Efficient Optimization and Hardware Acceleration of CNNs towards the Design of a Scalable Neuro-inspired Architecture in Hardware

The H. Vu, Ryunosuke Murakami, Yuichi Okuyama, Abderazek Ben Abdallah
Adaptive Systems Laboratory, Graduate School of Computer Science and Engineering The University of Aizu
Aizu-Wakamatsu, Japan
{d8182106, m5211126, okuyama, benab}@u-aizu.ac.jp

***Abstract*—Convolution Neural Networks (CNNs) are responsible for the major discoveries in image classification and they are considered as the core of most current computer vision systems. In the implementation of deep CNN, Field-Programmable Gate Arrays (FPGAs) offer a promising paradigm towards major leaps in computational performance while achieving high-energy efficiency. Although current CNN accelerations on FPGA have demonstrated good performance, one major issue is that previously proposed implementation do not achieve a good balance between latency, precision, and hardware complexity. In order to overcome this problem, this paper proposes a highly optimized FPGA implementation of a CNN, named NASH-CNN (Neuro-inspired ArchitectureS in Hardware for CNN). An application for handwritten digit recognition, based on MNIST dataset, is evaluated. The experiment shows that our implementation achieves better performance/accuracy/complexity balance when compared to previously proposed schemes.**

***Index Terms*—Convolutional Neural Networks; FPGA; Optimization; Hardware Acceleration; Performance-complexity Trade-offs.**

I. Introduction

Deep Neural Networks (DNNs) are now deployed for many modern AI (Artificial Intelligence) applications including computer vision [1], speech recognition [2], self-driving cars [3], cancer detection [4], gaming [5], and robotics [6], [7]. Software simulations of DNNs, such as Convolutional Neural Networks (CNNs), face the problem of scalability where biological computing systems are inherently parallel in their architecture whereas conventional systems are based on sequential processing architectures. Hardware implementations of DNNs have the advantage of computational speedup over software simulations and can take full advantage of their inherent parallelism. In particular, hardware implementations can respond to the demands of real-time and fault-tolerant applications.

In the hardware implementation of CNNs, Field-Programmable Gate Arrays (FPGAs) have often been considered as an efficient platform for prototyping and performance exploration [8]. The FPGA implementation exploits the inherent parallelism of CNNs and takes full advantage of its multiple multiply-accumulate units. In addition, FPGAs also have other benefits such as short development cycle [9], flexibility, and reconfigurability.

There have been some implemented FPGA-based accelerators for DNNs. For instance, *Zhou et al.* [10] proposed an accelerator for image recognition based on CNNs. They implemented a five-layers architecture for MNIST [11] handwritten digit recognition with 11-bits fixed-point precision on a Virtex7 FPGA. As a result, the FPGA-based CNN has a speedup of 16.42× compared to the same architecture when implemented on Matlab/CPU software platform. However, a drawback of this implementation was the low accuracy of recognition. In another work, *Ghaffari et al.* [12] proposed two architectures for the implementation of FPGA-based accelerators. One of which (called *Form1*) was suitable for small CNNs, while the other one (called *Form2*) was designed for large ones.
These systems were evaluated using High-Level Synthesis on a Xilinx Zynq FPGA. The evaluation of these systems used LeNet [11] and MNIST dataset. However, their main disadvantage is the high latency. In fact, *Form1* and *Form2* took 2 ms and 51 ms to process a single image, respectively. Another FPGA design, named *PCANet*, was also implemented by *Zhou et al.* [13]. PCANet is composed of the following components: patch-mean removal, PCA filter convolutions, binary quantization and mapping, block-wise histograms, and an output classifier using linear Support Vector Machine (SVM). When evaluated using MNIST dataset, the system took 7.588 μs to process a single sample and got an accuracy of about 99.46%. However, the system implementation used a significant amount of hardware resources.

In this paper, we propose a Neuro-inspired ArchitectureS in Hardware for CNN (named NASH-CNN[1]) which is implemented on an FPGA in order to provide an efficient balance between area cost, precision, and latency. This is the first stage of our design of a scalable based hardware for SNNs. To evaluate the proposed system, a CNN architecture for handwritten digit recognition is implemented in both NASH-CNN and CPU. Evaluation results show that the proposed system implementation provides better performance/complexity/accuracy balance when compared to other previous works.

