algorithms

Article

Model Parallelism Optimization for CNN FPGA Accelerator

Jinnan Wang [1], Weiqin Tong [1,2,*] and Xiaoli Zhi [1,2]

[1] School of Computer Engineering and Science, Shanghai University, Shanghai 200444, China
[2] Shanghai Engineering Research Center of Intelligent Computing System, Shanghai University, Shanghai 200444, China
* Correspondence: wqtong@shu.edu.cn

Abstract: Convolutional neural networks (CNNs) have made impressive achievements in image classification and object detection. For hardware with limited resources, it is not easy to achieve CNN inference with a large number of parameters without external storage. Model parallelism is an effective way to reduce resource usage by distributing CNN inference among several devices. However, parallelizing a CNN model is not easy, because CNN models have an essentially tightly-coupled structure. In this work, we propose a novel model parallelism method to decouple the CNN structure with group convolution and a new channel shuffle procedure. Our method could eliminate inter-device synchronization while reducing the memory footprint of each device. Using the proposed model parallelism method, we designed a parallel FPGA accelerator for the classic CNN model *ShuffleNet*. This accelerator was further optimized with features such as aggregate read and kernel vectorization to fully exploit the hardware-level parallelism of the FPGA. We conducted experiments with *ShuffleNet* on two FPGA boards, each of which had an Intel Arria 10 GX1150 and 16GB DDR3 memory. The experimental results showed that when using two devices, *ShuffleNet* achieved a 1.42× speed increase and reduced its memory footprint by 34%, as compared to its non-parallel counterpart, while maintaining accuracy.

Keywords: convolution neural network; model parallelism; field programmable gate array; inference accelerating

Citation: Wang, J.; Tong, W.; Zhi, X. Model Parallelism Optimization for CNN FPGA Accelerator. *Algorithms* **2023**, *16*, 110. https://doi.org/10.3390/a16020110

Academic Editors: Mircea-Bogdan Radac and Frank Werner

Received: 19 December 2022
Revised: 2 February 2023
Accepted: 6 February 2023
Published: 14 February 2023

1. Introduction

Convolutional neural networks (CNNs) are mainly used in computer vision applications, such as image classification, video recognition, and face detection [1,2]. Usually, CNN inference is a resource-consuming task, as intermediate results and weights occupy a large memory footprint, and various operations require a large amount of computation. The accuracy of CNN-based algorithms has improved significantly in the last decade due to the increased data and enhanced network structure.

A typical CNN architecture has multiple convolutional layers and classification layers. CNNs are computationally intensive, with more than a billion operations per input image, so GPUs are widely used to accelerate the training and inference tasks of CNNs [3]. However, their power consumption (>100 W) is too high for embedded applications, where energy efficiency is critical. Therefore, various hardware accelerators based on FPGAs, SoCs (CPC + FPGA), and ASICs have been proposed [4–6]. FPGA-based hardware accelerators have gained momentum due to their high reconfigurability, fast turnaround time (compared to ASICs), good performance, and better energy efficiency (compared to GPUs), especially with the availability of high-level synthesis (HLS) tools from FPGA vendors [7].

For CNNs with a large number of parameters, an FPGA card with limited resources cannot load the entire CNN model into on-chip storage without external storage (DDR), which greatly limits the computational power of the FPGA. For example, in CNN inference, when only one FPGA is used, the on-chip memory is often not sufficient to hold all the model's data. The model weights have to be stored in DRAM and need to be accessed frequently, resulting in a performance bottleneck. When there are enough FPGAs, each FPGA is responsible for one layer or several features in one layer of the model, so that the model weights needed for each FPGA are completely loaded into the on-chip memory and the performance bottleneck in DRAM is eliminated. This model parallelism is attractive because it can simultaneously optimize the latency, throughput, and memory footprint of CNN inference.

However, it is not easy to use model parallelism to assign the inference of CNN models to hardware devices, as CNN models are inherently tightly coupled structures [8]. Therefore, we focused on the CNN structure to explore model parallelism optimization for distributed CNN inference.

In this paper, we aimed to achieve more efficient model parallelism by decoupling the CNN structure. The contributions of this paper are summarized as follows:

(1) We decoupled the CNN network structure using group convolution and a new channel shuffle process to replace the original convolution and channel shuffle techniques. This loosened the connections between feature maps and provided a high efficiency and low memory usage for each device.
(2) We designed a parallel FPGA accelerator for the classic CNN model *ShuffleNet* using model parallelism. Additionally, this accelerator was optimized with several parallel strategies in the heterogeneous parallel programming framework OpenCL. This accelerator could leverage multiple devices to speed up inferencing and relieve resource constraints on the individual devices.

The rest of the paper is organized as follows. Section 2 provides a brief overview of the *ShuffleNet* structure, Intel OpenCL, and related works. Section 3 explains the modifications to the convolutional layer and channel shuffle features. Section 4 describes the optimization on OpenCL and FPGA hardware and a parallel computing architecture for CNN models. Section 5 presents the experimental results for *ShuffleNet* on the Altera FPGA platform and a comparison with prior works. The paper is concluded in Section 6 with future prospects.

2. Background and Related Works

2.1. *ShuffleNet*

ShuffleNet is an efficient CNN structure specifically designed for low-end devices such as mobile devices [9]. In order to significantly reduce the computation cost while maintaining accuracy, this new structure utilizes two operations: pointwise group convolution and channel shuffle. Experiments on ImageNet classification and MS COCO object detection illustrated that *ShuffleNet* has superior performance compared to other structures, e.g., presenting a lower top-1 error (7.8%) than the recent MobileNet structure at a computation performance of 40 MFLOPs [10].

