1.1 考虑硬件友好和网络小巧的设计思路

1.1.1 填充方式的确定

这里首先设卷积核尺寸为 K×K，图像尺寸为 R×C，以便后续定量的分析不同填充方式所需填充的层数、输出图像的尺寸以及卷积核的移动范围（这里 R=5，C=5，K=3）。图 1.1 (a)、(b)、(c)分别展示了“FULL”、“VALID”及“SAME”三种不同填充方式下卷积核的可移动范围，其中红色部分为卷积核，黄色部分为图像，绿色部分为填充值 0（为了不影响原来的图像像素信息，一般填充 0 值）。

(1). “FULL”方式：

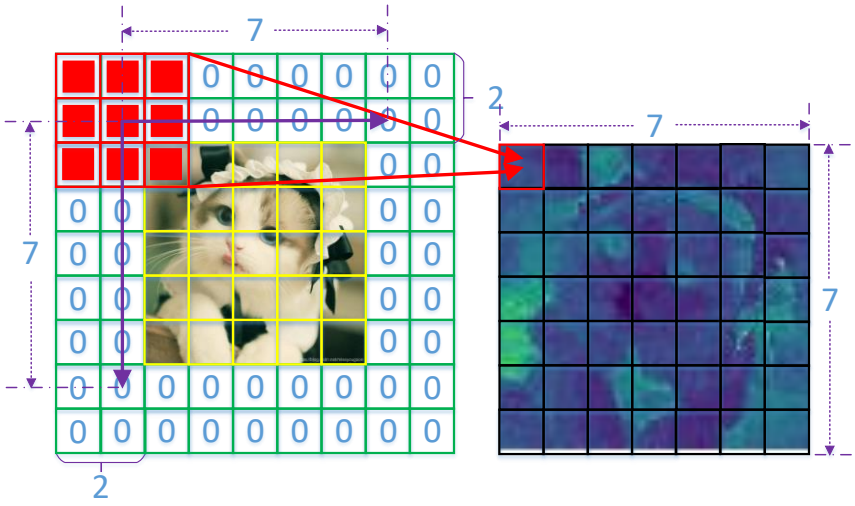
如图 1.1(a)所示，在“FULL”方式下，当卷积核和图像刚刚接触时，卷积操作便开始了。此种方式需在原有图像基础上向上下左右各填充 K-1 维的 0 值来保证卷积操作正常执行。完成卷积操作后，其输出特征图的尺寸为$(R+K-1)\times(C+K-1)$。

(2). “SAME”方式：

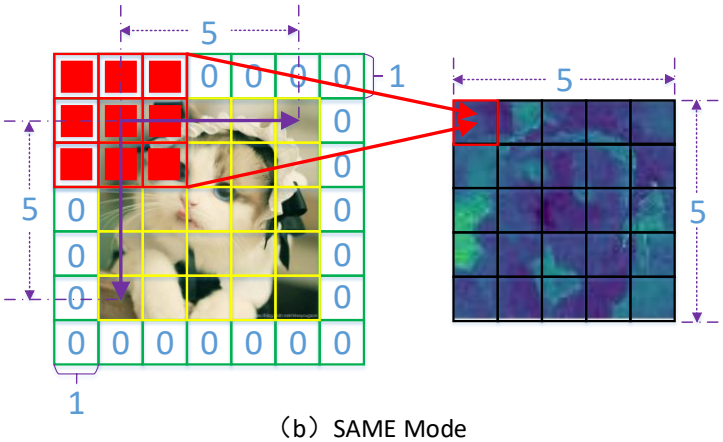
“SAME”方式的意思是当卷积运算的步进为 1 时，其输出特征图尺寸与输入图像尺寸相同，如图 1.1 (b)所示。为保证卷积运算的运行，也需要在原图基础上进行 0 值的填充，但“SAME”模式会尝试向左或向右均匀地填充 K−1 列的 0 值，即将会在左方填充$\lfloor(K-1)/2\rfloor$列 0 值，在右方填充$(K-1)-\lfloor(K-1)/2\rfloor$列 0 值（其中$\lfloor\ \rfloor$表示向下取整操作，另外向上向下的填充方式与向左向右类似）。

(3). “VALID”方式：

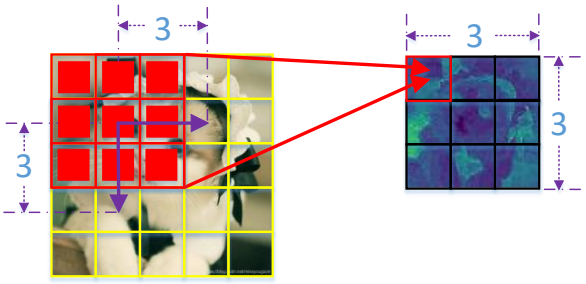
而在“VALID”方式下，如图 1.1 (c)所示，只有当卷积核的所有权值参数全部滑动进入图像内部时，卷积操作才开始。不难看出，其卷积核的移动范围更小，但此种方式并不需要对原图进行任何填充操作，而其输出特征图的尺寸为$(R-(K-1))\times(C-(K-1))$。



（a）FULL Mode



（b）SAME Mode



（c）VALID Mode

图1.1　图像的三种填充方式

当进行硬件实现时，若为加入大于 1 的卷积步进后，方便各层输入图像尺寸的计算，避免出现大量乘除运算从而带来时间的延迟和资源的增加，最好选择“SAME”模式。

1.1.2　卷积核尺寸的设定

(1). 一般将卷积核尺寸设为奇数：

- 卷积核尺寸为奇数可以保证锚点正好在中间，为以中心点做基准进行滑动卷积时带来选取操作上的便利。同时这种滑动方式能够有效防止位置信息的偏移。
- 另外，当采用“SAME” 方式对图像进行填充时，其上下左右填充的 0 层会是对称的。

(2). 倾向于使用尺寸为 3×3 的小卷积核级联代替大卷积核

- 3×3 是最小的能够捕获一个像素的八个邻域以及中心概念的尺寸。
- 如下图 1.2，在“VALID”填充模式下，当卷积操作的步进 S=1，两个具有尺寸为 3×3 卷积核的卷积层级联时，其感受野的效果与一层具有尺寸为 5×5 卷积核的卷积层所带来的相当。因此在保证感受野不变的前提下，可以级联具有小尺寸卷积核的卷积层，用以替代具有大尺寸卷积核的卷积层。

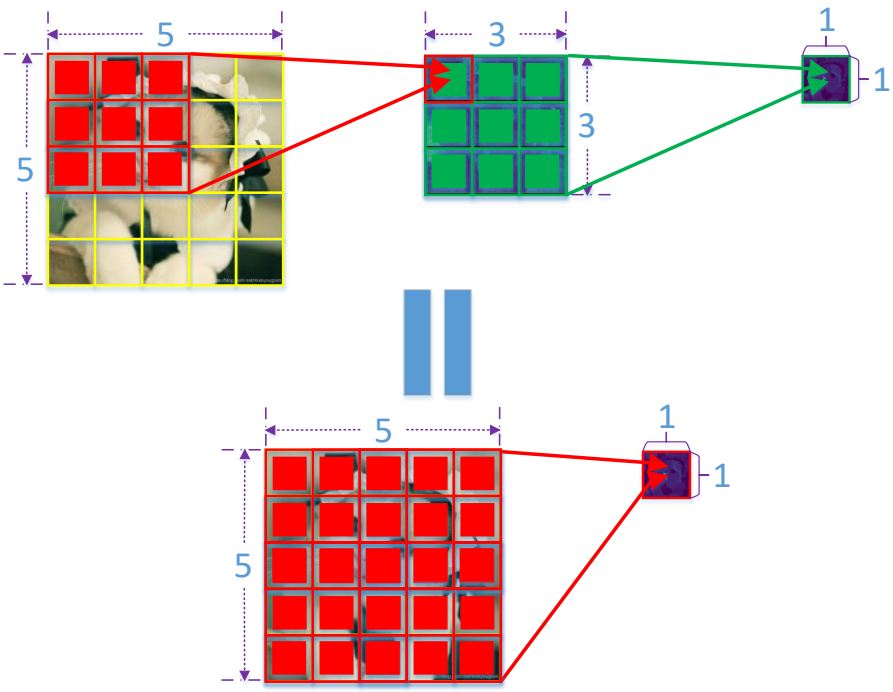


图1.2　小尺寸卷积核级联代替大尺寸卷积核

- 多层具有小尺寸卷积核的卷积层级联与一个具有大尺寸卷积核的卷积层相比，可以增加激励函数的运算次数，所以能够引入更多的非线性，来起到隐式正则化的作用。另外，多个卷积层级联与单个卷积层相比，能对原图提取出深层的更好的特征，这也是现如今网络做的越来越深的一部分原因。
- 多层具有小尺寸卷积核的卷积层级联相比一个具有大尺寸卷积核的卷积层，会带来参数的大量减少。假定所有卷积层的输入输出图像尺寸相同，且均为 N×R×C，则两层具有小尺寸 3×3 卷积核的卷积层仅有参数 2×3×3×N×

$N=18N^2$ 个，而一个具有大尺寸 5×5 卷积核的卷积层却共有参数 5×5×N×N =25 N^2 个，前者比后者在参数量上减少了 28%。

综合上述两点的分析，在保证预测性能不下降，甚至能够提升的前提下，大量减少参数量和计算量，以便在硬件实现时节省资源和降低功耗，将卷积核尺寸设为 3×3 是个不错的选择。

1.1.3 激活函数的选择

在进行激活函数选择前，首先引入梯度消失和梯度爆炸的概念，以便更好的解释具体的选择原因。假定有一个由三个卷积层级联而成的网络结构，每一层的输出为 $g_k(w_k \times x_k+b_k)$，而每一层的期望值为 y_k（其中 k 表示第 k 层网络，g 为该层的激活函数，w 为该层的权重，b 为该层的偏置）。则每一层的误差 $loss=|y_k-g_k(w_k \times x_k+b_k)|$，而当在训练过程中进行反向传播时，通常采用的是梯度下降法来更正参数，则需将 w 修改为 $w_k+r \times \triangle w_k$，其中 r 是学习率。此时 $\triangle w_3$ 便等于 $dloss/dw_3=(dloss/dg_3) \times (dg_3/dg_2) \times (dg_2/dg_1) \times (dg_1/dw_1)$，可见其中$(dg_3/dg_2)$和$(dg_2/sdg_1)$都是激活函数 g 的导数值。因此当 g 的导数值小于 1 时，在更深的网络层，△w 将变得非常小，便造成了梯度消失，使得权重每次迭代更新甚微甚至得不到更新，导致整体网络学习缓慢或受到停滞，令最终的检测效果大大折扣不说，还浪费了很长的时间。相反，当 g 的导数值大于 1 时，则会带来梯度爆炸现象，使得权重参数异常巨大或接近零值，以至超出所使用数据类型的表示范围而出现大量的 NAN 值，给网络带来很大的不稳定性。

