Vol.2018-HPC-165 No.24
2018/7/31

Towards Accelerating Deep Neural Network Training on FPGAs: Facilitating the Use of Variable Precision

(Unrefereed Workshop Manuscript)

Erwin de Haan[1,2,a)] Artur Podobas[1,b)] Satoshi Matsuoka[1,3,4,c)]

Abstract: The race for larger and deeper neural networks are leading researchers, vendors and practitioners to rethink architectural design decisions taken decades ago in hope to improve performance. Among these decisions, reducing the numerical format is thought to be one of prime candidates to increasing performance. Unfortunately, modern hardware has limited support for this, and the impact of modifying floating-point formats and its size remains shrouded in mystery. To help investigate what the effects of varying the precision during deep-learning training are, first an architecture using reconfigurable hardware needs to be developed. Through this work, we seek to accelerate arbitrary precision deep-learning training using Field-Programmable Gate-Arrays by leveraging the Intel FPGA SDK for OpenCL. We present our design decision and the future outlook of our framework, and quantitively describe its resource consumption and inferrence performance on modern FPGAs.

1. Introduction

The present paper describes our efforts toward a customized and generalized framework for design-space exploration using alternative numerical floating-point formats through FPGAs.

The recent explosion in artificial intelligence – in particular that of deep neural networks based on back-propagation – has triggered a storm in the introduction and creation of specialized compute devices. Commercial platforms such as Microsoft's BrainWave [3], Google's TPUs [4], and Fujitsu's DLU [5] are all examples of specialized circuitry dedicated to low-power, high performance deep-learning (DL) training and/or inference. Meanwhile, existing general-purpose manufacturers are empowering their architecture with custom floating point units that trade precision for performance, such as Intel's Knights-Mill [6] and NVIDIA's Volta-100 [7]. It is clear that artificial intelligence and deep-learning will have a prioritized presence in modern architecture – today and in the near future.

In the pursuit for faster and more energy efficient deep-learning architecture, there is a need to critically review long-standing architectural design decisions taken by computer architects decades ago. One of the oldest design decision concerns the representation of real-valued numbers, otherwise known as the floating-point representation. The IEEE-754 floating-point representation is one of the few relics that remain unchanged in computing today. Reducing the size of the floating-point representation can have dramatic (and positive) impacts on the performance: more compute per unit silicon, more compute per unit bandwidth, and lower power consumption. Several alternatives to the IEEE-754 format are indeed emerging, such as Microsoft's deep-learning format [3], Intel's FlexPoint [8], Google's custom TPU format [9], and Posits [10].

However, exploring the space around IEEE-754 floating-point representations and its alternatives pose a significant engineering problem: hardware and software infrastructure is to tightly coupled to the IEEE-754 standard that changing any part of it incurs a high engineering overhead. One alternative is to simulate alternative floating-point representations in software, for example using soft-float or MPFR libraries [11], but the resulting performance is often several magnitudes lower than hardware implementations, limiting the size of the study conducted. Furthermore, simulating alternative numerical formats in software are incapable of using any compiler optimizations, nor can they leverage the vector instruction often crucial to reach application performance in modern processors. Porting the full software infrastructure stack is possible (and inevitable), but is a non-trivial effort that requires changes to the compiler, standard libraries and (possibly) the Application-Binary-Interface (e.g. calling convention).

A better way to explore floating-point representation is to leverage Field-Programmable Gate-Arrays (FPGAs). An FPGA is a device consisting of several millions of re-programmable look-up tables (LUTs) that together with programmable routers give a very malleable silicon substrate second only in performance to Application-Specific Integrate Circuits (ASICs). Modern FPGAs

[1] Tokyo Institute of Technology, Meguro-ku, Tokyo 152–8550, Japan
[2] Delft University of Technology, Mekelweg 2, 2628 CD Delft, Netherlands
[3] RIKEN Center for Computational Science, Chuo-ku, Kobe, Hyogo 650-0047 Japan
[4] AIST - Tokyo Tech Real World Big-Data Computation Open Innovation Laboratory, Meguro-ku, Tokyo 152-8550, Japan
[a)] e.r.dehaan@student.tudelft.nl
[b)] podobas.a.aa@m.titech.ac.jp
[c)] matsu@is.titech.ac.jp

can be clocked at several hundreds MHz, and contain enough compute to rival even Graphics-Processing Units (GPUs), making them ideal to study architectural design choices. Furthermore, with the recent growth in popularity of High-Level Synthesis (HLS) tools, programming these devices can be as simple as writing C/C++ code, and delegate the software to hardware transformation effort to the compiler.

