DOI:10.7544/issn1000-1239.20210580
59(3):533-552,2022

基于自选尾数压缩的高能效浮点忆阻存内处理系统

丁文隆[1] 汪承宁[1,2] 童 薇[1,2]
[1](华中科技大学计算机科学与技术学院 武汉 430074)
[2](武汉光电国家研究中心(华中科技大学) 武汉 430074)
(wlding@hust.edu.cn)

Energy-Efficient Floating-Point Memristive In-Memory Processing System Based on Self-Selective Mantissa Compaction

Ding Wenlong[1], Wang Chengning[1,2], and Tong Wei[1,2]
[1](*School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan* 430074)
[2](*Wuhan National Laboratory for Optoelectronics (Huazhong University of Science and Technology), Wuhan* 430074)

Abstract Matrix-vector multiplication (MVM) is a key computing kernel for solving high-performance scientific systems. Recent work by Feinberg et al has proposed a method of deploying high-precision operands on memristive crossbars, showing its great potential on accelerating scientific MVM. Since different types of scientific computing applications have different precision requirements, providing appropriate computation methods for specific applications is an effective way to further reduce energy consumption. This paper proposes a system with mantissa compaction and alignment optimization strategies. Under the premise of implementing the basic function of high-precision floating-point memristive MVM, the proposed system is also possible to properly select the compaction bits of the floating-point mantissa according to application precision requirements. By neglecting the activation of the low-bit crossbars with less mantissa significance and the redundant alignment crossbars when performing computation, the energy consumption of computational crossbars and peripheral circuits are significantly reduced. The evaluation result shows that when the crossbar-based in-memory solutions of sparse linear systems have average solving residual of $0\sim10^{-3}$ order of magnitude compared with the software baseline, the average energy consumption of computational crossbars and peripheral analog-to-digital converters are reduced by 5%~65% and 30%~55% compared with the existing work without optimization, respectively.

Key words memristive crossbars; analog matrix-vector multiplication; energy-efficient scientific computing; in-memory parallel processing system; sparse linear algebra system

摘 要 矩阵向量乘法(matrix-vector multiplication, MVM)运算是高性能科学线性系统求解的重要计算内核.Feinberg等人最近的工作提出了将高精度浮点数部署在忆阻阵列上的方法,显示出其在加速科学MVM运算方面的巨大潜力.由于科学计算不同类型的应用对于求解精度的要求各不相同,为具体应用提供合适的计算方式是进一步降低系统能耗的有效途径.展示了一种拥有尾数压缩与对齐位优化

收稿日期:2021-06-08;**修回日期**:2021-09-29
基金项目:国家自然科学基金项目(61832007,61821003);中央高校基本科研业务费专项资金项目(2019kfyXMBZ037);之江实验室开放课题(2020AA3AB07)
This work was supported by the National Natural Science Foundation of China (61832007, 61821003), the Fundamental Research Funds for the Central Universities (2019kfyXMBZ037), and the Zhejiang Lab Open Fund (2020AA3AB07).
通信作者:童薇(tongwei@hust.edu.cn)

策略的系统,在实现高精度浮点数忆阻 MVM 运算这一基本功能的前提下,能够根据具体应用的求解精度要求选择合适的浮点数尾数压缩位数.通过忽略浮点数尾数权重较小的部分低位与冗余的对齐位的阵列激活,减小运算时阵列及外围电路的能耗.评估结果表明:当忆阻器求解相对于软件基线平均分别有 0～10^{-3}数量级的求解残差时,平均运算阵列能耗与模数转换器能耗相对于已有的优化前的系统分别减少了 5%～65%与 30%～55%.

关键词 忆阻器阵列;模拟矩阵向量乘法;高能效科学计算;存内并行处理系统;稀疏线性代数系统

中图法分类号 TP391

在科学与工程领域中,许多复杂的模型都会用线性系统 $\boldsymbol{Ax}=\boldsymbol{b}$ 的形式表达[1],其中,$\boldsymbol{A}$ 通常是一个庞大的高精度浮点稀疏矩阵[2].在这样的系统上进行求解会消耗大量时间与计算资源[3].目前主流的求解大规模稀疏线性系统的方法是克利洛夫(Krylov)子空间方法[1].该方法的求解过程涉及大量的矩阵向量乘法(matrix-vector multiplication, MVM)计算,所以改进 MVM 的计算模式是节省运算能耗与运算资源的关键途径.

近年来,关于用忆阻阵列进行原位 MVM 运算的有关加速器被不断地提出.最先提出的是一类用于执行机器学习和图形处理任务的忆阻加速器,它们通过向忆阻阵列施加电压,在阵列上原位地执行 MVM 运算[4-10].然而,这些忆阻加速器只提供 8～16 b 的计算,显然不支持一些以高精度浮点数为主的计算应用.于是,最新的研究提出了将 IEEE-754 双精度浮点数部署在阵列上的方式[11].该研究将浮点数的 53 b 尾数(包括前导 1)分位片地部署到不同的阵列上,最后通过乘加缩减树整合不同位片的结果,实现高精度浮点数运算.

然而,目前仍然没有任何工作提出在一个系统内,能够为不同精度的应用提供计算能耗优化的方法.本工作致力于提出一个基于忆阻阵列的模拟 MVM 运算系统,既能够为精度较高的应用提供无损的浮点数计算模式,又能够让较低精度的应用执行低能耗开销计算.具体地,本文工作的贡献主要有 4 个方面:

1) 设计了一种自选尾数压缩机制,对于某些低精度的求解应用,可以在阵列运算中选择性地忽略激活若干权值较小的低位阵列,从而保证在满足具体任务的求解精度的前提下,减少运算阵列以及外围电路的能耗.同时提出了一种动态的对齐位优化机制,摒弃原有的静态对齐位的设定模式,根据矩阵实际指数范围的要求来设置参与运算的对齐位阵列,减少冗余对齐位带来的能耗.

2) 针对于尾数压缩和对齐位优化策略,提出了一种叶子结点数可变的流水乘加缩减树的结构.解决了激活运算的阵列数不确定所带来的偏移策略失效以及无法进行流水的问题.这种新型的乘加缩减树结构能够在激活任意数量的尾数位和对齐位阵列的情况下,都正常地进行流水求和运算.

3) 基于现有工作中的分块映射与片位部署的思想以及上述所提到的优化改进策略,设计完成整个模拟忆阻 MVM 原位运算系统.

4) 将带优化策略的 MVM 原位模拟运算系统集成到高性能线性代数的算法框架中进行求解计算.评估在不同的求解精度下,系统中关键能耗指标的减少程度.

1 背景知识

由于忆阻器阵列存内运算具有高能效、高并行性等优势,受到国内外学者广泛地关注.到目前为止,用忆阻器阵列进行 MVM 运算的方法已经被广泛地研究,包括从用于机器学习模型的忆阻加速器的研究到应用于科学计算的忆阻加速器的研究.

1.1 用于机器学习和图形处理的忆阻加速器

近年来,随着机器学习工作的流行性日益增加,学术界[12]和工业界[13]都提出了许多有关于机器学习的专用加速器的建议.这些加速器主要为机器学习模型提供 MVM 运算,通过加速这一过程的运算,极大程度提升机器学习模型的训练速度.

1) 用忆阻阵列进行点积运算.为了加速机器学习任务中的 MVM 运算,最新的研究提出了一系列基于忆阻器原位计算的新型加速器[4-10].这一类忆阻加速器的工作原理是,将矩阵映射到忆阻阵列的存储单元中,使得每个存储单元的电导值与对应的矩阵系数成比例.矩阵值映射完成后,通过给每一行施加电压来模拟向量元素和矩阵元素的乘积.其中,

每一行施加的电压和向量元素成比例.通过采集每一列的电流,模数转换器(analog-to-digital converter, ADC)能够接收到矩阵每一行与向量的点积和.如图 1 所示,指定矩阵第 j 行第 i 列的电阻值为 R_{ij}(矩阵的一行映射到阵列的一列中),指定向量第 i 个元素为 V_i,则该存储单元贡献的电流为 V_i/R_{ij},求和得到该列的总电流为 $\sum_i V_i/R_{ij}$,这就是矩阵第 j 行与向量相乘得到的结果.

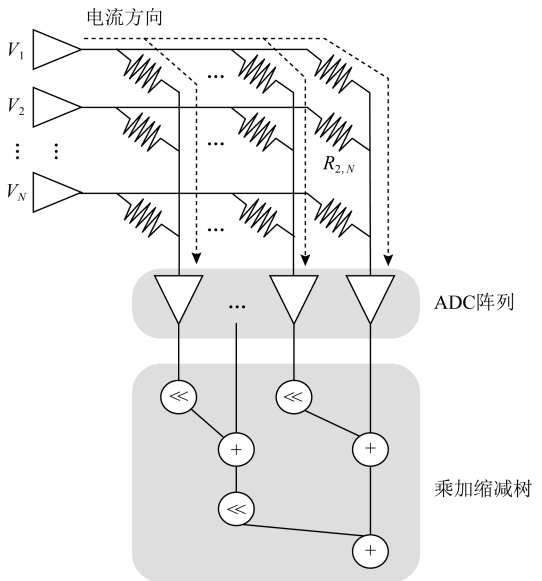


Fig. 1 Illustration of memristive crossbars computation

图 1 忆阻阵列计算示意图

2) 位切片(bit slicing)技术.由于模拟器件的精度受到非理想因素的限制,忆阻存储单元无法精确地表达映射值[14].针对这个问题,现有的工作提出了位切片技术,这种技术通过将矩阵元素按照位片映射到多个阵列上来减少对器件精度的依赖[4-7].一个位切片的例子,这个矩阵将被映射到 3 个二进制存储阵列中:

$$\begin{pmatrix}1 & 3\\ 4 & 6\end{pmatrix}=2^2\begin{pmatrix}0 & 0\\ 1 & 1\end{pmatrix}+2^1\begin{pmatrix}0 & 1\\ 0 & 1\end{pmatrix}+2^0\begin{pmatrix}1 & 1\\ 0 & 0\end{pmatrix}. \quad (1)$$

注意,在实际计算中,对于到来的电压 V_i 采取这种位切片技术.

3) 乘加缩减树.如图 1 所示,为了将各位片的点积结果进行求和运算,需要乘加缩减树进行整合[4-5],得到 MVM 运算的最终结果.乘加缩减树中有 2 类操作:①加法操作,即不同位片得到的结果求和;②左移操作,父结点整合左右子树时,高位片子树需要进行左移操作才能和低位子树对齐相加.一般地,乘加缩减树都是叶子结点数为 2 的整数次方的满二叉树,在这种树结构下,整合移位时将高位子树左移 2^{i-1} b 即可(i 为父结点到叶子结点的距离).

1.2 用于科学计算的忆阻加速器

1.1 节提到的忆阻 MVM 加速器是针对机器学习模型来设计的,只提供 8~16 b 的计算[4-9].虽然这个运算精度大概率不会影响机器学习任务的推理准确度,但是对于高精度的科学计算求解来说,这种运算精度是远远不够的[15].

