基于树莓派的高效卷积优化方法

郭晓龙,牛晋宇,杜永萍
(北京工业大学 信息学部,北京 100124)

摘 要:针对卷积神经网络(CNN)的巨大参数量和计算量而导致在树莓派等低功耗的边缘设备模型推理过程中产生耗时较大的问题,对网络上现有的开源推理框架进行了深入研究及对比分析,发现这些都属于通用型推理框架,并不能针对树莓派设备进行极致推理优化。因此,提出了基于RoofLine模型的定量分析方法,从访存和运算二个维度对Mobilenet等移动端网络架构模型进行卷积推理优化。研究采用了计算图优化方法,利用算子融合和内存重排做推理预处理,从而减少推理过程的计算量和访存开销;同时针对每一层的卷积参数量和特性,提出了9宫格分块策略和NEON指令流水线级别的优化。实验表明,所提出的优化方法在不同的分辨率下,相比腾讯的开源框架NCNN、阿里MNN和商汤PPL. NN在推理速度上取得了高于3倍的性能优化。

关键词:深度学习模型推理加速;计算图优化;算子融合;卷积优化;移动端推理框架
中图分类号:TP303 **文献标识码**:A **文章编号**:1673-629X(2023)05-0096-09
doi:10.3969/j. issn. 1673-629X. 2023. 05. 015

Optimization Method of Efficient Convolution Based on Raspberry Pi

GUO Xiao-long, NIU Jin-yu, DU Yong-ping
(School of Information Technology, Beijing University of Technology, Beijing 100124, China)

Abstract: In response to the problem of time-consuming in reasoning process of low-power edge devices such as Raspberry Pi due to the huge number of parameters and calculation amount of convolutional neural network (CNN), an in-depth study and comparative analysis of the existing open source reasoning framework on the network found that these are general reasoning frameworks, which cannot be optimized for the ultimate reasoning of Raspberry PI devices. Therefore, we propose a quantitative analysis method based on the RoofLine model to optimize the convolutional reasoning of mobile terminal network architecture models such as Mobilenet from two dimensions of memory access and operation. Firstly, by using the computational graph optimization method, operator fusion and memory arrangement as inference preprocessing, the amount of computation and memory access overhead in the inference process are reduced. Secondly, according to the CNN parameters and characteristics of each layer, the 9-grid block strategy and the optimization of NEON instruction pipeline are proposed. Experiments show that the proposed method achieves more than three times performance optimization under different resolutions in inferencing speed compared with Tencent's open-source framework NCNN, Alibaba MNN and Sensetime PPL. NN.

Key words: deep learning model inference acceleration; computational graph optimization; operator fusion; convolution optimization; mobile inference framework

0 引 言

随着深度学习的快速普及,卷积神经网络在图像识别、目标跟踪等视觉领域作为其算法的核心技术得到了广泛应用。通常卷积操作无论在参数量大小还是运算复杂度上已经占据了整个模型的80%以上。尤其针对低功耗等边缘设备,受其有限的算力和内存宽带资源的限制,能否在此类设备上落地应用成为亟需解决的优化问题。虽然各大厂商都推出了自己的推理框架,如Caffe、TensorFlow Lite、ncnn、mnn等,但是这些框架通常作为通用型推理框架,往往在某一特定平台(比如树莓派)下未必能够达到速度最优。比如Caffe使用Im2col技术做卷积优化,其核心思想是将卷积操作转化为两个矩阵相乘,为实现直接矩阵乘法,先要将输入数据和卷积核按照卷积规则在内存中重新

收稿日期:2022-05-23 **修回日期**:2022-09-27
基金项目:国家重点研发计划(2018YFC1900804,2019YFC1906002)
作者简介:郭晓龙(1983-),男,硕士研究生,研究方向为深度学习模型推理加速;通讯作者:杜永萍(1977-),女,博士,教授,研究方向为模式识别与智能信息处理。

排布变成二维矩阵。但在树莓派上其内存重排的耗时往往要大于直接卷积的耗时,还未包含后面的矩阵乘法运算耗时。此外还有国内的开源推理框架 ncnn,针对卷积的 padding 操作也需要提前做内存重排,以及卷积的 bias 和 relu 操作也需要重新遍历一遍内存。可见其昂贵的额外访存开销变成了资源的额外浪费。

1　计算图优化

计算图优化[1]的作用是在不影响模型数值精度的基础上,通过拓扑图变化达到减化计算、减少访问等系统开销的目的,有助于特定设备推理加速。在推理过程中,对于不断的图片检测输入,计算图优化部分只需要在前期做一次即可,后期的每一次推理都可以直接使用优化结果。因此,图优化是整个推理框架[2]的首要准备及重要工作,也是收益最大的一部分。

1.1　算子融合

该文优化的模型为 Retinaface[3],该模型基于 MobileNet 网络架构[4]。其中类似 Conv+BN+ReLU 的经典组合占据了模型的 80% 以上。为提升推理速度,可以采用算子融合的方法将 3 个算子变为 1 个算子,从而减少计算量和访存量。首先,Conv 和 BN 进行合并的推导公式可以参考文献[5],提前把卷积新的 Weight 和 Bias 计算好,替换原卷积对应的 Weight 和 Bias 值,同时删除 BN 算子。其次,对于 Relu 算子,可以采用本地合并计算(inplace relu)方法,就是在卷积运算的类中设置标志 bool relu,在卷积计算之后结果写到内存之前,根据保存的 relu 标志位来决定是否执行 std::max(x,0)。由此优化掉了一次全局内存遍历。最终如图 1 所示。

