2021 IEEE International Workshop on Rapid System Prototyping (RSP) | 978-1-6654-6956-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/RSP53691.2021.9806200

FPGA Prototyping of Systolic Array-based Accelerator for Low-Precision Inference of Deep Neural Networks

Soobeom Kim
Dept. of Computer Science and Engineering
Seoul National University
Seoul, Korea
soobeom15@gmail.com

Seunghwan Cho[§]
System LSI Business
Samsung Electronics
Hwaseong, Korea
seunghwan.cho21@gmail.com

Eunhyeok Park
Graduate School of Artificial Intelligence
Department of Computer Science
POSTECH
Pohang, Korea
canusglow@gmail.com

Sungjoo Yoo
Dept. of Computer Science and Engineering
Seoul National University
Seoul, Korea
sungjoo.yoo@gmail.com

***Abstract*—In this study, we aim to design an energy-efficient computation system for deep neural networks on edge devices. To maximize energy efficiency, we design a novel hardware accelerator that supports low-precision computation and sparsity-aware structured zero-skipping on top of the well-known systolic-array structure. In addition, we introduce a full-stack software platform, including a model optimizer, instruction compiler, and host interface, to translate the pre-trained PyTorch model to the proposed accelerator and orchestrate it automatically. We validate the entire system by prototyping the accelerator on the Xilinx Alveo U250 FPGA board and demonstrating the inference of the 4-bit ResNet-50 model through the software stack. According to our experiment, our platform shows 317 GOPS inference speed and 51.96 GOPS/W energy efficiency for ResNet-50 on Xilinx Alveo U250 FPGA at 108 MHz, which is comparable to the advanced commercial acceleration system in terms of energy efficiency.**

***Index Terms*—Neural network, convolutional neural network, low precision inference, neural processing unit, quantization, NPU on FPGA**

I. Introduction

Owing to the diverse usage of deep neural networks in edge devices, hardware acceleration is becoming increasingly important. In particular, accelerators on edge devices focus on energy efficiency due to tight resource constraints. Unlike the server-target accelerator that generates precise output and supports various operations [1], [2], edge-target accelerators limit the degree of freedom and focus on specialized features for energy efficiency [2]–[11].

Among the candidates, low-precision computation and sparsity-aware zero-skipping are the most promising optimization methods that can provide performance and energy benefit with minimal hardware overhead. By applying low-bit precision quantization to the deep neural network model, hardware accelerators can utilize more computation units and buffers within the same area, thereby achieving better energy efficiency [2], [9], [11]. Numerous studies related to quantization [12]–[14] reported that 8-bit quantization is possible without loss of accuracy [15], and recent studies [16]–[18] reported that 4-bit quantization is also applicable for advanced networks with negligible accuracy loss, which shows the auspicious potential of low-precision computation. In addition, sparsity-aware zero-skipping allows the skipping of ineffective operations in the neural network, associated with zero data, without quality degradation. The rectified linear unit (ReLU) introduces approximately half of the zero among

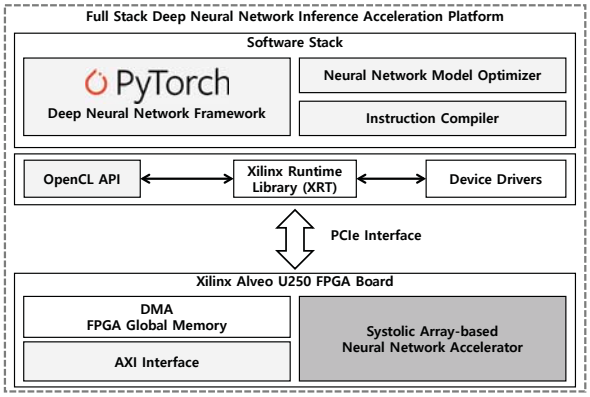

Fig. 1. Full stack deep neural network inference acceleration platform.

[§]This work was done during his Ph.D. period at Seoul National University.

