中国科学: 信息科学 2021 年 第 51 卷 第 2 期: 173–206
SCIENTIA SINICA Informationis
《中国科学》杂志社
SCIENCE CHINA PRESS
评述

内存计算研究进展

毛海宇, 舒继武*, 李飞, 刘喆

清华大学计算机系, 北京 100084

* 通信作者. E-mail: shujw@tsinghua.edu.cn

收稿日期: 2020–02–29; 修回日期: 2020–04–27; 接受日期: 2020–04–30; 网络出版日期: 2021–01–21

国家重点研发计划重点专项 (批准号: 2018YFB1003301) 和国家自然科学基金 (批准号: 61832011) 资助项目

摘要 随着应用数据处理需求的激增, 在传统冯 · 诺依曼 (von Neumann) 体系结构中, 处理器到主存之间的总线数据传输逐渐成为瓶颈. 不仅如此, 近年来兴起的数据密集型应用, 如神经网络和图计算等, 呈现出较严重的数据局部性, 缓存命中率低. 在这些新兴数据密集型应用的处理过程中, 中央处理器到主存间的数据传输量大, 导致系统的性能不佳且能耗变高. 针对传统冯 · 诺依曼体系结构的局限性, 内存计算通过赋予主存端一定的计算能力, 以缓解因数据量大以及数据局部性差带来的总线拥堵和传输能耗高的问题. 内存计算有两大形式, 一种是以高带宽的连接方式将计算资源集成到主存单元中 (近数据计算), 另一种是直接利用存储单元做计算 (存内计算). 这两种形式有各自的优缺点和适用场景. 本文首先介绍并分析了内存计算的提出和兴起原因, 然后从硬件和微体系结构方面介绍内存计算技术, 接着分析和总结了内存计算所面临的挑战, 最后介绍了内存计算给目前流行的应用带来的机遇.

关键词 内存计算, 近数据计算, 存内计算, 神经网络, 图计算

1 引言

近年来, 应用数据呈现爆炸式增长, 处理器和主存之间的带宽限制成为数据密集型应用的瓶颈. 此外, 目前流行的一些数据密集型应用, 如神经网络应用和图计算应用, 数据的局部性差. 这会导致处理器片上缓存命中率降低, 进而导致处理器和主存之间频繁地传输数据. 这样的大量数据传输除了使得总线拥堵并影响性能, 还会造成大量的能耗开销 [1~5]. 研究表明, 两个浮点数在 CPU (central processing unit) 和主存之间传输所需的能耗要比一次浮点数运算大两个数量级 [6,7]. 在大数据系统中, 能耗开销大会使得基于传统冯 · 诺依曼 (von Neumann) 结构的系统扩展性差, 甚至无法支持大型的数据密集型应用 [8~10].

内存计算 (processing in memory, PIM) 给总线上数据传输量过大的问题提供了一个根源性的解决方案. 它通过赋予内存一部分计算能力, 使其能直接处理一些形式单一且数据量大的计算 (例如向

引用格式: 毛海宇, 舒继武, 李飞, 等. 内存计算研究进展. 中国科学: 信息科学, 2021, 51: 173–206, doi: 10.1360/SSI-2020-0037
Mao H Y, Shu J W, Li F, et al. Development of processing-in-memory (in Chinese). Sci Sin Inform, 2021, 51: 173–206, doi: 10.1360/SSI-2020-0037

量乘矩阵). 内存计算有两种形式, 一种是将计算资源以高带宽的连接方式集成到主存单元中 (一般称之为近数据计算, near data computing, NDC) [11], 另一种是直接利用存储单元做计算 (一般称之为存内计算, compute in memory, CIM) [12]. 这两种形式都很大程度减少了中央处理器和主存之间的数据移动, 从而达到提升系统性能并降低能耗的目的. 数据表明, 近年来提出的内存计算架构相比于传统冯 · 诺依曼架构有着几十, 上百, 甚至上千倍的性能提升 [6, 7, 12]. 尽管在性能和能耗方面优势明显, 目前内存计算的应用仍面临着诸多挑战.

在近数据计算方面, 由于内存中用于数据处理的逻辑芯片计算能力较弱, 架构设计者需要分析和提取出程序中适合放到内存中做计算的部分, 其余留给中央处理器处理 [13]. 其次, 架构设计者还需要针对上层应用的特点, 对近数据计算中的逻辑芯片进行精心设计以取得大幅性能提升 [14]. 不仅如此, 近数据计算还缺乏高效透明的系统级支持. 虽然一些面向特定应用设计的近数据计算微体系结构提供了上层软件接口, 但是程序员需要对底层充分了解才能够使用该近数据计算系统 [15]. 也有研究者设计了对上层透明的内存计算系统接口, 但仅能适应特定的内存计算结构, 缺乏通用性 [16].

在存内计算方面, 由于用来直接做计算的内存单元能够支持的算子类型有限, 存内计算较难高效支撑复杂多样的计算模式 [12]. 因此, 研究者们常针对算子类型少且简单的数据密集型应用设计专用的计算体系结构和系统 [12]. 除此之外, 存内计算模块集成到现有系统结构中的形式尚不明确 [6, 12]. 虽然存内计算模块既可以作为存储模块也可作为计算模块, 但其作为内存的一部分和现有存储系统合作的方式仍有待研究. 另外, 存内计算模块硬件本身存在性能和可靠性的问题, 例如基于非易失存储的存内计算单元中非易失模块的可靠性问题 [17].

由此可见, 内存计算技术虽然潜力大, 但是其结构复杂, 与上层应用关系紧密, 应用到现有系统中仍面临着诸多挑战. 因此, 我们对现有的内存计算研究进行综述, 阐述内存计算从硬件架构到软件系统支持的相关技术方法和研究进展. 本文首先详细分析和介绍了内存计算的兴起原因, 然后介绍了内存计算的形式以及目前已有的内存计算微体系结构, 接着分析和总结内存计算所面临的挑战, 最后介绍内存计算给现在流行的应用带来的机遇.

2 内存计算的提出和兴起

2.1 内存计算的提出

20 世纪 70 年代, 超级计算机中单指令多数据流的性能不佳 [18]. 这是因为存在数据依赖的问题, 导致单指令多数据流无法进行高效的数据并行. 因此, 研究者们提出内存计算技术, 希望在存储端加上处理单元, 提前执行好部分计算以减少数据的依赖.

图 1 是 20 世纪提出的内存计算微体系结构. 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. 这种做法直观但并不实用. 一方面, 当时的技术能力还无法使计算单元和存储单元紧密地结合, 所加的硬件资源占用了较大的存储片上面积, 不能达到与其相匹配的计算资源利用率. 另一方面, 当时的应用所处理的数据量不大, 不能使所加的计算单元满载. 在当时的技术环境下, 这种直接在存储内部加计算单元的方式不仅难以获得较高的资源利用率, 而且不能达到占用面积小且能耗低的目的. 故而, 内存计算在 20 世纪并没有兴起.

2.2 内存计算的兴起

内存计算真正兴起是在 2010 年后, 数据呈现指数级暴增. 该阶段数据驱动的应用发展迅猛, 例如近年来流行的人工智能应用, 需要大量的数据进行模型训练, 推理时也需要处理大量的数据 [19, 20]. 以

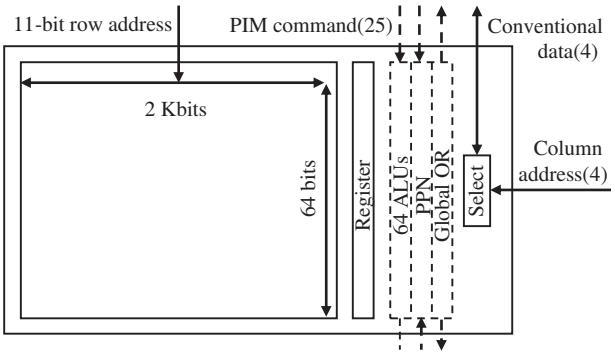

图 1 1995 年提出的内存计算微体系结构 [18]

Figure 1 Micro-architecture of PIM proposed in 1995 [18]

谷歌翻译为例, 近年来处理数据量不断增长, 目前一分钟需要翻译六千九百五十万个词 [21, 22]. 现代计算机中片上存储空间有限, 对于大规模的数据处理需求, 中央处理器将频繁地到主存取数据并把处理好的数据存回主存. 数据显示, 传输两个浮点数的能耗要比一次浮点运算大两个数量级 [6, 7]. 现在全球能源紧缺, 降低能耗是设计现代计算机的一大要点 [8~10, 23~25]. 因此, 研究者们重新考虑赋予内存一定的计算能力, 从而减少数据移动, 降低计算机系统运行能耗.

与此同时, 新型存储器件迅猛发展, 包括 3D 堆叠的存储器件, 如 HMC (hybrid memory cube)/HBM (high bandwidth memory) 和 3D XPoint, 以及交叉栅栏式 (crossbar) 结构的非易失性存储器件, 如 ReRAM 和 PCM. 图 2 [26] 展示了一种 3D 堆叠的存储结构. 其中, 最底层为逻辑层, 包含控制器和其他处理单元. 逻辑层上面堆叠有 DRAM 存储层, 他们通过高速地穿过硅片的通道 (through silicon via, TSV) 相连接. 图 2 中的一个立方体 (cube) 包含了 16 个拱 (vault), 每个拱的一层包含了两个阵列 (bank). 与传统的二维结构不同的是, 该 3D 结构最底层的逻辑层可以放置计算单元, 且可以通过超高速的 TSV 与上层的存储单元进行数据交互, 从而使得计算和存储结合得更加紧密. 图 3 展示了 ReRAM 的结构. 其中, 图 3(a) 是一个 ReRAM cell 的物理结构图, 以三明治模式将金属氧化物层夹在两个电极中间. 图 3(b) 展示了 SET 操作和 RESET 操作的电流和电压关系图. 图 3(c) 展示了 ReRAM 的 crossbar 的结构: 每个 cell 放置于字节线和比特线的交叉处. 这样的 NVM (non-volatile memory) 结构和传统的 DRAM 结构相比, 具有存储密度高和静态功耗低的优点, 同时其特殊的物理结构为存储和计算相结合提供了支持.

综上所述, 数据驱动的应用迅猛发展以及数据量指数级暴增驱动了内存计算的发展, 并且新型存储器件的快速发展为内存计算提供了技术保障. 因此, 内存计算在 2010 年后兴起.

3 内存计算架构与技术

内存计算技术是一个宏观的概念, 是将计算能力集成到内存中的技术统称. 集成了内存计算技术的计算机系统不仅能直接在内存中执行部分计算, 还能支持传统以 CPU 为核心的应用程序的执行 [12, 15, 16]. 区别于内存计算, 存算一体芯片将存储与计算相结合, 是一种 ASIC (application-specific integrated circuit) 芯片, 常用于嵌入式设备中, 针对一类特定的应用设计, 不能处理其他应用程序 [27]. 内存计算包括两大类: 近数据计算和存内计算. 两者的关系如图 4 所示, 它们在形式上不同, 但是在特定场景下可以融合设计. 近数据计算和存内计算的最大区别就是: 近数据计算的计算单元和存储单元

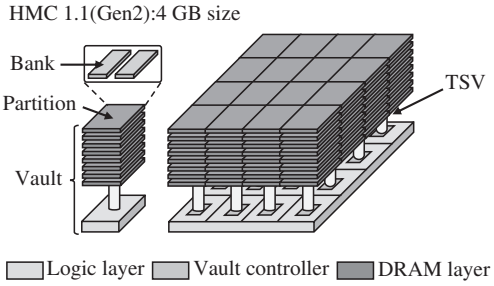

图 2 3D 堆叠的存储结构 [26]

Figure 2 3D-stacked memory [26]

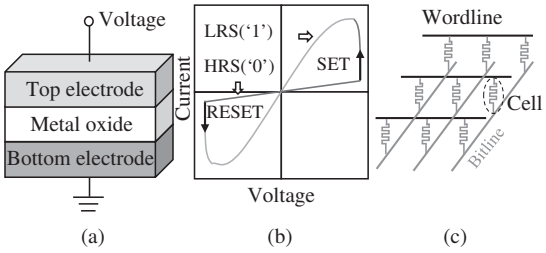

图 3 非易失性存储 [12]

Figure 3 Non-volatile memory [12]. (a) ReRAM cell structure; (b) the relationship of current and voltage in SET and RESET operation; (c) the crossbar structure of ReRAM

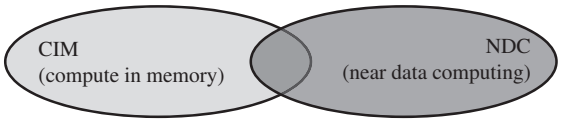

图 4 近数据计算和存内计算的关系

Figure 4 Relationship between CIM and NDC

依然分离, 而存内计算直接利用存储单元做计算, 计算和存储紧耦合. 下面将从硬件结构和所支持的计算操作两个方面具体介绍近数据计算和存内计算相关技术.

3.1 近数据计算

为缓解传统冯 · 诺依曼架构中总线上的数据传输问题, 近数据计算在存储周边放置计算单元, 这就需要高速通道进行连接. 因此, 近数据计算通常依赖于 3D 堆叠的内存结构, 如图 2 所示. 近数据计算系统中通常有一个或多个 NDC cube, 它们与 CPU 或者 GPU 相连接 (如图 5 所示 [15]), 多个 NDC cube 之间可能也会存在连接. 目前基于 3D 堆叠的近数据计算的研究主要集中在: (1) NDC cube 模块与现有系统的集成方式; (2) NDC cube 和 CPU/GPU 之间, NDC cube 之间的连接方式, 通信方式以及一致性协议; (3) NDC cube 中逻辑层的设计; (4) NDC 数据映射方式; (5) NDC 的软硬件接口及上层系统软件支持. 除了基于 3D 堆叠内存结构的 NDC, 还有基于 2D NVM 的 NDC 结构, 主要思想是对 NVM 中现有外围电路进行改造, 以支持特定类型的计算.

3.1.1 通用的近数据计算架构

通用的近数据计算架构方面代表性工作有: AMD Research 的 TOP-PIM [15], Carnegie Mellon Univeristy 的 TOM [16], University of Wisconsin-Madison 的 DRAMA [28] 和 NDA [7], Seoul National University 的 PEI [13], IBM Research 的 AMC (active memory cube) [29] 和基于多核 CPU 的近数据计算系统 [30], Stanford University 的 HRL [14], Brown University 为近数据计算设计的并发数据结构 [31], Georgia Institute of Technology 的 AxRAM [32], 以及 Chinese Academy of Sciences 的 proPRAM [33], 具体如下.

TOP-PIM [15] 提出了一种近数据计算的架构, 如图 5 所示, 存储单元不直接堆叠在中央处理器上, 而是堆叠在内存处理器上, 与中央处理器进行交互. TOP-PIM 在选择内存处理器时, 考虑了能耗和热

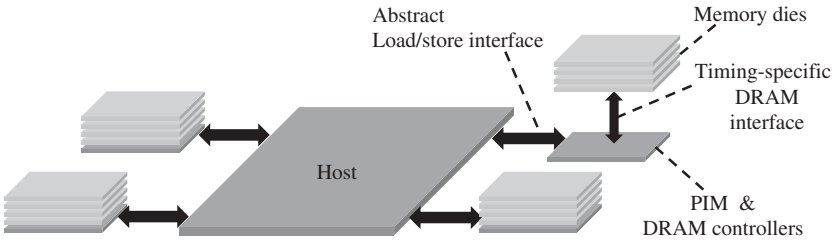

图 5 一个 NDC 系统的例子 [15]

Figure 5 An example of NDC system [15]

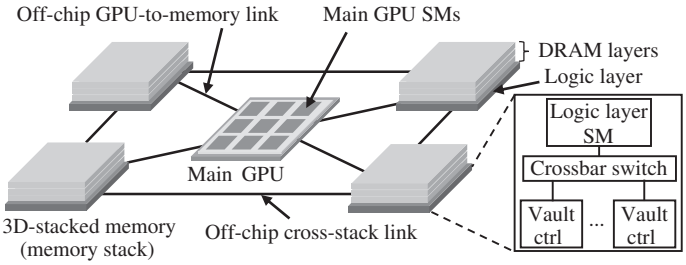

图 6 TOM 的结构 [16]

Figure 6 The architecture of TOM [16]

量的限制, 充分分析大量应用在性能和能耗方面的特征. TOP-PIM 认为面向吞吐量的 GPU 核更适用于高带宽高数据并行的场景. 实验表明, 在 22 nm 的工艺下, TOP-PIM 可以减少 76% 的能耗, 且仅带来 27% 的性能损失. TOP-PIM 测试的不是某种特定类型的应用, 无法对特定类型应用加速; 另外, 它将整个应用都放到内存计算中执行, 而内存计算中的计算资源更适用于单指令多数据流的计算, 难以支持复杂的多样的计算. 因此, TOP-PIM 在性能上不如传统冯 · 诺依曼系统.

