ConfuciuX: Autonomous Hardware Resource Assignment for DNN Accelerators using Reinforcement Learning

Sheng-Chun Kao
Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA
felix@gatech.edu

Geonhwa Jeong
Computer Science
Georgia Institute of Technology
Atlanta, GA
geonhwa.jeong@gatech.edu

Tushar Krishna
Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA
tushar@ece.gatech.edu

***Abstract*—DNN accelerators provide efficiency by leveraging reuse of activations/weights/outputs during the DNN computations to reduce data movement from DRAM to the chip. The reuse is captured by the accelerator's dataflow. While there has been significant prior work in exploring and comparing various dataflows, the strategy for assigning on-chip hardware resources (i.e., compute and memory) given a dataflow that can optimize for performance/energy while meeting platform constraints of area/power for DNN(s) of interest is still relatively unexplored. The design-space of choices for balancing compute and memory explodes combinatorially, as we show in this work (e.g., as large as $O(10^{72})$ choices for running MobileNet-V2), making it infeasible to do manual-tuning via exhaustive searches. It is also difficult to come up with a specific heuristic given that different DNNs and layer types exhibit different amounts of reuse.**

In this paper, we propose an autonomous strategy called ConfuciuX to find optimized HW resource assignments for a given model and dataflow style. ConfuciuX leverages a reinforcement learning method, REINFORCE, to guide the search process, leveraging a detailed HW performance cost model within the training loop to estimate rewards. We also augment the RL approach with a genetic algorithm for further fine-tuning. ConfuciuX demonstrates the highest sample-efficiency for training compared to other techniques such as Bayesian optimization, genetic algorithm, simulated annealing, and other RL methods. It converges to the optimized hardware configuration 4.7 to 24 times faster than alternate techniques.

***Index Terms*—DNN Accelerator; Machine Learning; Reinforcement Learning; Genetic Algorithm**

I. Introduction

Deep neural networks (DNNs) are being deployed into many real-time applications such as autonomous driving, mobile VR/AR, and recommendation systems. However, DNNs are often strictly constrained by end-to-end latency or energy. This has opened up extensive research on computationally efficient DNN models [62], [77] and hardware accelerators [3], [16], [20], [34], [39].

The architecture of DNN accelerators is determined by two key components: *dataflow* style and total *HW resources*. The dataflow comprises the computation order, parallelization-strategy, and tiling strategy employed by the accelerator [16], [38]. The HW resources comprise of the total on-chip compute

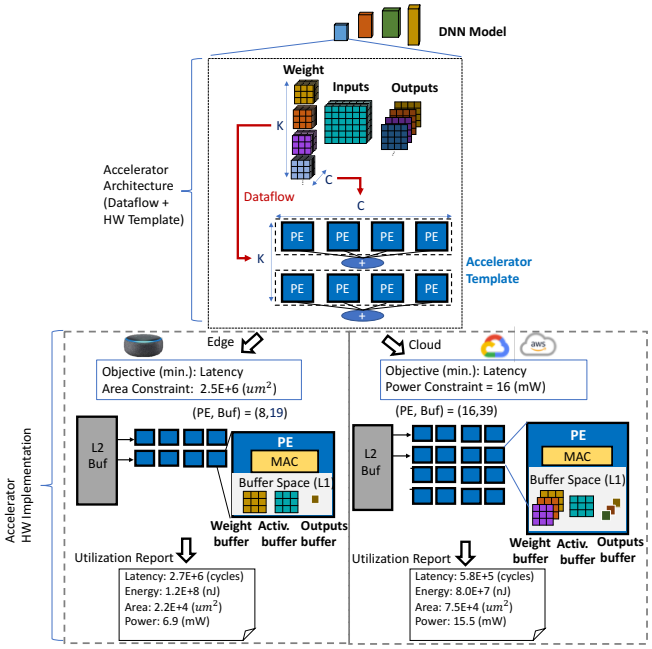


Fig. 1: Different HW resource combinations with the same NVDLA-style dataflow.

(hereby referred to as "PEs") and on-chip memory (hereby referred to as "Buffers"). The underlying network on chip (NoC) bandwidth, and the corresponding implementation [16], [39] and area depends on dataflow-style and assigned HW resources. For the same dataflow strategy, multiple resource assignments are possible, as shown in Figure 1. The dataflow and/or the HW resources are either fixed at design-time (which is the common-case), or can be tuned at compile time (if the accelerator is reconfigurable, such as CGRA-based [39] or FPGA [92]).

A lot of previous research has focused on designing efficient dataflow strategies to extract reuse. For e.g., NVDLA [3], Eyeriss [16], and ShiDianNao [20] are examples of DNN accelerators that employ different dataflow strategies. Frameworks like MAESTRO [38] and Timeloop [52] exist to contrast the

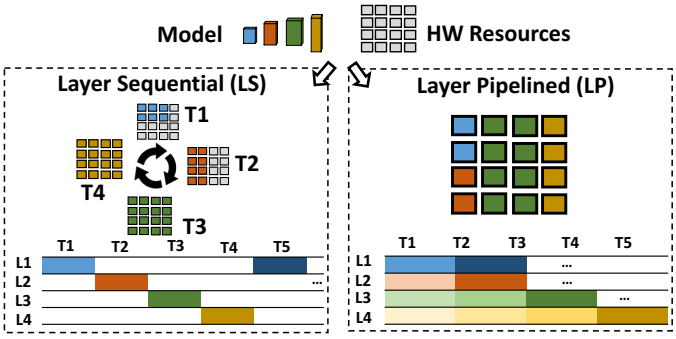


Fig. 2: DNN deployment scenarios and corresponding HW assignments. In Layer Sequential (LS), each layer of the model is mapped one by one on the entire accelerator, with all on-chip compute and memory assigned to it; in Layer Pipelined (LP), the entire model is mapped and run in a pipelined manner, with the compute and memory partitioned across all layers.

performance benefits of various dataflows. Most accelerators choose a dataflow strategy based on the expected dimensions and shapes of the DNNs they will run. For example, the NVDLA [3] dataflow keeps weights stationary at PEs, and parallelizes across input channels and output channels, as shown in Figure 1, optimizing for mid and late layers of many CNNs like ResNet [27] that exhibit this property. The Eyeriss [16] dataflow parallelizes across the activation and filter rows, and keeps filter rows stationary at the PEs. Reconfigurable accelerators like MAERI [39] allow the dataflow strategy to be configured for every layer [95].

Given a dataflow, the assignment of HW resources is the next crucial part of the DNN accelerator design process. In fact, for the same dataflow, different choices for HW resources can lead to drastically different latency and energy for a given DNN, as we show later in Figure 4. Some recent studies have shown that HW resource assignment plays a more important role in determining the accelerators' performance than its dataflow [89]. However, determining the policy for assigning HW resources is still very much an open problem, with prior works on HW Design-Space Exploration almost exclusively relying on exhaustive searches [15], [31], [33], [38], [42], [68], [82], [92]–[94].

The focus of this work is on the aforementioned HW resource assignment problem. The HW resource assignment depends on how they will be used by the DNN during runtime. We consider two deployment scenarios in this work, shown in Figure 2. Layer Sequential (LS) involves mapping and running the DNN layer by layer on the accelerator, while Layer Pipelined (LP) maps and runs the entire DNN model over the accelerator. The LS approach is typically leveraged in cloud settings for larger models [27] that do not fit on-chip, while the LP approach is popular when running smaller optimized models [62], [75], [77] on IoT devices.

The HW resource assignment problem is an optimization problem where the design goal is to achieve an objective such as minimum end-to-end latency or energy, while meeting some platform (IoT/Cloud) constraints such as maximum power or chip area. The design-space of valid solutions is non-trivial. Consider an LP deployment; suppose we have a total of P PEs and B buffers that fit within the area/power budget and need to be divided among N layers of the DNN. Assuming each layer gets at least one PE and one buffer, the number of combinations for PEs and buffers is $\binom{P-1}{N}$ and $\binom{B-1}{N}$ respectively [1]. This makes the total possible design choices $\binom{P-1}{N} \times \binom{B-1}{N}$, which is $O(10^{72})$ for an accelerator with 128 PEs, 128 buffers running the 52-layer MobileNet-V2. This design-space is nearly impossible to enumerate to search exhaustively for an optimum solution, as we discuss in Section II.

In this work, we develop an autonomous mechanism to efficiently search through the HW design-space. Figure 3 shows an overview of our proposed workflow called ConfuciuX. It takes the target model, platform constraint, deployment scenario (LS or LP), and optimization objective (latency/energy) as input, and determines an optimized HW assignment strategy (number of PEs and buffers). ConfuciuX leverages reinforcement learning (RL) to perform a global coarse-grained search, followed by a genetic algorithm (GA) for fine-grained tuning.

Recently RL has been demonstrated within compilers/mappers [7], [24], [46], [51] for tiling and mapping DNNs over accelerators. ConfuciuX focuses on leveraging RL for exploring the search space during accelerator design.

We evaluate ConfuciuX on popular DNN models, including MobileNet-V2 [62], ResNet-50 [27], and MnasNet [76], GNMT [85], Transformer [80] and NCF [28]. We evaluate our HW resource assignment method under three dataflow styles, NVDLA-style, Eyeriss-style, and ShiDianNao-style[1]. We evaluate both cloud and IoT device platform constraint setting. We also demonstrate a joint search for dataflow and HW assignments. We contrast our approach against other optimization mechanisms including Genetic Algorithm [29], simulated annealing [35], Bayesian optimization [54] and other state-of-the-art RL algorithms [23], [25], [43], [47], [66], [86] and observe that it consistently outperforms alternate schemes both in terms of solution quality (latency/energy) and search time (4.7 to 24 $\times$ faster).

The paper is organized as follows. Section II provides relevant background on DNN accelerators and relevant optimization methods; Section III describes ConfuciuX in detail; Section IV presents comprehensive evaluations; Section V presents related work and Section VI concludes.

II. Background and Motivation

A. DNN Accelerator Architecture

We discuss and define some key terms (highlighted in bold) that we will use throughout the paper.

*1) **Hardware Resources**:* Spatial DNN accelerators comprise an array of Processing Elements (called ***PE*** in this paper), as shown in Figure 1. Each PE has a MAC to compute partial

[1]We refer to them as -style since we follow the dataflow behavior part of these accelerators but allow flexibility in varying free dimensions (number of PEs and tile-sizes) at design/compile time.

