NNBench-X: A Benchmarking Methodology for Neural Network Accelerator Designs

XINFENG XIE, XING HU, PENG GU, SHUANGCHEN LI, YU JI, and YUAN XIE, University of California, Santa Barbara, United States

The tremendous impact of deep learning algorithms over a wide range of application domains has encouraged a surge of neural network (NN) accelerator research. Facilitating the NN accelerator design calls for guidance from an evolving benchmark suite that incorporates emerging NN models. Nevertheless, existing NN benchmarks are not suitable for guiding NN accelerator designs. These benchmarks are either selected for general-purpose processors without considering unique characteristics of NN accelerators or lack quantitative analysis to guarantee their completeness during the benchmark construction, update, and customization.

In light of the shortcomings of prior benchmarks, we propose a novel benchmarking methodology for NN accelerators with a quantitative analysis of application performance features and a comprehensive awareness of software-hardware co-design. Specifically, we decouple the benchmarking process into three stages: First, we characterize the NN workloads with quantitative metrics and select the representative applications for the benchmark suite to ensure diversity and completeness. Second, we refine the selected applications according to the customized model compression techniques provided by specific software-hardware co-design. Finally, we evaluate a variety of accelerator designs on the generated benchmark suite. To demonstrate the effectiveness of our benchmarking methodology, we conduct a case study of composing an NN benchmark from the TensorFlow Model Zoo and compress these selected models with various model compression techniques. Finally, we evaluate compressed models on various architectures, including GPU, Neurocube, DianNao, and Cambricon-X.

CCS Concepts: • **Hardware** → **Application specific integrated circuits**; • **General and reference** → *Evaluation*; • **Computer systems organization** → *Neural networks*;

Additional Key Words and Phrases: Neural networks, accelerator, software-hardware co-designs, benchmark

ACM Reference format:
Xinfeng Xie, Xing Hu, Peng Gu, Shuangchen Li, Yu Ji, and Yuan Xie. 2020. NNBench-X: A Benchmarking Methodology for Neural Network Accelerator Designs. *ACM Trans. Archit. Code Optim.* 17, 4, Article 31 (November 2020), 25 pages.
https://doi.org/10.1145/3417709

This work was supported in part by NSF 1816833, 1719160, 1725447, and 1730309.
Authors' address: X. Xie, X. Hu, P. Gu, S. Li, Y. Ji, and Y. Xie, University of California, Santa Barbara, California, 93106; emails: {xinfeng, xinghu}@ucsb.edu, {peng_gu, shuangchenli}@ece.ucsb.edu, maple.jiyu@hotmail.com, yuanxie@ece.ucsb.edu.

1544-3566/2020/11-ART31
https://doi.org/10.1145/3417709

1 INTRODUCTION

Neural network (NN) algorithms have demonstrated better accuracy than traditional machine learning algorithms in a wide range of application domains, such as computer vision (CV) [27, 28, 49, 51] and natural language processing (NLP) [35, 46, 50]. These breakthroughs indicate a promising future for their real-world deployment. Deploying these applications, especially for the inference stage, requires high performance under stringent power budgets, which boosts the emergence of accelerator designs for these applications. However, designing such an NN accelerator using application-specific integrated circuits (ASICs) is challenging, because NN applications are changing rapidly to support new functionalities and improve accuracies, while ASIC design requires a long design and manufacturing period. The accelerator design could be prone to becoming obsolete if the design fails to capture key characteristics of emerging models. Therefore, a benchmark to capture these workload characteristics is crucial to guiding NN accelerator design. Although there exist some benchmark suites for NN accelerator designs [1, 3, 11, 52], most of them overlook two critical perspectives when constructing the benchmark suite.

Prior studies usually lack quantitative analysis in the selection of applications for the benchmark suite. Without quantitative metrics for selecting applications, it will be difficult to maintain a representative benchmark suite in benchmark construction, update, and customization. First, when the benchmark suite is originally constructed, there is a risk that the empirically selected applications are not the most representative collection, although most of the existing benchmarks justify their representativeness afterward. Second, due to the rapid change of NN algorithms from the machine learning community, a benchmark suite needs to be updated periodically to consider new algorithms. However, without quantitative metrics, it is unclear if existing applications in the benchmark suite are representative of emerging algorithms. Finally, evaluating NN accelerators designed for a special application domain needs to filter some applications from a benchmark, which can hardly be achieved without quantitative metrics. For example, designing a smart camera does not need to evaluate sequence-to-sequence [50] models. The process of filtering unrelated applications needs quantitative metrics instead of empirical decisions to ensure representativeness.

In addition, prior benchmark selection does not take the specialty of NN accelerators into account, and hence is not suitable for evaluating software-hardware co-designs. Without this kind of consideration, existing benchmarks are not feasible to evaluate several state-of-the-art NN accelerators exploiting NN model compression techniques. First, most accelerators are incorporated with specialized hardware for software optimizations, so evaluating applications without these optimizations cannot provide insightful guidelines. For example, some accelerators, such as the TPU [32] and DianNao [12], exploit fixed-point arithmetic logic units (ALUs) for quantized models, while some accelerators, such as EIE [24] and Cambricon-X [64], exploit sparse tensor computations for pruned models. TPU and DianNao cannot benefit from the sparsity, while EIE and Cambricon-X could suffer from the overheads of control logic for running dense NN models. Second, without the consideration of hardware-software co-design, it is impossible to evaluate software-level optimizations and their impact on hardware designs. In particular, using only one set of workloads can hardly study the performance impact of software-level optimizations on specialized hardware designs, such as performance benefits that the hardware design can obtain if a new pruning algorithm can further prune half of the weights. Third, without considering software-hardware co-design during the process of composing benchmark suites, the application set could include similar and redundant applications, such as VGG and SparseVGG in BenchIP [52].

In this article, we propose an end-to-end benchmarking approach for software-hardware co-design to quantitatively select applications and benchmark software-hardware co-design by decoupling our approach into three stages: workload characterizations, software-level model compression strategies, and hardware-level accelerator evaluations. In the first stage, *application set*

selection, we characterize NN applications of interest without considering any software optimization techniques. After gathering their performance features, we select representative applications for the original application set. In the second stage, *benchmark suite generation*, users can refine the selected applications to generate the final benchmark suite according to their model compression strategies. New NN models for each application in the original benchmark suite will be generated according to software-level optimizations, such as quantizing and pruning techniques. In the last stage, *hardware evaluation*, users can provide the performance models of their accelerator designs together with the assumptions of interconnection and host. Accelerators are evaluated with the benchmark suite generated from the second stage. Power, performance, and area results are derived according to input performance models.

To demonstrate the functionality of our benchmark, we conduct a case study on designing NN accelerators for general NN applications. First, we comprehensively analyze 57 models with 224,563 operators from the TensorFlow (TF) Model Zoo [21]. Second, we generate benchmark suites by using several state-of-the-art software-level optimizations including quantizing and pruning NN models. Finally, we evaluate several representative accelerators including general-purpose processors (CPU and GPU), accelerator architecture (DianNao [12]), near-data-processing architecture (Neurocube [33]), and sparse-aware architecture (Cambricon-X [64]).

Our contributions can be summarized as follows:

- We propose a novel benchmarking method, which selects the benchmark by analyzing a user-input candidate application pool and covers software-hardware co-design configurations with high flexibility. Therefore, our benchmark method is able to provide guidelines for architecture design to tradeoff application compatibility, algorithm accuracy, and hardware performance.
- We conduct a case study of generating a general-purpose NN benchmark suite from the TF Model Zoo while applying state-of-the-art NN model compression techniques and evaluate it on representative architectures to demonstrate the functionality of our benchmark method. Our case study reveals that CV and NLP applications show very different performance characteristics and favor different compression techniques and hardware architectures.

2 BACKGROUND

In this section, we introduce the basics of NN accelerator system stacks and NN accelerator designs.

2.1 System Stack and the Representation of an NN Model

Modern NN development and deployment system stacks are decoupled into several levels. As shown in Figure 1, the whole system stack includes application, framework, primitive, and hardware levels. From top to bottom, the application level focuses on developing high accuracy algorithms and sometimes makes tradeoffs between accuracy and performance when exploring different NN structures. The framework level focuses on transforming high-level abstractions into hardware primitives by providing a flexible programming model and efficient runtime environment. Meanwhile, the primitive level provides simple and well-optimized primitives for the hardware. For example, cuDNN [40] provides well-optimized library for executing convolution on GPUs. At the bottom of the whole development and deployment stack, the hardware level provides efficient hardware platforms for executing NN applications.

Across these system stack levels, each NN model is represented by a computation graph, which abstracts tensor operators as vertexes and tensor operands as edges to present an NN model. The topology of computation graphs indicate the data dependency among tensor operators. Figure 2

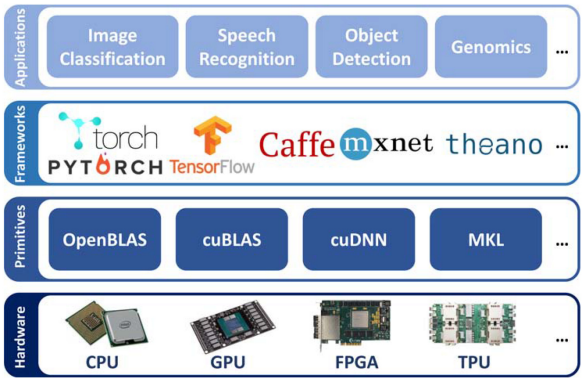


Fig. 1. System stack for the development and deployment of NN applications including (1) application layer, (2) framework layer, (3) primitive layer, and (4) hardware layer.

