Accelerating Large-Scale Deep Convolutional Neural Networks on Multi-core Vector Accelerators

Zhong Liu, Sheng Ma, Cheng Li, Haiyan Chen

Accelerating Large-Scale Deep Convolutional Neural Networks on Multi-core Vector Accelerators*

Zhong Liu✉, Sheng Ma✉, Cheng Li, and Haiyan Chen

College of Computer, National University of Defense Technology, Changsha, China
{zhongliu,masheng}@nudt.edu.cn, 1065202842@qq.com, hychen608@163.com

Abstract. This paper proposes an efficient algorithm mapping method for accelerating deep convolutional neural networks, which includes: (1) Proposing an efficient transformation method, which converts CNN's convolutional layer and fully connected layer computations into efficient large-scale matrix multiplication computations, and converts pooling layer computations into efficient matrix row computations; (2) Designing a set of general and efficient vectorization method for convolutional layer, fully connected layer and pooling layer on the vector accelerator. The experimental results on the accelerator show that the average computing efficiency of convolution layer and full connected layer of AlexNet, VGG-19, GoogleNet and ResNet-50 are 93.3% and 93.4% respectively, and the average data access efficiency of pooling layer is 70%.

Keywords: Multi-core Vector Accelerators · Convolutional Neural Network · Vectorization · AlexNet · VGG · GoogleNet · ResNet.

1 Introduction

Currently, more and more deep learning applications are being deployed to take advantage of the powerful computing power of supercomputers. For example, scientists from Princeton Plasma Physics Laboratory are leading an Aurora ESP project[1] that will leverage AI and exascale computing power to advance fusion energy research. Patton demonstrated a software framework that utilizes high performance computing to automate the design of deep learning networks to analyze cancer pathology images[2].The award-winning project[3] developed an exascale deep learning application on the Summit supercomputer.

We design an efficient scientific computing accelerator for building a prototype supercomputer system. It leverages a multi-core vector architecture for high-density computing. The accelerator achieves high computational efficiency

* Corresponding author: Zhong Liu (Email:zhongliu@nudt.edu.cn) and Sheng Ma (Email:masheng@nudt.edu.cn. This work is supported by the National Natural Science Foundation of China (No. 61572025).

in large-scale scientific computing applications, such as solving dense linear equations[4, 5]. Since deep learning is currently more and more widely used, and CNN is one of the representative algorithms of deep learning, it is very necessary to study how to improve the accelerator's performance in processing deep CNNs.

2 Background and related work

2.1 The Architecture of vector accelerator

Our proposed scientific computing accelerator is a high performance floating-point multi-core vector accelerator developed for high-density computing. The accelerator integrates 24 vector accelerator cores, and achieves a peak single-precision floating-point performance of 9.2TFLOPS with a 2GHz frequency.

The architecture of our proposed accelerator is shown in Fig. 1. The accelerator core consists of a Scalar Processing Unit (SPU) and a Vector Processing Unit (VPU). The SPU is responsible for scalar computing and flow control and provides a broadcast instruction to broadcast the data from the scalar register to the vector register of the VPU. The VPU integrates 16 Vector Processing Elements (VPEs), and provides the main computation capability.

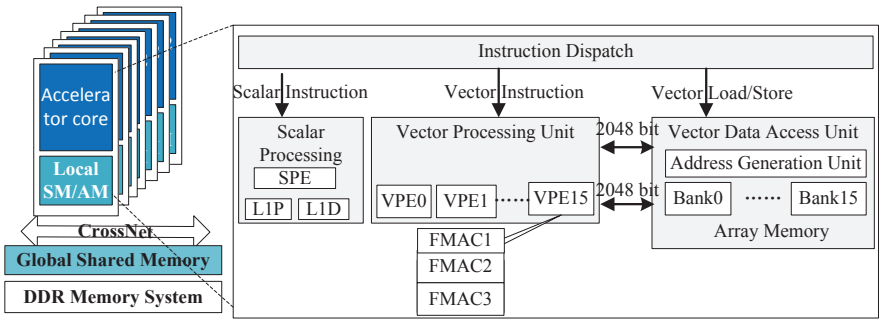

Fig. 1. The Architecture of vector accelerator.

