Hindawi
Computational Intelligence and Neuroscience
Volume 2017, Article ID 8348671, 8 pages
https://doi.org/10.1155/2017/8348671

Research Article

High Performance Implementation of 3D Convolutional Neural Networks on a GPU

Qiang Lan,[1,2] Zelong Wang,[1,2] Mei Wen,[1,2] Chunyuan Zhang,[1,2] and Yijie Wang[1,2]

[1]*College of Computer, National University of Defense Technology, Changsha 410073, China*
[2]*National Key Laboratory of Parallel and Distributed Processing, Changsha 410073, China*

Correspondence should be addressed to Qiang Lan; lanqiang_nudt@163.com

Received 16 April 2017; Revised 19 July 2017; Accepted 6 August 2017; Published 8 November 2017

Academic Editor: Athanasios Voulodimos

Convolutional neural networks have proven to be highly successful in applications such as image classification, object tracking, and many other tasks based on 2D inputs. Recently, researchers have started to apply convolutional neural networks to video classification, which constitutes a 3D input and requires far larger amounts of memory and much more computation. FFT based methods can reduce the amount of computation, but this generally comes at the cost of an increased memory requirement. On the other hand, the Winograd Minimal Filtering Algorithm (WMFA) can reduce the number of operations required and thus can speed up the computation, without increasing the required memory. This strategy was shown to be successful for 2D neural networks. We implement the algorithm for 3D convolutional neural networks and apply it to a popular 3D convolutional neural network which is used to classify videos and compare it to cuDNN. For our highly optimized implementation of the algorithm, we observe a twofold speedup for most of the 3D convolution layers of our test network compared to the cuDNN version.

1. Introduction

Convolutional neural networks have proven advantages over traditional machine learning methods on applications such as image classification [1–4], tracking [5, 6], detection [7–11]. However, the primary downside of convolutional neural networks is the increased computational cost. This becomes especially challenging for 3D convolution where handling even the smallest instances requires substantial resources.

3D convolutional neural networks have recently come to the attention of the scientific community. In [12], a database for 3D object recognition named ObjectNet3D is presented. The database focuses on the problem of recognizing the 3D pose and the shape of objects from 2D images. Another repository of 3D CAD models of objects is ShapeNet [13]. In [14], the authors propose VoxNet, a 3D convolutional neural network, to solve the robust object recognition task with the help of 3D information, while the authors of [15] propose a 3D convolutional neural networks for human-action recognition.

In the light of these successful applications, it is worthwhile to explore new ways of speeding up the 3D convolution operation. In this paper we do so by deriving the 3D convolution forms of the minimal filtering algorithms invented by Toom and Cook [16] and generalized by Winograd [17]. Our experiments show this algorithm to be very efficient in accelerating 3D convolutional neural network in video classification applications.

2. Related Work

Many approaches aim to directly reduce the computational cost within CNN. In [18], the authors analyse the algebraic properties of CNNs and propose an algorithmic improvement to reduce the computational workload. They achieve a 47% reduction in computation without affecting the accuracy. In [19], convolution operations are replaced with pointwise products in the Fourier domain, which can reduce the amount of computation significantly. Reference [20] evaluates two fast Fourier transform (FFT) convolution implementations, one based on Nvidia cuFFT [21] and the other based on Facebook's FFT implementation. The FFT method can achieve an obvious speeding up of performance when the filter size is large, and the disadvantage of the FFT

method is that it consumes much more memory than the standard method.

In [22], the authors use WMFA (Winograd Minimal Filter Algorithm) [17] to implement the convolution operation. In theory, fewer multiplications are needed in the WMFA, while not much extra memory is needed. WMFA is easy to parallelize; Lavin and Gray [22] implemented the algorithm on GPU, and they achieved better performance than the fastest cuDNN library. In [23], the authors show a novel architecture implemented in OpenCL on an FPGA platform; the algorithm they use to do the convolution is WMFA, which significantly boosts the performance of the FPGA. However, both works implemented 2D convolutional neural networks.

In this paper, we make four main contributions. Firstly, we derive the 3D forms of WMFA and design detailed algorithm to implement 3D convolution operation based on 3D WMFA. Secondly, we analyse the arithmetic complexity of 3D WMFA and prove 3D WMFA method can reduce computation in theory. Thirdly, we implement 3D WMFA for GPU platform and propose several optimization techniques to improve the performance of 3D WMFA. Finally, we evaluate the performance of 3D convolutional neural networks based on several implementations and prove the advantage of our proposed 3D WMFA method.

