CONCODE: HARD-CONSTRAINED DIFFERENTIABLE CO-EXPLORATION METHOD FOR NEURAL ARCHITECTURES AND HARDWARE ACCELERATORS

Anonymous authors
Paper under double-blind review

ABSTRACT

While DNNs achieve over-human performances in a number of areas, it is often accompanied by the skyrocketing computational costs. Co-exploration of an optimal neural architecture and its hardware accelerator is an approach of rising interest which addresses the computational cost problem, especially in low-profile systems (e.g., embedded, mobile). The difficulty of having to search the large co-exploration space is often addressed by adopting the idea of differentiable neural architecture search. Despite the superior search efficiency of the differentiable co-exploration, it faces a critical challenge of not being able to systematically satisfy hard constraints, such as frame rate or power budget. To handle the hard constraint problem of differentiable co-exploration, we propose ConCoDE, which searches for hard-constrained solutions without compromising the global design objectives. By manipulating the gradients in the interest of the given hard constraint, high-quality solutions satisfying the constraint can be obtained. Experimental results show that ConCoDE is able to meet the constraints even in tight conditions. We also show that the solutions searched by ConCoDE exhibit high quality compared to those searched without any constraint.

1 INTRODUCTION

The primary interest of most *Deep Neural Network* (DNN) researches has always been the application performance (i.e., accuracy). However, it also led to the rapid growth in the network size that require immense computational resources for execution. In recent years, many works have appeared to mitigate the resource problem, mostly belonging to one of these two categories – *network-side optimization* and *hardware-side optimization*. Network-side optimization refers to refining the architecture of a neural network to reduce computations while maintaining a comparable accuracy (Howard et al., 2017; Sandler et al., 2018; Iandola et al., 2017; Zhou et al., 2016; Frankle & Carbin, 2019; Gale et al., 2019; He et al., 2018; Zhu & Gupta, 2018; Renda et al., 2020). Hardware-side optimization, on the other hand, often involves improving DNN execution efficiency by using optimized hardware design, also known as hardware *accelerator*s (Chen et al., 2016; Jouppi et al., 2017; Chen et al., 2014; Jang et al., 2021; Lym et al., 2019). Unfortunately, effort from one side often hinders the benefits coming from the other. For example, the main advantage of depth-wise separable convolution operation often used in MobileNet family (Howard et al., 2017; Sandler et al., 2018) comes from its structure which uses a single channel for its operation. However, Google's TPU (Jouppi et al., 2017), a renowned accelerator, mainly utilizes channel-level parallelism for gaining speedup. In consequence, MobileNet results in a poor execution time on TPUs (Gupta & Akin, 2020).

Co-exploration of hardware accelerator and network architecture (Li et al., 2020; Fu et al., 2021; Choi et al., 2021; Abdelfattah et al., 2020; Hao et al., 2019; Lu et al., 2019; Yang et al., 2020) is therefore a natural direction to fulfill both goals of accuracy and hardware metrics such as latency, energy consumption and silicon chip area. To address the large search space of the co-exploration, differentiable *Neural Architecture Search* (NAS) based methods (Choi et al., 2021; Li et al., 2020; Fu et al., 2021) are considered as promising approaches due to their ability to quickly explore the search space compared to its reinforcement learning based counterparts (Abdelfattah et al., 2020; Hao et al., 2019; Lu et al., 2019; Yang et al., 2020).

Unfortunately, differentiable co-exploration has a serious drawback of being unable to deal with hard constraints that are critical in many real-world scenarios. For instance, one of the important constraints of object detection system (Redmon et al., 2016) is to meet the frame rate of the camera (e.g., 30 frames per second). In addition, a mobile subsystem running on a limited battery often has a power budget given by the system design decision (Jang et al., 2021). Because differentiable co-exploration methods rely on a single loss function, they often fail to satisfy the constraints, and have to blindly undergo a several repetitions of hyper-parameter tuning and re-exploration.

In order to address the problem, we present *ConCoDE* (**Con**strained **Co**design with **D**ifferentiable **E**xploration), which enables hard-constrained differentiable co-exploration of neural architecture and hardware accelerator. The key concept of our proposal is a gradient manipulation method that ensures the solution does not drift away from meeting the constraints. In addition to the gradients from the global loss function, we calculate the gradient of the hardware constraints, which is used to manipulate the gradient of the global loss, if any constraint violations, such that i) the dot product of the two are positive (i.e., they point to a similar direction), and ii) the direction can alleviate the violation of the constraints (although it little sacrifices other objectives).

To the best of our knowledge, this is the first work that considers hard constraints in a differentiable co-exploration problem. We conduct an extensive amount of evaluation to demonstrate that ConCoDE can 1) satisfy the hardware constraints even under tight constraints and 2) the searched solution does not compromise the quality (i.e., global loss function).

Our contributions can be summarized as follows.

- We propose a hard-constrained differentiable co-exploration method for network and accelerator in order to find valid solutions without trial-and-errors.
- We propose using gradient manipulation to gradually move solutions towards the constraint-satisfying region.
- We provide an extensive evaluation for ConCoDE to show the constraint-meeting capability and efficiency of its search method.

2 Related Work

2.1 Neural Architecture Search

Neural Architecture Search (NAS) refers to the technique of automating the neural network design process. Starting from some of the early works (Zoph et al., 2018; Real et al., 2019), many have grown to outperform human-designed architectures. Most common are the reward based methods such as Reinforcement-Learning (RL). However, such RL-based methods (Zoph & Le, 2017; Baker et al., 2017) require extensive search costs for evaluating every candidate network. Differentiable NAS (Liu et al., 2019) has been proposed as an efficient alternative, where we can take advantage of gradient flow in its update to reduce the huge time cost to a few orders of magnitude shorter time.

Regardless, optimizing solely on network performance is insufficient as they do not take hardware efficiency into account. Some recent works (Cai et al., 2019b; Wu et al., 2019a) that address this issue consider hardware costs on top of differentiable NAS by adding related loss terms. Some also attempt to reduce the network size for latency constraints using simple latency models (Berman et al., 2020; Nayman et al., 2021), but they cannot be used for co-exploration since the relation between accuracy and latency is not reflected in the model, in addition to the lack of hardware accelerator consideration.

2.2 RL-based Co-exploration

The early works on the co-exploration utilize RL-based method to leverage its simplicity. Each candidate network is trained for evaluation, while the accelerator design is analyzed for hardware efficiency. These values create rewards used by the agent to create the next candidate solution.

