# A CNN Hardware Accelerator in FPGA for Stacked Hourglass Network

Dongbao Liang, Jiale Xiao, Yangbin Yu, and Tao Su(✉)

San Yat-Sen University, Guangzhou 510006, Guangdong, China
sutao@mail.sysu.edu.cn

**Abstract.** Staked hourglass network is a widely used deep neural network model for body pose estimation. The essence of this model can be roughly considered as a combination of Deep Convolutional Neural Networks (DCNNs) and cross-layer feature map fusion operations. FPGA gains its advantages in accelerating such a model because of the customizable data parallelism and high on-chip memory bandwidth. However, different with accelerating a bare DCNN model, stacked hourglass networks introduce implementation difficulty by presenting massive feature map fusion in a first-in-last-out manner. This feature introduces a larger challenge to the memory bandwidth utilization and control logic complexity on top of the already complicated DCNN data flow design. In this work, an FPGA accelerator is proposed as a pioneering effort on accelerating the stacked hourglass model. To achieve this goal, we propose an address mapping method to handle the upsample convolutional layers and a network mapper for scheduling the feature map fusion. A 125 MHz fully working demo on Xilinx XC7Z045 FPGA achieves a performance of 8.434 GOP/s with a power efficiency of 4.924 GOP/s/W. Our system is 296× higher than the compared Arm Cortex-A9 CPU and 3.2× higher power efficiency, measured by GOP/s/W, than the GPU implementation on Nvidia 1080Ti.

**Keywords:** Stacked hourglass network · Convolutional Neural Network · Pose estimation · Hardware accelerator · FPGA

## 1 Introduction

In deep Convolutional Neural Networks (DCNNs), deeper layers yield larger perception fields to the original images and, therefore, encapsulate higher-level feature information, which is opposite to the shallow (near-input) layers. In body pose estimation tasks, a common practice to accurately locate body key points is to combine the abstract global features with shallow local features [1]. Following such motivation, stacked hourglass network are commonly used in pose estimation tasks and achieves an unprecedented accuracy on the MPII Human Pose dataset with an average 90.9% percentage of detections [1]. However, this design of model poses new challenges to the system design complexity on FPGAs compared to solely accelerating a DCNN model for classification tasks [4–7].

https://doi.org/10.1007/978-981-15-8135-9_8

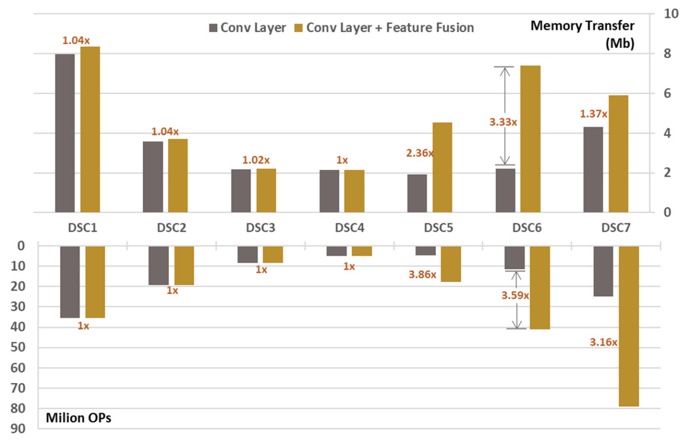

**Fig. 1.** Feature map fusion branches introduce up to 3.3 times more memory transfer and up to 3.59 times more operations to normal convolutional layers

In Fig. 1, DSC stands for depthwise separable convolution module [17] and DSC1-DSC7 are the layers in a single hourglass model (see model details in Sect. 2). The feature maps in the shallow layers are transferred between on- and off-chip for the feature fusion branches, which introduces up to 3.3 times more memory transfer and 3.59 times more computation compared to a simple cascaded convolutional layer structure. Such computation nature changes the workload from a compute-bound task into a memory-bound one [12, 19].

Central Processing Unit (CPU) suffers from its sequential execution nature when dealing with the parallel DCNN computation. General Purpose Graphic Processing Unit (GPGPU) is also a naturally fitted platform in accelerating DCNN applications, but the sweet point for GPGPUs normally requires a relatively large input batch size [11]. Additionally, the data synchronization mechanism in GPGPUs is not specially designed for neural network computation. Field-Programmable Gate Array (FPGA), on the other hand, gains its advantages in accelerating deep neural network tasks because of its customizable parallel logic and high power efficiency. There have been many implementations of DCNNs on FPGA in recent years [2–14]. Systems like NEURAghe [6] and SnowFlake [5] mainly target on DCNN classification task acceleration. As above-mentioned, DCNN classification models do not introduce extra computation and memory transfer from the feature fusion branches. It is hard to make straight and fair comparisons to these works. For example, if scales up a typical backbone DCNN accelerated in [5, 6] adding feature map fusion branches, the system performance reported in these works can massively drop due to the multi-scale feature fusion operations and newly introduced memory transfer. Although very few works exactly address the stacked hourglass network, its building block, depthwise separable convolution modules, are carefully considered in the system architecture designs in [10–12]. However, model redundancy reduction techniques are applied in these works to help realizing feasible FPGA mapping. These techniques, moreover, are orthogonal to our proposed system.

