Caffe *Barista*:
Brewing Caffe with FPGAs in the Training Loop

Diederik Adriaan Vink[†*], Aditya Rajagopal[†*], Stylianos I. Venieris[‡], Christos-Savvas Bouganis[†]

[†]*Intelligent Digital Systems Lab, Imperial College London*
{diederik.vink14, aditya.rajagopal14, ccb98}@ic.ac.uk

[‡]*Samsung AI Center, Cambridge*
s.venieris@samsung.com

[*]*Indicates equal contribution.*

Abstract—As the complexity of deep learning (DL) models increases, their compute requirements increase accordingly. Deploying a Convolutional Neural Network (CNN) involves two phases: training and inference. With the inference task typically taking place on resource-constrained devices, a lot of research has explored the field of low-power inference on custom hardware accelerators. On the other hand, training is both more compute- and memory-intensive and is primarily performed on power-hungry GPUs in large-scale data centres. CNN training on FPGAs is a nascent field of research. This is primarily due to the lack of tools to easily prototype and deploy various hardware and/or algorithmic techniques for power-efficient CNN training. This work presents *Barista*, an automated toolflow that provides seamless integration of FPGAs into the training of CNNs within the popular deep learning framework Caffe. To the best of our knowledge, this is the only tool that allows for such versatile and rapid deployment of hardware and algorithms for the FPGA-based training of CNNs, providing the necessary infrastructure for further research and development.

I. Introduction

Convolutional Neural Networks (CNNs) are one of the primary components across a wide variety of AI tasks, from face recognition [1] to drone navigation [2]. The process of deploying a CNN involves two stages. First, the CNN is trained [3] on a large amount of labelled data from a task-specific dataset. The second stage involves performing inference on unseen inputs for either classification [4, 5], detection [6, 7] or segmentation [8]. Inference is usually performed in resource- and power-constrained environments and hence the CNN needs to be deployed on power-efficient embedded devices such as mobile System-on-Chips (SoCs) [9], or FPGA-based platforms [10]. To this end, significant efforts have been invested towards custom FPGA-based accelerator designs for the inference stage of CNNs [11–15].

Due to its large computational demands and the massive datasets, CNN training is usually performed on powerful GPUs hosted in private clusters or data centres. For such setups, the power and cooling infrastructure constitutes the dominant factor of the operational expenses [16]. With GPUs being power-hungry, they become costly platforms to maintain. This fact has led industrial players to equip their servers with custom ASICs, such as Google TPU [17], Graphcore IPU [18] and Amazon Inferentia [19]. Nevertheless, the long development time and time-to-market together with their fixed functionality

limit the ASICs' ability to exploit model-specific optimisations and support the latest fast-paced algorithmic advances.

In this context, FPGAs constitute a promising alternative [20–22]. Due to their customisability and reconfigurability, FPGAs can attain competitive performance and power efficiency for flexible, power- and cost-efficient development and deployment of DL training workloads. At the same time, public cloud providers are increasingly offering access to FPGA platforms [23–25], increasing their accessibility and making the rapid low-cost deployment of FPGA designs feasible. Nevertheless, so far, FPGA-based CNN training has only slightly been explored [26–28] largely due to the lack of tools to easily prototype and deploy various hardware and/or algorithmic techniques for efficient CNN training.

The primary contribution of this work is *Barista*, an open-source toolchain[1] integrated into the widely used DL framework Caffe [29], that enables the rapid prototyping and deployment of FPGA-based kernels for CNN training. Additionally, the work provides a memory-aware model for the execution of an FPGA-based general matrix multiply (GEMM) kernel along with an initial HLS implementation of this kernel. In this manner, *Barista* allows both hardware researchers and machine learning experts to explore novel hardware and algorithmic techniques respectively for power-efficient training.

II. Background & Related Work

Barista enables rapid prototyping and deployment of hardware accelerators for DL training. Here, key challenges and requirements of architectures targeting such workloads are described and existing work on FPGA-based training is reviewed.

