中国人民大学

专 业 硕 士 学 位 论 文

（中文题目）基于近存计算技术的矩阵向量乘优化研究

（英文题目）Research on GEMV Optimization Based on Processing-in-Memory Technology

作 者 学 号：2023103756

作 者 姓 名：张 魁 耀 辉

所 在 学 院：信 息 学 院

专 业 名 称：大数据技术与工程

导 师 姓 名：王 晶

论 文 主 题 词：近数计算；矩阵向量乘；查找表

论文提交日期：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（LLM）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，其中矩阵向量乘算子（GEMV）在大模型推理中占主导地位。GEMV 因自身特点为内存瓶颈任务，GPU 等计算密集型硬件利用率低，在边端场景下常成系统瓶颈。近存计算是新兴计算范式，能解决传统计算架构内存瓶颈问题。新兴的近存计算硬件，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。

本论文基于近存计算技术研究算子 GEMV 的软硬协同加速方案：1）在近存计算硬件上，设计基于查找表的矩阵向量乘算法，使用查找表消除复杂计算，对近存计算硬件中的多级存储结构分别做出了访存优化，包括查找表分块算法减少通过 DMA 访问内存的次数，以及矩阵的行列重排减少对缓存的访问，增强寄存器的数据局部性；2）在模拟器上，为解决真实硬件上软件算法的计算瓶颈，基于周期精确近存激素模拟器设计硬件及指令以增强其计算能力，包括融合查表和加法指令用于高效快速查询乘积并累加，以及设计基于查找表的向量指令进行向量化的查表和访存，提高计算效率。最后对上述算法进行了详细的测试，分别在 CPU 和 GPU 以及近存硬件三个硬件平台上，进行了包括对 GEMV 算子总吞吐和能效比的测试。实验结果表明，在近存计算硬件上 GEMV 算子吞吐为 355GOPS，大概达到理论性能的 93%，是 CPU 平台的 14.7 倍，能效比大概是 CPU 平台的 8.6 倍，是 GPU 平台的 1.13 倍。

关键词：近数计算　矩阵向量乘　查找表

Abstract

The successful application of Large Language Models (LLMs) in multiple domains has propelled the advancement of artificial intelligence. Nevertheless, they are plagued by drawbacks such as high computational costs and substantial memory demands. Optimizing the inference efficiency and reducing resource consumption have emerged as core issues. Mainstream LLMs adopt the Decoder-Only architecture, within which the Generalized Matrix-Vector Multiplication (GEMV) holds a dominant position in LLM inferences. Due to its inherent characteristics, GEMV constitutes a memory bounded task, with the utilization of computing-intensive hardware like GPUs being low. It has always become a system bottleneck in edge scenarios. Processing-in-Memory (PIM) represents an emerging computing paradigm capable of resolving the memory bounded problem in traditional computing architectures. Emerging PIM hardware boasts advantages such as high bandwidth, high storage capacity, high parallelism, and high energy efficiency, yet it also suffers from limitations including weak computing capabilities and significant 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 hardware-software co-design solution have become subjects warranting in-depth research.

This thesis investigates the hardware-software co-design solution for GEMV based on Processing-in-Memory technology: 1) On the PIM hardware, a matrix-vector multiplication algorithm based on lookup tables is devised. Lookup tables are employed to eliminate complex computations, and memory access optimizations are implemented for the multi-level storage structure within the PIM hardware. This encompasses the lookup table blocking algorithm to reduce the number of memory accesses via DMA and the rearrangement of matrix rows and columns to minimize cache access and en-

hance data locality in registers. 2) On the simulator, to address the computational bottlenecks of software algorithms on real hardware, special processing unit and instructions are designed based on the cycle-accurate PIM simulator to enhance its computing capacity. This includes the fusion of table lookup and addition instructions for efficient and rapid query of products and accumulation, as well as the design of vector instructions based on lookup tables for vectorized lookup and memory access to boost computing efficiency. Finally, detailed tests were carried out on the aforementioned algorithms on three hardware platforms, namely CPU, GPU, and PIM hardware, including tests on the total throughput and energy efficiency of the GEMV operator. The experimental results indicate that the throughput of the GEMV operator on the PIM hardware amounts to 355 GOPS, approximately reaching 93% of the theoretical performance. It is 14.7 times that of the CPU platform, and the energy efficiency is approximately 8.6 times that of the CPU platform and 1.13 times that of the GPU platform.

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

目录

图目录

表目录

第 1 章 绪论

1.1 研究背景和意义

近些年来，大语言模型（Large Language Models，LLMs）在各个领域的成功应用，极大地推动了人工智能的发展。以 OpenAI 的 GPT（Generative Pre-trained Transformer）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模型特殊结构带来的推理瓶颈。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，已成为学术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 系列等）均为 Decoder-Only 架构，即只使用 Transformer 模型[1]中的解码器（Decoder）做生成[2, 3]。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，主要执行通用矩阵矩阵乘（Generalized Matrix Multiplication，GEMM）运算；2）解码阶段（Decoding），该阶段会逐 token 地计算和生成，主要执行通用矩阵向量乘（Generalized Matrix Vector Multiplication，GEMV）运算。

在“GPT 式”大模型的整个推理过程中，解码阶段占据主导地位[4]，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%[5]。这使得 GEMV 算子成为大模型推理的主要性能瓶颈，GEMV 算子的性能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，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，在常用

的计算密集型硬件如 GPU 上执行该算子的硬件利用率低，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 GEMV 操作合并成为 GEMM 操作以提高数据利用率[6]，但是在边端场景下，用户数量十分有限，batch 大小绝大多数情况为 1，这时 GEMV 往往称为系统瓶颈[7]。

近存计算（Processing-in-Memory，PIM）是一种新兴的计算范式，其秉持以数据为中心的思想，将计算置于存储侧，即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[8, 9, 10, 11, 12]，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 16 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软件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

大模型由最初的 Transformer[1]发展而来，经过几次的参数膨胀[2, 13, 14]量变引起质变具有了非常强的通用智能，广泛应用于智能客服、文本生成、金融、电商、教育和医疗等等众多领域[15]。参数膨胀引起了大模型的部署推理成本极大增高，边端用户难以以合理的速度本地推理即使是参数量较小的模型（7B）。这个时候许多大模型推理加速方法应运而生[16]，其中模型量化能够有效地缓解大模型推理的内存瓶颈[17]。模型量化按照量化的时机可以分为量化感知训练、量化感知微调和训练后量化。按照量化的数制映射方式可以分为线性量化和非

线性量化，其中以训练后线性量化方式的成本最为低廉和简单。这之中有许多出名的工作被提出，LLM.int8[18]通过矩阵离群值分解以较低的精度损失做到了8bit 权重 8bit 激活量化（W8A8 量化）。SmoothQuant[19]引入一种数学上的等价变换平滑了 W8A8 的激活量化和权重量化的难度。AWQ[20]作为一种仅权重量化，同样基于激活值分布对权重矩阵做分解实现了 W4 的量化。GPTQ[21]通过机器学习的方式逐层量化大模型，使用少量数据集校准，能够将权重量化到惊人的 3-4bit。ZeroQuant 系列[22, 23, 24]工作引入了多种推理加速手段包括知识蒸馏、低秩补偿来缓解 W8A8 甚至 W8A4 的量化带来的误差，并提出浮点数量化优化定点数量化的观点。

另一个能够显著加速大模型推理速度的方式是提升其推理过程中基本算子 GEMV 的性能[4]。GEMV 相比 GEMM 计算偏流式，计算强度和数据重用性都没那么高，其中，诸如 Strassen 算法[25]等算法层面的优化在现代计算机系统上的实现并不一定高效。有效的 GEMV 优化都是在计算机系统层面，结合计算硬件合理布局数据减少访存的优化[26]，常见的方法包括分块、数据打包、寄存器优化等等。

近存计算能够有效缓解内存瓶颈的应用。早在上世纪七十年代左右，存内计算悄然萌芽[27, 28]。上世纪九十年代，Wulf 等人系统性地定义了内存墙（Memory Wall）[29]，说明了 CPU 和内存速度不匹配的问题。有部分学者提出近存计算（PIM，Processing in Memory），希望在存储测引入计算减少数据的搬移以缓解内存墙的问题。RowClone[30]工作通过在 DRAM 的 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。Seshadri 等人[31, 32]提出一系列工作，通过简单修改内存单元的电路实现快速批量的逻辑计算。

在本世纪 10 年代，3D 堆叠架构技术的突破使得近存计算技术再度火热。与此同时由于 AI 技术的大火，有许多专用于神经网络的 PIM 芯片被推出。三星的 HBM-PIM[8]为 HBM（High Bandwidth Memroy）的每个内存 bank 配备了专门用于 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。三星的另一个产品 AxDIMM[9]将 DRAM 芯片和 FPGA 处理单元集成到一块有着 DDR4 标准接口的主板上，用于加速向量嵌入查找（embedding lookup）。海力士的 AiM[10]基于 GDDR6 内存，同样每个 bank 集成 PIM 单元用于神经网络的计算。国内的阿里也推出过近存计算产品[11]用于加速 AI 任务。

近几年，UPMEM 作为以第一款可以商用的近存计算处理器产品[12]，大受研究者青睐，其本身是一条有着标准 DDR4 接口的内存插块（Dual In-line Mem-

ory Module，DIMM），可以像正常的内存条一样插在 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 个内存处理单元（Dram Processing Unit，DPU）。每个 DPU 都拥有一个 14 级流水的 RISC 处理器，拥有 16 个线程。同时每个 DPU 拥有二级存储结构，包括 64KB 的 SRAM（称为 WRAM）和 64MB 的 DRAM（称为 MRAM）。

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[33, 34, 35, 36, 37, 38, 39, 40]、生物基因[41, 42, 43, 44, 45]以及人工智能[46, 47, 48, 49, 50, 51, 52]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[46]，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[47]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[48]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[50]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。

1.3 研究内容和创新点

结合研究背景和国内外研究现状不难看出，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 架构的高内存带宽优势，如何简化和避免复杂的计算是本研究课题的主要挑战。本课题主要的研究内容如图1.1所示，主要分为理论基础、实践优化和结果总结三个部分。理论基础部分主要围绕大模型推理加速和近存计算两大主题梳理基本概念，查阅相关文献，对研究现状进行分析，概括出 GEMV 的软件特点和 UPMEM 近存硬件的特性，为后续工作打下理论基础。实践优化部分由软硬件

协同优化方案组成。软件层面在商用近存计算硬件 UPMEM 上设计相关算法并优化，硬件方面基于近存计算模拟器，针对软件层面的优化瓶颈修改硬件以提升性能。结果总结部分主要是设计实验对上面做出的优化算法进行测试并分析。我们将从四个维度和三个硬件平台上（CPU，GPU，UPMEM）设计实验并分析，包括 1）矩阵向量乘算子的总吞吐；2）矩阵向量乘算子的能效比；3）算子执行时间的瓶颈分析；4）扩展性实验，分别测试算法的多线程扩展性和矩阵尺寸扩展性。基于上面的实验分析结果，总结优化效果，说明本次研究的不足并对未来进行展望。

基于上面的研究内容，概括本文的主要创新点如下：

1） 通过文献阅读充分了解了 UPMEM 硬件的特点，设计基于查找表的矩阵向量乘算法，分别对两级存储结构（WRAM、MRAM）做出了优化：提出了针对多级存储访存优化的查找表分块算法 LUT-M，通过分块载入查找表到缓存（WRAM）减少 DMA 访问 MRAM 次数增强 WRAM 的数据局部性；分别提出了针对缓存（WRAM）访存优化的矩阵行列重排算法，行重排算法 LUT-W-R 对矩阵的行进行重排减少 GEMV 中结果向量的读写次数从而减少对 WRAM 的访存。列重排算法 LUT-W-C 对矩阵进行列重排减少矩阵每一行计算中对查找表的访问从而减少 WRAM 的访问。这两种算法都减少数据从寄存器流出，增强寄存器的数据局部性，提升了 GEMV 算子的性能。

2） 通过分析基于查找表的矩阵向量乘算法的性能瓶颈，基于 UPMEM 的周期精确模拟器 PIMulator 修改增强 UPMEM 硬件：设计硬件单元增加了对融合查表加法指令的支持，主要用于优化 UPMEM 在进行查表时的内存地址计算并融合加法到单条指令中，减少上述算法在每次做查表加法运算时的指令数目；设计增加向量单元以支持向量指令，包括向量加法、移位和重排，设计基于向量指令的查表矩阵向量乘算法 LUT-SIMD，通过向量重排指令实现向量化查表，大大提升硬件计算能力和访存粒度，提升算法吞吐。

3） 对前两点的软硬协同优化设计了详细的基本测试，在 CPU 和 GPU 以及 UPMEM 三个硬件平台上，从四个维度进行实验，包括：测试各种优化算法下 GEMV 算子的总吞吐分析算子的绝对性能；测试三个平台下算子的能效比；测试各个算法优化下的执行时间的 IPC 和内存带宽利用率分析目

