面向迈创3000异构处理器的多头注意力机制多重并行优化

路 瑶 栾钟治 李 根 齐家兴 韩 斌 杨海龙 钱德沛

(北京航空航天大学计算机学院 北京 100191)

摘 要 针对迈创3000(MT-3000)异构多核处理器在带宽不足场景下多头注意力(Multi-Head Attention, MHA)计算效率低的问题,本文提出了一套涵盖算子优化、访存优化与调度优化的综合方案,以加速PyTorch框架中的MHA推理。通过融合MT-3000的VLIW指令集、片上存储层次和DMA传输特性,设计了三方面的优化策略:在算子方面,对Linear和Softmax等算子进行内核级优化与算子融合,减少计算与访存开销;在访存方面,利用广播机制与全局共享内存(GSM)优化数据流,降低主存带宽依赖;在调度方面,以行为粒度分块并行,隐藏数据传输延迟。实验表明,优化后的Linear算子单簇峰值性能达1.53 TFLOPS,占理论峰值的37.7%,较NVIDIA V100 GPU加速比最高达5.34倍;在典型大语言模型配置下(嵌入维度4096/8192,头数32/64),MHA机制相较NVIDIA V100 GPU实现最高23.53倍加速,且在单节点多簇环境中扩展性良好。本研究为MT-3000在长序列推理任务中的高效部署提供了解决方案,并为天河超算支持大语言模型等复杂AI任务奠定了技术基础。

关键词 MT-3000;多头注意力;性能优化;PyTorch;加速比

中图法分类号 TP338 **DOI号** 10.11897/SP.J.1016.2025.02049

Multi-Level Parallel Optimization of Multi-Head Attention Mechanism for the MT-3000 Heterogeneous Processor

LU Yao LUAN Zhong-Zhi LI Gen QI Jia-Xing HAN Bin YANG Hai-Long QIAN De-Pei

(Department of Computer Science and Engineering, Beihang University, Beijing 100191)

Abstract With the rapid adoption of Transformer models in artificial intelligence domains such as natural language processing (NLP) and computer vision, the computational efficiency of their core component—the multi-head attention (MHA) mechanism—has become critical for model inference performance. However, in high-performance computing (HPC) scenarios emphasizing low power consumption and energy efficiency, traditional GPU architectures face significant challenges due to memory bandwidth limitations and energy consumption issues. The MT-3000, a CPU-DSP heterogeneous multi-core processor designed for HPC, emerges as a promising alternative to GPUs through its Very Long Instruction Word (VLIW) architecture, high computational density, and exceptional energy efficiency (45.4 GFLOPS/W double-precision efficiency, 62% higher than NVIDIA V100 GPU). Nevertheless, MT-3000's limited main memory bandwidth

收稿日期:2025-01-16;在线发布日期:2025-05-28。本课题得到国家重点研发计划项目"面向新一代国产超算系统的应用支撑环境和开发框架"(No.2023YFB3001900)资助。**路 瑶**,博士研究生,中国计算机学会(CCF)会员,主要研究领域为高性能计算、并行计算。E-mail:luyuan@buaa.edu.cn。**栾钟治(通信作者)**,博士,副教授,中国计算机学会(CCF)会员,主要研究领域为分布式计算、高性能计算、并行计算、计算机系统结构、云计算和大数据处理等。E-mail:rick710055@263.net。**李 根**,硕士研究生,主要研究领域为高性能计算、并行计算。**齐家兴**,博士研究生,主要研究方向是日志分析和高性能计算。**韩 斌**,博士研究生,主要研究方向是高性能计算、并行计算、任务调度。**杨海龙**,博士,教授,主要研究方向为深度学习编译优化技术、大模型训推系统、高性能计算、性能分析与优化、稀疏数值算法。**钱德沛**,教授,博士生导师,中国科学院院士,主要研究方向包括高性能计算机体系结构、分布式系统、众核处理器并行编程等。

(13 GB/s) and frequent I/O operations constrain its performance in memory-intensive MHA computations. Existing optimization strategies primarily targeting GPU high-bandwidth memory (HBM) and Tensor Cores prove ineffective for alleviating DSP platform bandwidth bottlenecks, necessitating architecture-customized optimization solutions. To address this challenge, this paper proposes a comprehensive optimization framework encompassing kernel optimization, memory access optimization, and scheduling optimization to accelerate MHA inference in PyTorch ecosystems on MT-3000. Kernel optimization focuses on low-level refactoring of critical operators like Linear and Softmax: Matrix tiling (e.g., $M_1=6$, $N_1=128$), loop unrolling, and outer product method design are employed to maximize computation-data transfer parallelism. For instance, the Linear operator kernel is redesigned to decompose input matrices into sub-blocks fitting the DSP's on-chip vector memory (AM), with double-buffering techniques enabling overlapping of computation and DMA transfers. Operator fusion further integrates scaling, matrix multiplication (Matmul), and Softmax into unified computational units, reducing intermediate data storage and achieving I/O overhead reduction. The optimized Linear operator achieves 1.53 TFLOPS per cluster, reaching 37.7% of theoretical peak performance (4.05 TFLOPS), delivering 5.34× speedup over NVIDIA V100 GPU for 1024×4096 matrix dimensions. Memory optimization restructures dataflow to reduce DDR bandwidth dependency: Weight matrices with reusability are broadcast and loaded once into on-chip memory via global shared memory (GSM), while phase-aware dataflow scheduling combined with cache management ensures high data reuse rates. For example, intermediate attention scores in Softmax computation are cached in GSM (6 MB per cluster), eliminating redundant DDR transfers. Scheduling optimization introduces row-granular block parallelism, partitioning MHA tasks into row-wise subtasks distributed across 24 DSP cores per cluster. Asynchronous DMA transfers hide data transfer latency, and dynamic load balancing ensures equitable task allocation. Experiments demonstrate excellent scalability: A 4-cluster configuration achieves 3.98× throughput improvement over single-cluster with merely 3.5% latency increase (sequence length $S=1024$). Systematic evaluations validate the framework's effectiveness: For typical large language model configurations (embedding dimensions 4096/8192, 32/64 heads), the optimized MHA mechanism achieves up to 23.53× acceleration over NVIDIA V100 GPU ($S=128$, 64 heads), maintaining 7.87× performance advantage in long-sequence scenarios ($S=1024$). A parameter search space and memory constraint model provide general guidance for balancing tiling strategies and on-chip resource limitations. This study not only delivers practical solutions for efficient deployment of bandwidth-constrained AI tasks on MT-3000, but also lays technical foundations for domestic platforms like Tianhe supercomputers to support large-scale parametric models. Future work will explore hybrid-precision quantization, sparse computation acceleration, and dynamic pipeline parallelism to further unleash MT-3000's potential in next-generation AI inference.

Keywords MT-3000; multi-head attention; performance optimization; PyTorch; speedup

1 引 言

深度学习作为人工智能领域的核心技术,近年来在计算机视觉、自然语言处理和语音识别等领域取得了突破性进展[1]。其核心思想是通过构建多层神经网络,自动从海量数据中学习复杂的特征表示。随着模型规模的不断扩大和计算需求的激增,传统的循环神经网络(RNN)和卷积神经网络(CNN)在处理长序列数据时逐渐暴露出计算效率低、难以捕捉长距离依赖关系等问题。

为了克服这些局限,Transformer 模型应运而生。其核心组件是自注意力机制(Self-Attention)

和多头注意力机制(Multi-Head Attention)。它们使模型在处理序列数据时能够更加灵活和高效,通过注意力机制捕捉序列内部的长距离依赖关系。与传统的循环神经网络不同,Transformer 模型不依赖于逐时间步展开序列数据,而是利用注意力机制并行处理序列中的所有位置关系。这种并行处理能力显著提升了模型处理复杂任务的能力,尤其是在涉及长距离依赖的任务中[2]。

随着模型参数规模从十亿级向万亿级迈进,传统硬件加速器(如 GPU)在能效比和带宽利用率方面面临严峻挑战。在此背景下,以 MT-3000 为代表的数字信号处理器(DSP)凭借其低功耗、高性能以及针对特定应用的优化能力,成为一种具有潜力的替代方案。

在高性能计算(HPC)领域,能源消耗和功耗控制已成为关键考量因素。技术的进步推动了将数字信号处理器(DSP)集成到通用 HPC 系统中的趋势。DSP 以其 VLIW(超长指令字)架构和软件控制的片上暂存器内存而闻名,能够高效利用指令级并行性,同时确保执行过程的实时性和可预测性[3]。这些特性使 DSP 在 HPC 和 AI 应用中具有独特优势,既能实现高性能计算,又能显著降低能源消耗。然而,尽管已有研究聚焦于优化多核 DSP 上的通用矩阵乘法(GEMM)以提升计算效率[4-6],这些工作大多集中在计算优化层面,而忽略了带宽不足时的优化策略。

