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： 张彪

专业领域： 计算机技术

校内导师： 薛美盛 副教授

企业导师： 吕秀全 高级工程师

完成时间： 二〇二五年一月十五日

University of Science and Technology of China

A dissertation for master's degree

Design and implementation of Top-k operator based on domestic AI processorn

Author: Zhang Biao

Speciality: Computer Technology

Supervisor: Associate Prof. Xue MeiSheng

Advisor: Senior Engineer Lv XiuQuan

Finished time: January 15, 2025

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

在深度学习技术蓬勃发展的浪潮中，Top-k 查询算法已深度渗透至其各个细分领域，成为众多关键任务不可或缺的核心操作环节。然而，深度学习模型所涉及的数据规模呈指数级增长，这使得传统的 Top-k 查询算法在应对海量数据时陷入困境，其效率之低难以契合当下对快速响应与精确查询的严苛需求。在此背景下，设计具备高性能并行能力的 Top-k 查询算法，并深度融合深度学习处理器的硬件特质以优化数据处理效能，已然成为当前学术研究的前沿热点与关键方向。

DLP-M 系列处理器作为国产 AI 硬件架构的杰出代表，专为深度学习应用场景量身打造，具备指令集高度灵活、并行计算能力卓越以及硬件资源利用率出众等显著优势。遗憾的是，现有的 Top-k 查询算法大多聚焦于传统 CPU 或 GPU 环境下的优化策略，未能充分释放国产 AI 处理器所蕴含的巨大潜力。有鉴于此，本文紧密依托 DLP-M 的硬件特性，匠心独运地设计并成功实现了两种高效能的 Top-k 查询算法，并在深度学习任务的实战场景中充分验证了其卓越的性能优势与广泛的应用价值。具体而言，本文的主要学术贡献体现如下：

其一，针对大/小 k 两种差异化场景，基于 RadixSelect 算法进行深度拓展，精心设计并稳健实现了创新型的 Top-k 查询方法，并依据各场景独特需求设计了多核实现方案，有效拓宽了国产 AI 处理器上 Top-k 查询方法的边界与可能性。实证研究表明，在数据量达到较大规模时，该算法相较于原有的实现方案展现出更为优异的性能表现，在效率提升方面成效斐然。

其二，深度扎根于国产 AI 处理器体系架构，对 RadixSelect 并行算法展开全方位优化。通过充分发掘 DLP-M 处理器的硬件加速潜能与向量化指令优势，实现高效并行加速。同时，从桶操作的并行性、内存访问效率以及流水线指令等多个维度进行精细优化，使得算法吞吐量获得大幅跃升。在处理较大规模数据时，其性能表现已趋近于 NVIDIA A100 GPU，达到行业领先水平，为国产处理器在该领域的应用树立了新的标杆。

其三，着眼于深度学习模型的实际应用需求，开展功能性验证研究。为切实检验算法在真实应用场景中的效果，本文巧妙结合 Pytorch 深度学习框架，将优化后的 Top-k 查询算法深度融入神经网络并进行系统训练。实验数据显示，在涵盖百万级候选区域的大规模数据集中，该算法不仅显著加快了候选区域筛选的速度，而且在检测精度方面与基准方法维持相当水平，实现了效率与精度的完美平衡，为深度学习任务中 Top-k 查询问题的高效解决提供了创新性的解决方案与实践路径。

综上所述，本文所提出的基于 DLP-M 系列处理器的高效 Top-k 查询算法，

凭借其对硬件特性的精准把握与定制化优化策略，成功实现了深度学习任务中 Top-k 操作效率的质的飞跃，为学界与业界在高效处理海量数据的 Top-k 查询问题上开辟了崭新的视野与方向，具有重要的理论与实践意义。

关键词：高性能并行　Top-k　国产 AI 处理器

ABSTRACT

In the surging wave of the rapid development of deep learning technology, the Top-k query algorithm has deeply penetrated into various sub-fields of deep learning and has become an indispensable core operation in numerous key tasks. However, the data scale involved in deep learning models is growing exponentially, which has plunged traditional Top-k query algorithms into difficulties when dealing with massive data. Their low efficiency fails to meet the stringent requirements for rapid response and precise query in the current context. Against this backdrop, designing Top-k query algorithms with high-performance parallel capabilities and deeply integrating the hardware characteristics of deep learning processors to optimize data processing efficiency has already become the cutting-edge hotspot and key direction in current academic research.

The DLP-M series of processors, as an outstanding representative of domestic AI hardware architectures, are tailor-made for deep learning application scenarios and possess remarkable advantages such as highly flexible instruction sets, excellent parallel computing capabilities, and outstanding hardware resource utilization rates. Unfortunately, most of the existing Top-k query algorithms focus on optimization strategies in traditional CPU or GPU environments and fail to fully unleash the huge potential of domestic AI processors. In view of this, this paper closely relies on the hardware characteristics of DLP-M and ingeniously designs and successfully implements two highly efficient Top-k query algorithms. Their outstanding performance advantages and wide application values have been fully verified in practical deep learning tasks. Specifically, the main academic contributions of this paper are as follows:

Firstly, for two different scenarios of large and small k values, based on the RadixSelect algorithm, this paper conducts in-depth expansion, carefully designs and stably implements innovative Top-k query methods. Moreover, according to the unique requirements of each scenario, a multi-core implementation scheme is designed, effectively broadening the boundaries and possibilities of the Top-k query methods on domestic AI processors. Empirical studies show that when the data volume reaches a relatively large scale, this algorithm demonstrates a more excellent performance compared to the original implementation schemes, achieving remarkable results in efficiency improvement.

Secondly, this paper conducts a comprehensive optimization of the RadixSelect parallel algorithm based on the domestic AI processor architecture. By fully exploiting

the hardware acceleration potential and the advantages of vectorized instructions of the DLP-M processor, efficient parallel acceleration is achieved. Meanwhile, fine optimizations are carried out from multiple dimensions such as the parallelism of bucket operations, memory access efficiency, and pipeline instructions, resulting in a significant increase in the algorithm's throughput. When dealing with relatively large-scale data, its performance is approaching that of the NVIDIA A100 GPU, reaching the industry-leading level and setting a new benchmark for the application of domestic processors in this field.

Thirdly, focusing on the practical application requirements of deep learning models, this paper conducts functional verification research. To effectively test the effectiveness of the algorithm in real application scenarios, this paper skillfully combines the PyTorch deep learning framework, deeply integrates the optimized Top-k query algorithm into the neural network, and conducts systematic training. Experimental data show that in large-scale datasets containing millions of candidate regions, this algorithm not only significantly accelerates the screening speed of candidate regions but also maintains a comparable detection accuracy with the benchmark method, achieving a perfect balance between efficiency and accuracy. It provides an innovative solution and practical path for efficiently solving the Top-k query problem in deep learning tasks.

In conclusion, the highly efficient Top-k query algorithm based on the DLP-M series of processors proposed in this paper, relying on its precise grasp of hardware characteristics and customized optimization strategies, has successfully achieved a qualitative leap in the efficiency of the Top-k operation in deep learning tasks. It has opened up new horizons and directions for the academic and industrial communities in efficiently handling the Top-k query problem for massive data, and has important theoretical and practical significance.

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。

另外，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

Top-k 算法是一种从给定的数据集中找出前 k 个最大（或最小）元素的算法。这里的 k 是一个用户指定的正整数。例如，在一个包含 100 个整数的数组中，如果 k = 10，Top - k 算法将从这个数组中找出最大（或最小）的 10 个整数。在大数据和人工智能快速发展的背景下，Top-k 问题作为数据处理中一类重要的核心操作，广泛应用于搜索引擎、数据库管理系统、推荐系统、图神经网络等领域。如何在高性能计算环境中高效实现 Top-k 算法，尤其是并行算法，一直是国内外学术界和工业界关注的研究重点。，如搜索引擎中的结果排序（找出最相关的 k 个搜索结果）、数据挖掘中的频繁项挖掘（找出出现频率最高的 k 个项）、机器学习中的特征选择（选出对模型最重要的 k 个特征）等。

传统的 Top-k 查询算法主要基于通用处理器予以设计与实现。尽管基于通用处理器设计的算法在实现层面相对简易，然而，伴随待处理数据规模持续扩增，通用处理器已难以满足数据处理过程中对算力的需求。因此在并行 Top - K 算法领域，将 Top-k 算法高效的并行实现，一直是一个研究热点。根据实现方式，Top - k 算法可以大致分为基于排序的方法、基于选择的方法和基于概率的数据结构方法。

1. 基于排序的方法：此类别中的算法首先对整个数据集执行排序操作，随后提取前 k 个元素。典型的有完全排序法，像运用快速排序算法对整个数据集排序后选取前 k 个；还有部分排序法，例如改进的冒泡排序，仅执行 k 次冒泡过程以获取前 k 个元素。
2. 基于选择的方法：其中包含快速选择（Quickselect）算法，其与快速排序相似，不过仅针对划分后的特定部分开展递归操作，进而确定 Top - k 元素；另外还有堆（Heap）算法，借助大小为 k 的最小堆（用于查找 Top - k 小元素）或最大堆（用于查找 Top - k 大元素）来筛选出前 k 个元素。在堆算法中，先将数据集中的前 k 个元素构建成堆，接着遍历剩余元素，若元素与堆顶元素相比更符合条件（依据查找 Top - k 小或大元素而定），则替换堆顶元素并重新调整堆结构。
3. 基于概率的数据结构方法：比如采用 Bloom Filter + 计数的方式，先利用 Bloom Filter 初步甄别可能属于 Top - k 的元素，之后通过计数手段确定实际的 Top - k 元素；或者运用 Count - Min Sketch + 估计的方法，凭借对元素频率的估计来找出 Top - k 元素。这些方法在应对大规模数据时，能够凭借概率数据结构在空间利用上的高效性，以近似的方式定位 Top - k 元素。其中，Bloom Filter 是一种高效的概率数据结构，用于判断元素是否可能在集合中，其具有较低的空间复杂度；Count - Min Sketch 则主要用于频率估计，

通过特定的参数设置来平衡估计的准确性与资源消耗。

1.2.1　国外研究现状

在 Top-k 并行算法研究的早期阶段，国外学者奠定了重要的理论与算法基础。例如，Threshold Algorithm (TA) 和 No Random Access (NRA) 等算法被提出，为 Top-k 查询提供了基础框架[1]。这些算法通过设定阈值和限制随机访问等方式，初步解决了在大规模数据集中查找 Top-k 元素的基本问题，尽管其在效率和扩展性方面存在一定局限，但为后续研究提供了关键的起点。随着分布式计算技术的逐渐普及，如何在分布式环境中高效执行 Top-k 查询成为研究热点。Ilyas 等人（2003）提出了 Ranked Join 算法[2]。该算法采用剪枝策略，在分布式环境中的多个节点间对数据进行有效的筛选与连接操作。通过提前去除大量不可能成为 Top-k 结果的数据，显著减少了数据传输量和计算量，大大提高了 Top-k 查询在分布式环境中的效率，使得分布式系统在处理大规模数据的 Top-k 查询任务时具备了更强的实用性。在基于 MapReduce 的并行计算模型兴起后，Chierichetti 等人（2009）开发了一种高效的 Top-k 查询算法[3]。此算法巧妙地利用分治思想，将大规模数据集划分为多个子数据集，在各个子数据集上并行执行部分 Top-k 查询操作，然后再对各子结果进行合并与进一步筛选。这种方式有效地减少了跨节点的数据传输，充分发挥了 MapReduce 模型的并行处理优势，提高了整体查询效率，为在 MapReduce 框架下处理 Top-k 查询提供了一种经典的解决方案。近年来，基于 Spark 和 Flink 的流处理框架在大数据处理领域得到了广泛应用，Top-k 算法在动态数据流场景下的研究也取得了重要进展。Jain 等人（2016）结合增量计算技术和分布式环境，提出了一种适用于动态数据流的 Top-k 查询优化方法[4]。该方法能够实时处理不断流入的数据，在每个时间窗口内，通过增量计算快速更新 Top-k 结果，避免了对整个数据流的重新计算，有效降低了计算资源消耗，显著提升了算法在动态数据流场景下的实时性和效率，满足了诸如实时监控、网络流量分析等对数据时效性要求极高的应用需求。为了进一步提高 Top-k 并行算法的执行速度，硬件加速成为国外研究的一个重要方向。Satuluri 等人（2010）在 GPU 上实现了并行 Top-k 算法[5]。他们通过深入优化 CUDA 内核，充分挖掘 GPU 的大规模并行计算能力，对矩阵计算进行高效处理，使得算法在处理大规模矩阵数据的 Top-k 查询时，计算效率得到大幅提高。GPU 的高并行性和快速数据处理能力为 Top-k 算法提供了强大的加速支持，尤其适用于科学计算、图像处理等对计算资源要求较高的领域。Aly 等人（2015）则将目光投向了基于 FPGA 的 Top-k 查询优化[6]。FPGA 具有可灵活编程和低功耗的特点，他们针对这些特性设计了专门的 Top-k 查询电路结构，在一些低功耗场景中，如移动设备数据处理、传感器网络数据汇聚等，该方法表现出显著优势，能够在保证一定查询效率的同时，

大幅降低能耗，为在资源受限环境下的 Top-k 算法应用提供了新的思路。Wang 等人（2016）提出了异构计算框架[7]，将 CPU 与 GPU 的优势相结合，构建协同计算模型。在处理大规模数据时，该框架能够根据任务特点合理分配计算资源，让 CPU 负责控制和管理任务，GPU 专注于大规模并行计算部分，充分发挥了两种硬件平台的长处，从而获得更高的整体效率，为应对复杂多样的大数据处理任务提供了一种综合性的硬件加速解决方案。随着深度学习在人工智能领域的蓬勃发展，国外学者开始探索 Top-k 算法与深度学习的结合应用。Liu 等人（2020）研究了神经网络稀疏激活中的 Top-k 选择问题[8]。在神经网络训练过程中，通过 Top-k 算法对神经元的激活值进行筛选，保留最具影响力的前 k 个激活值，能够有效减少计算量和模型复杂度，提高训练效率。该算法在推荐系统和搜索引擎等领域得到了广泛应用，通过对用户行为数据或搜索结果的 Top-k 筛选与分析，能够更精准地为用户提供个性化推荐和搜索结果，提升用户体验和系统性能。

1.2.2　国内研究现状

国内学者针对本地化需求和实际应用场景，在分布式优化方面开展了大量研究工作。郭建生等（2007）提出了一种分布式索引方法[9]。该方法通过动态分区和负载均衡技术，根据数据的分布特点和节点的处理能力，动态地将数据划分为多个分区，并合理分配到各个节点上。这样不仅提高了节点的资源利用效率，避免了部分节点负载过重而其他节点闲置的情况，还能通过优化后的索引结构快速定位和访问数据，加速 Top-k 查询过程，提升了整个分布式系统的查询性能。张敏等（2014）结合云计算环境设计了一种基于任务调度的 Top-k 查询算法[10]。在云计算的分布式架构下，该算法充分考虑了不同任务的计算复杂度和数据依赖关系，通过合理的任务调度策略，将 Top-k 查询任务分配到多个虚拟机或容器中并行执行。同时，还采用了缓存机制和数据预取技术，减少了数据传输延迟和重复计算，有效提高了算法的扩展性和计算性能，使得云计算平台能够更高效地处理大规模数据的 Top-k 查询任务，为企业级大数据处理提供了有力支持。

针对实时流数据处理这一具有挑战性的领域，国内学者也取得了一系列成果。李明等（2018）提出了基于滑动窗口的并行 Top-k 算法[11]。在处理实时流数据时，该算法引入滑动窗口机制，将数据流划分为一个个连续的时间窗口，在每个窗口内采用并行计算的方式执行 Top-k 查询。并且，通过增量更新机制，只需对新流入窗口的数据进行处理，并与上一窗口的 Top-k 结果进行合并与调整，大大减少了重复计算，显著提升了算法的实时性，能够及时响应数据流中的变化，适用于金融交易数据监控、工业生产过程实时监测等对数据时效性要求极高的应用场景。张伟等（2019）研究了一种动态负载均衡策略[12]。在分布式流数据处理环境中，由于数据的动态性和不均匀性，容易出现数据倾斜问题，导致部分

节点负载过高而影响整体性能。该策略通过实时监测节点的负载情况和数据流量，动态地调整数据分配和任务调度，将负载较重节点上的部分任务转移到负载较轻的节点上，确保各个节点的负载相对均衡，从而提高了整个分布式系统在处理流数据的 Top-k 查询时的稳定性和效率，有效解决了因数据倾斜导致的性能瓶颈问题。

随着国产硬件技术的不断发展，国内学者积极探索 Top-k 算法在国产硬件平台上的应用与优化。王鹏等（2020）在飞腾 GPU 架构上优化了并行 Top-k 算法[13]。他们深入研究了飞腾 GPU 的硬件特性，如内存层次结构、计算单元性能等，对算法的数据结构和计算流程进行针对性优化。通过合理的矩阵分块计算策略，充分利用飞腾 GPU 的并行计算资源，提高了数据处理的并行度和内存访问效率，使得在国产飞腾 GPU 平台上的 Top-k 算法性能得到显著提升，为国产硬件在大数据处理领域的应用提供了技术支持和实践经验。赵丽等（2021）提出了一种基于 FPGA 的 Top-k 硬件加速模块[14]。该模块针对 FPGA 的可编程性和低延迟特性进行设计，通过硬件电路实现了 Top-k 算法的核心功能。在智能监控和工业物联网等低延迟、高吞吐率的应用场景中，该模块能够快速处理海量的传感器数据，实时筛选出关键信息，有效提高了系统的响应速度和数据处理能力，展示了 FPGA 在特定应用领域加速 Top-k 算法的优势和潜力。

