POLITECNICO DI TORINO
Repository ISTITUZIONALE

HW-Flow: A Multi-Abstraction Level HW-CNN Codesign Pruning Methodology

Original
HW-Flow: A Multi-Abstraction Level HW-CNN Codesign Pruning Methodology / Rohit Vemparala, Manoj; Fasfous, Nael; Frickenstein, Alexander; Valpreda, Emanuele; Camalleri, Manfredi; Zhao, Qi; Unger, Christian; Shankar Nagaraja, Naveen; Martina, Maurizio; Stechele, Walter. - In: LEIBNIZ TRANSACTIONS ON EMBEDDED SYSTEMS. - ISSN 2199-2002. - ELETTRONICO. - 8:1(2022), pp. 1-30. [10.4230/LITES.8.1.3]

Availability:
This version is available at: 11583/2971412 since: 2022-11-22T15:46:57Z

Publisher:
European Design and Automation Association (EDAA)

Published
DOI:10.4230/LITES.8.1.3

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

02 September 2025

HW-Flow: A Multi-Abstraction Level HW-CNN Codesign Pruning Methodology

Manoj-Rohit Vemparala
BMW Autonomous Driving, Munich, Germany

Nael Fasfous
Technical University of Munich, Munich, Germany

Alexander Frickenstein
BMW Autonomous Driving, Munich, Germany

Emanuele Valpreda
Politecnico di Torino, Turin, Italy

Manfredi Camalleri
BMW Autonomous Driving, Munich, Germany

Qi Zhao
BMW Autonomous Driving, Munich, Germany

Christian Unger
BMW Autonomous Driving, Munich, Germany

Naveen-Shankar Nagaraja
BMW Autonomous Driving, Munich, Germany

Maurizio Martina
Politecnico di Torino, Turin, Italy

Walter Stechele
Technical University of Munich, Munich, Germany

Abstract

Convolutional neural networks (CNNs) have produced unprecedented accuracy for many computer vision problems in the recent past. In power and compute-constrained embedded platforms, deploying modern CNNs can present many challenges. Most CNN architectures do not run in real-time due to the high number of computational operations involved during the inference phase. This emphasizes the role of CNN optimization techniques in early design space exploration. To estimate their efficacy in satisfying the target constraints, existing techniques are either hardware (HW) agnostic, pseudo-HW-aware by considering parameter and operation counts, or HW-aware through inflexible hardware-in-the-loop (HIL) setups. In this work, we introduce HW-Flow, a framework for optimizing and exploring CNN models based on three levels of hardware abstraction: *Coarse*, *Mid* and *Fine*. Through these levels, CNN design and optimization can be iteratively refined towards efficient execution on the target hardware platform. We present HW-Flow in the context of CNN pruning by augmenting a reinforcement learning agent with key metrics to understand the influence of its pruning actions on the inference hardware. With 2× reduction in energy and latency, we prune ResNet56, ResNet50, and DeepLabv3 with minimal accuracy degradation on the CIFAR-10, ImageNet, and CityScapes datasets, respectively.

2012 ACM Subject Classification Computing Methodologies → Artificial intelligence
Keywords and Phrases Convolutional Neural Networks, Optimization, Hardware Modeling, Pruning
Digital Object Identifier 10.4230/LITES.8.1.3
Received 2020-12-15 **Accepted** 2021-09-05 **Published** 2022-11-16
Editor Samarjit Chakraborty and Qing Rao
Special Issue Special Issue on Embedded Systems for Computer Vision

1 Introduction

Convolutional neural networks (CNN) are widely used for solving problems like image classification [13], semantic segmentation [3], object detection [45], complex autonomous driving tasks [2] and medical diagnosis of brain tumors [28]. Having outperformed hard-coded algorithms on challenging benchmarks such as ImageNet [30], CNNs also surpassed human-level accuracy. However, the computational complexity of these networks hampers their application in embedded environments. Most accurate CNN models require up to hundreds of megabytes for parameter storage [24] and billions of multiplications [32]. For instance, a ResNet-152 [13] trained on the ImageNet dataset [30] requires up to 244MB of learned parameters to execute 517 layers, with around 22 billions operations. Compression techniques have become an essential topic of research

© Manoj-Rohit Vemparala, Nael Fasfous, Alexander Frickenstein, Emanuele Valpreda, Manfredi Camalleri, Qi Zhao, Christian Unger, Naveen-Shankar Nagaraja, Maurizio Martina, and Walter Stechele;
licensed under Creative Commons Attribution 4.0 International (CC BY 4.0)

Leibniz Transactions on Embedded Systems, Vol. 8, Issue 1, Article No. 3, pp. 03:1–03:30
LITES Leibniz Transactions on Embedded Systems
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

for finding light-weight architectures capable of efficiently solving various deep learning tasks. Despite the remarkable compression rates of existing pruning methods, conventional approaches are either hardware (HW) agnostic, pseudo-HW-aware by considering proxies, or HW-aware through inflexible hardware-in-the-loop (HIL) setups, which lead to vendor lock-ins.

In embedded applications such as autonomous driving and robotics, the design of neural networks and the target HW accelerator goes hand in hand. During the early development phases, it is likely that the target platform is not fully defined, the HW is not available, or compilers are prone to errors, making a HIL-based approach challenging. Alternatively, proxy metrics, such as parameter (`Param`) and operation (`OP`) counts, offer a detached yet loosely correlated indication of hardware performance [1]. Reliance on proxy metrics oversimplifies the problem at hand and does not always guarantee improvements in energy or latency when deployed on real hardware. Directly choosing a hardware-platform restricts the CNN optimizer, and conversely, directly choosing the compression technique may not match the hardware architectures being designed.

This circumstance calls for a HW-CNN codesign paradigm that guarantees synergies in the process of deploying CNNs to real-world applications. This work aims to estimate the implementation metrics at different abstraction levels through various hardware models and a novel scheduler. This allows for either a top-down or a meet-in-the-middle codesign approach, giving the designer the ability to gradually traverse through the design abstraction levels, while permitting design space exploration and exploitation after each stage of refinement. With this flexibility, the designer can analyze the impact of various CNN pruning configurations and HW specific hyperparameters, without committing to a target hardware platform in the early design phases. This ultimately leads to a platform-aware optimization technique, which improves energy efficiency and/or latency at design time.

We remove the limitations of pure proxy and HIL-based neural network pruning and use a reinforcement learning (RL) agent to prune filters by considering the estimates of the proposed HW-Flow framework. With this method, we overcome the burden of wasting GPU-hours for optimizing CNNs, which do not guarantee an efficiency gain for a target hardware platform. We can assert that the decision of pruning rate for each layer is highly correlated to the target hardware constraint and inference platform. The estimates generated for a HW platform can directly influence *which* layers are pruned and to what extent. They also help the CNN designer understand which scheduling schemes and hardware dimensions are necessary to have a reasonable pruning rate/burden, to match the target application constraints. The contributions of this work can be summarized as follows:

- We introduce HW-Flow, a framework for optimizing and exploring CNN models based on three hardware abstraction levels, *Coarse* | *Mid* | *Fine*, by scheduling and mapping workloads onto potential HW-architectures. With this approach, we model different HW platforms without costly fabrication and explore the pruning potential of CNN models at each design phase.
- We reduce the time required to produce an optimal schedule using analytical search approaches, circumventing exhaustive and random sampling techniques used in literature.
- We augment a state-of-the-art learning-based pruning agent [16] with rewards in the form of model-based HW estimates (e.g., `OPs`, DRAM accesses, energy, and latency). Using this information, the agent produces an optimal pruning strategy required to meet the target constraints. We obtain different pruning configurations, which result in a 2× reduction of the respective KPIs (Key Performance Indicators), with minimal accuracy degradation.

2 Background

2.1 Convolutional Neural Networks

CNNs are deep neural networks which are well-suited for generating predictions based on multi-dimensional, localized input feature spaces, e.g. image processing applications. The convolution of an input activation A^{l-1} with the convolution kernel W^l produces an output feature map A^l, where each pixel of the feature map A^l can be computed as shown in equation 1.

$$A^l[c_o][h_o][w_o] = \sum_{c_i}^{\overbrace{C_i}^{\text{Inp.Ch}}} \; \sum_{k_w}^{\overbrace{K_w \;\; K_h}^{\text{Kernel.dim}}} \!\!\sum_{k_h} a^{l-1}_{c_i, w_o \cdot s + k_w, h_o \cdot s + k_h} \cdot w^l_{c_o, c_i, k_w, k_h}, \text{ where } A^l \in \mathbb{R}^{C_o \times H_o \times W_o} \tag{1}$$

The input feature maps (*Ifmaps*) denoted by A^{l-1} are composed of multiple channels C_i and spatial dimensions W_i, H_i. To compute the convolution operation, the kernel of dimensions $K_w \times K_h$ slides across the input 2-D map with stride size s. A dot-product is performed between the kernel pixels $w^l \in W^l$ and a sub-set of pixels $a^{l-1} \in A^{l-1}$ from the input volume. The dot-product accumulates the values across all input channels resulting in an output pixel. The convolution operation is the repetition of the aforementioned dot-product operation for the entire *Ifmap* with C_o filters, generating output feature maps (*Ofmaps*) $A^l \in \mathbb{R}^{W_o \times H_o \times C_o}$. Successive layers detect different features in the input image at different scales. The first layers are usually responsible of recognizing simple shapes, edges and patterns, while complex features can be detected at the deeper stages of the network. Fully Connected (FC) layers can be simplified considered a special case of the convolution operation by setting $W_i = K_w$, $H_i = K_h$, $W_o = 1$ and $H_o = 1$. These layers restrict weight reuse opportunities and demand high memory bandwidth during the inference. CNNs have produced better predictions than humans on computer vision applications such as image classification [13] and semantic segmentation [3] using supervised ground truth labels.

Image Classification. Out of O possible classes, the input image is predicted based on the output $\tilde{Y} \in \mathbb{R}^O$. It is typical to translate the problem into predicting the probability of each possible class given an input image, so that the output layer produces a vector with a fixed dimension of O. Several CNN topologies were proposed in the last decade to solve the image classification problem, dealing with different datasets such as CIFAR-10 [19] and ImageNet [30]. For instance, AlexNet was introduced by Krizhevsky et al. [20] as the first CNN topology for classifying the ImageNet dataset. The network consists of five convolutional layers, max-pooling layers, dropout layers and three fully-connected layers, where the last one maps to a 1000-element vector representing the number of possible classes for the ImageNet dataset. Other examples which followed in the next years include VGG-16 [24], Inception-Net [33], ResNet [13] and EfficientNet [34].

Image Semantic Segmentation. Segmentation-based CNNs such as FCN [25] and DeepLab [3] predict the class of each pixel in the input image from O possible categories. The semantic maps are derived from the logits $\tilde{Y} \in \mathbb{R}^{W \times H \times O}$ with O probability values per pixel. The CNN topology for this task follows an encoder-decoder architecture. The encoder network is a feature extractor having a similar architecture as image classification CNNs and the decoder network is a set of upsampling layers which restore the original image resolution in order to predict the pixel-wise class output. FCN uses transpose convolution to upsample features whereas DeepLab uses the bilinear upsampling method.

2.2 Reinforcement Learning

Reinforcement learning (RL) is an exploration-exploitation approach to optimize decisions based on interactions with a dynamic environment [18]. The optimization problem is solved using an agent which is connected to the considered environment via perception and action. The environment is characterized by its state $\mathcal{S}$ which describes the relevant features for the decision making process. At each step, the agent generates an action $\mathcal{A}$ which changes the state of the environment and outputs a reinforcement signal or reward $\mathcal{R}$ describing the quality of this state transition. Given a state and action space, the purpose is to find a policy which maps each state to the optimal action, which maximizes the sum of future rewards. RL-based systems often balance a trade-off between exploration and exploitation while predicting new actions. Exploitation is entirely based on the knowledge acquired by the agent and predicts actions that maximize the expected return value according to the gained experience. Exploration assumes that the current knowledge can be further improved by exploring different actions.

Supervised learning and reinforcement learning differ from each other in two major aspects. First, supervised learning is based on a set of data pairs where each input has a clearly defined ground-truth label. This is not the case for RL where each input (action) generates an immediate reward while the objective function is to maximize the sum of all future rewards. Second, RL is typically deployed in an online manner, i.e. RL-based systems are evaluated and trained concurrently to optimize the agent decisions for new input sequences. Deep Deterministic Policy Gradient (DDPG) [23] is an RL technique that outputs continuous action using Q-Networks and Actor-Critic based policy gradients. Here Q refers to the function which the algorithm predicts. Reinforcement learning has been applied in several fields such as robotics, gaming, traffic light control, and resource management systems [26, 39, 29]. In this work, we use a DDPG-based RL agent in order to decide the optimal pruning configuration of CNNs.

3 Related Work

Based on the target optimization metrics, we classify pruning techniques into four categories: HW-agnostic, pseudo-HW-aware, HIL-based, and HW-modeling-based pruning techniques, as compared in Table 1. Additionally, we discuss HW-modeling works that compute the HW estimates of CNN accelerators in literature.

HW-agnostic Pruning. The advantages of pruning were investigated in early works such as [6, 12]. Subsequent works determined the redundant weights based on an iterative method, without considering any target hardware resource constraints, e.g. simple magnitude-based pruning [11]. Recently, He et al. [15] pruned redundant filters using a geometric median heuristic. However, the efficiency term was limited to the pruning rate (PR), i.e., the ratio of pruned to total parameters. The PR was set constant to all the layers, which does not capture the energy or latency requirements of the target inference hardware. The work by Guo et al. [10], dynamically pruned CNNs irregularly based on a saliency function during training to produce efficient networks. Recently, Frickenstein et al. [9] proposed the auto-encoder-based low-rank filter-sharing technique (ALF), which utilizes sparse auto-encoders to extract the most salient features of convolutional layers, pruning redundant filters. The above works only target to compress the CNN model with minimal accuracy degradation without considering the benefits on the target HW platform.