1) 浮点数的忆阻阵列部署.为了解决科学计算中的精度问题,Feinberg 等人[11]首次提出了在忆阻加速器上部署科学计算.在 IEEE-754 标准中,双精度浮点数由 53 b 的二进制尾数(包括前导 1)、11 b 的二进制指数和一个符号位表示[16].这项工作首先提出了如何将 53 b 的尾数部署到忆阻阵列上进行 MVM 运算,它运用了多个存储阵列,每一个阵列代表矩阵中浮点数的一个位片,形成一个存储簇;然后再将不同的阵列通过乘加缩减树连接起来,如图 2 所示:

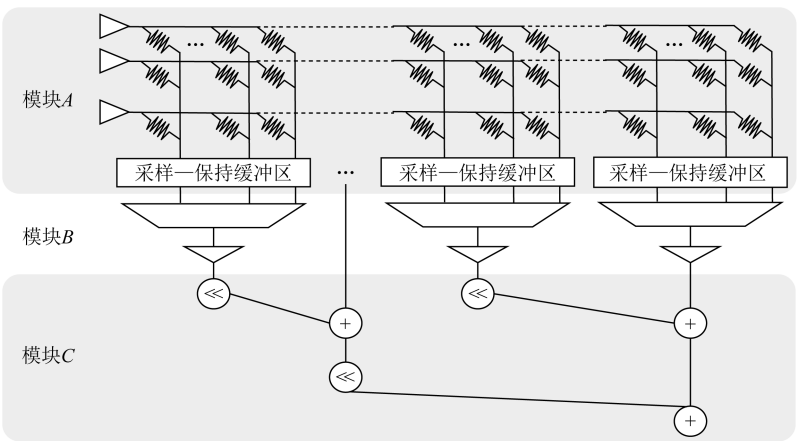


Fig. 2 Array organization in a memristive cluster

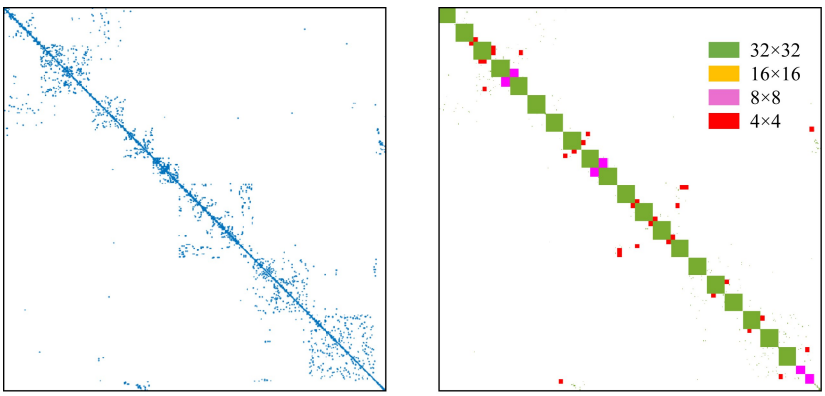
图 2 忆阻存储簇中的阵列组织

存储簇用于计算向量的一个位片与矩阵元素的乘积.在完成了位片映射后,进行 3 阶段的运算:1)电压加载在到模块 A 中的阵列上,列电流存储在采样—保持缓冲区中,该电流代表当前向量位片和对应的矩阵位片每一行的点积之和;2)在模块 B 中,选择器选择采样—保持缓冲区中代表相同行不同位片的电信号,经过 ADC 转化为数字信号后,送入乘加缩减树相加;3)模块 C 中乘加缩减树对模块 B 中输入的不同位片结果进行整合运算.

经过这 3 个阶段,得到了矩阵的 1 行与向量位片的点积结果.为获得最终结果,需要循环执行第 2 阶段与第 3 阶段,获得所有矩阵行与向量位片的乘积.

2) 异构阵列集合.Feinberg 等人[11]的工作还提出了一种异构矩阵分块思想.采用不同大小的块去捕捉稀疏矩阵中非 0 元素的密集区域,并为每个大小的块设定一个非 0 元素阈值,若达到阈值就将该分块映射到存储簇上[11].对比使用单一阵列大小的工作[4-5,9],这种异构设计极大程度上增加了阵列非 0 元素密度,减少了阵列映射开销,保证了阵列的并行性和能源效率.

图 3 展示了矩阵元素分块的一种示例.其中,数据来源是 SuiteSparse 矩阵数据集合[2]中的 685_bus 矩阵.其中,图 3(a)展示了该矩阵非 0 元素的分布情况,图 3(b)展示了分块之后的结果.分别采用了 32×32,16×16,8×8,4×4 这 4 种不同大小的矩阵块对原矩阵进行分块,阈值分别为 128,32,8,2.



(a) 矩阵非0元素分布情况 (b) 矩阵分块结果

Fig. 3 Example of matrix blocking (685_bus matrix)

图 3 矩阵元素分块示例(685_bus 矩阵)

3) 存储簇中对齐位的部署.完成了矩阵分块之后,需要将每一块的浮点数都映射到存储簇中.然而同一矩阵块中元素的二进制指数不尽相同.为了使得不同元素的尾数按照指数对齐,需要以块中指数最大的元素作为基准,对较小指数的元素实行向右偏移部署的策略,然后在空余位补充 0.

图 4 展示了存储簇中浮点数格式部署的一个例子.需要部署 10.5,6.5,0.3 这 3 个浮点数,其对应的二进制指数分别为 3,2,−2.部署时以 10.5 对应的指数作为基准,6.5 和 0.3 的部署分别向右偏移 1 b 和5 b,并将空余位置填充 0.

4) 定点硬件上执行浮点运算.在系统实现过程中,图 4 所述的利用填充 0 实现的对齐步骤是在浮点数映射过程中实现的,该过程处于预处理阶段.在

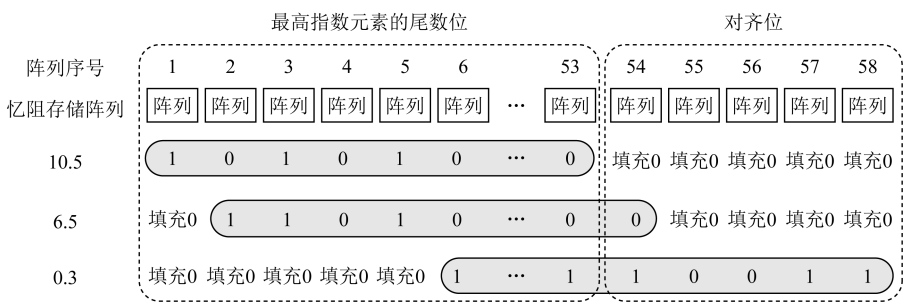


Fig. 4 Example of alignment bits deployment in cluster

图 4 存储簇中对齐位部署示例

阵列运算过程中,不会再出现动态的移位对齐等步骤,整个运算是在定点硬件上进行的.所以,同一个存储簇中的阵列运算,仅涉及到定点运算.通过二进制分片策略与预处理填充对齐位的策略,实现了在定点硬件上执行浮点运算.而在整合不同存储簇的运算结果时,会涉及到定点数转浮点数,浮点数相加等纯浮点运算.故所构造的系统是一种以定点硬件运算为主,纯浮点运算为辅的系统.

2 观察与动机

之前的工作所使用的都是固定位数的浮点数尾数部署方式,这种部署模式可能会产生额外能耗开销.本节将讨论这个问题以及对应的优化策略.

2.1 为不同的应用优化计算能耗开销

在科学与工程领域中,虽然多数的线性模型的矩阵元素都是浮点数,但是不同应用所需的求解精度各不相同.例如,求解网页排序算法(PageRank)、求解线性回归等大规模数理统计模型时,所需要的求解精度都不高[17-18].已有的研究显示,在执行机器学习、图形处理任务时,只使用 8～16 b 精度就能达到令人满意的准确率[4-9].但是,对于一些由偏微分方程离散化得到的大型稀疏性方程组而言,需要的求解精度比较高[1],其代表领域有航天航空领域以及量子力学领域等.然而,执行高精度的求解运算必然会消耗更多的能量,这对于求解精度需求较低的线性系统来说是不划算的.

现有工作中,高精度浮点数采用的是固定位片数的部署形式[11],在每一个存储簇中,有 53 b 的二进制尾数阵列、64 b 的对齐位阵列以及 9 b 的校验位阵列.在实际运算时,运算阵列与其外围电路的能耗都与激活计算的阵列个数呈正相关,也就是说,随着参与运算的阵列个数增多,计算能耗也会剧烈增加.然而,上述的浮点数部署模式是针对于通用 IEEE-754 双精度浮点数格式设计的,对于所有的双精度浮点应用都能够保证计算精度.对于低精度的求解应用而言,并不需要激活所有的 53 b 尾数、64 b 对齐位进行计算就能够达到允许精度范围内的解,所以按照原有固定方式计算必然会产生不必要的能耗.

2.2 尾数压缩与对齐位优化策略

为了解决 2.1 节所述的为不同应用优化计算能耗开销的问题,在本节中将论述一种尾数压缩和对齐位优化策略,动态地决定实际运算时激活的尾数以及对齐位阵列数量.

1) 尾数压缩策略.在实际计算时,可以根据具体的应用精度,自行选择参与运算的尾数位数,从而大幅度减少运算阵列带来的能量消耗.例如,如果选择 25 b 尾数参与运算,在实际运算时只会激活高 25 b 尾数阵列进行运算,而不会对剩下的低位阵列(低 28 b 的阵列)施加电压,也就不会产生这些阵列对应的运算阵列和外围电路能耗.

2) 对齐位优化策略.在之前的有关于位片型忆阻阵列科学计算的系统设计中[11],对齐位是固定的 64 b.然而,并不是所有的应用都需要多如 64 b 的指数范围.为了节省计算开销,采取一种动态的对齐位优化策略,只部署指数范围内的对齐位.例如,一个矩阵块中非 0 元素的二进制指数范围是 1～20,即只部署 19 b 的对齐位阵列即可,这样便会一定程度上减少对齐位冗余带来的阵列计算消耗.

2.3 叶子结点数可变的流水乘加缩减树

由于本文所提出的系统应用了尾数压缩和对齐位优化策略,导致不同应用的计算中,尾数和对齐位的位数之和是不固定的.在这种情况下,进行计算的乘加缩减树中叶子结点的个数是可变的.所以,为了解决尾数压缩和对齐位优化策略带来的叶子结点数可变的问题,需要设计一种新型乘加缩减树结构,来完成存储簇中不同位片结果的整合.

1)乘加缩减树中的层级流水概念.假设目前存储簇拥有 128 个忆阻阵列,阵列大小为 $n\times n$,即对应的乘加缩减树为 7 层.每加载一次向量,该树就需要计算 n 次,以得到矩阵块每一行和向量相乘的结果.然而,由于树结点之间的计算可以并行操作,且同层之间没有数据依赖,可以把乘加缩减树的每一层看作一个流水部件,构建一个 7 级的流水线.假设每个结点的计算时间为单位时间,这样计算 n 个结果的时间为 $6+n$.

2) 叶子结点数可变的流水.在以往固定位片的阵列部署模式下,叶子结点数是确定的,且数量往往是 2 的整数次方(若不为 2 的整数次方,则往往将叶子结点数补满到 2 的整数次方).在此情况下,所构造的树为满二叉树.现在由于叶子结点数可变,会导致整棵树不为原有的满二叉树形式,导致 2 个问题:①由于不是满二叉树,各叶子结点到根结点的距离不尽相同.若采用流水策略,各叶子结点数值无法同时到达根结点,产生计算错误.②若所计算的乘加缩减树不为满二叉树,则前面背景介绍中论述的高位子树偏移 2^{i-1}(i 为父结点到叶子结点的距离)与低位子树对齐相加的策略就会失效,使用该方法来计

算高位子树的偏移量会带来错误.为了解决这 2 个问题,需要提出叶子结点数可变的流水乘加缩减树(在 3.3.3 节详细论述).

3 基于异构阵列的自选尾数压缩系统设计

本节将展示一个基于异构阵列集合,拥有尾数压缩和对齐位优化策略,面向科学计算的 MVM 系统.

3.1 系统总体结构设计

如图 5 所示,系统主要由预处理模块以及阵列运算和整合模块构成.

1) 预处理模块.预处理部分主要负责处理原始矩阵,并提供阵列映射方案.具体地,该部分将对原始矩阵依次进行分块、分片、执行尾数压缩与对齐位优化策略,最终获得能直接映射在忆阻阵列上的 0-1 矩阵,并为后续计算提供阵列激活策略.

2) 阵列运算和整合模块.阵列运算和整合部分主要负责对于到来的向量 $\boldsymbol{x}$,通过阵列计算以及辅助处理子模块的整合,获得最终 $\boldsymbol{Ax}$ 的结果.具体地,需要先将向量位片按时序加载在存储簇上,然后在阵列上执行矩阵向量乘法运算,再在辅助处理子模块中对不同存储簇的结果以及未分块元素进行整合,最后得到矩阵向量的乘法结果 $\boldsymbol{b}$.