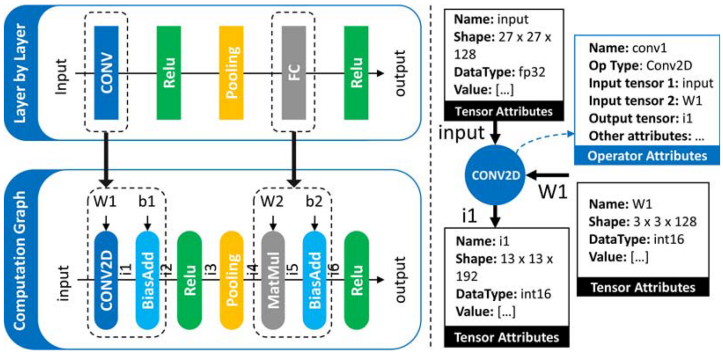


Fig. 2. An NN example represented by the layer-by-layer abstraction and the computation graph with the detailed components of a Conv2D operator to explain what is included in an operator.

provides an example NN represented by these two abstractions to demonstrate their differences. The computation graph abstraction brings a more flexible representation of NN models and modern frameworks, such as TensorFlow [2] and PyTorch [16], adopt computation graph as the programming model. Thus, the computation graph representation is general across different NN frameworks. Moreover, because the computation graph does not have any constraint on the graph topology, it is fully compatible with all widely used NN models including RNN models even though it could introduce loops in the computation graph. All models from TensorFlow Model Zoo [21] are represented by TensorFlow graphs, which is an implementation of the computation graph concept. In the rest of this article, we adopt this abstraction taking an NN model as a computation graph.

2.2 NN Accelerators

For the past few years, researchers have adopted two major guidelines to improve NN accelerator designs, i.e., the technology-driven architecture designs and the application-driven architecture designs.

From the technology perspective, researchers aim to utilize the physical properties of emerging hardware primitives to fundamentally improve the performance and energy efficiency of new architectures. Some key operations in NN applications that are bottlenecks, such as matrix multiplication, are especially suited to these architectures. New technologies such as emerging

Table 1. Classifying NN Benchmarks w.r.t. *Benchmark-suite* and *Benchmark-object*

	BenchNN [11]	BenchDL [48]	DeepBench [7]	Fathom [3]	BenchIP [52]	**Our Work**
Application	★	★		★	★	★ □
Framework		□				
Primitive			★		★	
Hardware	□		□	□	□	□

★: benchmark-suite; □: benchmark-object.

non-volatile memories and 3D die stacking provide new opportunities for the implementation of PIM accelerators [15, 30], near-data processing (NDP) accelerators [33], and neuromorphic chips [10], which demonstrate orders of magnitude improvement in energy efficiency and performance.

From the application perspective, researchers aim to simplify computational workloads and reduce memory footprint through algorithmic optimizations without significantly compromising application accuracy. Previous work [42, 54, 55] demonstrates that inference tasks do not require high numerical precision for weights and intermediate data. There are a number of ASIC designs [12–14, 32] leveraging these opportunities to improve the performance and energy efficiency for NN inference tasks. Another promising optimization strategy is pruning [25, 26], which removes unnecessary connections in NN models and makes tensor operations sparse. Some accelerators [4, 24, 58, 64] are designed to utilize the sparsity of either weights or activations. In addition, software-hardware co-design methodology [23, 43, 61] with architecture-aware NN model compression [34, 56] or compression-aware accelerator is proposed to figure out the best tradeoffs between accuracy and performance.

In addition to ASIC accelerators with fixed hardware architectures that rely on software to convert NN models into programs running on hardware, some accelerators, such as BrainWave [19] and xDNN [60], convert NN models completely into hardware and realize them through reconfigurable logic, especially FPGAs. Although studies of using FPGAs start from supporting only a limited set of models [41, 62], the design method is extended to support a wide spectrum of NN models [47, 63, 65]. These studies usually use hand-optimized RTL templates for key operations and rely on compiler support to efficiently leverage these well-optimized modules.

In this article, our benchmark aims to provide a comprehensive understanding of NN workloads to guide accelerator designs regardless of their technology, application domain, and design methods. Moreover, our benchmark helps domain-specific accelerator designs instead of accelerators tailored for only one or few models, because accelerators for a limited number of models have concrete design goals and the known set of representative workloads.

3 MOTIVATION

Although many NN benchmark suites have recently been proposed, through analyzing available suites, we see that many demands are not met. We first narrow down the analysis of existing suites by categorizing all previous benchmarks in terms of *benchmark-suite* and *benchmark-object*. Then, we highlight the novelty of this work by comparing it to BenchIP [52] and Fathom [3] in four detailed aspects.

All previous NN benchmarks can be categorized according to the *benchmark-suite* and *benchmark-object*. A *benchmark-suite* consists of a set of representative workloads to be evaluated on different *benchmark-objects*. We classify the benchmark-suite and benchmark-object into different levels in the system stack, as shown in Table 1. Although BenchNN [11] is one of

Table 2. The Uniqueness of Our Benchmarking Methodology

	Fathom	BenchIP	**Ours**
Analysis-based App. Selection	✗	✗	✓
Flexible with Update/Customize	✗	✗	✓
SW/HW Co-design	✗	*fixed*	*general*
Evaluation on Accelerators	✗	*ASIC*	*ASIC/NDP*

✗ means the corresponding feature is not supported, and ✓ means the corresponding feature is supported.

earliest efforts in building an NN benchmark, the benchmark-suite is a bit out of date without updates. Prior study [48] (denoted as BenchDL) proposes a benchmark suite for evaluating different deep learning software tools, i.e., frameworks in our system stack of NN applications. DeepBench [7] is a benchmark suite comparing the performance of different primitives on different platforms. However, benchmarking NN applications from the primitive layer loses the whole picture. Fathom [3] and BenchIP [52] serve a similar purpose as our work. However, they do not take software-hardware co-design as the benchmark object. *Different from all of them, our benchmarking methodology targets at capturing end-to-end application-to-hardware characteristics to guide architecture design for state-of-the-art NN workloads.* Since both Fathom and BenchIP serve a similar purpose of benchmarking NN accelerator designs, we further detail our comparison with Fathom and BenchIP in four aspects, as summarized in Table 2.

Quantitative analysis–based benchmark selection: Accelerator designers usually know the application domain they are interested in, which could include a large number of NN applications. Thus, it is important to select representative NN applications to guide hardware architecture design. Fathom and BenchIP pick their applications with some empirical guidelines but not by any quantitative analysis. Even though they show the effectiveness of their selected suits afterward, there is no guarantee that their selections are the **most** representative. *On the contrary, our approach selects benchmarks according to the results of extensive profiling and analyzing.* Our method characterizes NN applications through application features that are key to the performance, from the perspective of architecture designs. At the end of Section 6.1, we show how our method captures additional features that other benchmarks fail to cover.

Flexible with updates and customizations: We propose a benchmarking methodology, **not** simply a benchmark suite. By doing this, we are subject to updates due to the rapid developing NN algorithms. Statistics [52] have shown that within one year, the NN models proposed in top-tier conferences double. For a fixed benchmark suite, it is difficult to know whether to extend the suite and whether a new accelerator is needed when a new model appears. Although evaluating a new model on existing accelerators can help us understand its characteristics to some extent, the demand for updating the benchmark suite and designing a new accelerator would be challenging without an in-depth workload characterization. In addition, most of the accelerators target a certain application scenario (e.g., autonomous cars), instead of a general NN processor. A single one-for-all benchmark suite does not adequately address these needs. Instead, we generate different suites according to the user-customized candidate application pool.

SW/HW co-design: Recent NN accelerator designs usually include both software optimizations, such as model pruning and quantization, and hardware optimizations. Our benchmark method is the first for accelerators with a comprehensive awareness of software-hardware co-design. Although BenchIP [52] includes sparse models, such as Sparse VGG, into their application set as representative workloads, these considerations are insufficient due to two reasons. First, pruned models are very similar to their original models in their work. For example, Sparse VGG

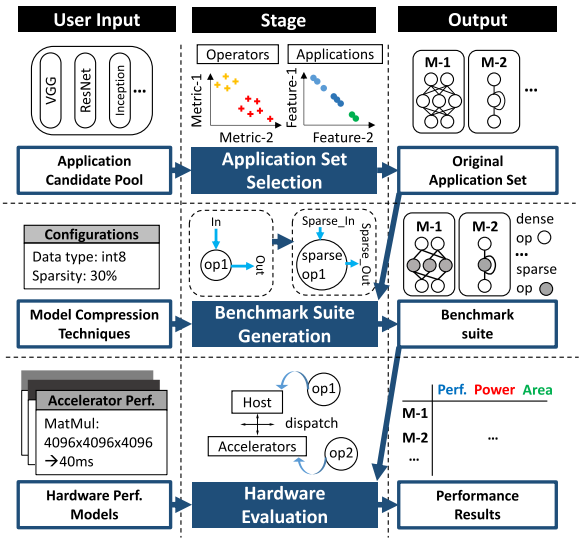


Fig. 3. Benchmark method overview with three main stages and their corresponding inputs and outputs.

performs very similar to VGG in terms of extracted performance features, making it redundant. Second, their sparsity benchmark cannot consider all model compression techniques. For example, structural sparsity [61] is not covered.