978-1-6654-6956-2/21/$31.00 ©2021 IEEE

the activations, and recent pruning studies [19] reported that approximately half of the weights of convolutional layers could also be set to zero without affecting accuracy. While this sparsity guarantees the theoretical speedup without quality degradation, the corresponding hardware support is essential to realize it due to the unpredictable pattern of zero data [3], [6]–[9], [11].

To realize the benefits of low-precision computation and sparsity-aware zero-skipping, we propose a novel hardware accelerator design. It is based on a well-known competent systolic array structure but it is extended to support the aforementioned optimization methods with minimal modification. In addition, by judiciously including the essential hardware components (i.e., vector unit, quantization unit, controller, etc.) on the microarchitecture, our hardware is flexible enough to support the network inference of diverse structures without the intervention of the host. This structure enables the entire process of inference of the network on top of the proposed accelerator with high efficiency.

Besides, we also introduce the full stack of software involved in neural network mapping to hardware operation and orchestration of the accelerator functionality. The software stack includes network optimizer, instruction compiler, and host-device interface. By combining the proposed hardware and software stack, we can accelerate the pre-trained model without the intervention of human experts.

For prototyping of the proposed system, we implement the entire system on top of the actual FPGA hardware, and demonstrate network inference through the software stack. Our extensive analysis shows that the proposed system achieves 51.96 GOPS/W, which, when compared to other approaches, is highly energy efficient.

The contributions of this paper are as follows:

- We design a novel hardware accelerator that is capable of low-precision computation and sparsity-aware structured zero-skipping and is flexible enough to support various networks.
- We introduce the full-stack deep neural network inference acceleration platform, which automates model translation from PyTorch [20] to the FPGA accelerator.
- We successfully demonstrate that the inference of 4-bit ResNet-50 on our platform achieves 317 GOPS and 51.96 GOPS/W on the Xilinx Alveo U250 FPGA board [21] at 108 MHz.

The remainder of this paper is organized as follows. Section II provides an overview of the proposed inference acceleration platform. Section III describes the hardware implementation details of the proposed neural network accelerator and acceleration system. Section IV describes the software stack details of the proposed inference acceleration system. Section V describes experimental environment and reports experimental results. Finally, Section VI concludes the paper.

II. Inference System Overview

Fig. 1 shows a system overview of the inference acceleration platform. Our neural network accelerator is designed to run the inference of convolutional neural networks efficiently, where the accelerator is optimized for the convolutional and matrix multiplication operations. Also, our neural network accelerator is equipped with dedicated quantization logic to support 4-bit integer precision. In addition, we prepared several components in the software stack to automate network translation and hardware operations. For instance, we include the model optimizer to convert the pre-trained full-precision model to an optimized model, with quantization and pruning. Also, our instruction compiler is designed to convert the PyTorch [20] model into binary instructions for the neural network accelerator, and we use simple APIs to control the neural network accelerator for ease of use in terms of data loading and host interface design. This solidified system enables us to exploit the proposed system without human intervention, which is essential for pushing the boundaries of the generalization of acceleration.

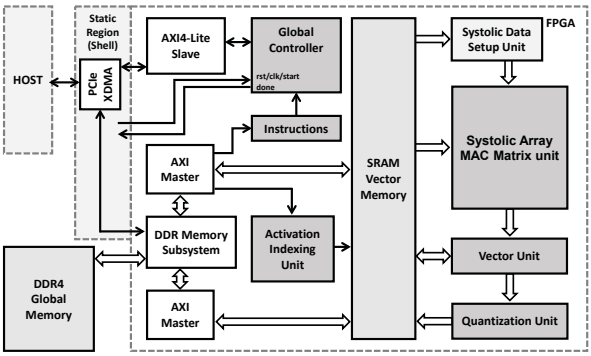

Fig. 2. Microarchitecture of neural network accelerator.

III. Hardware Implementation Details

A neural network is composed of multiple high-rank tensor operations, which can be decomposed into, and accelerated by, a hardware-friendly matrix and vector operations. To provide abundant computation performance with high energy efficiency, the proposed accelerator is equipped with a 1-D SIMD vector unit and a 2-D systolic array-based multiplier-accumulator (MAC) matrix unit. Fig. 2 shows the microarchitecture of the proposed neural network accelerator.

