第44卷第5期　　国防科技大学学报　　Vol. 44 No. 5
2022年10月　　JOURNAL OF NATIONAL UNIVERSITY OF DEFENSE TECHNOLOGY　　Oct. 2022

doi:10.11887/j.cn.202205009　　http://journal.nudt.edu.cn

并行规约与扫描原语在ReRAM架构上的性能优化*

金　洲,段懿洳,伊恩鑫,戢昊男,刘伟峰
(中国石油大学(北京)信息科学与工程学院,北京　102249)

摘　要: 规约与扫描是并行计算中的核心原语,其并行加速至关重要。然而,冯·诺依曼体系结构下无法避免的数据移动使其面临"存储墙"等性能与功耗瓶颈。近来,基于ReRAM等非易失存储器的存算一体架构支持的原位计算可一步实现矩阵-向量乘,已在机器学习与图计算等应用中展现了巨大的潜力。提出面向忆阻器存算一体架构的规约与扫描的并行加速方法,重点阐述基于矩阵-向量乘运算的计算流程和在忆阻器架构上的映射方法,实现软硬件协同设计,降低功耗并提高性能。相比于GPU,所提规约与扫描原语可实现高达两个数量级的加速,平均加速比也可达到两个数量级。分段规约与扫描最大可达到五个(平均四个)数量级的加速,并将功耗降低79%。

关键词: 规约;扫描;ReRAM;存算一体架构;并行计算
中图分类号: TN95　　**文献标志码:** A　　**开放科学(资源服务)标识码(OSID):**
文章编号: 1001-2486(2022)05-080-12

Accelerating parallel reduction and scan primitives on ReRAM-based architectures

JIN Zhou, DUAN Yiru, YI Enxin, JI Haonan, LIU Weifeng
(College of Information Science and Engineering, China University of Petroleum, Beijing 102249, China)

Abstract: Reduction and scan are two critical primitives in parallel computing. Thus, accelerating reduction and scan shows great importance. However, the Von Neumann architecture suffers from performance and energy bottlenecks known as "memory wall" due to the unavoidable data migration. Recently, NVM (non-volatile memory) such as ReRAM (resistive random access memory), enables in-situ computing without data movement and its crossbar architecture can perform parallel GEMV (matrix-vector multiplication) operation naturally in one step. ReRAM-based architecture has demonstrated great success in many areas, e.g. accelerating machine learning and graph computing applications, etc. Parallel acceleration methods were proposed for reduction and scan primitives on ReRAM-based PIM (processing in memory) architecture, the computing process in terms of GEMV and the mapping method on the ReRAM crossbar were focused, and the co-design of software and hardware was realized to reduce power consumption and improve performance. Compared with GPU, the proposed reduction and scan algorithm achieved substantial speedup by two orders of magnitude, and the average acceleration ratio can also reach two orders of magnitude. The case of segmentation can achieve up to five (four on average) orders of magnitude. Meanwhile, the power consumption decreased by 79%.

Keywords: reduction; scan; ReRAM; processing in memory; parallel computing

规约和扫描是并行计算中的两个核心原语,对诸多并行计算的性能有着显著影响。并行规约是并行点积、并行矩阵乘法、计数等并行算法中的一个常见操作[1-2]。并行扫描的应用包括排序(快速排序和基数排序)和合并、图计算(如最小生成树、连通分量、最大流、最大独立集、双连通分量等),以及计算几何(如凸包、多维二叉树构建、平面最近点对等)[3-12]。此外,规约和扫描还对基本线性代数子程序中稀疏矩阵-向量乘(sparse matrix vector multiplication, SpMV)、稀疏矩阵-矩阵乘(general sparse matrix matrix multiplication, SpGEMM)等矩阵运算的性能具有显著影响[13],这些矩阵运算在气象预报、大规模电力系统仿真等科学与工程计算中具有广泛的应用。因此,规约与扫描原语的并行加速具有重要的研究价值。

加速规约与扫描原语的一般思路是利用CPU、GPU等多核或众核处理器以并行的方式执行算法,可以带来成倍的性能提升[14]。然而,传

* **收稿日期:** 2021-12-27
基金项目: 国家自然科学基金资助项目(61972415);计算机体系结构国家重点实验室开放课题资助项目(CARCHA202115)
作者简介: 金洲(1990—),女,江苏盐城人,讲师,博士,E-mail:jinzhou@cup.edu.cn;
刘伟峰(通信作者),男,教授,博士,博士生导师,E-mail:weifeng.liu@cup.edu.cn