3. Fast 3D Convolution Algorithm

3.1. Preliminary: 3D Convolutional Neural Networks. For the 2D convolution, kernels have fixed width and height, and they are slid along the width and height of the input feature maps. For the 3D convolution, both feature maps and kernels have depth dimension, and the convolution also needs to slide along the depth direction. We can compute the output of a 3D convolutional layer using the following formula:

$$Y_{i,k,x,y,z} = \sum_{c=0}^{C-1} \sum_{t=0}^{T-1} \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} I_{i,c,x+t,y+r,z+s} F_{k,t,r,s,c}, \quad (1)$$

where $Y_{i,k,x,y,z}$ represents the result of a convolution operation at the kth channel feature and $I_{i,c,x+t,y+r,z+s}$ is one of the input features, while $F_{k,t,r,s,c}$ is one of the filters. Equation (1) represents a direct convolution method, which requires intensive computability. The detailed arithmetic complexity of this method is shown in Section 3.3.

3.2. 3D WMFA. We introduce a new, fast algorithm to compute a 3D convolutional layer. The algorithm is based on WMFA. In order to introduce the 3D WMFA, firstly, we will give a simple introduction to the 1D WMFA. WMFA computes output with a tile size of m each time; we use $F(m, r)$ to represent the output tile and r is the filter size. According to the definition of convolution, $2 \times 3 = 6$ multiplications are required to compute $F(2, 3)$, but we can reduce the number of multiplications to do the convolution if we use the following WMFA:

$$F(2,3) = \begin{bmatrix} i_0 & i_1 & i_2 \\ i_1 & i_2 & i_3 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix}, \quad (2)$$

where

$$\begin{aligned} m_1 &= (i_0 - i_2) f_0, \\ m_2 &= (i_1 + i_2) \frac{f_0 + f_1 + f_2}{2}, \\ m_4 &= (i_1 - i_3) f_2, \\ m_3 &= (i_2 - i_1) \frac{f_0 - f_1 + f_2}{2}. \end{aligned} \quad (3)$$

The number of multiplications needed is $\mu(F(2,3)) = 2 + 3 - 1 = 4$; however, four additions are needed to transform the input image, three additions to transform the filter, and four additions to transform the result of the dot product. We can use a matrix form to represent the computation:

$$Y = A^T \left[\left(B^T i \right) \odot (Gf) \right]. \quad (4)$$

We call the A^T, G, and B^T transform matrices, and the values of the transforming matrices are

$$\begin{aligned} B^T &= \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}, \\ G &= \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}, \\ A^T &= \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}. \end{aligned} \quad (5)$$

In (4), $i = \begin{bmatrix} i_0 & i_1 & i_2 & i_3 \end{bmatrix}^T$ and $f = \begin{bmatrix} f_0 & f_1 & f_2 \end{bmatrix}^T$ represent the input tile and filter tile, respectively. As described in [22], the format of the 2D WMFA is as follows:

$$Y = A^T \left[\left[GfG^T \right] \odot \left[B^T i B \right] \right] A, \quad (6)$$

where f is the filter with size $r \times r$ and i is the image with size $(m + r - 1) \times (m + r - 1)$. To compute $F(2 \times 2, 3 \times 3)$, we need $4 \times 4 = 16$ multiplications; however, $4 \times 9 = 36$ multiplications are needed according to the convolution definition. Therefore, 2D WMFA can reduce the number of multiplications by a factor of $36/16 = 2.25$ at the cost of increasing 32 additions in the data transformation stage, 28 floating point instructions at the filter transformation stage, and 24 additions at the inverse transformation stage. For a convolutional layer, the number of input channels and number of output channels are large, which means the input channels need to convolve different filters, so the transformed input tile can be reused as many times as the number of output channels. Each filter needs to be slid in x

Input: $I_0[in\_size][in\_size][in\_size]$
Temp array: $I_1[in\_size][out\_size][in\_size]$, $I_2[in\_size][out\_size][out\_size]$
Output: $I_3[out\_size][out\_size][out\_size]$
for $i = 0$ to in_size **do**
  for $j = 0$ to in_size **do**
    $I_1[i][0 : out\_size][j] = T_m I_0[i][0 : in\_size][j]$
  end for
end for
for $i = 0$ to in_size **do**
  for $j = 0$ to out_size **do**
    $I_2[i][j][0 : out\_size] = T_m I_1[i][j][0 : in\_size]$
  end for
end for
for $i = 0$ to out_size **do**
  for $j = 0$ to out_size **do**
    $I_3[0 : out\_size][i][j] = T_m I_2[0 : in\_size][i][j]$
  end for
end for

Algorithm 1: 3D winograd transformation.

and y direction of input channel during convolution, so each transformed filter is reused as many times as the number of subtiles of input channel. And since the output tile is reduced along the input channels, the inverse transformation is done after reduction; then the number of inverse transformation is determined by the number of output channels. Therefore, the cost of data transformation stage, filter transformation stage, and the inverse transformation stage keep low in real convolutional layer implementation.