The rest of the paper is organized as follows: Section 2

[1]NASH Project is partially supported by the University of Aizu Competitive Research Funding, CRF-2017, Fukushima, Japan

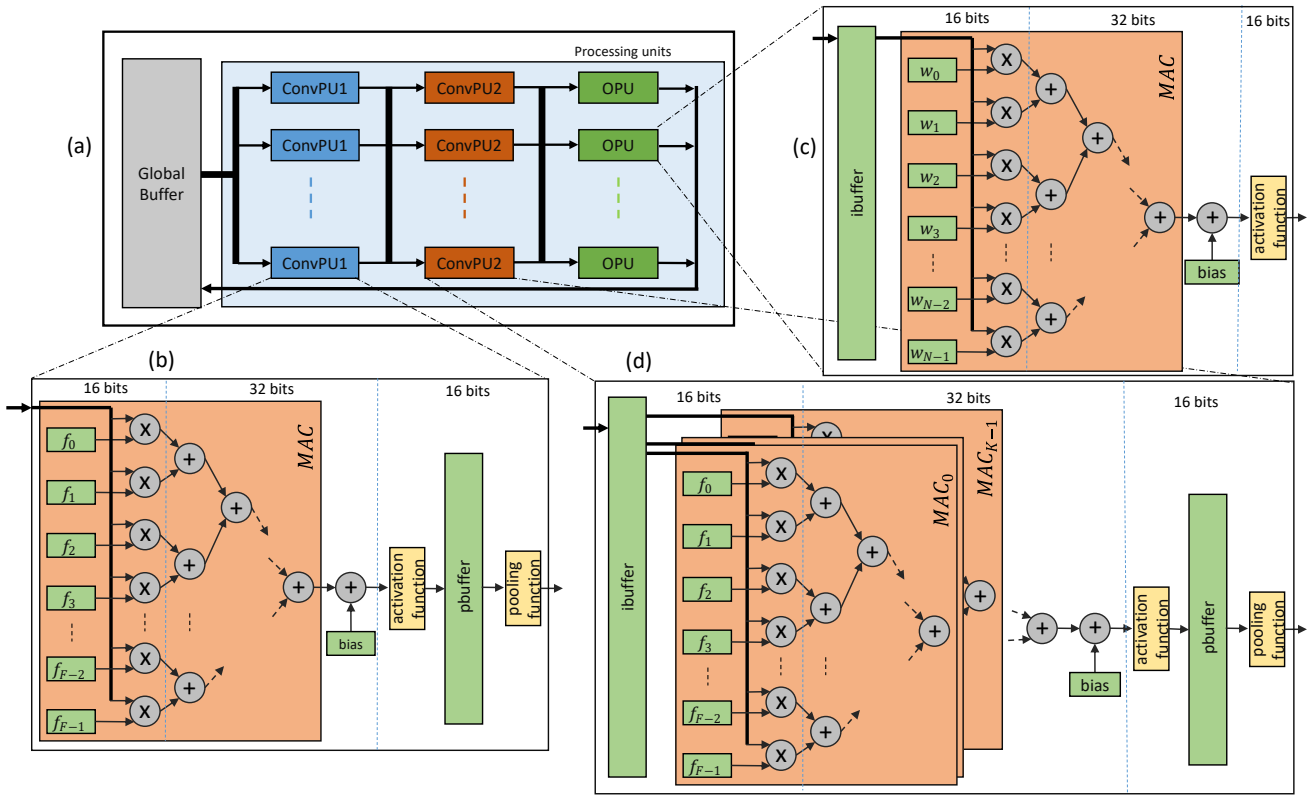

Fig. 1. NASH-CNN system architecture.

describes the CNN architecture used for the proposed hardware implementation. Section 3 covers the implementation of NASH-CNN. Section 4 presents the evaluation methodology and results. Section 5 shows deep discussion and further research towards the design of a scalable neuro-inspired architecture in hardware. Finally, Section 6 presents our concluding remarks.

II. NASH-CNN Architecture

Figure 1 (a) shows the top-level architecture of NASH-CNN. It is composed of neural *Processing units* and a *Global Buffer*. The point-to-point connections are used to connect the different elements of the system. There are three kinds of processing elements in NASH-CNN: *ConvPU1* and *ConvPU2* are designed to perform both convolutional and pooling calculations; *OPU* computes the calculation of a neuron in the output layer.

