计算机科学
COMPUTER SCIENCE

面向深度学习编译器TVM的算子融合优化

高伟, 王磊, 李嘉楠, 李帅龙, 韩林

引用本文

高伟, 王磊, 李嘉楠, 李帅龙, 韩林. 面向深度学习编译器TVM的算子融合优化[J]. 计算机科学, 2025, 52(5): 58-66.

GAO Wei, WANG Lei, LI Jianan, LI Shuailong, HAN Lin. Operator Fusion Optimization for Deep Learning Compiler TVM[J]. Computer Science, 2025, 52(5): 58-66.

相似文章推荐（请使用火狐或 IE 浏览器查看文章）

Similar articles recommended (Please use Firefox or IE to view the article)

基于大语言模型的刑事案件智能判决研究

Research on Intelligent Judgment of Criminal Cases Based on Large Language Models

计算机科学, 2025, 52(5): 248-259. https://doi.org/10.11896/jsjkx.241100100

基于TVM 的变维批处理小矩阵乘法的加速及应用

Accelerating Batched Matrix Multiplication for Variable Small Sizes Based on TVM andApplications

计算机科学, 2025, 52(5): 25-40. https://doi.org/10.11896/jsjkx.240500052

一种基于TVM的自动调度搜索优化方法

Automatic Scheduling Search Optimization Method Based on TVM

计算机科学, 2025, 52(3): 268-276. https://doi.org/10.11896/jsjkx.240100126

基于混合并行的分布式训练优化研究

Study on Distributed Training Optimization Based on Hybrid Parallel

计算机科学, 2024, 51(12): 120-128. https://doi.org/10.11896/jsjkx.231200128

基于数据局部性的循环分块选择算法

Tile Selection Algorithm Based on Data Locality

计算机科学, 2024, 51(12): 100-109. https://doi.org/10.11896/jsjkx.231100060

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.240100018

面向深度学习编译器 TVM 的算子融合优化

高 伟[1] 王 磊[1,2] 李嘉楠[1,2] 李帅龙[1,2] 韩 林[1]
1 国家超级计算郑州中心(郑州大学) 郑州 450001
2 郑州大学计算机与人工智能学院 郑州 450001
(yongwu22@126.com)

摘 要 算子融合是深度学习编译器中的一种编译优化技术,能够将多个算子合并为一个大的算子,有效降低计算和访存的成本。深度学习编译器 TVM 的算子融合方案中将算子按照功能特性进行分类,并设计融合规则,最后采用贪心算法进行融合。这种融合方案存在以下问题:首先,功能特性的算子分类方式下的融合规则不够通用,会错失算子融合机会,无法实现更大粒度的融合;其次,贪心的融合算法也无法实现算子融合的最优解。针对上述问题,对 TVM 进行改进,提出按照算子输入输出映射类型的算子分类方式,并设计通用的算子融合规则以扩大算子融合的粒度;提出基于动态规划的融合方案搜索算法和算子融合代价评估模型,并对搜索空间进行剪枝,使得算法能够在合理时间内搜索得到优化的融合方案。为评测融合方案的有效性,在 CPU 以及 DCU 等平台上对 VGG-16,Efficient-B0,MobileNet-V1,YOLO-V4 等深度学习模型的融合比和推理时延进行测试,实验结果表明,相较于 TVM 原有融合方案,所提方案融合比平均提升了 27%,推理时延平均获得了 1.75 的加速比。

关键词:深度学习编译器;TVM;算子融合;融合规则;动态规划
中图分类号 TP314

Operator Fusion Optimization for Deep Learning Compiler TVM

GAO Wei[1],WANG Lei[1,2],LI Jianan[1,2],LI Shuailong[1,2] and HAN Lin[1]
1 National Supercomputing Center in Zhengzhou(Zhengzhou University),Zhengzhou 450001,China
2 School of Computer and Artificial Intelligence,Zhengzhou University,Zhengzhou 450001,China

Abstract Operator fusion technique is an optimization method employed by deep learning compilers to combine multiple operators into a single,larger operator. This approach effectively reduces computation costs and memory access requirements. In the operator fusion scheme of deep learning compiler TVM,operators are categorized based on their functional characteristics,fusion rules are devised,and a greedy algorithm is utilized for fusion. However,this fusion scheme has the following problems. Firstly,the fusion rules derived from functional feature classification may not be sufficiently generalizable,leading to missed opportunities for operator fusion and limited granularity. Secondly,the greedy algorithm fails to achieve optimal solutions for operator fusion. To address these issues,improvements are made in TVM by introducing an operator classification method based on input/output mapping types and designing a more comprehensive set of fusion rules that expand the granularity of operator fusion. Additionally,a search algorithm for finding suitable fusion schemes and a cost evaluation model based on dynamic programming are proposed to prune the search space and enable efficient identification of optimal solutions within reasonable time frames. To evaluate the effectiveness of this enhanced fusion scheme,experiments are conducted using popular deep learning models such as VGG-16,Efficient-B0,MobileNet-V1 and YOLO-V4 on both CPU and DCU platforms. The experimental results show that compared with the original fusion scheme of TVM,the fusion ratio of deep learning models can be improved. The average fusion ratio is increased by 27%,and the average inference delay rate is 1.75.

Keywords Deep learning compiler,TVM,Operator fusion,Fusion rule,Dynamic programming

1 引言

随着深度学习在计算机视觉、自然语言处理等领域的广泛应用,深度学习模型的规模和复杂度均在不断增长。为满足不同场景下的应用部署需求,硬件加速器的种类也日渐繁多,当前广泛使用的深度学习框架,如 TensorFlow[1],Py-

到稿日期:2024-01-02 返修日期:2024-06-20
基金项目:河南省重大科技专项"国产先进计算平台创新生态及应用研究"(221100210600)
This work was supported by the Henan Province Major Science and Technology Project "Domestic Advanced Computing Platform Innovation Ecology and Application Research"(221100210600).
通信作者:韩林(strollerlin@163.com)

Torch[2]等,都难以很好地解决深度学习模型和硬件加速器的软硬件组合爆炸问题;同时,深度学习框架对于深度学习模型的执行往往采用调用已有高性能算子库的方式,无法支持新算子且无法进行更多的优化。为此,深度学习编译器[3-4]被提出,旨在将深度学习框架描述的模型转换为在各种硬件平台上能够高效运行的代码,从而实现软硬件之间的解耦,并提供更多优化机会。深度学习编译器的编译流程中,首先将深度学习模型转换为编译器内部的图中间表示(其中图中的每一个节点表示一个算子),并进行图优化。然后,模型的图中间表示会再变换为张量级中间表示,最后转换为代码实现。整个转换过程中,深度学习编译器能够结合模型以及硬件体系结构等特征信息进行高度优化,提升模型的执行效率。MLIR[5]和 TVM[6]是当前两种具有代表性的深度学习编译器。