A. Computation Model

Compared to the SIMD, MIMD, and SIMT compute models adopted in CPU and GPU, the simplified dataflow model, e.g., 1-D SIMD vector unit or 2-D systolic array-based matrix unit, could be beneficial for maximizing the efficiency of hardware implementation in terms of performance, energy consumption, and silicon area. In addition, the convolutional layer, one of the most commonly used operations, can be accelerated through matrix multiplication based on the channel-wise unrolling or an im2col operation. The combination of the vector unit and matrix unit is flexible enough to support various commercial neural networks with minimal overhead.

For this reason, we adopt a compute model consisting of a 2-D matrix unit and a 1-D vector unit followed by a 1-D quantization unit. These units have a special functionality

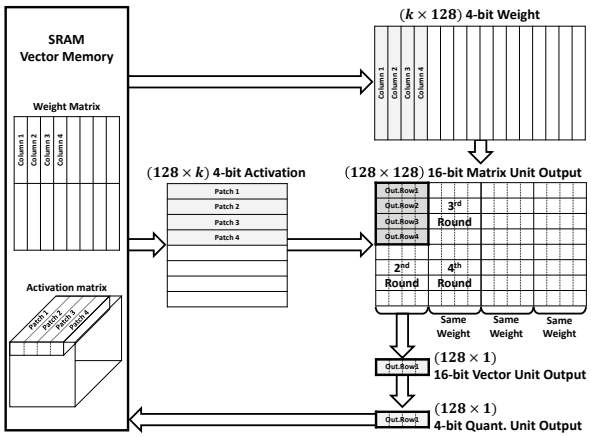

Fig. 3. Micro-level dataflow.

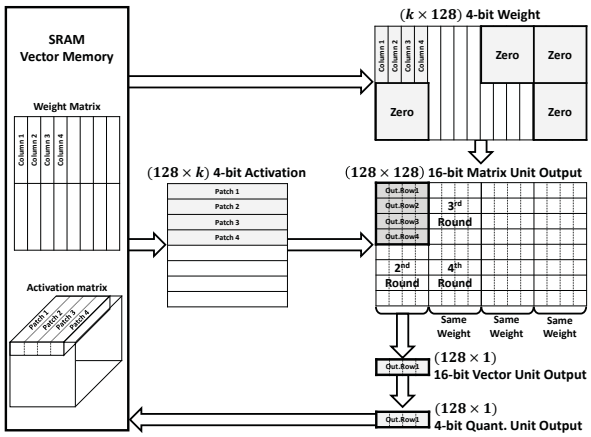

Fig. 4. Micro-level dataflow for sparse acceleration.

to support low-precision computation and zero-skipping. Near the computation units, a large SRAM vector memory is dedicated to provide sufficient bandwidth to the computation units and increase the data efficiency by reusing the data as much as possible. By judiciously designing the dataflow, all computation units can be fully utilized in parallel.

B. 2-D Systolic Array-based MAC Matrix Unit

Our neural network accelerator has a 2D systolic array-based multiplier-accumulator (MAC) matrix unit to achieve high throughput. Our MAC matrix unit consists of 128×128 MACs, and each MAC supports a 4-bit integer input, 22-bit integer accumulation, and 16-bit integer output. The systolic array operates matrix multiplication in an output stationary manner, which is more efficient than weight or activation stationary in terms of data movement and is more suitable for the implementation environment [3].

C. 1-D SIMD Vector Unit and Quantization Unit

For complex vector operations such as batch normalization, concatenation, and post-processing (including quantization), our neural network accelerator is equipped with a 1-D SIMD vector unit. The vector unit supports 16-bit fixed-point or 16-bit integer data precision and executes 128 element-wise vector operations in parallel. The vector unit is followed by a 1-D SIMD quantization unit in a pipeline manner. The quantization unit converts 16-bit output activation from the vector unit into 4-bit integer activation as an input for the next layer in 128 parallel element-wise quantization operations.