DNN Training. DNNs are generally trained offline, following an iterative process [30]. Each iteration comprises three steps: a *forward pass*, a *backward pass* (through backpropagation) and a *weight update*. The forward pass (inference task) calculates the loss for a given input. The backward pass employs the backpropagation algorithm to compute the gradient of the loss with respect to the trainable weights; the weight update step updates the weights using these gradients. In each iteration, training operates over mini-batches of labelled inputs from the training set; in this respect, *throughput* is the primary

---

[1]https://github.com/ICIdsl/caffe_fpga.git

performance metric of interest, in contrast to the inference task's low-latency requirements. Furthermore, with power and cooling being a critical expense in both public and private clusters, power efficiency constitutes another decisive metric.

FPGA-based CNN Training. Existing work on FPGA-based CNN training can be taxonomised in two broad categories: 1) costly multi-FPGA systems [26, 27] and 2) highly customised accelerators [31, 32]. F-CNN [26] enabled run-time reconfiguration through overlapping computation by utilising multi-device platforms. FPDeep [27] proposed a load balancing scheme to train CNNs across multiple FPGAs (>15). Focusing on single-FPGA setups, DarkFPGA [31] employed a batch-oriented data layout scheme optimised for a specific hardware design and is not applicable to other accelerators. Furthermore, [32] designed a low-precision training accelerator through hardware-algorithm co-design. By replacing the GEMM, the approach of this work enables training of any DNN that uses matrix multiplication.

All aforementioned works adopt proprietary front-ends, lacking integration with the traditional machine learning frameworks and the support that they provide, and/or propose workload-specific architectures that cannot be used for general DNN training. Similar to this work, *FeCaffe* [33] proposes a system that integrates Caffe with an FPGA. However, *Barista* provides more details on the challenges of developing a custom accelerator, as well as provides an analytical model for performance prediction allowing the tuning of the framework under diverse workloads. Additionally, unlike *FeCaffe*, the Caffe integration of the proposed system will be open-sourced to promote adoption and research, and can be deployed on any AWS F1 instance using the GEMM bitstream provided.

III. SYSTEM DESIGN

The *Barista* tooflow integrates with the Caffe framework and targets systems with PCIe-based FPGA accelerators. To this end, *Barista* consists of three components: 1) a software integration layer that enables the seamless integration of the FPGA accelerator with Caffe, 2) an FPGA-based hardware accelerator, and 3) an OpenCL runtime that orchestrates the CNN execution between a host CPU and the FPGA. Upon deployment, the FPGA device runs a kernel that is responsible for executing the matrix multiplications involved in the forward and backward pass of the CONV layers throughout CNN training. The CPU executes all other operations of the training process and coordinates the offloading of computations to the FPGA.

A. Caffe GEMM Execution Flow

This section provides a description of Caffe's native execution flow for CONV layers. Initially, Caffe selects which platform (CPU/GPU/FPGA) to execute on. Next, for each batch in each layer Caffe calls the batch-level GEMM function. At this point, the implementations for forward and backward pass start to deviate. For the forward pass, Caffe calls `im2col` on all the inputs and weights to convert them to matrices in order to execute convolutions as GEMMs. For the backward pass, the
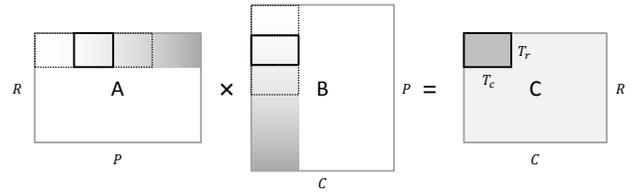

Fig. 1: Overview of the adopted blocked GEMM strategy.

gradients w.r.t the weights are calculated by multiplying the inputs with the gradients w.r.t the output. Then, it calculates the gradient w.r.t to the input for each element in the batch by multiplying the weights with the gradient w.r.t the output. All these matrices are split into tiles (Section III-B) and then fed to the FPGA GEMM kernel. As the forward pass is a GEMM, `im2col` is not required for backpropagation.