本文以 MT-3000 处理器为例,旨在提升多核 DSP 处理器上 MHA(Multi-Head Attention)的推理效率,特别是在带宽不足时。MT-3000 是一款专为高性能计算(HPC)任务设计的 CPU-DSP 异构多核处理器,具备卓越的计算性能和架构优势,能够高效处理复杂的计算密集型任务,同时保持高度的灵活性和可扩展性[7]。其双精度能效达到 45.4 GFLOPS/W,较 NVIDIA V100 GPU 提升了 62%。然而,在 MT-3000 上实现 MHA 时,低速带宽无法满足高计算速度的需求,当运算从计算密集型转变为 IO 密集型任务时,现有优化策略未能有效解决带宽瓶颈对计算速度的限制[5-6]。此外,对于 MHA 机制,多个头的注意力得分以及注意力权重在主存与片上缓存之间的频繁 IO 消耗远超计算本身所需时间,进一步加剧了性能瓶颈。

针对上述挑战,本文提出了一种面向 MT-3000 异构多核处理器的 MHA 机制优化方案。通过深度融合 MT-3000 的架构特性,从算子优化、访存优化和调度优化三个维度进行系统性设计,显著提升了 MHA 在带宽不足场景下的计算效率。本文的主要贡献包括:

算子优化:通过矩阵分块、内核级优化和算子融合技术,最大化计算与数据传输的并行性,使 Linear 算子在单簇上的峰值性能达到 1.53 TFLOPS,占理论峰值的 37.7%,相较 NVIDIA V100 GPU 实现最高 5.34 倍加速。

访存优化:利用广播机制和全局共享内存(GSM)优化数据流,降低主存带宽依赖,通过分阶段数据流调度和片上缓存管理减少 IO 开销。

调度优化:采用行粒度分块并行策略,隐藏数据传输延迟,确保高效计算执行。

系统验证:在典型大语言模型配置下(嵌入维度 4096/8192,头数 32/64),优化后的 MHA 机制相较 NVIDIA V100 GPU 实现最高 23.53 倍加速,并在单节点多簇环境中展现出良好的扩展性。

2　相关工作

2.1　MT-3000 异构多核处理器

MT-3000 是一款由国防科技大学设计并实现的异构多区域处理器,专为高性能计算(HPC)而设计[7]。MT-3000 包含 16 个通用 CPU、96 个控制核心和 1536 个加速器核心。这些核心被分为两个主要区域:通用区域(General Purpose Zone)和加速区域(Acceleration Zone)。MT-3000 处理器的通用区域包含 16 个通用 CPU 核心,每个核心配备私有的两级缓存(L1 和 L2)。L2 缓存容量为 512 KB,采用 16 路组相联方式,支持随机替换策略,并通过目录式缓存一致性协议(MESI 算法)确保数据一致性。L1 缓存包含在 L2 缓存中,作为 L2 的副本,支持数据同步。整个处理器被进一步细分为四个自治的簇(Clusters),每个簇包含 4 个 CPU、24 个控制核心和 384 个加速核心,而一个控制核心与 16 个加速核心组成一个加速器阵列,即一个 DSP 核,因此一个簇可以视作 4 个通用 CPU 和 24 个 DSP 核组成,此外还包括全局共享内存 GSM 和片外 DDR 内存空间,所有簇通过分层互连网络连接。结构如图 1 所示。

CPU 负责管理加速器阵列以及执行 I/O、通信和计算任务。DSP 核是主要的计算能力提供者,配备标量处理单元(SPU)和向量处理单元(VPU),并拥有自己的片上存储资源,包括 64 kB 的标量内存

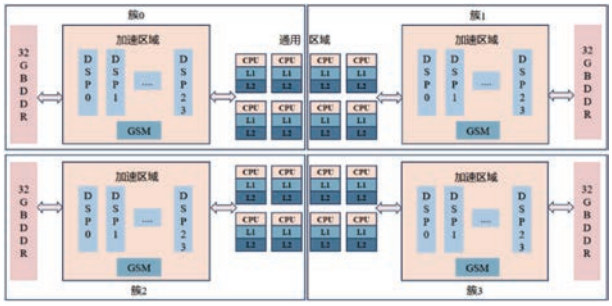

图 1 MT-3000 架构图

(SM)和 768KB 的向量内存(AM)。SM 与 AM 之间的数据通信通过寄存器(SVR)完成。每个 DSP 核支持 1024 位 SIMD 指令和超长指令字(VLIW)架构。控制核心处理加速阵列内的总体控制流,并将 SIMD 指令卸载到所有 16 个加速核心,每个加速核心包含 3 个可以并行的浮点乘加单元(VMAC)。每个簇中的 24 个 DSP 核可以共享一块 6MB 的全局共享内存(GSM),为跨核心的数据共享和通信提供了更大的空间和灵活性。单个 DSP 的结构如图 2 所示。

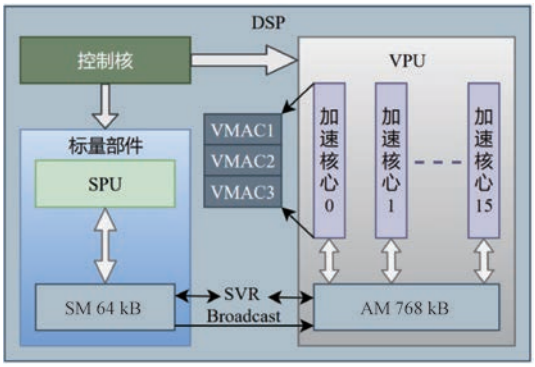

图 2 单个 DSP 结构

MT-3000 的存储结构为多级设计,带宽最慢的是主存 DDR,其次是 GSM,最快的是片上缓存 AM 和 SM。MT-3000 处理器中的 DMA 技术是其高性能计算架构的核心组成部分,它通过直接在内存和处理器之间传输数据来减轻 CPU 的负担并提升数据传输速率。DMA 支持加速器核心快速加载和存储数据,同时利用向量内存和标量内存等专用内存区域,满足不同计算单元的高速数据处理需求。DMA 促进了加速器核心之间的数据共享,它的异步操作允许数据传输与 CPU 的其他任务并行。此外,MT-3000 还可以采用广播的方式将数据从 DDR 或者 GSM 传输给片上内存。

2.2 Multi-Head Attention

自注意力机制(Self-Attention)和多头注意力机制(Multi-Head Attention)是深度学习中 Transformer 模型的核心组成部分,它们允许模型在处理序列数据时更加灵活和强大[8]。与传统的循环神经网络不同,Transformer 模型不依赖于序列数据逐时间步的展开,而是利用注意力机制来捕捉序列内部的长距离依赖关系。

Self-Attention 机制如图 3 所示。该机制允许序列中的每个元素评估其与序列中其他所有元素的关系,生成一个加权的输出表示。这个过程是通过创建查询(Q)、键(K)和值(V)向量,其中 Q、K、V 正是通过 Self-Attention 的输入进行线性变换得到的。然后计算这些向量间的注意力分数并进行归一化处理实现的。归一化后的注意力权重用于对值向量进行加权求和,从而得到每个位置的输出表示。

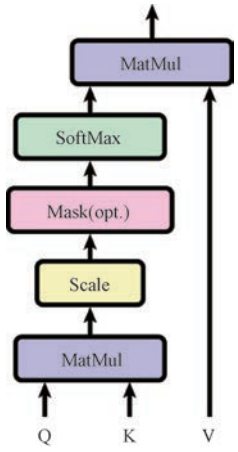

图 3 Self-Attention 机制

这种机制使得模型能够在处理当前元素时,考虑到序列中其他元素的重要性。

Multi-Head Attention 机制如图 4 所示。它是自注意力的扩展,通过并行运行多个自注意力层来

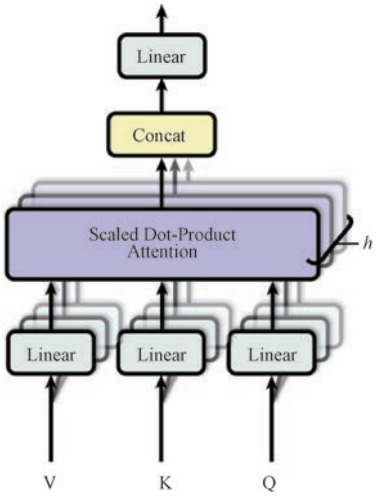

图 4 Multi-Head Attention 机制

增强模型的表达能力,每个层称为一个“头”。每个头学习到的是序列的不同表示子空间中的信息。具体来说,模型将查询、键和值通过不同的线性变换投影到不同的子空间中,然后在每个子空间中独立地计算注意力输出。这些输出随后被拼接起来,并通过另一个线性层进行转换,以生成最终的输出。这样做的好处是能够让模型同时从多个角度理解数据,捕获更丰富的信息。

总结来说,自注意力机制使模型能够关注序列内部的元素关系,而多头注意力机制则进一步提升了这种能力,使模型能够从多个表示子空间中同时学习,增强了模型对复杂数据结构的建模能力。

近年来,许多研究致力于优化多头注意力机制(MHA),主要方法包括模型压缩技术(如稀疏化[8-11]和量化[12-13]),这些方法通过降低计算复杂度来减少计算量。然而,这些方法以牺牲模型精度为代价,从而影响整体性能。其他优化策略包括子图融合[14]和 MHA 结构优化[15-17],这些方法主要面向传统硬件加速器(如 GPU),利用其高带宽显存和 Tensor Core 加速能力。此外,针对长序列任务的优化方法[18-19]也在不断探索。然而,这些方法大多基于 GPU 的高带宽特性,难以直接迁移到 DSP 架构上,因此现有的 MHA 优化方法在 DSP 上可能无法取得预期的性能提升,需要针对 DSP 的特定架构特点进行专门优化。

