参考：https://blog.csdn.net/qq_32998593/article/details/86177151

摘要：

首先要说明的是，卷积是指ConvNet中默认的卷积，而不是数学意义上的卷积。其实，ConvNet 中的卷积对于与数学中的 cross correlation。计算卷积的方法有很多种，常见的有以下几种方法：

滑窗：这种方法是最直观最简单的方法。 但是，该方法不容易实现大规模加速，因此，通常情况下不采用这种方法（但是也不是绝对不会用，在一些特定的条件下该方法反而是最高效的）。

im2col：目前几乎所有的主流计算框架包括 Caffe, MXNet 等都实现了该方法. 该方法把整个卷积过程转化成了GEMM过程，而GEMM在各种 BLAS 库中都是被极致优化的，一般来说，速度较快。

FFT：傅里叶变换和快速傅里叶变化是在经典图像处理里面经常使用的计算方法，但是，在 ConvNet中通常不采用，主要是因为在 ConvNet 中的卷积模板通常都比较小，例如 3×3 等，这种情况下，FFT 的时间开销反而更大，所以很少在CNN中利用FFT实现卷积。

Winograd： Winograd 是存在已久最近被重新发现的方法，在大部分场景中, Winograd方法都显示和较大的优势，目前cudnn中计算卷积就使用了该方法。

高效卷积算法之一：

1. BLAS简介

BLAS(Basic Linear Algebra Subprograms)是一组线性代数计算中通用的基本运算操作函数集合[1] 。BLAS库中函数根据运算对象的不同，BLAS库中函数根据运算对象的不同，分为3个level。

Level 1 函数处理单一向量的线性运算以及两个向量的二元运算。Level 1 函数最初出现在1979年公布的BLAS库中。

Level 2 函数处理 矩阵与向量的运算，同时也包含线性方程求解计算。 Level 2 函数公布于1988年。

Level 3 函数包含矩阵与矩阵运算。Level 3 函数发表于1990年。

PGEMM是Level 3中的库，其中函数的语法和描述为：

- Description: matrix matrix multiply

- Syntax: PGEMM(TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC)
- P: S(single float), D(double float), C(complex), Z(complex*16)

详情可以去BLAS的官方文档查询这个卷积实现算法。

向量和矩阵运算是数值计算的基础，BLAS库通常是一个软件计算效率的决定性因素。除了BLAS参考库以外，还有多种衍生版本和优化版本。这些BLAS库实现中，有些仅实现了其它编程语言的BLAS库接口，有些是基于BLAS参考库的Fortran语言代码翻译成其它编程语言，有些是通过二进制文件代码转化方法将BLAS参考库转换成其它变成语言代码，有些是在BLAS参考库的基础上，针对不同硬件(如CPU，GPU)架构特点做进一步优化。

几个比较著名的基于BLAS库开发的高级库包括：

Intel® Math Kernel Library

Intel® Math Kernel Library (Intel® MKL) accelerates math processing and neural network routines that increase application performance and reduce development time. Intel MKL includes highly vectorized and threaded Linear Algebra, Fast Fourier Transforms (FFT), Neural Network, Vector Math and Statistics functions. The easiest way to take advantage of all of that processing power is to use a carefully optimized math library. Even the best compiler can’ t compete with the level of performance possible from a hand-optimized library. If your application already relies on the BLAS or LAPACK functionality, simply re-link with Intel MKL to get better performance on Intel and compatible architectures.

cuBLAS

The NVIDIA CUDA Basic Linear Algebra Subroutines (cuBLAS) library is a GPU-accelerated version of the complete standard BLAS library that delivers 6x to 17x faster performance than the latest MKL BLAS.

clBLAS

This repository houses the code for the OpenCL™ BLAS portion of clMath. The complete set of BLAS level 1, 2 & 3 routines is implemented.

2. GEMM分析

GEMM在深度学习中是十分重要的，全连接层以及卷积层基本上都是通过GEMM来实现的，而网络中大约90%的运算都是在这两层中。而一个良好的GEMM的实现可以充分利用系统的多级存储结构和程序执行的局部性来充分加速运算。

Matrix-Matrix multiplication中，BLAS的两个矩阵相乘的标准接口是：

```
dgemm( transa, transb, m, n, k, alpha, A, lda, B, ldb, beta, C, ldc)
```

这个接口的主要功能为计算：

$$C := \alpha AB + \beta C$$

而transa和transb可以分别指定矩阵A和矩阵B是否进行转置；A的计算维度为m×k，矩阵B的计算维度为k×n，矩阵C的计算维度为m×n；ldX表示矩阵X的行数（这个的参数，可能有的同学会说，不是跟m或者n或者k一样吗？其实不然，这个参数的功能是让这个接口能够使得A本身比计算维度大，只使用子矩阵来进行计算，m≤lda）。接下来开始对这个接口进行优化。

GEMM算法基础：

$$A = \begin{pmatrix} a_{0,0} & \cdots & a_{0,k-1} \\ \vdots & & \vdots \\ a_{m-1,0} & \cdots & a_{m-1,k-1} \end{pmatrix}, B = \begin{pmatrix} b_{0,0} & \cdots & b_{0,n-1} \\ \vdots & & \vdots \\ b_{k-1,0} & \cdots & b_{k-1,n-1} \end{pmatrix}, \text{ and } C = \begin{pmatrix} c_{0,0} & \cdots & c_{0,n-1} \\ \vdots & & \vdots \\ c_{m-1,0} & \cdots & c_{m-1,n-1} \end{pmatrix}$$