D. Memory Hierarchy

The memory hierarchy is designed to store all parameters and intermediate activations on a large on-chip SRAM so that off-chip DRAM access is not necessary during neural network inference. In our design, a large 26 MB SRAM memory plays the key role of the addressable main memory of the CISC instruction-based neural network accelerator. This large SRAM vector memory is utilized as an instruction buffer, input and output buffer for systolic array-based matrix unit, SIMD pipeline buffer, etc. When the neural network accelerator is fully working, the overall aggregate SRAM bandwidth is approximately 5 TB/s.

E. Activation Indexing Unit

To support a convolutional operation with a 2-D systolic array-based matrix unit, the architecture requires specialized hardware. In this hardware, we design the dataflow such that the data is stored in channel-first order and the spatial dimension of the convolutional kernel is processed sequentially. This dataflow enables us to perform the convolutional operation without unnecessary data duplication, but address index generation is required depending on the shape and type of convolutional operation. We design an activation indexing unit to support the proper addressing of the data layout for various types of convolutional operations. This unit is designed with sufficient extensibility to support various types of convolution operations, e.g., normal 3D convolution, depth-wise separable convolution, 1×1 convolution, group convolution, and transposed convolution with any kernel size, stride, padding, and dilation.

F. Details of Computation Pipeline

Fig. 3 shows the micro-level dataflow of the proposed neural network accelerator. The systolic array-based MAC matrix unit performs convolutional or matrix multiplication operations, with a matrix size of 128×128, in an output stationary manner. The weight and input activation data from the SRAM vector memory are streamed into a systolic array-based MAC matrix unit with a granularity of 512-bit (4-bit×128). Accumulated results are popped out in a row-by-row manner and are fed into the 1-D SIMD vector unit. The vector unit takes two operands: one operand comes from the SRAM vector memory, and the other operand comes from either the output buffer of the matrix unit or SRAM vector memory.

To support the activation indexing operation for the convolutional layer, we adopt address translation between stored activation in SRAM vector memory and fetched activation in the input buffer of the MAC matrix unit. This translation

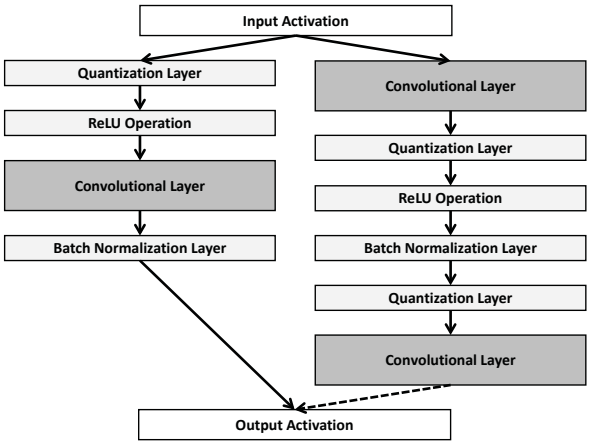

Fig. 5. Computation graph from PyTorch.

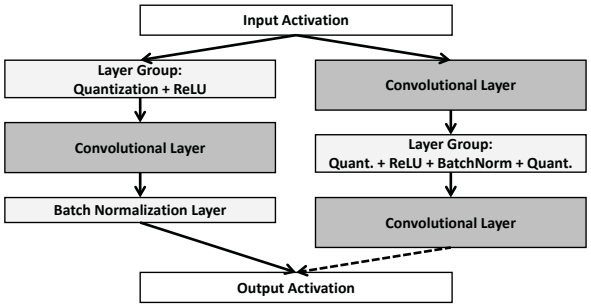

Fig. 6. Optimized computation graph with layer groups.

occurs with the support of hardware logic, an SRAM look-up table, and a software compiler. Owing to this address translation mechanism, our systolic array-based MAC matrix unit can easily support sparse convolutional and matrix multiplication operations with structured weight zero-skipping. Fig. 4 shows the micro-level dataflow of our neural network accelerator when sparse convolutional or sparse matrix multiplication operations are performed. If structured sparsity exists in the granularity of the 128×128 square block, our inference accelerator can skip the entire computation for the corresponding block, thereby significantly minimizing the computation cycle.

