doi:10.11887/j.issn.1001-2486.25090058 http://journal.nudt.edu.cn

面向AMX单元的矩阵算子优化方法

杨维铃,方建滨,董德尊*
(国防科技大学 计算机学院, 湖南 长沙 410073)

摘 要:在混合专家模型的推理过程中,矩阵算子构成了性能瓶颈,其中尤以注意力模块和专家计算所涉及的矩阵算子耗时最为显著。尽管现有方法已对GPU上的矩阵算子进行了深度优化,但鉴于GPU与CPU在内存架构及计算单元方面存在显著差异,这些优化方法难以直接迁移至CPU平台。为此,专门针对CPU的高级矩阵扩展单元,提出一种矩阵算子性能优化方案FlashMatrix。创新性地设计了高效的数据布局转换策略,有效规避了因数据布局转换而引发的额外内存访问开销;针对矩阵乘运算,精心构建了计算访存比最优的微内核,以实现寄存器的高效复用。实验结果表明,相较于当前CPU平台上最先进的矩阵计算库oneDNN,FlashMatrix平均实现了2.5倍的加速效果。对于端到端的推理性能,FlashMatrix实现了约1.2的加速比。

关键词:AMX单元;矩阵算子;数据布局;性能优化
中图分类号:TP301.6;TP393 **文献标志码**:A **文章编号**:1001-2486(2026)03-357-11

Matrix operator optimization method for AMX unit

*YANG Weiling, FANG Jianbin, DONG Dezun**
(College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China)

Abstract: In the inference process of mixture of experts models, matrix operators constitute the primary performance bottleneck, with those in the attention module and expert computation being particularly time-consuming. Although existing approaches have extensively optimized matrix operators on GPUs, the substantial differences between GPU and CPU architectures in memory hierarchy and compute units make these optimizations difficult to transfer directly to CPU platforms. To address this limitation, FlashMatrix was introduced as a matrix-operator optimization scheme tailored for CPU equipped with advanced matrix extensions. FlashMatrix incorporates an efficient data layout transformation strategy that avoids additional memory-access overhead caused by layout conversions, and employs a carefully designed micro-kernel for matrix multiplication that achieves an optimal compute-to-memory ratio through effective register reuse. Experimental results show that, compared with the state-of-the-art CPU matrix-computation library oneDNN, FlashMatrix delivers an average 2.5× speedup. For end-to-end inference performance, FlashMatrix achieves a speedup of approximately 1.2×.

Keywords: AMX unit; matrix operators; data layout; performance optimization

为应对快速发展的智能应用带来的计算挑战,主流处理器厂商正加速将新型计算单元整合至处理器核心架构中。以Intel的高级矩阵扩展(advanced matrix extensions, AMX)为代表的新一代矩阵计算单元[1],相较于传统向量计算单元(如Intel的AVX-512),实现了计算性能的跨越式提升,使单芯片峰值算力突破数百TFLOPS量级。这类先进计算单元呈现两大显著特征:其一,其高效运行高度依赖特定数据布局[2];其二,对内存访问带宽的需求较向量单元提升一个数量级。这种架构演进导致传统基于向量单元的优化方法失效,亟须探索与之适配的新型优化策略。

混合专家(mixture of experts, MoE)模型作为当前智能领域的研究热点,加速其推理已成为学术界与工业界关注的焦点。矩阵算子是MoE推理的核心计算组件,其性能直接决定了整体推理效率。具体而言,分组查询注意力(grouped query

收稿日期:2025-09-30
基金项目:国家自然科学基金委员会联合基金资助项目(U24B20151)
第一作者:杨维铃(1996—),男,湖北荆州人,博士研究生,E-mail:w.yang@nudt.edu.cn
***通信作者**:董德尊(1980—),男,黑龙江五常人,教授,博士,博士生导师,E-mail:dong@nudt.edu.cn
引用格式:杨维铃, 方建滨, 董德尊. 面向AMX单元的矩阵算子优化方法[J]. 国防科技大学学报, 2026, 48(3): 357-367.
Citation: YANG W L, FANG J B, DONG D Z. Matrix operator optimization method for AMX unit[J]. Journal of National University of Defense Technology, 2026, 48(3): 357-367.

attention,GQA)机制与专家前馈神经网络(feed forward neural network, FFN)构成了 MoE 模型推理中最关键的性能瓶颈。现有研究工作正围绕这两个算子展开深度优化[3-7]。

选择针对配备 AMX 单元的 CPU 进行矩阵算子优化,主要基于以下三方面考虑。首先,与 GPU 相比,CPU 为 MoE 推理提供了一种更具性价比的方案。CPU 具有更大的内存容量,可在单个节点上完成 MoE 模型的部署,而无须依赖多个 GPU。同时,CPU 的算力与内存带宽正持续提升,能够更好地满足 MoE 推理的性能需求。其次,CPU 的内存带宽显著高于 CPU-GPU 之间的数据传输带宽。在 GPU 资源有限的推理场景中,直接在 CPU 上执行大部分矩阵计算通常能获得更好的性能。例如,LIA 框架表明[5],在仅配备单个 GPU 的节点上运行大语言模型推理时,将主要计算任务放在配备 AMX 单元的 CPU 上执行能够显著提升性能,尤其是在输入批量规模仅为数百时。最后,与仅配备向量单元的 CPU 相比,AMX 能大幅提升矩阵算子性能。此外,尽管针对向量单元的矩阵优化技术已相当成熟,但针对 AMX 的优化研究仍有待深入探索。

AMX 单元带来的算力提升,以及 MoE 技术的快速发展,给现有矩阵算子的性能优化方法提出了新的挑战[8-14],其主要体现在两个方面:其一,矩阵数据布局的通用性问题[10-12];其二,难以高效支持 MoE 中的复杂矩阵算子[9,13-14]。以 Intel 的 oneDNN[8,10]库为例,该库通过即时编译技术动态生成微内核代码,在单个计算密集型算子,如矩阵乘(matrix multiplication,MM)上能够取得优异的性能。然而,对于诸如 GQA 和专家 FFN 等复杂矩阵算子,其性能显著下降,实测结果显示,GQA 仅能达到约 15 TFLOPS,专家 FFN 甚至不足 2 TFLOPS,远低于 CPU 理论峰值 157 TFLOPS。性能瓶颈主要来自两个方面:一是寄存器与 cache 中的数据复用不足,导致频繁且冗余的主存访问;二是为适配 AMX 对数据布局的要求,额外引入了布局转换开销。尽管 LIBSHALOM[13]和 NDIRECT[14]等库对通用布局下的矩阵算子进行了高度优化,但它们主要面向向量计算单元,且仅针对传统的单算子场景,难以高效支持 MoE 中的矩阵计算。