In most of the FPGA-based DCNN implementations, designers are benefited from configurable architecture to perform multiple functions of the algorithms and on-chip memories to overcome the limitation of the memory bandwidth. Works like [2] and [3] try to realize a general structure for all possible network structures with parameterized design, while other references [4] and [8] try to fit the CNN model into the embedded applications by dedicated memory arrangement and tiling strategies. Reference [9] and [10] explore the sparsity within the DCNN, using compression or pruning method to make fully use of the memory bandwidth. With the rapid development of the algorithms, new DCNN structures appears and reference [13] and [14] explore new mapping strategies for the networks with residual blocks like ResNet and Xception.

In this work, our new system architecture is designed for the stacked hourglass model with above difficulty handled by our proposed network mapper module and an address mapping method. The main contributions of this work are listed as below:

1. Our work is pioneering effort in accelerating the commonly used stacked hourglass network model on FPGAs. Our system achieves 296× higher than the compared CPU and 3.2× higher power efficiency than the examined GPU implementation.
2. To implement the logic for the feature map fusion branches, we propose a network mapper module for efficiently managing the storage and fetching of long-distant residual results. In addition, the network mapper also converts the stacked hourglass network into instructions for the accelerator automatically.
3. "Channel-first" data arrangement is proposed to enhanced the performance of $1 \times 1$ convolutions which are heavily used in depthwise separable convolutions.

This paper is organized as follows. Section 2 provides the background of stacked hourglass network and the separable depthwise convolution. Section 3 describes the architecture of the accelerator, including the processing engine and the organization of the on-chip memory. The dedicated design of control signals for the accelerator will be described in Sect. 4. Section 5 gives the experimental results on the performance of the accelerator, also provides a demo of the accelerator using on the real-time pose estimation application. The summary will be given in Sect. 6.

## 2 Background

### 2.1 Stacked Hourglass Network

Stacked hourglass network was first introduced in [1] to improve the accuracy of pose estimation task. Figure 2 gives us an overview of the architecture of an hourglass module and the stacked hourglass network. In the network, the input feature map is firstly scaled down to the intermediate with very low resolution but much more channels, then scaled up to the original size again. Scale down is done by the stride-n ($n > 1$) convolutions or pooling and the upsample performs the reverse operation. It's worth noticing that feature maps are combined across multiple resolutions by doing summations within the hourglass module, named by its shape. Residual module is also used heavily in the hourglass module and filters greater than $3 \times 3$ are abandoned here, which make it easier for the hardware design. The residual module sometimes is replaced by depthwise separable convolution to reduce the complexity of the network and the number of parameters further, which will be introduced in the next chapter. After the intermediate supervision process for the output of the hourglass module is included, hourglass modules can be stacked up successively to create a deeper network.

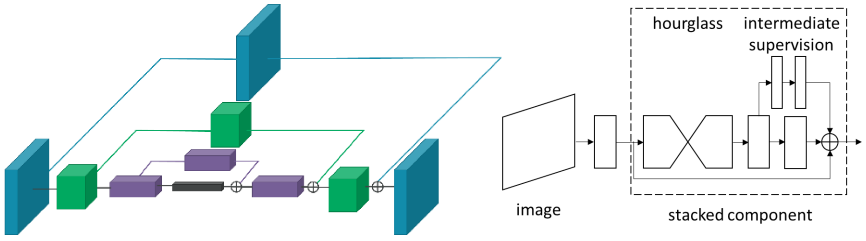

**Fig. 2.** An illustration of a single "hourglass" module (left), and the stacked hourglass network (right).

## 2.2 Depthwise Separable Convolution

Depthwise separable convolution was first introduced in [16]. In this kind of convolution, a standard convolution is split into two steps. Depthwise convolution firstly extracts features on the input feature map of different channels separately, and then pointwise convolution follows up to combine all the results in different channels with $1 \times 1$ convolution. Differences between standard convolution, depthwise convolution and pointwise convolution are illustrated in Fig. 3.