We can also apply the 3D WMFA to 3D convolution. To compute $F(2 \times 2 \times 2, 3 \times 3 \times 3)$, we apply the 3D Winograd transformation to the input tile and filter tile and apply 3D Winograd inverse transformation to the dot product of the transformed input image tile and the transformed filter tile. Algorithm 1 is a general form of the 3D Winograd transformation. In the algorithm, T_m is the transformation matrix; the transformation matrix can be G applied to transform the filter tile or B^T applied to transform the input image tile. The dot product of the transformed input image tile and transformed filter tile will be accumulated along the C channels, which can be converted to a matrix multiplication similar to the description in [22]

$$\begin{aligned} Y_{i,x,y,z,k} &= \sum_{c=0}^{C-1} I_{i,c,x,y,z} * F_{k,c} = \sum_{c=0}^{C-1} A^T \left[U_{k,c} \odot V_{c,i,x,y,z}\right] A \\ &= A^T \left[\sum_{c=0}^{C-1} U_{k,c} \odot V_{c,i,x,y,z}\right] A. \end{aligned} \tag{7}$$

Consider the sum

$$M_{k,i,x,y,z} = \sum_{c=0}^{C-1} U_{k,c} \odot V_{c,i,x,y,z}. \tag{8}$$

The previous equation can be divided into several submatrix multiplications; assume the output tile size is (ε, η, ν), using new coordinates $(i, \tilde{x}, \tilde{y}, \tilde{z})$ to replace (i, x, y, z), yielding

$$M^{\varepsilon,\eta,\nu}_{k,i,\tilde{x},\tilde{y},\tilde{z}} = \sum_{c=0}^{C-1} U^{(\varepsilon,\eta,\nu)}_{k,c} V^{(\varepsilon,\eta,\nu)}_{c,i,\tilde{x},\tilde{y},\tilde{z}}. \tag{9}$$

This equation represents the matrix multiplication, and it can be simplified as follows:

$$M^{(\varepsilon,\eta,\nu)} = U^{(\varepsilon,\eta,\nu)} V^{(\varepsilon,\eta,\nu)}. \tag{10}$$

Algorithm 2 gives the overview of the 3D WMFA. The algorithm mainly consists of four stages, which are Winograd transformation of the input feature tile; Winograd transformation of the filter tile; the matrix multiplication, which is converted from the dot product of the transformed input tile and the transformed filter tile; and the inverse Winograd transformation of the result of the matrix multiplication.

3.3. Arithmetic Complexity Analysis. For input feature maps with size $N \times C \times D \times H \times W$, filters with size $K \times C \times k \times k \times k$, and the output features with size $N \times K \times M \times P \times Q$, the total number of float operations in the multiplication stage can be represented as follows:

$$L_1 = 2N \left\lceil \frac{M}{m} \right\rceil \left\lceil \frac{P}{m} \right\rceil \left\lceil \frac{Q}{m} \right\rceil CK (m + r - 1)^3, \tag{11}$$

where r is the filter size and m is the size of the output subtile. However, if we use the direct convolution method, which is computed according to the definition of convolution, the total number of float operations is computed as follows:

$$L_2 = 2NMPQCKr^3. \tag{12}$$

Dividing L_1 by L_2 yields

$$\frac{L_2}{L_1} = \frac{m^3 * r^3}{(m + r - 1)^3}. \tag{13}$$

$P = N\lceil M/m\rceil\lceil P/m\rceil\lceil Q/m\rceil$ is the number of image tiles.
$\alpha = m + r - 1$ is the input tile size.
Neighbouring tiles overlap by $r - 1$.
$d_{c,b} \in R^{\alpha\times\alpha\times\alpha}$ is input tile b in channel c.
$g_{k,c} \in R^{r\times r\times r}$ is filter k in channel c.
$Y_{k,b} \in R^{m\times m\times m}$ is output tile b in filter k.
for $k = 0$ to K **do**
  for $c = 0$ to C **do**
    $u = T_k(g_{k,c}) \in R^{\alpha\times\alpha\times\alpha}$
    Scatter u to matrices U: $U_{k,c}^{(i,j,k)} = u_{i,j,k}$
  end for
end for
for $b = 0$ to P **do**
  for $c = 0$ to C **do**
    $v = T_d(d_{c,b}) \in R^{\alpha\times\alpha\times\alpha}$
    Scatter v to matrices V: $V_{c,b}^{(i,j,k)} = v_{i,j,k}$
  end for
end for
for $i = 0$ to α **do**
  for $j = 0$ to α **do**
    for $k = 0$ to α **do**
      $M^{(i,j,k)} = U^{(i,j,k)}V^{(i,j,k)}$
    end for
  end for
end for
for $k = 0$ to K **do**
  for $b = 0$ to P **do**
    Gather m from matrices M: $m_{i,j,k} = M_{k,b}$
    $Y_{k,b} = T_m(m)$
  end for
end for

