IEICE TRANS. INF. & SYST., VOL.E101–D, NO.8 AUGUST 2018

PAPER

Design and Implementation of Deep Neural Network for Edge Computing

Junyang ZHANG[†a)], ***Member***, **Yang GUO**[†b)], **Xiao HU**[†c)], ***and*** **Rongzhen LI**[†d)], ***Nonmembers***

SUMMARY In recent years, deep learning based image recognition, speech recognition, text translation and other related applications have brought great convenience to people's lives. With the advent of the era of internet of everything, how to run a computationally intensive deep learning algorithm on a limited resources edge device is a major challenge. For an edge oriented computing vector processor, combined with a specific neural network model, a new data layout method for putting the input feature maps in DDR, rearrangement of the convolutional kernel parameters in the nuclear memory bank is proposed. Aiming at the difficulty of parallelism of two-dimensional matrix convolution, a method of parallelizing the matrix convolution calculation in the third dimension is proposed, by setting the vector register with zero as the initial value of the max pooling to fuse the rectified linear unit (ReLU) activation function and pooling operations to reduce the repeated access to intermediate data. On the basis of single core implementation, a multi-core implementation scheme of Inception structure is proposed. Finally, based on the proposed vectorization method, we realize five kinds of neural network models, namely, AlexNet, VGG16, VGG19, GoogLeNet, ResNet18, and performance statistics and analysis based on CPU, gtx1080TI and FT2000 are presented. Experimental results show that the vector processor has better computing advantages than CPU and GPU, and can calculate large-scale neural network model in real time.

key words: *edge computing, vector processor, convolutional neural network, multi-core optimization*

1. Introduction

Edge computing [1]–[3] is a relatively new term in a number of areas, it involves network connectivity, data aggregation, chip design, sensor technology, industry applications and other industrial chain. As the name of edge, especially near the object or data source of the network edge side, is fused network, computing, storage, application of core capability of the open platform. Providing intelligent service on edge side, meet the demand of digital key industry in agile connectivity, real-time business, data optimization, application intelligence, security and privacy protection and other aspects of the key needs. In cloud computing, all data are gathered to the cloud data center, while edge computing emphasizes the "edge", that is the data is processed closer to the device. Today, we have moved from the internet era into the internet of everything (IoE) [4]. According to IDC' statistics, by 2020, there will be more than 50 billion terminal devices which will generate data greater than 40 ZB [5]. With the increase number of edge devices, network bandwidth has become a major bottleneck in cloud computing, but simply increase the network bandwidth can't meet the network requirements of new internet applications.

Why edge computing is so important? With the advent of the internet of everything, the number of edge devices is rapidly growing, such as smartphones, wearable devices and smart cameras which in turn generate large amounts of data on the edge devices. If the data on the edge devices is transmitted to the cloud to deal with, it will face the following major problems: 1) Response not timely, the data transmitted from the terminal device to the cloud, calculate and then return to the device, the delay of this process is too long, it will cause the application of dull response. 2) Data transmission bandwidth is insufficient. Internet of things data mostly have strong timeliness, these instantaneous mass data (probably audio and video data), if all upload to cloud computing platform, may cause connection bandwidth congestion. Usually, the post analysis value of these data is not high and all data upload will cause serious waste of bandwidth resources. 3) Cloud loads. If application data (including "real-time" data) upload, it will not only squeeze the bandwidth resources but also engulf the cloud computing and storage resources. 4) Data security and service guarantee. It will bring great challenges to data security and service reliability if only depend on cloud computing in the application of IoT. First of all, we need enough technology to ensure information security, however it will need much maintenance cost, once cloud computing failure, it may cause the application fall into paralysis. Secondly, there are too many "possible fault nodes" from device to cloud, such as gateway, transmission, operation system, software, storage, middleware platform, and powerful organization which can't guarantee cloud service never be interrupted. Therefore, we need edge computing to make up for the lack of cloud computing.

How does edge computing and cloud computing work together? Essentially, edge computing is "ground" cloud computing, and its most important ability is to inherit the intelligence of cloud computing. Edge computing using distributed computing architecture, the operation is scattered near the data source of the proximal equipment to deal with, sharing the workloads of the cloud platform, and no longer need return to the cloud center to process, real-time better, more efficient, shortest delay, even without network

Manuscript received February 1, 2018.
Manuscript revised March 30, 2018.
Manuscript publicized May 2, 2018.

[†]The authors are with the College of Computer, National University of Defense Technology, China.
a) E-mail: zhangjunyang11@nudt.edu.cn
b) E-mail: guoyang@nudt.edu.cn (Corresponding author)
c) E-mail: hu.xo@163.com
d) E-mail: lirongzhen11@nudt.edu.cn

DOI: 10.1587/transinf.2018EDP7044

and unable to access the cloud, it will not interfere with the "ground" calculation of the edge devices. When it has good network conditions, the cloud can reduce computing tasks on the local side and complete information storage and real-time data sharing. If network and cloud computing are limited, edge computing can independently process information, when the line connected, edge devices need to transfer relevant information to the cloud to realize information sharing and interaction with different edge devices. Therefore, edge computing and cloud computing are complementary relationships which can provide intelligent services to human beings through coordination.

What challenges of designing and implementing large scale neural networks on edge devices? Due to excellent performance of large-scale neural network models in image recognition [6], speech recognition [7], text processing [8], target detection [9], target tracking [10] and other fields, it has great research value to deploy large neural networks on edge devices to make the edge devices intelligent. Usually, large-scale neural network has huge amount of computation. Figure 1 shows the relationship between the recognition rate of Top1 and the amount of computation needed in several commonly used neural networks, it can be found that the higher the recognition rate is, the larger the computation amount is. Therefore, the deployment of large-scale neural network model on edge devices which with limited computing resources will face the following challenges. 1) Limited computing resources, the microprocessor on the edge device belongs to embedded processor, so the performance is not high. 2) Limited storage resources, embedded microprocessor has small memory on chip, and it is difficult to load neural network models such as VGG. 3) Power limitations, edge devices such as mobile phones, unmanned aerial vehicles, vehicle cameras, and so on, if the power consumption can't be reduced, it will seriously affect the use of such devices. 4) Architecture defects, the current edge processor architecture does not specifically optimize the neural network model, resulting in new computational method such as convolutional neural network (CNN) can't be effectively implemented on embedded microprocessors. 5) Lack of programming framework, although there are many deep learning programming frameworks such as Caffe [11], Torch, Tensorflow [12], Keras, etc., while most of them running only on CPUs or GPUs and it is difficult to run on specific embedded processors.

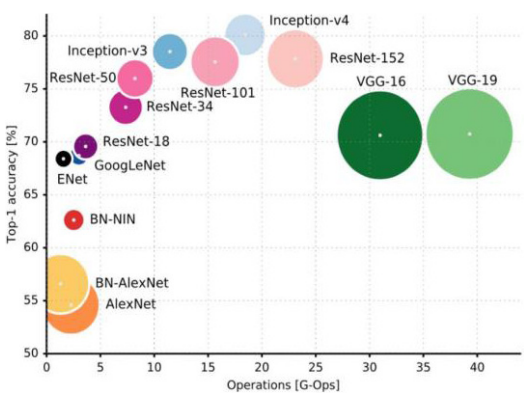

Fig. 1 Relationship between recognition rate of common neural networks and calculation amount

In view of the above problems, we propose a microprocessor architecture for edge computing based on deep learning algorithm. Taking the GoogLeNet neural network model as an example, this paper will introduce efficient implementation of large-scale neural network model on the embedded processor. The major contributions in this paper are summarized as follows:

1. A new 2D matrix convolution vectorization method is designed and implemented on the vector processor, and this method is extended to multidimensional matrix convolution, which greatly improves the calculation speed of the large convolutional neural network, it also improves the computational efficiency of the processor.

2. A vectorization operation of multidimensional max pooling is proposed. Two-dimensional pooling operation is hard to parallel, so we change it to multidimensional pooling operation. Set the vector register with zero as the initial value of max pooling, by fusing the ReLU activation function and the pooling operation to reduce repeated access to the intermediate data and further improve the computational efficiency of the model.

3. On the basis of single-core implementation, a multi-core Inception structure is proposed. The multi-core implementation avoids interaction between cores, so the performance can grow linear with the increase number of cores, the multi-core utilization rate can reach 92.59% on GoogLeNet.

4. Based on our implementation scheme, the large-scale neural network models of AlexNet, VGG16, VGG19 and ResNet18 are implemented on multi-core vector processor, and their performance is analyzed based on the mainstream CPU and high performance GPU. Finally, we put forward the next research plane.