Depthwise separable convolution is proved to have much less parameters and arithmetic operations to achieve comparable accuracy for popular CNN networks [15]. A simple mathematical proof is shown here. Considering an $D \times D \times M$ input feature map produces an $D \times D \times N$ output feature map after a convolution operation. If standard convolution is chosen, the number of parameters P for the kernel with size of $K \times K$ is

$$P_{SC} = K \times K \times M \times N \tag{1}$$

And the computational cost O of the standard convolution is

$$O_{SC} = D \times D \times K \times K \times M \times N \tag{2}$$

However, if depthwise separable convolution is chosen, the corresponding number of parameters and the computational cost is

$$P_{DSC} = K \times K \times M + M \times N \tag{3}$$

$$O_{DSC} = D \times D \times K \times K \times M + D \times D \times M \times N \tag{4}$$

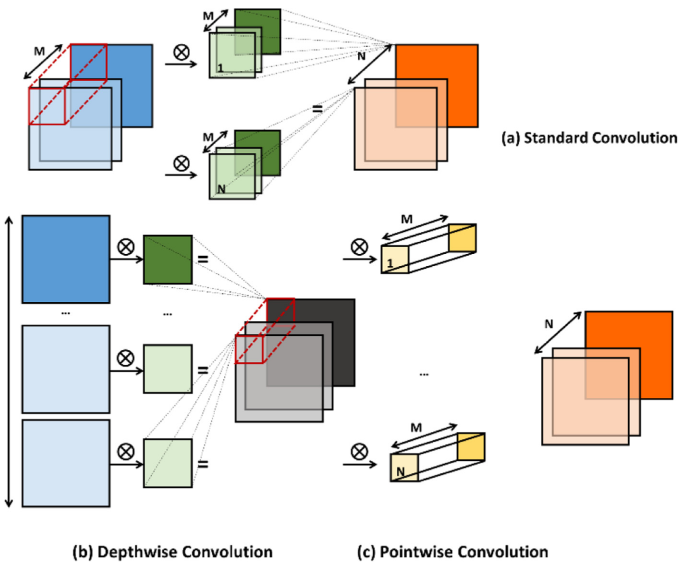

**Fig. 3.** Comparison of different kinds of convolutions.

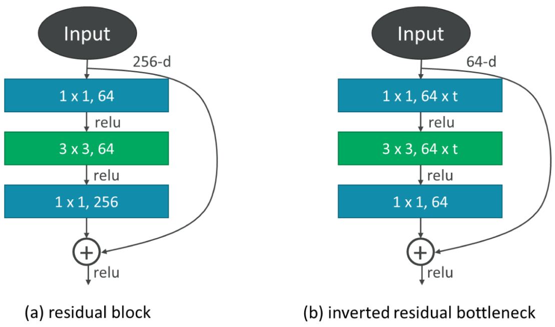

**Fig. 4.** Different kinds of residual blocks. Note that t (t > 1) in (b) is the expansion factor.

Thus, the reduction of the parameters and the computational cost comparing to the standard convolution is

$$F_P = \frac{K \times K \times M + M \times N}{K \times K \times M \times N} = \frac{1}{N} + \frac{1}{K^2} \tag{5}$$

$$F_O = \frac{D \times D \times K \times K \times M + D \times D \times M \times N}{D \times D \times K \times K \times M \times N} = \frac{1}{N} + \frac{1}{K^2} \tag{6}$$

A popular choice of K is 3 so the common reduction in number of parameters and the computational cost is 8 to 9 times.

Depthwise separable convolution is successfully applied to the usual object detection and classification tasks in MobileNetV1 [15], and the successor MobileNetV2 [18]. In MobileNetV2, performance is further improved by the new operation called bottleneck, also get its name from the shape. Within the structure, one more 1 × 1 convolution is placed before the depthwise convolution and the feature map fusion connects two bottlenecks from bottom to top. The inverted residual bottleneck layers (differ from the common residual block in [17]) is proved to be memory efficient evidently for feature maps with less channels in the bottleneck (Fig. 4). This structure is also widely used in stacked hourglass network acting as a substitution of residual modules.

## 3 Hardware Design

In this chapter, we present the architecture of the hardware accelerator for running stacked hourglass network in the inference phase. The acceleration lies in the dedicated data path design and memory management.

### 3.1 Overall Architecture

The block diagram in Fig. 5 shows the overview of the whole architecture of the accelerator. Within the accelerator, a control module receives the control instructions from the host CPU and transform them into internal control signals for the other module in the accelerator. Processing engine (PE) array is responsible for all calculations in stacked hourglass network. Buffer module moves input data and weight from external memory to on-chip buffer, also move the calculated results came out from PE array back to DRAM. Redistribution module is placed between PE array and buffer module for order rearrangement for input data and output results, which will be introduced in detail in Sect. 3.2.

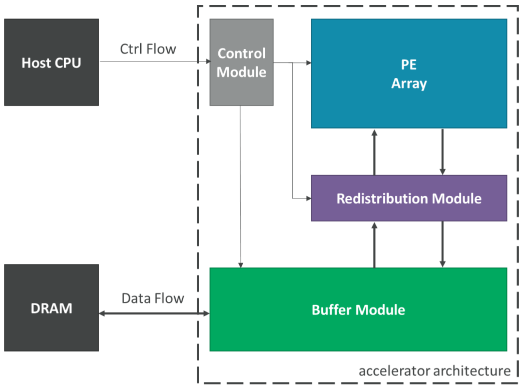

**Fig. 5.** Block diagram of the proposed hardware accelerator.

### 3.2 Processing Engine

In this paper, the hardware accelerator will use 16 PEs in the PE array for parallel computations. Two permutation blocks for each input vector deals with different data