算子融合是深度学习编译器 TVM 中的一个重要优化。它将模型转换后的图中间表示结合模型以及硬件运算特征,对图中间表示中可以融合的算子进行判断并融合为一个更大的算子,来减小算子计算和访存的开销。深度学习编译器中的算子融合优化通常采用基于规则的方式实现,即首先按照算子的功能进行分类,同时采用贪心算法对图中间表示中邻近的算子进行融合。按算子功能特性进行分类的方式,会使多数算子被划分为无法融合的类别,加之融合规则不够通用,从而无法实现粒度更大的融合。此外,基于贪心思想的融合策略也无法保证算子融合方案是最优解。

为解决上述问题,本文提出了新的算子融合编译优化方案,并基于开源深度学习编译器 TVM 进行实现。首先算子按照输入输出间的映射类型进行分类;然后基于该分类方式设计融合规则并构建出算子融合的搜索空间,在对搜索空间进行剪枝后,建立算子融合代价模型来评估融合代价;最后使用基于动态规划的融合方案搜索算法筛选出优化的算子融合方案。本文主要的贡献如下:

1)对算子融合的算子分类及融合规则进行改进,提出了按照算子输入输出映射的分类方式和融合规则;

2)提出了一种新的算子融合代价评估模型和基于动态规划的融合策略搜索算法;

3)通过扩展开源深度学习编译器 TVM,设计实现了基于映射的算子融合搜索方案,并进行了测试和分析,验证了算子融合搜索方案的有效性。

本文第 2 章介绍了研究背景及研究动机;第 3 章介绍了改进的算子融合映射分类分式以及融合规则;第 4 章介绍了基于动态规划的算子融合搜索算法以及评估模型;第 5 章给出了实验结果,并对实验结果进行了分析;最后总结全文。

2　背景与研究动机

本章介绍了本文工作的研究背景和算子融合优化的相关工作,并给出了本文工作的研究动机。

2.1　深度学习编译器 TVM

TVM 是一款开源的深度学习编译器,编译流程如图 1 所示。其采用了 Relay IR[7]和 TensorIR[8]两级中间表示,其中 Relay IR 是基于计算图的中间表示,可以进行算子融合等图优化,之后被向下转换为 TensorIR 的张量级中间表示,进行张量、循环等更细粒度级别的优化,通过多级中间表示可以覆盖更多优化范围。TVM 借鉴 Halide[9]的计算与调度分离的设计理念,还引入了自动调度机制。在用户给定调度的搜索空间后,自动调度机制会用实际运行的性能数据作为反馈来指导下一个调度的搜索方向,最终提升模型性能。最后,统一中间表示将通过不同的编译后端来生成适用于相应设备的代码。

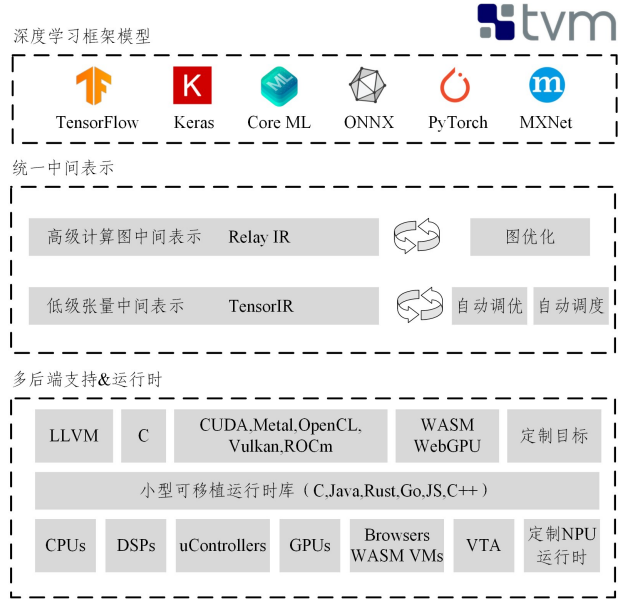


图 1　深度学习编译器 TVM 设计架构

Fig. 1　Design architecture of deep learning compiler TVM

2.2　算子融合优化

算子融合是深度学习编译器 TVM 中的一种图优化技术,在 Relay IR 这一基于计算图的中间表示上实现。计算图是一种用于表示模型计算逻辑和状态的有向无环图,包括节点和依赖边两部分,其中节点又可以分为数据节点、计算节点和控制节点,依赖边可以分为数据依赖和控制依赖。在计算图中,数据节点能够表示张量,计算节点能够表示算子,控制节点能够控制程序的执行顺序,节点之间的连接能够表示张量状态,也能够描述计算节点间的依赖关系。

算子融合将计算图作为输入,将多个算子合并为一个更大的算子后,输出分组融合后的计算图。其一般的实现过程如下:首先对计算图中的算子进行标记分组,其中标记为同一组的算子会被融合为一个新的算子,然后在代码生成过程中对组内的算子进行内联、分块、拆分和重排序等操作,以充分利用寄存器和缓存,存储计算结果实现内存复用,降低数据存储读写耗时,从而减少计算和内存访问的开销,提升 CPU、GPU、寄存器等设备资源的利用率。算子融合的作用与传统循环融合类似,包括消除冗余中间结果的实例化、减少不必要的输入扫描以及实现其他优化机会等。

算子融合优化由最初的手工融合发展为现阶段的自动融合。手工融合指在硬件厂商开发的高性能算子库中,手动将多个算子进行融合,即首先识别热点算子组合,然后通过手工方式实现这些组合对应的融合后算子,并将其注册到算子库中,最后添加优化将匹配的热点算子组合替换为融合算子。而自动融合相较于手工融合而言省去了手工识别算子组合和

重写融合后算子的步骤，主要聚焦于代码生成前的算子标记分组步骤，具体为将算子进行分类并设计基于经验定义的融合规则后，根据融合规则遍历计算图生成算子融合候选项，利用启发式算法进行融合标记。

2.3 相关工作

深度学习编译器 TVM，XLA[10] 等采用基于规则的算子融合方案，即基于特定规则采用启发式算法搜索计算图中相邻的可融合算子进行融合，最后为融合后算子进行代码生成。此外，一些工作对传统的规则融合进行了一定的改进，TASO[11] 提出了自动生成规则的方式，DNNFusion[12] 将算子按照映射类型进行分类，定义算子类别间的融合规则；针对基于规则的算子融合方案中的融合搜索过程，MetaFlow[13] 通过回溯法进行搜索，OCGGS[14] 提出了基于剪枝的动态规划的搜索算法，Optimus[15] 提出了算子融合内存代价模型来指导算子融合。

从硬件性能角度看，算子融合主要解决深度学习处理器所面临的内存墙和并行墙问题。上述算子融合的工作主要通过对计算图上存在数据依赖的“生产者-消费者”算子进行融合，从而提升中间张量数据的访存局部性，以此解决内存墙问题。AStitch[16]，Apollo[17]，RAMMER[18] 等工作被提出来解决并行墙问题，即通过向下拆解算子，将计算过程中互不依赖的低运算量子图打包为一个 Kernel 并行计算以提升资源利用率。

