软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(8):3896−3916 [doi: 10.13328/j.cnki.jos.007244] [CSTR: 32375.14.jos.007244]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

MTTorch: 面向 MT-3000 芯片和 Transformer 模型的 PyTorch 算子库实现与优化*

王昊天[1], 孙羽菲[1], 隋轶丞[1], 王嘉豪[1], 石昌青[1], 方建滨[2], 张玉志[1]

[1](南开大学 软件学院, 天津 300450)

[2](国防科技大学 计算机学院, 湖南 长沙 410073)

通信作者: 孙羽菲, E-mail: yufei_sun@sina.com

摘　要: 随着 Transformer 类大模型的飞速发展, 算力逐渐成为制约领域发展的瓶颈, 如何根据加速器硬件的结构特性加速和优化大语言模型的训练性能已成为研究热点. 面向天河新一代超算系统的加速芯片 MT-3000, 提出并实现了适用于 CPU+DSP 异构架构的 PyTorch 扩展库——MTTorch, 其核心是一个多核并行的算子库, 对 Transformer 类模型训练过程中的核心算子进行向量化实现和优化. 同时, 针对 MT-3000 架构特性, 提出了面向多核 DSP 的高性能规约算法及乒乓算法, 显著提升了算子的运算性能. MTTorch 还具有很好的通用性, 对于不同版本的 PyTorch 都可以动态链接库的形式进行加载, 不改变 PyTorch 的原生实现. 大量实验证明, 实现的核心算子在 MT-3000 芯片上有着很好的性能, 在单 DSP 簇上可以达到 8 倍的加速效果. 利用 MTTorch 在多节点执行训练任务时有着接近线性的加速比, 极大地提升了 Transformer 类模型在 MT-3000 芯片上的训练效率.

关键词: PyTorch; 高性能计算; Transformer 模型; 天河超级计算机; CPU+DSP 异构计算; 软件生态

中图法分类号: TP303

中文引用格式: 王昊天, 孙羽菲, 隋轶丞, 王嘉豪, 石昌青, 方建滨, 张玉志. MTTorch: 面向MT-3000芯片和Transformer模型的PyTorch算子库实现与优化. 软件学报, 2025, 36(8): 3896–3916. http://www.jos.org.cn/1000-9825/7244.htm

英文引用格式: Wang HT, Sun YF, Sui YC, Wang JH, Shi CQ, Fang JB, Zhang YZ. MTTorch: PyTorch Arithmetic Library Implementation and Optimization for MT-3000 Chip and Transformer Model. Ruan Jian Xue Bao/Journal of Software, 2025, 36(8): 3896–3916 (in Chinese). http://www.jos.org.cn/1000-9825/7244.htm

MTTorch: PyTorch Arithmetic Library Implementation and Optimization for MT-3000 Chip and Transformer Model

WANG Hao-Tian[1], SUN Yu-Fei[1], SUI Yi-Cheng[1], WANG Jia-Hao[1], SHI Chang-Qing[1], FANG Jian-Bin[2], ZHANG Yu-Zhi[1]

[1](College of Software, Nankai University, Tianjin 300450, China)

[2](College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China)

Abstract: With the rapid development of Transformer-based large models, computing power has gradually become a bottleneck in the development of this field. Research hotspots rely on how to accelerate and optimize the training performance of large language models based on the structural characteristics of accelerator hardware. This study proposes and implements MTTorch, a PyTorch extension library for the CPU+DSP heterogeneous architecture, which is applicable to the MT-3000 accelerator chip of the new generation of the Tianhe supercomputer. The core of MTTorch is a multi-core parallel operator library that vectorizes and optimizes the core operators during the training of Transformer-based models. Additionally, this study innovatively proposes a high-performance reduction algorithm and a ping-

* 基金项目: 国家重点研发计划 (2021YFB0300104); 先进计算与关键软件海河实验室科技项目 (22HHXCJC00001); 启元实验室创新基金 (2022-JCJO-LA-001-068)
收稿时间: 2023-11-13; 修改时间: 2024-03-13, 2024-05-13, 2024-06-14; 采用时间: 2024-06-25; jos 在线出版时间: 2024-12-31
CNKI 网络首发时间: 2025-01-02

© 中国科学院软件研究所 http://www.jos.org.cn

pong algorithm for multi-core DSP, significantly improving the computational performance of the operators. MTTorch also has good generality as it can be loaded as a dynamic link library for different versions of PyTorch without changing the native implementation of PyTorch. Extensive experiments show that the core operators implemented in this study have excellent performance on MT-3000 chip, achieving 8 times acceleration on a single DSP cluster. Using MTTorch for training tasks on multiple nodes achieves nearly linear acceleration, greatly improving the training efficiency of Transformer-based models on MT-3000 chip.

Key words: PyTorch; high performance computing; Transformer model; Tianhe supercomputer; CPU+DSP heterogeneous computing; software ecology

随着深度学习技术的发展, 深度神经网络取得了长足的进步, 模型的参数量以及训练数据也在飞速增长. 2020 年, OpenAI 发布的全球最大的预训练语言模型——GPT-3[1]参数量甚至达到了惊人的 1 750 亿, 2022 年发布的 ChatGPT[2]更是把大模型提升到了全新的高度. 当前的大语言模型都是在 Transformer 模型[3,4]的基础上改进而来. 然而, Transformer 模型的大规模训练需要巨大的计算资源. 因此, 如何合理利用已有计算设备, 尤其是国产大规模计算集群, 提高计算效率成为了一个关键问题[5,6].

为了提高深度学习模型对于计算设备的利用率以及简化模型搭建流程, 让研究人员更专注于模型设计, 众多知名互联网公司设计了多种机器学习框架. PyTorch 是一个基于 Python 的机器学习框架, 由 Facebook 开发并维护, 其采用命令式编程, 可见及所得, 接口相对比较统一, 使得用户能够更加灵活地构建和训练深度学习模型. 同时, PyTorch 还提供了丰富的工具包和 API, 使得用户能够轻松地进行数据处理、性能分析和调试等操作, 因而在工业界和学术界都得到了广泛的运用. 然而, 当前 PyTorch 官方只支持 CUDA (compute unified device architecture)[7]和 ROCm (radeon open compute) (https://rocmdocs.amd.com/en/latest/), 故而只能使用英伟达和 AMD 生产的 GPGPU 进行计算, 导致众多异构计算集群无法高效运行 PyTorch 任务, 令其在人工智能领域的巨大潜力难以发挥.

出于能源和功耗方面的考虑, 最先进的高性能计算平台往往采用低功耗嵌入式架构, 如数字信号处理器 (digital signal processor, DSP). MT-3000[8]是国防科技大学研发的一款用于高性能计算的异构多区域处理器, 搭载了 4 个 DSP 簇. 与 CPU、GPU 和 TPU 等设备不同, MT-3000 采用超长指令集 (very long instruction word, VLIW) 和加速阵列微架构, 每个加速器核心以 VLIW 的方式工作, 每 16 个加速器核心和一个控制核心被组织成一个加速阵列. 同时其也集成了直接存储器取存引擎 (direct memory access, DMA)[9] 用于内存操作. 基于 MT-3000 搭建的天河新一代超算系统[10]在高性能计算和深度学习领域都有着巨大的潜能. 同时, MT-3000 具备了完整的编程工具链和向量 C 语言支持, 使得用户可以在 C 语言层面充分利用 VLIW 和加速阵列微体系架构的组合潜力. 研究表明, 只有软件能够充分利用硬件特性, 加速器才能释放其潜力[11,12]. 因此, 面向新一代 DSP 加速设备开发 PyTorch 支持是利用众多高性能计算平台进行模型训练的必要一步.

本文面向 MT-3000 芯片的体系架构特点, 实现了一个 PyTorch 的扩展库——MTTorch. 由于 PyTorch 中包含的算子数量庞大, 结合训练大语言模型的需求, 本文聚焦于 Transformer 类模型, 对模型训练过程中的 15 个核心算子在 CPU+DSP 异构环境下进行精心优化.

本文的创新点和贡献如下.

(1) 提出并实现了面向 MT-3000 异构加速器的 PyTorch 扩展库——MTTorch. 结合 MT-3000 的体系结构特点, MTTorch 对 Transformer 类模型训练中的核心算子进行了实现和并行优化, 为新一代天河超算支持大模型训练打下了坚实的基础.

(2) 针对 MT-3000 体系结构的特点, 提出了面向多核 DSP 的高性能规约算法; 利用核内 DMA 和计算部件异步执行特性设计了乒乓算法. 基于上述优化策略, 并结合 DSP 上的向量寄存器特性, 有效提升了计算效率.

(3) 设计实现了面向 DSP 的设备管理模块以及线程组管理模块, 基于单例模式有效避免了 DSP 簇重复启闭造成的额外开销.

(4) 通过一系列实验, 验证了 MTTorch 所优化实现的算子均能正确运行, 并且获得了 2–8 倍的性能提升. 同时, 基于 MTTorch 的大模型训练任务展现出了接近线性的加速比.

© 中国科学院软件研究所 http://www.jos.org.cn

本文第 1 节给出深度学习算子和框架在不同设备上转换适配的相关工作. 第 2 节主要介绍了 MT-3000 体系架构以及编程模型的选择. 第 3 节介绍了 MTTorch 的整体架构以及核心算子的异构并行编程实现. 第 4 节设计实验验证了 MTTorch 在 MT-3000 芯片上的正确性和加速效果. 第 5 节对本文的工作进行总结并提出了未来的研究方向.

1 机器学习算子及框架在异构设备上扩展的相关工作

近年来, 大语言模型技术得到了飞速发展, 而语言模型的研究和应用却可能遭遇算力瓶颈. 传统应用的算力支撑更多地依靠 CPU, 而 AI 所需的智能算力在 CPU 之外, 还需要大量 GPU 参与. 为充分发挥通用加速器, 尤其是国产高性能异构计算集群在人工智能领域的巨大潜力, 众多研究者在算子和框架的移植优化领域做了大量工作.

李亿渊等人[13]通过自定义的矩阵分层分块技术, 在申威 26010 异构众核处理器上实现了高效的稀疏矩阵乘法操作. Yin 等人[14]在新一代数字信号处理器 FT-M7032 上结合硬件特性, 实现了一种可以自动装配微内核, 自动调整分块大小和并行策略的非规则通用矩阵乘法 (GEMM). 裴向东等人[15]针对飞腾异构多核 DSP 的体系架构特点与矩阵转置操作的特点, 利用 DSP 模 16 存储功能和隐式乒乓设计实现了适用于多种位宽的异构并行矩阵转置算法 ftmMT. 王庆林等人[16]针对飞腾多核处理器对 Winograd 快速卷积算法进行优化. 上述方法都是针对深度学习或科学计算中的某一个具体的算子, 如何在基于 DSP 的国产高性能异构设备上高效地使用深度学习框架训练模型还有待解决.