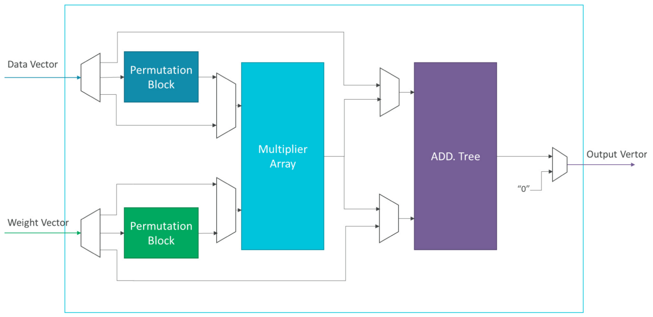

**Fig. 6.** Block diagram of the processing engine.

reuse schemes. There are 18 multipliers in the multiplier array responsible for two 3 × 3 convolutions at one cycle. Adder tree can perform different kinds of summation under different configurations. The architecture of a single PE is illustrated in Fig. 6.

**Vector-Based Calculation.** Each processing engine takes two vectors as input and outputs a result vector, acting like a vector processing unit. To maximize the reuse of the input vector, we carefully choose the size to be 16 × 16-bit, and the permutation block helps us to rearrange the order of data in the vector. Weight vector can be an array of weight data under convolution operations or an array of pixel data when operating residual summations.

**Permutation Block.** Reusing data between two adjacent convolution windows is a way to improve the power efficiency of the accelerator when performing depthwise convolution, and here the permutation block is designed to undertake the job. In the "Row-first" data arrangement (please refer chapter 3.3 for more information about data arrangement in buffers), 4 pixels in the same row can be fetched in one cycle. Figure 7 shows that in 3 × 3 convolution with one padding, two adjacent convolution windows of calculation need 4 pixels in a row at most. Permutation blocks re-arrange data from buffers into two separate vectors for two windows and send them to processing unit. In addition, pixels in the right-most column of window are temporarily store in the module for convolution in the next cycle. The re-arrangement and temporary storage improve the data utilization of input feature map, saving power by reducing read access to buffers.

**Depthwise Convolution.** Depthwise convolution performs convolution for each feature map in different input channels separately. Each processing engine is in charge of all multiplications and accumulations for one input channel so 16 channels can be calculated in parallel at a time. The output vector contains two valid results from two side-by-side 3 × 3 windows for depthwise convolution.

**Pointwise Convolution.** Pointwise Convolution performs convolution operations across different input channel for input pixel data. For the case that input channels are larger than 16, the intermediate result is registered at the output stage of PE and waits for the next intermedia result in the coming round. For more efficient calculation and data fetch, 16-pixel data in data vector will be data from different feature maps but

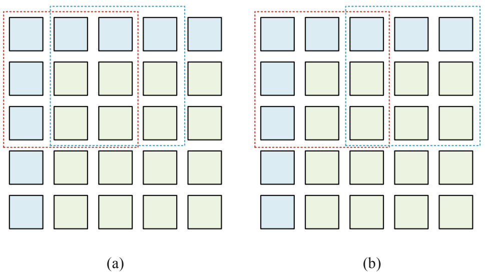

**Fig. 7.** Two adjacent convolution windows in different strided-convolutions. (a) is for stride-1 and (b) is for stride-2. In both figures, blue pixels stand for zero-padding pixels while green pixels are for pixels in feature maps. (Color figure online)

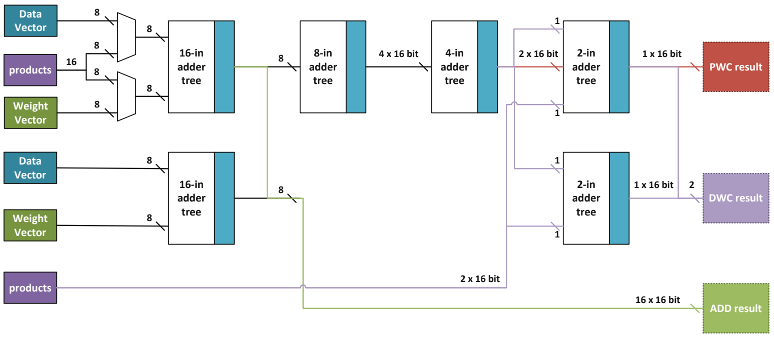

**Fig. 8.** Different configurations of adder tree. (Color figure online)

in the same row and column position, unlike the usual "row-first" order of data. More information about the data arrangement will be discussed in Sect. 3.3.

**Residual Summation.** As it is described in Sect. 2, residual summation performs at the end of every residual module. To reuse the resource of adder tree, residual enter the processing engine through weight vector and performs the summation with the result from the former layer.