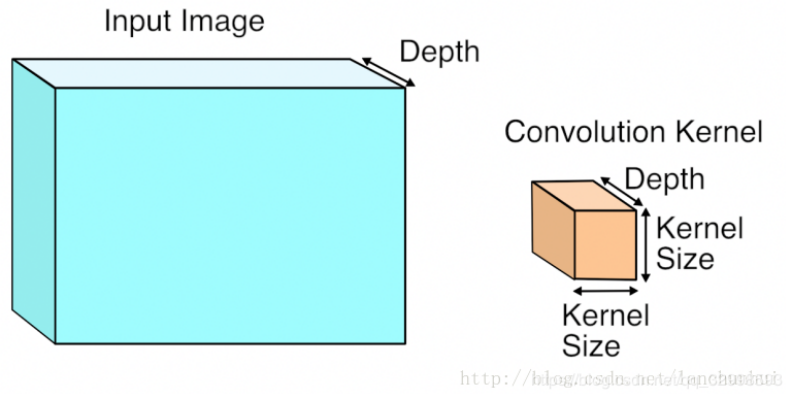
则根据矩阵相乘的定义，可得C=AB+C的计算公式为：

$$c_{i,j} = \sum_{p=0}^{k-1} a_{i,p} b_{p,j} + c_{i,j},$$

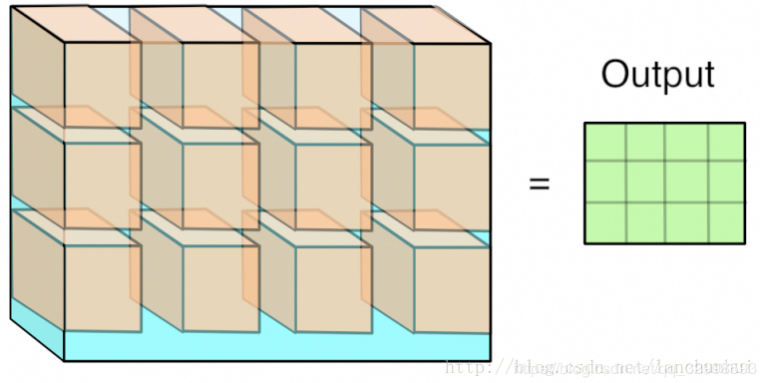
```
for i=0:m-1
 for j=0:n-1
 for p=0:k-1
 C(i,j):=A(i,p)*B(p,j)+C(i,j)
 endfor
 endfor
endfor
```

可以看出，根据定义实现的算法计算复杂度为O(mnk)，也就是O(n^3)。

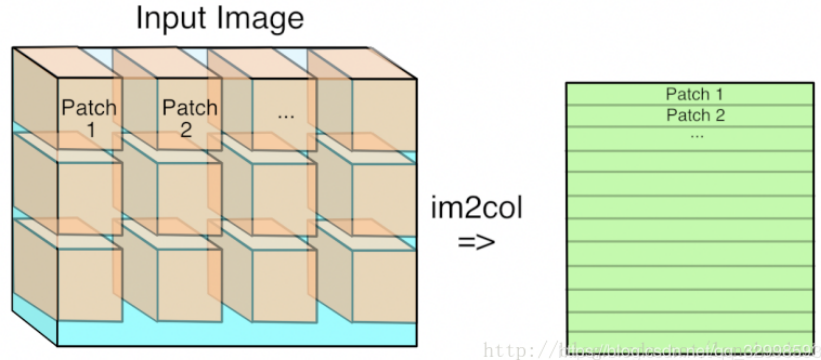
常规的卷积操作为：

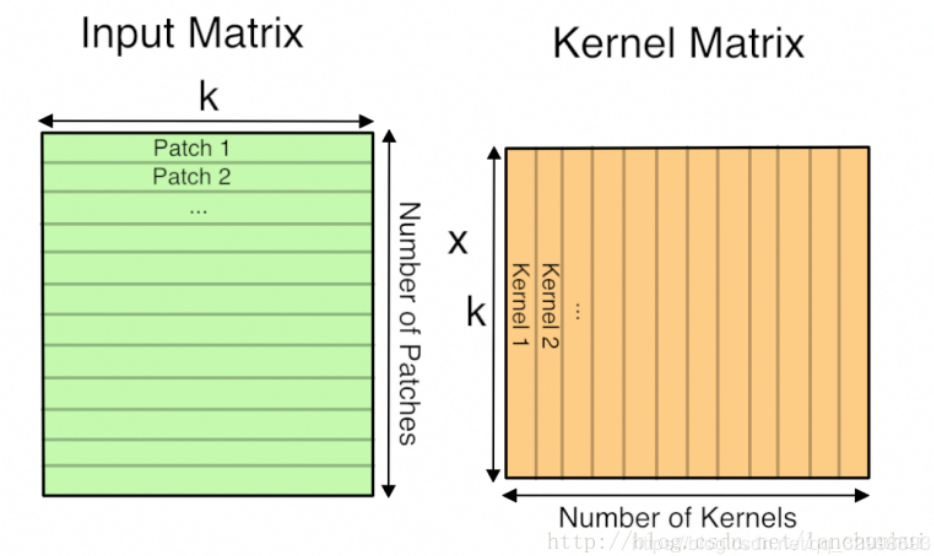


3维卷积运算执行完毕，得一个2维的平面：



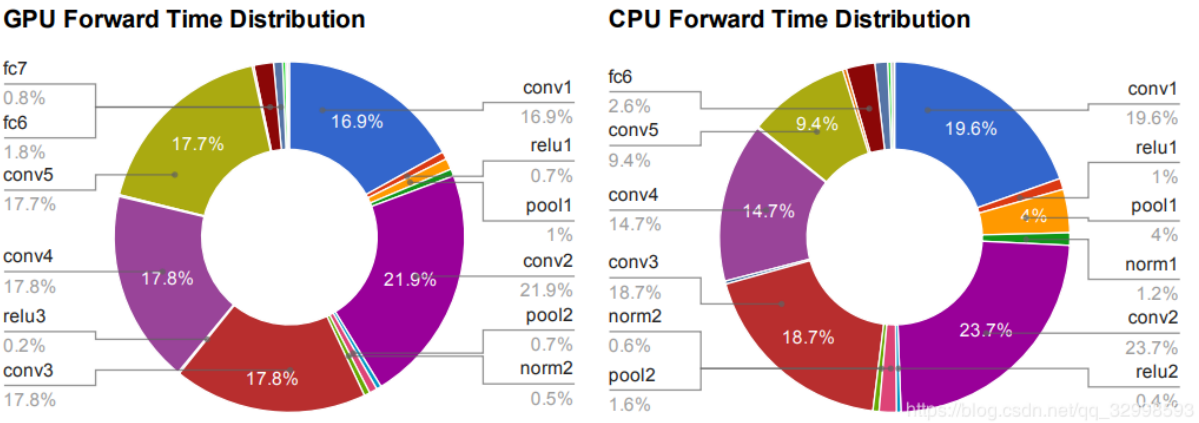
将卷积操作的3维立体变为二维矩阵乘法，可以调用BLAS中的GEMM库，按[kernel_height, kernel_width, kernel_depth] ⇒ 将输入分成 3 维的 patch，并将其展成一维向量：





由于内存中二维数组是以行优先进行存储的，因此B[k][j]存在严重的cache命中率问题，解决这个问题的方法是也将B进行一次沿对角线进行翻转，使得最里面的计算变成row直接取出来。

从下图可以看出，一个CNN模型运算过程中，基本上的时间都来消耗在CONV操作上。（图像来自于贾扬清毕业论文，论文地址）



高效卷积算法之二：

Winograd算法简介

这个算法用更多的加法来代替乘法，在FPGA上，由于乘法的计算消耗更多的资源和时钟，往往是很慢的。一般的乘法需要借助FPGA中的DSP来计算，DSP的大小有18 bits X 25 bits的乘法，如果两个浮点数较大，则需要更多的乘法。采用Winograd卷积算法，借助LUT来计算乘法，会节省FPGA的资源，加快速度。