Diversity of evaluation platforms: Because of the growing heterogeneity of hardware platforms, targeting only ASIC designs is not sufficient. We evaluate our benchmarks not only on CPU/GPU and ASICs but also on other innovative architectures such as NDP architectures. In addition, our evaluation method is not limited to any NN framework. Instead, we use the computation graph as a programming model with a general abstraction for the execution of NN applications across different platforms.

4 BENCHMARKING METHODOLOGY

An overview of our benchmarking method is shown in Figure 3. Our benchmarking method includes three stages. The first stage is *application set selection*, with an application candidate pool as its user input and original application set as its output [59]. The second stage is *benchmark suite generation*, with the model compression technique as the user input and the previous generated original application set as another input. The last stage is the *hardware evaluation*, which takes the generated benchmark suite and the hardware performance models as its inputs and then outputs the performance results. The rest of this section will introduce these three stages in detail.

4.1 Application Set Selection

In the first stage, application set selection, we select diverse and representative NN applications from the application candidate pool that includes the applications of the user's interests.

The proposed application set selection consists of two phases: operator-level and application-level analysis, as shown in Figure 4. Since tensor operators are the primitives of NN applications, operator-level analysis is conducted first, before application-level analysis. In the operator-level analysis, we extract all operators from the application candidate pool and use two important metrics, locality and parallelism, as the performance feature to represent an operator. Then, all the operators are clustered into several groups according to the extracted operator features.

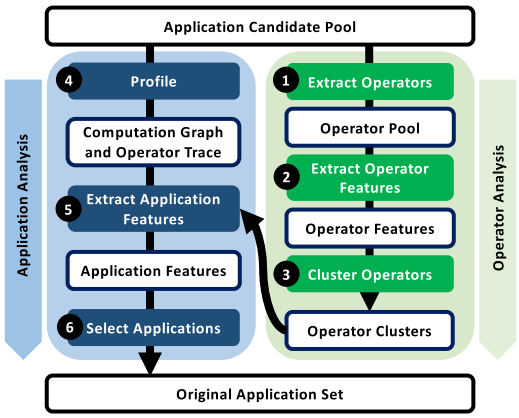


Fig. 4. Application set selection process with two phases: operator-level analysis phase and application-level analysis phase.

This process of getting operator clusters is detailed as Algorithm 1. After the operator-level clustering, application-level analysis is performed as the second phase. Applications are first profiled on baseline architectures before they are quantified by time breakdown on the different operator clusters. The process of getting application features is detailed as Algorithm 2. After obtaining these application features, we conduct a similarity analysis for all applications. Finally, an application set composed of diverse and representative workloads can be selected out of the application candidate pool. Instead of clustering operators according to their functionalities, as in prior work [3], our work is fundamentally different, because it clusters tensor operators according to their architectural features, i.e., locality and parallelism. We observe that functionality-based classification is not sufficient and can cause incorrect bottleneck characterization, as validated by the experiments at the end of Section 6.1.

4.1.1 Operator-level Analysis. As shown in Figure 4, we perform operator-level analysis in the first phase to extract operator features and cluster operators based on these operator features. Our operator-level analysis first extracts all operators from the applications in the application candidate pool. Then, we analyze operator features from the perspective of architecture designs. Finally, we cluster these operators.

To improve the generality of the generated benchmark suite, we use platform-independent metrics as the operator feature. Specifically, we define two platform-independent metrics, *Locality* and *Parallelism*, for the operator-level analysis to reflect general architecture considerations when designing accelerators for tensor operators. A common practice in accelerator design is to consider customized data-path designs, such as the different dataflow structures in Eyeriss [14], that can leverage both the locality of these operators and can utilize multiple processing elements (PEs) to exploit the available parallelism. Thus, these two platform-independent metrics can be useful to help understand operators from the viewpoint of architectural designs for overall demands. The definition of these two metrics used to represent the architectural feature of an operator is detailed as follows:

Locality. This metric is defined as the amount of data needed by an operator divided by the number of scalar arithmetic computations it needs. The amount of data needed by an operator is equal to the sum of the input tensor size and the output tensor size. Input tensors include all input data needed by this operator, such as model weights. Our locality metric reflects the overall locality of an operator, because it indicates the average times of a byte used in the scalar arithmetic

computations. Moreover, the average times of a byte used in the computation indicates the locality in an ideal memory system where a cache hit happens if the same location was accessed before. For example, when the locality metric of an operator equals to 0.1, it means that this operator performs an arithmetic scalar computation on 0.1 byte of data on average. In other words, each byte is used for 10 ($= \frac{1}{0.1}$) scalar computations on average. In an ideal memory system, this byte is accessed 10 times (1 access per arithmetic computations), and the miss rate is 10%, because only the first access of these 10 accesses will result in a cache miss. Another example is that when the locality metric of an operator equals to 12, it means that this operator performs an arithmetic scalar computation on 12 bytes of data on average. In this case, each data is accessed only once for the computation, and the miss rate in an ideal memory system is 100%, because there is no data-reuse. In summary, the cache miss rate of an operator in an ideal memory system is $min\{Locality, 100\%\}$ when the cache line size is 1 byte. Thus, lower values of this metric indicate better locality for the operator.

Parallelism. This metric is defined as the ratio of scalar arithmetic operations that can be executed in parallel, assuming sufficient hardware resources. Thus, the quantitative value of this metric falls into the range between 0 and 1. Higher values of this metric express greater available parallelism for the operator. This metric reflects the parallelism of computations in terms of data dependency. For example, a tensor *Add* operator that adds two tensors with N elements in an element-wise manner has N scalar-add operations. All of these scalar-add operations can be executed in parallel without any true dependency. Therefore, the parallelism for this tensor *Add* operator is 100%. Take a tensor *Max* operator as another example. The functionality of a tensor *Max* operator is to find the maximum value in the input tensor with N elements. A tree-based reduction can explore the parallelism with $logN$ sequential steps that must be executed in a sequential manner. In each step of this tree-based reduction, all of the N scalar-max operations can be executed in parallel given sufficient hardware resources. As a result, the parallelism for a tensor *Max* operator is $\frac{1}{logN}$.

After obtaining operator features in the aforementioned metrics, we can group operators into several clusters according to these operator features.

4.1.2 Application-level Analysis. As shown in Figure 4, we perform application-level analysis in the second phase to extract application features and select applications based on these application features. We define the performance feature of an application as the time breakdown on the different operator clusters obtained from the operator-level analysis. We denote the number of operator clusters as n. Specifically, the performance feature is denoted as $\vec{f} = (R_1, R_2, \ldots, R_n)$ where R_i represents the percentage of the elapsed time spent in the ith class operators. We profile each application from the application candidate pool on the baseline hardware, usually a CPU or a GPU, to obtain its time spent in each operator cluster. By analyzing applications in terms of time breakdown, benchmark users can have a better understanding of which operator class acts as a bottleneck on the baseline hardware. Because operators are grouped by their architecture features of both locality and parallelism, it provides clearer guidelines to design specialized hardware to accelerate the bottleneck operator cluster.

We rely on the application-level analysis phase to understand the application characteristics on baseline platforms. Thus, there are several major design decisions when we are building application features. First, we use profiling information on existing baseline platforms for a more accurate analysis. Although baseline platforms are usually general-purpose processors, such as CPU or GPU, they can be changed to other hardware devices, depending on design goals. For example, if NNBench-X is used to develop the second generation of TPU, the first version of TPU could be the baseline device [32]. Second, because this phase in the application set selection stage, this

ALGORITHM 1: Operator-level analysis to get operator clusters.

```
Input: A list of models (M) and the number of operator clusters (N)
Output: Operator cluster centers
Init All_Op_Features = []
for m in M do
    for op in m.operator_list() do
        op_features = ExtractOperatorFeatures(op)
        All_Op_Features.append(op_features)
    end for
end for
cluster_centers = kMeans(All_Op_Features, num_clusters=N)
Return cluster_centers
```

ALGORITHM 2: Application-level analysis to get application features.

```
Input: A list of models (M) and the centers of operator clusters (cluster_centers)
Output: The application features for each model (All_App_Features)
Init All_App_Features = []
for m in M do
    Init app_feature = [0.0] * len(cluster_centers)
    Init total_time = 0.0
    for op in m.operator_list() do
        op_features = ExtractOperatorFeatures(op)
        cluster_id = GetNearestClusterCenterID(op_features, cluster_centers)
        app_feature[cluster_id] += op.elapsed_time
        total_time += op.elapsed_time
    end for
    app_feature = app_feature / total_time
    All_App_Features.append(app_features)
end for
Return All_App_Features
```

phase needs to be independent from software-hardware co-design solutions to be evaluated by NNBench-X. Specifically, this phase does not take any software-hardware co-design solutions as inputs and extracts application features based on performance models of these co-designs, such as the roofline model [57]. Third, we do not consider inter-operator parallelism as a part of application features, because software frameworks usually take operators as the granularity of scheduling. These frameworks will offload operators to hardware instead of the whole computation graph and they are responsible to exploit inter-operator parallelism. However, when designing an accelerator taking the whole computation graph as inputs, this metric can be added into application features, as discussed in Section 7.

After this two-level analysis, we select representative applications out of the application candidate pool to build the original application set.

4.2 Benchmark Suite Generation

In the second stage, benchmark suite generation, we provide interfaces for users to customize their NN compression techniques to generate the final benchmark suite.