**Adder Tree.** Adder tree is configurable to do the summation in depthwise convolution, pointwise convolution and residual addition. It is configured into various number of stages of adders for different layers, as illustrated in Fig. 8. For vector addition used in residual summation, two vector inputs coming from data vector and weight vector will enter two separate 16-in adder trees and produce a 16 $\times$ 16-bit result vector, which only need 1 stage of adders. For pointwise operation, 16 products coming from the former multiplier stage will go through all the adder tree on the upper path to produce 1-element result. A little bit complicated case for depthwise convolution is that 18 products will come from the multiplier arrays (partial sums from two convolution windows), and an extra 2-in adder tree will deal with the 2 extra products (the purple lines in the figure shows how the dataflow goes in depthwise mode) in the last stage and there will be 2-element result for depthwise convolution.

**ReLU.** The ReLU operation, $f(x) = max(0; x)$ is optional in the network. The process is simply replaced the negative results by zero on the output stage.

## 3.3 Memory Organization

The new structure in stacked hourglass network requires much dedicated design for the data flow control to efficiently make use of the memory bandwidth. Therefore, in the proposed architecture, we adapt different strategies for different situations, trying to explore the efficient data flow methods.

**Table 1.** Percentage of different types of operation in stacked hourglass network.

| Operation type | Conv1 | Conv3 | Add |
|---|---|---|---|
| Percentage | 90.41% | 9.32% | 0.27% |

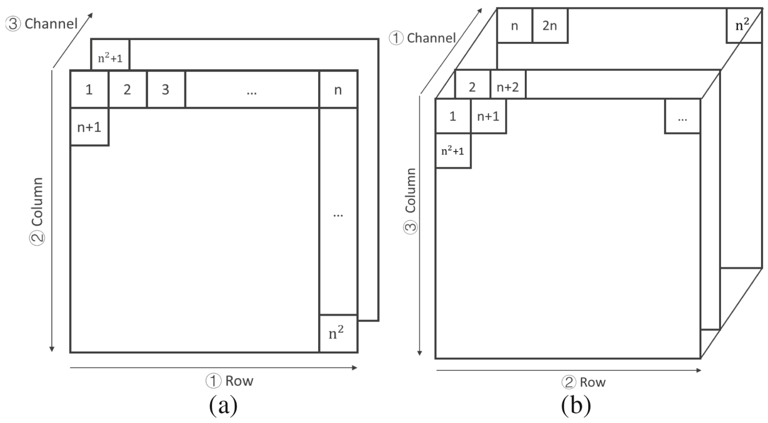

**Fig. 9.** "Row-first" data arrangement (a) and "Channel-first" data arrangement (b).

**Data Arrangement.** From a typical stacked hourglass network, we can see that the pointwise convolution occupies a large part of the whole calculations (Table 1). Pointwise convolution convolves feature maps from different channels and produces one feature map for one time. Traditional "Row-first" data arrangement (which means that data are arranged along the row direction first, then in column and channel direction successively, illustrated in Fig. 9(a)) brings trouble for this kind of convolution due to the discontinuous input data fetch, but works fine for the depthwise convolutions. "Channel-first" (which means that data are arranged along the channel direction first, then row and column direction successively, illustrated in Fig. 9(b)) arrangement is proposed to resolve the troubles. Moreover, to guarantee the computational efficiency for both depthwise and pointwise convolutions, the data arrangement is automatically exchanged during the output stage of the processing engine. Types of upcoming layer is told to the redistribution module as well, so the results streaming out to the result buffer will change to the proper arrangement here.

In our implementation, each PE is attached with 3 buffers, an input buffer storing input activations, an output buffer collecting the output results and a weight buffer to store weights. The input buffer and the output buffer are in the same size with 4 banks each and a total of 256-bit data width while the weight buffer is with the same data width but shorter in depth. When performing depthwise convolution and standard convolution, elements in different rows can be fetched in a time enabling parallel computation for multiple MAC operations in different rows for a convolute window. In contrast, multiple elements in different input channels can be fetch in a time when performing pointwise convolution in "channel-first" data arrangement and speed up the accumulation between input channels.

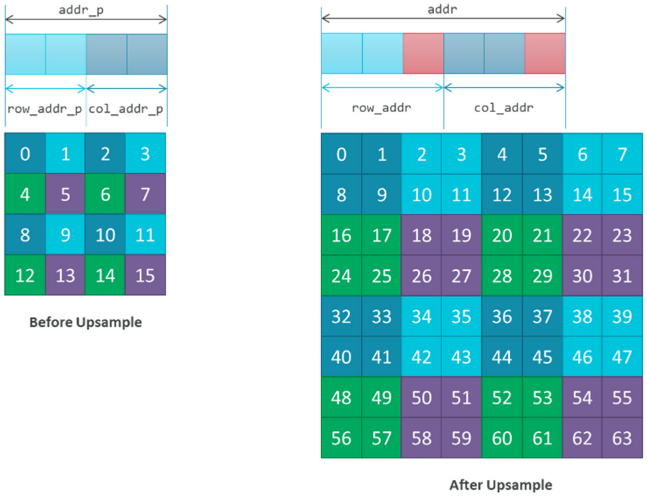

**Fig. 10.** An illustration of address mapping. Note that same color blocks in the result after upsample is represented the same origin pixel before the operation. (Color figure online)