B. Accelerator Architecture for CNN Training

The developed hardware architecture is designed to meet the typical requirements of CNN training workloads (Section II). The core compute block comprises a parametrised systolic array for the execution of blocked GEMM that is reused across both the forward and backward passes of CONV layers.

Blocked GEMM: As shown in Fig. 1, in the operation ($\mathbf{C}=\mathbf{AB}$), matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ are partitioned into tiles. Matrix $\mathbf{A}$ is partitioned into $\left\lceil \frac{R}{T_r} \right\rceil \cdot \left\lceil \frac{P}{T_p} \right\rceil$ tiles of size $T_r \times T_p$, matrix $\mathbf{B}$ is partitioned into $\left\lceil \frac{C}{T_c} \right\rceil \cdot \left\lceil \frac{P}{T_p} \right\rceil$ tiles of size $T_p \times T_c$ and the output matrix $\mathbf{C}$ is partitioned into $\left\lceil \frac{R}{T_r} \right\rceil \cdot \left\lceil \frac{C}{T_c} \right\rceil$ tiles of size $T_r \times T_c$. If $\frac{R}{T_r} \notin \mathbb{Z}$, then zeros are added to dimension R until $\frac{R}{T_r} \in \mathbb{Z}$. The same applies to P and C and this process will be referred to as *Tiling* through the rest of the paper.

From an operational perspective, the accelerator computes one tile of the output matrix $\mathbf{C}$ at a time, until all output tiles are computed. For the computation of a single tile, $\left\lceil \frac{P}{T_p} \right\rceil$ tiles from matrix $\mathbf{A}$ and $\mathbf{B}$ are processed. In the implemented blocking strategy, each output tile is cached in the on-chip memory of the accelerator until it has been fully formed. Consequently, the intermediate results of the tile are reused $\left\lceil \frac{P}{T_p} \right\rceil$ times before they are written back to the external memory, relaxing the bandwidth requirements of the accelerator.

Hardware Architecture: Fig. 2 shows the adopted hardware architecture for accelerating the blocked GEMM algorithm. The core of the design is a mesh of processing elements (PEs) and has a throughput of one output per cycle, when the necessary data are available in buffer A and B. The dimensions of the mesh are compile-time configurable with a total of $T_r \times T_c$ PEs. All of the inputs to the GEMM are pre-processed by the CPU into a tiled layout that is sequentially stored in memory. To better utilize the memory bandwidth, the two matrices to be multiplied are sent to the off-chip memory on the FPGA board in one transaction. When the execution of the kernel is triggered, the input tiles $T_r \times T_p$ and $T_p \times T_c$
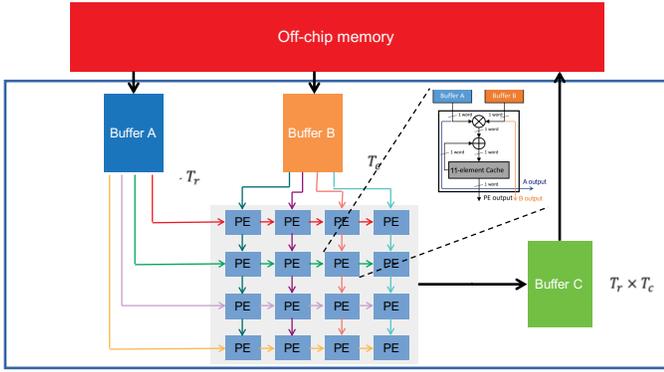
Fig. 2: Diagram of the systolic-array design.

are burst-read from the off-chip memory into buffers A and B which are stored on on-chip memories (*i.e.* BRAMs).

Processing Element: Fig. 2 also shows the internal design of each PE. Each PE is responsible for computing one element of a tile of the output matrix. From a hardware perspective, each PE contains a single multiply-accumulate unit and a local cache for storing the intermediate results of the output, until the final result is ready and written out to external memory. The dataflow depicted in Fig. 2 enables efficient data passing between PEs in a pipelined fashion, saving routing resources and improving the scalability potential of the design.