为了解决这些问题,提出了 FlashMatrix,它是专为加速 MoE 中的矩阵算子而设计,核心包含两项协同优化策略。其一,突破传统 oneDNN 库对权重矩阵进行布局转换的方法,转而只对 MoE 中规模显著更小的输入矩阵实施布局转换,并将转换操作放在 cache 和寄存器中执行。此策略大幅降低了布局转换开销,同时确保权重矩阵始终维持行主序/列主序等通用布局。其二,还设计了计算访存比最大的微内核,以充分复用 AMX 的寄存器。

实验在 Intel Gold 6430 和 Intel Platinum 8468V 上开展,测试涵盖三个核心算子:单个 MM、GQA 及专家 FFN。实验结果显示,FlashMatrix 在所有测试场景中均超过当前 CPU 平台最先进的 oneDNN 库的性能。尤其是对于专家 FFN 的性能,FlashMatrix 实现了平均 2.5 的加速比优势。这一成果为配备 AMX 单元的 CPU 平台上 MoE 的矩阵算子的优化提供了新思路。

1 背景介绍

1.1 MoE 的架构

MoE 模型由多个 Transformer 层堆叠构成[6]。每层主要包含注意力(Attention)模块和 FFN 模块,其中每层 FFN 模块由多个专家组成,每个专家又可视为一个独立的 FFN。如图 1 所示,每层输入的词元(token)在经过 Attention 模块处理后,会通过路由机制选择若干特定的专家进行后续处理。每个 token 所选专家的具体数量由模型架构决定。例如,Mixtral-8x22B 模型在每层的 8 个专家中选择 2 个进行计算[15]。MoE 与稠密模型的核心区别在于其引入了稀疏激活的专家机制。在稠密架构中,每层仅包含一个 FFN 模块,所有 token 都需要经过该模块进行完整计算;而在 MoE 架构中,每层构建了多个专家 FFN 子模块,通过

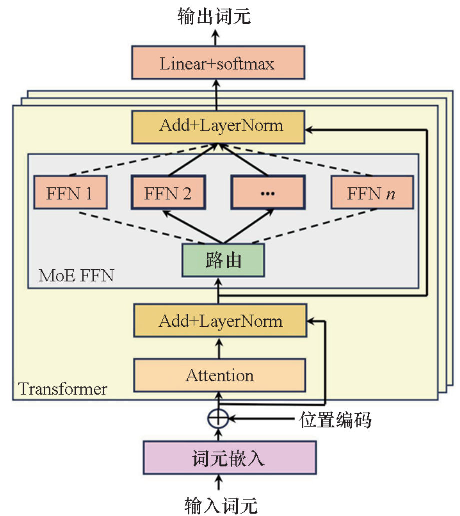


图 1 MoE 的整体架构

Fig. 1 Overall architecture of MoE

门控机制实现动态路由,仅激活少数专家参与当前 token 的计算,从而显著降低了实际计算量与内存访问开销。

1.2　MoE 推理的性能瓶颈

在单 CPU 与单 GPU 构成的计算平台上加速 MoE 推理,是当前学术界关注的热门课题[3-7]。在该硬件配置下,专家权重通常存储在 CPU 主存,推理过程中根据计算需求通过 PCIe 动态传输至 GPU 显存。现有研究表明,当输入的提示词长度较短且批量规模较小时,直接在 CPU 端执行专家计算反而能获得更优性能。这一现象主要源于 PCIe 总线的传输带宽(通常为 16 ~ 32 GB/s)显著低于 CPU 的主存带宽(200 GB/s 以上),此时 CPU-GPU 间的数据传输开销成为主要瓶颈,CPU 本地计算可有效避免数据传输。

图 2 展示了 MoE 模型 Mixtral-8x7B[16] 推理过程中各操作的运行时间占比。在测试中,批量大小被设置为 1,这符合本地大模型部署的实际推理场景需求。从图中可以看出,对于全部计算任务卸载至 GPU 的推理方法,PCIe 数据传输成为最显著的性能瓶颈,其占总体运行时间的比例高达 93.0%。而对于全部计算任务在 CPU 上执行的推理方法,FFN 模块和 Attention 模块的计算开销占比最大。相比于 GPU 方法,在小批量推理情况下,CPU 方法能够避免高开销的 PCIe 数据传输,从而提升性能。

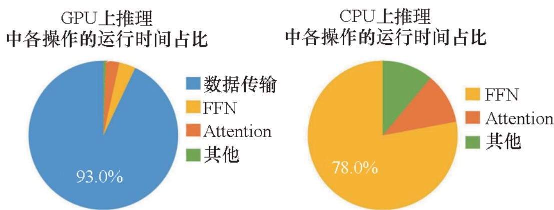


图 2　MoE 推理中各个操作的运行时间占比

Fig. 2　Runtime proportion of each operation in MoE inference

1.3　AMX 单元特征

AMX 单元是 Intel 专为加速智能应用中的矩阵计算而设计,支持 BF16/INT8 精度的输入数据,并分别输出 FP32/INT32 精度数据。相较于传统的 AVX-512,AMX 在 BF16 与 INT8 数据精度下分别可实现 16 倍和 8 倍的性能提升。每个 AMX 核心配备 8 个 tile 寄存器(tmm0 ~ tmm7),每个寄存器最大可配置为 1 024 B(16 行 ×64 B/行)。图 3 展示了 AMX 针对 BF16 数据类型的融合乘加(fused multiply-add, FMA)指令执行流程:单条指令可在 16 个时钟周期内完成 16 ×16 ×32 次计算(示意图采用 4 ×8 大小的矩阵来简化表示)。

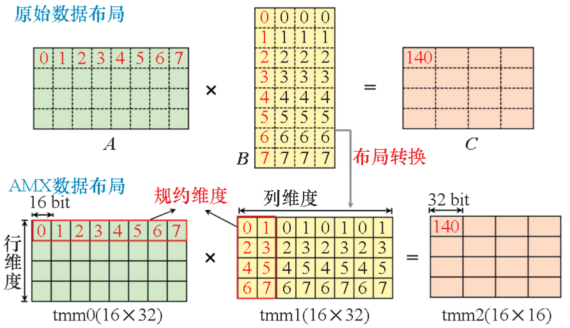


图 3　AMX 单元的 FMA 指令计算流程

Fig. 3　FMA instruction calculation flow in the AMX

AMX 的 tile 寄存器(tmm0 ~ tmm2)的数据布局要求如下:tmm0 的布局为 $[16][*][\underline{32}]$,tmm1 为 $[\underline{16}][*][16][\underline{2}]$,tmm2 为 $[16][*][16]$。其中下划线表示规约维度,* 表示跨行的步长。在图 3 中,tmm0 的数据布局与矩阵 **A** 的数据布局相匹配,而 tmm1 的数据布局与矩阵 **B** 的原始布局不同。tmm0 的布局称为通用布局,tmm1 的布局称为专用布局。由于 AMX 的 FMA 指令对输入矩阵的布局有明确要求,即必须一个为专用布局,另一个为通用布局,因此在启动计算之前,需对矩阵(如图 3 中的矩阵 **B**)进行必要的布局转换,以满足指令要求。

