计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2020.20200107
57(6):1140-1151,2020

面向飞腾多核处理器的 Winograd 快速卷积算法优化

王庆林　李东升　梅松竹　赖志权　窦　勇
(国防科技大学并行与分布处理国防科技重点实验室　长沙　410073)
(国防科技大学计算机学院　长沙　410073)
(wangqinglin@nudt.edu.cn)

Optimizing Winograd-Based Fast Convolution Algorithm on Phytium Multi-Core CPUs

Wang Qinglin, Li Dongsheng, Mei Songzhu, Lai Zhiquan, and Dou Yong
(*Science and Technology on Parallel and Distributed Processing Laboratory*, *National University of Defense Technology*, *Changsha* 410073)
(*College of Computer*, *National University of Defense Technology*, *Changsha* 410073)

Abstract　Convolutional neural networks (CNNs) have been extensively used in artificial intelligence fields such as computer vision and natural language processing. Winograd-based fast convolution algorithms can effectively reduce the computational complexity of convolution operations in CNNs so that they have attracted great attention. With the application of Phytium multi-core CPUs independently developed by the National University of Defense Technology in artificial intelligence fields, there is strong demand of high-performance convolution primitives for Phytium multi-core CPUs. This paper proposes a new high-performance parallel Winograd-based fast convolution algorithm after studying architecture characteristics of Phytium multi-core CPUs and computing characteristics of Winograd-based fast convolution algorithms. The new parallel algorithm does not rely on general matrix multiplication routines, and consists of four stages: kernels transformation, input feature maps transformation, element-wise multiplication, and output feature maps inverse transformation. The data movements in all four stages have been collaboratively optimized to improve memory access performance of the algorithm. The custom data layouts, multi-level parallel data transformation algorithms and multi-level parallel matrix multiplication algorithm have also been proposed to support the optimization above efficiently. The algorithm is tested on two Phytium multi-core CPUs. Compared with Winograd-based fast convolution implementations in ARM Computer Library (ACL) and NNPACK, the algorithm can achieve speedup of 1.05～16.11 times and 1.66～16.90 times, respectively. The application of the algorithm in the open source framework Mxnet improves the forward-propagation performance of the VGG16 network by 3.01～6.79 times.

Key words　multi-core CPUs; deep learning; convolutional neural networks; Winograd algorithms; parallel algorithms

摘　要　随着深度学习的快速发展,卷积神经网络已广泛应用于计算机视觉、自然语言处理等人工智能领域中.Winograd 快速卷积算法因能有效降低卷积神经网络中卷积操作的计算复杂度而受到广泛关注.

收稿日期:2020-02-26;**修回日期**:2020-04-13
基金项目:"核高基"国家科技重大专项基金项目(2018ZX01028101)
This work was supported by the National Science and Technology Major Projects of Hegaoji (2018ZX01028101).

随着国防科技大学自主研制的飞腾多核处理器在智能领域的推广应用,对面向飞腾多核处理器的高性能卷积实现提出了强烈需求.针对飞腾多核处理器的体系结构特征与 Wingorad 快速卷积算法的计算特点,提出了一种高性能并行 Winograd 快速卷积算法.该算法不依赖通用矩阵乘库函数,由卷积核转换、输入特征图转换、逐元素乘、输出特征图逆变换等 4 个部分构成,融合设计了 4 个部分的数据操作,并设计了与之配套的数据布局、多级并行数据转换算法与多级并行矩阵乘算法,实现访存性能以及算法整体性能的提升.在两款飞腾多核处理器上的测试结果显示,与开源库 ACL 和 NNPACK 中的 Winograd 快速卷积实现相比,该算法分别能获得 1.05～16.11 倍与 1.66～16.90 倍的性能加速;集成到开源框架 Mxnet 后,该算法使得 VGG16 网络的前向计算获得了 3.01～6.79 倍的性能加速.

关键词 多核 CPU;深度学习;卷积神经网络;Winograd 算法;并行算法

中图法分类号 TP183

随着深度学习的快速发展,卷积神经网络(convolutional neural networks, CNNs)[1-2]已广泛应用于语音处理、图像识别和自然语言处理等各种人工智能应用中.卷积神经网络通常由卷积层、池化层、激活层和全连接层等网络层构成.在卷积神经网络的计算中,80%以上的计算量集中在卷积层的实现上,从而卷积层的计算性能几乎决定了整个卷积神经网络的性能.

当前实现卷积计算的方法主要有直接卷积、GEMM、Winograd 以及 FFT 等 4 种[3].直接卷积按照卷积的定义进行直接实现,常表现出较差的性能[4].基于 GEMM 的卷积算法,也称为 im2col+GEMM 方法,将卷积计算转换为矩阵乘,调用 BLAS 库中的 GEMM 函数来完成计算,随后将 GEMM 的计算结果转换为卷积的输出.im2col+GEMM 的优点在于可以利用已经面向处理器高度优化的 BLAS 库,但可能引发临时存储空间需求的暴增[5].同时,BLAS 库中的 GEMM 函数主要面向高性能领域进行了优化,卷积计算中产生的矩阵具有不同的形态,常常不能获得最佳性能[6].基于 Winograd 与 FFT 的 2 种快速卷积实现,通过一定的转换算法来降低卷积的计算复杂度,能有效提升卷积的计算性能.2 种快速卷积最大的不同是,Winograd 快速卷积采用 Winograd 算法来进行转换,FFT 快速卷积则调用 FFT 算法来完成转换.在相同的分块大小下,相比 FFT 快速卷积算法,Winograd 快速卷积算法具有更低的计算复杂度[3],使得其在学术界与工业界均受到了广泛的关注与研究.

在学术界,Lavin 等人[3]第 1 次引入 Winograd 算法来进行 2 维卷积的计算,与直接卷积相比,降低了计算复杂度,基于 NVIDIA GPU 的实现,在 VGG 的所有卷积层上均获得了比 cuDNN v3 FFT 更高的性能.Budden 等人[7]将 Winograd 快速卷积算法扩展到高维卷积操作,通过高效向量化以及可扩展的多核并行等方式在 Intel CPU 上实现了 Winograd 快速卷积算法.Jia 等人[8]提出了一系列方法来优化 Intel 众核处理器上 N 维 Winograd 快速卷积实现,包括特殊的数据结构、高效的自动数组转换以及基于动态编译的批矩阵乘等,可实现任何 kernel 大小的卷积计算,与 CPU 上其他卷积算法相比,取得了平均 3 倍多的性能加速.Xygkis 等人[9]在物联网边缘设备上实现了 Winograd 快速卷积算法,在 Intel/Movidius Myriad2 平台上进行了性能评估,与其他卷积算法相比,性能提升高达 54%.Wang 等人[10]在 GPU 上采用 Winograd 快速卷积算法实现 3 维卷积计算,与 cuDNN v5 中的矩阵乘卷积相比,获得了平均 1.2 倍的性能加速.Wu 等人[11]面向 Intel Xeon Phi 平台的结构特性,设计与优化了 Winograd 快速卷积算法,与 Intel MKL DNN 的卷积实现相比,获得 2 倍多的性能加速.总的来说,学术界当前的研究主要面向 Intel CPU 和 NVIDIA GPU 展开,且不同程度地依赖其他高性能函数库.

在工业界,各大处理器提供商开发了面向各自处理器高度优化的深度学习库,如 Intel MKL-DNN[12],ARM Computer Library[13],NVIDIA cuDNN[14]等,这些深度学习库中均实现了 Winograd 快速卷积.

飞腾多核处理器[15-16]是国防科技大学计算机学院自主研制的系列高性能 64 位通用多核 CPU,兼容 ARMv8 指令集.随着其计算性能的提升,也可以像 Intel X86 CPU 一样用来处理深度学习应用.然而,目前缺乏面向飞腾多核处理器的高性能卷积实现,且相关其他高性能函数库也正在完善中.本文面向国产飞腾多核处理器的体系结构特征,研究 Winograd 快速卷积算法的计算特性,提出了一种新的高性能

并行 Winograd 快速卷积算法,不依赖其他计算库,能够有效提升飞腾多核处理器上的卷积计算性能.与开源库中的 Winograd 快速卷积算法相比,本文提出的算法获得了 1.05～16.90 倍的性能加速比.将本文提出的并行 Winograd 快速卷积算法集成到 Mxnet v1.50 框架后,VGG16 网络的前向计算获得了 3.01～6.79 倍的性能加速.

