RFNoC Neural Network Library using Vivado HLS

Edward J. Kreinar EJ@HE360.COM
Hawkeye 360, 196 Van Buren Street, Suite 450, Herndon, VA 20170

Abstract

The FPGA-based neural network library presented here provides an RF-Network on Chip (RFNoC) out-of-tree (OOT) module for efficiently deploying a trained neural network to an FPGA. The neural network module, rfnoc-hls-neuralnet (Kreinar, 2017), exposes a library of pre-optimized C++ neural network building blocks designed for the Vivado HLS tool. RFNoC provides a convenient input/output interface between hardware and software that is compatible with gnuradio. Ideally, the neural network designer will be able to deploy neural networks and evaluate resource vs. throughput tradeoffs without needing to develop and maintain repetitive glue code in FPGA and software. Presented examples demonstrate various use-cases in a simulation environment and on the E310, including image classification and modulation recognition, using both fully-connected and convolutional layers.

1. Introduction

Neural networks are rapidly surpassing decades of expert human experience across a wide variety of fields. In RF communications, however, neural networks have historically been used primarily for classification tasks such as modulation recognition (Yang et al., 2014) (Kawamoto & McGwier, 2016), while more difficult tasks such as demodulation, error correction, channel coding, etc, have remained firmly in the realm of "expert systems." Within the past year, researchers have begun to apply deep neural networks to RF communications in an attempt to replace the expert system with a machine-learned system that can perform modulation and demodulation while achieving the theoretical Shannon limit across a wide variety of operating environments (O'Shea & Hoydis, 2017).

It is expected that the trend of neural network algorithms will continue to grow in the field of RF communications (e.g., the recent DARPA broad-agency-announcement attempting to solve advanced RF challenges through machine learning (DARPA, 2017)), creating a natural requirement for hardware and software that can run neural network algorithms on RF data with low size, weight, power, and high processing throughput. Such requirements provoke a tempting opportunity for FPGA acceleration.

Unfortunately, an FPGA neural network implementation introduces several non-trivial challenges. In particular, a neural network cannot be provided as a one-size-fits-all solution; the neural network architecture (size, type, and number of layers) is a major driver of performance, and the variety of possible architectures is too large to create a generic FPGA solution. Such a solution would either consume too many FPGA resources or not achieve the desired throughput. Using a software-only approach, changing neural net architecture is trivial; however, in hardware, FPGA fabric cannot be arbitrarily reconfigured on the fly. Therefore, the true strength of an FPGA-based neural network is the ability for the designer to regenerate a resource-efficient FPGA implementation in a short of amount of time without needing to reinvent the wheel.

Based on these assertions, the goals of the RFNoC Neural Network Library are as follows:

1. Provide an HLS library of common neural network functions
2. Provide the FPGA architecture to wrap generated HDL code into an RFNoC compute engine (CE)
3. Use RFNoC to expose a software interface for the CE

The remainder of this section will focus on the background of Vivado HLS and RFNoC. The rfnoc-hls-neuralnet features, examples, and results are discussed in Section 2. An anecdotal workflow is presented in Section 3 in an effort to highlight the expected use-case of rfnoc-hls-neuralnet. Finally, follow-up actions are discussed in Section 4.

1.1. Vivado HLS

Vivado HLS is a Xilinx tool used to synthesize C, C++, or SystemC code into verilog or VHDL code. The HDL is produced according to "pragma" directives inserted into

Proceedings of the 7th GNU Radio Conference, Copyright 2017 by the author(s).

the C code (or into a separate directive file) that instruct the HLS compiler exactly *how* to synthesize the algorithm. Typical directives include actions such as how to unroll for-loops, how to partition arrays, and how to pipeline various segments of the source code.

In addition, Vivado HLS provides a fixed-point datatype class that is interchangeable with floating-point datatypes. Algorithms may be developed using floating-point, then easily switched to fixed-point mode for synthesis; Vivado HLS maintains all required fixed-point conversions during synthesis. Such prototyping is a powerful workflow for digital signal processing tasks where HDL algorithms are often required to be equivalent to the floating-point reference algorithm and bit-accurate between simulation and synthesis.

Vivado HLS also includes a specialized GUI that allows users to evaluate resource usage and algorithm throughput by comparing synthesized results between multiple sets of directives. In this way, a primary strength of Vivado HLS lies in the rapid assessment of resources, throughput, and performance tradeoffs.