Algorithm 2: 3D Convolutional layer implemented with WMFA $F(m \times m \times m, r \times r \times r)$.

Assuming $m = 2$ and $r = 3$, there is an arithmetic complexity reduction of $(2 * 2 * 2 * 3 * 3 * 3)/(4 * 4 * 4) = 216/64 = 3.375$. However, there are some extra computations in Winograd transformation stage, so we cannot achieve so much complexity reduction in reality. The detailed performance improvements are shown in Section 5.2.

4. Implementation and Optimizations

4.1. Implementation. We have three implementation versions for the 3D WMFA on a GPU. In our implementations, cuBLAS is called to do the multiplication. Furthermore, we manually implement six kernels according to Algorithm 2. Figure 1 shows the flow of our baseline implementation. The *imageTransform* kernel transforms all the image subtiles, the *filterTransform* kernel transforms all the filter tiles, and *ouputTransform* kernel inversely transforms the result of the multiplication. For the baseline implementation, we also have two additional kernels to reorganize the transformed image data and transformed filter data and one kernel to reorganize the result of the multiplication before the results go to the *outputTransform* kernel.

Winograd transformation algorithm is suitable for parallelization on GPU. Taking the *imageTransform* kernel as an example, the input to the *imageTransform* kernel is the input feature map. We assume the input feature maps have a size of $N \times C \times D \times H \times W$, where N is the batch size, C is the number of input channels, and $D \times H \times W$ is the size of a single channel. As is described in Algorithm 2, the number of image tiles for each channel is P, so the total number of image tiles is $P \times C$; all those image tiles can be transformed independently. For the baseline of the GPU implementation, the image data is stored in *NCDHW* order, and we set the number of threads in one block to be 32, each thread is responsible for processing Winograd transformation of one input subtile, and the number of blocks in the grid is set to $(\lceil N/32\rceil, \lceil M/m\rceil\lceil P/m\rceil\lceil Q/m\rceil, C)$. We can make full use of the large-scale parallel processing units of a GPU when the number of blocks is large.

For the *filterTransform* kernel, there are $K \times C$ filter tiles; we still set the number of threads of one block to be 32 and the number of blocks to $(\lceil K/32\rceil, C, 1)$. Before we call cuBLAS to implement the matrix multiplication, we need to reorganize the transformed filter and transformed image data. For transformed filter, there are $K \times C$ tiles in total, and each tile has size of $\alpha \times \alpha \times \alpha$, each time we gather one value from one tile to generate a submatrix of size $K \times C$. Figure 2 shows how the transformed data are reorganized in new layout on GPU; they are implemented by the kernels *reshapeTransformedImage* and *reshapeTransformedFilter* in

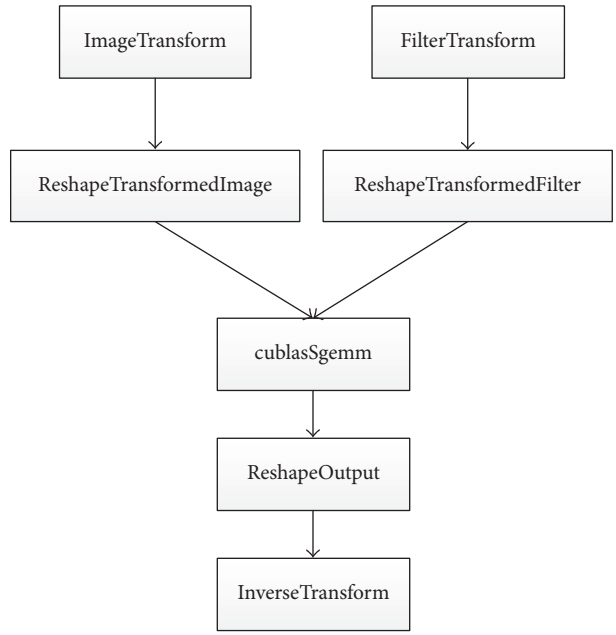

Figure 1: The computing flow of 3D WMFA.

our baseline implementation. They gather correlated and transformed filters and transformed image data to form two submatrices, and *SGEMM* from the cuBLAS library is called to do the multiplication. The result of the multiplication also needs to be reshaped before the inverse transformation.