Hao et al. (2019) conducts FPGA/DNN co-exploration to achieve a high accuracy-low latency design using coordinate descent. Lu et al. (2019) designed a framework that can jointly explore architecture, quantization, and hardware search space. Following a previous work (Jiang et al., 2019) that considers FPGA implementation performance, Jiang et al. (2020) further expands it into co-exploration by

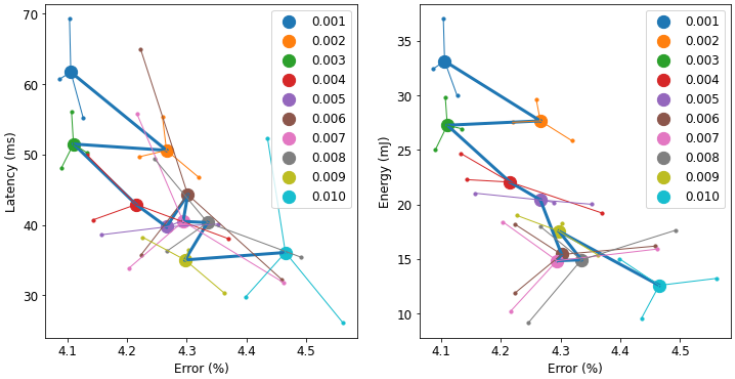


Figure 1: A motivational experiment. In each plot, we swept the hyperparameter λ_2 on the hardware cost from 0.001 to 0.010. The results from three searches done in identical settings are depicted with same colors with the average in the center as a larger dot. It is clear that the trajectory is not strictly linear to λ_2. Moreover, the variation within the same setting often overlaps that of the other settings, demonstrating the difficulty of satisfying the hard constraint of the co-exploration problem.

enabling hardware design changes. Some recent works (Abdelfattah et al., 2020; Yang et al., 2020) also take RL-based approach to navigate their co-exploration space.

These methods all inherit the same problem from RL-based NAS methods in which they require expensive training to evaluate each candidate solution. To worsen the matter, co-exploration requires even larger network/hardware search space than searching only for networks.

2.3 Differentiable Co-exploration

Li et al. (2020) was the first to express the network/hardware co-exploration problem as a differentiable mathematical formulation. However, their search space for the accelerator is severely limited to a single parameter that controls calculation parallelism. Auto-NBA (Fu et al., 2021) used a differentiable accelerator search engine to build a joint-search pipeline, and DANCE (Choi et al., 2021) trained auxiliary neural networks for hardware search and cost evaluation, which allowed gradient descent to directly consider the relation between hardware cost and the network parameters. However, none of the above properly addresses the hard constraint problem. In this work, we propose a holistic method of handling hard constraints on differentiable co-exploration.

3 Difficulty of Hard-constrained Differentiable Co-exploration

The most straightforward and naïve way to handle hard constraints within differentiable co-exploration would be to change the relative weight to the hardware cost. For example, below is a loss function from Cai et al. (2019b).

$$\mathcal{L}oss = \mathcal{L}oss_{CE} + \lambda_1 ||w||_2^2 + \lambda_2 Latency. \tag{1}$$

By increasing λ_2, we can instruct the search process to consider hardware metrics (e.g. latency, energy) more and yield a result that is more optimized towards them. However, giving a larger penalty only induces a soft guidance than a strict instruction, meaning it does not directly lead to reduction in the value of constrained metric. Figure 1 plots how changing λ_2 from 0.001 to 0.010 affects the latency/energy and the classification error for CIFAR-10 dataset. Searches were done three times for each setting and plotted with same colors. Even though some trend is observed as λ_2 increases, inconsistency in both direction and variance of the trajectory is more dominant. Such unpredictability makes it difficult to determine the appropriate magnitude of the hyperparameter that is needed to meet the constraint. Thus, trying to match the given constraint can become an arduous search that takes exhaustive amount of trials.

Some recent works tackle the hard constraint problem by introducing soft-constraints terms into the loss function. For example, Cai et al. (2019b) adds $(t/T)^\lambda$ where t is the measured latency for a searched network, T is the target latency and λ is a hyperparameter for controlling the accuracy and latency trade-off. Similarly, Hu et al. (2020) uses $\lambda \cdot max(t/T - 1, 0)$ instead, giving more explicit

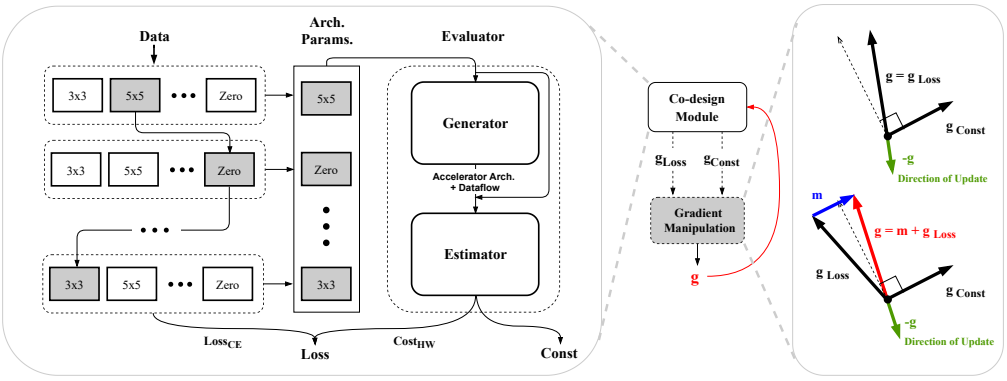


Figure 2: Our proposed method for hard constrained co-design. Left is a close-up of our co-design module, and right is the depiction of gradient manipulation. Gradient of the loss from the co-design module is manipulated using the gradient of the constraint to obtain a new gradient. This is used to update the module in the direction of lowering the value of the constrained metric.

consideration of the target latency. Compared to the simple scaling, manipulating the slope conditioned on the constraint offers more control. However, such methods does not guarantee satisfying the hard constraint, and require multiple trial-and-errors until a valid solution is found.

Despite the difficulties that lie in tackling a hard-constrained co-design problem, designing an effective strategy to it is indeed necessary. We propose a gradient manipulation based methods that can successfully achieve this goal.

4 Hard-Constrained Co-exploration

4.1 Problem Definition

The mathematical formulation of hard-constrained differentiable co-exploration is as below:

$$\begin{aligned} &\arg\min_{\alpha,\beta}(\mathcal{L}oss_{NAS}(w^*, net(\alpha)) + \lambda_{Cost} \cdot Cost_{HW}(eval(\alpha,\beta))), \\ &\text{s.t. } w^* = \arg\min_{w}(\mathcal{L}oss_{NAS}(w, net(\alpha))), \end{aligned} \tag{2}$$

while satisfying the hard constraint $t \leq T$. α and β denote network architecture parameters and hardware accelerator configuration, respectively. w is the weights of the supernet and $net(\alpha)$ is the dominant network architecture selected. $eval(\alpha, \beta)$ indicates the hardware metrics evaluated for α and β. Usually it includes the execution latency, energy consumption, and the chip area. The objective of co-exploration is expressed using two distinct evaluation metrics, which are neural architecture loss ($\mathcal{L}oss_{NAS}$. e.g., accuracy) and hardware cost ($Cost_{HW}$) defined from the user. Our goal here is to find the α and β that minimize the combination of $\mathcal{L}oss_{NAS}$ and $Cost_{HW}$. Both metrics are measured with optimal values of w that minimize each metric, respectively.