3.Winograd Algorithms

对于一维卷积，当输出为m，卷积核长度为r时，需要乘法数量为：

$$\mu\left(F\left(m,r\right)\right)=m+r-1$$

将一维卷积扩展到二维，如果输出维mxn，卷积核维rxs，需要乘法数量为：

$$\mu\left(F\left(m\times n,r\times s\right)\right)=\mu\left(F\left(m,r\right)\right)\mu\left(F\left(n,s\right)\right)=\left(m+r-1\right)\left(n+s-1\right)$$

3.1 F(2X2,3X3)

直接计算一维卷积**F(2X3)**需要2x3=6次乘法。

输入：[d0, d1, d2, d3], 卷积核为 [g0,g1,g2]，没有padding，步长为1，写成矩阵形式为：

$$F\left(2,3\right)=\begin{bmatrix}d_0 & d_1 & d_2\\ d_1 & d_2 & d_3\end{bmatrix}\begin{bmatrix}g_0\\ g_1\\ g_2\end{bmatrix}=\begin{bmatrix}m_1+m_2+m_3\\ m_2-m_3-m_4\end{bmatrix}$$

即：

$$m_1=\left(d_0-d_1\right)g_0$$

$$m_2=\left(d_1+d_2\right)\left(g_0+g_1+g_2\right)\frac{1}{2}$$

$$m_3=\left(d_2-d_1\right)\left(g_0-g_1+g_2\right)\frac{1}{2}$$

$$m_4=\left(d_1-d_3\right)g_2$$

这种计算方式需要2+3-1=4次乘法，4次加法，还有可以预计算的3次加法和2次乘法（卷积核默认为已知项）

将这种算法写成矩阵形式

$$Y = A^T \left[\left((Gg) \odot \left(B^T d \right) \right) \right]$$

$\odot$表示点乘，对于F(2,3)，以上矩阵分别为：

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

$$G = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 0.5 & 0.5 \\ 0.5 & -0.5 & 0.5 \\ 0 & 0 & 1 \end{bmatrix}$$

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

$$g = \begin{bmatrix} g_0 & g_1 & g_2 \end{bmatrix}^T$$

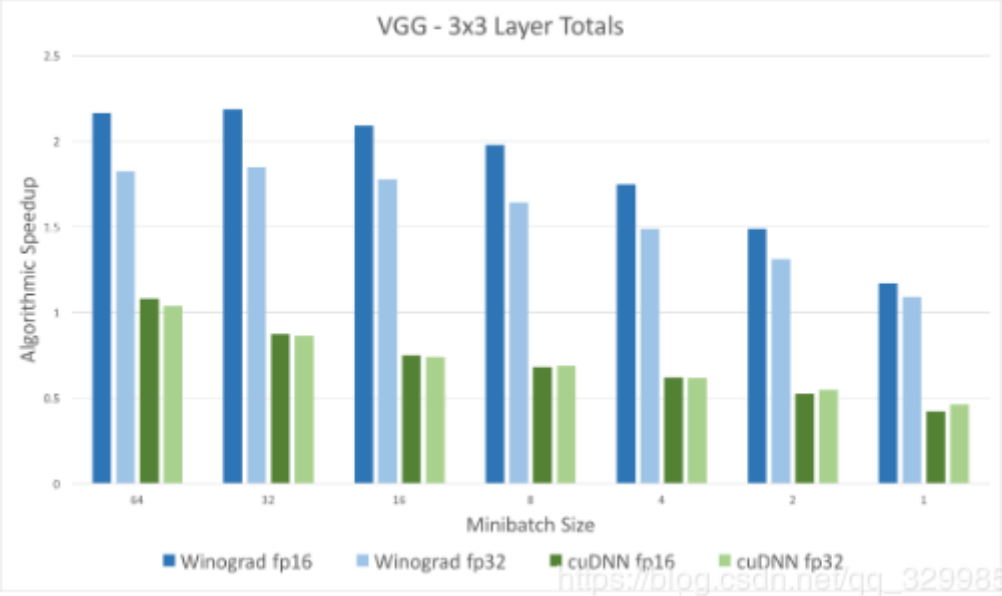
$$d = \begin{bmatrix} d_0 & d_1 & d_2 & d_3 \end{bmatrix}^T$$