本文要求所有算子的输入/输出矩阵数据布局必须采用通用布局(行主序或列主序)。这种约束的核心原因在于:注意力模块与专家 FFN 模块在 CPU 与 GPU 上哪个执行效率更优,本质上由输入 token 的批量大小及序列长度动态决定。若为适配 AMX 单元特性而预先将矩阵转换为对 AMX 友好的专用布局,当这些模块需要迁移至 GPU 执行时,则必须重新将数据布局还原。这一转换过程会产生显著内存访问开销。对 AMX 适配的数据布局与 GPU 的 CUDA 核心、Tensor 核心并不适配。因此,采用通用数据布局可避免跨设备执行时的二次转换开销,确保 MoE 模型在 CPU-AMX 与 GPU 间的灵活迁移能力。

1.4　研究的问题

高性能的 MoE 推理系统由上层的卸载策略和底层的算子优化构成。本文聚焦于在 AMX 的 CPU 平台上优化 MoE 中的矩阵算子,不关注卸载策略。具体算子包括专家 FFN 模块和 Attention 模块,具体算子的计算流程如图 4 所

示。需要说明的是,MoE 中完整的 GQA 由 4 个 MM 和缩放点积注意力(scaled dot-product attention, SDPA)构成。由于针对 MM 算子的优化研究已较为充分[13,17],本文主要聚焦于多个 MM 组合的性能优化,且所提出的方法同样适用于单个 MM。

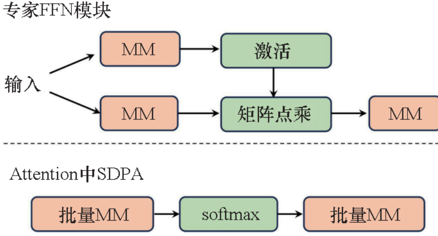


图4 专家 FFN 和 Attention 模块的计算流程

Fig. 4 Computational process of the expert FFN and the Attention module

2 优化方法

优化方法主要包括微内核设计和数据布局转换优化。由于 GQA 与专家 FFN 均以 MM 为基础构建,下面将先在 2.1 节阐述 MM 微内核设计方法,为后续优化奠定基础。随后,2.2 节和 2.3 节将分别讨论对专家 FFN 和 GQA 的优化。

为方便描述,文中的例子都使用 BF16 精度的数据,这是目前大语言模型推理的主流精度。矩阵的命名规则如下:权重矩阵记为 $\boldsymbol{B}$,输入矩阵记为 $\boldsymbol{A}$;矩阵的行维度用 M 表示,列维度用 N 表示,规约维度用 K 表示。

2.1 MM 的微内核设计

本文只讨论 MM 的微内核设计,分块和循环顺序直接参考 Goto 算法[18]。对微内核设计而言,要尽量提高计算访存比(compute-to-memory ratio, CMR),以充分复用寄存器中的数据,减少访存开销。所提出的微内核优化方法针对 AMX 单元,而现有研究多面向传统的向量计算单元(如 AVX-512、NEON 等)[8,13]。与向量寄存器相比,AMX 的 tile 寄存器具有二维结构且数量更少,其使用模式与约束条件均存在差异。此外,AMX 计算完成后还需将结果转移至向量寄存器以执行精度转换操作。

2.1.1 微内核大小

微内核的行维度大小为 m_r,列维度大小为 n_r,规约维度大小为 k_r。AMX 的 tile 寄存器的规约维度最大可配置为 32,为了最大化计算性能,直接设置 $k_r = 32$。在微内核的每次迭代中,处理器需要从矩阵 $\boldsymbol{A}$ 加载 $64 \times m_r$ B 的数到寄存器,从矩阵 $\boldsymbol{B}$ 加载 $64 \times n_r$ B 的数。微内核的计算访存比 C_r 表示如下:

$$C_r = \frac{2 \times m_r \times n_r \times 32}{64 \times m_r + 64 \times n_r} \quad (1)$$

FMA 指令包含了乘和加操作,因此计算过程中涉及乘 2 的系数调整。AMX 的 FMA 指令要求输入数据为 BF16 精度(2 B),输出数据为 FP32 精度(4 B)。因此,输入数据的大小分别为 $64 \times m_r$(行维度)和 $64 \times n_r$(列维度),而输出的数据为 $4 \times m_r \times n_r$。每个 AMX 核心有 8 个 1 024 B 的 tile 寄存器。微内核的大小必须满足如下约束:

$$64 \times (m_r + n_r) + 4 \times m_r \times n_r \leqslant 1\,024 \times 8 \quad (2)$$

利用拉格朗日乘子法,可推导出当 $m_r = n_r = 32$ 时,C_r 取得最大值。AMX 的 tile 寄存器的最大行/列维度可配置为 16。因此,在微内核中,矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 至少分别需要 2 个 tile 寄存器,$\boldsymbol{C}$ 需要 4 个 tile 寄存器。

2.1.2 微内核的实现细节

为充分挖掘 AMX 单元算力,使用 x86 汇编来实现微内核,具体实现如算法 1 所示。在数据加载方面,针对矩阵 $\boldsymbol{A}$ 使用 tileloaddt1 指令,该指令融合了数据预取与加载操作。在微内核计算过程中,矩阵 $\boldsymbol{A}$ 的数据源于 L2 cache,通过预取指令将其提前加载至 L1 cache,有效减小从 L2 cache 取数带来的延迟。由于预取指令与计算指令交织执行,不会产生额外开销。对于矩阵 $\boldsymbol{B}$,则采用 tileloadd 指令直接加载数据,无须额外预取,因为矩阵 $\boldsymbol{B}$ 的数据在微内核计算过程中来自 L1 cache。

算法1 微内核的实现细节

Alg. 1 Implementation details of a micro-kernel

输入:矩阵 $\boldsymbol{A}$、矩阵 $\boldsymbol{B}$
输出:矩阵 $\boldsymbol{C}$

```
for k = 0 to K, step 32, do
    (tmm0 ~ tmm1) = A[ * ][k:k + 32]  // tileloaddt1
    (tmm2 ~ tmm3) = B[ * ][16][16][2] // tileloadd
    // tdpbf16ps
    (tmm4 ~ tmm7) = FMA (tmm0 ~ tmm1) (tmm2 ~ tmm3)
end for
// 存储矩阵 C
栈空间 = (tmm4 ~ tmm7)  // 起始地址与缓存行对齐
(zmm0 ~ zmm31) = 空间
(zmm0 ~ zmm15) = (zmm0 ~ zmm31) // vcvtne2ps2bf16
C = (zmm0 ~ zmm15)
```

