LCP: A Low-Communication Parallelization Method for Fast Neural Network Inference for IoT

Ramyad Hadidi*[‖], Bahar Asgari[‡‖] Jiashen Cao[†], Younmin Bae[†], Da Eun Shim[†], Hyojong Kim[†], Sung-Kyu Lim[†], Michael S. Ryoo[§], Hyesoon Kim[†]

*Rain AI, [†]Georgia Tech, [‡]University of Maryland College Park, [§]Google & Stony Brook University

[‖]This work was done when authors were affiliated with Georgia Tech.

***Abstract*—Deep neural networks (DNNs) have stimulated research in diverse edge applications including robotics and Internet-of-Things (IoT) devices. However, IoT-based DNN inference poses significant challenges due to resource constraints. Further, as communication is costly, taking advantage of other available IoT devices by using data- or model-parallelism methods is not an effective solution. We introduce a low-communication parallelization (LCP) method to minimize communication overhead in distributed IoT systems. LCP models consist of multiple, largely-independent, narrow branches, providing enhanced distribution and parallelization opportunities while reducing memory and computational requirements. Implemented on AWS instances, Raspberry Pis, and PYNQ boards, as well as a customized 16mW 0.107mm^2 ASIC @7nm chip, LCP models yield maximum and average speedups of 56× and 7×, compared to original models, which could be improved by incorporating common optimizations such as pruning and quantization.**

***Keywords*– IoT, DNN, Inference, Parallel, Distributed, FPGA**

I. Introduction & Motivation

Deep neural networks (DNNs) have revolutionized many fields including Internet-of-things (IoT) systems. Yet, executing computationally heavy DNN inference locally in isolated networks (*e.g.*, smart homes, drones [1]) remains a challenge [2]. In these cases, acceptable accuracy, standalone operation, and unified ownership are key. The conventional solution to heavy DNN inference computations is cloud-based offloading. However, this approach has limitations: unavailability (*e.g.*, no Internet access), reliance on variable network latency, and scalability issues. Also, privacy concerns and personalization push for local inference. However, local inference demands high resources, clashing with the energy and computational constraints in IoT devices.

The Current Approach & Key Challenge: Existing methods enable local DNN inference by distributing computations among idle IoT devices using data- or model-parallelism. Data parallelism improves throughput by duplicating the model on each device for separate inferences, but requires concurrent inputs. Model parallelism distributes the model across devices for the same inference, but is limited by communication overhead and inter-layer data dependencies. An ideal method for IoT devices should minimize communication overhead and memory and computation requirements per node, but no existing distribution methods achieve all these goals.

Our Solution: To address the aforementioned challenge, we propose a low-communication parallelization (LCP) method that enables the following: *(i) Reduces Communication:* LCP models replace a wide, deep model with several narrow ones, reducing communication requirements as they only communicate for input and pre-final activations (see Table I). *(ii) Lowers Compute & Memory Footprints:* Fewer connections in LCP models lead to fewer parameters and lower computational demands compared to model-parallelism counterparts (Table I). *(iii) Enables Inter-Layer Parallelism:* The independent narrow branches in LCP models allow for inter-layer parallelism, unlike model parallelism which is restricted by inter-layer dependencies. *(iv) Recovers Accuracy Without Extra Parameters:* Any potential accuracy loss due to model splitting can be recovered by slightly increasing the branch size, but this still results in fewer overall parameters due to the reduction in unnecessary communication. LCP operates in conjunction with existing techniques like weight pruning and quantization that decrease model computation demands. LCP facilitates model distribution and parallelism in distributed systems, while other techniques implement accuracy/performance tradeoffs on individual nodes. These approaches can be applied to each branch in our method (see §IV-C), meaning LCP complements them.

TABLE I
Methods for distributing inference computations.

	Data Parallelism	Model Parallelism	Target	LCP
Memory Per Device	DNN	$\frac{1}{n}$DNN	$\frac{1}{n}$DNN	$\leq \frac{1}{n}$DNN
Communication Per Inference	IN/OUT	Intermediates +IN/OUT	IN/OUT	$\approx$ IN/OUT
Computation Per Device	DNN	$\frac{1}{n}$DNN	$\frac{1}{n}$DNN	$\leq \frac{1}{n}$DNN

DNN: Metrics associated with the entire model; n: Number of devices.

Experiments Overview: (1) We create and evaluate LCP models using image-recognition DNNs on various datasets (MNIST, CIFAR10/100, Flower102, and ImageNet) including all MLPerf image-recognition models, resulting in a total of 53 training evaluations. **(2)** We implement our method on three different distributed systems: a network of up to 10 Raspberry Pis (RPis), two PYNQ boards, and up to eight AWS instances, using RPis due to their widespread use in IoT applications. **(3)** We assess the performance of LCP on customized hardware. In addition to optimizing models based on hardware constraints, we modify the TPU architecture to be latency-optimized, suitable for IoT applications, and implement it on a Xilinx FPGA. **(4)** We evaluate the area and power efficiency of our tailored hardware using ASAP 7 nm for integration into IoT.

Contributions: Our contributions are as follows:

- We propose the first DNN parallelization to reduce the communication overhead for distributed inference for IoT.
- We generate LCP models, with inter-layer parallelism for fast inference at small memory and computation footprints.
- We investigate the impact of hardware/software co-design on inference performance, by tailoring the hardware of TPU for optimizing single-batch inference latency, and implement it on a small FPGA and as a tiny 0.107mm^2 low-power chip consuming only 16mW.

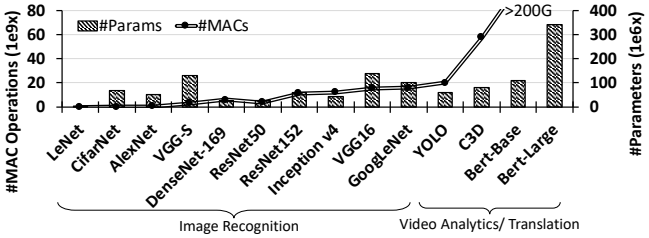

Fig. 1. DNNs #MAC operations/inference and parameters.

II. Challenges

We first explain inevitable resource limitation for executing DNNs causing the single device Pareto frontier. Then, we summarize current distribution methods and their limitations, causing straggler problem and limited scope of parallelism.