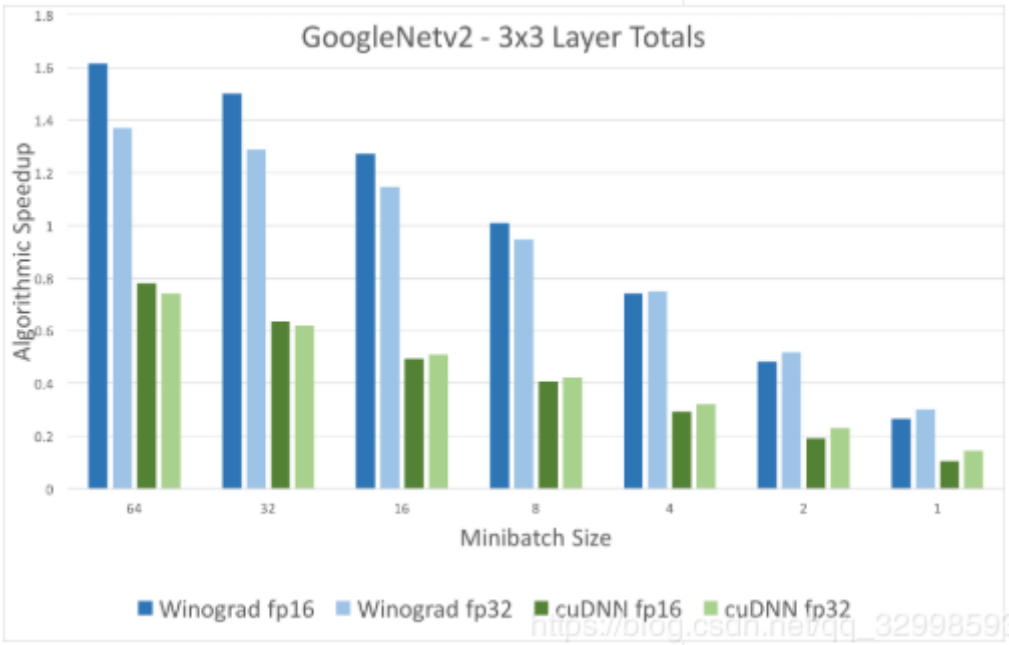
扩展为二维卷积的形式：

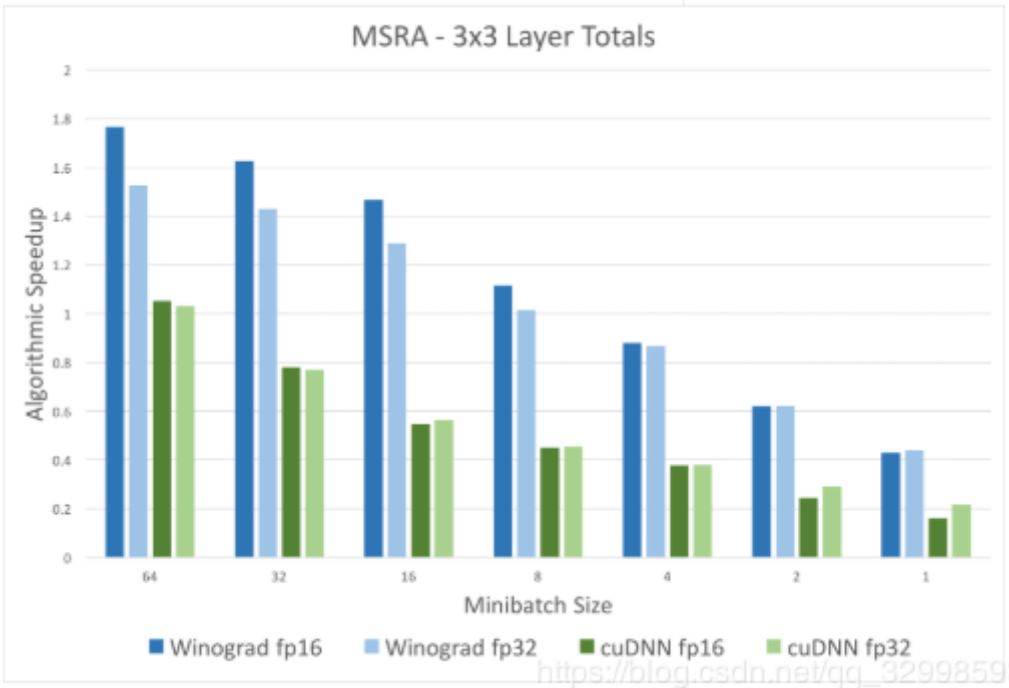
$$Y = A^T \left[\left[GgG^T \right] \odot \left[B^T dB \right] \right] A$$

因此，我们可以在filter 维度较小情况下应用winograd 做卷积。

以下是一些实际运算中，采用该方法的性能对比结果：

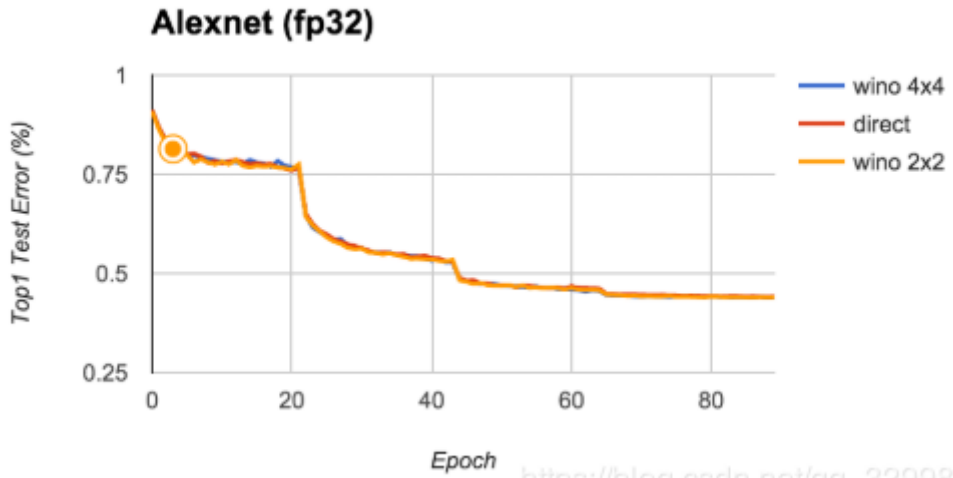






从上面的对比可以看出，利用Winograd算法进行卷积，比NVIDIA的深度学习库cuDNN中的卷积计算快很多。其中cuDNN的卷积是GEMM算法实现。batch_size越大，加速效果越明显，因为越大的batch_size，计算的负载并不是线性的增加，开辟的内存地址和GPU的显存被充分计算应用，增加的一部分空间，平均到一个样本数上，计算时间更短，速度更明显。这个理论在Google的一篇论文上也具体介绍了batch_size的影响。（图片来自于Intel官方提供的对比，对比图像地址）

反过来看精度，Winograd算法的实现和直接卷积（原始的卷积实现）的效果差不多4X4，2X2的Winograd卷积核效果都不怎么掉。说明Winograd算法是一种高效的卷积算法。有值得发掘的价值。



3.论文分析

1. Fast Algorithms for Convolutional Neural Networks

分析的第一篇文章是16年CVPR，由于是CVPR中第一次将Winograd算法引入，大篇幅介绍了Winograd算法的原理，矩阵的操作，转化，没有做任何优化，直接按照矩阵公式操作的。其中与FFT进行了比较。现在看来，在追求快速和高的FLOPS指标下，完成相同任务的MACC操作数越少，则越高效。这个在CNN的网络Pruning中是一个重要指标，相同时间，增加FLOPS，网络Parameter Size为目的。FFT并不占优势。作者比较了cuDNN和Winograd算法的TFLOPS的比较。