1.2. RFNoC

The open source RFNoC FPGA architecture (Braun et al., 2016) provides many benefits to an RF system designer. First, RFNoC exposes a process to develop and integrate FPGA “compute engines” with a standardized software interface that allows input and output data to be routed anywhere in the system— data may be routed to or from the processor, or to or from a different FPGA compute engine, and the data path is reconfigurable at run time.

Second, RFNoC also provides a reliable and modular transport layer between the FPGA and the processor. Two very different hardware platforms, the Ettus E300 and the Ettus X300 series radios, may use the exact same RFNoC FPGA and software code, while RFNoC handles FPGA to the processor interface. Furthermore, any aspect of the software interface that is specific to a particular RFNoC compute engine is dynamically attached to the correct FPGA component, and is also reusable between hardware platforms.

The RFNoC architecture represents a significant amount of “nuts and bolts” effort that is available to developers and users of compatible hardware platforms.

2. The rfnoc-hls-neuralnet Module

The features of the rfnoc-hls-neuralnet module can be roughly grouped into three areas aligned with the project goals:

1. C++ HLS software: Implements the neural network. Synthesized into FPGA code using Vivado HLS
2. Verilog infrastructure: Wraps the generated HLS outputs and interacts with the RFNoC architecture (AXI-Stream interface, correct framing, etc)
3. Gnuradio-companion interface: Python/C++ software that wraps the FPGA block with a software wrapper compatible with UHD and gnuradio

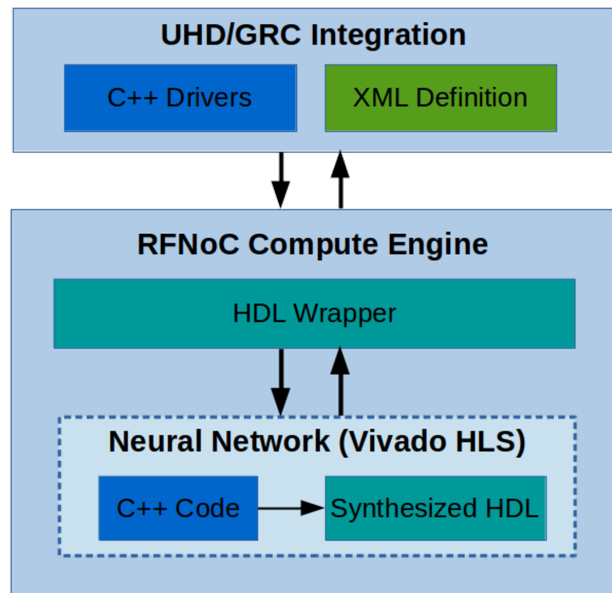

Figure 1. rfnoc-hls-neuralnet architecture

The interaction between the provided features is illustrated in Figure 1.

2.1. Provided Features

2.1.1. C++ HLS Software

The rfnoc-hls-neuralnet module exposes a limited set of HLS-optimized neural network building bocks. These software components, when applicable, are roughly modelled on the user interface of TensorFlow’s neural network module. The HLS library of rfnoc-hls-neuralnet currently supports:

1. Fully connected layer: Performs different HLS optimization options based on the size of the layer
2. IQ convolution layer. Multiple output channels. Architecture optimized for streaming IQ data.
3. One-Dimensional convolutional layer. Multiple input channels

Table 1. Synthesized HLS resource usage of neural network library components in rfnoc-hls-neuralnet

Component Name	Component Size	Sample Interval	BRAM18	DSP48	FF	LUT
Fully Connected Layer (Size In × Size Out)	10×40	9	0	8	658	1183
	40×40	9	8	8	592	1128
	784×10	1	8	10	510	843
	784×256	36	193	8	647	1222
IQ Convolution (IQ × Filter Size × Chan Out)	2×4×1	10	0	2	366	148
	2×4×3	15	0	6	575	333
	2×8×30	50	0	60	3374	2264
	2×8×128	166	1	64	13440	37542
1-D Convolution (1 × Filter Size × Channels)	1×4×2	4	0	4	333	70
	1×8×8	10	0	4	874	304
	1×16×64	16	16	4	1024	850
tanh	100	1	1	0	51	206
sigmoid	100	1	1	0	56	182
relu	100	1	0	0	12	45
relu6	100	1	0	0	15	50
maxpool (size 2, stride 2)	128×64	2	2	0	77	252

4. Various activation functions:
   - relu
   - relu6
   - sigmoid
   - tanh
5. Maxpool operation: size 2, stride 2

All neural network weights are currently required to be hardcoded into an HLS header file, which is HLS's preferred method for initializing and importing memory from file. Programmable weights are discussion in Section 4 (Follow-Up Actions) as a potential future modification. As-is, hardcoded synthesized weights implies that only pre-trained networks may be deployed to FPGA using rfnoc-hls-neuralnet.