随着深度学习领域对算力需求的日益增长, 越来越多的计算设备被运用于人工智能领域. 如何将基于 CUDA 的深度学习框架移植到不同的设备上受到了大量关注. OpenCL Caffe 是 AMD 开发的基于 OpenCL 的并行深度学习框架[17], 它可以利用不同平台上的硬件资源来实现高效的计算. 它具有跨平台、高效能的优点, 可以在不同的硬件设备上运行, 例如 GPU、CPU 和 FPGA 等. OpenCL Caffe 还提供了一些优化技术, 例如基于局部内存的优化、图像数据的预处理和统一的内存访问等, 以提高性能和效率. Chen 等人[18]针对 MT-3000 处理器提出了一个高效的卷积神经网络推理框架 PCIF, 该框架支持通过 ONNX 导入和优化训练好的深度学习模型, 通过计算图融合和多核并行向量化技术提高计算效率. 实验结果显示, PCIF 能有效加速神经网络的推理过程. 然而, 该框架目前仅实现了卷积神经网络中常用算子的前向传播功能, 尚未实现相关算子的反向传播功能. 因此 PCIF 仅适用于推理场景, 不支持模型训练. 此外, 该框架只适配了卷积网络相关的算子, 不支持 Transformer 类大语言模型的训练. 陈锐等人[19]探索了 CUDA 核函数向 OpenCL 核函数转换的基本流程, 并通过单例模式初始化, 缓存编辑文件以及使用规约操作代替原子操作等优化方式, 实现了基于 OpenCL 的 TensorFlow, 在 GPU、华为鲲鹏-920CPU 等设备上具有通用性. 然而, 由于 OpenCL 的通用性, 算子的实现难以针对特定的体系架构进行优化, 且当前对于 PyTorch 框架如何移植到 CPU+DSP 异构环境上还有待研究, 导致现有工作均无法充分发挥 DSP 异构芯片的性能.

本文的工作面向 MT-3000 处理器的异构众核架构, 提供了支持 DSP+CPU 异构训练的 PyTorch 扩展库——MTTorch, 针对训练 Transformer 类模型常用的 15 个算子, 设计了并行算法. 大量实验证明了 MTTorch 相比基于 CPU 同构环境的 PyTorch 有着更好的性能和可扩展性.

2 MT-3000 软硬件架构

2.1 MT-3000 硬件体系架构

MT-3000 是一款高性能异构多区域处理器, 由多核 CPU 和 4 个 DSP 簇构成, 是天河新一代超算系统的重要加速部件. 其 CPU 类似于简化版的 FT-2000 Plus 处理器, 具有 16 个兼容 ARMv8 架构的 CPU 核心, 用于进程管理和通信, 并运行操作系统. MT-3000 的加速核被平均分为 4 个自动加速集群 (DSP), 每一个 DSP 簇包含 24 个由 1 个控制核心和 16 个加速器核心组成的 DSP 核、一个大小为 6 MB 的全局共享内存 (global shared memory, GSM)、高带宽共享内存 (HBSM) 和芯片上的 DDR 内存空间. MT-3000 体系架构如图 1 所示.

© 中国科学院软件研究所 http://www.jos.org.cn

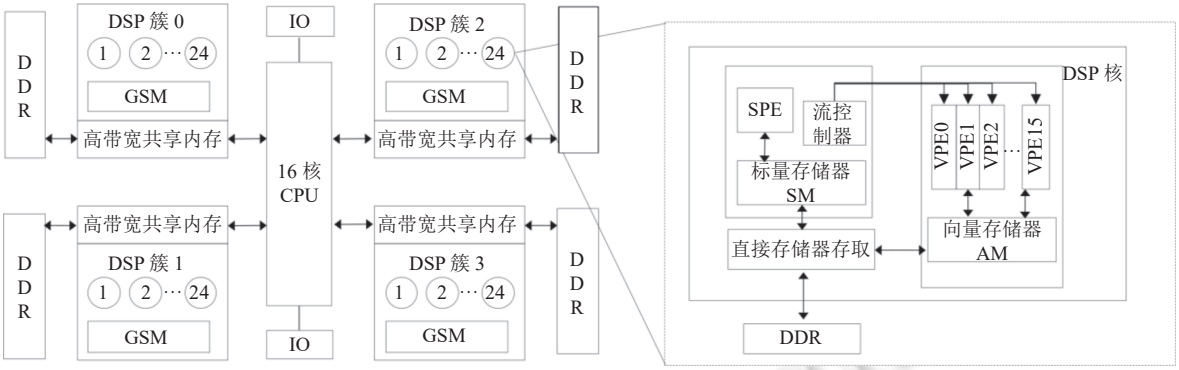

图 1　MT-3000 体系架构

每个 DSP 核基于 VLIW 架构实现, 可以同时执行 5 条标量指令和 6 条向量指令, 包含了标量处理单元 (SPU)、向量处理单元 (VPU) 和 DMA 引擎等主要结构. 标量处理单元主要由标量处理部件 (scalar processing elements, SPE), 用于控制向量运算的流控制器以及一个大小为 64 KB 的标量存储器 (scalar memory, SM) 组成. VPU 是 DSP 的主要算力来源, 包含了一个 768 KB 的向量存储器 (array memory, AM) 和 16 个单指令多数据 (SIMD) 模式执行的向量处理部件 (vector processing elements, VPE). VPE 的最小访存位宽为 64, 其上包含了 3 个高性能的浮点乘加单元.

在 MT-3000 上, 每个加速集群都是彼此独立地执行, 通用区域的 CPU 可以访问不同加速集群的所有 HBSM 和 DDR 空间, 而每个 DSP 只能访问它们自己的加速集群内的 GSM、HBSM 和 DDR, 用以完成计算密集型任务. 进一步, 在一个 DSP 中, 不同 DSP 核之间的 SM 和 AM 均无法共享, 核间数据交互一般通过共享内存 GSM 进行. DMA 引擎存在于每个 DSP 核中, 不同核之间互不干扰.

MT-3000 DDR 之间 DMA 的实测访存带宽为 16.22 GB/s, 在每个 DSP 簇上 DMA 在 DDR 和 AM 间的实测带宽为 42.62 GB/s, 而 AM 间访存带宽高达 921.6 GB/s[3], 因此在执行过程中, 尽可能要将运算数据先加载到 AM 或者 SM 中再进行计算. 表 1 展示了 MT-3000 中不同级别的内存及其特点, 在设计程序时应该尽可能为不同的功能使用最合适的内存区域.

表 1　MT-3000 DSP 核中不同内存

名称	大小	主要作用
DDR	32 GB	CPU和DSP同时可读可写内存, 带宽比较宝贵, 应避免频繁读写
GSM	6 MB	所有DSP核可见的2级缓存, 用于核间数据交互
SM	64 KB	每个核私有的标量存储器, 只能使用控制核心访问
AM	768 KB	向量存储器, 最小访存位宽为64位

2.2　MT-3000 编程模型介绍

MT-3000 为用户提供了不同层级的编程模型, 既有操作系统级别的汇编指令集, 低层级的加速数组管理接口和专业数学运算库. 也有高层级的异构多线程库 Hthreads[20]和 OpenCL 标准的实现 MOCL3[21].

MOCL3 作为 MT-3000 的 OpenCL 1.2 标准实现, 通过将局部存储映射到片上存储器和利用异步工作组复制内置函数, 显著提升了内存带宽和数据传输效率. 实验证明, 使用局部存储优化后, 性能提升约 2.8 倍[20].

Hthreads 是一个低开销和高可用性的异构编程接口, 既具有足够的灵活性, 可以在 C++ 这个层级进行开发, 又可以满足性能需求. 该接口由主机端 API 和设备端 API 组成, 通过中断和共享内存进行通信. 主机端 API 主要由线程组管理模块和调试性能分析模块组成, 使用类似 POSIX 的线程接口来管理加速区并降低内核启动开销, 性能上优于切换时钟模式. 设备端 API 主要由系统调用模块和资源管理模块组成, 提供丰富的硬件功能接口, 以满足

© 中国科学院软件研究所 http://www.jos.org.cn

用户对计算任务的需求. Hthreads 通过引入 DMA 技术实现高效的数据传输. 在优化案例中, Hthreads 性能显著优于 MOCL3[20]. 总体而言, Hthreads 的设计更贴近 MT-3000 芯片的底层架构, 在允许用户对 DSP 进行精细控制、降低内核启动开销的同时, 也兼顾了可编程性和灵活性.

图 2 展示了 MT-3000 的各层级编程模型, 越接近底层编程模型效率越高, 但是其开发和维护难度也就越大. 综合考虑性能、可编程性、可读性等因素, 本文选择可以嵌入到 C 语言中的高级编程模型 Hthreads 作为开发模型.

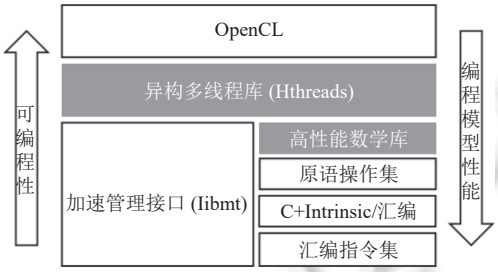

图 2 MT-3000 用户层编程模型

3 MTTorch 的实现与优化

3.1 MTTorch 设计背景

原生 PyTorch 框架中囊括了近千个算子, 为框架中所有算子都进行精心优化是一个庞大的工程. 考虑到大模型训练对于计算需求的迫切性, 本文工作聚焦于在新一代天河超算上实现和优化 Transformer 类大语言模型训练任务.

首先, 我们在 MT-3000 的 CPU 上训练 PyTorch Examples (https://github.com/pytorch/examples) 官方项目中的 Transformer 模型至 loss 下降趋于稳定, 并使用 PyTorch 中的 profile 工具详细统计分析了模型训练过程中的各算子的平均耗时 (如表 2 所示). 前 15 个算子的总耗时占比就高达 82% (如图 3 所示). 这为后续的算子优化提供了明确的方向和依据. 本文将这 15 个算子定义为 Transformer 类模型的核心算子, 优化好这 15 个算子就可以极大地提升模型的计算性能.

