Design and Scheduling of Convolutional Neural Network Accelerators for Cloud FPGAs: Post-print

Authors: Cai Ruichu, Yu Yang, Chunrong Zhong, Lu Ye, Chen Yao

Date: 2018-11-29T00:00:00+00:00

Abstract

The high computational complexity of convolutional neural networks hinders their widespread deployment in real-time and low-power applications. Existing software implementations fail to satisfy the stringent demands of convolutional neural networks regarding computational performance and power consumption. Traditional FPGA-oriented convolutional neural network construction approaches suffer from complex workflows, lengthy development cycles, and limited optimization space. To address these challenges, this paper proposes a convolutional neural network accelerator design and its scheduling mechanism tailored for cloud FPGAs, based on the characteristics of convolutional neural network computation patterns. By leveraging HLS-based design methodologies, introducing loop tiling parameters, and reordering convolutional layer loops, the network is constructed modularly with parameter expansion to further optimize accelerator processing. Through analysis of system task and resource characteristics, a scheduling scheme is derived and optimized from both control flow and data flow perspectives. In comparison with existing works, the proposed design provides a solution that simultaneously offers flexibility, low energy consumption, high energy efficiency, and high performance, while also exploring an efficient and general scheduling scheme for accelerators. Experimental results demonstrate that compared with CPU implementation, the design achieves 8.48× speedup for AlexNet, while the power consumption for Cifar is only 24.96% of that. Compared with a CPU+GPU implementation achieving 6.90× speedup for Cifar, although its performance for larger-scale networks falls short of GPU, the minimum power consumption is only 14.98% of that. When compared with existing research, it achieves up to 6.29× speedup. In particular, compared with accelerators generated by large platforms, even when achieving only comparable performance, it operates at a lower clock frequency.

Full Text

Preamble

Design and Scheduling of Convolutional Neural Network Accelerators for Cloud FPGAs

Cai Ruichu[1], Yu Yang[1], Zhong Chunrong[1], Lu Ye[2], Chen Yao[1],[3]†

(1. College of Computer Science, Guangdong University of Technology, Guangzhou 510006, China; 2. College of Computer & Control Engineering, Nankai University, Tianjin 300350, China; 3. Advanced Digital Sciences Center, Singapore 138602, Singapore)

Abstract: The high computational complexity of convolutional neural networks (CNNs) impedes their widespread deployment in real-time and low-power applications. Existing software implementations struggle to meet the performance and power requirements of CNNs, while traditional FPGA-oriented CNN construction methods suffer from complex design flows, long development cycles, and limited optimization space. To address these challenges, this paper proposes a design and scheduling mechanism for CNN accelerators targeting cloud FPGAs. By leveraging High-Level Synthesis (HLS) technology, introducing loop tiling parameters, and reordering convolutional layer loops, the design constructs networks modularly and performs parameter expansion to further optimize accelerator processing. Through analysis of system task and resource characteristics, scheduling schemes are summarized and optimized from both control flow and data flow perspectives. Compared with existing work, the proposed design provides a solution that simultaneously achieves flexibility, low energy consumption, high energy efficiency, and high performance, while also exploring efficient and universal scheduling schemes for accelerators. Experimental results demonstrate that compared with CPU implementations, the design achieves 8.48× speedup for AlexNet while consuming only 24.96% of the power for Cifar. Compared with CPU+GPU implementations that achieve 6.90× speedup for Cifar, although the performance for large-scale networks falls short of GPU, the minimum power consumption is only 14.98% of the GPU solution. Compared with existing research, the design achieves up to 6.29× speedup. Furthermore, compared with accelerators generated for large platforms, even when achieving only comparable performance, our design operates at a lower clock frequency.

Keywords: convolutional neural network; field programmable gate array; high-level synthesis; accelerator; scheduling

---

0 Introduction

In recent years, deep learning has achieved remarkable development, triggering tremendous transformations across various domains. Convolutional neural net-

work (CNN) methods have become dominant in image processing [1-2]. CNNs are widely employed in computer vision [3-4] and have emerged as the most effective approach for image recognition and classification tasks. Research indicates that CNNs are replacing many traditional composite algorithms used for vision tasks [5], with applications including face recognition [6] and traffic sign detection [7]. CNNs have become essential tools in numerous scientific fields, finding increasingly broad applications in computer vision, pattern recognition [8], and natural language processing [9].

