RESEARCH ARTICLE

Research on OpenCL optimization for FPGA deep learning application

Shuo Zhang, Yanxia Wu*, Chaoguang Men, Hongtao He, Kai Liang

College of Computer Science and Technology, Harbin Engineering University, Harbin 150001, China

* wuyanxia@hrbeu.edu.cn

Abstract

In recent years, with the development of computer science, deep learning is held as competent enough to solve the problem of inference and learning in high dimensional space. Therefore, it has received unprecedented attention from both the academia and the business community. Compared with CPU/GPU, FPGA has attracted much attention for its high-energy efficiency, short development cycle and reconfigurability in the aspect of deep learning algorithm. However, because of the limited research on OpenCL optimization on FPGA of deep learning algorithms, OpenCL tools and models applied to CPU/GPU cannot be directly used on FPGA. This makes it difficult for software programmers to use FPGA when implementing deep learning algorithms for a rewarding performance. To solve this problem, this paper proposed an OpenCL computational model based on FPGA template architecture to optimize the time-consuming convolution layer in deep learning. The comparison between the program applying the computational model and the corresponding optimization program provided by Xilinx indicates that the former is 8-40 times higher than the latter in terms of performance.

Citation: Zhang S, Wu Y, Men C, He H, Liang K (2019) Research on OpenCL optimization for FPGA deep learning application. PLoS ONE 14(10): e0222984. https://doi.org/10.1371/journal.pone.0222984

Editor: Wajid Mumtaz, National University of Sciences and Technology, PAKISTAN

Received: November 7, 2018

Accepted: September 11, 2019

Published: October 10, 2019

Data Availability Statement: All relevant data are within the paper and its Supporting Information files.

Funding: This work was supported by the National Key Research and Development Program (2016YFB1000400), the Harbin Outstanding Young Talents Fund (2017RAYXJ016), Nature Scientific Foundation of Heilongjiang Province (F2018008).

Competing interests: The authors have declared that no competing interests exist.

Introduction

Recently, artificial intelligence technology has attracted worldwide attention. As a new field of artificial intelligence, deep learning has an excellent strength to solve complex learning problems [1] [2]. However, with the progressive innovation of technology, the number of neural network models also increases rapidly. During the period from 2012 to 2018, as the number of neural network models increased, both the amount of model parameters and calculation increased rapidly. The size of the AlexNet model designed in 2012 and VGG-16 model designed in 2014 exceeded 200MB and 500MB respectively [3]. Meanwhile, the model parameters have increased from 60 million to 138 million. Hundreds of millions operations are required for each run. In order to improve performance, scholars have turned to designing more efficient deep neural networks [4].

In accelerating the application of deep learning, FPGA has attracted a lot of attention due to its advantages over GPU and ASIC. Compared with GPU, the acceleration design of FPGA is hardware design. Its power consumption is lower than GPU. The acceleration of FPGA can achieve higher performance under per power consumption. For example, in the reasoning stage of convolutional neural network, Microsoft team uses FPGA (Stratix V D5) to achieve the acceleration performance of 134 pictures processing per second and the power consumption is only 25 watts. If the superior FPGA (Arria 10 GX1150) is used, this acceleration performance is expected to 233 pictures processing per second, while the power consumption is basically unchanged. For high-performance GPU implementation (Caffe + cuDNN), the acceleration performance is 500-824 pictures processing per second, and the power consumption is 235 watts [5]. It means that FPGA has better energy efficiency compared with GPU [6] [7] [8] [9]. Unlike GPU and ASIC with fixed hardware architectures, FPGA is reconfigurable hardware, which means developers can connect the logical blocks within the FPGA through programmable connections to achieve their desired function [10]. This programmability enables developers to adjust their hardware design at any time according to the deep learning algorithm. However, hardware acceleration design based on FPGA requires software developers have a certain amount of hardware expertise, which is a high threshold for them. In recent years, FPGA programming environment has been greatly improved. Until now, the developers without corresponding hardware expertise have been allowed to develop FPGA with advanced programming languages such as C, C++ and OpenCL. It to some extent reduces the difficulty of FPGA development, shortens the FPGA development cycle and provides convenience for researchers and developers [11]. In order to reduce the difficulty of FPGA development, the key technologies in the automated high level synthesis tool chain are studied. These researches can be easily classified from different perspectives. From the perspective of the input language used by the user, it can be divided into C language and C–like language. The research uses C/C++ as its input language [12] [13] [14], this kind of research is divided into two categories when implementing automated generation of FPGA hardware architecture. One category of research is a complete automated generation tool chain. The process of generating hardware architecture is completely controllable, but the disadvantage is the insufficient universality of tools [14]. The other is to use the current mainstream hardware generation high level synthesis tool chain [12] [13], but it need to study the automation code generator in depth. The C/C++ language is translated by users to generate the input language supported by the commercial tool chain. The main research of this category is how to map one high-level language to another high-level language (such as OpenCL). Another kind of research work that directly uses C–like language (such as OpenCL) as an input language, focuses on different architecture of the CNN Accelerator [15] [16] [17]. However, because the same function of the program is implemented in different OpenCL code, the hardware architecture generated by the automation tool optimization is different. To implement efficient hardware circuits, developers need to constantly try to optimize various configuration combinations. Even though the push-button automated tool also requires iterative optimization. The key innovation point of this paper is proposed a computational model to help software engineers rationally design parameters without additional third-party tools, how to quickly reduce the iteratively written OpenCL code, and generate efficient hardware based on deeply loop pipelined architecture.

Convolutional neural network

As is often used in image processing, convolutional neural network is one of the most classical models in deep learning. Given an image and using filtering to extract features, the machine will obtain an image called feature map [17]. The most commonly used Valid convolution assumes that the input feature is one dimension and the filter is one dimension, there are Eq (1):

$$\begin{cases} y = con(x, w,'valid' = (y(1), \cdots, y(t), \cdots, y(n-m+1)) \in R^{n-m+1} \\ y(t) = \sum_{i=1}^{m} x(t+i-1)w(i) \end{cases} \tag{1}$$

In this equation, t = 1,2,. . .,n-m+1, and $n > m$.

In addition to CNN, most of the neurons in the neural network layer are fully connected. Although full-connected neurons can recognize more complex images, there are also some problems such as lack of flexibility and computational complexity. Convolution operation can reduce unnecessary weight connections to make the transformed images more robust.

The pooling layer is also called the subsampling layer. It usually connects with the convolution layer. Through partial correlation principle, on the one hand, it can improve the robustness of the system, and on the other hand, it can reduce the calculation of the characteristic pattern.

The propagation process in the maximum pooling layer is as Eq (2):

$$y_{ij}^{(k)} = \max(a_{(l_1 i+s)(l_2 j+t)}^{(k)}) \tag{2}$$

In the formula, L_1 and L_2 represent the core pool size. For pooling layer and convolution layer, we tend to pool after convolution and put an activation function after convolution. The activation function is a simple non-linear operation, which improves the ability of non-linear characterization. With the process of convolution-activation function-pooling, CNN can obtain more robust features.

