A Parallel Optimization of the Fast Algorithm of Convolution Neural Network on CPU

JiaHao Huang, Tiejun Wang*, Xuhui Zhu, Min Wei, Tao Wu, Xi Wu, Min Huang

Chengdu University of Information Technology, Chengdu, China,
tjw@cuit.edu.cn

***Abstract*—The application of convolutional neural network (CNN) is mainly based on the training over the big data, and it is necessary to obtain the relatively accurate values in real time. Although the GPU has several advantages over CPU, most of the clusters in mainstream parallel environments are composed of servers based on CPUs. So it is more useful to speed up the algorithms over the CPU-based clusters by parallelization. In this paper, an improved CNN algorithm based on Winogard algorithm was proposed. Firstly, we analyzed Winogard algorithm and found it has the best advantage in speed to small convolution kernel, such as 3×3. Then we increase the convolution process speed by reducing the complexity of convolution calculation. We also discussed the parallel implementation and optimization to the improved Winogard algorithm on CPU-based cluster. Finally, we compared the computing efficiency of different numbers of computing cores in the Intel Xeon Phi CPU platform and the results show that the best parallelism is about 6 TFLOPS.**

Keywords- Convolutional neural network; parallel; OpenMP; SIMD

I. INTRODUCTION

Convolution neural network (CNN) is a feedback neural network. CNN has been widely used in image pattern classification due to its relatively high recognition rate and no necessary to make more complex pre-processing to images in early stage. The time spent in training or identification is the key factor to the CNN algorithms. In order to reduce the time and guarantee the classification effect, multi-core convolutional neural networks based on 5-layer structure are often used in real application environment.

The first thing you can do to speed up the convolutional neural network is its structural optimization, but depending on the application scenario, there are no better algorithms to achieve the same good results for the time being. The other method we can do is to parallelize. Corresponding to the GPU parallel implementation, CPU parallel implementation is often simpler, in the practical application of the original algorithm is not too much change. One of the most popular parallel modes today is the OpenMP and MPI used on the CPU of the Intel platform. MPI parallel applications mainly in the node between the parallelization. This paper is directed to parallel processing and OpenMP applications within a node.

Convolution networks architecture with 3×3 small filters have better accuracy and less weight than shallow networks with large filters. We also perform further parallel optimizations on 3x3 small filters[2].

This paper uses convolution neural network parallel structure on CPU, through the use of OpenMP thread parallelization. Operating environment is Intel Xeon Phi(TM), according to the characteristics of its CPU, parallel threads will be assigned to each core for calculation. We will use Winogard[7] fast algorithms to speed up the convolution calculation layer and SIMD processing, and in different cases in the CPU operation flops comparison, as well as an analysis of the results of its operation.

II. RELATED WORK

For most of the CPU platform can be more good convolution neural network parallelism, and can make good use of shared storage features. We have previously learned about convolutional neural networks, comparing small filters with large filters. The parallel architecture proposed by Alex Krizhevsky[1] is well-known and is actually tested according to his model and found to be based on a GPU-based platform and a theory of parallel architecture based on distributed storage. We further found that Andrew Lavin[4] provided a fast algorithm for convolutional neural networks, and of course on a GPU based on distributed storage. We wonder if we can combine these two ideas and get better operational efficiency on parallel platforms with shared storage. Therefore, this paper was produced.

III. CONVOLUTION NEURAL NENTWORK

Alex Krizhevsky[1] proposed a better convolution neural network parallel method that is two different ways. In Convolutional layers cumulatively contain about 90%-95% of the computation, about 5% of parameters, and have large representations. But in Fully-connected layers contain about 5-10% of the computation, about 95% of the parameters, and have small representations. In particular, data parallelism appears attractive for convolutional layers, while model parallelism appears attractive for fully-connected layers.

In Convolutional layers, we assume that there are N examples as a batch. The N batch data is assigned to N cores operation. And each data is calculated by a computing core. We can be a batch of data as a computing task, of course, we can batch data again divided, according to the number of convolution filters K and channel C. The basic computational matrix is the input data for H×W and convolution kernel data of size M×N. Using matrices as basic arithmetic data, we can get the convolution calculation formula that we can get:

$$A_{i,c} = \sum_{c=1}^{C} D_{i,c} \times F_{k,c} \quad (1)$$

2157-1481/18/$31.00 ©2018 IEEE
DOI 10.1109/ICMTMA.2018.00009

In Fully-connected layers, we use model parallelism, all cores calculate one batch data and reduce unnecessary parameter data transfers. Because it is based on distributed storage system design, so in the case of N computing cores, each fully connected layer will receive the other N-1 convolutional batch data. The parallel approach to shared storage has not changed, but it will be faster to implement.