4.2 Co-exploration without Hard Constraint

Conceptually similar to many appraised works, our approach towards searching in a co-exploration space involves formulating the problem in a differentiable way. Figure 2 illustrates the overall architecture of the proposed method. In the leftmost part is the network search module. Like most differentiable NAS, this module searches for network architecture by choosing a path from the supernet. The network structure is then fed to the evaluator module.

The evaluator $eval()$ is implemented as a composition of a hardware generator $gen()$ and an estimator $est()$, where the hardware generator is jointly trained with the supernet training and the estimator is pre-trained. With these, we convert the objective in Eq. 2 as below:

$$\begin{aligned} &\arg\min_{\alpha}(\mathcal{L}oss_{NAS}(w^*, net(\alpha)) + \lambda_{Cost} \cdot Cost_{HW}(est(\alpha, gen(v^*, \alpha)))), \\ &\text{s.t. } w^* = \arg\min_{w}(\mathcal{L}oss_{NAS}(w, net(\alpha))), \\ &\quad v^* = \arg\min_{v}(Cost_{HW}(est(\alpha, gen(v, \alpha)))), \end{aligned} \tag{3}$$

where v is the weights for the hardware generator.

The hardware generator $gen()$ takes the architecture parameters and uses them to output the optimal hardware implementation (β from Eq. 2). We used a five-layer Multi-Layer Perceptron (MLP) with residual connections and ReLU as activation functions in between the layers. It is trained simultaneously with the network search module each time to effectively adapt to the given constraint, which varies under different scenarios. While the hardware cost function can take many forms, we use the following simple linear combination of the three hardware metrics:

$$Cost_{HW} = \lambda_E Energy + \lambda_L Latency + \lambda_A Area. \tag{4}$$

The estimator $est()$ shares a similar architecture to the generator, and is pre-trained using the output from other cost estimation frameworks such as MAESTRO (Kwon et al., 2020), Timeloop (Parashar et al., 2019), and Accelergy (Wu et al., 2019b). After pre-training, the entire evaluator is frozen during the exploration and is only used to infer the hardware cost given a network architecture.

We use ProxylessNAS (Cai et al., 2019b) as our NAS backbone with path sampling to train α, but our method is orthogonal to the NAS implementation and has the flexibility to choose from any differentiable NAS algorithms. Using ProxylessNAS with an additional weight decay term, the neural architecture loss becomes:

$$\mathcal{L}oss_{NAS} = \mathcal{L}oss_{CE} + \lambda_{Decay}||w||. \tag{5}$$

4.3 Hard-Constrained Co-exploration with Gradient Manipulation

In addition to the differentiable co-exploration methodology, we suggest the novel idea of gradient manipulation as an effective solution to the hard constraint problem. Direct manipulation of gradients is a strategy often used in achieving multiple goals, such as in continual learning (Saha et al., 2021; Lopez-Paz & Ranzato, 2017), differential equations (Kim et al., 2021), or market prediction (Li et al., 2021). In this paper, we present a solution to apply gradient manipulation to the co-exploration problem in the interest of satisfying hard constraints.

Figure 2 shows a high-level abstraction of our gradient manipulation method. The main idea is to artificially generate a force that can push the gradient in the direction that 'agree's with the constraint. The conditions under which the method is applied to compute the new gradient g are defined as below:

$$g = \begin{cases} g_{\mathcal{L}oss} & \text{, if } t \leq T \\ & \text{or } t > T \wedge g_{\mathcal{L}oss} \cdot g_{\mathcal{C}onst} \geq 0, \\ m_\alpha + g_{\mathcal{L}oss} & \text{, otherwise} \end{cases} \tag{6}$$

$$g_{\mathcal{C}onst} = \frac{\partial \max(t-T, 0)}{\partial \alpha}. \tag{7}$$

In the above equation, $g_{\mathcal{L}oss}$ is the original gradient from the global loss function defined as

$$\mathcal{L}oss = \mathcal{L}oss_{NAS} + \lambda_{Cost} \cdot Cost_{HW}, \tag{8}$$

as in Eq. 3, and $g_{\mathcal{C}onst}$ is the gradient of constraint loss that we define as: $\mathcal{C}onst = max(t-T, 0)$. t denotes the current value of constrained metric outputted from $est()$ such as latency or energy, and T is the target value (e.g., 33.3 ms for latency). Note that t is a function of α, and thus can be backpropagated to find the gradient with respect to α. There are two distinct outcomes that can be obtained from the following cases. In an ideal case where the $t \leq T$, the constraint is already met so we do nothing to the gradient. In the unfortunate case when the constraint is not met, we calculate for the dot product of the two gradients to determine the agreement in their direction. If $g_{\mathcal{L}oss} \cdot g_{\mathcal{C}onst} \geq 0$ (i.e., the angle between two gradients is less than 90°), it means gradient descent update will contribute towards satisfying the constraint. Thus it is interpreted as an agreement in direction and the same $g_{\mathcal{L}oss}$ is used unmodified. Top right of the Figure 2 depicts this scenario. However, if they disagree as illustrated in bottom right of Figure 2 (i.e., $g_{\mathcal{L}oss} \cdot g_{\mathcal{C}onst} < 0$), we force the gradient to shift its direction by m_α, which is obtained from $(m_\alpha + g_{\mathcal{L}oss}) \cdot g_{\mathcal{C}onst} \geq 0$ to guarantee decrease in target cost after gradient descent. It can be reformulated as $m_\alpha \cdot g_{\mathcal{C}onst} + g_{\mathcal{L}oss} \cdot g_{\mathcal{C}onst} = \delta$ where $\delta \geq 0$ is a small value for ensuring gradual movement towards satisfying the constraint.