2.3　PyTorch

PyTorch 是 Facebook 人工智能研究团队开发的开源深度学习框架,自 2016 年发布以来,以其易用性、灵活性和动态计算图(DCGs)迅速赢得了广泛的认可。动态计算图的设计为研究和快速原型开发提供了极大的灵活性,允许在运行时动态调整计算流程[20]。PyTorch 的 API 设计直观,与 Python 高度集成,简化了学习和使用过程,并且能够直接利用 Python 的标准调试工具进行调试。

随着 2019 年 1.0 版本的推出,PyTorch 不断扩展其功能,包括对 ONNX 的支持、分布式计算能力以及 C++前端的接口,这进一步增强了其应用的广泛性。活跃的社区支持为 PyTorch 提供了丰富的资源和预训练模型,例如 ResNet、VGG、Inception 等,极大地加速了新项目的启动和开发。

在数据处理方面,PyTorch 的张量操作与 NumPy 相似,同时支持 GPU 加速,使得用户能够通过简单的 API 在 CPU 和 GPU 之间灵活迁移张量和模型,实现高效的计算加速。PyTorch 的自动求导机制(autograd)进一步简化了深度学习模型的训练过程,使得梯度计算和优化变得轻松便捷。

PyTorch 对 Attention 机制全面支持,它提供的“nn. MultiheadAttention”模块是一个高度优化的实现,允许模型在处理序列数据时并行地关注信息的不同部分。这个模块不仅支持不同头的注意力权重的并行计算,还包括残差连接和层归一化,这些都是现代 Transformer 模型不可或缺的关键特性。PyTorch 的灵活性和强大功能使得研究人员和开发者能够轻松自定义和扩展注意力机制,以适应各种复杂的机器学习任务。通过 PyTorch,实现和训练包含 Attention 机制的模型变得更加高效和直观,进一步巩固了其在深度学习领域的重要地位。

然而,PyTorch 的默认实现主要针对 GPU 和通用 CPU 等常用架构,未能充分利用 MT-3000 的 VLIW 指令集和片上存储层次结构。因此,针对 MT-3000 的手动算子优化和数据流设计成为提升性能的关键。

综上所述,现有研究在针对 MT-3000 等低功耗 DSP 平台的 MHA 优化方面存在显著空白,本文旨在填补这一领域的研究空白。

3　动　机

在 MT-3000 上加速 MHA 时,首先考虑算子的加速,算子可以根据计算密度(OI)[21]进行分类。计算密度定义如公式(1)所示。

$$OI=\frac{Number\ of\ Operations}{Number\ of\ Memory\ Accesses} \tag{1}$$

其中,*Number of Operations* 表示计算操作数量,*Number of Memory Accesses* 表示内存访问次数。根据 Roofline 模型分析[22],不同的算子可以分为计算受限和 I/O 访存受限两类。低计算密度的算子(如 softmax、scale 等)属于 I/O 访存受限,而高计算密度的算子(如 GEMM、Linear)在带宽充足的情况下可以变为计算受限。然而,在现代 GPU 架构中,计算速度已经远远超过了 I/O 访存的速度[23-24],这使得 I/O 访存成为性能瓶颈的主要来源。

对于 MT-3000 处理器,单个 DSP 核心的运行频率为 1.8 GHz,单精度浮点乘加的理论峰值性能为 172.8 GFLOPS,其计算如公式(2)所示。

$$P=F\times N_{vpe}\times N_{vmac} \tag{2}$$

其中,F 表示 DSP 的频率,N_{vpe} 表示单个 DSP 中的加速核个数,N_{vmac} 表示一个加速核中的乘累加单

元个数。每个簇包含 24 个 DSP 核心，单精度浮点乘加的理论峰值性能可达 4.05 TFLOPS，但其 DMA 速度相对较低，DDR 到片上内存的理论带宽仅为 13 GB/s 左右。与现代 GPU 采用的 HBM(高带宽内存)相比[25]，MT-3000 的带宽显著较低，导致在几乎所有算子上都表现出 I/O 受限的特征。

因此，如何提高带宽利用率和减少 I/O 访存数据量成为优化 MT-3000 性能的关键。片上全局共享内存(GSM)的大小仅为 6 MB，而加速器内存(AM)的空间更小，这进一步加剧了内存访问的挑战。为了应对这一问题，我们通过数据流设计，利用 DDR 和 GSM 的广播机制来减少 DMA 访存次数，从而提高数据重用率。此外，算子融合是减少 I/O 访问的有效手段，通常由编译器自动完成[26]。然而，MT-3000 的编译器目前对此支持不足，因此需要手动实现算子融合以优化性能。

4 优化方案

在针对 MT-3000 平台的 MHA 机制优化中，我们实施了一套综合策略，该策略分为三个模块：算子优化、访存优化和算子调度优化。这种策略确保了从数据流到计算执行的每个环节都得到了精细的调整来优化整体性能。

算子优化针对 DSP 上的算子进行定制化优化，尤其是对 Linear 和 softmax 等关键操作。通过矩阵分块、内核优化以及算子融合等技术，本文提升了关键算子的执行效率。算子调度优化负责安排每个头内的算子执行流程，以及在不同计算簇上分配的具体策略。这一环节通过精确的调度算法，确保了资源的合理分配和计算任务的高效执行。访存优化致力于减少数据传输过程中的延迟，通过优化数据在片上存储资源(如 AM 和 SM)的布局，以及充分利用全局共享内存(GSM)的带宽优势，并结合 DMA 传输的特性，有效降低了访存开销，从而进一步提升了计算效率。

如图 5 所示，以上三个模块的优化工作是相互依赖和互补的，共同构成协调一致的优化流程，该流程提高了 MHA 在 MT-3000 处理器上的计算效率。

4.1 算子优化

如图 6 所示的计算流程图清晰地展示了 Linear、矩阵乘法(matmul)和 softmax 算子在整个计算过程中的核心地位。其中，B 表示批量大小(Batch)，S 表示序列长度(Sequence length)，D 表示嵌入维度(embedding Dimension)。为了显著提升整体计算效率，我们对这些关键算子进行了深度优化。此外，针对可以合并执行的算子，我们采用了融合设计策略，以减少数据传输，从而进一步增强了计算性能。

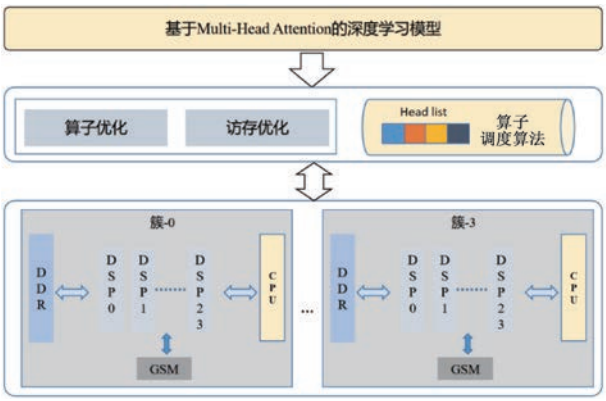

图 5 基于 MT-3000 的 MHA 机制优化方案

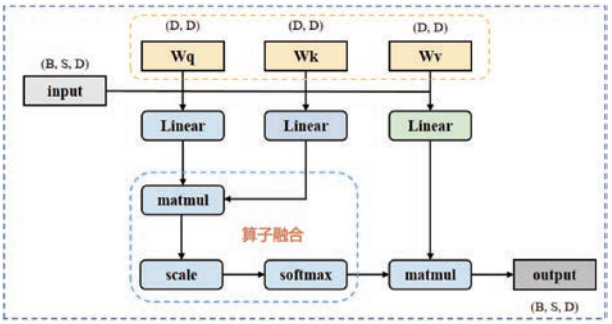

图 6 MHA 计算图

4.1.1 Linear 算子优化

Linear 算子的作用是对输入矩阵 $\boldsymbol{X}$ 进行线性映射，计算公式如公式(3)所示。

$$\boldsymbol{X}=\begin{bmatrix} x_{11},x_{12},\cdots,x_{1K} \\ x_{21},x_{22},\cdots,x_{2K} \\ \cdots \\ x_{M1},x_{M2},\cdots,x_{MK} \end{bmatrix},$$

$$\boldsymbol{W}=\begin{bmatrix} w_{11},w_{12},\cdots,w_{1N} \\ w_{21},w_{22},\cdots,w_{2N} \\ \cdots \\ w_{K1},w_{K2},\cdots,w_{KN} \end{bmatrix}, \tag{3}$$

$$\boldsymbol{b}=[b_1,b_2,\cdots,b_N],$$

$$\boldsymbol{Y}=\boldsymbol{X}\boldsymbol{W}+\boldsymbol{b}$$

其中，$\boldsymbol{W}$ 表示权重，$\boldsymbol{b}$ 表示偏置。基于 Goto[27] 的 Linear 算法已经被很多线性代数库使用，如 BLIS 框架[28]。然而，由于 MT-3000 处理器的计算性能和主存带宽不匹配，Linear 算子在该平台上表现出明显的 I/O 密集型特征，成为性能瓶颈。