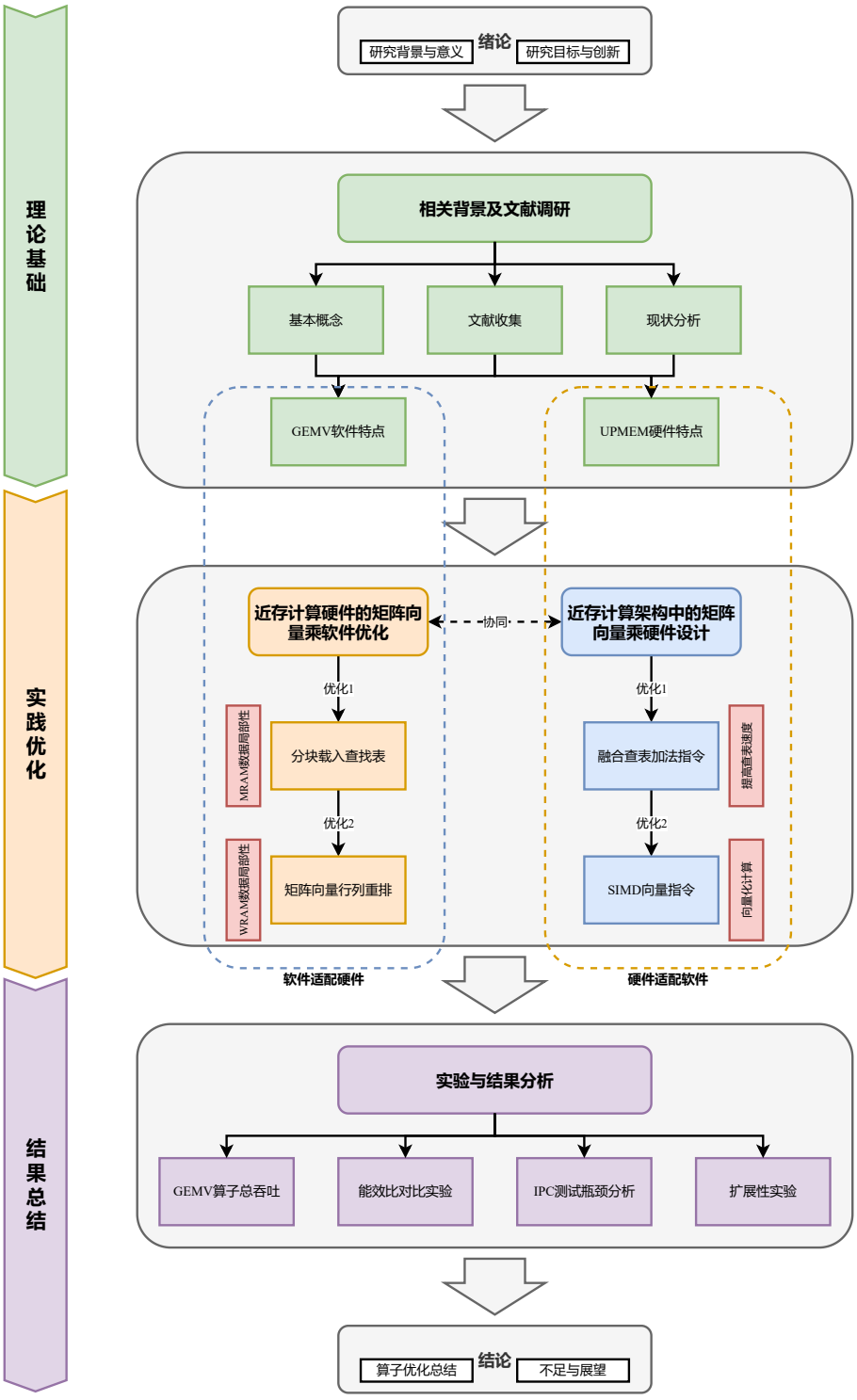

理论基础
实践优化
结果总结
绪论
研究背景与意义
研究目标与创新
相关背景及文献调研
基本概念
文献收集
现状分析
GEMV软件特点
UPMEM硬件特点
近存计算硬件的矩阵向量乘软件优化
协同
近存计算架构中的矩阵向量乘硬件设计
优化1
分块载入查找表
MRAM数据局部性
优化2
矩阵向量行列重排
WRAM数据局部性
优化1
融合查表加法指令
提高查表速度
优化2
SIMD向量指令
向量化计算
软件适配硬件
硬件适配软件
实验与结果分析
GEMV算子总吞吐
能效比对比实验
IPC测试瓶颈分析
扩展性实验
结论
算子优化总结
不足与展望

图 1.1 研究内容路线图

前算子的性能瓶颈；最后进行扩展性测试，包括多线程扩展性测试和矩阵尺寸测试分析 UPMEM 应对不同尺寸矩阵的通用性。

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 章是基于近存计算硬件的矩阵向量乘软件优化。该章节基于 UPMEM 硬件平台对 GEMV 算子提出了相关软件的设计和优化算法，基础算法是针对近存硬件的多级存储的局部性，基于二元计算查找表设计的算法。此后更进一步对硬件的缓存做出局部性优化，提出行重排和列重排的优化算法。

第 4 章是近存计算架构中的矩阵向量乘硬件设计。该章节首先分析软件算法的性能瓶颈，在此基础上对近存硬件进行进一步的设计，包括增加融合查表加法指令加速查表，以及使用向量单元向量化访存和计算加速矩阵向量乘。最终选取了 UPMEM 的周期精确模拟器 PIMulator 实现上述对 UPMEM 的硬件改动。

第 5 章是实验结果与分析。该章节主要从四个维度设计了测试，包括不同硬件平台上算子总吞吐，不同硬件平台的能效比，算子的性能瓶颈分析，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。该章节将对全文内容进行总结。首先对全文研究结果进行总结。接着，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。

第 2 章 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 大语言模型凭借惊人的通用智能被应用到了各行各业，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 [1]。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[53]；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[54]。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。

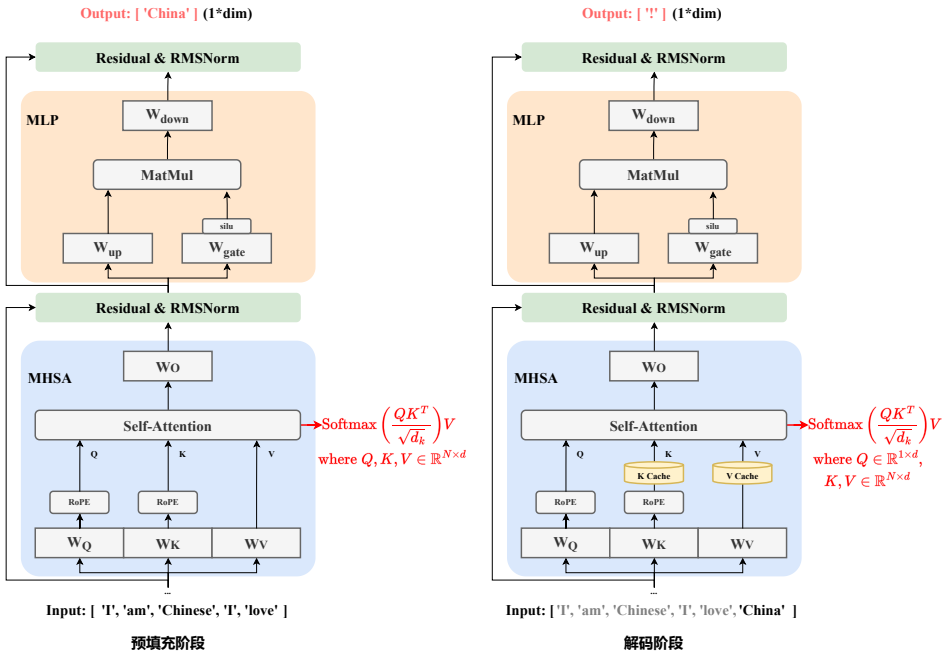


图 2.1 现代主流大模型推理的两个阶段，重绘自文章[16]

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[2]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[13]，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就可以达到非常好的效果。到了 GPT-3[14]，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。因 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[55]和少样本（Few Shot）微调能力[56]，大模型的训练数据和参数规模不断增长（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，常见的边端硬件几乎无法以可以接受的速度部署推理参数量较小的 70 亿参数模型。

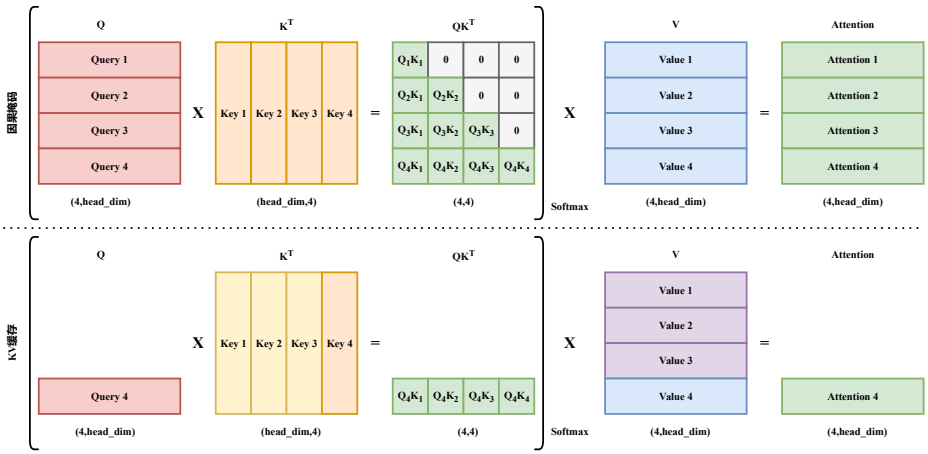


图 2.2 基于因果掩码和 KV 缓存的自注意力机制

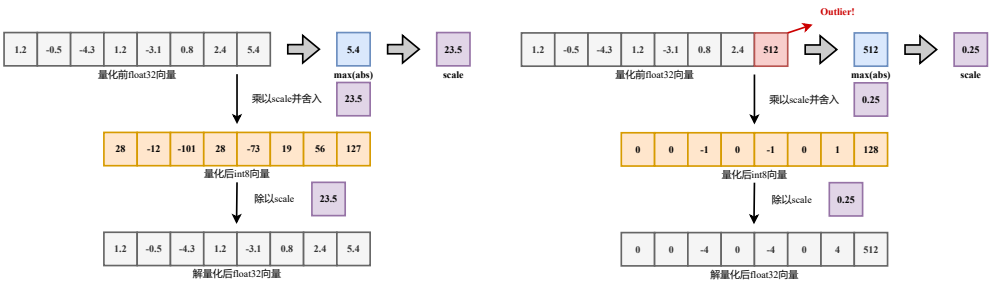
要想了解大模型在常用硬件平台上的推理瓶颈，就需要先了解大模型的结构，从开源的大模型入手是最佳选择。其中使用最广微调性能最高的开源大模型当属 Facebook 推出的 Llama 系列。Llama 系列的模型架构都秉承类似的结构，以 Llama2-7B[3]为例，模型一般由多层 Decoder 组成，每层 Decoder 分为多头注意力（Multi-Head Self-Attention，MHSA）和前馈神经网络（Feed-Forward Neural Network，FFN）两个主要部分。在 LLM 进行推理的时候，分为两个阶段：用户输入提示词进入网络到模型生成第一个 token 的阶段称作预填充阶段（The Prefilling Stage），在生成了第一个 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 GEMM，而在解码阶段执行的是 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。

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[4]，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[5]。这使得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、知识蒸馏、内存管理和批处理等技术[16]，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位宽参数（float32）通过一系列措施（目标是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。



(a) 正常对称量化　　(b) 含有离群值的量化

图 2.3 32 位浮点数向量对称量化为 8 位定点数

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[18]，该方法的主要思路在于离群值往往分布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[19]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各个通道的缩放因子 scale，将原激活除以对应的缩放因子作为新的激活，原矩阵的每个通道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）[20]，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 PTQ 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[21]本身的核心思想是对 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

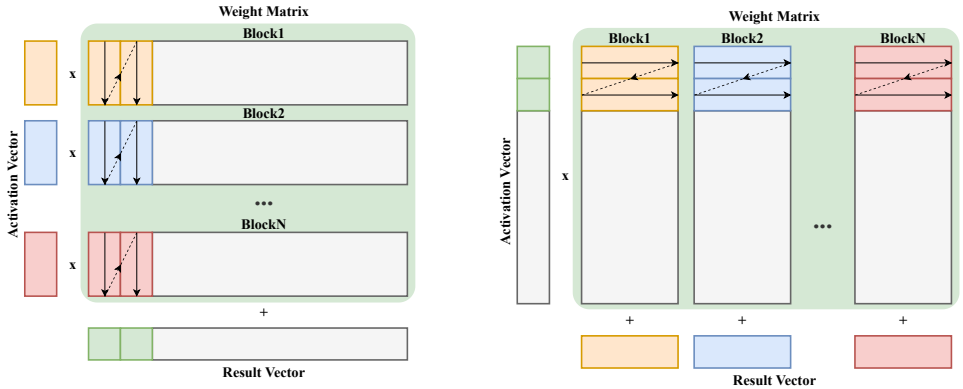
学习的方式进行优化，基于此前的 OBQ 工作[57]，对训练算法做了近似和加速，能够达到 3-4bit 的超低精度量化。Yao 等人开展了一系列 LLM 上的量化工作 ZeroQuant 系列[22, 23, 24]。ZeroQuant-V1 主要是针对 GPU 硬件构建了强大的推理后端并使用逐层知识蒸馏缓解量化带来的精度下降问题[22]；ZeroQuant-V2 则是针对常见的不同的 PTQ 方法进行了全面的分析，并提出了一种低秩补偿的技术来缓解量化带来的误差[23]；ZeroQuantFP 基于 GPTQ 的量化和低秩补偿，重点探索来浮点数据格式对于量化的影响，得出 float8 激活优于 int8、float8 权重与 int8 权重相当、float4 权重优于 int4 权重的结论[24]。

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 中左矩阵列数为 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）中最为基本的两个算子，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[25]，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 的优化；另一种，也是研究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2.4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（Activation Vector）和矩阵（Weight Matrix）的一行或一列的对应元素乘积之和，结果是最终结果向量（Result Vector）的一个元素。如图2.4(b)，GEMV 的外积可以定义为向量（Activation Vector）的某个元素与矩阵（Weight Matrix）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（Result Vector）。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。如图2.4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 矩阵向量乘基本优化方法