To fully utilize the computation units, the microarchitecture supports pipelined processing from the matrix unit, vector unit, and quantization unit. While instructions for the accelerator are executed in an in-order manner, concurrent instruction execution in matrix and vector units is allowed, with two separate datapaths, when there is no data dependency between the adjacent matrix and vector instructions. The instruction compiler examines the data dependency between adjacent instructions at compile time.

G. Instruction Set Architecture

To control the computation unit, 15 CISC instructions are designed for our neural network accelerator. Instructions are 32-bit wide with a 5-bit operation code, and the codes are generated by an instruction compiler for the given model and loaded in the SRAM instruction buffer before the inference starts. We designed two types of instructions: matrix unit and vector unit, for parallel execution regarding data dependency. The MAC matrix unit and vector unit have their own data movement instructions.

Instructions for the matrix unit perform three types of matrix operations: 1. Dense matrix multiplication 2. Dense convolution operation with dense weight and dense activation 3. Sparse matrix multiplication with sparse weight and dense activation. Instructions for each matrix operation are designed for 4-bit integer input and 16-bit fixed-point output. In addition, instructions for the MAC matrix unit have variable repeat fields in their operand, so each instruction with various operand has various average clock cycles per instruction (CPI).

Instructions for the vector unit perform vector operations such as element-wise addition, element-wise multiplication, ReLU activation, and optional quantization operation. As mentioned above, the vector unit takes two operands: one operand comes from the SRAM vector memory, and the other operand comes from either the output buffer of the matrix unit or SRAM vector memory. Instructions for the vector unit are designed for 16-bit fixed-point input and output. Clock cycles per instruction (CPI) for the vector unit is 1 when instructions for vector unit are executed in a pipelined manner.

IV. Software Stack Details

As shown in Fig. 1, our software stack includes a neural network model optimizer, instruction compiler, and API-based host-device interface. Our software stack is tightly integrated into the PyTorch [20] deep learning framework; thus, one can deploy the pre-trained model in the PyTorch framework to the proposed system seamlessly. After full-precision training, the network model is quantized to 4-bit precision on the PyTorch framework through the model optimizer. Then, the instruction compiler manipulates the neural network model into an optimized computation model form so that the hardware accelerator can exploit its parallelism. In addition, the instruction compiler translates the computation model into binary instructions for a neural network accelerator. Finally, the translated binary code is disposed to the accelerator through the host-device interface, and the interface controls data placement and communication. The details of each component are explained in the following subsections.

A. Neural Network Model Optimizer

Low-bit precision computation can offer better energy efficiency to hardware accelerators by allowing more computation units at a smaller energy consumption [9], [11]. To exploit this advantage on a neural network hardware accelerator, we need to quantize the deep neural network model into 4-bit integer without accuracy loss.

To realize the 4-bit model without accuracy loss, we design a quantization pipeline based on the learned step-size quantization [14], progressive quantization [18], and the knowledge

distillation approach [22]. Starting from the pre-trained full-precision model, the convolutional and matrix multiplication layers except the first and last layers are quantized into 4-bit through progressive quantization. Intermediate data in convolutional and matrix operations are quantized into a 16-bit fixed-point for accurate computation. After quantization, we re-trained the quantized neural network using a knowledge distillation approach to recover the original accuracy.

B. Instruction Compiler

The instruction compiler is one of the most important components of our deep neural network acceleration platform. Our in-house instruction compiler translates the quantized PyTorch model into binary instructions for the specialized accelerator.

Layer Group. The instruction compiler takes the quantized PyTorch model as an input. Fig. 5 shows the layer-wise computation graph of the ResNet-50 model from the PyTorch framework. Next, the instruction compiler performs graph optimization at the IR level by fusing multiple operations into one merged operation for efficient instruction generation. For example, a batch normalization layer and quantization layer can be fused into a single operation set, which can be processed by a vector unit through one instruction. Fig. 6 shows an example layer group composed of a batch normalization layer and quantization layer. This layer fusion is essential for increasing the utilization of the accelerator computation unit.