1 相关定义

1.1 卷积定义

令卷积层的输入特征图为 $\boldsymbol{I}[B][C][H_i][W_i]$,卷积核为 $\boldsymbol{F}[K][C][H_f][W_f]$,输出特征图为 $\boldsymbol{O}[B][K][H_o][W_o]$,其中 B 表示批大小,C 和 K 为输入和输出通道数,H_i,W_i,H_f,W_f,H_o 以及 W_o 分别为输入特征图、卷积核、输出特征图的空间大小.卷积神经网络第 1 层卷积的输入特征图为经过预处理后的原始输入,其他层卷积的输入特征图为前一网络层的输出特征图.基于以上参数定义,卷积层计算为

$$\boldsymbol{O}_{(b,k,h',w')}=\sum_{c=0}^{C-1}\sum_{hf=0}^{H_f-1}\sum_{wf=0}^{W_f-1}(\boldsymbol{I}_{b,c,h'\times s+hf-p_t,w'\times s+wf-p_l}\times\boldsymbol{F}_{k,c,hf,wf}),\tag{1}$$

其中,$0\leqslant b<B$,$0\leqslant k<K$,$0\leqslant h'<H_o$,$0\leqslant w'<W_o$,p_t 和 p_l 分别代表对输入特征图的上边与左边补零的数量,s 表示卷积计算过程中的步长大小.在本文的研究中,只考虑 $s=1$ 的卷积计算,这也是深度学习中应用较广的一类卷积计算.

以单通道特征图的卷积计算为基本单元,式(1)可以简写为

$$\boldsymbol{O}_{b,k}=\sum_{c=0}^{C-1}(\boldsymbol{I}_{b,c}*\boldsymbol{F}_{k,c}),\tag{2}$$

其中,$*$ 表示单通道特征图上的 2 维卷积.

1.2 Winograd 快速卷积算法

Winograd 卷积算法[3]源自 Winograd 提出的最小滤波算法(minimal filtering algorithm),通过对输入特征图和权重上的一系列变换来减少乘法运算次数.在 2 维数据情况下,如果输出和过滤器大小分别为 $m\times m$ 和 $r\times r$,则可将该最小滤波算法记为 $F(m\times m,r\times r)$,其矩阵形式为

$$\boldsymbol{z}=\boldsymbol{A}^{\mathrm{T}}[(\boldsymbol{P}\boldsymbol{y}\boldsymbol{P}^{\mathrm{T}})\odot(\boldsymbol{Q}^{\mathrm{T}}\boldsymbol{x}\boldsymbol{Q})]\boldsymbol{A},\tag{3}$$

其中,$\odot$ 表示逐元素乘,$\boldsymbol{y}$ 为 $r\times r$ 大小的过滤器,$\boldsymbol{x}$ 为 $(m+r-1)\times(m+r-1)$ 大小的输入特征图块,$\boldsymbol{A}^{\mathrm{T}}$,$\boldsymbol{P}$ 和 $\boldsymbol{Q}^{\mathrm{T}}$ 为与 $F(m\times m,r\times r)$ 算法匹配的转换系数矩阵,其值由 m 和 r 来决定,如 $F(4\times4,3\times3)$ 的转换系数矩阵为

$$\boldsymbol{A}^{\mathrm{T}}=\begin{pmatrix}1&1&1&1&1&0\\0&1&-1&2&-2&0\\0&1&1&4&4&0\\0&1&-1&8&-8&1\end{pmatrix},$$

$$\boldsymbol{P}=\begin{pmatrix}\frac{1}{4}&0&0\\-\frac{1}{6}&-\frac{1}{6}&-\frac{1}{6}\\-\frac{1}{6}&\frac{1}{6}&-\frac{1}{6}\\\frac{1}{24}&\frac{1}{12}&\frac{1}{6}\\\frac{1}{24}&-\frac{1}{12}&\frac{1}{6}\\0&0&1\end{pmatrix},$$

$$\boldsymbol{Q}^{\mathrm{T}}=\begin{pmatrix}4&0&-5&0&1&0\\0&-4&-4&1&1&0\\0&4&-4&-1&1&0\\0&-2&-1&2&1&0\\0&2&-1&-2&1&0\\0&4&0&-5&0&1\end{pmatrix}.$$

将 $F(m\times m,r\times r)$ 算法用于实现核大小为 $r\times r$ 的卷积计算的原始步骤为:1)将输入 $\boldsymbol{I}$ 的每个通道特征图划分为 $(m+r-1)\times(m+r-1)$ 大小的子块.相邻子块之间存在 $r-1$ 个元素重叠,从而每个输入通道产生 $\lceil H_o/m\rceil\lceil W_o/m\rceil$ 个块;2)调用 $F(m\times m,r\times r)$ 算法进行每个子块的计算;3)在单个通道内进行块之间的合并;4)在所有通道内进行累加求和.令 (α,β) 为单个通道内子块的坐标,则单个输出子块的计算为

$$\boldsymbol{O}_{b,k,\alpha,\beta}=\sum_{c=1}^{C}\boldsymbol{A}^{\mathrm{T}}[\boldsymbol{g}_{k,c}\odot\boldsymbol{d}_{b,c,\alpha,\beta}]\boldsymbol{A},\tag{4}$$

其中,$\boldsymbol{g}_{k,c}=\boldsymbol{P}\boldsymbol{F}_{k,c}\boldsymbol{P}^{\mathrm{T}}$,$\boldsymbol{d}_{b,c,\alpha,\beta}=\boldsymbol{Q}^{\mathrm{T}}\boldsymbol{I}_{b,c,\alpha,\beta}\boldsymbol{Q}$.

通过交换逐元素乘与累加顺序以及相关维度变换,式(4)最终转换为

$$\boldsymbol{O}_{b,k,\alpha,\beta}=\boldsymbol{A}^{\mathrm{T}}\boldsymbol{U}_{k,b,\alpha,\beta}\boldsymbol{A},\tag{5}$$

$$\boldsymbol{U}^{(\varphi,\gamma)}=\boldsymbol{G}^{(\varphi,\gamma)}\boldsymbol{D}^{(\varphi,\gamma)},\tag{6}$$

其中,(φ,γ) 表示逐元素乘中每个元素的坐标,$\boldsymbol{G}_{k,c}^{(\varphi,\gamma)}=(\boldsymbol{P}\boldsymbol{F}_{k,c}\boldsymbol{P}^{\mathrm{T}})^{(\varphi,\gamma)}$,$\boldsymbol{D}_{c,b,\alpha,\beta}^{(\varphi,\gamma)}=(\boldsymbol{Q}^{\mathrm{T}}\boldsymbol{I}_{b,c,\alpha,\beta}\boldsymbol{Q})^{(\varphi,\gamma)}$.

根据式(5)与式(6),Winograd 快速卷积算法的原始实现[3]如算法 1 所示.

算法 1. 基于 Winograd 最小滤波算法 $F(m\times m,r\times r)$ 的快速卷积算法.

输入:输入特征图 $\boldsymbol{I}$、卷积核为 $\boldsymbol{F}$;

输出:输出特征图 $\boldsymbol{O}$.

令 $\delta\times\delta=(m+r-1)\times(m+r-1)$ 为输入块的大小,相邻块之间重叠 $r-1$ 个元素;$\Gamma\times\Lambda=\lceil H_o/m\rceil\times\lceil W_o/m\rceil$ 为每个通道特征图上块的数量;$\boldsymbol{F}_{k,c}\in\mathbb{R}^{r\times r}$ 为第 k 个输出通道、第 c 个输入通道上的卷积核;$\boldsymbol{I}_{b,c,\alpha,\beta}\in\mathbb{R}^{\delta\times\delta}$ 为输入特征图中第 b 个样本上第 c 个输入通道上块坐标为(α,β)的分块;$\boldsymbol{O}_{b,k,\alpha,\beta}\in\mathbb{R}^{m\times m}$ 为输出特征图中第 b 个样本上第 k 个输出通道上块坐标为(α,β)的分块.