在矩阵 ***C*** 的存储过程中，要对 tile 寄存器中的数据进行精度转换。由于在 MoE 的矩阵计算中，输入与输出数据均采用 BF16 精度，而 FMA 指令的输出结果为 FP32 精度，因此必须对其进行 FP32 到 BF16 的转换。具体实现流程如下：首先将 tile 寄存器（tmm4 ~ tmm7）中的计算结果存储至临时缓冲区，随后从该缓冲区加载数据至向量寄存器（zmm），在向量寄存器中完成 FP32 到 BF16 的转换操作。需要注意的是，AMX 的 tile 寄存器与向量寄存器之间无法直接通信，必须通过临时缓冲区进行数据中转。该缓冲区无须显式申请，直接在程序栈空间分配 1 024 B 容量即可。为最大化存储带宽利用率，缓冲区的起始地址需与缓存行大小对齐，确保数据访问符合硬件优化的内存对齐要求。处理器对该缓冲区的访问在 L1 cache 中进行，L1 cache 的访存带宽很高，因此数据精度转换带来的开销可忽略。

2.2　专家 FFN 模块性能优化

所提出的 FFN 模块性能优化方法在保持权重矩阵为通用布局（即行/列优先）的前提下，依然能充分发挥 AMX 的计算性能。与此相比，oneDNN 需要在运行时对权重矩阵进行布局转换，从而引入额外的开销[8]；而 IPEX[19] 则通过在部署阶段提前进行布局转换以规避运行时的布局转换开销，但这种做法牺牲了布局的通用性，使其难以在 CPU 和 GPU 之间灵活地进行任务卸载。

2.2.1　数据布局设计

在现有的 MoE 推理优化工作中，以 KTransformers[7] 框架为例，其选择将专家权重矩阵预先转换为适配 AMX 的专用数据布局。这种优化策略在提升 CPU 端计算性能方面确实成效显著，然而，却在一定程度上牺牲了专家 FFN 卸载的灵活性。已有研究明确证实，在 GPU 资源有限的计算节点中，例如仅配备一个 CPU 与一个 GPU 的计算环境，专家计算究竟在哪个设备上执行能够获得更优性能，是随输入的负载特征动态变化的[4-5]。具体而言，输入提示词的长度、批量大小以及推理阶段（Decode 阶段或 Prefill 阶段）等因素，均会对专家 FFN 的卸载设备选择产生影响。若经过适配 AMX 数据布局优化的专家 FFN 需要卸载至 GPU 进行计算，那么就必须将专用布局还原为通用布局，以适配 GPU 的计算特性。这是因为，对于 GPU 而言，当数据在规约维度上采用连续存储方式时，其计算性能能够达到最佳状态，即要求矩阵 ***B*** 采用列优先的存储布局[5]。

为适配图 3 中 AMX 单元的布局要求，需要对专家 FFN 模块的矩阵 ***A*** 进行数据布局转换。同时，将权重矩阵 ***B*** 按照规约维度连续的方式存储，即列优先布局。这种布局方式具有广泛的通用性，能够同时满足 CPU 的 AMX 和 GPU 的 Tensor 核心的需求。上述布局设计主要基于两点原因。其一，采用通用布局来存储权重矩阵，可确保其在 GPU 和 CPU 上均具备良好的通用性。这使得专家 FFN 模块无须额外承担布局转换开销，可依据实际负载特性，灵活地将计算任务卸载至性能更优的设备上运行。其二，针对输入矩阵的布局转换操作，可与 MoE 中的专家选择操作合并为一个操作。通过这种方式，能够直接消除布局转换所带来的额外开销。布局设计方法源于对计算负载以及体系结构特性的观察，在保证硬件适配性的前提下，实现了计算效率与卸载灵活性的双重提升。

2.2.2　计算流程

专家 FFN 模块的计算流程如图 5 所示，可细分为四个步骤。第一步是对输入矩阵 ***A*** 做布局转换，这个转换步骤与 MoE 的专家选择合并，不会带来额外的数据移动开销。第二步执行两个 MM 运算，二者共享同一输入矩阵，计算结果分别存入可被 L3 cache 完全容纳的临时缓冲矩阵 $\boldsymbol{B}_{u1}$

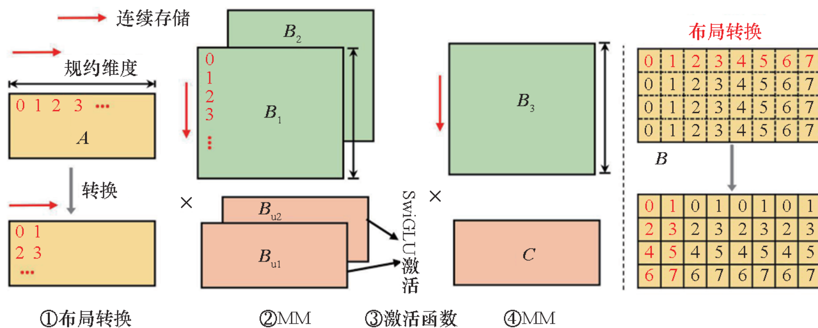


图 5　专家 FFN 模块的计算流程

Fig. 5　Computational process of the expert FFN module

和 $\boldsymbol{B}_{u2}$。其中，$\boldsymbol{B}_{u1}$ 和 $\boldsymbol{B}_{u2}$ 中的数据按照图3中的专用布局进行存储，以适配后续 MM 运算的需求。这个布局转换操作与向 $\boldsymbol{B}_{u1}$ 和 $\boldsymbol{B}_{u2}$ 写入数据的操作合并，直接在寄存器中完成，从而确保了整个流程无额外开销。第三步，对 $\boldsymbol{B}_{u1}$ 和 $\boldsymbol{B}_{u2}$ 中的结果进行 SwiGLU 激活计算，计算得到的结果写入 $\boldsymbol{B}_{u1}$。SwiGLU 是目前大语言模型中使用的主流激活函数[8-9]，其计算公式如下：

$$\mathrm{SwiGLU}(x_1, x_2) = \frac{x_1 \times x_2}{1 + \mathrm{e}^{-x_1}} \tag{3}$$

其中，x_1 与 x_2 分别代表 $\boldsymbol{B}_{u1}$ 和 $\boldsymbol{B}_{u2}$ 中处于相同位置的数据。在计算指数函数的值时，借鉴 XNNPACK[20] 的实现方法，将计算过程拆分为整数部分与小数部分分别处理。对于整数部分的计算，直接运用 x86 的硬件加速指令以提升计算效率；而对于小数部分，则采用泰勒展开至6次项进行近似计算。最后，将小数部分与整数部分的计算结果相乘，即可得到指数函数的最终计算结果。需要特别留意的是，在式(3)中，当 $x_1 > 128$ 时，直接将 e^{-x_1} 置为0；当 $x_1 < -128$ 时，直接将 SwiGLU 的结果置为0。上述两种情况，无须进行 exp 函数值的计算，否则可能引发浮点溢出中断，进而对程序的性能产生严重的负面影响。完成上述步骤后，进入第四步，将 $\boldsymbol{B}_{u1}$ 与权重矩阵 $\boldsymbol{B}_3$ 进行 MM 运算，从而得到最终结果矩阵 $\boldsymbol{C}$。

