软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.006051]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

一种面向稀疏卷积神经网络的 GPU 性能优化方法*

董 晓[1,2], 刘 雷[1], 李 晶[1], 冯晓兵[1,2]

[1](计算机体系结构国家重点实验室(中国科学院 计算技术研究所),北京 100190)

[2](中国科学院大学,北京 100190)

通讯作者: 刘雷, E-mail: liulei@ict.ac.cn

摘 要: 近些年来,深度卷积神经网络在多项任务中展现了惊人的能力,并已经被用在物体检测、自动驾驶和机器翻译等众多应用中.但这些模型往往参数规模庞大,并带来了沉重的计算负担.神经网络的模型剪枝技术能够识别并删除模型中对精度影响较小的参数,从而降低模型的参数数目和理论计算量,给模型的高效执行提供了机会.然而,剪枝后的稀疏模型却难以在GPU上实现高效执行,其性能甚至差于剪枝前的稠密模型,导致模型剪枝难以带来真正的执行性能收益.在本文中,我们提出了一种稀疏感知的代码生成方法,能够生成高效的稀疏卷积 GPU 程序.首先,我们为卷积算子设计了算子模板,并结合GPU的特点对模板代码进行了多种优化.算子模板中的源代码经过编译和分析被转换为算子中间表示模板,我们设计了一种稀疏代码生成方法,能够结合剪枝后的稀疏参数,基于中间表示模板生成对应的稀疏卷积代码.同时,我们利用了神经网络执行过程中的数据访问特点,对数据的访问和放置进行了优化,有效提升了访存吞吐量.最后,稀疏参数的位置信息被隐式编码在生成的代码中,不需要额外的索引结构,降低了访存需求.在实验中,我们证明了相对于 GPU 上已有的稀疏神经网络执行方法,本文提出的稀疏感知的代码生成方法能够有效提升稀疏卷积神经网络的性能.

关键词: 神经网络;稀疏;GPU;性能优化;卷积;代码生成

中图法分类号: TP311

Performance Optimizing Method for Sparse Convolutional Neural Networks on GPU

DONG Xiao[1,2], LIU Lei[1], LI Jing[1], FENG Xiao-Bing[1,2]

[1](State Key Laboratory of Computer Architecture (Institute of Computing Technology, Chinese Academy of Sciences), Beijing 100190, China)

[2](University of Chinese Academy of Sciences, Beijing 100190, China)

Abstract: In recent years, with dominating capability shown in plenty of tasks, deep convolutional neural networks have been deployed in applications including object detection, autonomous driving, machine translation, etc. But these models are accompanied by huge amounts of parameters and bring a heavy computational burden. The neural network pruning technique can recognize and remove parameters that contribute little to the accuracy, resulting in reduced amounts of parameters and decreased theoretical computational requirement, thus providing a chance to accelerate neural network models. However, it is hard for the pruned sparse models to achieve efficient execution on GPUs, and the performance of sparse models cannot even match their well-optimized dense counterparts. In this paper, we design a sparsity-aware code generating method, which can generate efficient GPU code for sparse convolutions in pruned neural networks. First, we design a template for convolution operators with several optimizations targeting GPU architecture. Through compiling and analyzing, the operator template is transformed to the intermediate representation template, which serves as the input to the designed algorithm to generate sparse convolution code according to specific sparse convolution parameters. Moreover, to improve memory throughput, we perform optimizations on data access and data placement based on the characteristics of memory access in neural networks. Finally, as the location information can be encoded into the generated code implicitly, the index structure for the sparse parameters can be eliminated, reducing the memory footprint during the execution. In experiments, we demonstrate the proposed sparse code generating method can improve the performance of sparse convolutional neural networks compared with current methods.

收稿时间: 2019-10-05; 修改时间: 2020-01-13, 2020-03-17; 采用时间：2020-04-01; jos 在线出版时间: 2020-04-21

Key words: neural networks; sparse; GPU; performance optimization; convolution; code generation

深度神经网络近些年来持续受到学术界和工业界的广泛关注.自从 2012 年 AlexNet[1] 在大规模图像分类问题中展示出惊人的能力以来,研究人员持续通过在神经网络模型结构和训练算法等领域的创新,借助日益增长的算力和大规模数据集,不断提升神经网络在各类任务中的表现.与此同时,众多企业也将神经网络模型应用到各种应用中.比较典型的应用包括物体检测与识别[2] [3] ,自动驾驶[4] ,机器翻译[5] 等.虽然这些模型在各类任务中展现出了惊人的精度,但这些模型会占用大量存储空间,同时在执行时伴随着巨大的计算开销.例如,用于图像分类和物体检测等计算机视觉应用的 ResNet50 网络模型[6] 包含超过 2500 万个参数,对一张形状为 224*224 的彩色图像进行分类需要执行 76 亿次运算.另一方面,神经网络模型能力的进步也依赖于模型规模的增长.早期用于简单的手写数字识别的 LeNet5 模型[7] 仅包含约 6 万个参数,而对于在复杂的 ImageNet[8] 大规模图像分类比赛中取得优异表现的 AlexNet 模型[1],其参数数目超过了 6000 万.庞大的参数规模和计算需求阻碍了神经网络模型的广泛应用,同时也使得实现神经网络的高效执行成为了一个既有很强实际意义,同时也十分紧迫的问题.

面对神经网络庞大的参数数目和海量的计算需求,研究人员提出了模型剪枝的方法来挖掘神经网络模型参数中的冗余性,对模型进行简化.由于被移除的参数不需要保留,同时与之相关的计算也可以省略,所以模型剪枝方法能够有效降低神经网络模型的存储开销和计算需求.在保证剪枝后的模型在目标任务上的精度损失在一定范围内的条件下,模型剪枝方法能够从神经网络中识别出对最终精度影响不大的参数,并将这些参数从网络中移除,生成一个精简的模型.在模型剪枝方法中,不对可移除参数的分布位置施加约束的非结构化剪枝一般能够最大限度地挖掘参数的冗余性.典型的非结构化模型剪枝方法可以在精度几乎没有损失的情况下,移除模型中超过 90%的参数[9] [10] [11] [12] ,同时将剪枝后稀疏模型的计算需求降低为剪枝前的约 10%.

尽管非结构化的模型剪枝方法有效降低了理论计算量,但在 GPU 平台上将这部分理论性能收益转换为实际的性能加速却面临着严峻的挑战.首先,与剪枝前的稠密计算相比,剪枝后的稀疏计算的计算密度更低.这使得 GPU 计算核心与 DRAM 之间的数据传输容易成为性能瓶颈,导致剪枝后的稀疏计算难以充分利用 GPU 的
计算能力.其次,对于稀疏数据,我们往往仅保留其中的非 0 元素,并使用额外的索引结构存储这些元素的位置信息,以节约存储空间.这两部分共同构成了稀疏数据的表达,例如典型的稀疏矩阵格式 CSR(Compressed Sparse Row)[13] 、CSC(Compressed Sparse Column)、COO(Coordinate)和稀疏张量格式 CSF(Compressed Sparse Fiber)[14] 等.稀疏参数表示中的索引结构增加了稀疏神经网络执行时的访存需求,进一步恶化了计算密度低的问题.最后,GPU 本身的执行模型和存储层次都比较复杂,而且目前已有的在 GPU 上进行稠密神经网络计算的 cuDNN[15] 和 cuBLAS[16] 等方法经过了专家精心的手工优化.因此,需要结合计算中的数据访问特点和 GPU 的体系结构特征,对数据的布局和任务划分等进行相应优化,才能将稀疏计算的理论加速效果转化为实际的性能收益.

在本论文中,我们提出了一种稀疏感知的卷积算子代码生成方法,能够为剪枝后稀疏的卷积神经网络生成高效的前向推理的执行代码.图 1 展示了我们方法的整个流程.首先,我们为卷积算子设计了**算子模板**.算子模板不考虑卷积参数的稀疏性.通过对算子模板的编译和分析,我们建立卷积算子的**中间表示模板**.基于对中间表示模板的分析,我们可以建立卷积参数与中间表示中指令的映射关系.之后,通过结合具体的稀疏模型参数,对中间表示模板进行分析和变换,从中识别并删除与无效参数相关的指令序列,获得针对稀疏参数的算子代码.中间表示模板可以在不同的稀疏参数间复用.另外,我们基于模型参数在神经网络执行过程中取值固定的特点,为模型参数和算子输入设计了不同的访问路径,提升了执行中的访存吞吐量.同时,在生成的稀疏卷积算子代码中,有效
参数的位置信息已经被隐式地编码在代码序列中,不再需要额外的索引结构,从而降低了运行中的访存需求,

提

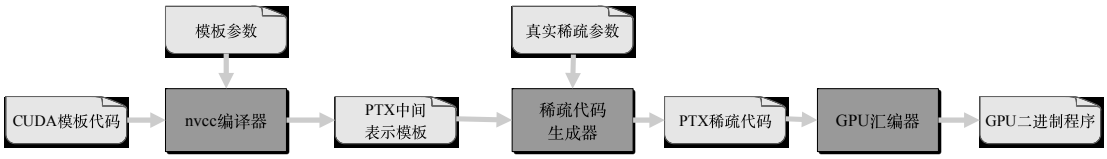

Fig.1 Overflow of the redundancy-aware code generating method

图 1 稀疏感知的算子代码生成流程

升了稀疏算子的计算密度.我们将上述技术整合为一个框架,并使用公开的神经网络模型和数据集进行了实验.通过实验,我们证明相对于 GPU 上已有的稠密执行方法和稀疏执行方法,本文所提出的方法能够有效提升稀疏卷积神经网络的性能.总结来看,我们在本文中做出了以下贡献:

(1) 我们提出了一种稀疏感知的卷积算子代码生成方法.以我们设计的算子模板为起点,我们建立了一种算子中间表示模板,基于中间表示模板的算法能够生成高效的稀疏卷积代码.

(2) 我们对稀疏卷积的内存访问设计了相应的优化方法.一方面,我们基于不同类型数据的访问特征,将不同类型的数据映射到 GPU 上不同的存储空间,充分利用了 GPU 的多种数据访问路径,提升了运行时的访存吞吐量;另一方面,我们将非 0 参数的位置信息编码在生成的代码中,消除了稀疏数据索引部分的开销,降低了运行时的访存需求.

(3) 我们通过实验说明了所提出方法的有效性.在来自 5 个卷积神经网络的 10 个卷积算子上,当稀疏程度达到 0.9 时,我们的方法可以在批大小为 64 时获得相对 cuBLAS 2.8-41.4 倍、相对 cuDNN 3.1-9.6 倍、相对 cuSPARSE 5.5-43.2 倍以及相对 Escoin 4.4-39.5 倍的加速效果;在批大小为 1 时,相对以上方法可以分别获得 1.2-3.2 倍、1.2-4.9 倍、2.4-15.6 倍以及 1.6-11.2 倍的加速效果.与结构化剪枝方法相比,我们也在类似的稀疏程度下展现了更好的性能收益.

本文内容按如下结构组织.第 1 节介绍相关背景知识,包括神经网络的剪枝方法以及 GPU 体系结构.第 2 节介绍本文的主要工作,包括算子和中间表示模板、稀疏感知的代码生成算法以及优化访存瓶颈的技术.第 3 节通过实验说明本文提出方法的有效性,展示了与已有工作的性能对比与分析结果,以及对本方法相关开销的分析.第 4 节概括了相关工作.第 5 节总结本文并对概括未来工作的方向.