Synthesized resource usage estimates are shown in Table 1, indicating nominal FPGA results at a few relevant component sizes. While the BRAM usage runs high in the fully-connected layer, the DSP48s blocks, Flip Flops (FF), and Lookup Tables (LUT) achieve fairly low device utilization overall. The sample interval represents how often the HLS block can accept a new input sample (i.e., algorithm throughput). The HLS neural network library uses size-specific compiler directives that change the physical hardware implementation for larger or smaller networks as appropriate to achieve an acceptable tradeoff of throughput vs resources. For specialized applications, HLS directives may be edited to specifically influence the synthesized HDL architecture.

It is worth noting that these software components represent just a small subsection of the possible operations available to neural network designers using TensorFlow or other state-of-the-art software toolkits. For the first pass through HLS implementations, the rfnoc-hls-neuralnet library focuses on several of the most common neural network components for FPGA synthesis. The fully-connected layers and 1D convolutions are particularly useful for RF processing and are sufficient to assemble a variety of networks demonstrating proof-of-concept functionality.

2.1.2. VERILOG INFRASTRUCTURE

The verilog infrastructure wraps around the HLS-generated output modules to interact safely with the RFNoC CE. In most cases, the RFNoC component will not be considered to be in "Simple Mode." Simple Mode in RFNoC is an easy-to-use paradigm where the number of inputs to the RFNoC CE is equal to the number of outputs and the FPGA packet size is constant.

Therefore, the rfnoc-hls-neuralnet library provides the *nnet_vector_wrapper.v* to help convert input and output vector sizes to the desired length based on the size of the implemented network. An RFNoC testbench can be generated using the same stimulus used in the HLS C++ testbench to verify the full rfnoc FPGA architecture is synthesized and runs successfully.

2.1.3. GNURADIO-COMPANION INTERFACE

For most blocks, no specialized C++ drivers are required because the interace between FPGA and processor uses the

Table 2. Synthesized HLS resource usage of rfnoc-hls-neuralnet examples

Example Name	Sample Interval	BRAM18	DSP48	FF	LUT
ex_1layer	1	8	10	516	849
ex_modrec	9	29	42	2441	4885
ex_2layer	36	214	18	1226	2239
ex_iqconv	33	106	102	14707	38470

default streaming interface. So, the RFNoC software interface is implemented as a gnuradio companion (GRC) xml file which declares an *ettus.rfnoc_generic* object in the python script. RFNoC handles the interface to the FPGA and can be accessed using a GRC flowgraph or a dedicated python application.

2.2. Examples and Results

There are four total examples in the rfnoc-hls-neuralnet module designed to highlight several aspects of the library's features. Two examples fully exercise the HLS, RFNoC, and GRC implementatons. These two examples have been tested successfully on the E310, running file input through the RFNoC CE. The examples are: 1) A 1-layer image classification network based on the Udacity "maching learning" online course (ex_1layer), and 2) An RF modulation recognition network that uses a set of expert features to identify modulation type (ex_modrec), discussed in more detail in Section 3.

Two additional examples are provided as "HLS-only" applications which demonstrate use of the neural network HLS library, but have not been implemented on hardware. They are: 1) a more advanced two-layer image classification network (ex_2layer), and 2) a convolutional network optimized for modulation recognization of streaming IQ RF data rather than manually-extracted features (ex_iqconv). The ex_iqconv example architecture consists of two convolutional layers separated by maxpool operations and 4 fully-connected layers (O'Shea & Hoydis, 2017). The first convolutional layer consumes raw RF data. The HLS testbench has been developed against fake weight parameters; there does not currently exist a trained set of network parameters available to use in the ex_iqconv example.

Synthesized resource estimates for the HLS code for all examples are shown in Table 2. The GRC flowgraph for example 2 is shown in Figure 2.