上述工作中，TVM 中的算子融合变换对算子的属性要求相对严格，因此错过了一些可以融合的优化机会；同时，TVM 以及 DNNFusion 的融合搜索算法都是贪心算法，只能得到局部最优解，MetaFlow，OCGGS，Optimus 虽然能够得到全局最优解，但其代价模型都基于运行时融合算子的实际测试结果，需要针对不同硬件重新测试，不具有泛化能力且实现较为复杂。为解决算子融合并行墙问题而提出的 Astitch，Apollo，RAMMER 等打破了算子边界，降低了编译器的耦合性。

2.4 研究动机

深度学习编译器 TVM 将算子分为 Injective，Reduction，Complex-out，Broadcast，Tuple 以及 Opaque 6 种类型，分别表示一对一映射、归约、计算复杂、广播、元组以及无法融合算子。从算子的类型中可以看出，TVM 的算子分类标签涉及到算子功能、映射关系、计算规模、操作数据等多种类别，这种混合标签使得算子无法得到完全的统一分类，而让更多算子被归到无法融合类型中。此外，在基于规则融合的方式下，这些不同标签分类下的融合规则往往无法覆盖到更多的融合情况，例如针对 MatMul+Reshape+Transpose+Add 等的算子组合形式，TVM 无法完成融合。

此外，TVM 采用基于支配树的算子融合搜索算法，即通过构建支配树来表示计算图中的算子依赖关系，然后利用贪心策略来选择最优的算子融合方案。在搜索过程中，算法会逐步合并相邻的算子，以减少计算和通信的开销，并且保证融合后的计算图依然是有效的。基于贪心思想的融合搜索算法只能保证实现局部最优解，而无法得到全局更优的算子融合方案。

为解决上述问题，本文提出按照算子输入输出间的映射类型进行分类，然后基于该分类方式设计更为通用的融合规则，进而构建出算子融合方案的搜索空间。在对搜索空间进行剪枝后，建立算子融合代价模型来评估融合代价，最后使用基于动态规划的融合方案搜索算法筛选出优化后的算子融合方案。本文的算子融合方案能够增大算子融合的机会，更充分地利用硬件算子资源，提升模型的推理性能。

3 算子映射融合优化

算子映射融合优化的核心流程如图 2 所示，可以分为创建计算图、划分计算图和重写计算图 3 个步骤。

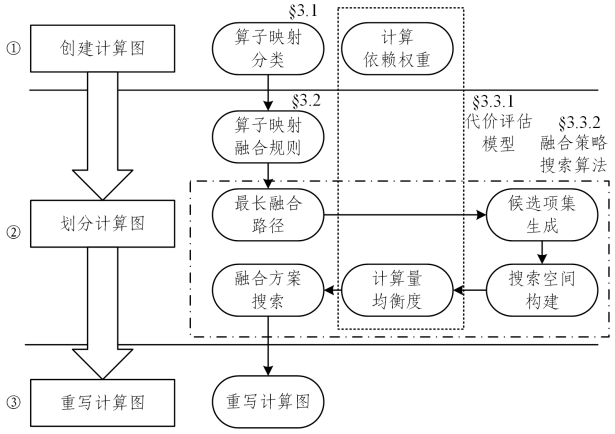


图 2 算子映射融合优化的核心流程

Fig. 2 Core flow of operator mapping fusion optimization

在创建计算图阶段，遍历的过程中需要根据算子定义中的映射类型分类，为计算图中的节点添加映射分类属性信息。同时，为应用算子融合代价评估模型，还需要根据算子输入输出的张量类型及形状等信息计算内存大小，作为计算图中依赖边的权重。

在划分计算图阶段，主要通过划分计算图的方式找到最优融合策略。本文提出了基于动态规划的融合策略搜索算法。首先基于创建计算图阶段的算子映射分类下的融合规则，生成计算图中算子的最长融合路径，然后生成融合候选项集来构造算子映射融合方案的搜索空间。针对搜索空间中的融合方案计算相应的计算量均衡度，并与创建计算图阶段得到的依赖权重一同构成算子融合代价评估模型，用于指导融合方案的搜索。

最后一个步骤是重写计算图，基于搜索到的融合策略，需要将融合后的算子在计算图层面进行重写，以便后续的进一步优化操作。

3.1 算子映射分类

本节首先介绍按照算子输入输出映射关系对算子进行分类的方法，随后基于这种分类方式分析不同类型算子之间的融合收益，进而确定算子融合规则。

3.1.1 算子映射分类方式

按照算子输入输出映射类型的不同可以将算子划分为一对一、一对多、多对多、重组、乱序等类别，同时将不属于这些类别的算子划分至无法融合类型。由于算子可能存在具有多个不同映射类型的情况，因此需要对这些映射类型的复杂度排序。当算子具有多个映射类型时，该算子的映射类型由更复杂的映射类型决定。算子映射类型的复杂程度由低到高依次是一对一、重组、乱序、一对多以及多对多类型。

下面将依次对上述各种类型的划分依据及代表性算子进行介绍。假设算子的输入为 $x[d_1,d_2,\cdots,d_n]$,其中 d_1,d_2,…,d_n 分别表示输入操作数 x 中每一维的元素。

1)一对一类型。一对一类型指对于操作数 x 的每一维度 d_n,经函数 $f_n(d_n)$ 计算后均存在一一映射的输出,算子输出如式(1)所示。此类别下的代表性算子包括 add 算子以及 ReLU 等激活函数算子等。

$$y[d_1,d_2,\cdots,d_n]=F_{1-1}(x[f_1(d_1),\cdots,f_n(d_n)]) \tag{1}$$

2)重组类型。重组类型指对于操作数 x 的每一维度 d_n,其输出是经函数 $f_n(d_n)$ 计算后的元素重新组合而来,算子输出如式(2)所示。此类别下的代表性算子包括 Reshape 算子等。

$$y[d_1,d_2,\cdots,d_n]=x[f_1(d_1),\cdots,f_n(d_n)] \tag{2}$$

3)乱序类型。乱序类型指对于操作数 x 的每一维度 d_n 经函数 $F(n)$ 置换,其输出是置换后对应维度的元素重新组合而来,算子输出如式(3)所示。此类别下的代表性算子包括 Transpose 转置算子等。

$$y[d_1,d_2,\cdots,d_n]=x[f_1(d_{F(1)}),\cdots,f_n(d_{F(n)})] \tag{3}$$

4)一对多类型。一对多类型指对于操作数 x 的每一维度 d_n,经函数 $f_n(d_n)$ 计算后会存在多个维度相应的输出,因此输出的维度大于输入维度,算子输出如式(4)所示,其中 e_1,e_2,…,e_m 分别表示输出 y 中每一维的元素,且 $m>n$。此类别的代表性算子包括 Expand 维度扩张算子等。

$$y[e_1,e_2,\cdots,e_m]=F_{1-M}(x[f_1(d_1),\cdots,f_n(d_n)]) \tag{4}$$

5)多对多类型。多对多类型包含多对一类型,指对于操作数 x 的多个维度元素,经一个或多个函数计算后会存在一个或多个维度的输出,算子输出如式(5)所示。此类别下的代表性算子包括 Conv2d 卷积算子等。

$$y[e_1,e_2,\cdots,e_m]=F_{M-M}\{x^1[f_1^1(d_1),\cdots,f_n^1(d_n)],\cdots,x^k[f_1^k(d_1),\cdots,f_n^k(d_n)]\} \tag{5}$$

