# Enabling Real-time DNN Switching via Weight-Sharing

Jianming Tong, Yangyu Chen, Yue Pan, Abhimanyu Bambhaniya, Alind Khare, Taekyung Heo, Alexey Tumanov, Tushar Krishna
Georgia Institute of Technology, Atlanta, USA
{jianming.tong, yangyuchen, ypan331, abambhaniya3, akhare39, taekyung, atumanov3}@gatech.edu, tushar@ece.gatech.edu

## ABSTRACT

There is a growing rise of applications that need to support a library of models with diverse latency-accuracy trade-offs on a Pareto frontier, especially in the health-care domain. This work presents an end-to-end system for training and serving weight-sharing models. On the training end, we leverage recent research in creating a family of models on the latency-accuracy Pareto frontier that share weights, reducing the total number of unique parameters. On the serving (inference end), we propose a novel accelerator ***FastSwitch*** that extracts weight reuse across different models, thereby providing fast real-time switching between different models.

## 1. INTRODUCTION

A significant fraction of machine learning (ML) and deep learning (DL) literature has so far been focused on optimizing for accuracy as the primary objective or minimizing accuracy loss while optimizing for other efficiency metrics, such as forward pass inference latency and various measures of cost (energy, power, area, dollars). ML models individually architected, trained, tuned, and accelerated can then be regarded as *individual* points in a multi-dimensional tradeoff space of accuracy and latency. Far less attention has been given to mechanisms and policies to navigate this tradeoff space efficiently. The importance of making such tradeoffs *dynamically* can be exemplified by prediction tasks targeting clinical environments, such as Intensive Care Units (ICU) or Emergency Room (ER) triage. In such cases real-time model serving is equally if not more important than accuracy, because ICU patient care is simultaneously more urgent and more expensive. Clinical decisions and the timeliness of those decisions affect both the quality of care for patients and the cost of care. In some cases (e.g., prediction of septic shock, cardiac arrest, or respiratory distress), it can be a matter of life and death. To compound the challenge, clinical environments typically feature multiple tiers of inference deployment, including bedside compute, limited on-site cluster resources, and HIPAA-compliant cloud resources. Developing individual models for each of these latency tiers is cumbersome, costly, and redundant. Even for the exact same deployment target and for the exact same latency-sensitive application, the dynamics of inference query ingest, variable network bandwidth, and the variability in the volume of patients as a function of time — all call for different points of optimality to be accessible for inference at any given time. Concretely, under low volume load and well behaved network conditions, costlier and more accurate models can be served. With load spikes (e.g., ER patient influx, rapidly evolving medical phenomena), lighter-weight faster models are best suited to keep up with real-time demand.

The rise of applications that can benefit from variable latency-accuracy choices is luckily met with a nascent research effort [1, 8] to co-train large *model families* that simultaneously target a large span of deployment scenarios. OFA [1] initially proposed a mechanism to simultaneously train $\approx 10^{19}$ Convolutional Neural Networks (CNNs) with a progressive shrinking technique that amortized the cost of training for all possible latency/accuracy choices. Importantly, it decoupled training this *SuperNet* structure from extracting a Pareto optimal frontier for specific target deployment. CompOFA [8] subsequently improved on the efficiency of training CNN *SuperNet*, by reducing the search space without sacrificing accuracy, Pareto optimality, or the density of the resulting Pareto Frontier. Fundamentally, this work lends us the ML mechanisms for developing neural network (NN) constructs that capture the entire latency-accuracy tradeoff space while matching the accuracy for individual points in this space at a fraction of the training cost.

This *SuperNet* structure is achieved by sharing the overlapping weights for NN subgraphs, also referred to as *SubNets*. [1,8] enable the extraction of individual subgraphs along the Pareto-optimal frontier for specified latency or accuracy thresholds. It's important to note that these *SubNet* NNs partially share their weights, with significant weight overlap. In fact, the *SuperNet* itself is equivalent to the largest *SubNet*, corresponding to the costliest and the most accurate maximum network. Once the whole *SuperNet* (with shared weights) is trained, no further re-training is necessary for any *SubNets* to be used. In fact, these *SubNets* can be directly *activated* for forward pass inference that will only use the specified *SubNet* configuration. Figure 1 shows the overall flow of the proposed system.

