面向某国产加速器的卷积算子代码生成器

王小龙[1]，王家梁[2]，吉青[2]，侯丰尧[3,4]

（1.郑州大学计算机与人工智能学院，河南 郑州 450000；2.曙光信息产业（北京）有限公司，北京 100193；
3.中国科学院高能物理研究所，北京 100049；4.散裂中子源科学中心，广东 东莞 523803）

摘　要：近年来某国产深度学习加速器发展迅速，硬件资源持续变化并引入一系列张量核心指令，使得开发者在该加速器上进行卷积算子的手工适配和优化面临巨大挑战。为此，本文提出了一种面向该国产加速器的卷积代码生成器，以简化卷积算子的适配与优化过程。该生成器提供配置参数作为对外接口，用户仅需配置参数即可生成特定的卷积算子。生成器本身由三层架构组成：指令层封装底层指令，并根据硬件架构进行区分；组件层根据预设的硬件架构信息组织相应指令，从线程块和线程束角度提供高度抽象且可复用的功能组件；算子构建层则通过隐式卷积算法拼接功能组件，最终生成卷积算子。为保证卷积算子的计算性能，生成器从两方面进行优化：使用向量化算法和线程划分算法优化全局访存性能；使用转置算法转化乘累加指令的线程结构以优化写回性能。测试结果表明：该生成器的优化算法可显著提升算子性能；在两种硬件版本下，NHWC 存储布局的卷积算子性能分别达到官方算子库中卷积算子性能的 95%与 90%。该生成器为国产加速器的卷积算子适配优化提供了一种全新的解决方案。

关键词：国产加速器；隐式卷积算法；自动代码生成；张量核心；向量化访存

中图分类号：TP332　　**文献标志码：**A　　**DOI：**10.19678/j.issn.1000-3428.0070632

Convolutional Operator Code Generator for a Domestic Accelerator

WANG Xiaolong[1], WANG Jialiang[2], JI Qing[2], HOU Fengyao[3,4]

(1.School of Computer and Artificial Intelligence, Zhengzhou University, Zhengzhou 450000, Henan, China;
2.Dawning Information Industry (Beijing) Co., Ltd., Beijing 100193, China;
3.Institute of High Energy Physics, Chinese Academy of Sciences, Beijing 100049, China;
4.Spallation Neutron Source Science Center, Dongguan 523803, Guangdong, China)

【Abstract】Domestic deep-learning accelerators have developed rapidly in recent years. Hardware resources have been changing continuously, and a series of tensor core instructions have been introduced. Therefore, manually adapting and optimizing the convolution operator in the accelerator is a considerable challenge for developers. To this end, this study proposes a convolution code generator for a domestic accelerator to simplify the adaptation and optimization process of the convolution operator. The generator provides configuration parameters as an external interface, and users only need to configure these parameters to generate specific convolution operators. The generator itself consists of a three-layer architecture: the instruction layer encapsulates the underlying instructions and categorizes them according to the hardware architecture; the component layer organizes the corresponding instructions according to the preset hardware architecture information and provides highly abstract and reusable functional components from the perspective of thread blocks and thread bundles; and the operator construction layer splices the functional components according to an implicit convolution algorithm and generates a convolution operator. To ensure the computing performance of the convolution operator, the generator is optimized from two aspects: using the vectorization and thread partitioning algorithms to optimize the global memory access performance and using the transposition algorithm to transform the thread structure of the multiply-accumulate instruction to optimize the write-back performance. The test results show that the optimization algorithm of the generator can significantly improve the operator performance; under two hardware versions, the convolution operator performance of the NHWC storage layout reaches 95% and 90% of the official operator performance. The generator provides a new solution for adaptation and optimization of the convolution operators of domestic accelerators.

【Key words】domestic accelerator; implicit convolution algorithm; automatic code generation; tensor core; vectorized memory access

---

基金项目：国家重点研发计划（2021YFB0300200）。

作者简介：王小龙，男，硕士研究生，主研方向为加速器算子优化；王家梁，硕士；吉青、侯丰尧（通信作者），博士。

收稿日期：2024-11-19　　**修回日期：**2025-02-14　　**E-mail：**15638921013@163.com

0 引言

近年来，通用图形处理器(GPGPU)已广泛应用于科学计算[1]、大数据分析和机器学习领域[2]。尤其在深度学习中，GPGPU 凭借其强大计算能力推动了卷积神经网络(CNN)的快速发展[3]。CNN 在图像处理、目标检测、语音识别等任务中的优异表现，使其成为深度学习领域的代表性模型。卷积计算是卷积神经网络的核心，卷积计算性能对于推理和训练十分重要[4]。

卷积网络结构存在差异，卷积规模往往不同。而不同卷积参数下，最优性能的卷积算子通常不唯一[5]，因此往往需要针对网络定向优化[6]。同时，硬件资源也会影响卷积算子性能。本文基于某国产深度学习计算加速器开展研究，该平台硬件版本在不断更新，不同硬件版本的硬件资源存在差异，导致同一算子在不同硬件版本下性能表现不同。该国产加速器还引入了张量核心单元，由于使用张量核心计算的卷积算子[7]性能远超传统的计算统一设备架构(CUDA)核心算子，因此也需要进行功能适配。而在优化与适配过程中，可能只需针对部分模块进行调整[8-9]，例如可仅调整使用张量核心替代 CUDA 核心计算，卷积算子的访存与写回模块无需改变，而人工开发时则需重复实现这些模块，引入了大量的重复工作。同时该国产加速器有着自己的硬件与指令特性，需结合这些特性进行针对性优化。

为了解决在该国产深度学习计算加速器上手工优化卷积算子面临的问题，并实现对该加速器的快速适配与优化，本文提出了一种面向该加速器的卷积算子代码生成器(称为 convGen)。本文工作的主要意义在于，解决了该国产加速器进行卷积算子优化与适配时的重复工作问题，并保证生成的算子性能与手工优化算子相近。本文的主要贡献如下：

1)将卷积问题模块化，将优化与适配问题转化为组件的细节调整。

2)基于模块化的设计思想，设计并实现基于三层架构的卷积代码生成器，以支持某国产加速器多硬件版本的卷积算子生成。代码生成方案使得生成全量的卷积算子后再遍历选优的方案成为可能，将定向优化问题转化为批量生成与测试问题。

3)提出向量访存负载算法和线程划分算法优化访存性能，并提出转置算法，通过转化乘累加指令的线程结构以优化写回性能。

1 相关背景

1.1 国内外加速器对比

本文基于的深度学习计算加速器是一种类 GPGPU 加速器[10-11]，该加速器在功能上可以完全兼容主流异构加速器。国产加速器使用异构计算可移植接口(HIP)编程模型，而 NVIDIA 则采用 CUDA[12]编程模型，两者皆为三层编程模型。三层编程模型由网格块(Grid)、线程块(Block)和线程(Thread)组成，Grid 是包含多个 Block 的集合，每个 Block 包含多个 Thread，Thread 是执行实际计算的最小单位。三层编程模型如图 1 所示。

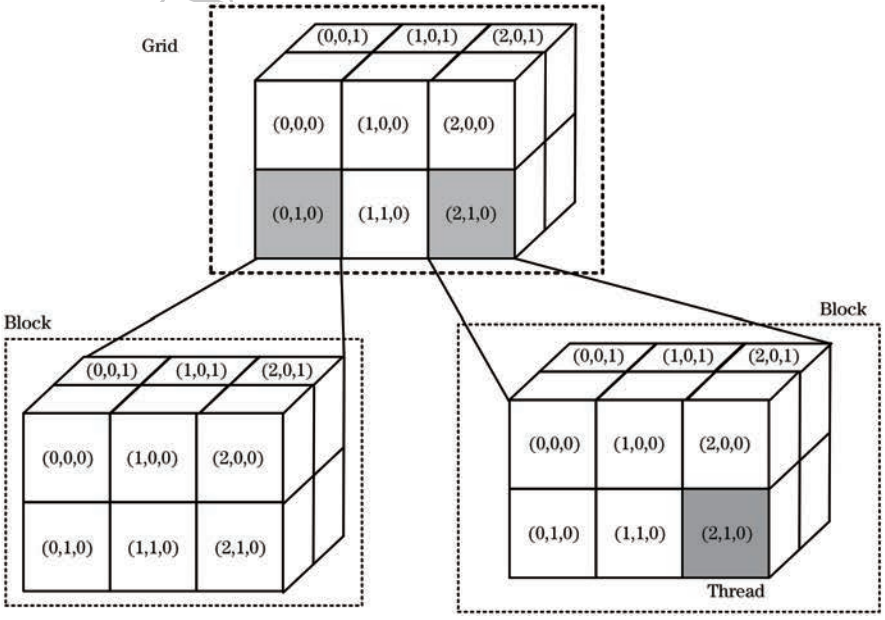

图 1 三层编程模型

Fig. 1 Three-layer programming model

本文所采用的某国产深度学习计算加速器(以下简称为“国产加速器”)与国外主流加速器在技术和应用重点上存在差异。以 NVIDIA-V100(以下称为“NVIDIA 加速器”)为例进行对比:

1)张量核心指令存在差异。以半精度浮点数(FP16)类型为例:NVIDIA 加速器乘累加指令的矩阵分块尺寸设置为 m8n8k4[13],而国产加速器则为 m16n16k16,其张量计算指令具有更高的计算负载。NVIDIA 加速器指令的输出线程结构为行优先结构,而国产加速器指令的线程结构为列优先结构,因此需根据国产加速器指令特性进行优化。

