# An Approach Towards Distributed DNN Training on FPGA Clusters

Philipp Kreowsky*†§, Justin Knapheide*§, Benno Stabernack*†
{philipp.kreowsky, justin.knapheide, benno.stabernack}@hhi.fraunhofer.de
*Fraunhofer Institute for Telecommunications, Heinrich Hertz Institute, Berlin, Germany
†University of Potsdam, Embedded Systems Architectures for Signal Processing, Potsdam, Germany

***Abstract*—We present NADA, a Network Attached Deep learning Accelerator. It provides a flexible hardware/software framework for effectively training deep neural networks on ethernet-based FPGA clusters. The NADA hardware framework instantiates a dedicated entity for each layer. Features and gradients flow through these layers in a tightly pipelined manner. From a compact description of a model and target cluster, the NADA software framework generates specific configuration bitstreams for each particular FPGA in the cluster. We demonstrate the scalability and flexibility of our approach by planning an example CNN on a cluster consisting of three up to nine Intel Arria 10 FPGAs. To verify NADAs effectiveness for commonly used networks, we train MobileNetV2 on a six-node cluster. We address the inherent incompatibility of the tightly pipelined layer parallel approach with batch normalization by using online normalization instead. Since the presented framework is work in progress some important topics will not be discussed. Especially power efficiency measurements have not been taken, as the implementation is not yet optimized towards practical clock frequencies.**

***Index Terms*—FPGA, Network Attached Accelerator, MobileNetV2, CNN, Layer Parallelism**

## I. Introduction

Machine learning is taking over more and more aspects of our everyday life. As Deep Neural Networks (DNNs) grow in complexity and training datasets grow in size, there is an ever growing demand for computational resources. Training a large DNN to a competitive accuracy today essentially requires a high-performance compute cluster [1].

There are different approaches to the acceleration and hence parallelization of DNNs: Today, Data Parallelism (DP) is most commonly utilized and supported by most popular frameworks such as PyTorch, TensorFlow and Caffe. A batch is split into micro-batches and each is processed in parallel on a separate compute node (e.g. GPU). The downside of this approach is that, after each batch, all parameter gradients from the individual compute nodes need to be added together to compute the parameter update. This communication overhead leads to diminishing returns for large clusters. In addition, all parameters have to be stored on each compute node which poses a problem for FPGAs with limited on-chip memory.

Training DNNs using Layer Parallelism (LP), where the individual layers are processed in a pipelined manner, already yields a significant speedup on GPU clusters [2]–[4]. On FPGA clusters, FPDeep [5], which also leverages the LP approach, shows promising results. However, it only supports a 1D cluster topology and neglects any form of normalization as is used by most modern DNNs, limiting its applicability.

Data centers are increasingly being set up as heterogeneous systems, in which FPGAs are used to complement CPU and GPU-based nodes [6]. FPGA-based Network Attached Accelerators (NAA) will be used for higher integration flexibility. These accelerators can be directly integrated into an existing data center infrastructure through a network interface, and are no longer housed in a carrier system. In order to use these NAA clusters for training ML methods, it is necessary to implement architectures that can scale beyond FPGA limits and use the existing network infrastructure.

This paper introduces our Network Attached Deep learning Accelerator called NADA. NADA is a flexible framework which provides a flow from a high-level, functional description of the DNN through generating register-transfer level code for an FPGA cluster to orchestrating the training.

Our contributions include:

- We evaluate NADA's flexibility by planning one model for various clusters.
- We give an example of our high-level model description.
- We use this example to illustrate our tool flow.
- We demonstrate the viability of NADA for real-world CNNs by training MobileNetV2 on the ImageNet dataset.

The structure of the paper is as follows: Related work is discussed in Section II, followed by an overview of the different parallelization strategies for DNN training in Section III. We explain our tool flow in Section IV, evaluate our framework in Section V and finally draw a conclusion with focus on future work in Section VI.

## II. Related Work

The training of Convolutional Neural Networks (CNNs) on GPUs is a well-researched field. Popular frameworks provide a user-friendy interface which hides the workload balancing and distribution from the developer. An overview of parallel deep learning techniques is given in [1], [7]. FPGA clusters already deliver promising results for inference of DNNs [8]–[12]. Much less work can be found on FPGA-based training frameworks, especially ones targeting whole clusters.

Zhao et al. [13] presented a reconfigurable framework which they call F-CNN. It focuses on a single-FPGA implementation,

§Equal contribution

using single precision floating point and batch parallelism. They evaluated their framework by training LeNet-5 on the MNIST dataset for handwritten digit recognition.

Luo et al. [14] proposed a framework called DarkFPGA for training DNNs on a single FPGA. They also use batch parallelism, but use an eight bit fixed point number format to support low precision training. The evaluation was performed with a small VGG-like CNN on the Cifar10 dataset.

Itsubo et al. [15] used an FPGA-based 10GbE switch to implement the gradient accumulation and parameter update part of training in hardware, serving a cluster of GPU workers.

Wang et al. [5] proposed FPDeep, an FPGA-cluster-based training framework for CNNs and provided experimental results for the implementation of AlexNet and VGG16/19. They used a deeply pipelined approach of hybrid Model Parallelism (MP) and LP where they distributed the CNN model onto a cluster of FPGAs. Due to this deeply pipelined approach, they seem to avoid caching activations for the back propagation. Claiming that a 1D topology outperforms a 2D topology, they used a cluster of eight daisy-chained FPGAs to evaluate the correctness and performance of their design by mapping a part of VGG16 onto their cluster. Namely, the third to the fifth convolution layer. Using a software simulator, they evaluated the performance of their framework for larger clusters and claimed linearity up to 100 FPGAs with 250 Gb/s bidirectional bandwidth.

