inspur 浪潮	浪潮（北京）电子信息产业有限公司
	Langchao (Beijing) Electronic Information Industry Co., Ltd

inspur 浪潮

TF2 设计说明书

文件编号：	
当前版本：	V1.0
编　　制：	AI&HPC
审　　核：	
批　　准：	
完成日期：	2019.09.12

浪潮（北京）电子信息产业有限公司

目录

1 文档介绍

1.1 文档目的

TF2 开源说明文档，帮助用户理解 TF2 开发原理、功能应用及开发方法。

1.2 读者对象

1) 客户；
2) 开发、测试、维护人员；
3) 以后版本的修改人员。

1.3 参考文档

《Inspur BSP for F10A 使用手册》。

1.4 术语与缩写解释

缩写、术语	解 释
DNN	深度神经网络
INQ	增量式网络量化方法
DNS	通道裁剪算法

2 浪潮 FPGA 计算加速引擎 TF2

2.1 概述

深度学习主要分为训练和推理两个阶段。训练阶段生成特定模型，推理阶段部署生成的模型到实际应用场景中。在实际应用场景中，如视频智能分析、机器翻译等场景，都对计算延时、功耗有较高的要求。但是对于目前广泛使用的深度学习模型，由于计算量大，常用 CPU 难以满足要求，故需要专用加速期间加速。

FPGA 是可编程的硬件逻辑电路，它可以根据具体算法制定特定电路，具有较高的性能功耗比。FPGA 可实现任意逻辑的电路，可以支持数据并行、任务并行、流水化等加速计算方式，使其可以最大化加速深度学习计算。所以 FPGA 越来越广泛应用于深度学习推理场景。

FPGA 的传统开发语言是硬件描述语言 Verilog/VHDL，其可以以文本形式来描述数字系统硬件的结构和行为的语言，用它可以表示逻辑电路图、逻辑表达式，还可以表示数字逻辑系统所完成的逻辑功能。

但是由于 Verilog/VHDL 是硬件编程语言，对于专注软件开发的软件工程师来说，学习难度很大。为了解决这个问题，FPGA 芯片生产厂商开发了基于 FPGA 的高级语言开发环境，如 OpenCL、C 语言的支持，大大降低了软件工程师的开发门槛。但是大家在开发的过程中发现，要使用 OpenCL、C 等高级语言开发出高性能的 FPGA 硬件逻辑电路仍是一件困难的事情，其仍然需要专业的数字电路架构设计经验。

基于以上原因，浪潮开发并开源了基于 FPGA 平台、可支持通用深度神经网络计算的推理引擎 TF2，可帮助大家快速搭建基于 FPGA 的深度学习应用。开源网址 https://github.com/TF2-Engine/TF2。

2.2 TF2 架构

浪潮 FPGA 计算加速引擎 TF2，可支持 Pytorch、TensorFLow、Caffe 等开源计算框架，用户经过编译技术即可适配训练完成的深度学习模型到 FPGA 上，而不需要任何的 FPGA 开发工作，故可帮助 AI 客户快速实现基于主流 AI 框架和深度神经网络模型 DNN 的 FPGA 线上推理，并通过首创的 FPGA 上 DNN 的移位运算技术获得 AI 应用的高性能和低延迟。

深度神经网络（DNN）如卷积神经网络（CNN）包含多个卷积层（Convolutional Layer）和全连接层（Fully-connected Layer），它们的主要计算是浮点数之间的乘加计算，因此 DSP 的计算能力是限制 FPGA 计算性能的关键因素。TF2 计算加速引擎的 FPGA 程序中使用移

位计算替代浮点乘法计算，使用少量的逻辑资源即可实现，完全移除了 DNN 计算对 DSP 浮点计算能力的依赖，可提高 FPGA 的计算能力，并降低计算功耗。

TF2 计算加速引擎由 Transforme Kit 和 Runtime Engine 两部分组成，如图 2-1 所示。第一部分是模型优化转换工具 TF2 Transform Kit，它具有模型压缩、模型裁剪、模型量化等功能，可最大化减小模型尺寸、减少计算量。Transform Kit 同时可以进行计算节点融合，融合多个计算节点为 1 个节点，降低数据访问带宽对计算性能的影响；第二部分是 FPGA 智能运行引擎 TF2 Runtime Engine，它可实现将 Transform Kit 优化转换的模型文件自动转化为 FPGA 目标运行文件。

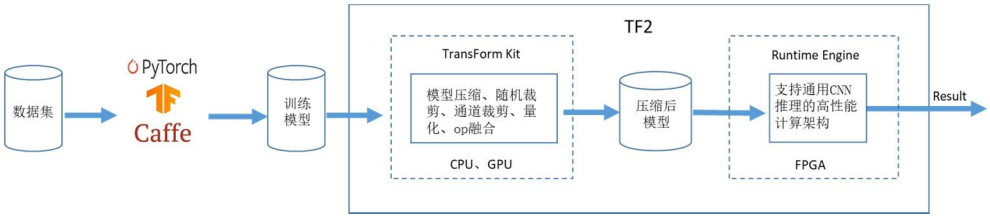

图 2-1 TF2 计算加速流程

3 TF2 模型压缩、裁剪和量化

3.1 压缩算法

3.1.1 压缩算法步骤

1) 将所有权重都表示为 2 的幂次方形式，参考 Incremental Network Quantization (INQ) 算法。
2) 将 1 中所训练的权重，使用 4bit 形式表示。

3.1.2 算法原理

⑴ INQ 算法

INQ 算法的核心思想为将所有权重量化为 2 的整次幂，算法流程图如下所示：

Input: X: the training data, $\{\boldsymbol{W}_l : 1 \le l \le L\}$: the pre-trained full-precision CNN model, $\{\sigma_1, \sigma_2, \cdots, \sigma_N\}$: the accumulated portions of weights quantized at iterative steps.

1: Initialize $\boldsymbol{A}_l^{(1)} \leftarrow \emptyset$, $\boldsymbol{A}_l^{(2)} = \boldsymbol{W}_l(i,j)$, $\boldsymbol{T}_l \leftarrow 1$, for $1 \le l \le L$

2: **for** n = 1, 2, … , N **do**

3: Reset the base learning rate and the learning policy

4: According to σ_n, perform layer-wise weight partition and update $\boldsymbol{A}_l^{(1)}$, $\boldsymbol{A}_l^{(2)}$ and $\boldsymbol{T}_l$

5: Based on $\boldsymbol{A}_l^{(1)}$, determine $\boldsymbol{P}_l$ layer-wisely

6: Quantize the weights in $\boldsymbol{A}_l^{(1)}$ by Equation (4) layer-wisely

7: Calculate feed-forward loss, and update weights in $\left\{\boldsymbol{A}_l^{(2)} : 1 \le l \le L\right\}$ by Equation (5)

8: **end for**

Output: $\{\widehat{\boldsymbol{W}}_l : 1 \le l \le L\}$: the final low-precision model with the weights constrained to be either powers of two or zero.

