CODEBench: A Neural Architecture and Hardware Accelerator Co-Design Framework

SHIKHAR TULI and CHIA-HAO LI, Princeton University, USA
RITVIK SHARMA, Stanford University, USA
NIRAJ K. JHA, Princeton University, USA

Recently, automated co-design of machine learning (ML) models and accelerator architectures has attracted significant attention from both the industry and academia. However, most co-design frameworks either explore a limited search space or employ suboptimal exploration techniques for simultaneous design decision investigations of the ML model and the accelerator. Furthermore, training the ML model and simulating the accelerator performance is computationally expensive. To address these limitations, this work proposes a novel neural architecture and hardware accelerator co-design framework, called CODEBench. It comprises two new benchmarking sub-frameworks, CNNBench and AccelBench, which explore expanded design spaces of convolutional neural networks (CNNs) and CNN accelerators. CNNBench leverages an advanced search technique, Bayesian Optimization using Second-order Gradients and Heteroscedastic Surrogate Model for Neural Architecture Search, to efficiently train a neural heteroscedastic surrogate model to converge to an optimal CNN architecture by employing second-order gradients. AccelBench performs cycle-accurate simulations for diverse accelerator architectures in a vast design space. With the proposed co-design method, called Bayesian Optimization using Second-order Gradients and Heteroscedastic Surrogate Model for Co-Design of CNNs and Accelerators, our best CNN–accelerator pair achieves 1.4% higher accuracy on the CIFAR-10 dataset compared to the state-of-the-art pair while enabling 59.1% lower latency and 60.8% lower energy consumption. On the ImageNet dataset, it achieves 3.7% higher Top1 accuracy at 43.8% lower latency and 11.2% lower energy consumption. CODEBench outperforms the state-of-the-art framework, i.e., Auto-NBA, by achieving 1.5% higher accuracy and 34.7× higher throughput while enabling 11.0× lower energy-delay product and 4.0× lower chip area on CIFAR-10.

CCS Concepts: • **Hardware** → **Hardware-software co-design**; • **Computing methodologies** → *Machine learning*; • **Computer systems organization** → *Embedded hardware;*

Additional Key Words and Phrases: Active learning, application-specific integrated circuits, hardware-software co-design, machine learning, Neural Architecture Search, neural network accelerators

S. Tuli and C.-H. Li contributed equally to this research.

This work was supported by NSF Grant No. CCF-1811109.

Authors' addresses: S. Tuli, C.-H. Li, and N. K. Jha, Princeton University, Department of Electrical and Computer Engineering, Princeton, NJ 08544, USA; emails: {stuli, chli, jha}@princeton.edu; R. Sharma, Stanford University, Department of Electrical Engineering, Stanford, CA 94305, USA; email: rsharma3@stanford.edu.

1539-9087/2023/04-ART51 $15.00
https://doi.org/10.1145/3575798

ACM Reference format:
Shikhar Tuli, Chia-Hao Li, Ritvik Sharma, and Niraj K. Jha. 2023. CODEBench: A Neural Architecture and Hardware Accelerator Co-Design Framework. *ACM Trans. Embedd. Comput. Syst.* 22, 3, Article 51 (April 2023), 30 pages.
https://doi.org/10.1145/3575798

1 INTRODUCTION

In the **machine learning (ML)** community, **convolutional neural network (CNN)** accelerator design has gained significant popularity recently [35]. Due to the high computational complexity of CNNs, developing a sophisticated hardware accelerator is essential to improve the energy efficiency and throughput of the targeted task. Custom accelerators, also sometimes called **application-specific integrated circuit– (ASIC)** based accelerators, outperform general-purpose processors like **central processing units (CPUs)** and **graphical processing units (GPUs)** [12, 83]. However, designing ASIC-based accelerators often comes with its own challenges, such as design complexity and long development time. Researchers have constantly strived to design more efficient accelerators in terms of throughput, energy consumption, chip area, and model performance. This has led to a large number of CNN accelerators, each incorporating different design choices and unique hardware modules to optimize certain operations. This requires a plethora of operation-specific hardware modules, sophisticated acceleration methods, and many other hyperparameters that define a vast design space in the hardware domain. Similarly, on the software side, many design choices for a CNN model result in an enormous design space as well. A challenge that remains is to *efficiently* find an optimal pair of software and hardware hyperparameters, in other words, a CNN–accelerator pair that performs well while also meeting the required constraints.

The accelerator design space is immense. There are many hyperparameters and one needs to choose their values while designing an accelerator for the given application [63]. They include the number and size of **processing elements (PEs)** and on-chip buffers, dataflow, main memory size and type, and many more domain-specific modules, including those for sparsity-aware computation and reduced-precision design (more details in Section 2.2). Similarly, the CNN design space is also huge. Many hyperparameters come into play while designing a CNN. They include the number of layers, convolution type and size, normalization type, pooling type and size, structure of the final **multi-layer perceptron (MLP)** head, activation function, training recipe, and many more (further details in Section 2.1). For a given task, one may search for a particular CNN architecture with the best performance. However, this architecture may not be able to meet user constraints on power, energy, latency, and chip area.

In the above context, given a hardware module, many works are aimed at tuning the CNN design instead, to optimize performance [19]. However, this limits us to the exploration of only the CNN design space with no tuning possible in the hardware space. Having a fixed CNN, one could also optimize the hardware (called *automatic accelerator synthesis*) in the context where searching the CNN space may be too expensive, especially for large datasets. This hardware optimization, however, requires a thorough knowledge of the architecture and design of ASIC-based accelerators, leading to long design cycles. However, exploring the CNN design space falls under the domain of **neural architecture search (NAS)**. Advancements in this domain have led to many NAS algorithms, including those employing **reinforcement learning (RL)**, Bayesian optimization, structure adaptation, and so on [18, 59, 74]. However, these approaches have many limitations including suboptimal convergence, slow regression performance, and are limited to fixed training recipes (details in Section 3.1). Tuli et al. [67] recently proposed a model for NAS in the space of transformer ML models [70]. It overcomes many of these challenges by leveraging a heteroscedastic surrogate model to search for the model's design decisions and its training recipe. However, this technique is not amenable to co-design between the two (namely, the accelerator and CNN) design spaces (details in Section 3.3.1).

This work also shows that a one-sided search leads to suboptimal CNN–accelerator pairs. This has led to recent advancements in *co-design* of both the software and hardware. Researchers often leverage RL techniques to search for an optimal CNN–accelerator pair [4, 34, 91]. However, most co-design works only use local search (mutation and crossover) and/or have limited search spaces, e.g., they only search over selected **field-programmable gate arrays (FPGAs)** or microcontrollers [4, 42]. Some works have also leveraged differentiable search of the CNN architecture [16, 41]. However, recent surveys have shown that these methods are much slower than surrogate-based methods and fail to explore potential clusters with higher performance models [57]. Limited search spaces, as shown by some very recent works [26, 67], often lead to suboptimal neural network models and even CNN–accelerator pairs (or, in general, the combination of the hardware architecture and the software algorithm). Thus, expanding existing design spaces in both the hardware and the software regimes is necessary. However, blindly growing these design spaces further prolongs design times and exponentially increases compute resource requirements.

To tackle the above challenges, we propose a framework for comprehensively and simultaneously exploring *massive* CNN architecture and accelerator design domains, leveraging a novel co-design workflow to obtain an optimal CNN–accelerator pair for the targeted application that meets user-specified constraints. These could include not just edge applications with highly constrained power envelopes, but also server applications where model accuracy is of utmost importance.

Our optimal CNN–accelerator pair outperforms the state-of-the-art pair [83], achieving 1.4% higher model accuracy on the CIFAR-10 dataset [21, 36], while enabling 59.1% lower latency and 60.8% lower energy consumption, with only 17.1% increase in chip area. This pair also achieves 3.7% higher Top1 accuracy on the ImageNet dataset while incurring 43.8% lower latency and 11.2% lower energy (with the same increase in chip area). Experiments with our expanded design spaces that include popular CNNs and accelerators, using our proposed framework, show an improvement of 1.5% in model accuracy on the CIFAR-10 dataset, while enabling 11.0× lower **energy-delay product (EDP)** and 34.7× higher throughput, with 4.0× lower area for our CNN–accelerator pair relative to a state-of-the-art co-design framework, namely Auto-NBA [26]. We plan to release trained CNN models, accelerator architecture simulations, and our framework to enable future benchmarking.

The main contributions of this article are summarized next.

- We expand on previously proposed CNN design spaces and present a new tool, called CNNBench, to characterize the vast design space of CNN architectures that includes a diverse set of supported convolution operations, unlike any previous work [41, 44, 80]. We propose `CNN2vec` that employs similarity measures to compare computational graphs of CNN models to obtain a dense embedding that captures architecture similarity reflected in the Euclidean space. We also leverage a new NAS technique, **Bayesian Optimization using Second-order Gradients and Heteroscedastic Surrogate Model for Neural Architecture Search (BOSHNAS)**, for searching our expanded design space. CNNBench also leverages similarity between neighboring CNN computational graphs for weight transfer to speed up the training process. Due to the massive design space, along with `CNN2vec` embeddings, weight transfer from previously trained neighbors, and simultaneous optimization of the CNN architecture and the training recipe, CNNBench achieves state-of-the-art performance while limiting the number of search iterations compared to previous works [72].

- We survey popular accelerators proposed in the literature and encapsulate their design decisions in a unified framework. This gives rise to a benchmarking tool, AccelBench, that runs inference of CNN models on any accelerator within the design space, employing cycle-accurate simulations. AccelBench incorporates accelerators with diverse memory configurations that one could use for future benchmarking, rather than using traditional ASIC templates [26, 78]. With the goal to reap the benefits of vast design spaces [26, 67], AccelBench is the first benchmarking tool for diverse ASIC-based accelerators, supporting variegated design decisions in modern accelerator deployments. Unlike previous works that use FPGAs or off-the-shelf ASIC templates, it builds accelerator designs from the ground up, mapping each CNN in a modular and efficient fashion. It supports 2.28×10^8 unique accelerators, a design space much more extensive than investigated in any previous work.
- To *efficiently* search the proposed massive design space, we present a novel co-design method, **Bayesian Optimization using Second-order Gradients and Heteroscedastic Surrogate Model for Co-Design of CNNs and Accelerators (BOSHCODE)**. To make the search of such a vast design space possible, BOSHCODE incorporates numerous novelties, including a hierarchical search technique that gradually increases the granularity of hyperparameters searched, a neural network surrogate-based model that leverages gradients to the input for reliable query prediction, and an active learning pipeline that makes the search more efficient. Here, by gradients to the input, we mean the gradients to the CNN–accelerator pair simulated in the next iteration of the active learning loop. BOSHCODE is a fundamental pillar for the joint exploration of vast hardware-software design spaces. CODEBench, our proposed framework, combines CNNBench (which trains and obtains the accuracy of any queried CNN), AccelBench (which simulates the hardware performance of any queried accelerator architecture), and the proposed co-design method, BOSHCODE, to find the optimal CNN–accelerator pair, given a set of user-defined constraints. Figure 1 presents an overview of the proposed framework. Figure 1(a) shows how the CNNBench and AccelBench simulation pipelines output the performance values for every CNN–accelerator pair. Figure 1(b) shows how BOSHCODE learns a surrogate model for this mapping from all CNN–accelerator pairs to their simulated performance values. CNNBench trains the CNN model to obtain model accuracy and feeds the checkpoints to AccelBench that obtains other performance measures using a cycle-accurate simulator.

The rest of the article is organized as follows. Section 2 discusses the CNN and accelerator design spaces and highlights the advantages of co-design over one-sided optimization approaches. Section 3 presents the co-design framework that uses BOSHCODE to search for an optimal CNN–accelerator pair. Section 4 describes the experimental setup and baselines considered. Section 5 discusses the results. Finally, Section 6 concludes the article.[1]

2 BACKGROUND AND RELATED WORK

In this section, we present background material and related works in the fields of CNN synthesis and accelerator design.

2.1 CNN Design Space

First, we discuss design choices in the CNN design space. We show how various CNN building blocks contribute to its performance or efficiency. We then discuss popular NAS techniques and hardware-aware NAS.

[1]The associated code for CNNBench is available at https://github.com/jha-lab/cnn_design-space. For the entire CODEBench pipeline, the code is available at https://github.com/jha-lab/codebench.