1 背景知识

在本节中,我们介绍与本文内容相关的背景知识.我们首先介绍神经网络模型剪枝的相关概念.之后,我们介绍 GPU 体系结构的相关知识,重点关注 GPU 的存储层次.

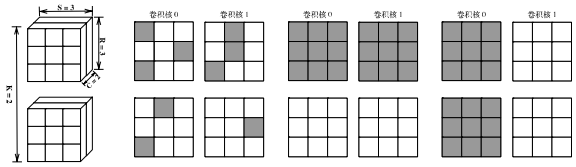

Fig.2 Example of a 2*2*3*3 convolution parameter tensor and different pruning methods

图 2 1 个 2*2*3*3 的卷积参数张量以及不同剪枝方法的例子

1.1 神经网络与模型剪枝

神经网络(Neural Network)是机器学习算法的一种,它使用相互连接的算子构建将输入数据映射到目标

输出的数学模型.神经网络中的算子也常被称为层,每个算子可能包含一些内部参数,并以之前的某些算子的输出为输入(第 1 个算子使用整个网络的输入),进行某种计算,产生输出结果.在模型的训练过程中,通过调整各个算子中的参数的取值,可以使整个神经网络模型逼近我们期望建立的目标函数.在训练完成后,神经网络中的所有的参数取值都固定下来,并被部署到相应的应用中.使用训练完成的神经网络处理输入数据的过程叫做前向推理(Inference).根据使用算子类型的不同,神经网络可以被进一步分为主要使用卷积层的卷积神经网络(Convolutional Neural Network)、主要使用全连接层的前馈神经网络(Feedforward Neural Network)以及使用长短期记忆模块的长短期记忆网络(Long Short-Term Memory)等.在本文中,我们主要关注卷积神经网络.这一类网络在计算机视觉领域的问题中已经获得了广泛应用,包括手写字符识别、图像分类和自动驾驶等.典型的卷积神经网络模型包括 LeNet[7] 、AlexNet[1] 、VGG[17] 和 ResNet[6] 等.卷积层的参数可以被抽象为一个 4 维张量,4 个维度分别是卷积核维度、通道维度、高和宽.图 2 展示了一个包括 2 个卷积核,每个卷积核包含 2 个通道,宽和高为 3 的卷积参数张量.

尽管神经网络模型参数规模巨大,已有研究发现,在训练完成的神经网络模型中,不同参数对模型最终精度的影响是不同的.存在相当比例的冗余参数.将其移除后,神经网络的精度不会发生明显变化.基于这一发现,研究人员提出了模型剪枝(Model Pruning)方法,从训练完成的神经网络中识别并删除部分参数,生成一个参数数目更少的轻量级稀疏模型.根据所删除的参数在原模型中的位置特点,模型剪枝方法可以被分为**结构化剪枝**[18] [19] [20] 和**非结构化剪枝**[11] [21] [22] 两类.结构化剪枝方法考虑参数在原模型中的分布特点,移除的参数不是随机分布在原模型的任意位置,而是具有一定的结构.以卷积参数为例,结构化剪枝方法以卷积核[19] ,或通道[18] ,或行、列[20] 等为单位进行剪枝.经过结构化剪枝获得的模型,其参数往往仍然具有规则的结构特点,可以被视为规模更小的稠密参数,因而通常可以直接使用针对稠密计算的优化库进行计算,所以一般不涉及稀疏计算问题.而非结构化剪枝将每个参数作为剪枝的基本单位,独立地对每个参数进行剪枝,因而可以删除任意位置的参数.典型的方法包括 Deep Compression[11] 、DNS[10] 、ADMM[9] 等.这些方法基于某种规则判断每个参数对最终精度的影响,识别出相对不重要的参数进行移除.图 2 右侧分别展示了非结构化剪枝和结构化剪枝的例子.其中结构化剪枝又包含以通道和卷积核为剪枝基本单位的情况.经过非结构化剪枝,非 0 参数的分布没有规律,模型参数变为不规则的稀疏张量(或稀疏矩阵).在稀疏模型上的计算也变成了稀疏模型参数与稠密输入数据的计算.在 GPU 上执行时,需要借助 cuSPARSE[23] 或 Escoin[24] 等稀疏计算库.由于非结构化剪枝不对剪枝参数的分布做任何限制和假设,所以往往能够获得相较于结构化剪枝更好的压缩效果[25] [26] ,能够更有效地降低模型的存储开销.目前的研究工作表明,非结构化剪枝能够实现对典型的卷积神经网络模型参数 9−100 倍[9] [10] [22] 的压缩,有效地解决了神经网络模型参数规模过大的问题,同时也显著降低了模型在推理过程中的计算需求.文献[27] 也按照剪枝粒度的不同概述了当前的模型剪枝工作.

1.2 GPU体系结构

GPU 由于其高性能和高能效比的特点,已经在高性能计算和深度学习等领域获得了广泛应用.GPU 一般包含多个流多处理器(streaming multiprocessor),每个流多处理器内又包含众多执行核心(CUDA core).在 GPU 程序载入时,GPU 线程按照用户指定的配置被分层组织并绑定到具体的执行核心上.线程首先被划分为线程块(thread block),线程块绑定到流多处理器上.线程块内的线程绑定到相应的流多处理器的执行核心上执行.在执行时,GPU 线程以 warp 为单位进行调度.一个 warp 包含相邻的 32 个线程.通过支持大量线程的并行执行,GPU 能够提供很高的峰值性能.

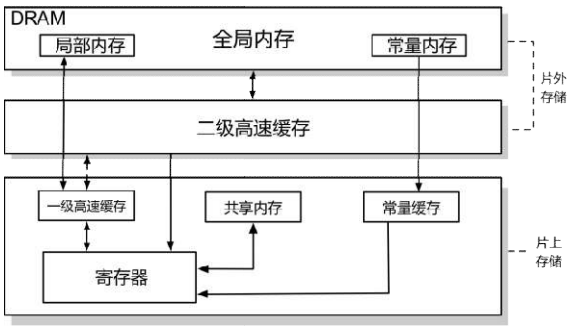

Fig.3 GPU memory hierarchy

图 3 GPU 存储层次示意图

为了满足大量线程执行中的访存需求,GPU 设计了复杂的存储层次和访存路径,如图 3 所示.全局内存基于 DRAM,其存储容量最大,一般在数十 GB,用于存放 GPU 程序的输入和输出数据;但其访存延迟比较高.对全局内存的访问一般会经过高速缓存.二级高速缓存(L2 cache)由所有流多处理器共享,容量一般为数 MB.一级高速缓存(L1 cache)位于流多处理器内部,由位于同一个流多处理器上的线程共享,容量一般为数十 KB.最快的存储部件是寄存器.一个流多处理器一般集成了数万个 32 位寄存器.这些寄存器被划分给不同的线程使用.另外,流多处理器上还有两种比较特殊的存储部件.共享内存(shared memory)是一块由编程人员手工控制使用的存储空间,容量一般为数十 KB,可以用于缓存程序执行中频繁访问的数据.而常量缓存(constant cache)可以用来缓存对常量内存的访问.常量内存也位于 DRAM 中,由编译器和驱动程序使用,一般用于保存在整个 GPU 程序生命周期中取值不变的数据.当常量缓存命中时,访问延迟很低.对常量内存的访问往往不经过高速缓存.常量缓存访问的另一个特点是,当一个 warp 内的线程访问常量内存中的相同位置时,这些线程的访问请求会被合并,并通过一次请求获得结果.

2 稀疏感知的算子代码生成

在这一节中,我们介绍所提出的稀疏感知的算子代码生成方法.我们首先介绍我们设计的算子模板.在模板设计中,我们考虑了多种针对 GPU 平台的优化.接下来我们介绍我们基于算子模板设计的中间表示模板,以及基于中间表示模板的分析和稀疏算子代码的生成过程.最后,我们分析了生成稀疏算子代码中的访存优化.我们主要以卷积算子为例,具体说明我们的稀疏算子代码生成过程.

Table 1 Definition of symbols in convolution

表 1 卷积中使用的符号定义

名称	描述
N	输入数据批大小
C	输入数据通道数目
H/W	输入数据高/宽
K	卷积核数目
R/S	卷积核的高/宽
H'/W'	卷积计算结果的高/宽

2.1 算子模板

在 GPU 平台上有多种卷积的计算方法,包括基于矩阵乘法(GEMM)的方法和直接卷积等.基于 GEMM 的方法需要对输入数据进行变换,这一过程需要对输入数据进行复制,增大了访存需求.考虑到通用性和稀疏计

算的计算密度问题,我们基于直接卷积的方式设计我们的卷积算子模板.我们首先定义要用到的符号.我们分别使用 *Input*,*Weight* 和 *Output* 表示输入数据、卷积参数和卷积输出,其中 $Input \in R^{N*C*H*W}$,$Weight \in R^{K*C*R*S}$,$Output \in R^{N*K*H'*W'}$.各个维度的含义见表 1.维度的大小用大写字母表示,而每个维度上的循环变量用相应的小写字母表示.同时,我们使用下标表示某个具体位置的元素,如 $Input_{n,c,h,w}$.

算法 1. 卷积模板代码

输入: *Input*,卷积输入数据; *Weight*,卷积参数
输出: *Output*,卷积计算结果

```
根据 blockIdx 和 threadIdx 计算每个线程计算任务的起始位置(Nt,Kt,Ht,Wt)
将输入数据载入共享内存中的数组 sInput
for n ← 0 to BN-1 do
 for h ← 0 to BH-1 do
  for w ← 0 to BW-1 do
   sum[0...BK-1] = 0
   for c ← 0 to C-1 do
    for r ← 0 to R-1 do
     for s ← 0 to S-1 do
      i = sInput[f(n, c, h, w, r, s, threadIdx)]      //从共享内存中读取相应的输入
      for k ← 0 to BK-1 do
       sum[k] += Weight_{k,c,r,s} □ i      //卷积参数与输入数据相乘,结果进行累加
     end for
    end for
   end for
  end for
  Output[n +Nt][0 +Kt][h +Ht][w +Wt] = sum[0]      //将计算结果写回全局内存
  Output[n +Nt][1 +Kt][h +Ht][w +Wt] = sum[1]
  ...
  Output[n +Nt][BK-1 +Kt][h +Ht][w +Wt] = sum[BK−1]
 end for
end for
end for
```

我们沿卷积输出结果 *Output* 的 4 个维度进行任务划分.每个 GPU 线程负责计算 *BN*□*BH*□*BW*□*BK* 大小的输出结果,并且相邻线程计算卷积输出中相邻位置的结果,其中 *BN* 表示数据批维度每个线程计算任务的大小,BK 表示卷积核维度每个线程计算任务大小,*BH* 和 *BW* 分别表示在高和宽维度每个线程计算任务大小.这样的任务划分方式可以挖掘卷积计算过程中的数据重用机会.首先,计算 *Output* 上相邻位置的结果时,使用的 *Input* 中的输入数据之间可能存在重叠.这带来了在不同线程之间重用输入数据的机会.我们利用共享内存实现在计算中对输入数据的复用.一个线程块内的线程计算出需要使用的输入数据区域,并共同将这部分数据从全局内存中读出,写入共享内存.在后续计算中,每个线程根据自己的线程编号,从共享内存中读取相应的数据.这样可以降低计算过程中访问输入数据的延迟.其次,计算 *Output* 不同位置的结果时,使用的卷积参数是相同的.所以 *Weight* 也可以被不同的线程复用.由于卷积参数的取值在整个计算过程中是固定不变的,所以卷积参数可以存储在常量内存中.同时,在计算时,不同的线程会同时访问相同的参数进行计算.因此可以利用常量内存访问的合并机制,降低访存需求.算法 1 展示了我们设计的卷积模板代码,其中 *blockIdx* 和 *threadIdx* 分别表示每个线程所在的线程块和线程在线程块内的位置信息,结合任务划分信息,每个线程可以计算出自己负责的计算区域.