因此无论现在已被广泛使用的 Sigmod 函数或 Tanh 函数，都会出现梯度消失的情况（其中 Tanh 的函数及其导数图像如图 1.3 所示），因为当其输入较大或较小时，其导数均接近于零。而当将其进行硬件实现时，因 Sigmod 函数或 Tanh 函数过高的复杂度只能采用多项式拟合或查表法来近似，以便求得较小的计算延迟，但这带来了极大的资源浪费和精度损失，并且这种较小的计算延迟相比简单的加减操作来说还是太大了。而 ReLu 函数，其函数及其导数图像如图 1.4 所示，因其导数始终为一，有效避免了梯度消失和梯度爆炸的现象，并且极其易于硬件实现，仅仅相当于一个与零相比的比较器，可直接通过判断符号位来实现。因此选用 ReLu 函数作为本文设计的激活函数。

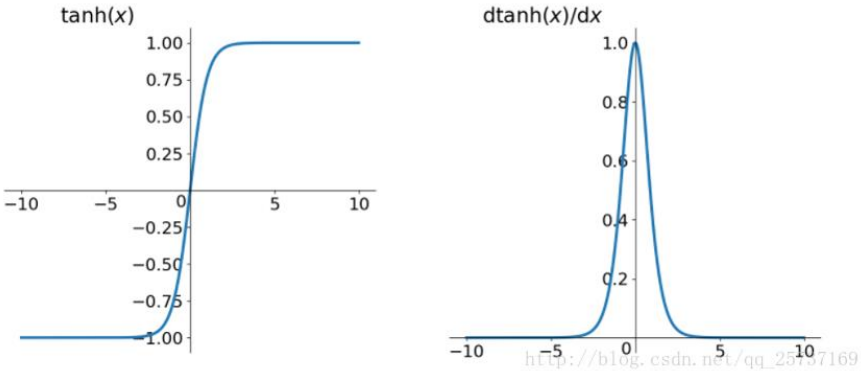


图1.3　Tanh 的函数及其导数图像

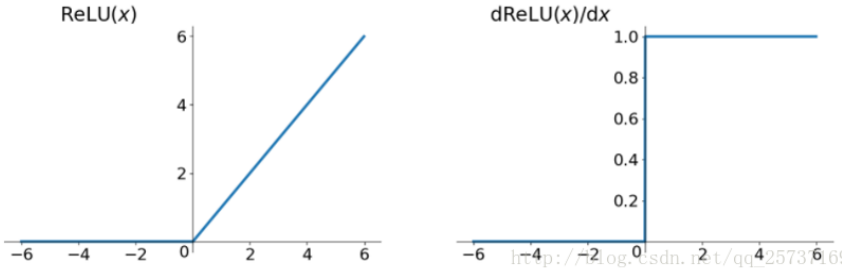


图1.4　ReLu 的函数及其导数图像

1.1.4　以步进代替池化

传统的深度卷积神经网络在其内部都是激活函数后接池化，来达到大大减少下一层参数和计算量的目的。而观察发现大多数激活函数，如 Sigmod 函数、Tanh 函数或 ReLu 函数等为单调递增函数，则卷积所得的结果经过上述激活函数后，并不会改变彼此之间的大小关系，因此当采用最大池化时，可以将激活与池化的操作顺序颠倒，利用池化降维的功效大大减少激活函数的非线性运算量，并带来速度的提升（当然，此种技巧并不适用于平均池化）。

然而此种方法所带来的运算量的减少还不够。由于池化并没有引入参数，相当于一个单纯的尺寸减小器，而在卷积操作过程中设置大于 1 的步进可以带来相同的降维效果。比如卷积步进为 2 和池化区域尺寸为 2×2 相比，输出特征图的尺寸相同，更可喜的是感受野也等效。另外，卷积步进=2 相比池化来说，可以减少 3⁄4 的激活运算量的同时还可以减少 3⁄4 的卷积运算量。因为在卷积核以步进为 2 进行滑动的过程中，当其在输入图像上的映射区域的中心坐标不是 2 的倍数时，可以设置条件不进行卷积运算而只进行数据的平移。

1.1.5 以平均池化代替全连接

全连接层可以对特征进行高度的提纯，并充当分类器的作用，但是与普通的卷积层相比，参数量过于庞大。假定需要分类的类别只有 4 种，但对于全连接层的输入图像的每一个像素都存在与之对应的 4 个权重，会带来极大的参数量，增加了硬件内部的存储以及因数据读取所造成的功耗，尤其是第一个与最后的卷积层相连的全连接层。

假设网络当中只有一个全连接层充当分类器，如果使用全局平均池化代替该全连接层，可以省去这一全连接层的全部参数。同时，假定最后一个卷积层的输出特征图的尺寸为 16×16×4，即输出通道为 4 恰为分类个数，全副图像池化相比全连接节省了 16×16×4=1024 倍的乘法运算。但如果最后一个卷积层的输出特征图的尺寸为 16×16×512 的话，使用该方法得到的是 512 个值，相比四分类来说个数太多，可以再接一个小的全连接层，但这样也比卷积后直接全连接去分类带来的运算量和参数量要少的多。另外，最大池化更适合提取极端特征，在池化区域尺寸较大时，会造成较多的信息丢失，而平均池化相当一个期望特征检测器，可以利用上所有的信息。经过实践验证，全副图像平均池化相比全连接，其最终的分类效果并没有带来任何减少。

1.1.6 尺寸不变深度增加

在硬件实现时，大多种任务都不能在一个时钟周期内完成，假定一个任务的处理时间 t 为 8 个时钟周期。但多次执行时可以在消除数据依赖的情况下，进行流水处理来缩短总的处理时间，如下图 1.5 所示。可毕竟一个任务需要一定的时延 t 才能产生结果，因此即使进行了流水处理，连续执行 8 次该项任务所需的总时间 T 并不是 8 个周期，而是 8+(8−1)=15 个周期，这个在图也有所示意。所以随着任务执行次数的减少，任务的处理时延 t 所占总体处理时间 T 的比例则会越来越大，将带来资源利用率的降低，最终导致整个网络的处理速度下降。

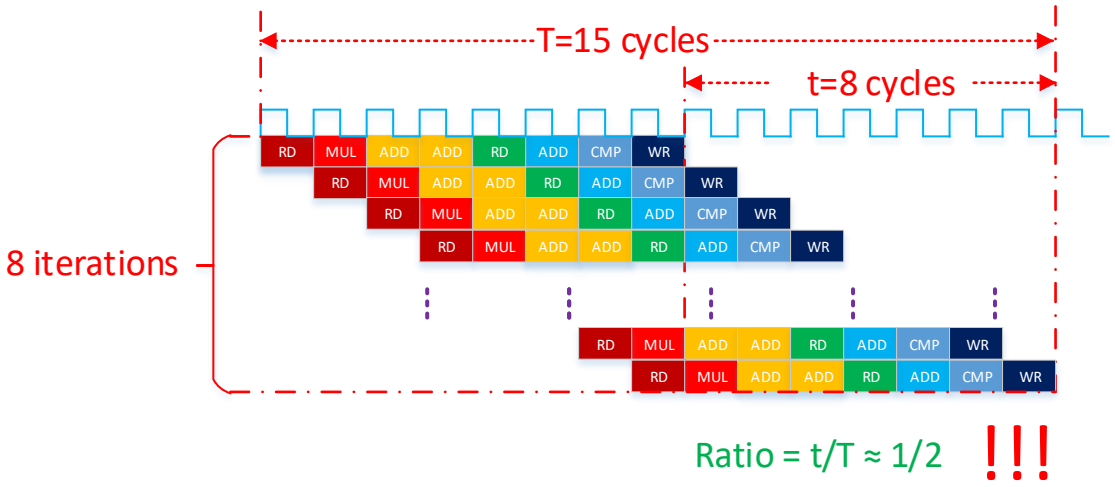


图1.5 流水任务的处理总时间

实际上，任务的执行次数与特征图的尺寸成正比，因此当特征图的尺寸减小到一定大小时，又想让网络做的更深以获得较好的预测精度，就需要保持特征图尺寸不变，那么卷积步进 S 应设为 1。此时如果特征图数量仍要逐层增加，将会给下一卷积层带来计算量、参数量及处理时间的指数级增加。另外，在硬件实现时，本人将在网络层与层之间采用流水架构来提高整体网络的处理速度，但是流水线的处理周期取决于流水线上具有最长处理时间的子任务，所以特征图数量增加给下一层卷积所带来的处理时间的增加，将会导致整个网络的处理速度比预计的要低很多。

解决方法就是采取将特征图的数量增加和减少交替的方案，如此，即不会带来参数量、计算量的增加，又保证了各卷积层处理时间大致相同。下表 1.1 简要给出了当特征图尺寸不变，网络深度增加时特征征图数量变化的具体实施方案（其中卷积核的尺寸为 1×1），及各层的计算量、参数量和处理时间。相对应的，表 1.2 给出了当特征图数量逐层增加的情况，两者对比可以佐证上述结论。

表1.1 特征图增减交替

Layer/Stride	Conv1/1	Conv2/1	Conv3/2
Input Size	32*256*256	64*256*256	32*256*256
Mult-Adds	32*256*256*64	64*256*256*32	32*256*256*64
Parameters	32*64	64*32	32*64
Time	T	T	T

表1.2 特征图逐层增加

Layer/Stride	Conv1/1	Conv2/1	Conv3/2
Input Size	32*256*256	64*256*256	128*256*256
Mult-Adds	32*256*256*64	64*256*256*128	128*256*256*256
Parameters	32*64	64*128	128*256
Time	T	4*T	16*T

1.1.7 深度可分离卷积

本人最早了解到深度可分离卷积是通过 MobileNet，MobileNet 是谷歌为移动设备打造的轻量级网络，其在目标检测、人脸识别、图像分类等应用上与其他网络相比参数少、计算小、速度快，且在未引入分辨率和通道数这两个超参数时预测精度几乎

没有任何损失。MobileNet 凭借所展现出来的上述优异的性能，迅速引起了广泛关注，并出现了大量移动端基于该网络的应用且收获了很好的效果。其核心就是采用深度可分离卷积代替传统的标准卷积。

深度可分离卷积可以理解为是传统的标准卷积的因式分解，一层深度可分离卷积本质上可以看作一层深度卷积后加一层逐点卷积来实现的。其中滤波功能由深度卷积实现，而逐点卷积则承担通道转换的作用。逐点卷积故名思意，其实就是传统的标准卷积，只不过是卷积核的尺寸为 1×1，可以看作一个点，因此叫做逐点卷积。而深度卷积与传统的标准卷积有着很大的区别，其卷积过程如下图 1.6 所示，用公式表达为公式(3-1)（其中 W、I、O 分别代表权重、输入和输出，而 K 和 N 分别代表卷积核的尺寸（长宽均为 K）及图像通道数量（输入输出均为 N），输入图像尺寸为 R×C，输出特征图尺寸为 H×L，卷积步进为 S）。