The *ShuffleNet* used in this paper consisted of one independent convolutional layer (CL), followed by 16 blocks, ending with a fully connected layer (FC). In addition, there were three max-pooling layers with a step size of 2 and an average pooling layer. The final Softmax layer output a vector of 1000 elements, representing 1000 possible image classes. The *ShuffleNet* architecture is shown in Table 1.

Table 1. *ShuffleNet* architecture.

Layer	Output Size	KSize	Stride	Repeat	Output Channels (G Groups)				
					G = 1	G = 2	G = 3	G = 4	G = 8
Image	224 × 224	3 × 3			3	3	3	3	3
CL	112 × 112	3 × 3	2	1	24	24	24	24	24
MaxPool	56 × 56		2						
Stage2	28 × 28		2	1	144	200	240	272	384
	28 × 28		1	3	144	200	240	272	384
Stage3	14 × 14		2	1	288	400	480	544	768
	14 × 14		1	7	288	400	480	544	768
Stage4	7 × 7		2	1	576	800	960	1088	1536
	7 × 7		1	3	576	800	960	1088	1536
GlobalPool	1 × 1	7 × 7							
FC					1000	1000	1000	1000	1000

2.2. OpenCL for FPGA

Traditionally, HDL, like VHDL and Verilog, has been used to describe the design of an FPGA. However, this is a time-consuming and lengthy process that requires in-depth knowledge of the underlying hardware. To make FPGAs easier to program, FPGA vendors and the research community have been actively developing HLS tools that take high-level language design descriptions as an input and generate a synthesizable hardware implementation for FPGAs [11–13].

OpenCL is an open, cross-platform, high-level parallel programming language that can be used for GPU and FPGA development. Figure 1 summarizes the development flow of an OpenCL-based FPGA. In the framework, FPGAs (as OpenCL devices) are connected to CPUs (as OpenCL hosts) through high-speed PCIe to form heterogeneous computing systems. An OpenCL code written in a dialect of C/C++ defines multiple parallel computation units (CUs) in the form of kernel functions, which are compiled and synthesized to run on the FPGA. On the host side, the C/C++ code runs on the CPU, providing an application programming interface (API) to communicate with the kernels implemented on the FPGA. This work used the Altera OpenCL SDK toolset to compile, implement, and profile the OpenCL code on the FPGA.

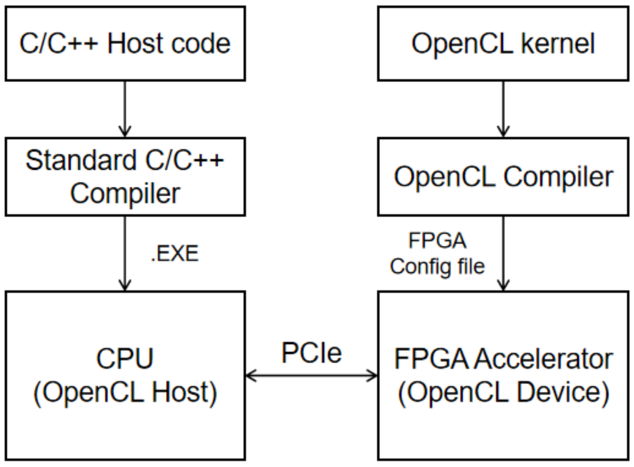

Figure 1. FPGA design flow of OpenCL-based CNN accelerator.

2.3. Related Work

Model parallelism and data parallelism are two common types of parallelism in distributed deep learning.

Model parallelism attempts to partition the learning model itself among several devices. The main works related to this paper are [14,15]. The authors of [14] designed three approaches to explore model parallelism for FCs in CNN training. The authors of [15] proposed an optimized parallel algorithm using the butterfly reduction communication strategy for CNN training in distributed GPUs. Both of these studies applied model parallelism in a fully connected layer, and neither of them considered applying model parallelism in a convolutional layer.

Data parallelism refers to the partitioning of data among several devices, which execute the same model in parallel. One representative work on data parallelism [16] proposed a local distributed mobile computing system for DNN applications involving the distribution of both the input and output as well as the weight across devices. However, it focused mainly on sparse fully connected layers and did not take weight-intensive convolutional layers into account. Furthermore, it used the biased one-dimensional partition (BODP) method that requires all devices to synchronize by exchanging data. Another important data-parallelism-based work [17] tried to split the feature map into 2D grids for convolutional layers. However, this caused overlapping computation and redundant tasks. Most data parallelism works barely pay attention to the characteristics of CNN structures and tend to ignore the cost of inter-device synchronization.

This paper focused on the model parallelism of multiple devices. This required not only reducing the communication between devices, but also optimizing the computing speed of the devices. The authors of [18] accelerated training and inference using mathematical transformations such as FFTs to reduce the multiplication numbers. The authors of [19–21] provided deep learning frameworks to efficiently exploit data-level parallelism in CNNs. The authors of [19] studied the effectiveness of various optimization strategies such as branch divergence elimination and memory vectorization. These GPU-based parallel methods can be learned.

Previous works [22–24] have proposed optimization techniques specific to FPGAs. The authors of [22] exploited layer parallelism and assembled convolutional components that could be adapted to the different layers of the common CNN. The authors of [23] provided an FPGA accelerator with a new deep-pipeline OpenCL kernel architecture and proposed data reuse and task mapping techniques to improve efficiency. The authors of [24] proposed a systematic design space exploration approach to maximize the OpenCL-based FPGA accelerator for a given CNN model. These works have one thing in common: they all made more efficient use of FPGAs by increasing their parallelism. This provided valuable experience to inform our use of openCL [11] and FPGAs [25] in this study.