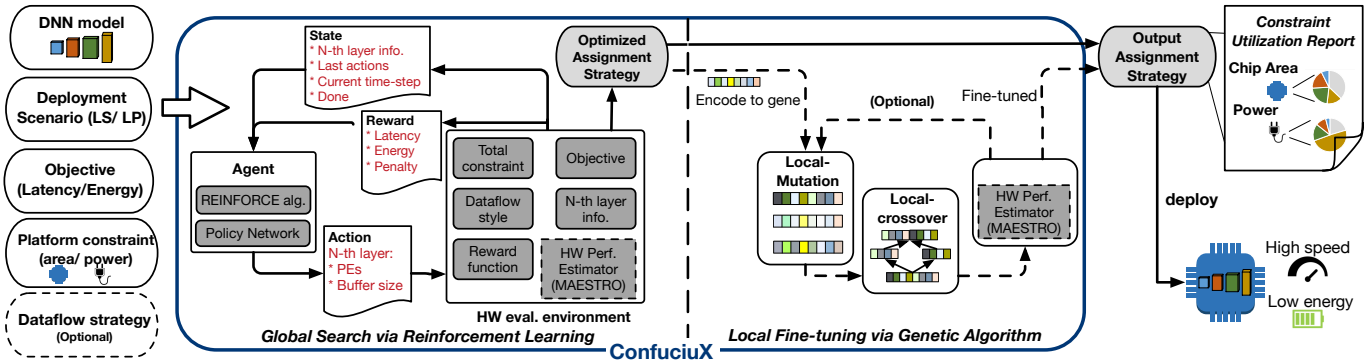


Fig. 3: Overview of ConfuciuX.

sums, and local (aka "L1") buffers (called ***Buffer*** in this paper) to store weights, activations, and partial sums. The accelerators also house a global shared (aka "L2") buffer to prefetch activations and weights from DRAM for the next tile of computation that will be mapped over the PEs and L1 buffers. Networks-on-Chip (NoCs) are used to distribute operands from the global L2 buffer to the L1 buffers in the PEs, and collect the partial or full outputs and write them back to the global L2 buffer.

*2) **Dataflow**:* The mechanism for orchestrating data from the global SRAM to the local PE buffers (i.e., computation order, parallelization strategy across PEs, and tiling strategy) is called *dataflow*. For example, accelerators like NVDLA [3], Eyeriss [17], ShiDianNao [20], TPU [34] all employ unique dataflow strategies [16], [38].

*3) **Design-Point**:* In this work, we assume that the number of PEs and buffers are free-variables[2] that can be tuned independently in an accelerator during design-time/compile-time (depending on the deployment scenario discussed later in Section II-C). We call each combination of (PE, buffer) as a unique ***design-point*** in this paper. Given a specific dataflow, each design-point in-turn determines the size and number of other components within the accelerator. For e.g., the number of PEs and L1 buffer sizes determine the minimum size of the global L2 buffer to hold the next tile of *unique data* that will need to be sent to the PE buffers [38], [52]. The L2 can then be sized to be double this value to prefetch the next tile from DRAM while the current one is being processed. Similarly, the design-point also determines the NoC bandwidth for stall-free distribution of operands to the PEs and collection of outputs [16], [39] for that dataflow. Thus, we choose only the PE and buffer size as the independent design-parameters in this work. It is certainly possible to let the L2 size and NoC bandwidth also be independent parameters, but this could lead over-provisioning (i.e., under-utilization) or under-provisioning (i.e., stalls) [38].

[2]The number of buffers depends on the maximum tile size of weights/inputs/outputs that the accelerator supports in each PE [38]. In this work, we control the buffer size by changing the tile size for filters.

B. DNN Accelerator Performance and Cost Modeling

The overall runtime, throughput and energy-efficiency of a DNN accelerator depends on three aspects: DNN model, mapping (dataflow and tile sizes), and HW resources [38]. We briefly discuss this cross dependence.

- **DNN Models.** There are myriads of DNN models and most of them are built using different combinations of some common layers. Convolutional layers (2D/depth-wise/point-wise) dominate in DNNs like ResNet-50, MobileNet-V2 [62], and InceptionNet [74] targeting image processing tasks. Fully connected layers or MLPs are often used as the last layer in many DNNs models, as hidden layers of RNNs, and in language models [57] and machine translation [85]. Different layer types expose different amounts of data reuse opportunities, which can be exploited by DNN accelerators depending on the mapping.
- **Mapping.** A mapping [38], [52] refers to the dataflow and specific tile sizes. The dataflow is the mechanism for reusing data across time (via buffers) and space (over wires). The tile sizes are bound by the L1 and L2 buffer sizes within the accelerator. The total number of tiles depends on the DNN model size and the dataflow strategy.
- **Hardware Resources.** The total number of PEs in the accelerator determines peak throughput, while the buffer sizes in each PE determine the amount of reuse that each PE can exploit within the tile of computation mapped on to it in each time iteration.

Cost Models. The interdependence between DNN model layer shape, mapping strategy and hardware resources is captured by cost models like MAESTRO [38] and Timeloop [52] that can *analytically* determine the reuse across time/space and accordingly estimate the runtime and utilization. These cost models can also estimate the area and power of the accelerator for a given design-point.

C. DNN Model Deployment on Accelerators

We focus on two DNN model deployment scenarios illustrated in Figure 2.

Layer Sequential (LS). We use the same underlying architecture to run the DNN model layer-by-layer. The specific (PE, buffer) design-point is chosen at design-time by some heuristic

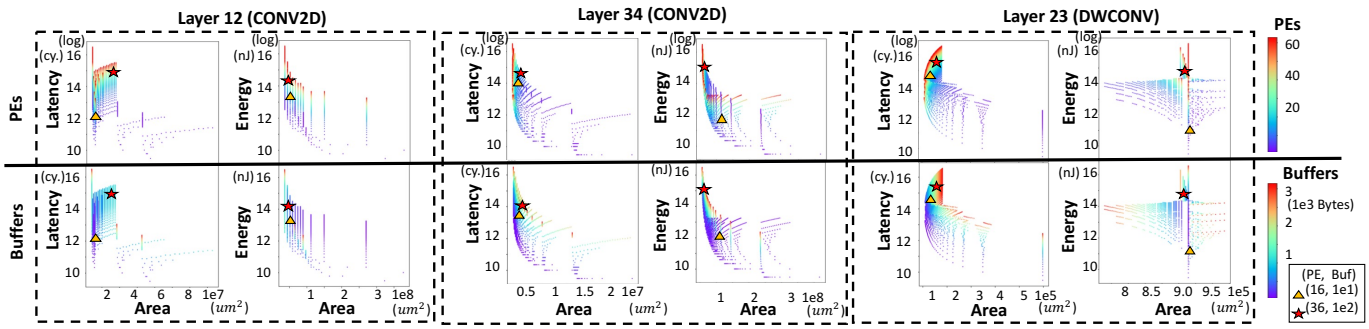


Fig. 4: Hardware design space for an accelerator running a few example layers from MobileNet-V2 using NVDLA-style dataflow in the MAESTRO cost model [38]. Each dot represents a design-point (number of PEs, L1 buffer per PE). For the same design-point, top half plots the number of PEs and bottom half plots the size of L1 buffers (in bytes). The corresponding latency, energy, and area are shown. Star and triangle icons represent two same design-points, and highlight how how their performance changes across different layers. The design-space for a given area-budget and/or latency/energy target is extremely large, and no design-point is optimal for all cases. (DWCONV: Depth-wise CONV.)

such as one that performs the best for most layers of the target DNNs. Naturally, over-provisioning and under-utilization may happen for some of the layers during deployment since they favor different HW resource configuration [11], [21], [22], [63]. We quantify this further in Section IV.

Layer Pipelined (LP). With the advancement of technology, more computation logic can sit in a single chip. Many efficient models are being designed to fit completely onto the chip for embedded platforms [62], [77]. LP maps and runs the entire DNN model over the accelerator. For this model, we assume an underlying accelerator can heterogeneously partition the (PE, buffer) resources at either design-time (e.g., ASIC [9], [90], [96]) or compile-time (e.g., CGRA [39]/FPGA [92]). The challenge becomes finding the optimum (PE, buffer) distribution for each layer, which is crucial for maximizing performance [15], [31], [33], [42], [68], [82], [93], [94].

D. Challenge: Design-Space for HW Resource Assignment

There can be myriad design points that fit the platform power/area constraint, each with drastically different performance/energy. As an example, we visualize the fine-grained hardware design space for a DNN accelerator in Figure 4, plotting the latency and energy when running three different layers of MobileNet-V2 on different accelerator design points with NVDLA-style dataflow. The numbers were obtained from MAESTRO [4]. Each point in the graphs is a design-point (i.e., {number of PEs, L1 Buffer per PE}). We sweep the PEs from 1 to 64, and number of filters that can be mapped from 1 to 800 (which in turn sweeps the L1 buffer size from 6B to 2800B). For the same design-point, the top half of Figure 4 shows the the number of PEs, and the bottom half shows the size of buffers. Each design-point leads to a unique latency, energy, and area consequence.

From Figure 4, we can conclude that for the same area, the range of possible latency and energy values is quite significant. This gets exacerbated when trying to find an optimal design point that works well across most layers in an LS deployment or for finding the combination of (PE, Buffer) per layer in an LP deployment. As mentioned in Section I, with just 128 PEs and 128 buffers, the design-space for MobileNet-V2 deployment is $O(10^{72})$, making it infeasible for any HW Design-Space Exploration (DSE) to sweep exhaustively. Most prior accelerator prototypes have picked specific design-points for their dataflow (e.g., 168 PEs in Eyeriss [17] and 64 PEs in ShiDianNao [20]) without exploring the design-space across different area/power constraints. This is the focus of this work.

E. Optimization Methods for Design-Space Exploration

The following optimization methods exist today for architects to perform Design-Space Exploration (DSE) and form our baselines.

Exhaustive search will lead to a global optimum, but is nearly impossible to sweep for vast design spaces. **Grid search** is an exhaustive search with a coarse-grain sampling step, which makes the process approachable.

Random search randomly samples design points in an unknown search space and keeps the best solution. It has been shown to be competitive for optimization problems in various fields [10], [41], [55], [61].

Simulated annealing [35] adds an exploitation step to random search (which is always exploring). It randomly samples and accepts points that improve the objective, but also with a certain probability accepts points that may worsen the objective. The probability is controlled by a hyper-parameter *temperature*. Higher temperature will increase the probability to accept worse points, causing more randomness, and vice versa. Simulated annealing is used for compiler optimization for CPU and software [98] and also tile and loop scheduling for DNN workload [13].

Genetic Algorithm (GA) [29] is a method where we encode the dimension of each design point as a gene. With all the dimensions specified, a design-point is called a genome. We initialize the algorithm with several randomly sampled design points (genomes) and these genomes form a generation. Then we evaluate the fitness of individuals of this generation. We keep well-performing individuals and use them to reproduce

the next generation with mutation and crossover. Generation by generation, GA will converge to an optimized point. STOKE [65] and TensorComprehensions [79] use GA to search the space of DNN code optimization.

Bayesian optimization [54] builds a surrogate for the objective and quantifies the uncertainty in that surrogate using a Bayesian machine learning technique. The optimization method can be constructed by the Gaussian process, Random forest, or Tree Parzen Estimator. It selects the next values to evaluate by applying criteria to the surrogate. It evolves the surrogate model and sampling criterion simultaneously. The concept is to limit the evaluation of the objective function by spending more time choosing the next values for sample efficiency. Some works use Bayesian optimization to search for DNN hyper-parameters [53], [69], [70].