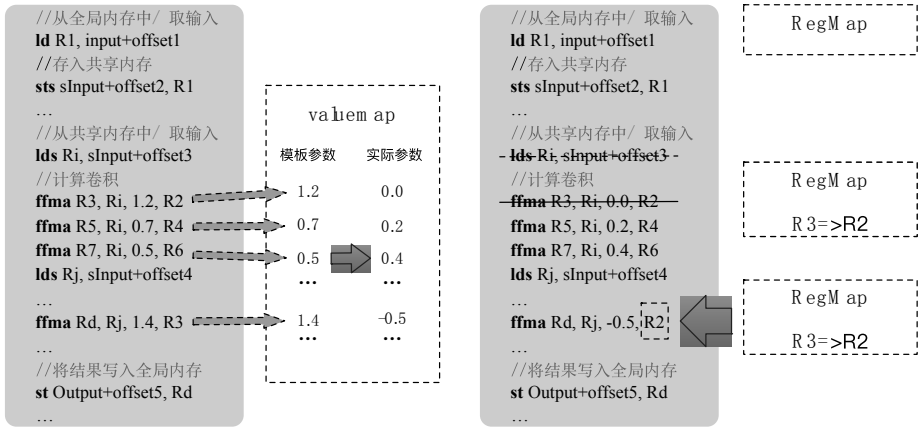

Fig.4 Example of intermediate representation template and sparse code generation

图 4 中间表示模板与稀疏代码生成示意图

2.2 中间表示模板

在算法 1 中,每一个卷积参数 $W_{k,c,r,s}$ 与相应的输入数据之间进行乘法,乘法的结果累加起来,最终得到卷积计算的结果.在稀疏情况下,这个过程中存在大量的冗余操作.具体来说,当卷积参数具有稀疏性时,会存在 $W_{k,c,r,s}$=0 的情况.此时,这个参数参与的乘法运算便是冗余的,因为删除这些乘法指令并不会影响最终的结果.另外,由稀疏性带来的冗余操作不仅局限于卷积参数直接参与的计算指令.对于输入元素 $Input_{n,c,h,w}$,如果与它进行计算的所有参数取值都为 0,那么对它的访问也同样是冗余的,相应的访存指令可以删除.同样地,如果计算某个卷积输出结果涉及的全部参数取值均为 0,那么对它的存储操作也是冗余的.因此,如果从稠密的卷积计算中识别并删除由稀疏参数造成的冗余指令,便可以获得针对稀疏参数的卷积代码.我们的稀疏算子代码生成方法就基于删除冗余操作指令的思想.

为了实现冗余指令的识别和删除,我们需要解决以下几个问题.第一,需要建立一种算子的中间表示,这种中间表示能够支持分析模型参数与计算指令的对应关系;其次,中间表示需要支持快速准确的指令依赖分析;最后,由于相同的算子可能被用在不同的神经网络中,对应不同的稀疏参数取值,所以该中间表示应该能够作为模板,适配不同的稀疏参数,并生成相应的代码.

面对上面的问题,我们设计了一种算子中间表示.该中间表示基于英伟达 PTX(Parallel Thread Execution)形式.PTX 是英伟达设计的一套虚拟的体系结构和指令集,而且是 GPU 程序编译过程中的中间结果.图 5 展示了 GPU 程序使用英伟达 nvcc 编译器的编译过程以及各阶段输出结果的形式.我们下面分别介绍该算子中间表示如何解决上面的三个问题.首先是建立每个模型参数与依赖的指令序列之间映射的问题.在算法 1 中,每

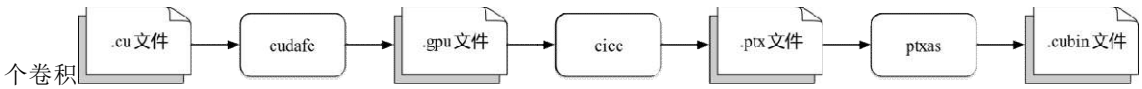

Fig.5 nvcc compilation stages and imtermediate results

图 5 nvcc 编译阶段与中间结果示意图

参数可以由作为下标的循环变量 k,c,r,s 唯一确定,而同时循环变量也可以确定一次迭代中具体的计算.所以一个直接的策略就是通过循环变量将卷积涉及的乘累加(fused multiply-add, FMA)指令与对应的卷积参数联系起来.但是,在后续编译器的指令调度过程中,编译器可能会对指令的顺序进行调整.所以在最终生成的代码中,乘累加指令出现的顺序与循环遍历的顺序是不一致的.我们需要设计新的机制将卷积参数与对应的指令关联

起来.考虑到模型参数和 GPU 指令两方面的特点,我们采用了一种基于参数取值的关联机制.GPU 的代数运算指令通常有一种直接编码立即数操作数的形式,例如图 4 展示了编码了立即数的单精度浮点数乘累加指令 FFMA, 指令 FFMA R3, Ri, 1.2, R2 表示执行 R3=Ri*1.2+R2 的计算,作为操作数之一的立即数 1.2 会直接编码在指令中.这种形式的指令能够将参与计算的数据与指令直接关联起来.在此之上,我们再建立卷积参数位置与参数取值之间的映射.具体来说,我们随机生成取值不相同且非 0 的**模板参数**,取值不同保证了由参数取值可以唯一确定参数的位置.在编译卷积算子模板的过程中,我们将模板参数作为立即数提供给编译器,同时,我们将涉及模型参数的循环进行展开,使得这些参数被编码到相应的计算指令中.这样,通过解析指令中立即数的取值,便可以唯一确定参数的位置.同时,循环展开消除了循环控制相关指令,也给编译器更大的空间进行指令调度等优化,有利于编译器生成性能更好的代码.由于循环展开会导致指令数目增多,使得编译生成的文件增大.我们在 3.7 节对编译生成的文件体积,以及循环展开对运行时指令获取的影响进行了分析.

PTX 也适合进行高效的指令间依赖关系分析.PTX 程序符合静态单赋值(SSA)的形式,且使用虚拟寄存器,PTX 代码中使用的每个寄存器都有唯一的一次定值.因此建立寄存器之间的依赖关系十分简单.通过解析指令中的寄存器的名字,我们可以高效地追踪指令之间的依赖关系.最后,由于我们生成的是不含 0 的模板参数,所以每个模型参数可能涉及的指令都被保留在了中间表示模板中.所以不论具体的稀疏参数如何,该中间表示都能用于生成对应的稀疏程序.图 4 展示了对应的模板中间表示的例子.

2.3 稀疏算子代码生成

基于算子的中间表示模板,我们设计了结合具体的稀疏参数,生成对应稀疏卷积代码的算法.稀疏算子代码生成算法主要完成两方面的工作.首先是将模板参数替换为真实的稀疏参数,保证计算过程中使用参数的正确性.第二是识别由稀疏参数带来的冗余指令,并对其进行删除;同时,还需要维护由于删除指令导致的寄存器依赖关系的变化,以保证程序的正确性.以上工作通过对 PTX 中间表示模板的遍历完成,该算法的伪代码如算法 2 所示.

在对指令的遍历过程中,算法使用了 2 个哈希表记录相关状态.*valueMap* 用于将模板参数取值映射到真实的稀疏卷积参数.*regMap* 用于处理由于冗余指令删除带来的寄存器依赖关系的变化,它记录了等价寄存器的映射关系.对于中间表示中的每一条 PTX 指令,算法首先通过解析识别指令的具体类型和各个组成部分,这由字符串匹配完成.对于乘累加指令 FFMA,我们首先检查是否需要修改指令的源寄存器,并通过查询 *valueMap* 对指令中的立即数进行替换.例如,对图 4 中的第一条 FFMA 指令 FFMA R3, Ri, 1.2, R2,经过查询 *valueMap*,模板参数 1.2 被替换为真实参数 0,并且由于 *regMap* 为空,不需要进行源寄存器替换.同时,由于替换后立即数操作数为 0,这条指令实际上进行了 R3=R2 的操作,因此 R3 和 R2 实际上是等价的.如果能够将后续对 R3 寄存器的引用替换为对 R2 的引用,那么这条 FFMA 指令便可以删除(图 4 中使用横线标注).我们在 *regMap* 中记录 R3=>R2 的映射关系,并删除这条指令.由于稀疏卷积参数涉及的指令为 FFMA 指令,被等价映射的寄存器为保存卷积计算临时结果的寄存器,这些寄存器会被后面的 FFMA 指令作为源操作数使用.由于对寄存器的引用一定出现在寄存器的定值之后,因此我们通过查询当前 *regMap* 中记录的寄存器映射关系,便可以实现相应的修改.当在后面的指令中遇到对 R3 的引用时,例如图 4 中的 FFMA Rd, Rj, 1.4, R3,由于 *regMap* 中已经记录了 R3 到 R2 的映射,因此,可以正确的将对 R3 的引用修改为对 R2 的引用.通过使用 *regMap* 记录寄存器的映射关系,我们可以删除使用 0 的 FFMA 计算指令.另外,寄存器映射关系也会影响存储指令.因为乘累加指令的目的寄存器保存了卷积计算的结果,因此会作为存储指令的源操作数.对于存储指令 ST,我们也检查其源寄存器(保存待写出值的寄存器)是否被重命名,并进行必要的修改.算法没有对可能存在的冗余访存指令进行显式的删除操作,这主要是基于对 GPU 的访存指令特点的考虑.GPU 通常支持多种宽度不同的访存指令,例如 32 位,64 位等,并且不同宽度的指令能够实现的访存吞吐量不同[28] .当多个元素的访问被打包到同一条访存指令中时,删除其中某个元素的访问,并将其拆分为多条宽度更小的访存指令可能会影响访存吞吐量.

因此,我们没有对访存指令进行直接删除,而是交给后续的汇编器 ptxas 决定,是否对涉及未被引用操作数的访存指令进行修改.

算法 2. 稀疏算子代码生成算法

输入: *ptxTemp*,中间表示模板; *paramTemp*,模板参数; *paramReal*,真实稀疏参数
输出: *ptxCode*,为真实稀疏参数生成的 PTX 代码

```
初始化 2 个 Map valueMap, regMap 为空
for valTemp in paramTemp, valReal in paramReal do
  valueMap[valTemp] = valReal       //建立模板参数与真实参数的映射
end for
for inst in ptxTemp do
  if inst.op == FFMA then       // rd=rs+rt*imme
    if regMap.find(inst.rt) != null then
      inst.rt = regMap[inst.rt]       //将源寄存器 rt 替换为其他寄存器
    end if
    if regMap.find(inst.rs) != null then
      inst.rs = regMap[inst.rs]       //将源寄存器 rs 替换为其他寄存器
    end if
    if valueMap.find(inst.imme) != null then
      inst.imme = valueMap[inst.imme]       //进行参数替换
      if inst.imme == 0 then
        regMap[inst.rd] = inst.rs       //建立寄存器映射
        continue
      end if
    end if
    if regMap.find(inst.rd) != null then
      regMap.remove(inst.rd)
    end if
  else if inst.op == ST then       // rd[offset]=rs
    if regMap.find(inst.rs)! = null then
      inst.rs = regMap[inst.rs]       //将源寄存器 rs 替换为其他寄存器
    end if
  end if
  ptxCode.append(inst)
end for
```

我们设计的稀疏代码生成算法的复杂度是 $O(n)$,其中 n 为中间表示模板的指令条数.对于每条指令,我们进行解析和哈希表查找的操作.解析基于字符串匹配进行,而哈希表的查找也可以快速实现.综合来说,稀疏代码生成算法的时间效率较好.

