

3.2 Pdist 算子优化

3.2.1 赛题介绍

Pdist (Pairwise Distance) 算子是深度学习与高性能计算领域中的基础算子，广泛应用于聚类分析（如 K-Means）、计算机视觉中的行人重识别（ReID）、自然语言处理中的语义相似度计算等场景。其核心功能是计算高维空间中样本集合的『两两』距离。

数学定义：给定输入样本矩阵 $X \in \mathbb{R}^{N \times M}$ ，其中 N 表示样本数量， M 表示特征维度。Pdist 算子计算任意两个样本向量 x_i 与 x_j ($0 \leq i < j < N$) 之间的 p -范数距离。输出结果通常被展平为一个长度为 $L = \frac{N(N-1)}{2}$ 的一维向量 Y 。

对于给定的 p 值，任意一对样本 (i, j) 之间的距离 y_k 计算公式如下：

$$y_k = \|x_i - x_j\|_p = \left(\sum_{m=0}^{M-1} |x_{i,m} - x_{j,m}|^p \right)^{1/p} \quad (3.1)$$

其中 k 是压缩存储后的线性索引。本赛题要求我们在 **Ascend 910B4 NPU** 上，基于 **Ascend C** 编程范式实现高性能的 Pdist 算子。要求精度与 Pytorch 原生 CPU Pdist 算子对齐，并尽可能获得更高的算子性能。

3.2.2 开发环境

- **算子运行环境：**Ascend 910B4 NPU 单卡。
- **CANN 工具包：**8.3RC1 版本，基于 Conda 管理开发环境。
- **性能测试工具：**综合使用 `msprof`、`msprof op`、`msprof op simulator` 等命令进行算子性能采样。
- **项目构建与管理：**基于 AscendC 工程化算子开发构建项目架构，基于单算子 API 调用进行算子运行和性能测试。

3.2.3 架构设计

我们目前针对不同的 p 值进行了特化设计，包括 $p = 1$ 距离（即曼哈顿距离 Manhattan Distance）、 $p = 2$ 距离（即欧几里得距离 Euclidean Distance）、 $p = \inf$ 距离（即切比雪夫距离 Chebyshev Distance）以及其他通用距离（General Distance）。

我们总共设计了两种 Tiling 策略从而提高算子的鲁棒性，使其能够应对各种规模的数据（比如 $(2, 1)$ 、 $(100, 400)$ 、 $(2024, 3000)$ 、 $(5000, 100000)$ 等等）并最大化利用运行平台的硬件资源（如 Unified Buffer 等）。大体设计是：如果单次计算中 UB 大小不足以存储两个完整的 M 维向量，我们将退化到 Huge 策略，否则大部分情况下采取 Normal 策略。

同时我们的 Tiling 策略充分根据运行平台信息计算实际计算策略，保证了在非 910 NPU 上的可移植性。

在实际开发中，我们发现目前的 `torch_npu` Pdist 算子在 FP16 的精度支持较差（比如略微大点的数据，即使最后的结果在 FP16 支持精度以内，由于计算过程中数据溢出而导致结果为 `inf` 的情况），我们特别针对有需要的 FP16 Pdist 计算进行了数据转换，尽最大可能保证了算子实现的正确性和鲁棒性。

我们的算子开发基于 C++ 模板 (Template) 范式，尽可能减少了冗余代码，并减少算子内部执行时的不必要的分支判断，同时提高了后续算子开发的可扩展性。

算子正确性基于随机数据生成，判断标准是绝对误差或者相对误差满足题目要求的精度标准。

3.2.4 优化手段

- 任务均分**: 动态适应 NPU 机器的矢量单元数量，并根据输出结果总量依次均分任务，实测中能够达到较为一致的负载均衡。
- 减少访存次数**: 由于 Pdist 结果计算的二重循环特性，第一维循环参与计算的向量在一整轮第二层循环中都是保持不变的，我们通过常驻 UB 减少了与 GM 间的搬运次数。（此条目针对中小规模数据的 Normal 算子，对于处理大数据规模的 Huge 算子，UB 无法常驻任意向量，需要采用其他优化手段，详见最后一条）
- 构建流水线**: 采用双缓冲 (Double Buffer) 技术，实现数据搬运和计算的并行化。
- 标量优化**: 减少不必要的标量操作（比如常驻 UB 张量的反复入出队），尽可能消除繁重的标量同步逻辑，提高矢量单元利用率。
- 批量处理**: 根据 UB 大小一次性批量搬入多个第二层循环的向量，包括一次性批量从 UB 向 GM 写回暂存的 Pdist 计算结果等，增加内存带宽的利用率并减少矢量单元流水线的中断和同步，增加吞吐量。
- 重写规约逻辑**: 实际开发和性能分析中，我们发现 AscendC 的通用规约 (Reduce) API 内部存在大量标量同步原语，十分影响矢量单元的计算效率和双缓冲的有效性，在我们的应用场景下存在极大的性能优化空间。于是我们基于更底层的块规约 (BlockReduce) 和全局规约 (WholeReduce) 重写了规约逻辑，实现了明显的性能提升 (30ms 至 18ms) 并极大地提高了矢量单元的利用率 ((2024, 3000) 大小的数据矢量单元利用率接近 95%)。



