates in episodes, where it samples actions from the policy network to determine CPU-GPU resource ratios and executes computations accordingly (*line 6∼12*), either on both devices with weighted averaging (*line 13*), solely on the CPU (*line 15*), or solely on the GPU (*line 17*). Transitions, including states, actions, rewards, and next states, are stored in the replay buffer (*line 19*), while total inference latency is updated based on rewards (*line 21*). After each episode, gradient updates improve the Q-functions and policy using Bellman backup, maximum entropy objectives, and dynamic temperature adjustment (*line 23∼30*). By dynamically adjusting the temperature parameter α, the algorithm maintains high exploration levels during the early stages of training and gradually converges to the optimal policy as training progresses.

Algorithm 1 SAC-based operator scheduling.

Input: DNN model M, input tensor T, environment E
Output: Operator Scheduling Strategy P

Initialize policy network $\pi_\theta(a|s)$, Q-networks $Q_{\phi_1}(s,a), Q_{\phi_2}(s,a)$, target networks $Q_{\phi_1}^{\text{target}}, Q_{\phi_2}^{\text{target}}$
Initialize replay buffer $\mathcal{D}$, temperature parameter α
$L \leftarrow 0$ ▷ Initialize inference latency
for episode = $1, 2, \ldots$ **do**
  $S \leftarrow E$.reset()
  for $t = 1, 2, \ldots$ **do**
    Sample action $\mathcal{A} \sim \pi_\theta(\cdot|\mathcal{S})$
    Execute action $\mathcal{A}$ using SAC policy $\pi_\theta(\mathcal{A}|\mathcal{S})$
    ξ = map_action_to_ratios($\mathcal{A}$)
    if $\xi \in (0, 1)$ **then**
      $P_{\text{cpu}} \leftarrow$ execute_on_cpu(M, T)
      $P_{\text{gpu}} \leftarrow$ execute_on_gpu(M, T)
      $P \leftarrow \xi \cdot P_{\text{cpu}} + (1 - \xi) \cdot P_{\text{gpu}}$
    else if $\xi = 0$ **then**
      $P \leftarrow$ execute_on_cpu(M, T)
    else
      $P \leftarrow$ execute_on_gpu(M, T)
    end if
    $\mathcal{D} \leftarrow (\mathcal{S}, \mathcal{A}, r, \mathcal{S}', \text{done})$
    $\mathcal{S}', r, \text{done}, \_ \leftarrow E.\text{step}(\mathcal{A})$
    $L \leftarrow L - r$
  end for
  for gradient step $g = 1, 2, \ldots$ **do**
    Sample mini-batch $B = \{(s, a, r, s', \text{done})\}$ from $\mathcal{D}$
    Compute target Q-values:
    Update $Q_\phi(\mathcal{S}, \mathcal{A})$ using Bellman backup
    Update $\pi_\theta(\mathcal{A}|\mathcal{S})$ via maximum entropy objective
    Update α dynamically
    Perform soft updates for target networks
  end for
end for

5 HYBRID INFERENCE ENGINE

The online hybrid inference engine coordinates the execution of operators on the CPU and GPU according to the scheduling policy generated by the operator scheduler. Traditional hybrid inference methods usually face challenges such as processor switching overhead and load imbalance. On the one hand, frequent data transfers between the CPU and GPU introduce significant latency. On the other hand, static resource allocation strategies, such as fixed batch sizes, cannot adapt to the dynamic characteristics of different operators and may cause memory overflows.

We employ asynchronous executions and dynamic batching to further improve inference efficiency while minimizing data transfer overheads and performance losses caused by unbalanced resource allocation. Asynchronous execution parallelizes task scheduling through CUDA streams, reducing the overhead of device switching and data transmission. Dynamic batching optimization dynamically adjusts the batch size according to input data characteristics and hardware constraints to improve resource utilization.

5.1 CPU-GPU Hybrid Execution

Data transfer. To minimize the extra cost of collaboration, particularly the latency induced by CPU-GPU data movement and synchronization—we implement a highly optimized data transmission pipeline. We utilize pinned host memory (i.e., page-locked memory) rather than standard pageable memory for all CPU-resident tensors involved in co-execution. Pinned memory enables the DMA (i.e., direct memory access) controller to transfer data directly between system RAM and GPU VRAM without CPU intervention, significantly maximizing bandwidth utilization.

Furthermore, we leverage CUDA Streams to strictly enforce asynchronous execution. Specifically, we issue asynchronous memory copy commands using `cudaMemcpyAsync` in non-default streams. This allows the GPU to execute inference kernels for the current batch while simultaneously fetching the next batch's data from the CPU, or vice versa, effectively hiding the transmission overhead behind computation. We explicitly quantify this collaboration cost by profiling the duration of these memory operations, and evaluated it in Sec. 6.3.

Result aggregation. After an operator completes the execution on the CPU or GPU, we aggregate the results using the GPU or CPU executors (i.e., threads running on the CPU). We employ a weighted average aggregation strategy with weights dynamically generated by the SAC-based scheduling algorithm. We formalize the aggregation result as:

$$P = \xi \cdot P_{\text{cpu}} + (1 - \xi) \cdot P_{\text{gpu}} \tag{14}$$

where P_{cpu} and P_{gpu} are the outputs of the CPU and GPU respectively, and ξ is the allocation ratios.