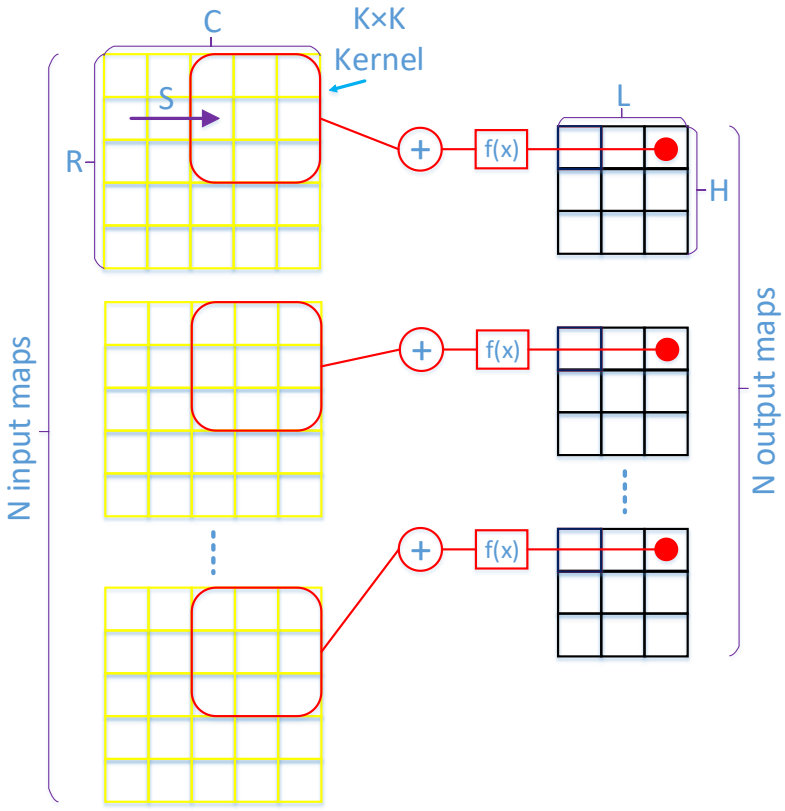


图1.6　深度卷积操作过程

$$O_{n,h,l}=\sum_{i=0}^{K-1}\sum_{j=0}^{K-1}W_{n,i,j}*I_{n,(h*S+i),(l*S+j)} \tag{3-1}$$

整个深度可分离卷积的计算过程与传统的标准卷积的对比如下图 1.7，其具体的含义反映到公式上就是当进行传统的标准卷积时，对于计算得到第一个输出特征图上

某一位置，假设坐标为(0,0)即左上角位置的像素值，需要经过如公式(3-2)的卷积运算(其中 W 为权值参数，I 代表输入，O 便是输出，而 K 和 N 分别代表卷积核的尺寸及输入图像通道数量，0 代表计算第一个输出特征图)。同理，其他输出特征图上同一位置像素值的计算也需要经过这样的过程。而对于深度可分离卷积，它是先完成公式等号左边的部分，并根据输入通道的数量将其等分，属于同一输入通道的计算被分在在同一部分，记为 Tmp，则 Tmp 的计算方式为公式(3-3)，然后每一部分乘上一个系数 α 再经过一个非线性函数 f 去拟合原先的值，如公式(3-4)。而在进行其他输出通道的相关计算时，将不再重复公式等号左边的部分，而直接乘上一个系数再经过一个非线性函数去拟合。如此，便大大减少了参数量和计算量。

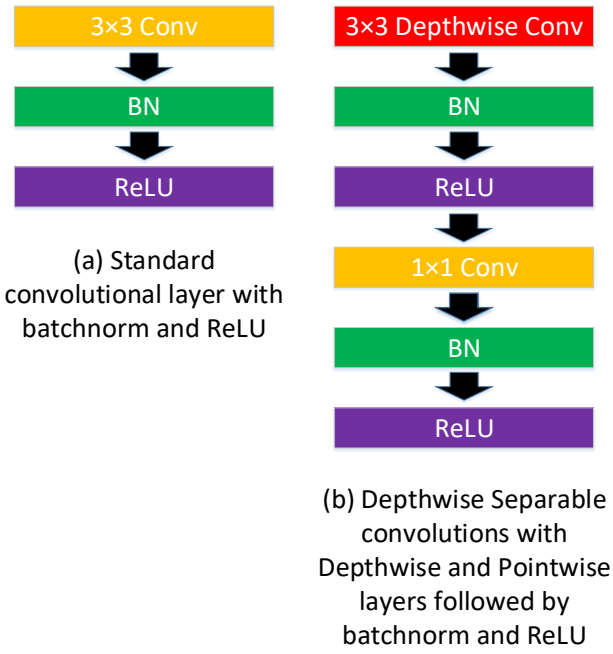


图1.7　标准卷积与深度可分离卷积的对比

$$\sum_{i=0}^{K-1}\sum_{j=0}^{K-1} W_{0,i,j} * I_{0,i,j} + \sum_{i=0}^{K-1}\sum_{j=0}^{K-1} W_{1,i,j} * I_{1,i,j} + \cdots + \sum_{i=0}^{K-1}\sum_{j=0}^{K-1} W_{n-1,i,j} * I_{n-1,i,j} = O_{0,0,0} \tag{3-2}$$

$$Tmp_n = \sum_{i=0}^{K-1}\sum_{j=0}^{K-1} W_{n,i,j} * I_{n,i,j} \tag{3-3}$$

$$f_2\left(\alpha_0 * f_1\left(Tmp_0\right) + \alpha_1 * f_1\left(Tmp_1\right) + \cdots + \alpha_{n-1} * f_1\left(Tmp_{n-1}\right)\right) = O_{0,0,0} \tag{3-4}$$

对于一个输入图像尺寸为 N×R×C，输出特征图尺寸为 M×H×L，具有 M 组尺寸为 K×K×N 的三维卷积核的标准卷积层，其计算量如公式所示(3-4)，而深度可分离卷积处理相同尺寸的输入图像且输出特征图尺寸也相同时的计算量如公式(3-5)所示，其中，卷积步进 S=1，则 H=R、L=C。可以得出计算量减少为前者的($1/M+1/K^2$)。

另外，图 1.8 给出了由标准卷积分解为深度卷积和逐点卷积时卷积核的变化。

$$K*K*H*L*N*M \tag{3-4}$$

$$K*K*H*L*N+H*L*N*M \tag{3-5}$$

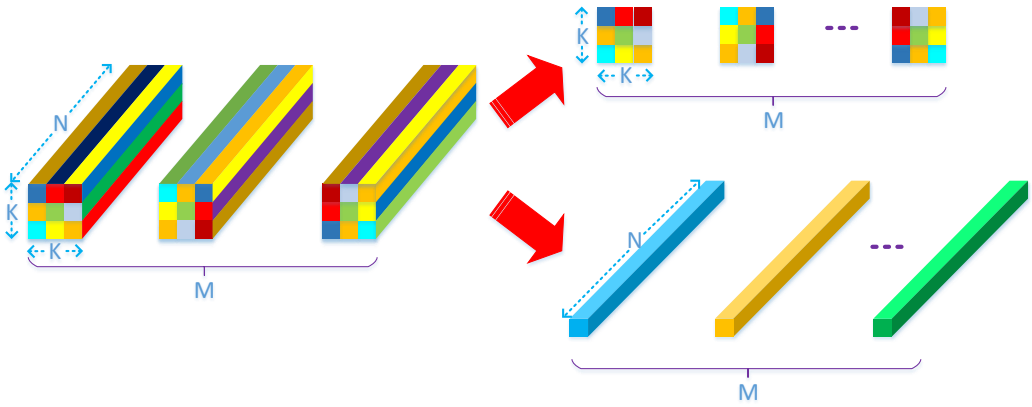


图1.8　卷积核的变化

如此操作的原因是采用标准卷积的传统深度卷积神经网络的参数大多都很小，接近于零，几乎对预测的结果起不到任何作用，而深度可分离卷积相当于稀疏了原标准卷积的参数矩阵，将很多非常小的系数去掉然后通过学习把重要的参数保留了下来，或者说它将这些非常小的系数联合在了一起去预测。

这里，增加非线性计算的原因是乘上一个系数仅仅是对原来值的伸缩，无法表征更复杂的关系，从而映射到其他输出特征图上去，而通过一个非线性函数后则能更好的拟合二者之间的关系。另外，根据输入通道的数量进行等分而不是直接组合在一起的原因是为了保留输入通道各图像之间的差异性。

表 1.3 将 MobileNet 与流行的网络，如 GoogleNet 和 VGG16，通过在 ImageNet 数据集上的表现进行了比较。从表中可见 MobileNet 有着和 VGG16 同样的检测精度，却只有前者 1/27 的计算复杂度和 1/32 的参数量。另外，与 GoogleNet 相比，在计算量和参数量都差不多只有前者 2/5 的同时，精度却还要高。由此得出，深度可分离卷积是多么的强大，大有取代标准卷积的趋势。

表1.3 MoblieNet 与先进网络的对比

Model	MoblieNet	GoogleNet	VGG 16
ImageNet Accuracy	70.6%	69.8%	71.7%
Mult-Adds	569M（Million）	1550M	15300M
Parameters	4.2M（Million）	6.8M	138M

此外，当一层深度可分离卷积总体的卷积步进为 2 时，由于深度卷积是负责滤波的，所以卷积步进 S=2 应加在深度卷积上而不是逐点卷积上。进行硬件实现时，由于深度可分离卷积与传统标准卷积在操作上的相似性，因此可不必将深度卷积和逐点卷积分开当作两层卷积，可以一并使用标准卷积的操作来处理。只不过是像是在卷积过程中对输入数据多了一部分处理，并且这部分处理是可以按通道分离独立进行，所以才可以合并操作。用公式形容便是（其中 I、O 分别代表输入和输出，而 N 和 M 分别代表输入输出通道数，输出特征图尺寸为 H×L，深度卷积的卷积核尺寸为 K×K、步进为 S，另外 W^d、W^p 分别是深度卷积和逐点卷积的权重，I^*便是处理后的输入）。如此便减少了一部分存储和时间开销。

$$O_{m,h,l} = \sum_{n=0}^{N-1} I^*_{n,h,l} * W^p_{n,h,l} \tag{3-6}$$

$$I^*_{n,h,l} = \sum_{i=0}^{K-1}\sum_{j=0}^{K-1} I_{n,(h*S+i),(l*S+j)} * W^d_{n,i,j} \tag{3-7}$$

除了深度可分离卷积，MobileNet 还引入了分辨率和通道数这两个超参数，并利用它们来降低卷积过程中特征图的数量和分辨率，以便更进一步的减少整体网络的参数量和计算量并缩短处理时间，但此种方法带来的预测精度的损失还是比较大的。

1.2 考虑资源功耗及速度灵活的设计思路

1.2.1 输出数据重利用

由于 DCNNs 过多的层数和 FPGA 的资源有限，导致很难将整个网络实现在一块 FPGA 上，因此许多的设计采用一层一层的加速方式，即在 FPGA 内部只实现了一层卷积神经网络。在当前层的计算过程中，其输出结果会暂存在片外，当处理完本层后会将其读入作为下一层的输入，并重新配置卷积各参数如输入输出通道数等，再开始下一层的计算。但由于 DCNNs 中的各卷积层并不相同，如图像尺寸等，因此在 FPGA 内只实现一层卷积网络极容易造成资源利用率的降低，使得总体使用的资源和处理速度上升。另外，这种方法在一个卷积层处理当前图像时，无法同时执行上一层对下一副图像的计算，而将整个网络实现在一块 FPGA 上时，可以引入流水机制来实现上一目的，如此将会为整个网络带来层数倍的提速。而解决在一块 FPGA 上实现整个网络这一难题的方法之一便是分块处理，即由于片内资源有限，无法同时处理一层内部的所有数据，便将其分块后一块一块的进行处理。因此分块处理可以极大的减少片内资源，使我们可以在资源有限的情况也能实现复杂的网络。而如此操作有可能会带来数