Lu et al. [16] used a reconfigurable approach to implement a DNN training accelerator on a single FPGA. It is centered around a reconfigurable processing element performing convolutions in eight bit fixed point and Batch Normalization (BN) cores using half precision floating point. They trained ResNet-20 on the Cifar10 dataset. Notably, they already raised concerns about the FPDeep approach facing challenges on models with BN.

Among these works, FPDeep is the only one targeting CNN training on FPGA clusters. We advance upon the state of the art by enabling CNN training on FPGA clusters while:

- Implementing directed acyclic graphs rather than just sequences of layers.
- Utilizing off-the-shelf ethernet rather than custom point-to-point connections.
- Optimizing the placement of layers across devices.

## III. Background

In classic CNN classifiers, data is fed through multiple convolution-, bias- and activation layers to extract increasingly complex features. The output of the final layer is then flattened and fed into one or more dense layers, performing the classification. The final layer of the network is typically a Softmax layer, which outputs confidence scores for each of the supported classes. CNNs can be structured as arbitrary directed acyclic graphs, with layers as nodes and edges describing which output connects to which input. We will subsequently refer to the data carried on these edges as *feature maps*.

In order to train a CNN model, the classification is performed on a labeled training dataset. In the context of training,

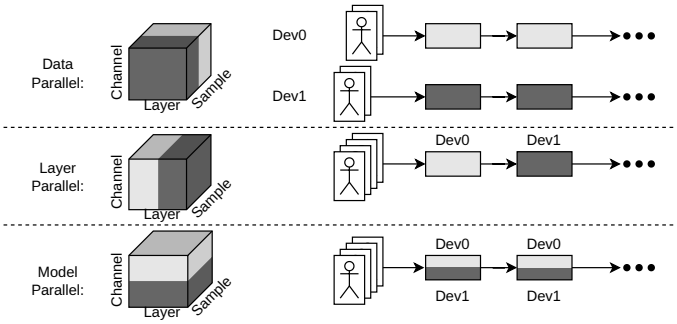


Fig. 1. The three basic parallelization approaches.

this is referred to as the *forward pass*. The loss is then calculated as the difference between the prediction and the one-hot-encoded label. The gradients of the loss with respect to the model's parameters need to be computed to perform the *parameter update* step using stochastic gradient descent [17]. This is called *gradient backpropagation* or *backward pass* and involves two steps: First, the gradients for all feature maps are calculated by applying the chain rule going backward through the model. Then, given these feature map gradients, the gradients for the model's parameters can be computed. For some layers, this involves the feature maps from the forward pass, which thus need to be buffered until they are needed. For convolution, the most computationally expensive layer in typical CNNs, the backward pass requires twice as many operations as the forward pass [1]. Thus, in addition to the much larger memory requirements, training of CNNs is also approximately three times as computationally intensive as inference.

### *Parallelization Strategies*

Following Wang et al. [5], we distinguish three ways to split the training task across multiple devices, illustrated in Figure 1 as the three dimensions of a cube.

- Data Parallelism (DP) splits the training data across the cluster. Each device runs the full model on a subset of the training set. Thus, if the subsets are already stored local to the respective devices, no communication is required for the forward and backward pass. However, because each device holds a copy of the full model, parameter updates need to be synchronized after each batch, incurring two times the total parameter size in network traffic per node.
- With Layer Parallelism (LP), the individual layers of the model are partitioned across the cluster. Features and gradients are transferred over the network during the forward and backward pass, while parameters stay local to the device they were placed on.
- Model Parallelism (MP) further splits each layer into multiple parts that are then individually placed onto separate devices. For example, a convolution with 16 input channels could be split into two convolutions of eight input channels each. Depending on the way the layer is split, the resulting parts may or may not share some or all of their parameters. If the parts have no

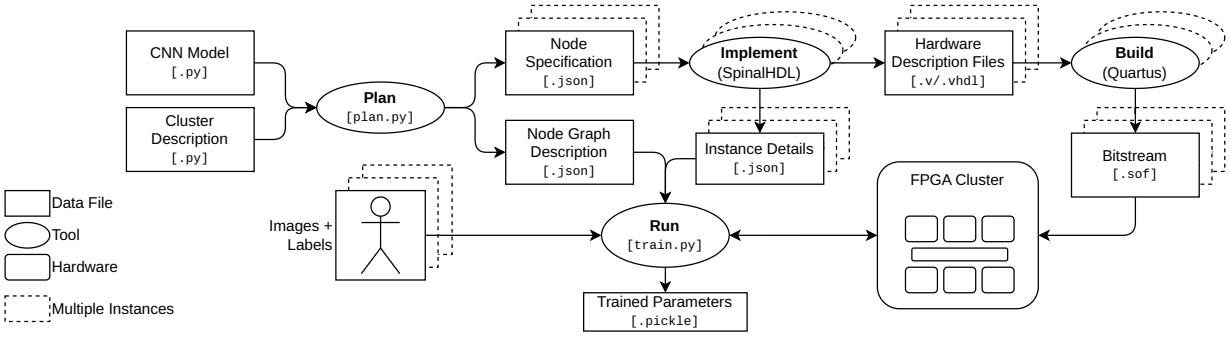


Fig. 2. NADA workflow from cluster- and model description to trained model parameters.

shared parameters at all, MP can alternatively be seen as a transformation of the CNN model before applying LP.

Note that some authors have used slightly different naming schemes. For example, *model parallelism* in [2] refers to what we call LP.

