2021 14th IEEE International Conference on Industry Applications (INDUSCON) | 978-1-6654-4118-6/21/$31.00 ©2021 IEEE | DOI: 10.1109/INDUSCON51756.2021.9529474

Real-Time Downhole Geosteering Data Processing Using Deep Neural Networks On FPGA

Qiyu Wan
ECOMS Lab, ECE Department
University of Houston
Houston, USA
qwan@uh.edu

Yuchen Jin
ECE Department
University of Houston
Houston, USA
yjin4@uh.edu

Xuqing Wu
College of Technology
University of Houston
Houston, USA
xwu8@central.uh.edu

Jiefu Chen
ECE department
University of Houston
Houston, USA
jchen82@central.uh.edu

Xin Fu
ECOMS Lab, ECE Department
University of Houston
Houston, USA
xfu8@central.uh.edu

***Abstract*—The success of machine learning has spread the deployment of Deep neural Networks (DNNs) in numerous industrial applications. As an essential technique in today's oilfield industry, geosteering requires performing DNN inference on the hardware devices that operates under the severe downhole environments. However, it can produce massive power dissipation and cause long delays to execute the computation-intensive DNN inference on the current hardware platforms, e.g., CPU and GPU. In this paper, we propose an FPGA-based hardware design to efficiently conduct the DNN inference for geosteering tasks in downhole environments. At first, a comprehensive analysis is presented to choose the optimal computation mapping method for the target DNN model. A detailed description of the customized hardware implementation is then proposed to accomplish a complete DNN inference on the FPGA board. The experimental results shows that the proposed design achieves 7×(1.4×) improvement on performance and 82× (1.3×) reduction on power consumption compared with CPU(GPU).**

***Index Terms*—geosteering, deep neural networks, FPGA**

I. Introduction

Geosteering is an essential drilling service of adjusting the well trajectory on the fly to reach the geological targets or to keep the wellbore within desired formation. An efficient and accurate inversion algorithm is the brain that enables geosteering by using logging-while-drilling (LWD) sensing data to reconstruct the formation structure near the borehole. Because solving geosteering inverse problems is computationally expensive, and due to the limitation of downhole computing resources, existing

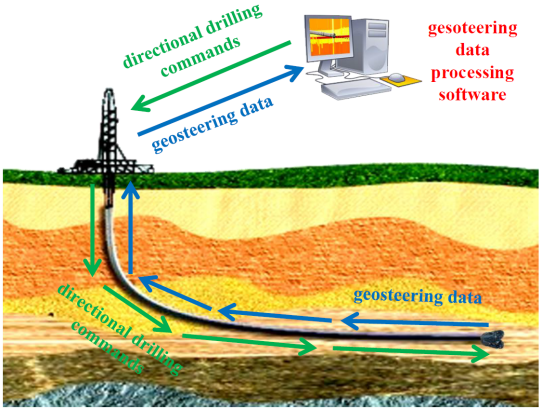

Fig. 1. Schematic of current oilfield geosteering service: well logging data are taken downhole and sent to surface, underground formation is inferred based on processing of well logging data, and then the drilling commands are made and sent back to downhole drilling tool to guide the borehole trajectory.

geosteering inversion programs are performed on the surface using the borehole sensed data transmitted from downhole to surface (see Fig. 1). Since the maximum data rate for mud pulse telemetry, a dominant transmission method used by the industry, is at 20 bits per second, data transmission from the downhole to the surface incurs serious delays. This bottleneck often leads to wrong drilling decision-making attributed to compromised formation evaluation because only a small portion of logging data can be sent to the surface to facilitate the geosteering inversion computation. In this paper we propose a Field Programmable Gate Array (FPGA) implemented deep neural network (DNN) to enable real-time downhole processing of all available geosteering data without any delay.

To accomplish the geosteering task, conventional optimization-based subsurface inversion process incurs large computation delay and is sensitive to initial values [1]. As machine learning has emerged as a panacea for complex optimization problems, recent efforts [2] has shown that by incorporating a DNN within the inversion model, the whole inverse process could be accomplished much faster by performing a DNN inference. However, this solution poses great challenges to the real-world implementation since DNN inference can still be time-consuming and energy-inefficient when executed on the traditional hardware platforms, i.e., CPU and GPU. [3], which could cause severe hazards to the downhole devices.