据的重复读取，从而增加读取操作的功耗。为减少数据的重复读取，就需要尽可能地重复利用数据，这就是数据重利用。在硬件实现中，数据的读取操作尤其是从片外到片内需要消耗很多的能量，有时甚至比内部的运算还要多，因此对于减少数据读取次数的数据重利用方式来说是降低功耗的有效方式之一。

而一层卷积网络中与片外存储有关的数据有输入、输出和权重，便有了不同的以相应数据种类为核心的数据重利用方式，因此选择一个合理的数据重利用方式能最大化地减少存储资源的访问次数。下面以一层卷积为例，定量的分析上述不同数据重利用方式总的存储访问次数并进行对比，从而设计出一个合理的适用于本文设计的重利用方式。一层卷积的加速器系统如图 1.9 所示，包含了加速器和片外存储（DDR SDRAM：Double Data Rate Synchronous Aynamic Random-Access Memory，双倍速率同步动态随机存取内存）。在加速器内部有总控器，卷积运算单元和读（RD：Read）、写（WR：Write）逻辑。读逻辑负责图像和权重的输入，Input 和 Weight Buffer 分别暂存输入图像和权重，然后卷积运算单元再将它们从 Buffer 并行的加载到内部寄存器中，并通过并行多个卷积引擎（CE：Convolution Engine）来进行卷积操作，在此过程中间数据可以暂存在其内部 Output Cache 或卷积运算单元外部、加速器内部的 Output Buffer 中，最后在写逻辑的作用下，将最终结果存到外部存储中并作为下一层卷积的输入。而总控制器控制着输入、计算和输出这三进之间的流水。本小节的论述都是基于此模型。

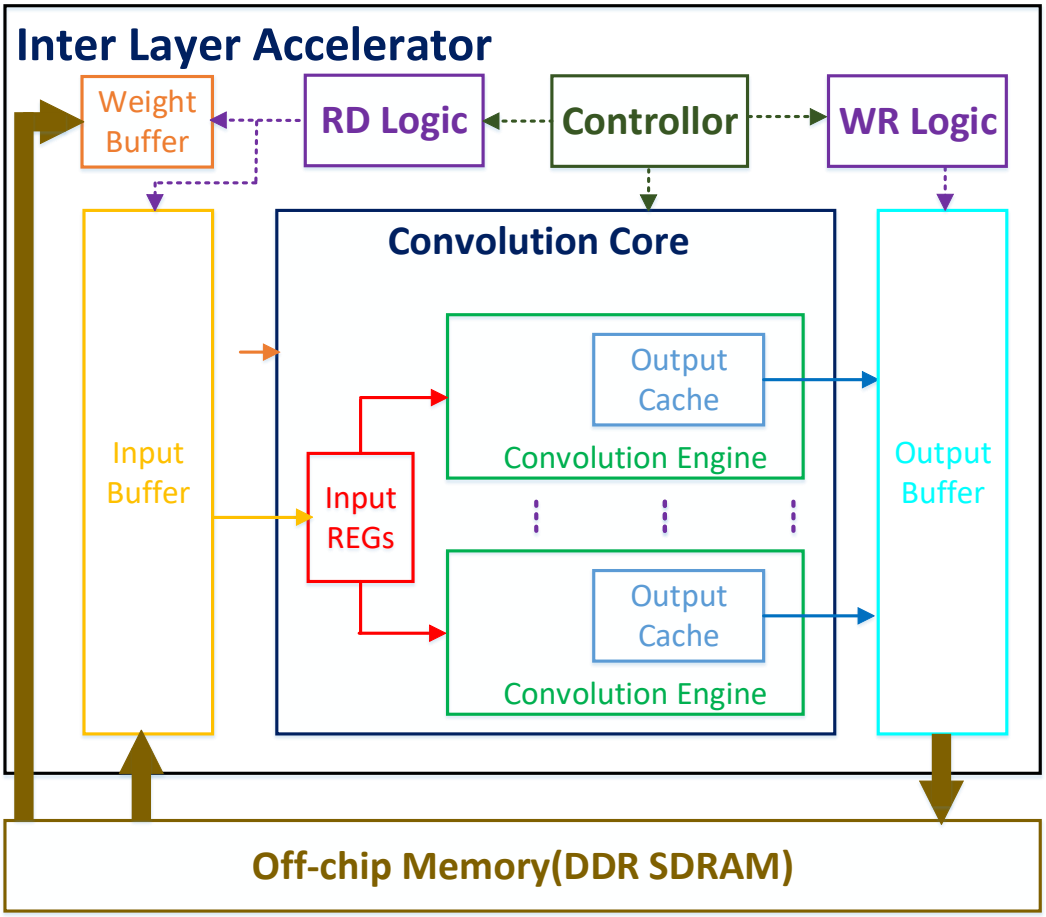
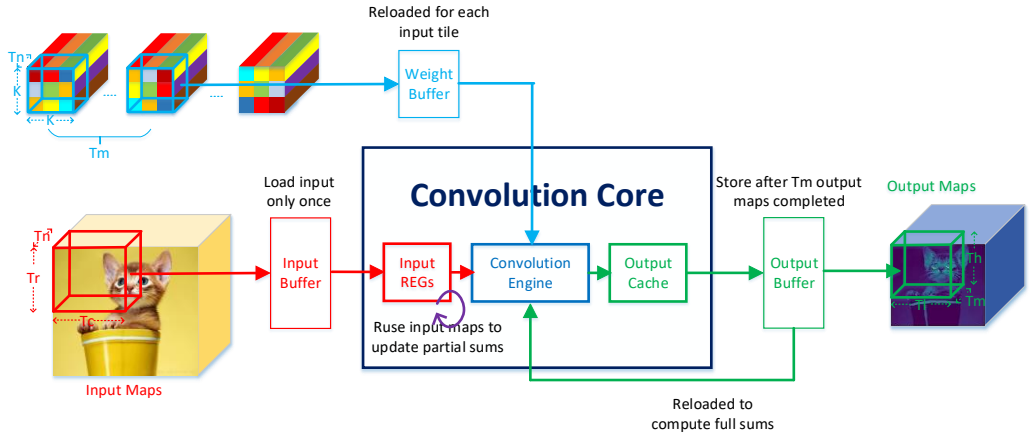


图1.9 一层卷积的加速系统

如图 1.10(a)，由于分块机制的存在，输入图像由尺寸为 R×C×N 分成了尺寸为 Tr×Tc×Tn 的小块，则此时的输出特征图也是小尺寸的，为 Th×Tl×Tm，另外 K 是卷积核的尺寸（长宽相同）。分块操作过程可以由图 1.10(b)中的伪代码等效，相比第二章中的多了几层 for 循环，而外围四层循环的上下顺序决定了数据重利用的方式。在一个卷积层的计算过程中，对于 Buffer 和片外 DDR SDRAM 的总访问次数都可以用以下公式(4-1)表示：



（a）

```
for (int h = 0; h < H; h += Th)//Loop H
  for (int l = 0; l < L; l += Tl)//Loop L
    for (int n = 0; n < N; n += Tn)//Loop N
      for (int m = 0; m < M; m += Tm)//Loop M
        for (int tn = 0; tn < Min(Tn, N - n); tn++)//Loop Tn
          for (int th = 0; th < Min(Th, H - h); th++)//Loop Th
            for (int tl = 0; tl < Min(Tl, L - l); tl++)//Loop Tl
              for (int tm = 0; tm < Min(Tm, M - m); tm++)//Loop Tm
                O[m][r][c] +=//MAC operation
                  ∑_{i=0}^{K-1} ∑_{j=0}^{K-1} W[m][n][i][j] * I[n][h * S + i][l * S + j];
```

（b）

图1.10 输入数据重利用

$$MA=TI*\alpha_i+TO*\alpha_o+TW*\alpha_w \tag{4-1}$$

其中 TI、TO 和 TW 分别代表当前卷积层输入、输出及权重的总数据量，而 α i，α o 和 α w 则表示它们在分块操作时重复读取和存入的总次数。

输入数据重利用即以输入图像数据为核心，最小化输入数据读取次数的重利用方式。如图 1.10 所示，该方式可分为三个阶段，这一过程可以用图中的多层循环描述：1）卷积计算单元将输入图像数据从片外 DDR 导入片内 Buffer 中，再加载到相应的输入寄存器上；2）这些输入数据将会得到充分的重利用，具体反映到图上的伪代码便是 N 循环安排在了 M 循环之上。此时，由于一次分块操作只读入了 Tn 个通道的输入图像，因此输出只是一个部分和而不是最终的结果。又因卷积运算单元内部的输出寄存器数量有限，需要将这些部分和暂存在 Output Buffer 中并导出到外部存储；3）当计算到下一 M 循环时，再从 Output Buffer 读出相应的上一部分和后与当前部分和进行累加，直到累加完成后将其存入到外部 DDR SDRAM 中。这里，先将部分和暂存在 Buffer 中再导入片外存储而不是直接将部分和存在片外的原因是：为了提高 DDR 的吞吐率而设置的突发长度较长时，由于 DDR 控制器的机制，在这一次突发过程中数据并不一定是连续存入 DDR 的，因此需要一个 Buffer 来充当缓存，另外还可以解决内部处理时钟与 DDR 不同的跨时钟问题（当 DDR 时钟与加速器内部处理时钟不

同时），Input Buffer 的存在也是基于此。

在输入数据重利用方式中，对于 Buffer 的访问次数来说，所有的输入数据只加载了一次，因此 α i=1。实际上当卷积核的尺寸 K 大于 1 时，α i 要比 1 稍大。如图 1.11，当将尺寸为 16×16 输入图像分成 8×8 的小块进行处理时若卷积核尺寸为 3×3、步进为 1，则在各个小块交界处有着数据的重复读取以保证卷积操作的正确性（其中填充方式为“SAME”，这里为方便计算和画图示意没有进行均匀填充，而只在图像左侧和下方进行了填充，实际实现中采用的还是均匀填充）。且总的读取数据量变为(16⁄8×(3−1)+16)×2=400，是原来 16×16=256 的 1.56 倍，而当 K 占 Tr 及 Tc 比例较小时，可以忽略此部分重复，如 Tr=16、Tc=32 时而 K 仅为 3。

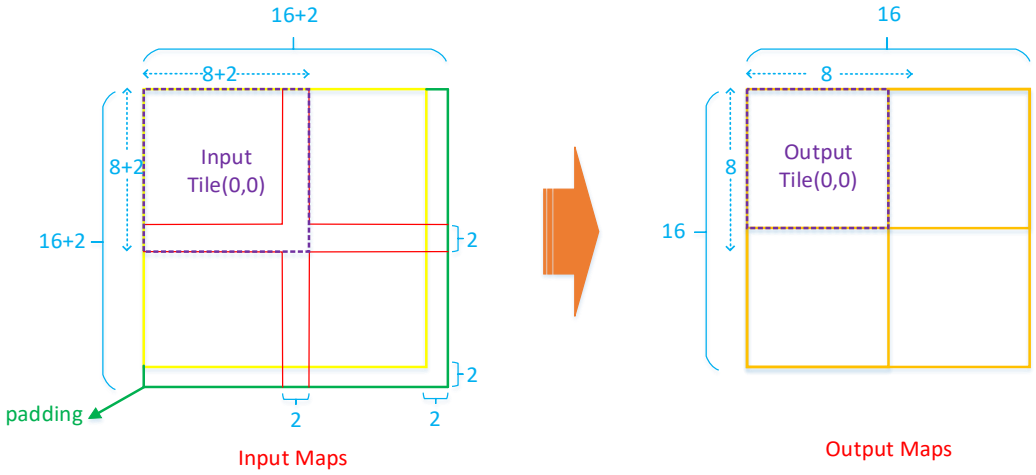


图1.11 分块导致的数据重叠

而对于 α o，因为需要将计算过程中的部分和暂存在 Output Buffer 中，则对 Output Buffer 的总访问次数为([(N⁄Tn)]−1)×TO，即 α o 为([(N⁄Tn)]−1)。最后对于权重，在每读入一小块输入时，都要加载相应 TW 的权重，因此权重的重复读取次数为 [(R⁄Tr])×[(C⁄Tc)]。