This stage is motivated by the success of model compression techniques, either quantizing or pruning, and the fact that state-of-the-art accelerator designs leverage these techniques for better computation and memory access efficiency by designing specialized hardware, either fixed-point

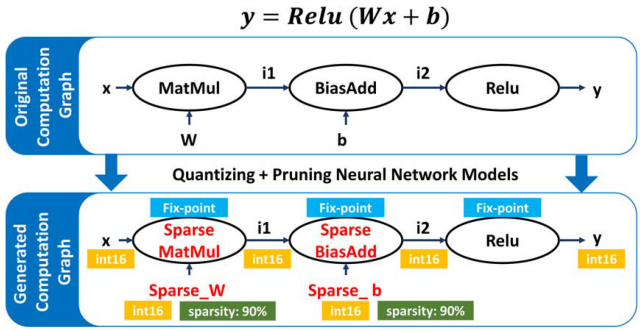


Fig. 5. An example for benchmark suite generation to generate a new computation graph according to user-provided quantizing (int16) and pruning strategies (sparse weights).

ALU or sparse tensor computation engines. Although we obtain a diverse and representative application set after the first stage, we cannot benchmark different accelerators using only one set of applications because of the diversity of NN model compression techniques.

Each application from the original application set is a computation graph. To customize different NN model compression techniques, we provide interfaces for the users to specify the data type of tensors in this computation graph. For tensors storing the pre-trained weights, users can overwrite these weights by using pruned weights so these tensors become sparse. Sparsity information can also be included as an additional attribute in the tensors storing weights. The sparsity of the tensors produced by activation functions, such as ReLU, can be computed in runtime. Figure 5 illustrates a case for these interfaces. Suppose we quantize the original application from the single-precision floating-point into 16-bit fixed-point and prune weights by 90%; the structure of the computation graph remains the same, but the operators and tensors are changed accordingly, as shown in Figure 5. Users can define and import model compression methods, and change the information of operators and tensors to generate the final benchmark suite according to their software-level studies in the training stage. Compression techniques resulting in intolerable accuracy degradations should not be imported into this stage. At the end of this stage, NNbench-X produces the final test set of applications composed of quantized and pruned NN models for evaluations.

Because our benchmark methodology provides interfaces for the users to specify their own compression methods instead of defining several patterns, NNBench-X is able to support a wide range of compression methods. For example, when NNBench-X is used to evaluate software-hardware co-designs exploiting the structural sparsity [6, 36], NNBench-X passes model weights to compression methods provided by the users to generate weights in structural sparse patterns. In this case, the pruned models with weights in structural sparse patterns will be in the generated benchmark suite at the end of this stage.

4.3 Hardware Evaluation

In the final stage, the hardware evaluation, we evaluate the generated benchmark suite on accelerator designs.

Although this stage can be completed by users with detailed simulation results of accelerators, we build a system-level simulator for fast performance estimation in the initial architecture design stages to provide high-level guidelines for accelerator designs. Our system-level simulator evaluates accelerators on the generated benchmark suite by using the performance models of the accelerator, the host, and the interconnection between the accelerator and the host. These performance models are provided by users so they can be as simple as a roofline model or as complicated

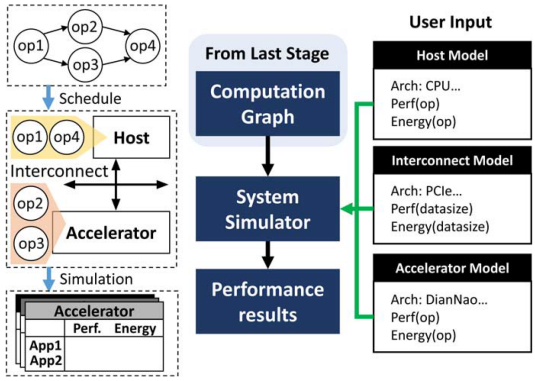


Fig. 6. The workflow of hardware evaluation with the user-inputs for hardware modeling including models for host, interconnect, and accelerators.

as a cycle-accurate simulator depending on the demands of hardware evaluation. For example, early design stages could use the roofline model to decide the balance between computation and memory resources while later design stages could need cycle-accurate simulators to model more hardware details. The inputs and outputs of our system-level simulator are shown in Figure 6. For each application in the generated benchmark suite, our simulator schedules operators into either the accelerator or the host by a first-come-first-serve scheduling algorithm. When an operator is not supported by the accelerator, it will be launched into the host with subsequent data transfer between the accelerator and the host. The performance results of running supported operators on accelerators and overheads of data transfer between the host and accelerators are provided by input hardware models that are a part of inputs to our system-level simulator. To demonstrate the usage of our system-level simulator, we use a simple but effective analytical model, the roofline model, in Section 6.2 to evaluate various architectures, including DianNao [12], Neurocube [33], and Cambricon-X [64].

Our system-level simulator plays a role similar to that of frameworks. Our straightforward scheduling policy may not consistently achieve optimal performance, but integrating accelerators into the whole system with developed primitives is time-consuming and impractical in the initial design space exploration stage for architectures. As the case study shows in Section 6.2, the performance speedups of different architectures could vary in orders of magnitudes. Therefore, our coarse but fast estimations can still provide insightful guidelines in architectural designs. Furthermore, the accuracy of estimation in this stage depends on the accuracy of performance models provided by users. Although we use a simple analytical model, roofline model, in Section 6.2 as a demo case, users can provide models capturing more hardware details to fit their demands exploiting various hardware designs. For example, when it is decided to use dataflow architectures in NN accelerators and our benchmark methodology is used to evaluate and compare different dataflow designs, the MAESTRO [37] framework can be used to provide the performance results of different architectures for supported operators. Another example is that when the users want to evaluate software-hardware co-designs exploiting structural sparsity, the user-provided performance models of hardware designs need to take the sparsity into account [6, 36]. In both examples, our system-level simulator is responsible to provide operator information, such as input tensor shapes and operator weights, while users need to implement their own performance models as the backend to return the performance results of running the operator on their accelerators. For accelerator designs in Section 6.2, we implement a roofline model as

the backend for various accelerator designs, which returns the performance by using the roofline model according to operator information and hardware specifications. For the performance of operators on real devices, such as CPU and GPU, we implement the backend performance model by running the operator on the real device and returning the measured time.

5 VALIDATION ON BENCHMARK METHODOLOGY

To validate the effectiveness of our benchmark methodology, we conduct case studies in Section 6. Before going to experimental results in Section 6, we explain how our case studies validate our benchmark methodology.

There are two major design goals of our benchmark methodology. First, our benchmark methodology is developed to quantitatively capture the architectural characteristics of applications from an input application pool to select diverse and representative ones. In Section 6.1, we conduct a case study of workload characterization on TensorFlow (TF) Model Zoo [21] covering a large number of models from different application domains. The characterization results on these models indicate that our benchmark methodology is able to distinguish workloads from different application domains and provide key architectural insights for application domains. Second, our benchmark methodology is developed to evaluate software-hardware co-design methods. In Section 6.2, we have several diverse software-hardware co-designs on the application set selected in Section 6.1. The evaluated software-hardware co-designs have a wide coverage, including both memory-centric accelerators [33] and compute-centric accelerators [12]. We also cover various model compression techniques, including both quantization [12, 33] and pruning [64]. The evaluation results on these software-hardware co-designs indicate that our benchmark methodology is able to capture key performance benefits of software-hardware co-designs.

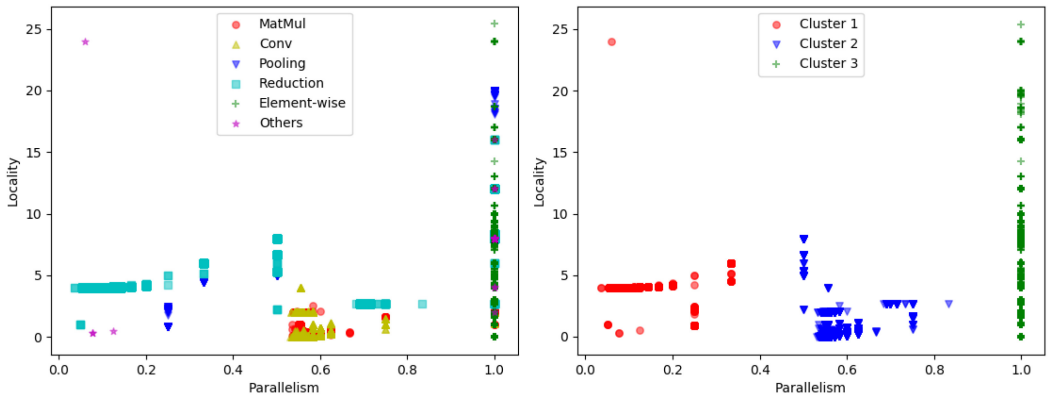
In summary, through these two main case studies on workload characterization and hardware evaluation, we are able to validate our benchmark methodology on its design goals.

6 CASE STUDY: FROM TENSORFLOW MODEL ZOO TO A BENCHMARK SUITE

We conduct a case study of benchmarking NN inference accelerators to demonstrate the usage of our benchmark approach. To this end, we set the TensorFlow (TF) Model Zoo [21] (with 57 NN models and 224,563 operators) as the application candidate pool, and our software-hardware co-design evaluation includes several state-of-the-art model compression techniques and hardware designs. The version of the TF Model Zoo we used in this case study contains 57 NN models from 24 different applications. These NN models have very diverse structures, including convolutional neural networks (CNNs) and recurrent neural networks (RNNs). From the perspective of learning algorithms, these models are from different learning methods, including supervised learning, unsupervised learning, and reinforcement learning. Thus, our application pool has very good coverage on existing NN applications from different application domains, with different model structures and trained by different learning algorithms. This section follows the three-step process introduced in Section 4. First, Section 6.1 studies our application set selection process to select representative applications from TF Model Zoo. By comparing to the application set of prior benchmarks, we also demonstrate the advantages by the end of Section 6.1. Then, Section 6.2 evaluates several software-hardware co-designs on these selected applications. In the process of both application set selection and evaluating software-hardware co-designs, we conclude several observations on application characteristics and architecture design guidelines from these studies.