Deep learning convolutional neural networks bring is that the convolution layer needs to consume a lot of memory [18], especially in the training process, because back-propagation needs all the intermediate values of forward transmission. If the size of the input image is $H \times W$ and the filter size is $m \times n$, the convolution can be expressed in the Eq (3):

$$z_{ij}^{(k)} = \sum_{s=0}^{m-1}\sum_{t=0}^{n-1} w_{st}^{(k)} x(i+s)(j+t) \tag{3}$$

In the equation, w is the weight of the kernel. However, the equation above is not enough with multiple convolution layers considered. Thus, a parameter is added to the kernel. The modified equation is as Eq (4):

$$z_{ij}^{(k)} = \sum_{c}\sum_{s=0}^{m-1}\sum_{t=1}^{n-1} w_{st}^{(k,c)} x_{(i+s)(j+t)}^{(c)} \tag{4}$$

In the equation, c represents the image channel. If the number of kernels is k and the channel is c, the convolution image size is $(M - m + 1) \times (N - n + 1)$ through the above equation. Assuming that the size of the convolution kernel is 5×5, 200 feature map with the size of 150×100 are needed to output. If the input is three channels, the whole process requires 225 million floating point multiplication. This process involves a large number of multiplication addition calculations, which requires a reasonable calculation computational model to improve the performance of the system. However, in the actual optimization, we should not only consider the optimization of computation, but also whether the storage resources on the FPGA chip can transmit the data needed for multiplication addition calculation at one time [19]. Assuming that ThroughputRate is the throughput of the system, it is affected by two aspects of computation and memory access. The relationship between system throughput and computation and memory access is shown in Eq (5):

$$ThroughputRate = \min(CalculatedPeak, MemoryPeak) \quad (5)$$

CalculatedPeak is the peak of computing power of calculation resources, and MemoryPeak is the maximum floating point performance of the memory support. According to the above equation, the overall throughput of the system is less than the minimum value [20] of two items of calculation and memory access. The execution of the computation requires data support. The original data convolution layer computation requires are usually copied from Host memory to FPGA off-chip global memory by the Host program. When the FPGA use that data, it usually needs to be read from the off-chip global memory. Besides, the data generated after the execution of the FPGA also need to be written back to the Host memory through the off-chip global memory. However, FPGA off-chip global memory is not on the FPGA chip, and FPGA usually takes much time to make data interaction with it. If no optimization is carried out, it usually has a great impact on the performance of the program [21]. At the same time, convolution calculation involves a relatively large amount of data, and a lot of data also need to be reused in calculation, which results in low computational efficiency. The paper [22] proposed a parallel acceleration strategy of CNN based on FPGA with OpenCL by the use of Xilinx SDAccel. But there is no optimization details and method to configure the parameters, it is difficult for researchers to reproduce. To deal with the above problems, this paper proposes an OpenCL optimization strategy with an effective solution to the low memory efficiency and the extra overhead generated by repeatedly read / written data from FPGA out of memory.

Loop pipelined architecture

The use of OpenCL tools to implement deep learning algorithms on FPGA greatly reduces the work of designers. For the process of the internal hardware architecture mapping of the FPGA is not considered, the development threshold is reduced. However, the process of converting OpenCL into FPGA bitstream through development tools is transparent to the designers, for whom it is hard to add into the project better hardware modules in other languages. In addition, most designers do not know the way to configure the bit width reasonably and the way improve the parallelism to make full use of the advantages of FPGA and the effects of data transfer on performance [23]. As a result, the deep learning algorithm designed by those designers has no obvious advantage in performance. In this section, an loop pipelined-architecture template is proposed for the optimization of convolutional neural network performance. By using the optimized architecture template given in this section and the configuration of specific technical parameters, it is verified by experiments that the performance of the algorithm can be improved effectively in the accelerated design of FPGA deep learning algorithm. The following section gives a detailed description of the optimized template architecture and configuration of technical parameters.

Loop pipelined architecture abstraction

Convolutional neural network algorithm involves a large number of computations. By making full use of the parallel characteristics of FPGA, convolution algorithm based on OpenCL on FPGA can be utilized to the advantages of FPGA. An optimized architecture template is proposed of OpenCL based on FPGA, which is shown in Fig 1.

Template parameters

According to the parameterized optimization architecture diagram, the following parameters are needed for calculation and determination:

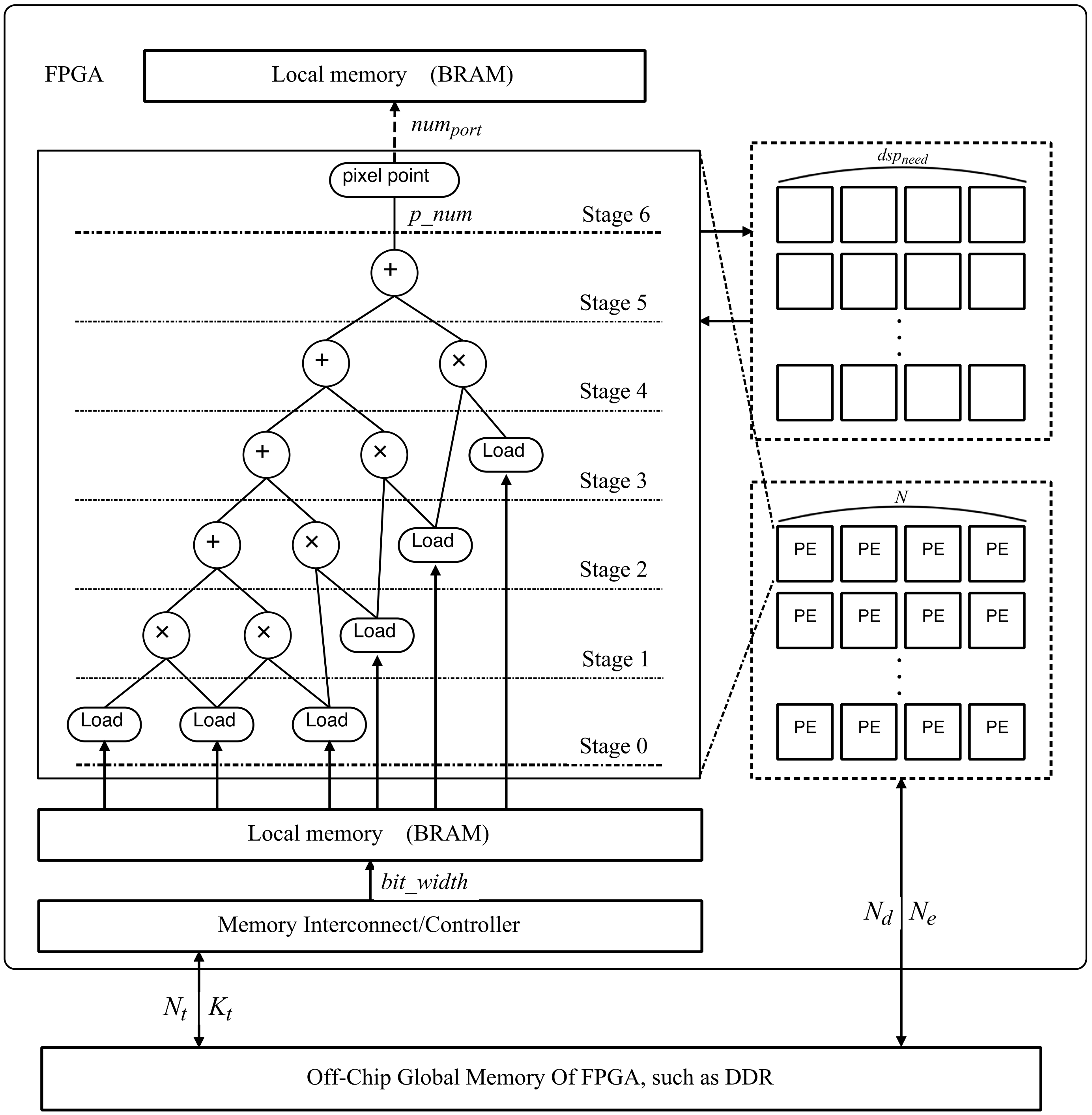


Fig 1. Hardware architecture template diagram.

https://doi.org/10.1371/journal.pone.0222984.g001

- Determining the parameter data N_d, the number of elements in the vector data type is N_e.

$$N_d = K_n \cdot N_e \tag{6}$$

- Determining the width of the data port *bit_width*.

$$bit\_width = \begin{cases} num\_bit, num\_bit \in \{32, 64, 128, 256, 512\} \\ default, num\_bit \notin \{32, 64, 128, 256, 512\} \end{cases} \tag{7}$$

  In the equation, *nun_bit* is the number of data digits corresponding to the data type.

- The theoretical value of the total number of data transfer times is N_t, and the average data amount of each data transfer is K_t.

$$N_t = \sum_{i=1}^{n} \left\lceil \frac{N_i}{Bu_i} \right\rceil \tag{8}$$

$$K_t = \frac{\sum_{i=1}^{n} N_i B_i}{N_t \cdot 8 \cdot 1000} \tag{9}$$

  In the equation, N_i is the total number of per variable data transfer and Bi is the data bits.