/* 卷积核转换 */

① for $k=0:1:K$ do

② for $c=0:1:C$ do

③ $\boldsymbol{g}=\boldsymbol{P}\boldsymbol{F}_{k,c}\boldsymbol{P}^{\mathrm{T}}\in\mathbb{R}^{\delta\times\delta}$;

④ 将$\boldsymbol{g}$ 中元素分散(scatter)到 $\boldsymbol{G}$ 中:$\boldsymbol{G}_{k,c}^{(\varphi,\gamma)}=\boldsymbol{g}_{\varphi,\gamma}$;

⑤ end for

⑥ end for

/* 输入特征图转换 */

⑦ for $b=0:1:B$ do

⑧ for $c=0:1:C$ do

⑨ for $\alpha=0:1:\Gamma$ do

⑩ for $\beta=0:1:\Lambda$ do

⑪ $\boldsymbol{d}=\boldsymbol{Q}^{\mathrm{T}}\boldsymbol{I}_{b,c,\alpha,\beta}\boldsymbol{Q}\in\mathbb{R}^{\delta\times\delta}$;

⑫ 将$\boldsymbol{d}$ 中元素分散(scatter)到 $\boldsymbol{D}$ 中:$\boldsymbol{D}_{c,b,\alpha,\beta}^{(\varphi,\gamma)}=\boldsymbol{d}_{\varphi,\gamma}$;

⑬ end for

⑭ end for

⑮ end for

⑯ end for

/* 逐元素乘 */

⑰ for $\varphi=0:1:\delta$ do

⑱ for $\gamma=0:1:\delta$ do

⑲ $\boldsymbol{U}^{(\varphi,\gamma)}=\boldsymbol{G}^{(\varphi,\gamma)}\boldsymbol{D}^{(\varphi,\gamma)}$; /* 通过调用线性代数库中通用矩阵乘函数来完成计算 */

⑳ end for

㉑ end for

/* 输出特征图逆转换 */

㉒ for $b=0:1:B$ do

㉓ for $k=0:1:K$ do

㉔ for $\alpha=0:1:\Gamma$ do

㉕ for $\beta=0:1:\Lambda$ do

㉖ 从$\boldsymbol{U}$ 中收集(gather)元素到 $\boldsymbol{u}$ 中:$\boldsymbol{u}_{\varphi,\gamma}=\boldsymbol{U}_{k,b,\alpha,\beta}^{(\varphi,\gamma)}$;

㉗ $\boldsymbol{O}_{b,k,\alpha,\beta}=\boldsymbol{A}^{\mathrm{T}}\boldsymbol{u}\boldsymbol{A}$;

㉘ end for

㉙ end for

㉚ end for

㉛ end for

2 飞腾多核处理器

飞腾多核处理器[15-16]是国防科技大学计算机学院自主研制的高性能 64 位通用多核 CPU,拥有 FT-1500A 和 FT-2000plus 两代产品,均包含桌面版与服务器版.当前,两代产品已广泛应用于相关领域的信息化与智能化建设中.为充分开发飞腾多核处理器的计算性能,必须了解其关键体系结构特征.本文主要面向飞腾多核处理器的服务器版进行优化,FT-1500A 和 FT-2000plus 两代产品服务器版均具有以下相同关键体系结构特征:

1) 兼容 ARMv8 指令集;

2) 拥有大量并行计算核;

3) 单个计算核既集成了标量处理单元,又集成了 128 bit 宽的向量处理单元,其中向量处理单元每周期可以执行 4 个单精度浮点乘加(FMA)操作;

4) 每个核每周期可以发射 2 个 Load 操作,或者 1 个 Load 操作和 1 个 Store 操作;

5) 单个计算核拥有 32 KB 的一级数据缓存;

6) 每 4 个计算核共享 2 MB 的二级缓存;

7) 支持硬件维护的片上一致性协议;

8) 采用高吞吐率的多通道内存.

兼容 ARMv8 指令集使得面向飞腾多核处理器的算法优化中可以部分调用 Neon intrinsics 函数[17]来实现向量化,而不必全部采用汇编指令,保证算法性能的前提下降低算法实现工作量.大量并行计算核、128 b 宽向量单元以及并行功能单元能够分别为 Winograd 快速卷积算法提供线程级、数据级以及指令级等多级并行计算能力.多级 cache 和多通道内存能够为 Winograd 快速卷积算法提供快速的数据访问能力.因而,本文将基于以上全部体系结构特征来进行算法的优化.

3 并行 Winograd 快速卷积算法与优化

本节将首先对本文提出的并行 Winograd 快速卷积算法进行整体介绍;然后详细介绍所采用的优化方法,并以 $F(4\times4,3\times3)$ 为例分别进行举例说明;最后对本文提出的优化方法进行可移植性分析.

3.1 算法整体设计

与直接卷积相比,Winograd 快速卷积算法通过一系列加法置换乘法的方式有效降低了卷积计算的算术复杂度[3].然而,从算法整体实现来看,存储访问量并没有降低,使得卷积算法整体的计算访存比变小.根据 Roofline 模型[18]可知,更小的计算访存比将使得算法的实现性能受到存储带宽的影响变得更大,给面向飞腾多核处理器上的并行 Winograd 快速卷积算法优化带来了巨大挑战.

结合 Winograd 快速卷积的计算访存特点和飞腾多核处理器的体系结构特征,本文提出了如算法 2 所示的并行 Winograd 快速卷积算法.与算法 1 所示的原始算法相比,本文提出的算法 2 由卷积核转换、输入特征图转换、逐元素乘以及输出特征图转换等 4 个多级并行部分组成,具有 5 个特点:

1) 不依赖通用矩阵乘库函数;

2) 将卷积核与输入特征图转换过程中的 scatter 操作与矩阵乘中的数据打包操作融为一体,即:在数据 scatter 时,直接按矩阵乘算法实现中的数据访问顺序进行存储,避免数据在内存中不必要的搬移;

3) 数据 scatter 和 gather 的最小粒度均设为飞腾多核处理器的向量单元能并行处理的元素个数 S,降低数据 scatter 和 gather 的总访存开销;

4) 设计了与上述方案相匹配的数据布局,确保算法实现过程中的最大局部性;

5) 提出了与上述方案相匹配的面向多核共享 cache 的多级并行数据转换与矩阵乘算法,有效提升算法整体性能.

算法 2. 面向飞腾多核处理器的并行 Winograd 快速卷积算法(基于 Winograd 最小滤波算法 $F(m\times m,r\times r)$).

输入:输入特征图 $\boldsymbol{I}$、卷积核 $\boldsymbol{F}$;

输出:输出特征图 $\boldsymbol{O}$.

令 $\delta\times\delta=(m+r-1)\times(m+r-1)$为输入块的大小,相邻块之间重叠 $r-1$ 个元素;S 为飞腾多核处理器的向量单元能并行处理的元素个数;$J=\Gamma\times\Lambda=\lceil H_o/m\rceil\times\lceil W_o/m\rceil$为每个通道特征图上块的数量;$\boldsymbol{F}_{k,c}\in\mathbb{R}^{r\times r}$为第 k 个输出通道、第 c 个输入通道上的卷积核;$\boldsymbol{I}_{b,c,\mu}\in\mathbb{R}^{\delta\times\delta}$为输入特征图中第 b 个样本上第 c 个输入通道上第 μ 个分块;$\boldsymbol{O}_{b,k,\alpha,\beta}\in\mathbb{R}^{m\times m}$为输出特征图中第 b 个样本上第 k 个输出通道上块坐标为(α,β)的分块.

/* 卷积核转换 */

① for $ks=0:K_r:K$ do in parallel
② for $cs=0:S:C$ do in parallel
③ for $cf=0:1:S$ do
④ for $kf=0:1:K_r$ do
⑤ $k=ks+kf$, $c=cs+cf$;
⑥ $\boldsymbol{g}^{\mathrm{T}}=\boldsymbol{P}(\boldsymbol{P}\boldsymbol{F}_{k,c})^{\mathrm{T}}\in\mathbb{R}^{\delta\times\delta}$;
⑦ 将$\boldsymbol{g}^{\mathrm{T}}$ 中元素分散(scatter)到 $\boldsymbol{G}$ 中,scatter 的最小粒度为 S 个元素;
⑧ end for
⑨ end for
⑩ end for
⑪ end for

