Neural Processing Letters (2019) 50:1951–1967
https://doi.org/10.1007/s11063-019-09981-z

# A Fast Algorithm for Convolutional Neural Networks Using Tile-based Fast Fourier Transforms

**Jinhua Lin**[1,2] · **Yu Yao**[1]

Published online: 16 January 2019

## Abstract

State-of-the-art convolution algorithms accelerate training of convolutional neural networks (CNNs) by decomposing convolutions in time or Fourier domain, these decomposition implementations are designed for small filters or large inputs, respectively. We take these two aspects into account, devote to a novel decomposition strategy in Fourier domain and propose a conceptually useful algorithm for accelerating CNNs. We extend the classical Fast Fourier Transform theory to meet the requirements of convolving large inputs with small filters in faster manner. The tile-based decomposition strategy is introduced into Fourier transforms to yield a fast convolution algorithm. The algorithm, called tFFT, is simple to program, implementing tile sized transformations in Fourier domain to minimize convolution time for modern CNNs. tFFT reduces the arithmetic complexity of CNNs by over a factor of 3 compared to FFT-based convolution algorithms. We evaluate the performance of tFFT by implementing it on a set of state-of-the-art CNNs, the experiments show good results at batch sizes from 1 to 128.

**Keywords** Convolutional neural network · Decomposition implementations · Small filters · Fourier transforms

## 1 Introduction

The deep learning community has successfully improved the performance of convolutional neural networks during a short period of time [1–4]. An important part of these improvements are driven by accelerating convolutions using FFT [5] based convolution frameworks, such as the cuFFT [6] and fbFFT [7]. These implementations are theoretically simple and offer effectiveness for fast training of convnets. Our destination in this paper is to construct a relatively enabling FFT based implementation for accelerating convolutional neural networks.

✉ Jinhua Lin
ljh3832@163.com

[1] School of Computer Application Technology, Changchun University of Technology, Nanhu Road No. 2055, Changchun, China

[2] Machinery and Electronics Engineering, University of Chinese Academy of Sciences, Yu Quan Road No. 19, Beijing, China

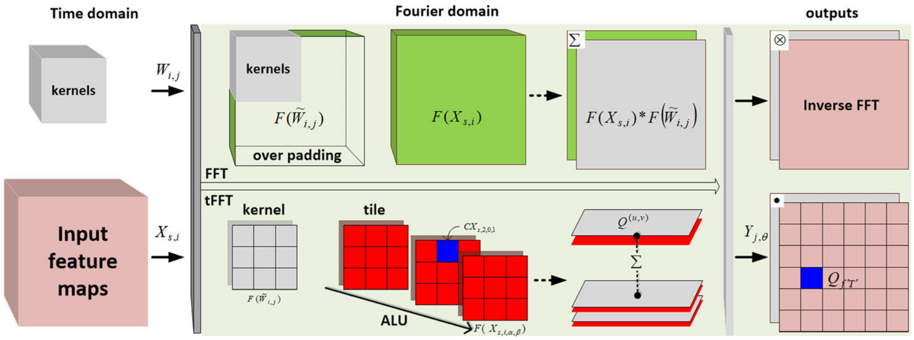


**Fig. 1** The tFFT architecture compared to conventional FFT. Note that all variables are explained in Sect. 4

Modern convolutional neural networks are challenged by computational expenses. Large amounts of training data are required for better accuracy while decreasing learning time simultaneously. It thus incorporates elements from FFT-based convolution strategy where the destination is to implement time domain convolutions by using point-wise products in Fourier frequency domain, and minimal filtering strategy where the destination is to compute minimal complexity convolution over small size filters. Given this, a seemingly complex implementation is required to develop good results. We present that a comparably simple, efficient and fast algorithm is able to exceed prior state-of-the-art convolution results at batch sizes from 1 to 128.

Our algorithm, called tFFT, outperforms conventional FFT-based convolution by introducing tile-based decomposition strategy into Fast Fourier Transform (Fig. 1).[1] This tile-based Fast Fourier Transform is an asymptotically recursive transformation applied to each tile divided inputs in Fourier domain, obtaining a convolution result by accumulating point-wise products with respect to these tiles in Fourier domain. Owning to the open-source fbFFT framework, tFFT is simple to program and fast to compute, which is convenient for a flexible extension of FFT-based convolution designs.

In theory tFFT extends conventional FFT in an intuitive manner, yet designing a tile-based FFT properly is important for efficient results. Above all, FFT was not exclusively developed for small sized filters. This is most obvious in how cuFFT performs inefficient mixed-radix convolution for CNNs with 3 × 3 kernels. In spite of a seemingly modest improvement, tFFT shows a great influence: first, tFFT speeds up convolutions by relative 5.10 to 15.34 TFLOPS on NVIDIA GTX 1080i GPU with peak throughput of 5.67, even obtaining bigger gains under larger size batches. Second, tFFT achieves an average arithmetic complexity reduction of 3.12× for ResNet-34, 2.64× for VGGNet-19 and 2.66× for AlexNet, and shows state-of-the-art results at batch sizes from 1 to 128.

In this paper, we present an efficient tile-based Fast Fourier Transform algorithm that performs multi-sized convolutions in a fast and accurate manner. The main contributions of this paper include:

1. A tile-based Fast Fourier Transform algorithm is proposed to minimize convolution time for modern CNNs (see Sects. 3 and 4).
2. A tile-based decomposition strategy is introduced into Fourier transform to yield fast transformation in forward and backward propagation passes (see Sects. 4.1 and 4.2).

[1] ~ denotes the size of zero padding of 2-D matrices.

3. The iterative-parallel strategy is used to implement the tFFT's convolution streamline on NVIDIA GTX GPU (see Sect. 5).

## 2 Related Work

Since the year of 2012, Convolutional neural networks (CNNs) have been rapidly developed as one of the most popular deep-learning architecture in computer vision community [8]. CNNs are widely used for image and speech recognition, object detection, semantic segmentation and other artificial intelligence fields. Yet CNNs require large amount of training data to guarantee high learning accuracy. Millions of weights learned from convnet layers exhaust a great deal of training cost. Given this, the performance improvement of CNNs have received significant attention in recent years.

The CUDA Deep Neural Network (cuDNN) [9] is a baseline framework, using CUDA to program for NVIDIA GPU [10], and performing convolutions directly in time domain for convnet layers. For the cuDNN based frameworks, straightforward convolutions are implemented between the inputs and the filters in each layer. cuDNN obtains the arithmetic complexity in proportion to the depth of convnets and the size of batches, this leads that cuDNN performs convolutions in exhaustive way for deep networks with large batches. Direct convolution adds a large overhead to GPU bandwidth, this can only trade off a few limited convolution time. Given this, cuDNN was improved by introducing Fast Fourier Transformation into framework, called cuFFT [6]. cuFFT implements convolutions in Fourier frequency domain and reduces the arithmetic complexity of convnet layers. cuFFT transforms the time domain convolutions into the Fourier domain point-wise products, this transformation strategy accelerates training and inference for modern CNNs with large number of feature maps. Yet cuFFT employs 2-radix FFT [11] to do convolutions for any size of filters. When the size of filer is much smaller than the size of input feature map, the filter must be padded to the equal size of input which is a power of 2 [12, 13]. This padding step significantly affects the speed of convolutions, especially for the state-of-the-art convnet architectures which widely use small filters [14–16]. For example, in the 2015 ILSVRC competition, Residual Networks (ResNet) won the championship on many objects including image classification, object detection and semantic segmentation etc. ResNet is a deep convolutional neural networks with $3 \times 3$ filter kernels in all layers. Therefore cuFFT is suboptimal for speeding up the state-of-the-art CNNs. Motivated by the limitations of cuFFT, fbFFT [7] emerged as the only open-source library for implementing FFT-based convolutions in certain sizes of batches. For fbFFT, the twiddle factors and Fourier coefficients are stored in registers per thread, it implements Fast Fourier Transforms within a single warp shuffles. fbFFT significantly relieves overhead to GPU bandwidth, being 1.5× faster than cuFFT at typical sizes of interest. Yet fbFFT obtains better performance only in certain batches, it performs poorly for batches with size of less 8 or over 64. To this end, a fast algorithm (FA) [17] for CNNs is proposed to overcome the limitation of fbFFT, being sensitive to batch sizes from 1 to 64. FA uses Winograd's minimal filtering algorithms [18, 19] to compute convolutions over small tiles of the inputs. FA is a tile-based convolution algorithm, obtaining minimal complexity convolutions by using polynomial multiplication strategy in time domain. FA shows a better performance only at batch sizes from 1 to 64 in parallel with a modest performance at batch sizes of over 128. Given this, we propose a tile-based Fast Fourier Transformation algorithm to speed up the state-of-the-art CNNs at batch sizes from 1 to 128.