3. Neural Network Design Workflow

The design workflow using the rfnoc-hls-neuralnet repo largely follows the outline of the features illustrated in Figure 1. This walkthrough specifically focuses on the modulation recognition (modrec) example based on the work of (Kawamoto & McGwier, 2016).

1. *HLS Floating-point Simulation*: In the initial stage, the developer first implements and simulates the desired neural network using floating point datatypes in C++ using the provided code from rfnoc-hls-neuralnet. Review the readme in the rfnoc-hls-neuralnet/rfnoc/hls folder to discuss how to create HLS code in the RFNoC framework. The general goal in this step is to identically replicate the performance of the selected neural network algorithm in the HLS framework.

For the modrec example, a snippet of neural network code is shown in Table 2.2, which is pulled directly from *ex_modrec.cpp*. In order to provide test input to the C++ block, a testbench calls the top-level neural network function with a set of data (ideally a dataset that has been verified with a known result). The testbench compares the calculated output to the known result to confirm successful operation.

Note that while the neural network code of Table 2.2 is relatively straightforward, there two major differences between the HLS-optimized code and a typical C or C++ implementation. First, the input/output interface to each function uses Vivado's *hls_stream* object, which provides a FIFO buffer that helps developers program C++ code that is naturally conducive to FPGA synthesis. Second, the neural network function library extensively uses template programming. When working with Vivado HLS, template programming allows the designer to specify a variety of useful constants to the HLS compiler, including both datatypes and network size. All of the layer sizes and weight/bias datatypes are specified using predefined typedefs in the example header.

2. *HLS Synthesis*: Next, the floating point data types of Step 1 are then converted into fixed point data types. Vivado provides a C++ class (*ap_fixed*) that simplifies the use of fixed point numbers. The "real world" value of the fixed point number may be used interchangeably with floating point; but the rounding, conversion, overflow, and all arithmetic/multiplication operations are handled by the HLS compiler. The designer will switch data types to fixed point numbers, compare results to simulation, and then adjust the fixed point data type format as required to meet the

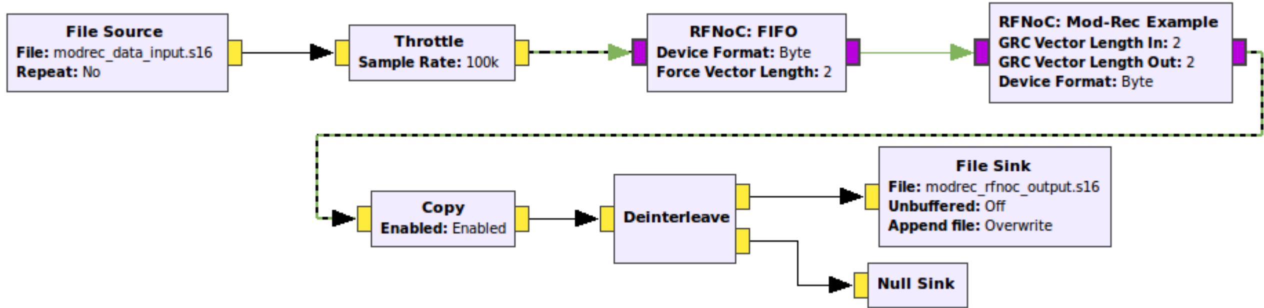

Figure 2. GRC flowgraph with file-based test stimulus for “ex_modrec” example

Table 3. C++ snippet of the 5-layer modulation recognition network *ex_modrec*

```
// LAYER 1
hls::stream<layer1_t> logits1, hidden1;
nnet::compute_layer<layer0_t, layer1_t, weight_t, bias_t, accum_t, N_LAYER_IN, N_LAYER_1>
                   (data_trunc, logits1, w1, b1);
nnet::relu6<layer1_t, layer1_t, N_LAYER_1>(logits1, hidden1);

// LAYER 2
hls::stream<layer2_t> logits2, hidden2;
nnet::compute_layer<layer1_t, layer2_t, weight_t, bias_t, accum_t, N_LAYER_1, N_LAYER_2>
                   (hidden1, logits2, w2, b2);
nnet::relu6<layer2_t, layer2_t, N_LAYER_2>(logits2, hidden2);

[...]

// LAYER 5
nnet::compute_layer<layer4_t, layer5_t, weight_t, bias_t, accum5_t, N_LAYER_4, N_LAYER_5>
                   (hidden4, res, w5, b5);
```

test vector. The process repeats until the fixed point results agree with the floating point results.