TOM [16] 将代码分成多个块, 通过编译器的静态分析, 判断出适合放到内存计算中执行的块代码, 避免了把整个代码都放到内存计算模块中执行. TOM 结构如图 6 所示, 其大致结构与 TOP-PIM 相似, 不同的是 TOM 支持各个 NDC cube 之间的通信. TOM 的提出是为了解决大数据时代 GPU 与主存之间带宽小的问题, 除了通过编译器静态分析代码块并选择合适的代码块放到内存计算中执行之外, TOM 还分析预测了哪些数据会被放到内存计算中的代码块访问, 并将这些数据放在相应代码块执行的 NDC cube 中, 以此来减少各个 NDC cube 之间的通信. TOM 中的代码分析和数据映射都对上层透明, 程序员可非常方便地使用内存计算. 实验显示, TOM 平均能提高 GPU 的主流应用 30% 的性能.

DRAMA [28] 和 NDA [7] 与其他近数据计算架构不同, 它们建立在商用的 DRAM 设备上, 力求对现有商用 3D 堆叠的 DRAM 硬件 (HMC 和 HBM) 改动最小, 并且对现在的存储系统架构改动最小. 图 7 展示了 DRAMA 和 NDA 的结构, 二者整体结构还是中央处理器通过总线和 DRAM DIMM 相连. 不同的是, 其中一部分 DRAM DIMM 使用了堆叠在 DRAM 上面的 CGRA (coarse-grained reconfigurable array) 加速器, 能够将数据密集型的操作放到此类近数据结构中进行处理. 实验显示, 这样的近数据处理结构能够带来 3 倍到 60 倍的性能提升, 减少了 63%∼96% 的系统能耗.

PEI [13] 提出了一套内存计算和现有系统结合的软硬件接口. 如图 8 所示, PEI 将内存计算指令计算单元 (PCU) 放在每个主处理器核以及每个内存计算核上, 使得指令可在主处理器端或内存计算

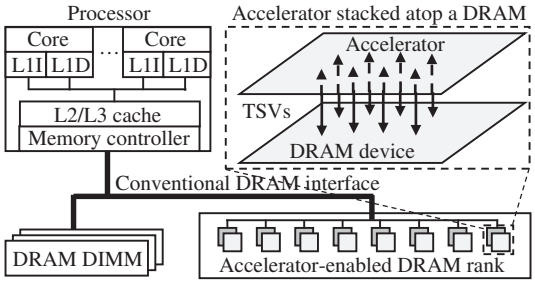

图 7 DRAMA 和 NDA 的结构 [7, 28]

Figure 7 The architecture of DRAMA and NDA [7, 28]

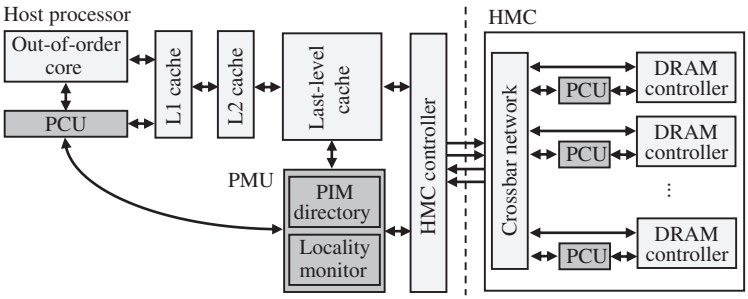

图 8 支持近数据计算的软硬件接口系统结构 [13]

Figure 8 The architecture of NDC-enabled instructions [13]

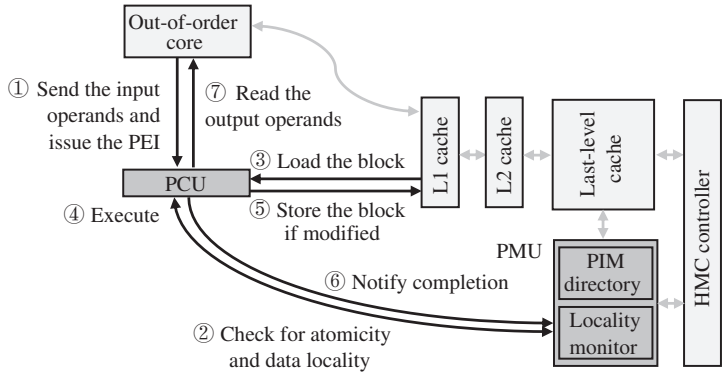

图 9 主机端支持近数据计算的指令执行 [13]

Figure 9 Host-side PEI execution [13]

端执行; 同时还加入了内存计算管理单元 (PMU), 与 LLC (last level cache) 以及主存控制器相互合作. 它的主旨是使用能够计算的存储指令和特殊指令实现简单的内存计算指令, 供上层应用所用. 在现有的顺序编程模型基础上, PEI 加入硬件单元来监测数据的局部性, 根据数据的局部性自动决定哪些操作在内存计算单元里执行. 因此, 内存计算系统能和现有的编程模型、缓存冲突处理机制, 以及虚拟内存管理很好地协作. 图 9 和 10 分别给出了主处理器端和存储端内存计算指令的执行步骤, 从中可以看出, PEI 能和现有的系统结构很好地协作.

AMC [29] 是一个面向百亿兆级超级计算机 (一秒钟执行 10^{18} 运算) 设计的内存计算系统, 图 11

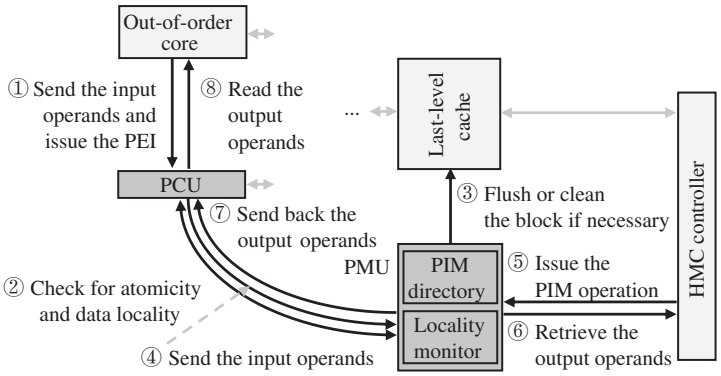

图 10 存储端支持近数据计算的指令执行 [13]

Figure 10 Memory-side PEI execution [13]

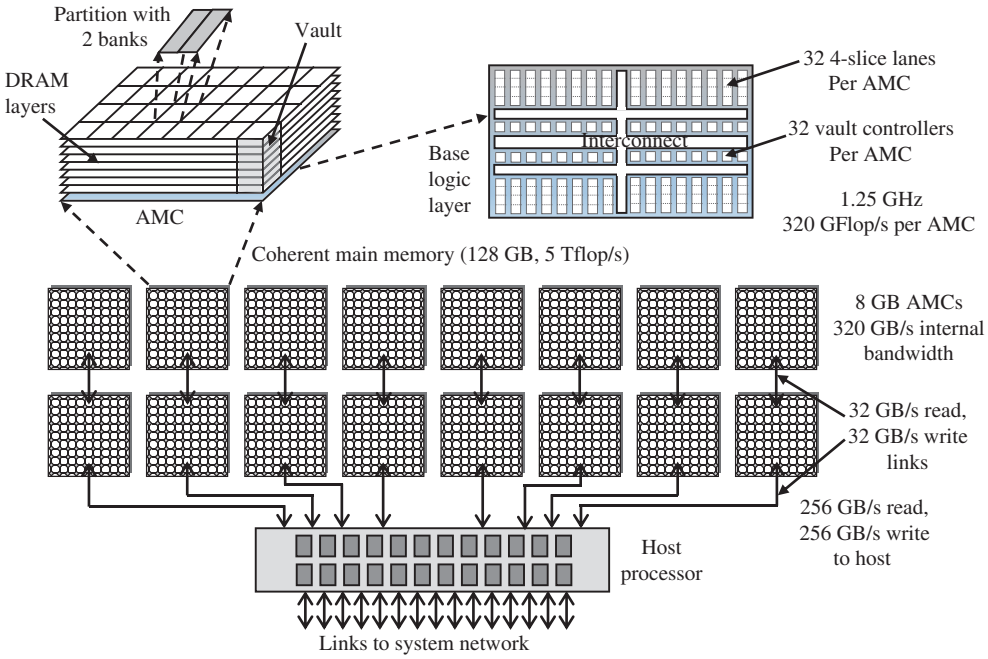

图 11 AMC 的系统结构 [29]

Figure 11 The architecture of AMC [29]

是其系统结构以及详细参数配置. AMC 在设计 NDC cube 的逻辑层时, 全方面考虑了常用科学应用的计算需求和百亿兆级计算机系统的低功耗需求, 力求最大化有限面积的利用率和能效比. 同时, AMC 通过软硬件接口将硬件微结构暴露给上层软件, 允许软件针对性能需求调整应用参数, 支持软件根据操作系统特点配置和调度 AMC 中的资源. AMC 使用 OpenMP 4.0 [29] 作为内存计算中节点粒度的编程接口, 相应编译器编译好程序供 AMC 执行. 实验显示, AMC 的内部带宽比传统中央处理器到主存的带宽大一个数量级; 此外, AMC 比传统处理器计算效率高一个数量级, 能以很高并行度执行负载的重要部分, 其能耗比传统系统一半还少. AMC 验证了内存计算比将存储移动到中央处理器端的模式效率更高.

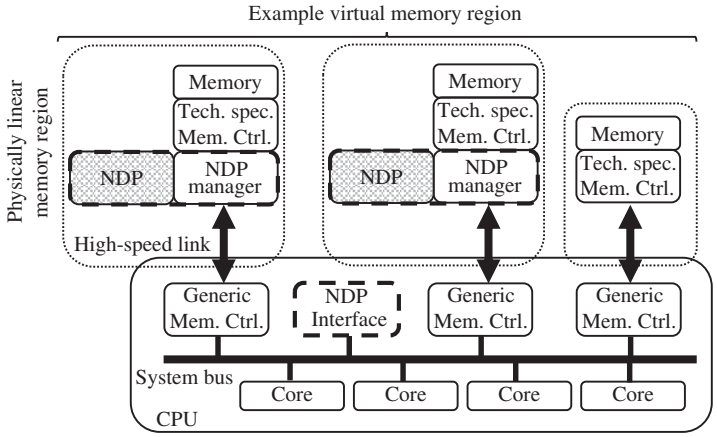

图 12　基于多核 CPU 的近数据计算 [30]

Figure 12　Multi-core CPU based near data computing [30]

Vermij 等 [30] 提出了一个基于多核 CPU 的近数据计算系统, 以支持近数据计算和现有系统的结合, 其结构如图 12 所示. 他们重新考虑了内存计算中的关键问题, 如数据冲突、数据排布、通信、地址映射和编程模型等, 并实现了一个基于软件和硬件的模拟器. 实验表明, 他们所提出的系统在 Graph500 测试中与 CPU 系统相比, 有 1.5 倍的性能提升.

HRL [14] 针对 NDC cube 逻辑层设计时, 片上功耗和面积限制与高带宽和充足计算量的矛盾, 提出了可重构的逻辑阵列, 能够灵活适用于大量的数据密集型应用. HRL 发现, ASIC 的设计性能好但缺少灵活性, 无法支持大量应用. FPGA 和 CGRA 通过可编程性提供了充足的灵活度. 但是 FPGA 是以比特为粒度的计算单元和互联单元可编程的硬件, 片上面积开销大; CGRA 能支持复杂数据流的强大互联, 能耗比 FPGA 更高, 在特殊计算和不规则数据上灵活度低, 性能不佳. 因此, HRL 提出了混合可重构的逻辑阵列, 用作 NDC cube 的逻辑层, 包含了 FPGA 块和 CGRA 块. 图 13 呈现了 HRL 阵列结构, 其中, FU (function unit) 是用类 CGRA 块实现的, 保证面积和能效比, 能够支持数学运算和逻辑操作, 包括加法、减法、乘法, 和比较操作; CLB (configurable logic block) 是用类 FPGA 块实现的, 用来实现一些不规则的控制逻辑和特殊函数 (比如神经网络中的激活函数); OMB (output multiplexer block) 是由多个多工器组成的, 放置在靠近输出的位置, 用来支持计算分支, 例如树形、瀑布型, 和并行型计算, 是一种能同时节省片上面积和能耗开销的分支实现方式. 另外, HRL 没有配置类似 FPGA 的 BRAM 缓存, 因为内存计算应用通常数据局部性差, 大缓存是额外负担, 不能提高系统的效率. 如上所述, HRL 综合了 FPGA 能耗低和 CGRA 面积利用率高的优点, 比基于 FPGA 的 NDC 系统性能高 2.2 倍, 比基于 CGRA 的 NDC 系统性能高 1.7 倍.

Liu 等 [31] 提出了适应于近数据计算的并发数据结构. 他们发现, 现在的服务器集群中通常有几百个核, 并发数据结构在吞吐和扩展性方面优于传统的顺序数据结构. 因此, 近数据计算系统如何支持并发数据结构, 以及并发数据结构如何利用近数据计算的优势, 成为研究重点. Liu 等发现, 并发数据结构性能优于普通的近数据计算的数据结构; 另外, 利用如数据融合、数据分块、流水线等技术面向近数据计算设计的并发数据结构, 性能优于传统 CPU 中的并发数据结构. 图 14 展示了他们设计的跳表结构实例, 以及该结构在近数据计算系统中的映射. 在这个例子中, 跳表被分成 3 部分存储到 3 个拱中, 每个部分都以哨兵点开头, 且哨兵点在 CPU 端有一份拷贝. 除此之外, 他们还实现了针对近数据计算设计的链表, 先进先出队列和管道, 在性能上优于现有的面向传统 CPU 系统的相关数据结构.

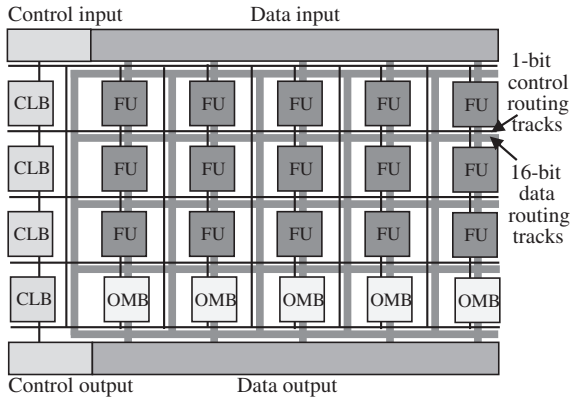

图 13 近数据计算的可重构逻辑层阵列 [14]

Figure 13 The reconfigurable logic for NDC [14]

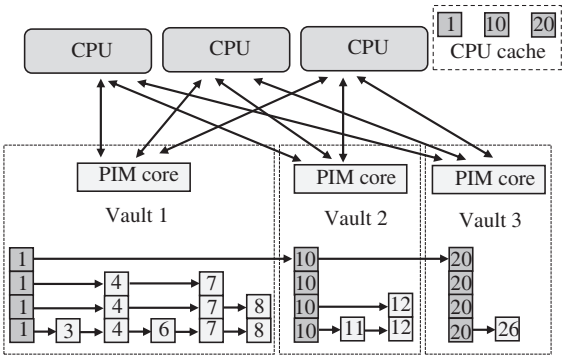

图 14 一个内存计算维护的跳表 [31]

Figure 14 An NDC-managed skip-list [31]

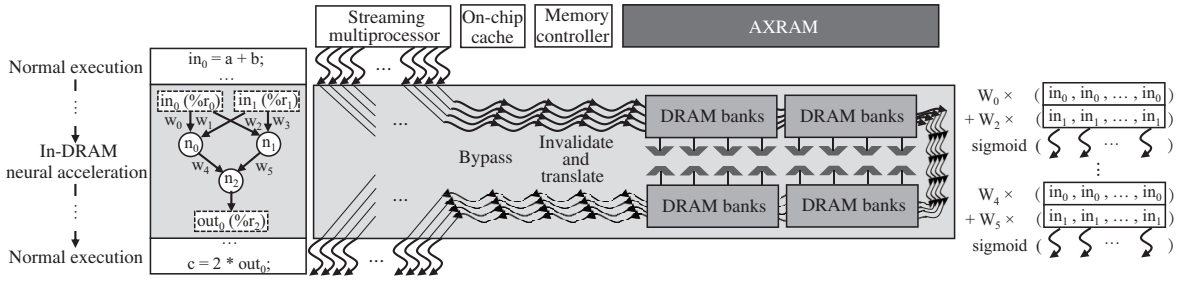

图 15 GPU 代码在近数据计算加速器中执行的流程 [32]

Figure 15 Execution flow of the GPU code on the NDC accelerator [32]

Yazdanbakhsh 等 [32] 认为将加速器集成到 NDC cube 中极具挑战, 需要加速器低能耗, 占用面积小, 还要支撑多样化的应用. 他们提出 AxRAM, 利用 GPU 应用的可近似性, 将不同区域代码中的计算转化成乘加 (multiply-accumulate operation, MAC) 等近似算子. AxRAM 用单一的算子近似应用中的操作, 从而使 NDC cube 中逻辑层设计简单、能耗低、占面积小. 图 15 展示了 AxRAM 的执行流程. 为了保持 GPU 的单指令多数据流的执行特性, 近数据计算端用多个 MAC 近似单元绕 GPU 翻译并执行命令, 这样做不需要改变 DRAM 的内部结构且不产生额外的内存开销. 实验结果显示, 与传统 GPU 系统相比, AxRAM 平均取得了 2.6 倍的性能提升以及 13.3 倍的能耗节约, 而只带来了 2.1% 的面积开销.

proPRAM [33] 是少数不基于 3D 堆叠结构的近数据计算架构. 它充分利用了 NVM 中的基础逻辑单元实现近数据计算, 例如数据比较写 (data-comparison write, DCW) 单元, 翻转写 (Flip-n-Write) 单元等对 SET/RESET 操作非常关键的单元 (如图 16 所示, 左边是 proPRAM 中的硬件结构, 右边是 DCW 单元). proPRAM 对硬件结构改动微小, 且改动对上层应用不可见, 能很好地支持数据密集型操作, 与 CPU 系统相比, 在数据密集型应用上能取得 15 倍的能耗节约.

