MULTI-OBJECTIVE OPTIMIZATION FOR HARDWARE-AWARE NEURAL ARCHITECTURE SEARCH

Anonymous authors
Paper under double-blind review

ABSTRACT

Hardware-aware Neural Architecture Search (HW-NAS) has been drawing increasing attention since it can automatically design deep neural networks optimized in a resource-constrained device. However, existing methods may be not optimal in terms of multi-object (accuracy, hardware-metrics). Thus, we propose a new multi-objective optimization method for searching promising architectures in HW-NAS. Our method addresses the architecture selection process in NAS. An architecture population is divided into small cells by given hardware cost metrics, then, top-ranked architecture is selected within each cell. The selected ones guide the direction of evolution in NAS. Despite its simplicity, this method leads to promising results, improving both accuracy and hardware metrics. Using latency as a hardware metric, we demonstrated that the optimized architectures were found that run faster and achieve similar accuracy. We can also significantly reduce cost of a search using both accuracy predictor and latency estimator and sharing pre-trained weights of super-network. Overall searching procedure takes under 1 minute on a single CPU. For a target hardware, we experimented on both CPU and Field Programmable Gate Array (FPGA). The codes are available at https://anonymous.4open.science/r/multi-objective-optimization-0E27/README.md.

1 INTRODUCTION

Designing neural architecture for a Deep Neural Network (DNN) is very important because the architecture mostly determines the performance of a DNN. However designing neural architecture is not trivial but complex and time-consuming since the architectures could be complicated, consisting of many layers which could be interconnected each other through residual connection (He et al., 2015), branch (Szegedy et al., 2014), and squeeze-and-excitation (Howard et al., 2019), etc. Designing of neural architecture requires repetitive experimentation and domain knowledge, thus it mainly relied on human expertise.

Recently Neural Architecture Search (NAS), automatically designing neural network architectures has been growing in popularity. NAS already demonstrated that the generated architectures are competitive or even higher accuracy (Zoph & Le, 2017; Zoph et al., 2018; Benmeziane et al., 2021) compared to the traditional hand-designed ones such as AlexNet (Krizhevsky et al., 2012), VGGNet (Simonyan & Zisserman, 2015), GoogleNet (Szegedy et al., 2014), ResNet (He et al., 2015), etc.

Despite the significant success achieved to date, applying NAS to real-world applications still poses substantial challenges and is not widely used. One of the reasons is that the generated neural network architectures are too complex or big to be deployed in resource-constrained hardware platforms, such as IoT, mobile device, and embedded systems.

To address the resource-constraint problems, Hardware-aware Neural Architecture Search (HW-NAS) (Tan et al., 2019; Wu et al., 2019) has emerged as one of the most promising techniques. HW-NAS can automate the designing process of deep neural architectures for a specific target hardware. Most popular hardware cost metrics are inference latency time, power consumption, and memory size. To adopt hardware awareness in NAS, a simple solution would be to include an additional proxy metric in existing NAS frameworks such as the number of parameters or the number of floating-point operations (FLOPs) that describe the latency of a neural architecture on the hardware.

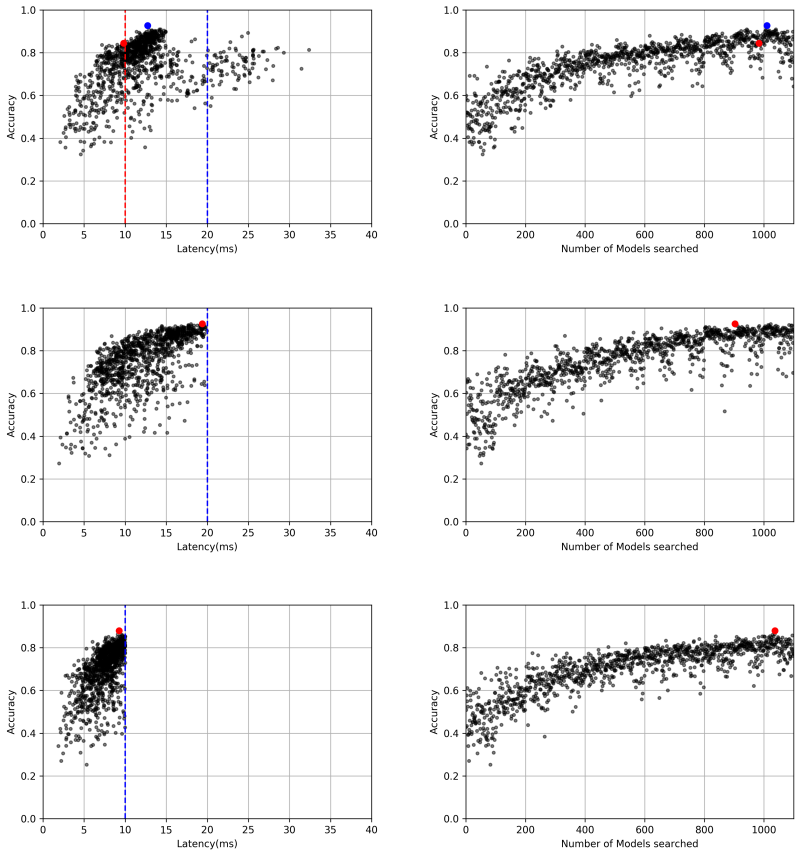

Figure 1: Mono-objective optimized search with constraint. **Left:** Accuracy versus latency **Right:** search progress of the experiment. Red dot and Blue dot are the best accuracy model under the constraint. **Top:** search candidate models with accuracy by repeating for many iterations. Then, to screen the models with latency constraints (e.g. 20ms, 10ms). **Middle:** search candidate models under latency constraint 20ms. **Bottom:** search candidate models under latency constraint 10ms.

Applying NAS algorithms based on the proxy metrics, however, has uncertainty since the proxy metrics do not always correlate well with measured hardware-metrics (Li et al., 2021). For instance, inference latency depends on the characteristics of a specific hardware platform. To address this issue, many hardware aware search algorithms directly retrieve latency from a given hardware platform (Tan et al., 2019; Wu et al., 2019; Zhang et al., 2020; Dong et al., 2021), and search for optimal architecture that can meet certain application or hardware constraints.

To satisfy such hardware constraints, for example, latency as the most dominant one, the following two methods are typically implemented. The first method is to search candidate architectures with accuracy by repeating for many iterations. Then, to screen only architectures satisfying the constraint of latency and to find the architecture with the highest accuracy within the constraint. The other method is to apply the constraint of latency during the search process and to search candidate architectures with accuracy under the constraint. Figure 1 illustrated the searching results for these two methods.

The above two methods are based on mono-objective optimization and do not guarantee an optimal solution for considering both accuracy and latency. Meanwhile, multi-objective optimization can be applied, with the weighted sum of accuracy and latency with a fixed ratio (Elsken et al., 2019; Benmeziane et al., 2021) as illustrated in Figure 12 or a weighted product method (Tan et al., 2019; Abdelfattah et al., 2020) as illustrated in Figure 13 and Figure 14. However, these approaches do not guarantee either that the final output is optimized in terms of multi-object . Here we propose

a simple and efficient multi-objective optimized method and redefine the terminology of optimality from the most accurate model for a predetermined latency threshold to the most accurate models for continuous latency thresholds instead of discrete latency thresholds.