This work began with considering the network structure, which is the basis for model parallelism in CNN inference. Unlike previous works, instead of using data parallelism in the convolutional layer, this study changed the structure to distribute the input and output data of the layer as well as the weight data across devices in a partitioned manner, eliminating the cost of inter-device synchronization while reducing the memory footprint of each device. Furthermore, we explored optimization strategies with an OpenCL framework for FPGA hardware to optimize deep learning inference in terms of data transmission and computational efficiency.

3. Approach

In this section, we describe in detail the decoupling methods for each layer in the CNN to effectively improve model parallelism.

3.1. Convolution

In general, the convolutional layer (CL) is the most resource-intensive layer in a CNN. In a CNN model, a CL contains hundreds of convolutional kernels (y). The convolutional operation aims to generate the output by sliding the convolutional kernel over the input. Since the convolutional kernel is a three-dimensional (width, height, depth) tensor, the core of convolution is a three-dimensional multiplicative accumulation operation. This is shown in Equation (1).

$$C_{OUT}(f_o, y, x) = \sum_{f_i=0}^{C_n} \sum_{k_Y=0}^{K} \sum_{k_x=0}^{K} W_l(f_o, f_i, k_y, k_x) \times C_{IN}(f_i, y+k_y, x+k_x) \quad (1)$$

where $C_{IN}(f_i, y+k_y, x+k_x)$ and $C_{OUT}(f_o, y, x)$ represent the neurons at location (x, y) in the input feature map f_i and the output feature map f_o, respectively; and $W_l(f_o, f_i, k_y, k_x)$ represent the weight of the corresponding position (k_x, k_y) in the nth layer of f_o obtained by convolution with f_i.

Previous works have parallelized the convolution layer through data partition. A common partitioning method for convolutional layers used in research is shown in Figure 2. Each device processes a portion of data from each input channel and generates the corresponding output. When the convolutional kernel is 1 × 1, the partitioning method in Figure 2 is reasonable. However, when the convolutional kernel exceeds 1 × 1, the data of each input channel require adjacent input data from neighboring devices, and thus communication between devices occurs. This kind of communication negatively affects model parallelism. A larger region of input data than the partitioned region itself, as shown by the red dashed box in Figure 2, has to be allocated to each device, because its computation depends on the neighboring data. To make matters worse, in this partitioning approach the kernel weights are not partitioned, and each device needs to keep all the weights, which has a slight impact on the memory footprint.

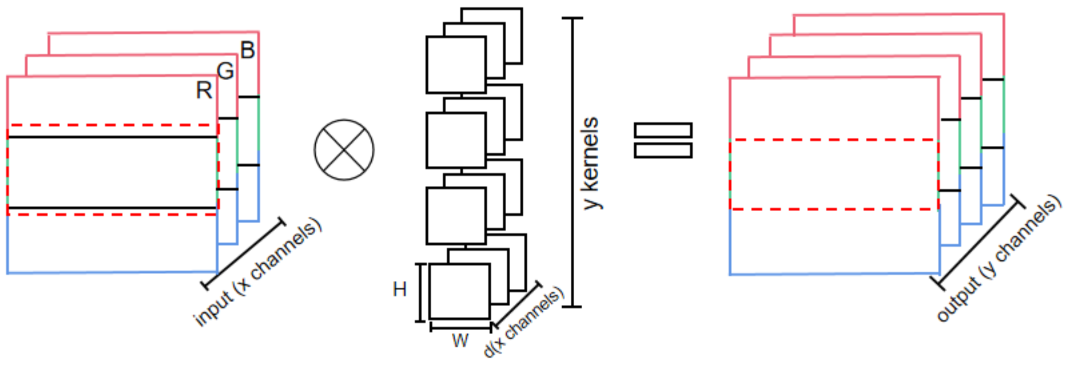

Figure 2. Partitioning method for convolutional layer (number of devices = 3).

If each device fetches the data of the region in the red dashed box in advance, the communication between devices can be eliminated. Unfortunately, this approach lacks universality, since the size of the excess depends on the size of the convolutional kernel.

To deal with the problems of the above partitioning methods, we proposed the replacement of this tightly coupled convolutional structure with group convolution in order to apply model parallelism.

For convolutional neural networks, group convolution is more suitable for model parallelism than convolutional layers. This is because group convolution first appeared in AlexNet to solve the problem of insufficient memory. It can increase the diagonal correlation between convolutional kernels and reduce the training parameters. Previous research works have indicated that group convolution can make a model less prone to overfitting, which is similar to the effect of regularization.

As shown in Figure 3, with the group convolution structure, each device acquires a part of the data of the input channel, while the weights are uniformly distributed on each device and are applicable to any convolutional kernel size. Assume that there are X feature maps (channel = X) and the number of group convolutions is M (group = M). The algorithm aims to divide the channel into M parts. Each group corresponds to X/M channels, which are connected independently. The size of the convolution kernel also changes from X*W*H to X*W*H/M. This change does not affect the dimensionality of the input and output feature maps, but significantly reduces the computational complexity and the number of model parameters and greatly reduces the memory occupation resources of each device, which is beneficial for hardware devices with limited resources.

