计 算 机 研 究 与 发 展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202220787
61(4): 955−972, 2024

申威 26010 众核处理器上 Winograd 卷积算法的研究与优化

武 铮 金 旭 安 虹
(中国科学技术大学计算机科学与技术学院 合肥 230026)
(zhengwu@mail.ustc.edu.cn)

Research and Optimization of the Winograd-Based Convolutional Algorithm on ShenWei-26010 Many-Core Processor

Wu Zheng, Jin Xu, and An Hong
(*School of Computer Science and Technology*, *University of Science and Technology of China*, *Hefei* 230026)

Abstract As a critical component, convolution is frequently applied in deep learning. The parallel algorithms of convolution have always been a popular research topic in high-performance computing. With the rapid development of Chinese homegrown ShenWei-26010 many-core processor in artificial intelligence, there is an urgent demand for high-performance convolutional algorithms on the processor. We propose an efficient convolutional design, that is, the fused Winograd-based convolutional algorithm, towarding ShenWei-26010 architectural characteristics and the computational features of Winograd-based convolution. Unlike the traditional Winograd-based convolutional algorithm that depends on the official GEMM (general matrix multiplication) library interface, the proposed algorithm owns the customized matrix multiplication implementation. The feature makes the execution process of the proposed algorithm visible, which can better adapt to common convolutions in reality. The proposed algorithm is composed of four parts: input Winograd transformation, filter Winograd transformation, core operation, and output Winograd inverse transformation. The four parts are fused together instead of executing each part separately. The core operation can gain the required transformed data in real time. Subsequently, the computational results are transformed inversely to the final output immediately. The fused execution improves the data locality of the proposed algorithm to reduce the memory access overhead significantly. Moreover, We design other optimization methods to enhance the performance of the proposed algorithm, such as merged Winograd-transformed mode, DMA (direct memory access) double buffering, the enhanced usage of on-chip storage, the elastic processing of output data tiles, and instruction reordering. The experiments show the performance of the proposed algorithm is 7.8 times that of the traditional Winograd-based convolutional algorithm on VGG network model. Moreover, we extract the common convolution from multiple typical convolutional neural networks to measure the hardware efficiency. The results show the proposed algorithm can significantly overperform the traditional Winograd-based convolutional algorithm for all the convolution cases. The best performance of the proposed algorithm is 116.21% of the theoretical peak performance of ShenWei-26010 processor, and the average one can reach 93.14%.

Key words deep learning; Winograd-based convolution; high-performance computing; parallel algorithm; ShenWei processor

摘 要 卷积作为深度学习中被频繁使用的关键部分,其并行算法的研究已成为高性能计算领域中的热

收稿日期: 2022-09-01; **修回日期**: 2023-05-22
基金项目: 国家重点研发计划项目(2018YFB0204102)
This work was supported by the National Key Research and Development Program of China(2018YFB0204102).

门话题.随着我国自主研发的申威26010众核处理器在人工智能领域的快速发展,对面向该处理器的高性能并行卷积算法提出了迫切的需求.针对申威26010处理器的架构特征以及Winograd卷积算法的计算特性,提出了一种高性能并行卷积算法——融合Winograd卷积算法.该算法不同于依赖官方GEMM(general matrix multiplication)库接口的传统Winograd卷积算法,定制的矩阵乘实现使得该算法的执行过程变得可见,且能够更好地适应现实中常见卷积运算.整个算法由输入的Winograd变换、卷积核的Winograd变换、核心运算和输出的Winograd逆变换4部分构成,这4个部分并不是单独执行而是融合到一起执行.通过实时地为核心运算提供需要的变换后数据,并将计算结果及时地逆变换得到最终的输出数据,提高了算法执行过程中的数据局部性,极大地降低了整体的访存开销.同时,为该算法设计了合并的Winograd变换模式、DMA(direct memory access)双缓冲、片上存储的强化使用、输出数据块的弹性处理以及指令重排等优化方案.最终的实验结果表明,在VGG网络模型的总体卷积测试中,该算法性能是传统Winograd卷积算法的7.8倍.同时,抽取典型卷积神经网络模型中的卷积进行测试,融合Winograd卷积算法能够在所有的卷积场景中发挥明显高于传统Winograd卷积算法的性能.其中,最大能够发挥申威26010处理器峰值性能的116.21%,平均能够发挥峰值性能的93.14%.

关键词 深度学习;Winograd卷积;高性能计算;并行算法;申威处理器

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)作为其最成熟的网络模型之一,被广泛应用于计算机视觉[1]、语音识别[2]、自动驾驶[3]、智能医疗健康[4]等领域,以获取更高的产业效率和更好的用户体验.而当前CNNs高准确率的背后是巨大的计算代价,随着数据集变大、网络参数变多以及模型结构变得更加的复杂,CNNs对运行效率的要求越来越高.对于整个CNNs来说,90%以上的计算量都集中在卷积层中[5],这也使得众核处理器上高性能并行卷积算法的研究成为了当前学术界和工业界的热门话题之一.

当前最受欢迎的卷积算法有4种[6],分别是直接卷积算法、GEMM卷积算法、FFT卷积算法和Winograd卷积算法.直接卷积算法是基于7层循环做卷积,虽然实现简单,但是因为数据局部性差而导致性能不高.GEMM卷积算法的核心是高效的矩阵乘实现,因为许多硬件平台上有可以直接使用的高效矩阵乘库,所以GEMM卷积算法成为了加速卷积计算算法中非常受欢迎的算法,该算法可以分为显式的GEMM卷积算法[7]和隐式的GEMM卷积算法[8].相比于直接卷积算法,GEMM卷积算法并不会改变整体的计算量,只是将离散的计算操作集中并连续执行,从而提高数据的局部性以实现卷积的高效执行.而FFT卷积算法[9]和Winograd卷积算法[10]则不同,两者都是通过将输入数据和卷积核数据线性变换,然后进行对应位相乘,中间结果再进行逆线性变换得到最终的输出数据.通过这种“变换—运算—逆变换”的过程,大大降低了卷积的计算复杂度.FFT卷积算法将直接卷积算法的计算复杂度从$O(OH^2\times FH^2)$降到了$O(OH^2\times\log OH)$[6],Winograd卷积算法则进一步将卷积的计算复杂度降到了$O((OH+FH-1)^2)$[10].

Winograd卷积算法因其较低的计算复杂度,受到了广泛的关注与研究.Park等人[11]针对卷积中大量的零权重和Winograd变换中额外的加运算限制,提出了ZeroSkip硬件机制和AddOpt数据重用优化,增强后的算法能够取得51.8%的性能提升.Jia等人[12]在Intel Xeon Phi上进行了任意卷积核维度的Wingorad卷积算法的优化与实现,对比最优的GPU实现,在2D CNNs上取得了旗鼓相当的性能,在3D CNNs上性能更佳.武铮等人[13]利用Intel KNL的MCDRAM、多Memory/SNC模式等微架构特性优化Winograd卷积算法实现.测试VGG16,对比MKL-DNN有2倍多的性能加速.Mazaheri等人[14]提出了一种基于符号计算的Winograd卷积算法,利用元编程和自动调谐引入了能够为GPU自动生成高效可移植Wingorad卷积实现的系统.Jia等人[15]提出了一种基于MegaKernel的Winograd卷积实现,通过映射算法将不同的计算任务分配给GPU线程块,并构建1个按照依赖关系来获取和执行任务的调度器.结果表明,与cuDNN的2种Winograd卷积实现相比,基于MegaKernel的Winograd卷积实现有1.25倍和1.7倍的性能加速.Castro等人[16]通过汇编代码实现性能热点部分的方法,提出了一种优化的Wingorad卷积算法OpenCNN,相比于cuDNN的Winograd卷积实现,在Turing RTX 2080Ti和Ampere RTX 3090上分别加速了1.76倍和

1.85 倍. 王庆林等人[17] 在融合数据 scatter 和矩阵乘数据打包的基础上, 针对飞腾多核处理器设计了一种不依赖矩阵乘库函数的 Winograd 卷积实现, 使得 Mxnet 中 VGG16 的前向计算获得了 3.01~6.79 倍的性能加速. 总的来说, 近些年 Winograd 卷积算法在通用处理器 Intel Xeon/Xeon Phi 和 NVIDIA GPU 上得到了快速发展. 与此同时, 许多其他硬件平台的 Winograd 卷积加速也在不断吸引着研究人员投身其中, 如国产多核处理器[17]、ARM CPU[18]、FPGA[19] 等.

申威 26010 众核处理器作为世界一流超算系统“神威·太湖之光”的核心算力来源, 其低功耗高性能的特性使得其在人工智能领域拥有巨大潜力, 8×8 的从核阵列、软件控制的存储器层次结构、硬件支持的寄存器通信、从核的双流水指令运行等独特的架构特征既给了研究人员充足的可控空间, 又提出了巨大的技术挑战. 然而, 该处理器上有关卷积算法的研究一直处于初级阶段, 仅有的一些研究工作[5,8,20] 也只是针对 GEMM 卷积算法在该处理器上的高效并行实现.

综上所述, 为了进一步探索申威 26010 处理器上卷积算法的潜力, 本文详细讨论了单精度 Winograd 卷积算法在该处理器上的高性能并行设计, 主要贡献有 3 点:

1) 提出了一种并行卷积算法——融合 Winograd 卷积算法, 并为该算法设计了匹配的定制矩阵乘, 使得该算法避免了传统 Winograd 卷积算法对官方 GEMM 库接口的依赖.

2) 提出的融合 Winograd 卷积算法具有可视的执行过程, 能够结合申威处理器典型架构特征进行更细粒度的计算访存优化. 同时, 通过设计合并的 Winograd 变换模式、DMA 双缓冲、片上存储的强化使用、输出数据块的弹性处理以及指令重排等优化方案, 在提升算法性能的同时, 也为未来处理器上的并行研究工作提供了有意义的参考借鉴.

3) 从优化效果、卷积性能和卷积神经网络性能 3 个方面进行了实验. 实验结果表明, 在 VGG 网络模型的测试中, 融合 Winograd 卷积算法性能高达传统 Winograd 卷积算法性能的 7.8 倍. 通过对典型 CNNs 中常见卷积的收集测试, 融合 Winograd 卷积算法最高可以发挥申威处理器峰值性能的 116.21%, 平均可以达到 93.14%. 同时, 通过测试对比定制矩阵乘和该处理器的通用 GEMM, 表明定制矩阵乘更有利于融合 Winograd 卷积算法性能的发挥.

1 相关背景

1.1 Winograd 卷积算法

CNNs 主要包括卷积层、下采样层和全连接层等, 其中卷积层和下采样层进行特征提取, 全连接层在提取的最终特征上进行识别. 综合来看, 卷积层是 CNNs 的关键动力, 也是整个网络执行过程中最耗时的部分. 考虑到卷积层的本质是卷积运算, 因而, 如何高效地设计卷积算法已经成为了一个热门研究话题. 本文选择计算复杂度最低的 Winograd 卷积算法作为研究对象, 主要研究工作为该算法在国产申威 26010 众核处理器上的高效并行.

对于通常的卷积运算, 可以将输入数据标记为 $\boldsymbol{in}[B][IC][IH][IW]$, 表示 B 个 IC 通道的样本, 每个通道可以看作一个大小为 $IH\times IW$ 的输入特征映射; 将卷积核数据标记为 $\boldsymbol{flt}[OC][IC][FH][FW]$, 表示 OC 组卷积核, 每组 IC 个卷积核中每个卷积核的高度和宽度分别为 FH 和 FW; 将输出数据标记为 $\boldsymbol{out}[B][OC][OH][OW]$, 表示 B 个 OC 通道的样本中每个通道可以看作一个大小为 $OH\times OW$ 的输出特征映射. 除此之外, 不同的填充大小和不同的跨步大小相互组合形成了不同的卷积形式, 可以把高和宽的填充大小分别标记为 $padH$ 和 $padW$, 类似地, 高和宽的跨步大小则为 $stdH$ 和 $stdW$. 卷积的执行过程为 IC 个输入特征映射和 IC 个卷积核一一对应卷积, 然后累加 IC 个局部卷积结果, 从而得到一个输出特征映射, 因此一个完整的卷积需要 $OC\times IC$ 个卷积核. 上述卷积过程可以简化为式(1)中关于 $\boldsymbol{in}$, $\boldsymbol{flt}$, $\boldsymbol{out}$ 的张量乘法和累加.

$$\begin{aligned} &out_{b,oc,oh,ow}=\sum_{ic=0}^{IC-1}\sum_{fh=0}^{FH-1}\sum_{fw=0}^{FW-1} in_{b,ic,ih,iw}\times flt_{oc,ic,fh,fw},\\ &ih=oh\times stdH+fh-padH,\\ &iw=ow\times stdW+fw-padW, \end{aligned} \tag{1}$$

其中 $0\leqslant b<B$, $0\leqslant oc<OC$, $0\leqslant oh<OH$, $0\leqslant ow<OW$.

Winograd 卷积起源于有限脉冲响应(finite impulse response, FIR)滤波的最小滤波算法[10], 该最小滤波算法由 r 拍的 FIR 滤波器生成 m 个输出, 也就是 $F(m,r)$. 此时, 算法运算需要 $\mu(F(m,r))=m+r-1$ 次乘法操作. 对于 1 维的 Winograd 最小滤波算法可以表示为矩阵的形式:

$$\boldsymbol{y}=\boldsymbol{A}^{\mathrm{T}}[(\boldsymbol{Gw})\odot(\boldsymbol{B}^{\mathrm{T}}\boldsymbol{x})]. \tag{2}$$

通过嵌套 1 维的 Winograd 最小滤波算法可以得到 2 维的 Wingorad 最小滤波算法 $F(m\times m, r\times r)$:

$$\boldsymbol{y}=\boldsymbol{A}^{\mathrm{T}}[(\boldsymbol{G}\boldsymbol{w}\boldsymbol{G}^{\mathrm{T}})\odot(\boldsymbol{B}^{\mathrm{T}}\boldsymbol{x}\boldsymbol{B})]\boldsymbol{A}, \tag{3}$$

其中$\boldsymbol{x}$表示输入, $\boldsymbol{w}$表示过滤器, $\boldsymbol{y}$表示输出; $\boldsymbol{A}^{\mathrm{T}}$, $\boldsymbol{G}$, $\boldsymbol{B}^{\mathrm{T}}$表示该算法的系数矩阵; ⊙表示矩阵的对应位相乘, 即 Hadamard 乘积. 如果把$\boldsymbol{x}$替换为卷积中的输入数据$\boldsymbol{in}$, $\boldsymbol{w}$替换为卷积中的卷积核数据$\boldsymbol{flt}$, $\boldsymbol{y}$替换为卷积中的输出数据$\boldsymbol{out}$, 参考文献[10]则可以得到 Winograd 卷积算法, 如算法 1 所示.