随着数据隐私保护意识的不断增强，国内学者在分布式环境中支持加密计算的 Top-k 算法研究方面也取得了新的突破。赵磊等（2022）研究了结合差分隐私机制的分布式 Top-k 查询算法[15]。该算法在分布式系统中对数据进行加密处理，使得各个节点在进行 Top-k 查询计算时，无法获取原始数据的具体信息，从而保护了数据隐私。同时，通过差分隐私技术的巧妙运用，在保证数据隐私的前提下，仍能以较高的准确率获取 Top-k 结果，为安全计算领域的 Top-k 算法应用提供了新的方向，满足了如医疗数据共享、金融数据联合分析等对数据隐私要求严格的应用场景需求。

1.3　AI 处理器国内外发展现状

1.3.1　国外发展现状

在国外，AI 处理器的发展长期处于世界前沿水平且呈现出蓬勃发展的态势。诸多知名企业和科研机构积极投身其中，投入大量资源进行研发创新。像英伟达，其 GPU 产品在 AI 领域占据重要地位，通过持续优化架构，如从 Volta 到 Ampere 再到 Hopper 架构的演进，在计算能力、能效比等关键指标上不断取得突破，广泛应用于数据中心的大规模 AI 训练任务以及科研领域的复杂计算场景。谷歌的 TPU 系列更是专门针对 AI 任务定制开发，从最初的 TPU 到后续各代升级，在

性能提升、能耗降低方面成果显著，为谷歌自身的云计算服务、搜索引擎优化以及语音识别等众多业务的智能化升级提供了强劲动力。英特尔通过收购 Habana Labs 等战略布局，推出 Gaudi 等系列 AI 芯片，在 AI 处理器市场积极竞争，其芯片在特定应用场景下展现出独特优势，例如在一些对数据处理精度和吞吐量有特殊要求的企业级应用中表现出色。此外，在技术路线方面，除了主流的 GPU 架构，FPGA 和 ASIC 也得到深入研究与应用。FPGA 以其灵活性在一些对实时性和可重构性要求较高的 AI 推理应用中发挥作用，如工业自动化中的智能检测与控制环节。ASIC 则凭借针对特定 AI 任务的定制化优势，在如智能安防领域的图像识别芯片、语音处理领域的专用芯片等方面实现了高效能与低成本的良好平衡，有力地推动了 AI 技术在各个行业的广泛渗透与深度应用，并且随着技术的成熟和市场需求的增长，国外 AI 处理器市场规模持续扩张，形成了较为完善的产业生态和市场格局，在全球 AI 处理器市场占据主导地位并引领着行业发展趋势。

1.3.2　国内发展现状

国内 AI 处理器产业在近年来奋起直追并取得了令人瞩目的成绩。以华为海思为例，其达芬奇架构 AI 芯片展现出强大的竞争力，通过创新的异构并行计算设计理念，在性能功耗比方面达到了国际先进水平，广泛应用于华为的智能手机终端以及服务器产品，构建起全场景 AI 应用能力，为华为在全球通信与智能设备市场的竞争提供了核心技术支撑。寒武纪作为专业的 AI 芯片设计企业，推出的思元系列芯片在智能安防监控系统、数据中心的 AI 计算任务等领域得到应用，并且不断探索新技术如采用 Chiplet 技术提升芯片性能与可扩展性，在国内 AI 芯片市场占据重要份额并逐步向国际市场拓展。地平线的征程系列 AI 芯片专注于边缘计算场景，特别是在智能汽车领域，为车辆的自动驾驶辅助系统提供高效的 AI 计算能力，助力国内智能汽车产业的快速发展。在产业生态建设方面，政府积极出台一系列扶持政策，包括设立专项产业投资基金提供资金支持、给予税收优惠政策鼓励企业创新研发、推动高校与企业合作培养专业人才等，为 AI 处理器产业营造了良好的政策环境。上下游企业之间的合作也日益紧密，从芯片设计、晶圆制造、封装测试到软件算法开发以及终端应用集成，逐步形成完整的产业链条，促进了产业协同发展。然而，不可忽视的是，国内 AI 处理器产业仍面临一些挑战。在芯片设计技术的高端层面，与国外顶尖水平相比在某些复杂算法处理能力和架构优化细节上尚有差距。在制造工艺方面，由于受到高端光刻机等关键设备进口限制，芯片制程工艺的先进性难以与国外领先企业相媲美，这在一定程度上影响了芯片的性能、成本和量产规模。同时，在资金投入的持续性和人才储备的丰富度上，与国外大型企业和成熟科研体系相比略显不足，但随着国内市场

对 AI 技术需求的爆发式增长以及外部技术封锁压力下的自主创新动力增强，国内 AI 处理器产业正处于快速发展的关键时期，蕴含着巨大的发展潜力和突破机遇，有望在未来逐步缩小与国外的差距并在国际市场上占据更为重要的地位。

1.4　论文主要内容及章节安排

1.4.1　论文主要工作

本文重点研究 Top - k 算法在国产 AI 平台上的实现与性能调优后的算子的精度和性能，主要研究内容包括:

(1) 分析 Top-k 算法，并介绍国产 AI 处理器的存储模型和异构编程模型，以此作为后面研究国产 AI 处理器数据级并行化和异步并行化的基础。

(2) 针对国产 AI 处理器对 Top-k 算子进行并行实现。

(3) 通过从计算效率和访存效率两方面对 Top-k 算子进行性能调优。

(4) 对 Top-k 算子的并行化方案以及性能调优后的方案进行精度与性能上的测试，并在深度学习框架上进行适配，进行集成测试以验证 Top-k 算子的可用性。

1.4.2　论文结构

本文共六个章节，各章内容组织如下:

第一章为绪论，首先介绍本文的研究背景及意义，接着对 Top-k 算法的国内外研究现状和应用现状以及 AI 处理器的国内外发展现状进行详细介绍，最后对本文研究内容和章节安排情况进行简要概括。

第二章为相关技术背景，首先介绍了 RadixSelect 算法的原理，接着对国产 AI 处理器的编程模型和内存模型进行介绍。最后介绍了面向国产 AI 处理器的深度学习框架——pytorch。

第三章为 Top-k 算子设计与实现，本章节主要基于 RadixSelect 算法，设计了 Top-k 算子，并且根据 Top-k 算子的输入规模和国产 AI 处理器的体系结构，设计了两种不同的并行方案。本章主要分为四部分，第一部分首先介绍了算子的计算流程，在此详细讲解了 Top-k 算子工作过程中的几个重要阶段。而后根据其计算流程，详细介绍了主机端的主要任务。最后依据 RadixSelect 算法原理和国产 AI 处理器的体系结构，详细的描述了两种并行实现方案和实现过程。

第四章为 Top-k 算子的性能优化。首先介绍算子的优化策略，接着分别从计算效率和访存效率两方面，对 Top-k 算子的优化方案进行全面讲解。

第五章为实验测试与分析，本章在多核深度学习处理器上，对三四章节所实现和优化的 Top-k 算子进行全面的精度测试与性能评估，并集成到深度学习框架中，验证 Top-k 算子的可用性。

第六章为总结与展望，本章对全文的工作进行整理总结，并对本文方法中存在的不足进行分析，然后提出后续改进的方向。

第 2 章　相关技术背景

本章主要包含三个部分：1. 首先对 Top-k 算法原理进行了详细的介绍，引出了 RadixSelect 算法并阐述了选择它进行实现的理由。2. 介绍了国产 AI 处理器的编程模型和内存模型。3. 为方便后续验证 Top-k 算子的可用性，介绍了面向国产 AI 处理器的 pytorch 框架。

2.1　Top-k 算法原理

如前文所述, 在并行 Top-k 算法领域, 已有诸多值得关注的研究成果。在众多成果当中，基于选择的并行 Top-k 算法相较于全排序算法具有明显的领先优势。基于选择的并行 Top-K 算法包括 QuickSelect [9]、BucketSelect [1]、RadixSelect [1] 以及 SampleSelect [31]。其中 QuickSelect [9] 以快速排序为基础, 然而其与传统排序算法的差异在于，它仅在包含第 K 个元素的子列表上展开迭代操作。此过程通过递归方式持续进行，直至所选枢轴最终确定为第 K 个元素，并且在前 K 个元素的确定过程中，通过递归收集逐步达成目标。BucketSelect、SampleSelect 和 RadixSelect 则采用多个枢轴, 将候选元素分配至数十到数百个桶中。BucketSelect 的枢轴确定依赖于候选元素的最小值与最大值；SampleSelect 通过对一小部分元素进行采样并排序，以获取更为适宜的枢轴；RadixSelect 与最高有效位基数排序 [14] 相似，依据元素的数位特征将元素分配至相应桶中。而相较于其他三种基于选择的方法，RadixSelect 具备若干显著优势：

1. 在最坏情况下，其时间复杂度为 $O(N)$。假设在 RadixSelect 算法中，一个 r 位数字被分割为 b 位数字, 那么最多仅需 $\left\lceil \frac{r}{b} \right\rceil$ 次迭代。因此, 无论是平均情况还是最坏情况, 其时间复杂度均为 $O(\left\lceil \frac{r}{b} \right\rceil N)$。相比之下, QUICKSELECT 在最坏情形下，每次迭代仅能移除一个元素。由此，处理大约 N 个元素时，N 次迭代将导致其最坏情况时间复杂度达到 $O(N^2)$。
2. 在 GPU 上，当采用 8 位或 11 位数字（分别对应 256 个或 2048 个桶）时，RadixSelect 能展现出高度的并行性，能够更为有效地削减计算工作量。
3. BucketSelect 和 SampleSelect 在选择枢轴时均需计算输入数据的统计信息，而 RadixSelect 的枢轴选择与输入数据相互独立。

基于以上原因，本文主要基于 RadixSelect 算法对 Top-k 算子进行并行实现。RadixSelect 算法基于基数排序思想，用于在无序数组中找到第 k 小（或大）的元素，其核心步骤如下：

1. 数位分组：将元素的二进制表示按每连续 b 位划分为一个数位（digit），从

最高有效位开始处理，每次迭代处理一个数位。对于 r 位元素，共需 $\left\lceil \frac{r}{b} \right\rceil$ 次迭代。

2. 计算直方图：在每次迭代中，提取每个元素当前处理数位的 b 位并转换为对应的 digit 值（范围是 $[0, 2^b - 1]$），然后计算直方图以记录每个 digit 值出现的频率，即直方图的第 i 个计数器记录 digit 值等于 i 的元素数量。
3. 计算前缀和：对直方图计算前缀和（inclusive prefix sum），得到前缀和数组 psum。（psum[i] 表示 digit 值小于或等于 i 的元素数量）。
4. 确定目标数位：通过前缀和数组找到第 k 小元素应具有的 digit 值 j，满足 psum[j - 1] < K 且 psum[j] ≥ K。
5. 筛选元素：将 digit 值小于 j 的元素确定为当前的 Top - k 元素，存储其值和索引；digit 值大于 j 的元素可直接丢弃；digit 值等于 j 的元素作为潜在结果存储到候选缓冲区，用于下一次迭代。
6. 更新参数：每次迭代结束后，更新 k（减去已确定的 Top - k 元素数量 psum[j - 1]）和 n（更新为直方图中 digit 值等于 j 的元素数量），为下一次迭代做准备。

在算法执行过程中，通过不断迭代上述步骤，逐步缩小候选元素范围，直到找到第 k 小（或大）的元素。为了更好的说明上述步骤，下图 2.1介绍了如何从 9 个元素当中获取 Top-4 个元素。

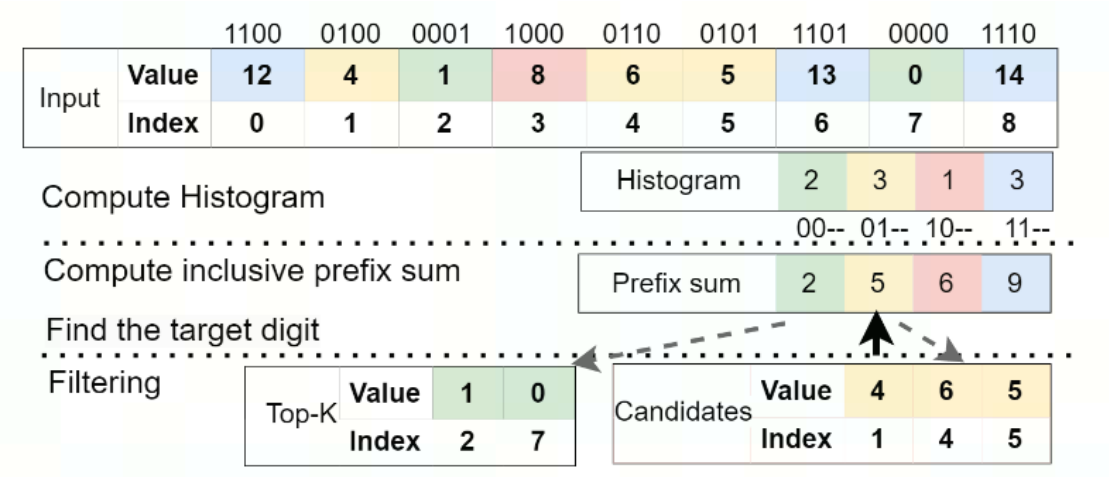

图 2.1　RadixSelect 流程图

对于 9 个 4 位元素的列表（以 2 位为一个数位），第一次迭代计算最高 2 位的直方图，而后根据直方图信息计算出前缀和，再根据第 k 小元素（k = 4）的条件确定目标数位为’01’，此时 digit 值为’00’ 的元素成为部分结果，digit 值为’01’ 的元素进入候选缓冲区用于下一次迭代，同时更新 k 和 n，继续后续迭代直至找到最终的 Top - k 元素。

2.2　国产 AI 处理器概述

本文基于寒武纪多核深度学习处理器（Deep Learning Processor-Multicore, DLP-M）实现 Top-k 算子，下面主要从体系结构，内存层次，编程模型三个方面对其进行介绍。

2.2.1　抽象硬件模型

DLP-M 硬件的基本组成单元是 MLU Core。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。每 4 个 MLU Core 核心构成一个 Cluster，另外，其每个 Cluster 内还会包含一个额外的 Memory Core 和一块被 Memory Core 和 4 个 MLU Core 共享的 SRAM（Shared RAM，共享存储单元）。Memory Core 不能执行向量和张量计算指令，只能用于 SRAM 与 DDR （Double Data Rate Synchronous Dynamic Random Access Memory，双倍速率同步动态随机存储器，DDR SDRAM 通常简称为 DDR）和 MLU Core 之间的数据传输。Cambricon BANG 异构并行计算平台对底层由 MLU 硬件构成的大规模并行计算系统进行了一系列抽象，屏蔽了具体硬件之间的细微差异，向用户展示了一个高度并行、灵活扩展和易于操控的抽象硬件模型。Cambricon BANG 异构并行编程模型由通用处理器和多个 MLU 领域专用处理器组成。其中，MLU 负责核心的大规模并行计算，而通用处理器则作为控制单元，负责复杂控制和任务调度等工作。整个抽象硬件模型分为 5 个层级：服务器级、板卡级、芯片级、处理器簇（Cluster）级和 MLU Core 级，每个层次都包括抽象的控制单元、计算单元和存储单元，如图 2.2所示。

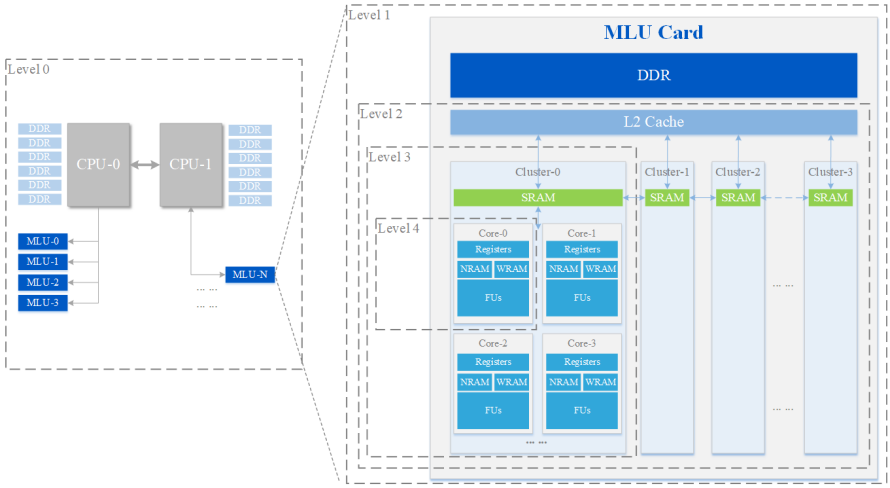

图 2.2　DLP-M 架构图

1. 第 0 级是服务器级，由多个 CPU 构成的控制单元、本地 DDR 存储单元和

多个 MLU 板卡构成的计算单元组成；

2. 第 1 级是板卡级，每个 MLU 板卡由本地控制单元、DDR 存储单元和 MLU 芯片构成的计算单元组成；
3. 第 2 级是芯片级，每个芯片由本地控制单元、本地存储单元（例如 L2 Cache）以及一个或者多个 Cluster 构成的计算单元组成；
4. 第 3 级是 Cluster 级，每个 Cluster 由本地控制单元、共享存储以及多个 MLU Core 构成的计算单元组成；
5. 第 4 级是 MLU Core 级，每个 MLU Core 由本地控制单元、私有存储单元和计算单元组成。在 MLU Core 内部支持指令级并行和数据级并行。

整个抽象硬件模型可以通过增加服务器数量、板卡数量、芯片数量、Cluster 数量或者 MLU Core 数量的方式自由扩展计算能力。本文的主要内容为算子开发工作，主要涉及到 1-4 级。