IV. WINOGARD FAST ALGORITHM

Winogard is a minimal filtering algorithm for computing m outputs with an r-tap filter, which we call $F(m, r)$[7]. The number of multiplications calculated in Winogard algorithm is:

$$\omega\left(F\left(m,r\right)\right)=m+r-1 \tag{2}$$

The number of multiplications calculated in standard algorithm is $\omega\left(m,r\right)=m\times r$. Then the efficiency of Winogard algorithm can be improved by k times where k is:

$$k=\frac{m\times r}{m+r-1} \tag{3}$$

$F(m, r)$ is an 1-dimension algorithm and we can easily get a 2-dimension version $F(m\times n, r\times s)$ by nesting it. Then the number in 2-dimension algorithm $F(m\times n, r\times s)$ can be defined as:

$$\begin{aligned}\omega\left(F\left(m\times n,r\times s\right)\right)&=\omega\left(F\left(m,r\right)\right)\times\omega\left(F\left(n,s\right)\right)\\&=\left(m+r-1\right)\times\left(n+s-1\right)\end{aligned} \tag{4}$$

, and the corresponding number of multiplications in standard algorithm is $\omega\left(m\times n,r\times s\right)=m\times n\times r\times s$. Then we can get the improvement:

$$k=\frac{m\times n\times r\times s}{\left(m+r-1\right)\times\left(n+s-1\right)} \tag{5}$$

To compute $F(m, r)$, $m+r-1$ data must be accessed as well as $(m+r-1)\times(m+r-1)$ data access to $F(m\times m, r\times r)$.

V. F(2×2, 3×3), F(3×3, 2×2), F(4×4, 3×3)

We can get the formula of $F(m, r)$ through Andrew Lavin[4] and Winogard[7]:

$$A=B^{T}\left[\left(Uf\right)\otimes\left(D^{T}d\right)\right] \tag{6}$$

where $\cdot$ indicates element-wise multiplication and for $F(2, 3)$ the matrices are:

$$B^{T}=\begin{bmatrix}1&1&1&0\\0&1&-1&-1\end{bmatrix}$$

$$U=\begin{bmatrix}1&0&0\\\frac{1}{2}&\frac{1}{2}&\frac{1}{2}\\\frac{1}{2}&-\frac{1}{2}&-\frac{1}{2}\\0&0&1\end{bmatrix},\ D^{T}=\begin{bmatrix}1&0&-1&0\\0&1&1&0\\0&-1&1&0\\0&1&0&-1\end{bmatrix}$$

$$f=\begin{bmatrix}f_0&f_1&f_2\end{bmatrix}^{T},\ d=\begin{bmatrix}d_0&d_1&d_2&d_3\end{bmatrix}^{T}$$

$F(m, r)$ is 1D algorithm, and we can easily get 2D algorithm by nesting it. So $F(m\times m, r\times r)$ like:

$$A=B^{T}\left[\left[UfU^{T}\right]\otimes\left[D^{T}dD\right]\right]B \tag{7}$$

Similarly, we can get the corresponding transform matrix B, U, D of $F(m\times m, r\times r)$ according paper [7] and the Chinese remainder theorem. We also get the transform matrix $F(3\times3, 2\times2)$ and $F(4\times4,3\times3)$ and apply it in practice. In the same time, we must also pay attention to the size m, r of the transform matrix, and which are related to the number of additions. The size of matrix B, U and D is $(m+r-1)\times(m+r-1)$, $(m+r-1)\times r$ and $m\times(m+r-1)$, respectively. We do not recommend larger m and r as that leads to more calculations and complexity in practice. Larger filters will lead to poor accuracy and more weights.

According formula(4), we need to do multiplications for 16 times to compute $F(2\times2, 3\times3)$ while we need 36 times multiplications using the standard algorithm. This is an arithmetic complexity reduction of 2.25. $F(3\times3, 2\times2)$ is the same as $F(2\times2, 3\times3)$. To compute $F(4\times4, 3\times3)$, we need 36 times multiplications while we need 144 times multiplications using the standard algorithm. Then the complexity of the algorithm is reduced by 4. Of course, it also adds some addition operations to get the $F(m\times m, r\times r)$, but the floating-point operation is very fast and it only takes very short time to the entire convolution calculation.