For updating α and w, we solve for optimal m_α with respect to α, which are the parameters for the network architecture. To minimize the effect of m_α on $g_{\mathcal{L}oss}$, we use a pseudoinverse-based

solution (Ben-Israel & Greville, 2003) that is known to minimize the size of $||m_\alpha||_2^2$. Using the pseudoinverse of $g_{\mathcal{C}onst}$,

$$m_\alpha^* = \frac{-(g_{\mathcal{L}oss} \cdot g_{\mathcal{C}onst}) + \delta}{||g_{\mathcal{C}onst}||_2^2} g_{\mathcal{C}onst}. \tag{9}$$

In order to control the magnitude of the pull, we use a small multiplying factor $p > 0$ on δ. The policy for updating δ using p is as follows: Some initial value δ_0 exists for δ. If the target metric fails to meet the constraint, δ is multiplied by $1 + p$ to strengthen the pull ($\delta^{'} = (1 + p)\delta$). In the other case when the constraint is satisfied, δ is reset to its initial value ($\delta^{'} = \delta_0$).

Note that we also train our v, weights for the differentiable hardware generator using gradient descent. Thus we compute for m_v^* in the same manner, but instead use $g_{\mathcal{C}ost_{HW}}$ in place of $g_{\mathcal{L}oss}$ for updating the generator.

Although a single constraint is already a challenging target, our method can be further generalized to accommodate multiple constraints. With the same idea of using inner products to determine the consensus in directionality, now the gradient is modified only in the direction of individual constraints that do not comply. We provide a more generalized formulation:

$$g = \begin{cases} g_{\mathcal{L}oss} & \text{, if } \bigwedge_{i=1}^{n}(t_i \leq T_i) \\ & \text{ or } \bigvee_{i=1}^{n}(t_i > T_i) \wedge g_{\mathcal{L}oss} \cdot g_{\mathcal{C}onst} \geq 0, \\ m_\alpha + g_{\mathcal{L}oss} & \text{, otherwise} \end{cases} \tag{10}$$

$$g_{\mathcal{C}onst} = \frac{\partial \sum_{i=1}^{n} \max(t_i - T_i, 0)}{\partial \alpha}, \tag{11}$$

where $Const$ is now defined as a sum of n constrained hardware metrics, T as their target values, and t denotes current values for each metric. m_α is computed the same way using Eq. 9.

5 Experiments

5.1 Experimental Environment

We have conducted the following experiments on ConCoDE using CIFAR10 (Krizhevsky et al., 2009) and ImageNet ILSVRC2012 (Krizhevsky et al., 2012) dataset. All experiments are conducted on PyTorch 1.9.0, CUDA 11.1 with an RTX3090 GPU.

As a neural architecture search backbone, we use ProxylessNAS (Cai et al., 2019b). The operation search space per layer follows MobilenetV2 architecture (Sandler et al., 2018), which consists of multiple settings of MBConv operation with kernel size $\{3, 5, 7\}$ and expand ratio $\{3, 6\}$. The total number of layers is 18 and 21 for CIFAR and ImageNet, respectively. Other settings for network search phase are identical to the original paper of the backbone.

The backbone hardware accelerator is based on Eyeriss (Chen et al., 2016). The accelerator is composed of a two-dimensional Processing Element (PE) array for parallel calculation. Each PEs has a Multiply-Accumulate (MAC) unit attached to a register file. In addition, *dataflow*, the computation order of DNN processing forms an important aspect that determines the efficiency of the data reuse. Therefore, hardware accelerator design space comprises PE array size from 12×8 to 20×24, register file size per PE from 16B to 256B, and dataflow in Weight-Stationary (WS) similar to (Jouppi et al., 2017), Output-Staionary (OS) similar to (Du et al., 2015) and Row-Stationary (RS) similar to (Chen et al., 2016).

To train the estimator, we first build a dataset by randomly sampling 10.8M network-accelerator pairs ($2.95\text{e}{-9}$ % of the total search space) from our search space which are evaluated on hardware metrics using Timeloop (Parashar et al., 2019) and Accelergy (Wu et al., 2019b). Using this dataset, the estimator network is trained for 200 epochs with the batch size of 256. The weight update is done using Adam optimizer with the learning rate of 1e-4. For all the hardware metrics reported, we have used the direct evaluation on the designed hardware from Timeloop and Accelergy instead of the values outputted by the estimator to avoid any possible error in the learned model.

Evaluation of network performance is done by training the final network architecture, which we train from scratch for 300 epochs using the batch size of 64. We use SGD optimizer with Nesterov

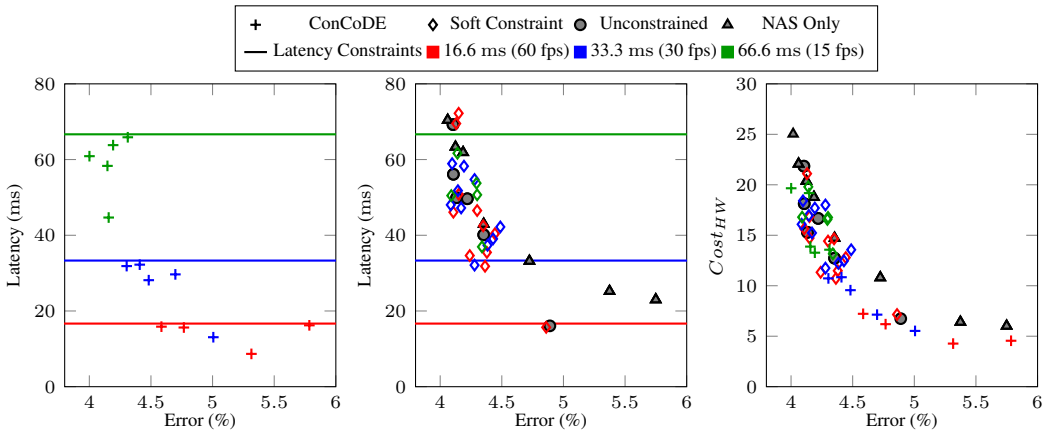


Figure 3: Co-exploration results. (left) and (mid) represent the Latency and (right) represent the hardware cost. Colored marks are hard-constrained and should result in a lower value than the horizontal line of the same color.

momentum (Nesterov, 1983), and cosine learning rate scheduling with 0.008 as its initial value, while weight decay term and momentum is 1e-3 and 0.9 respectively. Augmentation for the train data is adopted from AutoAugment (Cubuk et al., 2019), on both CIFAR and ImageNet dataset.