Scheduling and Translation. The instruction compiler schedules the optimized computation graph with layer groups to maximize the computation utilization of the neural network accelerator. After the layers are fused, the instruction compiler tries to find the computation pattern that can maximize the parallelism between the MAC matrix unit and vector unit in the neural network accelerator. Note that each component of the accelerator can be operated concurrently to maximize throughput. Thus, the binary code should consider this parallelism, which increases the importance of the proposed compiler. The scheduled computation pattern is translated into a series of binary instructions for the proposed accelerator.

C. Host Interface

We adopt OpenCL kernel call APIs to run our neural network accelerator on the FPGA. Two OpenCL kernels are defined to launch the accelerator operations. For data movement between the host memory and FPGA global memory, *clEnqueueReadBuffer* and *clEnqueueWriteBuffer* APIs are used.

Kernel-mode 0: Load All Parameters. The kernel call with mode 0 loads all the data from the FPGA global memory to the SRAM vector memory. The data include the compiled instructions, convolutional weights, scale and bias factors for batch normalization, and pre-calculated activation indexing offsets.

Kernel-mode 1: Load Input, Run Instructions, Store Output. The kernel call with mode 1 loads the input activation from the FPGA global memory to the SRAM vector memory.

Pseudocode 1 Processing flow of inference platform

```
// 1. Preparation Phase
// 1.1. Parameter and Instruction Movement
// (Host to FPGA Global DRAM)
ClEnqueueWriteBuffer(toFPGA, networkParameters);
ClEnqueueWriteBuffer(toFPGA, instructions);

// 1.2. Kernel mode 0:
// Parameter and Instruction Loading
// (FPGA Global DRAM to FPGA SRAM)
data_load(networkParameters, SRAM_vector_memory);
data_load(instructions, SRAM_instruction_buffer);


// 2. Repetition Phase
for (i = 0; i < testDataset.size(); i++) {
    // 2.1. Pre-processing on Host
    input_batch = pre_process(input_image[i]);

    // 2.2. Input data Movement
    // (Host to FPGA global DRAM)
    ClEnqueueWriteBuffer(toFPGA, input_batch);

    // 2.3. Kernel mode 1: Accelerator Run
    data_load(input_batch, SRAM_vector_memory);
    inference_output = inference(input_batch);
    data_store(inference_output, FPGA_DRAM);

    // 2.4. Output Data Movement
    // (FPGA global DRAM to Host)
    ClEnqueueReadBuffer(toHost, inference_output);

    // 2.5. Post-processing on Host
    final_results = post_process(inference_output);
}
```

Next, the instructions in the SRAM instruction buffer are executed sequentially. After running all the instructions in the instruction buffer, the output activations of neural network inference in the vector memory are copied from the SRAM vector memory to the FPGA global memory.

D. Processing Flow

Pseudocode 1 shows the processing flow of the proposed neural network inference acceleration platform. For seamless integration, simple OpenCL APIs are designed for our platform. The processing flow of our acceleration platform consists of the following two phases.

In the preparation phase, all parameters and instructions are loaded into the SRAM vector memory of the accelerator. First, OpenCL *clEnqueueWriteBuffer* API copies these parameters and instructions from the host memory to the FPGA global memory. Next, the OpenCL kernel call, with kernel-mode 0, loads parameters and instructions from the FPGA global memory to the SRAM vector memory of the neural network accelerator.

In the repetition phase, the following five steps were executed repeatedly to perform neural network inference. Initially, the host processor pre-processes the input image to generate input activation for the neural network accelerator. Next, pre-processed input activation is copied from the host memory to the FPGA global memory via the OpenCL *clEnqueueWrite-*

TABLE I
ENVIRONMENT SETUP

Host		FPGA	
CPU	Intel i9-10900K	**Board**	Xilinx Alveo U250
Memory	DDR4-3200 128 GB	**Tools**	Vivado 2020.1, Vitis 2020.1
Storage	SSD 1 TB	**XRT**	xrt_202010.2.6.655_7.4.1708-x86_64
OS	CentOS 7.4.1708	**XDMA**	xilinx-u250-xdma-201830.2-2580015.x86_64
Kernel	3.10.0-693.el7.x86_64	**Shell**	xilinx_u250_xdma_201830_2