2.2 Related work

To accelerate and optimize the computation of CNN, Maji et al. [6] propose a fast CNN optimization method, which takes advantage of the inherent redundancy in the convolutional layer to reduce the computational complexity of deep networks. Lin et al. [7] propose a novel global and dynamic pruning method to accelerate the CNN by pruning the redundant filter in the global range. Abtahi et al. [8] study the method of using FFT to accelerate CNN on embedded hardware. Dominik et al. [9] study the method of accelerating large-scale CNN with parallel

graphics multiprocessor. Lee et al. [10] study how to accelerate the training of deep CNNs by overlapping computation and communication.

To solve the problem that the CPU cannot meet the computational performance requirements of deep CNNs, a large number of researchers have studied how to accelerate the computation of CNNs on emerging processor platforms, such as TPU[11], GPU[12], FPGA[13, 14], NPU[15], embedded devices[16], ASIC[17], etc.

The vector accelerator is a novel architecture that has powerful computing performance while maintaining low power consumption. Some researches show that the vector accelerator has high computational efficiency in dealing with FFT[18] and matrix multiplication[5], and is suitable for accelerating large-scale CNN. Researchers also propose some vectorization methods of convolution computation on vector accelerators [19, 20]. They use the method of loading weight into vector array memory and loading input image feature data into scalar storage to complete the convolution computation. The common disadvantages are: (1) The weight cannot be effectively shared, the memory bandwidth is wasted, and the computing efficiency of the vector accelerator cannot be fully utilized; (2) Because the size of the channel dimension is uncertain, and it does not match the number of processing units of the vector accelerator; 3) Different CNN models have different channel dimensional sizes for different convolutional layers, which greatly affects the efficiency of loading data, and it is not universal; (4) It needs hardware support of reduce or shuffle and the hardware cost is high.

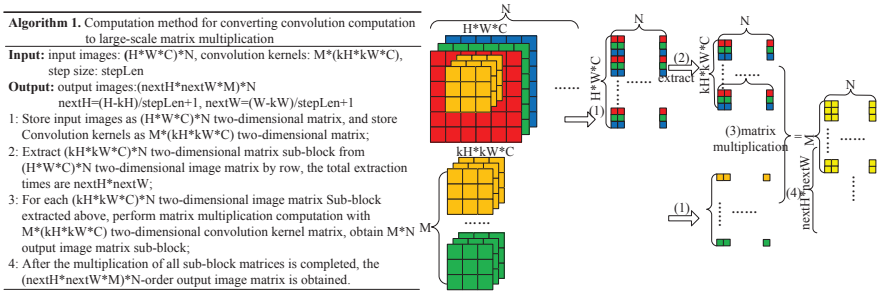

Fig. 2. Computation method for converting convolution computation to large-scale matrix multiplication.

3 Algorithm mapping of CNN on vector accelerator

3.1 Computation method for converting convolution operations to large-scale matrix multiplication

As shown in Fig. 2, we propose a computation method for converting convolution computation to large-scale matrix multiplication. It converts a large number

of inefficient small-scale convolution products into a large-scale matrix-matrix multiplications, which is very suitable for vectorization computations and can significantly improve the computational efficiency.

Suppose convolution computations are performed on N input images of size H*W*C with M convolution kernels of size kH*kW*C. First, store input images as a (H*W*C)*N two-dimensional matrix, and store convolution kernels as a M*(kH*kW*C) two-dimensional matrix. Second, extract (kH*kW*C) rows from input images matrix according to the convolution window, which form a new (kH*kW*C)*N matrix. Finally, the M*(kH*kW*C) convolution kernel matrix and the new (kH*kW*C)*N input image matrix are used to perform a large-scale matrix-matrix multiplication, and the computed result is M*N output images matrix. Therefore, the original M*N small convolution operations each with the size of kH*kW*C are transformed into a large-scale matrix-matrix multiplication with a M*(kH*kW*C) convolution kernel matrix and a (kH*kW*C)*N input image matrix.