The different parallelization approaches influence the scheduling of tasks across the devices. Both DP and LP first run the forward and backward pass, then do the parameter update (and synchronization). During the parameter update and synchronization step, the available compute resources are not meaningfully utilized. Thus, any delay induced here directly costs compute efficiency. For DP, this delay consists of the parameter update itself plus the time for collecting the gradients and distributing the updated parameters. With LP, we need to wait for the pipeline to drain before applying the gradient descent. Thus, the total delay is equal to the time required for the parameter update plus the latency of the forward and backward pass.

This means that for LP to be efficient, the pipelining needs to happen with very fine granularity. If the whole pipeline must have only a few frames of latency, the delay for each layer needs to be a small fraction of a frame. We call this approach *tightly pipelined LP*. Alternatively, Narayanan, Harlap et al. [2] operate in a batch-wise manner, inducing a whole batch of delay for each stage in the pipeline. To enable this, they buffer old versions of the parameters for the backward pass. In the memory-limited environment of an FPGA cluster, this would be prohibitively expensive.

The parallelization scheme will also influence the memory footprint. This is important because accesses to external memory are an expensive resource. They are orders of magnitude more energy-intensive than accesses to on-chip memory [18]. In addition, traditional FPGA accelerator cards often provide very limited DRAM bandwidth. Because of the long delay between forward- and backward pass and the limited amount of on-chip RAM, feature maps will always need to be stored in external memory. For DP, one additional read will be incurred for the forward pass because batches of feature maps need to be buffered between layers. If parameters do not fit into the on-chip RAM of a single device, they will need to be stored in external memory as well. With LP, the combined on-chip RAM of all devices is available to store parameters and gradients. If this is not enough however, a lot of external memory bandwidth will be required due to the tightly pipelined approach. The ratio between compute complexity and parameter memory requirements changes drastically between the first and last layers of a CNN. This means that fully utilizing the combined on-chip RAM incurs either a substantial drop in throughput or an increase in network bandwidth requirements. Generally, for data parallelism, parameters are transferred over the network, while for layer parallelism, feature maps are transferred wherever two consecutive layers are placed on separate devices. Here, the potential advantage of layer parallelism is that feature map transfer occurs in parallel with computation, while parameter transfer occurs in sequence with it.

### *Online Normalization*

Batch Normalization (BN) [19] is a layer type commonly used in CNNs to improve training speed and achievable accuracy. It operates channel-wise on each batch, scaling and shifting the feature map to approximate a normal distribution around zero. Because of this batch-wise operation, it is incompatible with our tightly pipelined parallelization approach. To work around this, we replace BN with Online Normalization (ON) [20]. Like BN, ON normalizes the features along the spatial dimension. However, instead of operating on each batch independently, it computes a frame-wise running average. Thus, the output depends on all previous inputs, making backpropagation non-trivial. Online normalization handles this by applying some compensating transformations in the backward pass, introducing two new hyper-parameters $\alpha_{\text{fwd}}$ and $\alpha_{\text{bwd}}$. According to Chiley et al. [20] online normalization achieves approximately the same accuracy as BN for ResNet50 on the ImageNet 2012 classification task.

## IV. Tool Flow

Our tool flow, shown in Figure 2, consists of four steps: *Plan*, *Implement*, *Build*, and *Run*. As a whole, it takes a CNN model, cluster description, and training dataset and generates trained model parameters.

To demonstrate our tool flow, we will illustrate the individual stages using a minimal demo network. The CNN model is defined in Python as shown in Listing 1.

```
def residual_block(output_channels, stride, source):
    x = conv(output_channels, 3, stride, source)
    x = relu(bias(x))
    x = bias(conv(output_channels, 3, 1, x))

    y = conv(output_channels, 1, stride, source)
    y = add(x, y)
    y = relu(y)
    return y

# input name, [vertical, horizontal, channels]
x = add_input("input", [32, 32, 3])
# output_channels, kernel_size, stride, input
x = conv(16, 5, 2, x)
x = relu(bias(x))

x = residual_block(32, 2, x)
x = residual_block(64, 2, x)
x = conv(128, 1, 1, x)

x = global_avg_pool(x)
x = dense(10, x)

ref = add_input("label_one_hot", [10])
x = soft_cross_loss(x, ref)
add_output("output", x)
```

Listing 1. Example CNN model description.

*Plan*

The *Plan* step ingests Python files describing the FPGA cluster as well as the CNN model and places each layer in the model on one of the FPGAs in the cluster. Using the model from Listing 1 and a cluster of four FPGAs, the performed mapping is illustrated in Figure 3.

We want to find mappings of $L$ layers to $N$ FPGAs, represented as $d_{l,n} \in \{0,1\}$ with $0 \leq l < L$ and $0 \leq n < N$, where $d_{l,n} = 1$ means layer $l$ is placed on FPGA $n$. Each layer must be placed exactly once: $\sum_{n=0}^{N-1} d_{l,n} = 1$ for all layers $l$. These mappings need to respect the available resources $D_n^{\text{dev}}, M_n^{\text{dev}}, B_n^{\text{dev}}, C_n^{\text{dev}} \in \mathbb{R}$, refering to DSP cores, on-chip RAM bits, DRAM bandwidth and network bandwidth for FPGA $n$, respectively. For each layer type, we manually define estimates of the implementation's resource consumption: $D_l^{\text{layer}}(t), M_l^{\text{layer}}(t), B_l^{\text{layer}}(t) \in \mathbb{R}$ as functions of the throughput $t \in \mathbb{R}$. The required bandwidth $C_{l,k}^{\text{layer}}(t)$ between layers $l$ and $k$ can be calculated as $tF_{l,k}$, where $F$ is the sum over all connections between $l$ and $k$ of the connection's feature map size.