完成冗余指令删除后,我们就获得了针对稀疏卷积参数的 PTX 程序.接下来该 PTX 程序经过汇编器 ptxas 优化(图 5),获得最终的二进制代码.由于 PTX 面向英伟达设计的虚拟 GPU 体系结构,基于 PTX 的中间表示可以为不同体系结构的 GPU 生成最终的二进制代码[1].另外,ptxas 会负责执行与具体 GPU 体系结构相关的优化,包括寄存器分配和指令调度等,所以我们生成的稀疏算子程序也能够受益于这些优化.最后,由于 PTX 中间表示模板可以在不同的稀疏模型参数上复用,相当于降低了获得中间表示模板的开销.我们通过实验发现,图 5 中的前两个编译阶段(从 cu 文件到 ptx 文件)占用了编译过程的大部分时间(90%以上),所以选用 PTX 作为中间表示层次可以复用开销最大的编译过程,降低后续生成具体稀疏算子代码的时间开销.在实验中,ptxas 编译生成的 ptx 代码所需的时间在几十到几百毫秒.

[1] 通过为 ptxas 汇编器提供不同的参数,指定具体的目标 GPU 平台

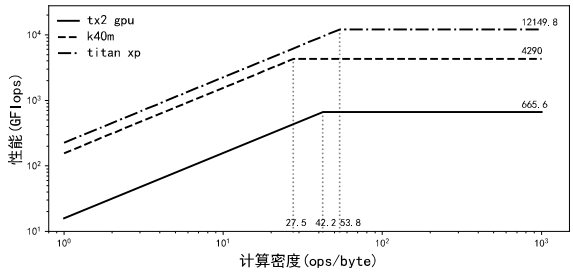

Fig.6 Roofline model for 3 GPUs

图 6 3 个 GPU 的 roofline 模型图

2.4 访存优化

尽管 GPU 能够提供强大的峰值计算性能,但达到峰值性能需要程序具有较高的计算访存比.访存密集型的应用很容易受到访存带宽的限制,难以实现满意的性能.我们通过 roofline 模型[29] 分析了不同计算密度下,GPU 程序的性能上限,如图 6 所示.横轴表示不同的计算密度,单位是操作数/字节,表示从全局内存访问的每字节数据所参与的操作数目.纵轴表示性能,使用每秒钟可执行的操作数目衡量.图中的折线表示在每个计算密度下能够达到的峰值性能.如果一个程序的计算密度在折线拐点的左侧,则该程序的峰值性能是访存受限的;如果程序位于拐点右侧,则该程序是计算受限的.我们选取了 3 种不同的 GPU,并标注了每个 GPU 的峰值性能和达到峰值性能所需要的计算密度的下限.其中,Tesla K40m 和 Titan Xp 是工作站级别的 GPU,而 Jetson TX2 是面向终端设备的 GPU.可以看到,计算能力更强的 GPU,其对计算密度的要求往往也越高.

我们对稀疏参数卷积的计算密度进行了分析.假设卷积参数 *Weight* 的稀疏程度(取值为 0 的参数在全部参数中所占的比例)为 p.对于经过非结构化剪枝的稀疏参数,取值为 0 的元素的分布没有规律,我们认为对任意位置*(k, c, r, s)*,其取值为 0 的概率 $P(Weight_{k,c,r,s}=0) = p$,则计算密度 OI(operational intensity)可以用公式 1 表示.

$$OI = \frac{N * H' * W' * K * (2 * R * S * (1-p) * C)}{sizeof(T) * \left(N * C * H * W + N * K * H' * W' + R * S * C * K * (1-p)\right)}$$

$$= \frac{2}{sizeof(T)} * \frac{1-p}{\frac{H * W}{H' * W' * K * R * S} + \frac{1}{C * R * S} + \frac{1-p}{N * H' * W'}} \#(1)$$

其中 T 表示输入数据、输出结果和模型参数的数据类型,*sizeof(T)*为常数.当稀疏程度 $p=0$ 时,公式 1 对应稠密卷积的情况.很容易从公式 1 中看出,随着稀疏程度 p 增大,计算密度 OI 会逐渐降低,使得访存容易成为瓶颈.因此在稀疏场景下,优化 GPU 程序的访存具有重要价值.

与一般的 GPU 稀疏程序优化相比,由稀疏代码生成算法生成的卷积程序采用了两种技术优化访存.首先,我们充分利用了神经网络稀疏参数在编译时取值确定的特点,将稀疏模型参数和输入数据存储在不同的存储空间,并通过不同的访问路径进行访问.直接编码在 PTX 指令中的常数会存储在 GPU 常量内存中,而卷积输入数据则位于全局内存中.一方面,我们充分利用了 GPU 提供的多种访存路径(图 3),输入数据由全局内存经过高速缓存达寄存器,并被存储在共享内存中反复使用.计算结果通过高速缓存写入全局内存.而取值固定的模型参数存储在常量内存中,经过片上的常量缓存进行访问.另一方面,对模型参数和输入数据使用不同的访存路径,避免了彼此之间在高速缓存的干扰.以 Tesla K40m GPU 为例,每个流多处理器对应的二级高速缓存空间一般仅有 100KB.假设有 1024 个线程活跃,则每个线程平均只有 100 字节的高速缓存空间,在单精度浮点数的

情况下,对应 25 个浮点数.算法 1 中,在两次对相同的模型参数 $W_{k,c,r,s}$ 的访问之间,我们需要访问 $channel□R□S$ 个输入数据元素[1].在真实的神经网络中,这个规模远超过了每个线程对应的高速缓存的空间.因此,如果对模型参数和输入数据使用相同的访问路径,将会损害数据局部性,造成访存吞吐量的降低.

第二个优化来源于我们对非 0 元素位置信息的编码方式.一般的稀疏程序采用某种稀疏格式表达稀疏数据,非 0 元素的取值与位置信息被分别存储.例如流行的 CSR 格式,非 0 元素的值存储在数组 *values* 中;每个非 0 元素的列号被单独存储在一个数组 *colIdx* 中,同时,每一行对应的非 0 元素在 *values* 数组中的起始位置被记录在数组 *rowPtr* 中.公式 2 计算了采用 CSR 格式时,位置信息占用的空间 $S_{location}$.

$$S_{location} = S_{colIdx} + S_{rowPtr}$$
$$= K * C * R * S * (1-p) * sizeof(T_{colIdx}) + (m+1) * sizeof(T_{rowPtr}) \#(2)\#$$

其中 m 表示稀疏矩阵的行数,具体取值取决于将 *Weight* 展开成矩阵的方式.非 0 元素取值所占空间 S_{value} 可用公式 3 计算.

$$S_{value} = K * C * R * S * (1-p) * sizeof(T_{value}) \#\#(3)$$

为了衡量位置信息所占用空间的比重,我们计算 $S_{location}$ 与 S_{value} 的比值,如公式 4 所示.

$$\frac{S_{location}}{S_{value}} = \frac{sizeof(T_{colIdx})}{sizeof(T_{value})} + \frac{(m+1) * sizeof(T_{rowPtr})}{K * C * R * S * (1-p) * sizeof(T_{value})} \#\#(4)$$

第一项取决于存储非 0 元素列号与存储非 0 元素取值所使用的数据类型.在实际的神经网络模型中,稀疏矩阵的列的宽度可能超过 256,所以非 0 元素列号至少需要 16 比特才能表示.对于每行的起始位置,其类型取决于整个矩阵中非 0 元素的数目,即$(1-p) * K * C * R * S$,一般也至少需要 16 比特表示.对于非 0 元素的取值,我们一般使用 32 比特单精度浮点数表示,则公式 4 中的第一项取值至少为 0.5.因此,与非 0 元素取值所占用的空间相比,位置信息所占用的空间至少会超出其一半的大小.而在当前 GPU 上的稀疏计算库[23] 中,这一项的值为 1.因此,非零元素位置信息所占用的空间是不能忽略的.在论文[25] [26] 中也提到了存储稀疏模型时,位置信息会带来额外的开销,并对性能造成负面影响.

我们生成的稀疏算子程序能够避免位置信息的访存开销.由于在编译时,我们展开了与模型参数相关的循环,并在稀疏代码生成阶段将真实的稀疏模型参数取值编码到了指令中.在生成的代码中,每个参数已经按照自己的位置与对应的输入数据进行计算.所以生成的稀疏程序不再需要额外的位置信息,避免了程序运行时对位置信息的访问,降低了访存需求.

Table 2 Convolution operators used in experiments

表 2 实验中使用的卷积算子信息

编号	H*W	C	K	R*S	卷积参数数目	计算量/百万次	来源模型	数据集
1	24*24	1	20	5*5	500	36.8	LeNet5	MNIST
2	8*8	20	50	5*5	25000	204.8	LeNet5	MNIST
3	32*32	3	32	5*5	2400	314.5	AexNet	Cifar10
4	16*16	32	32	5*5	25600	838.8	AlexNet	Cifar10
5	8*8	32	64	5*5	51200	419.4	AlexNet	Cifar10
6	56*56	64	64	3*3	36864	14797.5	ResNet	ImageNet
7	28*28	128	128	3*3	147456	14797.5	ResNet	ImageNet
8	224*224	3	64	3*3	1728	11098.1	VGG16	ImageNet
9	224*224	64	64	3*3	36864	236760.1	VGG16	ImageNet
10	112*112	64	128	3*3	73728	118380	VGG16	ImageNet

3 实验分析

在本节中,我们希望通过实验回答与本文所提出的稀疏感知的代码生成方法相关的 3 个关键问题.首先是

[1] 为了隐藏全局内存的访问延迟,在执行当前计算时会同时读取下一轮计算需要的数据

该方法的有效性如何,即本文的方法是否能够改进稀疏卷积的计算性能.第二个问题与该方法的稀疏适应性相关,即在不同稀疏程度下,本文方法生成的稀疏卷积代码性能如何变化?最后一个问题关注本方法的开销.为了进行冗余指令删除以及使用常量内存等优化,我们对代码进行了循环展开,使得指令数目显著增加.我们希望通过实验确定指令数目膨胀对代码体积和指令访问产生了什么样的影响.