3.1.2 针对机器学习的近数据计算架构

针对机器学习的近数据计算架构代表性工作有: Georgia Institute of Technology 的 BSSync

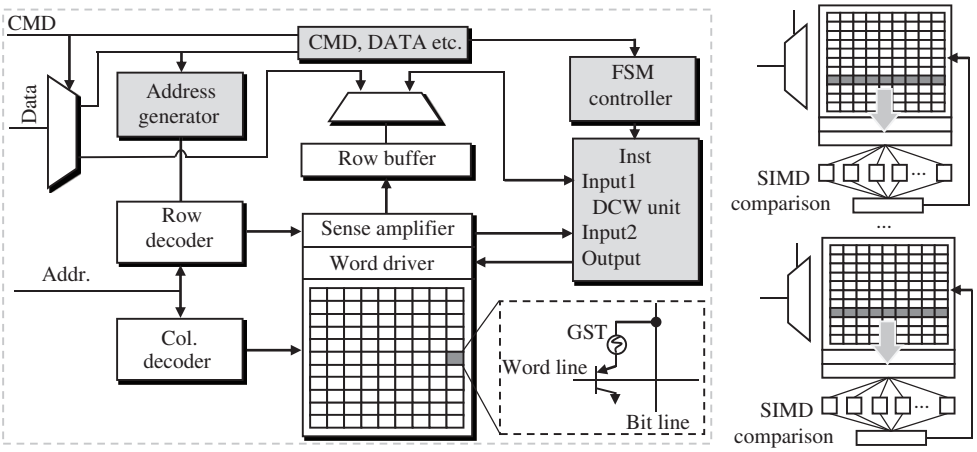

图 16　proPRAM 的结构和 DCW 单元 [33]

Figure 16　The architecture of proPRAM and DCW unit [33]

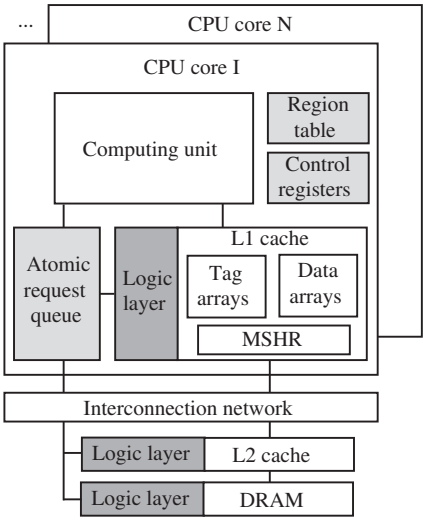

图 17　BSSync 的系统结构 [34]

Figure 17　The architecture of BSSync [34]

(bounded staled sync) [34] 和 Neurocube [35], Advanced Micro Devices 的 Co-ML [36], 具体如下.

BSSync [34] 指出, 在并行实现的机器学习应用中, 原子操作用来保障无锁状态下算法的收敛, 但带来很大的同步开销, 且同步产生的通信延迟不与占比大的计算延迟重叠. BSSync 发现, 在机器学习应用迭代收敛过程中, 可以用未更新的中间数据进行计算, 从而提出利用基于近数据计算的有边界一致性模型减少原子操作带来的延迟开销. 图 17 是 BSSync 系统结构, CPU 核里面增加了原子请求队列、控制寄存器以及区域表来实现边界一致性模型. 实验显示, BSSync 比机器学习应用在传统冯 · 诺依曼系统中的异步并行的实现快 1.33 倍.

Neurocube [35] 是一个针对神经网络计算设计的可编程、可扩展, 且节能的近数据计算系统架构. 图 18 是 Neurocube 架构, 左边是普遍使用的 NDC cube 结构, 右边是逻辑层设计. 逻辑层采用了细粒

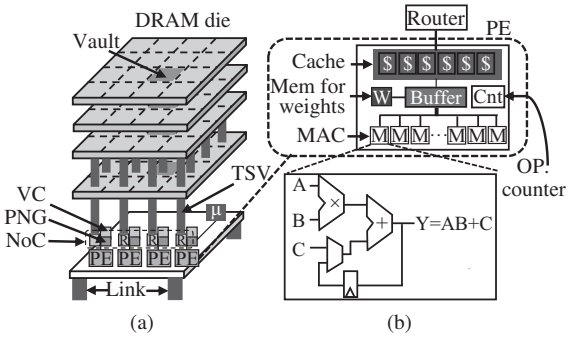

图 18 Neurocube 的微结构和它的逻辑层中处理单元的微结构 [35]

Figure 18 The micro-architecture of Neurocube and its processing element in the logic layer [35]

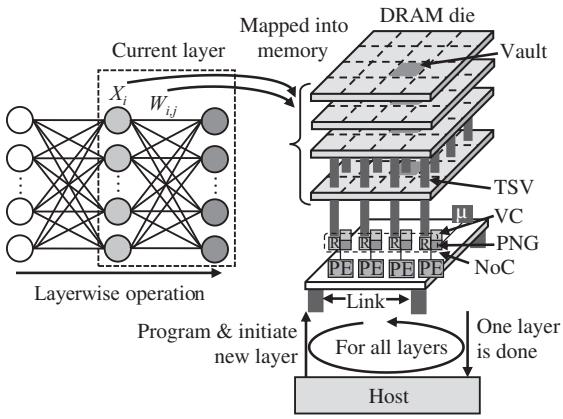

图 19 Neurocube 的执行流程 [35]

Figure 19 The execution flow of Neurocube [35]

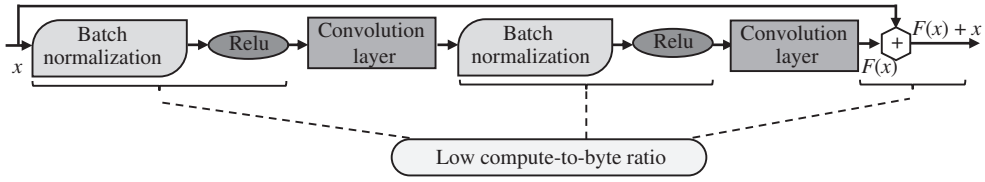

图 20 机器学习中计算/字节比率不高的部分 [36]

Figure 20 The low compute-to-byte ratio components in machine learning [36]

度可编程的设计模型, 以灵活支持神经网络计算. 其中, 每个 PE 有多个 MAC 单元支持神经网络中最常用的乘加操作, 同时还有存储权值的寄存器和缓存以及相应的计数器. 图 19 是 Neurocube 的执行流程. 它首先将神经网络存储到 NDC cube 的存储单元中, 包括每层数据、神经元状态、连接权值. 当一个层处理好之后, 与中央处理器交互一次, 然后执行下一层. Neurocube 通过对逻辑层硬件、数据映射方式、片上互联, 以及编程方式的精心设计, 使得神经网络计算在 NDC cube 中能够高效执行. 实验显示, 相比于 GPU 系统, Neurocube 有 4 倍的每瓦计算效率提升, 与 ASIC 系统相比, 灵活性更好、扩展能力更强.

不同于针对机器学习设计的注重优化乘加 (MAC) 操作的近数据计算系统, Co-ML [36] 提出, 虽然包含 MAC 操作的卷积层等计算占整个机器学习过程的比例大, 但这些计算是计算密集型的, 数据复用性好, 计算/字节比率高 (即一个字节从内存中读出来之后用来计算的次数多); 事实上, 机器学习过程中, 约 32% 的时间用于数据密集型计算, 这些计算的计算/字节比率低. 图 20 展示了神经网络中低计算/字节比率的计算部分. Co-ML 将这些低计算/字节比率的计算部分放在近数据计算端, 把 MAC 等操作放在主处理器上做. 实验显示, Co-ML 在机器学习的数据密集型计算上的加速达到了 20 倍, 总体有 14% 的性能提升.

3.1.3 针对图计算的近数据计算架构

针对图计算的近数据计算架构的代表性工作有: Seoul National University 的 Tesseract [37] 和 Geor-

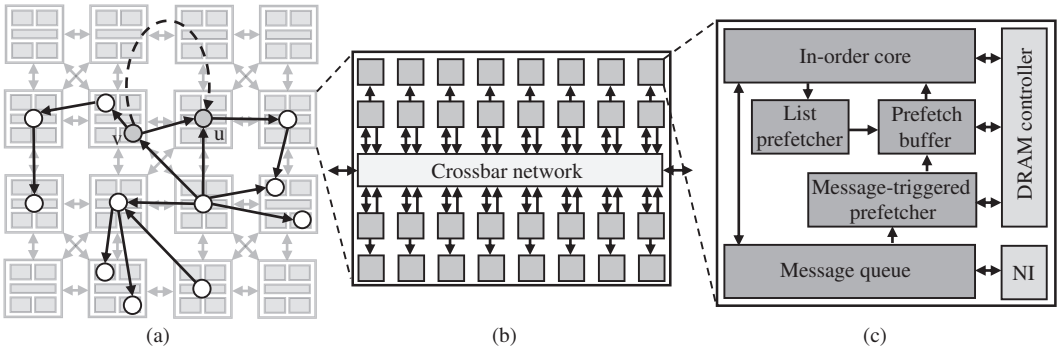

图 21 Tesseract 的系统结构 [37]

Figure 21 The architecture of Tesseract [37]. (a) Network of cubes (b) cube (HMC); (c) vault.

gia Institute of Technology 的 GraphPIM [38], 具体如下.

Tesseract [37] 是一个针对图计算的可编程的内存计算系统架构, 它综合了图计算的特点, 重新考虑了逻辑单元和存储单元的集成方式. 图 21 是 Tesseract 的系统结构, 左边是一个图计算在互联的 NDC cube 中执行的实例, 中间是一个 NDC cube 内部的连接结构, 右边是一个拱内部的逻辑层结构. Tesseract 逻辑层使用了顺序执行的计算核. Tesseract 还使用了可以隐藏远程访问延迟的消息传递机制, 以及为图计算定制的预取硬件, 和一系列支持这些操作的上层接口. 实验显示, 与传统的系统相比, Tesseract 能取得 87% 的能耗节约.

GraphPIM [38] 是一个针对图计算的, 基于商用 HMC2.0 的近数据计算完整解决方案. GraphPIM 主要解决利用近数据计算运行图计算的两大挑战: (1) 应该把图计算应用的哪些部分放到 NDC cube 中执行; (2) 如何把部分计算放到 NDC cube 中执行, 即如何提供中央处理器和近数据计算处理器的接口. GraphPIM 发现, 在图计算中, 原子操作是损害性能的主要原因, 因此应该把原子操作放到内存端, 避免不规则的数据访问带来的大通信开销. GraphPIM 使用现有的中央处理器指令, 把中央处理器端的原子操作指令映射到近数据处理端, 并使用无缓存的配置. 在 GraphPIM 架构下, 上层应用程序员无需额外的工作, 现有的指令集结构也不需要改动, 就可以在图计算的负载中使用近数据计算. 图 22 是 GraphPIM 的系统结构, GraphPIM 在图数据管理和硬件支持方面做了改动. 在虚拟地址空间中, GraphPIM 使用传统系统的指令绕过缓存来分配图数据; 在硬件端, GraphPIM 在中央处理器上加了一个 POU (PIM offloading unit) 用来决定哪些操作放到 NDC cube 中执行. 此外, GraphPIM 充分分析了图计算应用特征, 判断出哪些部分放到近数据处理端做会有性能提升. 实验显示, GraphPIM 与传统用 HMC2.0 当做主存的冯 · 诺依曼系统结构相比, 有 2.4 倍的性能提升和 37% 的能耗节约.

3.1.4 针对垃圾回收的近数据计算架构

Sungkyunkwan University 的 Charon [39] 是一个针对垃圾回收提出的近数据计算系统. Charon 发现了在大数据系统中垃圾回收的开销尤为严重: 在存储密集型的应用里, 垃圾回收几乎占了一半的执行时间. 通过分析垃圾回收算法, Charon 发现, 主要占 MinorGC 执行时间的操作是 Search, Copy 和 ScanPush; 主要占 MajorGC 执行时间的操作是 ScanPush, Bitmap Count 和 Copy. 不仅如此, 这些操作需要通过多线程并行提高吞吐量, 而且这些操作的空间局部性和时间局部性很差. 传统冯 · 诺依曼系统结构在执行这些操作时效率低下, 而近数据计算则非常适合. 因此, Charon 把这些操作放到近数据计算端处理. 图 23 是 Charon 的系统结构, 其中加了 Copy/Search, BitMap count 和 Scan&Push

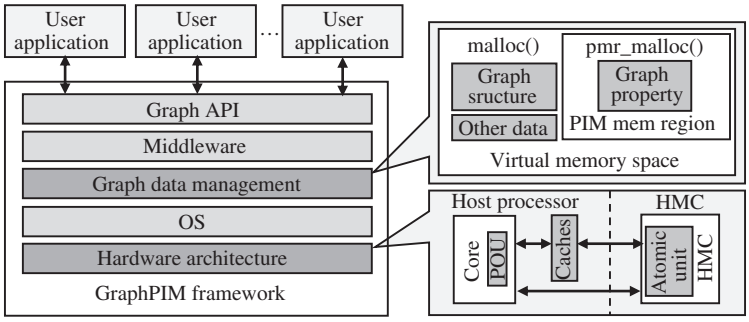

图 22 GraphPIM 的系统结构 [38]

Figure 22 The architecture of GraphPIM [38]

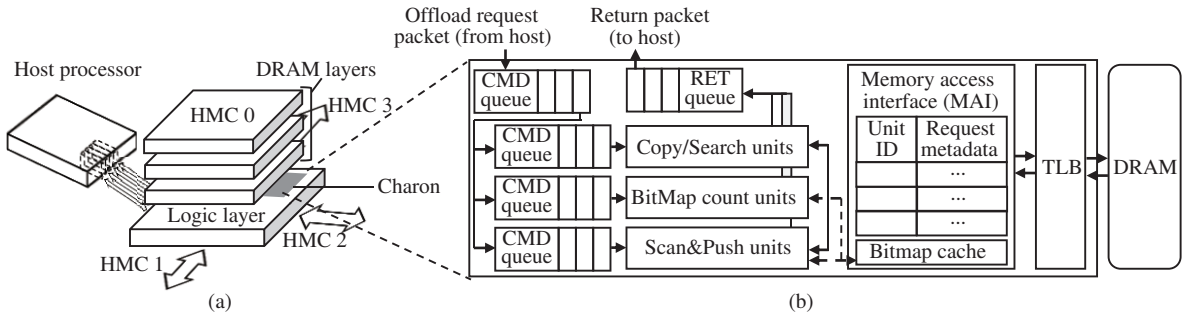

图 23 Charon 的系统结构 [39]

Figure 23 The architecture of Charon [39]

三个处理单元, 用来处理垃圾回收中对应的操作. 这些单元直接与主存交互, 避免了主处理器通过串行的低带宽读取数据, 从而减少了数据传输的时间和能耗开销. 实验显示, Charon 能取得 3.3 倍的性能提升和 60.7% 的能耗节约.