## 3 FFT-based Convolution

Massive training data required for accurate labels causes thousands of convolutions between input feature maps and filter kernels. Thus the critical operations for training convnets can be generally regarded as convolutions between multi-sized matrices. FFT-based convolution strategy accelerates training and inference by regarding convolutions as point-wise products in Fourier frequency domain [1, 2]. These convolutions are performed in three significant steps: forward propagation, backward propagation and update of kernel weights. Each convnet layer undergoes these three steps, before explaining the FFT-based convolution by theses three steps, we clear some definitions.

Assume a convnet layer consists of a set $f$ of input channels $X_{s,i}$ with size $H \times W$, $i \in f \subseteq s \in S'$, $s$ represents a minibatch of size $S1$. $S'$ represents a set of minibatches with size of 1–128. Each input and output feature can be part of a minibatch $S'$. The filter weights are denoted as $W_{j,i}$ with size $R \times S$, $j \in f' \subseteq s$, $f'$ represents a set of output feature matrices $Y_{s,j}$ with size $M \times N$. Given this, elements of input feature matrices are denoted as $X_{s,i,n1,n2}$, $n1 = 0, \dots, H-1$; $n2 = 0, \dots, W-1$; Elements of filters are denoted as $W_{j,i,k1,k2}$, $k1 = 0, \dots, R-1$, $k2 = 0, \dots, S-1$; Elements of output feature matrices are denoted as $Y_{s,j,m1,m2}$, $m1 = 0, \dots, M-1$, $\mathrm{m2} = 0, \dots, N-1$.

For the forward propagation, convolutions are performed between the input channels and the filter kernels. The convolution results are a set of output feature matrices. The implementation of this step is defined as:

$$Y_{s,j,m1,m2} = \sum_{i \in f} X_{s,i,n1,n2} * W_{j,i,k1,k2} \tag{1}$$

For the backward propagation, convolutions are performed between the loss gradients of the output feature matrices and the transposed filters [5]. Thus the loss gradients of the input feature matrices are defined as:

$$\frac{\partial L}{\partial X_{s,i,n1,n2}} = \sum_{j \in f'} \frac{\partial L}{\partial Y_{s,j,m1,m2}} * W^{-1}_{j,i,k1,k2} \tag{2}$$

Finally, the weights of filters are renewed by convolutions between the output gradients and the input feature matrices, these weight parameters are updated by the loss gradients with respect to filter kernels:

$$\frac{\partial L}{\partial W_{j,i,k1,k2}} = \sum_{s \in S1} \frac{\partial L}{\partial Y_{s,j,m1,m2}} * X_{s,i,n1,n2} \tag{3}$$

All formulas are computed in type of single precision float point number. $*$ denotes cross-correlation. The stride of filter is denoted as $str$, the outputs $Y_{s,j}$ are of size $((H-R)/str+1) \times ((W-S)/str+1)$. When the size of outputs meets the condition of $M*N \geq H*W+R*S-1$, the time domain circular convolutions with size of $M*N$ can be calculated by point-wise products in the Fourier frequency domain. When the condition can not be satisfied, zero padding around the borders of inputs or filters are introduced to avoid overlapping confusion. The convolutions between inputs and filters are equivalent to the multiplications between their Fourier transforms:

$$Y_{s,j,m1,m2} = \sum_{i \in f} X_{s,i,n1,n2} * W_{j,i,k1,k2} = \sum_{i \in f} F^{-1}(F(X_{s,i,n1,n2}) \cdot F(W_{j,i,k1,k2})) \tag{4}$$

where $\cdot$ denotes the point-wise product between 2-D matrices.$F()$ denotes that Fast Fourier Transform method is used to compute Discrete Fourier Transforms. Using DFT directly requires $O(N^2)$ operations to transform a set of complex numbers from time domain to frequency domain. An FFT is any method to compute the same results in $O(N\log(N))$ operations. In the presence of round-off error, FFT algorithms are much more accurate than evaluating the DFT definition directly. For details of FFT, see references [20, 21].

It is well known that Discrete Fourier Transforms can be accelerated by 2-radix Fast Fourier Transform method [6]. Using formula (4) to perform direct convolutions requires $((\tilde{H}-\tilde{R})/str+1)((\tilde{W}-\tilde{S})/str+1)\times\tilde{R}\tilde{S}$ multiplications for each channel, whereas the FFT-based convolution requires $3C\tilde{H}^2\log\tilde{H}+\tilde{H}^2$ complex multiplications or $12C\tilde{H}^2\log\tilde{H}+4\tilde{H}^2$ real multiplications. Each FFT requires $4C\tilde{H}^2\log\tilde{H}$ real multiplications. Applied to all convnet layers in the forward step, FFT-based convolutions requires $4C\tilde{H}\tilde{W}\log_2\tilde{H}(f'S1+fS1+f'f)+4S1f'f\tilde{H}\tilde{W}$ real multiplications. As for the backward step, it requires $4C\tilde{M}\tilde{N}\times\log_2\tilde{M}(f'S1+fS1+f'f)+4S1f'f\tilde{M}\tilde{N}$ real multiplications, and $4C\tilde{H}\log_2\tilde{H}\tilde{W}(f'S1+fS1+f'f)+4S1f'f\tilde{H}\tilde{W}$ real multiplications for updating of filter weights.

The conventional FFT-based convolution method uses zero padding strategy to keep the size of inputs or filters being a power of 2. For example, an input feature map is of size $31\times34$, this size will be padded to $64\times64$ to meet a power of 2, even worse when the filter is of size $3\times3$ (Fig. 1), the filter matrices will be equally padded to $64\times64$, this indicates that convolutions are done between two bigger size matrices compared to initial small size filters. For large scale training data, the arithmetic complexity of convolutions will be increased dramatically. The training time of CNNs will be increased under this over padding implementations. This paper uses a tile-based Fast Fourier Transform algorithm to overcome this inefficiency, we will introduce it in the next section.

## 4 Tile-based Fast Fourier Transforms (tFFT)

We introduce the tile-based decomposition strategy into Fast Fourier Transforms. The conventional large scale convolutions are decomposed into tile sized Fast Fourier Transforms. These tile sized transformations allow for fast parallel computation in GPU. The tFFT will be presented in this section, it implements tile-based convolutions in Fourier domain for CNNs. The implementations of tFFT is broadly outlined as: first the large size inputs are logically divided into tiles being close to the size of filter kernels, the transformations are performed to these logically divided tiles and filters in Fourier domain. Second we recursively decompose the prior tile divided inputs into butterfly operations, these operations are calculated in parallel streamlines of GPU. The large size convolutions are transformed into tile sized point-wise-products in Fourier domain. Finally the inverse transformation is performed to the sum, this reduces the cost of transformation. The algorithms are designed for the forward and backward passes respectively.