To tackle with these problems, we leverage the FPGA hardware platform to conduct the DNN inference efficiently. Compared with the CPU and GPU platforms, FPGA embraces important advantages when executing the DNNs in the extreme downhole environments. Firstly, an FPGA board can be designed to be a dedicated DNN accelerator that targets on DNN inference only, thus it will be much more energy-efficient and time-saving if running DNN inference on FPGA compared with general purpose processors like CPU/GPU. Secondly, modern FPGA boards can provide advanced communication interfaces such as SFC/SFC+ cages that can connect with fiber-optic cables, which can speed up the data transmission between the downhole devices and the drilling station. Lastly, the lifetime of the deployed hardware device should be considered in the high-temperature and high-vibration environment. For example, The maximum allowed operating temperature of FPGA boards can reach 150 °C for downhole applications while the thermal of NVIDIA TX2 GPU is limited below 80 °C. Moreover, among the CPU, GPU and FPGA chips that have the similar computing capability, the unit cost of an FPGA board is usually the smallest. Therefore, FPGA is the best hardware platform to conduct the DNN-based subsurface inversion task in the downhole environments.

The aforementioned benefits of FPGA motivates us to implement the target DNN inference on the real FPGA board. Prior studies [4]–[9] proposed different optimizations for DNN inference on FPGAs. However, none of them is perfectly suitable to our target DNN for the subsurface inversion task. Firstly, these works are oblivious of the special layer structures in our target DNN, which can lead to inefficient computations. Secondly, the hardware implementation of instance normalization layer (an important layer in the target DNN) has not been proposed yet. In this paper, we address this first design challenge by adopting an efficient computation mapping strategy to execute the target DNN inference through rationally analyzing the exclusive structures of the target network. Furthermore, we propose a light-weighted hardware implementation that realizes the full function of the instance normalization layer, bringing only negligible overhead to the overall system.

The rest of the paper is organized as follows. Section II introduces the detailed information of the target DNN implemented on the FPGA board and reviews the basic functions of different DNN layers. In Section III, two methods are proposed to implement the target DNN inference efficiently. In Section IV, the effectiveness of the proposed design is demonstrated by the experimental results.

II. DNN for Geosteering

The DNN for the geosteering task (denoted as Geo-DNN) is a convolutional neural network which is developed by [2]. The network contains 9 convolutional layers, 5 pooling layers and 1 fully-connected layer. Apart from the aforementioned three types of layers in common convolutional neural networks, the network has one instance normal-

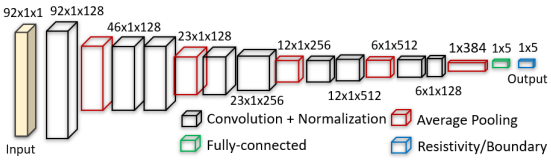

Fig. 2. Geo-DNN Structure.

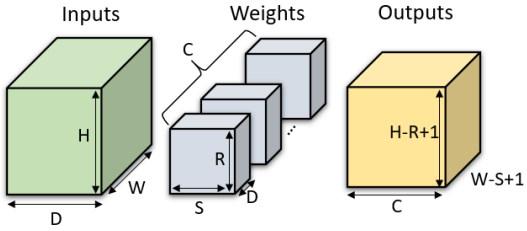

Fig. 3. Computations of a convolutional layer. The 3-D outputs are calculated by convolving the 3-D inputs and 4-D weights.

ization layer concatenated after each convolutional layer and a resistivity/boundary layer attached at the end of the fully-connected layer. The overall structure of the target DNN is depicted in Fig. 2. In this section, we review the functionality of different types of layers involved in the target network.

Convolutional layer

A convolutional layer aims to capture the features/representations of the input data. [10] Fig. 3 depicts three main components of a convolutional layer. The inputs of a convolutional layer is known as input feature maps (the green cube shown in the figure) that have the size of W (width), H (height) and D (depth). The layer also contains a 4-D weight matrix that is divided into a group of 3-D filters (grey cubes). Each filter is composed of D kernels while each kernel has the size of $R \times S$. Each filter performs 3-D convolution with the input feature map individually and generate one output feature map. Hence, the outputs of the convolutional layer are totally C output feature maps where C is the number of filters.