2.2.2　内存模型

抽象硬件模型提供了丰富的存储层次，包括 GPR（General Purpose Register，通用寄存器）、NRAM、WRAM、SRAM、L2 Cache、LDRAM（Local DRAM，局部 DRAM 存储单元）、GDRAM（Global DRAM，全局 DRAM 存储空间）等。GPR、WRAM 和 NRAM 是一个 MLU Core 的私有存储，Memory Core 没有私有的 WRAM 和 NRAM 存储资源。L2 Cache 是芯片的全局共享存储资源，目前主要用于缓存指令、Kernel 参数以及只读数据。LDRAM 是每个 MLU Core 和 Memory Core 的私有存储空间，其容量比 WRAM 和 NRAM 更大，主要用于解决片上存储空间不足的问题。GDRAM 是全局共享的存储资源，可以用于实现主机端与设备端的数据共享，以及计算任务之间的数据共享。抽象的硬件模型允许软件直接控制数据在各级存储之间的移动，从而高效地完成计算任务。为此，编译器对上层软件提供了丰富的地址空间声明，以及大量用于显式或隐式数据移动的机制和编程接口，以方便用户显式控制数据的存储空间。用户可以显式地控制数据在各个存储层次之间的移动，精确地控制数据搬运的时机和数据量，从而实现计算和 IO 之间的平衡，最大化计算效率。其内存层次图如图 2.3所示

1. GPR 是每个 MLU Core 和 Memory Core 私有的存储资源。MLU Core 和 Memory Core 的标量计算系统都采用精简指令集架构，所有的标量数据，无论是整型数据还是浮点数据，在参与运算之前必须先加载到 GPR。GPR 的最大位宽为 48 位，一个 GPR 可以存储一个 8bit、16bit、32bit 或者 48bit 的数据。GPR 中的数据不仅可以用来实现标量运算和控制流功能，还用于存储向量运算所需要的地址、长度和标量参数等。
2. NRAM 是每个 MLU Core 私有的片上存储空间，主要用来存放向量运算和

图 2.3　内存层次图

张量运算的输入和输出数据，也可以用于存储一些运算过程中的临时标量数据。相比 GDRAM 和 LDRAM 等片外存储空间，NRAM 有较低的访问延迟和更高的访问带宽。NRAM 的访存效率比较高但空间大小有限，而且不同硬件的 NRAM 容量不同。用户需要合理利用有限的 NRAM 存储空间，以提高程序的性能。对于频繁访问的数据，应该尽量放在 NRAM 上，仅仅当 NRAM 容量不足时，才将数据临时存储在片上的 SRAM 或者片外的 LDRAM 或者 GDRAM 上。

3. WRAM 是每个 MLU Core 私有的片上存储空间，主要用来存放卷积运算的卷积核数据。为了高效地实现卷积运算，WRAM 上的数据具有特殊的数据布局
4. SRAM 是一个 Cluster 内所有 MLU Core 和 Memory Core 都可以访问的共享存储空间。SRAM 可以用于缓存 MLU Core 的中间计算结果，实现 Cluster 内不同 MLU Core 或 Memory Core 之间的数据共享及不同 Cluster 之间的数据交互。SRAM 有较高的访存带宽，但是容量有限。用户需要合理利用有限的 SRAM 存储空间，以提高程序的性能。
5. L2 Cache 是位于片上的全局存储空间，由硬件保证一致性，目前主要用于缓存指令、Kernel 参数以及只读数据。L2 Cache 目前对于用户透明，在 Cambricon BANG 异构并行编程模型中暂时没有提供读写 L2 Cache 的接口。
6. LDRAM 是每个 MLU Core 和 Memory Core 私有的存储空间，可以用于存

储无法在片上存放的私有数据。LDRAM 属于片外存储，不同 MLU Core 和 Memory Core 之间的 LDRAM 空间互相隔离，软件可以配置其容量。与 GDRAM 相比，LDRAM 的访存性能更好，因为 LDRAM 的访存冲突比较少。

7. 与 LDRAM 类似, GDRAM 也是片外存储。位于 GDRAM 中的数据被所有的 MLU Core 和 Memory Core 共享。GDRAM 空间的作用之一是用来在主机侧与设备侧传递数据，如 Kernel 的输入、输出数据等。Cambricon BANG 异构编程模型提供了专门用于在主机侧和设备侧之间进行数据拷贝的接口。

2.2.3　编程模型

国产 AI 处理器的异构并行编程模型利用 CPU 和 MLU 协同计算，实现了 CPU 和 MLU 的优势互补。在并行编程模型中，CPU 作为主机侧的控制设备，用于完成复杂的控制和任务调度；而设备侧的 MLU 则用于大规模并行计算和领域相关的计算任务。执行的程序称作 Kernel 函数（核函数），在核函数被执行之前，往往需要根据任务规模来决定需要多少个 Job 进行处理。

在 MLU 上被调度的最小单位被称为 Job（作业），其被调度到 Cluster 执行（需要满足一定的规则才可以被调度），Job 之间一般也是逻辑独立的。一个 Job 至少包含一个 Task，具体视类型而定。因此被执行的最小单位为 Task（任务），其被调度到具体的 MLU-Core 上执行，执行一遍核函数中的程序代码。

而 Job 又可以分为不同的类型，其代表着此 Job 所需要的硬件资源数量，即一个 Job 在实际执行时会启动多少个物理 MLU Core 或者 Cluster。在其并行编程模型中支持两种任务类型：Block 类型和 UnionX 类型。

1. Block: 代表一个 Job 在执行时至少需要占用一个 MLU Core。对于 Block 类型的 Job，不支持共享 SRAM，不支持不同 Cluster 之间的通信。Block 类型的 Job 是所有 MLU-Core 硬件都支持的类型。当 Job 规模大于 1 时，由计算平台根据硬件资源占用情况决定所有 Job 占用的 MLU Core 数量：如果只有一个 MLU Core 可用，那么所有 Job 在同一个 MLU Core 上串行执行；如果有多个物理 MLU Core 可用，那么所有 Job 会被平均分配到所有可用的 MLU Core 上分批次执行。
2. UnionX: 表示一个 Job 在执行时至少需要占用 X 个 Cluster，其中，X 必须为 2 的整数次幂。一个拥有 M 个 Cluster 的板卡，X 的最大值为 $2^{\lfloor \log_2 M \rfloor}$。MLU-Core 对 Union 类型 Job 的支持与硬件的具体配置有关。例如，一些终端侧或者边缘侧的单核设备不支持 Union 类型，而一个拥有 8 个 Cluster 的硬件只能够支持 Union1、Union2、Union4 和 Union8 类型的 Job，无法支持 Union16 类型的 Job。

2.3　面向深度学习的处理器的 pytorch 框架

为了验证 Pytorch 为支持国产 AI 处理器，寒武纪定制了开源人工智能编程框架 PyTorch（以下简称 Cambricon PyTorch）。Cambricon PyTorch 借助 PyTorch 自身提供的设备扩展接口将 MLU 后端库中所包含的算子操作动态注册到 PyTorch 中，MLU 后端库可处理 MLU 上的张量和算子的运算。Cambricon PyTorch 会基于 Cambricon CNNL 库在 MLU 后端实现一些常用算子，并完成一些数据拷贝。为了能在 Torch 模块方便使用 MLU 设备，Cambricon PyTorch 在 PyTorch 后端进行了以下扩展：

1. 通过 Torch 模块可调用 MLU 后端支持的网络运算。
2. 对 MLU 暂不支持的算子，支持该类算子自动切换到 CPU 上运行。
3. Torch 模块中与 MLU 相关的接口的语义与 CPU 和 GPU 的接口语义保持一致。

为便于进行 PyTorch 和基于深度学习处理器的 Top-k 实现关系的理解，此处对国产 AI 处理器的整体软件栈结构进行介绍。在其软件栈中，如下图4.7所示。PyTorch 等通过调用 Cambricon CNNL 算子接口，实现对应的框架层算子在 MLU 上的高性能加速计算。而 Cambricon CNNL 作为一个网络运算库，提供了人工智能计算所需要的算子接口。Cambricon CNNL 算子的计算通过软件栈底层 CNRT（Cambricon Runtime Library，寒武纪运行时库）和 CNDrv 接口（Cambricon Driver API，寒武纪驱动 API）完成与寒武纪 MLU 设备底层驱动的交互和计算任务。

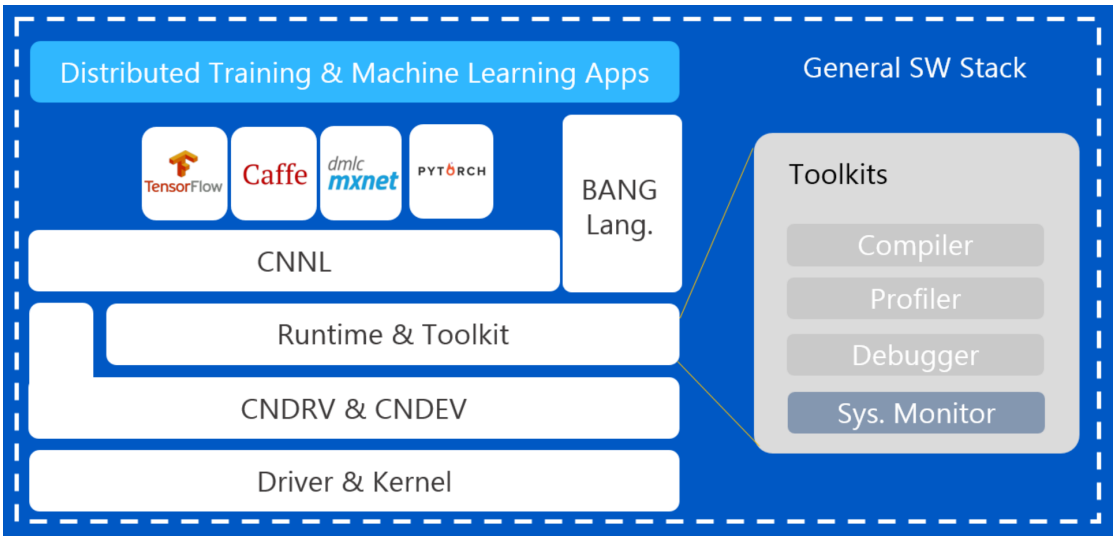

图 2.4　软件栈示意图

2.4　本章小结

本章首先详细阐述了 RadixSelect 算法的基本原理，为后续在国产 AI 处理器上实现高效的 Top-k 算子奠定了理论基础。随后，深入介绍了国产 AI 处理器的硬件架构，并从抽象硬件模型，内存模型和编程模型三个层面，详尽地介绍了基于国产 AI 处理器的异构计算平台，为后续如何对 Top-k 算子进行实现和性能优化指明了方向。最后介绍了面向国产 AI 处理器的 Pytorch 框架，为后续验证 Top-k 算子可用性提供了方案参考。

第 3 章　基于国产 AI 处理器的 Top-k 算子设计实现

尽管在国产 AI 处理器架构上已经实现了基于改进的冒泡排序算法的 Top-k 算子，但是当数据规模较大的时候，其性能仍然存在一定的不足，并且维护成本相对较高。因此本文结合国产 AI 处理器的异构计算特点，根据 RadixSelect 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，伴随大数据的迅猛发展与人工智能技术的持续演进，基于国产 AI 芯片的 Top - k 算子计算流程已逐步演化为一种典型的异构计算模式，此模式通常需主机端（如 CPU）与设备端（如 AI 处理器）协同运作。整个计算流程可划分为三个主要阶段：数据准备阶段、数据传输阶段以及核心计算阶段。

1. 数据准备阶段（于主机端执行）在 Top - k 算子的整个计算进程中，数据准备阶段起着至关重要的作用，此阶段通常由主机端承担主要职责。在该阶段，主机端（一般为 CPU）负责接收输入数据，并对其进行格式转换以及初步的数据清洗操作。鉴于 AI 处理器往往是针对特定任务进行优化的硬件，其处理能力与 CPU 存在显著差异，故而需对输入数据进行适配处理，以保障数据能够契合后续硬件计算的要求。

此外，在此阶段，主机端还需对输入数据实施解析与验证操作，旨在确保数据的合法性与有效性。具体工作涵盖检查输入数据的维度、类型等，以保证数据格式与设备端的预期相匹配。在实际应用场景中，此阶段的计算量相对而言较小，然而却是整个计算流程的关键环节，因为若数据准备工作出现偏差，后续计算将无法顺利推进。

2. 数据传输阶段（从主机端向设备端传递数据）完成数据准备工作后，后续步骤是将经过处理的数据从主机端传输至设备端（AI 处理器）。由于 AI 处理器在处理大规模数据时具备卓越的并行计算优势，此阶段的核心在于将数据高效地传递至设备端，并确保数据在设备端能够进行高效计算。在这一阶段，传输过程的效率直接关乎整体计算性能。倘若数据传输时间过长，将会对整个系统的响应时间产生不利影响。因此，主机端需选取适宜的传输方式，以降低数据传输延迟，确保数据能够尽快抵达设备端。

3. 核心计算阶段（于设备端执行）核心计算阶段由设备端（AI 处理器）负

责完成，其主要任务是依据 Top - k 算子的原理，借助设备端的硬件特性开展高效的并行计算。在传统的 CPU 环境中，Top - k 查询算法通常依赖于单线程或少量线程进行处理，计算效率较低。而在 AI 处理器中，凭借硬件的向量化计算指令，能够在多个计算单元上并行执行相同操作，从而大幅提升计算速度。故而，设备端的高效计算构成了整个 Top - k 查询计算流程的核心所在。

通过上述分工，充分发挥了 AI 处理器的硬件优势，显著提高了 Top-k 算法计算效率。在大规模数据处理场景中，该计算流程能够快速筛选出前 k 个最大（或最小）元素，满足数据处理需求。在深度学习领域，可用于模型训练中的数据筛选与特征提取等操作，加速模型训练和推理进程。图 3.1展示了本文 Top-k 算子的计算流程。

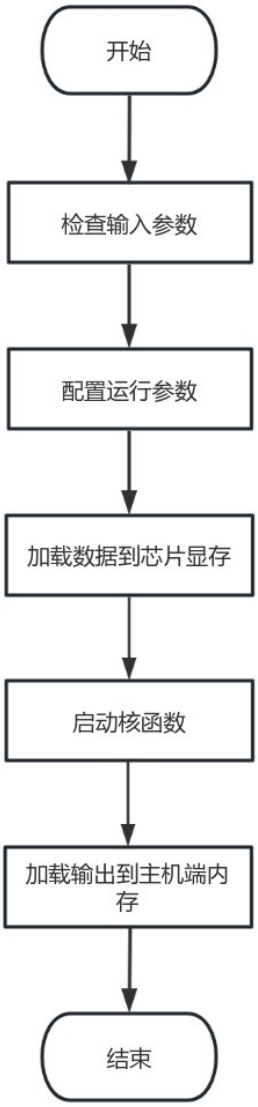

图 3.1　算子整体流程

3.2　主机端任务分析

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点数，支持不同的精度（int32 为整数，float32 为浮点数）。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 sort，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。

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
sort	bool	默认为 true，控制是否需要排序

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。

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

3.2.2　数据处理

（1） 设备端数据摆布方式分析

Top-k 算子需要支持多维度数据，为方便描述，假设其有四个维度，表示如下：dim0，dim1，dim2，dim3。由左到右表示维度从高到低，按照参数 dim 可以将多维度张量（此时为四个维度）转化为三个维度，设其为：left，dim，right，其

中，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; 当 0 < dim < 3 时，表示对 dim 求 Top - K，则此时 left 为 dim 前的维度的乘积，right 为 dim 后维度的乘积。

也就是说，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 核函数而言，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

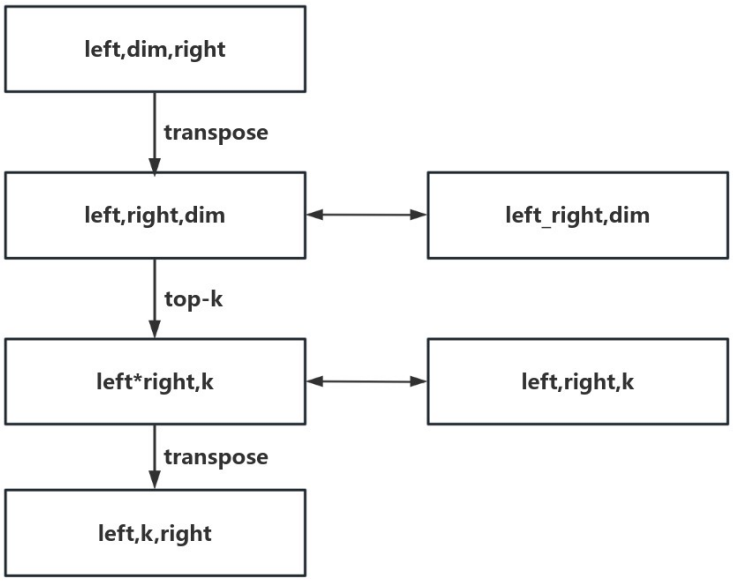

图 3.2　数据维度变化流程图

3.2.3　核函数配置

核函数的配置主要包括三个部分：即 kernel 函数的选取，Job 类型的选取和 Job 数量的确定。

（1）　核函数选取

本文基于 RadixSelect 算法和国产 AI 处理器的体系结构，实现了两个 Top-k 核函数，分别处理不同场景下的任务。在国产 AI 处理器的异构计算架构下，根据不同规模的输入合理选取核函数具有至关重要的意义。

首先，不同规模的输入对系统资源（如内存、计算单元等）的需求不同。在

处理大规模输入时，合适的核函数能够合理分配和管理内存资源，确保数据在计算过程中的高效存储和访问。例如，一些针对大规模数据优化的核函数可能采用特定的数据存储格式和访问模式，以减少内存带宽压力，提高数据读取速度，从而充分利用内存资源。同时，在计算单元利用方面，能够根据输入规模动态调整计算单元的使用，避免资源闲置或过度占用。对于小规模输入，选择适配的核函数可以避免资源浪费。若使用针对大规模输入设计的核函数处理小规模数据，可能会导致计算单元利用率低下，浪费宝贵的硬件资源。另外，在实际应用中，输入规模往往是动态变化的。能够根据不同规模输入灵活选取核函数，使得计算系统具有更好的适应性和可扩展性。无论是面对数据量不断增长的趋势，还是在处理不同规模数据混合的复杂场景中，系统都可以通过选择合适的核函数来应对，从而在不同的应用场景和数据环境下保持良好的性能表现，为系统的长期发展和广泛应用提供坚实基础。