图 3-1 INQ 算法流程图

•量化值集合

用 $\{\boldsymbol{W}_l, 0 \le l \le L\}$ 表示预训练 CNN 模型中的第 l 层的权重参数，L 表示 CNN 模型中卷积层与全连接层的个数。权值的量化规则，要将W_l 转化为量化值$\widehat{W_l}$，即 2 的整次幂或者 0，且值从下列集合中选取：

$$\boldsymbol{P}_l = \{\pm 2^{n_1}, \cdots, \pm 2^{n_2}, 0\} \tag{1}$$

其中，n_1与n_2均为整数，且满足 $n_2 \le n_1$，这样，n_1与n_2 将 $\boldsymbol{W}_l$ 中的非零项约束在了 $[-2^{n_1}, -2^{n_2}]$ 或者 $[2^{n_2}, 2^{n_1}]$的范围，$\boldsymbol{W}_l$中绝对值小于2^{n_2}的值将被裁剪掉（数值设为 0）。INQ 算法中，低比特位量化所用的比特位个数 b 是提前设定的，因此只需计算出 n_1，则 n_2 可根据 n_1 与 b 计算得出。n_1 的计算公式如下：

$$n_1 = floor\left(log_2 \frac{4 \times max\left(abs(\boldsymbol{W}_l)\right)}{3}\right) \tag{2}$$

其中，floor(·) 函数表示向下取整，max(·) 函数表示取所有输入元素的最大值，abs(·) 函数表示对每个元素取绝对值。得到 n_1 之后，可得 $n_2 = n_1 - b$。

•权值分组

对于 CNN 网络的第 l 层，权值分组定义如下：

$$\boldsymbol{A}_l^{(1)} \cup \boldsymbol{A}_l^{(2)} = \{\boldsymbol{W}_l(i,j)\}, \qquad and\ \boldsymbol{A}_l^{(1)} \cap \boldsymbol{A}_l^{(2)} = \emptyset \tag{3}$$

其中，$\boldsymbol{A}_l^{(1)}$ 表示将要被量化的权值组，$\boldsymbol{A}_l^{(2)}$ 表示需要重训练的权值组。权值分组时，$\boldsymbol{T}_l(i,j) = 0$ 表示$\boldsymbol{W}_l(i,j) \in \boldsymbol{A}_l^{(1)}$，$\boldsymbol{T}_l(i,j) = 1$ 表示$\boldsymbol{W}_l(i,j) \in \boldsymbol{A}_l^{(2)}$。

•量化转换规则

n_1 与 n_2 确定了集合$\boldsymbol{P}_l$，此时可以将 $\boldsymbol{W}_l$ 中的每个权值转换为 $\boldsymbol{P}_l$ 中的值：

$$\widehat{\boldsymbol{W}}_l(i,j)=\begin{cases}\beta\ \mathrm{sgn}\left(\boldsymbol{W}_l(i,j)\right) & \text{if } (\alpha+\beta)/2\le \mathrm{abs}\left(\boldsymbol{W}_l(i,j)\right)\le 3\beta/2\\ 0 & \text{otherwise}\end{cases} \tag{4}$$

其中，α 与 β 为集合$\boldsymbol{P}_l$ 中相邻的元素。

•权值更新

INQ 算法的权值更新使用随机梯度下降法（SGD），具体如下：

$$\boldsymbol{W}_l(i,j)\leftarrow \boldsymbol{W}_l(i,j)-\gamma\frac{\partial L}{\partial\left(\boldsymbol{W}_l(i,j)\right)}\boldsymbol{T}_l(i,j) \tag{5}$$

其中，γ 为正的学习率，L 为损失函数，$\boldsymbol{T}_l(i,j)$ 为二值掩码矩阵，其形状大小与$\boldsymbol{W}_l(i,j)$ 完全一致，其值为 0 或者 1。权值更新时，$\boldsymbol{T}_l(i,j)=0$ 表示对应的权值$\boldsymbol{W}_l(i,j)$ 已被量化，无需更新；$\boldsymbol{T}_l(i,j)=1$ 表示对应的权值$\boldsymbol{W}_l(i,j)$ 未被量化，需正常更新。

INQ 算法的权值更新过程如图 3-2 所示。

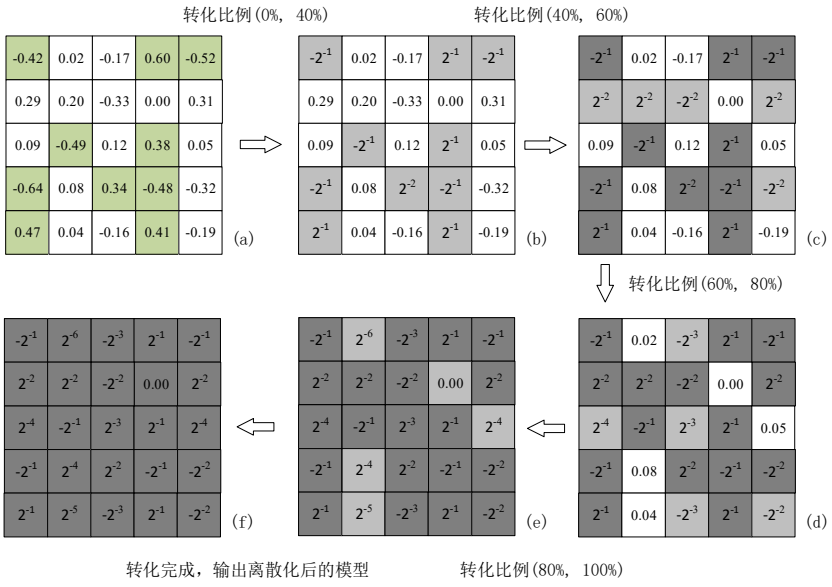

图 3-2：INQ 算法权值更新过程图。(a)权值分组：绝对值较大的绿色标记属于 $\boldsymbol{A}_l^{(1)}$，绝对值较小属于 $\boldsymbol{A}_l^{(2)}$，转化比例 0% → 40%；(b) $\boldsymbol{A}_l^{(1)}$ 组被量化（浅灰色），$\boldsymbol{A}_l^{(2)}$ 组未量化，正常更新（为了方便说明，图示中并未对$\boldsymbol{A}_l^{(2)}$ 组更新）；(c) 对未量化的值继续分组并量化，深灰色表示之前已经量化的值，浅灰色表示此次新量化的值，转化比例 40% → 60%；(d) 继续量化，转化比例 60% → 80%；(e) 继续量化，转化比例 80% → 100%；(f) 所有权值量化完成，输出结果。

⑵ 4bit 存储格式

权值通常使用浮点数据表示，为了使模型变得更小，结合 INQ 算法得到的模型数据特征（所有权值均为 2 的幂次方形式），这里采用 4bit 形式表示一个 float 数据。存储幂次方n，代替存储 2^n。每层权重幂次范围：$[n_1,n_2]$，其中$n_2=n_1+6$。3bit 可表示 0~7 共 8 个数，权重有正负区分，1 表示权重大于 0,0 表示权重小于 0。因此，使用 4bit 数据可表示一个 float 数据。

•单层权重存储格式

表 3-1 单层权重存储格式

char	char	short	short	short	short	short (or float)	short (or float)	…	short (or float)
minimum power	weight data type	N	C	H	W	weight value	weight value	…	weight value

minimum power：代表当前层最小的数据幂次方；

weight data type：0 代表 short，当前层存储的数据为卷积层或全连接层的数据；1 代表 float，当前层非卷积层和全连接层参数，如 bn 层等；

N/C/H/W：代表当前层数据的四个维度信息；

weight value：当前层所有数据。short 表示卷积层和全连接层参数，float 表示非全连接层和全连接层参数，比如 bn 层参数；使用 short 表示卷积层和全连接层参数的方法如下。