/* 输入特征图转换 */

⑫ for $bs=0:B_r:B$ do in parallel
⑬ for $cs=0:S:C$ do in parallel
⑭ for $\mu=0:1:\Gamma\times\Lambda$ do
⑮ for $cf=0:1:S$ do
⑯ for $bf=0:1:B_r$ do
⑰ $b=bs+bf$, $c=cs+cf$;
⑱ $\boldsymbol{d}^{\mathrm{T}}=\boldsymbol{Q}^{\mathrm{T}}(\boldsymbol{Q}^{\mathrm{T}}\boldsymbol{I}_{b,c,\mu})^{\mathrm{T}}\in\mathbb{R}^{\delta\times\delta}$;
⑲ 将$\boldsymbol{d}^{\mathrm{T}}$ 中元素分散(scatter)到 $\boldsymbol{D}$ 中,scatter 的最小粒度为 S 个元素;
⑳ end for
㉑ end for
㉒ end for
㉓ end for
㉔ end for

/* 逐元素乘 */

㉕ for $\eta=0:S:\delta^2$ do
㉖ for $cs=0:C_{l1}:C$ do
㉗ for $ks=0:K_{l2}:K$ do in parallel
㉘ for $bs=0:B_r:B$ do in parallel
㉙ for $\mu=0:1:\Gamma\times\Lambda$ do in parallel
㉚ for $kf=0:K_r:K_{l2}$ do
/* Kernel */
㉛ $\dot{\boldsymbol{u}}[B_r][K_r][S]+=\sum_{cf=0}^{C_{l1}}\dot{\boldsymbol{g}}_{cf}[K_r][S]\times\dot{\boldsymbol{d}}_{cf}[B_r][S]$;
㉜ 将 $\dot{\boldsymbol{u}}$ 存回到 $\boldsymbol{U}$ 中;
㉝ end for
㉞ end for
㉟ end for
㊱ end for

```
㊲    end for
㊳ end for
/* 输出特征图逆转换 */
㊴ for ks=0:K_r:K do in parallel
㊵   for bs=0:B_r:B do in parallel
㊶     for α=0:1:Γ do
㊷       for β=0:1:Λ do
㊸         for bf=0:1:B_r do
㊹           for kf=0:1:K_r do
㊺             b=bs+bf, k=ks+kf;
㊻             从U 中聚集(gather)元素到 u
                 中,gather 的最小粒度为 S
                 个元素;
㊼             O_{b,k,α,β}=A^T (A^T u^T)^T;
㊽           end for
㊾         end for
㊿       end for
(51)     end for
(52)   end for
(53) end for
```

3.2 数据布局

数据布局是指数据在内存中的存储方式,是决定并行算法性能的关键因素之一.在本算法设计中,影响数据布局的主要因素有 3 个:

1) 为避免本文算法在与主流机器学习框架集成时出现不必要的数据布局转换开销,算法设计过程中采用的输入张量(输入特征图 $\boldsymbol{I}$ 和卷积核 $\boldsymbol{F}$)与输出张量(输出特征图 $\boldsymbol{O}$)的数据布局应该与主流机器学习框架保持一致;

2) 在算法整体设计层面,将卷积核和输入特征图转换过程中的 scatter 操作与矩阵乘实现中的数据打包操作融为一体,且 scatter 和 gather 的最小粒度均为 S,从而相应内部张量($\boldsymbol{G}$,$\boldsymbol{D}$ 和 $\boldsymbol{U}$)的数据布局设计应该与矩阵乘算法实现中的访存顺序保持一致;

3) 在代码实现层面,为保证向量化实现的性能以及最小化 cache 与 TLB 缺失,要求数据布局设计应尽可能满足算法实现过程中对齐访问以及较小的访存范围的需求.

综合考虑这 3 个因素,本文设计了如表 1 所示的数据布局.输入特征图 $\boldsymbol{I}$ 与输出特征图 $\boldsymbol{O}$ 采用了 Mxnet[19],Caffe[20] 等主流框架中采用的 BCHW 格式,卷积核 $\boldsymbol{F}$ 采用对应的匹配格式,从而使得本算法不需数据布局的转换就可以直接集成到 Mxnet,Caffe 等框架中;内部张量 $\boldsymbol{G}$,$\boldsymbol{D}$ 和 $\boldsymbol{U}$ 分别用来存储矩阵乘打包后的输入与输出,其格式是由矩阵乘的实现来决定的,B_r,K_r 以及 C_{l1} 等为矩阵乘实现过程中所采用的不同层级分块参数.此外,所有张量在内存中均采用行优先方式进行存储.

Table 1 Data Layout of Tensors in Parallel Winograd-Based Fast Convolution Algorithm

表 1 并行 Winograd 快速卷积算法中主要张量的数据布局

Tensor	Element	Location in the Tensor (stored in memory)	Comment
Filters	$\boldsymbol{F}_{k,c,hf,wf}$	$\boldsymbol{F}[k][c][hf][wf]$	
Transformed Filters	$\boldsymbol{G}_{k,c}^{(\varphi,\gamma)}$	$\boldsymbol{G}[\eta/S][c/C_{l1}][k/K_r][c \bmod C_{l1}][k \bmod K_r][\eta \bmod S]$	$\eta=\varphi\times\delta+\gamma$
Input Feature Maps	$\boldsymbol{I}_{b,c,hi,wi}$	$\boldsymbol{I}[b][c][hi][wi]$	
Transformed Inputs	$\boldsymbol{D}_{b,c}^{(\mu,\varphi,\gamma)}$	$\boldsymbol{D}[\eta/S][c/C_{l1}][b/B_r][\mu][c \bmod C_{l1}][b \bmod B_r][\eta \bmod S]$	$\eta=\varphi\times\delta+\gamma$
Transformed Outputs	$\boldsymbol{U}_{b,k}^{(\mu,\varphi,\gamma)}$	$\boldsymbol{U}[k/K_r][b/B_r][\mu][\eta/S][b \bmod B_r][k \bmod K_r][\eta \bmod S]$	$\eta=\varphi\times\delta+\gamma$
Output Feature Maps	$\boldsymbol{O}_{b,k,ho,wo}$	$\boldsymbol{O}[b][k][ho][wo]$	

3.3 数据转换

Winograd 快速卷积算法的数据转换主要涉及卷积核转换、输入特征图转换、输出特征图逆转换等 3 个过程.从数据转换过程的计算式可知,由于转换系数矩阵的稀疏性,3 个过程均属于访存密集型计算.因而,在面向飞腾多核处理器的并行数据转换设计中,基于多级存储层次提升数据的局部性来降低存储开销是实现高性能数据转换的关键.

在 3 个转换过程中,当 m 和 r 的大小确定以后,转换系数矩阵 $\boldsymbol{A}^{\mathrm{T}}$,$\boldsymbol{P}$ 和 $\boldsymbol{Q}^{\mathrm{T}}$ 均为常数.对式(3)中数据转换进行转置变换,如式(7)~(10)所示.可以发现,经过转置变换以后,这些转换过程均变成了常数系数矩阵左乘变量矩阵,在本算法采用行优先方式存储矩阵的情况下,将有利于数据转换过程的向量化,提升数据转换过程中的访存效率.此外,在卷积核转换和输入特征图转换后将分别对产生的数据元素进行 scatter 操作.然而,逐元素乘部分(矩阵乘)将按照矩阵乘的访问顺序对数据元素进行打包(packing),

会再一次对数据元素进行移动操作.为进一步提升存储性能,有必要将2次数据操作合为一体.

$$\boldsymbol{g}=\boldsymbol{PyP}^{\mathrm{T}}\rightarrow\boldsymbol{g}^{\mathrm{T}}=\boldsymbol{P}(\boldsymbol{Py})^{\mathrm{T}}. \tag{7}$$

$$\boldsymbol{d}=\boldsymbol{Q}^{\mathrm{T}}\boldsymbol{xQ}\rightarrow\boldsymbol{d}^{\mathrm{T}}=\boldsymbol{Q}^{\mathrm{T}}(\boldsymbol{Q}^{\mathrm{T}}\boldsymbol{x})^{\mathrm{T}}. \tag{8}$$

$$\boldsymbol{u}=\boldsymbol{g}\odot\boldsymbol{d}\rightarrow\boldsymbol{u}^{\mathrm{T}}=\boldsymbol{g}^{\mathrm{T}}\odot\boldsymbol{d}^{\mathrm{T}}. \tag{9}$$

$$\boldsymbol{z}=\boldsymbol{A}^{\mathrm{T}}\boldsymbol{uA}\rightarrow\boldsymbol{z}=\boldsymbol{A}^{\mathrm{T}}(\boldsymbol{A}^{\mathrm{T}}\boldsymbol{u}^{\mathrm{T}})^{\mathrm{T}}. \tag{10}$$

根据上述分析,本文基于式(7)~(10)提出了基于scatter与packing为一体的数据转换算法,卷积核转换、输入特征图转换以及输出特征图逆转换分别如算法2中行①~⑪、行⑫~㉔、行㊴~㊸所示.