另外，对于片外 DDR SDRAM，当片内缓冲 Buffer 较大时会大大减少它的访问次数。其 α i，α o 和 α w 可以分别表示如下：

$$\alpha_i = 1$$

$$\alpha_o = \begin{cases} 0, & M * Th * Tl \le B_o \\ 2 * \left(\left\lceil \frac{N}{Tn} \right\rceil - 1 \right), & M * Th * Tl > B_o \end{cases} \tag{4-2}$$

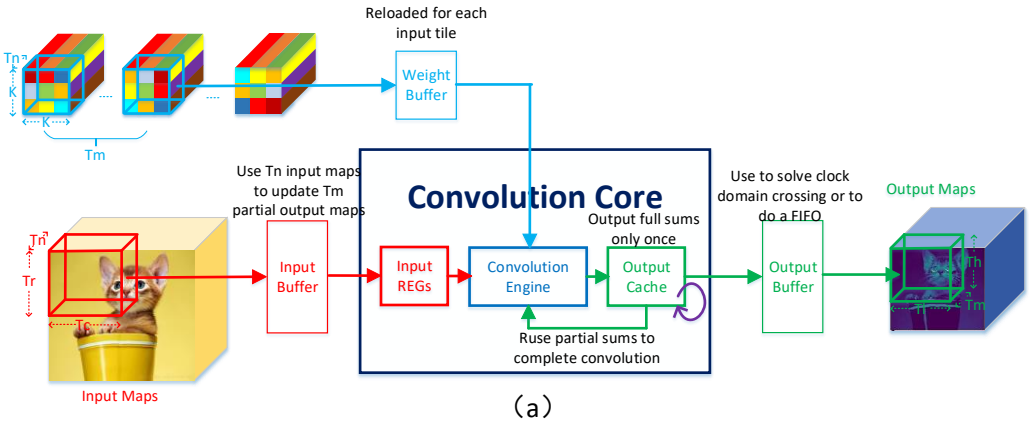
$$\alpha_w = \begin{cases} 1 & N*M*K*K \le B_w \\ \left\lceil \frac{R}{Tr} \right\rceil * \left\lceil \frac{C}{Tc} \right\rceil, & N*M*K*K > B_w \end{cases}$$

其中 Bo、Bw 分别代表 Output Buffer 和 Weight Buffer 的尺寸，意即如果片内 Buffer 能存储整个卷积层所有的权重或部分和，那么将不需再重复地读取权重和暂存部分和在 DDR SDRAM 中。但实际实现时，并不能将片内 Buffer 开那么大以防止资源不足，因此可以默认片外存储与片内 Buffer 的总访问次数相同，则在输出数据和权重数据重利用方式中将不再对片外 DDR SDRAM 的访问次数进行定量分析了。

同理，可以推导出输出数据重利用和权重数据重利用方式下的 α i，α o 和 α w 分别如公式(4-3)和(4-4)，其操作流程图分别如图 1.12(a)和图 1.13(a)所示，与之对应的伪代码描述分别图 1.12(b)和图 1.13(b)所示。

$$\begin{aligned} \alpha_i &= \left\lceil \frac{M}{Tm} \right\rceil \\ \alpha_o &= 1 \\ \alpha_w &= \left\lceil \frac{H}{Th} \right\rceil * \left\lceil \frac{L}{Tl} \right\rceil \end{aligned} \tag{4-3}$$

$$\begin{aligned} \alpha_i &= \left\lceil \frac{M}{Tm} \right\rceil \\ \alpha_o &= 2*\left(\left\lceil \frac{N}{Tn} \right\rceil - 1 \right) \\ \alpha_w &= \left\lceil \frac{R}{Tr} \right\rceil * \left\lceil \frac{C}{Tc} \right\rceil \end{aligned} \tag{4-4}$$

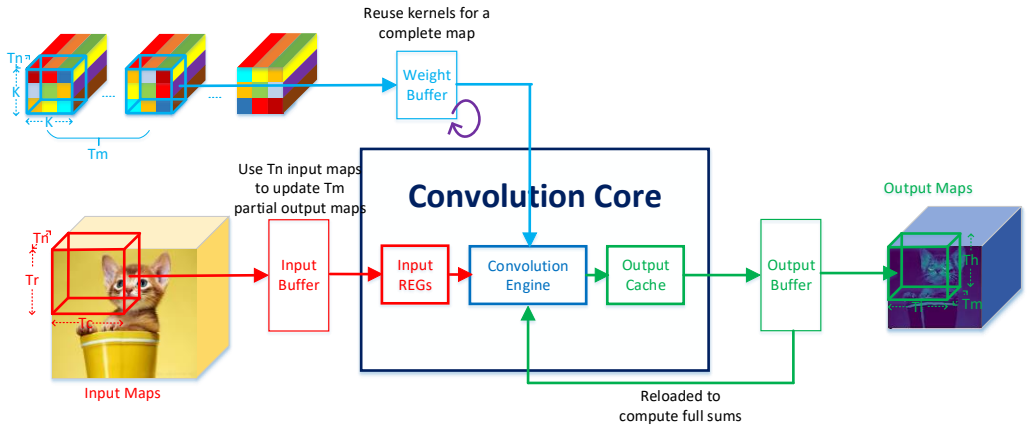


（a）

```
for (int h = 0; h < H; h += Th)//Loop H
  for (int l = 0; l < L; l += Tl)//Loop L
    for (int m = 0; m < M; m += Tm)//Loop M
      for (int n = 0; n < N; n += Tn)//Loop N
        for (int tn = 0; tn < Min(Tn, N - n); tn++)//Loop Tn
          for (int th = 0; th < Min(Th, H - h); th++)//Loop Th
            for (int tl = 0; tl < Min(Tl, L - l); tl++)//Loop Tl
              for (int tm = 0; tm < Min(Tm, M - m); tm++)//Loop Tm
                O[m][r][c] +=//MAC operation
                  ∑_{i=0}^{K-1} ∑_{j=0}^{K-1} W[m][n][i][j] * I[n][h * S + i][l * S + j];
```

（b）

图1.12 输出数据重利用



（a）

```
for (int m = 0; m < M; m += Tm)//Loop M
  for (int n = 0; n < N; n += Tn)//Loop N
    for (int h = 0; h < H; h += Th)//Loop H
      for (int l = 0; l < L; l += Tl)//Loop L
        for (int tn = 0; tn < Min(Tn, N - n); tn++)//Loop Tn
          for (int th = 0; th < Min(Th, H - h); th++)//Loop Th
            for (int tl = 0; tl < Min(Tl, L - l); tl++)//Loop Tl
              for (int tm = 0; tm < Min(Tm, M - m); tm++)//Loop Tm
                O[m][r][c] +=//MAC operation
                  ∑_{i=0}^{K-1} ∑_{j=0}^{K-1} W[m][n][i][j] * I[n][h * S + i][l * S + j];
```

（b）

图1.13 权重数据重利用

可以看出权重及输入数据重利用方式存在输出数据的 FeedBack，而 HLS 不支持在任务级流水中存在数据的 FeedBack，但又想使得数据输入、计算和输出流水起来以便提高处理速度，因此最好选择输出数据重利用方式来保证加入流水机制时能完全使用 HLS 来实现本设计，而不是 HLS 与传统 RTL 代码相结合方式，后者会增加开发难度和时长。

而根据上述理论证明，在使用“SAME”填充方式：Th=Tr，Tl=Tc 的情况下，当三种数据重利用方式的缓冲区尺寸相同时，并假设 N/Tn 不等于 1 的情况下，实际上也不应等于 1 以避免片内对于输入数据的存储过大，权重数据重利用方式的总存储访问次数最多。另外，当选择在输出通道上做并行加速时，Tm 越大加速倍数越大，Tn 越小片内存储越小，可以设计合理的 Tm 和 Tn 使得 M/Tm <N/Tn ，而此时采用输出数据重利用方式将会对存储的总访问次数最少。

另外，缓冲区尺寸在 HLS 中可以参数化以实现数据重利用的可重配置和提高设计的灵活性。

1.2.2 流水卷积电路

在卷积运算单元内部，一种典型的 MAC 电路结构如图 1.14，它有着与输入图像尺寸相同大小的输入寄存器矩阵。其操作流程基于输出数据重利用方式：卷积核按图中箭头方向进行移动，并选择出相应的输入后送入 DSP（乘法器）矩阵和加法器树组成的 MAC 运算单元。然后将其输出结果根据是否是第一个输入通道的相关计算加上对应的偏置或上一次的 MAC 结果，并由写控制逻辑根据是否是最后一个输入通道的计算决定将其暂存在内部寄存器中还是写出到输出 Buffer 中。其中，Tr、Tc 分别表示采用分块机制后一小块输入图像高和宽，假设 Tr=16、Tc=32，K 为卷积核尺寸（长宽均为 K，假设等于 3）。观察发现，该电路结构需要 3×3=9 个(16+(3−1))×(32+(3−1))=612 选 1 的选择器（这里加上(3−1)是因为采用了“SAME”图像填充方式，同样为方便计算和画图示意没有进行均匀填充，而只填充在了图像左侧和下方，实际实现中采用的是均匀填充），每个选择器的输入量和选择器的数量都很多，这会使得电路变得十分拥挤，浪费资源的同时还会带来超高的线延迟，并且当卷积核尺寸变大时会出现高扇出现象（扇出即是与一个逻辑门的输出相连的逻辑器件的数量）。线延迟的增大和超高的扇出都会使最终设计的工作时钟频率提不上去，从而导致处理速度的下降。

