中国人民大学
RENMIN UNIVERSITY OF CHINA

论文编码：RUC-BK-专业代码

# 硕士学位论文

# 基于近存计算技术的矩阵向量乘算子优化研究


作　　者：张魁耀辉

学　　院：信息学院

专　　业：大数据技术与工程

年　　级：

学　　号：2023103756

指导教师：王晶


论文成绩：4.0

日　　期：2025 年 05 月 01 日

# 独创性声明

本人郑重声明：所呈交的论文是我个人在导师的指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写的研究成果，也不包含为获得中国人民大学或其他教育机构的学位或证书所使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

论文作者（签名）：_____ 日 期：_____

# 关于论文使用授权的说明

本人完全了解中国人民大学有关保留、使用学位论文的规定，即：学校有保留送交论文的复印件，允许论文被查阅和借阅；学校可以公布论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存论文。

论文作者（签名）：_____ 日 期：_____

指导老师（签名）：_____ 日 期：_____

# 授权书影印件

# 摘要

大语言模型在多个领域的成功应用推动了人工智能发展，但其存在计算成本高、内存需求大等缺点，优化其推理效率、降低资源消耗成核心问题。主流大模型多为 Decoder-Only 架构，推理生成分预填充和生成两阶段，其中 GEMV 算子在大模型推理中占主导地位，是主要性能瓶颈。GEMV 因自身特点为内存瓶颈任务，使 GPU 等硬件利用率低，在边端场景下常成系统瓶颈。存内计算是新兴计算范式，能解决传统架构内存瓶颈问题，UPMEM 作为较成熟商用存算一体硬件，有高带宽、高存储、高并行性、高能效等优势，也存在计算能力弱、通信开销大等局限。如何从软件角度适配硬件加速，以及如何修改设计硬件优化其应用性能的软硬协同优化方案成为待深入研究的课题。本论文基于 UPMEM 硬件研究算子 GEMV 的软硬协同加速方案：设计基于查找表的向量矩阵乘法避免浮点乘法运算，通过分块载入查找表到高速内存减少 DMA 访问次数增强高速内存的数据局部性，进行矩阵和向量的行列重排适应不同大小的矩阵减少访问查找表的次数并减少数据从寄存器流出，增强寄存器的数据局部性；同时基于 UPMEM 的周期精确模拟器 uPimulator 修改硬件，增加查找表专用 FMA 指令用于高效快速查询乘积并累加，设计基于查找表的 SIMD 指令进行向量化的查表和访存。最后对软硬协同优化设计了详细的基本测试，分别在 CPU 和 GPU 以及 UPMEM 三个平台上，进行了包括对总 GEMV 算子计算性能的测试和详尽优化细分测试（breakdown）测试，此外还根据硬件特性做了扩展性测试和能效比测试，充分论证软硬协同优化的有效性。

**关键词：**近数计算　矩阵向量乘　查找表

# Abstract

The successful application of large language models in multiple fields has driven the development of artificial intelligence. However, they have disadvantages such as high computational costs and large memory requirements. Optimizing their inference efficiency and reducing resource consumption have become core issues. Most mainstream large models are of the Decoder-Only architecture, and the inference generation is divided into pre-filling and generation stages. Among them, the GEMV operator dominates in the inference of large models and is the main performance bottleneck. Due to its own characteristics, GEMV is a memory bottleneck task, which leads to low utilization of hardware such as GPUs and often becomes a system bottleneck in edge scenarios. In-memory computing is an emerging computing paradigm that can solve the memory bottleneck problem of traditional architectures. UPMEM, as a relatively mature commercial in-memory computing hardware, has advantages such as high bandwidth, high storage, high parallelism, and high energy efficiency, but it also has limitations such as weak computing power and high communication overhead. How to adapt hardware acceleration from a software perspective and how to modify the design of hardware to optimize its application performance through a software-hardware collaborative optimization scheme have become topics for in-depth research. This paper studies the software-hardware collaborative acceleration scheme of the GEMV operator based on UPMEM hardware: designing a vector-matrix multiplication based on lookup tables to avoid floating-point multiplication operations, reducing the number of DMA accesses by loading lookup tables in blocks to high-speed memory to enhance the data locality of high-speed memory,

and rearranging the rows and columns of matrices and vectors to adapt to matrices of different sizes to reduce the number of accesses to lookup tables and the outflow of data from registers, enhancing the data locality of registers; at the same time, modifying the hardware based on the cycle-accurate simulator uPimulator of UPMEM, adding a dedicated FMA instruction for lookup tables to efficiently and quickly query products and accumulate, and designing SIMD instructions based on lookup tables for vectorized lookup and memory access. Finally, detailed basic tests were designed for the software-hardware collaborative optimization, including tests on the overall GEMV operator computing performance and detailed optimization breakdown tests on three platforms: CPU, GPU, and UPMEM. In addition, scalability tests and energy efficiency ratio tests were conducted based on hardware characteristics to fully demonstrate the effectiveness of the software-hardware collaborative optimization.

**Key Words :** Processing-in-Memory GEMV Lookup-Table

# 目录

# 图目录

# 表目录

# 1 绪论

## 1.1 研究背景和意义

近年来，大语言模型（Large Language Models，LLMs）在各个领域的成功应用，已然成为推动人工智能发展的重要趋势。以 OpenAI 的 GPT 系列为代表的大模型，不仅在自然语言处理（Natural Language Processing，NLP）领域表现卓越，更逐步被应用于金融、医疗、教育、科学研究等众多行业，成为赋能各行各业的核心技术。

大模型的优点在于其强大的生成能力和通用性，但也存在显著的缺点，比如高昂的计算成本、巨大的内存需求以及推理过程中的性能瓶颈。因此，在“大模型时代”，如何优化大模型的推理效率，降低资源消耗，已成为科研界和工业界亟待解决的核心问题。

当前主流的大模型（如 GPT 系列、Llama 等）均为 Decoder-Only 架构，即只使用 Transformer 中的解码器（Decoder）做生成[1, 2]。Decoder 采用自回归生成方式完成文本生成任务，其推理生成过程主要分为两个阶段：1）预填充阶段（Prefilling），该阶段会将提示词（Prompt）中所有词元（token）嵌入为词向量并输入 Decoder，主要执行 GEMM 运算 2）解码阶段（Decoding），该阶段会逐 token 地计算和生成，主要执行 GEMV 运算。在“GPT 式”大模型的整个推理过程中，生成阶段占据主导地位，有数据表明，其代表性算子 GEMV 平均占据 82.3% 的 GPU 运行时间，而预填充阶段的代表性算子 GEMM 的平均耗时只占 2%，剩下的一些非线性算子总占比不到 20%[3]。这使得 GEMV 算子成为大模型推理的主要性能瓶颈，针对 GEMV 的优化对于提升大模型的推理效率至关重要，具有重大研究价值。

与 GEMM 不同，GEMV 的计算过程数据重用性低，矩阵的计算近乎流式，存在大量数据搬移操作，计算访存比低。这些特点决定了其为内存瓶颈任务，使得常用的计算密集型硬件如 GPU 的利用率低，难以充分发挥其硬件优势。虽然有技术将多个推理请求组成一个批次（batch），进而将将 GEMV 操作合并成为 GEMM 操作以提高数据利用率[4]，但是在边端场景下，用户数量十分有限，batch size 绝大多数情况为 1，这时 GEMV 往往称为系统瓶颈[5]。

存内计算（Processing-in-Memory，PIM）是一种新兴的计算范式，其秉持将计算置于数据侧的以数据为中心的思想，旨在通过将计算单元集成至存储部件附近，以较高的传输带宽存取数据进行计算。采用此种架构的硬件往往能凭借其独有高速传输通道和简单存储结构的具备高存储高带宽和低能耗的优势，能够充分解决传统冯诺依曼计算架构中的内存瓶颈问题。

近些年来许多 PIM 硬件被设计和提出[6, 7, 8, 9, 10]，在这其中 UPMEM 是目前较为成

熟的商用存算一体硬件，其硬件特点包括：1）高带宽和高存储，聚合带宽能达到 TB 级别。2）高并行性，拥有 2560 个 DPU，每个 DPU 24 个线程可以并发控制，能够以极细的粒度分割任务。3）高能效，存算一体架构减少了数据搬运的能耗开销。同时 UPMEM 也存在一些局限性：1）较弱的计算能力，对于浮点和乘除法缺乏硬件支持。2）通信开销大，与主机或 DPU 之间通信开销大。

UPMEM 的优势使得其特别适合卸载 GEMV 算子进行加速：高带宽和高存储可以有效解决 GEMV 算子乃至大模型推理过程中的内存瓶颈问题，同时 GEMV 中的矩阵和向量可以任意粒度切分且无数据依赖可以充分并行。但是 UPMEM 较弱的计算能力使得在加速 GEMV 时不得不做一些设计避免性能劣化。如何从软件角度适配硬件加速，以及如何修改设计硬件优化其应用性能的软硬协同优化方案成为待深入研究的课题。

## 1.2 国内外研究现状

早在上世纪七十年代左右，存内计算的思想就已经初具雏形[5, 11]，其核心思想是试在 DRAM 的存储单元上增加简单的逻辑电路使得其本身可以进行一些简单的运算。上世纪九十年代，Wulf 等人系统性地定义了内存墙（Memory Wall）的概念并对其进行了分析[12]。许多学者围绕该问题提出了不同的解决方案。其中，有部分学者提出存内计算（PIM，Processing in Memory）的思想，希望通过在存储器原地进行计算从而减少 CPU 的访存以达到更高的性能和更低的能耗。在从那开始有不少的工作被提出，如 RowClone[13]这篇文章提出在 DRAM 的同一 bank 内同时打开多行并利用共享的行缓存器（row buffer）实现行之间的快速复制，这种复制无需 CPU 参与数据搬移，大大提升了复制的效率。此外 Seshadri 等人[14, 15]提出一系列工作，通过利用内存单元本身的模拟特性以及对感应放大器（sense amplifier）的简单修改，实现了快速批量的按位与（AND）、或（OR）、非（NOT）等逻辑操作，计算完全发生在 DRAM 内部，不占用系统总线。

尽管还有相当一部分此类的工作被提出，但是时代的局限性使得 PIM 的工作难以落地。一方面是当时的制造工艺无法在内存芯片内集成较为复杂的逻辑单元。另一方面，在内存墙概念被提出的九十年代，互联网的数据量远不如现在庞大，没有弃用原先普通内存，更换造价更加相较高昂 PIM 型内存的迫切需求。

进入大数据时代以来，数据量逐渐增大，数据密集型场景逐渐增多，大量且频繁数据搬移造成的高延迟和高能耗等问题日渐凸显。此时存内计算被重新提出，其与存储侧计算的思想与大数据时代信息处理的特征不谋而合，重新受到了研究人员的青睐[16]。

与此同时，硬件方面的新进展为存内计算的复兴提供了坚实的土壤，3D 堆叠技术的出现极大程度上解决了此前 PIM 的逻辑集成难题，使得在同一块面积的芯片上集成更复杂高效的逻辑单元成为可能。3D 堆叠技术纵向堆叠内存芯片，形成多层结构：多层数据层堆叠存储数据，底部堆叠逻辑层执行计算，层与层之间通过硅穿孔 (Through-Silicon Vias, TSV)TSV 形成垂直互联，能够高效传输数据，提供极大的内部存储带宽。

在此背景下，许多工作被提出，比如 Junwhan Ahn 等人[17]提出 Tesseract 使用 3D 堆

叠技术加速大规模的图处理。与此同时由于人工智能浪潮的兴起，许多专用于神经网络的PIM 加速芯片也被设计出来，其中较为著名的就是三星的 HBM-PIM 产品[6]，该产品采用20nmDRAM 工艺，使用 3D 堆叠技术堆叠封装了 4 层裸芯（Die），每层 Die 的 bank 分组内集成了专门用于 16 位浮点数乘加操作的 PIM 单元以处理神经网络中的矩阵操作。此外三星的另一个产品 AxDIMM[7]也采用了存内计算/近存计算的技术，其将 DRAM 芯片（Chip）和 FPGA 处理单元集成到一块有着 DDR4 标准接口的主板上，其主要用于加速推荐模型（recommendation model）中的向量嵌入查找任务（embedding lookup）。海力士也提出过存内加速器 AiM[8]用于加速 AI，与三星的 HBM PIM 不同的是，AiM 基于 GDDR6 内存，每个 bank 集成 PIM 单元，通过设计互连总线和全局缓存实现各个 PIM 单元的高效通信。在国内方面，阿里也推出过近存计算产品[9]用于加速 AI 任务，同样采用的是 3D 堆叠技术将逻辑 Die 和数据 Die 堆叠封装在一起，通过 TSV 高速传输数据。逻辑 Die 上分别设计了用于向量排序和矩阵乘法的计算单元用于处理不同类型的任务。

然而上述工作因为各种复杂的原因难以落地使用和量产，甚至部分基于模拟器，使得PIM 技术的推广与使用犹如空中楼阁。近几年，一款号称真正可商用的 PIM 硬件横空出世：UPMEM 作为以第一款可以商用的近存计算处理器产品[10]，有着更加通用的处理能力、高速的内存带宽、低廉的接入成本以及完备的开发生态。UPMEM 本身是一条有着标准 DDR4 接口的内存插块（DIMM），可以像正常的内存条一样插在通用的 Intel 服务器上（一个服务器最多可以插入 20 条 UPMEM）。每个 UPMEM 插块包含两个 rank，每个 rank 有 64 个数据处理单元（Data Processing Unit，DPU）。每个 DPU 都拥有一个 14 级流水的 RISC 处理器，拥有 24 个线程。同时每个 DPU 拥有二级存储结构，包括 64KiB 的 SRAM（称为 WRAM）和 64MiB 的 DRAM（称为 MRAM）。

有许多工作对 UPMEM 的硬件特征做了系统且全面的测试[18, 19, 20, 21]，其中，Mutlu 等人[18]的测试工作较为全面且权威。在这些评测中，不难发现 UPMEM 的硬件优势可以概括为以下几点：1）高存储容量和传输带宽。20 条 UPMEM 共有 2560 个 DPU，其可用内存达到 160GB，远超市面上常用显卡的显存，其中 MRAM 的传输带宽大约为 700MB/s，当 2560 个 DPU 并行工作其聚合带宽能够达到 1.7TB/s；2）高并发和细粒度并发控制，2560 个 DPU 可以同时工作，每个 DPU 内部还可以控制 24 个线程进行更加细粒度的控制，整个 PIM 系统的并发线程数量能够达到近 3 万；3）高能效比，UPMEM 设备没有复杂的存储结构，减少数据的搬移可以大大降低能耗，测试表明，UPMEM 的能效比高于运行相同任务的 CPU 和 GPU[18]。拥有以上优点的同时，UPMEM 的缺点也十分明显：1）硬件资源十分有限，只支持 32bit 的整数加减法，其他的算术操作包括乘除和，浮点运算都是软件实现，效率较为低下；2）通信效率低，UPMEM 与主机端通信的传输效率不高，只有大批量传输连续数据时才能勉强达到 DDR4 内存的传输带宽，同时 DPU 之间彼此独立缺乏有效的通信手段，只能通过 CPU 主动中转数据，效率更加低下。这些硬件特性有优有劣使得任何基于 UPMEM 构建的应用都要充分利用或避免这些硬件特性。

UPMEM 自出现以来有许多研究者以此硬件结合相关应用做出了许多工作，涉及、数

据库[22, 23, 24, 25, 26, 27, 28, 29]、生物基因[30, 31, 32, 33, 34]以及人工智能[35, 36, 37, 38, 39, 40, 41]等各个领域。其中，UPMEM 的高并行和通信效率低的特性使得其非常适合用于加速神经网络的推理。Niloofar Zarif 将 UPMEM 用于 embedding lookup 任务的卸载[35]，对于目前较大的嵌入表（embedding table）加速效果尤为明显。Juan Gómez-Luna 等人[36]以简单直接的方式卸载了传统机器学习中的基础模型到 UPMEM 上，包括线性回归、逻辑回归、决策树、K 均值聚类，并做了全面丰富的测试，但是测试结果无一表明这些模型的推理都遭受了严重的计算性能瓶颈。Prangon Das[37]等人在 UPMEM 上分别卸载了嵌入二值神经网络（Embedded Binary Neural Network，eBNN）和 YOLOv3（主要是卸载 CNN 的卷积操作），其主要思想是将卷积神经网络（CNNs）的权重量化到低 bit 位，再通过查找表（Look Up Table，LUT）查询低 bit 浮点数乘积，以消除浮点乘法运算，但这会严重降低模型的精度。最与本课题应用场景相近的工作 PIM-DL[39]使用 UPMEM 推理 Bert，其通过将矩阵乘法转换为最近邻查找和向量加法，减少了对乘法的需求，从而提高了计算效率。但其最近邻查找是在 CPU 上完成的，而 UPMEM 只执行向量加法操作，并没将计算重担完全卸载到 UPMEM 上。

结合上面的文献不难看出，使用 UPMEM 推理神经网络的主要难点在于其羸弱的计算能力往往拖累系统整体性能，无法充分发挥 PIM 架构的高带宽，如何简化和避免复杂的计算是十分值得研究的课题。

## 1.3 研究内容和创新点

本毕业设计主要内容分为三个主要研究内容：

1) 基于 8bit 量化查表的 GEMV 算子算法优化，根据 UPMEM 自身硬件特点基于查找表设计 GEMV 算子，主要优化 GEMV 对 UPMEM 各级存储层次的访存策略以达到较高的数据局部性，分别包括在 MRAM 层面和 WRAM 层面的优化。

2) 基于 UPMEM 周期精确模拟器的硬件改动优化，根据 GEMV 查找表等软件实现的特点设计硬件，增加查找表专用 FMA 指令加速乘积查询，设计 SIMD 指令支持向量化查表和计算。

3) 从四个方面设计基于三个平台 CPU（MKL）、GPU（cuBLAS）和 UPMEM 的一系列对比实验：GEMV 算子的整体计算性能，各种优化手段（包括软硬协同）的提升细分测试（breakdown），UPMEM 的扩展性实验以及和 CPU、GPU 等多个平台的能效比实验，综合全面论证优化有效性。

同时，主要创新点为以下三个：

1) 通过文献阅读充分了解了 UPMEM 硬件的特点，设计基于查找表的向量矩阵乘法，通过分块载入查找表到高速内存减少 DMA 访问次数增强高速内存的数据局部性，进

行矩阵和向量的行列重排适应不同大小的矩阵减少访问查找表的次数并减少数据从寄存器流出，增强寄存器的数据局部性，提升 GEMV 算子的性能。

2) 基于 UPMEM 的周期精确模拟器 uPimulator 修改硬件，增加查找表专用 FMA 指令用于高效快速查询乘积，向量化访存减少读放大问题，设计 SIMD 指令访存向量化提高读写效率。

3) 对软硬协同优化设计了详细的基本测试，在 CPU 和 GPU 以及 UPMEM 三个平台上，进行了包括对总 GEMV 算子计算性能的测试和详尽优化细分测试（breakdown）测试，此外还根据硬件特性做了扩展性测试和能效比测试，充分论证软硬协同优化的有效性。

## 1.4 论文组织结构

本文共六个章节组成，其中每个章节的主要内容如下：

第 1 章是全文的绪论。主要介绍了本文研究背景与意义、国内外研究现状、研究内容与创新点，并在最后介绍了全文的组织结构。

第 2 章是相关工作。主要进行研究工作的综述，分别介绍了大模型推理加速相关技术和工作以及近存计算研究现状，梳理了已有文献的研究情况，从理论起源和内涵出发到研究现状。

第 3 章是基于模型量化和查找表的矩阵向量乘软件优化。本章基于 UPMEM 硬件平台对 GEMV 算子提出了相关软件的设计和优化算法，基础算法是基于分块查找表的载入算法，更进一步对不同矩阵尺寸进行了设计，为窄和宽矩阵分别做出行重排和列重排的优化算法。

第 4 章是基于近存计算模拟器的矩阵向量乘硬件优化。本章主要选取了 UPMEM 的周期精确模拟器 PIMulator 对 UPMEM 做硬件改动，包括增加查找表专用的 FMA 指令加速查表乘积累加的计算，以及增加 SIMD 单元支持向量化查表和访存。

第 5 章是实验结果与分析。本章主要设计了四个实验包括不同平台 GEMV 算子的总吞吐、总能效比、各种优化手段的细分（breakdown）以及算子的扩展性等等，通过分析实验结果说明研究工作的有效性，并对硬件本身特性做进一步分析和展望。

第 6 章是总结与展望。本章将对全文内容进行总结。本章首先对全文研究结果进行总结。接着，本章阐述了本研究在理论和实践贡献。最后，本章总结了本研究的不足，并提出了未来工作展望。

# 2 相关工作

本章节首先介绍大模型推理相关概念和工作，包括对现有流行大模型的架构和算子的介绍，对推理加速的技术尤其是模型量化的相关工作进行介绍，以及阐明推理性能和矩阵向量乘算子的关系。其次介绍近存计算的研究现状，首先介绍近存计算的发展历史以及相关工作，然后介绍第一款商用近存计算硬件 UPMEM 的硬件架构和特性，最后详细介绍各个领域基于 UPMEM 加速应用的相关工作。