### 4.1 Forward Propagation (fprop)

In the forward propagation pass, the outputs are computed by a sum of convolutions between the input feature matrices and the filters. The feature matrices are subdivided into tile sized matrices, the coordinates of each tile are denoted as $(\alpha,\beta)$. We nest the conventional fprop formula (1) to construct the tile-based convolutions for each channel $i$, filter $j$ and tile $(\alpha,\beta)$ as:

$$Y_{s,j,\alpha,\beta}=\sum_{i\in f}X_{s,i,\alpha,\beta}*W_{j,i} \tag{5}$$

The formula (5) is a sum of convolutions between tile sized input matrices and the weight matrices, being transformed into tile sized point-wise-products in Fourier domain as:

$$Y_{s,j,\alpha,\beta} = \sum_{i\in f} F^{-1}(F(X_{s,i,\alpha,\beta})\cdot F(W_{j,i})) = F^{-1}\left[\sum_{i\in f} F(X_{s,i,\alpha,\beta})\cdot F(W_{j,i})\right] \quad (6)$$

where the inverse transformation $F^{-1}$ is performed for all channels at once time. The cost of multiplications used for inverse transformation are apportioned among these channels. We rewrite the formula (6) as:

$$Q_{s,j,\alpha,\beta} = \sum_{i\in f} F(X_{s,i,\alpha,\beta})\cdot F(W_{j,i}) \quad (7)$$

The discrete Fourier transforms with respect to the tiles $X_{s,i,\alpha,\beta}$ is decomposed into $\alpha$-point and $\beta$-point DFT, the filter kernels $W_{j,i}$ is implemented in the same way. The decomposition procedure is denoted as:

$$\begin{aligned} F(X_{s,i,\alpha,\beta}) &= \sum_{n1=0}^{\alpha-1}\sum_{n2=0}^{\beta-1} X_{s,i,n1,n2} W_{\alpha}^{n1f1} W_{\beta}^{n2f2} \\ &= \sum_{n1=0}^{\alpha-1} W_{\alpha}^{n1f1} \sum_{n2=0}^{\beta-1} X_{s,i,n1,n2} W_{\beta}^{n2f2} \\ &= \sum_{n1=0}^{\alpha-1} \tilde{X}_{s,i,n1,f2} W_{\alpha}^{n1f1} \end{aligned} \quad (8)$$

where $\tilde{X}_{s,i,n1,f2}$ is a $\beta$-point DFT, $F(X_{s,i,\alpha,\beta})$ is a $\alpha$-point DFT. Thus the convolutions are decomposed into a sum of tile-sized DFTs, this decomposition step can be repeated until two elements left which consists of a butterfly operation. We use the parallel iterative strategy to compute these tile-sized DFTs in GPU. Each butterfly operation is computed by an arithmetic logic unit (ALU) in each level of tile-sized DFT, the tile-sized transformations are performed iteratively between these levels. A tile-based FFT uses $(\tilde{\alpha}\tilde{\beta})/2$ GPU processing unit per tile. The $(\tilde{\alpha}\tilde{\beta})/2$ number of butterfly operations are performed in parallel manner. The $\log(\tilde{\alpha}\tilde{\beta})$ number of tile-sized transformations are performed in iterative manner. The total execution time is calculated as $T_{bf}\log(\tilde{\alpha}\tilde{\beta})$ for each tile. $T_{bf}$ denotes the time spent by a butterfly operation. Thus our tile-based FFT uses $4\log(\tilde{\alpha}\tilde{\beta})$ real multiplications, whereas the conventional FFT-based algorithm uses $2\tilde{\alpha}\tilde{\beta}\log(\tilde{\alpha}\tilde{\beta})$ real multiplications for each tile. This indicates that the arithmetic complexity of tFFT is much smaller than the cuFFT.

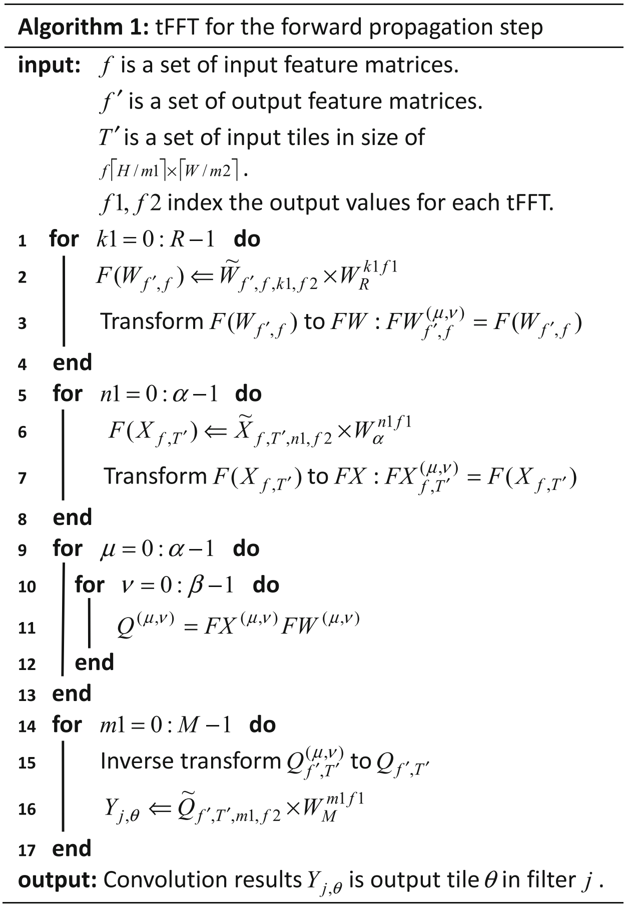
The tFFT is employed in each convnet layer for speeding up convolution implementations. Each layer contains the $f$ number of inputs, each input is divided into $T = \lceil H/m1\rceil \times \lceil W/m2\rceil$ tiles. The tFFT is performed for each tile and filter, the paired convolution results are iteratively integrated into a sum in each channel. We use notation $\theta$ to label the minibatch coordinates of tiles, and notation $(\mu, \nu)$ to label the point-wise products of each component in tFFT. Given this, we rewrite the formula (7) as:

$$Q_{j,\theta}^{(\mu,\nu)} = \sum_{i\in f} F\left(X_{i,\theta}^{(\mu,\nu)}\right) F\left(W_{j,i}^{(\mu,\nu)}\right) \quad (9)$$

The formula (9) is a matrix multiplication between tile-sized FFT coefficients and twiddle factors, being further rewritten as:

$$Q^{(\mu,\nu)} = FX^{(\mu,\nu)}FW^{(\mu,\nu)} \quad (10)$$

The multiplications in formula (10) can be efficiently computed on GPU [22, 23]. Figure 2 shows the tFFT's convolution streamline in forward propagation. The tFFT's convolution streamline is a parallel pipeline for performing tFFT. In this paper, the tile size is set to be 2, i.e., $\alpha = \beta = 2$, the $\alpha$-point DFT is performed between $X_{N,1,0,0}$ and $X_{N,1,0,1}$, thee $\beta$-point DFT is performed between $X_{N,1,1,0}$ and $X_{N,1,1,1}$, the computation results are denoted by $F(X_{N,1,2,2}) \rightarrow F(X_{N,f,2,2})$. This is a diagram explanation for Eq. 8–10. Our tile-based Fast Fourier Transforms algorithm for the forward propagation is presented in Algorithm 1.