Given these definitions, our mapping should maximize the throughput $t$, while satisfying all resource constraints:

Maximize $t$ subject to

$$\sum_{l=0}^{L-1} d_{l,n} D_l^{\text{layer}}(t) \leq D_n^{\text{dev}} \qquad \forall n < N,$$

$$\sum_{l=0}^{L-1} d_{l,n} M_l^{\text{layer}}(t) \leq M_n^{\text{dev}} \qquad \forall n < N,$$

$$\sum_{l=0}^{L-1} d_{l,n} B_l^{\text{layer}}(t) \leq B_n^{\text{dev}} \qquad \forall n < N,$$

$$\sum_{l=0}^{L-1} \sum_{k=0}^{L-1} d_{l,n} \left(1 - d_{k,n}\right) C_{l,k}^{\text{layer}}(t) \leq C_n^{\text{dev}} \qquad \forall n < N,$$

$$\sum_{n=0}^{N-1} d_{l,n} = 1 \qquad \forall l < L$$

Note that the resource estimates can depend on the throughput in complicated ways and are generally not differentiable, ruling out any gradient based optimization approaches. We approach this problem by doing a binary search for the highest implementable target throughput. For each given throughput we check the satisfiability of the above constraint using the CP-SAT solver from the Google OR-Tools library.

Our actual implementation adds three additional constraints:

- For reasons explained in the next subsection, the number of network connections for each FPGA is limited to a

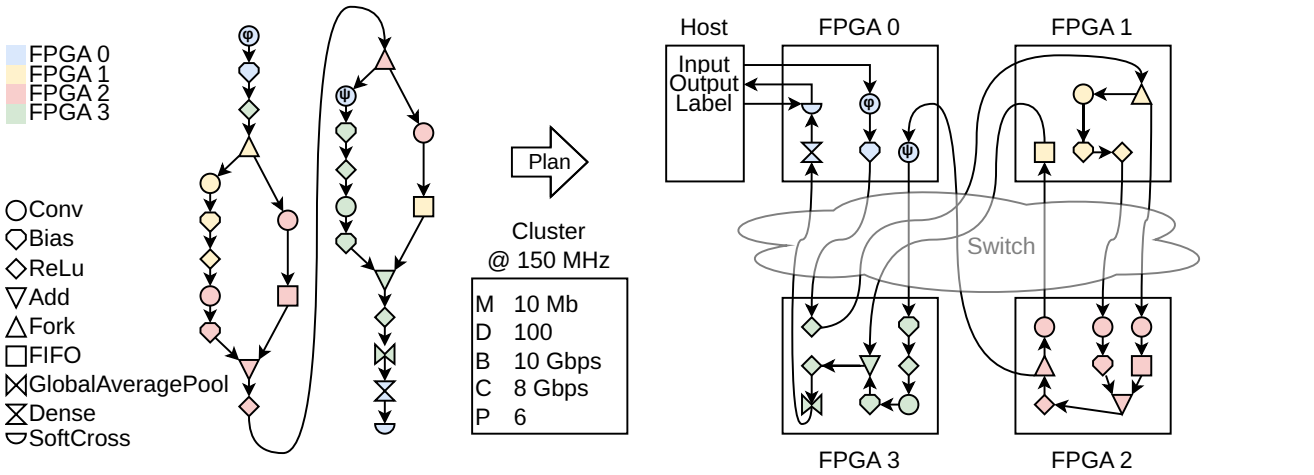


Fig. 3. Mapping the CNN model described in Listing 1 onto an FPGA cluster. FPGAs are connected via Ethernet. Additionally, FPGAs containing input or output layers are connected to the controlling host via PCIe. The convolution layers mapped onto FPGA 0 are named $\phi$ and $\psi$.

TABLE I
CONVOLUTION LAYER SPECIFICATIONS

| Specification | Parameter | Conv($\phi$) | Conv($\psi$) |
|---|---|---|---|
| Output channels | $p^{oc}$ | 16 | 64 |
| Kernel shape | $(p^{kv}, p^{kh})$ | (5, 5) | (3, 3) |
| Padding | | (2, 2, 2, 2) | (0, 0, 1, 1) |
| Stride | $(p^{sv}, p^{sh})$ | (2, 2) | (2, 2) |
| Parallel input channel | $p^{pic}$ | 1 | 2 |
| Parallel output channel | $p^{poc}$ | 4 | 2 |
| Parallel horizontal | $p^{ph}$ | 2 | 2 |
| Parallel I/O | $(p^{i}, p^{o})$ | (1, 1) | (1, 1) |
| Cache size [frames] | | 10 | 10 |
| Input Shape | $(p^{iv}, p^{ih}, p^{ic})$ | (32, 32, 3) | (8, 8, 32) |
| Induced delay [frames] | | 0.23 | 0.91 |
| Target throughput [fps] | | 3906.25 | 3906.25 |
| Actual throughput [fps] | | 3906.25 | 4069.01 |
| Output shape | $(p^{ov}, p^{oh}, p^{oc})$ | (16, 16, 16) | (4, 4, 64) |

TABLE II
UNIFIED LAYER INTERFACE

| Description | Direction | Type | # |
|---|---|---|---|
| Axi4 | master | Axi4 | $\leq 1$ |
| Features | input | Stream(Vector(Float32)) | $I$ |
| Features | output | Stream(Vector(Float32)) | $J$ |
| Feature gradients | input | Stream(Vector(Float32)) | $J$ |
| Feature gradients | output | Stream(Vector(Float32)) | $I$ |
| Parameter | input | Stream(Float32) | 1 |
| Parameter | output | Stream(Float32) | 1 |
| Parameter read request | input | Boolean | 1 |
| Parameter update request | input | Boolean | 1 |
| Parameter update done | output | Boolean | 1 |
| Learning rate | input | Float32 | 1 |
| Momentum | input | Float32 | 1 |
| Frame done | output | Boolean | 1 |

maximum of $P \in \mathbb{N}^+$.