2)共享内存矩阵加载指令(ldmatrix)存在差异。NVIDIA 加速器某指令的加载粒度为 m8n8,而国产加速器则为 m16n32,国产加速器一次可加载更多数据,具有更高带宽利用率。

3)线程束的设计存在差异。NVIDIA 加速器线程束比国产加速器更小,因此其在处理不同规模的任务时更加灵活,而国产加速器线程束大小为 64,其在大规模规则吞吐场景和计算密集场景下表现更好。

国产加速器与国外主流加速器存在差异,因此在适配与优化过程中,需结合国产加速器的特性进行优化。

1.2 深度学习计算加速器

本文研究基于某国产深度学习计算加速器开展,其硬件架构如图 2 所示。

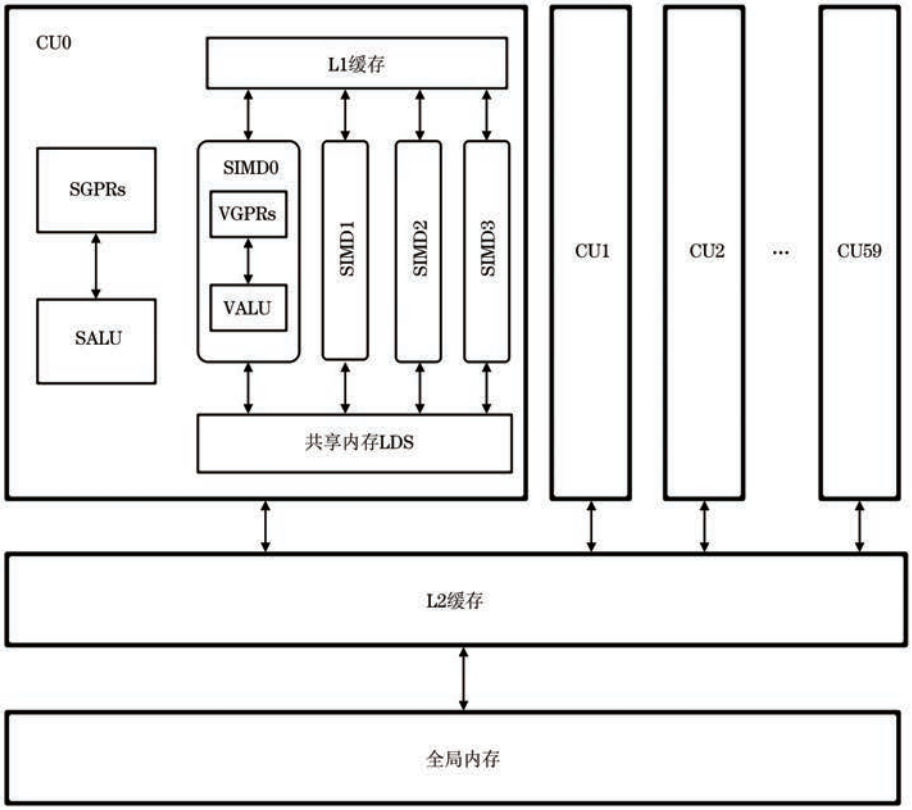

图 2 某国产加速器硬件结构

Fig. 2 Hardware structure of a domestic accelerator processor

图 2 描述了该国产加速器的硬件结构,包含 5 层内存结构,分别为全局内存、L1 缓存、L2 缓存、寄存器、共享内存。其中,全局内存、共享内存和寄存器是可编程对象,而缓存并不能直接编程,主要由硬件管理用于数据缓存复用。

1.3 算子生成

目前算子生成方式可以分为两种类型:一种是通过以 CUTLASS[14] 为代表的模板库生成算子;另一种是通过高级代码生成器生成算子,如以 TVM[15] 和多级中间表示(MLIR)[16] 工具链为代表的深度学习编译器。

1.3.1 模板库生成算子方式

模板库生成加速器算子的方式近年来受到广泛关注,其中 NVIDIA 的 CUTLASS 和 AMD 的 CK 库是使用模板库生成算子的典型代表。CUTLASS 作为 NVIDIA 提供的开源 CUDA C++模板库,提供了高效且高度抽象的组件,用于执行矩阵乘法、卷积计算和其他张量计算。模板库常由加速器厂家维护,因而可以对加速卡硬件进行更加精确的控制,实现与底层硬件紧密结合的优化,从而生成更加高效

的算子。但是也因为模板库通常与硬件厂商绑定，所以无法通过简单移植实现跨平台使用。

1.3.2 编译器生成算子

编译器也可以实现算子生成，如 TVM 和 MLIR 工具链。TVM 是为了实现深度学习框架在不同硬件平台上部署并解决性能问题而设计的编译器系统，其将高层模型转化为中间表示，然后自动进行优化并生成高效的算子代码；而 MLIR 工具链是由 Google 开发的编译器基础设施，旨在解决深度学习和加速计算中的多样化硬件需求。MLIR 工具链生成算子的核心思想是通过灵活的 MLIR 进行代码优化，使得编译器可以跨越多个硬件平台，实现高效的算子生成和性能优化。由于编译器生成的代码追求多平台的兼容性，无法针对特定硬件特性进行深度优化，这往往导致生成的算子在特定硬件上的性能难以达到手动调优或模板库生成算子的性能。

1.4 卷积计算

1.4.1 深度学习卷积计算

卷积计算是将一个小的权重矩阵（卷积核）与输入数据的局部区域进行元素逐一相乘并累加的过程[17]，基本计算过程如图 3 所示。

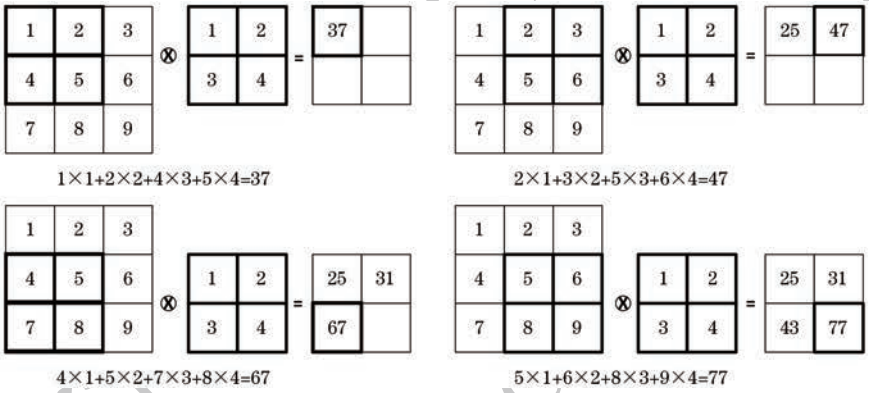

图 3 卷积计算实例

Fig. 3 Examples of convolution computation

在实际深度学习网络中，一次卷积计算往往是多批次、多通道的，网络中最常见的存储布局包括 NCHW 和 NHWC 两种格式[18]。在这两种布局中，N 表示批次大小（Batch Size），C 表示通道数（Channels），H 表示图像的高度（Height），W 表示图像的宽度（Width）。在 NCHW 布局下，先将内存中的数据按批次进行组织，再按通道划分，每个通道内部连续存放由高度和宽度组成的二维图像数据。而 NHWC 布局将通道维度放在最后一维，使图像的通道信息在内存中连续存储。

深度学习中常见的浮点精度有单精度浮点数（FP32）、FP16 和双精度浮点数（FP64）。FP32 适用于大多数任务，但消耗更多计算资源；FP16 能提高训练速度并节省内存，但可能会导致精度损失。因此，深度学习常用混合精度计算[19]：计算过程使用 FP32，存储采用 FP16，这样可提升性能，减少内存占用，同时保持足够的计算精度。

1.4.2 卷积算法

常见卷积算法可以分为 4 种，分别为直接卷积算法[20]、快速傅里叶变换（FFT）卷积算法[21]、Winograd 算法[22]和隐式卷积算法[23]。

convGen 中主要通过隐式卷积算法计算卷积。隐式卷积算法将卷积的滑动计算问题逻辑上转化为矩阵乘法问题[24]，借助矩阵分块等优化方法，可以有效提升卷积算子性能，其计算过程如图 4 所示。

2 卷积算子代码生成器分级架构

convGen 由三层架构组成，即指令层、组件层和算子构建层。指令层专注于具体指令的执行，直接与硬件交互，负责完成底层操作的实现并根据硬件架构进行区分；组件层从线程束和线程块负载出发，向算子构建层提供高复用的功能组件，并且根据预配置的硬件信息，动态选择适配的底层硬件指令进行实现；算子构建层以卷积算子的功能为核心，通过隐式卷积算法拼接功能组件，生成隐式卷积算子。三层结构划分如图 5 所示。

convGen 对外提供了一些可配置项，用户通过配置这些参数项可以生成特定的卷积算子，如表 1 所示。其中：Arch 表示目标架构；Direction、Datatype 和 Layout 表示卷积算子种类；blockShape 表示线程块的计算负载；warpShape 表示线程束计算负载。架构、方向、精度和存储布局决定了卷积问题种类，而线程块与线程束负载则为可配置的性能优化项。

