TensorFlow 中 OpenCL 核函数的实现与优化

陈 锐 孙羽菲 程大果 郭 强 陈禹乔 石昌青
隋轶丞 张宇哲 张玉志
(南开大学软件学院 天津 300450)

摘 要 目前,异构计算技术已经被广泛应用于人工智能领域,旨在利用以 GPGPU 为主的并行加速设备和 CPU 协同工作,更高效地完成大规模的并行计算.深度学习模型的构建、训练以及推理离不开机器学习框架的支持,但目前主流的机器学习框架基本仅支持 CUDA 异构编程模型.CUDA 的私有性和封闭性导致机器学习框架严重依赖于英伟达 GPGPU.众多其它厂商的硬件加速器,尤其是国产加速器难以充分发挥其在深度学习中的潜力.使用开源统一异构编程标准 OpenCL 代替私有的 CUDA 编程模型,是打破这一技术壁垒的有效方法.本文提出了 TensorFlow 中 CUDA 到 OpenCL 核函数的代码转换方案,总结整理了核函数转换的基本规则、典型难点问题的解决方法以及 OpenCL 核函数的性能优化等关键技术.本文首次完成了 TensorFlow 2.2 版本中 135 个 OpenCL 核函数的实现.经一系列测试验证,转换生成的 135 个 OpenCL 核函数能够在多种支持 OpenCL 标准的加速器上正确运行,优化后,近八成的 OpenCL 核函数在英伟达 Tesla V100S 上达到了与 CUDA 核函数相当的计算性能.测试结果验证了本文提出的 CUDA 到 OpenCL 核函数转换方案的通用性及有效性,包含 OpenCL 核函数的 TensorFlow 版本能够在直接适配跨厂商加速器设备的同时保持较好的计算性能.

关键词 硬件加速器;异构编程环境;CUDA;OpenCL;TensorFlow
中图法分类号 TP312 **DOI 号** 10.11897/SP.J.1016.2022.02456

Implementation and Optimization of OpenCL Kernels in TensorFlow

CHEN Rui SUN Yu-Fei CHENG Da-Guo GUO Qiang CHEN Yu-Qiao SHI Chang-Qing
SUI Yi-Cheng ZHANG Yu-Zhe ZHANG Yu-Zhi
(*College of Software, Nankai University, Tianjin* 300450)

Abstract Heterogeneous computing technology has been widely used in artificial intelligence, aiming to use GPGPU-based parallel acceleration devices and CPUs to work together to complete large-scale artificial intelligence parallel computing tasks more efficiently. Deep learning models cannot be built, trained, and inferred without the support of machine learning frameworks. However, today's mainstream machine learning frameworks could only support private, closed CUDA heterogeneous programming models, which leads them to rely heavily on NVIDIA GPGPUs. Many other vendors' hardware accelerators, especially domestic ones, struggle to realize their full potential in deep learning. The use of the open-source unified heterogeneous programming standard OpenCL instead of the private CUDA programming model and the porting of mainstream machine learning frameworks to domestic hardware accelerators are of great importance in breaking the technical barriers of foreign countries in the field of machine learning frameworks and hardware

收稿日期:2021-11-30;在线发布日期:2022-05-30.本课题得到国家重点研发计划项目(2021YFB0300104)资助.**陈 锐**,博士研究生,主要研究方向为深度学习异构计算和高性能计算.E-mail:rzchen@mail.nankai.edu.cn.**孙羽菲**(通信作者),博士,特聘研究员,主要研究领域为深度学习与异构计算.E-mail:yufei_sun@sina.com.**程大果**,硕士研究生,主要研究方向为智能计算.**郭 强**,硕士研究生,主要研究方向为深度学习与高性能计算.**陈禹乔**,硕士研究生,主要研究方向为深度学习框架.**石昌青**,硕士研究生,中国计算机学会(CCF)会员,主要研究方向为深度学习与高性能计算.**隋轶丞**,博士研究生,主要研究方向为异构计算与深度学习算法.**张宇哲**,硕士研究生,主要研究方向为性能优化与并行计算.**张玉志**,博士,教授,中国计算机学会(CCF)会员,主要研究领域为人工智能.

accelerators, promoting the application of domestic hardware accelerators in the new generation of artificial intelligence and building a software ecology based on domestic acceleration chips. Considering that implementing OpenCL kernels is the core and fundamental work for adding a novel OpenCL backend to TensorFlow. This paper investigates the code conversion scheme for CUDA to OpenCL kernels in TensorFlow. Specifically, we summarize and organize the basic rules of CUDA to OpenCL kernel conversion, the typically complex problems encountered during the conversion process, and the solutions to the typical difficulties. Meanwhile, we discuss a series of techniques for performance optimization of the OpenCL kernels generated by the conversion method in this paper. Lastly, this paper shows a method for integrating calls to OpenCL kernels in TensorFlow. To the best of our knowledge, we are the first to implement 135 OpenCL kernels in TensorFlow version 2.2. Extensive experiments are conducted on the OpenCL kernels generated by the conversion method proposed in this paper. Correctness experiments in this paper show that 135 OpenCL kernels in the NVIDIA Tesla V100S accelerator card environment could agree with the corresponding CUDA kernels within a certain error range for different data input types. Performance experiments indicate that nearly 80% of OpenCL kernels could achieve comparable computational performance to CUDA kernels on NVIDIA Tesla V100S. By comparing the execution efficiency of OpenCL kernels before and after performance optimization, the optimization method presented in this paper could significantly improve the computational performance of the kernels. We have also tried to deploy the converted OpenCL kernels on other computing devices that are not based on the NVIDIA GPU architecture. The experimental results show that the converted OpenCL kernels run correctly on different computing devices with good generality. The evaluation results validate the generality and effectiveness of the CUDA to OpenCL kernel conversion scheme proposed in this paper, and the version of TensorFlow that includes OpenCL kernels can be directly adapted to cross-vendor accelerator devices while maintaining good computational performance. The study of CUDA to OpenCL kernel conversion and kernel optimization methods in machine learning frameworks in this paper can provide strong support for further exploration of automatic or semi-automatic kernel conversion techniques.

Keywords　hardware accelerator; heterogeneous programming environment; CUDA; OpenCL; TensorFlow

1　引　言

近年来，以深度学习为代表的人工智能算法和模型受到了广泛的关注与研究. 在图像识别、文本处理等多个领域中，深度学习取得突破性进展[1-2]. 与此同时，随着模型参数量以及训练所需数据规模的增长，完成模型训练和推理依赖更加强大的计算能力. 为了提升深度学习模型对计算设备的使用效率以及简化模型的构建与计算流程，国内外知名公司和科研机构设计实现了多种机器学习框架. 目前，以TensorFlow为代表的机器学习框架一般使用CPU+加速器的异构计算架构提升模型的计算速度[3-4]. 异构计算架构能够发挥 CPU 擅长调度管理的特点以及加速器在并行度、单机计算峰值等方面的优势. 相比于传统同构计算，异构计算架构计算性能的效率更高、延迟更低[5].

得益于英伟达 GPU 在数据计算中的高性能和完善的开发生态，TensorFlow、PyTorch[6] 等主流机器学习框架的官方版本基本采用闭源的 CUDA(Compute Unified Device Architecture)[7] 开发. 近年来，随着异构计算技术在人工智能中的应用日益广泛，AMD、英特尔等国外硬件厂商通过 ROCm(Radeon Open Compute)① 和不同 SYCL 实现[8-10] 进一步丰富了机器学习框架可兼容的计算设备种类. 然而，ROCm 主

① New AMD ROCm™ Information Portal - ROCm v4.5 and Above. https://rocmdocs.amd.com/en/latest/, 2021, 10, 31

要用于支持 AMD GPU，现有 SYCL 实现①仅能支持少数厂商的加速设备. 主流机器学习框架基本仅支持以国外厂商 GPU 芯片为主的计算设备，通用加速器尤其是国产加速器在新一代人工智能应用中的巨大潜力难以发挥. 国内外最新的深度学习研究成果[11-13]大多使用英伟达、AMD 等国外芯片公司的产品计算，人工智能技术的创新与发展已与这些芯片公司牢牢的绑定在了一起.

OpenCL(Open Computing Language)作为一个面向异构系统的通用开源编程标准，能够适用于多核 CPU、GPU、FPGA、DSP 等不同架构的计算设备[14]. 从表 1 中整理的常见开源机器学习框架不难发现，主流机器学习框架基本不支持 OpenCL 编程标准，也无法兼容国产加速硬件. 尽管谷歌、百度等公司推出的 TensorFlow Lite②、PaddlePaddle Lite③中包含 OpenCL 后端，但这些轻量级机器学习框架主要面向移动终端以及嵌入式设备，仅支持模型的推理计算. 需要消耗大量计算资源的模型训练场景中，仅有 Caffe[15]、Theano[16]等代码规模较小、流行度不高或已经停止更新的机器学习框架能够兼容通用加速器. 在当今国外对我国实行技术封锁、芯片禁运的背景下，利用 OpenCL 开源通用的特性，实现主流机器学习框架对 OpenCL 的支持，将主流机器学习框架移植到国产加速器上，对于打破国外在机器学习框架和硬件加速器领域的技术壁垒、推进国产加速器在新一代人工智能领域的应用以及打造基于国产加速芯片的软件生态，具有十分重要的意义.

表 1 常见开源机器学习框架基本信息汇总

框架名称	开发/维护者	应用场景	是否支持 OpenCL	可兼容的加速器
TensorFlow	Google	模型训练、推理	否	英伟达、AMD 等少数国外厂商加速器
MXNet[17]	Apache 基金会	模型训练、推理	否	英伟达、AMD 等少数国外厂商加速器
PyTorch	Facebook	模型训练、推理	否	英伟达、AMD 等少数国外厂商加速器
Caffe	BVLC	模型训练、推理	是	大部分厂商加速器
Caffe2④	Facebook	模型训练、推理	否	英伟达、AMD 等少数国外厂商加速器
Theano	MILA	模型训练、推理	是	大部分厂商加速器
CNTK[18]	微软	模型训练、推理	否	仅英伟达 GPU
PaddlePaddle[19]	百度	模型训练、推理	否	仅英伟达 GPU
TensorFlow Lite	Google	仅模型推理	是	移动终端及嵌入式设备等
PaddlePaddle Lite	百度	仅模型推理	是	移动终端及嵌入式设备等
PyTorch Mobile⑤	Facebook	仅模型推理	否	移动终端及嵌入式设备等

作为最主流也最具代表性的机器学习框架，本文对如何在 TensorFlow 2.2 版本中增加 OpenCL 后端进行了探索与研究. 主要聚焦于项目中最重要和核心的工作：TensorFlow 中 OpenCL 核函数的实现、集成以及优化.

从图 1 中不难看出，实现 OpenCL 核函数是在 TensorFlow 中添加 OpenCL 后端的核心和基础工作. 对于 TensorFlow 中适合在加速器上并行执行的计算，源码中的 CUDA 核函数是很好的参考. 因此，参照 TensorFlow 中已有 CUDA 实现对核函数进行转换来实现 OpenCL 核函数是更高效和明智的研发选择.