统的冯·诺依曼体系结构中计算单元与存储单元之间频繁的数据吞吐会严重影响总体计算性能。与计算本身功耗相比,操作数移动功耗可达 10 倍之多[15]。同时,为了满足对内存带宽不断增长的需求,片外互连传输速率的增长速度快于每比特能量下降的速度,从而导致更高的峰值功耗[16]。

近来,采用基于忆阻器等新型非易失存储器件的存算一体技术逐渐成为一个新的热点研究内容[11]。由其构成的内存架构最显著的特点是具有原位计算[9]的能力,可从根本上避免数据的移动。其中,应用较为广泛的一种是氧化还原阻性存储器(resistive RAM, ReRAM)。相比其他非易失存储器,如相变存储器(phase-change RAM, PCM)、自旋转移扭矩存储器(spin-transfer torque RAM, STT-RAM)等,ReRAM 具有集成密度大、开关速度快、操作功耗低、开启/关闭阻值比高、多值存储、耐久性高,以及与 CMOS 制备工艺兼容性好等诸多优点,是更具吸引力的方案[11]。

基于 ReRAM 的存算一体架构可使用结构化的交叉阵列和基本的欧姆定律并行地计算矩阵向量乘法(matrix-vector multiplication, GEMV),可参见文献[16 - 18]。ReRAM 架构已在机器学习、图计算等依赖 GEMV 的计算密集型应用中展现了巨大的潜力。Chi 等[17]最早设计了基于 ReRAM 的深度学习加速器 PRIME。He 等[19]提出了考虑 ReRAM 状态和功耗之间关系的神经网络加速器 SARA。3DICT[20]实现了神经网络训练中的反向传播。在图计算方面,Dai 等[21]提出了一种在混合存储多维数据集阵列上进行图形处理的存内处理架构 GraphH;Huang 等[22]提出了基于三维 ReRAM 的大规模并行图处理加速器 RAGra; Nai 等[23]提出了一种图计算的全栈解决方案 GraphPIM。然而,如何将规约和扫描原语应用到只适合 GEMV 运算的 ReRAM 架构上,仍存在以下多个关键挑战。

1)如何将不依赖 GEMV 的规约与扫描原语在忆阻器阵列上实现加速,尤其是如何将不同长度的数据映射到固定尺寸的、只支持较小矩阵运算的交叉阵列上。

2)如何将科学计算中所需的较高精度数据映射到只支持 2 ~6 bit 的 ReRAM 单元上,并一步实现矩阵的乘加运算。

3)如何设计电路和架构使其尽可能地匹配加速算法并实现数据重用,减少擦写次数,避免写操作带来额外功耗。

本文面向基于 ReRAM 的存算一体架构,以矩阵乘法的形式进行规约和扫描算法的加速,将其不同规模问题高效地映射到固定尺寸的忆阻器阵列上,同时展示实现上述算法所需的电路设计与硬件架构,实现软硬件协同设计。并与当前最先进的 GPU 实现方法进行对比,验证了所提算法的高性能与低功耗。

1　研究背景

1.1　规约和扫描算法

将在这一小节简要介绍规约、扫描以及分段规约与分段扫描四个核心原语,简单回顾其基本并行算法以及当前最先进的 GPU 加速并行算法。首先,给出如下基本定义。

定义 1　给定长度为 n 的输入序列,如 x_1, x_2, x_3, …, x_n, 规约原语计算并输出结果:

$$\sum_{j=1}^{n} x_j \tag{1}$$

定义 2　给定长度为 n 的输入序列,如 x_1, x_2, x_3, …, x_n, 扫描原语计算并输出包含 n 个结果的序列:

$$x_1, x_1 + x_2, \cdots, \sum_{j=1}^{n} x_j \tag{2}$$

定义 3　给定输入序列包含 k 个长度为 m 的子段,分段规约原语计算并输出包含 k 个结果的序列:

$$\sum_{j=1}^{m} x_j, \sum_{j=m+1}^{2m} x_j, \cdots, \sum_{j=(k-1)m+1}^{km} x_j \tag{3}$$

定义 4　给定输入序列包含 k 个长度为 m 的子段,分段扫描原语计算并输出包含 km 个结果的序列:

$$x_1, x_1 + x_2, \cdots, \sum_{j=1}^{m} x_j, x_{m+1}, \cdots, \sum_{j=m+1}^{2m} x_j, \cdots,$$
$$x(k-1)m + 1, \cdots, \sum_{j=(k-1)m+1}^{km} x_j \tag{4}$$