为解决这一问题,我们提出了一种优化算法。令 $\boldsymbol{X}[M,K]$表示输入矩阵,$\boldsymbol{W}[K,N]$表示权重矩阵,$\boldsymbol{b}[1,N]$表示偏置向量。由于我们的内核设计特性(算法 2 所示),当权重矩阵的列数 N 较小时,在 N 维度上并行会导致 24 个 DSP 核之间的负载不均衡。因此,我们选择在输入矩阵的行维度 M 上进行并行计算,以确保各 DSP 核的负载均衡,从而显著提高整体计算效率。优化后的算法如算法 1 所示。

算法 1. Linear 优化算法

输入:矩阵 $\boldsymbol{X}[M,K]$,矩阵 $\boldsymbol{W}[K,N]$,向量 $\boldsymbol{b}[1,N]$

输出:矩阵 $\boldsymbol{Y}[M,N]$

Begin

1. *Broadcast*(**DDR**−>24**AM**, b);
2. FOR $m_0=0$; $m_0<M$; $m_0+=M_g$ do
3. FOR $k_0=0$; $k_0<K$; $k_0+=K_g$ do
4. *DMA*(**DDR**−>**GSM**, $\boldsymbol{X}([m_0: m_0+M_g), [k_0: k_0+k_g))$−>$X_g$);
5. FOR $m_2=0$; $m_2<M_g$, $m_2+=P*M_2$ do
6. FOR $n_2=0$; $n_2<N$; $n_2+=N_2$ do
7. *DMA* (**DDR**−>$\boldsymbol{P}*$**AM**, $Y([m_0+m_2+Pid*M_2: m_0+m_2+Pid*M_2+M_2), [n_2: n_2+N_2))$−>$\boldsymbol{Y}_2$);
8. FOR $k_2=0$; $k_2<K_g$; $k_2+=K_2$ do
9. *DMA*(**GSM**−>$\boldsymbol{P}*$**SM**, $\boldsymbol{X}_g([m_2+Pid*M_2: m_2+Pid*M_2+M_2), [k_2: k_2+K_2))$−>$\boldsymbol{X}_2$);
10. *Broadcast*(**DDR**−>24**AM**, $\boldsymbol{W}([k_0+k_2: k_0+k_2+K_2), [n_2: n_2+N_2))$−>$\boldsymbol{W}_2$);
11. FOR $m_1=0$; $m_1<M_2$; $m_1+=M_1$ do
12. FOR $n_1=0$; $n_1<N_2$; $n_1+=N_1$do
13. $\boldsymbol{X}_2([m_1: m_1+M_1), [0:K_2))$−>$\boldsymbol{X}_1$, $\boldsymbol{W}_2([0: K_2), [n_1: n_1+N_1))$−>$\boldsymbol{W}_1$, $\boldsymbol{Y}_2([m_1: m_1+M_1), [n_1: n_1+N_1))$−>$\boldsymbol{Y}_1$;
14. *Kernel*($\boldsymbol{X}_1*\boldsymbol{W}_1+\boldsymbol{b}_1=\boldsymbol{Y}_1$);
15. *DMA*($\boldsymbol{P}*$**AM**−>**DDR**, $\boldsymbol{Y}_2$−>$\boldsymbol{Y}([m_0+m_2+Pid*M_2: m_0+m_2+Pid*M_2+M_2), [n_2: n_2+N_2))$);

End

在算法 1 中,首先将偏置向量 $\boldsymbol{b}$ 全搬入 AM 中。再将输入矩阵 $\boldsymbol{X}$ 分批搬入 GSM,搬入大小为 $\boldsymbol{X}_g[M_g,K_g]$,以充分利用 GSM 作为片上缓存的高带宽特性;接着将 $\boldsymbol{X}_g$ 中的数据进一步拆分出 P 个子矩阵 $\boldsymbol{X}_2[M_2,K_2]$,并将其搬运至 SM。并从权重矩阵 $\boldsymbol{W}$ 中取出对应的子矩阵 $\boldsymbol{W}_2[K_2,N_2]$,将其广播到 P 个 DSP 的 AM 空间中,计算得到 P 个 $\boldsymbol{Y}_2[M_2,N_2]$,写回到主存对应位置。为了进一步优化计算效率,$\boldsymbol{X}_2$ 和 $\boldsymbol{W}_2$ 分成多个 $\boldsymbol{X}_1$ 和 $\boldsymbol{W}_1$ 进行计算,每个 $\boldsymbol{X}_1$ 和 $\boldsymbol{W}_1$ 计算得到 $\boldsymbol{Y}_1$ 的过程称之为内核。为了提高 Linear 算子的计算效率,本文采用了外积法来设计内核,通过最大化计算与数据传输的并行性,显著提升了性能。详细的内核设计如算法 2 所示。

算法 2. Linear 内核设计

输入:矩阵 $\boldsymbol{X}_1[M_1,K_1]$,矩阵 $\boldsymbol{W}_1[K_1,N_1]$

输出:矩阵 $\boldsymbol{Y}_1[M_1,N_1]$

Begin

1. *VPU_load* $\boldsymbol{Y}_1$ to $VR\_y[0: M_1),[0: N_1/32)$;
2. FOR $k=0$; $k<K_1$; $k++$do
3. *//Loop unrolled*
4. FOR $m=0$; $m<M_1$; $m++$do
5. *SPU_load* $\boldsymbol{X}_1$[m][k] to R_x[m];
6. *Broadcast* R_x[m] to VR_x[m];
7. *//Loop unrolled*
8. FOR $n=0$; $n<N_1/32$; $n++$do
9. *VPU_load* $\boldsymbol{W}_1[k, n]$ to $VR\_w[n]$;
10. *//Loop unrolled*
11. FOR $m=0$; $m<M_1$; $m++$do
12. FOR $n=0$; $n<N_1/32$; $n++$do
13. *Mula* $VR\_x[m]$, $VR\_w[n]$, $VR\_y[m][n]$−>$VR\_y[m][n]$;
14. *VPU_store* $VR\_y[0: M_1),[0: N_1/32)$ to $\boldsymbol{Y}_1$;

End

在算法 2 中,通过外积法首先将每个内核的输出 Y_1 固定在向量寄存器 VR_y 中,避免其频繁加载以及存储带来的开销。接着采用内部循环展开技术,对计算过程、输入矩阵 $\boldsymbol{X}_1$ 的加载以及权重矩阵 $\boldsymbol{W}_1$ 的加载到寄存器的操作进行了解耦和并行化处理。这种处理方式显著降低了数据传输的开销,同时提升了计算效率。

为了最大化计算与数据传输的并行性,我们对 $\boldsymbol{X}_2$ 和 $\boldsymbol{W}_2$ 的传输与计算过程实施了双缓冲技术。在优化过程中需要确保每个分块的数据量不超过片上内存的限制。其中 $\boldsymbol{X}_2$ 需要两个缓冲区,因此其占用的大小不能超过 SM 容量的一半,这一约束条件由公式(4)给出。

$$2\times D\_Size\times M_2\times K_2\leqslant SM\_Size \tag{4}$$

其中,D_Size 表示单个数据大小,本文采用 FP32,因此 D_Size 为 4B,SM_Size 表示 SM 的容量大小。$\boldsymbol{W}_2$ 和 $\boldsymbol{Y}_2$ 也需要两个缓冲区来存放,b 需要一个缓冲区存放,因此这三者同时需要的缓冲区应该小于等于 AM 的大小,需要满足公式(5)的约束。

$$(2\times K_2\times N_2+2\times M_2\times N_2+N)\times D\_Size\leqslant AM\_Size \tag{5}$$

而 $\boldsymbol{X}_g$ 的大小需要满足 GSM 的容量约束,如公式(6)所示。

$$D\_Size \times K_g \times M_g \leqslant GSM\_Size \quad (6)$$

为了提高计算效率并符合主流大语言模型(如 Llama[29] 和 Qwen[30])的架构特性,这些模型通常采用单头嵌入维度为 128 的设计,因此本文将 Linear 内核参数配置为 $M_1=6$ 和 $N_1=128$。在参数选择上,M_2 应尽可能为 M_1 的整数倍,N_2 应为 N_1 的整数倍,而 K_2 应尽量选择为 2 的幂次。此外,K_g 和 M_g 应尽可能为 K_2 和 M_2 的整数倍,以确保计算过程的高效性。为了降低数据搬运的总开销,K_g、N_2 和 M_2 需要尽可能选择较大的值。然而,由于这些参数之间存在相互制约关系,需要在它们之间进行合理的折中。因此,本文通过综合考虑计算效率和存储资源限制,经过计算得到了表 1 所示的参数搜索空间,为后续优化提供了指导。

表 1 参数搜索空间

序号	M_1	N_1	M_g	K_g	N_2	K_2	M_2
1	6	128	288	4k	128	512	12
2	6	128	720	2k	256	256	30
3	6	128	720	2k	512	128	30
4	6	128	1440	1k	256	128	60
5	6	128	1440	1k	512	64	60