Pseudo HW-aware Pruning. The authors of [14] proposed structured channel pruning, where the saliency of individual channels is determined through Lasso regression. The pruning ratio for each layer is based on handcrafted heuristics which targets lower *proxy* metrics such as `OPs` and

Params. In more recent works, automated pruning methods have gained popularity. Huang et al. [17] trained layer-specific agents, which receive the kernel matrix as a state and produce actions to prune exact filters. Contrary to [16], here the agent has a more complex task of learning the features of a layer rather than simply its sparsity ratio. The agent's reward is formulated using a multi-objective cost function, which aims to find CNN models with both high accuracy and low *proxy* metrics. Reward functions based on proxy metrics do not guarantee an improvement for HW deployment, as we demonstrate in the following sections. The work in [36] identifies redundant weights for different regularizations during the training process using a HW loss formulation. The HW loss is limited to optimization of *proxy* metrics.

HW-aware Pruning. As hardware platforms tend to be complex, the effects of arbitration, stalls, etc., may be severely understated if hardware estimations purely rely on proxy metrics. By considering real hardware metrics, hardware-in-the-loop (HIL) training frameworks have been used to verify the advantages of CNN optimization techniques pragmatically [42, 7, 16]. NetAdapt [42] prunes filters based on a preexisting look-up table of hardware metrics obtained ahead of time from a mobile device. This is a costly pseudo-HIL approach, as building the look-up table is tedious and time consuming, requiring the designer to execute all possible workloads and layer dimensions to be accurate and complete. For this method to work, the hardware would need to be decided and readily available before the CNN optimization process starts. Another drawback to the approach is that the pruning technique is performed in a layer-wise manner, which is susceptible to local minima, as inter-layer effects on the hardware platform and prediction accuracy are not considered. ChamNet [7] also adopts a look-up table strategy to estimate the latency with a Bayesian energy predictor and performs neural architectural search. The predictors for the HW metrics also require the "ready-to-use" target HW platform to perform optimization. Furthermore, if the target hardware is changed, the effort to recollect the data for the new look-up table and the Bayesian optimizer needs to be taken into account. HW-NAS-Bench [22] presents a dataset to evaluate various CNN configurations on different HW platforms. The dataset is generated by performing extensive real HW measurements on NAS-specific search spaces [8, 40] Furthermore, the dataset does not cover exploration of HW specific hyper-parameters which impacts the CNN compilation/scheduling procedures. The work in AMC-AutoML [16] demonstrated an RL pruning agent, producing channel sparsity ratios for each layer as its action after every episode. Based on the magnitude obtained from the L2-norm heuristic and the sparsity ratio of each layer given by the RL agent, the redundant channels are pruned. The work demonstrated results of both proxy metrics (OPs and Params) and HIL-based timing evaluation using TF-Lite. Another HIL-based optimization technique, HAQ [38], resorts to RL-based exploration to determine suitable, layer-wise quantization levels for weights and activations in the CNN model. The reward function, including real hardware metrics, is generated by directly executing the inference of a CNN model on a Field Programmable Gate Array (FPGA) design which supports quantized computations [31].

Hardware Modeling. The deterministic nature of CNN inference execution on hardware makes analytical hardware modeling an intuitive approach to simulate aspects of the synthesis and deployment phases. Timeloop [27] is a HW-modeling tool that exploits CNN execution determinism to offer accurate estimates of a given hardware description. It shows the strength of HW modeling, circumventing the need for cycle-accurate CNN hardware simulators and/or synthesized hardware in the early phases of development. The tool provides the flexibility of changing the cost of hardware operations (e.g. read, write, multiply-accumulate) and the memory hierarchy, among other design parameters. Based on the data movement constraints set by the designer, the tool searches the scheduling solution space in an exhaustive or randomly sampled manner, thereby providing the HW estimates. The schedule search time could either last significantly long with exhaustive search or lead to a sub-optimal solution with random sampling. Interstellar [43] proposes formal dataflow definitions. Unlike Timeloop, the authors of Interstellar use the Halide programming language to represent the HW-architecture and data movement constraints. The influence of memory hierarchy and dataflows on energy efficiency and latency is investigated thoroughly. MAGNet [37] considers various CNN architectures and hardware constraints generating an optimal RTL and mapping strategy to execute the CNNs efficiently. It explores various tiling strategies and dataflows by proposing a highly configurable processing element array. Yang et al. [41] leverage a HW-model to estimate the energy requirements of each layer. The layers with the highest energy contribution present a good starting point for the pruning process, based on the L2-norm heuristic. However, energy estimates do not influence the sparsity ratio directly. The work is also limited to optimizing normalized energy, but not latency, which is an equally important parameter for real-time applications. In this work, we remove the limitations of pure proxy and HIL-based neural network pruning by introducing a multi-abstraction level HW-model for estimating the efficiency of CNN architectures. Instead of exhaustive and random sampling search techniques, we analytically reduce the size of the search space, and thereby the search-time, without sacrificing schedule efficiency. A deep deterministic policy gradient (DDPG) based learning agent is augmented with key rewards and state information, allowing it to understand the influence of its pruning actions on the inference hardware for energy and latency, and enabling HW-CNN codesign-based optimization.

Table 1 Classification of pruning, modeling techniques and their advantages.

Advantage	Agnostic	Proxy	HIL	HW-Model	HW-Flow
	[15, 10, 9]	[14, 17, 36]	[16, 42, 7, 22]	[27, 41]	[Ours]
Accuracy optimization:	✓	✓	✓	✓	✓
`OPs`/`Params` optimization:	✗	✓	✓	✓	✓
Energy/latency optimization:	✗	✗	✓	✓	✓
HW-design exploration:	✗	✗	✗	✓	✓
Learning based agent:	✗	✓	✓	✗	✓
HW-CNN codesign (abstraction/refinement):	✗	✗	✗	✗	✓

4 Hardware-Flow

In the case of general-purpose HW, such as off-the-shelf GPUs and CPUs, HIL-based approaches may indeed be less cumbersome than building a hardware model. For many real-time, energy, and latency-critical applications, these platforms are not applicable at deployment time. Carefully designing custom hardware, which meets specific criteria for an application, necessitates following a HW-SW codesign paradigm. In the top-down approach, this implies iteratively going through different levels of abstraction and performing some iterations of exploration before fixing some parameters and refining the design to one abstraction level lower. During inference, CNN forward-pass executions are entirely deterministic, making it hard to justify the need for synthesized hardware to observe training or optimization effectiveness. This makes hardware modeling an attractive and economical alternative for rapid prototyping and testing of different HW-aware optimization strategies.

In this paper, the HW-Flow framework is based on an interaction between the CNN environment f, the pruning agent π and the hardware model μ (shown in Figure 1). In detail, the agent receives a layer-wise state $\mathcal{S}^l$ and an accuracy term ψ, from the environment. The environment's accuracy/precision ψ is computed with respect to the logits $\tilde{Y}$ and labels of the validation set. Depending on the level of abstraction, the CNN f is simulated and scheduled on the HW-model μ, returning estimates φ of an embedded application for the agent to produce a pruning action $\mathcal{A}^l$.

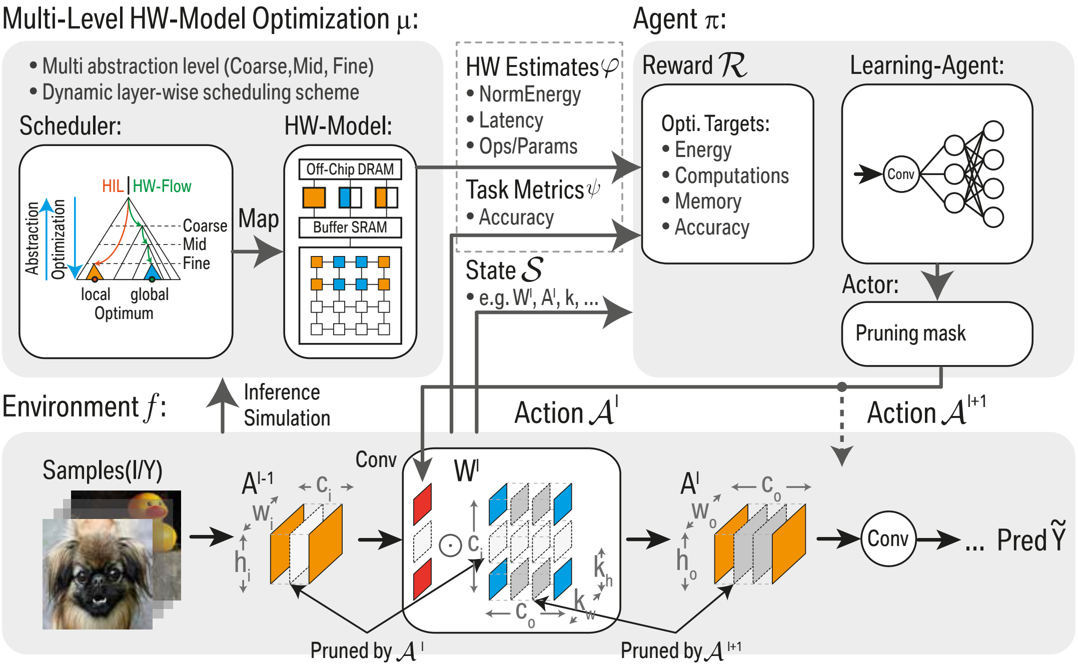


Figure 1 Overview of HW-Flow enabling HW-model based pruning. The CNN environment (bottom) is pruned by a DDPG agent (right). The distinction of layer-wise sparsity is based on the three proposed HW-modeling abstraction levels - *Coarse*, *Mid* and *Fine*, which estimate the complexity of CNN workloads.

4.1 Problem Formulation

Without loss of generality, in an L-layer CNN, the convolutional layer $l \in \{1, ..., L\}$ receives an input feature map $A^{l-1} \in \mathbb{R}^{H_i \times W_i \times C_i}$, where H_i, W_i, and C_i indicate the spatial height, width, and input channels respectively. A^0 is the input image I to the CNN, as shown in Figure 1 (bottom). The weights $W \in \mathbb{R}^{K_h \times K_w \times C_i \times C_o}$ are the trainable parameters of the individual layers, here K_h, K_w, and C_o are the kernel dimensions and the number of output channels (filters) respectively. The input A^{l-1} is convolved with the weights W^l, where the kernels are moved over the input with stride s. In detail, the task of the agent π is to prune the input channels C_i of the environment by zeroizing the binary pruning mask $\mathcal{A}^l = \{0,1\}^{1\times 1\times C_i \times 1}$. To select the most salient channels, the Hadamard product $\odot$ is applied, giving a sparse representation $\tilde{W}^l \in \mathbb{R}^{K_h \times K_w \times C_i \times C_o} = W^l \odot \mathcal{A}^l$. Referring to Figure 1, zeroizing an input channel in the l layer will zero out the corresponding output feature map from the $l-1$ layer. Consequently, the kernels of all filters in the $l-1$ layer are also zeroed-out. Channel-wise pruning removes several weights from the CNN at once, causing a significant loss in accuracy. To mitigate this negative effect and guarantee an energy and latency efficient compression, the learning-based agent π has to learn good actions $\mathcal{A}^l$. The goal of HW-Flow is to complement well-established proxies, such as `OPs` and `Params` count, with more elaborate HW-model based estimates, which are conducive to finding efficient CNNs for embedded applications.

4.2 Deep Deterministic Policy Gradient-based Agent

The DDPG agent's architecture, including the actor and the critic, is adopted from He et al. [16]. The agent is augmented with key rewards and state information, allowing it to understand the influence of its pruning actions on the inference hardware with respect to the energy estimates φ_E and the latency estimates φ_L, elaborated in Section 4.4. The newly adapted state $\mathcal{S}$ is composed of the following layer information of the environment's f: the index of the layer l, stride s and the layer dimensions after pruning $\tilde{C}_o, \tilde{C}_i, W_i, H_i$. It should be noted that the multi-level estimates φ obtained from the HW-models are considered to be part of the state $\mathcal{S}$, where $\varphi = [\varphi^0, ..., \varphi^l, ..., \varphi^L]$ ensembles either layer-wise energy estimates φ_E or latency estimates φ_L. As expressed in equation 2, the action $\mathcal{A}^{l-1}$ is applied for composing the input state of the agent.

$$\mathcal{S}^l =< l, s, \tilde{C}_o, \tilde{C}_i, W_i, H, \varphi^l, \sum_{i=0}^{l-1} \varphi^i, \sum_{j=l+1}^{L} \varphi^j, \mathcal{A}^{l-1} > \tag{2}$$

In this work, the agent is trained using one of the two search protocols, either the estimate *balanced* or estimate *constrained* reward function, as defined in equation 3. The *balanced* reward of equation 3 is inspired by [17]. When the HW-constraints are unknown in the early stages of the design, the reward function can be formulated to achieve at least a target accuracy ψ^* before optimizing the performance estimate term. A trade-off between the accuracy term $(1 - (\psi^* - \psi)/b)$ and the estimate term $\log(\varphi^*/\varphi)$ after each pruning action is the goal of the estimate *balanced* reward function. Parameter b influences the turning point between a negative and positive reward $\mathcal{R}$, encouraging the agent to improve the accuracy when the difference between ψ^* and ψ is larger than b. When this condition is met, the agent starts to optimize the trade-off between accuracy and hardware estimates. This reward can also be extended to optimize multiple KPI's by appending several logarithmic terms. The estimate *constrained* compression improves the reward $\mathcal{R}$ by maintaining higher prediction accuracy ψ after each pruning action. This encourages the agent to prune the CNN model while minimizing the accuracy degradation when the HW-constraints are strictly stipulated.

$$\mathcal{R} = \begin{cases} (1 - \frac{\psi^* - \psi}{b}) \cdot \log(\frac{\varphi^*}{\varphi}), & \text{if } \textit{balanced} \\ \psi, & \text{otherwise } \textit{constrained} \end{cases} . \tag{3}$$

The estimate term in the reward represents the benefits obtained from pruning with respect to the HW-model μ, giving the estimate φ^* of the unpruned base model and φ after each episode of the agent.

4.3 HW-model Abstraction Levels

The HW-Flow framework proposes 3 levels of abstraction, *Coarse*, *Mid*, and *Fine*, for estimating the complexity of a CNN workload (Table 2). At the *Coarse* level, the goal would be to narrow down the CNN architectures which would suit an application's accuracy requirements, while maintaining a reasonable number of `OPs` and `Params`.