**Address Mapping for Upsample.** Upsample is the operation to scale up the intermediate result to the same size of the residual so that they can perform the summation. To avoid the waste of memory space and bandwidth, address mapping method is proposed here to combine the upsample operation and the following residual summation together in the accelerator. A simple demonstration of address mapping is illustrated in Fig. 10. After upsample in the usual stride 2, one pixel is scaled up into a 2 $\times$ 2 result block with same value in it. When the summation tries to fetch the result of the upsample operation, it actually fetches the input of the upsample 4 times with simple address mapping. Moreover, for some specific circumstance, the mapping is further simplified to the interception and concatenation of the address without any complex address calculations.

**Tiling for Data.** Tiling is necessary for various sizes of input feature maps to accommodate in the on-chip buffer with fixed and limited size. Moreover, data arrangement for different layer is different as mentioned in data arrangement, direction of tiling differs as well. For operands of depthwise convolutions or residual summations, tiling is along the channel direction and for pointwise convolutions, tiling is along the row direction.

## 4 Network Mapper

This hardware accelerator heavily relies on the control instructions sent from the host CPU to control the behaviors and to configure the hardware accelerator. To maximize the performance for the network on the hardware, we design a dedicated network mapper for the proposed accelerator. The mapper acts as a compiler for transforming the high-level network descriptions (like excel form) to the corresponding code-like control instructions.

### 4.1 Overview of Network Mapper

To perform the calculation of one particular layer of stacked hourglass network, the hardware accelerator need to know the information about size of feature map, how the layer cascade, parameters for efficient tiling, etc. It is necessary to do the calculation for those parameters in advance to reduce the extra burden on the accelerator, which is the job for network mapper (Fig. 11) Network mapper first reads through the architecture of the network and do the analysis and calculations. The major job will be as follows.

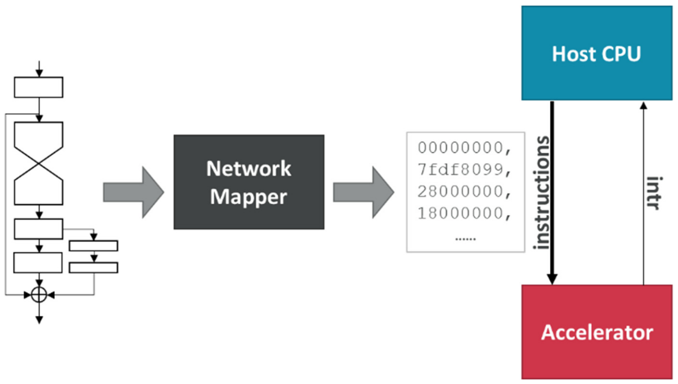

**Fig. 11.** The function of network mapper.

1) Layer Cascade: Some layers can be cascade to reduce the memory bandwidth for less intermediate results, for example, ReLU activation is performed after the calculation of the former layer, and the upsample operation is done with the following residual summation as mentioned in Sect. 3.3.

2) Gather-Scatter Setup: A buffer controller is in charge of the communication between multiple banks in the accelerator and the external memory, acting like a simplify gather-scatter DMA. Different sizes of feature map result in different transfer length for layers and they are calculated in network mapper.
3) Feature Map Tiling: Tiling is always necessary in the acceleration and the tiling method is described in Sect. 3.3. Network mapper figure out the best tiling strategy for different size of feature maps ahead and configure the accelerator through control instructions.

Once the architecture of the network is fixed, control instructions can be generated immediately. The accelerator follows the instructions to calculate for one frame and loop over the instructions for video stream. The size of the instructions for a typical stacked hourglass network is not greater than 20 KB for each frame. Due to the tiny size of the codes, they can be either stored in the on-chip memory or sent by the host CPU for convenient control scheme.

### 4.2 Residual Optimizing

As mentioned in Sect. 2, stacked hourglass network has more than one kind of residual summation branch for multiple resolutions. As the nested hourglass become deeper, it is impossible to store all the intermediate residual results in the on-chip memory for the limited memory space. Also, how to choose the right intermediate residual exactly for the summation has become a challenge as a residual will be used for multiple times. We proposed a method to control the residual storage inspired by the use of stacks in CPUs. When reaching a residual layer during the network structure read-in, the network mapper will record the residual level and how many times this residual result will be used, and a new memory space is allocated to store this residual. When reaching an add layer, the processing engine looks for the latest residual for sum operation, and the corresponding residual count will be decreased by one. The residual count counted down to zero means that the residual result is no longer needed, so the memory space will be set free for subsequent use. After finishing all the calculations in the network, memory space occupied by residuals should be cleaned up for all the residuals have been used and freed. The network mapper follows these principles to allocate the memory space for accelerator in advance. This method is also applied to the data flow branches of the intermediate supervision process between two stacked hourglass modules.

