中国科学技术大学

专业硕士学位论文

中国科学技术大学

基于国产 AI 处理器的 Top-k 算子的设计与实现

作者姓名： xxx

专业领域： 电子信息

校内导师： XXX 教授

企业导师： XXX 高级工程师

完成时间： 二〇二四年十二月十七日

University of Science and Technology of China

A dissertation for master's degree

An example of thesis template for University of Science and Technology of China v3.3.5

Author: xxx

Speciality:

Supervisor: Prof. XXX

Advisor: Senior Engineer XXX. math-font=xits

Finished time: December 17, 2024

中国科学技术大学学位论文原创性声明

本人声明所呈交的学位论文，是本人在导师指导下进行研究工作所取得的成果。除已特别加以标注和致谢的地方外，论文中不包含任何他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的贡献均已在论文中作了明确的说明。

作者签名：_____ 签字日期：_____

中国科学技术大学学位论文授权使用声明

作为申请学位的条件之一，学位论文著作权拥有者授权中国科学技术大学拥有学位论文的部分使用权，即：学校有权按有关规定向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅，可以将学位论文编入《中国学位论文全文数据库》等有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。本人提交的电子文档的内容和纸质论文的内容相一致。

控阅的学位论文在解密后也遵守此规定。

☑ 公开 ☐ 控阅（____ 年）

作者签名：_____ 导师签名：_____

签字日期：_____ 签字日期：_____

摘　要

伴随着深度学习技术的快速发展，Top-k 查询算法在深度学习的各个细分领域得到了广泛应用，成为许多任务中的核心操作。然而，深度学习模型中庞大的数据规模使传统 Top-k 查询算法在处理海量数据时效率低下，难以满足快速响应和精确查询的需求。为此，设计高性能并行的 Top-k 查询算法，并充分结合深度学习处理器的硬件特性优化数据处理效率，成为当前研究的重要方向。PuDianNao 系列处理器是专为深度学习应用设计的国产 AI 硬件架构，具有指令集灵活、并行计算能力强及硬件资源利用率高等特点。然而，现有算法多针对传统 CPU 或 GPU 环境优化，未能充分挖掘国产 AI 处理器的潜力。本文结合 PuDianNao 的硬件特性，设计并实现了两种高效的 Top-k 查询算法，并在深度学习任务中验证了其性能优势与应用效果。主要贡献如下：

1. 基于 Radix-Select 的 Top-k 查询算法的设计与实现

本文设计了一种基于 Radix-Select 算法的 Top-k 查询方法，充分利用了 PuDianNao 处理器在排序操作中的硬件加速能力。通过优化桶操作的并行性、内存访问效率以及流水线指令，大幅提升算法吞吐量。实验结果表明，该算法在数据量较大时，性能相比传统 CPU 基于快速排序的 Top-k 算法提升了 20 倍以上，相较于 NVIDIA A100 GPU 也表现出性能优势，有效降低了查询延迟。

2. 基于 Quick-Select 的 Top-k 查询算法的设计与实现

针对 PuDianNao 的多核架构，设计了一种基于 Quick-Select 的并行 Top-k 查询算法。Quick-Select 通过分区思想逐步缩小搜索范围，结合处理器的多核特性，设计了高效的负载均衡机制和任务分配策略，使算法能在多个计算核上并行执行。通过优化递归调用，减少了不必要的分区和内存操作，进一步提升查询效率。

3. 基于深度学习模型的功能性验证

为验证算法的实际应用效果，本文结合 Pytorch 深度学习框架，将优化后的 Top-k 查询算法应用于 ResNet107 网络的目标检测任务。实验表明，在包含百万级候选区域的大规模数据集中，该算法显著提升了候选区域筛选的速度，同时保持了与基准方法相当的检测精度。

本文提出的基于 PuDianNao 系列处理器的高效 Top-k 查询算法，通过结合硬件特性进行定制化优化，显著提升了深度学习任务中 Top-k 操作的效率。为高效处理海量数据的 Top-k 查询问题提供了新的解决方案。

关键词：高性能并行　Top-k　国产 AI 处理器

ABSTRACT

This is a sample document of USTC thesis LaTeX template for bachelor, master and doctor. The template is created by zepinglee and seisman, which orignate from the template created by ywg. The template meets the equirements of USTC thesis writing standards.

This document will show the usage of basic commands provided by LaTeX and some features provided by the template. For more information, please refer to the template document ustcthesis.pdf.

Key Words: dissertation, abstract, keywords

目　录

符 号 说 明

a	The number of angels per unit area
N	The number of angels per needle point
A	The area of the needle point
σ	The total mass of angels per unit area
m	The mass of one angel
$\sum_{i=1}^{n} a_i$	The sum of a_i

第 1 章　绪　　论

1.1　研究背景及意义

近年来，随着深度学习和人工智能技术的快速发展，全球范围内对数据计算与处理能力的需求急剧上升。特别是在深度学习、推荐系统、搜索引擎和目标检测等应用场景中，Top-k 查询算法作为核心数据筛选工具，被广泛应用于从大规模数据集中选取最优数据。Top-k 查询的主要任务是从海量数据中快速筛选出具有最高优先级或得分的前 k 个数据元素，为后续的模型处理或决策提供支持。尤其是在深度学习模型中，Top-k 查询被频繁用于多个关键操作，例如筛选权重较大的神经元、从候选区域中选出优先级最高的目标框，以及在自然语言处理任务中选取概率最高的词汇或短语。它直接影响模型的运行效率和预测性能，是深度学习系统中不可或缺的重要组成部分。

然而，随着数据规模的指数增长和模型复杂度的不断提升，传统基于 CPU 和 GPU 的 Top-k 查询算法在计算性能、能耗比和响应时间等方面的局限性逐渐显现，难以满足当前深度学习与大数据处理场景中的高性能需求。Top-k 查询算法的核心挑战在于如何在海量数据中以更低的时间复杂度和计算成本完成快速排序和筛选操作。传统算法通常基于排序（如快速排序或堆排序）或分治策略（如快速选择算法），但这些方法在面对超大规模数据集时，往往计算代价高昂且对硬件资源的利用率不够高。此外，随着实时性要求的增加，例如推荐系统中的动态数据筛选、目标检测中的实时候选框生成，传统算法在处理延迟和能耗控制方面的劣势尤为突出。

与此同时，作为人工智能技术的重要支撑，AI 芯片的研发和应用近年来成为全球科技竞争的关键领域。AI 芯片是专为深度学习等人工智能任务设计的硬件加速器，它不同于传统的通用 CPU 或 GPU，而是针对特定任务进行了优化，能够在算力密集型任务中提供更高效的处理能力。例如，AI 芯片通常配备大规模并行计算单元和专用的硬件加速模块，支持高吞吐量的矩阵运算和深度学习推理计算。近年来，国产 AI 芯片作为我国推进科技自主可控、实现关键技术突破的重要组成部分，取得了快速发展。以寒武纪、华为昇腾、天数智芯等为代表的国产芯片在算力、能效比和算法支持等方面已接近国际一流水平。

然而，与国外成熟的 CPU 和 GPU 生态相比，国产 AI 芯片在算法优化和生态完善方面仍有较大的提升空间。许多现有算法的设计和实现主要针对通用 CPU 或 GPU 的硬件架构进行优化，例如利用 GPU 的多线程并行性或 CPU 的缓存特性。而国产 AI 芯片通常具有不同的硬件架构和指令集设计，例如多核异构计算、片上存储和流水线优化等，其性能潜力未被充分挖掘，导致部分任务在国产芯片

上的性能表现并未达到最优。如何通过算法与硬件的深度协同优化，充分发挥国产 AI 芯片的硬件特性，解决传统方法在处理海量数据中的性能瓶颈，成为当前研究的重要方向。

在这一背景下，研究基于国产 AI 芯片的 Top-k 查询算法具有重要意义。首先，该研究有助于提升国产芯片的应用价值和市场竞争力。通过针对芯片架构特性的定制化优化，可以设计出更高效的 Top-k 查询算法，使国产 AI 芯片在深度学习和大数据处理任务中的性能表现更加突出。这将显著增强国产芯片的市场竞争力，推动其在人工智能相关领域的广泛应用，为我国芯片产业的发展提供技术支撑。