Once satisfied with the CNN's computational complexity, the *Mid*-level estimates take intermediate design parameters such as memory hierarchy, partitioning, and bandwidth into consideration, which decide whether the off-chip to on-chip communication infrastructure is suitable for the considered HW design. At the *Mid*-level, HW-Flow provides refined metrics such as off-chip to on-chip data transfer volumes, computation-to-communication ratio (CTC) and off-chip energy $\varphi_{E,\texttt{off-chip}}$ due to external memory accesses. Using this information, the candidate CNNs can be tested for various criteria, such as respecting the communication bandwidth constraints expected

Table 2 Input, output and optimization details of HW-Flow's abstraction levels.

Level	Input	Optimization	Output
Coarse:	• CNN graph	• N/A	• `OPs`/`Params`
Mid:	• Memory hierarchy • Memory size/partitioning • Off-chip bandwidth/burst length • Compute array sizes	• Loop tiling • Loop reordering	• CTC ratio • Off-chip energy/accesses: $\varphi_{E,\texttt{off-chip}}$ • Mid Latency $\varphi_{L,\texttt{bandwidth}}$
Fine:	• Complete memory/compute hierarchy • PE specification • Supported dataflows	• Loop unrolling • Interleaving • Folding • Mapping exploration	• Total inference energy: $\varphi_{E,\texttt{NE}}$ • Breakdown of datatype energy • Total inference latency: $\varphi_{L,\texttt{inference}}$ • Detailed data movement schedule

on the hardware platform, resulting in latency estimates $\varphi_{L,\texttt{bandwidth}}$ at the *Mid* level. In Section 5, we demonstrate how the *Mid*-level can provide an intermediate evaluation closer to the detailed *Fine*-level without requiring the designer to decide the low-level details of the compute array at this stage. This eases the design process into the next stage and provides one more stepping-stone before narrowing down the complete HW design.

HW-Flow's *Fine* estimates provide metrics that consider specific details of the accelerator, such as the detailed structure and dimensioning of the processing element (PE) array and the scheduling strategies supported by the on-chip interconnect, which determines the possible communication channels between the PEs. The estimates at this level provide normalized energy costs $\varphi_{E,\texttt{NE}}$ of each datatype $\texttt{dtype} \in \{\texttt{ifmap}, \texttt{ofmap}, \texttt{psum}, \texttt{weight}\}$ at each memory level of the system. The agent π receives estimates for the energy and latency for a particular scheduling strategy relative to each layer of the investigated network. At the *Mid* and *Fine* levels, HW-Flow additionally searches for scheduling solutions which optimize the criteria provided by the designer, e.g. latency, energy or a trade-off. The latency is the time interval between the stimulation of the host and its response from the accelerator for a particular neural network, i.e. time between the pre-processed input and the output of the neural network. The total energy consumption of the CNN accelerator is determined using the data movement cost at various memory hierarchies and compute cost in processing element array.

For a human designer, the three levels allow the structured traversal of the HW and CNN design space, which may be a daunting task otherwise. The designer starts with a set of potential CNNs and tests their pruning potential in terms of `OPs`, `Params` and task-related accuracy. Once a subset of CNNs with high pruning ratios and acceptable task-related accuracies is narrowed down, the *Mid*-level is then critical in designing the communication infrastructure between the host and the accelerator, as well as dimensioning the on-chip SRAM. After pruning, an under-dimensioned SRAM can result in high communication effort with off-chip DRAM, resulting in higher stress on the off-chip to on-chip interconnect. An over-dimensioned SRAM can lead to area-on-chip and fabrication cost problems. Therefore, the pruning potential of the CNN needs to be evaluated *alongside* the on-chip SRAM dimensions and the interconnect stress at the *Mid*-level. Finally, at the Fine-level, the designer can use the findings at the *Coarse* and *Mid*-level and further specify the computation infrastructure of the accelerator (such as PE count, register files sizes, dataflow). Tackling all three levels by searching for an efficient HW and CNN configuration at once would potentially lead to sub-optimal results. This would also result in more GPU hours of CNN pruning and HW search due to the difficulty in understanding which part of the system (CNN or HW) needs to be adjusted to meet the demands of the application at hand.

4.4 HW-model Optimizer

Model: The core structure of the HW-model in HW-Flow is based on two generic blocks, namely the memory and compute blocks. Arbitrary memory hierarchies can be instantiated using these generic blocks. Each block is accounted for its position in the hierarchy by referring the memory below it and the level at which it is placed. The highest level represents the largest memory, where all the data fits. Memory blocks can be detailed with their total or datatype-wise segmented size. Below last-level memories, a compute block can be instantiated, as shown in Figure 2. The compute block is defined by several parameters, including the number of processing elements (PE), interconnect dimensions, and register file sizes. The register files in each PE can be specified according to their size and segmentation, as shown in Figure 2 (top/right). Using these blocks, diverse compute architectures can be described. In this paper, we focus on modeling architectures similar to [4], with a single on-chip buffer and a compute core with an array of PEs, as depicted in Figure 2.

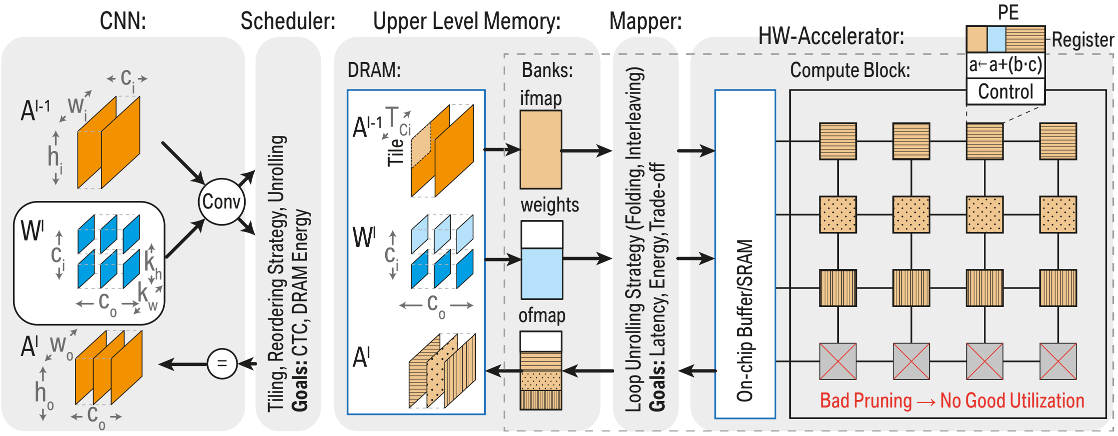
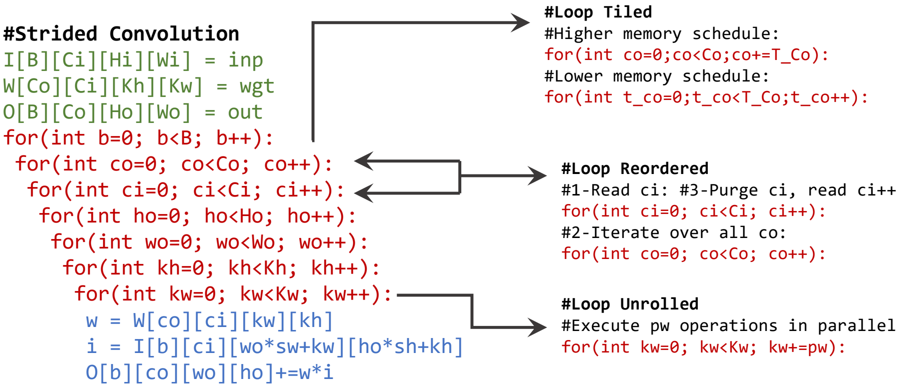


Figure 2 Scheduling strategies for data movement optimization.

Scheduler. The energy contribution of data movement cannot be disregarded for efficient execution of CNNs. For most cases, it constitutes the majority of the total power consumption to execute CNN models. CNNs are commonly represented in a nested loop format, as expressed in Figure 3. The for-loops shown present many reuse opportunities. The main computation



```
#Strided Convolution
I[B][Ci][Hi][Wi] = inp
W[Co][Ci][Kh][Kw] = wgt
O[B][Co][Ho][Wo] = out
for(int b=0; b<B; b++):
 for(int co=0; co<Co; co++):
  for(int ci=0; ci<Ci; ci++):
   for(int ho=0; ho<Ho; ho++):
    for(int wo=0; wo<Wo; wo++):
     for(int kh=0; kh<Kh; kh++):
      for(int kw=0; kw<Kw; kw++):
       w = W[co][ci][kw][kh]
       i = I[b][ci][wo*sw+kw][ho*sh+kh]
       O[b][co][wo][ho]+=w*i
```

```
#Loop Tiled
#Higher memory schedule:
for(int co=0;co<Co;co+=T_Co):
#Lower memory schedule:
for(int t_co=0;t_co<T_Co;t_co++):
```

```
#Loop Reordered
#1-Read ci: #3-Purge ci, read ci++
for(int ci=0; ci<Ci; ci++):
#2-Iterate over all co:
for(int co=0; co<Co; co++):
```

```
#Loop Unrolled
#Execute pw operations in parallel
for(int kw=0; kw<Kw; kw+=pw):
```

Figure 3 Nested for-loop representation of strided convolution.

is at the core of the inner-most loop and many elements are accessed in multiple iterations of the higher loops. Specifically, reuse occurs when the indices of the parameters involved in the inner-most computation remain fixed for some loops before iterating in others. In hardware, this translates to a single element being stored at a lower level memory for multiple iterations before being purged to make space for new data. For optimal reuse to occur, no single element should be read more than once from a higher level memory. This implies that during all the iterations that a single element is involved in, all the other elements that it is reused against also fit in the lower level memory. Practically, due to memory constraints, the parameters required by the entire nested-loop do not fit in the lowest-level of the memory hierarchy. A standard method of exploiting the entire hierarchy is to relax this constraint and split the for-loops into shallower loops through a technique called *loop-tiling*. As shown in Figure 2, the loop tiling strategy effectively decides which tiles $T \in \{T_{C_i}, T_{C_o}, T_{H_o}, T_{W_o}, T_{K_w}, T_{K_h}, T_B\}$ of CNN computation will take place in one round of communication with a lower level memory. Note that T_B is the tiling along the batch dimension when performing batch processing. The tiling strategy is selected based on the amount of on-chip buffer Buf, respecting the inequality in equation 4.

$$\underbrace{T_{H_i} \times T_{W_i} \times T_{C_i} \times T_B}_{\text{Input Tile}} + \underbrace{T_{H_o} \times T_{W_o} \times T_{C_o} \times T_B}_{\text{Output Tile}} + \underbrace{K_h \times K_w \times T_{C_i} \times T_{C_o}}_{\text{Weight Tile}} \leq Buf \tag{4}$$

To generate an output tile with spatial dimensions T_{W_o}, T_{H_o} with stride s and kernel k, an input tile with spatial dimensions T_{W_i}, T_{H_i} are required. The relation between input and output tiles is given in equation 5.

$$\begin{aligned} T_{W_i} &= (T_{W_o} - 1) \cdot s + K_w \\ T_{H_i} &= (T_{H_o} - 1) \cdot s + K_h \end{aligned} \tag{5}$$

The order of the loops can also be manipulated dynamically for each layer without affecting the algorithm through *loop-reordering*. As an example in Figure 3, loop C_i can be swapped with loop C_o, allowing a single element c_i to reside longer on the lower-level memory while iterating over all possible elements $c_o \in C_o$. This can help extract improved reuse opportunities since the lower-level loops remain on the lower-level memories of the hardware architecture, thus closer to the compute units. Particularly for *Mid*-level estimates, these permutations directly impact the number of DRAM accesses and consequently DRAM energy. This work considers three loop orders, namely Input Reuse Order (IRO), Weight Reuse Order (WRO), and Output Reuse Order (ORO) schemes inspired by the work in [35]. Switching dynamically between these three reuse schemes allows to schedule the entire CNN exploiting the reuse opportunities of different layers. As an example, layers with a very large kernel can benefit from ORO and WRO schedules, whereas layers with large feature maps (e.g. the first layers of most conventional CNN) will benefit the most with IRO schedule. This is refereed to as dynamic loop tiling. Finally, once a memory level is distributed spatially, further loops can be unrolled over the parallelism degree offered by the hardware architecture through *loop-unrolling*. In Figure 3, the kernel's elements can be assigned to spatially distributed processing elements, executing several K_w loop iterations in parallel during a single clock cycle. HW-Flow's mapper component optimizes the execution schedule through this technique, as detailed in the following subsection.

Using the aforementioned strategies, the scheduler builds a matrix of possible tilings and loop orders. Each potential solution in the matrix is checked for *legality*, by assessing whether its transfer size breaches the memory restrictions at lower levels. The volumes $\mathcal{V}$ moved between the memory levels are calculated based on the memory occupation and the number of invocations required. To analytically reduce the size of the search space, a Computation-to-Communication (CTC) *hall-of-fame* is constructed after the evaluation of all legal solutions, which contains only a top percentage of the highest CTC loop tilings/orderings. Equation 6 represents a CTC ratio formula, inspired by the work in [44]. γ represents a bandwidth-correction term to account for the burst-length of the memory transfers. The numerator is the number of operations/complexity of a particular workload. The denominator is the overall DRAM access along with bandwidth scaling for input, weights, and outputs for a particular workload.

$$\begin{aligned} &\texttt{CTC} = \frac{2 \cdot H_o \cdot W_o \cdot K_h \cdot K_w \cdot C_o \cdot C_i}{\sum_{\texttt{dtype}} \gamma_{\texttt{dtype}} \cdot \mathcal{V}_{\texttt{dtype}}}, \\ &\texttt{dtype} \in \{\texttt{ifmap}, \texttt{ofmap}, \texttt{psum}, \texttt{weight}\} \end{aligned} \tag{6}$$

The hall-of-fame solutions are passed on to the mapper for *Fine*-level estimations. An analysis on the hall-of-fame size and the efficiency of the final schedule produced is presented in Section 5.4.