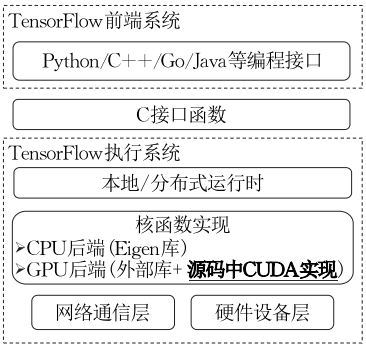


图 1 TensorFlow 框架结构示意图

本文的主要创新和贡献包括：

(1) 提出了一套 TensorFlow 下 CUDA 到 OpenCL 核函数代码的转换流程；整理了核函数转换时可能遇到的典型问题以及解决方法；首次实现了 TensorFlow 2.2 版本中总计 135 个核函数的转换.

(2) 总结并实现了 OpenCL 核函数在 TensorFlow

① SYCL Implementations in Development. https://www.khronos.org/sycl/, 2021,12,5

② Deploy machine learning models on mobile and IoT devices. https://www.tensorflow.org/lite, 2021,11,2

③ Paddle Lite: Multi-platform high performance deep learning inference engine. https://github.com/PaddlePaddle/Paddle-Lite, 2021,11,2

④ Caffe2: Portable High-Performance Deep Learning Framework from Facebook. https://developer.nvidia.com/blog/caffe2-deep-learning-framework-facebook/

⑤ End-to-end workflow from Training to Deployment for iOS and Android mobile devices. https://pytorch.org/mobile/home/, 2021,10,31

中的性能优化、集成调用以及测试方法.

(3)通过一系列的实验，验证了本文提出的方法有效可靠且经优化后，大部分 OpenCL 核函数在相同设备上的计算性能已与 CUDA 核函数相当.转换生成的 OpenCL 核函数具有良好的通用性，可在不同架构的计算设备中正确运行.

本文第 2 节介绍近几年 CUDA 向 OpenCL 转换的相关研究工作以及一些能够支持 OpenCL 后端的机器学习框架；第 3 节主要介绍本文提出的 TensorFlow 下 CUDA 到 OpenCL 核函数代码的转换流程和优化技术，包括基础核函数的转换方式、典型问题和难点问题的解决方法以及性能优化方法；为了对转换生成的 OpenCL 核函数进行测试，本文在第 4 节中介绍一种 OpenCL 核函数在 TensorFlow 中的集成调用方式；第 5 节设计的实验验证本文转换方法的正确性、优化的有效性以及在不同架构计算设备上的通用性；第 6 节对本文的工作进行总结并提出了未来的研究方向.

2　相关工作

2.1　CUDA 代码自动转换方法

近年来，随着不同硬件架构的计算设备被应用于数据计算，如何将基于 CUDA 的异构计算程序转换为通用的 OpenCL 程序受到了大量的关注.现有 CUDA 向 OpenCL 代码自动转换的方法一般使用编译器对 CUDA 代码的语法和接口函数进行解析，根据设定好的转换规则自动对抽象语法树或中间表示进行修改，从而完成代码转换的工作.本文在表 2 中展示了一些代表性的研究工作及其特性.

表 2　CUDA 到 OpenCL 自动转换工作整理

工作名称	输入	能否覆盖全部核函数	改动方式	能否适用于机器学习框架
CU2CL[20]	CUDA	否	基于语法树	否
SWAN[21]	CUDA	否	基于源码	否
CUDA to OpenCL[22]	CUDA	否	基于语法树	否
CUDA-on-CL[23]	CUDA	否	基于 LLVM IR	部分适用

Martinez 等人提出的 CU2CL[20]是一个基于 Clang 编译器的 CUDA 到 OpenCL 自动转换器.CU2CL 通过对 Clang 编译生成的语法树进行修改，重新生成了 OpenCL 代码，从而实现了代码的转换.Harvey 等人提出的 Swan[21]是一个 CUDA 源码到 OpenCL 源码的转换工具，包含了代码处理工具 Swan 以及接口库 libswan 库.在转换过程中首先利用 Swan 对 CUDA 代码进行解析，生成统一标准的 SWAN 代码.之后将 SWAN 代码与 libswan 的 OpenCL 后端进行链接，得到可执行的 OpenCL 代码，从而实现 CUDA 到 OpenCL 的转换.类似的，CUDA to OpenCL[22]也是针对源码的转换，主要基于 Cetus 编译框架实现.上述方法都是针对单个 CUDA 程序的转换，均已在 2017 年左右停止更新，仅能够支持 CUDA 5.0 以下版本.因此，对于较新 CUDA 版本中包含的核函数嵌套等新特性以及部分 C++实现上述自动转换方法都无法处理.

Perkins提出的CUDA-on-CL[23]首先通过 Clang 编译器将 CUDA 代码编译为 LLVM IR，之后使用专门设计的 OpenCL 生成器将 LLVM IR 转换成 OpenCL 代码.CUDA-on-CL 中的 OpenCL 生成器主要基于人工编写的 CUDA 到 OpenCL 的转换规则实现.该方法解决了在处理 C++特性上的部分限制并完成了对 TensorFlow 1.11 等低版本的转换，实现了可部署在 OpenCL 设备上的 tf-coriander①.

本文在相同软硬件环境下分别使用tf-coriander、TensorFlow 1.11(CPU)以及TensorFlow 1.11(GPU)对 tf-coriander 项目中的 8 个测试文件进行了测试.从表 3 展示的实验结果可以看出，该方法实现的部分算子性能很低，甚至远低于 CPU 的计算性能，不具有实用性.

表 3　Tf-coriander 项目测试结果

测试名称	Tf-coriander	TensorFlow 1.11 (CPU)	TensorFlow 1.11 (GPU)
linear_regression	**0.0352**	0.0059	0.0139
nearest_neighbor	**2.9140**	1.3894	1.7814
autoencoder	**18.3394**	1.8937	0.5652
multilayer_perceptron	**19.0856**	1.7902	1.3327
recurrent_network	**1.1037**	0.1234	0.0273
dynamic_rnn	**1.2331**	0.0795	0.0490
bidirectional_rnn	**1.3897**	0.1111	0.0283
convolutional	**0.0296**	0.0033	0.0003

综上所述，对于较新版本的 TensorFlow，已有 CUDA 到 OpenCL 的自动转换方法主要存在以下

① tf-coriander. https://github.com/pint1022/tf-coriander, 2021,11,2

三点限制：

(1)无法对 CUDA 核函数中使用的模板、类等 C++特性的代码进行转换；

(2)无法对 CUDA 核函数中可能包含的 cub、cuSparse 等其他库中的函数进行转换；

(3)自动转换后的 OpenCL 核函数在加速器上的性能和可用性无法保证，也难于优化.

2.2 支持 OpenCL 的机器学习框架

Theano 是由蒙特利尔大学 MILA 研发团队使用 Python 开发的机器学习框架，该框架中包含了部分基础的 OpenCL 实现. 然而，近年来使用 Theano 进行开发的应用并不多，该框架已于 2017 年停止更新.

OpenCL Caffe 是 AMD 研究团队基于 Caffe 开发的能够支持 OpenCL 的框架[24]，采用人工的方式将 Caffe 中 CUDA 核函数转换为 OpenCL 核函数. OpenCL Caffe 的实现过程包含了核函数转换以及性能调优两个阶段. 在核函数转换阶段，OpenCL Caffe 根据深度神经网络的模型结构，对部分模型中的层进行转换，其他部分采用 CPU 实现. 在性能调优阶段，通过减少 OpenCL 核函数编译次数、增加数据和任务并行度等方式提升了框架的整体性能. Caffe 的代码规模较小，支持的接口函数的数量也远远少于主流 TensorFlow 等框架且早已停更.

TensorFlow Lite 是一款可部署在移动终端以及嵌入式设备中执行推理运算的轻量级机器学习框架. 包含了 OpenGL、OpenCL 等后端，能够兼容多种计算设备. 与 TensorFlow Lite 相似，PaddlePaddle Lite 是由百度设计的一款轻量级框架，支持在 OpenCL、华为麒麟 NPU、华为昇腾 NPU 等多种硬件上部署. TensorFlow Lite 和 PaddlePaddle Lite 主要用于移动和嵌入式设备上的模型推理，尚不支持模型训练.

除了上述包含 OpenCL 后端的机器学习框架外，一些机器学习框架还尝试通过 SYCL 标准实现对 OpenCL 设备的兼容[25-26]. 然而，受限于无法支持 SPIR[27](Standard Portable Intermediate Representation)等编译 SYCL 代码过程中所需的关键依赖，现有 SYCL 实现仅支持少数特定 OpenCL 设备. TensorFlow 对 SYCL 的支持尚属于试验阶段，目前只能在 TensorFlow 较旧的版本中使用①.

基于对已有自动转换工作以及包含 OpenCL 后端的机器学习框架的分析与研究. 本文选择参考 TensorFlow 中已有 CUDA 核函数计算逻辑，采用人工转换的方式实现 TensorFlow 中的 OpenCL 核函数. 本文提出了一整套 TensorFlow 下 CUDA 代码到 OpenCL 代码的转换流程，解决了一系列核函数转换时可能遇到的典型问题以及对应的解决方法. 此外，本文对转换完成后 OpenCL 核函数计算性能的优化进行了深入的探索.

3 CUDA 到 OpenCL 核函数的转换

3.1 CUDA 与 OpenCL 的相关基础知识

CUDA 是英伟达公司针对英伟达 GPU 推出的并行计算架构，该架构包含了 CUDA 指令集架构(ISA)以及 GPU 内部的并行计算引擎. CUDA 的应用程序分为主机(host)端和设备(device) 端两个部分. 其中，主机端代码在 CPU 上执行，用于流程控制，设备端代码(又称为“核函数”)负责具体计算任务，在 GPU 上执行. TensorFlow 中的 CUDA 主机端以及核函数代码主要基于 C++开发.

OpenCL 是一个用于在异构平台中编写程序的开放标准. 与 CUDA 类似，OpenCL 应用程序也由主机端与核函数两部分组成. OpenCL 1.2 标准中，主机代码可以使用 C 或 C++编写，核函数代码仅能使用 C 语言编写. 在 OpenCL 2.0[28]及以上版本中核函数代码支持使用部分 C++特性开发. 由于目前能够支持 OpenCL 2.0 及以上的硬件种类不多，尤其是考虑到大部分国产硬件加速器主要支持 OpenCL 1.2 标准. 本文主要研究了将 CUDA 转换为 OpenCL 1.2 标准的核函数.

为了更加直观得对 CUDA 与 OpenCL 进行比较，表 4 中整理了 CUDA 与 OpenCL 在硬件术语、变量名称等相关概念中存在的对应关系.