## 2.1 大模型推理介绍

### 2.1.1 基本概念

自 2022 年 11 月 OpenAI 发布聊天机器人产品 ChatGPT 以来，其强大的语言理解对话能力和逻辑推理能力震惊了世界，并以势不可挡之式席卷全球，并受到了资本的热捧。其背后拥有强大智能的 GPT（Generative Pre-trained Transformer）大语言模型（Large Language Model，LLM）凭借惊人的通用智能应用到了各行各业，开启了 LLM 时代。

GPT 模型的结构源于 Google 在 2017 年提出的一个基本模型结构 Transformer [42]。Transformer 模型本身是为解决序列到序列（Seq2Seq）的自然语言翻译问题，构造了编码器（Encoder）模型用于理解待翻译序列，再使用解码器（Decoder）模型结合 Encoder 的理解生成翻译序列。在 Encoder 和 Decoder 中创造性地使用了自注意力（Self-Attention）机制，使得模型的性能表现异常优异，击败了当时其他的翻译模型。基于 Transformer 结构，自然语言处理（Natuarl Language Processing，NLP）领域的模型逐渐形成了三条不同的发展路径，分别是 Encoder-Only 架构、Encoder-Decoder 架构和 Decoder-Only 架构。Encoder-Only 架构的模型非常擅长做语义理解，其应用领域往往是文本分类，其中最著名的模型当属 Bert[43]；Encoder-Decoder 架构和 Transformer 的应用领域类似，主要是用在机器翻译领域，其中比较著名的模型同是 Google 公司的 T5[44]。Bert 和 T5 模型在前 GPT 时代基本确定了 NLP 模型的预训练 + 微调的部署范式：由大公司使用大量数据集训练基础通用模型（预训练模型），在此基础上由使用者自行用少量高质量数据微调模型以适应不同的下游任务。

Decoder-Only 架构的模型与前面两者不同，可以直接自回归地生成自然语言，常常应用到文本续写领域和聊天机器人，其中代表模型就是 OpenAI 的 GPT 模型。最早的 GPT-1[1]仍然采用有监督和无监督混合训练的方式直接生成下一个词元，效果并不是很理想。到了 GPT-2[45]，GPT 模型开始使用无监督的训练方式，加大了训练数据量并提升模型参

数规模到 150 亿，无须微调模型而只需简单地通过写提示词（Prompt）就可以达到非常好的效果。到了 GPT-3[46]，进一步将模型参数量提升到了惊人的 1750 亿，并专注于提升模型的上下文学习能力（In-Context-Learning，ICL）。量变引起了质变，GPT-3 的效果非常优异，为接下来的 GPT3.5 以及 ChatGPT 的爆火做了铺垫。得益于 Decoder-Only 架构模型在大量数据集上的良好零样本（Zero Shot）学习能力[47]和少样本（Few Shot）微调能力[48]，大模型的参数规模不断增长（从前 GPT 时代 Bert 的 1.1 亿参数到 GPT-3 的 1750 亿），参数量的膨胀对于大模型推理提出了严峻的挑战，大公司旗下的云计算中心需要使用集群分布式推理上千亿大模型，常见的硬件几乎无法部署推理参数量最小的 70 亿参数模型。

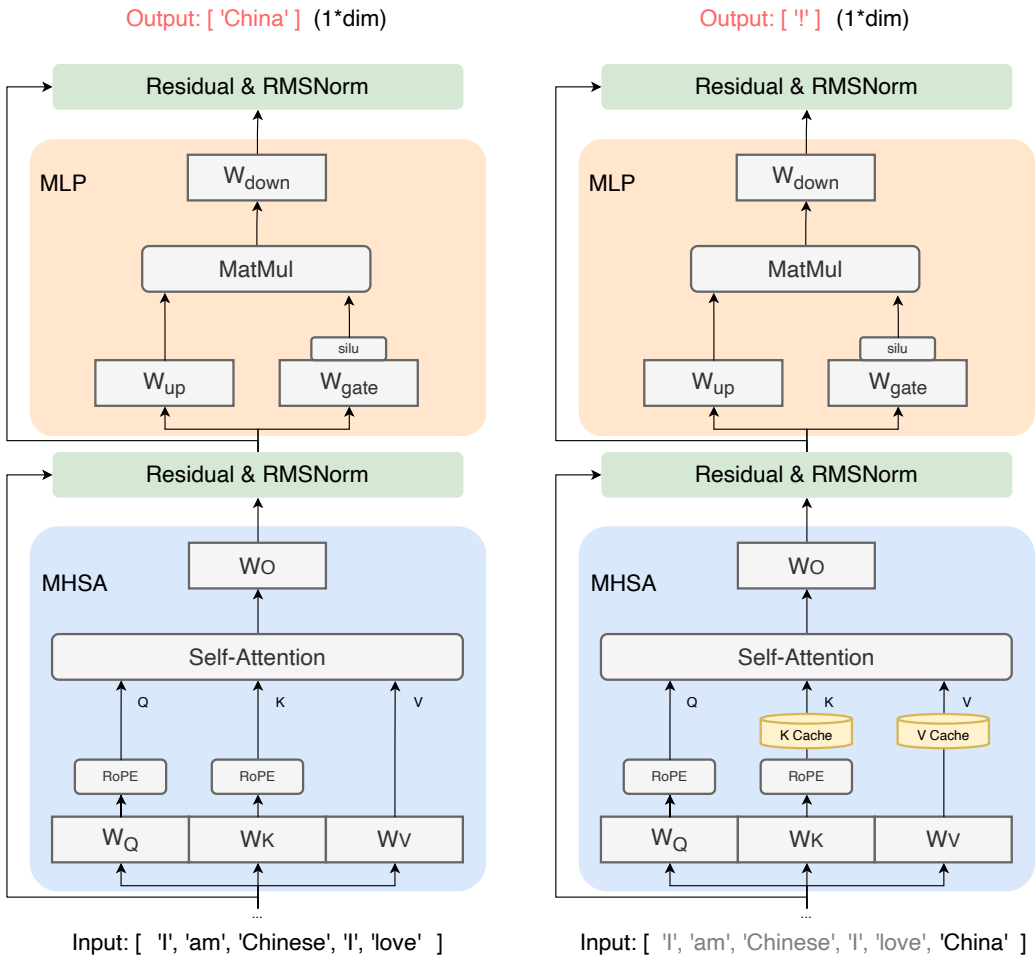



**图 2.1** 现代主流大模型推理示意图。图左为 Prefill 阶段的单层 Decoder 的计算，图右为 Decode 阶段对应的计算

要想了解大模型在常用硬件平台上的推理瓶颈，就需要先了解大模型的结构，从开源的大模型入手是最佳选择。其中使用最广微调性能最高的开源大模型当属 Facebook 推出的 Llama 系列。Llama 系列的模型架构都秉承类似的结构，以 Llama2-7B[2]为例，模型一般由多层 Decoder 组成，每层 Decoder 分为多头注意力（Multi-Head Self-Attention，MHSA）

和前馈神经网络（Feed-Forward Neural Network，FFN）两个主要部分。在 LLM 进行推理的时候，分为两个阶段：用户输入提示词（Prompt）进入网络到模型生成第一个词元（Token）的阶段称作预填充阶段（The Prefilling Stage），再生成了第一个 token 之后，LLM 不断自回归生成下一个 token 直到输出终止符的阶段称为解码阶段（The Decoding Stage）。如图2.1所示，两个阶段执行的计算并不相同，主要体现在 Self-Attention 部分，预填充阶段执行的是通用矩阵矩阵乘（Generalized Matrix Multiplication，GEMM），而在解码阶段执行的是通用矩阵向量乘（Generalized Matrix Vector Multiplication，GEMV）。两者不同的原因主要在于现代 Decoder-Only 架构的 LLM 在自回归生成阶段普遍采用因果注意力（Causal Self-Attention）并通过 KVCache 机制减少计算量，如图2.2所示，因果注意力会使用因果掩码对 QK 乘积矩阵的上三角置 0：当生成第四个 token 时，Query1 和 Key2 到 Key4 点注意力都被掩码置 0 无效化，为的是防止其与未来生成的 token 做注意力计算以保持因果一致性。第四个 token 进行推理时真正的新数据就是 Query4、Key4、Value4 和 Attention4。因而 KVCache 就是将 K 向量和 V 向量缓存起来，每次只需要最新生成的 Query 向量参与运算，减少重复的注意力计算，此时 Self-Attention 的计算为 GEMV。

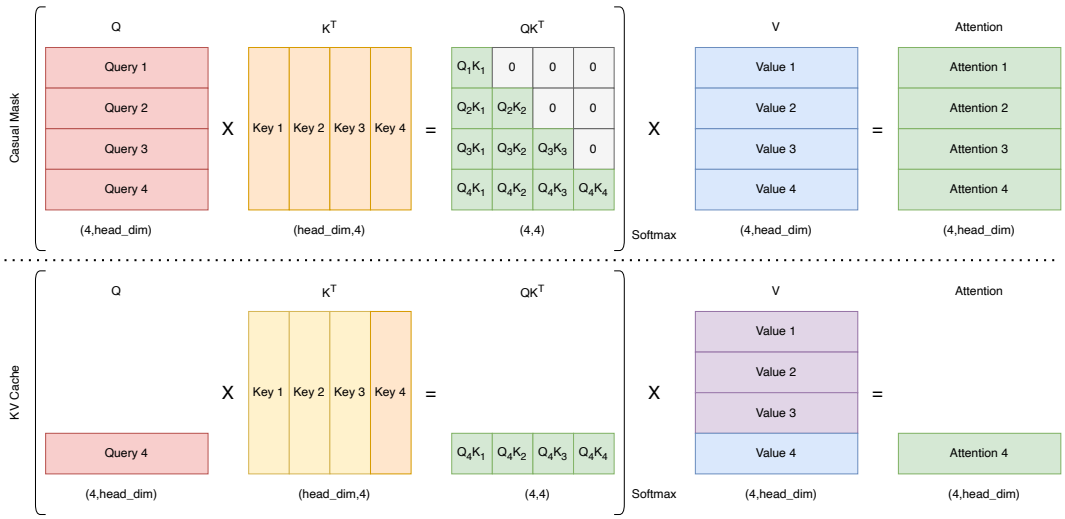



**图 2.2** Self-Attention 计算简介。图左为 Prefill 阶段，使用因果掩码，图右为 Decoder 阶段，使用 KVCache

大模型的推理往往表现为内存瓶颈，尤其是在用户量不多的本地或边端场景，推理中的解码阶段占据主导地位，有数据表明，解码阶段代表性算子 GEMV 平均占据 82.3% 的 GPU 运行时间，而预填充阶段的代表性算子 GEMM 的平均耗时只占 2%，剩下的一些非线性算子总占比不到 20%；同时在执行 GEMV 算子时，GPU 的硬件利用率显著地低于 GEMM 算子，并且将绝大部分的时间消耗在内存拷贝和数据传输上，表现为内存瓶颈[3]。这使得缓解内存数据的搬移成为大模型推理的主要性能瓶颈，针对 GEMV 的优化对于提升大模型的推理效率至关重要，具有重大研究价值。

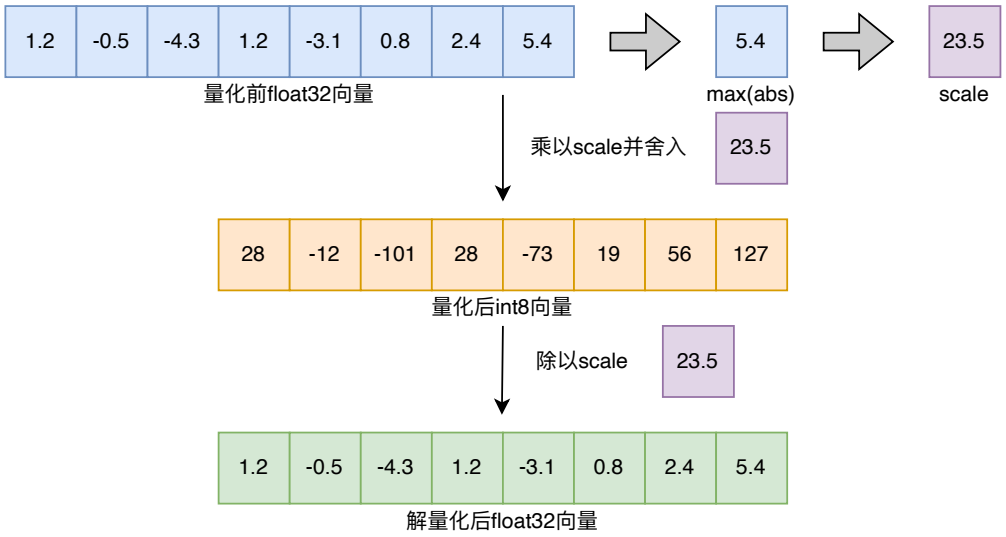
### 2.1.2 大模型量化技术

加速大模型推理有许多手段，包括模型量化、模型压缩、矩阵稀疏化、知识蒸馏、内存管理和批处理等等技术[49]，其中最流行以及能够显著降低模型的内存瓶颈的方法当属模型量化。所谓的量化就是指将模型的高 bit 参数（float32）通过一系列措施（最小化推理精度损失）转化为较低的位宽存储。这样做能够极大缩减模型尺寸，在硬件资源有限的情况下支持低精推理，使得本地大模型或边端大模型成为可能。以一组 float32 向量量化成 int8 向量为例，如图2.3(a)所示，取得这组 float32 向量的最大绝对值 5.4，将其映射到 int8 的动态范围 $[-128, 127]$，计算得到缩放因子 scale 为 23.5，将向量的每个元素与 scale 相乘并做整数舍入得到 int8 向量，此时完成量化；当需要量化后的向量参与计算时需要先解量化，将 int8 向量的每个元素与 scale 相除进行解量化，还原成 float32 向量。

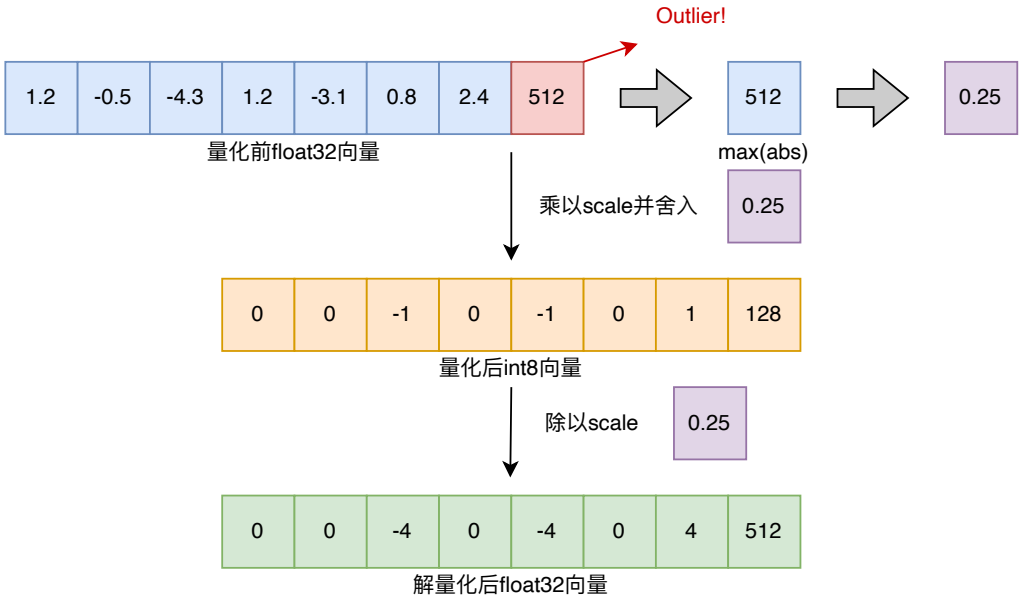
上述量化是非常标准的对称量化：所谓对称量化就是取得原数值域的一个对称区间量化到目标数值区域，一般使用最大绝对值确定对称区间。而与之相对的非对称量化就是量化原数值域的非对称区间，使用的是最小值和最大值；对称量化是非对称量化的一种特殊情况，二者都可以用公式2.1来表示，其中 r 是原始浮点数，S 是缩放因子 scale，Z 是零点偏移，q 是量化后的数值。当为对称量化时，零点无偏移，$r_{max} - r_{min} = 2r_{max_abs}$。量化一般采用训练后量化 (Post Training Quantization，PTQ) 的方式以降低量化成本（简单的数学变换或者少量校准数据集），主要的量化对象就是激活值和权重，按照上述的方式将激活和权重全部量化到 8bit 的量化称为 W8A8（weight 8bit activation 8bit）量化。但是假如激活向量或权重矩阵中，存在一个特别大的值如图2.3(b)所示，为 512，那么 scale 计算就得到为 0.25，将原 float32 向量与 scale 相乘再舍入取整为 int8，发现绝大部份数都变为了 0，0 在浮点数制中是一个特殊值，任何数与 0 相乘或相除都是 0，因此将 int8 向量反量化后，float32 向量中的大部分数据也都变为 0，从而导致精度严重损失。

$$q = round\left(\frac{r}{S} + Z\right), \quad S = \frac{r_{max} - r_{min}}{q_{max} - q_{min}}, \quad Z = q_{min} - \frac{r_{min}}{S} \tag{2.1}$$

为了解决这种离群值带来的量化精度影响，许多工作提出了不同的方案。其中比较出名的 int8 量化方案为 Dettmers 等人提出的 LLM.int8[50]，该方法的主要思路在于离群值分布往往在特定的维度且比较稀疏，因此可以将离群值提取出来用 float16 进行计算，剩下的量化到 int8，这样的混合精度分解能够很大程度提升量化精度。SmoothQuant 采用了不同的思路[51]做 W8A8 量化，其观察到当前的 LLM 的激活相对权重难以量化，因此提出了一种数学上等价的逐通道缩放变换，引入一个对角矩阵存放激活逐通道的缩放因子 scale factor，将原激活除以对应的缩放因子作为新的激活，原矩阵等价地乘以对应的缩放因子作为新的权重，这样就能将激活中难以量化的离群值平滑到了权重当中，二者都变得容易量化了。Lin 等人提出了一种 W4A16 的量化（仅量化权重）[52]，其核心思想是基于激活值的分布挑选对最终结果影响较大的权重，将这些显著权重保留精度分解，剩余权重采用低 bit 量化，以较小的量化误差达到大幅度减少内存占用的效果。在量中同样有一类工作可以实现非常低 bit 的量化，即基于机器学习的量化，其中最具代表性的当属 GPTQ。GPTQ 本




(a) 正常对称量化




(b) 含有离群值的量化

**图 2.3** Float32 向量对称量化为 Int8 向量示意图

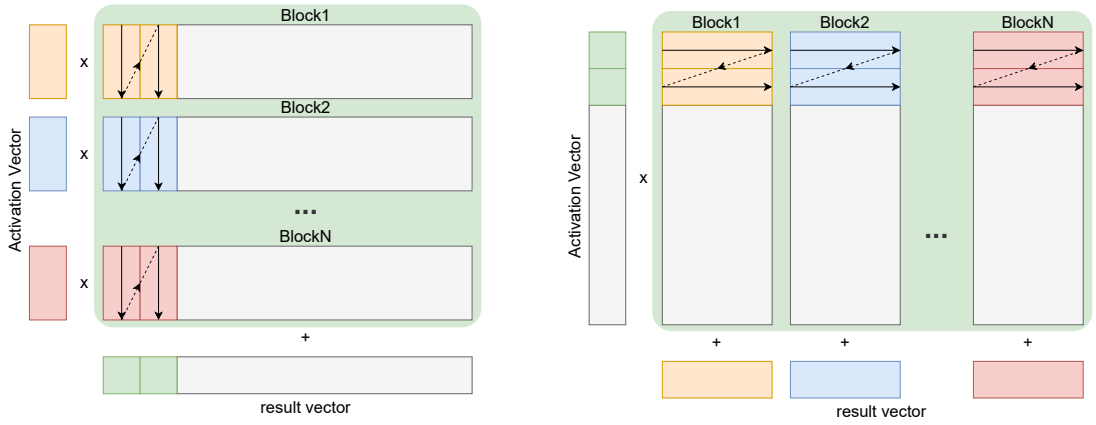
身的核心思想是对 LLM 进行逐层量化，希望在该层找到一个新权重（量化过后），使得其和使用老权重相比输出之间的误差尽可能小。GPTQ 将这个作为训练目标使用机器学习的方式进行优化，基于此前的 OBQ 工作[53]，对训练算法做了近似和加速，能够达到 3-4bit 的超低精度量化。Yao 等人开展了一系列 LLM 上的量化工作：ZeroQuant 系列[54, 55, 56]。ZeroQuant-V1 主要是针对 GPU 硬件构建了强大的推理后端并使用逐层知识蒸馏缓解量化带来的精度下降问题[54]；ZeroQuant-V2 则是针对常见的不同的 PTQ 方法进行了全面的分析，并提出了一种低秩补偿的技术来缓解量化带来的误差[55]；ZeroQuantFP 基于 GPTQ 的量化和低秩补偿，重点探索来浮点数据格式对于量化的影响，得出 float8 激活优于 int8，float8 权重与 int8 权重相当，float4 权重优于 int4 权重的结论[56]。