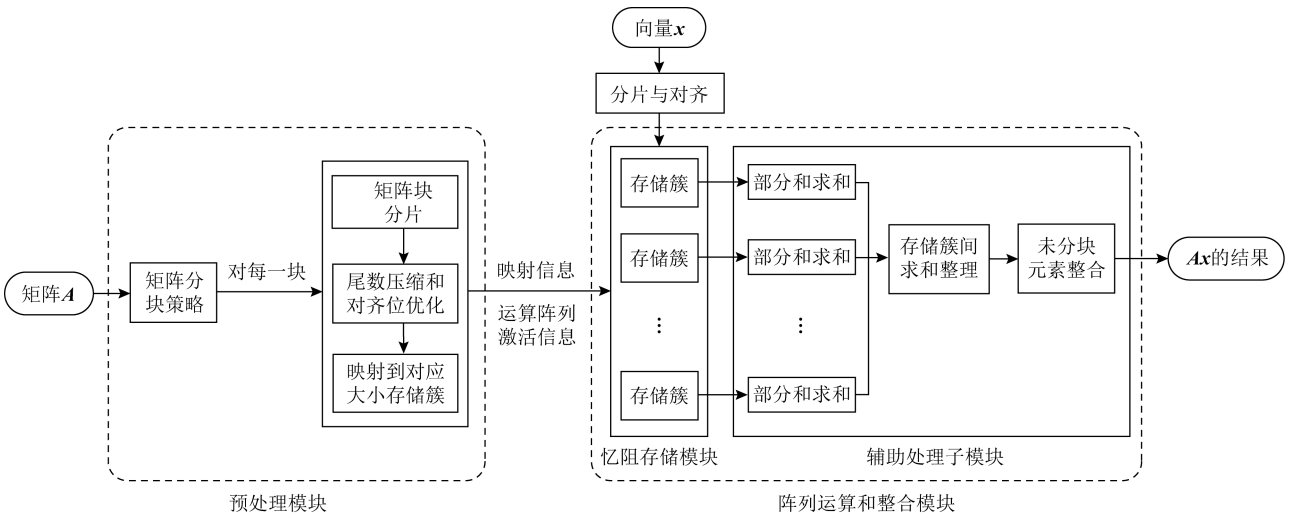


Fig. 5 Overall structure design of the system

图 5 系统总体结构设计

3.2 预处理模块的详细设计

图 6 展示了预处理模块主要任务的设计流程:

1) 矩阵分块与分片.由用户输入最大块的边长 L 和最大块中允许的最少非 0 元素个数阈值 p.分别运用边长为 L,$L/2$,$L/4$,$L/8$ 的块来采集矩阵中非 0 元素密集的区域,其对应的非 0 元素阈值分别

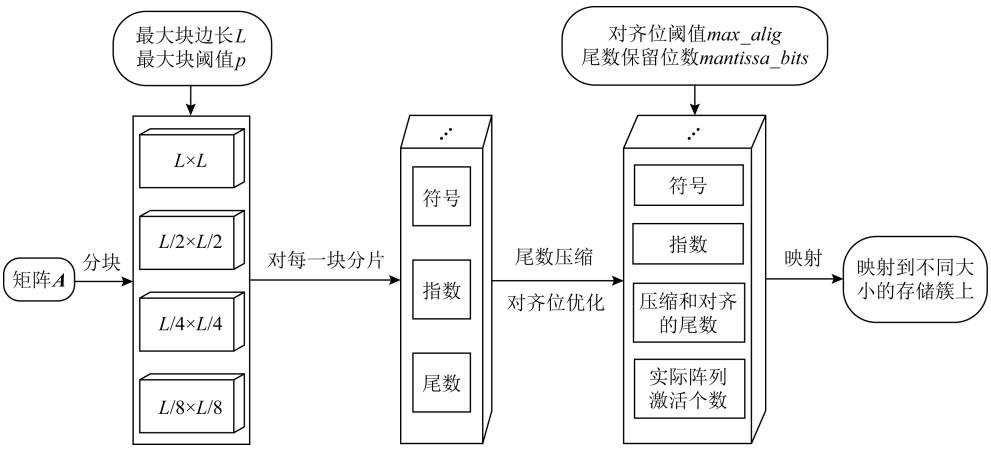


Fig. 6 Design of the pre-processing step

图 6 预处理过程设计

为 p,$p/4$,$p/16$,$p/64$.在分块过程中,先用边长为 L 的块去铺排整个矩阵,根据阈值 p 去捕获达到密度的块,若密度达不到,则用 4 个边长为 $L/2$ 的块去铺排这个边长为 L 的矩阵块,捕获非 0 元素数量超过阈值 $p/4$ 的区域;以此类推,用 $L/4$,$L/8$ 的块去铺排.若有非 0 元素没有被任何块捕获,则其会进入未分块元素集合,等待辅助处理子模块统一整合处理.

对于每个浮点数而言,需要获得其 IEEE-754 双精度浮点数表示形式(或者其科学记数法表示形式),进而提取其 53 b 尾数(包括前导 1)、其指数以及其符号位.对于同一个矩阵块,需要对其中所有的非 0 元素执行此操作,获得一个具有($sign$,$significand$,$expval$)三元组元素的矩阵,其中 $sign$ 表示符号位;$significand$ 表示有效数,即带前导 1 的共 53 b 尾数;$expval$ 表示对应二进制指数.

2) 尾数压缩与对齐位优化.为实现自选尾数压缩策略,可以在系统上增加一个尾数选择压缩的接口 $mantissa\_bits$,其意义是用户设定的尾数保留位数,保留高 $mantissa\_bits$.

为实现冗余对齐位的优化,需要在矩阵分块之后,对矩阵块中每个非 0 元素的指数进行简单处理,获得该块中最大的二进制指数 $maxexp$ 和最小的二进制指数 $minexp$,本系统参考矩阵块中指数的实际范围来进行对齐位阵列的激活.同时,本系统同时还会给出一个可供用户设置的对齐位数阈值 max_alig.对齐位数的设置形式:

$$alig\_bits=\min(maxexp\text{-}minexp, max\_alig). \quad (2)$$

在一般情况下,对齐位数 $alig\_bits=maxexp-minexp$,这样做就可以用最少的对齐位数覆盖整个矩阵块的指数范围,相对于固定的对齐位方案而言减少了不必要的计算能耗.给定用户设置的阈值 max_alig 的目的是为了防止在矩阵块中有个别指数极小的元素,导致 $maxexp$ 和 $minexp$ 差距过大,需要用到过多的对齐位,白白耗费阵列资源.当然,如果具体的问题对求解精度要求不高但对于节省能耗的要求较高,用户也可以根据节省能耗和求解精度的具体要求选择较小的对齐位阈值,这一点是灵活的.

如果矩阵块中有元素的指数超过了阈值所限制的范围,则在预处理步骤中这些元素会重新被归类到未分块元素中,而不会在后续进行阵列映射.尾数压缩和对齐位优化的过程都在预处理步骤进行,即在阵列映射前完成.可以通过这 2 个步骤确定每一个存储簇中具体需要激活使用的阵列数量.

图 7 展示了尾数压缩与对齐位优化策略在预处理模块中整体的执行流程:

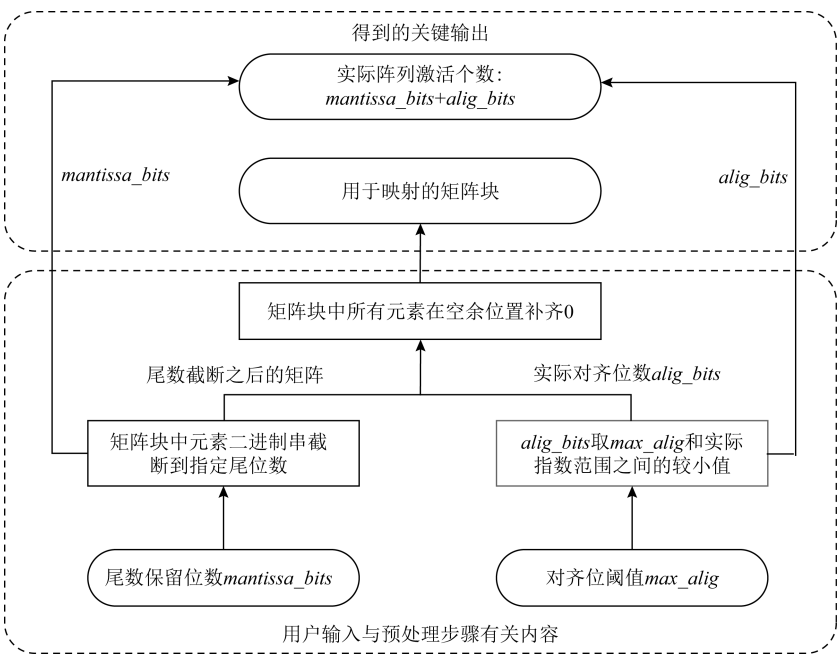


Fig. 7 Design of mantissa compaction and alignment optimization

图 7 尾数压缩与对齐位优化的设计

3) 映射.对于每一个矩阵块,需要将其映射到对应的存储簇上.假设矩阵块大小为 $N\times N$,一共需要映射的阵列个数为 M 个.则对应的存储簇由 M 个 $N\times N$ 的忆阻存储阵列组成,将得到的矩阵块尾

数二进制串按照从高位到低位映射到这 M 个存储阵列上即可.

3.3 阵列运算与整合模块的详细设计

本节主要阐述,在一个已经完成映射的忆阻异构阵列集合中,从加载向量 $\boldsymbol{x}$,到计算出最后的结果 $\boldsymbol{Ax}$,这一完整过程的设计.

3.3.1 阵列运算与整合模块结构设计

图 8 展示了阵列计算和整合模块的设计结构.其中,在每个存储簇的输入缓冲区中,对于到来的向量 $\boldsymbol{x}$ 进行分片和对齐,并以时序展开,依次输入存储簇中进行运算.存储簇通过阵列点积的计算以及乘加缩减树的计算,将向量 $\boldsymbol{x}$ 的当前位片值与当前矩阵块的点积结果存在输出缓冲中.接下来的工作由辅助处理子模块完成,包括整合同一存储簇不同向量位片的结果,整合不同存储簇及未分块元素的结果.全局缓存区用于存储必要中间值以及最后的结果.

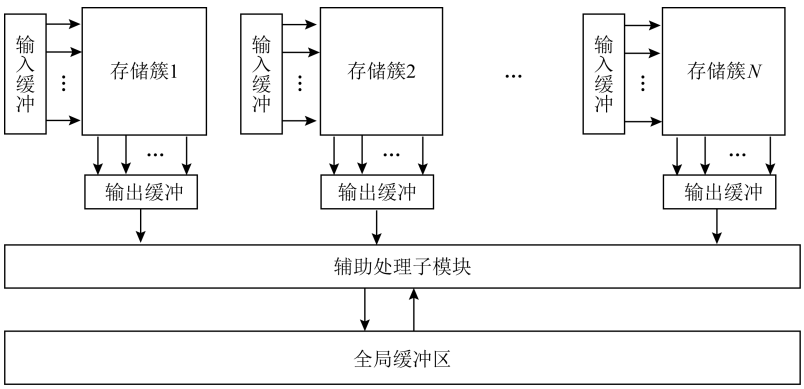


Fig. 8 Crossbar computation and integration module structure

图 8 阵列运算与整合模块的结构

3.3.2 负数的处理

在矩阵 $\boldsymbol{A}$ 和向量 $\boldsymbol{x}$ 中,会出现负浮点数,本节将对这 2 种负数的处理方式进行论述.

1) 矩阵块中的负数处理.为了处理矩阵 $\boldsymbol{A}$ 映射到存储阵列上的负数值,可以将一个矩阵块中的正数值和负数值分开映射(以三元组中的符号位元素作为判断标准),然后在后续整合模块的时候相减即可.这样,同一存储簇中就要维护 2 个阵列集合,分别对应该矩阵块的正非 0 元素和负非 0 元素.在存储簇 c 中,正存储阵列集合和负存储阵列集合分别为 c_{pos} 和 c_{neg}.

2) 向量的负数处理.对于向量 $\boldsymbol{x}$ 来说,可以用负电压来代表 $\boldsymbol{x}$ 中的负数元素.但是这会导致一个问题,即产生的电流值会有负数,进而经过 ADC 转换之后会有负数值进入乘加缩减树.然而本工作中实现的乘加缩减树没有集成关于减法(或补码)的运算,这会导致其无法处理负数值.为处理这个问题,可以给进入乘加缩减树的电流转换值都增加一个数值为 N 的偏移,其中 N 为该存储阵列的大小.这样便能够将电流转换的取值范围从 $-N\sim N$ 转换到 $0\sim 2N$.然后,由于在整合模块中正数阵列的值要和负数阵列的值一一相减,所以增加的偏移值会在运算中刚好抵消,而不用额外操作来消除偏移值.

3.3.3 叶子结点数可变的流水乘加缩减树的设计

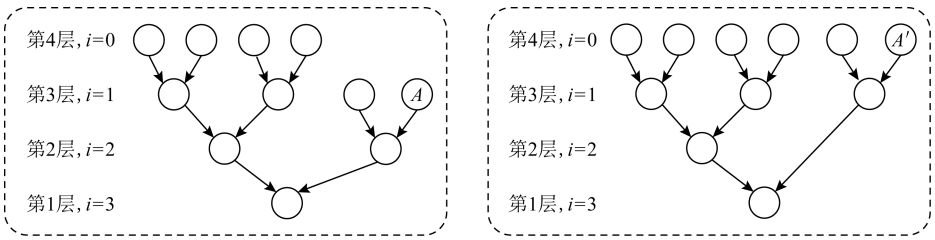
如 2.3 节所述,相对于普通的叶子结点数固定的满二叉树形式,叶子结点数可变的树形式存在 2 个问题:移位策略失效、流水过程计算出错.下面将针对这 2 个问题提出正确的树的结构设计.