表 2 Transformer 类模型中耗时大的算子

函数名称	算子功能	时间占比 (%)	平均执行时间 (ms)
mm	矩阵乘法	46.20	19.454
softmax/log_softmax	归一化指数函数	12.47	141.763
add/sub/mul/div	加, 减, 乘, 除	11.24	3.87
softmax_back/log_softmax_back	softmax的反向传播	3.95	44.959
bmm	批矩阵乘法	2.41	3.309
native layer_norm	层内归一化	2.00	2.147
sum	Tensor求和	1.94	2.511
copy_	数据拷贝	1.88	2.452
ReLU_back/GELU_back	激活函数反向传播	0.95	28.01
_cat	Tensor拼接	0.82	3.23
ReLU/GELU	激活函数	0.57	15.91

表 2 中算子可分为 3 类. 第 1 类是如 cat、copy 等访存密集型操作, 并不涉及计算, 可以直接使用 DMA 进行实现. 第 2 类为 pointwise 类型算子, 即 Tensor 中对应位置进行计算操作, 主要为计算耗时, 访存消耗较小. 第 3 类为需要进行规约操作的算子, 如 layer_norm. 这一类算子在计算和访存方面都存在瓶颈问题, 且在规约过程中可能遇到数值溢出. 综上所述, 本文面向 MT-3000 的体系结构特性, 对耗时占比高的 15 个核心算子进行了精心优化.

© 中国科学院软件研究所 http://www.jos.org.cn

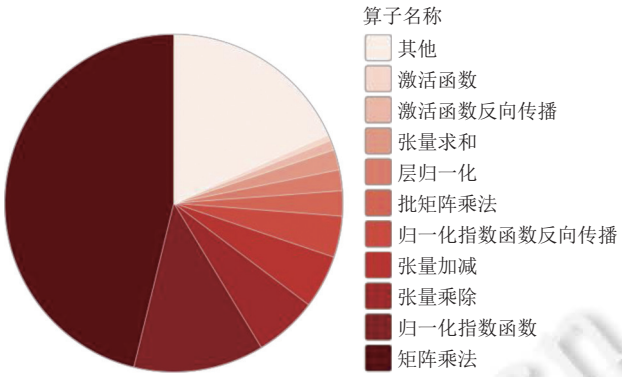

图 3　MTTorch 核心算子在训练任务中的计算时长占比情况

3.2　MTTorch 的总体设计思路

MTTorch 本质上是对 PyTorch 官方实现的一个扩展, 通过将 MTTorch 以动态链接库的形式进行挂载, 即可在 MT-3000 上实现异构训练. MTTorch 由设备管理模块和算子库两部分组成, 其整体架构如图 4 所示.

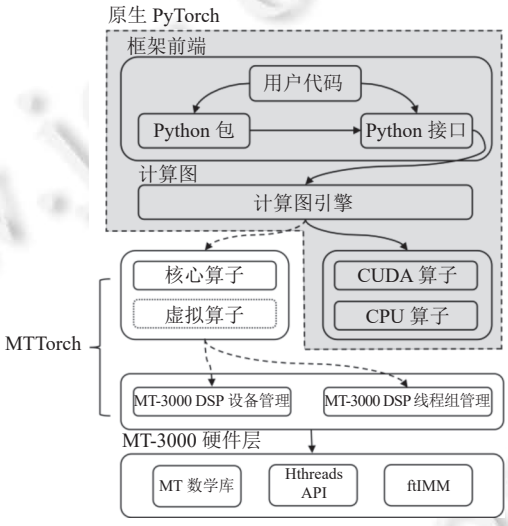

图 4　MTTorch 整体架构

设备管理模块保证了深度学习框架后端可以准确、高效地调用 MT-3000 执行深度学习任务. 每当任务被启动, MTTorch 首先需要通过执行 hthread_dev_open() 初始化设备, 随后利用 hthread_dat_load() 挂载包含设备端核函数的.dat 文件. 然而, 考虑到在一次模型训练任务中, 设备仅需开启一次. 为了避免不必要的重复操作, 本文特别设计了基于单例模式的设备管理类来维护一个 DSP 对应的句柄. 单例模式的核心思想在于确保某个类在整个系统中仅存在一个实例, 并提供全局访问点, 从而防止重复初始化. 当 MTTorch 需要调用设备运算时, 会首先检查 DSP 句柄是否为空. 如果句柄为空, 即设备尚未启动, 则执行相应的 API 来初始化设备并进行内存检查; 若句柄非空, 则直接跳过启动阶段进入后续的运行流程. 这样的设计不仅提高了效率, 还确保了系统的稳定性和资源的高效利用.

同样, 当使用 Hthreads 编程模型调用 DSP 计算时, 需要先根据对应的 DSP 句柄和要申请的线程创建线程组, 并返回一个 threadID 以唯一标识该线程组. 然而, hthreads_group_create 函数本身会产生约 1.2 ms 的开销. 为了避免这种开销, 本文使用单例模式设计了一个线程组管理类, 如图 5 所示, 该类可以缓存已有的 threadID. 如果当前

© 中国科学院软件研究所 http://www.jos.org.cn

需要的资源与已有的线程组匹配, 则可以直接调用 hthreads_group_exec 函数进行计算, 无需重新创建线程组. 因为 hthreads_group_exec 函数的固有开销为约 0.6 ms, 所以使用这种方法可以减少 50% 的固有启动开销.

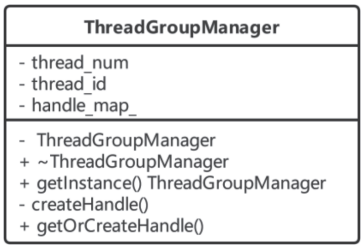

图 5 Hthreads 线程组单例类图

在算子库层面, MTTorch 采用 CPU+DSP 异构计算设计. 对于核心算子, 采用 MTTorch 的 Hthreads 编程模型进行并行优化. 对于训练大语言模型时占比极少的算子和并行实现带不来性能优化的算子采用虚拟算子的方法实现. 虚拟算子是指注册在 DSP 后端但仍使用 CPU 进行计算的算子. 由于一个计算图上所有算子的设备后端需要是一致的, 因此这类算子需要在 DSP 后端进行注册, 但实际仍在 CPU 端执行.

不适宜并行优化的算子, 主要有内存中非连续的 Tensor 参与计算和 Tensor 数据量过小这两种情况. 非连续 Tensor 难以实现批量访存, 无法充分利用向量处理单元; 输入 Tensor 过小时, 整体计算时间也就是几毫秒的级别, 由于 DSP 线程组启动时有 1 ms 左右的固有开销, 导致其会弱化加速的效果, 甚至没有加速效果. MTTorch 中算子的执行流程如图 6 所示.

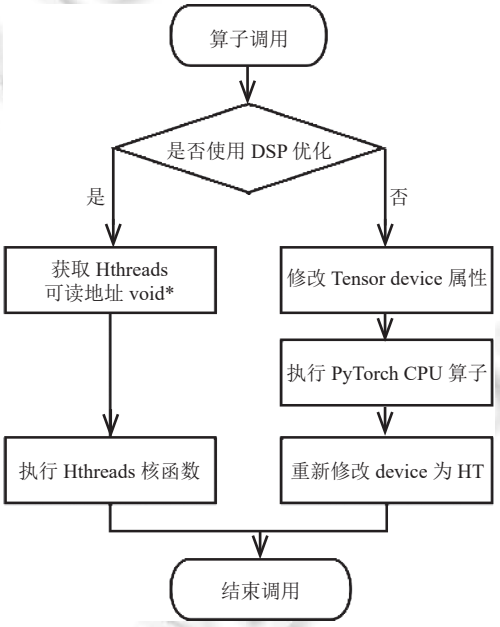

图 6 MTTorch 中算子执行流程

此外, 本文将 ftIMM[14]集成到了 MTTorch 框架中, 通过对 ftIMM::SGEMM 的调用实现了适用于模型训练的高效 mm 和 bmm 算子. 为了最大限度地利用 DMA 带宽进行数据存取, 用户必须确保每次 DMA 拷贝的内存空间尽可能连续, 并且大小为 1 024 位的倍数. 因此在 MTTorch 中, 矩阵的存储和遍历均采用了行主序 (Row-major) 的方式, 以确保数据在内存中的布局与 DMA 的存取方式相匹配, 从而提升数据处理的效率. 由于高效的矩阵乘法实现需要精心设计, 充分利用指令的流水线, 这对于 Hthreads 这一层级的编程模型有着很大的挑战. 我们将在后续的

© 中国科学院软件研究所 http://www.jos.org.cn

研究中探索基于 Hthreads 的高性能矩阵乘法的实现方式.

MTTorch 的设计具有很好的通用性. MTTorch 中所有算子均使用 PyTorch 原生的 TORCH_LIBRARY_IMPL 方法在 PyTorch 后端进行注册, 以供图引擎调用. 其实现的算子以扩展库的形式编译和原生 PyTorch 框架解耦. 用户只需通过 torch.ops.load_library 方法在 Python 端将编译好的 MTTorch 动态链接库进行挂载, 即可在 MT-3000 的 DSP 簇上调用算子进行高效训练. 本文的设计思路几乎未改变 PyTorch 的源码和框架结构, 因此广泛适用于各版本的 PyTorch 框架.

3.3 虚拟算子

在实现虚拟算子时, 一种直观的实现方法是在算子内将输入的 Tensor 搬移到 CPU 控制的 DDR 区域执行 CPU 分支的算子, 再在计算结束之后将结果搬移到 DSP 可读写的 DDR 区域. 然而不同设备之间的切换需要执行 memcpy 函数进行内存拷贝. 经过测试, MT-3000 上 DMA 对 DDR 的访存带宽只有 16.22 GB/s, 这导致内存拷贝所占用的时间是不可忽视的. 针对这类算子, 本文结合 MT-3000 的设备特性, 提出了一种无需内存拷贝的虚拟算子实现方法. 在 PyTorch 中, Tensor 在设备上的储存分为头信息区和存储区两个部分, 头信息区保存 Tensor 的形状 (size)、步长 (stride)、设备 (device)、数据的索引等信息; 存储区 (storage) 保存该 Tensor 的真实物理地址. 得益于 MT-3000 片上的 CPU 可以读写所有 DSP 簇所占用的 DDR 内存, 开发人员只需要修改头信息区的 device 属性便可以创建一个设备为 CPU 的 Tensor, 由于该 Tensor 存储区指针并没有变, 从而避免了在切换不同后端时候的频繁内存拷贝.

3.4 计算密集型算子向量化

本文采用向量化优化来处理点对点计算操作 (pointwise), 如加法 (add)、减法 (sub)、除法 (div) 和乘法 (mul). MT-3000 的 DSP 性能主要依赖于其矢量处理单元 (VPU). 每个 VPU 包含 16 个加速核心 (VPE), 它们以同步的方式并行工作, 每个核心能够处理 64 位宽的双精度数据, 以充分发挥 VLIW 微架构的计算潜力. 对于单精度数据, 一个 VPE 能够将 2 个单精度数打包到一个 64 位宽的数据中, 因此一个向量可以同时支持 32 个单精度数的计算. 这意味着 DSP 具有出色的向量化支持. 在算子向量化实现时, 应该注意算子原生实现应满足以下特点.