As CNN models evolve toward larger and deeper architectures, and as data volumes continue to grow, the required memory capacity and number of operations increase exponentially. During CNN inference, the computation pattern exhibits highly repetitive pipelines and parallelism [10]. However, general-purpose processors follow the von Neumann architecture, whose sequential instruction execution characteristic is ill-suited for exploiting CNN parallelism. Software-based CNN implementations cannot satisfy application demands in terms of real-time performance and power consumption, making CNN computation patterns highly amenable to hardware acceleration.

While GPUs can deliver sufficient performance levels [2,11,16], their high power consumption and low energy efficiency prevent them from meeting application acceleration requirements. ASIC solutions [17] require high design and manufacturing costs while offering limited flexibility. Among these different hardware acceleration devices, FPGAs have emerged as ideal candidates for hardware acceleration. Compared with embedded platforms, cloud FPGAs aim to provide substantial logic and memory resources, offering generic service logic such as PCIe, DDR control, and clock management. Using FPGAs in cloud environments enables both fast and energy-efficient CNN inference.

Nevertheless, FPGA design presents significant challenges: the learning curve for target hardware design is steep, and development cycles are long. While FPGA capacity has grown rapidly, the development efficiency of traditional design tools has not increased correspondingly. Consequently, higher engineering application complexity implies longer development cycles. High-Level Synthesis (HLS) [15] represents an effective technology for improving FPGA development efficiency, making it possible to automatically generate optimized algorithmic execution code for FPGAs using high-level programming languages (e.g., ANSI C/C++ or LabVIEW). HLS enables rapid design space exploration from both hardware and software perspectives, bringing great convenience to FPGA design and substantially reducing FPGA development time—even below that of DSP and GPU [24]—without compromising performance.

This paper introduces the design and scheduling scheme for CNN accelerators targeting cloud FPGAs. The main contributions are:

a) Construction of a reusable CNN accelerator template library supporting HLS, simplifying the design and generation of CNN accelerators on cloud FPGAs while fully leveraging the parallel advantages of FPGAs and the

application benefits of cloud computing platforms.

b) Proposal of a universal accelerator scheduling mechanism that maximally utilizes hardware resources and off-chip data transfer bandwidth to minimize network model processing latency.

---

1 Related Work

As a multiply-accumulate (MAC)-intensive algorithm, CNN has inspired numerous accelerators in ASIC, GPU, and FPGA-based literature [10-12]. While GPUs can provide high processing performance for deep learning algorithms compared with traditional CPU platforms, their higher power consumption leads to low computational efficiency. ASIC solutions can achieve optimal balance between performance and power but suffer from high design and manufacturing costs and lack advantages in flexibility. FPGAs, with their abundant computational resources, high energy efficiency, short development cycles, and strong reconfiguration capabilities, can meet CNN operational requirements, fully exploit parallel characteristics in CNNs [13], and achieve CNN computational acceleration under low-power constraints, striking an appropriate balance between performance and power consumption [14].

In [13], the authors implemented a CNN accelerator using Zynq SoC with an 8-way parallel engine. However, this accelerator is designed for specific convolution kernels; when kernel sizes vary, it cannot fully exploit parallel computational performance, resulting in substantial resource waste. Savich et al. [18] implemented a multilayer perceptron network on FPGA and analyzed in detail the impact of fixed-point and floating-point representations as well as serial and parallel implementations on hardware-based neural network performance. However, this implementation did not provide a specific and complete FPGA design scheme for CNNs.

By employing loop optimization techniques [19] and well-designed caching [20], CNNs can exhibit deterministic data access patterns, thereby maximizing data reuse and reducing off-chip data transfer. In [12], Zhang et al. used polyhedral methods to evaluate these loop nest optimization techniques, employing the roofline model [21] to find effective balance among optimizations to improve performance and reduce FPGA area. However, their method only tested layers 1-5 of the network [2], artificially producing high compute-to-communication (CTC) ratios by avoiding evaluation of communication-intensive fully connected layers (layers 6-8). Literature [22] also presented similar tests of only layers 1-5, implying that fully connected layers would be handled by the host rather than using a universal accelerator design for both convolutional and fully connected layers.