5.2 Experimental Results

Figure 3 plots the co-exploration experimental results from multiple techniques, on Proxyless-NAS (Cai et al., 2019b) backbone and CIFAR-10 dataset. In the experiments, we have set three different constraints for the latency: 16.6ms, 33.3ms, and 66.6ms, which correspond to 60 frames per second (fps), 30 fps, and 15 fps, respectively. For comparison, we implemented the following:

- **NAS only**: A plain neural architecture search only, where a best-effort hardware accelerator was designed *after* the NAS has been complete. Multiple solutions were found by setting various λ_2 (the flops penalty term) from Eq. 1.
- **Unconstrained**: Co-exploration based on ConCoDE without hard constraints, similar to the differentiable co-exploration methods by Choi et al. (2021) or Fu et al. (2021). As in NAS only method, multiple solutions were found by scaling λ_{Cost} of Eq. 8 without changing the cost function itself.
- **Soft Constraint**: To represent the conventional method of using soft constraints, we have added the term $\lambda_{Soft} \cdot max(t/T - 1, 0)$ to $Cost_{HW}$ as in (Hu et al., 2020).
- **ConCoDE**: The proposed method with $p = 1\text{e}{-2}$.

In all experiments, we have used $\lambda_E = 2.9$, $\lambda_L = 6.2$, and $\lambda_A = 1.0$ from Eq. 4 to make a fair comparison. For the NAS only method, hardware cost function was used just for the hardware design phase. For evaluation on a different cost function, please refer to the appendix.

Figure 3 (left) and (mid) shows the relation between error and latency. The colored horizontal bars represent the three latency targets we have applied. It can be easily seen that all solutions found by ConCoDE satisfy the given hard constraints. For a tight constraint of 16.6 ms, most solutions have latency values just below 16.6 ms. On the other hand, solutions for 66.6 ms, a relatively loose constraint, yields some solutions far below the constraint, because those solutions exhibit better global loss (Eq. 8). Soft-constraint based methods, however, often fail to meet the constraints, as depicted with the diamond markers placed over the same-colored horizontal lines. As tighter the constraint gets, the more they fail to meet the constraints. Solutions from 'Unconstrained' co-exploration and 'NAS only' demonstrate that getting a trade-off between latency (constrained metric) and accuracy is possible, but there is no control for enforcing a solution with certain target value.

Table 1: Experimental Results Showing the Quality of Solutions

Index	Constrained	Latency (ms)	Energy (mJ)	Chip Area (mm^2)	Error (%)	$Cost_{HW}$	Global Loss
A	Unconst. Original	69.23	37.00	2.53	4.10 ± 0.16	21.84	0.632
	Latency	**43.99**	21.79	2.10	4.20 ± 0.07	13.87	0.624
	Energy	51.98	**29.18**	2.53	4.38 ± 0.17	17.44	0.630
	Chip Area	64.00	34.82	**2.53**	4.05 ± 0.06	20.56	0.629
	All	**63.72**	**12.09**	**1.86**	4.12 ± 0.18	13.29	0.623
B	Unconst. Original	49.65	27.53	2.53	4.22 ± 0.06	16.67	0.638
	Latency	**48.02**	27.33	2.53	4.27 ± 0.09	16.41	0.644
	Energy	95.02	**24.45**	1.89	4.05 ± 0.10	20.76	0.648
	Chip Area	54.74	29.81	**2.53**	4.11 ± 0.13	17.96	0.645
	All	**41.32**	**8.59**	**1.86**	4.35 ± 0.05	9.50	0.629
C	Unconst. Original	56.11	29.81	2.53	4.11 ± 0.10	18.13	0.662
	Latency	**51.81**	9.49	1.89	4.38 ± 0.11	11.06	0.645
	Energy	44.78	**24.38**	2.53	4.12 ± 0.02	15.15	0.656
	Chip Area	53.37	26.63	**2.10**	4.43 ± 0.07	16.44	0.668
	All	**41.53**	**8.81**	**1.86**	4.48 ± 0.20	9.59	0.645

*Bold colored numbers indicate that they are under constraint of the same colored non-bold numbers.

5.3 Solution Quality Found by ConCoDE

Figure 3 (right) plots $Cost_{HW}$ and error together, which allows evaluating quality of the solutions. Because Figure 3 (left) and (mid) overlook the other hardware metrics (energy and chip area), comparing the $Cost_{HW}$ and error together is required for evaluation of the solution quality. Even though the global loss function differs in individual solutions, plotting $Cost_{HW}$ and error in two independent axes allows a fair comparison in terms of Pareto-optimality.

From the plot, it is clear that the co-exploration methods (ConCoDE, Soft Constrained, and Unconstrained) yield solutions of better quality than the 'NAS only'. On the other hand, quality of solutions from ConCoDE shows no particular degradation from the others.

To further study the quality of the solutions found by ConCoDE, we have conducted another set of experiments. We selected a few solutions found from 'Unconstrained' method as 'anchor' solutions and listed them in Table 1. From those, we chose either one or all three of the hardware metrics to be fixed as the hard constraint, and performed co-explorations using ConCoDE. Because it is guaranteed that such solution exists, a good method should be able to find a solution meeting the constraint, of at least a similar quality. The results are shown in Table 1. As in the Section 5.2, all of the 12 cases we have examined succeeded in finding a valid solution. Sometimes, the resulting hardware metrics were much small than the targets, but in those cases, the error was larger than that of the anchors, and show similar global loss values.

5.4 Results from ImageNet Dataset

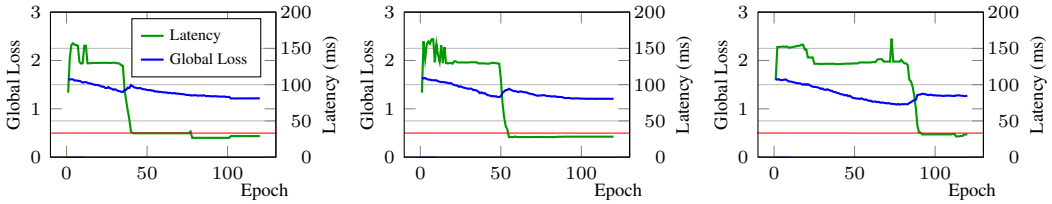
Table 2 shows the exploration results from ImageNet (ILSVRC2012) dataset (Krizhevsky et al., 2012). We have performed an unconstrained co-exploration. Then, similar to Table 1, hard-constrained co-exploration was performed using the values from unconstrained co-exploration as anchors. As the results show, ConCoDE satisfies the latency/energy constraints with competitive global loss values.

5.5 Sensitivity Study on Pulling Control

In ConCoDE, p is the only tuning knob that controls the pulling magnitude. Figure 4 illustrates how the global loss and latency changes over latency-constrained (33.3 ms) explorations, with varying p of 1e-2, 7e-3, and 4e-3.

Table 2: Experimental Results for ImageNet

Constrained	Latency (ms)	Energy (mJ)	Chip Area (mm^2)	Error (%)	$Cost_{HW}$	Global Loss
Unconstrained	165.98	47.35	2.56	25.46	29.37	2.043
Latency	**84.47**	36.20	2.40	26.22	20.64	2.053
Energy	78.75	**32.23**	2.40	27.94	18.68	2.130



(a) p=1e-2 (b) p=7e-3 (c) p=4e-3

Figure 4: Sensitivity to p on ConCoDE. The red lines represents latency constraint at 33.3 ms.