(1) 访存地址无冲突或程序中不存在原子操作;

(2) 向量循环体的循环次数可计算且循环体只有一个出口和入口.

算子实现中包含了上述特点即可使用向量化进行优化. 此外, 一个 DSP 簇上的 AM 缓存大小为 768 KB, 因此在设计实现时应该根据输入大小为 Tensor 分段处理, 防止 AM 内存溢出. 具体实现过程见算法 1.

算法 1. 自适应大小的向量化核函数.

输入: 矩阵 A, 总长度 L;
输出: 矩阵 C.

```
1. 计算均分给每个核的 Tensor 片段长度 L_c;
2. FOR m=0; L_C; L DO in parallel
3.    coreID = get_thread_id();
4.    offset=min(coreID×L_c, L-coreID×L_c);
5.    cacheLen = MaxVectorCount×32 //单次 load 到 AM 大小;
6.    FOR i=0; tmpLen; len – 1 DO
7.       tmpLen = (L – i) > cacheLen ? cacheLen : (L – i);
8.       vector_load(A[offset+i]) //子矩阵传入 AM 空间;
9.       FOR j=0;1;  L_c/32 DO
10.          Vector C++ execute;
```

© 中国科学院软件研究所 http://www.jos.org.cn

11. **END**
12. *vector_store*(*C*[*offset*+*i*]) //将结果保存到 DDR;
13. **END**
14. **END**

在算子中根据输入输出以及计算中间结果的大小, MaxVectorCount 可以取 128, 256, 512 或 1024. 一个 VPE 中集成了 3 个浮点乘加计算单元, 为了充分利用处理器 VLIW 的性能, 在实现时要将加减操作 (a–b) 转换为乘加和乘减 (a×1–b) 操作.

对于计算过程中的常数变量, 本文通过 Hthreads 的广播指令将其广播成 VPU 中的一个向量常量. 然而由于硬件的特性, 存储器一次可处理的最低位宽为 64 位, 单个单精度数据并不支持广播. 本文使用 C 语言中的联合体 union 实现 32 位常数到 64 位的打包和解包, 它允许将多个不同类型的成员变量存储在同一个内存空间中. 一个 union 的长度为 64 位宽, 代表 1 个双精度浮点数或者长度为 2 的单精度浮点数组. 实现代码如图 7 所示.

```
1.  typedef union{
2.    double d;
3.    float f[2];
4.  } word;
5.  lvector float bcast_f32(float var){
6.    word w;
7. w.f[0] = var;
8. w.f[1] = var;
9. double tmp = w.d;
10.  return (lvector float)bcast(tmp);
11.  }
```

图 7　单精度浮点数打包

3.5　激活函数及其反向传播算子实现

激活函数可以使模型更加稳定, 显著提高模型性能, 是深度神经网络中不可缺少的模块. 本文实现了语言模型中最常用的两个激活函数.

ReLU 激活函数是一种常用的非线性函数, 其表达式为 $y = \max(x, 0)$, 具有计算速度快, 训练过程稳定的特点, 因此广泛运用在深度学习模型中. ReLU 前向传播和反向传播公式为:

$$\begin{aligned} ReLU(x) &= \max(0, x) \\ \mathrm{d}x &= (y > 0)\ ?\ \mathrm{d}y : 0 \end{aligned} \tag{1}$$

由于硬件位宽限制, 向量 *C* 接口的条件赋值语句仅支持双精度数据. 对于单精度下的 ReLU 反向传播中的条件赋值操作, 本文采用位运算实现 *res* = (*A*>*B*) ? *C* : *D*, 得益于 VLIW 的架构, 取得了明显的加速效果, 见算法 2.

算法 2. 基于位运算的条件赋值方法.

输入: *A*, *B*, *C*, *D*;
输出: *res*.

1. *vo* = *gt*(*A*, *B*); //判断是否赋值
2. *vzero* = *mov* 0 to *VPEs*; //构造全 0 向量
3. *vot* = *sub*(*vo*, *vzero*); //条件向量格式化为全 0 或全 1
4. *x*64 = (lvector unsigned long int)C;
5. *y*64 = (lvector unsigned long int)*D*;
6. *vo*64 = (lvector unsigned long int)*vot*; //强制类型转换
7. //位运算赋值

© 中国科学院软件研究所 http://www.jos.org.cn

```
8. von = not(vo64);  //逻辑非
9. tmp1 = and(x64, vo64);  //逻辑与
10. tmp2 = and(y64, von);  //逻辑与
11. rett = or(tmp1, tmp2);  //逻辑或
12. res = (lvector float)(rett);  //强制类型转换
```

GELU 也是一种常用的激活函数, 相较于传统的激活函数如 Sigmoid 和 ReLU, GELU 在神经网络中具有更好的性能, 同时也更易于优化. 它在训练时收敛速度更快, 并且比 ReLU 更能适应各种深度神经网络结构, 因此被广泛应用于自然语言处理、计算机视觉等领域, 其公式为:

$$GELU(x)=0.5\times x\times\left(1+\mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right) \tag{2}$$

其中, erf 为高斯误差函数. 其反向传播可由下式得到.

$$\frac{\mathrm{d}y}{\mathrm{d}x}=0.5\times\left(1+\mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right)+\frac{x}{\sqrt{2\pi}}\exp\frac{-x^2}{2} \tag{3}$$

MT-3000 提供了一系列高性能数学库 (HPML), 其中的数值计算库 libvm 包含了向量的 erf 和 exp 操作. 利用 libvm 和算法 1 防止 AM 内存溢出的思路, 本文使用向量指令实现了高效的 GELU 前向传播以及反向传播算子.

3.6 基于规约的算子实现与优化

归约操作是指将一组数据中的所有元素进行某种计算 (例如求和、求最大值、最小值等), 并得到单一的结果. 在 PyTorch 中, 存在着大量基于规约的算子. 本文结合 MT-3000 DSP 的硬件特性, 设计了一个高效的多核规约流程, 并借助其实现了高效的 softmax 和 layer_norm 算子.

3.6.1 MT-3000 多核规约

本节首先介绍了实现上述算子最关键的步骤——基于多核的并行规约算法. 归约的串行操作非常简单, 但是由于每次循环都需要利用上一次循环的结果, 因此无法直接向量化. 二分法并行规约是规约算法的常见思路. 其将源数组划分为多个子数组分别规约, 再对子数组的结果进行规约, 如此递归得到结果. 然而, 当前的规约算法大多是针对 CUDA 平台或特定的指令集, 并不适用于 MT-3000 的异构众核架构. 本文结合 MT-3000 的硬件架构, 实现了一个 DSP 上的多核向量化规约算法, 流程如图 8 所示.

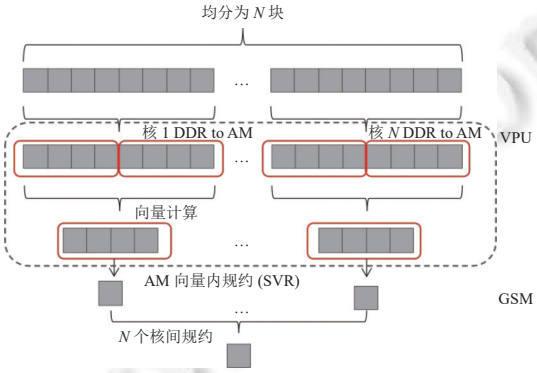

图 8 MT-3000 DSP 多核规约流程

3.6.1.1 核内分段计算

算法首先根据参与规约的核数 N 将数组分为 N 份. 值得注意的是, 由于 VPE 位宽限制, 每一份的长度应该为偶数. 本文使用 DMA 指令将分好的子数组搬移到 AM 存储器中用于接下来的计算. 使用 DMA 一方面可以充分利用各级内存间的带宽, 另一方面也方便开发人员针对访存进行更精细的优化, 这将会在第 3.6.3 节中展开. 搬移

© 中国科学院软件研究所 http://www.jos.org.cn

结束后, 对于各个核 AM 中的数组以向量的形式循环执行计算, 将该核子数组规约到一个 1 024 位宽的向量中 (图 8 中框内部分).

3.6.1.2 VPU 内规约

通过上述方法, 我们在 24 个 DSP 核上都得到了对应分区的规约结果并且保存在对应 VPU 的一个 VPE 上. 对于双精度 (double) 的数据, 可以直接使用 mov_from_svrX 接口, 将 VPE 中 16 个数卸载到 SVR 寄存器进行合并. 然而, 对于单精度浮点数, 由于 VPE 的最小访存单位为 64 位, 因此无法直接进行对应的操作. 同图 7 中的思路, 本文先将取出来的 64 位解包成 2 个对应的单精度浮点数, 再进行合并.

3.6.1.3 DSP 簇核间规约

经过以上两步, 我们在 N 个计算核上分别得到了对应分段的规约结果, 保存为 N 个标量值. 进一步, 本文实现了多核之间的数据规约算法. 由于核间的规约需要在核间交互数据, 而多核可共享的 DDR 又是带宽受限的. 因此, 利用 GSM 缓存比 DDR 内存访存带宽更大的特性, 本文将子数组规约结果和核间规约的临时共享地址 *shared_mem* 都创建在所有核共享的 GSM 缓存中, 以减少对 DDR 的频繁访存.

具体来说, 本文将 DSP 簇中的不同核分成若干个子组, 每个子组内的计算核执行一定的归约操作 (例如求和), 然后将子组的结果进行合并, 得到一个较小的结果集合. 然后再将较小的结果集合分成若干个子组, 重复上述操作, 直到最终只剩下一个结果. 核间数据规约算法流程见算法 3. 其中, *core_num* 代表本机规约使用的核心数, *shared_mem* 是位于 GSM 的中间变量存储地址, warp_size 是常量, 一般为 32.

算法 3. 核间数据规约.

