高技术通讯 2023 年 第33 卷 第8 期:823-835

doi:10.3772/j.issn.1002-0470.2023.08.005

基于层间融合的神经网络访存密集型层加速①

杨 灿② 王重熙 章隆兵③

(处理器芯片国家重点实验室(中国科学院计算技术研究所) 北京 100190)

(中国科学院计算技术研究所 北京 100190)

(中国科学院大学 北京 100049)

摘 要 近年来,随着深度神经网络在各领域的广泛应用,针对不同的应用场景,都需要对神经网络模型进行训练以获得更优的参数,于是对训练速度的需求不断提升。然而,现有的研究通常只关注了计算密集型层的加速,忽略了访存密集型层的加速。访存密集型层的操作主要由访存带宽决定执行效率,单独提升运算速度对性能影响不大。本文从执行顺序的角度出发,提出了将访存密集型层与其前后的计算密集型层融合为一个新层执行的方式,将访存密集型层的操作作为对融合新层中输入数据的前处理或输出数据的后处理进行,大幅减少了访存密集型层在训练过程中对片外内存的访问,提升了性能;并针对该融合执行方案,设计实现了一个面向训练的加速器,采用了暂存前处理结果、后处理操作与计算密集型层操作并行执行的优化策略,进一步提升了融合新层的训练性能。实验结果显示,在面积增加6.4%、功耗增加10.3%的开销下,训练的前向阶段、反向阶段的性能分别实现了67.7%、77.6%的提升。

关键词 神经网络;训练;加速器;卷积神经网络(CNN);访存密集型层;批归一化(BN)层

0 引 言

近年来,深度神经网络在图像、语音、自然语言处理等领域有了广泛而深入的应用,针对不同的应用场景,都需要对神经网络模型进行训练以获得更优的参数,于是对训练速度的需求也不断提升,相应地,学术界和产业界也都将目光投向了神经网络模型的训练过程,提出了各类方法、各种加速器结构来实现对训练过程的加速。Google 推出了新一代加速器 TPUv3[1],支持多类深度学习应用的训练。NVIDIA 在最新的 A100 图形处理器(graphics processing unit, GPU)中进一步升级了专门用来加速深度学习任务的 Tensor 核心。华为的 Ascend[2] 采用了 3D Cube 单元实现对卷积和矩阵乘法的加速。Cambricon-Q[3]通过由专用的集成电路(application specific integrated circuit, ASIC)加速核心与近数据处理引擎的混合结构实现高效量化训练的方法。RaPiD[4]使用支持超低精度训练的加速器来提高训练的性能。Sigma[5]设计了一种新的约简树(reduction tree)结构来实现对稀疏、不规则的矩阵乘法的加速。

上述针对训练的加速器结构通常只关注了计算密集型层——例如卷积层、全连接层的加速,且通常将它们的操作化为矩阵乘法运算进行,并未涉及到神经网络模型中的其他访存密集型层——例如批归一化(batch normalization, BN)层、非线性激活层的加速。文献[6]提出了针对批归一化层、缩放平移

① 中国科学院战略性先导科技专项(XDC05020100)资助项目。

② 女,1992 年生,博士生;研究方向:计算机系统结构,加速器设计;E-mail:yangcan@ict.ac.cn。

③ 通信作者,E-mail:lbzhang@ict.ac.cn。

(收稿日期:2022-02-23)

层的推理阶段的加速方案,通过将其操作化为加法与乘法,映射到专门的运算单元中直接进行计算来提升性能。文献[7]提出了在推理过程中将批归一化层、缩放平移层、ReLU 层与卷积层合并处理的方案,采用对卷积层输出数据马上进行批归一化、缩放平移、ReLU 等运算来提高数据处理的效率。由于推理过程中批归一化操作所需的每个通道的均值与方差是已知的常数,于是批归一化操作退化为缩放平移操作,能够采用文献[6,7]的方案进行加速。然而,在训练过程中,批归一化层的均值和方差需要由这次迭代中所有样本的数据现场算出,不能直接退化为缩放平移操作,于是,文献[6,7]的方案完全不适用。文献[8]提出了一个通过将元素和与元素平方和化为矩阵运算的方法来加速批归一化层的方案,但在实际处理中,批归一化层作为访存密集型层,主要由访存带宽决定了执行效率,单独提升运算速度对提升整体的执行效率意义不大。

针对以上问题,本文首先确定了访存密集型层——批归一化层、加法层、非线性激活层是卷积神经网络模型中重要的组成部分,分析了它们的常见连接以及它们在训练的前向过程与反向过程中的计算特征后,提出了在训练的各阶段中,将访存密集型层与其前后的计算密集型层融合为一个新层执行的方式。访存密集型层的操作作为对融合新层中输入数据的前处理或者原始输出数据的后处理进行,大大减少了访存密集型层执行过程中与片外内存交互的数据量以及训练过程中需存储在片外内存中的数据量。在此基础上,本文设计实现了一个新的面向训练的加速器,能够高效地支持融合新层的前向过程与反向过程的执行,并采用了暂存前处理结果、后处理操作与计算密集层的运算操作并行执行的优化策略,进一步提升了融合新层的性能。实验结果显示,本文提出的融合方案与加速器结构在面积增加 6.4%、功耗增加 10.3% 的开销下分别实现了对训练的前向阶段、反向阶段 67.7%、77.6% 的性能提升。

1 卷积神经网络模型特性

1.1 卷积神经网络的常用层

在卷积神经网络(convolutional neural network, CNN)模型中,常用的层包括了卷积层(convolutional layer,CONV)、池化层、全连接层和非线性层(nonlinear layer, NonL)。随着 CNN 在算法上的发展,CNN 模型中还出现了另外两类常用的层。

第 1 类是批归一化(BN)层。在训练的过程中,网络的参数不断变化导致了网络中激活值的分布不断发生变化,于是需要更低的学习速率和更精细的参数初始化来保证训练的精度以及收敛性,因而降低了训练的速度。为了解决这个问题,Google 提出了批归一化[9]的方法,通过对每层的输入进行归一化,固定住每层输入的均值和方差,减少内部协变量偏移(internal covariate shift),使得训练过程能够使用更高的学习速率,从而加速网络的训练。常用的 CNN 模型中都包含了 BN 层,主要的层间连接方式有 CONV→BN→NonL 和 CONV→BN。

第 2 类是捷径连接(shortcut connections)与堆积层(stacked layers)的结果累加形成的累加层(Add)。文献[10]提出了一类残差网络模型,设计了“捷径连接”来实现需要的映射。捷径连接执行恒等映射,并将其结果与堆积层的结果累加起来,如图 1 所示。该类捷径连接与累加结构获得了巨大的成功,在后续多种常用的 CNN 模型中出现,如 ResNext[11]、Wide ResNet[12]、MobileNet v2[13]等;并且成为神经架构搜索中的网络模型的基本组成部件之一,出现在了 MobileNet v3[14]、MNASNet[15]、RegNet[16]等轻量级模型中。于是,捷径连接与堆积层的结果的累加层也成为了 CNN 模型中的一个常用层。

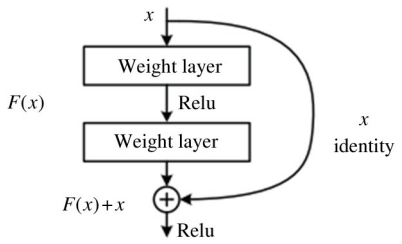

图 1 残差学习块[11]