1) Global Buffer: The *Global buffer* is dedicated to buffering the input samples and the results. This buffer can communicate with DRAM through the asynchronous interface and with *ConvPU1s*, as is shown in Fig. 1 (a). The size of the buffer depends on the size of input example and its result. The buffer is designed so that *ConvPU1s* can access a part of an input map (a sliding window) in each clock cycle. As a result, there is no need for an input buffer in *ConvPU1s*.

2) ConvPU1: Fig. 1 (b) shows the architecture of a *ConvPU1*. There is only one $F \times F$ filter in each processing unit. The values of the filters are pre-calculated by software and then statically configured into *ConvPU1s*. A buffer, called *pbuffer*, is dedicated to store the convolutional output map for pooling calculation.

The *ConvPU1* performs both convolutional and pooling calculations. This means that the calculations of both convolution and pooling layers are computed in this unit. At each clock cycle, the dot-product between the input plane and the kernel is calculated in parallel; then, the result is also accumulated in parallel. The multiply-accumulate result is then fed to the *activation function* unit to perform the activation function calculation. *Pbuffer* stores the convolutional output map. Finally, *pooling function* performs the pooling calculation over the convolutional output plane. Therefore, an output matrix is generated and sent to the next computation units.

3) ConvPU2: The *ConvPU2* is composed of various components, as illustrated in Fig. 1 (d). A buffer, called *ibuffer*, is dedicated to contain output maps from the previous processing units. There are K mask filters which are pre-calculated and statically implemented in this module. Furthermore, K *multiply-accumulate* calculation units (from MAC_0 to MAC_{K-1}) are implemented to perform convolutions in parallel. Similarly to *ConvPU1*, *ConvPU2* also contains *activation function*, *pbuffer*, and *pooling* are also designed.

In addition, the sizes of the *pbuffer* in ConvPU1 and

ConvPU2 are calculated by using the following equations:

$$W_p = \frac{W_i - F + 2P}{S} + 1 \quad (1)$$

$$H_p = \frac{H_i - F + 2P}{S} + 1 \quad (2)$$

where W_p and W_i are the widths of *pbuffer* and *ibuffer*, respectively, H_p and H_i are the heights of *pbuffer* and *ibuffer* respectively, F is the filter size, P is the number of zero-paddings used in the CNN architecture, and S is the stride of the filter over the input example.

4) OPU: Represented in Fig. 1 (c), *OPU* performs the calculation between the input values and connection weights as a conventional neuron. Output maps of previous units are buffered in *ibuffer*. Like the other computation units, weights are also pre-computed and statically implemented. In addition, the multiply-accumulate units are designed to perform the calculation in parallel. Therefore, OPU takes one clock cycle to yield the output value.

III. Application for Handwritten Digit Recognition

A. Network Architecture

To evaluate the performance of NASH-CNN, a CNN architecture for handwritten digit recognition is built and run on both NASH-CNN and a general purpose CPU. The network architecture is similar to the one described in [14]. *MNIST* dataset is used for training and testing. The network includes five separate layers, as shown in Fig. 2. The parameters of the CNN architecture are summarized in Table I.

TABLE I
Parameters of the network architecture.

Layer	Kernels/Weights	Layer size
Input	-	$[28 \times 28] \times 1$
Conv1	$[5 \times 5] \times 6$	$[24 \times 24] \times 6$
Pool1	$[2 \times 2] \times 6$	$[12 \times 12] \times 6$
Conv2	$[5 \times 5] \times 6 \times 12$	$[8 \times 8] \times 12$
Pool2	$[2 \times 2] \times 12$	$[4 \times 4] \times 12$
Hidden	-	$[192 \times 1] \times 1$
Output	$[192 \times 1] \times 10$	$[1 \times 1] \times 10$

B. NASH-CNN Implementation

1) Global buffer and Neural Processing Elements: The CNN architecture described in Section III-A is implemented onto an FPGA. The system implementation is composed of a *Global Buffer*, six *ConvPU1s*, twelve *ConvPU2s*, and ten *OPUs*. The number of neural *Processing units* corresponds to the number of neurons required in the CNN architecture.

The accuracy of the calculations needs to be taken into consideration in the system implementation. The whole system is implemented using 16-bits fixed-point calculations instead of floating-point ones because of limitation of FPGA resource. In the case of *multiply-accumulate* units, the results are truncated from 32 to 16-bits after accumulation instead of multiplication. This is a guarantee for maintaining the calculation precision. On the other hand, the rounding-to-nearest-integer method is applied to reduce the error.