The engine synchronizes the CUDA stream via `torch.cuda.synchronize` before aggregation to ensure the correctness of result aggregation. Additionally, the engine caches the intermediate results of the CPU and GPU in shared memory for direct access by subsequent operations, thereby reducing the overhead of repeated calculations and data transfers. Note that the operators preloaded in GPU are processed in situ, while the CPU calculates and transfers results for its operators to the GPU for aggregation [44].

Figure 4 illustrates an example of how SparOA co-executes operators across CPU and GPU. It classifies operators based on the proposed offline operator scheduler, assigning operators with high sparsity and low computational intensity (e.g., operators 3, 5, 6, 8) to CPU memory and others to GPU memory. Both the CPU and GPU concurrently process their assigned operators. The GPU computes operators 1, 2, 4, 7, and 8, leveraging its parallel processing capabilities, while the CPU simultaneously handles operators 3, 5, 6, 8. Upon completion of all operators on the CPU, its output is transferred to the GPU that performs the final aggregation step,

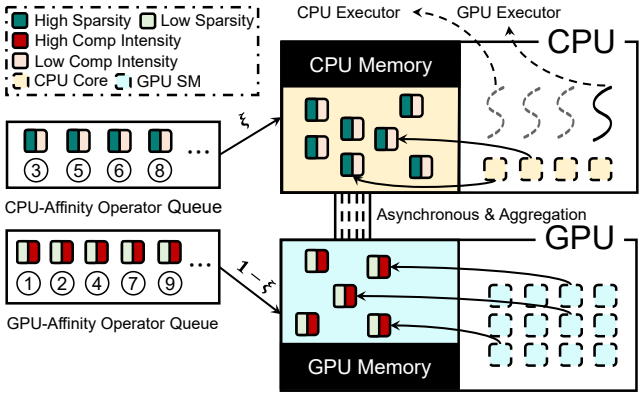


Figure 4: Example showing how SparOA calculates different operators.

combining the results from all activated operators using a weighted average strategy.

5.2 Dynamic Batching

Dynamic batching aims to dynamically adjust the batch size based on input data characteristics and hardware constraints, thereby improving throughput while retaining inference accuracy. First, we must ensure that the batch size does not exceed available memory and meets real-time constraints; second, we must adjust the batch size according to operator sparsity and computational intensity to balance computation and data transfer overhead.

To this end, we use a gradient descent-based dynamic batching optimization algorithm, combining hardware-aware and input data-driven partitioning strategies to dynamically adjust the batch size. As shown in Algorithm 2, the gradient is calculated based on the inference latency and memory usage of the current batch size. The batch size is then gradually adjusted along the gradient direction until a (possibly local) optimal value is found (*line 6*).

During this process, the algorithm continuously checks whether the batch size satisfies hardware and real-time constraints (*line 7∼14*). If the current batch size exceeds available memory, it is halved to avoid overflow. Additionally, the algorithm partitions the batch based on the sparsity and computational intensity characteristics of the input data. For instance, for highly sparse inputs, the batch size can be increased to leverage the parallel computing capabilities of the GPU; for inputs with high computational intensity, we reduce the batch size to shrink inference times.

Algorithm 2 Dynamic batching optimization.

Input: Initial batch size B_0, input tensor T
Output: Optimized batch size B^*
1: Initialize $B \leftarrow B_0$
2: Evaluate initial latency $L(B)$ and memory usage $M(B)$
3: Set learning rate η, convergence threshold ϵ, and maximum available memory $M_{\max}$
4: **while** $|L(B) - L(B_{\text{prev}})| > \epsilon$ **do**
5: $\quad \nabla_B L(B) = \frac{\partial L(B)}{\partial B}$
6: $\quad B \leftarrow B - \eta \cdot \nabla_B L(B)$
7: $\quad$ **if** $M(B) > M_{\max}$ and $L(B) > T_{\text{real-time}}$ **then**
8: $\quad\quad B \leftarrow B/2$
9: $\quad$ **end if**
10: $\quad$ **if** sparsity(T) > threshold **then**
11: $\quad\quad B \leftarrow \min(2B, M_{\max})$
12: $\quad$ **else if** computational_intensity(T) > threshold **then**
13: $\quad\quad B \leftarrow B/2$
14: $\quad$ **end if**
15: $\quad L(B_{\text{prev}}) \leftarrow L(B)$
16: **end while**

6 EVALUATION

6.1 Prototype

We implemented SparOA in 3,200 lines of Python. SparOA supports common DNN operators including convolution, fully connected, activation, normalization, pooling, and attention. The RL-based operator scheduler in Algorithm 1 uses SAC from `stable baselines3` [40]. Training data is collected from runtime characteristics (sparsity, computational intensity, input/output size, and hardware utilization).

For the threshold predictor, we employ a Transformer-LSTM architecture with 128-dimensional hidden layers and 4 attention heads. The predictor takes sparsity, computational intensity, channels, height, and width as inputs, trained with Adam optimizer (learning rate 10^{-4}) for 100 epochs. We use 80% of samples for training and 20% for testing.

6.2 Experiment Setup

We execute experiments on two edge devices configurations as detailed in Table 1, representing both high-end and low-end hardware scenarios.