**Algorithm 1:** tFFT for the forward propagation step

```
input:  $f$ is a set of input feature matrices.
        $f'$ is a set of output feature matrices.
        $T'$ is a set of input tiles in size of
        $f\lceil H/m1\rceil\times\lceil W/m2\rceil$.
        $f1, f2$ index the output values for each tFFT.
for $k1 = 0 : R-1$ do
    $F(W_{f',f}) \Leftarrow \widetilde{W}_{f',f,k1,f2} \times W_R^{k1f1}$
    Transform $F(W_{f',f})$ to $FW : FW_{f',f}^{(\mu,\nu)} = F(W_{f',f})$
end
for $n1 = 0 : \alpha-1$ do
    $F(X_{f,T'}) \Leftarrow \widetilde{X}_{f,T',n1,f2} \times W_\alpha^{n1f1}$
    Transform $F(X_{f,T'})$ to $FX : FX_{f,T'}^{(\mu,\nu)} = F(X_{f,T'})$
end
for $\mu = 0 : \alpha-1$ do
    for $\nu = 0 : \beta-1$ do
        $Q^{(\mu,\nu)} = FX^{(\mu,\nu)} FW^{(\mu,\nu)}$
    end
end
for $m1 = 0 : M-1$ do
    Inverse transform $Q_{f',T'}^{(\mu,\nu)}$ to $Q_{f',T'}$
    $Y_{j,\theta} \Leftarrow \widetilde{Q}_{f',T',m1,f2} \times W_M^{m1f1}$
end
output: Convolution results $Y_{j,\theta}$ is output tile $\theta$ in filter $j$.
```

## 4.2 Backward Propagation (bprop)

The backpropagation adjusts the weights of neurons by computing the loss gradients with respect to the input feature matrices and the filters. During the backward step, the input feature matrices consist of a set of tile sized matrices, the gradients of the inputs correspond to a sum of gradients with respect the tile-sized matrices. These tile-based gradients are equal to the convolutions between the transposed filters and the loss gradients of the outputs, being propagated to the previous layer for updating the gradients of the weights. The loss gradients of the weights are calculated by convolving the input feature matrices with the gradients of

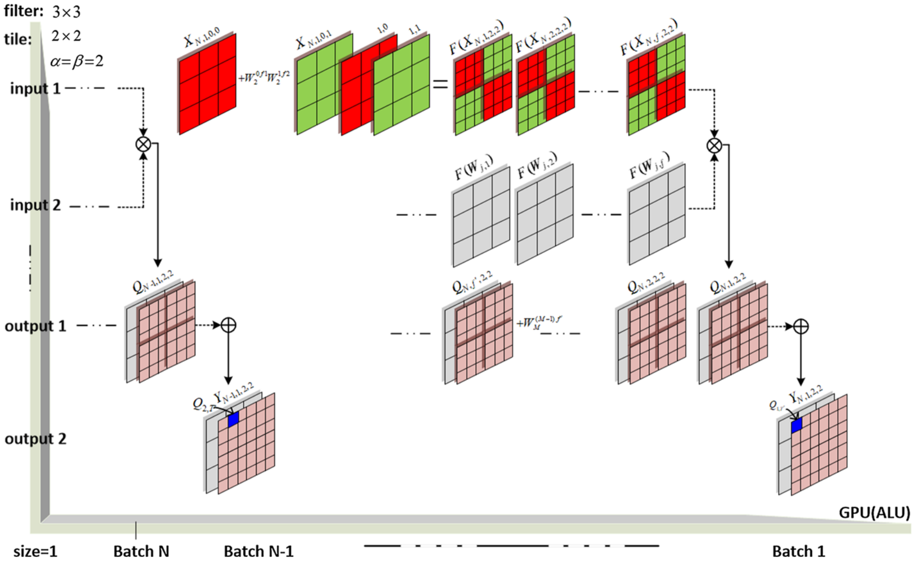


**Fig. 2** The tFFT's convolution streamline in forward propagation pass. The gradients of the weights can be computed by the same convolution streamline that is used for forward pass

the outputs. We nest the formula (2) to design the tile-based gradient convolutions for each channel $i$, filter $j$ and tile $(\alpha, \beta)$ as:

$$\frac{\partial L}{\partial X_{s,i,\alpha,\beta}} = \sum_{j\in f'} \frac{\partial L}{\partial Y_{s,j,\alpha,\beta}} * W_{j,i}^{-1} \tag{11}$$

The time domain convolutions correspond to the Fourier domain multiplications between the transposed filters and the gradients with respect to the tile-sized output feature matrices. Thus the convolutions in formula (11) are transformed to the point-wise products in Fourier domain, we rewrite the formula (11) as:

$$\frac{\partial L}{\partial X_{s,i,\alpha,\beta}} = F^{-1}\left[\sum_{i\in f'} F\left(\frac{\partial L}{\partial Y_{s,j,\alpha,\beta}}\right)\cdot F(W_{j,i}^{-1})\right] \tag{12}$$

We perform the tile-based Fourier transformation for each gradient of output by using the same implementations that is used for forward propagation. Each convnet layer generates the $f'$ number of output feature matrices, the gradients of outputs are divided into $T = \lceil M/n1\rceil \times \lceil N/n2\rceil$ tiles of size $(n1+k1-1)\times(n2+k2-1)$. We denote the coordinates of tiles as $\varphi$ in each minibatch. Given this, we rewrite the formula (12) as:

$$Q_{i,\varphi}^{(\mu,\nu)} = \sum_{j\in f'} F\left(\frac{\partial L^{(\mu,\nu)}}{\partial Y_{j,\varphi}}\right) F\left(W_{j,i}^{-(\mu,\nu)}\right) \Leftrightarrow Q^{(\mu,\nu)} = FL^{(\mu,\nu)}FW^{(\mu,\nu)} \tag{13}$$

The same as the prior inferences, we calculate the gradient with respect to the weights by convolving the outputs' gradients with the inputs:

$$\begin{aligned}\frac{\partial L}{\partial W_{j,i}} &= \sum_{s\in S1} \frac{\partial L}{\partial Y_{s,j,\alpha,\beta}} * X_{s,i,\alpha,\beta} \Leftrightarrow Q_{j,i}^{(\mu,\nu)} \\ &= \sum_{s\in S1} F\left(\frac{\partial L^{(\mu,\nu)}}{\partial Y_{j,\varphi}}\right) F\left(X_{i,\varphi}^{(\mu,\nu)}\right) \Leftrightarrow Q^{(\mu,\nu)} = FL^{(\mu,\nu)} FX^{(\mu,\nu)}\end{aligned} \tag{14}$$

Based on the above inferences, we use the Algorithm 1 to compute the gradients with respect to the inputs, and Algorithm 2 to compute the gradients with respect to the weights. Our tile-based Fast Fourier Transformation algorithm for the backward propagation step is presented in Algorithm 2.

**Algorithm 2:** tFFT for the backward propagation step

```
input:
    $T_1$ is a set of input tiles in size of $f\lceil H/m1\rceil\times\lceil W/m2\rceil$.
    $T_2$ is a set of output tiles in size of $f'\lceil M/n1\rceil\times\lceil N/n2\rceil$.
for $n1 = 0 : \alpha - 1$ do
    $F(X_{f,T_1}) = \widetilde{X}_{f,T_1,n1,f2} \times W_{\alpha}^{n1f1}$
    Transform $F(X_{f,T_1})$ to $FX : FX_{f,T_1}^{(\mu,\nu)} = F(X_{f,T_1})$
end
for $m1 = 0 : M - 1$ do
    $F(Y_{f',T_2}) = \widetilde{Y}_{f',T_2,m1,f2} \times W_{M}^{m1f1}$
    Transform $F(Y_{f',T_2})$ to $FL : FL_{f',T_2}^{(\mu,\nu)} = F(Y_{f',T_2})$
end
for $\mu = 0 : \alpha - 1$ do
    for $\nu = 0 : \beta - 1$ do
        $Q^{(\mu,\nu)} = FL^{(\mu,\nu)} FX^{(\mu,\nu)}$
    end
end
for $k1 = 0 : R - 1$ do
    Inverse transform $Q_{f',f}^{(\mu,\nu)}$ to $Q_{f',f}$
    $\partial Y_{j,i} = \widetilde{Q}_{f',f,k1,f2} \times W_{R}^{k1f1}$
end
output: $\partial Y_{j,i}$ is the gradient with respect to the weights
    per input $i$ and filter $j$.
```

### 4.3 Arithmetic complexity analysis

Up to now, the fastest convolution algorithm for CNNs is a tile-based time domain convolution algorithm build upon Winograd's minimal filtering algorithms. This fast algorithm with tile size of $\alpha \times \beta$ requires $S1\lceil H/m1\rceil\lceil W/m2\rceil f \times f' \times \tilde{\alpha}\tilde{\beta}$ multiplications. Using the split-radix