(a) AscendC 规约 API

(b) 我们实现的规约逻辑

图 3.6: 不同 Reduce 实现的性能分析流水图对比

- 最小化精度转换增加的计算开销**: 由于理论上 FP16 的计算性能是 FP32 的两倍（虽然在实际测试中 AscendC API 似乎并没有很明显的展现相关特性），所以即使由于要保证精度我们需要将 FP16 数据转换成 FP32 进行规约求和，在其他不会导致精度溢出的地方我们尽最大可能应用了 FP16 来进行运算（比如两个向量求差，差值取绝对值等）。
- 针对 Huge 算子提高数据复用率**: 最后一周我们对之前算子无法承受的更大数据做了适配。进一步测试发现，大数据中访存超越计算成为更严重的瓶颈，所以减少内存访问尤为重要。由于 Huge 规模数据的处理需要把一对向量拆成多段计算，所以在 Normal 中采用的让第一层循环的向量常驻 UB 的优化是无法实现的，这就导致我们需要频繁的访问同一块内存，造成了很大的浪费。我们选择通过将几次计算放在一起，复用同一块第一层循环向量的位置。这个优化提高了数据的复用率，并减少了访存次数，实测能够获得 20% 左右的性能提升。

3.2.5 算子性能

本节将展示一些算子性能，包括任务书要求和我们的进一步测试。所有性能采样均基于 `msprof op` 工具。

基础数据性能

我们首先对任务书给出两个样例形状以及被删除的大数据形状的缩小版进行了性能采样。结果如下：

表 3.2: Pdist 算子性能测试 (us)

N	M	p	float16	float32
100	400	1.0	25.22	23.94
		2.0	25.20	23.56
		∞	24.40	25.36
		3.0	53.28	50.40
2024	3000	1.0	10512.03	11051.92
		2.0	11343.03	11159.28
		∞	11528.57	10475.25
		3.0	103247.77	98264.96
5000	40000	1.0	672501.75	1438629.63
		2.0	719346.19	1453939.38
		∞	742856.56	1464983.25
		3.0	9444155.00	¹

小数据测试时有一定的性能波动，此处结果仅供参考。一个值得注意的点是大规模数据下 FP32 的性能近乎退化为 FP16 的一半，此处分析详见节 3.2.6。

随机数据性能

为了进一步测试算子的通用泛化性和通用性能，我们生成了一些符合规律的随机尺寸数据对我们的算子性能进行测试，并且测试了算子相较于机器 CPU 侧 Pytorch Pdist 算子的加速比来验证我们的优化效果。

配置信息：

- **CPU:** Kunpeng 920 (192 Cores with NEON)
- **NPU:** Ascend 910B4 (40 AI Vector Cores)

测试数据信息：

我们设计了三类数据，为了验证通用性能特别防止了任何对齐的数据出现。描述如下：

- **固定 N , 变动 M :** 其中 $N = 251$ 。
- **固定 M , 变动 N :** 其中 $M = 169$ 。
- **变动 N , 变动 M :** 包括： $N = M$ 、 $N = 4M$ 、 $N = 8M$ 以及 $4N = M$