2.3 Attention 模块性能优化

所提出的 Attention 优化方法将布局转换与 MM 融合，使得布局转换在寄存器与 cache 内完成，从而显著降低了转换带来的额外开销。相比之下，FlashAttention 专为 GPU 平台设计，由于 GPU 对数据布局没有特殊要求，因此无须针对布局转换进行优化[17,21]。而在 oneDNN 中，第一个批量 MM 的中间结果被写回主存，随后第二个批量 MM 在执行时再从主存中读取这些数据，导致了大量不必要的内存访问开销。表1给出了 Attention 模块中用到的变量名及其含义。

表1 Attention 模块中用到的变量名及其含义

Tab.1 Variable names and their meanings used in the Attention module

变量名	变量名含义
s_1	序列长度
h_q	$\boldsymbol{Q}$ 矩阵的头数量
h_{kv}	$\boldsymbol{K}$、$\boldsymbol{V}$ 矩阵的头数量
d	每个注意力头的大小

2.3.1 计算流程

GQA 中的 Attention 的计算流程由输入投影(3个 MM)、SDPA 和输出投影组成(1个 MM)。MM 已经被现有工作高度优化，不是 Attention 模块的性能瓶颈，本工作聚焦于对 SDPA 的优化，其表示式如下：

$$\mathrm{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \mathrm{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d}}\right)\boldsymbol{V} \tag{4}$$

SDPA 主要由两个批量 MM 和位于它们之间的 softmax 操作构成。其中，批量 MM 不是 SDPA 的性能瓶颈所在。这是由于在 GQA 机制下，矩阵 $\boldsymbol{K}$ 和 $\boldsymbol{V}$ 会被矩阵 $\boldsymbol{Q}$ 复用 h_q/h_{kv} 次，在此过程中主存的数据访问量很小，这一特性有助于发挥 AMX 的计算性能。SDPA 的核心性能瓶颈在于 softmax 操作需对中间结果进行频繁访问。

softmax 操作需要对第一个批量 MM 所产生的中间结果进行三次访存操作，其具体计算流程如下：

$$\mathrm{softmax}(x_{i,j}) = \frac{\mathrm{e}^{x_{i,j} - \max_i}}{\sum_{j=1}^{s_l} \mathrm{e}^{x_{i,j} - \max_i}} \tag{5}$$

第一次访存是为了获取该中间结果矩阵每行的最大值；第二次访存是计算每个数据的指数，以及每行的累加和；第三次访存则是针对每个元素进行除法运算，以实现归一化。SDPA 中所有矩阵都采用通用布局存储，而 AMX 单元对矩阵布局有特定的要求。由于二者布局方式不一致，在实际计算过程中，需进行额外的布局转换操作。

借鉴 FlashAttention[10-11] 中的分块 softmax 方法，以消除 softmax 带来的访存开销，并在 L2 cache 中进行访存操作。图6展示了 GQA 中的 SDPA 的计算流程。分块的大小 t_s 由 L2 cache 的容量来决定。为了提升不规则 MM 的性能，第一个批量 MM 产生的结果被存储在一个可以被 L2 cache 容纳的缓冲区中，所有针对中间结果的读写都在这个缓冲区中进行。当 softmax 操作完成后，第二个 MM 直接从缓冲区中读取数据进行计算。每个中间结果矩阵的理论规模为 $s_1 \times s_1$，由于采用了分块计算，中间结果矩阵只需反复利用大小为 $32 \times t_s$ 的缓冲区即可完成 SDPA 的计算。2.1.1节中微内核的大小为32。现代处理器多采用最近、最少使用的 cache 替换策略，在一个缓冲区中进行反复读写还能提高 cache 命中率。

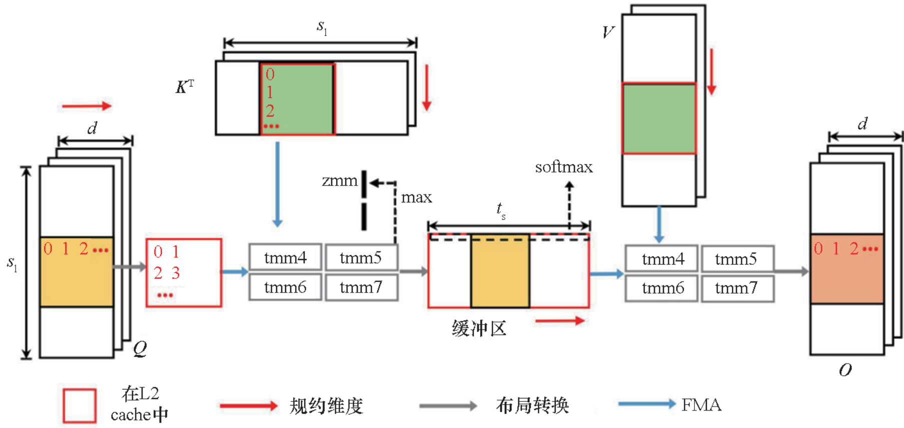


图 6　GQA 中的 SDPA 的计算流程

Fig. 6　Computational process of SDPA in GQA

为了适配图 3 中 AMX 的数据布局要求,引入了三个布局转换操作,这是与现有工作(如 FlashAttention)最大的区别。这些操作都在 L2 cache 或向量寄存器中进行,不会带来显著开销。第一步,将矩阵 $\boldsymbol{Q}$ 的通用布局转换为特定布局。这个转换是分块进行,只需要对主存进行一次访问,转换后的数据就被存放在 L2 cache 中。第二步,针对计算过程中得到的中间结果,同样要将其转换为特定布局。此操作在向量寄存器内执行,借助 x86 的数据交叉指令即可实现。这个转换操作是为了便于第二个批量 MM 利用 AMX 单元。第三步,对于结果矩阵 $\boldsymbol{O}$,本文会在寄存器内对其进行转置操作,以此保证该矩阵在内存中按照行优先布局存储。

2.3.2　实现细节

为了在寄存器中完成数据布局操作,需将转换操作与 MM 的存储操作融合,具体流程详见算法 2。在完成 $\boldsymbol{QK}^{\mathrm{T}}$ 计算后,所得结果会按照通用布局存储于 tmm4 ~ tmm7 寄存器中。由于 AMX 的 tile 寄存器不具备执行转换操作的能力,因此,需先将 tile 寄存器中的数据经由栈空间转移至向量寄存器,再由向量寄存器执行相应的转换操作。与直接将 tmm4 ~ tmm7 中的数据存回缓冲区的做法相比,将转换操作与存储操作融合的方式仅增加了两条指令,即 vpermt2d 和 vpshufb,这几乎不会带来额外开销。转换后的数据布局情况如图 3 中 tmm1 寄存器所示。反之,若先将计算结果全部存回缓冲区,之后再执行布局转换操作,就需要对 L2 cache 进行额外的读和写操作。而且,中间结果的数据规模庞大,这种做法会导致转换操作产生较大的开销。