4.1.2 Softmax 算子优化

在 MHA 中,softmax 算子将缩放后的点积结果转换为归一化的概率分布。其计算公式如下所示:

$$softmax(z_i)=\frac{e^{z_i}}{\sum e^{z_i}} \quad (7)$$

对于 Z 中的每个元素 z_i,计算其指数 e^{z_i},然后除以所有 e^{z_i} 的和。由于 MHA 中 softmax 是以行为单位进行的,因此其优化如算法 3 所示,假设此时 $\boldsymbol{Y}[M,K]$ 位于 AM 之中。在 softmax 算子的优化中,本文按照行进行拆分,每 32 行作为一个处理单位。对于每个单位,首先计算每个元素的指数值,然后通过循环展开的方式加速每行的向量求和操作。

算法 3. softmax 优化算法

输入:矩阵 $\boldsymbol{Y}[M,N]$

输出:矩阵 $softmax(\boldsymbol{Y}[M,N],\ dim=-1)$

Begin

1. FOR $m_2=0$; $m_2<M$; $m_2+=32$ do
2. //*Loop unrolled*
3. FOR $m_1=m_2$; $m_1<\min(M,\ m_2+32)$; m_1++;
4. *VPU_load* $\boldsymbol{Y}[m_1][0]$ to $VR\_sum[m_1-m_2]$ & *element-wise exponent*;
5. FOR $n=1$; $n<N/32$; $n++$;
6. //*Loop unrolled*
7. FOR $m_1=m_2$; $m_1<\min(M,\ m_2+32)$; m_1++;
8. *VPU_load* $Y[m_1][n]$ to $VR\_p[m_1-m_2]$ & *element-wise exponent*;
9. $VR\_sum[m_1-m_2]+=VR\_p[m_1-m_2]$;
10. *VR_sum vector reduction & element-wise reciprocal & transfer to R_sum*;
11. *Broadcast R_sum to VR_sum*;
12. FOR $n=0$; $n<N/32$; $n++$;
13. //*Loop unrolled*
14. FOR $m_1=m_2$; $m_1<\min(M_2,\ m_2+32)$; m_1++;
15. *VPU_load* $Y[m_1][n]$ to $VR\_p[m_1-m_2]$;
16. $VR\_p[m_1-m_2]\ *=VR\_sum[m_1-m_2]$;
17. *VPU_store* $VR\_p[m_1-m_2]$ to $\boldsymbol{Y}[m_1][n]$;

End

为了进一步优化性能,本文对 32 个 FP32 向量进行内部规约(vector reduction)求和。为了加速这一过程,我们采用手写汇编的方式嵌入内核,以实现高效的向量内部归约,具体的实现设计如算法 4 所示。

算法 4. 向量内部规约求和

输入:*VR_sum*[0:32]

输出:*R_sum*[0:32]

1. 使用 *bale* 指令对每个 *VR_sum* 寄存器的高低 32 位进行统一打包。
2. 每两个 *VR_sum* 寄存器一组进行相加。
3. 对相加后的 16 个 *VR_sum* 寄存器进行分组,每隔四个作为同一组,分成 4 组。
4. 使用混洗操作 VSTDW0M16 和 VSTDW1M16 将每组寄存器中的数据搬运到 AM 中,之后再加载回 *VR_sum* 寄存器。
5. 对每组 *VR_sum* 寄存器进行相加,得到每个向量的部分和。
6. 重复以上 4~5 步操作,最终得到 32 个浮点向量内部和的归约结果。

其中,VSTDW0M16 和 VSTDW1M16 指令的作用如表 2 所示。这两条指令是 MT-3000 处理器特有的向量混洗操作,其核心功能是将向量寄存器中的数据按特定模式重新排列后写入 AM。

表 2 VSTDW0M16 和 VSTDW1M16 混洗操作

指令	访存操作前向量寄存器对中的数据	访存操作后 AM 空间存放的数据
VSTDW0M16 VR1:VR0, *AR	VR0:{B0,B1,…B13,B14,B15}	{B0, B16, B32,
VSTDW1M16 VR7:VR6, *+AR[16]	VR1:{B16,B17,…B29,B30,B31}	B48, B1, B17,
	VR6:{B32,B33,…B45,B46,B47}	B33,B49,…B15,
	VR7:{B48,B49,…B61,B62,B63}	B31,B47,B63}

部分的汇编代码如表 3 所示。

表 3　算法 4 部分汇编代码样例

```
__asm__ (
        "smvaga.M1%0, AR0\n"
        "snop 1\n"
        "vldw *+AR0[0],VR0\n"
        "| vldw *+AR0[1],VR1\n"
        "vldw *+AR0[2],VR2\n"
        "| smvaga.M1%1, AR4\n"
        "| vldw *+AR0[3],VR3\n"
        "snop 8\n"
        "vstdw0m16 VR1:VR0, *+AR4[0]\n"
        "| vstdw1m16 VR3:VR2, *+AR4[16]\n"
        :
        :"r"(am_z1),"r"(am_z)
    );
```

4.1.3　算子融合

在深度学习模型的实现中,算子融合是一种常用的优化技术,其核心思想是通过减少数据在不同存储层次之间的移动来提升计算效率。如图 6 所示,在 MHA 的实现中,矩阵乘法(matmul)是一个高计算密度的算子,其后通常紧跟着缩放(scale)和 softmax 这两个低计算密度的算子。为了优化性能并减少不必要的数据传输,我们采用了算子融合策略,将这三个算子合并为一个统一的算子。

具体来说,本文将缩放(scale)运算直接嵌入到 matmul 的内核计算之后,在计算得到注意力得分后立即进行 scale 操作,并在此基础上继续计算 softmax 的逐元素指数部分。随后,利用算法 3 中的 softmax 内核加速剩余的归一化操作。通过这种紧耦合的设计,避免了将注意力得分在不同存储层次之间来回搬运,从而消除了低速 DDR 带宽带来的延迟,并充分利用高速片上缓存来提升整体计算性能。

4.2　算子调度优化

在进行算子调度优化的过程中,我们首先深入分析了 MHA 的计算细节,如图 7 所示。

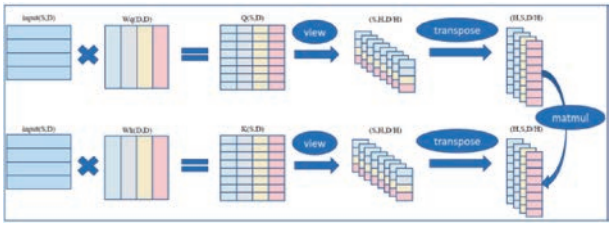

图 7　MHA 的计算流程分解

通过图 7,可以清晰地看到每个头的计算流程细节。具体来说,输入数据经过 Linear 层得到键(K)、查询(Q)和值(V)之后,通过 view 和 transpose 操作,进而被组织成多个注意力头。进一步分析发现,每个头的 K,Q,V 实际上是将原始的 K,Q,V 按列分割得到的。基于这一发现,本节提出一种基于 MT-3000 架构的 MHA 优化策略,通过分阶段数据流调度与片上缓存管理,显著降低 I/O 访存开销与空间复杂度,提升长序列场景下的计算效率。由于片上缓存的有限,我们需要在行的粒度上进行分块,对于单个头而言,我们的算法设计如下。

算法 5. MHA 分阶段调度优化

输入:输入矩阵 $Input[S,D]$,权重矩阵 $W_q[D,D]$,$W_k[D,D]$以及 $W_v[D,D]$

输出:矩阵 $Output[S,D]$

Begin

1. 通过算法 1 计算得到 Q、K、V。
2. 将 K 转置为 K^{T}。
3. 将 Q_i 划分为行块(M 行 Q_i)加载至 SM,将 $K_i{}^{\mathrm{T}}$ 按列分块(N 列)加载至 AM,经过迭代得到 M 行注意力得分,将其部分存入 GSM,另一部分存入 AM。
4. 对注意力得分进行归一化之后得到注意力权重。
5. 将注意力权重搬运至 GSM。
6. 将 M 行注意力权重与 V_i 进行 $matmul$ 运算,得到单个头的最终输出 O_i 的 M 行。结果返回 DDR。
7. 重复步骤 3～6,计算所有头的结果存回 DDR,得到 $Output$。

End

通过分阶段调度优化策略,我们以单个注意力头为基本调度单元,采用行级分块粒度对计算流程进行精细化调度,充分发挥片上缓存的高带宽优势。本文将整个计算流程划分为四个关键阶段:矩阵转置、注意力得分计算、注意力权重生成以及最终输出生成。通过流水线调度机制,将注意力得分矩阵与权重矩阵的传输过程完全隐藏,这种深度优化的流水线设计有效消除了显式数据传输带来的额外延迟,使得整体 I/O 开销得到显著压缩。

4.3　访存优化

访存优化的核心目标是减少数据传输量,这通过优化片上存储资源的布局、充分利用全局共享内存(GSM)的高带宽特性,以及有效利用 DMA 传输来实现。这种优化策略显著降低了 IO 访存的数据量,并提升了整体的计算效率。优化前后的数据传输过程对比如图 8 所示。