¹耗时较长，`msprof op` 工具未能返回 profile 结果，`msprof` 工具显示运行时间为 10583748.843 us

测试结果：

下面三张图分别绘制了上述三类数据规模下算子运行时间的对数和计算量量级 ($O(N^2M)$) 对数的关系：

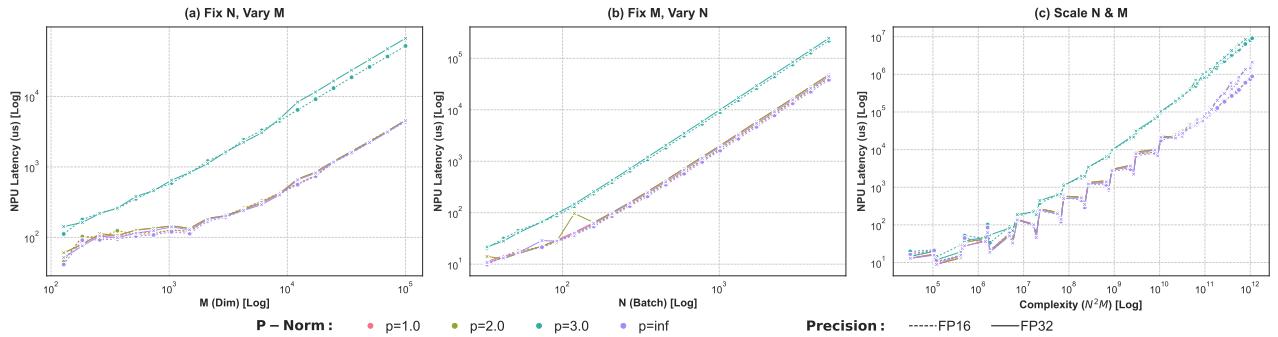


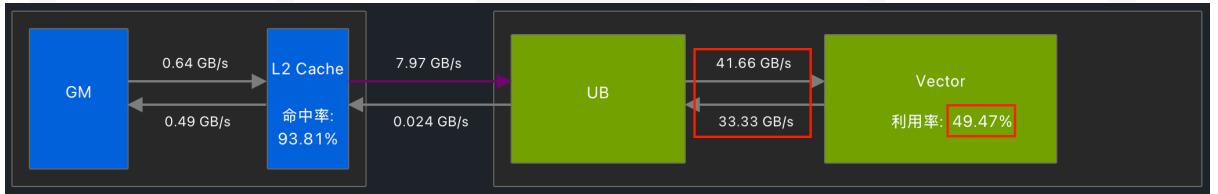
图 3.7: 三类数据规模下算子运行时间与计算量的关系 (对数)

可以看到图表中的折线近似为一条直线 (图 3.7(c) 有较大波动是因为混合了多个比例的 N 和 M), 这说明无论是维度 N 变化、维度 M 变化抑或一起变化, 算子运行时间与计算量总是大致呈线性关系增长的, 进而证明了我们算子的稳定性。

3.2.6 瓶颈分析

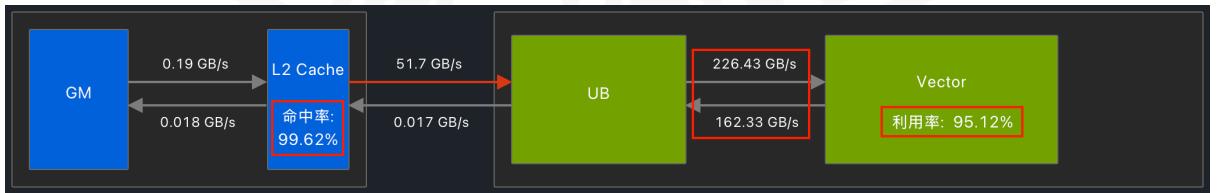
首先是一些宏观的分析。通过对于不同数据规模的 Profile 结果的分析, 我们有如下结论:

- 数据量较小时。**矢量单元利用率仅有约 $40\% \sim 60\%$ 左右。这是由于此时计算和访存都很快, 矢量单元吞吐量和数据搬运单元的带宽都未被充分利用, 标量逻辑占比较大, 指令发射不足。



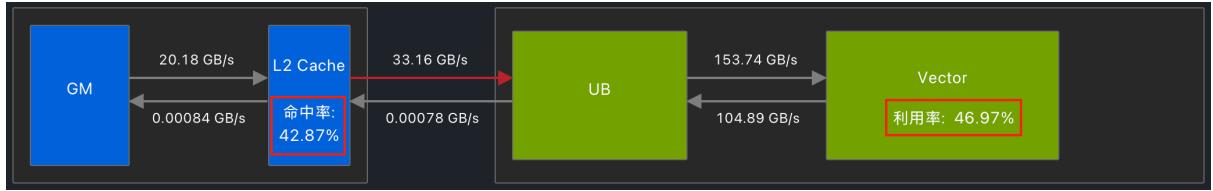
(a) 小量级数据 (100, 400) 的硬件负载

- 数据量增加到一定规模。**矢量单元利用率会逐步上升至 90% 甚至更高。此时由于更多的数据量, 标量同步等占比被掩盖, 又由于 NPU 有很大的 L2 缓存和足够大的 UB, 访存命中率极高 ($> 99\%$), 于是计算成为瓶颈, 矢量单元最忙。(Compute Bound)