Gokhale et al. [23] designed a CNN coprocessor using numerous DSP units and off-chip DDR memory in FPGAs. To achieve control of complex operational

modules, this design still required a PC as a host machine, which also participated in partial computations. Like several aforementioned works, it did not consider well-designed scheduling schemes for complex system logic and module control.

For large-scale accelerators, storage and logic resources (excluding DSPs) in cloud FPGAs are typically sufficient. However, current CNN algorithms involve massive weight data, generating numerous intermediate results during computation with input data. This limitation makes on-chip RAM resources in FPGA platforms typically insufficient to buffer all data, requiring substantial off-chip memory in application systems.

---

2 HLS and Convolutional Neural Networks

2.1 Advantages and Limitations of HLS Technology

Traditionally, designing FPGA code using Register Transfer Level (RTL) description languages suffers from complex processes, long cycles, and limited optimization space. Leveraging HLS technology enables rapid design space exploration from both hardware and software perspectives, bringing great convenience to FPGA design and substantially reducing FPGA development cycles while maintaining performance. HLS-based optimization is a practical approach to improving overall design efficiency through techniques such as specifying pipeline, unrolling loops, and handling data transfers. HLS is highly efficient, allowing FPGA hardware accelerators to be designed using high-level languages like C, C++, and OpenCL. Compared with high-level programming languages like C++, HLS requires memory requirements and target function interfaces to be declared as static before compilation. Certain C/C++ syntax cannot be synthesized, such as operating system-dependent functions, dynamic memory allocation, and standard template libraries. To ensure efficient and usable designs, this paper fully considers these limitations.

2.2 CNN Model Functional Layers

A typical convolutional neural network consists of convolutional layers, pooling layers, fully connected layers, and activation layers, as shown in Figure 1 [Figure 1: see original paper].

The input feature map of a convolutional layer undergoes convolution operations through local connections with convolution kernels. The convolution results are accumulated and added with a bias to produce the output feature map. This process extracts different features from input images, and stacking multiple convolutional layers extracts more advanced features. Generally, a convolutional layer can be expressed as:

$$\sum_{l}^{l-1} X = X * W + B \quad i \in N_j$$

where X_j^l represents the output feature map corresponding to the j-th convolution kernel in layer l, N_i represents the selection of input feature maps for the current convolution, $W_{i,j}^l$ represents the i-th weight coefficient of the j-th convolution kernel in layer l, and B_j^l represents the bias coefficient corresponding to the j-th convolution kernel in layer l.

The pooling layer operates similarly to the convolutional layer. Connected to convolutional layers, pooling layers perform secondary feature extraction. Maximum or average sampling is typically employed for pooling operations to reduce the scale of input matrices. Pooling operations effectively reduce feature map dimensions to save substantial memory for storing intermediate results, and pixel selection ensures the most relevant features propagate to the next layer.

The fully connected layer, also called a linear layer, is a special type of convolutional layer. It integrates local information from convolutional or pooling layers that distinguishes categories, obtaining vector-form output through linear spatial transformation. The number of output feature maps from the last linear layer equals the number of recognition categories.

Activation functions in activation layers compress pixel values to specific function ranges, preventing unlimited value increase due to multiply-accumulate operations in convolutional layers. These functions smooth classification boundaries by adding nonlinearity. Commonly used activation functions include Sigmoid, ReLU, and Tanh. A convolutional layer after nonlinear transformation through an activation function can be represented as Equation (2), where f is the activation function.

---

3 Template Function Library Design Based on HLS

This chapter focuses on CNN network models, further optimizing the design methodology from literature [12] through modular network construction, improved computational parallelism, enhanced resource utilization efficiency, and increased generality of accelerator templates.

3.1 Performance Optimization

A typical convolutional layer consists of 6-level for-loops [12]. Drawing from [12], this design adopts a blocking implementation approach, introducing loop tiling parameters for convolutional layer accelerators to tile loops along input and output dimensions. By reordering convolutional layer loops, the design can leverage loop unrolling and pipelining optimizations provided by HLS tools for hardware generation optimization. Finally, cache optimization directives are

applied to enable parallel execution of convolutional computing units and cache unit data fetching, thereby improving intra-accelerator parallelism. Building upon this, before high-level synthesis of the convolutional layer accelerator, its values are determined as fixed accelerator parameters and expanded. Offset parameters are designed according to FPGA-generated accelerator hardware address mapping characteristics to represent address offsets. Using address offsets can improve on-chip resource utilization efficiency, increase data cache size, reduce data caching frequency, and accelerate convolutional computation processes. The optimized convolutional computation process is shown in Figure 2 [Figure 2: see original paper].