算法 2　在向量寄存器中执行布局转换

Alg. 2　Perform layout transformation in vector registers

```
输入:矩阵 Q,矩阵 K
输出:矩阵 b_u

计算 QK^T      //计算结果存放在 tmm4 ~ tmm7 寄存器
//存储矩阵 b_u
for i = 0 to 4, step 1, do
    stack = tmm[4 + i]      // tilestored
    for j = 0 to 16, step 1, do
        (ymm0, ymm1) = stack [32 × j]
        // vpermt2d + vpshufb 指令
        zmm0 = 将 ymm0 与 ymm1 中的数据交叉
        b_u[512 × i + 32 × j] = zmm0
    end for
end for
```

3　实验结果

3.1　实验设置

实验在 Intel Gold 6430 (6430) 和 Intel Platinum 8468V(8468V)处理器上开展,具体平台的参数配置如表 2 所示。

表 2　实验用的处理器参数

Tab. 2　Processor parameters for the experiment

CPU 型号	核数	BF16 算力/TFLOPS	主存带宽/(GB/s)
6430	64	157.3	614.4
8468V	96	230.4	614.4

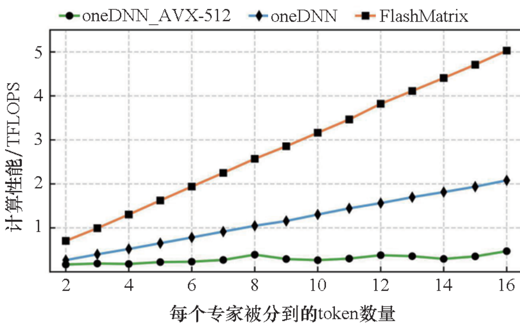
实验分别评测了算子性能及 MoE 模型的端到端推理性能。算子性能评测涵盖三类矩阵算子:FFN 模块、Attention 模块及单个矩阵乘,其性能结

果为运行 100 次取平均值。这些算子的矩阵规模均取自两个真实的 MoE 模型，即 Mixtral-8x22B[15] 和 Qwen3-235B-A22B[22]。在端到端推理性能评测中，FlashMatrix 被集成到 IPEX，并将其性能与 IPEX 原生实现、PyTorch 及 oneDNN 进行对比，性能结果为运行 5 次取平均值。所有实验均采用 BF16 数据精度，这是当前业界最常用的精度。IPEX 是 Intel 专门为加速深度学习模型的推理与训练而设计的优化框架；oneDNN 则是 Intel 为提升深度学习算子性能而开发的高性能计算库，为上层框架提供底层算子加速支持。

3.2 FFN 模块的性能

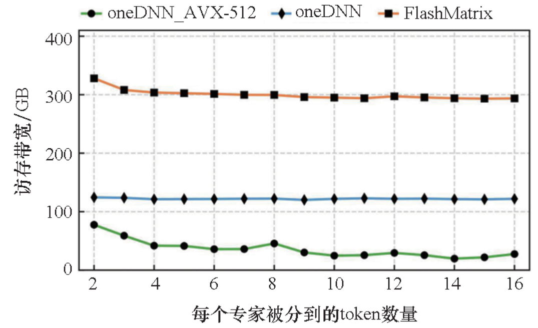
本实验评测了来自 Mixtral-8x22B 中的专家 FFN 模块。该模块包含三个权重矩阵，其中两个矩阵规模为 6 144 × 16 384，另一个为 16 384 × 6 144。实验对比了 FlashMatrix 与分别采用 AVX-512 单元的 oneDNN(oneDNN_AVX-512)和 AMX 单元的 oneDNN(oneDNN)之间的性能差异。

在不同 token 数量下专家 FFN 的性能如图 7 所示，FlashMatrix 相较于 oneDNN，在计算性能和访存性能方面，平均加速比均为 2.5。对于专家 FFN 而言，权重矩阵 $\boldsymbol{B}$ 的访问是最大的性能瓶颈。oneDNN 为了匹配 AMX 单元的数据布局要求，对权重矩阵进行布局转换，这会严重降低性能。FlashMatrix 消除了权重矩阵的布局转换操作，从而实现了显著的性能提升。从图 7 中还能发现一个现象：FFN 模块的计算性能几乎随着输入 token 数量的增加而呈线性增长。这主要是因为 FFN 模块属于访存受限型任务，其权重矩阵的规模远大于输入矩阵。而输入 token 数量的增加主要影响输入矩阵的规模，对总访存量的影响几乎可以忽略不计。这也是图 7 中各个方法的访存性能几乎不随 token 数量波动的原因。



(a) 不同 token 数量下专家 FFN 的计算性能

(a) Computational performance of expert FFN under different token counts



(b) 不同 token 数量下专家 FFN 的访存性能

(b) Memory performance of expert FFN under different token counts

图 7 在不同 token 数量下专家 FFN 的性能

Fig. 7 Performance of expert FFN under different token counts

图 8 展示了专家 FFN 运行期间的内存占用情况。从图中可以看出，在每个专家 FFN 的计算流程中，FlashMatrix 仅占用约 600 MB 的内存空间，这一内存占用值远低于 oneDNN 的 1 000 MB。这是由于 FlashMatrix 在运算过程中没有数据布局转换操作，无须额外开辟内存空间来存储转换后的权重矩阵。而且，由于权重矩阵在内存占用中占据主导地位，所以图 8 所显示的内存占用量几乎不会随着 token 数量的变化而出现明显波动。

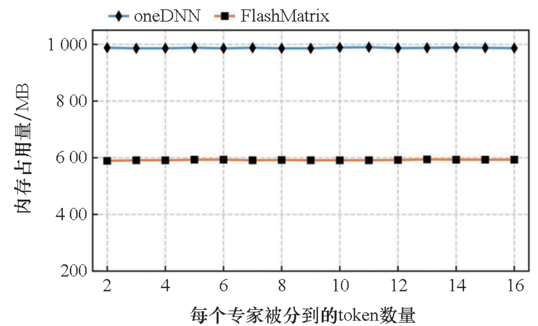


图 8 在不同 token 数量下专家 FFN 的内存占用量

Fig. 8 Memory usage of expert FFNs under different token counts

图 9 展示了在激活不同专家数量的情况下 FFN 的性能，其中每个专家被分到的 token 数量设置为 16。从图中可以看出，FlashMatrix 相较于 oneDNN，平均加速比依旧保持在 2.5。但随着专家数量的增加，FlashMatrix 和 oneDNN 的性能出现轻微下降。经分析，这一现象是由于数据规模扩大后，跨 NUMA 架构进行数据访问所引发的。后续的工作将通过设计新的并行化策略，来解决这一问题。