在线程级并行方面,卷积核转换在输出通道和输入通道2个维度上进行,输入特征图转换在批大小、输入通道2个维度上进行,输出特征图逆转换在批大小、输出通道2个维度上进行.并且,批大小、输入通道、输出通道3个维度上线程级调度的最小粒度分别为B_r,S和K_r,其中B_r和K_r为3.4节矩阵乘实现过程中的分块参数,S为飞腾多核处理器的向量单元能并行处理的元素个数.

鉴于矩阵系数矩阵$\boldsymbol{A}^{\mathrm{T}}$,$\boldsymbol{P}$和$\boldsymbol{Q}^{\mathrm{T}}$对于固定的$m$和$r$来说均为稀疏性常数矩阵,在常数系数左乘变量矩阵如$\boldsymbol{Py}$,$\boldsymbol{Q}^{\mathrm{T}}\boldsymbol{x}$以及$\boldsymbol{A}^{\mathrm{T}}\boldsymbol{u}^{\mathrm{T}}$等实现方面,本文采用手工重排的方法直接消除函数实现中不必要以及重复的操作,进一步降低计算复杂度,并采用ARM Neon intrinsics函数来实现转换过程中的向量化计算.在scatter部分,直接按照矩阵乘的访问顺序对数据进行存储,避免重复的数据移动操作.同时,scatter和gather部分的最小数据粒度均设为S个元素,将有利于改善cache局部性,有效降低scatter和gather时的访存次数,提升整体性能.此处的最小粒度主要是由飞腾多核处理器中cache的缓存行大小以及3.4节矩阵乘优化实现所决定.

以$F(4\times4,3\times3)$的输入特征图转换为例,本文提出的优化算法与原始实现对比如图1所示.输入特征图$\boldsymbol{I}$的每个通道特征图被划分为$J=\Gamma\times\Lambda$个6×6大小的子块,相邻子块之间存在2个元素重叠.在原始输入特征图转换中,每个6×6大小的输入子块经过$\boldsymbol{Q}^{\mathrm{T}}\boldsymbol{xQ}$转换后获得$6\times6$大小的张量,然后将其中的每个元素分散到36个不同的位置,输入特征图$\boldsymbol{I}$的所有子块均转换完毕后,最后形成36个$C\times(B\times J)$大小的张量.从图1可以发现,6×6大小张量的元素分散过程需要36次存储操作,数据局部性较差,使得原始输入转换的开销特别大.在本文的优化输入特征图转换中,每个子块经过$\boldsymbol{Q}^{\mathrm{T}}(\boldsymbol{Q}^{\mathrm{T}}\boldsymbol{x})^{\mathrm{T}}$转换后依然得到$6\times6$大小的张量,通过9次向量存储操作分散到9个不同的位置,每次存储4个元素(飞腾多核处理器中一个缓存行可以存储4个单精度浮点数),数据局部性好,能有效降低输入转换的开销,同时直接按照3.4节分块矩阵乘算法的分块要求存储,又进一步降低了算法整体的访存开销.在输入转换过程中,每个子块之间的计算是相互独立的,考虑到多个子块输入数据的不连续性,因而采用向量并行与指令并行来开发每个子块转换内部的并行性,将不同批次和不同通道的转换分配到多个核中来完成.又考虑到3.4节中要求$C_{l1}\times B_r\times4$个元素是连续存储的,从而将数据转换分配到多核中进行线程级并行化时,批大小和输入通道上的最小划分粒度

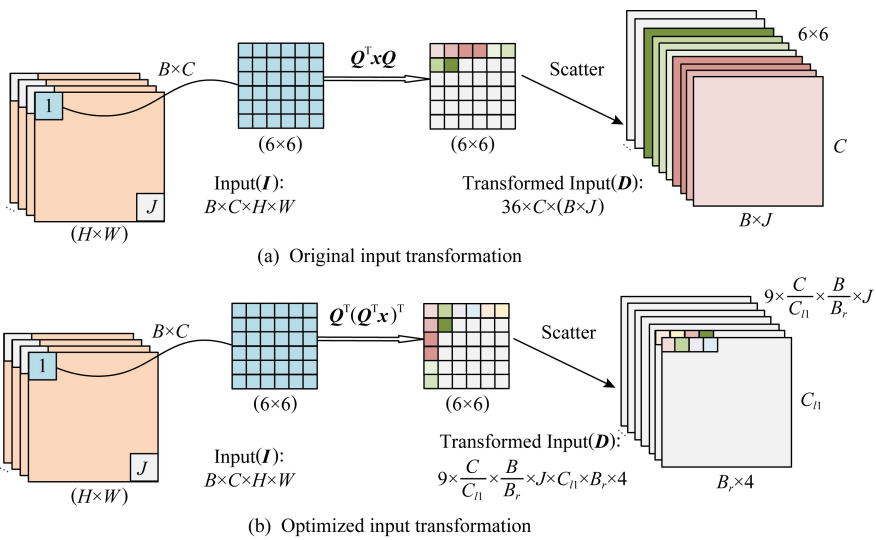


Fig. 1 Input transformations about $F(4\times4,3\times3)$

图1 $F(4\times4,3\times3)$算法的输入转换

分别设为 B_r 和 S,进一步优化转换过程中单线程的数据局部性,如算法 2 中行⑫~⑬所示.

3.4 矩阵乘

在逐元素乘部分,实质上是执行 δ^2 个矩阵大小分别为 $K\times C$ 与 $C\times(B\times\Gamma\times\Lambda)$ 的实数矩阵乘.从并行计算的角度来看,可以从 δ^2,K 以及 $B\times\Gamma\times\Lambda$ 等 3 个维度进行并行处理.在飞腾多核处理器中,即可利用单核中的向量单元进行数据级并行计算,也可利用多个核心进行线程级并行处理.为有效开发飞腾多核处理器的计算潜力,需要在向量数据级并行与多核线程级并行之间进行权衡设计.同时考虑到矩阵是由数据转换过程中采用 scatter 方式构建的,以及还将基于矩阵乘的结果采用 gather 方式构建输出特征图逆变换的输入小矩阵 $\boldsymbol{u}$,本文提出了如算法 2 中行㉕~㊳所示的矩阵乘实现.

本文选择采用向量单元来同时计算 S 个矩阵乘.由于飞腾多核处理器的片上存储通常至少依次由寄存器、一级缓存和二级缓存等层次构成,本文也通过矩阵分块的方式来提升每一层次的数据重用性.矩阵 $\boldsymbol{G}$,$\boldsymbol{D}$ 和 $\boldsymbol{U}$ 分别被划分成大小为 $C_{l1}\times K_r\times S$ 的子矩阵 $\dot{\boldsymbol{g}}$、大小为 $C_{l1}\times B_r\times S$ 的子矩阵 $\dot{\boldsymbol{d}}$ 以及大小为 $B_r\times K_r\times S$ 的子矩阵 $\dot{\boldsymbol{u}}$.每个子矩阵 $\dot{\boldsymbol{u}}$ 的计算方法为

$$\dot{\boldsymbol{u}}_{bf,ko,i}\ +=\sum_{cf=0}^{C_{l1}}\dot{\boldsymbol{g}}_{cf,ko,i}\times\dot{\boldsymbol{d}}_{cf,bf,i},\tag{11}$$

其中,$0\leqslant bf<B_r$,$0\leqslant ko<K_r$ 以及 $0\leqslant i<S$.并且,在每个子矩阵 $\dot{\boldsymbol{u}}$ 的计算过程中,$\dot{\boldsymbol{u}}$ 的所有元素均将保留在寄存器中直到不再使用为止,从而 $\dot{\boldsymbol{u}}$ 将在寄存器级被重用 C_{l1} 次.为 $\dot{\boldsymbol{g}}$,$\dot{\boldsymbol{d}}$ 以及 $\dot{\boldsymbol{u}}$ 子矩阵分别分配 K_r,B_r 以及 $B_r\times K_r$ 个向量寄存器.全部使用的寄存器个数 $B_r+K_r+B_r\times K_r$ 受限于飞腾多核处理器中总的向量寄存器数.同时,式(11)计算过程中的 3 个矩阵也将全部被填充进飞腾处理器的一级缓存才能获得最好的性能,这也使得 3 个子矩阵大小将受限于一级缓存的大小.根据参考文献[21],式(11)计算过程中的计算访存比 γ 为

$$\gamma=\frac{2\times B_r\times K_r\times C_{l1}}{K_r\times C_{l1}+(2K_r+C_{l1})\times B_r}.\tag{12}$$

从而,在以上相关限制条件下,也应该使得计算访存比 γ 的值尽可能大.