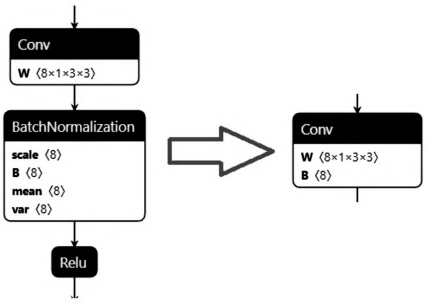

图 1　Conv+BN+ReLU 算子融合

1.2　算子布局优化

为了充分利用 CPU 计算硬件特性,该文采用 NEON SIMD 方式进行计算加速。树莓派 4B 采用的 ARM A72 64 位 CPU,NEON 指令宽度是 128 bit,即 16 字节,而 float 占 4 个字节,因此一条 NEON 指令可以处理 4 个 float 数据。又根据卷积的运算公式是卷积核与图像数据的对应位置相乘再相加到结果中,因此与 NEON 中的 FMLA 指令的处理方式相同。基于此,该文采用一种名为 NHW4C 的存储形式,将同一像素的连续的 4 通道像素连续存储在一起。每 4 通道所有 $W * H$ 数据组成一组,一共分成 $C/4$ 组。

2　卷积优化方法

经过上一步的计算图优化之后,已经合并了所有的 BN+ReLU 算子,拓扑图已经变成连续相邻的卷积算子,且所有卷积算子的 Weight 和 Bias 参数格式已转换为 NHW4C。为了评估卷积的优化方法[6],该文采用 Roofline 模型分析方法。

Roofline 模型[7]分析方法使用计算强度(Operational Intensity)进行定量分析,并给出了模型在计算平台上所能达到理论计算性能上限的公式。公式表达为算力上限 π 除以带宽上限 β,两个指标相除即可得到计算平台的计算强度上限 I_{max},单位是 FLOPs/Byte。树莓派 4B 的峰值算力 π = 1.5 GHz * 1(FMLA 指令为单发射) * (4+4)(FMLA 指令一次可以完成 4 次加法和 4 次乘法) = 12G Flops/s。内存带宽 β = 4G Bytes/s。因此树莓派 4B 的理论计算强度 I_{max} 上限 = 12/4 = 3 FLOPs/Byte(这里计算的值均为理论值,实测可能要稍低一点)。若计算出的强度低于 3 FLOPs/Byte 为带宽瓶颈区域,说明因为访存过大无法完全发挥平台算力。高于 3 FLOPs/Byte 为计算瓶颈区域,说明此时算法已经 100% 地利用了 CPU 的全部算力。该文的优化目标就是尽量接近 I_{max} 理论上限。树莓派 4B 的理论计算强度如图 2 所示。

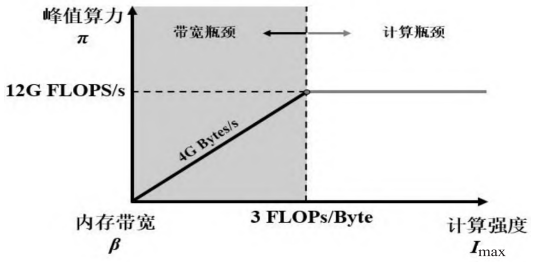

图 2　树莓派 4B Roofline 模型

2.1　3×3 普通卷积

Retinaface 模型的第一个算子就是 3×3 普通卷积,输入 tensor 为(1,IC,IW,IH),输出 tensor 为(1,OC,OW,OH),卷积核 tensor 为(OC,IC,3,3),输入通道 IC = 3,输出通道 OC = 8,步长 stride = 2。算法 1 给出了一个普通卷积的常规计算方法。

算法 1:3×3 普通卷积原始算法。

输入:input tensor I, kernel filter F, padding P, stride S;

输出:output tensor O。

```
1  Mat IP = enlarge input I with padding P
2  O = add(O, bias)
3  for c = 0 to C_o − 1 do
4      for w = 0 to W_o − 1 do
```

5 for $h = 0$ to $H_o - 1$ do
6 for $k = 0$ to $C_i - 1$ do
7 for $i = 0$ to $W_f - 1$ do
8 for $j = 0$ to $H_f - 1$ do
9 $O_{c,w,h} += \mathrm{IP}_{k,w\times s+i,h\times s+j} \times F_{c,k,i,j}$
10 $O = \max(0.f, O)$

算法 1 流程与 ncnn 中卷积算法基本一致,除了卷积计算部分(第 3 ~9 行)采用 SIMD 外。算法 1 对内存的所有操作包括:

(1)对输入数据进行 padding 扩张。读取输入数据后写入到扩张的新内存。共需要 IC * IW * IH+IC *(IW+2 * padding) *(IH+2 * padding)内存读写。可省略为 2 * IC * IW * IH。

(2)将 bias 值提前写到输出数据中(第 2 行)。共需要 OC * OW * OH 内存读写。最后对输出数据进行 Relu 操作(第 10 行),先读取后写入,共 2 * OC * OW * OH。

(3)4 层 for 循环中(第 3 ~6 行)输入数据共读取了 OC * IC * IW * IH,输出数据先读取后写入总量 2 * OC * IC * OW * OH。

(4)卷积核共需 OC * IC * 3 * 3 内存读取。

算法 1 的计算访存比为:

[OC * IC * OW * OH * 3(KW) * 3(KH) * 2]/[((2 + OC) * IC * IW * IH + 5 * OC * IC * OW * OH + OC * IC * 3 * 3) * sizeof(float)]

由 IC=3,OC=8,stride=2,可知 OW=IW/2,OH=IH/2。因此可以进一步简化为:

$$\frac{27 * IW * IH}{60 * IW * IH + 216} < 0.45 \ \mathrm{FLOPs/Byte}$$

可见算法 1 的计算强度仅为树莓派 4B 的理论计算强度的 15%。已经明显属于严重访存瓶颈区域了,严重受限于访存瓶颈的限制,无法发挥硬件的算力。

2.1.1 访存优化

由于第一个算子的输入数据(Input Tensor)是每一次推理过程的外部传入的(通常为图像的 RGB 输入格式),其格式为 NCHW,这里也可以将 NCHW 转为 NHW4C 再进行推理(如 ppl. nn 框架就是这么做的)。但即使输入数据尺寸为 320 * 320,3 通道的 float 类型图片数据,占用内存空间也达到了 1 200 KB,在树莓派 4B 上一次 Reorder 转换耗时就要 3 ms,一次 padding 扩张需要 2 ms。尺寸为 800 * 800 的 Reorder 转换耗时高达 18 ms,padding 扩张也高达 12 ms。所以进行一次内存格式转换是非常昂贵的。因此,算法 2 采用直接卷积方法,意味着输入数据为 NCHW,卷积核和输出数据为 NHW4C。同时对于算法 1 中第 1 行代码的 padding 操作,也采用 9 宫格分块的策略进行优化。该文提出的算法 2 总体的优化思路是减少所有不必要的访存开销。

