文章编号:1007－6735(2024)06－0610－10　DOI: 10.13255/j.cnki.jusst.20241016003

面向飞腾处理器平台的快速卷积算法优化

赵亚飞[1]，　杨耀功[2]，　王永刚[2]，　魏继增[1]
（1. 天津大学 智能与计算学部，天津 300354；2. 飞腾信息技术有限公司，天津 300459）

摘要： 为解决卷积神经网络难以在计算资源受限设备上部署的问题，面向国产 FT-2000/4 多核处理器提出一种高性能的快速卷积算法 FastInfer。采用分块策略优化通用矩阵乘法，将处理器访问频率高的数据存入更靠近处理器的缓存中，从而提高计算过程中的访存效率。配合分块方案设计实现高性能的矩阵乘法微内核，使用向量外积运算更新数据，提高计算访存比，实现最大程度掩盖访存指令的延迟。最终实验结果表明，FastInfer 在 FT-2000/4 处理器上的峰值计算性能达到 99.56 GFLOPS。在不同输入规模的通用矩阵乘法测试中，FastInfer 性能是 OpenBLAS 算法的 1.07 倍和 1.52 倍。在卷积测试中，FastInfer 性能是 ARM Compute Library 算法的 1.32 倍，实现了在 FT-2000/4 多核处理器上的高性能卷积计算。

关键词： 深度学习；快速卷积算法；并行计算；通用矩阵乘法
中图分类号： TP 391.41　　**文献标志码：** A

Fast convolution algorithm optimization for Phytium processor

ZHAO Yafei[1]，　YANG Yaogong[2]，　WANG Yonggang[2]，　WEI Jizeng[1]
(1. *College of Intelligence and Computing, Tianjin University, Tianjin 300354, China*;
2. *Phytium Technology Co., Ltd., Tianjin 300459, China*)

Abstract: To address the issue of deploying convolutional neural networks on resource-constrained devices, a high-performance fast convolution algorithm（FastInfer）was proposed for the FT-2000/4 multi-core processor. The algorithm optimized general matrix multiplication using a block-based strategy, storing frequently accessed data closer to the processor's cache to improve memory access efficiency during computation. In addition, a high-performance matrix multiplication microkernel was designed and implemented, utilizing vector outer product operations to update data and enhance the

收稿日期: 2024−10−16
基金项目: 国家自然科学基金资助项目（61402321）；天津市自然科学基金资助项目（23JCYBJC01770）；2024 年第一批天津市制造业高质量发展专项资金资助项目（24ZGNGX00020）
第一作者: 赵亚飞（2001−），男，硕士研究生．研究方向：计算机体系结构．E-mail：zhaoyf@tju.edu.cn
通信作者: 魏继增（1981−），男，副教授．研究方向：计算机体系结构．E-mail：weijizeng@tju.edu.cn

引文格式: 赵亚飞，杨耀功，王永刚，等．面向飞腾处理器平台的快速卷积算法优化[J]．上海理工大学学报，2024，46(6)：610-619.
Citation: ZHAO Yafei, YANG Yaogong, WANG Yonggang, et al. Fast convolution algorithm optimization for Phytium processor[J]. Journal of University of Shanghai for Science and Technology, 2024, 46(6)：610-619.

memory-to-computation ratio. This design maximized the masking of memory instruction latency. Experimental results demonstrated that FastInfer achieved a peak computational performance of 99.56 GFLOPS on the FT-2000/4 processor. In tests with general matrix multiplication at various input scales, FastInfer outperformed OpenBLAS by 1.07 and 1.52 times. In convolution tests, FastInfer performed 1.32 times better than the ARM Compute Library, achieving high-performance convolution computation on the FT-2000/4 multi-core processor.

Keywords: *deep learning*; *fast convolution algorithm*; *parallel computing*; *general matrix multiplication*

卷积运算是典型的计算密集型和访存密集型任务。在卷积神经网络的推理过程中，80% 的时间用于卷积层的计算，因此，卷积层的优化对于提升整个卷积神经网络的效率和性能显得尤为重要。并且，随着卷积神经网络的应用场景变得更为复杂，模型的层数进一步加深，这对部署设备的计算能力提出了更高的要求，限制了卷积神经网络在计算资源受限的设备（如 CPU）上的部署[1]。

目前优化卷积运算较为主流的算法有 Im2col 和 Winograd[2]。Im2col 算法将输入图像和卷积核转换为矩阵，从而将难以优化的卷积操作转换为当今具有良好优化实现能力的矩阵乘法计算[2]。Winograd 最小滤波算法通过减少卷积操作的乘法次数来提升计算效率，但仅适用于卷积核较小的情况，通用性较差[3]。

Im2col 算法实现起来简单灵活，并且能够支持任意大小的卷积核，受到了广泛的关注和研究。Anderson 等[4] 对 Im2col 算法的内存占用和并行性进行优化，在 LeNet 模型和 CIFAR-10 数据集上分别达到了 1.93 倍和 1.61 倍的加速效果。Dukhan 等[5] 提出了一种称为间接卷积算法的 Im2col 快速卷积改进算法，通过引入间接缓冲区来代替 Im2col 缓冲区，避免了高昂的内存复制成本，相较于基于通用矩阵乘法的传统方法，算法性能提升 1.03～1.23 倍。吴焕等[6] 提出一种针对卷积访存连续性的优化策略，性能比 Intel 的 MKL（math kernel library）算法提升 40%。Alvarenga 等[7] 研究了多种卷积算法的性能比较，评估了来自 1097 个真实深度学习模型的 9243 个卷积操作，结果表明，在快速卷积算法中，Im2col 结合通用矩阵乘法能够在实际应用中表现出优良的性能。Zhang 等[8] 提出将 Im2col 与 Winograd 算法结合的快速卷积方法，将多维卷积分解为一维卷积，降低算法的空间复杂度和数据访问不连续性，性能相比 cuDNN 算法中最快的基准算法提高了 0.788～2.05 倍。

FT-2000/4 处理器是我国飞腾公司研发的一款面向桌面应用的高性能通用处理器。在深度学习领域，目前 FT-2000/4 处理器相关生态较为薄弱，缺少深度学习库对其进行专门的优化适配。

为实现卷积神经网络在飞腾处理器上的高性能部署，面向 FT-2000/4 处理器对卷积神经网络的性能瓶颈——卷积运算进行加速。针对 FT-2000/4 处理器的浮点计算能力和缓存参数配置，提出一套合理的通用矩阵乘法分块方案，将处理器频繁访问的数据存入更靠近处理器的缓存中，最大程度掩盖处理器访存的延迟。为配合分块方案进一步提高访存效率，设计实现高性能的矩阵乘法内核函数，使用向量外积更新数据以提高计算访存比，并且使用 ARM NEON 向量指令和手动指令重排等提高程序的并行性。最终实验与多个开源的高性能线性代数库和 ARM 官方开源的计算库进行对比，证明了算法在 FT-2000/4 处理器上的有效性。

1　相关背景

1.1　Im2col 算法

Im2col 卷积优化算法最早出现在深度学习框架 Caffe 中，它将难以优化的直接卷积运算转换为当前具有良好优化实现的通用矩阵乘法运算[9-12]。目前，Im2col 卷积优化算法在多个流行的深度学习框架中使用，是卷积最重要的优化算法之一。