Models. We select five DNN models that are widely deployed in real applications, including ResNet-18 [15], MobileNet-v3-small [18], MobileNet-v2 [41], ViT-B16 [8] and Swin Transformer [32]. Table 2 lists the parameters, computational intensity, and number of operators. We run each DNN model 10 times and compute average results.

Datasets. The input data are derived from ImageNet-2012 [23] and MS COCO-2014 [28]. The ImageNet-2012 dataset is used to evaluate the inference performance of CNN-based models, such as ResNet-18 and MobileNet. The MS COCO-2014 dataset is suitable for testing the performance of attention-based DNN models, such as Vision Transformer and Swin Transformer. The two datasets comprehensively evaluate the hybrid inference efficiency of SparOA under different architectures and input characteristics.

Baselines. We compare SparOA with:

Edge GPU	AI Performance	DRAM	CPU	GPU	Power
Jetson Orin Nano	67TOPS (INT8)	8GB 102GB/s	6×ARM Cortex-A78AE v8.2@1.7GHz	1024×Ampere@1GHz	7-25W
Jetson AGX Orin	275TOPS (INT8)	64GB 204.8GB/s	12×ARM Cortex-A78AE v8.2@2.2GHz	2048×Ampere@1.3GHz	15-60W

Table 1: Configurations of edge devices used in the evaluation.

Model	Parameters	Computational Intensity	Number of Operators
ResNet-18	11.7M	1.8GFLOPs	53
MobileNet-v3-small	3.5M	0.3GFLOPs	112
MobileNet-v2	2.5M	0.05GFLOPs	121
ViT-B16	86M	17.6GFLOPs	65
Swin Transformer	28M	4.5GFLOPs	125

Table 2: Configurations of DNN models.

- CPU-Only: the whole model is executed on the CPU.
- GPU-Only (PyTorch) [37]: a deep learning framework with dynamic graph. PyTorch dispatches operators one by one sequentially.
- TensorFlow [1]: a deep learning framework with static graphs. The scheduler in TensorFlow also dispatches operators sequentially.
- TensorRT [36]: schedules the computation graph after the automatic fine-tuning of the kernel to the multi-stream in the GPU to execution in parallel.
- TVM [4]: a machine learning compiler framework that utilizes AutoTVM [5] and AutoScheduler [56] to generate efficient schedules.
- IOS [7]: an inter-operator scheduler, which utilizes operator fusion and inter-operator parallelism to accelerate inference.
- POS [55]: a learning-based operator scheduler, which utilizes operator fusion, subgraph reuse, inter- and intra-operator parallelism to accelerate inference.
- CoDL [21]: a state-of-the-art CPU-GPU collaborative inference framework, accelerates inference through hybrid-type-friendly data sharing.
- SparOA w/o RL: a variant of SparOA without operator scheduling strategy.
- SparOA with Greedy: a variant of SparOA based on a greedy strategy for operator scheduling.
- SparOA with DP: a variant of SparOA based on dynamic programming for operator scheduling.

6.3 Overall Performance

Figure 5 reports the comparison of inference latency. For AGX Orin, SparOA achieves up to 50.7× speedup over baselines. The most significant improvement is against CPU-Only (e.g., 50.7× on MobileNet-v3). Against DNN compilation frameworks (e.g., TensorRT, TVM, IOS, POS) and the SOTA co-execution baseline (e.g., CoDL), SparOA achieves 1.22×∼1.31× speedup. These SOTA baselines generate a fixed execution plan. In contrast, the SAC-based scheduler in SparOA can dynamically adapt to real-time hardware conditions, resulting in lower latency.

Futhermore, SAC-based RL scheduling further improves performance by 1.17×∼1.42× over non-RL variants (Greedy, DP). For compute-intensive models (ViT-B16, Swin Transformer), SparOA reduces latency by 1.13×∼1.31× over compilation frameworks by dynamically offloading high-computational intensity operators to GPU while executing sparse activation layers on CPU. For Orin Nano, SparOA achieves 1.24×∼11.43× acceleration, demonstrating its scalability across resource-constrained environments.

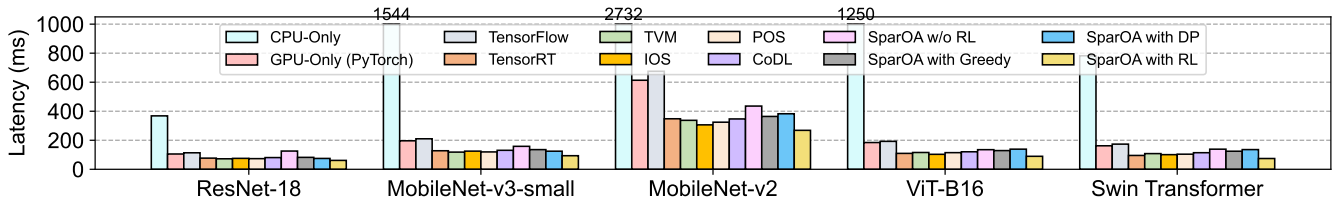
As shown in Figure 6, SparOA's RL scheduler dynamically adjusts operator allocation based on real-time hardware states, increasing GPU operator load share to 72.6% compared to 55.6% (Greedy) and 60.8% (DP). In Figure 7, this adaptive scheduling significantly reduces data transfer latency by 14.1%∼20.8% compared to static scheduling (i.e., SparOA without RL), while dynamic batching optimizes batch sizes for different operator types to maximize resource utilization.