（2） Job 数量确定

对于 Job 数量而言，这同样涉及到多个方面的重要考量，对于充分发挥国产 AI 处理器的计算能力和优化程序性能至关重要。不同的应用程序具有不同的数据规模和计算复杂度。确定合适的 Job 数量可以根据具体的计算任务需求，将数据和计算负载合理分配到各个 Job 中。对于大规模数据处理任务，可能需要较多的 Job 来并行处理数据，以加速计算过程；而对于相对较小规模的任务，过多的 Job 可能会导致线程管理开销增加，反而降低性能。因此，根据数据规模和计算需求确定 Job 数量能够实现计算资源的有效利用，提高计算效率。

通过前文介绍，我们知道 Top-k 算子能够处理多维度张量，但是通过一定的数据预处理操作，对于 Top-k 算子的核函数而言，其所需要处理的输入是一个二维张量，并且是在第二个维度提取 Top-k 个最大/最小元素。

为了方便描述，我们暂且假设问题为：设其核函数输入为 input，input_shape = [A, B]（在 B 维度进行操作）。输出为 output，output_shape=[A,k]。

其中求取最小的 Top-k 个值。input 的数据类型为 unsigned_int（无符号整型）。

我们可以发现，输入数据的每一行之间相互独立，因此我们可以根据行数来确定 Job 的数量，即启动 A 个 Job，每个 Job 处理一行数据，获取此行数据的 k 个最小值。Job 与数据的对应关系可以通过 Job 的索引来进行确定。因此我们可以将问题进一步聚焦为仅仅只是对于一个长度为 B 的数组来求取 Top-k 个最小的值。

（3） Job 类型选取

在前述内容中提及，国产 AI 处理器以 Task 作为最小执行单位，以 Job 作为最小调度单位，且 Job 存在多种类型。不同的类型通常意味着不同的并行粒度以及 Task 之间独特的协同方式。

就 Block 类型的 Job 而言，其调度条件相对宽松，仅需一个 MLU - Core 处于空闲状态即可被调度执行。在执行过程中，Job 之间不能进行通信，而此时 Job 仅仅只有一个 Task，因此各个 Task 之间也是逻辑独立的。这种独立性体现在 MLU-Core 执行期间，无需进行通信交互，每个 Task 能够自主完成自身计算任务而不依赖于其他 Task 的执行状态或中间结果。与之不同的是 UnionX 任务，其调度要求更为严苛，需要多个 MLU - Core 同时处于空闲状态方能启动调度。在执行期间，各个 Task 之间存在复杂的协作关系，具体表现为线程之间需要进行同步操作，以确保任务执行的先后顺序和逻辑正确性。此外，在涉及数据访问时，还需在 GDR（全局数据存储）或 SMEM（共享内存）上严格维护数据一致性，保证各个 Task 对数据的读写操作不会产生冲突和错误结果。

因此，Job 类型决定了 Task 的数量，而对于一个 Job 而言，往往数据规模越大，所需要的 Task 数量往往越多。因此，对于 Top-k 问题而言，我们可以根据具体实现细节，B 的大小和 MLU-Core 的资源配置情况来确定具体的任务类型。而这里的资源配置指国产 AI 处理器的各个层级的内存容量，具体信息如表 3.3所示:

表 3.3　内存容量表

参数名称	数据类型
NRAM / MLU-Core	512KB
WRAM / MLU-Core	512KB
SRAM / Cluster	2MB
GDRAM	80G

3.3　设备端并行算法设计与实现

3.3.1　Top-k 整体流程设计分析

基数选择（RadixSelect）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 大元素的数进行运算，从而达到减少问题规模的效果，另外其输出结果并不是有序的，因此我们需要对 RadixSelect 的输出结果根据参数 sort 来判断是否有必要进行排序。另外，考虑到国产 AI 处理器的 I/O 特点，我们需要根据 dim 与输入的维度信息关系，来判断是否有必要对输入/输出进行转置操作。因此，设备端的计算流程如下图 3.3 所示:

对于 Transpose 核函数和 Sort 核函数, 可以选用国产 AI 平台现阶段已开发好的核函数进行计算; 对于 RadixSelect 核函数，其主要依托上述提到的 RadixSelect 算法并结合国产 AI 处理器的体系结构进行并行设计实现。

在 RadixSelect 算法中，一个数位（digit）对应着一个元素二进制表示中的一

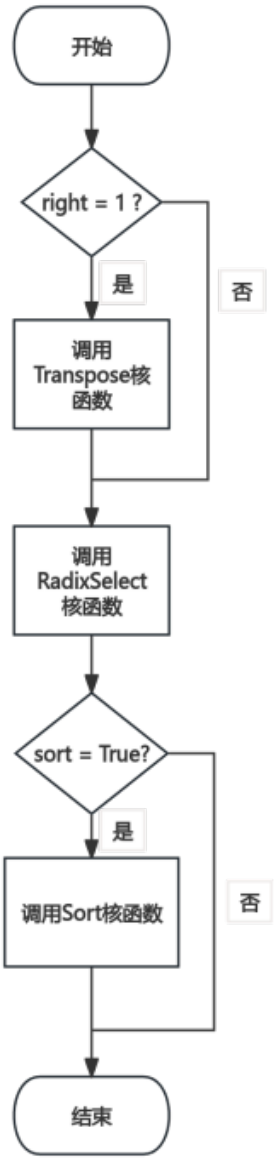

图 3.3　设备端整体流程图

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，设为 CountingSort，其主要包含四步：1. 求取直方图（HISTOGRAM-KEYS）2. 扫描桶（SCAN-BUCKETS）3. 找到目标桶（FIND-TARGET-BUCKET）4. 目标元素筛选（FILTER）其伪代码见算法 **??**。

基于基数排序（Radix Sort）思想的 RadixSelect 算法，在处理大规模数据时，通过按位分组和并行处理，能有效减少计算时间。特别是在国产 AI 处理器等并行计算设备上，RadixSelect 算法可以充分利用设备的并行计算能力，将数据分配到多个计算单元上同时处理。目前 RadixSelect 算法实现过程中，应当解决的主要问题为以下两个方面：

Algorithm 1 CountingSort Algorithm

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

```
1  Bucket ← init 0 // Initialize buckets
   // HISTOGRAM - KEYS
2  for j ← 0 to N − 1 do
3      Bucket[D[j]] ← Bucket[D[j]] + 1
4  end
   // SCAN - BUCKETS
5  Sum ← 0
    for i ← 0 to 2^r − 1 do
6      Val ← Bucket[i]
       Bucket[i] ← Sum
       Sum ← Sum + Val
7  end
   // FIND TARGET BUCKET
8  for i ← 0 to 2^r − 1 do
9      if Bucket[i] ⩾ k then
10         POS ← Bucket[i]
           break
11     end
12 end
   // FILTER
13 for j ← 0 to N − 1 do
14     if D[j] ⩾ POS then
15         continue
16     end
17     A ← Bucket[D[j]]
       R[A] ← K[j]
       Bucket[D[j]] ← A + 1
18 end
```

1. 并行方案: Top-k 的问题适用场景极为宽泛，不同的输入往往对应着不同的实现。在较大数据规模下，对于较大的 k 值，由于片上内存有限，需要将中间计算结果保存在片外空间上。而对于较小的 k 值，可以将 Top-k 元素常驻在片上内存中，快速响应后续的比较和更新操作，减少对内存的访问延迟。因此，本文根据 k 值与片上空间的大小关系进行了两种并行方案的设计与实现，具体见 3.3.2 和 3.3.3。
2. 数据类型和正负性: RadixSelect 需要从最高位来获取输入数据的数位（digit），需要数据满足“位数越高，数值大小的权重越高”这一规律。然而各个类型数据在计算机内存中的存储形式，并不是按照这一形式进行存放的。例如对于整型数据而言，负数的数值越大，其对应的二进制越小。因此需要根据不同的数据类型对输入数据进行预处理操作，使其满足算法要求。同样的，对于大部分有符号数据类型而言，其在内存中的存储形式同样违反“位数越高，数值大小的权重越高”这一规律。因此需要根据不同数据类型的正负数的存储规则对输入数据进行预处理操作。具体分析见 3.3.4。

3.3.2　小 k 场景下的 RadixSelect 的并行设计及实现

为了方便问题的描述，假设输入为无符号整型数据（unsigned）且求取 Top-k 小元素。考虑到此时 k 值较小，中间结果可存放在 NRAM 空间上。根据前文的分析，对于 input-shape = [A,B]，我们可以根据 A 的大小来确定 Job 的数量，单个 Job 处理输入的一行，各个 Job 负责的数据范围，可以根据 TaskIdx，TaskIdy 和 TaskDim 等内置变量进行数据与 Task 之间的映射。在对更复杂的场景进行分析之前，我们在此场景下，先进行单核版本的 RadixSelect 算法的设计和实现。即假设此时单个 MLU-Core 可以一次性放下 input 的一行，即 B 此时相对较小，Job 类型设置为 Block 即可。其单核实现如下：

（1） 单核实现

1. 初始化

   首先，需要初始化一个名为 $\boldsymbol{Buckets}$ 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 $\boldsymbol{Buckets}$ 数组的主要作用是用于统计数组 $\boldsymbol{D}$ 中每个元素出现的次数。初始化变量 bit_mask=0xF0000000，此变量用于获取输入中各个元素数位，需要根据当前迭代论次进行调整右移。初始化数组 masks，数组长度与输入长度保持一致，此数组是为了辅助数据过滤。

2. 构建直方图

   使用 bang_band_scalar 向量指令获取当前迭代论次中各数据的数位存于 $\boldsymbol{D}[j]$ 中。而后利用 bang_histogram 向量指令，遍历输入数组 $\boldsymbol{D}$ 中的每一个元素，统计 $\boldsymbol{D}$ 中各元素出现的个数，记录在 $\boldsymbol{Buckets}$ 中。

3. 扫描桶 遍历 Buckets 数组，计算出每个桶的累积和，由于桶的数量较小，由于使用标量串行运算，因此此步骤可能会成为即性能瓶颈，但可以通过将 digit 设置为较小的值进行避免。

4. 找到目标桶

   再次遍历 $\boldsymbol{Bucket}$ 数组，目的是找到第一个满足 $\boldsymbol{Bucket}[i] \leqslant k$ 且 $\boldsymbol{Bucket}[i+1] \geqslant k$ 的桶。将桶的下标 i 使用 key_bin 记录下来。

5. 过滤

   在此阶段，会再次遍历数组 $\boldsymbol{D}$，目的是将数组 $\boldsymbol{D}$ 中的前 k 大元素挑选出来，并放入结果数组 $\boldsymbol{R}$。首先，使用向量指令 bang_le 将小于 key_bin 的 D 元素使用 mask 记录下来。而后使用向量指令 bang_filter 指令利用 input 和 mask 数组将数据过滤出来，保存在最终结果 output 中。然后，使用向量指令 bang_eq 将等于 key_bin 的 D 元素使用 mask 记录下来，同时更新 k 值。而后使用向量指令 bang_filter 指令利用 input 和 mask 数组将数据过滤出来，

保存在 input 中。作为下一轮次迭代的输入。

单核 Top-k 算子适配于处理中等或较小的数据规模，并具备处理多维数据的功能，这使其在网络模型数据处理中尤为有效。然而，该算子并未直接为多核并行处理提供内置逻辑。对于较大的数据规模，需要配合其他特定算子实施处理策略。具体操作流程包括：在执行 Top-k 算子之前，首先应用基于深度学习处理器的 Split 算子，对数据集进行适当划分，以适配至各个单独的处理核心，期望数据能够被单核心处理。而后每个核心独立完成指定的 Top-k 计算任务后，应用 Concat 算子将各核返回的数据集成为最终结果。此步骤完成后，系统可继续执行随后的数据传输与处理操作。这一处理策略确保了在维持计算效率的同时，能够对大规模数据集进行有效管理和运算。单核在多核运行环境下的执行顺序如下图3.4所示。

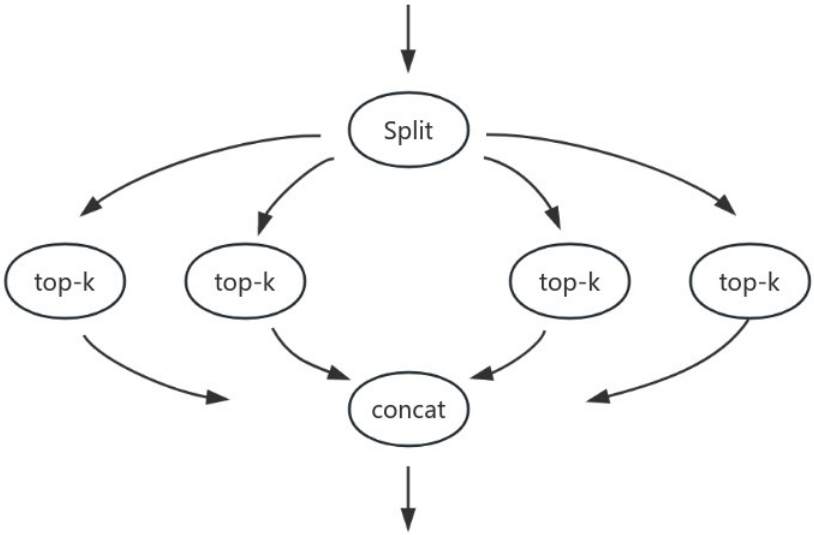

图 3.4　多核拆分示意图

Split 和 Concat 操作是 I/O 类型算子，也就是说单核 RadixSelect 算法的实现方案在多核上执行，势必会引入 Split 和 Concat 的额外时间开销，在某些极端的场景下，这可能会成为性能瓶颈。

（2） 多核实现

由于单核 RadixSelect 的整个操作都是在 NRAM 空间上进行, 需要其输入, 输出和各种中间变量的内存容量大小总和不能超过 NRAM 的空间大小（即 512kb），所以对 B 的大小要求比较严格（数量级在 10^5）。而若是想处理更大规模的数据（B 较大时），则需要在深度学习框架侧调用额外的算子去配合使用。但是这又会增加额外的时间开销，导致算子的整体性能较差。因此针对小 k 场景，设计多核 RadixSelect 是十分必要的。

大规模数据处理方法之一是充分发挥深度学习处理器的多核优势，减轻多核拆分后单核数据处理的负担。首先，依据处理器核心数量 *corenum*，按照数据规模对查询工作进行划分，从而确定每个核心所需处理的数据量。每个核心对划分后的数据执行 Top-k 查询，会返回 k 个数。多核完成 Top-k 查询后，返回的临时数据总量为 $corenum \times k \times 2$。这里的系数 2 是因为包含了返回的 k 个数据及其对应的下标。之后，对总量为 $corenum \times k \times 2$ 的数据进行 Top-k 归并操作。若数据量仍然较大，则再次采用多核 Top-k 方法处理；若数据量不大，则直接用单核 Top-k 进行归并操作，最终完成整个 Top-k 查询。需要注意的是，临时数据需要在片外进行缓存，缓存内容为每个核心查询返回的 Top-k 值及其对应的下标。对应的操作流程可概括为图3.5所示。

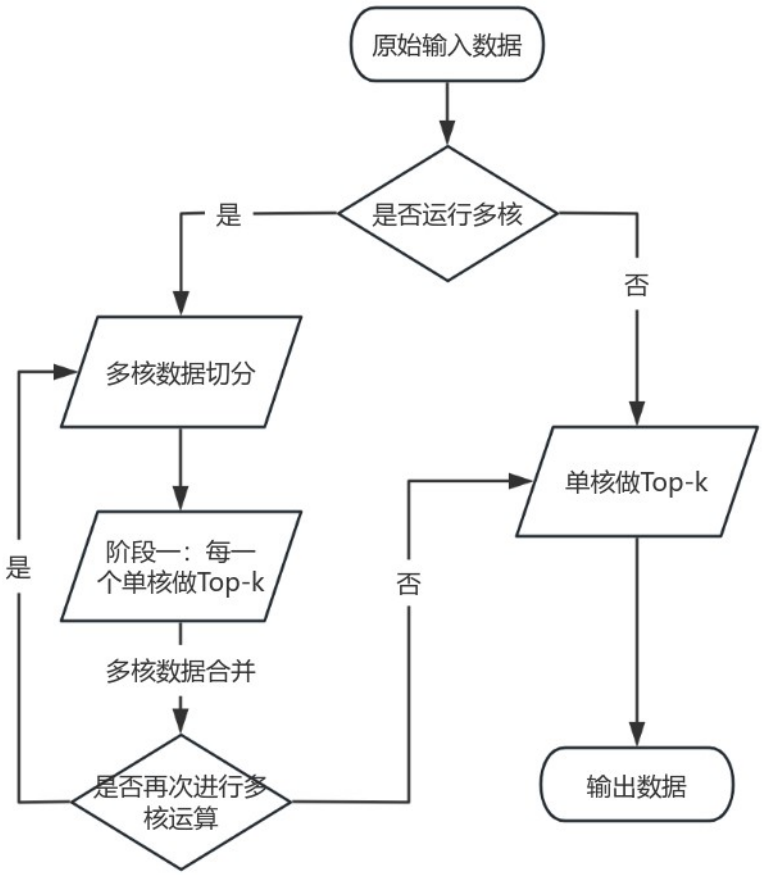

图 3.5　Top-k 多核实现流程图

在核心数量和数据量大小的综合影响下，我们对 Top-k 查询性能进行了详细分析。对于较小的数据量，如果使用多核 Top-k 查询并不能带来明显的性能提升，则推荐使用单核 Top-k 进行查询。相反，对于较大的数据量，应该在多个核心上分配查询任务，使每个核心分别完成自己的 Top-k 查询，并根据剩余数据量的大小决定是否需要进一步的查询迭代。