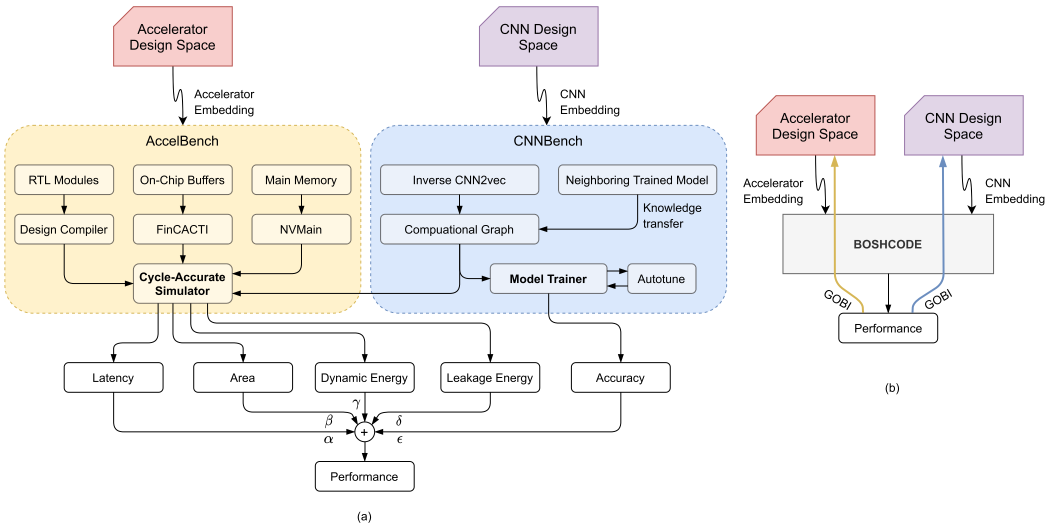

Fig. 1. CODEBench pipeline that includes CNNBench, AccelBench, and a novel co-design method, BOSHCODE. (a) The CNN and accelerator design spaces are sampled for novel CNN–accelerator pairs that are simulated using the CNNBench and AccelBench frameworks. (b) BOSHCODE learns a surrogate function from these design spaces to predict the performance of each CNN–accelerator pair, implementing GOBI to predict the next pair to be simulated in the active learning framework.

2.1.1 *Popular CNN Architectures.*

The CNN design space we consider encompasses popular CNNs of various sizes. The first is LeNet [38], one of the earliest successful CNNs. AlexNet was the first CNN to propose grouped convolutions [37]. MobileNet introduced bottleneck connections that enable parameter reduction [52] (we use MobileNet-V2 as our baseline for subsequent discussions). ResNet proposed residual connections that aided backpropagation of gradients as networks grew deeper [30]. ShuffleNet presented the shuffled convolution, an extension to grouped convolution, that introduced the *channel shuffle* operation for improved regularization and thus efficiency [89]. Other target CNNs we can represent within our design space include GoogleNet, Xception, SqueezeNet, DenseNet, and VGG and EfficientNet families [17, 31, 32, 35, 58, 64, 65].

2.1.2 *Neural Architecture Search.*

NAS incorporates various search techniques that algorithmically find new neural architectures within a pre-defined space based on a given objective [25]. Prior works have implemented NAS using a variety of techniques. A popular approach is to use an RL algorithm, REINFORCE, that is superior to other tabular approaches [74]. Other techniques include **Gaussian-process based Bayesian optimization (GP-BO)** [59], structure adaptation methods based on grow-and-prune synthesis [18], differentiable and proxy-based architecture search [10, 45], local search techniques including mutation, and **evolutionary search (ES)** [73]. Recently, NAS has also seen the application of surrogate models for CNN performance prediction owing to various drawbacks of the methods mentioned above, including high compute cost, slow convergence, and so on [57]. Exploiting surrogate performance results in training much fewer models to predict the accuracy of CNNs in the entire design space under some confidence constraints. However, these predictors are computationally expensive to train, bottlenecking the search process. NASBench-301 [57] uses a **graph isomorphism network (GIN)** that regresses performance on graphs. Recent works show that it is slow enough to bottleneck the search process [67].

One of the state-of-the-art NAS techniques for CNNs, BANANAS [72], implements Bayesian Optimization using a **neural network (NN)** surrogate and predicts performance uncertainty using ensemble networks that are compute-heavy. BANANAS uses mutation and crossover to get the set of current best-performing models and obtains the next best-predicted model in this local space. Instead, we leverage BOSHNAS [67] to efficiently search for the following query in the *global* space curated by our CNNBench framework. Due to random cold restarts, BOSHNAS can search over diverse models in the architecture space. Moreover, BANANAS also uses path embeddings that perform suboptimally in search over a diverse space of CNN models [14]. To address this, we propose a novel embedding, `CNN2vec`, which is dense and thus more amenable to surrogate modeling.

2.1.3 Benchmarking for NAS. Previously, different works on NAS used disparate training pipelines, search spaces, and hyperparameter sets, and did not evaluate other methods under comparable settings. To address these issues, recent works proposed various NAS benchmarks [57, 80]. For instance, BANANAS uses the NASBench-101 dataset to empirically show how it improves upon other NAS algorithms [72]. However, these benchmarks also have their limitations. NASBench-101 [80] is only a tabular dataset of limited size and its results do not transfer well to realistic search spaces [73]. As explained above, NASBench-301 [57] is a surrogate NAS benchmark that uses GIN as its performance predictor. However, a GIN is slow to evaluate and chooses a static training recipe for every CNN model in the design space. In other words, previous works only consider the *epistemic* uncertainty and not the *aleatoric* uncertainty in predictions. The former, also called reducible uncertainty, arises from a lack of knowledge or information, and the latter, also called irreducible uncertainty, refers to the inherent variation in the system to be modeled (in our context, this corresponds to the change in performance of a given CNN architecture with different training recipes).

Another work in this direction trains a large network and derives sub-networks for different tasks and targeted hardware platforms [9]. However, it only considers pyramidal structures, has a limited library of building blocks, uses a static training recipe, and employs sparse embeddings that are not amenable to performance predictors. CNNBench, our proposed benchmarking framework, generates a vast design space of CNNs using an expanded space of building blocks and decouples the surrogate model from the graph modeling process. It does so using the proposed `CNN2vec` embeddings to speed up the search process (details in Section 3.1).

2.1.4 Hardware-aware NAS. NAS alone is hardly useful if one cannot run the best-performing CNNs on the hardware at hand. Recent works have thus focused on *hardware-aware* NAS, which constrains and directs CNN search based on the targeted hardware platform [7]. ChamNet proposed accuracy and resource (latency and energy) predictors and leveraged GP-BO to find the optimal CNN architecture for a given platform [19]. However, it used ES to search for a variant of the base NN that may miss other clusters of high-performing networks, which is a drawback of *elitist* algorithms [20, 57, 72]. Furthermore, it trains a separate latency predictor for every hardware platform, which is a time-consuming endeavor. ProxylessNAS used a proxy-based differentiable search technique. However, it only works with latency and not other hardware performance measures, and is also slower than state-of-the-art surrogate-based methods. Many other works have strived to address hardware-aware NAS, but under a limited scope, often restricted to commercial edge devices, FPGAs, and off-the-shelf ASIC templates [19, 34, 39].

2.2 Accelerator Design Space

Next, we present design choices in the accelerator design space. Many ASIC-based accelerators were proposed in previous works to tackle the task of CNN acceleration, i.e., attain the highest model accuracy while also achieving high energy and area efficiency. These accelerators address the task with various methodologies and custom hardware modules. We present a brief survey of

Table 1. The Hyperparameters and Design Choices of Previous Works on CNN Accelerators

Accelerator Name	Technology Node	Clock Rate (MHz)	Area (mm^2)	Number of PEs	Number of MAC Units Per PE	Number of Multipliers Per MAC Unit	Weight Buffer Size	Activation Buffer Size	Other Buffer Size	Precision	Main Memory Interface	Main Memory Type	Dataflow	Sparsity-Aware Scheme
Eyeriss	65nm CMOS	200	16	168	1	1	108 KB		87 KB	int16	2D	DRAM	RS	Data Gating
Eyeriss V2	65nm CMOS	200	37	192	1	2	192 KB		105 KB	int8	2D	DRAM	RS	CSC Compression
DianNao	65nm CMOS	980	3.02	1	16	16	32 KB	4 KB	8 KB	int16	2D	DRAM	OS	N/A
DaDianNao (1 node)	28nm CMOS	606	67.73	16	16	16	32 MB	4 MB	N/A	int16	2D	DRAM	OS	N/A
ShiDianNao	65nm CMOS	1,000	4.86	64	1	1	128 KB	128 KB	32 KB	int16	2D	DRAM	OS	N/A
Cambricon-X	65nm CMOS	1,000	6.38	16	1	16	32 KB	16 KB	4 KB	int16	2D	DRAM	WS	Step Indexing
Cambricon-S	65nm CMOS	1,000	6.73	16	1	16	32 KB	16 KB	1 KB	int16	2D	DRAM	WS	Direct Indexing
Cnvlutin	65nm CMOS	1,000	71	16	16	16	32 MB	N/A	4 MB	int16	2D	DRAM	OS	ZFNAf Compression
SPRING	14nm FinFET	700	151	64	72	16	24 MB	12 MB	4 MB	int20	Monolithic 3D	RRAM	OS	Binary-Mask

some seminal works on accelerator design and show how they motivate the need for a vast design space and automated search for an optimal architecture.

2.2.1 *Popular Accelerators.*

The Eyeriss v1 and Eyeriss v2 [13] accelerators reduce memory accesses through an efficient row stationary data reuse strategy (also called a *dataflow*). Other dataflows have been investigated, including input stationary, output stationary (OS), and weight stationary (WS) dataflows [63]. Eyeriss v2 further exploits sparsity by directly processing input feature maps and filter weights in the **compressed sparse column (CSC)** format. DianNao [11], DaDianNao [12], and ShiDianNao [23] form another family of accelerators that first enabled sophisticated buffer designs, following the OS dataflow [63]. Cambricon-X [87] and Cambricon-S [90] are successors of the DianNao series, encapsulating dedicated indexing modules to exploit sparsity in NNs. Cnvlutin [6] is another accelerator that focuses on eliminating the ineffectual data in the input feature maps in CNNs and uses the zero-free neuron array format to compress data. Finally, SPRING [83], a state-of-the-art accelerator, leverages three-dimensional (3D) non-volatile memory for increased bandwidth, exploits reduced-precision techniques for data movement efficiency, and OS dataflow for data reuse.

Table 1 lists the hyperparameters and design choices of the aforementioned accelerators. These works propose various methodologies, however, each with its overhead. Thus, performing a rigorous comparison of these methodologies is important [63]. This can be facilitated by benchmarking various accelerator designs to trade off improvements offered by them with the overheads incurred. Moreover, one needs to ascertain the values of many hardware hyperparameters to design an efficient accelerator, such as the number of PEs, size of on-chip buffers, and so on. These hyperparameters allow ASIC-based accelerators to be highly customized for targeted applications and achieve high efficiency. However, the development of such accelerators often comes at the cost of long design cycles and requires considerable computational resources and domain expertise. Hence, many works have strived to automate the accelerator design process.

2.2.2 *Automatic Accelerator Synthesis.*

The accelerator architecture must adequately match the targeted CNN model to achieve optimal CNN acceleration and enable maximum utility and parallelism. Several works target automatic accelerator synthesis methods to explore their self-defined design spaces systematically [55, 84]. However, they miss the benefits and improvements obtained by exploring the CNN design space simultaneously. This motivates the need for a co-design framework that explores the CNN and accelerator architecture design spaces simultaneously.

2.3 Hardware-software Co-design

Due to the benefits of both NAS and automated accelerator design, hardware-software co-design methods explore an optimal CNN–accelerator pair. Many RL-based methods target co-design [4, 34]. However, these frameworks are only applicable to FPGA-based accelerators. Zhang et al. [85] propose a rigorous design space of FPGA-based accelerators. However, it does not consider various memory types (e.g., off-chip **dynamic random-access memory (DRAM)**, **high-bandwidth memory (HBM)**, and **resistive random-access memory (RRAM)**), which limits its applicability. Recent works factor this in with a memory bandwidth parameter [51, 85]. However, the memory configuration also affects the memory controller and area (for monolithic adoption), which previous works do not target. Finally, these works do not consider the batch size (and the corresponding tiling parameters) that affects both model accuracy and hardware performance. NAHAS [91], another RL-based exploration method, performs co-design based on the design space of ASIC-based accelerators. Nevertheless, the framework only focuses on a small CNN design space with limited CNN architectural hyperparameters. Moreover, it does not take the energy consumption of the accelerators and the cost of the main memory system into account when performing the evaluation. EDD [41] and DANCE [16] instead exploit differentiable search of the CNN model. Evolution-based and differentiable search algorithms only perform a local search around the parent designs, thus missing out on other potentially superior CNN–accelerator pairs in the design space [57]. Gibbon [62] leverages ES (drawbacks discussed in Section 2.1.4) on a design space of in-memory accelerators, although with an adaptive parameter priority pipeline. However, it is limited to in-memory accelerators, does not employ uncertainty in exploration, and does not tune the training recipe for the chosen CNN model. Another recent work, NAAS [44], leverages an evolution-based algorithm to explore the ASIC-based hardware space together with compiler mapping strategies. The above shortcomings motivate the development of a thorough and sophisticated co-design method based on comprehensive CNN and accelerator design spaces. Next, we present our proposed solution that tackles these challenges.

3 THE CODEBENCH FRAMEWORK

We now discuss the proposed co-design pipeline in detail.

3.1 CNNBench

We discuss different parts of the CNNBench pipeline (summarized in Figure 2) next.