In this paper, we contribute with:

- Proposed design for a generalized FPGA training architecture targeting variable numerical formats, and
- Preliminary evaluation and analysis of core computation patterns for inference with respect to performance and resource utilization

The remainder of our paper is structure in the following way: Section 2 positions our work against other work, similar efforts. Section 3 gives a background to machine learning and FPGAs, and describes our proposed architecture. Section 4 overviews our experimental methodology, and is followed by our preliminary results in Section 5. We conclude in Section 6.

2. Related Work

Field-Programmable Gate-Arrays (FPGAs) have extensively been used to probe and explore various architectural design decisions, including those of deep-learning. The large majority of existing work limits themselves to inference, primarily due to the simpler design layout (little intermediate data needs storing). Levering FPGAs has allowed researchers to decrease the number of bits allocated to the numerical representation, going as far as inferring popular networks such as AlexNET [15] using as few as single bit [22, 23] (binary) weights. Inferring networks that use few bits to represents weights (called Quantized Neural Networks [24]) requires subtle yet necessary changes to the training phase [16, 17, 24]. Lately, modern deep-learning frameworks are having support for reduced precision training, some driven by leading FPGA vendors such as Xilinx [14]. Dicecco et al. [21] are working on a framework similar to ours that trains neural networks using FPGAs while modifying the numerical format. Their work primary focus on the IEEE-754 format and convolution layers where-as we aspire to include alternative format such as posits [10]. Using posits in deep-learning was intended the focus of Langroudi et al. [12]'s work; however, they only compared the performance against fixed-point and only *stored* the weights as posit (the computation was still done using IEEE-754).

FPGAs are not unique to vary the numerical precision of deep neural networks, and several ASICs have been produced for quantized neural networks. Most of these focus only on inference (for embedded deployment) for low-power embedded domains; these include YodaNN [18], BinaryEYE [19] and ChipMunk [20]; these ASICs were shown to consume up to two magnitudes lower power compared to similar FPGA solutions.

Commercially most vendors do support some form of reduced precision mode. Intel's Knight's Mill [6] architecture replaces one of the AVX-512 vector units of Knight's Landing with a Vector Neural Network Instruction (VNNI) unit, capable of performing dot-products in a mixed IEEE-754 16-bit and 32-bit mode. A similar process (reduced precision multiplication, full precision accumulation) is also applied in NVIDIA's Volta-100 line GPUs. These mixed-precision units drastically improve the Artificial-Intelligence compute capabilities (AI-FLOPs) of modern systems, trading silicon resources for increased compute with arguably small impact on training performance. Vendors also embrace drastically different formats, notably in Intel's NERVANA [13] (likely where FlexPoint [8] is used) and Google's TPU [9] (which have more bits in the exponent compared to the mantissa).

3. A FPGA-based NN Framework

3.1 Field-Programmable Gate-Arrays

FPGAs can be used for a great variety of things, from accelerating computational algorithms to implementing a hardware design before taping out full chip. They are devices that can belong to the class of fine-grained reconfigurable devices.
FPGAs are made up of a reconfigurable fabric. The fabric they are made up of consists of vast arrays of switches and other routing hardware to connect all the functional elements, like multipliers, block RAMs and LUTs, in any way possible. The LUTs can be used to implement any mathematical function, like addition, multiplication, boolean functions and more. This makes them very suitable for the research described in this paper. It gives total freedom to design a hardware implementation to explore algorithms in a hardware setting without having to resort to software simulations and the performance limitations that follow from those.