F. Reinforcement Learning for Design-Space Exploration

Reinforcement Learning (RL) algorithms are often used in games [48], [78] as they are useful in sequential decisions. More formally, this is a Markov decision process (MDP). In this work, we show that determining the appropriate number of PEs and buffers for a series of DNN layers to minimizes the overall platform latency/energy while staying within an area or power budget can be viewed as a MDP. Therefore we find RL algorithms to be a promising approach for this problem to increase sample efficiency of the search, compared to baseline optimization methods that use no information from the current state.

Reinforcement Learning Terminology. The goal of an RL agent is to continuously interact with an **environment**, observe the current **state**, take one or more **actions**, observe the **reward** from the environment, and update its underlying **policy network**. With time, the policy network learns to predict actions that can maximize reward. We discuss our RL-based HW resource exploration next.

III. ConfuciuX

In this work, we cast the DNN accelerator resource assignment DSE challenge as a reinforcement learning (RL) problem using REINFORCE [73] for a global search, followed by a GA for local fine-tuning. Figure 3 demonstrates the workflow of ConfuciuX. We provide a high-level overview next. The inputs to ConfuciuX are the target DNN model, the deployment scenario, the optimizing objective, and the platform constraints, and it outputs an optimized HW resource assignment strategy, as shown in Figure 3. The first stage of ConfuciuX trains a RL agent to recommend an optimized assignment of PEs and buffers for a target DNN, platform constraint, and deployment scenario. The agent is trained by having it continuously generate resource assignments as "actions" which are evaluated by a detailed but fast analytical model for DNN accelerators called MAESTRO [4] that acts as the environment (Env). The "rewards" output by the environment are used to train the underlying policy network, with the aim of maximizing the reward. We incorporate platform constraints (area/power) as inputs to the environment to punish actions that violate the constraints. To speed up the search process, the RL agent searches through the HW assignments in a coarse-grained manner. Once it converges to an optimized strategy, we fine-tune the assignment further using GA. We discuss the various components of ConfuciuX next.

A. RL Agent

In our system, the RL agent processes the target DNN model in a layer-wise manner. We term the whole process (episode in RL parlance) as an epoch. We treat each layer as a different time-step. At each time-step, the agent makes two actions per-layer: the number of PEs and Buffers. It interacts with the environment to collect the rewards. We feed the rewards, along with the previous layer's actions to the policy function, to help the agent optimize sequential decisions. The policy network gets updated at the end of each epoch. An epoch terminates when the agent fails the constraint or successfully made $2N$ actions for a N-layer model.

*1) **Choice of RL Algorithm: REINFORCE**:* Modern RL algorithms [23], [25], [43], [47], [66], [86] typically use two underlying neural networks - an "actor" and a "critic". The actor formulates the policy for taking actions while the critic approximates the value function that predicts expected reward to help the training of policy. We experimented with a suite of RL algorithms for ConfuciuX and found that REINFORCE [73] works best. We show these results in Section IV-C3. REINFORCE only has an actor network (no critic), and updates its underlying policy network directly using rewards from the Env. Since the design space of HW resource assignments is extremely discrete and irregular, we observed that RL algorithms with critic networks fail to approximate the value function accurately and in turn disturb the policy learning process. We show this later in Section IV-C3.

*2) **Policy Network Architecture**:* The policy network in REINFORCE is a neural network tasked to learn the policy to maximize the probability of receiving better reward. We use an RNN as the policy network with one LSTM hidden layer of size 128. The reasoning behind an RNN-based network is as follows. We impose a hard constraint on the overall area (or power) consumption. Each action (PE, buffer) adds to area/power. Thus, any future action should depend on the previous action. The recurrent connections in the RNN capture this relationship and learn the constraint. We implemented and evaluated both RNN-based and MLP-based policy networks and provide a quantitative analysis in Section IV-G.

B. Observation (State)

We construct a 10 dimensional observation space. At t^{th} time step, the observation (O_t) is expressed as follows

$$O_t = (K_t, C_t, Y_t, X_t, R_t, S_t, T_t, A_t^{PE}, A_t^{Buffer}, t) \quad (1)$$

The layer shape (assuming convolutions) results in the first 7 dimensions[3]. K_t and C_t are number of output and input channels. Y_t and X_t are the size of Y and X axis of the input

[3]for other layers like MLP/GEMM, we use three dimensions (M,N,K) to describe the (M,K), (K,N) and (M,N) matrices.

TABLE I: The level values of action pair.

	Action level values
PEs	1, 2, 4, 8, 12, 16, 24, 32, 48, 64, 96, 128
Buffers (e.g., NVDLA-style)	19, 29, 39, 49, 59, 69, 79, 89, 99, 109, 119, 129

activations. R_t and S_t are the size of Y and X axis of the weight kernel. T_t is the indicator of layer type such as CONV or DWCONV (Depth-wise CONV). A_t^{PE} and A_t^{Buffer} are the actions of the previous layer which we feed into the RNN controller. The last dimension t indicates t^{th} layer. Finally, we normalize all the dimensions of observation to the range of [-1, 1] to stabilize the training.

C. Action Space

At each time step, the agent makes an action pair (PE, Buffer), which formulates the action space. To efficiently step through the huge design space (Section II-D), the RL agent uses coarse-grained steps to navigate through it. In particular, we use $L = 12$ different values for the PEs and Buffers, as shown in Table I. We demonstrate the effect of L later in Section IV-G. The specific values for PE at each level are chosen by the marginal observed return of HW performance to the number of PEs. For example, increasing PE from 1 to 2 could potentially double the HW performance, while increasing PE from 64 to 65 would provide slight or mostly no improvement. We choose the Buffer size value at each level according to the input of dataflow-style at design time. In NVDLA-style, with 3×3 weight as an example, we dispatch the computation to each PE along K dimension (Figure 1). Each PE would receive k number of 3×3 weights, 3×3 corresponding inputs, and generate k number of outputs, which makes the buffer size $9 \times k + 9 \times 1 + 1 \times k$, where k=1, 2,..., 12, as shown in Table I. Note that once the RL agent converges, ConfuciuX uses fine-grained steps using GA to get to an optimized configuration, as described later in Section III-G.

D. Platform Constraint and Objective

Each action pair (PEs, Buffers) defines the per-layer power/area constraint consumption and the per-layer energy/latency cost, which are our optimization targets. The goal of the accelerator design process is to optimize the cost for running the entire model, while meeting the platform constraint.

Constraint. The accelerator is constrained by the budget of the targeted platform. We consider two categories of constraints: power and chip area. We have full flexibility to design architecture such as assigning a different number of PEs and Buffers or changing dataflow-style, as long as the design meets the constraint. In this paper, we evaluate power and area constraints across cloud and IoT platforms, as described in Section III.

Objective. We evaluate two design objectives in this work: minimum overall latency and minimum overall energy cost when the entire model is run on the accelerator, either via LS or LP. Other objectives can also be considered (say EDP or Power/Area for instance). While approaching the objective, the design should always fit the platform constraint. Minimizing the latency and energy is a non-trivial task since their dependence on the number of PEs and Buffers is not straight-forward. For e.g., increasing PEs would increase the level of parallelism; however, it would also increase the number of fetched data, which could potentially increase the latency. As for energy, increasing PEs and Buffers would increase the power; however, it could potentially decrease the energy because of the shorter execution time.

Co-optimization of layers. The RL agent is trained to be aware of the platform constraints. The agent should learn to optimize the resource assignment for each layer and the allocation of the constraint budget at hand to each layer simultaneously. We use RNN as the backbone of RL agent to enable it to memorize its entire epoch of decisions so that it could be aware of the consumption of the total budget. The Env checks the budget that is still left (L^{budget}) at every time step, and penalizes the RL agent once it is violated.

E. Reward Function

Reward. Since we are executing in a sparse reward domain, where the performance is only given at the end of the episode, we train the agent with a temporal layer-wise performance feedback for reward shaping. The sum of the layer-wise performance does not directly indicate the final entire model performance, which is our objective. However, it guides the RL agents.

We construct the reward function R as follows.

$$R = \begin{cases} P_t - P^{min}, \text{ if } L^{budget} \geq 0 \\ Penalty, \text{ otherwise} \end{cases} \quad (2)$$

P_t is the HW performance[4] of the current layer. P^{min} is the current lowest layer-wise performance across all time-steps and all epochs. This is tracked during the training process.

We find that the P^{min} term stabilizes the training. The insight behind it is as follows. First, as shown in Figure 1, the reward value for HW performance, such as number of cycles, can be extremely large, which can make the relative improvement seem insignificant across epochs. Thus, keeping a P^{min} across all epochs emphasizes the relative difference. Second, the term P^{min} makes the reward always positive while the platform constraint is not violated, which makes the RL agent easier to learn from positive reward and negative penalty.

Penalty. We penalize the RL agent when the resource constraint is violated. To teach the RL agent to forbid the failing point with reasonable penalty, we accumulate all the rewards experiences in this episode, and use negative of the accumulated value as a penalty. The reason is that the range of reward for different HW performance (latency, energy) can have an order of magnitude difference. Therefore, a threshold-based constant penalty [2], [56], [87], which is usually applied, is not feasible. Also, we need the penalty that is at the correct scale so that it is large enough to penalize the agent and small

[4]We use the term performance for generality. It could be latency, or energy, or any other objective we are minimizing.

enough to not deviate the learned policy too much once bad decision is made.

At the end of the episode, we normalize rewards in each time step to standard distribution and use the standardized reward to train the agent. We also apply a discount factor (d). We empirically found $d = 0.9$ is a generic good default value for this problem.

F. Interactive Environment (Env)

Structure. The Env is initialized with the target model(s), dataflow, platform constraint, and the optimizing objective (latency/energy). Env tracks the consumed constraints of each time step and the P^{min} across all episodes.

HW performance estimator (eval). We use MAESTRO [4], an open-source DNN accelerator microarchitectural model, to determine the performance of each accelerator design-point during the training process. MAESTRO takes the DNN model, dataflow-style, and HW configuration as an input. Internally, it estimates all possible reuse opportunities for the given dataflow and HW resources, and estimates statistics such as latency, energy, runtime, power, and area. MAESTRO's HW model assumes a spatial DNN accelerator with PEs, L1 buffers, a shared L2 buffer, and an NoC between the buffers. It can support any dataflow (specified via a data-centric DSL [38]). The number of PEs is an input parameter, while the L1 and L2 buffer sizes are estimated based on the tile-sizes for the dataflow. It supports both layer-wise and model-wise evaluation.

G. Local fine-tuning using GA

We use a two-stage optimization to search for a fine-grained solution, as shown in Figure 3. The first and major part is the RL based coarse-grained global search. The second is the Genetic Algorithm (GA) based fine-grain local search. We use two-stage optimization for efficiency, since increasing the level of actions, L by 1 would increase the design space by $(\frac{L+1}{L})^{2N}$. Using MobileNet-V2 and $L = 12$ as an example, we would increase the design space discussed in Section II-D by another 64 times.