Since both convolutional and pooling computations require extensive loops with similar data acquisition and computation processes, the aforementioned convolutional accelerator optimization methods are equally applicable to pooling layer accelerators. However, because pooling does not change input feature dimensions, pooling layer accelerators share input and output feature dimensions.

3.2 Templated Construction of Network Layer Functions

During CNN forward propagation, inter-layer operations are independent, and identical layer types share the same operational pattern. Therefore, universal template functions can be designed for different layer types to implement CNN forward computation. CNN forward propagation proceeds sequentially layer-by-layer according to network structure, meaning adjacent layers have data dependencies and cannot be processed in parallel. Accordingly, based on the performance optimization design above, designing sufficiently universal network layer templates enables reuse of single-layer computational resources to conserve hardware resources, allowing accelerators to implement complete CNN forward computation with higher computational energy efficiency.

To enable CNNs of different scales to reuse hardware computational units, hardware computational units must be designed flexibly with parameterized configuration, giving accelerator templates the ability to be configured and restructured according to different network models. Data types, data bit widths, accelerator cache sizes, and data processing parallelism degrees are all accelerator template parameters during programming implementation. Meanwhile, data acquisition methods, data processing modules, and data output modules are also parameterized and can be adjusted and restructured based on applications.

a) **Convolutional Layer**: The convolutional layer obtains output through three-dimensional multiplication and accumulation of input feature maps and convolution kernel weights, processing three-dimensional input and output data. Based on the computational nature and blocking implementation of convolutional layers, the parameter list is designed as output/input feature numbers, output feature size, kernel size, input/output feature offsets, stride size, padding size, and activation type. According to FPGA-generated accelerator hardware address mapping characteristics,

offset parameters are designed to represent offsets. The convolutional accelerator template is shown in Figure 3 [Figure 3: see original paper].

Where Tn and Tm are input/output buffer feature numbers, I_{BUF} and O_{BUF} are input/output buffer sizes, and W_{BUF} is the weight buffer size.

b) **Pooling Layer**: The pooling layer adopts a sliding window approach similar to the convolutional layer to process input data, computing maximum or average values for each filter' s input. Considering different connection relationships between convolutional and pooling layers in various networks, pooling layers are instantiated as independent accelerators identical to convolutional accelerators to enhance applicability and reconfigurability for different models. Adopting the same accelerator design as convolutional layers, and since pooling does not change input feature dimensions, only one feature dimension parameter is needed. Its parameter definition is identical to the convolutional accelerator, as shown in Figure 4 [Figure 4: see original paper].

c) **Fully Connected Layer**: Analysis of CNN characteristics reveals that fully connected layer computation differs from convolutional layers only by reducing internal loop operations within convolution kernels. The fully connected layer can be regarded as a special form of convolutional layer. Therefore, using convolutional layer accelerators for fully connected layer computation can conserve hardware resources through accelerator reuse, improving accelerator computational efficiency and achieving efficient resource utilization.

d) **Data Storage Banks**: Corresponding data storage banks are allocated for the above accelerators, also parameterized. Data storage banks store three-dimensional input and output data, as well as corresponding weights and biases. Using multiple storage interfaces, the parameter list for input/output storage banks includes input/output buffer feature numbers and input/output buffer sizes. The parameter list for weight storage banks includes input/output buffer feature numbers and weight buffer size. The structure is shown as three-dimensional data storage banks in Figure 3.

---

4 Accelerator Scheduling Mechanism

This chapter introduces the CNN accelerator system architecture based on the aforementioned accelerator template design and the corresponding universal accelerator scheduling mechanism, including software-hardware partitioning, system construction flow, task-resource scheduling models, and corresponding scheduling optimization schemes.