Mapper. Many dataflow strategies have been explored in literature [4, 43]. Reuse opportunities in CNNs include convolutional, weight, input, and partial sum reuse. In this work, we focus on three dataflows, namely weight-stationary, output-stationary, and row-stationary. The weight-stationary dataflow unrolls the dimensions T_{C_i} and T_{C_o} as P_{C_i} and P_{C_o} across the spatially distributed computation array. Each PE holds complete kernels ($K_w \times K_h$) and corresponding input feature slices. Spatial reduction of partial sums can occur inside the PE; however, accumulation across input channels requires `psum` traversal over the spatial computation array. The output-stationary dataflow similarly unrolls P_{C_i} and P_{C_o}; however, the `psums` remain stationary in each processing element, while input feature map pixels traverse the array and kernel pixels are updated once they are exhaustively used over the tile. Finally, row-stationary as introduced in [4] unrolls the T_{H_o} dimension horizontally across the array as P_{H_o}. Each K_h column of PEs is responsible for the complete computation of an entire row of the output W_o, while the neighboring set of K_h PEs computes the output row below that. Folding and replication techniques are applied to fit this unrolling method on the physical array dimensions. All three dataflows enable interleaving of channel computation within a single PE to maximize the use of the register files.

HW-Flow's mapper analytically determines the viability of a particular dataflow, based on the hardware details such as the interconnect dimensions, processing array size, and scratchpad configuration. HW-Flow attempts to find a mapping that optimizes a given criterion (energy, latency, or a trade-off) while respecting the dataflow's restrictions. As presented in Figure 3, unrolling a subset of a loop's iterations as P spatially distributed computations, improves execution time. Assuming a filled pipeline, the latency of a layer φ_L can be estimated as the product of intertile and intratile latency as shown in equation 7. The intertile latency is computed based on the number of tiles required to transfer from off-chip memory to on-chip memory. Based on the PE unrolling procedure of the tiles available in the on-chip memory, the intratile latency is calculated. In equation 7, the kernel dimensions K_h and K_w are not tiled, as such granular tilings result in performance degradation for modern CNN models with small kernel sizes.

$$\begin{aligned} \tilde{\varphi}_{L,\text{interTile}} &= \left\lceil \frac{C_o}{T_{C_o}} \right\rceil \cdot \left\lceil \frac{C_i}{T_{C_i}} \right\rceil \cdot \left\lceil \frac{H_o}{T_{H_o}} \right\rceil \cdot \left\lceil \frac{W_o}{T_{W_o}} \right\rceil \\ \tilde{\varphi}_{L,\text{intraTile}} &= \left\lceil \frac{T_{C_o}}{P_{C_o}} \right\rceil \cdot \left\lceil \frac{T_{C_i}}{P_{C_i}} \right\rceil \cdot \left\lceil \frac{T_{H_o}}{P_{H_o}} \right\rceil \cdot \left\lceil \frac{T_{W_o}}{P_{W_o}} \right\rceil \cdot \left\lceil \frac{T_{K_h}}{P_{K_h}} \right\rceil \cdot \left\lceil \frac{T_{K_w}}{P_{K_w}} \right\rceil \\ \tilde{\varphi}_{L,\text{total}} &= \tilde{\varphi}_{L,\text{interTile}} \times \tilde{\varphi}_{L,\text{intraTile}} \end{aligned} \tag{7}$$

A particular mapping produces reuse factors for each datatype at different memory levels. We denote a reuse factor with $R^{\texttt{dtype}}_{\texttt{level}}$, where $\texttt{level} \in \{\texttt{Offchip}, \texttt{Onchip}, \texttt{Array}, \texttt{Registers}\}$. Reuse factors are dependent on tiling and unrolling strategies, as well as data interleaving [4], where a single computation element switches between multiple sets of the same datatype in order to extend the utilization of its registers. Once a legal mapping is found, the energy contributions of each datatype at each memory level can be computed. Equation 8 shows an example of the energy consumption calculation at a particular memory level for a single datatype [4]. The read/write cost term $\mathcal{C}$ of a particular memory level can be set based on the fabrication technology or a relative normalized cost to other memory types in the hardware architecture. The energy estimates of all datatypes at all memory levels can be calculated similarly and summed up to obtain the total layer energy φ_E.

$$\begin{aligned}&\varphi_{E,\text{Level}}(\texttt{dtype}) = (\prod_{\texttt{off-chip}}^{\texttt{Level}} R_{\texttt{level}}^{\texttt{dtype}}) \cdot \mathcal{C}_{\texttt{Level}}\\ &\forall\ \texttt{dtype} \in \{\texttt{ifmap}, \texttt{ofmap}, \texttt{psum}, \texttt{weight}\}\end{aligned} \tag{8}$$

Finally, the mapping found is fed back to the scheduler, determining whether the tiling factors it provided were adequate. The possible combinations for legal schedules are evaluated and compared. These two optimization problems are codependent, as a tiling strategy that optimizes off-chip data movement may result in a mapping that under-utilizes the processing elements for a particular dataflow and vice versa. Therefore, a feedback loop, such as the one in HW-Flow, is essential in finding an optimal scheduling strategy for the overall system.

4.5 Search Space for HW-model Optimizer

For *Fine*-level estimates, creating a complete schedule implies choosing a fixed set of tiling factors $\{T_{C_i}, T_{C_o}, T_{H_o}, T_{W_o}\}$ and unrolling factors $\{P_{C_i}, P_{C_o}, P_{H_o}, P_{W_o}, P_{K_h}, P_{K_w}\}$. We restrict $T_{K_h} = K_h$ and $T_{K_w} = K_w$, and therefore omit them from the tiling factors set. Modern CNNs employ small kernel sizes, making it unreasonable to tile them during computation. Furthermore, tiling the kernel dimension generates a large amount of partial sums which can quickly become *parasitic* due to memory consumption and on-chip movement, if not collapsed into an output pixel. We can define two subspaces in the scheduling search space: tiling space $\mathcal{T}$ and mapping space $\mathcal{P}$. Equation 9 defines the size of the subspaces. *Ord* defines the reordering possibilities of the outer (off-chip memory) loops of the convolution. In this work, we consider three distinct orderings, IRO, ORO, and WRO, relating to inputs, outputs, or weights being kept longer on the on-chip memory respectively [35].

$$\begin{aligned}|\mathcal{T}| &= C_i \cdot C_o \cdot H_o \cdot W_o \cdot Ord\\ |\mathcal{P}_\tau| &= T_{C_i} \cdot T_{C_o} \cdot T_{H_o} \cdot T_{W_o} \cdot T_{K_h} \cdot T_{K_w} \quad \forall \tau \in \mathcal{T}\end{aligned} \tag{9}$$

$|\mathcal{T}|$ and $|\mathcal{P}_\tau|$ represent the cardinality of the tiling space and mapping space associated with a single tiling $\tau \in \mathcal{T}$ respectively. Therefore, the size of $\mathcal{P}_\tau$ is directly dependent on a single solution $\tau = \{T_{C_i}, T_{C_o}, T_{H_o}, T_{W_o}, Ord\} \in \mathcal{T}$. Restricting $\mathcal{T}$ directly reduces the number of total $\mathcal{P}_\tau$ searches necessary for finding a schedule. $\mathcal{T}$ may contain a single solution τ which results in a single mapping $\rho \in \mathcal{P}_\tau$, that is optimal for the overall schedule, in terms of latency, energy, or both. The trade-off in restricting the size of $\mathcal{T}$ is between schedule search speed and the optimality of the found schedule. To avoid evaluating drastically sub-optimal tilings, we analytically reduce the size of $\mathcal{T}$, and maintain solutions τ, which have a higher probability of producing efficient ρ mappings. The search for the optimal mapping ρ can also be expedited with further sampling techniques.

A straightforward approach to restricting the search space is to uniformly sample equidistant solutions in $\mathcal{T}$. We choose uniform sampling over random sampling to consider, at a minimum, a single candidate from each neighborhood in the search space. For fixed dimensions C_i, C_o, H_o, and W_o, the distance between two solutions depends on the sampling step. For small search spaces, the sampling step can be set to a small integer value. Therefore, for all experiments on the CIFAR-10 dataset, the sampling step was set to 2, effectively halving the number of tiles from each dimension. For the larger CNN models, better suited for the ImageNet dataset, integer steps are less effective. The size of a particular dimension C_i, C_o, H_o, and W_o, varies greatly between the first layer of the CNN towards the last. This makes the choice of a single integer step-size for all dimensions either grossly large to maintain simulation speed or small to maintain optimality at the cost of prohibitively increased search time. We use a ratio-based sampling to overcome this problem, where the step-size is a fixed fraction of the total dimension. This decouples the dimensions of the CNN from the number of τ mappings to be evaluated. We also allow each dimension to have its own ratio, providing more flexibility in finely searching smaller dimensions and coarsely searching larger ones. One more technique to aggressively reduce the search space is to find all the factors (divisors) of a particular dimension and declare those as the possible tiling factors. Since a factor will always give an integer number of tiles, this method usually leads to near-optimal results and is scalable to larger CNNs.

The CTC ratio metric is elaborated in Section 4.4 for choosing a reasonable tiling solution. Based on the intuition that a high CTC tiling solution τ could result in an efficient mapping, we analytically reduce the search space by creating a CTC hall-of-fame (HOF). In the first step, we evaluate the CTC ratio for all $\tau \in \mathcal{T}$, which is a fast and parallelizable operation. A set percentage of $\mathcal{T}$ with the highest CTC ratios among all the solutions is entered in the HOF. Only members of the HOF have their respective $\mathcal{P}$ searched for mapping solutions.

For *Mid*-level estimates, evaluating the outputs shown in Table 2 is a fast and parallel operation, which can be done by either sampling the tiling space or exhaustively, since the total number of solutions to evaluate at this level is $|\mathcal{T}|$. For *Fine*-level, we have a total number of full schedule evaluations equal to $\sum_{\tau}^{\mathcal{T}} |\mathcal{P}_{\tau}|$, which rapidly grows with $\mathcal{T}$, emphasizing the importance of good search space reduction techniques to maintain reasonable search time, without cutting out the optimal solutions in the space.

5 HW-model Design Space Exploration

We evaluate the HW-Flow framework by exploring the HW estimations at various abstraction levels in Section 5.1. We explore the design choices and estimates at the *Mid* and *Fine*-level abstractions in Section 5.2 and Section 5.3 respectively. We improve the schedule search time of HW-Flow by systematically reducing the search space of the proposed HW-model optimizer using a detailed ablation study in Section 5.4. Finally, in Section 5.5, the optimal mapping is validated with the estimates reported in Eyeriss [4] and compared against the HW modeling framework Timeloop [27]. An advantage of using HW-models over HIL-based methods is the flexibility of prototyping and testing multiple target architectures before committing to a final design for synthesis and fabrication. We report various HW configurations with different PE array sizes, memory costs, SRAM buffer and register sizes in Table 3. Column 3 indicates the data access cost from higher memory levels (DRAM) to lower levels (RF) relative to one MAC operation. This section uses the HW-Flow-Val model with 16-bit word length to explore the *Coarse*, *Mid* and *Fine* abstraction levels and validate the modeling tool.

5.1 HW Estimations at Various Abstraction Levels

In this subsection, we demonstrate the HW estimates produced across various abstraction levels and discuss their use in the context of HW development. For this purpose, we interpret the influence of on-chip buffer size through DRAM access counts and throughput of the HW-Flow-Val

Table 3 Hardware configurations used for experiments and validation. RS refers to row-stationary dataflow.

Architecture Spec / Hardware Model	**PE Array**	**Memory Cost** DRAM, SRAM, Array, RF	**SRAM size** <KB>	**Register Words** filter, ifmap, psums
HW-Flow - Val	16 × 16	200, 6, 2, 1	128	192, 12, 16
Timeloop [27]	16 × 16	200, 7.41, 0, 1	128	192, 12, 16
Eyeriss-like-168 PE (RS)	12 × 14	200, 6, 2, 1	128	224, 12, 14
Eyeriss-like-256 PE (RS)	16 × 16	200, 13.84, 2, 1	256	224, 12, 14
Eyeriss-like-1024 PE (RS)	32 × 32	200, 155.35, 2,1	3072	224, 12, 14
Eyeriss-like-Deeplab (RS)	32 × 32	200, 155.35, 2,1	3072	224, 37, 16

model in Figure 4. We obtain the *Coarse*-level estimations for DRAM access counts (indicated in red) by simply summing-up the layer-wise transfer volumes of `ifmaps, ofmaps` and `weights`. This is equivalent to considering that the HW has unbounded buffers and communication bandwidth. Similarly, we consider that all the compute units in the PE array are fully occupied and report the accelerator's throughput. These assumptions in the initial phases of development allow the designer to choose the CNN topologies that suit the application under consideration. For *Mid*-level estimations, we optimize layer-wise schedules for minimum DRAM energy by considering the possibility of dynamic tiling and reordering schemes as described in Section 4.4. We report the sum of DRAM accesses across all layers (indicated in green). To calculate the accelerator's throughput at *Mid*-level, we consider the external memory bandwidth as 8 words/cycle. For *Fine*-level estimations, we optimize the row-stationary dataflow to obtain a trade-off between normalized energy and latency (indicated in blue).

We perform the measurements for four CNN architectures, namely AlexNet, VGG-16, ResNet18, and ResNet152. We observe that as the on-chip buffer size increases, larger tiles of input, weights, and outputs can be stored on the buffer, thereby decreasing the number of DRAM Accesses. We also observe that the *Mid* and *Fine* estimations for all the CNN architectures could meet the ideal *Coarse* estimations at higher buffer sizes ($\geq$ 512KB). We notice that the *Fine*-level estimations produce a higher number of DRAM accesses, as the schedule must consider more complex HW details and constraints at this level. The AlexNet architecture achieves the least throughput among other architectures, as a considerable number of operations and parameters are assigned to fully-connected layers. The throughput produced at the *Fine*-level considers the dataflow and the underlying unrolling scheme and therefore achieves closer estimates compared to real target deployment. We observe that the throughput saturates at 128KB of buffer size for both *Mid* and *Fine* level estimations for different CNN architectures. A slight decrease in throughput happens for AlexNet and ResNet18 at *Fine*-level scheduling for on-chip memories larger than 128KB. This due to the *Fine*-level schedule simultaneously optimizing for inference energy (not shown in the figure) as the on-chip memory grows.