我们通过实验逐一回答上面的问题.在 3.1 节中,我们首先介绍实验配置,包括对比方法、实验使用的卷积算子信息以及实验平台等.3.2-3.5 节从不同角度说明本文方法的有效性.其中 3.2 和 3.3 节通过在实验卷积算子上的性能对比和分析,展示相对其他方法的性能优势.3.2 节主要关注稠密计算方法和其他非结构化稀疏优化方法,而 3.3 节探索了与结构化剪枝方法的性能对比问题.3.4 和 3.5 节分别对冗余指令删除和访存路径优化这两个主要的优化技术的直接效果进行了分析.3.6 节回答稀疏适应性的问题,我们评估了不同稀疏程度下本方法生成代码的性能,并与其他方法进行了对比分析.3.7 节针对开销问题,具体分析了生成代码体积的变化和对指令访问的影响.

3.1 实验配置

我们在一台配有英伟达 Tesla K40m 的服务器上进行实验.Tesla K40m 具有 15 个流多处理器,显存容量为 12GB.默认情况下,每个流多处理器上有 16KB 的一级高速缓存和 48KB 的共享内存.另外,每个流多处理器还有 8KB 的常量缓存,用于缓存对卷积参数的访问.所有流多处理器共享容量为 1536KB 的二级高速缓存.

为了尽可能覆盖在卷积神经网络中使用的各种参数规模和计算量不同的卷积算子,我们从流行的卷积神经网络中选择了 10 个有代表性的算子,作为实验中进行优化的对象.实验中使用的算子的具体信息如表 2 所示.我们在表中列出了每一个算子的具体信息.可以看到,我们选取的算子覆盖了仅需要 36M 计算的简单算子,也包含需要 236G 计算的复杂算子.参数最少的算子仅包含 500 个参数,而最大的算子包含超过 14 万个参数.在实验使用的批大小为 64 和 1.在执行生成的代码时,对于每一个算子,我们根据输出张量形状和任务划分信息计算出加载 CUDA kernel 时的线程块与线程的形状.

为了说明我们提出的方法的有效性,我们选取了多种在 GPU 上执行卷积神经网络的方法作为对比对象.这些对比对象可以分为两大类.第一类是不利用参数中稀疏性的方法,包括 cuDNN[15] 和 cuBLAS[16] .cuDNN 内封装了多种优化的卷积实现,并且会在运行时根据输入数据规模和具体平台等信息选择最优实现.cuBLAS 中的 Sgemm 被用来实现卷积,我们使用了基于矩阵展开的卷积方法,并将第一步输入张量展开和第二步矩阵乘法的总时间作为 cuBLAS 方法的执行时间.另外,虽然结构化剪枝一般并不造成稀疏计算的问题,考虑到其也是一类利用神经网络参数冗余性,加速神经网络执行的方法,我们也将其选做对比方法,并使用 cuDNN 实现结构化剪枝后的计算,我们选择了通道剪枝[18] 和卷积核剪枝[19] 两个结构化剪枝方法进行对比.对于通道剪枝,我们比较删除卷积操作一半输入通道的情况;对于卷积核剪枝,比较删除一半卷积核的情况.此外,由于卷积核数目减半会导致卷积结果的通道数为之前的一半,使得后续卷积的输入通道数目减少.因此我们也考虑了同时将输入通道和卷积核数目减半的情况.虽然不考虑利用参数的稀疏性,但 cuDNN 和 cuBLAS 经过专家的精心调优,可以实现非常好的性能.另一类方法是与本文工作目的相同,同样可以进行稀疏神经网络计算的方法.我们选择了 Escoin[24] 和 cuSPARSE[23] 进行对比.对于 cuSPARSE,我们使用与 cuBLAS 类似的方法,基于 cuSPARSE 中的稀疏-稠密矩阵乘法实现卷积,并考虑输入张量展开的时间.Escoin 也提供了稀疏卷积的 GPU 优化实现,我们从 github 上下载了它的代码.

实验中使用的 cuDNN,cuBLAS 和 cuSPARSE 的版本分别为 7.6.5,9.0,和 9.0.Escoin 使用了当前的最新版本(github commit e89275f961847319e6b0331f0dc163a3293fad4c).实验在英伟达 GPU 编程环境 CUDA 9.0 下进行,使用 nvcc 9.0 编译器编译代码.

3.2 与稀疏和稠密卷积方法的对比分析

在这一节中,我们分析我们的优化方法对其他可用于稀疏卷积优化执行的方法的性能收益.大部分模型剪枝工作没有详细给出每一层的压缩比例,考虑到已有模型压缩工作在实验中所涉及的模型上均实现了超过 10 倍的压缩,我们在本节中使用 0.9 的稀疏程度进行性能测试.在 3.6 节中我们进一步分析了不同稀疏程度下的性能.我们使用在每个算子上执行 10 次的平均时间表示每个方法的性能.图 7 和图 8 分别展示了批大小为 64 和批大小为 1 时的实验结果,我们计算了本文方法生成的算子对其他方法的加速比.数值大于 1 表示我们的方法性能更优,小于 1 表示对比方法的性能更好.

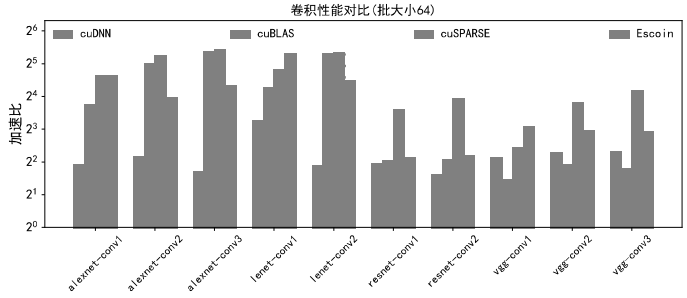

Fig.7 Performance comparison with baseline methods under batchsize=64

图 7 批大小为 64 时不同方法的性能对比

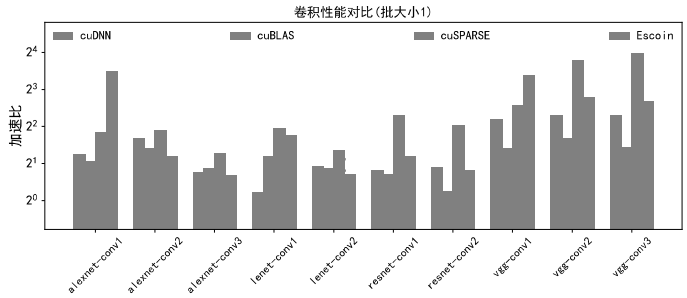

Fig.8 Performance comparison with baseline methods under batchsize=1

图 8 批大小为 1 时不同方法的性能对比

我们从几个不同的角度对实验结果进行分析.从不同的对比方法看,首先,在所有的对比方法中,cuSPARSE 几乎在所有的卷积算子上的性能都是最差的,这反映了对 GPU 平台上的稀疏计算进行优化的难度.即使在 0.9 的稀疏程度下,其性能也难以达到精心调优的稠密计算库的水平.尽管 cuBLAS 和 cuDNN 不考虑稀疏参数带来的优化机会,但规则的访存模式和细致的手工调优仍然能够实现很好的性能.与对比方法相比,我们生成的稀疏算子代码在所有的卷积上都展现了超过其他所有方法的性能.具体来说,在批大小为 64 时,相对于 cuBLAS,cuDNN,cuSPARSE 和 Escoin,我们的方法分别实现了 2.8−41.4 倍,3.1−9.6 倍,5.5−43.2 倍和 4.4-39.5 倍的加速比;在批大小为 1 时,相对 cuBLAS,cuDNN,cuSPARSE 和 Escoin 的加速比范围分别为 1.2-3.2 倍,1.2-4.9 倍,2.2-14.6 倍和 1.6-11.2 倍.考虑到 cuBLAS 和 cuSPARSE 方法基于矩阵展开实现,将输入张量展开的时间包含在内.为了更细致地比较计算部分的性能,我们进一步对 cuBLAS 和 cuSPARSE 方法的性能进行了分解,计算输入张量展开部分在总时间中所占的比例.对于 cuBLAS,在批大小为 64 和批大小为 1 时,展开输入

张量的时间在 10 个算子上平均所占比例分别为 3.16%和 6.22%.具体来看,当批大小为 64 时,占比为 0.11%-12.69%.批大小为 1 时,占比总体来看有所上升,为 0.32%-16.91%.对于 cuSPARSE,由于其第二步矩阵乘法的性能显著差于 cuBLAS,因此输入张量展开时间占比更低.批大小为 64 时不超过 9.01%,平均为 1.62%,批大小为 1 时不超过 11.26%,平均为 2.07%.相对于 cuDNN,生成的稀疏算子均获得了显著的加速效果.上面的实验结果说明,相比于现有的稀疏神经网络执行方法,我们所提出的方法能够有效利用压缩后的稀疏模型参数,加速网络执行.

其次,优化效果也与算子本身的特征以及批大小有关.批大小为 64 时,除 lenet-conv1 以外,我们的方法对 cuDNN 的加速效果在不同算子间比较稳定.我们通过 profiling 发现,虽然 cuDNN 对 lenet-conv1 和 lenet-conv2 均采用了 fft 算法,但内部使用了不同的 kernel 实现[1],造成了明显的性能差异.尽管 lenet-conv1 的计算需求仅为 lenet-conv2 的约 18%,但执行时间却是 lenet-conv2 的 1.3 倍.另外,cuBLAS 中的矩阵乘法 kernel 在计算量较大的 5 个算子(2 个 resnet 算子和 3 个 vgg 算子)上的性能显著好于剩余的计算量较小的 5 个算子,平均性能差异可达 12.6 倍.因此我们的方法相对 cuBLAS 的加速效果对于小规模算子更加明显.cuSPARSE 和 Escoin 也有类似的现象,在计算量较大的算子和其他算子上的平均性能有显著的差异.

批大小为 1 时,卷积参数失去了在不同输入数据之间的复用机会.第一次读取卷积参数会导致片上的常量缓存发生缺失,需要从位于片外 DRAM 上的常量内存中进行读取,延迟很高.而当批大小为 64 时,这一开销可以被后续通过常量缓存的加速访问分摊.因此总体来看,批大小为 1 时的性能收益较 64 时有所下降.在 3 个来自 vgg 的算子上,我们的方法取得收益更加明显.这是因为 vgg 算子的计算结果规模很大,经过任务划分后允许更多 GPU 线程并发执行,所以可以通过线程级并行更好地掩盖访存延迟;resnet 算子虽然访问的参数规模与 vgg 算子类似,但由于计算结果规模显著小于 vgg 算子,因此线程级并行机会更少.其他算子虽然卷积参数数目较少,但其输出结果规模也远小于 vgg 算子,所以也难以通过线程间的并行有效掩盖访问延迟.另外,对于 lenet-conv1 算子,cuDNN 采用 implicit gemm[15] 算法替换了 fft,使得 lenet-conv1 的执行时间少于 lenet-conv2,消除了批大小为 64 时 lenet-conv1 与 lenet-conv2 之间的性能倒挂现象.此外,相对于 Escoin,我们的方法在 alexnet-conv1 上的收益非常显著.由于 Escoin 内部根据数据形状、批大小和参数的稀疏程度等信息硬编码了一些算子到具体 kernel 的映射规则,这些规则并不准确,负责 alexnet-conv1 的 kernel 中存在大量非合并的全局内存访问,导致其性能发生了显著降低.