例如，在一个拥有 48 个处理核心的系统中处理超过 960 万条数据，目标是获取 1 万个 Top-k 数据的情形中，尽管每个核心完成了初次 Top-k 筛选，但总体上仍有大量数据需要进一步处理。因此，每次多核查询后，都需要精确地调整任务类型和核心数量，以确保数据处理的高效性。

具体地，设需要的核心数为 Corenum，单个核心可以处理的数据量为 num，则有:

$$\text{Corenum} = \left\lceil \frac{B-1}{\text{num}} + 1 \right\rceil,$$

其中 B 为总数据量。在 UnionX 结构中的 X 值由下式给出，Union0 视为 Block:

$$X = \begin{cases} 2^{\lfloor \log_2(\text{Corenum}-1) \rfloor - 1} & \text{when Corenum} > 2, \\ 1 & \text{when Corenum} = 2, \\ 0 & \text{when Corenum} = 1. \end{cases}$$

值得注意的是，由于硬件的限制，大规模数据查询后，需要将计算结果暂存到外部存储中，为后续的查询过程服务。这种频繁的数据交换和多次核心启动虽然确保了处理大规模数据的能力，但也相应地增加了时间成本。

3.3.3　大 k 场景下的 RadixSelect 的并行设计及实现

前面所介绍的各个实现方案，充分利用了“小 k”场景下的一个重要特性——NRAM 空间能够存放完整的临时输出。而由于 NRAM 的空间十分有限，因此 k 的范围不能够太大，否则将会导致程序崩溃。因此，针对大 k 场景下的 RadixSelect，必须考虑将临时输出存储在 GDR 空间上。

回顾 RadixSelect 的串行算法，由于其所有阶段都存在循环依赖关系，所以不能直接进行并行化（或向量化）。例如，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 3.6所示。

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-

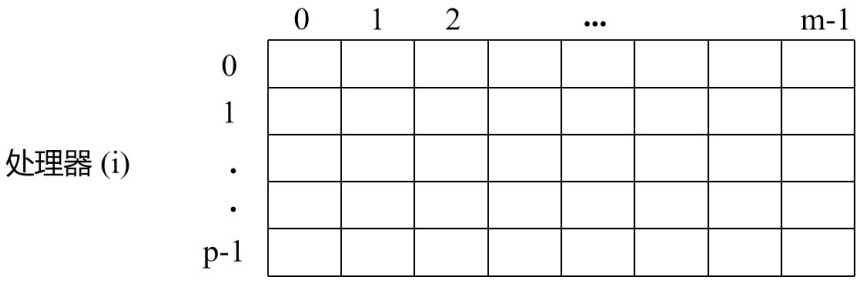

图 3.6　处理器组织方式示意图

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 3.7。

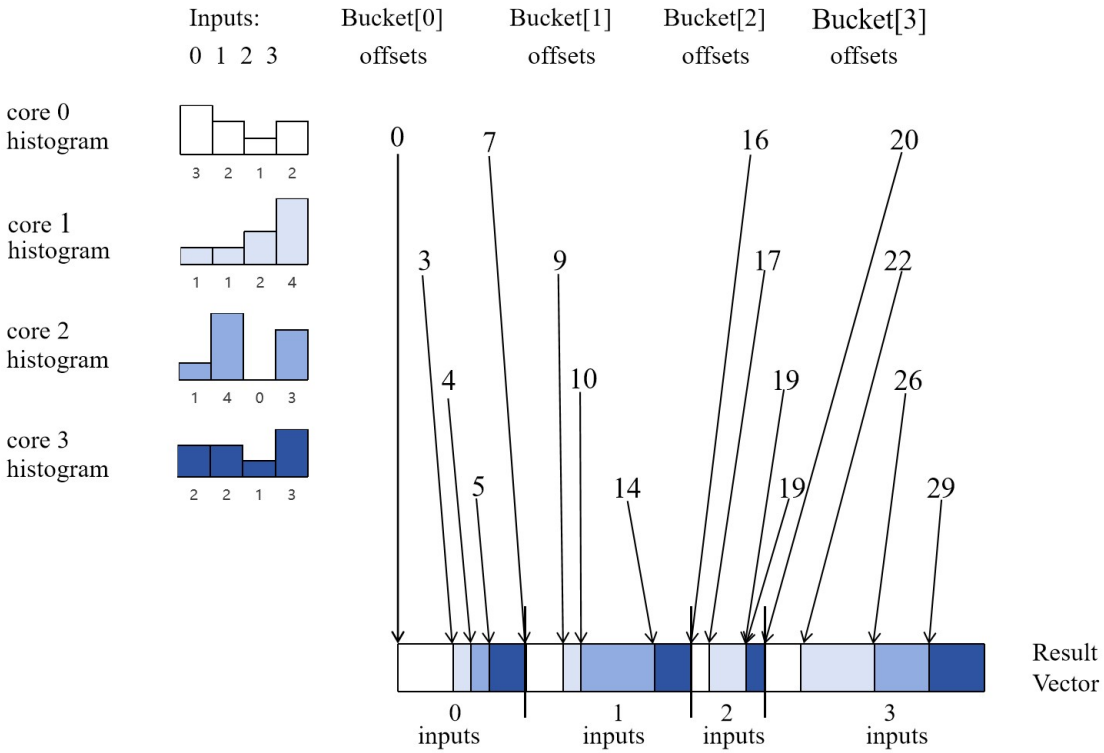

图 3.7　并行 RadixSelect 的扫描步骤

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 Job 负责数据的若干行，而后每个 Job 对所分配的行依次处理。
2. 对于 Job 正在处理的某一行数据而言，由于数据量较大，因此采用多核进行计算（即此时 Job 类型为 UnionX）。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，需要在 GDR 上初始化一个名为 $Buckets$ 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 $Buckets$ 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

为了节省片上内存空间，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 $Buckets\_local$（处在 NRAM 空间）中。在处理器处理完所有的局部数据后，将 $Buckets\_local$ 复制到片外内存 $Buckets$ 中。

③ 扫描桶

此步骤主要是计算出每个桶的累积和。已知共有 $2^{digit} \times processor$ 个桶，则每个 MLU-Core 处理 $2^{digit} \times processor$ 个桶，算出来局部前缀和后保存在 P_local

（大小为 processor）中，再使用 MLU-Core0 对 P_local 求取前缀和保存在 P_local 中。而后每个 MLU-Core 再将负责的 Buckets 的部分加上对应的 P_local 的值获得全局前缀信息。

④ 找到目标桶

让 MLU-Core0 再次遍历 *Buckets* 数组，目的是找到第一个满足 $Buckets[i] \geqslant k$ 的桶。在此阶段，主要使用 bang_gt_cycle 和 bang_filter 向量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 *input* 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 *input* 中的数据筛选出来，再根据 Buckets 中的偏移值（此时以涵盖全局信息）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，作为下一轮次的输入数组。

3.3.4　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 *int*32 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 3.8：

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	0xFFFFFFFE	0x00000001	0x00000000	0xFFFF FFB	0x00000003	0xFFFFFFFF	0x00000002
二进制表示	11111111 11111111 11111111 11111110	00000000 00000000 00000000 00000001	00000000 00000000 00000000 00000000	11111111 11111111 11111111 11111011	00000000 00000000 00000000 00000011	11111111 11111111 11111111 11111111	00000000 00000000 00000000 00000010

图 3.8　整型输入数据示意图

原始数据经过基数排序后为 data1, 其表示见下图 3.9: :

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 3.9　整型数据未处理排序结果示意图

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为 bang_bxor_scalar 和 bang_mlu 即逐个比特位取反，通过上述处理之后，数据将会正常显示。如图 3.14所示。

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 3.10　整型正确输出示意图

（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 3.11　float32 数据类型结构图

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：

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

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 3.12　浮点型输入数据示意图

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 3.13　浮点型未处理排序结果示意图

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：

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

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 3.14　整型正确输出示意图

3.3.5　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

本章首先概述了国产 AI 处理器 Top-k 算子的计算流程，对主要工作进行介绍。随后详细介绍了主机端应该进行的主要任务,这些任务包括: 参数准确性检查, 核函数配置，输入预处理和输出后处理。最后详细介绍了设备端基于 RadixSelect 算法的并行实现：首先作出了小 k/大 k 场景的划分，而后基于 MLU-Core 的片上资源分别进行了并行设计与实现。

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　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国产 AI 处理器中，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 BangC 编程中，应该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。对于 MLU-Core 内的访存行为而言，其主要针对 NRAM 空间，在数据到达 NRAM 空间后，数据已经是连续存储。但对于 MLU-Core 针对 GDR 进行访问时，MLU-Core 之间的不连续的访问模式将可能会导致 GDR 的 bank 冲突。而所谓 bank 冲突即：GDR 被划分为多个 bank（优点是可以隐藏部分延时），GDR 的吞吐是建立在 bank 负载均衡的前提下，如果有多个请求同时落在同一个 Bank 内，那它一次只能响应其中一个，导致这些请求会被序列化，这就是 Bank Conflict（Bank 冲突）。发生冲突的 bank 负载是满的，问题是其他 bank 的请求是不足的，也就是负载不均衡，导致带宽利用率不足。bank 冲突和请求的地址，GDR 的形态以及地址映射相关。在国产 AI 处理器上是 5HBM × 8 channel * 32KB=1.25MB。通过 MLU-Core 间连

续访存可以有效的规避 Bank 冲突。主要通过更改 task 与数据之间的映射关系即可实现。参考下图 4.1：

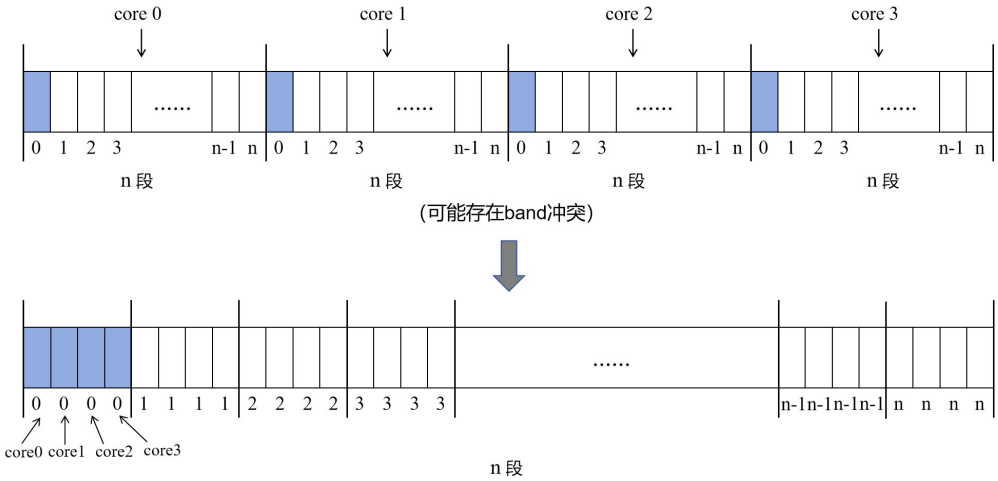

图 4.1　Cluster 级连续 I/O 示意图

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，其对齐要求见下表 4.1。

表 4.1　存储类型及其对齐方式

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

此优化方法体现在对各个变量进行空间划分时，大小尽可能满足数据对齐的倍数即可。

（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。对于 Top-k 算子而言，其 filter 阶段可以运行流水机制。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 4.2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 4.2可知，通过软流水，在同一个

图 4.2　三级流水示意图

时间片内可以进行加载（Load）、计算（Compute）以及存储（Store）操作。当 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，仅对于过滤（FILTER）阶段的计算，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 算子的过滤（FILTER）计算阶段，可以利用 SRAM 来完成结果的缓存。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 4.3 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，

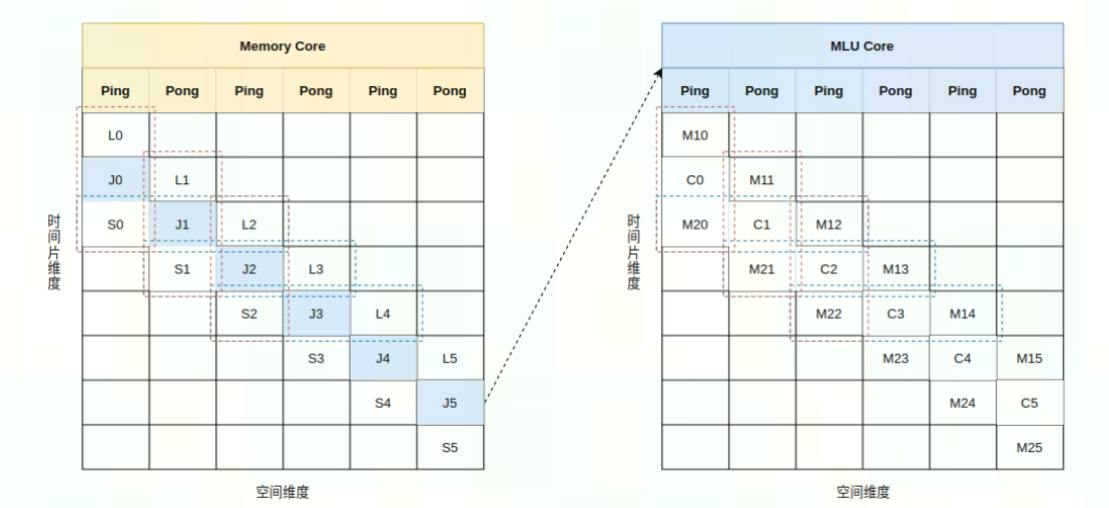

图 4.3　五级流水示意图

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

（1） 算法优化

对于 RadixSelect 算法而言，在特定的情况下，数据可能会落入所有的桶中，这将会使得 FILTER 阶段的操作变得没有意义，因为此时不存在无效数据。因此在 FIND TARGET BUCKET 阶段，可以对各个桶进行检查：当桶中的数据等于本轮迭代所有数据个数时，直接进入到下一次迭代。这将有效的减少冗余计算。更改后的 FIND TARGET BUCKET 阶段伪代码片段如下：

Algorithm 2 CountingSort Algorithm-Early Stop

```
// FIND TARGET BUCKET WITH EARLY STOP
STOP ← False
 for i ← 0 to 2^r − 1 do
    if Bucket[i] = AllNum then
        STOP ← True
        break
    end
    if Bucket[i] ⩾ k then
        POS ← Bucket[i]
        break
    end
end
if STOP = True then
    break
end
```

（2） 预处理计算优化

对于 Top-k 算子而言，RadixSelect 算法仅将 Top-k 大/小的值筛选出来。对于内部排序还需要根据 sorted 参数来决定是否使用 RadixSort 算子进行计算。而正负性等数值特性对于 RadixSort 和 RadixSelect 的影响是相同的，因此通过将 RadixSelect 和 RadixSort 的预处理方式进行对齐，能够有效的减少指令数目，提高 Top-k 算子的计算效率。

通过前文的分析，我们总结出 RadixSort 和 RadixSelect 预处理方案如下：

1. Top-k 小，整型，所有数的符号位进行取反操作。
2. Top-k 小，浮点型，对正数进行符号位取反，对负数全部 bit 位进行取反，即非负数与 0x80000000 进行位异或操作，负数与 0xffffffff 进行位异或操作
3. Top-k 大，整型，对所有数据的非符号位实施取反操作。
4. Top-k 大，浮点型，数据的符号位保持不变且负数无需进行任何改动，对非负数的非符号位进行取反操作。

而后，根据 sorted 参数来判断是否需要运行 RadixSort 核函数。若是需要运行，此时 RadixSelct 的输出不需要进行后处理，直接将其作为 RadixSort 的输入即可。

（3） 指令流水线优化

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。例如每次迭代的开始，需要用到 bang_write_value 指令对 Buckets 数组进行初始化。而在 bang_write_value 指令 Compute 流执行，同样可实现给指定内存段赋初值的 memset_nram 指令则在 Move 流执行。因此通过使用 memset_nram 指令替代 bang_write_value，可以实现向量运算与内存段赋初值操作并行执行，减少总的指令运行耗时。

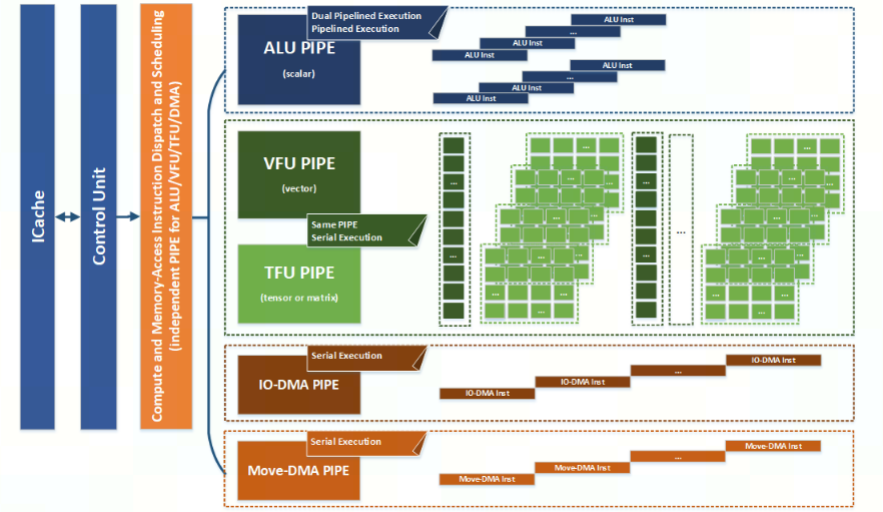

图 4.4　多队列指令流水线

（4） 指令融合优化

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 4.5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。在 RadixSelect 实现阶段，数据预处理阶段满足指令融合的要求，因此在此阶段可以进行指令融合。