Resource Limitation & Pareto Frontier: DNN models comprise numerous layers, with custom weights learned during back-propagation training. In inference, feed-forward computations use these static learned parameters on batched inputs. The most resource-demanding layers, usually fully-connected and convolution layers, possess a significant amount of multiply-accumulate operations and parameter sizes as shown in Figure 1. More recent models incorporate more parameters and perform more computations for improved feature understanding over their predecessors. *In short, this trend of modern models will inevitably surpass the capabilities of any resource-constrained device.*

IoT platforms face resource constraints. Figure 2 shows the latency per image for state-of-the-art image recognition models on RPi, despite optimizations like pruning, quantization, low-precision inference, and handcrafted models, optimized for ARMv8 architectures using the ELL tool. The Pareto frontier sets the upper limit of single-device performance. High-accuracy models typically exhibit latencies above 400ms, and in most cases, it exceeds 100ms. This situation worsens with larger, more complex DNNs beyond image-recognition, indicating the challenges in executing such computations on a single IoT device. *In other words, even after applying all optimization techniques for DNNs, the single device Pareto frontier limits the widespread applicability of DNNs in several IoT domains necessitating distribution and parallelization.*

Current Distribution Methods: **(1)** Data parallelism (Figure 3a) applies to independent inputs but does not serve IoT environments due to its latency, throughput focus, and unchanged per-node computation and memory footprint (Table I). **(2)** Model-parallelism (Figure 3b) divides inference computations for the same request but suffers from communication overhead and single-chain dependency that limits parallelism scope. Figure 4 presents a simple example for distributing a fully connected (`fc`) layer, illustrating two extremes of model parallelism: Input and output splitting. *Although model parallelism reduces the compute and memory footprint per node; the single-chain dependency between consecutive layers limits the parallelism scope within a single inference and causes communication overhead.* **(3)** SplitNet [3] (Figure 3c) manually splits the model based on dataset semantics in intermediate to final layers, leading to issues including imbalanced workload and high communication overhead.

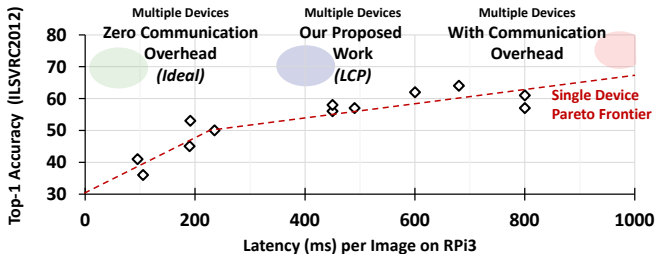

Fig. 2. Latency-Accuracy Pareto Frontier – Single device: Latency per image on RPi3 for ILSVRC models with the optimized platform-specific compilation ELL tool. Multiple devices: Breaking the single device Pareto frontier, but with significant communication overhead.

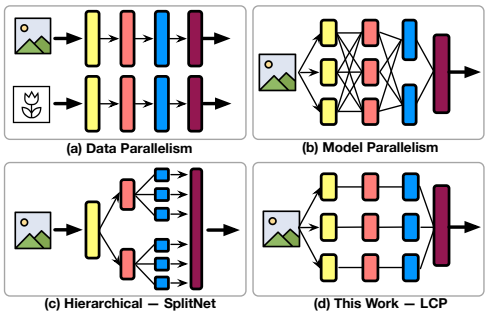

Fig. 3. Overview of distribution/parallelism methods.

Communication Overhead & Limited Parallelism: Current distribution methods suffer from high communication overhead and limited parallelism due to single-chain dependency between layers. This results in straggler problems, especially in wireless IoT devices. For instance, Figure 5 shows latency in a distributed system of six RPis executing AlexNet with model parallelism. Average delay is approximately twice as long as the bounded computing time. Figure 6a illustrates the interconnections and communication overhead in VGG-S with model parallelism. Despite compression techniques, the number of connections remains unchanged.

The single-chain dependency between consecutive layers limits the available parallelism that could be harvested by the aforementioned methods. The limitation is that after the computations of a single/few layer(s) are done, the intermediate results must be merged before being forwarded to the next layer. Such merging acts as a global barrier, which similar to parallel programming, limits the gained performance speedup. *In summary, with parallel execution on multiple devices, ideally, we could pass the frontier in Figure 2. However current distribution methods are limited by the communication overhead and the inherent inter-layer data dependency. The next section proposes LCP models, which significantly reduce communication and allow inter-layer parallelism.*

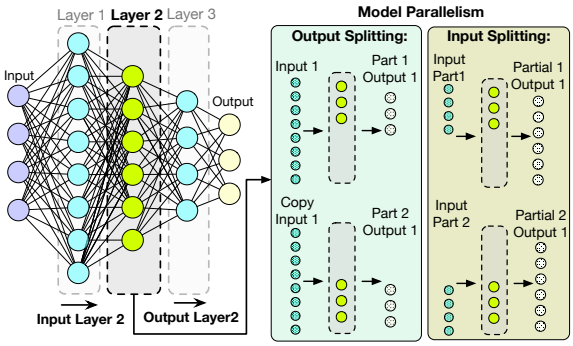

Fig. 4. Model parallelism for a fully connected layer.

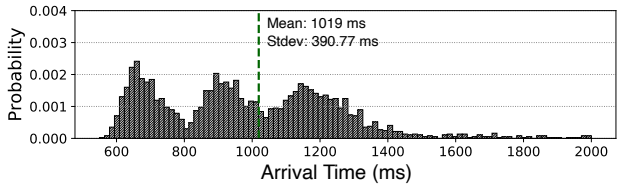

Fig. 5. Histogram of prediction latencies on a six RPi system executing AlexNet with model parallelism (§IV-B).

III. LCP For Fast Inference

We propose the LCP method, which replaces a large model with several narrow branches, communicating only for input and pre-final activation (Figure 3d), as illustrated in Figure 6b for a two-branch LCP model of VGG-S. This section details the LCP model design and its key low-communication features, followed by a discussion on tailoring a systolic architecture for IoT computing.