- Each network hop introduces some amount of latency, which can lead to stalls when two branches that have taken different paths across the cluster join. In order to avoid having to calculate and compensate for these delays, joining branches are required to have taken an equal number of network hops from the input.
- Input- and output layers are forced onto FPGA 0. We transfer inputs and outputs via PCIe and in our cluster, only one FPGA is directly connected to the host.

For each FPGA, the *Plan* step emits a json file describing the layers to be implemented and their internal connections. For the convolution layer, the required specifications are listed in Table I. An additional json file contains information about the connections between the FPGAs.

*Implement*

NADA is built on top of an accelerator framework [21] previously developed in our group. It provides access to DRAM, a UDP stack, and a Command Control Interface (CCI) which can be accessed via either PCIe or over the network. We use SpinalHDL to implement a socket for the accelerator framework, which executes the part of the CNN mapped to the specific device.

Each layer is implemented in a separate component which provides a unified interface, as shown in Table II. It contains an optional AXI master interface for DRAM access. A set of control registers connected to the CCI interface is used to check the status of the accelerator, to set learning rate and momentum, to trigger parameter updates and to write and read back parameters. Features and feature gradients are transferred in a pipelined manner using stream interfaces with valid / ready handshake. Layers can have multiple in- and outputs. E.g. the add layer adds features from $I = 2$ inputs and generates $J = 1$ output. To meet the desired throughput for both the input and output of each layer, parallel transfer of $p^{i}$ input features and $p^{o}$ output features can be performed. During the backward pass, when feature gradients stream in the opposite direction, parallel transfer of $p^{o}$ input feature gradients and $p^{i}$ output feature gradients is carried out.

Layers on different FPGAs are connected using the UDP stack provided by the accelerator framework. UDP provides neither flow control nor error handling. To work around this, priority-based ethernet flow control [22] is used, assigning each connection of an FPGA a dedicated priority. Ethernet supports a maximum of eight different priorities, thus limiting us to eight connections.

We detect packet loss by adding a counter to the beginning of each packet. The receiving end checks that these packet numbers always increment by one. In our current setup with only a single switch, we have not yet observed any packet loss. Because of this, errors are only detected but not handled in hardware. Instead, the software controlling the accelerator recovers from a previous state if necessary.

The SpinalHDL implementation emits Verilog code, which

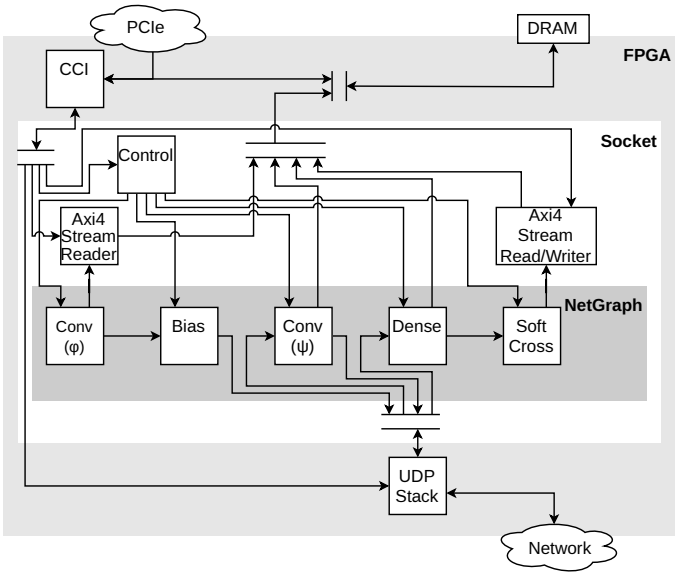


Fig. 4. Architecture implemented on FPGA 0 in the cluster.

TABLE III
ESTIMATED AND ACTUAL RESOURCE CONSUMPTION OF CONVOLUTION LAYERS.

| Parameter | Conv($\phi$) | | Conv($\psi$) | |
|---|---|---|---|---|
| | Estimated | Actual | Estimated | Actual |
| on-chip RAM [Kb] | 272.9 | 518.4 | 1446.6 | 1659.5 |
| DSPs | 32 | 32 | 38 | 38 |

TABLE IV
RESOURCE CONSTRAINTS PER FPGA

| Symbol | Resource | Available Total | for Planning |
|---|---|---|---|
| $M$ | on-chip RAM | 55 Mb | 35 Mb |
| $D$ | DSPs | 1518 | 1500 |
| $B$ | DRAM Bandwidth† | 50 Gbps | 50 Gbps |
| $C$ | Ethernet Bandwidth† | 40 Gbps | 40 Gbps |
| $P$ | PFC connections | 8 | 6* |
| | Clock Rate | 100 MHz | 100 MHz |

†Full duplex.
*Maximum number supported by the switch.