算法 2:3×3 普通卷积优化算法。

输入:input tensor I, kernel filter F, bias B, padding P, stride S;

输出:output tensor O。

1 out_neon_w_start = (left_padding + $s - 1$) / s;
2 out_neon_h_start = (top_padding + $s - 1$) / s;
3 out_neon_h_end = H_o - bottom_padding / s;
4 out_neon_w = (out_neon_w_end - out_neon_w_start) >>2
5 out_left_start = out_neon_w_start + out_neon_w<<2
6 out_left_w = W_o - out_left_start
7 for oc = 0 to C_o /4 - 1 in parallel do
8 for ic = 0 to $C_i - 1$ do
9 for $i = 0$ to W_f do
10 for $j = 0$ to H_f do
11 $K_{i,j}$ = SIMD_Load($F_{oc,ic,i,j}$)
12 out_bias = (ic == 0) ? B_{oc} :0
13 relu0 = ((ic == $C_i - 1$) && relu_flag) ? 0 : -max_float
14 for top = 0 to out_neon_h_start - 1 do
15 direct_padding($I_{ic,\ -left\_padding,\ top\times s-top\_padding}$, $O_{oc,0,top}$, K, W_o, out_bias, relu0)
16 for top = out_neon_h_start to out_neon_h_end - 1 do
17 direct_padding($I_{ic,\ -left\_padding,\ top\times s-top\_padding}$, $O_{oc,0,top}$, K, output_neon_w_start, out_bias, relu0)
18 direct_padding($I_{ic,\ out\_left\_start\times s-left\_padding,\ top\times s-top\_padding}$, $O_{oc,out\_left\_start,top}$, K, out_left_w, out_bias, relu0)
19 for top = out_neon_h_start to (out_neon_h_end - 1)/2 step 2 do
20 direct_simd($I_{ic,\ out\_neon\_w\_start\times s-left\_padding,\ top\times s-top\_padding}$, $O_{oc,out\_neon\_w\_start,top}$, K, out_neon_w, out_bias, relu0)
21 for top = out_neon_h_end to $H_o - 1$ do
22 direct_padding($I_{ic,\ -left\_padding,\ top\times s-top\_padding}$, $O_{oc,0,top}$, K, W_o, out_bias, relu0)

23 Function direct_padding(IB, OB, K, ow, bias, relu0):
24 for $w = 0$ to ow - 1 do
25 v = SIMD_fadd(SIMD_Load($OB_{w\times 4}$), bias)
26 for $i = 0$ to W_f do
27 for $j = 0$ to H_f do
28 if not in padding:
29 SIMD_fma_lane(v, $K_{i,j}$, $IB_{w\times s+i,j}$, i)
30 v = SIMD_fmax(v, relu0)
31 SIMD_Store($OB_{w\times 4}$, v)

32 Function direct_simd(IB, OB, K, ow, bias, relu0):
33 for w = 0 to ow - 1 do
34 for k = 0 to 3 do
35 V_k = SIMD_fadd(SIMD_Load($OB_{(w\times 4+k)\times 4,\ 0}$), bias)
36 W_k = SIMD_fadd(SIMD_Load($OB_{(w\times 4+k)\times 4,\ 1}$), bias)

```
37      for i=0 to W_f do
38        for j=0 to H_f do
39          for k=0 to 3 do
40            SIMD_fmla( V_k, K_{i,j}, IB_{(w×4+k)×s+i, j} )
41            SIMD_fmla( W_k, K_{i,j}, IB_{(w×4+k)×s+i, j+s} )
42      for k=0 to 3 do
43        V_k =SIMD_fmax( V_k, relu0)
44        W_k =SIMD_fmax( W_k, relu0)
45        SIMD_Store( OB_{(w×4+k)×4, 0}, V_k )
46        SIMD_Store( OB_{(w×4+k)×4, 1}, W_k )
```

9 宫格分块策略。为了避免对输入数据进行 padding 的扩张,提出了一种 9 宫格的分块策略,如图 3 所示。9 宫格的思想是建立一个虚拟的 padding 区域(虚线框),而实际的输入数据尺度是不变的(实线框)。之所以这样建立,是因为 padding 区域填充的元素都为 0,而输入数据的 0 乘以任何卷积核元素都为 0。例如图 3 中白底的卷积框所对应的卷积结果 = $k_1 * i_1 + k_2 * i_2 + k_3 * i_3 + k_4 * i_4 + k_5 * i_5 + k_6 * i_6 + k_7 * i_7 + k_8 * i_8 + k_9 * i_9$。其中 k_1/ k_2/ k_3/ k_4/ k_7 所在区域为 padding,所有值全为 0。因此只需要计算非 padding 区域即可,最后简化为: $k_5 * i_5 + k_6 * i_6 + k_8 * i_8 + k_9 * i_9$。这就是算法 2 中 direct_padding 函数所实现的功能。同理,算法只需对所有卷积核落在虚线框的 padding 区域中(即 9 宫格的外圈 8 格)统一调用 direct_padding 函数做特殊处理。那么剩下的中间 input 区域都是有效区域,卷积核的所有元素都可参与运算(如黑底的卷积框),就可以利用 SIMD 进行加速处理。9 宫格的分块策略虽然增加了一些算法复杂度,但是节省了一次全量内存拷贝,相比于昂贵的内存开销,所增加的逻辑处理耗时要小的多。但是这里有一点需要注意的地方是,对于 padding 区域地址的计算会产生内存溢出,如算法 2 第 15 行对输入数据的地址计算,它指向了外侧虚线框的起始位置。这样使用主要是为了地址计算的统一性,只要提前判断,溢出的地址不访问,就不会报错(即便是内存泄露工具检测也不会报错)。