子矩阵 $\dot{\boldsymbol{u}}$ 计算过程的外层循环顺序主要是由一级缓存和二级缓存的数据重用,以及相关的数据布局所决定的.本文选择在一级缓存和二级缓存中分别重用子矩阵 $\dot{\boldsymbol{d}}$ 和 $\dot{\boldsymbol{g}}$.K_{l2} 控制着 $\dot{\boldsymbol{d}}$ 在一级缓存中的重用次数,同时也受到二级缓存大小的限制.在二级缓存中重用 $\boldsymbol{G}$ 的子矩阵 $\dot{\boldsymbol{g}}$,重用次数主要由外层 bs 和 μ 循环的迭代次数决定,从而 2 个迭代次数在设计中应尽可能最大化.

在飞腾多核处理器的体系结构中,一共有 32 个向量寄存器可以使用.在本文的设计中,K_r 和 B_r 的大小均取为 4,采用手工汇编代码完成子矩阵 $\dot{\boldsymbol{u}}$ 的计算(算法 2 行㉛),其中通过向量计算指令、访存指令以及预取指令的交错排布,实现向量级并行、指令级并行以及计算与访存的重叠.在算法 2 的行㉜,将 $\dot{\boldsymbol{u}}$ 存回内存 $\boldsymbol{U}$ 中时,本文将矩阵乘结果的 δ^2/S 维置于 $\Gamma\times\Lambda$ 维之后,如表 1 中 $\boldsymbol{U}$ 的数据布局所示,可有效降低输出特征逆变换的 gather 操作的访存范围,在对矩阵乘性能影响较小的情况下较大幅度地提升输出特征逆变换的访存性能.在线程级并行方面,在输出通道、批大小、单个特征图的子块数等 3 个维度

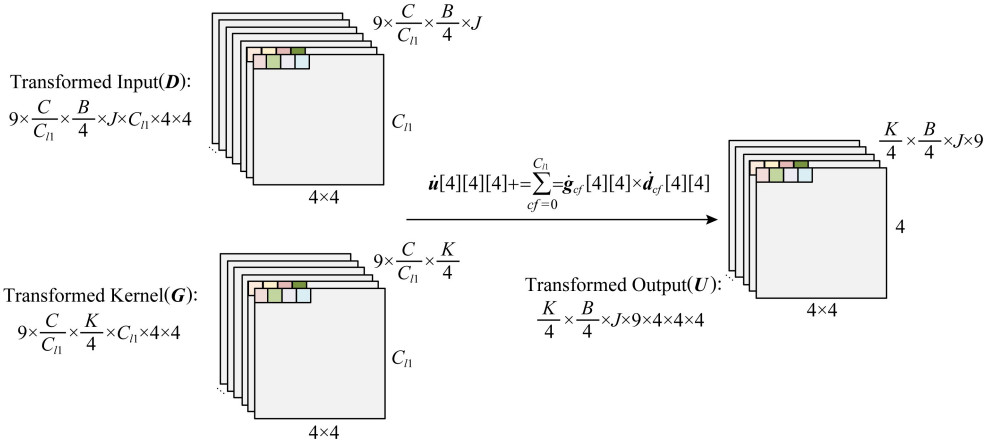


Fig. 2 Optimized matrix multiplication about $F(4\times4,3\times3)$

图 2 $F(4\times4,3\times3)$算法的优化矩阵乘

上进行任务调度,其最小调度粒度分别为输出通道上的分块系数 K_{l2}、批大小上的分块系数 B_r 以及1,分别如算法2中行㉗～㉙所示.

由于在3.2节的数据转换实现中,已按矩阵乘的访问需求对数据进行了排列,使得在本节的矩阵乘中不再需要对数据元素进行打包.以 $F(4\times4,3\times3)$ 算法为例,本文提出的优化矩阵乘的输入输出如图2所示.在本文的优化实现中,输入为输入特征图和卷积核转换后按分块矩阵乘要求排列的输出结果 $\boldsymbol{D}$ 和 $\boldsymbol{G}$,分别为 $9\times(C/C_{l1})\times(B/B_r)\times J$ 个 $C_{l1}\times B_r\times4$ 大小的子矩阵和 $9\times(C/C_{l1})\times(K/K_r)$ 个 $C_{l1}\times K_r\times4$ 大小的子矩阵;输出 $\boldsymbol{U}$ 为 $(K/K_r)\times(B/B_r)\times J\times9$ 个 $B_r\times K_r\times4$ 大小的子矩阵,其中 $J=\Gamma\times\Lambda$ 和 $B_r=K_r=4$.在输出 $\boldsymbol{U}$ 中,将来自同一输出子块的 9×4 个元素尽可能存储在较小范围的内存空间中,有利于降低输出逆转换过程中的数据gather开销.采用向量级并行与指令级并行来开发输入大小分别为 $C_{l1}\times4\times4$ 和 $C_{l1}\times4\times4$ 的矩阵乘中的并行性,在平衡数据重用的基础上,将多个子矩阵乘分发到多个并行计算核中,如算法2中行㉗～㉙所示.

3.5 可移植性分析

如3.1节叙述,本文并行Winograd快速卷积算法整体设计思想是实现Winograd相关转换与矩阵乘的一体化设计,不依赖外部计算库,是与硬件平台体系结构无关的.尽管如此,3.2～3.4节所介绍的相关优化细节均是与第2节所介绍的飞腾多核体系结构关键体系结构特征强相关的.因此,本文提出的面向飞腾多核处理器的并行Winograd快速卷积算法优化对于面向其他硬件平台的Winograd快速卷积算法实现具有一定的借鉴作用,但还需面向目标硬件平台的关键体系结构特征进行进一步的优化才能获得好的性能.

4 实验结果与分析

本节介绍并行Winograd快速卷积算法与开源库的性能对比测试,以及将该算法集成到典型深度学习框架后进行神经网络批量前向计算时的性能加速比测试.

4.1 实验设置

本文的测试实验是在飞腾FT-1500A 16核处理器和FT-2000plus 64核处理器上完成的,两款处理器的详细配置如表2所示.由于FT-2000plus 64核处理器属于非一致性内存架构(non-uniform memory architecture, NUMA),在本文的测试中均采用Linux的numactl工具来实现数据在所有节点之间的均匀分布.

Table 2 Specifications of Phytium Multi-core Processors

表2 飞腾多核处理器的详细配置

Parameter	Phytium FT-1500A	Phytium FT-2000plus
Architecture	ARMv8	ARMv8
Frequency/GHz	1.5	2.3
Cores	16	8 Panels, 8 Cores/Panel
L1D Cache	32 KB/Core	32 KB/Core
L2 Cache	2 MB/4 Cores	2 MB/4 Cores
L3 Cache	8 MB/16 Cores	

本文主要实现了 $F(4\times4,3\times3)$ 和 $F(6\times6,3\times3)$ 两种Winograd算法.在卷积层性能对比部分,将本文并行Winograd快速卷积算法分别与ARM开发的计算库(ARM compute library, ACL)[13]中 $F(4\times4,3\times3)$ 算法以及NNPACK库[22]中 $F(6\times6,3\times3)$ 算法进行了对比,其中ACL的版本是v19.02.测试中的卷积层配置囊括了VGG16[23],AlexNet[24]以及ResNet[25]网络中的所有3×3卷积层.