In this paper, we propose the next logical step — ability to *serve* (i.e., run inference on) these *SuperNets* after selecting the appropriate *SubNet* for the target latency-accuracy tradeoff. Current DNN accelerators can support the aforementioned multi-DNN switching scenario via two mechanisms, (1) *spatial switching* — where the entire *SuperNet* is deployed, making all *SubNets* simultaneously available, and routing queries (i.e., input activations) to the appropriate *SubNet* in real-time. This approach would require memory capacity larger than the sum of weights for all possible *SubNets* either via an extremely high-capacity memory or several accelerators (one per *SubNet*) making this solution impractical and non-scalable. (2) *temporal switching* — re-loading the appropriate *SubNet* that offers the pareto-optimal choice w.r.t. a specified latency constraint. This would be most memory-efficient (and our baseline), but requires model-switching latency coming into the critical path of the inference latency.

This work proposes a third alternative, that we call *spatial-temporal switching*—leveraging the weight-shared structure of the *SuperNet* mechanism by minimizing weights loaded on the critical path of the queries, while maximizing the reuse of the weights shared between *SubNet* DNNs. We assert that

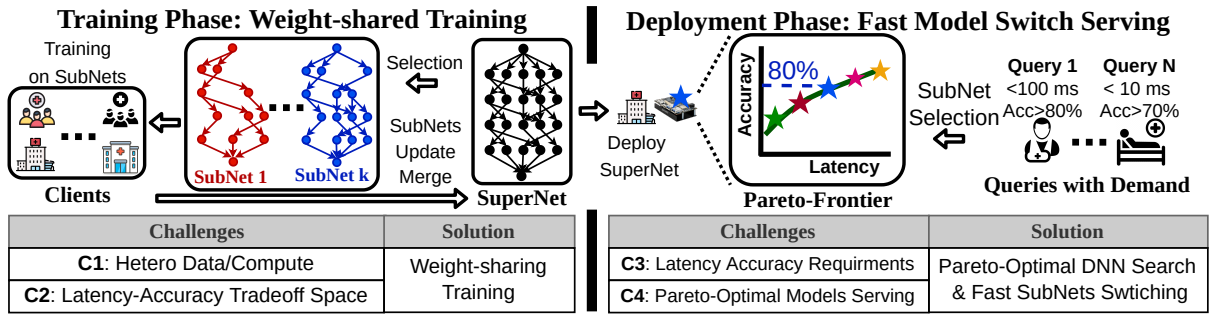



Figure 1: Overview of the proposed system enabling weight-shared training and fast model switch serving.

the spatio-temporal approach to adaptive navigation of the tradeoff space offers the optimal choice with respect to several success metrics, including the fraction of queries that satisfy their latency deadlines (Service Level Objective attainment).

To the best of our knowledge, no DNN accelerator today supports the idea of spatio-temporal model switching. Recall that the efficiency of an accelerator comes from its ability to leverage reuse from inputs, weights, and outputs within a single model via its *dataflow* strategy [2, 3, 5]. We show that spatio-temporal switching adds another dimension of reuse - namely *SubNet* reuse. To this end, we propose ***FastSwitch***—the first DNN inference accelerator design enabling fast *SubNet* switching through spatio-temporal model weight reuse, reducing the total amount of data movement between off-chip and on-chip when switching *SubNets*. ***FastSwitch*** has the following features.

- ***FastSwitch*** is a parameterizable architecture template consisting of scalable execution unit architecture and on-chip buffer organization to leverage all potential reuse opportunities, including SubNet weight reuse.
- We develop a design space exploration (DSE) framework to explore the best HW parameters for ***FastSwitch*** under a given sequence of DNNs and on-chip memory budget.
- The proposed ***FastSwitch*** template implemented with the best DSE choice on both an embedded FPGA board and a cloud FPGA enables end-to-end query serving, outperforming the temporal model switching alternative by $8\% \sim 12\%$.

## 2. SYSTEM OVERVIEW

### 2.1 Weight-sharing Network Serving Flow

Figure 1 illustrates the proposed weight-sharing training and serving flow with 4 key challenges addressed.

**Training Phase.** While OFA [1] proposed a mechanism for producing weight shared supernetworks in a centralized fashion, we train the supernetworks in a federated fashion. Unlike conventional federated learning (FL), where a whole model is shared between participating clients, weight shared FL training requires a framework for *SubNet* distribution and subsequent weight aggregation for overlapping model parameters during FL training. Importantly, the outcome of both FL and centralized weight shared training is a supernetwork that can be served by ***FastSwitch*** during the deployment phase. For each FL training round, different *SubNets* can be assigned to different clients based on their compute resources or data heterogeneity (**C1**). At the end of each local client's training epoch, the updated weights for the given *SubNet* are sent to the server to be aggregated into the joint *SuperNet* (e.g. residing on the central server or in the cloud Figure 1). Weight shared training amortizes the cost of training a family of DNNs individually, incurring the $O(1)$ training cost, while any choice of $k$ Pareto-optimal points requires $\approx O(k)$ cost.