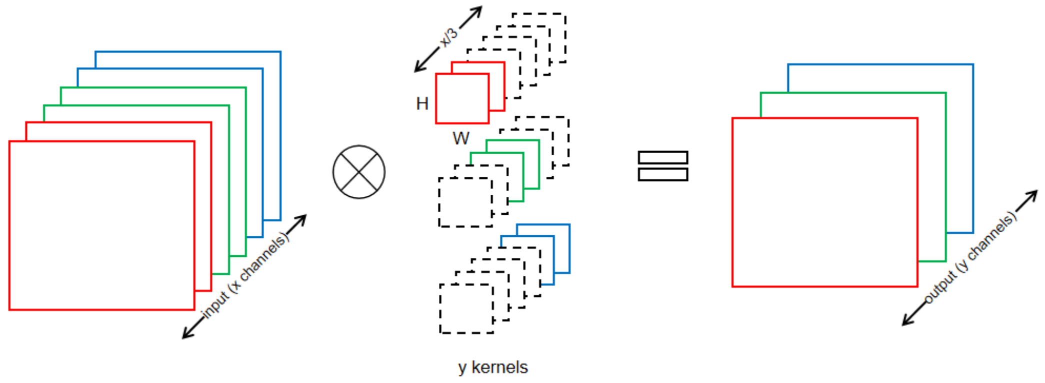

Figure 3. Group convolution (different colors represent different groups).

However, the disadvantage of using group convolution instead of convolution is very obvious: since each output channel only uses the input channels within the same group, there is no information exchange between groups, resulting in a loss of accuracy.

3.2. Inside Shuffle

To alleviate the problems mentioned above, channel shuffle was employed to compensate for the accuracy of the CNN model. Since channel shuffle requires the exchange of data between groups of channels, it imposed new synchronization points while compensating for accuracy.

When using model parallelism, some neural networks need synchronization points because their network structure inherently requires communication to remove data dependencies. For example, the *ShuffleNet* structure used in this paper comprised 16 blocks, which meant that 16 synchronization points were needed when distributing the inference of *ShuffleNet*.

Is it possible to retain the shuffle feature without adding synchronization points? For neural networks that need synchronization points, we modified the channel shuffle so that it did not add new synchronization points. The details of this process are shown in Algorithm 1.

Algorithm 1. Inside-Shuffle.

```
Input: data: output from the upper layer
       I_input: Store a portion of the data needed for each device, I_input ⊂ data
       N: a parameter for internal shuffle of data
Output: result: reordered data is used to provide to the next layer
1.Create local variable I_input
2.Read data from Channel to I_input
3.for i = 0, 1, . . . , Channel/(N * Device_Number) do
4.   for j = 0, 1, . . . , N do
5.      for k = 0, 1, . . . , input_size do
6.         Convert the one-dimensional array I_input into
               a three-dimensional array (i, j, k)
7.         Transpose(i, j)   #Tranpose(·) is used to exchange data
8.         Store result to the channel in the original order.
9.      end for
10.  end for
11.end for
```

First, it is necessary to comprehend the implementation of the original channel shuffle process. Channel shuffle is the "reorganization" of the feature map after group convolution. The data of each group are divided into n parts and exchanged with each other to ensure that the information flows between different groups.

Next, we introduce inside shuffle, as depicted in Algorithm 1, wherein each block uses residual learning, which guarantees the interactivity and protects the integrity of the information by directly bypassing the input information to the output. Its function is similar to that of channel shuffle.

In a word, the inside shuffle algorithm divides the data of each group into n parts, and intra-group exchange is used instead of inter-group exchange, so that no new synchronization points are added while the "reorganization" feature is retained.

3.3. Other Layers

Except for the convolutional layer and channel shuffle layer, CNNs contain other layers, such as BN, DWconv, and Maxpool. Looking closely at the computation of these layers, it is found that each input feature map is computed with weight parameters to obtain the output feature map, and the data of other feature maps are not needed during this period. Unlike convolution layers, the feature maps of these layers do not closely relate to each other. Therefore, it can be seen that model parallelism can be applied directly to these layers. Depending on the number of devices, the input feature maps and parameters are evenly spread out and computed on each device alone.

4. Optimizations

In this study, we developed a parallel FPGA accelerator for *ShuffleNet* and implemented it with an OpenCL framework on an FPGA. This section first describes the optimization strategies with the OpenCL framework for FPGA hardware, then outlines the overall architecture of the parallel FPGA accelerator.

4.1. Pipelined Computation

OpenCL allows the programmer to invoke the kernel in one of two configurations: NDRange and single task. For NDRange, the kernel relies on work items to partition the data, and each work item is independent. For single task, the kernel uses only one work item, allowing pipelined execution in a kernel loop. This study used single task to invoke the kernel. NDRange requires the setting of the global_work_size and local_work_size parameters to ensure that the data are reasonably distributed among the different work items, but each work item takes a different amount of time to process the data, which requires a synchronization operation. Single task uses pipelined execution, so there is no need to synchronize the execution of a single work item in succession.

4.2. Channel Communication

The OpenCL-based FPGA accelerator data flow is as follows. First, the kernel receives the data in the input buffer; then, it stores the result in the output buffer after computing; finally, the host side reads the result into memory. The whole process needs to ensure that the kernel has stored the complete data in the output buffer when the host side reads the content in the output buffer. For neural networks with several different layers, such as convolutional layers and pooling layers, the host side and the kernel side need to constantly exchange data, resulting in a significant amount of communication time. In this case, the Intel FPGA SDK for the OpenCL channel provided a flexible way to allow data to be passed from one kernel to another, reducing time-consuming interactions and enabling a "write once, compute many" programming model. Note that it was necessary to add #pragma OPENCL EXTENSION cl_intel_channels: enable before running channels to allow extensions. This was because the default behavior of channels is blocking, and the cl_intel_channels extension provided a way to access data stored in multiple channels with a single instruction. This allowed for the more efficient accessing and processing of data stored in multiple channels.