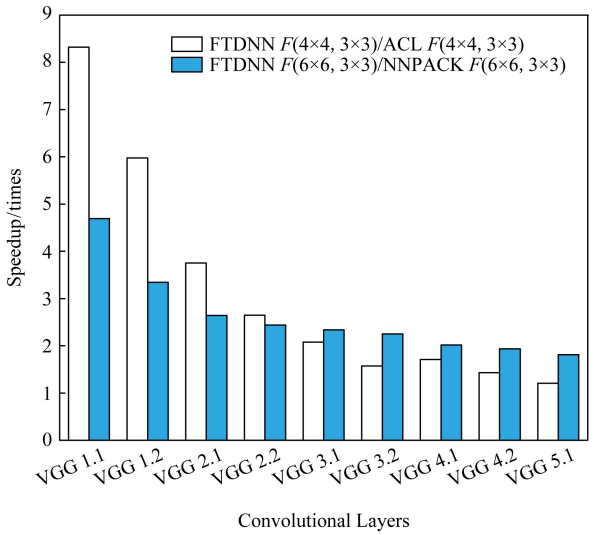
在卷积神经网络性能测试部分,将本文并行Winograd快速卷积算法集成到Mxnet v1.50框架中形成了Mxnet新版本,标记为Mxnet-FTDNN.原始Mxnet v1.50版本标记为Mxnet-origin.基于Mxnet-FTDNN和Mxnet-origin两个版本,分别测试了不同批大小下VGG16网络的前向计算性能,然后进行了加速比的计算.

本文所有测试均是迭代运行10次,取运行时间或者处理速度的中位值为最终结果.在后面性能分析部分,均采用FTDNN来标记本文提出的算法.

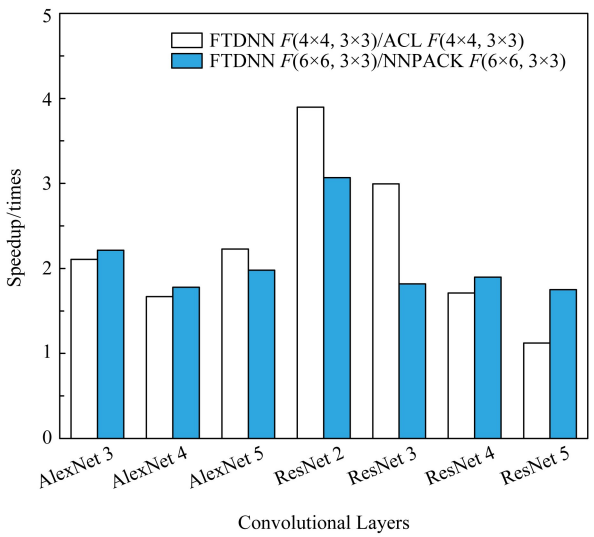
4.2 卷积层性能比较

本文算法基于开源库ACL和NNPACK的加速比,在飞腾FT-1500A与FT-2000plus处理器上的测试结果分别如图3和图4所示.其中,横坐标表示网络结构中的不同网络层,纵坐标表示加速比.由于ACL和NNPACK分别仅支持 $F(4\times4,3\times3)$ 和 $F(6\times6,3\times3)$,因而分别进行相应的比较.

对于VGG16网络的卷积层,在飞腾FT-1500A处理器上,与ACL相比,本文算法可以实现1.20～8.30倍的性能加速;与NNPACK相比,加速比为1.8～4.70倍.在飞腾FT-2000plus处理器上,与ACL和NNPACK相比的加速比分别为1.05～16.11倍与



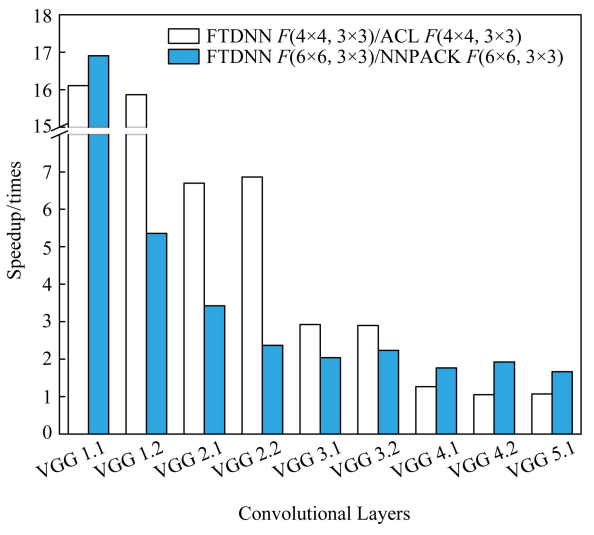
(a) VGG



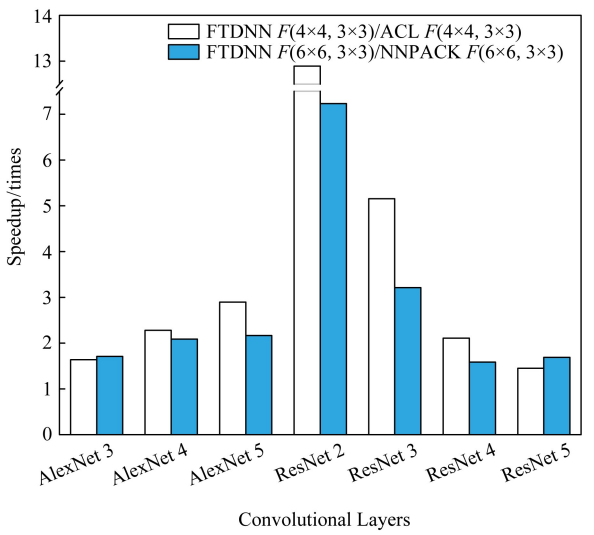
(b) AlexNet and ResNet

Fig. 3 Speedup between different Winograd-based convolution algorithms on FT-1500A

图 3 FT-1500A 上不同 Winograd 卷积算法之间的性能加速比



(a) VGG



(b) AlexNet and ResNet

Fig. 4 Speedup between different Winograd-based convolution algorithms on FT-2000plus

图 4 FT-2000plus 上不同 Winograd 卷积算法之间的性能加速比

1.66～16.90 倍.从趋势上来看,从 VGG16 的 Conv1.1 层到 Conv5.1 层,加速比逐渐下降.主要原因在于从 VGG 1.1 层到 VGG 5.1 层的输入输出特征图空间大小逐渐减小,通道数逐渐增加,在当前的批大小设置下数据转换部分对整个算法性能的影响整体呈逐渐下降趋势造成的.

对于 AlexNet 的 3 个 3×3 卷积层,输入输出特征图空间大小相同.与 ACL 相比,本文算法在 FT-1500A 和 FT-2000plus 上的加速比分别为 1.67～2.23 倍与 1.64～2.90 倍;与 NNPACK 相比,相应的加速比分别是 1.78～2.21 倍与 1.71～2.16 倍.

对于 ResNet 的 4 个 3×3 卷积层,从第 2 层到第 5 层,输入输出特征图空间大小逐渐减小,在 2 个平台上的加速比总体变化趋势与 VGG16 相似.与 ACL 相比,本文算法可获得 1.12～12.89 倍的加速;与 NNPACK 相比,加速比为 1.69～7.23 倍.

综上可知,与开源库 ACL 以及 NNPACK 中的 Winograd 快速卷积实现相比,本文算法分别能获得 1.05～16.11 倍与 1.66～16.90 倍的性能加速,证明了本文算法的有效性.并且,数据转换部分所占比重越大,加速效果越明显,说明了本文算法有效的最核心原因是融合设计与各个部分的优化设计有效改善

了整个 Winograd 快速卷积算法的访存效率,同时各部分的多级并行设计能够有效利用飞腾多核处理器的多级并行计算能力也是至关重要的.

4.3 卷积神经网路性能比较

与 Mxnet v1.50 原始版本 Mxnet-origin 相比,集成了本文并行 Winograd 快速卷积算法的 Mxnet 新版本 Mxnet-FTDNN 的性能加速比在飞腾 FT-1500A 与 FT-2000plus 处理器上的测试结果如图 5 所示.在飞腾 FT-1500A 处理器上,Mxnet-FTDNN 获得了 3.01～4.63 倍的性能加速.在飞腾 FT-2000plus 上,Mxnet-FTDNN 获得的最小与最大加速比分别为 3.81 倍和 6.79 倍.从趋势上看,在两款处理器上,Mxnet-FTDNN 与 Mxnet-origin 之间的性能加速比均随着批大小逐渐增加.主要原因在于,Winograd 快速卷积算法与 Mxnet 中原有卷积算法的算法复杂度之比随着批大小的增加而逐渐降低,从而使得本文实现与 Mxnet 原始版本相比的优势随批大小增加而变得更加明显.