•short 存储格式（卷积和全连接层）

每个 float 数据存储其幂次方，幂次方由一个 4bit 数据存储，因此一个 short 数据可存储 4 个权重值。4bit 与 float 数据码本如表 3-2 所示。

表 3-2 4bit 与 float 数据码本

4bit 码本	0000	0001	0010	0011	0100	0101	0110	0111
2 power	-2^-6	-2^-5	-2^-4	-2^-3	-2^-2	-2^-1	-2^-0	0.0
float (*-1)	0.015625	0.03125	0.0625	0.125	0.25	0.5	1.0	0.0
4bit 码本	1000	1001	1010	1011	1100	1101	1110	1111
对应 float	2^-6	2^-5	2^-4	2^-3	2^-2	2^-1	2^-0	N/A
float	0.015625	0.03125	0.0625	0.125	0.25	0.5	1.0	N/A

•对于不同的卷积核大小，数据存储格式如下

1*1 卷积核：4 个 weight 组成一个 short；

2*2 卷积核：2 个 weight 组成一个 short；

3*3 卷积核：3 个 weight 组成一个 short；

n*n 卷积核：n>3 n/3=a...b 3 个 weight 组成 1 个 short（有 a 组） 剩下 b 个 weight 组成一个 short；

例如：

5*5 卷积核：5/3=1...2 则 3 个 weight 组成 1 个 short（有 1 组）剩下 2 个 weight 组成一个 short（高位无效），依次存储。

7*7 卷积核：7/3=2...1 则 3 个 weight 组成 1 个 short（有 2 组）剩下 1 个 weight 组成一个 short（高位无效），依次存储。

3.1.3 实验结果

用压缩后的模型 alexnet、vgg16、googlenet、resnet18、resnet50 和 resnet50-cp 分别在 Imagenet2012 数据集上验证模型精度，结果如下表所示，压缩后的模型精度（inq-accuracy）比未压缩的模型精度（src-accuracy）高；ssd 网络模型在 VOC2007 数据集上的精度也基本没有损失。

表 3-3 Imagenet2012 数据集分类性能结果

network	src-accuracy		inq-accuracy	
	top1	top5	top1	top5
alexnet	0.5676	0.799001	0.568679	0.800041
vgg16	0.682819	0.882702	0.705479	0.899462
googlenet	0.688919	0.889781	0.685699	0.888742
resnet18	0.664547	0.874125	0.688719	0.890282
resnet50	0.72768	0.9101	0.746562	0.924844
resnet50-cp	0.7289	0.9118	0.725739	0.912482

表 3-4 VOC2007 数据集检测性能

network	src	inq
	MAP	MAP
ssd	0.777276	0.775654

3.2 裁剪算法

3.2.1 算法介绍

基于 DNS 算法改进的通道裁剪算法 DNS-CP，首先将每层输入通道个数减少，其次在裁剪好的网络上进行微调，达到减少计算量的同时可保持高精度的目的。

3.2.2 算法细节

•权重更新公式

$$\boldsymbol{W}_k^{(i,j)} = \boldsymbol{W}_k^{(i,j)} - \beta \frac{\partial}{\partial \left(\boldsymbol{W}_k^{(i,j)} \boldsymbol{T}_k^{(i,j)} \right)} L(\boldsymbol{W}_k \odot \boldsymbol{T}_k), \quad \forall (i,j) \in I \tag{6}$$

其中，$W_k^{(i,j)}$表示神经网络第 k 层中角标为(i,j)的权重系数；$T_k^{(i,j)}$表示神经网络第 k 层中角标为(i,j)的权重二值掩码，即 mask blob，其值为 0 或 1，0 表示其对应的权重被删除，

1 表示其对应的权重被保留，T_k的形状大小与W_k相同；β为正的学习率；$L(\cdot)$表示损失函数；⊙表示 Hadamard 乘积算子；I表示权重系数矩阵W_k的角标范围。

•二值掩码矩阵 $\boldsymbol{T_k}$（mask blob）的更新公式

$\boldsymbol{T_k}$ 按照一定的概率进行更新，当$\sigma(iter) > r$ 时，则 $\boldsymbol{T_k}$进行更新；当$\sigma(iter) < r$ 时，则不更新，r 是[0, 1]之间的随机数。

概率函数的表达式如下：

$$\sigma(iter) = \frac{1}{(1 + \gamma * iter)^{power}} \tag{7}$$

其中，iter 为当前迭代的步数，γ 与 $power$ 为超参数，需要用户定义，通常为大于 0 的实数。

$$h_k\left(\boldsymbol{W}_k^{(i,j)}\right)\begin{cases}0 & if\ a_k > |\mu_k| \\ \boldsymbol{T}_k^{(i,j)} & if\ a_k \le |\mu_k| \le b_k \\ 1 & if\ b_k < |\mu_k|\end{cases} \tag{8}$$

其中，$a_k < b_k$，分别为判定二值掩码是否更新的边界。函数$h_k(\cdot)$表示，如果权值μ_k的绝对值小于a_k，则二值掩码 $\boldsymbol{T}_k^{(i,j)}$变为 0，意味着 $\boldsymbol{W}_k^{(i,j)}$将会被裁剪掉；如果μ_k的绝对值大于b_k，则二值掩码 $\boldsymbol{T}_k^{(i,j)}$变为 1，意味着 $\boldsymbol{W}_k^{(i,j)}$将会被保留；如果μ_k的绝对值介于a_k与b_k之间，则 $\boldsymbol{T}_k^{(i,j)}$的值暂时不变，意味着 $\boldsymbol{W}_k^{(i,j)}$是否被保留取决于 $\boldsymbol{T}_k^{(i,j)}$更新前的值。μ_k为第 k 个通道所有参数绝对值的算数平均值。

$$\begin{cases}a_k = \max(0, \mu - c\_rate \times std) \\ b_k = \max(0, \mu + c\_rate \times std)\end{cases} \tag{9}$$

其中，μ 与std 分别表示当前 layer 中所有参数绝对值的算数平均值与标准差，c_rate为用户输入的超参数，一般取 0.1，max(·)函数返回其参数当中的最大值。

•DNS-CP 算法流程

Input: **X**: training datum (with or without label), $\widehat{W}_k$: $0 \le k \le C$: the reference model,
α: base learning rate, f: learning policy.

Initialize $\boldsymbol{W}_k \leftarrow \widehat{W}_k, \boldsymbol{T}_k \leftarrow 1$, $\forall\ 0 \le k \le C$, $\beta \leftarrow 1$, and $iter \leftarrow 0$.

repeat

Choose a minibatch of network input from **X**

Forward propagation and loss calculation with $(W_0 \odot T_0)$,..., $(W_C \odot T_C)$

Backward propagation of the model output and generate ∇L

for $k = 0, \ldots, C$ do

Update $\boldsymbol{T}_k$ by function $h_k(\cdot)$ and the current $\boldsymbol{W}_k$, with a probability of σ(iter)

Update $\boldsymbol{W}_k$ by formula (1) and the current loss function gradient ∇L

end for

Update: iter ← iter + 1 and β ← f(α; iter)

until iter reaches its desired maximum

Output:$\{\boldsymbol{W}_k; \boldsymbol{T}_k : 0 \le k \le C\}$: the updated parameter matrices and their binary masks.