在优化之前,数据的传输路径是从 DDR 内存直接到 AM/SM,完成计算后再将结果传回 DDR。测试表明,DDR 与 AM 之间的 DMA 传输速率为

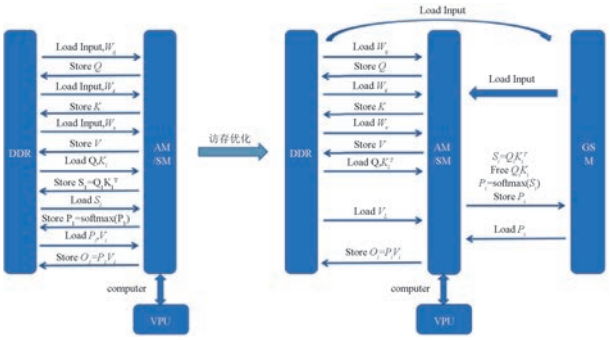

图 8 访存优化前后数据传输对比

13GB/s，而 AM 与 GSM 之间的传输速率则高达 160GB/s，存在量级上的差距。基于这一现象，我们提出了一种策略，将可以复用的数据以及中间结果存储在 GSM 和 AM 中，并在计算完成后立即释放，从而减少 DDR 与 AM 之间的数据传输次数。这种策略的调整使得数据传输更多地利用了 AM 与 GSM 之间的高速通道，显著优化了整体访存效率。

例如计算 QKV 阶段，其访存优化前后的对比如表 4 所示。优化之前，计算 QKV 的每个部分都需要将 Input 从 DDR 搬运 N/N_2 次到 SM，这在 Input 规模较大时会导致显著的访存开销。而优化之后只需要搬运一次到 GSM，之后从 GSM 进行搬运即可。此外，本文的分阶段调度策略通过隐藏注意力得分和注意力权重的传输，进一步优化了整体访存效率。通过这些优化措施，我们成功减少了不必要的数据传输，提升了计算性能。

表 4 QKV 计算访存优化前后数据传输对比

矩阵	Input(DDR→GSM)	Input(GSM→SM)	W_{qkv}(DDR→AM)	QKV(DDR→AM)	QKV(AM→DDR)	Input(DDR→SM)
传输方式	DMA	DMA	Broadcast	DMA	DMA	DMA
优化前	0	0	$M/(P*M_2)$	K/K_g	K/K_g	N/N_2
优化后	1	N/N_2	$M/(P*M_2)$	K/K_g	K/K_g	0

5 实验结果与分析

本节通过系统性的实验评估验证了所提出的优化算法在 MT-3000 异构多核处理器上的性能表现。实验以处理器通用 CPU 核心的未优化实现以及 Nvidia V100 GPU 为性能基线，从算子微观开销分析、计算加速效果验证以及系统扩展性三个维度展开。首先，通过对 Linear 算子的计算流程进行细粒度分解，量化了内核计算、线程组创建和内存管理操作在不同输入规模下的时间占比，揭示了小规模场景下调度开销主导、大规模场景下计算密集型特征显著的关键规律。其次，基于典型大语言模型参数配置(嵌入维度 4096/8192，头数 32/64)，对优化后的 Linear 算子和 MHA 机制进行加速比测试。最后，通过单节点多簇并行实验验证了算法的扩展性，展现了良好的负载均衡特性与分布式计算潜力。上述实验从多维度证实了本文提出的算子融合、数据流调度与存储层次优化策略的有效性，为 MT-3000 处理器在复杂 AI 工作负载中的高效部署提供了实证依据。

5.1 Linear 算子的各部分开销测试

在本节中，我们首先对 MT-3000 处理器上 Linear 算子的实现进行时间开销分析。通过将整个计算流程分解为三个关键阶段：线程组创建(create group)、计算(calculate)以及内存申请与释放(malloc free)，系统量化了各阶段的耗时。针对不同规模的输入矩阵 $\boldsymbol{X}$ 和权重矩阵 $\boldsymbol{W}$，表 5 展示了各阶段的绝对耗时与总执行时间。不同规模矩阵 Linear 算子的各部分时间占比如图 9 所示。

表 5 不同规模矩阵 Linear 的各部分时间 (单位：ms)

矩阵规模	calculate	create group	malloc free	Total time
$\boldsymbol{X}$(128×128)，$\boldsymbol{W}$(128×128)	0.167	0.301	0.193	0.661
$\boldsymbol{X}$(256×256)，$\boldsymbol{W}$(256×256)	0.195	0.312	0.204	0.711
$\boldsymbol{X}$(1024×1024)，$\boldsymbol{W}$(1024×128)	0.329	0.317	0.226	0.872
$\boldsymbol{X}$(512×512)，$\boldsymbol{W}$(512×512)	0.343	0.325	0.231	0.899
$\boldsymbol{X}$(1024×2048)，$\boldsymbol{W}$(2048×128)	0.398	0.301	0.229	0.928
$\boldsymbol{X}$(1024×2048)，$\boldsymbol{W}$(2048×256)	0.508	0.308	0.246	1.062
$\boldsymbol{X}$(1024×2048)，$\boldsymbol{W}$(2048×2048)	2.725	0.309	0.238	3.272
$\boldsymbol{X}$(1024×4096)，$\boldsymbol{W}$(4096×4096)	10.474	0.361	0.229	11.064

通过对不同规模矩阵下 Linear 算子的各部分时间开销分析，可以发现显著的性能特征差异。

在小规模矩阵场景下(如 $\boldsymbol{X}$(128×128)、$\boldsymbol{W}$(128×128))，线程组创建(create group)与内存申请/释放

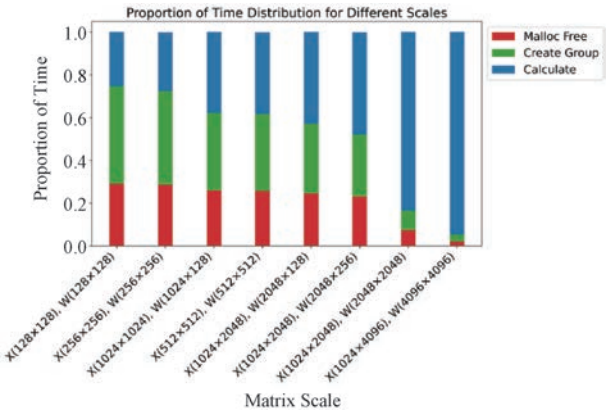

图 9　Linear 在不同规模下的各部分时间占比

(malloc free)操作的时间占比分别高达 45.54%和 29.20%,这表明线程初始化的固定开销在小规模计算中成为主要性能瓶颈。此时,计算流程的启动和管理成本占主导地位,导致优化潜力受限。然而,随着矩阵规模的扩大(如 $\boldsymbol{X}$(1024×4096)、$\boldsymbol{W}$(4096×4096)),线程组创建的时间占比降至 3.26%,内存申请/释放的占比亦显著降低。这一现象源于大规模计算中核心计算任务的耗时随数据量增长,而线程初始化等管理性操作的固定开销被稀释。因此,在大规模场景下,计算成为性能主导因素,而线程初始化的相对影响显著减弱,因此大规模场景则更依赖计算内核与数据流设计的深度优化。

我们通过公式(8)对 Linear 算子的计算性能进行定量评估。假设 $\boldsymbol{X}$ 的规模是$[M,K]$,$\boldsymbol{W}$ 的规模是$[K,N]$。

$$Per=\frac{MKN}{1024^3}\times\frac{1000}{time}GFLOPS \tag{8}$$

根据公式,在小规模场景中,如 $\boldsymbol{X}$(128×128)、$\boldsymbol{W}$(128×128),实测性能仅为 11.72 GFLOPS。这主要源于小规模计算无法有效实现负载均衡,且内核层面的优化空间未得到充分挖掘。随着矩阵规模扩大,性能呈现显著提升:当输入维度增至 $\boldsymbol{X}$(1024×2048)、$\boldsymbol{W}$(2048×2048)时,实测性能跃升至 1467.89GFLOPS;在更大规模场景 $\boldsymbol{X}$(1024×4096)、$\boldsymbol{W}$(4096×4096)下,性能进一步达到 1527.74 GFLOPS,达到单簇理论峰值性能(4.05 TFLOPS)的 37.7%。这表明,本文提出的分块策略、双缓冲技术与 VLIW 指令级优化有效缓解了主存带宽限制。

5.2　Linear 算子的加速比测试

本节通过对比实验评估了 Linear 算子在 MT-3000 处理器上的加速性能。实验采用 PyTorch 框架,以 NVIDIA V100 GPU 和 MT-3000 通用 CPU 为对照平台,在 batch size=4 的配置下进行全流程性能对比。为贴合大语言模型的实际应用场景,实验参数设置遵循主流模型架构:输入矩阵维度为 $\boldsymbol{X}[S,D]$,权重矩阵维度为 $\boldsymbol{W}[D,D]$,其中嵌入维度 $D=4096$(与 Llama 7B、Qwen 7B 等模型设计对齐),序列长度 S 从 128 至 1024 以 128 为步长递增。针对不同规模的矩阵 $\boldsymbol{X}$ 和权重 $\boldsymbol{W}$,Linear 在不同平台上的执行时间如表 6 所示。

表 6　Linear 在不同平台上的执行时间(单位:ms)

输入行数	DSP	V100	CPU
$S=128$	3.34	17.82	1196.34
$S=256$	4.24	21.85	2250.13
$S=384$	5.04	24.46	3395.38
$S=512$	6.07	27.86	4494.68
$S=640$	7.35	30.27	5794.08
$S=768$	8.79	33.46	6624.15
$S=896$	9.66	36.90	7936.59
$S=1024$	10.81	39.94	8740.79

