NNBench-X: Benchmarking and Understanding Neural Network Workloads for Accelerator Designs

Xinfeng Xie, Xing Hu, Peng Gu, Shuangchen Li, Yu Ji, and Yuan Xie

Abstract—The tremendous impact of deep learning algorithms over a wide range of application domains has encouraged a surge of neural network (NN) accelerator research. An evolving benchmark suite and its associated benchmark method are needed to incorporate emerging NN models and characterze NN workloads. In this paper, we propose a novel approach to understand the performance characteristic of NN workloads for accelerator designs. Our approach takes as input an application candidate pool and conducts an operator-level analysis and application-level analysis to understand the performance characteristics of both basic tensor primitives and whole applications. We conduct a case study on the TensorFlow model zoo by using this proposed characterization method. We find that tensor operators with the same functionality can have very different performance characteristics under different input sizes, while operators with different functionality can have similar characteristics. Additionally, we observe that without operator-level analysis, the application bottleneck is mischaracterized for 15 out of 57 models from the TensorFlow model zoo. Overall, our characterization method helps users select representative applications out of the large pool of possible applications, while providing insightful guidelines for the design of NN accelerators.

Index Terms—Neural network, workload characterization, benchmark

1 Introduction

Neural Network (NN) algorithms have demonstrated greater accuracy than other machine learning algorithms in a wide range of application domains, including computer vision (CV) [1], [2], [3], [4], [5] and natural language processing (NLP) [6], [7]. 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 supporting the development 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 functionality and to improve accuracy, while ASIC development requires a long design and manufacturing period. Any accelerator design would thus risk 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 the NN accelerator design.

Existing benchmark suites, such as Fathom [8], BenchIP [9], MLPerf [10], and AI Matrix [11], serve the purpose of selecting representative applications to guide the hardware design. However, these benchmarks may not pay enough attention to the following two problems. First, the applications are empirically selected to cover various application domains instead of employing a consistent, quantitative approach. Second, due to the lack of a quantitative guideline, existing benchmarks are difficult to update or customize. With the rapid development of application domains, existing benchmarks must undergo an opaque and difficult process to determine whether new applications sufficiently differ from those in the existing suite, and whether the existing suite still accurately represents the state of the application domain.

In this paper, we address these problems by developing a bottom-up approach to select representative NN applications out of a large application candidate pool. We propose a novel characterization method, based on operator-level and application-level analysis, to quantify the performance features of both tensor primitives and end-to-end applications. Based on this method, we can select representative and diverse benchmarks out of the pool and update the pool promptly by adding emerging applications and removing obsolete ones. Our case study on a comprehensive set of NN applications, the TensorFlow (TF) model zoo [12], reveals that:

- Tensor operators with the same functionality can exhibit different performance features under different input tensor shapes while operators can have similar performance features although they are different functionally.
- Without the help of operator-level analysis, which clusters operators by their performance features instead of functionality, the application bottleneck could be incorrectly characterized.

2 Characterization Method

In this section, we first provide an overview of our characterization process, then describe operator-level analysis and application-level analysis in details.

2.1 Overview

The proposed characterization process consists of two stages, operator-level and application-level analysis, as shown in Fig. 1a. Because tensor operators are the primitives of NN applications, operator-level analysis is conducted first, before application-level analysis. Two important metrics, locality and parallelism, are selected to represent the range of operator behavior and are used to cluster the analyzed operators into several groups. After this operator-level clustering, application-level analysis is performed as the second stage. Applications are first profiled on baseline architectures before they are quantified by time breakdown on the different operator clusters. After obtaining these application features, we conduct similarity analysis for all applications. Finally, a benchmark suite 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 like Fathom [8], 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 in Section 3.3.

2.2 Operator-Level Analysis

As shown in Fig. 1a, we perform operator-level analysis in the first stage to extract operator features and cluster operators based on these 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. *As a highlight, we use platform-independent metrics as the operator feature to improve the generality of the generated benchmark suite.* Finally, we cluster these operators. Two metrics are employed to represent the architectural behavior of an operator.