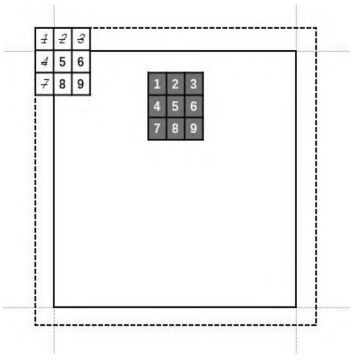

图 3　卷积计算的 padding 策略

访存合并。此外 Add bias 的处理也可以优化到算法逻辑中,因为每一层的输出数据都是所有输入数据的全部 channel 层的卷积求和。因此,该文的做法是在每一次对输入数据 channel 的遍历都强制加上 out_bias,只不过这里的 bias 只有在 input channel 第一层的时候才是真的 bias 参数,其他层为 0。这么做的目的是为了指令流水线不被 if 指令中断,因为 CPU 的分支预测如果发生错误,就需要清空流水线,重新加载正确的分支,会产生 Pipeline Bubbles,相对成本较高。而多加的一条 SIMD_fadd 指令最多只消耗 4 个时钟周期,而且吞吐量为 2 条,即 4 个时钟周期可以同时处理两条 fadd 相加指令。最后 Relu 的处理也类似,是在 input channel 最后一层,因为最后一层的卷积结果与前面所有层相加之后才是最终输出结果。因此,只有最后一层,relu0 的值才为 0,其他情况是 float 负数较大值,故 SIMD_fmax 只有 relu = 0 时才会生效。SIMD_fmax 指令延迟 3 个周期,吞吐量 2 条。

算法 2 中对内存的所有操作包括:

(1)由于输出数据和卷积核都采用了 NHWC4 格式(输入数据仍为 NCHW),每次可以利用 SIMD 同时处理 4 个 channel,因此最外层循环次数降为 OC/4,所以输入数据读取降到了 OC/4 * IC * IW * IH。

(2)9 宫格分块策略只是分开读取,但读取总量不变,因此输出数据先读取后写入总量为 2 * OC/4 * IC * OW * OH * 4。

(3)卷积核共需 OC/4 * IC * 3 * 3 * 4 内存读取。

算法 2 的计算访存比为:

$$[OC * IC * OW * OH * 3(KW) * 3(KH) * 2] / [(OC/4 * IC * IW * IH + 2 * OC * IC * OW * OH + OC * IC * 3 * 3) * sizeof(float)]$$

进一步简化为:

$$\frac{27 * IW * IH}{18 * IW * IH + 216} < 1.5 \text{ FLOPs/Byte}$$

最终算法 2 的计算强度相当于算法 1 提升了 3 倍,相当于树莓派 4B 的理论一半。加上 NEON 指令的加持,推理速度得到大幅改进。

2.1.2　SIMD 指令优化

使用 SIMD 指令计算量不变,只是利用 ARM NEON 指令一次可以处理 4 个 float 的乘加操作[8],提升计算速度,所以 Roofline 模型中算力上限 π 是不变的。SIMD 的优化方法分为二部分:其一是 padding 区域的处理,这里使用了 NEON Intrinsics 函数,每次处理 1 个元素的 4 个通道卷积结果(如 direct_padding 函数)。其二是非 padding 区域的处理,这里使用了 Asm Volatile 内联汇编方式,每次处理 8 个元素的 4 个通道卷积结果(如 direct_simd 函数)。

根据 Cortex A72 优化手册,LD1 指令,一次可以

读取 4 个 float,指令延时(Exec Latency)为 5 个时钟周期,吞吐量(Execution Throughput)为 1,每时钟只能发射 1 条 LD1 指令。FMLA 指令,一次针对 4 个 float(Q-form)进行乘加操作,指令延时为 7 个时钟周期,每时钟只能发射一条 FMLA 指令。但 LD1 和 FMLA 所使用的执行端口(Utilized Pipelines)不同,这意味着对于数据无关的二条指令是可以并行执行的,如图 4 所示。

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
ASIMD load, 1 element, multiple, 1 reg, D-form	LD1	5	1	L	
ASIMD load, 1 element, multiple, 1 reg, Q-form	LD1	5	1	L	

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
ASIMD FP divide, Q-form, F64	FDIV	12-36	1/32-1/10	F0	3
ASIMD FP max/min, normal	FMAX, FMAXNM, FMIN, FMINNM	3	2	F0/F1	
ASIMD FP max/min, pairwise	FMAXP, FMAXNMP, FMINP, FMINNMP	3	2	F0/F1	
ASIMD FP max/min, reduce	FMAXV, FMAXNMV, FMINV, FMINNMV	6	1	F0/F1	
ASIMD FP multiply, D-form	FMUL, FMULX	4	2	F0/F1	2
ASIMD FP multiply, Q-form	FMUL, FMULX	4	1	F0/F1	2
ASIMD FP multiply accumulate, D-form	FMLA, FMLS	7 (3)	2	F0/F1	1
ASIMD FP multiply accumulate, Q-form	FMLA, FMLS	7 (3)	1	F0/F1	1

图 4 Cortex A72 优化手册

根据 A72 优化手册[9],一条读取指令 LD1 的耗时比 FMLA 少了 2 个时钟周期,意味着一条 FMLA 指令就可以遮盖 LD1 取数据的耗时,因此只要流水线设计合理,就不会使得 LD 指令变成瓶颈,运算器才不会因等待而浪费。鉴于 FMLA 指令的延迟为 7 周期,每周期吞吐量为 1,因此最少需要 7 条无数据写依赖关系(读依赖不影响)的 FMLA 指令才能填满流水线。根据上述结论,算法设计为每次迭代输出 2 行,每行 4 个像素,共需要 8 条 FMLA 指令。一条 FMLA 指令可处理 4 个 float,因此每次迭代共输出 32 个卷积中间结果。图 5 所示为 stride = 2 的单次迭代卷积所有参数个数。