3.2 Vectorization method of Valid convolution layer

Based on the proposed conversion method, the computation of convolution layer is converted into matrix multiplication, which is defined as follows:

$$O[m,n] = \sum_{v=0}^{V-1} F[m,v] * I[v,n] + b[n] \tag{1}$$

The above formula can be regarded as the matrix multiplication of an M*V matrix and an V*N matrix, and the result is then added to the bias vector, where $n \in [0, N)$ and $m \in [0, M)$.

Algorithm 2. The vectorization method of valid convolution layer

Input: input images matrix I: S*N, convolution kernels F:M*V, bias vector b:M*1, S=H*W*C, V=kH*kW*C, step size: stepLen
Output: output images matrix O: (nextH*nextW*M)*N
nextH=(H-kH)/stepLen+1, nextW=(W-kW)/stepLen+1

```
1:for h=0 to nextH-1
2:    for w=0 to nextW-1
3:        Set up an input image buffer B of size (kH*kW*C)*N
4:        Let the starting row position for extraction is:
          (h*W+w)*stepLen*C
5:        for k=0 to kH-1
6:            Continuously extract kW*C rows to B from the input
              image matrix
7:            Move down the extraction position W*C rows
8:        end
9:        Perform matrix multiplication computatIon on the
          convolution kernel matrix F and the above extracted matrix
          B and the bias vector b
10:   end
11:end
```

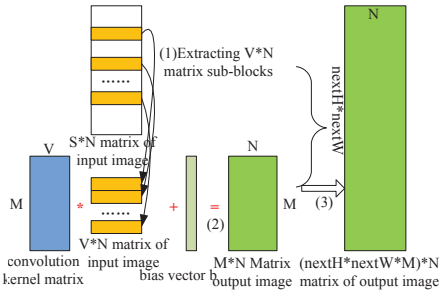

Fig. 3. The vectorization method of valid convolution layer.

As shown in Fig. 3, we can get the vectorization method of the valid convolution layer based on matrix multiplication. The M*V matrix is a convolution

kernel matrix, it is stored in a kernel-dimension-last manner, and is continuously stored in the vector accelerator's off-chip memory as a two-dimensional matrix. Each row of the above M*V matrix stores a single convolution kernel, and the storage order in this row is the channel dimension first, followed by the convolution kernel width dimension, and finally the convolution kernel height dimension. The V*N matrix is a matrix sub-block consisting of V rows extracted from the S*N two-dimensional matrix of input image by row. Each column of the above S*N matrix stores a single input image, and the storage order in this column is the channel dimension first, then the image width dimension, and finally the image height dimension.

3.3 Vectorization method of Same convolution layer

Usually, the same convolution always applies a new memory area in advance according to the new image size, fills the 0 element area with 0, and copies the element value of the previous image in other areas. Then, the same method as valid convolution is used to perform convolution computation on the new image. The disadvantages of this method are: (1) At least double the memory overhead; (2) The memory location of the 0 element is discontinuous, resulting in a large operation overhead of padding; (3) It takes a lot of time for copying the original image data.

As shown in Fig. 4, we propose a vectorization method for the same convolution layer, which logically has a "padding 0" operation in the computation process, but does not physically require a "padding 0" operation. The same convolution computation determines whether the input image matrix row corresponding to the convolution kernel data is a 0-element row according to the element value of the vector Z.