The usual FPGA work-flow consists of writing in a hardware description language, synthesizing the design and finally place and routing the design onto the FGPA resources. This is a lengthy process and complex designs written in hardware description languages are hard to maintain. In this work we will focus on using the OpenCL interface to the FPGA. OpenCL is a framework specified by the Khronos Group[*1] in 2009, to work with all kinds of processors, accelerators and other heterogenous compute devices, in recent years the FPGA vendors have made efforts to make it easier to leverage FPGA resources by computer scientists and not only hardware designers. Now OpenCL is one of the ways to leverage high level synthesis. High-level synthesis takes a design specification in a high level language like C or in this case the modified version of C99 that is used for OpenCL kernels and creates an RTL design description for this design automatically after which the normal work-flow for synthesis, and place and route is resumed.

3.2 Neural Networks

An Artificial Neural Network (ANNs) is a system inspired by the biological nervous system – a system built around the concepts of *neuron*s as the main computational agents. Figure 1 illustrates an example of a typical "deep" neural network: a network that consists of a input layer, and output layer, and a number of *hidden* layers in-between. The number of hidden layers in mod-

[*1] Specification available at: `https://www.khronos.org/opencl/`

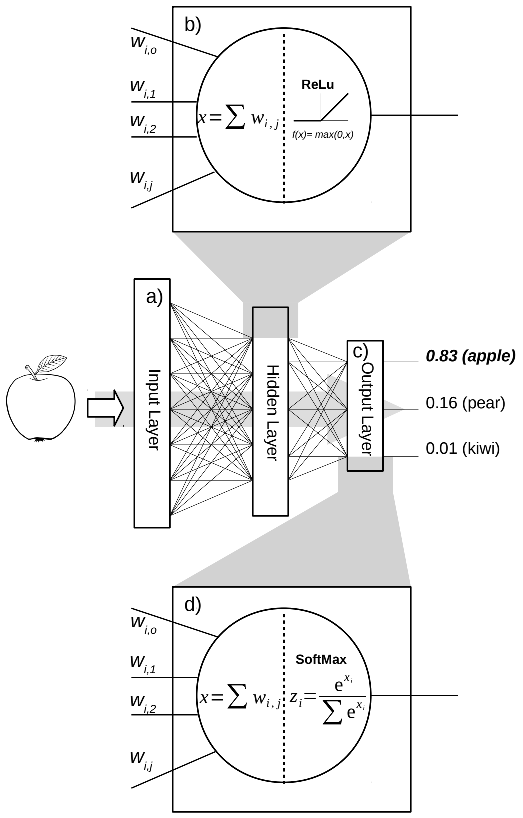

Fig. 1 The basics of artificial "deep" neural networks, showing the input (a) layer, the hidden (b) layer, and the output (c-d) layer.

ern neural networks can be very large: AlexNet (2012) [15] has eight, VGG16 (2014) [25] has 16 layers, GoogleNet (2014) [26] has 22 layers, and ResNet (2016) [27] has 152 layers. The trend seems to favor larger and deeper neural networks. The input layer of a neural network (in our example, a classifier) contains the (possibly pre-processed) inputs, such as images (for image recognition), words (in natural language processing), or motifs (in graph processing). In our humble example, the input is an image of an apple, where each pixel has an all-to-all connection to the hidden layer– a so-called *dense* layer. The dense-layer (or fully-connected layer) is the most basic of all layers, and nearly all (e.g. convolution) layers are derived from it. Each layers consists of a number of *neurons* – the main compute agent of the network. Each edge that connects to a particular neuron is called a *synapse*, and comes with a *weight* ($w_{i,j}$) in the figure, where i is the index of the neuron. The neuron integrates the weights (Fig. 1:b) of all input synapses, adds a local bias, and applies an activation function to compute the output of the neuron. The activation function varies between networks, but commonly used activation functions include: the sigmoid ($f(x) = \frac{1}{1+e^{-x}}$), the rectifier (ReLu, $f(x) = max(0, x)$), or hyperbolic tangent ($f(x) = tanh(x)$). In Fig. 1:b (and commonly in image processing network), the simple and compute-*in*expensive ReLu activation function is used in the hidden-layers of the network. The final layer of our classifier neural network is an output layer, Fig. 1:c. In the output layer, the activation function is often a *softmax* (Fig. 1:d), which scales all outputs such that their sum equals one.