RL shows higher sample efficiency and converges to better optimum point comparing to other optimization methods, as shown later in Section IV. GA is simple and fast, but converging to less optimum value comparing to RL or sometimes cannot converge. According to the observation of the behavior of GA, it sometimes fails to learn the constraint and optimize the objective simultaneously, leading to a great portion of populations actually violating the constraint, which pollute the genomes of the future generation. However, if we start GA with a good initialization and mutate/crossover genes carefully, which decreases the complexity of the problem, GA could reach good result. Therefore, GA becomes a good candidate as a second stage fine-tuning if we initialize it with the first-stage solution. Even though a continuous RL algorithm [23], [25], [43] could be another candidate for the second stage, we find that the problem complexity of the second stage is simple that GA is adequate to tackle it. The details of the GA algorithm are described next.

TABLE II: Platform constraint settings.

Platform Constraint	Descriptions
Unlimited	No constraint. Since we set each action to 12-level, we measured the maximum constraint (power/area) consumption, $C^{max}_{power/area}$, by evaluating entire model with uniform action pair (p_{12th}, b_{12th}).
Cloud	Loose constraint. We set the constraint at *50%* $C^{max}_{power/area}$.
IoT	Tight constraint. We set the constraint at *10%* $C^{max}_{power/area}$.
Extreme small IoT (IoTx)	Extremely tight constraint. We set the constraint at *5%* $C^{max}_{power/area}$.

Initialization. Assuming a DNN model with N layers, a design-point would include N actions for PEs and N actions for Buffers. We encode this design-point into a genome with $2N$ genes, where a gene represents an action for PE or Buffers. We initialize the first population with the genome formulated by the solution from the first (RL) stage.

Local mutation. We mutate the gene locally. We only mutate the gene by a step difference of the current value. For e.g., for a gene representing PE=64, we could mutate it to value in the range of [60, 68] when the step is 4. This conservative mutation can reduce the number of invalid genomes, which does not conform to the constraint, and assure we have good portion of valid parents to reproduce.

Local crossover. The crossover of two genomes is unlikely to conform to the constraint, since it can break the learnt relationship between HW resource assignment of each layer. For e.g., suppose we have parents A and B, both with good fitness and lie within the constraint. However, A tends to assign more HW resources on early layers and B tends to assign more HW resources on late layer. When we blend their genes for the next generation, the platform constraints might get violated by some children: a child with early genes from A and late genes from B may over-request HW resources for every layer, violating the constraints. Alternately, a child with early genes from B and late genes from A may under-request HW resources for each layer, leading to less performance. Thus, we crossover the genome locally within a parent by exchanging genes for (PE, Buffers) between two layers of a model. In other words, we pick two pairs of genes representing the (PE, Buffers) of two layers of a models and swap them. This conservative self-crossover preserves most of the learnt relationship between layer and resources and adds an exploration effect.

IV. Evaluations

A. Methodology

*1) **DNN Models**:* In our evaluations, we consider three CNN models with different complexity: MobileNet-V2 [62], MnasNet [75], and ResNet-50 [27]. We also evaluate three GEMM-based ML models: GNMT [85] for machine translation, transformer [80] for language understanding and NCF [28] for collaborative filtering.

*2) **Accelerator Platforms**:* We consider three different classes of platforms: Cloud server, IoT device and extremely small IoT, and, for comparison, an unconstrained platform as shown in Table II. We consider three dataflows: NVDLA-style [3] (-dla) (parallelizing K and C dim.), Eyeriss-style [16] (-eye) (parallelizing Y and R dim.), ShiDianNao-style

TABLE III: Converged solution of LP deployment.

Model	Obj. (min.): Latency Cstr.: Area	Optimization Results (cycles)		
		GA	PPO2	**Con'X (global)**
MbnetV2-dla	IoT	NAN	3.6E+07	**3.2E+07**
MbnetV2-eye	IoTx	NAN	4.1E+07	**3.7E+07**
MbnetV2-shi	IoTx	NAN	4.9E+07	**4.0E+07**
Mnasnet-dla	Cloud	**2.1E+07**	2.3E+07	**2.1E+07**
Mnasnet-eye	IoTx	NAN	4.1E+07	**3.9E+07**
Mnasnet-shi	IoTx	NAN	4.4E+07	**4.3E+07**
Resnet50-dla	Cloud	**2.2E+08**	2.4E+08	**2.2E+08**
Resnet50-eye	Cloud	2.3E+08	2.3E+08	**2.2E+08**
Resnet50-shi	Cloud	3.2E+08	3.4E+08	**3.1E+08**
GNMT-dla	IoTx	NAN	2.4E+08	**5.4E+07**
GNMT-eye	IoT	9.8E+07	1.4E+08	**9.8E+07**
GNMT-Shi	IoT	2.5E+10	**2.4E+10**	**2.4E+10**
Transformer-dla	IoTx	NAN	7.3E+08	**1.6E+06**
Transformer-eye	IoT	3.5E+06	6.6E+05	**1.9E+05**
Transformer-shi	IoT	7.3E+08	**7.2E+08**	**7.2E+08**
NCF-dla	IoTx	NAN	7.3E+07	**7.2E+07**
NCF-eye	Cloud	1.20E+06	1.4E+06	**1.1E+06**
NCF-shi	IoT	**6.6E+08**	**6.6E+08**	**6.6E+08**
Bold indicates the best results among GA, PPO2 and this work.				
NAN indicates that constraint (area) not met in Eps (5000) epochs.				

TABLE IV: Converged solutions after 5000 epochs for various optimization methods across four platforms with different constraints. DNN=MobileNet-V2, Dataflow=NVDLA-style, Deployment=LP

Objective (min.)	Constraint	Optimization Results (cycles)					
		Grid	Random	SA	GA	Bayes. Opt.	**Con'X (global)**
Latency	Area: Unlimited	5.3E+08	3.6E+07	6.2E+07	**2.1E+07**	3.7E+07	**2.1E+07**
Latency	Area: Cloud	5.3E+08	3.4E+07	9.6E+07	**2.1E+07**	3.7E+07	**2.1E+07**
Latency	Area: IoT	5.3E+08	7.1E+07	NAN	NAN	7.2E+07	**3.2E+07**
Latency	Area: IoTx	5.3E+08	NAN	NAN	NAN	NAN	**5.6E+07**
Latency	Power: Cloud	5.3E+08	3.6E+07	9.9E+07	**2.2E+07**	3.7E+07	**2.2E+07**
Latency	Power: IoT	5.3E+08	NAN	1.2E+08	NAN	1.6E+08	**3.7E+07**
Latency	Power: IoTx	5.3E+08	NAN	NAN	NAN	NAN	**5.6E+07**
Energy	Area: Unlimited	8.5E+09	1.8E+09	1.3E+09	**1.2E+09**	1.5E+09	**1.2E+09**
Energy	Area: Cloud	8.5E+09	1.8E+09	1.3E+09	**1.2E+09**	1.5E+09	**1.2E+09**
Energy	Area: IoT	8.8E+09	NAN	NAN	NAN	2.4E+09	**1.4E+09**
Energy	Area: IoTx	8.6E+09	NAN	NAN	NAN	NAN	**1.7E+09**
Energy	Power: Cloud	8.5E+09	1.8E+09	1.3E+09	**1.2E+09**	1.5E+09	**1.2E+09**
Energy	Power: IoT	8.6E+09	NAN	NAN	NAN	4.1E+09	**1.4E+09**
Energy	Power: IoTx	8.6E+09	NAN	NAN	NAN	NAN	**1.7E+09**
Results: Latency: (cycles), Energy: (nJ). **Bold** indicates the best results among algorithms.							
NAN indicates that constraint (area/power) not met in Eps (5000) epochs.							

[20] (-shi)(parallelizing Y and X dim). We use L=12 levels of action values for PE and Buffers, where $(p_{n_{th}}, b_{k_{th}})$ represents assigning n_{th}-level of PEs and k_{th}-level of Buffers.

*3) **Baseline Optimization/Search Methods**:* We evaluate the following optimization methods as baselines.

Grid search. We enumerate through the design space with the stride of s in the L=12 level, (e.g., $(p_{1_{th}}, b_{1_{th}})$, $(p_{1_{th}}, b_{(1+s)_{th}})$...). We set maximum epochs Eps. We emulate through the design space until the number of sampling points reached Eps.

Random search. We randomly sample Eps design points and keep the best solutions as a result.

GA [29]. The baseline is a general GA algorithm, not the specially designed local fine-tuning one as described in Section III-G. The GA is set with 100 population, and $\lceil \frac{Eps}{100} \rceil$ generations. The mutation rate and crossover rate is set as 0.05.

Simulated Annealing [35]. The simulated annealing is implemented with temperature of 10 with step size of 1 and adopted to discrete integer space.

Bayesian optimization [54]. We set the algorithm to run for Eps iterations, where Eps points are sampled by the algorithm. We adopt it to discrete integer space. We set the number of the optimizer to 5 for the Gaussian process, since we empirically find this setting has better performance.

State-of-the-art RL algorithms. We consider state-of-the-art RL algorithms that are successful in many control problems. We consider both continuous and discrete methods. We compare with **A2C** [47], **ACTKR** (Actor Critic using Kronecker-Factored Trust Region) [86], and **PPO2** [66]. Both continuous and discrete versions of the three algorithms are experimented. Across all the experiments, we found the discrete version converge to better value. Hence, we will only show the result of the discrete version in the comparisons table. We also consider **DDPG** [43], **SAC** [25], and **TD3** [23] in continuous space. All comparisons run for Eps epochs.

*4) **ConfuciuX (Global)**:* We only consider the first-stage global search, Con'X (global), throughout the comparisons against baseline methods, for fairness. The second-stage fine-tuning can be added on top of the first-stage results. The benefit of the second-stage is explicitly discussed in Section IV-E.

B. Per-layer study for LS deployment

We start by showing the HW performance of different action pairs $(p_{n_{th}}, b_{k_{th}})$ with 12 level of values each, which is Y-axis and X-axis in Figure 5. We sweep through $(p_{n_{th}}, b_{k_{th}})$ with exhaustive search and color it with their corresponding latency/energy value. Red indicates large latency/energy values while purple indicates small ones. For each layer, the contour is drastically different. Each layer require distinct action pairs $(p_{n_{th}}, b_{k_{th}})$ to reach optimal values (purple). The contour becomes a flat region when the PEs or Buffers are over-provisioned. Latency of layer-12 as an example, when PE is larger than the 9_{th} level and Buffer is larger than the 3_{rd} level, the latency remains the same because of over-provisioning. The two separate purple region in latency of Layer-34 indicates that there are two region of tiling size, which we map to the buffer, can optimize the latency. For Layer-23 (DWCONV), increasing the tile size of the mapping dimension (K) does not help because of the irrelevance of each output channel (K) in DWCONV. As for energy, larger number of PEs and Buffers can potentially decrease the energy because of shorter execution time as in layer-12 and layer-34. We can observe there are sweet spot for buffer size in Layer-23, where all the channel is mapped to one PE. At this end, increasing PE would not increase the energy, since extra PE will be idle. Also decreasing Buffers cause more times of fetching, which increase the energy consumption.