3.3 与结构化剪枝方法的对比分析

在这一节中,我们与基于通道剪枝和卷积核剪枝的结构化剪枝方法进行比较.具体来说,对每一个实验算子,我们考虑了删除一半输入通道(PC, pruning channel)、删除一半卷积核(PF, pruning filter)以及同时删除一半输入通道和一半卷积核(Both)三种情况.由于结构化剪枝产生的是规模更小的稠密参数,我们使用 cuDNN 实现了上面三种剪枝后的卷积,并与我们生成的代码进行了性能对比.由于 PC 和 PF 可以移除卷积中 50%的参数,我们使用了在 0.5 稀疏程度下生成的优化代码进行对比,对于同时删除输入通道和卷积核的情况(Both),其稀疏程度可达 0.75,我们使用 0.8 稀疏程度下的代码对比.

[1] lenet-conv1:cgemm_strided_batched_sm35_ldg_nt_64x8x64x16x16; lenet-conv2:fermiPlusCgemmLDS128_batched

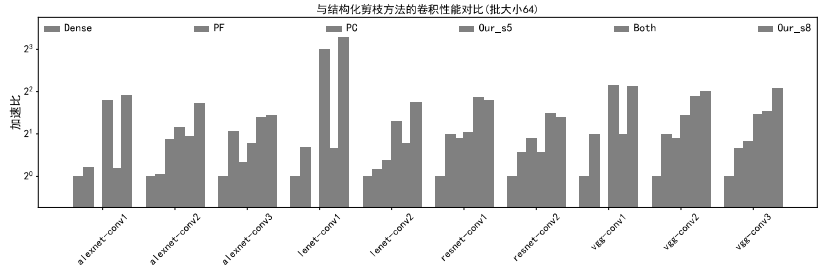

Fig.9(a) Performance comparison with structured pruning methods under batchsize=64

图 9(a) 与其他结构化剪枝方法的性能对比,批大小为 64

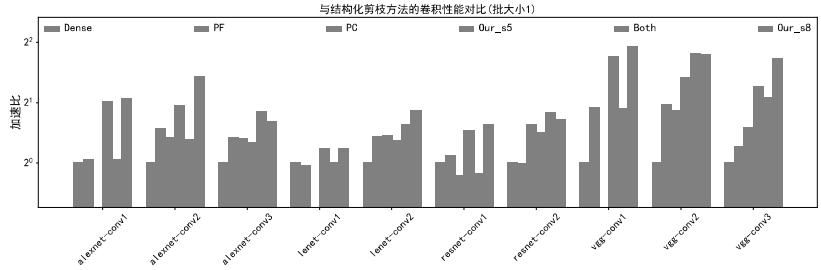

Fig.9(b) Performance comparison with structured pruning methods under batchsize=1

图 9(b) 与其他结构化剪枝方法的性能对比,批大小为 1

图 9 展示了批大小为 64 和批大小为 1 时的实验结果.对于输入通道数目很小且是奇数的算子(alexnet-conv1, lenet-conv1, vgg-conv1),我们不对其输入通道进行删除,因此 PC 跳过了这些算子;同时,由于不删除输入通道,因此 Both 和 PF 在这些算子上性能一致.对于每一个算子,我们将各个方法的性能相对稠密情况下 cuDNN 的性能(Dense)做了归一化,数值大于 1 表示性能好于稠密情况下的 cuDNN.

相对于 PF 与 PC,我们生成的代码在大部分算子上实现了更好的性能.在批大小为 64 时,相对 PC 和 PF 在 10 个算子上分别实现了平均 1.3 倍和 2.1 倍的加速;相对 Both 实现了 2.1 倍的加速.批大小为 1 时结果类似,对 PC,PF 和 Both 的加速比分别为 1.3 倍、2.1 倍和 1.5 倍.我们也发现,由于 cuDNN 对内部实现方法的选择策略等原因,算子在不同剪枝情况下的性能变化与计算量不完全一致.例如,在批大小为 1 时,对于 alexnet-conv2,cuDNN 对 PF 对应的卷积选择使用 fft 实现,而对 PC 和 Both 都使用了 implicit gemm 的方法,另外,通过 nvprof 我们发现,PC 和 Both 实际上执行的单精度浮点操作数目十分接近,我们猜测为了利用预先调优的内部 kernel 实现,cuDNN 可能对 Both 的情况作了数据补齐,导致了冗余计算.对批大小为 1 时的 resnet-conv1 算子,cuDNN 对 PC 和 Both 均使用显式的 im2col 接 GEMM 的方法实现,而对 PF 使用了 implicit gemm 的方法,造成 PC 与 PF 虽然计算量类似,但 PC 性能显著差于 PF;而 Both 虽然计算需求仅为 PF 的一般,但性能仍然不及 PF.

3.4 删除指令数目分析

在这一节中,我们分析冗余指令删除对最终卷积算子性能的影响.表 3 展示了在稀疏程度从 0.1 逐步增加到 0.9 的过程中,生成稀疏算子代码时删除的指令数目和算子获得的加速效果.我们使用每个稀疏算子相对于其稠密版本的加速比衡量其性能改善.我们选取了 3 个有代表性的算子.可以看到,随着稀疏程度增加,中间表示模板中有越来越多的指令被删除,而同时稀疏算子代码的性能也逐渐提升.

我们又进一步比较了在基于 PTX 的中间表示中删除的指令数目与在最终机器指令中减少的指令数目.由于 PTX 中的一条 FFMA 指令对应于机器代码中的一条乘累加机器指令,如果在稀疏情况下,减少的机器指令数目多于 PTX 指令数目,说明 ptxas 汇编器对生成的 PTX 稀疏算子代码进行了额外的优化,从中进一步消除了其他的冗余指令,例如访存指令等.我们发现,在实验的 10 个算子中,减少的机器指令数目平均为删除的 PTX 指令数的 1.35−1.75 倍,证明了我们生成的 PTX 稀疏算子代码能够支持汇编器进行进一步的冗余指令删除.

Table 3 Redundant instruction number and speedups under various sparsity levels

表 3 不同稀疏程度下删除的冗余指令数目和加速比

稀疏程度		0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
删除指令数目	AlexNet-conv1	490	974	1344	1792	2330	2784	3222	3730	4220
	ResNet-conv1	7444	14718	22194	29522	36990	44318	51776	59254	66330
	VGG-conv2	7432	14790	22210	29434	36766	44098	51536	59028	66238
加速比	AlexNet-conv1	1.000	1.063	1.133	1.133	1.308	1.417	1.417	1.545	1.545
	ResNet-conv1	1.035	1.124	1.189	1.300	1.470	1.697	2.049	2.533	2.826
	VGG-conv2	1.066	1.140	1.445	1.927	2.457	3.147	3.387	3.661	4.471

3.5 访存优化分析

在这一节中,我们对稀疏卷积代码生成方法中的访存优化技术的直接效果进行评估.由于英伟达 GPU 上的性能剖析工具 nvprof 不支持直接测量常量内存和常量缓存的访问情况,我们使用两个其他指标间接说明优化的效果.首先,我们测试全局内存的平均访问吞吐量.由于对稀疏参数的访问不再经过全局内存,与位于全局内存中的输入数据使用了不同的访问路径,避免了相互之间的干扰,因此全局内存的访问吞吐量可以获得提升.另外,我们也测量了 DRAM 的平均访问吞吐量,观察同时使用全局内存和常量内存对 DRAM 吞吐量的改进.

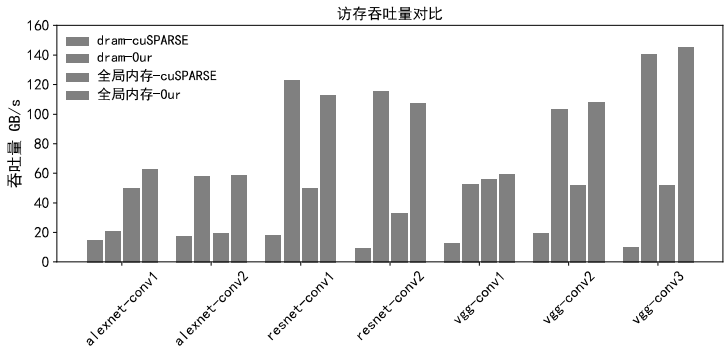

Fig.10 Comparison on memory throughput

图 10 访存吞吐量对比

我们通过与基于 cuSPARSE 的卷积实现进行比较,说明访存优化的效果.图 10 展示了实验结果.可以看到,生成的稀疏算子在所有的卷积中都实现了更好的 DRAM 和全局内存访问吞吐量.其中,稀疏算子 DRAM 吞吐量是 cuSPARSE 的 1.4−14 倍,全局内存访问吞吐量是 cuSPARSE 的 1.1−3.2 倍.

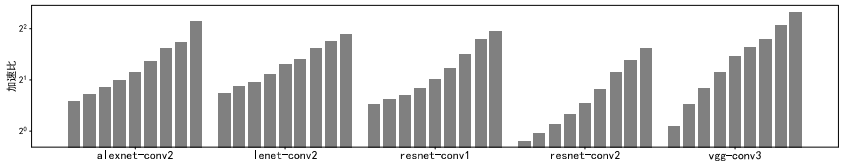

Fig.11(a) Speedups over cuDNN under various sparsity levels with batchsize=64

图 11(a) 不同稀疏程度下对 cuDNN 的加速比,批大小为 64

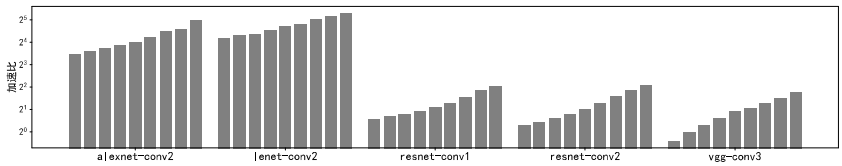

Fig.11(b) Speedups over cuBLAS under various sparsity levels with batchsize=64

图 11(b) 不同稀疏程度下对 cuBLAS 的加速比,批大小为 64

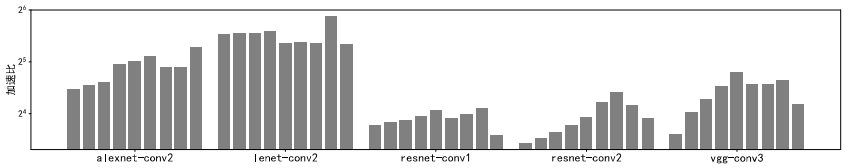

Fig.11(c) Speedups over cuSPARSE under various sparsity levels with batchsize=64

图 11(c) 不同稀疏程度下对 cuSPARSE 的加速比,批大小为 64

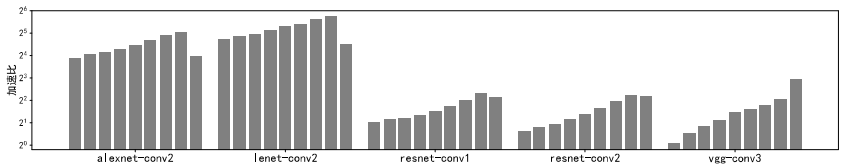

Fig.11(d) Speedups over Escoin under various sparsity levels with batchsize=64

图 11(d) 不同稀疏程度下对 Escoin 的加速比,批大小为 64

3.6 稀疏程度对性能的影响

这一节通过实验研究本文稀疏卷积代码生成方法对不同稀疏程度的适应性.由于同一个算子可能在同一个神经网络的不同位置以及不同神经网络中使用,因此可能对应不同稀疏程度的模型参数.为了进一步评估在不同稀疏程度下,我们的方法将稀疏性转化为性能收益的能力,我们随机生成了稀疏程度从 0.1 到 0.9 的 9 种稀疏参数,之后分别进行稀疏优化,获得对应不同稀疏程度的算子代码.之后,我们计算不同稀疏程度下,每个算子对 cuDNN,cuBLAS,cuSPARSE 和 Escoin 的加速比.图 11 展示了实验结果.由于空间限制,我们仅选择 5 个算子作为代表,其他算子趋势类似,其中 alexnet-conv2 和 lenet-conv2 的计算量较小,剩余 3 个算子计算量较大.