1) 解决移位策略失效的问题

在乘加缩减树是满二叉树时,假设当前结点到叶子结点的距离为 i,现在该结点需要整合左右 2 棵子树的元素,即将右子树(规定右子树为高位子树)的值左移 2^{i-1} 位,和左子树相加即可.例如,某个叶子结点数为 8 的满二叉树,这棵树共有 3 层.考虑其最高位元素,它相对于最低位的元素需要左移 7 b.它在从叶子结点到达根结点前需要经历 $i=1$, $i=2$, $i=3$ 这 3 层中对应结点的共 3 次移位,总共移动 $2^{1-1}+2^{2-1}+2^{3-1}=1+2+4=7$ b,刚好符合移位要求.对于编号为奇数的叶子结点也同理,例如第 5 个元素,其需要左移 4 b.它在到达根结点之前需要经历 $i=3$ 层结点的 1 次移位(因为在 $i=1$ 与 $i=2$ 层中其为左子树不需移位),即总共移动 $2^{3-1}=4$ b,符合要求.

现在由于应用了尾数压缩与对齐优化策略,叶子结点的数量不会固定为 2 的整数次方,乘加缩减树不再为满二叉树,会导致原有移位策略失效.若按

照传统方法,从根结点开始构造该 1 棵叶子结点为 6 的尽量平衡的二叉树的话,结果如图 9(a)所示.其中,结点 A 的实际偏移量应为 5,而按照原有偏移策略算得的偏移量为 $2^{2-1}+2^{3-1}=2+4=6$,产生错误.而如图 9(b)模式构造树,符合原有的计算模式,其偏移为 $2^{1-1}+2^{3-1}=1+4=5$,偏移值正确.



(a) 传统的拥有错误移位计算模式的树的构造 (b) 拥有正确移位计算模式的树的构造

Fig. 9 Comparison of trees with wrong and correct shift computation modes

图 9 拥有错误与正确移位计算模式的树的对比

当前的目标是设计拥有正确移位模式的乘加缩减树的构造策略.即需要对任意数量的叶子结点,找到一种通用策略,构造如图 9(b)所示的正确移位模式的树.受到偏移模式以及一些从叶子结点开始构造的树(如霍夫曼树)的启发,可以选择由叶子结点开始构造这样 1 棵具有正确模式的计算树.具体地,从叶子结点层开始,两两相邻的结点为一组,构造下一层父结点;若本层有落单结点(如果结点数为奇数则最后一个结点为落单结点),则该落单结点和新构造的父结点一起组成一个新的集合,继续构造接下来一层的父结点.如此循环地执行该构造策略,当新的集合中的结点数为 1 时,树的构造停止,该结点为根结点.

如此构造乘加缩减树,便可用 i 的值计算出这些结点所在的实际层数,保证每个结点在运算中能获得正确的偏移量,到达根结点的时候获得了正确的总移位值.图 10 左图用这种方法构造了 1 棵有 11 个叶子结点的具有正确移位计算模式的乘加缩减树.可以看到,在 $i=0$ 这一层中,结点 A 落单,它与 $i=1$ 层的 5 个结点一起,构造了 $i=2$ 这一层的结点,然而,在这一过程中,结点 A 所对应的 i 值不变,仍然为 0.同理,对于结点 B,其为 $i=2$ 这一层的落单结点,它与 $i=3$ 这一层的结点一起构造了 $i=4$ 这一层的结点,同时,结点 B 对应的 i 值仍然不变,即 $i=2$.

所以,对于结点 A 的计算过程而言,其到达根结点 C 需要经过结点 B 和结点 C 的移位计算,其中在结点 B 处移位 $2^{2-1}=2$ b,在结点 C 处移位 $2^{4-1}=8$ b,一共移位 $2+8=10$ b,符合要求.

2) 解决无法进行流水操作的问题

想要在改进后的乘加缩减树中执行层级流水策略,会因为各叶子结点到根结点的距离不同而导致计算错误.如图 10 左图所示,叶子结点 D 到根结点 C 的路径长为 4,而叶子结点 A 到根结点 C 的路径长为 2.由于每进行一次流水操作都要对叶子结点加载一批新的数值,所以,当结点 A 经过 2 次流水操作到达结点 C 时,同一时刻输入的结点 D 值还没到达结点 C;当结点 D 值到达结点 C 时,此时从结点 A 到达结点 C 的值已经是 2 个周期之后加载在结点 A 的值.所以,在计算过程中始终面临着数据达到不同步的错误.

① 结点维护先进先出队列.为了解决无法流水的问题,可以让每一个结点都维护一个先进先出队列,用队列的长度来弥补某些叶子结点到根结点的距离比其他叶子结点短的问题.图 10 右图展示了图 10 左图中乘加缩减树对应的队列结构.可以发现,由于结点 A,B,C 这一计算路径长度为 2,这个值要小于满二叉树部分的计算路径(例如从结点 D 经历 3 个中间结点到结点 C,其路径长为 4),所以需要结点 A 和结点 C 的队列长度为 2,其余结点的队列长度为 1.当父结点需要整合左右子树的值时,只需要从左右子树取出其队首的值进行操作即可,在图 10 右图中实线代表该子树需要移位后才能进入父结点进行相加操作,虚线代表可以直接进入父结点进行加法操作.具体地,若当前需要计算的父结点的取值队列为 cur,左右子树的取值队列分别为 $left$ 和 $right$,则计算整合的过程描述为

$$cur.push\ (left.pop+right.pop\ll 2^{i-1}). \quad (3)$$

图 10 右图描述了第 6 次向树的叶子结点中加载数值时,树各结点中队列的存储情况.其中,t_i 代表该结点队列中存储值在数据流中的时间顺序,

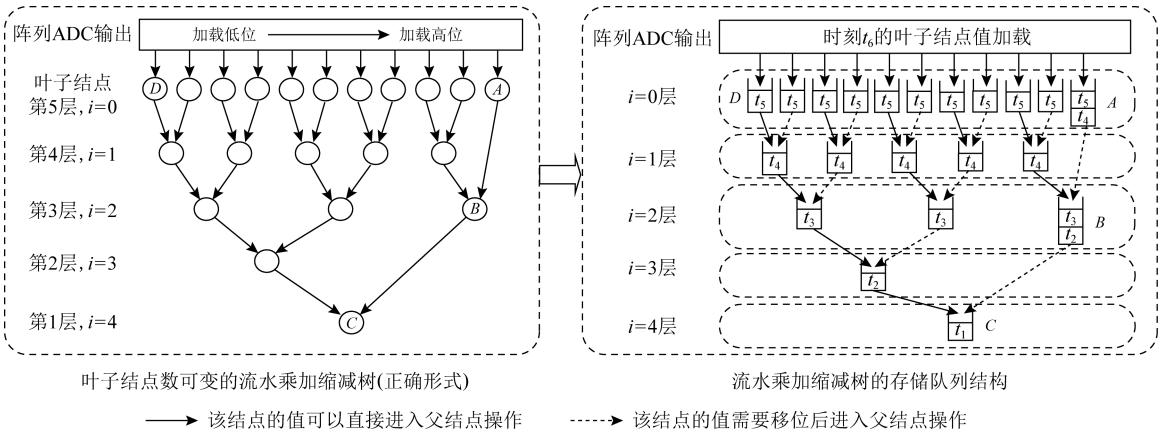


Fig. 10 Example of the correct pipelined multiply-and-add tree

图 10 正确形式的流水乘加缩减树的结构示例

例如 t_4 代表该值是由第 4 次加载在叶子结点中的数据流经过乘加缩减树对应位置计算得到的值.可以看到,在结点 A 和结点 C 中分别需要存储 t_4,t_5 和 t_2,t_3 的值,以保证在 $i=2$ 层计算时,结点 B 要从结点 A 中获得 t_4 时间顺序的值,且保证在 $i=4$ 层计算时,结点 C 要从结点 B 中获得 t_2 时间顺序的值.

② 维护正确的队列长度与存储值.现在设计的关键是要让所有叶子结点的计算路径上都存储相应长度的队列,这样层级流水才能取得正确的值.由此,在流水线没有充满的时候,不能够一次性地激活所有流水层级,而是一级一级地激活.以图 10 为例,第 1 次加载数据时,需要激活层 $i=0$ 计算;第 2 次加载数据时,激活层 $i=1$,即此时 $i=0$ 和 $i=1$ 这 2 层参与计算,直到最后第 5 次加载数据时,激活流水所有层级计算;并且在随后的计算中,都是每一次加载数据,然后流水的所有层级一起计算.这样一来,在激活 $i=1$ 的时候,只有 $i=0$ 和 $i=1$ 这 2 个层级在计算,结点 A 按照顺序储存了 2 个待加值;同理,在激活层 $i=2$ 和层 $i=3$ 后,结点 B 按顺序储存了结果,符合预期.

3.3.4 辅助处理子模块的运算设计

图 11 展示了该过程中辅助处理子模块的处理步骤与全局缓冲区中的内容变化:

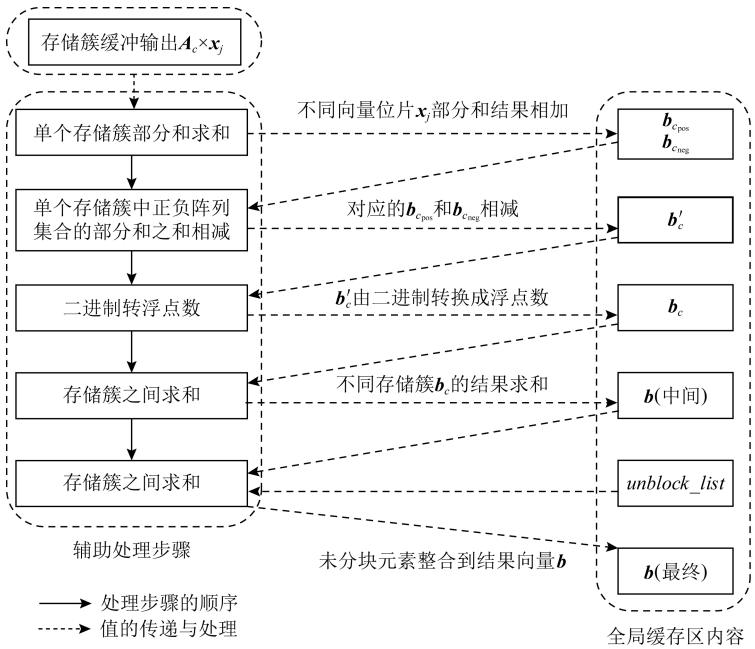


Fig. 11 Auxiliary processing module workflow

图 11 辅助处理子模块工作流程

经过了阵列运算与乘加缩减树整合之后,每个存储簇的缓冲输出中都储存了矩阵块与向量位片的乘积结果,下面称这个结果为关于该向量位片的部分和.

辅助处理子模块将对这些结果进行整合,获得最后的 $\boldsymbol{Ax}$ 结果.具体解释:

1) 设该存储簇 c 存储的矩阵为 $\boldsymbol{A}_c$,其中正数与负数存储阵列集合为 c_{pos} 和 c_{neg},向量第 j 位片对应的部分和为 $\boldsymbol{A}_c \times \boldsymbol{x}_j$.将不同位片的部分和相加,得到当前矩阵块正数元素以及负数元素阵列与向量 $\boldsymbol{x}$ 的乘积结果 $\boldsymbol{b}_{c_{\text{pos}}}$ 与 $\boldsymbol{b}_{c_{\text{neg}}}$.

2) $\boldsymbol{b}_{c_{\text{pos}}}$ 与 $\boldsymbol{b}_{c_{\text{neg}}}$ 的值相减,得到当前矩阵块 $\boldsymbol{A}_c$ 与向量 $\boldsymbol{x}$ 的最终乘积结果 $\boldsymbol{b}'_c$.

3) 将二进制串 $\boldsymbol{b}'_c$ 转换成浮点数 $\boldsymbol{b}_c$.转换完成之后,整合各存储簇 c 间 $\boldsymbol{b}_c$ 的结果,得到结果向量 $\boldsymbol{b}$ 的中间结果.

4) 将未分块元素列表 *unblock_list* 中的元素都整合到向量 $\boldsymbol{b}$ 的中间结果中,得到 $\boldsymbol{b}$ 的最终结果.

4 基于异构阵列的自选尾数压缩系统实现

本工作将用软件模拟实现所设计的忆阻 MVM 系统.具体地,将用 Python 程序模拟实现硬件的存储结构、运算框架的构造以及数据在系统中的计算.本节将对系统中关键的存储结构、系统处理流程的具体实现作出详细的介绍.