2. Related Works

Although the neural network greatly improves our image recognition, speech recognition, and natural language processing capability, it also brings a huge computing demands, especially for edge devices, because it requires higher computational performance, lower power consumption, robustness, and lower price. For deep learning applications, the current hardware acceleration mainly depends on graphics processing unit (GPU) [13], [14], compared to traditional general-purpose processor (CPU), the core computing power of GPU is greater and easy to parallel, while the GPU's power consumption is too high for edge devices. Hundreds of watts consumption on edge devices is a huge challenge, therefore, GPU is not suitable for deployment on edge devices. As a result, many architectures for edge computing is now emerging, such as the tensor processing unit (TPU) [15], which is customized by Google for infer-

ence stage, its core has a 65,536 8-bit matrix multiply array and 28 MB on-chip memory, it can provide 92 tera operations per second (TOPS) of peak performance, as TPU is mainly used and deployed in Google and not for external sale, which limits the promotion and use of TPU.

In June 2016, Vimicro announced China's first mass-produced embedded processing network chip (NPU), mainly for embedded video surveillance, intelligent driving assistance, unmanned aerial vehicles, robots, etc. The processor is designed mainly for the characteristics of CNN, in one clock cycle, the NPU processor can simultaneously perform 64 long bit multiply accumulate (MAC) operations or 128 short bit MAC operations, and each NPU processor has 38 giga operations per second (GOPs) long bit or 76 GOPs short bit processing capacity. In 2016, CEVA company launched the XM6 digital signal processing (DSP) embedded neural network processor, allegedly to build a programmable DSP which support for deep learning, it used mainly for unmanned aerial vehicle (UAV), smart phones, surveillance cameras, etc. The processor is based on their existing DSP structure and optimized for CNN which contains scalar processing unit, vector processing unit, and a matrix convolution acceleration structure. It is said that the matrix convolution acceleration structure can greatly reuse the repetitive data in the convolution calculation to improve computational efficiency of the algorithm. At the same time, the CEVA deep neural network (CDNN) software programming framework was constructed, which can migrate the commonly used neural network model from a pre-training network to an embedded system. While the above two edge-oriented processors do not introduce the detailed implementation of the accelerated neural network, it only introduces the processor's associated architecture and performance of individual models, therefore, the real implementation efficiency of the model is remains to be turned over.

In particular, it is worth mentioning that the institute of computing technology, chinese academy of science, Chen Yunji et al., who proposed the Cambrian series of processors, the design of neural network accelerator of the Cambrian series in 2013 ASPLOS, 2014 MICRO, 2015 ASPLOS, ISCA, 2016 ISCA, MICRO and other international top conference published, it has made an important influence in the world and becomes the representative of the dedicated neural network accelerators. In 2015, they presented ShiDianNao [16] which is a dedicated neural network processor, it can be embedded in mobile phones and other terminals used for video, image intelligence assistants and with extremely low power consumption, it compared to mainstream GPU can have 28× performance, 4700× performance power ratio, while most of these processors are only used simulation and support small neural networks in their papers. Due to the rapid change of algorithm in deep learning, the algorithm model realized by the above processors is small, and current neural network models such as GoogLeNet [17], [18], VGGNet [19] and ResNet [20] involve large number of calculation amount. Therefore, we need a microprocessor architecture that is more adaptable to large-scale neural network model calculation.

3. Processor Architecture for Edge Computing

3.1 Architecture of FT2000

FT2000 is a high-performance floating-point multi-core vector processor for high-density calculation developed by the National University of Defense Technology. Its single chip integrates 12 vector processor cores, with 1 GHz frequency, and double-precision peak performance up to 1,152 giga floating-point operations per second (GFlops). The single-core structure is shown in Fig. 2. Each core has vector processing unit (VPU) and scalar processing unit (SPU). The SPU is responsible for the scalar calculation and flow control, the VPU is responsible for the vector calculation and includes 16 vector processing elements (VPEs). Each VPE contains a local register file and three floating-point multiply-accumulate units (FMACs), two load/store, and one BP, for a total of six parallel parts. The local register file contains 1024 64-bit registers. The SPU can support broadcast instructions to broadcast scalar data to vector registers and can exchange data with shared registers between VPUs. Vector processor cores support 11 launches and variable-length VLIW instruction, including five scalar and six vector instructions, and the instruction dispatch unit to identify and dispatch the package to the corresponding functional unit to execute. Vector instructions can execute simultaneously on each VPE, the computing unit supports three concurrently executed double-precision floating-point operations, and the vector access unit supports two 2048-bit vector data Load/Stores. In addition, FT2000 provides scalar memory (SM) including L1D and L1P, which can be configured into full SRAM, full cache, or a combination of both, and provides array memory (AM) of 768 KB for vector access. The multi-core adopt shared memory architecture can support multi-core shared global cache (GC) between double data rate (DDR) and L1D, which helps achieve exchange and sharing of data between cores, DDR can support up to 128 GB of memory space, and the data in DDR can broadcast to each core through broadcast instructions.

3.2 Comparison of FT2000 with Other Processor Architectures

FT2000 and DaDianNao [21], NPU are similar in that they are all multi-core architecture, FT2000 is 12 cores, DaDianNao and NPU are 4 cores. In addition, FT2000 and CEVA-XM6 are vector processors that includes vector processing unit and scalar processing unit, the main difference is that CEVA-XM6 is designed for accelerating only matrix convolution, while FT2000 is optimized by improvement of the algorithm. The similarity between FT2000 and Cambricon [22] are that they are all programmable by instruction set, using instruction set can quickly realize different kinds of neural network, except that Cambricon only simulates and does not have a slice. FT2000 and TPU

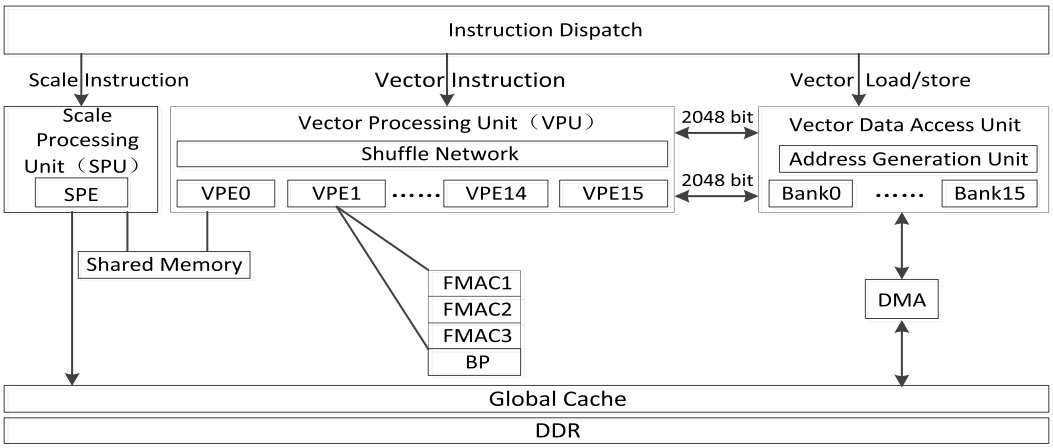

Fig. 2 Architecture of FT2000.

Table 1 Comparison of parameters between FT2000 and current mainstream neural network accelerators

Type	DianNao [23]	DaDianNao	ShiDianNao	PuDianNao	CEVA-XM6	TPU	NPU	FT2000
Data width	16	16	16	16	16	16/8	16/8	64/32
Frequency (GHz)	0.98	0.6	1	1	1.5	0.7	—	1
Performance	452 Gops	—	194 Gops	1,056 Gops	—	91.75 Tops	152 GOPS	1.2 TFlops
Area (mm2)	3	68	4.86	3.51	—	331	—	40
Power (W)	0.485	16	0.3201	0.596	—	40	—	30
Technology (nm)	65	28	65	65	20	28	—	40
Cores	s	m	s	s	s	s	m	m
Universal	no	yes	no	no	no	no	no	yes

(Gops - billions of fixed-point operations per second; Tops - Tera Operations Per Second; TFlops - tera floating-point operations per second; s – single-core; m – multi-core)

are similar in their architecture, except that FT2000 is a general-purpose neural network accelerator, while TPU only supports CNN, LSTM, and MLP. In a word, FT2000 is a kind of low power multi-core vector processor which can support the large scale, more kinds, and focus on the embed domain. Table 1 is a comparison of parameters between FT2000 and the current mainstream neural network accelerators, it can be seen that in addition to FT2000 supports 32/64 bit floating-point, most other processors only support low numerical precision, 8/16 fixed-point or floating-point, their chip frequency, peak performance, chip area, power, technology, number of cores and universal have great differences. So it is worth studying how to evaluate so many neural network accelerators.