3.4 Vectorization method of pooling layer

The computation of the pooling layer includes two steps. The first step is to extract the V*N matrix from the S*N input image matrix by row into the AM of the vector accelerator. The matrix size corresponding to each kernel is V*p, and the total number of extractions is nextH*nextW, where p is the number of VPE, nextH=(H-kH)/stepLen+1, nextW=(W-kW)/stepLen+1. The second step is to perform the following vectorization computation of pooling layer for each matrix sub-block extracted.

The algorithm is shown in Fig. 5. As can be seen, all computations and data accesses are performed along the row dimension, which has high memory access efficiency and is easy to be vectorized. At the same time, the computation of the maximum value or the average value is performed in the same VPE without the need for shuffling operations, which is convenient for software pipeline and high computation efficiency.

Algorithm 3. The vectorization method of same convolution layer

Input: input images matrix I: S*N, convolution kernels F:M*V, bias vector b:M*1, S=H*W*C, V=kH*kW*C, step size: stepLen padding:pad=((stepLen-1)H-stepLen+kH)/2.
Output: output images matrix O:(H*W*M)*N
1:for h=0 to H-1
2: for w=0 to W-1
3: Construct a vector Z of length V to record whether the image row corresponding to the convolution kernel element is a 0 element row.
4: Set up an input image buffer B of size (kH*kW*C)*N
5: for k=0 to kH-1
6: Determine whether (h>(pad-1) && (w+k)>(pad-1) && h<(H+pad) && (w+k)<(W+pad)) is true;
7: If yes, continuously extract the C rows from the ((h-pad) *W+w-pad+k)*C-th row of input image to B; at the same time, set the continuous C elements starting at the (h-pad)*W+w-pad+k position of the vector Z to 1;
8: If not, do not extract rows from the input image; at the same time, set the continuous C elements at the (h-pad)*W+w-pad+k position of the vector Z to 0;
9: end
10: Perform matrix multiplication computation on the convolution kernel matrix F, the above extracted matrix B and the bias vector b
11: end
12:end

Fig. 4. The vectorization method of same convolution layer.

3.5 Vectorization method of fully connected layer

The computation of the fully connected layer can be regarded as the product of an M*U matrix and a U*N matrix, and the result of the product is then added to the bias vector. Where the size of the weight matrix of the fully connected layer is M*U, and it is continuously stored in the off-chip memory of the vector accelerator in the input-feature-dimension-first-manner. The size of input image matrix is U*N, and it is continuously stored in the off-chip memory of the vector accelerator in the image-dimension-first-manner. Therefore, the original N matrix-vector multiplications of M*U matrix and a U vector are transformed into a large-scale matrix-matrix multiplication with M*U weight matrix and U*N input image matrix.

Algorithm 4. The vectorization method of max pooling layer

Input: input images matrix I: S*N, convolution kernels F:kH*kW, S=H*W*C, V=kH*kW*C, step size:stepLen
Output: output images matrix O:(nextH*nextW*M)*N nextH=(H-kH)/stepLen+1, nextW=(W-kW)/stepLen+1

```
1:for h=0 to nextH-1
2:    for w=0 to nextW-1
3:        Set up an input image buffer B of size (kH*kW*C)*N in AM
4:        Let the starting line position for extraction is:
          (h*W+w)*stepLen*C
5:        for k=0 to kH-1
6:            Continuously extract kW*C rows to B from the input
              image matrix
7:            Move down the extraction position W*C rows
8:        end
9:        For matrix B extracted above, take one row for each C rows,
          and take KH * kW rows in total; Use the vector comparison
          instruction to compare the kH*kW rows data row by row to
          obtain the row result of the maximum value.
10:       The above comparison cycle is performed C times, and obtain
          a C*N result matrix, which is transmitted to DDR in order.
11:   end
12:end
```

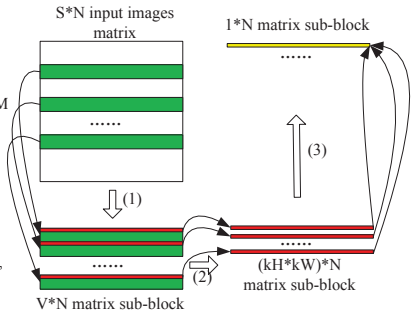

Fig. 5. The vectorization method of pooling layer.