Pooling layer A pooling layer usually appears between two consecutive convolutional layers. It reduces the size of input feature maps, i.e., width and height, and extract more important information for the next layer. The pooling function can be max or average. In our target Geo-DNN, all the pooling layers are average pooling layers. The averaging function averages all the values of a certain sliding window and generate one output value. In Geo-DNN, the size of sliding window is 1×2 in all pooling layers.

Fully-connected layer A fully-connected layer is commonly attached at the tail of the DNN and used to generate the final outputs. It can be interpreted as a special case of a convolutional layer by flattening the input feature maps and weights into a 2-D matrix.

Instance normalization layer Normalization techniques are widely used in many popular networks to improve the training accuracy. [11], [12]. Among the branches of emerging normalization techniques, the Geo-DNN adopts the *instance normalization* technique in all normalization layers. The unique feature of instance normalization is that it normalizes across each channel in each training example. Given the neuron index j, the channel index c and the index of training example i, the detailed computations can be described in the following equations:

$$\bar{x}_{ic} = \frac{1}{L}\sum_{j=1}^{L} x_{ijc} \tag{1}$$

where x_{ijc} represents the j-th neuron in the c-th channel of the i-th training example and L is the number of neurons in each channel. The above equation calculates the mean value of each channel and then the standard deviation (std) can be obtained by:

$$\sigma_{ic} = \sqrt{\frac{1}{L}\sum_{j=1}^{L}(x_{ijc} - \bar{x}_c)^2} \tag{2}$$

The normalized values are calculated as:

$$y_{ijc} = \gamma_c \left(\frac{x_{ijc} - \bar{x}_{ic}}{\sigma_{ic} + \varepsilon} \right) + \beta_c \tag{3}$$

where γ_c and β_c are the parameters that vary across different channels and ε is a constant value. Finally, the normalized values are fed into a PReLU function and the outputs y' can be expressed as:

$$y'_{ijc} = \begin{cases} y_{ijc}, & y_{ijc} >= 0 \\ \alpha_c y_{ijc}, & y_{ijc} < 0 \end{cases} \tag{4}$$

where α_c are also the parameters that vary across different channels.

Resistivity/boundary layer The resistivity/boundary function takes the output of the fully-connected layer as input and performs non-linear operations (e.g. Sigmoid function) on it. Given constant values a_1, b_1, a_2, b_2, input data x, the function of this layer can be described as:

$$y = a_2 + b_2 \cdot Sigmoid(a_1 + b_1 x) \quad (5)$$

III. Methodology

To improve both performance and energy-efficiency of the Geo-DNN inference task in the downhole environment, we propose two major design methodologies that embrace the advantages brought by flexible FPGA platform. Firstly, in Section III-A, for the convolutional layers, we discuss the existing computation mapping strategies and reveal the most efficient one for the target Geo-DNN by rationally considering the structure of each layer. Secondly, in section III-B, we give a comprehensive description of our complete implementation for the whole inference process. Especially for the instance normalization layer, a fully-customized hardware design is proposed to efficiently support the complex computation flow.

A. Layer-wise Analysis of Computation Mapping

Due to the computation independence between different output neurons, the operations in a convolutional layer can be processed in parallel. Prior DNN accelerators leveraged this opportunity and proposed various parallelism schemes to accelerate the DNN inference [7], [13]–[15]. The key difference between these parallelism schemes is that they map the data (e.g., neurons/weights) to the computation unit (also known as processing element (PE)) along the different dimensions. Fig. 4 illustrates three types of popular computation mapping methods in existing DNN accelerators. In all these mapping methods, either neurons or weights are firstly mapped to the PEs and then being processed simultaneously.

W&H-mapping W&H-mapping strategy [14] is conducted along the width and height dimension. In this strategy, the output neurons, which are spatially distributed on an output feature map, e.g. N_1, N_2, N_3, N_4, are mapped to PEs and being calculated in parallel.