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。除此之外有许多相关硬件被设计提出对 GEMV 的加速提供了支持，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[27, 28]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[29]。许多学者围绕该问题提出了不同的解决方案，其中，有部分学者提出近存计算（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]被提出，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[30]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[31, 32]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高昂 PIM 型内存的迫切需求[62]。

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[63]。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。此时近存计算被重新提出，其在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[64]提出 Tesseract 使用 3D 堆叠技术加速大规模的图处理。

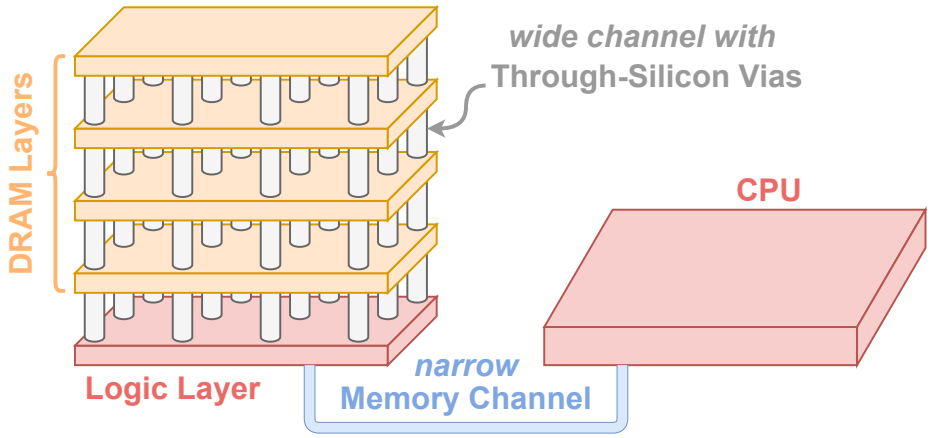


图 2.5 3D 堆叠内存结构，重绘自文章[65]

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[8]，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[9]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[10]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[11]，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[12]，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 DIMM 插块，可以像正常的内存条一样插在 Intel CPU 的服务器上。每个双路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）。如图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（Dram Processing Unit）独占其中一个内存 bank。因此一台服务器能够拥有 $20 \times 2 \times 8 \times 8 = 2560$ 个 DPU 并行处理任务。

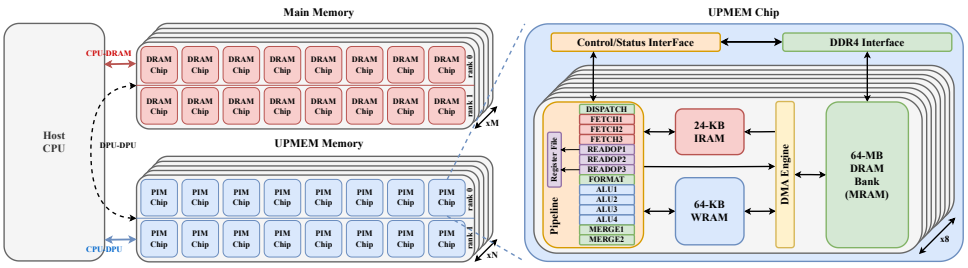


图 2.6 UPMEM 硬件架构，重绘自文章[66]

每个 DPU 拥有一个工作频率在 400MHz 的标量顺序（Scalar In-order）多线程的 RISC 核心，拥有 16 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[12]，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 16 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（缓存），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。具体参数可以看表格2.1。

有许多工作对 UPMEM 的硬件特征做了系统且全面的测试[66, 67, 68, 69, 70]，其中，Gómez-Luna 等人[66]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 16 个线程进行更加细粒度的控制，虽然大多数常规

表 2.1 UPMEM 基本参数

名称	参数
Frequency	400 MHz
Register width	32 -bit (64 -bit support)
DPU-MRAM/DRAM Bandwidth	1 GB/s
Threads	16
Main RAM (MRAM/DRAM)	64 MB
Instruction RAM (IRAM)	24 kB (4,096 × 48 -bit encoded instructions)
Work RAM	64 kB

测试集（Benchmark）中，超过 11 个线程并发就会让核心达到饱和。按照 11 个线程来计算，整个 PIM 系统的并发线程数量能够达到近 3 万。

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[66]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[66]，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。[66]。

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大量工作。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[71]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[72]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[73]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[74]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[75]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[76]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[41, 42, 43, 44, 45]，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[33]将 skyline 算子卸载到 UPMEM。清华的 Kang 等人做了一系列的工作[34, 35, 36]将数据库常见的索引查询如跳表、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[37, 38]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[39]将查询计划的优化也卸载到了 UPMEM 上。Lim 等人[40]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[46]将 UPMEM 用于卸载巨型嵌入标的嵌入查找（Embedding Table Lookup）任务，加

速效果尤为明显。Gómez-Luna 等人[47]将机器学习中经典的模型和算法卸载到了 UPMEM 上，包括线性回归、逻辑回归、决策树、K 均值聚类，并对其分别测试了准确性、性能和扩展性，结果表明 UPMEM 相对于 CPU 有巨大提升，与 GPU 几乎持平。Das 等人[48]在 UPMEM 上分别卸载了嵌入二值神经网络（Embedded Binary Neural Network，eBNN）和高度量化的卷积神经网络 YOLOv3，使用查找表消除浮点乘除计算，这种低 bit 量化在小模型上表现优异，但是不一定适用于大模型。Giannoula 等人[49]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。PIM-DL[50]使用 UPMEM 推理 Bert，使用聚类算法构建向量乘以矩阵的查找表，通过对输入向量切分子空间以对查找表进行降维，从而消除 GEMV 算子为最近邻查找和向量加法，提高了计算效率。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[51]提出 SwiftRL 以解决强化学习（Reinforcement 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[52]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

本章首先详细地对大模型推理加速的技术和论文做了详尽介绍，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介绍，并阐述了相关优化方法。本章还主要介绍了近存计算研究现状。首先梳理的近存计算的发展历程，包括近存计算的产生原因、发展和现状。然后着重介绍了第一款可商用的近存计算的硬件 UPMEM，包括其硬件架构和软件栈，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。

第 3 章 基于近存计算硬件的矩阵向量乘软件优化

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[66]，因此需要考虑量化。在上一章提到，浮点量化格式往往优于定点量化[24]；同时测试表明 UPMEM 的访存较计算更快，计算访存比仅有 1:4[66]，因此非常适合用访存换计算。基于上面的两点我们使用基于 FP8 数据格式[77]的查找表算法。由于 FP8/FP4 的相关量化工作已经非常成熟了，其中 FP8 的量化效果非常好，使用数据集校准时推理困惑度与使用 FP16 相比差距不到 1%，完全可以接受（相对的 FP4 量化要稍差一些）[77, 24, 78, 79, 80]。因此本文将不讨论具体的量化方案而聚焦于基于 FP8 数据格式的矩阵向量乘算子的设计：GEMV 内核的输入为 FP8 数据格式的激活向量和的权重方阵，最终计算得到的结果为 FP8 数据格式的结果向量。此外为了和 Llama2-7B 保持一致，没有特别说明，本章中涉及到的权重矩阵的维度皆为 4096×4096，同时本章所有讨论的优化都是建立在单个 DPU 的推理优化。

本章将基于量化后 FP8 数据格式的查找表算法优化 GEMV 算子，主要分两个层次进行优化，首先优化 WRAM 的数据局部性，减少 MRAM 访存；其次优化寄存器的数据局部性，减少 WRAM 的访存。

3.1 基于多级存储的查找表分块算法

UPMEM 的 MRAM 访问是通过 DMA 引擎，速度较慢，并且只在大批量数据传输时带宽较高，针对该硬件特性，本节针对存储层级 MRAM 做访存优化，提高 WRAM 的数据局部性。

3.1.1 矩阵向量乘查找表基础

查找表（Look Up Table，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。

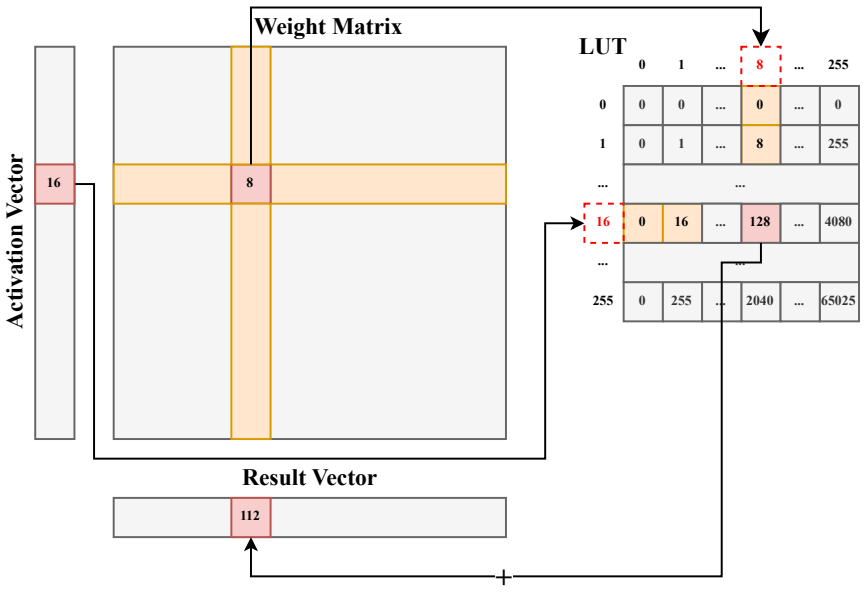


图 3.1 基于查找表的矩阵向量乘

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 LUT，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（Activation Vector）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（Result Vector）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 256×256 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 的乘法拆成多步 4bit 的乘法、加法和移位操作，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。

3.1.2 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 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（Activation Vector）和结果向量（Result Vector）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。

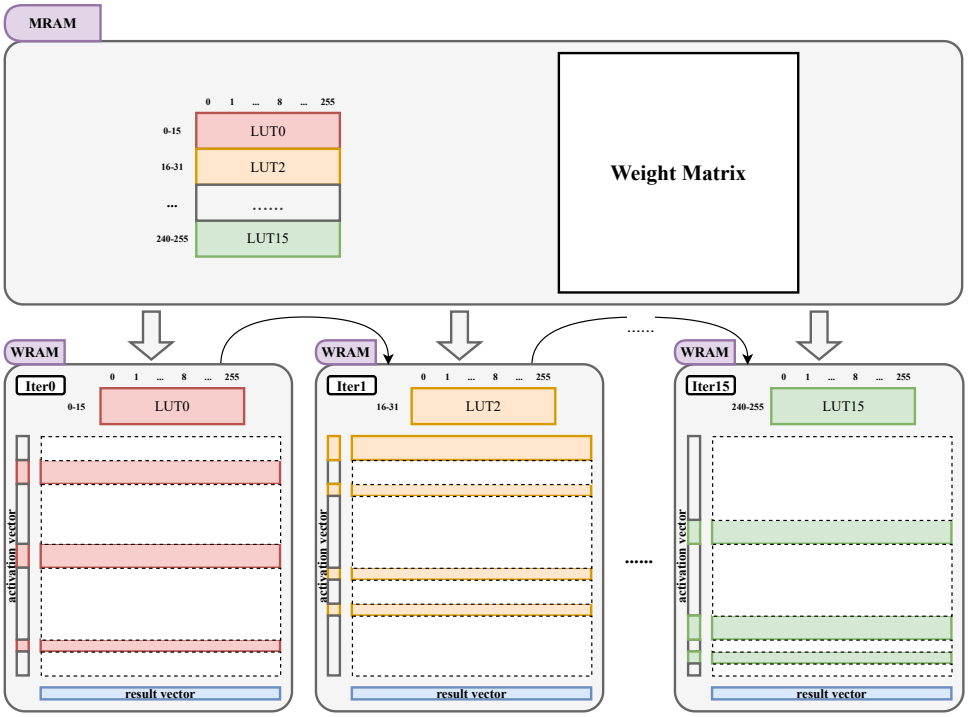


图 3.2 分块载入查找表卸载 GEMV 中的乘法

该算法需要进行子查找表个数次迭代，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行索引值区间内，对于满足要求的元素，将该元素所对应的权重矩阵（Weight Matrix）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，实现了非常良好的 WRAM 数据局部性。

3.1.3 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，但是在计算加法时无法简单套用类似的思路：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[77]，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[12]，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。

分析量化大模型的推理流程，以 W8A8 的量化为例，在进行线性层推理时，会先将 8bit 量化后权重和激活解量化到 16bit，再调用矩阵乘法库中的 16bit 矩阵向量乘算子，得到 16bit 的结果，再转换回 8bit 的结果向量传递给下一个算子。

受此启发，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 FP8 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 $max(e-1\square 0)$ 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 MapLUT，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 MapLUT 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。

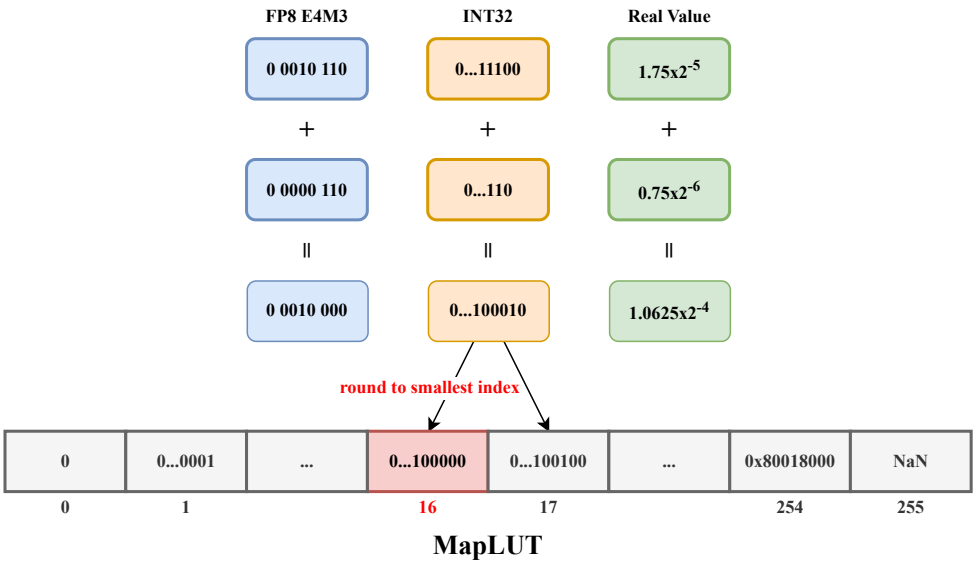


图 3.3 数制映射表卸载 GEMV 中的加法

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（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，同时端到端的误差可以由量化方法保证）。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 IEEE754 标准的向零舍入（Round toward Zero）是同一种舍入规则。