Im2col 算法将多维的卷积核和输入图像张量转换为二维矩阵，然后将变换后的矩阵相乘得出卷积运算结果。算法的主要流程有卷积核变换、输入图像变换和变换后矩阵相乘，其中卷积核变

换可以在算子初始化时进行，只需计算一次[13]。

Im2col 算法主要流程如下：

a. 卷积核变换。

卷积层中卷积核的维度为$(C_{\text{out}},C_{\text{in}},K_{\text{h}},K_{\text{w}})$，其中：$C_{\text{out}}$是卷积层输出通道数；$C_{\text{in}}$是卷积层输入通道数；$K_{\text{h}}$是卷积核高度；$K_{\text{w}}$是卷积核宽度。卷积核变换较为简单，只需要将二维单个卷积核拉伸成长度为$K_{\text{h}}\times K_{\text{w}}$的向量，然后将同一输出通道的卷积核按顺序存入矩阵的同一行，变换过程如图 1 所示。卷积核变换后的输出矩阵大小为$(C_{\text{out}},C_{\text{in}}\times K_{\text{h}}\times K_{\text{w}})$。

b. 输入图像变换。

对单通道输入图像进行变换的过程和卷积计算过程相似，卷积核在输入图像上滑动取值，将卷积核选中的数据复制填入变换后的矩阵中，变换过程如图 2 所示。变换后的矩阵大小为$(K_{\text{h}}\times K_{\text{w}},$

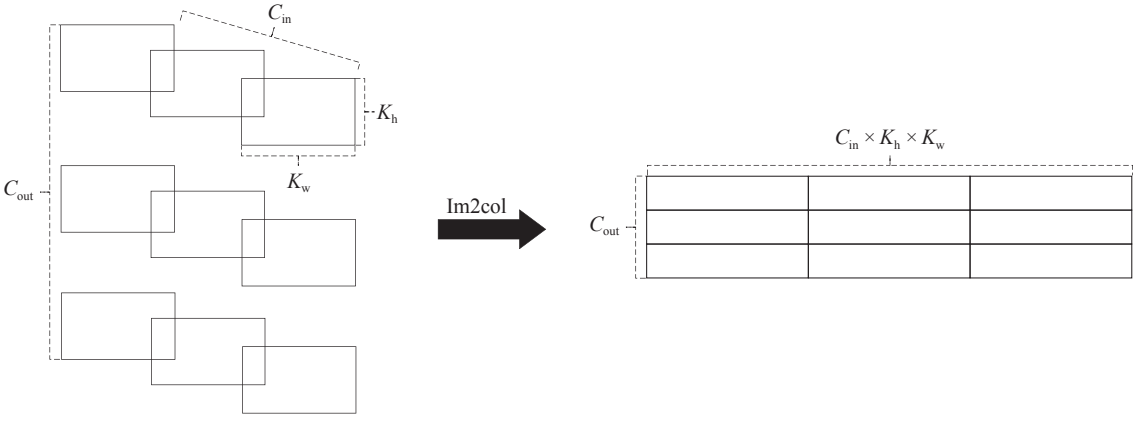
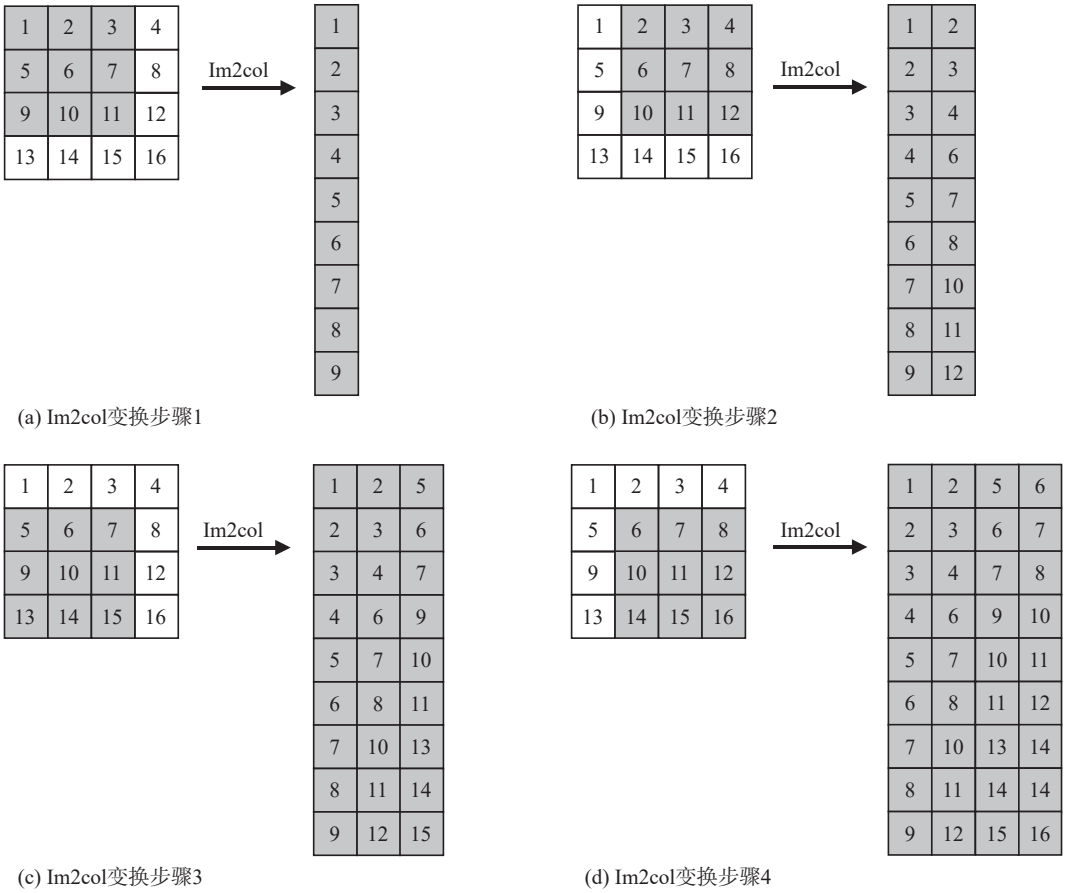


图 1 Im2col 卷积核变换示意图

Fig.1 Im2col convolutional kernel transformation



(a) Im2col变换步骤1 (b) Im2col变换步骤2

(c) Im2col变换步骤3 (d) Im2col变换步骤4

图 2 单通道输入图像 Im2col 变换过程示意图

Fig.2 Schematic diagram of Im2col transformation process for single channel input image

$H_{\text{out}} \times W_{\text{out}}$)，其中：矩阵的高度为卷积核大小 $K_{\text{h}} \times K_{\text{w}}$；宽度为输出图像大小 $H_{\text{out}} \times W_{\text{out}}$；$H_{\text{out}}$，$W_{\text{out}}$ 分别为输出图像的高度和宽度。对于多通道输入图像，只需将每个通道的变换矩阵排成一列即可。多通道输入图像的 Im2col 变换矩阵的维度为 $(C_{\text{in}} \times K_{\text{h}} \times K_{\text{w}}, H_{\text{out}} \times W_{\text{out}})$。

c. 变换后矩阵相乘。

对卷积核 $\boldsymbol{X}$ 和输入图像 $\boldsymbol{Y}$ 进行 Im2col 变换后得到卷积核变换矩阵 $\boldsymbol{X}^{\text{T}}$ 和输入图像变换矩阵 $\boldsymbol{Y}^{\text{T}}$，其中：$\boldsymbol{X}^{\text{T}}$ 的维度为 $(C_{\text{out}}, C_{\text{in}} \times K_{\text{h}} \times K_{\text{w}})$；$\boldsymbol{Y}^{\text{T}}$ 的维度为 $(C_{\text{in}} \times K_{\text{h}} \times K_{\text{w}}, H_{\text{out}} \times W_{\text{out}})$。令 $\boldsymbol{Z}^{\text{T}} = \boldsymbol{X}^{\text{T}} \times \boldsymbol{Y}^{\text{T}}$，可得到变换后的输出矩阵 $\boldsymbol{Z}^{\text{T}}$，维度为 $(C_{\text{out}}, H_{\text{out}} \times W_{\text{out}})$，再将其维度调整为 $(C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$，便得到最终的输出图像 $\boldsymbol{Z}$。