4.1.1 Software-Hardware Partitioning

This paper adopts a software-hardware co-design approach. Combining HLS technology with the "processor+accelerator" architecture implemented on cloud FPGA platforms, task models are mapped to hardware platforms through partitioning and co-design of tasks on software and hardware. The software portion on the cloud Host-CPU serves as the system controller. The CPU runs software code and allocates accelerator tasks to hardware. The hardware side can launch accelerators to achieve computational acceleration based on different acceleration tasks. CNN application execution primarily divides into three phases: data preprocessing, CNN inference, and classification result organization/output. Among these, convolution, pooling, fully connected, and activation functions in CNN inference demand high computation and relatively high data throughput, and are thus partitioned for hardware accelerator implementation. The other two phases are more closely connected to application frontends/backends, with lower computational demands but higher interface requirements, and are partitioned for Host-CPU execution. Additionally, accelerator state control, data transformation and transfer, CNN application execution time measurement, and system debugging are also managed by the Host-CPU. The target system software-hardware partitioning is shown in Figure 5 [Figure 5: see original paper].

4.1.2 System Construction Flow

Building a CNN application first requires understanding network model configuration and weight information. By constructing a CNN model analyzer to analyze network model description files trained using the Caffe deep learning framework [25], they are extracted as network configuration parameters and weight data files. The network configuration parameter file includes network structure, all layer types and input/output feature dimensions, convolution kernel sizes, stride sizes, padding sizes, and other information. Then, based on system software-hardware partitioning, the task model is mapped to software and hardware ends. By extracting network model features and combining them with CNN accelerator templates, accelerator design and generation are realized. According to FPGA on-chip resource structure and characteristics, accelerator buffer construction and generation are implemented. Finally, task dispatching between CPU and FPGA and task-resource scheduling are completed. The system construction flow is shown in Figure 6 [Figure 6: see original paper].

4.2.1 Task Scheduling

Based on the software-hardware co-design and modeling analysis of CNN data access pattern characteristics, the processing of each CNN layer on FPGA can be divided into three operational phases: input, computation, and output phases.

a) **Input Phase**: Moves input data required for computation from CPU off-chip DRAM to FPGA on-chip BRAM. Network weight data has already

been transferred during the preprocessing phase and is not counted in this input phase. Input data portions are burst-read into temporary buffers and moved to accelerator-generated input buffers.

b) **Computation Phase**: When operational units start, they read configuration information from the host side and invoke accelerator launch commands to execute computation based on this information. This phase performs all subsequent computations required after the input phase, executing element-wise multiplication and accumulation between inputs and weights, with results stored in accelerator output buffers.

c) **Output Phase**: Transfers results from accelerator output buffers obtained in the previous phase back to CPU off-chip memory, serving as input data for the next network layer.

Due to the blocking implementation approach, system logic and module control are relatively complex, requiring scheduling of CNN layer tasks. The Host-CPU in the system is responsible for controlling accelerator states and computation processes, as well as data transfer, implemented through host control code. The entire inference process is encapsulated as a task unit and assigned to network accelerators. Network accelerators are encapsulated as PCIe devices, and application call sets do not need to know hardware details. Based on the above analysis, Figure 7 [Figure 7: see original paper] illustrates the task scheduling process for executing convolutional layers.

4.2.2 Resource Scheduling

While storing model parameters and intermediate results in on-chip BRAM can significantly improve performance, FPGA often cannot meet resource requirements for high parallelism due to limitations in computational resources and transfer bandwidth. Therefore, data buffers must be allocated in off-chip dynamic memory to store intermediate results and model parameters. The overall network resources are divided into computational resources and cache resources. Layer computation parameters, weights, and input/output buffers allocated to accelerators belong to computational resources, while cache resources store model parameters and intermediate results. The overall control of resource scheduling is completed by the general processor. When the input phase of task scheduling begins execution, data is transferred from CPU off-chip memory to accelerator input buffers, then control launches the accelerator for computation. After computation completes, results from accelerator output buffers are transferred to CPU off-chip memory. Corresponding to the input and output phases in task scheduling, the data transfer directions between the two resource types map as cache resources $\rightarrow$ computational resources and computational resources $\rightarrow$ cache resources, respectively. Based on the above description, the resource scheduling process is summarized in Figure 8 [Figure 8: see original paper].

4.3.1 Data Flow

During convolution, partial pixels overlap between different positions in the same filter on input feature maps, necessitating storage of some input pixels across different iterations. Considering hardware resources and bandwidth availability, partial data is transferred to accelerator input buffers during each convolutional layer' s input phase. The time consumed by this data transfer phase counts toward network processing time and requires optimization to reduce network processing latency. Input phase waiting time can be minimized by storing only partial input data required for computing Tr rows of output feature maps each time. The design stores Tr * S rows of input, where Tr and Tc are accelerator-configured output feature sizes, and S and Tn are the layer' s stride size and number of channels in the input feature buffer, respectively. Figure 9 [Figure 9: see original paper] illustrates the optimized operational process. Initially, the first Tr * S rows of the window are filled to provide sufficient data to begin computation. When the convolution kernel finishes sliding across these Tr * S rows, a new Tr * S row of input data is loaded into the window for the next iteration, continuing computation in the same manner until all output feature maps are generated.