4.1 硬件结构的实现

本系统需要模拟的硬件主要包含忆阻运算阵列(存储簇)以及乘加缩减树结点.本系统将用 Python 的类(class)中的成员变量来模拟各个存储结构所包含的存储项,若在运算过程中需要请求使用硬件资源,则只需要用软件编程方法实例化对应的类即可.具体地,本系统存储结构需要实现 3 种类.首先是有关阵列的 2 个类,分别是单个忆阻阵列和相同大小忆阻阵列组成的存储簇,表 1 展示了其模拟存储结构(类中需要储存的成员);其次是乘加缩减树结点硬件所对应的类,表 2 展示了其模拟存储结构.

Table 1 Data Storage Structure of a Storage Cluster and a Single Memristive Crossbar

表 1 存储簇和单个忆阻阵列的数据存储结构

类	存储项	数据类型	存储项解释
忆阻阵列	*size*	整数	阵列大小
	mat	0 或 1 矩阵	阵列映射值
	sign	整数	正数/负数阵列的标志
存储簇	*x*	整数	对应矩阵块左上角元素在原矩阵中的行数值
	y	整数	对应矩阵块左上角元素在原矩阵中的列数值
	minexp	整数	对应矩阵块非 0 元素最小指数
	size	整数	存储簇中单个忆阻阵列的大小
	len	整数	存储簇中需激活的忆阻阵列的个数
	c_{pos}	忆阻阵列表	存储簇 c 中所有映射正数的阵列
	c_{neg}	忆阻阵列表	存储簇 c 中所有映射负数的阵列

Table 2 Storage Structure of Nodes in Multiply-and-Add Trees

表 2 乘加缩减树结点的存储结构

存储项	数据类型	初始值	存储项解释
strq	二进制串队列	空队列	该结点当前存储值
left	乘加缩减树	None	当前结点左子树
right	乘加缩减树	None	当前结点右子树
i	整数	−1	到叶子结点的距离

注:None 表示空树(空指针).

4.2 系统处理流程的实现

在实现了模拟忆阻存储结构的基础上,可以对忆阻阵列进行映射与计算.下面将介绍整个系统处理过程的实现要点.

4.2.1 整体实现流程

本节将从总体角度介绍如何用程序模拟数据在忆阻系统中的计算.图 12 显示了整个系统的实现流程,下面将对照该图论述数据在系统中的计算以及系统的总体实现结构.

首先,从文件中读取原始矩阵 $\boldsymbol{A}$(具体地,用 Python 的 scipy.io 库读取原始".mat"文件,具体数据信息在 5.1.2 节中介绍).然后,进入预处理过程,该过程得到了经过尾数压缩与对齐位优化后的异构二进制矩阵块,以及各矩阵块对应的信息.对每一个

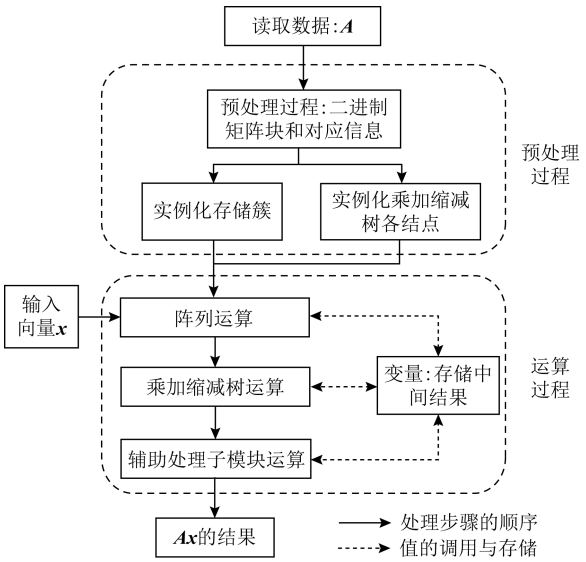


Fig. 12 Overall system implementation process

图 12 系统整体实现流程

矩阵块,都实例化对应的存储簇和乘加缩减树结点的 Python 类,以达到用软件方法模拟实现硬件资源的目的.预处理过程(包括分块分片、尾数压缩和对齐位优化过程)都用 Python 源代码实现,封装成一个 Python 程序模块,预处理具体数据时调用该模块即可.最后进入运算过程,分为 3 个步骤:1)阵列运算,将向量 $\boldsymbol{x}$ 与忆阻阵列中 0-1 矩阵进行点乘,运算结果存储在对应变量中(用变量模拟图 8 所示的缓冲输出).2)进行乘加缩减树运算,按照 3.3.3 节中的描述,编程实现每一个结点的移位和加法操作,同时维护结点正常的队列长度,保证数据流在树结构中的正常路径.3)辅助处理子模块中运算的过程.主要包括部分和的求和、定点数转浮点数、不同存储簇结果与未分块元素的整合,用变量来模拟图 8 中的全局缓存区.用 Python 源程序实现整个运算过程,封装成一个模块,接收已完成数据映射的忆阻存储结构以及输入向量 $\boldsymbol{x}$,返回矩阵向量乘法的结果.

4.2.2 预处理过程与阵列运算的实现

预处理过程主要包括分块、分片、尾数压缩和对齐位优化、矩阵映射 4 个步骤,完成了映射之后,便可以加载向量元素进行阵列运算.

1) 分块信息表 *block_list* 和未分块信息表 *unblock_list* 的构造与维护.在矩阵分块过程中,按照块由大到小循环遍历整个矩阵,这个过程中需要记录分块元素信息,存在 *block_list* 中.表中元素为元组$(x,y,size)$,分别代表该分块的左上角在原矩阵中的行数值、列数值以及块的大小.如果遍历到最小块仍然满足不了分块阈值要求,或者遍历完整个矩阵后有没有处理到的边角元素时,需要记录其中的非 0 元素(即未分块元素),存在 *unblock_list* 中.该表中元素为元组(x,y,val),分别代表该元素在原矩阵中的行数值、列数值以及对应的浮点数值.

2) 矩阵块处理映射过程.当 *block_list* 构造完成之后,便可以进行相应处理将每个矩阵块映射到对应的存储簇上.首先,根据$(x,y,size)$提取当前需要处理的映射块 $\boldsymbol{A}'=\boldsymbol{A}[x:x+size,y:y+size]$.对该矩阵块 $\boldsymbol{A}'$进行分片、尾数压缩与对齐位优化,获得一个提供最终映射信息的矩阵 $\boldsymbol{A}''$,其矩阵元素为(符号位,压缩与对齐后的有效数二进制串,指数)三元组.通过这个信息矩阵不难将矩阵块按位片映射在表 1 所示的存储簇结构上,存储关键数据信息.需注意,处理过程中要根据“符号位”的数值为 1 或 0,选择将其映射在负数阵列集合还是正数阵列集合中.

3) 阵列运算.完成映射后,需要根据当前存储簇结构存储的 y 和 *size* 值,提取需要相乘的向量片段 $\boldsymbol{x}'=\boldsymbol{x}[y:y+size]$.然后,对 $\boldsymbol{x}'$进行分片与对齐,按照位片从高到低加时序地载到存储簇的电压端.对于每一次向量位片的加载,经过流水乘加缩减树的运算整合,输出其与矩阵块的乘积,即该向量位片对应的部分和.

4.2.3 部分和求和的实现

1) 部分和求和计算.每加载一次 $\boldsymbol{x}$ 位片,乘加缩减数就输出一次部分和.在此,采用动态策略进行部分和的求和运算,即每产生 1 次部分和,就进行 1 次加法运算,每一次求和计算:

$$\begin{cases}\boldsymbol{b}_{c_{\text{pos}},j_{\text{sum}}}=\boldsymbol{b}_{c_{\text{pos}},j_{\text{sum}}}\ll 1+\boldsymbol{b}_{c_{\text{pos}},j}, & \text{处理正数阵列},\\ \boldsymbol{b}_{c_{\text{neg}},j_{\text{sum}}}=\boldsymbol{b}_{c_{\text{neg}},j_{\text{sum}}}\ll 1+\boldsymbol{b}_{c_{\text{neg}},j}, & \text{处理负数阵列},\end{cases} \tag{4}$$

其中,$c_{\text{pos}}(c_{\text{neg}})$为当前进行部分和求和的存储簇 c 对应的正数(负数)阵列集合,j 代表加载第 j 个 $\boldsymbol{x}$ 位片,向量 $\boldsymbol{b}$ 则为该系统最后需要求得的结果,即 $\boldsymbol{Ax}$ 的结果.$\boldsymbol{b}_{c_{\text{pos}},j}$($\boldsymbol{b}_{c_{\text{neg}},j}$)则代表当前存储簇中正数(负数)阵列集合在第 j 个 $\boldsymbol{x}$ 位片的作用下得到的部分和结果;$\boldsymbol{b}_{c_{\text{pos}},j_{\text{sum}}}$($\boldsymbol{b}_{c_{\text{neg}},j_{\text{sum}}}$)代表前 j 个部分和结果相加得到的最终结果.

2) 部分和求和的提前终止策略.在上述过程中,随着输入的 $\boldsymbol{x}$ 位片指数降低,其在结果中的权值降低.事实上,在后续二进制转浮点的操作中,结果中多余的低位片数值将会被截断.本工作利用文献[11]提到的方法,应用部分和求和提前终止策略,设定结果最多保留位数为 m(一般 $m=53$),则当结

果 $\boldsymbol{b}_{c_{\text{pos}},j_{\text{sum}}}$($\boldsymbol{b}_{c_{\text{neg}},j_{\text{sum}}}$)的高 m 位稳定不变时,求和过程结束.记结束时,当前存储簇 c 中 c_{pos} 和 c_{neg} 对应的部分和求和结果是 $\boldsymbol{b}_{c_{\text{pos}}}$ 和 $\boldsymbol{b}_{c_{\text{neg}}}$.

求和终止需要满足 3 个条件:1)后续求和区域与高 m 位没有重叠.2)高 m 位后面出现一个值为 0 的阻隔位,用于吸收后续求和元素产生的进位.3)值为 0 的阻隔位出现之后,需要再计算 1 次,若阻隔位没有变成 1,则求和才能终止.这一步是为了避免阻隔位翻转让后续计算向高 m 位产生进位.部分和求和终止时的形式如图 13 所示:

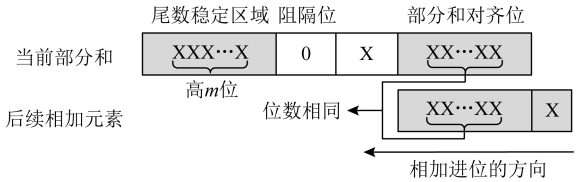


Fig. 13 Partial sum early termination strategy

图 13 部分和求和提前终止策略

4.2.4 定点数转浮点数

在部分和的求和结束之后,需要将向量和矩阵块的乘积由二进制转换为浮点数.由于正数和负数阵列集合求和结果 $\boldsymbol{b}_{c_{\text{pos}}}$ 和 $\boldsymbol{b}_{c_{\text{neg}}}$ 的求和过程是完全对称的,即它们在处理负数电流时的偏移量完全一致.所以对二者作二进制减法就可直接消除偏移,即 $\boldsymbol{b}'_c=\boldsymbol{b}_{c_{\text{pos}}}-\boldsymbol{b}_{c_{\text{neg}}}$.要转换成浮点数,另外一个重要的因素是指数,科学乘法计算的核心是尾数相乘、指数相加.对于本工作中的情况,计算指数:

$$expval_c=len\_c+min\_x+min\_c-mantissa\_bits, \tag{5}$$

其中,len_c 表示 $\boldsymbol{b}'_c$ 中向量元素二进制串的长度,min_x 为 $\boldsymbol{x}$ 的最低位片指数,min_c 为该存储簇对应元素的最低二进制指数,$mantissa\_bits$ 是用户规定的尾数保留位数,m 是部分和求和时尾数稳定区域的位数.于是,根据 $expval_c$ 和 $\boldsymbol{b}'_c$ 的值,不难将二进制串转换为浮点数,记结果浮点数结果向量为 $\boldsymbol{b}_c$.

4.2.5 不同存储簇结果与未分块元素的整合

4.2.2~4.2.4 节所述的阵列计算、部分和的求和计算,均是针对同一个存储簇中的运算而言的.要获得 $\boldsymbol{Ax}$ 的结果,需要将不同的存储簇中的结果整合起来.在这一过程中,需要全局地维护结果向量 $\boldsymbol{b}$.