3.1.2　算子映射分类实现

深度学习编译器 TVM 中算子的定义和注册都使用 Python 编写,而对应算子的实现在其提供的 TOPI 算子库中,TOPI 算子库中的算子使用 C++或 Python 编写,算子的定义和实现之间通过 Python 绑定相连接调用。因此,对于算子映射的分类仅需要在 TVM 的定义中添加属性即可。如图 3 所示,为实现映射分类属性的可重用,首先需要将算子的映射分类属性定义至属性头文件中;然后将 5 种类别算子映射类型按复杂程度由低到高指定属性值,并将无法融合类型的属性值置为最高;最后根据算子定义中的输入输出映射关系进行算子分类,并在算子注册时添加映射分类属性。

```
tvm/include/tvm/relay/op_attr_types.h
enum OpMapPatternKind {
        OneToOne = 0,
        Reorganize = 1,
        Shuffle = 2,
        OneToMany= 3,
        ManyToMany = 4,
        NotFuse = 5
};
using MapOpPattern = int;

//relay.nn.conv2d
RELAY_REGISTER_OP("nn.conv2d")
……
.set_attr<MapOpPattern>("MapOpPattern", ManyToMany)
```

图 3　算子映射分类实现示例

Fig. 3　Example of operator mapping classification implementation

3.2　算子映射融合规则

按算子映射类型进行分类后,不同类型的算子融合完成后并不一定会带来正收益,因此需要对不同类型间的融合情况进行分析,将能实现正收益的组合方式作为算子融合规则。

3.2.1　算子映射融合收益分析

1)一对一类型+任意类型。一对一类型与任意类型融合过程中,将一对一类型算子的输出作为后继任意类型算子的输入。由于一对一类型的映射已知,对后继任意类型算子输入的内存访问可被映射至一对一类型算子输入的内存访问,从而缩短了内存访问时间,因此融合后能够带来正收益。

2)重组/乱序类型+任意类型。重组和乱序两种类型都是一对一类型的变体,因此与后继任意类型的算子进行融合也会带来正收益。但重组和乱序类型会改变数据复制和访问的数据,因此与后继为一对多和多对多类型的算子融合时,需要判断一对多和多对多类型的算子是否需要以连续的内存访问顺序复制和访问输入张量,如果是则无法在该融合过程中产生正收益。

3)一对多类型+多对多类型。一对多类型与后继为多对多类型的算子融合过程中,一对多类型的算子通常是将其输入张量通过复制等方式对张量维度进行扩充后得到输出,而这一过程会影响后继多对多类型算子以连续的内存访问方式进行读取,因此无法带来正收益。

4)多对多类型+多对多类型。多对多类型与后继为多对多类型的算子融合,由于会涉及多个输入输出的读取访问,进而会加大寄存器和缓存的访问压力,往往不会产生正收益。多对多类型中的多对一算子与后继为一对多类型的算子融合后,由于数据访问模式是不确定的,因此是否带来正收益需要结合算子进行进一步具体分析。

3.2.2　算子映射融合规则设计

依据上述不同映射类型算子间融合情况的收益分析,可以将其中始终能带来正收益的组合作为融合规则,并且将其中需要具体分析的情况作为规则的补充。融合完成后的算子可能仍会继续与其他算子进行融合,因此需要确定融合完成后算子的类型。考虑到算子映射类型的复杂程度与算子融合难易呈正相关,以及工程实现的易操作性等因素,可将融合前算子中复杂程度最大的映射类型作为融合后的算子映射类型。最终整理得到的算子融合规则如表 1 所列。

表 1　算子映射融合规则

Table 1　Operator mapping fusion rules

规则	算子 1 类型	算子 2 类型	算子 1 和 2 融合后类型
1	一对一	任意类型	任意类型
2	重组	一对一/重组/乱序	重组/重组/乱序
3	乱序	一对一/重组/乱序	乱序
补充 1	重组/乱序	一对多/多对多	一对多/多对多
补充 2	多对多(多对一)	一对多	多对多(多对一)

3.3　基于动态规划的融合策略搜索

本节介绍了基于动态规划的融合策略搜索算法,首先介绍算子融合代价评估模型,然后基于 3.2.2 节介绍的算子映射融合规则生成融合方案,最后从融合方案构成的搜索空间中选择使代价评估模型最小的方案,完成算子融合策略的搜索。

3.3.1 算子映射代价评估模型

通过算子融合将多个算子融合为一个大的算子后，小算子之间传输的数据会变为大算子内部的中间计算结果，这些中间结果可以利用寄存器和缓存进行存储，从而降低数据存储访问以及计算的开销，进而使用融合后算子间输入输出变量的内存大小近似估计算子融合的内存访问开销和计算开销。融合后算子间输入输出变量的内存大小在融合后的计算图中可以表示为不同融合组别间箭头上的权重之和。计算过程可以形式化地表示为式(6)：

$$FusedCost_{\mathrm{Mem}}=\sum_{i=0}^{N_1-1}\sum_{j\in next(i)}\omega_{ij} \tag{6}$$

其中，ω_{ij} 表示融合后计算图节点 i 与其后继节点 j 间的权重，即传输的变量内存大小；N_1 为融合后计算图节点总数。

此外，当模型规模较大时，通常需要在多核或多机设备上并行执行，这时算子之间计算量越均衡，模型的执行效率越高，因此可以使用融合算子组间算子个数的方差去近似估计融合后算子计算量的均衡性。计算过程可以形式化地表示为式(7)：

$$FusedCost_{Par}=\frac{1}{N_2}\sum_{i=0}^{N_2-1}(n_i-\bar{n}) \tag{7}$$

其中，n_i 表示融合组别 i 融合的算子总数，$\bar{n}$ 为所有融合组别平均融合的算子数，N_2 为融合组别总数。基于上述因素，综合内存和并行建立的代价模型如式(8)所示。同时，考虑到两种代价量化程度不一，需要结合模型规模和硬件平台确定归一化因子 α 和 β。

$$FusedCost=\alpha\cdot FusedCost_{\mathrm{Mem}}+\beta\cdot FusedCost_{\mathrm{Par}} \tag{8}$$