1.2　ARM 架构下常用加速技术

1.2.1　FT-2000/4 处理器概述

FT-2000/4 是一款面向桌面应用的通用处理器，内部集成 4 个飞腾自主研发的处理器核心 FTC663，兼容 64 位 ARMv8 指令集，16 nm 制程，主频最高 3.0 GHz，最大功耗 10 W。

FTC663 处理器核采用乱序三发射超标量体系结构，支持动态分支预测和全局历史缓存区，核内集成 L1 缓存。FT-2000/4 处理器片上存储分为 3 级缓存结构。其中：L1 缓存为处理器核私有，每个处理器核分别有 32 KB 的 L1 指令缓存和 L1 数据缓存；二级缓存为处理器簇内 2 个处理器核共享，大小为 2 MB，全片总计 4 MB；三级缓存为片上所有处理器核共享，全片总计 4 MB。

1.2.2　ARM NEON 向量扩展

SIMD（single instruction multiple data）中的一条指令可以同时处理多个数据。如图 3 所示，向量运算部件对向量 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 执行乘法操作，将结果存入向量 $\boldsymbol{C}$ 中，相当于同时对 4 个标量执行乘法运算。因此，SIMD 能够有效提高处理数据的吞吐量。

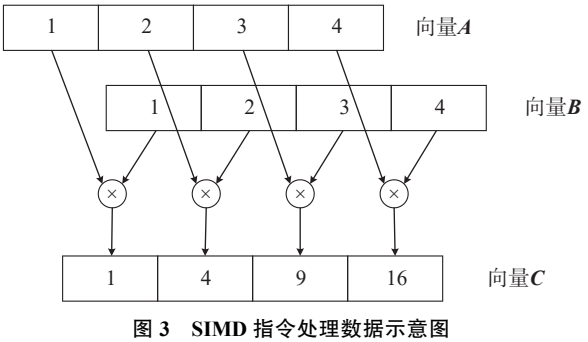


图 3　SIMD 指令处理数据示意图

Fig.3　Schematic diagram of SIMD instruction processing data

ARM NEON 是 ARM 架构中针对大规模并行运算设计的 128 位高级 SIMD 扩展，主要用于加速多媒体处理、信号处理和其他计算密集型应用。如今 NEON 技术又满足了深度学习应用对于大规模数据计算的需求，因此被广泛用于深度学习加速[14-15]。

为了支持 NEON 指令集，ARM 架构添加了向量寄存器和硬件执行单元，这些寄存器和执行单元与传统的标量寄存器和执行单元相互独立，专门用于执行 NEON 向量指令。在 AArch64 模式下，每个计算核心拥有 32 个 128 位的向量寄存器，编号从 V0 至 V31。ARMv8 的向量寄存器可看作是 32 个 128 位的寄存器，Q0 至 Q31；32 个 64 位的寄存器，D0 至 D31；32 个 32 位的寄存器，S0 至 S31；32 个 16 位的寄存器，H0 至 H31；32 个 8 位的寄存器，B0 至 B31，或者是以上的组合去使用。

2　快速卷积算法（FastInfer）

2.1　高性能通用矩阵乘法算法设计

本文使用矩阵乘法分块策略并结合 FT-2000/4 多核处理器硬件特性对通用矩阵乘法进行深度优化，该算法的核心思想在于优化计算过程的内存访问，将处理器频繁访问的数据存入更靠近处理器的缓存中，同时提高计算访存比以掩盖访存指令的延迟。

算法基本流程是先对矩阵进行分块[16-17]，确保分块后的子矩阵能够全部存入各级缓存中，如图 4 所示。在分块的同时将子矩阵打包，优化数据的存储组织方式，使数据存储得紧凑，提高后续计算时的访存效率。打包后的子矩阵会被继续划分，然后送入高性能微内核中计算。

算法的具体流程包含如下内容：

a. 简单分块。

第一层循环将输入矩阵 $\boldsymbol{B}$ 和输出矩阵 $\boldsymbol{C}$ 分别切分成 (K, n_{c}) 和 (M, n_{c}) 大小的子矩阵；第二层循环将输入矩阵 $\boldsymbol{A}$ 切分成 (M, k_{c}) 大小的子矩阵，将 $\boldsymbol{B}$ 进一步切分成 $(k_{\text{c}}, n_{\text{c}})$ 大小的子矩阵 $\boldsymbol{B}_{\text{c}}$；第三层循环将 $\boldsymbol{A}$ 进一步切分成 $(m_{\text{c}}, k_{\text{c}})$ 子矩阵 $\boldsymbol{A}_{\text{c}}$，将 $\boldsymbol{C}$ 切分成 $(m_{\text{c}}, n_{\text{c}})$ 大小的子矩阵 $\boldsymbol{C}_{\text{c}}$。$\boldsymbol{A}_{\text{c}}$ 将会被存入 L3 缓存中，而 $\boldsymbol{C}_{\text{c}}$ 和 $\boldsymbol{B}_{\text{c}}$ 将会被存入 L2 缓存中。矩阵分块的参数设置决定着矩阵乘法最终实现的性能，通过