- *The authors are with the University of California, Santa Barbara, Santa Barbara, CA 93106. E-mail: {xinfeng, xinghu, yuanxie}@ucsb.edu, peng_gu@umail.ucsb.edu, shuangchenli@ece.ucsb.edu, maple.jiyu@hotmail.com.*

Manuscript received 7 Nov. 2018, revised 3 Jan. 2019, accepted 29 Jan. 2019, Date of publication 6 Feb. 2019; date of current version 19 Mar. 2019.
(Corresponding author: Xinfeng Xie.)
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/LCA.2019.2898196

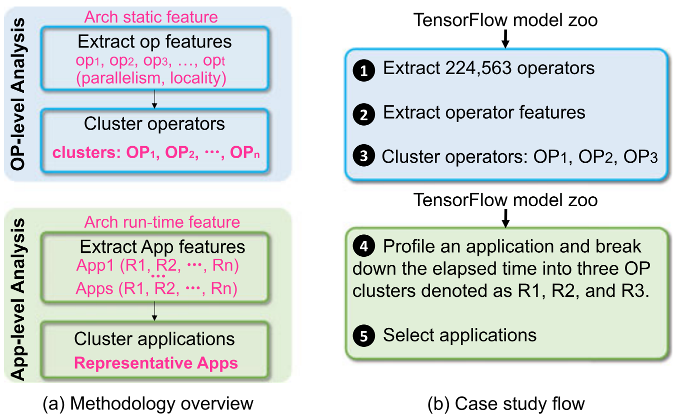


Fig. 1. Proposed characterization process.

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. This metric reflects both temporal locality and spatial locality in an ideal memory system where a cache hit will occur if the same location was accessed before. Lower values of this metric indicate better locality for the operator.

Parallelism. This metric is defined as the ratio of scalar arithmetic operations which can be executed in parallel, assuming sufficient hardware resources. 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 adding 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 percent. 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 N scalar-max operations can be executed in parallel given sufficient hardware resources. As a result, the parallelism for a tensor *Max* operator is $\frac{1}{logN}$.

We define these two metrics for the operator-level analysis to reflect 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 [13], 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 which operators are similar from the viewpoint of architecture designs.* After obtaining operator performance features in the aforementioned metrics, we can group operators into several clusters according to these features.

2.3 Application-Level Analysis

As shown in Fig. 1a, we perform application-level analysis in the second stage to extract application features and select applications based on these features. We define the performance feature of an application as the time breakdown on the different operator clusters obtained from 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, either 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 about 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.

3 Case Study: TensorFlow Model Zoo

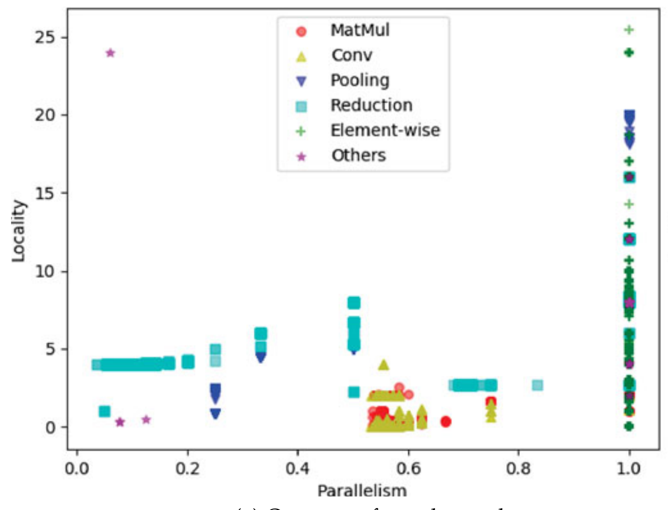
To demonstrate the usage and effectiveness of our characterization method, we introduce a case study where we characterize general NN inference applications. To this end, we use the TensorFlow Model Zoo [12] (with 57 NN models and 224,563 operators) as the application candidate pool, and hence conduct an extensive study. The TF Model Zoo includes a majority of the state-of-the-art research models, and it keeps track of numerous updated applications from the machine learning community. This section follows the characterization process introduced in Section 2. In addition, we conclude with several observations and architecture design guidelines from these case studies to show the advantages of our methodology.