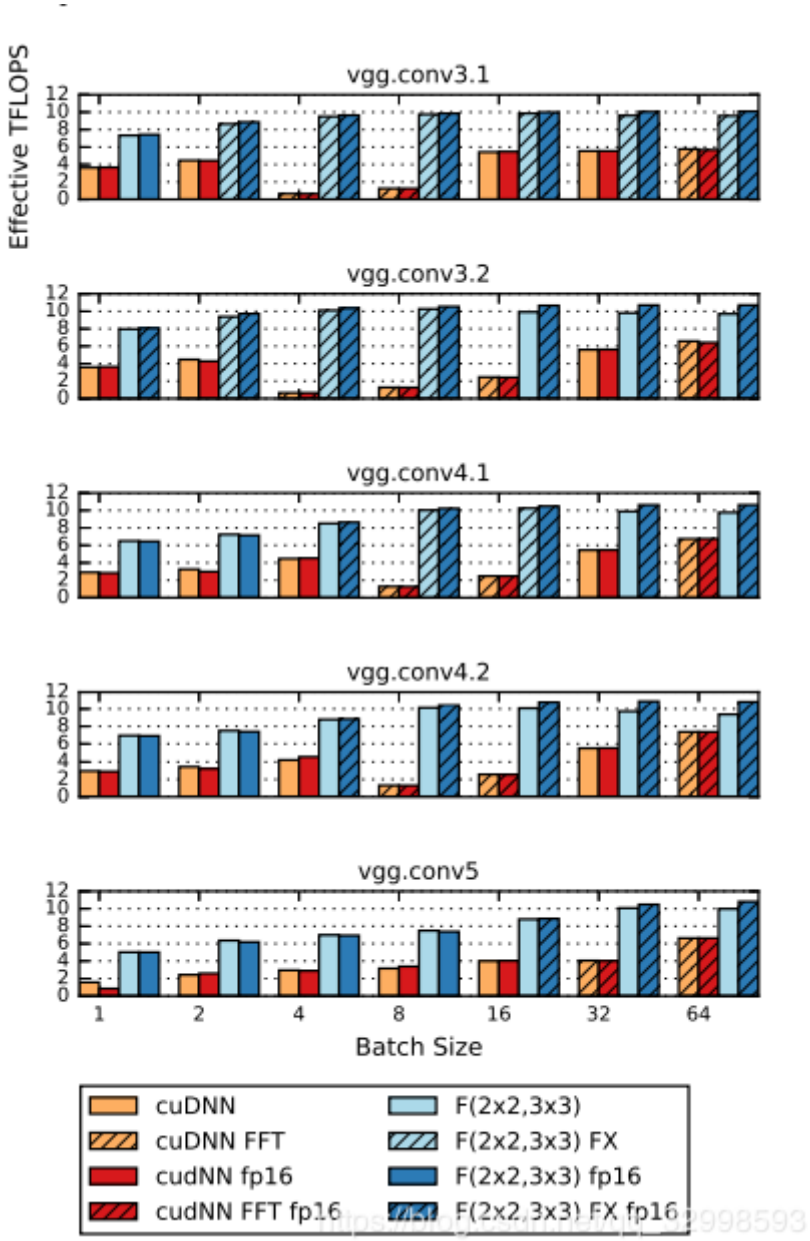
N	cuDNN		F(2x2,3x3)		Speedup
	msec	TFLOPS	msec	TFLOPS	
1	12.52	3.12	5.55	7.03	2.26X
2	20.36	3.83	9.89	7.89	2.06X
4	104.70	1.49	17.72	8.81	5.91X
8	241.21	1.29	33.11	9.43	7.28X
16	203.09	3.07	65.79	9.49	3.09X
32	237.05	5.27	132.36	9.43	1.79X
64	394.05	6.34	266.48	9.37	1.48X

Table 5. cuDNN versus $F(2 \times 2, 3 \times 3)$ performance on VGG Network E with fp32 data. Throughput is measured in Effective TFLOPS, the ratio of direct algorithm GFLOPs to run time.

N	cuDNN		F(2x2,3x3)		Speedup
	msec	TFLOPS	msec	TFLOPS	
1	14.58	2.68	5.53	7.06	2.64X
2	20.94	3.73	9.83	7.94	2.13X
4	104.19	1.50	17.50	8.92	5.95X
8	241.87	1.29	32.61	9.57	7.42X
16	204.01	3.06	62.93	9.92	3.24X
32	236.13	5.29	123.12	10.14	1.92X
64	395.93	6.31	242.98	10.28	1.63X

Table 6. cuDNN versus $F(2 \times 2, 3 \times 3)$ performance on VGG Network E with fp16 data.

从上表可以看出，Winograd算法的TFLOPS比cuDNN的计算性能高，相同的网络，处理时间更低。



从上图可以看出，Winograd算法比cuDNN的GEMM算法拥有更高的计算能力，更低的操作数。

2. Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs

下面是商汤中Winograd算法FPGA实现，17年发表在FCCM中的论文。

用更多的加法来替代乘法，传统的卷积实现，需要6层循环，而采用Winograd算法，只需要4层循环。

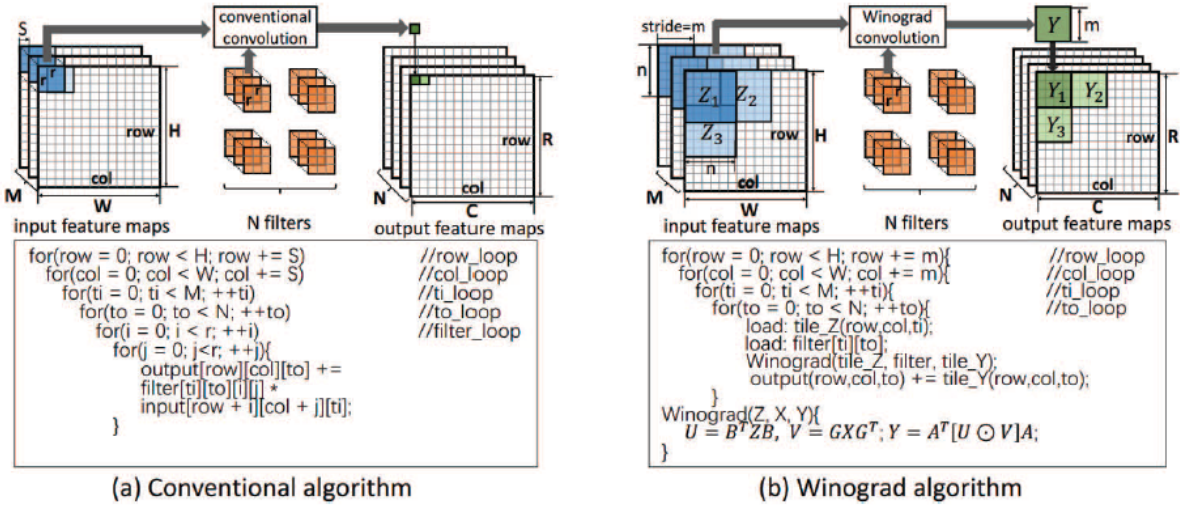


Figure 1: Comparison of conventional and Winograd convolution algorithms. We assume the stride S is 1 for Winograd algorithm.