输入: *core_num*, *sub_reduce*, *shared_mem*, warp_size;

输出: *sub_reduce*.

```
1. FOR each core DO in parallel:
2.     int coreID = get_thread_id();
3.     FOR i = (warp_size >>1) TO 0 STEP >>=1 DO:
4.         shared_mem[coreID] = sub_reduce
5.         barrier;
6.         int temp = coreID+i<core_num?shared_mem[coreID+i] : 0;
7.         Reduce_op(sub_reduce, temp);
8.     END
9. END
```

3.6.2 基于规约的算子实现

3.6.2.1 sum 以及 softmax

基于上述的规约算法, 开发人员自然而然可以实现 sum 算子. 对于 softmax 算子和 log_softmax 算子, 我们首先给出其计算公式:

$$\begin{cases} \sigma_{\mathrm{SM}}(x_i)=\dfrac{\exp(x_i-x_{\max})}{\sum\limits_{j=1}^{K}\exp(x_j-x_{\max})} \\ \sigma_{\mathrm{LSM}}(x_i)=(x_i-x_{\max})-\log\left(\sum\limits_{j=1}^{K}\exp(x_j-x_{\max})\right) \end{cases} \tag{4}$$

区别于原始的 softmax 公式, 本文将 e 的指数减去所有输入的最大值, 缩放成小于等于 0 的数, 防止造成数值溢出. 反向传播可由:

© 中国科学院软件研究所 http://www.jos.org.cn

$$\mathrm{d}x_i = y_i \times \left(\mathrm{d}y_i - \sum_{j=1}^{K} (\mathrm{d}y_j \times y_j) \right), \mathrm{d}x_i^{\log} = \mathrm{d}y_i - \exp(y_i) \times \sum_{j=1}^{K} \mathrm{d}y_j \tag{5}$$

计算得到. 对于前向传播需要进行规约操作求出数组的最大值和总和, 再按照公式进行向量化计算. 对于反向传播则只需要规约求和直接计算即可, 因为 y_i 为 softmax 之后的计算结果, 取值范围为 [0–1], 不会造成溢出.

3.6.2.2 Layer_norm

PyTorch 中的 Layer_Normalization (层归一化) 是一种用于深度神经网络的归一化技术, 其在每个样本的特征维度上进行归一化. Layer_norm 适用于处理变长序列数据, 如文本数据, 可以提高模型的稳定性和收敛速度, 是大型语言模型中必不可少的一环. 计算过程涉及将输入张量沿指定维度计算均值和标准差, 并使用它们进行归一化. 其计算公式为:

$$y = \frac{x - E(x)}{\sqrt{Var(x) - \epsilon}} \times \gamma + \beta \tag{6}$$

其中, $E(x)$ 代表数据的均值, $Var(x)$ 代表数据的标准差, ϵ 是一个用于保持数值稳定的常数, γ 和 β 是该层的可学习参数.

在传统的计算均值和标准差的方法中, 通常需要将所有数据相加, 然后除以数据个数来得到均值, 再遍历每一个样本数据求得标准差. 然而, 当数据量非常大时, 将所有数据相加可能导致精度溢出的问题, 同时进行两次遍历也会造成较大的时间开销. 为了解决这个问题, 并且实现并行化计算, 本文在 DSP 上实现了适用于多核并行的 Welford 迭代计算算法. 程序只需要对数据进行一次遍历, 并利用迭代公式:

$$\begin{cases} Mean(x_k) = Mean(x_{k-1}) + \dfrac{x_k - Mean(x_{k-1})}{k} \\ S(x_k) = S(x_{k-1}) + (x_k - Mean(x_{k-1})) \times (x_k - Mean(x_k)) \\ Var(x_k) = \dfrac{S(x_k)}{k} \end{cases} \tag{7}$$

即可计算 k 个数据的均值和标准差, 避免了精度溢出的问题. 其中, $S(x)$ 是迭代计算标准差的中间结果. 此外, Welford 算法的合并公式:

$$\begin{cases} delta = Mean(x_2) - Mean(x_1) \\ Var(x) = Var(x_1) + Var(x_2) + delta^2 \times n_1 \times \dfrac{n_2}{n_1 + n_2} \\ Mean(x) = Mean(x_1) + delta \times \dfrac{n_2}{n_1 + n_2} \end{cases} \tag{8}$$

也使得可以将数据进行分段处理, 不同的数据段间互不干涉, 从而可以通过一次规约来求得结果, 其中 n_1 和 n_2 代表要合并的两段子数据的长度.

具体来说, 在核内分段计算阶段利用迭代公式将每个核内的数据在不同向量间进行迭代, 最终得到长为 32 的迭代结果. 接着, 在 VPU 内规约和 DSP 簇核间规约时采用合并公式, 将不同数据段迭代得到的结果最终规约为一个值.

3.6.3 访存优化

在 MT-3000 DSP 上, 每个核都有若干个直接存储器存取 (DMA). 使用 DMA 在各级存储之间搬移数据无需占用计算核心, 可以使得计算和访存最大化的重叠, 提高性能.

乒乓算法是一种经典的访存优化技术, 用于实现异步情况下计算和数据传输之间的重叠. 本文在访存时采用乒乓算法, 通过手动编程实现通信和计算最大限度的互相隐藏, 从而取得良好的加速效果, 乒乓算法具体流程如算法 4 所示. DSP 上实现乒乓操作的核心是在 VPU 的 AM 存储器上申请数据所需大小 2 倍的空间, 以存放 2 份相同大小但不同时序的数据互为对方的缓冲. 具体来说: 除去第 1 轮次的数据读取和最后一轮次的数据写回, DSP 核心在进行本轮的数据计算同时将下一轮数据通过异步 DMA 加载到 AM 中, *dma_p2p* & *dma_wait*; VPU 计算完成后, 再通过异步 DMA 方式将本线程负责的数据块从 AM 拷贝回 DDR. 异步 DMA 方式将计算与主存访问相重叠.

© 中国科学院软件研究所 http://www.jos.org.cn

算法 4 展示了 DDR 加载到 AM 的乒乓流程, 数据写回 DDR 的流程同理, 在此不再赘述.

算法 4. DSP 乒乓算法.

输入: 矩阵 *A*, 模型批次 batch_size, 每个批次的数据量 *L*;
输出: 矩阵 *C*.

```
1. coreID = get_thread_id();
2. offset=min(coreID×L_c, L-coreID×L_c); //批次内偏移量
3. malloc buffer1, buffer2, buffer3, buffer4; //双缓冲区设置
4. FOR m=0 to batch_size DO in parallel
5.    A_b = A + m×L + offset; //每批次单核处理数据
6.    if m==0 then
7.       ch1 = dma_p2p(A_b, buffer1); //dma 异步传输
8.       dma_wait(ch1);
9.       ch2 = dma_p2p(A_b+L, buffer2); //预取下一批次数据
10.      else
11.      if m%2==0 then
12.         dma_wait(ch1); //异步等待
13.         if m ! = batch_size–1 then
14.            ch2 = dma_p2p(A_b+L, buffer2);
15.         end if
16.      else
17.         dma_wait(ch2);
18.         if m ! = batch_size–1 then
19.            ch1 = dma_p2p(A_b+L, buffer1);
20.         end if
21.      end if
22.   end if
23.   if m%2==0 then
24.      buffer3 = kernel_compute(buffer1);
25.   else
26.      buffer4 = kernel_compute(buffer2);
27.   end if
28. END
```

需要注意的是, 虽然乒乓操作可以通过手动循环预取降低访存延迟, 但双缓冲区的设计也使得 AM 存储器的实际存储空间减半, 在编程时应该注意分块的大小防止内存溢出. 例如在计算 softmax 时, AM 存储器需存储当前输入, 下一批次输入, 当前输出, 下一批次输出以及计算中间结果共 5 个数据块. 此时每个核处理的向量数量应该小于 512 个.

4 实验分析

本文设计并完成了一系列实验回答以下问题:

© 中国科学院软件研究所 http://www.jos.org.cn

RQ1: MTTorch 中的算子的正确性如何?

RQ2: MTTorch 中的虚拟算子是否避免了引入额外开销?

RQ3: MTTorch 中核心算子的性能提升效果如何?

RQ4: 使用 MTTorch 进行模型训练时的性能和加速比如何?

本文在由 MT-3000 芯片构建的天河新一代超算系统上进行测试. 使用 PyTorch 1.10 (https://pytorch.org/docs/1.10/) 版本作为软件环境, 以 CPU 同构环境下的 PyTorch 作为基准线, 并在编译过程中集成了 OpenBLAS (https://github.com/xianyi/OpenBLAS) 等 CPU 上的高性能库. 为公平起见, 本文在单个 MT-3000 片上用 4 个 CPU 核和单个 DSP 簇进行比较. 区别于科学计算, 深度学习训练任务的一大不同在于任务对加速器内存的需求更大. MT-3000 片上 DDR 总量为 64 GB, 常规状态下, 每块 DSP 可分得的内存大小为 12 GB 左右. 为了保证大语言模型训练可以更稳定地运行, 只启用一块 DSP 簇, 保证其可支配 48 GB 内存, 防止内存溢出 (out of memory, OOM). 在算子测试阶段, 采用 Python 的单元测试框架 unittes (https://docs.python.org/3/library/unittest.html) 进行单元测试, 并使用 PyTorch 中的 profile 工具对算子性能进行分析. 在模型测试阶段, 本文在单节点和多节点情况下分别进行了模型训练性能和可扩展性测试. 考虑到 MTTorch 的设计模式, 本文的方法也适用于其他 PyTorch 版本.

4.1 RQ1: MTTorch 中算子的正确性

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, 用户安装 Python 之后可以通过 import unittest 引入并直接使用. 利用 unittset 包, 对 15 个核心算子和 21 个大模型所需的虚拟算子在 3 个不同数据规模下进行了单元测试, 基准为 CPU 算子的计算结果. 本文采用 torch.testing.assert_close 判断结果是否相等, 相对误差阈值设置为 1E–06, 这是主流深度学习框架进行测试时候采用的通用值. 由于虚拟算子的实现逻辑一致, 因此在实验中以 nll_loss_forward 为代表进行说明, *表示该算子为虚拟算子. 表 3 展示了 MTTorch 中的算子在不同数据规模下的正确性测试结果, 其中√表示算子通过了正确性测试, 实验结果为 5 次随机实验后综合得出.

表 3 MTTorch 算子正确性测试

测试算子	2^{18}	2^{24}	2^{28}
add/sub	√	√	√
mul/div	√	√	√
GELU	√	√	√
ReLU	√	√	√
GELU_back	√	√	√
ReLU_back	√	√	√
softmax	√	√	√
log_softmax	√	√	√
softmax_back	√	√	√
log_softmax_back	√	√	√
layer_norm	√	√	√
mm	√	√	√
sum	√	√	√
nll_loss_forward*	√	√	√

由结果可得, 无论是核心算子还是虚拟算子, 计算误差都小于 1E–06.

4.2 RQ2: MTTorch 中虚拟算子的性能

在本节中, 对比了虚拟算子和 CPU 同构环境下 PyTorch 算子的性能. 数据规模为 2^{22} 个单精度浮点数.