6.4 Threshold Predictor

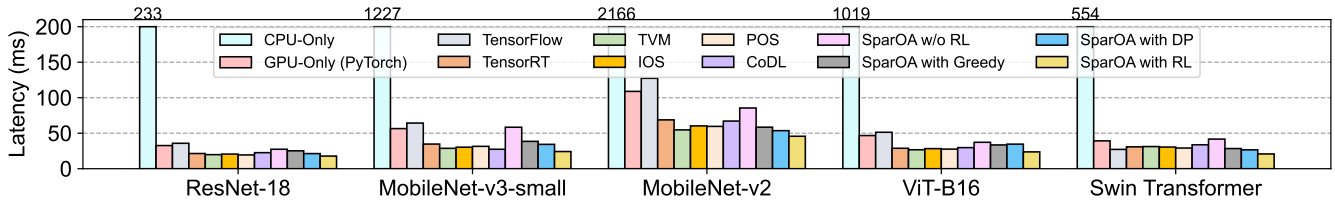
We test the threshold predictor's performance through rigorous benchmarking against linear regression (LR) and CNN-based predictors, as summarized in Table 3. Our Transformer-LSTM threshold predictor achieves 92.3% accuracy on Sparsity and 90.6% on Computational Intensity (±10% tolerance), significantly outperforming CNN (36.2% and 38.5%) and linear models (23.7% and 20.4%). It combines Transformer and LSTM to capture both global dependencies and temporal correlations across operators. For Swin Transformer's attention layers, it reduces threshold estimation errors from 13.8% to 2.1%. Despite this high accuracy, our predictor remains lightweight (∼4MB), enabling effective deployment on resource-constrained devices.

6.5 Hybrid Inference Engine

Figure 8 illustrates the end-to-end batching overhead of SparOA on Orin Nano and AGX Orin. Using CUDA streams, it reduces switching overhead by 52% and achieves 78% overlap between data transfer and computation. The gradient-based batching algorithm dynamically adjusts batch sizes (1-512), reducing batching overhead to 2.3%∼8.6% compared to 15.4%∼28.7% in static frameworks. For ViT-B16, it selects

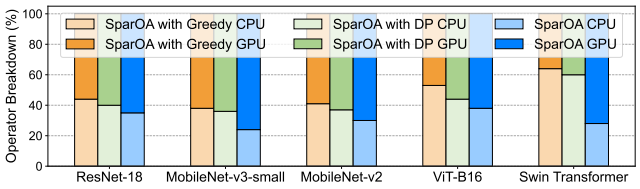


(a) NVIDIA Jetson Orin Nano

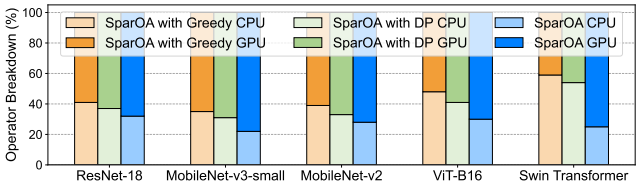


(b) NVIDIA Jetson AGX Orin

Figure 5: Inference latency comparison of SparOA against the baselines, depending on target device.



(a) NVIDIA Jetson Orin Nano

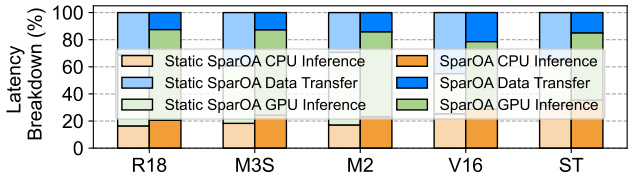


(b) NVIDIA Jetson AGX Orin

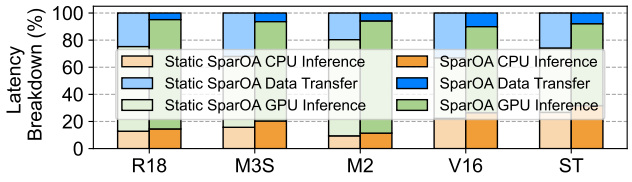
Figure 6: Operator distribution on CPU and GPU during inference.

Predictor	Metric		Model Size
	Sparsity	Computational Intensity	
LR	23.7%	20.4%	~7B
CNN	36.2%	38.5%	~0.5MB
Ours	**92.3%**	**90.6%**	**~4MB**

Table 3: ±10% prediction accuracy and model size of the threshold predictors.



(a) NVIDIA Jetson Orin Nano



(b) NVIDIA Jetson AGX Orin

Figure 7: Latency breakdown for static SparOA (i.e., without RL) and SparOA during inference.

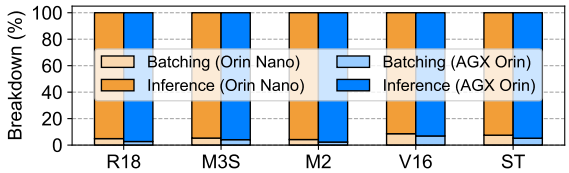
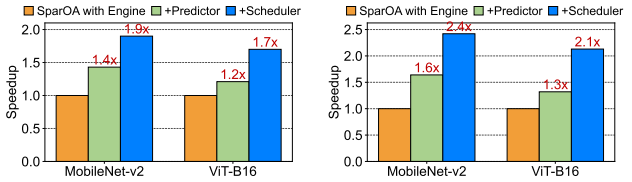


Figure 8: End-to-end batching overhead of SparOA. The Y-axis displays the percentage breakdown between batching overhead and inference time.

batch size 32 for attention operators and 8 for activation operators to balance GPU occupancy and memory usage.