3.1 Characterization Process

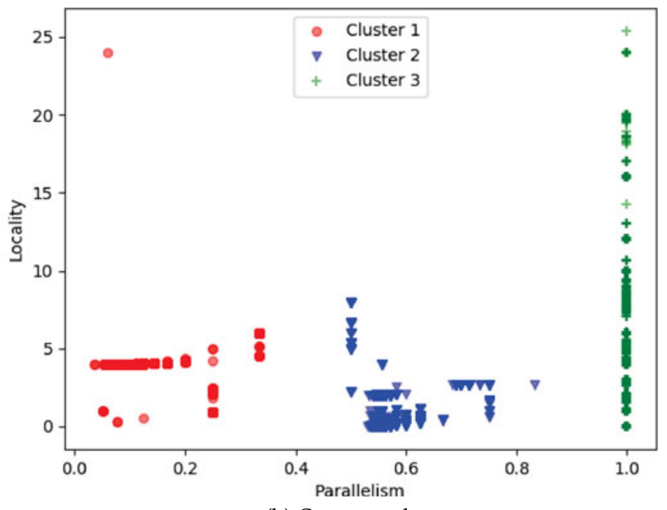
The characterization process of this case study is shown in Fig. 1b. We apply *operator-level analysis* to all applications from the TensorFlow Model Zoo [12]. We first *extract all 224,563 operators* (Fig. 1b-❶) from 57 models in the application candidate pool. Next, we *extract operator features* (Fig. 1b-❷) by measuring the locality and parallelism as defined in Section 2.2 for each operator. The result distribution of operator features is shown in Fig. 2a. In the last step of *operator-level analysis*, we *cluster operators* (Fig. 1b-❸) based on extracted operator features. We use the k-means method in this case study, resulting in three operator clusters which are shown in Fig. 2b. After this, we conduct the *application-level analysis*. Because most accelerator designs compare their performance to two kinds of general purpose processors, CPU and GPU, we *profile and break down the elapsed time into three operator clusters* (Fig. 1b-❹) for all applications from the application candidate pool on an Intel Xeon E5-2680 CPU and an NVIDIA Titan Xp GPU, respectively. 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 Figs. 4a and 4b, where the x-axis represents R_2, the y-axis represents R_3, and $R_1 = 1 - R_2 - R_3$ because $R_1 + R_2 + R_3 = 1$. Finally, we *select applications* (Fig. 1b-❺) based on the distribution of application features, selecting the CPU as the baseline architecture in this example. Because the R_1 component is negligible for most applications, we select ten applications with features evenly distributed along the line $R_2 + R_3 = 1$ to come up with the benchmark suite, NNBench-X. The distribution of these ten applications are shown in Fig. 3. Brief descriptions for these ten applications can be found in Table 1.

3.2 Observations and Insights

Observations on the Operator-Level Analysis. We make several observations from the results of operator clustering (Figs. 2a and 2b). 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



(a) Operators from the pool.



(b) Operator clusters.

Fig. 2. The distribution of operator features.

(100 percent). 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. Therefore, application features indicate the distribution of execution time on these operator clusters, which helps identify the application bottleneck from the perspective of operator clusters, and 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 Figs. 4a and 4b, 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 of Figs. 4a and 4b label different application domains including computer vision, natural language processing, hybrid CV and NLP (CV+NLP), information and coding, and others. Among these application domains, the domain *CV+NLP* includes applications requiring both CV and NLP algorithms, such as image captioning where CV network architectures extract image features and NLP network architectures generate the final output caption. The domain labeled as *Information and Coding* includes applications using neural networks for traditional information and coding tasks, such as file compression and decompression. The domain labeled as *Others* includes the rest of the applications, most of applications in this category belong to applications using the 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.