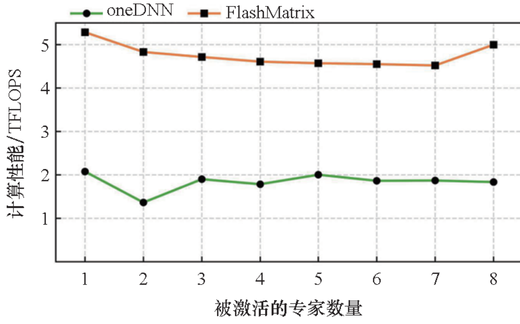
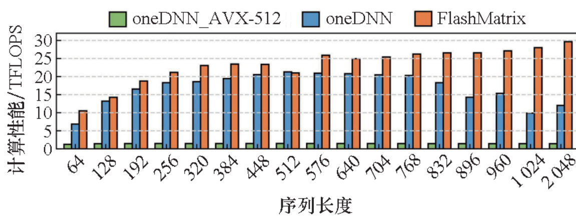


图 9 在激活不同专家数量下 FFN 的性能

Fig. 9 Performance of FFN under different numbers of activated experts

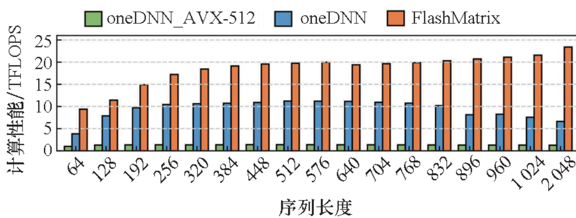
3.3 Attention 模块的性能

图 10 展示了在不同序列长度下,Attention 模块的性能变化情况。在此实验中,批量大小设定为 32,该批量大小是该领域较为常见的设置[9]。随着序列长度增加,FlashMatrix 相对于 oneDNN 的性能优势更加显著。这是因为序列长度增加时,对中间结果的访问所引发的性能瓶颈问题越严重,FlashMatrix 通过将中间结果保存在 L2 cache 中,可有效减少访存开销。此外,FlashMatrix 在 Mixtral-8x22B 模型中的 Attention 性能表现优于在 Qwen3-235B-A22B 模型中的表现。这是由于 Mixtral-8x22B 模型中每个注意力头的大小为 128,大于 Qwen3-235B-A22B 模型中的 64。较大的注意力头尺寸能够更充分地发挥 AMX 的计算性能,从而使得 FlashMatrix 在该模型中展现出更优的 Attention 性能。



(a) Mixtral-8x22B 中的 Attention 模块

(a) Attention module in Mixtral-8x22B



(b) Qwen3-235B-A22B 中的 Attention 模块

(b) Attention module in Qwen3-235B-A22B

图 10 不同序列长度下 Attention 模块在 6430 平台上的性能

Fig. 10 Performance of the Attention module on 6430 platform under different sequence lengths

图 11 展示了 Attention 模块在 8468V 上的性能。从图中可以看出,相较于在 6430 平台上的情况,FlashMatrix 在 8468V 平台上相对 oneDNN 的性能优势更为显著。这是因为 8468V 的计算访存比高于 6430,导致内存访问带来的性能瓶颈更严重,而 FlashMatrix 可有效减少内存访问量。

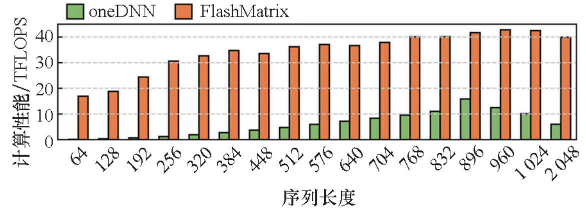
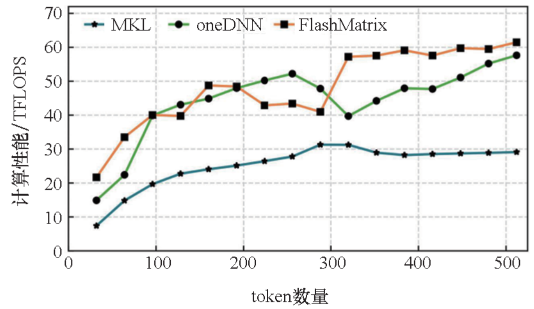


图 11 不同序列长度下 Attention 模块在 8468V 上的性能(Mixtral-8x22B)

Fig. 11 Performance of the Attention module on 8468V under different sequence lengths (Mixtral-8x22B)

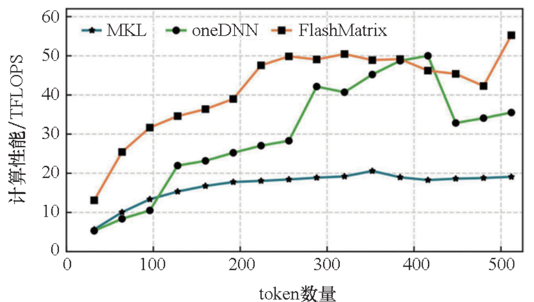
3.4 单个矩阵乘的性能

图 12 展示了单个 MM 在 8468V 上的性能。由图可知,FlashMatrix 的性能优势显著,相较于英特尔数学内核库(math kernel library, MKL)和 oneDNN,其平均加速比分别达到了 2.1 和 1.3。这主要得益于 FlashMatrix 消除了针对权重矩阵的布局转换开销,而该转换过程正是 MM 中最为



(a) Mixtral-8x22B 中的 MM

(a) MM in Mixtral-8x22B



(b) Qwen3-235B-A22B 中的 MM

(b) MM in Qwen3-235B-A22B

图 12 在不同 token 数量下矩阵乘的性能

Fig. 12 Performance of MM under different token counts

严重的性能瓶颈。此外,MKL 性能最差的原因在于,它仅支持 FP32 精度的输出,而 FlashMatrix 和 oneDNN 均支持 BF16 精度输出,这使得 MKL 在写回结果阶段产生了更大的开销。

3.5 FlashMatrix 与 GPU 上工作的性能对比

为了验证 CPU 在 MoE 推理中的重要作用,分别使用 FlashMatrix 在 6430 CPU 上,以及利用 cuBLAS[23] 在 A6000 GPU 上,对专家 FFN 的执行时间进行对比分析。A6000 的峰值算力为 158 TFLOPS,峰值访存带宽为 765 GB/s。从图 13 中可以看出,FlashMatrix 对每个专家 FFN 的运行时间比 cuBLAS 慢大约 1 ms。然而,FlashMatrix 无须进行专家权重的传输操作,而运行在 GPU 上的专家 FFN 则必须进行该操作。专家权重通过 PCIe 传输至 GPU 所耗费的时间,远远超过了专家 FFN 在 GPU 上的实际执行时间。