1) 不同存储簇结果的整合.遍历所有的存储簇进行运算,都会产生对应的 MVM 运算结果 $\boldsymbol{b}_c$,对所有存储簇 c 整合到 $\boldsymbol{b}$ 中:

$$b[i+c.x]=b[i+c.x]+b_c[i],\quad i=0,1,\cdots,c.size-1. \tag{6}$$

2) 未分块元素整合.计算的最后一步是将 $unblock\_list$ 中的元素整合到结果向量 $\boldsymbol{b}$ 中,对于 $unblock\_list$ 中的第 i 个元素(x_i,y_i,val_i)进行计算(假设 $unblock\_list$ 的长度为 len_u):

$$b[x_i]=b[x_i]+x[y_i]\times val_i,\quad i=0,1,\cdots,len\_u-1. \tag{7}$$

4.3 预处理部分开销分析

整个系统的开销主要包括预处理开销与运算开销.其中,运算开销主要包括模拟电路开销(以运算阵列开销为主)与数字电路开销(以 ADC 转换开销为主).运算开销的评估与分析将在第 5 节作详细讨论,本节主要对预处理部分的关键开销进行定性分析与评估.

预处理部分的主要开销来自于原始矩阵分块开销、确定块内指数范围开销、构造叶子结点数可变的乘加缩减树开销这 3 个方面.其中,后 2 个开销是原系统[11]中没有涉及到的,即是由于应用了尾数压缩和对齐位优化策略,系统在预处理部分多出的开销.

1) 原始矩阵分块开销.假设一个矩阵中非 0 元素的个数为 N_{nz},按照本文提出的算法,将用 4 种不同大小的块对原矩阵的非 0 元素密集部分进行捕捉,每个元素最多遍历 4 次,即在最坏情况下需要进行 $4\times N_{\text{nz}}$ 次操作.已有的工作发现,在平均情况下,对于原始矩阵分块的开销大约为 $1.8\times N_{\text{nz}}$ 次操作[11].

2) 确定块内指数范围开销.为确定实际需要激活的运算对齐位个数,需求得每个矩阵块元素中最大指数和最小指数的差值,这就要遍历矩阵块中所有非 0 元素.由于矩阵块捕捉的都是原始矩阵中非 0 元素的密集区域,假设原矩阵中非 0 元素密度为 K,边长为 l,则这一步骤总开销(对整个原始矩阵)与 Kl^2 成比例.

3) 构造叶子结点数可变的乘加缩减树开销.对于每个新的测试负载,其所需的求解精度各不相同,即可压缩的尾数位数和设置的对齐位阈值不同,这意味着每次应用新的负载需要重新编程构造 1 棵树.假设其叶子结点数为 n(尾数位和对齐位之和为 n),则树高为 $\lceil \text{lb}\ n \rceil$,初始化整棵树的开销与 $n\times\lceil \text{lb}\ n \rceil$ 成比例.

4.4 实现上的限制分析

本系统在实现时会遇到一些限制,本节将从硬件物理实现和实际测试负载应用 2 个角度讨论.

1) 硬件物理实现.本文所提出系统的实现与评估都是通过编写软件程序模拟仿真实现的.这是由于目前的忆阻阵列制备工艺尚不成熟,不能够支持如此大规模、拥有多阵列尺寸大小的忆阻系统的物理实现.但使用软件实现的模拟系统在评估测试中也有其独特的优势,可以灵活调节软件参数来改变分块大小、压缩位数等重要参数,快速得到各种情况下的理论结果.

2) 实际测试负载应用.尚未找到一种方法,能对所有的测试负载都找到效果可观的尾数压缩和对齐位优化方案.实际应用该系统时,可能会由于部分参数选择不当(如尾数压缩位数、对齐位阈值),达不到需要的优化效果或精度要求.

5 评估与分析

本节将在线性代数求解框架中集成所实现的系统,对不同求解精度下的各种能耗节省比例进行评估.

5.1 评测数据与评估方法

下面将对模拟 MVM 内核集成到算法框架中的方法、评估所用的数据以及评估方式作详细论述.

5.1.1 MVM 内核的集成方法

1) 算法求解框架.常用的高性能线性代数迭代求解算法有共轭梯度法(conjugate gradient method, CG)[19]、双共轭梯度稳定法(biconjugate gradient stabilized method, BICGSTAB)[20].在本课题中,将使用 Python 的 scipy.sparse.linalg 库中已经实现的 CG/BICGSTAB 算法框架来进行测试评估.这种算法框架支持以线性运算符(linear operator)的形式将模拟 MVM 内核应用于迭代求解,算法框架(以 CG 为例):

$$\boldsymbol{x}=cg(\boldsymbol{A},\boldsymbol{b},\boldsymbol{x}_0,tol,maxtier,\boldsymbol{M},callback),$$
$$flag=\begin{cases}0, \text{ 函数 } cg \text{ 迭代收敛},\\ \text{其他值, 函数 } cg \text{ 未迭代收敛}.\end{cases} \tag{8}$$

在输出端元素中,$\boldsymbol{x}$ 为线性系统的解,$flag$ 为收敛标志(0 代表收敛).在输入端参数中,$\boldsymbol{A}$ 代表原始矩阵或者线性运算符,$\boldsymbol{b}$ 代表线性方程组的右手向量,$\boldsymbol{x}_0$ 代表解向量的初始值,默认为零向量;tol 代表解的容差,即 $|\boldsymbol{b}-\boldsymbol{A}\boldsymbol{x}|<tol$ 时则迭代停止;$maxiter$ 是最大迭代次数;$\boldsymbol{M}$ 是预条件子矩阵;$callback$ 是每次迭代结束时调用的函数,在本工作中可以用这个函数统计算法迭代次数.以上输入参数在后续测试中,如未特殊说明,都取软件求解时设定的默认值.

2) 运用线性运算符集成 MVM 内核.在式(8)中,输入参数 $\boldsymbol{A}$ 可以是线性运算符,其作用是包装一个函数,函数输入为 $\boldsymbol{x}$,返回 $\boldsymbol{A}\boldsymbol{x}$ 的结果给算法求解框架.Python 的 scipy.sparse.linalg 库提供线性运算符的包装方案,其框架:

$$\boldsymbol{A}=LinearOperator(\boldsymbol{A}.shape,matvec,rmatvec), \tag{9}$$

其中,$matvec$ 和 $rmatvec$ 是 2 个关键函数,它们的输入都是 $\boldsymbol{x}$,返回分别为 $\boldsymbol{A}\boldsymbol{x}$ 和 $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{x}$.为实现 MVM 内核的集成,需编写这 2 个函数,在函数体中调用实现的 MVM 模拟系统,通过输入的向量 $\boldsymbol{x}$ 和固定的矩阵 $\boldsymbol{A}$,计算得到对应的 $\boldsymbol{A}\boldsymbol{x}$ 或 $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{x}$,作为函数的返回.

3) 预条件子的使用.在式(8)中,输入参数 $\boldsymbol{M}$ 代表预条件子矩阵.预条件子是在解决大型稀疏矩阵求解问题时的一种常用优化手段[21-22].关于预条件子矩阵,通俗的解释就是需要找到一个和稀疏矩阵 $\boldsymbol{A}$ 相似的矩阵 $\boldsymbol{M}$,对于对原等式 $\boldsymbol{A}\boldsymbol{x}=\boldsymbol{b}$ 作出 $\boldsymbol{M}^{-1}\boldsymbol{A}\boldsymbol{x}=\boldsymbol{M}^{-1}\boldsymbol{b}$ 的变换.其中 $\boldsymbol{M}^{-1}\boldsymbol{A}$ 和 $\boldsymbol{M}^{-1}\boldsymbol{b}$ 都可以直接算出,而且由于 $\boldsymbol{M}$ 和 $\boldsymbol{A}$ 为相似的矩阵,所以 $\boldsymbol{M}^{-1}\boldsymbol{A}$ 的值接近于单位矩阵,极大程度上减少了求解的步数.有一种通用的预条件子方法为 0 级不完全 LU 分解(incomplete LU factorization with level-0 fill in, ILU(0)),该方法利用单位下三角矩阵 $\boldsymbol{L}$ 和上三角矩阵 $\boldsymbol{U}$ 构造 $\boldsymbol{M}=\boldsymbol{L}\boldsymbol{U}$,使得 $\boldsymbol{M}$ 接近于 $\boldsymbol{A}$,并在矩阵中执行 0 级填充,保持矩阵原有的稀疏模式.在 Python 的 scipy.sparse.linalg 库中,有 spilu 这样一个求解类,可以用于构造 ILU(0)预条件子,并集成到算法框架中.在后续测试中,存内硬件求解器都将集成 ILU(0)预条件子.

5.1.2 评测数据集

评测将采用 SuiteSparse 矩阵集合[2]中的矩阵作为数据集.SuiteSparse 矩阵集合是科学与工程领域中的实际应用产生的稀疏矩阵集合,该集合被广泛地用于稀疏矩阵算法的开发和性能评估.

根据需要,选用了大小不一的 6 个稀疏矩阵,作为评测数据.其中,部分数据文件只给出了矩阵 $\boldsymbol{A}$ 的值但是没有给出右手向量 $\boldsymbol{b}$ 的值.对于这部分数据,在进行 CG/BICGSTAB 算法求解时参照文献[21]的做法,构造全是 1 的右手向量 $\boldsymbol{b}$ 进行测试.表 3 显示了 6 个矩阵的名称、矩阵行数($rows$)、非 0 元素数量(N_{nz})、平均每行非 0 元素个数($N_{\mathrm{nz}}/rows$)等基本信息.

为了后续评估不同求解精度下的能量消耗比例,需要一些有关于忆阻器件的参数来辅助进行能耗评定.表 4 列举出了一些真实的忆阻器件参数[9,23].

Table 3 Evaluated Matrices Dataset

表 3 评测矩阵数据集

测试矩阵	*rows*	N_{nz}	$N_{nz}/rows$
msc00726	726	34 518	47.55
Chem97ZtZ	2 541	7 361	3.00
mesh3em5	289	1 377	4.76
685_bus	685	3 249	4.74
nos5	468	5 172	11.05
bcsstk27	1 224	56 126	45.85

Table 4 Memristive Device Parameters

表 4 忆阻器件参数

参数	数值	表示意义
R_{on}/kΩ	2	二进制位为 1 时映射的电阻值
R_{off}/MΩ	3	二进制位为 0 时映射的电阻值
V_{read}/V	0.2	二进制位为 1 时电压端的值

5.1.3 评估方法

在下面的评估中,主要考虑只进行对齐位优化、尾数保留 35 b、尾数保留 25 b、尾数保留 15 b 这 4 种优化策略(3 种尾数保留策略也是在对齐位优化的基础上进行).在忆阻 MVM 运算中,这 4 种策略所激活的阵列个数依次减少,会导致求解精度下降,但是会获得极大的能耗节约.

为了评估所设计的自选尾数压缩机制和动态对齐位优化机制的有效性和效果,将在 CG 和 BICGSTAB 这 2 种算法下评估 4 种策略的 3 个指标:1)相对于软件求解基线的相对残差及对应的迭代次数.2)运算阵列能耗相对于能耗基线的优化程度.3)ADC 能耗相对于能耗基线的优化程度.其中,能耗基线指在原系统[11]对应的条件下(尾数 53 b、对齐位 64 b 固定不变)测得的对应种类的能耗(在本文的测试中,对齐位阈值也选择为 64 b).

通过评估这 3 个指标,可以分析出随着激活运算阵列的减少,计算精度的下降(通过计算对于软件求解基线的相对残差来体现求解的精确程度),运算阵列和 ADC 能耗的优化程度.验证所提出的系统是否可以在为高精度应用提出无损浮点计算功能的同时,又能有效降低较低精度应用的能耗开销,以体现系统实现方案的有效性和良好效果.

5.2 不同策略下的求解精度与迭代次数

由于只进行对齐位优化的情况下并不会影响 MVM 的计算精度,即此策略下的解向量与迭代次数与软件求解完全一致,所以不再对其进行单独测定.本节主要关注 5.1 节提到的 3 种尾数压缩策略.