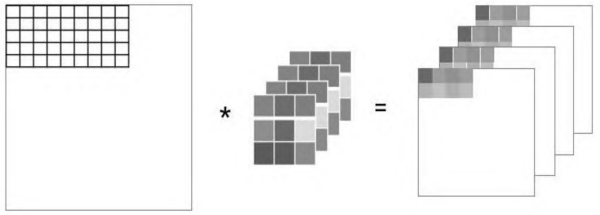

图 5 3×3 普通卷积单次迭代示意图

指令流水线设计。图 5 所示的输入数据格式是 NCHW,卷积核 Weight、Bias 以及输出数据格式为 NHW4C。为了避免 Reorder 操作,一条 FMLA 指令计算方法是将输入数据的 1 个元素分别与卷积核的 1 个元素 4 通道(同色块)相乘,结果与输出数据的 1 个元素 4 通道(同色块)相加。同理分别对卷积核的 9 个元素遍历运算,将每次运算结果产生的 8 * 4 个 float 输出数据求和累加,就得到了单通道输入图像的卷积结果(算法 2 第 37 ~ 41 行)。其中 8 * 4 个 float 输出数据是存放在 neon 寄存器中的,所以累加求和的操作并不会每次都要访存。故将算法分成了 9 组,每组有连续的 8 条不存在写依赖关系的 FMLA 指令,对输入数据的 LOAD 指令可以穿插在每组中间,而卷积核的 9 条 LD 指令在调用函数前就加载到 neon 寄存器中了。按此设计就可以充分发挥处理器流水线作业。同理,遍历输入数据的所有 3 个通道,将所有输出通道的卷积结果累加就完成了 3×3 普通卷积最终结果。这就是算法 2 中,函数 direct_simd 所实现的功能。

循环展开优化。实际编写代码除函数内第 1 行循环(第 33 行代码)外,direct_simd 函数中的其他 for 循环都是不存在的。算法 2 中这样写是为了避免占用过长的篇幅。实际上 for 循环属于分支预测指令,如果预测失败,同样会导致清空流水线,重新加载正确的分支。因此,为了保证流水线不会产生气泡,direct_simd 里第 34 行到第 46 行代码都是展开的,这种方法叫作循环展开优化[10]。

2.2 逐通道卷积(Depthwise Conv)

Mobilenet 中逐通道卷积尺寸通常为:输入 tensor 为(1,C,IW,IH),输出 tensor 为(1,C,OW,OH),卷积核 tensor 为(1,C,3,3)。所有 tensor 的通道数是一致的。因为卷积运算只关心当前通道,因此不需要遍历输入层和卷积核的所有通道[11],所以相比于 3×3 普通卷积和逐点卷积访存效率要高得多。未优化版本如算法 3 所示。

算法 3:逐通道卷积原始算法。

输入:input tensor I,kernel filter F,padding P;

输出:output tensor O。

```
1  Mat IP = enlarge input I with padding P
2  O = add(O, bias)
3  for c = 0 to C - 1 do
4  for w = 0 to W_o - 1 do
5  for h = 0 to H_o - 1 do
6  for i = 0 to W_f - 1 do
7  for j = 0 to H_f - 1 do
8     O_{c,w,h} += IP_{c,w+i,h+j} × F_{c,i,j}
9  O = max(0, f, O)
```

整体流程与算法 1 基本一致,只是中间少了一层对输入层通道的遍历。算法 3 中对内存的所有操作包括:

(1)对输入数据进行 padding 扩张约为 2 * C * IW * IH。

(2)Addbias:C * OW * OH。Relu 操作:2 * C * OW * OH。

(3)输入数据读取 C * IW * IH,输出数据读写 2 * C * OW * OH。

(4)卷积核共需 C * 3 * 3 内存读取。

算法 3 的计算访存比为:

$[C*OW*OH*3(KW)*3(KH)*2]/[(3*C*IW*IH + 5*C*OW*OH + C*3*3)*sizeof(float)]$

由 stride =1,OW=IW,OH=IH,则进一步简化为:

$$\frac{9*OW*OH}{16*OW*OH+18} < 0.56\ \text{FLOPs/Byte}$$

同样算法 3,也是严重访存瓶颈区域了,严重受限于访存瓶颈的限制。远远无法达到树莓派 4B 的理论上限 3 FLOPs/Byte。

2.2.1　访存优化

与 3×3 普通卷积算法的优化方法相同,减少所有不必要的访存开销。同样采用 9 宫格分块策略和访存合并。因此,该文提出的逐通道卷积优化方法,如算法 4 所示。

算法 4:逐通道卷积优化算法。

输入:input tensor I ,kernel filter F ,bias B ;

输出:output tensor O 。

```
1 Function depthwise_simd(IB,OB, K ,ow,bias,relu0):
2   for w =0 to ow-1 do
3     for k =0 to 3 do
4       V_k =SIMD_fadd(SIMD_Load(OB_(w×4+k)×4,0), bias)
5       W_k = SIMD_fadd(SIMD_Load(OB_(w×4+k)×4,1), bias)
6     for i =0 to W_f do
7       for j =0 to H_f do
8         for k =0 to 3 do
9               SIMD _ fmla ( V_k , K_i,j , SIMD _ Load (IB_(w×4+k)×4+i,j))
10              SIMD _ fmla ( W_k , K_i,j , SIMD _ Load (IB_(w×4+k)×4+i,j+1))
11    for k =0 to 3 do
12      V_k =SIMD_fmax( V_k ,relu0)
13      W_k =SIMD_fmax( W_k ,relu0)
14      SIMD_Store(OB_(w×4+k)×4,0 , V_k )
15      SIMD_Store(OB_(w×4+k)×4,1 , W_k )
```