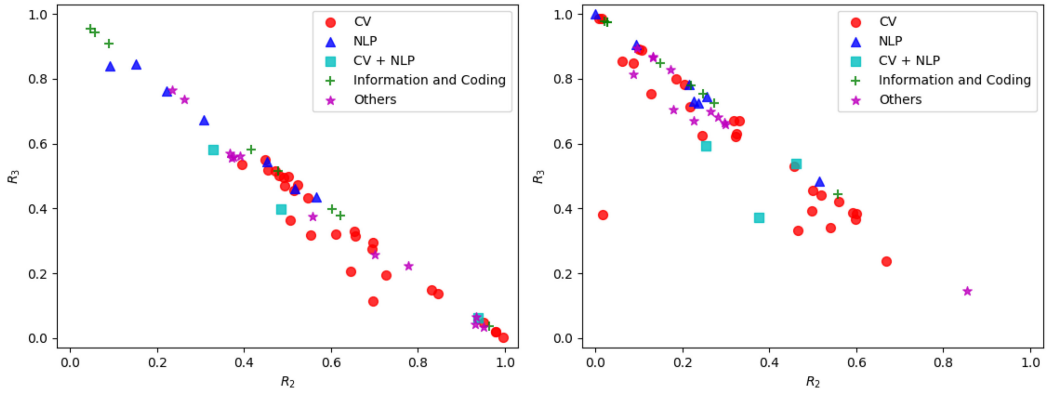
6.1 Application Selection from TensorFlow Model Zoo

As the first step of our analysis flow, we apply the operator-level analysis to most of the applications from the TensorFlow Model Zoo [21]. We first perform *extract operators* (Figure 4-❶)



(a) Operators from the pool. (b) Operator clusters.

Fig. 7. The distribution of operator features for all operators from the application candidate pool (TF Model Zoo) and the clustering results by running k-means.



(a) CPU (b) GPU

Fig. 8. The distribution of application features using CPU and GPU as baseline devices.

to all 224,563 operators from the application candidate pool. We then *extract operator features* (Figure 4-❷) and measure both the locality and the parallelism of the operators as defined in Section 4.1. The resulting distribution of operator features is shown in Figure 7(a). It labels different operator functionalities, including matrix multiplication (MatMul), convolution (Conv), pooling, reduction, element-wise, and other irregular operators (Others) where computations and memory accesses are dependent on input tensor values. Based on the performance feature distribution, we conduct *cluster operations* step (Figure 4-❸), which groups these operators into three clusters. We apply the k-means algorithm and obtain the cluster results shown as Figure 7(b). After this, we conduct an application-level analysis. Because most accelerator designs compare their performance to two kinds of general-purpose processors, CPU and GPU, we *profile* (Figure 4-❹) all applications from the application candidate pool on Intel Xeon E5-2680 CPU and NVIDIA Titan Xp GPU. To *extract application features* (Figure 4-❺), we use the three operator classes from previous operator analysis. The application performance feature in this case study is denoted as $\vec{f} = (R_1, R_2, R_3)$, where R_1, R_2, and R_3 represent the time breakdown of an application into three operator clusters. The performance feature distributions measured on CPU and GPU are shown as Figure 8(a) and

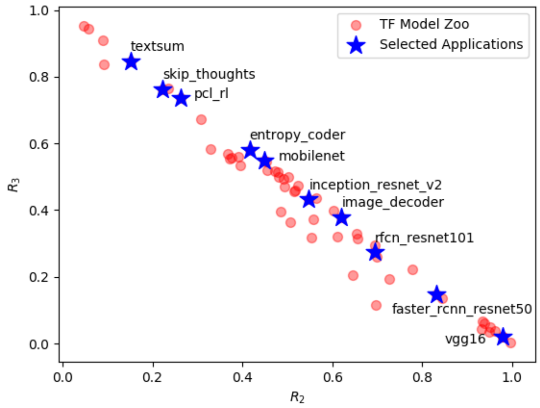


Fig. 9. The distribution of application features for selected applications out of the application candidate pool (TF Model Zoo).

Table 3. Brief Descriptions for 10 Applications Selected into the Original Application Set

Application	Description	Application Domain
textsum [46]	Text summarization	Natural Language Processing
skip_thoughts [35]	Sentence-to-vector encoder	Natural Language Processing
pcl_rl [38]	Reinforcement learning	Others
entropy_coder [31]	Image file compression	Information and Coding
mobilenet [28]	Image classification	Computer Vision
inception_resnet_v2 [27, 51]	Image classification	Computer Vision
image_decoder [53]	Image file decompression	Information and Coding
rfcn_resnet101 [20]	Object detection	Computer Vision
faster_rcnn_resnet50 [45]	Object detection	Computer Vision
vgg16 [49]	Image classification	Computer Vision

Figure 8(b). Since $R_1 + R_2 + R_3 = 1$, we plot two-dimensional scatter figures where x-axis stands for the R_2, y-axis stands for the R_3, and R_1 can be derived by $1 - R_2 - R_3$. Finally, we *select applications* (Figure 4-❻). Based on the distribution of the application features on CPU, we select 10 diverse and representative applications as the original application set by evenly sampling the application candidate pool. The distribution of these 10 applications is shown in Figure 9. Brief descriptions for these 10 applications can be found in Table 3.

Observations on the operator-level analysis. We classify operators into several categories to obtain observations on their architectural characteristics. The operator categories are designed to reflect operator functionalities or data access patterns. Among these operator categories, matrix multiplication (MatMul), convolution (Conv), and pooling attract intensive attention in many accelerator designs because of their importance in early NN models, such as VGG models [49]. The activation functions are also very common in NN models, such as ReLU operation in convolutional neural networks [27, 49, 51], and all of them are vector-like element-wise operations. Thus, we create a category as Element-wise in Figure 7(a) for all operators performing vector-like operations. We also create a separate category named as reduction for operators with reduction patterns, such as the *Softmax* and *Argmax* operations. Although these five categories cover most of the operators, we put the rest of operators into the last category as others.

We make several observations from the results of operator clustering (Figures 7(a)–7(b)). First, convolution and matrix multiplication operators are similar to each other, and most of them have good locality. Because of existing reduction patterns along some tensor dimensions, such as input channels in convolution operators, these two kinds of operators possess moderate parallelism. Second, all element-wise operators have identical parallelism while the computation intensity on each tensor element can vary significantly. Because of fully parallel scalar operations for all elements in element-wise operators, element-wise operators have the largest degree of parallelism (100%). Third, *operators with the same or similar functions can have very different performance features*, such as reduction and pooling operators. Clustering these operators by functions and designing hardware accordingly would result in bottleneck misprediction.

Architecture implications of operator clusters. The application feature in our work is directly associated with the breakdown of execution time spent on different operator clusters. Since we cluster operators according to their architecture features, i.e., locality and parallelism, operators in the same cluster could favor similar architecture designs. Specifically, operators in the first cluster have limited parallelism and moderate locality, whose execution time contributes to R_1. These operators could benefit from the locality optimizations while they can hardly benefit from more parallel processing elements (PEs). Operators from the second cluster have both moderate parallelism and locality, such as matrix multiplication and convolution, whose execution time contributes to R_2. These operators could benefit from parallel PE design, more computation resources, and optimizations on locality, such as the careful design of data-flow to exploit data reuse. Finally, operators from the third cluster can be fully parallelized whose execution time contributes to R_3. Increasing the number of PEs is helpful to exploit the parallelism while these operators will become bounded by memory bandwidth when the number of PEs is sufficient.

From the perspective of applications, application features indicate the distribution of execution time on these operator clusters. Thus, these application features help identify the application bottleneck from the perspective of operator clusters, which further provides architecture design guidelines. For example, an application with a large R_2 indicates that its bottleneck comes from operators in the second cluster, which could prefer architecture designs with more computation resources or larger on-chip memory. Similarly, an application with a large R_3 could prefer memory-centric architectures for higher effective memory bandwidth, because it is bounded by operators in the third cluster.

Observations on the application-level analysis. For the application-level analysis in Figures 8(a)–8(b), we summarize the following observations: First, *Conv*, *MatMul*, and *Element-wise* operators take up a majority of the application time in most of the applications, since most of the applications distribute near the line $R_2 + R_3 = 1$. Second, in contrast to CPU, GPU is more likely to be bounded by R_1, due to its more powerful computing resource and higher memory bandwidth. In addition, R_3 takes a larger percentage on GPU, indicating there are opportunities for GPU memory system optimization. Third, the consideration of application scenarios reveals additional trends. Both Figures 8(a) and Figure 8(b) label different application domains, including computer vision (CV), natural language processing (NLP), hybrid CV and NLP (CV+NLP), information and coding, and others. We classify applications into application domains according to the task of applications. Applications for traditional CV or NLP tasks are labeled as CV or NLP, respectively. The task of some applications is mixed by traditional CV or NLP tasks. For example, image captioning requires image understanding and caption generation, where image feature extraction is a CV task while the caption generation involving text summary is an NLP task. The application domain of these mixed tasks is denoted as *CV+NLP*. In addition to these traditional CV or NLP tasks, some tasks focus on the coding of information, such as file compression, decompression, and encryption. The application domain of these tasks related to information and coding is

labeled as *Information and Coding* although they could need domain knowledge related to CV or NLP when handling corresponding information, such as image compression. The domain labeled as *Others* includes the rest of the applications; most of the applications in this category belong to applications using reinforcement learning, such as robotics applications. *Most CV applications are bounded by operations from R_2 (mostly Conv and MatMul). On the contrary, most NLP applications are bounded by operations from the R_3 (mostly element-wise operators).* This indicates that memory-centric computing architectures can be helpful for these NLP applications.