After the neural network is implemented in fixed point, HLS synthesis may run to create HDL code. The HLS directives in the neural network library have been optimized to provide an acceptable tradeoff of resources versus throughput results for a few network sizes– if necessary, edit HLS directives to improve resource usage or algorithm throughput.

3. *RFNoC HDL Compute Engine and Testbench*: After HDL is generated, the resulting module is inserted into an RFNoC CE. The RFNoC CE attaches to the RFNoC crossbar, which provides data routing throughout the rest of the FPGA and to the procesor. For the current example, the CE is *noc_block_modrec.v*. As discussed, the neural network user logic does not typically contain the same number of inputs as outputs; therefore, simple mode of the *axi_wrapper* block must be set to zero, and the user connects the input and output axi-stream buses to the *nnet_vector_wrapper*. The *nnet_vector_wrapper* is a lightweight wrapping block that simply frames the data according to the neural network size from HLS.

Once the RFNoC CE is created, the user testbench can be created to stimulate the CE. In the provided examples, the test stimulus is chosen to be exactly the same as the HLS testbench data; using the same test stimulus provides a comforting knowledge that the synthesized FPGA code is functionally equivalent to the simulated C++ code. In the testbench folder, run the simulation to validate the CE functionality. Refer to the included readme files for simulation instructions. The RFNoC CE is now ready to be inserted into an FPGA image.

4. *Hardware Integration*: Finally, the CE is built into an FPGA image using the typical RFNoC image building workflow. At this point, the FPGA image can be programmed to the Ettus hardware of choice. The newly-created RFNoC block does not require custom C++ drivers, but it does need several xml definitions to be used in GRC. The rfnoc-hls-neuralnet module provides some basic examples of how to interface with the generated block.

Inside rfnoc-hls-neuralnet/grc, the xml file identifies the block interface to GRC; this can be found in *fpgannet_exmodrec.xml*. Once the GRC xml file is ready, the RFNoC block can be inserted into a GRC flowgraph for user input and output, and then run locally or on the E300 series embedded devices.

4. Follow-Up Actions

The rfnoc-hls-neuralnet OOT module remains a work in progress. Several follow-up actions are currently queued for further development. In order of decreasing priority, the suggested improvements are:

1. Additional neural network layer types: While the fully-connected layers and convolutional layers cover many types of neural networks for RF processing, a few more "building blocks" could be helpful as part of the library, specifically 1) 2D convolutional layers, 2) recurrent neural networks, and 3) softmax operation (while usually only needed on the output of a network during training, it's nevertheless a common structure often used in software).

2. Test with live streaming data: The current repo demonstrates example RFNoC neural network blocks on hardware, but does not take the additional step of integrating with live RF data. Live integration at a high data rate would be an excellent demonstration application for the rfnoc-hls-neuralnet library.

3. Programmable weights: Programmable neural network weights were originally on the development roadmap, but were eventually deemed too ambitious for the proof-of-concept architecture presented here. In order to maintain simplicity, weights are required to be fixed in the current implementation; however, programmable weights could become a higher priority improvement depending on demand.

4. Improve weight storage: The BRAM usage of Vivado HLS generated code tends to be higher than expected. This is due to HLS's BRAM packing algorithms and that HLS apparently uses BRAM18s while BRAM36s may also be available. It is likely that most neural networks synthesized for the RFNoC architecture will be memory-limited rather than computationally limited.

5. Provide alternate neural network architectures: Researchers have shown that "binarized" neural networks can be very efficient on FPGAs while maintaining high performance levels (Zhao et al., 2017). A binarized neural network could efficiently store weights and perform multiplications (which evaluate into basic binary operations for one bit values), but this manifests as a slightly different HLS structure than assumed here.

5. Related Work

At a high level view, the last few years have seen a resurgence in the popularity of FPGAs due to their customizability and power savings over GPUs for high-computing tasks. Correspondingly, high level synthesis compilers, which have existed since the 90s (Knapp, 1996), are becoming attractive tools to develop neural networks.