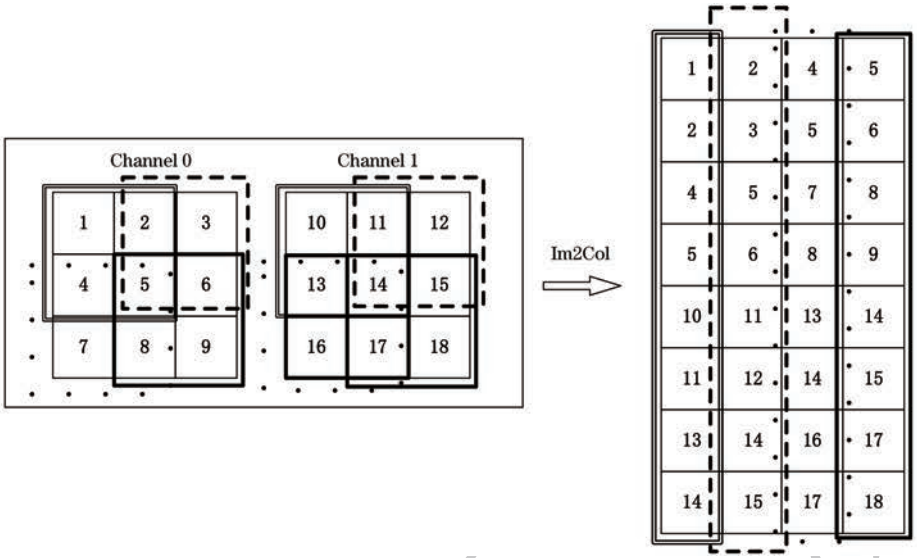

图 4 隐式卷积算法计算过程

Fig. 4 Computation process of implicit convolution algorithm

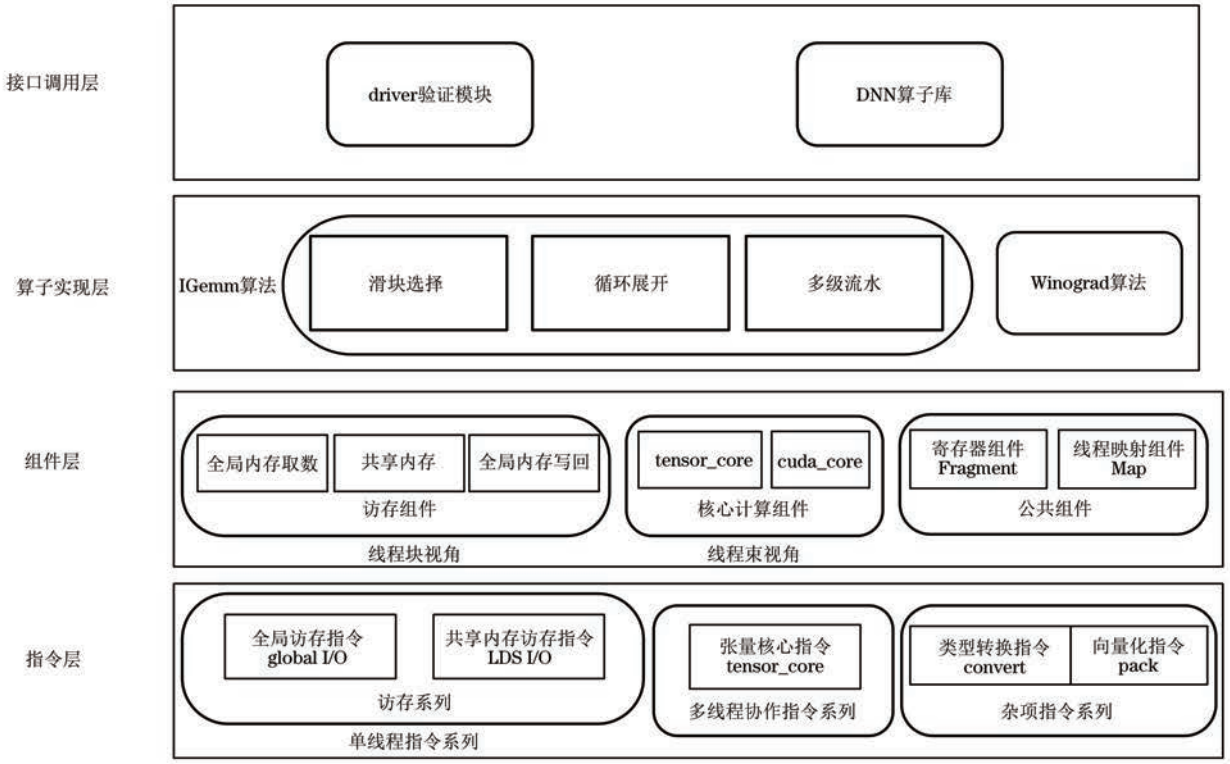

图 5 convGen 总体架构

Fig. 5 convGen overall architecture

表 1 convGen 可配置参数项

Table 1 convGen configurable parameters

可配置项	含义	配置
Arch	目标硬件平台	K/B
Direction	卷积方向	fwd/bwd/wrw
Datatype	计算精度	FP32/FP16/Int8
Layout	存储布局	NCHW/NHWC
blockShape	线程块负载	<bM, bN, bK>
warpShape	线程束负载	<wM, wN, wK>

2.1 指令层

指令层封装了丰富的硬件指令，提供基础的硬件功能。指令层根据指令的可执行硬件版本对其进行标记与区分，便于组件层依据硬件平台选择不同指令实现功能组件。从执行单元角度划分，指令可分为单线程执行指令和多线程协作指令。

2.1.1 单线程执行指令

单线程执行指令指的是在类 cuda_core 硬件核心上执行的指令，由于这些指令在多硬件版本通用，

因此不根据硬件架构区分。单线程执行指令包括但不限于全局内存访存指令、共享内存访存指令、数据类型转换指令等。单线程执行指令种类较多，并且不同指令可以实现相同功能。例如全局访存指令存在 buffer_load_dword，buffer_load_dwordx2，buffer_load_dwordx3，buffer_load_dwordx4 指令，想要连续取 4 个单精度数据[25]，可以使用 1 次 buffer_load_dwordx4 指令，或者连续调用 4 次 buffer_load_dword 指令。指令的选择会影响算子性能，如使用 buffer_load_dwordx4 指令进行 1 次访存比连续调用多次 buffer_load_dword 指令的性能更好。因此，尽可能采用向量化指令一次访问多个数据元素，从而提高数据访问效率，显著提升算子的整体性能。

2.1.2 多线程协作指令

多线程协作指令指的是在张量核心(tensor_core)硬件核心上执行的指令，这类指令以线程束级别执行，线程束内全部线程协作完成计算[26]。本文所采用的国产加速器上提供了一系列的多线程协作指令，例如乘累加系列指令和矩阵加载系列指令。但是不同硬件版本对这些指令的支持存在差异，例如该加速器的 K 版本不支持共享内存矩阵加载系列指令。相同功能的张量指令在不同硬件上也存在差异，例如单精度类型的乘累加指令在 K 系列硬件上累加深度为 4，而在 B 系列硬件上累加深度则为 8。因此，指令层针对不同硬件架构对指令进行划分和标记，以便组件层根据目标架构选择相应的指令实现功能。

多线程协作指令具有更高的执行效率[27]，但使用方法也受到一定限制。多线程协作指令对线程与寄存器的关系有严格的使用规定。convGen 设计寄存器映射表(registerMap)和线程映射表(threadMap)用于描述张量核心指令结构。以 K 系列硬件中乘累加指令为例，如图 6 所示。

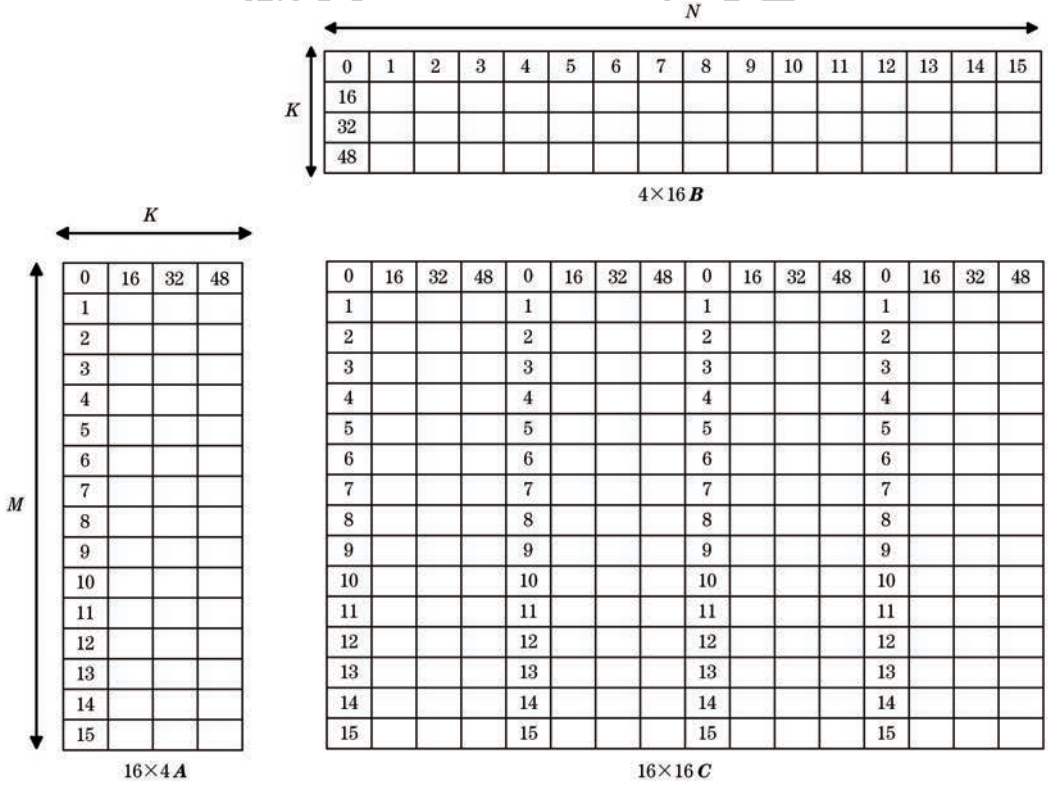

图 6 mmac_16x16x4_f32 指令

Fig. 6 mmac_16x16x4_f32 instruction

图 6 为国产加速器 K 系列硬件的乘累加计算指令结构图。该指令通过张量核心，在硬件层面计算了一个 $\boldsymbol{A}(16,4)*\boldsymbol{B}(4,16)=\boldsymbol{C}(16,16)$ 的矩阵乘法。其中：数字代表线程编号，相同数字代表同一线程寄存器中数据相对位置，可知该乘累加指令的每个线程输出 4 个数据且在逻辑上并不连续，在 N 方向上隔 4 出 1；线程结构则以 M 方向为连续方向，将线程束划分为 16×4 的二维结构。