4.2. Optimizations. We make two optimizations to achieve higher performance. The first optimization is to align memory access, which can make memory access more efficient and increase the cache hit rate. The second optimization is to combine the transformation kernel with the reshape kernel to reduce global memory access.

The storing order of data and how data is accessed by a thread can significantly affect the performance. For the baseline implementation, the image data is stored in *NCDHW* order, and threads in the same block access data along the N-dimension, which means data accessed by threads keeps long distances. However, the size of the cache on a GPU is limited; therefore, if the distance between the data items accessed by threads is larger than the size of the cache line, then each thread needs to access global memory separately, causing lots of memory accesses. In our first optimization version, we change the storage order of image data to *CDHWN*. Since the image data is stored starting from the N-dimension, data accessed by all threads in the same block is continually stored, and all data items loaded from the global memory are useful, which means the bandwidth is fully used. The same optimization method is applied to the filter transformation and inverse transformation kernel.

Based on the first optimization, we apply our second optimization to improve performance further. The second optimization is to reduce the number of global memory accesses. For the baseline implementation, after the filter transformation or image transformation kernel is executed, the transformed filter or transformed image data need to be stored in a new layout using *reshapeTransformedFilter* and *reshapeTransformedImage* kernel and using the *ReshapeOutput* kernel before the inverse transformation, while on our second optimized version, we move the work of the reshape kernel to the transformation kernel, so in the optimized transform kernel, after the inputs are transformed, the result will be stored directly back in the expected layout. In the optimized inverse transformation, before inverse transformation, the required data is gathered directly from the global memory.

Table 1: Properties of the GeForce GTX 1080.

Parameters	Values
CUDA capability major/minor version number	6.1
Total amount of global memory	8 GB
CUDA cores	2560
L2 cache Size	2 MB
Warp size	32
Total number of registers available per block	64 KB

5. Experiments

5.1. Experimental Setup. All experiments are evaluated on a GeForce GTX 1080 GPU, which has a total amount of 8 GBytes global memory and has 20 multiprocessors. Detailed parameters of the GTX 1080 are shown in Table 1.

5.2. Performance Evaluation. We apply our 3D WMFA to a widely used 3D neural network called v3d [9], which is used to classify videos. The 3D neural network has five convolutional layers; Table 2 shows the information about these 3D convolutional layers.