- The number of DSP needed for the calculation is *dsp_need*.

$$dsp_{need} = N_{unroll} \cdot K \tag{10}$$

  In the equation, N_{unroll} is the unrolling loop degree in the unrolling loop scheme; K is the number of using DSP for each loop iteration. This value can be obtained from the resource use report. The number of DSP required for each multiplication operation can be obtained through the related development board documents, and then K also can be calculated.

- Calculate the number of sub parts of the array data memory, *p_num*.

$$p_{num=} \begin{cases} \min(num, num_{\max}), loop\ \ partition \\ \min\left(\frac{num\_d}{addr\_i}, num_{\max}\right), block\ \ partition \end{cases} \tag{11}$$

  In the equation, *num_d* is the total number of data in the array. num is the number of consecutive data addresses for each calculation. *num_max* is the upper limit of the array partition supported by the compiler. *addr_i* is the address interval of each of the adjacent data.

- The cyclic boundary L_b.

$$L_b = \begin{cases} x, cyclic\ \ boundary\ \ is\ \ variable \\ c, cyclic\ \ boundary\ \ is\ \ constant \end{cases} \tag{12}$$

- Determine the number of the configuration computing unit, *N*.

$$N < \min\left(\frac{1}{B}, \frac{1}{D}, N_k\right) \tag{13}$$

In the equation, B and D are the percentages of BRAM resources consumed by a single cell and the percentage of DSP resources consumed by a single cell respectively, and N_k is the number of cells restricted by the compiler.

- The number of memory ports for storing data is numport, and the theoretical parallelism of the data calculation is *v_cal*. The maximum parallelism of data reading/writing is *v_data*, and these three parameters can be obtained from the program execution information.

Computational model

1. The relevant parameters are obtained by calculation out of hardware architecture template drawing and parameter calculation equation. The specific OpenCL optimization technology and related parameters are selected by the following algorithm steps.

(1) Judge whether the address stored in the off-chip global memory of the corresponding parameter is continuous. If so, entered (2); if not, the data vectorization is not optimized.

(2) Judge whether the data quantity contained in parameter is suitable for data vectorization. N_e represents the number of elements in vector data type supported by OpenCL compiler. $N_e \in \{2, 3, 4, 8, 16\}$. Traverse the value of N_e. If $\exists K_n \in N^+$ can make N_d and N_e satisfy the formula (1), the value of K_n is recorded. After the traversal is completed, the whole value of K_n recorded is combined into a set called L. If the set L is not empty, enter (3). If the set is empty, the data vectorization optimization is not carried out.

(3) The minimum value in the set is recorded as L_{min}. The data are grouped in ascending order of size, and the number of groups is L_{min}. The number of data in each group is N_d/L_{min}. After completing the grouping, each group of data are used as a whole to replace the original data in kernel. If they can be substituted equivalently and do not affect the correct execution of the program, they enter (4). Otherwise, remove the L_{min} from the set L and repeat (3).

(4) Carry out data vectorization optimization. N_d/L_{min} represents the number of elements contained in vector type data.

2. Configure the number of data ports and the bit width. The bit width of a data port is usually related to the data type of data transmitted through the port. At present, the OpenCL compiler supports a bit width of 32, 64, 128, 256 and 512 bits. If the data type corresponding to the data bit $num\_bit \in \{32, 64, 128, 256, 512\}$, the port bit width is set as *num_bit*. Otherwise, keep the default setting. By default, the OpenCL compiler automatically configures the bit width of the data port according to the actual situation.

3. Assuming that there are n global variables involved in the data transfer(read), the total number of data transfers (read) per variable is $N_1, N_2, \cdots, N_n$. The data digits are $B_1, B_2, \cdots, B_n$, and the burst length of data read and write is $Bu_1, Bu_2, \cdots, Bu_n$. The burst length of burst read-write model is usually 16, and the length of non-burst read-write model is 1. According to the procedure execution report, it is judged whether the new optimization is carried out. The steps are as follows:

(1) Record the total number of data transfers (reading/writing) in the program execution report and the average amount of data (reading/writing) per data transfer. The values are N_r, N_w, K_r, K_w respectively.

(2) According to the Eqs (3) and (4), calculate the total number of data transfers (reading/writing) and the average amount of data (reading/writing) per data transfer after memory optimization. The values obtained are N_{tr}, N_{tw}, K_{tr}, and K_{tw} respectively.

(3) If N_{tr} is less than N_r (or K_{tr} is greater than K_r) or N_{tw} is less than N_w (or K_{tw} is greater than K_w), and the difference is larger, it is necessary to adjust the optimization; Otherwise, there is no need to be re-optimized.

4. The set A is all iterations in the nested loop to be analyzed. Unrolling loop and optimizing array partition are carried out according to the following process:

(1) The analysis is started from the most inner loop in the set A. If the layer is already the outermost loop or the cycle order of the layer cannot be exchanged with the innermost loop, record the innermost loop and all loops that can exchange order with the inner loop as the set B. Remove the elements in set B from set A and enter (2); Otherwise, analyze the outer loop.

(2) According to the Eq (10), the number of DSP dspneed required for the scheme is calculated. And then, compare dspneed with the total number of DSP on-chip dsptotal. If $dsp_{need} < dsp_{total}$, and the array division that conforms to the computational parallelism of the scheme can be realized, enter (3); Otherwise, analyze the next scheme.

(3) In this scheme, if all loops in set B are fully expanded and set A is not empty, then enter (1) and calculate the degree of parallelism; otherwise, enter (4).

(4) Optimization is carried out according to the unrolling loop scheme and the corresponding array partition scheme. Analyze whether an array partition that satisfies the computation parallelism in (2) can be achieved. Next, analyze the data after the unrolling loop and group the data, and then the array stored on the same FPGA on-chip memory is divided into a set. Analyze each group of data sequentially, and select the corresponding analysis method according to its storage method on FPGA in the forms of one-dimensional array and multi-dimensional array. If all arrays can be partitioned to satisfy computational parallelism, it is shown that an efficient array partition can be made for the unrolling loop scheme. Otherwise, the effective array partition cannot be carried out.

(5) It is analyzed from two aspects: one-dimensional array and multi-dimensional array. The steps of one-dimensional array analysis are given as follows:

(a) Analyze the address characteristics of each calculation involving data after the loop unrolling. If the addresses are continuous, carry out cyclic division to the array and enter (c). If the address is not continuous but the interval is uniform, carry out block division to the array and enter (c). If the data address characteristics do not meet the both of the above conditions, enter (b). The calculation method of dividing the number of sub-parts storing array data memory is like Eq (11).

(b) If $num\_d < num\_reg$, the array is to be divided entirely and enter (c), otherwise it cannot be effectively divided.num_reg is the total number of FPGA on-chip registers available.

(c) Verify whether the parallelism of data reading/writing after array partition satisfies the parallelism of computation in the unrolling loop scheme. If it is satisfied, the array partition is effective and the array partition scheme is recorded; otherwise, the array partition is invalid.

The steps of multi-dimensional array analysis are as follows:

(a) According to the array dimension, the array is completely divided on this dimension. Verify whether the parallelism of data reading/writing after array partition satisfies the parallelism of computation in the unrolling loop scheme.

(b) If there exists a dimension that can be realized, the multi-dimensional array is regarded as one dimension array of this dimension, and the analysis method is carried out according to the one dimension array; otherwise, it enters (c). It should be noted that in the analysis of complete partitioning, what needs to be considered is not the restriction of registers on the FPGA chip, but the number of subparts of the array partitioning $p\_num < num\_max$.

(c) If $num\_d < num\_reg$, the multi-dimensional array is to be divided entirely, otherwise it cannot be effectively divided.

(6) Optimization based on the cyclic pipelining is as follows:

(a) In loop unrolling, if there is a circular boundary L_b which is x, no optimization is made; Otherwise, enter (2).