3.2 CUDA 核函数转换为 OpenCL 核函数的基本流程

TensorFlow 源码包含的 CUDA 核函数数目众多、计算类型和功能多样，总结出一套标准转换流程能够极大地提升转换效率和代码质量，减少在转换过程中的代码编写错误. 本文基于对 TensorFlow 源码中 CUDA 核函数实现特点的研究与分析，将一般情况下 TensorFlow 中 CUDA 到 OpenCL 核函数的转换流程归纳为以下三步：

① codeplaysoftware/tensorflow. https://github.com/codeplaysoftware/tensorflow, 2021,12,5

表 4　CUDA 与 OpenCL 基本概念对照表

	CUDA	OpenCL
硬件术语对照	SM(Stream Multiprocessor)	CU(Compute Unit)
	Thread	Work-item
	Block	Work-group
	Global memory	Constant memory
	Shared memory	Local memory
	Private memory	Local memory
核函数限定符对照	__global__ function	__kernel function
	__device__ function	无直接对应
	__constant__ variable declaration	__constant variable declaration
	__device__ variable declaration	__global variable declaration
	__shared__ variable declaration	__local__ variable declaration
内建变量对照	gridDim	get_num_groups()
	blockDim	get_local_size()
	blockIdx	get_group_id()
	threadIdx	get_local_id()
	blockIdx * blockDim+threadIdx	get_global_id()
	gridDim * blockDim	get_global_size()
核函数同步接口对照	syncthreads()	barrier()
	threadfence()	无直接对应
	threadfence_block()	mem_fence()
	无直接对应	read_mem_fence()
	无直接对应	write_mem_fence()
主机端接口对照	cudaGetDeviceProperties()	clGetDeviceInfo()
	cudaMalloc()	clCreateBuffer()
	cudaMemcpy()	clEnqueueRead(Write)Buffer()
	cudaFree()	clReleaseMemObj()
	kernel〈〈〈…〉〉〉()	clEnqueueNDRangekernel()

(1) 核函数中 C++实现替换为 C 实现

TensorFlow 中的 CUDA 核函数基于 C++开发，而 OpenCL 1.2 标准中核函数代码只支持 C99 标准. 因此，对于 C++中 *reinterpret_cast*()等操作符需要进行替换. 对于部分包含结构体或类作为入参的 CUDA 核函数，需要将结构体或类里面的成员声明为 OpenCL 核函数的入参，类中的方法改写成能被 OpenCL 核函数直接调用的子函数.

(2) 核函数修饰符和接口函数替换

在核函数的转换过程中，需要基于表 4 中整理的 CUDA 与 OpenCL 的对应关系对 CUDA 核函数修饰符与接口函数进行替换. 例如，将 CUDA 核函数中的*__global__*限定符修改为 OpenCL 中的*__kernel*；将 CUDA 核函数中用于同步的 *syncthreads*()函数替换为 OpenCL 中的 *barrier*().

从图 2 中展示的对角生成矩阵转换示例中可以

```
__global__ void MatrixDiagKernel(
    const int num_threads, const int num_rows, const int num_cols,
    const int num_diags, const int max_diag_len, const int lower_diag_index,
    const int upper_diag_index, const T padding_value,
    const bool left_align_superdiagonal, const bool left_align_subdiagonal,
    const T* diag_ptr, T* output_ptr) {
  GPU_1D_KERNEL_LOOP(index, num_threads) {
    const int batch_and_row_index= index / num_cols;
    const int col = index -batch_and_row_index* num_cols;
    const int batch = batch_and_row_index/ num_rows;
    const int row = batch_and_row_index-batch*num_rows;
    const int diag_index= col-row;
    const int diag_index_in_input= upper_diag_index-diag_index;
    const int content_offset=
      ComputeContentOffset(diag_index, max_diag_len, num_rows, num_cols,
                           left_align_superdiagonal, left_align_subdiagonal);
    const int index_in_the_diagonal= col-max(diag_index, 0)+content_offset;
    if (lower_diag_index<= diag_index&&diag_index<= upper_diag_index){
      output_ptr[index] =
          diag_ptr[batch* num_diags* max_diag_len+
              diag_index_in_input* max_diag_len+ index_in_the_diagonal];
    } else {
      output_ptr[index] = padding_value;
    }
  }
}
```

```
string kernel_src="__kernel void MatrixDiagKernel(const int num_threads, const int num_rows,          \
                          const int num_cols, const int num_diags,          \
                          const int max_diag_len, const int lower_diag_index, \
                          const int upper_diag_index,          \
                          const " + tName+ " padding_value,          \
                          const int left_align_superdiagonal,          \
                          const int left_align_subdiagonal,          \
                          __global const " + tName+ "* diag_ptr,          \
                          __global " + tName+ "* output_ptr) {          \
  for(int index=get_global_id(0); index<num_threads; index+= get_global_size(0)){          \
    const int batch_and_row_index= index/num_cols;    \
    const int col=index-batch_and_row_index* num_cols;    \
    const int batch=batch_and_row_index/num_rows;    \
    const int row=batch_and_row_index-batch*num_rows;    \
    const int diag_index=col-row;    \
    const int diag_index_in_input=upper_diag_index-diag_index;    \
    const int content_offset= \
    ComputeContentOffset(diag_index, max_diag_len, num_rows, num_cols,   \
                         left_align_superdiagonal, left_align_subdiagonal); \
      const int x_offset=diag_index>0?diag_index: 0;    \
      const int index_in_the_diagonal=col-x_offset+content_offset;    \
      if (lower_diag_index<=diag_index&&diag_index<=upper_diag_index) {   \
          output_ptr[index] = \
          diag_ptr[batch*num_diags*max_diag_len+   \
          diag_index_in_input*max_diag_len+index_in_the_diagonal];   \
      } else {   \
        output_ptr[index] = padding_value;   \
      }   \
    }                                                \
  } \n";
```

图 2　CUDA 核函数与转换后 OpenCL 核函数对比(对角矩阵生成核函数)

发现,区别于一般情况下的 CUDA 核函数代码,TensorFlow 源码中的 CUDA 核函数内通常包含对 CUDA 内建变量或接口的封装.例如,TensorFlow 中用于设置英伟达 GPU 中计算资源的 *GPU_1D_KERNEL_LOOP*(*index*,*N*)在转换时需要被替换为 for(int *index*=*get_global_id*(0);*index*<*N*;*index*+=*get_global_size*(0)).

此外,还需要对CUDA代码中主机端的部分接口函数进行修改.例如,对于CUDA主机端代码中用于将核函数发送到GPU上的*cudaLaunchKernel*()函数,需要替换为OpenCL中的*clEnqueueNDRangeKernel*()函数.

(3)核函数中 device 函数代码重写

CUDA 核函数中经常将部分计算功能抽象为"*__device__*"修饰符限定的函数在 CUDA 核函数中调用.由于在 OpenCL 标准中没有对应的修饰符,本文在转换时会重新实现一个功能相同的子函数,在 OpenCL 核函数中调用.

3.3 复杂核函数转换中的典型问题

TensorFlow 中存在部分实现相对复杂的CUDA核函数,仅通过 3.2 节中介绍的基本转换流程无法实现对这部分核函数的转换.因此,本文整理了以下四个转换过程中的典型问题:

(1)CUDA 核函数中的模板问题;

(2)包含共享变量的 CUDA 核函数;

(3)不同内存类型间的数据传输;

(4)CUDA 原子操作的转换.

本文对所列典型问题进行了细致的研究,并给出了一系列可行的解决方法.

3.3.1 CUDA 核函数中的模板问题

综合考虑精度、运行效率和底层加速硬件支持情况等因素,TensorFlow 等机器学习框架中需要对不同的数据类型进行处理.例如,在不同的深度学习模型中可能需要处理 *half*、*int32*、*float32*、*double* 等多种数据类型的入参.为了减少冗余的代码,TensorFlow 中的 CUDA 核函数使用了 C++的模板功能[29].模板是实现代码重用机制的一种工具,它可以实现类型参数化,即将类型定义为参数,从而实现了代码可重用性.

对于仅能支持 C99 的 OpenCL 1.2 核函数代码,无法使用 C++中的模板.为了能够在 OpenCL 核函数中支持不同数据类型入参,可使用以下两种方式对 CUDA 核函数进行转换:

(1)为不同的数据类型分别实现对应的 OpenCL 核函数,即用多个OpenCL核函数代替单个CUDA核函数.

(2)获取参数类型,将数据类型以字符串形式作为参数传递进 OpenCL 核函数代码中.

对于代码规模庞大的 TensorFlow 来说,使用第一种转换方式会产生大量冗余代码且难以进行后续更新和维护.因此,本文采用第二种方式对 CUDA 代码中的模板进行改写,通过图 3 中所示方法,将变量的数据类型名称转换为字符串.获取到的类型名称字符串可用于后续 OpenCL 核函数中对变量进行声明.

```
1. static const string GetFullName(const char* name){
2. int status = -1;
3. char* fullName=abi::__cxa_demangle(name, NULL, NULL, &status);
4. const char* const demangledName=(status == 0)? fullName:name;
5. string ret_val(demangledName);
6. free(fullName);
7. return ret_val;
8. }
```

图 3 数据类型名称转换成字符串示例

其他基于 CUDA 开发的机器学习和数学库中,参数模板也同样有着广泛应用.本文给出的模板转换方法为解决 CUDA 到 OpenCL 核函数转换任务中 OpenCL 无法支持模板的问题提供了一种通用可行的解决思路.

3.3.2 包含共享变量的 CUDA 核函数

内存访问效率往往能够影响核函数的计算性能.片上内存(如 CUDA 共享内存)具有延迟低、带宽高的特点,其访问速度远大于板载内存(如全局内存)的访问速度.因此,当核函数需要频繁读写一段数据时,TensorFlow 源码中通常采用图 4 所示方法在 CUDA 核函数中动态创建共享内存(*shared memory*)中的变量并设置内存对齐.

```
1. GPU_DYNAMIC_SHARED_MEM_DECL(8, char, s_buf);
2. AccT* s_data= reinterpret_cast<AccT*>(s_buf);
3. for (int32 index=threadIdx.x; index <bias_size; index+=blockDim.x){
4.  s_data[index]=AccT(0);
5. }
6. __syncthreads();
```

图 4 TensorFlow 声明 CUDA 共享内存类型变量示例

OpenCL 中局部内存(*local memory*)与 CUDA 的共享内存在概念上对应.但与 CUDA 不同的是,在 OpenCL 核函数执行前,程序已将存放数据所需的资源分配好,OpenCL 标准不支持在核函数内动态分配局部变量.因此,简单使用关键字替换无法对包含共享变量的 CUDA 核函数进行转换.

为了保持与 CUDA 核函数一致的内存使用方式,本文通过以下流程实现了对 CUDA 共享变量的

替换，包括在主机端声明 OpenCL 局部变量、为局部变量设置内存对齐以及在 OpenCL 核函数内对局部变量的初始化：

(1) 在主机端调用 *clSetKernelArg*() 函数并在数据指针位置传入 NULL，完成对局部变量的声明；

(2) 将步骤(1)中声明的局部变量以函数入参形式传入 OpenCL 核函数，并使用 *__attribute__*((*aligned*(*sizeof*(*T*)))) 设置内存对齐；

(3) 在 OpenCL 核函数内，通过全局或私有变量完成对局部变量初始化.

相似 CUDA 共享变量的使用方式同样常见于其他机器学习框架. 本文给出的关于 CUDA 共享变量转换方法也可应用于其他机器学习框架的代码转换，具有较好通用性.