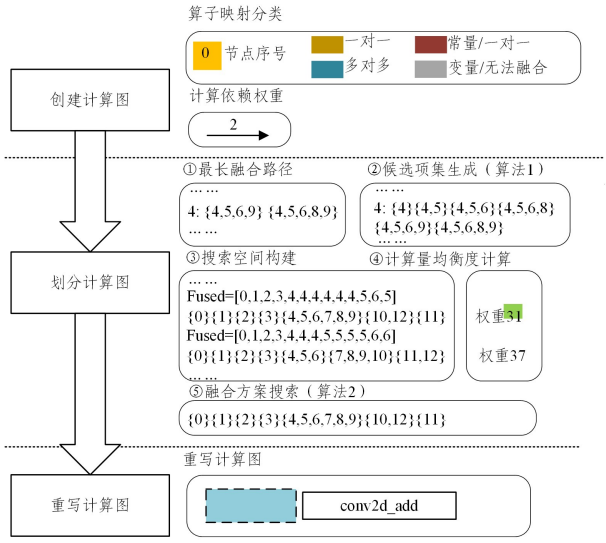
3.3.2 融合策略搜索算法

融合策略搜索算法的流程如图 4(a)所示，包括最长融合路径生成、候选项集生成、搜索空间构建、计算量均衡度计算以及融合方案搜索等步骤。本小节将整个流程划分为融合候选项集构建和融合方案生成两个部分，融合候选项集构建包括最长融合路径生成和候选项集生成等步骤，融合方案生成包括搜索空间构建、计算量均衡度计算以及融合方案搜索等步骤。

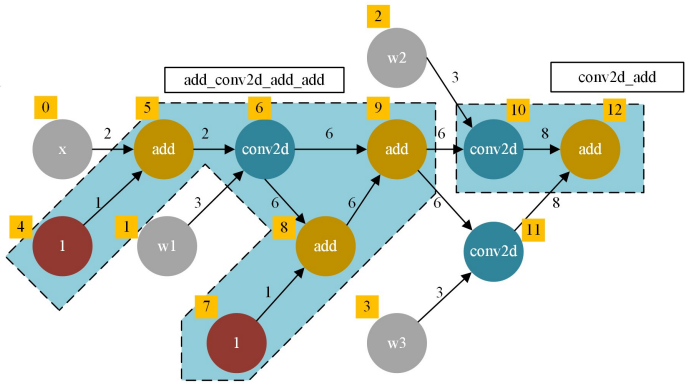
同时，通过图 4(b)所示的一个简单的算子映射融合示例来演示算法的运行。结合图 4(a)所示的算子映射融合核心流程，在运行融合策略搜索算法前，需要对计算图中的节点进行映射类型的分类，并计算依赖权重。其中节点的分类用不同颜色表示，依赖权重用节点间箭头上的数字表示。然后基于分类和权重计算后的计算图进行后续的融合方案搜索流程。

1)融合候选项集构建

融合候选项集指可以相互融合的算子组成的集合，这些集合能够构成融合方案的搜索空间，因此在进行全局融合方案搜索前需要构建融合候选项集。融合候选项集的构建需要对计算图进行深度优先遍历，具体过程如算法 1 所示。首先需要将计算图的每个节点都作为初始节点通过 DFS()函数进行深度优先遍历，并通过函数 JudgeFuseRule()判断是否符合 3.2.2 节的算子融合规则，如果符合则继续依次向后进行遍历，否则停止向后遍历，并输出以当前节点开始的一条可融合路径；然后回退以寻找其他最长可融合路径；最后取每个节点最长可融合路径的前缀作为融合候选项集。



(a)融合流程图



(b)融合示例计算图

图 4 算子映射融合流程示例

Fig. 4 Example of operator mapping fusion process

图 4 所示的算法运行示例中，从序号为 4 的一对一类型常量节点开始的可融合路径共有{4,5,6,9}和{4,5,6,8,9}两条。当找到各节点的所有向后最长可融合路径后，这些可融合路径的前缀子集均是整个计算图的融合候选项，如节点 4 的融合路径{4,5,6,9}中前缀{4}，{4,5}等均可构成融合方案的搜索空间，因此取这些可融合路径的所有前缀子集即可构成节点 4 的融合候选项集。

算法 1 融合候选项集构建

输入：计算图 G

输出：候选项集 R

```
1. def DFS(start,path,visited,G)
2.     visited[start]=true
3.     path.push_back(start)
4.     if(start==G.size()-1)||(G.node[start].pattern==Not-
```

```
    Fuse)
5.    /* 遇到不可融合节点 */
6.      paths.push_back(path)
7.    else
8.      for now in G.node[start].next
9.      /* 当前节点与所有后继节点的融合 */
10.       if !visited[now]
11.         if JudgeFuseRule( G.node[start],G.node[now])
12.           /* 算子映射融合规则判断 */
13.             path.push_back(now)
14.           else
15.             paths.push_back(path)
16.           end if
17.           continue
18.           DFS(now,path,visited,G)
19.         end if
20.       end for
21.     end if
22.     path.pop_back()
23.     /* 回退以考虑其他融合路径 */
24.     visited[start]=false
25./* 全图深度优先搜索最长融合路径 */
26.for node_id in G.size()
27.     DFS(node_id,path,visited,G)
28.  end for
29./* 取各节点最长融合路径构成候选项集 */
30.R=getAllPrefixSubsets(paths)
```

2）融合方案搜索

基于融合候选项集和算子融合代价模型，融合方案搜索指在融合候选项集组成的搜索空间中，通过基于动态规划的算子融合策略搜索算法筛选出使算子融合代价最小的方案，流程如算法 2 所示。首先使用函数 Init()初始化融合参数，算法中使用 Fused 数组表示融合方案，数组索引与计算图中节点索引相对应，数组元素为大于等于－1 的整数，当值为－1 时表示对应节点未融合，当值为其他值时表示融合组别序号，相同的值即代表可融合至同一组。融合搜索从索引为 0 的节点开始，融合组别序号从 0 开始编号。图 4 所示的融合方案数组 Fused=[0,1,2,3,4,4,4,5,5,5,5,6,6]中，索引 4，5，6 的组别序号均为 4，即对应计算图中索引序号为 4，5，6 的节点可融合为一个算子。

之后，调用函数 Group_Fuse_Search()从索引为 0 的节点开始搜索，搜索过程中如果遇到不可融合节点，则将该节点单独作为一个融合组别。然后调用函数 Group_Fuse_Search()递归地为其他节点搜索融合方案。图 4 所示的融合方案数组 Fused=[0,1,2,3,4,4,4,5,5,5,5,6,6]中，索引 0 到 3 对应的节点均为无法融合类型，因此其分别被单独划分至一个组中，即代表无法融合。