3.1.5 近数据计算小结

近数据计算中逻辑层的设计较为灵活, 可以针对不同系统的需求设计通用的处理器或者专用的加速器. 在设计针对通用应用的近数据计算系统时, 由于放到内存端的通用处理器一般性能较弱, 需要考虑自动化地分割应用程序的计算部分, 把能从近数据计算中获益的部分放到内存中处理. 在设计针对特定类型应用的近数据计算系统时, 需要仔细分析应用特点, 抽取算子, 设计对应的数据流. 除了逻辑层的设计, 近数据计算系统结构设计还需要考虑: 各个内存块之间的连接方式, 包括通信方式和数据一致性协议、数据映射策略、与现有系统集成方式、软硬件接口设计.

3.2 存内计算

和近数据计算不同, 存内计算直接使用内存单元做计算, 主要利用电阻和电流电压的物理关系表达运算过程. 存内计算依赖于新型的非易失性存储器, 如 ReRAM 和 PCM 等.

在所有存内计算操作中, 最普遍的是利用基尔霍夫定律 (Kirchoff's Law) 进行向量乘矩阵操作 [40]. 原因在于: (1) 它能够高效地将计算和存储紧密结合; (2) 它的计算效率高 (即, 在一个读操作延迟内能完成一次向量乘矩阵); (3) 目前流行的数据密集型应用中, 如机器学习应用和图计算应用, 向量乘矩阵

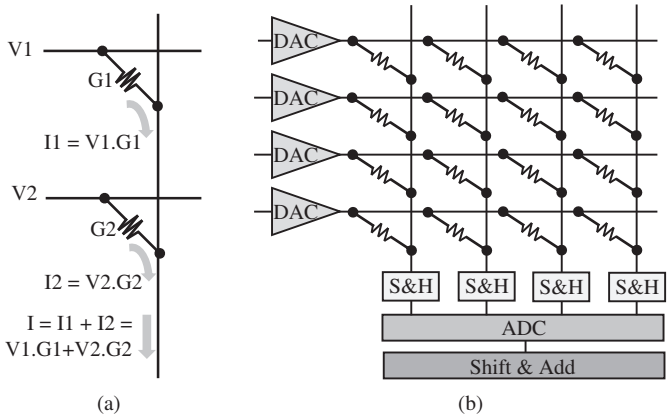

图 24　基于基尔霍夫定律的向量乘矩阵存内计算实现 [40]

Figure 24　PIM realization of vector matrix multiplication based on Kirchoff's Law[40]. (a) Multiply-accumulate operation; (b) vector-matrix multiplier.

的计算占了总计算量的 90% 以上[41].

除了向量乘矩阵操作, 存内计算还能利用电阻、电流及电压的物理关系实现查询, 按比特与/或/非等操作. 本小节首先综述基于向量乘矩阵的存内计算研究, 然后综述其他存内计算技术.

3.2.1　基于向量乘矩阵的存内计算

图 24[40] 是存内计算支持向量乘矩阵的最基本单元, 展示了存内计算使用基尔霍夫定律, 在将近一个读操作延迟内完成一次向量乘矩阵操作的过程. 左图中计算的是一个 2×1 的向量 $(V1, V2)$ 乘以一个 1×2 的向量 $(G1, G2)^{\mathrm{T}}$, 其中, $(G1, G2)^{\mathrm{T}}$ 用 ReRAM 阻值表示, 事先存在 ReRAM 中, $(V1, V2)$ 用电压表示, 加到对应的字节线上. 根据基尔霍夫定律, 比特线上最后输出的电流值就代表了 $(V1, V2) \times (G1, G2)^{\mathrm{T}}$ 的计算值. 同理, 扩展到右图的向量乘矩阵操作, ReRAM 阵列中存储着要做计算的矩阵, 将向量转化成电压加在字节线上, 通过比特线得到的输出就是相应的结果向量. 由于向量乘矩阵操作是神经网络和图计算中的主要操作, 这种内存计算结构得到了高效利用.

基于向量乘矩阵的存内计算代表性工作有: Hewlett Packard Laboratories 的 DPE[42], University of Utah 的 ISAAC[40], University of Santa Barbara 的 PRIME[12], University of Pittsburgh 的 PipeLayer[6], Tsinghua University 的 TIME[43], Tsinghua University 的 LerGAN[44], IBM Research 的 PCM+CMOS 存内计算[45], University of Rochester 的 SC[46], Duke University 的 GraphR[47]. 下文将综述这些工作如何支持神经网络应用或图计算应用, 以及其他包含向量乘矩阵的应用.

DPE[42] 是一个专门针对向量乘矩阵操作设计的存内计算加速器. 它提供了一个转化算法, 可将实际的全精度矩阵存储到精度有限的 ReRAM 存内计算阵列中, 减少器件问题以及外围电路问题对计算结果的影响. 图 25 是 DPE 的工作流程, 分为 3 个部分: 转换、写入、计算. 首先将矩阵映射到合适的 ReRAM 阵列中. 这个过程利用了对输入的预先了解以及 ReRAM 阵列参数共同优化来决定最后写入 ReRAM 阵列的数据. 而后通过写入阶段, 再进入计算阶段. 计算阶段将预先准备好的输入数据转成信号, 再传入 ReRAM 阵列中并读取输出数据. 如果还有其他计算操作, 则将临时输出传送到下一个 ReRAM 阵列中; 如果没有, 则结束计算. DPE 测试结果显示, 只用 4 bit 的 DAC/ADC (电信号转模拟信号单元/模拟信号转电信号单元) 就能保证计算结果没有精度损失, 相比于数字的 ASIC 向量乘矩阵加速器, 能取得 1000 到 10000 倍的性能提升.

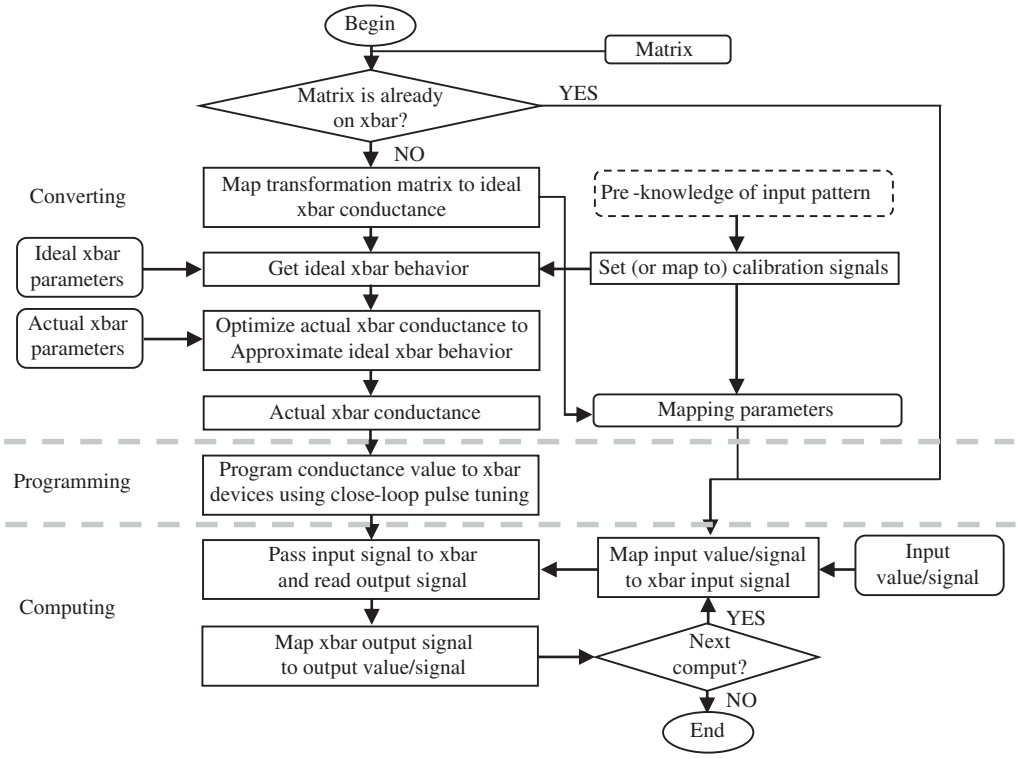

图 25 DPE 的工作流程 [42]

Figure 25 The work flow of DPE [42]

ISAAC [40] 是一个针对神经网络推理设计的存内计算架构, 图 26 是其整体架构. 一个芯片上包含多个存内计算阵列 (tile), 它们通过 C-mesh 的片上网络连接, 可以互相通信. 存内计算阵列里有用于池化层计算的最大池化单元 (Max Pool, MP), 用于激活层计算的 Sigmoid 单元, 用于数据缓存的 eDRAM buffer, 用于中间数据移位加操作的 S+A 单元、用于存放临时输出的输出数据寄存器, 以及支持原地向量乘矩阵操作的基础单元 (in-situ multiply accumulate, IMA). 每个 IMA 中包含 4 个基于 ReRAM 阵列的向量乘矩阵单元、电模互转单元 (DAC, ADC)、输入寄存器、移位加操作单元, 以及输出寄存器. ReRAM 阵列的个数和其他电路单元的设计考虑了向量乘矩阵的计算延迟以及片上网络的带宽, 充分利用了片上资源. 该结构在做推理时, 采用了 pipeline 的方式将硬件时分复用, 以加快整个推理的过程. 然而, 推理过程中会有很多由归一化操作产生的气泡, 当推理任务松散时, ISAAC 的 pipeline 效果并不理想. 相比于针对神经网络的加速器 DaDianNao, ISAAC 有 14.8 倍的性能提升和 5.5 倍的能耗节约.

PRIME [12] 也是一个针对神经网络推理设计的存内计算架构, 图 27 是其系统结构. 在一般的加速器结构中, 计算加速单元作为 CPU 的协处理器放在 CPU 旁边, 通过总线与主存相连 (如图 27(a) 所示). 在基于 3D 堆叠的近数据计算架构中, 加速单元靠近主存堆叠, 并通过总线与 CPU 相连 (如图 27(b) 所示). 在 PRIME 中, 直接使用 ReRAM 单元做计算. 其中, 一个 ReRAM bank 分为 3 部分: 用作存储的 Mem subarrays、用作计算的 FF subarrays, 以及用作缓存的 Buffer subarray. 计算阵列和缓存阵列进行数据交互, 缓存阵列和存储阵列进行数据交互. 与 ISAAC 不同的是, PRIME 不用片上 eDRAM 作为缓存, 也不使用输入输出寄存器, 而是直接使用 ReRAM 阵列作为缓存和存储. 与基

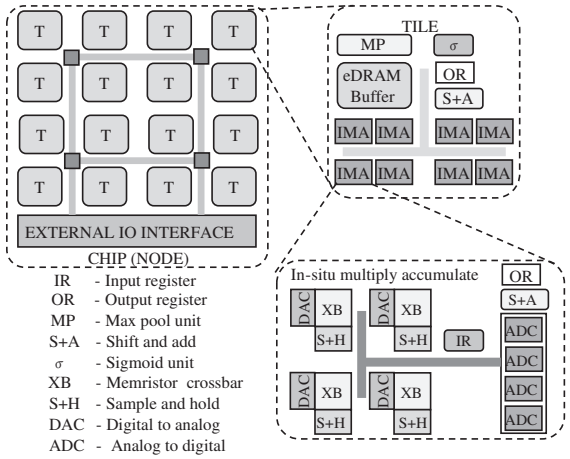

图 26 ISAAC 的系统结构 [40]

Figure 26 The architecture of ISAAC [40]

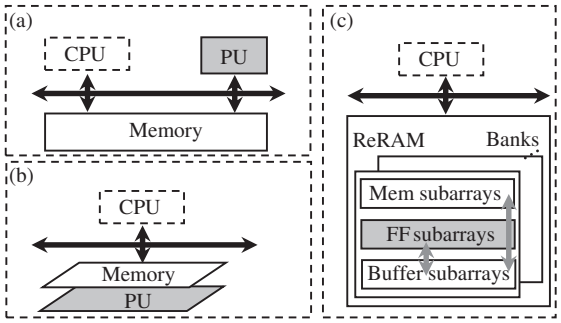

图 27 PRIME 的系统结构 [12]

Figure 27 The architecture of PRIME [12]. (a) Processor-coprocessor arch; (b) PIM with 3D integration; (c) PRIME

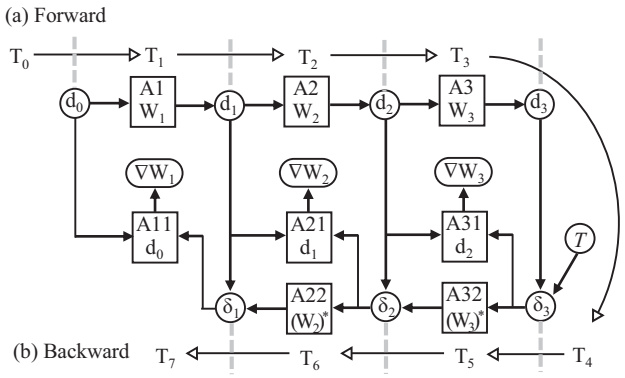

图 28 PipeLayer 训练一个神经网络时的数据流 [6]

Figure 28 The data flow in PipeLayer when training a neural network [6]

于 CPU 的神经网络处理器相比, PRIME 能够取得 2360 倍的性能提升和 895 倍的能耗节约.

PipeLayer [6] 是一个针对神经网络训练设计的存内计算系统架构, 图 28 展示了其训练一个三层神经网络的数据流情况. 其中, 圆形圈出的数据存在普通 ReRAM 中, 方块中的数据存在基于 ReRAM 的存内计算阵列中. PipeLayer 通过合理地复制多份权重数据 (图中的 A1, A2, A3, A11, A21, A31, A22, A32) 实现少气泡的 pipeline 结构, 同时使得反向传播阶段的误差传递和权值计算并行, 从而提高使用存内计算训练神经网络的计算效率. 实验显示, 与 GPU 系统训练神经网络相比, PipeLayer 有 42 倍的性能提升和 7 倍的能耗节约.

TIME [43] 也是一个针对神经网络训练的存内计算系统架构, 与 PipeLayer 不同的是, 为了减少训练时权重矩阵更新带来的高延迟和高能耗的问题, 它采取权重矩阵复用的方法, 而不是将权重矩阵复制多份来保证训练过程的高度并行. 同时, TIME 还支持增强学习的训练. 图 29 是增强学习网络的推理和训练过程. 它拥有两个网络, 训练过程会产生一个将 A 网络的权值拷贝到 B 网络, 而后更新 B 网络的操作 (A 网络的 W_m 替换 B 网络的 W_0). TIME 通过重用 ReRAM 阵列网络的方式, 提出了一个

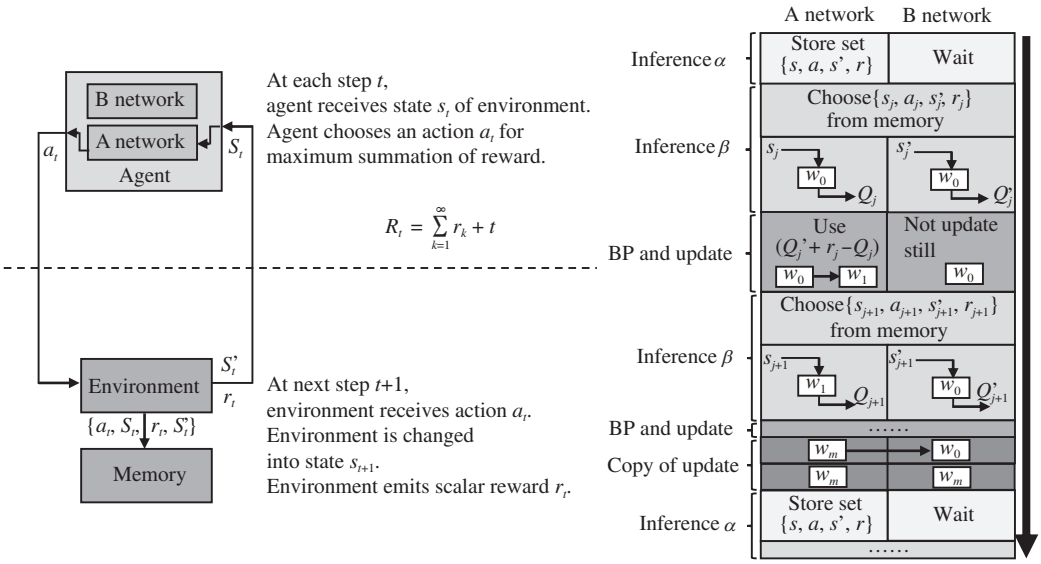

图 29 深度强化学习的推理和训练过程 [43]

Figure 29 The inference and training process of reinforcement learning [43]

特殊的数据映射操作来消除拷贝操作带来的写操作开销. 实验结果显示, 与 ASIC 加速器相比, 针对有监督的神经网络, TIME 能取得 5.3 倍的能耗节约; 针对强化学习网络, TIME 能取得 126 倍的能耗节约.