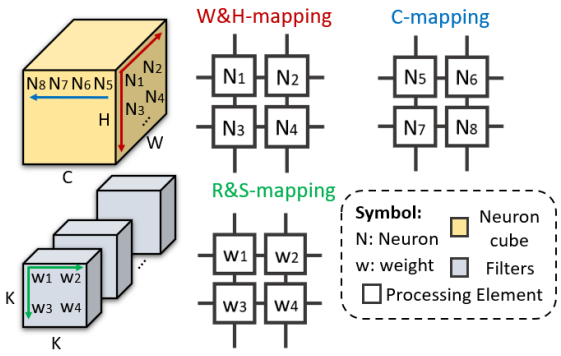

Fig. 4. Illustration of three types of DNN computation mapping strategies: W&H-mapping that maps along red arrows, C-mapping that maps along blue arrow and R&S mapping that maps along green arrows.

C-mapping C-mapping strategy [13], [15] is performed along the channel (include both input channel (referred as depth in Fig.3) and output channel) dimension. In this strategy, the output neurons, which locates at the same position in the feature map but from different channels, e.g. N_5, N_6, N_7, N_8, are mapped to PEs and being calculated in parallel.

R&S-mapping R&S-mapping strategy [7] targets on the parallel computation of weights. It is conducted along the width and height dimension of each kernel. In this strategy, the weights that are spatially distributed on a kernel, e.g. w_1, w_2, w_3, w_4, are mapped to PEs and multiplies with their corresponding inputs in parallel.

We observed that, in some cases no neurons/weights will be allocated to some of the PEs due to the mismatch between the size of the PE array and the size of neurons/weights. In some cases the PEs could become idle for a long time and waste power and resources. For example, assume the size of the PE array is 2×2 and an output feature map has the width of 1 and the height of 4, the W&H mapping can only map two neurons to the PEs at the same time. Therefore, arbitrarily choosing a computation mapping strategy for our network may result in lower PE utilization, which could further lead to degradation in terms of performance and energy-efficiency. This motivates us to find the best one among these three mapping methods. To determine the most suitable computation mapping strategy for our network, we rationally calculate the PE utilization level by analyzing the structure of

each layer in our target network. In this case, we assume the size of the PE array to be 8×8, which is a common configuration in the existing DNN accelerators.

TABLE I
PE UTILIZATION OF DIFFERENT COMPUTATION MAPPING FOR GEO-DNN.

Layer	Output			Kernel		PE utilization		
L	H	W	C	R	S	W&H	C	R&S
conv1-1	92	1	128	3	1	12.5%	12.5 %	4.7%
conv2-1	46	1	128	3	1	12.5%	100 %	4.7%
conv3-1	23	1	128	3	1	12.5%	100 %	4.7%
conv4-1	12	1	256	3	1	12.5%	100 %	4.7%
conv5-1	6	1	512	3	1	9.4%	100 %	4.7%
fc	1	1	384	384	1	1.6%	100 %	12.5%

Table I shows the PE utilization of three computation mapping strategies when executing different layers.

- The PE utilization of the W&H-mapping is related to the width (W) and height (H) of the output feature maps. Since the width of the output feature maps is always 1, only 1/8 (12.5%) of PEs can be utilized in W&H-mapping and even fewer PEs are utilized when executing the last two layers.
- The PE utilization of the C-mapping depends on the number of channels (C) of input/output feature maps. Since C is always larger than the PE size except for the first layer, the PE utilization can reach 100% for most layers.
- The PE utilization of the R&S-mapping relates to the size of the kernels (R and S). As the kernel size of Geo-DNN is always 3×1 for all the convolutional layers, only 3/64 (4.7%) of PEs can be utilized in R&S-mapping. This number slightly increases when it is executing the fully-connected layer.

According to the above analysis, we conclude that C-mapping achieves the best PE utilization level for our target network, which can potentially boost the performance and improve the energy-efficiency. We thus choose C-mapping strategy as the basic computation flow to implement our hardware design.

B. Hardware Implementation

1) Overall Architecture: As shown in Fig. 5, the overall system is comprised of a global buffer, a PE array, a resistivity/boundary module, a instance normalization module, a non-linear unit and a

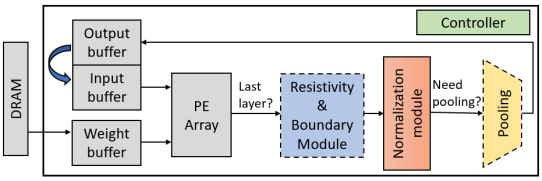

Fig. 5. Architecture overview.

controller. The resistivity/boundary module and the instance normalization module are customized hardware design to support the special functions/layers, i.e., resistivity/boundary operations and instance normalization layers in our target network.