图 3-3 DNS 算法流程

3.2.3 实验结果

用裁剪后的 resnet50 模型在 Imagenet2012 数据集上做了分类性能测试，结果如下表所示，裁剪量达到 50%时的模型精度没有损失，反而有所升高；裁剪量达到 60%时的模型精度略有降低。

表 3-5 Imagenet2012 数据集分类性能

resnet50				
pruned	top1	top5	top1-gap	top5-gap
0	0.727662	0.910144		
0.5	0.728943	0.911824	0.13%↑	0.17%↑
0.6	0.718322	0.907944	0.93%↓	0.22%↓

3.3 特征图量化方法

3. 3. 1 简介

经过对模型的压缩，我们得到了 2 的−n指数形式或者大小的参数（filter）：$2^{-1}, 2^{-2}, 2^{-3}, 2^{-4}, 2^{-5}$... 或者 0.5, 0.25, 0.125, 0.0625, 0.03125, 0.015625... 为了使 FPGA 能够进行低比特移位推理，我们需要进一步对激活值进行量化。一般的激活值分布在 0~1之间，目标是要把 FP32 激活值量化到 int8 的范围之内即 -128~127 之间，如图 3-4 所示，通过量化得到的系数 Q（scale factor）来调整输入输出的大小。

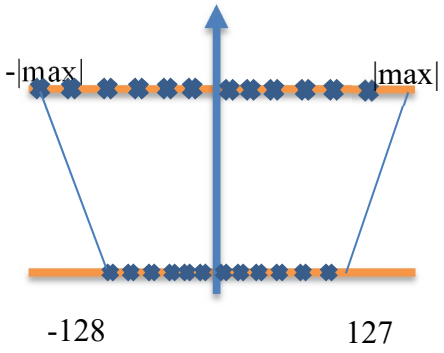

图 3-4 量化示意图

3.3.2 量化方法

（1）求一组数据绝对值最大值

$$max_a = max(abs(T_a)) \tag{10}$$

其中max_a表示绝对值最大值，T_a表示张量，下标a代表 activations 即激活值。

（2）求该组数据量化系数

$$\text{A:}\quad LQ_a = \frac{128}{max_a} \tag{11}$$

$$\text{B:}\quad Q_a = round(\log_2(\frac{128}{max_a})) \tag{12}$$

其中LQ_a, Q_a为量化激活值的系数，可以看到量化分为两种方式，一般通常使用 A 方式，则该组数据可以表示为：

$$[a_{int8}] = round(LQ_a \cdot [a_{fp32}]) \tag{13}$$

但对于 FPGA 的移位推理来说，本量化方法采用 B 方式，则该组数据可以表示为：

$$[a_{int8}] = round(pow(2, Q_a) \cdot [a_{fp32}]) \tag{14}$$

采用 B 种量化方式，我们得到的量化系数完全可以以 2 的指数幂的形式来存储，配合压缩后的参数来使用，卷积计算可以表示为：

$$[a_{int8}]_2 = \sum [a_{int8}]_1 * filter * 2^{-Q_{a1}} * 2^{Q_{a2}} \tag{15}$$

$$[a_{int8}]_2 = \sum [a_{int8}]_1 * 2^{-n} * 2^{-Q_{a1}} * 2^{Q_{a2}} \tag{16}$$

其中$[a_{int8}]_1, [a_{int8}]_2$分别为量化后的 int8 大小的输入层和输出层的值，$filter$ 为压缩参数，形式为2^{-n}，由此可知卷积计算可以通过移位来完成。

（3）量化示例

激活值 Activations: $T_a = [3, 6, -2, 8, 10, -16 \ldots, 5], max(abs(T_a)) = 16;$

量化系数 Scale factor: $Q_a = round\left(\log_2 \frac{128}{16}\right) = 3;$

量化激活值 Quantized-Activations: $a_{int8} = [3, 6, \ldots, 5] \times 2^3 = [24, 48, \ldots, 40];$

（4）量化方式

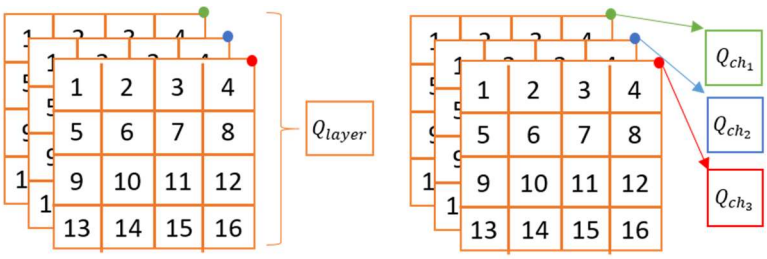

图 3-5 两种量化方式

如上图所示，量化可以分层量化或者分通道量化：

•分层量化

$$Q_{layer} = round\left(\log_2\left(\frac{128}{abs\left(max([a]_{layer})\right)}\right)\right) \tag{17}$$

其中$[a]_{layer}$为网络某层全部的激活值，Q_{layer}为该层量化系数。

•分通道量化

$$[Q_{ch}] = round\left(\log_2\left(\frac{128}{abs(max([a]_{ch}))}\right)\right) \tag{18}$$

其中$[a]_{ch}$为网络某层各通道的激活值，$[Q_{ch}]$为该层各通道的量化系数。

为了保证模型的精度本量化采用的是通道量化的方式。虽然量化系数较分层量化多出很多，但整个网络的量化系数也只占据几十 Kb 的大小，对 FPGA 这样资源较少的硬件来说也可忽略不计。

3.3.3 量化流程

3.3.2 介绍了本量化的方法和原理，在实际的操作中，对于激活值的量化要考虑到整个数据集的特征，量化系数并非根据单张或者某张图片得到，而是最好要涵盖数据集中各个类别的特征。

（1）利用 Caffe DNS 等压缩工具对模型参数 filter 进行压缩，并保证模型的精度基本没有损失，模型参数大小或者形式为2^{-n}。

（2）从数据集的每个类别中各随机挑选一张图片，由压缩的权值进行 FP32 推理，每次推理所得激活值进行暂存，并将激活值的绝对值与下一张推理所得激活值的绝对值进行对比，如果下一张相应位置的元素绝对值较大则替换，否则保持不变。依此动态迭代使得各层各通道中激活值中每一个元素的绝对值为所有图片推理所得激活值中相应位置元素的最大值，将其保存为激活值文件，该文件包含了每个类别的信息。

（3）对上述步骤得到的激活值按通道进行量化，得到网络每层各通道的量化系数$[Q_{ch}]$。

（4）根据每一层各通道$[Q_{ch}]$分布，计算得到该层的Q均值，并将该均值设置为Q值的上限，对于通道中大于该均值的Q强制等于该值，对于小于该值的Q保持不变。

（5）根据量化好的参数和激活值的量化系数Q进行模型的 int8 推理计算。将 int8 计算所得的激活值与 FP32 推理所得激活值进行对比，得到 norm1 误差。并在相应的数据集上验证推理精度，如果精度满足要求则输出量化指数Q，则完成量化。如果推理精度比原始 FP32 推理精度低 1%以上，则根据每层相应的 norm1 error 对Q值进行微调：即选择 norm1 error 相对较大的层，以 1 为步长降低本层内量化指数上限，对通道内的Q值重新判断赋值，即大于该上限的强制等于该上限，不大于的保持不变，直到网络推理数据集测试精度达到要求为止。具体流程可参考量化流程图，上述步骤可总结为：

a) filter_fp32 -> DNS -> 4 bit filter;

b) bottom_features_int8 = bottom_features_fp32*scale_bottom_features;

c) top_features_int32 = conv(bottom_features_int8, filter_int4);

d) top_features_int8 = top_features_int32*scale_top_features/scale_bottom_features;