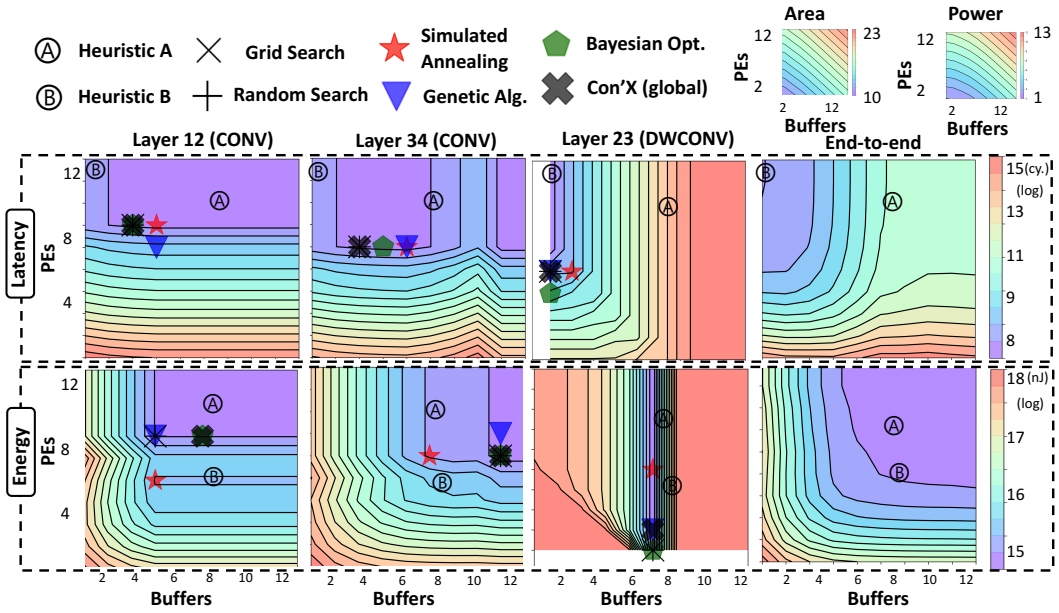


Fig. 5: Searching for per-layer PEs/Buffers configurations to optimize latency/energy with different techniques. Purple indicates lower (better) and red indicates higher (worse) latency/energy. **Heuristic A**: Determine the PEs/Buffers with the most compute-intensive layer (Layer-38) and apply the same configuration for all the layers. **Heuristic B**: Determine the PEs/Buffers by the configuration that optimizes end-to-end whole model latency/energy.

TABLE V: Comparison of search-time and converged solutions across state-of-the-art RL techniques.

Model	Objective (min.)	Constraint	A2C		ACKTR		PPO2		DDPG		SAC		TD3		**Con'X (global)**	
			Optimized Results	Search Time	Optimized Results	Search Time	Optimized Results	Search Time	Optimized Results	Search Time	Optimized Results	Search Time	Optimized Results	Search Time	Optimized Results	Search Time
MbnetV2	Latency	Area: IoT	5.4E+07	2:45	4.0E+07	12:01	3.6E+07	1:34	1.1E+08	24:20	4.7E+07	10:23	3.8E+07	8:22	**3.2E+07**	0:25
MbnetV2	Latency	Area: IoTx	9.6E+07	1:20	5.6E+07	3:55	**5.6E+07**	1:20	1.8E+08	11:41	1.0E+08	3:00	8.2E+07	2:34	**5.6E+07**	0:35
MbnetV2	Latency	Power: IoT	6.5E+07	1:49	4.7E+07	3:59	4.5E+07	2:13	1.7E+08	27:32	6.2E+07	10:09	1.8E+08	5:56	**3.7E+07**	0:43
MbnetV2	Latency	Power: IoTx	7.3E+07	1:00	6.6E+07	4:05	6.8E+07	1:15	6.3E+09	3:30	1.1E+08	2:30	7.6E+07	2:25	**5.6E+07**	0:32
MbnetV2	Energy	Area: IoT	1.8E+09	0:36	1.7E+09	3:55	1.5E+09	1:25	2.3E+09	3:10	2.0E+09	1:21	1.8E+09	3:34	**1.2E+09**	0:48
MbnetV2	Energy	Power: IoT	1.9E+09	0:31	1.8E+09	2:15	1.6E+09	0:57	2.0E+09	9:56	1.9E+09	1:40	**1.4E+09**	3:50	**1.4E+09**	0:53
ResNet50	Latency	Area: Cloud	3.8E+08	6:38	3.5E+08	16:51	2.4E+08	6:37	2.5E+08	25:27	2.5E+08	13:09	2.4E+08	8:36	**2.2E+08**	0:46
ResNet50	Latency	Power: Cloud	3.8E+08	6:38	3.5E+08	8:06	2.4E+08	6:47	2.5E+08	25:18	2.6E+08	8:05	2.4E+08	8:04	**2.3E+08**	0:46
ResNet50	Energy	Area: Cloud	1.4E+10	6:38	1.4E+10	14:35	9.1E+09	6:46	1.3E+10	25:11	1.1E+10	12:40	1.3E+10	9:38	**7.6E+09**	1:20
ResNet50	Energy	Power: Cloud	1.6E+10	6:45	1.3E+10	16:26	9.7E+09	6:57	1.3E+10	16:26	1.1E+10	12:12	1.3E+10	12:46	**7.6E+09**	1:05
MnasNet	Latency	Area: IoT	4.6E+07	5:04	3.7E+07	12:32	3.3E+07	5:08	6.4E+07	23:12	3.6E+07	9:29	7.2E+07	6:32	**2.8E+07**	0:27
MnasNet	Latency	Power: IoT	6.5E+07	5:23	4.3E+07	12:12	4.2E+07	5:40	9.8E+07	25:50	5.6E+07	13:04	9.1E+07	6:02	**3.5E+07**	0:42
MnasNet	Energy	Area: IoT	1.8E+09	0:58	1.6E+09	3:48	**1.4E+09**	1:43	2.1E+09	9:52	1.8E+09	3:25	1.7E+09	2:56	**1.4E+09**	0:30
MnasNet	Energy	Power: IoT	1.8E+09	0:36	1.8E+09	3:39	1.7E+09	1:15	2.9E+09	9:44	1.9E+09	2:50	2.7E+09	1:27	**1.4E+09**	0:39
Memory Overhead (MB)			5.3		5.3		5.3		13.9		16		18.1		2.1	

Results: Latency: (cycles), Energy: (nJ) Search time: (hrs:mins) **Bold** indicates the best results among algorithms.

Con'X consistently finds the optimal action pair for each layer, and its solution is as-good or better (fewer PEs and buffers for same latency or energy) than the baseline methods and two common heuristics. Figure 5 also shows that there is no action pair that suits all the layers. Thus, for a LS scenario, a designer can use Con'X to find optimal configurations for each layer, and then pick the one that provides optimum values across most layers.

C. LP Deployment

Next, we consider LP deployment (i.e., all layers of the model mapped on the accelerator) with platform constraints. For all the comparisons, we compare the algorithm performance by comparing their best solutions after $Eps = 5,000$ epochs.

*1) **Converged solutions across DNNs, Dataflows, and Platforms**:* We ran baseline optimization methods and RL algorithms for a suite of DNNs (CNN- and GEMM-based) with varying dataflow styles and platform constraints. The objective is set to minimize the latency of the entire model. Therefore the lower the reached value, the better the solution is. In the interest of space, we show the results with the best performing baselines, GA and PPO2, in Table III. GA can reach good optimized value when the constraint is loose (cloud), but it fails in some tight constraint cases (IoT, IoTx). Both PPO2 and Con'X(global) can find solutions in any type of constraint. Across all the experiments, Con'X(global) finds the solution with the same or better performance than PPO2 and GA.

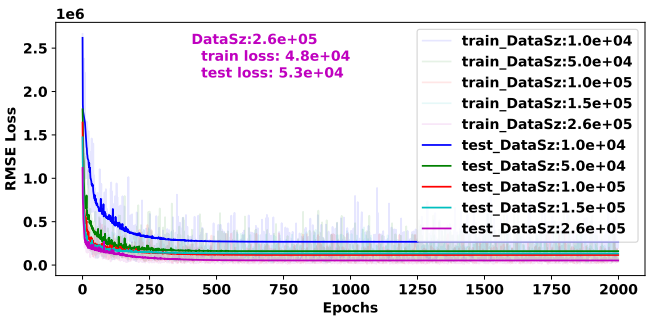
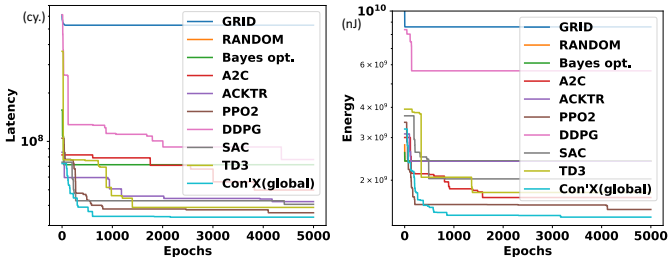


Fig. 6: The learning curve of the critic network.



(a) Obj.(min.): Latency, Cstr.: IoT area **(b) Obj.(min.): Energy, Cstr.: IoT area**

Fig. 7: The fast convergence and sample efficiency of Con'X (global).

*2) **Deep-dive with optimization methods**:* Table IV compares the solutions attained by various optimization methods and Con'X(global) for MobileNet-V2 under four platform constraints for a NVDLA-style accelerator. The objective is set to minimize the latency or energy of the entire model. Random, SA, and GA fail to come up with a feasible solution when faced with tight constraint (IoT). Also, Bayesian optimization fails in extreme tight constraint (IoTx). Con'X(global) successfully learns the constraint behavior and optimizes the objective together. Con'X(global) generates the most optimized design points with 86% lower latency and 70% lower energy, on average across baselines.

*3) **Deep-dive with RL algorithms**:* We compare Con'X(global) with other state-of-the-art RL algorithms in the same setting as the previous experiment, as shown in Table V. All the RL agents are able to find feasible solutions in all situations. Considering the complexity of the algorithm, DDPG [43], SAC [25], and TD3 [23] generally consume more search time and memory overhead. Across all comparisons, we find Con'X(global) and PPO2 [66] reach better objective value. Con'X(global) converges to the optimized value 4.7 to 24 times faster than alternate RL algorithms.