Precision-aware interleaving: Depending on the adopted precision and target device, the latency of a multiplier is Q cycles. As a result, a direct implementation would require each input to wait for Q cycles until the previous result would be ready for the accumulation. To alleviate this, when $Q>1$, we employ an interleaving technique that computes $Q+1$ independent intermediate results in a pipelined manner, storing them in the PE's cache. As a final step, all $Q+1$ partial values are accumulated into the final result. From a performance perspective, this strategy enables a throughput of 1.

C. OpenCL Runtime

The CPU-FPGA interactions and the FPGA execution are orchestrated by *Barista*'s OpenCL runtime. Prior to performing a GEMM operation, this module is responsible for allocating the necessary memory and tiling the input matrices given the selected tile sizes. Next, it coordinates all CPU-FPGA transfers, launches the FPGA execution and finally collects and untiles the final result to comply with the expected GEMM output format. The runtime is executed by the host CPU and employs `aligned_storage` vectors for tiles to ensure FPGA word aligned storage on the off-chip memory.

IV. PERFORMANCE MODEL

To select the dimensions of the mesh that would yield the highest performance, a performance model was built which estimates the attainable execution time of the hardware design. The performance model consists of two components: 1) the estimated execution time for the processing of a matrix multiplication by the systolic array (Eq. (3)) and 2) the estimated

memory transfer time for transferring the matrices **A**, **B** and **C** between the host and the FPGA's off-chip memory.

Off-chip memory transfer time: The design requires $T_r + T_c$ inputs per cycle. By denoting the wordlength (bits) of an element by WL the required data that needs to be accessed from the off-chip memory is Data_{mem}, where the last term captures the data written back to the memory per tile:

$$\text{Data}_{\text{mem}} = WL \cdot \left\lceil \frac{R}{T_r} \right\rceil \cdot \left\lceil \frac{C}{T_c} \right\rceil \cdot ((T_r \cdot P + T_c \cdot P) + T_c \cdot T_r)$$

Overall, given a memory bandwidth of B_{mem}, the latency for accessing the off-chip memory per matrix multiplication is:

$$\text{Latency}_{\text{mem}} = \frac{\text{Data}_{\text{mem}}}{B_{\text{mem}}} \quad (1)$$

Compute time: The number of clock cycles that is needed for the developed system to process the matrix-multiplication computation (*i.e.* **C=AB**, **A** is a $R \times P$ matrix, and **B** is a $P \times C$ matrix and the output tile size is $T_r \times T_c$) is:

$$\text{Cycles}_{\text{compute}} = \left\lceil \frac{R}{T_r} \right\rceil \cdot \left\lceil \frac{C}{T_c} \right\rceil \cdot \\ \left(\left(\left\lceil \frac{P}{T_p} \right\rceil \cdot (T_p + T_c + T_r - 2) \right) + (Q+1)^2 \right) \quad (2)$$

IP execution time: Total GEMM kernel execution latency, when the data are already available in the off-chip memory, is:

$$\text{Latency}_{\text{total}} = \frac{\text{Cycles}_{\text{compute}}}{f_{\text{clk}}} + \text{Latency}_{\text{mem}} \quad (3)$$

where f_{clk} denotes the clock frequency of the FPGA device.

PCIe transfer time: The PCIe transfer time captures the latency for the communication of the data from the CPU to off-chip memory. $\text{Data}_{\text{PCIe}} = WL \cdot (R \cdot P + C \cdot P + R \cdot C)$ captures the data to be transferred. Eq. (4) captures the transfer latency given the PCIe bandwidth B_{PCIe}:

$$\text{Latency}_{\text{PCIe}} = \frac{\text{Data}_{\text{PCIe}}}{B_{\text{PCIe}}} \quad (4)$$

$$\text{Overall latency} = \text{Latency}_{\text{PCIe}} + \text{Latency}_{\text{total}} \quad (5)$$

Resource Usage Model: A model for estimating resource usage as a function of the configurable parameters T_r, T_c and T_p was developed. Eq. (6) and (7) model resource usage.