3.1.1 Design Space and Model Cards. The CNNBench design space (represented by Figure 2(a)) is formed by *computational graphs* for a diverse set of CNN architectures. All models in CNNBench are combinations of one or more operations. These operations include not only traditional convolutions, but also many more, in light of recent state-of-the-art CNN architectures. CNNBench supports *2D convolution* operations of various kernel sizes (1×1 to 11×11) with different number of channels (4 to 8256), number of groups (4 or 8), padding (1 to 3), along with *ReLU*, *SiLU* [50], and other activation functions (further details in Section 4.1). We also support *depthwise separable convolutions* [52], *3D convolutions*, and *transposed convolutions* [35]. Each convolutional block in our design space is a combination of the convolution operation itself, followed by a *batch-normalization* step and the activation function [80].

Unlike the NASBench-101 dataset [80], our design space also includes *dropout* operations [60], *channel-shuffle* operation with different group sizes [89] along with max and average *pooling* with different kernel sizes, padding, and strides. In line with the EfficientNet family, we also include bilinear *upsampling* operations to support very deep networks that would otherwise suffer from

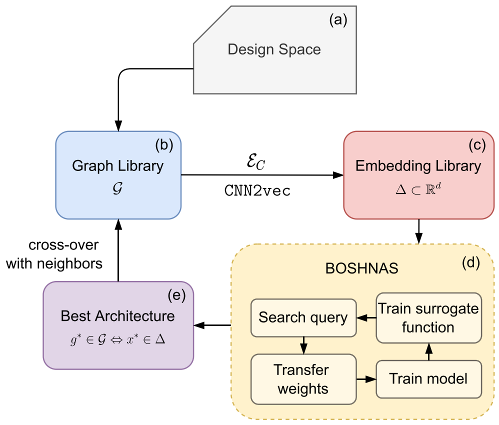

Fig. 2. Flowchart of different parts of the CNNBench pipeline.

vanishing spatial dimensions [65]. This expanded set of operations makes the design space flexible enough for more accelerator-friendly models with respect to efficiency and hardware capacity.

3.1.2 Block-level Computational Graphs. Using the operation blocks defined above, we create various possible CNN architectures in the form of computational graphs. These forward flows of operations in the network determine the computational graph. Figure 3(a) shows the computational graph of LeNet [38]. Using the set of permissible operation blocks, we create all possible computational graphs for the design space (represented by Figure 2(b); details in Section 4). Figure 3(b) shows the computational graph of another possible CNN in our design space. We create each computational graph in the form of *modules*, where each module is a set of interconnected blocks with one input and one output. For instance, in Figure 3(b), there are two modules, one for the convolutional operations (yellow) and the other for the MLP head (orange). We stack multiple such modules to create deep and complex CNN architectures in our design space.

These modules are serially connected to form the computational graphs in the design space. This restricts the creation of highly complex graphs. For our design space, we limit the size of convolutional modules to five operations (including *input* and *output* blocks) with a maximum of eight edges in the graph per module. The final head can have up to eight operations that are sequentially connected. Then, using the set of computational blocks, we create the design space of CNN architectures in this modular format.

3.1.3 Levels of Hierarchy. Creating graphs using the above approach may lead to an extremely large design space. To make the exploration more tractable, we propose a hierarchical search method that searches the design decisions in an increasingly granular fashion. One can consider each CNN architecture created in the design space to be composed of multiple stacks of modules. In this context, a stack is just a serial connection of modules where each module in the stack is inherently the same. For instance, if the number of modules in a stack is 10, i.e., $s = 10$, in a CNN with 31 modules, then the first 10 modules would be the same, then the next 10, and the next 10 after that, after which there would be a module for the MLP head. To go from one level of the hierarchy to the next, we consider a design space constituted by a finer-grained neighborhood of these models. We derive the neighborhood by pairwise crossover between the best-performing models and their neighbors in the current level of the hierarchy where the number of modules per stack is s (found using a graph-similarity method discussed later), in a space where the number of modules per stack is $s/2$ (or any integer divisor of s). We explain this crossover in detail next.

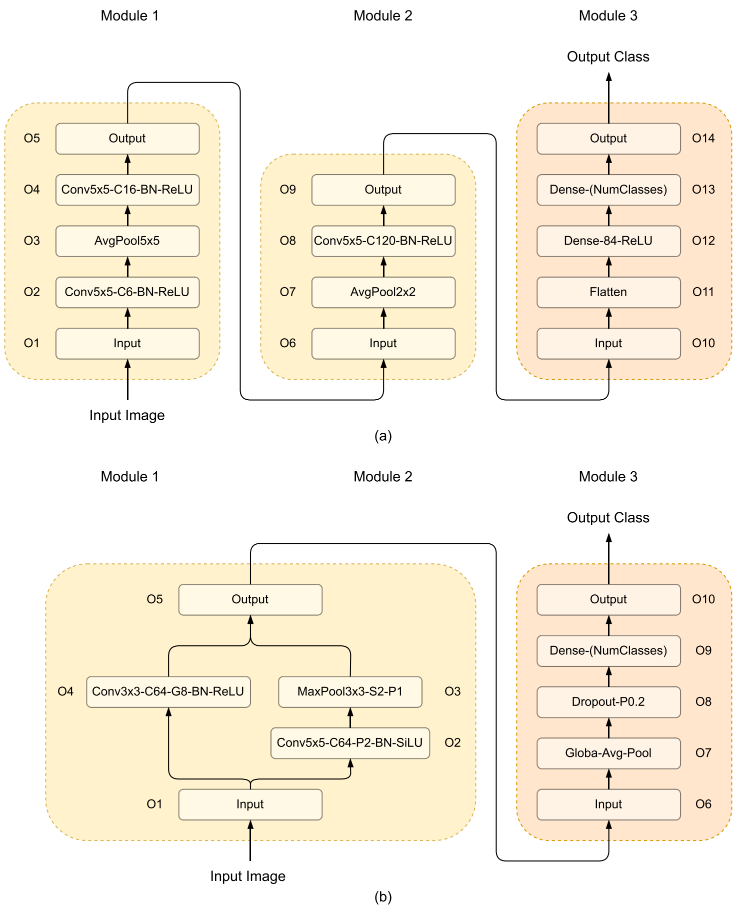

Fig. 3. Computational graph of (a) LeNet and (b) a complex CNN in the CNNBench design space. *Dense-(NumClasses)* is a feed-forward layer based on the number of output classes for the given dataset [21, 36]. Stride and padding default to 1 if not mentioned.

3.1.4 Crossover between CNN Models. We obtain new models in the subsequent level of the hierarchy by performing a crossover between the best models in the previous level (with the number of modules per stack = s) and their neighbors. Figure 4 presents a working example of a crossover between two neighbors. Each stack, respectively, has s modules that are exactly the same. Once two well-performing CNNs at this level of hierarchy (or in other words, with a stack size of s) are encountered, we explore architectures that are more granular *interpolants* between these models, i.e., with a smaller stack size, where modules can be the same only up to a smaller s. For every stack depth, we create a *local* space of operation blocks by considering a union of the set of operations used in stacks at the same depth (say, A and C, as in Figure 4). Then, new modules are generated by sampling from these local spaces (denoted by $A \cup C$) and stacked based on the new s/K modules per stack ($K \in \mathbb{N}, K \leq s$). As a concrete example, if stacks A and C had $s = 10$ modules within them (again, each module in the stack being the same), then a design space from their modules is formed ($A \cup C$). Modules are then sampled from this space and stacked with a shorter stack size, say $s = 5$, to generate the design space for the next level of the hierarchy. Finally, we add the modules for the MLP heads at the end based on a sample from the union of

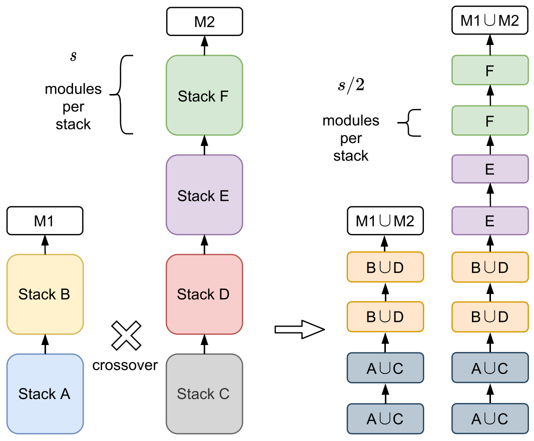

Fig. 4. Crossover between neighboring models. New models are generated by creating local design spaces at each stack depth (where the number of modules per stack is s) and creating combinations of operations at half the stack size ($s/2$) for more granularity. MLP heads are denoted by M1 and M2, respectively.

operations in the respective MLP heads of the neighbors. Expanding the design space in such a fashion retains the original hyperparameters that result in high performance while also exploring finer-grained internal representations learned by combinations of these hyperparameters at the same level.

3.1.5 Isomorphism Detection. Two computational graphs may be isomorphic. Hence, processing both would be redundant. We use recursive hashing to detect graph isomorphism as follows [80]. For every node in the computational graph, we concatenate the hash of its input, the hash of that node, and its output and then take the hash of the result. For nodes with multiple inputs or outputs, we sort the hashes of these inputs or outputs and concatenate them to get a representative hash of the input or output. We use SHA256 as our hashing function. Doing this for all nodes and then hashing the concatenated hashes gives us the resultant hash of a given computational graph. We have empirically verified that this algorithm does not cause *false positives* for the design space in consideration. Even if we had found a slight false-positive rate, it would only have added a small amount of redundancy to the search pipeline.

3.1.6 `CNN2vec` *Embeddings.* To run architecture search on our CNN design space, we generate a library of dense embeddings (see Figure 2(c)) for all the CNN architectures. We refer to them as `CNN2vec` embeddings, denoted by $\mathcal{E}_C$. We do this by first calculating the **Graph Edit Distance (GED)** for all model pairs in the design space [5]. Unlike other approaches like the Weisfeiler–Lehman kernel, GED bakes in *domain knowledge* in graph comparisons by using a weighted sum of node insertion, deletion, and substitution costs. For the GED computation, we first sort all possible operation blocks according to their computational complexity. Then, we weight the insertion and deletion cost for every block based on its index in this sorted list, and the substitution cost between two blocks based on the difference in the indices in this sorted list. GED also takes into account edge costs (set to a small value, $\epsilon_{edge} = 1 \times 10^{-9}$) when calculating distances between graphs.

Some previous works have also employed deep and compute-heavy neural models for conversion of a computational graph to an embedding [48, 77, 86]. These methods were tested on small design spaces and are not scalable to the vast spaces used in this work. For instance, D-VAE [86] encodes only 19,020 neural architectures with only 37.26% uniqueness. GIN-based models [48, 77] are too expensive to encode all models in the CNNBench design space efficiently. Hence, we use the fast and accurate GED-based distance metric when generating the `CNN2vec` embeddings.

Given N graphs in the CNN design space of computational graphs ($\mathcal{G}$), we compute the GED for all possible graph pairs. This gives us a dataset of $P = \binom{N}{2}$ distance pairs. To train the embedding, we minimize the mean-squared error as the loss function between the predicted Euclidean distance and the corresponding GED. For the design space in consideration, we generate embeddings of d dimensions for every level of the hierarchy. More concretely, to train the embedding $\mathcal{E}_C$, we minimize the loss

$$\mathcal{L}_{\mathcal{E}_C} = \sum_{1 \leq i \leq P, 1 \leq j \leq P, i \neq j} (\mathbf{d}(\mathcal{E}_C(g_i), \mathcal{E}_C(g_j)) - \text{GED}(g_i, g_j))^2,$$

where $\mathbf{d}(\cdot, \cdot)$ is the Euclidean distance and the GED is calculated for the corresponding computational graphs g_i, $g_j \in \mathcal{G}$. `CNN2vec` embeddings are superior to path encodings used in Reference [72], because they are dense and hence more amenable to search in surrogate models [14]. They also decouple the embedding process from the search flow to speed up training [72]. Further, since we derive these embeddings from graph distances, we can better interpret *interpolants* between two CNN architectures. Finally, since this method learns a tabular embedding by directly propagating the gradients of the above loss backward, one can train the embeddings speedily with large batch sizes and high parallelization with little memory overhead on a GPU. We use these embeddings in our search process.

3.1.7 Weight Transfer among Neighboring Models. Training each model in the design space is computationally expensive. Hence, we rely on weight sharing to initialize a query CNN model, thus setting the weights closer to the optimum to train new queries while minimizing exploration time (we also employ early stopping). This reduces the overall search time by 32% [67] relative to training from scratch. Previous works have implemented weight transfer between two CNN models [8, 24]. However, such works only support weight transfer between models with wider or deeper versions of the same computational blocks. However, CNNBench supports weight transfer between *any* two models in its design space due to the modularity inherent in the representation and implementation.

For every new query (q) that we need to train, we find k nearest neighbors of its corresponding computational graph in the design space (we use $k = 100$ in our experiments), found based on the Euclidean distance from the `CNN2vec` embeddings (that have a one-to-one correspondence with the GED of the respective computational graphs) of other CNNs. Here, the set of neighbors of q is denoted by N_q, and $|N_q| = 100 \ \forall \ q$.