Buffer API. As a kernel run stage, OpenCL kernel call with kernel-mode 1 loads input activation from the FPGA global memory to the SRAM vector memory, executes all the instructions in the instruction buffer, and stores the output activation from the SRAM vector memory of the neural network accelerator to the FPGA global memory. Subsequently, output activation is copied from the FPGA global memory to the host memory via OpenCL *clEnqueueReadBuffer* API. Finally, the host processor post-processes the output activation to obtain inference results, such as classification results and performance metrics.

V. EXPERIMENTS

A. Experiment Setup

In order to show the superiority of the proposed system, we validate our acceleration platform by demonstrating it on real hardware. Table I presents the detailed experimental setup of our implementation. We use Xilinx Vivado framework [23] for RTL design, software simulation, synthesis, place and route, bit-stream generation, and Xilinx Vitis Unified Software Platform [24] for host-device integration and memory interface design. We integrate our framework with the PyTorch deep learning framework for the quantization and optimization of a deep neural network model [20], and deploy the accelerator design on the Xilinx Alveo U250 FPGA board, which is equipped with 54 MB, 38 TB/s SRAM, and 64 GB, 77 GB/s DDR4 DRAM, 11,288 DSP blocks, and 1,341,000 LUTs for logic slices [21].

We select the acceleration target as ResNet-50 because it is used as a reference model on the MLPerf benchmark [25][1]. We quantized ResNet-50 neural network model into 4-bit integer precision without accuracy loss through our optimization pipeline, and the model is deployed to the accelerator using the designed software stack without human intervention.

B. Results

According to our observations, the proposed accelerator shows 317 GOPS inference speed and 51.96 GOPS/W energy efficiency, with a batch size 1 at 108 MHz, when we deploy the 4-bit quantized ResNet-50 model. Note that these qualitative values are measured on a real FPGA board. According to the scoreboard of the MLPerf [25] v0.7 inference benchmark and industrial FPGA solution [31], the proposed system performed comparable inference speeds on a similar scale and better energy efficiency.

[1]Since our neural network accelerator supports various types of convolutional and matrix multiplication operations, our inference platform can support not only the ResNet-50 model, but also many other 4-bit quantized deep neural network models such as ResNet [26], EfficientNet [27], VGG [28], GoogleNet [29], and MobileNet [30].

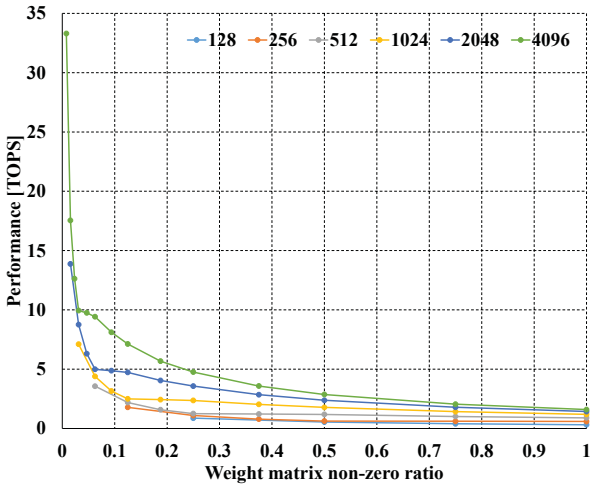

Fig. 7. Performance of the accelerator at various sparsity and matrix size.

Fig. 7 shows the effective throughput of our accelerator implemented on the FPGA when it runs sparse matrix multiplication. The sparsity of the matrix is configured as a ratio of random 128×128 square zeros while computing the $512\times k$ activation matrix and $k\times512$ weight matrix. As the zero ratios of the weight matrix increase, the throughput, for all cases, increases regardless of the k size. However, when the k size is sufficiently large, the throughput increases linearly with zero ratios of weight matrix. This is because the latency of storing the output of matrix multiplication can be completely hidden behind the concurrent matrix operation, only when the input matrix dimension k is sufficiently large.