Inferring (predicting) using a deep neural network is done by connecting the input to what is to be predicted, and sequentially activating each layer – from the front to the back of the network – and propagating the output of each layer into the next. This effort is called *forward propagation*. Training the network is done in a similar but reverse fashion – errors are calculated at the output and propagated backwards. At this point in time, our framework focuses on forward propagation.

3.3 FPGA Neural Network Framework

We aspire to create a neural network framework that allows design-space exploration with different numerical data formats. Our framework, illustrated in Fig. 2, consists of a number of components. We have chosen to rely on Python as the high-level frontend for our framework. It is through Python that a user creates networks, selects inputs, and decides what data formats to use. Once a particular configuration is selected, our framework generates data-type oblivious training and inference code that targets Intel's OpenCL SDK for FPGAs. By data-type oblivious we mean that all operations associated with any particular data-type representation remain unresolved at the point of creation, as they will not depend on any of the native data-types or functionality of the OpenCL language. Instead, our Python framework invokes a data-type selector, which automatically generates a low-level hardware description language (HDL) library that contains commonly used operators for the selected data-type. Once the Register-Transfer Level (RTL) library has been generated, it is merged with the data-type oblivious OpenCL template to yield the final description, which now can be synthesized using EDA tools and used. Invisible to the user, our host runtime system handles all communication, invocation and control over the FPGA from the host side.

The present paper focuses on the IEEE-754 part of inference, where training and alternative data-types are on-going and future work. Although we do have a posit [28] generator ready to be used, it has yet to be integrated into our flow.

3.3.1 Design overview

Figure 3 shows an overview of the system. The backwards kernels are a work in progress, so for now we leverage external frameworks for training. The backwards kernels are the kernels that calculate the error and back propagate the error while updating the weights and biases in the network used during training. The host code – written in Python – does one forward kernel call for every layer during inference. Finally it does a final softmax kernel call to get the final prediction. For performance this could obviously be done on the host, given the limited size of most final layers, but performance is not the main goal of this effort. The main goal is to simplify changing the numerical format, and thus all the calculations need to be done in that format, and an FPGA is better at accommodating this without a huge performance impact.

3.3.2 Design details

All weight matrices are transposed in memory, this to optimize the memory access pattern of the matrix-matrix multiplication. The main inference is performed using an blocking implementation of the matrix multiplication. First a block is fetched from the matrices containing the activations and weights and is placed

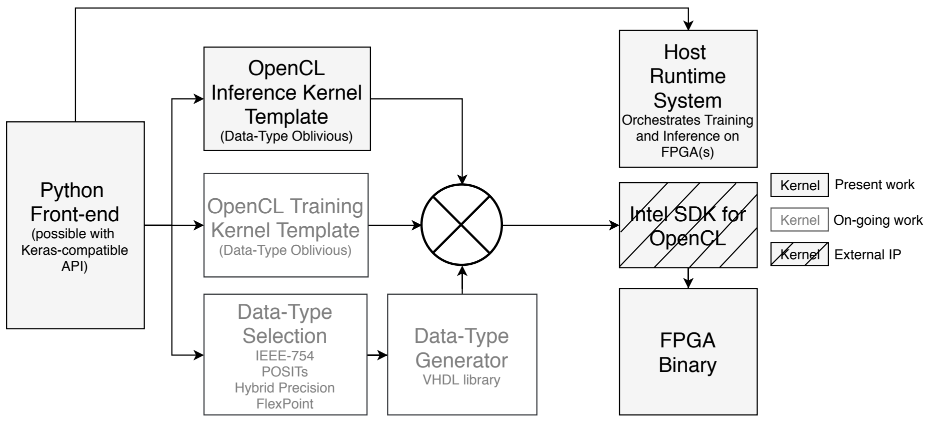

Fig. 2 Overview over our framework, showing blocks existing (black) and components currently worked on (gray).