convGen 使用 registerMap 描述寄存器的离散存储结构，使用 threadMap 描述线程二维划分结构。因此，乘累加指令的矩阵运算结构可表示为 threadMap 在 registerMap 上的重复排布。以图 6 中指令的输出矩阵 $\boldsymbol{C}$ 为例，其寄存器映射表为 registerMap<1,4>，线程的结构为 Map<16,4>，

线程映射表为 threadMap < threadId% 16, (threadId/16)%4>,则指令的计算结果输出过程可以表述为 16×4 的二维线程束(threadMap)在 N 方向(registerMap<1>,1 代表 N 方向)上滑动 4 次,逐点完成数据输出。不同硬件版本的张量核心指令皆可通过映射结构表示,其细节存在差异,这里不再论述。

2.2 组件层

组件层从线程束和线程块的角度组织与调用底层指令。该层将常见操作进行抽象和整合,形成高度复用的功能组件。随着后续的硬件指令扩展,仅需根据指令层中的硬件标签选择相应指令实现对应组件功能,便可实现算子适配。

2.2.1 Fragment 组件

寄存器是加速器编程的核心。在 convGen 中,寄存器功能由 Fragment 组件承担。Fragment 组件由数据类型和数据量定义,通过模板配置项在编译期确定。

2.2.2 全局访存组件

全局访存组件负责从全局内存中获取数据,加载数据到寄存器。在第 2.1.1 节中,提到了不同访存指令可以实现相同功能,采用向量化的高负载访存指令可以提升性能。因此,convGen 在全局访存模块中采用编译期计算最大向量化访存负载算法(算法 1)来计算每个线程的最大向量化访存负载,以保证访存的性能。

算法 1 编译期计算最大向量化访存负载算法

输入 线程块内线程数 BlockThreadNum,总访存任务量 FragH、FragW,初始访存负载 BasicWidth 为 128/Sizeof(DataType),数据类型 DataType

输出 最大向量化访存负载(VW)

1. 定义编译期递归函数 VW/*定义 VW 算法函数*/

2. if (BlockThreadNum * BasicWidth > FragH * FragW)/*判断当前向量负载下,总的访存任务是否超过目标任务量,此为 VW 算法的递归条件*/

3. 如果条件为真,递归调用 VW(FragH, FragW, BlockThreadNum, BasicWidth/2)/*如果负载太大,递归调用 VW 函数,并减小 BasicWidth*/

4. 如果条件为假,返回 BasicWidth/*如果负载大小合适,返回 BasicWidth,结束递归*/

在国产加速器中,128 bit 是最大访存负载,同时也支持 8、16、32、64 bit 的访存负载。算法引入了数据类型变量,将访存负载(比特长度)转化为对应数据类型的向量长度。而为了尽可能使用高负载访存指令,算法初始设置 128 bit 的访存负载,再逐步递归减少访存负载,判断当前负载是否可用。最终,算法计算出每个线程能够访存某一类型数据的最大数量,即最大向量化访存负载。通过编译期计算最大向量化访存负载算法,能够在编译期计算每个线程的访存负载,从而优化访存效率。

连续线程访问内存中连续数据,可以实现合并访存,提升访存性能。convGen 使用了连续线程行优先任务划分算法(算法 2)划分全局访存的线程结构。

算法 2 连续线程行优先任务划分算法

输入 线程块内线程数 BlockThreadNum,总访存任务量 FragH、FragW,最大向量化访存负载(VW)

输出 线程二维结构 ThreadMap,单次访存任务量 EachMap,访问次数 CntMap

1. StrideHeight = min(FragH, BlockThreadNum * VW) /*计算 H 方向单次访问量*/

2. StrideWidth = (BlockThreadNum * VW/StrideHeight) /*计算 W 方向单次访存任务量*/

3. SegsHeightCnt = FragH/StrideHeight/*计算 H 方向访存次数*/

4. SegsWidthCnt = FragW/StrideWidth/*计算 W 方向访存次数*/

5. InlineThreadCnt = StrideHeight/VW/*计算连续方向连续向量化访存线程数量*/

6. ThreadMap = Map<(threadId%InlineThreadCnt) * VW, threadId/InlineThreadCnt>/*输出划分后的线程二维结构*/

7. EachMap = Map < StrideHeight, StrideWidth > /*输出单次访存任务量*/

8. CntMap = Map<SegsHeightCnt, SegsWidthCnt> /*输出全部任务访存次数*/

全部访存任务量可能较大,因此算法首先对线程块的全部访存任务进行了切分,将访存任务拆分为 CntMap 次,一次线程块访存的任务量为 EachMap,一次基础访存任务则由二维线程结构(ThreadMap)承担。其中线程结构的第一个参数(threadId % InlineThreadNum)为连续线程,可承担显存中连续存储数据的访存任务,而第二个参数(threadId /InlineThreadNum)则承担不连续维度的数据访存任务。

在 NHWC 存储格式下,卷积前向推理计算如图 7 所示。图 7(a)表示经隐式卷积转化后数据的逻辑结构,图 7(b)表示图像和卷积核的数据在显存中实际存储结构。

在图 7(b)所示场景下,原始图像(Input)和卷积核(Weight)的通道维度(C)数据,在全局内存中是连续存储的,则可分配连续线程(ThreadMap<0>,0

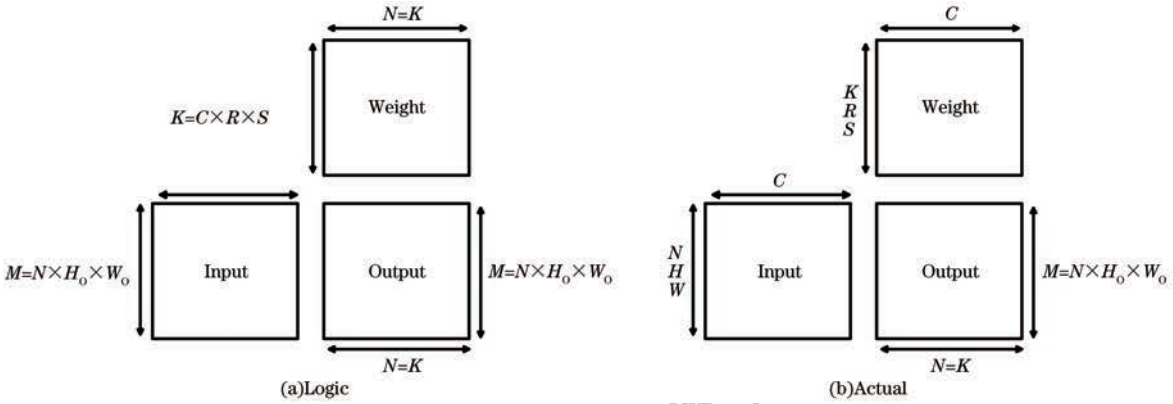

图 7 前向卷积矩阵存储示意图

Fig.7 Schematic diagram of forward convolution matrix storage

代表第一个参数)用于承担原始图像的 C 维度中数据的访存任务,并以 VW 为单位(算法 1 可得)进行向量化访存,非连续线程(ThreadMap<1>)则可承担 N、H、W 等不连续维度的数据访存任务。

2.2.3 共享内存访存组件

共享内存访存组件则根据硬件版本有所区分,例如 K 系列部分硬件版本并不支持矩阵加载指令,需要通过 CUDA 的核心软实现进行访存操作。而 K 系列的后续硬件版本与 B 系列则支持矩阵加载指令。因此,共享内存访存根据目标硬件架构并参考指令层中的硬件标签进行了差异化实现。共享内存访存组件的功能实现与计算组件类似,这里不再展开论述。

2.2.4 计算组件

计算组件从线程束视角管理和组合底层指令,并根据目标硬件架构(可配置参数中 Arch 项)动态选择硬件指令。从计算负载角度(warpShape)出发,线程束可多次使用乘累加指令以提高计算负载。某线程束调用多次乘累加指令来完成复杂计算任务过程如图 8 所示。