The on-chip buffer design is based on the fundamental concept of Double Buffering to guarantee parallel read-compute requirements. Double buffering employs ping-pong operations to overlap data transfer time with computation, improving processing efficiency. On-chip buffers are divided into two groups for input and output feature maps, respectively. Each buffer set contains multiple independent, address-continuous buffers, with the number of buffer groups in each input buffer set and output buffer set equal to Tn and Tm, respectively. Figure 10 [Figure 10: see original paper] exemplifies the timing of several computation and data transfer phases. For input feature maps, in phase one, the computation engine processes input0 while simultaneously copying data required for the next input phase to input1, which will perform the opposite operation in the next input phase. For output feature maps, after completing N/Tn phases of computation and data copying, the resulting output feature maps are stored in output buffer sets. When output0 space is maximally utilized, output1 is used and output data stored in output0 is transferred, with the next output phase performing the opposite operation.

4.3.2 Control Flow

The process of implementing CNN inference on cloud FPGA platforms comprises both software and hardware implementation tasks. Among the three task phases, input and output phases are controlled and implemented by the software side, while the computation phase is executed by the hardware side through accelerator launch commands invoked by the software side. The operations and resources of software and hardware execution parts are independent of each other. This paper adopts Double Buffering design, enabling the current computation phase to parallelize to some degree with the next input phase and

the previous output phase, thereby reducing overall network latency. Corresponding to data flow optimization, the control flow scheduling optimization results in the task scheduling flow shown in Figure 11 [Figure 11: see original paper].

---

5 Experimental Evaluation

5.1.2 Network Models and Test Datasets

Cifar, AlexNet, VGG-16, and their quantized versions were selected as experimental subjects. The quantization method is based on literature [26]; in this evaluation, quantized version input data and weights are quantized to 16-bit. The Cifar-10 dataset and the 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) validation set serve as experimental test datasets.

The following aspects were evaluated in experiments:

a) **Evaluation Metrics**: The generated accelerators were assessed, analyzing processing performance, energy consumption, and energy efficiency across several networks.

b) **Evaluation Targets**: The generated system performance was demonstrated and compared with software implementations and other existing designs.

5.3 Results and Analysis

a) **Accelerator Performance**: Overall performance was measured by timing network inference processing on the cloud FPGA platform implementation. By analyzing network parameters, Table 1 measures and reports accelerator parameter settings and performance for each network model.

Table 1 Accelerator Configuration and Performance of Network Models

Network Model	Data Type	Accelerator Configuration (Tm, Tn, Tr, Tc, I_{BUF})	Time (ms/image)
Cifar	float	32,32,4,4,5	
AlexNet	float	133,10,4,4,19	
VGG-16	float	64,19,4,4,32	

b) **Comparison with Pure Software Versions**: Results are shown in Table 2 . Acceleration for all network models uses latency data measured from CPU implementations as baselines to demonstrate speedup ratios. Energy consumption and energy efficiency of GPU and FPGA are also compared with CPU.

Table 2 Comparison Results with Software Versions

Platform	Clock (MHz)	Energy (Watt)	Delay/Image (ms)	Speedup (×)
E5-2430 CPU				
CPU+GPU E5-2609 + K80				
This Work (CPU+FPGA) AWS F1	125MHz			

As shown in Table 2 , for small networks, CPU can achieve performance comparable to GPU. However, leveraging architectural flexibility, FPGA can outperform both CPU and GPU. Compared with CPU implementation, this design achieves 8.48× speedup for AlexNet while consuming only 24.96% of the power for Cifar. Compared with CPU+GPU implementation achieving 6.90× speedup for Cifar, although performance for large-scale networks falls short of GPU, the minimum power consumption is only 14.98% of the GPU solution. This analysis demonstrates that compared with CPU implementation, the generated system always performs better in terms of performance and energy efficiency, thanks to FPGA' s higher parallel execution capability. Compared with GPU implementation, even when the accelerator system performance does not reach GPU peak performance, it can still leverage FPGA' s high computational density advantages to improve energy efficiency, demonstrating significant energy consumption advantages.

c) **Comparison with Published Research**: Results are shown in Table 3 .