1.2 训练过程中的计算特征

深度神经网络的训练过程一般分为 2 个部分,即前向传播(forward propagation, FP)和反向传播(backward propagation,BP)。在 FP 过程中,将 *mini-*

batch 个样本作为网络模型的输入,按照网络的顺序从前至后依次进行计算,得到该网络模型的输出结果。接着,按照训练方法中规定的误差计算方式,计算出这批样本的误差值,然后开始 BP 过程。在 BP 过程中,按照链式法则将误差在网络模型中由后向前传播,并利用每层的特征图和误差值计算出权值梯度。最后,根据权值梯度和训练算法中给定的更新权值的方法对权值进行更新后,开始下一批样本的迭代。

使用 GPU(NVIDIA GeForce RTX 2080 Ti)执行了多个常用 CNN 模型的训练过程,用 PyTorch Profiler 测量了每个层在 FP 和 BP 阶段的运行时间。图 2 中给出了各模型的 BN、NonL 和 Add 层在 FP 和 BP 过程中的执行时间,执行时间分别按照 FP 和 BP 过程中 CONV 层的执行时间进行了归一化。从图 2 中可以看出,FP 过程中 BN、NonL 和 Add 的平均执行时间分别是 0.31、0.19 和 0.1,总共达到了 0.6;BP 过程中 BN、NonL 和 Add 的平均执行时间分别是 0.46、0.25 和 0.09,总共达到了 0.8。其中,ResNet50 的 FP、BP 过程,BN、NonL、Add 的执行时间总和分别是 0.85 和 1.11。以上观察说明,除了 CONV 层,BN 层、NonL 层和 Add 层的执行时间在 CNN 的训练过程中也占了很大的比重,对它们进行加速可以有效地缩短训练的时间。

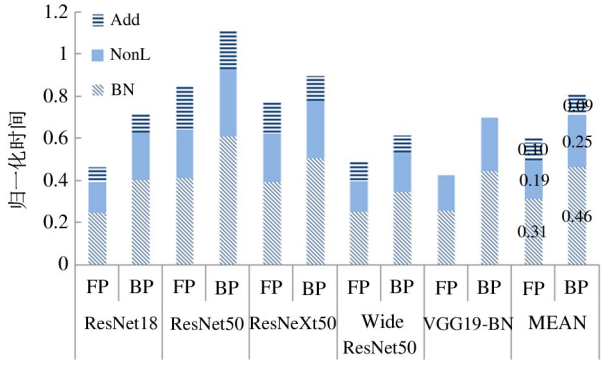

图 2　FP、BP 中 BN、NonL、Add 层的执行时间

1.2.1　BN 层的计算过程

$\{x_i, i = 1, \cdots, m\}$ 是本次训练迭代中 BN 层的 *mini-batch* 个样本的同一个通道上的 m 个输入值,$\{y_i\}$ 是 BN 层的输出结果。BN 层的 FP 过程[10],首先计算出 $\{x_i\}$ 的均值 μ_B 和方差 σ_B^2:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{1}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \tag{2}$$

然后,利用均值和方差对每个输入值进行归一化得到 $\{\hat{x}_i\}$,ε 是训练迭代中使用的极小量:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} \tag{3}$$

最后,使用 BN 层给定的参数 γ 和 β,对 $\{\hat{x}_i\}$ 进行线性变换操作得到 BN 层的输出结果 $\{y_i\}$:

$$y_i = \gamma\hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i) \tag{4}$$

执行 BN 层的 FP 过程时,需要将 BN 层的输入值从片外内存中读到片上,计算出均值和方差。由于片上的缓存区容量有限,在放不下 *mini-batch* 个样本的输入值时,执行式(3)、(4)中的计算需要从片外再读 1 次 BN 层的输入值到片上,才能计算出 BN 层的结果并存回片外内存中,供下一层使用。因此,访存量是 BN 层输入值的 3 倍。

文献[10]给出了 BN 层进行反向误差传播的计算公式:

$$\frac{\partial l}{\partial \hat{x}_i} = \frac{\partial l}{\partial y_i} \times \gamma \tag{5}$$

$$\frac{\partial l}{\partial \sigma_B^2} = \sum_{i=1}^{m}\frac{\partial l}{\partial \hat{x}_i} \times (x_i - \mu_B) \times \frac{-1}{2}(\sigma_B^2 + \varepsilon)^{-\frac{3}{2}} \tag{6}$$

$$\frac{\partial l}{\partial \mu_B} = \sum_{i=1}^{m}\frac{\partial l}{\partial \hat{x}_i} \times \frac{-1}{\sqrt{\sigma_B^2 + \varepsilon}} + \frac{\partial l}{\partial \sigma_B^2} \times \frac{1}{m} \times \sum_{i=1}^{m} -2(x_i - \mu_B) \tag{7}$$

$$\frac{\partial l}{\partial x_i} = \frac{\partial l}{\partial \hat{x}_i} \times \frac{1}{\sqrt{\sigma_B^2 + \varepsilon}} + \frac{\partial l}{\partial \sigma_B^2} \times \frac{2(x_i - \mu_B)}{m} + \frac{\partial l}{\partial \mu_B} \times \frac{1}{m} \tag{8}$$

其中,$\{x_i\}$ 和 $\{y_i\}$ 分别是 BN 层前向计算的输入值和输出值,$\frac{\partial l}{\partial y_i}$ 和 $\frac{\partial l}{\partial x_i}$ 分别是 BN 层反向计算的输入误差值和输出误差值,$\frac{\partial l}{\partial \hat{x}_i}$、$\frac{\partial l}{\partial \sigma_B^2}$ 和 $\frac{\partial l}{\partial \mu_B}$ 都是计算 $\frac{\partial l}{\partial x_i}$ 的过程中用到的中间表达式。在执行时,需要先从片外内存中读 *mini-batch* 个 $\frac{\partial l}{\partial y_i}$ 和 x_i 到片上,计算出

$\frac{\partial l}{\partial \sigma_B^2}$ 和 $\frac{\partial l}{\partial \mu_B}$，由于片上缓存区容量有限，放不下时，执行式(8)中操作需要重新从片外内存中读 $\frac{\partial l}{\partial y_i}$ 和 x_i 到片上，算出 $\frac{\partial l}{\partial x_i}$ 并写回片外内存中。因此，其访存量是 BN 层输入值的 5 倍。

1.2.2 NonL 和 Add 层的计算过程

NonL 层最常用的激活函数是 ReLU。NonL 层的 FP 从片外内存中读入前一层的结果到片上，进行规定的激活操作后，将结果存回到片外，共 2 倍输入值的访存量；NonL 层的 BP 从片外读入输入误差值和 FP 过程的输入特征图到片上，进行计算后存回片外，共 3 倍输入值的访存量。

Add 层的 FP 和 BP 过程相同，都是从片外读入 2 个需要累加的输入值，进行加法操作后，将结果存回片外，共 3 倍输入值的访存量。

BN、NonL 和 Add 层在 FP 和 BP 过程的操作都是访存密集型的，计算过程中对运算单元的利用率极低。

2 加速访存密集型层的方案

为了实现对 BN、NonL、Add 层的加速，本节提出了一个将访存密集型层与计算密集型层融合执行的方式，能够减少运行过程中与片外内存交互的数据量，从而提升性能。

2.1 NonL、Add 层的融合方案

不管是 FP 还是 BP，NonL、Add 层的输出结果仅需同一个样本中与结果同一位置的输入作源操作数即可算得。于是，有 2 种直观的融合方式：一是可以将其操作直接与其后执行的相邻层进行融合，读入 NonL 或 Add 层的输入值后，立即算出结果作为其后相邻层的输入值，接着进行其后相邻层的计算。对于融合后的新层，以其后相邻层的计算为主，NonL 和 Add 层的操作相当于对融合新层的输入值的一个“前处理”。二是可以将其操作与执行顺序的前一层进行融合，相当于对前一层的输出结果增加了一个“后处理”，成为融合新层的新输出结果。