5.2 Mid-Level Estimations and Design Choices

In Figure 5, we evaluate the number of DRAM accesses for different loop ordering schemes. We also evaluate the *Mid*-level throughput estimation for different external bandwidth considerations (4, 8, 16 words/cycle). Among IRO, WRO and ORO loop ordering schemes, we observe that the IRO scheme produces DRAM accesses close to the layer-wise dynamic ordering scheme for AlexNet. For ResNet18, we observe that the ORO scheme achieves DRAM access closer to the dynamic order. This emphasizes the importance of a dynamic ordering scheme as different workloads prefer reuse

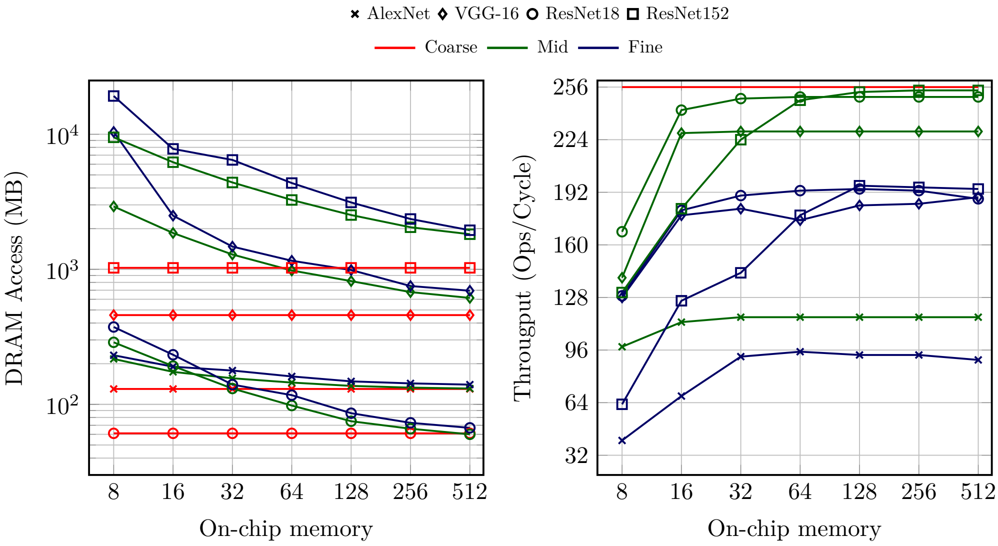


Figure 4 Analysis of DRAM Access and Throughput on varying the on-chip buffer size and different CNN architectures.

for different datatypes. Fixing the dynamic ordering scheme, the throughput of the accelerator is analyzed for AlexNet and ResNet-18 under different external memory bandwidth considerations. We observe a significant improvement in the throughput of AlexNet as the bandwidth increases. For ResNet18, the throughput saturates at 8 words/cycle. Improving the throughput for AlexNet depends on the choice of external memory bandwidth as AlexNet has several memory-bounded fully-connected layers compared to ResNet18.

5.3 Fine-Level Estimations and Dataflows

The row-stationary (RS), weight-stationary (WS) and output-stationary (OS) dataflows, detailed in the Section 4.4, are used to explore the *Fine*-level estimates. The mapper searches for a trade-off between normalized energy and latency while respecting each dataflow's unrolling rules and the HW's memory and compute capacity checks. We obtain the *Fine* estimates for AlexNet and ResNet18 models for different on-chip buffer sizes considering the HW-Flow-Val model. We observe that the RS dataflow is the most energy efficient at all the buffer sizes. This is due to RS maximizing the data reuse at the register-level, for all the datatypes [4]. OS and WS dataflows maximize the compute utilization of PE arrays, albeit with higher normalized energy requirements. The WS dataflow achieves higher throughput using larger buffer sizes ($\geq 32KB$) for ResNet18.

5.4 Search Space Exploration for Fine-level Estimations

We perform multiple experiments to measure the sensitivity of the scheduling tool under the search space sampling strategies described in Section 4.5. Although all schedules produced under any of the sampling strategies are valid, it is favorable to maintain schedules which are close or comparable to the optimum for a particular CNN workload. To explore the search space, we use the HW-Flow-Val model with RS dataflow to estimate convolutional layers of the AlexNet model with 16-bit weights and activations, as it offers a diverse set of workloads with different kernel sizes and strides. The input batch size is set to 16. AlexNet consists of convolutional workloads with strides 4, 2 and 1 and kernels sizes 11, 5 and 3. Grouped convolution is performed for layers 2, 4 and 5.

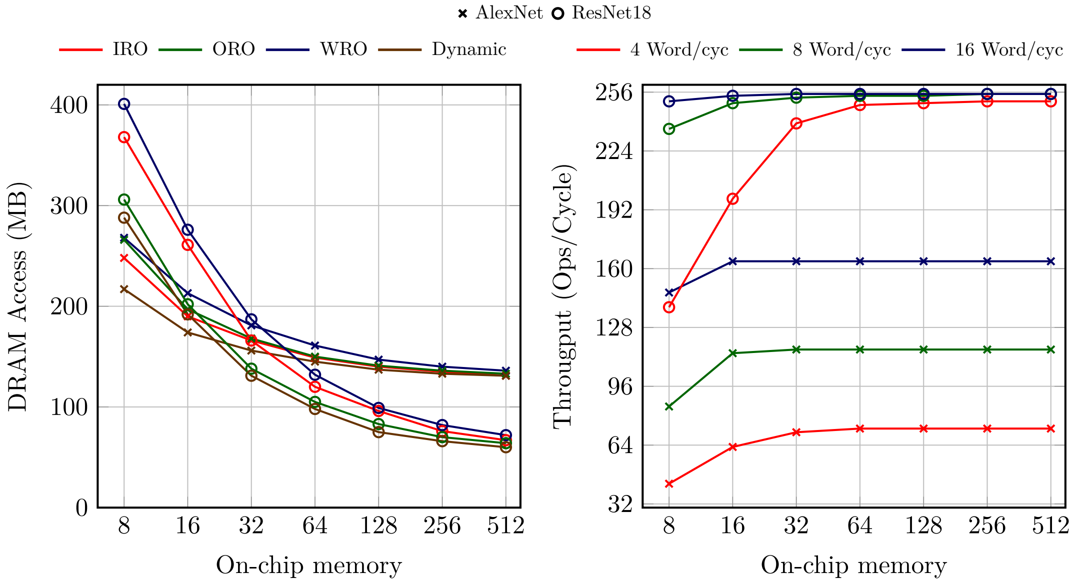


Figure 5 Influence of loop reordering schemes and external memory bandwidth on DRAM access and throughput of the HW accelerator.

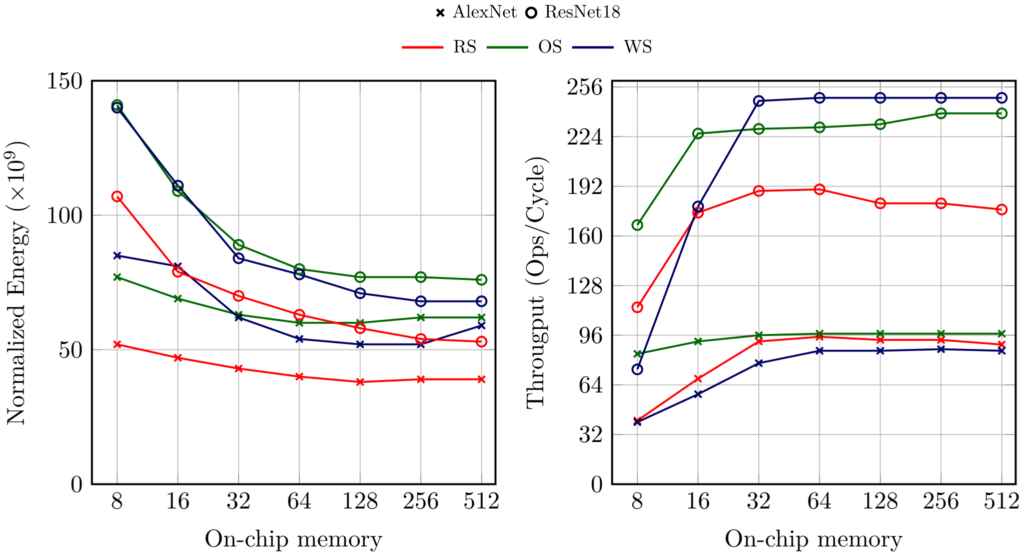


Figure 6 Influence of dataflows selection on normalized energy and throughput of the HW accelerator.

Table 4 shows the search time needed for different analytical search strategies to produce a schedule. The quality of the search method can be measured by its corresponding mapping goal. Three different $\mathcal{T}$ sampling rates (5%, 10%, 20%) are explored in Figure 7 with 1% CTC-HOF. We observe that the normalized energy increases as we limit the exploration by increasing the sampling rate. Based on the trade-off between evaluation speed (see Table 4) and schedule optimality (Figure 7), we sample the tile space with 10% for ImageNet experiments to obtain HW estimates for the pruning process. This results in an overall shorter search time (up to ×2.5) at the cost of degradation in mapping optimization goal. We also highlight the tile sampling method by computing divisors in Table 4. We observe that the divisors-based sampling method produces a schedule with the lowest energy consumption. However, this method might produce sub-optimal results in case of channel pruning when the agent finds prime-number of filters for a particular layer. For CIFAR-10 experiments, we use smaller, integer steps of 2, as these CNNs have a small scheduling search space.

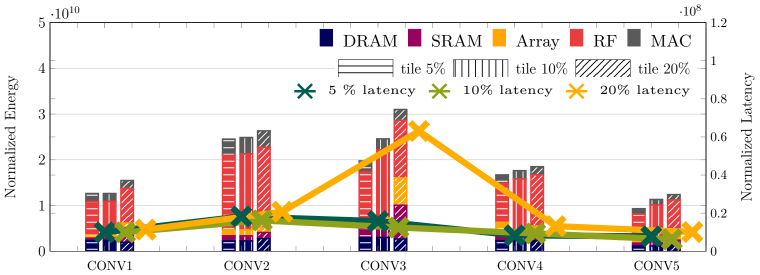


Figure 7 Sensitivity analysis for search space sampling of tiling factors.

The sensitivity analysis of the CTC-HOF tile space reduction technique is shown in Table 4. The results show that the (10% $\mathcal{T}$, 1% HOF) strategy is very effective, providing a speedup of 2.14× compared to (10% $\mathcal{T}$, 100% HOF) schedule without sacrificing the optimality of the schedule. Combining these methods is critical in maintaining a reasonable exploration time for multiple pruning experiments. We finally use the sampling strategy (10% $\mathcal{T}$, 1% HOF) with an overall search time reduction of 20× compared to the search strategy (5% $\mathcal{T}$, 100% HOF). Once a HW-CNN pair is found, the HW-optimizer can run with a more exhaustive search strategy and provide an improved schedule for the final deployment stage. For reference, we also present the search/calculation time for *Mid* and *Coarse* estimate levels in Table 4. In addition to facilitating the proposed codesign approach, the two higher abstraction levels are much faster to estimate, allowing agent π to run for more episodes than with *Fine*-level estimates, for the same amount of time.

5.5 Validation with Eyeriss and Timeloop

Validation: To validate the correctness of HW-Flow's modeling and mapping components, we compare its estimates with the Eyeriss architecture [4] and its Timeloop model [27] for AlexNet [21] inference, which has diversified kernel sizes, strides and input/output dimensions. Figure 8 shows a breakdown of normalized energy contributions of each datatype at each memory level for the convolutional layers. We observe that HW-Flow tracks the original Eyeriss results similar to Timeloop. A slight offset is observed, which can be attributed to small differences in the energy references used during the search. The overlapping line charts show the latency estimates of both frameworks.

6 Experimental Results

In Sections 6.1 to 6.6, we demonstrate the influence of channel pruning on different abstractions, reward functions, target HW architectures and mapping schemes using the estimates generated by HW-Flow. The pruning is evaluated based on CIFAR-10 [19] and ImageNet [30] for the classification task and CityScapes [5] for the semantic segmentation task. The 50K train and 10K

Table 4 Schedule search duration and optimality under different search space reduction strategies for AlexNet on Eyeriss-like-256. All schedules optimize for a trade-off between latency and energy, unless marked otherwise. Similar to Eyeriss [4], we normlize DRAM energy (column 3) and total energy (column 4) to the cost of one MAC operation.

Search Strategy	**Search Time** [s]	**DRAM Energy** [$\times 10^9$]	**Energy** [$\times 10^9$]	**Latency** [$\times 10^6$cycles]
10% $\mathcal{T}$, 1% HOF*	9.87	10.76	**83.25**	379
10% $\mathcal{T}$, 1% HOF**	10.13	155.82	257.49	**65**
10% $\mathcal{T}$, 1% HOF	10.23	11.06	91.89	67
5% $\mathcal{T}$, 1% HOF	25.59	12.10	83.96	60
10% $\mathcal{T}$, 1% HOF	10.23	11.06	91.89	67
20% $\mathcal{T}$, 1% HOF	**7.23**	10.47	104.61	118
divisors $\mathcal{T}$, 1% HOF	12.86	14.60	**71.61**	65
5% $\mathcal{T}$, 100% HOF	215.42	12.10	83.96	**60**
5% $\mathcal{T}$, 1% HOF	25.59	12.10	83.96	60
10% $\mathcal{T}$, 100% HOF	23.03	11.06	91.89	67
10% $\mathcal{T}$, 10% HOF	15.24	11.06	91.89	67
10% $\mathcal{T}$, 1% HOF	**10.76**	11.06	91.89	67
divisors $\mathcal{T}$, 100% HOF	51.47	9.67	**72.44**	65
divisors $\mathcal{T}$, 1% HOF	12.86	14.60	71.61	65
Coarse Estimates	0.10	-	-	-
Mid Estimates 5% $\mathcal{T}$***	5.23	**8.13**	-	-

All simulations were run with 24 threads on an Intel Xeon E5-2698 Process

Mapping goal : *energy, **latency, ***dram access

test images of CIFAR-10 are used to train and evaluate the base models, respectively. The images have a resolution of 32×32 pixels. ImageNet consists of $\sim$ 1.28M train and 50K validation images with a resolution of 256×256 pixels. The CityScapes dataset consists of 2975 training images and 500 validation images, including their corresponding ground truth labels. The images of size 2048×1024 show German street scenes along with their pixel-level semantic labels of 19 classes. The pruning experiments are performed using the agent detailed in Section 4.2, for 150 episodes of pruning exploration and 400 for learning and exploitation. After the agent selects the best action corresponding to the reward, the environment is fine-tuned for 60 epochs with a learning rate of 1e-03 for CIFAR-10 experiments. For ImageNet experiments, we fine-tune for 20 epochs with an initial learning rate of 1e-02, step decay of 1e-01 for every 5 epochs. For CityScapes experiments, we fine-tune the model with a learning rate of 1e-02 for 240 epochs with a polynomial learning rate. If not otherwise mentioned, all hyper-parameters specifying the task-related training were adopted from the CNN's base model. The batch size is set to 4 to evaluate the HW estimates.