4 Experimental results and performance analysis

We conduct the experiments on a 24-core vector accelerator. Its peak single-precision floating-point performance is 9.2TFLOPS at a 2 GHz frequency, and its the peak memory bandwidth is 307GB/s. Because the VPE length of accelerator is 16, the batch size is set to an integer multiple of 16, which is is suitable for vectorization calculation. In our experiments, the value of batch size is 96.

4.1 Performance of convolution layers

We evaluate the performance of our proposed design with several typical modern CNN workloads, including the AlexNet, VGG-19, GoogleNet, and ResNet-50 network models.

Fig. 6(a) shows the computational efficiency of 6 different types of convolution layer of GoogleNet's Inception (4e) module. The lowest efficiency is seen for the #5x5 reduce convolution layer, which is 62.8%. Yet, the computation operation of this type of layer accounts for only 2% of the total operations. Thus, it only has minor effect on the overall efficiency. The computational efficiency of the other 5 types of convolution layers exceeds 90%, which makes the weighted computational efficiency of the Inception (4e) module reach 93.2%.

Fig. 6(b) shows the computational efficiency of the 5 different types of convolution layers of ResNet-50's block-2 module. The lowest efficiency is seen for the conv2 convolution layer as 84.1%, and the highest one is seen for the conv4 convolution layer as 93.7%. The weighted computational efficiency of block-2 modules reaches 88%.

Fig. 7(a) shows the computational efficiency of all types of convolution layers for AlexNet, VGG-19, GoogleNet, and ResNet-50. The computational efficiency

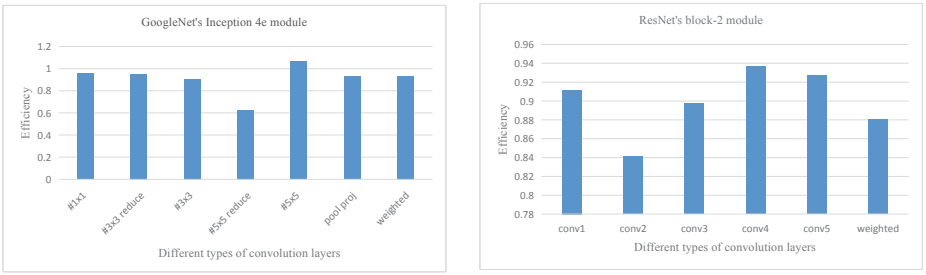

Fig. 6. The computational efficiency of GoogleNet's Inception(4e) and ResNet's block2.

of the 5 convolution layers of AlexNet is generally high, and its total weighted computational efficiency is as high as 103.9%. The first convolution layer of VGG-19 has the lowest efficiency. It is due to that the convolution kernel of this layer is the smallest one as 3x3x3, making the computational efficiency of this layer to be only 37.6%. However, the computation operations of this layer accounts for only 0.44% of the total operations, which does not affect the overall computational efficiency of VGG-19, and the total weighted computational efficiency reaches 93.5%. Similarly, the computational efficiency of the first convolution layer of ResNet-50 is 63.3%. Thus, the computation operations of this layer accounts only for 3.1% of the total operations, and its weighted computational efficiency reaches 87.9%. The computational efficiency of the first two layers of GoogleNet is 63.3% and 59.6% respectively, and the proportion of computation operands of the two layers is 5.1% and 0.5% respectively, and the total weighted computational efficiency reaches 88.2%.