4. Structure of GoogLeNet

GoogLeNet was presented by Google in the 2014 imagenet large scale visual recognition challenge and became the champion model, it is a 22-layers deep neural network and its top-5 error rate up to 6.66%, which is significant lower than the 2012 champion model (AlexNet). It is also more complex than the VGGNet and has more calculation types, including 1×1, 3×3, 5×5 and other different matrix convolution sizes, different stride, pad operation, max pooling, average pooling, local response normalization (LRN), and ReLU activation function, it almost covers all the calculation types in CNN. So this study takes the GoogLeNet as an example to introduce the detailed implementation on FT2000 vector processor. Figure 3 shows related parameters about GoogLeNet, including the type of calculation, convolutional kernel size, stride, the number of convolutional kernels, number of parameters, and calculation amount.

Figure 4 shows the overall structure of GoogLeNet, the main layer includes convolutional layer, pooling layer, LRN layer, and other major operations. The biggest feature is the middle part of GoogLeNet, which is composed with 9 Inception modules, the structure of a single Inception module is shown in Fig. 5. The Inception module improves the recognition rate of the model by combining the features extracted with different convolutional kernel sizes, and greatly reduces the number of model parameters. In Sect. 5, we will describe the detail implementation of GoogLeNet on FT2000.

type	kernel size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	Pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
Inception(3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
Inception(3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
Inception(4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
Inception(4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
Inception(4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
Inception(4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
Inception(4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
Inception(5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
Inception(5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout(40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

Fig. 3 Related parameters of GoogLeNet.

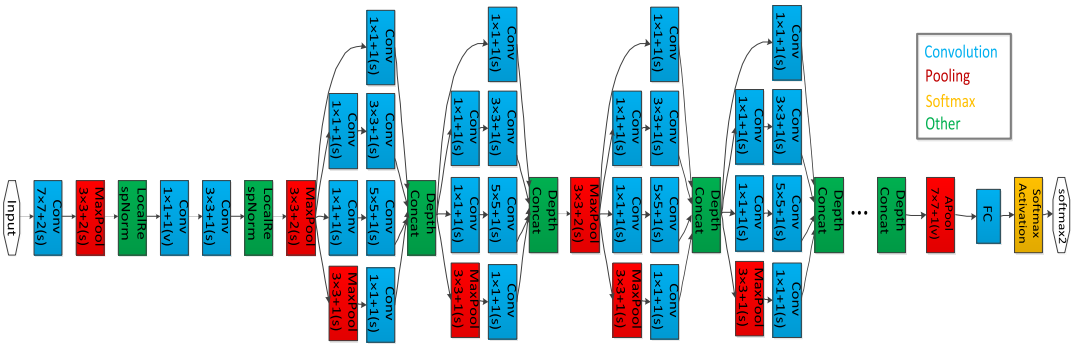

Fig. 4 Overall structure of GoogLeNet.

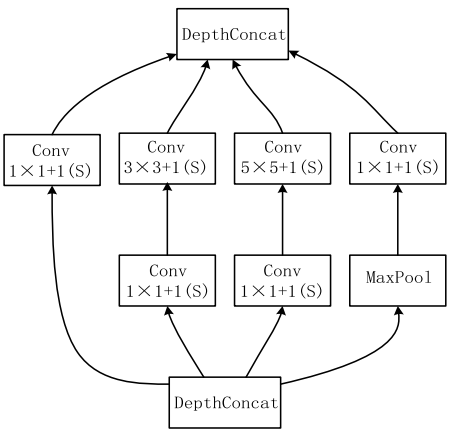

Fig. 5 Structure of Inception.

5. Design and Implementation of the GoogLeNet Network Model

5.1 Data Layout Analysis

In general, the CNN includes convolutional layers, pooling layers, normalized layers, fully connected layers, and a softmax layer. The computing time of convolutional layers accounts for about 85% of the total model [22], thus accelerating convolution calculation in CNN becomes a hotspot in current neural network acceleration. As the convolution calculation is mainly carried out by a large input feature map and a small convolutional kernel, the convolutional kernel is smaller in size, such as 1×1, 3×3, 5×5, whereas the input feature map has a larger scale, such as $224 \times 224 \times 3$ in GoogLeNet. If the double-precision (64-bit) to calculate, then an input feature map will occupy about 1.15 MB, while on-chip storage resource for embedded devices is a valuable resource. For FT2000, the scalar memory is 96 KB, the vector memory is 768 KB, and the external storage space DDR can support up to 128 GB. Based on this, we use the rearrangement of convolutional kernel data placed in vector memory, the input feature map is placed in DDR, which can be broadcast to the AM by broadcast operation. This method can greatly improve the reuse rate of convolutional kernel data, and it is also possible to greatly expand the storage capacity of the input feature maps, because they can be placed in a larger DDR memory space. As GoogLeNet involves multidimensional convolution operations, this paper uses a convolutional kernel of $3 \times 3 \times 16$ dimension as an example to illustrate the data arrangement on the FT2000. As shown in Fig. 6, the same color indicates the same convolutional kernel, whereas different color represents different convolutional kernels. The 16 convolutional kernels are

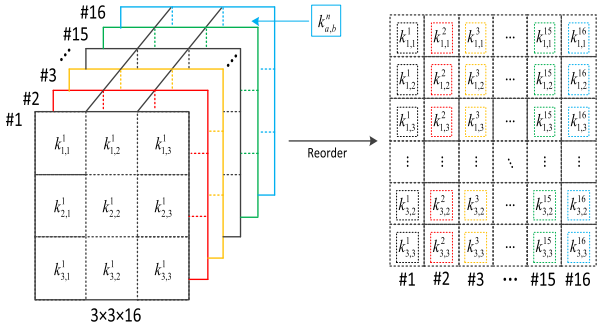

Fig. 6 Reorder of 16 dimensional convolutional kernel data.

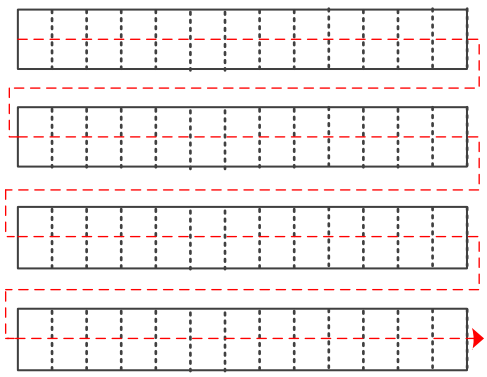

Fig. 7 Arrangement of input feature in DDR.

sorted sequentially in the third dimension, and the result is shown in Fig. 6, where n is the n-th convolutional kernel, a and b represent element in row a, column b of the n-th convolutional kernel.

For deep learning applications based on edge devices, we do not need to carry out network training on edge devices, the training is carried out in servers with GPU and the model is deployed on edge devices after training. Therefore, the edge device is only used in the inference and the model parameters are fixed. During the running of model, the parameters are not trained, therefore, we can use the third-party software, such as MATLAB® to reorder the weight parameters as shown in Fig. 6. Once the weight parameters are completed, they can be deployed in edge device memory for unlimited use and do not use any edge device's computing resources.

The image recognition involves different images collected by the camera, so it changes over time and the image size is always large. In this implementation, we put the input feature map into DDR, and the feature map is sorted in the normal order, that is from top to bottom, and from left to right, as shown in Fig. 7.

5.2 Multidimensional Matrix Convolution Realization

5.2.1 Single Input Feature Map with Multiple Convolutional Kernels

Since the convolutional layers in most CNN and GoogLeNet

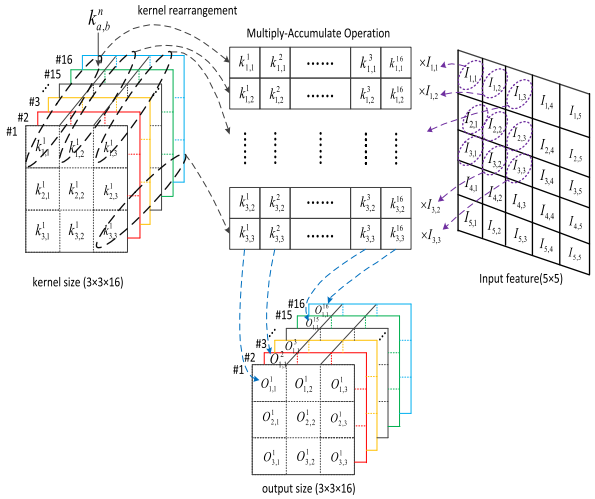

Fig. 8 Convolution calculation of single input feature map and 16 input convolutional kernels

are multidimensional convolution, that is not single input feature map and single convolutional kernel for convolution calculation, but it is multiple input feature maps and multiple input convolutional kernels to calculate multiple output feature maps.