LerGAN [44] 是一个针对训练对抗生成网络 (GAN) 设计的存内计算系统架构. 与传统 CNN/DNN 不同, 对抗生成网络有两个网络, 并且使用跨步卷积代替原来的池化层. 上述存内计算系统架构直接用于对抗生成网络加速难度很大, 很多零相关的操作占据了大量的存内计算空间, 并且复杂的数据流使得存内计算的片上互联成为瓶颈. 基于此, LerGAN 首先提出了去除零相关的操作, 通过重构卷积核以及相应的数据映射, 能够去除因跨步卷积和外圈补零带来的零相关操作. 另外, 基于 GAN 训练时的数据流结构, LerGAN 还提出了一种三层堆叠的存内计算阵列结构, 分别映射前向传播层、误差传播层, 以及权值计算层, 使得 GAN 训练的数据传输路径变短, 且路由变少. 为了融合这两项技术, LerGAN 使用内存控制器控制数据的映射以及相应的片上互联重配, 以使得数据传输尽可能少且各部分计算速度尽可能一致. 实验显示, 和针对 CNN 的存内计算系统相比, LerGAN 能取得 7.46 倍的性能提升和 7.68 倍的能耗节约.

IBM 的研究人员提出了一种用 PCM+CMOS 的存储单元来做存内计算的方法 [45], 能在同一个阵列中实现全连接神经网络的前向传播、反向传播和权值计算. 图 30 是 PCM+CMOS 的存内计算结构. 图 30(a) 部分是一个存内计算阵列, 包含了多个行. 图 30(b) 是其中一个行的结构, 包含多个存储单元 (图 30(c)) 和一个共享电容单元. 该结构的特殊之处在于图 30(c) 中的存储单元, 该单元由两个 PCM cell (G^+ 和 G^-) 和一个电容器 (g) 组成. 其中, PCM 单元用来存储权值的高位, 正值存在 G^+ 中, 负值存绝对值在 G^- 中; 电容器单元用来存储权值的低位. 在训练时, 权值的高位改变少, 所以使用寿命短且非易失的 PCM 单元来存; 相反, 频繁变化的低位就用电容器单元来存. 图 31 展示了使用该结构训练一个全连接神经网络的过程. M 层作为输入首先进入存内计算的阵列中 (图 31(b) 左侧两个阵列), 输出进入下一层权值所存放的阵列中, 依此类推 (所有实线箭头表示前向的数据流). 前向传播完成后, 在原地进行反向传播 (图中虚线部分标出), 不需要转置权值矩阵. 该结构能支持原地的前

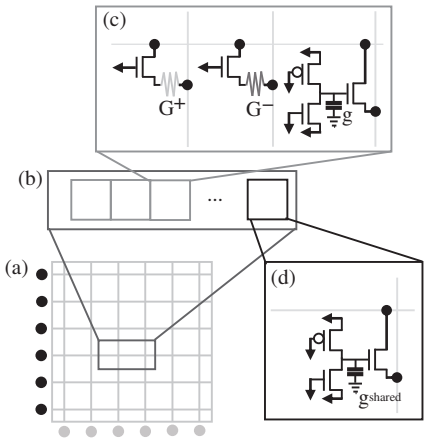

图 30　PCM+CMOS 存内计算阵列 [45]

Figure 30　PIM array based on PCM+CMOS [45]. (a) PIM array; (b) reference cell; (c) PCM+CMOS cell; (d) a shared CMOS cell

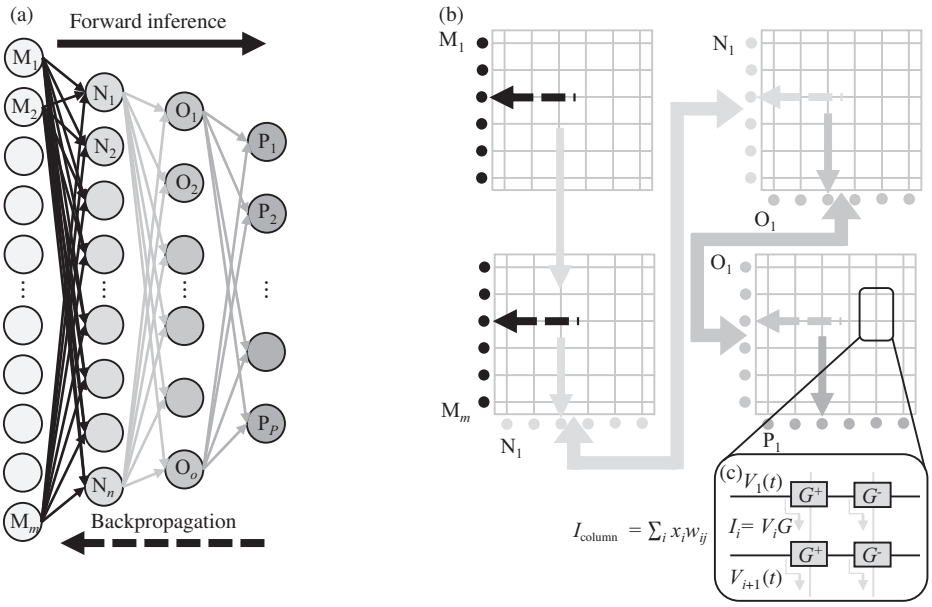

图 31　全连接网络放到 PCM+CMOS 存内计算阵列中执行的示例 [45]

Figure 31　The example of training a fully-connected NN on PCM+CMOS-based PIM [45]. (a) A neural network; (b) a PIM structure

向反向传播, 但不适用于卷积神经网络的训练, 而现在大多数流行的神经网络都有卷积层的计算, 这是此工作的一个局限. 实验结果显示, 相比较于 GPU, 该结构对全连接的网络能有两个数量级的性能提升, 仅伴随不到 1% 的精度损失.

SC [46] 是一个针对科学计算提出的存内计算系统架构. 线性代数在科学计算和工程中普遍存在, 用专门的硬件加速线性代数计算, 有助于提高相关应用的运行速度, 减少能耗. 向量乘矩阵就是线性代数中的一个重要算子. 前述存内计算用于加速向量乘矩阵的系统结构有很大的局限性: 只支持定点的低精度计算, 而科学计算需要全精度的浮点运算支持. SC 通过探索指数分布的局部性, 提供基于

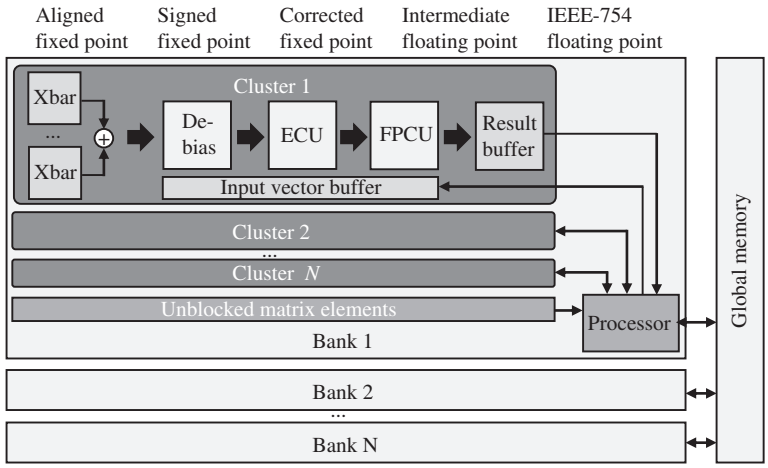

图 32 SC 的系统结构 [46]

Figure 32 The architecture of SC [46]

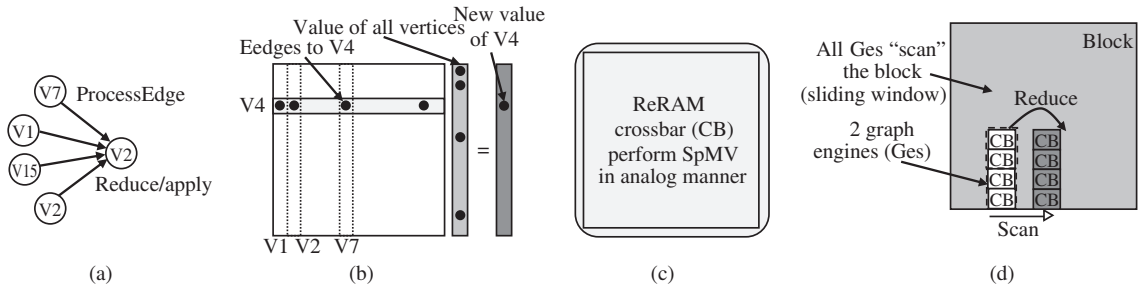

图 33 图计算在 GraphR 上执行的一个示例 [47]

Figure 33 An example of graph computing application executed on GraphR [47]. (a) Vertex program in graph view; (b) vertex program in matrix view; (c) ideal case: CB of $|V| \times |V|$, $|V|$ is the number of vertices in a graph; (d) realistic case: memory ReRAM stores a block of graph; ReRAM Ges process/scan subgraphs (sliding window)

定点计算的浮点计算支持, 提出了支持快速低功耗的全精度浮点数向量乘矩阵的存内计算硬件架构. 图 32 是 SC 的硬件系统结构, 由多个 ReRAM 阵列组成. 每个阵列包含多个集群和一个通用处理器, 可处理 ReRAM 阵列不支持的计算. 每个集群中有大小不同的计算阵列来支持高效的稀疏矩阵计算. SC 结合了现有的 GPU 系统来处理数据, 可广泛应用, 相比于纯 GPU, 能够取得 10.3 倍的性能提升和 10.9 倍的能耗节约.

GraphR [47] 是一个针对图计算提出的存内计算系统架构. GraphR 把一个图分成多个子图, 探索子图之间的并行性, 以提高性能, 并减少因矩阵稀疏性带来的资源浪费. 图 33 展示了一个子图在 ReRAM 阵列中做计算的实例. 该例中, 有一个通过 4 个点和 V4 点连接的程序, 这 4 个点的值用于更新 V4 的值 (图 33(a)). GraphR 先把这个计算转化成图 33(b) 所示的向量乘矩阵操作, 其中矩阵是 V4 的邻接矩阵, 向量是其他点的值, 最终计算得到 V4 的更新值. 最简单的方法就是把此图直接转化为矩阵, 映射到 ReRAM 阵列中做计算, 但是会造成很大的资源浪费. 因此, GraphR 用小 ReRAM 阵列, 例如 4×4 或 8×8 (之前的工作中通常用 64×64 或 128×128), 来组成图处理引擎 (graph-processing engine, GE), 处理和扫描每一个子图. 实验表明, 相比于 CPU, GraphR 能取得 16 倍的性能提升和 34 倍的能耗节约; 相比于 GPU, 有 1.69 到 2.19 倍的性能提升和 4.11 到 8.91 倍的能耗节约; 相比于近数据计算,

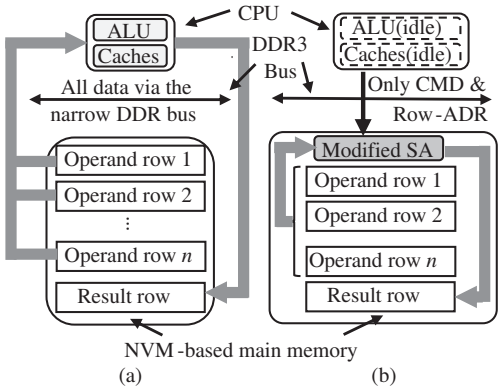

图 34 Pinatubo 和冯 · 诺依曼系统结构比较 [48]

Figure 34 The comparison between Pinatubo and von Neumann architecture [48]. (a) Conventional approach; (b) Pinatubo

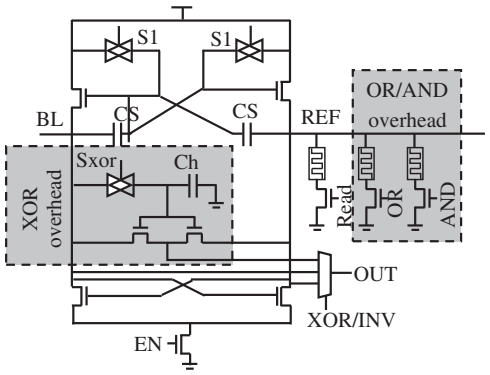

图 35 Pinatubo 的读出放大器 [48]

Figure 35 The sense amplifier of Pinatubo [48]

有 1.16 到 4.12 倍的性能提升和 3.67 到 10.96 倍的能耗节约.

3.2.2 基于逻辑操作的存内计算

基于逻辑操作的存内计算代表工作有: University of California, Santa Barbara 的 Pinatubo [48], Delft University of Technology 的 Scouting Logic [49] 和 XOR/XNOR 存内计算系统 [50], University of California, San Diego 的 MPIM [51] 和 MAPIM [52], 具体如下.

Pinatubo [48] 是一个针对大量比特位操作的存内计算系统架构. 图 34 对比了传统冯 · 诺依曼系统结构和 Pinatubo 的系统结构在执行批量比特位操作的过程. 图 34(a) 中, CPU 先通过有限的总线资源把内存数据读取到 cache 中, 再用 CPU 中的 ALU 单元对数据做计算, 得到结果后将数据通过总线存入到内存中. 图 34(b) 展示的是 Pinatubo 的存内计算操作, CPU 只发送指令和行地址给内存, 内存直接通过读操作将两个参与计算的行读取到修改了的读放大器中, 读放大器可以直接计算出两行操作数的与、或, 以及异或的值, 然后存放到新的行中. 在这个计算操作中, 只有命令和行地址从总线上传输, 避免了传统结构中的大量操作数的传输. 图 35 展示了修改后的能支持数据读取、与、或, 以及异或计算的读放大器. NVM 中数据读取的本质是让给定的电流值经过要读取的电阻后, 和读放大器中的已知的阻值作比较来确定是 0 还是 1. 基于此原理, Pinatubo 同时读取两行或者多行操作数, 在读放大器端加上异或、或、与的参照电路, 通过简单的读操作完成比特位逻辑运算. 每个参照电路都有一个开关, 当指定操作类型后, 相应的参照电路将接入放大器中, 获得最终的结果. 实验结果显示, 对于大量的比特位逻辑运算, Pinatubo 能取得 500 倍的性能提升和 28000 倍的能耗节约; 在普通应用中, 能取得 1.12 倍的性能提升和 1.11 倍的能耗节约.