Regardless of the value of p, the curve for the constrained value shows a similar trend. At the beginning, the global loss becomes mainly optimized, while the latency stays steady for certain number of epochs. During this phase the pulling magnitude δ (See Eq. 9) is still growing, and is not strong enough to make meaningful changes. At certain point (around epoch 40 in p=1e-2), δ becomes strong enough to pull the solution towards lowering latency. In this phase, the global loss slightly increases due to a stronger movement towards latency optimization. When the latency satisfies the constraint ($< 33.3\,\mathrm{ms}$), global loss starts to decrease while maintaining the latency at the same level. The latency occasionally increases above the target, but it is quickly pulled down within a few steps. Altering p changes when the latency drop starts while the shape of the curves are similar to others. One interesting aspect is that regardless of the value, there is no significant discrepancy between the final solution in terms of the global loss and the latency, which shows that ConCoDE is relatively insensitive to the hyperparameter p.

6 Discussion

One-shot NAS methods. Recently, one-shot NAS methods (Cai et al., 2019a; Guo et al., 2019; Nayman et al., 2021) are drawing much attention, which trains the supernet only once and searches for a sub-network for a given purpose. However, one-shot NAS method is not a directly satisfying solution for the constrained co-search task. First, hardware metrics of each network need to be measured on the whole hardware accelerator search space, then compared with each other, which is a non-trivial process that takes a long time. Furthermore, cost for training the entire supernet itself is often multiple times more expensive than that of a conventional differentiable NAS, and searching for each subnetwork is also costly, making the break-even number of deployments very high.

Nonetheless, if multiple deployments are planned over diverse constraints, we believe ConCoDE can utilize one-shot NAS as its backbone thanks to its differentiability. One way to apply one-shot NAS method onto ConCoDE is to attach differentiable architecture selector on pre-trained supernet. Since both components are differentiable neural networks, gradient manipulation technique can be applied to them in a coherent manner. Expanding our experiment further to include one-shot NAS as our network search backbone is left as future work.

Expanding Design Space. One might wonder whether the scheme used in ConCoDE can be applied to other co-exploration methods. For example, Li et al. (2020) and Fu et al. (2021) optimize the quantization level of each layer in addition to the neural architecture. We believe ConCoDE can accommodate such additional design space into consideration. Provided that there exists a analytic or simulation model for those additional spaces (which is not difficult to build), satisfying hard-constraint through gradient manipulation is likely to work with little extra effort.

7 Conclusion

In this paper, we proposed ConCoDE, a hard-constrained differentiable co-exploration method for neural network and hardware accelerator. By conditionally applying gradient manipulation that moves the solution towards meeting the constraints, hard constraints can be reliably satisfied with high-quality solutions. We believe this proposal would ease the development of DNN based systems by a significant amount.

REFERENCES

Mohamed S Abdelfattah, Lukasz Dudziak, Thomas CP Chau, Royson Lee, Hyeji Kim, and Nicholas D Lane. Best of Both Worlds: AutoML Codesign of a CNN and its Hardware Accelerator. In *DAC*, 2020.

Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing Neural Network Architectures using Reinforcement Learning. In *ICLR*, 2017.

Adi Ben-Israel and Thomas NE Greville. *Generalized Inverses: Theory and Applications*, volume 15. Springer Science & Business Media, 2003.

Maxim Berman, Leonid Pishchulin, Ning Xu, Matthew B Blaschko, and Gérard Medioni. AOWS: Adaptive and Optimal Network Width Search with Latency Constraints. In *CVPR*, 2020.

Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-All: Train One Network and Specialize it for Efficient Deployment. In *ICLR*, 2019a.

Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware. In *ICLR*, 2019b.

Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning. In *ASPLOS*, 2014.

Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE JSSC*, 2016.

Kanghyun Choi, Deokki Hong, Hojae Yoon, Joonsang Yu, Youngsok Kim, and Jinho Lee. DANCE: Differentiable Accelerator/Network Co-Exploration. In *DAC*, 2021.

Ekin Dogus Cubuk, Barret Zoph, Dandelion Mané, Vijay Vasudevan, and Quoc V. Le. AutoAugment: Learning Augmentation Strategies From Data. In *CVPR*, 2019.

Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. Shidiannao: Shifting vision processing closer to the sensor. In *ISCA*, 2015.

Jonathan Frankle and Michael Carbin. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. In *ICLR*, 2019.

Yonggan Fu, Yongan Zhang, Yang Zhang, David Cox, and Yingyan Lin. Auto-NBA: Efficient and Effective Search Over the Joint Space of Networks, Bitwidths, and Accelerators. In *ICML*, 2021.

Trevor Gale, Erich Elsen, and Sara Hooker. The State of Sparsity in Deep Neural Networks. In *ICML*, 2019.

Zichao Guo, Xiangyu Zhang, Haoyuan Mu, Wen Heng, Zechun Liu, Yichen Wei, and Jian Sun. Single Path One-Shot Neural Architecture Search with Uniform Sampling. *arXiv preprint arXiv:1904.00420*, 2019.

Suyog Gupta and Berkin Akin. Accelerator-aware Neural Network Design using AutoML. *arXiv preprint arXiv:2003.02838*, 2020.

Cong Hao, Xiaofan Zhang, Yuhong Li, Sitao Huang, Jinjun Xiong, Kyle Rupnow, Wen-mei Hwu, and Deming Chen. FPGA/DNN Co-Design: An Efficient Design Methodology for IoT Intelligence on the Edge. In *DAC*, 2019.

Yang He, Guoliang Kang, Xuanyi Dong, Yanwei Fu, and Yi Yang. Soft Filter Pruning for Accelerating Deep Convolutional Neural Networks. In *IJCAI*, 2018.

Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. In *CVPR*, 2017.

Yibo Hu, Xiang Wu, and Ran He. TF-NAS: Rethinking Three Search Freedoms of Latency-Constrained Differentiable Neural Architecture Search. In *ECCV*, 2020.

Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level Accuracy with 50x Fewer Parameters and¡ 0.5 MB Model Size. In *ICLR*, 2017.

Jun-Woo Jang, Sehwan Lee, Dongyoung Kim, Hyunsun Park, Ali Shafiee Ardestani, Yeongjae Choi, Channoh Kim, Yoojin Kim, Hyeongseok Yu, Hamzah Abdel-Aziz, et al. Sparsity-Aware and Re-configurable NPU Architecture for Samsung Flagship Mobile SoC. In *ISCA*, 2021.