其次，该研究将推动深度学习和大数据处理领域的技术创新。Top-k 查询算法是多个关键任务中的基础操作，其性能直接影响整个系统的效率。通过结合国产 AI 芯片的硬件特点进行优化，可以大幅提升查询效率，降低数据处理延迟，为深度学习模型的训练和推理提供更好的支持。这一研究还将为其他领域的高性能算法设计提供参考，如推荐系统、搜索引擎和金融风控等需要实时数据筛选的场景。

再次，该研究对于实现科技自主可控、保障国家数据安全具有重要意义。在当前国际科技竞争加剧的背景下，数据处理与计算能力已成为衡量国家科技实力的重要指标。通过基于国产 AI 芯片的算法优化研究，可以逐步减少对国外硬件平台和技术生态的依赖，形成自主可控的技术体系，提升我国在人工智能领域的核心竞争力。同时，国产芯片在重要领域中的广泛应用，也将进一步保障我国的数据安全和技术主权。

最后，该研究还将为软硬件协同优化提供新的实践经验。AI 芯片的性能提升不仅依赖于硬件设计，还需要与上层算法和应用深度融合。通过针对国产 AI 芯片的硬件架构设计高效的 Top-k 查询算法，可以探索硬件资源的最佳使用方式，例如如何优化流水线执行效率、如何提高内存访问效率等。这一过程将为未来国产芯片的设计和优化积累宝贵经验，推动软硬件协同发展的创新。

综上所述，基于国产 AI 芯片的 Top-k 查询算法研究，不仅在理论上为高效算法设计提供了新思路，还在实践中推动了国产芯片技术的应用落地和生态建设。通过结合硬件特性进行定制化优化，该研究能够显著提升深度学习和大数据处理任务中的计算效率，同时推动人工智能技术的全面发展。这一研究在国家科技战略和产业实践中均具有重要价值，为我国人工智能产业的高质量发展提供了有力支撑。

1.2　Top-k 算法国内外研究现状

1.2.1　国外研究现状

国外学者在 Top-k 查询算法的研究中取得了显著的进展。例如，Fagin 等人提出的基于排序的 Top-k 查询算法通过优化排序过程，大大提高了查询效率[?]。在 TensorFlow 中，Google Brain 团队针对深度学习任务中 Top-k 查询的优化，提出了专门的硬件加速方案[?]。此外，针对大规模数据集，许多研究还探索了并行计算和分布式算法的应用，如利用 MapReduce 架构优化 Top-k 查询性能[?]。

1.2.2　国内研究现状

在中国，随着国产 AI 芯片的崛起，Top-k 查询算法的研究也逐渐聚焦于硬件加速与并行化优化。国内一些学者针对国产 AI 处理器（如华为的 Ascend、寒武纪的系列处理器）进行了针对性的算法设计。例如，李强等人针对国内 AI 芯片提出了一种基于快速选择的 Top-k 查询算法，能够充分利用芯片的多核架构进行并行计算，从而提高查询性能[?]。

1.2.3　结论

综上所述，Top-k 查询算法的研究在国内外均取得了丰富的成果，然而随着数据规模的不断扩大和计算需求的提升，基于 AI 芯片的定制化 Top-k 查询算法将是未来研究的重要方向。

1.3　AI 处理器国内外发展现状

1.4　论文主要内容及章节安排

第 2 章　相关技术背景

2.1　Top-k 算法原理

2.2　PuDianNao 处理器概述

2.3　面向深度学习的处理器的 pytorch 框架

2.4　本章小结

第 3 章　基于国产 AI 处理器的 Top-k 算子设计实现

传统 top-k 查询算法主要是基于通用处理器进行设计和实现的，虽然基于通用处理器设计的算法实现相对简单，但是随着待处理数据规模的不断增大，通用处理器已经无法满足数据处理对于算力的需求。而深度学习处理器专门针对深度学习神经网络处理海量数据处理而设计，为深度学习网络模型进行数据处理提供了强大算力，为各类深度学习算法实现加速，而这一特性是传统 Top-k 查询所缺失的。因此设计实现基于深度学习处理器的 Top-k 算法显得尤为重要。尽管在国产 AI 处理器架构上已经实现了基于冒泡排序的实现，但是当数据规模较大的时候，其性能仍然存在一定的不足。因此本文结合国产 AI 处理器的异构计算特点，根据 radix-select 算法和 quick-select 算法，设计了 Top-k 算子。具体而言，首先概述了 Top-K 算法在异构架构下的计算流程。随后，深入探讨了主机端（CPU）在数据预处理阶段的职责与关键操作。最后，针对国产 AI 处理器的硬件设计，详尽阐述了设备端 Top-k 查询算法的实现策略。

3.1　国产 AI 处理器 Top-k 算子计算流程

在现代计算领域，随着大数据的迅速发展和人工智能技术的不断进步，基于国产 AI 芯片的 Top-k 算子计算流程已经逐渐发展为一种典型的异构计算模式，通常需要主机端（如 CPU）和设备端（如 AI 处理器）协同工作。整个计算流程可以分为三个主要阶段：数据准备和预处理阶段、数据传输阶段和核心计算阶段。

1. 数据准备和预处理阶段（主机端执行）

在整个 Top-k 算子计算过程中，数据准备和预处理阶段是非常关键的部分，通常由主机端负责。在这一阶段，主机端（通常是 CPU）负责接收输入数据，进行格式转换和初步的数据清洗。由于 AI 处理器通常是为特定任务优化的硬件，其处理能力与 CPU 有很大不同，因此需要对输入数据进行适配，以确保数据能够适应后续的硬件计算要求。

此外，在这一阶段，主机端还需要对输入进行解析和验证，以确保其合法性和有效性。这些工作包括检查输入数据的维度、类型，确保数据格式与设备端预期相符等。在实际应用中，这一阶段的计算量相对较小，但却是整个过程的关键环节，因为如果数据准备不当，后续计算将无法顺利进行。

2. 数据传输阶段（主机端向设备端传递数据）

数据准备完成后，接下来的步骤是将处理后的数据从主机端传输到设备端（AI 处理器）。由于 AI 处理器在处理大规模数据时具有极大的并行计算优势，这

一阶段主要是将数据有效地传递到设备端，并确保数据在设备端能够高效地进行计算。在这一阶段，传输过程的高效性直接影响到整体计算的性能。如果数据传输时间过长，会影响整个系统的响应时间。因此，主机端需要选择合适的传输方式，以降低数据传输延迟，确保数据能尽快到达设备端。

3. 核心计算阶段（设备端执行）

核心计算阶段由设备端（AI 处理器）完成，主要任务是根据 Top-k 算子的原理，利用设备端的硬件特性进行高效的并行计算。在传统的 CPU 环境下，Top-k 查询算法通常依赖于单线程或少量线程的处理，效率较低。而在 AI 处理器中，通过硬件的向量化计算指令，可以在多个计算单元上并行执行相同的操作，大大提高计算速度。因此，设备端的高效计算是整个 Top-k 查询计算流程的核心所在。

综上所述，通过主机端负责数据准备和管理，设备端负责高效计算，整个计算流程结合了主机端的灵活性和设备端的计算优势，形成了一种典型的异构计算架构。通过这种分工，能够充分发挥 AI 处理器的硬件优势，提高 Top-k 算法的计算效率。这一计算流程不仅适用于大规模数据处理，还能够在深度学习和大数据分析等领域得到广泛应用。图 3.1展示了该计算流程的具体实现。

3.2　主机端数据准备

3.2.1　接口设计及参数检查

拟设计实现的 Top-k 算子输入参数要求如表3.1所示，其中，input 参数表示输入的张量（tensor），其往往含有多个维度。根据所使用的数据类型，输入张量的元素值可以为整数或浮动小数，支持不同的精度（int32 为整数，float32 为浮动小数）。dim 表示待操作的维度，k 表示需要得到的前 k 大/小的值。假设 input 数据总共有 n 个维度，并且每个维度的大小为 $\{x_0, x_1, \cdots, x_i, \cdots, x_{n-1}\}$，则 dim 和 k 所需要满足的数量关系如下：

$$0 \leqslant dim \leqslant n-1$$

$$\text{设 dim = i，则：} 0 \leqslant k \leqslant x_i$$

对于 largest，其为 bool 类型，为 true 时表示取最大的 k 个值，为 false 时，取最小的 k 个值。对于 sorted，其为 bool 类型，为 true 时表示结果需要排序，为 false 时，不需要将结果进行排序。