Scouting Logic [49] 指出存内计算受限于 NVM 有限的寿命; 所有的计算更新都在 NVM 中, 缺乏传统计算机中寿命很长的片上缓存来辅助减少对 NVM 的写操作. 因此, Scouting Logic 提出只通过读操作执行这些逻辑单元, 而不改动 NVM 存储的数据值. 其核心思想与 Pinatubo 一致, 主要是改动了读出放大器的设计, 从而占用面积更小, 性能更高. MPIM [51] 为同时支持逻辑运算和搜索操作运算 (在 3.2.3 小节中介绍) 的存内计算架构, 其中的逻辑运算操作原理与 Pinatubo 相同. 实验显示, 相比于 GPU, MPIM 能取得 19 倍的性能提升和 5.5 倍的能耗节约. Lebdeh 等 [50] 提出了一个针对 XOR

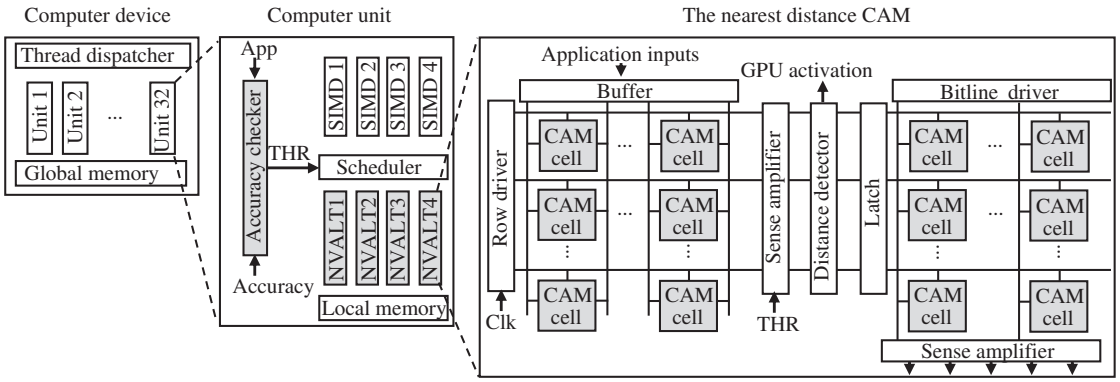

图 36 带有 NVALT 块的 GPU 体系结构 [53]

Figure 36 The architecture of GPU with NVALT block [53]

和 XNOR 操作设计的存内计算系统, 该系统基于两个输入的混合 ReRAM 阵列和 XNOR 门设计, 不需要额外的 ReRAM 阵列和计算, 能够取得 54% 的时间节约和 56% 的能耗节约. MAPIM [52] 指出之前针对批量位逻辑运算的存内计算架构没有考虑并行, 使得存内计算未取得潜在的高性能. 因此, MAPIM 提出基于阵列并行的高性能存内计算系统架构, 能够支持多个比特线的请求, 并且共享支持位逻辑运算的读放大器, 使得占面积大的读放大器利用率高, 从而提升整体系统结构的性能. 与之前的存内计算系统相比, MAPIM 能够取得 16 倍的性能提升和 1.8 倍的能耗节约.

3.2.3 基于搜索操作的存内计算

基于搜索操作的存内计算代表工作有: University of California, San Diego 的 NVALT [53] 和 NVQuery [54], University of California, Irvine 的 MAP [55], Tsinghua University 的 SQL-PIM [56], 具体如下.

NVALT [53] 是一个基于存内计算设计的近似查找表, 专门用于加速 GPU. GPU 的应用展现出了非常高的数据相似性和局部性, 如 FFT (fast fourier transform) 和图像处理, 由重复的包含很多乘加操作的块构成. NVALT 通过探寻这些应用的数据局部性, 对这些基础应用建立高效的近似功能单元, 来加速 GPU 的计算. 图 36 是集成有 NVALT 的 GPU 架构. NVALT 块放置在每一个单指令多数据流 (single instruction multiple data, SIMD) 处理通道的旁边. 当应用在 GPU 上执行时, 首先经过精度核查, 精度允许的条件下, 调度器会把指令放到 NVALT 块上执行. NVALT 块使用线下预处理的方式, 识别并存储每个程序常用的数据输入模式和对应的数据输出模式. 运行时, NVATL 搜索存储在 CAM 里的输入数据, 然后返回和输入模式最相似的条目所对应的输出结果. 系统可根据用户的不同精度需求调度近似的 NVALT 核和精确的 GPU 核. 实验显示, 在精度损失控制在 10% 之内的情况下, NVALT 平均能取得 4.5 倍的能耗节约和 5.7 倍的性能提升.

MAP [55] 是一个基于存内计算的近似计算协处理器. 图 37 是 MAP 的系统结构, 配备有基于忆阻器的内容寻址存储 (resistive content addressable memory, RCAM)、控制器、指令缓存器和一些专用寄存器 (例如键值寄存器、掩码寄存器、标志寄存器). 指令所需要的数据全部存储在 RCAM 中. RCAM 用两个忆阻器 cell 来存储一比特位的正负部分. 系统运行时, 指令寄存器先把指令发送到控制器, 控制器生成相应的掩码和键放到寄存器中. 键寄存器用来存放被写或者被比较的键值, 而掩码寄存器用来显示哪些比特位在被写或被比较时激活. 当执行比较操作时, 比较电路找出和给出的键以及掩码相

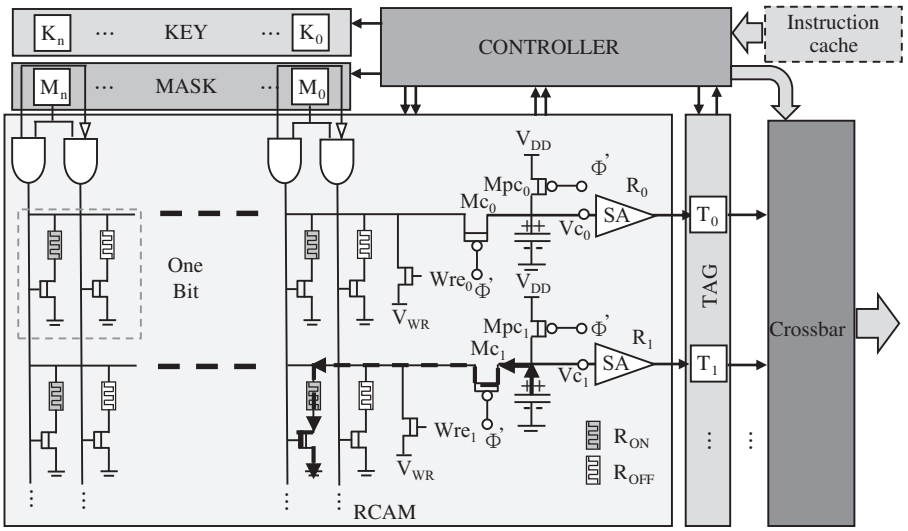

图 37　MAP 的体系结构 [55]

Figure 37　The architecture of MAP [55]

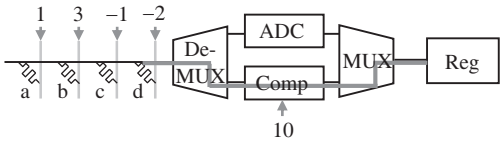

图 38　一个限制查询语句在 SQL-PIM 中实现的例子 [56]

Figure 38　An example of restriction operation in SQL-PIM [56]

吻合的行, 然后做标记并存储到内存中. 由于高度并行, 查找一个 512 行的表只需大约 2 ns. 实验显示, 和传统的冯 · 诺依曼架构相比, MAP 能取得 80 倍的能耗节约和 20 倍的性能提升.

Sun 等 [56] 提出了一个针对关系型数据库的存内计算系统结构 (简称 SQL-PIM). 在数据库应用里, 该结构的存储部分既支持从表中直接读行的操作, 又可以支持直接读取列的操作, 减少了传统计算机中片上缓存不命中带来的时间和能耗的开销. SQL-PIM 实现了限制查询语句, 规划查询语句和聚合查询语句. 限制查询语句是找出表中符合给出规定的一系列数据, 这些规定可以是数值逻辑或者是非逻辑的条件语句, 通常用 WHERE 语法来操作; 规划查询语句是找出表里含有特定参数的条目或者特定的列, 通常用 SELECT 语法进行操作; 聚合查询语句是对一些给定条件的条目做加操作, 通常用类似 SUM 语法来求一系列值的和.

图 38 展示了一个 SQL-PIM 实现的限制查询语句的实例: select * from Table where $(a+3\times b)-(c+2\times d)>10$. 其中, $\{a,b,c,d\}$ 是表中四列数, 存储在 ReRAM 阵列中. $\{a,b,c,d\}$ 前面的系数 $\{1,3,-1,4\}$ 以电流方式加到比特线上. 最后, 结果电流通过包含存有 10 的比较电路, 输出 0/1. 结果为 1 的, 就是符合限制查询语句的条目.

图 39 是 SQL-PIM 的结构, 分为用于存放表格条目的 PIM 部分和用于支持比较等操作的外围电路部分. 指令通过控制器后, 将相应的参数发送到这两个模块上, 然后两个模块通过共享的缓存进行中间结果的传输. 最后, 结果写回到 ReRAM 中. 除此之外, SQL-PIM 还能在不改变结构化存储的前

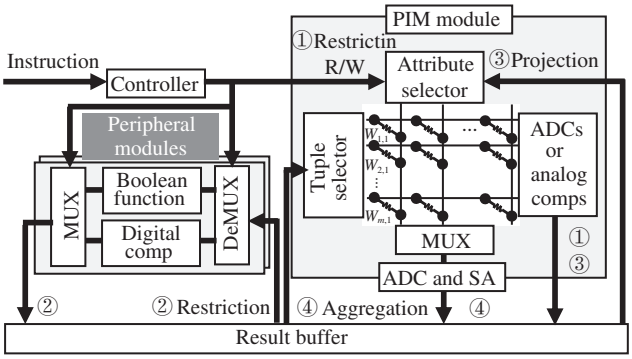

图 39 基于 ReRAM 的 SQL 存内查询结构 [56]

Figure 39 The structure of ReRAM-based PIM for SQL query [56]

提下支持增、删、改、查操作. 针对大的数据库表, SQL-PIM 提出了一个特殊关联分割的方法, 将大表存储在多个存内计算阵列中, 同时减少每个计算阵列之间的相互通信. 实验显示, 与传统的内存数据库相比, SQL-PIM 能节约 4∼6 个数量级的能耗.

NVQuery [54] 也是利用 RCAM 支持多种查询语句的存内计算加速器, 其系统结构和 MAP [55] 相像. NVQuery 能够支持聚合、预测、按位操作, 以及精确的最近距离查找. 为了支持最近距离查找, NVQuery 提出了比特线驱动的策略, 将权重加到相应的比特位上. 实验显示, 与传统的冯 · 诺依曼系统结构相比, NVQuery 带来 49.3 倍的性能提升和 32.9 倍的能耗节约.

3.2.4 存内计算小结

存内计算支持的算子较少, 设计灵活度不如近数据计算的逻辑层, 但是存内计算用于支持特定算子 (目前主要是向量乘矩阵算子) 的性能很高且能耗低. 存内计算的核心思路是利用新型存储的物理结构和特性来支持应用程序中频繁出现的算子. 同时, 存内计算相关研究还关注: 存内计算模块互联和数据流的设计; 数据映射策略; 外围电路的优化和复用; 与现有存储系统的融合.

3.3 内存计算架构与技术小结

内存计算面向的应用有如下特点: (1) 数据密集, 在普通冯 · 诺依曼结构中有大量的内存访问; (2) 数据局部性差, 片上缓存命中率低; (3) 计算密集且计算形式简单, 易于并行. 内存计算包含两大类: 近数据计算和存内计算. 近数据计算通常使用 3D 堆叠的内存结构, 在内存中集成计算逻辑芯片, 并用高速通道将计算单元和内存单元相连接, 存算依然分离. 存内计算直接使用内存单元做计算, 利用电流、电压、电阻等物理量之间的关系表示某类计算. 近数据计算相比于存内计算灵活度更高, 能支持较多算子; 存内计算虽然能支持的算子较为单一, 目前能支持向量乘矩阵算子, 按位逻辑操作, 或者搜索操作, 但是其计算速度快且能耗低. 设计内存计算架构时, 设计者需要根据应用场景的需求 (应用中算子的类型、延迟和能耗的限制等) 进行选择, 必要时也可结合使用近数据计算和存内计算两种技术, 充分利用两者优点.

4 内存计算面临的挑战

内存计算设计时需要考虑计算机中多个层级, 如图 40 所示. 内存计算本身存在一些问题, 与现有

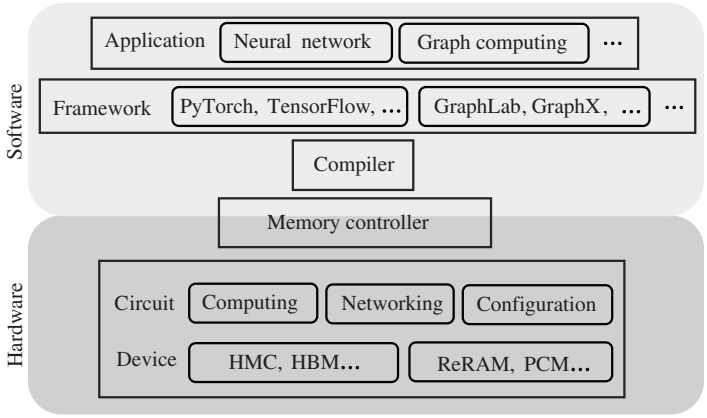

图 40　内存计算设计中各层级示意图

Figure 40　The description of various levels in the PIM design

的系统相结合也会带来新的问题, 每个具体问题都涉及图 40 中的某些层级. 解决这些问题是内存计算可以在计算机系统中发挥作用, 提升系统性能和降低能耗的关键. 本节将分析并总结近数据计算和存内计算本身及其与现有系统集成会带来的挑战.

4.1　近数据计算面临的挑战

4.1.1　高能耗问题

在近数据计算系统中, 对程序模块的分割不当或者过度使用近数据计算来处理数据反而会增加系统能耗. 该问题的产生通常与编译器的设计和内存控制器的设计相关. Kim 等 [57] 分析了集成有传统处理器和 NDC cube 的系统的能耗特征, 发现 LLC MPKI (misses per kilo instructions) 是一个决定应用是否能在近数据系统中高效执行的关键特征: 通常高 MPKI 表示可从近数据计算系统中获得高的能耗节约. 因此, 他们认为应将高 MPKI 的计算放到 NDC cube 中做, 把其余部分留在传统系统中做.

4.1.2　硬件配置问题

因为近数据计算的 NDC cube 中逻辑芯片通常使用 FPGA, CGRA, 和 ASIC 处理器等非通用处理器, 所以硬件配置与应用需求的匹配成为一个挑战. 该问题通常与电路设计和器件选择强相关. Gao 等 [14] 提出了一种异构且可重构的逻辑阵列, 同时利用 FGPA 和 CGRA. 通过分析适合近数据计算处理的大量应用, 结合两者长处, 设计 FPGA 和 CGRA 以及逻辑层的多路选择器的个数, 以满足近数据计算应用的高性能低能耗的需求. 针对特定应用场景或者特定应用, 如何配置近数据计算中的逻辑芯片部件仍有待探索和研究.

4.1.3　计算资源利用率问题

近数据计算的计算资源基本均匀分布在整个内存中, 直接使用传统的数据存储方式会导致计算资源分配不均匀, 计算核与其所需的数据相距远等问题, 进而导致内存计算的计算资源利用率低. 该问题的产生通常与编译器设计和内存控制器设计相关. 内存中数据的存储需要考虑到计算资源的利用率, 尽量使所有计算资源能够时分复用, 管道串行, 要避免因计算资源集中在某一块使用而导致长时间的数据等待.

对于特定的存内计算架构, 通常采取设计合适的数据映射策略来提高计算资源的利用率的思路.

而数据映射策略取决于不同的内存计算系统架构和运行在其上的应用 [13]. 因此, 在设计上层系统时, 需要同时考虑两者来设计合适的数据和计算核的排布.

4.1.4 缺少灵活的系统支持

运行在近数据计算系统中的应用需要在中央处理器和近数据计算端来回切换, 因此, 相应的系统应支持这样的灵活切换. 目前的近数据计算都是针对特定的硬件结构或特定应用提供的上层软件支持 [13], 灵活性不够.

近数据计算运用到系统中时, 还需要考虑近数据计算中的多任务调度以及近数据计算任务和传统访存任务的调度问题. 首先, 由于近数据计算的计算资源均匀分布在内存中, 一个好的多任务调度策略能使计算资源利用率高, 同时减少系统中的冲突和中断. 其次, 在不处理内存计算任务时, 近数据计算模块也可以用作普通存储模块, 因此, 需要一个内存计算任务和传统任务的调度策略, 来优化内存计算和传统存储混合的系统结构性能.

另外, 近数据计算中有多个计算模块, 各计算模块之间可能会共享数据, 包括并发地对数据进行增删改查的操作. 因此, 近数据计算需要一个解决冲突问题的方案, 以保证其数据的一致性.

以上问题都属于近数据计算的系统问题, 通常与系统级的设计强相关, 也与编译器和内存控制器相关.