TABLE V
EXAMPLE CNN

| Input | Operator | t | c | n | s |
|---|---|---|---|---|---|
| $224^2 \times 3$ | im2col $3 \times 3$ | | 27 | 1 | |
| $112^2 \times 27$ | bottleneck | $\frac{32}{27}$ | 16 | 1 | 1 |
| $112^2 \times 16$ | bottleneck | 4 | 16 | 1 | 2 |
| $56^2 \times 16$ | bottleneck | 6 | 24 | 2 | 2 |
| $28^2 \times 24$ | bottleneck | 6 | 48 | 2 | 2 |
| $14^2 \times 48$ | bottleneck | 6 | 64 | 2 | 1 |
| $14^2 \times 64$ | bottleneck | 6 | 128 | 1 | 2 |
| $7^2 \times 128$ | bottleneck | 6 | 256 | 1 | 1 |
| $7^2 \times 256$ | conv2d $1 \times 1$ | | 1024 | 1 | 1 |
| $7^2 \times 1024$ | avgpool $7 \times 7$ | | 1024 | 1 | |
| $1^2 \times 1024$ | conv2d $1 \times 1$ | | 1000 | 1 | |

TABLE VI
BOTTLENECK RESIDUAL BLOCK

| Input | Operator | Output |
|---|---|---|
| $h \times w \times k$ | $1 \times 1$ conv2d, ReLU6 | $h \times w \times (tk)$ |
| $h \times w \times tk$ | $3 \times 3$ dwise s=s, ReLU6 | $\frac{h}{s} \times \frac{w}{s} \times (tk)$ |
| $\frac{h}{s} \times \frac{w}{s} \times tk$ | $1 \times 1$ conv2d | $\frac{h}{s} \times \frac{w}{s} \times k'$ |

is then synthesized for each FPGA.

Figure 4 shows a block diagram of the hardware architecture implemented in FPGA0. Due to the third additional constraint, the input- and output layers (Convolution layer $\phi$, the softmax- and cross entropy loss function) are placed onto this device. Additionally, the bias layer following the first convolution and the convolution layer $\psi$ are implemented on FPGA0.

The layer specifications for both convolutions on FPGA0, emitted by the *Plan* stage, are given in Table I. Each layer must at least support the overall target throughput.

For the given model and cluster, placement was successful at 3906 fps. The estimated and required resource consumption of the convolution layers $\phi$ and $\psi$ are given in Table III.

*Run*

To run the actual training, the bitstream files generated in the previous step need to first be programmed to the FPGAs in the cluster. The training script then uses the information passed on from then *Plan* and *Implement* steps to discover the FPGAs on the network by a unique ID assigned to their socket and to set up the network connections between them.

Initial parameters are written to each layer via CCI. Then, for each batch, inputs are written to the DRAM of FPGA 0, and all FPGAs in the cluster are signaled to start processing. Once all layer implementations have finished processing the batch, parameters are updated with the accumulated gradients and the next batch is started. At any point, the current parameters can be read back via CCI. Currently, this is done once at the end of each epoch. But should the network become less reliable, more frequent snapshots are easily implemented.

## V. EVALUATION

The evaluation of NADA is split into two parts: **1.** Evaluating the scalability and flexibility by planning an example CNN on a cluster of three up to nine FPGAs. **2.** Building and training MobileNetV2 [23] utilizing real hardware. We have access to a cluster of six Arria 10 GX1150 FPGAs connected via 40 Gbps Ethernet using a Mellanox SN2100 switch. Resource constraints per device are given in Table IV.

*Scalability*

The example CNN is defined in Table V. Each line describes a sequence of one or more layers repeated $n$ times, where the first has stride $s$, while all others have stride 1. All layers in the sequence have $c$ output channels. The bottleneck residual blocks, described in Table VI, have $k$ input- and $k'$ output channels with stride $s$ and expansion factor $t$. All convolutions are followed by normalization and bias.

Training of this CNN on the ImageNet dataset requires $\sim$700 MOPS / frame. It has $\sim$1.8 M parameters in total of which $\sim$1.0 M belong to the Dense layer whose parameters are stored in DRAM. As we train in single precision floating point, a minimum of 50 Mb of on-chip RAM is required just for the parameters and corresponding gradients. Considering overheads (e.g. UDP and PCIe), successful placement requires at least three FPGAs. Figure 5 shows the throughput for the example CNN on a cluster of up to nine nodes. Without MP planning, it shows linearity up to six devices. Utilizing straight forward MP for the first two bottleneck groups results in a continued linear rise. In each bottleneck group, the output of the first convolution layer is sliced and then processed in separate layers. Designs *MP2* and *MP4* were sliced two and four times, respectively. From seven devices onwards, planning *MP2* results in a better throughput and only for nine devices is *MP4* beneficial.

Figure 6b shows the resource utilization planned for *MP4*. Clearly, it is DRAM bandwidth bound with all devices utilizing well over 90% and an average of 97%. Figure 6c

shows the device utilization for the example CNN without MP. Single layer implementations become so resource intensive that is impossible to have a high resource utilization across all devices, resulting in a lower throughput.

As shown in Figure 6a the most computationally complex layer type is the convolution, but a non-negligible share of the resources is spent on the normalization. Especially the DRAM bandwidth required for the backward pass should be easily alleviated in the future by computing the output of the forward pass again instead of buffering it in DRAM.

As shown in Table IV, we allow the planning step to utilize only 35 Mb of the 55 Mb of on-chip RAM available on each FPGA. This is to compensate for overheads not captured in our resource estimates. On one hand, the estimates neglect some components of our hardware architecture, like the memory interconnect and the UDP/IP stack. More significantly though, on-chip RAM bits are simply a poor proxy metric for the actual constrained resource, the Block RAM (BRAM) block. On the Arria 10 FPGA, each BRAM block has a fixed size of 20 kb and a minimum depth of 512 words. If the memories in the design do not partition nicely into these blocks, large overheads can quickly accumulate.