3.3.3　不同内存类型间的数据传输

为了优化运算性能，CUDA 核函数内经常需要在不同内存类型间(如全局-局部、局部-私有等)进行数据传输. TensorFlow 中 CUDA 核函数一般使用传递指针的方式传输数据. 然而，OpenCL 1.2 标准中并不支持不同内存类型间的指针传输. 因此，本文特别研究了 OpenCL 核函数中不同内存类型间数据的传输方法. 一般情况下，数据可以采用逐个数组元素赋值的方法传输. 特别地，在全局和局部内存之间还可以调用异步复制函数(*async_work_group_copy*())进行数据拷贝[30].

OpenCL 核函数中使用直接赋值方法传输数据时的难点在于无法直接获取传输数据的总量以及需要动态开辟内存空间. 为解决传输数据总量无法直接获取的难点，本文提前对 TensorFlow 中 CUDA 主机端代码进行分析. 计算出需要传递的数据总量并将该值作为 OpenCL 核函数的入参，用于后续赋值运算. 对于动态开辟内存空间的难点，本文将需要开辟的空间大小值转换成字符并添加到 OpenCL 核函数代码字符串中. 例如，对于全局到局部内存的数据传输，本文采用图 5 所示方法将全局变量中的数据赋值到局部变量中.

```
1. string StrN=to_string(length);
2. string kernel_str="__kernel foo(Args..., __global int* input,
3. const int N){
4. __local l_data ["+StrN+"];
5. for(int i; i<N; i++){
6.         __local l_data[i]=input[i];
7. }
8. }"
```

图 5　OpenCL 全局内存数据传递到局部内存示例

使用 OpenCL 提供的异步拷贝函数传输数据时，由于异步复制函数在两次执行复制操作间，不会执行隐式源数据同步操作，因此在核函数中调用异步复制后通常需要 *barrier*() 函数进行数据同步. 另外，由于异步拷贝会被工作组内所有工作项执行，因此不能使用条件判断语句绕过异步复制，即在条件判断语句中只能使用直接赋值的方式进行数据传输. 本文使用图 6 所示方法完成了异步赋值的数据传输.

```
1.  string StrN = to_string(length);
2.  string kernel_str = "__kernel foo(Args..., __global int* input,
3.              const int N, __global int* output){
4.    const int local_id=get_local_id(0)
5.  __local l_data ["+StrN+"];
6.  l_data [local_id]=0;
7.  barrier(CLK_LOCAL_MEM_FENCE);
8.  event_t evt;
9.  evt = async_work_group_copy(l_data, input, size, &evt);
10. barrier(CLK_LOCAL_MEM_FENCE);
11. ……
12. evt = async_work_group_copy(output, l_data, size, &evt);
13. barrier(CLK_LOCAL_MEM_FENCE);
14. }"
```

图 6　OpenCL 异步赋值的数据传输

3.3.4　CUDA 原子操作的转换

当多个线程同时访问同一内存地址时会存在竞态条件. 使用原子操作可以避免竞争，确保线程对某个内存单元完成读-修改-写的过程不被其他线程影响. TensorFlow 在 *biasadd*、*resize*、*dilation* 等多个算子中使用了原子操作保证计算结果的效率与正确性.

与 CUDA 不同的是，OpenCL 标准中的原子操作仅支持整型数据作为输入. 在将其他数据类型原子操作的 CUDA 核函数转换成 OpenCL 核函数时，需要对原子操作进行重载. 本文利用循环 CAS (Compare And Swap)算法实现原子操作，图 7 展示了 *float* 类型原子加的 OpenCL 实现示例. 对于原子乘、原子除等其他原子操作，只需要替换第 8 行中的运算符即可. 本文首次将 CAS 算法应用于 TensorFlow 中 CUDA 到 OpenCL 原子操作的转换，该方法对于其他机器学习框架的转换以及自动转换的规则设计具有一定借鉴意义.

```
1.  void atomic_add(volatile __global float *source, const float operand) {
2.    union {
3.      unsigned int intVal;
4.      float floatVal;
5.    } newVal, prevVal;
6.    do {
7.      prevVal.floatVal = *source;
8.      newVal.floatVal = prevVal.floatVal + operand;
9.    } while (atomic_cmpxchg((volatile __global unsigned int *)source,
10.                   prevVal.intVal, newVal.intVal)
11.                   != prevVal.intVal);
12.  }
```

图 7　OpenCL 中 *float* 类型原子加的实现示例

3.4 优化加速

通过实际的测试，我们发现转换后 OpenCL 核函数在性能上存在进一步优化和提升的空间. 因此，本文特别研究了 OpenCL 代码在设备初始化、编译执行等过程中的特性，设计实现了以下三种优化方法：

(1) 使用单例模式对 OpenCL 初始化；

(2) 缓存 OpenCL 核函数编译结果；

(3) 使用归约算法优化原子操作.

3.4.1 使用单例模式对 OpenCL 初始化

OpenCL 程序的主机端代码初始化时，需要通过 *clCreateContext*()、*clCreateCommandQueue*() 等接口函数执行查找 *platform*(平台) 和 *device*(设备)、创建 *context*(上下文) 和 *commandqueue*(命令队列) 等操作. 完成 OpenCL 核函数运算后，需要通过调用 *clRelease*() 将初始化时申请将设备、上下文、命令队列等资源进行释放.

图 8 展示了在机器学习模型中出现频率较高的 6 个 OpenCL 核函数在单次执行时，OpenCL 接口函数的时间占比情况. 从图中不难看出，OpenCL 初始化与释放相关的 *clCreateContext*()、*clCreateCommandQueue*()、*clRelease*() 等接口合计占用了约 90% 的代码执行时间. 在机器学习模型中单个操作经常需要一次执行多个核函数. 这意味着相关 OpenCL 接口函数会反复被调用，整个模型的执行效率将大幅降低.

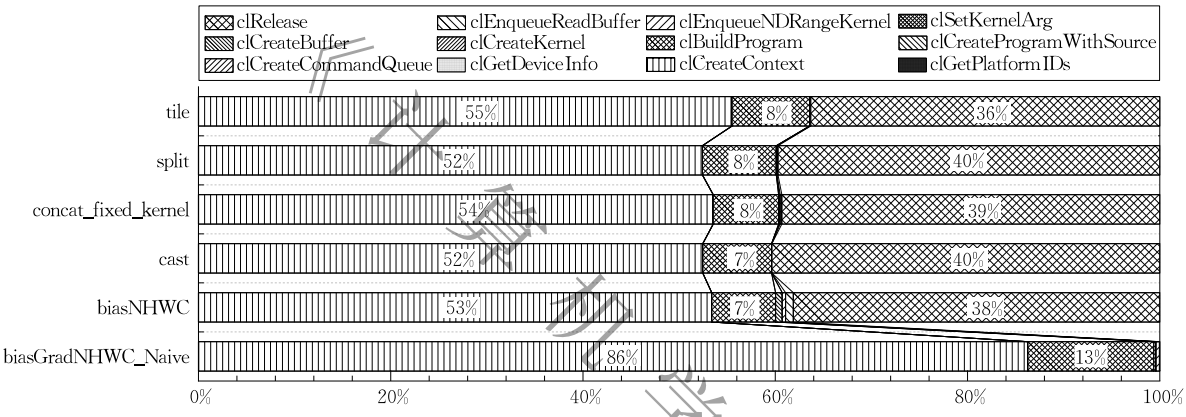


图 8 典型 OpenCL 核函数代码中 OpenCL 接口执行时间占比统计

实际上，根据 OpenCL 标准，查找平台、申请设备以及创建上下文等与初始化相关的接口函数在运行不同核函数时只需调用一次. 因此，本文采用 C++ 中的单例模式对这些接口函数进行封装. 图 9 展示了本文针对 OpenCL 中部分接口函数设计的类图.

执行 OpenCL 代码时，如果单例对象未被创建则新建一个对象并且对其中的成员变量进行初始化. 如果单例对象已经被创建，则直接返回已有对象. 对于初始化设备、创建上下文或创建命令队列等操作，可通过 *GetDevice*()、*GetContext*()，*GetQueue*() 这些接口来获取已完成初始化的 *device*、*context*、*commandqueue* 等变量.

本文首次将 C++ 中单例设计模式应用于 TensorFlow 的 OpenCL 后端实现. 使用单例模式一方面可以减少重复冗余的 OpenCL 代码，另一方面能够有效地避免重复执行初始化接口带来的额外开销，提升代码的运行效率. 这种设计模式对于其他基于 OpenCL 的机器学习框架或数学库的设计具有一定的启发意义.

3.4.2 缓存 OpenCL 核函数编译结果

执行 OpenCL 核函数的基本流程是：首先，调用 *clCreateProgramWithSource*() 函数将 OpenCL 核函数的字符串编译为 *cl_program* 对象. 其次，将 *cl_program* 对象作为参数传入 *clCreateKernel*() 函数中生成 *cl_kernel* 对象. 最后，将 *cl_kernel* 发送到设备上运行. 从以上流程不难看出，若一个 OpenCL 程序中多次调用某个核函数，程序会反复编译这个核函数，严重影响了 OpenCL 代码的执行效率.

基于 OpenCL 核函数源码编译生成的 *cl_program* 对象可以重复使用，单个 *cl_program* 对象可以用来生成多个 *cl_kernel* 对象. 在图 9 展示的 *Singleton* 单例类中，本文构造了 *unordered_map*〈*string*，*cl_program*〉类型的变量 *program_record*，用于将核函数源码编译生成的 *cl_program* 以核函数的字符串作为索引进行缓存. 程序首先检查在缓存中是否存在与待调用核函

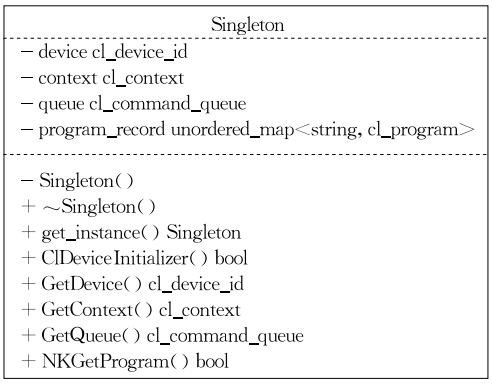


图 9　TensorFlow 单例类图

数字符串对应的 *cl_program*. 若存在则直接取出使用，若不存在则对核函数字符串进行编译，同时将生成的 *cl_program* 进行缓存.

将 OpenCL 核函数编译结果进行缓存的方式可以减少编译相同 OpenCL 核函数时的性能损耗，提升程序整体的运行效率. 其他基于 OpenCL 的机器学习框架或数学库，也可采用相同的思路进行设计或实现进一步优化.

3.4.3　使用归约算法优化原子操作

当核函数中需要多次调用原子操作修改相同内存地址时，由于进程间的竞争关系，一系列原子操作只能串行完成. 因此，使用原子操作的计算性能有时甚至不如单线程循环. 如图 10 所示，尽管在 OpenCL 中通过多个工作组(*work group*)并行完成原子操作能够提升一定的计算性能，但输入数据量较大时，进程阻塞导致的性能损耗仍不可忽视.