输出参数表如表3.2所示。其中，output 是最终的输出 tensor，包含通过 Top-k 查询计算得到的 Top-k 大/小值。其数据类型可以是 int32 或 float32，具体取决于输入数据的类型以及计算要求。index 是与 output 对应的下标。它指示从输入数据的第 dim 维度的哪个位置获取的 Top-k 元素。这在需要返回结果的源数据位置时非常有用，尤其是在需要追踪或处理原始数据时。

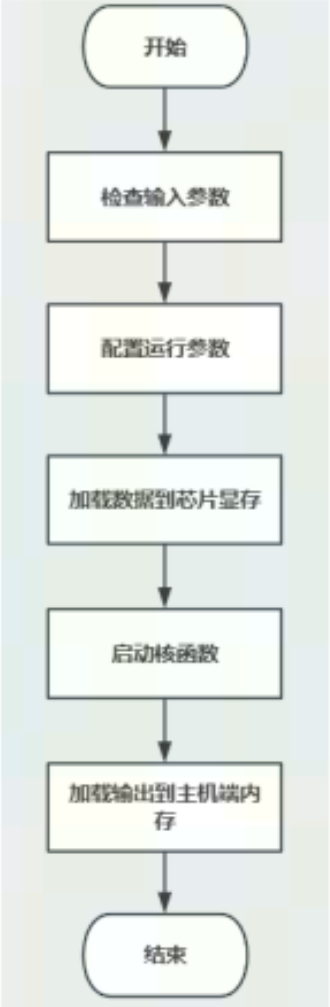

图 3.1　算子整体流程

表 3.1　输入参数表

参数名称	数据类型	描述
input	int32/float32	输入 tensor，可以是任意维度
dim	int32	表示对第 dim 维度进行操作
k	int32	表示取排行前 k 的数据
largest	bool	默认为 true, 控制取最大还是最小的值
sorted	bool	默认为 false，控制是否需要排序

3.2.2　运行参数配置

（1） radix-select 任务类型

需注意的是，radix-selct 设备端的并行方案需要多次启动 kernel 函数，并且需要多核协同完成任务，因此每次发起的 kernel 的任务类型为 UnionX。而具体需要多少个 core，则根据当前的数据规模来计算得到。

（2） quick-select 任务类型

quick-select 的 kernel 函数的任务类型为 UnionX，X 的具体值根据输入数据的大小来进行确定。

表 3.2　输出参数表

参数名称	数据类型	描述
output	/int32/float32	输出 tensor
index	int32/uint32	输出数据在排序维度对应的下标

3.2.3　数据预处理及后处理

3.3　设备端 Top-k 算法实现

3.3.1　设备端数据摆布方式分析

Top - K 需要支持多维度数据, 可以表示如下: dim0, dim1, dim2, dim3。由左到右表示维度从高到低，按照 dim 可以将四个维度转化为：left，dim，right，left 为最高维度，dim 为中间维度。当 dim = 3 时表示对 dim3 维度进行 Top - K 查询，此时 right = 1，left = dim0 × dim1 × dim2。当 dim = 0 时，表示对 dim0 求 Top - K，则对应的 left 和 right 分别为：1，dim1 × dim2 × dim3。

综上所述可以将任意维度的数据抽象成一个三维张量，其形状为 left，dim，right，其中，dim 是待操作维度。但是当 right ≠ 1 时，待操作维度不在最低维，这意味进行数据 I/O 时，将有可能浪费有限的带宽。因此，在这种情况下需要针对数据后面两个维度进行转置（transpose）操作。此时张量的形状为（left，right，dim），进一步的，对于 Top - k 算子而言，其输入的形状可以抽象为（left_right，dim），其中 left_right = left × right。当 Top - k 任务完成之后，输出的形状将为（left_right, k), 此时需要对结果再次进行转置操作, 其输出形状为（left, k, right），作为最终 Top - k 算子的最终结果。其形状的转变流程如图 3.2

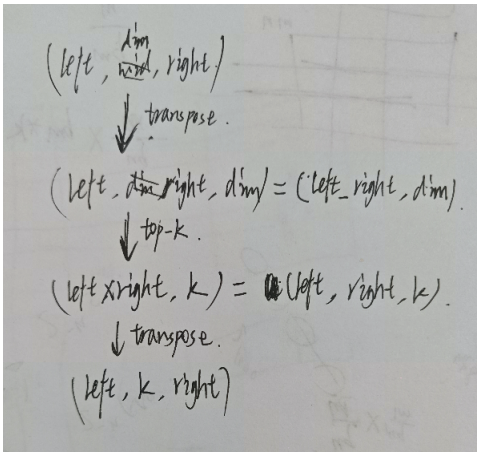

图 3.2

3.3.2　单核 radix-select 分析及实现

基数选择（radix-select）是基数排序的一种应用，用于在一组数据中找到具有特定排名的元素。其算法过程与基数排序算法相似，但与之不同的是，其在迭

代的过程中，RadixSelect 仅对包含第 k 大元素的数进行基数排序，达到减少问题规模的效果。在 radix-select top - K 算法中，一个数位（digit）对应着一个元素二进制表示中的一组连续的 b 位。该算法从最高有效数位（most significant digit）到最低有效数位（least significant digit）处理一个元素，每次迭代处理一个数位。对于一个由 r 位组成的元素，需要进行 $\lceil \frac{r}{b} \rceil$ 次迭代。每次迭代主要包含四步，其伪代码见算法 3.1。

算法 3.1　Counting - Sort Algorithm

Input: Array D with n entries, integer $k \leqslant n$
Output: array R is the Top-k elements

```
Bucket ← init 0 // size is 2^digit
HISTOGRAM - KEYS
for j ← 0 to N − 1 do
    Bucket[D[j]] ← Bucket[D[j]] + 1
end
SCAN - BUCKETS
Sum ← 0
for i ← 0 to 2^r − 1 do
    Val ← Bucket[i]
    Bucket[i] ← Sum
    Sum ← Sum + Val
end
FIND TARGET BUCKET
for i ← 0 to 2^r − 1 do
    if Bucket[i] ⩾k then
        POS ← Bucket[i]
        break;
    end
end
FILTER
for j ← 0 to N − 1 do
    If D[j] ⩾POS continue;
    A ← Bucket[D[j]]
    R[A] ← K[j]
    Bucket[D[j]] ← A + 1
end
```

（1） 步骤详解

1. 初始化桶

首先，需要初始化一个名为 $Bucket$ 的数组，其大小设置为 2^{digit}，并且将数组中的所有元素初始化为 0。其中 digit 为读取数据的位数，这个 $Bucket$ 数组的主要作用是用于统计数组 D 中每个元素出现的次数。

2. 构建直方图

在此阶段，会遍历数组 D 中的每一个元素 $D[j]$，统计 $D[j]$ 中元素出现的个数，记录在 $Bucket$ 中。

3. 扫描桶

计算出每个桶的累积和，而这个累积和对于确定每个元素在排序后的位置起到了关键作用。

4. 找到目标桶

再次遍历 $Bucket$ 数组，目的是找到第一个满足 $Bucket[i] \geqslant k$ 的桶。

5. 过滤

在此阶段，会再次遍历数组 D，目的是将数组 D 中的前 k 大元素挑选出来，并放入结果数组 R 中。

3.3.3　多核 radix-select 分析及实现

上述串行算法由于在所有阶段都存在循环依赖关系，所以不能直接并行化（或向量化）。如果试图对” 构建直方图”（HISTOGRAM - KEYS）的迭代进行并行处理，可能会有多个处理器同时尝试对同一个桶（bucket）进行递增操作，产生资源竞争。而如果锁定桶，当有许多键具有相同数位时，桶将成为串行瓶颈，将会极大地降低性能。

基于以上的分析，可以通过为每个处理器分配一组独立的桶（本地桶），进而在不需要对桶进行锁定的情况下实现算法并行化: 即每个处理器负责其自己的局部数据（$\frac{N}{P}$ 子集），并将它们列入自己的本地桶集合中。此时，可以将所有的桶看作是一个矩阵 $Buckets[i, j]$，如下图 3.3。