We make the following contributions in this paper:

- We improve the efficiency of HW-NAS as multi-objective optimization. For the optimized method, we propose a new approach based on epsilon-constraint method (Yang et al., 2014), which divides an architecture population into many small cells based on hardware-metrics, then for each cell, select top-ranked architectures. For instance, in the case of latency as a hardware metric, the architectures achieving top accuracy are selected from each cell and guide the direction of evolution in NAS.
- We significantly reduce the computational cost of the search phase in NAS. General NAS requires huge computing costs for training and evaluating accuracy of sub-networks. We built an accuracy predictor to predict the accuracy of a sub-network ,which correlation coefficient of validation sample is 0.9635. This predictor-based approach significantly reduces computing cost than general NAS and makes NAS more accessible.
- We developed a latency estimator, which provides accurate hardware latency value and can be easily implemented and reproducible. Setting up measurement pipeline on a specific hardware platform is not trivial and requires domain knowledge in a hardware inference. Alternative way is to use pre-collected hardware cost look-up (Cai et al., 2020; Li et al., 2021). However, we noticed that there is a gap between estimated value from a hardware-latency look-up table and measured latency value. This gap could misguide HW-NAS to suboptimal points. A latency estimator can eliminate complex data collection pipelines for hardware devices. We built DNN-based latency estimators for both CPU and Field Programmable Gate Array (FPGA) using thousands of (architecture, latency value) pairs. The correlation coefficient of latency estimators is above 0.98 for validation samples. Using both of accuracy predictor and latency estimator, we can reduce search process under 1 minute on a single CPU.

2 RELATED WORKS

A general NAS process consists of three main components: search space, search strategy, and evaluation methodology. A search space is a set of neural network architectures. It determines how neural network operators can be connected to form a valid network and which operators are allowed. A search space is explored by a search strategy, which samples a population of neural architectures' candidates. Then, the accuracy of the sampled architecture is evaluated using an evaluation methodology. Based on the measured accuracy, the search strategy guides more promising architectures for a next generation in the search space.

Search space There are two kinds of search spaces, which are layer-wise search space and block-based search space (Benmeziane et al., 2021). Block-based approach stacks a set of blocks to form larger and deeper architectures, which is inspired from many state-of-arts hand-designed architectures such as ResNet and Mobilenet (Howard et al., 2017). Meanwhile, Layer-wise search determines the number of layers and dimensions of each layer. A whole architecture is generated from a pool of operators, where the first and the last layers are typically fixed by its operators, the remaining layers need to be optimized. Example operators are convolution, pooling, and activation etc. Differently connecting these operators gives rise to a different architecture. A key aspect of designing a well-performing deep neural network is deciding the type and number of neurons and how to compose and connect them. Additionally, the architectural hyper-parameters such as stride and the number of channels in a convolution are also needed to be optimized. We construct a layer-wise search space with a fixed macro-architecture. In the case of Hardware-aware search space, a restricted pool of operators is typically used. Some architectures' operators are eliminated because they do not perform well on a target hardware or are not supported by target hardware Intermediate Representation (IR). In this work, we want our search space to be more hardware-friendly, therefore, we did not include batch normalization, depth-wise convolution, squeeze-and-excitation (Howard et al., 2019), and swish activation (Ramachandran et al., 2017).

Search strategy defines how candidate architectures are explored from the search space, either with a generator or at random, and how the exploring strategy is updated to find promising architectures according to an objective function. Search strategy algorithms usually fall into one of two categories: Reinforcement Learning (RL) (Zoph & Le, 2017; Zoph et al., 2018) and Evolutionary Algorithms (EA) (Real et al., 2017; 2019). When using Reinforcement Learning, a recurrent neural network (RNN) controller first samples a candidate architecture, and then trains it to convergence to measure its performance on the task of desire, for example, accuracy on image classification. The controller then uses the performance result as a guiding signal to find more promising architectures. This process is repeated for many iterations. In the case of Evolutionary Algorithms, the initial population is generated by randomly sampling architectures. Each architecture of the initial population is trained using a train dataset and evaluated its performance on validation dataset. Based on the performance, a small portion of architectures which are top ranked are selected. The selected architectures, called an elite population, are applied through a series of transformations to generate a new population. These transformations are mutations and crossover operations (Real et al., 2017; Benmeziane et al., 2021). Mutation transforms one architecture into another through deletion, insertion, substitution procedure. These mutation procedures are inspired by Genetics. Genetic mutation is a change in a DNA sequence since deletion, insertion, substitution happen in a mistake as DNA sequence is being copied (NIH, a). While crossover operations merge two architectures into a new one by randomly selecting each component of sequence from the two architectures. For search strategy, we are using EA and transforming a neural architecture to another one using mutations and crossover operations.

Evaluation methodology To guide Search strategy, the performance of a candidate architecture is needed to be evaluated. To evaluate the accuracy, which is the most popular performance metric, the architecture is needed to be trained to convergence. Training a candidate architecture to convergence requires intensive computing and usually takes several GPU hours. One of the alternative ways to speed up the evaluating process is to estimate an accuracy without training an architecture. For example, once-for-all network (Cai et al., 2020) decouples search process and training process using over-parameterized networks as a super-network and progressive shrinking algorithm to progressively fine-tune smaller sub-networks that share weights with the larger ones. BRP-NAS (Łukasz Dudziak et al., 2021) and ProxylessNAS (Cai et al., 2019) binarized the architecture parameters(1 or 0). NPENAS (Wei et al., 2020) proposed accuracy predictor based on Bayesian optimization or a graph-based neural network. Then, the accuracy predictor is implemented in Evolutionary Algorithm. However, their ranking method is mono-objective and takes 2 or 3 GPU days for search. Neural predictor (Wen et al., 2019) built Neural Predictor using a regression model. However, the correlation coefficient of validation data is only 0.649, which is much lower than 0.9635 in this work. For predicting performance without training, Blockswap (Turner et al., 2020) used block substitution for neural architecture search, and choose networks with block by passing a single minibatch of training data through randomly initialized networks and gauging their Fisher potential. For performance, they used parameter count instead of inference time. It takes under 5 minutes on a single GPU. Ours takes under 1 minute on a single CPU. Zero-Cost Proxies (Abdelfattah et al., 2021) used a single mini-batch of training data to compute a model's score as an accuracy predictor. However, the correlation coefficient of validation data is only 0.82. Distilling Optimal Neural Networks (Moons et al., 2021) built accuracy predictor based on blockwise knowledge distillation. The correlation coefficient of valid accuracy is 0.91. In our accuracy evaluating method, we built an accuracy predictor using more than 2,000 of (architecture, accuracy) pairs. Then, we estimated accuracy for a given architecture, which is 1000x less expensive in computing than general NAS evaluation methodology. Our search procedure takes less than 1 minute using a single CPU.

Hardware cost evaluation To apply NAS on a specific hardware, the constraints should be considered. For example, μNAS (Liberis et al., 2020) used a set of constraints of peak memory usage, model size and latency as hardware metrics. Hardware-agnostic metrics such as number of parameters or FLOPs do not guarantee low inference latency on different hardware platform (Li et al., 2021). While these metrics have been commonly used to estimate the hardware cost, many works have pointed out that DNNs with fewer FLOPs are not necessarily faster or more efficient (Tan et al., 2019; Wu et al., 2019; Zhang et al., 2020). μNAS (Liberis et al., 2020) used the number of multiply-accumulate operations as a proxy for model latency. However, they claimed that FLOPs is not good proxy for latency, which conflicts each other. Hardware metrics such as inference latency time or power consumption need to be measured by a real-time execution of each architecture on a targeted platform. Since the measured latency or power value are not persistent but fluctuate in time, averaging the values after measuring many times is required. To reduce the burden of the cost of

measuring hardware metrics, once-for-all network (Cai et al., 2020) calculated hardware latency by summing up the latency value of all operations in an architecture based on a look-up table for each operation. However, we found that there is a gap between the calculated latency value based on a look-up table and measured latency value. This gap is quite sensitive to input image size or batch size. Our guess is that data loading, tiling/scheduling, data movement may additionally contribute to real-world latency on a hardware platform. BRP-NAS (Łukasz Dudziak et al., 2021) also proposed a Graph convolutional networks-based predictor for latency. However, they did not report the correlation coefficient of validation data for the latency predictor. In this work, we developed a latency estimator based on measured latency values using a DNN. The correlation coefficient of our latency estimators are 0.9885 on AMD EPYC CPU, 0.9333 on Intel Xeon CPU and 0.9865 on Xilinx Zynq FPGA for validation samples.