Table 3 Comparison Results with Existing Methods

Work	Platform	Network	Data Type	Clock (MHz)	DSP Utilization (%)	Energy (Watt)	Delay/Image (ms)
Literature [27]	VX690T	VGG-16	fixed16				59.9 (41.77)
Literature [28]	Arria 10	VGG-16	fixed16				
Literature [28]	Stratix V	VGG-16	fixed16				
Literature [29]	Stratix V	VGG-16	fixed8-16				
	VU118	VGG-16	fixed16				

The accelerator system generated by this design achieves comparable or even superior performance in per-image processing latency and energy efficiency compared with existing work, reaching up to 6.29× speedup. Compared with accelerators designed on smaller platforms [27], this design-generated accelerator system achieves better performance by fully utilizing on-chip resources. Compared with accelerators generated for large platforms [28-29], this design-generated system achieves comparable or better performance, and even when achieving only comparable performance, it operates at lower clock frequencies.

---

6 Conclusion

Based on the highly repetitive pipeline and parallel characteristics of CNN computation patterns, this paper proposes a design and scheduling mechanism for CNN accelerators targeting cloud FPGAs. Experimental results demonstrate that the accelerator can effectively improve operational speed while reducing power consumption. Evaluation results for network models including Cifar, AlexNet, and VGG-16 implemented using this design system show comparable or better performance compared with existing work. Compared with CPU and GPU, the power consumption and performance on cloud FPGAs demonstrate significant advantages.

Future work will explore the possibility of splitting inference across multiple accelerators. The existing work employs modular design with high system extensibility, allowing various new computation types to be easily integrated into the design. Future efforts will further expand the design to support new features and provide more adaptive solutions for CNN mapping on FPGAs.

References

[1] Zhou Feiyan, Jin Linpeng, Dong Jun. Review of convolutional neural network [J]. Chinese Journal of Computers, 2017, 40 (6): 1229-1251.

[2] Krizhevsky A, Sutskever I, Hinton G E. Imagenet classification with deep convolutional neural networks [C]// Advances in Neural Information Processing Systems. New York: ACM Press, 2012: 1097-1105.

[3] Saxena S, Verbeek J. Heterogeneous face recognition with CNNs [C]// Proc of the European Conference on Computer Vision. Berlin: Springer Press, 2016: 483-491.

[4] Ji Shuiwang, Xu Wei, Yang Ming, et al. 3D convolutional neural networks for automatic human action recognition: USA, US8345984 [P/OL]. (2010-06-11) [2013-01-01]. http://www.google.com/patents/US8345984.

[5] Shao Hong, Chen Shuang, Zhao Jieyi, et al. Face recognition based on subset selection via metric learning on manifold [J]. Frontiers of Information Technology & Electronic Engineering, 2015, 16 (12): 1046-1058.

[6] Levi G, Hassncer T. Age and gender classification using convolutional neural networks [C]// Proc of Computer Vision and Pattern Recognition Workshops. Piscataway, NJ: IEEE Press, 2015: 34-42.

[7] Peemen M, Mesman B, and Corporaal H. Speed sign detection and recognition by convolutional neural networks [C]// Proc of the 8th International Automotive Congress. 2011: 162–170.

[8] Lee W Y, Park S M, Jang I H, et al. CNN-based shoe-upper pattern recognition and generation of adhesive point [J]. Journal of Institute of Control Robotics & Systems, 2017, 23 (9): 725-731.

[9] Cai Huiping, Wang Lidan, Duan Shukai. Sentiment classification model based on word embedding and CNN [J]. Application Research of Computers, 2016, 33 (10): 2902-2905.

[10] Sankaradas M, Jakkula V, Cadambi S, et al. A massively parallel coprocessor for convolutional neural networks [C]// Proc of IEEE International Conference on Application-Specific Systems, Architectures and Processors. Piscataway, NJ: IEEE Press, 2009: 53-60.

[11] Potluri S, Fasih A, Vutukuru L K, et al. CNN based high performance computing for real time image processing on GPU [C]// Proc of Workshop on Nonlinear Dynamics & Synchronization & Intl Symposium on Theoretical Electrical Engineering. Berlin: Springer Press, 2011: 1-7.