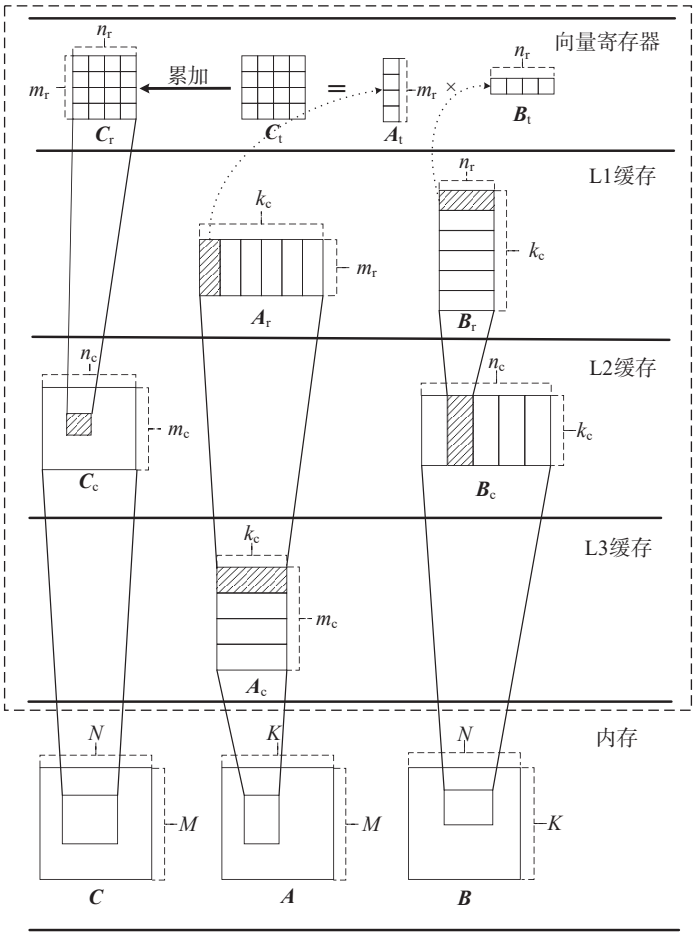


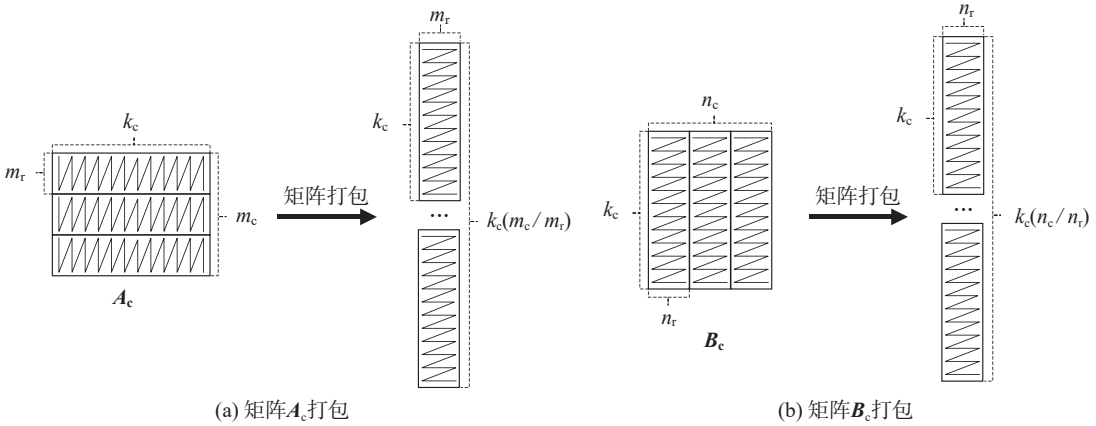
图 4 FastInfer 执行过程中矩阵存储位置变化示意图

Fig.4 Schematic diagram of matrix storage location changes during FastInfer execution

合理设置分块参数m_c，k_c和n_c的大小，能够最大程度掩盖访存指令的延迟。

b. 数据打包。

对矩阵进行打包是因为分块后的矩阵能够全部存入对应的缓存中，但是数据的访问并不连续，需要对数据重新组织排列以适配微内核中的数据访问模式，从而提高缓存的命中率[18]。从图 5 中$\boldsymbol{A}_c$和$\boldsymbol{B}_c$的元素访问顺序可知，$\boldsymbol{A}_c$中的元素访问按列进行，相邻访问的元素跨行存储，这种零散的访存行为体现不出任何的空间局部性，必然会带来大量的缓存缺失。而$\boldsymbol{B}_c$中的元素访问虽然按行进行，但在访问相邻行的元素时跨度较大，缓



(a) 矩阵$\boldsymbol{A}_c$打包　　(b) 矩阵$\boldsymbol{B}_c$打包

图 5 通用矩阵乘法数据打包存储示意图

Fig.5 Schematic diagram of GEMM data packaging and storage

存访问情况同样不够理想。因此，必须对数据进行打包以提高数据访问的空间局部性，进而提高缓存的命中率。

对数据进行打包的方式需要根据微内核计算的方式进行设计，如图 5 所示。对于$\boldsymbol{A}_{\mathrm{c}}$，将其每一列元素重新排列放至同一行，相邻列的数据位于相邻行；对于$\boldsymbol{B}_{\mathrm{c}}$，将其每一个分块的数据单独提取，按照顺序摆放。

c. 内核划分。

打包后的矩阵进行切分。在第四层循环中按照行方向将矩阵$\boldsymbol{A}_{\mathrm{c}}$切分成$(m_{\mathrm{r}},k_{\mathrm{c}})$大小的$\boldsymbol{A}_{\mathrm{r}}$，将$\boldsymbol{C}_{\mathrm{c}}$切分成$(m_{\mathrm{r}},n_{\mathrm{c}})$大小的子矩阵。在第五层循环中按照列方向将矩阵$\boldsymbol{B}_{\mathrm{c}}$切分成$(k_{\mathrm{c}},n_{\mathrm{r}})$的$\boldsymbol{B}_{\mathrm{r}}$，将$\boldsymbol{C}_{\mathrm{c}}$继续切分成$(m_{\mathrm{r}},n_{\mathrm{r}})$大小的$\boldsymbol{C}_{\mathrm{r}}$。

d. 微内核计算。

为对分块后的子矩阵进行高效计算，本文设计并实现了一个高性能的矩阵乘法微内核函数，在实现时使用汇编代码编写，利用 ARM NEON 向量指令和手动指令重排等技术提高程序的指令和数据并行性。

由于在微内核中完成全部的计算任务，因此，必须将微内核的性能提升为高性能。$\boldsymbol{A}_{\mathrm{r}}$和$\boldsymbol{B}_{\mathrm{r}}$是计算过程中访问频率最高的矩阵，将其存入 L1 缓存中，而$\boldsymbol{C}_{\mathrm{r}}$足够小，能够被全部存入向量寄存器中。微内核采用外积公式更新$\boldsymbol{C}_{\mathrm{r}}$，每次从$\boldsymbol{A}_{\mathrm{r}}$取长度为$m_{\mathrm{r}}$的列向量$\boldsymbol{A}_{\mathrm{t}}$，从$\boldsymbol{B}_{\mathrm{r}}$取长度为$n_{\mathrm{r}}$的行向量$\boldsymbol{B}_{\mathrm{t}}$，将列向量$\boldsymbol{A}_{\mathrm{t}}$和行向量$\boldsymbol{B}_{\mathrm{t}}$进行外积运算得到维度为$(m_{\mathrm{r}},n_{\mathrm{r}})$的矩阵$\boldsymbol{C}_{\mathrm{t}}$，如式 (1) 所示。接下来将$\boldsymbol{C}_{\mathrm{t}}$累加到$\boldsymbol{C}_{\mathrm{r}}$所存储的寄存器中。$\boldsymbol{A}_{\mathrm{r}}$共有$k_{\mathrm{c}}$列，因此需要重复上述操作$k_{\mathrm{c}}$次。

$$\boldsymbol{C}_{\mathrm{t}}=\boldsymbol{A}_{\mathrm{t}}\otimes\boldsymbol{B}_{\mathrm{t}}=\begin{bmatrix}A_{\mathrm{t}}^{1}\\A_{\mathrm{t}}^{2}\\\vdots\\A_{\mathrm{t}}^{m_{\mathrm{r}}}\end{bmatrix}\otimes\begin{bmatrix}B_{\mathrm{t}}{}^{1} & B_{\mathrm{t}}{}^{2} & \cdots & B_{\mathrm{t}}{}^{n_{\mathrm{r}}}\end{bmatrix}=\begin{bmatrix}A_{\mathrm{t}}^{1}B_{\mathrm{t}}{}^{1} & A_{\mathrm{t}}^{1}B_{\mathrm{t}}{}^{2} & \cdots & A_{\mathrm{t}}^{1}B_{\mathrm{t}}{}^{n_{\mathrm{r}}}\\A_{\mathrm{t}}^{2}B_{\mathrm{t}}{}^{1} & A_{\mathrm{t}}^{2}B_{\mathrm{t}}{}^{2} & \cdots & A_{\mathrm{t}}^{2}B_{\mathrm{t}}{}^{n_{\mathrm{r}}}\\\vdots & \vdots & & \vdots\\A_{\mathrm{t}}^{m_{\mathrm{r}}}B_{\mathrm{t}}{}^{1} & A_{\mathrm{t}}^{m_{\mathrm{r}}}B_{\mathrm{t}}{}^{2} & \cdots & A_{\mathrm{t}}^{m_{\mathrm{r}}}B_{\mathrm{t}}{}^{n_{\mathrm{r}}}\end{bmatrix}\tag{1}$$