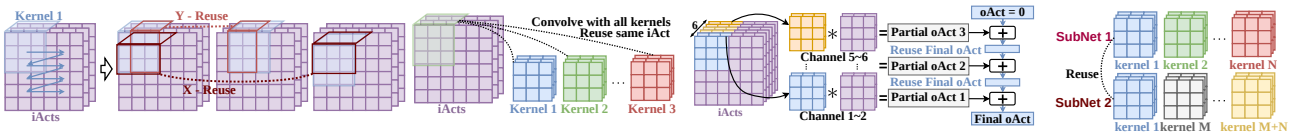
**Deployment Phase.** Once trained to convergence, the *SuperNet* undergoes Neural Architecture Search (NAS) to extract a set of Pareto optimal *SubNet* configurations at desired density/granularity, thereby creating the desired latency/accuracy tradeoff space (**C2**). The system is then ready to serve queries, each requesting specific point on the extracted Pareto frontier, fulfilling (**C3**). We assume a policy engine at a higher level making *SubNet* choice decisions and focus on the *mechanism* of serving them efficiently in this paper. To enable (**C4**), we directly serve the *SuperNet* using our novel hardware accelerator ***FastSwitch*** (prototyped on FPGA) with the following novel features. First, due to our weight-shared training approach, the storage and loading time overheads for the entire *SuperNet* (i.e., family of *SubNets*) is comparable to the largest *SubNet*. Second, we build an optimal dataflow and memory hierarchy within the accelerator to reuse shared weights *across* different weight-shared pareto-optimal *SubNets* to amortize the loading overheads and enable rapid, real-time DNN switching on a single accelerator more *efficiently*.

*In the interest of space, we do not go into details of our FL training and pareto-optimal NAS in this paper; we focus on the architecture of our accelerator for addressing **C4**.*

### 2.2 Data Reuse Opportunities in Accelerator

The serving latency consists of both the *SubNets* switching latency and inference latency of a single *SubNet*. To reduce *SubNets* switching latency, the hardware architecture design must reduce the amount of data being transferred between off-chip and on-chip, i.e. to reuse data fetched from off-chip as much as possible. Further, to reduce inference latency, the hardware needs to improve the throughput by introducing parallelism, which could be also achieved by reusing data locally in computation logic. Therefore, **the essential goal of reducing data movement when switching *SubNets* is to increase data reuse on-chip.**

We show all possible data reuse related to convolution in Figure 2. Here we refers input activation, weights and output activation as iAct, weight and oAct, separately.




(a) iAct Reuse - Sliding Window Overlap (b) iAct Reuse - Multi-Kernel (c) oAct Reuse - Partial Sum Accumulate (d) Weights Reuse - *SubNets*

Figure 2: Data reuse opportunities in serving different *SubNets* leveraged within ***FastSwitch***.

- **iAct Reuse - Sliding Window Overlap** (Figure 2a): With the kernels sliding over the input feature map by a step of specific number of pixels (termed as stride), the input activation in the overlap regions of different sliding windows could be stored on-chip and reused by a single kernel at different sliding window locations.
- **iAct Reuse - Multi-Kernel** (Figure 2b): All kernels need to convolve with input feature map in the same sliding fashion. The input feature map could be stored inside on-chip storage and reused when multiple kernels are sliding together in the same pace.
- **oAct Reuse - Partial Sum Accumulation** (Figure 2c): For layers deep inside the neural network, the number of channel will usually explode to thousands which exceeds the accumulation range of the execution units. And thus part of the partial sum has to be sent back to off-chip memory for temporal storage. Then they will go back to on-chip buffer for final accumulation.
- **Weights Reuse - Multi-iAct Tiles** (Figure 2b): Different iAct tiles will convolve with same kernels so that weights of kernels are reused.
- **Weights Reuse - *SubNets*** (Figure 2d): The weight-sharing feature renders different *SubNets* sharing some weights in common, which could sit inside the on-chip storage and reused by different *SubNets*.