4.3. Aggregate Read

Memory access times refers to the number of read and write operations performed using the global memory during the execution of the kernel programs. If the access to the global memory is too frequent, it will cause considerable performance loss. Aggregate access can effectively reduce the number of access occasions. The input data should be reorganized so that they are read in memory storage order, e.g., a[0], a[1], a[2], a[3] instead of a[0], a[7], a[1], a[8]. This reduces the number of cache misses and makes good use of the data locality, thus effectively increasing the speed of memory access. For output data, this aggregated read technique is also applicable.

4.4. Kernel Vectorization

A possible method to further improve the throughput is kernel vectorization, which works in an SIMD manner, with the compiler completing the vectorization by memory aggregation. Before memory aggregation, multiple read operations access the same storage area, which requires more complex control logic, generates access conflicts, and requires four accesses to complete the entire fetch operation. After memory aggregation, only one read operation is required, which is equivalent to turning multiple read operations into one, improving the efficiency of data reads. Although the compiler automatically coalesces memory accesses, this is not always accurate. Thus, the programmer is required to use explicit vectorized access operations as much as possible to achieve better access performance. Note that the number of vectorizations can only be 2, 4, 8, or 16.

4.5. Overall Architecture

Based on the above optimization strategies, an overall architecture could be designed. In the process of inference, interactions between the kernels of each layer were transmitted through channels, data were read by aggregation, and kernels were vectorized. In this design, the data transfer process overlapped with the parallel execution of the kernels on the FPGA, which avoided the impact on the FPGA computational performance. The overall architecture of design is shown in Figure 4. The overall architecture was composed of one CPU, two FPGAs, and DDR3 memory. The CPU played the role of synchronizing the data in the middle of the multi-FPGA devices during CNN inference.

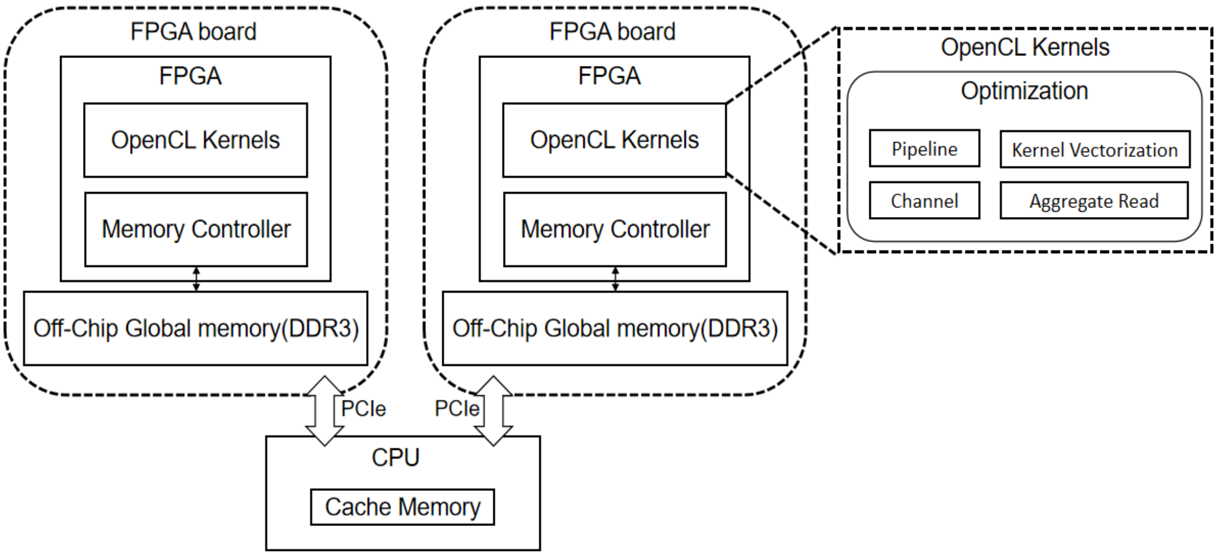

Figure 4. Model parallel computing architecture.

The proposed FPGA internal design is shown in Figure 5, including the design of each module and the data flow control. MemRD was used to obtain the image data and weights from the host side. MemWR was used to write the computation results from the kernel side to the host side. Block represents the computation unit corresponding to the CNN. Each Block computation ended with residual learning, which needed to be written back to the host side via MemWR for synchronization. In addition, the on-chip memory was implemented using M20K memory blocks, each with a 20K bits capacity. Only when the memory depth was greater than the maximum depth of one M20K block was another M20K block added.

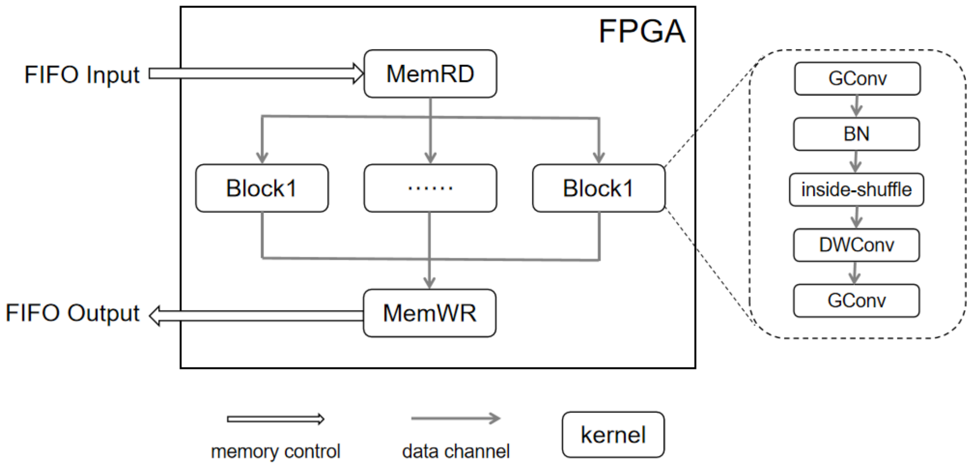

Figure 5. Proposed FPGA internal design.