The engine employs a weighted averaging strategy (e.g., $P = 0.7 \cdot P_{\text{gpu}} + 0.3 \cdot P_{\text{cpu}}$) that reduces aggregation errors by 14%. SparOA achieves 89% GPU utilization for ViT-B16 (vs. 63% in TensorRT) and reduces memory usage by 37% for MobileNet-v2, enabling efficient deployment on resource-constrained edge devices.



(a) NVIDIA Jetson Orin Nano (b) NVIDIA Jetson AGX Orin

Figure 9: Inference performance breakdown.

6.6 Ablation Study

Figure 9 quantifies the contribution of each SparOA component. We evaluate MobileNet-v2 (CNN) and ViT-B16 (Transformer) on both edge devices. The baseline hybrid engine (SparOA w/o Predictor/Scheduler) is normalized.

Threshold Predictor (+Predictor) dynamically adjusts sparsity/computational intensity thresholds, reducing redundant computations (e.g., pruning low-contribution kernels in MobileNet-v2). Gains are higher for MobileNet-v2 (1.4×∼1.6× speedup) than ViT-B16 due to the latter's sparse attention variability. Learning-based Scheduler (+Scheduler) prioritizes GPU offloading for compute-intensive operators (e.g., depthwise convolutions, matrix multiplications), further improving speedups (MobileNet-v2: 1.9×∼2.4×; ViT-B16: 1.7×∼2.1×). Hardware constraints limit gains on Orin Nano, where memory bottlenecks restrict batch optimization.

Overall, MobileNet-v2 benefits most from sparsity optimization and GPU offloading. ViT-B16 relies on scheduler-driven resource allocation for compute/memory balancing. High-end devices unlock parallelism, and low-end devices face memory limits.

6.7 Convergence Time

Figure 10 compares convergence times of scheduling algorithms. Greedy converges rapidly (0.04∼0.24*s*) but ignores hardware states, resulting in 22% higher latency than SAC. DP requires excessive time (39∼415*s*) due to exhaustive search, yet yields suboptimal strategies (e.g., 63ms vs SAC's 48ms for MobileNet-v2).

SAC achieves optimal performance with 33∼46*s* convergence time. Though slower to converge than Greedy, its dynamic adaptation to hardware states delivers superior runtime efficiency. Notably, SAC demonstrates sublinear convergence growth with model complexity, enabled by its parallel policy evaluation mechanism. This trade-off proves valuable as the convergence overhead is amortized over numerous inference executions.

6.8 System Overhead

6.8.1 Power/Energy Consumption. As shown in Figure 11(a), SparOA's power consumption is higher than single-processor baselines. This is a necessary trade-off for activating both

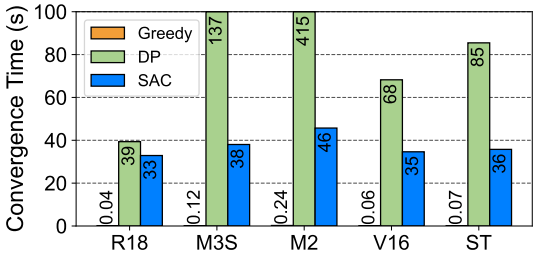
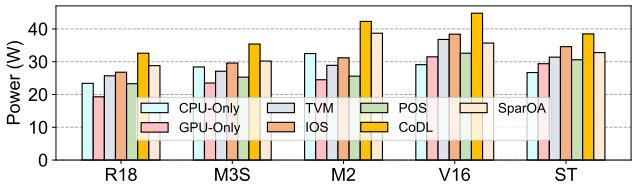
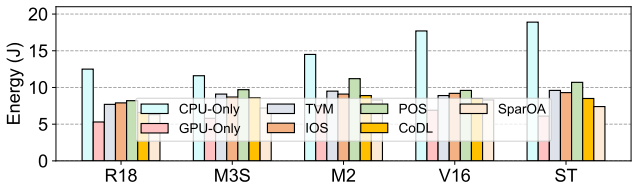


Figure 10: Convergence time of different scheduling algorithms on NVIDIA Jetson AGX Orin.



(a) Power consumption per inference



(b) Energy consumption per inference

Figure 11: Power/energy consumption measurements on NVIDIA Jetson AGX Orin.

CPU and GPU resources. Also, SparOA consumes slightly more power than compiler frameworks such as TVM (e.g., 34% higher) and IOS (e.g., 24% higher). Notably, SparOA's power draw is consistently more efficient than CoDL. In Figure 11(b), SparOA achieves the lowest energy-per-inference. This superior energy efficiency, resulting from a controlled power trade-off, leads to a significant reduction in latency.