量化流程如图 3-6 所示。

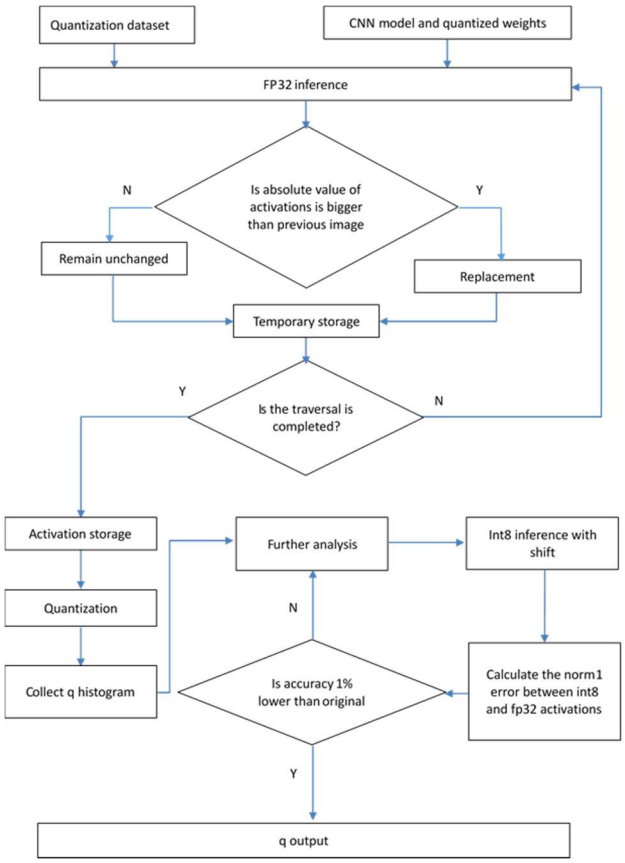

图 3-6 特征图量化流程

3.3.4 量化结果

本量化方法分别对 ResNet50, GoogLeNet, SqueezeNet 以及 SSD 进行了验证，Resnet50 和 GoogLeNet 均在 ImageNet 数据集上进行了分类精度的验证，SqueezeNet 在人脸检测及识别数据集 LFW 上进行了人脸识别精度的验证，SSD 在目标检测数据集 VOC2007 上进行了物体识别的 mAP 精度验证，验证结果如下表所示，与 fp32 相比量化后精度基本没有损失。

表 3-6 模型结果精度对比

Model		ResNet50		GoogLeNet		SqueezeNet		SSD	
Dataset		ImageNet				LFW		VOC2007	
Data type		Fp32	Int8	Fp32	Int8	Fp32	Int8	Fp32	Int8
Accuracy	Top1	71.45%	71.28%	51.9%	51.85%	92.85%	92.82%	77.56%	77.14%
	Top5	90.8%	90.6%	78.6%	78.8%				

3.4 TF2 移位计算实现方法

经过模型压缩裁剪、特征图像量化后，就可以将依赖于 DSP 的乘加运算（MAC）转化移位相加运算（SAC），方法如下：

压缩后的 weights 表示为：

$$weight = (-1)^s * 2^m \tag{19}$$

量化后的 feature 数据表示为：

$$Qfeature = feature * 2^{-Q} \tag{20}$$

假设输入特征图像是 *feature*，输入 weight 数据为 *weight*，输入特征图像的通道数为 *N*，weight 的大小为 *k*k*，卷积的结果为 *result*。则卷积可表示为：

$$result = \left(\sum_{i=1}^{N} \sum_{j=1}^{k*k} feature[i][j] * weight[i][j] \right) + bias \tag{21}$$

将公式（19）（20）带入（21）可得量化后的结果：

$$Qresult * 2^{Q2} = \left(\sum_{i=1}^{N} \sum_{j=1}^{k*k} Qfeature[i][j] * 2^{Q1} * (-1)^s * 2^m \right) + bias \tag{22}$$

即：

$$Qresult\_part = \sum_{i=1}^{N} \sum_{j=1}^{k*k} Qfeature[i][j] * 2^{Q1-Q2} * (-1)^s * 2^m \tag{22}$$

因此，依赖于 DSP 资源的 MAC 运算可转换成逻辑资源即可实现的 SAC:

$$Qresult\_part = \sum_{i=1}^{N} \sum_{j=1}^{k*k} (Qfeature[i][j] \ll (Q1 - Q2 + m)) * (-1)^s \tag{23}$$

4 TF2 架构

4.1 TF2 顶层架构

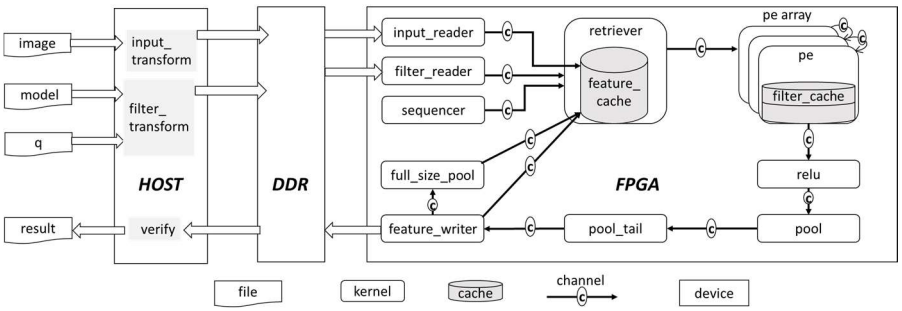

图 4-1 TF2 顶层架构

TF2 顶层架构上图所示，由宿主机（HOST）和 OpenCL 设备（FPGA）构成了 OpenCL 平台。

HOST 端的功能为：

a) 搭建 OpenCL 平台，开辟所需的存储空间；

b) 读取外部图像数据文件 image 数据，经过维度变化、向量化（按向量的方式存储）、量化计算后，以写队列的方式传送给 input_reader kernel。

c) 读取 feature 量化数据 q 和压缩裁剪后模型 model 数据，完成相关运算（参考 3.4），按向量存储后，以写队列的方式传送给 filter_reader kernel。

d) 读取 DDR 里由 feature_writer kernel 输出的推理结果，并验证。

FPGA 端的功能为实现多个 kernel，以向量化并行运算的方式完成 DNN 所有层的推理计算。具体实现过程请参考 4.2。

4.2 TF2 在 FPGA 上的具体实现

如图 4-1 所示，DDR 中图像数据由 input_reader kernel 读入，filter weight 由 filter_reader kernel 读入；sequencer kernel 在推理计算过程中输出当前的层号、feature 数据与 filter weight 的读偏移地址及相关控制信号；retriver kernel 为 pe kernels 提供当前层用于卷积运算的 feature 数据、filter weight 数据及读写段地址（地址=段地址+偏移地址）以及相关的控制信号，并存储当前层输出的 feature 数据（若当前层 feature 输出数据配置为写入片上存储器时）、下一层的 filter weight 数据，为下一层的运算做准备；relu kernel、pool kernel、pool_tail kernel 则实现 DNN 中的激活层和池化层运算，feature writer kernel 把当前层的输出 feature 数据传送给 retriver kernel 进行缓存或输出到 DDR（结果或调试数据）；full_size_pool kernel 实现常位