规约与扫描的一种常见并行方式是将其拆分为多个可并行的子任务,并对其结果进一步规约或扫描,通过多次迭代得到最终结果。GPU 上的规约和扫描大多通过 warp 级、block 级以及 grid 级三个层次联合实现。一个 block 包含多个 warp,warp 级的规约结果在 block 级再次规约,在 grid 级对 block 级结果进行规约,并形成最终完整的规约结果。warp 级的规约和扫描操作通常通过 shuffle 指令来完成,算法 1 展示了其计算过程,一个 warp 中的多个线程通过寄存器来共享数据,而无须同步或共享内存。

算法 1 GPU 上使用 shuffle 的规约和扫描算法

Alg. 1 Reduction and scan algorithms on GPU with shuffle instructions

输入:待求解数列

输出:求解后返回值

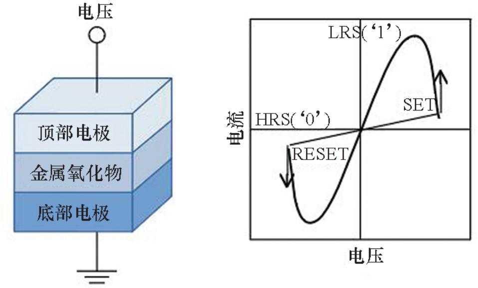
```
_device_half warp_reduce(half val){
  for(int offset = WARP_SIZE/2; offset > 0; offset/ = 2)
    val + = _shfl_down_sync(0xFFFFFFFFU, val, mask);
  return val;}
_device_half warp_scan(half val){
  for(int offset = 1; offset < WARP_SIZE; offset * = 2)
    auto n = _shfl_up_sync(0xFFFFFFFFU, val, mask);
    if(laneid > = offset)
      val + = n;
  return val;}
```

1.2 ReRAM

ReRAM 是一种新兴的非易失性存储器,具有高密度、快速读取和低漏功率等优点,是极具前途的构建未来存算一体体系结构的非线性元器件[9]。ReRAM 可通过改变单元电阻来存储信息,其中一种实现方式是以金属氧化层作为阻变特性材料,其金属 - 绝缘体 - 金属结构如图 1(a)所示,由顶部电极、底部电极以及夹在电极之间的金属氧化物层组成。通过施加外部激励,ReRAM 单元可以实现在高阻态(关闭状态)和低阻态(开启状态)之间切换,这两种状态可分别用于表示逻辑的 0 和 1。其伏安特性曲线如图 1(b)所示。

ReRAM 的 crossbar 交叉阵列[15]结构还可以天然地实现乘加运算,通过模拟计算的方式将 $O(n^2)$ 复杂度的 GEMV 计算变为 $O(1)$ 复杂度,极易应用到有大量矩阵向量乘计算的神经网络及神经形态芯片中。ReRAM 的交叉阵列结构如图 2 所示,将 ReRAM 单元连接到同一行的称为字线,连接到同一列的称为位线,电压从字线输入,电流从位线输出。将矩阵中的值编程为各 ReRAM 单元的电导 G,向量以电压的形式输入,连接到相同字线的 ReRAM 单元共享同一个输入电压 V_i,根据欧姆定律和基尔霍夫电流定律(Kirchhoff's current law, KCL),可得到流过各 ReRAM 单元的电流 $I = V \cdot G$,连接到相同位线的 ReRAM 单元输出的电流之和即为 GEMV 的一个结果。