In 2015, a team of researchers published an analysis on the structure of convolutional neural networks (CNNs) and advised on proper HLS synthesis directives for computation and memory optimizations (Zhang et al., 2015). More recently, a group at Cornell implemented and evaluated performance versus power for a so-called "binarized" convolutional neural network (CNN) in an FPGA using the Vivado HLS and SDSoC tools (Zhao et al., 2017). SDSoC provides the FPGA/software interface, while Vivado HLS provides the C++ to HDL synthesis functionality. The architecture shows promising results, though the software is not available as open source.

Other researchers have implemented OpenCL-based solutions that synthesize CNNs to HDL code (Suda et al., 2016), or pure HDL solutions for neural networks. Perhaps the closest comparison to this work is a Master's thesis from 2016 (Gschwend, 2016) that presents ZynqNet, a proof-of-concept project demonstrating FPGA acceleration of CNNs on Zynq hardware using HLS and custom userspace IO for the FPGA/processor interaction, and is available via github. Many of the recent neural network architectures targeting FPGAs perform 2D CNNs in a variety of ways (typically for image processing applications), which the rfnoc-hls-neuralnet library does not currently support.

6. Conclusions

Overall, the work presented in the rfnoc-hls-neuralnet OOT module represents a modest beginning along the path of deploying real-time, high rate, low power, and useful neural networks for RF communications research and prototyping. A few of the architectural kinks have been worked out, including 1) HLS optimizations for both fully-connected and 1D-convolutional layers, and 2) streaminglining the RFNoC FPGA integration process for adding HLS outputs into a user's CE. The end result is a library that is capable of synthesizing and interfacing some neural networks with ease.

The unique contribution of the rfnoc-hls-neuralnet library is an open source solution for efficiently developing neural networks to target FPGAs, specifically designed for RF

communications research and prototyping. It is the author's hope that by providing the open source rfnoc-hls-neuralnet module, community interest and effort might help guide additional development in this area.

References

Braun, Martin, Pendlum, Jonathan, and Ettus, Matt. Rfnoc: Rf network-on-chip. *Proceedings of the GNU Radio Conference*, 1(1), 2016. URL `http://pubs.gnuradio.org/index.php/grcon/article/view/3`.

DARPA. The radio frequency spectrum + machine learning = a new wave in radio technology, 2017. URL `https://www.darpa.mil/news-events/2017-08-11a`.

Gschwend, David. Zynqnet: An fpga-accelerated embedded convolutional neural network. Master's thesis, ETH Zürich, 2016.

Kawamoto, Darek and McGwier, Robert. Rigorous moment-based automatic modulation classification. *Proceedings of the GNU Radio Conference*, 1 (1), 2016. URL `http://pubs.gnuradio.org/index.php/grcon/article/view/7`.

Knapp, David W. *Behavioral Synthesis: Digital System Design Using the Synopsys Behavioral Compiler*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996. ISBN 0-13-569252-0.

Kreinar, E.J. Rfnoc-hls-neuralnet, 2017. URL `https://github.com/Xilinx/RFNoC-HLS-NeuralNet`.

O'Shea, Timothy J. and Hoydis, Jakob. An introduction to machine learning communications systems. *CoRR*, abs/1702.00832, 2017. URL `http://arxiv.org/abs/1702.00832`.

Suda, Naveen, Chandra, Vikas, Dasika, Ganesh, Mohanty, Abinash, Ma, Yufei, Vrudhula, Sarma, Seo, Jaesun, and Cao, Yu. Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pp. 16–25, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3856-1. doi: 10.1145/2847263.2847276. URL `http://doi.acm.org/10.1145/2847263.2847276`.

Yang, Faquan, Li, Zan, Li, Hongyan, Huang, Haiyan, and Pan, Zhongxian. Method of neural network modulation recognition based on clustering and polak-ribiere algorithm. *Journal of Systems Engineering and Electronics*, 25(5):742–747, Oct 2014. doi: 10.1109/JSEE.2014.00085.

Zhang, Chen, Li, Peng, Sun, Guangyu, Guan, Yijin, Xiao, Bingjun, and Cong, Jason. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pp. 161–170, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3315-3. doi: 10.1145/2684746.2689060. URL `http://doi.acm.org/10.1145/2684746.2689060`.

Zhao, Ritchie, Song, Weinan, Zhang, Wentao, Xing, Tianwei, Lin, Jeng-Hau, Srivastava, Mani, Gupta, Rajesh, and Zhang, Zhiru. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pp. 15–24, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4354-1. doi: 10.1145/3020078.3021741. URL `http://doi.acm.org/10.1145/3020078.3021741`.