3.1.4 算法小结

算法 3.1 基于多级存储的查找表分块算法（LUT-M）

输入： $Vector[M], Matrix[M][N], LUT[256][256], MapLUT[256]$;

输出： $Result\_8[N]$;

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

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，32bit 的结果向量的大小为 16KB，转成对应的 FP8 向量需要占用 4KB 的空间，但是可以直接覆盖写入激活向量以节省空间，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 41KB，剩余 23KB 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。值得一提的是，本算法可以适用于不同量化位宽的数据格式，包括 FP8 量化中指数位数小于 5 的所有浮点格式以及低于 8bit 数据位宽的所有量化格式。

3.2 基于缓存的矩阵行列重排算法

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[66]。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。

3.2.1 矩阵行重排

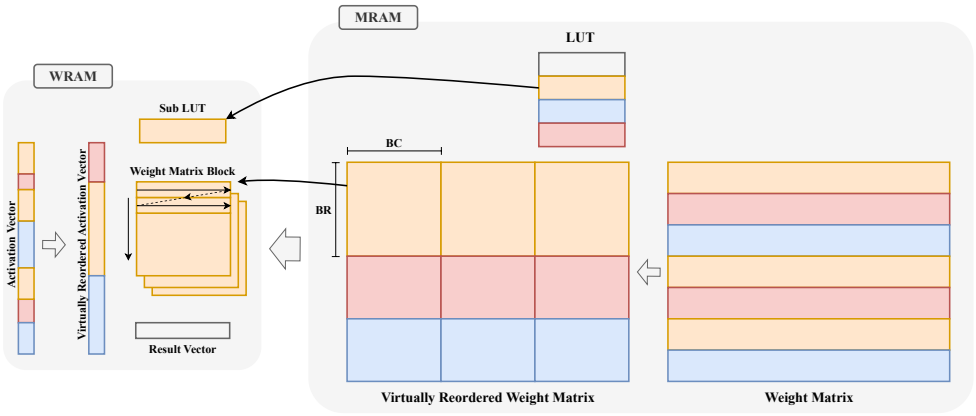


图 3.4 矩阵行重排计算示意图

针对算法3.1，可以观察到每完成权重矩阵的一行计算就要读写各一次结果向量，有很大的开销，这里基于此前提到的算法做出修改如图3.4所示，此前由于

考虑权重矩阵一行的向量会较长无法同时载入多行，在这里可以依据激活向量的值的分布对权重矩阵进行分块，划分成 BR（Block Row）行 BC（Block Column）列的子矩阵。

对于子矩阵，可以一次性全部载入，进而使用 GEMV 的内积计算方式，将结果向量某个元素的值缓存在寄存器中，完成了子矩阵的一列计算后再写入内存。注意这里不改变矩阵的行列主存格式，依然按照行主存，原因是这里不真实地对矩阵的行进行重排，仅仅是在载入时挑选满足需求的子矩阵行，称之为虚拟重排（Virtually Reordered）。因此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，这里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，结果加到结果向量对应位置中。每次载入子查找表，就以 BR 行的粒度载入 BC 列的子矩阵，当该 BR 行的所有子矩阵完成了计算后，取寻找下一个 BR 行的子矩阵直至结束激活向量的所有访问。

具体如算法3.2所示，设置 M 和 N 都为 4096，且一次性载入的子矩阵大小上限为 16KB，可以将 BR 和 BC 都设置为 128，此时新增的数据结构有子矩阵 SubMat，占用空间 16KB；有 Index 数组保存行索引，占用 256B；有 Offset 数组用于保存该行查找表的偏移，占用 1KB 空间，因此一共多占用了近 13KB 的 WRAM 的空间，相比于算法3.1剩余 8KB 的空间，分配给各个线程的堆栈足够了。在使用行重排之前，每一行矩阵的计算都要读写各一次 4096 维度的结果向量。而使用了行重排之后，假设每次子矩阵都是 128 行，那么每 128 行才会读写各一次 4096 维度的向量，单就读写结果向量的提升来说，是 128 倍。当然每次处理矩阵元素时，不仅仅只做读写向量的操作，还包括读取矩阵元素、查表、相加以及其中隐含的算术逻辑操作，实际提升需要进一步测试。

3.2.2 矩阵列重排

同样针对算法3.1，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，然而由于矩阵权重的每个元素为 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，但是实际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。

算法 3.2 行重排的矩阵向量乘算法（LUT-W-R）

输入： $Vector[M], Matrix[M][N], LUT[256][256], MapLUT[256], BR, BC$;

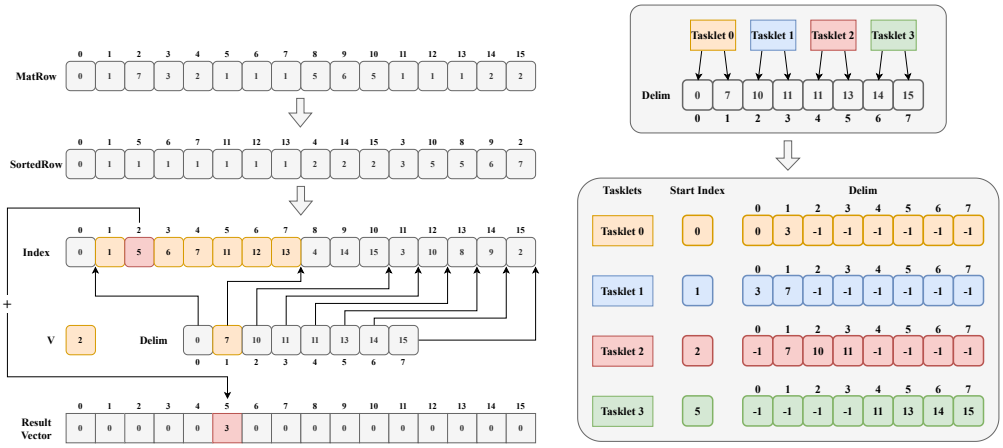
输出： $Result\_8[N]$;

define $SubLUT[16][256],\ SubMat[BR][BC],\ Result\_32[N]$
for $i \leftarrow 0$ **to** 15 **do**
  mram_read$(SubLUT, LUT[16i \cdots 16i+15], 16K)$ ▷ parallel read
  define $Offset[BR],\ Index[BR],\ vidx = 0$
  while $vidx < M$ **do**
    $row \leftarrow 0$
    while $row < BR$ **and** $vidx < M$ **do** ▷ parallel in BR for each tasklet
      if $Vector[vidx]$ **in** $[16i, 16i+15]$ **then**
        $Index[row] \leftarrow vidx,\ row \leftarrow row + 1$
        $Offset[row-1] \leftarrow SubLUT[Vector[vidx]$&0xF$]$
      end if
      $vidx \leftarrow vidx + 1$
    end while
    for $j = 0$ **to** $N - 1$ **step** BC **do**
      for $k = 0$ **to** row $- 1$ **do** ▷ parallel in BR for each tasklet
        mram_read$(SubMat[k], \&Matrix[Index[k]][j], BC)$
      end for
      for $k = j$ **to** $j + BC - 1$ **do** ▷ parallel in BC for each tasklet
        $temp \leftarrow Result\_32[k]$
        for $l = 0$ **to** $row - 1$ **do**
          $temp \leftarrow temp + *(Offset[l] + SubMat[l][k] \times ele\_width)$
        end for
        $Result\_32[k] \leftarrow temp$
      end for
    end for
  end while
end for
$Result\_8 \leftarrow$ **BinarySearch**$(Result\_32, MapLUT)$ ▷ parallel in N
return $Result\_8$

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 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，计数完成的这 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（查找表的某一行），然后读取 Delim 数组可以知道，Index 数组的索引 $(0, 7]$ 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 长度平均分配给各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。

算法 3.3 列重排的矩阵向量乘算法（LUT-W-C）

输入： $Vector[M], DelimMat[M][TN][257], IndexMat[M][N], LUT[256][256], MapLUT[256]$;

输出： $Result\_8[N]$;

define $SubLUT[16][256],\ IndexRow[N],\ Delim[257],\ Result\_32[N]$
for $i \leftarrow 0$ **to** 15 **do**
    mram_read$(SubLUT, LUT[16i \cdots 16i+15], 16K)$ ▷ parallel read
    for $j \leftarrow 0$ **to** $M-1$ **do**
        if $Vector[j]$ **not in** $[16i, 16i+15]$ **then**
            continue
        end if
        mram_read$(IndexRow, IndexMat[j], 2N)$ ▷ parallel read
        mram_read$(Delim, DelimMat[j][TaskletId], 514)$
        $k \leftarrow Delim[0]$
        while $Delim[k] \neq -1$ **do**
            $product \leftarrow SubLUT[Vector[j]\&0xF][k]$
            for $l \leftarrow$ (**if** $k = 0$ **then** 0 **else** $Delim[k-1]+1$) **to** $Delim[k]$ **do**
                $Result[IndexRow[l]] \leftarrow Result[IndexRow[l]] + product$
            end for
            $k \leftarrow k+1$
        end while
    end for
end for
$Result\_8 \leftarrow$ **BinarySearch**$(Result\_32, MapLUT)$ ▷ parallel in N
return $Result\_8$

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3.3。

假使 M 和 N 都为 4096，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，当 TN（TaskletNum）为 16 时，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，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 WRAM reads 变为 4.5K WRAM reads，差不多有两倍的性能提升。和此前算法3.2类似，实际过程中查表操作的占比未知，提升需要进一步测试。

3.3 本章小结

本章主要是介绍了在近存计算 UPMEM 硬件上对 GEMV 算子的软件优化。首先简要提及了基于查找表的量化手段，介绍了查找表卸载二元算术运算基础。在此基础上提出针对多级存储访存优化的矩阵向量乘算法 LUT-M，该算法使用查找表的方法卸载了 FP8 数据格式的乘法和加法的二元运算，拥有非常好的 WRAM 数据局部性。随后，更进一步提出了针对缓存访存优化的矩阵向量乘算法，分别对矩阵进行行重排（LUT-W-R）和列重排（LUT-W-C），充分考虑 GEMV 计算中 WRAM 的访问模式，优化寄存器局部性，减少无意义的内存读写。

第 4 章 近存计算架构中的矩阵向量乘硬件设计

本章主要介绍借助近存计算模拟器的修改硬件，优化矩阵向量乘的设计。第一小节是对上一章提出的三个软件算法的性能分析。之后介绍新增融合查表加法指令对矩阵向量乘的优化手段。然后详细介绍通过增加向量单元和指令，向量化计算矩阵向量乘的设计和算法。最后说明上述设计实现的模拟器平台，简要介绍模拟器特性和硬件修改实现方式。

4.1 软件优化性能瓶颈分析

表 4.1 软件优化性能分析

算法	执行耗时	IPC	MBU
LUT-M	698.7 ms	96.96%	3.70%
LUT-W-C	467.6 ms	84.38%	11.66%
LUT-W-R	298.8 ms	76.97%	10.67%

在上一章提出了三种算法优化后，本小节对每种算法下的矩阵向量乘算子的在近存硬件上的执行情况进行了测试，分别测试其在特定规模和的矩阵和线程配置下的执行时间、周期数以及指令数。我们计算了不同算法的计算利用率（Instructions Per Cycle，IPC）和内存带宽利用率（MRAM Bandwidth Utilization，MBU）。这里 MBU 的计算公式为4.1，MRAMRead 是算子执行过程种从 MRAM 读取的数据（MB），ExecuteTime 是算子执行时间（秒），通过此前的工作知道理论带宽大概为 628MB/s[66]），因此可以算出利用率。矩阵仍然采用的是 4096×4096 的 FP8 格式，开启 16 个 Tasklets，在单一 DPU 上的测试如表4.1所示。

$$MBU = \frac{MRAM\ Read}{ExecuteTime \times 628} \times 100\% \tag{4.1}$$

可以看到，在三种算法中，从 LUT-M、LUT-W-C 再到 LUT-W-R，IPC 是下降的，相应的 MBU 是上升的，这说明我们在针对 WRAM 的优化中通过增加 MRAM 的访存次数来减少 WRAM 的访存，可以看到执行时间在逐渐降低证明这种优化有效：LUT-M 虽然 MRAM 的访问少但是执行时间长，通过额外访问 MRAM 获取一些辅助性数据或增加访问 MRAM 的次数提高寄存器的局部性可以有效的提升算子性能，是一种权衡（trade off）。

尽管如此，能观察到 IPC 的值始终在 70% 以上，相应地 MBU 的值在 10% 左右，这充分说明了 UPMEM 这个硬件平台的特性为计算瓶颈，与传统的冯诺依曼架构不同，UPMEM 的访存带宽很快几乎不会成为瓶颈。想要进一步提升算子性能，本文认为仅仅在软件层面做算法的优化十分有限，需要着手与 UPMEM 硬件结构做出优化，比如为 UPMEM 增添额外的计算单元和算术能力以增强 UPMEM 的计算强度。这里将不探讨简单的硬件参数修改诸如提高主频、增加浮点单元支持硬件浮点算术计算等等，我们将结合我们的软件算法特点（LUT）去设计硬件的微架构的更改。