Now that we have a set of neighbors, we need to determine which ones are more amenable to weight transfer. Naturally, we would like to transfer weights from the corresponding trained neighbor closest to the query, as such models intuitively have similar initial internal representations. We calculate this similarity using a new *biased overlap* metric that counts the number of modules from the input to the output that are in common with the current graph (i.e., have exactly the same set of operations and connections). We stop counting the overlaps when we encounter different modules, regardless of subsequent overlaps. This ranking could lead to more than one graph with the same *biased overlap* with the current graph. Since the internal representations learned would depend on the subsequent set of operations as well, we break ties based on the embedding distance of these graphs from the current one.

We now have a set of neighbors for every graph that are ranked based on both *biased overlap* and embedding distance. This helps increase the probability of finding a trained neighbor with high overlap. As a hard constraint, we only consider transferring weights if the *biased overlap fraction*

ALGORITHM 1: BOSHNAS

```
Result: best architecture
Initialize: overlap threshold ($\tau_{WT}$), convergence criterion, uncertainty sampling prob. ($\alpha_P$), diversity
  sampling prob. ($\beta_P$), surrogate model ($f$, $g$, and $h$) on initial corpus $\delta$, design space $g \in \mathcal{G} \Leftrightarrow x \in \Delta$;
while convergence criterion not met do
    wait till a worker is free;
    if prob. $\sim U(0,1) < 1 - \alpha_P - \beta_P$ then
        $\delta \leftarrow \delta \cup$ {new performance point $(x, o)$};
        fit(surrogate, $\delta$) using Equation (2);
        $x \leftarrow$ GOBI($f$, $h$) ;                          /* Optimization step */
        for $n$ in $N_x$ do
            if $n$ is trained & $O_f(x, n) \geq \tau$ then
                $W_x \leftarrow W_n$;
                send $x$ to worker;
                break;
    else
        if $1 - \alpha_P - \beta_P \leq$ prob. $< 1 - \beta_P$ then
            $x \leftarrow \mathbf{argmax}_x(k_1 \cdot \sigma + k_2 \cdot \hat{\xi})$ ;   /* Uncertainty sampling */
            send $x$ to worker;
        else
            send random $x$ to worker ;                   /* Diversity sampling */
```

($O_f(q,n) = \textit{biased overlap}/l_q$, where q is the query model, $n \in N_q$ is the neighbor in consideration, and l_q is the number of modules in q) between the queried model and its neighbor is above a threshold τ_{WT}. If the query model meets the constraint, then we transfer the weights of the shared part from the corresponding neighbor to the query and train it under this weight initialization. Otherwise, we may have to train the query model from scratch. We denote the weight transfer operation by $W_q \leftarrow W_n$.

3.1.8 BOSHNAS. The state-of-the-art NAS technique in the CNN design space, namely BANANAS [72], uses an ensemble NN to predict uncertainty in model performance. However, ensemble NNs are dramatically more compute heavy than a single NN predictor. Researchers have also leveraged NNs to convert an optimization problem into a **mixed-integer linear program (MILP)** to search for better-performing points in the design space [47]. However, solving the MILP problem is computationally expensive and is often the bottleneck in this solution. Instead, we leverage a novel framework, namely BOSHNAS (see Figure 2(c)), for searching over a space of CNN architectures for the first time. BOSHNAS runs **gradient-based optimization using backpropagation to the input (GOBI)** [68] on a *single* and *lightweight* NN model that predicts not only model performance but also the epistemic and aleatoric uncertainties. It leverages an *active learning* framework to optimize the **upper confidence bound (UCB)** estimate of CNN model performance in the embedding space. We use the estimates of aleatoric uncertainty to further optimize the training recipe for every model in the design space. We explain the BOSHNAS pipeline in detail next. Algorithm 1 shows its pseudo-code.

Uncertainty types: To overcome the challenges posed by an unexplored design space, it is important to consider uncertainty in model predictions to guide the search process. Predicting model performance deterministically is not enough to estimate the next most probably best-performing model. We leverage UCB exploration on the predicted performance of unexplored models. This can arise from not only the approximations in the surrogate modeling process but also parameter initializations and variations in model performance due to different training recipes, namely the *epistemic* and *aleatoric* uncertainties.

Surrogate model: BOSHNAS uses **Monte Carlo (MC)** dropout [27] and a **Natural Parameter Network (NPN)** to model the epistemic and aleatoric uncertainties, respectively. The NPN not only helps with a distinct prediction of aleatoric uncertainty that we can use for optimizing the training recipe once we are close to the optimal architecture but also serves as a superior model than Gaussian processes, Bayesian Neural Networks, and other **Fully Connected Neural Networks (FCNNs)** [71]. Consider the NPN network $f(x;\theta)$ with a CNN embedding x as an input and parameters θ. The output of such a network is the pair $(\mu, \sigma) \leftarrow f(x;\theta)$, where μ is the predicted mean performance and σ is the aleatoric uncertainty. To model the epistemic uncertainty, we use two deep surrogate models: (1) teacher (g) and (2) student (h) networks. The teacher network is a surrogate for the performance of a CNN, using its embedding x as an input. The teacher network is an FCNN with MC Dropout (parameters θ'). To compute the epistemic uncertainty, we generate n samples using $g(x, \theta')$. The standard deviation of the sample set is denoted by ξ. To leverage GOBI and avoid numerical gradients due to their poor performance, we use a student network (FCNN with parameters θ'') that directly predicts the output $\hat{\xi} \leftarrow h(x, \theta'')$, a surrogate of ξ.

Active learning and optimization: For a design space $\mathcal{G}$, we first form an embedding space Δ by transforming all graphs in $\mathcal{G}$ using the `CNN2vec` embedding. Assuming we have the three networks f, g, and h initialized on a randomly sampled set of pre-trained models (δ), we use the following UCB estimate,

$$\text{UCB} = \mu + k_1 \cdot \sigma + k_2 \cdot \hat{\xi} = (f(x,\theta)[0] + k_1 \cdot f(x;\theta)[1]) + k_2 \cdot h(x, \theta''), \tag{1}$$

where $x \in \Delta$, k_1, and k_2 are hyperparameters. To generate the next CNN to test, we run GOBI using the AdaHessian optimizer [79] that uses second-order updates to x (∇_x^2UCB) up to convergence. From this, we get a new query embedding, x'. We find the nearest embedding for a valid CNN architecture based on the Euclidean distance of all available CNN architectures in the design space Δ, giving the next closest model x. We then train this model (from scratch or after weight transfer from a nearby trained model with sufficient overlap) on the desired dataset to give the respective performance. Once we receive the new datapoint (x, o), we train the models using the loss functions on the updated corpus, δ',

$$\begin{aligned}
\mathcal{L}_{\text{NPN}}(f, x, o) &= \sum_{(x,o)\in\delta'} \frac{(\mu - o)^2}{2\sigma^2} + \frac{1}{2}\ln \sigma^2, \\
\mathcal{L}_{\text{Teacher}}(g, x, o) &= \sum_{(x,o)\in\delta'} (g(x,\theta') - o)^2, \\
\mathcal{L}_{\text{Student}}(h, x) &= \sum_{x, \forall(x,o)\in\delta'} (h(x,\theta'') - \xi)^2,
\end{aligned} \tag{2}$$

where $\mu, \sigma = f(x, \theta)$, and we obtain ξ by sampling $g(x, \theta')$. The first is the aleatoric loss to train the NPN model [71]; the other two are squared-error loss functions. We can run multiple random cold restarts of GOBI to get multiple queries for the next step in the search process.

To summarize, starting from an initial pre-trained set δ in the first level of the hierarchy $\mathcal{G}_1$, we run until convergence the following steps in a multi-worker compute cluster. To trade off between exploration and exploitation, we consider two probabilities: uncertainty-based exploration (α_P) and diversity-based exploration (β_P). With probability $1 - \alpha_P - \beta_P$, we run second-order GOBI using the surrogate model to maximize UCB in Equation (1). Adding the converged point (x, o) in δ, we minimize the loss values in Equation (2) (line 6 in Algorithm 1). We then generate a new query

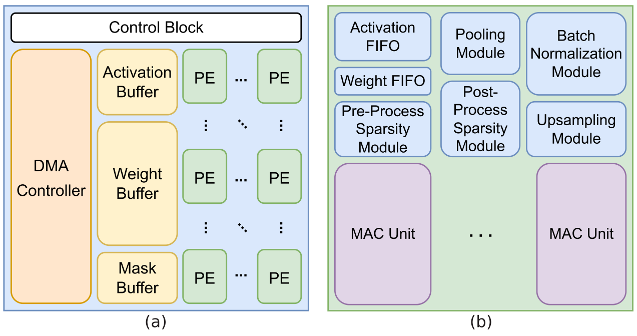

Fig. 5. Accelerator/PE: (a) Generic layout of an accelerator in AccelBench. (b) Layout of a PE in an accelerator in AccelBench.

point, transfer weights from neighboring models, and train it (lines 7–11). With α_P probability, we sample the search space using the combination of aleatoric and epistemic uncertainties, $k_1 \cdot \sigma + k_2 \cdot \hat{\xi}$, to find a point where the performance estimate is uncertain (line 16). To avoid getting stuck in a localized search subset, we also choose a random point with probability β_P (line 18). Once we converge in the first level, we continue with subsequent levels ($\mathcal{G}_2, \mathcal{G}_3, \ldots$) by forming subsequent design spaces for each level of the hierarchy, as explained in Sections 3.1.3 and 3.1.4. The time complexity of one iteration of the active learning loop is $O(n_\delta)$ where n_δ is the number of models in the corpus at that iteration.

3.2 AccelBench

Our accelerator benchmarking framework, AccelBench, is built upon a systolic 2D array architecture for generic ASIC-based accelerators. Figure 5(a) shows the generic layout of the accelerators incorporated in our design space. Taking inspiration from SPRING [83], a state-of-the-art accelerator, all accelerators in AccelBench have a control block to handle the CNN configuration sent from the CPU and to control the acceleration operations. The direct memory access controller communicates with the main memory system to load and store the input feature maps and filter weights from and to the on-chip buffers, respectively. We split the on-chip storage into three parts: *activation*, *weight*, and *mask* buffers. The activation buffer stores the input feature maps and output partial sums, while the weight buffer holds the filter weights [83]. We compress the data stored in both the activation and weight buffers in a zero-free format through a dedicated sparsity-aware binary-mask scheme to reduce the memory footprint. We design the mask buffer to store the binary mask vectors used in the binary-mask scheme for sparsity-aware acceleration (more details below). The 2D PE array executes the main computations of CNN acceleration and the MAC operations. AccelBench expands the design space by scaling various hyperparameters in a CNN accelerator, including the number of PEs, number and design of the MAC units, batch size used for running CNN inference, size of on-chip buffers, and size and configuration of the main memory system. We present the details of these scalable hyperparameters next.

3.2.1 Processing Elements. Figure 5(b) shows the layout of the PE and its built-in modules. It buffers input activations from the feature maps and weight data from the filters into the activation **first-in-first-out (FIFO)** and weight FIFO pipelines, respectively. To exploit sparsity in CNN weights, to improve efficiency, we employ the binary-mask scheme used in SPRING [83] to skip ineffectual MAC computations. This scheme uses two binary masks to indicate non-zero data in both input activations and filter weights. The *pre-process sparsity module* takes in data from the FIFOs and uses the binary masks to eliminate all ineffectual MAC operations, e.g., when either

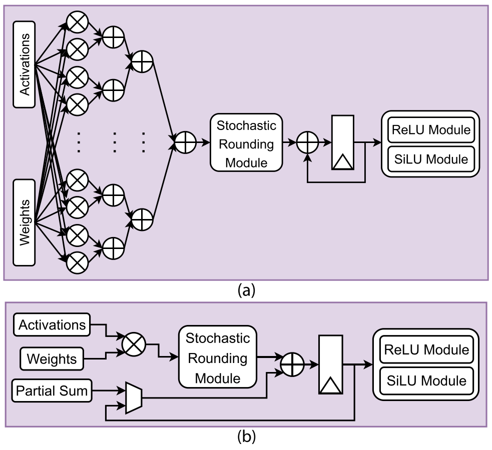

Fig. 6. Layouts of two different MAC units: (a) 16-multiplier and (b) 1-multiplier.

the input activation or the weight is zero. It then sends the zero-free MAC operations to the MAC units to complete the computations. The PE passes the outputs generated from the MAC units through the *post-process sparsity module* to eliminate the zeros and maintain a zero-free format to reduce the memory footprint. The *pooling module* supports three different pooling operations, i.e., max pooling, average pooling, and global average pooling. The *batch normalization module* executes batch normalization operations used in modern CNNs to reduce covariance shift [33]. Finally, the *upsampling module* processes the upsampling operations used to upscale the feature maps (as explained in Section 3.1).