A. Tailoring Models

Design Procedure: Figure 7 describes the design procedure of LCP models. We start by inputting the DNN model and its per-layer memory and computation footprints. Similarly, we input the specification of the hardware, such as memory size, computation capability, and any overhead associated with executing a DNN on our hardware. For instance, several DNN frameworks have a memory overhead because of the framework. The splitter procedure, described in Procedure 1, in a while loop, splits the model, cuts the connection, and measures the approximate footprints of each branch. The $\text{Division}_{\text{Factor}}$, a hyperparameter, defines the granularity of division/splitting. Here, we assume the $\text{Division}_{\text{Factor}}$ of two, but any number is viable. The loop exits when a single branch is fitted on a device (both memory and computation wise). If the number of devices is fewer than the number of branches, the execution is still possible, but will be inefficient. Then, we remove non-branch connections in a simple operation that keeps only one connection per layer. The derived model from the splitter is the *split-only model*. By training the split-only model and testing it, we measure its accuracy. The split-only models have fewer parameters and MAC operations than the original models (Table II) in total. Hence, after distribution, each branch has less computation and memory footprint than its model-parallelism version.

As a result of fewer number of parameters and removing several connections, a slight accuracy drop in split-only LCP models is expected. Depending on the accuracy requirement of

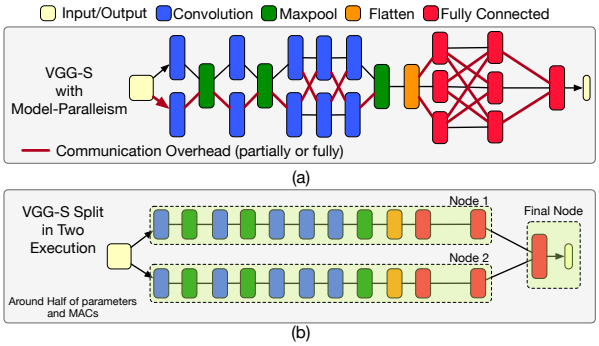

Fig. 6. VGG-S (a) model parallelism and (b) LCP versions.

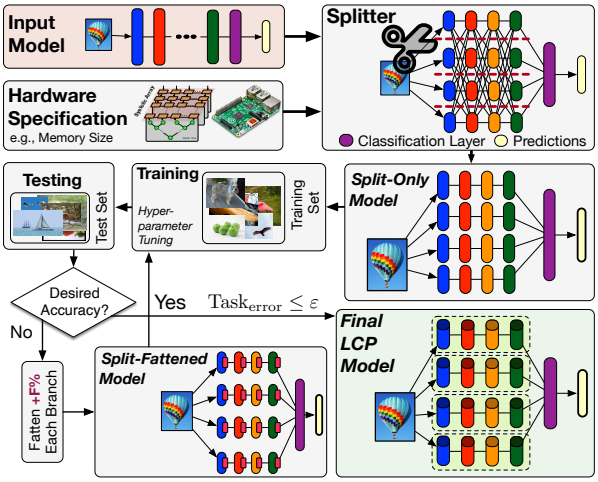

Fig. 7. Design Procedure of LCP models.

the task, we either fatten each branch by $F\%$, a hyperparameter, or output the resulted model. We assumed a maximum of 3% bound for $\text{Task}_{\text{error}}$. Fattening each branch by $F\%$ is done by increasing the number of channels and output features of convolution and fully connected layers of the split-only model, respectively. Note that theses new *split-fattened models* are fattened within each branch. Thus, even with a high fattening percentage, still they have fewer parameters and MAC operations than the original model (see Table III). When the accuracy is in the acceptable error range for our task, $\text{Task}_{\text{error}}$, we output the model architecture and its weights. It is expected that with similar number of parameters after fattening, LCP models achieve the same level of accuracy [4]. We showcase LCP models in §IV-A covering MLPerf.

Key Features of LCP Models: LCP models are designed by considering their underlying computation domain and have the following key features to address the challenges discussed in §II: **(1)** LCP models only communicate for input and pre-final activation. Therefore, they significantly reduce communication overhead in a distributed system. Additionally, the low communication load per inference helps with the straggler problem. This is in contrast to model parallelism, which highly depends on communication among all the intermediate layers; **(2)** LCP models split the size of a layer, so the total parameter size and computation complexity of the model are reduced. Therefore, they require fewer parameter sizes, less computation complexity, and no communication between the nodes for intermediate layers. These lower memory and

Procedure 1: LCP Splitter (in Figure 7)

Input : DNN: Layer configurations $[0:n]$
DNN_{Mem}, DNN_{MAC}: DNN memory and computational footprints
$\text{Division}_{\text{factor}}$: Division Factor for splitting
Dev_{Mem}, Dev_{MAC}: Hardware specification
Output: DNN: Layer configurations $[1:n]$

```
Split(DNN, DNN_Mem, DNN_MAC, Division_factor, Dev_Mem, Dev_MAC)
    Mem_fit ← 0; MAC_Mac ← 0;
    while not Mem_fit and not MAC_Mac do
        Mem_fit ← DNN_Mem < Dev_Mem
        MAC_Mac ← DNN_MAC < Dev_MAC
        for layer [0..n − 1] in DNN do
            layer.width ← layer.width/ Division_factor
        RemoveNonBranchConnections(DNN)
    return <DNN>
```

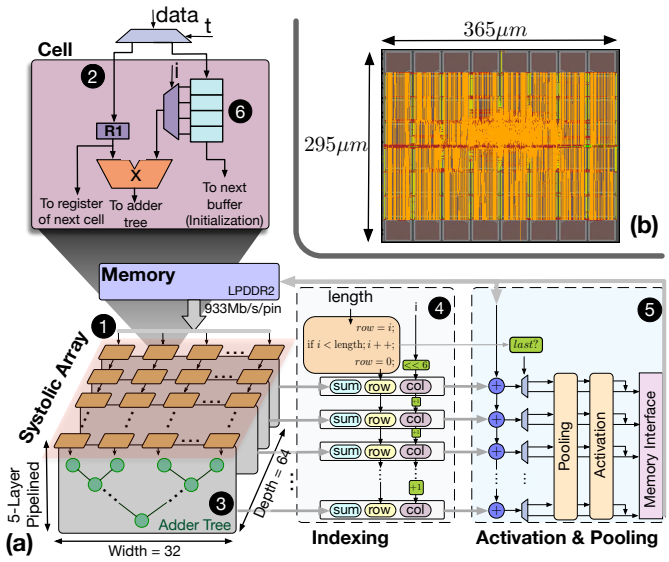

Fig. 8. Details of Tailored Hardware for IoT: (a) Microarchitecture overview, and (b) Layout of ASIC design at 7nm.

computation footprints allow IoT devices to efficiently operate within their limited resources (*e.g.*, no swap space activities due to limited memory); **(3)** LCP models replace the original wide model with several narrow independent branches. Since the computations of branches are independent, in contrast to the single-chain of dependency in the original model, the scope of parallelism is not limited with each layer anymore. In other words, LCP models go beyond intra-layer parallelism.