4.2 查表和加法融合的指令设计

在传统 CPU 平台上编程时，如果需要访问数组的某个元素，需要拿到数组的首地址 addr 和元素索引 index，以及每个元素所占字节数为 width，通过计算 addr+index*width 得到真正的内存地址然后再去访存。这种乘加的操作在编译时会做优化，假如数组是连续访问的，那么会优化指令递增地址：只有第一次访问数组需要计算乘加，后面访问只需要在地址寄存器累加上 width 即可。但是在上述基于查找表的算法，数组的访问往往不是连续的，无法消除乘加操作，因此这里可以设计增加一系列指令专门用于访问数组（查找表），如图4.1所示，分别设计了适用于不同数据位宽的指令，如 LHSI（Lookup Halfword Signed by Index），该指令接受 dst 寄存器、base 寄存器、idx 寄存器和移位立即数 shift：dst 寄存器是目标寄存器，base 寄存器是数组的首地址，idx 寄存器是数组的索引，shift 指示的是元素宽度（2byte 宽度是移位 1 位），这条指令的操作就是将 idx 寄存器的值左移 shift 位，然后与 base 寄存器的值相加得到地址 addr 并访存 2 个字节，

将结果存入 dst 寄存器（符号位扩展）。这样数组的访问就可以通过这一条指令完成。

图 4.1 FLA 指令集设计

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。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。因此可以基于此前的查表指令，修改 UPMEM 硬件支持融合查表加法指令，如图4.1所示，FMLA（Fused Multiply Lookup Add）指令接受 add 寄存器，lut_base 寄存器，idx 寄存器和移位立即数 shift：add 寄存器用于存放累加结果，lut_base 寄存器存放的是查找表的首地址，idx 寄存则是查找表的索引，shift 同样指示的是元素宽度，这条指令的操作类似此前的查表指令，先取

得 lut_base[idx] 结果，与 add 寄存器相加并累加到 add 中。至于 FMAL（Fused Multiply Add Lookup）指令则是受 armv8 的 LDADDA 指令启发[1]，专门用于解决需要频繁读写结果向量的操作，其会从内存中读取某个数进行加法运算后再存回。这一系列的指令我们称之为 FLA（Fused Lookup Add）指令集。

```
;Result:r0, k:r2, SubLUT[Vector[j]&0xF]:r4, MatRow:r6

;MatRow
add r1, r6, r2
lbu r1, r1, 0
;SubLUT
lsl_add r3, r4, r1, 2
lw r3, r3, 0
;Result
lsl_add r5, r0, r2, 2
lw r7, r5, 0
;add
add r7, r7, r3
sw r5, 0, r7
```

(a) LUT-M 算法

```
;temp:r0, Offset:r2, l:r4, SubMat:r6, k:r8, ele_width:4

;Offset
lsl_add r1, r2, r4, 3
ld r1, r1, 0
;SubMat
lsl_add r3, r8, r4, 7
add r5, r6, r3
lbu r5, r5, 0
;addr
lsl_add r7, r1, r5, 2
lw r7, r7, 0
;add
add r0, r0, r7
```

(b) LUT-W-R 算法

```
;Result:r0, IndexRow:r2, l:r4, product: r6

;IndexRow
lsl_add r1, r2, r4, 1
lhu r1, r1, 0

;Result
lsl_add r3, r0, r1, 2
lw r5, r3, 0

;add
add r5, r5, r6
sw r3, 0, r5

```

(c) LUT-W-C 算法

```
;Result:r0, k:r2, SubLUT[Vector[j]&0xF]:r4, MatRow:r6

;MatRow
lbui r1, r6, r2
;SubLUT
lwi r3, r4, r1, 2
;add
fmal r0, r2, 2, r3

```

(d) LUT-M 算法优化

```
;temp:r0, Offset:r2, l:r4, SubMat:r6, k:r8, ele_width:4

;Offset
ldi r1, r2, r4, 3
;SubMat
lsl_add r3, r8, r4, 7
lwi r5, r6, r3, 0
;add
fmla r0, r1, r5, 2
```

(e) LUT-W-R 算法优化

```
;Result:r0, IndexRow:r2, l:r4, product: r6

;IndexRow
lhui r1, r2, r4, 1
;add
fmal r0, r1, 2, r6


```

(f) LUT-W-C 算法优化

图 4.2 不同 GEMV 算法查表累加语句汇编分析

对算法3.1、3.2、3.3中查表累加语句进行汇编分析：算法 LUT-M 对应的是`Result[k]+=SubLUT[Vector[j]&0xF][MatRow[k]]`,其编译指令序列如图4.2(a)所示；算法 LUT-W-C 对应`Result[IndexRow[l]]+=product`，其指令序列如图4.2(c)所示；算法 LUT-W-R 对应的是语句`temp+=*(Offset[l]+SubMat[l][k]*ele_width)`，其指令序列如图4.2(b)所示。每做一次数组的访问或者累加操作，至少需要两条甚至以上的指令，现在使用图4.1中所示的指令集对上述汇编代码进行重写，能够得到优化后的汇编代码如图4.2(d)、4.2(e)、4.2(f)所示，指令数目大大减少，由原先的 6 至 8 条指令减少到 2 至 4 条，由 2 到 3 倍的提升；由于最深层循环的循环次数是最多的，主要决定程序的性能，因此理论上使用融合查表加法指令能够对程序有 2 到 3 倍的性能提升。

4.3 基于查表的向量指令设计实现

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

[1]https://developer.arm.com/documentation/dui0801/g/A64-Data-Transfer-Instructions/

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之一，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 512bit，大大提升了数据处理的效率。

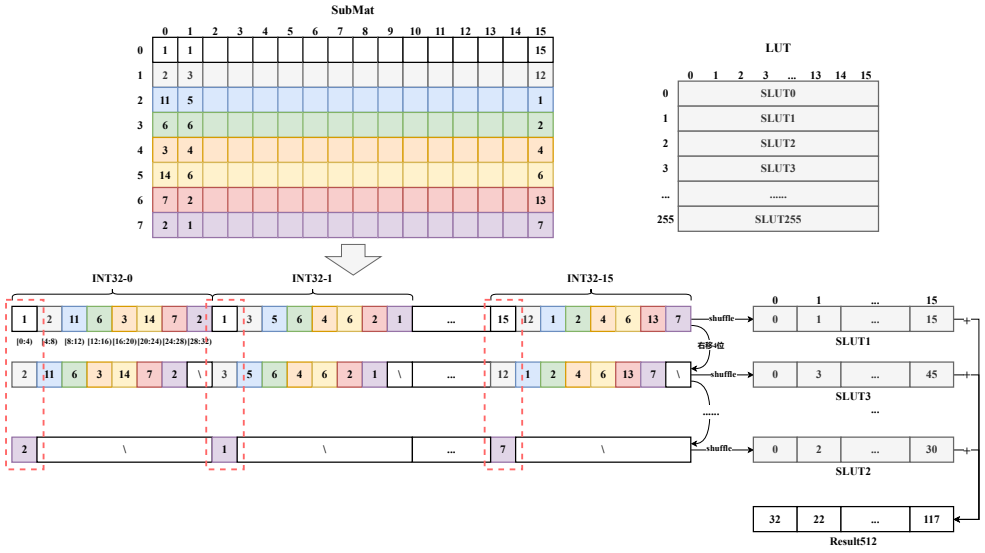


图 4.3 基于 SIMD 指令的矩阵构建和 GEMV 计算

对于更低 bit 量化，参考 Intel AVX 向量指令集增添 SIMD 指令进行加速：`__m512i _mm512_permutexvar_epi32(__m512i idx, __m512i a)`,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 位整型数组成，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 256×16 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，现以 Intel 的 AVX-512 向量为例，对于行主存的矩阵，现将其切分成 8×16 的子块 SubMat 如图4.3，对这个子块进行重排，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。

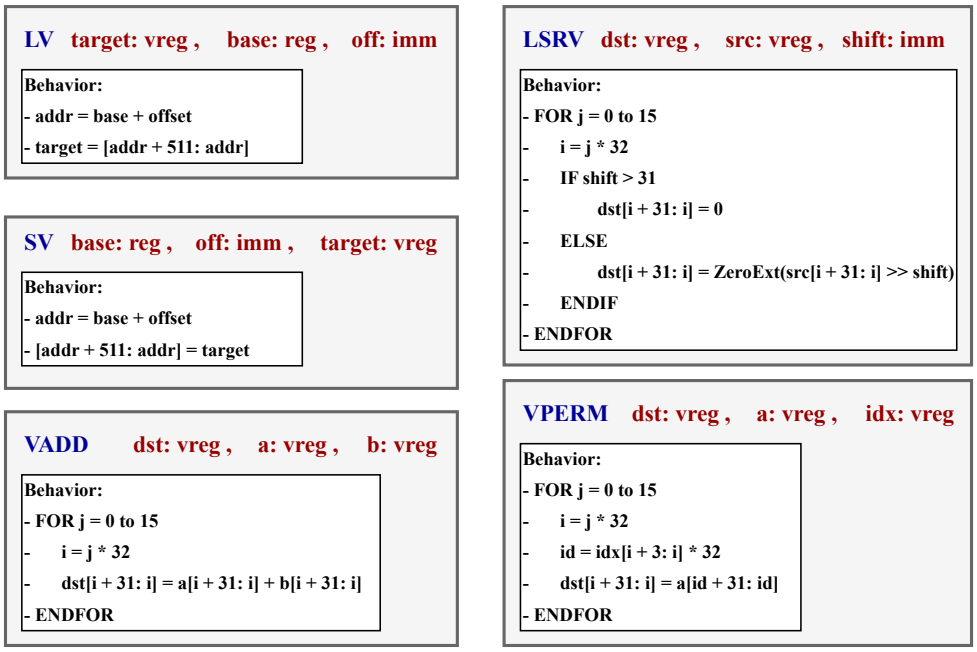


图 4.4 向量指令集设计

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。再基于 DPU 内部线程并行化，每个 tasklet 处理一列并均分所有列。具体到单个 AVX-512 向量，仍然以 IntelAVX 指令为例，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 到 AVX-512 向量 a 中（使用_mm512_loadu_si512指令载入寄存器），然后载入权重的 AVX-512 向量到 index 寄存器中，执行_mm512_permutexvar_epi32操作得到结果，使用_mm512_add_epi32指令将其加到 AVX-512 累加结果寄存器上。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同步开销。

算法 4.1 基于 SIMD 指令的矩阵向量乘算法（LUT-SIMD）

输入： $Vector[M], TransMat[N/16][M*8], LUT[256][16], MapLUT[256]$;

输出： $Result\_8[N]$;

1: **define** $MatBuffer[1024],\ Result\_32[N]$
2: **for** $i \leftarrow TaskletId$ **to** $N/16-1$ **step** $TaskletNum$ **do**
3: **for** $j \leftarrow 0$ **to** $M*8-1$ **step** 1024 **do**
4: **mram_read**$(MatBuffer, \&TransMat[i][j], 1024)$
5: **for** $k \leftarrow 0$ **to** 1023 **step** 64 **do**
6: **LV**$(A, \&MatBuffer[k], 64)$
7: **for** $l \leftarrow 0$ **to** 7 **do**
8: **LV**$(Index, LUT[Vector[(j+k)/8+l]], 0)$
9: **VPERM**$(Temp, A, Index)$
10: **VADD**$(Adder, Adder, Temp)$, **LSRV**$(A, A, 4)$
11: **end for**
12: **end for**
13: **end for**
14: **SV**$(\&Result\_32[i*16], 0, Adder)$
15: **end for**
16: $Result\_8 \leftarrow$ **BinarySearch**$(Result\_32, MapLUT)$ ▷ parallel in N
17: **return** $Result\_8$

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，如图4.4所示，使用 LV 和 SV 指令来从内存某处加载/存储 512bit 的向量到向量寄存器中。使用 LSRV 指令对 512bit 向量中的每个 32bit 数进行右移，右移的位数不超过 31 位。当然还要使用 VADD 执行两个 512bit 向量逐元素加法，以 VPERM 指令使用 index 向量对 a 向量进行重排。同时要高效完成上述工作，每个线程至少需要 4 个 512 向量寄存器，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。考虑到硬件实现，由于我们只是支持 SIMD 的部分计算，仅仅包括重排、加法和位移（向量的 Load/Store 比较简单），因此向量计算单元的电路设计不会特别复杂，同时由于 UPMEM 工作特性，线程数量大于 11 对于性能没有明显提升，存在大量空闲的寄存器（UPMEM 有 16 个物理线程每个线程都有 24 个 32bit 寄存器），因此有足够的空间留给向量寄存器堆。

给出基于 SIMD 向量指令的算法如算法4.1所示，这里卸载的是 $8bit \times 4bit$ 的乘法，因此查找表的维度是 256×16，刚好可以放进 WRAM 中且每一行都是一个 512bit 的向量。权重矩阵需要重排，假设 M 和 N 都为 4096，原本的维度是 4096×4096，使用 int8 存储数组的维度是 4096×2048，重排之后行缩小 8 倍，列扩大 8 倍变为 512×16384，再按照 512bit（64B）的粒度进行行列转置，维度就变成 256×32768。上述操作中最深层循环的操作是指令 LV、VPERM 和 VADD，每次执行这三条指令就可以一次性对 16 个数进行查表并累加，大大提升了性能。

4.4 基于近存模拟器的硬件设计实现

为了实现上述硬件的修改并验证性能，需要使用一个能够模仿 UPMEM 硬件计算的软件。PIMulator 是一个 UPMEM 的周期精确模拟器[70]，能够满足我们修改硬件的需求。它由两个关键组件组成，如图4.5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。