两种方式下，都能够直接减少 NonL 和 Add 层与片外内存的交互数据量；并且，只需要将融合新层的输出值存在片外，内存中不需要存融合新层中内部层的结果，减少了训练过程中对内存容量的需求。

2.2 BN 层融合方案

与 NonL、Add 层不同，BN 层的一个结果与这次训练过程中使用的 *mini-batch* 个样本的该通道上的所有源操作数都相关，不能直接与其前后相邻层融合。将式(3)带入式(4)中，可以得到：

$$y_i = F_1 \times x_i + F_2 \tag{9}$$

其中，F_1 和 F_2 是关于 BN 层每个通道的参数项，计算方式为

$$F_1 = \frac{\gamma}{\sqrt{\sigma_B^2 + \varepsilon}} \tag{10}$$

$$F_2 = \beta - \mu_B + F_1 \tag{11}$$

由式(9)可以看出，在算出关于每个通道的参数项 F_1 和 F_2 后，BN 层的 FP 就化为了仅与同一样本的同一位置的源操作数相关的计算。

将式(5)～(7)带入式(8)中，可以得到：

$$\frac{\partial l}{\partial x_i} = F_1 \times \frac{\partial l}{\partial y_i} + B \times x_i + C \tag{12}$$

其中，F_1 和 B、C 都是关于 BN 层每个通道的参数项，B 和 C 的计算方式为

$$B = F_1 \times F_3 \times (\mu_B \times S_1 - S_2) \tag{13}$$

$$C = -\mu_B \times B - F_1 \times S_1 \tag{14}$$

其中，

$$F_3 = \frac{1}{\sigma_B^2 + \varepsilon} \tag{15}$$

$$S_1 = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial l}{\partial y_i} \tag{16}$$

$$S_2 = \frac{1}{m}\sum_{i=1}^{m} \left(\frac{\partial l}{\partial y_i} \times x_i\right) \tag{17}$$

由式(12)可以看出，在算出了每个通道的参数项 B 和 C 后，BN 的 BP 就化为了仅与同一样本的同一位置的源操作数相关的计算。于是，式(9)和式(12)中的操作都能够直接与其后相邻层进行融合执行。

从式(12)～(15)中可以看出，FP 需要计算出均值 μ_B、方差 σ_B^2 和参数项 F_1、F_3 供 BP 阶段使用，并且需要计算出参数项 F_2 供 FP 阶段使用。

由式(10)、(11)、(15)可以看出，由于 γ、β 和 ε

都是训练中给定的数值,算出 μ_B 和 σ_B^2 后就能得到参数项 F_1、F_2 和 F_3。将式(2)展开,可得:

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m} x_i^2 - \mu_B^2 \tag{18}$$

由式(1)和(18)可以看出,计算 μ_B 和 σ_B^2 最关键的是计算出每个通道的 $\sum_{i=1}^{m} x_i$ 和 $\sum_{i=1}^{m} x_i^2$。其中,x_i 是 BN 层的输入值,也是 BN 前一层的输出值,因此,$\sum_{i=1}^{m} x_i$ 和 $\sum_{i=1}^{m} x_i^2$ 的计算可以与 BN 执行顺序的前一层进行融合,在前一层的原操作完成后,对输出值进行一个“后处理”,计算出需要的累积和。

从式(13)、(14)中可以看出,由于 F_1、F_3 和 μ_B 都是 FP 过程中已经算好的参数项,要算出参数项 B 和 C 仅需算出 S_1 和 S_2。从式(16)和(17)可得,计算 S_1 和 S_2 即计算每个通道中 $\frac{\partial l}{\partial y_i}$ 以及 $\frac{\partial l}{\partial y_i} \times x_i$ 的累积和。由于 $\frac{\partial l}{\partial y_i}$ 是 BN 层的输入误差值,也是 BN 在 BP 执行过程中的前一层的输出误差值,于是,可以将这 2 对累积和的计算与前一层进行融合,在前一层的原操作完成以后,对输出值进行一个“后处理”,计算出需要的累积和。

于是,BN 层的 FP 和 BP 过程都可以分为 2 个步骤:第 1 步为计算累积和及参数项,与执行顺序的前一层进行融合,作为其输出值的“后处理”出现;第 2 步为式(9)、(12)中的仅与同一样本的同一位置的源操作数相关的计算,与执行顺序的后一层进行融合,作为对融合新层的输入值的“前处理”出现。

2.3　常见层间连接的融合方案

下文中使用 ifmap 和 ofmap 分别代表 FP 过程中一个层的输入数据与输出数据;使用 ierr 和 oerr 分别代表 BP 过程中一个层的输入误差值与输出误差值;使用 WG(weight gradient generation)来表示训练中反向传播时计算权值梯度的过程;使用 $C1$、$C2$、$C3$ 代表卷积层。

(1) $C1 \rightarrow$ReLU$\rightarrow C2$

FP 时,ReLU 与 $C2$ 融合,仅将 $C1$ _ ofmap 存回片外,不需要将 ReLU _ ofmap 存回片外。于是,ReLU 层的片外访存量为 0。

BP 时,由于 $C2$ 的 WG 过程需要用到 ReLU _ ofmap,即会把 $C1$ _ ofmap 从片外读到片上,于是选择 ReLU 与 $C2$ 融合,对 $C2$ _ oerr 进行后处理计算 $C2$ _ oerr × ReLU′($C1$ _ ofmap) 作为融合新层的结果,存回到片外,供 $C1$ 做 BP 时使用。ReLU 化为后处理进行,需要用到 $C1$ _ ofmap,但由于 $C2$ 的 WG 过程本身就会将 $C1$ _ ofmap 读到片上,于是,ReLU 带来的片外访存量为 0。

(2) $C1 \rightarrow$BN$\rightarrow C2$

FP 时,BN 的第 1 步与 $C1$ 融合,计算出参数项;第 2 步与 $C2$ 融合。仅将 $C1$ _ ofmap 存回片外。因此,BN 层没有额外的片外访存,片外访存量为 0。

BP 时,BN 的第 1 步与 $C2$ 融合。由于 WG 时需要用到 BN _ ofmap,于是从片外读 $C1$ _ ofmap 到片上,参与 BN 与 $C2$ 融合层的后处理和 $C2$ 的 WG 的计算。BN 的第 2 步与 $C1$ 融合,计算 $C1$ 的 BP 时,读入 $C2$ _ oerr 和 $C1$ _ ofmap 执行式(12)中的计算,算出 $C1$ _ ierr 后执行 $C1$ 层的操作。于是,$C1$ 融合层的执行除了读 $C2$ _ oerr 作为输入值外,还因为前处理多读了 $C1$ _ ofmap 到片上,相当于 BN 层给融合层的执行带来了 1 倍输入值的片外访存量。