VI. PARALLEL OPTIMIZATION ON CPU

OpenMP is a set of guided compilation processes based on shared memory parallel system, and it makes program run in parallel automatically. As we can see from the paper by Vladimir Mironov[5], the speedup by parallel can be reached at several times even dozens of times. Especially to huge amount of data in which there no significant dependence each other, the speedup is even more distinguished. In this paper, we only demonstrated in one server with Intel Xeon Phi(TM) CPU for parallel. If need to parallelize across a large-scale cluster or multiple servers, only MPI can be used on this basis.

Convolutional layers cumulatively contain almost all of the computation. If we want to get good practice effects on the CPU, in addition to the rational use of the above algorithm and parallel structure, but also take full advantage of the characteristics of the Intel compiler.

Substituting $Y=UfU^{T}$ and $V=D^{T}dD$ to formula(7) we can get:

$$A=B^{T}\left[Y\otimes V\right]B \tag{8}$$

Defined $P=\left[H/m\right]\left[W/m\right]$ tiles per channel C in Andrew Lavin's paper. Labeling tile that calculates with a filter coordinates as (x, y). We rewrite the formula(1) for a single input data i, filter k, and tile coordinate (x, y) as:

$$A_{i,c}=\sum_{c=1}^{C}D_{i,c}\times F_{k,c}=B^{T}\left[\sum_{c=1}^{C}Y_{k,c}\otimes V_{c,i,x,y}\right]B \tag{9}$$

Substituting $M = YV$, we have:

$$M_{k,i,x,y} = \sum_{c=1}^{C} Y_{k,c} \otimes V_{c,i,x,y} \quad (10)$$

And simplify the notation by collapsing the image/tile coordinates (*i, x, y*) down to an 1-dimension *b* as:

$$M_{k,b} = \sum_{c=1}^{C} Y_{k,c} \otimes V_{c,b} \quad (11)$$

Since our environment is the Intel Xeon Phi (TM) family of CPUs, one of the CPUs has 256 cores. The compiler is Intel's ICC (Intel C++ Compiler) compiler. Through the Intel compiler, OpenMP compile parallel, according to a certain way assigned to your assigned core, according to the situation of dynamic allocation and static allocation tasks to different cores, and strive to fill the calculation of each core that you assigned. In practice, the data to be subjected to the same calculations are placed in the same vector register and calculated simultaneously using SIMD, thereby drastically reducing the number of calculations.

We make up a vector of discrete data that will do the same operation. For $V = D^T dD$, split it into two parts. The first is $V' = D^T d'D$, d' is a vector of σ discrete data. σ is determined by the size of the SIMD register of the CPU, where vector register size is 512 bytes, $\sigma = 512/n$ (*n* is the number of bytes of a single data). And the second is $V'' = D^T d''D$ where d'' is the remaining $n = P\%\sigma$ data. It will be a single computing. The same as $M = YV$ and $A = B^T MB$. $Y = UfU^T$ can be calculated outside the convolutional layer, and it cannot be changed to SIMD.

Intel have provided many intrinsics on SIMD including intel@SSE, AVX, AVX512. Intel@SSE was introduced in early CPU version, while AVX and AVX512 were introduced in the current popular CPUs. So we are using AVX512 type of intrinsics in the implementation, such as __m512 _mm512_load_ps, __m512 _mm512_set_epi32 and etc. When we use SIMD, we should pay attention to two points. Firstly, SIMD instructions generally operate faster on 16 byte blocks that are 16-byte aligned in memory. Therefore, in the application and allocation of memory need to pay attention byte aligned, we can choose to use *alignas* or _*m*512 keywords. If the application memory is not 16-byte aligned, the compiler will be adjusted to it, so the efficiency has decreased. Secondly, since every instruction operates on a block of 16 bytes, if a data vector is not a multiple of 16 bytes in size, one will have to deal with edge effects. The simplest solution is zero-padding[8]. In this paper, we put the data that is not a multiple of 16 bytes in size calculated separately in practice.

Algorithm 1 Compute convolutional layers with Winograd minimal filtering algorithm *F*(*mxm,rxr*)

N is tile size = x×y

#pragma omp parallel for
for (k = 0; k < K; ++k){
  for (c = 0; c < C; ++c){
    $Y_{k,c} = Uf_{k,c}U^T$
  }
}

#pragma omp parallel for
for (b = 0; b < K; ++b){
  for (c = 0; c < C; ++c){
    for(i=0;i<N/σ ;i+=σ){
      SIMD: $V' = D^T d'_{c,b,i} D$
    }
    for(;i<N;i++){
      $V' = D^T d''_{c,b,i} D$
    }
  }
}