B. Tailoring Hardware

Our work also optimizes fast inference under resource constraints and costly communication, using a tailored hardware microarchitecture for DNNs. Our microarchitecture, shown in Figure 8a, resembles systolic arrays in TPU and can be implemented on small FPGAs or tiny low-power chips (i.e., 0.107 mm^2 as shown in Figure 8b). We arrange systolic array cells in a 32x64 array ❶, reducing connections by linking only the first row to memory. Each cell of the first row is only connected to one data stream line ❷. To optimize data flow, we partition the streaming operand into blocks of width 32, and split the stationary operand into 32×64 blocks. These blocks are then sequentially mapped to memory addresses. Our design, connected to *LPDDR2* memory, results in a peak throughput of 217.6,GOPs/s. We implement three key modifications: (1) Adder Trees: We use adder trees instead of MAC-based arrays, reducing latency from O(n) to O(log(n)) ❸. (2) Simple Indexing Logic: Our data-driven model, with indexing logic ❹, manages data flow and operation end signals. By comparing the length and index (i), the end of the operations in the current layer is detected. The end of the current layer signals the start of activation and pooling functions for that layer ❺. (3) Buffering Stationary Operands: To reduce latency and easy context switching, we integrate a buffer ❻ at each cell for stationary operands.

IV. Experimental Studies

This section presents our experiments on generating LCP models and their deployment on RPi, TVM-enabled PYNQ boards, and AWS instances. We also discuss FPGA implementation for IoT and ASIC chip design evaluation. Details for each experiment are provided at the start of their subsections.

A. Generating LCP Models

Training Specifications: We train all the models, including the original model, from scratch to conduct a fair comparison (normalization layers are included). The training is done with an exponential learning rate with a decay factor of 0.94, initial learning rate $1e{-2}$, number of epoch per decay of two or 10, a dropout rate of 50%, and L2 regularization with weight decay of $5e{-4}$. We use ADAM optimizer with $\beta_1 = 0.9$ and $\beta_2 = 0.99$. All biases are initialized to zeros and all weights are initialized with a normal distribution of mean 0 and a standard deviation of $4e{-2}$. All of our models are trained until the loss is flattened or least for 12 epochs. Test and accuracy measurements are done on at least 10% of datasets that have never been used in training to provide an unbiased evaluation of the model. For LCP, the $\text{Division}_{\text{Factor}}$, F, and ε, are 2%, 10%, and $\approx$3%, respectively.

Datasets: We use the following datasets: (1) MNIST, which contains 70k grayscale handwritten 28x28 images in 10 classes; (2) CIFAR10, which contains 60k colored 32x32 images in 10 classes; (3) CIFAR100, which contains 60k colored 32x32 images in 100 classes; (4) Flower102, which contains 16,378 colored 224x224 images of flowers in 102 classes; and (5) ImageNet, which contains 1.33 M colored 224x224 images in 1000 classes.

Models: We use the representative model for each dataset, LeNet, LeNet-FC, VGG-S, CifarNet, VGG16, AlexNetv2, ResNet-18/50, and MobileNet. We cover all image-recognition models in MLPerf. In total, for brevity, we only report 53 instances of training results to show LCP extensibility using five datasets and nine models. Our additional results (not reported) with ResNet-34, DenseNet, and DarkNet19 confirms extendibility. Simple sequential DNNs serve as a basis to confirm our method, while ResNets and MobileNet showcase LCP with modern models.

Split-Only Models: For split-only models, we use $\text{Division}_{\text{Factor}}$ of two, which results in models with two, four, and eight branches. Except the width, defined as output features in fully connected layers and the number of output channels (i.e., filters) in convolution layers, the rest of the parameters are similar to the original model as Splitter Procedure 1 only touches widths. Table II lists the training results. Figure 9a illustrates the accuracy difference of our models, shown in Table II. As shown, the maximum accuracy drop is around 5% for CifarNet. Note that this accuracy drop occurs when we reduced the parameter size of our model extensively (around $1/8$). Figure 9b and c show reduction in the number of parameters and computation compared with the original DNN model; as seen, each split reduces both by about split_{factor} times. This is because each convolution

TABLE II
RESULTS OF SPLIT-ONLY LCP MODELS.

Model Name	Dataset	Layers†	Top-1 Accuracy	# Param	# MAC Opr.
LeNet-FC*	**MNIST**	**3fc**	**97.95**	**266.6k**	**266.2k**
LeNet	**MNIST**	**2fc-3c-2p**	**98.76**	**61.7k**	**61.5k**
LeNet-split2	MNIST	3fc-6c-4p	98.86	31.5k	30.5k
LeNet-split4	MNIST	5fc-12c-8p	98.93	16.1k	16.0k
LeNet-split8	MNIST	9fc-24c-16p	98.81	8.8k	8.5k
CifarNet*	**Cifar10**	**2fc-2c-2p-2n-1d**	**80.72**	**797.97k**	**14.79M**
CifarNet	**Cifar100**	**2fc-2c-2p-2n-1d**	**52.87**	**815.34k**	**14.81M**
CifarNet-split2	Cifar100	5fc-4c-4p-4n-2d	51.22	410.48k	9.33M
CifarNet-split4	Cifar100	9fc-8c-8p-8n-4d	48.48	208.05k	6.59M
CifarNet-split8	Cifar100	17fc-16c-16p-16n-8d	47.98	106.85k	5.23M
VGG-S*	**Cifar100**	**3fc-5c-2p-1n-2d**	**50.33**	**76.15M**	**154.09M**
VGG-S	**Flower102**	**3fc-5c-3p-1n-2d**	**88.14**	**60.79M**	**1.85G**
VGG-S-split2	Flower102	5fc-10c-6p-2n-4d	89.31	30.50M	1.01G
VGG-S-split4	Flower102	9fc-20c-12p-4n-8d	87.55	15.26M	591.65M
VGG-S-split8	Flower102	17fc-40c-24p-8n-16d	85.66	7.64M	382.51M
ResNet-18	**ImageNet**	**18c-2p-17n**	**70.68**	**11.69M**	**1.82G**
ResNet-18-split2	ImageNet	35c-3p-34n	69.85	6.11M	0.98G
ResNet-18-split4	ImageNet	69c-5p-68n	68.07	3.32M	0.55G
ResNet-18-split8	ImageNet	137c-9p-136n	66.76	1.93M	0.34G

† fc: fully-connected, c: convolution, p: pooling, n: normalization, and d: dropout.
* Detailed results are removed for brevity, refer to Figure 9. The results follows the same trend.

and fully connected layer in the split version create fewer outputs; therefore, the next layer requires fewer parameters. In the next section, we restore the accuracy of LCP models with split-fattened models.