### 2.1.3 矩阵向量乘算子

GEMV 作为基本的线性代数运算被广泛地使用于各种科学计算和神经网络计算当中，其本身可以视作 GEMM 中矩阵维度为 1 的一种特例。关于 GEMM 和 GEMV 的计算加速优化方法一直是研究人员的研究热点，因为 GEMM 和 GEMV 作为 BLAS（Basic Linear Algebra Subprograms）中最为基本的两个，其性能的提升将极大提升建立于其基础上的应用性能。GEMM 的优化主要分为两个方面，分别是数学算法上的优化和计算机系统层面的优化。前者主要是在数学算法上减少 GEMM 需要执行计算量，其中最经典的算法是 Strassen 算法[57]，其将相乘的每个矩阵分别分解为大小相同的四个子矩阵，通过对四个子矩阵执行一系列的矩阵加法和乘法运算得到最终的乘积矩阵，成功将矩阵乘法的时间复杂度从 $(n^3)$ 优化到 $(n^{\log_2 7})$，但是由于 GEMV 中向量无法进行有效地分块，因此 Strassen 算法无法直接应用到 GEMV 上的优化；另一种，也是工作最多且更加有效的优化，通过将矩阵进行合适的分块和内存布局，减少计算机在执行计算过程中的重复的和开销大的访存，增强数据的局部性。常见的工作都是根据所使用硬件的离计算核心最近的一级存储器件（比如 CPU 中是 L1 Data Cache，GPU 中 CUDA 的 shared memory[58]）的存储大小，对矩阵进行合适大小的分块以在执行每个小块的计算时访存都能落到最近的存储器而无需访问更低速的内存。

GEMV 的常规计算方式有两种，如图2.1.3所示，可以类似向量的内积和外积定义 GEMV 的内积和外积概念。如图2.4(a)，GEMV 的内积可以定义为向量和矩阵的一行或一列的对应元素乘积之和，结果是最终结果向量的一个元素。如图2.4(b)，GEMV 的外积可以定义为向量的某个元素与矩阵对应行或列的所有元素乘积，结果是一个向量，将所有向量对应相加得到最终结果向量。在传统 CPU 平台下，GEMV 的内积效率更高，因为其数据局部性更好，内积的累加结果基本能够留存在寄存器中，不需要频繁读写结果向量；当然 GEMV 的外积在某些场景下也有用武之地。除了考虑寄存器效率外，高速缓存（Cache）的命中率也非常重要：无论是何种方式，只需要 Cache 容量大于矩阵的两行或两列所占的空间，命中率就会很高，如果矩阵的两行或两列所占的空间过大，则需要将矩阵或向量在某一个维度上进行分块，以达更高缓存命中率。分块后，对于大多数的存储器来说，顺序读取

效率是最高的，因此需要将数据按照顺序访问的方式进行重排，称为数据打包。如2.1.3所示，将矩阵按行或列分成了 N 块，对每一块进行数据打包依次计算。




(a) GEMV 内积　　(b) GEMV 外积

**图 2.4** GEMV 基本计算和优化方法

常规的优化手段如上所述，当然还可以使用多线程进行优化，此时只需要考虑各个线程的数据划分和线程间的同步开销即可。除此之外许多相关硬件被设计并对其加速提供了支持，这其中包括基于单指令多数据（Single Instruction Multiple Data，SIMD）的 Intel 的向量单元和 AVX 指令集，可以在 0.5 个周期内完成 512bit 向量的融合乘加运算（Fused Multiply Accumulate，FMA）操作[59]。还有 Nvidia 基于单指令多线程（Single Instruction Multiple Threads，SIMT）模型设计的 CUDA Core[58]，以及专门用于 GEMM 计算的 Tensor Core[60]等等，都极大地提升了 GEMM 和 GEMV 的计算效率。

## 2.2 近存计算研究现状

### 2.2.1 近存计算技术发展

早在上世纪七十年代左右，近存计算的思想就已经[5, 11]初具雏形，这时普遍提到的概念是“Logic-in-Memory”，其核心思想是在动态随机存取存储器（Dynamic Random Access Memory，DRAM）的存储单元上增加简单的逻辑电路使得 DRAM 本身可以进行一些简单的运算。在九十年代，Wulf 等人针对处理器（CPU）和存储器（DRAM）之间不断增大的速度差异，通过科学的建模和实验进行了分析，通过系统性的分析和实验，提出了内存墙（Memory Wall）的概念以说明 CPU 和 DRAM 之间速度存在的难以逾越的鸿沟[12]。许多学者围绕该问题提出了不同的解决方案，其中，有部分学者提出存内计算（Processing in Memory，PIM）的思想，希望通过在存储器原地进行计算从而减少 CPU 的访存以达到更高的性能和更低的能耗。同年就有工作[61]，该结构通过在存储阵列旁加了一些计算单元 (例如 ALU), 用于支持存储阵列内部的数据处理。又如 RowClone 这篇工作[13]提出在 DRAM

的同一存储体（Bank）内同时打开多行并利用共享的行缓存器（Row Buffer）实现行之间的快速复制，这种复制无需 CPU 参与数据搬移，大大提升了复制的效率。此外还有 Seshadri 等人[14, 15]的一系列工作，通过利用内存单元本身的模拟特性以及对感应放大器（Sense Amplifier）的简单修改，实现了大批量的按位与（AND）、或（OR）、非（NOT）逻辑操作，由于计算完全发生在 DRAM 内部，因此不占用内存带宽，可以达到非常高的吞吐。

尽管还有相当一部分此类的工作被提出，但是时代的局限性使得 PIM 的工作难以落地。一方面是当时的制造工艺无法在内存芯片内集成较为复杂的逻辑单元。另一方面，在内存墙概念被提出的九十年代，互联网的数据量远不如现在庞大，没有弃用原先普通内存，更换造价更加相较高昂 PIM 型内存的迫切需求[16]。

本世纪 10 年代以来，人类社会进入大数据时代，数据量呈现指数级爆炸，而且由于人工智能的兴起，数据密集型场景逐渐增多，大量且频繁数据搬移造成的高延迟和高能耗等问题日渐凸显：跨内存层次结构移动数据的能耗将比执行双精度浮点运算的成本高出两个数量级[62]。想要消除这种不必要开销的迫切需求日益增长，计算机系统逐渐从以计算为中心的架构向着以数据为中心的架构发展。此时存内计算被重新提出，其与存储侧计算的思想与大数据时代信息处理的特征不谋而合，重新受到了研究人员的青睐。

与此同时，硬件方面的新进展为存内计算的复兴提供了坚实的土壤——3D 堆叠技术（3D-Stacking）的出现极大程度上解决了此前 PIM 的逻辑集成难题，使得在同一块面积的芯片上集成更复杂高效的逻辑单元成为可能。如图2.5所示，3D 堆叠技术纵向堆叠内存芯片，形成多层结构。3D 堆叠的内存立方体在最底层集成逻辑层，为存储侧计算单元提供设计空间，逻辑层上方即是堆叠的存储层。层与层之间通过硅穿孔 (Through-Silicon Vias, TSV)TSV 形成垂直互联。TSV 能够高效传输数据，同时一个存储立方可能包含大量的 TSV 进行垂直互联，因而可以提供极大的内部存储带宽。凭借着新硬件技术，Ahn 等人[17]提出 Tesseract 使用 3D 堆叠技术加速大规模的图处理。

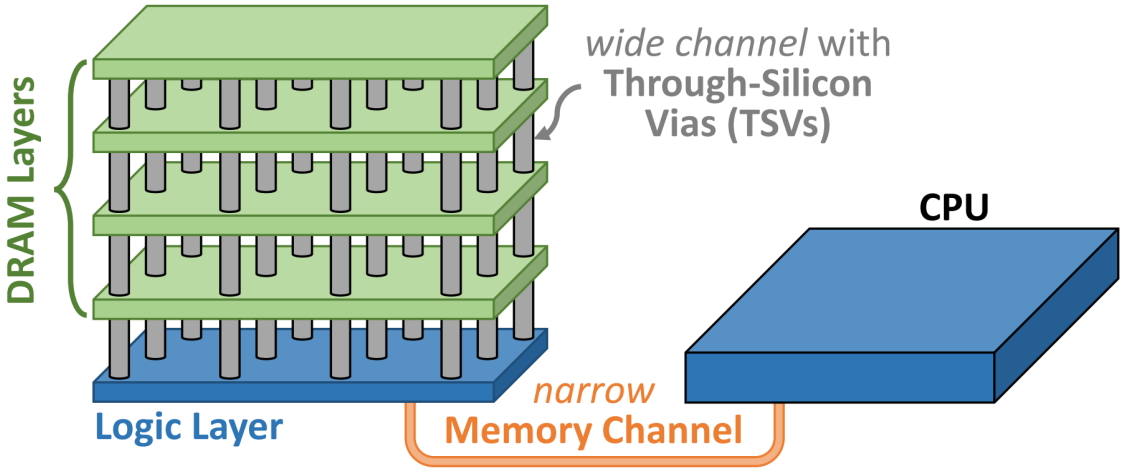



**图 2.5** 3D 堆叠内存结构

与此同时由于人工智能（Artificial Intelligence，AI）的兴起，许多专用于神经网络的 PIM

加速芯片也被设计出来，其中较为著名的就是三星的 HBM-PIM（High Bandwidth Memory，HBM）产品[6]，该产品采用 20nmDRAM 工艺，使用 3D 堆叠技术堆叠封装了 4 层裸芯（Die），每层 die 的 bank 组内增加了专门用于做 16 位浮点数乘加操作的 PIM 单元以处理神经网络中的矩阵操作。此外三星的另一个产品 AxDIMM[7]也采用了近存计算的技术，其将 DRAM 芯片（Chip）和一块 FPGA 处理单元整合到一块有着 DDR4 标准接口的主板上，主要用于加速推荐模型（Recommendation Model）的向量嵌入查找任务（Embedding Lookup）。海力士也提出过存内加速器 AiM[8]用于加速 AI，与三星的 HBM PIM 不同的是，AiM 基于 GDDR6（Graphics Double Data Rate V6）内存，为每个 bank 装配 PIM 单元，通过设计互连总线和全局缓存实现各个 PIM 单元的高效互连。近些年国内的公司阿里巴巴推出过近存计算产品[9]，同样采用的是 3D 堆叠技术将逻辑 die 和数据 die 堆叠封装在一起，通过 TSV 高速传输数据。逻辑 die 上分别设计了用于向量排序和矩阵乘法的计算单元处理不同类型的任务。

### 2.2.2 近存硬件 UPMEM

然而上述工作因为各种复杂的原因难以落地使用和量产，甚至大部分基于模拟器，使得 PIM 技术的推广与使用犹如空中楼阁。近几年，一款号称真正可商用的 PIM 硬件横空出世：UPMEM 作为第一款可以商用的近存计算处理器产品[10]，有着更加通用的处理能力、高速的内存带宽、低廉的接入成本以及完备的开发生态。UPMEM 本身是一条 2400MHz 的有着标准 DDR4 内存接口的内存插块（Dual In-line Memory Module，DIMM），可以像正常的内存条一样插在 Intel CPU 的服务器上。每个双核服务器最多可以插入 20 条 UPMEM（需要为每个 CPU 留出空余的 DIMM 插口插入普通的 DDR4 内存）。如2.6所示，一个 UPMEM 内存条上有 2 个 rank，每个 rank 拥有 8 个 chip，每个 chip 中包含着 8 个 bank，每个 DPU 独占其中一个内存 bank。因此一台服务器能够拥有 2560 个 DPU 并行处理任务。

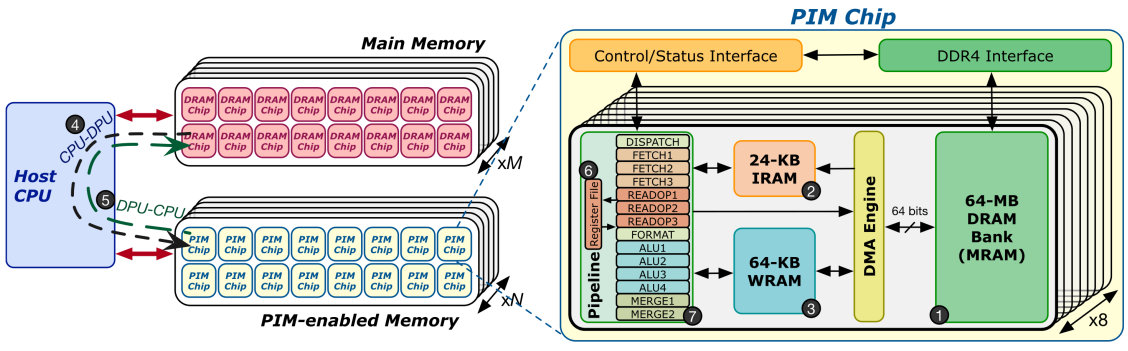



**图 2.6** UPMEM 微架构，图左为 UPMEM 内存和主机 CPU 的交互，图右为 UPMEM 芯片内部组件

每个 DPU 拥有一个标量顺序（Scalar In-order）多线程的 RISC 核心，拥有 24 个物理线程和一个精调的（fine-grained）14 级流水线。其中为了避免实现复杂的数据转发和流水线互锁电路 [10]，同一线程内的两个连续指令必须间隔 11 个周期才能被调度（只有最后三级 ALU4、MERGE1、MERGE2 可以和下一条指令的 DISPATCH、FFTCH 阶段并行执行）。

因此至少同时有 11 个线程同时运行才能最大程度利用流水线。同时每个 DPU 的每个线程都有 24 个可用的 32bit 的寄存器，需要分奇偶访问。

UPMEM 采取哈佛架构，将数据和指令分别存放，拥有 24KB 的指令存储器 IRAM（能存放 4096 条 48 位指令）和 64KB 的数据存储器 WRAM。DPU 在工作时会在取指阶段访问 IRAM，在访存和执行算术运算时访问 WRAM。此外还存在存储层级 MRAM，其就是每个 DPU 独占达的 DRAM Bank，用于存放和 CPU 进行通信的数据，拥有较大的存储空间（64MB）。WRAM 和 MRAM 除了在存储容量上的不同之外，由于 WRAM 本身属于静态 RAM（Static Random Access Memory，SRAM），而 MRAM 就是正常的动态 RAM（Dynamic Random Access Memory，DRAM），二者的访问速度存在着数量级的差别。再加上 DPU 无法直接访问 MRAM 需要通过 WRAM 进行中转，因此常规的访问方式是经过内置的 DMA 引擎将程序用到的大批量的数据从 MRAM 传输到 WRAM 中，再在 WRAM 中去频繁访存和计算。可以将 WRAM 视作通常意义上的 Cache，而 MRAM 则类似于普通内存，不同之处在于 WRAM 对开发人员不透明，需要手动管理。

UPMEM 在硬件的基础上开发了一套完整的软件栈和工具链方便开发人员在其上搭建应用，包括 DPU 程序的运行时库、编译器，以及主机端调用 DPU 程序的 API 库，以及功能强大的调试器 dpu-lldb。UPMEM 的编程模式为单程序多数据（Single Program Multiple Data，SPMD）模型，DPU 及其独占的 bank 相对于 CPU 来说类似于一个协处理器，CPU 端被称为 host 端，而 DPU 端被称为 device 端。由 CPU 主动地将编译好的 DPU 程序装载入 DPU 并准备数据传输到 DPU，再启动 DPU 执行程序，执行结束后收集结果或执行下一轮的任务。CPU 和 DPU 之间的通信需要使用 UPMEM 提供的主机端 API 库，而 DPU 和 DPU 之间无法直接通信需要到 CPU 搬移到内存中进行中转。

有许多工作对 UPMEM 的硬件特征做了系统且全面的测试[18, 21, 20, 19, 63]，其中，Gómez-Luna 等人[18]的测试工作较为全面且权威。在这些评测中，不难发现 UPMEM 的硬件优势主要有下面几点：

1) 高存储容量和传输带宽，忽略 WRAM 等高速缓存，MRAM 本身有 64MB，2560 个 DPU 共有 160GB 内存，远超市面上常用显卡的显存，每个 MRAM 的传输带宽大约为 700MB/s，当 2560 个 DPU 并行工作其聚合带宽能够达到 1.7TB/s，WRAM 的聚合带宽更是能够达到 6.8PB/s。

2) 高并发和细粒度并发控制，如上所述，2560 个 DPU 可以同时工作，每个 DPU 内部还可以控制 24 个线程进行更加细粒度的控制，虽然大多数测试中在跑常规测试集（Benchmark）时，超过 11 个线程并发就会让核心达到饱和，按照这个方式计算，整个 PIM 系统的并发线程数量能够达到近 3w。

3) 高能效比，PIM 设备由于减少了数据的多层级搬移可以大大降低能耗，文章[18]中测试，UPMEM 的能效比高于运行同等任务且优化成熟的 CPU 和 GPU。

虽然 UPMEM 有上述的许多优点，作为商用近存计算硬件能够充分支持并行计算，但同时 UPMEM 的局限性也十分明显：

1) UPMEM 由于硬件资源十分有限，只支持 32bit 的整数加减法和 8bit 的乘法，其他的算术操作包括 64bit 的相关操作，乘除操作，浮点操作都是使用软件实现，效率较为低下；同时 UPMEM 本身主频不高，DPU 处理器的规模受限，即使是 32bit 加法的 MRAM 计算访存比也只有 1:4[18]，这些充分说明 UPMEM 的计算能力弱，更加适合内存瓶颈的任务。

2) UPMEM 的通信效率低，host 和 device 的传输效率本身不高，只有大批量传输连续数据时才能勉强达到 DDR4 内存的传输带宽，同时 DPU 直接彼此独立缺乏有效的通信手段，只能通过 CPU 主动中转数据，效率更加低下。因此 UPMEM 不适合那些需要多核频繁通信的任务。[18]。

### 2.2.3 UPMEM 相关应用

自 UPMEM 可商用以来，有许多研究者就此硬件做出了许多工作。其中较为基础的一类是针对 UPMEM 硬件特征构建软件栈或对硬件做修改。如 Khan 等人[64]针对包括 UPMEM 在内的诸多近存计算硬件构建了编译器，以支持开发人员在更高的抽象层级编程。同样的，Chen 等人[65]也针对 UPMEM 硬件抽象更为高级的软件栈，包括对 DPU 的元数据管理、主从通信以及批处理模式三个大主要模块。同时，还有部分工作专注于拓展 UPMEM 基础软件库。Giannoula 等人[66]开发了基于 UPMEM 的稀疏矩阵向量乘法（Sparse Matrix Vector Multiplication，SpMV）库，支持多种稀疏格式的矩阵以及数据格式，设计了多种数据映射和优化方法以适应不同的场景。Item 等人[67]在 UPMEM 上开发了一套基于查找表（Look UP Table，LUT）和 CORDIC (coordinate rotation digital computer) 迭代的超越函数库，以一定的精度范围内支持了包括三角函数、指数对数、双曲线、平方根等复杂计算。Noh 等人[68]针对 UPMEM 的 DPU 之间通信慢的问题，开发了一套支持多种通信模式的高效通信框架。除此之外，有部分工作在了解了 UPMEM 硬件的优缺点后，试图对 UPMEM 的硬件本身进行修改。北大的 Zhou 等人[69]为解决 UPMEM 的通信慢的问题，增设外部数据连接电路联通物理相邻的 DIMM，并设计数据转发和传输算法提高数据传输效率。

另外一大类重要的工作专注于利用现有 UPMEM 硬件去加速不同领域的应用，主要涉及到生物基因、数据库以及人工智能领域。生物基因领域主要是使用 UPMEM 加速基因测序和基因比对工作[30, 31, 32, 33, 34]，这类工作的本质是字符串匹配为内存瓶颈任务，因此适合使用 UPMEM 硬件进行加速。数据库领域有许多工作对 UPMEM 关注密切。比如早期的工作[22]将 skyline 算子卸载到 UPMEM。清华的 Kang 等人做了一系列的工作[23, 24, 25]将数据库常见的索引如跳表、前缀树的查询卸载到了 UPMEM 上，并充分设计了负载均衡算法保证查询速度。一些工作[26, 27]将数据库最基本的查询算子，全部或部分卸载到了 UPMEM 上。Baumstark 等人[28]将查询计划的优化也卸载到了 UPMEM 上。Lim 等

人[29]将数据库中的连接查询（Join）卸载到 UPMEM 上，通过巧妙地移位和排序解决了 UPMEM 因内存交错（Bank Interleave）带来的数据传输性能损失。