当模型规模较大时，融合候选项集所组成的搜索空间非常大，为缩短搜索时间，需要对搜索空间进行剪枝，即针对每个可融合节点，通过函数 findLongestPath()找到该节点的最长融合候选项集，将其划分至一个融合组别后，对计算图中剩余未融合节点调用函数 Other_Fuse_Search()进行融合。剩余节点融合过程中优先从候选项集中筛选最长的候选项进行融合，直至所有节点均被设置了融合组别。剩余未融合节点的融合方案搜索过程中，为了避免总是优先选择开始索引序号较小的候选项集，算法每次都会从随机选择的索引序号开始，向后循环搜索。一次融合方案生成后，由于可能存在融合后算子类型为一对一类型，还可继续与其他类型算子融合的情况，因此还需要调用函数 Group_Fuse_Second()进行二次融合判断。

融合方案生成后需要根据 3.3.1 节介绍的代价模型通过函数 WeightFuse()计算该方案的权重代价，最后在所有融合方案中选择代价函数最小的方案。图 4 所示的融合方案数组 Fused=[0,1,2,3,4,4,4,4,4,4,5,6,5]中，代价函数中融合组别间变量内存大小权重的计算则是对计算图中非融合组别内部边的权重进行求和得到，选择使代价函数最小的融合方案。最后将融合组别内的算子通过重写计算图的方式，使其表示为大的算子，从而完成算子融合优化，进行后续的编译优化过程。

算法 2　融合方案搜索算法

输入：计算图 G，候选项集 R

输出：融合策略 F

```
1.    /* 算法由两个函数 Group_Fuse_Search()和 Other_Fuse_
      Search()的定义以及主程序 Group_Fuse_Search() 3 部分组成
      */
2.def Group_Fuse_Search(G,Result,fused,start,group_id)
3.      /* 不可融合节点进行单独融合 */
4.      if G.node[start].pattern==NotFuse
5.        fused[start]=group_id
6.        /* 从下一节点开始搜索融合方案 */
7.        Group_Fuse_Search(G,R,fused,++start,++group_id)
8.    else
9.        /* 搜索可融合节点的融合方案 */
10.       for node_id in UnfuseNode(fused)
11.         for id in findLongestPath(R,fused,node_id)
12.       /* 融合最长融合路径上的所有节点 */
13.           fused[id]=group_id
14.       end for
15.       /* 搜索剩余未融合节点的融合方案 */
16.         Other_Fuse_Search(G,R,fused,++group_id)
17.         /* 回退以考虑其他融合方案 */
18.         for id in findLongestPath(R,fused,node_id)
19.           fused[id]=-1
20.       end for
21.       group_id--
22.     end for
23.   end if
24.def Other_Fuse_Search(G,R,fused,group_id)
25./* 随机选择未融合节点融合候选项融合 */
26.   for node_id in findUnfusedPath(R,fused)
27.     fused[node_id]=group_id
28.   end for
29.   if FindUnfused(fused)==true
30.     Other_Fuse_Search(G,R fused,++group_id)
```

```
31.   else
32.     /*二次融合判断*/
33.     Group_Fuse_Second(G,R,fused)
34.     end if
35.   /*评估代价模型计算及筛选*/
36.   if bestWeight > WeightCount(fused)
37.     F=fused
38.     bestWeight=WeightCount(fused)
39.   end if
40.   for node_id in findUnfusedPath(R,fused)
41.     fused[node_id]=-1
42.   end for
43.   group_id--
44./*初始化*/
45.Init(Fused,G.size(),-1)
46.Group_Fuse_Search(G,Result,fused,0,0)
```

4 实验评估

本章介绍针对系统设计与实现的实验评估，并对实验结果进行分析。4.1节介绍实验评估的要素与方案；4.2节介绍实验的环境、数据集以及模型；4.3节讨论实验结果，并对实验结果进行分析。

4.1 评估要素

为评估算子映射分类下的融合规则以及基于动态规划的融合策略搜索算法等组成的算子融合策略的有效性，将本文方法与TVM原融合规则及搜索算法进行融合前后中间表示大小、模型层数变化以及推理时间等要素的变化对比。

1)中间表示大小。算子融合优化前后中间表示的大小能够从宏观上体现优化效率。在实验评估过程中，记录融合前的中间表示大小为 I_0，算子融合优化后的中间表示大小为 I_1，针对每个测试模型，计算其中间表示大小相对变化的压缩比 C 为：

$$C=\frac{I_0}{I_1} \tag{9}$$

当压缩比大于1时表示融合方案的优化是有效的，当压缩比大于TVM原有融合方案的压缩比时，表示从宏观上说明改进是有效的。

2)模型层数。与中间表示大小相比，算子映射融合前后的模型层数变化能从微观上体现优化效率。在实验评估过程中，记录融合前的模型层数为 L_0，算子融合优化后的模型层数为 L_1，针对每个测试模型，计算其模型层数相对变化的融合比 F 为：

$$F=\frac{L_0}{L_1} \tag{10}$$

采用融合比要素评估整体内部的细节优化情况。

3)推理时间。算子映射融合能够有效减少模型层数，降低Kernel访存开销，且融合后可以减少调用的Kernel数量和调用总时长，因此也能显著影响模型的推理时间。为了验证本文提出的映射算子融合优化的通用性和平台无关性，除了上述基于CPU平台的测试外，也在Hygon DCU处理器上对模型优化前后的推理时间进行了测试，并对性能加速比进行评估。在实验评估过程中，记录融合前的模型推理时间为 T_0，算子融合优化后的模型推理时间为 T_1，针对不同平台下的每个测试模型，计算其模型推理时间相对变化的性能加速比 S 为：

$$S=\frac{T_0}{T_1} \tag{11}$$

4.2 实验环境与方案

本文采用Hygon C86处理器为测试平台，使用PyTorch中的预训练卷积神经网络VGG-16，Efficient-B0，MobileNet-V1，YOLO-V4等在ImageNet数据集上对TVM的算子融合优化进行测试评估，TVM的版本为0.12.dev0。整个测试评估过程主要是对应用算子映射融合优化前后模型的层数变化、中间表示大小以及推理时间等要素进行比较，同时通过与TVM中原有算子融合优化效果进行对比，来验证优化效果。