$$\text{DSP blocks} = \underbrace{(T_r \cdot T_c)}_{\text{\# of PEs}} \cdot \underbrace{V}_{\text{DSPs/MAC unit}} \quad (6)$$

$$\text{BRAM} = WL \left(\underbrace{T_r \cdot T_p}_{\text{buffer A}} + \underbrace{T_p \cdot T_c}_{\text{buffer B}} + \underbrace{T_r \cdot T_c \cdot (Q+1)}_{\text{buffer C}} \right) \quad (7)$$
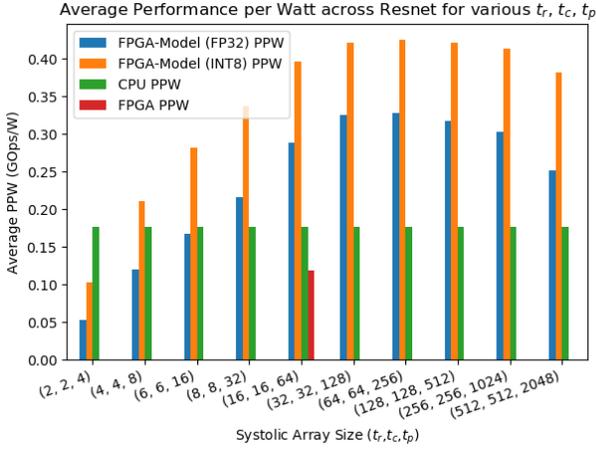
The factor of $Q+1$ for buffer C is due to interleaving.

Fig. 3: Average PPW across ResNet20 for various $\langle T_r, T_c, T_p \rangle$.

V. EVALUATION

The performance of the tool was evaluated on the Xilinx Virtex UltraScale+ XCVU9P FPGA hosted on the F1 instances on Amazon Web Services (AWS) [23]. This device has 2586k logic cells, 6840 DSPs and 75.9Mb BRAM. The DSPs take $Q=10$ cycles for an FP32 multiply and $V=5$ DSPs are used per FP32 MAC unit. For INT8 operations, $Q=1$ and $V=1$.

An FP32 GEMM accelerator (Section III-B) was developed using Vivado HLS and synthesised using SDAccel 2018.2. The design was clocked at 250 MHz and the accelerator was configured with $\langle T_r, T_c, T_p \rangle$ set to $\langle 16, 16, 64 \rangle$ using the performance model (Section IV). This was the highest performing design that would route with the current HLS implementation. It used 18.8%, 10.8%, 8.8% and 14.1% of the available DSPs, LUTs, FFs and BRAM respectively. This design was verified locally on a Xilinx Alveo U250 FPGA. Two widely used CNNs, AlexNet [3] and ResNet20 [34] were trained on the CIFAR10 dataset, and *Barista* was compared with the CPU [and GPU] implementation on Caffe [29].