Multi-objective optimization HW-NAS needs to adapt optimization method to achieve its multi-objects. We compared two state of art HW-NAS methods, which are Once-for-all networks (Cai et al., 2020) and MnasNet (Tan et al., 2019). We implement optimization method of once-for-all networks in Fig.1 and a optimization method of MnasNet in Fig.13 and 14. Codesign-NAS (Abdelfattah et al., 2020) defined the multi-objective problem as a sequence of multiplication of objects, which is similar to MnasNet (Tan et al., 2019). Then, the object sequence is treated as reward for reinforcement learning. MOBO (Paria et al., 2019) proposed a Multi-Objective Bayesian Optimization algorithm for exploration of specific parts of Parent front. μNAS (Liberis et al., 2020) used Bayesian optimization for search algorithm. Distilling Optimal Neural Networks (Moons et al., 2021) used evolutionary search to find pareto-optimal architectures using the accuracy predictor and on-device measurement. Pareto-optimization is another multi-objective optimization algorithm (Deb et al., 2002; Tan et al., 2019; Dong et al., 2018). Pareto frontier is a set of Pareto optimal solutions. We tried to apply Pareto-optimal algorithm based on distance metric in NSGA-II (Deb et al., 2002). The Pareto-optimal points were located in the left bottom portion from population in the graph of accuracy versus latency, which is similar with Fig.3 in NSGA-II (Deb et al., 2002). However, we need top front points over all x-axis (latency) as shown in Figure 11. We proposed a simple alternative method, which is based on epsilon-constraint method (Yang et al., 2014).

3 METHODS

To demonstrate the efficiency of our proposed multi-object optimization method, we built our own search space based on VGGNet (Simonyan & Zisserman, 2015) and used EA as our search strategy. We aim to discover neural architectures with high accuracy and low latency on our target device, Xilinx Zynq ZU9EG FPGA on ZCU102 evaluation board. FPGA has been used as an AI acceleration platform with a high flexibility in terms of the hardware resources. The FPGA chip on the board can be directly converted to an embedded system without extra work. In addition, we evaluated latency on AMD EPYC 7F52 16-Core CPU as a baseline for latency evaluation.

For our search space, we imported blocks from VGGNet and stack them to form a architecture. Since VGGNet was introduced at ILSVRC (ImageNet Large Scale Visual Recognition Competition) 2014, the architecture has become one of the most popular architectures for many applications including image classification and object detection. VGGNet consists of 5 blocks of convolutional layers and one block of dense layers, where the number of convolutional layers in each block is different depending on the sub-networks. VGGNet has 4 sub-networks, denoted as 11, 13, 16, 19 by counting the total number of convolutional layers and dense layers in the architecture. The architectures for 4 VGG sub-networks are illustrated in Appendix.

We built a super-network based on VGG-19 architecture. The architecture of our super-network is illustrated in Figure 2 (Top). In the super-network, the first two blocks consist of 2 convolution layers, the next three blocks consist of 4 convolution layers, and the last block consists of 2 dense layers. The flatten layer from original VGGNet is substituted with global average pooling layer to reduce wide bandwidth during inference. The super-network includes the largest values for the number of blocks, and layers, and channels. This super-network was trained only once to get the weights of all the parameters of every sub-network. Then, we trained each sub-network using the pre-trained weights of the parameters of the super-network instead of starting from random initialization.Our approach speeds up the convergence of sub-networks and provides the consistent result of the accuracy of the sub-network.

Directly applying NAS to a large-scale ImageNet dataset (Deng et al., 2009) is computationally too expensive or impossible using a single GPU. To reduce the training time, we choose 20 classes from the original 1000 classes to train the super-network and sub-networks. The 20 classes are chosen with the least number of images per class in the training dataset. Our architecture search framework is implemented in keras (Chollet et al., 2015). We applied transfer learning to train the super-network. Using the pre-trained weight for the original 1000 classes from keras library, we fine-tuned the super-network of the 20 classes for 60 epochs with learning rate as 0.001 and stochastic gradient descent (SGD) optimizer.

We built a search space based on the super-network. The architecture of search space is illustrated in Figure 2 (Bottom). In the search space, the number of layers of each block and channels of each layer need to be optimized with a given input image size. The kernel size of convolution operators is fixed in this work, but can be optimized further. The input image size can be chosen from [128, 160, 192, 224, 256]; the number of layers of each block (denoted as depth) can be chosen from [1,2] for the first two convolutional blocks, [1,2,3,4] for the next three convolutional blocks, [0,1,2] for the last dense block; Each block has its own number of channels as [64, 128, 256, 512, 512, 4096]. The number of channels in each layer (denoted as width) can be chosen from the ratio of [0.25, 0.5, 0.75, 1] to the original number; for the kernel size, we fixed it as 3x3 convolution. Therefore, the size of the search space is roughly $((4^2+4)^2 \times (4^4+4^3+4^2+4)^3 \times (4^2+4+1)) \times 5 \approx 10^{12}$. Each neural network architecture is encoded as a numerical sequence vector like the string of a DNA sequence. Then the sequence vector is transformed to a new vector by mutating the sequence or crossing over two sequences as described in Section 2. The new sequence vector is decoded as a new neural architecture. Applying these transformations many times, a new population is generated.

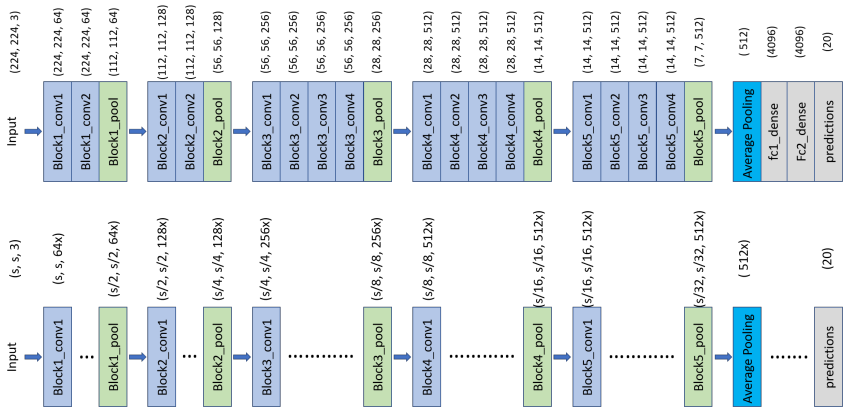

Figure 2: **Top:** the architecture of super-network in this work **Bottom:** Search space in this work, where s is input image size in [128, 160, 192, 224, 256], x is width ratio in [0.25, 0.5, 0.75, 1], the number of layer in each block (depth) is needed to be optimized.