从图 9 可以看出, 虚拟算子性能和 CPU 实现的算子不相上下, 这表明通过修改设备属性改变算子运算后端的方法几乎未引入额外开销, 达到了我们的设计目标. 同时, 在大多数情况下, 虚拟算子的性能还好于 CPU 算子. 这

© 中国科学院软件研究所 http://www.jos.org.cn

可能是因为 CPU 端内存同时被操作系统使用, 使得缓存需要频繁进行替换, 造成性能的损失. 而片上的每一个 DSP 簇都是一个裸机系统, 其内存区域并没有操作系统, 因此更容易缓存命中.

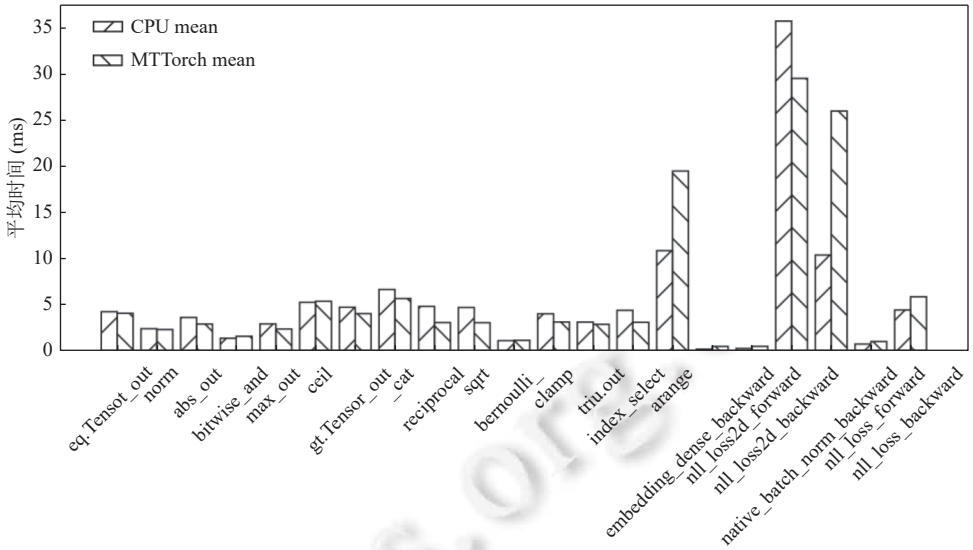

图 9 虚拟算子与 CPU 同构环境算子性能对比

4.3 RQ3: MTTorch 中核心算子的性能提升

4.3.1 DSP 算子与 CPU 算子性能对比

本文使用 PyTorch 中的 profile 工具统计算子的执行时间. Profile 是 PyTorch 框架中用于对模型进行性能分析和调试的工具之一, 它提供了一个简单的方式来收集 PyTorch 操作的计算时间、内存使用、执行次数等信息. 为了能够直观地评估 MTTorch 中基于 Hthreads 实现的核心算子与 MT-3000 CPU 算子性能上的差异, 本文设计了公式 (9) 中的性能评价指标:

$$P_{\text{source}}=\begin{cases}-\dfrac{T_{\text{Hthreads}}}{T_{\text{CPU}}}, & T_{\text{Hthreads}}\geqslant T_{\text{CPU}}\\ \dfrac{T_{\text{CPU}}}{T_{\text{Hthreads}}}, & T_{\text{Hthreads}}<T_{\text{CPU}}\end{cases} \tag{9}$$

其中, T_{Hthreads} 代表 MTTorch 中基于 Hthreads 实现的核心算子运行时间, T_{CPU} 为 MT-3000 CPU 上算子的运行时间. 在相同情况下, $P_{\text{source}}>1$ 表示 Hthreads 核心算子性能优于 CPU, 且越大代表 Hthreads 核心算子的性能提升的倍数越大; $P_{\text{source}}<-1$ 表示 MT-3000 CPU 的算子较好.

本文分别统计了在不同数据规模下 (2^{18}–2^{28}), MT-3000 CPU 和 Hthreads 核心算子的运行时间, 结果以 10 个不同的 MT-3000 节点运行取平均为准. 图 10 展示了 CPU 算子和 Hthreads 核心算子性能对比.

从图 10 中可以看出, 在数据量小的情况下, CPU 算子略微好于 MTTorch, 但并没有倍数的优势. 而随着数据量的逐渐增大, Hthreads 核心算子的性能优势逐渐明显. 大部分算子都取得了 2–8 倍的加速效果. 这是由于 DSP 线程组的启动有 1 ms 左右的固有开销, 当数据量少的时候, 这个固有开销可能会将 Hthreads 带来的性能提升抵消掉. 因此, 在实现算子时应该根据传入 Tensor 的数据量大小动态地选择使用 DSP 加速还是使用虚拟算子.

此外, 我们发现, 使用 Hthreads 进行实现的算子在稳定性方面要远好于 CPU 同构环境下的算子. 在不同节点上, 使用 Hthreads 加速后的标准差通常都维持在 [0, 1] 之间, 而 CPU 同构环境下的算子, 其标准差却会随着数据规模的增大而不断上升. 考虑算子的稳定性对于大规模分布式训练是必要的, 因为在多节点训练的场景下, 由于短板效应, 模型的整体运行时间往往是由集群中性能最差的节点决定的, 更稳定的节点更有助于系统发挥其性能的上限.

© 中国科学院软件研究所 http://www.jos.org.cn

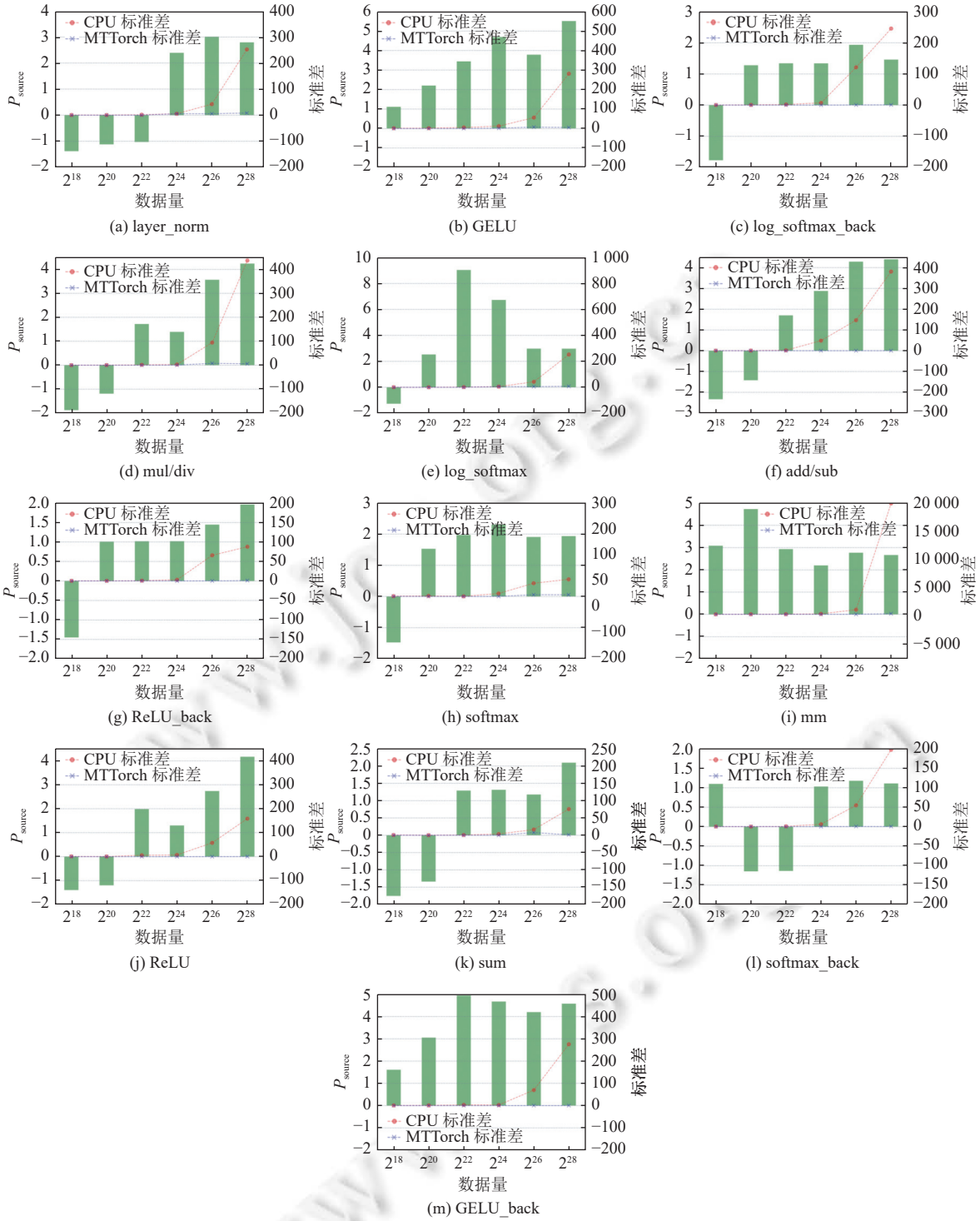

图 10　15 个核函数在不同输入数据规模下使用 CPU 和 DSP 运算时间和标准差对比

4.3.2　DSP 算子与 GPU 算子性能效率对比

进一步, 本文设计实验对比了 MTTorch 和 PyTorch 中原生的 CUDA 算子的性能效率. MT-3000 每个 DSP 核

© 中国科学院软件研究所 http://www.jos.org.cn

心运行在 1.8 GHz 下, 双精度浮点峰值性能为 518.4 GFlops. 相比之下, NVIDIA Tesla V100S 的双精度峰值性能则高达 8.177 TFlops. 为了公平对比, 本文利用公式 10, 通过计算性能效率的比值将不同硬件峰值性能进行归一化.

$$P_{\text{norm}}=\frac{\dfrac{N/T_{\text{DSP}}}{F_{\text{DSP}}}}{\dfrac{N/T_{\text{GPU}}}{F_{\text{GPU}}}}=\frac{T_{\text{GPU}}\times F_{\text{GPU}}}{T_{\text{DSP}}\times F_{\text{DSP}}} \tag{10}$$

其中, N 代表执行该算子需要的浮点数计算次数, T 代表执行该算子的时间, F 代表对应硬件的浮点数峰值性能. P_{norm} 越大, 表明 MTTorch 中通过 DSP 优化的算子性能效率越好. 归一化后 MTTorch 和 PyTorch CUDA 算子性能效率比如图 11 所示.