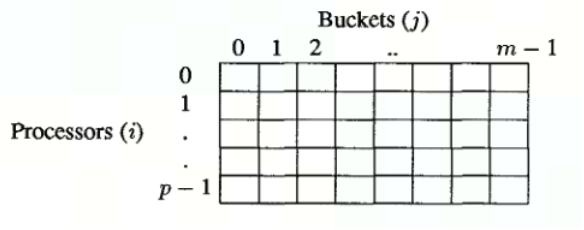

图 3.3

其中 i 是处理器数, j 是本地桶的桶数。在每个处理器都拥有自己的桶之后, 每个处理器在进行基数选择的第一阶段, 第三阶段和第四阶段（HISTOGRAMKEYS，FIND TARGET BUCKET 和 FILTER）时，都在自己的局部数据上进行工作，从而解除所有依赖项。但需要注意到，为了保证结果的正确性，第三阶段（SCAN-BUCKETS ）将必须进行修改。因为如果每个处理器仅扫描自己的本地桶，将仅仅只能得到局部数据的 Top-k 结果，导致结果错误。而通过分析我们发现，$Buckets[i][j]$ 代表在进行 SCAN-BUCKETS 处理后，A 中元素最终在结果 R 中的偏移位置，具体关系见下图 3.4。

假如该算法用 4 个处理器和 4 个桶来对 0 - 3 的值进行排序。偏移量是通过扫描桶来计算的。扫描之后，每个处理器在具有特定值的键的最终输出中都有一个起始位置。例如，处理器 3 将把值为“0”的键从输出中的第 5 个位置开始放

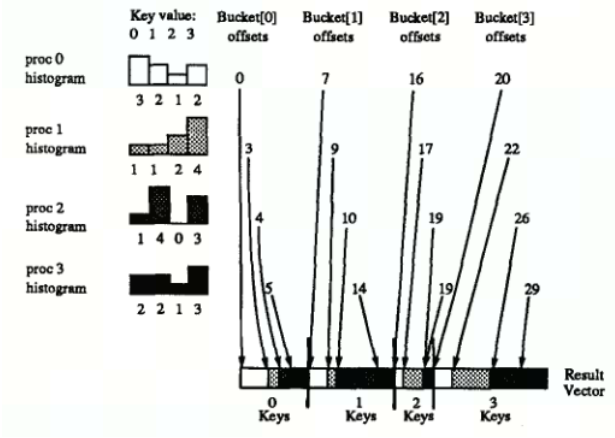

图 3.4　并行 radix-select 的扫描步骤

置，把值为“1”的键从第 14 个位置开始放置，如偏移量所示。

所以需要将矩阵 $Buckets[i,j]$ 在列方向上以及行方向上进行扫描操作，以汇总全局信息，其数学公式见 3.1

$$\text{Buckets}[i,j]' = \sum_{k=0}^{p-1}\sum_{m=0}^{j-1}\text{Buckets}[k,m] + \sum_{k=0}^{i-1}\text{Buckets}[k,j] \tag{3.1}$$

也就是说，偏移量是所有处理器中小于 j 的数位总数，再加上小于 i 的处理器中数位等于 j 的数量。这个总和可以通过将 $Buckets$ 矩阵按列优先顺序展开，然后对展开后的矩阵执行“SCAN - BUCKETS”操作来进行计算。同时，找到目标桶这一阶段，Bucket 数组中，仅第一行中的数据代表着全局的前缀信息，因此仅需要 Core0 进行这一阶段的主要任务即可。在并行实现中，主要工作流程分为以下几步：

1. 对于输入数据（二维 Tensor），将数据按照行进行切分，每个 Cluster 负责数据的若干行，而后每个 Cluster 对所分配的行依次处理。
2. 对于 Cluster 正在处理的某一行数据而言，采用多核进行计算。因此需要将整行数据平均分到各个处理器上。
3. 在单核上，由于片上空间有限，因此需要将局部数据进一步划分成 repeat 段，循环依次 load 到片上进行操作，统计出局部前缀和数组 P_local。
4. 根据 P_local，按照前文分析计算出全局前缀和数组 P。
5. 根据 P，让 Core0 执行算法的第四阶段（FIND TARGET BUCKET），找到 k_bins。
6. 根据 k_bin 和 P，执行过滤操作。将 k_bin 之前的数据复制到输出所在的内存，并记录好 offset。而后将 k_bin 中的数据，作为下一次迭代的输入。

（1） 并行实现

以下结合国产 AI 处理器的内存层次和指令，对算法的各个主要阶段进行阐述：

① 初始化桶

首先，需要初始化一个名为 $Bucket$ 的数组，其大小设置为 $2^{digit} \times processor$，并且将数组中的所有元素初始化为 0。其中 digit 为读取数据的位数，processor 为具体任务类型所对应的处理器数目，这个 $Bucket$ 数组的主要作用是用于统计数组 D 中每个元素出现的次数。每个处理器管理的数组大小为 2^{digit}。

② 构建直方图

为了节省内存空间，不定义数组 D 所需要的空间，而是在此阶段使用 bang_band_cycle 指令，根据输入和循环所处的遍数计算出 D，而后使用 bang_histogram 统计当前统计 $D[j]$ 中元素出现的个数，记录在 $Bucket\_local$ 中。在处理器处理完所有的局部数据后，将 $Bucket\_local$ 复制到片外内存 $Bucket$ 中。

③ 扫描桶

计算出每个桶的累积和, 。

④ 找到目标桶

再次遍历 $Bucket$ 数组，目的是找到第一个满足 $Bucket[i] \geqslant k$ 的桶。在此阶段，主要使用 bang_gt_cycle 和 bang_filter 指令来确定目标桶的下标。

⑤ 过滤

在此阶段, 会再次遍历输入数组 A 求得 D, 根据目标桶的下标得到 digit_num, 而后使用 bang_gt 和 bang_filter 指令将 A 中的数据筛选出来，再根据 Bucket 中的偏移值将数据复制到指定位置。另外将等于 digit_num 的数据筛选出来放到数组 R 中。

(2)　基数选择对数据类型与数值正负性的适应性分析

基数选择作为一种重要的排序类算法，其在不同数据类型以及数值正负性处理上具有特定的要求与表现，以下将对此展开深入探讨。

① 数据类型要求

基数选择在整数领域应用广泛且具有不同处理方式。对于固定长度的无符号整数，如常见的 32 位无符号整数数组，可依据其二进制表示直接实施基数选择。选择过程可选择从最高有效位（MSB）至最低有效位（LSB），按位依次对整数进行选择操作。例如，在对一组无符号 32 位整数选择时，先依据最高位的值将整数分配至不同桶中，完成一轮选择后，再依据次低位进行同样操作，直至处理完所有位，最终选择出来目标数。但对于有符号整数, 浮点数，由于符号位的存在，会导致负数比正数更大的非预期效果。因此需要针对不同的数据类型以及数据的正负性进行进一步的分析。

(一) 整数类型

针对整数类型数据，基数选择可基于其特定二进制表示形式开展。

在遵循计算机中常用的二进制补码表示的整数表示中，以 $int32$ 数据类型为

例，整数在内存中的存储分为以下几部分：

符号位（*Sign bit*）：表示整数的正负，0 表示正，1 表示负。对于 *int*32 类型来说，其最高位（第 31 位）就是符号位。

数值部分（*Value part*）：剩余的 31 位（第 0 位至第 30 位）用于表示整数的绝对值大小（在正数情况下，其对应的二进制数值就是实际的数值；在负数情况下，这 31 位表示的是该负数绝对值的补码形式）。

具体而言，在正数时，例如十进制的 5 用二进制表示就是 00000000 00000000 00000000 00000101，其最高位符号位为 0，后面 31 位就是实际数值 5 的二进制表示。

而对于负数，比如十进制的 –5，先取其绝对值 5 的二进制表示 00000000 00000000 00000000 00000101，然后进行补码运算（按位取反再加 1），得到 11111111 11111111 11111111 11111011，此时最高位符号位为 1，表示这是一个负数，后面 31 位按照补码规则来表示其绝对值大小，通过这样的形式实现了在计算机内存中用 32 位对 *int*32 类型整数（包含正负）的存储，而基数选择算法在处理 *int*32 类型整数时，若直接按照无符号整数的逻辑进行运算，将会产生负整数大于非负整数的现象。其根本原因就是因为符号位的存在以及负整型数据的编码规则，整型数据的各个比特位的重要性并不是从最低位依次递增的，因此需要进行预处理操作。