Structure of the PE array The function of a PE array is to efficiently perform the convolution operations and matrix multiplications for convolutional layers and fully-connected layers, respectively. This function is achieved as the PE array is capable of conducting large amount of multiply–accumulate operations (MACs) in parallel. The number of PEs can be customized based on the requirement of computation delay. For simplicity we introduce our design assuming that 8 PEs a re deployed in the PE array. For each PE, it is responsible for producing the output neurons located at one output channel. There are three parts resides in a PE: a multiplier array, a adder tree and a accumulator. The multiplier array has 4 identical 32-bit floating point multipliers working in parallel. At each cycle, each of the four multipliers receives an input activation and a weight from a certain input channel. The generated products are collected by the adder tree and summed up as a partial sum. The accumulator accumulates the partial sums until it gets the final value of the output neuron. Both the adder tree and the accumulator use the same 32-bit floating point arithmetic as the multipliers.

Global buffer The global buffer is split into three sub-buffer: weight buffer, input buffer and output buffer. The width of the weight sub-buffers is fixed as $\#\_of\_multipliers\_per\_pe \times \#\_of\_PEs \times 32$-bits so that it can provide sufficient data to all the PEs at each cycle. The depths of the weight sub-buffers are configured to accommodate all the weights of the conv5-2 layer, which contains the largest volume of weights. The weights of next layer are fetched from the DRAM when the exe-

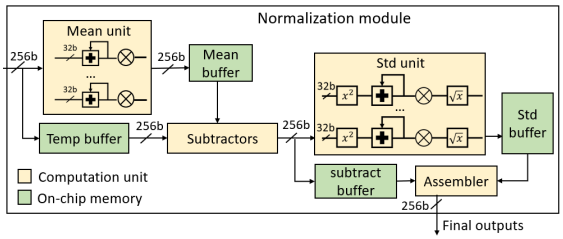

Fig. 6. High-level Architecture design for instance normalization layer.

cution of the current layer is finished. To reduce the large power consumption caused by frequent off-chip memory (DRAM) access, we allocate a large amount of BRAM resources as input/output buffer to store the intermediate feature maps so that all the intermediate data accesses are kept on-chip. Since the input activations are shared across all 8 PEs, the width of input/output buffer is set to be $\#\_of\_multipliers\_per\_pe \times 32$-bits. The depth is set to be large enough so that it can hold the largest intermediate feature map in our target network. When finishing executing one layer, the input buffer and output buffer swap their role with each other to avoid unnecessary data movement.

Non-linear unit The non-linear unit operates to implement the average pooling layers and the ReLU activation function. It consists of 8 pooling unit and 1 activation unit. Each pooling unit is composed of an accumulator and a multiplier and the activation unit is implemented using simple logic gate.

Controller The controller is implemented as a Finite State Machine (FSM) that has three stages: convolution, pooling and normalization. The main function of the controller is to organize the entire execution flow of the DNN inference. Specifically, it sends control signals (e.g., reset, enable...) to the PEs and dynamically generate the correct memory address for BRAM read/write.

2) Design of Instance Normalization Module: Fig. 6 illustrates the high-level design of the instance normalization module. The workflow of this module can be divided into three phases. In the first phase, the mean unit, which consists of 8 accumulators and 8 multipliers, receives the a 256-bit output data block from the PE array. The data block is split into 8 32-bit output values and allocated to different accumulators and multipliers. The mean unit accumulates and averages the output data and calculate the mean values of 8 output channels, which implements the function of Equation 1. The mean values are then temporarily stored in the mean buffer with 256 bit per entry. Once the mean values of all output channels are obtained, the module goes into second phase and starts calculating the subtract values of each neuron. The subtract values, on one path, are fed into std unit, i.e., an array of square modules, accumulators, multipliers and square root modules, to obtain the standard deviation (std), which corresponds to Equation 2. On another path, these subtract values are temporarily stored in the subtract buffer for later reuse. In the final phase, the subtract values and standard deviations are fetched from their corresponding buffers and fed into an assembler. The assembler comprises of basic logic gates, multipliers and adders to perform the operations in Equation 3 and Equation 4.