To address the huge computing cost and speed up the evaluating procedure of accuracy and latency, we built an accuracy predictor for any architecture in the search space and a latency estimator with a given target hardware. We collected more than 2,000 of (architecture, accuracy) pairs after training architectures and measuring accuracy against the validation dataset. Based on the pairs, we built an accuracy predictor using a DNN. The 2,000s of architectures were generated using EA and fine-tuned using the pre-trained weight of the super-network for 10 epochs with learning rate 0.001 and SGD optimizer. Training with the pre-trained weight from the super-network produces consistent accuracy of sub-networks, which makes it possible to build an accuracy predictor. The collected (architecture, accuracy) pairs were used to train an accuracy predictor. We used two hidden layers of a neural network that has 800 neurons in each layer as the accuracy predictor. Given an architecture, we encode input image size, the number of layers in each block, the number of channels in each layer into a one-hot vector, respectively. Then, we concatenate these vectors into one large vector that represents a whole architecture. The one-hot vector is then fed to the multi-layer neural network to predict accuracy. The architecture of the accuracy predictor and the correlation between predicted accuracy and actual one are shown in Appendix. Our predictor-based approach accelerates the NAS search process by significantly reducing heavy search cost.

Since model size or FLOPs does not guarantee high inference speed on a hardware platform, measuring real-world latency is required. However deploying a model on a certain hardware platform and directly measuring latency hundred times can be another complicated data collection pipeline. Therefore, we built a latency estimator based on real-world latency values. In the case of CPU, we measured 100 times latency values after warming up 50 times for each architecture on AMD EPYC 7F52 processor. Then, we averaged them for thousands of architecture. In the case of FPGA, we measured 30 times latency values for each architecture on Xilinx Zynq ZU9EG after quantization, compiling, and runtime with Xilinx Intermediate Representation (XIR) and Deep-Learning processor Unit (DPU) (Xilinx, a). Our latency estimator provides more reliable latency data without a complicated hardware measurement pipeline.

Using EA as a search strategy, the typical searching procedure is described as follows. The initial population is generated with random sampling. We fixed the population size as 100. The architectures of the initial population are evaluated for their accuracy and latency. Then, the architectures which accuracy are within the top 25 percentile are selected as elite population. From the elite population, new architectures are constructed by applying transformation procedures such as mutation and crossover, which cause a random modification of the architecture and are described in detail in Section 2. The new architectures are added to the next generation of population. Once the next generation is constructed with its fixed population size, the architectures inside of the next generation are evaluated for their accuracy and latency for selecting elite population. This procedure is repeated many cycles as illustrated in Figure 5 to evolve and get improved architecture in terms of accuracy and latency. For each cycle, a new elite population is selected by competing the next generation of population with the previous elite population. This process is called tournament evolution (Goldberg & Deb, 1991; Real et al., 2017). Meanwhile, if the new elite population is selected directly from the next population without competing with the previous elite population, it is called aging evolution (Real et al., 2019).

As an architecture selection process in EA, we propose upfront, a simple and efficient method to select the elite population based on epsilon-constraint method (Yang et al., 2014). Upfront algorithm divides current population by small cell based on hardware constraint Threshold ΔT grid. The grid can be one-dimensional if there is only one hardware constraint such as latency or power consumption, or two-dimensional if there are two hardware constraints. Then, Upfront selects a top accuracy architecture within each cell. The selected architectures would be promising candidates in terms of accuracy and hardware constraints. The detailed implementation is described in Appendix.

General NAS can be mathematically described as an optimization problem to find an architecture a with the highest accuracy $acc(a)$ on the validation dataset within the search space A.

$$\arg\max_{a \in A} acc(a) \tag{1}$$

Meanwhile, HW-NAS seeks to find an architecture a that achieves the highest accuracy $acc(a)$ while the inference latency $lat(a, h)$ is under Threshold T for a given hardware platform h.

$$\begin{aligned} &\arg\max_{a \in A} acc(a) \\ &\text{s.t. } lat(a, h) \leq T \end{aligned} \tag{2}$$

To implement HW-NAS by mono-objective optimization, there are typically two approaches. One is to search candidate architectures with accuracy like equation 1 by repeating for many iterations. Then, to screen the models by applying latency constraint. The other is to search candidate architectures under latency constraint like equation 2.

To get an optimal point, we should consider multi-objective optimization. When we apply multi-objective optimization on HW-HAS, we should satisfy the following two conditions at the same time.

$$\arg\max_{a \in A} acc(a) \cap \arg\min_{a \in A} lat(a, h) \tag{3}$$

The multi-objective optimization can be practically implemented using the weighted sum of accuracy and latency with fixed weight ratio, w_1 and w_2.

$$\arg\max a \in A[w_1 acc(a) + \frac{w_2}{lat(a, h)}] \tag{4}$$

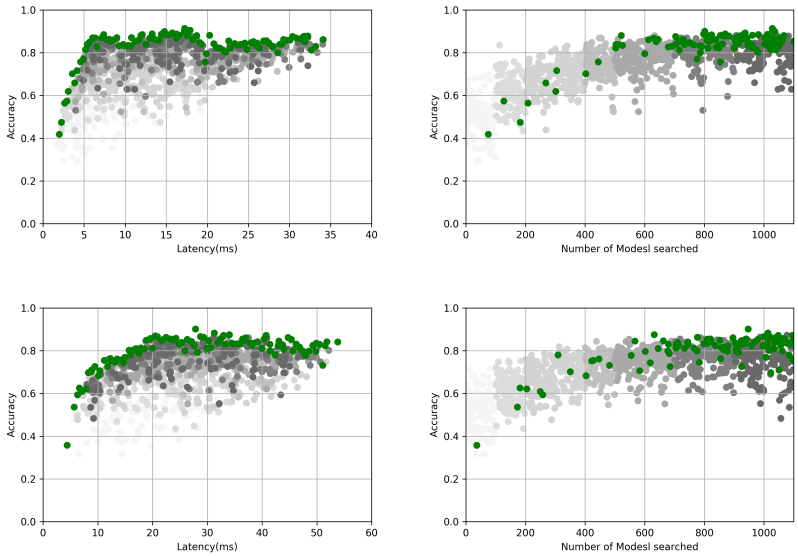

Figure 3: Multi-objective optimized search using this work. **Left:** accuracy versus latency **Right:** search progress of the experiment, where green points are the architectures with the optimal trade-offs between accuracy and inference latency. **Top:** Target hardware is Xilinx Zynq ZU9EG FPGA **Bottom:** Target hardware is AMD EPYC 7F52 CPU.

There is another way to implement multi-objective optimization using a customized weighted product method (Tan et al., 2019). Given an architecture a, accuracy $acc(a)$, latency $lat(a, h)$, and latency Threshold T, the optimization is defined as:

$$\arg\max_{a \in A} acc(a) \times [\frac{lat(a,h)}{T}]^w \tag{5}$$

where w is the weight factor defined as:

$$\begin{aligned} w &= \alpha, \quad \text{if} \quad lat(a,h) \leq T \\ &= \beta, \quad \text{otherwise} \end{aligned} \tag{6}$$

In the case of hard constraint, the suggested weight factor values are α = 0, β = -1. In the case of soft constraint, the weight factor values are $\alpha = \beta$ = -0.07.

In the case of upfront, our multi-object optimization method, we divided hardware performance range as a small cell. Then, we seek max accuracy within each cell ΔT_{ij}.

$$\arg\max_{a \in A} acc(a) \big| \Delta T_{ij} \tag{7}$$

4 EXPERIMENTAL RESULTS

Figure 1 shows evolutionary search results through mono-objective optimized procedure for 10 cycles, and inference latency constraint as 10ms or 20ms. As expected, the validation accuracy improves as search progresses for all the cases. However, the best selected architecture is not guaranteed as the optimal point in terms of accuracy and latency.