5.2.1 评估标准

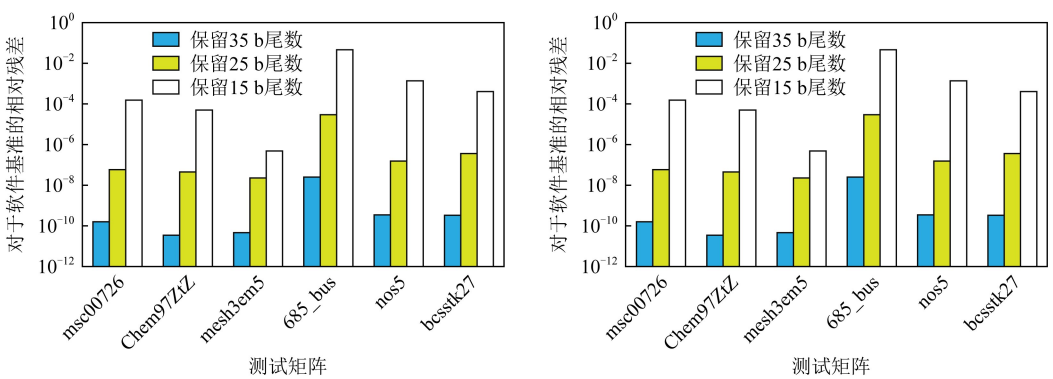
用残差来描述不同策略下求解精度相对于软件求解基线的差距,求解残差:

$$\varepsilon=\frac{|\boldsymbol{x}-\boldsymbol{x}_0|}{|\boldsymbol{x}_0|}. \tag{10}$$

5.2.2 评估结果与分析

本节展示在各种尾数压缩策略下,求解结果相对于软件基线的相对残差以及算法求解的迭代次数.

1) 各种尾数压缩策略下解的相对残差.图 14 分别显示了在 BICGSTAB(图 14(a))、CG(图 14(b))测试算法下,3 种尾数压缩策略相对于软件基准的相对残差.总体来说,在对 6 个数据集测试结果进行对数平均处理后,3 种策略的相对残差分别在 10^{-9},10^{-7},10^{-3} 这一数量级.具体地,对于同一种尾数压缩策略,2 种算法解向量的相对残差是几乎相同的,可以认为压缩策略影响的是映射矩阵本身的性质,不会因为算法框架的改变而发生太大的结果变化.对于每一种算法的每个测试矩阵而言,随着保留位数的减少,相对残差都会成数量级地增大.

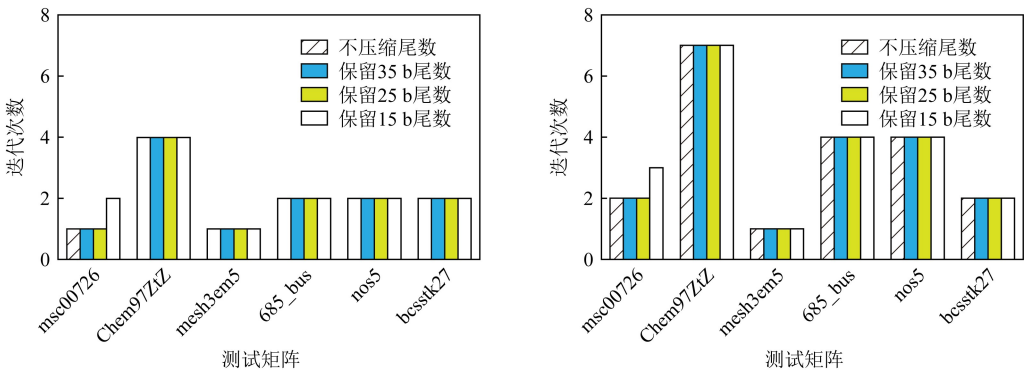


(a) BICGSTAB算法下各种尾数压缩策略的相对残差 (b) CG算法下各种尾数压缩策略的相对残差

Fig. 14 Relative residuals of solution vectors under various compaction strategies

图 14 各种压缩策略下的解向量的相对残差

2) 各种尾数压缩策略下算法的迭代次数.图 15 显示了在 2 种测试算法下各策略对应的算法迭代次数.总体来说,进行适当范围内的尾数压缩不会对迭代次数产生大的影响.但是,如果压缩的位数太多,保留位数太少,可能会增大迭代次数(如图 15 中保留 15 b 尾数时的测试矩阵 msc00726 的测试结果).为了进一步验证该想法,分别继续测试了测试矩阵 msc00726 在只保留 15 b,12 b,9 b,6 b 这 4 种尾数位情况下 2 种算法的迭代次数.结果显示,BISGTAB 算法的迭代次数对应分别为 2,2,3,5,CG 算法的迭代次数对应分别为 3,3,4,9.可以看出,如果尾数保留位数过少,会显著增加求解迭代次数,使得迭代求解过程需要更多的时间和能耗.故在实际应用中,需要谨慎选择尾数压缩位数,尽量不选择过少的保留位数(如 15 b 以下),以防止尾数压缩策略给求解带来的负面影响.



(a) BICGSTAB算法下各种尾数压缩策略的迭代次数 (b) CG算法下各种尾数压缩策略的迭代次数

Fig. 15 Algorithm iteration times under various compaction strategies

图 15 各种压缩策略下的算法迭代次数

5.3 不同优化策略下的运算阵列能耗评估

运算阵列能耗是模拟电路中最大的能耗来源.进行阵列运算时,电压加载到电阻上,会产生瞬时功率.为了使得采样—保持数组能够获得正确的电流值,通常需要使得电流维持一段时间,所以会产生能量消耗.本节将展示在 4 种优化策略下,运算阵列能耗相对于能耗基线的减少比例.能耗基线的值是在已有未优化系统[11]的条件下评估得到的,其尾数为固定的 53 b,对齐位为固定的 64 b.

5.3.1 评估指标

参数如表 4 所示,每一个存储单元的瞬时功率:

$$P=\begin{cases}\dfrac{V_{\text{read}}^2}{R_{\text{on}}}, & \text{映射值为 1.}\\ \dfrac{V_{\text{read}}^2}{R_{\text{off}}}, & \text{映射值为 0.}\end{cases} \tag{11}$$

采样—保持数组需要电流维持的时间主要取决于其 RC 时间常数,它与分辨率,即 lb N 是成比例的,其中 N 是当前忆阻存储阵列的大小(行数),即一个存储单元的计算能耗与 $P\times$lb N 是成比例的[11].

为了获得进行一次 $\boldsymbol{Ax}$ 运算所对应的运算阵列能耗 E_{crossbar},需要依次进行 3 个具体步骤计算:

1) 对于加载 1 次电压(1 个向量位片)而言,需要对所有阵列中的所有存储单元的能耗值进行求和,得到忆阻存储结构的整体能耗;

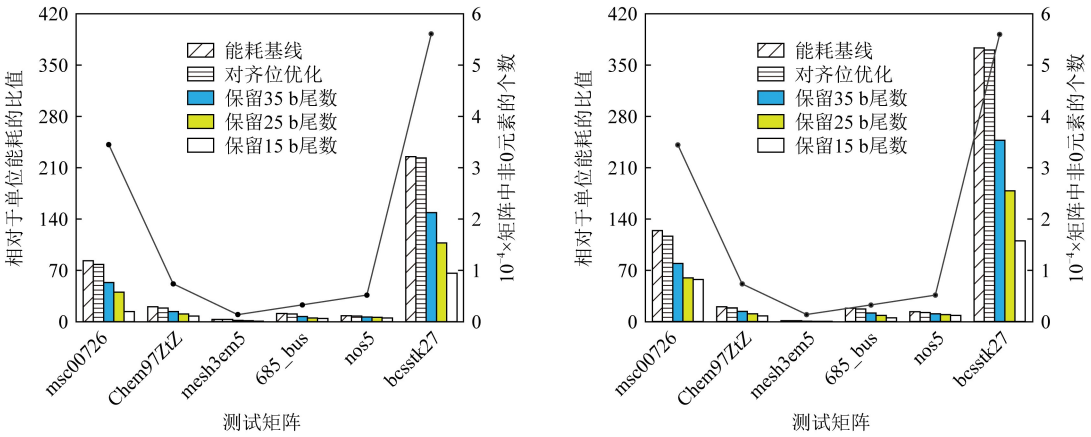
2) 对于每一次向量位片的加载,将上述整体能耗进行累加,得到 E_{crossbar};

3) 为了求解迭代过程中的运算阵列总能耗,需要对所有 $\boldsymbol{Ax}$ 步骤的 E_{crossbar} 求和,得到一个与迭代阵列消耗总能量成比例的值 $E_{\text{crossbar\_tol}}$.

5.3.2 评估结果与分析

评估求解过程中,各策略 $E_{\text{crossbar\_tol}}$ 的相对比例.

1) 各种压缩/优化策略下的运算阵列能耗比例.以测试矩阵 mesh3em5 在 BICGSTAB 算法下的测试得到的 $E_{\text{crossbar\_tol}}$ 为单位能量,画出在 2 种算法下,所有矩阵在各种策略下的能耗值,如图 16 所示.首先,对于同一个算法同一个测试矩阵而言,随着每个存储簇中激活运算的阵列越来越少(从只进行对齐位优化,到尾数保留位数分别只有 35 b,25 b,15 b),运算阵列消耗的能量也逐渐变少,符合预期.同时,可以观察到,矩阵中非 0 元素越多,消耗的阵列能量明显越多.这是因为其非 0 元素多,导致其映射电阻中 R_{on} 较多,导致能耗较大.并且注意到,消耗的运算阵列能量与原始矩阵大小没有直接关系,比如在 6 个测试集中,最大的矩阵(行数最多的)为测试矩阵 Chem97ZtZ,但是其中的非 0 元素没有测试矩阵 bcsstk27 多,所以消耗能量明显比测试矩阵 bcsstk27



(a) BICGSTAB算法下的运算阵列能耗比例 (b) CG算法下的运算阵列能耗比例

Fig. 16 Computation crossbar energy ratio under various strategies

图 16 各种策略下的运算阵列能耗比例

小很多.对于同一个测试矩阵而言,由于 CG 算法的迭代次数多于 BICGSTAB,所以消耗的运算阵列能量也成比例的增加.

2) 各种压缩/优化策略下的运算阵列平均能耗优化.表 5 展示了对于所有 6 个测试矩阵平均值而言,使用每一种策略时运算阵列能耗相对于能耗基线减少的比例.可以发现,BICGSTAB 和 CG 这 2 种方法的能量优化比例是几乎相同的,并且随着运算时激活阵列的减少,能耗优化比例会越来越大.具体地,在相对于软件方法有 0～10^{-3} 范围的求解残差时,2 种算法下的平均运算阵列能耗都相对已有的未优化系统减少了 5%～65%.

Table 5 Average Computation Crossbar Energy Optimization Ratio

表 5 运算阵列平均能耗优化比例

压缩策略	相对残差	运算阵列平均能耗优化比例/%	
		BICGSTAB	CG
对齐位优化	0	5.26	5.28
保留 35 b 尾数	10^{-9}	33.55	33.43
保留 25 b 尾数	10^{-7}	49.16	49.09
保留 15 b 尾数	10^{-3}	65.67	62.18

5.4 不同优化策略下的 ADC 能耗评估

ADC 是将采集-保持数组中的电信号转化为数字信号的部件,只有经过这种转换,乘加缩减树才能对阵列中的计算结果求和.ADC 能耗是外围数字电路能耗的主要来源.本节中将展示 4 种优化策略下的 ADC 的能耗相对于能耗基线(已有未优化系统[11]条件下评估得到)的减少.

5.4.1 评估指标

假设一个阵列的大小是 N,则其所需要的 ADC 分辨率为 $\mathrm{lb}\,N$.因为 ADC 的平均功耗随分辨率呈指数增长[24-26],故其功耗和 N 成比例.又因为 ADC 的转换时间与分辨率 $\mathrm{lb}\,N$ 成正比,所以列电流的转换能耗与 $N\times\mathrm{lb}\,N$ 成正比[11].

假设某个存储簇中有 M 个存储阵列.即对于每一个存储簇,每一个单独的向量 $\boldsymbol{x}$ 位片要进行 $M\times N$ 次列电流的计算,则存储簇进行一次计算 ADC 的总能耗与 $M\times N\times N\times\mathrm{lb}\,N$ 成正比.

为了获得一次 $\boldsymbol{Ax}$ 的完整 ADC 能耗 E_{ADC}.需要在算法执行过程中对所有存储簇,所有的向量 $\boldsymbol{x}$ 位片的 ADC 能量求和,得到最终 E_{ADC}.注意 E_{ADC} 的值只是与真实的 ADC 能耗成比例,而非真实 ADC 能耗值,其只会被用来计算相对于能耗基线的能量节省比例.

最后,为了计算求解迭代过程中的 ADC 总能耗,需要对所有 $\boldsymbol{Ax}$ 计算产生的 E_{ADC} 求和,得到求解整个算法 ADC 消耗的总能量值 $E_{\mathrm{ADC\_tol}}$,该值与迭代求解过程中真实的 ADC 能量消耗成比例.