(3) $C1 \rightarrow$BN$\rightarrow$ReLU$\rightarrow C2$

FP 时,BN 的第 1 步与 $C1$ 融合,计算出参数项;BN 的第 2 步与 ReLU 和 $C2$ 一起融合为一个新层,BN 第 2 步和 ReLU 都作为融合新层的前处理执行。仅将 $C1$ _ ofmap 存回片外,于是,BN 和 ReLU 没有额外的访存,片外访存量为 0。

BP 时,BN 的第 1 步与 ReLU 和 $C2$ 融合,都作为对 $C2$ _ oerr 的后处理执行。由于式(12)、(16)、(17)中的 $\frac{\partial l}{\partial y_i}$ 就是 BN _ ierr,有:

$$\text{BN\_ierr} = C2\text{\_oerr} \times \text{ReLU}'(F_1 \times C1\text{\_ofmap} + F_2) \tag{19}$$

将式(19)带入式(16)、(17)中,就能计算出 BN 第 1 步的参数项。BN 的第 2 步与 $C1$ 融合,计算 $C1$ 的 BP 时,读入 $C2$ _ oerr 和 $C1$ _ ofmap,执行将式(19)带入式(12)后的计算,得到 $C1$ _ ierr 后执行 $C1$ 层的操作。

$C2$ 的后处理需要 $C1$ _ ofmap,由于 $C2$ 的 WG

本身需要将 *C*1 _ ofmap 读到片上,于是不增加新的片外访存。*C*1 融合新层的执行由于前处理多读了 *C*1 _ ofmap 到片上,相当于 BN 和 ReLU 带来了 1 倍输入值的片外访存量。

(4) *C*1→BN→Add→ReLU→*C*2

1) *C*1 的同个残差块中的捷径连接没有卷积计算

FP 时,BN 的第 1 步与 *C*1 融合;BN 的第 2 步与 Add、ReLU 层和 *C*2 一起融合成一个新层,从片外读入 *C*1 _ ofmap 算出 BN _ ofmap,从片外读入 *C*1 同个残差块中捷径连接的输出结果与 BN _ ofmap 相加,做 ReLU 操作,得到 *C*2 _ ifmap 后进行 *C*2 层的操作。这个过程中,除了要将 *C*1 _ ofmap 存回片外,由于 *C*2 的 WG 需要用到 *C*2 _ ifmap,选择将算好的 C2 _ ifmap 也存到片外。因此,在计算 *C*2 融合层时,需要从片外读入捷径连接的输出值且需要将算好 *C*2 _ ifmap 存到片外,相当于 BN、Add、ReLU 带来了 2 倍输入值的片外访存量。

BP 时,将 BN 的第 1 步、ReLU、误差 Add 与 *C*2 融合为一个新层,都作为对 *C*2 _ oerr 的后处理执行。用 Scut _ oerr 代表 *C*2 同一残差块中捷径连接的误差,在 *C*2 算出了 *C*2 _ oerr 后,继续进行:

$$BN\_ierr = (C2\_oerr + Scut\_oerr) \times ReLU'(ReLU\_ifmap) \quad (20)$$

计算,将 BN _ ierr 作为融合新层的结果存回片外,并继续使用算出的 BN _ ierr 值进行 BN 层的第 1 步。BN 的第 2 步与 *C*1 融合,计算时,读入 BN _ ierr 和 C1 _ ofmap 执行式(12)中的计算,算出 *C*1 _ ierr 后执行 *C*1 层的操作。

由于 *C*2 的 WG 直接使用 *C*2 _ ifmap,于是 *C*2 融合新层的后处理需要多读 Scut _ oerr 和 *C*1 _ ofmap 到片上,带来了 2 倍输入值的访存量;*C*1 融合新层的前处理时需要多读 *C*1 _ ofmap 到片上。因此,BN、ADD、ReLU 总共带来了 3 倍输入值的片外访存量。

2) *C*1 同个残差块中的捷径连接是 *C*3→BN3

FP 时,BN3 的第 1 步与 *C*3 融合;BN3 的第 2 步与 BN 层的第 2 步、Add、ReLU 和 *C*2 一起融合为一个新层,作为对该新层的前处理出现。与 1) 中相同,除了将 *C*2 _ ofmap 存回片外,还要将前处理结果 *C*2 _ ifmap 存回片外。同样与 1) 中相同,BN、Add、ReLU 带来了 2 倍输入值的片外访存量。

BP 时,将 BN 的第 1 步、BN3 的第 1 步、ReLU、误差 Add 与 *C*2 融合为一个新层,都作为对 *C*2 _ oerr 的后处理执行。同样使用式(20)计算出 BN _ ierr 作为融合新层的结果存回片外,并分别利用 BN _ ierr 执行 BN 的第 1 步和 BN3 的第 1 步。BN 的第 2 步与 *C*1 融合。BN3 的第 2 步与 *C*3 融合。

由于 *C*2 的 WG 直接使用 *C*2 _ ifmap,于是 *C*2 融合新层的后处理需要多读 Scut _ oerr、*C*1 _ ofmap、*C*3 _ ofmap 到片上,带来了 3 倍输入值的访存量;*C*1 融合新层的前处理时需要多读 *C*1 _ ofmap 到片上;*C*3 融合新层的前处理时需要多读 *C*3 _ ofmap 到片上。因此,BN、ADD、ReLU 总共带来了 5 倍输入值的片外访存量。

2.4 减少的片外访存量和占用的片外内存容量

表 1 中总结了 2.3 节中各类常见层间连接的访存密集型层在 FP 和 BP 过程中单独执行或融合执行时与片外内存交互的数据量,以一个访存密集型层的 *mini-batch* 个样本的输入数据量为单位。1.2.1 节和 1.2.2 节说明了 ReLU、BN、Add 的 FP 和 BP 过程在单独执行时分别有 2、3、3 倍和 3、5、3 倍输入值的片外访存量。情况 3 包含了 1 个 BN 和 1 个 ReLU,因此 FP 和 BP 分别为 5 倍和 8 倍输入值访存量;情况(4)-1)包含了 1 个 BN、1 个 Add 和 1 个 ReLU,因此 FP 和 BP 分别为 8 倍和 11 倍输入值访存量;情况(4)-2)包含了 2 个 BN、1 个 ADD 和 1 个 ReLU,因此 FP 和 BP 分别为 11 倍和 16 倍的输入值访存量。

表 1 访存密集型层片外访存量

	访存密集型层片外访存量	(1)	(2)	(3)	(4)-1)	(4)-2)
FP	不融合	2	3	5	8	11
	融合	0	0	0	2	2
BP	不融合	3	5	8	11	16
	融合	0	1	1	3	5

从表 1 中可以看出,融合后的访存密集型层执

行时与片外内存交互的数据量在各类常见连接下都得到了大幅度的减少。

表2中给出了2.3节中各类常见层间连接的访存密集型层在训练过程中单独执行或融合执行时占用的片外内存的容量,以一个访存密集型层的 *mini-batch* 个样本的输入数据量为单位。情况(1)~(3)中,完全消除了BN、ReLU层在训练过程中对片外内存容量的占用;情况(4)-1)和(4)-2)由于需要在FP时将前处理结果 *C*2_ifmap 存回片外供WG使用,于是占用1倍输入值的内存容量。