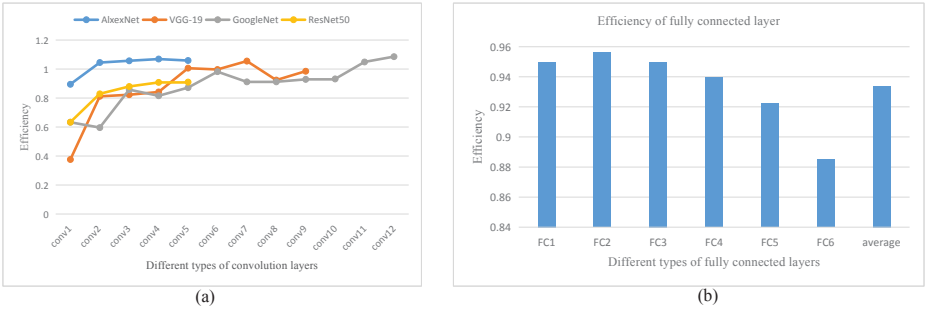

Fig. 7. The computational efficiency of all types of convolution layers and fully connected layer for AlexNet, VGG-19, GoogleNet, and ResNet-50.

Parts of the convolution layers achieve more than 100% computational efficiency because these convolution layers are Same convolution layers. These

same convolution layers expand the scale of calculation matrix and increase the floating-point operations of convolution layer by padding 0 elements. However, the proposed vectorization algorithm of the Same convolution layer records all 0-element rows by setting up a Z vector, and omits the multiplication and addition calculation of 0-element rows. Therefore, the vectorization algorithm of Same convolution layer proposed in this paper not only gains the advantages of Valid convolution layer method, but also improves the computational performance by reducing the multiplication and addition calculation of 0-element rows, which makes the computational efficiency exceeding 100%.

In general, all convolution layers of the four network models have achieved high computational efficiency. A few convolution layers with small convolution kernels have lower computational efficiency, but the proportion of the computation operands of the layers is relatively small, which makes the weighted computational efficiency of the four network models very high.

4.2 Performance of fully connected layers

The AlexNet, VGG-19, GoogleNet, and ResNet-50 all contain fully connected layers. AlexNet and VGG-19 have 3 fully connected layers, while GoogleNet and ResNet-50 only have one fully connected layer. As shown in Table 1, it is divided into 6 types according to the size of weight matrix.

Table 1. The six different types of fully connected layers.

Classification	Weight matrix	Network model
FC1	4096x9216	AlexNet
FC2	4096x25088	VGG-19
FC3	4096x4096	AlexNet,VGG-19
FC4	1000x4096	AlexNet,VGG-19
FC5	1000x2048	GoogleNet
FC6	1000x1024	ResNet-50

Fig. 7(b) shows the computational efficiency of the 6 types of fully connected layers. The computational efficiency of the fully connected layer is high because the matrix size of fully connected layers is large. The smallest weight matrix size is 1000*1024, and its computational efficiency is 88.5%. The largest weight matrix size is 4096*25088 and its computational efficiency is 95.6%. The average computational efficiency is 93.4%.

4.3 Performance of pooling layers

The AlexNet, VGG-19, GoogleNet, and ResNet-50 contain 16 different types of pooling layers. Fig. 8(a) shows the computational efficiency and data access

efficiency of 16 different types of pooling layers, where the left ordinate is used to identify the data access efficiency, and the right ordinate is used to identify the computational efficiency. It can be seen from the figure that the computational efficiency of the pooling layer is low, about 0.22% on average, because the computation density of the pooling layer is very low. However, the data access efficiency of the pooling layer is very good, about 70% on average. This is because all data access of the pooling layer is performed in matrix rows according to the method proposed in this paper, and the data access efficiency is high.