Table 1 contrasts ***FastSwitch*** against some prior arts in terms of leveraging reuse opportunities.

Table 1: Reuse comparison (prior works v.s. ***FastSwitch***).

| Work | iActs Reuse Sliding Window Overlap | iActs Reuse Multi-Kernel | oAct Reuse Partial Sum | weights Reuse iAct Tiling | **Weights Reuse *SubNets*** |
|---|---|---|---|---|---|
| MAERI [5] | ✓ | ✗ | ✗ | ✓ | temporal ✗ |
| NVDLA [6] | ✗ | ✓ | ✓ | ✓ | temporal ✗ |
| Eyeriss [2] | ✓ | ✓ | ✗ | ✓ | temporal ✗ |
| Xilinx DPU [10] | ✗ | ✓ | ✓ | ✓ | temporal ✗ |
| ***FastSwitch*** | ✓ | ✓ | ✓ | ✓ | spatial temporal ✓ |

# 3. FASTSWITCH ARCHITECTURE

Figure 3 shows a high-level architecture of ***FastSwitch***.

## 3.1 Compute Array

**Dot Product Engine (DPE).** The key building block of DNN accelerators is the ability to compute *dot-products*. The Google TPU systolic array [4] computes fixed-size dot products in each column by keeping weights stationary and forwarding (streaming) inputs from one column to the other, NVDLA [6] employes dedicated dot product engines (DPEs) of size 64, while flexible accelerators [5,7] have DPEs of configurable sizes (enabled via all-to-all connectivity between the buffers and PEs). In this work, we picked fixed size DPEs of size 9, inspired by the common case of $3 \times 3$ filters in most CNNs, to keep hardware cost simple. Within the DPEs, we leveraged an adder-tree for reduction, similar to NVDLA.

**Dataflow.** To further increase the throughput, we instantiate a 2D array of DPEs to boost the parallelism as shown in the Fig. 3. In the vertical axis, iActs pass through DPEs of different rows in the store-and-forward fashion. In the horizontal axis, weights of kernel are broadcasted to all DPEs in the same row such that DPE at different columns share the same weights. These weights remain stationary. Therefore, the number of row indicates the total number of kernels in DPE Array targeting *iAct Reuse - Multi-Kernels* (Fig. 2b), noted by $K_P$. While the number of column stands for total number of iAct sliding windows, i.e. *iAct Reuse - Sliding Window Overlap* (Fig. 2a), noted as $Y_P$.

## 3.2 On-chip Buffers

We designed a custom on-chip buffer hierarchy to tile the overall workload, both for reordering tiles for DPE Array and leveraging other reuse opportunities in tiles not supported by the DPE Array. The on-chip buffers are divided into multiple groups as illustrated by different colors in Figure 3.

### 3.2.1 *On-chip Buffers for Weights*

**Persistent Buffer (PB)**. The PB is designed to target at enabling *Weights Reuse - SubNets* in Figure 2d, i.e. store the common weights from different *SubNets* such that the hardware does not need to fetch them when incoming queries request to change *SubNets*. For example, the kernel 1 in Figure 2d will be stored inside PB and reused when ***FastSwitch*** switching from *SubNet* 1 into *SubNet* 2 such that kernel 1 will not be fetched from off-chip memory again.

**Dynamic Buffer (DB)**. The DB is a typical on-chip storage to store the weights of requested *SubNet*. By adopting a PB, only non-common weights need to be fetched from off-chip to the on-chip storage. For example, all kernels except the common part (kernel 2 to kernel $N$) will be stored in DB when targeting at *SubNet* 1, and will be replaced by kernel $M$ to kernel $M+N$ when switching into *SubNet* 2.

**Data Accessing for PB and DB**. An example of weights storage on-chip is shown in Figure 4, weights are firstly tiled. After that, dynamic weights will be fetched from DRAM into DB such that the common weights are shared across different queries. All weights in DB and PB will finally be unified and supplied to DPE Array.

### 3.2.2 *On-chip Buffer for iActs and OActs*

**Streaming Buffer (SB)** is designed to store the entire iAct tile and thus support multiple kernels iAct reuse (Figure 2b).

**Line Buffer (LB)** is designed to support iAct reuse in multiple sliding windows (Figure 2a) [9]. We extend the original line buffer stride supporting through skipping consecutive sliding windows when stride is larger than 1.

**Output Buffer (OB)** provides in-place accumulation for oAct of different channel such that only the final oActs will be sent off-chip to reduce the data movement.