Split-Fattened Models Accuracy is a defining factor in several applications. Thus, we provide a remedy to restore the accuracy of split-only models. By adding more parameters to each branch, we aim to create larger layers in the split-only models. To do so, for each layer (but classification layer), in every branch, we increase the width by a fraction. Fattening by 20% means the output size in each layer is increased 1.2x. We fatten every branch in 10% steps as Procedure 1 shows. Our experiments focus on split8, which have the highest accuracy drops. Figure 10 shows a summary of these models. As seen, 40% split-fattened models have higher accuracy than the original model while having fewer parameters and MAC operations. On average (for 30% and 40% models), with 4.61x–3.81x fewer parameters and 2.95x–2.5x fewer MAC operations, split-fattened models achieve accuracy within our error bound of 3%, $\text{Task}_{\text{error}}$, while they jointly optimize memory, computation, and communication for IoT.

ImageNet Models: Table III illustrates the results of ImageNet models. For the sake of brevity, we only show split8 and one fattened model. As shown, `f40` models restore the accuracy

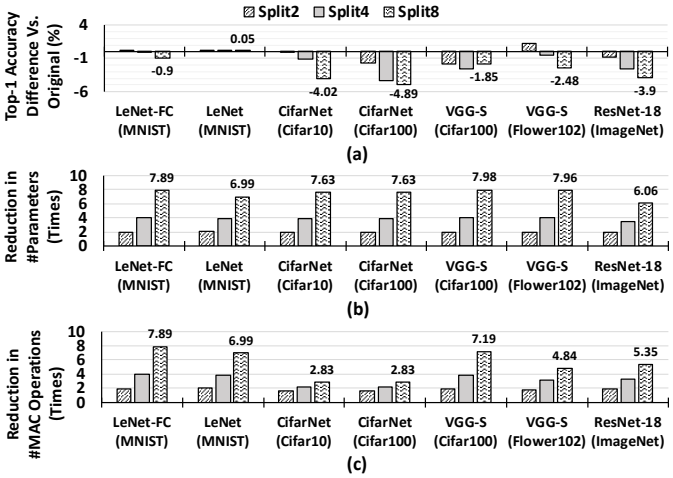

Fig. 9. Split-Only Models: (a) Accuracy, (b) reduction in the number of parameters, and (c) reduction in the number of MAC operations in comparison with the original model.

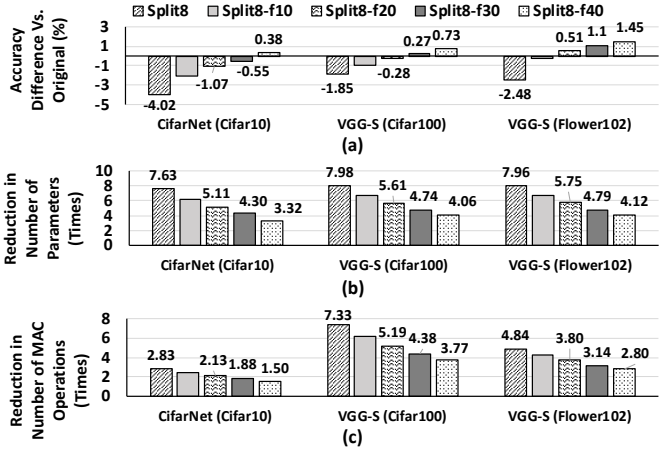

Fig. 10. Split-Fattened Models – Common visual models (a) Accuracy difference, (b) reduction in the number of parameters, and (c) reduction in the number of MAC operations in comparison with the original one (Table II).

within 3% of the original model. The tradeoff for 3% accuracy loss is about 4x fewer parameters, 4x fewer computations, and 8x less communication load (vs. model parallelism). Figure 11 presents a comparative analysis for the communication load between distributed original models with model parallelism and distributed LCP models. Since LCP models avoid communication between their branches, the communication load is reduced significantly. In short, although split models are more complex than the original models in terms of the number of layers and connections, they achieve more parallelism with less communication load.

B. Exploring Performance on RPis, PYNQs, and AWS

RPi Experiments Setup: To study the benefits of LCP models versus only model-parallelism methods, we deploy several models on a distributed system of Raspberry Pi 3s (RPis), the specifications in Table IV. On each RPi, with the Ubuntu 16.04 operating system, we use TensorFlow and Apache Avro, a remote procedure call (RPC) and data serialization framework, for communication between RPis. We measure power using a USB digital multimeter. A local WiFi network with the measured bandwidth of 62.24 Mbps and a measured client-to-client latency of 8.83 ms for 64 B is used. All the real-world experiments are full-system measurements with all overheads included without any simulations/estimations.

RPi Performance & Energy: Figure 12 presents latency of inference per image on RPis. On a single device, AlexNet has 2.8 seconds latency, while VGG16 achieves 9.4 seconds latency. By deploying model-parallelism variants of the models on four and eight RPis, we achieve a maximum of 0.42s latency, a 6.6x increase, for AlexNet. But, for VGG16, on four RPis, we observe a slowdown, which is caused by high communication latency. LCP variants of split4 and split8 can reach up to 115 ms and 400 ms latency per image for AlexNet

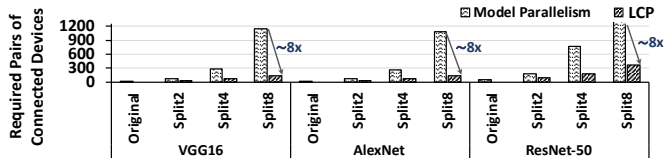

Fig. 11. Communication reduction with LCP models compared to model parallelism (required pairs of connections).