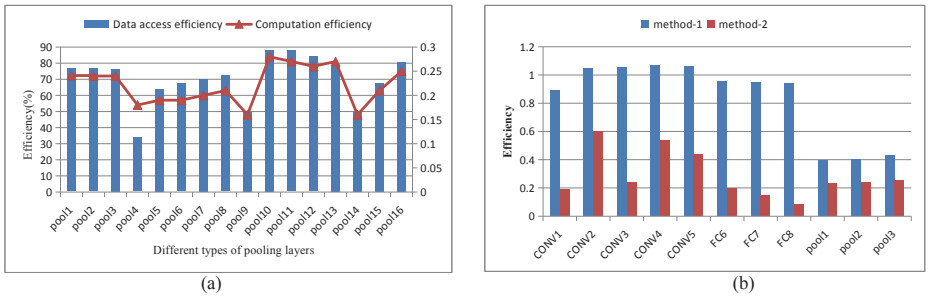

Fig. 8. (a)The computational efficiency of pooling layer for AlexNet, VGG-19, GoogleNet and ResNet-50. (b) Performance comparison of different vectorization method of AlexNet.

4.4 Performance comparison

As shown in Fig. 8(b), we take AlexNet as an example to compare the computational performance of the convolution layer, fully connected layer and pooling layer of different vectorization methods, where the performance data of pooling layer is data access efficiency. The method-1 is the vectorization method proposed in this paper, in which input images are stored in the image-dimension-first manner; and the method-2 is the vectorization method in which input images are stored in the channel-dimension-first manner. As can be seen from the figure, the method-1 significantly improves the computational efficiency of convolution layer and fully connected layer, and the data access efficiency of pooling layer of AlexNet.

Table 2 compares the CNN inference performance of our accelerator with the other three high-performance GPUs[21]. As can be seen from the table, NVIDIA V100 achieves the best performance by images/s, followed by our accelerator. However, NVIDIA V100 has the best performance only on the AlexNet by images/TFLOPS, our accelerator achieved the best performance on the GoogleNet and ResNet-50 by images/TFLOPS.

Table 2. Comparison of CNN inference performance.

Classification	AlexNet		VGG-19		GoogleNet		ResNet-50	
	images /s	images /TFLOPS	images /s	images /TFLOPS	images /s	images /TFLOPS	images /s	images /TFLOPS
our accelerator	3777	410	279	31	2679	291	1183	129
NVIDIA V100	8700	621	500	36	4000	286	1700	121
NVIDIA T4	2600	321	100	12	1100	136	500	62
NVIDIA P4	2600	473	100	18	1100	200	500	91

5 Conclusion

This paper analyzes the computational characteristics of feature images and convolution kernels in typical CNN models such as AlexNet, VGG, GoogleNet, and ResNet. A general method to accelerate the computation of deep CNNs is proposed according to the architectural characteristics of the multi-core vector accelerator. It is worth noting that this method does not optimize a specific CNN model, but it is general suitable for various typical deep CNN models. Experimental results show that the method proposed in this paper can take full advantage of the parallel computing advantages of multi-core vector accelerators and has high computing efficiency. It can accelerate the computation of deep CNNs

References

1. Aurora ESP Projects, https://www.alcf.anl.gov/science/projects/Aurora ESP/all. Last accessed 24 Aug 2020
2. Patton R M, Johnston J T, Young S R, Schuman C D, Potok T E, Rose D C, Lim S H, Chae J. Exascale Deep Learning to Accelerate Cancer Research[J]. 2019.
3. Kurth T, Treichler S, Romero J, Mudigonda M, Luehr N, Phillips E, Mahesh A, Matheson M, Deslippe J, Fatica M, Prabhat, Houston M. Exascale Deep Learning for Climate Analytics. Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis. November 2018, Pages 1-12.
4. Sheng Ma, Zhong Liu, Chen, Shenggang, Huang, Libo, Guo, Yang, Wang, Zhiying, Zhang, Meidi. Coordinated DMA: Improving the DRAM Access Efficiency for Matrix Multiplication, IEEE Transactions on Parallel and Distributed Systems, 2019, 30(10):2148-2164.
5. Zhong Liu, Xi Tian, Sheng Ma. The Implementation and Optimization of Parallel Linpack on Multi-core Vector Accelerator. 2019 IEEE 21st International Conference on High Performance Computing and Communications, IEEE (2019), pp.2261-2269.
6. Maji, Partha and Mullins, Robert. 1D-FALCON: Accelerating Deep Convolutional Neural Network Inference by Co-optimization of Models and Underlying Arithmetic Implementation. in: Proceedings of the 26th International Conference on Artificial Neural Networks, 2017, Springer,21-29.