但由图 6 可知,乘累加指令输出结果的线程结构为列优先结构,在写回时列优先的线程划分无法有效适应行优先存储的硬件架构。因此,在计算组件中采用转置算法进行优化。

转置计算方法如下:计算组件交换指令的入参顺序,以转化写回过程的线程结构实现写回优化。其中,C_Naive=MMAC(A,B)可以表示逻辑上的一次乘累加计算,其行为用数学描述可以为 $\boldsymbol{M}_{\text{C\_Naive}}=\boldsymbol{A}^{\text{T}}*\boldsymbol{B}$。而 convGen 组件层中使用 MMAC(B,A)的计算顺序,此时组件层中乘累加计算逻辑为 C_Swap=MMAC(B,A),即 $\boldsymbol{M}_{\text{C\_Swap}}=\boldsymbol{B}^{\text{T}}*\boldsymbol{A}$。而为保证计算结果的正确性,需对 C_Swap 进行一次转置变换实现,将 $\boldsymbol{M}_{\text{C\_Swap}}$ 转化为 $\boldsymbol{M}_{\text{C\_Swap}}^{\text{T}}$,不难证明 $\boldsymbol{M}_{\text{C\_Naive}}=\boldsymbol{M}_{\text{C\_Swap}}^{\text{T}}=(\boldsymbol{B}^{\text{T}}*\boldsymbol{A})^{\text{T}}$,而转置变换可通过将写回线程的列优先结构转化为行优先结构实现。交换入参转置和线程结构转置的两次变换,最终结果不变,但线程结构转变为行优先结构,可以更好地适配行优先存储的硬件架构,通过合并访存优化性能。

2.3 算子构建层

算子构建层从 Grid-Block 视角出发,通过隐式卷积算法映射线程与全局内存地址拼接功能组件,最终生成卷积算子。在第 3 章,详细展示了在算子构建层使用隐式卷积算法生成卷积算子的过程。

3 卷积代码生成

隐式卷积算法加速卷积计算,按照数据流动方向划分,可分为以下 4 个阶段:全局访存阶段,共享内存阶段,乘累加计算阶段,全局写回阶段。流程如图 9 所示。

3.1 全局访存

全局访存负责从全局内存加载数据到寄存器。convGen 设计并实现了全局取数模块,使用隐式卷积算法,将组件层中的通用线程结构映射至输入图像和卷积核的全局内存空间,并进行实际访存。全局访存示意图如图 10 所示。

3.1.1 隐式卷积的索引变换

在全局取数模块中,convGen 设计并实现了地址变换模块,负责将全局线程索引映射到显存地址。该模块接收全局线程索引和卷积规模参数,依据隐式卷积算法的转换规则,将每个线程索引映射到各自的全局空间,作为实际执行的逻辑规则。

卷积计算的方向较多,例如前向推理、反向传播和权重更新,每个方向下矩阵逻辑映射也不相同,因此 convGen 在这里做了大量的特化处理,用于处理不同卷积方向不同矩阵的映射关系,如图 10 中 determineMatrixType 部分所示。

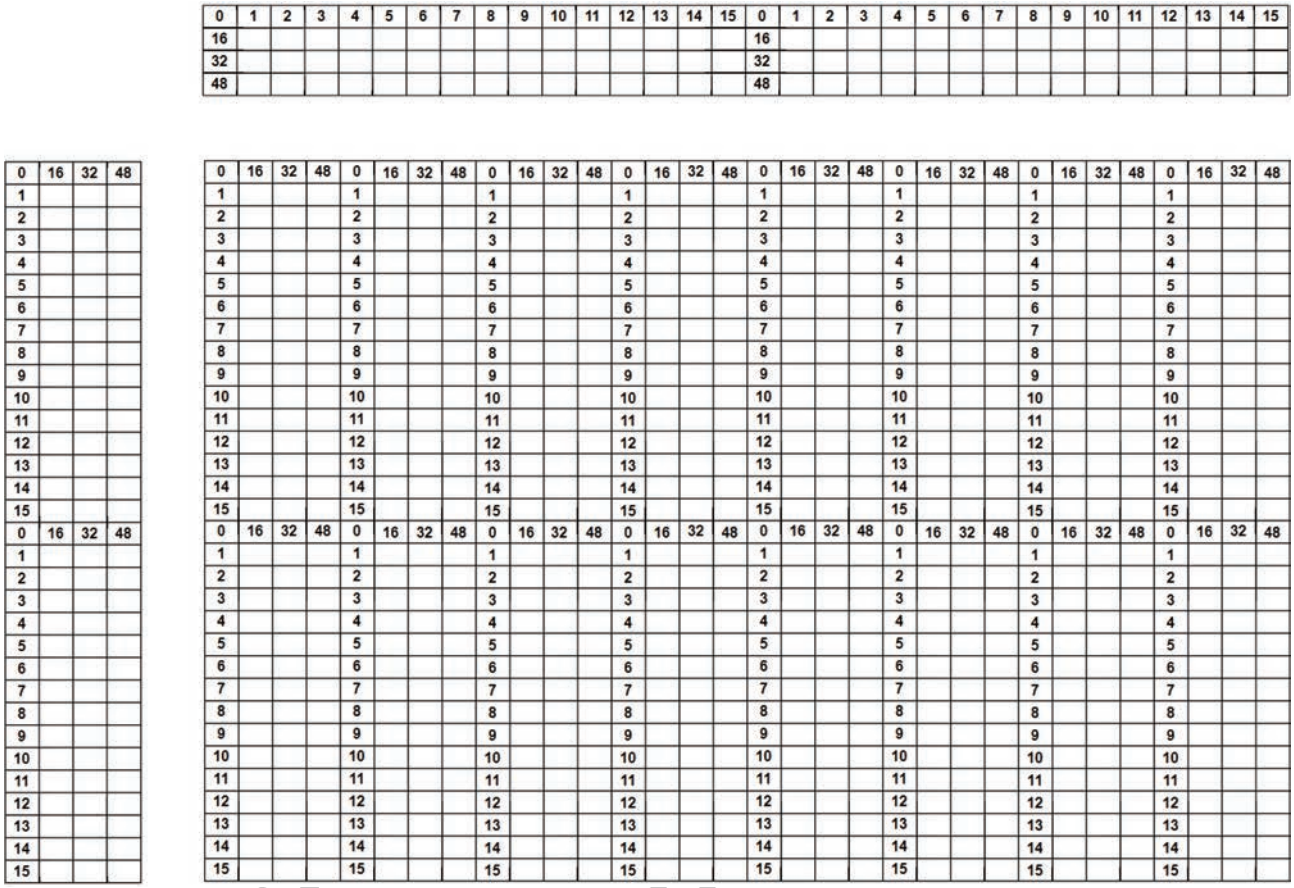

图 8 单线程束组合使用乘累加指令示意图

Fig. 8 Schematic diagram of single warp combination using multiply-accumulate instruction

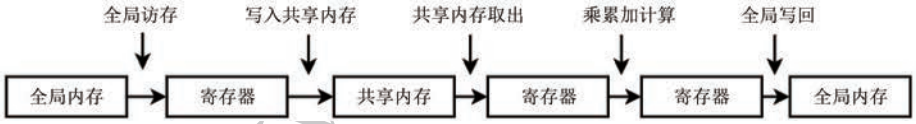

图 9 算子构建层数据流动方向

Fig. 9 Data flow direction at the operator composition layer

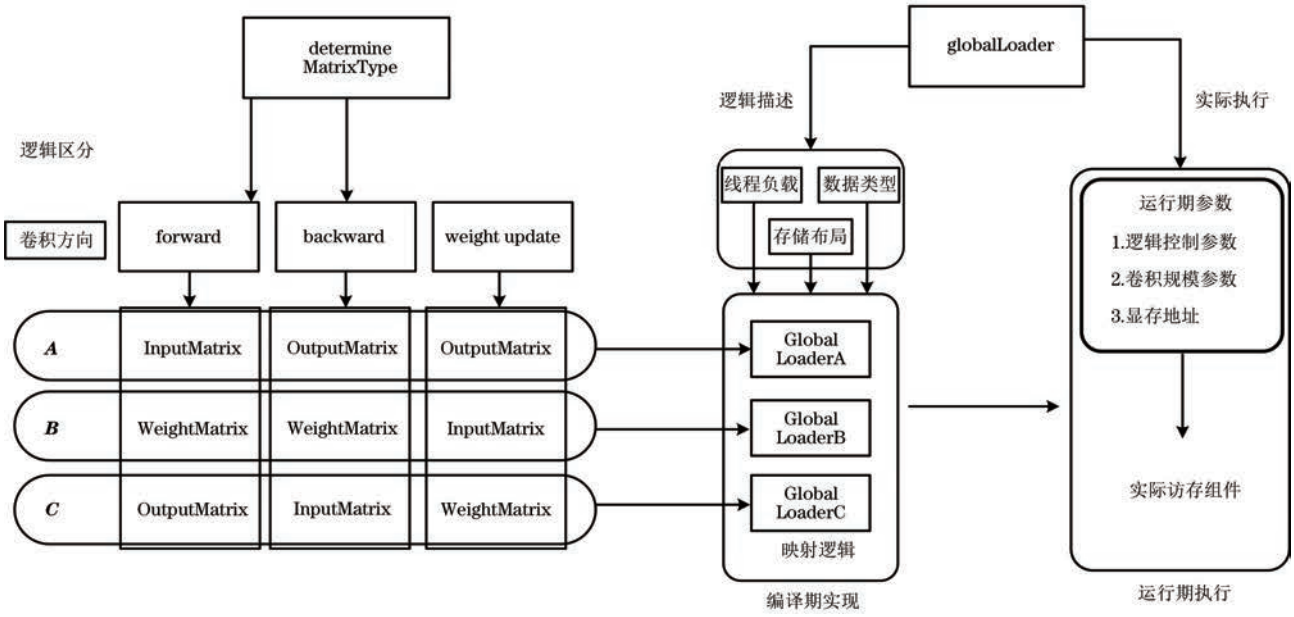

图 10 全局访存逻辑与执行设计图

Fig. 10 Global access logic and execution design diagram

3.1.2 实际取数访存

实际执行模块根据索引变换后的映射地址调用组件层访存组件实现访存，将数据由显存加载至寄存器中。具体可通过第 2.2.2 节介绍的全局访存组件实现。

3.2 共享内存

共享内存是可复用数据的缓存区[28]。经过全局访存取数的逻辑映射后，卷积问题实质上就转化为矩阵乘法问题。矩阵乘法可以借助共享内存实现数据复用，减少访存。共享内存操作可根据数据的读写过程划分为写入过程和取出过程。