Figure 12 shows the search result by applying equation 4 with different weight ratio of w_1 and w_2. When the w_2 is larger than w_1, where the impact of latency is dominant, the accuracy does not improve but decreases as search is in progress. The best accuracy model came from random sampling, not evolutionary search. When the w_2 is much smaller than w_1, where the factor of

latency is minor, the accuracy is improved as searching is in progress. But the evolution progress is quite similar to the case of the mono-objective optimization in Figure 1 without hardware constraint. Weighted product method (Tan et al., 2019) can be an alternative, whose result is shown in detail in Appendix. It turns out that it is very hard to optimize the multi-object using the weighted sum or weighted product.

Figure 3 shows the search result using equation 7. In this work, we used latency as a hardware constraint. The first generation is the initial population from random sampling. After that, accuracy is improved every generation. We highlighted the architectures with optimal trade-offs between accuracy and inference latency as green dots. The green dots are mainly distributed in recent generations. From the highlighted architectures, we have freedom to choose an optimal architecture depending on hardware constraint of inference latency as one-shot.

We compare typical mono-objective search results and multi-objective search results using upfront in Figure 4. The data points came from the optimal architectures by considering both accuracy and latency from Figure 1 and Figure 3. We also include the results from VGGNet 11/13/16/19 sub-networks with 20 classes and global average pooling layer. The VGGNet sub-networks were fine-tuned with the same hyper-parameters for the other sub-networks. They were trained with the pre-trained weight of the super-network for 10 epochs, learning rate as 0.001 and SGD optimizer. The VGGNet sub-networks show a typical trade-off between accuracy and latency. Meanwhile, the optimized architectures using EA can keep its good accuracy even with very low latency regimes. The benefit of EA in a low latency regime is much obvious in the case of multi-objective optimization search using upfront.

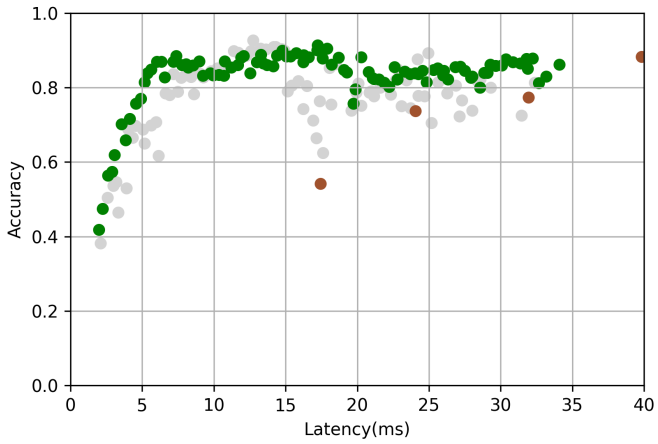

Figure 4: Comparison between mono-objective search and multi-objective search in Accuracy versus latency. Target hardware is Xilinx Zynq ZU9EG FPGA. The gray dots are mono-objective search result and green dots are multi-objective search result using upfront algorithm. The brown dots are VGGNet 11/13/16/19 sub-networks.

5 CONCLUSIONS

This work presents a simple and efficient approach for multi-objective optimization for neural architecture search targeting a certain hardware platform. Our main contribution is implementing epsilon-constraint based multi-objective optimization algorithm, selecting an architecture whose accuracy is the top within each cell of hardware cost metrics. This produces optimized architectures which run faster and achieve similar accuracy. We also were able to significantly reduce computing cost of the architecture search process by building both accuracy predictor and latency estimator with high correlation coefficient of validation samples. Searching procedure takes under 1 minute on a single CPU in this work.

REFERENCES

Mohamed S. Abdelfattah, Łukasz Dudziak, Thomas Chau, Royson Lee, Hyeji Kim, and Nicholas D. Lane. Best of both worlds: Automl codesign of a cnn and its hardware accelerator, 2020.

Mohamed S. Abdelfattah, Abhinav Mehrotra, Łukasz Dudziak, and Nicholas D. Lane. Zero-cost proxies for lightweight nas, 2021.

Hadjer Benmeziane, Kaoutar El Maghraoui, Hamza Ouarnoughi, Smail Niar, Martin Wistuba, and Naigang Wang. A comprehensive survey on hardware-aware neural architecture search, 2021.

Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware, 2019.

Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment, 2020.

François Chollet et al. Keras, 2015. https://keras.io.

K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002. doi: 10.1109/4235.996017.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255, 2009. doi: 10.1109/CVPR.2009.5206848.

Jin-Dong Dong, An-Chieh Cheng, Da-Cheng Juan, Wei Wei, and Min Sun. Dpp-net: Device-aware progressive search for pareto-optimal neural architectures, 2018.

Zhen Dong, Yizhao Gao, Qijing Huang, John Wawrzynek, Hayden K. H. So, and Kurt Keutzer. Hao: Hardware-aware neural architecture optimization for efficient inference, 2021.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Efficient multi-objective neural architecture search via lamarckian evolution, 2019.

David E. Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of Genetic Algorithms*, pp. 69–93. Morgan Kaufmann, 1991.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for mobilenetv3, 2019.

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, 2017.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, 2012.

Chaojian Li, Zhongzhi Yu, Yonggan Fu, Yongan Zhang, Yang Zhao, Haoran You, Qixuan Yu, Yue Wang, and Yingyan Lin. Hw-nas-bench:hardware-aware neural architecture search benchmark, 2021.

Edgar Liberis, Łukasz Dudziak, and Nicholas D. Lane. μnas: Constrained neural architecture search for microcontrollers, 2020.

Bert Moons, Parham Noorzad, Andrii Skliar, Giovanni Mariani, Dushyant Mehta, Chris Lott, and Tijmen Blankevoort. Distilling optimal neural networks: Rapid search in diverse spaces, 2021.

NIH. Mutation, a. https://www.genome.gov/genetics-glossary/Mutation.

Biswajit Paria, Kirthevasan Kandasamy, and Barnabás Póczos. A flexible framework for multi-objective bayesian optimization using random scalarizations, 2019.

Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions, 2017.

Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc Le, and Alex Kurakin. Large-scale evolution of image classifiers, 2017.

Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search, 2019.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions, 2014.

Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. Mnasnet: Platform-aware neural architecture search for mobile, 2019.

Jack Turner, Elliot J. Crowley, Michael O'Boyle, Amos Storkey, and Gavin Gray. Blockswap: Fisher-guided block substitution for network compression on a budget, 2020.

Chen Wei, Chuang Niu, Yiping Tang, Yue Wang, Haihong Hu, and Jimin Liang. Npenas: Neural predictor guided evolution for neural architecture search, 2020.

Wei Wen, Hanxiao Liu, Hai Li, Yiran Chen, Gabriel Bender, and Pieter-Jan Kindermans. Neural predictor for neural architecture search, 2019.

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, 2019.

Xilinx. Xilinx vitis ai user guide, a. https://www.xilinx.com/vitis-ai/ug1414-vitis-ai.

Zhixiang Yang, Xinye Cai, and Zhun Fan. Epsilon constrained method for constrained multiobjective optimization problems: Some preliminary results. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO Comp '14, pp. 1181–1186, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328814. doi: 10.1145/2598394.2610012. URL https://doi.org/10.1145/2598394.2610012.

Li Lyna Zhang, Yuqing Yang, Yuhang Jiang, Wenwu Zhu, and Yunxin Liu. Fast hardware-aware neural architecture search, 2020.

Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning, 2017.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition, 2018.

Łukasz Dudziak, Thomas Chau, Mohamed S. Abdelfattah, Royson Lee, Hyeji Kim, and Nicholas D. Lane. Brp-nas: Prediction-based nas using gcns, 2021.