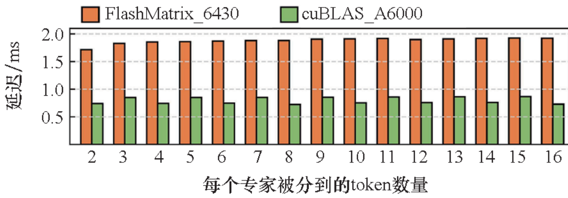


图 13　专家 FFN 在 CPU 和 GPU 上的运行时间对比

Fig. 13　Comparison of the execution time of the expert FFN on CPU and GPU

3.6 端到端的性能

表 3 展示了 Mixtral-8x7B [16] 模型在批量规模为 1 时的推理性能,该设置契合当前大模型本地化部署的典型使用场景[7]。IPEX 是 Intel 提供的用于加速模型训练与推理的深度学习优化框架,其默认实现在表 3 中标记为 IPEX;其底层算子可分别替换为 oneDNN 和 FlashMatrix,对应于表中的 IPEX + oneDNN 与 IPEX + FlashMatrix。上述三种推理方案均在单个 6430 CPU 上执行。相比之下,PyTorch 的实现则将全部推理计算任务卸载至 GPU,CPU 仅负责存储模型参数,其测试在 A100 GPU 进行。

实验结果表明,IPEX + FlashMatrix 方案取得了最佳的推理性能,这主要归功于 FlashMatrix 对 FFN 模块和 Attention 模块的深度优化,有效降低了由数据布局转换所带来的额外开销。尽管 PyTorch 将全部计算任务卸载至 GPU 执行,但受限于 PCIe 带宽与传输延迟,其数据搬移成本较高,从而使得整体推理性能显著低于基于 CPU 的 MoE 推理方案。

表 3　Mixtral-8x7B 模型的推理性能

Tab. 3　Inference performance of the Mixtral-8x7B model

推理方法	吞吐 tokens/s
PyTorch	0.82
IPEX + oneDNN	5.66
IPEX	6.10
IPEX + FlashMatrix	7.13

4 结论

为了适配 AMX 单元的数据布局要求,提出了高效的布局转换策略,并且在 cache 和寄存器中执行转换操作,显著减少了数据转换带来的访存开销。此外,设计了计算访存比最大的微内核来充分复用寄存器,以减少对 cache 和主存的访问。实验结果显示,FlashMatrix 相对于 CPU 上最先进的计算库 oneDNN,大幅提升了专家 FFN 和 Attention 模块的性能,为 MoE 在 CPU 上的高效推理提供了基础。

参考文献(References)

[1] GEORGANAS E, KALAMKAR D, VORONIN K, et al. Harnessing deep learning and HPC kernels via high-level loop and tensor abstractions on CPU architectures [C]//Proceedings of 2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2024: 950 – 963.

[2] BINDER E, SUDARSANAM A, SUNKAVALLI R, et al. FATHOM: fast attention through optimizing memory[C]//Proceedings of 2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2025: 1166 – 1178.

[3] FANG Z Y, HUANG Y G, HONG Z C, et al. Klotski: efficient mixture-of-expert inference via expert-aware multi-batch pipeline [C]//Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, 2025: 574 – 588.

[4] KAMAHORI K, TANG T, GU Y L, et al. Fiddler: CPU-GPU orchestration for fast inference of mixture-of-experts models [C]//Proceedings of International Conference on Learning Representations (ICLR), 2025.

[5] KIM H, WANG N C, XIA Q R, et al. LIA: a single-GPU LLM inference acceleration with cooperative AMX-enabled CPU-GPU computation and CXL offloading[C]//Proceedings of the 52nd Annual International Symposium on Computer Architecture, 2025: 544 – 558.

[6] CAO S Y, LIU S, GRIGGS T, et al. MoE-lightning: high-throughput MoE inference on memory-constrained GPUs[C]// Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, 2025: 715 – 730.

[7] CHEN H T, XIE W Y, ZHANG B X, et al. KTransformers: unleashing the full potential of CPU/GPU hybrid inference for MoE models[C]//Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles, 2025: 1014 – 1029.

[8] UXL. OneAPI deep neural network library (oneDNN)[EB/OL]. (2025 – 8 – 16) [2025 – 09 – 02]. https://github.com/oneapi-src/oneDNN.

[9] FU X, YANG W L, DONG D Z, et al. Optimizing attention by exploiting data reuse on ARM multi-core CPUs[C]//Proceedings of the 38th ACM International Conference on Supercomputing, 2024: 137 – 149.

[10] GEORGANAS E, AVANCHA S, BANERJEE K, et al. Anatomy of high-performance deep learning convolutions on SIMD architectures[C]//Proceedings of the SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, 2019: 830 – 841.

[11] PARK J, BIN K, LEE K. mGEMM: low-latency convolution with minimal memory overhead optimized for mobile devices[C]//Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services, 2022: 222 – 234.

[12] DE LIMAS SANTANA A, ARMEJACH A, CASAS M. Efficient direct convolution using long SIMD instructions[C]//Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, 2023: 342 – 353.

[13] YANG W L, FANG J B, DONG D Z, et al. LIBSHALOM: optimizing small and irregular-shaped matrix multiplications on ARMv8 multi-cores[C]//Proceedings of SC21: International Conference for High Performance Computing, Networking, Storage and Analysis, 2021.

[14] WANG P Y, YANG W L, FANG J B, et al. Optimizing direct convolutions on ARM multi-cores[C]//Proceedings of SC23: International Conference for High Performance Computing, Networking, Storage and Analysis, 2024: 1 – 14.

[15] Mistral AI. Mixtral: 8x22b[EB/OL]. [2025 – 09 – 03]. https://ollama.com/library/mixtral:8x22b.

[16] Mistral AI. Mixtral of experts[EB/OL]. [2025 – 11 – 20]. https://mistral.ai/news/mixtral-of-experts.

[17] DAO T, FU D Y, ERMON S, et al. FLASHATTENTION: fast and memory-efficient exact attention with IO-awareness[C]// Proceedings of the 36th Conference on Neural Information Processing Systems. 2022.

[18] GOTO K, VAN DE GEIJN R A. Anatomy of high-performance matrix multiplication[J]. ACM Transactions on Mathematical Software (TOMS), 2008, 34(3): 1 – 25.

[19] Intel. Intel® extension for PyTorch[EB/OL]. [2025 – 11 – 20]. https://github.com/intel/intel-extension-for-pytorch.

[20] Google. XNNPACK[EB/OL]. [2025 – 09 – 13]. https://github.com/google/XNNPACK.

[21] SHAH J, BIKSHANDI G, ZHANG Y, et al. FlashAttention-3: fast and accurate attention with asynchrony and low-precision [C]//Proceedings of Advances in Neural Information Processing Systems, 2024: 68658.

[22] QwenLM. Qwen3[EB/OL]. [2025 – 09 – 03]. https://github.com/QwenLM/Qwen3.

[23] NVIDIA. cuBLAS: basic linear algebra on NVIDIA GPUs[EB/OL]. [2025 – 10 – 25]. https://developer.nvidia.com/cublas/.