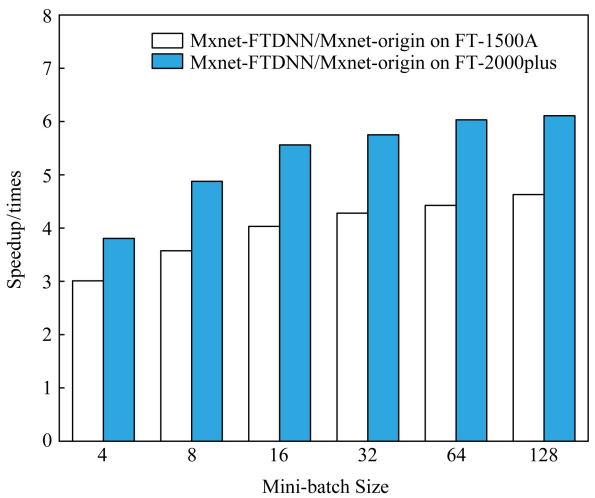


Fig. 5 Speedup between different versions of Mxnet

图 5 不同版本 Mxnet 之间的性能加速比

5 总结与展望

随着国产飞腾多核处理器在智能领域的应用,对于国产飞腾多核处理器上的高性能卷积实现提出了强烈需求.面向飞腾多核处理器的体系结构特征和 Winograd 快速卷积的计算特点,本文提出了一种并行 Winograd 快速卷积算法来实现飞腾多核处理器上的高性能卷积计算.该算法不依赖通用矩阵乘库函数,由卷积核转换、输入特征图转换、逐元素乘、输出特征图逆变换等 4 个部分构成,实现了卷积核转换和输入特征图转换部分数据 scatter 操作与逐元素乘部分的数据打包操作的融合,并设计了与之匹配的数据布局以及数据转换与矩阵乘多级并行算法.在 FT-1500A 16 核处理器与 FT-2000plus 64 核处理器上,与开源库 ACL 以及 NNPACK 中的对应 Winograd 快速卷积算法相比,本文算法获得了不同程度的性能加速.并且,将本文算法集成到 Mxnet 框架中,与原始 Mxnet 框架基于 VGG16 网络的前向计算性能比较,又进一步证明了本文算法对提升神经网络计算性能的有效性.

下一步将重点研究本文优化算法在其他主流平台上的适应性,同时面向智能应用的计算需求继续研究飞腾多核处理器上的深度学习并行优化技术.

参 考 文 献

[1] Lecun Y, Bengio Y, Hinton G. Deep learning [J]. Nature, 2015, 521(7553): 436-444

[2] Jia A, Shen Siqi, Li Dongsheng, et al. Predicting the implicit and the explicit video popularity in a user generated content site with enhanced social features [J]. Computer Networks, 2018, 140(JUL.20): 112-125

[3] Lavin A, Gray S. Fast algorithms for convolutional neural networks [C] //Proc of 2016 IEEE Conf on Computer Vision and Pattern Recognition. Piscataway, NJ: IEEE, 2016: 4013-4021

[4] Cho M, Brand D. MEC: Memory-efficient convolution for deep neural network [C/OL] //Proc of the 34th Int Conf on Machine Learning. 2017: 815-824 [2020-02-03]. http://proceedings.mlr.press/v70/cho17a.html

[5] Wang Qinglin, Mei Songzhu, Liu jie, et al, Parallel convolution algorithm using implicit matrix multiplication on multi-core CPUs [C/OL] //Proc of 2019 Int Joint Conf on Neural Networks. Piscataway, NJ: IEEE, 2019: 1-7. [2020-02-03]. https://doi.org/10.1109/IJCN N .2019.8852012

[6] Zhang Jiyuan, Franchetti F, Low T. High performance zero-memory overhead direct convolutions [C/OL] //Proc of the 35th Int Conf on Machine Learning. 2018: 5776-5785. [2020-02-03]. http://proceedings.mlr.press/v80/zhang18d.html

[7] Budden D, Matveev A, Santurkar S, et al. Deep tensor convolution on multicores [C/OL] //Proc of the 34th Int Conf on Machine Learning. 2017: 615-624. [2020-02-03]. http://proceedings.mlr.press/v70/budden17a.html

[8] Jia Z,Zlateski A, Durand F, et al. Optimizing N-dimensional, Winograd-based convolution for manycore CPUs [J]. ACM SIGPLAN Notices, 2018, 53(1): 109-123

[9] Xygkis A, Soudris D, Papadopoulos L, et al. Efficient Winograd-based convolution kernel implementation on edge devices [C/OL] //Proc of the 55th ACM/ESDA/IEEE Design Automation Conf. Piscataway, NJ: IEEE, 2018: 1-6. [2020-02-03]. https://doi.org/10.1109/D AC.2018.8465825

[10] Wang Zelong, Lan Qiang, He Hongjun, et al. Winograd algorithm for 3D convolution neural networks [C] //Proc of Int Conf on Artificial Neural Networks. Berlin: Spring, 2017: 609-616

[11] Wu Zheng, An Hong, Jin Xu, et al. Research and optimization of fast convolution algorithm Winograd on Intel platform [J]. Journal of Computer Research and Development, 2019, 56(4): 151-161 (in Chinese)
(武铮, 安虹, 金旭, 等. 基于 Intel 平台的 Winograd 快速卷积算法研究与优化[J]. 计算机研究与发展, 2019, 56(4): 151-161)

[12] Intel. Intel® math kernel library for deep neural networks [OL]. [2019-11-01]. https://intel.github.io/mkl-dnn/index.html

[13] ARM. ARM compute library [OL]. [2019-11-01]. https://arm-software.github.io/ComputeLibrary/v19.02/

[14] Chetlur S, Woolley C, Vandermersch P, et al. cuDNN: Efficient primitives for deep learning [J]. arXiv preprint, arXiv:1410.0759, 2014

[15] Phytium. FT-1500A/16 [OL]. [2019-11-01]. http://www.phytium.com.cn/Product/detail?language=1&product_id=9

[16] Phytium. FT-2000plus/64 [OL]. [2019-11-01]. http://www.phytium.com.cn/Product/detail?language=1&product_id=7

[17] ARM. Neon intrinsics reference [OL]. [2019-11-01]. https://developer.arm.com/architectures/instruction-sets/simd-isas/neon/intrinsics

[18] Williams S, Waterman A, Patterson D. Roofline: An insightful visual performance model for multicore architectures [J]. Communications of the ACM, 2009, 52(4): 65-76

[19] Chen Tianqi, Li Mu, Li Yutian, et al. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems [J]. arXiv preprint, arXiv:1512.01274, 2015

[20] Jia Yangqing, Shelhamer E, Donahue J, et al. Caffe: Convolutional architecture for fast feature embedding [J]. arXiv preprint, arXiv:1408.5093, 2014

[21] Goto K, Geijn R. Anatomy of high-performance matrix multiplication [J]. ACM Transactions on Mathematical Software, 2008, 34(3): 1-25

[22] Dukhan M. NNPACK [OL]. [2019-11-01]. https://github.com/Maratyszcza/NNPACK

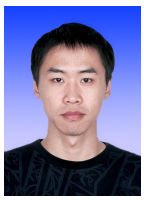
[23] Karen S, Andrew Z. Very deep convolutional networks for large-scale image recognition [J]. arXiv preprint, arXiv: 1409.1556, 2014

[24] Krizhevsky A, Sutskever I, Hinton G. ImageNet classification with deep convolutional neural networks [C] //Proc of the 25th Int Conf on Neural Information Processing Systems-Volume 1. Red Hook, New York: Curran Associates Inc., 2012: 1097-1105

[25] He Kaiming, Zhang Xiangyu, Ren Shaoqing, et al. Deep residual learning for image recognition [C] //Proc of 2016 IEEE Conf on Computer Vision and Pattern Recognition. Los Alamitos, CA: IEEE Computer Society, 2016: 770-778

Wang Qinglin, born in 1987. PhD, assistant professor. Member of CCF. His main research interests include parallel algorithm, reconfigurable computing and machine learning.

Li Dongsheng, born in 1978. PhD, professor, PhD supervisor. Senior member of CCF. His main research interests include parallel and distributed computing, high performance system software, and machine learning.



Mei Songzhu, born in 1984. PhD, assistant professor. Member of CCF. His main research interests include big data analysis, distributed computing, and machine learning.

Lai Zhiquan, born in 1986. PhD, assistant professor. Member of CCF. His main research interests include high performance system software, distributed machine learning and power-aware computing.

Dou Yong, born in 1969. PhD, professor, PhD supervisor. Senior member of CCF. His main research interests include high performance computer architecture, high performance embedded microprocessor, reconfigurable computing, and machine learning.