**Table 1** The architectures of convnet layers for AlexNet, VGGNet-19 and ResNet-34

| | Alex-Net | | VGGNet-19 | | ResNet-34 | |
|---|---|---|---|---|---|---|
| Layer | $H \times W \times S$ | $K/d$ | $H \times W \times S$ | $K/d$ | $H \times W \times S$ | $K/d$ |
| conv1 | $227 \times 227 \times 1$ | 96/1 | $224 \times 224 \times 3$ | 64/2 | $224 \times 224 \times 3$ | 64/1 |
| conv2_x | $27 \times 27 \times 96$ | 256/1 | $112 \times 112 \times 64$ | 128/2 | $112 \times 112 \times 64$ | 64/6 |
| conv3_x | $13 \times 13 \times 256$ | 384/1 | $56 \times 56 \times 128$ | 256/4 | $56 \times 56 \times 128$ | 128/8 |
| conv4_x | $13 \times 13 \times 384$ | 384/2 | $28 \times 28 \times 256$ | 512/4 | $28 \times 28 \times 256$ | 256/12 |
| conv5_x | $13 \times 13 \times 256$ | 256/1 | $14 \times 14 \times 512$ | 512/4 | $14 \times 14 \times 512$ | 512/6 |
| | FC:1000 | /1 | FC:1000 | /3 | FC:1000 | /1 |

Alex-Net layers uses $11 \times 11$, $5 \times 5$ and $3 \times 3$ filters. VGGNet-19 and ResNet-34 uses $3 \times 3$ filters in all layers. $K$ indicates the number of filters in each layer. $D$ indicates the number of repeated convolution layer

FFT algorithm [24] and the fast CGEMM [25, 26] reduces the arithmetic complexity further, yet a large tile size of $64 \times 64$ is required to meet the efficiency of Fourier transformation. A $64 \times 64$ tile requires 4096 byte memory to transform a single filter channel into the output unit, yet the state-of-the-art architecture of CNNs widely uses small filters with size of $3 \times 3$. This leads to a large number of wasted GPU memories and a serious reduction of training speed. This paper presents a faster algorithm for the state-of-the-art CNNs using a tile-based Fast Fourier Transforms. Our tFFT decomposes the dimension of convolutions by implementing tile sized transformations in Fourier domain. The performance of tFFT is not limited to a large tile size. For example, when the tile is size of $4 \times 4$, the conventional FFT-based convolution algorithm uses $2\tilde{H}^2 \log \tilde{H} \times (Sf' + Sf + ff') + 4Sff'\tilde{H}^2$ multiplications, the tFFT uses $S\lceil H/m1\rceil\lceil W/m2\rceil f \times f' \times 4\log(4 \times 4)$ multiplications, this is a 3.41× reduction of the arithmetic complexity. For the state-of-the-art architectures of CNNs, the arithmetic complexity of tFFT is $S\lceil H/m1\rceil\lceil W/m2\rceil f \times f' \times 4\log(\tilde{\alpha}\tilde{\beta})$ in terms of implementing real multiplications. Our tFFT decomposes the large input feature matrices into tile sized convolutions and employs tile-based Fast Fourier Transformation algorithm to reduce the computation complexity efficiently, yet all of the FFT-based convolution methods use the complex numbers to do multiplications, this limits the performance of tFFT. To this end, we introduce fast CGEMM into tFFT to further accelerate convolution implementations in layers. Compared to the conventional FFT-based convolution algorithm, tFFT combined with CGEMM presents at least 4.55× reduction of the arithmetic complexity. Other state-of-the-art comparison results will be presented in the experiment section.

## 5 Experiments

In this section, we perform a systematic comparison of tFFT to the state of the art. We use three popular convolutional neural networks which are AlexNet [27], VGGNet-19 and ResNet-34 for all experiments, the architecture of convolution layers for these networks are presented in Table 1. We use single precision metric to evaluate the accuracy of tFFT, and millisecond metric to evaluate the speed of tFFT. We compute the Tflop (floating point operations) values of each layer by the Eq. (15) in run time of milliseconds.

$$T = (Sff' \times k1 \times k2 \times m1 \times m2)/(run\ time(ms)) \tag{15}$$

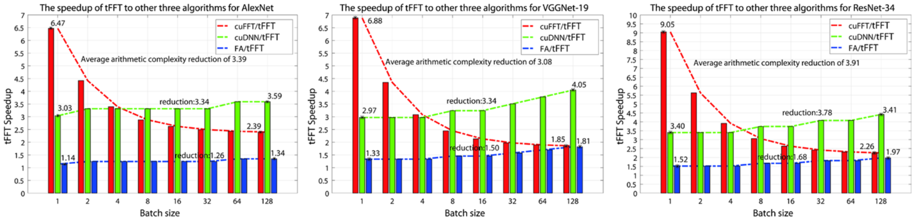


**Fig. 3** The relative performance of tFFT compared to the state-of-the-art algorithms. The speedup is measured by the running time ratio of cuFFT, cuDNN, FA to tFFT

We use these Tflop values to evaluate the efficiency of tFFT in reduction of arithmetic complexity. When the Tflop value for a convnet layer is higher than the GPU peak throughput, this indicates that the algorithm performs convolutions in more efficient way in this layer.

### 5.1 The Relative Performance of tFFT

In this section, we test the relative performance of tFFT compared to other three advanced algorithms at batch sizes from 1 to 128. We use the Algorithm 1 to perform the forward propagation for AlexNet, VGGNet-19 and ResNet-34. The run time is calculated for each CNNs at various batch sizes. We quantify the speedup of GPU implementations of these algorithms on NVIDIA GTX 1080i GPU with peak throughput of 5.67 Tflop. Figure 3 shows the speedup performance of our tFFT compared to the state-of-the-art algorithms which are cuFFT, cuDNN and Winograd-based fast algorithm (labeled FA). We compute the ratio of each algorithm to tFFT in terms of the run time, the run time is calculated by counting the convolution time in millisecond on NVIDIA GTX 1080i GPU. A higher ratio suggests that tFFT is more faster than other algorithms with respect to training and inference of convolutional neural networks. Unless the radio is less than 1, otherwise tFFT performs better at these networks.

For AlexNet, the maximum ratio of cuFFT to tFFT is 6.47× at $S = 1$, along with the increase of batch sizes, this ratio is reduced to 2.40× at $S = 128$. This indicates that tFFT performs better than cuFFT especially for small batch sizes. This is due to the tile-based decomposition strategy used by tFFT. The traditional FFT-based convolution algorithm (cuFFT or fbFFT) uses a single input of size $H \times W$ to perform fast Fourier transformation. This transformation is inefficient for modern convolutional neural networks (AlexNet, ResNet etc.) with large batches and small filters. For tFFT, the large inputs are divided into small tiles, our algorithm decomposes the large Fourier transforms into a tile-sized Fourier transforms in forward and backward passes. Compared to traditional FFT-based algorithms, tFFT obtains a better performance at batch sizes from 1 to 128, the next experiment shows that the tFFT's superior performance will tend to be stable at larger batch sizes. Compared to cuFFT, we obtain an average arithmetic complexity reduction of 3.9 for AlexNet in forward propagation pass. cuDNN directly performs convolutions in time domain without transformation. The ratio of cuDNN to tFFT is 3.03× at $S = 1$ and 3.59 at $S = 128$, this indicates that tFFT keeps the opposite proportion of increment to cuDNN in terms of convolution time. At large $S$, cuDNN spends much time for convolutions, in contrast to cuDNN, tFFT reduces the convolution time for AlexNet and ResNet-34. This suggests that tFFT reduces the arithmetic complexity for CNNs with different depths at large batches. Unless the batch size is more than 8, cuDNN shows a higher performance to cuFFT, this verifies the fact that cuFFT is