The advantage of our methodology. We first demonstrate the advantage of the operator-level analysis by showing how misleading bottleneck diagnosis would occur if the aforementioned analysis is neglected. Without operator-level clustering, one has to extract the application feature with function-based operator clustering. For example, as described by Fathom, *Add* operators are clustered as the category *Elementwise Arithmetic*, but *transpose* operators are clustered as another category, *Data Movement*. However, when using our operator-level analysis, these two clusters should be in the same category (R_3 in our notation), since they have very similar architecture features in terms of locality and parallelism. There would be an issue in the case where R_3 is the application's bottleneck, but as part of R_3, neither *Elementwise Arithmetic* nor *Data Movement* individually shows as a bottleneck. *The bottleneck is then misunderstood. The described problem happens for 15 out of 57 models in the TF Model Zoo.* Taking application *video_prediction_stp* [18] for example, according to the performance feature defined in Fathom, it will show *Conv2D* as the bottleneck (taking 38% of total time). However, the elapsed time of operators from the R_3 cluster takes 52% of total time, making R_3-like operators (memory-intensive highly parallel operators) the actual bottleneck, not *Conv2D*. Instead of accelerating *Conv2D*, which would result in more computation resources or larger on-chip memory, our analysis recommends that the architecture should be designed with higher effective memory bandwidth, such as processing-in-memory architectures [15, 22, 30, 33] for R_3-like operators, because they take the majority of the elapsed time.

Second, our benchmark process selects more diverse and representative applications. Compared to Fathom, our method selects applications from a large application candidate pool based on extracted application features. Therefore, our analysis-based selection guarantees the diversity and representativeness of selected applications from the viewpoint of performance features. To understand the representativeness of Fathom applications on the TF Model Zoo, we go through the same application analysis process for applications (eight applications in total) from Fathom. The results measured on the CPU and the GPU are shown as Figures 10(a) and 10(b). Through comparisons, we can conclude that the application selection in Fathom is fairly good due to its similar distribution as TF Model Zoo. However, compared with Fathom, our benchmark selection in Figure 9 is more evenly distributed, making it more representative as a general benchmark. For example, the two selected benchmark applications in the orange circle in Figure 10(a) are too close to each other, making one of them redundant. In addition, some applications are underrepresented, such as applications in green circles in Figure 10(a) and 10(b). The applications from Fathom in these green circles are not sufficiently representative of the other applications with similar characteristics.

6.2 Benchmark Generation and Hardware Evaluation

We need the benchmark generation step (Section 4.2) after application selection to plug in the NN compression setup. This step is user-customized. According to our evaluation target, we generate our benchmark suite with three configurations: no compression (for GPU), quantized 16-bit fixed-point (for DianNao), and 16-bit fixed-point quantized and 90%/95% pruned (for Cambricon-X).

Finally, we conduct studies on evaluating several state-of-the-art software-hardware solutions in this section. In particular, we evaluate GPU (Titan Xp), Neurocube [33], DianNao [12], and Cambricon-X [64] with different model compression techniques. Among these hardware platforms

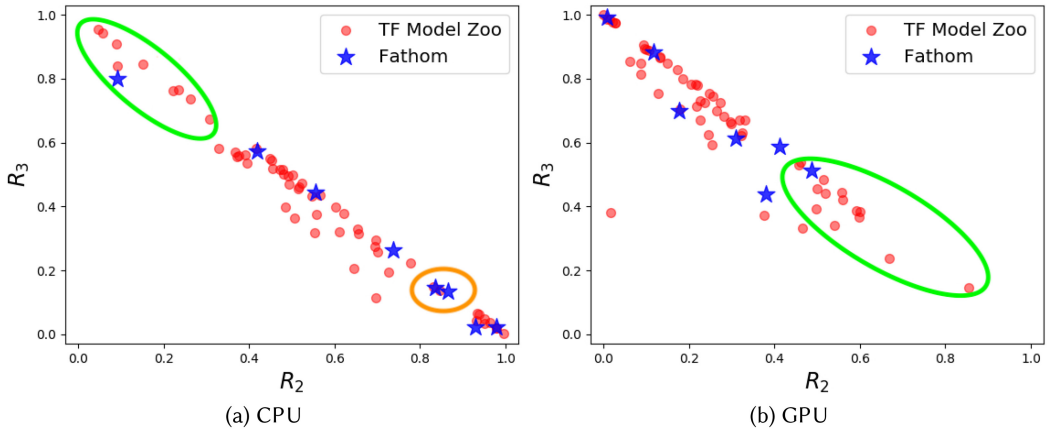


Fig. 10. The application feature distribution of applications from our application candidate pool (TF Model Zoo) compared to the distribution of applications in Fathom.

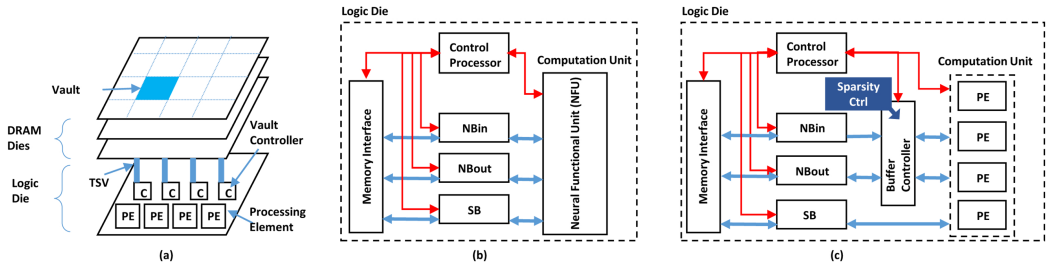


Fig. 11. The architecture overview of (a) Neurocube, (b) DianNao, and (c) Cambricon-X to distinguish key architecture differences among them: (a) an NDP design, (b) a compute-centric design, and (c) a compute-centric design with the support for sparsity.

we evaluated, GPU is a representative many-core processor exploiting the massive parallelism in tensor operators. Neurocube is an NDP design that exploits an internal memory bandwidth of memory cubes to accelerate memory-bound operators, while DianNao is a compute-centric accelerator design with on-chip computation and data movements tailored for NN applications. Both of these two platforms are designed for computing fixed-point arithmetic, which needs the help of NN model quantization from the software-level. Cambricon-X has a similar design as DianNao, except that its design is intensively customized to exploit the sparsity of NN models, which needs the help of NN model pruning. For the purpose of architecture comparison, Figure 11 shows the architecture of Neurocube, DianNao, and Cambricon-X.

Table 4 includes comparisons among these platforms in terms of power, performance, and area. These numbers are collected from official product specifications or their original papers. Due to the lack of detailed power models and area models on these platforms, such as the off-chip DRAM power and area data of DianNao and Cambricon-X, we only estimate the performance in our case studies. We use our system-level simulator to estimate the performance of these platforms compared to the CPU baseline implementation. According to the performance results presented in the original papers, we derive an analytical model based on the roofline model [57] to estimate the performance of each supported tensor operators on accelerators. Results on the GPU are profiled and measured from the execution on a real machine. We assume that these heterogeneous platforms

Table 4. The Description of Hardware Platforms

Platform	GPU	Nuerocube	DianNao	Cambricon-X
Peak Comp (GOPs)	12,100	132.4	482	528
Peak Mem (GB/s)	547.7	320	250	250
TDP (W)	250	21.5	0.485[1]	0.954[1]
Area (mm^2)	471	68	3.02[1]	6.38[1]
Tech Node (nm)	16	15	65	65

[1]These power and area data are from their original papers without considering the power consumption and area cost of DRAM dies.

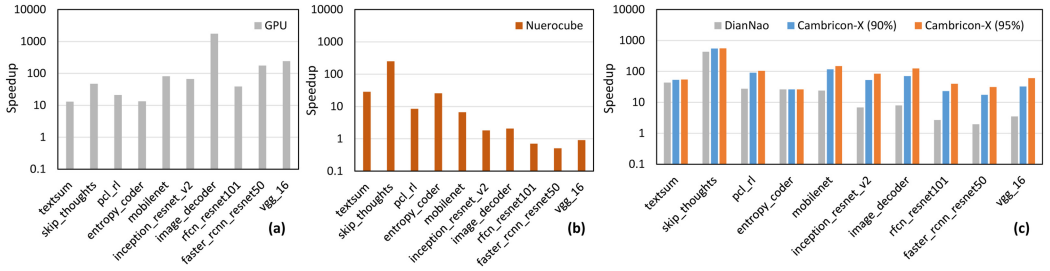


Fig. 12. The speedups over CPU baseline of applications on (a) GPU without any model compression, (b) Neurocube with models quantized into 16-bit fixed-point, (c) DianNao with models quantized into 16-bit fixed-point, Cambricon-X (90%) with models further pruned 90% weights, and Cambricon-X (95%) with models further pruned 95% weights.

are connected to a host CPU, Intel Xeon E5-2680 CPU, through PCIe, and any unsupported operator will be offloaded into the CPU for computation. The time of execution on the host CPU and data transfers triggered by offloading unsupported operators will be counted in the final elapsed time. However, we exclude the time used for transferring input data and model weights into these platforms, because transferring different batches of input data can overlap in real-world inference stage, and loading trained weights into these platforms is a one-time overhead.