To illustrate this, we take a look at two devices from the *no MP* design on the nine FPGA cluster. According to the resource estimates, device 2 has the lowest on-chip RAM utilization, at just $\sim$0.16 Mb, while device 1 has the highest at $\sim$26 Mb. The synthesis for both designs reports an actual utilization of $\sim$4.8 Mb for device 2 and $\sim$31 Mb for device 1, consistent with a small static overhead of $\sim$5 Mb. Even the smaller design however uses an astonishing 68% (1858 blocks) of the available 2713 BRAM blocks. The larger design still gets built successfully, but utilizes every single BRAM block available. This is possible because once all BRAM blocks are exhausted, the place-and-route tool (Quartus) will start implementing smaller on-chip RAMs as distributed RAM or even registers instead. This behavior makes it even harder to select a proper threshold for the planning step. We arrived at the current value of 35 Mb by trial and error, reducing the target until builds started to consistently succeed.

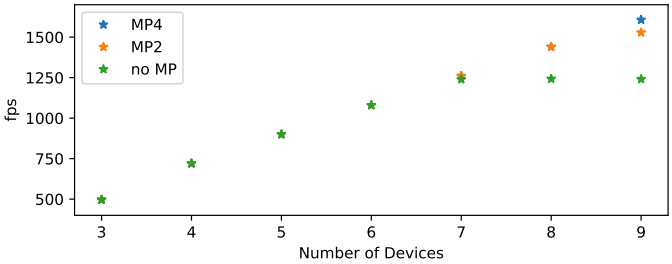


Fig. 5. Achieved throughput for training the example CNN on a cluster of Arria 10 GX1150 FPGAs. Planned without Model Parallelism (*no MP*) or with a factor of two (*MP2*) and four (*MP4*), respectively. For $N \leq 6$, all three perform the same. For $N \leq 8$, *MP4* performs the same as *MP2*.

ALMs and registers were not considered for the *Plan* step because their utilization is negligible. The *no MP* design on

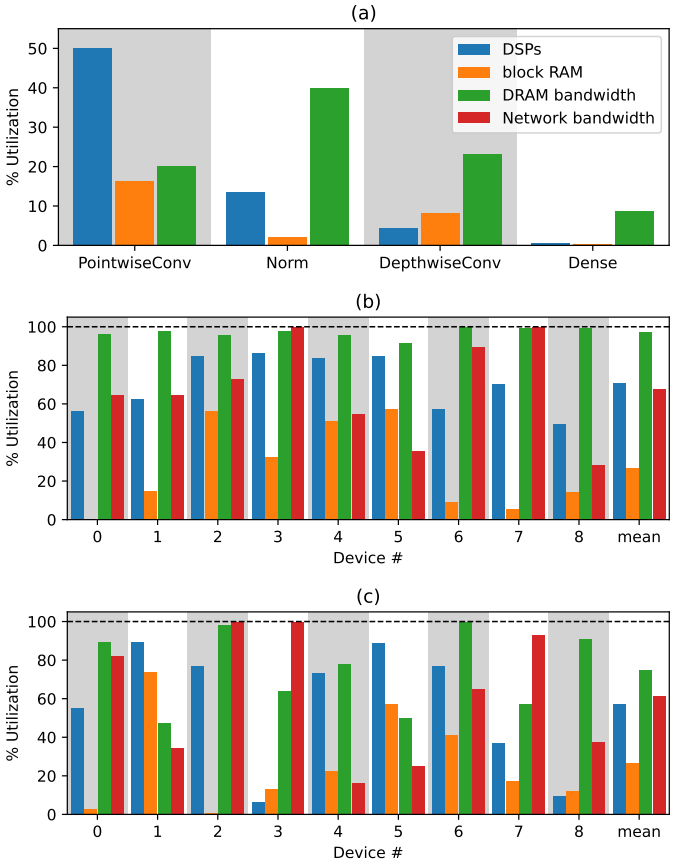


Fig. 6. Planned resource utilization for training the example CNN with *MP4* per layer (a) and per device (b) and also per device with *no MP* (c).

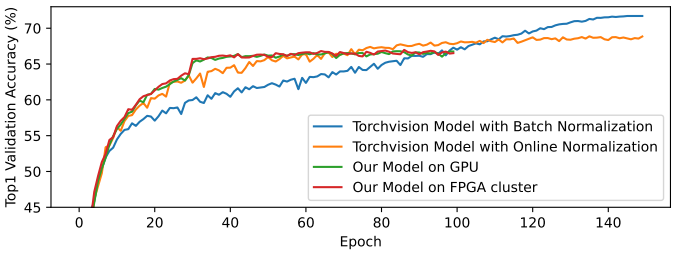


Fig. 7. Training performance of the various model variants. The torchvision models use a learning rate of 0.1 with cosine annealing and a momentum of 0.95. For our implementation, the learning rate is 0.04 with a step decay after 30 epochs and a momentum of 0.91. Training was stopped after 100 epochs as no more accuracy improvement was observed.

the nine FPGA cluster required a maximum of 36% ALMs and 18% registers with an average of 25% and 13%, respectively.

### *MobileNetV2*

We implement MobileNetV2 for our framework, replacing BN with ON. We apply the NADA tool flow for our cluster of six nodes and run training on the ImageNet dataset for 100 epochs. The planning and synthesis on an AMD-EPYC-7443 took 24 minutes and 3 hours, respectively.