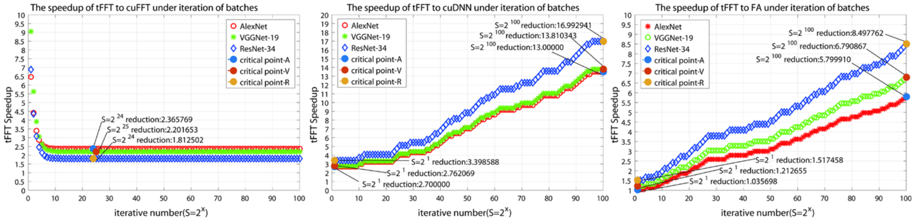


**Fig. 4** The stability of tFFT to other state-of-the-art algorithms at large iterative number of batches. Critical point-A, -V and -R denote the stable performance of tFFT, where the speedup ratio is proportional to the deep of networks

more applicable to large size convolutions. The ratio of FA to tFFT is 1.33 × at $S = 1$, this is a modest gains for relative performance compared to other two algorithms. Yet the curve of arithmetic complexity keeps rising, tFFT obtains 1.81× speedup over FA at size 128. FA is limited to small tile sizes, FA requires more transformation complexity to trade off the minimal multiplication times gained from small tile size. The worst tFFT performance occurs for layers where tile sizes are equal to the input sizes, in this case tFFT performs equally to cuFFT in terms of multiplication times, while FA shows a higher transformation complexity surpassing any savings in multiplication stages, this part will be tested in Sect. 5.3.

To this end, compared to classical FFT-based algorithms, tFFT obtains a better performance at batch sizes from 1 to 128. tFFT achieves an average arithmetic complexity reduction of 3.12× for ResNet-34, 2.64× for VGGNet-19 and 2.66× for AlexNet, and shows state-of-the-art results at batch sizes from 1 to 128.

## 5.2 The Robustness of tFFT

We evaluate the robustness of tFFT and compare the speedup performance of tFFT to cuFFT, cuDNN and FA under large iterative number of batches. The iterative number of batches are ranged from 1 to 100, we balance the memory latency against convolution accuracy by reducing the multiplication precision to half floating precision. Figure 4 shows the stability of tFFT measured by millions of iterative convolutions. Compared to cuFFT, tFFT obtains an arithmetic complexity reduction of 2.36 at $S = 2^{24}$ for AlexNet and 1.81 for ResNet-34. Along with the deepening of networks, the relative performance of tFFT shows a modest increment to cuFFT. While the iterative number of batches reaches the critical point, for example, the number is 24 for AlexNet, the speedup of tFFT to cuFFT remains stable throughout. This indicates that tFFT achieves at least 1.81× speedup over cuFFT at $S = 2^{100}$ batches.

Compared to cuDNN, tFFT obtains an arithmetic complexity reduction of 3.39 at $S = 2^1$ and 16.99 at $S = 2^{100}$ for ResNet-34. Relative performance gains for AlexNet and VGGNet-19 are more modest than ResNet-34, this suggests that tFFT performs much better than direct convolution in terms of computing convolutions for deeper nets at large batches. Under the increasing of the iterative number of batches, the arithmetic complexity for implementation of tFFT tends to be reduced in stable manner, unless the GPU memory used for convolutions reaches saturation. The reduction ratio is approximate to the speedup of tFFT per batch. With respect to ResNet-34 at batch sizes from $2^1$ to $2^{100}$, the reduction ratio can reach 0.13 per patch.

Compared to FA, tFFT obtains an arithmetic complexity reduction of 8.49 for ResNet-34, 6.79 for VGGNet-19 and 5.79 for AlexNet at $S = 2^{100}$ batches. Two tile-based convolution algorithms show a different performance for large iterative convolutions. tFFT outperforms FA at large iterative number of batches. The transformation complexity of FA increases quadratically with the tile size, yet the transformation complexity of tFFT tends to be increasing in more steady manner, a set of sampled critical points are shown in Fig. 4, the reduction ratio can reach 0.06 per patch for ResNet-34. The worst performance occurs for tFFT when the tile size is equal to input size in default, the arithmetic complexity of tFFT equals to the complexity of conventional FFT-based convolutions. For AlexNet, the worst performance appears in the first layer where the tile size is set to be 11, while FA has tiles of size 4, the arithmetic complexity is only reduced by 1.03 for tFFT. This indicates that in spite of using much bigger tiles, tFFT still performs a little better than FA. Yet this modest performance is only for the first layer of AlexNet, the throughput for tFFT is still 1.91× higher than FA for deeper networks at large batches. The next experiment will show the throughput by each layer for tFFT and FA.

To this end, tFFT performs better than the state of the arts in terms of computing convolutions for deeper nets at large batches. tFFT achieves at least 1.81× speedup over cuFFT at $S = 2^{100}$ batches, 13.0× speedup over cuDNN and 5.79× speedup over FA.

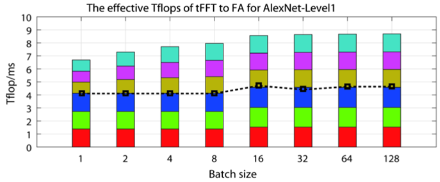
### 5.3 The Throughput of tFFT

We evaluate the acceleration of GPU implementations of tFFT and FA on NVIDIA GTX 1080i GPU with peak throughput of 5.67 Tflop. The formula (14) is used for yielding the throughput a tile-based convolution algorithm needs to achieve in each layer. We carry out throughput experiments with AlexNet at batch sizes from 1 to 128. We compute Tflop values for three propagation passes respectively (fprop, bprop and wgrad) in each convnet layer. The Tflop value is inversely proportional to the arithmetic complexity of an algorithm. For our GPU configuration, when Tflop value is higher than 5.67 TFLOPS, this indicates that the algorithm reduces the arithmetic complexity of convolutions to a certain degree. A higher value of Tflop indicates a higher reduction of arithmetic complexity.

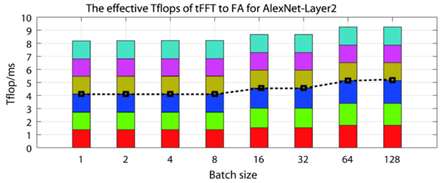
Figure 5 shows the throughput of implementation tFFT and FA by each AlexNet layer at batch sizes from 1 to 128. Looking at the dotted line in the left half of Fig. 5, it shows an upward trend, this indicates that the worst performance occurs for $S = 1$, given this, we tabulate the throughput at $S = 1$ and analysis the worst performance of tFFT. For the forward propagation pass, tFFT is 1.78× at $S = 1$, the throughput is 15.34 TFLOPS. For the backward propagation pass, tFFT still performs better than FA, and yields 14.97 TFLOPS throughput and 1.78× speedup. For computing the gradient of weights, the performance of tFFT is still very good at 15.53 TFLOPS and 1.79× speedup.

Compared to FA, tFFT achieves a better performance at every layer and batch size. The throughput of FA is negatively affected by a little bigger tile size. In the third convnet layer, the tile is size of 6($\tilde{\alpha} = 6$), for the forward pass, tFFT is 2.71× faster than FA at $S = 1$, the throughput is 10.2 TFLOPS. FA shows a dramatically increment of transformation complexity, unless reducing the accuracy of convolutions. tFFT allows for stable transformation without reducing precision. For layer 4, the tile size is equal to the input size, the worst performance of tFFT appears for this layer when $S = 1$. The throughput of tFFT is 5.10 TFLOPS in forward pass, where FA performs at 2.66 TFLOPS.

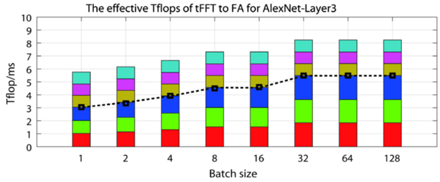
To this end, tFFT speeds up convolutions by relative 5.10 to 15.34 TFLOPS on NVIDIA GTX 1080i GPU with peak throughput of 5.67, even obtaining bigger gains under larger size batches.



