基于昇腾NPU的底层算子编译器中归约类算子设计与优化

Part1：

中文名称	基于昇腾NPU的归约类算子设计与优化
英文名称	Design and Optimization of Operators Based on Ascend NPU

Part2：课题简介

1.背景描述与研究动机

1.1 深度学习编译器的重要地位和工作流程

深度学习编译器在 AI 模型开发和部署中不可或缺，显著提升了模型在不同硬件上的执行效率。受 LLVM 架构[1]启发，现代编译器采用多层中间表示（MLIR），在不同层次优化从高层张量操作到底层硬件指令的映射，从而简化了硬件适配流程。这种多层表示使编译器能够在不同的抽象层次上执行优化，适配从高层张量操作到底层硬件指令[2]。

深度学习编译器的典型工作流程包括前端解析、IR 优化和硬件指令生成，以分阶段方式提升整体性能并减少手动优化[3]。

1.2 扩展到深度学习加速器的原因及意义

大模型中产生的计算负荷（如归约操作）使得 CPU 等通用芯片难以高效处理，因此工业界和学术界纷纷设计专用加速器（如 NPU），以提供更高效的计算性能。

对于不同的加速器设计方向，业界也有不同的硬件实现。对于偏定制化的硬件架构，面向深度学习计算任务，业界提出了神经网络加速器（NPU）。昇腾 NPU[4] 使用 VECTOR 和 CUBE 运算单元来加速常用的深度学习运算，满足模型在训练和推理中的高效需求

1.3 归约类算子在深度学习推理和训练中的作用

在编译过程中，算子是核心计算操作的基础，如矩阵乘法和归约操作。编译器通过优化算子的执行顺序和内存分配，实现并行化和资源高效利用。

归约算子则在深度学习推理和训练中尤为关键，用于将张量数据压缩为单一值或小集合，支撑损失计算、梯度求解等操作。这类算子的优化不仅加速了模型计算，还能有效提升硬件资源利用率，尤其适合大规模并行深度学习任务。

2.问题描述与当前挑战

2.1 基于 NPU 的深度学习编译器带来的问题和挑战

- 随着大模型和各类 NPU 芯片的快速发展，基于人工优化的算子开发模式对开发团队带来了巨大负担，因此设计能自动编译高层算子到硬件的编译器逐渐成为趋势。算子编译器通过模版、搜索算法和优化求解等方式实现循环变换、指令映射和内存分配等优化。然而，现有编译器在兼容高层应用和 NPU 硬件指令方面存在问题。NPU 由于支持多种指令和多级存储，数据流处理复杂，增加了循环、内存和并行优化的难度，导致算子适配差、利用率低等问题。

- **访存带宽低**：NPU 依赖片上 DMA 实现数据搬移，访存能力弱，难以支持 Element-Wise 操作。
- **缓存空间小**：片内缓存有限，难以保持统一抽象，增加了内存分配和延迟优化难度。
- **同步开销大**：细粒度同步对性能影响大，需精确插入同步指令。
- **并行优化难度高**：缺乏自动流水并行优化，导致多核算子并行不匹配。

2.2 算子自动生成

目前多个开源深度学习框架/编译器已经支持自动生成算子代码。许多编译器的设计灵感来自 Halide[5]，包括 TVM[6]、nvFuser[7]、NNC[8]。这些设计有独立的语义语言和调度语言，可以在不改变程序语义的情况下尝试不同的调度。MLIR[9]生态系统中出现了很多更新的编译器，包括 IREE[10]。Triton[11]也使用 MLIR 作为其内部表示方法。使用 Triton能够比手写库 [^12,13]更快地生成内核，而且输入代码也很简单。很少有编译器能够始终如一地做到这一点，许多只是直接调用这些库而不试图在生成复杂内核方面改进。

本研究拟针对 NPU 中间表达同上层和底层硬件接口不兼容的问题，为 NPU 中差异性和多样性的张量表示和张量计算核设计中间表达和调度抽象，设计适配上层应用的简单高效的编程接口，覆盖不同的硬件后端，自动生成大模型的关键算子和常规算子，解决相同算子在不同硬件上的重复开发问题，实现对新硬件的快速支持。需要解决的关键问题包括：

- 算子种类繁多，特点各异
- 理解并丰富现有算子中间表达
- 实现算子代码模版

2.3 归约类算子技术瓶颈

在深度学习和科学计算中，归约类算子（如求和、乘积、最大值等）是基础计算工具，广泛用于矩阵和卷积操作等任务。Triton 编译器通过 tile-based 并行方法实现了这些算子的 GPU 加速，大幅提升训练和推理效率。然而，GPU 的 tile-based 技术在 NPU（神经网络处理单元）上效果有限，因 NPU 更偏向特定张量运算和内存管理模式。

本课题将以 Triton 中的 `sum` 算子为例，研究其在昇腾 NPU 上的优化实现，以期实现类似 GPU 的高效归约性能，并提出一种跨架构优化方法，提升深度学习编译器在多硬件上的通用性。3.研究目标与设计思路

3.1 归约类算子在 昇腾 NPU 上的实现

将深度学习框架中的算子编译到可以执行的特定硬件上的指令可以使用多种不同的深度学习编译器框架，如 TVM，MLIR等。基于 MLIR 的高度灵活性和多层次抽象特性，这对于适配 NPU 特定的硬件架构尤其关键。