Our design is not yet optimized for timing, running at 100 MHz, which results in 4 days to train 100 epochs. At this clock rate, our six-node cluster (20 nm technology)

ideally provides 1.8 TFLOPS. At 1.8 GFLOP per frame, this leads to a maximum theoretical throughput of 1000 fps. We actually achieve a throughput of 395 fps, or 359 fps including parameter update, equivalent to 35.9 % of the maxmimum. As a point of reference, we train the same model using PyTorch on one Tesla V100 GPU (12 nm). It provides 14 TFLOPS of theoretical performance and reaches a throughput of 831 fps, i.e. 10.7 % of the maximum. Regarding DRAM bandwidth utilization, the Tesla V100 provides 7.2 Tbps. Our cluster provides a total of 600 Gbps, just 8.3 % of the GPU's bandwidth. While NADA's absolute performance still lacks behind the GPU for this model and cluster, it makes more efficient use of the devices: By more than a factor of three with respect to compute performance and a factor of five in DRAM bandwidth.

Figure 7 shows the validation accuracy achieved after each epoch of training. As a baseline, it contains the MobileNetV2 model from the torchvision library, trained using cosine annealing. Replacing BN with ON in this model leads to a drop in accuracy of $\sim 3\%$. To avoid randomness when comparing to the reference implementation, our model does not contain the Dropout layer used by the torchvision variant of MobileNetV2 before the final Dense layer. This is likely the reason our model as implemented for NADA performs slightly worse still. To ensure the correctness of our hardware implementation, we export the model from NADA back to PyTorch. As expected, this variant run on the GPU performs essentially the same as on the FPGA cluster.

## VI. Conclusion

NADA is an easy-to-use framework for flexible and scalable DNN training on FPGA clusters. It operates in a tightly pipelined layer-parallel manner, circumnavigating the inherent incompatibility of this approach with batch normalization by using online normalization instead. We showcased the scalability of NADA using an example CNN, and we demonstrated its viability for real world applications by training MobileNetV2.

### *Future Work*

Our focus at the moment is on supporting larger models and achieving more competitive throughput. The main obstacle in supporting larger models is the limited on-chip RAM. In order to make full use of the already available resources, we are working on a more accurate memory estimation operating with BRAM blocks rather than on-chip RAM bits. Much larger gains can be made by integrating more modern FPGAs into our setup. Heterogenous clusters will also serve as a key enabler towards achieving more competitive throughput. FPGAs supporting HBM can help utilize all the available compute power, even for layers with a low ratio of compute complexity to feature map size.

In general, we plan to use NADA as a testbed for further exploration of parallelization and optimization strategies for distributed DNN training.

## References


[1] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–43, 2019.

[2] D. Narayanan *et al.*, "Pipedream: Generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 1–15.

[3] N. K. Unnikrishnan and K. K. Parhi, "LayerPipe: Accelerating deep neural network training by intra-layer and inter-layer gradient pipelining and multiprocessor scheduling," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, Nov. 2021.

[4] Y. Huang *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," in *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, 2019.

[5] T. Wang *et al.*, "Fpdeep: Scalable acceleration of cnn training on deeply-pipelined fpga clusters," *IEEE Transactions on Computers*, vol. 69, pp. 1143–1158, 2020.

[6] O. Terzo *et al.*, Eds., *Heterogeneous Computing Architectures*. CRC Press, Sep. 2019.

[7] G. Lacey, G. W. Taylor, and S. Areibi, "Deep learning on fpgas: Past, present, and future," 2016.

[8] C. Zhang *et al.*, "Energy-efficient CNN implementation on a deeply pipelined FPGA cluster," in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*. ACM, Aug. 2016.

[9] A. Shawahna, S. M. Sait, and A. El-Maleh, "FPGA-based accelerators of deep learning networks for learning and classification: A review," *IEEE Access*, vol. 7, pp. 7823–7859, 2019.

[10] S. Nambi *et al.*, "Expan(n)d: Exploring posits for efficient artificial neural network design in fpga-based systems," *IEEE Access*, vol. 9, pp. 103 691–103 708, 2021.

[11] Y. Zhu *et al.*, "Distributed recommendation inference on fpga clusters." Dresden, Germany: IEEE, 2021, pp. 279–285.

[12] Z. Choudhury *et al.*, "An FPGA overlay for CNN inference with fine-grained flexible parallelism," *ACM Transactions on Architecture and Code Optimization*, vol. 19, no. 3, pp. 1–26, May 2022.

[13] W. Zhao *et al.*, "F-cnn: An fpga-based framework for training convolutional neural networks," in *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. London, UK: IEEE, 2016, pp. 107–114.

[14] C. Luo *et al.*, "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)*. San Diego, CA, USA: IEEE, 2019, pp. 45–52.

[15] T. Itsubo *et al.*, "Accelerating deep learning using multiple gpus and fpga-based 10gbe switch," in *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. Västerås, Sweden: IEEE, 2020, pp. 102–109.

[16] J. Lu, J. Lin, and Z. Wang, "A reconfigurable DNN training accelerator on FPGA," in *2020 IEEE Workshop on Signal Processing Systems (SiPS)*. IEEE, Oct. 2020.

[17] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*. Physica-Verlag HD, 2010, pp. 177–186.

[18] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. San Francisco, CA, USA: IEEE, 2014, pp. 10–14.

[19] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on Machine Learning*, vol. 37, Lille, France, 07–09 Jul 2015, pp. 448–456.

[20] V. Chiley *et al.*, "Online normalization for training neural networks," in *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[21] F. Steinert *et al.*, "Hardware and software components towards the integration of network-attached accelerators into data centers." Kranj, Slovenia: IEEE, 2020, pp. 149–153.

[22] *IEEE Standard for Local and metropolitan area networks–Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks–Amendment 17: Priority-based Flow Control*, Std.

[23] M. Sandler *et al.*, "MobileNetV2: Inverted residuals and linear bottlenecks," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. IEEE, Jun. 2018.