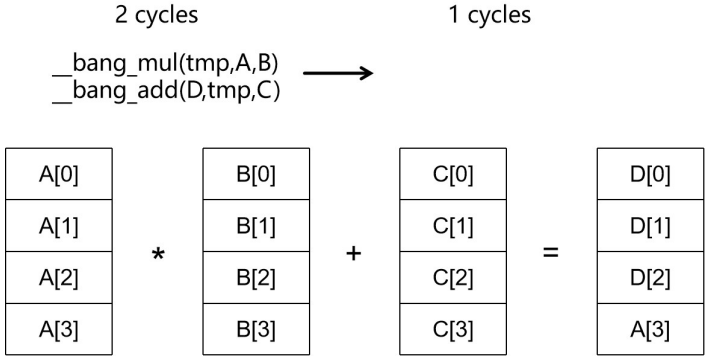

图 4.5　指令融合示例图

4.3.2　Cluster 级优化

MTP Cluster 架构如图 4.6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。而任务类型描述的是任务的硬件需求，任务规模描述的是任务的划分方式，而任务映射则建立了具体任务与物理硬件之间的映射关系。为了便于描述任务规模，在 BANG C 编程语言中引入了 cnrtDim3_t 数据类型。其中，与任务规模相关的内置变量包括：taskDim 、taskDimX 、taskDimY 以及 taskDimZ。对于 UnionN 类型的任务，需要至少有 N 个 Cluster 空闲时才能下发任务。同时还要满足约束 taskDimX % (N * coreDim) = 0 。最小的并行度为 clusterDim * coreDim 。在不考虑任务展开的情况下，迭代次数 =taskDimZ * taskDimY * taskDimX / (N * coreDim) 。驱动根据当前硬件的实时利用率决定是否将任务展开，尽可能占满所有的硬件资源。例如：当任务类型为 Union1, 任务规模为 X=8, Y=1, Z=1 时，默认情况下，该任务会在一个 Cluster 上经过两轮迭代才成完成。如果，驱动在下发任务时发现硬件至少有两个 Cluster 空闲，那么还会做任务展开，让 Union1 类型的任务同时占用 2 个 Cluster，这样只需要一轮迭

代即可完成。

在定义 Job 类型为 UnionX 类型时, Job 被调度的物理单元将会以 X 个 Cluster 为单位。通过前文的介绍，我们知道 RadixSelect 核函数的 Job 数量由 A 来确定，即每个 Job 处理输入的一行，因此将会有 A 个 Job。具体的 Job 类型由 B 的大小来确定，或者说单个 Job 所占的硬件资源由 B 确定。

因此，若是按照上述固定的规则但是当 A 的数量较少，B 的数值很大时，将会出现某些 MLU-Core 不满足 Job 的构成规则而处于闲置的状态。例如，假如芯片上有 24 个 Cluster（96 个 MLU-Core）, 并且此时的 A 数值为 9，B 数值同样很大（大于 200w），此时 Job 类型将会被定义为 Union4 类型, 即单个 Job 将会占据 16 个 Cluster（64 个 MLU-Core），剩下的 8 个将会一直处于闲置状态，Cluster 的利用率为 75%。而假若此时设定为 Union3 类型，则对于整个计算任务而言，Cluster 的利用率为 100%。因此需要对此中计算规模进行优化。分析其根本原因，是 Job 的任务粒度太大导致的。因此将多核运行的 Job 类型分配确定为最高上限为 Union3。

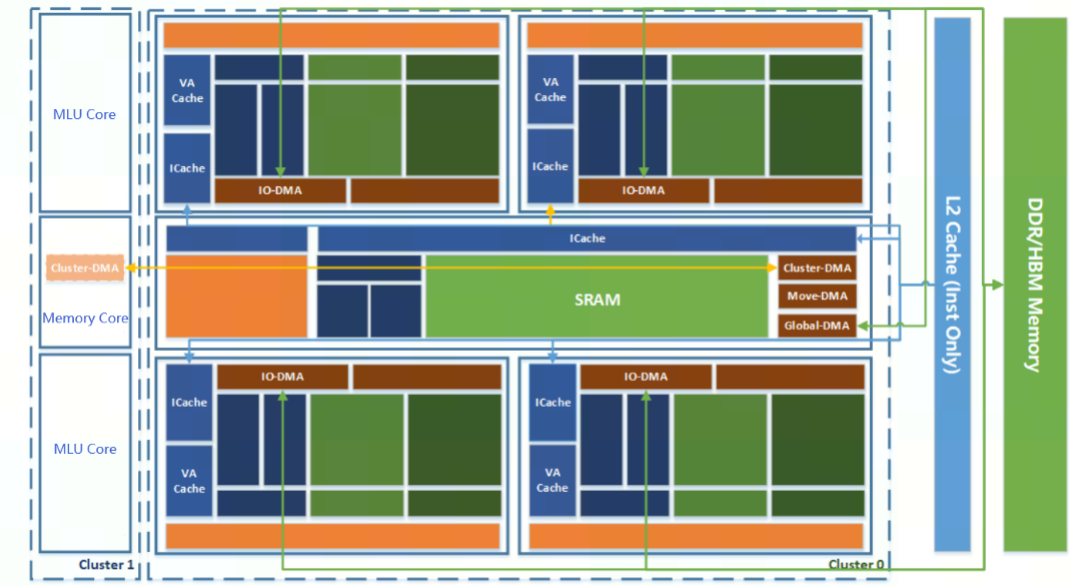

图 4.6　DLP-M Cluster 架构

4.3.3　芯片级优化

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设备端与主机端之间的 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，使得在输入的 A 维度较大的情况下，可以充分利用 MLU 芯片上的 Cluster 计算资源，并行地进行 Top-k 计算任务，从而提升算子的并行处理性能。假设定义了 6 个队列，在实际实现上，首先读取输入，同样设输入规模为 [A,B], 然后用 A 除以 6，得到每个任务队列至少需要处理的输入行数 num_per_queue。

当 n 不能被 6 整除时，将余下的待处理行从编号为 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数据。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 算子的计算。最后再次调用 rtMemcpyAsync 异步数据迁移接口将计算好的结果从设备端迁移到主机端即可。通过这样的操作，可以有效的将主机端与设备端之间的 IO 与计算相互掩盖。达到减少整个任务计算时间的效果，如图 4.7所示。

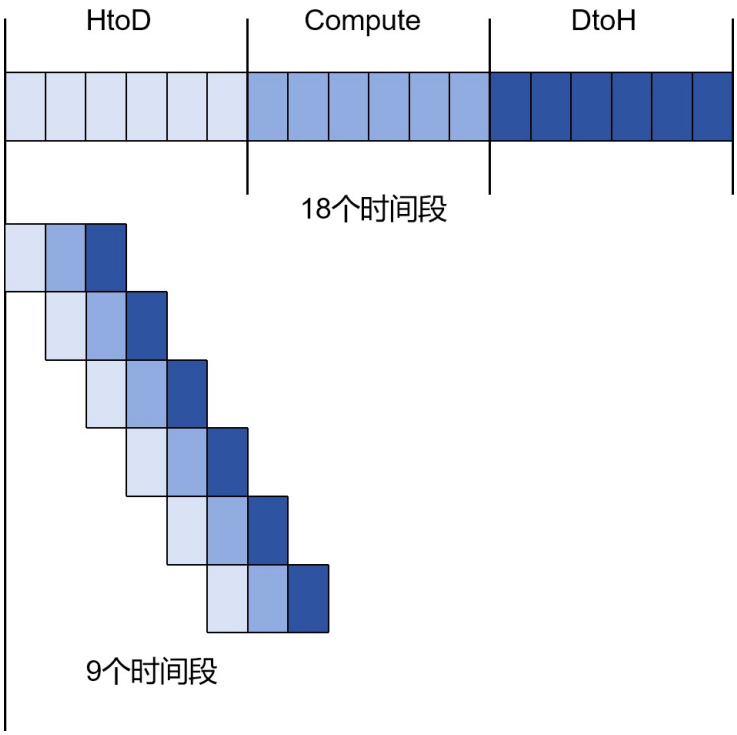

图 4.7　多队列优化效果图

第 5 章　基于国产 AI 处理器的 Top-k 算法测试验证

本章首先介绍了 Topk-k 算子测试用到的实验环境，同时也描述了整个算子测试的流程。接着自行编写脚本生成大量的测试用例，用于测试自实现 Top-k 算子的准确度是否满足要求，以确认功能是否正确。然后测试了不同输入规模下，Topk 算子相对于全排序 Top-k 算子和 Nvidia 芯片上 Top-k 算子的性能表现情况。最后将自实现 Top-k 算子集成在 Pytorch 深度学习框架中，对目前比较流行的模型进行训练，并通过检测结果验证了 Top-k 算子的可用性。

5.1　实验环境与测试流程

5.1.1　实验环境

测试的环境配置如表 5.1所示。

表 5.1　硬件环境配置

类型	配置信息
AI 处理器	DLP-M 处理器
CPU	Intel Xeon Silver 4114
Nvidia GPU	A100-40G
操作系统	CentOS - 7
开发语言	BC 4.7.0, C++11, Python 2.7.5
编译器	CC 4.7.1, gcc 4.8.5

本次实验以 CPU+ 国产 AI 处理器（以下简称 MLU）异构编程模式进行, 用到了 BC、C++ 以及 Python 三种编程语言。BC 编程语言用于实现 MLU 端的 Top-k 算法计算过程；C++ 编程语言则用于实现 CPU 端的数据准备；Python 编程语言用于编写测例用例生成脚本。CC（Compiler Collection，BC 语言编译器）是基于 Clang 和 LLVM 开发的 BC 编译器主驱动程序，负责将 BC 源码文件编译为 MLISA 汇编文件。

5.1.2　测试流程

1. 通过 RT 接口初始化设备,RT 包含设备管理、内存管理、任务队列管理、设备端程序执行、通知管理等功能，使用 BC 编写的程序需要借助于 RT 提供的接口才能运行在 MLU 设备上。
2. 初始化 Top-k 描述符，并为保存 Top-k 参数内容的结构体创建一个句柄。同时计算出当前输入参数下计算得到的 Top-k 特征向量需要占用的内存空间大小, 并分配对应的内存空间。
3. 准备输入数据，调用 RT 接口分配设备内存，并将输入数据拷贝到设备内

存中。

4. MLU 端调用自实现的 Top 算子，得到 Top-看特征向量。CPU 端调用 RT 接口等待任务队列执行完成。
5. 调用 RT 接口将 Top-k 算子的计算结果复制到主机侧。
6. 主机侧调用 Top-k 的串行实现，得到比对的计算结果。
7. 将设备端的计算结果与主机端的计算结果进行比较。
8. 调用 RT 接口，释放主机端和设备端的内存以及任务队列等资源。

整个计算流程如图 5.1所示:

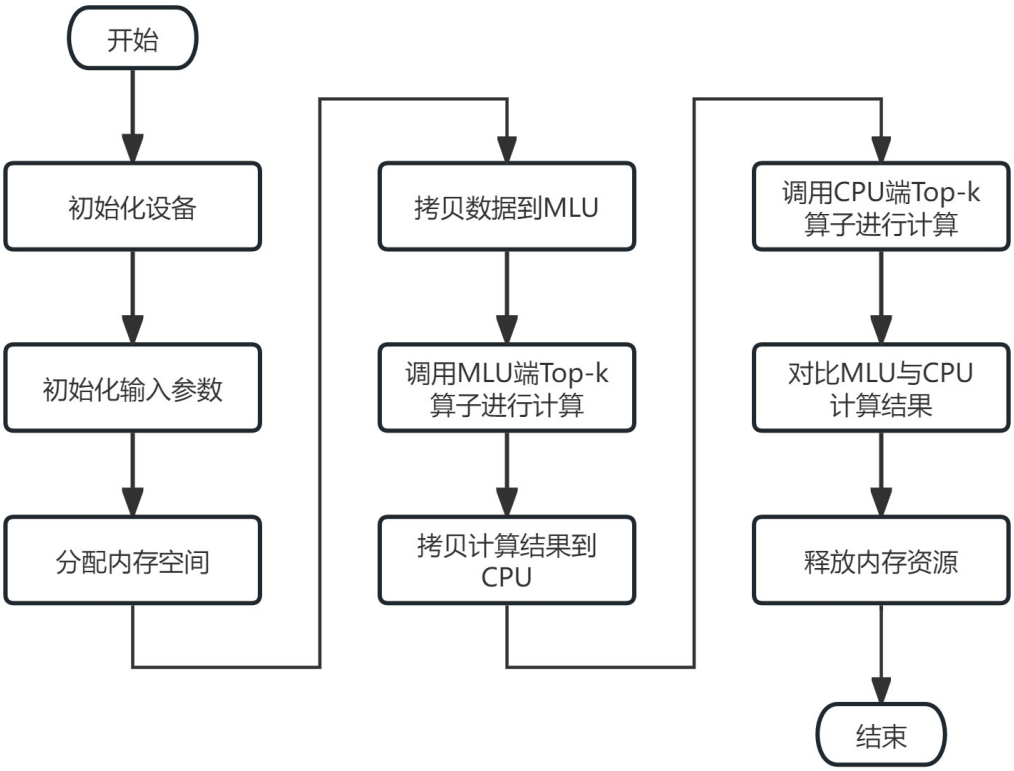

图 5.1　RadixSelect 测试流程图

5.2　Top-k 算子功能测试

在完成深度学习处理器的 Top - K 算子实现后，便可以开展算子的功能性与性能验证工作。首先是功能验证，其主要内容如下:

1. 对不同维度（dim）下 Top-k 查询结果的正确性进行验证。
2. 针对不同 k 值下的查询结果正确性进行验证。
3. 检验经 largest 排序后的数据返回是否正确。

在进行 Top-k 正确性验证时，将计算结果与 CPU 计算结果进行对比，误差计算公式见，其总 mlu_output 表示设备端的计算结果，cpu_output 表示主机端使

用 CPU 的计算结果。

$$diff1 = \frac{\sum |baseline - mlu|}{\sum |baseline| + \epsilon} \tag{5.1}$$

$$diff2 = \frac{\sum (baseline - mlu)^2}{\sum (baseline)^2 + \epsilon} \tag{5.2}$$

其中，diff1 表示与 CPU 返回的 Top-k 计算结果的相对误差，diff2 表示与 CPU 返回的 Top-k 数据的均方差。考虑到 Top-k 算子仅仅只是筛选类算子，不在原数据上做具体的操作。因此 diff1 和 diff2 应当为 0 才能满足精度要求。由于 Top-k 算子需要输出具体的值和坐标（index），因此以下的精度表格中，diff1 = diff1_value + diff1_index, diff2 = diff2_value + diff2_index。本部分主要展示输入数据的数据类型为 float 和 int 的误差。小 k 场景下的 Top-k 算子的精度测试结果如表 **??**所示。片上内存空间大小主要针对 NRAM 内存大小，结合具体实现过程，将不小

表 5.2　小 k 场景下 Top-k 功能测试表

数据规模	dim	k	largest	diff1	diff2
（10000）int	0	100	True	0.000000	0.000000
（1024,3000000）int	1	8000	False	0.000000	0.000000
（1,1000000,4,2）int	1	5000	True	0.000000	0.000000
（1,1000,4,2）int	1	500	False	0.000000	0.000000
（1024,3,300,4,8）float	2	300	False	0.000000	0.000000
（100, 3, 30000, 4, 10240000）float	4	20	True	0.000000	0.000000
（1,1,1,10240000）float	3	8192	False	0.000000	0.000000
（1,1000,10）float	1	500	True	0.000000	0.000000

于 10000 的 k 值视为大 k 场景。其精度测试结果见表 5.3。由测试结果可以看出，基于 RadixSelect 算法实现的 Top-k 算子，分别在大/小 k 场景下，皆满足对应参数下的精度要求，与 CPU 计算输出结果的各误差均为 0%。

表 5.3　大 k 场景下 Top-k 功能测试表

数据规模	dim	k	largest	diff1	diff2
（102400）int	0	12000	True	0.000000	0.000000
（1024,3000000）int	0	10000	False	0.000000	0.000000
（100, 3, 30000, 4, 8）int	2	15000	True	0.000000	0.000000
（400, 2000000）int	1	15000	False	0.000000	0.000000
（1,1,1,10240000）float	3	1000000	False	0.000000	0.000000
（1,1000000,4,2）float	1	20000	True	0.000000	0.000000
（1,1,1,10240000）float	3	1000000	False	0.000000	0.000000
（200,33669）float	1	14931	True	0.000000	0.000000

5.3　Top-k 算子性能测试

在本节中，我们针对两种场景下的核函数进行性能测试。首先是对比优化前后的性能提升效果。而后分别与 DLP-M 原版本 Top-k 算子，Nvidia-A100 Top-k

算子显卡进行性能对比。

5.3.1　小 k 场景下 RadixSelect 算子性能测试

（1）　优化前后性能对比

在使用前文中的各种优化手段后，本节对（100, 1310720）这一数据规模进行了实验测试，以说明对于任务规模优化，I/O 优化，计算效率优化后的加速效果。具体数据如表 5.4所示。在经过优化之后，性能加速效果维持在 48% 左右。

表 5.4　小 k 场景优化前后测试表

数据规模	k	dim	largest	优化前耗时（us）	优化后耗时（us）	加速比
（100,1310720）float	16	1	True	5970.96	3042	0.4905
（100,1310720）float	32	1	True	5988.87	3050	0.4907
（100,1310720）float	50	1	True	6168.98	3200	0.4813
（100,1310720）float	75	1	True	6185.77	3211	0.4809
（100,1310720）float	100	1	True	6162.35	3187	0.4828
（100,1310720）float	1000	1	True	6727.01	3493	0.4807
（100,1310720）float	8192	1	True	13016.7	6859	0.4731

具体效果如图 zhexian-lk-upgrade.png 所示，对比效果图我们可以发现，在相同输入规模下，随着 k 的增加，优化前后的时间都在逐渐增加。这反映了对于相同的输入规模而言，k 的值越大，单个 MLU-Core 的任务量也就越大，在执行 RadixSelect 算法时，其“FIND TARGET BUCKET”与”FILTER“中的任务也将更复杂。