(b) If there is a loop which enables all loops to unfold optimization, the involved cycles are to be its sub-cycles, with the most inner loop selected and enter (3); otherwise, the optimization of cyclic flow is not carried out.

(c) If the L_b of all the sub loops in the inner loop is c, the optimization of the cyclic flow is performed; otherwise, it will not be optimized.

(7) According to the program execution report, the percentage of BRAM(B) resources and the percentage of DSP(D) resources consumed by a single computing element are obtained. Calculate the values of 1/B and 1/D, and then compare them with N_k to find out the minimum, with the limit conditions for configuring the number of computing element N obtained according to the Eq (13).

(8) According to the program execution report, the number of storing data memory ports is num_{port}, and the theoretical parallelism of data calculation is v_cal. If the data is not stored in registers and $num_{port} = 1$, it can be judged that the computing does not make full use of data reading/writing parallelism, because the OpenCL compiler can allocate up to two data ports for each memory at most. To deal with this situation, it is generally necessary to re-optimize the calculation. If $num_{port} = 2$, it illustrates that the parallelism of data reading/writing is fully utilized. At this time, compare the values of A and B. If $v\_call > v\_data$, repartition the array.

Experimental section

The compiler tool used in this experiment is the Xilinx SDx tool, and the FPGA development board produced by the Alpha Data company is the ADM-PCIE-7V3 board. Linux is the execution environment of the Host terminal. The specific environment of this experiment is shown in Table 1 and the specific configuration of the ADM-PCIE-7V3 board is given in Table 2.

Optimization example

In this section, this paper introduces the OpenCL example of convolution layer on FPGA firstly. Based on this example, the computational model is applied to the convolution layer.

Table 1. Experimental software and hardware environment.

software environment	operating system	Centos release 6.7 final
	compiler tools	SDx 2016.3
hardware environment	Host terminal processer	Intel (R) Xeon (R) CPU E5-2620 @ 2.00GHz
	FPGA development board	ADM-PCIE-7V3board

https://doi.org/10.1371/journal.pone.0222984.t001

Table 2. ADM-PCIE-7V3 the specific configuration of the board.

FPGA	Logic unit	Host interface	DDR3	Network interface	Power dissipation (typical value)
Virtex-7 VX690t-2(28nm)	693K	PCIe Gen3 x8	16GB ECC DDR3 1333MT/sec	Dual SFP+(10GbE)	23-26W

https://doi.org/10.1371/journal.pone.0222984.t002

Table 3. Parameters of a single convolution layer.

Input		Convolution Kernel			Output	
Feature Map Size	Channels	Kernel Size	Stride	Padding	Feature Map Size	Channels
27×27	48	5×5	1	2	27×27	256

https://doi.org/10.1371/journal.pone.0222984.t003

Later the application of the computational model is explained in detail. Finally, the results of the optimized program execution are given.

Convolution layer OpenCL example on FPGA. In the convolutional neural network model, the operation of each convolution layer is consistent and it is convolution operation. The difference lies in data processing and the scale of data, so the optimization methods and ideas are basically the same in the optimization of different convolution layer. Accordingly, this section focuses on the example of a single convolution layer in convolutional neural network. The example program given in this section is an ordinary convolution layer program without any optimization, whose parameters are shown in Table 3.

The number of convolution kernel channels is 48, and the number of convolution kernels is 256. The convolution layer is mainly implemented in the OpenCL kernel program, and the Host is mainly responsible for configuring the environment required by the kernel program, calling the kernel program, carrying out data transfer with kernel, etc. The specific implementation of pseudo code is shown in Algorithm 1.

Algorithm 1: Realization of pseudo code in convolution layer

```
Input: *image, input feature map data     *weights, input weight data
Output: *out output feature map data
async_work_group_copy(local_image,image,i_channel*ISize*ISize, 0);
async_work_group_copy(local_weight,weights,o_channel*i_channel*WSize
  *WSize, 0);
index←0;
outputLoop: for o_num←0 to o_channel do
  outYAxis: for o_y←0 to OSize do
    outXAxis: for o_x←0 to OSize do
      sum←0;
      convInchan: for conv_num←0 to i_channel do
        convILoop:for conv_y←0 to WSize do
          convJLoop: for conv_x←0 to WSize do
            x_padding←o_x*Stride+con_x-Padding;
            y_padding←o_y*Stride+con_y-Padding;
          if 0≤y_padding<Isize and 0≤x_padding<Isize then
            sum += local_image[con_num * ISize * ISize+y_padding
            * ISize + x_padding]*local_weight[((o_num*i_channel +
            conv_num)*WSize+conv_y)*WSize+conv_x];
          end
        end
      end
    end
    local_out[index++]=sum;
  end
 end
end
async_work_group_copy(out, local_out, o_channel*OSize*OSize, 0);
```

The original convolution layer program is compiled, deployed and run to get information about data transfer as shown in Table 4. where it can be seen that there are multiple data transfers between the kernel of convolution layer and the off-chip global memory, and the average data amount for each data transfer is only 4 bytes. Therefore, the data transfer efficiency is low.

Example optimization scheme. The basic program of convolution layer is optimized according to the hardware architecture diagram and computational model proposed in the third and fourth section. The main work of this section is to apply the computational model to the example program, and give the parameters needed for each step and the detailed optimization scheme.

According to the first step of the computational model, the amount of data N_d contained in the parameter is 307200. Traverse N_e, and figure out $K_n \in \{19200, 38400, 76800, 102400, 153600\}$ according to the parameter calculation Eq (6), and K_n is not empty. Take out the minimum value 19200. The data are grouped in ascending order according to the address. The number of groups is 19200, and the number of data in each group is 16. Carry out the vector optimization to the data which can equivalently replace the original data used in the kernel.

According to the second step of the computational model, the global parameter is set to be _global int 16* pre, and the bit width of data transfer between the computing unit and the memory interconnection / memory controller is set to be 16×32, namely, 512 bits. According to the parameter Eq (8), the bit width is determined to be 512 bits, which makes it accessible to support a single data transfer of 512 bits. Thus, the number of data transfer each time variesfrom 1 to 16, which can effectively reduce the number of memory transfer data.

According to the third step of the computational model, N_t and K_t are calculated by the parameter calculation Eqs (8) and (9). For N_{tr} is larger than N_r in the program execution report and N_{tw} is larger than N_w in the execution report in the example, there is no need to re-optimize this time.

The data transfer between convolution layer kernel and off-chip global memory can be obtained after the first three steps of optimization according to the computational model, as shown in Table 5, which indicates that the number of data transfers between the convolution layer kernel program and the global memory off-chip is greatly reduced, and the average amount of data transfer is increased to 64 bytes.

Table 4. Related information of data transfer in the convolution layer basic program.

	Transfer Type	Number of Transfers	Transfer Rate(MB/s)	AvgBandwidth Utilization(%)	Avg Size(KB)	Avg Time(ns)
Data Transfer: Kernels and Global Memory	Read	408969216	63.049	0.547	0.004	27.240
	Write	186624	0.029	2.4975E-4	0.004	15.000

https://doi.org/10.1371/journal.pone.0222984.t004

Table 5. Information related to data transfer after memory optimization.