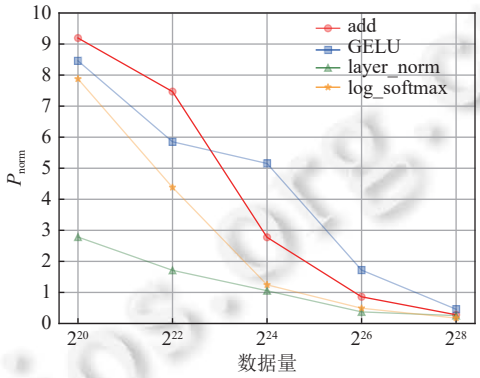

图 11 CUDA 算子和 MTTorch 算子性能效率对比

从图中可得, 在数据规模较小时, 例如小于 2^{24}, 在归一化硬件差距后, 本文实现的算子相较于 NVIDIA Tesla V100S GPU 更加高效, 这是因为在数据规模较小时 V100S 并不能完全利用所有 5 120 个计算核心, 而 DSP 的 24 个核可以充分利用. 然而 NVIDIA Tesla V100S 显存 (graphical double data rate, GDDR) 带宽为 1 134 GB/s, 相比之下 DSP 上 DMA 在 DDR 和 AM 间的实测带宽仅为 42.62 GB/s. 因此在数据规模较大时 (大于 2^{26}), 与 V100S 相比, DSP 的数据访存会成为影响 MTTorch 核心算子性能的瓶颈.

4.3.3 消融实验

为了证明乒乓算法的有效性, 本文通过对 log_softmax、layer_norm、softmax_back、softmax、log_softmax_back 进行消融实验, 测试了乒乓算法在不同输入数据规模下的优化效果. 实验结果如后文图 12 所示. 可以看出, 当数据规模超过 2^{20} 时, 采用乒乓算法相较于仅使用向量化优化的算子能够带来更大的性能提升. 同时, 在数据规模较大的场景中, 使用乒乓算法依然能够保持显著的性能优势.

4.4 RQ4: 使用 MTTorch 进行模型训练时的性能和加速比

本节验证了 MTTorch 在训练两种不同结构的语言模型时相比于 CPU 同构环境下 PyTorch 的性能和加速比提升. 这两种语言模型, 一个是 PyTorch 官方 Example 库中的 Transformer 模型; 另一个是复现的 GPT-3[1]结构的 4 层 GPT 模型, 嵌入向量长度为 1 024.

图 13 展示了在单个 MT-3000 片上, 训练两个模型的性能对比. 可以看到, 相比 CPU 同构环境下的 PyTorch, MTTorch 可以带来约 41% 的性能提升. 模型的性能并没有成倍增加, 这是由于本文重点关注核心算子的性能提升, 为了规避模型并行等相关技术环节, 因此模型参数量以及每一层的数据规模并不大, 不能很好地发挥 DSP 在大规模密集计算下的优势. 在训练大模型时, 往往采用模型并行的方式, 每个节点只计算大模型中的数层. 更大的参数矩阵将更有利于 MTTorch 的加速.

本文的实验平台是基于 MT-3000 的高性能计算集群, 能否高效利用集群中的海量节点进行分布式训练是成功训练大语言模型的关键. 为了测试 MTTorch 的可扩展性, 本文使用 horovod[22]在不同规模的节点个数 (从 2 个节点到 970 个节点) 下进行数据并行训练. 训练的加速比如图 14 所示.

© 中国科学院软件研究所 http://www.jos.org.cn

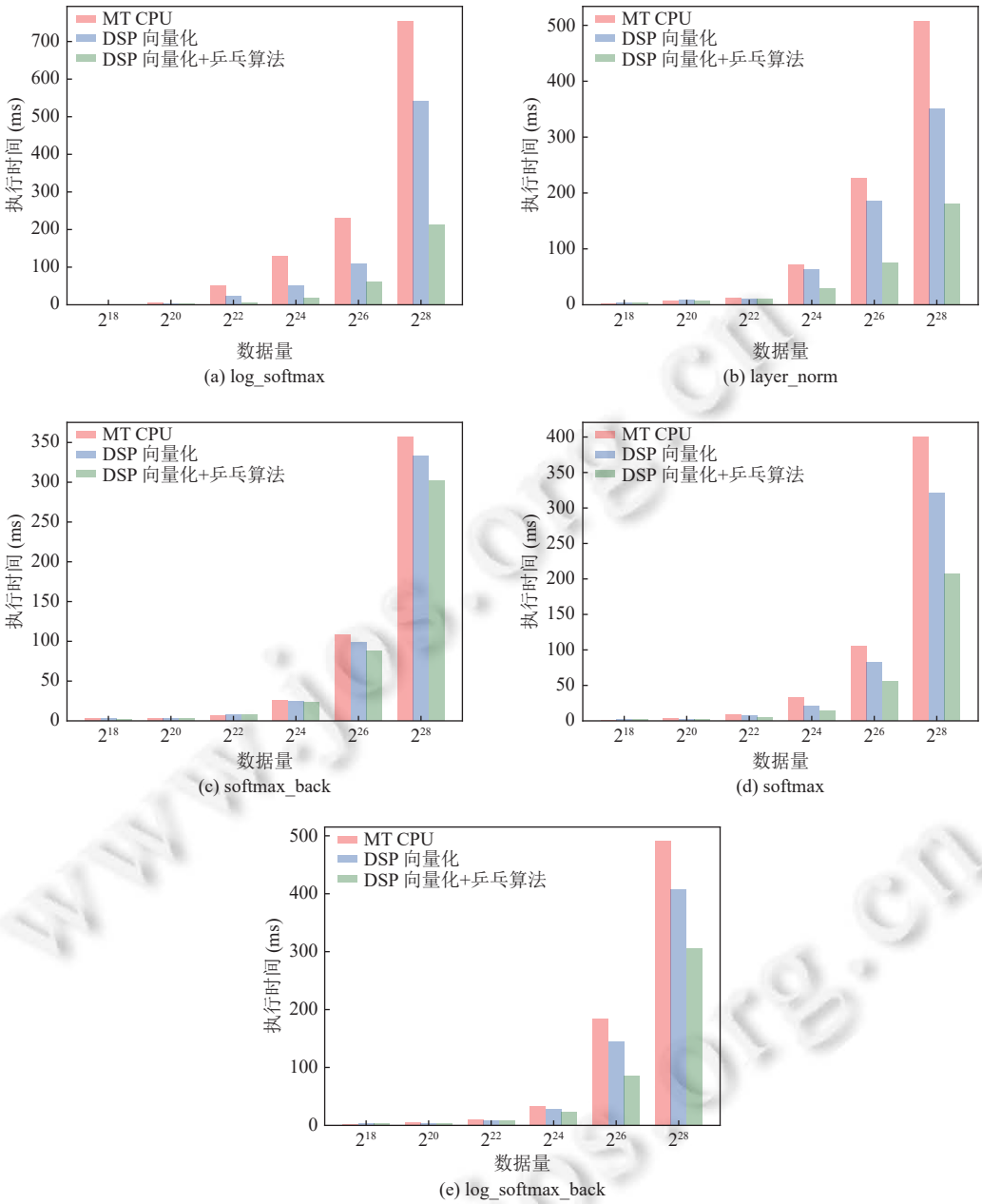

图 12　算子优化消融实验

可以看出, 两个模型的加速比都要显著优于 CPU 同构环境下的 PyTorch, 且随着节点的增加加速比逐渐呈线性增加, 说明本文实现的方法可以为基于 MT-3000 搭建的高性能计算集群带来更强的可扩展性, 使得 DSP+CPU 异构集群在人工智能领域的潜力可以被进一步激发.

5 总　结

本文结合训练大语言模型的实际需求, 介绍了面向天河新一代超算系统加速芯片 MT-3000 的 PyTorch 扩展库——MTTorch 的设计和实现. MTTorch 运用单例模式对 DSP 设备和线程组进行高效管理. 算子层面, 本文利用

© 中国科学院软件研究所 http://www.jos.org.cn

MT-3000 DSP 加速区内存 CPU 可读写的特性, 设计了虚拟算子技术. 这保证了计算图中算子支持完备. 同时, 针对训练 Transformer 模型过程中的核心算子, 从算子实现向量化、DSP 多核规约算法、乒乓算法等角度介绍了深度学习算子在 MT-3000 上并行加速的思路和实现, 并取得了 2–8 倍的性能提升. 本文实现的 MTTorch 作为一个原生 PyTorch 的扩展库, 可以实现不同版本 PyTorch 的即插即用, 新增算子无需修改 PyTorch 源码, 具有很高的可编程性.

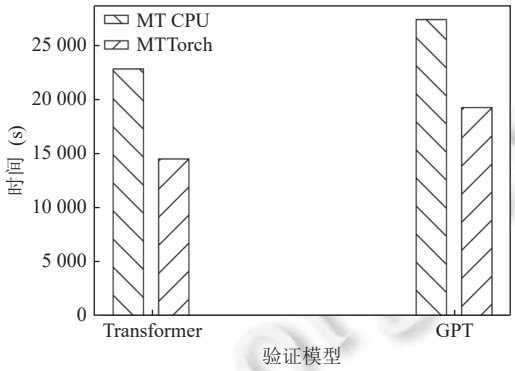

图 13 单节点模型运行时间对比

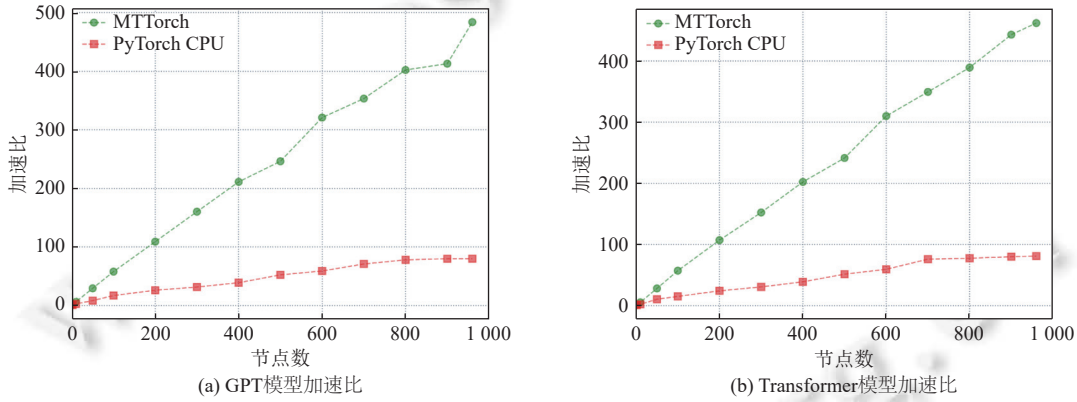

图 14 MTTorch 可扩展性测试

作者希望本文所分享的经验和方法能够支持未来基于 DSP 的高性能计算集群的中文大语言模型训练任务. 在今后的工作中, 作者将继续探索更多深度学习算子在 MT-3000 片上的优化方法, 实现更高性能的通用矩阵乘法和卷积操作. 同时, 针对于矩阵转置等离散内存操作探索更加高效的解决方案.