As previously explained, the proposed implementation contains the following components:

- **Global buffer:** this unit is designed for storing the input map and its result. In this implementation, the buffer has two sizes of $28\times28\times16$-bits and 10×16-bits for the input and output example, respectively. During the execution of *ConvPU1*, a sliding window of 5×5 in size is accessed at a single clock cycle.
- **Neural Processing elements:** parameters of the neural processing units presented in Section II are configured in order to be suitable for the adopted application. These computation units use the sigmoid function as an activation function and mean-pooling. In *ConvPU1*, there is a filter with the size of 5×5 (F is equal to 5). Besides, the *pbuffer* has a $24 \times 24 \times 16$-bits capacity to store the convolutional results. Each *ConvPU2* has a $12 \times 12 \times 16$-bits input buffer, six 5×5 filters (K is equal to 6) and an $8 \times 8 \times 16$-bits *pbuffer*. There is also an input buffer with 192×16-bits size in each *OPU* unit. This is also corresponding to the pre-configured 192 weights (N = 192).

2) Activation Function: In this implementation, the sigmoid function (equation 3) is used as an activation function of the neurons in convolutional and output layers. Because this is a nonlinear function, special attention must be paid when implementing it on FPGA.

$$y = \frac{1}{1 + e^{-x}} \quad (3)$$

There are some methods to implement the sigmoid function such as piecewise linear (PWL) approximations, piecewise second-order approximations, and combinational approximations [15]. An efficient implementation can be measured by three factors: accuracy, speed, and area cost. In this paper, a non-linear approximation [16] is employed to implement the sigmoid function. In order to improve the accuracy, the interval range of variables from -∞ to +∞ is divided into twenty-five piece-wise intervals corresponding to their functions, as is illustrated in Table II.

Compared to the work in [16], some coefficients of the piece-wise function in the implemented function, named *TqSigmoid*, are re-calculated in order to reduce the approximation error. In the interval range of [-2,-1], the approximate function is re-calculated by Matlab. Fig. 3 shows the evaluation results of the re-calculated sigmoid function and its absolute error. As is shown in Fig. 3 (c), the absolute error of *TqSigmoid* is sharply reduced after re-calculation. Furthermore, the mean squared error of the *TqSigmoid* is equal to 1.09×10^{-6}.

3) Learning Algorithm: There are two learning techniques often employed to implement a CNN on FPGAs. In the on-chip learning, where weights are directly calculated and determined in hardware; thus, it does not require any other computation platforms during the CNN training. As a result, the implemented system can adapt to different applications.

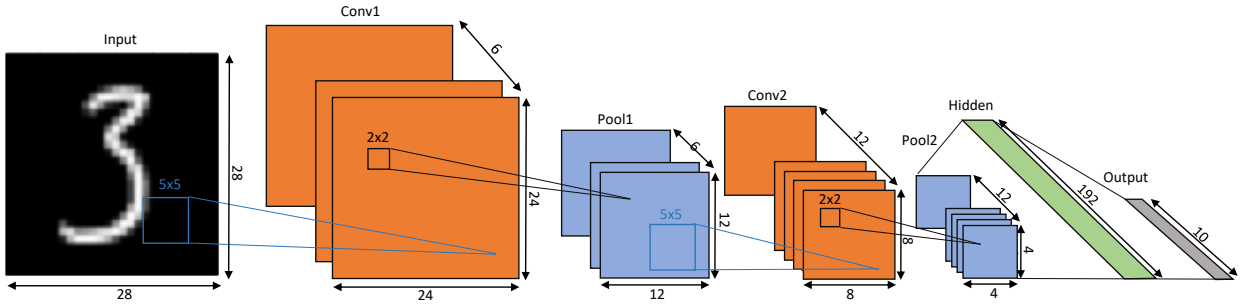

Fig. 2. Network architecture for MNIST evaluation.