为了更好的说明问题，且基数选择算法与基数排序算法具有很强的一致性，下文以基数排序来进行举例，说明问题的现象以及对应的解决方案。设原始数据为 data0，具体数据为 [-2, 1, 0, -3, 3, -1, 2] 7 个 int32 数据。其表示见下图：

index	2	1	6	4	3	0	5
10进制表示	0	1	2	3	-3	-2	-1
16进制表示	0x00000000	0x00000001	0x00000002	0x00000003	0xFFFF FFB	0xFFFFFFFE	0xFFFFFFFF
二进制表示	00000000 00000000 00000000 00000000	00000000 00000000 00000000 00000001	00000000 00000000 00000000 00000010	00000000 00000000 00000000 00000011	11111111 11111111 11111111 11111011	11111111 11111111 11111111 11111110	11111111 11111111 11111111 11111111

图 3.5

原始数据经过基数排序后的 data1 为：

index	2	1	6	4	3	0	5
10进制表示	0	1	2	3	-3	-2	-1
16进制表示	0x00000000	0x00000001	0x00000002	0x00000003	0xFFFF FFB	0xFFFFFFFE	0xFFFFFFFF
二进制表示	00000000 00000000 00000000 00000000	00000000 00000000 00000000 00000001	00000000 00000000 00000000 00000010	00000000 00000000 00000000 00000011	11111111 11111111 11111111 11111011	11111111 11111111 11111111 11111110	11111111 11111111 11111111 11111111

图 3.6

经过基数排序后，结果与预期不符：结果的排布方式为正数为非降序，负数为非降序，且负数在正数后。对于这种问题目前的解决方案有两种：

a. 当升序排序时，预先得到负数和非负数的总个数，然后据此将正数和负数

写回的 index 进行偏移。b. 对数据进行预处理，对符号位进行变换，在最后再将变换后的数据进行还原。

因为方案 b 可以直接使用位操作指令，性能更快，且实现更加简便，因此这里使用了方案 b 进行实现。具体的实现方案如下：

预处理：因为期望非负数在负数之后，所以直接将所有数的符号位进行取反操作，即直接将输入数组中的各个元素与 0x80000000 进行位异或。

后处理：恢复原来的数据，基数排序完成后 index 是符合期望的，现在需要做的是将原始数据 revert 回来；因此将输出数据再次与 0x80000000 进行位异或。

上述步骤主要使用到的平台指令及具体作用如下：

1. 按 bit 取反：bang_bxor_scalar 和 bang_mlu

通过上述处理之后，数据将会正常显示：

index	3	0	5	2	1	6	4
10进制表示	-3	-2	-1	0	1	2	3
16进制表示	0xFFFF FFB	0xFFFFFFFE	0xFFFFFFFF	0x00000000	0x00000001	0x00000002	0x00000003
二进制表示	11111111 11111111 11111111 11111011	11111111 11111111 11111111 11111110	11111111 11111111 11111111 11111111	00000000 00000000 00000000 00000000	00000000 00000000 00000000 00000001	00000000 00000000 00000000 00000010	00000000 00000000 00000000 00000011

图 3.7

（二）浮点数类型

针对浮点数类型数据，基数选择可基于其特定二进制表示形式开展。在遵循 IEEE 754 标准的浮点数表示中，以 float32 数据类型为例，浮点数在内存中的存储分为以下三部分：

符号位（Sign bit）：表示浮点数的正负，0 表示正，1 表示负。指数部分（Exponent）：表示浮点数的指数，用偏移量编码（偏移量也称为偏置，为 127）。尾数部分（Mantissa）：表示浮点数的有效数字，通常采用归一化表示（即小数点左边默认为 1）。

图 3.8

这种结构设计使得浮点数在数值表示上具有独特的规律，虽然不同于整型数据，但是其中从 0 到 30 位（除符号位）对数字大小的重要性仍然是依次递增。同样的，由于符号位的存在，其存在与有符号整数相似的问题（但不完全相同）。以 float 数据类型为例，half 类型同理。设原始数据为 data0，具体数据为 [-2, 1, 0, -3, 3, -1, 2] 7 个 float32 数据，对其进行为了更好的说明问题，此处我们同样以基数排序进行问题的说明。

原始数据经过基数排序后的 data1 为：

index	0	1	2	3	4	5	6
10进制表示	-2	1	0	-3	3	-1	2
16进制表示	0xC0000000	0x3F800000	0x00000000	0xC0400000	0x40400000	0xBF800000	0x40000000
二进制表示	11000000 00000000 00000000 00000000	00111111 10000000 00000000 00000000	00000000 00000000 00000000 00000000	11000000 01000000 00000000 00000000	01000000 01000000 00000000 00000000	10111111 10000000 00000000 00000000	01000000 00000000 00000000 00000000

图 3.9

index	2	1	6	4	5	0	3
10进制表示	0	1	2	3	-1	-2	-3
16进制表示	0x00000000	0x3F800000	0x40000000	0x40400000	0xBF800000	0xC0000000	0xC0400000
二进制表示	00000000 00000000 00000000 00000000	00111111 10000000 00000000 00000000	01000000 00000000 00000000 00000000	01000000 01000000 00000000 00000000	10111111 10000000 00000000 00000000	11000000 00000000 00000000 00000000	11000000 01000000 00000000 00000000

图 3.10

经过基数排序后，结果与预期不符：结果的排布方式为正数为非降序，负数为非升序，且负数在正数后。对于这种问题目前的解决方案有两种：

a. 当升序排序时，预先得到负数和非负数的总个数，然后据此将正数和负数写回的 index 进行偏移，同时由于负数为降序排序，所以将 data1 中负数内部进行 reverse。

b. 对数据进行预处理，对符号位进行变换，在最后再将变换后的数据进行还原。

因为方案 b 可以直接使用位操作指令，性能更快，且实现更加简便，因此这里使用了方案 b 进行实现。具体的实现方案如下：

预处理：因为期望非负数在负数之后，所以考虑将所有数的符号位进行取反操作；同时期望负数内部为升序排列，所以考虑将负数的非符号位进行取反操作；即：对正数进行符号位取反，对负数全部 bit 位进行取反，即非负数数与 0x80000000 进行位异或操作, 负数与 0xffffffff 进行位异或操作;（任何数与 0 异或保持不变，与 1 异或会取反）

后处理：恢复原来的数据，基数排序完成后 index 是符合期望的，现在需要做的是将原始数据 revert 回来；即将对正数进行符号位取反，对负数全部 bit 位进行取反，得到最终结果 RES。

通过上述处理之后，数据将会正常显示：

index	3	0	5	2	1	6	4
10进制表示	-3	-2	-1	0	1	2	3
16进制表示	0xC0400000	0xC0000000	0xBF800000	0x00000000	0x3F800000	0x40000000	0x40400000
二进制表示	11000000 01000000 00000000 00000000	11000000 00000000 00000000 00000000	10111111 10000000 00000000 00000000	00000000 00000000 00000000 00000000	00111111 10000000 00000000 00000000	01000000 00000000 00000000 00000000	01000000 01000000 00000000 00000000

图 3.11

上述步骤主要使用到的平台指令及具体作用如下：

a. 用来得到区分非负数和负数的 mask：bang_band_scalar 和 bang_eq_scalar

b. 按 bit 取反：bang_bxor_scalar 和 bang_mlu

（3） 基数选择对最大/最小值的分析

基于前文针对正负性所展开的讨论，在实施 radix - select 算法的进程中，若预先对数据予以预处理，则 radix - select 函数的功能性能够得到优化，进而无需对数值的正负特性以及数据类型予以过度的关切。鉴于数据预处理所具有的关键意义，本文深入探究了借助数据预处理来达成 radix - select 函数与最大值或最小值相解耦的可行性。通过深入且细致的分析后可知，恰似处理正负性问题的情形，凭借数据预处理的策略，能够将“Top - k 大”的问题有效地转换为“Top - k 小”的问题，从而显著地削减 radix - select 函数在实现过程中的复杂程度。