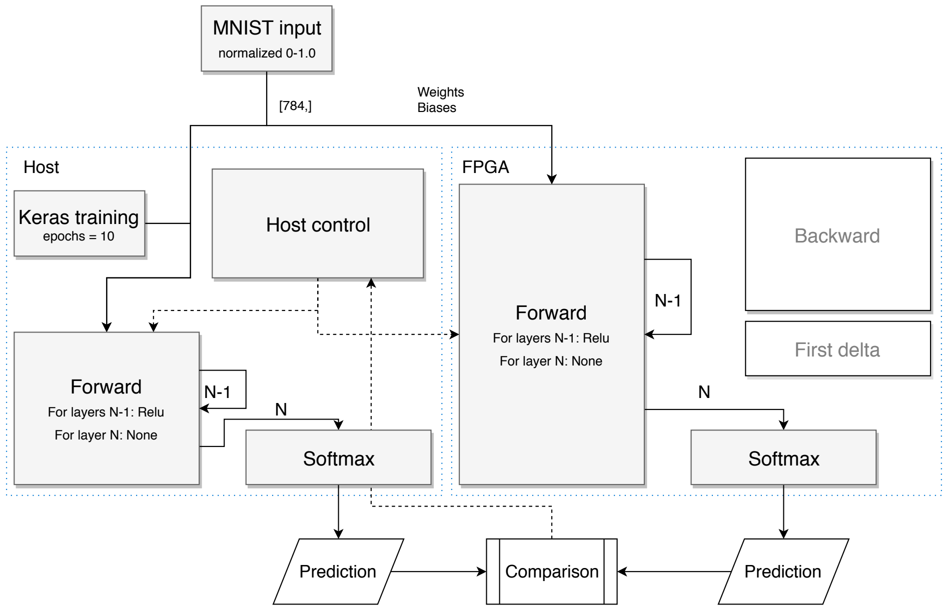

Fig. 3 Block diagram of the implementation, on the left the host code, and on the right the device code. The host side orchestrates all the computation on the device side.

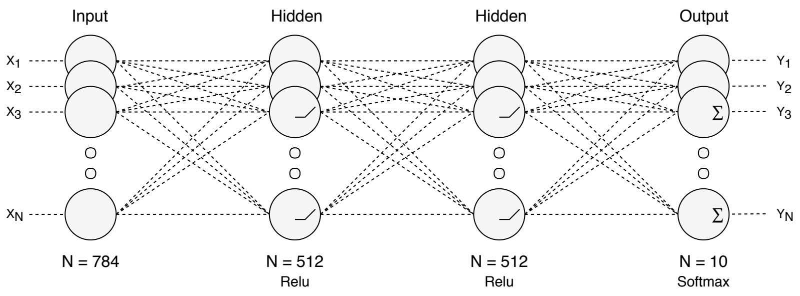

Fig. 4 MNIST sample network configuration

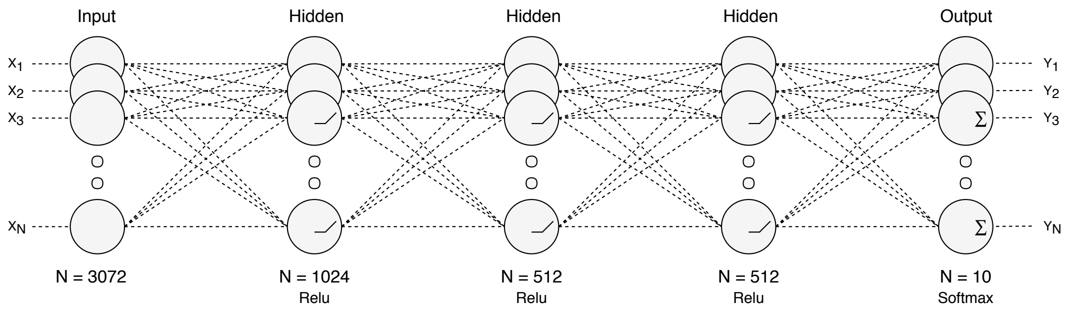

Fig. 5 CIFAR-10 sample network configuration

in the block RAM on the FPGA. The computation result is also stored on the block RAM, before a routine writes it back to global memory. The block RAM has a much lower latency than the global memory. This increases the throughput by a significant margin, almost 3× in a naive implementation. Block RAM is equivalent to the shared or local memory of contemporary GPUs.