Figure 8 shows the calculation of a $5 \times 5 \times 1$ input feature and a $3 \times 3 \times 16$ convolutional kernels. The number of output features is determined by the number of convolutional kernels. Therefore, when pad is zero and stride is one, then the size of output feature maps is $3 \times 3 \times 16$, wherein in this example, a, $b \in [1, 3]$, $n \in [1, 16]$, the element $O^n_{a,b}$ represents the element in the a-th row and the b-th column of the n-th channel. When a, b, n is one, it indicates the element in the first row and the first column of the first output feature.

When the 16 convolutional kernels sliding over the input feature, elements at the same position within the 16 output features can be calculated, simultaneously. The following is the detail vectorization calculation process of the first element in 16 output feature maps (as shown in Fig. 9, other calculation process of elements is similar).

Cycle #1 : Use the vector VLOAD instruction to load the first row elements $(k^1_{1,1}, k^2_{1,1}, k^3_{1,1}, \ldots, k^{16}_{1,1})$ to the vector register VR0. The scalar processing unit loads the first element $I_{1,1}$ of the input feature map and uses the scale vector broadcast instruction to broadcast the element to vector register VR1. Use the multiply-add instruction VFMULAD to multiply VR0 by VR1 and accumulate the result in the accumulator register **ACC**.

Cycle #2 : Use the vector VLOAD instruction to load the second row elements $(k^1_{1,2}, k^2_{1,2}, k^3_{1,2}, \ldots, k^{16}_{1,2})$ to the vector register VR0. The scalar processing unit loads the second element $I_{1,2}$ of the input feature map and uses the scale vector broadcast instruction to broadcast the element to vector register VR1. Use the multiply-add instruction VFMULAD to multiply VR0 by VR1 and accumulate the result in the accumulator register **ACC**.

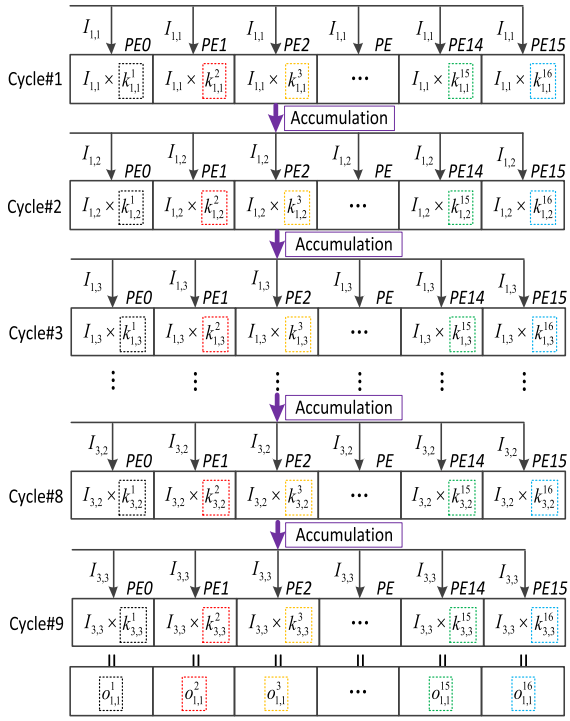

Fig. 9 Vectorization of 16 output feature maps

Cycle #3 : Use the vector VLOAD instruction to load the third row elements $(k^1_{1,3}, k^2_{1,3}, k^3_{1,3}, \ldots, k^{16}_{1,3})$ to the vector register VR0. The scalar processing unit loads the third element $I_{1,3}$ of the input feature map and uses the scale vector broadcast instruction to broadcast the element to vector register VR1. Use the multiply-add instruction VFMULAD to multiply VR0 by VR1 and accumulate the result in the accumulator register **ACC**.

...

Cycle #9 : Use the vector VLOAD instruction to load the ninth row elements $(k^1_{3,3}, k^2_{3,3}, k^3_{3,3}, \ldots, k^{16}_{3,3})$ to the vector register VR0. The scalar processing unit loads the ninth element $I_{3,3}$ of the input feature map and uses the scale vector broadcast instruction to broadcast the element to vector register VR1. Use the multiply-add instruction VFMULAD to multiply VR0 by VR1 and accumulate the result in the accumulator register **ACC**.

With the implementation of cycle #1 to cycle #9 of the calculation process, a total number of eight cumulative cycles, you can simultaneously complete the first element within the 16 output feature maps, that is, $(o^1_{1,1}, o^2_{1,1}, o^3_{1,1}, \ldots, o^{16}_{1,1})$. According to the stride and direction of the convolutional kernel on the input feature map, we can control the loading order of the input feature map and repeat cycle #1 to #9 to complete all output feature maps. It should be noted that we can simultaneously calculate a number of output feature maps by this calculation method, namely, how many PEs the processor own how many output feature maps you can calculate at the same time, each output feature map shares elements of input feature map and enjoys different convolutional kernel elements. In the implementation process of FT2000, because of the sufficient numbers of PEs, the performance of FT2000 is not fully exploited in some layers.

5.2.2 Multiple Input Feature Maps with Multiple Convolutional Kernels

Figure 10 is a schematic of matrix convolution of 2 output channels and 3 input channels, the input dimension is $(3 \times 5 \times 5)$, 5 indicates input feature size,3 represents input feature numbers, and the kernel size is $(2\times3\times3\times3)$, namely (output feature channel, input feature channel, kernel width, kernel height). when the stride is 1, padding is 0, then output channel is $(2 \times 3 \times 3)$. Figure 10 shows the computational process for the first element of the 2-channel output feature maps. As can be seen from Fig. 10, putting the rearrangement kernel elements into specified address, through the loop to control loading order of input feature maps, and then use multiply-add operation can get the first elements of the two output feature maps simultaneously. Repeat the above loop can get all the elements of two output feature. Algorithm of 2 output channels calculate the elements as follows:

```
1: Begin:
2: Rearrangement kernels
3: for i = 0; i < output_size_width; i++
4:   for j=0; j<output_size_height; j++
5:     Acc_0 = 0
6:     Acc_1 = 0
7:     Acc_2 = 0
8:     for s = 0; s < kernel_width; s++
9:       for t = 0; t < kernel_height; t++
10:        temp_0 = svbcast (input[channel_0][i+s][j+t])
11:        temp_1 = svbcast (input[channel_1][i+s][j+t])
12:        temp_2 = svbcast (input[channel_2][i+s][j+t])
13:        Acc_0 = Multiply_add (kernel[0][s][t], temp_0, Acc_0)
14:        Acc_1 = Multiply_add (kernel[1][s][t], temp_1, Acc_1)
15:        Acc_2 = Multiply_add (kernel[2][s][t], temp_2, Acc_2)
16:      end for
17:    end for
18:    output_feature [i][j] = Acc_0 + Acc_1 + Acc_2
19:  end for
20: end for
```

The rearrangement process of the convolutional kernel $(2\times3\times3\times3)$ is shown in Fig. 11. We place the same channel elements with the same position in the adjacent position.

5.3 Multidimensional Max Pooling Implementation

The calculation method in the convolutional neural network includes the convolution operation, the pooling operation, and the other corresponding auxiliary operations, pooling layer is located after convolutional layers. Generally speaking, we want to use these features to do classification, and in theory we can use all the extracted features to train the classifier, but it will face a huge computational challenge. Suppose a 96×96 input image, it has learned 400 features defined on the 8×8 image, each feature and input image convolution will have a dimension $(96-8+1)\times(96-8+1) = 7921$.