6.8.2 Memory Usage. In Figure 12, SparOA employs a sharded storage strategy for co-execution (i.e., CPU stores sparse attention heads while GPU stores dense weights). This results in an average of 23.1% increase in memory overhead compared to GPU-Only. While SparOA requires more memory than single-processor methods, its memory footprint is comparable to other baselines (e.g., IOS and POS), and is lower than CoDL. This modest memory overhead is an explicit trade-off for enabling hybrid scheduling and is justified by the substantial improvements in inference latency.

7 RELATED WORK

Optimization of DNN Inference. Model adaptations, including model compression [14], pruning, and quantization [16,

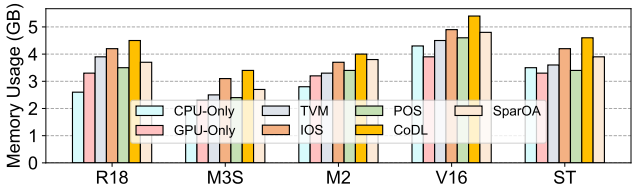


Figure 12: Memory usage measurements, using the NVIDIA Jetson AGX Orin as the test device.

27, 29, 47], reduce computational complexity but often sacrifice accuracy and fail to exploit heterogeneous hardware. DNN compilers [33, 43, 53, 58] like TVM [4], Ansor [56], TensorRT [36], IOS [7], and POS [55] generate optimized, hardware-specific code, but typically target a single processor. Other works address orthogonal challenges: AdaptiveNet [51] tackles device diversity by selecting an optimal subnet from a supernet post-deployment, while FlexNN [24] manages memory constraints by slicing models that exceed RAM. SparOA is distinct from these approaches, we optimize a single, fixed model assumed to be in memory, focusing on hybrid operator scheduling across the CPU and GPU, rather than model selection or I/O management.

Co-execution of heterogeneous processors. Edge devices feature diverse processors [21], which several frameworks aim to exploit. CoDL [21] optimizes inference latency through dynamic operator scheduling based on processor affinity. Other systems partition tasks for multi-DNN inference (e.g., Band [20] and BlastNet [30]), automatically branch models (NN-Stretch [49]), or split workloads for specific tasks and hardware [2, 19, 48]. However, these methods, including CoDL, primarily rely on static processor characteristics for scheduling. They largely overlook how intrinsic model properties, such as dynamic sparsity and computational intensity, affect performance. SparOA addresses this limitation by incorporating both metrics to achieve a more fine-grained operator-processor matching.

DNN performance prediction. Existing performance predictors range from kernel-level mathematical models [25, 26] (e.g., nn-Meter [54], nnPerf [6]) to ML-based predictors that learn from historical data (e.g., MAPLE-Edge [35], LitePred [12]). These methods often ignore heterogeneous processor characteristics or require extensive data, focusing on predicting latency for a single platform. SparOA's threshold predictor is fundamentally different. It is not a general-purpose latency predictor; rather, it is a specialized, lightweight model designed to capture the non-linear relationship between sparsity and computational intensity to accurately determine the optimal scheduling thresholds for hybrid CPU-GPU execution.

8 DISCUSSION

Hardware Heterogeneity. Extending SparOA to support more heterogeneous AI accelerators [20] (e.g., NPUs, TPUs) presents significant challenges. The operator scheduler would need to learn more complex allocation policies across multiple hardware types with vastly different characteristics. One promising approach is that SparOA integrates hierarchical reinforcement learning [34, 39] that first determines broad hardware categories before making finer-grained decisions within the accelerator cluster. This approach would maintain scheduling efficiency while expanding hardware support.

Training Overhead and Model Adaptation. When DNN architectures change significantly, the predictor may require complete retraining. To address this, SparOA can be combined with transfer learning [52, 57], which enables the predictor to quickly adapt to new models with minimal additional training by leveraging knowledge from previously seen architectures. Additionally, meta-learning [11, 50] could further enable SparOA to generalize across model families, reducing the training overhead for novel architectures.

Multi-Task Inference. Edge devices typically involve multiple concurrent tasks with varying service-level objectives (SLOs) [22]. In multi-DNN inference [31, 55], the scheduler in SparOA cannot prioritize critical tasks. To this end, we will extend SparOA with a multi-level scheduler that incorporates task priority metrics and SLOs. Meanwhile, combining SparOA with techniques like knowledge distillation [38] could create lightweight versions of DNNs for non-critical tasks, while reserving full resources for high-priority models. Furthermore, integrating with edge orchestration systems like Kubernetes (k8s) [3] could enable coordinated resource management across multiple applications running on the same edge device.

9 CONCLUSION

This work presents SparOA, a framework that redefines hybrid inference on edge devices by treating sparsity and computational intensity as orthogonal, joint scheduling metrics. SparOA integrates a lightweight predictor with a dynamic RL-based scheduler, enabling it to adapt to real-time hardware fluctuations. Extensive evaluations on NVIDIA Jetson platforms demonstrate that SparOA outperforms SOTA compilers and co-execution baselines in terms of inference latency and energy efficiency.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.

[2] Hao Bao, Zhi Zhou, Fei Xu, and Xu Chen. 2024. COUPLE: Orchestrating Video Analytics on Heterogeneous Mobile Processors. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 1561–1574.

[3] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. In *Proceedings of the 2016 USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*. 13–25. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/burns

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[5] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to optimize tensor programs. *Advances in Neural Information Processing Systems* 31 (2018).