算法 1. Winograd 卷积算法.

$\alpha=m+r-1$是输入数据块的大小, 且相邻 2 个块间$r-1$重叠;

$T=B\lceil OH/m\rceil\lceil OW/m\rceil$是输出数据块的数量;

$\boldsymbol{in}_{t,ic}\in\mathbb{R}^{\alpha\times\alpha}$是$ic$通道上索引为$t$的输入数据块, $\widetilde{\boldsymbol{in}}$是$\boldsymbol{in}$通过 Winograd 变换后的数据, $\widetilde{\boldsymbol{IN}}$则是$\widetilde{\boldsymbol{in}}$通过分散(scatter)后用于核心运算中矩阵乘的数据;

$\boldsymbol{flt}_{oc,ic}\in\mathbb{R}^{r\times r}$是$oc$组内索引为$ic$的卷积核数据块, $\widetilde{\boldsymbol{flt}}$是$\boldsymbol{flt}$通过 Winograd 变换后的数据, $\widetilde{\boldsymbol{FLT}}$则是$\widetilde{\boldsymbol{flt}}$通过 scatter 后用于核心运算中矩阵乘的数据;

$\boldsymbol{out}_{t,oc}\in\mathbb{R}^{m\times m}$是$oc$通道上索引为$t$的输出数据块, $\widetilde{\boldsymbol{OUT}}$是核心运算中矩阵乘后的输出数据, $\widetilde{\boldsymbol{out}}$则是$\widetilde{\boldsymbol{OUT}}$通过收集(gather)后用于 Winograd 逆变换的数据;

/* 卷积核数据的 Winograd 变换 */

① for oc=0,1,⋯,$OC-1$ do

② for ic=0,1,⋯,$IC-1$ do

③ $\widetilde{\boldsymbol{flt}}=\boldsymbol{G}\boldsymbol{flt}_{oc,ic}\boldsymbol{G}^{\mathrm{T}}\in\mathbb{R}^{\alpha\times\alpha}$;

④ 将$\widetilde{\boldsymbol{flt}}$中元素 scatter 到$\widetilde{\boldsymbol{FLT}}$: $\widetilde{\boldsymbol{FLT}}_{oc,ic}^{(\xi,\nu)}=\widetilde{\boldsymbol{flt}}_{\xi,\nu}$;

⑤ end for

⑥ end for

/* 输入数据的 Winograd 变换 */

⑦ for t=0,1,⋯,$T-1$ do

⑧ for ic=0,1,⋯,$IC-1$ do

⑨ $\widetilde{\boldsymbol{in}}=\boldsymbol{B}^{\mathrm{T}}\boldsymbol{in}_{t,ic}\boldsymbol{B}\in\mathbb{R}^{\alpha\times\alpha}$;

⑩ 将$\widetilde{\boldsymbol{in}}$中元素 scatter 到$\widetilde{\boldsymbol{IN}}$: $\widetilde{\boldsymbol{IN}}_{t,ic}^{(\xi,\nu)}=\widetilde{in}_{\xi,\nu}$;

⑪ end for

⑫ end for

/* 核心运算 */

⑬ for ξ=0,1,⋯,$\alpha-1$ do

⑭ for ν=0,1,⋯,$\alpha-1$ do

⑮ $\widetilde{\boldsymbol{OUT}}^{(\xi,\nu)}=\widetilde{\boldsymbol{FLT}}^{(\xi,\nu)}\widetilde{\boldsymbol{IN}}^{(\xi,\nu)}$;

⑯ end for

⑰ end for

/* 输出数据的 Winograd 逆变换 */

⑱ for t=0,1,⋯,$T-1$ do

⑲ for oc=0,1,⋯,$OC-1$ do

⑳ 从$\widetilde{\boldsymbol{OUT}}$中 gather 元素到$\widetilde{\boldsymbol{out}}$: $\widetilde{\boldsymbol{out}}_{\xi,\nu}=\widetilde{\boldsymbol{OUT}}_{t,oc}^{(\xi,\nu)}$;

㉑ $\boldsymbol{out}_{t,oc}=\boldsymbol{A}^{\mathrm{T}}\widetilde{out}\boldsymbol{A}$;

㉒ end for

㉓ end for

对于 Wingorad 卷积算法, 系数矩阵$\boldsymbol{A}^{\mathrm{T}}$, $\boldsymbol{G}$, $\boldsymbol{B}^{\mathrm{T}}$是由$m$和$r$决定的, 以$F(2\times2,3\times3)$为例, 此时有

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

$$\boldsymbol{B}^{\mathrm{T}}=\begin{pmatrix}1&0&-1&0\\0&1&1&0\\0&-1&1&0\\0&1&0&-1\end{pmatrix}.$$

1.2 申威 26010 众核处理器

申威 26010 众核处理器[21-22]是由上海高性能集成电路设计中心自主研发的一款国产异构众核处理器, 支持 64 b 自主神威指令级系统, 采用分布式共享存储和片上计算阵列. 如图 1 所示, 该处理器单芯片由 4 个等价的核组(core group, CG)构成, 核组间通过片上网络(network on chip, NoC)互连. 每个核组由 1 个主核(management processing element, MPE)和 64 个从核(computing processing element, CPE)组成, 共计 260 个计算核心. 每个核组私有 1 个 4 路 128 b 的 DDR3 主存控制器(memory controller, MC)和 1 个协议处理单元(protocol processing unit, PPU), 并通过 MC 直连 1 块 8 GB 的 DDR3 主存.

主核和从核的工作频率都是 1.45 GHz, 两者都支持 256 b 的浮点向量乘加指令, 不同的是主核有 2 条浮点运算流水, 从核仅有 1 条浮点运算流水. 同时, 双精度数据运算和单精度数据运算共用双精度浮点运算单元的微体系结构特征, 使得该处理器上浮点数据的向量长度都为 4. 基于此, 单核组从核阵列的理论单精度浮点峰值是 742.4 GFLOPS, 主核是 23.2 GFLOPS.其中计算性能约 97% 来源于从核阵列, 可见在该处理器上进行性能优化最为关键的任务就是如何高效地组织利用从核阵列的各种资源, 以充分

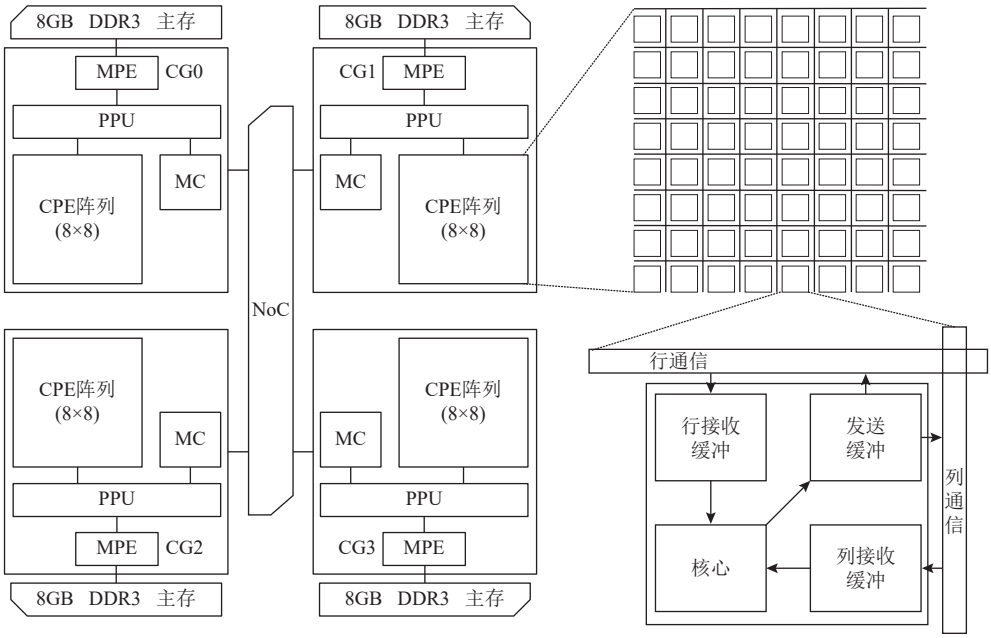


Fig. 1 The architecture of ShenWei-26010 processor

图 1 申威 26010 的处理器架构

发挥从核阵列的计算性能. 每个主核拥有 2 级私有缓存, 包括 1 个 32 KB 的 L1 指令缓存、1 个 32 KB 的 L1 数据缓存以及 1 个 256 KB 的 L2 缓存. 每个从核都有 1 个 16 KB 的 L1 指令缓存和 1 个 64 KB 本地设备内存(local device memory, LDM). 从核阵列上的 64 个从核共享一个大小为 64 KB 的直接映射的 L2 指令缓存.

该处理器为了尽可能缓解片外访存的压力, 提供了 2 个核心技术. 一个是不同的主从核间的数据访问方式: 1)gld/gst 离散访主存, 即是从核直接对主存进行读写操作, 这种方式的好处是简单易用, 缺陷就是速度很慢, 访存延迟高达 278 个时钟周期; 2)DMA(direct memory access)批量式访主存, 即是从核先通过 DMA 操作将主存数据提取到 LDM, 然后再对 LDM 内的数据进行相关操作, 整个过程的访存延迟较低, 大约 29 个时钟周期. 其中, DMA 支持多种访存模式, 应用最为广泛的有 PE_MODE, ROW_MODE, BROW_MODE. 另一个是寄存器通信实现同一核组内 64 个从核的片上数据共享, 为了有效支持这一机制, 每个从核上配备了发送缓冲区、行接收缓冲区和列接收缓冲区, 发送缓冲区可以缓冲 6 个寄存器消息, 2 个接收缓冲区可以分别缓冲 4 个寄存器消息. 寄存器通信机制需要注意 3 点: 1)通信时数据的大小固定为 256 b; 2)行和列都不同的 2 个从核之间不能直接进行通信, 需要借助同行或者同列上的从核作为中间转折点; 3)通信的过程是匿名的, 当多个从核发送消息到某个从核时, 该从核基于先到先得的策略接收消息. Benchmarking[23] 显示每个从核阵列上寄存器通信的带宽在 P2P 模式和广播模式下分别可以达到 637.25 GBps 和 1 115.25 GBps.

申威 26010 处理器每个从核支持 2 条硬件流水线 P0 和 P1. 其中, P0 支持浮点和整数的标量/向量操作, P1 支持数据迁移、比较、跳转和整数标量操作. 2 条流水线共享 1 个指令解码器(instruction decoder, ID)和 1 个指令队列, 每个时钟周期 ID 对队列中前 2 条指令进行检测, 当满足 3 种情况时 2 条指令可以同时被加载到 2 条流水中: 1)2 条指令同已发射未完成的指令不存在冲突; 2)2 条指令间不存在写后读和写后写冲突; 3)2 条指令可以分别被 2 条流水处理. 不难看出, 如何混合交差程序中 2 种类型的指令, 使 P0 和 P1 能够并行运行, 对于该处理器性能的发挥起着重要的作用.

2 Winograd 卷积算法的并行优化

首先介绍提出的并行 Winograd 卷积算法的整体设计思路. 然后以 $F(2\times2,3\times3)$为例, 结合申威 26010 处理器架构特征, 详细阐述该算法的各种计算和访存的优化方案. 最后, 对研究工作的通用性进行了分析, 以期望能够对其他众核处理器上卷积研究提供有益的参考借鉴.

2.1 融合 Winograd 卷积算法

如算法 1 所示，Winograd 卷积算法的基本运算流程可以分为 4 个部分：

1）通过系数矩阵$\boldsymbol{G}$和$\boldsymbol{G}^{\mathrm{T}}$对$\boldsymbol{flt}[IC][OC][FH][FW]$进行 Winograd 变换，得到$\widetilde{\boldsymbol{flt}}[IC][OC][\alpha][\alpha]$，然后再将变换后的元素 scatter 到$\widetilde{\boldsymbol{FLT}}[\alpha][\alpha][OC][IC]$；

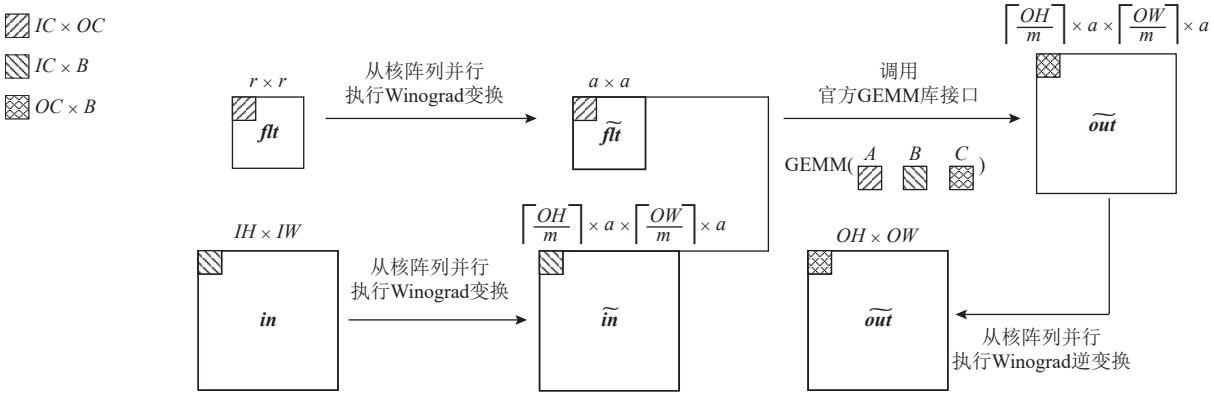
2）通过系数矩阵$\boldsymbol{B}^{\mathrm{T}}$和$\boldsymbol{B}$对$\boldsymbol{in}[B][IC][IH][IW]$进行 Wingorad 变换，得到变换后数据$\widetilde{\boldsymbol{in}}[B][IC]\left[\left\lceil\frac{OH}{m}\right\rceil\right][\alpha]\left[\left\lceil\frac{OW}{m}\right\rceil\right][\alpha]$，然后再将变换后的元素 scatter 到$\widetilde{\boldsymbol{IN}}[\alpha][\alpha][IC]\left[\left\lceil\frac{OH}{m}\right\rceil\right]\left[\left\lceil\frac{OW}{m}\right\rceil\right]B$；