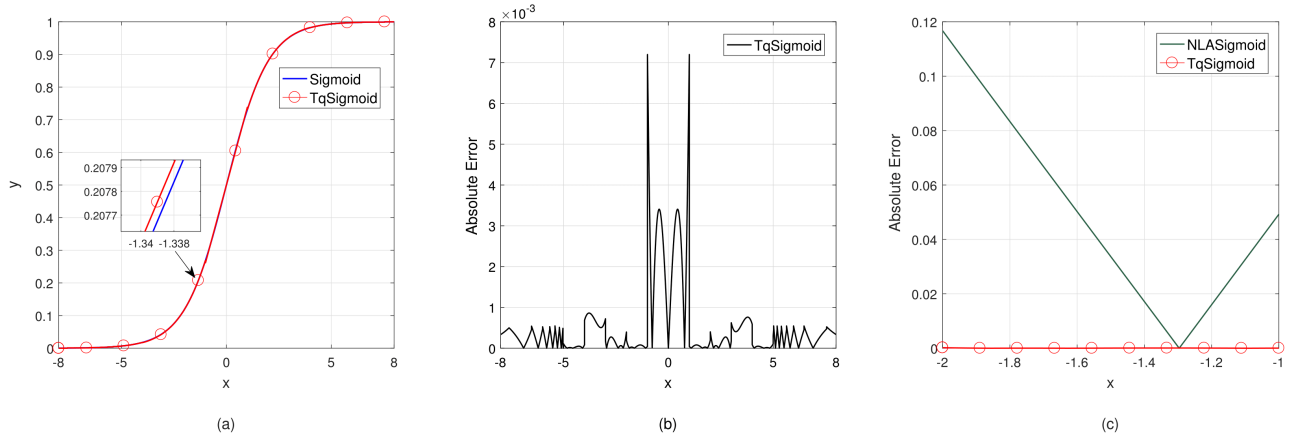

Fig. 3. Evaluation results of the approximate sigmoid function: (a) *TqSigmoid* and the sigmoid function (b) absolute error of *TqSigmoid* (c) absolute error of *NLASigmoid* [16] and *TqSigmoid* in the piece-wise interval of [-2:-1].

TABLE II
THE INTERVAL AND FUNCTIONS FOR THE SIGMOID FUNCTION APPROXIMATION.

Interval	Function Form
(1,1)	$y = 0.2383x + 0.5000$
[-2,-1]	$y = 0.0467x^2 + 0.2896x + 0.5118$
[-3,-2)	$y = 0.0298x^2 + 0.2202x + 0.4400$
[-4,-3)	$y = 0.0135x^2 + 0.1239x + 0.2969$
[-5,-4)	$y = 0.0054x^2 + 0.0597x + 0.1703$
[-5.03,-5)	$y = 0.0066$
[-5.2,-5.03)	$y = 0.0060$
[-5.41,-5.2)	$y = 0.0050$
[-5.66,-5.41)	$y = 0.0400$
[-6,-5.66)	$y = 0.0030$
[-6.53,-6)	$y = 0.0020$
[-7.6,-6.53)	$y = 0.0010$
[-∞,-7.6)	$y = 0$
[1,2)	$y = -0.0467x^2 + 0.2896x + 0.4882$
[2,3)	$y = -0.0298x^2 + 0.2202x + 0.5600$
[3,4)	$y = -0.0135x^2 + 0.1239x + 0.7030$
[4,5)	$y = -0.0054x^2 + 0.0597x + 0.8297$
[5,5.0218)	$y = 0.9930$
[5.0218,5.1890)	$y = 0.9940$
[5.1890,5.3890)	$y = 0.9950$
[5.3890, 5.6380)	$y = 0.9960$
[5.6380,5.9700)	$y = 0.9970$
[5.9700,6.4700)	$y = 0.9980$
[6.4700,7.5500)	$y = 0.9990$
[7.5500, +∞)	$y = 1$

On the other hand, weights in the case of off-chip learning are commonly pre-calculated by software. These weights are then statically configured into the FPGA. This method takes full advantage of the software calculation precision.

In the case of NASH-CNN, we use off-chip learning because of the following reasons. First, this method has lower area cost compared to the on-chip learning. This is due to the large amount of hardware resource required to perform the on-chip training process. For example, DSPs are used for operators, and registers for stored values as well as some parameters. Second, in this work, we focus on the balance between hardware complexity, accuracy, and latency in the feed-forward computation. Therefore, the implementation of on-chip learning is out of the scope of this work. Fig. 4 depicts how to implement the learning rule in NASH-CNN (see section IV-A for more details).

IV. Evaluation

A. Evaluation Methodology

In order to evaluate the performance of the proposed NASH-CNN, the CNN architecture for handwritten digit recognition is implemented in both software and FPGA. First, the network is implemented using Matlab tool running on a PC with a 3.40 GHz Intel Core-i7 4770 and 16 GB of RAM. This implementation not only aims to determine the kernels and

TABLE III
AREA COST COMPARISON.