3.2.2 MAC Units. The MAC units, which are inside each PE, execute the MAC operations. We include two kinds of MAC units in AccelBench. Figure 6(a) shows the design of the 16-multiplier MAC unit. Many custom accelerators use similar designs, including DianNao, DaDianNao, Cambricon-X, Cambricon-S, Cnvlutin, and SPRING. The MAC unit consists of 16 multipliers that can perform 16 pairs of multiplications between the input activations and weights in parallel. The results from the multipliers feed into a 16-input adder tree to perform the accumulation operation. The accumulated sum passes through a dedicated *stochastic rounding module* to reduce the number of bits of precision to decrease the memory footprint (details given later). The truncated result accumulates the previous partial sum. The module that executes the activation function, either ReLU or SiLU, takes this result as input. Figure 6(b) shows the design of the 1-multiplier MAC unit. Many custom accelerators (including Eyeriss v1, Eyeriss v2, and ShiDianNao) use a similar design to favor certain dataflows over others. The MAC unit contains only one multiplier to perform one multiplication between the input activation and the filter weight. This MAC design accumulates the result with previous partial sums from the on-chip buffer or the internal register. The *stochastic rounding module* again truncates this result and then sends it to either the ReLU or SiLU module.

All accelerators in AccelBench use the stochastic rounding algorithm [29] to quantize the data to a fixed-point representation. SPRING [83] first used this algorithm. Unlike the traditional deterministic rounding scheme that rounds a real number to its nearest discrete integer, the stochastic rounding algorithm rounds a real number x to $\lfloor x \rfloor$ or $\lfloor x \rfloor + \epsilon_R$ stochastically, as follows:

$$Round(x) = \begin{cases} \lfloor x \rfloor & \text{with probability } \frac{\lfloor x \rfloor + \epsilon_R - x}{\epsilon_R} \\ \lfloor x \rfloor + \epsilon_R & \text{with probability } \frac{x - \lfloor x \rfloor}{\epsilon_R} \end{cases}, \tag{3}$$

where ϵ_R denotes the smallest positive discrete integer supported in the fixed-point format and $\lfloor x \rfloor$ represents the largest integer multiple of ϵ_R less than or equal to x. The advantage of this approach over traditional rounding is that it does not lose information over multiple passes of the data instance. SPRING shows that using a fixed-point representation with four IL bits and 16 FL bits can represent a CNN with negligible accuracy loss, where IL (FL) denotes the number of bits in the integer (fraction) part. Hence, in our experiments, we set IL = 4 and FL = 16. We use the same configuration for all hardware processing modules in AccelBench.

3.2.3 Batch Size. To leverage parallel computation in accelerators, AccelBench supports multiple batch sizes during inference. More CNN operations can be computed in parallel with a larger batch size, resulting in higher throughput. However, processing with a large batch size requires sufficient hardware computation capacity, e.g., sufficient PEs, MAC units, on-chip storage, and memory bandwidth. An optimal batch size for an accelerator would depend on the features of the targeted CNN model. AccelBench thus provides flexibility to maneuver through various factors simultaneously, namely hardware capacity and CNN architecture, that affect final performance; parameters like batch size impact both the hardware and CNN performance.

3.2.4 On-chip Buffers. To improve on-chip data bandwidth and enable sophisticated buffer design, we partition the on-chip storage into three parts: activation, weight, and mask buffers. The optimal on-chip buffer size depends on the throughput capability of the hardware computational modules and CNN model size. The proposed optimization framework finds a balance between area and energy while searching for an optimal CNN–accelerator pair.

3.2.5 Main Memory. CNN computations involve a large number of parameters, such as the input feature maps and filter weights. A sufficiently large main memory is crucial for storing all the CNN weights and input images (for a given batch) in the CNN accelerator. Moreover, a high-bandwidth interface is indispensable to keep the accelerator running at a high utilization rate. AccelBench supports three different main memory systems, namely, DRAM, 3D HBM, and monolithic 3D RRAM. In addition, we provide different memory configurations for each memory type with respect to the numbers of banks, ranks, and channels.

To increase the off-chip bandwidth (beyond conventional DRAM) for CNN computations, **through-silicon via– (TSV)** based 3D memory interfaces, such as HBM, have been used in high-end GPUs and specialized CNN accelerators [81]. Unlike an HBM that uses TSVs, the monolithic 3D memory interface employs monolithic 3D integration to fabricate the chip tier-by-tier on only one substrate wafer. The tiers are connected through **monolithic inter-tier vias (MIVs)**, whose diameter is one to two orders of magnitude smaller than that of TSVs, enabling a much higher MIV density and thus a higher bandwidth [82]. SPRING leverages this monolithic 3D memory system based on non-volatile RRAM to deliver high memory bandwidth and energy efficiency [83].

3.2.6 CNN Mapping. Figure 7 shows a convolutional layer of a CNN. To map the MAC operations of a convolutional layer onto an accelerator, data need to be partitioned into smaller chunks for placement onto on-chip buffers. This is called data *tiling* [63]. After performing data tiling, one can explore the computational parallelism in a convolutional layer by unrolling the data chunks in different dimensions. This is called loop *unrolling* [63]. There are a total of seven dimensions of parallel computations we can explore in both input feature maps and filter weights. $N_{ib}, N_{if}, N_{ix}, N_{iy}, N_{of}, N_{kx}$, and N_{ky} denote the input batch size, number of input feature map channels, width and height of the input feature maps, number of output feature map channels, and width and height of the filter weights, respectively [84]. We specify the number of parallel computations among these dimensions as $P_{ib}, P_{if}, P_{ix}, P_{iy}, P_{of}, P_{kx}$, and P_{ky}, respectively. We use these parameters to determine the number of PEs and MAC units of the accelerator. The number of

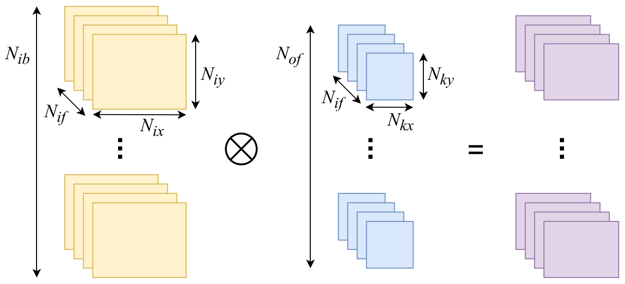

Fig. 7. The convolutional layer of a CNN.

PEs: $\#PEs = P_{ib} \times P_{ix} \times P_{iy}$. The number of MAC units inside each PE: $\#MAC\ units = P_{of} \times P_{kx} \times P_{ky}$. We set P_{if} to either 16 or 1, depending on the type of MAC unit used in the accelerator.

Choosing the dataflow is also an important design choice while building a CNN accelerator. We choose the dataflow (namely OS) based on the state-of-the-art accelerator, SPRING [83]. This accelerator selects the OS dataflow based on profiled performance on popular CNN architectures. Thus, OS is the dataflow of choice for all architectures in our design space. Some recent works have shown the advantages of reconfiguring the dataflow on a layerwise basis [69]. However, this would add significant interconnect overhead to the architectures in the AccelBench design space. Altering the dataflows also requires redesigning the compiler mapping strategies, including the loop unrolling order and data tiling size, and the connectivity between the MAC units and PEs [63]. In addition, adding dataflow support would significantly expand the accelerator design space, leading to longer search times. The fact that hierarchical search is impossible in accelerator designs exacerbates this further. Hence, we leave dataflow exploration to future work.

3.2.7 Accelerator Embeddings. To run BOSHCODE with the AccelBench framework, we represent each accelerator with a 13-dimensional vector. The elements of this vector represent the hyperparameters in our design space. We present more details in Section 4.2.

3.3 Co-design Pipeline

We leverage both CNNBench and AccelBench in the co-design process. We describe various parts of the co-design pipeline next.

3.3.1 BOSHCODE. BOSHNAS basically learns a function (comprising f, g, and h; cf. Section 3.1.8) that maps the CNN embedding to model performance. It then runs GOBI to get the next CNN architecture that maximizes performance. BOSHCODE extends this idea to CNN–accelerator pairs. It uses the same three functions in BOSHNAS (namely f, g, and h). However, we modify these functions to incorporate design spaces of both CNNs and accelerators and implement co-design. Again, we model the epistemic uncertainty by the teacher networks (g, with MC dropout) and a student network h and the aleatoric uncertainty by an NPN (f). The performance measure for optimization is a convex combination of latency, area, dynamic energy, leakage energy, and model accuracy. Mathematically,

$$\begin{aligned}\text{Performance} = \alpha \times (1 - \text{Latency}) + \beta \times (1 - \text{Area}) + \gamma \times (1 - \text{Dynamic Energy}) \\ + \delta \times (1 - \text{Leakage Energy}) + \epsilon \times \text{Accuracy},\end{aligned} \tag{4}$$

where $\alpha + \beta + \gamma + \delta + \epsilon = 1$ are hyperparameters, and we normalize the values of the individual performance measures with their maximum values (hence, reside in the $[0, 1]$ interval). Thus, for edge applications where the power envelope of devices is highly restricted, users can set

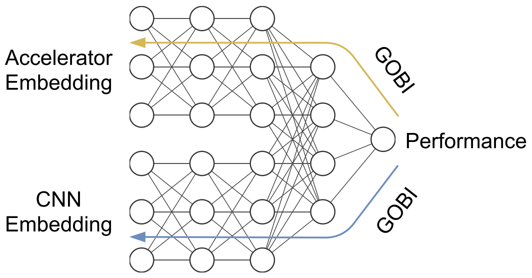

Fig. 8. Schematic of the BOSHCODE teacher function. Dropout layers have been omitted for simplicity.

the hyperparameters γ and δ high. However, for server-side deployments, where accuracy is of utmost importance, one can set ϵ high.

3.3.2 Learning the Surrogate Function. Using the dataset derived from mapping CNN–accelerator pairs to their respective performance values, BOSHCODE learns a surrogate function that predicts performance (along with the uncertainty) for all pairs in the design space. It leverages the active learning pipeline in BOSHNAS to query CNNs and accelerator architectures in their respective design space to get an optimal pair that maximizes the given performance function. As noted above, BOSHCODE does not use separate models for the CNN and accelerator but a hybrid one instead. This aids in learning the performance of every CNN–accelerator pair, which is a function of the independent CNN and accelerator design decisions and their interdependence. In other words, we execute GOBI on representations learned on information from both the CNN and accelerator parameters and also representations learned specifically for either the CNN or the accelerator.

Figure 8 shows a simplified schematic of the teacher network in BOSHCODE. It maps the CNN–accelerator embeddings to the corresponding performance measures, which are a combination of the 16-dimensional `CNN2vec` embeddings and the 13-dimensional accelerator embeddings. As explained above, the network learns separate representations of performance for the CNN and accelerator and then combines them to give a final performance prediction. The student network learns the epistemic uncertainty from this teacher network through sampling (cf. Section 3.1.8). Then, we run GOBI on the combined and separate representations (of the student network) to find the optimal CNN–accelerator pair that maximizes the UCB estimate of the performance. Here, the gradients are backpropagated to the input (i.e., the CNN–accelerator pair) to obtain the next pair to query in the active learning loop. The algorithm iteratively evaluates CNN–accelerator pairs (using the CNNBench and AccelBench frameworks) to obtain the best-performing pair. Moreover, as explained in Section 3.1.8, we implement this in a hierarchical fashion to efficiently search the massive design space of 9.3×10^{820} CNN–accelerator pairs (details of design spaces given in Section 4). More concretely, we gradually drop the stack size s from 10 to 1 over multiple iterations as the search process goes through numerous levels of the hierarchy. This is a crucial step in making the search feasible.

Any performance predictor has some inaccuracies. Uncertainty modeling and subsequent search mitigate the impact of such errors. Hence, while running the BOSHCODE pipeline, we leverage uncertainty-based sampling to achieve the optimal solution. When BOSHCODE reaches the optimal CNN–accelerator pair, it leverages aleatoric uncertainty in prediction to query the same pair multiple times. This aids in validating the optimality of the converged pair. Previous works do not test their surrogate model accuracy around the converged optimal point, thus leading to sub-optimal solutions. BOSHCODE also leverages diversity sampling to reduce modeling error on unexplored points in the design space.

3.3.3 Inverse Design. To add constraints to the CNN and accelerator design spaces, we limit the respective spaces to certain vectors. In other words, BOSHCODE only subsequently tests CNN–accelerator pairs within a pre-defined tabular design space. For instance, if the user wants to confine the accelerators within a certain area constraint, then we restrict GOBI to select the nearest vector (from the reached local optimum in the continuous space) in the tabular accelerator search space that satisfies the constraint.

4 EXPERIMENTAL SETUP

In this section, we discuss the setup and assumptions behind various experiments we performed in this work.

4.1 CNN Design Space

As described in Section 3.1, we use the operation blocks of various functions, convolution, pooling, MLP layers, activation functions, and so on, to define the design space. This results in 618 operation blocks in all. We describe these operations next.