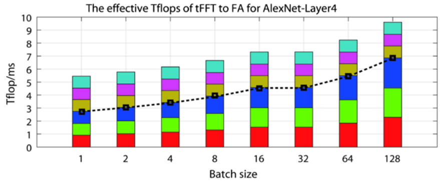
| Layer1 | FA/ms | | tFFT/ms | | speadup |
|---|---|---|---|---|---|
| $f=1,\ f'=96,\ \mathit{filter\ size}: 11\times11,\ \mathit{input\ size}: 227\times227,\ \widetilde{\alpha}=4$ | | | | | |
| | time | Tflop | time | Tflop | times |
| fprop | 4.09 | 8.57 | 2.54 | 15.34 | 1.78 × |
| bprop | 4.19 | 8.37 | 2.60 | 14.97 | 1.78 × |
| wgrad | 4.04 | 8.67 | 2.51 | 15.53 | 1.79 × |



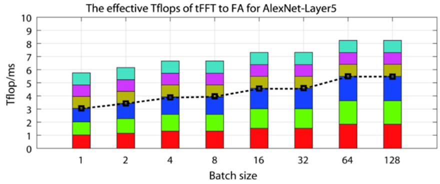
| Layer2 | FA/ms | | tFFT/ms | | speadup |
|---|---|---|---|---|---|
| $f=96,\ f'=256,\ \mathit{filter\ size}: 5\times5,\ \mathit{input\ size}: 27\times27,\ \widetilde{\alpha}=4$ | | | | | |
| | time | Tflop | time | Tflop | times |
| fprop | 16.50 | 9.05 | 16.27 | 9.18 | 1.01 × |
| bprop | 16.90 | 8.83 | 16.68 | 8.96 | 1.01 × |
| wgrad | 16.31 | 9.16 | 16.08 | 9.29 | 1.01 × |



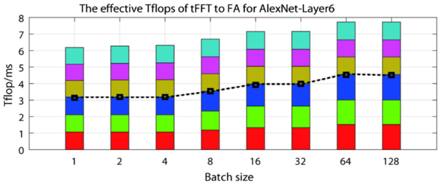
| Layer3 | FA/ms | | tFFT/ms | | speadup |
|---|---|---|---|---|---|
| $f=256,\ f'=384,\ \mathit{filter\ size}: 3\times3,\ \mathit{input\ size}: 13\times13,\ \widetilde{\alpha}=6$ | | | | | |
| | time | Tflop | time | Tflop | times |
| fprop | 39.61 | 3.77 | 14.60 | 10.2 | 2.71 × |
| bprop | 40.60 | 3.68 | 15.01 | 9.95 | 2.68 × |
| wgrad | 39.13 | 3.82 | 14.47 | 10.3 | 2.70 × |



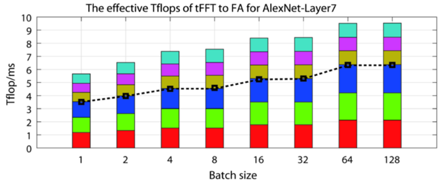
| Layer4 | FA/ms | | tFFT/ms | | speadup |
|---|---|---|---|---|---|
| $f=384,\ f'=384,\ \mathit{filter\ size}: 3\times3,\ \mathit{input\ size}: 13\times13,\ \widetilde{\alpha}=13$ | | | | | |
| | time | Tflop | time | Tflop | times |
| fprop | 84.12 | 2.66 | 43.95 | 5.10 | 1.91 × |
| bprop | 86.21 | 2.60 | 45.04 | 4.97 | 1.91 × |
| wgrad | 83.10 | 2.69 | 43.42 | 5.16 | 1.91 × |



| Layer5 | FA/ms | | tFFT/ms | | speadup |
|---|---|---|---|---|---|
| $f=384,\ f'=384,\ \mathit{filter\ size}: 3\times3,\ \mathit{input\ size}: 13\times13,\ \widetilde{\alpha}=4$ | | | | | |
| | time | Tflop | time | Tflop | times |
| fprop | 24.74 | 9.06 | 21.97 | 10.20 | 1.26 × |
| bprop | 25.35 | 8.84 | 22.52 | 9.95 | 1.26 × |
| wgrad | 24.44 | 9.18 | 21.71 | 10.33 | 1.25 × |



| Layer6 | FA/ms | | tFFT/ms | | speadup |
|---|---|---|---|---|---|
| $f=256,\ f'=256,\ \mathit{filter\ size}: 3\times3,\ \mathit{input\ size}: 13\times13,\ \widetilde{\alpha}=4$ | | | | | |
| | time | Tflop | time | Tflop | times |
| fprop | 21.14 | 10.04 | 19.93 | 10.65 | 1.06 × |
| bprop | 21.67 | 9.79 | 20.42 | 10.39 | 1.06 × |
| wgrad | 20.89 | 10.16 | 19.69 | 10.78 | 1.06 × |



| Layer7 | FA/ms | | tFFT/ms | | speadup |
|---|---|---|---|---|---|
| $f=256,\ f'=4096,\ \mathit{filter\ size}: 6\times6,\ \mathit{input\ size}: 6\times6,\ \widetilde{\alpha}=4$ | | | | | |
| | time | Tflop | time | Tflop | times |
| fprop | 53.17 | 7.09 | 31.89 | 11.83 | 1.67 × |
| bprop | 54.50 | 6.92 | 32.68 | 11.54 | 1.67 × |
| wgrad | 52.53 | 7.18 | 31.50 | 11.98 | 1.67 × |

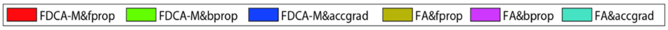


**Fig. 5** AlexNet throughput (TFLOPS) vs. Batch size for tFFT and FA on NVIDIA GTX 1080i GPU with peak throughput of 5.67 Tflop

**Table 2** Segmentation accuracy of several state-of-art networks

| | Input-CNN | AP | $\text{AP}^2$ | $\text{AP}^4$ | $\text{AP}^\text{b}$ | $\text{AP}^\text{r}$ | $\text{AP}^\text{m}$ |
|---|---|---|---|---|---|---|---|
| PointNet | Point cloud + PointNet | 24.24 | 28.5 | 25.5 | – | 39.8 | – |
| cfDSS | cuFFT + DSS | 22.5 | 26.4 | 23.4 | 29.3 | 28.5 | 35.5 |
| fbDSS | fbFFT + DSS | 22.1 | 23.2 | 25.5 | 27.8 | 30.5 | 38.8 |
| tfCNN | tFFT + DSS + AlexNet | 23.8 | 20.5 | 19.8 | 28.5 | 29.5 | 29.8 |
| tfCNN | tFFT + DSS + VGGNet-19 | 25.5 | 25.2 | 25.9 | 29.2 | 30.8 | 30.4 |
| tfCNN | tFFT + DSS + ResNet-34 | **26.5** | 27.1 | 26.1 | 30.5 | **40.2** | 39.6 |