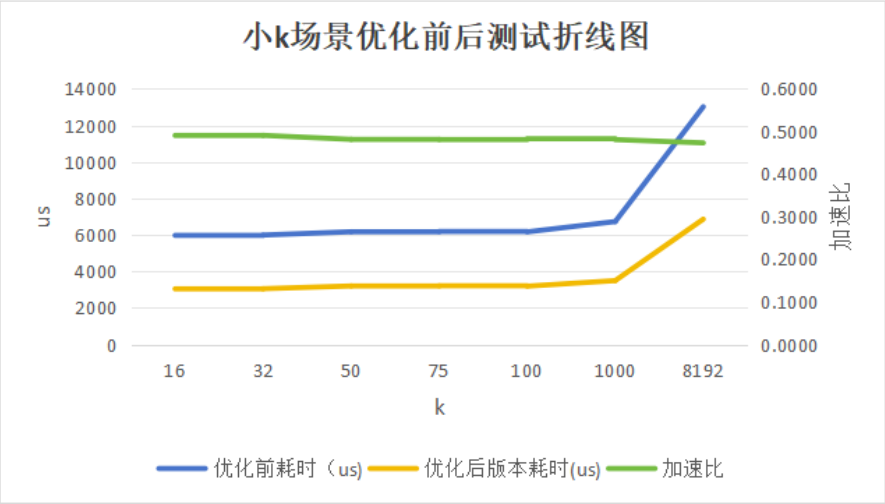

图 5.2　小 k 场景优化前后对比图

（2）　与原版本性能对比

原版本小 k 场景下的 Top-k 算子主要使用的是基于冒泡排序的 Top-k 算子，其串行算法的时间复杂度为 O（kN）。在 k 和 N 都不太大的时候，其性能表现尚可，但是当数据规模较大时，Top-k 算子的性能将会受到影响。同时，当数据中存在 Nan 和 Inf 时，原版本实现为了满足与 CPU 结果对齐的要求，将会引入很

多对于正常数据而言不必要的操作，性能将会大幅度下降。此时 Top-k 算子主要基于 RadixSort 全排序算法。其性能对比数据如表 5.5所示。可以发现小 k 场景下的实现明显优于原版本。

表 5.5　小 k 场景与原版本性能对比表

数据规模	k	dim	largest	RadixSelect 耗时	原版本耗时	加速比
（1,80000）float	100	1	True	14.99	78.22	0.8093
（16,136960）float	10000	1	True	280	1535	0.8176
（32,136960）float	10000	1	True	400	3060	0.8693
（64,136960）float	10000	1	True	790	6405	0.8767

将 B 和 k 分别固定为 136960 和 10000, 对于不同的 A 的数值进行绘图，如图 5.3所示。可以发现随着输入数据规模的增大，相较于原来版本的加速效果越来越好。

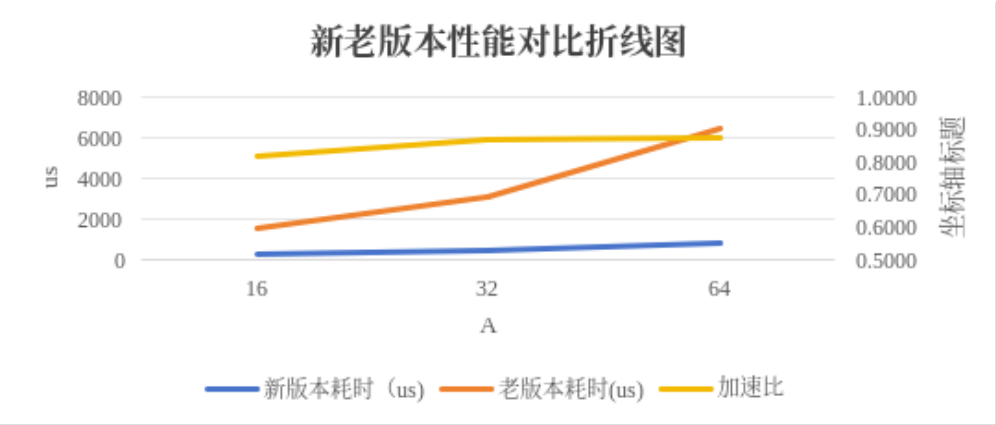

图 5.3　小 k 场景优化前后对比图

（3）　与 A100-GPU 性能对比

小 k 场景下 RadixSelect 算子性能与 NVIDIA A100 GPU 进行性能对比，其性能测试数据如表 5.6所示。

表 5.6　小 k 场景下 DLP-M/A100 性能对比表

数据规模	k	MLU590 实测 (us)	A100 耗时 (us)	加速比
（1,16384）	1024	25	105	0.76
（1,16384）	2048	23	115	0.80
（1,16384）	4096	31	116	0.73
（1,16384）	8192	35	180	0.81
（1,65536）	1024	40	72	0.44
（1,65536）	2048	46	75	0.39
（1,65536）	4096	56	76	0.26
（1,65536）	8192	74	138	0.46
（1,131072）	1024	45	79	0.43
（1,131072）	2048	51	80	0.36
（1,131072）	4096	67	82	0.18
（1,131072）	8192	102	145	0.30

在相同输入规模，不同 k 值下进行绘图，如图 **??** 和 5.4所示。随着输入规模的增大，DLP-M 平台与 A100 平台上的 Topk 算子的耗时都逐步增加，并且随着数据规模的增大，加速比总体上呈现逐步下降的趋势，当 k 过大时（如 8192），DLP-M 的加速比骤然变大，这应当主要由于 DLP-M 与 A100 的体系结构差异导致的，即此时 A100 上的 Top-k 算子可能切换到了别的版本实现中，但由于时间问题，并没有进一步深究。整体而言，在输入较小时，相较于 A100 可以达到 3.5-4 倍的提速。在输入较大时，相较于 A100 可以达到 1.3-1.8 倍的提速。

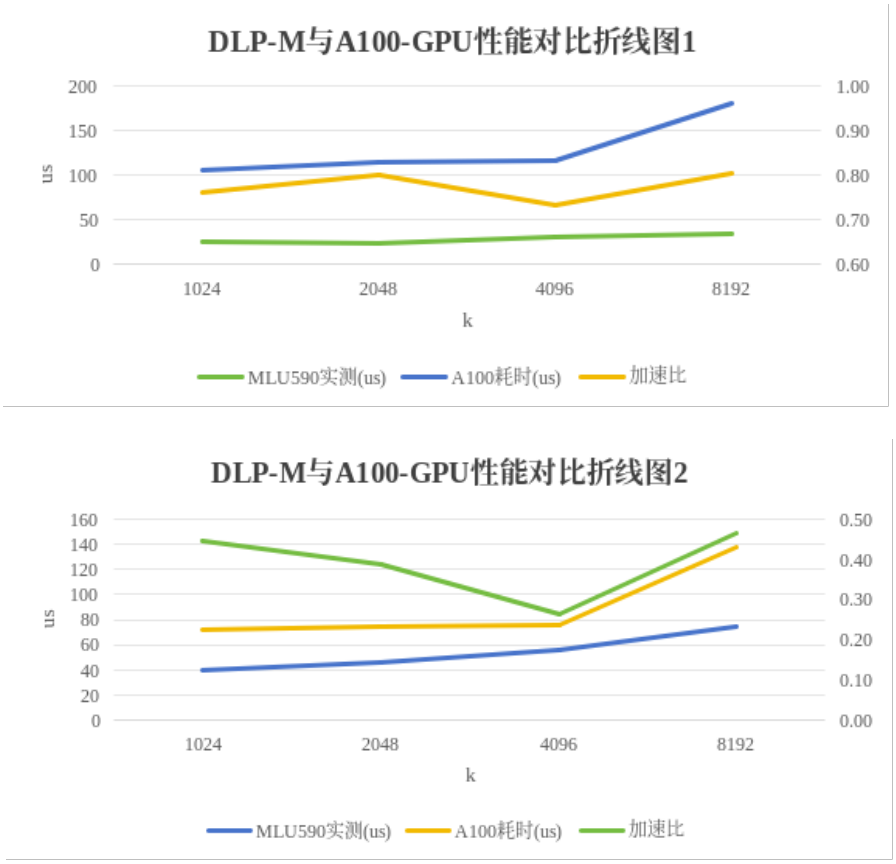

图 5.4　小 k 场景优化前后对比图

5.3.2　大 k 场景下 RadixSelect 算子性能测试

（1）　优化前后性能对比

通过对大 k 场景下的 RadixSelect 算子实现进行优化后，其在部分测试样例上的表现如表所示。在对于（100,1310720）这一输入规模下，随着 k 的增大耗时逐渐增加。相较于优化前，加速比维持在 0.13-0.18 之间。

（2）　与原版本性能对比

在大 k 场景下，大 k 版本的 Top-k 算子主要使用全排序版本，仅在最后输出时，将符合要求的 k 个数复制到主机端，因此原来版本的耗时比较稳定，而对于 RadixSelect 而言，随着 k 的值越来越大，每个 ML U-Core 的任务量也越来越大，在执行 RadixSelect 的耗时逐步增加，因此其加速比逐渐降低，其测试数据

表 5.7　大 k 场景优化前后性能对比表

数据规模	k	dim	largest	优化前耗时	优化后耗时	加速比
（100,1310720）float	16384	1	True	5222	4453	0.1473
（100,1310720）float	32768	1	True	6128	5283	0.1379
（100,1310720）float	65536	1	True	7299	6279	0.1397
（100,1310720）float	131072	1	True	8540	7061	0.1732

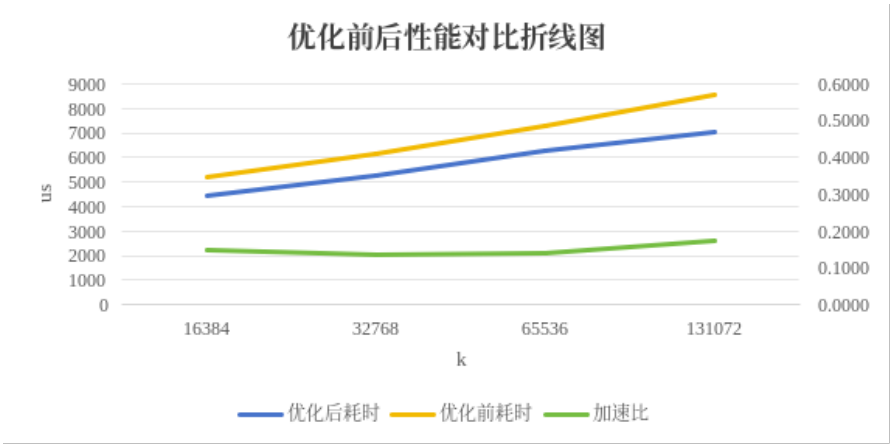

图 5.5　大 k 场景优化前后对比图

如表 5.8所示，折线图如图 5.6所示。

表 5.8　大 k 场景与原版本性能对比表

数据规模	k	dim	largest	RadixSelect 耗时	原版本耗时
（100,1310720）float	16384	1	True	4453	33832.4
（100,1310720）float	32768	1	True	5283	34012.9
（100,1310720）float	65536	1	True	6279	34171.8
（100,1310720）float	131072	1	True	7061	34197

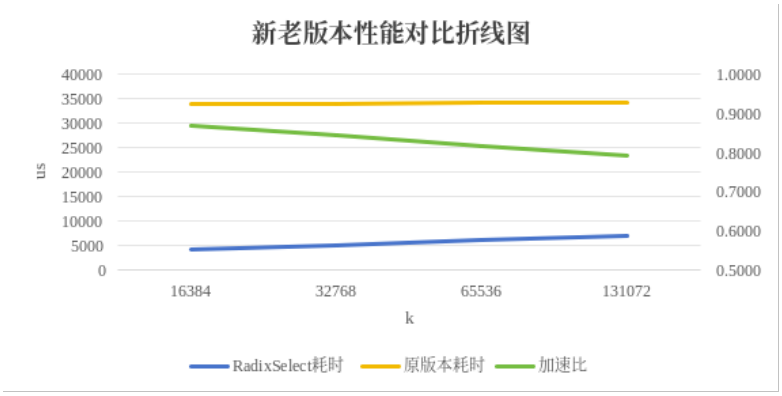

图 5.6　与原版本性能对比图

（3）　与 A100-GPU 性能对比

大 k 场景下 RadixSelect 算子性能与 NVIDIA A100 GPU 进行性能对比，在相同输入规模，不同 k 值下进行绘图，如图 5.7所示。随着输入规模的增大，DLP-M 平台与 A100 平台上的 Topk 算子的耗时都逐步增加，并且随着数据规模的增大，加速比总体上呈现逐步下降的趋势。整体而言，相较于 A100 可以达到 1.4-2.3 倍

的提速。

表 5.9　大 k 场景下 DLP-M/A100 性能对比表

数据规模	k	MLU590 实测 (us)	A100 耗时 (us)	加速比
(1,65536)	16384	68.5	145	2.1
(1,65536)	32768	92.5	143	1.6
(1,131072)	16384	76.2	153	2.0
(1,131072)	32768	85.5	155	1.8
(1,131072)	65536	102.1	157	1.5
(1,262144)	16384	89	165.3	1.9
(1,262144)	32768	101.2	167.1	1.65
(1,262144)	65536	121.7	170.1	1.4

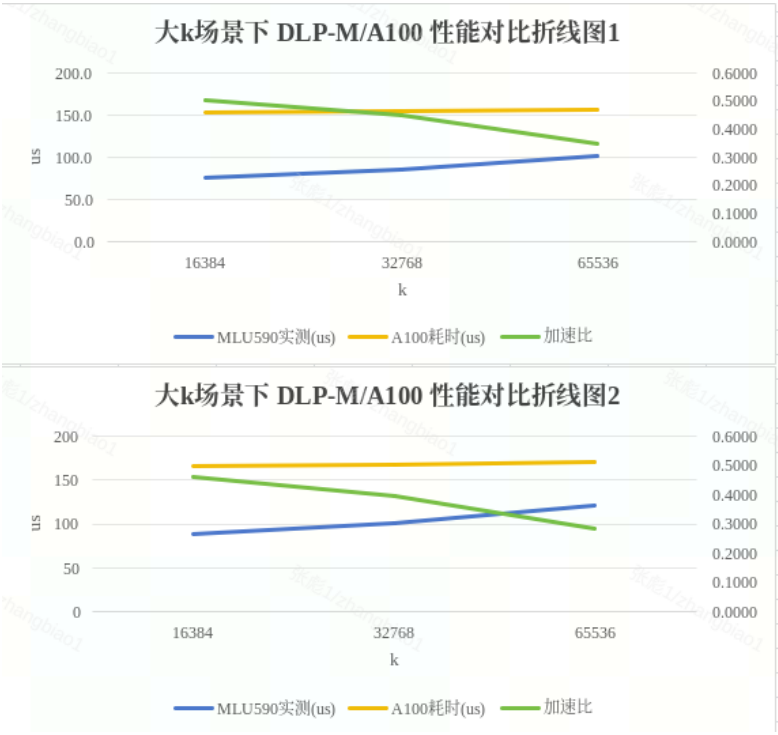

图 5.7　大 k 场景优化前后对比图

5.4　Top-k 算子集成测试

Top-k 算子的集成测试主要通过面向国产 AI 处理器的 Pytorch 框架进行。它通过 PyTorch 社区的后端集成机制允许用户在使用原生社区 PyTorch 的基础上灵活、快速的接入寒武纪 MLU 后端。为神经网络计算提供了强大的 AI 加速。

5.4.1　Pytorch 后端集成机制和网络搭建简介

（1）　Pytorch 后端集成机制

PyTorch 的后端集成核心步骤围绕着如何使新后端与框架无缝协作，主要包括注册内核、生成器、设备保护以及元数据的序列化和反序列化等操作，确保新后端在 PyTorch 中能高效运行并提供完整功能。在深度学习研究和实际应用

中，针对特定硬件或自定义计算逻辑的需求日益增长。为了支持用户在不修改框架核心代码的前提下扩展其功能，PyTorch 提供了一种灵活的扩展机制，称为 PrivateUseOne。该机制通过注册自定义设备类型和操作逻辑，允许用户在 PyTorch 环境中实现特定需求。PrivateUseOne 的核心功能是支持用户注册一个私有设备类型，并为其实现自定义张量操作。例如，当需要在特殊硬件设备（如实验性芯片或 FPGA）上运行张量计算时，用户可以通过 PrivateUseOne 定义专属的调度逻辑和运算规则。此机制基于 PyTorch 的底层设备调度框架，使得自定义设备与原生设备类型（如 CPU、GPU）无缝集成。

实现方法包括以下几个步骤：

1. 设备类型注册用户通过调用 PyTorch 提供的接口注册自定义设备类型。
2. 扩展张量运算用户可以实现特定的张量操作逻辑（如加法、矩阵乘法等），并将其关联到自定义设备。
3. 调度机制利用 PyTorch 的调度器，将相关计算操作分发到用户定义的设备中执行。

在 MLU 后端成功进行注册后，PyTorch Eager 模式下算子间数据传递和存储的基本单元是 tensor。PyTorch 根据 tensor 中的 device 属性值将算子分发到不同设备。以 abs() 算子为例，在 dispatch 阶段会根据 input_tensor 的设备属性值将算子调用分发到具体设备，如下图所示。