写入共享内存模块将寄存器中的数据写入共享内存，以支持后续的数据复用；而共享内存取数模块则根据核心计算组件的要求，从共享内存中取出数据。该过程通过第 2.2.3 节介绍的共享内存访存组件实现，从而完成共享访存获取乘累加数据的操作。得益于分级设计，这一过程无需考虑访存组件的实现细节。

3.3 乘累加核心计算

乘累加计算是卷积计算的核心操作。在算子构建层中，乘累加计算可以仅通过转发组件层中核心计算组件来完成。但是否采用预取(prefetch)优化策略是影响乘累加核心计算性能的关键因素。在国产加速器中，由于访存硬件与计算硬件相互分离，访存与计算可以实现并行操作。通过一次数据预取，可以在执行当前计算的同时加载下一次所需数据，从而实现访存与计算的流水化处理。但在循环次数较少的情况下，预取操作可能反而导致性能劣化。为此，convGen 针对不同情况特化两种实现方案，通过模板参数将其是否采用预取策略设置为可配置选项，便于灵活选择是否启用预取优化策略。

3.4 全局写回

全局写回模块负责将计算完成的数据写回至全局内存。使用逻辑结构将全局线程映射到全局内存地址，实际执行结构则调用底层写回指令将卷积计算结果写回内存。由于存在混合精度计算，除了表示数据的写回结构，全局写回模块还需处理数据类型转换。最终将计算结果以正确的类型和结构写回全局内存[29]，从而完成卷积计算。

通过以上 4 个步骤，最终在算子构建层组合出一个卷积算子。

4 实验结果与分析

4.1 实验环境与参数

在本实验中，性能测试基于某国产深度学习计算加速器展开。该平台由多个节点组成，每个节点包含 1 个 CPU 和 4 个加速器。计算节点之间通过 25 GB/s 带宽的高速网络进行互连；加速器则使用 16 GB 的高带宽内存(HBM2)，通过 PCIe 与 CPU 连接。加速器包括 K 版本和 B 版本两种型号。实验平台的软件版本为 24.10。

卷积规模参数由常见卷积神经网络抽取，包括 YOLOv3/YOLOv5/YOLOv8 网络、Mask R-CNN 网络、ResNet50 网络等，卷积参数如表 2 所示。其中，N 表示输入矩阵的批次数，C 表示输入矩阵和卷积核矩阵的通道数，H 和 W 分别表示输入矩阵的高和宽，H_{Filter}、W_{Filter} 分别表示卷积核的高和宽，K 表示卷积核的数量，H_{Pad}、W_{Pad} 是输入矩阵在高和宽方向补边的大小，H_{Stride}、W_{Stride} 表示卷积核在输入矩阵两个维度上的滑动步长。

表 2 卷积问题参数

Table 2 Convolution problem parameters

参数	设置
N	128
C	512
H	28
W	28
K	256
H_{Filter}	1
W_{Filter}	1
H_{Pad}	0
W_{Pad}	0
H_{Stride}	2
W_{Stride}	2

convGen 支持生成多种卷积算子，包括多种数据布局(NCHW、NHWC)，多种卷积类型(前向、反向、权重)，多种数据类型(FP32、FP16、INT8 等)，计算结果都已通过正确性验证。

4.2 全局访存相关算法测试

在表 2 参数设置下，采用全局访存优化算法卷积算子和不采用全局访存优化算法卷积算子的性能对比如图 11 所示。

通过编译期计算最大向量化访存负载算法，可以在编译阶段确定每个线程的最大向量化访存负载，从而有效减少访存指令数量。连续线程行优先分配算法则优先分配连续线程，访问内存中连续存储数据，从而进一步优化访存效率。综合两种全局访存优化算法，可以实现向量化访存和合并访存。经过全局访存优化的卷积算子性能达到常规的逐点访存的卷积算子性能的 385.15%。

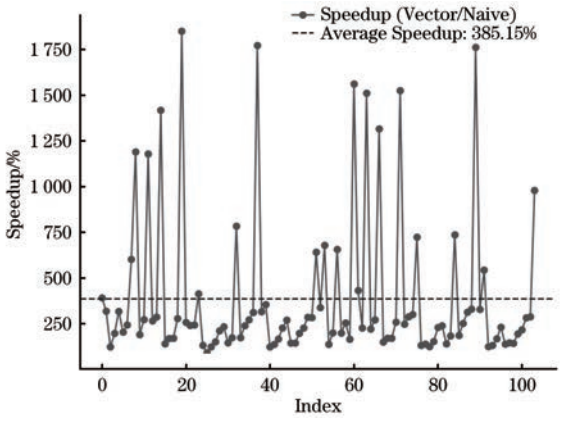

图 11 全局访存优化加速比

Fig. 11 Global memory access optimization acceleration ratio

4.3 转置算法实验测试

对转置算法进行性能测试，并与常规方法进行对比。抽取其中 100 组实验结果展示，如图 12 所示。

由图12可知，经由转置算法交换写回逻辑后，平均取得了 7%的性能提升。100 条卷积参数中，88%的卷积参数实现了性能优化，而约 12%的参数出现了性能劣化。对实验结果进行分析，笔者认为性能提升的原因是：经过转置后线程结构转化为行优先结构，更适合底层行优先的存储结构，有效实现合并访存，实现写回性能提升。而对性能劣化的场景进行分析：回顾第 2.1.2 节与第 2.2.4 节可知，转置后的写回过程转化为线程束在 M 方向的滑动写回过程，在 N 方向的连续排列线程的结构会由于边界判断原因，更容易引起线程分化，当线程分化造成的性能劣化影响超过了转置操作所带来的预期收益时，会导致性能的整体下降。而在 N 方向输出数据量较少的情况下，这种性能劣化超越转置收益的情况较容易发生。但是从全量测试结果来看，采用转置算法的卷积算子整体上仍然具有稳定的性能提升收益，因此 convGen 采用了转置算法进行优化。

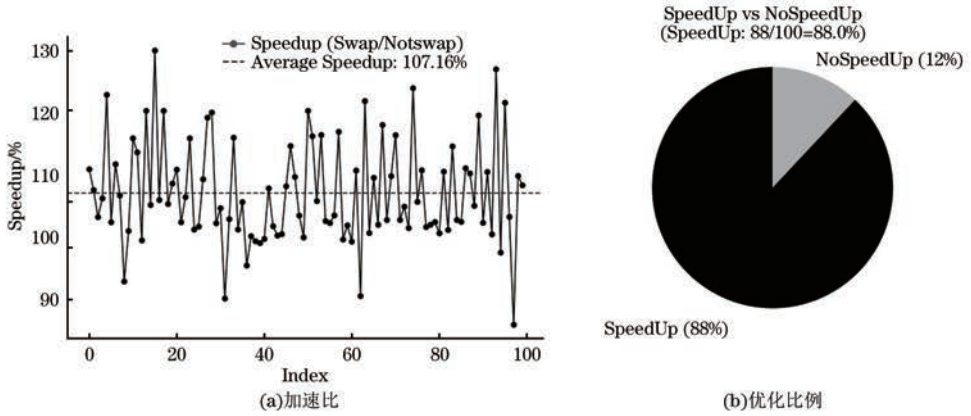

图 12 转置算法与常规方法性能比较

Fig. 12 Performance comparison between transposition algorithm and conventional method

提升算子数据写回全局内存性能，核心思路是实现线程合并写回，并将离散的逐点写回转化为向量化写回，以减少写回指令数。目前转置算法仅针对乘累加指令做了特殊适配性优化，线程划分结构仍然局限于乘累加指令的二维结构，且仍为逐点写回。一个可能的解决方案是使用共享内存交换寄存器中数据，以实现线程结构的重新划分，并实现向量化写回。后续，convGen 将尝试这一方案以进一步提升性能。

4.4 组合策略测试

经由第 4.2 节和第 4.3 节的优化算法后，convGen 可以根据模板参数生成一个特定的卷积算子。但是算子的优化配置参数项对于卷积性能十分重要[30]，优化配置参数项（blockShape，warpShape）的笛卡儿积构成了算子择优的解空间。在 NHWC 存储格式和 FP16 数据类型下，convGen 共生成 104 个卷积算子，实验对生成的全部算子进行性能测试。以表 2 中卷积问题规模作为测试参数，实验对比了全部生成的算子耗时和官方算子库中最优算子耗时，实验结果如图 13 所示。

经过全量遍历测试后，最优的卷积算子耗时为 0.176 80 ms，而官方算子在相应卷积参数下相应的算子耗时为 0.290 72 ms，convGen 最优算子性能是官方算子性能的 160%。经过分析，官方算子在该组卷积参数下，采用了 block＜64，64，32＞，warp＜32，32，32＞的算子配置，每个线程块包含 4 个线程束，共 256 个线程。而 convGen 的最优算子的配置为 block ＜128，128，32＞，warp＜128，64，32＞，每个线程块则仅仅包含 2 个线程束，共 128 个线程。在表 2 的参数配置下，convGen 采用的大分块策略可以显著减少线程块的数量和线程块的调度开销。从访存和计算比例的角度分析，