3）进行$\alpha\times\alpha$次的矩阵乘运算，从而得到中间输出数据$\widetilde{\boldsymbol{OUT}}[\alpha][\alpha][OC]\left[\left\lceil\frac{OH}{m}\right\rceil\right]\left[\left\lceil\frac{OW}{m}\right\rceil\right]B$；

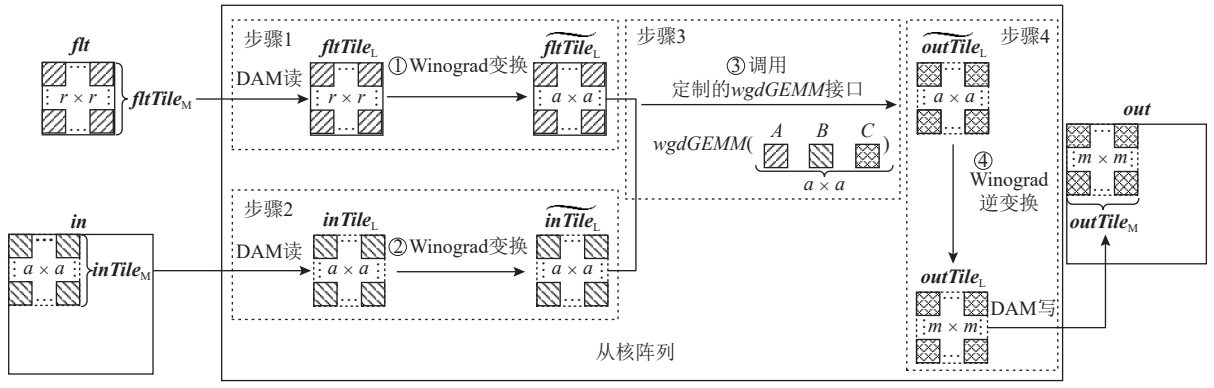
4）从$\widetilde{\boldsymbol{OUT}}[\alpha][\alpha][OC]\left[\left\lceil\frac{OH}{m}\right\rceil\right]\left[\left\lceil\frac{OW}{m}\right\rceil\right]B$中 gather 元素得到$\widetilde{\boldsymbol{out}}[B][OC]\left[\left\lceil\frac{OH}{m}\right\rceil\right][\alpha]\left[\left\lceil\frac{OW}{m}\right\rceil\right][\alpha]$，然后通过系数矩阵$\boldsymbol{A}^{\mathrm{T}}$和$\boldsymbol{A}$对$\widetilde{\boldsymbol{out}}[B][OC]\left[\left\lceil\frac{OH}{m}\right\rceil\right][\alpha]\left[\left\lceil\frac{OW}{m}\right\rceil\right][\alpha]$进行 Winograd 逆变换，从而得到最终的输出数据$\boldsymbol{out}[B][OC][OH][OW]$.

现代处理器存在的普遍问题就是访存速度无法跟上计算能力，申威 26010 处理器在这方面尤为严重，其每字节浮点计算率高达 33.84[23]. 而通用处理器 Intel KNL 7290 和 NVIDIA Tesla V100 分别为 7.05[24] 和 7.78[25]，可见申威 26010 处理器每字节的片外主存数据访问需要匹配远高于通用处理器的计算量. 对于许多工作来说，要想充分发挥该处理器的性能，访存受限无疑是一个巨大的挑战. 在上述 Winograd 卷积算法的基本运算流程中，scatter 和 gather 过程中的维度变化都是大量的离散访存操作，这将造成难以忍受的高额访存开销. 为了解决上述问题，设置了新的数据格式——$\boldsymbol{in}[IH][IW][IC][B]$，$\boldsymbol{flt}[FH][FW][IC][OC]$，$\boldsymbol{out}[OH][OW][OC][B]$. 这些数据格式的核心便是通过直接使用卷积过程中天然的矩阵乘关系（对应矩阵乘参数M，N，K分别为OC，B，IC），在避免 scatter 和 gather 过程中的维度变化造成的高额访存开销的同时，也省去了存储中间数据而需要的额外内存资源. 另一方面，将矩阵乘关系放到低维度中可以尽可能地提高整个算法执行过程中访存的连续性.

如图 2（a）所示，Winograd 卷积最直接的实现方



(a) 传统Winograd卷积算法



(b) 融合Winograd卷积算法

Fig. 2 Basic framework of the fused Winograd convolution algorithm

图 2 融合 Wingorad 卷积算法的基本框架

法就是在步骤 3 中调用官方 GEMM 库接口，而步骤 1、步骤 2 和步骤 4 利用从核阵列并行执行 Winograd 变换和 Winograd 逆变换，这是一种传统的 Winograd 卷积算法. 该算法的优点是实现简单方便，缺点是中间数据需要消耗大量片外存储资源，极低的数据重用导致频繁的片外访存，高额的访存开销导致即便有着高性能的 GEMM 库接口也难以实现卷积的高效运行.

为了充分挖掘 Winograd 卷积算法在申威 26010 处理器上的潜力，本文提出了一种不依赖官方 GEMM 库接口的算法. 该算法能够将原本独立执行的 Winograd 变换、核心运算和 Winograd 逆变换融合到一起，以充分发挥 Winograd 卷积算法本身潜在的数据重用，从而尽可能降低访存对算法执行效率的影响，将其称之为“融合 Winograd 卷积算法”. 如图 2(b)所示，将原卷积数据中的最后 2 维看成 1 个元素，那么 $\boldsymbol{in}$, $\boldsymbol{flt}$, $\boldsymbol{out}$ 则分别可以看成 $IH\times IW$, $FH\times FW$, $OH\times OW$ 的 2 维数组，且三者的单元素大小分别为 $IC\times B$, $IC\times OC$, $OC\times B$. 在融合 Winograd 算法中，步骤 1 是通过 DMA 读取主存上 $r\times r$ 的卷积核数据块 $\boldsymbol{fltTile}_{\mathrm{M}}$ 到 LDM 空间中的 $\boldsymbol{fltTile}_{\mathrm{L}}$，再通过 Winograd 变换得到 $\alpha\times\alpha$ 的 $\widetilde{\boldsymbol{fltTile}}_{\mathrm{L}}$；步骤 2 与步骤 1 类似，先是读取 $\alpha\times\alpha$ 的输入数据块 $\boldsymbol{inTile}_{\mathrm{M}}$ 到 LDM 空间中的 $\boldsymbol{inTile}_{\mathrm{L}}$，然后通过 Winograd 变换得到 $\alpha\times\alpha$ 的 $\widetilde{\boldsymbol{inTile}}_{\mathrm{L}}$；为了能够更好地适应融合 Winograd 卷积算法中核心运算的实际情况，定制了高效的矩阵乘实现，并提供了便于该算法调用的从核函数接口 *wgdGEMM*. 步骤 3 则是通过一一对应的方式调用 $\alpha\times\alpha$ 次的 *wgdGEMM*，从而得到 $\alpha\times\alpha$ 的 $\widetilde{\boldsymbol{outTile}}_{\mathrm{L}}$；步骤 4 执行 Winograd 逆变换得到 $\boldsymbol{outTile}_{\mathrm{L}}$，并 DMA 写回到主存中的对应位置 $\boldsymbol{outTile}_{\mathrm{M}}$. 考虑整个过程中 $\boldsymbol{fltTile}_{\mathrm{M}}$ 是固定的，而 $\boldsymbol{inTile}_{\mathrm{M}}$ 和 $\boldsymbol{outTile}_{\mathrm{M}}$ 则是通过移动滑窗获取的，所以 $\boldsymbol{fltTile}_{\mathrm{M}}$ 将会被反复使用 $\left\lceil\frac{OH}{m}\right\rceil\left\lceil\frac{OW}{m}\right\rceil$ 次. 为了最大化 $\boldsymbol{fltTile}_{\mathrm{M}}$ 的时间局部性，设计算法仅且只执行 1 次步骤 1，并将变换后的结果 $\widetilde{\boldsymbol{fltTile}}_{\mathrm{L}}$ 长期驻留在片上存储 LDM 中，直到卷积结束. 步骤 2、步骤 3 和步骤 4 则会随着滑窗的移动获取不同的 $\boldsymbol{inTile}_{\mathrm{M}}$ 和 $\boldsymbol{outTile}_{\mathrm{M}}$，继而执行 $\left\lceil\frac{OH}{m}\right\rceil\left\lceil\frac{OW}{m}\right\rceil$ 次得到最终的输出数据 $\boldsymbol{out}$.

2.2 算法优化方案

后续内容将以 $F(2\times2,3\times3)$ 为例进行详细阐述，即 $m=2$, $r=3$, $\alpha=4$. 在 2.1 节融合 Winograd 卷积算法的基础上，结合申威 26010 处理器的架构特征以及卷积运算的实际情况，进一步探索细粒度的访存和计算的优化方案.

2.2.1 合并的 Winograd 变换模式

考虑到申威 26010 处理器在进行矩阵乘的计算 kernel 设计时，需要进行向量加载和寄存器通信. 而该处理器仅支持双精度数据情况下单条指令完成向量加载和寄存器通信. 如果是单精度数据的话，则需要分 2 条指令进行，且这 2 条指令之间存在写后读关系，将极大地降低计算 kernel 的指令级并行度. 因此，选择在片上存储 LDM 中提前进行单精度数据和双精度数据的类型转换，从而保证送入 *wgdGEMM* 中的源数据都是双精度数据，以最大化卷积算法的指令级并行度. 为此，可以设置融合 Winograd 卷积算法中 $\boldsymbol{inTile}_{\mathrm{L}}$, $\boldsymbol{fltTile}_{\mathrm{L}}$, $\boldsymbol{outTile}_{\mathrm{L}}$ 用于存储 LDM 上的单精度数据，$\widetilde{\boldsymbol{inTile}}_{\mathrm{L}}$, $\widetilde{\boldsymbol{fltTile}}_{\mathrm{L}}$, $\widetilde{\boldsymbol{outTile}}_{\mathrm{L}}$ 则用于存储 LDM 上的双精度数据. 此时，对于输入数据块和卷积核数据块的 Winograd 变换，以及输出数据块的 Winograd 逆变换可以表示为：

$$\begin{aligned}\widetilde{\boldsymbol{inTile}}_{\mathrm{L}}&=(\mathrm{double})(\boldsymbol{B}^{\mathrm{T}}\boldsymbol{inTile}_{\mathrm{L}}\boldsymbol{B}),\\ \widetilde{\boldsymbol{fltTile}}_{\mathrm{L}}&=(\mathrm{double})(\boldsymbol{G}\boldsymbol{fltTile}_{\mathrm{L}}\boldsymbol{G}^{\mathrm{T}}),\\ \boldsymbol{outTile}_{\mathrm{L}}&=(\mathrm{float})(\boldsymbol{A}^{\mathrm{T}}\widetilde{\boldsymbol{outTile}}_{\mathrm{L}}\boldsymbol{A}).\end{aligned}\tag{4}$$

在 $F(2\times2,3\times3)$ 中，$\boldsymbol{B}^{\mathrm{T}}$, $\boldsymbol{G}$, $\boldsymbol{A}^{\mathrm{T}}$ 为确定的常数矩阵，具体值可以参见 1.1 节. 此时，$\boldsymbol{inTile}_{\mathrm{L}}$ 的维度为 $4\times4\times IC\times B$，可以将其简单视为 $16\times IC\times B$. 相应的，$\widetilde{\boldsymbol{inTile}}_{\mathrm{L}}$ 维度也可以表示为 $16\times IC\times B$. 那么对于输入数据块的 Winograd 变换直观上可以分为二次矩阵乘运算，称之为“分离的 Winograd 变换模式”，如图 3 所示. 其中 i=0,1,⋯,$IC\times B-1$，整个过程需要 $224\times IC\times B$ 次浮点运算. 这种 Winograd 变换方式虽然简单直观，但是不仅浮点运算量大，而且中间数据 $\boldsymbol{tmp}$ 会增大片上存储资源 LDM 的开销，如果引入向量化则更会使这种额外的 LDM 开销成倍增加.

为了解决上述问题，设计了合并的 Winograd 变换模式，通过将 $\boldsymbol{B}^{\mathrm{T}}\boldsymbol{inTile}_{\mathrm{L}}\boldsymbol{B}$ 的二次矩阵乘运算合并到一次，直接获取 $\widetilde{\boldsymbol{inTile}}_{\mathrm{L}}$ 中每个元素关于 $\boldsymbol{inTile}_{\mathrm{L}}$ 中 16 个元素的线性关系，结果如图 4 所示.

通过合并的 Winograd 变换模式，将浮点运算次数降低到了 $48\times IC\times B$，仅为原计算量的 21.4%. 同时，消除了中间数据带来的额外 LDM 开销，使得向量化的使用不再受限制. 因为申威 26010 处理器的浮点向量长度为 4，如果在合并的 Winograd 变换模式中加入向量化，计算量将进一步降低至原计算量的 5.35%.