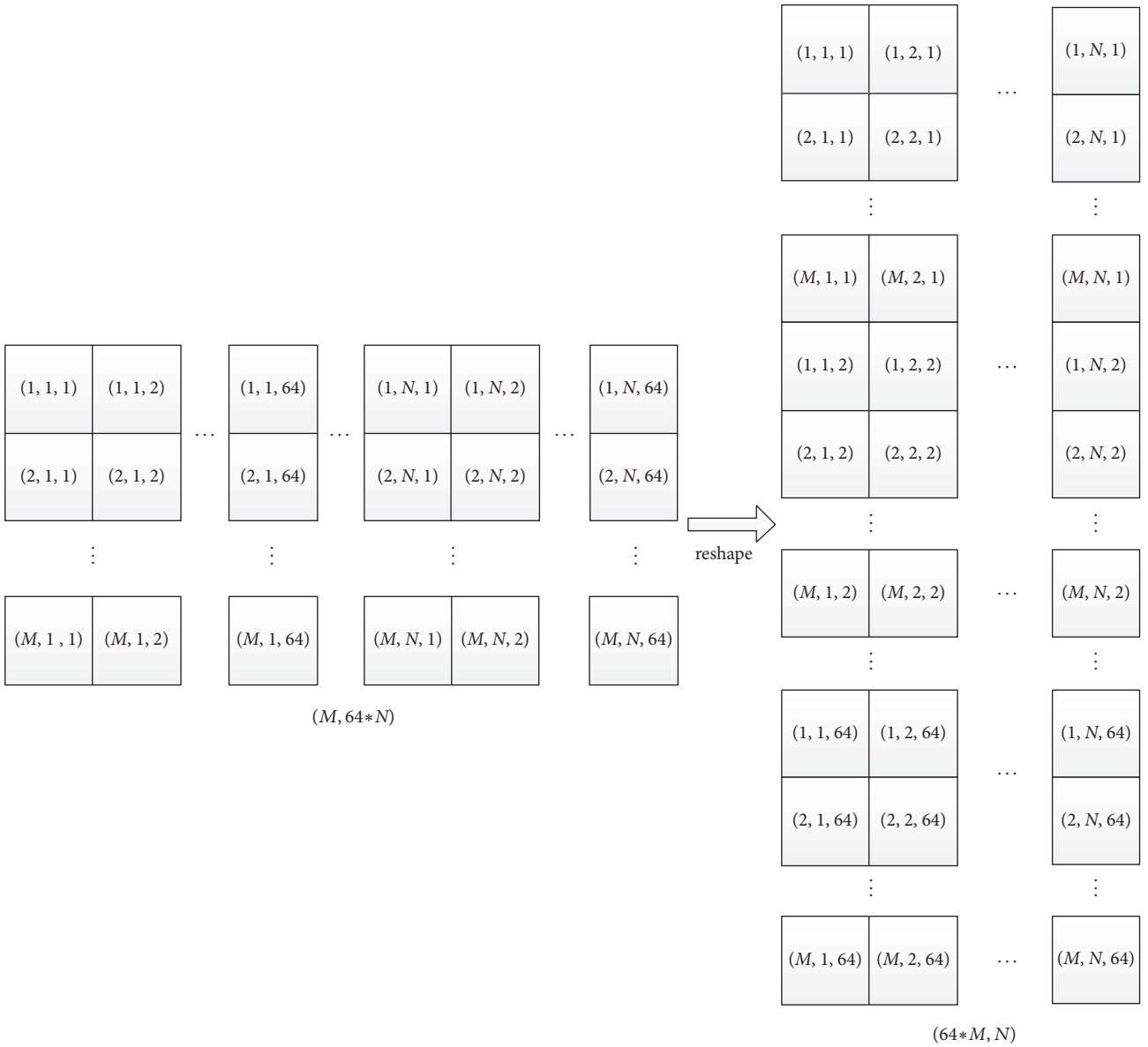

Figure 2: For an input matrix, its size is $(M, N * \alpha * \alpha * \alpha)$; α is the tile size, here equal to 4. After the reshape kernel is applied, lots of small submatrices with new layouts are generated.

Table 2: Convolution layers of a 3D network; the filter size in all layers is $3 \times 3 \times 3$, and the GFLOPS columns calculate the number of flops operations in each convolutional layer. Assume the batch size is 32.

Layer	$C \times D \times H \times W \times N$	K	GFLOPS
conv1	$3 \times 16 \times 112 \times 112 \times 32$	32	16.65
conv2	$32 \times 16 \times 56 \times 56 \times 32$	64	88.8
conv3	$64 \times 8 \times 28 \times 28 \times 32$	256	88.8
conv4	$256 \times 4 \times 14 \times 14 \times 32$	256	44.4
conv5	$256 \times 2 \times 7 \times 7 \times 32$	256	5.55

Firstly, we evaluate the performance of our three implementations on the 3D convolutional layers, except for the first convolutional layer, which has only three input channels and is not yet supported in the algorithm. Figure 3 shows the increase in speed we achieved after we used two optimizations. For the first optimization, we observe a 3 to 4 times speeding up for all these test convolution layers compared to the baseline implementation. However, for the second optimization, the maximum speeding up can be close to 42 for the third convolution; even the minimum speeding up is about 13 for the last convolution layer. The first optimization makes memory access more efficient, and the second optimization reduces lots of unnecessary global memory accesses. Since the latency of global memory access on a GPU is large, we achieve a good performance improvement in the second optimization.

We explore the detailed performance for one specific convolution layer to see how these two optimizations improve

Table 3: Performance of cuDNN SGEMM versus that of the 3D WMFA on 3D convolution layers. Performance is measured in effective TFLOPS.

Layer	$C \times D \times H \times W \times N$	K	TFLOPS		Speedup
			cuDNN SGEMM	3D WMFA	
conv2	$32 \times 16 \times 56 \times 56 \times 32$	64	1.21	1.28	1.05
conv3	$64 \times 8 \times 28 \times 28 \times 32$	256	2.38	3.31	1.39
conv4	$256 \times 4 \times 14 \times 14 \times 32$	256	2.4	4.72	1.96
conv5	$256 \times 2 \times 7 \times 7 \times 32$	256	1.46	2.1	1.44

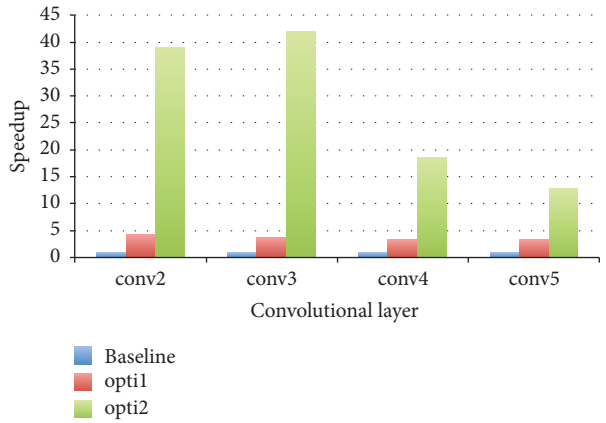

Figure 3: Speedup with different optimizations on 3D convolution layers.