UPMEM 的高并行和细粒度控制特性使得其非常适合用于 AI 和神经网络的场景。有相当一部分工作使用 UPMEM 加速神经网络的推理。Zarif 等人[35]将 UPMEM 用于嵌入查找（Embedding Lookup）任务的卸载，对于目前较大的嵌入表（Embedding Table）加速效果尤为明显。Gómez-Luna 等人[36]以简单直接的方式卸载了传统机器学习中的基础模型到 UPMEM 上，包括线性回归、逻辑回归、决策树、K 均值聚类，并做了全面丰富的测试，但是测试结果无一表明这些模型的推理都遭受了严重的计算性能瓶颈。Das 等人[37]在 UPMEM 上分别卸载了嵌入二值神经网络（Embedded Binary Neural Network，eBNN）和 YOLOv3（主要是卸载卷积操作），其主要思想是将卷积神经网络（Convolutional Neural Network，CNN）的权重量化到低 bit 位，再通过查找表查询低 bit 浮点数乘积，以消除浮点乘法运算，但这会严重降低模型的精度。Giannoula 等人[38]将图神经网络（Graph Neural Network，GNN）的推理卸载到了 UPMEM 上，测试结果表明对于稀疏图和较为内存瓶颈的场景中，UPMEM 的推理效率提升非常大。最与本课题应用场景相近的工作 PIM-DL[39]使用 UPMEM 推理 Bert，其通过将矩阵乘法转换为最近邻查找和向量加法，减少了对乘法的需求，从而提高了计算效率。但其最近邻查找是在 CPU 上完成的，而 UPMEM 只执行向量加法操作，并没将计算重担完全卸载到 UPMEM 上。最近也开始有研究者尝试使用 UPMEM 加速神经网络的训练过程。Gogineni 等人[40]提出 SwiftRL 以解决强化学习（Reinforce Learning，RL）中的内存瓶颈问题，将 Tabular Q-learning 和 SARSA（State-Action-Reward-State-Action）等强化学习算法在 UPMEM 上实现，并适应不同应用场景。Rhyner 等人[41]希望在 UPMEM 实现分布式随机梯度下降算法（Stochastic Gradient Descent，SGD）探究 UPMEM 的 AI 训练能力和硬件特点。上述两个训练工作的结果都表明只有在内存瓶颈的应用场景下，UPMEM 的训练性能提升较大，而且由于训练过程中存在跨节点通信，扩展性较差无法随数据规模线性扩展。

## 2.3 本章小结

本章详细地对大模型推理加速的相关技术和论文做了详尽的介绍，从大模型的发展历程和模型结构的分析，了解大模型的推理成本很高，明白了加速推理的关键因素在于解决内存瓶颈或者提升基本算子的性能。随后介绍了模型量化的基本原理，以及用于加速大模型以缓解内存瓶颈的相关工作。然后对大模型推理阶段的基本算子矩阵向量乘做了基本的介绍，并阐述了相关的优化方法。

随后主要介绍了近存计算研究现状。首先梳理的近存计算的历史，包括近存计算的产生原因、发展和现状，一并介绍了其中具有代表性的工作。然后着重介绍了号称第一款可商用的近存计算的硬件 UPMEM，包括其硬件微架构和软件栈，同时结合相关测评工作说明了改硬件的基本特性。最后详细地介绍了基于 UPMEM 硬件上做的相关科研工作，涉及生物基因、数据库和机器学习等诸多领域，对研究现状有了充分的认识。

# 3 基于模型量化和查找表的矩阵向量乘软件优化

## 3.1 量化基本说明

基于在第二章介绍的 GEMV 的通用两种方法，直接使用 float32 数据格式进行 GEMV 的计算的性能非常差，原因是 UPMEM 硬件不支持浮点数的算术运算而使用软件模拟，其性能大约是 int32 数据类型格式的算术吞吐的十分之一[18]，因此需要将权重矩阵和激活向量量化到 8bit 进行计算。在上一小节提到，浮点量化格式往往优于定点数量化[56]；同时测试表明 UPMEM 的读写速度相较于其计算核心更快，计算访存比仅有 1:4[18]，因此非常适合用访存换计算。我们直接使用基于 FP8 数据格式的查找表算法，会比直接使用 int8 的矩阵向量乘法内核的“性价比”更高（模型精度/算子性能）。为提高量化精度我们可以使用机器学习的方式以少量的校准数据集调整量化后参数，针对反向传播中 LUT 不可导的问题，可以使用直通估计器（Straight-Through Estimator，STE）解决[70]。STE 简单来说就是跳过 LUT 自身的梯度下降，将误差直通传递到上一层，如通过调整上一层的矩阵权重来实现误差最小化，这是一种非均匀量化的手段。

本章所讨论的权重矩阵和激活向量的维度如果没有说明都和 Llama2-7B 保持一致为 4096，权重矩阵的维度为 $4096 \times 4096$，分到 32 个头后，单个 DPU 负责一个头的自注意力计算（算子融合），矩阵的维度为 $4096 \times 128$，本章所有讨论的优化都是建立在单个 DPU 的推理优化。因此单个 DPU 的 GEMV 内核的正常输入为 4096 维度 FP8 数据格式的激活向量，4096 维度 FP8 数据格式的权重方阵，最终计算得到的结果也应该是 4096 维度的结果向量。

UPMEM 的 MRAM 访问通过 DMA 引擎速度较慢，并且需要大批量数据传输，应该尽量减少 MRAM 的访问。UPMEM 访问 WRAM 的速度较快，当流水线充满时，且访问带宽不受访问模式影响（顺序、随机），且任何 8byte 以下的数据访问都只会消耗一个时钟周期。这里为后续算法做时间复杂度分析，建模 WRAM 的一次读或写消耗为 $read$。

## 3.2 基于查找表分块的矩阵向量乘算法

### 3.2.1 矩阵向量乘查找表基础

查找表（LUT）是非常典型的存储换计算的技巧，常常被用在某些边端设备或者计算能力有限的硬件上以支持复杂计算。由于 UPMEM 硬件本身较弱的计算能力，以及较

低的计算访存比，非常适合使用访存换计算的方法。一般意义上的查找表就是对于函数 $f(x_1, x_2, \cdots, x_n) = y$，在一定的定义域范围和精度内穷举所有自变量的组合，并提前计算得到每种组合对应的函数值，制作成一张映射表，此后的函数计算无需计算而只需查表即可。由于计算机只能离散有限地表示数值，每种特定位宽的数是天然可穷举的，例如对于 $nbit$ 的数作为自变量，其本身有 $2^n$ 种不同的数值，两个 $nbit$ 的数作为自变量，则有 $2^{2n}$ 个不同的组合。

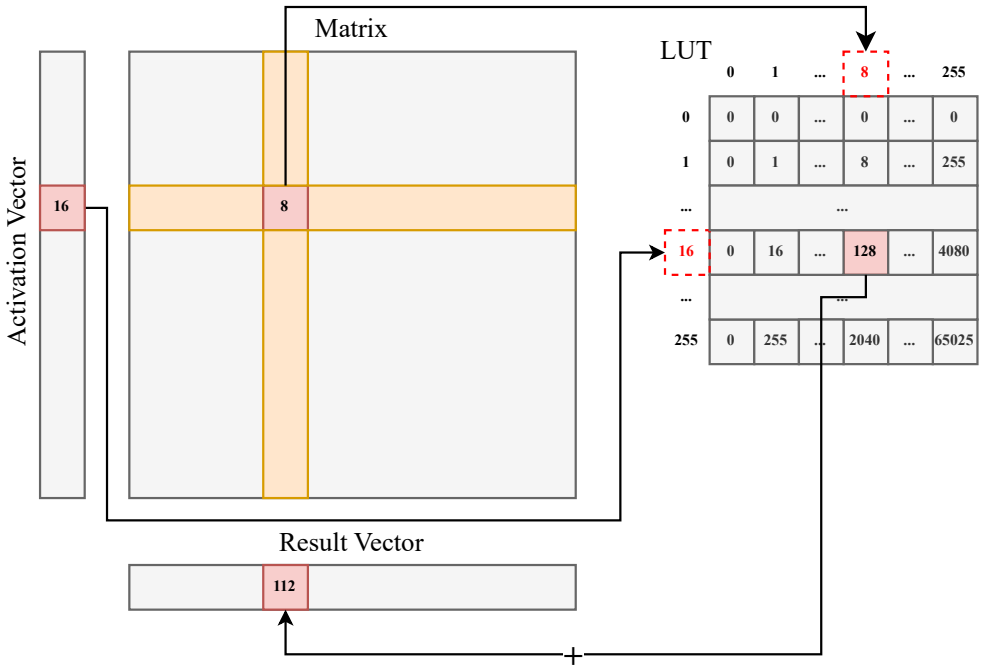



**图 3.1** 使用查找表卸载 GEMV 中的乘法示意图

这里举例 GEMV 的 8bit 乘法查找表的设计，如图3.1所示，其中有一张数据类型为 uint8 的乘法查找表（为了直观采用 uint8 数据格式）。由于两个操作数都为 8bit，因此输入一共有 $256 \times 256 = 2^{16}$ 种组合，因此可以提前构建一个 256 行 256 列二维数组存储查找表，第一个操作数的数值代表行索引，第二个操作数的数值代表列索引，行索引和列索引交叉索引到的元素值为两个操作数的乘积。例如在进行 GEMV 运算时，要计算激活向量中某个值为 16 的元素和权重矩阵中某个值为 8 的元素的乘积，现只需要访问提前构建好的二维数组（查找表）的第 16 行第 8 列的元素即可，得到答案为 128，再加到结果向量的对应位置上。当然这里加法计算同样可以通过构建一张 8bit 的加法查找表消除。这样，8 位宽下任意数据格式的任意二元算术运算都可以按照上述的查表方式将计算转换为访存。同时上述访存过程仅仅涉及最简单的数组索引计算，几乎能够得到所有硬件的支持，从而降低了对硬件算术能力的要求。

上述乘法查找表的构建存在冗余，因为乘法满足交换律，因此上述 $256 \times 256$ 的矩阵

只需要保存上三角或下三角即可。甚至能够进一步将 8bit 的乘法拆成 4bit 的乘法的加法，但这两种做法无疑都会增加计算复杂度，对于在 UPMEM 上需要频繁访问的查找表而言是不合适的。同时查找表能够按照行划分成为子表，例如上述 uint8 的乘法查找表，当能够确定某个操作数的变化范围在 [0,64) 时，只需要上述查找表的前 64 行即可满足计算，即每一行和多行的组合都是属于 8bit 乘法查找表的子查找表，当空间受限时，可以按需载入子查找表。在此我们特别约定，1）查找表的大小和构建方式都如上述所描述，n bit 的二元运算查找表表项为 $2^{2n}$；2）二元运算查找表通过两个索引（分别是行列索引）确定查找值，约定这里的索引在后文称作索引值（row/col index），查表得到的值为元素值（element）；3）后文如果没有特别说明，子查找表指的是原查找表二维数组的不同行的组合，即确定行索引值的范围的子表。

### 3.2.2 分块载入查找表卸载乘法

直接使用 8bit 乘法查找表的方法在 UPMEM 中卸载 GEMV 计算存在诸多问题，首当其冲的就是 WRAM 的空间限制：由于查找表在执行 GEMV 运算时需要频繁访问，因此必须要将其载入 WRAM 这种高速内存中才会有较好的性能。然而 8bit 的查找表光是表项就有 $256x256 = 2^{16} = 64K$ 项，如果按照表中每个元素刚好占用 1 字节，则查找表需要占用空间 64KB，而 WRAM 的容量只有 64KB，除了用户数据之外至少需要为各个线程的堆栈留存一部分空间，因此无法将整个查找表载入 WRAM。

解决办法就是分块载入查找表，每次只执行数据范围落在当前载入的子查找表覆盖的范围内的计算。但是此时需要注意的一点就是在于数据的重用性：权重矩阵无法一次性全部载入 WRAM，从 MRAM 载入 WRAM 的带宽远低于 WRAM 与核心交互的带宽，分多次载入子查找表后需要避免因为计算不完整而重复载入矩阵。基于上述考虑，我们设计了基于分块载入查找表卸载 GEMV 中的乘法的算法，具体如图3.2所示，激活向量和结果向量完整载入 WRAM 中，将查找表拆分成 16 个子查找表，行索引范围被划分为在 $[0-15), [16-31), \cdots, [240, 256)$16 个区间。该算法需要进行子查找表次数个迭代，每次迭代流程大致为：将子查找表载入 WRAM 中，遍历激活向量，对于每个元素判断其值是否在当前子查找表的索引值区间内，对于满足要求的元素，将该元素所对应的矩阵的行向量从 MRAM 载入 WRAM，执行查表乘法计算并加到结果向量上；对于不满足要求的元素直接跳过，等待下一次迭代再进行判断。如此一来完全没有将数据重复地从 MRAM 搬移到 WRAM 中，无论是查找表还是权重矩阵，都只从 MRAM 中搬移到 WRAM 中一次，实现了非常的 WRAM 数据局部性。

### 3.2.3 基于数制映射表卸载加法

想要完整卸载 GEMV 算子的所有计算仅仅使用上述分块查找表卸载乘法远远不够，上述方法虽然使用分块载入的方式解决空间上的限制，但是在计算加法时如果想要使用类似的思路，由于使用的数制是 FP8，硬件加法无法支持，同时由于累加的操作特性，FP8 加法的

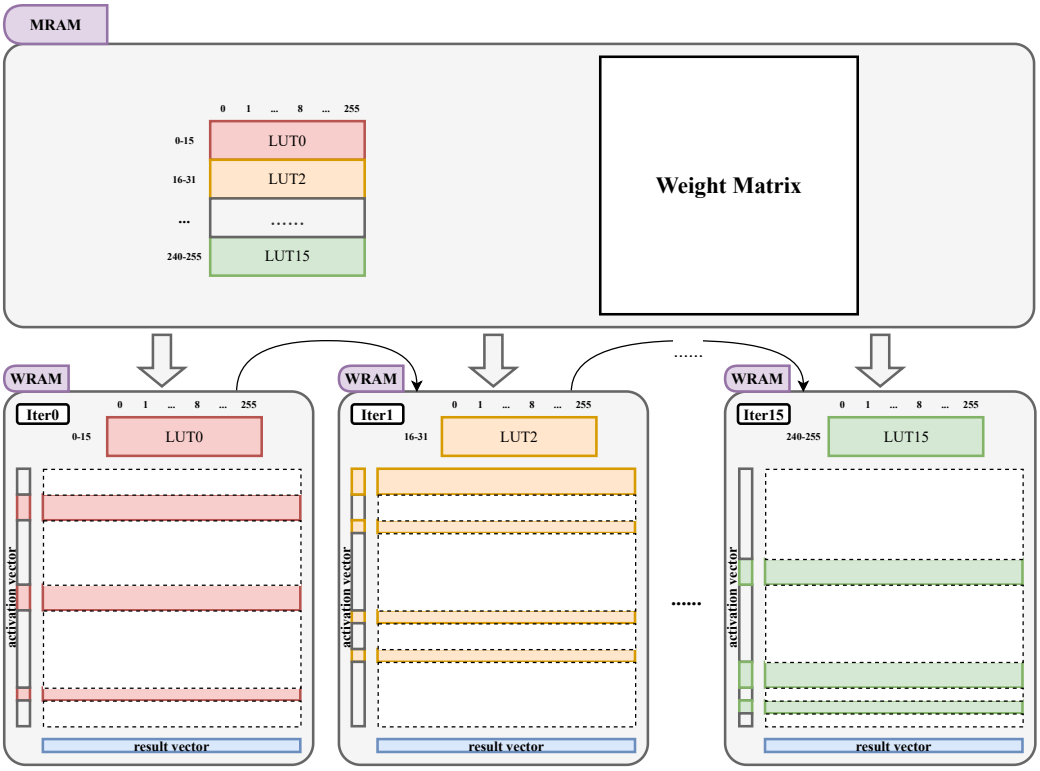



**图 3.2** 分块载入查找表卸载 GEMV 的算子乘法示意图，图上面表示 MRAM 中存放的数据，图下面表示 WRAM 和迭代计算

所在查找表区间是动态变化的，无法事先静态地确定并载入子查找表，直接从 MRAM 中读取性能就会非常差。一种方式是直接使用软件模拟：FP8 常用的有两种格式[71]，如表3.1所示，我们选用 E4M3 格式的 FP8 进行推理，因其相较于 E5M2 具有更高的精度（E5M2 相较于 E4M3 拥有更高的动态范围更适合训练）。其数据格式并不完全符合 IEEE754，取消了无穷的表示缩减了 NaN 的表示范围以容纳更多规格化数，其他部分均符合 IEEE754 的标准。

**表 3.1** FP8 常用两种格式二进制细节

| | E4M3 | E5M2 |
|---|---|---|
| Exponent bias | 7 | 15 |
| Infinities | N/A | $S.11111.00_2$ |
| NaN | $S.11111.11_2$ | $S.11111.\{01, 10, 11\}_2$ |
| Zeros | $S.0000.00_2$ | $S.00000.00_2$ |
| Max normal | $S.1111.10_2 = 1.75 * 2^8 = 448$ | $S.11110.11_2 = 1.75 * 2^{15} = 57,344$ |
| Min normal | $S.0001.00_2 = 2^{-6}$ | $S.000001.00_2 = 2^{-14}$ |
| Max subnormal | $S.0000.11_2 = 0.875 * 2^{-6}$ | $S.000000.11_2 = 0.75 * 2^{-14}$ |
| Min subnormal | $S.0000.01_2 = 2^{-9}$ | $S.000000.01_2 = 2^{-16}$ |

如果直接使用软件模拟，计算两个浮点数的流程的大致为：对阶、尾数求和、规格化、舍入、溢出处理，虽然不用处理无穷等特殊情况，但是上述几个步骤中涉及到大量的移位操作和逻辑操作，由于 UPMEM 的一个周期至多只能执行一条指令[10]，因此这些位运算和逻辑运算指令开销不能忽视，而且由于规格和非规格数的区别存在大量的条件判断和跳转语句，同样非常影响性能。我们注意到，可以将 E4M3 格式的 FP8 展开成 int32，使用 int32 进行加法计算和中间结果的保存（硬件不支持 int32 乘法但是支持加法），在得到最终的结果向量后再转回 FP 8 数制以便后续的传输。具体的展开形式可以很简单，E4M3 的符号位不变置于 int32 的符号位，同时根据指数位置判断该数是否为规格化数，若是规格化数，则需要将尾数的低三位前面添 1 形成 4 位尾数；若不是规格化数，则正常取尾数。然后假设指数部分的值为 e，将尾数左移 $max(e-1□0)$ 位即可。如图3.3所示，FP8 数 0 0010 110 转为 int32 为 0...11100，FP8 数 0 0000 110 转为 int32 为 0...110，二者相加得 0...100010，这个时候反转回 FP8 应该首先判断是否为规格数，显然该数为规格化数，需要将 int32 数右移直至前导第一个 1 到最低第 4 位上（计算舍入）即可。

但其实同样可以使用一张数制映射表代替上述复杂的逻辑操作，如图3.3中的 FP8_TO_INT32_LUT 是一个表项为 256 的一元映射查找表，提前将 FP8 展开存储在该张映射表中，在计算时展开操作就可以换为查表操作。更进一步，原本的乘法查找表元素为单字节 FP8，现在将其替换为其对 int32 的映射项，变为四字节。这样查询出来的乘积就是 int32 的展开格式，直接加到同样每个元素扩展为 32 位的结果向量中。最后，基于查找表 FP8_TO_INT32_LUT 再通过二分查找将结果向量中的每个元素还原成 FP8 以便后续的传输。

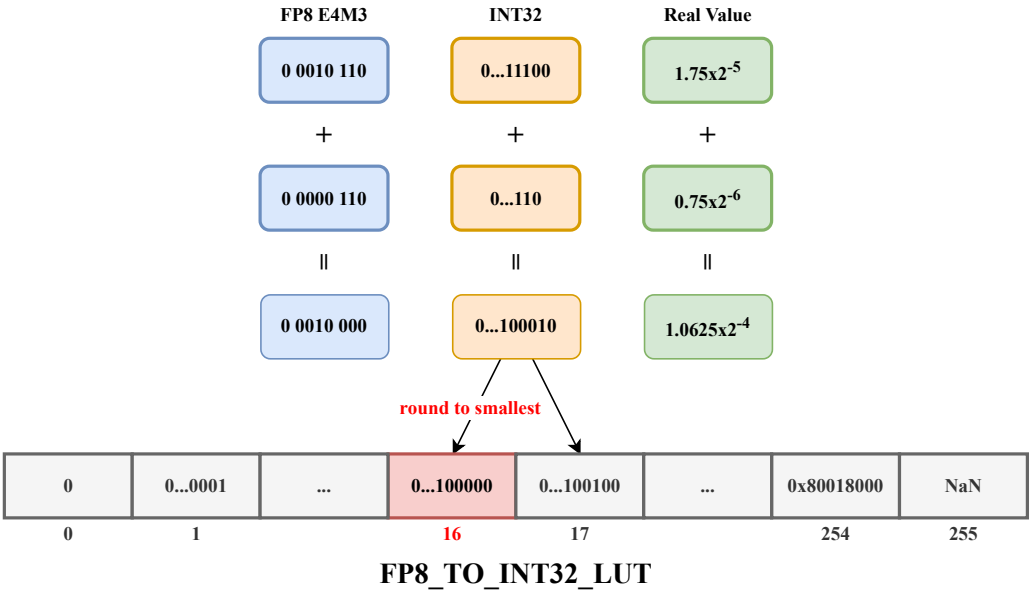



**图 3.3** 分块载入查找表卸载 GEMV 的算子乘法示意图，图上面表示 MRAM 中存放的数据，图下面表示 WRAM 和迭代计算

在查找的过程中，可能会遇到舍入问题，即计算出来的元素值无法精准匹配查找表中的元素，其原因是使用 int32 进行加法运算时保留了相当的精度没有舍入，这个时候会出现待查找的值处于查找表两个相邻元素值之间，如图3.3所示，为了简化计算加速查找我们直接选择索引值较小的那个数作为查找结果：一方面转为 int32 进行中间结果的加法运算相对于直接使用 FP8 计算保留了相当大精度，这里只是将最终结果进行转换因此精度损失整体来讲不大；另一方面精度损失仍然可以通过离线的基于机器学习的量化工作优化消除。这种舍入方式我们这里称之为向最小索引值舍入（Round to Smallest Index），这种舍入其实本质和向零舍入（Round toward Zero）是同一种舍入规则。