类似地，对于 $\boldsymbol{fltTile}_{\mathrm{L}}$ 和 $\widetilde{\boldsymbol{fltTile}}_{\mathrm{L}}$，通过合并的

第1次矩阵乘运算

$$\boldsymbol{tmp}[0:15]=\begin{pmatrix}1&0&-1&0\\0&1&1&0\\0&-1&1&0\\0&1&0&-1\end{pmatrix}\begin{pmatrix}inTile_{\mathrm{L}}[0][i]&inTile_{\mathrm{L}}[1][i]&inTile_{\mathrm{L}}[2][i]&inTile_{\mathrm{L}}[3][i]\\inTile_{\mathrm{L}}[4][i]&inTile_{\mathrm{L}}[5][i]&inTile_{\mathrm{L}}[6][i]&inTile_{\mathrm{L}}[7][i]\\inTile_{\mathrm{L}}[8][i]&inTile_{\mathrm{L}}[9][i]&inTile_{\mathrm{L}}[10][i]&inTile_{\mathrm{L}}[11][i]\\inTile_{\mathrm{L}}[12][i]&inTile_{\mathrm{L}}[13][i]&inTile_{\mathrm{L}}[14][i]&inTile_{\mathrm{L}}[15][i]\end{pmatrix}$$

第2次矩阵乘运算

$$\widetilde{\boldsymbol{inTile}}_{\mathrm{L}}[0:15][i]=(\mathrm{double})\begin{pmatrix}tmp_{\mathrm{L}}[0]&tmp_{\mathrm{L}}[1]&tmp_{\mathrm{L}}[2]&tmp_{\mathrm{L}}[3]\\tmp_{\mathrm{L}}[4]&tmp_{\mathrm{L}}[5]&tmp_{\mathrm{L}}[6]&tmp_{\mathrm{L}}[7]\\tmp_{\mathrm{L}}[8]&tmp_{\mathrm{L}}[9]&tmp_{\mathrm{L}}[10]&tmp_{\mathrm{L}}[11]\\tmp_{\mathrm{L}}[12]&tmp_{\mathrm{L}}[13]&tmp_{\mathrm{L}}[14]&tmp_{\mathrm{L}}[15]\end{pmatrix}\begin{pmatrix}1&0&0&0\\0&1&-1&1\\-1&1&1&0\\0&0&0&-1\end{pmatrix}$$

Fig. 3 Separated Winograd-transformed mode

图 3 分离的 Winograd 变换模式

$$\begin{aligned}
&\widetilde{inTile}_{\mathrm{L}}[0][i]=(\mathrm{double})((inTile_{\mathrm{L}}[0][i]-inTile_{\mathrm{L}}[8][i])-(inTile_{\mathrm{L}}[2][i]-inTile_{\mathrm{L}}[10][i]))\\
&\widetilde{inTile}_{\mathrm{L}}[1][i]=(\mathrm{double})((inTile_{\mathrm{L}}[1][i]-inTile_{\mathrm{L}}[9][i])+(inTile_{\mathrm{L}}[2][i]-inTile_{\mathrm{L}}[10][i]))\\
&\widetilde{inTile}_{\mathrm{L}}[2][i]=(\mathrm{double})((inTile_{\mathrm{L}}[2][i]-inTile_{\mathrm{L}}[10][i])-(inTile_{\mathrm{L}}[1][i]-inTile_{\mathrm{L}}[9][i]))\\
&\widetilde{inTile}_{\mathrm{L}}[3][i]=(\mathrm{double})((inTile_{\mathrm{L}}[1][i]-inTile_{\mathrm{L}}[9][i])-(inTile_{\mathrm{L}}[3][i]-inTile_{\mathrm{L}}[11][i]))\\
&\widetilde{inTile}_{\mathrm{L}}[4][i]=(\mathrm{double})((inTile_{\mathrm{L}}[4][i]+inTile_{\mathrm{L}}[8][i])-(inTile_{\mathrm{L}}[6][i]+inTile_{\mathrm{L}}[10][i]))\\
&\widetilde{inTile}_{\mathrm{L}}[5][i]=(\mathrm{double})((inTile_{\mathrm{L}}[5][i]+inTile_{\mathrm{L}}[9][i])+(inTile_{\mathrm{L}}[6][i]+inTile_{\mathrm{L}}[10][i]))\\
&\widetilde{inTile}_{\mathrm{L}}[6][i]=(\mathrm{double})((inTile_{\mathrm{L}}[6][i]+inTile_{\mathrm{L}}[10][i])-(inTile_{\mathrm{L}}[5][i]+inTile_{\mathrm{L}}[9][i]))\\
&\widetilde{inTile}_{\mathrm{L}}[7][i]=(\mathrm{double})((inTile_{\mathrm{L}}[5][i]+inTile_{\mathrm{L}}[9][i])-(inTile_{\mathrm{L}}[7][i]+inTile_{\mathrm{L}}[11][i]))\\
&\widetilde{inTile}_{\mathrm{L}}[8][i]=(\mathrm{double})((inTile_{\mathrm{L}}[8][i]-inTile_{\mathrm{L}}[4][i])-(inTile_{\mathrm{L}}[10][i]-inTile_{\mathrm{L}}[6][i]))\\
&\widetilde{inTile}_{\mathrm{L}}[9][i]=(\mathrm{double})((inTile_{\mathrm{L}}[9][i]-inTile_{\mathrm{L}}[5][i])+(inTile_{\mathrm{L}}[10][i]-inTile_{\mathrm{L}}[6][i]))\\
&\widetilde{inTile}_{\mathrm{L}}[10][i]=(\mathrm{double})((inTile_{\mathrm{L}}[10][i]-inTile_{\mathrm{L}}[6][i])-(inTile_{\mathrm{L}}[9][i]-inTile_{\mathrm{L}}[5][i]))\\
&\widetilde{inTile}_{\mathrm{L}}[11][i]=(\mathrm{double})((inTile_{\mathrm{L}}[9][i]-inTile_{\mathrm{L}}[5][i])-(inTile_{\mathrm{L}}[11][i]-inTile_{\mathrm{L}}[7][i]))\\
&\widetilde{inTile}_{\mathrm{L}}[12][i]=(\mathrm{double})((inTile_{\mathrm{L}}[4][i]-inTile_{\mathrm{L}}[12][i])-(inTile_{\mathrm{L}}[6][i]-inTile_{\mathrm{L}}[14][i]))\\
&\widetilde{inTile}_{\mathrm{L}}[13][i]=(\mathrm{double})((inTile_{\mathrm{L}}[5][i]-inTile_{\mathrm{L}}[13][i])+(inTile_{\mathrm{L}}[6][i]-inTile_{\mathrm{L}}[14][i]))\\
&\widetilde{inTile}_{\mathrm{L}}[14][i]=(\mathrm{double})((inTile_{\mathrm{L}}[6][i]-inTile_{\mathrm{L}}[14][i])-(inTile_{\mathrm{L}}[5][i]-inTile_{\mathrm{L}}[13][i]))\\
&\widetilde{inTile}_{\mathrm{L}}[15][i]=(\mathrm{double})((inTile_{\mathrm{L}}[5][i]-inTile_{\mathrm{L}}[13][i])-(inTile_{\mathrm{L}}[7][i]-inTile_{\mathrm{L}}[15][i]))
\end{aligned}$$

Fig. 4 Merged Winograd-transformed mode

图 4 合并的 Winograd 变换模式

Winograd 变换模式，整个变换过程中计算量降为原计算量的 11.43%；对于 $\boldsymbol{outTile}_{\mathrm{L}}$ 和 $\widetilde{\boldsymbol{outTile}}_{\mathrm{L}}$，通过合并的 Winograd 变换模式，整个变换过程中计算量降为原计算量的 9.52%.

2.2.2 DMA 双缓冲

申威 26010 处理器支持异步的 DMA 访存，因此有可能通过精心设计算法，从而尽可能地将 DMA 访存开销分摊到核心运算上，缓解该处理器的片外访存压力，提高并行算法性能.

算法 2. 基于 DMA 双缓冲的融合 Winograd 卷积算法.

① 开始 $\boldsymbol{fltTile}_{\mathrm{M}}$ 到 $\boldsymbol{fltTile}_{\mathrm{L}}$ 的 DMA 读；

② 结束 $\boldsymbol{fltTile}_{\mathrm{M}}$ 到 $\boldsymbol{fltTile}_{\mathrm{L}}$ 的 DMA 读；

③ $\widetilde{\boldsymbol{fltTile}}_{\mathrm{L}}=(\mathrm{double})(\boldsymbol{G}\boldsymbol{fltTile}_{\mathrm{L}}\boldsymbol{G}^{\mathrm{T}})$；

④ 计算首轮 DMA 双缓冲需要的 oh'，ow'；

⑤ 基于 oh'，ow' 开始 $\boldsymbol{inTile}_{\mathrm{M}}$ 到 $\boldsymbol{inTile}_{\mathrm{L}}[cmpt]$ 的 DMA 读；

⑥ 结束 $\boldsymbol{inTile}_{\mathrm{M}}$ 到 $\boldsymbol{inTile}_{\mathrm{L}}[cmpt]$ 的 DMA 读；

⑦ 开始一轮空的 $\boldsymbol{outTile}_{\mathrm{L}}[ldst]$ 到 $\boldsymbol{outTile}_{\mathrm{M}}$ 的 DMA 写；

⑧ for oh=0,1,…, $\left\lceil\frac{OH}{2}\right\rceil-1$ do

⑨ for ow=0,1,…, $\left\lceil\frac{OW}{2}\right\rceil-1$ do

⑩ 计算下一轮 DMA 双缓冲需要的 oh'，ow'；

⑪ 基于 oh'，ow' 开始 $\boldsymbol{inTile}_{\mathrm{M}}$ 到 $\boldsymbol{inTile}_{\mathrm{L}}[ldst]$ 的 DMA 读；

⑫ $\widetilde{\boldsymbol{inTile}}_{\mathrm{L}}=(\mathrm{double})(\boldsymbol{B}^{\mathrm{T}}\boldsymbol{inTile}_{\mathrm{L}}[cmpt]\boldsymbol{B})$；

⑬ 初始化 $\widetilde{\boldsymbol{outTile}}_{\mathrm{L}}$ 值为 0；

⑭ $wgdGEMM(\widetilde{\boldsymbol{fltTile}}_{\mathrm{L}}[0:16],\widetilde{\boldsymbol{inTile}}_{\mathrm{L}}[0:16],\widetilde{\boldsymbol{outTile}}_{\mathrm{L}}[0:16])$；

⑮ $\boldsymbol{outTile}_{\mathrm{L}}[cmpt]=(\mathrm{float})(\boldsymbol{A}^{\mathrm{T}}\widetilde{\boldsymbol{outTile}}_{\mathrm{L}}\boldsymbol{A})$；

⑯ 结束 $\boldsymbol{inTile}_{\mathrm{M}}$ 到 $\boldsymbol{inTile}_{\mathrm{L}}[ldst]$ 的 DMA 读；

⑰ 结束 $\boldsymbol{outTile}_{\mathrm{L}}[ldst]$ 到 $\boldsymbol{outTile}_{\mathrm{M}}$ 的 DMA 写；

⑱ 基于 oh,ow 开始 $\boldsymbol{outTile}_{\mathrm{L}}[cmpt]$ 到 $\boldsymbol{outTile}_{\mathrm{M}}$

的 DMA 写；

⑲ 交换*cmpt*和*ldst*的值；

⑳ end for

㉑ end for

㉒ 结束$\boldsymbol{outTile}_{\mathrm{L}}[ldst]$到$\boldsymbol{outTile}_{\mathrm{M}}$的 DMA 写.

如算法 2 所示，DMA 双缓冲的核心思想是通过预先执行 1 次循环的 DMA 操作，以消耗部分双倍的 LDM 片上存储为代价，使得相邻 2 次循环间的 DMA 操作和核心运算能够无依赖并行，从而掩藏掉部分访存时间. 其中，*ldst*表示存储 DMA 操作所需数据的 LDM 空间，*cmpt*表示存储当前核心运算所需数据的 LDM 空间. 其中$\boldsymbol{fltTile}_{\mathrm{L}}$用于 DMA 读取卷积核数据块，并将$\boldsymbol{fltTile}_{\mathrm{L}}$在 Winograd 变换后的数据以双精度的形式放入$\widetilde{\boldsymbol{fltTile}}_{\mathrm{L}}$. 考虑到 Winograd 卷积中卷积核数据的反复使用，$\widetilde{\boldsymbol{fltTile}}_{\mathrm{L}}$中数据将会常驻 LDM 空间，直至卷积结束. 同时，设置$\boldsymbol{inTile}_{\mathrm{L}}[cmpt]$和$\boldsymbol{inTile}_{\mathrm{L}}[ldst]$用于双缓冲输入数据块$\boldsymbol{inTile}_{\mathrm{M}}$的 DMA 读取，设置$\boldsymbol{outTile}_{\mathrm{L}}[cmpt]$和$\boldsymbol{outTile}_{\mathrm{L}}[ldst]$用于双缓冲输出数据块$\boldsymbol{outTile}_{\mathrm{M}}$的 DMA 写回. 相应地，分配$\widetilde{\boldsymbol{inTile}}_{\mathrm{L}}$和$\widetilde{\boldsymbol{outTile}}_{\mathrm{L}}$用于存储当前核心运算所需要的对应双精度数据. 在算法 2 中，实现了下一次$\boldsymbol{inTile}_{\mathrm{M}}$的 DMA 读取和上一次$\boldsymbol{outTile}_{\mathrm{M}}$的 DMA 写回同当前核心运算的并行执行，理论最优情况下将会实现大约 2 倍的性能提升.

2.2.3 片上存储的强化使用

申威 26010 处理器提供了用户可控的片上存储 LDM，但是单个从核的 LDM 仅有 64 KB，有限的 LDM 容量要求研究人员不得不精心设计算法，以尽可能提高片上存储资源的使用效率. 在算法 2 中，双缓冲虽然能够很好地实现核心运算和 DMA 访存的并行，但是也会使部分使用中的 LDM 空间翻倍，同时考虑到存储双精度数据带来的额外 LDM 消耗，这些都给本就有限的 LDM 带来了巨大压力. 为了缓解算法 2 中片上存储的使用压力，设计了 2 种优化方案：展开的 LDM 强化使用和交错的 LDM 强化使用. 两者都是从算法设计的角度，通过重新组织算法的执行流程，寻找能够节省的 LDM 空间.

假设$B = IC = OC$，如图 5 所示，算法 2 中初始使用的 LDM 空间大小为 $580B^2$字节. 在展开的 LDM 强化使用中，将核心运算的整体展开并拆分成 16 次独立的 *wgdGEMM*，依次标识为 *wgdGEMM*[0]~[15]. 同时，结合 2.2.1 节中合并的 Winograd 变换模式下的线性关系，如式(5)所示.