Weiwen Jiang, Xinyi Zhang, Edwin H-M Sha, Lei Yang, Qingfeng Zhuge, Yiyu Shi, and Jingtong Hu. Accuracy vs. Efficiency: Achieving Both Through Fpga-Implementation Aware Neural Architecture Search. In *DAC*, 2019.

Weiwen Jiang, Lei Yang, Edwin Hsing-Mean Sha, Qingfeng Zhuge, Shouzhen Gu, Sakyasingha Dasgupta, Yiyu Shi, and Jingtong Hu. Hardware/Software Co-Exploration of Neural Architectures. *IEEE TCAD*, 2020.

Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *ISCA*, 2017.

Jungeun Kim, Kookjin Lee, Dongeun Lee, Sheo Yon Jhin, and Noseong Park. DPM: A Novel Training Method for Physics-Informed Neural Networks in Extrapolation. In *AAAI*, 2021.

Alex Krizhevsky, Geoffrey Hinton, et al. Learning Multiple Layers of Features from Tiny Images, 2009. URL `http://www.cs.utoronto.ca/~kriz/learning-features-2009-TR.pdf`.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet Classification with Deep Convolutional Neural Networks. In *NeurIPS*, 2012.

Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. MAESTRO: A data-centric approach to understand reuse, performance, and hardware cost of DNN mappings. *IEEE Micro*, 2020.

Duanshun Li, Jing Liu, Jinsung Jeon, Seoyoung Hong, Thai Le, Dongwon Lee, and Noseong Park. Large-Scale Data-Driven Airline Market Influence Maximization. In *KDD*, 2021.

Yuhong Li, Cong Hao, Xiaofan Zhang, Xinheng Liu, Yao Chen, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. EDD: Efficient Differentiable DNN Architecture and Implementation Co-Search for Embedded AI Solutions. In *DAC*, 2020.

Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable Architecture Search. In *ICLR*, 2019.

David Lopez-Paz and Marc'Aurelio Ranzato. Gradient Episodic Memory for Continual Learning. In *NeurIPS*, 2017.

Qing Lu, Weiwen Jiang, Xiaowei Xu, Yiyu Shi, and Jingtong Hu. On Neural Architecture Search for Resource-Constrained Hardware Platforms. In *ICCAD*, 2019.

Sangkug Lym, Armand Behroozi, Wei Wen, Ge Li, Yongkee Kwon, and Mattan Erez. Mini-Batch Serialization: CNN Training with Inter-Layer Data Reuse. In *MLSys*, 2019.

Niv Nayman, Yonathan Aflalo, Asaf Noy, and Lihi Zelnik-Manor. HardCoRe-NAS: Hard Constrained diffeRentiable Neural Architecture Search. *arXiv preprint arXiv:2102.11646*, 2021.

Y. E. Nesterov. A Method for Solving the Convex Programming Problem with Convergence Rate $O(1/k^2)$. *Dokl. Akad. Nauk SSSR*, 1983.

Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel Emer. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In *ISPASS*, 2019.

Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized Evolution for Image Classifier Architecture Search. In *AAAI*, 2019.

Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You Only Look Once: Unified, Real-time Object Detection. In *CVPR*, 2016.

Alex Renda, Jonathan Frankle, and Michael Carbin. Comparing Fine-tuning and Rewinding in Neural Network Pruning. In *ICLR*, 2020.

Gobinda Saha, Isha Garg, and Kaushik Roy. Gradient Projection Memory for Continual Learning. In *ICLR*, 2021.

Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetv2: Inverted Residuals and Linear Bottlenecks. In *CVPR*, 2018.

Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search. In *CVPR*, 2019a.

Yannan Nellie Wu, Joel S. Emer, and Vivienne Sze. Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs. In *ICCAD*, 2019b.

Lei Yang, Zheyu Yan, Meng Li, Hyoukjun Kwon, Liangzhen Lai, Tushar Krishna, Vikas Chandra, Weiwen Jiang, and Yiyu Shi. Co-Exploration of Neural Architectures and Heterogeneous ASIC Accelerator Designs Targeting Multiple Tasks. In *DAC*, 2020.

Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. *arXiv preprint arXiv:1606.06160*, 2016.

Michael Zhu and Suyog Gupta. To Prune, or Not to Prune: Exploring the Efficacy of Pruning for Model Compression. In *ICLR Workshops*, 2018.

Barret Zoph and Quoc V. Le. Neural Architecture Search with Reinforcement Learning. In *ICLR*, 2017.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning Transferable Architectures for Scalable Image Recognition. In *CVPR*, 2018.

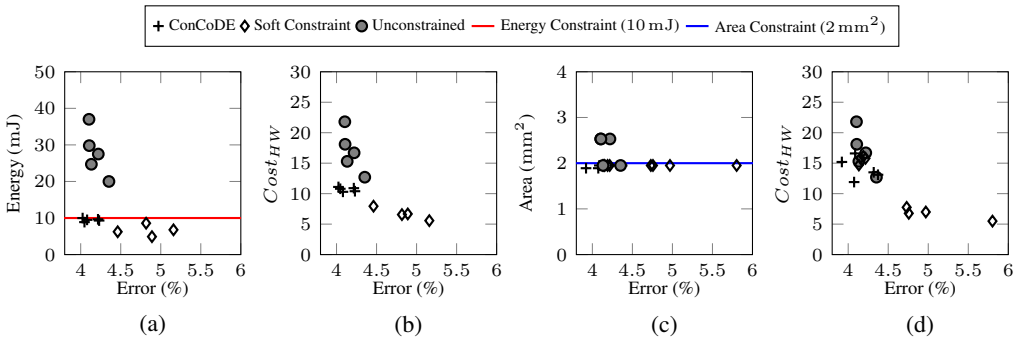


Figure 5: Experimental results for energy- and area- constrained co-exploration. Results in (a) and (b) are the energy-constrained and those in (c) and (d) are area-constrained exploration.

A Energy- and Area-Constrained Solutions

Figure 5 plots the co-exploration results similar to Figure 3 when energy and area are constrained. We have set $10\,\mathrm{mJ}$ as the energy target, and $2\,\mathrm{mm}^2$ as the area target. As shown in the figure, all solutions found by ConCoDE meet the constraints, and the resulting constrained values are closer to the target than that of the soft-constrained search. Area constraints are relatively easier to satisfy, because the chip area depends solely on the accelerator design, and is independent of the neural network architecture. Therefore, both ConCoDE and the soft constrained method were able to satisfy the constraint with similar quality.