Figure 3 shows the average performance-per-watt (PPW) measured in GOp/s/watt across all CONV layers during the training process (i.e. forward and backward passes) of ResNet20 using *Barista* for the FPGA and CPU. [For the GPU, the average PPW across all Resnet20 CONV layers was 1.54.] AWS power profiling showed that the FPGA used 8W of power when running these designs, compared to the CPU's (Intel Xeon E5-2686v4@2.3GHz) 145W and [GPU's (NVIDIA GTX 1080Ti) 279W]. Profiling was performed using Caffe's internal timers and the design was verified by comparing the FPGA's to the CPU's output. It is seen that for all sizes of kernel larger than $\langle 8, 8, 32 \rangle$, both the FP32 (blue bars) and INT8 (orange bars) model predictions outperform the CPU (green bars). Additionally, for sizes of kernel larger than $\langle 64, 64, 256 \rangle$ the performance degrades from performing a large number of zero ops due to tiling (Section III) when the kernel size starts to significantly exceed the sizes of the input matrices. However, the implementation of the $\langle 16, 16, 64 \rangle$
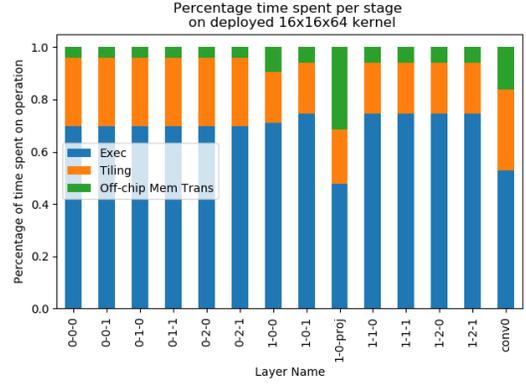
(a) Implemented
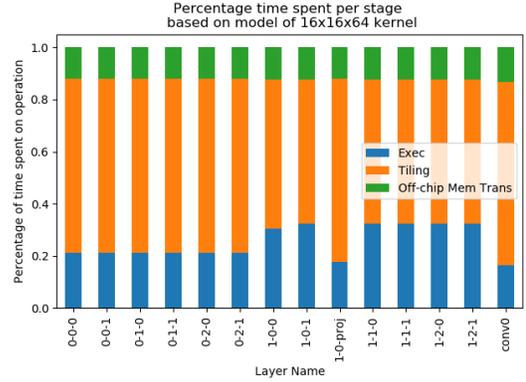
(b) Model

Fig. 4: Relative time spent on each stage for various ResNet20 layers. Layer names have format (group-residual block-conv).

kernel (red bar) does not outperform the CPU (green bars).

To identify the bottlenecks causing the difference in expected and achieved performance, further profiling was performed using OpenCL. Fig. 4a breaks down the relative time spent on each stage of GEMM execution for the implemented kernel using profiled data. The kernel execution (blue), which includes off-chip to FPGA memory transfers, is seen to be the biggest bottleneck at the moment taking more than 50% of the time in all CONV layers. Kernel execution (blue) profiling through Xilinx Vitis' profiler showed that memory bandwidth utilisation for kernel to off-chip memory transfers was in the range of about 10%. Nevertheless, compute unit utilisation rates are at least 70% indicating the system can be further improved by exploiting memory optimisations. Fig. 4b shows the same breakdown but now using data from the model for estimates of kernel execution time (blue) and host to off-chip memory transfer time (green). Profiled time was used for tiling (orange), which is performed on the CPU. The model assumes full utilisation of the DDR4 bandwidth (30Gbps) between off-chip memory and the kernel. Fig. 4b demonstrates that with full bandwidth utilisation, the bottleneck is shifted from the FPGA kernel execution (blue) to tiling on the CPU (orange).

TABLE I: AlexNet predicted best FPGA, CPU and GPU PPW

CONV Layer	conv1	conv2	conv3	conv4	conv5
$\langle \mathbf{T_r}, \mathbf{T_c}, \mathbf{T_p} \rangle$	$\langle 32, 32, 74 \rangle$	$\langle 32, 32, 64 \rangle$	$\langle 36, 36, 64 \rangle$	$\langle 32, 32, 64 \rangle$	$\langle 32, 32, 64 \rangle$
FPGA PPW	0.59	0.29	0.078	0.076	0.073
CPU PPW	0.35	0.24	0.089	0.13	0.11
GPU PPW	0.13	0.58	0.43	0.50	0.28

Reducing the DDR4 bandwidth assumption to 3Gbps (10%) in the model predicts a performance close to that achieved by the implemented kernel, supporting the bottleneck analysis from Xilinx's Vitis tool.