### 3.2.4 算法小结

至此，可以完全卸载 GEMV 算子的全部运算到 UPMEM 上，我们在这里对上述方法进行总结，给出算法如3.1所示，假设 M 和 N 都为 4096，那么激活向量大小为 4KB，8bit 和 32bit 的结果向量的大小分别为 4KB 和 16KB，将查找表分成 16 份，每个子查找表的大小为 16KB，载入的矩阵一行的大小为 4KB，FP8 到 int32 的映射表大小为 1KB，总共占用 WRAM 空间 45KB，剩余 19KB 给各个线程分配堆栈空间完全足够用。在这种设计下，权重矩阵的所有行总共只需要从 MRAM 中载入 WRAM 中一次，LUT 的每个子表同样也只载入一次，FP8 和 int32 的相互映射也是直接在 WRAM 中完成，充分提高了数据的重用性。

**算法 3.1** 基于查找表分块的矩阵向量乘算法-LUTBlock

**输入：** $Vector[M], Matrix[M][N], LUT[256][256], FP8\_to\_INT32[256]$;

**输出：** $Result\_8[N]$;

**define** $SubLUT[16][256]$
**define** $MatRow[N]$
**for** $i \leftarrow 0$ **to** $15$ **do**
    **mram_read**$(SubLUT, LUT[16i \cdots 16i+15])$ ▷ parallel in 16 for each tasklet
    **for** $j \leftarrow 0$ **to** $M-1$ **do**
        **if** $Vector[j]$ **not in** $[16i, 16i+15]$ **then**
            **continue**
        **end if**
        **mram_read**$(MatRow, Matrix[j])$ ▷ parallel in N for each tasklet
        **for** $k \leftarrow 0$ **to** $N-1$ **do** ▷ parallel in N for each tasklet
            $Result[k] \leftarrow Result[k] + SubLUT[Vector[j]\&0xF][MatRow[k]]$
        **end for**
    **end for**
**end for**
**define** $Result\_32[N]$
$Result\_8 \leftarrow$ **BinarySearch**$(Result\_32, FP8\_to\_INT32)$ ▷ parallel in N for each tasklet
**return** $Result\_8$

## 3.3 针对不同尺寸矩阵的矩阵向量乘优化

在上一小节主要针对 MRAM 的读写做了优化，提高了 WRAM 的数据局部性，这一小节主要针对 WRAM 做优化，提高寄存器的数据重用。

我们知道在 Llama2-7B 的 MHSA 中，会将一个完整的 4096 长度的词向量通过线形层成映射到 32 个子空间，对应到 32 个头（会将 $4096 \times 1$ 的向量映射成 32 个 $128 \times 1$ 的向量），本质上属于降维操作，（单个头）线形层是一个 $4096 \times 128$ 的窄矩阵；在最终计算得到 attention 向量后，又会通过一个线性层将 $128 \times 1$ 的向量重新映射为 $4096 \times 1$ 并最终合并各个头的结果，属于升维操作，此时线形层是一个 $128 \times 4096$ 的宽矩阵。一般意义上，对于一个 $M \times N$ 的矩阵，当 $M \gg N$ 时，认为这个矩阵是所谓的窄矩阵；当 $M \ll N$ 时，认为这个矩阵是所谓的窄矩阵，但这里我们认为 $4096 \times 128$ 或 $4096 \times 256$ 就属于窄矩阵，$128 \times 4096$ 或 $256 \times 4096$ 就属于宽矩阵（并且在后文中使用 128 维度作为示例）。这两种形状的矩阵在 LLM 的推理过程中十分常见（MLP 部分的矩阵可以随意切），可以针对这两种特殊的矩阵进行优化。

### 3.3.1 窄矩阵行重排

对于窄矩阵（4096 × 128），其特点在于行数相较于列数非常多，频繁地读写结果向量有很大的开销，我们这里基于此前提到的算法做出修改如图所示，此前由于考虑权重矩阵一行的向量会较长我们无法同时载入多行，在此场景下列数较少，可以一次性载入多行，形成子矩阵。在 WRAM 中设置一次性载入的子矩阵大小上限为 16KB，对于 4096 × 128 的权重矩阵，可以一次性载入落在子查找表区间的 128 行权重矩阵。注意这里我们不改变矩阵的行列主存格式，依然按照行主存，原因是子矩阵的行之间不一定是连续的，改为列主存的话无法充分利用 DMA 引擎的大批量连续数据传输带宽高的特性，好在 WRAM 的访存模式对访存带宽并无影响[18]，这里我们仍然按照列进行访存和计算，将子矩阵的一列和对应的激活向量进行向量内积操作，结果加到对应的结果向量中。实际上，矩阵行重排的优化技巧对矩阵的形状没有特别严格的限制，对于宽矩阵仍然可以使用行重排，但是此时在保证同时载入的行数的情况下，就无法读取每一行的全部列，可以针对这种情况进行合理分块，如图3.3.2所示，设置 BR 和 BC 分别代表分块的行（Row）长度和列（Col）长度，从 MRAM 中将一个矩阵块 Block 以及子查找表读入 WRAM 中按照 GEMV 内积的方式计算结果，理论上加速比和窄矩阵是一致的（保证一次载入的行越多越好），给出对应的算法3.2。

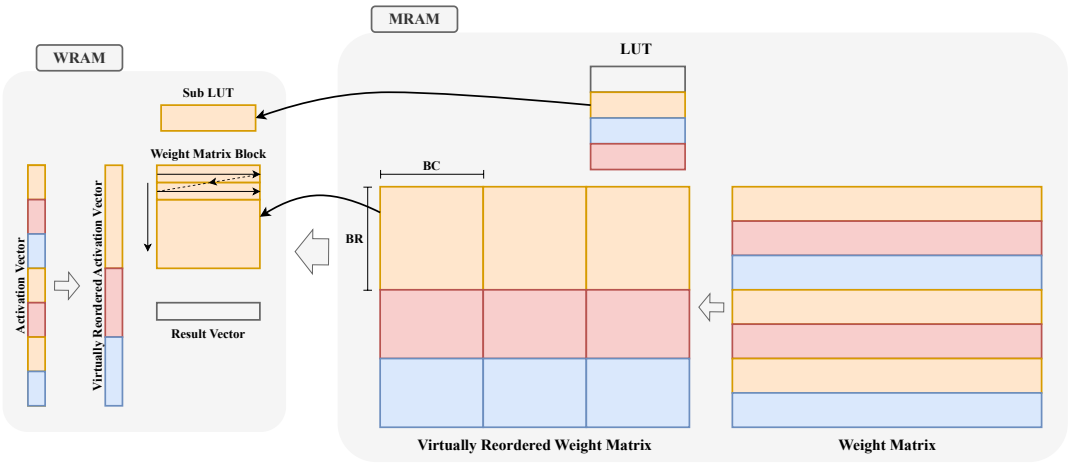



**图 3.4** 矩阵行重排优化计算示意图

此时没有任何新增的数据结构，WRAM 的空间一定是放得下的，现在来计算 WRAM 的读写次数。在使用行重排之前，每次载入子查找表，每个 tasklets 需要完整地读一遍激活向量，4096 次结果向量的读取和写入，得到 WRAM 的激活向量和结果向量的总次数为 4096+128*4096/16+4096*2/16=36.5K read。而使用了行重排之后，假设刚好每次载入子查找表时，激活向量值域落在子查找表区间的行数都为 256，那么每个 tasklets 只需要读写 2 次结果向量，但是对应子矩阵的激活向量需要重新读 128 次，则相应的总次数为 4096+256+256x128/16+2=6K，有将近 6 倍的提升。

**算法 3.2** 窄权重矩阵行重排的矩阵向量乘算法-LUTRow

**输入：** $Vector[M], Matrix[M][N], LUT[256][256], FP8\_to\_INT32[256]$;

**输出：** $Result\_8[N]$;

**for** $i \leftarrow 0$ **to** 15 **do**
  **define** $SubLUT[16][256]$
  **mram_read**$(SubLUT, LUT[16i \cdots 16i+15])$ ▷ parallel in 16 for each tasklet
  **for** $j \leftarrow 0$ **to** $M-1$ **do**
    **if** $Vector[j]$ **not in** $[16i, 16i+15]$ **then**
      **continue**
    **end if**
    **define** $MatRow[N]$
    **mram_read**$(MatRow, Matrix[j])$ ▷ parallel in N for each tasklet
    **for** $k \leftarrow 0$ **to** $N-1$ **do** ▷ parallel in N for each tasklet
      $Result[k] \leftarrow Result[k] + SubLUT[Vector[j]\&0xF][MatRow[k]]$
    **end for**
  **end for**
**end for**
**define** $Result\_32[N]$
$Result\_8 \leftarrow$ **BinarySearch**$(Result\_32, FP8\_to\_INT32)$ ▷ parallel in N for each tasklet
**return** $Result\_8$

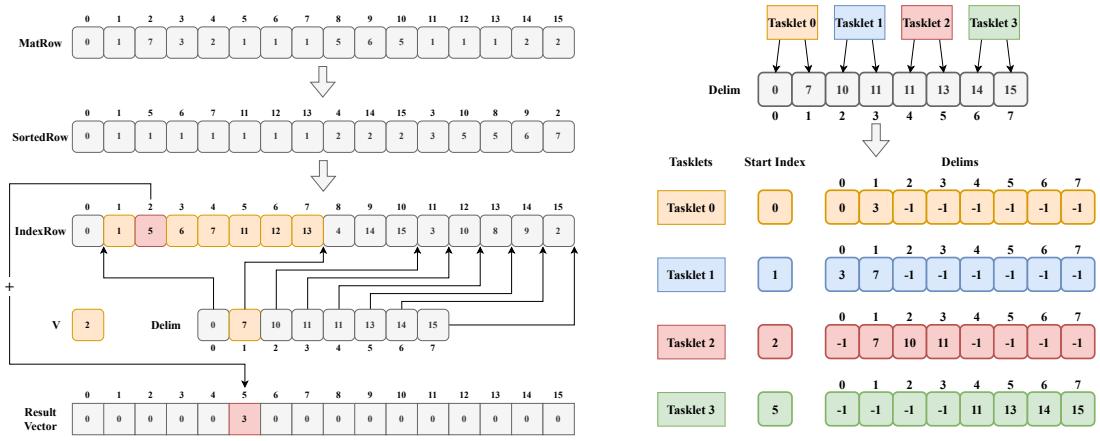
### 3.3.2 宽矩阵列重排

对于宽矩阵（$128 \times 4096$），我们可以观察到，当激活向量中的某个元素和权重矩阵的一行做乘积时，需要频繁地查找子查找表以获取乘积，然而由于矩阵权重为 8bit，总共只有 256 种值，对应的乘积也只有 256 种值，因此理论上最多只需要查询 256 次 LUT，但是现实的计算情况是矩阵一行中的每一个元素都重新去查询 LUT，一共查询了 4096 次。如果将矩阵的一行按照数值排序，相同的连续数值只需要查表一次，这样就能避免重复查询 LUT，提高访存效率。

按照此思想，我们提出列重排如图3.5(a)所示，为方便展示工作原理，这里以 3bit 数据位宽和一个 16 个元素的向量为例，对于权重矩阵的每一行，首先我们保留其索引值，按照元素值进行排序，排序后我们构建一个 8 个元素的桶（数组）对应 3bit 数据的所有取值情况，对权重矩阵每一行 0-7 数值出现的次数进行计数，计数完成后，对于着 8 个桶，我们称之为 Delim 数组，$Delim[i]$ 代表的含义就是值为 i 的元素出现的次数；接下来设置状态转移方程为3.1。

$$\text{delim}[i] = \begin{cases} \text{delim}[i] - 1, & i = 0 \\ \text{delim}[i] + \text{delim}[i-1], & i > 0 \end{cases} \tag{3.1}$$

这样更新之后，Delim 才被真正称之为分界数组，然后再将排序后的索引构成的 Index 数组替换为权重矩阵对应的行向量即可。这个时候 delim 数组的值就是指示着矩阵元素值发生变化的分界线。使用这样两个数据结构进行 GEMV 运算，如果确定了激活向量的元素的值 V 为 2 后，相当于确定了要查找的子查找表（哪一行），然后读取 Delim 数组可以知道，索引 $(0,7]$ 的元素值都是 1，因此可以通过查找 LUT 确定这个 7 个数的乘积都为 1（uint8 查找表），再读取载入的 Index 数组确定累加到结果向量的索引完成计算。




(a) 列重排 Index 和 Delim 数组构建　　(b) 列重排并行计算优化

**图 3.5** 矩阵列重排优化计算示意图

但是由于数据的分布情况未知，简单按照 Delim 长度平均分配给个线程的任务可能出现严重的负载不均，无法充分利用 DPU 的并行计算能力。可以看到图3.5(b)中，这 16 个元素的向量交由 4 个 tasklet 进行数据划分执行 GEMV，如果按照简单地 Delim 数组长度平均分，即 tasklet0 负责 Delim 数组中索引 0 和 1 的计算，tasklet1 负责 2 和 3 等等，那么就出现负载不均衡：tasklet0 负责了 8 个元素的计算，几乎占据了整个向量计算量的一半，而其他 tasklet 的计算量非常低，甚至有些只有两个元素的计算量，这些 tasklet 计算完成后就会被闲置等待 tasklet0 完成工作。回到实际情况，试想一种极端情况：权重矩阵的某一行 4096 个元素，值全部都是 1，这样所有的计算任务都会交由 tasklet0 执行，其他线程需要等待 tasklet0 计算完成才能进行下一行的计算，性能会大大降低。因此为了负载均匀，需要重新设计 Delim 数组的形式：为每个 tasklet 配备一个 Delim 数组，然后将任务均分到每个 tasklet 独享的 Delim 数组中。如图3.5(b)所示，将前 4 个元素的计算分给 tasklet0，则 tasklet0 需要执行一个值为 0 的计算和三个值为 1 的计算，那么只需要在 Delim 数组的 0 号索引填 0，一号索引填 3 即可，在二号索引填-1 表示任务结束不用继续读取 Delim 数组；tasklet1 负责 5-8 号元素的计算，需要执行四个值为 1 的计算，只需要在一号索引填 7，Start Index 处填 1 表示任务起始的计算值，这样 tasklet1 就会从 1 号索引的前一号索引读取起始位置为 3+1=4，然后读取 4、5、6、7 号元素执行计算。其他的 tasklet 类似。如此以来就能避免负载不均衡的问题。给出对应的算法表示3.3。

**算法 3.3** 宽权重矩阵列重排的矩阵向量乘算法-LUTCol

**输入：** $Vector[M], DelimMat[M][16][257], IndexMat[M][N], LUT[256][256], FP8\_to\_INT32[256]$;

**输出：** $Result\_8[N]$;

**define** $SubLUT[16][256]$
**define** $IndexRow[N]$
**define** $Delim[257]$
**for** $i \leftarrow 0$ **to** $15$ **do**
    **mram_read**$(SubLUT, LUT[16i \cdots 16i+15])$ ▷ parallel in 16 for each tasklet
    **for** $j \leftarrow 0$ **to** $M-1$ **do**
        **if** $Vector[j]$ **not in** $[16i, 16i+15]$ **then**
            **continue**
        **end if**
        **mram_read**$(IndexRow, IndexMat[j])$ ▷ parallel in N for each tasklet
        **mram_read**$(Delim, DelimMat[j][tasklet\_id])$
        $k \leftarrow Delim[0]$
        **while** $Delim[k] \neq -1$ **do**
            $product \leftarrow SubLUT[Vector[j]\&0xF][k]$
            **for** $l \leftarrow$ (**if** $k = 0$ **then** $0$ **else** $Delim[k-1]$) **to** Delim[k] **do**
                $Result[IndexRow[l]] \leftarrow Result[IndexRow[l]] + product$
            **end for**
            $k \leftarrow k+1$
        **end while**
    **end for**
**end for**
**define** $Result\_32[N]$
$Result\_8 \leftarrow$ **BinarySearch**$(Result\_32, FP8\_to\_INT32)$ ▷ parallel in N for each tasklet
**return** $Result\_8$

相比算法3.1，MRAM 中多了 DelimMat，Matrix 变为 IndexMat，因为需要记录索引因此矩阵的元素从一字节变为两字节，MRAM 仍然足够；WRAM 中多了 16 个 Delim 数组，大概占用 8KB 的空间，且 MatRow 变为 IndexRow 多占用 4KB 空间，一共多占用 12KB，如果此时宽矩阵的行数为 128 较小，激活向量占用不了 4KB，总共占用 WRAM 空间 53KB，剩余 11KB 给 16 个线程分配堆栈空间足够。同时每个 tasklet 在处理一行矩阵时由原来的 4096 次查表，变成了现如今最多 256 次查表，仅仅只是多读取了一个 Delim 数组，每行矩阵大概由原本的 8K reads 变为 4.5K reads，差不多有两倍的性能提升。

## 3.4 本章小结

本章主要是介绍了在 UPMEM 硬件上对 GEMV 算子的软件优化，介绍了基本量化方式和查找表基础。在此基础上提出基于查找表分块的矩阵向量乘算法 LUTBlock，该算法拥有非常号的 WRAM 数据局部性，充分优化了数据的流动。随后，针对 Llama2-7B 推理过程中 MHSA 涉及到常见的两种矩阵尺寸，分别是 $4096 \times 128$（窄矩阵）和 $128 \times 4096$（宽矩阵）分别做出了优化：行重排（LUTRow）和列重排（LUTCol），充分考虑 GEMV 计算中访问 WRAM 的特征，优化寄存器局部性，减少无意义的内存读写。

# 4 基于近存计算模拟器的矩阵向量乘硬件优化

本章主要是基于此前的软件优化中出现的硬件架构方面的痛点对 UPMEM 硬件本身做修改，使用的是基于 UPMEM 的周期精确模拟器 PIMulator[63]，分别做两大硬件改动：1）给 UPMEM 增添查找表专用的硬件单元支持乘加操作；2）给 UPMEM 增添 SIMD 单元支持向量操作。基于这两大硬件改动修改软件算法观察 GEMV 算子表现，对 UPMEM 的硬件架构进行探究并给出未来发展建议。

## 4.1 PIMulator 简介

PIMulator 是一个 UPMEM 的周期精确模拟器[63]，它由两个关键组件组成，如图4.1所示：一个是与 UPMEM 指令集架构（Instruction Set Architecture, ISA）兼容的软件编译工具链，以及一个经过真实 UPMEM-PIM 硬件交叉验证的硬件性能模拟器。

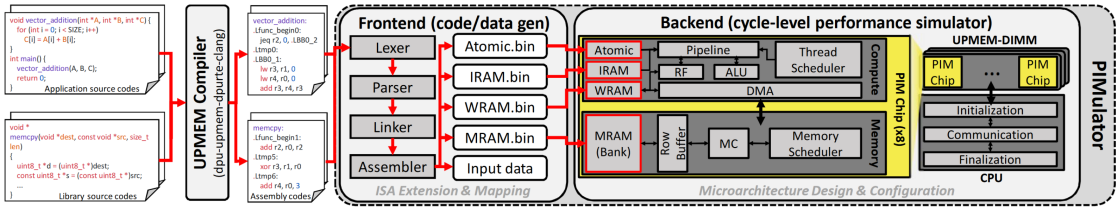



**图 4.1** PIMulator 架构图

PIMulator 的软件编译工具链的一部分基于开源的 UPMEM 软件开发工具包（Software Development Kit，SDK）提供的 LLVM[72]编译器工具链（dpu-upmem-dpurte-clang），另一部分包括自定义的编译器、链接器和汇编器。PIMulator 主要是用 UPMEM SDK 提供的编译器将用户所书写的能在真实硬件上运行的源代码以及和 UPMEM 兼容的 C 标准库程序进行预处理、编译和汇编成二进制对象，最终链接形成 UPMEM 的可执行二进制文件。同时，PIMulator 使用自定义设计的链接器和汇编器，而不是直接使用 UPMEM SDK 的链接器。这是因为 UPMEM 的链接器与 UPMEM 硬件的微架构紧密绑定，限制了对 PIM 硬件架构的自定义能力。例如，当编译后的程序大小或 WRAM 内存使用量超过物理 IRAM 或 WRAM 容量时，UPMEM 的链接器会报错。