#pragma omp parallel for
for (b = 0; b < B; ++b){
  for (k = 0; k < K; ++k){
    for (c = 0; c < C; ++c){
      for(i=0;i<N/σ ;i+=σ){
        SIMD: $M'_{k,b} += Y'_{k,c,i} V'_{c,b,i}$
      }
      for(;i<N;i++){
        $M'_{k,b} += Y''_{k,c,i} V''_{c,b,i}$
      }
    }
  }
}

#pragma omp parallel for
for (b = 0; b < B; ++b){
  for (k = 0; k < K; ++k){
    for(i=0;i<N/σ ;i+=σ){
      SIMD: $A = B^T M'_{k,b,i} B$
    }
    for(;i<N;i++){
      $A = B^T M''_{k,b,i} B$
    }
  }
}

In the above algorithm, we can further take advantage of platform features Intel's. In the article by Beata Bylina and Jarosław Bylina[6], it introduces linear affinities on matrix decomposition and use the OpenMP and MK. In matrix element multiplication, a can be replaced by the b function in

MKL (math kernel library). In matrix element multiplication, $M = YV$ can be replaced by the **sgemm** function in MKL and better results can be achieved. We used *–xhost*, *-openmp*, *-O2* and –MKL during compilation. *–xhost*: Tells the compiler to generate instructions for the highest instruction set available on the compilation host processor. *–openmp*: Tells the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel. *–O2*: Default optimization. Favors speed optimization with some increase in code size. Same as option O. Intrinsics, loop unrolling, and inlining are performed. *–MKL*: Tells the compiler to link to certain libraries in the Intel® Math Kernel Library (Intel® MKL). In practice, we used these options to get better results.

VII. RESULTS

In order to understand the number of threads and the parallelism of the code, we tested the fast algorithm. The operation data are shown in tables.

Table 1 standard algorithm with 1 core

num	C-H-W-K	GFLOPS	Time (ms)
1	1-40-1024-32	3.71	386.34
2	64-20-512-64	7.10	6102.70
3	256-6-512-256	5.60	27521.21
4	256-6-512-512	7.98	38581.11
5	512-6-512-512	7.91	77917.27
6	512-6-1024-2048	7.80	633066.62

The operating environment is 1 thread and 1 core. Batch 64, Channel C, Input data size H×W, Filters K.

Table 2 F(2×2, 3×3) algorithm with 1 core

num	C-H-W-K	GFLOPS	Time (ms)
1	1-40-1024-32	2.76	518.19
2	64-20-512-64	57.84	748.92
3	256-6-512-256	87.90	1752.19
4	256-6-512-512	96.55	3190.37
5	512-6-512-512	101.01	6098.92
6	512-6-1024-2048	112.48	43903.76

The operating environment is 1 thread and 1 core. Batch 64, Channel C, Input data size H×W, Filters K.

Table 3 F(2×2, 3×3) algorithm with 64 cores

num	C-H-W-K	GFLOPS	Time (ms)
1	1-40-1024-32	63.44	22.56
2	64-20-512-64	1648.86	26.28
3	256-6-512-256	2445.98	62.96
4	256-6-512-512	2754.91	111.81
5	512-6-512-512	2967.14	207.62
6	512-6-1024-2048	3738.14	1321.01

The operating environment is 64 threads and 64 cores. In practice, the result of 64 threads is better than 256 threads, so we chose the best result. Batch 64, Channel C, Input data size H×W, Filters K.

Table 4 F(4×4, 3×3) algorithm with 64 cores

num	C-H-W-K	GFLOPS	Time (ms)
1	1-40-1024-32	80.43	17.80
2	64-20-512-64	1901.09	22.78
3	256-6-512-256	3889.96	39.59
4	256-6-512-512	4436.60	39.59
5	512-6-512-512	4488.06	137.26
6	512-6-1024-2048	6064.27	814.30

The operating environment is 64 threads and 64 cores. In practice, the result of 64 threads is also best result for *F*(4×4, 3×3) algorithm. Batch 64, Channel C, Input data size H*W, Filters K.

In the standard algorithm, we find that different data volumes, filters, and channels have little impact on the actual GFLOPS calculations. F(2×2, 3×3) fast algorithm, computation has low efficiency when C and K are relatively small. But when the C and K values are relatively large, the actual calculated GFLOPS will be high. This is also the advantage of this fast algorithm, where the amount of multiplication is drastically reduced when the amount of data is slightly larger, and other prepended and post-converted transforms become negligible. The same is true for the F(4×4, 3×3) algorithm.