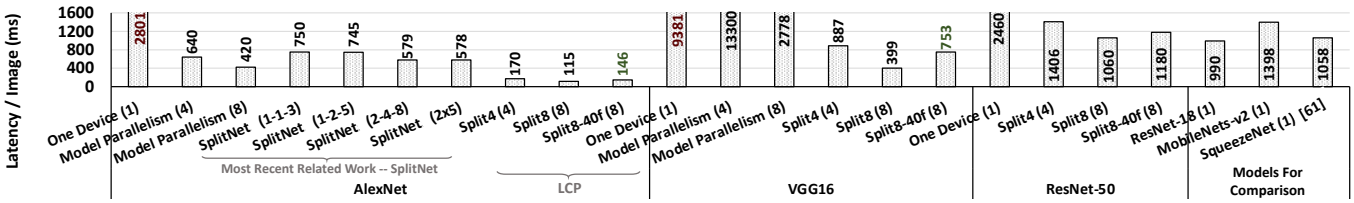

Fig. 12. Latency per image: Model-parallelism, SplitNet [3], and LCP models on RPi (number in parenthesis is #devices).

TABLE III
RESULTS OF IMAGENET LCP MODELS.

Model Name	Dataset	Top-1 Acc.	Top-5 Acc.	# Param.	# MAC MAC Opr.
AlexNet	**ImageNet**	**57.02**	**80.32**	**50.3M**	**678.97M**
AlexNet-split8	ImageNet	49.03	73.10	6.32M	145.37M
AlexNet-split8-f40	ImageNet	54.68	77.06	12.11M	244M
VGG16	**ImageNet**	**70.48**	**90.02**	**138.36M**	**15.47G**
VGG16-split8	ImageNet	58.67	81.54	7.64M	2.01G
VGG16-split8-f40	ImageNet	67.24	89.23	33.78M	3.87G
ResNet-50	**ImageNet**	**75.4**	**93.1**	**22.80M**	**4.87G**
ResNet-split8	ImageNet	61.79	81.22	5.42M	0.88G
ResNet-split8-f40	ImageNet	72.12	92.19	8.60M	1.18G
MobileNet	**ImageNet**	**71.7**	**90**	**4.24M**	**4.86G**
MobileNet-split8	ImageNet	59.68	83.23	1.12M	0.93G
MobileNet-split8-f40	ImageNet	68.05	89.12	2.12M	1.34G

For `[model_name]-f[number]`, number represent the percentage of fattening.

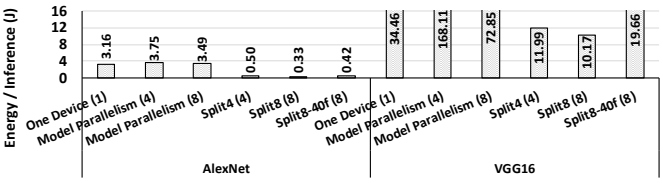

Fig. 13. All devices energy per inference: Model-parallelism, and LCP on RPi (number in parenthesis is #devices).

and VGG16, respectively. This is because LCP models are lightweight and parallelizable and have low communication. Figure 13 shows measured energy per inference for RPi implementations. To compare with previous related work, SplitNet [3], Figure 12 presents the performance of SplitNet models for AlexNet with different configurations. As seen, the performance is worse than LCP models. This is because SplitNet creates more merging/synchronization points with its tree-structured model design. The resulting model exponentially introduces more merging/synchronization with increased depth, which also does not equally split all the layers (causing load balancing issues). Finally, SplitNet performs parallelization based on dataset semantics, which means every dataset and model needs to be manually split. §II provided more reasons on this performance difference.

TABLE IV
SPECIFICATION OF RPI, PYNQ FPGA, AND AWS.

Raspberry Pi 3B+				
CPU	1.2 GHz Quad Core ARM Cortex-A53			
Memory	1 GB LPDDR2 SDRAM @ **933Mb/s/pin**			
Die Size	$\approx 196mm^2$ @ **28 nm**			
Edge FPGA (Zynq Artix 7 XC7Z020)				
Utilization		DSP48E	FF	LUT
	#Unit	96	5427	2343
	%	44	5	4
Static Power	0.121 W			
Dynamic Power	Signals: 0.009 W	Logic: 0.003 W		
AWS				
AWS Instance Specification	T2.micro 1 vCPU, 1 GB Memory, 64 GB Storage			

TVM Experiments on PYNQ Boards: As a real-world example for IoT FPGA implementation, we use TVM [5] on the PYNQ board. PYNQ is designed for embedded applications. We use the TVM VTA stack on the PYNQ as the architecture (RISC-style instructions) and only change the models (ResNet-18 vs. LCP ResNet-18 Split2 with < 1 accuracy drop). In this way, we can measure the benefits of LCP models without relying on any special tailored hardware. Our performance result shares the entire system pipeline performance, from a live camera feed to prediction output on two boards versus one board. Figure 14a shows a 2.7× speedup, including all communication and system overheads, network latency, and jitter because LCP models are parallelized on two devices and, in total, they have lower computation and memory footprints. The measured reduction in memory footprint is shown Figure 14b.

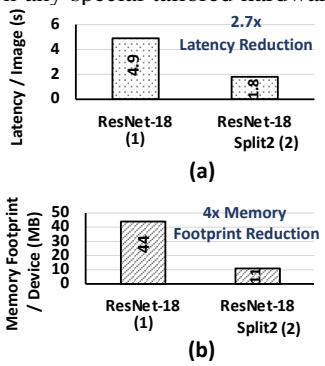

Fig. 14. TVM Experiments: (a) Latency per image, (b) memory footprint per device (number in parenthesis is #devices).

AWS Experiments: To see the reduced communication and distributed execution benefits of LCP models further, we deploy AlexNet, VGG16, and ResNet-50 models on AWS T2.micro instances with only one vCPU and 1 GB memory per instance. Figure 15 presents the derived statistics. In all cases, LCP models not only reduces the average latency but also significantly reduce maximum latency. Splits four and eight have lower speedup compared with our RPi experiments because all the 4/8 instances are not hosted on the same machine; thus, the communication cost is higher than the usual IoT-specific cases that this paper targets.