PIMulator 的硬件周期精确模拟器设计参考了 UPMEM 的用户手册和公开的微架构信息，主要设计点包括：1）对 DPU 计算核心进行建模：DPU 核心被建模为一个 14 级顺序流水线处理器，充分模拟了流水线的调度算法以及奇偶寄存器堆访问时的限制。2）DRAM

子系统建模：PIMulator 的 DRAM 子系统模拟基于 GPGPU-Sim 的周期级 DRAM 模拟[73]。由于 UPMEM-PIM 的内存调度策略细节未公开，PIMulator 采用了首行优先，先到先服务（First Row, First Come First Served，FR-FCFS）算法来调度内存事务。3）CPU-DPU 通信：设定固定通信值来模拟 CPU 和 DPU 之间的通信延迟，其值通过 UPMEM-PIM 真实硬件的系统性的性能分析来调整。UPMEM 使用 Intel AVX 指令进行 CPU-DPU 通信，PIMulator 同样模拟了这种通信的不对称带宽特性。

PIMulator 的设计采用了模块化结构，将 SPMD 的前端代码/数据生成与后端模拟器清晰分离，类似于 GPGPU-Sim 的设计。这种设计使得 PIMulator 可以轻松扩展以模拟和评估新的软硬件架构。例如希望改动 PIMulator 的硬件，为某种计算提供相应指令，只需要在前端修改自定义的编译、链接和汇编的相关代码识别对应汇编指令，然后在后端模拟器中添加相应的指令处理逻辑即可。

PIMulator 有多个版本的实现①，在论文中实现的是前后端分离的单线程版，使用的 Python 实现前端，C++ 做硬件模拟后端，没有多线程支持，但它的平均模拟速率为 3KIPS（千条指令每秒），与 GPGPU-Sim 相当，在这个版本中，由于 UPMEM 的当前编程模型以及其通信和同步原语的工作方式，DPU 大多数情况下作为独立的处理器运行。PIMulator 另有一版基于 Golang 开发的前后端一体的版本，充分利用多协程（coroutine）实现了模拟速度提 8.5 倍的提升并且内存占用减少 7.5 倍。

## 4.2 查表专用 FMA 指令

在矩阵乘法的计算中，能够观察到一个非常基本的操作：向量和矩阵的元素相乘再与结果向量中的元素相加，将这种乘加操作融合成为单个操作，称为融合乘加（Fused Multiply Add，FMA）操作。在许多 AI 加速芯片中都设计有 MAC 单元（multiplier-accumulator unit）支持 FMA 操作和指令，其中也包括 Intel 的 CPU。将乘加操作融合成为一条指令的优势一方面在于能够使得指令数目减少，减少取指和译码的时间，另一方面就是能够设计专门的电路优化计算，减少消耗周期数，从而整体提高算术吞吐。

类似地，我们可以基于查找表为 UPMEM 增加 LFMA（Look-up Fused Multiply Add，LFMA）单元，专门用于处理使用查找表代替乘法的 FMA 操作。同时，使用查找表查找元素时，需要先计算偏移，再去访存，会花费两条指令，由于 UPMEM 每个周期最多只能执行一条指令，这种计算偏移的开销不可忽略，因此也可以将这部分的计算融入 LFMA 指令当中，设计指令为`LFMA add: reg, base: reg, offset: reg, ele_width: imm`执行的操作是`add+=[base+offset*ele_width]`。但是由于某些情形下只需查表而无需算加法，因此再设计一个查表专用指令 LWO（Load With Offset），类似的`LFMA target: reg, base: re`执行的操作为`target=[base+offset*ele_width]`。

如图4.2所示，在算法3.1的最里层循环里，执行的计算为`Result[k]+=SubLUT[Vector[j]&0xF`

① 模拟器开源仓库：https://github.com/VIA-Research/uPIMulator

```
r0:SubLUT, r2:MatRow, r4:k, r6:Result

lbu r1, r2, r4
lsl r1, r1, 2
lw r3, r0, r1
lsl r5, r4, 2
lw r7, r6, r5
add r7, r7, r3
sw r6, r5, r7
```

(a) 原本汇编代码

```
r0:SubLUT, r2:MatRow, r4:k, r6:Result

lbu r1, r2, r4
lwo r7, r6, r4, 4
lfma r7, r3, r1, 4
sw r6, r5, r7
```

(b) 优化后汇编代码

**图 4.2** 查表专用指令汇编优化

这个语句只有 k 在最内层循环发生改变,因此可以简化为`Result[k]+=SubLUT[MatRow[k]];`可以看到，上述的语句的汇编中，查表 SubLUT 需要计算偏移因为查找表的每个元素是 4 字节，同样最后做累加时需要一起计算偏移做累加，因为结果向量的每个元素也是 4 字节。使用了 LFMA 和 LWO 指令后，指令数量从原本的 7 条减少到 4 条，性能几乎翻倍。

## 4.3 基于 SIMD 指令查表的矩阵向量乘算法

SIMD（Single Instruction Multiple Data）是一种特殊的计算模式，即“单指令多数据”，它允许处理器通过一条指令同时对多个数据执行相同的操作，从而显著提高数据处理的效率和性能。在 SIMD 架构中，处理器执行一条指令，但这条指令会同时作用于多个数据单元，同时执行相同的算术操作，因此 SIMD 指令非常适用于矩阵和向量的数据计算。Intel 是 SIMD 计算模式的主要推动者，为多种架构的处理器配备了 SIMD 处理单元和对应的指令集[59]，SIMD 的数据位宽从一开始的 128bit 逐渐增长到 256bit 最后到 521bit，大大提升了数据处理的效率。

对于更低 bit 量化的权重矩阵，我们可以为 UPMEM 增添 SIMD 指令进行加速。在 Intel AVX-512 指令中,有如下重排指令:`__m512i _mm512_permutexvar_epi32(__m512i index,` 它接受两个 AVX-512 向量，并返回一个 AVX-512 向量，这些 AVX-512 向量由 16 个 32 位整形数组成，这个函数的作用是：按照 index 向量中指示的索引将 a 向量重排，比如 index 向量中第 1 个元素值是 3，表示会将 a 向量中第 4 个元素重排到新向量的第 1 个位置。值的注意的是 index 向量中每个 int32 只用到了最低 4bit 的数值（确保索引值不超过 16）。这种重排与查表操作存在相同之处，查表本身也是给出一张向量（子查找表），读取权重矩阵元素的值作为索引，再去查找表中取得乘积。上述函数中 a 向量为子查找表，权重矩阵的一行的前 16 个元素为 index 向量，相当于使用权重矩阵的值对子查找表做重排，可以一次性查出 16 个乘积结果。假设权重矩阵数据宽度为 4bit，输入向量仍然是 8bit 的位宽，那么查找表为 $256 \times 16$ 的二维数组，仍然展开 FP8 到 int32，那么 LUT 的每一行就是一个 AVX512 向量，刚好满足上述指令的条件。

要想使用 SIMD 计算 GEMV，首先需要对权重矩阵进行重新划分，对于行主存的矩阵，

现将其切分成 $8 \times 16$ 的子块 SubMat 如图4.3，对这个子块我们对其进行重排，将其转换成一个 AVX-512 向量: 将该 AVX-512 向量分组，分为 16 组 INT32，将 SubMat[0][0] 的元素放置到 INT32-0 的最低四位 $[0:4)$，再将 SubMat[1][0] 的元素放置到 INT32-1 的 $[4:8)$，依次类推，直至遍历完 8 行，这样我们就填满了 INT32-0，同样的道理，处理 SubMat 矩阵的第二列，得到 INT32-1，当处理完 SubMat 的所有行和所有列之后，填满了该 AVX-512 向量。将该向量代替原子矩阵进行存储。完成了所有子块的重排之后，矩阵划分结束，其结果是行数减小到原来的 8 倍，列数扩大到原来的 8 倍。这时由于执行的是 $8bit \times 4bit$ 的乘法，查找表的大小为 $256 \times 16 \times 4Byte = 16KB$，可以完全载入 WRAM，因此可以直接按照 GEMV 内积的方式计算。

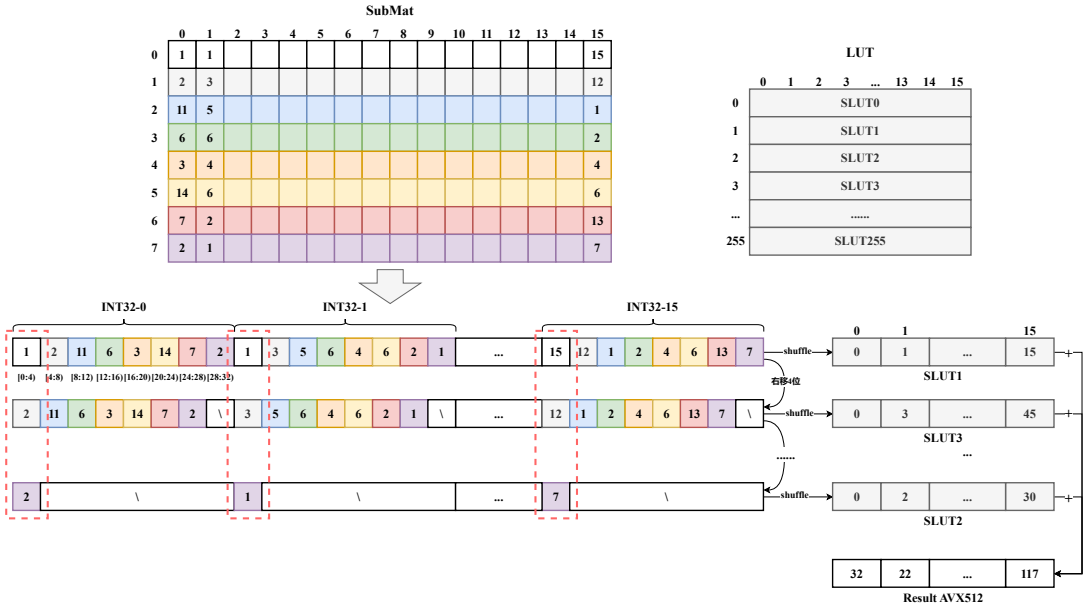



**图 4.3** 使用 SIMD 重排指令的矩阵构建和计算示意图

具体的计算方式也需要将重排后的权重矩阵从行主存改为列主存，也就是以 512bit 的粒度进行转置。转置过后，原来一列的元素现在可以按照行访问提高访问效率，再并行化，每个 tasklet 处理一列。具体到单个 AVX-512 向量计算仍然以图4.3为例，计算该 AVX-512 向量时需要先载入对应的子查找表 SLUT1，载入该行子查找表到 AVX-512 向量 a 中（使用`_mm512_loadu_si512`指令载入寄存器），然后载入该 AVX-512 向量到 index 中，执行`_mm512_permutexvar_epi32`操作得到结果向量，使用`_mm512_add_epi32`指令将其加到累加 AVX-512 结果寄存器上。然后使用`_mm512_srli_epi32`指令对 index 向量右移 4 位，再重复刚才的过程；左移 8 次后完成整个 AVX-512 向量的计算，再类似依次执行该列的所有 AVX-512 向量的计算，执行完成后再将 AVX-512 的累加寄存器写到结果向量的对应位置上。不同的 tasklet 负责不同的列，其对应的结果向量的位置也不同，但是之间互相不干扰无竞争。

要想完成上述工作，需要给 UPMEM 增加相应的向量单元并适配相应的指令集，除了

需要之前介绍的_mm512_permutexvar_epi32指令的支持之外，还需要引入一些新的指令，比如_mm512_loadu_si512指令用于载入 AVX-512 向量，_mm512_add_epi32指令用于向量加法运算，_mm512_srli_epi32指令用于右移运算，_mm512_storeu_si512指令用于将 AVX-512 向量存储到内存中。同时要高效完成上述工作，每个线程至少需要 4 个 AVX-512 寄存器，其中两个分别充当 a 和 index，剩下的两个一个用于暂存向量重排的结果，一个用作累加寄存器。我们将这些指令和寄存器通过 PIMulator 加到 UPMEM 的软硬架构中，根据 Intel AVX-512 指令手册设置运算的每指令周期数（Cycle Per Instruction，CPI），都设置成 1 个时钟周期。

## 4.4 本章小结

# 5 实验结果与分析

本章节主要是对此前在 UPMEM 上进行软硬优化的 GEMV 算子的综合测试，第一小节介绍环境配置平台，第二小节重点测试 UPMEM 近存计算硬件平台的 GEMV 算子总计算能力，并与其他常见的硬件计算平台进行对比，第三小节重点测试在 UPMEM 上的 GEMV 算子优化的效果和运行时间的细分（breakdown），分析瓶颈和优化空间；第四小节介绍 GEMV 算子的扩展性，分析改算子在矩阵尺寸发生变化时的通用性。

## 5.1 环境配置介绍

### 5.1.1 硬件平台

我们实验的近存计算平台是在 UPMEM 官方建议的 UPMEM 服务器上，该服务器配备了双插槽的英特尔至强 4210 CPU，每颗 CPU 拥有 10 个核心 20 个线程，每个核心工作在 2.20GHz 的基准频率，拥有 32KB 的 L1 缓存、1MB 的 L2 缓存和 13.75MB 的 L3 缓存。每个 CPU 配备 6 个内存通道，支持 DDR4-2400 的内存。我们为每个 CPU 的 5 个通道插满 UPMEM DIMM，剩余的一个通道配置常规 DDR4-2400 的内存。每个内存通道插入两根 UPMEM DIMM，每个 UPMEM DIMM 上配有 128 个 DPU。因此一共有 $2\times5\times2\times128=2560$ 个 DPU 可以同时工作，每个 DPU 的存储容量为 64MB，因此 UPMEM 内存的存储容量一共是 160GB。普通 CPU 内存有 128GB。

同时实验对比使用的 CPU 平台是在配备了双插槽的英特尔至强 6242 CPU 的服务器平台上，每颗 CPU 拥有 16 个核心 32 个线程，每个核心工作在 2.80GHz 的基准频率，拥有 32KB 的 L1 缓存、1MB 的 L2 缓存和 22MB 的 L3 缓存。每个 CPU 配备 6 个内存通道，支持 DDR4-2933 的内存。该平台上没有插 UPMEM DIMM，全部配备的是标准 DDR4 内存共 256GB。之所以 CPU 平台和 UPMEM 平台要分开成两套硬件测试，而不能复用 UPMEM 平台的原因是，UPMEM 平台的 CPU 的 6 个内存通道有 5 个都被 UPMEM 占用，CPU 内存传输带宽变为原来的六分之一，这样的对比实验并不公正。

GPU 的硬件装配在 CPU 平台上，由于通过 PCIE 接口连接而并不影响性能。GPU 平台配备了一块 Nvidia A6000 GPU，其核心代号为 GA102，安培架构，拥有 10752 个 CUDA Core 和 336 个 Tensor Core，单精度浮点性能达 38.7TFLOPS。其拥有 48GB 的 GDDR6 显存，384bit 的传输位宽，显存带宽能达到 768GB/s，足以容纳我们评估中使用的数据。

### 5.1.2 数据准备和矩阵尺寸选择

我们选择被广泛使用的开源模型 Llama2-7b-chat，使用 WikiText2[①] 和 PTB[②] 作为量化校准数据集对原始权重进行 FP8/FP4 量化，以机器学习的方式构建 FP8/FP4 查找表，以随机选取的量化后的权重矩阵（MHSA 中的线性层）作为测试数据。我们在主要的算子吞吐测试和优化细分测试中选择两种不同的 GEMV 数据尺寸，分别是 $4096 \times 256$，$256 \times 4096$，分别对应 Llama2-7B 推理过程中可能出现的窄的降维矩阵和宽的升维矩阵（选择 256 维度方便与 SIMD 指令的测试统一维度），在扩展性测试中我们将使用不同矩阵尺寸进行测试。在数据宽度方面，近存计算平台分别选择 Float32 和 FP8（E4M3）两种数据格式进行测试，相应的在 CPU 和 GPU 平台选择 Float32 和 INT8 进行测试（硬件不支持 FP8）。每个 DPU 使用 $4096 \times 256$ 尺寸的矩阵，用满 UPMEM 所有 DPU 进行总的吞吐测试，矩阵的大小为 $4096 \times 256 \times 2560$，假设采用 32bit 的数据格式，矩阵所占空间最大为 $4096 \times 256 \times 2560 \times 4 = 10GB$，上述硬件平台足以存储。

### 5.1.3 基线设置

对于 PIM 平台，真实硬件的测评使用 UPMEM SDK（版本 2024.1.0）编译在 UPMEM-DIMM 执行，模拟器的优化工作通过比较真实硬件性能和周期推算性能表现。我们在近存计算硬件 UPMEME 上的优化分别与 CPU 和 GPU 平台的 Float32 和 INT8 推理做比较，CPU 平台使用英特尔数学核心函数库（Intel Math Kernel Library，MKL）[74]，是英特尔官方开发的一套高性能数学计算库，里面包含 BLAS 接口，针对 Intel（至强处理器）硬件特性进行了深度优化（如使用 SIMD 指令和寄存器），同时能够简单高效地支持多线程和并行计算。而 GPU 平台我们将基于 CUDA（12.2）使用 cuBLAS 库[③]：cuBLAS 是 NVIDIA 官方开发的一个高性能线性代数库，专为 CUDA 平台设计，充分利用了 NVIDIA GPU 的硬件特性，能够显著加速矩阵乘法（GEMM）、向量运算和其他线性代数任务。对于下发的任务包括 GEMV，在支持 Tensor Core 的 GPU 硬件上，cuBLAS 会自动优化选择使用 Cuda Core 还是 Tensor Core 以到达最佳性能。

## 5.2 GEMV 算子运算性能和能效比对比

### 5.2.1 GEMV 算子运算性能对比/

GEMV 算子的运算性能我们选择性能指标每秒浮点运算次数 (Floating-point Operations Per Second, FLOPS) 来衡量，对于矩阵尺寸为 $M \times N$ 的 GEMV 操作来说，具体的计算公式为5.1，其中 Latency 为 GEMV 算子的时延，单位为秒。5.1分别显示了在近存计算平台