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[2]（Software Development Kit，SDK）提供的 LLVM[81]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

[2]https://sdk.upmem.com

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。

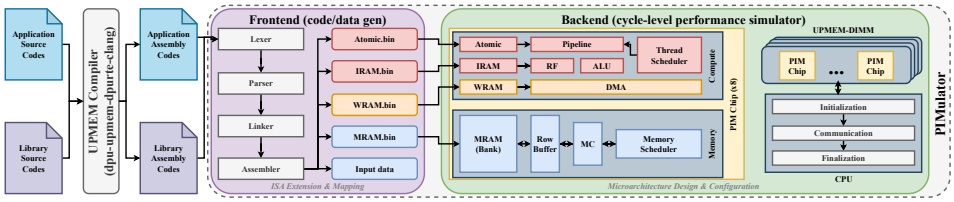


图 4.5 PIMulator 架构图，重绘自文章[70]

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[82]。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 有多个版本的实现[3]，在论文中实现的是前后端分离的单线程版，使用 Python 实现前端，C++ 做硬件模拟后端，没有多线程支持，平均模拟速率为 3KIPS（千条指令每秒），与 GPGPU-Sim 相当。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。

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。本文的硬件架构改动使用 PIMulator 的

[3]https://github.com/VIA-Research/uPIMulator

这种前后端分离架构实现，具体来说对于改动一的新增的融合查表加法单元，在定义好指令的格式后，会在前端直接修改 GEMV 算子的汇编代码并使用自定义的 FLA 指令集，然后修改前端汇编器，包括语法分析和词法分析的程序，使得前端程序能够正确识别我们新增的指令。与此同时，在后端的逻辑部分新增对应指令的逻辑计算，以函数的形式暴露给后端模拟仿真，就相当于增添了相应的硬件单元。对于改动二 SIMD 指令也是类似的流程，不过稍有不同之处在于 SIMD 指令需要自行添加向量寄存器并识别，这个需要在前后端的寄存文件中去定义向量寄存器。

4.5 本章小结

本章主要介绍了在近存计算模拟器上修改硬件加速矩阵向量乘的设计。首先对此前提出的软件算法进行了分析，分析算子的性能瓶颈在于商用硬件的计算性能上，阐明了修改硬件的原因和必要性。接着介绍了设计的融合查表加法指令集，通过对矩阵乘法汇编代码的分析和比较表明引入该融合查表加法指令会大幅减少指令数目并加速程序。然后是对 UPMEM 增加向量单元的设计，详细描述了权重矩阵的重排流程、SIMD 指令集的设计和基于向量指令的 GEMV 算法，通过向量化访存和计算能够极大提高程序效率。最后介绍了上述硬件改动的实现，包括模拟器平台 PIMulator 的介绍和修改方式。

第 5 章 实验结果与分析

本章节主要是对此前在 UPMEM 上进行软硬协同优化的 GEMV 算子的综合测试，第一小节介绍环境配置平台，第二小节重点在 UPMEM 近存计算硬件平台上进行算子综合测试，包括与其他常见的硬件计算平台进行对比，第三小节重点在 PIMulator 模拟器平台上测试硬件优化的效果。

5.1 环境配置介绍

5.1.1 硬件平台

本文实验的近存计算平台在 UPMEM 官方推荐的 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 $2 \times 5 \times 2 \times 128 = 2560$ 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。具体详细的硬件配置可以见表5.1。

表 5.1 硬件平台配置

硬件平台	制程	处理核心			内存		功耗
		核心数	工作频率	峰值性能	容量	总带宽	
Intel Xeon 6242 CPU	14 nm	16	2.8GHz	89.6 GFLOPS	255.9 GB	256 GB/s	150 W
NVIDIA A6000 GPU	8 nm	10752	1.41GHz	38.7 TFLOPS	48 GB	768 GB/s	300 W
UPMEM	2x nm	2560	400MHz	1024 GOPS	160 GB	1.7 TB/s	256 W

5.1.2 数据准备和矩阵尺寸选择

本文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。我们在主要的测试中选取的 GEMV 数据尺寸为 4096×1024，对应 Llama2-7B 推理过程中的线性层；在扩展性测试中我们将使用不同矩阵尺寸测试 DPU 的扩展性。在数据宽度方面，近存计算平台选择 FP8（E4M3）数据格式进行测试，在 CPU 和 GPU 平台选择 Float32 进行测试（精度由现有量化方法工作保证）。每个 DPU 使用 4096×1024 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 $4096 \times 1024 \times 2560$，假设使用 FP32 的数据格式，矩阵所占空间最大为 $4096 \times 1024 \times 2560 \times 4Byte = 40GB$，上述硬件平台足以存储。

5.1.3 基线设置

对于商用近数计算硬件平台的测试，我们设置基线（baseline）分别为 CPU、GPU 和 UPMEM 平台的 Float32 矩阵向量乘。PIM 平台使用 UPMEM SDK（版本 2024.1.0）编译在 UPMEM-DIMM 执行；CPU 平台使用英特尔数学核心函数

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

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

库（Intel Math Kernel Library，MKL）[83]，它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（包含 OpenMP 以及 SIMD 指令），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。对于近存计算模拟器平台的测试，我们主要测试硬件修改对算子的提升，基线设置为第三章提出的三个算法 LUT-M3.1、LUT-W-R3.2和 LUT-W-C3.3以及朴素 GEMV 内积 LUT-FP4。

5.2 基于近存计算平台的软件优化测试与分析

5.2.1 总吞吐对比测试

我们定义 GEMV 算子的吞吐为每秒操作运算次数 (Operations Per Second, OPS) 来衡量，对于矩阵尺寸为 $M \times N$ 的 GEMV 算子来说，具体的计算公式为5.1，其中 $M \times N$ 的矩阵中的每个元素都要进行一次乘法和加法，因此是两次操作；ExecuteTime 为 GEMV 算子的多次连续执行的平均耗时，单位为秒。图5.1分别显示了在近存计算平台（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，LUT-W-R 算法配置子矩阵大小为 32×512。

$$Throughput = \frac{2 \times M \times N}{ExecuteTime} \times 10^{-9}\ GOPS \tag{5.1}$$

结果显示，与 CPU 平台的 MKL 库中的矩阵向量乘法相比，UPMEM 平台的最大吞吐算法 LUT-W-R 算法有 14.7 倍的提升，而相比较 GPU 的 cuBlas 而言，吞吐量基本持平有微小的差距。CPU 平台使用 MKL 库充分利用了 Intel CPU 的向量指令和多线程（OpenMP，设置线程数量为 16）并行计算，但是由于 CPU 平台有限的内存带宽和复杂的存储层级，GEMV 受到数据传输的限制，性能与近存计算平台相差较大。对于 GPU 的表现符合预期，因为 GPU 的内部带宽并

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

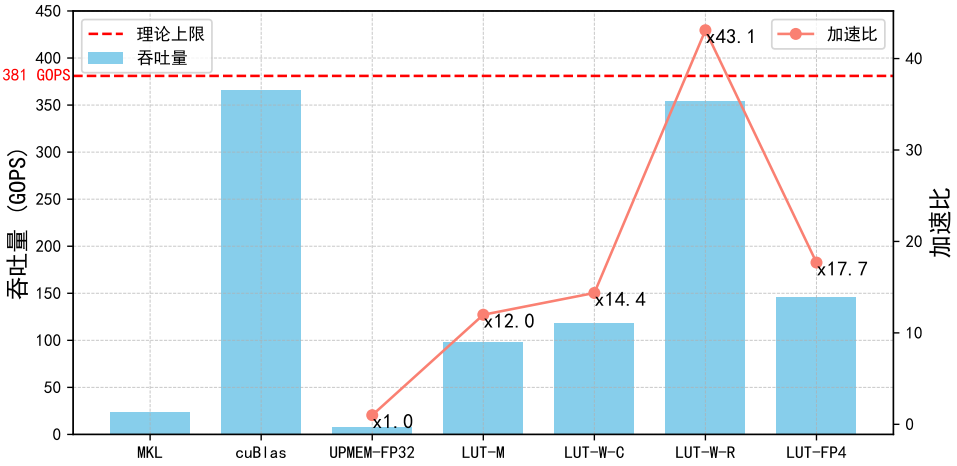


图 5.1 不同平台及算法下 GEMV 总吞吐

不低，此前在第四章的分析中发现 UPMEM 平台的内存带宽利用率最高也只达到 10% 左右，因此二者在数据传输方面的表现是类似的。同时 GPU 的核心数量并不少，且每个核心的主频更是有 1.41GHz 是 UPMEM 的核心主频的 2 3 倍，因此理应相较于 UPMEM 更快，但是 UPMEM 平台凭借着高速的内部带宽和精调的优化算法缩小了与 GPU 平台的差距。同时在 GPU 测试的过程中，我们发现计算吞吐（Compute SM Troughput）仅达 47.57%，远远没有充分利用硬件计算的能力，然而内存带宽利用率却以及达到 96.89% 表现为内存瓶颈。

同时可以看到在 UPMEM 平台上，软件优化的加速比。以最基础的 GEMV 算法 UPMEM-FP32 为基准，该算法就是普通地使用 UPMEM 上的软件模拟 Float32 直接进行计算而不采用查找表，其性能非常低，吞吐仅达 8GOPS。使用了针对 MRAM 的分块载入查找表的算法 LUT-M 后，相比 UPMEM-FP32 有巨大的提升，加速比高达 12 倍。之后是针对 WRAM 优化的两种算法将矩阵行重排 LUT-W-R 和矩阵列重排 LUT-W-C。在这之中，LUT-W-C 的算法的提升有限，达到 UPMEM-FP32 的 14.4 倍，而 LUT-W-R 算法提升较高，是各种软件优化算法中吞吐最高的算法，达到 355GOPS，是 UPMEM-FP32 的 43.1 倍。LUT-FP4 的算法是卸载 W4A8 量化的矩阵向量乘法，因为权重进一步量化到了 4bit，因此乘法查找表的大小刚好只有 16KB，完全能够放在 WRAM 中，因此执行的算法与 UPMEM-FP32 是类似的，同样是矩阵向量的内积，只不过乘法换成了查表操作。此处比较奇怪的是，LUT-FP4 算法虽然相比 UPMEM-FP32 算法有 17.7 倍

的提升，但是却弱于 LUT-W-R 算法，而 LUT-W-R 算法是载入子矩阵同样进行内积操作，理应比 LUT-FP4 要更慢。我们通过研究两种算法的汇编代码后发现，LUT-W-R 的算法汇编代码进行了循环展开而 LUT-FP4 完全没有展开，换言之 LUT-W-R 的编译优化非常好，性能远超 LUT-FP4（在第三小节的测试也侧面说明了这一点），因此出现了 LUT-FP4 算子吞吐弱于 LUT-W-R 的情况，推测主要原因在于 UPMEM 的软件栈不够完善。

同时为了评估硬件限制对 UPMEM 平台的影响，将 LUT-W-C 算法下的 GEMV 的实际性能与理论性能进行了比较。理论性能通过公式5.2计算。其中 N_{dpu} 表示 UPMEM 核心的数量，为 2560。F 表示核心频率，为 400MHz。N_{inst} 表示执行一次 FMA 操作所需的指令数，包括 5 条指令：1 条用于加载权重，1 条用于地址生成，1 条用于乘法，1 条用于加法，以及 1 条用于循环中的杂项操作。理论带宽上限为 381GOPS，测试的 LUT-W-C 算法的 GEMV 算子吞吐为 355GOPS，大概达到理论性能的 93%。

$$MaxThroughput = \frac{2 \times N_{dpu} \times F}{N_{inst}} \, GOPS \tag{5.2}$$

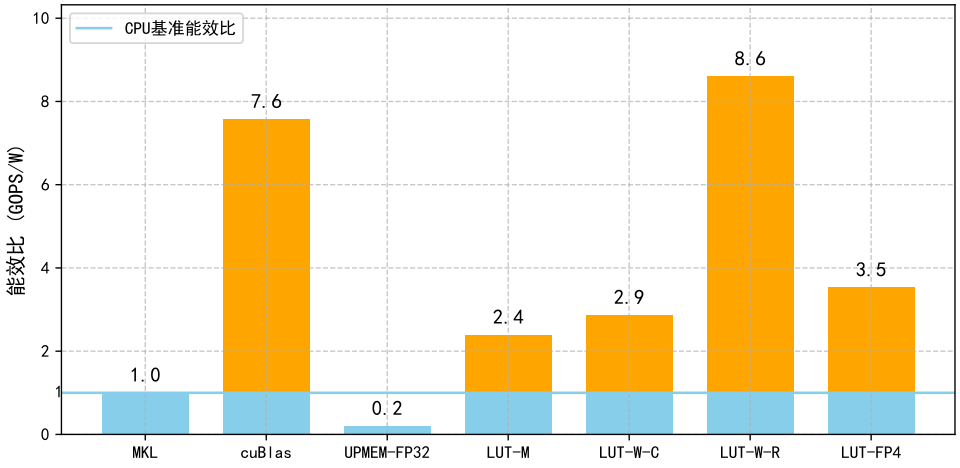
5.2.2 能效比测试与瓶颈分析

近存硬件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5.3，其中算子吞吐与上一小节中的测试结果保持一致，Energy 为能耗，单位瓦特（W）。