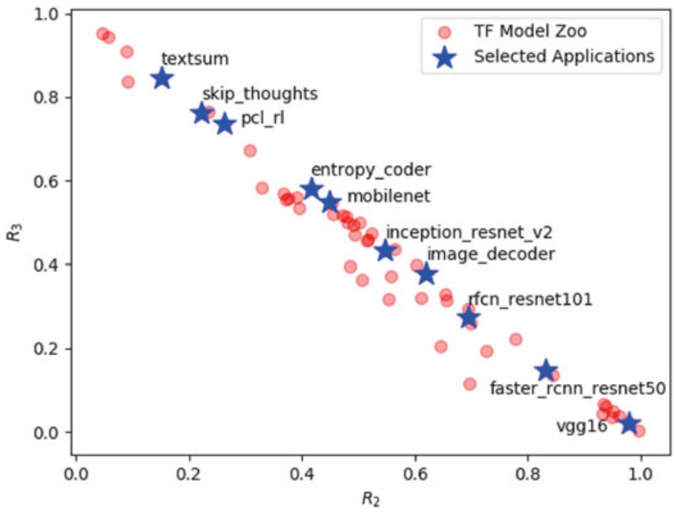


Fig. 3. The feature distribution of selected applications.

3.3 The Advantage of Our Methodology

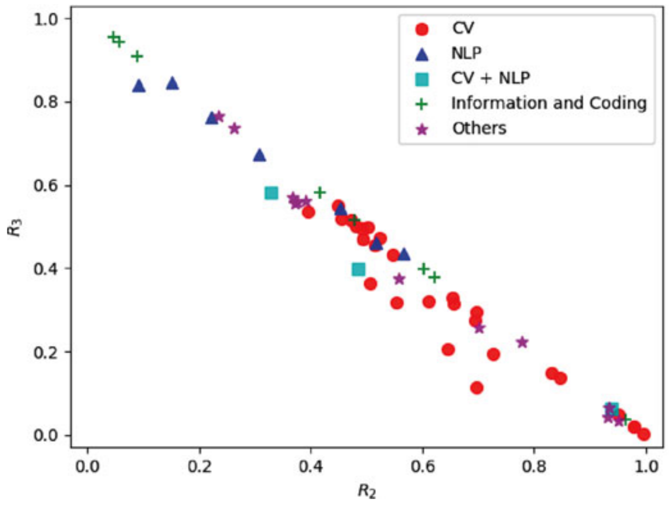
We first demonstrate the advantage of the *operator-level analysis* by showing how misleading bottleneck diagnosis would occur if the aforementioned analysis is neglected. Without *operator-level* clustering, one has to extract the application feature with function-based operator clustering. For example, 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 percent of total time). However, the elapsed time of operators from the R_3 cluster takes 52 percent of total time, making

TABLE 1
Brief Descriptions for Ten Applications in NNBench-X

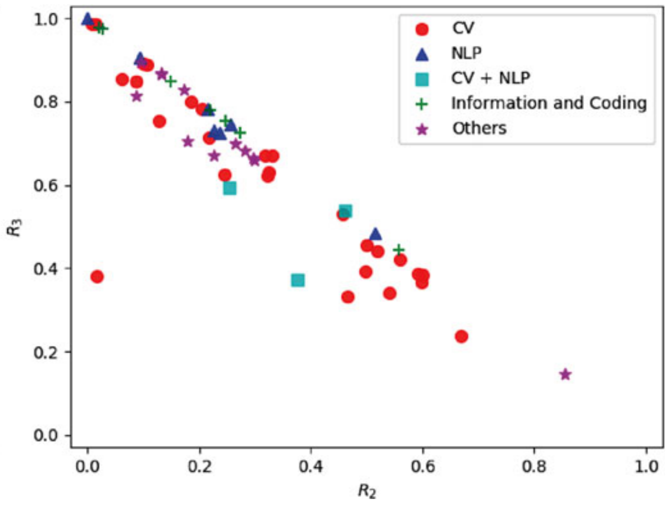
Application	Description
textsum [6]	Text summarization
skip_thoughts [7]	Sentence-to-vector encoder
pcl_rl [14]	Reinforcement learning
entropy_coder [15]	Image file compression
mobilenet [1]	Image classification
inception_resnet_v2 [2], [3]	Image classification
image_decoder [16]	Image file decompression
rfcn_resnet101 [4]	Object detection
faster_rcnn_resnet50 [5]	Object detection
vgg16 [17]	Image classification