5. Experiment

5.1. Experimental Setup

The FPGA chips selected for this experiment were two Intel Arria10 GX1150 FPGAs embedded in an Inspur F10A board. The software environment was Intel FPGA SDK for OpenCL pro19.1 with a BSP provided by the Wave FPGA development team, using an Intel® Xeon® Gold6128 CPU as the OpenCL host and DMA for data transmission between devices. The board support package (BSP) was a layer between the motherboard hardware and the driver in the operating system. It provided a functional package for the upper layer drivers to access the hardware device registers and make them work better on the hardware motherboard.

We selected two datasets (MNIST and CIFAR-10) to verify the accuracy and performance of the inside shuffle algorithm. These two datasets comprised 10,000 images for testing, and the rest of the images were used for training. On this basis, we used the *ShuffleNet* model to verify the effectiveness of the model parallelism, evaluating it in terms of resource utilization, speed-up ratio, and energy consumption.

5.2. Accuracy

In this study, we first evaluated the accuracy loss of the inside shuffle algorithm to verify that it fell within acceptable limits. We chose the lightweight neural network *ShuffleNet* and ensured that the training and test sets used in each version were the same for the fairness of the experiment. The resulting data are shown in Table 2. For both the MNIST and CIFAR-10 datasets, the top-1 accuracy of the inside shuffle algorithm was slightly higher than the original shuffle algorithm whether the group was equal to 2 or 3. Overall, the top-1 accuracy of each version remained basically the same.

Table 2. Accuracy of experimental results.

CNN Model	Channel Shuffle	Group	Top-1 Accuracy	
			MNIST	CIFAR-10
ShuffleNet	Original Shuffle	2	98.90%	84.44%
	Inside Shuffle	2	99.00%	84.75%
	Original Shuffle	3	98.70%	83.52%
	Inside Shuffle	3	98.83%	84.11%

Compared with original shuffle, inside shuffle eliminated the synchronization points caused by the model parallelism while maintaining the original shuffle characteristics. This not only allowed the exchanging of data, but also further exposed the high degree of parallelism.

According to the results of the two datasets, based on the understanding of the original shuffle algorithm, the inside shuffle algorithm did not have much impact on the accuracy, which verified the effectiveness of inside shuffle optimization.

5.3. Performance

This section presents the evaluation of the performance of the model in terms of time and energy consumption. Table 3 shows the experimental results for both versions of *ShuffleNet*.

Table 3. Experimental results.

Device	Version	Optimizations		Number	Time (ms)	Power (W)
		Channel	Others			
Inspur F10A	*ShuffleNet*	X	X	1	1045.198	23.5
		X	√	1	329.869	23.4
		X	√	2	228.358	20.5
	I-ShuffleNet	√	√	1	186.635	23.5
		√	√	2	130.521	21.1

In Table 3, *ShuffleNet* is the original model, while *I-ShuffleNet* is the model with the inside shuffle algorithm. The *optimizations* are divided into *channel* and *others*. Note that *others* represents three optimization methods: pipelined computation, aggregate read, and kernel vectorization. *Number* is the number of FPGA devices. *Time* is the average inference time in milliseconds for many experiments, and *Power* is the average power consumption in watts for each device during inference.

The power consumption of the FPGA includes not only the FPGA chip, but also the power consumption of the off-chip memory and other devices on the board.

On Inspur F10A, compared to *ShuffleNet*, *I-ShuffleNet* reduced the inference time by almost two-fold for the same number of devices. This was due to the introduction of the channel, which reduced the number of data interactions between the FPGA and the CPU. When there was only one device, the channel achieved the highest time reduction. In the case of a single device, the number of interactions was only two: one write and one write out. In the case of multiple devices, due to the *ShuffleNet* network structure, data synchronization was required after each block was calculated, so the number of interactions was equal to the number of blocks, i.e., 16. This is why the degree of time reduction was not very large.

As presented in Figure 6, the time comparison showed that the model parallelism had a good acceleration effect when using two FPGAs. The running time required for each layer basically demonstrated a linearly proportional reduction, achieving a speed-up ratio of 1.9×. However, in the *I-ShuffleNet* version, the model with two devices was 1.42× faster than the model with one device. Because model parallelism requires synchronization time, it is not linearly proportional.