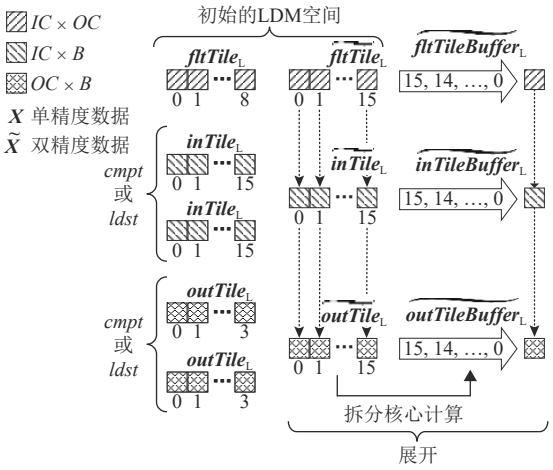


Fig. 5 Unfolding LDM enhanced usage

图 5 展开的 LDM 强化使用

$$
\begin{aligned}
&\widetilde{\boldsymbol{fltTile}}_{\mathrm{L}}[i] = (\mathrm{double})(f_i^{\mathrm{flt}}(\boldsymbol{fltTile}_{\mathrm{L}}[0], \boldsymbol{fltTile}_{\mathrm{L}}[1], \cdots, \boldsymbol{fltTile}_{\mathrm{L}}[8])), i \in [0,15],\\
&\widetilde{\boldsymbol{inTile}}_{\mathrm{L}}[i] = (\mathrm{double})(f_i^{\mathrm{in}}(\boldsymbol{inTile}_{\mathrm{L}}[0], \boldsymbol{inTile}_{\mathrm{L}}[1], \cdots, \boldsymbol{inTile}_{\mathrm{L}}[15])), i \in [0,15],\\
&\boldsymbol{outTile}_{\mathrm{L}}[i] = (\mathrm{float})(f_i^{\mathrm{out}}(\widetilde{\boldsymbol{outTile}}\mathrm{L}[0], \widetilde{\boldsymbol{outTile}}\mathrm{L}[1], \cdots, \widetilde{\boldsymbol{outTile}}\mathrm{L}[15])), i \in [0,3].
\end{aligned}
\tag{5}
$$

因此，不再需要提前准备好$16\times IC\times OC$的$\widetilde{\boldsymbol{fltTile}}_{\mathrm{L}}$、$16\times IC\times B$的$\widetilde{\boldsymbol{inTile}}_{\mathrm{L}}$和$16\times OC\times B$的$\widetilde{\boldsymbol{outTile}}_{\mathrm{L}}$，以存储核心运算的全部源数据. 此时，只需要配置$IC\times OC$的$\widetilde{\boldsymbol{fltTileBuffer}}_{\mathrm{L}}$、$IC\times B$的$\widetilde{\boldsymbol{inTileBuffer}}_{\mathrm{L}}$和$OC\times B$的$\widetilde{\boldsymbol{outTileBuffer}}_{\mathrm{L}}$作为缓冲区，紧邻*wgdGEMM*[*i*]之前完成$\boldsymbol{fltTile}_{\mathrm{L}}[i]$和$\boldsymbol{inTile}_{\mathrm{L}}[i]$的 Winograd 变换并将数据存放入$\widetilde{\boldsymbol{fltTileBuffer}}_{\mathrm{L}}$和$\widetilde{\boldsymbol{inTileBuffer}}_{\mathrm{L}}$，计算结果存放入$\widetilde{\boldsymbol{outTileBuffer}}_{\mathrm{L}}$. 然后，紧邻 *wgdGEMM*[i] 之后完成$\boldsymbol{outTile}_{\mathrm{L}}$[0]~[3]基于$\widetilde{\boldsymbol{outTileBuffer}}_{\mathrm{L}}$中结果的线性叠加. 如此以流水的形式配合 *wgdGEMM*[0]~[15] 反复搭配执行 16 次，即可得到最终 Winograd 逆变换后的输出数据. 通过上述展开的 LDM 强化使用，可以将算法 2 中使用的 LDM 空间大小降为 $220B^2$字节，仅为初始使用的 LDM 空间大小的 37.9%.

通过对 2.2.1 节中$\widetilde{\boldsymbol{inTile}}_{\mathrm{L}}$和$\boldsymbol{inTile}_{\mathrm{L}}$经合并的 Winograd 变换处理后得到的线性关系的分析，可以发现$\widetilde{\boldsymbol{inTile}}_{\mathrm{L}}$[0]~[15]的每个变换并不需要所有的$\boldsymbol{inTile}_{\mathrm{L}}[cmpt]$[0]~[15]，而只需要其中的少数几个. 因此有可能通过调整计算和访存的顺序，从而使得部分$\widetilde{\boldsymbol{inTile}}_{\mathrm{L}}$被 *wgdGEMM* 使用完后，剩余$\widetilde{\boldsymbol{inTile}}_{\mathrm{L}}$不再需要的$\boldsymbol{inTile}_{\mathrm{L}}[cmpt]$的空间得以被释放. 由此，设计了交错的 LDM 强化使用. 如图 6 所示，将原本的用于双缓冲

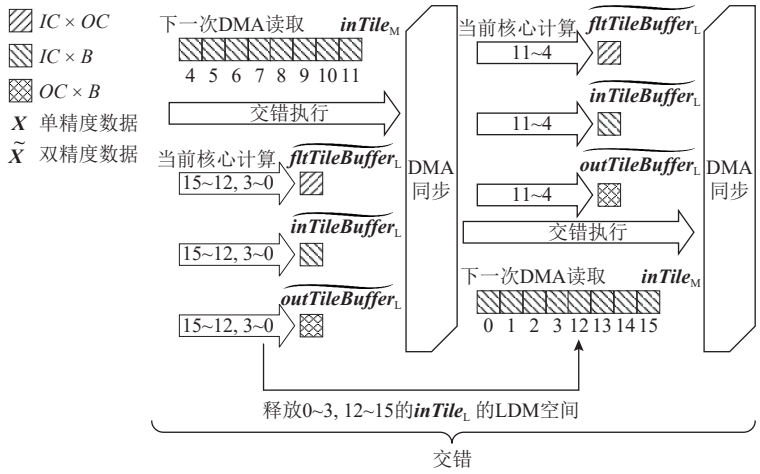


Fig. 6 Interleaving LDM enhanced usage

图 6 交错的 LDM 强化使用

输入数据块$\boldsymbol{inTile}_{\mathrm{M}}$的 DMA 读取的 2 个$\boldsymbol{inTile}_{\mathrm{L}}$[16]合并成 1 个$\boldsymbol{inTile}_{\mathrm{L}}$[24],其中$\boldsymbol{inTile}_{\mathrm{L}}$[4]~[11]和$\boldsymbol{inTile}_{\mathrm{L}}$[16]~[23]的 LDM 空间用于正常双缓冲$\boldsymbol{inTile}_{\mathrm{M}}$[4]~ [11]. 同时,将原本一轮 16 次 *wgdGEMM* 的核心运算分割成为 2 部分,前半部分核心运算为 *wgdGEMM*[0]~ [3] 以及 *wgdGEMM*[12]~ [15] 的计算,并相应地匹配$\boldsymbol{inTile}_{\mathrm{M}}$[4]~[11]的预取. 通过 2.2.1 节中合并的 Winograd 变换后的线性关系,可以得知后半部分核心运算 *wgdGEMM*[4]~[11] 的计算过程中将不再需要$\boldsymbol{inTile}_{\mathrm{L}}$[0]~[3]和$\boldsymbol{inTile}_{\mathrm{L}}$[12]~[15]的数据,因此对于后续$\boldsymbol{inTile}_{\mathrm{M}}$[0]~[3]和$\boldsymbol{inTile}_{\mathrm{M}}$[12]~[15]的预取可以直接使用前半部分核心运算释放掉的$\boldsymbol{inTile}_{\mathrm{L}}$[0]~[3]和$\boldsymbol{inTile}_{\mathrm{L}}$[12]~[15]. 由此,实现了这部分 LDM 空间在双缓冲过程中逻辑上分离但物理上共用,从而大大节省了输入数据块在双缓冲时片上存储资源的使用. 通过上述交错的 LDM 强化使用,可以将算法 2 中使用的 LDM 空间大小降为 $188B^2$ 字节,仅为初始使用的 LDM 空间大小的 32.4%.

2.2.4 输出数据块的弹性处理

在融合 Winograd 算法的优化中,都是以单个输出数据块为计算粒度设计算法,对于片上存储 LDM 的使用由 B, IC, OC 决定. 但是并非任何情况下 LDM 都能够得到充分使用,当 B, IC, OC 三者较小时,会出现大量 LDM 空间闲置的情况. 针对上述问题,设计了输出数据块的弹性处理方案:通过增大算法基于输出数据块的计算粒度,使闲置的 LDM 空间能够被使用起来,从而进一步探索融合 Winograd 算法中潜在的数据局部性.

在进行 Winograd 卷积时,移动输入数据的滑窗获取输入数据块时,2 个相邻的输入数据块之间会产生 $r-1$ 的重叠. 因此,对于以单个输出数据块为计算粒度的融合 Winograd 算法,输入数据会出现反复被读取的情况. 如图 7 所示的卷积中,对于优化前的算法,输入数据将会被平均反复读取 2.56 次. 通过引入输出数据块的弹性处理,假设算法的计算粒度由单个输出数据块变为 2 个输出数据块,那么每次只需要读取4×6的$\boldsymbol{inTile}_{\mathrm{M}}$相当于之前读取 2 次 4×4的$\boldsymbol{inTile}_{\mathrm{M}}$. 因此,优化后的算法中输入数据的平均反复读取降为了 1.92 次,相应地,输入数据的访存开销降低了 25%. 在不考虑 LDM 容量的情况下,可以设置单次计算输出数据块的数量为 $\left\lceil \frac{OH}{2} \right\rceil \left\lceil \frac{OW}{2} \right\rceil$,此时将完全消除输入数据的不必要访存,使其访存局部性达到最大.

2.2.5 定制的矩阵乘

融合 Winograd 卷积算法相比传统 Winograd 卷

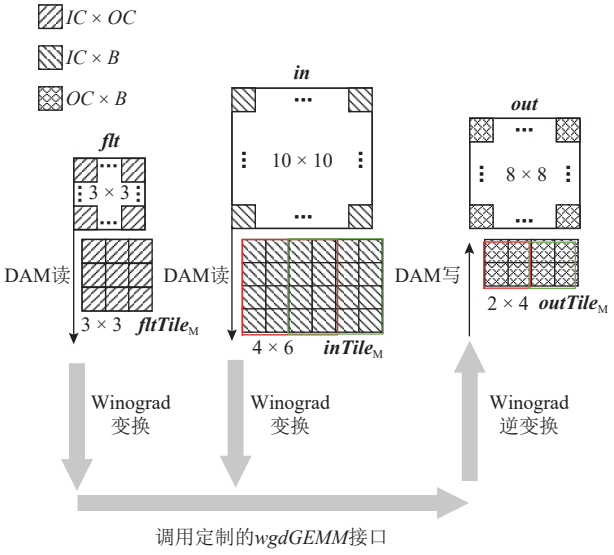


Fig. 7 Processing two output data blocks in a single run

图 7 单次处理 2 个输出数据块

积算法的一大优势就在于其不依赖于 GEMM 库接口，这使得整个 Winograd 卷积的执行过程不再是透明的，从而让更细粒度的计算访存优化成为可能. 为此，参考申威 26010 处理器上的通用 GEMM[26]，定制了匹配该算法的矩阵乘实现. 为了方便后续说明，将算法中的卷积参数映射到对应的矩阵乘参数. 其中，矩阵 $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$分别对应每次 DMA 访存的 $\boldsymbol{fltTile}_{\mathrm{M}}$, $\boldsymbol{inTile}_{\mathrm{M}}$, $\boldsymbol{outTile}_{\mathrm{M}}$数据块，矩阵乘参数 M, N, K分别对应卷积中的 OC, B, IC.

如图 8 所示，整个定制矩阵乘实现可以分为 2 部分：一部分为原始矩阵数据到单核组的任务映射；一部分为融合 Winograd 卷积算法执行过程中调用的 *wgdGEMM*. 首先，前者由 Winograd 变换、Winograd 逆变换和核组级分块构成，对于 Winograd 变换和 Winograd 逆变换过程，在 2.1 节中已经进行了详细的介绍，而核组级分块直接采用文献 [26] 中的分块原理，详细过程可以参考文献 [26] 中的研究工作. 其次，需要注意 Winograd 转换和 Winograd 逆变换的线程级并行同 *wgdGEMM* 的线程级并行间是相对应的. 以 $\boldsymbol{fltTile}_{\mathrm{M}}$为例，其在 *wgdGEMM* 中的对应矩阵 $\boldsymbol{A}$在进行线程级任务划分时，是通过对 $K_{\mathrm{cg}}\times M_{\mathrm{cg}}$矩阵块进行8×8网格划分从而实现单个从核的任务映射. 同理，$\boldsymbol{fltTile}_{\mathrm{M}}$也将通过对3×3个 $IC_{\mathrm{cg}}\times OC_{\mathrm{cg}}$（$IC_{\mathrm{cg}}=K_{\mathrm{cg}}$, $OC_{\mathrm{cg}}=M_{\mathrm{cg}}$）数据块的8×8网格划分以实现卷积核的 Winograd 变换过程的线程级并行. 类似地，$\boldsymbol{inTile}_{\mathrm{M}}$需要通对4×4个 $IC_{\mathrm{cg}}\times B_{\mathrm{cg}}$（$IC_{\mathrm{cg}}=K_{\mathrm{cg}}$, $B_{\mathrm{cg}}=N_{\mathrm{cg}}$）数据块的8×8网格划分以实现输入的 Winograd 变换过程的线程级并行，$\boldsymbol{outTile}_{\mathrm{M}}$需要通过对2×2个 $OC_{\mathrm{cg}}\times B_{\mathrm{cg}}$（$OC_{\mathrm{cg}}=M_{\mathrm{cg}}$, $B_{\mathrm{cg}}=N_{\mathrm{cg}}$）数据块的8×8网格划分以实现输出的 Winograd 逆变换过程的线程级并行. 最后，将结合融合 Winograd 卷积算法中矩阵乘与文献 [26] 的 2 点关键不同之处对 *wgdGEMM* 的实现进行详细介绍：1）不同一. 矩阵乘不再是非转置的矩阵乘，而是矩阵 $\boldsymbol{A}$转置的矩阵乘；2）不同二. M, N, K通常情况下小于 1 000[27].