在本文的实现中，微内核函数的核心计算部分采用内联汇编代码编写，并且使用 NEON 向量指令和 4×1 循环展开对其进行优化。在开始计算之前，首先使用 ldr 向量加载指令将$(m_{\mathrm{r}},n_{\mathrm{r}})$大小的矩阵$\boldsymbol{C}_{\mathrm{r}}$加载入 q0 至 q3 的向量寄存器，然后进入$k_{\mathrm{c}}$次的循环。由于对循环进行了 4×1 循环展开，所以循环内部每次会加载 4 列$\boldsymbol{A}_{\mathrm{r}}$中的元素和 4 行$\boldsymbol{B}_{\mathrm{r}}$中的元素，分别保存至 q4 至 q7 和 q8 至 q11 的寄存器中。数据加载完毕后，使用 fmal 向量–标量累加乘法指令更新$\boldsymbol{C}_{\mathrm{r}}$。

为更好地掩盖访存指令的延迟，提高指令的并行性，在实现算法时，使用了指令重排的优化技术，手动将 ldr 指令和 fmal 指令交错排列，在加载数据的同时进行乘加运算，实现算数指令掩盖部分加载指令的延迟。循环结束后使用 str 向量存写指令将 q0 至 q3 向量寄存器的值写回$\boldsymbol{C}_{\mathrm{r}}$中。微内核中每次循环分别加载 16 个$\boldsymbol{A}_{\mathrm{r}}$和$\boldsymbol{B}_{\mathrm{r}}$中的共 32 个单精度浮点数，执行 128 次浮点运算，计算访存比为 4，这是算法取得高性能的重要因素。访存指令的延迟要远大于算数运算指令，更高的计算访存比意味着可以在访存指令等待操作数返回的同时发射大量算数指令到流水线中，从而掩盖访存指令的延迟。

e. 边界处理。

在本文的实现中，参数m_{r}和n_{r}被设置为 4，因此在微内核函数中最小的计算单元为(4,4)大小的子矩阵，需要保证输入矩阵的大小M和N必须是 4 的倍数，否则矩阵会留有一些边界无法计算。而在实际应用中无法保证每次输入矩阵大小都满足条件，因此，必须对边界情况进行处理。本文的做法是对输入矩阵尺寸不足 4 的部分进行补 0，计算完毕后再根据原尺寸进行裁剪。

对算法进行分析总结可知，算法取得高性能的关键点在于微内核函数中使用外积公式来更新数据，从而获得了更高的计算访存比。另外，合理的分块策略充分发挥了处理器多级缓存的优势，配合上数据打包策略使得处理器访问数据变得更加连续，有效掩盖了访存指令的延迟。

2.2　通用矩阵乘法参数选择

m_{r}和n_{r}参数的选择主要受到 NEON 向量寄存器大小和数量的限制。本文中通用矩阵乘法实现的是单精度浮点数矩阵乘法，单精度浮点数的宽度为 32 位，而 NEON 向量寄存器宽度为 128 位，能够容纳 4 个单精度浮点数。因此，m_{r}和n_{r}的大小应该考虑 4 的倍数。

在微内核函数中对大小为$(m_{\mathrm{r}},n_{\mathrm{r}})$的$\boldsymbol{C}_{\mathrm{r}}$进行求解，需要使用$(m_{\mathrm{r}}n_{\mathrm{r}})/4$个向量寄存器存储$\boldsymbol{C}_{\mathrm{r}}$。另

外，微内核计算部分采用4×1循环展开优化，每次取4列$\boldsymbol{A}_{\mathrm{r}}$和4行$\boldsymbol{B}_{\mathrm{r}}$中的元素计算，共需要$m_{\mathrm{r}}$个向量寄存器存储$\boldsymbol{A}_{\mathrm{t}}$，$n_{\mathrm{r}}$个向量寄存器存储$\boldsymbol{B}_{\mathrm{t}}$。$m_{\mathrm{r}}$和$n_{\mathrm{r}}$的选择应满足式(2)的条件，此时$m_{\mathrm{r}}$和$n_{\mathrm{r}}$可供选择的值只有4和8。当$m_{\mathrm{r}}=n_{\mathrm{r}}=8$时，计算正好用满全部32个向量寄存器，而在实际中会保留一部分寄存器留给编译器分配，因此，最终将m_{r}和n_{r}设置为4。

$$\frac{m_{\mathrm{r}}n_{\mathrm{r}}}{4}+m_{\mathrm{r}}+n_{\mathrm{r}}\leqslant 32 \tag{2}$$

接下来考虑如何设置m_{c}，n_{c}和k_{c}的大小。FT-2000/4处理器的L1缓存为每个核私有，大小为32 KB。为将$\boldsymbol{A}_{\mathrm{r}}$和$\boldsymbol{B}_{\mathrm{r}}$全部存入L1缓存中，则需要满足

$$4(k_{\mathrm{c}}m_{\mathrm{r}}+k_{\mathrm{c}}n_{\mathrm{r}})\leqslant 32\times 1\,024 \tag{3}$$

算法选择在N的维度上并行，这就意味着各个线程需要在缓存中保留一份$\boldsymbol{B}_{\mathrm{c}}$和$\boldsymbol{C}_{\mathrm{c}}$进行并行处理，如图6所示。FT-2000/4是4核8线程处理器，因此将OpenMP的并行线程数μ设置为8，开启8个线程并行计算。$\boldsymbol{B}_{\mathrm{c}}$和$\boldsymbol{C}_{\mathrm{c}}$均保存在L2缓存中，而L2缓存大小为2 MB，为每两个处理器核共享，片上总容量为4 MB，则需要满足

$$4(k_{\mathrm{c}}n_{\mathrm{c}}+m_{\mathrm{c}}n_{\mathrm{c}})\leqslant\frac{4\times 1\,024\times 1\,024}{\mu} \tag{4}$$