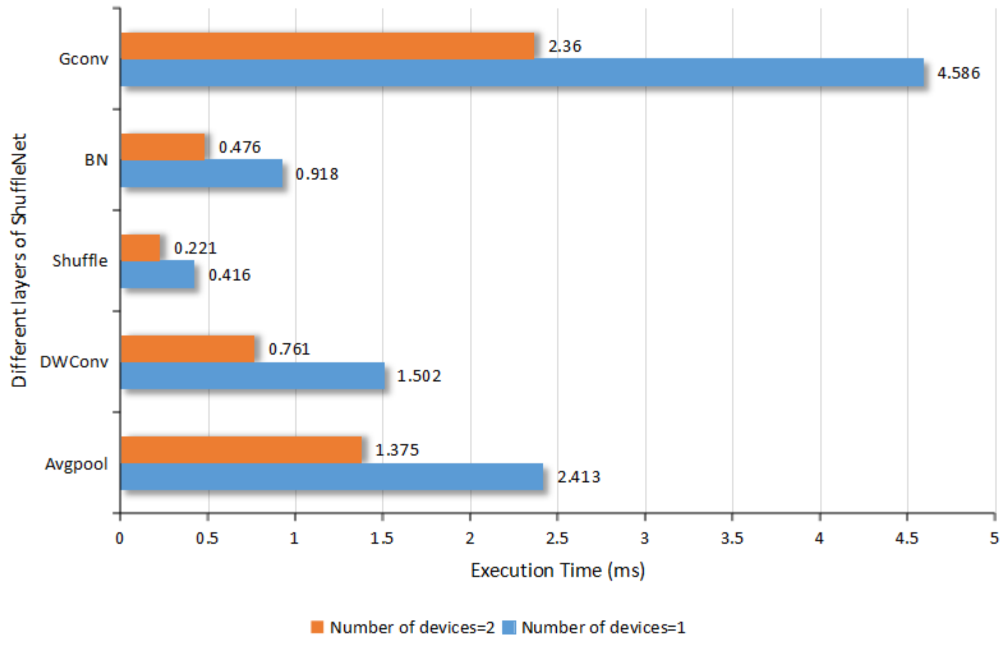

Figure 6. Comparison of computing time of different layers of *I-ShuffleNet*.

The resources used for the entire network inference process are shown in Table 4. The largest change due to the number of devices was in the use of RAMs. The number of RAMs required ranged from 1707 to 1265, with a 34% reduction in memory utilization. This reduction in memory utilization opens up the possibility of implementing neural network inference using devices with lower amounts of resources.

Table 4. Comparison of resource utilization.

Number of FPGA Devices	ALUTs	FFs	RAMs	DSPs
1	139,651 (16%)	224,468 (13%)	1707 (63%)	137 (9%)
2	148,715 (17%)	249,448 (15%)	1265 (47%)	143 (9%)

This paper demonstrated the experimental effect of CNN model parallelism when using two FPGAs. Of course, this method could be used to add more devices and act on multiple devices. However, it is not the case that more devices bring better results. In terms of the speed-up ratio, an increase in the number of devices was able to reduce the network layer computation time in a nearly linear proportion, but the time for device synchronization could not be reduced. Thus, as the number of devices increases, the key to the speed-up ratio will change from the network computation time to the device synchronization time, which will make the speed-up ratio reach a bottleneck. The same is true for accuracy. An increase in the number of devices will lead to excessively detailed data exchange within the channel, which may achieve the opposite effect. It is expected that the experimental results are most appropriate when the number of devices is 3–4.

6. Conclusions

For hardware with limited resources, model parallelism for multiple devices is often employed. However, due to the structural characteristics of tightly coupled CNNs, it is difficult to use model parallelism methods to achieve a high degree of parallelism in multi-device inference.

In this paper, we proposed an effective decoupling method using group convolution and a new channel shuffle algorithm to replace the original convolution and channel shuffle approaches, aiming to achieve data and model separation while eliminating inter-device synchronization costs. As the number of devices increased, the computation time decreased in an approximately linear proportion.

Using the proposed model parallelism method, we designed a parallel FPGA accelerator for *ShuffleNet*. We realized FPGA acceleration by pipelined computation, a channel mechanism, aggregate read, and kernel vectorization. Benefiting from these approaches, the experimental results showed that our paralleled *ShuffleNet* FPGA accelerator exhibited a high degree of model parallelism while maintaining accuracy. When using two FPGAs, it achieved a 1.42× higher speed and a power consumption of about 20 W, as well as a 34% memory footprint reduction.

In subsequent research, we will study the direct communication between FPGAs to reduce the communications delay and further optimize the model parallelism of FPGA accelerators.

Author Contributions: J.W., W.T. and X.Z. conceived the idea, designed and performed the experiments, and analyzed the results. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the Chinese Universities Industry-University-Research Innovation Fund under grant no. 2020HYA02011, in part by the Natural Science Foundation of Shandong Province under grant no. ZR2019LZH002 and Shanghai science and technology committee under grant no. 22511106000.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations were used in this manuscript:

CPU	Central processing unit
FPGA	Field-programmable gate array
GPU	Graphics processing unit
DDR2	Double data rate 2
ASCI	Application-specific integrated circuit
HLS	High-level synthesis
DRAM	Dynamic random-access memory
MFLOPs	Million floating-point operations per second
FC	Fully connected layer
CL	Convolutional layer
HDL	Hardware description language
VHDL	Very-high-speed integrated circuit hardware description language
SDK	Software development kit
DMA	Direct memory access