通过加速比指标评估优化后的 MT-3000 DSP 与 NVIDIA V100 GPU 之间的性能差异。该指标直观反映了 MT-3000 在并行计算效率上的提升幅度。基于表 6 中不同输入规模下的实测数据,图 10 绘制了加速比随输入规模变化的趋势曲线。

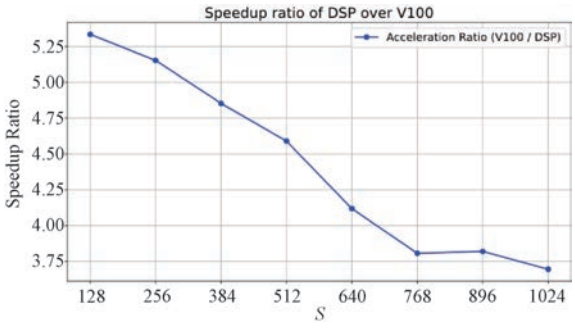

图 10　Linear 算子在不同规模下的加速比

结合图表可以看出,DSP 在各类矩阵规模下均显著优于 V100,但其优势随着矩阵行数(S)的增加呈现递减趋势。具体而言,在小规模矩阵($S\leqslant384$)时,DSP 的加速比稳定在 4.8×以上,最高可达 5.34×($S=128$),这主要得益于本文提出的优化机制,例如将数据传输与计算过程并行、高效的数据局部性利用和内核设计,使其在小规模并行任务中能够充分发挥算力优势。然而,在中大规模矩阵($512\leqslant S\leqslant1024$)时,加速比逐渐下降至 3.69×($S=1024$),这是由于随着计算量的增长,V100 的数据搬运到显存的时间占比下降,抵消了部分 DSP 的优化效果。尽管加速比有所下降,但 DSP 在绝对执行

时间上仍全面领先 V100，例如在 $S=1024$ 时，DSP 耗时仅为 V100 的 27%（10.81ms vs. 39.94ms）。这表明优化后的 DSP 更适合大语言模型中常见的宽维度（$D=4096$）线性层计算场景。而相对于单纯使用 CPU，当 $S=896$ 以及 $S=1024$ 时，加速比分别高达 821.59 倍以及 808.58 倍，DSP 展现出对 CPU 的压倒性优势，体现了我们优化的必要性。

5.3 单节点上的 MHA 加速测试

本节通过系统性实验评估 MHA 机制的单节点计算效能，并与 NVIDIA V100 GPU 进行对比分析，以验证算法的优化效果。实验采用实际应用场景中的典型配置：Batch size 固定为 4，分别构建 32 头（头维度 128）和 64 头（头维度 128）的注意力机制，该配置方案与当前主流大语言模型参数设置保持兼容性。执行时间如表 7 和表 8 所示。

表 7 32 头执行时间对比

输入行数	DSP	V100
$S=128$	24.22	353.38
$S=256$	31.55	357.73
$S=384$	39.84	361.81
$S=512$	52.01	366.75
$S=640$	60.18	372.42
$S=768$	66.90	380.27
$S=896$	73.84	387.67
$S=1024$	83.08	394.78

表 8 64 头执行时间对比

输入行数	DSP	V100
$S=128$	59.32	1395.69
$S=256$	72.54	1412.04
$S=384$	89.78	1425.84
$S=512$	109.56	1448.23
$S=640$	131.75	1466.16
$S=768$	157.71	1496.64
$S=896$	172.28	1515.50
$S=1024$	194.27	1529.73

为了全面评估算法的性能，本文选择了多种序列长度（sequence length）进行测试，将序列长度 S 设置为 128 至 1024，每隔 128 取一个数据点，以模拟不同输入序列长度的场景。并将不同头数的加速比绘制成折线图，如图 11 所示，直观展示了 MHA 在不同配置下的性能表现。

根据实验数据，MT-3000 处理器在 MHA 机制上的性能表现显著优于 NVIDIA V100 GPU，尤其是在高头数（head=64）场景下展现了更强的加速潜力。当头数从 32 增至 64 时，MT-3000 的加速比大幅提升。以输入行数 $S=128$ 为例，head=32 时加速比为 14.59×，而 head=64 时加速比跃升至 23.53×。这一现象表明，V100 在高并行多头计算中面临显存带宽瓶颈，而 MT-3000 通过优化数据流调度（如 DMA 广播、算子融合）和高效利用片上缓存，显著降低了 I/O 访存开销。随着输入序列长度 S 从 128 增至 1024，MT-3000 的加速比呈现逐步下降趋势，但整体仍保持显著优势。在 head=64 时，$S=128$ 的加速比为 23.53×，而 $S=1024$ 时降至 7.87×。这一趋势是由于 V100 的数据搬运至显存时间占比减小导致的，尽管如此，MT-3000 通过分层存储架构和异步 DMA 传输，仍能在大规模输入下维持高效计算。例如，在 $S=1024$ 时，MT-3000 耗时 194.27 ms 完成计算，而 V100 需 1529.73 ms，表明 MT-3000 在长序列任务中具备实际应用价值。在典型的大语言模型配置中（如嵌入维度 8192 搭配 64 头），MT-3000 在小规模输入（$S=128\sim512$）时加速比高达 13.22×～23.53×，适合实时推理任务（如短文本生成、交互式对话）。对于长序列任务（$S=1024$），加速比仍保持在 7.87×，显示其在复杂场景下的适应性。

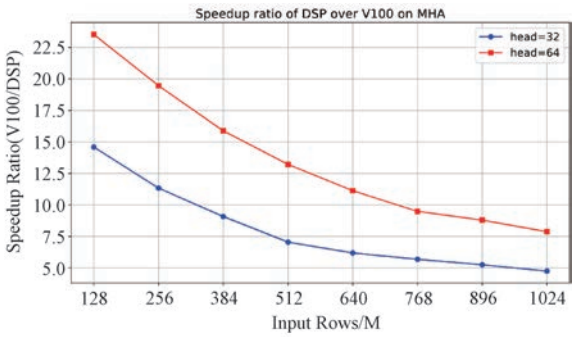

图 11 MHA 在不同规模下的加速比

5.4 多簇上的 MHA 扩展性测试

本节针对 MT-3000 验证了多簇并行计算的扩展效率。基于 32 头注意力机制配置，通过每簇独立处理 batch_size=1 任务进行测试。如表 9 所示，当输入序列长度从 $S=128$ 扩展至 $S=1024$ 时，四簇全激活模式相较单簇计算的时延增幅始终控制在 3.5%以内，其中 $S=1024$ 场景下四簇计算时延为 83.08 ms，较单簇 80.63 ms 仅增加 2.45 ms，展现出优异的扩展线性度。

这种准线性时延增长主要归因于多簇调度机制的固有开销。当 CPU 进行多簇任务分配时，每个新增的簇都需要独立的线程组初始化和同步操作，而这些管理操作的耗时与激活的簇数量呈正相关关系。尽管存在调度开销，系统整体吞吐量仍呈现出

近似线性扩展的趋势。以 $S=1024$ 为例,四簇模式单位时间内可完成 48.15 个样本计算(83.08 ms/样本),较单簇模式(80.63 ms/样本)实现 3.98 倍吞吐量提升,充分证明 MT-3000 的分布式架构能有效将计算资源转化为实际性能增益。

表 9　32 头 MHA 扩展性

输入行数	1 簇	2 簇	3 簇	4 簇
$S=128$	21.97	22.77	23.32	24.22
$S=256$	28.95	29.80	30.65	31.55
$S=384$	37.29	38.09	38.94	39.84
$S=512$	49.46	50.26	51.11	52.01
$S=640$	57.63	58.43	59.28	60.18
$S=768$	64.35	65.15	66.00	66.90
$S=896$	71.29	72.09	72.94	73.84
$S=1024$	80.63	81.43	82.28	83.08

结合实验数据分析,本文提出的优化方案在 MT-3000 异构多核处理器上展现出显著性能优势:针对核心算子 Linear 的计算效率较通用 CPU 核心实现最高 821.59 倍加速($S=896$,$D=4096$)。系统在多簇并行环境中实现近线性扩展,四簇模式下吞吐量较单簇提升 3.98 倍($S=1024$),时延增幅严格控制在 3.5%以内,体现分布式计算的负载均衡优势。优化后的 MHA 机制在典型大语言模型配置($D=8192$,64 头)下实现 13.22－23.53 倍加速比($S=128$－512),且在长序列场景($S=1024$)仍维持 7.87 倍性能优势,突破了传统 GPU 在多头并行计算中的带宽瓶颈。

6　总结与未来工作

本文针对 MT-3000 异构多核处理器提出了一种针对 Multi-Head Attention(MHA)机制的高性能并行实现方法,通过深入分析其架构特性,设计了涵盖算子优化、访存优化和调度优化的综合策略。算子优化方面,采用矩阵分块、内核级优化和算子融合技术,使 Linear 算子在单簇上的峰值性能达到 1.53 TFLOPS,占理论峰值的 37.7%,相较 NVIDIA V100 GPU 最高加速 5.34 倍;访存优化方面,利用广播机制和全局共享内存(GSM)优化数据流,降低主存带宽依赖,同时通过分阶段数据流调度和片上缓存管理减少 I/O 开销;调度优化方面,采用行粒度分块并行策略,隐藏数据传输延迟,确保高效计算执行。实验表明,优化后的 MHA 机制在典型大语言模型配置下(嵌入维度 4096/8192,头数 32/64)相较 NVIDIA V100 GPU 最高加速 23.53 倍,且在单节点多簇环境中展现出良好的扩展性,为长序列推理任务中的高效部署提供了技术支持。