A further experiment on tailoring the kernel architecture to the workload was conducted using the model. A grid-search was performed across various values of T_r, T_c and T_p for designs that are expected to fit on the chosen board based on the memory model. For ResNet20, the kernel which is predicted to have the highest PPW on average across all layers of the network is $\langle 36, 36, 72 \rangle$ with a performance of 0.33 GOp/s/W compared to the CPU's 0.18 (+83%). Layer-wise tuning showed that although different-sized kernel performed better on different layers, there is no overall difference in achieved PPW compared to using a single $\langle 36, 36, 72 \rangle$ kernel for all layers. For AlexNet, however, this exploration showed that tailoring the kernel to the layer can provide overall PPW benefits. Table I describes the performance of the best kernels per layer and shows that for some layers a CPU performs better than an FPGA for FP32 computations. By selectively performing FPGA-based GEMM for conv1 and 2, otherwise using the CPU, the overall achieved PPW is 0.24 compared to the CPUs 0.18 (+33%) and 0.22 (+10%) achieved if all layers use one $\langle 32, 32, 64 \rangle$ kernel.

VI. CONCLUSION AND FUTURE WORK

Caffe *Barista* enables hardware designers to rapidly prototype novel custom accelerators by seamlessly replacing the provided kernel with one that implements the same interface. The model suggests that up to 83% higher PPW [compared to a CPU] can be achieved for a lower absolute power consumption using custom precision arithmetic and/or increasing memory bandwidth utilisation through batching and on-chip tiling. From the perspective of a DL researcher, *Barista* allows running any combination of optimisers (*e.g.* SGD, RMSProp, AdaGrad), learning rate schedules and a variety of other training-related parameters or algorithms that are natively supported by or can be implemented in Caffe. To the best of our knowledge, *Barista* is the first open-source tool that allows for such versatile and rapid deployment of hardware and algorithms related to the training of CNNs on FPGAs.

VII. ACKNOWLEDGEMENTS

REFERENCES

[1] J. Deng, J. Guo, N. Xue, and S. Zafeiriou, "ArcFace: Additive Angular Margin Loss for Deep Face Recognition," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 4685–4694. 1

[2] A. Kouris and C. Bouganis, "Learning to Fly by Myself: A Self-Supervised CNN-Based Approach for Autonomous Navigation," in *International Conference on Intelligent Robots and Systems (IROS)*, 2018, pp. 1–9. 1

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2012, pp. 1097–1105. 1, 4

[4] J. Wang, Y. Yang, J. Mao, Z. Huang, C. Huang, and W. Xu, "CNN-RNN: A Unified Framework for Multi-Label Image Classification," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 2285–2294. 1

[5] S. Hershey, S. Chaudhuri, D. P. W. Ellis, J. F. Gemmeke, A. Jansen, R. C. Moore, M. Plakal, D. Platt, R. A. Saurous, B. Seybold, M. Slaney, R. J. Weiss, and K. Wilson, "CNN Architectures for Large-Scale Audio Classification," in *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2017. 1

[6] R. Girshick, "Fast R-CNN," in *International Conference on Computer Vision (ICCV)*, 2015, pp. 1440–1448. 1

[7] A. Kouris, C. Kyrkou, and C.-S. Bouganis, "Informed Region Selection for Efficient UAV-based Object Detectors: Altitude-aware Vehicle Detection with CyCAR Dataset," in *International Conference on Intelligent Robots and Systems (IROS)*, 2019, pp. 51–58. 1

[8] J. Long, E. Shelhamer, and T. Darrell, "Fully Convolutional Networks for Semantic Segmentation," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015. 1

[9] M. Almeida, S. Laskaridis, I. Leontiadis, S. I. Venieris, and N. D. Lane, "EmBench: Quantifying Performance Variations of Deep Neural Networks Across Modern Commodity Devices," in *The 3rd International Workshop on Deep Learning for Mobile Systems and Applications (EMDL)*, 2019, pp. 1–6. 1

[10] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart, *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Glasgow, GBR: Strathclyde Academic Media, 2014. 1

[11] S. I. Venieris, A. Kouris, and C.-S. Bouganis, "Toolflows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions," *ACM Computing Surveys*, vol. 51, pp. 1–39, 2018. 1

[12] R. DiCecco, G. Lacey, J. Vasiljevic, P. Chow, G. Taylor, and S. Areibi, "Caffeinated FPGAs: FPGA framework For Convolutional Neural Networks," in *2016 Interna-*

tional Conference on Field-Programmable Technology (FPT), 2016, pp. 265–268.