7. Shaohui Lin, Rongrong Ji, Yuchao Li, Yongjian Wu, Feiyue Huang, Baochang Zhang. Accelerating Convolutional Networks via Global & Dynamic Filter Pruning,Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence. 2018, pp. 2425-2432.
8. T. Abtahi, C. Shea, A. Kulkarni and T. Mohsenin. Accelerating Convolutional Neural Network With FFT on Embedded Hardware. in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 26, no. 9, pp. 1737-1749, Sept. 2018.
9. Dominik Scherer, Hannes Schulz, and Sven Behnke. Accelerating Large-Scale Convolutional Neural Networks with Parallel Graphics Multiprocessors. 20th International Conference on Artificial Neural Networks (ICANN), Thessaloniki, Greece, September 2010.
10. Sunwoo Lee, Dipendra Jha, Ankit Agrawal, Alok Choudhary, and Wei-keng Liao. Parallel Deep Convolutional Neural Network Training by Exploiting the Overlapping of Computation and Communication. 2017 IEEE 24th International Conference on High Performance Computing.
11. N.P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al., In-data center performance analysis of a tensor processing unit, in: Proceedings of the IEEE/ACM International Symposium on Computer Architecture (ISCA), 2017, pp. 1-12.
12. M. Imani, D. Peroni, Y. Kim, A. Rahimi, T. Rosing. Efficient neural network acceleration on GPGPU using content addressable memory. In: Proceedings of the IEEE/ACM Proceedings Design, Automation and Test in Eurpoe (DATE), 2017, pp. 1026-1031.
13. C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, J. Cong. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In: Proceedings of the ACM Symposium on FPGAs, 2015, pp. 161-170.
14. A. Rahman, J. Lee, K. Choi. Efficient FPGA acceleration of convolutional neural networks using logical-3D compute array. In: Proceedings of the IEEE/ACM Proceedings Design, Automation and Test in Eurpoe (DATE), 2016, pp. 1393-1398.
15. Intel Neural Network Processor, https://www.intel.ai/intel-nervana-neural-network-processor-architecture-update. Last accessed 24 Aug 2020
16. Y. Wang, H. Li, X. Li. Re-architecting the on-chip memory sub-system of machine-learning accelerator for embedded devices. In: Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2016, p. 13.
17. M. Hu, J.P. Strachan, Z. Li, E.M. Grafals, N. Davila, C. Graves, S. Lam, N. Ge, J.J. Yang, R.S. Williams. Dot-product engine for neuromorphic computing: programming 1T1M crossbar to accelerate matrix-vector multiplication. In: Proceedings of the ACM/IEEE Design Automation Conference (DAC), 2016, p. 19.
18. Zhong Liu,Xi Tian,Xiaowen Chen,Yuanwu Lei,Man Liao. Efficient Large-scale 1D FFT Vectorization on Multi-core Vector Accelerator.2019 IEEE 21st International Conference on High Performance Computing and Communications, IEEE (2019), pp.484-491.
19. Zhang J Y, Guo Y, Hu X. Design and implementation of deep neural network for edge computing. IEICE Trans InfSyst, 2018, 101: 1982-1996.
20. Yang, Chao, Chen, Shuming, Wang, Yaohua, Zhang, Junyang. The Evaluation of DCNN on Vector-SIMD DSP. IEEE Access. VOLUME 7, 2019,PP. 22301-22309.
21. INFERENCE using the NVIDIA T4, https://www.dell.com/support/article/zh-cn/sln316556/inference-using-the-nvidia-t4?lang=en. Last accessed 24 Aug 2020