References

1. Li, Z.; Liu, F.; Yang, W.; Peng, S.; Zhou, J. A survey of Convolutional Neural Networks: Analysis, applications, and prospects. *IEEE Trans. Neural Netw. Learn. Syst.* **2021**, *32*, 6999–7019. [CrossRef] [PubMed]
2. Ma, Y.; Huang, C. Facial expression recognition based on deep learning and attention mechanism. In Proceedings of the 2021 3rd International Conference on Advanced Information Science and System (AISS 2021), Sanya, China, 26–28 November 2021; Volume 2021, pp. 30–35. [CrossRef]
3. Sang, H.; Xiang, L.; Chen, S.; Chen, B.; Yan, L. Image Recognition Based on Multiscale Pooling Deep Convolution Neural Networks. *Complexity* **2020**, *2020*, 6180317. [CrossRef]
4. Zhang, C.; Li, P.; Sun, G.; Guan, Y.; Xiao, B.; Cong, J. Optimizing FPGA-based accelerator design for deep convolutional Neural Networks. In Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2015. [CrossRef]
5. Gokhale, V.; Jin, J.; Dundar, A.; Martini, B.; Culurciello, E. A 240 G-ops/S mobile coprocessor for Deep Neural Networks. In Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops, Columbus, OH, USA, 23–28 June 2014. [CrossRef]

6. Chen, Y.; Luo, T.; Liu, S.; Zhang, S.; He, L.; Wang, J.; Li, L.; Chen, T.; Xu, Z.; Sun, N.; et al. Dadiannao: A machine-learning supercomputer. In Proceedings of the 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, Cambridge, UK, 13–17 December 2014. [CrossRef]
7. Munshi, A. The opencl specification. In Proceedings of the 2009 IEEE Hot Chips 21 Symposium (HCS), Stanford, CA, USA, 23–25 August 2009. [CrossRef]
8. Du, J.; Zhu, X.; Shen, M.; Du, Y.; Lu, Y.; Xiao, N.; Liao, X. Model parallelism optimization for distributed inference via decoupled CNN structure. *IEEE Trans. Parallel Distrib. Syst.* **2021**, *32*, 1665–1676. [CrossRef]
9. Zhang, X.; Zhou, X.; Lin, M.; Sun, J. *ShuffleNet*: An extremely efficient convolutional neural network for mobile devices. In Proceedings of the 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA, 18–22 June 2018. [CrossRef]
10. Howard, A.G.; Zhu, M.; Chen, B.; Kalenichenko, D.; Wang, W.; Weyand, T.; Andreetto, M.; Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv* **2017**, arXiv:1704.04861.
11. Intel. Intel FPGA SDK for Opencl Pro Edition: Programming Guide. Available online: https://www.mouser.com/datasheet/2/612/aocl_programming_guide-1301807.pdf (accessed on 23 August 2022).
12. Xilinx. Vivado High Level Synthesis. Available online: https://www.xilinx.com/video/hardware/vivado-high-level-synthesis.html (accessed on 23 August 2022).
13. Canis, A.; Choi, J.; Fort, B.; Lian, R.; Huang, Q.; Calagar, N.; Gort, M.; Qin, J.J.; Aldham, M.; Czajkowski, T.; et al. From software to accelerators with LegUp high-level synthesis. In Proceedings of the 2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES), Montreal, QC, Canada, 29 September–4 October 2013. [CrossRef]
14. Krizhevsky, A. One weird trick for parallelizing convolutional neural networks. *arXiv* **2014**, arXiv:1404.5997.
15. Jin, H.; Qamar, S.; Zheng, R.; Ahmad, P. Single binding of data and model parallelisms to parallelize convolutional neural networks through multiple machines. *J. Intell. Fuzzy Syst.* **2018**, *35*, 5449–5466. [CrossRef]
16. Mao, J.; Chen, X.; Nixon, K.W.; Krieger, C.; Chen, Y. MoDNN: Local Distributed Mobile Computing System for Deep Neural Network. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*; IEEE: Piscataway, NJ, USA, 2017. [CrossRef]
17. Zhao, Z.; Barijough, K.M.; Gerstlauer, A. DeepThings: Distributed Adaptive Deep Learning Inference on resource-constrained IOT edge clusters. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2018**, *37*, 2348–2359. [CrossRef]
18. Michael, M.; Mikael, H.; Yann, L.C. Fast training of convolutional networks through ffts. *arXiv* **2013**, arXiv:1312.5851.
19. Kang, D.; Kim, E.; Bae, I.; Egger, B.; Ha, S. C-good. In Proceedings of the International Conference on Computer-Aided Design (ICCAD), San Diego, CA, USA, 5–8 November 2018. [CrossRef]
20. Huynh, L.N.; Balan, R.K.; Lee, Y. DeepSense. In Proceedings of the 2016 Workshop on Wearable Systems and Applications—WearSys '16, Singapore, 30 June 2016. [CrossRef]
21. Huynh, L.N.; Lee, Y.; Balan, R.K. Deepmon. In Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services, Niagara Falls, NY, USA, 19–23 June 2017. [CrossRef]
22. Liu, Z.; Dou, Y.; Jiang, J.; Xu, J.; Li, S.; Zhou, Y.; Xu, Y. Throughput-optimized FPGA accelerator for deep convolutional neural networks. *ACM Trans. Reconfigurable Technol. Syst.* **2017**, *10*, 17. [CrossRef]
23. Wang, D.; An, J.; Ke, X. PipeCNN: An OpenCL-Based FPGA Accelerator for Large-Scale Convolution Neuron Networks. *arXiv* **2016**, arXiv:1611.02450.
24. Suda, N.; Chandra, V.; Dasika, G.; Mohanty, A.; Ma, Y.; Vrudhula, S.; Seo, J.S.; Cao, Y. Throughput-optimized opencl-based FPGA accelerator for large-scale convolutional neural networks. In Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 21–23 February 2016. [CrossRef]
25. Aydonat, U.; O'Connell, S.; Capalija, D.; Ling, A.C.; Chiu, G.R. An opencl™ deep learning accelerator on arria 10. In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2017. [CrossRef]