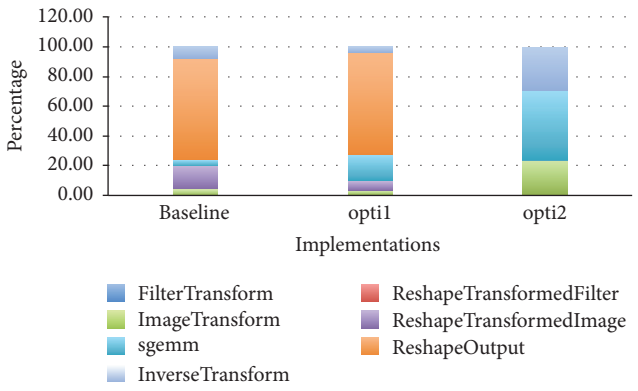

Figure 4: Time percentage distribution of each kernel in each implementation version for a specific convolution layer.

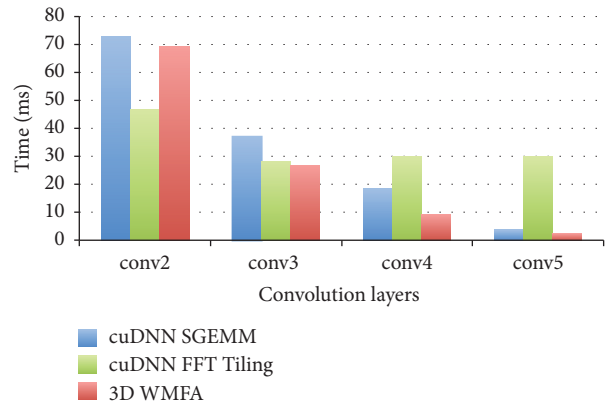

Figure 5: Execution time of different methods on 3D convolution layers.

each kernel separately. We profile the time percentage of each kernel in each implementation version for conv3, which takes most of the computation time among all these convolution layers. Figure 4 shows the profiling results; the kernel *ReshapeOutput* in both baseline and the first optimization takes up the most time, since the kernel contains lots of global memory accesses. However, in the second optimization version, there are no *reshape* kernels and the kernel *sgemm* takes most of the execution time. In all three implementations, the kernel *filterTransform* takes only about 0.1% of the total time.

Finally, we compare our best optimized implementation with the cuDNN library. The cuDNN library is the fastest deep-learning library. There are two algorithms available to implement a 3D convolution layer: one converts the convolution to a matrix multiplication and the other exploits the FFT tiling method to implement the convolution. We use *cuDNN SGEMM* and *cuDNN FFT Tiling* to represent the two methods called in the cuDNN library, and we use *3D WMFA* to represent our algorithm. Figure 5 shows the execution time of these three methods on four convolution layers. The *3D WMFA* method is about 30% slower than the *cuDNN FFT Tiling* method on the conv2 layer; however, it is a bit faster than *cuDNN SGEMM* method. The *cuDNN FFT Tiling* method achieves a fast speed at the cost of consuming a large amount of memory. Since parameters C and K are not large in the conv2 layer, this makes the matrix multiplication in *3D WMFA* on small scale, which affects the performance. However, with parameters C and K increased on layers conv3, conv4, and conv5, the *3D WMFA* method achieves better performance than the other two methods. We added the execution time of each layer for each method, the total time of all layers is 132.6 ms for cuDNN SGEMM method, 135.4 ms for cuDNN FFT Tiling method, and 108.2 ms for 3D WMFA which is better than the other two methods.

We can also calculate the performance of these two methods in *TFLOPS*. Table 3 shows the effective *TFLOPS* of *cuDNN SGEMM* and *3D WMFA* method. We achieve a maximum speedup of 1.96 compared to *cuDNN SGEMM*.

6. Conclusions

A 3D convolution layer requires a high computational cost and consumes lots of memory. We designed a 3D WMFA

to implement 3D convolution operation. Compared to traditional convolution methods, such as SGEMM or FFT, the 3D WMFA can reduce computation, in theory. When we implemented the algorithm on a GPU, we observed the expected performance of the algorithm in the experiments. For some 3D convolution layers, we even achieve close to 2 times speedup compared to cuDNN library.