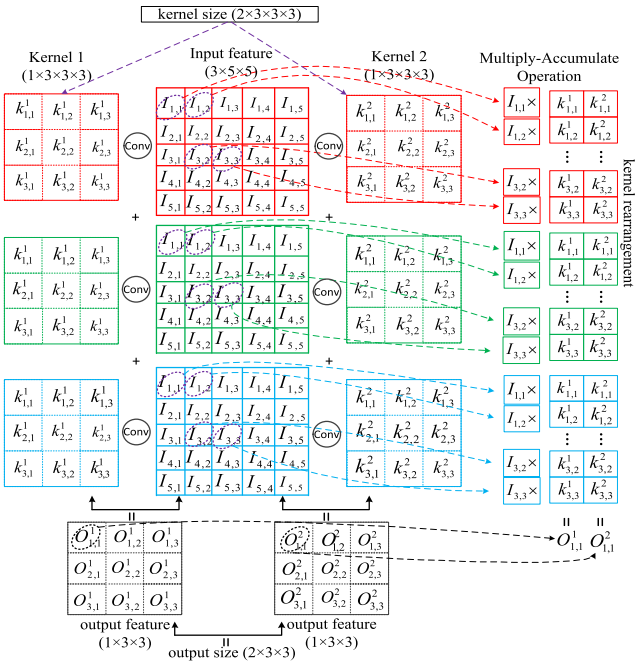

Fig. 10 Matrix convolution of 2 output channels and 3 input channels

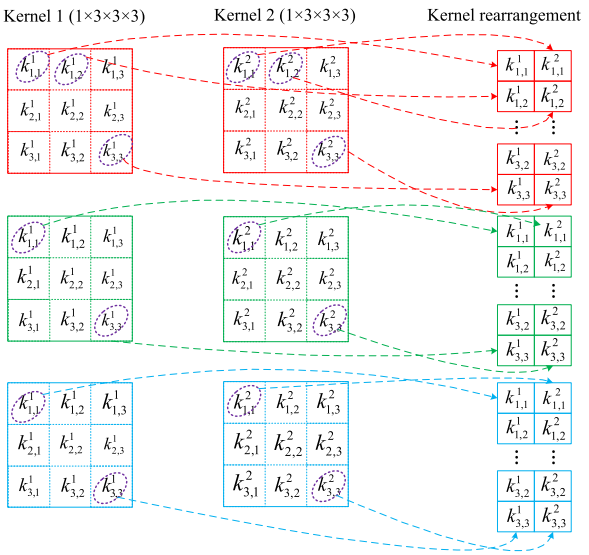

Fig. 11 Rearrangement of convolutional kernel ($2 \times 3 \times 3 \times 3$)

There are 400 features, so each image will have a convolution feature vector with the dimension $89 \times 89 \times 400 = 3{,}168{,}400$. Such a large classifier is easy to overfit, so it is necessary to reduce the dimension of the output feature maps by pooling operation in CNN.

Two-dimensional max pooling calculation process is shown in Fig. 12. As seen in Fig. 12, the two-dimensional max pooling operation is to take the maximum value in the $k \times k$ pooling window as the pooling result. As the two-dimensional max pooling operation is generally a 2×2 pooling window with a moving stride of 2, the parallelism is very low on vector processor which based on single instruction

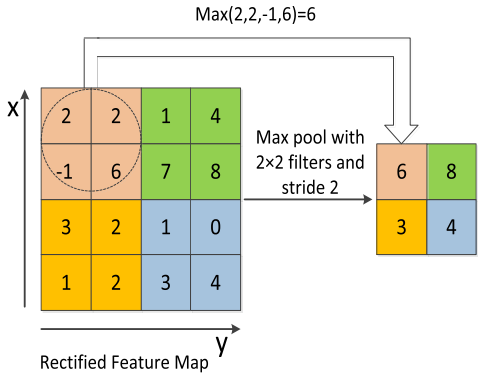

Fig. 12 Two-dimensional max pooling diagram.

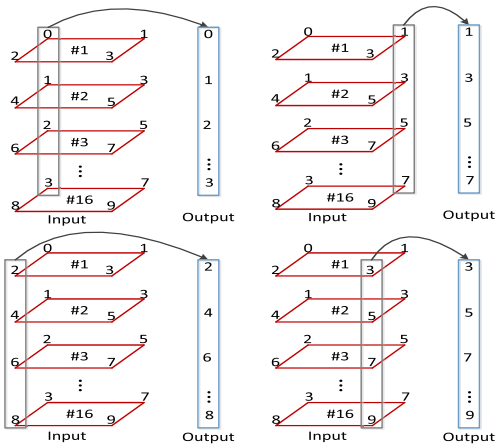

Fig. 13 Multidimensional max pooling operation.

multiple data (SIMD), and there are large number feature maps need to carry out the pooling operation, therefore, we use multidimensional max pooling operation to improve the parallel computation.

As the pooling layer located after the convolutional layer, in Sect. 5.2.1 we use 16 two-dimensional matrix convolution as an example for parallel analysis. Next, we take sixteen 2×2 max pooling as an example to illustrate the parallelization process of multidimensional max pooling based on the FT2000.

The max pooling vectorization of 16 output feature maps is carried out as shown in Fig. 13. The vector register Vin0 [i], $(1 \le i \le 16)$ initialized to zero is sequentially compared with the above vector loading results Vout[i] = (Vin0[i] > Vin1[i]) ? Vin0[i] : Vin1[i], $(1 \le i \le 16)$. This step not only calculates 16 max pooling results simultaneously, but also performs the ReLU by comparing with the register which initialized to zero, and it is equivalent to fusing activation function and max pooling to further optimize the convolutional neural network.

5.4 Local Response Normalization

Local response normalization was first proposed in AlexNet [24], which is mainly used to smooth the data

between multidimensional feature maps. In AlexNet, the recognition rate of the model is improved by using this operation, and LRN operation is often used in current large-scale convolutional neural network. GoogLeNet also uses LRN operation, and the LRN calculation formula is as follows:

$$b^i_{x,y} = a^i_{x,y} \Bigg/ \left(k + \alpha \sum_{j=\max(0,i-n/2)}^{\min(N-1,i+n/2)} \left(a^j_{x,y}\right)^2 \right)^{\beta} \tag{1}$$

where i is the result of the i-th feature map after ReLU at the position (x, y), n is the number of feature maps for the LRN operation at the same position, N is the total number of feature maps, and b is the result of the LRN operation. The dimension is the same as that before the LRN, and the parameters k, alpha, and beta are all super-parameters, generally, $k = 2$, alpha = 1e-4 and beta = 0.75.

Here we take n equal 5 and PE equal 16 as an example to illustrate the vectorization process of the LRN operation. Through analysis, we can decompose formula (1) as follows:

1. Calculate the square value of the corresponding position elements of the input feature maps: $V_1 = \left(a^j_{x,y}\right)^2$.

2. Calculate the square sum of the corresponding elements of adjacent n input feature maps: $V_2 = \sum_{j=\max(0,i-n/2)}^{\min(N-1,i+n/2)} V_1$.

3. Use multiply-add instructions to complete the calculation of the super-parameters: $V_3 = \alpha \times V_2 + k$.

4. Use vector exponent operation instructions to complete the reciprocal operation of beta power: $V_4 = \frac{1}{V_3{}^{\beta}}$.

5. Use the multiplication instruction to complete the calculation of the entire LRN layer: $V_5 = a^i_{x,y} \times V_4$.

Through vectorization, we can complete the LRN operation of different channel feature maps in parallel. FT2000 has 48 MAC units, so it can calculate 48 double-precision results or 96 single-precision results, if the 12-core is all used, it can calculate 576 double-precision elements and 1,152 single-precision elements at the same time. The implementation method also improves the execution speed of the convolutional neural network.

5.5 Multi-Core Implementation Analysis of the Inception Structure

GoogLeNet uses Inception structure to fuse different scale feature maps, which improves the recognition rate of the network model. The structure obtained the highest recognition rate in the ILSVRC in 2014, since the feature maps in the Inception structure are extracted from the different size convolutional kernels, such as 1×1, 3×3, 5×5. In order to ensure the same output feature size, the convolution operation uses a different pad operation and stride, so the calculation of the Inception structure is relatively complex, a typical Inception structure is shown in Fig. 5.

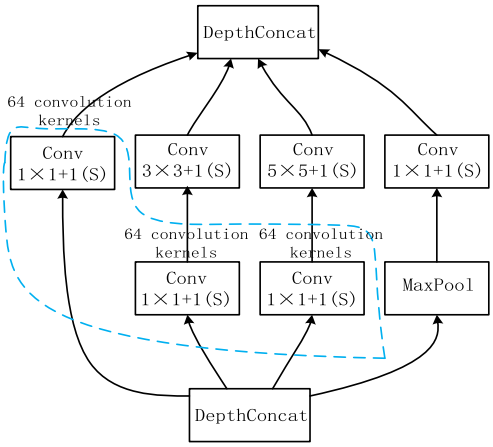

Fig. 14 First-level multi-core partition of the Inception structure.