The OpenCL SDK from Intel used for this design tries to pipeline loops in single work item kernels, as opposed to NDRange kernels (that are more truthful to the SIMT paradigm). To get maximum performance the aim is to get an initiation interval of one. That is to say that a new loop iteration is scheduled every clock cycle. To get this to work on the inner most loop – one that is essentially a reduction – an array is used to store the output of every iteration in separate locations, which enables the loop to be fully unrolled. For the loops surrounding the inner most loop, serial data-dependencies are to be avoided whenever possible, so different loop iterations are fully data-independent. For the outermost loop that processes one block this behavior is not possible, since the output block is written to multiple times. Multiple different blocks can theoretically be processed on the FPGA should space allow for it. Future revisions of this framework will also include a training phase, and we thus postponed further parallelization efforts until the kernel is complete.

3.3.3 Design decisions

One of the main choices with the largest impact on implementation is whenever to keep all weight matrices in memory transposed. During inference this is very beneficial, as it makes most reads sequential. In early stages of development it sped up the kernel performance by a factor of 3. During training, using stochastic gradient descent, this can be a problem though, since the weight matrix should now be transposed back for the most efficient memory access patterns. But since the arithmetic density of the training process is higher, the effect might be slightly less pronounced. Another decision was to execute the outermost loop, the one that loops through all the layers, to the host. This loop cannot be pipelined due to a serial data-dependency, so it makes more sense to execute this loop on the host CPU. This saves space on the FPGA and simplifies the hardware design.

4. Methodology

4.1 Experimental Platform

The software versions and hardware configuration are shown in respectively Table 1 and Table 2. All kernels were compiled with the flags "`-O3 -fp-relaxed`". "-O3" is an optimization flag, configuring the compiler to enable resource-driven optimizations. The "`-fp-relaxed`" flag gives the compiler some more freedom with the order of operations using a balanced tree hardware implementation, this gives about a 10–15 % performance benefit.

Table 1 Used software versions

Software	Version
Python	v3.6.3
Keras	v2.2.0
TensorFlow	v1.8.0
Intel Quartus	v17.1.0
Intel OpenCL	v1.1

Table 2 Test hardware

Hardware	Configuration
Intel i7-3930K	3.20 GHz
DDR3 RAM	1333 MHz Dual Channel
Arria 10 (×1)	Nallatech 510T

4.2 Test Networks and Inputs

The used example networks are multi-layer perceptrons (MLPs). This means that all layers are arrays of neurons and every neuron is connected to every neuron in the next layer. These layers are often called "Dense" layers. The first data set used was MNIST [1], MNIST is a collection of 28×28 gray scale images of hand written arabic numerals. So the data set provides 784 inputs. The structure of the sample MNIST network is shown in Fig. 4. In total a three layer network plus the input layer, or three sets of weight matrices and bias vectors. The data set was trained using the first 60 000 images from the MNIST data set. The last 10 000 images are used as the test data set.

The second data set used was CIFAR-10 [2], CIFAR-10 is a collection of 32×32 color images divided into 10 classes, providing 3072 inputs. The structure of the sample CIFAR-10 network is shown in Fig. 5. The data set was trained using the first 50 000 images from the data set. The last 10 000 images were used as the test data set. All training was done using a batch size of 128. We implemented the CPU code based on NumPy's array.dot function.

5. Results

The host (NumPy based) code benchmarked at 60 GFLOPS for straight-forward inference of the first network and about 160 GFLOPS for the second network. A first naive implementation on the FPGA clocked in at about 2.5 GFLOPS for this problem size. Transposing the weight matrix sped up the kernel to around 7.5 GFLOPS. The results for the different final FPGA kernels are shown in Table 3.

Table 3 FPGA Kernel performance

Block size	GFLOPS MNIST	GFLOPS CIFAR-10
64×64	17.8	18.9
16×16	4.3	4.4

The performance is memory-bound for these kernels. However, given our experience with alternative numerical formats (most notably posits [28]), we do not expect performance to be degraded when changing numerical format.