于 DNN 结尾处的全尺寸池化运算。

4.2.1 Input Reader Kernel

input_reader kernel 从 DDR 中取出 HOST 端存储的 feature 数据，以 for 循环展开的方法，每个 cycle 取出一个 feature 向量，传送给 retriever kernel 向量大小为 *W_VECTOR*C_VECTOR*。feature 向量化方法及循环方向如图 4-2 所示，在列（W）方向和输入通道方向（C）进行向量化，因此，大小为 *C*H*W* feature 包含的向量数量为：

$$N = ceil\left(\frac{C}{C\_VECTOR}\right) * H * ceil\left(\frac{W}{W\_VECTOR}\right) \tag{24}$$

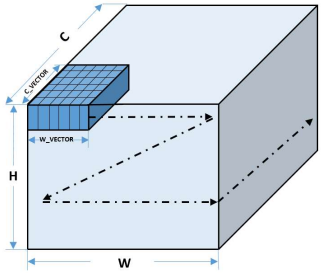

图 4-2 input_reader 向量化

4.2.2 Filter Reader Kernel

与 input_reader kernel 类似，filter 向量大小为 *FW_VECTOR* C_VECTOR*，每个 cycle 读取一个向量大小的 weight 并通过 channel 传送给 retriever kernel。除此之外，还把 filter weight 的写偏移地址以及通道号 n 通过 channel 一起传送给 retriever kernel。注意，如下图所示，由于卷积是 feature 与 *N_VECTOR* 个 filter 并行运算，因此 filter 向量可以理解为 *N_VECTOR*FW_VECTOR*C_VECTOR*，总的向量个数为：

$$ceil\left(\frac{N}{N\_VECTOR}\right) * N\_VECTOR * ceil\left(\frac{C}{M * C\_VECTOR}\right) * R * ceil\left(\frac{FW}{FW\_VECTOR}\right) \tag{25}$$

$$M = \begin{cases} 1 & ,FW=3 \\ FW\_VECTOR & ,FW=1 \end{cases} \tag{26}$$

注意：为了保证 pe kernel pipeline 的一致性，提高运算效率，当 filter 为 1x1 时，每次读取 3 个输入通道 weight，保证卷积运算的一致性。

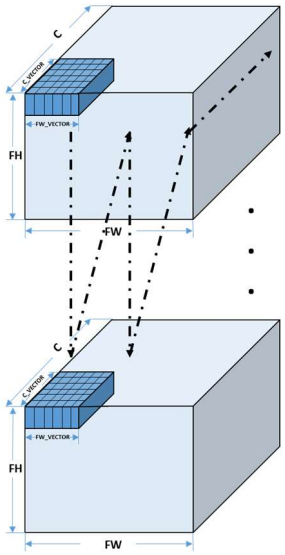

图 4-3 filter 向量化及读取顺序

4.2.3 Sequencer Kernel

在推理计算过程中 sequencer kernel 输出当前的层号、feature 和 filter 数据的读偏移地址及 pe kernel 控制信号，feature 读偏移地址帮助 retriever kernel 从 feature_cache 里读取 feature 数据；filter 读偏移地址与 retriever kernel 的 filter 层地址合成了 pe kernel 读 filter_cache 的地址。因此，sequencer kernel 与 retriever kernel 控制着整个 DNN 推理计算的时序。

4.2.3.1 Sequencer cycle 循环

sequencer kernel 的循环 cycle 即卷积 cycle，应以并行的、向量化的方式理解。如下图所示，input feature 向量与 *N_VECTOR* 个 filter 向量并行卷积得到 output feature 向量，得到一个 output feature 向量所需卷积 cycle 为：

$$N = ceil\left(\frac{C}{C\_VECTOR}\right) * R * ceil\left(\frac{FW}{FW\_VECTOR}\right) \tag{27}$$

大小为 *N*OH*OW* 的 output feature 包含的向量数目为：

$$N = ceil\left(\frac{N}{N\_VECTOR}\right) * OH * ceil\left(\frac{OW}{WOW\_VECTOR}\right) \tag{28}$$

$$WOW\_VECTOR = \begin{cases} OW\_VECTOR & ,FW=3 \\ W\_VECTOR & ,FW=1 \end{cases} \tag{29}$$

注意：

a) 当 filter 为 1x1 时，输出 feature 向量大小为 *W_VECTOR*N_VECTOR*。另，*N_VECTOR* 个 filter 向量并行运算，因此循环 cycle 计算中不能乘以 *N_VECTOR*。

b) cycle 循环嵌套原则：内横向，中竖向，外纵向。

c) 公式（28）针对 *stride=1* 的卷积层，*stride=2* 的卷积层 *OH* 需要除以 2，当 *stride=2*

时横向数据的变化在 pool_tail kernel 中完成变换。

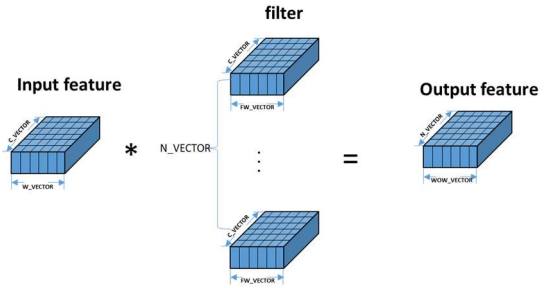

图 4-4 并行卷积示意图

4.2.3.2 Sequencer 控制 filter 读取

filter weights 由 filter_reader kernel 从 DDR 读取后通过 channel 发送给 retriever kernel。由于 channel 为阻塞读写，若 retriever kernel 不读取 channel 内的 filter weights 数据，当 channel 写满后，filter_reader kernel 处于等待状态，直至 channel 被 retriever kernel 读取。

根据 cycle 循环嵌套的原则，最外层的 *ceil(N/N_VECTOR)*次循环可以理解为 DNN 每层的卷积运算分为 *ceil(N/N_VECTOR)*批次完成。retriever kernel 每次读取一批 filter weights 传送给 pe kernel 存储到 filter_cache 里，用于下一批次卷积运算。

sequencer kernel 在每个批次开始时开始计数 cycle，若所计数的 cycle 小于 N_VECTOR 个 filter 大小，sequencer kernel 则认为 filter 正在被 pe kernel 缓存到 filter_cache 里，retriever kernel 根据此标志判断是否允许读取 filter_reader kernel。

4.2.3.3 Sequencer 控制 feature 和 filter 读偏移地址

sequencer 根据卷积向量的维度计算 feature 和 filter 的读偏移地址，然后在 retriever kernel 里与层地址合成完整的读地址。Page, bank 的概念：