HW metrics such as DRAM accesses, normalized energy, and latency are generated based on the different variants of an Eyeriss-like architecture [4] mentioned in Table. 3. These metrics are also reported similarly in the works of Timeloop [27] and Eyeriss [4]. In Table 5-10, the reported normalized energy estimates are relative to the cost of one MAC operation and are therefore unitless. The latency is reported as number of clock cycles. Additionally, we report the accuracy and pruning rate for each experiment. The pruning rate indicates the number of operations reduced relative to the baseline CNN. For comparison with the state of the art, we measure the memory required to store training parameters under 16-bit fixed-point representation. To demonstrate the effect of pruning on different HW-architectures, we scale the variants of spatial, eyeriss-like accelerators from 168 to 256 and 1024 PEs (Table 3).

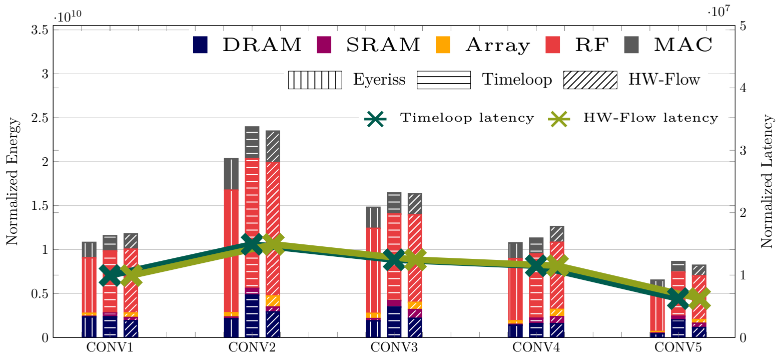


Figure 8 Validation with the Eyeriss accelerator [4] and Timeloop [27]. Note: [4] does not report layerwise latencies.

6.1 Pruning on Different Abstraction Levels

In this experiment, we fix the target hardware architecture to an Eyeriss-like 256 PE accelerator and perform pruning based on HW-Flow's *Coarse*, *Mid* and *Fine* estimates. At each level, we choose the *constrained* reward function (equation 3), and set the target metric reduction to 50% of the baseline value (unpruned model execution). To observe the impact of the metrics at each abstraction level on the hardware architecture, we evaluate all the generated pruned networks on the *Fine*-level model.

The experiments in Table 5 serve as an example on how the three HW abstraction levels (*Course* | *Mid* | *Fine*) can be used to narrow down the range of CNNs and HW architectures which result in an optimal task-to-resource mapping. When using the HW-Flow design methodology, the *Coarse*-level helps the designer evaluate the pruning potential of a set of different CNNs. The designer only needs to have a rough number of `OPs` in mind, a target CNN memory footprint, and an estimation of the desired accuracy. For the purpose of demonstration, we use ResNet56 as our baseline CNN model, with a task-accuracy of 93.59% on the CIFAR-10 dataset. We analyze the compression capability by constraining 50% of `OPs`. After evaluating the CNNs' compression potential, a set of promising candidates can be narrowed down.

The search can be refined to take the on-chip memory hierarchy and dimensioning into consideration at the *Mid*-level. The focus at this level is to choose the correct on-chip memory size and the amount of communication that needs to take place between the host and the accelerator. An under-dimensioned on-chip SRAM would lead to more stress on the communication infrastructure since more rounds of communication are necessary with the DRAM. A large SRAM, although costly, might relieve the complexity of a high-speed, high-bandwidth interconnect. In this context, HW-Flow's *Mid*-level can play a pivotal role in helping the designer dimension the SRAM and the off-chip to on-chip interconnect while considering the pruning potential of the CNN. In Table 5, we check if the on-chip SRAM (256KB) is in a good range to achieve reductions in DRAM accesses without having to over-prune our CNN and lose the task-related accuracy goal with the available loop tiling and reordering possibilities. We observe that the agent prunes 63.84% of `OPs` constraining DRAM accesses to 50%.

Going deeper to the *Fine*-level, the pruning rate is relaxed as the efficient Eyeriss-like architecture is able to meet the constraint requirements without a high pruning rate. Consequently, this preserves the network's accuracy equivalent to the *Coarse* level, while meeting lower target energy and latency.

Table 5 ResNet56 pruned at different HW abstraction levels for Eyeriss-like 256 PE configurations. RS refers to row-stationary dataflow.

Prune configuration	**Acc**	**PR**	**Energy**	**Latency**
(< `constraint` >;< `level` >;< `hw_model` >)	[%]	[%]	[$\times 10^9$]	[$\times 10^3$cycles]
Baseline (not pruned); 256 PE - RS	93.59	-	3.76	2350
-50% Ops *; Coarse; 256 PE - RS	93.03	50.00	2.08	1219
-50% DRAM access *; Mid; 256 PE - RS	91.82	63.84	1.50	862
-50% Energy *; Fine; 256 PE - RS	93.14	54.00	1.88	1159
-50% Latency *; Fine; 256 PE - RS	93.24	50.89	2.05	1176

* : reduction required to meet constraint | (matched for))

6.2 Pruning on Different Rewards

To demonstrate the application of HW-Flow to a hardware design problem, we consider the three candidate Eyeriss-like hardware accelerators with 168, 256, and 1024 PEs. In this experiment, the agent performs pruning based on the two types of reward functions proposed in equation 3, namely estimate *constrained* and *balanced*.

Estimate Constrained. The agent is tasked with pruning the ResNet56 model trained on CIFAR-10 such that it meets a given fixed constraint while minimizing the accuracy degradation of the compressed network. The constraint is set to 50% energy or latency reduction relative to the baseline leader, i.e. the accelerator which performs the best for the target metric. The results in Table 6 show several interesting trends. We observe that the 168 PE variant is the baseline leader for energy-constrained pruning and the 1024 PE accelerator as a baseline leader for latency constrained pruning. With 1024 PEs, there is an ample capacity to improve latency, requiring a lower pruning rate to meet the application constraint. Conversely, the CNN can be pruned more effectively for 168 and 256 PEs when considering an energy-constrained application. For both cases, choosing the correct hardware platform results in a pruned network with higher accuracy. Figure 9a-c shows the agent's decisions across the episodes for the three HW platforms with energy and latency constrained experiments. The noisy actions taken in the exploration phase (first 150 episodes) allow the agent to collect data on the environment and then start convergence and optimization in the following 400 episodes. For Figure 9a-latency and 9b-latency, the agent heavily prunes the model to achieve the target constraints, resulting in an accuracy degradation (marked as red in Table 6 if $\geq 2\%$). In Figure 9c-energy, the agent struggles to meet the desired constraints, resulting in an accuracy degradation after fine-tuning. These critical observations can facilitate the choice of a suitable hardware for a given application constraint.

Estimate Balanced. As detailed in Section 4.2 and equation 3, the balanced estimate reward encourages the agent to maintain the target accuracy ψ^*, while minimizing the estimates φ. Here, ψ^* and b are set to 0.5, 0.125 respectively. Figure 10 demonstrates episode-wise reward plots for the balanced reward formulation of ResNet20 and ResNet56 configurations under various HW platforms. From Table 7, we observe that all the configurations optimized for energy and latency undergo minimal degradation in prediction accuracy with different latency and energy estimates. The Eyeriss-like 168 PE configuration achieves the best energy, whereas 1024 PEs achieves the best latency. Generally, for experiments in Figure 10, we observe an improvement in accuracy and reduction in HW metrics as the number of episodes increase. We also observe that the agent

Table 6 Pruning ResNet56 on CIFAR-10 using estimate *constrained* reward $\mathcal{R}$ on Eyeriss-like accelerators.

Prune configuration (< `constraint` >;< `level` >;< `hw_model` >)	**Acc** [%]	**PR** [%]	**Energy** [$\times 10^9$]	**Latency** [$\times 10^3$cycles]
Baseline (not pruned); Fine; 168 PE - RS	93.59	-	3.72	3377
Baseline (not pruned); Fine; 256 PE - RS	93.59	-	3.76	2350
Baseline (not pruned); Fine; 1024 PE - RS	93.59	-	5.52	588
Target Energy (-50%)**; Fine; 168 PE - RS*	92.63	58.16	1.85	1644
Target Energy (-50%)**; Fine; 256 PE - RS	93.14	54.00	1.88	1159
Target Energy (-66%)**; Fine; 1024 PE - RS	91.09	75.22	1.88	170
Target Latency (-92%)**; Fine; 168 PE - RS	86.89	93.14	0.40	269
Target Latency (-87%)**; Fine; 256 PE - RS	89.66	87.94	0.59	306
Target Latency (-50%)**; Fine; 1024 PE - RS*	92.92	52.68	3.07	294

*: Baseline leader | **: reduction required to meet constraint | (violated constraint) (matched constraint)

Table 7 Pruning ResNet56 on CIFAR-10 using the estimate balanced reward function on Eyeriss-like accelerators.

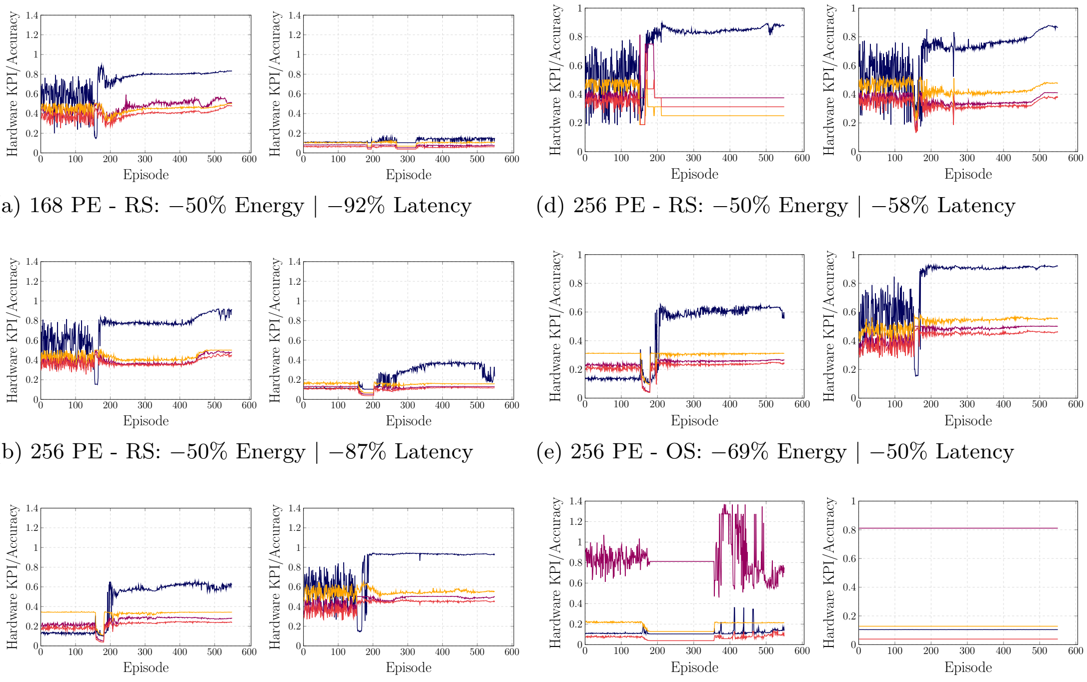
Prune configuration (< `reward` >;< `level` >;< `hw_model` >)	**Acc** [%]	**PR** [%]	**Energy** [$\times 10^9$]	**Latency** [$\times 10^3$cycles]
Baseline (not pruned); Fine; 168 PE - RS	93.59	-	3.72	3377
Baseline (not pruned); Fine; 256 PE - RS	93.59	-	3.76	2350
Baseline (not pruned); Fine; 1024 PE - RS	93.59	-	5.52	588
Energy balanced; Fine; 168 PE - RS	91.94	69.14	1.41	1309
Energy balanced; Fine; 256 PE - RS	92.56	62.22	1.61	913
Energy balanced; Fine; 1024 PE - RS	92.30	62.09	2.69	238
Latency balanced; Fine; 168 PE - RS	92.64	56.50	1.94	1658
Latency balanced; Fine; 256 PE - RS	92.58	59.75	1.69	975
Latency balanced; Fine; 1024 PE - RS	92.97	57.14	3.18	276

finds a pruning strategy for challenging HW configurations (168 PE latency constraint or 1024 PE energy constraint), with minimal accuracy degradation. We also observe quick convergence for ResNet56 pruning on the 256 PE accelerator in Figure 10e.

6.3 Pruning on Different Mappings

The following experiment is performed to evaluate the relationship between effective pruning and an efficient dataflow. We compare the target hardware model, with 256 PEs, against two variants with identical specification, except for their dataflows. Here, the three dataflows, weight-stationary (WS), output-stationary (OS), and row-stationary (RS), described in Section 4.4, are compared in their potential for improved execution of pruned CNNs.