（一）整数类型为了更为清晰地阐释相关问题，并且鉴于基数选择算法与基数排序算法之间存在着高度的一致性，下文将以基数排序为例，阐述问题的现象及其对应的解决方案。

设定原始数据为 data0，其具体数据为 [-2, 1, 0, -3, 3, -1, 2] 这 7 个 int32 类型的数据。倘若在此情形下需要获取前 Top - k 大的元素，那么我们能够通过对原数据进行适当的更改，从而将问题转化为“获取前 Top - k 小的元素”。

依据上文的分析，若直接对 data0 执行基数排序操作，将会得到 [0, 1, 2, 3, -3, -2, -1] 这样的排序结果。考虑到我们实际所期望的结果是降序输出，就负数与正数的整体相对位置而言，其输出结果是与预期相符的，故而数据的符号位无需进行变更。然而，针对负数内部的排序结果以及正数内部的排序结果来讲，均与预期的降序结果相互背离。因此，需要对所有数据的非符号位实施取反操作。通过上述一系列的操作处理，我们能够简洁高效地将原始问题进行转换。

（二）浮点数类型为了更为有效地说明问题，鉴于基数选择算法与基数排序算法之间的紧密关联性，以下将以基数排序作为示例，用以说明问题的现象及其对应的解决方案。

设定原始数据为 data0，其具体数据为 [-2.0, 1.0, 0.0, -3.0, 3.0, -1.0, 2.0] 这 7 个 float32 类型的数据。若在当前情形下需要获取前 Top - k 大的元素，那么我们可通过对原数据进行调整，将问题转化为“获取前 Top - k 小的元素”。

依据前文的分析，若直接对 data0 执行基数排序，将会得到 [0.0, 1.0, 2.0, 3.0, -1.0, -2.0, -3.0] 这样的结果。考虑到我们所需求的结果是降序输出，对于负数而言，无论是负数内部的排序结果，还是负数相对于正数的排序结果，均与预期相契合，因此数据的符号位保持不变且负数无需进行任何改动。而对于非负数部分，由于所期望的输出应当是数值越大其相对顺序越小，所以需要对非负数的非符号位进行取反操作。通过上述的操作流程，我们能够便捷地将原始问题予以转

换。

3.4　本章小结

第 4 章　基于国产 AI 处理器的 Top-k 算法性能优化

本章首先介绍了阿姆达尔定律和古斯塔夫森定律，为 Top-k 算子的性能优化指明了方向，即提升 Top-k 算子程序的并行度。接着从计算效率方面入手，充分利用国产 AI 处理器支持多级并行计算的优势，提高了 Top-k 算子的并行计算能力。最后从访存效率角度出发，通过多种优化技巧减少了 Top-k 算子的访存耗时。

4.1　算子优化策略

阿姆达尔定律是计算机科学领域的一条经验定律，该定律表明了并行计算、存储系统获得的加速比和处理器数量以及程序并行化程度之间的关系，其计算公式见 (4.1)。

$$S = \frac{1}{1 - P + \frac{P}{N}} \tag{4.1}$$

式 (4.1) 中，S 表示并行化前后程序获得的加速比，P 表示程序中可实现并行化处理部分的比例，N 则表示并行处理器的数量。从该式可以看出，当 P 不变时，程序用到的并行处理器数量越多，就能获得越大的加速比。但是当 N 远大于 P 时，程序获得的加速比会逐渐趋近于固定的数值。即当 N 趋近于无穷大时，式 (4.1) 近似为 (4.2)。

$$S = \frac{1}{1 - P} \tag{4.2}$$

式 (4.2) 表明即使处理器数量足够的多，倘若程序并行化处理代码占比不高的话，那么程序依然无法获得很好的加速比。因此要想获得更大的加速比，这就需要开发者充分发掘程序中可进行并行化处理的部分，从而尽量提升程序中并行化处理代码的占比。

根据阿姆达尔定律，在处理器足够多的条件下，当程序并行化处理比例达到 95% 时，程序获得的加速比也依然只有 20 倍。这是因为阿姆达尔定律限定了处理问题的规模以及程序可并行化的比例。在实际中通常使用大量的处理器来处理大规模的问题，这些大规模的问题也能分成多个小规模的问题进行并行处理。在这样的背景下，古斯塔夫森定律被提出，其公式见 (4.3)。

$$S = N - P \cdot (N - 1) \tag{4.3}$$

式 (4.3) 中，S 表示并行化前后程序获得的加速比，N 表示并行处理器的数

量，F 表示程序中串行部分代码的占比。该式表明，当程序的串行部分代码占比足够小，即程序并行化程度足够大时，程序获得的加速比和处理器数量成正比。因此在程序并行化程度足够高的情况下，增加处理器数量可以很好地提升程序运行的性能。通过上述两个定律可知，在处理器数量不变的条件下，若想对实现的 Top-k 算子子程序进行性能调优，就需要尽可能地提升算子代码的并行化程度。根据 MLU 系列 AI 处理器的硬件架构特性，可以从计算效率和访存效率两个方面提高算子程序的并行度。

4.2　Top-k 算法访存效率优化

4.2.1　radix-select Top-k 算法的 digit 调优

以 float-int32+ 默认 kernel 作为一般情况分析：radix 的选择（对应代码变量 digit 的大小），会在以下三个大方向上影响运行时间：1.I/O 时间，当 digit 为 1 的时候，bucket_num=2，float 对应 32bit，需要 32 次捞取，digit 为 4，bucket_num=16，需要 8 次捞取，digit 为 32，bucket_num=232，需要捞取 1 次，radix 选择的越大，则 I/O 时间越短。

4.2.2　显存 I/O 优化

在国产 AI 处理器的复杂体系结构中，显存 I/O 操作的效率是制约整体系统性能提升的关键因素之一。数据双缓冲、数据对齐以及数据缓存作为重要的优化策略，从不同角度对显存 I/O 进行精细化调控，旨在最大程度地提高数据传输速率、降低延迟并提升资源利用率，以满足国产 AI 处理器在多样化人工智能任务中的严苛需求，增强其在全球 AI 计算领域的竞争力与适应性。

（1） 数据对齐优化分析

显存带宽利用率是指在图形处理单元（GPU）或其他使用显存的计算设备中，实际使用的显存带宽与显存理论带宽的比率。显存带宽是指显存与核心之间数据传输的速率，它的计算公式为：带宽 = 显存频率 × 显存位宽 ÷8（单位为字节 / 秒）。例如，一款显存频率为 1000MHz、显存位宽为 128 位的显卡，其理论带宽为 1000MHz×128÷8 = 16000MB/s。显存带宽利用率优化的目的是尽可能地提高这个比率，使显存与片上内存（或寄存器）之间的数据传输更加高效，从而提升整个系统的性能。针对国产 AI 处理器，常用的优化方法包括数据布局优化。a. 连续内存访问：在编程中，尽可能地保证数据在显存中的存储是连续的。因为在访问连续内存时，缓存命中率更高，数据传输效率也更高。例如，在处理图像数据时，将图像的像素按照行优先或列优先的顺序连续存储在显存中，而不是分散存储。b. 数据对齐：确保数据的存储地址按照的内存访问要求进行对齐。不同

的架构对内存对齐有不同的要求，一般来说，将数据的起始地址对齐到缓存行大小（如 32 字节或 64 字节）的倍数，可以提高内存访问效率。对于国产 AI 处理器的不同层次的内存，其对其要求见下表 4.1。

存储类型	地址对齐	数据对齐
NRAM	16B	128B
SRAM	16B	128B
WRAM	64B	128B
GDRAM	512B	512B
LDRAM	512B	512B

表 4.1　存储类型及其对齐方式

（2） 双缓冲优化分析

数据双缓冲机制基于并行处理思想，于国产 AI 处理器与显存间构建缓冲 A 和缓冲 B 两个独立缓冲区。数据传输时，缓冲 A 数据被处理，缓冲 B 可填充，反之亦然。设数据处理时间为 T_{process}，数据填充时间为 T_{fill}，传统单缓冲总时间 $T_{\text{single}} = T_{\text{process}} + T_{\text{fill}}$，双缓冲总时间 $T_{\text{double}} = \max(T_{\text{process}}, T_{\text{fill}})$，该并行模式减少数据等待时间，提升数据传输效率与显存 I/O 吞吐率。在国产 AI 处理器显存与计算单元中间，存在多级的内存层次。其中共享内存和 NRAM 可以通过程序来进行访存行为的控制。因此可以基于这一点将双缓冲优化手段应用于 Top-k 算子的开发过程中。