① https://huggingface.co/datasets/mindchain/wikitext2/tree/main

② https://aistudio.baidu.com/datasetdetail/67

③ https://docs.nvidia.com/cuda/cublas/index.html

（UPMEM）、CPU 平台和 GPU 平台上的 GEMV 算子运算性能，UPMEM 使用了全部 2560 个 DPU，每个 DPU 配置 16 个 tasklet。

$$GFLOPs = \frac{2 \times M \times N}{Latency} \times 10^9 \tag{5.1}$$

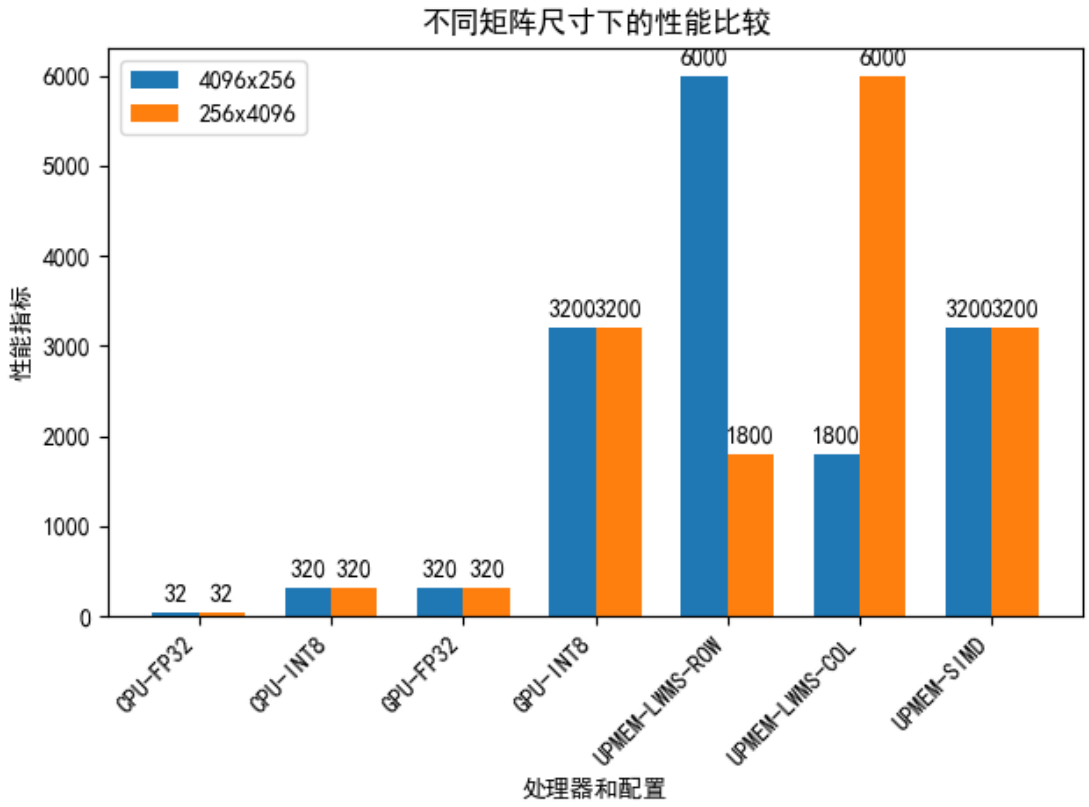
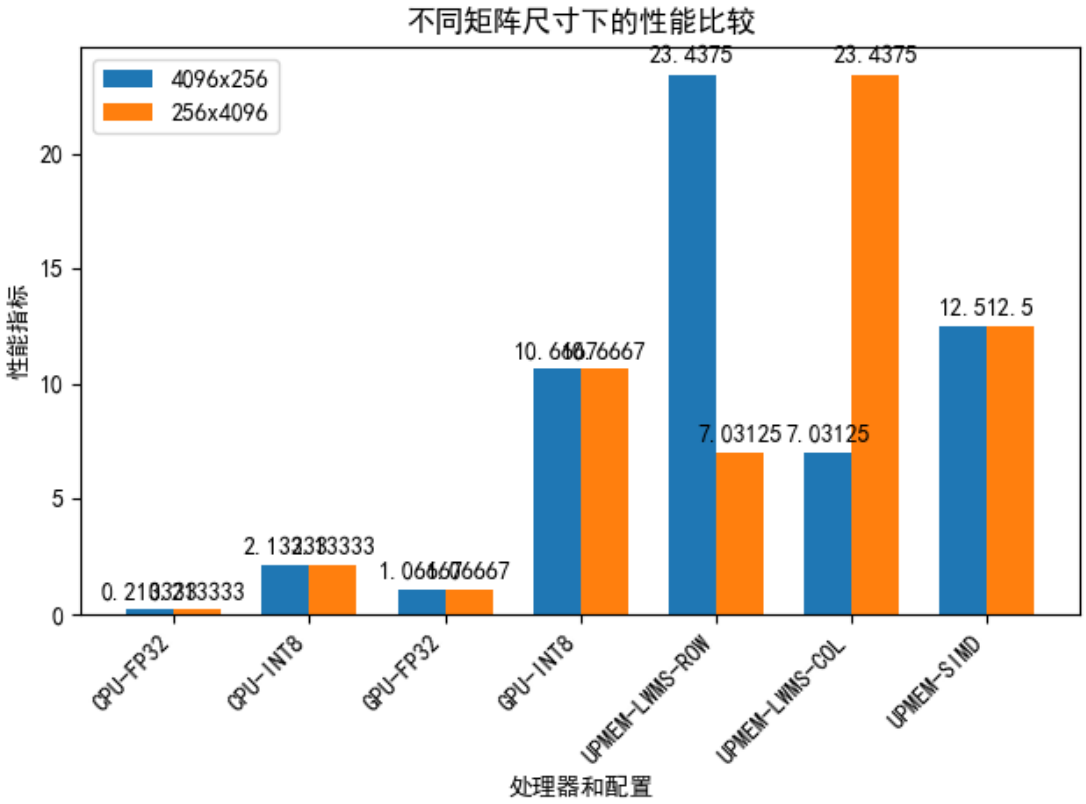



**图 5.1**

可以看到无论是对于哪种矩阵尺寸，在 UPMEM 硬件平台上，GEMV 算子的计算性能峰值能达 387GFLOPS，远超 CPU 和 GPU 平台的计算性能。CPU 平台的计算性能最差，原因大概是因为 10GB 的矩阵数据量庞大，CPU 花费大量时间参与矩阵数据的搬移工作，表现为严重的内存瓶颈。GPU 平台的计算性能表现良好，得益于 GPU 大量高度并行的 CUDA 计算核心和高速的内存带宽传输数据，但是在实验过程中我们通过 Nsight 工具观察到 GPU 的硬件利用率普遍不高。UPMEM 经过精心调优的 GEMV 算子的计算性能最强，而且可以明显地看到对于窄的矩阵，行重排的优化效果最为明显，大概相比较 CPU 有 12 倍的提升，相较 GPU 有 2 倍的提升，而对于宽的矩阵，列重排的效果最为明显，相较于 CPU 有 10 倍的提升，GPU 有 1.5 倍的提升。同时可以看到使用的 SIMD 指令的 GEMV 性能远高于其他算子，相较于 CPU 有 15 倍的提升，相较于 GPU 有 3 倍的提升，这得益于对于硬件的定制化修改以及更低位宽的权重量化。

### 5.2.2 GEMV 算子能效比对比

近存平台的一大优势在于其减少了数据移动的开销从而能够更加节能，非常适合边端对功耗有严格要求的设备，因此我们在测试计算能力的同时也测试了各个硬件平台的能效比。我们使用 Intel VTune Profiler 来测试 CPU 平台上的能耗，同时使用 Nsight 工具来测试 GPU 平台的能耗。对于 UPMEM 平台，由于官方没有提供能耗测试工具，因此我们只能通过 UPMEM SDK 的 dpu-diag 工具测试 DPU 的内核和 DRAM Bank 的静态功耗，大约为 12.8w 每根 DIMM 条。能效比定义简单地计算公式为：$EnergyEffiency = \frac{GFLPOPS}{Energy}$，其中 GFLOPS 为计算性能与上一小节中的测试结果保持一致，Energy 为能耗，单位为 GFLOPS/W。

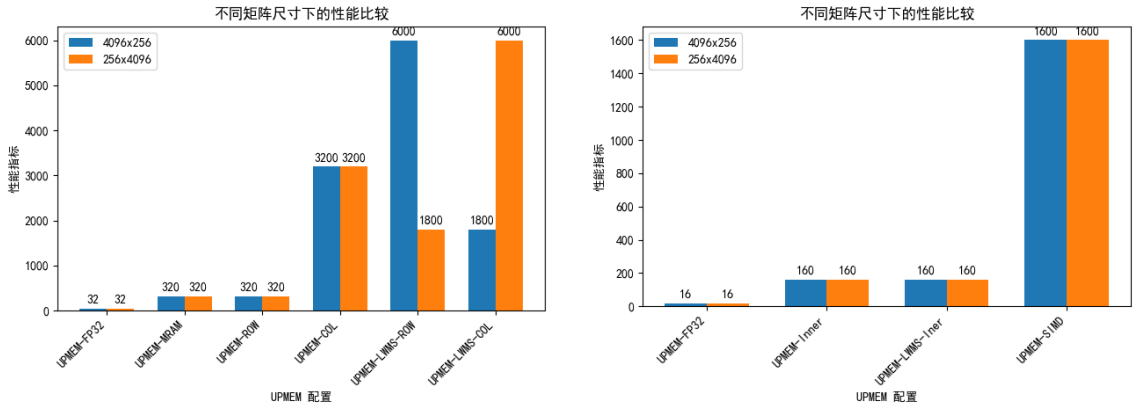



测试结果如5.2.2所示，CPU 平台的能效比最差，其原因主要是由于 CPU 将大量的计算用于搬移数据而非计算，导致计算性能远低于其他两个硬件平台。GPU 的能效比较为良好，得益于其本身强悍的内存带宽以及强大的计算性能。UPMEM 的能效比在三者之间最高，平均达到了 CPU 的 5 倍，GPU 的 1.4 倍，这主要是得益于所有 DPU 的超高的聚合带宽以及优化良好的软硬件协同设计。

## 5.3 GEMV 算子优化和执行时间细分

### 5.3.1 GEMV 算子优化细分

我们在第三章和第四章分析 UPMEM 的硬件特点, 通过软硬协同设计一步一步优化了 GEMV 算子，为分析每步优化的有效性和对总体性能的影响，我们需要将每步优化拆来进行测试，具体测试结果如5.3.1所示，所有的测试都在单 DPU 上进行，测试算子在特定矩阵尺寸下的执行时间，Tasklet 数量设置为 16。




(a) W8A8 数据格式　　(b) W4A8 数据格式

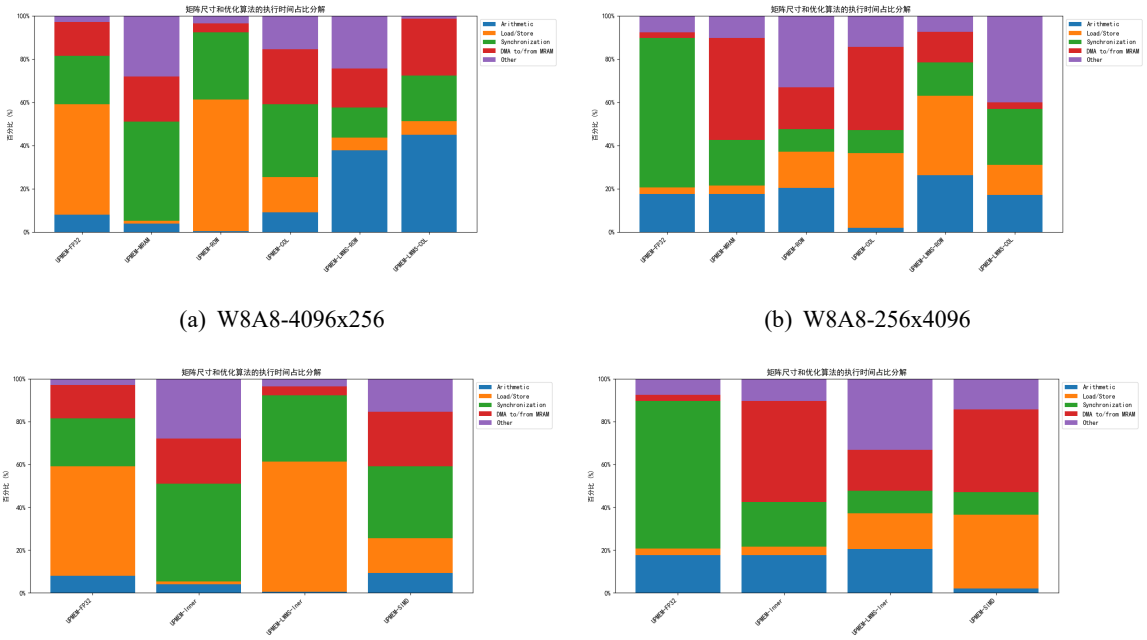
**图 5.2** GEMV 算子优化 Breakdown

我们先看 W8A8 数据格式的优化5.2(a),基准是未经优化的 FP32 的 GEMV 算子（UPMEM-FP32），然后是基于查找表分块的优化（UPMEM-MRAM），然后分别是在真实硬件上的行、列重排（UPMEM-ROW，UPMEM-COL）和模拟器上的基于新增 FMA 指令的行列重排优化（UPMEM-FMA-ROW，UPMEM-FMA-COL）。数据结果显示相较于 UPMEM-FP32，UPMEM-MRAM 的优化提升巨大，有 30 倍的提升；对于窄矩阵（$4096 \times 256$），行重排的提升有 8 倍，对于宽矩阵来说（$256 \times 4096$），列重排的提升大概 2.5 倍，但是行重排之于宽矩阵，列重排之于窄矩阵提升就不大了；在模拟器行增加了查找表专用的 FMA 指令后，平均提升大概 3-4 倍，与理论分析一致。

对于 W4A8 数据格式的 GEMV 算子优化，基准仍然是 UPMEM-FP32，由于查找表可以完全放入 WRAM 因此没有多余的设计，在 UPMEM 上就是很简单地使用 GEMV 的内积效率就很高，因此第二个优化就是 UPMEM-Inner，对于 FP32 的提升非常高达 50 倍；在模拟器上仍然可以使用 FMA 指令加速，提升大概也是 3-4 倍，使用 SIMD 指令加速提升大概有 16 倍，可以看到初开 SIMD 对权重数据位宽的要求外，SIMD 指令用于查表的提升仍然很大。

### 5.3.2 GEMV 算子执行时间细分

为了分析每种优化的的瓶颈，我们需要对每种优化下 GEMV 算子的执行时间进行细分，分别统计在计算（Arithmetic），访存（WRAM Load/Store），线程间同步（Synchronization），MRAM 的数据传输耗时（DMA to/from MRAM），以及其他这 6 种操作上的耗时，分析瓶颈所在和优化有效性。我们仍然在单个 DPU 内测试，设置 Tasklet 的数量为 16.




(a) W8A8-4096x256　　(b) W8A8-256x4096

(c) W4A8-4096x256　　(d) W4A8-256x4096

**图 5.3** GEMV 算子优化 Breakdown

我们可以看到，UPMEM 的绝大部份的时间都耗费在访存和计算上，同步时间和 DMA 时间都可以忽略不计，这说明我们的优化算法对于数据局部性的优化非常的成功同时高度利用了多线程并行化计算。

## 5.4 扩展性测试

扩展性测试可以分为强弱扩展性测试，所谓强扩展性就是在问题规模一定的情况下提高计算能力，而弱扩展性就是在问题规模变大时，等比例扩大计算能力使得总问题规模比上计算能力保持不变。在这里问题规模其实就是矩阵的大小，计算能力在大的粒度上可以认为是 DPU 的数量，在小的粒度上可以认为 Tasklet 的数量。由于 GEMV 可以按照列随意切分矩阵的粒度，因此在这里我们不考虑 DPU 粒度的计算能力，将具体实验测试限制在单个 DPU 中，同时由于 Tasklet 的特殊，我们取消弱扩展性测试改为负载能力测试，即给定计算能力下扩大数据规模，观察负载能力随数据规模的变化。

### 5.4.1 强扩展性测试

对于强扩展性测试，我们测试的矩阵大小分别是 4096 × 256 和 256 × 4096，分别对应宽和窄矩阵，具体的 GEMV 算子我们选择 W8A8 数据格式下的行、列重排，即 UPMEM-ROW-FMA，UPMEM-COL-FMA，W4A8 数据格式下选择 UPMEM-SIMD，设置 Tasklet 数量从 1 逐渐增加到 24，测试各个算子的执行时间：

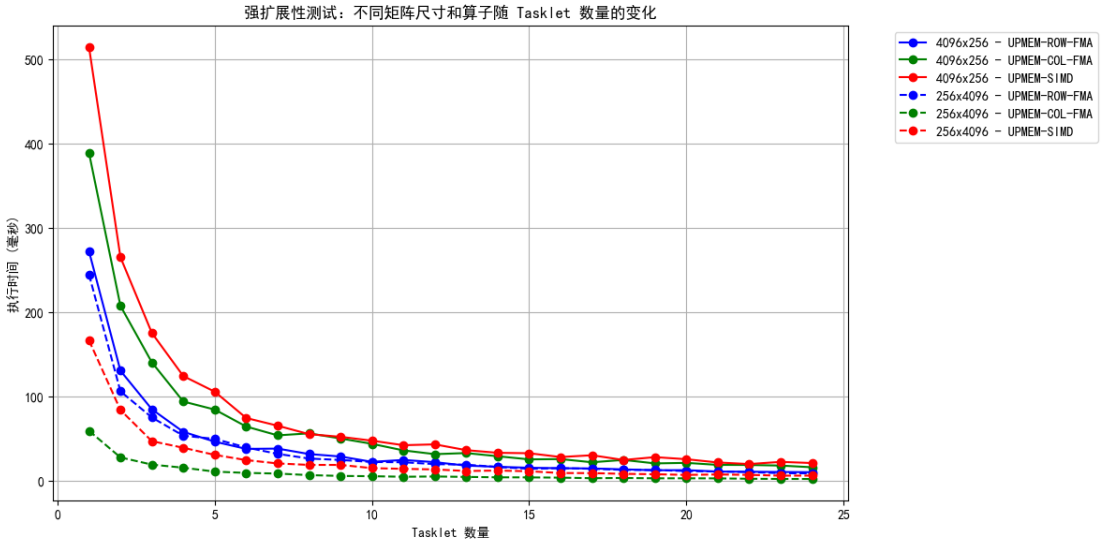
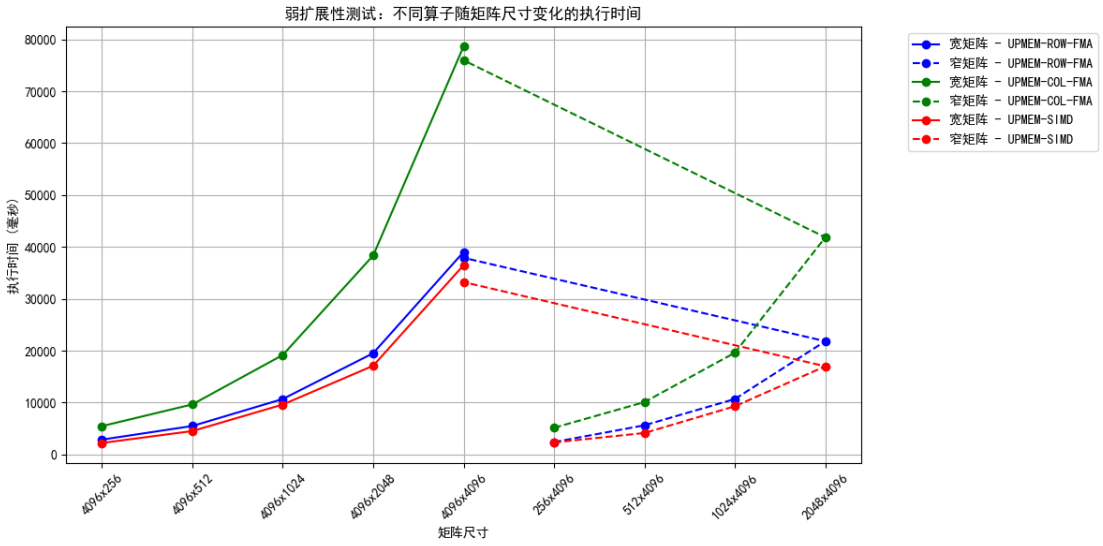



**图 5.4**

可以看到随着 Tasklet 的增加各个算子的执行时间随之减少，当 Tasklet 超过 11 个时，计算达到饱和。

### 5.4.2 负载能力测试

我们同样选择 UPMEM-ROW-FMA，UPMEM-COL-FMA，和 UPMEM-SIMD 这三个算子，设置 Tasklet 数量为 16，改变工作负载，将矩阵的尺寸从 4096 × 256 翻倍变化到 4096 × 4096，同样将 256 × 4096 翻倍变化到 4096 × 4096，测试各个算子的执行时间：


弱扩展性测试：不同算子随矩阵尺寸变化的执行时间
执行时间（毫秒）
矩阵尺寸
4096x256
4096x512
4096x1024
4096x2048
4096x4096
256x4096
512x4096
1024x4096
2048x4096
宽矩阵 - UPMEM-ROW-FMA
窄矩阵 - UPMEM-ROW-FMA
宽矩阵 - UPMEM-COL-FMA
窄矩阵 - UPMEM-COL-FMA
宽矩阵 - UPMEM-SIMD
窄矩阵 - UPMEM-SIMD

# 6 总结与展望

## 6.1 总结

## 6.2 展望

# 参考文献

[1] Radford A, Narasimhan K. Improving Language Understanding by Generative Pre-Training [C/OL]//. 2018. https://api.semanticscholar.org/CorpusID:49313245.

[2] Touvron H, Martin L, Stone K R, et al. Llama 2: Open Foundation and Fine-Tuned Chat Models[J/OL]. ArXiv, 2023, abs/2307.09288. https://api.semanticscholar.org/CorpusID:259950998.

[3] Kim J H, Ro Y, So J, et al. Samsung PIM/PNM for Transfmer Based AI : Energy Efficiency on PIM/PNM Cluster[C]//2023 IEEE Hot Chips 35 Symposium (HCS). 2023: 1-31. DOI: 10.1109/HCS59251.2023.10254711.

[4] Yu G I, Jeong J S, Kim G W, et al. Orca: A Distributed Serving System for Transformer-Based Generative Models[C/OL]//16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). Carlsbad, CA: USENIX Association, 2022: 521-538. https://www.usenix.org/conference/osdi22/presentation/yu.

[5] Kautz W. Cellular Logic-in-Memory Arrays[J]. IEEE Transactions on Computers, 1969, C-18(8): 719-727. DOI: 10.1109/T-C.1969.222754.

[6] Lee S, Kang S h, Lee J, et al. Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology : Industrial Product[C]//2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA). 2021: 43-56. DOI: 10.1109/ISCA52012.2021.00013.

[7] Ke L, Zhang X, So J, et al. Near-Memory Processing in Action: Accelerating Personalized Recommendation With AxDIMM[J]. IEEE Micro, 2022, 42(1): 116-127. DOI: 10.1109/MM.2021.3097700.

[8] Lee S, Kim K, Oh S, et al. A 1ynm 1.25V 8Gb, 16Gb/s/pin GDDR6-based Accelerator-in-Memory supporting 1TFLOPS MAC Operation and Various Activation Functions for Deep-Learning Applications[C]//2022 IEEE International Solid-State Circuits Conference (ISSCC): vol. 65. 2022: 1-3. DOI: 10.1109/ISSCC42614.2022.9731711.