采用原始的卷积操作，需要6层for循环运算。而采用Winograd算法，在里面的两层，只需要调用Winograd(X,F,Y)这个计算模块，就能直接计算出卷积参数。少了60%以上的乘法操作。在FPGA实现中，首先就要避免直接计算乘法，尽量用移位来代替。在乘法资源如此昂贵的FPGA上，这个Winograd算法能够减少乘法操作，当然很受业界欢迎。带来了一大部分科研人员致力于Winograd算法在FPGA的高效实现。

作者利用line buffer架构，将输入的数据进行缓存，同时将每一组kernel从cache上取出来，放进设计好的PE模块中进行Winograd矩阵乘法。

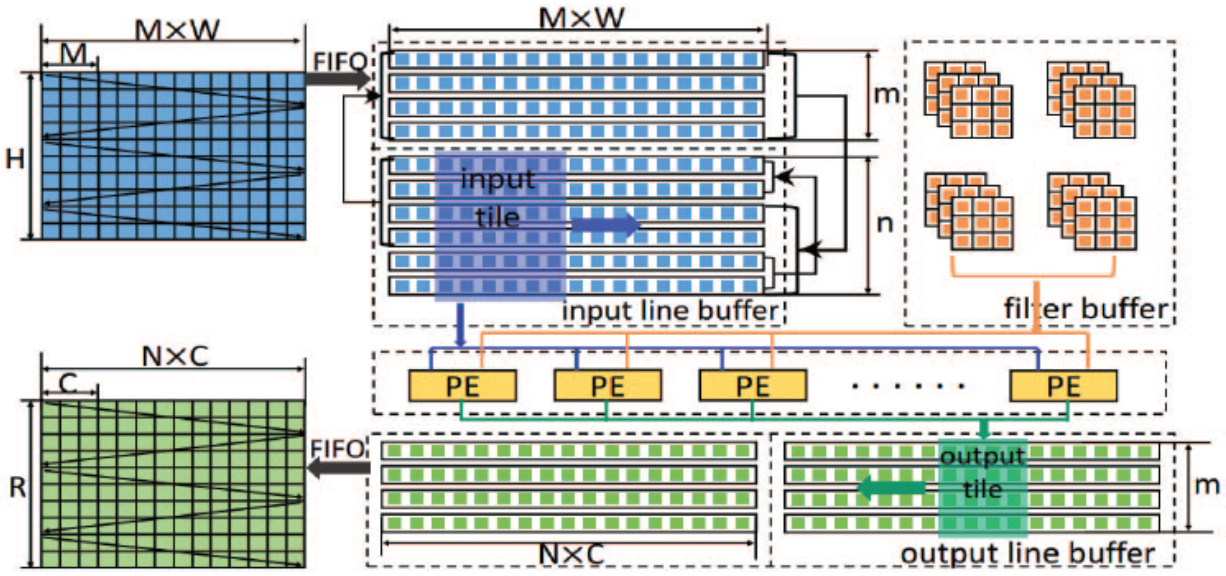


Figure 2: Architecture overview

这里具体的目的就是按模块划分好各个功能，然后利用片上的line buffer，也就是BlockRAM来进行缓存，每6行就拿去做Winograd卷积，然后Stride为4。PE为一个专门的Winograd卷积模块。如下图所示为具体的操作：

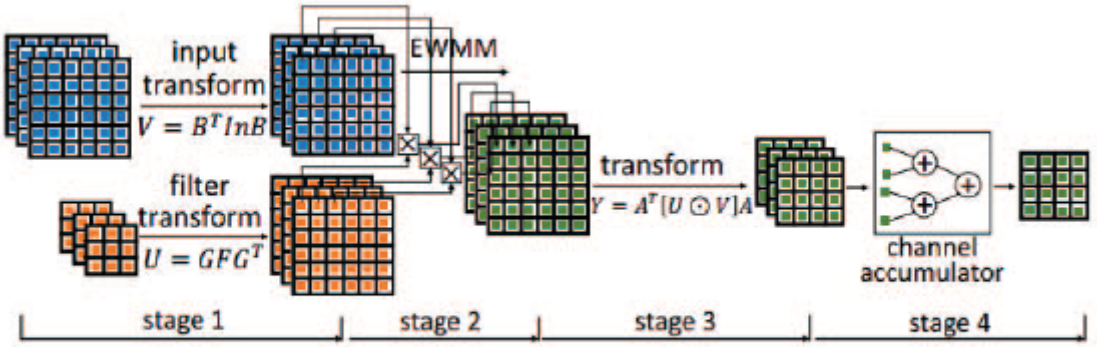


Figure 3: Winograd PE design

每一个矩阵的转换需要的矩阵是固定的(A,B,G)是离线的，将矩阵的乘法分为4步，为了节省片上的BRAM，先对input和filter进行转换，这里面是采用的LUT进行，由于转换矩阵B和G的特点，只需要进行位运算。第二步，进行EWMM计算，第三步，计算Y的转换矩阵，最后，按通道累加得到对应的output。

3. Efficient Sparse-Winograd Convolutional Neural Networks

这是MIT韩松指导的论文，发表在ICLR 2018年会议上，延续了他的风格，做剪枝，网络的稀疏化，减少了计算过程中的乘法次数。将稀疏性引入卷积转换上。

将剪枝后的稀疏性用在Winograd卷积实现上，充分减少乘法的次数，将近10.4x，6.8x和10.8x的减少。如果在传统算法上进行ReLU和Prune，在未变换之前，Activation Layer和Kernel Layer是稀疏的，当进行变换后，在做EWMM时，一样的是非稀疏的乘法，没减少乘法次数。而作者将变化在ReLU和Prune之前，使得经过稀疏操作的矩阵就是能够直接进行Winograd算法的矩阵，减少了乘法次数。