3) Design of Resistivity/Boundary Module: The unique operations required by resistivity/boundary function is the sigmoid function that demands an exponential operation on the input data x:

$$S(x) = \frac{e^x}{e^x + 1} \tag{6}$$

To achieve the above function, we adopt the existing Xilinx IP core [16] to implement the exponential function and use the a few more multipliers and adders to accomplish the whole function.

IV. Experimental Results

A. Baselines and design extension

To demonstrate the effectiveness of the proposed methodology, we compare our FPGA design with two baseline implementation on CPU and GPU. We choose an Intel Core i5-7400 processor for the CPU implementation and a NVIDIA Jetson TX2 board, a state-of-the-art machine learning inference engine for the GPU implementation. The principle of such comparison is the same as that in the existing NN acceleration studies, such as [4], [7], [8]. To demonstrate the flexibility and scalability of our methodology, we further extend the design to four classes which have different hardware configurations, i.e., different number of PEs (from 8 to 64). The performance and energy consumption of all the implementations are compared in subsection IV-D.

B. Network training and datasets

The Geo-DNN is trained offline and reaches convergence in 5 epochs. The training and validation datasets contain 2 million sample data in total and we use 10% of the total data for validation. The input features of the Geo-DNN are the sample data measured by the working station. Each input is a 1×92 vector. The output of the Geo-DNN is the result of the inverse technique, which is a simple 1×5 vector. The batch size is set to be 64 during training.

C. Experiment setup

The overall system design is implemented on the Xilinx Virtex7 FPGA VC709 Evaluation Board. All the hardware modules are implemented with Verilog language and we use Vivado 2018.3 SDK to simulate and synthesize the hardware design. The basic function of all the hardware modules are validated by comparing the outputs of Geo-DNN between our hardware implementation and the software implementation. The software version of Geo-DNN is constructed in TensorFlow framework. The hardware system operates constantly at a clock frequency of 100MHz. The performance and energy consumption of our hardware design are measured during processing one input data with the size of 92×1. The execution time is obtained from the simulation results and the energy consumption is calculated by Xilinx Power Estimator. [17]

D. Results

Hardware utilization For the design that the PE array has the size of 64, we report the hardware resource utilization in Table II. As can be seen, our implementation well exploits the hardware resources on board, especially for BRAM since all the data are stored in the on-chip memory to reduce the DRAM access energy.

TABLE II
HARDWARE RESOURCE UTILIZATION.

Resource type	Number of usage	Total resource on board
Look-up table	271410	433000
Flip-flop	467682	866000
BRAM	1280	1470

Performance and energy consumption As shown in Table III, our proposed FPGA implementation

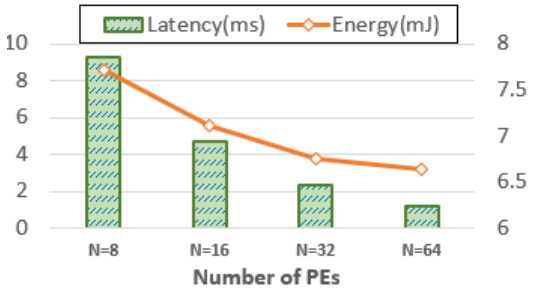

Fig. 7. Performance and Energy consumption comparison of four types of proposed implementations.

with 64 PEs achieves significant performance improvement ($7\times$) and energy reduction ($82\times$) compared with the CPU implementation. The proposed design also outperforms the GPU implementation by $1.4\times$ and $1.3\times$ in terms of performance and energy, respectively. Fig. 7 further exhibits the performance trend when we scale up the size of PE array. It can be observed that the performance is improved by $7.7\times$ if the number of PEs increases from 8 to 64. It also shows that the power dissipation for Geo-DNN inference decreases as more PEs are added on board. The main reason is that the reduced latency constantly has a larger impact on energy consumption than the increased on-chip power introduced by more PEs. However, the increased on-chip power may introduce higher risks of cooling system hazards. Such problem can be alleviated by adjusting the number of PEs and re-synthesizing according to the dynamic downhole environment, which is enabled by the reconfigurable FPGA platform.