4.2 存内计算面临的挑战

4.2.1 硬件寿命问题

NVM 的寿命有限, 例如 PCM 的 SLC (single level cell, 一个 cell 只能存 0 或 1, 即一个比特位) 的寿命只有 $10^7 \sim 10^8$, ReRAM 的 SLC 的寿命只有 $10^7 \sim 10^9$ [17,58,59]. MLC (multi-level cell, 一个 cell 能存多个比特位) 的寿命问题更加严重, 通常只有 $10^4 \sim 10^5$ 次写, 甚至更低. 对于传统 NVM 存储, 磨损均衡是延长寿命的有效方法. 磨损均衡算法通过每隔一段时间改变逻辑地址到物理地址的映射, 使得写操作在整个 NVM 中均衡. 然而这种方法在基于 NVM 的存内计算中并不适用, 因为存储于 NVM 中的数据还直接用作计算. 如果直接使用传统的磨损均衡算法交换数据所存储的位置, 计算结果将是错误的. 该问题由器件相关问题引起, 除了选择和配置合适的器件外, 还可以通过上层设计来缓解, 例如应用中算法的设计和内存控制器的设计等.

ISAAC [40] 通过在片上加 eDRAM 减少对 NVM 的写. IBM 的研究人员 [45] 通过用 CMOS+PCM 做一个 cell 的方式, 使寿命的 CMOS 单元承受频繁的更新操作. Long-live-time [60] 提出了一种针对神经网络训练的 CIM 硬件寿命延长方法, 通过改变神经网络权值更新方法 (每次选误差最大的行更新而不是全部更新), 再结合行粒度的磨损均衡算法, 延长基于 NVM 的存内计算硬件的寿命. 我们正在进行的工作将神经网络和 NVM 的特点综合考虑, 从而延长基于 NVM 的存内计算硬件寿命. 针对其余应用的 NVM 存内计算硬件寿命的延长方法仍待探究.

4.2.2 可靠性问题

NVM 写出错问题以及外围电路对输出模拟域电流转成电信号产生误差的问题使基于 NVM 的存内计算可靠性不佳. NVM 的 cell 会因为写电流过高或寿命已到而产生 stuck-at fault (阻值停留在某个固定值不可改变). 传统存储中, 可以将发生错误的 cell 值存到别的物理位置, 然后改变原逻辑地址到物理地址的映射来容这种错误. 而这种方式在基于 NVM 的存内计算中并不适用, 存储在存内计算的 NVM 中的数据还要直接用作计算, 数据之间的相对物理位置不能被改变, 否则计算结果会出错. 与

寿命问题相似, 可靠性问题也是由器件相关问题引起的, 除了选择合适的器件外, 还可以通过上层设计来缓解.

Xia 等[61] 利用神经网络中权值的稀疏性 (一些位置的权值为 0) 来容存内计算 NVM 上 stuck-at-0 的硬件错误. Xia 等[62] 还通过利用存正负值的一对存内计算 NVM 阵列来互相容错. Liu 等[63] 提出分析识别出神经网络中重要的部分, 把此部分放到可靠性高的存内计算硬件上去做. 我们正在进行的工作将采取更加灵活的方式, 综合利用神经网络和 NVM 本身的特点来容更多类型的 stuck-at 错误. 由于外围电路的误差而造成的可靠性降低问题仍待解决.

4.2.3 数据精度问题

每个 NVM 的 MLC 能存的比特位有限 (目前最高可存 7 比特位)[12], 这样的数据精度在神经网络中通常不够使用. 因此, 很多基于 NVM 的存内计算用多个 MLC 或者 SLC 来存一个数据[6,40], 但这样会产生较大的支持中间数据计算的外围电路开销. 不仅如此, 基于 NVM 的存内计算目前不支持浮点数计算, 所有计算数据都先要转换成定点数后再做计算. 这样会带来结果精确度损失甚至结果不正确的问题. 而一些应用, 例如大规模的神经网络训练, 需要高精度的数据来保证网络的高准确度. 因此, 可以通过软硬件层协同设计, 来支持高精度浮点数的运算.

4.2.4 高能耗问题

由于 NVM 的写能耗比 DRAM 高, 而基于 NVM 的存内计算有大量的原地更新, 在没有缓存的帮助下会产生大量的 NVM 写操作, 这使得基于 NVM 的存内计算系统能耗过高. 同时, 用于支持电模转换 (ADC, DAC) 的外围电路能耗过大, 也会使得存内计算系统能耗高. 该问题与器件层强相关, 可以选择写能耗较低的器件, 还可以通过上层的设计减少器件的写能耗开销.

Mao 等[44] 探索神经网络中数据的结构化稀疏性, 在算法和编译器层面, 通过重构数据块的方式去掉与零操作相关的数据, 从而减少 NVM 的写能耗. 在应用算法和电路层面, Ni 等[64] 通过二值化的 ReRAM 存内计算阵列减少电模信号相互转换的能耗. Wang 等[65] 使用脉冲神经网络 (spiking neural network, SNN), 在应用算法和电路层面, 减少电模信号的相互转换能耗, 还改变了 SNN 训练的算法来减少取样的能耗. Narayanan 等[66] 在使用 SNN 降低能耗的同时, 在内存控制器层面, 探索神经网络中计算的并行性来提高针对 SNN 的存内计算的吞吐. Ankit 等[67] 在电路设计层面, 利用低能耗的 SNN 实现了一个可重构的连接, 为 SNN 中计算的数据流服务, 从而降低基于 NVM 的存内计算的能耗. Xia 等[68] 通过分析数据的分布, 在应用算法层面, 提出新的数据量化方法, 将中间数据转化成 1 比特位的数据从而取消 DAC, 另外选择必要的输入信号减少输出的个数, 从而减少 ADC 的能耗.

4.2.5 外围电路占用面积过大问题

基于 NVM 的存内计算中, 外围电路的 ADC 和 DAC 不仅能耗占比大, 还占用了很大的片上面积[40,44]. 此外, 用于中间数据处理的外围电路也占了存内计算的较大一部分面积. 尽管 NVM 有很高的存储密度, 当用作存内计算的计算核时, 还需要大面积的外围电路支撑, 这使得整个结构占用面积大. 该问题主要与电路的设计相关.

Mao 等[69] 通过 3D 堆叠的方式, 将外围电路放到最下层做成一个外围电路资源共享池, 由堆叠在其上的 NVM 存内计算阵列共享. 基于此, 通过时分复用的方式, 极大地减少了外围电路的使用, 从而减小了基于 NVM 的存内计算架构的面积.

4.2.6 硬件配置问题

存内计算中存储单元只用于支持某些特定的算子, 其余的计算需要外围电路的支撑. 这给外围电路的设计带来了很大的压力: 如果能支持的运算多, 就会带来外围电路面积过大, 利用率不高的问题; 如果能支持的运算少, 就会使得该存内计算芯片适用范围小. 如针对神经网络的存内计算架构 PRIME, 外围电路只支持 Sigmoid 激活函数 [12], 需要其他激活函数的神经网络在该架构中无法执行. 这很大程度限制了存内计算的使用. 该问题也与电路的设计强相关.

Ji 等 [70] 提出了一种可重构的基于 ReRAM 的存内计算阵列 FPSA. 他们提供了脉冲神经阵列和可重构的逻辑阵列, 使得神经网络计算核到存内计算硬件的映射变得灵活, 并以可重构的连接来灵活支持计算时数据流. 用于其他应用的存内计算硬件配置问题仍待研究.

4.2.7 计算资源利用率问题

和近数据计算一样, 存内计算的计算资源也分布在整个内存中. 因此数据存储需要考虑计算资源的利用率, 在编译器层面和内存控制器层面, 针对具体的存内计算系统架构和运行在其上的应用, 设计对应的数据映射策略 [44].

4.2.8 缺少灵活的系统支持

存内计算模块作为有一定计算能力的存储模块, 在集成到现有的混合存储系统时, 需要考虑不同的集成方式. 例如基于非易失性存储的内存计算模块可以和传统的 DRAM 并列接到总线上, 也可以和纯 NVM 存储以及 DRAM 存储并列连接到总线上, 而后通过内存控制器来管理. 除与传统混合存储相结合外, 内存计算模块还可作为 CPU 的协处理器. 对于这些架构, 存内计算模块亟需一套相对应的软硬件接口支持, 以保证其高效且灵活的应用 [12, 44].

除此之外, 和近数据计算一样, 存内计算运用到系统中时, 也需要考虑多任务调度以及存内计算任务和传统任务的调度问题, 同时要保证存内计算中数据的一致性. 该挑战主要涉及系统级的设计和内存控制器的设计.

4.2.9 缺少与现有存储系统适配的能力

内存计算模块不仅用作存储, 还用作计算 [12]. 存储于内存计算中的数据需要以某种特定的排布来配合其中配置好的数据流, 从而达到高性能计算的目的 [44]. 传统面向内存的虚拟地址空间管理会破坏内存计算中的特定数据排布, 并不适用于内存计算模块. 因此, 内存计算亟需相应的虚拟地址空间的支持, 同时还需要和传统存储相配合的访存模式, 以保证上层应用使用内存计算模块的安全性. 该挑战主要与系统级的设计和内存控制器的设计相关.

5 内存计算所提供的机遇

内存计算技术在处理存储密集型、数据局部性差、且计算形式较为单一、易于并行的应用上优势明显. 目前流行的 3 类应用: 机器学习, 图计算和基因工程, 在传统系统中因总线带宽有限和片上片下数据移动太频繁而性能低且能耗高. 内存计算技术恰好弥补了传统计算机体系结构在这 3 种流行应用上的缺陷. 本节主要介绍内存计算给这 3 种应用带来的机遇.

5.1 机器学习

机器学习的核心是神经网络 [71]. 神经网络类型很多, 包括: 卷积神经网络 [72], 主要用于图像处理; 循环神经网络 [73], 主要用于自然语言处理; 深度神经网络是卷积网络或者循环神经网络的强化版 [74],

有更多的网络层; 生成对抗网络 [75] 是无监督学习中的最受关注的网络类型; 还有深度强化学习 [76], 是一种在线学习方式, 基于环境的改变而调整自身的行为. 这些网络各具特点, 大的网络会对存储造成很大的压力, 连接多的网络会给计算和传输造成很大的压力. 他们的共同点在于都有很多向量乘矩阵操作, 占到所有计算的 90% 以上 [41]. 因此基于 NVM 的存内计算适合用于支持神经网络计算.

5.2 图计算

随着互联网数据爆炸式的增长以及人们对数据单元之间关系的关注, 图计算在新型应用中占有较大地位 [77~81]. 图计算的应用包括: 网络安全 [82]、社交媒体 [83]、网页评分和引用排序 [84]、自然语言处理 [85]、系统生物学 [86]、推荐系统 [87] 等. 由于图计算应用的数据局部性差且数据量很大, 目前用来处理图计算的系统面临性能不佳且能耗过高的难题.

图计算中大量操作都可以转换成矩阵乘的形式, 因此可以用基于 NVM 的存内计算来处理. 综合考虑图数据的预处理, 稀疏矩阵的分隔和映射, 以及硬件控制和数据流设计, 能够高效且低能耗地支持图计算应用. 图计算也可以用基于 HMC 的 NDC 来处理, 通过分析现在图计算中哪些操作适合放到 HMC 中处理, 然后有针对性地设计逻辑层和上层的指令集, 能取得较高的性能提升和能耗节约. 所以内存计算给图计算的发展提供了很好的机遇.

5.3 基因工程

近年来, 随着基因工程的发展, 生物学信息数据呈指数级增长 [88, 89]. 这种生物数据的暴增给现在诸如基因序列匹配的应用带来了很大的挑战 [90, 91]. 目前有一些针对基因序列查找的软件方面的加速方法 [92, 93], 也有利用基因处理的高并行性的硬件加速方法 [94, 95]. 但是由于生物数据的量太过庞大, 传统计算机系统片上片下数移动量太大, 系统能耗是一个很大的问题 [94, 95]. 基于 RCAM 的存内计算能利用 CAM 的极速查找能力, 提供很高的硬件并行度, 同时在存内处理能降低数据移动的能耗, 适用于大规模生物数据处理. 因此, 存内计算给基因工程的发展提供了机遇.

6 总结

在数据爆炸时代, 内存计算技术为解决传统冯 · 诺依曼架构中总线拥堵问题以及片上片下数据传输能耗过高问题提供了解决方案. 内存计算技术得益于新型 3D 堆叠技术和非易失存储技术的发展. 内存计算主要有两种形式: 近数据计算和存内计算. 近数据计算技术通常依赖于 3D 堆叠的内存结构, 存算依然分离, 但计算部分到存储部分带宽较大, 计算部分灵活性较高. 存内计算技术通常依赖于新型的非易失存储, 直接利用存储来做计算, 存算紧耦合, 能够为特定算子 (例如向量乘矩阵) 提供极高性能的计算, 但缺乏一定的灵活性. 内存计算技术总体还处于发展初期, 有一些硬件和软件支持上的问题. 但是内存计算的潜力很大, 能够给目前流行的机器学习应用, 图计算应用, 和基因工程提供高效低能耗的系统结构支持.

参考文献

1 Mutlu O, Ghose S, Gómez-Luna J, et al. Processing data where it makes sense: enabling in-memory computation. Microprocessors Microsyst, 2019, 67: 28–41

2 Mutlu O, Ghose S, Gómez-Luna J, et al. Enabling practical processing in and near memory for data-intensive computing. In: Proceedings of the 56th Annual Design Automation Conference 2019, Las Vegas, 2019. 1–4

3 Singh G, Gómez-Luna J, Mariani G, et al. NAPEL: near-memory computing application performance prediction via ensemble learning. In: Proceedings of the 56th ACM/IEEE Design Automation Conference (DAC), Las Vegas, 2019. 1–6
4 Boroumand A, Ghose S, Patel M, et al. CoNDA: efficient cache coherence support for near-data accelerators. In: Proceedings of the 46th International Symposium on Computer Architecture, Phoenix Arizona, 2019. 629–642
5 Ghose S, Boroumand A, Kim J S, et al. Processing-in-memory: a workload-driven perspective. IBM J Res Dev, 2019, 63: 1–19
6 Song L, Qian X, Li H, et al. Pipelayer: a pipelined reram-based accelerator for deep learning. In: Proceedings of 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA). Austin: IEEE, 2017. 541–552
7 Farmahini-Farahani A, Ahn J H, Morrow K, et al. NDA: near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules. In: Proceedings of 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA). Burlingame: IEEE, 2015. 283–295
8 Springer R, Lowenthal K D, Rountree B, et al. Minimizing execution time in MPI programs on an energy-constrained, power-scalable cluster. In: Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York: ACM, 2006. 230–238
9 Xiao P, Han N. A novel power-conscious scheduling algorithm for data-intensive precedence-constrained applications in cloud environments. Int J High Performance Comput Netw, 2014, 7: 299–306
10 Patki T, Lowenthal K D, Rountree B, et al. Exploring hardware overprovisioning in power-constrained, high performance computing. In: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing. Eugene Oregon: ACM, 2013. 173–182
11 Pugsley S H, Jestes J, Balasubramonian R, et al. Comparing implementations of near-data computing with in-memory mapreduce workloads. IEEE Micro, 2014, 34: 44–52
12 Chi P, Li S, Xu C, et al. Prime: a novel processing-in-memory architecture for neural network computation in reram-based main memory. SIGARCH Comput Archit News, 2016, 44: 27–39
13 Ahn J, Yoo S, Mutlu O, et al. PIM-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In: Proceedings of 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA). Portland: IEEE, 2015. 336–348
14 Gao M, Kozyrakis C. HRL: efficient and flexible reconfigurable logic for near-data processing. In: Proceedings of 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA). Barcelona: IEEE, 2016. 126–137
15 Zhang D, Jayasena N, Lyashevsky A, et al. TOP-PIM: throughput-oriented programmable processing in memory. In: Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing. Vancouver: ACM, 2014. 85–98
16 Hsieh K, Ebrahimi E, Kim G, et al. Transparent offloading and mapping (TOM): enabling programmer-transparent near-data processing in GPU systems. In: Proceedings of ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 2016. 44: 204–216
17 Xu C, Niu D, Muralimanohar N, et al. Understanding the trade-offs in multi-level cell ReRAM memory design. In: Proceedings of 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC). Ausin: IEEE, 2013. 1–6
18 Gokhale M, Holmes B, Iobst K. Processing in memory: the Terasys massively parallel PIM array. Computer, 1995, 28: 23–31
19 LeCun Y, Bengio Y, Hinton G. Deep learning. Nature, 2015, 521: 436–444
20 Goodfellow I, Bengio Y, Courville A. Deep Learning. Cambridge: MIT Press, 2016
21 Soomro T R. Google translation service issues: religious text perspective. J Global Res Comput Sci, 2013, 4: 40–43
22 Vazquez-Calvo B, Zhang L T, Pascual M, et al. Fan translation of games, anime, and fanfiction. Language, Learn Tech, 2019, 23: 49–71
23 Li C, Qouneh A, Li T. iSwitch. SIGARCH Comput Archit News, 2012, 40: 512–523
24 Li C, Zhou R, Li T. Enabling distributed generation powered sustainable high-performance data center. In: Proceedings of 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA). Shenzhen: IEEE, 2013. 35–46
25 Li C, Zhang W, Cho C, et al. SolarCore: solar energy driven multi-core architecture power management. In: Proceedings of 2011 IEEE 17th International Symposium on High Performance Computer Architecture. San Antonio: IEEE,