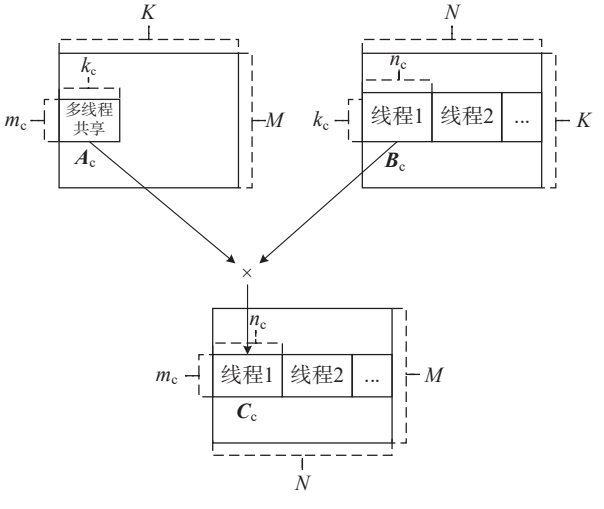


图6 通用矩阵乘法分块多线程加速示意图

Fig.6 Schematic diagram of GEMM block multi-thread acceleration

$\boldsymbol{A}_{\mathrm{c}}$不存在$N$的维度，只需在L3缓存中保存一份，被所有线程共享。L3缓存大小为4 MB，为片上所有处理器核共享，则需要满足：

$$4k_{\mathrm{c}}m_{\mathrm{c}}\leqslant 4\times 1\,024\times 1\,024 \tag{5}$$

接下来从最大化计算访存比的角度确定m_{c}、k_{c}和n_{c}的取值。将$\boldsymbol{A}_{\mathrm{r}}$和$\boldsymbol{B}_{\mathrm{r}}$加载入缓存、将$\boldsymbol{C}_{\mathrm{r}}$从缓存加载入寄存器和写回缓存所需要的总的访存次数为$k_{\mathrm{c}}m_{\mathrm{r}}+k_{\mathrm{c}}n_{\mathrm{r}}+2m_{\mathrm{r}}n_{\mathrm{r}}$，总的浮点运算次数为$2m_{\mathrm{r}}n_{\mathrm{r}}k_{\mathrm{c}}$，则计算访存比$\theta_{\mathrm{r}}$为

$$\theta_{\mathrm{r}}=\frac{2m_{\mathrm{r}}n_{\mathrm{r}}k_{\mathrm{c}}}{k_{\mathrm{c}}(m_{\mathrm{r}}+n_{\mathrm{r}})+2m_{\mathrm{r}}n_{\mathrm{r}}} \tag{6}$$

代入$m_r=n_r=8$，则有

$$\theta_{\mathrm{r}}=\frac{32k_{\mathrm{c}}}{8k_{\mathrm{c}}+32} \tag{7}$$

显然k_c越大，计算访存比越大，基于式(3)和式(7)取得$k_{\mathrm{c}}=1\,024$。

同理，计算$\boldsymbol{A}_{\mathrm{c}}$，$\boldsymbol{B}_{\mathrm{c}}$，$\boldsymbol{C}_{\mathrm{c}}$所需总的访存次数为$k_{\mathrm{c}}m_{\mathrm{c}}+k_{\mathrm{c}}n_{\mathrm{c}}+2m_{\mathrm{c}}n_{\mathrm{c}}$，总的浮点乘法和加法运算次数为$2m_{\mathrm{c}}k_{\mathrm{c}}n_{\mathrm{c}}$，则计算访存比$\theta_{\mathrm{c}}$为

$$\theta_{\mathrm{c}}=\frac{2m_{\mathrm{c}}k_{\mathrm{c}}n_{\mathrm{c}}}{k_{\mathrm{c}}(m_{\mathrm{c}}+n_{\mathrm{c}})+2m_{\mathrm{c}}n_{\mathrm{c}}} \tag{8}$$

当$n_{\mathrm{c}}\ll k_{\mathrm{c}}$时，有

$$\theta_{\mathrm{c}}=\frac{2m_{\mathrm{c}}n_{\mathrm{c}}}{m_{\mathrm{c}}+n_{\mathrm{c}}} \tag{9}$$

因此，当$m_{\mathrm{c}}=n_{\mathrm{c}}$时，计算访存比取得最大值，且$m_{\mathrm{c}}n_{\mathrm{c}}$越大，计算访存比越大。基于式(4)和式(9)取得$m_{\mathrm{c}}=n_{\mathrm{c}}=32$。

2.3 算法步骤

算法实现为六重循环，其中前五重循环用于对矩阵进行划分，最后一重循环安排在微内核中，用于矩阵乘法的累加运算。算法具体步骤如下：

步骤1 将输入矩阵$\boldsymbol{B}$和输出矩阵$\boldsymbol{C}$分别切分成(K,n_{c})和(M,n_{c})大小的子矩阵。

步骤2 将输入矩阵$\boldsymbol{A}$切分成(M,k_{c})大小的子矩阵，将$\boldsymbol{B}$进一步切分成$(k_{\mathrm{c}},n_{\mathrm{c}})$大小的子矩阵$\boldsymbol{B}_{\mathrm{c}}$。

步骤3 对$\boldsymbol{B}_{\mathrm{c}}$进行打包。

步骤4 将$\boldsymbol{A}$进一步切分成$(m_{\mathrm{c}},k_{\mathrm{c}})$子矩阵$\boldsymbol{A}_{\mathrm{c}}$，将$\boldsymbol{C}$切分成$(m_{\mathrm{c}},n_{\mathrm{c}})$大小的子矩阵$\boldsymbol{C}_{\mathrm{c}}$。

步骤5 对$\boldsymbol{A}_{\mathrm{c}}$进行打包。

步骤6 按照行方向将$\boldsymbol{A}_{\mathrm{c}}$切分成$(m_{\mathrm{r}},k_{\mathrm{c}})$大小的$\boldsymbol{A}_{\mathrm{r}}$，同时将$\boldsymbol{C}_{\mathrm{c}}$切分成$(m_{\mathrm{r}},n_{\mathrm{c}})$大小的子矩阵。

步骤7 按照列方向将$\boldsymbol{B}_{\mathrm{c}}$切分成$(k_{\mathrm{c}},n_{\mathrm{r}})$的$\boldsymbol{B}_{\mathrm{r}}$，将$\boldsymbol{C}_{\mathrm{c}}$继续切分成$(m_{\mathrm{r}},n_{\mathrm{r}})$大小的$\boldsymbol{C}_{\mathrm{r}}$。

步骤8 将$\boldsymbol{A}_{\mathrm{r}}$加载入寄存器$\boldsymbol{A}_{\mathrm{t}}$，令$\boldsymbol{A}_{\mathrm{t}}=\boldsymbol{A}_{\mathrm{r}}^{i}$$(i=0,1,\cdots,k_{\mathrm{c}})$。

步骤9 将$\boldsymbol{B}_{\mathrm{r}}$加载入寄存器$\boldsymbol{B}_{\mathrm{t}}$，令$\boldsymbol{B}_{\mathrm{t}}=\boldsymbol{B}_{\mathrm{r}}^{i}$$(i=0,1,\cdots,k_{\mathrm{c}})$。

步骤 10　对 $\boldsymbol{A}_t$ 和 $\boldsymbol{B}_t$ 进行向量外积运算，令 $\boldsymbol{C}_t = \boldsymbol{A}_t \otimes \boldsymbol{B}_t$。

步骤 11　将 $\boldsymbol{C}_t$ 累加到 $\boldsymbol{C}_r$。

步骤 12　重复执行上述操作，直到所有分块计算完毕。

3　实验与结果分析

3.1　实验环境和方法

本文进行实验的硬件环境资源如表 1 所示。

表 1　实验平台硬件资源参数配置

Tab.1　Hardware parameter configuration of Phytium development board

参数	值
处理器	FT-2000/4
处理器架构	ARMv8-a
处理器频率	2.6 GHz
处理器核心数	4核8线程
L1指令高速缓存	32 KB
L1数据高速缓存	32 KB
L2高速缓存	2 MB
L3高速缓存	4 MB
内存	8 GB DDR4
硬盘	64 GB mSATA硬盘