Analysis of critic networks. In many advanced RL algorithms such as A2C [47], ACTR [86], PPO2 [66], DDPG [43], SAC [25] and TD3 [23], critic networks are used to approximate the underlying value functions, which in turn train the policy network. The REINFORCE-based used in ConfuciuX, on the other hand, only has an actor network that learns directly from the reward. As Table V shows, we found that REINFORCE [73] in Con'X(global) converges to better solutions than all the actor-critic RL algorithms. Our intuition is that this is because the function of the HW performance of the accelerator are too discrete and irregular for a critic neural network to learn well, and this in turn adversely affects the learning of the policy networks. To verify this intuition, we extract the critic network from the implemented alternate RL algorithms [23], [25], [43], [47], [66], [86] and conduct a standalone experiment to test its ability to approximate the underlying value function. The task is to take the "state values" as input and predict the corresponding reward of that state. We use per-layer latency of MobileNet-V2 as reward. We use mean square error and gradient decent to train the network. We show the root mean square error (RMSE) when training with different size of data, as shown in Figure 6. 260,000 is the maximum possible data points critic network can experience under the RL tasks of $Eps = 5,000$ with MobileNet-V2. We can observe that the training and testing loss is hard to converge to a feasible value (the best RMSE is 5.3e+4, which means the predicted latency (reward) by critic network is in average 5.3e+4 cycles difference to the ground-truth ones) which means the critic network did not learn reward value well. This could potentially misguide the policy network.

TABLE VI: Dataflow and Hardware co-automation.

Model	Obj. (min.): Latency Cstr: Area	Optimization Results (cycles)			
		Con'X-dla (global)	Con'X-shi (global)	Con'X-eye (global)	**Con'X-MIX (global)**
MbnetV2	IoT	3.2E+07	3.1E+07	2.9E+07	**2.0E+07**
MbnetV2	IoTx	5.6E+07	4.0E+07	3.7E+07	**3.5E+07**
MnasNet	Cloud	2.1E+07	3.0E+07	2.9E+07	**6.6E+06**
MnasNet	IoT	2.8E+07	3.5E+07	3.4E+07	**1.8E+07**
ResNet50	Cloud	2.2E+08	3.1E+08	2.2E+08	**7.7E+07**
ResNet50	IoT	4.4E+08	4.3E+08	3.1E+08	**3.0E+08**
ResNet50	IoTx	6.3E+08	6.2E+08	4.9E+08	**4.4E+08**
GNMT	Cloud	2.4E+10	1.1E+07	9.7E+07	**6.0E+06**
NCF	Cloud	6.6E+08	2.6E+06	1.1E+06	**6.8E+05**
NCF	IoT	2.6E+06	6.6E+08	1.2E+06	**7.0E+05**

*4) **Sample efficiency and convergence**:* In the experiments against baseline optimization methods and other RL algorithms, we found Con'X(global) has the fastest convergence rate. We show two convergence traces as examples in Figure 7 for MobileNet-V2. With rapid convergence, our method heads toward the objective with more sample efficiency. On the contrary, the exhaustive search needs to enumerate and search through L^{2N}, 12^{104}=$O(10^{112})$, data points for the search space of 52-layer MobileNet-V2 with two actions per layer and 12 level of values per action, which is near impossible to finish.

D. Dataflow-HW co-automation

We extend ConfuciuX to co-automate the per-layer dataflow style decision. Rather than manually picking one of the dataflow style, we let the agent make this decision. To take one step further, we let the agent do fine-grained per-layer dataflow style decision, which we termed as MIX-strategy. The agent now makes three decisions per-layer: PEs, Buffers, and dataflow style. We found Con'X-MIX can not only pick the best dataflow-style for a model but also take advantage of MIX-strategy to pick different dataflow-style in different layers,

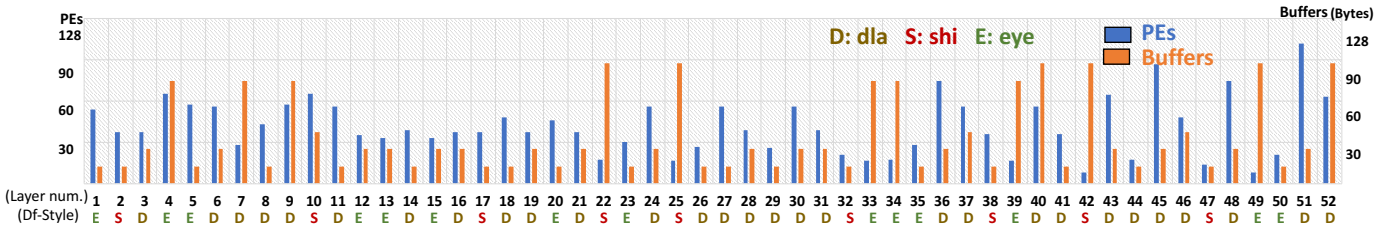


Fig. 8: Dataflow-HW co-automation with ConfuciuX for MobileNet-V2. (Obj.(min.):Latency,Cstr:IoT area).

TABLE VII: Two-stage optimization of ConfuciuX.

Model	Obj. (min.): Latency Cstr: Area	Initial valid value (cy.)	Con'X (global search)		**Con'X (fine-tuning)**	
			Optimized Results (cy.)	Impr. (%)	Optimized Results (cy.)	Impr. (%)
MbnetV2-dla	IoT	7.3E+07	3.2E+07	56.1%	**2.5E+07**	22.7%
MnasNet-dla	IoT	1.0E+08	2.8E+07	71.7%	**2.3E+07**	17.9%
ResNet50-dla	Cloud	1.4E+09	2.2E+08	84.1%	**2.0E+08**	7.0%
ResNet50-dla	IoT	7.1E+08	4.4E+08	37.9%	**3.3E+08**	24.0%
GNMT-dla	IoT	4.6E+09	9.5E+06	99.8%	**6.2E+06**	34.6%
NCF-dla	IoT	6.6E+07	2.6E+06	96.1%	**1.8E+05**	93.1%

as shown in Figure 8 for MobileNet-V2.In general, if there are no HW resource constraints, system will favor eye/shi at early layers (larger activations), which parallelize along activations dimensions, and favor dla at late layers (larger K/C), which parallelizes along channel dimensions (K/C) in CNN-based networks. However, when considering HW constraint, it becomes a compound decision trading-off among PE, Buffers, dataflow-style, and area. From the experiment listed in Table VI, we can observe that in a more relaxed constraint, dla performs better than the other two since most layers in CNN-based networks have large K/C dim. However, in a tighter constraint, the parallelization ability of dla will be restricted; Eye/shi, which parallelize activations dim (whose values shrink layer-by-layer quickly in most CNNs) become more efficient choices. This observation can also explain the fact that system chooses eye/dla for some of the later layers in Figure 8. From the experiments listed in Table VI, Con'X-MIX further improves the optimization results by 4% to 69% comparing to the best-performing Con'X-dla/shi/eye.

E. Benefit of Two-stage Optimization

In the above comparing with baseline experiments, we did not use local fine-tuning for fairness. We now show its effectiveness. We use local GA of 20 populations and run for 2,000 generations. We use local crossover rate of 0.2, local mutation rate of 0.05, and local mutation step of 4 .

*1) **The effect of fine-tuning**:* We show the two stage optimization results in Table VII. We show one of the trace of reached-value along epochs in Figure 9, which is the first row in Table VII and the third row in Table IV. In this case, pure GA cannot find valid solution because of the tight constraint (IoT). The first-stage global search of Con'X learns to generate a valid solution first, whose value is recorded as initial valid value in Table VII. Then, Con'X starts to optimize the value

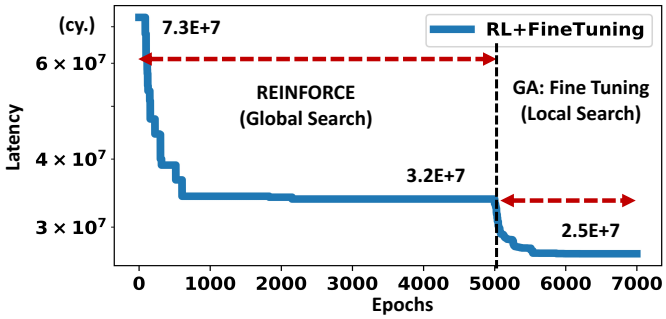


Fig. 9: Overall latency as a function of epochs across two-stage optimization in ConfuciuX (MobileNet-V2, Obj.(min.):Latency, Cstr.:IoT area).

while conforming to constraint and reached an optimized point. The first stage improves the values from 56% to 99% compared to the initial valid values. Then, local fine-tuning using GA to further optimizes the solutions, and they improves by another 7% to 93% than the output of the first stage, which are 66% to 99% improvement over the initial value.

*2) **Analysis of Design-Points found by ConfuciuX**:* In Figure 10, at the top, we show how ConfuciuX allocates area to different components (total PE, total buffers, and per-layer) for MobileNet-V2 and ResNet-50 in an experiment with total area constrained. The per-layer assignment is highly heterogeneous, which can be seen by the per-layer PE and Buffer assignment shown at the bottom of Figure 10. In particular, in MobileNet-V2, we observe that the DWCONV layers are assigned less resources on both PEs and Buffers. This could be because they require less computation and we are limited by the platform area constraint, which makes the agent reduce the assigned resources. In ResNet-50, we find that the agent assigns more Buffers to the layers that have the larger number of input/output channel size (e.g., layers 37,43, and 47).

F. LP deployment at compile time

ConfuciuX can also be used for LP deployment at compile time. One common use-case is for FPGA-based accelerator design. As is common for FPGAs, we impose the maximum number of PEs and Buffers as constraint (which would depend on the specific FPGA board). We consider both cloud and edge FPGAs as constraints. The baseline is configured with uniform number of PE and Buffers for each layer with NVDLA-style dataflow. In Table VIII, we show that Con'X(global)-dla

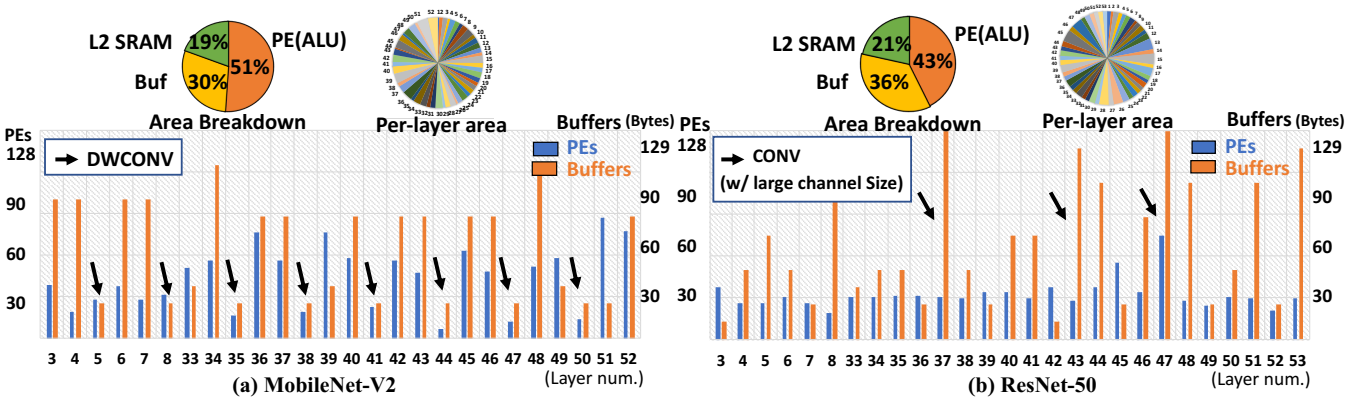


Fig. 10: The solution for (a) MobileNet-V2 and (b) ResNet-50 (Obj.(min.):Latency, Cstr:IoT area).