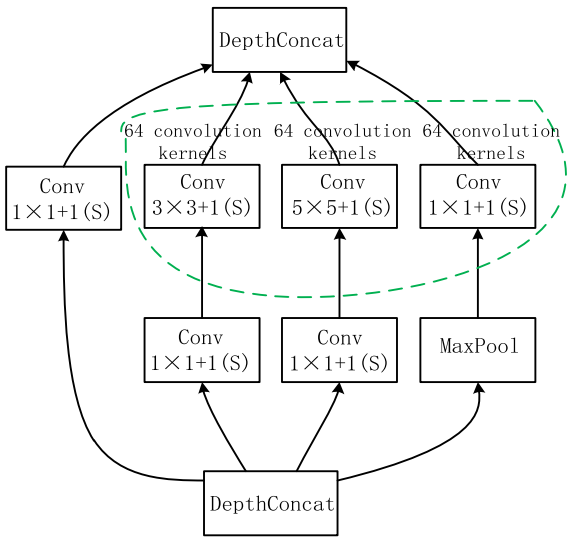

Fig. 15 Second-level multi-core partition of Inception structure.

The single-core parallelism for the various operations was analyzed in Sects. 5.1 to 5.4, and this section focuses on the multi-core parallelism of the Inception structure.

We know that in edge computing, the single-core performance of edge devices cannot be infinitely improved because of its area, power consumption, battery life, and other factors, so the multi-core processor is an important research direction in current embedded processor design. Integrating multiple computing cores into a single chip can decompose large computational tasks, and then, through multi-core parallel processing, the processor's computing speed can be improved. Through the analysis of the vectorization of multidimensional matrix convolution in Sect. 5.2, we know that the matrix convolution calculation can be done in parallel on a single-core vector processor. Therefore, in the multi-core implementation of the Inception structure, it is reasonable to assign the Inception structure calculation task to 12 computing cores of the FT2000 on average, because the multi-core parallel program computing time is determined by the

Table 2 Three kinds of platform parameters comparison

	Technology (nm)	Peak performance (GFlops)	Power (W)	Frequency (GHz)	Cores
CPU	32	118.4	65	4.3	4
GPU	16	11,500	235	1.5	3,584
FT2000	40	1,200	30	1	12

core with the longest computation time. Figure 14 is our first-level 12-core partition scheme for Inception structure. Suppose there are 192 convolutional kernels with the size 1×1, according to our single-core implementation scheme, the single-core can calculate 16 output maps at the same time. Therefore, we equally divide the 12-core computation tasks from the third dimension, namely $192/12 = 16$, then FT2000 can complete the calculation of 16 output feature maps on average for each core. As the tasks of the 12 cores are equally divided, the 192 output feature maps can be completed simultaneously.

Figure 14 shows that the first level of the Inception structure is all 1×1 convolutional kernel, so the division of multi-core computing tasks is relatively simple. As the first level calculation results need to be provided to the second level, and the second level convolutional kernel is 1×1, 3×3, and 5×5. The calculation type is not consistent, so the multi-core tasks division of the second-level cannot be completely equal, it is also assume that the number of convolutional kernels of 1×1, 3×3, and 5×5 is all 64, as shown in Fig. 15. To facilitate reuse single-core procedure, we try to put the similar calculation into one core in the multi-core task division as much as possible, therefore, we have sixty-four 1×1 convolution operations calculated by $64/16 = 4$ cores, sixty-four 3×3 convolution operations and sixty-four 5×5 convolution operations calculated by the respective 4 cores. Because the number of convolutional kernels is consistent, and the computation of 5×5 convolutional kernel is larger than the 1×1 and 3×3, so the total computation time of the second level is determined by the completion time of the 5×5 convolution operation. As in all Inception structures of GoogLeNet, the number of different convolutional kernel sizes will change, so in the multi-core Inception implementation process, it is difficult to achieve equal division of multicore tasks, as long as the division of computing tasks can be relatively balanced.

6. Experimental Results and Analysis

6.1 Experimental Setup

The comparison platform of this experiment is an AMD A10-6700 APU, 4-core, 4-thread, DDR3 8 GB memory, 3.7 GHz. GPU have been widely used to support Machine Learning in industry due to their speed advantage over SIMD CPU. Hence, we take a modern GPU card (NVIDIA GTX1080TI, 11.5 TFlops peak, 11 GB GDDR5, 484.4 GB/s memory bandwidth, 16nm technology, CUDA SDK8.0) as the baseline. The single core of FT2000 platform has a vector memory of 768 KB, scalar memory of 96 KB, and DDR can support up to 128 GB memory space, 1 GHz, three platforms implement the same five network models. CPU and GPU implementations are based on the Caffe implementation in Ubuntu16.04, and the test time is calculated by setting two times () functions. The power consumption of CPU and FT2000 is measured by the actual power consumption of the instrument through the power tester. The running power of GPU is observed in real time by nvidia-smi. feiteng platform using the CCS-based software development platform of FT2000 to complete five kinds of network model vectorization code compilation and performance statistics. Table 2 is the three kinds of platform-related parameters comparison.

Table 3 Main operational type of five neural network models

Models	AlexNet	VGG16/19	GoogLeNet	ResNet18
Layers	8	16/19	22	18
Data Aug.	+	+	+	+
Inception	–	–	+	–
Kernel size	11, 5, 3	3	7, 1, 3, 5	7, 1, 3, 5
Dropout	+	+	+	+
LRN	+	–	+	–

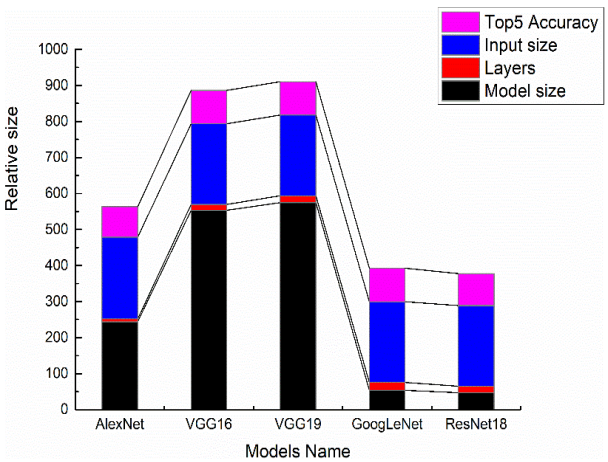

Fig. 16 Correlation parameters comparison of five network models

6.2 Benchmarks

In the experiment, we implement five main neural network models based on our vectorization method, namely AlexNet, VGG16, VGG19, GoogLeNet and ResNet18. The main operational types of this five neural network mod-

Table 4 Parameters information of GoogLeNet on FT2000.

Inception	Calculation (FLOPS)	Time (ms)	Weight (MB)	Cores	Utilization ratio of multi-core
Inception_1	257,453,056	1.43	3.2	11	91.67%
Inception_2	610,340,864	2.51	7.49	11	91.67%
Inception_3	148,063,104	0.61	6.09	11	91.67%
Inception_4	176,719,872	0.73	7.52	11	91.67%
Inception_5	200,603,648	0.84	8.49	12	100%
Inception_6	237,934,592	0.93	9.97	12	100%
Inception_7	340,923,968	1.42	14.9	12	100%
Inception_8	116,574,528	0.48	17.9	12	100%
Inception_9	141,762,880	0.59	24.7	12	100%

els are shown in Table 3. As can be seen from Table 3, GoogLeNet covers almost all types of calculations, and uses the Inception structure, so the neural network model is the most complicated. Therefore, the detailed design and implementation of GoogLeNet are carried out, and the same design method is used in other network models.

For embedded processors, model size, input image size, number of layers will also have an impact on the performance. In order to conduct a better performance analysis, we calculated the relevant parameters of the five neural network models as shown in Fig. 16. we can see that the five neural network models have the largest difference in model size, followed by number of layers, input size and accuracy is basically the same.

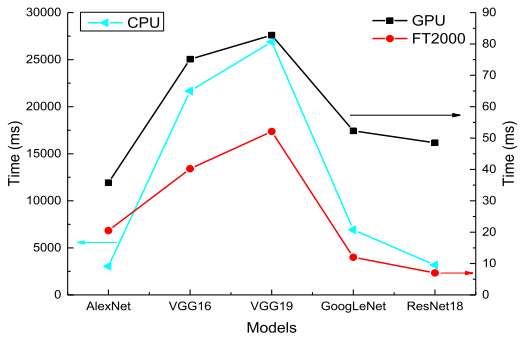

Fig. 17 Performance curves of five neural network models on CPU, GPU and FT2000

6.3 Performance Statistics and Analysis

In the implementation of GoogLeNet, because of the large-scale of the model and the number of layers, we mainly perform statistical analysis on the performance of 9 Inception structures.