[9] Niu D, Li S, Wang Y, et al. 184QPS/W 64Mb/mm23D Logic-to-DRAM Hybrid Bonding with Process-Near-Memory Engine for Recommendation System[C]//2022 IEEE International Solid-State Circuits Conference (ISSCC): vol. 65. 2022: 1-3. DOI: 10.1109/ISSCC42614.2022.9731694.

[10] Devaux F. The true Processing In Memory accelerator[C]//2019 IEEE Hot Chips 31 Symposium (HCS). 2019: 1-24. DOI: 10.1109/HOTCHIPS.2019.8875680.

[11] Stone H S. A Logic-in-Memory Computer[J]. IEEE Transactions on Computers, 1970, C-19(1): 73-78. DOI: 10.1109/TC.1970.5008902.

[12] Wulf W A, McKee S A. Hitting the memory wall: implications of the obvious[J/OL]. SIGARCH Comput. Archit. News, 1995, 23(1): 20-24. https://doi.org/10.1145/216585.216588. DOI: 10.1145/216585.216588.

[13] Seshadri V, Kim Y, Fallin C, et al. RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization[C]//2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). 2013: 185-197.

[14] Seshadri V, Hsieh K, Boroum A, et al. Fast Bulk Bitwise AND and OR in DRAM[J]. IEEE Computer Architecture Letters, 2015, 14(2): 127-131. DOI: 10.1109/LCA.2015.2434872.

[15] Seshadri V, Lee D, Mullins T, et al. Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology[C]//2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). 2017: 273-287.

[16] Balasubramonian R, Chang J, Manning T, et al. Near-Data Processing: Insights from a MICRO-46 Workshop[J]. IEEE Micro, 2014, 34(4): 36-42. DOI: 10.1109/MM.2014.55.

[17] Ahn J, Hong S, Yoo S, et al. A scalable processing-in-memory accelerator for parallel graph processing[C]//2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA). 2015: 105-117. DOI: 10.1145/2749469.2750386.

[18] Gómez-Luna J, Hajj I E, Fernandez I, et al. Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System[J]. IEEE Access, 2022, 10: 52565-52608. DOI: 10.1109/ACCESS.2022.3174101.

[19] Falevoz Y, Legriel J. Energy Efficiency Impact of Processing in Memory: A Comprehensive Review of Workloads on the UPMEM Architecture[C/OL]//Euro-Par 2023: Parallel Processing Workshops: Euro-Par 2023 International Workshops, Limassol, Cyprus, August 28–September 1, 2023, Revised Selected Papers, Part II. Limassol, Cyprus: Springer-Verlag, 2024: 155-166. https://doi.org/10.1007/978-3-031-48803-0_13. DOI: 10.1007/978-3-031-48803-0_13.

[20] Nider J, Mustard C, Zoltan A, et al. A Case Study of Processing-in-Memory in off-the-Shelf Systems[C/OL]//2021 USENIX Annual Technical Conference (USENIX ATC 21). USENIX Association, 2021: 117-130. https://www.usenix.org/conference/atc21/presentation/nider.

[21] Friesel B, Lütke Dreimann M, Spinczyk O. A Full-System Perspective on UPMEM Performance[C/OL]//DIMES '23: Proceedings of the 1st Workshop on Disruptive Memory

Systems. Koblenz, Germany: Association for Computing Machinery, 2023: 1-7. https://doi.org/10.1145/3609308.3625266. DOI: 10.1145/3609308.3625266.

[22] Zois V, Gupta D, Tsotras V J, et al. Massively parallel skyline computation for processing-in-memory architectures[C/OL]//PACT '18: Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques. Limassol, Cyprus: Association for Computing Machinery, 2018. https://doi.org/10.1145/3243176.3243187. DOI: 10.1145/3243176.3243187.

[23] Kang H, Gibbons P B, Blelloch G E, et al. The Processing-in-Memory Model[C/OL]//SPAA '21: Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures. Virtual Event, USA: Association for Computing Machinery, 2021: 295-306. https://doi.org/10.1145/3409964.3461816. DOI: 10.1145/3409964.3461816.

[24] Kang H, Zhao Y, Blelloch G E, et al. PIM-Tree: A Skew-Resistant Index for Processing-in-Memory[J/OL]. Proc. VLDB Endow., 2022, 16(4): 946-958. https://doi.org/10.14778/3574245.3574275. DOI: 10.14778/3574245.3574275.

[25] Kang H, Zhao Y, Blelloch G E, et al. PIM-trie: A Skew-resistant Trie for Processing-in-Memory[C/OL]//SPAA '23: Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures. Orlando, FL, USA: Association for Computing Machinery, 2023: 1-14. https://doi.org/10.1145/3558481.3591070. DOI: 10.1145/3558481.3591070.

[26] Bernhardt A, Koch A, Petrov I. pimDB: From Main-Memory DBMS to Processing-In-Memory DBMS-Engines on Intelligent Memories[C/OL]//DaMoN '23: Proceedings of the 19th International Workshop on Data Management on New Hardware. Seattle, WA, USA: Association for Computing Machinery, 2023: 44-52. https://doi.org/10.1145/3592980.3595312. DOI: 10.1145/3592980.3595312.

[27] Baumstark A, Jibril M A, Sattler K U. Accelerating Large Table Scan using Processing-In-Memory Technology[G]//BTW 2023. Bonn: Gesellschaft für Informatik e.V., 2023: 797-814. DOI: 10.18420/BTW2023-51.

[28] Baumstark A, Jibril M A, Sattler K U. Adaptive Query Compilation with Processing-in-Memory[C]//2023 IEEE 39th International Conference on Data Engineering Workshops (ICDEW). 2023: 191-197. DOI: 10.1109/ICDEW58674.2023.00035.

[29] Lim C, Lee S, Choi J, et al. Design and Analysis of a Processing-in-DIMM Join Algorithm: A Case Study with UPMEM DIMMs[J/OL]. Proc. ACM Manag. Data, 2023, 1(2). https://doi.org/10.1145/3589258. DOI: 10.1145/3589258.

[30] Lavenier D, Roy J F, Furodet D. DNA mapping using Processor-in-Memory architecture[C]//2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM). 2016: 1429-1435. DOI: 10.1109/BIBM.2016.7822732.

[31] Lavenier D, Cimadomo R, Jodin R. Variant Calling Parallelization on Processor-in-Memory Architecture[C]//2020 IEEE International Conference on Bioinformatics and Biomedicine (BIBM). 2020: 204-207. DOI: 10.1109/BIBM49941.2020.9313351.

[32] Chen L C, Yu S Q, Ho C C, et al. RNA-seq Quantification on Processing in memory Architecture: Observation and Characterization[C]//2022 IEEE 11th Non-Volatile Memory Systems and Applications Symposium (NVMSA). 2022: 26-32. DOI: 10.1109/NVMSA56066.2022.00014.

[33] Chen L C, Ho C C, Chang Y H. UpPipe: A Novel Pipeline Management on In-Memory Processors for RNA-seq Quantification[C]//2023 60th ACM/IEEE Design Automation Conference (DAC). 2023: 1-6. DOI: 10.1109/DAC56929.2023.10247915.

[34] Abecassis N, Gómez-Luna J, Mutlu O, et al. GAPiM: Discovering Genetic Variations on a Real Processing-in-Memory System[J/OL]. bioRxiv, 2023. eprint: https://www.biorxiv.org/content/early/2023/07/29/2023.07.26.550623.full.pdf. https://www.biorxiv.org/content/early/2023/07/29/2023.07.26.550623. DOI: 10.1101/2023.07.26.550623.

[35] Zarif N. Offloading embedding lookups to processing-in-memory for deep learning recommender models[D]. University of British Columbia, 2023.

[36] Gómez-Luna J, Guo Y, Brocard S, et al. An Experimental Evaluation of Machine Learning Training on a Real Processing-in-Memory System[EB/OL]. 2023. https://arxiv.org/abs/2207.07886. arXiv: 2207.07886 [cs.AR].

[37] Das P, Sutradhar P R, Indovina M, et al. Implementation and Evaluation of Deep Neural Networks in Commercially Available Processing in Memory Hardware[C]//2022 IEEE 35th International System-on-Chip Conference (SOCC). 2022: 1-6. DOI: 10.1109/SOCC56010.2022.9908126.

[38] Giannoula C, Yang P, Fernandez I, et al. PyGim : An Efficient Graph Neural Network Library for Real Processing-In-Memory Architectures[J/OL]. Proc. ACM Meas. Anal. Comput. Syst., 2024, 8(3). https://doi.org/10.1145/3700434. DOI: 10.1145/3700434.

[39] Li C, Zhou Z, Wang Y, et al. PIM-DL: Expanding the Applicability of Commodity DRAM-PIMs for Deep Learning via Algorithm-System Co-Optimization[C/OL]//ASPLOS '24: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. La Jolla, CA, USA: Association for Computing Machinery, 2024: 879-896. https://doi.org/10.1145/3620665.3640376. DOI: 10.1145/3620665.3640376.

[40] Gogineni K, Dayapule S S, Gómez-Luna J, et al. SwiftRL: Towards Efficient Reinforcement Learning on Real Processing-In-Memory Systems[C]//2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). 2024: 217-229. DOI: 10.1109/ISPASS61541.2024.00029.

[41] Rhyner S, Luo H, Gómez-Luna J, et al. PIM-Opt: Demystifying Distributed Optimization Algorithms on a Real-World Processing-In-Memory System[C/OL]//PACT '24: Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques. Long Beach, CA, USA: Association for Computing Machinery, 2024: 201-218. https://doi.org/10.1145/3656019.3676947. DOI: 10.1145/3656019.3676947.

[42] Vaswani A, Shazeer N, Parmar N, et al. Attention is All you Need[C/OL]//Guyon I, Luxburg U V, Bengio S, et al. Advances in Neural Information Processing Systems: vol. 30. Curran Associates, Inc., 2017. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[43] Devlin J, Chang M W, Lee K, et al. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding[C/OL]//Burstein J, Doran C, Solorio T. Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers). Minneapolis, Minnesota: Association for Computational Linguistics, 2019: 4171-4186. https://aclanthology.org/N19-1423/. DOI: 10.18653/v1/N19-1423.

[44] Raffel C, Shazeer N, Roberts A, et al. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer[EB/OL]. 2023. https://arxiv.org/abs/1910.10683. arXiv: 1910.10683 [cs.LG].

[45] Radford A, Wu J, Child R, et al. Language Models are Unsupervised Multitask Learners [C/OL]//. 2019. https://api.semanticscholar.org/CorpusID:160025533.

[46] Brown T B, Mann B, Ryder N, et al. Language models are few-shot learners[C]//NIPS '20: Proceedings of the 34th International Conference on Neural Information Processing Systems. Vancouver, BC, Canada: Curran Associates Inc., 2020.

[47] Wang T, Roberts A, Hesslow D, et al. What Language Model Architecture and Pretraining Objective Work Best for Zero-Shot Generalization?[EB/OL]. 2022. https://arxiv.org/abs/2204.05832. arXiv: 2204.05832 [cs.CL].

[48] Dai D, Sun Y, Dong L, et al. Why Can GPT Learn In-Context? Language Models Implicitly Perform Gradient Descent as Meta-Optimizers[EB/OL]. 2023. https://arxiv.org/abs/2212.10559. arXiv: 2212.10559 [cs.CL].

[49] Zhou Z, Ning X, Hong K, et al. A Survey on Efficient Inference for Large Language Models [EB/OL]. 2024. https://arxiv.org/abs/2404.14294. arXiv: 2404.14294 [cs.CL].

[50] Dettmers T, Lewis M, Belkada Y, et al. LLM.int8(): 8-bit matrix multiplication for transformers at scale[C]//NIPS '22: Proceedings of the 36th International Conference on Neural Information Processing Systems. New Orleans, LA, USA: Curran Associates Inc., 2022.

[51] Xiao G, Lin J, Seznec M, et al. SmoothQuant: accurate and efficient post-training quantization for large language models[C]//ICML'23: Proceedings of the 40th International

Conference on Machine Learning. Honolulu, Hawaii, USA: JMLR.org, 2023.

[52] Lin J, Tang J, Tang H, et al. AWQ: Activation-aware Weight Quantization for On-Device LLM Compression and Acceleration[J/OL]. GetMobile: Mobile Comp. and Comm., 2025, 28(4): 12-17. https://doi.org/10.1145/3714983.3714987. DOI: 10.1145/3714983.3714987.

[53] Frantar E, Singh S P, Alistarh D. Optimal brain compression: a framework for accurate post-training quantization and pruning[C]//NIPS '22: Proceedings of the 36th International Conference on Neural Information Processing Systems. New Orleans, LA, USA: Curran Associates Inc., 2022.

[54] Yao Z, Aminabadi R Y, Zhang M, et al. ZeroQuant: efficient and affordable post-training quantization for large-scale transformers[C]//NIPS '22: Proceedings of the 36th International Conference on Neural Information Processing Systems. New Orleans, LA, USA: Curran Associates Inc., 2022.

[55] Yao Z, Wu X, Li C, et al. Exploring Post-training Quantization in LLMs from Comprehensive Study to Low Rank Compensation[J/OL]. Proceedings of the AAAI Conference on Artificial Intelligence, 2024, 38(17): 19377-19385. https://ojs.aaai.org/index.php/AAAI/article/view/29908. DOI: 10.1609/aaai.v38i17.29908.

[56] Wu X, Yao Z, He Y. ZeroQuant-FP: A Leap Forward in LLMs Post-Training W4A8 Quantization Using Floating-Point Formats[EB/OL]. 2023. https://arxiv.org/abs/2307.09782. arXiv: 2307.09782 [cs.LG].

[57] Strassen V. Gaussian elimination is not optimal[J/OL]. Numerische Mathematik, 1969, 13(4): 354-356. https://doi.org/10.1007/BF02165411. DOI: 10.1007/BF02165411.

[58] Luebke D. CUDA: Scalable parallel programming for high-performance scientific computing[C]//2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro. 2008: 836-838. DOI: 10.1109/ISBI.2008.4541126.

[59] Jeong H, Kim S, Lee W, et al. Performance of SSE and AVX Instruction Sets[EB/OL]. 2012. https://arxiv.org/abs/1211.0820. arXiv: 1211.0820 [hep-lat].

[60] Choquette J, Gandhi W, Giroux O, et al. NVIDIA A100 Tensor Core GPU: Performance and Innovation[J]. IEEE Micro, 2021, 41(2): 29-35. DOI: 10.1109/MM.2021.3061394.

[61] Gokhale M, Holmes B, Iobst K. Processing in memory: the Terasys massively parallel PIM array[J]. Computer, 1995, 28(4): 23-31. DOI: 10.1109/2.375174.

[62] Kestor G, Gioiosa R, Kerbyson D J, et al. Quantifying the energy cost of data movement in scientific applications[C]//2013 IEEE International Symposium on Workload Characterization (IISWC). 2013: 56-65. DOI: 10.1109/IISWC.2013.6704670.

[63] Hyun B, Kim T, Lee D, et al. Pathfinding Future PIM Architectures by Demystifying a Commercial PIM Technology[C]//2024 IEEE International Symposium on High-Performance

Computer Architecture (HPCA). 2024: 263-279. DOI: 10.1109/HPCA57654.2024.00029.

[64] Khan A A, Farzaneh H, Friebel K F A, et al. CINM (Cinnamon): A Compilation Infrastructure for Heterogeneous Compute In-Memory and Compute Near-Memory Paradigms [EB/OL]. 2024. https://arxiv.org/abs/2301.07486. arXiv: 2301.07486 [cs.AR].

[65] Chen J, Gómez-Luna J, El Hajj I, et al. SimplePIM: A Software Framework for Productive and Efficient Processing-in-Memory[C]//2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT). 2023: 99-111. DOI: 10.1109/PACT58117.2023.00017.

[66] Giannoula C, Fernandez I, Luna J G, et al. SparseP: Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Architectures[J/OL]. Proc. ACM Meas. Anal. Comput. Syst., 2022, 6(1). https://doi.org/10.1145/3508041. DOI: 10.1145/3508041.

[67] Item M, Oliveira G F, Gómez-Luna J, et al. TransPimLib: Efficient Transcendental Functions for Processing-in-Memory Systems[C]//2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). 2023: 235-247. DOI: 10.1109/ISPASS57527.2023.00031.

[68] Noh S U, Hong J, Lim C, et al. PID-Comm: A Fast and Flexible Collective Communication Framework for Commodity Processing-in-DIMM Devices[C]//2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA). 2024: 245-260. DOI: 10.1109/ISCA59077.2024.00027.

[69] Zhou Z, Li C, Yang F, et al. DIMM-Link: Enabling Efficient Inter-DIMM Communication for Near-Memory Processing[C]//2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA). 2023: 302-316. DOI: 10.1109/HPCA56546.2023.10071005.

[70] Liu Z, Cheng K T, Huang D, et al. Nonuniform-to-Uniform Quantization: Towards Accurate Quantization via Generalized Straight-Through Estimation[C]//2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). 2022: 4932-4942. DOI: 10.1109/CVPR52688.2022.00489.

[71] Micikevicius P, Stosic D, Burgess N, et al. FP8 Formats for Deep Learning[EB/OL]. 2022. https://arxiv.org/abs/2209.05433. arXiv: 2209.05433 [cs.LG].

[72] Lattner C, Adve V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation[C]//CGO '04: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization. Palo Alto, California: IEEE Computer Society, 2004: 75.

[73] Bakhoda A, Yuan G L, Fung W W L, et al. Analyzing CUDA workloads using a detailed GPU simulator[C]//2009 IEEE International Symposium on Performance Analysis of Systems and Software. 2009: 163-174. DOI: 10.1109/ISPASS.2009.4919648.

[74] Wang E, Zhang Q, Shen B, et al. Intel Math Kernel Library[M/OL]//High-Performance Computing on the Intel® Xeon Phi™: How to Fully Exploit MIC Architectures. Cham: Springer International Publishing, 2014: 167-188. https://doi.org/10.1007/978-3-319-06486-4_7. DOI: 10.1007/978-3-319-06486-4_7.

# 致谢

时光荏苒，两年的专业硕士学习生涯即将画上句号，在这段旅程的终点，我心中满是感恩，千言万语如潮水般涌上心头。

我要将最诚挚的敬意与感谢献给我的指导老师王晶教授。从论文选题时的迷茫与徘徊，到构思阶段的反复斟酌，再到撰写过程中的字斟句酌，直至最终定稿的每一个细节，王老师都给予了我悉心的指导和耐心的帮助。她严谨的治学态度，体现在对每一个数据、每一处引用的严格把关；她渊博的学术知识，犹如一座取之不尽的宝库，总能在我困惑时提供多元的思路；她精益求精的工作作风，更是深深烙印在我的心中，激励着我不断追求卓越。当研究陷入瓶颈，数据出现偏差，思路陷入僵局时，王老师总是能以其敏锐的洞察力，迅速发现问题的关键所在，为我指明前行的方向，用温暖而坚定的话语鼓励我坚持下去。她不仅是我学术道路上的引路人，在生活中，她的为人处世、对待困难的从容态度，也成为我人生路上的榜样，让我深刻领悟了为学与为人的真谛。回首这段历程，若没有王老师的指导与支持，我绝不可能顺利完成这篇毕业论文。

在实验室的日子里，师兄师弟们也给了我莫大的帮助。初入实验室时，面对复杂的机器环境和复杂的操作流程，我满心茫然。师兄们凭借丰富的经验，手把手地教我科研，从选题的发现、文献的调研，到实验步骤的具体实施，每一个环节都耐心示范。他们还毫无保留地分享自己的研究心得，讲述曾经遇到的问题及解决方法，让我少走了许多弯路。师弟们积极向上的态度和对科研的热情，也时刻感染着我。在实验紧张忙碌时，大家相互打气；在数据出现异常时，一起查阅资料、分析原因。我们围坐在实验台前，激烈讨论研究方案，那些一起在实验室忙碌的日夜，不仅充实了我的知识储备，更让我收获了珍贵的友谊。

而家人，始终是我最坚实的后盾。我的母亲，在生活中给予我无微不至的关怀。清晨，她早早起床准备营养丰富的早餐；夜晚，当我还在书桌前埋头苦读时，她会悄悄端来一杯热牛奶。家中的大小事务，她默默承担，从不让我操心，让我能够心无旁骛地专注于学业。每当我遇到挫折情绪低落时，母亲总是用温暖的话语安慰我，回忆我过往取得的成绩，鼓励我重新振作。她的坚韧与乐观，一直潜移默化地影响着我，是我不断前进的动力源泉。

在未来的日子里，我会带着这份感恩，将所学用于实践，不辜负所有给予我帮助的人。愿王老师的教诲如春风化雨，培育出更多优秀的学子，桃李满天下；愿实验室的伙伴们在科研道路上，能突破重重难关，一帆风顺；愿家人平安健康，岁月温柔以待。