ReRAM 的交叉阵列被证明可以有效地加速



(a) ReRAM 物理结构
(a) ReRAM physical structure

(b) ReRAM 电子特性
(b) ReRAM electrical characteristics

图 1 ReRAM 物理结构及电子特性

Fig. 1 ReRAM physical structure and its electrical characteristics

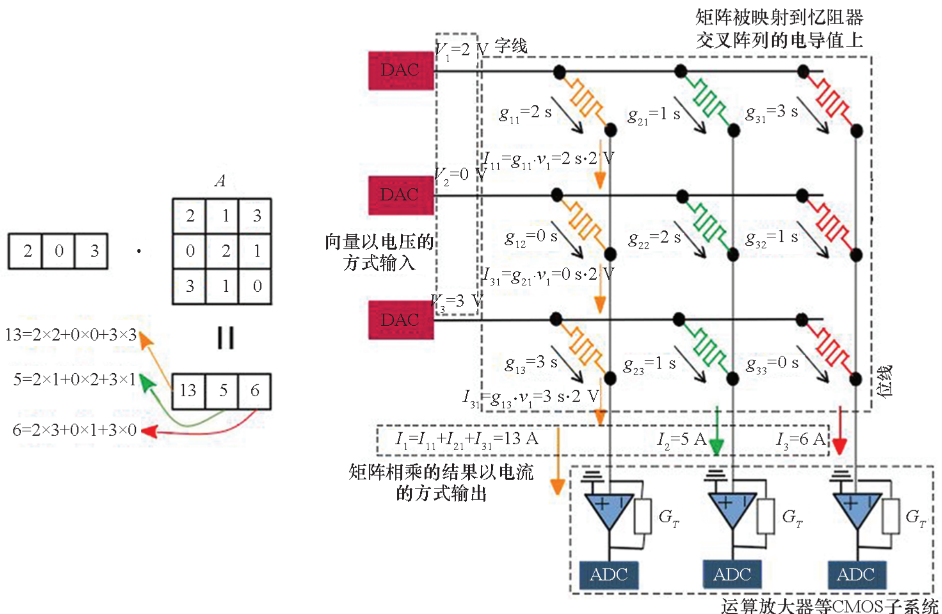


图 2 使用 ReRAM 的 crossbar 交叉阵列实现的乘加运算

Fig. 2 Implementation of GEMV operation through the ReRAM crossbar

神经网络计算,其中主要考虑卷积层和全连接层,全连接层的本质为矩阵向量乘,卷积层则可转化为矩阵矩阵乘,使得非常适合在 ReRAM 上进行计算,如将数值较为稳定的学习参数(卷积层的卷积核和全连接层的权值)映射到计算单元,利用 ReRAM 天然的并行计算优势提高训练效率[17]。ReRAM 也被用于处理图计算问题,通过稀疏矩阵压缩[15]、切片技术或剪枝[24],可将图计算中的矩阵运算映射到 ReRAM 阵列中,提高计算效率。总的来说,通过压缩稀疏格式和找到良好的映射方法,ReRAM 可以很好地提高神经网络、图计算等应用的效率。

2　并行加速算法

与 GPU 类似,在基于 ReRAM 存算一体架构上的并行规约与扫描操作也可以被划分到多个忆阻器阵列上并行计算,并将结果再次分配到多个忆阻器阵列上进行并行规约与扫描,通过多次迭代得到最终结果。因此,忆阻器阵列上的高效规约算法是忆阻器阵列间进一步并行的基础和前提。这里,将重点阐述忆阻器阵列上对不同长度数据进行规约与扫描操作的计算方法,即忆阻器阵列级别的规约与扫描算法,阵列间的操作则与阵列上类似。核心在于将扫描与规约运算转换为矩阵乘法或矩阵乘加的形式,映射到忆阻器阵列上。

2.1　规约

规约操作可较为直观地变为如式(5)所示矩阵向量乘的方式,但忆阻器的阵列是固定且较小的,无法将任意尺寸的规约操作直接映射并实现到忆阻器阵列上。本文将以 16×16 的忆阻器阵列为例,介绍适用于固定尺寸忆阻器架构的、利用 GEMV 实现的规约算法。该方法可扩展至任意尺寸的忆阻器阵列。

$$\boldsymbol{R}(x_1, x_2 \cdots x_n) = [1\ 1\ \cdots\ 1] \times \begin{bmatrix} x_1 & \cdots & 0 \\ \vdots & & 0 \\ x_n & \cdots & 0 \end{bmatrix} \tag{5}$$

首先,考虑较为简单的两种情况。①对包含 16 个子段(长度均为 16)总长为 256 的输入序列进行分段规约,其计算方式如图 3、图 4 所示,将输入序列以列优先存储的方式映射到忆阻器阵列上,则每个子段可被映射到忆阻器阵列的一列上,利用式(5)矩阵向量乘法的方式,以全 1 向量作为输入电压,每条位线上的输出电流即为分段规约的结果。②对子段长度为 256 的输入序列进行规约,并将元素以列优先的方式将其映射至忆阻器阵列上,复用上述情况①的计算过程,将情况①获得的结果再次以列优先存储的方式映射到忆阻器阵列上,与全 1 向量相乘,通过一次迭代即可获得最终结果,如图 5、图 6 所示,该过程共需完成两次 GEMV 运算。