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 [19], [20], [21], [22], 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 (8 applications in total) from Fathom. The results measured on the CPU and the GPU are shown as Figs. 5a and 5b. 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 Fig. 3 is more evenly distributed, making it more representative as a general benchmark. For example, the two selected benchmark applications in the orange circle in Fig. 5a are too close to each other, making one of them redundant. In addition, some applications are underrepresented, such as applications in green circles in Figs. 5a and 5b. The applications from Fathom in these green circles are not sufficiently representative of the other applications with similar characteristics.

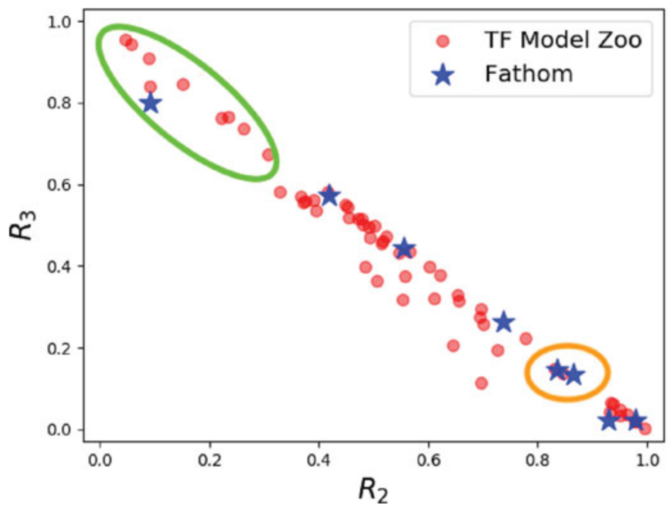


(a) CPU

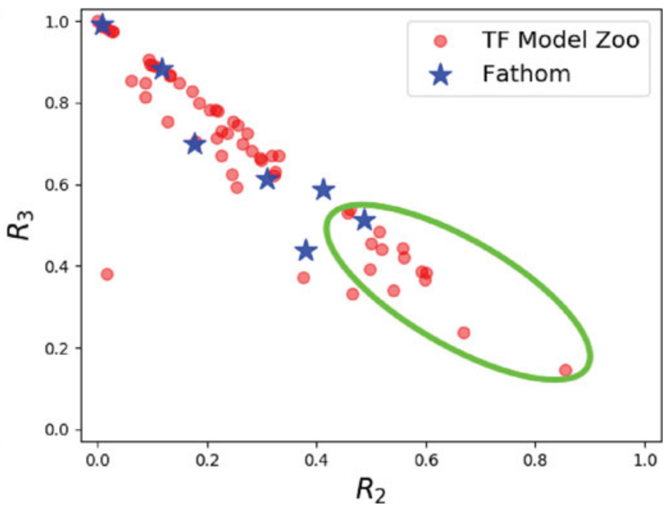


(b) GPU

Fig. 4. The distribution of application features.



(a) CPU



(b) GPU

Fig. 5. The distribution of application features compared to Fathom.

4 Conclusion

In this paper, we propose a novel benchmarking method to understand the performance characteristics of NN workloads. We conduct an operator-level analysis to extract architecture-independent performance features, and to then cluster similar operators together. Our application-level analysis then breaks down the time spent by each application into three operator clusters, identifying application bottlenecks and providing accelerator design guidance. We have two observations from the case study on the TensorFlow Model Zoo: 1) operators with the same functionality can have very different parallelism and locality features while operators with different functionality can have similar ones, and 2) operator-level analysis helps us identify and understand the performance bottleneck of applications. Finally, based on our characterization results, we select ten representative applications out of the TensorFlow model zoo into the benchmark suite, NNBench-X, which serves as a benchmark for general NN processor designs.