Table 4 displays the main parameters information of GoogLeNet based on FT2000. The calculation counts the multiply-add operations that are involved in the Inception. Because the FT2000 processor needs to load hexadecimal values in the calculation process, we use MATLAB® convert the single-precision floating-point value into hexadecimal value through a script file, the weight parameter represents the size of hexadecimal weight file, computing time uses the corresponding FT2000 software development environment, and uses the clock () function to count the calculation time of each Inception by setting a breakpoint. As we can know that the parameters of different Inception layers are different due to the size and number of convolutional kernels. The smallest calculation amount is Inception_8, mainly because the upper layer of Inception_8 uses max pooling operation, the size of feature map changes from 14×14 to 7×7, which greatly reduces the number of calculation amount. The largest calculation amount is Inception_2, mainly because the number of input feature maps and output feature maps in Inception_2 are big, and the calculation of an output feature map involves the calculation of all input feature maps. The largest weight parameter is Inception_9, mainly because the number of input and output feature maps are very large. Through analysis of the calculation time, it can be found that the calculation amount and the calculation time are positively correlated. Furthermore, in the actual implementation process, due to the unequal division of multi-core tasks, it is necessary to use a little time for synchronous operation of multi-core and the data transmission time between intra-core and out-of-core. In the statistics of multi-core utilization, FT2000 can achieve an average utilization rate of 92.59% in the multi-core implementation of all Inception structures. Part of the Inception layers does not reach 12-core utilization, mainly because in the 12-core division of the Inception, only 11 cores can be assigned enough 16 weight matrix not 12 cores. According to division scheme of 16 weight matrix for a group, if all cores are forced to get the computing task, then, the task of some core will too much and some core will too few. Consequently, even if the multi-core utilization increases, and the total computing time will not necessarily decrease, while it increases the difficulty of programming.

Figure 17 is the execution time of five neural network models on CPU, GPU and FT2000, it can be seen that VGG19 has the longest execution time on CPU, GPU and FT2000, which are 26,881 ms, 82.2 ms, 52.1 ms respectively, the main reason is that the VGG19 model and parameters are the largest within the five neural network mod-

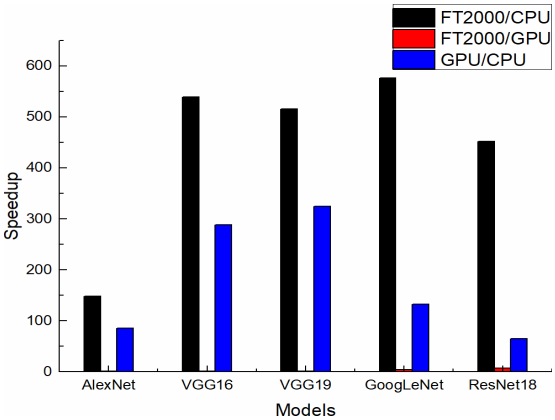

Fig. 18 Speedup of FT2000 over CPU, GPU and speed up of GPU over CPU

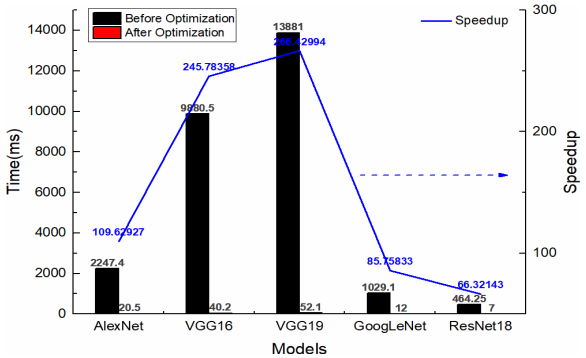

Fig. 19 Optimization of five kinds of neural network models based on FT2000

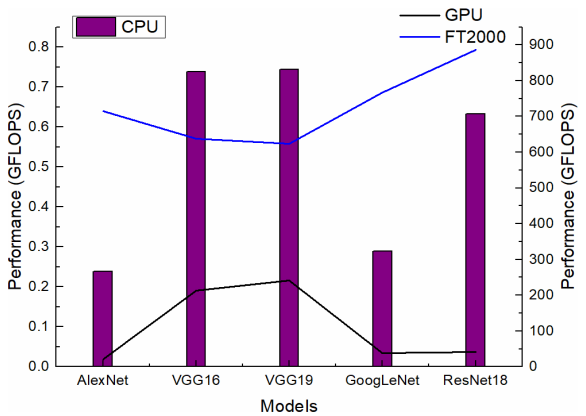

Fig. 20 Measured performance of five neural network models on CPU, GPU and FT2000

els, as shown in Fig. 16. While the AlexNet runs least time on the CPU and GPU, 3,047.4 ms and 35.75 ms respectively. ResNet18 has a minimum execution time of 7 ms on FT2000, though AlexNet model size (243.9 MB) is larger than ResNet18 (46.8 MB), but running less time on CPU and GPU than on ResNet18. We know that the number layer of ResNet18 (18) is greater than AlexNet (8), indicating that the layer has a higher performance impact on the CPU and GPU than model size. But for embedded processors, the effect of chip storage on its performance is obviously higher than other factors, because the larger the model size, the more frequent of data movement inside and outside the chip is needed, which has a great effect on the performance.

Figure 18 shows the speedup of FT2000 compared with CPU, GPU and GPU compared with CPU. It can be seen that FT2000 and GPU have the highest speedup of 539.3× and 324.7× relative to CPU, the average speedup is 446.5× and 179.2×. FT2000 relative GPU achieved a maximum speedup of 6.9× and a minimum speedup of 1.58×, the average speedup is 3.30×. As the single-precision peak performance of GPU is about five times that of FT2000, with 3,584 computing cores, 11GB memory, is a powerful performance server level acceleration card, therefore, FT2000 does not get higher speedup. (Speedup (FT2000/CPU) is the ratio of the model's execution time on the CPU to the execution time on the FT2000, likewise, speedup ratio of FT2000 relative to the GPU and GPU relative to the CPU calculates in the same way).

Figure 19 is the basic implementation, optimization implementation and speedup of AlexNet, VGG16, VGG19, GoogLeNet and ResNet18 in FT2000. The basic implementation is based on general convolution realization method and multi-core partitioning method, while the optimization implementation is achieved by using the vectorization method and multi-core partitioning scheme which is proposed in this paper. It can be seen that by using the vectorization method proposed in this paper, we can get the maxmum 266.4×, the minimum 66.3× and average speedup of 154.8×.

The following are performance and computational efficiency analysis about five kinds of neural network model on CPU, GPU and FT2000, performance formula is defined as:

$$R = \frac{O}{t} \tag{2}$$

The efficiency formula is defined as:

$$E = R \times \frac{1}{p_k} \tag{3}$$

where R represents the measured performance, O represents total floating-point operand of a model, unit is GFlops, t represents the actual execution time of the model, unit is s, P_k represents peak performance of processor, and E represents the processor's computational efficiency.

As can be seen from Fig. 20, the measured performance of CPU in the five neural network models is not high, the highest performance in the VGG19 is 0.744 GFlops, the lowest performance in AlexNet is 0.239 GFlops, and the average measured performance is 0.528 GFlops. Similarly, the measured performance of GPU on VGG19 is the highest, which is 241.55 GFlops, the lowest measured performance on AlexNet is 20.34 GFlops, and the average measurement performance is 110.85 GFlops. The highest measured performance of FT2000 on ResNet18 is 885.714 GFlops, the lowest measured performance on VGG19 is 623.88 GFlops,

and the average measured performance is 725.78 GFlops. Through the analysis of the five neural network models, we can see that the bigger the model size, the higher the measured performance on CPU and GPU, such as VGG16 and VGG19, the smaller the model size, the lower the performance on CPU and GPU, such as GoogLeNet and ResNet18. While for the FT2000 with limited on-chip resources, the larger the model size, the lower the performance, such as the VGG19, so for embedded processors we need smaller neural network models such as Squeezenet, mobilenet, etc.

For a more fair comparison, we count the efficiency of the five neural network models AlexNet, VGG16, VGG19, GoogLeNet and ResNet18 on the CPU, GPU and FT2000. Figure 21 shows that the CPU achieves a maximum efficiency of 0.63% on VGG19 model, a minimum of 0.20% on AlexNet and the average computational efficiency is 0.45%. While the maximum computational efficiency on the GPU is 2.1%, the minimum computational efficiency is 0.17%, and the average computational efficiency is 0.96%. The maximum computational efficiency of the FT2000 is 73.8%, the minimum computational efficiency is 51.99%, and the average computational efficiency is 60.48%. By contrast, we found that the FT2000 achieved high computational efficiency compared with CPU and GPU, but the average computational efficiency is only 60.48% and there is still much room for improvement. The reason why the calculation efficiency is not high on a large neural network like VGG19 is that the memory in the FT2000 is too small and only 768 KB at present, therefore, when the weight matrix of the model is too big, the FT2000 can only by DMA moving intermediate calculation results to extra-core DDRs, the frequent data migration affects the computational efficiency of FT2000. So the next step is to increase the FT2000's internal memory. The computational efficiency of CPU is low because it is mainly oriented to traditional logical computing, but it is difficult to obtain higher computational efficiency for the new convolutional neural network. Although GPU has large numbers of computing cores, big storage and as a mainstream neural network training platform. However, it will encounter a low computational load on the edge device, which leads to waste of computational resources, because the edge device is mainly used for inference, and there is also large computational power consumption. Therefore, we propose a new edge-oriented architecture FT2000, which is expected to provide a new reference for the research of current neural network accelerator.