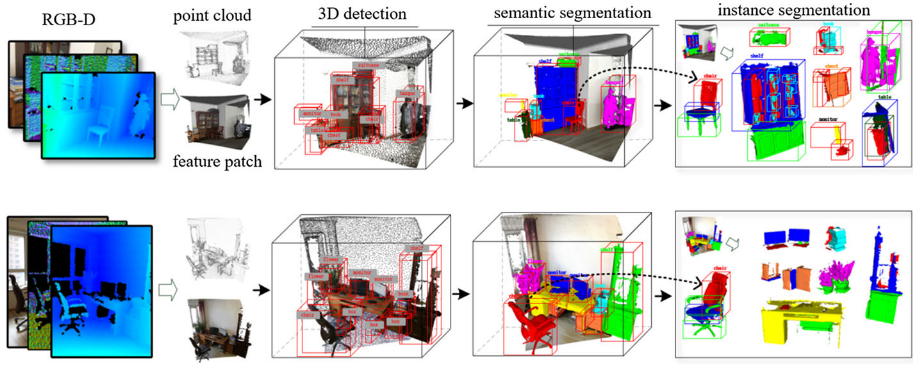


**Fig. 6** Segmentation results of tfCNN on 3D amodal scenes with 26.5 AP

## 5.4 Precision Analysis for tFFT

We evaluate the accuracy of tFFT by evaluating the segmentation performance of tFFT-based network (tfCNN). We employ DSS [28] as the baseline framework of tfCNN. We integrate three FCNs respectively into tfCNN as 2D detectors to extract color features. Three FCNs are AlexNet, ResNet-16 and VGG-net. In Table 2, tfCNN is compared to the state-of-the-art networks in 3D instance segmentation. tfCNN performs on par to other state-of-the-art methods, they are PointNet [29], cuFFT + DSS (called for short: cfDSS) and fbFFT + DSS (called for short: fbDSS). PointNet achieves 3D detection and segmentation without convolutions. cfDSS employs CUDA implementation of cuFFT algorithm to perform convolutions in each layer of DSS. fbDSS uses deep sliding windows to perform amodal 3D detection and segmentation. The superscripts of AP represent object detection and segmentation in different scales. tfCNN outperforms PointNet by 2.26 in AP and only 0.4 in $\text{AP}^\text{r}$, this indicates that using point cloud as input improves segmentation precision in larger invisible regions. fbCNN uses 3D volumes as single input without color information reduces segmentation precision by 1.7 AP, this indicates that color input is important for convnet layers which allows for accurate extraction of features from 3D amodal proposals. tfCNN guarantees accurate segmentation of amodal instances in 26.5 AP and 40.2 $\text{AP}^\text{r}$, this little higher points indicates the effectiveness of tfCNN for predicting invisible part of 3D amodal object.

The segmentation results of tfCNN are shown in Fig. 6. tfCNN achieves robust segmentation of 3D instances even in overlapping regions. This indicates that tFFT is compatible with the state of art networks, tFFT can be integrated into CNNs intuitively to accelerate convolutions.

## 6 Conclusion

A tile-based Fast Fourier Transforms algorithm is proposed for modern CNNs to accelerate convolution operations in each convnet layer. tFFT decomposes the dimension of convolutions by implementing tile sized transformations in Fourier domain. tFFT reduces the average arithmetic complexity by over 2.64 at batch sizes from 1 to 128 compared to the state-of-the-art algorithms. tFFT reduces the training and inference time for the state-of-the-art CNNs including AlexNet, VGGNet, and ResNet. tFFT accelerates training of CNNs by improving the baseline frameworks of convolutions, it can be integrated into modern CNNs for accelerating convolutions.

**Acknowledgements** This work was supported by National Natural Science Foundation of China (Grant No. 51705032), National High-tech R&D Program (Grant No. 2014AA7031010B) and Science-Technology Project of the thirteenth Five-Year Plan (Grant No. 2016345).

## References

1. LeCun Y, Bottou L, Orr GB, Müller K-R (2002) Efficient backprop. Neural Netw Tricks Trade 1524:9–50
2. Bahrampour S, Ramakrishnan N, Schott L, Shah M (2016) Comparative study of caffe, neon, theano, and torch for deep learning. Comput Sci. https://arxiv.org/abs/1511.06435
3. Shen B, Wang Z, Qiao H (2017) Event-triggered state estimation for discrete-time multidelayed neural networks with stochastic parameters and incomplete measurements. IEEE Trans Neural Netw Learn Syst 28(5):1152–1163
4. Liu H, Wang Z, Shen B, Liu X (2017) Event-triggered h∞ state estimation for delayed stochastic memristive neural networks with missing measurements: the discrete time case. IEEE Trans Neural Netw Learn Syst 29(8):3726–3737
5. Heideman MT, Johnson DH, Burrus CS (1985) Gauss and the history of the fast fourier transform. Arch Hist Exact Sci 34(3):265–277
6. Lee K, Park DC (2017) Fast training of convolutional neural networks and its application to image classification. Inf Int Interdiscip J. https://arxiv.org/abs/1312.5851
7. Vasilache N, Johnson J, Mathieu M, Chintala S, Piantino S, Lecun Y (2014) Fast convolutional nets with fbfft: a gpu performance evaluation. https://arxiv.org/abs/1412.7580
8. Liu W, Wang Z, Liu X, Zeng N, Liu Y, Alsaadi FE (2017) A survey of deep neural network architectures and their applications. Neurocomputing 234:11–26
9. Chetlur S, Woolley C, Vandermersch P et al. (2014) cudnn: efficient primitives for deep learning. https://arxiv.org/abs/1410.0759
10. Nickolls J (2008) Parallel computing experiences with cuda. Micro IEEE 28(4):13–27
11. Rockmore DN (2000) The fft: an algorithm the whole family can use. Comput Sci Eng 2(1):60–64
12. Cooley JW, Tukey JW (1965) An algorithm for the machine calculation of complex fourier series. Math Comput 19(90):297–301
13. Highlander T, Rodriguez A (2016) Very efficient training of convolutional neural networks using fast fourier transform and overlap-and-add. https://arxiv.org/abs/1601.06815
14. Simonyan K, Zisserman A (2014) Very deep convolutional networks for large-scale image recognition. https://arxiv.org/abs/1409.1556
15. Szegedy C, Liu W, Jia Y, Sermanet P, Reed S, Anguelov D et al (2015) Going deeper with convolutions. In: 2015 IEEE conference on computer vision and pattern recognition (CVPR), pp 1–9. https://arxiv.org/abs/1409.4842
16. He K, Zhang X, Ren S, Sun J (2015) Deep residual learning for image recognition. https://doi.org/10.1109/CVPR.2016.90
17. Lavin A, Gray S (2015) Fast algorithms for convolutional neural networks. Comput Vision Pattern Recognit 10:4013–4021. https://doi.org/10.1109/CVPR.2016.435

18. Winograd S (1980) Arithmetic complexity of computations, vol 43(2). Society for Industrial & Applied Mathematics, Philadelphia, pp 625–633
19. Coppersmith D, Winograd S (1987) Matrix multiplication via arithmetic progressions. J Symb Comput 9(3):251–280
20. Gentleman WM, Sande G (1966) Fast fourier transforms—for fun and profit. November, Fall Joint Computer Conference. AFIPS '66 (Fall) Proceedings of the November 7–10, 1966, fall joint computer conference, pp 563–578
21. Schatzman JC (1996) Accuracy of the discrete fourier transform and the fast fourier transform. SIAM J Sci Comput 17(5):1150–1166
22. Li S, Dou Y, Niu X, Lv Q, Wang Q (2017) A fast and memory saved gpu acceleration algorithm of convolutional neural networks for target detection. Neurocomputing 230:48–59
23. Penas M, Penedo MG, Carreira María J (2008) A neural network based framework for directional primitive extraction. Neural Process Lett 27(1):67–83
24. Takahashi D (2001) An extended split-radix fft algorithm. IEEE Signal Process Lett 8(5):145–147
25. Lavin A (2015) Maxdnn: an efficient convolution kernel for deep learning with maxwell gpus. Comput Sci. https://arxiv.org/abs/1501.06633
26. Le Gall F (2014) Powers of tensors and fast matrix multiplication, pp 296–303. https://arxiv.org/abs/1401.7714v1
27. Krizhevsky A, Sutskever I, Hinton GE (2012) ImageNet classification with deep convolutional neural networks. Int Conf Neural Inf Process Syst 1:1097–1105
28. Song S, Xiao J (2015) Deep sliding shapes for amodal 3d object detection in rgb-d images. In: IEEE conference on computer vision and pattern recognition (CVPR). https://arxiv.org/abs/1511.02300v1
29. Charles RQ, Su H, Kaichun M, Guibas LJ (2017) PointNet: deep learning on point sets for 3D classification and segmentation. In: 2017 IEEE conference on computer vision and pattern recognition (CVPR), pp 201–210. https://doi.org/10.1109/cvpr.2017.16