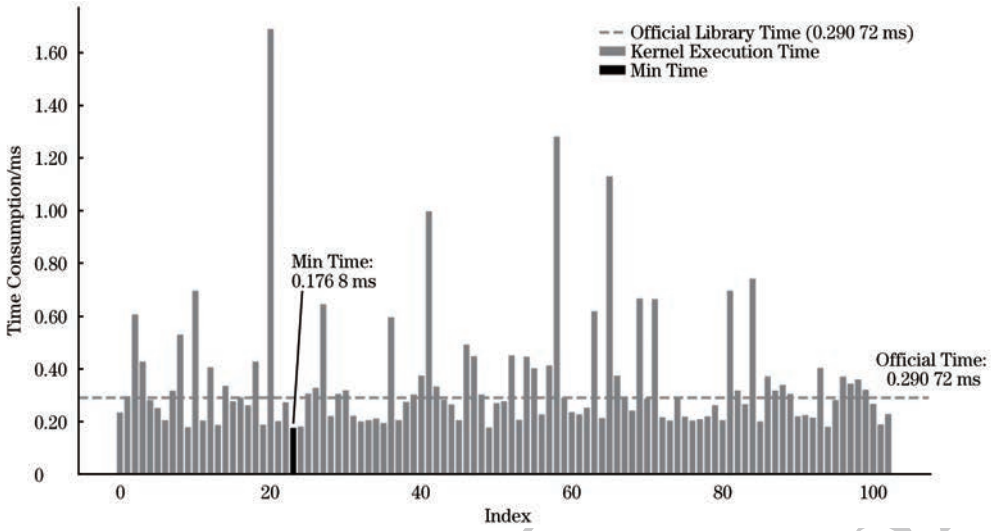

图 13 某卷积参数的全量算子耗时

Fig. 13 Total operator time consumption for a convolution parameter

convGen 配置参数下的算子策略显著提升了计算与访存的比例(算存比)。随着张量核心指令的引入,在计算性能不是优化瓶颈的场景下,convGen 采用的高算存比的配置策略,使得每次访存能够支持更多的计算操作,从而更好地针对现有的瓶颈进行优化。

而在卷积参数较小情况下,滑块策略则完全相反。采用较小的滑块配置可以占用更多的计算核心,具有更高的并行度和计算性能。

convGen 通过配置模板参数来生成算子的方式,使得生成全量的卷积算子后再遍历选优的方案成为可能,能有效覆盖手工优化时的难以覆盖的滑块策略,从而获得某一卷积参数下最优的卷积算子配置。

4.5 全量数据性能测试

4.5.1 热点卷积核测试

在全部卷积参数中,卷积核尺寸为 1×1 的有 506 条,3×3 的有 404 条,共占 74%,因此这两种卷积核尺寸能够反映常规卷积神经网络特征,因而对其进行耗时分析。本节实验以前向推理 FP16 数据类型、NHWC 存储布局为例进行测试,并随机抽取 100 条实验结果用于展示。实验在 K 版本硬件上开展。图 14、图 15 分别展示了 1×1 卷积核与 3×3 卷积核 convGen 最优算子耗时和官方算子耗时。

在 1×1 的卷积核尺寸下,convGen 性能约为官方手写算子库性能的 98%;在 3×3 的卷积核尺寸下,convGen 性能可达官方算子性能的 99%。可见 convGen 生成的卷积算子在常见卷积核参数下,其性能已经接近官方算子性能。

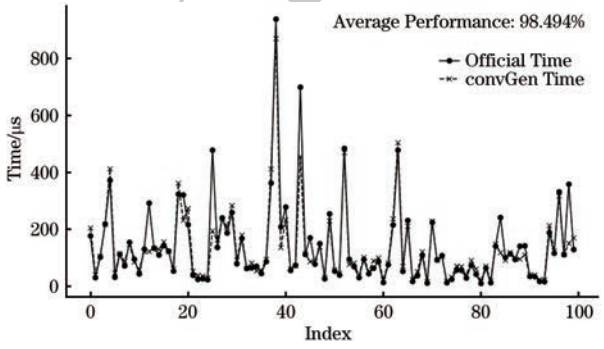

图 14 convGen 算子与官方耗时对比(1×1 卷积核)

Fig. 14 Comparison of convGen Operator and Official Time (1×1 convolutional kernel)

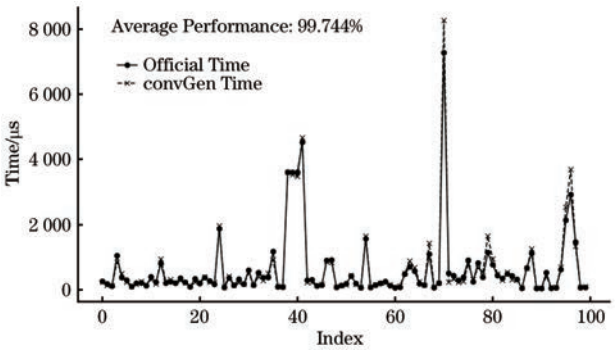

图 15 convGen 算子与官方耗时对比(3×3 卷积核)

Fig. 15 Comparison of convGen operator and official time (3×3 convolutional kernel)

4.5.2 整体性能测试

本节对 convGen 生成的卷积算子进行测试。实验分别基于 K 硬件版本和 B 硬件版本加速器进行,测试了 NCHW 和 NHWC 两种数据布局、FP32 和 FP16 两种数据类型的算子性能,这 4 种情况能够基本覆盖当前主流网络的卷积问题种类。K 版本硬件下测试结果如图 16 和图 17 所示,B 版本硬件

下测试结果如图 18 和图 19 所示。图中,纵坐标表示 convGen 最优算子耗时与国产加速库中算子耗时之比。

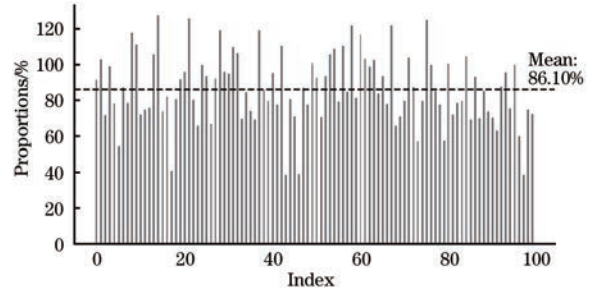

图 16 K 硬件版本下算子性能(NHWC、FP16)

Fig. 16 Operator performance under K hardware version (NHWC、FP16)

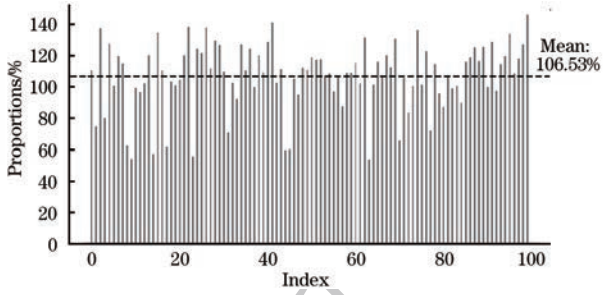

图 17 K 硬件版本下算子性能(NHWC、FP32)

Fig. 17 Operator performance under K hardware version (NHWC, FP32)

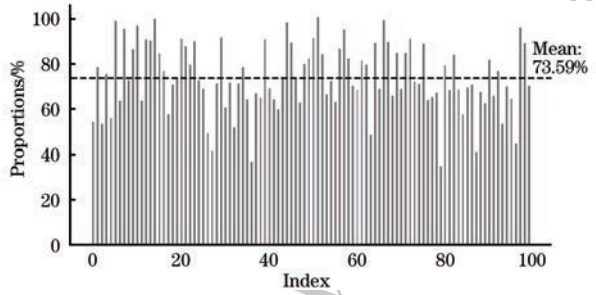

图 18 B 硬件版本下算子性能(NHWC、FP16)

Fig. 18 Operator performance under B hardware version (NHWC, FP16)

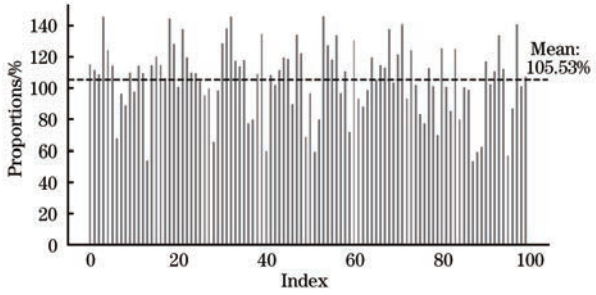

图 19 B 硬件版本下算子性能(NHWC、FP32)

Fig. 19 Operator performance under B hardware version (NHWC, FP32)

在 NHWC 存储布局下,K 版本和 B 版本 FP16 类型的卷积算子性能分别是官方算子库中卷积算子(下文简称“官方算子”)性能的 86.10%和 73.59%,而 FP32 类型的卷积性能是官方算子性能的 106.53%和 105.53%,都已经超过官方算子性能。不同数据类型的算子性能存在差异。分析发现,FP16 性能不及官方算子的原因是,官方算子针对 FP16 类型进行了重排优化,将两个 FP16 数据重新排列并转换为一个 FP32 数据,从而有效减少了与内存相关的指令数量,并且由于数据重排方案,改变了共享内存的写入与访问的线程结构,显著降低了共享内存的 bank 冲突率[31],从而提升了算子性能。因此,后续 convGen 也将引入相关优化方案。