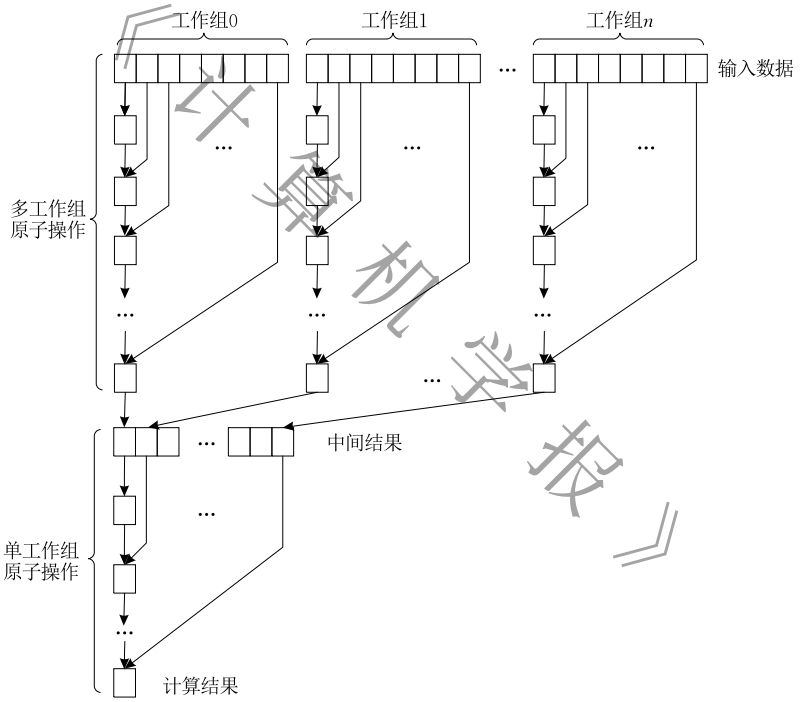


图 10　TensorFlow 中多工作组原子操作计算示意图

为了充分利用 OpenCL 多线程处理计算任务的优势，进一步提升转换后 OpenCL 核函数的性能. 本文特别研究了如何使用归约思想优化 OpenCL 原子操作的方法. 如图 11 所示，考虑到 OpenCL 仅支持工作组内线程同步，在计算归约结果时本文使用两步归约(*two-stage reduction*)算法：首先，完成工作组内部归约并保存每个组的归约结果. 其次，使用单个工作组对第一步中所有归约结果再次执行归约，得到最终结果.

归约算法能否正确运行的关键在于 OpenCL 核函数中全局工作线程(*global_work_size*)和局部工作线程(*local_work_size*)大小的设置以及保存中间结果所需空间的计算. 本文中将局部工作线程数设为 2 的 N 次方，并基于输入数据对全局工作线程数以及保存第一步中间结果所需空间的大小进行调整.

本文首次将基于 OpenCL 的两步归约算法应用于对 TensorFlow 原子操作的优化中，为提升类似机器学习框架中原子操作的计算性能提供了一种可行的优化思路.

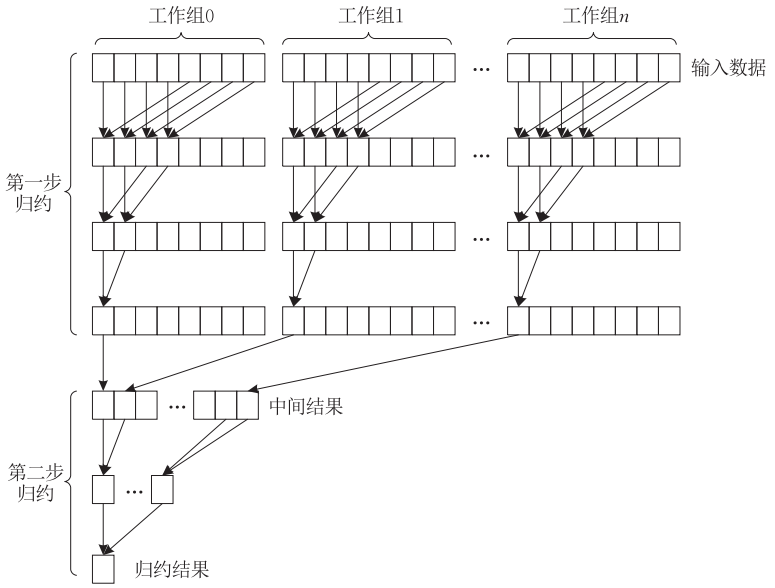


图 11 两步归约计算示意图

4 OpenCL 核函数在 TensorFlow 中集成和调用方法

为了能够使用 TensorFlow 的 Python 前端对本文转换的 OpenCL 核函数进行调用和测试，本文对 OpenCL 核函数在 TensorFlow 中集成和调用方法进行了探索和研究.

4.1 TensorFlow 中 OpenCL 核函数集成

以在 TensorFlow 中创建一个名称为 *FooBar* 的算子为例，OpenCL 核函数在 TensorFlow 中的集成可通过以下步骤实现[①]：

(1)算子接口的定义和声明

TensorFlow 中一般调用 *REGISTER_OP*()宏对算子的接口进行定义和声明. 该宏可以定义算子的名称、输入输出形状等基本信息. 图 12 展示了算子定义和声明代码的示例.

```
1. REGISTER_OP("FooBar")
2.     .Input("input:int32")
3.     .Output("output:int32")
4.     .SetShapeFn([](::tensorflow::shape_inference::InferenceContext* c){
5.       c->set_output(0, c->input(0));
6.       return Status::OK();});
```

图 12 TensorFlow 中算子接口声明代码示例

本例中名称为 *FooBar* 的算子将一个数据类型为 *int*32 的张量 *input* 作为输入，并输出一个数据类型为 *int*32 的张量 *output*. 调用了形状函数 *SetShapeFn*() 来确保输出张量与输入张量形状相同.

TensorFlow 对于算子的命名规则有所限制，即必须采用驼峰命名法且对于注册在相同设备上的所有算子，算子名是唯一的. 在 Python 端生成的接口函数会根据此驼峰命名生成对应的下划线命名，因此 *FooBar* 在 Python 中的名称为 *foo_bar*.

(2)实现算子的计算核心类

完成算子接口定义后，还需要为算子提供一种或多种具体实现(即本例中的 *ClFooBarOp*). 图 13 展示了计算核心类的实现方式，TensorFlow 中算子的计算核心类一般继承于通用的 *OpKernel* 类. *Compute*() 方法是 *OpKernel* 类的核心，主要用于输入输出数据的处理以及执行具体计算. 本文将具体的 OpenCL 核函数调用过程封装在自定义的 *ClKernelLauncher*() 函数中.

```
1.  class ClFooBarOp : public OpKernel {
2.   public:
3.    explicit ClFooBarOp(
4.    OpKernelConstruction* context) : OpKernel(context) {}
5.    void Compute(OpKernelContext* context) override {
6.        // Deal with input & output
7.        // And allocate memory for output
8.        ...
9.        // Execute OpenCL Host Code
10.       ClKernelLauncher(...);
11.   }
12. };
```

图 13 TensorFlow 计算核心类的实现示例

(3)将算子注册到 TensorFlow 系统

实现 *FooBar* 的计算内核后，还需要使用 *REGISTER_KERNEL_BUILDER*() 宏将其注册到 TensorFlow 中，注册方式如图 14 所示.

```
1. REGISTER_KERNEL_BUILDER(Name("FooBar")
                .Device(DEVICE_CPU), ClFooBarOp);
```

图 14 TensorFlow 内核注册方式示例

① Create an op. https://www.tensorflow.org/guide/create_op, 2021,9,20

4.2　TensorFlow 中 OpenCL 核函数的调用

为了能够通过 TensorFlow 的 Python 前端对 OpenCL 算子进行调用.本文使用了 TensorFlow 官方介绍的方法,使用动态链接库对 OpenCL 算子进行了封装.在编译动态链接库时需要确保系统中已安装满足 TensorFlow 编译条件的 g++版本.使用图 15 展示的编译命令可将 *foo_bar* 算子编译为在 Python 中可调用的动态链接库文件.

```
1. TF_CFLAGS=($(python -c'import tensorflow as tf;
                print(" ".join(tf.sysconfig.get_compile_flags()))'))
2. TF_LFLAGS=($(python -c'import tensorflow as tf;
                print(" ".join(tf.sysconfig.get_link_flags()))'))
3.  g++ -std=c++11 -shared foo_bar.cc -o foo_bar.so -fPIC
                ${TF_CFLAGS[@]} ${TF_LFLAGS[@]} -O2
```

图 15　g++动态链接库编译命令示例

在 Python 中使用 *foo_bar* 算子时,需要调用 *tensorflow.load_op_library*()函数对包含 *foo_bar* 算子的动态链接库进行加载,如图 16 所示.加载完成后算子的使用方式与 TensorFlow 中 CUDA 算子完全一致,传入正确的输入数据即可完成计算.

```
1. foo_bar_module = tensorflow.load_op_library('./foo_bar.so')
2. foo_bar = foo_bar_module.foo_bar
```

图 16　TensorFlow 的 Python 前端中动态链接库加载方法

5　实验分析

本文设计并完成了一系列的实验回答了如下问题:

RQ1 转换后 OpenCL 核函数的正确性如何?

RQ2 转换后 OpenCL 核函数的性能如何?

RQ3 优化后核函数的性能提升效果如何?

RQ4 转换后 OpenCL 核函数的通用性如何?

为了验证本文总结的 CUDA 到 OpenCL 核函数转换方法的正确性及性能,本文使用了表 5 所列的软硬件环境进行了实验.

表 5　实验软硬件环境

软硬件名称	参数规格
CPU	Intel(R) Xeon(R) Gold 5218 CPU@2.30 GHz
RAM	187 GB DDR4 2933 MT/s
GPU	NVIDIA Tesla V100S
NVIDIA CUDA Toolkit	CUDA-10.2
OpenCL	CUDA 10.2 OpenCL 1.2
Host compiler	GCC7.5
TensorFlow	TensorFlow 2.2

考虑到英伟达 GPU 可以同时支持 CUDA 和 OpenCL 核函数的运行,本文中的实验采用 CPU+英伟达 GPU 的异构计算环境,选择 CUDA-10.2 中自带的 OpenCL 库,保证了在相同环境下执行 CUDA 核函数和 OpenCL 核函数.由于本文的转换目标为 TensorFlow 2.2 版本中的 CUDA 核函数,因此本文中的实验以 TensorFlow 2.2 中使用 GPU 或 CPU 的计算结果为基准.