4.3 实验结果与分析

本节基于4.1节介绍的中间表示大小、模型层数变化以及推理时间等评估要素，通过实验对算子映射融合优化的优化效果进行评估，给出实验结果并进行分析。

4.3.1 中间表示大小

表2列出了VGG-16，Efficient-B0，MobileNet-V1，YOLO-V4等模型应用算子融合优化前后的中间表示大小的变化情况，并统计了压缩比。从表中可以看出，测试模型中压缩比最高为Efficient-B0模型的3.09倍，模型大小由优化前的108 MB压缩至35 MB；压缩比最低为YOLO-V4模型的1.50倍，模型大小由优化前的329 MB压缩至220 MB，模型的平均压缩比为2.36。此外，应用算子映射融合后的模型中间表示大小均显著减小。实验结果表明，算子映射融合这种算子融合方式是可行的。算子映射融合优化前后中间表示片段对比如图5所示。

```
// 算子映射融合前的中间表示片段
……
%24 = nn.conv2d(%23, %aten::_convolution_7.weight, padding=[1, 1, 1, 1], channels=512, kernel_size=[3, 3])
/* ty=Tensor[(1, 512, 28, 28), float32] span=aten::_convolution_7:0:0 */;
 %25 = nn.bias_add(%24, %aten::_convolution_7.bias)
/* ty=Tensor[(1, 512, 28, 28), float32] span=aten::_convolution_7:0:0 */;
 %26 = nn.relu(%25)
/* ty=Tensor[(1, 512, 28, 28), float32] span=aten::relu__7:0:0 */;
……
```

```
// 算子映射融合优化后的中间表示片段
……
%52 = fn (%p012: Tensor[(1, 256, 28, 28), float32] /* ty=Tensor[(1, 256, 28, 28), float32] */, %p18: Tensor[(512, 256, 3, 3), float32] /* ty=Tensor[(512, 256, 3, 3), float32] */, %p27: Tensor[(512), float32] /* ty=Tensor[(512), float32] */, Primitive=1) -> Tensor[(1, 512, 28, 28), float32] {

%16 = nn.conv2d(%p012, %p18, padding=[1, 1, 1, 1], channels=512, kernel_size=[3, 3])
/* ty=Tensor[(1, 512, 28, 28), float32] span=aten::_convolution_7:0:0 */;
%17 = nn.bias_add(%16, %p27)
/* ty=Tensor[(1, 512, 28, 28), float32] span=aten::_convolution_7:0:0 */;
nn.relu(%17)
 /* ty=Tensor[(1, 512, 28, 28), float32] span=aten::relu__7:0:0 */

} /* ty=fn (Tensor[(1, 256, 28, 28), float32], Tensor[(512, 256, 3, 3), float32], Tensor[(512), float32]) -> Tensor[(1, 512, 28, 28), float32] */;
……
```

图5 算子映射融合优化前后中间表示片段对比

Fig. 5 Comparison of intermediate representation segments before and after operator mapping fusion optimization

图5中，在没有进行算子融合的情况下，每个算子的输出

都需要写入全局内存；而在算子映射融合的情况下，conv2d，bias_add 以及 relu 等多个算子组合被融合至一个函数中，所有的输出可以合并为一个输出，从而减小了中间表示的大小。

表 2　算子映射融合优化中间表示大小及压缩比对比

Table 2　Operator mapping fusion optimizes intermediate representation size and compression ratio comparison

模型	优化前/MB	优化后/MB	压缩比
VGG-16	161	66	2.44
Efficient-B0	108	35	3.09
MobileNet-V1	110	46	2.39
YOLO-V4	329	220	1.50

4.3.2　模型层数

为基于模型层数优化前后的相对变化来评估整体内部的细节优化情况，图 6 给出了 VGG-16，Efficient-B0，MobileNet-V1，YOLO-V4 等模型在应用 TVM 原有融合方案以及本文提出的不同融合规则和融合算法的组合方案下的融合比，其中纵坐标表示相对融合比，横坐标表示不同模型下的优化组合方案，$Rule_T$ 表示 TVM 原有融合规则，$Rule_M$ 表示本文提出的算子映射融合规则，ALG_{greedy} 表示 TVM 原有基于支配树的融合策略贪心搜索算法，ALG_{DP} 表示本文提出的基于动态规划的融合策略搜索算法。深蓝色图例表示未优化，融合比为 1；橙色图例 $Rule_T+ALG_{greedy}$ 的组合表示深度学习编译器 TVM 中的原有算子融合方案；灰色图例 $Rule_M+ALG_{greedy}$ 的组合表示用本文提出的映射融合规则替换 TVM 原有融合规则后的方案；黄色图例 $Rule_T+ALG_{DP}$ 的组合表示用本文提出的搜索算法替换 TVM 原有搜索算法后的方案；蓝色图例表示本文提出的映射融合规则和搜索算法的组合方案。

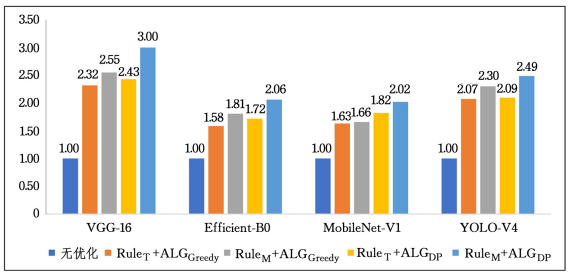


图 6　算子映射融合优化模型层数融合比对比（电子版为彩图）

Fig. 6　Comparison of operator mapping fusion optimization mode layer fusion ratio

观察 4 种模型的测试结果可知，仅使用映射规则替换相较于 TVM 的原有融合规则方案的融合比平均提升了 13%左右，表明映射规则能够覆盖更多的算子融合情况；仅使用基于动态规划的搜索算法替换除了在 YOLO-V4 模型中融合比无明显提升外，在其他模型中的融合比均有提升。通过分析后发现，YOLO-V4 在 TVM 原有融合规则下使用原有的贪心算法搜索已近似最优解的融合方案，因此无明显提升，这说明本文提出的搜索算法在 TVM 原有规则和本文提出的映射融合规则两种情况下，均能找到近似最优解的融合方案。同时，使用本文提出的映射融合规则和基于动态规划的搜索算法对 TVM 算子融合方案进行完全替换后，融合比最高为 VGG-16 的 3.00 倍，最低为 MobileNet-V1 的 2.02 倍。与 TVM 原方案相比，融合比均有明显提升，平均提升约 27%。

4.3.3　推理时间

为验证优化与平台无关，图 7 和图 8 分别给出了 VGG-16，Efficient-B0，MobileNet-V1，YOLO-V4 等模型应用本文提出的算子映射融合方案后在 CPU 和 DCU 上的推理时间。本文方法在 CPU 上的性能加速比最高为 Efficient-B0 的 1.95 倍，最低为 MobileNet-V1 的 1.60 倍，平均性能加速比为 1.75，相比 TVM 原有融合方案分别提升了约 30%。在 DCU 上的性能加速比最高为 Efficient-B0 的 1.75，最低为 YOLO-V4 的 1.38，平均性能加速比为 1.51，相比 TVM 原有融合方案分别提升了约 22%。分析结果可知，映射算子融合优化只聚焦于平台无关的计算图层面，优化与平台无关，具有通用性。