Our simulation results are shown in Figure 12. The original application set is evaluated on the GPU, and results are shown in Figure 12(a). Figure 12(b) presents the performance results on Neurocube for applications quantized into 16-bit fixed-point data-type. Figure 12(c) presents the performance results for DianNao and Cambricon-X. Applications executed on DianNao are also quantized into 16-bit fixed-point. We evaluate two pruning strategies for applications executed on Cambricon-X, which prunes 90% and 95% weights of models, denoted as Cambricon-X (90%) and Cambricon-X (95%), respectively.

Insights from the result. By evaluating three representative accelerator designs with various compression configurations, we make the following observations from Figure 12: First, GPU can benefit these applications with a higher R_2 ratio in their performance features. These applications are usually computation-bound. Since applications on the x-axis are ordered by the increasing order of R_2, applications closer to the right direction along the x-axis spend more time in the second cluster operators, of which most are convolution and matrix multiplication operations. As shown in Figure 12(a), GPU obtains higher speedups on applications on the right side of the x-axis. Second, near-data computing architectures favor applications (mostly NLP related) with a higher R_3 ratio. Figure 12(b) shows that Neurocube achieves higher speedups on applications on the left side of the x-axis. Finally, *we found that weight pruning is less attractive for NLP applications than it is for CV applications.* Figure 12(c) shows the comparison of DianNao and Cambricon-X in terms

of performance benefits from pruning NN model weights, which reduces the computation and memory workloads of matrix multiplication and convolution operations. Comparing Cambricon-X (90%) to DianNao, Cambricon-X can achieve higher speedups than DianNao, which mainly benefits from the reduction of computation and memory workloads due to pruned models. Such speedups are more significant for computation-bound applications as opposed to memory-bound applications. The results of models with different sparsities, Cambricon-X (90%) and Cambricon-X (95%), indicate that pruning more weights can have slight benefits on memory-bound applications while significant benefits on computation-bound applications.

7 DISCUSSION

Software-hardware co-design in MLPerf. Neural network applications, especially the inference stage, benefit from the hardware-software co-design methodology. Thus, our work urges taking the whole software-hardware co-design solution as a benchmark object instead of benchmarking pure hardware designs by providing a fixed set of applications. The recently released MLPerf inference benchmark [44] includes an Open Division under the same motivation as our study, although they are a preliminary release and the rules of Open Division are immature. Compared to the immature rules in this preliminary release, our methodology provides a concrete interface to take the model compression techniques as the input and generate the compressed models as the output. Our work takes model compression techniques as the software optimizations in the end-to-end methodology, and our case studies reveal new insights for the impact of software optimizations on hardware designs. We still need to further refine stages in our methodology to embrace a larger scope of software solutions varying model architectures for the same prediction task, which is an important perspective of our future work.

Extensibility of our benchmark methodology. There are many configuration choices in our case study, which should be configured case-by-case. For example, we use locality and parallelism as operator features to capture various architecture designs. They are sufficient to indicate the overall architecture demand, such as compute-centric vs. memory-centric designs, because these two metrics are major considerations among different architecture designs to capture memory access patterns and computation intensity. However, these two metrics are not able to capture finer-grain locality and parallelism characteristics. When finer-grain operator characteristics are needed, the operator-level analysis phase needs to be adapted to new features, such as adding the reuse distance [17] to reflect the average distance between data reuses. Another example is adding new application features. We consider time breakdowns in application features because we think inter-operator parallelism is usually implemented in software frameworks, such as TensorFlow [2], for a higher flexibility of scheduling. However, when designing an accelerator taking the whole computation graphs as inputs and exploiting inter-operator parallelism at the hardware-level, the characteristics of computation graphs, such as the average of node degrees, can be added to the application features. In summary, configurations in our benchmark methodology are not fixed and some of them are tailored to our case study. We expect this benchmark methodology to be used by varying configurations case-by-case. Despite the change of configurations, such as adding reuse distance into the operator features, the key principles of our benchmark methodology, selecting applications quantitatively and benchmarking software-hardware co-designs, remain the same.

NNBench-X for new NN workloads. Because of the promising results from NN techniques, there are new algorithms developed for challenges in various applications. In these fast-growing algorithm studies, our benchmark methodology is feasible to characterize new NN workloads to provide insights for accelerator designs. For example, Bayesian neural networks (BNNs) [9, 39] attract attention due to their ability to deal with uncertainty during the estimation. In our benchmark methodology, we decompose BNN models into operators and go through the application

Table 5. The Performance of Image Classification Models on the CPU and FPGA of Amazon Web Service F1 Instance for Processing a Single Image with the Size 224 x 224 (Width x Height)

Models	bvlc_googlenet	resnet50_v1	squeezenet	vgg16
CPU (8 vCPUs)	103.92 ms	91.71 ms	42.41 ms	324.08 ms
FPGA (Xilinx Virtex UltraScale+)	1.57 ms	3.82 ms	1.43 ms	10.11 ms
Speedup	66.19	24.01	29.66	32.06

set selection flow to understand their characteristics. The only class of operators in BNN models different from existing NN models in the TF Model Zoo is sampling to generate the learned distribution of weights. Because each element in weights is sampled independently from its distribution in BNNs, these sampling operators have 100% parallelism. Thus, these operators belong to the third cluster in Figure 7(b) and their execution time contributes to R_3. BNN models with these sampling operators will have a larger R_3 than NN models with the same NN architecture. As a result, BNN models could have a larger demand on memory bandwidth, which is the same as most of the workloads bounded by R_3. If the performance of BNN models is significantly bounded by these sampling operators, efficient sampling implementations in the accelerator should also be considered [8]. These architectural implications will be helpful when designing new accelerators for BNN models.

NNBench-X for FPGA software-hardware co-designs. In addition to ASIC accelerators, FPGA is a promising platform to efficiently support various NN models. One promising software-hardware co-design among FPGA solutions is optimizing RTL templates for several widely used operators and developing a compilation flow to generate FPGA designs based on these templates on a given NN model represented by a computation graph. Similar to other software-hardware co-designs in ASICs, our benchmark methodology is able to be used to evaluate these FPGA solutions. To demonstrate how to use NNBench-X to evaluate FPGA solutions, we conduct a case study on Xilinx xDNN [60] in this section. We assume hardware designers are developing FPGA software-hardware co-designs for image classification applications. *In the first stage of NNBench-X*, our benchmark methodology takes a pool of image classification models as the input and selects representative ones from them. In this case study, we compose an application candidate pool including 10 image classification models, and the first stage of NNBench-X produces an application set with 4 of them (bvlc_googlenet [51], resnet50_v1 [27], squeezenet [29], and vgg16 [49]). *In the second stage of NNBench-X*, we pass these 4 models to the second stage to quantize models. At the end of these stages, quantized models are generated according to the quantization methods provided by xfDNN middleware, which is the software optimization part of the xDNN co-design [60]. *In the last stage of NNBench-X*, we run these quantized models generated by the second stage on FPGA hardware. Specifically, we run these quantized models on Amazon Web Service (AWS) f1.2xlarge instance, which has eight vCPUs and one Xilinx Virtex UltraScale+ FPGAs [5]. Experimental results are shown in Table 5. These results demonstrate a significant performance benefit of the FPGA co-design over the CPU baseline. In summary, we demonstrate that the benchmark methodology developed in this work is general to evaluate FPGA software-hardware co-designs through this case study. We hope our benchmark methodology can be helpful to improving future FPGA solutions for NN workloads.

8 CONCLUSION

In this article, we propose a novel end-to-end benchmarking method for NN accelerator designs. To select the most representative NN applications and evaluate software-hardware co-designs, our benchmark method is composed of three stages: application set selection, benchmark suite

generation, and hardware evaluation. The application set selection stage selects representative NN applications according to quantitative metrics to ensure the diversity of the benchmark suite. The benchmark suite generation and hardware evaluation stages refine the selected applications according to user-provided model compression techniques and evaluate the compressed models on accelerator designs. We conduct a study of benchmarking several state-of-the-art NN accelerator designs to demonstrate the usage of our benchmark method. We analyze applications from the TensorFlow Model Zoo and observe that applications from the same application domains have similar bottlenecks. Moreover, we evaluate several state-of-the-art software-hardware co-design solutions, including hardware designs for quantized and pruned NN models. From our case studies, we observe that computation-centric and memory-centric architectures can have different benefits for different application domains. Also, we find that pruning NN models provides little benefit to memory-bound applications. Through our case studies and observations, we are convinced that our benchmark method is practical and feasible to provide insightful guidance to NN accelerator designs.

REFERENCES

[1] MLPerf. 2018. MLPerf. Retrieved from https://mlperf.org/.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI'18)*, Vol. 16. 265–283.

[3] Robert Adolf, Saketh Rama, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2016. Fathom: Reference workloads for modern deep learning methods. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'16)*. IEEE, 1–10.

[4] 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 43rd International Symposium on Computer Architecture (ISCA'16)*. IEEE, 1–13.

[5] Amazon. 2020. Amazon EC2 F1 Instances. Retrieved from https://aws.amazon.com/ec2/instance-types/f1/.

[6] Bahar Asgari, Ramyad Hadidi, Hyesoon Kim, and Sudhakar Yalamanchili. 2019. Eridanus: Efficiently running inference of DNNs using systolic arrays. *IEEE Micro* 39, 5 (2019), 46–54.