5.1　测试工具

本文通过第 4 节介绍的方式将转换后 OpenCL 核函数集成到 TensorFlow 中,使用 TensorFlow 的 Python 前端分别调用 OpenCL 和 CUDA 算子.借助了 Python 中自带的单元测试框架 unittest 设计及实现了标准化测试.unittest 单元测试提供了创建测试用例、测试套件以及批量执行的方案①.unittest 在安装 Python 后就可以通过 *import unittest* 引入后直接使用.为了能够生成直观且可交互的 HTML 测试报告,本文引入了 unittest 模块的扩展文件.基于 unittest 和 HTMLTestRunner.py 即可以实现核函数的测试需求②.图 17 展示了核函数的测试流程

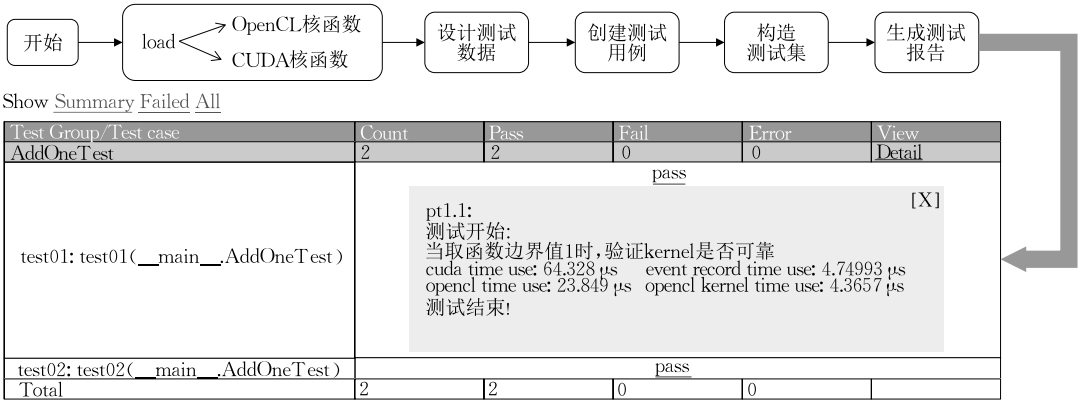


图 17　核函数测试流程及测试报告样例

① unittest introduction. https://docs.python.org/zh-cn/3.7/library/unit-test.html, 2021,9,12

② HTMLTestRunner introduction. http://tungwaiyip.info/software/HTMLTestRunner.html, 2021,8,30

以及 *AddOne*()核函数的测试结果报告. 其中,*load* 即为 *tensorflow.load_op_library*('*xxx.so*')(详见第 4 节). 本文实验的主要测试内容包括:不同数据类型数据(*int*16(*short*),*int*32,*int*64(*long*),*float*32,*float*64(*double*))、不同数据量级(2^{14},2^{15},…,2^{20})等. 最终,借助生成的测试报告,可直观地对核函数测试结果以及性能进行分析.

5.2 实验对象及评价标准

实验首先对比了 135 个转换后 OpenCL 核函数与 TensorFlow 2.2 源码中 CUDA 核函数在不同输入数据类型下的计算结果,验证了转换后 OpenCL 核函数的正确性及可靠性. 其次,为了展示使用本文转换方法生成的 OpenCL 核函数的整体性能,实验统计了转换后 OpenCL 核函数的运行时间并与对应 CUDA 核函数的运行时间进行了对比. 同时,为了研究输入数据量的变化对 OpenCL 核函数运行效率的影响,本文对表 6 中列举的 7 个具有代表性的核函数进行了测试. 此外,本文使用表 7 中列举的 2 个具有代表性的核函数说明了本文 3.4 节中介绍的优化方法对 OpenCL 核函数计算性能的提升效果. 最后,为了验证完成转换的 OpenCL 核函数在不同计算设备上的通用性,本文在华为鲲鹏-920CPU 以及国产 MT-3000 异构加速器环境下对转换后 135 个 OpenCL 核函数的正确性进行测试. 由于在非英伟达 GPU 环境中无法使用 CUDA 核函数,通用性实验以 TensorFlow 2.2 版本中 CPU 的计算结果为基准.

表 6 代表性核函数及功能介绍

核函数	功能
BiasNHWCKernel	将输入张量的最后一维与偏置张量(一维张量)进行向量加法
Cast	将源类型的输入张量转换成目标类型的输出张量
Concat_xh	按照特定方式将两个张量连接
COOMatrixToSparseTensorKernel2D	将两个二维张量合并,生成一个交替排列的稀疏矩阵
DiagGpuKernel	将给定的矩阵对角化
MaxPoolGradBackwardNoMaskNCHW	计算无掩码 NCHW 格式数据最大池化的梯度
Relu_int8x4_kernel	以字节为单位,计算每个元素的 ReLU 函数的值并输出

表 7 优化测试核函数、核函数功能以及优化方法

核函数	功能	优化方法
MaxPoolForwardNHWC	NHWC 格式输入张量的最大池化	单例模式 单例模式+缓存机制
BiasGradNCHW_SharedAtomics	NCHW 格式的输入张量中相同通道的数据累加	归约算法替换原子操作

OpenCL 核函数计算结果的精度与编译生成 *cl_program* 对象时是否添加 *-cl-fp32-correctly-rounded-divide-sqrt* 等编译选项相关. 考虑到 OpenCL 标准的通用性要求,并非所有计算设备都能够支持这些编译选项. 因此,默认情况下编译 *cl_program* 对象时不会添加这些编译选项. 这是导致相同功能的 CUDA 和 OpenCL 核函数在计算结果上可能会存在精度差异的主要原因. 在验证核函数转换的正确性时,本文设计了如式(1)所示的正确率评价指标,即为在特定精度下核函数测试集的正确率.

$$Acc@(error)=\frac{\sum_{k\in R}k}{|R|} \tag{1}$$

其中 *error* 代表误差精度(本文实验中选取了 1e-2 到 1e-5),k 代表在设定误差精度下测试通过的核函数,$|R|$ 代表测试集中核函数的总个数.

为了能够直观地评估本文转换的 OpenCL 核函数与 TensorFlow 原有 CUDA 核函数在性能上的差异,本文设计了式(2)中所示的性能评价指标.

$$Per_{\text{score}}=\lg\frac{T_{\text{OpenCL}}}{T_{\text{CUDA}}} \tag{2}$$

其中,T_{OpenCL} 为 OpenCL 核函数的运行时间,T_{CUDA} 为对应 CUDA 核函数的运行时间. 在相同条件下 Per_{score} 越接近零,则代表 OpenCL 核函数与 CUDA 核函数性能越接近,为负代表 OpenCL 核函数性能优于 CUDA 核函数,为正代表 CUDA 核函数性能优于 OpenCL 核函数.

5.3 RQ1:转换后OpenCL核函数运算结果的正确性

表 8 展示了转换后 135 个 OpenCL 核函数在不同输入数据类型下的 $Acc@(error)$ 值. 对于全部整

表 8 OpenCL 核函数正确性测试结果

允许误差	$Acc@(error)$/%				
	*int*16(*short*)	*int*32	*int*64(*long*)	*float*32	*float*64(*double*)
1e-02	100	100	100	100.00	100
1e-03	100	100	100	94.07	100
1e-04	100	100	100	91.11	100
1e-05	100	100	100	88.89	100

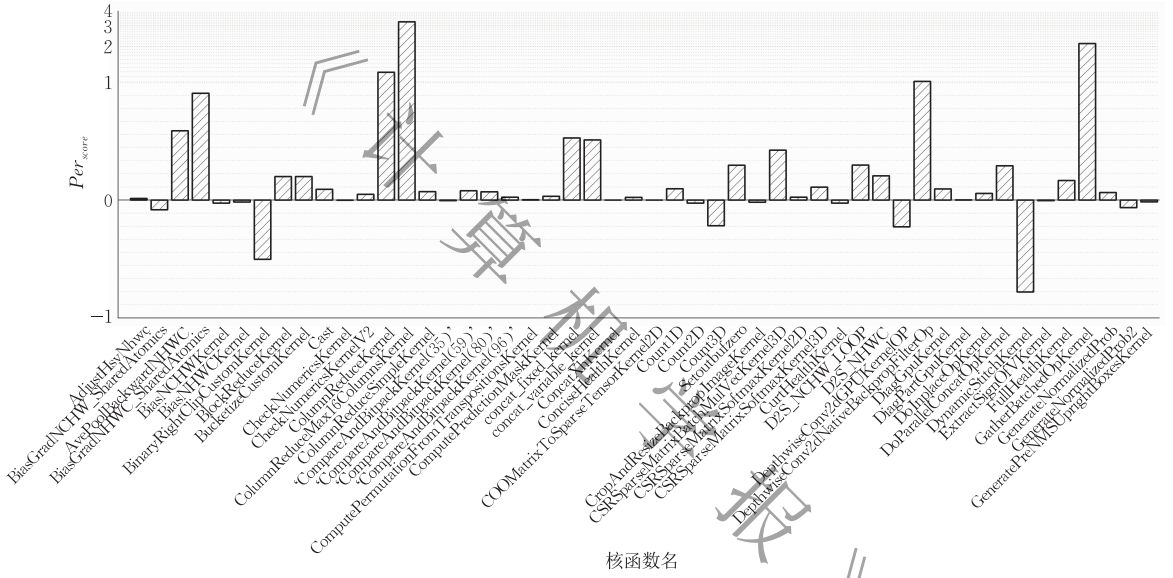
型数据和双精度浮点型数据，OpenCL 核函数的全部 $Acc@(error)$ 值都是 100%. 对于单精度浮点型数据，当误差为(1e-5)时，转换后 OpenCL 核函数的 $Acc@(error)$ 为 88.89%，在误差在(1e-2)以内 $Acc@(error)$ 可达到 100%.

通过对具体 OpenCL 核函数的分析发现，计算单精度浮点型数据时产生误差的核函数中一般包含除法运算或 *rsqrt*()、*tanh*()等数学函数. 这些误差主要是由 5.2 节中提到的 OpenCL 核函数在编译 *cl_program* 时编译选项的设置引入，并不是本文总结的转换方法导致的错误.

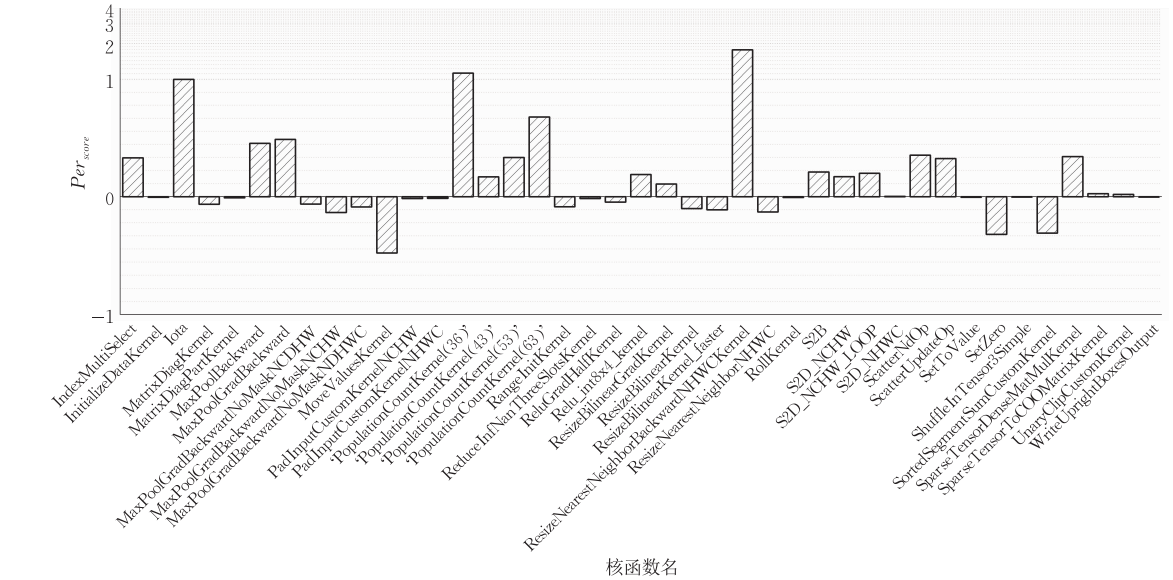
5.4　RQ2：转换后 OpenCL 核函数的性能

为了能够准确地对测试集中核函数的性能进行评价，本文通过实验分别统计了 TensorFlow 中相同数据规模下 OpenCL 的核函数与 CUDA 核函数的运行时间. 对于 OpenCL 核函数，本文的实验记录了 1000 次核函数计算的平均用时作为单个 OpenCL 核函数的用时；对于 CUDA 核函数，本文实验使用了 CUDA 自带统计工具 *nvprof* 对 CUDA 核函数的运行时间进行了统计. 通过实验发现，*nvprof* 仅能够获取到部分 CUDA 核函数时间信息. 因此，我们在实验中对比了 91 个能够通过 *nvprof* 统计工具统计出时间信息的 CUDA 核函数和对应的 OpenCL 核函数.