MLIR 支持多层中间表示（IR），可在更高层次表达模型计算逻辑，同时在底层灵活映射到 NPU 的特定硬件指令，使得编译器能够针对 NPU 的硬件特性（如数据流和存储架构）进行优化。MLIR 的方言机制允许开发者定义特定领域的操作符，使得我们可以根据 NPU 的要求自定义归约算子并进行优化。

相比之下，TVM 等框架更多地面向 GPU 和 CPU 等通用硬件的优化需求，且其调度方式相对固定，难以针对 NPU 的特定架构特性进行深层次定制。MLIR 提供了更灵活的硬件抽象和优化框架，可以在各个层级上进行精细化调整，这种灵活性对于适配和充分利用昇腾 NPU 的计算资源至关重要。此外，MLIR 提供了 LLVM 支持的优化路径，使得编译器能够以 LLVM 的成熟优化技术为基础，在不同硬件间实现高效的跨架构支持。

3.2 归约类算子实现的实际思路

在昇腾 NPU 上实现归约类算子的过程需要将 `tl.sum` 算子逐步翻译为 NPU 可执行的 `npubinary` 指令集。为此，整个实现流程分为四个关键组件：Triton、TritonShared、MLIR 和 LLVM 项目、以及 CCEC 编译器。以下是实现的具体思路：

1. **Triton 阶段**：首先通过 Triton 编译器来解析并生成初始的计算图。`triton.sum` 算子在此阶段被转化为高层次的张量运算表达形式。Triton 使用 tile-based 并行方法处理归约操作，以便分解大规模数据处理任务，并初步适配 NPU 计算的基本架构需求。
2. **TritonShared 阶段**：在这个阶段，`triton.sum` 的计算图被转化为通用的中间表示并与 NPU 硬件特性进一步融合。TritonShared 作为中间层，将高层的张量操作与底层的硬件架构细节桥接，确保算子表达符合 NPU 对数据流、内存布局和并行执行的要求，同时为后续的 MLIR 转化奠定基础。
3. **MLIR 和 LLVM 阶段**：接下来，TritonShared 的中间表示通过 MLIR（多层次中间表示）进一步降低到 LLVM 表示。MLIR 支持灵活的方言机制，使得开发者能够根据 NPU 特性定制化处理算子。MLIR 在这一阶段优化数据流操作，确保符合 NPU 的计算需求。随后，LLVM 项目将 MLIR 低层表示转换为 NPU 硬件能够识别的指令格式，充分利用 LLVM 的优化能力。
4. **CCEC 编译器阶段**：最后，经过 LLVM 生成的低层代码在 CCEC 编译器中被编译为具体的 `npubinary` 指令，适配昇腾 NPU 的执行单元。CCEC 编译器负责将这些指令准确映射到 NPU 的硬件资源上，并最终生成高效的、可执行的归约算子代码。

这一流程充分利用了 Triton、MLIR 和 CCEC 编译器在不同层级的优化能力，确保 `triton.sum` 归约算子在昇腾 NPU 上实现高效执行。

4.方法与实现细节

5.实验设计与验证

6.总结

---

1. The LLVM Compiler Infrastructure https://github.com/llvm/llvm-project. ↩

2. Katel, N., Khandelwal, V., & Bondhugula, U. (2021). High Performance GPU Code Generation for Matrix-Matrix Multiplication using MLIR: Some Early Results. *arXiv preprint arXiv:2108.13191*. Retrieved from https://arxiv.org/abs/2108.13191 ↩

3. Li, M., Liu, Y., Liu, X., Sun, Q., You, X., Yang, H., Luan, Z., Gan, L., Yang, G., & Qian, D. (2021). The Deep Learning Compiler: A Comprehensive Survey. *IEEE Transactions on Parallel and Distributed Systems, 32*(3), 708–727. https://doi.org/10.1109/TPDS.2020.3030548 ↩

4. H. Liao *et al*., "Ascend: a Scalable and Unified Architecture for Ubiquitous Deep Neural Network Computing : Industry Track Paper," *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Seoul, Korea (South), 2021, pp. 789-801, doi: 10.1109/HPCA51647.2021.00071. ↩

5. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., & Amarasinghe, S. (2013). Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (pp. 519–530). Association for Computing Machinery. https://doi.org/10.1145/2491956.2462176 ↩

6. Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., Guestrin, C., & Krishnamurthy, A. (2018). TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. *arXiv preprint arXiv:1802.04799*. Retrieved from https://arxiv.org/abs/1802.04799 ↩

7. nvFuser, a deep learning compiler for PyTorch. https://pytorch.org/blog/introducing-nvfuser-a-deep-learning-compiler-for-pytorch ↩

8. NNC walkthrough: how PyTorch ops get fused. https://dev-discuss.pytorch.org/t/nnc-walkthrough-how-pytorch-ops-get-fused/125 ↩

9. C. Lattner *et al*., "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation," *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Seoul, Korea (South), 2021, pp. 2-14, doi: 10.1109/CGO51591.2021.9370308. ↩

10. IREE: Intermediate Representation Execution Environment. https://github.com/iree-org/iree ↩

11. Tillet, P., Kung, H. T., & Cox, D. (2019). Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (pp. 10–19). Association for Computing Machinery. https://doi.org/10.1145/3315508.3329973 ↩