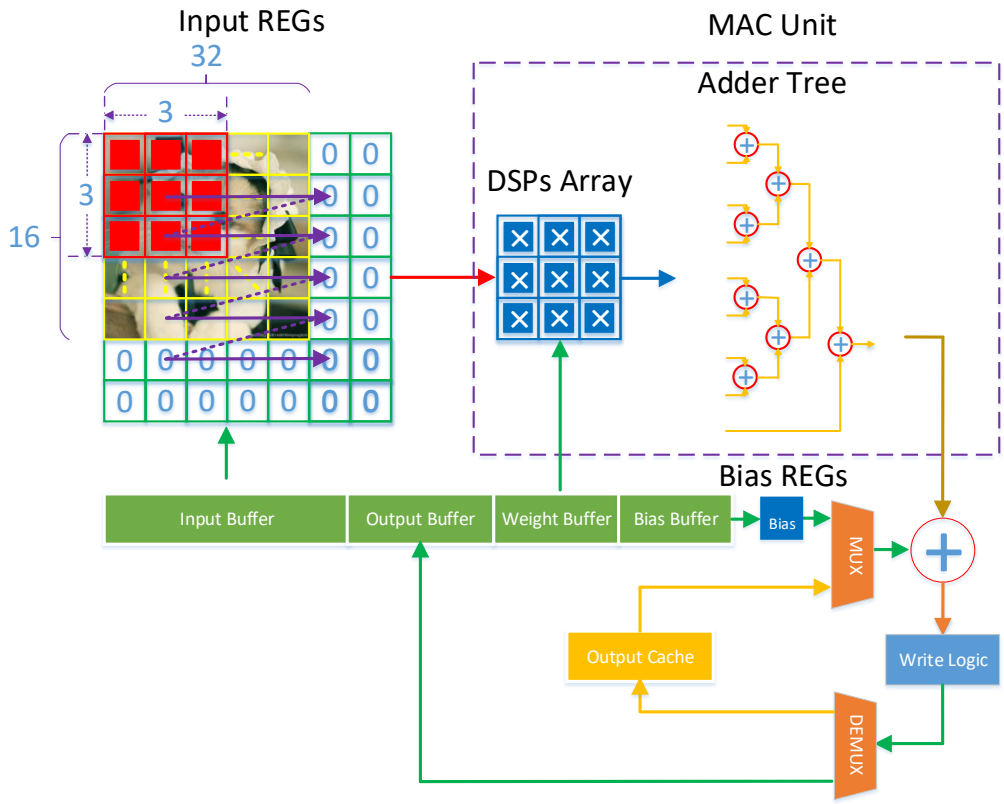


图1.14　典型卷积电路结构

一种方式是改变卷积核的移动方式，如图 1.15，当卷积核一开始从左到右移动到一行的末尾后，并不是像图所示意的那样再从左到右开启新的一行，而是向下平移一行后再从右到左进行移动，如此循环。这样可以使得 3×3 的片选矩阵中只有 3 个寄存器在卷积的一开始和卷积核换行时对输入图像进行选择，而其他 6 个寄存器可以通过向左右或向上平移得到。例如卷积核从 1 位置移动到 2 位置时，片选矩阵的第一行数据由第二行向上平移得到，第二行数据由第三行向上平移得到，而第三行数据通过片选输入图像得到。

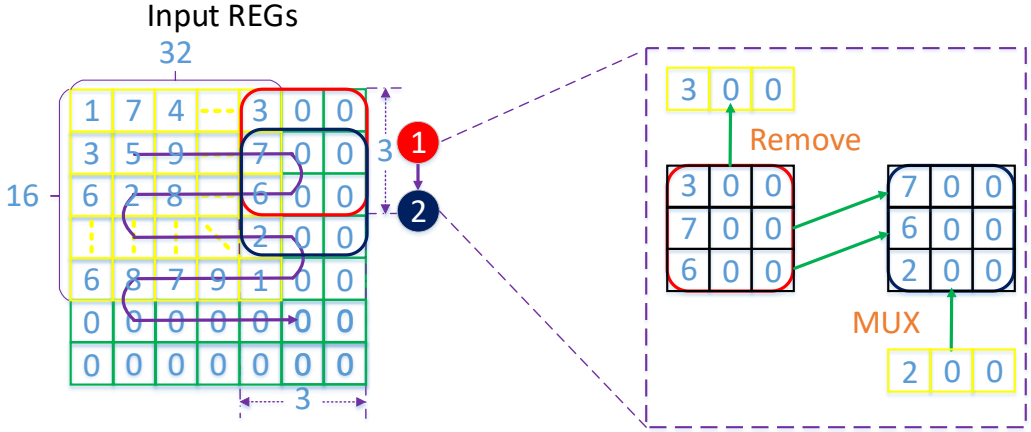


图1.15　改进型卷积电路结构

虽然图 1.15 对图 1.14 进行了改进，将选择器的数量减少为前者的 1/K，但还是存在着大输入量的选择器，另外虽然输入寄存器的数量只和分块后单个通道的输入图像尺寸相同，也还是较大。于是引入如图 1.16 的流水卷积电路结构，该结构与典型的 MAC 电路结构在乘加等运算操作上的处理相同，都用到了加法器树而不是逐一相加来减少总体加法的逻辑时延，但在乘加操作时对输入的获取有着较大的区别。如图所示，以一个长为((Th+(k−1))×(k−1)+ k)的寄存器向量代替寄存器矩阵，先将输入图像数据的前段按顺序存入向量中，以卷积核尺寸和图像宽为基准将其固定位置的数据导入到乘加矩阵中，因此该方式省去了选择器。并且每一时钟周期都将向量中的各元素向右平移一次，并从输入 Buffer 中顺序读入一个图像数据填充在向量的最左端，像流水一样（注意这里是顺序读入输入数据，而不是典型 MAC 电路结构中的片选，原因是输入图像进行填充后在输入 Buffer 中是一行一行有规律地存储的）。为保证 MAC 运算单元能正确获得卷积核尺寸的输入数据，只需((Th+(k−1))×(k−1)+ k)个寄存器，相比典型 MAC 电路结构节约了大量的寄存器，而且操作简单，节省了一些控制逻辑。只是该电路结构在最开始准备数据时有着要比典型 MAC 电路结构更长的等待时间，但当分块的尺寸稍大时可以忽略。

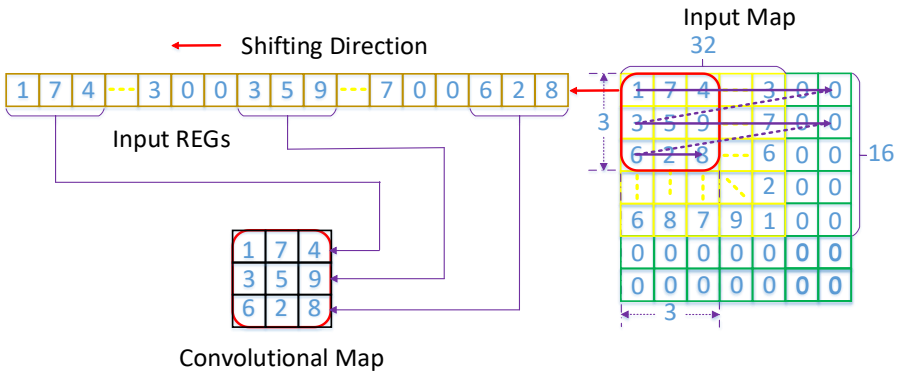


图1.16　流水卷积电路结构

1.2.3　深度流水并行架构

在并行方面，一处便是在上述流水卷积电路中，一个卷积核中所有权重和输入的乘法运算是并行的，而对它们的相乘结果加法操作是流水，因此可以带来卷积核尺寸 K×K 的提速。这里，并行在 HLS 中是通过 UNROLL 指令实现的。另外一处便是在卷积运算内部，有着在输出通道上的并行操作，如图 1.17 所示，当读入一副输入图像，将会加载相应的 Tm 个卷积核，并同时计算 Tm 个输出特征图上的像素值。如此

将会提升分块后输出图像通道数量 Tm 倍的速度。在此过程中将有 Tm 个独立的权重 Buffer 以便能够提升带宽，使得流水卷积电路能够及时的得到想要的数据，防止因数据读取速度而限制了整体处理速度。在 HLS 中，可以对权重矩阵在输出通道维度上进行 PARTITION 命令，便可以将其划分成 Tm 个独立的缓冲队列，并且使用 STREAM 指令将这些队列映射为 FPGA 中的 FIFO（First Input First Output，先入先出）memories。同时，各种数据从 Buffer 加载到内部寄存器的过程也是互相独立的、并行的并不是顺序执行的，比如加载完输入再加载权重。

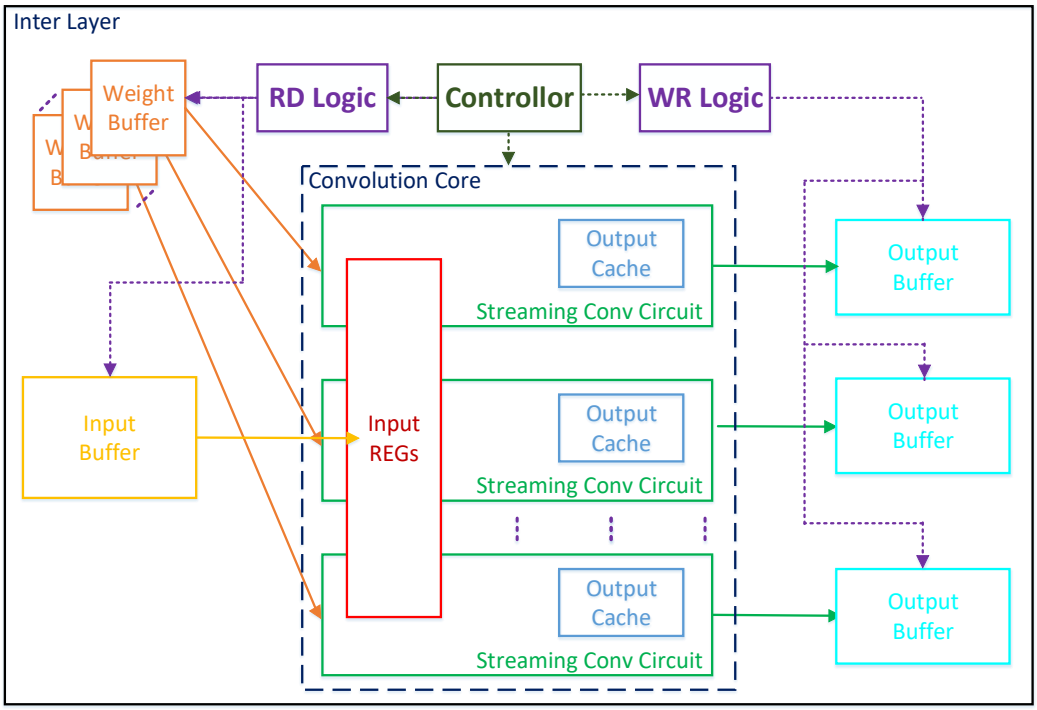


图1.17　卷积内部并行结构

而在流水设计方面，同样有着两处：1）在高层上，采用网络各层流水的方式，如图 1.18。以一层标准卷积、一层深度可分离卷积、一层平均池化以及一层全连接为例，可见当 separate_conv 层处理第一幅输入图像时，standard_conv 层便可以同时处理第二幅图像，如此可以提高层数倍的处理速度。另外，由于标准卷积和可分离卷积操作相似，只是后者的计算量较低，所以二者的处理时间相同。而池化只是求均值，操作简单所以时间相对较短，但全连接短的原因是经过池化后输入图像的尺寸非常小。因此，标准卷积或可分离卷积的时间才能代表一副图像的处理速度。同时，可以发现，当池化处理完当前图像后并没有立即开始下一幅图的处理而是在等待，等待上一卷积层对下一副图的处理结果出来后池化才进行下一步，保证了操作时数据的正确性。而促使这一功能的实现得益于如图 1.19 中的 HLS 代码，这段代码描述了只有当前一任务完成并将所有数据存入前端缓冲，而且后端缓冲有空间可以存储本层计算将要产生