5.1 FPGA Resource Utilization

The kernel resource utilization is shown in Table 4. The main limiting factor is the available block RAM. To stop the calculation of extra padding lines in the matrices, the block size should be a multiple of most of the layer sizes. Larger block sizes would exhaust the block RAM on the used FPGA device. The next logical size up from 64 would be 84 (3×28) would use to much of the block RAM already. This is because there are three block buffers and the relationship between block size and used block RAM is square.

Table 4 FPGA Hardware utilization for different block sizes

Block size	Logic	DSPs	RAMs	Fmax
64×64	33 %	10 %	68 %	172.8 MHz
16×16	17 %	3 %	30 %	209.16 MHz

5.2 Inferrence Performance

The MNIST network test set accuracy after training on the CPU using Keras is 97.9 %. And the CIFAR-10 accuracy after training is 47.9 %. The FPGA gets slightly different floating point results due to rounding differences but is matches the accuracy.

6. Conclusion

We have introduced and described our efforts to create a general-purpose deep-learning framework for design space exploration of numerical formats. While still in the early stages, we have focused on generality. We have shown interoperability with Keras, where we can train weights offline and import them into our framework with little effort. We evaluated the performance and resource consumption of our design on the FPGAs, and made sure that the inferred accuracy is correct. We are currently working on implementing the training phase, as well as the choice of numerical format to be used in both the inference and training part of our framework.

7. Acknowledgment

This work was partially supported by JSPS KAKENHI Grant Number JP16F16764, the JSPS Postdoctoral fellowship under grant P16764. This work was also supported by JST CREST Grant Number(s) JPMJCR1303 and JPMJCR1687.

References

[1] LeCun, Yann and Cortes, Corinna and Burges, CJ: MNIST handwritten digit database, *AT&T Labs* `http://yann.lecun.com/exdb/mnist`, Accessed: 2018-07, 2010

[2] Krizhevsky, Alex, Vinod Nair, and Geoffrey Hinton: The CIFAR-10 dataset, `http://www.cs.toronto.edu/kriz/cifar.html`, Accessed: 2018-07, 2014

[3] Chung, Eric and Fowers, Jeremy and Ovtcharov, Kalin and Papamichael, Michael and Caulfield, Adrian and Massengill, Todd and Liu, Ming and Lo, Daniel and Alkalay, Shlomi and Haselman, Michael and others: Serving DNNs in Real Time at Datacenter Scale with Project Brainwave, *IEEE Micro*, vol. 38, number. 2, pp 8–20, 2018

[4] Google Blog (online): Google Supercharges Machine Learning Tasks with TPU Custom Chip, `https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html`, Accessed: 2018-06-25

[5] Fujitsu: Post-K Development and Introducing DLU - Fujitsu, `http://www.fujitsu.com/global/Images/post-k-development-and-introducing-dlu.pdf`, 2016

[6] Bradford, Dennis and Chinthamani, Sundarama and Corbral, Jesus and Hassan, Adhiraj and Janik, Ken: Knights Mill: Intel Xeon Phi Processor for Machine Learning, *Hot Chips 2017*, 2017

[7] NVIDIA: Artificial Intelligence Architecture, `https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/`, Accessed: 2018-06-25

[8] Koster, Urs and Webb, Tristan and Wang, Xin and Nassar, Marcel and Bansal, Arjun K and Constable, William and Elibol, Oguz and Gray, Scott and Hall, Stewart and Hornof, Luke and others: FlexPoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks, *Advances in Neural Information Processing Systems*, pp 1742–1752, 2017

[9] The Next Platform: Tearing Apart Google's TPU 3.0 AI CoProcessor, `https://www.nextplatform.com/2018/05/10/tearing-apart-googles-tpu-3-0-ai-coprocessor`, Accessed: 2018-06-25

[10] Gustafson, John L and Yonemoto, Isaac T: Beating Floating Point at its Own Game: Posit Arithmetic, *Supercomputing Frontiers and Innovations*, vol. 4, num. 2, pp 71–86, 2017

[11] Fousse, Laurent and Hanrot, Guillaume and Lefèvre, Vincent and Pélissier, Patrick and Zimmermann, Paul: MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding, *ACM Transactions on Mathematical Software (TOMS)*, vol. 32, num. 2, pp. 13, 2007