Parameters/Systems	PCANet2015 [13]	Form12016 [12]	Form22016 [12]	Yongmei2015 [10]	Proposed NASH-CNN
Target device	Virtex-7 980T	Xilinx Zynq	Xilinx Zynq	Virtex-7 vx485tffg1761-2	Virtex-7 v2000tflg1925-1
FFs/Registers (F/R)	358848-R	54075-F	35399-F	66364-F	18299-F
LUT	265460	14832	39879	51125	65386
BRAM	64 (36E1)	27	3	0	0
DSP	3599 (48E1)	20 (48E)	90 (48E)	638 (48E)	1095 (48E1)
Precision	-	25-bits fixed-point	25-bits fixed-point	11-bits fixed-point	16-bits fixed-point

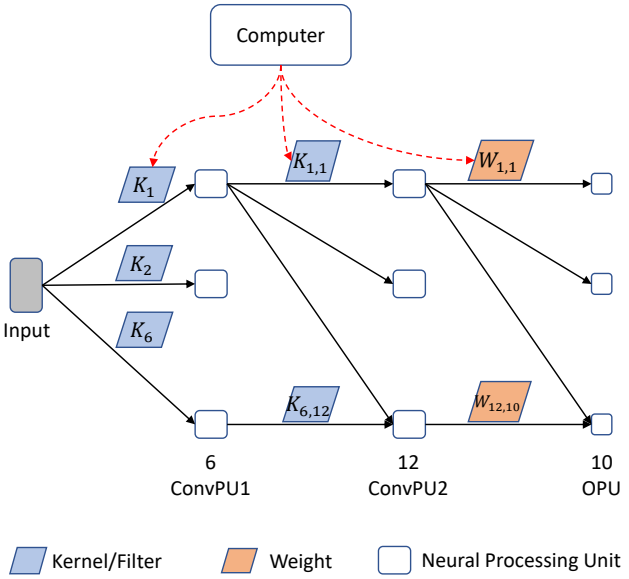

Fig. 4. Explanation of the learning algorithm implementation.

weights for off-chip learning mechanism; but, also to compare the latency between the PC and NASH-CNN platforms. The MNIST dataset is used as the training dataset containing 60,000 examples and a test set of 10,000 examples. The network is trained by 100 epochs with the batch size of 50. The PC takes an execution time of 6025.31 s for the training task.

Second, the network architecture is designed in Verilog-HDL and synthesized with Vivado 2017.1 tool. The target device for NASH-CNN is Virtex-7 v2000tflg1925-1. As previously mentioned, the system is implemented by using 16-bits fixed-point calculations and the network is trained by using off-chip learning method. This means that the weights are trained in software at a previous stage, then they are statically configured into FPGA.

TABLE IV
NASH-CNN POWER ESTIMATION.

Power consumption	Power (mW)
Dynamic	382
Static	641
Total	1,023

B. Complexity Evaluation

1) Area cost: Table III shows the comparison results between the proposed system and other implementations in terms of area cost. These architectures were also evaluated by using the MNIST dataset. In overall, *PCANet2015* [13] used the largest amount of hardware resource in terms of all parameters when compared to the other implementations. NASH-CNN requires the smallest number of flipflops while using more LUTs and DSPs when compared to *Form12016* [12], *Form22016* [12], and *Yongmei2015* [10].

2) Power consumption: Table IV shows the power estimation of NASH-CNN system. We use Xilinx Vivado tool perform the power estimation. The total power consumption of the system is 1,023 mW, in which the dynamic part occupies over one-third of the total power.

C. Performance Evaluation

1) Latency: Fig. 5 compares the implemented systems in terms of latency. *PCANet2015* is the fastest system when compared to the others. It takes only 7.58us to calculate a single example. Whereas, *Form12016* and *Form22016* show higher latency, reaching $2,000$ μs and $51,000$ μs to process the same single example, respectively. Using the same CNN architecture, the latency in [10] is lower when compared to NASH-CNN, reaching up to 25.43 μs. The proposed FPGA classifier was compared to the PC Matlab implementation. The software implementation spends an execution time of 492 μs for an input example while the proposed NASH-CNN requires only 32.04 μs. This means that our FPGA based implementation achieves up to a 15.35× speedup over the software one.

2) Accuracy: As shown in Fig. 6, *PCANet2015* achieves the highest accuracy amongst the compared implementations. This is thanks to its network architecture which is different than the others. Both *Yongmei2015* and NASH-CNN use the same CNN architecture. Nevertheless, NASH-CNN provides higher accuracy.