### 3.2.3 *Size of On-chip Buffers*

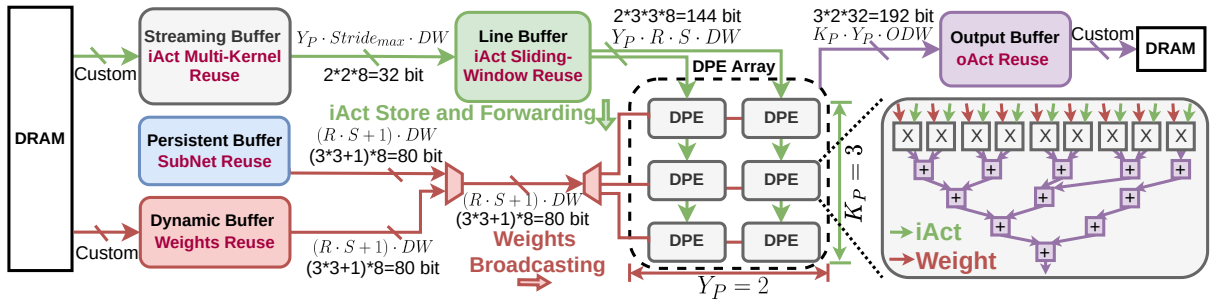



Figure 3: The Overall ***FastSwitch*** Architecture ($K_P = 3, Y_P = 2$)

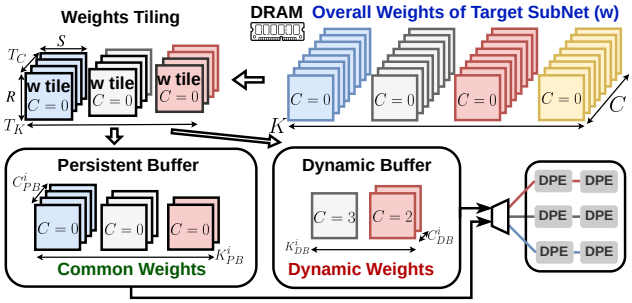



Figure 4: Weights Storage Breakdown in PB and DB.

Table 2: Dimensions of 2D on-chip Buffers

| Buffer | Width | Height |
|---|---|---|
| DB | $DB_W = K_P \cdot (R \cdot S+1) \cdot DW$ | $DB_H = T_K \cdot T_C / K_P$ |
| SB | $SB_W = Y_P \cdot stride_{max} \cdot DW$ | $SB_H = T_C \cdot T_X \cdot \lceil \frac{T_Y}{Y_P \cdot stride_{max}} \rceil$ |
| LB | $LB_W = Y_P \cdot stride_{max} \cdot DW$ | $LB_H = R \cdot \lceil \frac{T_Y}{Y_P \cdot stride_{max}} \rceil$ |
| OB | $OB_W = \lceil BW/ODW \rceil \cdot ODW$ | $OB_H = (T_X - R + 1) \cdot \lceil \frac{T_Y - S + 1}{Y_P} \rceil \cdot K_P / OB_W$ |
| PB | $PB_W = K_P \cdot (R \cdot S+1) \cdot DW$ | $PB_H$ = Custom Value |

Note: $R$: Filter rows, $S$: Filter cols, $DW$: iAct or weight datawidth, $ODW$: oAct datawidth, $stride_{max}$: max stride in workload. All the buffers are organized in 2D array (depth = total number of data, width = bandwidth), $BW$: off-chip bandwidth.

On-chip buffers are designed to provide temporal on-chip storage for iAct tiles and weight tiles. Therefore, the maximal sizes of iAct tile $(T_C, T_X, T_Y)$ and weight tile $(T_K, T_C, R, S)$ determine sizes of on-chip buffers as shown in Table 2.

## 3.3 Design Space Exploration

The design space of ***FastSwitch*** is large with multiple trade-off considerations. We design a two-phase DSE framework. In the first phase, we determine the appropriate size of the DPE array under the hardware resource budget, essentially determining $Y_P$ and $K_P$. In the second phase, the sizes of on-chip buffers are determined (Sec. 3.2.3). We currently employ a brute-force grid-search through all parameters, optimizing for minimizing overall latency.

# 4. PRELIMINARY EVALUATION

**Deployment Platform.** We implemented the proposed ***FastSwitch*** on two FPGA boards, Ultra 96, ZCU104 and Alevo U280. Then we model the proposed ***FastSwitch*** in DSE tool and evaluate the ideal potential performance delivered through PB and impact of different components.