从图 11(a)中可以看出,随着稀疏程度的增加,我们生成的稀疏算子相对 cuDNN 的性能收益越明显.生成的稀疏算子的性能随稀疏程度的增加逐渐提升,表明我们所提出的优化方法有能力对不同的稀疏程度发挥作

用.对于批大小为 64 的情况,在 0.1 的稀疏程度下,算子可以接近或超过 cuDNN 的性能.在稀疏程度达到 0.5 时,相对 cuDNN 可以获得至少 1.5 倍的加速.如果批大小为 1,稀疏程度在 0.1-0.9 之间时,我们的方法相对 cuDNN 可以实现 0.7-4.9 倍的性能收益.

对于 cuBLAS 也有类似的结果,批大小为 64 时,在稀疏程度达到 0.2 的情况下,生成的代码的性能可以超过或与 cuBLAS 持平,随稀疏程度增加,获得的性能收益也越明显.在稀疏程度达到 0.5 时,在全部算子上可以获得平均 10.4 倍的加速.而当批大小为 1 时,获得的加速比在 0.6-3.2 倍的范围内.另外,由于 cuBLAS 在计算量不同的 kernel 上的性能存在显著差异,在 alexnet-conv2 和 lenet-conv2 上,我们的方法在很低的稀疏程度下就取得了明显的性能收益.

对于其他的稀疏计算方法,相对于 cuSPARSE 和 Escoin,生成的稀疏卷积算子展示出了更明显的性能优势.批大小为 64 时,稀疏算子在计算量较小的卷积上展示出了更好的加速效果.这一方面得益于我们的卷积模板更加高效;同时,由于这些卷积的代码长度较短,循环展开后编译器能够更有效地进行指令调度等优化.另一方面,cuSPARSE 和 Escoin 对小卷积的优化不足.类似 cuBLAS,他们在小卷积上的性能与大规模卷积的性能之间存在明显差距.这使得在很低的稀疏程度下,我们生成的算子性能就可以显著超过 cuSPARSE 和 Escoin.另外,当稀疏程度很高时(0.8-0.9),cuSPARSE 的性能相对较低稀疏程度有所改善.但我们生成的稀疏算子仍能获得 5.5−59.5 倍的加速.在所有的算子和稀疏程度下,我们的方法都能获得相对 cuSPARSE 至少 5.5 倍的加速.对于批大小为 1 的场景,在 0.1-0.9 的稀疏程度下,我们的方法对于 cuSPARSE 可以获得 1.2-22.9 倍的加速.

我们的方法对 Escoin 的性能优势也很明显.在批大小为 64 和 1 时,我们的方法在所有稀疏程度和算子上平均可实现 18.1 倍和 4.1 倍的加速.我们在代码中发现,Escoin 中存在众多硬编码的固定的优化参数(例如 tile size 等)以及 kernel 选择规则,这些参数与具体问题规模和平台相关,限制了 Escoin 在其他平台和算子上的性能表现.例如,当稀疏程度从 0.8 增加到 0.9 时,Escoin 将使用不同的 kernel.图 11(d)中前 3 个算子因此获得了明显的性能提升.然而这一硬编码的规则并没有充分考虑算子间的差异,对于 vgg-conv3 和 alexnet-conv1,新的 kernel 则会造成性能下降的问题.在不同稀疏程度下,我们生成的代码都展现了明显的性能优势,证明在使用 GPU 执行稀疏卷积神经网络时,我们的方法是更好的选择.

3.7 开销分析

这一节关注开销问题.由于本文的代码生成方法与稀疏参数的取值紧密相关,稀疏参数的取值被编码在生成的 ptx 代码中,并经过 ptxas 汇编器编译为二进制 cubin 文件;同时,由于循环展开,在 ptx 和 cubin 中的指令数目会显著增加.这可能导致生成的文件体积较大,并在执行过程中给指令的访问带来压力.我们对生成文件的体积进行分析,同时通过 profiling 分析循环展开后指令数目膨胀对运行时的指令访问造成的影响.

首先,我们对各个算子在卷积参数稠密、稀疏程度为 0.5 和稀疏程度为 0.9 时,对应的 ptx 代码和编译生成的 cubin 二进制文件的体积进行了统计,结果展示在表 4 中.

Table 4 Sizes of ptx and cubin files in KB under various sparsity levels

表 4 不同稀疏程度下 ptx 与 cubin 文件大小,单位 KB

类型	稀疏程度	alexnet-conv1	alexnet-conv2	alexnet-conv3	lenet-conv1	lenet-conv2	resnet-conv1	resnet-conv2	vgg-conv1	vgg-conv2	vgg-conv3
ptx	0	244.19	2670.04	5348.87	55.54	2581.98	4207.35	16156.37	191.94	4070.51	8075.77
	0.5	135.47	1404.52	2822.49	35.79	1387.26	2386.16	8756.73	114.38	2193.88	4293.33
	0.9	51.15	421.74	854.96	16.87	448.33	998.64	3023.07	56.27	746.77	1410.06
cubi	0	63.16	603.23	1205.79	17.29	599.5	958.16	3617.48	53.66	900.41	1775.91

n										
					4					
	0.5	37.35	325.60	651.48	12.41	330.66	557.98	2019.41	34.54	500.04	972.04
	0.9	16.41	98.54	196.79	7.29	106.79	232.54	709.35	19.16	177.29	333.29

由于 ptx 是文本形式的文件,因此其体积比 cubin 更大.在 0.9 的稀疏程度下,执行的 cubin 文件大小为 7.29KB-709.35KB.除了存储开销,我们也考虑循环展开造成的指令数目膨胀对运行时指令访问的影响.我们继续使用 nvprof 工具,统计由于指令访问延迟导致线程阻塞的比例.我们将我们的方法生成的 kernel 与 cuDNN 和 cuSPARSE 进行了对比,实验中使用的稀疏程度为 0.9,在 10 个算子上的实验结果见表 5.

相对于 cuDNN 和 cuSPARSE,我们生成的 kernel 在运行时由于指令访问延迟带来的阻塞比例更低.由于循环展开消除了部分与控制流相关的指令,使得对指令的访问变为顺序访问的模式,GPU 上的指令访问部件也可以更好地预测下一步将要执行的指令,并进行对指令的读取.

Table 5 Percentage of stalls caused by instruction fetch delay

表 5 由于获取下一条指令的延迟导致执行阻塞的比例

算子		alexnet-conv1	alexnet-conv2	alexnet-conv3	lenet-conv1	lenet-conv2	resnet-conv1	resnet-conv2	vgg-conv1	vgg-conv2	vgg-conv3
方法	cuDNN	4.89%	5.02%	6.81%	6.30%	4.77%	3.95%	4.86%	4.91%	3.60%	3.28%
	cuSPARSE	5.03%	5.96%	6.58%	6.41%	5.11%	4.03%	4.08%	9.87%	4.23%	3.74%
	Our	1.64%	2.12%	3.24%	2.67%	4.31%	2.83%	3.50%	1.49%	2.27%	2.70%

4 相关工作

在这一节中,我们对相关工作进行分析.我们主要介绍其他以加速稀疏神经网络执行为目标的工作.从实现方法区分,已有工作可以分为软件层面的优化方法和硬件层面的加速器设计.

4.1 稀疏神经网络性能优化

Intel 提出了 GS[26] 算法优化 CPU 上的稀疏卷积.GS 将稀疏卷积视为稀疏矩阵-稠密矩阵乘法问题,稠密矩阵基于卷积的输入变换得到.GS 将稠密矩阵的生成集成在了计算中,以降低访存需求,提升计算密度.另外,GS 使用了 tiling 和向量化等技术优化在 CPU 上的数据访问和指令吞吐量.GS 使用 CSR 存储稀疏模型参数,与本文的方法相比会占用额外的存储空间并导致更多的运行时访存,作者在论文中发现基于 CSR 表示的稀疏参数会带来约 1 倍的存储开销.SparseCNN[30] 也利用了为特定稀疏矩阵定制相应运算的思想.作者对模型压缩后引入的稀疏矩阵和稠密矩阵乘法在 CPU AVX256 指令集上进行了手工实现.作者对开源的 OpenBLAS[31] 进行了修改,基于 OpenBLAS 中的 tiling 框架,作者去除了与无效分块进行计算的代码.虽然基于类似的消除冗余计算的思想,但本文的方法基于一种通用的中间表示,不需要对同一算子的不同稀疏参数重复编写和手工调优代码.SDC[32] 在执行神经网络前为稀疏参数引入了一个额外的预处理步骤,计算每个有效参数在输入张量上对应的偏移量.在后续进行计算时,基于这个偏移量访问输入数据.SDC 降低了运行时计算每个有效参数对应的输入数据位置的计算量,但对每个有效参数,仍然需要一个额外的偏移量记录位置信息.本文的方法避免了位置信息的记录,进一步降低了计算过程中与位置信息相关的计算和访存需求.另外,以上工作均面向 CPU,本文的方法针对 GPU 平台.上述工作的优化技术并不能直接应用到 GPU 平台上.本文基于 GPU 的特点设计了算子模板,并结合 PTX 指令的特点设计了冗余指令删除的稀疏代码生成方法.同时利用了 GPU 存储层次和访存路径的特点,优化了稀疏卷积运行时的访存性能.

Escoin[24] 也针对稀疏卷积在 GPU 平台上性能差的问题.它使用 GS 中的卷积算法,并对其在 GPU 上的实现进行了优化.作者利用了共享内存和高速缓存实现数据重用,改善访存吞吐量.本文的方法采用了从稠密

代码中删除冗余指令的方法,为具体的稀疏参数定制对应的算子代码.另外,由于采用GS提出的稀疏卷积计算法,Escoin 也继承了 CSR 稀疏格式占用额外位置信息的缺点.也有一些工作从稀疏性的分布规律入手,进行稀疏卷积在 GPU 上的优化.这些工作对稀疏数据中非零元素的分布进行了约束和假设,降低优化稀疏计算的难度.Scott 等在论文[33] 中为参数具有分块稀疏特点的全连接算子和卷积算子设计了高效的 GPU 实现,并展示了基于分块稀疏参数构建小世界 LSTM 等算法的有效性.在论文[34] 中,作者提出了均衡剪枝方法,将矩阵每一行分为等宽的多个块,在剪枝时,要求所有块内保留相同数目的非 0 元素.作者利用这一性质实现了不同 GPU 线程间的负载均衡,并使用共享内存解决在输入数据上的不连续访存问题.与以上工作相比,我们的方法不对剪枝后稀疏模型参数的分布做任何约束,因此可以在通过任意剪枝方法获得的模型上工作,同时也给模型剪枝算法留下了更大的空间,有助于剪枝算法删除更多的参数.