同时，MLU Core 支持多指令流并行，Topk 算子各计算阶段涉及加载输入数据、计算输入数据和存储计算结果。加载与存储指令分配到数据搬移队列，计算指令分配到数据计算队列。有数据依赖的计算与访存指令时序分开，无依赖的则并发执行，减少指令等待时间，提高访存效率，提升 Top-k 算子性能，此计算与访存并行通过软流水实现。软流水是特殊指令重排，将同一循环内不同循环迭代间计算和访存重组，使处理同一批数据的计算和访存分散到不同循环迭代，消除同一循环迭代内计算与访存依赖关系，让指令并行执行，相互隐藏执行时间，减少循环体总执行时间，后续将阐述 Top-k 算子访存效率优化中的三级流水和五级流水机制。

① 三级流水

常用的三级流水，如图 4.1所示。其中 L 代表 Load 操作，表示将数据从 GDRAM 或者 SRAM 加载到 NRAM 的过程；C 代表 Compute，表示数据在 MLU Core 上运算的过程；S 代表 Store 操作，表示将计算结果从 NRAM 存回到 GDRAM 或者 SRAM 的过程。在空间维度上，片上空间被划分为 Ping 和 Pong 两块，每块都可以用来临时存放输入数据和输出数据；在时间片维度上，可进行的访存或计算操作包括了 Load，Compute 和 Store。从图 4.1可知，通过软流水，在同一个时间片内可以进行加载（Load）、计算（Compute）以及存储（Store）操作。当

图 4.1

MLU 核心进行计算操作时，各个存储空间也在不断地进行数据传输操作，从而实现了计算与访存的并行，减少了数据等待时间，进而提高了访存效率。

需要注意的是，由于片上随机存取存储器（NRAM）的容量有限，对于计算过程较为复杂的计算任务，通常会将 NRAM 划分成多段。若再进行乒乓分区，MLU 核心一次能处理的数据量将会进一步减少。在这种情况下，软流水往往会带来负面效果。因此，对于 MLU 上的梯度插值和索引计算过程，也就是投票（Vote）计算阶段，不进行软流水操作；而对于直方图（Hist）阶段的计算，则可以进行三级流水操作。

② 五级流水

五级流水是在三级流水的基础上，利用硬件内存核心（Memory Core）实现的。它以静态随机存取存储器（SRAM）作为缓冲区，将加载（Load）和存储（Store）操作各分为两部分，一部分由 MLU 核心完成，另一部分由内存核心（Memory Core）完成。在 Top-k 算子的梯度（Grad）计算阶段，利用 SRAM 来完成图像边缘梯度的填充。实际上，SRAM 也可用于对整个计算过程实现五级软流水操作。整个五级流水过程如下：首先由 Memory Core 将数据从 GDRAM 搬运到 SRAM 中，然后再由 coreDim（以 4 为例）个 MLU Core 分别从 SRAM 搬运部分数据到 NRAM 中进行计算，计算结果先由 coreDim 个 MLU Core 搬运到 SRAM 上，再由 Memory Core 从 SRAM 搬运到 GDRAM 上。五级软件流水运行过程，如图 4.2 所示。在空间维度上，片上空间 SRAM 和 NRAM 都被划分为 Ping 和 Pong 两块，每块都可以用来临时存放输入向量和输出向量；在时间片维度上，Memory Core 逻辑可以分为三部分: Load、Job（MLU Core 处理）和 Store；MLU Core 逻辑也分为三部分：Move1、Compute 和 Move2。从 MLU Core 的角度来看，M1->C->M2 构

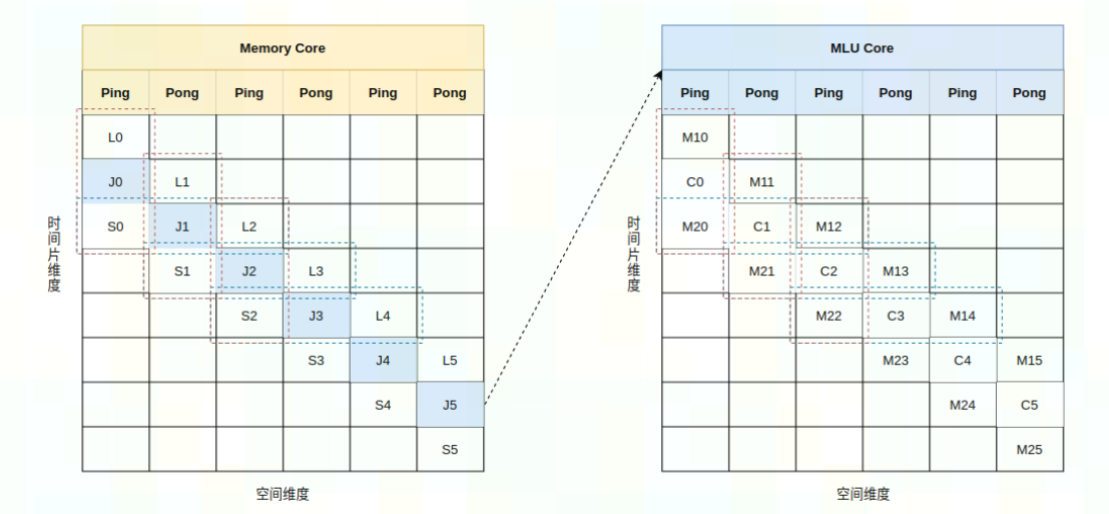

图 4.2

成了三级流水，从由 Memory Core 的角度来看，五级流水就可以理解成 L->MLU Core->S 的三级流水。以 L0->J0->S0 为例，当 L0 将数据从 GDRAM 搬运到 SRAM 上以后，通过 MLU Core 的三级流水处理将结果保存到 SRAM 上，然后由 S0 将结果搬运到 GDRAM 上。

4.3　Top-k 算法计算效率优化

基于 MLU 系列 AI 处理器的并行计算系统支持服务器级、板卡级、芯片级、Cluster 级、MLU Core 级、指令级以及数据级共七个级别的并行计算。Top-k 算子计算效率优化的过程，就是将其计算任务映射到不同的并行层次的过程。服务器级和板卡级的并行计算通常是面向 AI 训练和大规模推理任务，通过调用 CL（Communications Library，通信库）和 RT（Runtime Library，运行时库）接口实现。而其他层级的并行计算能力则是可以通过 BC 语言编程实现。例如在上个章节中设计实现的 Top-k 算子，通过使用 BC 智能编程语言提供的向量化操作指令，对计算数据进行向量化处理，大大减少了整个算法过程需要执行的指令数量和循环迭代次数，从而提升 Top-k 特征向量的求取速度，这属于数据级并行计算。指令级并行更多是通过提升访存效率来实现对算子的整体性能优化，对计算效率提升较为有限，因此可将其结合到 MLU Core 级并行优化方案中一同进行。综上所述。接下来将从 MLU Core 级并行、Cluster 级并行以及芯片级方面入手，来进行 Top-k 算子的计算效率优化工作。

4.3.1　Core 级优化

MLU-Core 内具有多条独立执行的指令流水线（以下简称为 PIPE），分别为负责标量运算的 ALU-PIPE、负责向量运算的 VFU-PIPE、负责张量运算的

TFU-PIPE、负责片上和片外数据搬移的 IO-DMA PIPE 和负责片上数据搬移的 Move-DMA PIPE。如图 **??**所示，Inst- Cache 中的一段指令序列顺序开始执行后，经过分发和调度被分配进了多个 PIPE 队列中，进入 PIPE 前指令是按顺序执行译码的，而进入不同计算或访存队列后则由硬件解决寄存器重命名或读写依赖。这意味着不同计算或者访存类型指令是可以并行执行的。