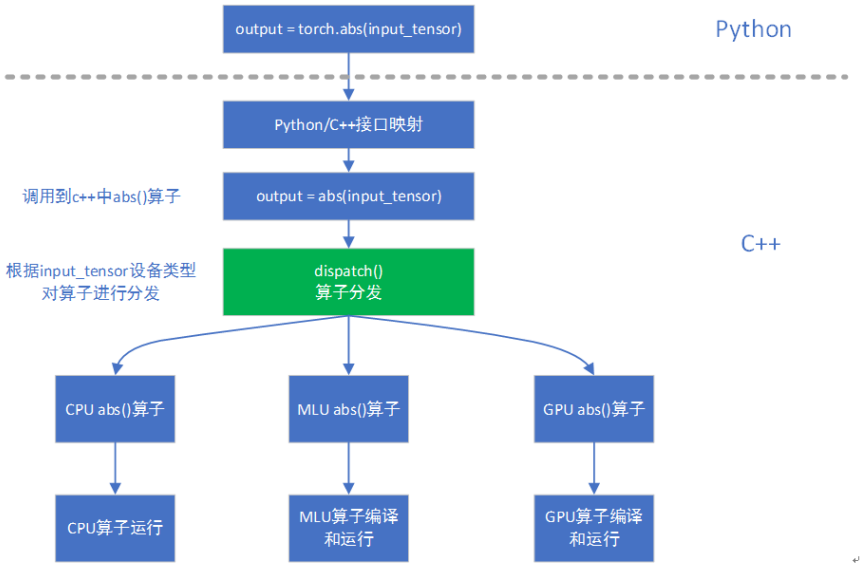

图 5.8　算子分发机制图

而在算子调用分发到具体设备之前，需要进行算子注册。其基本流程如下：算子注册的基本步骤如下：

1. 定义算子接口：使用 PyTorch 提供的 `native_functions.yaml` 文件定义算子的接口及其功能。在该文件中明确指定算子的名称、输入输出参

数以及调度规则。

2. 实现算子功能：在 C++ 或其他语言中实现算子的实际计算逻辑。对于 MLU 硬件，可以基于 Cambricon SDK 提供的算子库进行开发。
3. 绑定算子与调度逻辑：使用 PyTorch 的 `RegisterDispatchKey` 机制，将自定义算子绑定到 MLU 的计算后端。这一过程需要将算子注册到相应的 Backend（如 `MLU`）。
4. 测试算子功能：编写测试用例验证算子的正确性，并使用 PyTorch 提供的单元测试框架进行全面测试，以确保算子在不同输入条件下的性能和正确性。

其注册的关键点如下：

- 文件结构：修改 `native_functions.yaml` 和相关 C++ 文件是算子注册的核心工作，不同后端（如 CPU、CUDA、MLU）需要明确区分。
- 设备适配：结合 Cambricon SDK，针对 MLU 硬件优化算子的性能，支持异构计算场景（如多设备调度）。
- 调度与调用：注册后，PyTorch 会根据输入张量的设备类型，自动调用对应的算子实现。

通过以上步骤，我们可以将 RadixSelect Top-k 算子集成到 Pytorch 中，并在国产 AI 处理器上进行 Top-k 计算，供上游的模型进行使用。

5.4.2　多卡训练

Cambricon PyTorch （CATCH）目前支持对 tensor 进行单机多卡和多机多卡的卡间集合通信，对网络进行单机多卡和多机多卡的分布式训练。目前支持以下通信原语：卡间广播（broadcast）、卡间规约（allreduce、reduce、reduce_scatter）、卡间收集（allgather、all-to-all），支持进程间同步（barrier）。其中，卡间规约又包含 4 种操作：求和、求连乘、求最大值、求最小值。原生框架的 Distributed Data Parallel 分布式训练机制在单个机器上有两种使用模式：a、单进程多卡；b、多进程多卡，每个进程用一张卡（官方推荐模式）。

当前阶段，MLU 设备间通信依赖 MLU 设备通信后端 CNCL（Cambricon Communications Library，寒武纪通信库）。仅需要在初始化进程组实例时，将 backend 参数变换为 cncl，即可以基于国产 AI 处理器使用分布式训练框架进行分布式训练，继而进行算子的集成测试。

5.4.3　模型和数据集准备

在集成测试中，本文选用 Mixtral8×7B 大模型，其主要基于 Transformer 架构，支持上下文长度达到 32k token，并且前馈块被 Mixture-of-Expert（MoE）层

取代。模型主要参数如下表 5.10。

表 5.10　模型参数表

Parameter	Value
dim	4096
n_layers	32
head_dim	128
hidden_dim	14336
n_heads	32
n_kv_heads	8
context_len	32768
vocab_size	32000
num_experts	8
top_k_experts	2

其中，在整个模型架构中，本文所涉及的 Top-k 算子主要集中在 MOE 层。其主要结构如图 5.9所示，

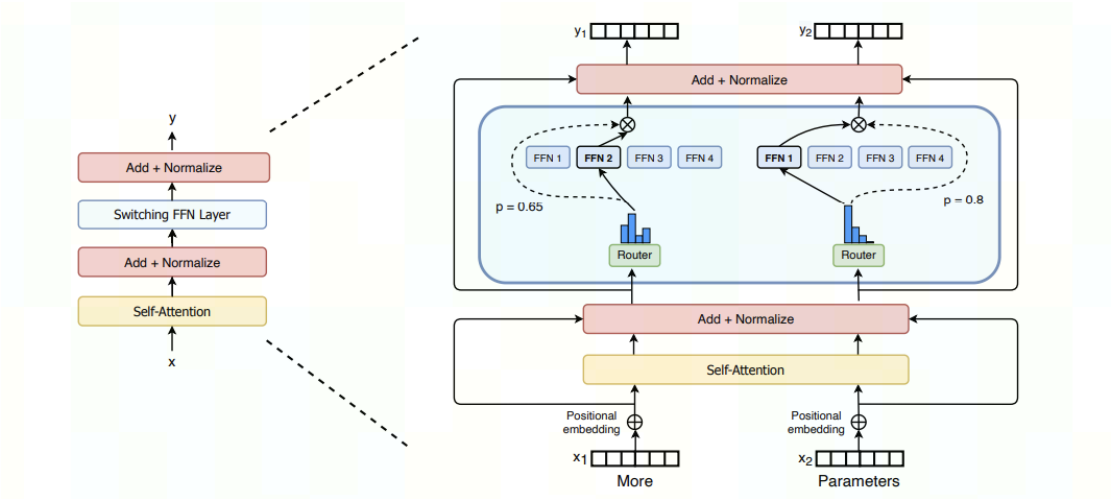

图 5.9　MOE 模型结构图

而 MOE 层的模型结构主要有两个关键部分组成:

1. 稀疏 MoE 层: 这些层代替了传统 Transformer 模型中的前馈网络 (FFN) 层。MoE 层包含若干“专家”(Mixtral8×7B 中有 8 个)，每个专家本身是一个独立的神经网络。在实际应用中，这些专家通常是前馈网络 (FFN)，但它们也可以是更复杂的网络结构，甚至可以是 MoE 层本身，从而形成层级式的 MoE 结构。
2. 门控网络或路由: 这个部分用于决定哪些 token 被发送到哪个 expert。例如，在下图中，“More”这个 token 可能被发送到第二个专家，而“Parameters”这个 token 被发送到第一个专家。有时，一个 token 甚至可以被发送到多个专家。token 的路由方式是 MoE 使用中的一个关键点，因为路由器由学习的参数组成，并且与网络的其他部分一同进行预训练。

计算过程如下述公式。

$$MoE(x) = \sum_{i=1}^{n}(G(x)_i E_i(x)) \tag{5.3}$$

$$G(x) = TopK(\text{softmax}(W_g x + \epsilon)) \tag{5.4}$$

上述第 1 个公式表示了包含 n 个专家的 MoE 层的计算过程。具体来讲，首先对样本 x 进行门控计算，W 表示权重矩阵；然后，由 Softmax 处理后获得样本 x 被分配到各个 expert 的权重；然后，只取前 k（通常取 1 或者 2）个最大权重；最终，整个 MoE Layer 的计算结果就是选中的 k 个专家网络输出的加权和。

在确定好模型之后，需要进行数据集的准备。在本集成测试实验中，使用的数据集为 OpenWebText，其是一个大规模的文本数据集，在自然语言处理（NLP）领域，特别是在无监督预训练语言模型的开发中具有重要地位。它包含数十亿个单词，能够为语言模型提供丰富的语言信息。它主要用于预训练语言模型，如 OpenAI 的 GPT（Generative Pretrained Transformer）系列模型。在预训练阶段，语言模型通过学习 OpenWebText 中的文本序列，捕捉语言的统计规律和语义信息，从而构建语言的内在表示。在进行训练之前，需要将其转换为 raw-data 形式，相关命令如下：

5.4.4　结果展示

由于训练成熟的大语言模型需要大量的训练技巧和硬件资源，对于算子的集成测试而言，其是没有必要的。因此本节中仅对比模型在国产 AI 芯片和 NVIDIA GPU 前 100 step 的训练表现，以此来作为国产 AI 芯片上的 Top-k 算子可用的依据。

GPU 与 MLU 模型训练 loss 如图 5.10所示，我们可以发现，在整个训练过程中，国产 AI 芯片的训练 loss 趋势与 NVIDIA A100 GPU 的 loss 趋势基本保持吻合。为了更好的衡量两只之间的吻合程度，我们进行相对误差分析，其公式如 5.5所示：

$$diff = \frac{|mluloss - gpuloss|}{gpuloss} \tag{5.5}$$

对于相对误差，一般要求其低于 0.01。GPU 与 MLU 模型训练 loss 相对误差如图 5.11所示，我们可以发现，在每个 step 中，相对误差基本都维持在 0.01 以下，满足了基本要求。

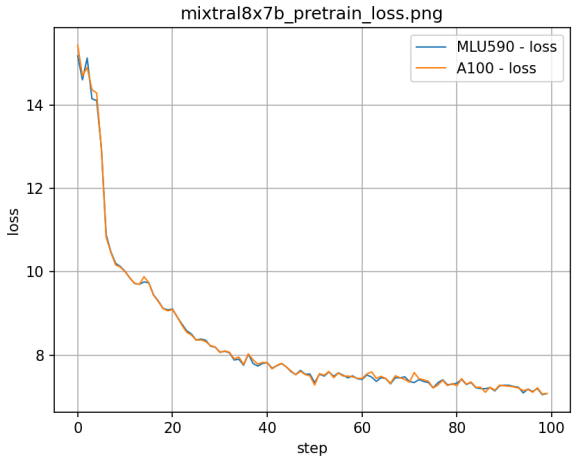

图 5.10　GPU/MLU 训练图

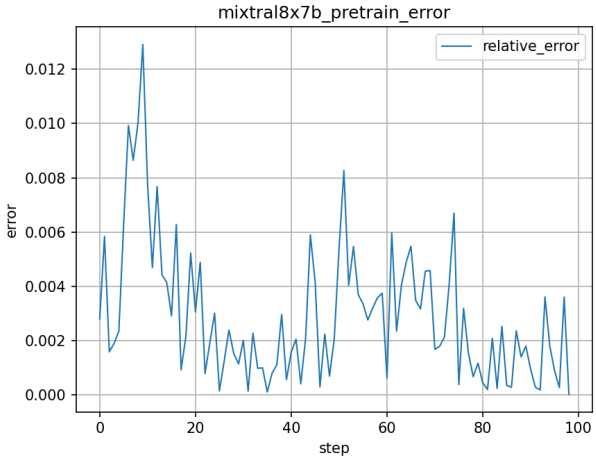

图 5.11　GPU/MLU loss 误差图

5.5　本章小结

本章首先介绍了 Top-k 算子测试实验所依赖的软硬件环境，紧接着对 Top-K 算子的测试流程进行了系统的概述。鉴于 Top-k 算子输入参数的多样性和复杂性，为了确保测试能够全面覆盖所有参数的取值，基于大/小 k 两种场景分别进行了功能测试和性能测试。最后将 Top-k 算子注册在 Pytorch 框架中，基于 Mixtral8×7B 大语言模型进行 Top-k 算子的集成测试。验证了本文 Top-k 算子的可用性。

第 6 章　总结与展望

6.1　本文工作总结

在本文中，我们针对国产 AI 处理器设计并实现了一种高效的 Top-k 查询算子，基于 RadixSelect 算法。通过深入的设计、优化与广泛的性能测试，本文展示了该算子在国产 AI 处理器上的应用潜力和优势，显著提高了 Top-k 查询的效率和准确性。本文的研究工作可详细总结如下：

（1） 算子设计与实现

本文基于 RadixSelect 算法，为国产 AI 处理器设计了一种高效的 Top-k 查询算子。RadixSelect 算法是一种基于基数排序的选择算法，适用于处理大规模数据集的 Top-k 查询问题。算子的设计考虑到了国产 AI 处理器的硬件架构特性，包括其多核并行计算能力和高效的内存访问机制。通过合理地设计数据流和计算流，算子能够充分利用处理器的并行计算资源，有效地提升查询效率。算子实现的过程中，特别关注了数据并行性的提升和硬件资源的充分利用。通过分析处理器的内存层次和计算单元特性，我们优化了数据加载、处理和存储过程，以减少内存访问延迟和提高数据处理速度。此外，针对国产 AI 处理器的特定优化指令集，我们实现了算子的多版本优化，以适应不同的运行条件和性能需求。

（2） 性能优化

针对国产 AI 处理器的硬件特性，本文实施了多种性能优化策略。这些策略主要包括计算效率和访存效率的优化，具体如下：

① 计算效率优化

为了提高算子的计算效率，我们采用了向量化和并行化技术。通过将 Top-k 查询操作分解为多个可以并行执行的小任务，算子能够在多个核心上同时运行，显著加快了处理速度。此外，我们还利用了国产 AI 处理器支持的特定指令，如 SIMD 指令，来加速关键的计算步骤，如数值比较和排序。

② 访存效率优化

访存效率的优化主要通过优化数据的存储布局和访问模式来实现。我们设计了专门的数据缓冲策略，以减少内存访问的开销。通过对数据进行预取和重新布局，算子能够更有效地利用处理器的缓存系统，减少数据传输延迟。

（3） 测试验证

我们对设计的 Top-k 查询算子进行了广泛的功能性和性能测试。功能测试验证了算子在不同配置和数据集上的正确性和稳定性。性能测试则比较了算子在国产 AI 处理器上的运行效率与其他平台（如 CPU 和 NVIDIA GPU）的性能。测试结果表明，本文实现的 Top-k 查询算子在功能上完全符合预期，在性能上则显

著优于传统的 CPU 实现和现有的 GPU 实现。对于大规模数据集的查询，算子能够在保证高准确性的同时，提供远超传统技术的查询速度，展示了其在实际应用中的高效性和可靠性。

6.2　未来工作展望

尽管本文的研究已经取得了一定的成果，但在 Top-k 查询算子的设计与优化方面仍有进一步的研究空间。未来的研究可以从以下几个方向进行：

（1） 算法优化与创新

基于国产 AI 处理器的体系结构，继续探索更高效的算法和数据结构来优化 Top-k 查询算子，特别是在大 k 场景下。算法的创新也是提升性能的关键，如结合机器学习技术预测数据特性，动态调整查询策略。

（2） 硬件适应性研究

随着国产 AI 处理器技术的快速发展，未来的研究可以包括算子在不同处理器架构上的适应性研究。这将有助于算子在更广泛的应用领域中的部署和使用，特别是在面对新兴的 AI 硬件时，能够快速适应其架构和性能特点。

（3） 跨领域应用

探索 Top-k 查询算子在更多领域和场景下的应用，如生物信息学、金融分析和社交网络分析等。每个领域的数据特性和查询需求不同，需要对算子进行相应的调整和优化，以满足特定应用场景的需求。比如在进行物理模拟时，需要的精度往往较高，因此需要对 Top-k 算子进行更高精度的数据类型支持。

参 考 文 献

[1] FAGIN R, LOTEM A, NAOR M. Optimal aggregation algorithms for middleware[J]. Journal of Computer and System Sciences, 2001, 63(3): 614-656.

[2] ILYAS I F, AREF W G, ELMAGARMID A K. Ranked join algorithms for relational databases [J]. ACM Transactions on Database Systems (TODS), 2003, 28(1): 1-51.

[3] CHIERICHETTI F, KUMAR R, TOMKINS A. Fast top-k query algorithms for streaming and static data using mapreduce[J]. Proceedings of the VLDB Endowment, 2009, 2(1): 410-421.

[4] JAIN N, MISHRA S, SRINIVASAN B. Top-k query optimization for apache spark streaming [A]. 2016.

[5] SATULURI V, PARTHASARATHY S, RUAN Y. Parallel top-k: Scalable and accurate top-k graph pattern matching in the gpu[J]. Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, 2010: 397-408.

[6] ALY S A, EL-KHARASHI M W. Accelerating top-k queries using fpga[J]. 2015 International Conference on Reconfigurable Computing and FPGAs (ReConFig), 2015: 1-6.

[7] WANG H, LIU Y, ZHANG Y, et al. Hybridcpu - gpu parallel top-k join algorithm[J]. Cluster Computing, 2016, 19(1): 165-177.

[8] LIU Z, DENG W, LI G. Deep top-k sparsification for neural network compression[A]. 2020.

[9] GUO J, ZHANG Y, WANG Y. Distributed index method for top-k queries[J]. Journal of Computer Research and Development, 2007, 44(12): 2107-2113.

[10] ZHANG M, CHEN Z, LI X. Efficient top-k query algorithm based on task scheduling in cloud computing environment[J]. Journal of Software, 2014, 25(11): 2751-2762.

[11] LI M, WANG H, LIU X. Parallel top-k algorithm based on sliding window for real-time stream data[J]. Journal of Computer Applications, 2018, 38(9): 2587-2591.

[12] ZHANG W, LI Y, ZHAO X. Top-k query dynamic load balancing strategy in distributed environment[J]. Journal of Computer Engineering and Applications, 2019, 55(18): 104-109.

[13] WANG P, ZHANG Q, LI Z. Optimization of parallel top-k algorithm on ft-gpu architecture [J]. Journal of Computer-Aided Design Computer Graphics, 2020, 32(12): 2011-2019.

[14] ZHAO L, LIU J, WANG S. Fpga-based top-k hardware acceleration module[J]. Journal of Electronics Information Technology, 2021, 43(10): 2531-2537.

[15] ZHAO L, ZHANG Y. Distributed top-k query algorithm combined with differential privacy mechanism[J]. Journal of Computer Science and Technology, 2022, 37(3): 619-632.

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

参与的实践项目及研究成果

1. 算子库开发与优化: 参与国产 AI 处理器算子库的日常开发与维护工作
2. Top-k 算子设计与实现: 设计并优化了基于国产 AI 处理器的 Top-k 算子