将 16 个 *wgdGEMM* 绑定到一起作为融合 Winograd

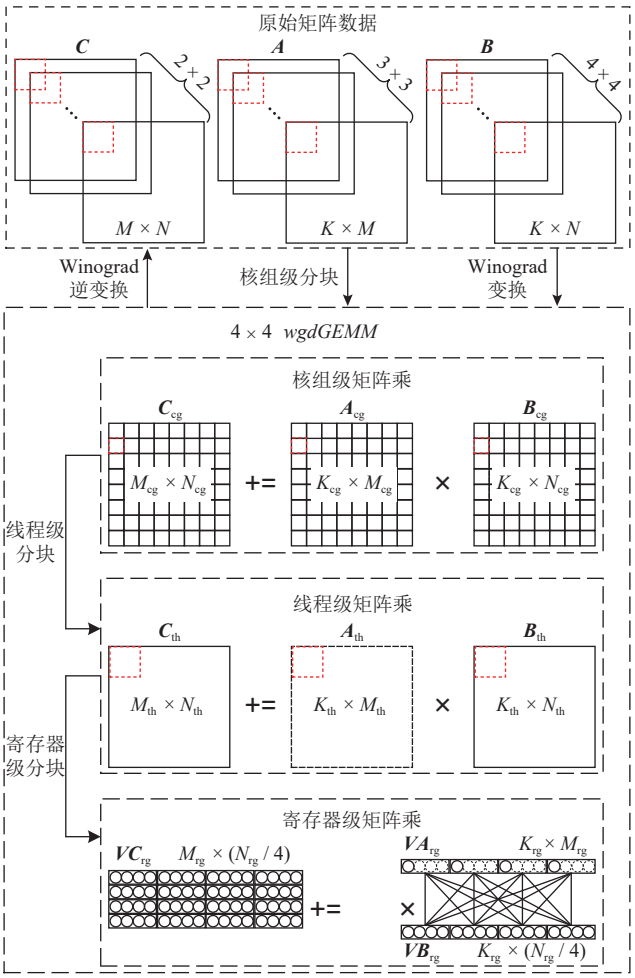


Fig. 8 Customized matrix multiplication implementation

图 8 定制的矩阵乘实现

卷积算法的单位计算粒度. 其中, 单个 *wgdGEMM* 可以分为 3 个层次, 分别为核组级矩阵乘、线程级矩阵乘和寄存器级矩阵乘. 对于核组级矩阵乘, $\boldsymbol{A}_{cg}$, $\boldsymbol{B}_{cg}$, $\boldsymbol{C}_{cg}$分别对应算法中存储双精度数据的 LDM 缓冲区 $\widetilde{fltTileBuffer}_{L}$, $\widetilde{inTileBuffer}_{L}$, $\widetilde{outTileBuffer}_{L}$. 当 LDM 空间足够的情况下, M_{cg}, N_{cg}, K_{cg}分别与 M, N, K相等. 然后, 基于单个核组上的8×8从核阵列, 对 $\boldsymbol{A}_{cg}$, $\boldsymbol{B}_{cg}$, $\boldsymbol{C}_{cg}$进行8×8网格形式的线程级分块, 从而将单核组的矩阵乘映射到单线程的矩阵乘. 为了满足$\boldsymbol{C}_{th}$的计算需求, 需要通过广播-广播的寄存器通信方案[26], 分别对 $\boldsymbol{A}_{th}$和$\boldsymbol{B}_{th}$进行广播. 在对从核进行 $\boldsymbol{A}_{th}$, $\boldsymbol{B}_{th}$, $\boldsymbol{C}_{th}$数据分配时, 考虑到“不同一”, 定制矩阵乘不再采用单一的行对行映射, 而是按照行对行映射的方式为每个从核分配$\boldsymbol{B}_{th}$和$\boldsymbol{C}_{th}$, 按照行对列映射的方式为每个从核分配$\boldsymbol{A}_{th}$. 由此, 可得第i行第j列的从核将分配到$\boldsymbol{A}_{cg}[j][i]$, $\boldsymbol{B}_{cg}[i][j]$, $\boldsymbol{C}_{cg}[i][j]$. 通过这种混合映射方法, 使得在广播-广播的寄存器通信中, 分别对$\boldsymbol{A}_{th}$进行行广播, 对$\boldsymbol{B}_{th}$进行列广播, 从而使得每个从核上寄存器通信缓冲区资源能够得到充分利用. 为了能够尽可能发挥 *wgdGEMM* 的指令级并行效率, 通过寄存器级分块进一步将线程级矩阵乘划分为更细粒度的寄存器级矩阵乘, 从而方便最底层核心指令序列的手动重排. 考虑到“不同一”中可以先对$\boldsymbol{A}_{th}$进行转置, 将其维度从$K_{th}\times M_{th}$变为$M_{th}\times K_{th}$, 再直接运用文献 [26] 中的方法. 但是我们更希望能够避免这一转置带来的额外访存开销, 为此, 基于申威 26010 处理器单精度向量长度为 4 这一特性, 设计了如图 8 所示的寄存器级矩阵乘映射方案: 1)通过 vldd 指令依次装入$\boldsymbol{C}_{th}$中元素, 每次 4 个元素, 得到一个$\boldsymbol{VC}_{rg}[M_{rg}][N_{rg}/4]$的向量数组; 2)通过 ldder 指令依次装入$\boldsymbol{A}_{th}$的单个元素进行向量扩展, 得到一个$\boldsymbol{VA}_{rg}[K_{rg}][M_{rg}]$的向量数组, 并行广播; 3)通过 vldc 指令依次装入$\boldsymbol{B}_{th}$中元素, 每次 4 个元素, 得到一个$\boldsymbol{VB}_{rg}[K_{rg}][N_{rg}/4]$的向量数组, 并列广播; 4)通过 vmad 指令对$\boldsymbol{VA}_{rg}$和$\boldsymbol{VB}_{rg}$的所有向量数据进行全相连的乘法运算, 并同$\boldsymbol{VC}_{rg}$中对应向量元素进行累加运算$VC_{rg}[i][j]+=\sum_{0}^{K_{rg}-1}VA_{rg}[k][i]\times VB_{rg}[k][j]$; 5)通过 vstd 指令将计算结果写入$\boldsymbol{C}_{th}$的对应位置. 其中, 每执行一次方案 1 和方案 5, 相应的方案 2~4 将会被执行$\lceil K_{th}/K_{rg}\rceil$次, 不难看出方案 2~4 组成了寄存器级矩阵乘的核心指令序列. 为了保证寄存器级矩阵乘的运算效率, 取$K_{rg}=1$使得累加运算过程不存在数据依赖. 同时, 考虑到申威 26010 处理器除去零寄存器和栈指针(stack pointer, SP)寄存器能够自由使用的向量寄存器数为 30, 所以有

$$M_{rg}+\frac{N_{rg}}{4}+M_{rg}\frac{N_{rg}}{4}\leqslant 30. \tag{6}$$

再考虑整个线程级矩阵乘的计算访存比, 所以有

$$\frac{2M_{th}N_{th}K_{th}}{4M_{th}K_{th}\dfrac{N_{th}}{N_{rg}}+K_{th}N_{th}\dfrac{M_{th}}{M_{rg}}+2M_{th}N_{th}}\approx\frac{2}{\dfrac{4}{N_{rg}}+\dfrac{1}{M_{rg}}}. \tag{7}$$

在保证涉及到的每个向量元素能够匹配 1 个独立向量寄存器的情况下, 为了最大化计算访存比, 减少不必要的访存开销, 结合式(6)(7), 可得$M_{rg}=\dfrac{N_{rg}}{4}=4$. 因此, 对于寄存器级矩阵乘, 分配 4 个向量寄存器存储$\boldsymbol{VA}_{rg}$、4 个向量寄存器存储$\boldsymbol{VB}_{rg}$和 16 个向量寄存器存储$\boldsymbol{VC}_{rg}$.

申威 26010 处理器每个从核支持 2 条不同的流水线 P0 和 P1, 其中 P0 支持浮点和整数的标量/向量操作, 而 P1 支持数据迁移、比较、跳转和整数标量操作. 为了能够获得高性能的 *wgdGEMM*, 可以通过手写汇编保证寄存器级矩阵乘核心指令序列尽可能精简, 同时手动重排指令序列以充分发挥从核的双流水指令运行机制. 通过对上述寄存器级矩阵乘的描述, 可以得到理想情况下向量寄存器的分配. 其中, 4 个向量寄存器用于载入$\boldsymbol{VA}_{rg}$数据, 标记为*AR*0~*AR*3; 4 个向量寄存器用于载入$\boldsymbol{VB}_{rg}$数据, 标记为*BR*0~*BR*3; 16 个向量寄存器用于存储$\boldsymbol{VC}_{rg}$数据, 标记为*CR*00~*CR*33. 由此, 可以得出寄存器级矩阵乘最内层循环核心指令序列如图 9(a)左侧所示, 执行时间为 25 个时钟周期, 此时整个执行过程几乎处于单流水状态. 为了能够真正启动从核的双流水模式, 对初始指令序列手动重排, 重排后如图 9(a)右侧所示. 在最内层循环开始前, 首先预取第 1 次计算所需的 *AR*0, *AR*1, *BR*0, *BR*1, 然后在最内层循环指令序列中实现当前循环的计算前部分和访存后部分的双流水, 以及当前循环的计算后部分和下一轮循环的访存前部分的双流水, 最终优化后的指令序列的执行时间为 16 个时钟周期, 性能提升约 56.25%. 此时, 虽然从核双流水的性能得以充分发挥, 但是$M_{rg}=4$且$N_{rg}=16$, 却要求矩阵乘中M是 32 的倍数且N是 128 的倍数. 虽然当M和N不满足此要求时可以使用填充的方式先满足倍数要求再进行运算, 但是考虑到“不同二”, 这种开销往往是不可忽视的. 例如, 当 $M=64$, $N=64$, $K=128$时, 依照图 9(a)中实现, 需要额外 1 倍的计算和访存, 这些不必要的计算和访存将极大地降低算法的性能.

为了尽可能缓解这一问题, 依照$M_{rg}=4$且$N_{rg}=16$时指令重排的思想, 对$M_{rg}\in\{1,2,3,4\}$和$N_{rg}\in\{4,8,12,$

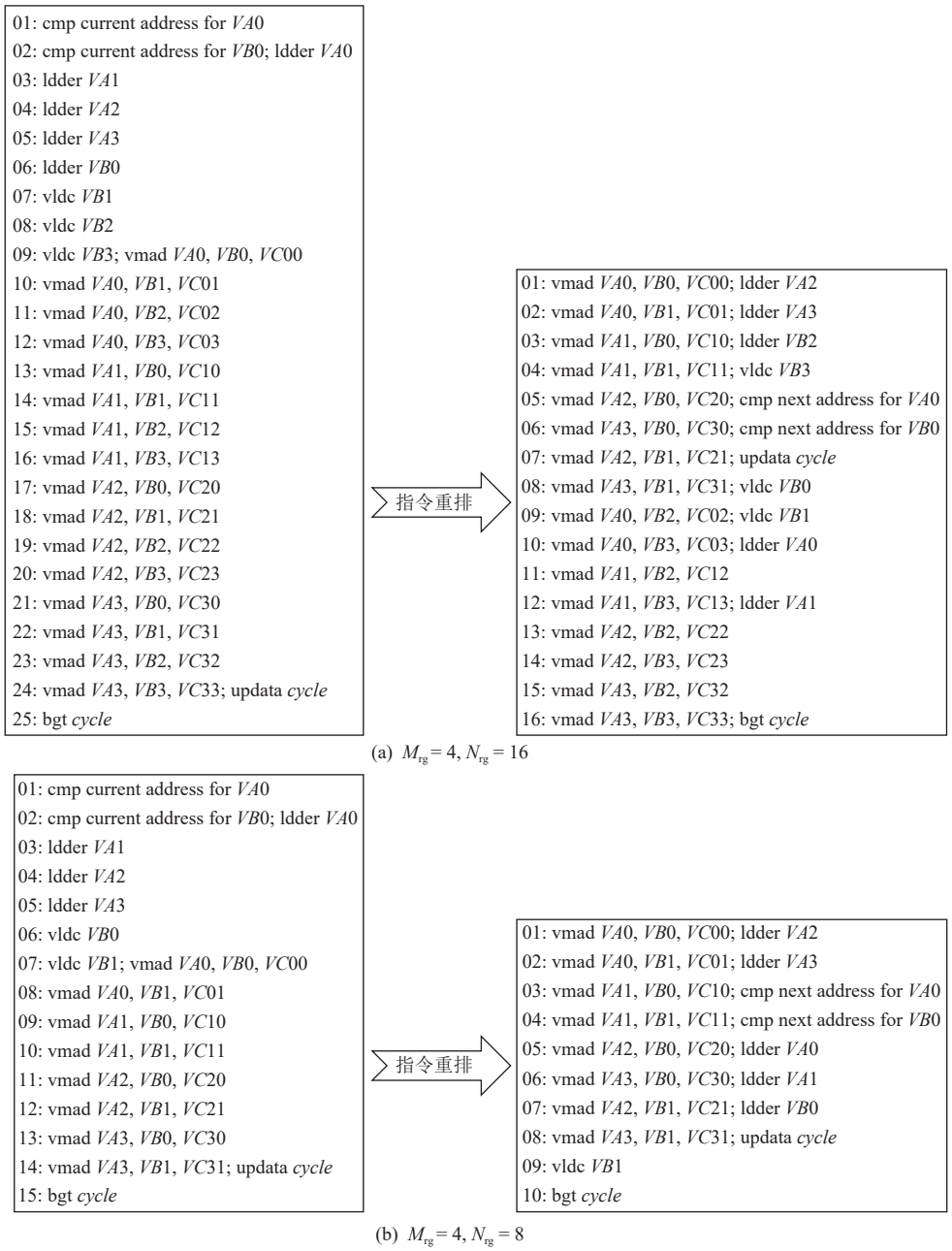


Fig. 9 Instruction reordering

图 9 指令重排

16)组成的 16 种情况中余下的 15 种情况的实现分别进行了指令重排，如图 9(b)所示 $M_{rg}=4$ 且 $N_{rg}=8$ 时的情况. 为了配合这一实现，将 M_{th} 分为 2 部分 $[0, M_{th}-\mathrm{mod}(M_{th},4))$ 和 $[M_{th}-\mathrm{mod}(M_{th},4), M_{th})$. 类似地，$N_{th}$ 也被分为 $[0, N_{th}-\mathrm{mod}(N_{th},16))$ 和 $[N_{th}-\mathrm{mod}(N_{th},16), N_{th})$. 通过 M_{th} 的 2 部分和 N_{th} 的 2 部分的一一对应关系可以将寄存器级矩阵乘分为 4 个部分. 此时，对于 $M=64$, $N=64$, $K=128$ 的情况则可以直接运算，不需要任何多余的计算和访存开销.

2.3 通用性分析

本文研究工作虽然是面向国产申威 26010 处理器探索并行卷积算法的高性能实现，但是仍然对其他众核处理器硬件平台具有一定的借鉴意义. 如 2.1 节中 Winograd 变换、核心运算和 Winograd 逆变换融合执行以尽可能减少分离执行造成的高额访存开销的思想，2.2.1 节中合并 Winograd 变换过程并向量化以避免存储中间数据带来的额外片上存储资源的开销并降低变换过程中的计算量，以及 2.2.4 节中弹性处理输出数据块的数量以充分利用片上存储资源，这些优化方案都是脱离硬件平台特征的，可以直接应用于其他众核处理器. 除此之外，其他的优化方案，比如 DMA 双缓冲、片上存储的强化使用、定制矩阵

乘实现这些工作则是同申威 26010 处理器架构特征紧密联系的, 虽然无法直接应用于其他众核处理器平台, 但是对于某些类似架构特征的硬件平台仍然可以提供一些参考价值.