V. DISCUSSION

To current proposed architectural optimization is suitable for CNNs with two convolution-pooling layers and one fully-connected layer. As shown in Figure 1, the system has three separate layers, in which computation units are connected via point-to-point. Therefore, it is easy to mapping CNNs which have the same number of layers compared to the

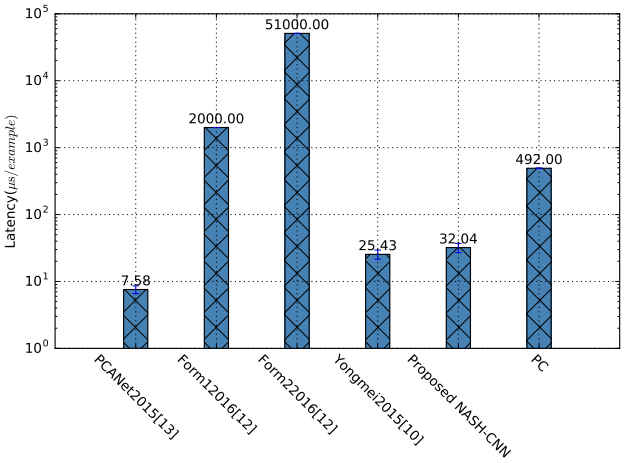

Fig. 5. Performance comparison.

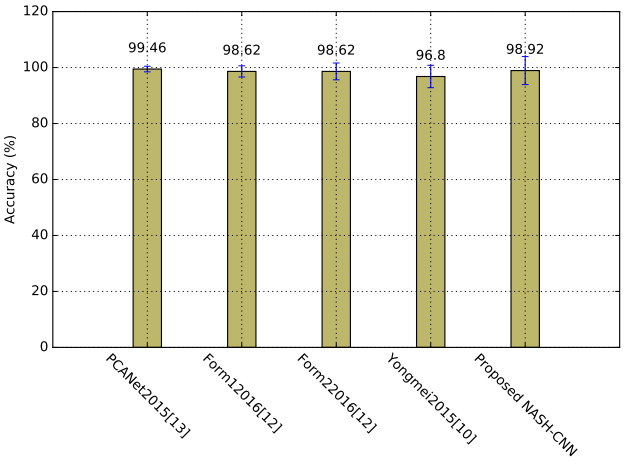

Fig. 6. Accuracy comparison.

proposed system. Besides, the system can only be expanded by increasing the number of neurons in each layer.

On the other hand, the system can be also configured in order to implement multiple-layer CNNs. Since the hardware resource of FPGAs is limited, area cost needs to be taken into consideration when implementing. Adding more layers of computation units leads to consuming a large the amount of hardware resource. Hence, the current three-layers architecture will be kept during implementation. In that case, computation units are configured and considered as a computation frame. CNNs are then split into every single three-layers frame before mapping into the system. For example, a six-layers CNN can be divided into two three-layers parts. In the first stage of computation, the input map and filters are loaded from the global buffer. After calculated, the output map will be sent back to the buffer for the next calculation. By applying this technique, though there is an increase in execution time of the system, it does not consume more hardware resource of FPGAs.

As a future work, we plan to investigate a scalable spiking neuro-inspired architecture in hardware. The architecture will be based on our earlier proposed 3D network-on-chip (NoC) [17]–[19] and supports Random-Access-Buffer (RAB), Bypass-Link-on-Demand [20], [21] techniques. The idea is based on stacking planar dies on top of one another and connecting them with Through-Silicon Vias (TSVs). This architecture aims demonstrate that the problem of limited scaling of interconnect in reconfigurable hardware can be alleviated through the use of a 3D-NoC. As a result, the communication latency would be enormously reduced. The NoC [17]–[19] paradigm is particularly attractive for spiking neural networks as it offers scalability, parallelism, and flexibility. In a NoC system, information is transmitted as packets, allowing the nodes on the network to implement advanced features such as prioritization and load balancing.

VI. Conclusion

In this paper, we presented an FPGA-based acceleration and optimization for CNNs. A handwritten digit recognition application is evaluated in both software and the proposed NASH-CNN. 16-bits fixed-point calculations were used to implement the system components. During the implementation, the calculation precision was paid attention in order to reduce the calculation error. Evaluation results show that NASH-CNN achieved a speed-up of more than 15× when compared to general-purpose CPU-based implementation. When compared to previously proposed implementations, NASH-CNN provides an efficient trade-off between area cost, accuracy, and latency. Further more, the current architecture is our first step towards the design of scalable spiking architecture in hardware.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems*, ser. NIPS'12. USA: Curran Associates Inc., 2012, pp. 1097–1105.