TABLE VIII: Resource assignments for LP deployment at compile time of ConfuciuX.

Objective: Latency		Baseline-dla			ConfuciuX-dla						ConfuciuX-MIX					
					1st: global search			**2nd: local-finetuning**			1st: global search			**2nd: local-finetuning**		
		Used Cstr.		Optimized	Used Cstr.		Optimized	Used Cstr.		Optimized	Used Cstr.		Optimized	Used Cstr.		Optimized
Platform Constraint	Model	PEs	Bufs	Results (cy.)	PEs	Bufs	Results(cy.)	PEs	Bufs	Results(cy.)	PEs	Bufs	Results	PEs	Bufs	Results(cy.)
Cloud FPGA	ResNet50	4081	7786	2.352E+08	4080	6338	**2.346E+08**	4096	7958	**2.2E+08**	4000	7758	**8.1E+07**	4092	7942	**6.6E+07**
Cstr: PE: 4096, Buf: 8KB	MbnetV2	4056	7716	2.5E+07	4032	3710	**2.3E+07**	4088	7912	**2.0E+07**	4064	3944	**9.4E+06**	4096	7898	**6.0E+06**
Edge FPGA	ResNet50	212	3206	1.8E+09	256	2290	**1.5E+09**	256	3962	**1.2E+09**	256	3942	**1.1E+09**	256	3922	**8.0E+08**
Cstr: PE: 256, Buf: 4KB	MbnetV2	208	3356	1.13E+08	256	2468	**1.08E+08**	256	3838	**6.9E+07**	248	3442	**8.7E+07**	256	3920	**5.7E+07**

TABLE IX: Different configurations of the policy network.

Net Type	Obj. (min.): Latency Cstr: Area	Action Level: 10		**Action Level: 12**		Action Level: 14	
		Optimized Results(cy.)	Used Cstr.	Optimized Results(cy.)	Used Cstr.	Optimized Results(cy.)	Used Cstr.
MLP	Cloud	2.3E+07	63.9%	**2.2E+07**	96.1%	3.0E+07	26.3%
RNN	Cloud	**2.1E+07**	86.7%	**2.1E+07**	95.0%	2.3E+07	68.8%
MLP	IoT	3.4E+07	97.5%	**3.3E+07**	97.6%	4.2E+07	95.2%
RNN	IoT	3.3E+07	99.2%	**3.2E+07**	96.6%	4.2E+07	89.1%
MLP	IoTx	8.3E+07	99.0%	9.4E+07	88.8%	**4.6E+07**	97.9%
RNN	IoTx	7.1E+07	97.4%	**5.6E+07**	90.0%	5.4E+07	99.8%

performs better than baseline-dla. Then we show that local-finetuning in Con'X(global)-dla can further improves the value by 7% to 36%. Finally, we show the two stage results of ConfuciuX-MIX, where the final reached value is 50% to 72% better than baseline-dla.

G. Policy Network Exploration

We show our design decision process of the policy network. First is the action levels, L, where we pick L=12, in the experiments. By decreasing L, we decrease the complexity of the problem but worsen the granularity, and vice versa. As shown in Table IX, L=12 is the sweet spot we found. We also experimented with different type of policy networks: MLP-based and RNN-based, as Table IX shows. We found RNN-based networks converging to better results, which may be owing to the fact that RNN is taking advantage of remembering the consumed constraint of previous layers.

H. Summary

We summarize some key results here. For global search, we observe that RLs can explore an extremely large design space more effectively and efficiently compared to the baseline optimization methods. Next, we find that REINFORCE, which does not rely on value network, can converge faster and reach similar or better results than alternate RL methods in the discrete and irregular HW performance exploration problem. Next, we demonstrate that our formulation of REINFORCE-based (Con'X(global)) can not only explore the HW configuration but also effectively explore the dataflow-style decision simultaneously and further optimize the results by 4% to 69%. Finally, after a coarse-grained solution is found, we show that using a specialized GA for fine-tuning the result locally can optimize the result by another 7% to 93%.

V. Related Works

Accelerator HW Design-Space Exploration. Fine-grained HW resource assignment has been studied extensively for LS deployment on FPGAs [26], [49], [97]. Whole-model LP deployment has been shown to be more efficient than LS deployments with uniform resource assignments for every layer [15], [31], [33], [42], [68], [82], [93], [94]. Many works have focused on allocating resources for convolution layers in a LP deployment within one FPGA [68], [94], across multiple FPGAs [30], [31], [33], [42], [82] or in cloud FPGA platforms [15]. Some works have focused on HW DSE for ASIC accelerators [60], [64], [67] or templated systolic array structures [18], [83]. Some general frameworks execute the design space exploration at the architecture level, supporting both ASIC and FPGA [38], [89]. Yang et. al, [89] further shows that the HW resource assignment dominates the performance of accelerator comparing to dataflow exploration. For design space exploration, most of these prior works employ grid/exhaustive

search, while techniques for pruning the exploration spaces are manually developed. However, with myriads of DNN models being designed on a daily basis, it becomes harder to manually design and tune the policy for the newly constructed search space. In this work, we develop a ML-based method to automate the search process with high sample efficiency for both LP and LS scenario.

ML-based methods for DNN compilation and mapping. ML methods have found value in mapping/compiling DNNs over hardware. TensorComprehensions [79] uses genetic algorithm, AutoTVM [13], [14] uses simulated annealing and boosted tree, Reagen et. al, [59] uses Bayesian optimization, RELEASE [7] uses RL, ATLAS [84] uses black box optimizations, some compiler design [12], [50] use profile-guided optimization to perform target-independent front-end compiler optimizations on DNNs or linear algebra computations. Some recent works use RL on HW/SW co-exploration to explore both DNN and its mapping over hardware [6], [32], [44], [88]. The problem of mapping the DNN computation graph over multiple devices (CPU/GPU/TPU [34]) has also been explored through manual heuristics [8], [72], [91] and RL [24], [46], [51]. In contrast to these works, this work looks at fine-grained design-time assignment of compute and memory within an accelerator.

Dataflow style optimization. Architecture design of ML accelerators include resource assignment and dataflow style design. Dataflow style is a scheduling and compiler optimization problem, which has been studied for decades for the generic platform such as CPU or GPU [5], [19], [36], [37], [40], [58], [71], [81], or for FPGA [45], [92], while they apply grid search for reaching their objectives. For ML accelerators, some mainstream dataflow style are manually designed and proven to be efficient, becoming prominent or commercialized [3], [16], [20], [34]. In this work, we focus on the resource assignment part of the accelerator design flow, and utilize some prominent dataflow styles [3], [16], [20], [34].

VI. Conclusion

While efficient DNN models and dataflow style are widely studied for ML accelerators, HW resource assignment is relatively unexplored. In this paper, we propose ConfuciuX, an autonomous strategy to find out the optimized HW resource assignment for a given DNNs, a dataflow style and platform constraints. ConfuciuX leverages RL for the global search, augmented with GA for fine-tuning. We quantitatively experiment on different models, platform constraints and dataflow styles. ConfuciuX demonstrates the highest sample-efficiency compared to other optimization and RL methods. This works shows the promise of leveraging ML within the DNN accelerator design workflow, with opportunities for future work across new ML algorithms for learning dataflow/hardware behavior, and DNN-dataflow-hardware co-design.

Acknowledgement

This work was supported by NSF Award 1909900 and a Google Faculty Award. We thank Hyoukjun Kwon for help with MAESTRO setup and feedback on the writing. We acknowledge Arun Ramamurthy, Siva Theja Maguluri and Ananda Samajdar for helpful technical discussions. A special thanks to Cliff Young for motivating us to pursue this research direction.

References

[1] "Stars and bars (combinatorics)," https://en.wikipedia.org/wiki/Stars_and_bars_(combinatorics).
[2] "Openai gym," https://gym.openai.com/, 2016.
[3] "Nvdla deep learning accelerator," http://nvdla.org, 2017.
[4] "Maestro tool," http://maestro.ece.gatech.edu/, 2020.
[5] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.
[6] M. S. Abdelfattah, Ł. Dudziak, T. Chau, R. Lee, H. Kim, and N. D. Lane, "Best of both worlds: Automl codesign of a cnn and its hardware accelerator," *arXiv preprint arXiv:2002.05022*, 2020.
[7] B. H. Ahn, P. Pilligundla, and H. Esmaeilzadeh, "Reinforcement learning and adaptive sampling for optimized dnn compilation," *arXiv preprint arXiv:1905.12799*, 2019.
[8] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.
[9] S. Bang, J. Wang, Z. Li, C. Gao, Y. Kim, Q. Dong, Y.-P. Chen, L. Fick, X. Sun, R. Dreslinski *et al.*, "14.7 a 288μw programmable deep-learning processor with 270kb on-chip weight storage using non-uniform memory hierarchy for mobile intelligence," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 2017, pp. 250–251.
[10] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of machine learning research*, vol. 13, no. Feb, pp. 281–305, 2012.
[11] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *Proceedings of the 37th annual international symposium on Computer architecture*, 2010, pp. 247–257.
[12] P. P. Chang, S. A. Mahlke, and W.-M. W. Hwu, "Using profile information to assist classic code optimizations," *Software: Practice and Experience*, vol. 21, no. 12, pp. 1301–1321, 1991.
[13] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "{TVM}: An automated end-to-end optimizing compiler for deep learning," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 578–594.
[14] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to optimize tensor programs," in *Advances in Neural Information Processing Systems*, 2018, pp. 3389–3400.
[15] Y. Chen, J. He, X. Zhang, C. Hao, and D. Chen, "Cloud-dnn: An open framework for mapping dnn models to cloud fpgas," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 73–82.
[16] Y.-H. Chen *et al.*, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *JSSC*, vol. 52, no. 1, pp. 127–138, 2016.
[17] Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," in *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*, 2016, pp. 262–263.
[18] J. Cong and J. Wang, "Polysa: polyhedral-based systolic array auto-compilation," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.
[19] S. Dave, Y. Kim, S. Avancha, K. Lee, and A. Shrivastava, "Dmazerunner: Executing perfectly nested loops on dataflow accelerators," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–27, 2019.
[20] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *International Symposium on Computer Architecture (ISCA)*, 2015.