For standard and F(2×2, 3×3) algorithm we can find some conclusions. The fast algorithm F(3, 2) does not perform as well as the standard algorithm only when the amount of data is small and the C and K values are small. This is due to its pre-processing and parallel structure of the relatively large proportion of expenses. In others, especially in the K value is larger and the amount of data is relatively large, GFLOPS upgrade almost reached 15 times. It proves that this algorithm in practical applications, the greater the amount of data to enhance the speed of the more obvious.

F(2×2, 3×3) and F(4×4, 3×3) fast algorithms, again, when C and K are small, the boost is not obvious, with only 1/3 less. However, when C and K are relatively large, their promotion can be doubled. Of course, F(4×4, 3×3) memory consumption will be larger, but only some of padding data and transforms matrix. Relative to the entire data set, the proportion is small.

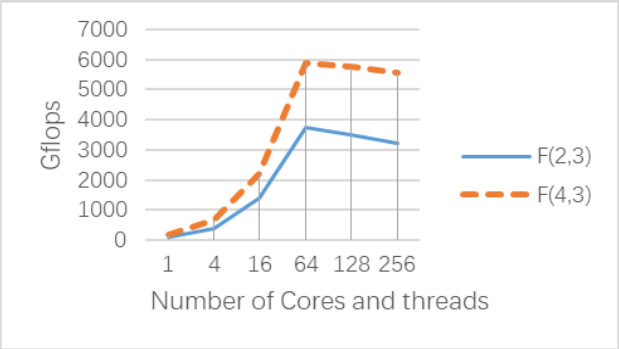

Figure 1 The relationship between the number of running cores and GFLOPS.

For the different number of cores in the fast algorithm, though the number of cores is enlarged to 64 cores. However, the actual data did not improve so much, GFLOPS to enhance the amount of about 30 times. Just from the time to shorten the point of view, but also about 30 times. In convolutional neural networks, this is a very good performance improvement. It is even more helpful for the training of large data sets. With regard to the change in the

number of cores and the trend of GFLOPS, as well as the trend in time and number of cores, we can see from Figure 1 and 2. The data we choose are the num 6 in Table 1 as the benchmark test data in Figure 1 and 2.

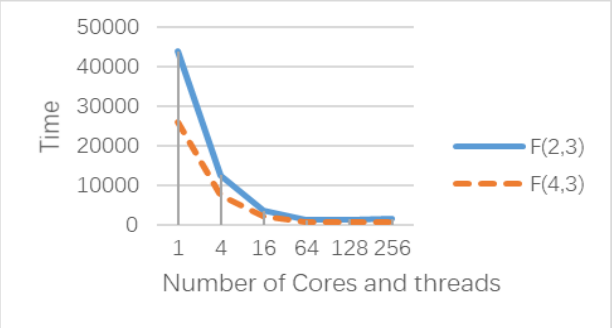

Figure 2 The relationship between the number of running cores and Time.

Figure 2 shows that the improvement of the previous auditing is very obvious for the actual computing speed increase until the number of cores reaches 64(The number of cores and threads is based on a test of the power of 2) and GFLOPS reaches the peak at the same time. Then when we increase the number of cores using to calculation in fact, GFLOPS were reduced in contrast. It is due to the optimization of the amount of data and the cost of actually allocating threads. The number of 64 cores in this set of data is the best, beyond which, there will be excessive thread allocation and environmental overhead, resulting in a decrease in practice. In the actual multi-group data test, not all data will be lower than the 64 core GFLOPS, some will improve, but in general, the trend is not large. In all data sets, GFLOPS tend to be stable after more than 64 cores are calculated. Figure 3 can also reflect that when the number of cores is greater than 64, the time changes tend to be stable.

REFERENCES

[1] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. arXiv preprint arXiv:1404.5997, 2014.

[2] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167, 2015.

[3] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556, 2014

[4] Andrew Lavin. Fast Algorithms for Convolutional Neural Networks. arXiv:1509.09308v2 2015.

[5] Vladimir Mironov. An e icient MPI/OpenMP parallelization of the Hartree-Fock method for the second generation of Intel® Xeon PhiTM processor. In Proceedings of SC17, 2017.

[6] Beata Bylina and Jarosław Bylina. OpenMP Thread Affinity for Matrix Factorization on Multicore Systems. IEEE Catalog Number: CFP1785N-ART, 2017.

[7] Shmuel Winograd. Arithmetic complexity of computations, volume 33. Siam, 1980.

[8] Vanhoucke V, Mao M Z. Improving the speed of neural networks on CPUs[J]. Deep Learning & Unsupervised Feature Learning Workshop Nips, 2011.