2011. 205–216
26 Hadidi R, Asgari B, Mudassar B A, et al. Demystifying the characteristics of 3D-stacked memories: a case study for hybrid memory cube. In: Proceedings of 2017 IEEE International Symposium on Workload Characterization (IISWC). Seattle: IEEE, 2017. 66–75
27 Pei J, Deng L, Song S, et al. Towards artificial general intelligence with hybrid Tianjic chip architecture. Nature, 2019, 572: 106–111
28 Farmahini-Farahani A, Ho Ahn J, Morrow K, et al. DRAMA: an architecture for accelerated processing near memory. IEEE Comput Arch Lett, 2015, 14: 26–29
29 Nair R, Antao S F, Bertolli C, et al. Active memory cube: a processing-in-memory architecture for exascale systems. IBM J Res Dev, 2015, 59: 1–14
30 Vermij E, Hagleitner C, Fiorin L, et al. An architecture for near-data processing systems. In: Proceedings of the ACM International Conference on Computing Frontiers. Como: ACM, 2016. 357–360
31 Liu Z, Calciu I, Herlihy M, et al. Concurrent data structures for near-memory computing. In: Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures. Washington: ACM, 2017. 235–245
32 Yazdanbakhsh A, Song C, Sacks J, et al. In-DRAM near-data approximate acceleration for GPUs. In: Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques. Limassol Cyprus: ACM, 2018. 34
33 Wang Y, Han Y, Zhang L, et al. ProPRAM: exploiting the transparent logic resources in non-volatile memory for near data computing. In: Proceedings of the 52nd Annual Design Automation Conference. San Francisco: ACM, 2015. 47
34 Lee J H, Sim J, Kim H. SSync: processing near memory for machine learning workloads with bounded staleness consistency models. In: Proceedings of 2015 International Conference on Parallel Architecture and Compilation (PACT), San Francisco: IEEE, 2015. 241–252
35 Kim D, Kung J, Chai S, et al. Neurocube: a programmable digital neuromorphic architecture with high-density 3D memory. In: Proceedings of 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA). Seoul: IEEE, 2016. 380–392
36 Aga S, Jayasena N, Ignatowski M. Co-ML: a case for collaborative ML acceleration using near-data processing. In: Proceedings of the International Symposium on Memory Systems. Alexandria: ACM, 2019. 506–517
37 Ahn J, Hong S, Yoo S, et al. A scalable processing-in-memory accelerator for parallel graph processing. SIGARCH Comput Archit News, 2016, 43: 105–117
38 Nai L, Hadidi R, Sim J, et al. GraphPIM: enabling instruction-level PIM offloading in graph computing frameworks. In: Proceedings of 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA). Austin: IEEE, 2017. 457–468
39 Jang J, Heo J, Lee Y, et al. Charon: specialized near-memory processing architecture for clearing dead objects in memory. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture. Columbus: ACM, 2019. 726–739
40 Shafiee A, Nag A, Muralimanohar N, et al. ISAAC: a convolutional neural network accelerator with in-situ analog arithmetic in crossbars. SIGARCH Comput Archit News, 2016, 44: 14–26
41 Chen Y H, Krishna T, Emer J S, et al. Eyeriss: an energy-efficient reconfigurable accelerator for deep convolutional neural networks. IEEE J Solid-State Circ, 2017, 52: 127–138
42 Hu M, Strachan J P, Li Z, et al. Dot-product engine for neuromorphic computing: programming 1T1M crossbar to accelerate matrix-vector multiplication. In: Proceedings of the 53rd annual design automation conference. Austin: ACM, 2016. 19
43 Cheng M, Xia L, Zhu Z, et al. Time: a training-in-memory architecture for memristor-based deep neural networks. In: Proceedings of the 54th Annual Design Automation Conference. Austin: ACM, 2017. 26
44 Mao H, Song M, Li T, et al. LerGAN: a zero-free, low data movement and PIM-based GAN architecture. In: Proceedings of 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). Fukuoka: IEEE, 2018. 669–681
45 Ambrogio S, Narayanan P, Tsai H, et al. Equivalent-accuracy accelerated neural-network training using analogue memory. Nature, 2018, 558: 60–67
46 Feinberg B, Vengalam U K R, Whitehair N, et al. Enabling scientific computing on memristive accelerators. In: Proceedings of 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA). Los Angeles:

IEEE, 2018. 367–382
47 Song L, Zhuo Y, Qian X, et al. GraphR: accelerating graph processing using ReRAM. In: Proceedings of 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA). Vienna: IEEE, 2018. 531–543
48 Li S, Xu C, Zou Q, et al. Pinatubo: a processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In: Proceedings of the 53rd Annual Design Automation Conference. Austin: ACM, 2016. 173
49 Xie L, Nguyen H A D, Yu J, et al. Scouting logic: a novel memristor-based logic design for resistive computing. In: Proceedings of 2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI). Bochum: IEEE, 2017. 176–181
50 Lebdeh M A, Abunahla H, Mohammad B, et al. An efficient heterogeneous memristive XNOR for in-memory computing. IEEE Trans Circ Syst I, 2017, 64: 2427–2437
51 Imani M, Kim Y, Rosing T. MPIM: multi-purpose in-memory processing using configurable resistive memory. In: Proceedings of 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC). Makuhari Messe: IEEE, 2017. 757–763
52 Sim J, Kim M, Kim Y, et al. MAPIM: mat parallelism for high performance processing in non-volatile memory architecture. In: Proceedings of the 20th International Symposium on Quality Electronic Design (ISQED). Santa Clara: IEEE, 2019. 145–150
53 Imani M, Peroni D, Rosing T. Nvalt: nonvolatile approximate lookup table for GPU acceleration. IEEE Embedded Syst Lett, 2018, 10: 14–17
54 Imani M, Gupta S, Arredondo A, et al. Efficient query processing in crossbar memory. In: Proceedings of 2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED). Taipei: IEEE, 2017. 1–6
55 Yantir H E, Eltawil A M, Kurdahi F J. Approximate memristive in-memory computing. ACM Trans Embed Comput Syst, 2017, 16: 1–18
56 Sun Y, Wang Y, Yang H. Energy-efficient SQL query exploiting RRAM-based process-in-memory structure. In: Proceedings of 2017 IEEE 6th Non-Volatile Memory Systems and Applications Symposium (NVMSA). Hsinchu: IEEE, 2017. 1–6
57 Kim H, Kim H, Yalamanchili S, et al. Understanding energy aspects of processing-near-memory for HPC workloads. In: Proceedings of the 2015 International Symposium on Memory Systems. Washington: ACM, 2015. 276–282
58 Mao H, Zhang X, Sun G, et al. Protect non-volatile memory from wear-out attack based on timing difference of row buffer hit/miss. In: Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE). Lausanne: IEEE, 2017. 1623–1626
59 Qureshi M K, Karidis J, Franceschini M, et al. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In: Proceedings of 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). New York: IEEE, 2009. 14–23
60 Cai Y, Lin Y, Xia L, et al. Long live time: improving lifetime for training-in-memory engines by structured gradient sparsification. In: Proceedings of 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC). San Francisco: IEEE, 2018. 1–6
61 Xia L, Huangfu W Q, Tang T, et al. Stuck-at fault tolerance in RRAM computing systems. IEEE J Emerg Sel Top Circ Syst, 2018, 8: 102–115
62 Xia L, Liu M, Ning X, et al. Fault-tolerant training with on-line fault detection for RRAM-based neural computing systems. In: Proceedings of the 54th Annual Design Automation Conference 2017. Austin: ACM, 2017. 1–6
63 Liu C, Hu M, Strachan J P, et al. Rescuing memristor-based neuromorphic design with high defects. In: Proceedings of 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC). Austin: IEEE, 2017. 1–6
64 Ni L, Wang Y, Yu H, et al. An energy-efficient matrix multiplication accelerator by distributed in-memory computing on binary RRAM crossbar. In: Proceedings of 2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC). Macao: IEEE, 2016. 280–285
65 Wang Y, Tang T, Xia L, et al. Energy efficient RRAM spiking neural network for real time classification. In: Proceedings of the 25th edition on Great Lakes Symposium on VLSI. Pittsburgh: IEEE, 2015. 189–194
66 Narayanan S, Shafiee A, Balasubramonian R. INXS: bridging the throughput and energy gap for spiking neural networks. In: Proceedings of 2017 International Joint Conference on Neural Networks (IJCNN). Anchorage: IEEE, 2017. 2451–2459
67 Ankit A, Sengupta A, Panda P, et al. Resparc: a reconfigurable and energy-efficient architecture with memristive

crossbars for deep spiking neural networks. In: Proceedings of the 54th Annual Design Automation Conference 2017. Austin: ACM, 2017. 1–6
68 Xia L, Tang T, Huangfu W, et al. Switched by input: power efficient structure for RRAM-based convolutional neural network. In: Proceedings of the 53rd Annual Design Automation Conference. Austin: ACM, 2016. 1–6
69 Mao H, Shu J. 3D memristor array based neural network processing in memory architecture. J Comput Res Dev, 2019, 56: 1149–1160
70 Ji Y, Zhang Y, Xie X, et al. FPSA: a full system stack solution for reconfigurable reram-based nn accelerator architecture. In: Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems, Providence, 2019. 733–747
71 Witten I H, Frank E. Data mining: practical machine learning tools and techniques with Java implementations. SIGMOD Rec, 2002, 31: 76–77
72 Sharif Razavian A, Azizpour H, Sullivan J, et al. CNN features off-the-shelf: an astounding baseline for recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops. Columbus: IEEE, 2014. 806–813
73 Manning C D, Surdeanu M, Bauer J, et al. The Stanford CoreNLP natural language processing toolkit. In: Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations, Baltimore, 2014. 55–66
74 Schmidhuber J. Deep learning in neural networks: an overview. Neural Netw, 2015, 61: 85–117
75 Goodfellow I, Pouget-Abadie J, Mirza M, et al. Generative adversarial nets. In: Proceedings of Advances in Neural Information Processing Systems, 2014. 2672–2680
76 Kaelbling L P, Littman M L, Moore A W. Reinforcement learning: a survey. J Artif Intell Research, 1996, 4: 237–285
77 Low Y, Gonzalez J E, et al. Graphlab: a new framework for parallel machine learning. 2014. ArXiv: 1408.2041
78 Low Y, Gonzalez J, Kyrola A, et al. Distributed graphlab: a framework for machine learning in the cloud. 2012. ArXiv: 1204.6078
79 LeBeane M, Song S, Panda R, et al. Data partitioning strategies for graph workloads on heterogeneous clusters. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. Austin: IEEE, 2015. 1–12
80 Gonzalez J E, Low Y, Gu H, et al. Powergraph: distributed graph-parallel computation on natural graphs. In: Presented as Part of the 10th USENIX Symposium on Operating Systems Design and Implementation, Hollywood, 2012. 17–30
81 Chen R, Shi J, Chen Y, et al. Powerlyra: differentiated graph computation and partitioning on skewed graphs. ACM Trans Parall Comput, 2019, 5: 1–39
82 Vigna G, Kemmerer R A. NetSTAT: a network-based intrusion detection approach. In: Proceedings of the 14th Annual Computer Security Applications Conference (Cat. No. 98EX217). Scottsdale: IEEE, 1998. 25–34
83 Agichtein E, Castillo C, Donato D, et al. Finding high-quality content in social media. In: Proceedings of the 2008 International Conference on Web Search and Data Mining. Palo Alto: ACM, 2008. 183–194
84 Page L, Brin S, Motwani R, et al. The Pagerank Citation Ranking: Bringing Order to the Web. Stanford InfoLab, 1999. http://ilpubs.stanford.edu:8090/422/
85 Biemann C. Chinese whispers: an efficient graph clustering algorithm and its application to natural language processing problems. In: Proceedings of the 1st Workshop on Graph Based Methods for Natural Language Processing, 2006. 73–80
86 Chesler E J, Lu L, Shou S, et al. Complex trait analysis of gene expression uncovers polygenic and pleiotropic networks that modulate nervous system function. Nat Genet, 2005, 37: 233–242
87 Linden G, Smith B, York J. Amazon.com recommendations: item-to-item collaborative filtering. IEEE Internet Comput, 2003, 7: 76–80
88 Shendure J, Ji H. Next-generation DNA sequencing. Nat Biotechnol, 2008, 26: 1135–1145
89 Shendure J, Balasubramanian S, Church G M, et al. DNA sequencing at 40: past, present and future. Nature, 2017, 550: 345–353
90 Altschul S. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Nucleic Acids Res, 1997, 25: 3389–3402
91 Jha M, Malhotra R, Acharya R. A generalized lattice based probabilistic approach for metagenomic clustering. IEEE/ACM Trans Comput Biol Bioinf, 2017, 14: 749–761

92 Altschul S F, Gish W, Miller W, et al. Basic local alignment search tool. J Mol Biol, 1990, 215: 403–410
93 Ning Z. SSAHA: a fast search method for large DNA databases. Genome Res, 2001, 11: 1725–1729
94 Lancaster J, Buhler J, Chamberlain R D. Acceleration of ungapped extension in Mercury BLAST. Microprocessors Microsyst, 2009, 33: 281–289
95 Ling C, Benkrid K. Design and implementation of a CUDA-compatible GPU-based core for gapped BLAST algorithm. Procedia Comput Sci, 2010, 1: 495–504

Development of processing-in-memory

Haiyu MAO, Jiwu SHU*, Fei LI & Zhe LIU

Department of Computer Science and Technology, Tsinghua University, Beijing 100084*, China*
* Corresponding author. E-mail: shujw@tsinghua.edu.cn

Abstract With the explosive increase of processed data, data transmission through the bus between CPU and the main memory has become a bottleneck in the traditional von Neumann architecture. On top of this, popular data-intensive workloads, such as neural networks and graph computing applications, have poor data locality, which results in a substantial increase of the cache miss rate. Processing such popular data-intensive workloads hinders the entire system since the data transmission causes long latency and high energy consumption. Processing-in-memory greatly reduces this data transmission by equipping the main memory with computation ability, alleviating the problems of poor performance and high energy consumption caused by a large amount of data and a poor data locality. Processing-in-memory consists of two different approaches. One method involves integrating computation resources into the main memory with high-bandwidth interconnects (i.e., near data computing). The other method consists of employing memory arrays to compute directly (i.e., computing-in-memory). These two approaches have their own advantages and disadvantages, as well as suitable scenarios. In this survey, the birth and development of processing-in-memory is firstly introduced and discussed. Its techniques, ranging from hardware to microarchitecture, are then presented. Furthermore, the challenges faced by processing-in-memory are analyzed. Finally, the opportunities that processing-in-memory offers for popular applications are discussed.

Keywords processing-in-memory, near-data computing, computing-in-memory, neural network, graph computing

Haiyu MAO was born in 1993. She received her B.S. degree in Software Engineering from Northeastern University in 2015. She is currently a Ph.D. candidate in the Department of Computer Science and Technology, Tsinghua University. Her research interests include nonvolatile memory, processing-in-memory, memory security, and machine learning accelerators.

Jiwu SHU was born in 1968. He received his Ph.D. degree in Computer Science from Nanjing University, Nanjing, in 1998. Currently, he is a professor in the Department of Computer Science and Technology at Tsinghua University. His research interests include storage security and reliability, nonvolatile memory-based storage systems, and parallel and distributed computing.

Fei LI was born in 1993. He received his B.S. degree in Computer Science and Technology from Tsinghua University in 2015. Currently, he is a master's student in the Department of Computer Science and Technology at Tsinghua University. His research interests include nonvolatile memory, flash-based storage systems, system software, and security.

Zhe LIU was born in 1991. He received his B.S. degree in Information Security from Shanghai Jiao Tong University in 2012. He is currently a postgraduate student in the Department of Computer Science and Technology, Tsinghua University. His research interests include nonvolatile memory and open-channel SSD.