由图 18 看出，转换后约 80%核函数的 Per_{score} 值分布在(−1,1)区间内. 这说明转换后 OpenCL 核函数大部分与 CUDA 核函数在运算时间上接近，部分



(a) OpenCL核函数性能



(b) 对应CUDA核函数性能

图 18　OpenCL 核函数与对应 CUDA 核函数性能对比图

OpenCL 核函数在性能上优于 CUDA(Per_{score} 为负值). 然而,实验结果表明,有少部分 OpenCL 核函数的计算性能较差,运行时间远超 CUDA 核函数. 经过对这部分核函数分析可以发现,性能较差的 OpenCL 核函数往往包含原子操作、条件判断等需要串行处理的操作,严重影响了核函数运算性能. CUDA 对于这些耗时较多的操作一般进行了针对性的优化.

为了验证转换后 OpenCL 核函数对不同规模输入数据量的鲁棒性,本文通过实验对 7 个典型的核函数在不同数据量级上的运算时间进行了统计,从图 19 中可以看出,转换后 OpenCL 核函数与 TensorFlow 源码中 CUDA 核函数的运算时间随着数据规模的增加运算时间增长相对平缓. 而使用 CPU 运算相同功能时,随着数据规模的增加运算时间显著增长. 由此可见,本文转换生成的 OpenCL 核函数随着输入数据规模的增长鲁棒性较好,运算效率没有明显波动.

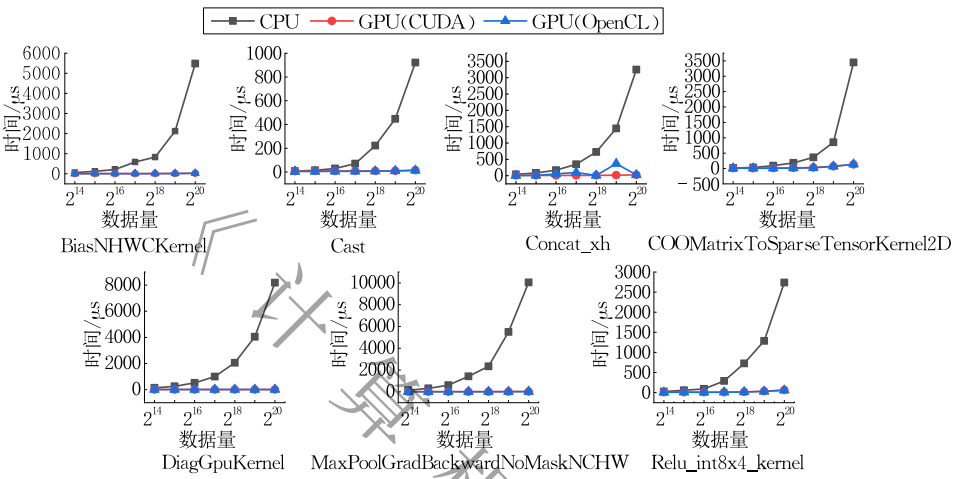


图 19 7 种典型核函数在不同输入数据规模下使用 CPU、GPU(OpenCL)、GPU(CUDA)运算时间的对比

5.5 RQ3:优化后 OpenCL 核函数性能

本文3.4.1节中介绍了使用单例模式优化 OpenCL 的初始化流程,提升 TensorFlow 中 OpenCL 代码整体运行效率的方法. 为了展示该方法对 OpenCL 代码计算性能的优化效果,实验对 *MaxPoolForwardNHWC* 核函数进行了测试. 以深度学习中经常使用的 *float*32 类型数据作为输入,测试了不同输入规模下核函数执行 1000 次的平均时间. 如图 20(a)所示,经过优化后 OpenCL 核函数的运行时间相比于优化前运算速度有明显提升. 这是由于程序首次调用初始化设备、创建上下文等 OpenCL 接口函数后,经过单例模式优化后的 OpenCL 核函数在后续执行时无需重复调用这部分接口函数.

由于 OpenCL 采用运行时编译的机制,每次调用 OpenCL 核函数时都需要对其进行编译. 重复编译相同的 OpenCL 核函数会带来较大的时间损失. 为此,本文使用了 3.4.2 节中介绍的缓存机制,将编译过的 OpenCL 核函数进行缓存. 实验对 *MaxPoolForwardNHWC* 核函数进行了测试,对比使用单例模式+缓存机制的优化方式与仅使用单例模式优化

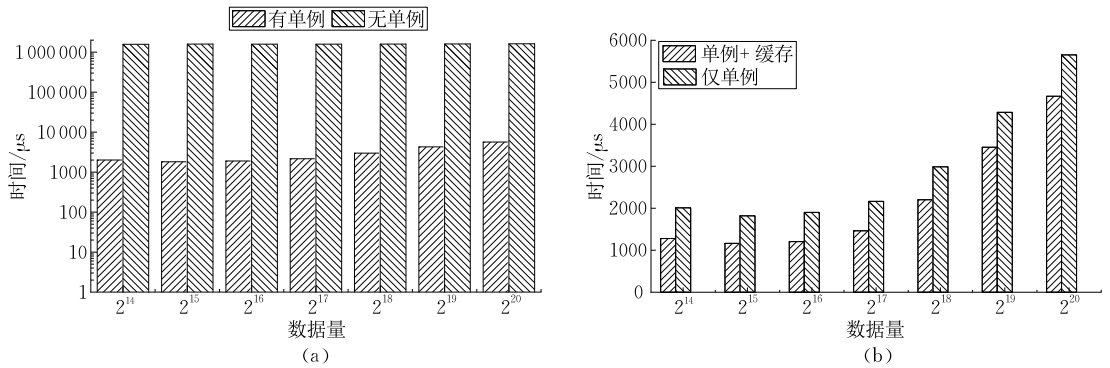


图 20 不同输入数据规模下使用不同优化策略核函数性能对比((a)使用单例进行优化与未优化时核函数在不同输入数据规模下的性能对比;(b)使用单例+缓存的组合优化模式与仅使用单例优化时核函数在不同输入数据规模下性能对比)

在计算性能上的差异.从图 20(b)中可以看出,相比于仅使用单例模式优化,将两种优化方法进行结合可以使整个 OpenCL 程序的性能在仅使用单例模式优化的基础上进一步提升 15%～30%.

本文以 *BiasGradNCHW_SharedAtomics* 为例,测试了使用归约算法替换原子操作在不同输入数据规模下的优化效果.实验中,输入数据量从 2^{14} 到 2^{20} 递增,分别统计了包含原子操作的 OpenCL 核函数、包含原子操作的 CUDA 核函数以及使用归约算法的 OpenCL 核函数执行 1000 次的平均时间.图 21(a)展示了以 *float*32 类型的数据作为输入的实验结果;图 21(b)展示了使用了 *double* 类型数据作为输入的实验结果.从中可以看出,相比于使用原子操作,基于归约算法的 OpenCL 核函数在不同的输入数据类型下的都能显著提升计算速度.同时,随着输入数据规模的不断增加,性能提升效果更加明显.另外,从实验中还能看出,由于 CUDA 更加擅长处理 *float*32 类型数据.因此,对于 *float*32 类型的输入,经本文优化后 OpenCL 核函数的运算速度与使用原子加的 CUDA 核函数性能接近,而对于 *double* 类型的数据,经本文优化后的 OpenCL 核函数性能优于 CUDA 核函数.

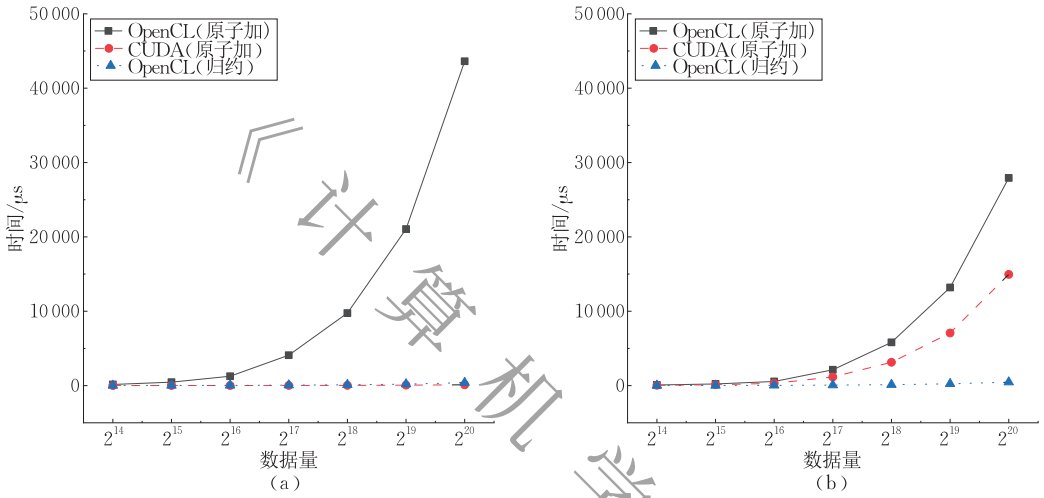


图 21　归约算法替换原子加操作的优化效果((a)输入数据类型为 *float*32 时算子使用 OpenCL 原子加、CUDA 原子加以及 OpenCL 归约算法在不同数据规模下的性能对比;(b)输入数据类型为 *double* 时算子使用 OpenCL 原子加、CUDA 原子加以及 OpenCL 归约算法在不同数据规模下的性能对比)

5.6　RQ4:转换后 OpenCL 核函数的通用性

为了进一步验证转换后 OpenCL 核函数在不同计算设备上的通用性,本文将完成转换的 135 个核函数在非英伟达 GPU 的两个环境中进行了部署与测试.在华为鲲鹏-920 CPU 环境使用了开源的 PoCL-1.6[31] 作为 OpenCL 1.2 的实现;在 MT-3000 异构加速器环境使用了国防科技大学设计的 MOCL3 作为 OpenCL 1.2 标准的实现.