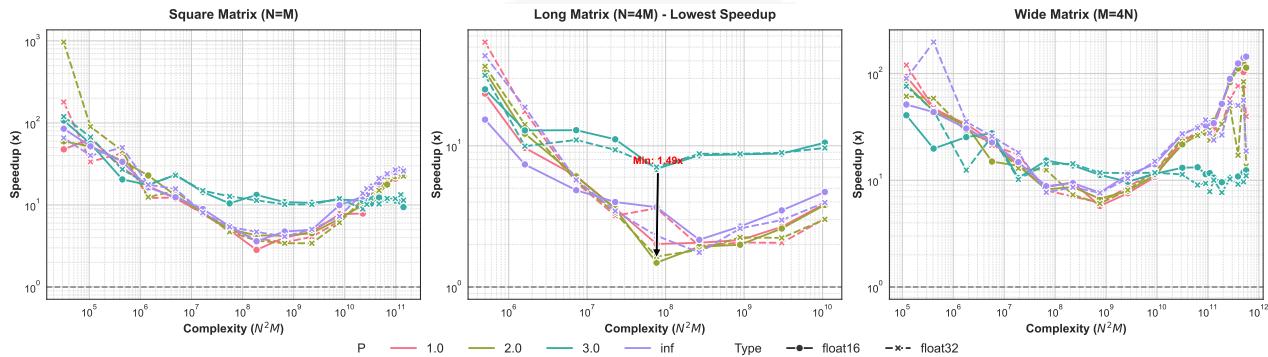
(b) 一定规模量级数据 (2024, 3000) 的硬件负载

- 数据量进一步增大, 尤其触发了 Huge Tiling 处理逻辑之后。**由于我们目前的设计, 一是造成了更多的访存, 二是数据跨度更大, L2 缓存的命中率急剧下降, 此时矢量单元利用率只有 $40\% \sim 50\%$, 可以看到即使双缓冲策略也无法有效掩盖访存延迟, 此时的瓶颈主要是访存, 同时 FP32 和 FP16 的速度差异被凸显, 接近减半的速度也正是访存占比很高的证明。(Memory Bound)



(c) 大量级数据 (5000, 40000) 的硬件负载

接下来是针对我们进行的随机规模数据测试的结果分析。在整个随机测试中，NPU Pdist 算子相对于 CPU Pdist 算子获得了完全正的加速比，但是最大值为约 177 倍，最小值仅约 1.49 倍，平均加速比 24.7 倍。我们将相关结果整理为了如下图表：



同时结合上一节的实验结果，我们有如下分析：

- **固定 M 变动 N 的加速比最低。** 这是由于我们的 M 很小，矢量单元计算压力不大，但是由于要生成非常多对距离，所以标量单元需要发射大量的短小矢量计算任务，标量单元成为性能瓶颈，同时由于较多的计算结果数，访存也在进一步增加，从而导致计算单元无法被充分利用，流水线常被打断。
- **整体加速比随着计算量级呈 U 形分布。** 计算量级较小时，标量等启动开销占比大，此时 NPU 和 CPU 差异并不明显，得益于 NPU 较大的 UB 和 L2 缓存，NPU 可以大批量进行计算，所以相比 CPU 会很快；随着计算量级增大，CPU 得益于平台极大的内存以及较多的核心开始充分并行，NPU 优势缩小（尤其针对 N 大 M 小的情景）；待计算量级进一步变大时，CPU 同样遭受了访存瓶颈，此时得益于 NPU 更高的访存带宽以及更高密度的矢量运算效率，加速比会再次上升。

3.2.7 未来工作

- 按需增加更多特化 pdist 算子的适配工作。Ascend C Power API 性能很差。
- 针对超大规模数据缓存急剧下降进一步优化，比如设计新的处理逻辑（数据重排、复用）等。
- 减少数据规模较小时的标量单元开销。
- 尝试对 L2 范数距离应用 Cube 单元计算。