Depthwise 卷积对 padding 的处理同样采用 9 宫格分块策略,与算法 2 相同,这里就不重新列出了。

算法 4 中对内存的所有操作包括:

(1)输入数据读取为 $C/4*IW*IH$。

(2)输出数据读写总量为 $2*C/4*OW*OH*4$。

(3)卷积核共需 $C/4*3*3*4$ 内存读取。

算法 4 的计算访存比为:

$[C*OW*OH*3(KW)*3(KH)*2]/[(C/4*IW*IH + 2*C*OW*OH + C*3*3)*sizeof(float)]$

由 stride =1,OW=IW,OH=IH,则进一步简化为:

$$\frac{9*OW*OH}{4.5*OW*OH+18} < 2\ \text{FLOPs/Byte}$$

最终算法 4 的强度接近于树莓派 4B 的理论 2/3。比优化前提升了 3.5 倍。

2.2.2　SIMD 指令优化

指令流水线设计。与 3×3 普通卷积需要遍历二层(OC/4、IC)通道相比,逐通道卷积只需遍历一层(C/4)通道。另外,由于 3×3 普通卷积每次只能处理 1 个输入元素的 1 通道,而 NHW4C 格式的输入数据每次卷积运算就可以处理 1 个输入元素的 4 通道。可见仅靠 NHW4C 格式,就能将处理能力提升 4 倍。首先,为保证最少 7 条无数据写依赖关系的 FMLA 指令来填满流水线,算法设计为每次迭代输出 2 行,每行 4 个像素,共需要 8 条 FMLA 指令。其次,因卷积核大小为 3 * 3,需要分别对卷积核的 9 元素运算,故分成了 9 组,每组有连续的 8 条 FMLA 执令,LD 指令可以穿插在每组中间。每次迭代共输出 32 个卷积最终结果,如图 6 所示。

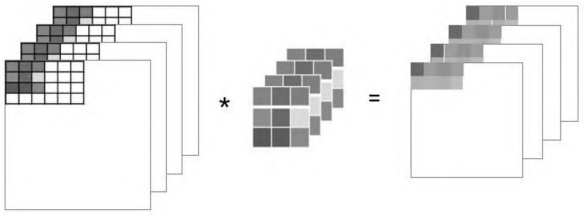

图 6　逐通道卷积单次迭代示意图

循环展开优化。同样为了避免占用过长的篇幅,实际编写代码除函数内第 1 行循环(第 3 行代码)外,depthwise_simd 函数中的其他 for 循环都是不存在的。采用循环展开优化技术,从而避免了分支预测指令和流水线产生气泡。

2.3　逐点卷积

Mobilenet 中逐点卷积尺寸通常为:输入 tensor 为(1,IC,IW,IH),输出 tensor 为(1,OC,OW,OH),卷积核 tensor 为(OC,IC,1,1)。未优化版本如算法 5 所示。

算法 5:逐点卷积原始算法。

输入:input tensor I ,kernel filter F ;

输出:output tensor O 。

```
1   O =add( O ,bias)
2   for c =0 to C_o -1 do
3     for w =0 to W_o -1 do
4       for h =0 to H_o -1 do
5         for k =0 to C_i -1 do
6           O_c,w,h += IP_k,w,h × F_c,k
7   O =max(0. f , O )
```

算法 5 中对内存的所有操作包括:

(1)Add bias:OC * OW * OH。Relu 操作:2 * OC

* OW * OH。

(2)输入读取 OC * IC * IW * IH,输出读写 2 * OC * IC * OW * OH。

(3)卷积核共需 OC * IC * 1 * 1 内存读取。

算法 5 的计算访存比为:

[OC * IC * OW * OH * 1(KW) * 1(KH) * 2]/[(OC * IC * IW * IH + 2 * (1 + IC) * OC * OW * OH + OC * IC * 1 * 1) * sizeof(float)]

由 padding = 0, stride = 1, OW = IW, OH = IH, IC 通常在 8 ~ 256 之间,则进一步简化为:

$$\frac{1}{6+\frac{4}{IC}+\frac{2}{OW*OH}} < \frac{1}{6} < 0.167 \text{ FLOPs/Byte}$$

逐点卷积的计算强度为树莓派 4B 的理论的 5.6%。

2.3.1 访存优化

对于 1×1 卷积,TensorFlow Lite[12] 的做法是转换成二个矩阵的乘法[13-14],然后使用 OpenBLAS 高性能矩阵乘法库[15] 来完成卷积运算。而该文由于采用 NHW4C 格式,转换成两个矩阵乘法的格式内存开销较大。因此,逐点卷积优化方法与之前相同,减少所有不必要的访存开销。优化如算法 6 所示。

算法 6:逐点卷积优化算法。

输入:input tensor I, kernel filter F, bias B;

输出:output tensor O。

```
1  for oc = 0 to C_o/4 - 1 in parallel do
2    for ic = 0 to C_i/4 - 1 do
3      out_bias = (ic == 0) ? B_oc : 0
4      relu0 = ((ic == C_i - 1) && relu_flag) ? 0 : -max_float
5      for n = 0 to 3 do
6        K_n = SIMD_Load(F_oc,ic,n*4)
7      for w = 0 to (W_o * H_o)/8 - 1 do
8       for k = 0 to 7 do
9         V_k = SIMD_fadd(SIMD_Load(O_oc,(w×8+k)×4), bias)
10        X_k = SIMD_Load(I_ic,(w×8+k)×4)
11        for n = 0 to 3 do
12          for k = 0 to 7 do
13            SIMD_fmla(V_k, K_n, X_k[n])
14        for k = 0 to 7 do
15          V_k = SIMD_fmax(V_k, relu0)
16          SIMD_Store(O_oc,(w×8+k)×4, V_k)
```

逐点卷积没有 padding 参数,因此不需要 9 宫格分块策略。算法 6 中对内存的所有操作包括:

(1)输入数据读取为 OC/4 * IC/4 * IW * IH * 4。

(2)输出数据读写总量为 2 * OC/4 * IC/4 * OW * OH * 4。

(3)卷积核共需 OC/4 * IC/4 * 1 * 1 * 4 * 4 内存读取。

算法 6 的计算访存比为:

[OC * IC * OW * OH * 1(KW) * 1(KH) * 2]/[(OC/4 * IC * IW * IH + OC/2 * IC * OW * OH + OC * IC * 1 * 1) * sizeof(float)]

由 stride = 1, OW = IW, OH = IH 则进一步简化为:

$$\frac{2}{3+\frac{4}{OW*OH}} < \frac{2}{3} < 0.667 \text{ FLOPs/Byte}$$

优化后的计算强度也仅为理论的 22%,与逐通道卷积相差很大。从最终实验也知,逐点卷积效率是三者中最差的。

2.3.2 SIMD 指令优化

由于逐点卷积的卷积核大小是 1 * 1 的,卷积操作每次只需要一个输入元素。所以输入/输出 Tensor 由宽和高组成的二维数组可以看成连续排列的一维数组,且输入和输出数据的一维数组长度相等,同时输入/输出的格式都是 NHW4C。逐点卷积的设计原则依然是组成 8 条 FMLA 指令流水线,因此算法 6 中单次迭代(第 7 ~ 16 行)可以对 8 个输出变量进行 1×1 卷积运算,如图 7 所示。

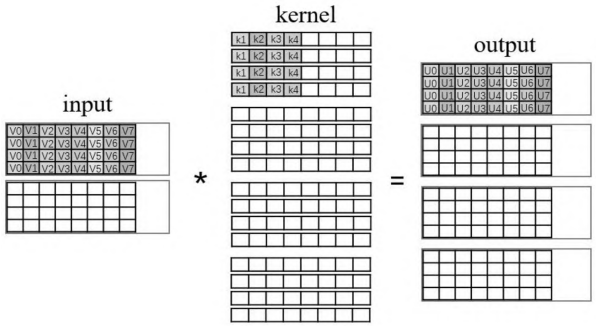

图 7 逐点卷积单次迭代示意图

指令流水线设计。图 7 所示逐点卷积所有参数的例子,图的左侧是输入 Tensor(1,8, W, H),中间卷积核 Tensor (16,8,1,1),右侧输出 Tensor(1,16, W, H)。其中输入数据和输出数据可以看成一维数组,长度为 $W * H$,NHW4C 格式由 4 个连续通道数据排列在一起,因此输入 Tensor 分成了上下两组(8/4),输出 Tensor 分成了四组(16/4)。卷积核大小为 16 * 8 个,同样分成了 4 组,每组中又分成 2 组(与输入数据通道数相对应)。算法的设计思想是,对于每一组的输出数据是所有组的输入数据卷积后累加求和而得(第 2 ~ 16 行代码)。为求得一个输出元素的 4 通道数据 U_0,需要对每组的输入数据分别用每一通道与 1 个卷积核 4 通道相乘,产生的 4 条 FMLA 指令结果累加而成。如:① $U_0 = k_1 * V_0[0]$ ② $U_0 += k_2 * V_0[1]$ ③ $U_0 += k_3 * V_0[2]$ ④ $U_0 += k_4 * V_0[3]$。由于这 4 条 FMLA

指令存在写依赖关系,因此同时处理 8 个输出元素 U_0 ~ U_7,将上述 4 步的每一步分散在 8 个输出元素中,从而组成 8 条无写依赖的 FMLA 指令(第 11 ~ 13 行)。对于不满 8 条的剩余输入/输出数据,可以利用 NEON Intrinsics 函数,采用同样方法,每次可处理 1 个元素的 4 通道卷积。

循环展开优化。同样代码中第 5 ~ 14 行的所有 for 循环需要展开处理,与前面方法相同不再重复。

3　实验结果

测试平台为:树莓派 4B 4 GB 内存,Raspberry Pi OS 64 位系统,Retinaface mnet 0. 25 模型。该文提出的三种算法分别与国产开源框架腾讯 ncnn、商汤 ppl. nn 及阿里 mnn 做对比。分别测试三种不同分辨率图片在不同平台耗时。测试方法使用源码 Release 编译,开启 OpenMP 及 O2 编译优化选项,并在源码中添加耗时打印语句。注:这里统计的耗时不包含图优化部分,只有每一层的算子推理耗时。具体如表 1 ~ 表 3 所示。

表 1　3×3 普通卷积耗时分析　ms

测试平台	800 * 800	512 * 512	400 * 400
文中算法	14	6	4
ncnn	32	12	7
ppl. nn	34	14	8
mnn	25	10	6

表 2　逐通道卷积耗时分析　ms

测试平台	800 * 800	512 * 512	400 * 400
文中算法	12	5	3
ncnn	46	19	14
ppl. nn	26	9	5
mnn	52	21	13

表 3　逐点卷积耗时分析　ms

测试平台	800 * 800	512 * 512	400 * 400
文中算法	78	28	17
ncnn	97	37	27
ppl. nn	85	34	22
mnn	78	32	19

实验结果表明,针对普通卷积和逐通道卷积,由于 ncnn 框架需要提前做 padding 操作和单独的 Add bias 及 relu 操作,而该文提出的 9 宫格分块策略和访存合并优化避免了无畏的浪费,从而大大减少了访存时间,从一定程度上减少了卷积的推理耗时。在普通卷积实验中,ppl. nn 之所以是几种平台中最差的,是因为需要提前将 NCHW 转为 NHW4C 之后才做卷积运算,而 ncnn 虽然不用提前做 reorder 转换,但多余的 3 次全量内存访问也严重影响耗时。根据 2. 1 节分析,算法受限于访存瓶颈,因此文中算法采用直接计算(同时计算 1 通道输入数据和 4 通道输出数据),虽然无法发挥全部算力,但相比一次昂贵的 reorder 开销,实验表明推理耗时可以减少 50% 以上。