如上所述, Winograd 融合卷积算法可以分为与架构无关的优化方案和与架构相关的优化方案 2 个部分, 对于新一代的申威 26010Pro 处理器[28-29], 主要关注于架构相关的优化方案的适用性. 而架构相关的优化方案主要是 DMA 双缓冲、片上存储的强化使用、定制矩阵乘实现, 其中申威 26010Pro 处理器对于 DMA 双缓冲是依旧支持的; 片上存储 LDM 则由原来的 64 KB 增加到了 256 KB, 更多的片上存储资源更有利于降低主存的访问频率; 定制矩阵乘的实现中, 最大变化来自于寄存器通信机制的取消和 SIMD 指令向量长度的增加, 但是仍然具备对单核组从核间的数据通信的支持, 也就是新的 RMA (remote memory access), 向量长度的增加则更有利于浮点性能的发挥. 因此, 综合上述分析不难看出, 融合 Winograd 卷积算法对于申威 26010Pro 处理器同样适用.

3 实验结果与分析

申威 26010 处理器的配置如表 1 所示.

本次实验全部在申威 26010 处理器单核组上进行, 因为跨不同核组的并行通常由更高编程级别的用户自己处理[26,30]. 实验中将本文提出的不依赖官方 GEMM 库接口的融合 Winograd 卷积算法标记为 fusedWgdConv, 类似地, 将依赖官方 GEMM 库接口的传统 Winograd 卷积算法作为基准测试对象, 并标记为 simpleWgdConv. 从 4 个不同的角度设计实验, 从而充分展示研究工作的成果和价值. 1)通过对不同优化方案的累加设计实验, 测试各个优化方案的效果; 2)抽取典型卷积神经网络模型 AlexNet, GoogleNet, ResNet,VGG 中的常见卷积层, 测试评估 fusedWgdConv 在实际应用场景中的性能; 3)基于 fusedWgdConv, 在深度学习框架 Caffe 中测试不同批大小下 VGG 网络模型的卷积性能提升. 4)对 fusedWgdConv 调用的 *wgdGEMM* 的效果进行了测试. 为了保证测试结果的精确性, 所有测试均迭代 10 次, 去掉 1 个最优值和 1 个最差值, 取剩余 8 个测试结果的平均值.

Table 1 Configure Parameters for ShenWei-26010 Processor

表 1 申威 26010 处理器配置参数

规格	数值
频率/GHz	1.45
核数	260
主存容量/GB	32
主存带宽/GBps	144
TDP/W	250
单精度峰值性能/ TFLOPS	>3
双精度峰值性能/ TFLOPS	>3

3.1 优化效果测试

选取 VGG 中的卷积作为测试实例, 并以 simpleWgdConv 作为测试基准, 测试并记录 fusedWgdConv 在不同优化方案累加下相比于 simpleWgdConv 的性能加速.

通过对不同优化方法的累加, 可以得到 7 个版本的 fusedWgdConv, 分别为: 1)INIT, 融合 Winograd 变

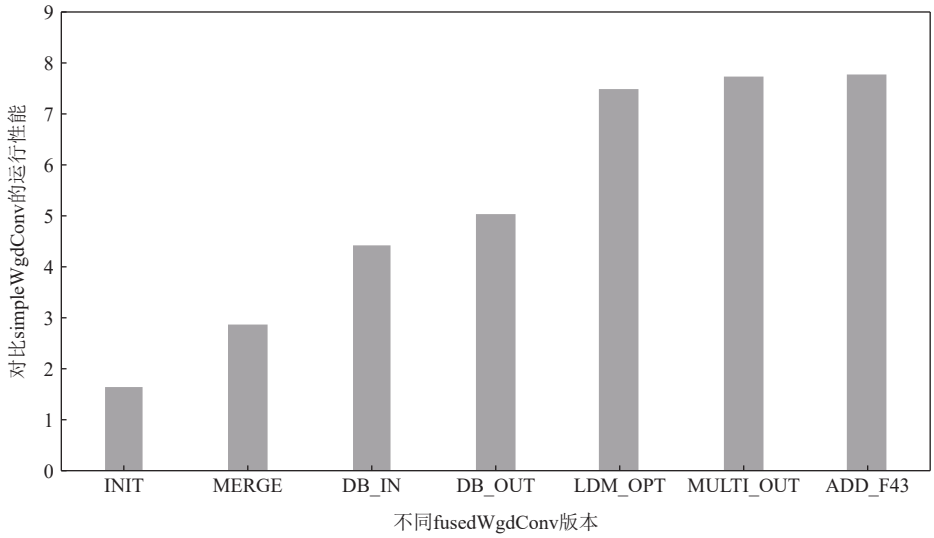


Fig. 10 Performance comparison of different fusedWgdConv versions and simpleWgdConv

图 10 不同 fusedWgdConv 版本和 simpleWgdConv 的性能对比

换、核心运算和 Winograd 逆变换的执行过程，初步实现 fusedWgdConv；2）MERGE，在 INIT 版本的基础上引入合并的 Winograd 变换模式；3）DB_IN，在 MERGE 版本的基础上，双缓冲实现***in***的 DMA 访存和核心运算的并行；4）DB_OUT，在 DB_IN 版本的基础上，双缓冲实现***out***的 DMA 访存和核心运算的并行；5）LDM_OPT，在 DB_OUT 的基础上，引入展开的 LDM 强化使用和交错的 LDM 强化使用，缓解有限片上存储资源的使用压力，降低算法的片外访存开销；6）MULTI_OUT，在 LDM_OPT 的基础上，考虑小规模卷积时 LDM 大量闲置的情况，引入输出数据块的弹性处理机制，使得任何情况下算法都能充分利用片上存储资源；7）ADD_F43，通过添加 $F(4\times4,3\times3)$的实现，探索不同输出数据块大小的影响.

如图 10 所示，初步实现了 fusedWgdConv 的 INIT 版本相比 simpleWgdConv 有 67% 的性能提升，可见 INIT 的基本设计思路能够明显降低 simpleWgdConv 依赖官方 GEMM 库接口造成的高额访存开销. 依据 Winograd 变换系数能够提前确定这一特征，将原本分离的 Winograd 变换模式替换为合并的 Winograd 变换模式，同时引入向量化，不仅消除了额外的 LDM 使用，而且大大降低了 Winograd 变换过程中的计算量，基于此的 MERGE 版本性能是 simpleWgdConv 的 2.9 倍. 申威 26010 处理器支持 DMA 的异步执行，通过对 fusedWgdConv 的精心设计实现了***in***和***out***的 DMA 访存同核心运算的并行运行，使得大部分的片外访存开销得以被计算掩藏. 基于此的 DB_IN 版本相比 MERGE 版本性能提升了 52.76%. 进一步优化后的 DB_OUT 版本相比 MERGE 版本性能提升了 74.14%. 最终双缓冲使得 fusedWgdConv 运行性能达到了 simpleWgdConv 的 5.05 倍. 有限的片上存储资源限制了 fusedWgdConv 的性能提升，为了能够尽可能提升片上存储的使用效率，设计了展开的 LDM 强化使用以减少数据类型变换带来的额外 LDM 消耗，以及交错的 LDM 强化使用降低$\boldsymbol{inTile}_{\mathrm{L}}$双缓冲的 LDM 需求，由此 LDM_OPT 版本性能相比 DB_OUT 版本进一步提升了 48.51%. MULTI_OUT 版本中的输出数据块弹性处理仅对小规模卷积有作用，因此对算法性能提升并不明显，其最终性能是 simpleWgdConv 的 7.75 倍. ADD_F43 版本相比于 MULTI_OUT 版本的性能提升非常微弱，主要原因在于 fusedWgdConv 单核组每次至少要处理 1 个输出数据块所涉及到的 Winograd 变换、Winograd 逆变换和核心运算，造成片上存储资源的开销很高. 当算法由 $F(2\times2,3\times3)$扩展到 $F(4\times4,3\times3)$时，单个输出数据块的规模增加了 4 倍，大大增加了对片上存储资源的需求，而申威 26010 处理器有限的片上存储资源不足以很好地支撑 fusedWgdConv 的进一步扩展.

综合来看，fusedWgdConv 相比 simpleWgdConv 对于申威 26010 处理器上的卷积有显著的性能提升.

3.2 卷积性能测试

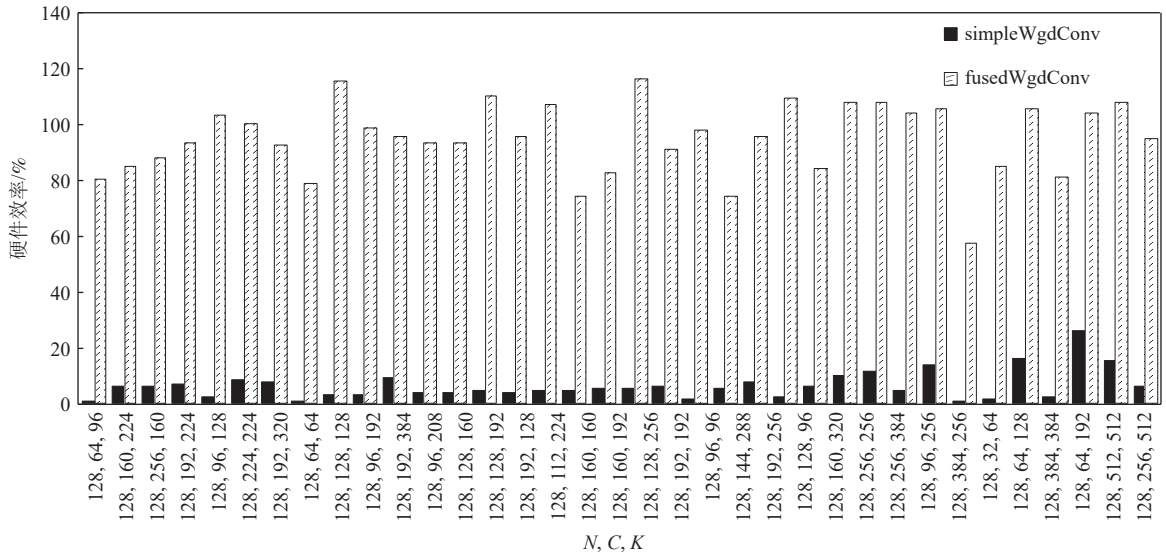
为了能够进一步验证 fusedWgdConv 对常用卷积是否具有现实意义，抽取典型卷积神经网络模型 AlexNet, GoogleNet, RestNet,VGG 中卷积层的卷积参数，并测试对比 fusedWgdConv 和 simpleWgdConv 的卷积性能. 一般来说，卷积性能测试时的测试标准是运行时的性能，但是为了能够更好地观察算法对硬件处理器的使用效果，实验选择运行时相对理论峰值性能的百分比作为测试标准，计算公式为硬件效率$=\dfrac{\text{运行时性能}}{\text{理论峰值性能}}$.

如图 11 所示，将抽取的卷积按照计算量由小到大排列，同时对 2 种常见的卷积形式进行测试分析：1）填充为 0 且跨步为 1；2）填充为 1 且跨步为 1. 从总体卷积测试来看，fusedWgdConv 在无填充和有填充的情况下平均硬件效率分别为 95.08% 和 91.2%，分别是 simpleWgdConv 下卷积性能的 14.54 倍和 14.52 倍. 可以看出，fusedWgdConv 的卷积能够很好地适应现实中的卷积场景. 从单个卷积测试来看，无填充时，fusedWgdConv 性能最小时是 simpleWgdConv 性能的 3.02 倍，最大时可以达到 62.81 倍；有填充时，fusedWgdConv 相比 simpleWgdConv，卷积性能与无填充时差不多，在 3.01~62.72 倍不等. 其中，无填充和有填充情况下，fusedWgdConv 的最高硬件效率可以分别达到 116.21% 和 111.41%. 可以看出，fusedWgdConv 能够很好地发挥申威 26010 处理器的硬件性能，且相比 simpleWgdConv 有明显的性能提升.

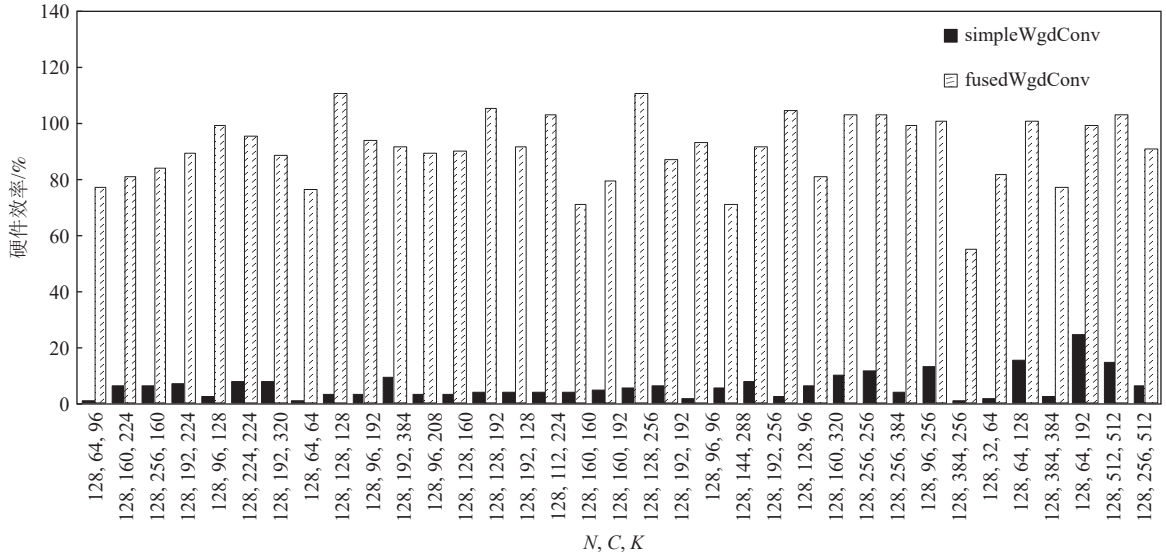
3.3 卷积神经网络性能测试

为了测试评估本文工作在实际使用中的效果，以原始版本的 Caffe 为基准，基于 fusedWgdConv 测试不同批大小下 VGG 模型的卷积性能提升. 同时，考虑到 Caffe 中卷积的数据格式为 $N-C-H-W$，fusedWgdConv 的数据格式为 $H-W-C-N$，标记 Caffe-FWC 和 Caffe-FWC-DFT 这 2 个版本的 Caffe，分别表示基于 fusedWgdConv 的 Caffe 和基于 fusedWgdConv 且进行数据格式转换的 Caffe.