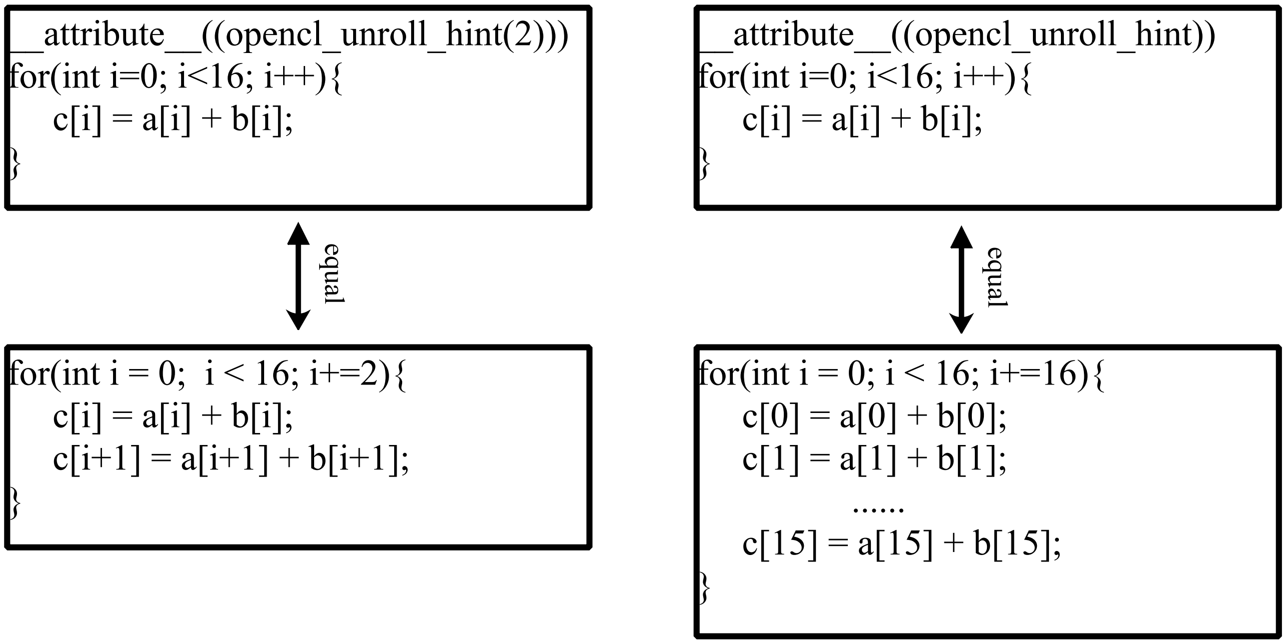
	Transfer Type	Number of Transfers	Transfer Rate(MB/s)	AvgBandwidth Utilization(%)	Avg Size(KB)
Data Transfer: Kernels and Global Memory	Read	21387	1.198	0.010	0.064
	Write	11664	0.654	0.006	0.064

https://doi.org/10.1371/journal.pone.0222984.t005

According to the fourth step of the computational model, the dsp_{need} is figured out to be 251 by the parameter calculation Eq (10). The total number of DSP on-chip is 3600. The number of DSP needed is less than the total number and set *B* is fully unrolled loop. Fig 2 shows the use of optimized instruction and the equivalent code after unrolling. Graph (a) shows the equivalent code when the unrolling factor is 2, and graph (b) shows the equivalent code when the unrolling factor is by default.

For the size of convolution kernel is 5×5, the theoretical parallelism of computation is 25. However, the input characteristic graph data involved in the calculation and the convolution kernel data are stored locally in the form of one-dimensional arrays. Without the array partition optimization, the OpenCL compiler only assigns two ports to it at most. That is, the degree of parallelism of reading is 2, which is much less than the degree of parallelism of calculation, so the arrays need to implement array partitioning.

According to the fifth step of the computational model, the *p_num* of cyclic partitioning and block partitioning are calculated by Eq (11). Since the total number of *num_d* in the array is less than that of *num_reg* of available registers on-chip, the array is completely partitioned. In the process of loop unwrapping and array partitioning optimization, the last three layers of the convolution layer implementation code (calculation of single pixel in output characteristic graph) are optimized and the corresponding array partitioning is carried out. Meanwhile, for the convenience of optimization, this section divides the last three layers into double three-layer according to convolution multiplication and addition. For the two three-layer loops are consistent in architecture, the corresponding optimization strategies are nearly identical. The



```
__attribute__((opencl_unroll_hint(2)))
for(int i=0; i<16; i++){
    c[i] = a[i] + b[i];
}
```

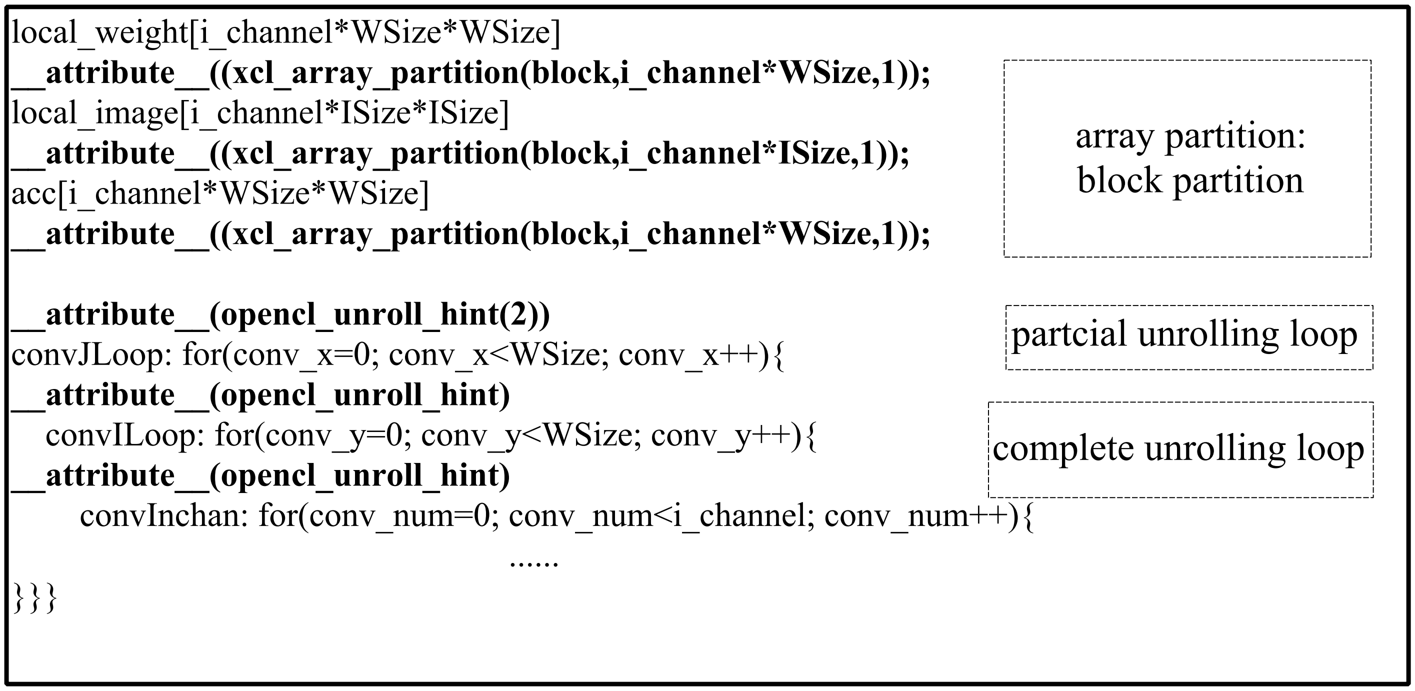
```
for(int i = 0;  i < 16; i+=2){
    c[i] = a[i] + b[i];
    c[i+1] = a[i+1] + b[i+1];
}
```

```
__attribute__((opencl_unroll_hint))
for(int i=0; i<16; i++){
    c[i] = a[i] + b[i];
}
```

```
for(int i = 0; i < 16; i+=16){
    c[0] = a[0] + b[0];
    c[1] = a[1] + b[1];
          ......
    c[15] = a[15] + b[15];
}
```

Fig 2. Example of using unrolling loop instruction.

https://doi.org/10.1371/journal.pone.0222984.g002



```
local_weight[i_channel*WSize*WSize]
__attribute__((xcl_array_partition(block,i_channel*WSize,1));
local_image[i_channel*ISize*ISize]
__attribute__((xcl_array_partition(block,i_channel*ISize,1));
acc[i_channel*WSize*WSize]
__attribute__((xcl_array_partition(block,i_channel*WSize,1));

__attribute__(opencl_unroll_hint(2))
convJLoop: for(conv_x=0; conv_x<WSize; conv_x++){
__attribute__(opencl_unroll_hint)
   convILoop: for(conv_y=0; conv_y<WSize; conv_y++){
__attribute__(opencl_unroll_hint)
      convInchan: for(conv_num=0; conv_num<i_channel; conv_num++){
                                            ......
}}}
```