- Channel shuffle in groups of {1, 2, 4, 8} [89].
- Dropout with probability of $\{0.1, 0.11, 0.2, 0.3, \ldots, 0.9\}$ [60].
- Upsampling to size of {240, 260, 300, 380, 465, 528, 600, 800} [65].
- Maxpool and Avgpool in kernel sizes of $\{3 \times 3,\ 5 \times 5\}$ with a padding of either 0 or 1 and stride of 1 or 2.
- Convolution in kernel sizes of $\{1 \times 1,\ 3 \times 3,\ 5 \times 5,\ 7 \times 7,\ 11 \times 11\}$ with output channels in $\{4, \ldots, 8256\}$ (98 values), with groups in {4, 8, 16} or depthwise layers where the groups are equal to the number of input channels. The padding and strides supported are from {0, 1, 2, 3} and {1, 2, 4}, respectively. For all convolutional layers, we support both ReLU and SiLU activations [50].
- Flatten and global-average-pool for flattening the output from the convolutional layers. Global-average-pool computes an average across the spatial dimensions to output a vector of length equal to the number of output channels from the convolutional layers [43].
- MLP layers with the number of hidden neurons in {84, 120, 1024, 4096} and, finally, the number of classes.

To limit the size of the design space, we do not consider all combinations but only those prevalent in popular CNN architectures. We limit the CNN computational graphs to a depth of 90 modules (excluding the final head module), each module with a maximum of five vertices (including the *input* and *output* operations) and limited to eight edges in modules with convolutional operations. For the final head module, we limit the number of vertices to eight (it only has linear feed-forward connections of fully connected and dropout layers). This leads to a total design space of size 4.239×10^{812}. This is much larger than any other NAS design space studied before [9, 57, 80]. We form the first level of the hierarchy by creating graphs with a stack size of 10, resulting in 2.089×10^{85} CNN architectures in the first level (details in Section 3.1.3). This not only eases the search for the best-performing CNN but also reduces the disparity in CNN and accelerator search space sizes that might lead to challenges in optimization.

The `CNN2vec` embedding has dimension set to $d = 16$ after running grid search. To do this, we minimize the distance prediction error while also keeping d small using knee-point detection. We obtain the hyperparameter values for BOSHNAS through grid search. We use overlap threshold $\tau_{WT} = 80\%$ and constants $k_1 = k_2 = 0.5$ (see Section 3.1) in our experiments. We set the uncertainty and diversity sampling probabilities to $\alpha_P = 0.1$, $\beta_P = 0.1$. The algorithm reaches the convergence criterion when the change in performance is within 10^{-4} after five iterations.

Table 2. Hyperparameter Values Used in the Proposed Accelerator Design Space

Hyperparameter	**Permissible values**
P_{ib}	1, 2, 4
P_{if}	1, 16
P_{ix}	1–8
P_{iy}	1–8
P_{of}	1, 2, 4, 8
P_{kx} and P_{ky}	1, 3, 5, 7
batch size	1, 64, 128, 256, 512
activation buffer size	1–24 MB, in multiples of 2
weight buffer size	1–24 MB, in multiples of 2
mask buffer size	1 MB, 2 MB, 3 MB, 4 MB
main memory type	1: Monolithic 3D RRAM 2: Off-chip DRAM 3: HBM
main memory configuration	RRAM: 1: [16, 2, 2], 2: [8, 2, 4], 3: [4, 2, 8], 4: [2, 2, 16], 5: [32, 2, 1], 6: [1, 2, 32] DRAM: 1: [16, 2, 2], 2: [8, 2, 4], 3: [32, 2, 1], 4: [16, 4, 1] HBM: 1: [32, 1, 4]

We set a fixed training time of 200 epochs for every CNN on the CIFAR-10 dataset [36]. Early stopping can result in lower training time for certain CNNs. We use a batch size of 128 for training. We automatically tune the training recipe for every CNN to tune the hyperparameter values. We leverage the Asynchronous Successive Halving scheduler [40], implemented using random search over different hyperparameter values, in the training recipe. CNNBench supports various optimizers including Adam, AdamW [46], and so on, along with various learning-rate schedulers including cosine annealing, exponential, and so on. We sample learning rates in a log-uniform manner in the $[1 \times 10^{-5}, 1 \times 10^{-2}]$ interval. We sample other optimizer parameters as follows: $\beta_1 \sim \mathcal{U}(0.8, 0.95)$, $\beta_2 \sim \mathcal{U}(0.9, 0.999)$, and weight decay $\ln(\lambda) \sim \mathcal{U}(\ln(1 \times 10^{-5}), \ln(1 \times 10^{-3}))$. Here, $\mathcal{U}$ refers to the uniform distribution.

4.2 Accelerator Design Space

Figure 1(a) shows the workflow of AccelBench. We implement all the supported modules in AccelBench at the **register-transfer level (RTL)** with System Verilog. We synthesize the RTL modules with Synopsys Design Compiler [3] using a 14-nm FinFET cell library [28] to estimate delay, power consumption, and area. We model on-chip buffers with FinCACTI [54] and the main memory systems with NVMain [22, 49]. Finally, we plug in the estimated delay, power consumption, and area of the RTL modules, on-chip buffers, and main memory systems into a custom cycle-accurate accelerator simulator implemented in Python. AccelBench's accelerator simulator takes the Python object of the CNN model generated using CNNBench as its input and estimates the computation latency, energy consumption, and area of the accelerator in consideration.

As mentioned in Section 3.2, we encode each accelerator in AccelBench with a 13-dimensional vector, which contains the hyperparameter values used to expand the accelerator design space. We show the ranges of all hyperparameter values in Table 2. Note that we set $P_{kx} = P_{ky}$ to match the conventional square matrix structure of the filter weights. The three entries in every list for main memory configurations are the number of banks, ranks, and channels, respectively. Using these respective ranges, we get a design space with 2.28×10^8 accelerators, much larger than in any previous work [4, 34, 44, 91].

4.3 Co-design and Baselines

For running BOSHCODE, we use the following parameter values to obtain the performance measure: $\alpha = 0.2$, $\beta = 0.1$, $\gamma = 0.2$, $\delta = 0.2$, and $\epsilon = 0.3$ (see Section 3.3). Other hyperparameters are the same as described in Section 4.1 All models are trained on NVIDIA A100 GPUs and 2.6-GHz AMD EPYC Rome processors. The entire process of training BOSHCODE to get the best CNN–accelerator pair took around 600 GPU-days.

As described in Table 1, we incorporate various accelerators into our AccelBench framework. For a fair comparison of various performance measures, we implement an analogous accelerator in the AccelBench design space by transferring the design decisions into a 13-dimensional vector. However, direct comparison with respective results is challenging, since each work uses a separate baseline for comparison and the reported results are often normalized. Further, individual works implement accelerators with different hardware modules, CNN mapping strategies, technology nodes, and clock frequencies. Thus, AccelBench serves as a common benchmarking platform that compares diverse accelerator architectures along fairly determined parameter values and performance measures.

We also compare our work with various co-design baselines. Co-Exploration [34] explores the hardware and software spaces simultaneously, with different objectives: optimizing either the hardware design decisions (OptHW) or the software CNN architecture (OptSW). It works with up to three Xilinx FPGAs (XC7Z015) and searches for a pipelining strategy. BoBW [4] implements RL-based co-design on the NASBench-101 dataset [80] for CNNs with a library of FPGAs. It uses CHaiDNN, a library for acceleration of CNNs on FPGAs [1]. The CHaiDNN FPGA accelerator has various configurable parameters [4]. NASAIC [78] and NAAS [44] leverage RL and ES, respectively, on a space of accelerators based on the NVDLA [2], ShiDianNao [23], and so on, dataflow templates. We also add a state-of-the-art differentiable search based baseline, i.e., Auto-NBA [26], which has a reasonably large design space of size 5.3×10^{63}. It employs the Accelergy platform, an energy estimator tool for accelerators [75]. Its design space is based on recent FPGA-based accelerators [56, 88] and adopts a chunkwise pipelined architecture.

5 RESULTS

In this section, we validate our claims by comparing the CODEBench framework with the aforementioned baselines.

5.1 BOSHNAS on a CNN Design Space

Figure 9(a) shows the performance of BOSHNAS with respect to the state-of-the-art NAS technique, BANANAS [72], on the NASBench-101 dataset [80]. The figure plots the final loss on the test set for CNNs found after a given number of queries from either NAS technique. As we can see, BOSHNAS results in a lower average test loss than BANANAS after 50 queries on the NASBench-101 design space. Figure 9(b) shows an ablation analysis of BOSHNAS on the NASBench-101 dataset, with the BOSHNAS model being used once without second-order gradients and once without modeling heteroscedasticity, i.e., the NPN model f. These plots justify the need for heteroscedastic modeling and second-order gradients in the space of CNN models. The heteroscedastic model forces the optimization of the training recipe when the framework approaches optimal architectural design decisions. Second-order gradients, however, help the search avoid local optima and saddle points and also aid faster convergence.

5.2 Co-design vs. One-sided Approaches

Figure 10 highlights the benefits of hardware-software co-design over one-sided approaches, namely automatic accelerator synthesis and hardware-aware NAS, as described in Section 2. We

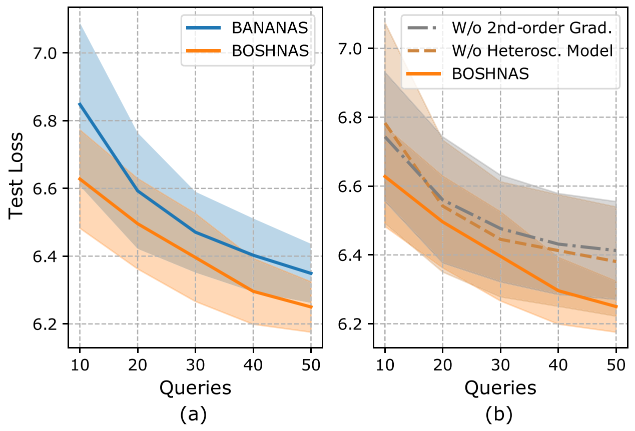

Fig. 9. Performance of BOSHNAS and BANANAS: (a) NASBench-101 dataset and (b) ablation analysis of BOSHNAS. Fifty trials are run for all techniques. Plotted with 90% confidence intervals.

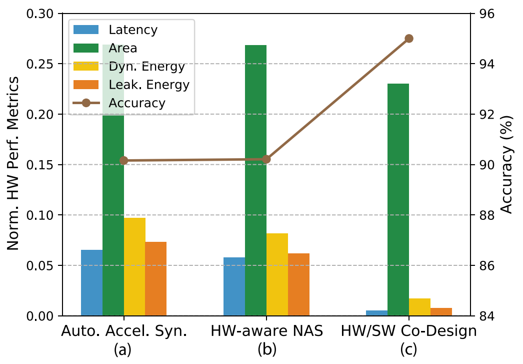

Fig. 10. Performance measures when three approaches are applied to the design space: (a) fixing CNN to MobileNet-V2 and searching the accelerator space, (b) fixing the accelerator to SPRING [83] and performing NAS, and (c) co-design in the space of CNN–accelerator pairs.

formulate the CNN and accelerator design spaces as per the hyperparameter ranges presented in Sections 4.1 and 4.2. Then, we search for a CNN–accelerator pair that optimizes the selected performance measure, which is a combination of latency, area, dynamic and leakage energies, and accuracy (see Equation (4)) for every CNN–accelerator pair.

Figure 10(a) shows the best normalized performance measures achieved after searching the space of accelerator design decisions using BOSHCODE. Here, we force gradients to the CNN embedding to zero to find the next accelerator to simulate in the search process. Figure 10(b) shows the results when we explore the CNN design space while fixing the accelerator instead. In this case, we force the gradients to the accelerator embedding to zero in the two-input BOSHCODE network. Fixing the accelerator and searching the CNN design space does not result in noticeable gains in accuracy but results in slight improvements in latency and energy values. This is due to the search process landing on another equally performing CNN but one that is smaller in terms of the number of parameters (reduced from 3.4 to 2.9 M). This also improves hardware resource utilization. Finally, Figure 10(c) shows the best performance achieved from co-design in the CNN and accelerator spaces using our proposed BOSHCODE approach. This enables higher flexibility in search, resulting in much higher accuracy and significant improvements in the hardware

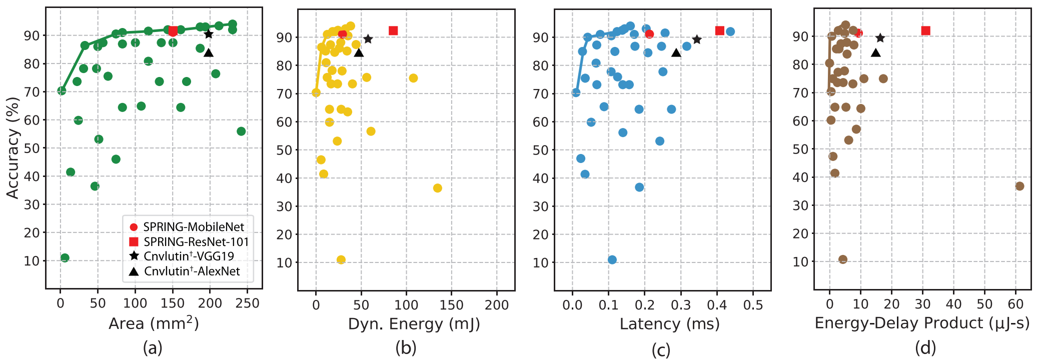

Fig. 11. Pareto frontier between accuracy and (a) chip area, (b) dynamic energy consumption, (c) latency, and (d) energy-delay product (calculated using the sum of dynamic and leakage energies). Reduced number of points (compared to 245 trained) have been plotted for clarity. Points far from the Pareto frontier have been neglected in the plot. †Cnvlutin-like [6] accelerator (used in conjunction with VGG19 and AlexNet) implemented in AccelBench for fair comparisons (see Section 4.3).

performance measures. Co-design achieves 4.3% higher model accuracy on the CIFAR-10 dataset, while enabling 85.7% lower latency (averaged per batch of images), 79.4% lower total energy consumption (including both dynamic and leakage energies), and 14.8% lower chip area, when compared to automatic accelerator synthesis. However, relative to hardware-aware NAS, co-design achieves 4.2% higher model accuracy at 84.6% lower latency and 14.8% lower chip area. Figure 10 uses the following values for normalization: 9 ms for latency, 774 mm^2 for chip area, 735 mJ for dynamic energy, and 280 mJ for leakage energy.