Page：每个 feature 向量是一个 page，大小为 *W_VECTOR * C_VECTOR*；

Bank：每 *C_VECTOR* 个数据为一个 bank，因此一个 page 有 *W_VECTOR* 个 bank；

CACHE_PAGE_SIZE：为了存储一层的 featrue 数据需要的 page 数目，最大的输入层为 *256*56*56*，因此有：

*CACHE_PAGE_SIZE = ceil(256,C_VECTOR)*56*ceil(56,W_VECTOR)*；

Feature cache 层偏移地址 = *CACHE_PAGE_SIZE*；

4.2.4 Retriever Kernel

根据图 4-1 可见，retriever kernel 汇集 input_reader kernel、sequencer kernel、filter_reader

kernel、feature_writer kerenl 和 full_size_pool kernel 的信息，控制 *N_VECTOR* 个 pe kernel 并行的进行卷积运算，是 DNN 推理计算的控制中心。retriever 通过 input_reader、feature_writer、full_size_pool 获取 feature 数据进行缓存，通过 filter_reader 获取 filter 数据，进而传送给 pe kernel 进行缓存；retriever 通过 sequencer 获取 feature 和 filter 的读地址偏移，通过 input_reader 和 feature_writer 获取 feature 写地址，通过 filter_reader 获取 filter 写地址。

retriever kernel 把整个推理计算 cycles 分为四种工作状态：input_reading、filter_preloading、conving 和 sequencer_idle。

input_reading：读取并缓存 input_reader kernel 读取的 DDR 图像数据。

filter_preloading：预先读取 filter weight，pe kernel 开辟了双 buffer 空间，在计算过程中，当前批次（即 *N_VECTOR* 个）的 filter weight 读取和下一批次 filter weight 的缓存同时进行，保证 pe kernel 不间断的并行计算。因此，在开始计算之前首先要预读取 filter weight。filter 读取还发生在下一阶段每批次卷积的开始（参考 4.2.3.2）。

conving：此阶段完成所有 DNN 层的卷积运算，retriever kernel 根据 sequencer 提供的偏移地址从 feature_cache 里读出 feature 数据，根据 sequencer 提供的 filter 读控制标志从 filter 读取 filter 数据和 filter 缓存地址，为 pe kernel 提供卷积所需的数据；此外，还为 pe kernel 提供数据可用标志，用于 filter 缓存使能和数据输出使能。

sequencer_idle：卷积计算过程中某些层需要一定的等待时间。

4.2.5 Pe Kernel

pe kernel 完成 DNN 所有层的卷积计算，采用了移位相加（SAC）的技术代替了乘加运算（MAC）。如图 4-5 所示，根据卷积运算的原理，一个 feature 向量与多个 filter 向量卷积得到输出 feature 向量，多个并行的 pe kernel 的数据输入采用了菊花链的方式，feature 向量与 filter 向量依次流经多个 pe kernel，pe kernel 开辟了双 buffer，当 filter 向量的序号 n 与当前的 pe kernel 序号相符合时，pe kernel 就把此向量缓存下来，最终第 n 个 pe kernel 里存储了第 n 个 filter 的全部 weight，用于当前批次的计算，并且预读取了下一批次的第 n 个 fitler 的 weight，实现不间断计算。

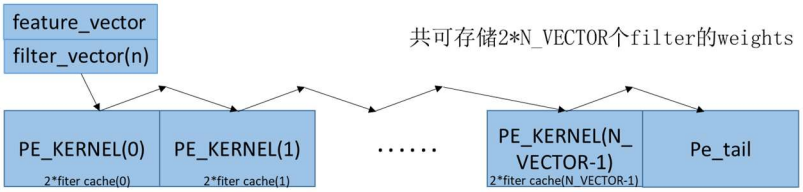

图 4-5 pe kernel 流程简图

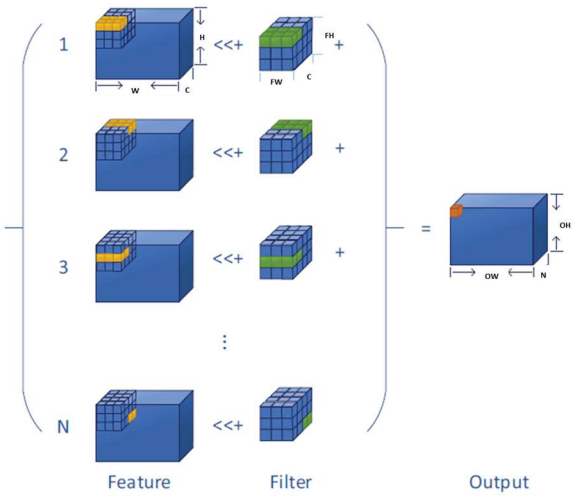

图 4-6 卷积示例

将输入 feature、filter、输出 feature 在列（W）和输入通道（C）方向上进行量化。因此，输入 feature 向量为 *W_VECTOR*C_VECTOR*，filter 向量为 *FW_VECTOR*C_VECTOR*，若 filter 个数为 *N_VECTOR*，则输出 feature 向量为 *OW_VECTOR*N_VECTOR*。如图 4-6 为例，所有向量的行高为 1。*FW_VECTOR* 大小为 1 或 3，卷积步长为 1，则有：$OW\_VECTOR = W\_VECTOR - FW\_VECTOR + 1$。如下图为例，输出通道为 1，即 filter 个数为 1，$W\_VECTOR = FW\_VECTOR = FW = 3$，$C\_VECTOR = 2$，$C = 4$，$R = 3$ 则有 $OW\_VECTOR = 1$，$N\_VECTOR = 1$。此时，输出一个 feature 向量需要的卷积次数 N 为：

$$N = ceil\left(\frac{C}{C\_VECTOR}\right) * R * ceil\left(\frac{FW}{FW\_VECTOR}\right) = 8 \quad (30)$$

输出 feature 共有 M 个向量：

$$M = ceil\left(\frac{N}{N\_VECTOR}\right) * OH * ceil\left(\frac{OW}{OW\_VECTOR}\right) \quad (31)$$

因此，所需要的卷积次数为 M*N。

注意：

a) 有 *N_VECTOR* 个 pe kernel 并行运算。

b) 当 *FW=3* 时，每个 pe kernel 内部 *OW_VECTOR*FW_VECTOR* 个 SAC 并行运算，每个 SAC 将 *C_VECTOR* 个数据移位相加。

c) 当 *FW=1* 时，每个 pe kernel 内部 *W_VECTOR* 个 SAC 并行运算，此时，公式（31）中的 *OW_VECTOR* 替换为 *W_VECTOR*。

4.2.6 Relu Kernel

对卷积计算后的 feature 数据进行激活，注意向量经过 pe kernel 运算后的变换，每个 pe kernel 的输出是 relu kernel 输入数据的一个通道。

4.2.7 Pool Kernel

此 kernel 完成最大池化层运算，通过 W 方向上的最大池化和 H 方向上的最大池化两步完成池化运算。注意的要点如下：

a) relu kernel 处理时没有区分 *FW=1* 和 *FW=3*，在此 kernel 中判断处理；
b) 根据 pe kernel 并行运算机理以及 relu 是对卷积的结果进行计算，并不是每个 cycle 都有可用的结果，因此，在 pool 计算前要先判断是否有可用的结果；
c) 当 *stride=2* 时，卷积和池化只是对 H（竖向）做了处理，横向处理在 pool_tail kernel 中完成；
d) pool_window 大小可调；
e) 根据 pool 的原理以及 stride 的大小，相邻两次 pool 数据有重叠，因此，本次 pool 窗口右边的数据可作为下次 pool 窗口左边的数据，减少数据读取次数；
f) 若 DNN 中当前层后没有 pool 层，输入直接赋值给输出；
g) 注意数据处理维度的变换，卷积运算是按通道进行的，而 pool 层是在 W 和 H 方向上进行的。