表2　访存密集型层占用片外内存容量

访存密集型层占用片外内存容量	(1)	(2)	(3)	(4)-1)	(4)-2)
不融合	1	1	2	3	4
融合	0	0	0	1	1

融合后的执行方案大幅减少了训练过程中在片外内存中存储的数据量,能够容纳更大规模的样本在同一加速器芯片上进行训练,缓解因片外内存容量的限制而带来的训练的扩展性或性能上的问题。

3　加速器设计

基于上述融合方案,本文设计实现了一个面向训练的加速器,能够高效地支持融合新层的执行,采用了暂存前处理结果、后处理操作与计算密集层的运算操作并行执行优化的策略,进一步提升了融合新层的性能。

3.1　加速器结构

图3展示了一个面向训练的加速器结构,其中,实线边框模块与实线连接线构成了加速器的基础结构,支持每个层单独执行。基础结构包含了 *M* 行 *N* 列的处理单元(processing unit, PU),主要用来执行计算密集型层的计算;有3块分离的片上缓存区Fbuf_In、Wbuf和Fbuf_Out,Fbuf_In主要用来存放计算密集型层的输入值,Wbuf用来存放权值以及权值梯度,Fbuf_Out主要用来存放每层计算出的输出值以及WG所需的ifmap值;每块Fbuf_In都连接着一个输入寄存器堆Ifreg,给对应行的所有PU提供输入值;每块Wbuf都连接着一个权值寄存器堆Wreg,给对应列的所有PU提供权值。每个PU中包含了多个乘加计算单元及一个输出寄存器堆Ofreg,Ofreg与Fbuf_Out直接相连,用来存计算过程中产生的中间值和最终的运算结果以及存WG过程用到的ifmap值。同一列中相邻2个PU形成一个组,共同拥有一个累加单元Adder,用来实现2个PU算出的结果的累加。Fbuf_Out还连接着一个向量单元VecU,用来处理非访存密集型层中的计算。指令队列Inst_Queue从Host处接收指令并转换成控制信号控制每个PU列的执行。

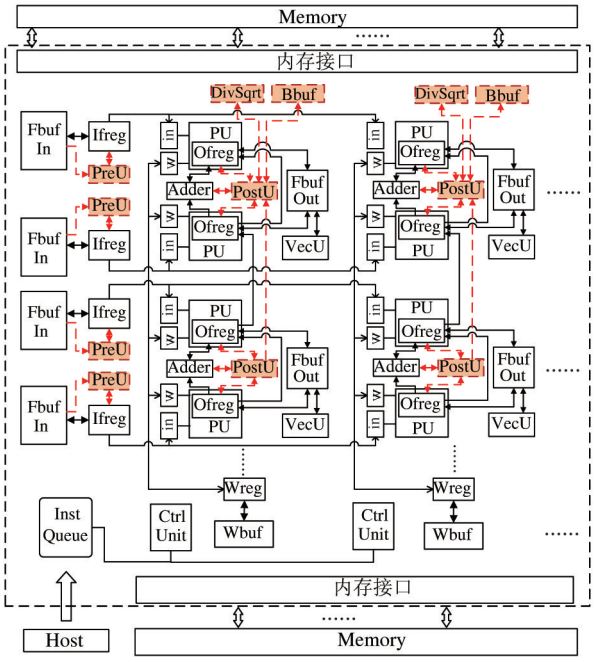

图3　加速器的结构

图3中虚线边框模块(带阴影模块)与虚线连接线是为了支持非访存密集型层的融合执行而增加的硬件结构。在每行的Fbuf_In和Ifreg旁增加了一个PreU,用来执行对融合新层输入值的前处理。每列中相邻2个PU共同拥有一个PostU模块,用来执行对融合新层的运算结果的后处理。每一列中增加了一个除法开方单元DivSqrt,用来处理BN第1步中计算参数项涉及到的除法和开方运算。每列中还增加了一块缓存区Bbuf,用来存储BN计算累积值或参数项过程中需要的参数值(γ和β)、计算的中间结果、算出的参数项;并为融合新层的前处理或

后处理提供需要的参数项值。

对于每个从 Fbuf _ In 中读出的数据,都增加一份如图 4 所示的 PreU 单元,其中主要包含了 1 个乘加单元、1 个 16 比特的寄存器 SrcReg 和一些多选逻辑。从 Fbuf _ In 中读出的数据可以写到 Ifreg 或者 SrcReg 中,和常数值以及参数项 F 一起,生成乘加单元的源操作数,乘加结果写回到 Ifreg 或者 SrcReg 中。前处理过程的最终结果写到 Ifreg 中,然后送到各行的 PU 中进行后续的计算。

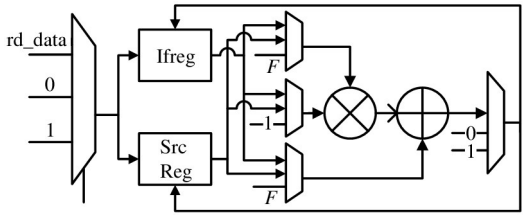

图 4　PreU 的结构

PostU 的结构如图 5 所示,主要包含了 4 个乘加单元和 2 项 4 ×32 比特的寄存器 Comp _ Reg 和 Acc _ Reg。乘加单元使用来自 Ofreg、Comp _ Reg、Acc _ Reg 的数据,常数值 0、1 以及参数项 F 作为被乘数、乘数和加数,乘加结果写回到 Ofreg 中或 Comp _ Reg、Acc _ Reg 中。

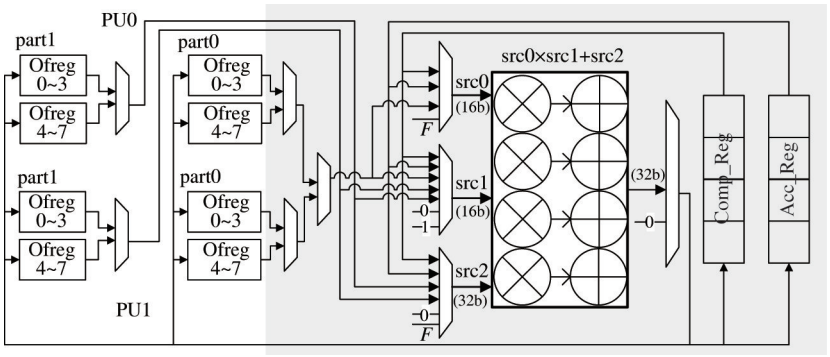

图 5　PostU 的结构

3.2　映射到加速器上执行的方式

3.2.1　前处理操作的加速方案

融合新层的前处理的执行时间可能会导致后续计算的阻塞,为了缓解这部分影响,执行过程中,将 Fbuf _ In 分成 2 部分使用,一部分存融合新层的原输入值,当第 1 次从 Fbuf _ In 中读入一行原输入值进行计算时,将该行输入值的前处理结果从 Ifreg 中写回到 Fbuf _ In 的第 2 部分中去;后续计算直接从 Fbuf _ In 的第 2 部分中取算好的前处理结果到 Ifreg 中,直接参与计算。于是,后续关于该行的输入值的计算都不会再受到前处理的影响。

3.2.2　后处理操作的执行方案

为了消除后处理给融合新层带来的影响,采用了让 PU 的计算与对结果的后处理并行执行的方案。如图 5 所示,将一个 PU 中的 8 项 Ofreg 平分成 2 份 ping-pong 使用,一份用来放 PU 中正在计算的中间结果(32 比特浮点数);另一份存储着已经算出的原始输出结果(16 比特浮点数),将其作为源操作数送到 PostU 中执行需要的后处理。由于原始输出结果只需要 part0 部分的 Ofreg 即可放下,于是空余的 part1 部分可以用来放后处理过程中需要的其他源操作数。