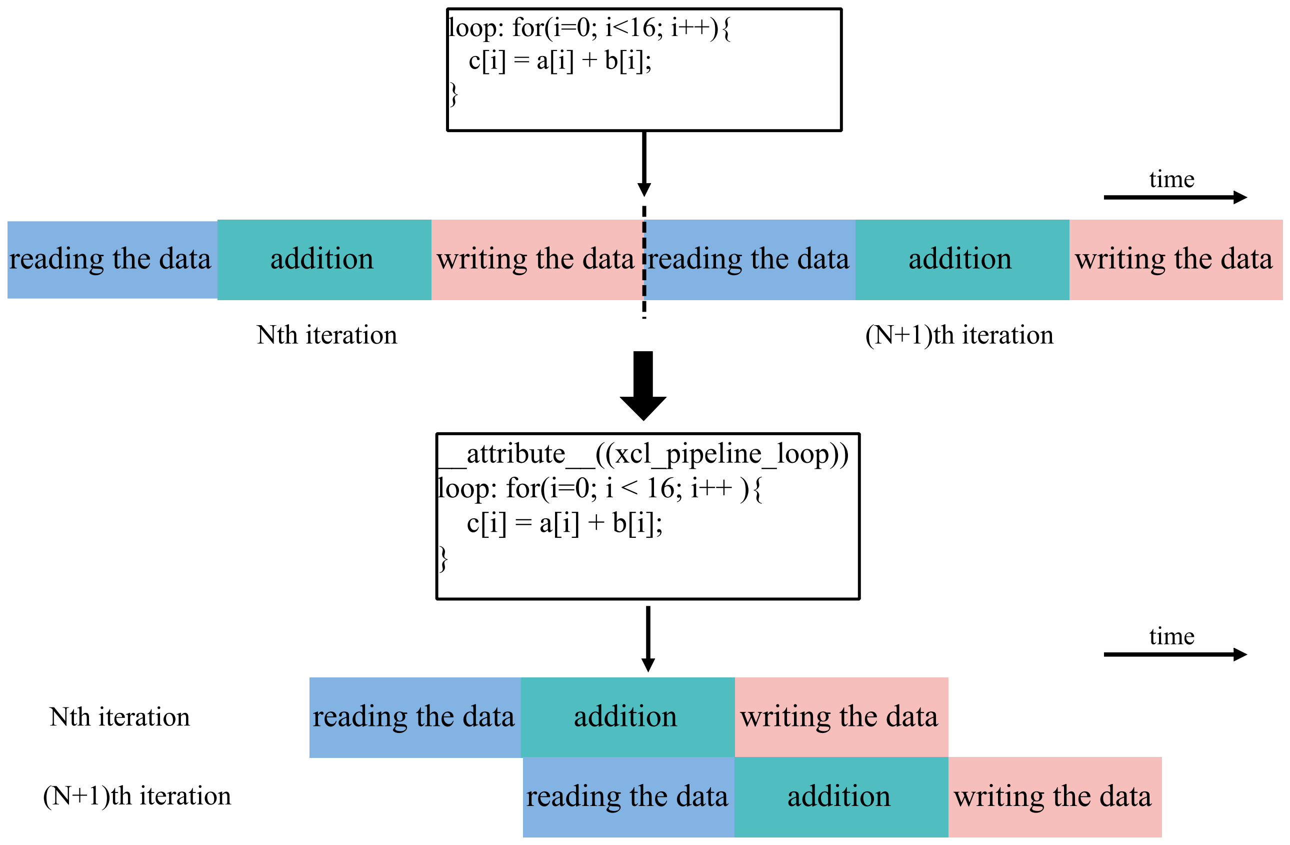
Fig 3. The specific code optimization of convolution multiplication calculation.

https://doi.org/10.1371/journal.pone.0222984.g003

specific code of the optimization of the convolution multiplication calculation is shown as in Fig 3. In this optimization, the inner and outer two-layer cycles are completely unrolled, and the theoretical calculation parallel degree is 48×5, namely 240. The theoretical value of data reading/writing parallelism involved in the calculation is 48×5×2, namely 480. Since the parallelism of data reading and writing is greater than that of computing, the outer loop is partially unrolled in order to match the parallelism of the both and the unrolling factor is 2 in this optimization.

According to the sixth step of the computational model, the cyclic pipelining instruction __attribute__ ((xcl_pipeline_loop)) can be used when writing an OpenCL program. The main function of this instruction is to ensure the FPGA performs each iteration of the loop in a pipelining manner by adding the instruction outside of the for-loop. The entire loop boundaries are constants and the L_b is obtained from Eq (12) to optimize cyclic pipelining for c. Fig 4 shows the use of loop instructions and the execution situation of loops before and after the use of instructions. Figure (a) gives the performance of the cycle execution without using loop pipelining optimization, and figure (b) demonstrates the use of loop pipelining.

According to the seventh step of the computational model, the 1/B and 1/D in the Eq (13) are 22.1 and 14.3 respectively according to the execution report. The compiler limits the number of computing units to 10. According to the parameter calculation Eq (13), the number of N is less than 10. In this range, it is the best to use six computing units for this example program, so six computational are configured elements this time. The kernel program is split into 6 working groups, with each containing only one work items and the specific OpenCL kernel optimization code shown in Fig 5. The output characteristic graph data are stored in order according to the channel. To make the address space of the output data in the off-chip global memory continuous

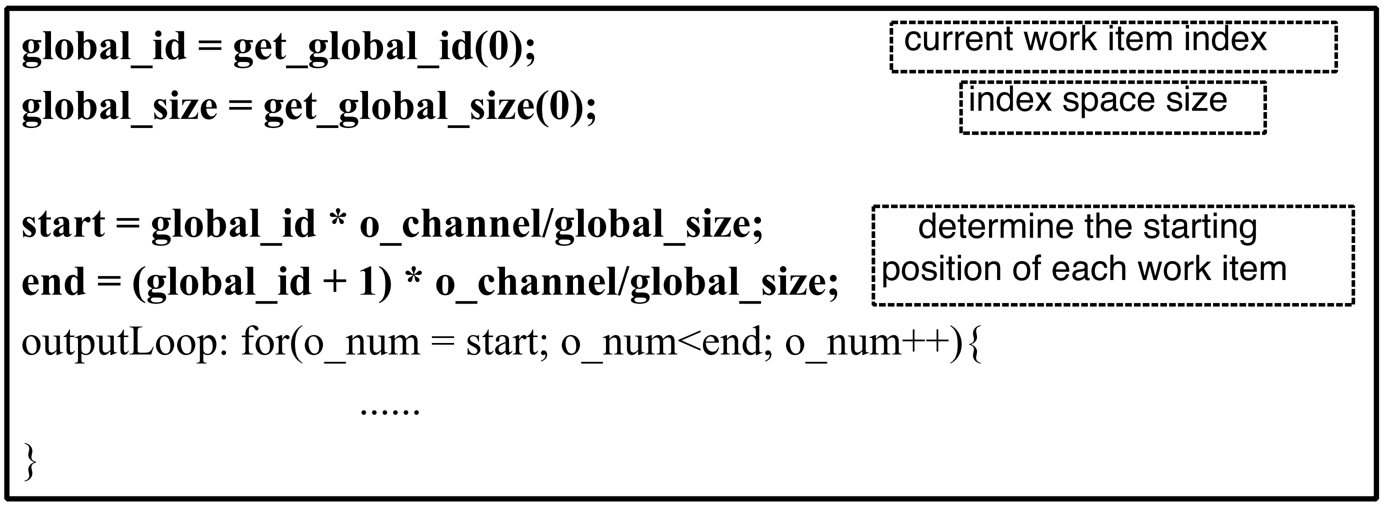


```
loop: for(i=0; i<16; i++){
  c[i] = a[i] + b[i];
}
```

```
__attribute__((xcl_pipeline_loop))
loop: for(i=0; i < 16; i++ ){
   c[i] = a[i] + b[i];
}
```

Fig 4. Using examples of loop instructions.

https://doi.org/10.1371/journal.pone.0222984.g004



```
global_id = get_global_id(0);
global_size = get_global_size(0);

start = global_id * o_channel/global_size;
end = (global_id + 1) * o_channel/global_size;
outputLoop: for(o_num = start; o_num<end; o_num++){
                    ......
}
```

current work item index
index space size
determine the starting position of each work item

Fig 5. Kernel optimization code for multiple computing units.

https://doi.org/10.1371/journal.pone.0222984.g005

Table 6. Optimized kernel program execution time.