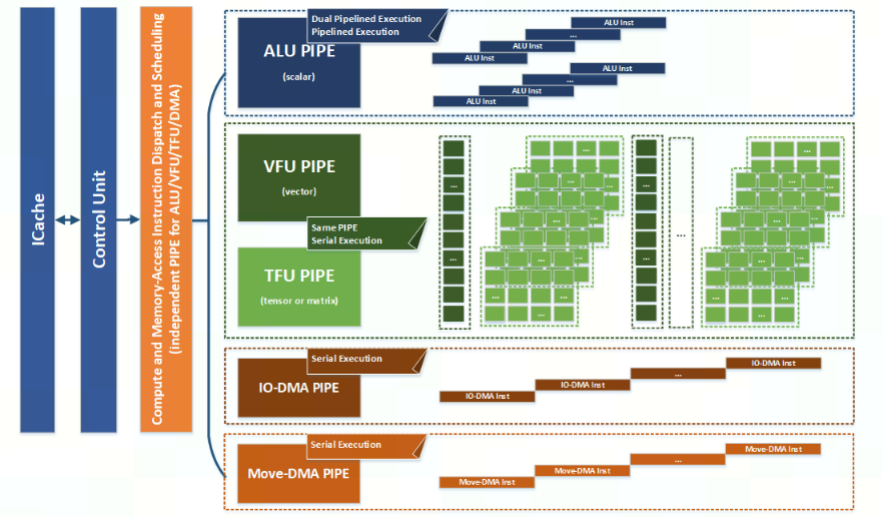

图 4.3

另外，BC 编程语言还提供了多个指令融合操作接口，这类接口不仅可以减少数据中转和 NRAM 空间占用，还可以利用硬件的指令融合功能达到更高的性能。在整个 Top-k 算子的计算流程中，先后多次使用了 bang_mul、bang_add 和 bang_sub 等向量操作指令。通过使用 bang_fusion 指令融合接口，可以实现三个输入向量的乘加、乘减、加乘、减乘、减减、加加、加减、减加运算。从而进一步减少计算过程中用到的指令数量，提升算子的性能。如图 4.4所示，使用融合指令替换原指令组合后，可以在一条指令时间内完成乘加操作。

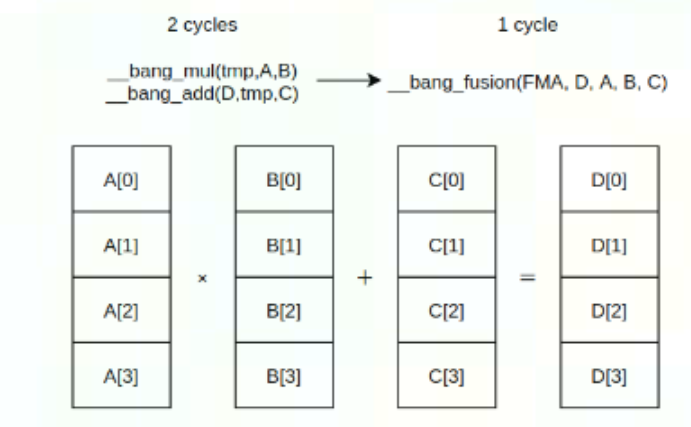

图 4.4

4.3.2　Cluster 级优化

MTP Cluster 架构如图 ref 所示，MTP Cluster 由 4 个 MLU Core 和一个 Memory Core 组成，是 MTP 架构中的最小执行单元。每个 MLU Core 是具备完整计算、IO 和控制功能的处理器核心，可以独立完成一个计算任务，也可以与其他 MLU Core 协作完成一个计算任务。

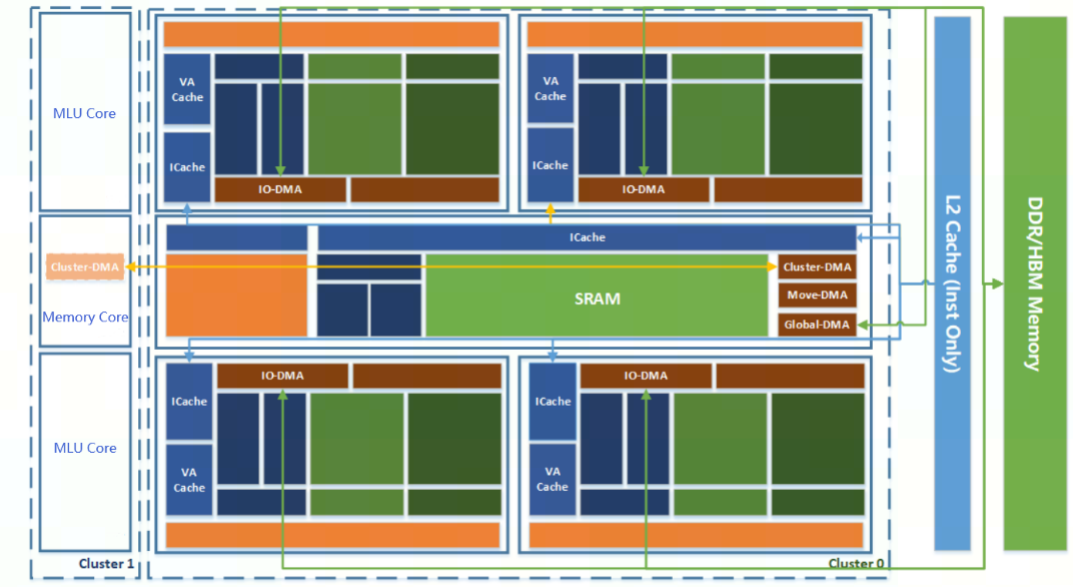

图 4.5

4.3.3　芯片级优化

MLU 芯片由本地控制单元、本地存储单元以及一个或者多个 Cluster 构成的计算单元组成。以 MLU300 系列板卡上的思元芯片为例，该芯片含有 6 个 Cluster，而设计实现的 Top-k 算子使用一个 Cluster 处理一个计算任务。MLU 芯片支持多队列并行，这意味着可以实现多个任务队列并行地执行计算任务及其 IO 操作。RT 提供了丰富且灵活的队列管理接口、异步 Kernel 执行接口和异步拷贝接口。使用这些接口可以实现硬件资源的高效利用，使得在输入图像数量较多的情况下，可以充分利用 MLU 芯片上的 Cluster 计算资源，并行地进行图像计算任务，从而提升算子的并行处理性能。实际实现上，首先读取输入待处理图像数量 n，然后用 n 除以 6，得到每个任务队列至少需要处理的输入图像数量 num_per_cluster。当 n 不能被 6 整除时，将余下的待处理输入图像从编号为 0 的队列开始，依次加入对应的队列当中，即可得到各队列需要处理的输入图像数量。再根据输入图像的大小，便可计算出每个队列需要处理的输入图像数据大小。接着使用 rtQueueCreate 接口创建好对应的队列，再通过 rtMemcpyAsync 异步数据迁移接口将各队列待处理的图像数据从主机端迁移到设备端。数据迁移完成后即可启 Kernel 核函数进行 Top-k 描述子的计算。最后再次调用 rtMemcpyAsync 异步数据迁移接口将

计算好的 Top-k 描述子从设备端迁移到主机端即可。

第 5 章　基于国产 AI 处理器的 Top-k 算法测试验证

5.1　实验环境与测试流程

5.2　Top-k 算法功能性能测试

5.3　Top-k 算法可用性测试

5.4　本章小结

第 6 章　总结与展望

6.1　本文工作总结

6.2　未来工作展望

附录 A　补 充 材 料

A.1　补充章节

补充内容。

致　　谢

在研究学习期间，我有幸得到了三位老师的教导，他们是：我的导师，中国科大 XXX 研究员，中科院 X 昆明动物所马老师以及美国犹他大学的 XXX 老师。三位深厚的学术功底，严谨的工作态度和敏锐的科学洞察力使我受益良多。衷心感谢他们多年来给予我的悉心教导和热情帮助。

感谢 XXX 老师在实验方面的指导以及教授的帮助。科大的 XXX 同学和 XXX 同学参与了部分试验工作，在此深表谢意。

在读期间发表的学术论文与取得的研究成果

已发表论文

1. A A A A A A A A A A
2. A A A A A A A A A A
3. A A A A A A A A A A

待发表论文

1. A A A A A A A A A A
2. A A A A A A A A A A
3. A A A A A A A A A A

研究报告

1. A A A A A A A A A A
2. A A A A A A A A A A
3. A A A A A A A A A A