4.2.8 Pool Tail Kernel

此 kernel 主要完成 2 个功能：

a) 当 pool stride 大于 1 时，去掉 pool 计算多余的输出数据；
b) 整合结果数据为 *W_VECTOR*，再输出给 feature writer，以提高 feature writer 访问外部 DDR 的效率。

4.2.9 Feature Writer Kernel

此 kernel 实现两个功能，一是首先判断是否有残差层，若有则与 pool_tail 结果相加并激活，若没有则与 0 相加并激活；然后将当前层的输出反馈给 retriever，作为下一层的输入，若为全尺寸池化层前的卷积层，则将数据发送到 full_size_pool kernel 进行平均池化；二是根据配置将输出 feature 数据写入 DDR。

4.2.10 Full Size Pool Kernel

全尺寸最大池化或均值池化运算。

5 TF2 运行环境及性能

5.1 运行环境

TF2 基于 Intel OpenCL 开发环境开发。运行环境安装包括两步：

1) 安装 Intel SDK for OpenCL 16.1 版本；

2) 安装板卡相关 BSP。

浪潮目前有多种不同的 FPGA 板卡，如 F10A、FS10、F37X 等，皆在支持不同的客户需求。其中 F10A 是全球一款支持 Arria 10 芯片的半高半长的 FPGA 加速卡，我们首先在它上面适配了 TF2。如果用户的 FPGA 板卡为 F10A，在 2）步时安装 Inspur BSP for F10A，详细流程请参考《Inspur BSP for F10A 使用手册》。

5.2 性能

基于 F10A 板卡和 TF2 推理引擎，我们测试了多种典型的深度设计神经网络，如 GoogLenet、SqueezeNet、Facenet、ResNet50 等多个网络，在基本没有精度损失的前提下，都达到了超过十倍的加速比。在板卡 F10A 上运行经过 TF2 引擎优化加速的 SqueezeNet、GoogLeNet 和 FaceNet(MTCNN+SqueezeNet)的性能如表 5-1 所示

表 5-1 TF2 性能

NetWork	Throughput(fps)
SqueezNet	1485
GoogLeNet	306
FaceNet(MTCNN+SqueezeNet)	1020

6 总结和展望

目前，我们实现了完整的 FPGA 应用加速方案，包括模型压缩、裁剪、量化等算法，以及高性能 DNN 计算架构，可帮助大家快速实现 FPGA 应用。后期，我们计划在 TF2 中支持更多的网络模型、更多的压缩算法，支持稀疏计算，支持更低比特计算，支持 NLP 模型计算，以及支持更多的板卡，持续优化 TF2 计算架构，以最大化帮助用户以最优的性能、最快的速度搭建实际场景应用。欢迎大家加入 TF2 开发社区，共同发展 FPGA 开发生态。

7 参考论文

1) A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen. Incremental Network Quantiza-tion: Towards Lossless CNNs with Low-precision Weights. International Con-ference on Learning Representations, 2017.
2) Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. In NIPS, 2015.
3) Song Han, Jeff Pool, John Tran, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. ICLR, 2016.
4) Intel. Intel® FPGA SDK for OpenCL™ Programming Guide. Intel Corporation, 2017.
5) Intel. Intel® Arria® 10 Device Overview. Intel Corporation, 2017.
6) Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C Ling, and Gordon RChiu. 2017. An Open deep learning accelerator on arria 10. In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. ACM, 55–64.
7) Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. 2017. Incremental network quantization: Towards lossless cnns with low-precision weights.arXiv preprint arXiv:1702.03044 (2017).
8) Y. LeCun, Y. Bengio, G. Hinton. Deep learning. Nature, vol. 521, no. 7553, pages 436–444, 2015.
9) L. W. Chan, F. Fallside. An Adaptive Training Algorithm for BackPropagation Networks. Comp. Speech and Language, vol. 2, pages 205-218, 1987.
10) K. He, X. Zhang，S. Ren, J. Sun. Identity Mappings in Deep Residual Net-works. European Conference on Computer Vision , pages 630-645, 2016.
11) E. Nurvitadhi, D. Sheffield, J. Sim, A. Mishra, G. Venkatesh, D. Marr. Acceler-ating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC. International Conference on Field Programmable Logic & Applications, pages 1-4, 2016.
12) C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, J. Cong. Optimizing FPGA-based Ac-celerator Design for Deep Convolutional Neural Networks. FPGA, 2015.
13) A. H. Nagpurwala, C. Sundaresan, C. Chaitanya. Implementation of HDLC controller design using Verilog HDL. International Conference on Electri-cal , pages 7-10, 2013.
14) U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, G. R. Chiu. An OpenCL(TM) Deep Learning Accelerator on Arria 10. arXiv, 2017, arXiv:1701.03534.

15) S. Han, H. Mao, W. J. Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. Fiber,vol. 56, pages 3-7, 2015.
16) M. Bečvář, P. Štukjunger. Fixed point algorithm in FPGAs. Acta Polytechnica, Vol. 45, pages 389–393, 2005.
17) M. Courbariaux, Y. Bengio, and J.-P. David. Binaryconnect: Training deep neural networks with binary weights during propagations. NIPS, 2015.
18) M. Courbariaux and Y. Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. arXiv preprint arXiv:1602.02830, 2016.
19) F. Li and B. Liu. Ternary weight networks. NIPS Workshop on Efficient Methods for Deep Neural Networks, 2016.
20) Y. Boo, W. Sung. Structured Sparse Ternary Weight Coding of Deep Neural Networks for Efficient Hardware Implementations. IEEE International Work-shops on Signal Processing Systems, pages 1-6, 2017.
21) S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, W. J. Dally. EIE: Efficient Inference Engine on Compressed Deep Neural Network. International Symposium on Computer Architecture, Vol. 44, pages 243-254, 2016.
22) C. B. Jin, S. Li, T. D. Do, H. Kim. Real-Time Human Action Recognition Using CNN Over Temporal Images for Static Video Surveillance Cameras. Pacific Rim Conference on Multi-media , Vol. 9315, pages 330-339, 2015.
23) M. Foedisch, A. Takeuchi. Adaptive Real-Time Road Detection Using Neural Networks. International IEEE Conference on Intelligence Transportation Sys-tem, pages 167-172, 2004.
24) Convolutional neural network. https://en.wikipedia.org/wiki/Convolutional_-neural_network.
25) Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient dnns. In NIPS, 2016.
26) F. N. Iandola., et al.. SqueezeNet: AlexNet-level accuracy with 50x fewer pa-rameters and< 0.5 MB model size. arXiv pre-print arXiv:1602.07360,2016.
27) Mark Horowitz. Energy table for 45nm process. Stanford VLSI wiki.
28) Sameh Galal. Energy-Efficient Floating-Point Unit Design. Volume 60, IEEE Transactions on Computers, Pages 913~922, 2012.
29) G. A. Vera, M. Pattichis, J. Lyke. A Dynamic Dual Fixed-Point Arithmetic Architecture for FPGAs. International Journal of Reconfigurable Computing, 518602, 2011.
30) P. Judd., et. al. Reduced-Precision Strategies for Bounded Memory in Deep Neural Nets. Vol. 37, Computer Science, 2015.
31) S. Winograd. Arithmetic Complexity of Computations. volume 33. Siam, 1980.

8 附录

8.1 附录 1 审批记录表

角色	签名	日期	备注