[6] Haolin Chu, Xiaolong Zheng, Liang Liu, and Huadong Ma. 2023. nnPerf: Demystifying DNN runtime inference latency on mobile platforms. In *Proceedings of the 21st ACM Conference on Embedded Networked Sensor Systems*. 125–137.

[7] Yaoyao Ding, Ligeng Zhu, Zhihao Jia, Gennady Pekhimenko, and Song Han. 2021. Ios: Inter-operator scheduler for cnn acceleration. *Proceedings of Machine Learning and Systems* 3 (2021), 167–180.

[8] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929* (2020).

[9] Hongxiang Fan, Stylianos I Venieris, Alexandros Kouris, and Nicholas Lane. 2023. Sparse-dysta: Sparsity-aware dynamic and static scheduling for sparse multi-dnn workloads. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 353–366.

[10] Weiwei Fang, Wenyuan Xu, Chongchong Yu, and Neal N Xiong. 2023. Joint architecture design and workload partitioning for dnn inference on industrial iot clusters. *ACM Transactions on Internet Technology* 23, 1 (2023), 1–21.

[11] Bin Feng, Zequn Liu, Nanlan Huang, Zhiping Xiao, Haomiao Zhang, Srbuhi Mirzoyan, Hanwen Xu, Jiaran Hao, Yinghui Xu, Ming Zhang, et al. 2024. A bioactivity foundation model using pairwise meta-learning. *Nature Machine Intelligence* 6, 8 (2024), 962–974.

[12] Chengquan Feng, Li Lyna Zhang, Yuanchi Liu, Jiahang Xu, Chengruidong Zhang, Zhiyuan Wang, Ting Cao, Mao Yang, and Haisheng Tan. 2024. {LitePred}: Transferable and Scalable Latency Prediction for {Hardware-Aware} Neural Architecture Search. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1463–1477.

[13] Cheng Gongye, Hongjia Li, Xiang Zhang, Majid Sabbagh, Geng Yuan, Xue Lin, Thomas Wahl, and Yunsi Fei. 2020. New passive and active attacks on deep neural networks in medical applications. In *Proceedings of the 39th international conference on computer-aided design*. 1–9.

[14] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[16] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. 2018. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European conference on computer vision (ECCV)*. 784–800.

[17] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[18] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. 2019. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF international conference on computer vision*. 1314–1324.

[19] Kai Huang, Xiangyu Yin, Tao Gu, and Wei Gao. 2024. Perceptual-Centric Image Super-Resolution using Heterogeneous Processors on Mobile Devices. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*. 1361–1376.

[20] Joo Seong Jeong, Jingyu Lee, Donghyun Kim, Changmin Jeon, Changjin Jeong, Youngki Lee, and Byung-Gon Chun. 2022. Band: coordinated multi-dnn inference on heterogeneous mobile processors. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*. 235–247.

[21] Fucheng Jia, Deyu Zhang, Ting Cao, Shiqi Jiang, Yunxin Liu, Ju Ren, and Yaoxue Zhang. 2022. CoDL: efficient CPU-GPU co-execution for deep learning inference on mobile devices.. In *MobiSys*, Vol. 22. 209–221.

[22] Seah Kim, Hyoukjun Kwon, Jinook Song, Jihyuck Jo, Yu-Hsin Chen, Liangzhen Lai, and Vikas Chandra. 2023. Dream: A dynamic scheduler for dynamic real-time multi-model ml workloads. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*. 73–86.

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2017. Imagenet classification with deep convolutional neural networks. *Commun. ACM* 60, 6 (2017), 84–90.

[24] Xiangyu Li, Yuanchun Li, Yuanzhe Li, Ting Cao, and Yunxin Liu. 2024. Flexnn: Efficient and adaptive dnn inference on memory-constrained edge devices. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*. 709–723.

[25] Ying Li, Yifan Sun, and Adwait Jog. 2023. Path forward beyond simulators: Fast and accurate gpu execution time prediction for dnn workloads. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 380–394.

[26] Zhuojin Li, Marco Paolieri, and Leana Golubchik. 2023. Predicting inference latency of neural architectures on mobile devices. In *Proceedings of the 2023 ACM/SPEC international conference on performance engineering*. 99–112.

[27] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2024. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of Machine Learning and Systems* 6 (2024), 87–100.

[28] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. 2014. Microsoft coco: Common objects in context. In *Computer vision–ECCV 2014: 13th European conference, zurich, Switzerland, September 6-12, 2014, proceedings, part v 13*. Springer, 740–755.

[29] Yujun Lin, Haotian Tang, Shang Yang, Zhekai Zhang, Guangxuan Xiao, Chuang Gan, and Song Han. 2024. Qserve: W4a8kv4 quantization and system co-design for efficient llm serving. *arXiv preprint arXiv:2405.04532* (2024).

[30] Neiwen Ling, Xuan Huang, Zhihe Zhao, Nan Guan, Zhenyu Yan, and Guoliang Xing. 2022. Blastnet: Exploiting duo-blocks for cross-processor real-time dnn inference. In *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems*. 91–105.

[31] Zihan Liu, Jingwen Leng, Zhihui Zhang, Quan Chen, Chao Li, and Minyi Guo. 2022. VELTAIR: towards high-performance multi-tenant deep learning services via adaptive compilation and scheduling. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 388–401.