## 5 Experimental Result

The proposed accelerator architecture is implemented on the Zynq-7000 ZC706 evaluation board (XC7Z045), which contains 218,600 LUTs, 545 block RAMs and 900 DSPs. The implementation result and the performance comparison will be discussed below and then followed by a pose estimation demonstration using the proposed accelerator.

## 5.1 Implementation Result

We choose the number of PEs to be 16 and every PE is paired with an input buffer, a weight buffer and an output buffer. The size the input buffer and output buffer is same with 1536 × 256-bit each and the size of a weight buffer is 64 × 256-bit.

**Table 2.** Resource utilization of our implementation

| Module | LUT | DSP | BRAM |
|---|---|---|---|
| PE array | 22394 (10.3%) | 288 (32.0%) | 0 (0.0%) |
| Buffer | 51412 (23.5%) | 9 (1.0%) | 448 (82.2%) |
| Total | 73806 (33.8%) | 297 (33.0%) | 448 (82.2%) |

**Table 3.** Performance comparison with other implementations

| Work | [13] | Embedded CPU | Desktop CPU | GPGPU | Ours |
|---|---|---|---|---|---|
| Platform | Arm Kyro | Arm Cortex-A9 | Intel i7-6800 k | Nvidia 1080 Ti | **Zynq XC7Z045** |
| Tech node (nm) | 14 | 28 | 14 | 16 | **28** |
| Clock (MHz) | 2150 | 667 | 3400 | 1480 | **125** |
| Power (W) | – | – | 30 | 55 | **8.6** |
| Problem complexity (GOP) | 0.608 | 0.953 | 0.953 | 0.953 | **0.953** |
| Computation time (ms) | 75 | 33427 | 121.80 | 56.21 | **112.76** |
| Performance (GOP/s) | 8.107 | 0.0285 | 7.824 | 16.954 | **8.434** |

The resource utilization is shown in Table 2. The stacked hourglass algorithm needs 0.953 GOPs, mostly of them are multiplications and summations. Our system achieves an average performance of 8.434 GOP/s and the frame rate reach 8.85 fps.

Since our design is for the embedded applications, and also is the first implementation for the hourglass network as far as we know, the performance comparison will be made with the embedded CPU within the Zynq XC7Z045, which is an ARM Cortex-A9 core. We also make the comparison with the performance mentioned in [15] for the similar algorithm structure of depthwise separable convolution. Our implementation achieves the average performance of 8.434 GOP/s, which is 296× faster than the embedded CPU.

Regardless of extra memory access for the multiple resolution summation of residual blocks in stacked hourglass network, our implementation can still slightly override the performance to MobileNetV2 implemented in Kyro CPU. In addition, we also run the same hourglass network on the modern desktop CPU and GPGPU. It's surprise that our implementation does better jobs than the desktop CPU, by 1.08× and 3.76× on calculation speed and power efficiency respectively. Our works also wins 3.18× power efficiency when comparing to GPGPU. The performance of our system and different devices is shown in Table 3.

Considering the nature of the stacked hourglass network that residual results with multiple resolution are created and accumulated during the inference phase, frequent off-chip memory access seems inevitable and it's the main reason for the low utilization of PE in our work. We consider it unfair to compare our works with other works implemented on the same ZC706 which runs the holistic network structures without branches.

## 5.2 Application

We also use the proposed hardware accelerator to build an embedded system for real-time pose estimation demo on ZC706 evaluation board as well. The system is in the heterogeneous architecture. The whole stacked hourglass network is calculated within the proposed hardware accelerator while the embedded ARM Cortex-A9 dual core CPU is responsible for the control of the video capture, weights preload, image pre-processing and the HDMI display. The accelerator runs at the frequency of 125 MHz. The block diagram of the whole system is presented on Fig. 12.

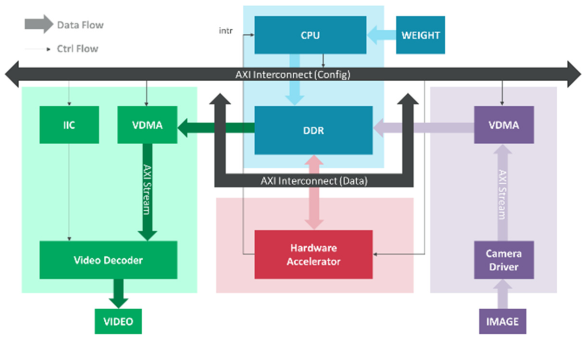

**Fig. 12.** Block diagram of the demo system.

Figure 13 shows the overview of the system and the pose estimation application demo. As shown on the monitor, the system is able to reproduce the skeleton and the key points of the human body with relatively high precision.