5.3 Optimal CNN–Accelerator Pair

Figure 11 shows the Pareto frontiers of CNN–accelerator pairs in our design space. We have also included other state-of-the-art pairs proposed in the literature, including MobileNet [52] and ResNet-101 [30] on SPRING [83] and VGG-19 [58] and AlexNet [37] on Cnvlutin [6]. One can observe these existing CNN–accelerator pairs to be far from the Pareto frontier. In other words, BOSHCODE finds pairs that outperform previously proposed CNN architectures and accelerator designs in user-defined performance measures.

For the performance measure defined by the convex combination of different parameter values (namely, latency, area, dynamic energy, leakage energy, and model accuracy, as calculated using Equation (4)), we get an optimal CNN–accelerator pair by running BOSHCODE on the entire design space. The optimal CNN is close to ResNet-50 (we call it ResNet*), and the optimal accelerator is similar to SPRING (we call it S*). Table 3 compares this optimal pair (denoted by S*-ResNet*) with the state-of-the-art baseline in various performance measures (we choose MobileNet with SPRING, as it gives the highest performance measure compared to other off-the-shelf CNNs, and denote the pair by S-MobileNet). BOSHCODE achieves 59.1% lower latency, 60.8% lower energy, and 1.4% higher accuracy compared to the state-of-the-art CNN–accelerator pair at the cost of 17.1% increase in area. A different convex combination of hyperparameter values ($\alpha, \beta, \gamma, \delta$, and ϵ) would result in BOSHCODE converging to a different CNN–accelerator pair. In our experiments, a low value of $\beta = 0.1$, which corresponds to the weight contributed to the chip area, results in a slightly higher area relative to the state-of-the-art pair. When comparing the two pairs on the ImageNet dataset [21], we see a 43.8% lower latency, 11.2% lower energy consumption, and 3.7% higher model accuracy. Due to the transferability of the optimal model from our search space to the

Table 3. Comparison of Optimal CNN–Accelerator Pair with a State of the Art

Performance measure	CIFAR-10		ImageNet	
	S-MobileNet	S*-ResNet*	S-MobileNet	S*-ResNet*
Latency (ms)	0.22	**0.09**	0.57	**0.32**
Area (mm^2)	**152**	178	**152**	178
Dynamic Energy (mJ)	28.4	**14.7**	41.2	**37.5**
Leakage Energy (mJ)	19.8	**4.2**	6.3	**4.7**
Accuracy (%)	93.4	**94.8**	72.1	**75.8**

Top1 accuracy is reported for the ImageNet dataset.
Bold values correspond to the lowest latency, area, energy, or the highest accuracy.

Table 4. Comparison of Baseline and Proposed Co-design Techniques

Framework	Platform	Search space	Accuracy (%)	Area (mm^2)	FPS (s^{-1})	EDP (μJ-s)
Baselines						
Co-Exploration (OptHW) [34]	FPGA	5.9×10^3	80.2	–	130	1612.9
Co-Exploration (OptSW) [34]	FPGA	5.9×10^3	85.2	–	130	1612.9
BoBW [4]	FPGA	4×10^9	94.2	**102**	8.1	–
NASAIC [78]	DLA	1.1×10^6	93.2	525	1749	571.8
NAAS [44]	DLA	1×10^{24}	93.2	525	6562	365.7
Auto-NBA [26]	Accelergy [75]	5.3×10^{63}	93.3	710	320	18.7
Ablation Analysis						
Hardware-Aware NAS (Ours)	AccelBench	4.2×10^{812}	91.8	245	1709	45.8
CODEBench (Ours; DRAM only)	AccelBench	1.6×10^{820}	93.9	184	8620	14.8
CODEBench (Ours)	AccelBench	9.3×10^{820}	**94.8**	178	**11,111**	**1.7**

Energy values reported are for each input frame.
Bold values correspond to the lowest area, EDP, or the highest accuracy or FPS.

ImageNet dataset, our optimal pair also outperforms other state-of-the-art approaches like differentiable search that are directly trained on the ImageNet dataset. For context, ProxylessNAS [10] achieves 74.6% Top1 accuracy with a latency of 78 ms on a Google Pixel 1 smartphone. The accuracy is 1.2% lower than that of S*-ResNet* and the latency is 243.75× higher than that of S*-ResNet*.

5.4 Co-design with an Expanded Design Space

Table 4 presents a detailed comparison of results for the CODEBench framework and various co-design baselines. Here, Co-Exploration [34] runs co-design in a design space of three Xilinx FPGAs (XC7Z015; fabricated on 28-nm process and normalized to 14 nm), chip area for which is unknown. BoBW [4] does not model energy consumption of the hardware architectures in its design space. For NASAIC [78] and NAAS [44], we report the die area for the Xavier system-on-chip that has two instances of NVDLA [2], fabricated in a 12-nm process [53] and normalized to 14 nm [61]. For NASAIC and NAAS, we assume a 700-MHz clock rate assumed for calculation of **frame-rate per second (FPS)** from the reported latency in cycles [44]. For Auto-NBA, we use Auto-NBA-Mixed [26] to report the value of FPS. Moreover, the same baseline reported the EDP in J-cycles and we assumed a 700-MHz clock rate for conversion to μJ-s.

Our framework not only achieves the highest model accuracy on the CIFAR-10 dataset but also improves upon other hardware performance measures including FPS or throughput and EDP. CODEBench achieves 1.5% higher accuracy and 34.7× higher throughput while having 11.0× lower EDP and requiring 4.0× lower chip area, when compared to a state-of-the-art co-design framework, i.e., Auto-NBA. These substantial gains compared to traditional co-design frameworks became possible due to a search over a massive design space of CNN–accelerator pairs (9.3×10^{820}), compared to only 5.3×10^{63} in Auto-NBA. We can attribute high reductions in energy consumption to the use of sparsity-aware computation, monolithic 3D RRAM interface instead of an off-chip DRAM [82], and so on. For fair comparisons, we have normalized the results for different technology nodes [61] and added a CODEBench optimal pair with the constraint that RRAM is unavailable. The co-design case outperforms hardware-aware NAS with a lower chip area and a higher throughput. We attribute this to a smaller CNN model that is a better fit for the searched accelerator, resulting in better utilization of resources and parallelism.

5.5 Discussion

For running the co-design pipeline, we trained all CNN models in our design space on the CIFAR-10 dataset. As seen from Table 3, and also noted above, performance of models on CIFAR-10 directly translates to that on ImageNet as well [92]. Running the BOSHCODE framework on the ImageNet dataset would incur high compute costs. Hence, this trend of transferable performance is of utmost importance to scalable search [80]. Due to the incorporation of highly diverse CNNs and accelerator architectures, future researchers can directly use these surrogate models to minimize time and costs and quickly search specialized subset spaces of interest. Additionally, we could further expand both the CNN and the accelerator design spaces with more operations in the former and more hardware modules and memory types in the latter. In this context, we could also employ the co-design pipeline for in-memory accelerators and neuromorphic architectures [62, 66]. Due to the proposed scalable method, one can search even larger CNN design spaces with a more aggressive hierarchical search, i.e., increasing granularity in fewer steps. One could also exploit dynamic CNN model inference [76] and on-the-fly quantization during inference [15] to further reduce latency and energy consumption. We leave these extensions to future work.

6 CONCLUSION

In this work, we presented CODEBench, a unified benchmarking framework for simulating and modeling CNN–accelerator pairs and their performance measures (model accuracy, latency, area, dynamic energy, and leakage energy). We developed a novel CNN benchmarking framework, CNNBench, that leverages the proposed `CNN2vec` embedding scheme and BOSHNAS (that uses this embedding scheme, Bayesian modeling, and second-order optimization) to efficiently search for better-performing CNN models within a given design space. Our proposed CNN design space is richer compared to that in any previous work. To optimize hardware performance at the same time, we also proposed a new accelerator benchmarking pipeline, AccelBench. It targets not only a vast design space of ASIC-based accelerator architectures but also simulates various performance measures of these hardware designs. We then use a novel co-design approach, namely BOSHCODE, that searches for the best CNN–accelerator pair. Our proposed approach results in a CNN–accelerator pair with 1.4% higher CNN accuracy compared to a state-of-the-art pair, i.e., SPRING-MobileNet, while enabling 59.1% lower latency and 60.8% lower energy values. CODEBench outperforms the state-of-the-art co-design framework, i.e., Auto-NBA [26], resulting in 1.5% higher CNN accuracy and 34.7× higher throughput, while enabling 11.0× lower EDP and 4.0× lower chip area.

ACKNOWLEDGMENTS

We performed the simulations presented in this article on computational resources managed and supported by Princeton Research Computing at Princeton University.

REFERENCES

[1] 2022. CHaiDNNv2—HLS based DNN Accelerator Library for Xilinx Ultrascale + MPSoCs. Retrieved from https://github.com/Xilinx/CHaiDNN.
[2] 2022. NVIDIA Deep Learning Accelerator. Retrieved from http://nvdla.org.

[3] 2022. Synopsys Design Compiler. Retrieved from https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html.

[4] Mohamed S. Abdelfattah, Łukasz Dudziak, Thomas Chau, Royson Lee, Hyeji Kim, and Nicholas D. Lane. 2020. Best of both worlds: AutoML codesign of a CNN and its hardware accelerator. In *Proceedings of the ACM/EDAC/IEEE Design Automation Conference*. Article 192, 6 pages.

[5] Zeina Abu-Aisheh, Romain Raveaux, Jean-Yves Ramel, and Patrick Martineau. 2015. An exact graph edit distance algorithm for solving pattern recognition problems. In *Proceedings of the International Conference on Pattern Recognition Applications and Methods*, Vol. 1. 271–278.

[6] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *Proceedings of the ACM/IEEE International Symposium of Computer Architecture*, Vol. 44. 1–13.

[7] Hadjer Benmeziane, Kaoutar El Maghraoui, Hamza Ouarnoughi, Smail Niar, Martin Wistuba, and Naigang Wang. 2021. Hardware-aware neural architecture search: Survey and taxonomy. In *Proceedings of the International Joint Conference on Artificial Intelligence*. 4322–4329.

[8] Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. 2018. Efficient architecture search by network transformation. In *Proceedings of the AAAI Conference on Artificial Intelligence and Innovative Applications of Artificial Intelligence Conference and AAAI Symposium on Educational Advances in Artificial Intelligence*. Article 340, 2787–2794.

[9] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. 2020. Once-for-all: Train one network and specialize it for efficient deployment. In *Proceedings of the International Conference on Learning Representations*.

[10] Han Cai, Ligeng Zhu, and Song Han. 2019. ProxylessNAS: Direct neural architecture search on target task and hardware. In *Proceedings of the International Conference on Learning Representations*.

[11] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Vol. 42. 269–284.

[12] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A machine-learning supercomputer. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. 609–622.

[13] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. 2019. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE J. Emerg. Select. Top. Circ. Syst.* 9, 2 (2019), 292–308.

[14] Hsin-Pai Cheng, Tunhou Zhang, Yixing Zhang, Shiyu Li, Feng Liang, Feng Yan, Meng Li, Vikas Chandra, Hai Li, and Yiran Chen. 2021. NASGEM: Neural architecture search via graph embedding method. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 7090–7098.

[15] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. 2018. PACT: Parameterized clipping activation for quantized neural networks. CoRR abs/1805.06085.

[16] Kanghyun Choi, Deokki Hong, Hojae Yoon, Joonsang Yu, Youngsok Kim, and Jinho Lee. 2021. DANCE: Differentiable accelerator/network co-exploration. In *Proceedings of the ACM/IEEE Design Automation Conference*. 337–342.

[17] François Chollet. 2017. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1800–1807.

[18] Xiaoliang Dai, Hongxu Yin, and Niraj K. Jha. 2019. NeST: A neural network synthesis tool based on a grow-and-prune paradigm. *IEEE Trans. Comput.* 68, 10 (2019), 1487–1497.