[12] Zhang Chen, Li Peng, Sun Guangyu, et al. Optimizing FPGA-based accelerator design for deep convolutional neural networks [C]// Proc of ACM//SIGDA International Symposium on Field-Programmable Gate Arrays. New York: ACM Press, 2015: 161-170.

[13] Cadambi S, Majumdar A, Becchi M, et al. A programmable parallel accelerator for learning and classification [C]// Proc of International Conference on Parallel Architectures and Compilation Techniques. New York: ACM Press, 2010: 273-284.

[14] Wei Xuechao, Yu Codyhao, Zhang Peng, et al. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs [C]// Proc of Design Automation Conference. ACM, New York: ACM Press, 2017.

[15] Farrens M, Chong F T, Oskin M. HLS: combining statistical and symbolic simulation to guide microprocessor designs [C]// Proc of International Symposium on Computer Architecture. New York: ACM Press, 2000: 71-82.

[16] Strigl D, Kofler K, Podlipnig S. Performance and scalability of GPU-based convolutional neural networks [C]// Proc of Euromicro International Conference on Parallel, Distributed and Network-Based Processing. Piscataway, NJ: IEEE Press, 2010: 317-324.

[17] Ferry C. Hardware accelerated convolutional neural networks for synthetic vision systems [C]// Proc of IEEE International Symposium on Circuits and

Systems. Piscataway, NJ: IEEE Press, 2010: 257-260.

[18] Savich A W, Moussa M, Areibi S. The impact of arithmetic representation on implementing MLP-BP on FPGAs: A study [J]. IEEE Trans on Neural Networks, 2007, 18 (1): 240-252.

[19] Peemen M M, Mesman B B, Corporaal H H. Optimal iteration scheduling for intra-and inter-tile reuse in nested loop accelerators [D]. Eindhoven: Eindhoven University of Technology, 2013.

[20] Peemen M, Setio A A A, Mesman B, et al. Memory-centric accelerator design for convolutional neural networks [C]// Proc of IEEE, International Conference on Computer Design. Piscataway, NJ: IEEE Press, 2013: 13-19.

[21] Williams S, Waterman A, Patterson D. Roofline: An insightful visual performance model for floating-point programs and multicore architectures [J]. Office of Scientific & Technical Information Technical Reports, 2009, 52 (4): 65-76.

[22] Motamedi M, Gysel P, Akella V, et al. Design space exploration of FPGA-based deep convolutional neural networks [C]// Proc of Design Automation Conference. Piscataway, NJ: IEEE Press, 2016: 575-580.

[23] Gokhale V, Jin Jonghoon, Dundar A, et al. A 240 G-ops/s mobile coprocessor for deep neural networks [C]// Proc of IEEE Conference on Computer Vision and Pattern Recognition Workshops. Piscataway, NJ: IEEE Press, 2014.

[24] Baklouti M, Ammar M, Marquet P, et al. A model-driven based framework for rapid parallel SoC FPGA prototyping. [J]. Journal of Biological Chemistry, 2011, 272 (12): 7797-800.

[25] Jia Yangqing, Shelhamer E, Donahue J, et al. Caffe: Convolutional architecture for fast feature embedding [C]// Proc of ACM International Conference on Multimedia. New York: ACM Press, 2014: 675-678.

[26] Gysel P, Motamedi M, Ghiasi S. Hardware-oriented approximation of convolutional neural networks [J]. arXiv preprint arXiv: 1604. 03168, 2016.

[27] Zhang Chen, Fang Zhenman, Zhou Peipei, et al. Caffeine: towards uniformed representation and acceleration for deep convolutional neural networks [C]// Proc of International Conference on Computer-Aided Design. New York: ACM Press, 2016: 12.

[28] Ma Yufei, Cao Yu, Vrudhula S, et al. An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks [C]// Proc of International Conference on Field Programmable Logic and Applications. Piscataway, NJ: IEEE Press, 2017: 1-8.

[29] Qiu Jiantao, Wang Jie, Yao Song, et al. Going deeper with embedded FPGA platform for convolutional neural network [C]// Proc of ACM//SIGDA International Symposium on Field-Programmable Gate Arrays. New York: ACM Press, 2016: 26-35.