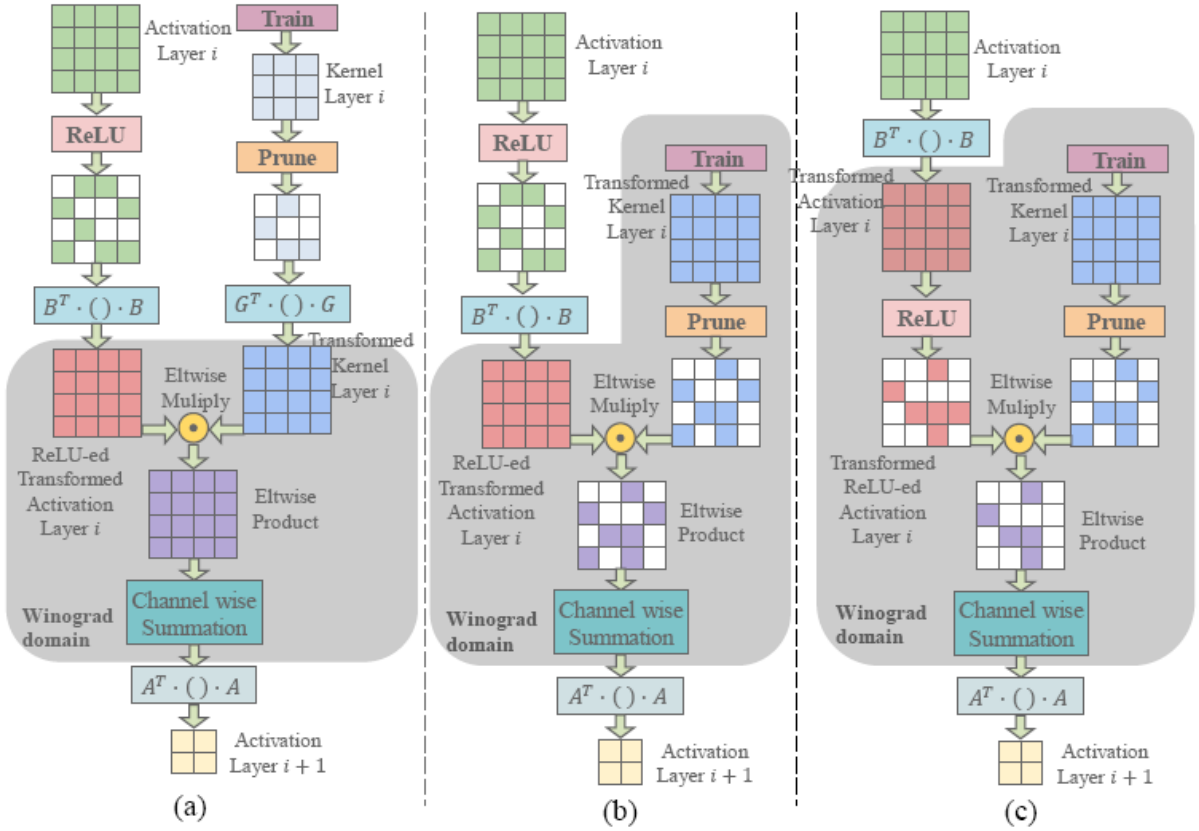


Figure 1: Combining Winograd convolution with sparse weights and activations. (a) Conventional Winograd-based convolution fills in the zeros in both the weights and activations. (b) Pruning the 4×4 transformed kernel restores sparsity to the weights. (c) Our proposed Winograd-ReLU CNN. Moving the ReLU layer after Winograd transformation also restores sparsity to the activations.

具体的内容可以在论文中去查看。这个图解释的非常清楚，相同的稀疏方法加在不同的地方，对结果的不同影响。

其中做稀疏化的公式为：

$$S = A^T[[\text{Prune}(GgG^T)] \odot [\text{ReLU}(B^T dB)]]A$$

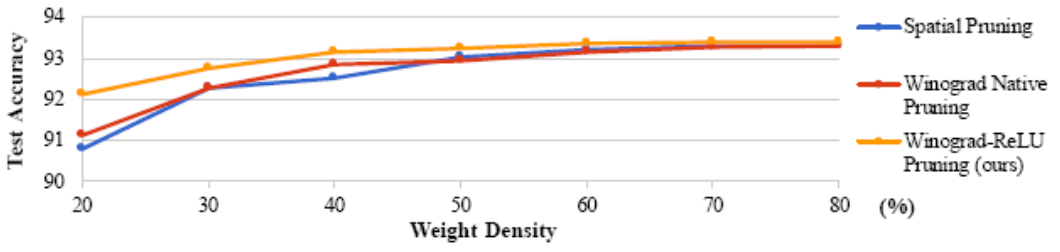


Figure 2: Test accuracy vs density for the three models in Figure 1 on VGG-nagadomi.

最终VGG在CIFAR10上的精度比较，可以看出在大密度剪枝的情况下，最终的分类精度没有掉太多。且带来的13.3x的计算负载降低。减少了权值的数量，减少了乘法次数。

Table 1: VGG-nagadomi weight and activation density on CIFAR-10.

Layer	Spatial Baseline CNN Pruning (Han et al., 2015)			Winograd CNN Native Pruning (Li et al., 2017)			Winograd-ReLU CNN Pruning (ours)		
	Density		Workload	Density		Workload	Density		Workload
	Weight	Act		Weight	Act		Weight	Act	
conv0	80%	100%	80%	80%	100%	80%	80%	100%	**80%**
conv1	60%	50%	30%	60%	100%	27%	40%	46%	**8%**
conv2	60%	19%	12%	60%	100%	27%	40%	39%	**7%**
conv3	60%	37%	22%	60%	100%	27%	40%	40%	**7%**
conv4	60%	18%	11%	60%	100%	27%	40%	40%	**7%**
conv5	60%	26%	15%	60%	100%	27%	40%	38%	**7%**
conv6	60%	24%	14%	60%	100%	27%	40%	35%	**6%**
conv7	60%	35%	21%	60%	100%	27%	40%	36%	**6%**
conv total	-	-	20%(5.1×)	-	-	27%(3.7×)	-	-	**8%(13.3×)**
overall	-	-	21%(4.7×)	-	-	29%(3.5×)	-	-	**10%(10.4×)**

4. Towards a Uniform Template-based Architecture for Accelerating 2D and 3D CNNs on FPGA

这是一篇在FPGA 2018 顶会上发表的一篇论文，作者将Winograd算法的2D卷积扩展到3D卷积，在FPGA上实现。建立了统一的计算模板。具体示意图如图所示。

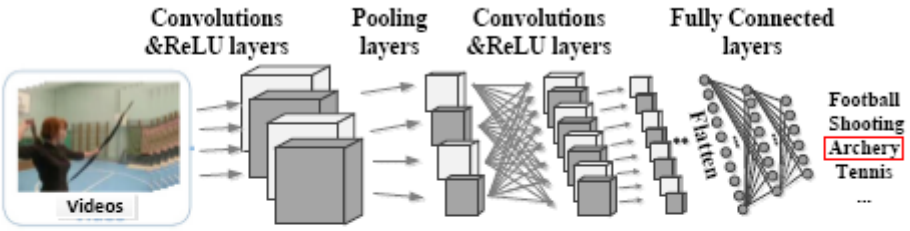


Figure 1: A real-life 3D CNN model for video classification.

Table 1: Analysis of C3D

Layers	Ops (GFLOPS)	Data Transfer(MB)		Time (ms)
		In+Out	Weights	
CONV	38.4(99.9%)	99.0(27.7%)	17.7(26.7%)	31.9(97.3%)
ReLU	0.0(0%)	96.7(27.1%)	0.0(0.0%)	0.0(2.3%)
Pool	0.0(0%)	161.0(45.1%)	0.0(0.0%)	0.0(0.2%)
FC	0.0(0.1%)	0.1(0.0%)	48.8(73.3%)	0.7(0.2%)

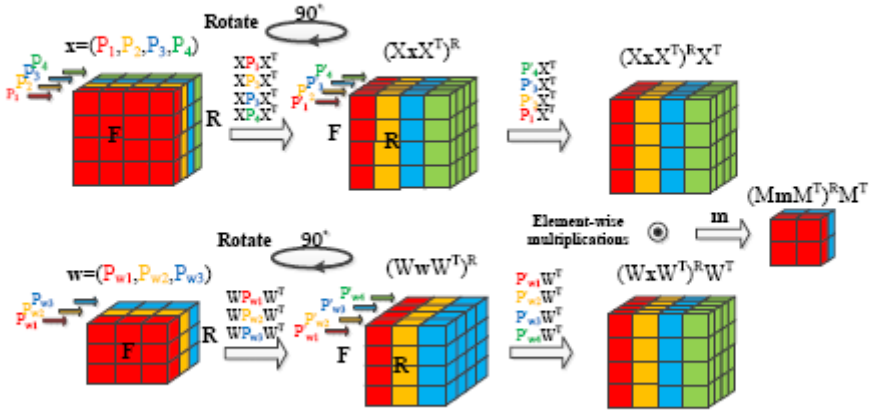


Figure 2: The process of the 3D Winograd algorithm.