Kernel Execution		Total Times(ms)				
		9.760				
	Computational Unit	Global Work Size	Local Work Size	Number of Calls	Xilinx(ms)	Optimized Program
Computational Unit Utilization	1	6:1:1	1:1:1	1	290.751	8.730
	2	6:1:1	1:1:1	1	290.859	8.930
	3	6:1:1	1:1:1	1	291.721	9.026
	4	6:1:1	1:1:1	1	291.684	8.887

https://doi.org/10.1371/journal.pone.0222984.t006

Table 7. Performance comparison of the different optimization programs in the convolution layer OpenCL.

	Original Convolution Layer Program	The Optimization Program Provided by Xilinx Company	The Example Optimization Program	Speedup Ratio
Execution Time (ms)	1142.26	291.977	9.76	29x

https://doi.org/10.1371/journal.pone.0222984.t007

when each computing unit transmits data with the off-chip global memory, this division of the kernel program is based on the number of channels in the output characteristic graph.

According to the eighth step of the computational model, unrolling loop and array partition optimization of the last three layer loops of convolution operation are carried out. The degree of parallelism of the optimized convolution multiplication is 480, which is two fifths of that of the ideally optimized convolution multiplication. Multiple computing units optimization of the outermost loop of the first three layers is carried out, and the parallelism of the output feature image pixel calculation after optimization is 6, which is much less than that of the pixel calculation of the output feature image after optimization. The optimization of cyclic flow is carried out for the inner loop. Finally, by comparing the values of additions, it is found that there is no need to repartition the array.

Optimization performance analysis of the program. According to the computational model proposed, the example code is directed toward optimization, whose result compared to the latest Xilinx optimization program [17] is shown in Table 6. From the runtime of each cell and the whole kernel, it is found that these four cells are basically executed in parallel.

The final optimization result of this example program is shown in Table 7. The final execution time of the example program is 9.76 milliseconds after optimization. Moreover, this paper also tests the performance of the convolution layer optimization program provided by Xilinx, as summarized in Table 5 where it can be seen that the final performance of the program is 29 times higher than that of the optimization program provided by Xilinx company.

The final optimization results of this experiment are compared with the CPU implementation [20], as indicated in Table 8. The 1-thread in the table is set as single thread execution,

Table 8. Comparison of optimization examples and CPU implementation.

	Intel Xeon 2.20GHz		FPGA
	1-thread –O3	16-thread –O3	
Execution Time (ms)	94.66	27.0	9.76
Speedup Ratio	1x	3.5x	9.7x
Power(Watt)	95.00	95.00	25.00
Energy(J)	8.99	2.57	0.24

https://doi.org/10.1371/journal.pone.0222984.t008

Table 9. CNN configurations.

Layer	1	2	3	4	5	6	7	8
Input_Ch	64	128	64	48	64	48	64	96
Output_Ch	128	192	96	96	128	128	96	128
Row	11	13	17	19	23	27	33	36
Col	11	13	17	19	23	27	33	36
Kernel	3×3	3×3	3×3	3×3	3×3	3×3	3×3	3×3
Stride	1	1	1	1	1	1	1	1

https://doi.org/10.1371/journal.pone.0222984.t009

and the 16-thread is set as the 16-thread execution. -O3 represents the optimization level of a compiler is -O3.

From Table 8, it can be seen that the performance of the optimized convolution on FPGA is 9.76 times higher than that of single-thread CPU, 2.8 times higher than that of 16-thread CPU. Also, it is indicated that the energy consumption of the convolution program optimized by the computational model proposed and implemented on FPGA is significantly lower than that of CPU.

Comparison and analysis of different scale convolution programs

In order to analyze the performance of different scale convolution programs, eight kinds of convolution layer programs are set up according to the ascending order as shown in Table 9. Layer 1 is one of scales, and the number of input and output channels is 64 and 128 respectively. Input a picture sample of a size of 111164 with the convolution kernel of 3364 size, the number of convolution kernels are 128 and the step size is 1. The result of the final output after the convolution operation is 1111128. The other 7 scales of convolution program analysis methods are the same with Layer1.

The computational model proposed is applied to the convolution layer of different scale, and its performance is compared with the corresponding optimization program provided by Xilinx company, as shown in Table 10. Convolution of different scales is optimized based on the computational model proposed, and its optimized time consumption is significantly reduced while compared with that of the optimization program by Xilinx. Fig 6 shows the speed-up ratio of Layer1L̃ayer8 where the higher the speed-up ratio, the better the optimization effect is. Accordingly, the optimized program has higher performance than optimized program of Xilinx.

Table 10. Performance comparison of different convolution scale optimization program with Xilinx company.

Convolution scale	Execution time (ms)	
	The optimization program provided by Xilinx company	The program optimized by the computational model
Layer1	12.4	1.4
Layer2	48.81	3.2
Layer3	21.5	2.3
Layer4	20.4	2.6
Layer5	52	4.6
Layer6	54.5	5.6
Layer7	216.2	8.1
Layer8	508.4	12.67