However, the computation and memory requirements of 3D convolution obviously increase with more complex 3D neural networks. In our future work, we will implement $F(4\times 4\times 4, 3\times 3\times 3)$ to reduce the computation further to ease the intensive computation problem and adopt a FP16 data type to compute, which can save half of the memory usage directly. It is also necessary to parallel the convolution computation among multi-GPUs or multinodes.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

The authors gratefully acknowledge support from the National Key Research and Development Program under no. 2016YFB1000401; the National Nature Science Foundation of China under NSFC nos. 61502509, 61402504, and 61272145; the National High Technology Research and Development Program of China under no. 2012AA012706; and the Research Fund for the Doctoral Program of Higher Education of China under SRFDP no. 20124307130004.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 26th Annual Conference on Neural Information Processing Systems (NIPS '12)*, pp. 1097–1105, Lake Tahoe, Nev, USA, December 2012.

[2] M. Lin, Q. Chen, and S. Yan, Network in network, CoRR abs/1312.4400,.

[3] C. Szegedy, W. Liu, Y. Jia et al., "Going deeper with convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR '15)*, pp. 1–9, Boston, Mass, USA, June 2015.

[4] K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, abs/1409.1556, CoRR, 1409.

[5] J. Fan, W. Xu, Y. Wu, and Y. Gong, "Human tracking using convolutional neural networks," *IEEE Transactions on Neural Networks*, vol. 21, no. 10, pp. 1610–1623, 2010.

[6] S. J. Nowlan and J. C. Platt, "A convolutional neural network hand tracker," *Advances in Neural Information Processing Systems*, pp. 901–908, 1995.

[7] M. Szarvas, A. Yoshizawa, M. Yamamoto, and J. Ogata, "Pedestrian detection with convolutional neural networks," in *Proceedings of the 2005 IEEE Intelligent Vehicles Symposium*, pp. 224–229, Las Vegas, Nev, USA, June 2005.

[8] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," in *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 779–788, Las Vegas, Nev, USA, June 2016.

[9] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and F.-F. Li, "Large-scale video classification with convolutional neural networks," in *Proceedings of the 27th IEEE Conference on Computer Vision and Pattern Recognition, (CVPR '14)*, pp. 1725–1732, Columbus, Ohio, USA, June 2014.

[10] S. Lawrence, C. L. Giles, A. C. Tsoi, and A. D. Back, "Face recognition: a convolutional neural-network approach," *IEEE Transactions on Neural Networks*, vol. 8, no. 1, pp. 98–113, 1997.

[11] "Batch size for training convolutional neural networks for sentence classification," *Journal of Advances in Technology and Engineering Research*, vol. 2, no. 5, 2016.

[12] Y. Xiang, W. Kim, W. Chen et al., "Objectnet3D: A large scale database for 3D object recognition," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9912, pp. 160–176, 2016.

[13] A. X. Chang, T. A. Funkhouser, L. J. Guibas et al., *Shapenet: An information-rich 3d model repository., CoRR abs/1512.03012*, An information-rich 3d model repository, Shapenet.

[14] D. Maturana and S. Scherer, "VoxNet: A 3D Convolutional Neural Network for real-time object recognition," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2015*, pp. 922–928, Hamburg, Germany, October 2015.

[15] S. Ji, W. Xu, M. Yang, and K. Yu, "3D Convolutional neural networks for human action recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 1, pp. 221–231, 2013.

[16] D. E. Knuth, *The Art of Computer Programming*, vol. 2, Seminumerical Algorithms, Addison-Wesley Longman Publishing Co, Boston, Mass, USA, 3rd edition, 1997.

[17] S. Winograd, *Arithmetic complexity of computations*, vol. 33 of *CBMS-NSF Regional Conference Series in Applied Mathematics*, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, Pa., 1980.

[18] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8681, pp. 281–290, 2014.

[19] M. Mathieu, M. Henaff, and Y. LeCun, Fast training of convolutional networks through ffts, CoRR abs/1312.5851,.

[20] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun, *Fast convolutional nets with fbfft: A GPU performance evaluation, CoRR abs/1412.7580*, Fast convolutional nets with fbfft, A GPU performance evaluation.

[21] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of Computation*, vol. 19, pp. 297–301, 1965.

[22] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016*, pp. 4013–4021, July 2016.

[23] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An OpenCL™ deep learning accelerator on Arria 10," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2017*, pp. 55–64, Monterey, Calif, USA, February 2017.