后处理过程由 2 类操作构成。第 1 类是由原始输出结果算出融合新层的新输出结果,存回片外。

如图 6 所示,使用 Ofreg 0 ~ 3 存放 PU0 和 PU1 中正在计算的中间结果;Ofreg 4 ~ 7 的 part0 中存储着已经算完的原始结果,依次执行关于 PU0 和 PU1 中的原始结果的后处理第 1 类操作。

关于 PU0 的后处理:①先从 PU0 的 Ofreg 4 ~ 7 的 part0 读出第 1 列的 4 个原始结果,分别从 PU0 和 PU1 的 Ofreg 4 ~ 7 的 part1 读出第 1 列的其他源操作数,送到 PostU 的 4 个乘加单元中进行计算,得到 4 个新结果后,写回到 PU0 的 Ofreg 4 ~ 7 的 part0 中的第一列中;然后对②处的列进行相同的操作,直到 PU0 的 Ofreg 4 ~ 7 的 part0 中的所有原始结果都处理完成,得到所有的输出结果为止;③将 PU0 的

Ofreg 4 ~ 7 的 part0 中的新输出结果写回到 Fbuf _ Out 中。关于 PU1 的后处理,与 PU0 中类似执行即可。

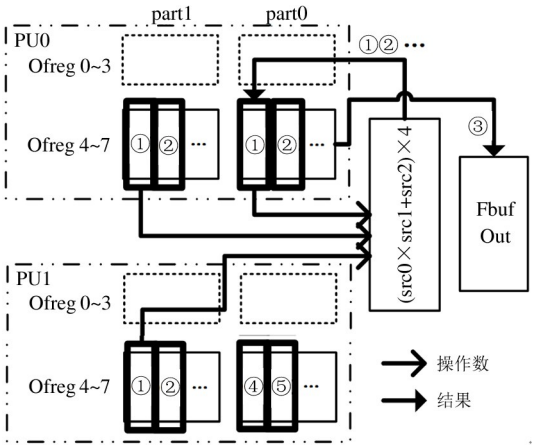

图 6　后处理第 1 类操作的执行过程

后处理过程的第 2 类操作是根据输出结果计算 BN 的累积值和参数项。

如图 7 所示,计算累积值时:①分别从 Ofreg 4 ~7 的 part0 和 part1 的第 1 列读出 4 个结果值和 4 个其他源操作数,以及从 Comp _ Reg 中读出 4 个已有的累积值,送到 PostU 的 4 个乘加单元中,计算出 4 个新的属于同一个累加值的部分结果,存到 Comp _ Reg 中;然后对②处的列执行相同的操作,直到 Ofreg 4 ~ 7 的 part0 中的所有结果都处理完为止;③接着使用 Adder 将 Comp _ Reg 中的 4 个部分结果累加为 1 个累积值,累积值写回到 Acc _ Reg 中;由于每列 PU 执行同一个输出通道的计算,当 PU0 ~ 3 当前计算的这个输出通道的计算完成后,④使用 PU0/1 旁的 Adder 将 PU0/1 和 PU2/3 对应的 Acc _ Reg 中关于不同样本同一通道的累积值累加起来,得到新的累积值先写回到 PU0/1 对应的 Acc _ Reg 中;最后⑤存到 Bbuf 中。

等所有样本关于一个通道的累积值计算完成后,使用每列第 1 个 PostU(PU0/1 对应的 PostU)进行参数项计算。从 Bbuf 中读出源操作数,将源操作数和中间结果都存到 Comp _ Reg 和 Acc _ Reg 中;并使用该列的 DivSqrt 进行参数项中的除法、开方计算。最终算出关于该通道的参数项,写回到 Bbuf 中。

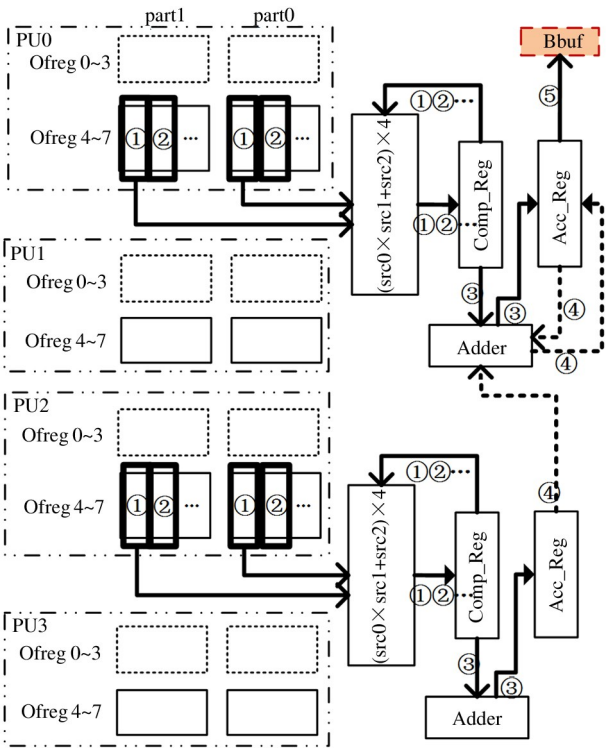

图 7　后处理第 2 类操作计算累积值的过程

4　实验与结果

本节介绍实验环境和实验数据,着重从性能的角度评估了本文提出的加速器结构及加速访存密集型层的融合方案。

4.1　实验方法

利用 Verilog 语言在寄存器传输级(register transfer level, RTL)实现了图 3 所示的加速器。使用 Synopsys Design Compiler 在 ST 28 nm 工艺下进行综合。使用 Synopsys VCS 工具对加速器设计进行模拟和验证,测量其性能。

4.2　加速器配置

实验中加速器采用 16 行 32 列的 PU,每个 PU 中有 32 个乘加运算单元,共 16 384 个乘加单元。每个 Adder 中采用 4 个 32 比特浮点格式的加法器。每行 PU 对应 1 块 768 kB 的 Fbuf _ In,共 12 MB。每列 PU 对应 1 块 36 kB 的 Wbuf,共 1152 kB。2 个 PU 对应 1 块 64 kB 的 Fbuf _ Out,共 16 MB。每列 PU 对应 1 块 6 kB 的 Bbuf,共 192 kB。Fbuf _ In 每次读写 16 个数据,于是每行的 PreU 中包含了 16 个乘加运算单元。PU 中一项 Ofreg 为 16 × 32 比特。加速器

的频率为 800 MHz。关于 Wbuf 和 Bbuf 的访存带宽为 25.6 GB/s,关于 Fbuf _ In 和 Fbuf _ Out 的访存带宽为 205 GB/s。该配置下,主运算阵列的乘加单元数量、片上缓存区大小、带宽都与主流训练加速器相近。

4.3　Benchmark

为了评估本文提出的加速器的性能,选择了 Pytorch 的 torchvision. models 中提供的 resnet18 和 resnet50 作为测试模型。训练过程中 *mini-batch* 设为 32。