未来工作将聚焦于在 MT-3000 处理器上构建高效的大模型推理优化框架。首先,开发针对大语言模型的量化推理引擎,通过混合精度计算降低存储需求并加速低精度矩阵运算。其次,针对处理器多簇架构特性优化模型并行与流水线并行机制,实现计算任务的动态负载均衡,提高吞吐量并降低长序列推理延迟。第三,探索稀疏计算加速,结合模型剪枝技术提升推理性能。通过系统化的协同优化方案,最终实现在 MT-3000 平台上对百亿至千亿参数大模型的高效推理支持,为复杂 AI 任务提供核心技术支持。

致　谢　本论文由国家重点研发计划资助,资助编号为 2023YFB3001900。

参　考　文　献

[1] LeCun Y, Bengio Y, Hinton G. Deep learning. nature, 2015, 521(7553): 436-444

[2] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need//Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17). Red Hook, USA, 2017: 6000-6010

[3] Wang Y, Li C, Liu C, et al. Advancing DSP into HPC, AI, and beyond: challenges, mechanisms, and future directions. CCF Transactions on High Performance Computing, 2021, 3: 114-125

[4] Igual F D, Ali M, Friedmann A, et al. Unleashing the high-performance and low-power of multi-core DSPs for general-purpose HPC//SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. Salt Lake City, USA, 2012: 1-11

[5] Yin S, Wang Q, Hao R, et al. Optimizing irregular-shaped matrix-matrix multiplication on multi-core DSPs//Proceedings of the 2022 IEEE International Conference on Cluster Computing (CLUSTER). Heidelberg, Germany, 2022: 451-461

[6] Yu K, Qi X, Zhang P, et al. Optimizing general matrix multiplications on modern multi-core DSPs//Proceedings of the 2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS). San Francisco, USA, 2024: 964-975

[7] Lu K, Wang Y, Guo Y, et al. MT-3000: A heterogeneous multi-zone processor for HPC. CCF Transactions on High Performance Computing, 2022, 4(2): 150-164

[8] Ma X, Fang G, Wang X. Llm-pruner: On the structural pruning of large language models. Advances in Neural Information Processing Systems, 2023, 36: 21702-21720

[9] Frantar E, Alistarh D. Sparsegpt: Massive language models

can be accurately pruned in one-shot//Proceedings of the International Conference on Machine Learning. Honolulu, USA, 2023: 10323-10337

[10] Wang S, Li B Z, Khabsa M, et al. Linformer: Self-attention with linear complexity. arXiv preprint arXiv:2006.04768, 2020

[11] Huang L, Yuan Y, Guo J, et al. Interlaced sparse self-attention for semantic segmentation. arXiv 2019. arXiv preprint arXiv:1907.12273

[12] Lin J, Tang J, Tang H, et al. AWQ: Activation-aware Weight Quantization for On-Device LLM Compression and Acceleration. Proceedings of Machine Learning and Systems, 2024, 6: 87-100

[13] Dettmers T, Lewis M, Belkada Y, et al. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. Advances in Neural Information Processing Systems, 2022, 35: 30318-30332

[14] Aminabadi R Y, Rajbhandari S, Awan A A, et al. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale//SC22: International Conference for High Performance Computing, Networking, Storage and Analysis. Dallas, USA, 2022: 1-15

[15] Dao T, Fu D, Ermon S, et al. Flashattention: Fast and memory-efficient exact attention with io-awareness. Advances in Neural Information Processing Systems, 2022, 35: 16344-16359

[16] Dao T. Flashattention-2: Faster attention with better parallelism and work partitioning. arXiv preprint arXiv:2307.08691, 2023

[17] Kao S C, Subramanian S, Agrawal G, et al. Flat: An optimized dataflow for mitigating attention bottlenecks//Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. Vancouver, Canada, 2023: 295-310

[18] Kitaev N, Kaiser Ł, Levskaya A. Reformer: The efficient transformer. arXiv preprint arXiv:2001.04451, 2020

[19] Beltagy I, Peters M E, Cohan A. Longformer: The long-document transformer. arXiv preprint arXiv:2004.05150, 2020

[20] Paszke A. Pytorch: An imperative style, high-performance deep learning library. arXiv preprint arXiv:1912.01703, 2019

[21] Sze V, Chen Y H, Yang T J, et al. Efficient processing of deep neural networks: A tutorial and survey. Proceedings of the IEEE, 2017, 105(12): 2295-2329

[22] Yuan Z, Shang Y, Zhou Y, et al. Llm inference unveiled: Survey and roofline model insights. arXiv preprint arXiv: 2402.16363, 2024

[23] Choquette J, Gandhi W, Giroux O, et al. Nvidia a100 tensor core gpu: Performance and innovation. IEEE Micro, 2021, 41(2): 29-35

[24] Khalilov M, Timoveev A. Performance analysis of CUDA, OpenACC and OpenMP programming models on TESLA V100 GPU. Journal of Physics: Conference Series. IOP Publishing, 2021, 1740(1): 012056

[25] Jia Z, Maggioni M, Staiger B, et al. Dissecting the NVIDIA volta GPU architecture via microbenchmarking. arXiv preprint arXiv:1804.06826, 2018

[26] Li M, Liu Y, Liu X, et al. The deep learning compiler: A comprehensive survey. IEEE Transactions on Parallel and Distributed Systems, 2020, 32(3): 708-727

[27] Goto K, Geijn R A. Anatomy of high-performance matrix multiplication. ACM Transactions on Mathematical Software (TOMS), 2008, 34(3): 1-25

[28] Van Zee F G, Van De Geijn R A. BLIS: A framework for rapidly instantiating BLAS functionality. ACM Transactions on Mathematical Software (TOMS), 2015, 41(3): 1-33

[29] Touvron H, Lavril T, Izacard G, et al. Llama: Open and efficient foundation language models. arXiv preprint arXiv: 2302.13971, 2023

[30] Bai J, Bai S, Chu Y, et al. Qwen technical report. arXiv preprint arXiv:2309.16609, 2023

LU Yao, Ph. D. candidate. His main research interests are parallel computing and high-performance computing.

LUAN Zhong-Zhi, Ph. D., associate professor. His main research interests include resource management in distributed environments, supporting methods and technologies for application development on heterogeneous computing systems, performance analysis and optimization of high-performance computing applications, etc., focusing on advancing computational efficiency and innovation across diverse research domains.

LI Gen, M. S. candidate. His main research interests are parallel computing and high-performance computing.

QI Jia-Xing, Ph. D. candidate. His main research interests are log analysis and high-performance computing.

HAN Bin, Ph. D. candidate. His main research interests are parallel computing, high-performance computing and task scheduling.

YANG Hai-Long, Ph. D., professor. His research interests include parallel and distributed computing, HPC, performance optimization and energy efficiency.

QIAN De-Pei, professor, member of Chinese Academy of Sciences. His research interests include distributed computing, high performance computing and computer architecture.

Background

This paper addresses a critical challenge at the intersection of high-performance computing (HPC) and artificial intelligence (AI): optimizing Multi-Head Attention (MHA) mechanisms on bandwidth-constrained heterogeneous processors. MHA is a core component of Transformer-based models, which have revolutionized natural language processing (NLP) and other AI domains. However, as model sizes grow to billions or even trillions of parameters, traditional hardware accelerators such as GPUs face significant challenges in balancing computational efficiency and memory bandwidth utilization. Current international research has primarily focused on optimizing MHA on GPU architectures, leveraging high-bandwidth memory (HBM) and Tensor Core acceleration. Despite these efforts, existing solutions often struggle with bandwidth-constrained scenarios, particularly on low-power processors like digital signal processors (DSPs).

In this work, we propose a comprehensive optimization framework for MHA on the MT-3000 processor, a low-power, high-performance DSP designed for HPC tasks. By integrating operator optimization, memory access optimization, and scheduling optimization, we significantly enhance MHA inference efficiency in bandwidth-limited environments. Our optimizations achieve a peak performance of 1.53 TFLOPS for the Linear operator on a single cluster, reaching 37.7% of the theoretical peak and achieving up to 23.53× acceleration over NVIDIA V100 GPUs in typical large-language-model configurations. These results demonstrate the potential of MT-3000 as an efficient alternative for AI workloads, particularly in long-sequence inference tasks.

This research is part of the National Key R&D Program of China project titled "Application Support Environment and Development Framework for Next-Generation Domestic Supercomputing Systems" (Grant No. 2023YFB3001900). The project aims to establish a diverse application support environment and development framework to facilitate the adaptation and upgrade of diverse autonomous application software to next-generation domestic supercomputing systems. This work contributes to advancing the development and efficient operation of framework software for scientific intelligence computing on next-generation domestic supercomputing systems.