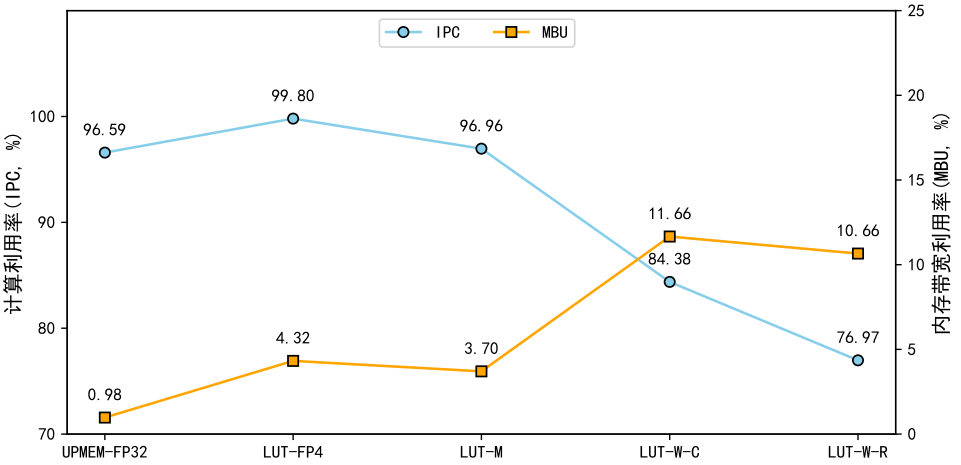
$$EnergyEffiency = \frac{Troughput}{Energy} \, GOPS/W \tag{5.3}$$

能效比的实验如图5.2(a)所示，其中以 CPU 平台的能效比作为基准（1），对其他平台和算法的能效比进行了测试和数据整理，可以看到除了 UPMEM-FP32 算法因过低的算子吞吐导致能效比较低外，其他平台的能效比皆高于 CPU 平

台，其中最高能效比的算法为 LUT-W-R，是 CPU 平台的 8.6 倍，是 GPU 平台的 1.13 倍，结果充分显示了近存计算平台的低能耗的优势。



(a) 能效比实验



(b) 瓶颈分析

图 5.2 不同平台及算法下 GEMV 能效比和瓶颈分析

同时在这里对第四章的表4.1进行补充测试，完善了 UPMEM 平台下各种算法的计算利用率和内存带宽利用率的数据如图5.2(b)所示，此时矩阵的大小为 4096×4096，设置 16 个 Tasklets，LUT-W-R 算法配置子矩阵大小为 32×64。此前已分析过 CPU 平台和 GPU 平台都是内存瓶颈，反观 UPMEM 平台的各种算法，IPC 都相当之高而内存带宽利用率仅最高达 11%，是非常明显的计算瓶颈

的表现。同时可以看到逐步优化算法会将 IPC 降低而 MBU 提高，是非常典型的使用访存减少计算的优化手段，这意味着在 UPMEM 这种近存平台编程或者加速应用需要转变此前的冯诺依曼架构计算机的编程定式——减少访存，在近存平台上访存变得更加的“便宜”，我们往往需要用访存换计算。

此外实验中比较特殊的点在于 LUT-FP4 相较于 LUT-M，从 MRAM 读写的字节数量是相同的（FP4 仍然用 8bit 存储减少取数的开销），但是由于 LUT-FP4 能够执行 GEMV 内积而 LUT-M 只能执行 GEMV 外积导致 LUT-FP4 的 WRAM 访问更少，因此运行时间更短，IPC 更高且 MBU 更高。

5.2.3 扩展性测试

本小节将从两个方面对算法进行扩展性测试，首先测试 UPMEM 每个 DPU 中的多线程扩展性，即保持矩阵的尺寸不变，增多 Tasklet 数量，观察算法性能。另一个方面，我们将测试算法处理矩阵尺寸的扩展性，即保持 Tasklet 的数量不变，增大矩阵的尺寸观察算法的性能。

对于多线程扩展性，本小节测试的矩阵大小为 4096×4096，具体的算法选择以下几种：UPMEM-FP32，LUT-FP4，LUT-M，LUT-W-C，LUT-W-R。设置 Tasklet 数量从 1 逐渐翻倍到 16，测试各个算子的执行时间如图5.3所示。可以看到无论是哪种算法，随着 Tasklet 的成倍增加，各个算法的执行时间随之成倍减少（对数坐标），然而当 Tasklet 的数量从 8 变化到 16 时，执行时间只减少到了原来的 1.3 倍，这个刚好符合从 8 变化打 11 扩大 1.3 倍的数据对应关系，也同样验证了 UPMEM 的流水线在 11 个线程时就已经充满，多增加线程无继续提升计算吞吐的硬特性。

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 4096×1 的向量映射成 32 个 128×1 的向量），本质上属于降维操作，（单个头）线形层是一个 4096×128 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 128×1 的向量重新映射为 4096×1 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 128×4096 的宽矩阵。同时由于 DPU 的通信开销低的特性，往往对算子进行融合，比如将 QKV 的单个头的线性层合并为大矩阵，此时的维度就是 4096×384，且 MLP 部分的矩阵可以随意切分，因此在实际的大模型推理过程中会遇到不同尺寸的矩阵。此前受制于多个平台的内存容量问题，我们只测试了 4096×128 的矩阵性能，当矩阵的行列发生变化时，算法的性能是否能够随

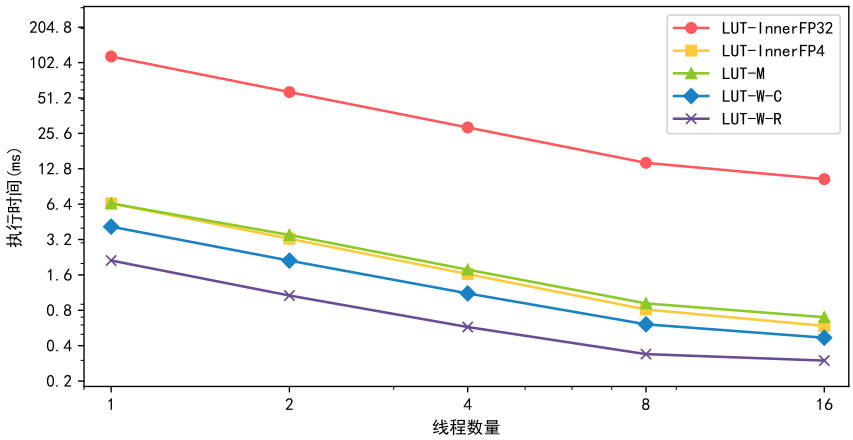


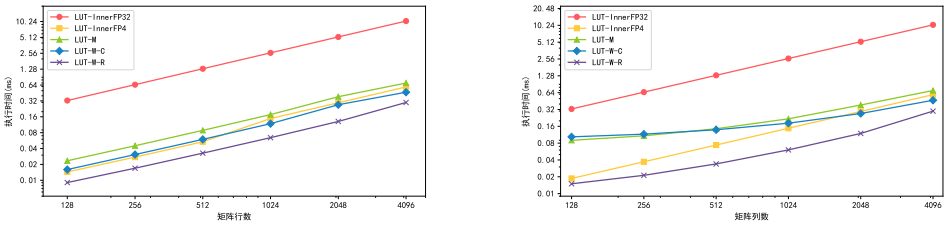
图 5.3 UPMEM 平台不同 GEMV 算法的多线程扩展性测试

之正常扩展，这是影响推理效率的关键问题。

对于矩阵尺寸扩展性测试，同样选择 UPMEM-FP32，LUT-FP4，LUT-M，LUT-W-C，LUT-W-R 这几种算法，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如图5.4所示。

对于改变矩阵的行数来说，如图5.4(a)，成倍地将矩阵的行数从 128 翻倍到 4096，算子的执行时间基本上是成倍地增加，说明上述的算法对于矩阵的行变化并不敏感。对于改变矩阵的列数来说，如图5.4(b)，同样的翻倍手段，大部分算法都是符合成倍减少的特性，对列变化不敏感，只有 LUT-M 和 LUT-W-C 除外。这两个算法对于列的成倍增加，算法的执行时间并未成倍的增加，而是低于 2 倍地增加，换句话说，当成倍减少矩阵的列时，LUT-M 和 LUT-W-C 的算法反而会“耗时增加”。

其中的原因在于这两个算法都是以行为单位进行同步和计算的，LUT-M 和 LUT-W-C 每次会载入权重矩阵的一行，当列数变小时，因 MRAM 的 DMA 引擎的特点（大批量数据载入带宽更高），DMA 的效率就会变低，相对应的执行时间就会变多。同时 LUT-W-C 的算法的优化的理论在于矩阵一行的所有元素不必都查一次表而只需查 256 次，即通过减少查表的次数来减少访存。当列数变小时，甚至小于 256 时，LUT-W-C 的查表次数反而可能超过未经过优化的查表次数，因此相比于 LUT-M，LUT-W-C 的“耗时增加”会更严重。



(a) 固定列为 4096 改变矩阵行　　(b) 固定行为 4096 改变矩阵列

图 5.4 UPMEM 平台不同 GEMV 算法矩阵尺寸扩展性测试

此外可以看到 LUT-W-R 在增加矩阵的列时，其执行时间并不是等比增长，而是耗时越来越多，原因在于 LUT-W-R 算法本身对权重矩阵的行列并不敏感，而是对其分块处理的子矩阵的大小敏感，子矩阵越大执行耗时越少。当增加矩阵的列时，会较为严重地挤占 WRAM 的空间导致子矩阵的变小，从而需要更多的 DMA 操作从 MRAM 中读取数据，从而造成执行耗时增加因此可以得出结论，LUT-W-R 的算法更加适合行数远大于列数的“窄矩阵”，而 LUT-W-C 的算法更加适合列数远大于行数的“宽矩阵”。

5.3 基于模拟器平台的硬件设计测试与分析

在真实硬件平台测试后 GEMV 算子的各项数据后，需要对基于模拟器平台的硬件优化进行测试。我们分别做了两部分的硬件优化，分别是增加了融合查表加法指令集（FLA）和向量指令（SIMD）：使用 FLA 指令优化第三章提出的三个基于查表的算法 LUT-M、LUT-W-C 和 LUT-W-R；使用 SIMD 指令优化 GEMV 朴素内积 LUT-FP4 的算法 LUT-SIMD。基于 UPMEM 周期精确模拟器 PIMulator 对上述优化进行测试，通过模拟器模拟器的周期数和主频测算算子执行耗时，进而得到算子优化后的吞吐。设置矩阵的尺寸为 4096×4096，设置 16 个 Tasklets，LUT-W-R 算法配置子矩阵大小为 32×64。具体的加速如图5.5所示。

对于算法 LUT-M、LUT-W-C 和 LUT-W-R 而言，使用了 FLA 指令集后算子的吞吐提升分别在 14.88%、27.82% 和 0%，在这里对结果进行解释。对于 LUT-M 和 LUT-W-C 而言，提升大概在 15% 30% 左右较为优秀但是没有达到理论预期。在汇编分析的代码段4.2中，FMA 操作的提升普遍在 2 3 倍，但是上述假设的情况过于理想，没有考虑寄存器的使用限制以及编译优化手段，在真实的算法汇

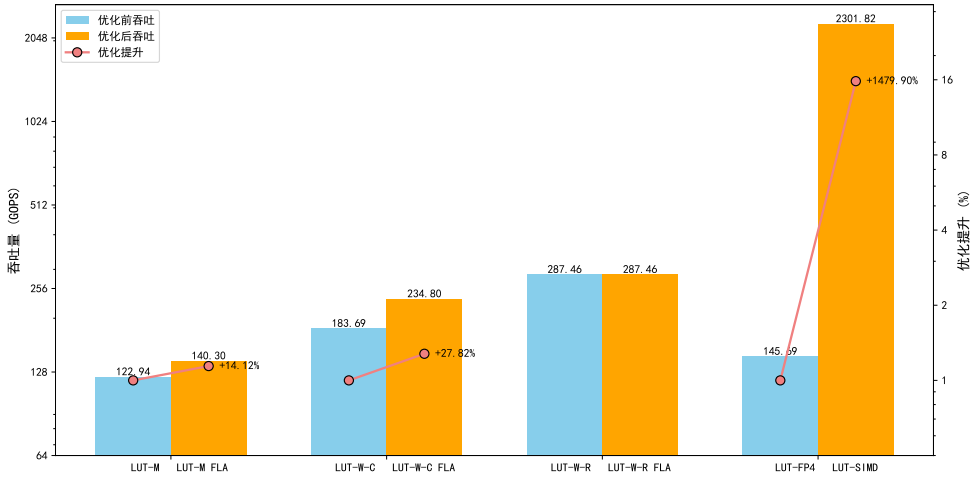


图 5.5 硬件架构修改对于 GEMV 算子吞吐提升——编译优化 O3

编代码中，能够使用我们设计的查找指令的代码段仅仅在查表指令代码段一处，并且通过一定的编译优化对指令进行了重排，无法通过人手动地更改整体代码段以达到最佳优化，其结果就是使用我们设计的 FLA 指令往往只能优化 FMA 操作中的一条到两条指令：比如 LUT-M 仅有一处查表可以使用 FLA 指令增强，就是最深循环出的查表语句；而 LUT-W-C 算法有两处可以使用 FLA 指令增强，一处是查表得乘积，另一处是查表得累加索引。而我们在之前的理论最大算子吞吐的计算过程中展示过，理论上一次 FMA 操作往往需要消耗 5 条指令，因此优化大概在 15% 30% 左右符合预期。

对于算法 LUT-W-R 而言，我们无法通过汇编代码分析找到任何可以优化的代码段。我们分析发现 LUT-W-R 的算法编译优化非常好，将查表操作的额外开销部分通过指令重排优化掉了，从而导致我们无法直接从汇编代码入手优化算法，因此并无加速效果。从上面的测试可以看出，UPMEM 平台的软件生态较为完善，编译优化较好，其中的部分原因可能是 UPMEM 本身基于 RISC 指令集，本身指令简单且优化成熟。然而很多其他的近存计算平台，由于底层的硬件和支持的指令集并不像 UPMEM 这样简单且流行，仍然存在不完善的软件链，编译优化不好的问题。因此基于上面的实验，我们做了补充实验5.6，对于各个算法不开启编译优化，其余配置保持一致测试了结果。