4.4　实验和结果

图 8 给出了 FP 阶段 ResNet50 的各个 ResBlock 在本文提出的加速器上融合执行、单独执行的时间。从图 8 中可以看出,每个 ResBlock 融合执行的时间都比单独执行的时间短,其中性能提升最大的是 conv2 _ x blk0,提高了 3 倍;conv2 _ x blk1/2、conv3 _ x blk1/2/3 的提升也较大,分别达到了 1.3 倍和 1 倍;conv5 _ x blk1/2 的提升较小,约为 10%。整个 FP 过程融合执行的性能比单独执行时提高了87.0%。

图 9 给出了 BP 阶段 ResNet50 的各个 ResBlock 在本文提出的加速器上融合执行、单独执行的时间。除了 conv5 _ x 的 blk1 和 blk2,其他 ResBlock 的融合执行时间都比单独执行时间短,性能提升了 0.41 ~ 2.8 倍,整体上融合执行时性能比单独执行时提升了 1 倍。

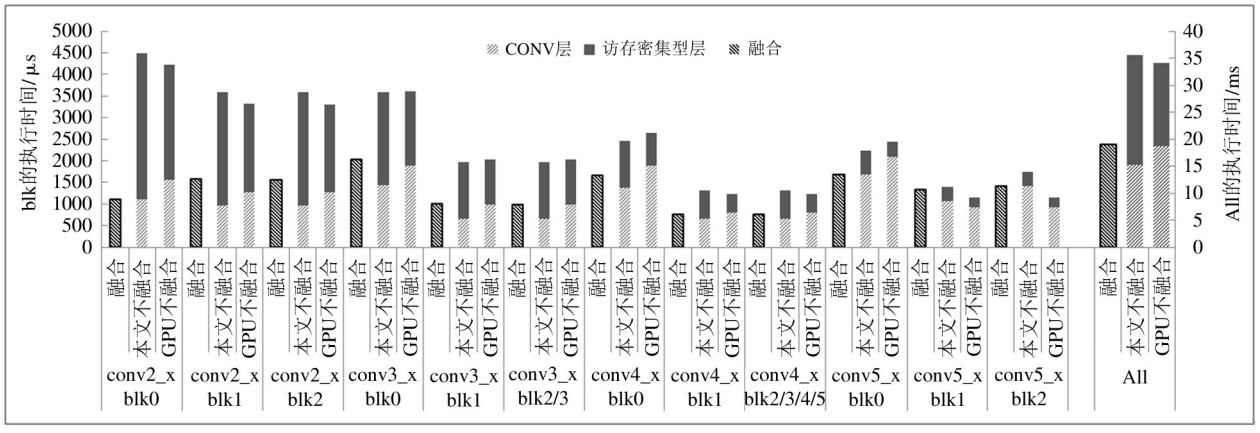

图 8　FP 阶段 ResNet50 各 ResBlock 融合、单独执行的时间

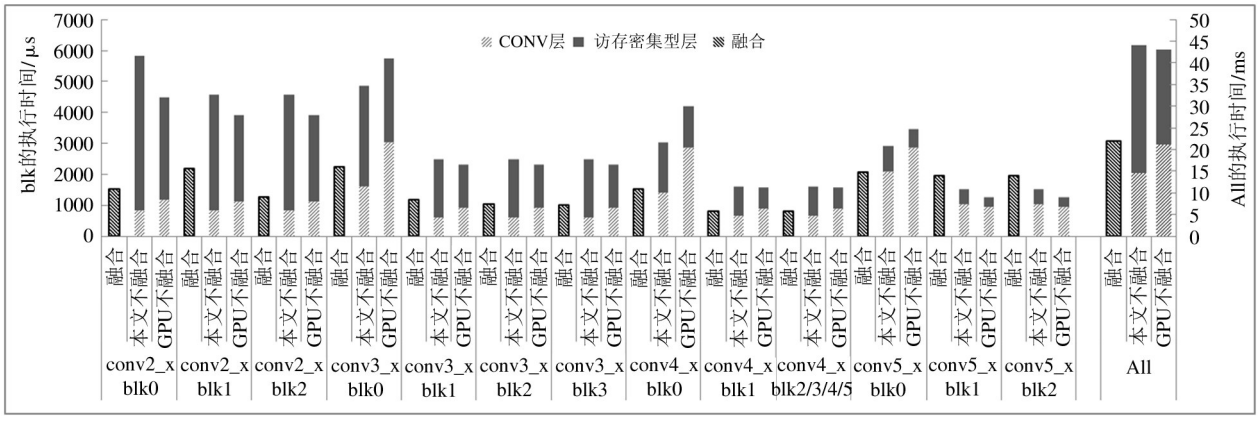

图 9　BP 阶段 ResNet50 各 ResBlock 融合、单独执行的时间

图 8 和 9 中还给出了各 ResBlock 在 GPU(NVIDIA GeForce RTX 2080 Ti)上单独执行时的时间。单独执行时,本文加速器与 GPU 的执行时间相差在 5% 以内;采用融合方案后,本文加速器在FP 和 BP阶段分别比 GPU 的性能提高了 78.9% 和95.7%。

从图 8 和 9 中可以看出,ResNet50 网络模型中

前面的 ResBlock 采用融合方案执行带来的性能提升更大,后面的更小。这是因为前面的 ResBlock 中采用了更大的 ifmap、ofmap、ierr、oerr,因而在单独执行访存密集型层时,需要消耗更长的时间来与片外内存交互数据,于是在融合方案下能够消除更多的时间,带来更大的性能提升。

另外,由图 8 和 9 中还可以看出,融合后的执行时间比单独执行时卷积层的执行总时间要长,这主要由 3 个方面的原因造成。第 1,融合新层的前处理可能导致源操作数不能及时准备好送到 PU 中执行运算;第 2,融合新层的后处理可能导致在 PU 计算完成后另一部分的后处理没完成而使 PU 的运算阻塞;第 3,融合新层相比于原始的卷积层,可能增加了执行过程中需要的源操作数,引起访存时间的增加,当访存带宽不够时,阻塞 PU 的计算。图 10 中给出了 ResNet50 的 2 个 ResBlock 块中的各融合新层在 FP 和 BP 时由上述 3 类原因而增加的执行时间的占比。由于前处理增加的时间占比最大的是 BP 时的 conv3 _ x blk0 中的 conv1,达到了 15%,平均下来是 5%,说明 3.2.1 节中的前处理加速方案很好地消除了前处理带来的影响。图 10 中,只有 BP 时 conv2 _ x blk1 中的 conv0 有后处理对执行时间的明显影响,其他融合新层中后处理的执行时间都隐藏在了卷积层运算执行的过程中,说明 3.2.2 节中的后处理并行执行方案几乎消除了后处理对执行时间的影响。从图 10 中可以看出,FP 时的 conv0、BP 时的 conv0 和 conv2 中由于访存而增加的时间占比很高,平均在 50%。这是因为这些融合层执行时涉及到了 Add 层的融合以及 BP 时关于 1 个或 2 个 BN 层的后处理,相比于原始的卷积层需要更多的访存量。