4.2 稀疏神经网络加速器

面对稀疏神经网络计算难以在现有处理器上获得有效加速的问题,也有一些工作尝试通过设计新的加速器进行解决.由于稀疏模型参数数目显著降低,且数据搬运的能耗往往高于代数计算[11] ,EIE[35] 将稀疏模型的参数放入片上缓存中,节约了大量的能耗.SCNN[36] 同时利用参数中的稀疏性和 ReLU 激活函数[1] 在输入数据中引入的稀疏性,利用笛卡尔积计算卷积.在论文[37] 中,作者在 GPU 上增加了额外的部件,用于在运行时跳过取值为 0 的参数对应的冗余指令.与以上工作相比,本文使用了纯软件的方法在现有的 GPU 平台上实现了稀疏卷积的加速,不需要对现有硬件平台进行修改.

5 总结与未来工作

在本文中,我们提出了一种加速剪枝后稀疏卷积神经网络在 GPU 上执行的优化方法.我们基于从稠密代码中删除冗余指令的思想,设计实现了一个稀疏优化框架.在算子模板中,我们将模板参数与计算指令绑定,并建立基于 PTX 的算子中间表示模板.基于中间表示模板和具体的稀疏参数取值,通过分析识别冗余的指令,生成对应的 GPU 程序.为了改善稀疏计算的访存瓶颈,我们利用常量缓存加速稀疏参数的访问;同时在生成的算子代码中隐式编码非 0 参数的位置信息,避免了存储位置信息带来的额外访存需求.通过实验,我们验证了本文所提出的方法能够有效改善稀疏卷积在 GPU 上的执行效率,并且相对已有方法实现了显著的加速效果.

未来,我们计划从多个方面改进当前的工作.首先是算子模板的编译速度.由于需要确定参数与对应指令的关系,我们需要将模型参数涉及的计算进行循环展开.当算子规模比较大时,会导致展开后的代码序列很长,这会影响编译生成 PTX 中间表示的速度.尽管生成的 PTX 中间表示模板可以用于同一算子的不同稀疏参数,但加快编译生成 PTX 的速度能帮助我们探索在更多算子上的性能情况,同时也使我们可以在算子模板上尝试更多的优化技术.由于展开的循环序列执行非常类似的计算,我们希望能够利用这一特点,在循环之间复用 PTX 代码,以改进编译生成 PTX 中间表示的速度.第二,我们希望探索在 CPU 上基于稠密中间表示模板,建立优化稀疏程序的思路.

References:

[1] Krizhevsky A, Sutskever I, Hinton GE. ImageNet classification with deep convolutional neural networks. In: Bartlett P L, Pereira F C N, Burges C J C, Bottou L, Weinberger K Q, eds. Proceedings of Advances in Neural Information Processing Systems (NIPS). Lake Tahoe, Nevada, United States: Curran Associates, 2012. 1106–1114.

[2] He K, Gkioxari G, Dollár P, Girshick RB. Mask r-cnn. In: Proceedings of IEEE International Conference on Computer Vision (ICCV). Venice, Italy: IEEE Computer Society, 2017. 2980–2988.

[3] Liu W, Anguelov D, Erhan D, Szegedy C, Reed SE, Fu C-Y, Berg AC. SSD: single shot multibox detector. In: Leibe B, Matas J, Sebe N, Welling M, eds. Proceedings of European Conference on Computer Vision (ECCV). Amsterdam, The Netherlands: Springer, 2016. 9905: 21–37.

[4] Chen X, Ma H, Wan J, Li B, Xia T. Multi-view 3d object detection network for autonomous driving. In: Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR). Honolulu, HI, USA: IEEE Computer Society, 2017. 6526–6534.

[5] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser L, Polosukhin I. Attention is all you need. In: Guyon I, Luxburg U von, Bengio S, Wallach H M, Fergus R, Vishwanathan S V N, Garnett R, eds. Proceedings of Advances in Neural Information Processing Systems (NIPS). Long Beach, CA, USA: Curran Associates, 2017. 5998–6008.

[6] He K, Zhang X, Ren S, Sun J. Deep residual learning for image recognition. In: Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR). Las Vegas, NV, USA: IEEE Computer Society, 2016. 770–778.

[7] Lecun Y, Bottou L, Bengio Y, Haffner P. Gradient-based learning applied to document recognition. Proceedings of the IEEE, 1998, 86(11): 2278–2324.

[8] Deng J, Dong W, Socher R, Li L-J, Li K, Li F-F. ImageNet: a large-scale hierarchical image database. In: Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR). Miami, Florida, USA: IEEE Computer Society, 2009. 248–255.

[9] Ye S, Zhang T, Zhang K, Li J, Xu K, Yang Y, Yu F, Tang J, Fardad M, Liu S, Chen X, Lin X, Wang Y. Progressive weight pruning of deep neural networks using admm. CoRR, 2018, abs/1810.07378.

[10] Guo Y, Yao A, Chen Y. Dynamic network surgery for efficient dnns. In: Lee D D, Sugiyama M, Luxburg U von, Guyon I, Garnett R, eds. Proceedings of Advances in Neural Information Processing Systems (NIPS). Barcelona, Spain: Curran Associates, 2016. 1379–1387.

[11] Han S, Mao H, Dally WJ. Deep compression: compressing deep neural network with pruning, trained quantization and huffman coding. In: Bengio Y, LeCun Y, eds. Proceedings of International Conference on Learning Representations (ICLR). San Juan, Puerto Rico. 2016.

[12] Gale T, Elsen E, Hooker S. The state of sparsity in deep neural networks. CoRR, 2019, abs/1902.09574.

[13] Tinney WF, Walker JW. Direct solutions of sparse network equations by optimally ordered triangular factorization. Proceedings of the IEEE, IEEE, 1967, 55(11): 1801–1809.

[14] Smith S, Ravindran N, Sidiropoulos ND, Karypis G. SPLATT: efficient and parallel sparse tensor-matrix multiplication. In: Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS). Hyderabad, India: IEEE Computer Society, 2015. 61–70.

[15] Chetlur S, Woolley C, Vandermersch P, Cohen J, Tran J, Catanzaro B, Shelhamer E. CuDNN: efficient primitives for deep learning. CoRR, 2014, abs/1410.0759.

[16] Nvidia C. Cublas library. NVIDIA Corporation, Santa Clara, California, 2008, 15(27): 31.

[17] Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition. In: Bengio Y, LeCun Y, eds. Proceedings of International Conference on Learning Representations (ICLR). San Diego, CA, USA. 2015.

[18] He Y, Zhang X, Sun J. Channel pruning for accelerating very deep neural networks. In: Proceedings IEEE International Conference on Computer Vision (ICCV). Venice, Italy: IEEE Computer Society, 2017. 1398–1406.

[19] Li H, Kadav A, Durdanovic I, Samet H, Graf HP. Pruning filters for efficient convnets. In: Bengio Y, LeCun Y, eds. Proceedings of International Conference on Learning Representations (ICLR). Toulon, France. 2017.

[20] Wen W, Wu C, Wang Y, Chen Y, Li H. Learning structured sparsity in deep neural networks. In: Lee D D, Sugiyama M, Luxburg U von, Guyon I, Garnett R, eds. Proceedings of Advances in Neural Information Processing Systems (NIPS). Barcelona, Spain: Curran Associates, 2016. 2074–2082.

[21] Han S, Pool J, Tran J, Dally WJ. Learning both weights and connections for efficient neural network. In: Cortes C, Lawrence N D, Lee D D, Sugiyama M, Garnett R, eds. Proceedings of Advances in Neural Information Processing Systems (NIPS). Montreal, Quebec, Canada: Curran Associates, 2015. 1135–1143.

[22] Dong X, Chen S, Pan SJ. Learning to prune deep neural networks via layer-wise optimal brain surgeon. In: Guyon I, Luxburg U von, Bengio S, Wallach H M, Fergus R, Vishwanathan S V N, Garnett R, eds. Proceedings of Advances in Neural Information Processing Systems (NIPS). Long Beach, CA, USA: Curran Associates, 2017. 4857–4867.

[23] Naumov M, Chien L, Vandermersch P, Kapasi U. Cusparse library. GPU Technology Conference. 2010.

[24] Chen X. Escoin: efficient sparse convolutional neural network inference on gpus. CoRR, 2018, abs/1802.10280.

[25] Mao H, Han S, Pool J, Li W, Liu X, Wang Y, Dally WJ. Exploring the granularity of sparsity in convolutional neural networks. In: Proceedings of IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPR Workshops). Honolulu, HI, USA: IEEE Computer Society, 2017. 1927–1934.

[26] Park J, Li SR, Wen W, Tang PTP, Li H, Chen Y, Dubey P. Faster cnns with direct sparse convolutions and guided pruning. In: Bengio Y, LeCun Y, eds. Proceedings of International Conference on Learning Representations (ICLR). Toulon, France. 2017.

[28] Zhang X, Tan G, Xue S, Li J, Zhou K, Chen M. Understanding the gpu microarchitecture to achieve bare-metal performance tuning. In: Sarkar V, Rauchwerger L, eds. Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP). Austin, TX, USA: ACM, 2017. 31–43.

[29] Williams S, Waterman A, Patterson DA. Roofline: an insightful visual performance model for multicore architectures. Commun. ACM, 2009, 52(4): 65–76.

[30] Liu B, Wang M, Foroosh H, Tappen MF, Pensky M. Sparse convolutional neural networks. In: Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR). Boston, MA, USA: IEEE Computer Society, 2015. 806–814.

[31] Wang Q, Zhang X, Zhang Y, Yi Q. AUGEM: automatically generate high performance dense linear algebra kernels on x86 cpus. In: Gropp W, Matsuoka S, eds. Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC). Denver, CO, USA: ACM, 2013. 25:1–25:12.

[32] Daultani V, Ohno Y, Ishizaka K. Sparse direct convolutional neural network. In: Cong F, Leung A C-S, Wei Q, eds. Proceedings of International Symposium on Neural Networks. Sapporo, Hakodate, and Muroran, Hokkaido, Japan: Springer, 2017. 10261: 293–303.

[33] Gray S, Radford A, Kingma DP. Gpu kernels for block-sparse weights. CoRR, 2017, abs/1711.09224.

[34] Yao Z, Cao S, Xiao W, Zhang C, Nie L. Balanced sparsity for efficient dnn inference on gpu. In: Proceedings of AAAI Conference on Artificial Intelligence (AAAI). Honolulu, Hawaii, USA: AAAI Press, 2019. 5676–5683.

[35] Han S, Liu X, Mao H, Pu J, Pedram A, Horowitz MA, Dally WJ. EIE: efficient inference engine on compressed deep neural network. In: Proceedings of ACM/IEEE Annual International Symposium on Computer Architecture (ISCA). Seoul, South Korea: IEEE Computer Society, 2016. 243–254.

[36] Parashar A, Rhu M, Mukkara A, Puglielli A, Venkatesan R, Khailany B, Emer JS, Keckler SW, Dally WJ. SCNN: an accelerator for compressed-sparse convolutional neural networks. In: Proceedings of Annual International Symposium on Computer Architecture (ISCA). Toronto, ON, Canada: ACM, 2017. 27–40.

[37] Park H, Kim D, Ahn J, Yoo S. Zero and data reuse-aware fast convolution for deep neural networks on gpu. In: Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES). Pittsburgh, Pennsylvania, USA: ACM, 2016. 33:1–33:10.

附中文参考文献:

[27] 雷杰, 高鑫, 宋杰, 王兴路, 宋明黎.深度网络模型压缩综述.软件学报,2018,29(2):251-266.