[13] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong, "FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates," in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017, pp. 152–159.

[14] L. Jiao, C. Luo, W. Cao, X. Zhou, and L. Wang, "Accelerating low bit-width convolutional neural networks with embedded FPGA," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–4.

[15] Y. Yu, T. Zhao, K. Wang, and L. He, "Light-OPU: An FPGA-Based Overlay Processor for Lightweight Convolutional Neural Networks," in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2020, p. 122–132. 1

[16] C. Kozyrakis, "Resource Efficient Computing for Warehouse-scale Datacenters," in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2013, pp. 1351–1356. 1

[17] N. P. Jouppi *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 1–12. 1

[18] Z. Jia, B. Tillman, M. Maggioni, and D. P. Scarpazza, "Dissecting the Graphcore IPU Architecture via Microbenchmarking," *ArXiv*, vol. abs/1912.03413, 2019. 1

[19] Amazon, "Amazon Inferentia ML Chip," https://aws.amazon.com/machine-learning/inferentia/, 2020, [Retrieved: June 18, 2020]. 1

[20] A. Kouris, S. I. Venieris, and C. Bouganis, "CascadeCNN: Pushing the Performance Limits of Quantisation in Convolutional Neural Networks," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 155–1557. 1

[21] S. I. Venieris and C. Bouganis, "f-CNNx: A Toolflow for Mapping Multiple Convolutional Neural Networks on FPGAs," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 381–3817.

[22] S. I. Venieris and C. S. Bouganis, "fpgaConvNet: Mapping Regular and Irregular Convolutional Neural Networks on FPGAs," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–17, 2018. 1

[23] Amazon, "F1 instance in Amazon AWS," https://aws.amazon.com/ec2/instance-types/f1/, [Retrieved: June 18, 2020]. 1, 4

[24] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger, "A Configurable Cloud-Scale DNN Processor for Real-Time AI," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 1–14.

[25] Huawei, "FPGA Accelerated Cloud Server on Huawei Cloud," https://www.huaweicloud.com/en-us/product/fcs.html, [Retrieved: June 18, 2020]. 1

[26] W. Zhao, H. Fu, W. Luk, T. Yu, S. Wang, B. Feng, Y. Ma, and G. Yang, "F-CNN: An FPGA-based Framework for Training Convolutional Neural Networks," in *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2016, pp. 107–114. 1, 2

[27] T. Geng, T. Wang, A. Sanaullah, C. Yang, R. Patel, and M. Herbordt, "A Framework for Acceleration of CNN Training on Deeply-Pipelined FPGA Clusters with Work and Weight Load Balancing," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 394–3944. 2

[28] H. Zeng and V. Prasanna, "GraphACT: Accelerating GCN Training on CPU-FPGA Heterogeneous Platforms," in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2020, p. 255–265. 1

[29] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional Architecture for Fast Feature Embedding," in *Proceedings of the 22nd ACM International Conference on Multimedia (MM)*, 2014, p. 675–678. 1, 4

[30] L. Bottou, "Large-Scale Machine Learning with Stochastic Gradient Descent," in *Proceedings of COMPSTAT'2010*, 2010, pp. 177–186. 1

[31] C. Luo, M. Sit, H. Fan, S. Liu, W. Luk, and C. Guo, "Towards Efficient Deep Neural Network Training by FPGA-Based Batch-Level Parallelism," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 45–52. 2

[32] S. Fox, J. Faraone, D. Boland, K. Vissers, and P. H. W. Leong, "Training Deep Neural Networks in Low-Precision with High Accuracy Using FPGAs," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019, pp. 1–9. 2

[33] K. He, B. Liu, Y. Zhang, A. Ling, and D. Gu, "FeCaffe: FPGA-Enabled Caffe with OpenCL for Deep Learning Training and Inference on Intel Stratix 10," in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2020. 2

[34] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778. 4