由于上述两个环境均不支持 CUDA,本文实验以 TensorFlow 2.2 版本使用两个环境中 CPU 的计算结果为基准.根据 5.2 节中对 OpenCL 核函数正确性测试的经验,对于 *short*、*int*32、*long* 以及 *double* 类型的数据,实验设定的允许误差为 1e-5;对于 *float*32 类型的数据,实验设定的允许误差为 1e-2.表 9 中展示了 135 个 OpenCL 核函数在不同硬件环境下使用不同数据类型测试的结果.

表 9　不同硬件环境下 OpenCL 核函数测试结果

数据类型	允许误差	华为鲲鹏-920 CPU/%	MT-3000 异构加速器/%
*int*16(*short*)	1e-5	100	100
*int*32	1e-5	100	100
*int*64(*long*)	1e-5	100	100
*float*32	1e-2	100	100
double	1e-5	100	100

通过实验结果可以看出,本文转换后的 OpenCL 核函数可运行在不同类型计算设备中且计算结果正确无误.这表明本文转换后的 OpenCL 核函数具有良好的通用性.

6　总　结

机器学习框架对人工智能领域的发展起着举足轻重的作用.目前,以 TensorFlow 为代表的主流机器学习框架一般使用 CUDA、ROCm 等国外芯片厂

商根据自有硬件特点设计的异构编程框架开发.随着异构编程技术的不断发展以及硬件加速种类的不断丰富,如何使主流机器学习框架支持通用跨平台的异构编程标准受到了广泛的关注.作为一种通用开放的异构计算标准,OpenCL 支持在不同架构的计算设备上使用统一的接口编写异构程序.受此启发,本文对在 TensorFlow 中新增 OpenCL 后端进行了探索与研究,提出了一套将 TensorFlow 中的 CUDA 核函数转换为 OpenCL 核函数的方法,对转换过程中遇到的难点与挑战进行了分析并给出了可行的解决方法.此外,本文基于对 TensorFlow 中核函数特点的研究提出了一系列的优化策略.大量的实验验证了本文总结的转换方法的正确性.通过对比优化前后的 OpenCL 核函数运算性能可以看出,提出的优化方法能够显著提升核函数的运行效率.与此同时,不同硬件平台上的实验表明本文转换后 OpenCL 核函数具有很好的通用性.

目前,TensorFlow 的版本仍在快速迭代,如何增强核函数转换工作的可维护性,在 TensorFlow 版本更新后快速完成对应 OpenCL 核函数转换是本文后续研究的重点和方向.本文关于 CUDA 向 OpenCL 核函数转换以及机器学习框架中核函数优化的研究为后续相关工作提供了经验,可以为进一步的探索自动或半自动核函数转换技术提供有力的支撑.

参 考 文 献

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

[2] Goodfellow I, Bengio Y, Courville A. Deep Learning. Cambridge, USA: MIT Press, 2016

[3] Abadi M, Agarwal A, Barham P, et al. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467, 2016

[4] Abadi M, Barham P, Chen J, et al. TensorFlow: A system for large-scale machine learning//Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). Savannah, USA, 2016: 265-283

[5] Mittal S, Vetter J S. A survey of CPU-GPU heterogeneous computing techniques. ACM Computing Surveys (CSUR), 2015, 47(4): 1-35

[6] Paszke A, Gross S, Massa F, et al. PyTorch: An imperative style, high-performance deep learning library. Advances in Neural Information Processing Systems, 2019, 32: 8026-8037

[7] Sanders J, Kandrot E. CUDA by Example: An Introduction to General-Purpose GPU Programming. Boston, USA: Addison-Wesley Professional, 2010

[8] Reinders J, Ashbaugh B, Brodman J, et al. Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems Using C++ and SYCL. Cham, Switzerland: Springer Nature, 2021

[9] Alpay A, Heuveline V. SYCL beyond OpenCL: The architecture, current state and future direction of hipSYCL//Proceedings of the International Workshop on OpenCL. Munich, Germany, 2020: 1

[10] Doumoulakis A, Keryell R, O'Brien K. SYCL C++ and OpenCL interoperability experimentation with triSYCL//Proceedings of the 5th International Workshop on OpenCL. Toronto, Canada, 2017: 1-8

[11] Ramesh A, Pavlov M, Goh G, et al. Zero-shot text-to-image generation//Proceedings of the International Conference on Machine Learning (PMLR). Virtual Event, 2021: 8821-8831

[12] Radford A, Kim J W, Hallacy C, et al. Learning transferable visual models from natural language supervision//Proceedings of the International Conference on Machine Learning (PMLR). Virtual Event, 2021: 8748-8763

[13] Brown T, Mann B, Ryder N, et al. Language models are few-shot learners. Advances in Neural Information Processing Systems, 2020, 33: 1877-1901

[14] Munshi A. The OpenCL specification//Proceedings of the 2009 IEEE Hot Chips 21 Symposium (HCS). Stanford, USA, 2009: 1-314

[15] Jia Y, Shelhamer E, Donahue J, et al. Caffe: Convolutional architecture for fast feature embedding//Proceedings of the 22nd ACM International Conference on Multimedia. Orlando, USA, 2014: 675-678

[16] Bergstra J, Breuleux O, Bastien F, et al. Theano: A CPU and GPU math expression compiler//Proceedings of the Python for Scientific Computing Conference (SciPy). Austin, USA, 2010, 4(3): 1-7

[17] Chen T, Li M, Li Y, et al. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. arXiv preprint arXiv:1512.01274, 2015

[18] Seide F, Agarwal A. CNTK: Microsoft's open-source deep-learning toolkit//Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. San Francisco, USA, 2016: 2135

[19] Ma Yan-Jun, Yu Dian-Hai, Wu Tian, et al. PaddlePaddle: An open-source deep learning platform from industrial practice. Frontiers of Data and Computing, 2019, 1(1): 105-115(in Chinese)
(马艳军, 于佃海, 吴甜等. 飞桨: 源于产业实践的开源深度学习平台. 数据与计算发展前沿, 2019, 1(1): 105-115)

[20] Martinez G, Gardner M, Feng W. CU2CL: A CUDA-to-OpenCL translator for multi-and many-core architectures//

Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems. Tainan, China, 2011: 300-307

[21] Harvey M J, De Fabritiis G. Swan: A tool for porting CUDA programs to OpenCL. Computer Physics Communications, 2011, 182(4): 1093-1099

[22] Nandakumar D. Automatic Translation of CUDA to OpenCL and Comparison of Performance Optimizations on GPUS[M. S. dissertation]. University of Illinois at Urbana-Champaign, Urbana, Illinois, 2011

[23] Perkins H. CUDA-on-CL: A compiler and runtime for running NVIDIA CUDA™ C++11 applications on OpenCL™ 1.2 devices//Proceedings of the 5th International Workshop on OpenCL. Toronto, Canada, 2017: 1-4

[24] Gu J, Liu Y, Gao Y, et al. OpenCL Caffe: Accelerating and enabling a cross platform machine learning framework//Proceedings of the 4th International Workshop on OpenCL. Vienna, Austria, 2016: 1-5

[25] Goli M, Iwanski L, Richards A. Accelerated machine learning using TensorFlow and SYCL on OpenCL devices//Proceedings of the 5th International Workshop on OpenCL. Toronto, Canada, 2017: 1-4

[26] Burns R, Lawson J, McBain D, et al. Accelerated neural networks on OpenCL devices using SYCL-DNN//Proceedings of the International Workshop on OpenCL. Boston, USA, 2019: 1-4

[27] Keryell R, Reyes R, Howes L. Khronos SYCL for OpenCL: A tutorial//Proceedings of the 3rd International Workshop on OpenCL Palo Alto, USA, 2015: 1-1

[28] Kaeli D R, Mistry P, Schaa D, et al. Heterogeneous Computing with OpenCL 2.0. San Mateo, USA: Morgan Kaufmann, 2015

[29] Prata S. C++ Primer Plus. Boston, USA: Addison-Wesley Professional, 2011

[30] Koshiba A, Sakamoto R, Namiki M. OpenCL runtime for OS-driven task pipelining on heterogeneous accelerators//Proceedings of the 2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA). Hakodate, Japan, 2018: 236-237

[31] Jääskeläinen P, de La Lama C S, Schnetter E, et al. pocl: A performance-portable OpenCL implementation. International Journal of Parallel Programming, 2015, 43(5): 752-785

CHEN Rui, Ph. D. candidate. His research interests include deep learning heterogeneous computing and high performance computing.

SUN Yu-Fei, Ph. D., professor. Her research interests include deep learning and heterogeneous computing.

CHENG Da-Guo, M. S. candidate. His research interest is intelligent computing.

GUO Qiang, M. S. candidate. His research interests include deep learning and high performance computing.

CHEN Yu-Qiao, M. S. candidate. His research interest is deep learning framework.

SHI Chang-Qing, M. S. candidate. His research interests include deep learning and high performance computing.

SUI Yi-Cheng, Ph. D. candidate. His research interests include heterogeneous computing and deep learning algorithms.

ZHANG Yu-Zhe, M. S. candidate. His research interests include performance optimization and parallel computing.

ZHANG Yu-Zhi, Ph. D., professor. His research interests is artificial intelligence.

Background

The research in this paper belongs to the problem of generic machine learning frameworks and CUDA to OpenCL code conversion in the field of artificial intelligence. It is well known that deep learning models are generally built, trained, and reasoned about through machine learning frameworks. Currently, machine learning frameworks use CPU+accelerator heterogeneous computing to accelerate computation. However, mainstream machine learning frameworks can only support CUDA or ROCm heterogeneous programming frameworks, which are compatible with computing devices from a few foreign vendors such as NVIDIA, AMD, and Intel. Other vendors, especially domestic hardware vendors, design and produce acceleration devices that only support the OpenCL common programming specification. The inability to support mainstream machine learning frameworks makes it difficult to realize the great potential of these hardware accelerators in the field of deep learning. Considering the generic, cross-platform nature of OpenCL, by converting machine learning frameworks to support OpenCL, domestic acceleration devices can support machine learning frameworks. For TensorFlow

version 2.0 and above code, there is currently no available automatic conversion method to convert the CUDA backend in it to an OpenCL backend. This paper investigates a more comprehensive conversion of TensorFlow version 2.2 using a manual conversion approach by referring to the existing CUDA implementation in the TensorFlow source code.

This paper is the first work to manually convert 135 CUDA kernels under TensorFlow. This research summarises the conversion of CUDA kernels to OpenCL kernels under the TensorFlow framework, and sorts out a series of typical problems that may be encountered during code conversion in machine learning frameworks and Solutions. This project also summarizes the methods for calling, optimizing, and testing OpenCL kernels in TensorFlow. A series of experiments have shown that the conversion method in this paper is correct and reliable, and the computational performance of the OpenCL kernels generated by the conversion has been significantly improved after optimization. This project provides a reference for the conversion of other deep learning frameworks to OpenCL.

The research in this paper is supported by the National Key R&D Program of China (No. 2021YFB0300104). It realizes the performance optimization of intelligent computing and the deployment of intelligent applications on an intelligent computing platform in a domestic supercomputing system. The research in this paper plays a key role in the subject to support the deployment of intelligent applications.