The baseline estimates of the unpruned network show the energy and latency variation caused by dataflows (Table 8). All three dataflows present unique non-dominated solutions for baseline energy and latency. Similar to the estimate constrained experiment in Section 6.2, we set the constraint with respect to the baseline leader dataflow. Figure 9d-f shows the agent's highly varying actions depending on the dataflow and the target constraint. RS results in the lowest baseline energy, whereas the OS has the lowest baseline latency. We can observe that the agent obtains minimum accuracy degradation for RS when constraining for energy. When constraining



(a) 168 PE - RS: −50% Energy | −92% Latency

(d) 256 PE - RS: −50% Energy | −58% Latency

(b) 256 PE - RS: −50% Energy | −87% Latency

(e) 256 PE - OS: −69% Energy | −50% Latency

(c) 1024 PE - RS: −69% Energy | −50% Latency

(f) 256 PE - WS: −79% Energy | −58% Latency

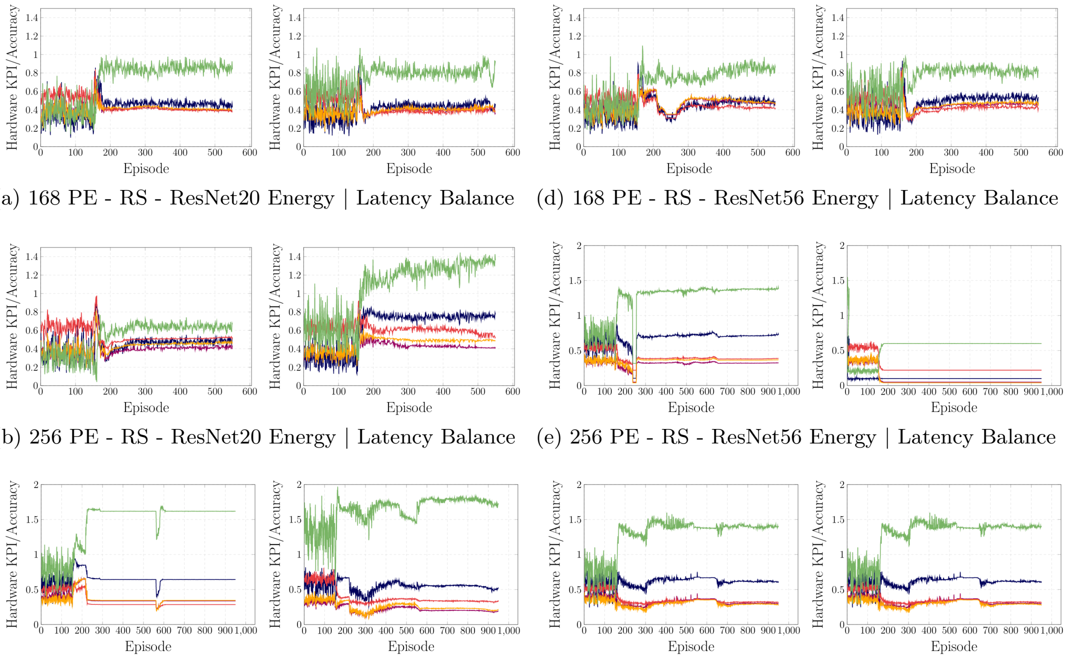
Figure 9 Training curves of the agent detailing the ▬ Reward, reduction in ▬ Latency ▬ Energy ▬ OPs at every episode. We calculate reward by computing prediction accuracy on 10000 randomly sampled images from training dataset.

Table 8 Constraining dataflows relative to 50% of the baseline leader (RS for energy and OS for latency).

Prune configuration	**Acc**	**PR**	**Energy**	**Latency**
(< `constraint` >;< `level` >;< `hw_model` >)	[%]	[%]	[$\times 10^9$]	[$\times 10^3$cycles]
Baseline (not pruned); Fine; 256 PE - OS	93.59	-	5.87	1960
Baseline (not pruned); Fine; 256 PE - WS	93.59	-	5.77	1991
Baseline (not pruned); Fine; 256 PE - RS	93.59	-	3.76	2350
Target Energy (-68%); Fine; 256 PE - OS	91.84	72.05	1.88	584
Target Energy (-68%); Fine; 256 PE - WS	90.06	84.53	1.75	1308
Target Energy (-50%); Fine; 256 PE - RS *	93.14	54.00	1.88	1159
Target Latency (-50%); Fine; 256 PE - OS *	92.91	52.11	3.06	981
Target Latency (-51%); Fine; 256 PE - WS	84.17	96.20	0.71	1612
Target Latency (-58%); Fine; 256 PE - RS	92.36	61.05	1.72	984

*: Baseline leader | (violated constraint) (matched constraint)

for latency, the agent achieves better accuracy for OS and RS. WS demands higher pruning rate when constraining both energy and latency thereby resulting in lower accuracy (marked as red in Table 8). We can also see that the agent does not change its action across several episodes when constraining latency under WS dataflow (see Figure 9f). Thus, we can conclude that the row stationary dataflow is an optimal mapping scheme to achieve efficient energy and latency.



(a) 168 PE - RS - ResNet20 Energy | Latency Balance (d) 168 PE - RS - ResNet56 Energy | Latency Balance

(b) 256 PE - RS - ResNet20 Energy | Latency Balance (e) 256 PE - RS - ResNet56 Energy | Latency Balance

(c) 1024 PE - RS - ResNet20 Energy | Latency Balance (f) 1024 PE - RS - ResNet56 Energy | Latency Balance

Figure 10 Training curves of the agent detailing the ▬ Reward ▬ Accuracy ▬ Normalized Latency ▬ Normalized Energy ▬ Normalized OPs at every episode.

6.4 Layer-wise Analysis of ResNet18

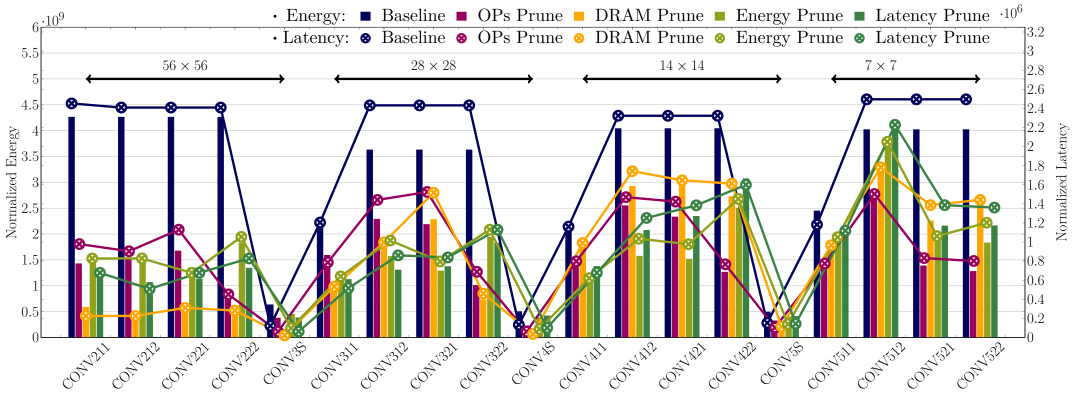


Figure 11 Energy consumption and latency of the pruned layers in ResNet18 on an Eyeriss-like 256 PE accelerator under different pruning constraints.

Based on different pruning constraints, the layer-wise analysis of the ImageNet-trained ResNet18, scheduled on an Eyeriss-like 256 PE accelerator, is presented in Figure 11. The architecture of ResNet18 consists of four stages based on the output feature map spatial size. The results detail the achieved *Fine* estimates (normalized energy and latency). We observe that the agent's pruning rate decision for each layer depends on the constrained HW metric. The layers with higher spatial output sizes (56 × 56) are aggressively pruned when constraining DRAM accesses. On the other hand, the pruning rate for the layers with smaller spatial output size (7×7) is observed higher when constraining for `OPs`. We also observe a lower drop in energy and latency ($\leq 50\%$) for layers such as `CONV411`, `CONV511`, `CONV512` to avoid accuracy degradation for all kinds of pruning constraints. The prediction accuracy and the pruning rates of the four configurations are reported in Table 10.

6.5 Pruning DeepLabv3 for Semantic Segmentation

Using the HW-Flow estimations, we prune DeepLabv3 [3] (using ResNet18 backbone) on the CityScapes dataset. For the DeepLab-based CNN, the bottleneck layers consist of two residual blocks with a dilation rate of 2 and an Atrous Spatial Pyramid Pooling (ASPP) block with dilation rates $\{1, 8, 12, 18\}$. To obtain *Fine*-level estimates for dilated convolutional layers from the HW-Flow framework, we adapt the row-stationary dataflow. The rows of PEs responsible for the dilated parts of the kernel can either be clock-gated or removed from the logical mapping. This implies that the diagonal reuse of input pixels across the spatial array is disrupted. This phenomenon is equivalent to a regular convolution with a large stride, where not every row of the input feature is shared directly with the diagonal neighbor PE [4]. Nevertheless, a non-direct neighbor PE may still reuse the input feature map row. In this case, the potential to reuse an input feature map row at the PE array-level depends on the degree of unrolling P_{H_o}, the dilation rate, and the stride. We use an Eyeriss-like architecture with a large PE array to perform inference of the DeepLabv3 model. In Table 9, we highlight that the DeepLabv3 cannot be scheduled on the standard Eyeriss architecture [4] (Eyeriss-168). This is due to the $ifmap$ register files being dimensioned to hold at-most 12 pixels at a time (see Table 3), which is a decision made by the designers in [4] to support the largest kernel size row in AlexNet (11 pixels). The dilated convolution layers in DeepLabv3, can have up to 36 pixel rows at a time, for a 3×3 kernel with a dilation rate of 18. Increasing the PE array dimensions would not resolve this issue, as it is inherent to the pipeline and dataflow constraints of the Eyeriss-like architecture. We increase the $ifmap$ register sizes to 37 pixels per PE (i.e. $36 + 1$) to make all layers schedulable on the accelerator and obtain baseline estimates.

Table 9 Pruning DeepLabv3 on the CityScapes dataset.

Prune configuration (< `reward` >;< `level` >;< `hw_model` >)	**mIOU** [%]	**PR** [%]	**Memory** [MB]	**Energy** [$\times 10^9$]	**Latency** [$\times 10^6$cycles]
Baseline (not pruned); Fine; Eyeriss-like 168 PE	69.68	-	33.26	NS	NS
Baseline (not pruned); Fine; Eyeriss-like 1024 PE	69.68	-	33.26	NS	NS
Baseline (not pruned); Fine; Eyeriss-like-Deeplab	69.68	-	33.26	1541	267.4
Ops Constrained (Ours); Coarse; Eyeriss-like-Deeplab	69.69	50.00	25.48	954	174.9
Energy Constrained (Ours); Fine; Eyeriss-like-Deeplab	69.88	51.90	29.05	820	161.5
Latency Constrained(Ours); Fine; Eyeriss-like-Deeplab	69.79	60.36	16.87	677	119.6

(NS: Not Schedulable) (matched constraint)

We constrain the number of operations, energy, and latency during the pruning process to 50% as shown in Table 9. There is no degradation in the mIOU (mean intersection over union) evaluation metric for different pruning constraints. We could derive that the unpruned DeepLabv3 model is over-parameterized for the CityScapes dataset. We observe that a higher pruning rate is required to constrain latency to 50%. We highlight the pruned models' effectiveness by demonstrating the semantic predictions on three sample images of the CityScapes dataset. We observe that the pruned models could produce better predictions (terrain in column 1, bikers in column 2, motorcycle and terrain in column 3) due to their higher generalization capability. By analyzing the layer-wise pruning ratios for different target constraints, we observe that the agent heavily prunes the ASPP and decoder blocks. For energy-constrained pruning, the agent only finds redundant operations in the decoder blocks.

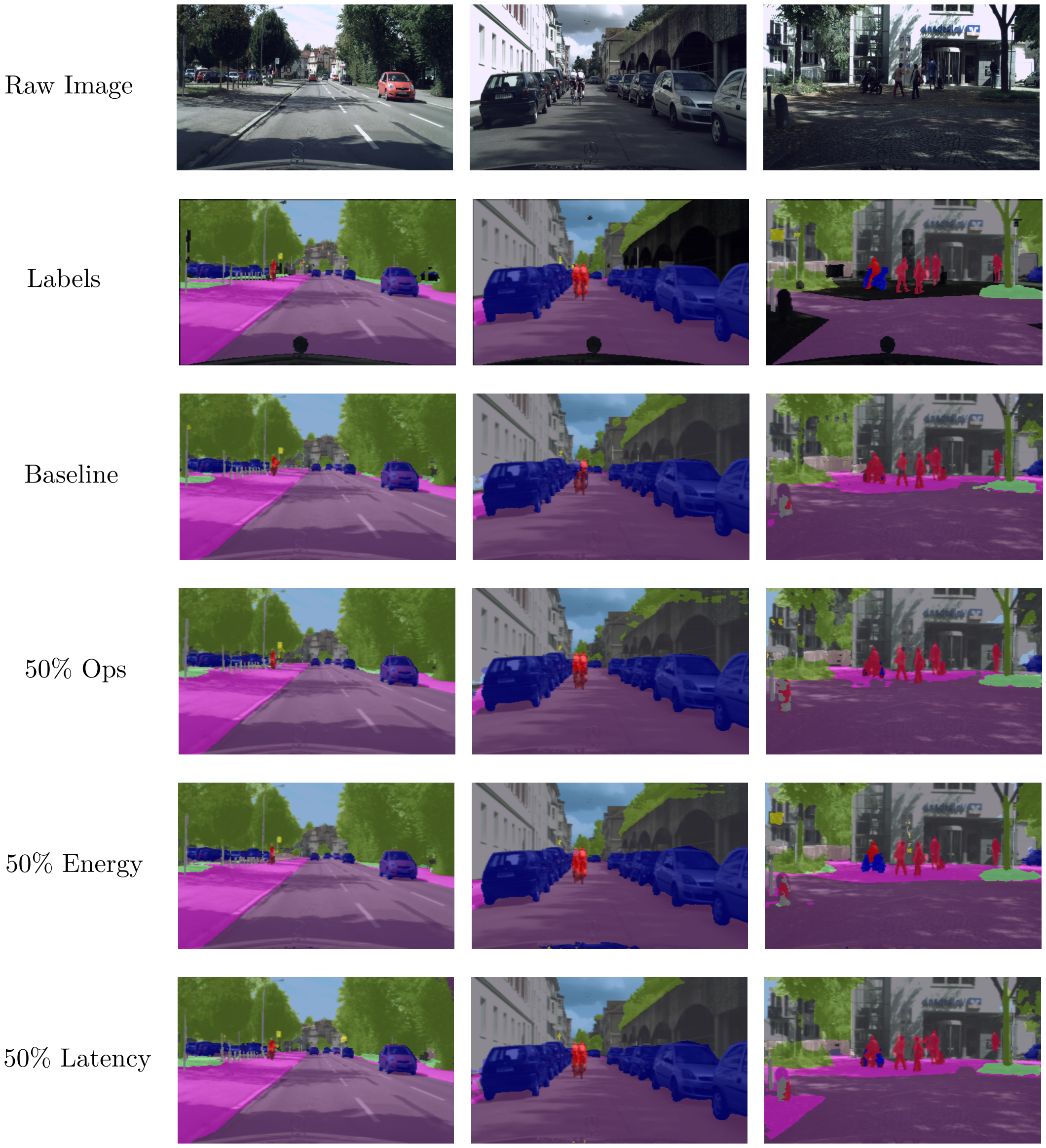


Figure 12 Qualitative results for pruned models on different scenarios in the CityScapes dataset. Black regions are unlabeled in the original dataset.