[7] Baidu. 2018. DeepBench. Retrieved from https://github.com/baidu-research/DeepBench.

[8] Subho S. Banerjee, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2019. AcMC 2: Accelerating Markov chain Monte Carlo algorithms for probabilistic models. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. 515–528.

[9] Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. 2015. Weight uncertainty in neural networks. *arXiv preprint arXiv:1505.05424* (2015).

[10] Geoffrey W. Burr, Robert M. Shelby, Severin Sidler, Carmelo Di Nolfo, Junwoo Jang, Irem Boybat, Rohit S. Shenoy, Pritish Narayanan, Kumar Virwani, Emanuele U. Giacometti, Bulent N. Kurdi, and Hyunsang Hwang. 2015. Experimental demonstration and tolerancing of a large-scale neural network (165,000 synapses) using phase-change memory as the synaptic weight element. *IEEE Trans. Electron Dev.* 62, 11 (2015), 3498–3507.

[11] Tianshi Chen, Yunji Chen, Marc Duranton, Qi Guo, Atif Hashmi, Mikko Lipasti, Andrew Nere, Shi Qiu, Michele Sebag, and Olivier Temam. 2012. BenchNN: On the broad potential application scope of hardware neural network accelerators. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'12)*. IEEE, 36–45.

[12] 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 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 269–284.

[13] 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 47th IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 609–622.

[14] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2017. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE J. Solid-State Circ.* 52, 1 (2017), 127–138.

[15] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. Prime: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory. In *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 27–39.

[16] PyTorch Core team. 2017. PyTorch. Retrieved from http://pytorch.org/.

[17] Chen Ding and Yutao Zhong. 2003. Predicting whole-program locality through reuse distance analysis. In *ACM Sigplan Not.*, Vol. 38. ACM, 245–257.

[18] Chelsea Finn, Ian Goodfellow, and Sergey Levine. 2016. Unsupervised learning for physical interaction through video prediction. In *Proceedings of the International Conference on Advances in Neural Information Processing Systems*. 64–72.

[19] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi et al. 2018. A configurable cloud-scale DNN processor for real-time AI. In *Proceedings of the ACM/IEEE 45th International Symposium on Computer Architecture (ISCA'18)*. IEEE, 1–14.

[20] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 580–587.

[21] Google. 2018. TensorFlow Models. Retrieved from https://github.com/tensorflow/models.

[22] Peng Gu, Xinfeng Xie, Yufei Ding, Guoyang Chen, Weifeng Zhang, Dimin Niu, and Yuan Xie. 2020. iPIM: Programmable in-memory image processing accelerator using near-bank architecture. In *Proceedings of the ACM/IEEE 47th International Symposium on Computer Architecture (ISCA'20)*. IEEE, 804–817.

[23] Kaiyuan Guo, Song Han, Song Yao, Yu Wang, Yuan Xie, and Huazhong Yang. 2017. Software-hardware codesign for efficient neural network acceleration. *IEEE Micro* 37, 2 (2017), 18–25.

[24] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 243–254.

[25] Song Han, Huizi Mao, and William J. Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[26] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. In *Proceedings of the International Conference on Advances in Neural Information Processing Systems*. 1135–1143.

[27] 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.

[28] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[29] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).

[30] Yu Ji, Youyang Zhang, Xinfeng Xie, Shuangchen Li, Peiqi Wang, Xing Hu, Youhui Zhang, and Yuan Xie. 2019. FPSA: A full system stack solution for reconfigurable ReRAM-based NN accelerator architecture. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. 733–747.

[31] Nick Johnston, Damien Vincent, David Minnen, Michele Covell, Saurabh Singh, Troy Chinen, Sung Jin Hwang, Joel Shor, and George Toderici. 2017. Improved lossy image compression with priming and spatially adaptive bit rates for recurrent networks. *arXiv preprint arXiv:1703.10114* (2017).

[32] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Hurt Dan, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th International Symposium on Computer Architecture*. ACM, 1–12.

[33] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. 2016. Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory. In *Proceedings of the ACM/IEEE 43rd International Symposium on Computer Architecture (ISCA'16)*. IEEE, 380–392.

[34] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. 2015. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530* (2015).

[35] Ryan Kiros, Yukun Zhu, Ruslan R. Salakhutdinov, Richard Zemel, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. 2015. Skip-thought vectors. In *Proceedings of the International Conference on Advances in Neural Information Processing Systems*. 3294–3302.

[36] H. T. Kung, Bradley McDanel, and Sai Qian Zhang. 2019. Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. 821–834.

[37] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. 2019. Understanding reuse, performance, and hardware cost of DNN dataflow: A data-centric approach. In *Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture*. 754–768.

[38] Ofir Nachum, Mohammad Norouzi, Kelvin Xu, and Dale Schuurmans. 2017. Bridging the gap between value and policy based reinforcement learning. In *Proceedings of the International Conference on Advances in Neural Information Processing Systems*. 2772–2782.

[39] Radford M. Neal. 2012. *Bayesian Learning for Neural Networks*. Vol. 118. Springer Science & Business Media.

[40] Nvidia. 2017. cuDNN. Retrieved from https://developer.nvidia.com/cudnn.

[41] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. 2016. Going deeper with embedded FPGA platform for convolutional neural network. In *Proceedings of the ACM/SIGDA International Symposium on Field-programmable Gate Arrays*. ACM, 26–35.

[42] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. XNOR-Net: Imagenet classification using binary convolutional neural networks. In *Proceedings of the European Conference on Computer Vision*. Springer, 525–542.

[43] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. 2016. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 267–278.

[44] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou. 2020. MLPerf inference benchmark. In *Proceedings of the ACM/IEEE 47th International Symposium on Computer Architecture (ISCA'20)*. 446–459.

[45] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster R-CNN: Towards real-time object detection with region proposal networks. In *Proceedings of the International Conference on Advances in Neural Information Processing Systems*. 91–99.

[46] Alexander M. Rush, Sumit Chopra, and Jason Weston. 2015. A neural attention model for abstractive sentence summarization. *arXiv preprint arXiv:1509.00685* (2015).

[47] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *Proceedings of the 49th IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE, 1–12.

[48] Shaohuai Shi, Qiang Wang, Pengfei Xu, and Xiaowen Chu. 2016. Benchmarking state-of-the-art deep learning software tools. In *Proceedings of the 7th International Conference on Cloud Computing and Big Data (CCBD'16)*. IEEE, 99–104.

[49] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[50] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to sequence learning with neural networks. In *Proceedings of the International Conference on Advances in Neural Information Processing Systems*. 3104–3112.

[51] 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.

[52] Jin-Hua Tao, Zi-Dong Du, Qi Guo, Hui-Ying Lan, Lei Zhang, Sheng-Yuan Zhou, Cong Liu, Hai-Feng Liu, Shan Tang, Allen Rush, Willian Chen, Shao-Li Liu, Yun-Ji Chen, and Tian-Shi Chen. 2017. BENCHIP: Benchmarking intelligence processors. *arXiv preprint arXiv:1710.08315* (2017).

[53] George Toderici, Damien Vincent, Nick Johnston, Sung Jin Hwang, David Minnen, Joel Shor, and Michele Covell. 2016. Full resolution image compression with recurrent neural networks. *arXiv preprint* (2016).

[54] Peiqi Wang, Dongsheng Wang, Yu Ji, Xinfeng Xie, Haoxuan Song, XuXin Liu, Yongqiang Lyu, and Yuan Xie. 2019. QGAN: Quantized generative adversarial networks. *arXiv preprint arXiv:1901.08263* (2019).

[55] Peiqi Wang, Xinfeng Xie, Lei Deng, Guoqi Li, Dongsheng Wang, and Yuan Xie. 2018. HitNet: Hybrid ternary recurrent neural network. In *Proceedings of the International Conference on Advances in Neural Information Processing Systems*. 604–614.

[56] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning structured sparsity in deep neural networks. In *Proceedings of the International Conference on Advances in Neural Information Processing Systems*. 2074–2082.

[57] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[58] Xinfeng Xie, Dayou Du, Qian Li, Yun Liang, Wai Teng Tang, Zhong Liang Ong, Mian Lu, Huynh Phung Huynh, and Rick Siow Mong Goh. 2017. Exploiting sparsity to accelerate fully connected layers of CNN-based applications on mobile SoCs. *ACM Trans. Embedd. Comput. Syst.* 17, 2 (2017), 1–25.

[59] Xinfeng Xie, Xing Hu, Peng Gu, Shuangchen Li, Yu Ji, and Yuan Xie. 2019. NNBench-X: Benchmarking and understanding neural network workloads for accelerator designs. *IEEE Comput. Archit. Lett.* 18, 1 (2019), 38–42.

[60] Xilinx. 2020. Xilinx xDNN Processing Engine. Retrieved from https://github.com/Xilinx/ml-suite.

[61] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. 2017. Scalpel: Customizing DNN pruning to the underlying hardware parallelism. In *Proceedings of the 44th International Symposium on Computer Architecture*. ACM, 548–560.

[62] 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*. ACM, 161–170.

[63] Chen Zhang, Guangyu Sun, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2018. Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks. *IEEE Trans. Comput.-aided Des. Integ. Circ. Syst.* 38, 11 (2018), 2072–2085.

[64] 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 49th IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE, 1–12.

[65] 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 (ICCAD'18)*. IEEE, 1–8.

Received October 2019; revised July 2020; accepted August 2020