A SEARCHING PROCEDURE

Our overall searching procedure in EA is illustrated in Figure 5.

B VGG SUB-NETWORKS

We illustrated the architectures of 4 VGG sub-networks in Figure 6. Each sub-network consists of 5 convolutional blocks and 1 dense block. The size of the feature map is decreased by half for each block due to the pooling layer. The number of layers in each convolutional block varies depending on the VGG sub-networks.

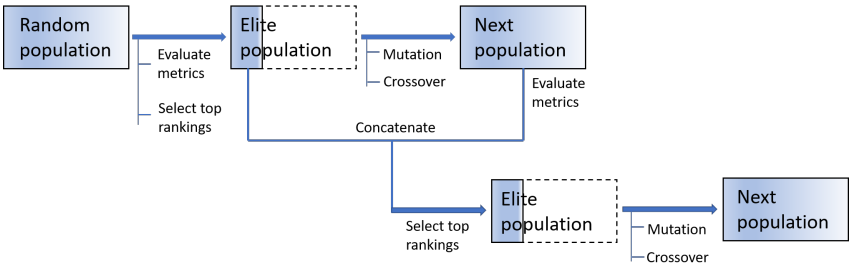

Figure 5: Searching procedures in Evolutionary Algorithms. The procedure of selecting elite population and generating next population is repeated many cycles.

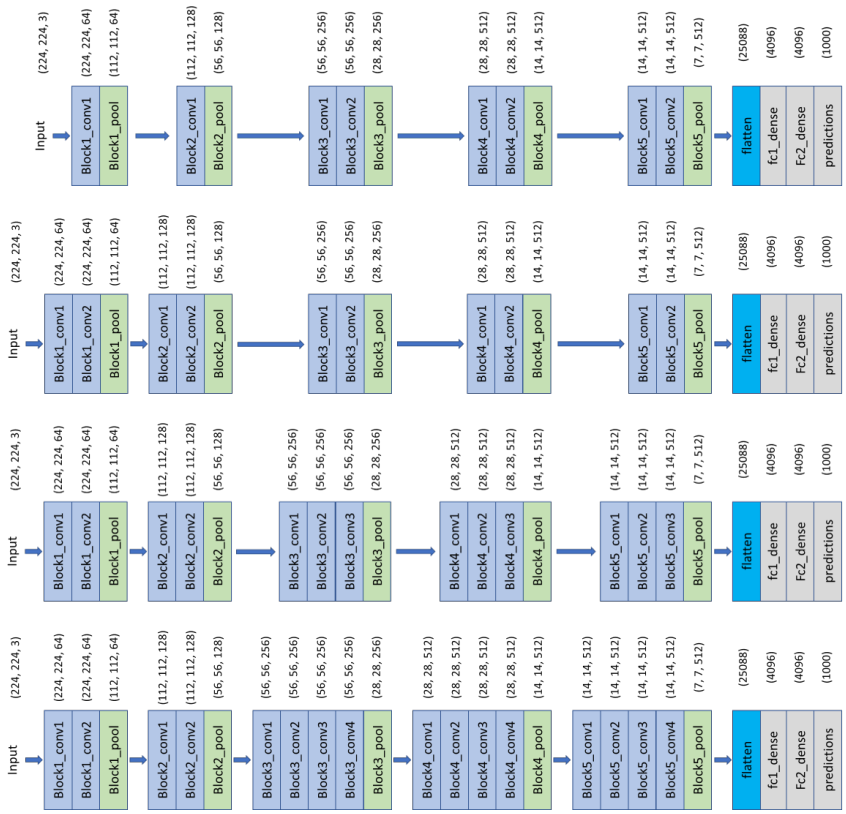

Figure 6: The architecture of VGGNet sub-networks; VGGNet-11,13,16,19 from top to bottom

C PREDICTING ACCURACY

Figure 7(Left) shows correlation between predicted accuracy and actual accuracy using our accuracy predictor. The accuracy predictor is a Deep Neural Network(DNN) with two hidden layers and predicts accuracy with a given architecture. The architecture of the predictor is illustrated in Figure 7(Right). The input format is a one-hot vector to represent the architecture of a sub-network. The Pearson correlation coefficient of train and validation data is 0.9978 and 0.9635 respectively. This high correlation is available since the sub-networks are trained with the weight parameters from the super-network.

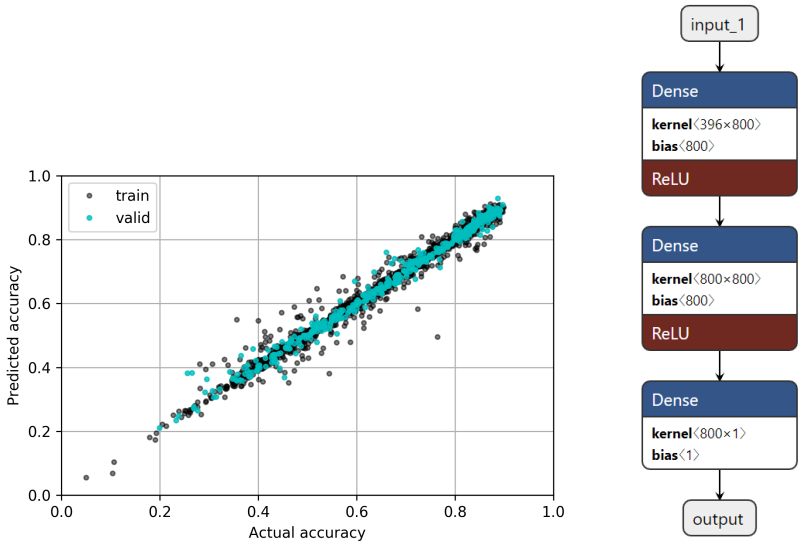

Figure 7: **Left:** Predicted accuracy versus Actual accuracy. The Pearson Correlation Coefficient of valid data is 0.9635. **Right:** the architecture of accuracy predictor

D PREDICTING LATENCY

Figure 8(Top) shows the relationship between actual latency and FLOPs on AMD EPYC 7F52 CPU. Figure 8(Bottom) shows correlation between predicted latency and actual latency on AMD EPYC 7F52 CPU using our latency predictor. The latency estimator is a DNN with two hidden layers and predicts CPU latency with a given architecture. The architecture of the estimator is the same as illustrated in Figure 7(Right). The input format is a one-hot vector to represent the architecture of a sub-network. The Pearson correlation coefficient of train and validation data is 0.9998 and 0.9885 respectively.

Figure 9(Top) shows the relationship between actual latency and FLOPs on Intel Xeon(R) Gold 6252 CPU. It is interesting that the relationship of latency versus FLOPs are not linear in the case of Intel CPU. Figure 9(Bottom) shows correlation between predicted latency and actual latency on Intel Xeon(R) Gold 6252 CPU using our latency predictor. The architecture of the latency estimator is the same as illustrated in Figure 7(Right). The input format is a one-hot vector to represent the architecture of a sub-network. The Pearson correlation coefficient of train and validation data is 0.9748 and 0.9333 respectively.