[32] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. 2021. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF international conference on computer vision*. 10012–10022.

[33] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling holistic deep learning compiler optimizations with {rTasks}. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 881–897.

[34] Ofir Nachum, Shixiang Shane Gu, Honglak Lee, and Sergey Levine. 2018. Data-efficient hierarchical reinforcement learning. *Advances in neural information processing systems* 31 (2018).

[35] Saeejith Nair, Saad Abbasi, Alexander Wong, and Mohammad Javad Shafiee. 2022. Maple-edge: A runtime latency predictor for edge devices. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 3660–3668.

[36] NVIDIA Corporation. 2025. *NVIDIA TensorRT*. https://developer.nvidia.com/tensorrt Accessed on 2025-03-26, Version 8.2.

[37] A Paszke. 2019. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703* (2019).

[38] Gilles Puy, Spyros Gidaris, Alexandre Boulch, Oriane Siméoni, Corentin Sautier, Patrick Pérez, Andrei Bursuc, and Renaud Marlet. 2024. Three pillars improving vision foundation model distillation for lidar. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 21519–21529.

[39] Molei Qin, Shuo Sun, Wentao Zhang, Haochong Xia, Xinrun Wang, and Bo An. 2024. Earnhft: Efficient hierarchical reinforcement learning for high frequency trading. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 14669–14676.

[40] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. 2021. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of machine learning research* 22, 268 (2021), 1–8.

[41] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4510–4520.

[42] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge computing: Vision and challenges. *IEEE internet of things journal* 3, 5 (2016), 637–646.

[43] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. 2023. Welder: Scheduling deep learning memory access via tile-graph. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 701–718.

[44] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. 2024. Powerinfer: Fast large language model serving with a consumer-grade gpu. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 590–606.

[45] Hsin-Hsuan Sung, Jou-An Chen, Wei Niu, Jiexiong Guan, Bin Ren, and Xipeng Shen. 2023. Decentralized {Application-Level} adaptive scheduling for {Multi-Instance}{DNNs} on open mobile devices. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 865–877.

[46] Haotian Tang, Shang Yang, Zhijian Liu, Ke Hong, Zhongming Yu, Xiuyu Li, Guohao Dai, Yu Wang, and Song Han. 2023. Torchsparse++: Efficient point cloud engine. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 202–209.

[47] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. 2019. Haq: Hardware-aware automated quantization with mixed precision. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 8612–8620.

[48] Qi Wang, Weiwei Fang, Liang Qian, Yanming Chen, and Neal N Xiong. 2024. An intelligent co-scheduling framework for efficient super-resolution on edge platforms with heterogeneous processors. *IEEE Internet of Things Journal* 11, 10 (2024), 17651–17662.

[49] Jianyu Wei, Ting Cao, Shijie Cao, Shiqi Jiang, Shaowei Fu, Mao Yang, Yanyong Zhang, and Yunxin Liu. 2023. Nn-stretch: Automatic neural network branching for parallel inference on heterogeneous multiprocessors. In *Proceedings of the 21st Annual International Conference on Mobile Systems, Applications and Services*. 70–83.

[50] Yongxian Wei, Zixuan Hu, Zhenyi Wang, Li Shen, Chun Yuan, and Dacheng Tao. 2024. Free: Faster and better data-free meta-learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 23273–23282.

[51] Hao Wen, Yuanchun Li, Zunshuai Zhang, Shiqi Jiang, Xiaozhou Ye, Ye Ouyang, Yaqin Zhang, and Yunxin Liu. 2023. Adaptivenet: Post-deployment neural architecture adaptation for diverse edge environments. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*. 1–17.

[52] Yi Xin, Junlong Du, Qiang Wang, Zhiwen Lin, and Ke Yan. 2024. Vmt-adapter: Parameter-efficient transfer learning for multi-task dense scene understanding. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 38. 16085–16093.

[53] Chen Zhang, Lingxiao Ma, Jilong Xue, Yining Shi, Ziming Miao, Fan Yang, Jidong Zhai, Zhi Yang, and Mao Yang. 2023. Cocktailer: Analyzing and optimizing dynamic control flow in deep learning. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 681–699.

[54] Li Lyna Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang, and Yunxin Liu. 2021. Nn-meter: Towards accurate latency prediction of deep-learning model inference on diverse edge devices. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*. 81–93.

[55] Ziyang Zhang, Huan Li, Yang Zhao, Changyao Lin, and Jie Liu. 2023. POS: An operator scheduling framework for multi-model inference on edge intelligent computing. In *Proceedings of the 22nd International Conference on Information Processing in Sensor Networks*. 1–1.

[56] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating {High-Performance} Tensor Programs for Deep Learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 863–879.

[57] Xin Zhou, Dingkang Liang, Wei Xu, Xingkui Zhu, Yihan Xu, Zhikang Zou, and Xiang Bai. 2024. Dynamic adapter meets prompt tuning: Parameter-efficient transfer learning for point cloud analysis. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 14707–14717.

[58] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, et al. 2022. {ROLLER}: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 233–248.

[59] Ligeng Zhu, Lanxiang Hu, Ji Lin, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. 2023. Pockengine: Sparse and efficient fine-tuning in a pocket. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 1381–1394.