C. FPGA Experiments

FPGA Experiments Setup: We implement our tailored microarchitecture on a ZYNQ XC7Z020 FPGA targeting PYNQ-z1 boards. We use Xilinx Vivado HLS for implementation and verify the functionality of our implementation using regression

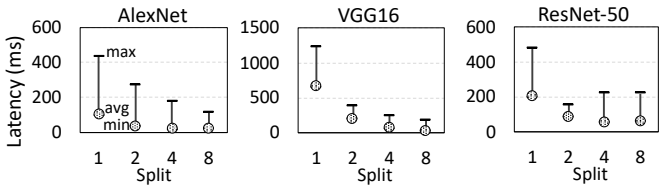

Fig. 15. Average, minimum, and maximum latency of distributed LCP on AWS T2.micro instances with 1 vCPU and 1 GB memory per instance.

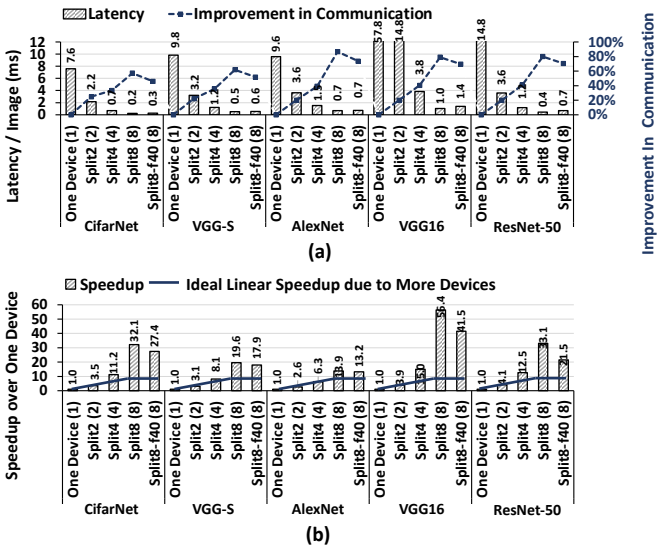

Fig. 16. FPGA with tailored hardware latency and speedup: (a) Latency per image, (b) speedup over one device (number in parenthesis is #devices).

tests. We use relevant *#pragrma* as hints to describe our desired microarchitectures in C++. We synthesize and implement our design using Vivado and report post-implementation (*i.e.*, place & route) performance numbers and resource utilizations. Inputs and output of our design are transferred through the AXI stream interface. The clock frequency is set to 100 MHz. Communication for multiple devices is estimated with the network provided in §IV-B.

FPGA Performance: Figure 16 shows the experiment results for our IoT-tailored hardware. The latency per image is shown in Figure 16a, with improvement in communication overhead versus model-parallelism methods (86% and 60% for 8split and 4split). Depending on the model, the inference per latency on a single device is between 4–29ms; a 221–325× speedup compared to RPi results for AlexNet and VGG16. Our designed LCP models achieve acceptable performance for IoT computing, which is 10s of inferences per second, around 1–10ms. As observed, the accuracy loss of our split-only models can be easily restored by fast split-fattened models of `f40` with a negligible performance overhead (maximum of 20 ms). Figure 16b illustrates the speedup over one device. The ideal linear speedup shows the ideal scaling speedup with more available devices. As shown, we achieve superlinear speedups. An important parameter in scaling concerns how the *overheads* scale. The superlinear speedup stems from the dramatic reduction of communication overhead as parallelism increases. In traditional data and model parallelism, such overhead increases, which causes sublinear speedup. Figure 17 compares latency per image for LCP and model parallelism. On average, LCP models are 3.76x, 8.89x, and 7.17x faster than their model-parallelism counterparts for AlexNet, VGG16, and ResNet-50 (4 and 8 devices), respectively. LCP achieves a maximum and average speedups of 56x and 7x, compared to the originals (Figure 18, base bars).

Quantization & Pruning: Techniques that reduce the footprint of DNNs can be applied to each individual LCP branch. Basically, the target output for each LCP branch is now

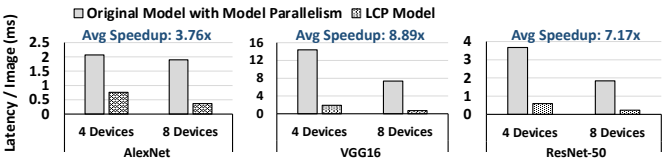

Fig. 17. Latency per image for IoT FPGA with tailored hardware comparing LCP vs. model parallelism.

its pre-final activations during optimizations. We study the benefits of lossless quantization and structured pruning on top of our LCP models. Based on our experiment, with 3.13 (<integer.fraction>) quantization, our models do not lose accuracy. Similarly, applying structured pruning [6], for which systolic arrays gain benefits, reduces the size of parameters between 40%–50% per convolution layer without an accuracy drop. Other pruning algorithms increase the sparsity of the data, which is not necessarily beneficial for systolic arrays. Figure 18 presents the speedup gained from these techniques normalized to the baseline implementation for each model, the execution performance of which shown in Figure 16a. Quantization and pruning themselves, improve the performance of the original models by 1.96x and 2.2x, respectively, and 4.31x when applied together. When quantization and pruning are combined with LCP, the overall performance speedup becomes 14.41x and 16.31x, respectively. Compared to the original models, LCP + quantization and pruning achieves up to 244x speedup (VGG16-split8), and an average of 33x.

D. ASIC Implementation

We implement the ASIC design of LCP using an Arizona State Predictive PDK (ASAP) 7nm technology node. Our tool chain includes the Synopsys design compiler (DC) for synthesis, Cadence Innovus for place and route, and Cadence Tempus for timing and power analysis. As an input to our ASIC design, we use our same Verilog code generated by Vivado HLS. Figure 8b show the layout of our chip of size 0.107 mm^2 (i.e., $295\mu m \times 365\mu m$). The memory cells shown in the figure represent the FIFO buffers, used for pipelining. Figure 19 shows the power consumption of our ASIC design. The breakdown of power consummation leading to a total 16.1 mW is listed in Figure 19a. As a comparison point, Eyeriss [7] and EIE [8] consume ≈250 mW and ≈590 mW, respectively. Besides, as Figure 19b shows, power distributes uniformly, which prevents hot spot creation.