[2] L. Deng, J. Li, J. T. Huang, K. Yao, D. Yu, F. Seide, M. Seltzer, G. Zweig, X. He, J. Williams, Y. Gong, and A. Acero, "Recent advances in deep learning for speech research at microsoft," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, May 2013, pp. 8604–8608.

[3] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *2015 IEEE International Conference on Computer Vision (ICCV)*, Dec 2015, pp. 2722–2730.

[4] A. Esteva, B. Kuprel, R. A. Novoa, J. Ko, S. M. Swetter, H. M. Blau, and S. Thrun, "Dermatologist-level classification of skin cancer with deep neural networks," *Nature*, vol. 542, no. 7639, pp. 115–118, Jan. 2017.

[5] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016.

[6] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-end training of deep visuomotor policies," *J. Mach. Learn. Res.*, vol. 17, no. 1, pp. 1334–1373, Jan. 2016. [Online]. Available: http://dl.acm.org/citation.cfm?id=2946645.2946684

[7] M. Pfeiffer, M. Schaeuble, J. I. Nieto, R. Siegwart, and C. Cadena, "From perception to decision: A data-driven approach to end-to-end motion planning for autonomous ground robots," *arXiv preprint arXiv:1609.07910*, 2016.

[8] G. Lacey, G. W. Taylor, and S. Areibi, "Deep learning on fpgas: Past, present, and future," *arXiv preprint arXiv:1602.04283*, 2016.

[9] K. Cheung, S. R. Schultz, and W. Luk, "Neuroflow: A general purpose spiking neural network simulation platform using customizable processors," *Frontiers in Neuroscience*, vol. 9, 2016.

[10] Y. Zhou and J. Jiang, "An fpga-based accelerator implementation for deep convolutional neural networks," in *2015 4th International Conference on Computer Science and Network Technology (ICCSNT)*, vol. 01, Dec 2015, pp. 829–832.

[11] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," vol. 86, no. 11, Nov 1998, pp. 2278–2324.

[12] S. Ghaffari and S. Sharifian, "Fpga-based convolutional neural network accelerator design using high level synthesize," in *2016 2nd International Conference of Signal Processing and Intelligent Systems (ICSPIS)*, Dec 2016, pp. 1–6.

[13] Y. Zhou, W. Wang, and X. Huang, "Fpga design for pcanet deep learning network," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2015, pp. 232–232.

[14] R. B. Palm, "Prediction as a candidate for learning deep hierarchical models of data," Master's thesis, Technical University of Denmark, DTU Informatics, 2012.

[15] M. T. Tommiska, "Efficient digital implementation of the sigmoid function for reprogrammable logic," *IEE Proceedings - Computers and Digital Techniques*, vol. 150, no. 6, pp. 403–411, Nov 2003.

[16] Z. Xie, "A non-linear approximation of the sigmoid function based on fpga," in *2012 IEEE Fifth International Conference on Advanced Computational Intelligence*, Oct 2012, pp. 221–223.

[17] A. B. Abdallah and M. Sowa, "Basic Network-on-Chip Interconnection for Future Gigascale MCSoCs Applications: Communication and Computation Orthogonalization," 2006.

[18] K. N. Dang, A. B. Ahmed, Y. Okuyama, and A. B. Abdallah, "Scalable design methodology and online algorithm for tsv-cluster defects recovery in highly reliable 3d-noc systems," *IEEE Transactions on Emerging Topics in Computing*, vol. PP, no. 99, pp. 1–1, 2017.

[19] K. N. Dang, A. B. Ahmed, X. T. Tran, Y. Okuyama, and A. B. Abdallah, "A comprehensive reliability assessment of fault-resilient network-on-chip using analytical model," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 11, pp. 3099–3112, Nov 2017.

[20] A. B. Ahmed and A. B. Abdallah, "Adaptive fault-tolerant architecture and routing algorithm for reliable many-core 3d-noc systems," *Journal of Parallel and Distributed Computing*, vol. 93-94, no. Supplement C, pp. 30 – 43, 2016.

[21] A. B. Ahmed, M. Meyer, Y. Okuyama, and A. B. Abdallah, "Adaptive error- and traffic-aware router architecture for 3d network-on-chip systems," pp. 197–204, Sept 2014.