将矩阵的转换，卷积的操作，全部作为整体，这样会减少更多的乘法次数，会节省FPGA上的乘法器资源。

```
L1: for(row=0;row<R; row+=2){
L2:  for(col=0;col<C; col+=2){
L3:   for(m=0; m<M; m++){
L4:    for(n=0; n<N; n++){
        Load(In, n);  //4*4 input  tile
        Load(W, m ,n); //3*3 filter tile
        2D_Trans_X(In, Tin);
        2D_Trans_W(W, Tw);
        2D_Mul(Tin,Tw, P);
        2D_Trans_M(P, Tp);
        2D_Accumulate(Sum,Tp);
    }
    Send(Out,Sum,m); //2*2 output tile
}}}
```

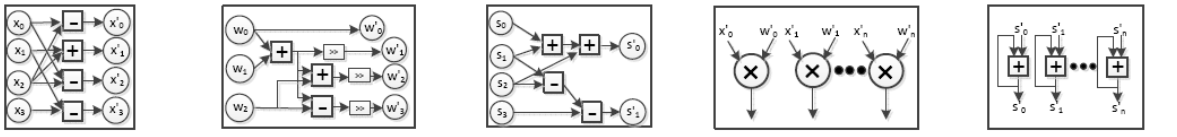
(a) CONV layers of 2D CNN

```
L1:for(dep=0;dep<Z;dep+=2){
L2: for(row=0;row<R; row+=2){
L3:  for(col=0;col<C; col+=2){
L4:   for(m=0; m<M; m++){
L5:    for(n=0; n<N; n++){
        Load(In, n);  //4*4*4 input  tile
        Load(W, m ,n); //3*3*3 filter tile
        2D_Trans_X(In, Tin);
        2D_Trans_W(W, Tw);
        Rotate(Tin, Tin^R);
        Rotate(Tw, Tw^R);
        1D_Trans_X(Tin^R, Tin^T);
        1D_Trans_W(Tw^R, Tw^T);
        3D_Mul(Tin^T,Tw^T, P);
        2D_Trans_M(P, Tp);
        Rotate(Tp, Tp^R);
        1D_Trans_M(Tp^R, Tp^T);
        3D_Accumulate(Tp^T,Sum);
    }
    Send(Out, Sum,m); //2*2*2 output tile
}}}}
```

(b) CONV layers of 3D CNN

Figure 3: Simplified pseudocode of CONV layers in 2D and 3D CNNs with Winograd algorithms.

这个卷积的实现，对比第二篇论文，即商汤在FPGA上的实现图，可以看出两篇文章的for循环的层数都是相同的，本文在3D的Winograd算法进行了优化，做了更多的处理，使得原本的2D矩阵乘法能够扩展到3D上。其实现过程更加复杂。



(a) Transform template (X) (b) Transform template (W) (c) Transform template (M) (d) Multiplication template (e) Accumulation template

Figure 4: Proposed templates for 2D and 3D CNNs ($F(2,3)$).

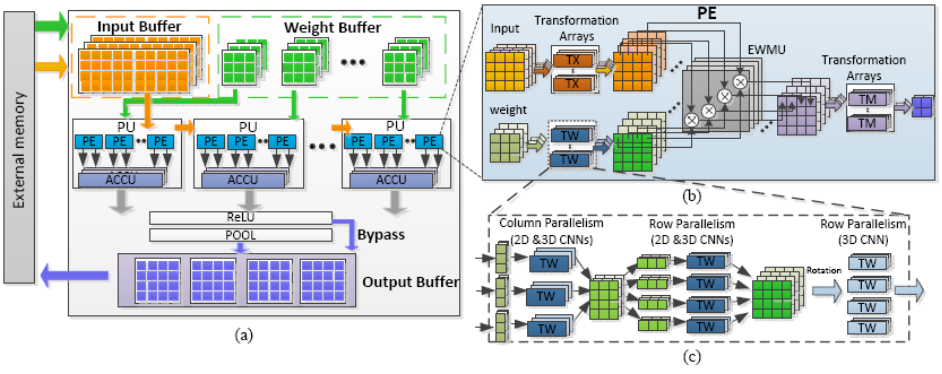


Figure 5: (a) Overview of the template-based architecture; (b) Processing element; (c) Architecture of the transformation arrays.

具体的实现过程如上图所示。将矩阵的转置，矩阵的乘法，先在模板中进行变换，最后在PE中进行Winograd最后的element-wise matrix multiplication。其实可以看出，这个模块和商汤的那篇文章很像，整个流水和处理的结构都很相似。

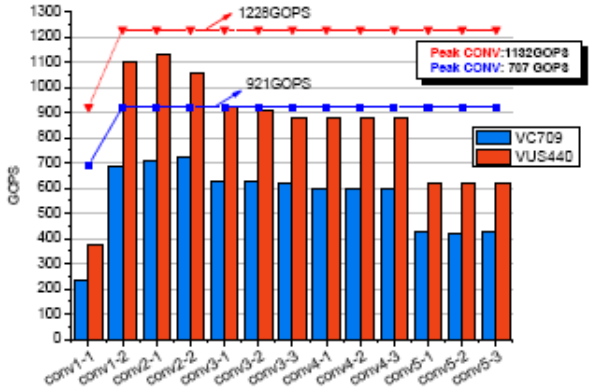


Figure 8: Evaluation results of VGG16.

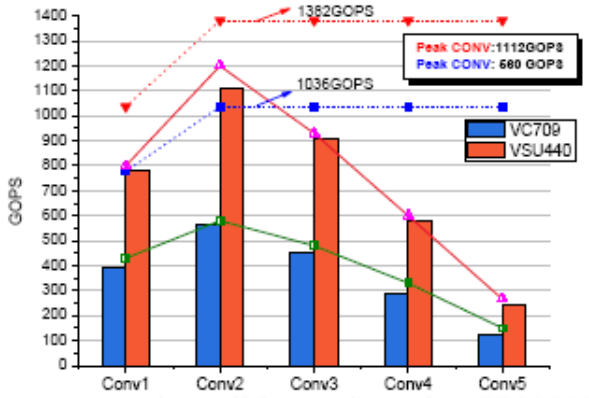


Figure 9: Evaluation results of C3D.

从最后的评估结果可以看出，性能都有很大的提升。在1.0 TOPS以上去了。更多的内容可以查看论文。