NCHW 的数据布局在两种硬件版本下生成的算子性能都较差,这是因为受卷积滑动原理的限制,隐式卷积算法映射后,线程访问的数据在全局内存中常为非连续存储,使得无法进行向量化访存。官方算子多采用组合计算策略,将 NCHW 布局转换为 NHWC 布局后进行计算[32],从而规避了 NCHW 布局带来的访存性能问题。因此,后续 convGen 也将引入多算子组合计算方案。

不同硬件版本,算子性能也存在差异。分析发现计算单元(CU)核心数量的变化、底层指令的周期变化等硬件资源差异会改变算子的性能瓶颈。本文不再讨论硬件的差异对算子性能影响,未来将针对不同硬件瓶颈进行优化。

5 结束语

本文基于某国产加速器,设计并实现一种卷积代码生成器 convGen,并对其进行优化。通过配置模板参数,convGen 可生成特定的卷积算子,不仅能避免手写算子导致的重复性工作,还可以更灵活地探索和组合多种优化策略。convGen 采用了多种优化算法提升算子性能,内部兼容 K/B 两个硬件版本指令,支持生成两个平台的卷积算子。测试结果显示,在 NHWC 布局下,生成的算子在 K 和 B 两个硬件版本下平均性能可分别达到官方算子性能的 95%和 90%,能够满足实际使用的性能要求。未来将针对性能瓶颈对 convGen 进行优化,并扩展其对更多架构的兼容能力,使 convGen 不仅持续兼容国产加速器的未来硬件版本,未来还计划尝试将其移植到 NVIDIA 等其他通用异构计算平台。

参考文献

[1] ZHAO G, SUN N W, SHEN S, et al. GPU-accelerated

target strength prediction based on multiresolution shooting and bouncing ray method[J]. Applied Sciences, 2022, 12(12): 6119.

[2] GOLOSIO B, VILLAMAR J, TIDDIA G, et al. Runtime construction of large-scale spiking neuronal network models on GPU devices[J]. Applied Sciences, 2023, 13(17): 9598.

[3] HU Y X, LIU Y H, LIU Z Y. A survey on convolutional neural network accelerators: GPU, FPGA and ASIC[C]// Proceedings of the 14th International Conference on Computer Research and Development (ICCRD). Shenzhen, China: IEEE Press, 2022: 100-107.

[4] CHEN Y P, DAI X Y, LIU M C, et al. Dynamic convolution: attention over convolution kernels[C]// Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). Seattle, USA: IEEE Press, 2020: 11027-11036.

[5] KWON H, CHATARASI P, SARKAR V, et al. Maestro: a data-centric approach to understand reuse, performance, and hardware cost of dnn mappings[J]. IEEE Micro, 2020, 40(3): 20-29.

[6] XIE X R, LIN J, WANG Z F, et al. An efficient and flexible accelerator design for sparse convolutional neural networks[J]. IEEE Transactions on Circuits and Systems I: Regular Papers, 2021, 68(7): 2936-2949.

[7] JIA Z, MAGGIONI M, STAIGER B, et al. Dissecting the NVIDIA volta GPU architecture via microbenchmarking[EB/OL]. [2024-10-05]. http://arxiv.org/abs/1804.06826.

[8] ALWAN E H, KETRAN R M, HUSSEIN I A. A comprehensive survey on loop unrolling technique in code optimization[J]. Journal of University of Babylon for Pure and Applied Sciences, 2024, 32(1): 108-117.

[9] DAGHAGHI S, MEISBURGER N, ZHAO M N, et al. Accelerating slide deep learning on modern CPUs: vectorization, quantizations, memory optimizations, and more[EB/OL]. [2024-10-05]. http://arxiv.org/abs/2103.10891.

[10] 庞文豪, 王嘉伦, 翁楚良. GPGPU 和 CUDA 统一内存研究现状综述[J]. 计算机工程, 2024, 50(12): 1-15.
PANG W H, WANG J L, WENG C L. Survey on GPGPU and CUDA unified memory research status[J]. Computer Engineering, 2024, 50(12): 1-15. (in Chinese)

[11] 曹义魁, 陆忠华, 张鉴, 等. 面向国产加速器的 CFD 核心算法并行优化[J]. 数据与计算发展前沿, 2021, 3(4): 93-103.
CAO Y K, LU Z H, ZHANG J, et al. Parallel optimization of CFD core algorithms based on domestic processor[J]. Frontiers of Data & Computing, 2021, 3(4): 93-103. (in Chinese)

[12] BARCA G M J. COMP4300/8300 parallel systems introduction to GPU architecture & programming[EB/OL]. [2024-10-05]. https://comp.anu.edu.au/courses/comp4300/assets/lectures/Lecture_GPUs_partI.pdf.

[13] SUN W, LI A, GENG T, et al. Dissecting tensor cores via microbenchmarks: latency, throughput and numeric behaviors[J]. IEEE Transactions on Parallel and Distributed Systems, 2023, 34(1): 246-261.

[14] SUN W, LI A, STUIJK S, et al. How much can we gain from tensor kernel fusion on GPUs?[J]. IEEE Access, 2024, 12: 126135-126144.

[15] JIANG Z H, ZHENG L M, YAN E, et al. TVM: an automated optimizing compiler for deep learning[C]// Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). [S. l.]: USENIX, 2018: 578-594.

[16] KATEL N, KHANDELWAL V, BONDHUGULA U. MLIR-based code generation for GPU tensor cores[C]// Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction. New York, USA: ACM Press, 2022: 117-128.

[17] TRIPATHI M. Analysis of convolutional neural network based image classification techniques[J]. Journal of Innovative Image Processing, 2021, 3(2): 100-117.

[18] 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. New York, USA: ACM Press, 2024: 1072-1081.

[19] HIGHAM N J, MARY T. Mixed precision algorithms in numerical linear algebra[J]. Acta Numerica, 2022, 31: 347-414.

[20] XU R, MA S, GUO Y. Performance analysis of different convolution algorithms in GPU environment[C]//Proceedings of the IEEE International Conference on Networking, Architecture and Storage (NAS). Chongqing, China: IEEE Press, 2018: 1-10.

[21] SHEVGUNOV T, EFIMOV E, GUSCHINA O. Estimation of a spectral correlation function using a time-smoothing cyclic periodogram and FFT interpolation—2N-FFT algorithm[J]. Sensors, 2022, 23(1): 215.

[22] 童敢, 黄立波. Winograd 快速卷积相关研究综述[J]. 计算机科学与探索, 2022, 16(5): 959-971.
TONG G, HUANG L B. Review of Winograd fast convolution technique research[J]. Journal of Frontiers of Computer Science and Technology, 2022, 16(5): 959-971. (in Chinese)

[23] NAKASATO N. A fast GEMM implementation on the cypress GPU[J]. ACM SIGMETRICS Performance Evaluation Review, 2011, 38(4): 50-55.

[24] 李茂文, 曲国远, 魏大洲, 等. 面向 GPU 计算平台的神经网络卷积性能优化[J]. 计算机研究与发展, 2022, 59(6): 1181-1191.
LI M W, QU G Y, WEI D Z, et al. Performance optimization of neural network convolution based on GPU platform[J]. Journal of Computer Research and Development, 2022, 59(6): 1181-1191. (in Chinese)

[25] KORCH M, RAITHEL P, WERNER T. Implementation and optimization of a 1D2V PIC method for nonlinear kinetic models on GPUs[C]//Proceedings of the 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP). Västerås, Sweden: IEEE Press, 2020: 30-37.

[26] ZACHARIADIS O, SATPUTE N, GÓMEZ-LUNA J, et al. Accelerating sparse matrix-matrix multiplication with GPU Tensor Cores[J]. Computers & Electrical Engineering, 2020, 88: 106848.

[27] MARKIDIS S, DER CHIEN S W, LAURE E, et al. NVIDIA tensor core programmability, performance & precision[C]//Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).

Vancouver, Canada: IEEE Press, 2018: 522-531.

[28] NEMATOLLAHI N, SADROSADATI M, FALAHATI H, et al. Efficient nearest-neighbor data sharing in GPUs[J]. ACM Transactions on Architecture and Code Optimization, 2021, 18(1): 1-26.

[29] BASSO P M, DOS SANTOS F F, RECH P. Impact of tensor cores and mixed precision on the reliability of matrix multiplication in GPUs[J]. IEEE Transactions on Nuclear Science, 2020, 67(7): 1560-1565.

[30] WILLEMSEN F J, SCHOONHOVEN R, FILIPOVIČ J, et al. A methodology for comparing optimization algorithms for auto-tuning[J]. Future Generation Computer Systems, 2024, 159: 489-504.

[31] 刘仲, 李程, 田希, 等. MVSim: 面向 VLIW 多核向量处理器的快速、可扩展和精确的体系结构模拟器[J]. 计算机工程与科学, 2024, 46(2): 191-199.
LIU Z, LI C, TIAN X, et al. MVSim: a fast, scalable and accurate architecture simulator for VLIW multi-core vector processors[J]. Computer Engineering & Science, 2024, 46(2): 191-199. (in Chinese)

[32] ITO Y, NAKANO K. A GPU implementation of dynamic programming for the optimal polygon triangulation[J]. IEICE Transactions on Information and Systems, 2013, E96.D(12): 2596-2603.

文字编辑 金胡考
栏目编辑 宋 圆