5.4.2 评估结果与分析

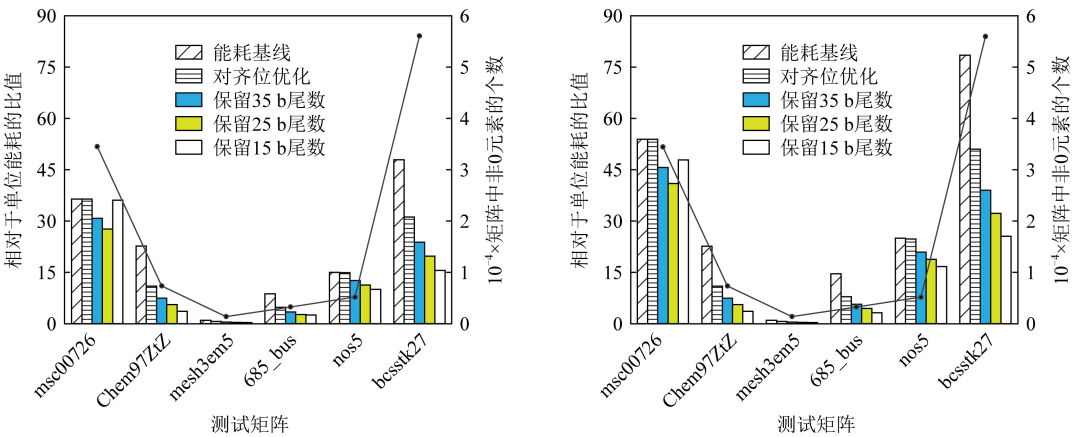
评估求解过程中,各策略下 $E_{\mathrm{ADC\_tol}}$ 的相对比例.

1) 各种压缩/优化策略下的 ADC 能耗比例.以测试矩阵 mesh3em5 在 BICGSTAB 算法下的测试得到的 $E_{\mathrm{ADC\_tol}}$ 为单位能量,画出所有矩阵在各种算法、各种执行策略下的能耗比例.图 17 显示出一些与运算阵列能耗优化(图 16)相同的性质.比如,矩阵块的非 0 元素越多,ADC 的能耗明显越多,因为需要更多的阵列进行运算,对应的 ADC 转换过程也就越多.

再比如由于 CG 算法的迭代次数明显比 BICGSTAB 算法要多,所以其 ADC 能耗也成比例的多.

对同一个算法同一个测试矩阵而言,随着每个存储簇中激活运算的阵列越来越少,ADC 能耗基本上也逐渐变少,符合预期.但是需要注意,对于测试矩阵 msc00726 和测试矩阵 nos5 而言,应用对齐位优化策略相对于能耗基线并没有一个明显的 ADC 能耗下降,这是因为其矩阵块内指数范围太大,超过了设定的阈值 64,对齐位优化效果不明显.

对于大多数矩阵而言,ADC 能耗随着保留的尾数位数的减少而降低.但是对于测试矩阵 msc00726 而言,在保留 15 b 尾数的时候,其 ADC 能耗反常地比保留 25 b 时要多,这是由于尾数截断过多使得算法迭代的次数增加,导致要用到更多计算步骤来使得算法收敛.所以须注意,由于压缩位数过多导致迭代次数增加,最终导致计算能耗过多的情况也会发生,对于 ADC 能耗来说可能更明显,需要谨慎选择压缩位数.



(a) BICGSTAB算法下的ADC能耗比例 (b) CG算法下的ADC能耗比例

Fig. 17 ADC energy ratio under various strategies

图 17 各种策略下的 ADC 能耗比例

2) 各种压缩/优化策略下的 ADC 平均能耗优化.表 6 展示了对于所有 6 个测试矩阵平均值而言,每种执行策略的 ADC 能耗相对于能耗基线减少的比例.和运算阵列能耗一样,各种策略在应用于 BICGSTAB 和 CG 这 2 种算法的情况下,对 ADC 能耗减少的比例几乎是一致的,且随着运算激活阵列数的减少,能量优化比例逐渐增大.具体地,在相对于软件方法有 0～10^{-3}范围的求解残差时,2 种算法下的平均 ADC 能耗都相对于已有的未优化系统减少了 30%～55%.

Table 6 Average ADC Energy Consumption Optimization Ratio

表 6 ADC 平均能耗优化比例

压缩策略	相对残差	ADC 平均能耗优化比例/%	
		BICGSTAB	CG
对齐位优化	0	27.66	28.29
保留 35 b 尾数	10^{-9}	43.06	43.67
保留 25 b 尾数	10^{-7}	51.68	52.23
保留 15 b 尾数	10^{-3}	53.55	57.23

与运算阵列能耗优化的情况不同的是,ADC 能耗在只应用对齐位优化时,其能耗优化程度约为 30%,它大于同样策略下的运算阵列优化的数值(约 5%).但是,随着存储簇中阵列激活数量(保留尾数位数)的逐渐减少,ADC 能量优化的效果没有运算阵列能量明显.运算阵列的能耗在保留 15 b 的时候优化比例在 65%左右,而 ADC 能耗在尾数保留位数 15 b 时优化比例只有 55%左右.

6 结束语

在科学与工程领域中,浮点数线性系统的求解问题十分普遍.现有的研究提出将 IEEE-754 双精度浮点数部署在忆阻阵列上,进行高能效的存内计算.然而,还没有研究提出为不同求解精度的浮点线性系统优化运算能耗.为解决这个问题,本文创新性地提出了一种具有尾数压缩与对齐位优化策略的浮点 MVM 计算系统,并针对该策略的特征提出了一种叶子结点数可变的流水乘加缩减树.评估结果表明:本系统既可进行高精度浮点数无损运算,又可对低精度求解应用大幅度优化运算阵列与 ADC 的能耗.

作者贡献声明:丁文隆提出了本文工作的思路,设计完成了实验,并撰写论文;汪承宁协助改进系统框架,在论文写作过程中提供理论基础的帮助;童薇对整个工作提出指导意见并修改论文.

参 考 文 献

[1] Saad Y. Iterative Methods for Sparse Linear Systems [M]. Boston: PWS Publishing Company, 2003

[2] Davis T A, Hu Yifan. The University of Florida sparse matrix collection [J]. ACM Transactions on Mathematical Software, 2011, 38(1): 1:1-1:25

[3] Vogelsberger M, Genel S, Springel V, et al. Properties of galaxies reproduced by a hydrodynamic simulation [J]. Nature, 2014, 509(7499): 177-182

[4] Bojnordi M N, Ipek E. Memristive Boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning [C] //Proc of the 22nd IEEE Int Symp on High Performance Computer Architecture (HPCA). Piscataway, NJ: IEEE, 2016: 1-13

[5] Shafiee A, Nag A, Muralimanohar N, et al. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars [J]. ACM SIGARCH Computer Architecture News, 2016, 44(3): 14-26

[6] Chi Ping, Li Shuangchen, Xu Cong, et al. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory [J]. ACM SIGARCH Computer Architecture News, 2016, 44(3): 27-39

[7] Song Linghao, Qian Xuehai, Li Hai, et al. Pipelayer: A pipelined ReRAM-based accelerator for deep learning [C] // Proc of the 23rd IEEE Int Symp on High Performance Computer Architecture (HPCA). Piscataway, NJ: IEEE, 2017: 541-552

[8] Song Linghao, Zhuo Youwei, Qian Xuehai, et al. GraphR: Accelerating graph processing using ReRAM [C] //Proc of the 24th IEEE Int Symp on High Performance Computer Architecture (HPCA). Piscataway, NJ: IEEE, 2018: 531-543

[9] Hu Miao, Strachan J P, Li Zhiyong, et al. Dot-product engine for neuromorphic computing: Programming 1T1M crossbar to accelerate matrix-vector multiplication [C/OL] // Proc of the 53rd ACM/EDAC/IEEE Design Automation Conf (DAC). Piscataway, NJ: IEEE, 2016 [2021-06-08]. https://ieeexplore.ieee.org/document/7544263

[10] Mao Haiyu, Shu Jiwu. 3D memristor array based neural network processing in memory architecture [J]. Journal of Computer Research and Development, 2019, 56(6): 1149-1160 (in Chinese)
(毛海宇, 舒继武. 基于 3D 忆阻器阵列的神经网络内存计算架构[J]. 计算机研究与发展, 2019, 56(6): 1149-1160)

[11] Feinberg B, Vengalam U K R, Whitehair N, et al. Enabling scientific computing on memristive accelerators [C] //Proc of the 45th ACM/IEEE Annual Int Symp on Computer Architecture (ISCA). Piscataway, NJ: IEEE, 2018: 367-382

[12] Chen Yunji, Luo Tao, Liu Shaoli, et al. Dadiannao: A machine-learning supercomputer [C] //Proc of the 47th Annual IEEE/ACM Int Symp on Microarchitecture. Piscataway, NJ: IEEE, 2014: 609-622

[13] Jouppi N P, Young C, Patil N, et al. In-datacenter performance analysis of a tensor processing unit [C] //Proc of the 44th Annual Int Symp on Computer Architecture. New York: ACM, 2017: 1-12

[14] Wang Chengning, Feng Dan, Tong Wei, et al. Cross-point resistive memory: Nonideal properties and solutions [J]. ACM Transactions on Design Automation of Electronic Systems, 2019, 24(4): 46:1-46:37

[15] Bailey D H. High-precision floating-point arithmetic in scientific computation [J]. Computing inScience and Engineering, 2005, 7(3): 54-61

[16] IEEE Standards Board, American National Standards Institute. ANSI/IEEE Std 754-1985: IEEE Standard for Binary Floating-Point Arithmetic [S]. Piscataway, NJ: IEEE, 1985

[17] Sun Zhong, Pedretti G, Bricalli A, et al. One-step regression and classification with cross-point resistive memory arrays [J]. Science Advances, 2020, 6(5): 2378:1-2378:8

[18] Sun Zhong, Pedretti G, Mannocci P, et al. Time complexity of in-memory solution of linear systems [J]. IEEE Transactions on Electron Devices, 2020, 67(7): 2945-2951

[19] Hestenes M R, Stiefel E. Methods of conjugate gradients for solving linear systems [J]. Journal of Research of the National Bureau of Standards, 1952, 7(6): 409-436

[20] Van der Vorst H A. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems [J]. SIAM Journal on Scientific and Statistical Computing, 1992, 13(2): 631-644

[21] Anzt H, Dongarra J, Kreutzer M, et al. Efficiency of general Krylov methods on GPUs—An experimental study [C] // Proc of the 30th IEEE Int Parallel and Distributed Processing Symp Workshops (IPDPSW). Piscataway, NJ: IEEE, 2016: 683-691

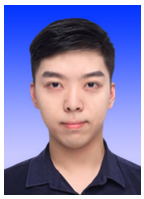
[22] Anzt H, Gates M, Dongarra J, et al. Preconditioned Krylov solvers on GPUs [J]. Parallel Computing, 2017, 68: 32-44

[23] Niu Dimin, Xu Cong, Muralimanohar N, et al. Design of cross-point metal-oxide ReRAM emphasizing reliability and cost [C] //Proc of the 32nd IEEE/ACM Int Conf on Computer-Aided Design (ICCAD). Piscataway, NJ: IEEE, 2013: 17-23

[24] Kull L, Luu D, Menolfi C, et al. 28.5 A 10b 1.5GS/s pipelined-SAR ADC with background second-stage common-mode regulation and offset calibration in 14nm CMOS FinFET [C] //Proc of the 64th IEEE Int Solid-State Circuits Conf (ISSCC). Piscataway, NJ: IEEE, 2017: 474-475

[25] Kull L, Toifl T, Schmatz M, et al. A 3.1 mW 8 b 1.2 GS/s single-channel asynchronous SAR ADC with alternate comparators for enhanced speed in 32 nm digital SOI CMOS [J]. IEEE Journal of Solid-State Circuits, 2013, 48(12): 3049-3058

[26] Saberi M, Lotfi R, Mafinezhad K, et al. Analysis of power consumption and linearity in capacitive digital-to-analog converters used in successive approximation ADCs [J]. IEEE Transactions on Circuits and Systems I: Regular Papers, 2011, 58(8): 1736-1748



Ding Wenlong, born in 1998. Bachelor. His main research interests include large-scale network and storage systems.

丁文隆,1998 年生.工学学士.主要研究方向为大规模网络与存储系统.

Wang Chengning, born in 1994. PhD. His main research interests include high-density memristive nanodevices and arrays, three-dimensional integrated memristive systems, and analogue parallel computation-in-memory for novel applications.

汪承宁,1994 年生.博士.主要研究方向为高密度忆阻纳米器件阵列、3 维集成忆阻系统和模拟并行存内运算的新应用.

Tong Wei, born in 1977. PhD, associate professor. Her main research interests include computer architecture, in memory/storage computing and non-volatile memory/storage.

童　薇,1977 年生.博士,副教授.主要研究方向为计算机系统结构、存算融合和非易失存储.