V. Related Work

We overview DNN computation reduction methods, distribution techniques, and DNN hardware accelerators. Model-independent techniques reduce DNN computational and memory requirements without changing the architecture. Pruning [9], [10] removes nearly-zero weights while quantization or low-precision inference [11], [12] simplifies calculations. Other methods involve resource partitioning [13], weight binarization [14], [15], and hardware-aware optimizations [16]. However, some techniques reduce accuracy. With increasing IoT use, industry has created optimized frameworks such as ELL library and Tensorflow Lite. Others developed mobile-specific models [17] with efficient operations or models to

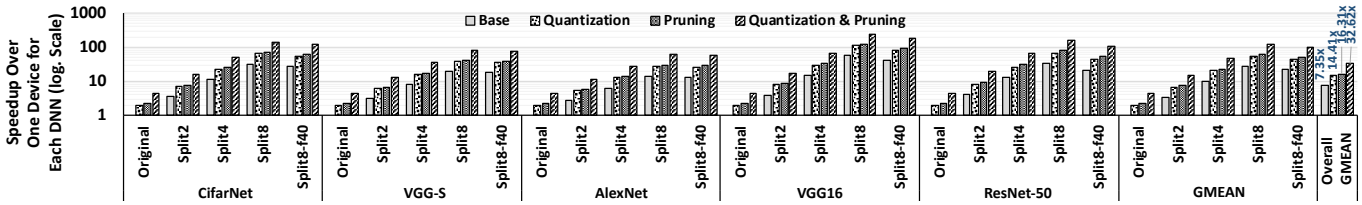

Fig. 18. FPGA with tailored hardware with lossless ($\leq 0.1\%$) quantization & structured pruning to achieve additional speedup.

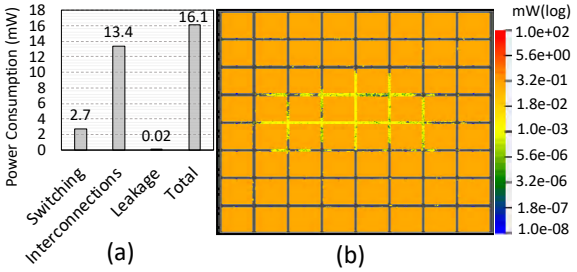

Fig. 19. Power Consumption for 7-nm ASIC Design @800MHz: (a) breakdown (b) distribution.

reduce parameters. However, they often sacrifice accuracy for efficiency [18], or lack efficient parallelism. Few papers, like SplitNet [3], focus on model parallelizability but face issues with branch imbalances and device invariance. Recently, automated design process has seen increased interest [4], [19], [20]. In [21], in follow up to LCP, we propose the relaxation of the single-chain dependency constraint in neural architecture search (NAS), facilitating higher concurrency and distribution opportunities in deep learning architectures. This approach, complemented with a new generator and transformation block, points towards a promising direction for reducing inference latency and improving computational efficiency in modern deep learning models. Distributing large DNN models has been also explored [22]–[24].

VI. Conclusions

This paper proposed LCP models, designed for efficient DNN inference in IoT systems. LCP models optimize communication while reducing memory and computation by utilizing several narrow independent branches. We presented our results on RPis, FPGAs for IoT, AWS instances, and a tailored systolic-based hardware.

References

[1] R. Hadidi, B. Asgari, S. Jijina, A. Amyette, N. Shoghi, and H. Kim, "Quantifying the design-space tradeoffs in autonomous drones," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 661–673.

[2] R. Hadidi, J. Cao, Y. Xie, B. Asgari, T. Krishna, and H. Kim, "Characterizing the deployment of deep neural networks on commercial edge devices," in *2019 IEEE International Symposium on Workload Characterization (IISWC)*, 2019.

[3] J. Kim, Y. Park, G. Kim, and S. J. Hwang, "Splitnet: Learning to semantically split deep networks for parameter reduction and model parallelization," in *ICML*. JMLR. org, 2017, pp. 1866–1874.

[4] S. Xie, A. Kirillov, R. Girshick, and K. He, "Exploring randomly wired neural networks for image recognition," in *IEEE ICML*, 2019, pp. 1284–1293.

[5] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Q. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: end-to-end optimization stack for deep learning," *arXiv preprint arXiv:1802.04799*, pp. 1–15, 2018.

[6] S. Anwar, K. Hwang, and W. Sung, "Structured pruning of deep convolutional neural networks," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 13, no. 3, p. 32, 2017.

[7] 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, 2017.

[8] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 243–254.

[9] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," in *44th International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 548–560.

[10] J. Lin, Y. Rao, J. Lu, and J. Zhou, "Runtime neural pruning," in *Advances in Neural Information Processing Systems (NIPS)*, 2017, pp. 2181–2191.

[11] U. Köster, T. Webb, X. Wang, M. Nassar, A. K. Bansal, W. Constable, O. Elibol, S. Gray, S. Hall, L. Hornof *et al.*, "Flexpoint: An adaptive numerical format for efficient training of deep neural networks," in *NIPS*, 2017, pp. 1742–1752.

[12] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *ICML*, 2016, pp. 2849–2858.

[13] J. Guo, S. Yin, P. Ouyang, L. Liu, and S. Wei, "Bit-width based resource partitioning for cnn acceleration on fpga," in *25th Annual IEEE FCCM*, 2017.

[14] F. Li, B. Zhang, and B. Liu, "Ternary weight networks," *arXiv preprint arXiv:1605.04711*, 2016.

[15] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or- 1," *arXiv preprint arXiv:1602.02830*, 2016.

[16] B. Asgari, R. Hadidi, and H. Kim, "Ascella: Accelerating sparse computation by enabling stream accesses to memory," 2020.

[17] 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 preprint arXiv:1704.04861*, 2017.

[18] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4510–4520.

[19] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *CVPR*, 2018, pp. 8697–8710.

[20] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," 2016.

[21] R. Hadidi, J. Cao, M. S. Ryoo, and H. Kim, "Reducing inference latency with concurrent architectures for image recognition at edge," in *2023 IEEE International Conference on Edge Computing and Communications (EDGE)*, 2023.

[22] R. Hadidi, J. Cao, M. Woodward, M. S. Ryoo, and H. Kim, "Distributed perception by collaborative robots," *IEEE Robotics and Automation Letters*, vol. 3, no. 4, pp. 3709–3716, 2018.

[23] R. Hadidi, J. Cao, M. S. Ryoo, and H. Kim, "Towards collaborative inferencing of deep neural networks on internet of things devices," *IEEE Internet of Things Journal*, 2020.

[24] R. Hadidi, J. Cao, B. Asgari, and H. Kim, "Creating robust deep neural networks with coded distributed computing for iot," in *2023 IEEE International Conference on Edge Computing and Communications (EDGE)*, 2023.