In the case of FPGA, we could not collect actual FPGA latency for thousands of sub-networks since the sub-networks are required to be quantized, compiled, and deployed on a FPGA board to measure latency. Instead, we counted FLOPs of thousands of sampled sub-networks. Then, based on its FLOPs, hundreds of sub-networks were selected, so FPGA latency values would be evenly distributed over a wide range. Then, we executed the selected sub-networks on a FPGA board after quantization, compiling, and runtime with Xilinx Intermediate Representation (XIR) and Deep-Learning processor Unit (DPU) (Xilinx, a). The DPU is a programmable engine optimized for deep neural networks and a group of parameterizable IP cores pre-implemented on the hardware. The DPU defines which operators to support depending on the DPU types. The XIR is a graph-based intermediate representation of the machine learning algorithms which is designed for compilation and efficient deployment of the DPU on the FPGA platform. After all, we measured actual inference time of the selected sub-networks on a FPGA board and calculated latency as the mean value for the inference times with 30 different input images.

Figure 10 (Top) shows the correlation between the measured FPGA latency versus FLOPs. The Pearson Correlation coefficient is 0.9939, which shows very strong correlation between the actual

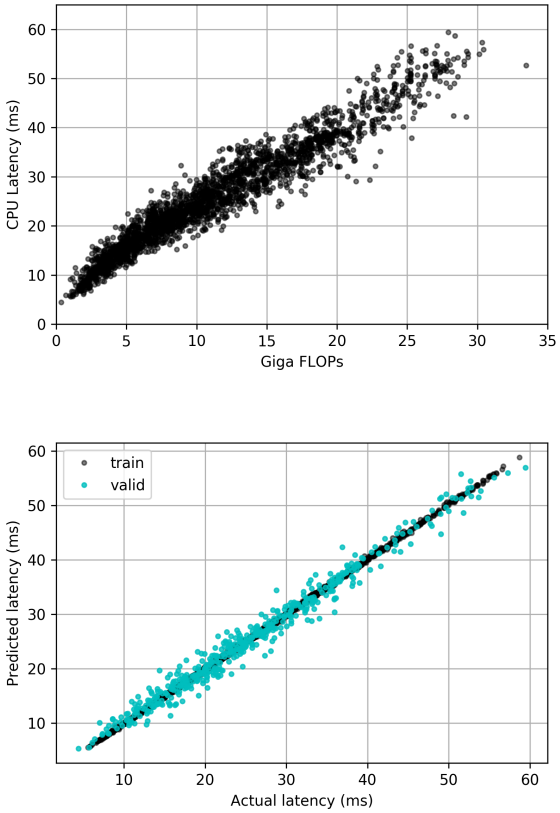

Figure 8: Hardware: AMD EPYC 7F52 CPU, **Top:** CPU latency versus FLOPs **Bottom:** Predicted latency versus Actual latency

latency and the FLOPs. Thus, FPGA latency is calculated based on the FLOPs of an architecture using a linear regression model.

Using the calculated latency values and its architecture, we built an latency estimator. The architecture of FPGA latency estimator is same as illustrated in Figure 7(Right). The input format is one-hot vector to represent the architecture of a sub-network. Figure 10 (Bottom) shows the correlation between predicted latency and actual latency on Xilinx Zynq ZU9EG FPGA using our latency predictor. The Pearson correlation coefficient of train, validation, test data is 0.9996, 0.9892, and 0.9865 respectively.

E UPFRONT IMPLEMENTATION

We propose upfront, an algorithm to select top candidates from a population in terms of accuracy with a given hardware constraints. The hardware constraints can be one variable such as latency or two variables consisting of latency and power consumption. In this illustration, we show latency as the hardware constraint. The algorithm divides current population into small cells based on hardware-metric grid. Then, the algorithm selects a top accuracy architecture within each cell. The selected architectures would be promising candidates in terms of accuracy and hardware constraints as illustrated in Figure 11 .

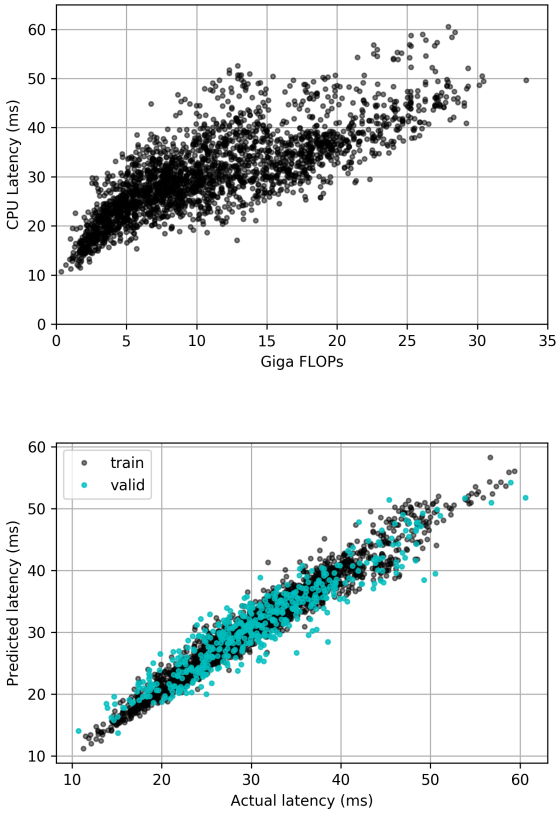

Figure 9: Hardware: Intel Xeon(R) Gold 6252 CPU, **Top:** CPU latency versus FLOPs **Bottom:** Predicted latency versus Actual latency

F MULTI-OBJECTIVE OPTIMIZED SEARCH BY A WEIGHTED SUM METHOD

Figure 12 shows the searching result for the weighted sum of accuracy and latency with different weight ratios.

G MULTI-OBJECTIVE OPTIMIZED SEARCH BY A WEIGHTED PRODUCT METHOD

Figure 13 and Figure 14 show the searching results for the weighted product method with hard and soft constraint respectively. The accuracy is improved as searching is in progress. But the evolution progress is quite similar to the case of the mono-objective optimization as shown in Figure 1. This approach does not guarantee that the final output is optimized in terms of multi-object.

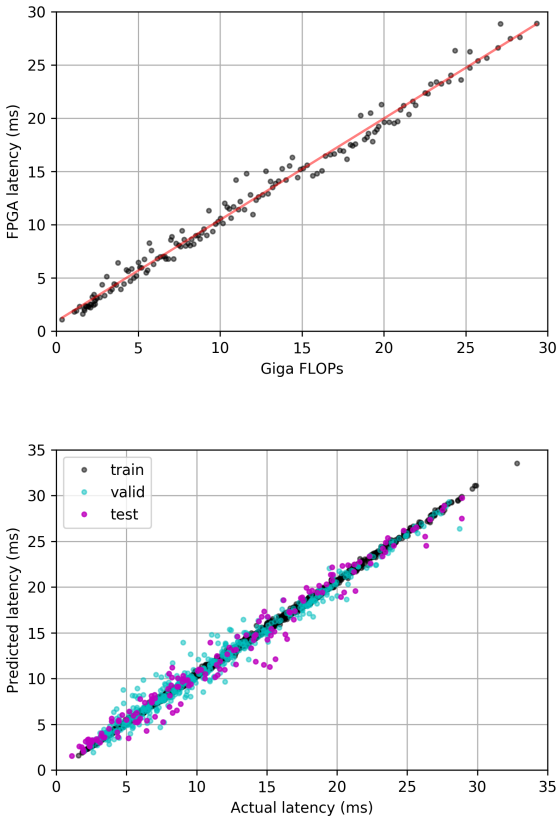

Figure 10: Hardware: Xilinx Zynq ZU9EG FPGA, **Top:** FPGA latency versus FLOPs. The red line is the linear regression with y intercept 0.9524 and slope 0.9215. **Bottom:** Predicted latency versus Actual latency. The train and valid data are calculated values using the linear regression. The test data is measured value.