B Results on an Alternative $Cost_{HW}$

Figure 6 presents the co-exploration performed on a different hardware cost function. We have used $\lambda_E = 9.5$, $\lambda_L = 4.4$, and $\lambda_A = 1.0$, which is a more energy-oriented setting than that from the experiments in the main body. Despite the difference in the $Cost_{HW}$, the results of co-exploration shows the same observations. ConCoDE achieves solutions within constraint, and soft-constrained exploration misses the target especially when the constraint is tight. Also, the overall quality of the solution found by ConCoDE is of at least similar quality compared to the other methods.

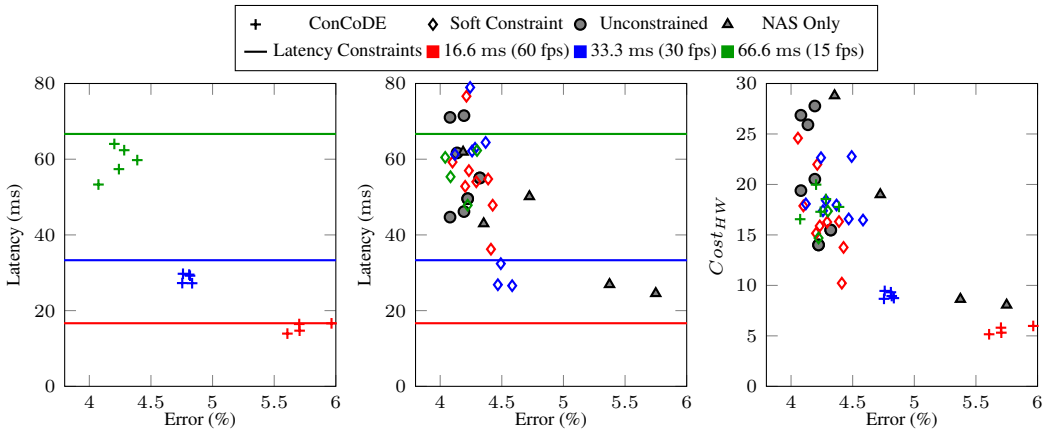


Figure 6: Co-exploration results on an alternative $Cost_{HW}$.

Table 3: Experimental Results on CIFAR-100 Dataset

Constrained	Latency (ms)	Energy (mJ)	Chip Area (mm^2)	Error (%)	$Cost_{HW}$	Global Loss
16.6 ms (60 fps)	16.34	8.82	2.23	22.90 ± 0.51	6.81	1.588
	7.23	3.68	1.95	26.21 ± 0.35	3.92	1.718
33.3 ms (30 fps)	33.24	19.38	2.53	20.89 ± 0.42	12.27	1.539
	29.94	15.58	2.10	21.37 ± 0.22	10.32	1.568
66.6 ms (15 fps)	57.76	31.94	2.53	20.33 ± 0.21	18.95	1.518
	43.45	22.40	2.10	20.73 ± 0.11	13.98	1.560

C Results on CIFAR-100 Dataset

Table 3 presents the co-exploration performed on CIFAR-100 dataset. For the three latency target values (16.6 ms, 33.3 ms, and 66.6 ms) ConCoDE found a number of solutions. For each target value, we chose one at the lowest error, and another at the lowest $Cost_{HW}$. They all satisfy the constraints, with different trade-offs. Interestingly, the pair of accelerator designs found from 33.3 ms and 66.6 ms constraints were identical. This is because when the dataset and the backbone network architecture is decided, the number of channels as well as the spatial dimension of the activation map are fixed. These values govern most of the parallelism that can be exploited by the accelerator and thus the solutions ended up in a convergent evolution.

D Background on Hardware Accelerators for DNNs

To overcome the ever-increasing computational intensity of the DNN execution, much efforts to accelerate the processing by specialized hardware have been made from both industry and academia.

The DNN accelerators usually focus on the parallel computation of MAC (Multiply-Accumulate) operation and local data reuse. For the parallel computation, DNN accelerators often utilize a two-dimensional mesh array of PEs (Processing Elements) that is connected to a global buffer that temporarily stores data for rapid accesses. Each PE computes an element-wise multiplication of a input activation and a weight parameter in a collaborative way for neural network operations.

Another important aspect of accelerator efficiency is data reuse, which attempts to reduce the main memory access by utilizing limited amount of on-chip buffers and register files. This essentially converts to the problem of organizing calculation order of loops in DNN processing. The DNN layer usually consists of multiple computational loops. For example, a CNN layer calculation comprises seven for-loop dimensions, consisted of (H,W,C) for input activation, (R,S,K) for kernel weight, and (N) for batch size.

The scheme that maps and orders these loops is commonly called *dataflow* and this is closely related to architecture design philosophy. Many pieces of research about dataflow have been made to achieve a different aspect of hardware efficiency. Weight-Stationary (WS, Jouppi et al. (2017)), Output-Stationary (OS, Du et al. (2015)), and Row-Stationary (RS, Chen et al. (2016)) dataflow policies are renowned examples. Each has its own advantages, such as energy efficiency (RS), execution time (WS), or on-chip memory requirement (OS) as well as the parallelism they utilize.

Table 4: Details of Selected Solutions

Constraint	Evaluated Metrics			Network Stats		Hardware Design			
	Error (%)	Latency (ms)	$Cost_{HW}$	#Layers	#Parameters	PE_X	PE_Y	RF	Dataflow
16.6 ms (60 fps)	4.58 ± 0.11	15.86	7.22	13	0.88 M	16	16	4	WS
	5.31 ± 0.05	8.66	4.27	7	0.36 M	16	8	4	WS
33.3 ms (30 fps)	4.30 ± 0.17	31.86	10.73	16	1.0 M	20	8	4	WS
	4.70 ± 0.11	29.67	7.14	10	0.62 M	12	8	64	RS
66.6 ms (15 fps)	4.00 ± 0.11	60.90	19.66	18	1.5 M	16	16	4	WS
	4.19 ± 0.12	63.81	13.27	17	1.1 M	12	8	64	RS

In the co-exploration problem, the task of hardware accelerator design is to decide those factors, such that it fits well with the neural architecture being co-designed. Among many design factors, we chose number of PEs along the two dimensions, register file size, and dataflow as our search space.

E DETAILS OF SELECTED SOLUTIONS

In this section, we present some details of a few selected solutions from that of Figure 3 in Table 4. As in Table 3, for each of the three constraint, we chose one solution that shows the best accuracy (the lowest error), and another that shows the best $Cost_{HW}$. One noticeable trend is that for tighter latency constraints, the networks get smaller with less parameters and layers. This shows that the network capacity had to be traded off with latency to meet the constraint. Also, there were more solutions with row-stationary (RS) for the best $Cost_{HW}$ solution, which aligns with the previous findings (Chen et al., 2016).