TABLE III
PERFORMANCE AND POWER COMPARISON ON GEO-DNN.

Hardware platform	Latency(ms)	Energy(J)
Intel Core i5-7400	8.42	0.54730
NVIDIA Jetson TX2	1.64	0.00849
Proposed FPGA implementation	1.20	0.00664

Overhead analysis The normalization module contains several on-chip buffers to store the intermediate results, e.g., subtract buffer, to avoid recomputations. Such design reduces the energy consumption of computations but increases the resource usage of on-chip BRAMs. Fig. 8 shows that the resource overhead brought by extra on-chip buffers in the normalization layer is limited under 8%.

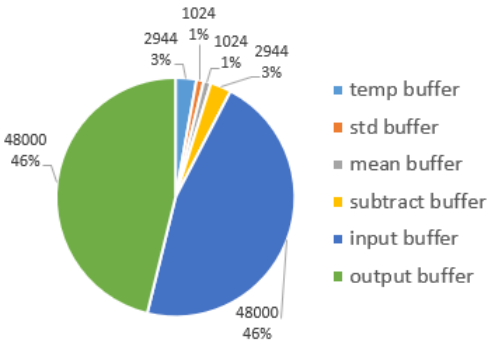

Fig. 8. The breakdown of the intermediate data buffers. The numbers represent the memory size (Byte) of different types of buffers.

V. Conclusion

This paper proposes an FPGA-based hardware design to accelerate the Geo-DNN inference and improve the energy-efficiency of the hardware devices that operating in the downhole environment. A layer-wise analysis of computation mapping strategy is presented to pick the best mapping scheme for the target network. Moreover, a complete hardware implementation is illustrated and especially a fully customized module design for the instance normalization layer is proposed. The proposed design achieves 7× (1.4×) improvement on performance and 82× (1.3×) reduction on power consumption compared with CPU(GPU).

References

[1] W. Lei, F. Yiren, Y. Chao, W. Zhenguan, D. Shaogui, and Z. Weina, "Selection criteria and feasibility of the inversion model for azimuthal electromagnetic logging while drilling (lwd)," *Petroleum Exploration and Development*, vol. 45, no. 5, pp. 974–982, 2018.

[2] Y. Jin, X. Wu, J. Chen, and Y. Huang, "A physics-driven deep learning network for subsurface inversion," in *2019 United States National Committee of URSI National Radio Science Meeting (USNC-URSI NRSM)*. IEEE, 2019, pp. 1–2.

[3] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "A survey of fpga-based neural network accelerator," *arXiv preprint arXiv:1712.08934*, 2017.

[4] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*, 2015, pp. 161–170.

[5] 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. 1. IEEE, 2015, pp. 829–832.

[6] G. Feng, Z. Hu, S. Chen, and F. Wu, "Energy-efficient and high-throughput fpga-based accelerator for convolutional neural networks," in *2016 13th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*. IEEE, 2016, pp. 624–626.

[7] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "Cnp: An fpga-based processor for convolutional networks," in *2009 International Conference on Field Programmable Logic and Applications*. IEEE, 2009, pp. 32–37.

[8] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," *Microsoft Research Whitepaper*, vol. 2, no. 11, pp. 1–4, 2015.

[9] M. Peemen, A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*. IEEE, 2013, pp. 13–19.

[10] S. Albawi, T. A. Mohammed, and S. Al-Zawi, "Understanding of a convolutional neural network," in *2017 International Conference on Engineering and Technology (ICET)*. Ieee, 2017, pp. 1–6.

[11] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*. PMLR, 2015, pp. 448–456.

[12] D. Ulyanov, A. Vedaldi, and V. Lempitsky, "Instance normalization: The missing ingredient for fast stylization," *arXiv preprint arXiv:1607.08022*, 2016.

[13] 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.

[14] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 92–104.

[15] Q. Wan and X. Fu, "Fast-bcnn: Massive neuron skipping in bayesian convolutional neural networks," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 229–240.

[16] "Xilinx IP core," https://www.xilinx.com/products/intellectual-property/floating_pt.html, [Online; accessed 19-Jan-2021].

[17] "Xilinx Power Estimator," https://www.xilinx.com/products/technology/power/xpe.html, [Online; accessed 19-Jan-2021].