https://doi.org/10.1371/journal.pone.0222984.t010

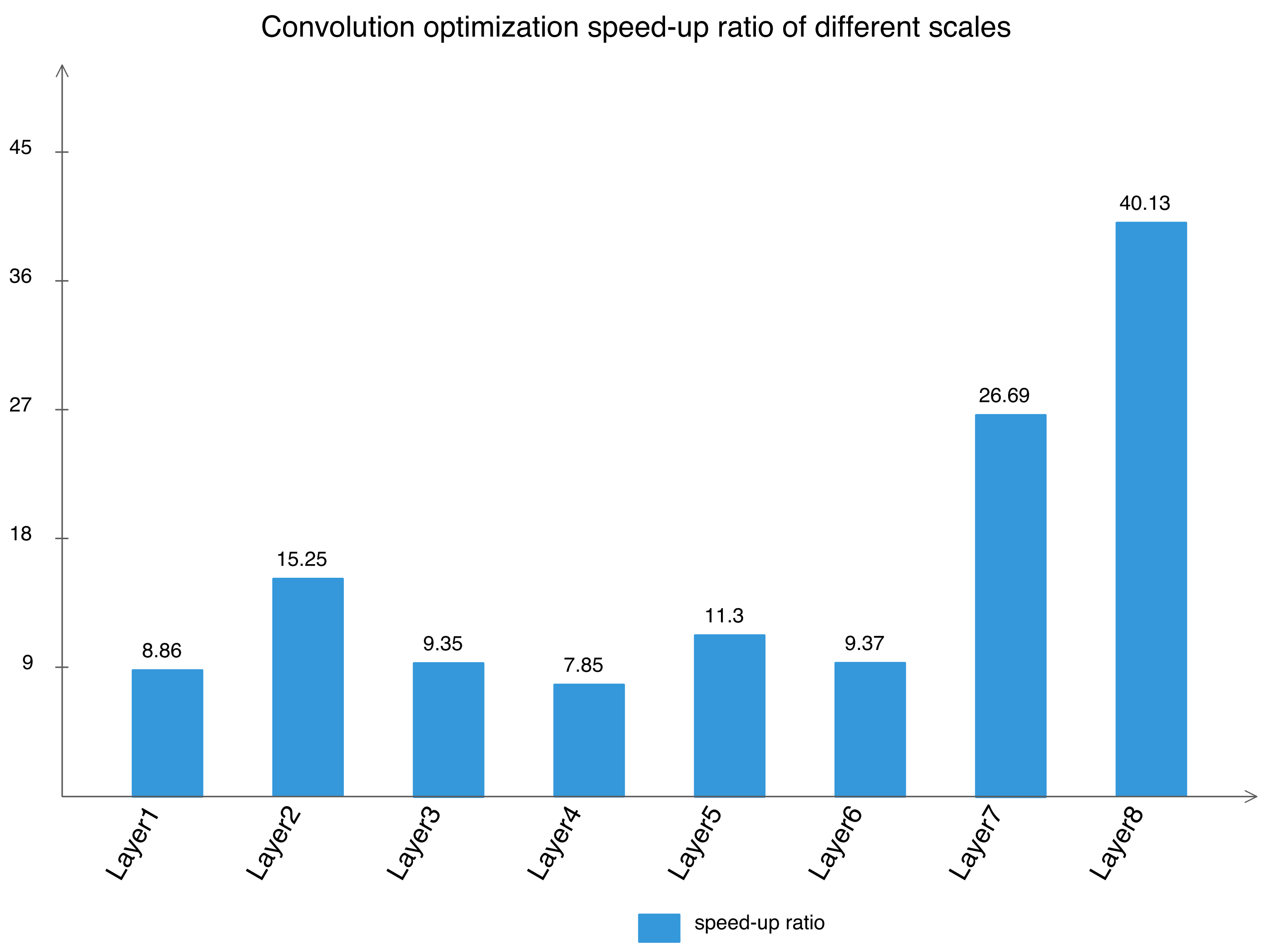


Fig 6. Optimized speed-up ratio of different scale convolution.

https://doi.org/10.1371/journal.pone.0222984.g006

This paper put the code link into the paper openly accessible for other researchers to study and explore new accelerator method for deep neural networks. It can be found at the following link: https://github.com/PoetryAndWine/FPGA_CNN_Acceleration.

Conclusion

This paper proposes an computational model based on OpenCL, which enables the transformation of the OpenCL model on GPU/CPU to FPGA. This computational model is used to help software programmers without fundamental hardware knowledge for a quick implementation in deep learning algorithm with high performance using FPGA. In terms of performance, the computational model not only reduces the cost of data interaction, but also improves the efficiency of data calculation. In terms of adaptability, the computational model is flexible and suitable for convolution layers of different sizes. The results of the proposed computational model applied to convolution layers of different scales show that the performance of the proposed computational model is 8-40 times higher than that of the corresponding optimization program provided by Xilinx Company.

Supporting information

S1 File. Convolution layer optimization code and the performance data. (Xilinx and this paper).
(ZIP)

Acknowledgments

Our work is supported by the National Key Research and Development Program (2016YFB1000400), the Harbin Outstanding Young Talents Fund (2017RAYXJ016),Nature Scientific Foundation of Heilongjiang Province (F2018008).

Author Contributions

Formal analysis: Shuo Zhang, Hongtao He.

Funding acquisition: Yanxia Wu.

Methodology: Chaoguang Men.

Project administration: Yanxia Wu.

Resources: Kai Liang.

Supervision: Yanxia Wu, Chaoguang Men.

Writing – original draft: Shuo Zhang, Kai Liang.

Writing – review & editing: Shuo Zhang, Hongtao He.

References

1. Yu Q, Wang C, Ma X, Li X, Zhou X. A Deep Learning Prediction Process Accelerator Based FPGA. Ieee/acm International Symposium on Cluster, Cloud and Grid Computing. IEEE, 2015:585-594.
2. Lecun Y, Bengio Y, Hinton G. Deep learning. Nature, 2015, 521(7553):436. https://doi.org/10.1038/nature14539 PMID: 26017442
3. Véstias M, Duarte RP, de Sousa JT, Neto H. Parallel dot-products for deep learning on FPGA. Field Programmable Logic and Applications (FPL), 2017 27th International Conference on. IEEE, 2017: 1-4.
4. Zhu J, Qian Z, Tsui CY. LRADNN: High-throughput and energy-efficient Deep Neural Network accelerator using Low Rank Approximation. Design Automation Conference. IEEE, 2016:581-586.
5. Lacey G, Taylor GW, Areibi S. Deep Learning on FPGAs: Past, Present, and Future. arXiv: Distributed, Parallel, and Cluster Computing. 2016
6. Chen DT, Singh DP. Fractal video compression in OpenCL: An evaluation of CPUs, GPUs, and FPGAs as acceleration platforms. Asia and south pacific design automation conference. 2013:297-304
7. Zhang C, Sun G, Fang Z, Zhou P, Pan P, Cong J. Caffeine: towards uniformed representation and acceleration for deep convolutional neural networks. International conference on computer aided design. 2016.
8. Nurvitadhi E, Sim J, Sheffield D, Mishra A, Krishnan S, Marr D. Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC. Field programmable logic and applications. 2016:1-4.
9. Ouyang J, Lin S, Qi W, Wang Y, Yu B, Jiang S. SDA: Software-defined accelerator for large-scale DNN systems. Hot Chips 26 Symposium. IEEE. 2016:1-23
10. Nurvitadhi E, Sim J, Sheffield D, Mishra A, Krishnan S, Marr D. Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC International Conference on Field Programmable Logic and Applications. IEEE, 2016:1-4.
11. Stone JE, Gohara D, Shi G. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. Computing in Science & Engineering, 2010, 12(3):66–73. https://doi.org/10.1109/MCSE.2010.69

12. Wei X, Yu C, Zhang P, Chen Y, Wang Y, Hu H, et al. Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs. The 54th Annual Design Automation Conference 2017. ACM, 2017.
13. Abdelouahab K, Pelcat M, Serot J, Bourrasset C, Quinton JC, Berry F. Hardware Automated Dataflow Deployment of CNNs. arXiv:1705.04543v3.2017
14. Huang Q, Lian R, Canis A, Choi J, Xi R, Brown S, et al. The Effect of Compiler Optimizations on High-Level Synthesis for FPGAs. IEEE International Symposium on Field-programmable Custom Computing Machines. IEEE, 2013.
15. Abdelfattah MS, Hagiescu A, Singh D. Gzip on a chip: high performance lossless data compression on FPGAs using OpenCL. Proceedings of the International Workshop on OpenCL 2013 & 2014.
16. Farabet C, Martini B, Akselrod P, Talay S, LeCun Y, Culurciello E. Hardware accelerated convolutional neural networks for synthetic vision systems. IEEE International Symposium on Circuits & Systems. IEEE, 2010.
17. Qiu J, Wang J, Yao S, Guo K, Li B, Zhou E, et al. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. ACM, 2016.
18. Ko BS, Kim HG, Oh KJ, Choi HJ. Controlled dropout: A different approach to using dropout on deep neural network. IEEE International Conference on Big Data and Smart Computing. IEEE, 2017:358-362.
19. Suda N, Chandra V, Dasika G, Mohanty A, Ma Y, Vrudhula S, et al. Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks. Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. 2016:16-25.
20. Zhang C, Li P, Sun G, Guan Y, Xiao B, Cong J. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. Acm/sigda International Symposium on Field-Programmable Gate Arrays. ACM, 2015:161-170.
21. Czajkowski TS, Aydonat U, Denisenko D, Freeman J, Kinsner M, Neto D, et al. From OpenCL to high-performance hardware on FPGAs. Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on. IEEE, 2012: 531-534.
22. Luo L, Wu Y, Qiao F, Yang Y, Wei Q, Zhou X, et al. Design of FPGA-Based Accelerator for Convolutional Neural Network under Heterogeneous Computing Framework with OpenCL. International Journal of Reconfigurable Computing, 2018, 2018:1–10. https://doi.org/10.1155/2018/1785892
23. Tapiador R, Riosnavarro A, Linaresbarranco A, Kim M, Kadetotad D, Seo J. Comprehensive Evaluation of OpenCL-based Convolutional Neural Network Accelerators in Xilinx and Altera FPGAs. Robotic and Technology of Computers Lab report. 2016.