Acknowledgments

This work was supported in part by US National Science Foundation 1500848/172544/1730309 and by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation program sponsored by DARPA.

References

[1] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “MobileNets: Efficient convolutional neural networks for mobile vision applications,” *arXiv: 1704.04861*, 2017.

[2] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2015, pp. 1–9.

[3] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[4] R. Girshick, J. Donahue, T. Darrell, and J. Malik, “Rich feature hierarchies for accurate object detection and semantic segmentation,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2014, pp. 580–587.

[5] S. Ren, K. He, R. Girshick, and J. Sun, “Faster R-CNN: Towards real-time object detection with region proposal networks,” in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2015, pp. 91–99.

[6] A. M. Rush, S. Chopra, and J. Weston, “A neural attention model for abstractive sentence summarization,” *arXiv:1509.00685*, 2015.

[7] R. Kiros, Y. Zhu, R. R. Salakhutdinov, R. Zemel, R. Urtasun, A. Torralba, and S. Fidler, “Skip-thought vectors,” in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2015, pp. 3294–3302.

[8] R. Adolf, S. Rama, B. Reagen, G.-Y. Wei, and D. Brooks, “Fathom: Reference workloads for modern deep learning methods,” in *Proc. IEEE Int. Symp. Workload Characterization*, 2016, pp. 1–10.

[9] J.-H. Tao, Z.-D. Du, Q. Guo, H.-Y. Lan, L. Zhang, S.-Y. Zhou, L.-J. Xu, C. Liu, H.-F. Liu, S. Tang, et al., “BENCHIP: Benchmarking intelligence processors,” *J. Comput. Sci. Technol.*, vol. 33, no. 1, pp. 1–23, 2018.

[10] MLPerf, “MLPerf Benchmark Suite,” 2018. [Online]. Available: https://mlperf.org/

[11] W. Wei, L. Xu, L. Jin, W. Zhang, and T. Zhang, “Ai matrix-synthetic benchmarks for DNN,” *arXiv: 1812.00886*, 2018.

[12] Google, “TensorFlow models,” 2018. [Online]. Available: https://github.com/tensorflow/models

[13] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, “Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks,” *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.

[14] O. Nachum, M. Norouzi, K. Xu, and D. Schuurmans, “Bridging the gap between value and policy based reinforcement learning,” in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 2772–2782.

[15] N. Johnston, D. Vincent, D. Minnen, M. Covell, S. Singh, T. Chinen, S. J. Hwang, J. Shor, and G. Toderici, “Improved lossy image compression with priming and spatially adaptive bit rates for recurrent networks,” in *Proc. IEEE Conf. Computer Vision and Pattern Recognition*,, pp. 4385–4393, 2018.

[16] G. Toderici, D. Vincent, N. Johnston, S. J. Hwang, D. Minnen, J. Shor, and M. Covell, “Full resolution image compression with recurrent neural networks,” in *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, pp. 5306–5314, 2017.

[17] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *arXiv:1409.1556*, 2014.

[18] C. Finn, I. Goodfellow, and S. Levine, “Unsupervised learning for physical interaction through video prediction,” in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2016, pp. 64–72.

[19] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, “PRIME: A novel processing-in-memory architecture for neural network computation in reram-based main memory,” *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 27–39, 2016.

[20] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, “Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory,” in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit.*, 2016, pp. 380–392.

[21] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, “DRISA: A dram-based reconfigurable in-situ accelerator,” in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2017, pp. 288–301.

[22] S. Li, A. O. Glova, X. Hu, P. Gu, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, “SCOPE: A stochastic computing engine for dram-based in-situ accelerator,” in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2018, pp. 696–709.