致谢 感谢国家超级计算天津中心以及国防科技大学计算机学院给予作者实验环境.

References:

[1] Brown TB, Mann B, Ryder N, Subbiah M, Kaplan J, Dhariwal P, Neelakantan A, Shyam P, Sastry G, Askell A, Agarwal S, Herbert-Voss A, Krueger G, Henighan T, Child R, Ramesh A, Ziegler DM, Wu J, Winter C, Hesse C, Chen M, Sigler E, Litwin M, Gray S, Chess B, Clark J, Berner C, McCandlish S, Radford A, Sutskever I, Amodei D. Language models are few-shot learners. In: Proc. of the 34th Int'l Conf. on Neural Information Processing Systems. Vancouver: Curran Associates Inc., 2020. 159.

[2] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I. Attention is all you need. In: Proc. of the 31st Int'l Conf. on Neural Information Processing Systems. Long Beach: Curran Associates Inc., 2017. 6000–6010.

[3] Ouyang L, Wu J, Jiang X, Almeida D, Wainwright CL, Mishkin P, Zhang C, Agarwal S, Slama K, Ray A, Schulman J, Hilton J, Fraser

© 中国科学院软件研究所 http://www.jos.org.cn

Kelton, Miller L, Simens M, Askell A, Welinder P, Christiano P, Leike J, Lowe R. Training language models to follow instructions with human feedback. In: Proc. of the 36th Int'l Conf. on Neural Information Processing Systems. New Orleans: Curran Associates Inc., 2022.

[4] Radford A, Narasimhan K, Salimans T, Sutskever I. Improving language understanding by generative pre-training. 2018. https://www.semanticscholar.org/paper/Improving-Language-Understanding-by-Generative-Radford-Narasimhan/cd18800a0fe0b668a1cc19f2ec95b5003d0a5035

[5] Tian Z, Chen YF. Performance optimization of molecular dynamics simulation on Sunway TaihuLight system. Ruan Jian Xue Bao/Journal of Software, 2021, 32(9): 2945–2962 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5978.htm [doi: 10.13328/j.cnki.jos.005978]

[6] Hong WJ, Li KL, Quan Z, Yang WD, Li KQ, Hao ZY, Xie XH. PETSc's heterogeneous parallel algorithm design and performance optimization on the Sunway TaihuLight system. Chinese Journal of Computers, 2017, 40(9): 2057–2069 (in Chinese with English abstract). [doi: 10.11897/SP.J.1016.2017.02057]

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

[8] Lu K, Wang YH, Guo Y, Huang C, Liu S, Wang RB, Fang JB, Tang T, Chen ZY, Liu BW, Liu Z, Lei YW, Sun HY. MT-3000: A heterogeneous multi-zone processor for HPC. CCF Trans. on High Performance Computing, 2022, 4(2): 150–164. [doi: 10.1007/s42514-022-00095-y]

[9] Wang ZL. Design and implementation of high-performance DMA transmission modes for scientific computation on GPDSP [MS. Thesis]. Changsha: National University of Defense Technology, 2015 (in Chinese with English abstract).

[10] Gong CY, Liu J, Bao WM, Pan DM, Gan XB, Li SG, Chen XG, Xiao TJ, Yang B, Wang RB. Review on ecological construction of domestic high-performance parallel application software in post Moore Era. Journal of System Simulation, 2022, 34(10): 2107–2118 (in Chinese with English abstract). [doi: 10.16182/j.issn1004731x.joss.21-1365]

[11] Fang J, Varbanescu AL, Sips H. A comprehensive performance comparison of CUDA and OpenCL. In: Proc. of the 2011 Int'l Conf. on Parallel Processing. IEEE, 2011. 216–225. [doi: 10.1109/ICPP.2011.45]

[12] Shen J, Fang JB, Sips H, Varbanescu AL. Performance gaps between OpenMP and OpenCL for multi-core CPUs. In: Proc. of the 41st Int'l Conf. on Parallel Processing Workshops. Pittsburgh: IEEE, 2012. 116–125. [doi: 10.1109/ICPPW.2012.18]

[13] Li YY, Xue W, Chen DX, Wang XL, Xu P, Zhang WS, Yang GW. Performance optimization for sparse matrix-vector multiplication on Sunway architecture. Chinese Journal of Computers, 2020, 43(6): 1010–1024 (in Chinese with English abstract). [doi: 10.11897/SP.J.1016.2020.01010]

[14] Yin SF, Wang QL, Hao RC, Zhou TY, Mei SZ, Liu J. Optimizing irregular-shaped matrix-matrix multiplication on multi-core DSPs. In: Proc. of the 2022 IEEE Int'l Conf. on Cluster Computing (CLUSTER). Heidelberg: IEEE, 2022. 451–461. [doi: 10.1109/CLUSTER51413.2022.00055]

[15] Pei XD, Wang QL, Liao LY, Li RC, Mei SZ, Liu J, Pang ZB. Optimizing parallel matrix transpose algorithm on multi-core digital signal processors. Journal of National University of Defense Technology, 2023, 45(1): 57–66 (in Chinese with English abstract). [doi: 10.11887/j.cn.202301006]

[16] Wang QL, Li DS, Mei SZ, Lai ZQ, Dou Y. Optimizing Winograd-based fast convolution algorithm on Phytium multi-core CPUS. Journal of Computer Research and Development, 2020, 57(6): 1140–1151 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.2020.20200107]

[17] Gu JL, Liu YB, Gao Y, Zhu MH. OpenCL caffe: Accelerating and enabling a cross platform machine learning framework. In: Proc. of the 4th Int'l Workshop on OpenCL. Vienna: ACM, 2016. 8. [doi: 10.1145/2909437.2909443]

[18] Chen J, Zhong L. An efficient parallel CNN inference framework for multi-zone processor. In: Proc. of the 24th Int'l Conf. on High Performance Computing & Communications; Proc. of the 8th Int'l Conf. on Data Science & Systems; Proc. of the 20th Int'l Conf. on Smart City; Proc. of the 8th Int'l Conf. on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys). IEEE, 2022. 1350–1357. [doi: 10.1109/HPCC-DSS-SmartCity-DependSys57074.2022.00210]

[19] Chen R, Sun YF, Cheng DG, Guo Q, Chen YQ, Shi CQ, Sui YC, Zhang YZ, Zhang YZ. Implementation and optimization of OpenCL kernels in TensorFlow. Chinese Journal of Computers, 2022, 45(11): 2456–2474 (in Chinese with English abstract). [doi: 10.11897/SP.J.1016.2022.02456]

[20] Fang JB, Zhang P, Huang C, Tang T, Lu K, Wang RB, Wang Z. Programming bare-metal accelerators with heterogeneous threading models: A case study of Matrix-3000. Frontiers of Information Technology & Electronic Engineering, 2023, 24(4): 509–520. [doi: 10.1631/FITEE.2200359]

[21] Zhang P, Fang JB, Yang CQ, Tang T, Huang C, Wang Z. MOCL: An efficient OpenCL implementation for the matrix-2000 architecture.

© 中国科学院软件研究所 http://www.jos.org.cn

In: Proc. of the 15th ACM Int'l Conf. on Computing Frontiers. Ischia: ACM, 2018. 26–35. [doi: 10.1145/3203217.3203244]

[22] Sergeev A, Del Balso M. Horovod: Fast and easy distributed deep learning in TensorFlow. arXiv:1802.05799, 2018.

附中文参考文献:

[5] 田卓, 陈一峯. 神威太湖之光上分子动力学模拟的性能优化. 软件学报, 2021, 32(9): 2945–2962. http://www.jos.org.cn/1000-9825/5978.htm [doi: 10.13328/j.cnki.jos.005978]

[6] 洪文杰, 李肯立, 全哲, 阳王东, 李克勤, 郝子宇, 谢向辉. 面向神威·太湖之光的 PETSc 可扩展异构并行算法及其性能优化. 计算机学报, 2017, 40(9): 2057–2069. [doi: 10.11897/SP.J.1016.2017.02057]

[9] 王占立. 面向 GPDSP 科学计算的高性能 DMA 传输方式的设计与实现 [硕士学位论文]. 长沙: 国防科技大学, 2015.

[10] 龚春叶, 刘杰, 包为民, 潘冬梅, 甘新标, 李胜国, 陈旭光, 肖调杰, 杨博, 王睿伯. 后摩尔时代国产高性能并行应用软件生态建设综述. 系统仿真学报, 2022, 34(10): 2107–2118. [doi: 10.16182/j.issn1004731x.joss.21-1365]

[13] 李亿渊, 薛巍, 陈德训, 王欣亮, 许平, 张武生, 杨广文. 稀疏矩阵向量乘法在申威众核架构上的性能优化. 计算机学报, 2020, 43(6): 1010–1024. [doi: 10.11897/SP.J.1016.2020.01010]

[15] 裴向东, 王庆林, 廖林玉, 李荣春, 梅松竹, 刘杰, 庞征斌. 多核数字信号处理器并行矩阵转置算法优化. 国防科技大学学报, 2023, 45(1): 57–66. [doi: 10.11887/j.cn.202301006]

[16] 王庆林, 李东升, 梅松竹, 赖志权, 窦勇. 面向飞腾多核处理器的 Winograd 快速卷积算法优化. 计算机研究与发展, 2020, 57(6): 1140–1151. [doi: 10.7544/issn1000-1239.2020.20200107]

[19] 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. TensorFlow 中 OpenCL 核函数的实现与优化. 计算机学报, 2022, 45(11): 2456–2474. [doi: 10.11897/SP.J.1016.2022.02456]

王昊天(2000－), 男, 博士生, 主要研究领域为异构计算, 高性能计算, 自然语言处理.

石昌青(1997－), 男, 博士生, CCF 学生会员, 主要研究领域为深度学习, 异构计算, 高性能计算.

孙羽菲(1976－), 女, 博士, 特聘研究员, 主要研究领域为深度学习, 异构计算, 人工智能.

方建滨(1984－), 男, 博士, 助理研究员, CCF 杰出会员, 主要研究领域为多核并行编程, 并行编译器, 性能建模, 可扩展算法.

隋轶丞(1998－), 男, 博士生, 主要研究领域为高性能计算, 深度学习算法.

张玉志(1964－), 男, 博士, 教授, 博士生导师, CCF 专业会员, 主要研究领域为人工智能, 模式识别, 自然语言处理.

王嘉豪(2000－), 男, 硕士生, 主要研究领域为深度学习, 高性能计算.

© 中国科学院软件研究所 http://www.jos.org.cn