的所有数据时，才开始本层的计算。因此，要想实现流水，两层中间的缓冲必须能存下两倍的前一层网络所能产生的所有数据。另外，在HLS中，流水机制是靠DTAFLOW命令做到的。2）另一处便是在一层网络内部，数据输入、运算（卷积或求均值等）和输出之间的流水，其处理和编码方式与1）相同，因此，相比三者顺序执行来说又提高了将近3倍的速度。

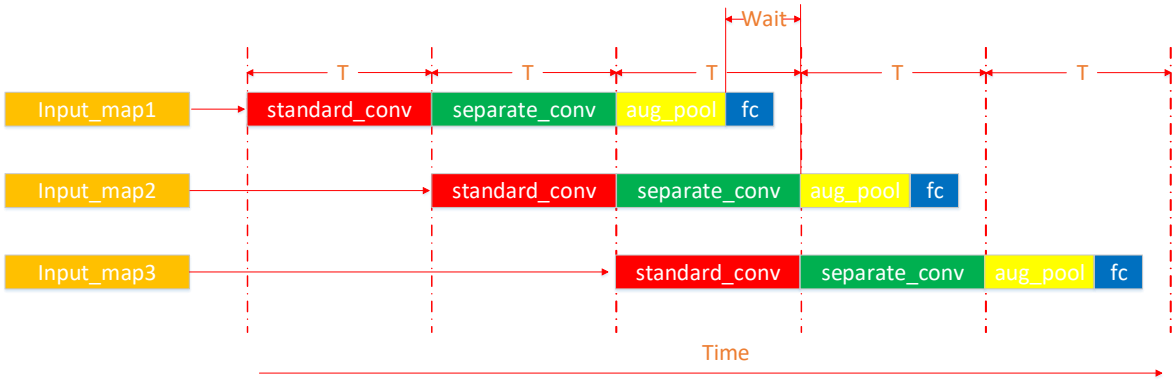


图1.18　层与层的流水

```
/*********************************************************************/
//parameter: task_cntrlfpre,task_cntrl2nxt
//implementation: The parameter task_cntrlfpre actually represents
//        a FIFO, and when the previous task is completed, the value
//        of end_flag is written to the FIFO. The value is 1 when
//        processing the last image, 0 otherwise. The parameter
//        task_cntrl2nxt is similar to task_cntrlfpre.
/*********************************************************************/
void task(stream<bool>& task_cntrlfpre, stream<bool>& task_cntrl2nxt){
        bool end_flag;
        while(1){
                while(1)

        if((!task_cntrlfpre.empty())&&(!task_cntrl2nxt.full()))
                                        break;

                /*the calculations of this task, thus omitted*/

                task_cntrlfpre.read_nb(end_flag);
                task_cntrl2nxt.write_nb(end_flag);
                if(end_flag==1)
                        break;
        }
}
```

图1.19　用来实现流水功能的简单HLS代码

1.2.4　模板化设计

为充分发挥在使用 HLS 基于 C++语言进行开发时的灵活性，采用模板的方式编写函数，如下所示函数模板，可以灵活的设置CONVERSION：卷积类型（标准卷积

或深度可分离卷积），N，M，R，C：卷积总输入通道数、输出通道数、图像输入长、宽，Tn、Tm、Tr、Tc：分块后输入通道数、输出通道数、图像输入长、宽，而 M_size: =R*C，K_size: =K*K。

```
template<typename CONVERSION, int N, int M, int R, int C, int Tn, int Tm, int Tr, int Tc, int K,
int M_size, int K_size, int S>
void conv(dtype *inputs, dtype *weights, dtype *betas, dtype *outputs, stream<bool>& cntl)
{
        kernel_conv<CONVERSION> conv_opr;
        conv_opr.template apply<N, M, R, C, Tn, Tm, Tr, Tc, K, M_size, K_size, S>(inputs, weights, betas, outputs, cntl);
}
```

图1.20　基于 C++的 HLS 函数模板

并汇集成 HLS 库，包含标准卷积、深度可分离卷积、ReLU、AvgPool 及全连接等深度卷积网络所需的函数，比便后续研究人员调用。这个用 C++编写的 HLS 函数库不仅仅集成了第二章节的减少参数量的方法，还与普通的 C++函数库不同，它结合了 HLS 的 pragma 命令参数，其生成出来的 Verilog/VHDL 的 RTL 代码实现了本章节分块、流水及并行的硬件思想。另外，规范了输入输出接口和编程方式，如下图 1.21。该函数使用 while 实现了流水，使用 pingpang 标志实现了缓冲区的乒乓机制，使得后续研究人员可以不用管外部数据的读取方式，直接将自己的想法参考现有函数中的编程方式编写成代码后，封装成一个函数替换图中的 conv 即可。这里参考现有函数中的编程方式的原因是，本章在介绍 HLS 时说到因每个人编程方式的不同，即不同的 C 代码，HLS 生成出来的 RTL 代码不同，在处理速度和资源消耗上将会有很大的差距，所以在 HLS 中 C 代码的编写还是需要较强的硬件功底。本人因有着较多的使用传统 RTL 设计的 FPGA 项目经历，认为自己的硬件功底还是可以的，而且在 HLS 上的探索已有一年，对这几个函数的优化还是有自信的。当然也更希望后来者更够将其更好的优化，那必然是欣喜的，本人只是为硬件功底欠佳和 HLS 使用不熟练的人提供一个参考。如此，使得该函数库可以很方便且有效的得到扩充。而且这只是一个开端，如同一个初生的婴儿一样，并希望它能够得到茁壮的成长，函数越来越多，优化越来越大。如果可以的话，最终实现以 HLS 为依托将 Python 代码直接实现为硬件描述语言，让在 TensorFlow 平台上使用 Python 开发的深度学习网络能像使用 GPU 那样方便的使用 FPGA 进行加速。

```
template<int N, int M, int R, int C, int Tn, int Tm, int Tr, int Tc, int K,
int M_size, int K_size, int S>
void inter_layer(dtype *inputs, dtype *weights, dtype *betas, dtype *outputs, int inputs_ofst,
                int outputs_ofst, stream<bool>& pre_ready, stream<bool>& nxt_ready)
{
        bool progress_flag;
        bool pingpang_flag = 0;
        bool end_flag;
        stream<bool> cntl;
#pragma HLS STREAM variable=cntl depth=1
      while(1)
        {
                while(1)
                        if((!pre_ready.empty())&&(!nxt_ready.full()))
                                break;

                conv<SEPARABLE_CONV, N, M, R, C, Tn, Tm, Tr, Tc, K, M_size, K_size, S>(&inputs[pingpang_flag * inputs_ofst],
                                weights, betas, &outputs[pingpang_flag * outputs_ofst], cntl);

                pingpang_flag = !pingpang_flag;

                if (!cntl.empty())
                {
                        cntl.read_nb(end_flag);
                        pre_ready.read_nb(progress_flag);
                        nxt_ready.write_nb(progress_flag);
                }
                if (progress_flag == 0)
                        return;
        }
}
```

图1.21　定义好的输入输出接口 HLS 代码

为了使后续研究人员不仅仅局限于本文所提出的 EfficientNet，能够根据自身对资源、速度和功耗的需求，来调用本文依据诸多优化方法所编写的 HLS 库并设置参数，来实现适用于自己的网络。本文还提出了一种设计空间探索模型 Design Space Exploration，给出了在给定各参数如输入输出特征图尺寸的情况下，一层标准卷积的资源、功耗及时间的计算公式，如表 1.4 所示（这里，仅给出标准卷积相关计算公式的原因是：当使用第三章所提的以平均池化代替全连接的方法后，网络中的计算量和参数量集中在卷积层，且本章深度流水并行架构中提到了网络的处理时间也取决于标准卷积，而其他层对网络性能的影响不大）。

假设该卷积层总的输入特征图尺寸为 $R\times C\times N$、输出特征图尺寸为 $H\times L\times M$，分块后输入特征图尺寸为 $Tr\times Tc\times Tn$、输出特征图尺寸为 $Th\times Tl\times Tm$，卷积核的尺寸为 $K\times K$、步进为 S，填充方式为“SAME”，工作频率为 f，每一输入像素的处理时延为 t，每一次存储访问操作产生的功耗为 E_{memory}，每一次乘加操作产生的功耗为 $E_{operation}$（其中 R 和 C 分别是总的输入特征图的高和宽，N 是总的输入通道数，H 和 L 分别是总的输出特征图的高和宽，M 是总的输出通道数，Tr 和 Tc 分别是分块后输入特征图的高和宽，Tn 是分块后输入通道数，Th 和 Tl 分别是分块后输出特征图的高和宽，Tm 是分块后输出通道数，K 是正方形卷积核的边长）。

表1.4 一层标准卷积的资源、功耗及速度计算

Time (s)	$\frac{R\times C\times N\times M}{Tm}\times\frac{R+t-1}{R}\times\frac{1}{f}$	
Power	$2\times\left(R\times C\times N\times\left\lceil\frac{M}{Tm}\right\rceil+H\times L\times M+K\times K\times M\times N\times\left\lceil\frac{H}{Th}\right\rceil\times\left\lceil\frac{L}{Tl}\right\rceil\right)\times E_{memory}$ $+\frac{R\times C\times K\times K\times M\times N}{S^2}\times E_{operation}$	
DSP	$K\times K\times Tm\times 4$	
BRAM (18KB)	Input Buffer	$\frac{2\times Tn\times(Tr+K-1)\times(Tc+K-1)}{1024}\times\left\lceil\frac{16}{18}\right\rceil$
	Weight Buffer	$\frac{2\times Tn\times Tm\times K\times K}{1024}\times\left\lceil\frac{16}{18}\right\rceil$
	Output Buffer	$\frac{2\times Tm\times Th\times Tl}{1024}\times\left\lceil\frac{16}{18}\right\rceil$

这里在 Power 的计算公式中乘 2 的原因是在本文硬件设计中存在着片内 Buffer 和片外 DDR SDRAM 两处存储访问。而在 DSP 的计算公式中乘 4 是因为本文硬件实现采用的数据类型是位宽为 16bit 的半精度浮点数，每两个半精度浮点数相乘需消耗 4 个 DSP。另外，为保证流水机制能被有效地执行，两相邻任务之间的缓冲必须能存下两倍的上一任务所产生的所有数据，因此在 BRAM 的计算公式中需乘 2。需要注意的是，Time 的计算公式忽略了处理过程中控制逻辑所消耗的时间。

1.3 调试所遇问题及其解决方案

1.3.1 HLS 对不同的 C 代码风格理解不一

因编码方式、代码风格、定义限制的不同，HLS 生成的 VHDL/Verilog 代码也会各不相同，其不仅仅在效率上有较大的差别，甚至有的还会影响最终硬件设计的正确性（即使它在 C 环境下运行正确，但毕竟使用的是串行指令流，而在 FPGA 中使用的却是并行思想）。因此，由于 HLS 还不够智能和成熟，目前并不能完全不管底层硬件实现细节直接让纯软件工程师上手做出一个性能良好功能正确的 FPGA 算法加速器，还是需要一定的硬件功底，以便清楚的了解哪里需要缓冲或等待来保证数据的正确性。本人在使用 HLS 开发本文设计时，遇到的这种因 C 代码风格不一 HLS 理解不一的情况有很多，下面就以在卷积运算和流水机制中遇到的对本文设计影响较大的两个为例，阐述这一现象。