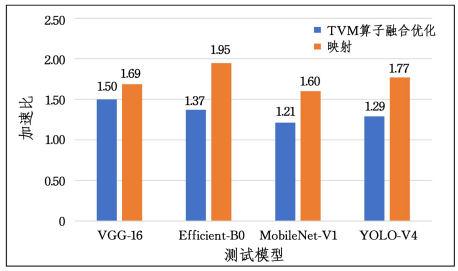


图 7　算子映射融合优化 CPU 端推理时间加速比

Fig. 7　Operator mapping fusion optimizes CPU-side inference time speedup

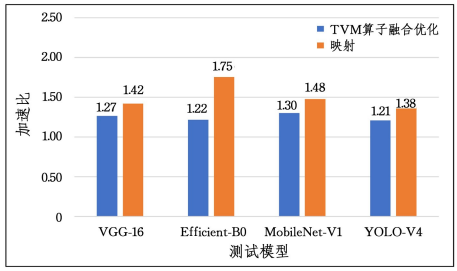


图 8　算子映射融合优化 DCU 端推理时间加速比

Fig. 8　Operator mapping fusion optimizes DCU-side inference time speedup

算子映射融合优化带来的推理性能提升主要有以下两方面的原因。一方面是内存读写开销的减小。算子映射融合通过增大融合机会来减小中间结果的生成，进而降低中间表示的大小。当使用具有多层缓存结构的 CPU 等硬件进行模型推理时，融合算子内的数据往往位于缓存中，提高了缓存命中率，同时也减少了融合前各算子间所需的数据传输次数，降低了内存访问次数和内存消耗量。另一方面是硬件并行性的提升。算子映射融合减少了模型层数，能够合并多个算子为更大的任务单元，与未融合前分属单独的计算任务相比，有效降低了计算任务间切换的开销。同时，由于深度学习的计算密集型计算任务多为矩阵循环运算，因此算子映射融合能够实现对循环的融合，减小并行调度的循环数量，使硬件对计算任务实现更粗粒度的执行，提升 CPU 等硬件资源的利用率。

结束语　本文提出了新的算子融合编译优化方案，包括映射融合规则和新的基于动态规划的搜索算法，并基于开源

深度学习编译器 TVM 进行实现，优化了算子融合这一优化过程，提升了模型执行的效率。同时，算子融合优化针对的是深度学习编译器中平台无关的计算图优化过程，对于编译器前后端的不同框架模型以及硬件加速器均可适用，具有一定的通用性。未来的工作是对方案中搜索算法的搜索时间进行进一步压缩，同时由于当前基于规则的融合中，融合规则仍是基于经验和专业知识去设计的，尚未覆盖更多的融合情况，因此基于强化学习的融合规则设计也是未来算子融合的优化方向之一。

参 考 文 献

[1] ABADI M, BARHAM P, CHEN J, et al. TensorFlow: a system for large-scale machine learning[C]// 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). Savannah, GA, USA, Berkeley, CA, USA: USENIX Association, 2016: 265-283.

[2] PASZKE A, GROSS S, MASSA F, et al. Pytorch: an imperative style, high-performance deep learning library[J]. Advances in Neural Information Processing Systems, 2019, 32: 8026-8037.

[3] LI M Z, LIU Y, LIU X Y, et al. The deep learning compiler: a comprehensive survey[J]. IEEE Transactions on Parallel and Distributed Systems, 2021, 32(3): 708-727.

[4] ZHANG H B, XING M J, WU Y J, et al. Compiler technologies in deep learning co-design: a survey[J/OL]. https://spj.science.org/doi/10.34133/icomputing.0040.

[5] LATTNER C, AMINI M, BONDHUGULA U, et al. MLIR: a compiler infrastructure for the end of Moore's law[J]. arXiv: 2002.11054, 2020.

[6] CHEN T Q, MOREAU T, JIANG Z H, et al. TVM: an automated end-to-end optimizing compiler for deep learning[C]// 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). Carlsbad, CA, USA, Berkeley, CA, USA: USENIX Association, 2018: 578-594.

[7] ROESCH J, LYUBOMIRSKY S, WEBER L, et al. Relay: a new IR for machine learning frameworks[C]// Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2018). Philadelphia PA, USA, New York, NY, USA: Association for Computing Machinery, 2018: 58-68.

[8] FENG S Y, HOU B H, JIN H Y, et al. TensorIR: an abstraction for automatic tensorized program optimization[C]// Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS 2023). Vancouver, BC, Canada, New York, NY, USA: Association for Computing Machinery, 2023: 804-817.

[9] RAGAN-KELLEY J, BARNES C, ADAMS A, et al. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines [C]// Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13). Seattle, WA, USA, New York, NY, USA: Association for Computing Machinery, 2013: 519-530.

[10] SNIDER D, LIANG R F. Operator fusion in XLA: analysis and evaluation[J]. arXiv: 2301.13062, 2023.

[11] JIA Z H, PADON O, THOMAS J, et al. TASO: optimizing deep learning computation with automatic generation of graph substitutions[C]// Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19). Huntsville, Ontario, Canada, New York, NY, USA: Association for Computing Machinery, 2019: 47-62.

[12] NIU W, GUAN J X, WANG Y Z, et al. DNNFusion: accelerating deep neural networks execution with advanced operator fusion[C]// Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021), Vitural Canada. New York, NY, USA: Association for Computing Machinery, 2021: 883-898.

[13] JIA Z H, THOMAS J, WARSZAWSKI T, et al. Optimizing DNN computation with relaxed graph substitutions [J]. Proceedings of Machine Learning and Systems, 2019, 1: 27-39.

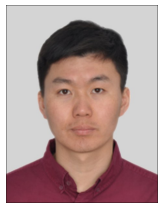
[14] FANG J Z, SHEN Y Y, WANG Y, et al. Optimizing DNN computation graph using graph gubstitutions[J]. Proceedings of the VLDB Endowment, 2020, 13(12): 2734-2746.

[15] CAI X Y, WANG Y, ZHANG L. Optimus: an operator fusion framework for deep neural networks[J]. ACM Transactions on Embedded Computing Systems, 2022, 22(1): 1-26.

[16] ZHENG Z, YANG X D, ZHAO P Z, et al. AStitch: enabling a new multi-dimensional optimization space for memory-intensive ML training and inference on modern SIMT architectures[C]// Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2022). Lausanne Switzerland, New York, NY, USA: Association for Computing Machinery, 2022: 359-373.

[17] ZHAO J, GAO X, XIA R J, et al. Apollo: automatic partition-based operator fusion through layer by layer optimization[J]. Proceedings of Machine Learning and Systems, 2022, 4: 1-19.

[18] MA L X, XIE Z Q, YANG Z, et al. RAMMER: enabling holistic deep learning compiler optimizations with rtasks[C]// 14th USENIX Conference on Operating Systems Design and Implementation (OSDI 20). Berkeley, CA, USA: USENIX Association, 2020: 881-897.



GAO Wei, born in 1985, Ph.D, associate professor. His main research interest is high performance computing.

HAN Lin, born in 1978, Ph.D, professor. His main research interests include high performance computing, advanced compilation, parallel programming and optimization.

（责任编辑：何杨）