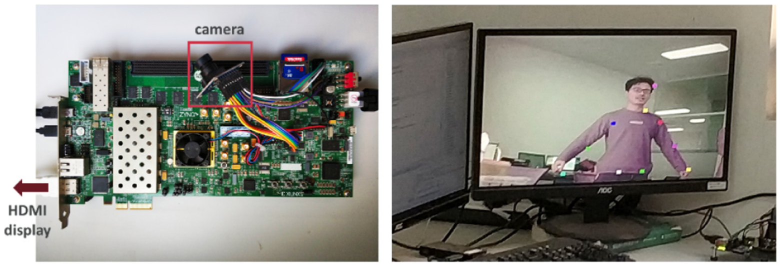

**Fig. 13.** FPGA evaluation board used for the pose estimation demo (left) and the demonstration (right).

## 6 Conclusion

In this article, a hardware accelerator implementation is purpose for the stacked hourglass network. The architecture of the accelerator is optimized for the depth separable convolutions and the multiple resolution residual summations that are frequently seen in the stacked hourglass network. With such dedicated design, high accuracy pose estimation applications can be done on the portable device. As an example, our accelerator implemented on Zynq XC7Z045 can achieve an average performance of 8.434 GOP/s with high power efficiency of 0.981 GOP/s/W under the 125 MHz working frequency. Also, a pose estimation demo is also presented here using the purpose hardware design.

## References

1. Newell, A., Yang, K., Deng, J.: Stacked Hourglass Networks for human pose estimation. arXiv:1603.06937v2 [cs. CV], July 2016
2. Chen, Y., Krishna, T., Emer, J.S., Sze, V.: Eyeriss: an energy-efficient reconfigurable accelerator for deep convolutional neural networks. IEEE J. Solid-State Circ. **52**(1), 127–138 (2017)
3. Luo, T., et al.: DaDianNao: a neural network supercomputer. IEEE Trans. Comput. **66**(1), 73–88 (2017)
4. Qiu, J., et al.: Going deeper with embedded FPGA platform for convolutional neural network. In: Proceeding of FPGA, Monterey, CA, USA, pp. 26–35 (2016)
5. Gokhale, V., Zaidy, A., Chang, A.X.M., Culurciello, E.: Snowflake: an efficient hardware accelerator for convolutional neural networks. In: Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS 2017), pp. 1–4 (2017)
6. Meloni, P., et al.: NEURAghe: exploiting CPU-FPGA synergies for efficient and flexible CNN inference acceleration on Zynq SoCs. ACM Trans. Reconfigurable Technol. Syst. **11**(3), 1–24 (2018)
7. Su, J., et al.: Neural network based reinforcement learning acceleration on fpga platforms. ACM SIGARCH Comput. Archit. News **44**(4), 68–73 (2016)
8. Guo, K., et al.: Angel-eye: a complete design flow for mapping CNN onto embedded FPGA. IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst. **37**(1), 35–47 (2018)
9. Kim, D., Ahn, J., Yoo, S.: ZeNA: zero-aware neural network accelerator. IEEE Des. Test **35**(1), 39–46 (2018)
10. Aimar, A., et al.: NullHop: a flexible convolutional neural network accelerator based on sparse representations of feature maps. IEEE Trans. Neural Netw. Learn. Syst. **30**(3), 644–656 (2019)

11. Bianco, S., Cadene, R., Celona, L., Napoletano, P.: Benchmark analysis of representative deep neural network architectures. IEEE Access **6**, 64270–64277 (2018)
12. Su, J.: Artificial neural networks acceleration on field-programmable gate arrays considering model redundancy. Imperial College London Ph.D. thesis (2018)
13. Lin, X., Yin, S., Tu, F., Liu, L., Li, X., Wei, S.: LCP: a layer clusters paralleling mapping method for accelerating inception and residual networks on FPGA. In: 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC), San Francisco, CA, pp. 1–6 (2018)
14. Ma, Y., Kim, M., Cao, Y., Vrudhula, S., Seo, J.: End-to-end scalable FPGA accelerator for deep residual networks. In: 2017 IEEE International Symposium on Circuits and Systems (ISCAS), Baltimore, MD, pp. 1–4 (2017)
15. Howard, A.G., et al.: MobileNets: efficient convolutional neural networks for mobile vision applications. arXiv:1704.04861 [cs], April 2017
16. Chollet, F.: Xception: deep learning with depthwise separable convolutions. arXiv:1610.02357v3 [cs], April.2017
17. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. arXiv:1512.03385v1 [cs. CV], December 2015
18. Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., Chen, L.: MobileNetV2: inverted residuals and linear bottlenecks. arXiv:1801.04381 [cs. CV], January 2018
19. Venkataramani, S., et al.: ScaleDeep: a scalable compute architecture for learning and evaluating deep networks. In: ISCA 2017 Proceedings of the 44th Annual International Symposium on Computer Architecture, pp. 13–26 (2017)