[21] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello, "Hardware accelerated convolutional neural networks for synthetic vision systems," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*. IEEE, 2010, pp. 257–260.

[22] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "Cnp: An fpga-based processor for convolutional networks," in *2009 International Conference on Field Programmable Logic and Applications*. IEEE, 2009, pp. 32–37.

[23] S. Fujimoto, H. Van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," *arXiv preprint arXiv:1802.09477*, 2018.

[24] Y. Gao, L. Chen, and B. Li, "Spotlight: Optimizing device placement for training deep neural networks," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. Stockholmsmässan, Stockholm Sweden: PMLR, 10–15 Jul 2018, pp. 1676–1684. [Online]. Available: http://proceedings.mlr.press/v80/gao18a.html

[25] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," *arXiv preprint arXiv:1801.01290*, 2018.

[26] C. Hao, X. Zhang, Y. Li, S. Huang, J. Xiong, K. Rupnow, W.-m. Hwu, and D. Chen, "Fpga/dnn co-design: An efficient design methodology for 1ot intelligence on the edge," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.

[27] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[28] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T.-S. Chua, "Neural collaborative filtering," in *Proceedings of the 26th international conference on world wide web*, 2017, pp. 173–182.

[29] J. H. Holland, "Genetic algorithms," *Scientific american*, vol. 267, no. 1, pp. 66–73, 1992.

[30] W. Jiang, E. H.-M. Sha, X. Zhang, L. Yang, Q. Zhuge, Y. Shi, and J. Hu, "Achieving super-linear speedup across multi-fpga for real-time dnn inference," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–23, 2019.

[31] W. Jiang, E. H.-M. Sha, Q. Zhuge, L. Yang, X. Chen, and J. Hu, "Heterogeneous fpga-based cost-optimal design for timing-constrained cnns," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2542–2554, 2018.

[32] W. Jiang, L. Yang, E. Sha, Q. Zhuge, S. Gu, Y. Shi, and J. Hu, "Hardware/software co-exploration of neural architectures," *arXiv preprint arXiv:1907.04650*, 2019.

[33] W. Jiang, X. Zhang, E. H.-M. Sha, Q. Zhuge, L. Yang, Y. Shi, and J. Hu, "Xfer: A novel design to achieve super-linear performance on multiple fpgas for real-time ai," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 305–305.

[34] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 1–12.

[35] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.

[36] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–29, 2017.

[37] A. Klöckner, "Loo. py: transformation-based code generation for gpus and cpus," in *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, 2014, pp. 82–87.

[38] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna, "Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 754–768.

[39] H. Kwon, A. Samajdar, and T. Krishna, "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 461–475, 2018.

[40] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "Deepx: A software accelerator for low-power deep learning inference on mobile devices," in *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. IEEE, 2016, pp. 1–12.

[41] H. Larochelle, D. Erhan, A. Courville, J. Bergstra, and Y. Bengio, "An empirical evaluation of deep architectures on problems with many factors of variation," in *Proceedings of the 24th international conference on Machine learning*, 2007, pp. 473–480.

[42] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou, and L. Wang, "A high performance fpga-based accelerator for large-scale convolutional neural networks," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2016, pp. 1–9.

[43] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[44] Q. Lu, W. Jiang, X. Xu, Y. Shi, and J. Hu, "On neural architecture search for resource-constrained hardware platforms," *arXiv preprint arXiv:1911.00105*, 2019.

[45] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 45–54.

[46] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device placement optimization with reinforcement learning," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017, pp. 2430–2439.

[47] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*, 2016, pp. 1928–1937.

[48] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[49] M. Motamedi, P. Gysel, V. Akella, and S. Ghiasi, "Design space exploration of fpga-based deep convolutional neural networks," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2016, pp. 575–580.

[50] D. Novillo, "Samplepgo-the power of profile guided optimizations without the usability burden," in *2014 LLVM Compiler Infrastructure in HPC*. IEEE, 2014, pp. 22–28.

[51] A. Paliwal, F. Gimeno, V. G. Nair, Y. Li, M. Lubin, P. Kohli, and O. Vinyals, "Reinforced genetic algorithm learning for optimizing computation graphs," 2020.

[52] A. Parashar, P. Raina, Y. S. Shao, Y.-H. 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 *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2019, pp. 304–315.

[53] M. Parsa, A. Ankit, A. Ziabari, and K. Roy, "Pabo: Pseudo agent-based multi-objective bayesian hyperparameter optimization for efficient neural accelerator design," *arXiv preprint arXiv:1906.08167*, 2019.

[54] M. Pelikan, D. E. Goldberg, E. Cantú-Paz *et al.*, "Boa: The bayesian optimization algorithm," in *Proceedings of the genetic and evolutionary computation conference GECCO-99*, vol. 1, 1999, pp. 525–532.

[55] N. Pinto, D. Doukhan, J. J. DiCarlo, and D. D. Cox, "A high-throughput screening approach to discovering good forms of biologically inspired visual representation," *PLoS computational biology*, vol. 5, no. 11, 2009.

[56] I. Popov, N. Heess, T. Lillicrap, R. Hafner, G. Barth-Maron, M. Vecerik, T. Lampe, Y. Tassa, T. Erez, and M. Riedmiller, "Data-efficient deep reinforcement learning for dexterous manipulation," *arXiv preprint arXiv:1704.03073*, 2017.

[57] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI Blog*, vol. 1, no. 8, p. 9, 2019.

[58] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *Acm Sigplan Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[59] B. Reagen, J. M. Hernández-Lobato, R. Adolf, M. Gelbart, P. Whatmough, G.-Y. Wei, and D. Brooks, "A case for efficient accelerator design space exploration via bayesian optimization," in *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 2017, pp. 1–6.

[60] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 267–278.

[61] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution strategies as a scalable alternative to reinforcement learning," *arXiv preprint arXiv:1703.03864*, 2017.

[62] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.

[63] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. P. Graf, "A massively parallel coprocessor for convolutional neural networks," in *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*. IEEE, 2009, pp. 53–60.

[64] G. Santoro, M. R. Casu, V. Peluso, A. Calimera, and M. Alioto, "Energy-performance design exploration of a low-power microprogrammed deep-learning accelerator," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 1151–1154.

[65] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic superoptimization," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 305–316, 2013.

[66] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[67] Y. S. Shao, S. L. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks, "Co-designing accelerators and soc interfaces using gem5-aladdin," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[68] Y. Shen, M. Ferdman, and P. Milder, "Maximizing cnn accelerator efficiency through resource partitioning," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 535–547.

[69] D. Stamoulis, E. Cai, D.-C. Juan, and D. Marculescu, "Hyperpower: Power-and memory-constrained hyper-parameter optimization for neural networks," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 19–24.

[70] D. Stamoulis, T.-W. Chin, A. K. Prakash, H. Fang, S. Sajja, M. Bognar, and D. Marculescu, "Designing adaptive neural networks for energy-constrained image classification," in *Proceedings of the International Conference on Computer-Aided Design*, 2018, pp. 1–8.

[71] M. Steuwer, T. Remmelg, and C. Dubach, "Lift: a functional data-parallel ir for high-performance gpu code generation," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2017, pp. 74–85.

[72] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, 2014, pp. 3104–3112.

[73] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in neural information processing systems*, 2000, pp. 1057–1063.

[74] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Thirty-first AAAI conference on artificial intelligence*, 2017.

[75] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "Mnasnet: Platform-aware neural architecture search for mobile," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 2820–2828.

[76] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "Mnasnet: Platform-aware neural architecture search for mobile," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 2820–2828.

[77] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," *arXiv preprint arXiv:1905.11946*, 2019.

[78] R. R. Torrado, P. Bontrager, J. Togelius, J. Liu, and D. Perez-Liebana, "Deep reinforcement learning for general video game ai," in *2018 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2018, pp. 1–8.

[79] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," *arXiv preprint arXiv:1802.04730*, 2018.

[80] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[81] R. Wei, L. Schwartz, and V. Adve, "Dlvm: A modern compiler infrastructure for deep learning systems," *arXiv preprint arXiv:1711.03016*, 2017.

[82] X. Wei, Y. Liang, X. Li, C. H. Yu, P. Zhang, and J. Cong, "Tgpa: tile-grained pipeline architecture for low latency cnn inference," in *Proceedings of the International Conference on Computer-Aided Design*, 2018, pp. 1–8.

[83] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput cnn inference on fpgas," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[84] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE, 1998, pp. 38–38.

[85] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[86] Y. Wu, E. Mansimov, R. B. Grosse, S. Liao, and J. Ba, "Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation," in *Advances in neural information processing systems*, 2017, pp. 5279–5288.

[87] Y. Wu and Y. Tian, "Training agent for first-person shooter game with actor-critic curriculum learning," 2016.

[88] L. Yang, Z. Yan, M. Li, H. Kwon, L. Lai, T. Krishna, V. Chandra, W. Jiang, and Y. Shi, "Co-exploration of neural architectures and heterogeneous asic accelerator designs targeting multiple tasks," *arXiv preprint arXiv:2002.04116*, 2020.

[89] X. Yang, M. Gao, J. Pu, A. Nayak, Q. Liu, S. E. Bell, J. O. Setter, K. Cao, H. Ha, C. Kozyrakis *et al.*, "Dnn dataflow choice is overrated," *arXiv preprint arXiv:1809.04070*, 2018.

[90] S. Yin, P. Ouyang, S. Zheng, D. Song, X. Li, L. Liu, and S. Wei, "A 141 uw, 2.46 pj/neuron binarized convolutional neural network based self-learning speech recognition processor in 28nm cmos," in *2018 IEEE Symposium on VLSI Circuits*. IEEE, 2018, pp. 139–140.

[91] W. Yonghui, M. Schuster, Z. Chen, Q. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey *et al.*, "Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[92] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015, pp. 161–170.

[93] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, and J. Cong, "Energy-efficient cnn implementation on a deeply pipelined fpga cluster," in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, 2016, pp. 326–331.

[94] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "Dnnbuilder: an automated tool for building high-performance dnn hardware accelerators for fpgas," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[95] Z. Zhao, H. Kwon, S. Kuhar, W. Sheng, Z. Mao, and T. Krishna, "mrna: Enabling efficient mapping space exploration for a reconfiguration neural accelerator," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2019, pp. 282–292.

[96] S. Zheng, P. Ouyang, D. Song, X. Li, L. Liu, S. Wei, and S. Yin, "An ultra-low power binarized convolutional neural network-based speech recognition processor with on-chip self-learning," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, no. 12, pp. 4648–4661, 2019.

[97] G. Zhong, A. Prakash, S. Wang, Y. Liang, T. Mitra, and S. Niar, "Design space exploration of fpga-based accelerators with multi-level parallelism," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 1141–1146.

[98] S. Zhong, Y. Shen, and F. Hao, "Tuning compiler optimization options via simulated annealing," in *2009 Second International Conference on Future Information Technology and Management Engineering*. IEEE, 2009, pp. 305–308.