6.6 Results Summary and Discussion

In this section, we compare HW-Flow with other channel pruning works proposed in literature. Table 10 details various pruning configurations of ResNet variants trained on CIFAR-10 (ResNet20, ResNet56) and ImageNet (ResNet18, ResNet50), evaluated on an Eyeriss-like 256 PE accelerator. The table also includes other pruning methods and baseline models for reference. FPGM [15] and Channel-Pruning [14] do not consider HW metrics as optimization targets or constraints, but rather limit the compressed models only to be efficient with respect to computational complexity. Due to a lack of information given by the authors, the estimation of the energy and the latency is not possible for these works.

Table 10 Constrained and balanced pruning configurations using HW-Flow on ResNet variants, compared to other works in literature. HW estimates measured on Eyeriss-like-256.

Prune configuration	**Acc**	**PR**	**Memory**	**Energy**	**Latency**
(< `reward` >;< `level` >)	[%]	[%]	[MB]	[$\times10^9$]	[$\times10^3$cycles]
ResNet20					
Baseline (not pruned)	92.48	-	0.54	1.22	765
FPGM [15] (HW agnostic)	91.09	42.20	-	-	-
Ops Constrained [16]; Coarse	91.78	50.00	0.33	0.82	464
DRAM Constrained (Ours); Mid	90.78	70.87	0.27	0.43	236
Energy Constrained (Ours); Fine	91.46	56.12	0.35	0.61	359
Latency Constrained (Ours); Fine	90.53	48.55	0.35	0.66	383
ResNet56					
Baseline (not pruned)	93.59	-	1.69	3.76	2350
FPGM [15] (HW agnostic)	92.89	52.60	-	-	-
Channel-pruning [14] (proxy)	91.80	50.00	-	-	-
Ops Constrained [16]; Coarse	93.03	50.00	1.21	2.08	1219
DRAM Constrained (Ours); Mid	91.82	63.84	1.21	1.50	862
Energy Constrained (Ours); Fine	93.14	54.00	1.11	1.88	1159
Latency Constrained (Ours); Fine	93.24	50.89	1.15	2.05	1176
ResNet18					
Baseline (not pruned)	68.33	-	23.34	64.85	37796
FPGM [15] (HW agnostic)	67.81	41.80	-	-	-
Ops Constrained [16]; Coarse	67.66	50.00	16.94	32.89	18906
DRAM Constrained (Ours); Mid	66.38	54.46	15.94	30.17	16755
Energy Constrained (Ours); Fine	66.58	50.63	14.69	32.32	18280
Latency Constrained(Ours); Fine	66.92	49.70	16.61	33.62	18889
ResNet50					
Baseline (not pruned)	76.06	-	51.00	361.12	206873
FPGM [15]	74.83	53.50	-	-	-
Channel-pruning [14] (proxy)	72.30	50.00	-	-	-
Ops Constrained [16]; Coarse	73.25	50.00	22.67	178.55	103968
DRAM Constrained (Ours); Mid	72.17	58.61	17.16	148.67	86411
Energy Constrained (Ours); Fine	73.69	49.82	24.12	180.91	104411
Latency Constrained(Ours); Fine	74.35	49.68	25.62	180.93	103576

(violated constraint) (matched constraint)

The accuracy and HW-estimates for AMC [16] are re-implemented by constraining the `OPs` in HW-Flow's *Coarse*-level estimation. We observe that constraining DRAM accesses by 50% using *Mid*-level estimation demands higher pruning rate as there is little room for optimizing the HW schedule. This results in accuracy degradation compared to other pruning configurations from other target constraints (see DRAM constrained pruning for ResNet56, ResNet50 in Table 10). HW-Flow is able to constrain energy and latency precisely to 50% of its baseline metrics by using *Fine*-level HW estimates during the pruning process. For ResNet50, the energy and latency constrained solutions by HW-Flow produce 0.44% and 1.10% better prediction accuracy compared to the work in AMC (OPs constrained). AMC also prunes the CNNs based on latency, but it is only limited to general-purpose HW platforms (Pixel 1 and TitanX GPU). We should note that HW-Flow can prune CNN models at different HW abstraction levels and with a customizable, accurate HW optimizer/modeler, thus allowing for a HW/CNN co-design approach.

7 Conclusion

Optimization of CNNs and the design of resource-constrained HW platforms go hand in hand. In this paper, we propose HW-Flow, a framework for optimizing and exploring CNN models based on three levels of hardware abstraction: *Coarse*, *Mid* and *Fine*. We propose analytical search techniques to systematically traverse through the scheduling and mapping space, thereby generating accurate HW estimates. We show that the pruning rate is an inaccurate proxy metric for HW efficiency. With HW-aware pruning using *Fine*-level estimates, HW-Flow achieved ×2 energy and latency reduction with minimal loss in prediction accuracy compared to its baseline unpruned models. We extend the investigation to segmentation tasks, where observations on pruning rates of decoder and ASPP blocks were made with respect to the pruning target. DeepLabv3's energy and latency were reduced by ∼50%, while improving the accuracy of the baseline, over-parameterized model. HW-Flow can prune CNN models at different HW abstraction levels and with a customizable and accurate HW modeling technique, facilitating a HW-CNN codesign approach.

References

1 Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware. In *International Conference on Learning Representations (ICLR)*, 2019. URL: `https://dblp.org/rec/conf/iclr/CaiZH19.bib`.

2 C. Chen, A. Seff, A. Kornhauser, and J. Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 2722–2730, December 2015. `doi:10.1109/ICCV.2015.312`.

3 Liang-Chieh Chen, Yukun Zhu, George Papandreou, Florian Schroff, and Hartwig Adam. Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation. In *The European Conference on Computer Vision (ECCV)*, Cham, 2018. Springer International Publishing. `doi:10.1007/978-3-030-01234-2_49`.

4 Y. Chen, J. Emer, and V. Sze. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2016. `doi:10.1109/ISCA.2016.40`.

5 Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The Cityscapes Dataset for Semantic Urban Scene Understanding. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016. URL: `https://dblp.org/rec/journals/corr/CordtsORREBFRS16.bib`.

6 Yann Le Cun, John S. Denker, and Sara A. Solla. Optimal Brain Damage. In *Advances in Neural Information Processing Systems (NeurIPS)*. Morgan Kaufmann Publishers Inc., 1990.

7 Xiaoliang Dai, Peizhao Zhang, Bichen Wu, Hongxu Yin, Fei Sun, Yanghan Wang, Marat Dukhan, Yunqing Hu, Yiming Wu, Yangqing Jia, Peter Vajda, Matthew Uyttendaele, and Niraj K. Jha. Chamnet: Towards efficient network design through platform-aware model adaptation. *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019. `doi:10.1109/CVPR.2019.01166`.

8 Xuanyi Dong and Yi Yang. Nas-bench-201: Extending the scope of reproducible neural architecture search. In *International Conference on Learning Representations*, 2020. URL: `https://openreview.net/forum?id=HJxyZkBKDr`.

9 Alexander Frickenstein, Manoj-Rohit Vemparala, Nael Fasfous, Laura Hauenschild, Naveen-Shankar Nagaraja, Christian Unger, and Walter Stechele. Alf: Autoencoder-based low-rank filter-sharing for efficient convolutional neural networks. In *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*, (DAC), 2020. `doi:10.1109/DAC18072.2020.9218501`.

10 Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic Network Surgery for Efficient DNNs. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NeurIPS)*. Curran Associates, Inc., 2016. URL: `https://dblp.org/rec/conf/nips/GuoYC16.bib`.

11 Song Han, Jeff Pool, John Tran, and William Dally. Learning both Weights and Connections for Efficient Neural Network. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors,

Advances in Neural Information Processing Systems (NeurIPS). Curran Associates, Inc., 2015.

12 Babak Hassibi, David G. Stork, Gregory Wolff, and Takahiro Watanabe. Optimal Brain Surgeon: Extensions and Performance Comparisons. In *Advances in Neural Information Processing Systems (NeurIPS)*, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc. doi:10.1109/ICNN.1993.298572.

13 K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016. doi:10.1109/CVPR.2016.90.

14 Y. He, X. Zhang, and J. Sun. Channel Pruning for Accelerating Very Deep Neural Networks. In *IEEE International Conference on Computer Vision (ICCV)*, 2017. doi:10.1109/ICCV.2017.155.

15 Yang He, Ping Liu, Ziwei Wang, Zhilan Hu, and Yang Yang. Filter Pruning via Geometric Median for Deep Convolutional Neural Networks Acceleration. *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018. doi:10.1109/CVPR.2019.00447.

16 Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. AMC: AutoML for Model Compression and Acceleration on Mobile Devices. In *European Conference on Computer Vision (ECCV)*, 2018. doi:10.1007/978-3-030-01234-2_48.

17 Qiangui Huang, Shaohua Kevin Zhou, Suya You, and Ulrich Neumann. Learning to Prune Filters in Convolutional Neural Networks. *IEEE Winter Conference on Applications of Computer Vision (WACV)*, 2018. doi:10.1109/WACV.2018.00083.

18 Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

19 Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images, 2009. University of Toronto.

20 Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

21 Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems (NeurIPS)*. Curran Associates, Inc., 2012. doi:10.1145/3065386.

22 Chaojian Li, Zhongzhi Yu, Yonggan Fu, Yongan Zhang, Yang Zhao, Haoran You, Qixuan Yu, Yue Wang, Cong Hao, and Yingyan Lin. {HW}-{nas}-bench: Hardware-aware neural architecture search benchmark. In *International Conference on Learning Representations*, 2021. URL: https://openreview.net/forum?id=_0kaDkv3dVf.

23 Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

24 Shuying Liu and Weihong Deng. Very deep convolutional neural network based image classification using small training sample size. In *2015 3rd IAPR Asian Conference on Pattern Recognition (ACPR)*, pages 730–734, 2015. doi:10.1109/ACPR.2015.7486599.

25 Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

26 Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM workshop on hot topics in networks*, pages 50–56, 2016.

27 A. Parashar, P. Raina, Y. S. Shao, Y. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019. doi:10.1109/ISPASS.2019.00042.

28 S. Pereira, A. Pinto, V. Alves, and C. A. Silva. Brain tumor segmentation using convolutional neural networks in mri images. *IEEE Transactions on Medical Imaging*, 35(5):1240–1251, May 2016. doi:10.1109/TMI.2016.2538465.

29 Martin Riedmiller and Thomas Gabel. On experiences in a complex and competitive gaming domain: Reinforcement learning meets robocup. In *2007 IEEE Symposium on Computational Intelligence and Games*, pages 17–23, 2007.

30 Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3), 2015. doi:10.1007/s11263-015-0816-y.

31 Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Networks. In *ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, ISCA '18. IEEE Press, 2018. doi:10.1109/ISCA.2018.00069.

32 V. Sze, Y. Chen, T. Yang, and J. S. Emer. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE (Volume: 105, Issue: 12)*, 105(12), November 2017.

33 Christian Szegedy, W. Liu, Y. Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, D. Erhan, V. Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

34 Mingxing Tan and Quoc Le. EfficientNet: Rethinking model scaling for convolutional neural networks. In *Proceedings of the 36th International Conference on Machine Learning*, 2019.

35 F. Tu, S. Yin, P. Ouyang, S. Tang, L. Liu, and S. Wei. Deep convolutional neural network architecture with reconfigurable computation patterns. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(8):2220–2233, 2017. doi:10.1109/TVLSI.2017.2688340.

36 Manoj Rohit Vemparala, Nael Fasfous, Alexander Frickenstein, Sreetama Sarkar, Qi Zhao, Sabine Kuhn, Lukas Frickenstein, Anmol Singh, Christian Unger, Naveen Shankar Nagaraja, Christian Wressnegger, and Walter Stechele. Adversarial robust model compression using in-train pruning. *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 66–75, 2021.

37 R. Venkatesan, Y. Shao, Miaorong Wang, Jason Clemons, S. Dai, M. Fojtik, Ben Keller, Alicia Klinefelter, N. Pinckney, Priyanka Raina, Y. Zhang, B. Zimmer, W. Dally, J. Emer, Stephen W. Keckler, and B. Khailany. Magnet: A modular accelerator generator for neural networks. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019. doi:10.1109/ICCAD45719.2019.8942127.

38 Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. HAQ: Hardware-Aware Automated Quantization With Mixed Precision. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

39 Marco A Wiering. Multi-agent reinforcement learning for traffic light control. In *Machine Learning: Proceedings of the Seventeenth International Conference (ICML'2000)*, pages 1151–1158, 2000.

40 Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Péter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 10726–10734, 2019.

41 T. Yang, Y. Chen, and V. Sze. Designing Energy-Efficient Convolutional Neural Networks Using Energy-Aware Pruning. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017. doi:10.1109/CVPR.2017.643.

42 Tien-Ju Yang, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, and Hartwig Sze, Vivienneand Adam. NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications. In *The European Conference on Computer Vision (ECCV)*. Springer International Publishing, 2018. URL: https://dblp.org/rec/journals/corr/abs-1804-03230.bib.

43 Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, Christos Kozyrakis, and Mark Horowitz. Interstellar: Using halide's scheduling language to analyze dnn accelerators. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 369–383, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3373376.3378514.

44 C. Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2016. doi:10.1145/2966986.2967011.

45 Xingyi Zhou, Dequan Wang, and Philipp Krähenbühl. Objects as points. In *arXiv preprint*, 2019. arXiv:1904.07850.