这时对于算法 LUT-M，使用了 FLA 指令集后算子性能提升能达到 17.64%，有少许提升。算法 LUT-W-C 的提升为 16.67%，该算法的 FLA 指令增强和 O3

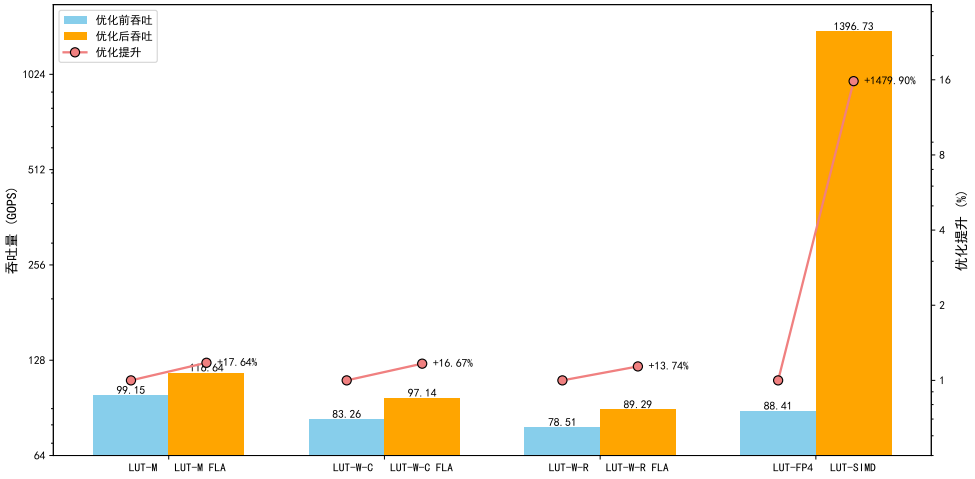


图 5.6 硬件架构修改对于 GEMV 算子吞吐提升——编译优化 O0

编译一样有两处，但是提升反而相比于 O3 编译减少了，其原因在于 LUT-W-C 的未经编译优化其复杂的源码结构使得其编译产生的汇编代码非常低效，吞吐量甚至低于 LUT-M（LUT-W-R 也是类似），并且我们在编译的汇编指令中发现了许多废指令，我们未将这些废指令或非查找表的指令优化计入，因此整体 LUT-W-C 的提升变低了。LUT-W-R 此时就有了 13.74% 的提升，提升较低的原因和其总吞吐低有关。

可以看到，对于编译优化较差的平台我们的 FLA 指令增强效果较好，尤其对于算法 LUT-W-C，有两次随机数组访问（查表），优化效果是最好的。但是我们提出的一系列 FLA 指令集其实不止局限于查表的优化，可以用于多种场景，许多移位、访存和累加操作都可以通过一条 FLA 指令完成。此时的工作应该在编译器层面进行，直接进行汇编指令的替换不会有效果。

对于使用 SIMD 指令加速的效果，不论编译优化等级，比较 LUT-FP4 和 LUT-SIMD 可以发现，加速效果约为 16 倍（提升是 1479% 差不多就是原来的 16 倍），这与理论提升高度符合，因为可以一次性向量完成 16 个元素的查表和加法，同时我们简单地修改了 PIMulator 模拟器的让其支持向量指令且保证 CPI（Cycle Per Instruction）为 1，与其他的 RISC 指令并无明显差异，因此原本 16 个元素的计算需要 16 次载入激活、载入矩阵、查表、累加，而使用向量指令后，就变成了 1 次载入子表、矩阵向量移位、重排、累加（每 8 次操作载入一次矩阵向量），因此性能提升非常大。

当然我们在模拟器上实现时并未考虑实际的硬件工艺和电路设计是否允许这样的支持，比如是否能够在较小的芯片内集成支持 512bit 向量计算的向量单元，并且能支持指令的 CPI 达到 1。当然我们可以结合更低 bit 权重量化的工作减小向量的长度，比如 3bit 的权重量化只需要 256bit 的向量运算，而 2bit 只需要 128bit。

在真实硬件平台上测试的能效比、瓶颈分析和扩展性实验，在模拟器平台上测试结论基本保持一致，在此不多赘述。

5.4 本章小结

本章对此前的软硬协同优化方案做了充分的测试及结果分析。首先介绍了测试的环境和基本配置，包括所使用的三大硬件平台的基本介绍和硬件配置，测试数据的配置以及对比基线的设置。然后在商用近存计算硬件 UPMEM 上做了一些的测试和分析，包括对各种基于 UPMEM 的优化算法的 GEMV 算子总吞吐测试，对比了 CPU 和 GPU 平台的算子吞吐，分析得到结论：UPMEM 上 GEMV 算子强于 CPU 平台并于 GPU 平台持平；同时包括对三个硬件平台 GEMV 计算的能效比测试以及计算瓶颈分析，分析得到近存计算平台 UPMEM 的能效比高于 CPU 和 GPU 平台且 GEMV 的优化算法表现为计算瓶颈；还包括对 GEMV 算子的扩展性测试，测试得到此前提出的 GEMV 算法对于 Tasklet 有良好的扩展性（不超过 11 个线程），而对于矩阵的尺寸大多也同样具有良好的扩展性，其中 LUT-M 和 LUT-W-C 算法更适合在行数远小于列数的矩阵，而 LUT-W-R 则更适合行数远大于列数的矩阵。最后在 PIMulator 模拟器上测试硬件的修改对于 GEMV 算子性能的提升，其中增加 FLA 指令集对于 LUT-M 和 LUT-W-C 的提升在 15% 30% 左右，增加向量处理单元对于 LUT-FP4 算法的提升在 16 倍。

第 6 章 总结与展望

本章将对全文内容进行总结与展望。首先是对全文的设计进行简要介绍，接着分析了现有研究工作的不足，提出了未来研究和改进的空间。

6.1 总结

本文主要是基于近存计算技术，针对优化大模型推理中最基本的算子矩阵向量乘做出优化和研究。首先是介绍大模型推理加速和近存计算的相关工作，包括大模型推理的基本概念、大模型量化的相关工作以及矩阵向量乘的相关优化手段，介绍了近存计算的发展历程，详细介绍了商用近存计算硬件 UPMEM 的硬件架构和特性，以及基于 UPMEM 的相关应用，重点介绍了 UPMEM 被用于加速神经网络的相关文献。

基于上述文献调查，我们在商用近存计算硬件上设计了基于查找表的矩阵向量乘的相关算法。首先我们针对 UPMEM 的 MRAM 层级设计了算法 LUT-M，通过分块载入查找表减少通过 DMA 传输数据到 WRAM 的次数，充分优化了 WRAM 的数据局部性。随后我们进一步针对 UPMEM 更高速的存储层级 WRAM 设计了矩阵行列重排算法 LUT-W-R 和 LUT-W-C。通过矩阵行重排，将矩阵分块载入，减少做 GEMV 的累加操作时频繁地读写结果向量；通过矩阵列重排，按照权重值重排构造分界数组，减少每一行矩阵计算时的查表次数。这两种基于 WRAM 的算法都优化了寄存器数据的局部性，减少了对 WRAM 的访问。

同时，我们测试每个上述软件算法在 UPMEM 上的执行 CPU 和内存带宽利用率，分析性能瓶颈为计算，普通的基于访存优化的算法难以解决。因为我们基于 UPMEM 时钟精确模拟器 PIMulator 修改硬件。首先我们增设新的硬件单元加入了对融合查表加法指令的支持：融合查表加法指令是将查表操作（包括计算内存地址和访存）和累加操作融合到一条指令中，通过减少指令数量和融

合指令执行加速算法。然后我们增加了向量单元以支持部分 SIMD 指令，使用 SIMD 指令的重排操作可以方便快速地进行多个数的乘法查表，同时以向量加法和向量位移指令辅助，可以大大提升访存和计算效率。

为了测试上说优化效果，在三个硬件平台 CPU、GPU、UPMEM 上我们设计了详尽的实验。首先我们测试 GEMV 算子的总吞吐，我们用计算量与执行时间的比值得到吞吐指标，在 UPMEM 上能达到 355GOPS，是理论性能的 93%，是 CPU 平台总吞吐的 14.7 倍有较大提升。同时测试各个平台的能效比，用吞吐与功耗的比值标识，结果表明 UPMEM 的能耗比大概是 CPU 平台的 8.6 倍，是 GPU 平台的 1.13 倍，充分发挥了 UPMEM 近存计算硬件的优势。

6.2 展望

本文在商用近存计算硬件 UPMEM 上卸载矩阵向量乘算子，提出了一系列的基于查找表的优化方法，并在 UPMEM 模拟器 PIMulator 上做出了硬件架构的修改，总体工作使得 GEMV 算子在 UPMEM 上的吞吐量表现较好。但是上述工作仍然有局限性和需要改进的地方，总结为以下几点：

1） 基于 PIMulator 的 UPMEM 硬件修改和探索不够深入。本文的工作对主要对硬件做了两处修改，包括增加融合查表指令和向量指令，更多地是基于指令集的简单修改，试图改善 UPMEM 计算瓶颈的问题，但是没有更多探索新的硬件架构诸如英伟达 GPU 的编程模型 SIMT，或者将传统 CPU 常见的超标量、超流水、乱序等技术引入 UPMEM 的架构中来，以及 WRAM 和 MRAM 的 DMA 传输引擎的分析和优化空间等等，都是值得深入探究的问题。

2） 硬件设计指令部分实现不够完善。本文的工作是基于 UPMEM 编译器编译用户代码为汇编代码后，直接对汇编代码进行模式匹配和人工分析进行指令的替换，这样存在无法充分利用编译优化的问题，而事实上本文也遇到了算法 LUT-W-R 的编译后的代码无法直接使用此前设计的指令优化的问题。因此最佳方式是从编译器入手，也就是 UPMEM 官方使用的 LLVM 项目入手增加指令，此类工作需要有深厚的编译优化基础，可以在将来对诸如 LUT-W-R 等算法进行进一步的优化。

参考文献

[1] 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.

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

[3] 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.

[4] 毛秋力, 沈庆飞, 李秀红. 面向算力中心的大模型推理优化技术[J]. 质量与认证, 2024(09): 40-44. DOI: 10.16691/j.cnki.10-1214/t.2024.09.007.

[5] 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.

[6] 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.

[7] Ibrahim M A, Islam M, Aga S. PIMnast: Balanced Data Placement for GEMV

Acceleration with Processing-In-Memory[C/OL]//SC-W '24: Proceedings of the SC '24 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis. Atlanta, GA, USA: IEEE Press, 2025: 970-981. https://doi.org/10.1109/SCW63240.2024.00137. DOI: 10.1109/SCW63240.2024.00137.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 孙思涵. 大模型及其应用前景分析——从大模型应用场景到推理算力在边缘的展望[J]. 江西通信科技, 2024(03): 1-2. DOI: 10.16714/j.cnki.36-1115/tn.2024.03.006.

[16] 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].

[17] 王睿, 张留洋, 高志涌, 等. 面向边缘智能的大模型研究进展[J]. 计算机研究与发展, 1-18.

[18] 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.

[19] 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.

[20] 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.

[21] Frantar E, Ashkboos S, Hoefler T, et al. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers[EB/OL]. 2023. https://arxiv.org/abs/2210.17323. arXiv: 2210.17323 [cs.LG].

[22] 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.

[23] 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.

[24] 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].

[25] 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.

[26] 殷建. 基于 GPU 的矩阵乘法优化研究[D]. 山东大学, 2015.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

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

[47] 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].

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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 Tech-

nologies, 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.

[54] 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].

[55] 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].

[56] 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].

[57] 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.

[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] 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.

[63] 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.

[64] 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.

[65] Ghose S, Hsieh K, Boroumand A, et al. Enabling the Adoption of Processing-in-Memory: Challenges, Mechanisms, Future Research Directions[EB/OL]. 2018. https://arxiv.org/abs/1802.00320. arXiv: 1802.00320 [cs.AR].

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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].

[72] 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.

[73] 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.

[74] 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.

[75] 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.

[76] Zhou Z, Li C, Yang F, et al. DIMM-Link: Enabling Efficient Inter-DIMM Com-

munication for Near-Memory Processing[C]//2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA). 2023: 302-316. DOI: 10.1109/HPCA56546.2023.10071005.

[77] 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].

[78] Kuzmin A, Baalen M V, Ren Y, et al. FP8 Quantization: The Power of the Exponent[EB/OL]. 2024. https://arxiv.org/abs/2208.09225. arXiv: 2208.09225 [cs.LG].

[79] Shen H, Mellempudi N, He X, et al. Efficient Post-training Quantization with FP8 Formats[EB/OL]. 2024. https://arxiv.org/abs/2309.14592. arXiv: 2309.14592 [cs.LG].

[80] Zhang Y, Zhao L, Cao S, et al. Integer or Floating Point? New Outlooks for Low-Bit Quantization on Large Language Models[EB/OL]. 2023. https://arxiv.org/abs/2305.12356. arXiv: 2305.12356 [cs.LG].

[81] 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.

[82] 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.

[83] 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；当研究陷入瓶颈，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。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。师弟们积极向上的态度和对科研的热情，也时刻感染着我。我们排座在实验室内，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。

而在远方的家中，母亲是我永远的温暖港湾。在外地上大学和攻读硕士的日子里，母亲给予了我无尽的关爱与支持。求学期间，每当我遭遇挫折，母亲总是在电话那头耐心倾听，用温柔的话语安慰我、鼓励我，让我在异乡也能感受到家的温暖。母亲，您默默承担起生活的琐碎，只为让我能安心追求学业，您的爱是我前行路上源源不断的动力。

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；愿实验室的伙伴们在科研道路上，能突破重重难关；愿家人平安健康，岁月温柔以待。