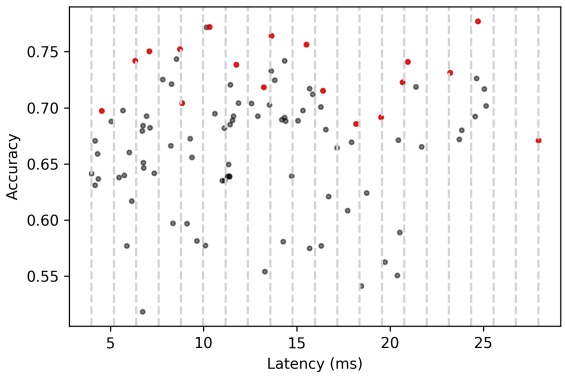

Figure 11: Example of selecting top accuracy architecture within each cell based on latency grid. The red dot is the selected architecture from each cell.

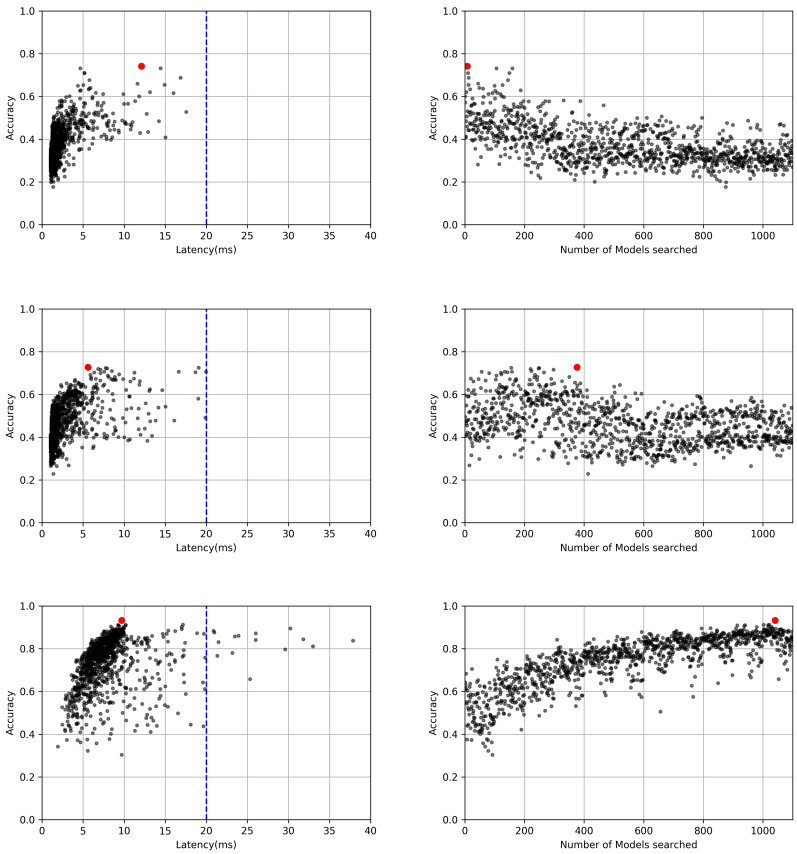

Figure 12: Multi-objective optimized search by applying the weighted sum of accuracy and latency. **Left:** accuracy versus latency **Right:** search progress of the experiment, Red dot is the best accuracy model under the latency constraint 20ms. **Top:** the weight ratio of accuracy and latency is 1/5. **Middle:** the weight ratio is 1/1. **Bottom:** the weight ratio is 1/0.5.

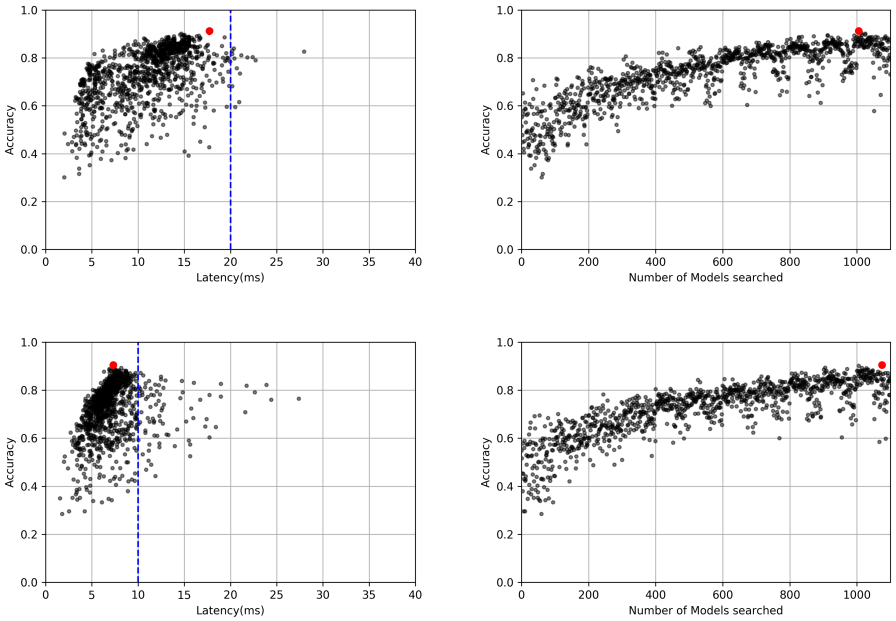

Figure 13: Multi-objective optimized search by applying weighted product method with hard constraint **Left:** accuracy versus latency **Right:** search progress of the experiment, Red dot is the best accuracy model under the latency constraint 20ms. **Top:** target latency T = 20ms **Bottom:** target latency T = 10ms

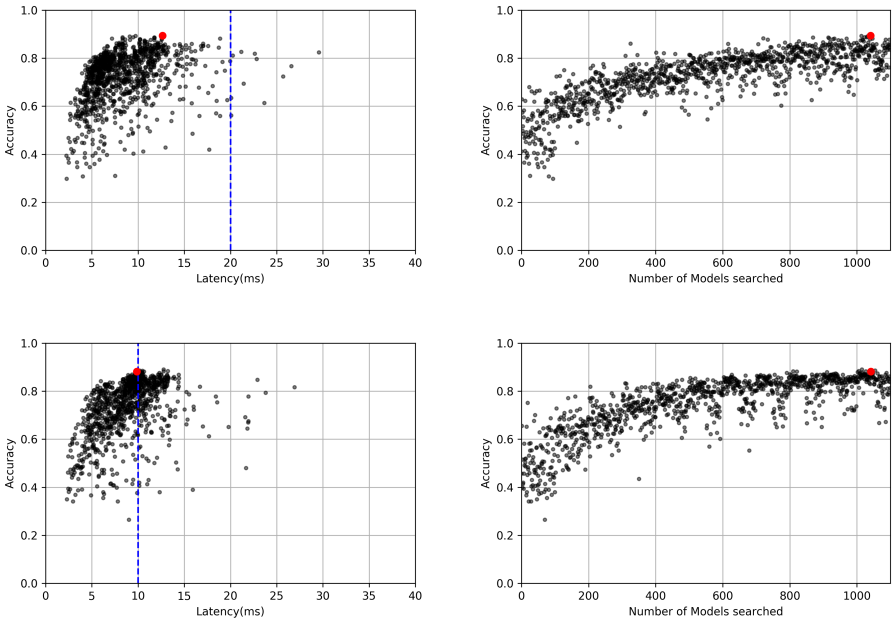

Figure 14: Multi-objective optimized search by applying weighted product method with soft constraint **Left:** accuracy versus latency **Right:** search progress of the experiment, Red dot is the best accuracy model under the latency constraint 20ms. **Top:** target latency T = 20ms **Bottom:** target latency T = 10ms