VI. CONCLUSION

In this study, we proposed a complete system for an inference acceleration platform that enables the seamless deployment of the trained model from PyTorch to the FPGA accelerator. The proposed system includes a novel energy-efficient microarchitecture and its RTL implementation, and the software stack, including a model optimizer, instruction compiler, and API-based host–device interface. We validated the superiority of the proposed system by demonstrating on an actual FPGA. Extensive experiments show that our accelerator achieved comparable inference speeds on a similar scale with better energy efficiency than industrial FPGA solutions.

ACKNOWLEDGMENTS

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) [NO.2021-0-01343, Artificial Intelligence Graduate School Program (Seoul National University)].

REFERENCES

[1] "Nvidia a100 tensor core gpu," https://www.nvidia.com/en-us/data-center/a100/, accessed: 2021-07-29.

[2] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[3] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 367–379, 2016.

[4] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 269–284, 2014.

[5] T. Luo, S. Liu, L. Li, Y. Wang, S. Zhang, T. Chen, Z. Xu, O. Temam, and Y. Chen, "Dadiannao: A neural network supercomputer," *IEEE Transactions on Computers*, vol. 66, no. 1, pp. 73–88, 2016.

[6] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 1–13, 2016.

[7] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[8] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 27–40, 2017.

[9] D. Kim, J. Ahn, and S. Yoo, "Zena: Zero-aware neural network accelerator," *IEEE Design & Test*, vol. 35, no. 1, pp. 39–46, 2017.

[10] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 764–775.

[11] E. Park, D. Kim, and S. Yoo, "Energy-efficient neural network accelerator based on outlier-aware low-precision computation," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 688–698.

[12] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan, "Pact: Parameterized clipping activation for quantized neural networks," *arXiv preprint arXiv:1805.06085*, 2018.

[13] S. Jung, C. Son, S. Lee, J. Son, J.-J. Han, Y. Kwak, S. J. Hwang, and C. Choi, "Learning to quantize deep networks by optimizing quantization intervals with task loss," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 4350–4359.

[14] S. K. Esser, J. L. McKinstry, D. Bablani, R. Appuswamy, and D. S. Modha, "Learned step size quantization," *arXiv preprint arXiv:1902.08153*, 2019.

[15] S. Migacz, "NVIDIA 8-bit inference width TensorRT," *GPU Technology Conference*, 2017.

[16] E. Park, J. Ahn, and S. Yoo, "Weighted-entropy-based quantization for deep neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 5456–5464.

[17] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv preprint arXiv:1606.06160*, 2016.

[18] E. Park and S. Yoo, "Profit: A novel training method for sub-4-bit mobilenet models," in *European Conference on Computer Vision*. Springer, 2020, pp. 430–446.

[19] S. Anwar, K. Hwang, and W. Sung, "Structured pruning of deep convolutional neural networks," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 13, no. 3, pp. 1–18, 2017.

[20] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, pp. 8026–8037, 2019.

[21] "Xilinx alveo u250 data center accelerator card," https://www.xilinx.com/products/boards-and-kits/alveo/u250.html, accessed: 2021-07-28.

[22] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[23] "Xilinx vivado design suite," https://www.xilinx.com/products/design-tools/vivado.html, accessed: 2021-07-29.

[24] "Xilinx vitis unified software platform," https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html, accessed: 2021-07-29.

[25] P. Mattson, V. J. Reddi, C. Cheng, C. Coleman, G. Diamos, D. Kanter, P. Micikevicius, D. Patterson, G. Schmuelling, H. Tang *et al.*, "Mlperf: An industry standard benchmark suite for machine learning performance," *IEEE Micro*, vol. 40, no. 2, pp. 8–16, 2020.

[26] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[27] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International Conference on Machine Learning*. PMLR, 2019, pp. 6105–6114.

[28] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[29] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[30] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[31] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi *et al.*, "A configurable cloud-scale dnn processor for real-time ai," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 1–14.