项目开发在本地 Windows 主机上进行，通过集成开发环境远程连接远程开发板，编译运行时使用开发板上的工具链和依赖库。实验分成 2 个部分对 FastInfer 的整体性能进行验证。

a. 通用矩阵乘法性能测试。

通用矩阵乘法测试的输入矩阵尺寸为 $M = N = K$ 的方阵，按照输入尺寸分为大尺寸和小尺寸通用矩阵乘法测试。大尺寸测试中矩阵大小取值从 100 到 1 000，步长为 100；小尺寸测试中矩阵大小取值从 10 到 100，步长为 10。通用矩阵乘法测试在进行性能测试的同时完成正确性检验。对于每种输入尺寸的测试，在测试前首先随机生成矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$，然后调用未优化的通用矩阵乘法，生成正确的参考矩阵 $\boldsymbol{RC}$。测试过程中调用 FastInfer 中的总用矩阵乘法 10 次得到对比矩阵 $\boldsymbol{C}$，同时记录 10 次中通用矩阵乘法运行的最短时间作为运行时间 T。性能统计完毕后对 $\boldsymbol{RC}$ 和 $\boldsymbol{C}$ 进行逐元素比较，如果二者的精度误差小于 10^{-5}，则视优化后的通用矩阵乘法实现正确。另外，本文将 FastInfer 的优化效果与一些面向 ARM 架构优化的基本线性代数库进行对比，其中包括 Eigen[19]，BLIS[20]，OpenBLAS[21]。

b. 卷积性能测试。

由于卷积神经网络中 3×3 大小的卷积核最为常用，因此在卷积性能测试中将卷积核尺寸固定为 3×3。测试输入图像大小根据 ImageNet 数据集中标准图像大小 224×224 进行设置。在卷积神经网络中，卷积层后通常紧接池化层将输入图像的尺寸减半，因此，测试输入图像尺寸设置为 224×224，112×112，56×56 和 28×28。ARM Compute Library[22] 是 ARM 开源的高性能计算库，面向 ARM 处理器优化，尤其是为移动设备和嵌入式平台上的计算密集型任务而优化，提供了矩阵计算、卷积运算、FFT、图像处理等操作高效实现。本文在测试时选择将 ARM Compute Library 作为基准进行性能对比。

3.2　通用矩阵乘法性能分析

图 7 展示了不同规模通用矩阵乘法的性能，图 7(a)为小规模下的通用矩阵乘法性能对比，FastInfer 的峰值性能达到 54.97 GFLOPS；OpenBLAS 的峰值性能达到 53.54 GFLOPS；BLIS 的峰值性能为 38.68 GFLOPS；Eigen 的峰值性能达到 45.74 GFLOPS。图 7(b)为大规模输入下的通用矩阵乘法性能对比，其中，FastInfer 的峰值性能达到 99.56 GFLOPS，为处理器理论峰值计算能力的 82.97%；OpenBLAS 的峰值性能达到 99.84 GFLOPS；BLIS 的峰值性能达到 97.28 GFLOPS；Eigen 的峰值性能达到 75.46 GFLOPS。综合来看，FastInfer 在矩阵规模较大的通用矩阵乘法上的性能表现基本和 OpenBLAS 齐平，计算效率为 OpenBLAS 的 1.07 倍，但在小尺寸的输入上 FastInfer 性能大幅度优于其他 3 个线性代数库，计算效率是 OpenBLAS 的 1.52 倍。

观察通用矩阵乘法的性能变化趋势，可以得出不同版本的通用矩阵乘法性能随着输入矩阵尺寸增大而提高，当输入规模达到一定阈值，通用矩阵乘法的性能趋于稳定。这是因为在小矩阵输入的情况下，输入矩阵可以完全存储在处理器的缓存中，缓存缺失率较低，内存访问的开销得以显著降低，通用矩阵乘法的计算效率高。但随着输入矩阵规模增大，缓存中只能容纳部分矩阵，

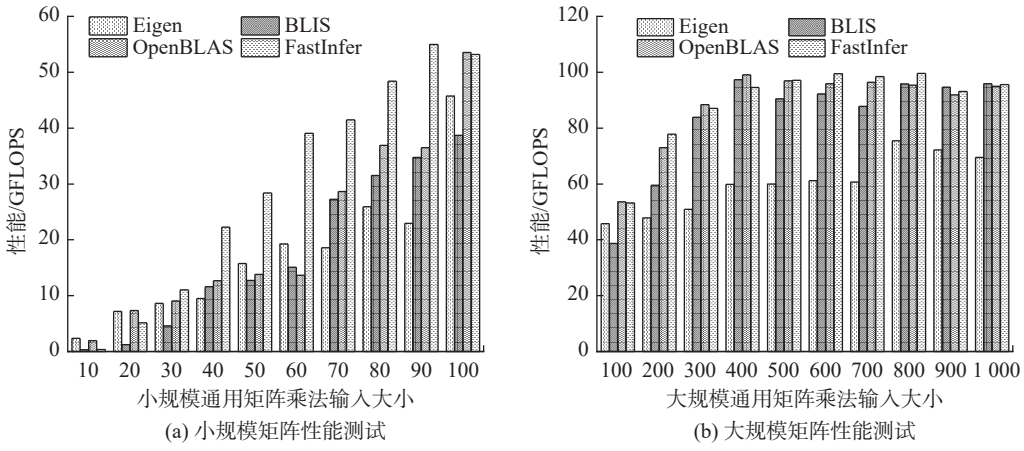


图 7 不同规模的通用矩阵乘法性能对比

Fig.7 Performance comparison of GEMM for different scales

导致访存出现大量缓存缺失，这种缓存缺失会迫使处理器频繁地从主存中加载数据，降低计算效率。而 FastInfer 随着输入矩阵尺寸增大，通用矩阵乘法的性能提高直至稳定。这是因为合理的分块大小确保了被分割的小矩阵块可以驻留在多级缓存中，减少了内存访问延迟。这使得即使在处理非常大的矩阵时，通用矩阵乘法的性能依然能够保持稳定。在处理小矩阵时，尽管理论上应该达到更高的性能，但由于分块后的块数较少，多核处理器的全部核心无法充分利用。特别是在使用 OpenMP 工具进行多线程加速时，如果线程数大于实际计算任务的并行度，会导致一些处理器核心闲置，无法完全发挥处理器的计算能力。