如图 12 所示，Caffe-FWC 的性能加速比在 3.4~9.8 之间. 而 Caffe-FWC-DFT 虽然仍能带来可观的性能提升，但是数据格式转换大约占了整个卷积执行过程的 40.5%，因此是不容忽视的. 但是数据转换过程



(a) 填充为0且跨步为1



(b) 填充为1且跨步为1

Fig. 11 Hardware efficiency of ShenWei-26010 processor for different convoltuions

图 11 不同卷积下申威 26010 处理器的硬件效率

完全是可以避免的，只需要将卷积神经网络模型中的所有网络层格式设计成$H-W-C-N$即可. 此时在实际使用中仅需对样本数据进行 1 次的$N-C-H-W$到$H-W-C-N$的数据格式转换即可，其代价相对于后续成千上万次的网络模型的循环迭代是非常微小的. 这也是我们目前正在致力于的整个 DNN 库的研究工作，具体内容会在以后的文章中进行详细阐述.

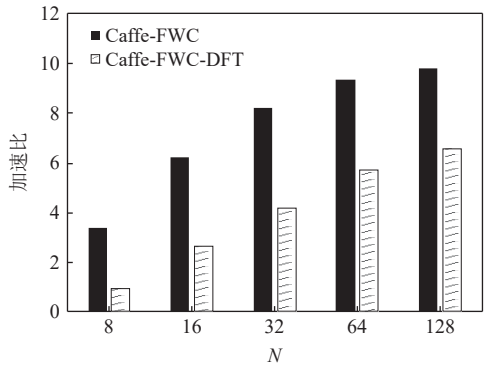


Fig. 12 Performance speedup of Caffe with different versions

图 12 不同版本 Caffe 的性能加速比

3.4 定制矩阵乘效果测试

fusedWgdConv 的关键点之一就是与其匹配的定制矩阵乘，如 2.2.5 节所示，*wgdGEMM* 在申威 26010 处理器现有的通用 GEMM[26] 的基础上，针对 2 点不同进行了设计. 针对“不同一”，设计了混合映射的广播-广播的寄存器通信以及避免转置开销的寄存器

级矩阵乘. 相应的，针对”不同二”，对 $M_{rg} \in \{1,2,3,4\}$ 和 $N_{rg} \in \{4,8,12,16\}$ 组成的 16 种情况下寄存器级矩阵乘的核心指令序列都进行了手写汇编和指令重排，而文献 [26] 只考虑了 $M_{rg}=4$ 且 $N_{rg}=16$ 的情况.

实验选择 8 组矩阵乘场景作为测试对象，其中每组矩阵乘满足 2 点：一是矩阵 $\boldsymbol{A}$ 转置；二矩阵的维度不同时满足 $M_{rg}=4$ 和 $N_{rg}=16$. 同时，选取申威 26010 处理器现有的 GEMM[26] 作为基准，对比测试 *wgdGEMM* 的效果. 假设 GEMM 同 *wgdGEMM* 一样所需数据已存在于 LDM 中，如表 2 所示为两者的运行时间. 其中，*wgdGEMM* 相对 GEMM 性能加速比为 0.202 3~0.278 8 不等，平均性能加速比约为 0.239 9. 由此可见，针对 fusedWgdConv 定制矩阵乘是有必要的，且 *wgdGEMM* 相比 GEMM 有明显的性能提升.

Table 2 Running Time for *wgdGEMM* and GEMM

表 2 *wgdGEMM* 和 GEMM 的运行时间

M	*N*	*K*	运行时间/ms		加速比
			wgdGEMM	GEMM	
32	64	32	0.11	0.13	0.203 7
64	64	64	0.23	0.29	0.278 8
128	64	128	0.68	0.82	0.202 3
256	64	256	2.17	2.75	0.266 6
512	64	512	8.03	10.06	0.252 1
1 024	64	1 024	32.68	40.58	0.241 8
2 048	64	2 048	128.05	158.48	0.237 7
4 096	64	4 096	512.74	633.80	0.236 1

4 总结展望

随着我国自主研发的申威 26010 众核处理器在人工智能领域的快速发展，对该处理器上高性能卷积算法的实现也提出了更高的要求. 而该处理器上卷积算法现有的研究尚处于初级阶段，本文针对这一问题，结合申威 26010 处理器的架构特征，提出并实现了一种高性能并行的融合 Winograd 卷积算法. 该算法有 2 个主要特点：1）不依赖于官方 GEMM 库接口，设计了匹配该算法的定制矩阵乘，并结合现实卷积计算特征和通用矩阵乘算法，在零开销情况下完成了矩阵转置，并对其寄存器级矩阵乘的核心指令序列进行了各种情况的指令重排；2）Winograd 变换、核心运算和 Winograd 逆变换过程的融合优化避免了三者分开执行时造成的反复读取主存数据带来的高额访存开销. 此外，还设计了合并的 Winograd 变换模式加速 Winograd 变换过程；DMA 双缓冲实现从核阵列上计算和访存的并行；展开和交错的 LDM 强化使用方法以提高有限片上存储资源的使用效率；输出数据块的弹性处理避免小规模卷积下片上存储资源的浪费，通过这些优化保证了融合 Winograd 卷积算法在申威 26010 处理器上的高性能并行. 在实验分析中，以依赖官方 GEMM 库接口的传统 Winograd 卷积算法为基准，从优化效果和卷积性能 2 个方面进行测试分析，证明了融合 Winograd 卷积算法不仅有远高于传统 Winograd 卷积算法的性能，而且能够很好地应用于现实卷积场景.

未来将对本文工作进行扩展，结合其他网络层优化设计以及主流深度学习框架，进一步探索申威 26010 众核处理器上深度学习的并行优化.

作者贡献声明：武铮提出算法思路，设计并实现具体的优化方案，撰写并修改论文；金旭参与算法实现、实验设计以及数据分析；安虹负责指导论文撰写.

参 考 文 献

[1] Khan S, Rahmani H, Shah S A A, et al. A guide to convolutional neural networks for computer vision[J]. Synthesis Lectures on Computer Vision, 2018, 8(1): 1−207

[2] Abdel-Hamid O, Mohamed A, Hui Jiang, et al. Convolutional neural networks for speech recognition[J]. IEEE/ACM Transactions on Audio, Speech, and Language Processing, 2014, 22(10): 1533−1545

[3] Ouyang Zhenchao, Niu Jianwei, Liu Yu, et al. Deep CNN-based real-time traffic light detector for self-driving vehicles[J]. IEEE Transactions on Mobile Computing, 2019, 19(2): 300−313

[4] Litjens G, Kooi T, Bejnordi B E, et al. A survey on deep learning in medical image analysis[J]. Medical Image Analysis, 2017, 42(9): 60−88

[5] Zhang Yi, Shu Bing, Yin Yan, et al. Efficient processing of convolutional neural networks on SW26010 [C] //Proc of the 16th IFIP Int Conf on Network and Parallel Computing. Berlin: Springer, 2019: 316−321

[6] Xu Rui, Ma Sheng, Guo Yang. Performance analysis of different convolution algorithms in GPU environment [C] //Proc of the 13th IEEE Int Conf on Networking, Architecture and Storage. Piscataway, NJ: IEEE, 2018: 45−54

[7] San Juan P, Castelló A, Dolz M F, et al. High performance and portable convolution operators for multicore processors [C] //Proc of the 32nd IEEE Int Symp on Computer Architecture and High Performance Computing. Piscataway, NJ: IEEE, 2020: 91−98

[8] Fang Jiarui, Fu Haohuan, Zhao Wenlai, et al. swdnn: A library for accelerating deep learning applications on Sunway Taihulight [C] // Proc of the 31st IEEE Int Parallel and Distributed Processing Symp. Piscataway, NJ: IEEE, 2017: 615−624

[9] Nguyen-Thanh N, Le-Duc H, Ta D T, et al. Energy efficient techniques using FFT for deep convolutional neural networks [C] // Proc of the 9th of Int Conf on Advanced Technologies for Communications. Piscataway, NJ: IEEE, 2016: 231−236

[10] Lavin A, Gray S. Fast algorithms for convolutional neural networks [C] //Proc of the 29th IEEE Conf on Computer Vision and Pattern

Recognition. Piscataway, NJ: IEEE, 2016: 4013−4021
[11] Park H, Kim D, Ahn J, et al. Zero and data reuse-aware fast convolution for deep neural networks on GPU [C] //Proc of the 11th IEEE/ACM/IFIP Int Conf on Hardware/Software Codesign and System Synthesis. Piscataway, NJ: IEEE, 2016: 271−280
[12] Jia Zhen, Zlateski A, Durand F, et al. Optimizing *N*-dimensional, Winograd-based convolution for manycore CPUs [C] //Proc of the 23rd ACM SIGPLAN Symp on Principles and Practice of Parallel Programming. New York: ACM, 2018: 109−123
[13] 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): 825−835 (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): 825−835)
[14] Mazaheri A, Beringer T, Moskewicz M, et al. Accelerating Winograd convolutions using symbolic computation and meta-programming [C] //Proc of the 15th European Conf on Computer Systems. New York: ACM, 2020: 616−629
[15] Jia Liancheng, Liang Yun, Li Xiuhong, et al. Enabling efficient fast convolution algorithms on GPUs via MegaKernels[J]. IEEE Transactions on Computers, 2020, 69(7): 986−997
[16] Castro R L, Andrade D, Fraguela B B. OpenCNN: A Winograd minimal filtering algorithm implementation in CUDA[J]. Mathematics, 2021, 9(17): 1−19
[17] Wang Qinglin, Li Dongsheng, Mei Songzhu, et al. Optimizing Winograd-based fast convolution algorithm on Phytium multi-core CPUs[J]. Journal of Computer Research and Development, 2020, 57(6): 1140−1151 (in Chinese)
(王庆林, 李东升, 梅松竹, 等. 面向飞腾多核处理器的 Winograd 快速卷积算法优化 [J]. 计算机研究与发展, 2020, 57(6): 1140−1151)
[18] Meng Jintao, Zhuang Chen, Chen Peng, et al. Automatic generation of high-performance convolution kernels on ARM CPUs for deep learning[J]. IEEE Transactions on Parallel and Distributed Systems, 2022, 33(11): 2885−2899
[19] Lu Liqiang, Liang Yun. SpWA: An efficient sparse winograd convolutional neural networks accelerator on FPGAs [C/OL] //Proc of the 55th Annual Design Automation Conf. Piscataway, NJ: IEEE, 2018[2022-08-21].https://dl.acm.org/doi/abs/10.1145/3195970.3196120
[20] Zhao Wenlai, Fu Haohuan, Fang Jiarui, et al. Optimizing convolutional neural networks on the Sunway Taihulight supercomputer[J]. ACM Transactions on Architecture and Code Optimization, 2018, 15(1): 310−335
[21] Fu Haohuan, Liao Junfeng, Yang Jinzhe, et al. The Sunway TaihuLight supercomputer: System and applications[J]. Science China Information Sciences, 2016, 59(7): 1−16
[22] Lin J, Xu Zhigeng, Cai Linjin, et al. Evaluating the SW26010 many-core processor with a micro-benchmark suite for performance optimizations[J]. Parallel Computing, 2018, 77(3): 128−143
[23] Xu Zhigeng, Lin J, Matsuoka S. Benchmarking SW26010 many-core processor [C] //Proc of the 31st IEEE Int Parallel and Distributed Processing Symp Workshops. Piscataway, NJ: IEEE, 2017: 743−752
[24] Intel. Intel Xeon Phi processor [EB/OL]. [2022-08-21]. https://software.intel.com/en-us/xeon-phi/x200-processor
[25] NVIDIA. NVIDIA TESLA V100 [EB/OL]. [2022-08-21]. https://www.nvidia.com/en-gb/data-center/tesla-v100/
[26] Wu Zheng, Li Mingfan, Chi Mengxian, et al. Runtime adaptive matrix multiplication for the SW26010 many-core processor[J]. IEEE Access, 2020, 8: 156915−156928
[27] Li Xiuhong, Liang Yun, Yan Shengen, et al. A coordinated tiling and batching framework for efficient GEMM on GPUs [C] //Proc of the 24th Symp on Principles and Practice of Parallel Programming. New York: ACM, 2019: 229−241
[28] Li Fang, Liu Xin, Liu Yong, et al. SW_Qsim: A minimize-memory quantum simulator with high-performance on a new sunway supercomputer [C/OL] //Proc of the 33rd Int Conf for High Performance Computing, Networking, Storage and Analysis. New York: ACM, 2021[2022-08-21].https://dl.acm.org/doi/10.1145/3458817.3476161
[29] Chen Xin, Gao Yingxiang, Shang Honghui, et al. Increasing the efficiency of massively parallel sparse matrix-matrix multiplication in first-principles calculation on the new-generation Sunway supercomputer[J]. IEEE Transactions on Parallel and Distributed Systems, 2022, 33(12): 4752−4766
[30] Jiang Lijuan, Yang Chao, Ao Yulong, et al. Towards highly efficient DGEMM on the emerging SW26010 many-core processor [C] //Proc of the 46th Int Conf on Parallel Processing. Piscataway, NJ: IEEE, 2017: 422−431

Wu Zheng, born in 1992. PhD. His main research interests include parallel computer system architecture and machine learning.
武　铮, 1992 年生. 博士. 主要研究方向为并行计算机系统结构、机器学习.

Jin Xu, born in 1992. PhD. His main research interests include machine learning and distributed system.
金　旭, 1992 年生. 博士. 主要研究方向为机器学习、分布式系统.

An Hong, born in 1963. PhD, professor, PhD supervisor. Her main research interests include chip multiprocessor architecture, parallel computer system architecture, parallel programming environment and tools, and large data parallel storage and processing.
安　虹, 1963 年生. 博士, 教授, 博士生导师. 主要研究方向为片上多处理器架构、并行计算机系统结构、并行编程环境与工具、大数据并行存储与处理.