**Workload.** We adopt the elastic weight-sharing ResNet-50 as the overall *SuperNet* [1], and sizes of *SubNets* ranges from the minimal *SubNet* (i.e. weights are reused by all other *SubNets*, 24.95 MB) to the maximal *SubNet* (i.e. *SuperNet*, 183.51 MB). For different *SubNets*, the shared weights could take up 13% $\sim$ 100% of entire weights. To evaluate

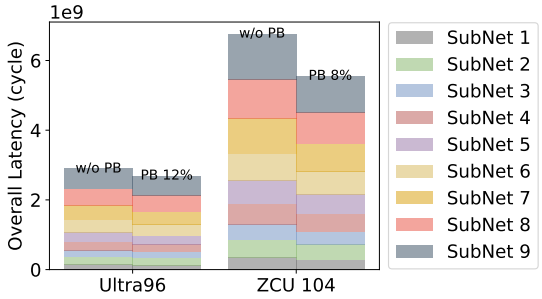



Figure 5: Latency breakdown of ***FastSwitch*** v.s. Baseline.

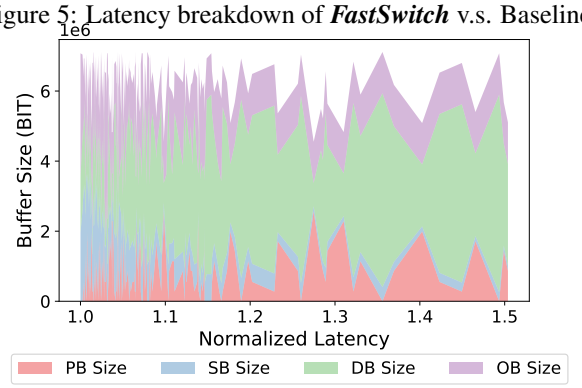



Figure 6: Reuse impact exploration using DSE for Ultra 96.

***FastSwitch*** with all possible cases, a sequence of 9 *SubNets* with sizes uniformly sampled from the minimal *SubNet* to *SuperNet* is generated as evaluation workload.

**Inference Latency Results.** Figure 5 shows the proposed ***FastSwitch*** v.s. baseline (***FastSwitch*** without PB) on the both two devices. With common weight preloaded in PB, only non-common (Figure 2d) weights get fetched on-chip such that the overall latency could be reduced by 8% $\sim$ 12 %.

**Impact of Different Reuse on Performance.** Figure 6 presents performance of all design choices implementing ***FastSwitch*** on Ultra 96 board, the less the x-value is, the faster the design choice achieve. On the right-hand size, OB dominates on-chip storage breaking the balance among on-chip buffers thus achieving bad performance. On the left-hand size, none of reuse dominates achieving balance among different reuse and thus delivering best performance.

# 5. CONCLUSION

This work introduces a system for training weight-shared NNs and serving them in real-time, deployed on FPGAs. We identify a new opportunity for weight reuse - across SubNets, that we exploit via an accelerator called FastSwitch. Preliminary results show 8-12% speedup over a SOTA baseline.

# REFERENCES


[1] H. Cai, C. Gan, and S. Han, "Once for all: Train one network and specialize it for efficient deployment," *CoRR*, vol. abs/1908.09791, 2019. [Online]. Available: http://arxiv.org/abs/1908.09791

[2] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[3] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices," *arXiv preprint arXiv:1807.07928*, 2018.

[4] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[5] H. Kwon, A. Samajdar, and T. Krishna, "MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[6] NVIDIA. (2016) NVIDIA Deep Learning Accelerator (NVDLA). [Online]. Available: http://nvdla.org/primer.html

[7] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 58–70.

[8] M. Sahni, S. Varshini, A. Khare, and A. Tumanov, "CompOFA – compound once-for-all networks for faster multi-platform deployment," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=IgIk8RRT-Z

[9] C. Wang, Y. Liu, K. Zuo, J. Tong, Y. Ding, and P. Ren, "ac 2 slam: Fpga accelerated high-accuracy slam with heapsort and parallel keypoint extractor," in *2021 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2021, pp. 1–9.

[10] Xilinx. (2022) Xilinx Deep Learning Unit (DPU). [Online]. Available: https://docs.xilinx.com/r/en-US/ug1414-vitis-ai/Deep-Learning-Processor-Unit