3.3 卷积性能分析

图 8 展示了 FastInfer 在不同输入尺寸下相对于 ARM Compute Library 的计算性能加速比。加速比定义为同一卷积输入尺寸下 ARM Compute Library 卷积运行时间和 FastInfer 卷积运行时间的比值。在 FT-2000/4 处理器上，FastInfer 的最高加速比为 1.69，平均加速比为 1.32。在输入图像尺寸相同的情况下，卷积加速比随着输入通道数的增加而提高。这是因为 Im2col 快速卷积算法最主要的运行时间用于通用矩阵乘法计算，而通用矩阵乘法对输入矩阵的尺寸比较敏感，当面对一些小尺寸矩阵或者形状不规则（比较“高瘦”和“矮胖”）的矩阵时无法发挥峰值计算能力。而经过 Im2col 转换后的矩阵尺寸与卷积核大小以及输入图像的维度有关，Im2col 转换后的矩阵并不是标准的方阵，而是一些较长或较宽的矩阵。对于卷积核，其变换后的矩阵 $\boldsymbol{X}^{\mathrm{T}}$ 大小为 $(C_{\mathrm{out}}, C_{\mathrm{in}}\times K_{\mathrm{h}}\times K_{\mathrm{w}})$，矩阵形状较宽。对于输入图像，其变换后的矩阵 $\boldsymbol{Y}^{\mathrm{T}}$ 大小为 $(C_{\mathrm{in}}\times K_{\mathrm{h}}\times K_{\mathrm{w}}, H_{\mathrm{out}}\times W_{\mathrm{out}})$，矩阵形状较长。因此，当卷积运算输入或输出通道数较小时，矩阵乘法输入矩阵会过长或过宽，导致通用矩阵乘法无法发挥峰值性能，卷积加速比较低。

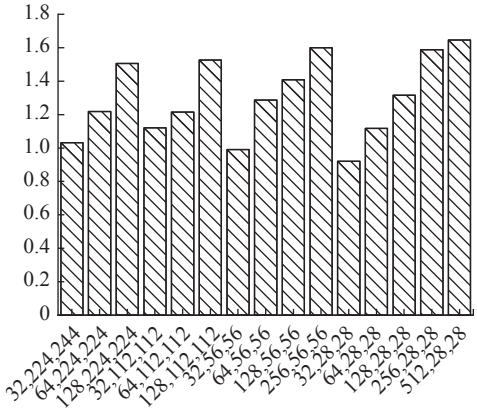


图 8 卷积性能加速效果

Fig.8 Acceleration effect of convolution operator

4 结 论

本文面向飞腾 FT-2000/4 多核处理器提出一种高性能的 Im2col 快速卷积算法 FastInfer，对卷积运算加速以提高卷积神经网络在飞腾平台上的运行效率。为将 Im2col 算法实现为高性能，FastInfer 对 Im2col 算法的核心部分——通用矩阵乘法进行深度优化。采用矩阵乘法分块策略，通过设置合理的分块参数，将处理器访问频率高的数据存放到更靠近处理器的缓存中，从而隐藏访存指令的延迟。同时，为了进一步提高计算过程中的访存

效率，设计实现高性能的矩阵乘法内核函数，利用 NEON 向量指令和手动指令重排等提高指令并行性和数据并行性，并使用向量外积公式更新数据以提高计算访存比。在 FT-2000/4 处理器上的实验表明，FastInfer 的峰值计算能力达到 99.56 GFLOPS，为处理器理论峰值浮点计算能力的 82.97%。与 ARM Compute Library 相比，FastInfer 的卷积计算平均加速比为 1.32，实现了卷积算法在飞腾处理器上的高性能。

参考文献：

[1] SZE V, CHEN Y H, YANG T J, et al. Efficient processing of deep neural networks: a tutorial and survey[J]. Proceedings of the IEEE, 2017, 105(12): 2295−2329.

[2] 李创, 刘宗林, 刘胜, 等. 快速卷积算法的综述研究 [J]. 计算机工程与科学, 2021, 43(10): 1711−1719.

[3] LAVIN A, GRAY S. Fast algorithms for convolutional neural networks[C]//2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). Las Vegas: IEEE, 2016: 4013–4021.

[4] ANDERSON A, VASUDEVAN A, KEANE C, et al. High-Performance low-memory lowering: GEMM-based algorithms for DNN convolution[C]//2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). Porto: IEEE, 2020: 99–106.

[5] DUKHAN M. The indirect convolution algorithm [EB/OL].[2019-07-03]. https://arxiv.org/abs/1907.02129.

[6] 吴焕, 吴俊敏. 基于 Caffe 加速卷积神经网络前向推理 [J]. 计算机工程与设计, 2018, 39(12): 3686−3691.

[7] ALVARENGA L, FERRARI V, SOUZA R, et al. ConvBench: a comprehensive benchmark for 2D convolution primitive evaluation[EB/OL].[2024-07-15]. https://arxiv.org/abs/2407.10730.

[8] ZHANG Z Y, ZHANG P F, XU Z P, et al. Im2col-winograd: an efficient and flexible fused-Winograd convolution for NHWC format on GPUs[C]//Proceedings of the 53rd International Conference on Parallel Processing. Gotland: Association for Computing Machinery, 2024: 1072–1081.

[9] RAMÍREZ C, CASTELLÓ A, MARTÍNEZ H, et al. Parallel GEMM-based convolution for deep learning on multicore RISC-V processors[J]. The Journal of Supercomputing, 2024, 80(9): 12623−12643.

[10] BARRACHINA S, DOLZ M F, SAN JUAN P, et al. Efficient and portable GEMM-based convolution operators for deep neural network training on multicore processors[J]. Journal of Parallel and Distributed Computing, 2022, 167: 240−254.

[11] TRUSOV A V, LIMONOVA E E, NIKOLAEV D P, et al. p-im2col: simple yet efficient convolution algorithm with flexibly controlled memory overhead[J]. IEEE Access, 2021, 9: 168162−168184.

[12] WANG X C, WANG Y X, LI J. An industrial-grade solution for convolutional neural network optimization and deployment[C]//2023 4th International Seminar on Artificial Intelligence, Networking and Information Technology (AINIT). Nanjing: IEEE, 2023: 46–50.

[13] YANG Z W, LU L, WANG R M. A batched GEMM optimization framework for deep learning[J]. The Journal of Supercomputing, 2022, 78(11): 13393−13408.

[14] MACIÁ-LILLO A, BARRACHINA S, FABREGAT G, et al. Optimizing convolutions for deep learning inference on ARM Cortex-M processors[J]. IEEE Internet of Things Journal, 2024, 11(15): 26203−26219.

[15] TRUSOV A, LIMONOVA E, NIKOLAEV D, et al. Fast matrix multiplication for binary and ternary CNNs on ARM CPU[C]//2022 26th International Conference on Pattern Recognition (ICPR). Montreal: IEEE, 2022: 3176–3182.

[16] WU S X, ZHAI Y J, HUANG J J, et al. FT-GEMM: a fault tolerant high performance GEMM implementation on x86 CPUs[C]//Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing. Orlando: ACM, 2023: 323–324.

[17] ALAEJOS G, CASTELLÓ A, MARTÍNEZ H, et al. Micro-kernels for portable and efficient matrix multiplication in deep learning[J]. The Journal of Supercomputing, 2023, 79(7): 8124−8147.

[18] TRUSOV A V, LIMONOVA E E, NIKOLAEV D P, et al. On fast computing of neural networks using central processing units[J]. Pattern Recognition and Image Analysis, 2023, 33(4): 756−768.

[19] Eigen v3[EB/OL]. [2024-10-20]. http://eigen.tuxfamily.org.

[20] VAN ZEE F G, VAN DE GEIJN R A. BLIS: a framework for rapidly instantiating BLAS functionality[J]. ACM Transactions on Mathematical Software (TOMS), 2015, 41(3): 14.

[21] OpenBLAS: An optimized BLAS library[EB/OL]. [2024-10-20]. http://www.openmathlib.org/OpenBLAS/.

[22] ARM compute library[EB/OL]. [2024-11-03]. https://www.arm.com/products/development-tools/embedded-and-software/compute-library.

（编辑：丁红艺）