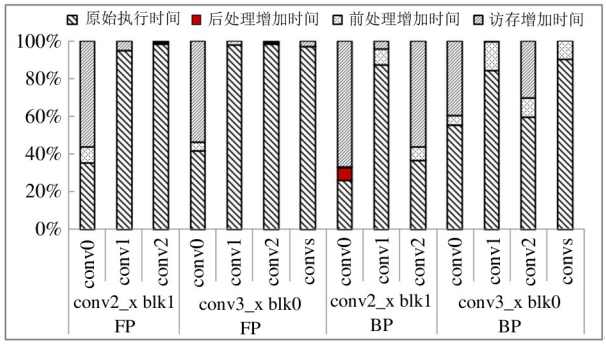

图 10　融合新层各类增加时间的占比

表 3 分别给出了图 3 中所示的加速器基础部分与支持融合的加速器的各组成部分的面积与功耗。加速器基础部分的面积和功耗分别为145.6 mm^2 和 75.2 W,支持融合后的面积和功耗分别为 154.9 mm^2 和 82.9 W,分别增加了 6.4% 和 10.3%。面积与功耗的增长主要由 PU 中的 PostU、Ifreg 中的 PreU 以及 Bbuf 带来。

表 3　加速器的面积与功耗

加速器	基础加速器		支持融合加速器	
	面积 /mm^2	功耗 /W	面积 /mm^2	功耗 /W
整体	145.6	75.2	154.9	82.9
PU	45.4	41.7	50.6	46.4
Ifreg	2.4	2.2	4.8	4.5
Wreg	5.4	3.0	5.4	3.0
Fbuf _ Out	58.7	17.8	58.7	17.8
Fbuf _ In	29.4	8.9	29.4	8.9
Wbuf	4.2	1.3	4.2	1.3
Bbuf	0.0	0.0	1.7	0.7
Inst _ Queue	0.0	0.0	0.0	0.0
Ctrl Unit	0.3	0.2	0.3	0.2

表 4 中给出了 ResNet18 和 ResNet50 的 FP、BP 阶段融合执行比单独执行性能提升的百分比,FP 平均提升了 67.7%,BP 平均提升了 77.6%。

表 4　FP 和 BP 阶段融合执行性能提升百分比

加速器	FP 提升	BP 提升
ResNet18	87.0%	100.7%
ResNet50	48.5%	54.5%
平均	67.7%	77.6%

5　结　论

本文针对卷积神经网络模型中的批归一化层、加法层、非线性激活层等访存密集型层以及其常见连接,提出了在训练的前向过程与反向过程中,将访存密集型层与其前后的计算密集型层融合为一个新

层执行的方式,将访存密集型层的操作作为对融合新层中输入数据的前处理或者原始输出数据的后处理进行,大幅减少了访存密集型层执行过程中与片外内存交互的数据量以及训练过程中需存储在片外内存中的数据量;并针对该融合执行方案,设计实现了一个专门的面向训练的加速器,采用了暂存前处理结果、后处理操作与计算密集层的运算操作并行执行的优化策略,大幅提升了融合新层在训练各阶段执行的性能。实验结果显示,基于 2 个常见的卷积神经网络模型,在面积仅增加 6.4%、功耗增加 10.3% 的开销下,训练 FP 和 BP 阶段的性能分别实现了 67.7% 和 77.6% 的提升。

参考文献

[1] NORRIE T, PATIL N, YOON D H, et al. Google's training chips revealed: TPUv2 and TPUv3[C]//2020 IEEE Hot Chips 32 Symposium. Palo Alto: IEEE, 2020: 1-70.

[2] LIAO H, TU J, XIA J, et al. Ascend: a scalable and unified architecture for ubiquitous deep neural network computing: industry track paper[C]//2021 IEEE International Symposium on High-Performance Computer Architecture. Seoul: IEEE, 2021: 789-801.

[3] ZHAO Y, LIU C, DU Z, et al. Cambricon-Q: a hybrid architecture for efficient training[C]//2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture. Valencia: IEEE, 2021: 706-719.

[4] VENKATARAMANI S, SRINIVASAN V, WANG W, et al. RaPiD: AI accelerator for ultra-low precision training and inference[C]//2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture. Valencia: IEEE, 2021: 153-166.

[5] QIN E, SAMAJDAR A, KWON H, et al. Sigma: a sparse and irregular GEMM accelerator with flexible interconnects for DNN training[C]//2020 IEEE International Symposium on High Performance Computer Architecture. San Diego: IEEE, 2020: 58-70.

[6] 周聖元, 杜子东, 陈云霁. 稀疏神经网络加速器设计[J]. 高技术通讯, 2019,29(3):222-231.

[7] 腾讯科技(深圳)有限公司. 基于神经网络的数据处理方法、装置、设备及存储介质: CN110163337A[P]. 2019-08-23.

[8] 深圳芯英科技有限公司. 数据处理方法、装置、芯片以及计算机可读存储介质: CN111428879A[P]. 2020-07-17.

[9] IOFFE S, SZEGEDY C. Batch normalization: accelerating deep network training by reducing internal covariate shift[C]//International Conference on Machine Learning. Lille: ICML, 2015:448-456.

[10] HE K, ZHANG X, REN S, et al. Deep residual learning for image recognition[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. Las Vegas: IEEE, 2016:770-778.

[11] XIE S, GIRSHICK R, DOLLÁR P, et al. Aggregated residual transformations for deep neural networks[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. Honolulu: IEEE, 2017:1492-1500.

[12] ZAGORUYKO S, KOMODAKIS N. Wide residual networks[EB/OL]. (2016-05-23)[2022-01-20]. https://arxiv.org/pdf/1605.07146.pdf.

[13] SANDLER M, HOWARD A, ZHU M, et al. Mobilenetv2: inverted residuals and linear bottlenecks[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. Salt Lake City: IEEE, 2018: 4510-4520.

[14] HOWARD A, SANDLER M, CHU G, et al. Searching for Mobilenetv3[C]//Proceedings of the IEEE/CVF International Conference on Computer Vision. Seoul: IEEE, 2019:1314-1324.

[15] TAN M, CHEN B, PANG R, et al. Mnasnet: platform-aware neural architecture search for mobile[C]//Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. Long Beach: IEEE, 2019: 2820-2828.

[16] RADOSAVOVIC I, KOSARAJU R P, GIRSHICK R, et al. Designing network design spaces[C]//Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. Seattle: IEEE, 2020:10428-10436.

Accelerating memory intensive layer of neural networks with layer fusion

YANG Can, WANG Chongxi, ZHANG Longbing
(State Key Laboratory of Processors, Institute of Computing Technology,
Chinese Academy of Sciences, Beijing 100190)
(Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190)
(University of Chinese Academy of Sciences, Beijing 100049)

Abstract

Recently, deep neural networks are widely used in various fields. It is necessary to train each neural network model to get better model parameters for different application scenarios. Thus, the demand for training speed is increasing. However, the existing research usually focuses on the acceleration of computation intensive layers but ignores the acceleration of memory intensive layers. The efficiency of the memory intensive layer is mainly determined by the memory bandwidth, thus only improving the computation speed has little effect on the layer's performance. From the perspective of execution order, this paper proposes a method of fusing the memory intensive layer and the computation intensive layer before it or behind it into a new fused layer, and the operation of memory intensive layer is performed as the pre-processing of the input data or the post-processing of the origin output data in the fused layer. Thus, it reduces the access of the memory intensive layer to off-chip memory greatly during training. Based on the fusion method, a new accelerator for training is implemented, adopting the optimization strategy to further improve the training performance, including temporarily storing the pre-processing results and concurrently executing the operations of post-processing and the computation intensive layer. The experimental results show that the performance is improved by 67.7% and 77.6% respectively in the forward propagation stage and backward propagation stage of training at the cost of 6.4% increase in area and 10.3% increase in power.

Key words: neural network, training, accelerator, convolutional neural network (CNN), memory intensive layer, batch normalization (BN) layer