[19] Xiaoliang Dai, Peizhao Zhang, Bichen Wu, Hongxu Yin, Fei Sun, Yanghan Wang, Marat Dukhan, Yunqing Hu, Yiming Wu, Yangqing Jia, Peter Vajda, Matt Uyttendaele, and Niraj K. Jha. 2019. ChamNet: Towards efficient network design through platform-aware model adaptation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, Vol. 10. 11390–11399.

[20] Duc-Cuong Dang, Anton Eremeev, and Per Kristian Lehre. 2021. Escaping local optima with non-elitist evolutionary algorithms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 12275–12283.

[21] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 248–255.

[22] Xiangyu Dong, Cong Xu, Yuan Xie, and Norman P. Jouppi. 2012. NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 31, 7 (2012), 994–1007.

[23] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*. 92–104.

[24] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2019. Efficient multi-objective neural architecture search via Lamarckian evolution. In *Proceedings of the International Conference on Learning Representations*.

[25] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2019. Neural architecture search: A survey. *J. Mach. Learn. Res.* 20, 55 (2019), 1–21.

[26] Yonggan Fu, Yongan Zhang, Yang Zhang, David Cox, and Yingyan Lin. 2021. Auto-NBA: Efficient and effective search over the joint space of networks, bitwidths, and accelerators. In *Proceedings of the International Conference on Machine Learning*, Vol. 139. 3505–3517.

[27] Yarin Gal and Zoubin Ghahramani. 2016. Dropout as a Bayesian approximation: Representing model uncertainty in deep learning. In *Proceedings of the International Conference on Machine Learning*, Vol. 48. 1050–1059.

[28] Abdullah Guler and Niraj K. Jha. 2018. Hybrid monolithic 3-D IC floorplanner. *IEEE Trans. VLSI Syst.* 26, 10 (2018), 1868–1880.

[29] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. In *Proceedings of the International Conference on Machine Learning*. 1737–1746.

[30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.

[31] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q. Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2261–2269.

[32] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. CoRR abs/1602.07360.

[33] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the International Conference on Machine Learning*, Vol. 37. 448–456.

[34] Weiwen Jiang, Lei Yang, Edwin Hsing-Mean Sha, Qingfeng Zhuge, Shouzhen Gu, Sakyasingha Dasgupta, Yiyu Shi, and Jingtong Hu. 2020. Hardware/software co-exploration of neural architectures. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 39, 12 (2020), 4805–4815.

[35] Asifullah Khan, Anabia Sohail, Umme Zahoora, and Aqsa Saeed Qureshi. 2020. A survey of the recent architectures of deep convolutional neural networks. *Artif. Intell. Rev.* 53, 8 (2020), 5455–5516.

[36] Alex Krizhevsky. 2009. *Learning Multiple Layers of Features from Tiny Images*. Master's thesis, Department of Computer Science, University of Toronto.

[37] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *Proceedings of the International Conference on Neural Information Processing Systems*, Vol. 1. 1097–1105.

[38] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[39] Chaojian Li, Zhongzhi Yu, Yonggan Fu, Yongan Zhang, Yang Zhao, Haoran You, Qixuan Yu, Yue Wang, Cong Hao, and Yingyan Lin. 2021. HW-NAS-Bench: Hardware-aware neural architecture search benchmark. In *Proceedings of the International Conference on Learning Representations*.

[40] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-tzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. 2020. A system for massively parallel hyperparameter tuning. In *Proceedings of the Machine Learning and Systems*, Vol. 2. 230–246.

[41] Yuhong Li, Cong Hao, Xiaofan Zhang, Xinheng Liu, Yao Chen, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. 2020. EDD: Efficient differentiable DNN architecture and implementation co-search for embedded AI solutions. In *Proceedings of the ACM/EDAC/IEEE Design Automation Conference*. Article 130, 6 pages.

[42] Ji Lin, Wei-Ming Chen, Yujun Lin, John Cohn, Chuang Gan, and Song Han. 2020. MCUNet: Tiny deep learning on IoT devices. In *Proceedings of the International Conference on Neural Information Processing Systems*, Vol. 33. 11711–11722.

[43] Min Lin, Qiang Chen, and Shuicheng Yan. 2014. Network in network. CoRR abs/1312.4400.

[44] Yujun Lin, Mengtian Yang, and Song Han. 2021. NAAS: Neural accelerator architecture search. In *Proceedings of the ACM/IEEE Design Automation Conference*. 1051–1056.

[45] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2018. DARTS: Differentiable architecture search. In *Proceedings of the International Conference on Learning Representations*.

[46] Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In *Proceedings of the International Conference on Learning Representations*.

[47] Sanjai Narain, Emily Mak, Dana Chee, Brendan Englot, Kishore Pochiraju, Niraj K. Jha, and Karthik Narayan. 2022. Fast design space exploration of nonlinear systems: Part I. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 41, 9 (2022), 2970–2983.

[48] Xuefei Ning, Yin Zheng, Tianchen Zhao, Yu Wang, and Huazhong Yang. 2020. A generic graph-based neural architecture encoding scheme for predictor-based NAS. In *Proceedings of the European Conference on Computer Vision*. 189–204.

[49] Matthew Poremba, Tao Zhang, and Yuan Xie. 2015. NVMain 2.0: A user-friendly memory simulator to model (non-)volatile memory systems. *IEEE Comput. Arch. Lett.* 14, 2 (2015), 140–143.

[50] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. 2017. Searching for activation functions. CoRR abs/1710.05941.

[51] Brandon Reagen, José Miguel Hernández-Lobato, Robert Adolf, Michael Gelbart, Paul Whatmough, Gu-Yeon Wei, and David Brooks. 2017. A case for efficient accelerator design space exploration via Bayesian optimization. In *Proceedings of the IEEE/ACM International Symposium on Low Power Electronics and Design*. 1–6.

[52] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.* 4510–4520.
[53] David Schor. 2018. Hot Chips 30: Nvidia Xavier SoC. Retrieved from https://hc30.hotchips.org/.
[54] Alireza Shafaei, Yanzhi Wang, Xue Lin, and Massoud Pedram. 2014. FinCACTI: Architectural analysis and modeling of caches with deeply-scaled FinFET devices. In *Proceedings of the IEEE Annual Symposium on VLSI.* 290–295.
[55] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2014. Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture.* 97–108.
[56] Yongming Shen, Michael Ferdman, and Peter Milder. 2017. Maximizing CNN accelerator efficiency through resource partitioning. *SIGARCH Comput. Arch. News* 45, 2 (2017), 535–547.
[57] Julien Siems, Lucas Zimmer, Arber Zela, Jovita Lukasik, Margret Keuper, and Frank Hutter. 2022. Surrogate NAS benchmarks: Going beyond the limited search spaces of tabular NAS benchmarks. In *Proceedings of the International Conference on Learning Representations.*
[58] Karen Simonyan and Andrew Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. In *Proceedings of the International Conference on Learning Representations.*
[59] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. 2012. Practical Bayesian optimization of machine learning algorithms. In *Proceedings of the International Conference on Neural Information Processing Systems*, Vol. 25. 2951–2959.
[60] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.* 15, 56 (2014), 1929–1958.
[61] Aaron Stillmaker and Bevan Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180nm to 7nm. *Integration* 58 (2017), 74–81.
[62] Hanbo Sun, Chenyu Wang, Zhenhua Zhu, Xuefei Ning, Guohao Dai, Huazhong Yang, and Yu Wang. 2022. Gibbon: Efficient co-exploration of NN model and processing-in-memory architecture. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition.* 867–872.
[63] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel Emer. 2020. *Efficient Processing of Deep Neural Networks.* Morgan & Claypool Publishers.
[64] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 1–9.
[65] Mingxing Tan and Quoc Le. 2019. EfficientNet: Rethinking model scaling for convolutional neural networks. In *Proceedings of the International Conference on Machine Learning*, Vol. 97. 6105–6114.
[66] Shikhar Tuli and Debanjan Bhowmik. 2020. Design of a conventional-transistor-based analog integrated circuit for on-chip learning in a spiking neural network. In *Proceedings of the International Conference on Neuromorphic Systems 2020.* Article 14, 8 pages.
[67] Shikhar Tuli, Bhishma Dedhia, Shreshth Tuli, and Niraj K. Jha. (2022) FlexiBERT: Are current transformer architectures too homogeneous and rigid? CoRR abs/2205.11656.
[68] Shreshth Tuli, Shivananda R. Poojara, Satish N. Srirama, Giuliano Casale, and Nicholas R. Jennings. 2022. COSCO: Container orchestration using co-simulation and gradient based optimization for fog computing environments. *IEEE Trans. Parallel Distrib. Syst.* 33, 1 (2022), 101–116.
[69] Miheer Vaidya, Aravind Sukumaran-Rajam, Atanas Rountev, and P. Sadayappan. 2022. Comprehensive accelerator-dataflow co-design optimization for convolutional neural networks. In *Proceesings of the IEEE/ACM International Symposium on Code Generation and Optimization.* 325–335.
[70] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the International Conference on Neural Information Processing Systems*, Vol. 30. 5998–6008.
[71] Hao Wang, Xingjian Shi, and Dit-Yan Yeung. 2016. Natural-parameter networks: A class of probabilistic neural networks. In *Proceedings of the International Conference on Neural Information Processing Systems.* 118–126.
[72] Colin White, Willie Neiswanger, and Yash Savani. 2021. BANANAS: Bayesian optimization with neural architectures for neural architecture search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 10293–10301.
[73] Colin White, Sam Nolen, and Yash Savani. 2020. Local search is state of the art for NAS benchmarks. CoRR abs/2005.02960.
[74] Ronald J. Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.* 8, 3–4 (1992), 229–256.
[75] Yannan Nellie Wu, Joel S. Emer, and Vivienne Sze. 2019. Accelergy: An architecture-level energy estimation methodology for accelerator designs. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design.* 1–8.

[76] Wenhan Xia, Hongxu Yin, Xiaoliang Dai, and Niraj K. Jha. 2022. Fully dynamic inference with deep neural networks. *IEEE Trans. Emerg. Top. Comput.* 10, 2 (2022), 962–972.

[77] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How powerful are graph neural networks? In *Proceedings of the International Conference on Learning Representations.*

[78] Lei Yang, Zheyu Yan, Meng Li, Hyoukjun Kwon, Weiwen Jiang, Liangzhen Lai, Yiyu Shi, Tushar Krishna, and Vikas Chandra. 2020. Co-exploration of neural architectures and heterogeneous ASIC accelerator designs targeting multiple tasks. In *Proceedings of the ACM/EDAC/IEEE Design Automation Conference.* Article 163, 6 pages.

[79] Zhewei Yao, Amir Gholami, Sheng Shen, Mustafa Mustafa, Kurt Keutzer, and Michael Mahoney. 2021. ADAHESSIAN: An adaptive second order optimizer for machine learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 10665–10673.

[80] Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. 2019. NAS-Bench-101: Towards reproducible neural architecture search. In *Proceedings of the International Conference on Machine Learning*, Vol. 97. 7105–7114.

[81] Chris Ying, Sameer Kumar, Dehao Chen, Tao Wang, and Youlong Cheng. 2018. Image classification at supercomputer scale. CoRR abs/1811.06992.

[82] Ye Yu and Niraj K. Jha. 2018. Energy-efficient monolithic three-dimensional on-chip memory architectures. *IEEE Trans. Nanotechnol.* 17, 4 (2018), 620–633.

[83] Ye Yu and Niraj K. Jha. 2022. SPRING: A sparsity-aware reduced-precision monolithic 3D CNN accelerator architecture for training and inference. *IEEE Trans. Emerg. Top. Comput.* 10, 1 (2022), 237–249.

[84] Ye Yu, Yingmin Li, Shuai Che, Niraj K. Jha, and Weifeng Zhang. 2021. Software-defined design space exploration for an efficient DNN accelerator architecture. *IEEE Trans. Comput.* 70, 1 (2021), 45–56.

[85] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* 161–170.

[86] Muhan Zhang, Shali Jiang, Zhicheng Cui, Roman Garnett, and Yixin Chen. 2019. D-VAE: A variational autoencoder for directed acyclic graphs. In *Proceedings of the International Conference on Neural Information Processing Systems*, Vol. 32. 1588–1600.

[87] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-X: An accelerator for sparse neural networks. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture.* 1–12.

[88] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-Mei Hwu, and Deming Chen. 2018. DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design.* 1–8.

[89] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2018. ShuffleNet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.* 6848–6856.

[90] Xuda Zhou, Zidong Du, Qi Guo, Shaoli Liu, Chengsi Liu, Chao Wang, Xuehai Zhou, Ling Li, Tianshi Chen, and Yunji Chen. 2018. Cambricon-S: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture.* 15–28.

[91] Yanqi Zhou, Xuanyi Dong, Daiyi Peng, Ethan Zhu, Amir Yazdanbakhsh, Berkin Akin, Mingxing Tan, and James Laudon. 2021. NAHAS: Neural Architecture and Hardware Accelerator Search. Retrieved from https://openreview.net/forum?id=fgpXAu8puGj.

[92] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. 2018. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 8697–8710.

Received 25 August 2022; revised 27 October 2022; accepted 29 November 2022