Table 5 statistics the power consumption of CPU, GPU and FT2000 in running five kinds of neural network models. The static power consumption is CPU, GPU and FT2000 power consumption in standby state, and the running power is test the average power consumption of 20 images, actual power is the running power minus the static power. The power of CPU and FT2000 obtained through the power tester, shown in Fig. 22, GPU power consumption is by using nvidia-smi to real-time statistics. What needs to be explained is that FT2000 is a board, so the fans and various

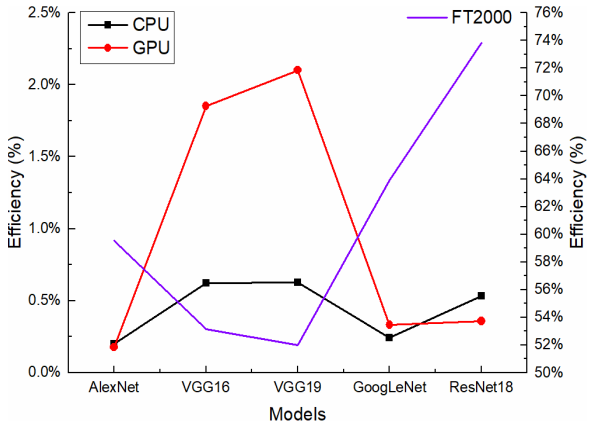

Fig. 21 Implementation efficiency of five neural network models on CPU, GPU and FT2000

Table 5 Performance comparison and analysis of 3 platforms

Platforms	CPU	GPU	FT2000
Static power (W)	45	19	20
Running power (W)	99.2	132	21.6
Actual power (W)	54.22	113	1.6

Fig. 22 Power tester

interfaces on it increased static power consumption, while by testing, you can see the average power consumption is only 1.6 W.

In addition, for edge processors, the price is also an important factor affecting large-scale deployment of edge devices. The price of GPU is too expensive for edge computing. Therefore, the use of FT2000 for edge computing is worth considering, no matter from performance, power, or price.

7. Conclusion

In this paper, a microprocessor architecture for edge computing is presented, combined with five deep learning applications. We presented a detailed description of the sophisticated implementation process of GoogLeNet, from the aspects of feature data layout, multidimensional matrix convolution, multidimensional max pooling, LRN, ReLU activation function, etc. Different vectorization schemes are

proposed for different calculation models and extended to other neural network models. Finally, the performance comparison and analysis based on multi-core CPU and high-performance GPU are carried out, the experimental results show that the FT2000 vector processor architecture can well adapt to the complex neural network model, and provides a new reference for the current edge devices in deep learning applications. FT2000 vector processor is a general-purpose microprocessor, and related research has shown that 16-bit or even 8-bit can achieve better recognition accuracy. Therefore, our next step is based on the vector processor and by reducing the calculation precision, increasing the scale vector broadcast and core memory to further explore microprocessor architecture that adapts to edge computing.

Acknowledgments

We would like to thank the project of (2016YFB0200401) the National Key Research and Development Program of China and (60133007, 61572025) the National Natural Science Foundation of China.

References

[1] W. Shi, H. Sun, J. Cao, et al., "Edge Computing-An Emerging Computing Model for the Internet of Everything Era," Journal of Computer Research and Development, vol.54, no.5, pp.907–924, 2017 (in Chinese).
[2] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge Computing: Vision and Challenges," IEEE Internet Things J. vol.3, no.5, pp.637–646, 2016.
[3] W. Shi and S. Dustdar, "The Promise of Edge Computing," Computer, vol.49, no.5, pp.78–81, 2016.
[4] E.C. David, "The once and future Internet of everything," [2016-12-03], http://sites.nationalacademies.org/cs/groups/cstbsite/documents/webpage/cstb_160416.pdf
[5] V. Turner, J.F. Gantz, D. Reinsel, et al., "The digital universe of opportunities: Rich data and the increasing value of the Internet of things," [2016-12-03], https://www.emc.com/collateral/analyst-reports/ide-digital-universe-2014.pdf
[6] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A.C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," International Journal of Computer Vision, vol.115, no.3, pp.211–252, 2015.
[7] A. Graves, A.-R. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," IEEE International Conference on Acoustics, Speech and Signal Processing, IEEE, pp.6645–6649, 2013.
[8] Z. Liu, Y. Li, F. Ren, et al., "A Binary Convolutional Encoder-decoder Network for Real-time Natural Scene Text Processing," arXiv preprint arXiv:1612.03630, 2016.
[9] W. Liu, D. Anguelov, D. Erhan, et al., "SSD: Single Shot MultiBox Detector," European Conference on Computer Vision, pp.21–37, 2016.
[10] A. Milan, S.H. Rezatofighi, A.R. Dick, et al., "Online Multi-Target Tracking Using Recurrent Neural Networks," AAAI Conference on Artificial Intelligence, pp.4225–4232, 2017.
[11] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," Proceedings of the 22nd ACM International Conference on Multimedia, pp.675–678, 2014.
[12] M. Abadi, P. Barham, J. Chen, et al., "TensorFlow: a system for large-scale machine learning," Operating Systems Design and Implementation, pp.265–283, 2016.
[13] F. Nasse, C. Thurau, and G.A. Fink, "Face Detection Using GPU-Based Convolutional Neural Networks," Lecture Notes in Computer Science, vol.5702, pp.83–90, 2009.
[14] S. Potluri, A. Fasih, L.K. Vutukuru, et al., "CNN -Based High Performance Computing for Real Time Image Processing on GPU," The Workshop on Nonlinear Dynamics & Synchronization & Intl Symposium on Theoretical -Electrical Engineering, pp.1–7, 2011.
[15] D.H. Yoon, D.H. Yoon, D.H. Yoon, et al., "In-Datacenter Performance Analysis of a Tensor Processing Unit," Proceedings of the 44th Annual International Symposium on Computer Architecture, pp.1–12, 2017.
[16] Z. Du, R. Fasthuber, T. Chen, et al., "ShiDianNao:shifting vision processing closer to the sensor," ACM Sigarch Computer Architecture News, vol.43, no.3, pp.92–104, 2015.
[17] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," IEEE Computer Vision and Pattern Recognition, pp.1–9, 2015.
[18] C. Szegedy, V. Vanhoucke, S. Ioffe, et al., "Rethinking the inception architecture for computer vision," Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp.2818–2826, 2016.
[19] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," Computer Science, 2014.
[20] K. He, X. Zhang, S. Ren, et al., "Deep Residual Learning for Image Recognition," Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp.770–778, 2016.
[21] Y. Chen, T. Luo, S. Liu, et al., "DaDianNao: A machine-learning supercomputer," Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, pp.609–622, 2014.
[22] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon, An Instruction -Set Architecture for Neural Networks," Acm Sigarch Computer Architecture News, vol.44, no.3, pp.393–405, 2016.
[23] T. Chen, Z. Du, N. Sun, et al., "DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning," ACM Sigplan Notices, vol.49, no.4, pp.269–284, 2014.
[24] A. Krizhevsky, I. Sutskever, and G.E. Hinton, "ImageNet classification with deep convolutional neural networks," International Conference on Neural Information Processing Systems, pp.1097–1105, 2012.

Junyang Zhang is an IEICE member, received the Master degree in 2014, both from National University of Defense Technology. He is currently a Ph.D. student of National University of Defense Technology. His research interests include artificial intelligence, deep learning and artificial intelligence chip.

Yang Guo was born in 1971. Ph.D., professor and PhD supervisor at the college of computer, National University of Defense Technology. Member of CCF. His main research interests include microprocessor design, verification, and artificial intelligence chip.

Xiao Hu was born in 1977. Ph.D., associate researcher at the college of computer, National University of Defense Technology. His main research interests include digital signal processor architecture and artificial intelligence chip.

Rongzhen Li received the Master degree in 2013, both from National University of Defense Technology. He is currently a Ph.D. student of National University of Defense Technology. His research interests include parallel and distributed system and cloud computing.