由于逐通道卷积的计算量较小,因此访存的增加反而变成最大的瓶颈,所以对于 ncnn 和 mnn 没有做访存优化(采用了单独的 padding、Add bias、relu 操作)。mnn 虽然采用三分块(上中下)策略,对于 padding 策略依然使用了额外的内存布局转换,同时代码处理逻辑也是过于复杂,影响指令流水线,最后由于 mnn 采用 NEON Intrinsics 指令,相比 Asm Volatile 内联汇编方式要慢 10% 左右,而导致结果为所有平台中测试结果最差。而该文提出的 9 宫格算法和访存计算合并方法,对于输入数据、卷积核和输出数据,只需要从头遍历一遍即可完成卷积计算。相比于其他平台减少了至少 2 次以上的全量访存操作。同时又能完全发挥 neon 指令算力(同时计算 4 通道输入数据和 4 通道输出数据),最终算法的计算访存比达到为理论峰值的 2/3。多种因素叠加使得优化高于其他平台 1 倍以上。且 ppl. nn 和 mnn 的代码逻辑过于复杂,过多的 if 语句也会导致流水线得不到充分发挥,指令预处理失败率也要高出很多。而该文从 ARM A72 CPU 指令性能进行分析,设计了一组能充分利用 CPU 流水线的指令序列,可以在同等计算量下最大化发挥处理器的算力,同时在减少分支预测和循环展开技术的加持下,最终提出的普通卷积和逐通道卷积算法耗时相比其他平台提升幅度高达 75% ~367% 。

逐点卷积由于不需要 padding 操作,但 ncnn 依然存在多余的 Add bias 和 relu 单独操作。mnn 采用了与该文类似的计算访存合并算法,因此也证实了计算访存合并方法有利地加速推理。由表 3 可见,与同分辨率下的逐通道卷积耗时相比也大了 5 ~6 倍。原因有几方面,首先,逐点卷积的计算访存比非常低,优化后的算法也只能达到理论的 22% ,相比于逐通道卷积达到理论的 67% ,大部分都浪费在访存上,算力得不到充分利用。其次,在相同分辨率下逐点卷积数据量要比逐通道卷积数据量多了 50% ,导致计算量和访存量都相应增加。最终相比其他平台提升幅度仅为 9% ~ 59% 。

由于篇幅限制,仅对 Retinaface 典型的 3 种卷积算法进行针对性优化。至少在树莓派 4B 这一设备上,相比于国内成熟的开源框架 ncnn 提升了 24% ~ 367% 。即使与商汤最新开源的 ppl. nn 框架相比,也有 9% 到 143% 不同幅度的领先。

4 结束语

针对树莓派 4B 设备,基于 Roofline 模型进行定量分析,提出了针对性的优化方法。从计算图优化,合并算子,NHW4C 格式转换,到 9 宫格分块策略,SIMD 加速方法等所有策略都是为了优化推理耗时。实验结果表明,在推理耗时上比国内大厂的开源框架 ncnn 和 ppl. nn 都有较大幅度的提升。限于篇幅原因,并未对卷积的其他类型进行分析,如 dilation>1 的空洞卷积。另外对于通道数比较深的卷积方法使用 winograd 方法[16]速度会更有优势。

参考文献:

[1] 张承龙,曹华伟,王国波,等. 面向高通量计算机的图算法优化技术[J]. 计算机研究与发展,2020,57(6):1152-1163.

[2] 张 潇,支 天. 面向多核处理器的机器学习推理框架[J]. 计算机研究与发展,2019,56(9):1977-1987.

[3] DENG J,GUO J,ZHOU Y,et al. RetinaFace:single-stage dense face localisation in the wild[J]. arXiv:1905.00641,2019.

[4] HOWARD A G,ZHU Menglong,CHEN Bo,et al. MobileNets:efficient convolutional neural networks for mobile vision applications[J]. arXiv:1704.04861,2017.

[5] PIERRE G,ANTOINE C. Batch norm folding:an easyway to improve your network speed[R/OL]. [2020-06-30]. https://scortex.io/batch-norm-folding-an-easy-way-to-improve-your-network-speed/.

[6] 王一超,廖秋承,左思成,等. 一种 ARM 处理器面向高性能计算的性能评估[J]. 计算机科学,2019,46(8):95-99.

[7] WILLIAMS S,WATERMAN A,PATTERSON D. Roofline:an insightful visual performance model for floating-point programs and multicore architectures[J]. Office of Scientific & Technical Information Technical Reports,2009,52(4):65-76.

[8] 贺爱香,顾乃杰,苏俊杰. 基于多核 ARM 体系结构的基础函数优化方法[J]. 计算机工程,2018,44(5):47-52.

[9] Arm Developer. Cortex® -A72 software optimization guide[S/OL]. [2015-03-10]. https://documentation-service.arm.com/static/5ed75eeeca06a95ce53f93c7? token=.

[10] 廖继荣,董海涛. 利用循环展开最大化软件流水线性能[J]. 纯粹数学与应用数学,2004,20(3):6.

[11] KAISER L,GOMEZ A N,CHOLLET F. Depthwise separable convolutions for neural machine translation[J]. arXiv:1706.03059,2017.

[12] ASHFAQ S. Accelerating deep learning model inference on arm CPUs with ultra-low bit quantization and runtime[J]. arXiv:2207.08820,2022.

[13] ZHANG P,LO E,LU B. High performance depthwise and pointwise convolutions on mobile devices[J]. arXiv:2001.02504,2020.

[14] 冉德成,吴 东,钱 磊. 面向深度学习推理的矩阵乘法加速器设计[J]. 计算机工程,2019,45(10):40-45.

[15] GOTO K,GEIJN R A V D. Anatomy of high-performance matrix multiplication[J]. ACM Transactions on Mathematical Software,2008,34(3):1-25.

[16] LAVIN A,GRAY S. Fast algorithms for convolutional neural networks[J]. arXiv:1509.09308,2015.