[12] Langroudi, Seyed HF and Pandit, Tej and Kudithipudi, Dhireesha: Deep Learning Inference on Embedded Devices: Fixed-Point vs Posit, *arXiv preprint arXiv:1805.08624*, 2018

[13] Intel: Intel Nervana Neural Network Processor, `https://ai.intel.com/intel-nervana-neural-network-processor/`, Access: 2018-06

[14] Xilinx: Xilinx ML Suite, `https://www.xilinx.com/applications/megatrends/machine-learning.html`, Accessed: 2018-16

[15] Krizhevsky, Alex and Sutskever, Ilya and Hinton, Geoffrey E: Imagenet classification with deep convolutional neural networks, *Advances in neural information processing systems*, pp 1097–1105, 2012

[16] Courbariaux, Matthieu and Bengio, Yoshua and David, Jean-Pierr: Binaryconnect: Training deep neural networks with binary weights during propagations, *Advances in neural information processing systems*, pp 3123–3131, 2015

[17] Zhou, Shuchang and Wu, Yuxin and Ni, Zekun and Zhou, Xinyu and Wen, He and Zou, Yuheng: DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients *arXiv preprint arXiv:1606.06160*, 2016

[18] Andri, Renzo and Cavigelli, Lukas and Rossi, Davide and Benini, Luca: YodaNN: An Architecture for Ultralow Power Binary-Weight CNN Acceleration *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, num. 1, pp 48–60, 2018

[19] Moons, Bert and Bankman, Daniel and Yang, Lita and Murmann, Boris and Verhelst, Marian: BinarEye: An always-on energy-accuracy-scalable binary CNN processor with all memory on chip in 28nm CMOS, *Custom Integrated Circuits Conference (CICC), 2018 IEEE*, pp 1–4, 2018

[20] Conti, Francesco and Cavigelli, Lukas and Paulin, Gianna and Susmelj, Igor and Benini, Luca: Chipmunk: A systolically scalable 0.9 mm 2, 3.08 Gop/s/mW@ 1.2 mW accelerator for near-sensor recurrent neural network inference, *Custom Integrated Circuits Conference (CICC), 2018 IEEE*, pp 1–4, 2018

[21] DiCecco, Roberto and Sun, Lin and Chow, Paul: FPGA-based training of convolutional neural networks with a reduced precision floating-point library, *Field Programmable Technology (ICFPT), 2017 International Conference on*, pp 239–242, 2016

[22] Shimoda, Masayuki and Sato, Shimpei and Nakahara, Hiroki: All binarized convolutional neural network and its implementation on an FPGA, *Field Programmable Technology (ICFPT), 2017 International Conference on*, pp 291–294, 2017

[23] Umuroglu, Yaman and Fraser, Nicholas J and Gambardella, Giulio and Blott, Michaela and Leong, Philip and Jahre, Magnus and Vissers, Kees: Finn: A framework for fast, scalable binarized neural network inference, *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp 65–74, 2017

[24] Courbariaux, Matthieu and Hubara, Itay and Soudry, Daniel and El-Yaniv, Ran and Bengio, Yoshua: Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1, *arXiv preprint arXiv:1602.02830*, 2016

[25] Simonyan, Karen and Zisserman, Andrew: Very deep convolutional networks for large-scale image recognition, *arXiv preprint arXiv:1409.1556*, 2014

[26] Szegedy, Christian and Liu, Wei and Jia, Yangqing and Sermanet, Pierre and Reed, Scott and Anguelov, Dragomir and Erhan, Dumitru and Vanhoucke, Vincent and Rabinovich, Andrew: Going deeper with convolutions, *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp 1–9, 2015

[27] He, Kaiming and Zhang, Xiangyu and Ren, Shaoqing and Sun, Jian: Deep residual learning for image recognition, *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp 770–778, 2016

[28] Podobas, Artur and Matsuoka, Satoshi: Hardware Implementation of POSITs and their application in FPGAs *International Parallel and Distributed Processing Symposium Workshops*, to appear, 2018