(1). 当卷积步进大于 1 时，为了减少计算所产生的功耗，应使得卷积计算有条件

的进行。于是便设计了如图 1.22 左方所示的代码，但其由 HLS 生成的硬件代码综合出来的电路却不是预想的那样。以其中一个 DSP 为例，如图 1.23 左方所示，变成了对 DSP 输出结果的有条件选择（其中 tr、tc 表示当前处理像素所在当前分块的行和列，S 是步进，假设大于 1）。当输入图像像素各不相同时，则对于该电路结构，即使卷积步进比 1 大，每个周期 DSP 的输入也不相同，即 DSP 内部始终有着电平的反转，便会一直产生着动态功耗，从而违背了设计的初衷。而对其做了如图 1.22 右方所示的改进，使得对输出结果的有条件选择变为对输入的有条件的选择，其电路结构如图 1.23 右方所示。如此，在图像的前(K−1)行时，并不能满足((tr%S == S−1) && (tc%S == S−1))的条件，DSP 的输入始终是一样的为零，则在其内部没有电平的反转，便减少了此时的动态功耗。由于这样的计算在本设计中存在量非常巨大，积少成多，采用图中的设计能使得最终的设计在功耗上有着将近 1W 的降低。

```
if ((tr % S == (S - 1)) && (tc % S == (S - 1)))
{
        dtype mac_dat = mac(input_regs, weight_regs);
        /*Subsequent calculations are omitted*/
}
```

```
if ((tr % S == (S - 1)) && (tc % S == (S - 1)))
{
        input_tmp = input_regs;
        weight_tmp = weight_regs;
}
else
{
        input_tmp = (dtype)0;
        weight_tmp = (dtype)0;
}
dtype mac_dat = mac(input_regs, weight_regs);
/*Subsequent calculations are omitted*/
```

图1.22　卷积所遇到的问题及其解决方案

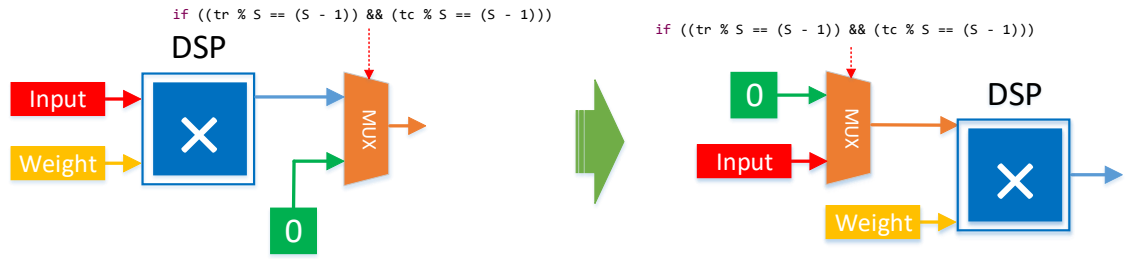


图1.23　DSP 工作时两种不同的电路结构

(2). 再如图 1.24 上方的数据加载程序，缓冲 nxt_ready 设计的目的，是为了当数据全部加载完成后，再在 nxt_ready 中写入一标志，使后端程序判断该 FIFO 是否为空来确认前端数据是否准备好，并根据该标志是否为 1 来判断是终断还是继续程序的执行。这里，只有当数据全部加载完成才写入标志的原因是为了保证在后端程序持续处理一段数据时该数据的正确性，而且在 C 程序中，任务①、②、③、④这样书写的顺序可以正确实现上述功能。但是，当引入 DATAFLOW 命令进行流水操作时，load_data 函数中的任务①、②、③、④都会并行。其中，输入、权重及偏置分别存

储在不同的Buffer中，它们之间的读取并行是可行的而且也是期望的。但是nxt_ready与之并行却违背的设计的初衷，使得数据还未加载完成，后端程序判断nxt_ready不为空认为数据已全部准备好，便开始了计算，而此时的数据是不确定的是错误的。原来在执行DATAFLOW命令时，如果任务之间没有数据的交互便会并行，观察发现任务④确实与其他任务没有任何关系。而当将任务①、②、③INLINE后，HLS无法以函数级判断原来各任务输入输出的关系，这些操作便变成了串行，功能正确。但输入、权重及偏置的读取失去了并行增加了处理时间，因此要想使得几个任务之间能够实现并行，不仅要使用DATAFLOW命令，还得要将各任务封装成函数。而要解决任务④也与之并行的问题便是做如图1.24下方的改进，判断所有任务是否完成再写入标志。具体的做法便是在任务①、②、③中都加入一段逻辑，当任务内操作完成时，像end_flag缓冲中写入结束标志，然后在load_data函数中判断每个任务的结束缓冲是否为空来确认所有任务是否完成。这也是图1.21比图1.19多了一个cntrl FIFO的原因。

```
void load_data(stream<bool>& task_cntrlfpre, stream<bool>& task_cntrl2nxt){
        bool end_flag;
        while(1){
                while(1)

        if((!task_cntrlfpre.empty())&&(!task_cntrl2nxt.full()))
                                        break;

                load_inputs();    1

                load_weights();   2

                load_biases(); 3

                task_cntrlfpre.read_nb(end_flag);
                task_cntrl2nxt.write_nb(end_flag);   4

                if(end_flag==1)
                        break;
        }
}
```

```
void load_data(stream<bool>& task_cntrlfpre, stream<bool>& task_cntrl2nxt){
        bool end_flag;
        stream<bool> input_cntl;
#pragma HLS STREAM variable=input_cntl depth=1
        stream<bool> weight_cntl;
#pragma HLS STREAM variable=weight_cntl depth=1
        stream<bool> bias_cntl;
#pragma HLS STREAM variable=bias_cntl depth=1
        bool input_end_flag;
        bool weight_end_flag;
        bool bias_end_flag;
        while(1){
                while(1)
                        if((!task_cntrlfpre.empty())&&(!task_cntrl2nxt.full()))
                                        break;

                load_inputs(input_cntl);    1

                load_weights(weight_cntl);  2

                load_biases(bias_cntl); 3

                if ((!input_cntl.empty())&&(!weight_cntl.empty())&&(!bias_cntl.empty()))
                {
                        input_cntl.read_nb(input_end_flag);
                        weight_cntl.read_nb(weight_end_flag);
                        bias_cntl.read_nb(bias_end_flag);
                        task_cntrlfpre.read_nb(end_flag);
                        task_cntrl2nxt.write_nb(end_flag);   4
                }

                if(end_flag==1)
                        break;
        }
}
```

图1.24　流水所遇到的问题及其解决方案

1.3.2　浮点数的截断误差

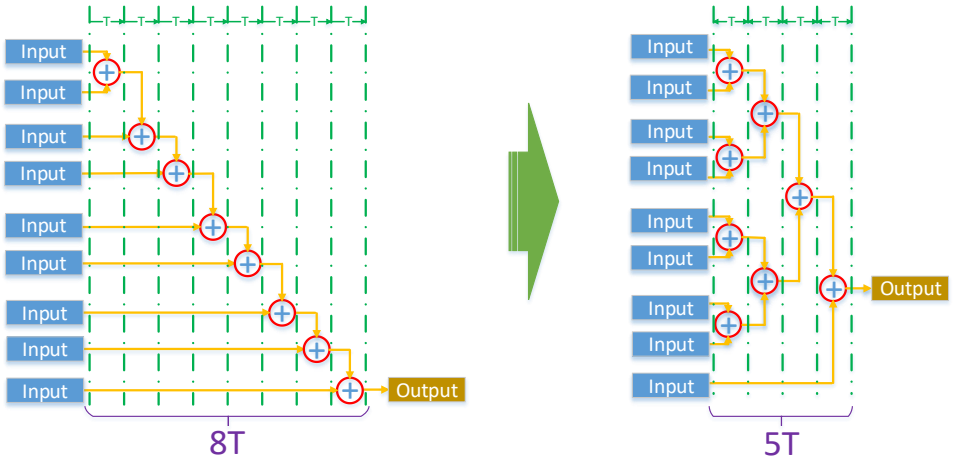
如图 1.25 所示的代码，在 C 程序中，根据 for 循环的执行顺序，sum 是逐一与 mul_dat 矩阵的各元素相加的，反映到底层硬件实现细节便是图 1.26 左方所示。而由 HLS 生成 VHDL/Verilog 代码后综合而成的电路却是加法树，如图 1.26 右方所示，以此减少处理延迟。因此导致了 C 程序和硬件程序对 mul_dat 矩阵各元素的相加顺序不一，再加上截断误差这一浮点数特有的毛病，使得 C 程序和硬件程序在其他操作均相同的情况下，输出结果相似却不一致，这曾困扰本人很长一段时间，以为自己设计不正确所导致，一层一层一步一步一根信号一根信号地排查了好久才发现。因此，想要 C 程序结果与硬件输出一致，就得改变 C 程序相加顺序使其和硬件一致即可。另外，基于 TensorFlow 平台开发的 Python 程序与 C 程序运行结果也不一样但是相似，

原因也是 TensorFlow 对代码在底层硬件实现细节上做了优化，其对浮点数的处理顺序与使用同样 for 循环的 C 程序并不一样，这也是使用 TensorFlow 开发的网络比从卷积到数据的读取都是自己用 C 编写的网络来说，运行速度要快的原因。

```
dtype output = (dtype)0;

for (int i = 0; i < 3; i++)
#pragma HLS UNROLL
        for (int j = 0; j < 3; j++)
#pragma HLS UNROLL
                output += inputs[i *3 + j];
```

图1.25　浮点累加的 HLS 代码



（a）C ACC　　（b）HLS ACC

图1.26　HLS 对浮点累加的优化

1.3.3　如何提高系统工作频率

在使用 HLS 进行硬件开发时，可以将某一大数据量并行的封装成两个函数，使得 HLS 将其认为是两个独立的任务，会产生两份驱动，从而减少高扇出。还可以在生成代码和实现的过程中，将设计的时钟约束的比期望的小一些，如此 HLS 会根据逻辑器件的逻辑延迟适当的在一段逻辑中插入寄存器来提高时钟频率，而实现则是通过将关系紧密的逻辑单元摆放在相近的地方以此减少线延迟从而提高时钟频率。另外还在实现过程中，还可以在实现设置中勾选上功耗、布局布线优化等选项，让工具自动地优化，其优化的方式便是复制寄存器，拉近距离等。

1.3.4 HLS 不支持任务级流水中的 FeedBack

当任务级的流水中出现数据的 FeedBack 时，HLS 便会中断流水机制。解决方式之一便是使用两个独立的变量代表出现 FeedBack 的这一数据的输入输出，使得 HLS 认为这是两个不同的区域，从而绕过了 FeedBack 而可以进行代码的转换。但是在 HLS IP （Intellectual Property，知识产权）核外部，这两个接口是连接在同一区域上，而完成这一连接任务便需要使用传统的 RTL 代码来完成。而 HLS 一次只能对一个 C 函数进行转换，并生成相应的 IP 核，但一个网络有着诸多的层数，当选择输入或权重数据重利用方式时，每一层网络中都存在输出数据的 FeedBack，则每一层为绕过 FeedBack 都要拆成两个函数来使用 HLS 分别封装成 IP，再在外部用传统的 RTL 将其连接，工程量太大，而且一旦出错，调试会更加困难。因此想要缩短开发周期，最好将全部操作都基于 HLS 来实现，所以需得选择输出数据重利用方式，并设计合理的尺寸使得选用此方式时对存储的访问次数最低。