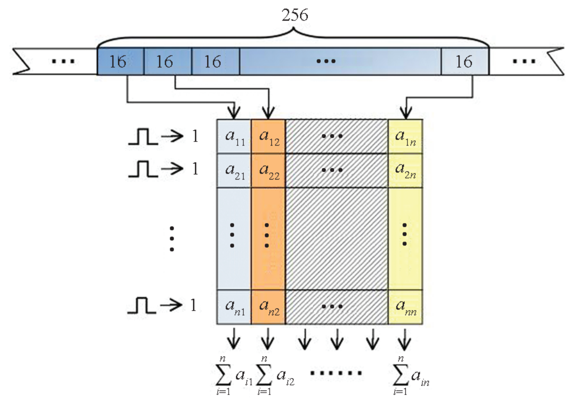


图 3　Reduction16 的计算方式(矩阵形态)

Fig. 3　Calculation method of reduction16(in matrix form)

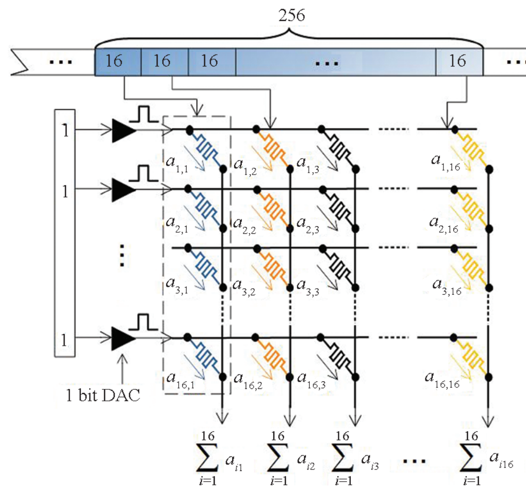


图 4　Reduction16 的计算方式(忆阻器形态)

Fig. 4　Calculation method of reduction16(in crossbar form)

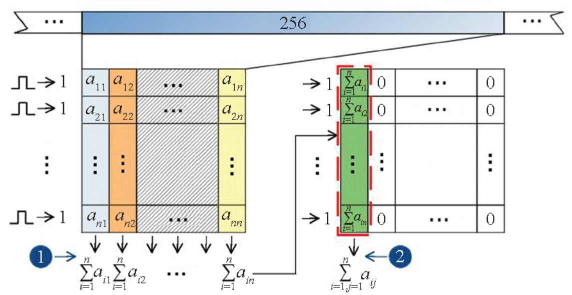


图 5　Reduction256 的计算方式(矩阵形态)

Fig. 5　Calculation method of reduction256(in matrix form)

有上述两个基本原语之后,任何子段规约均

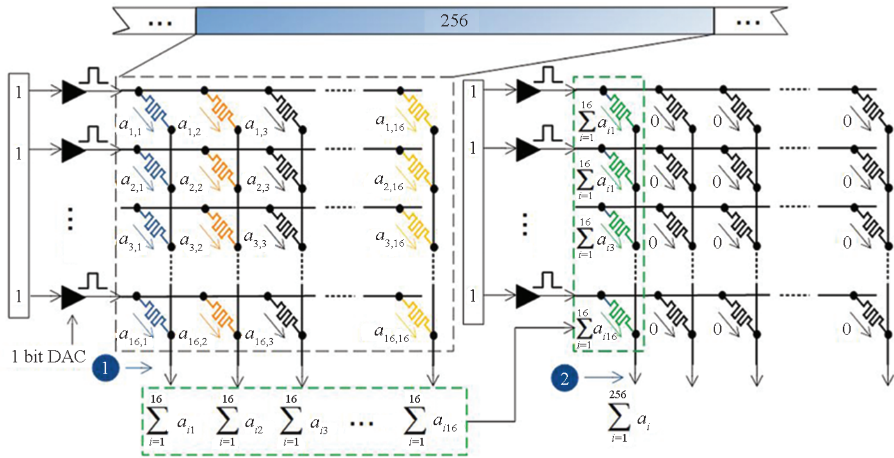


图 6 Reduction256 的计算方式(忆阻器形态)

Fig. 6 Calculation method of reduction256(in crossbar form)

可以表示为 16 倍数或 256 倍数长度。这里将分别考虑这两种情况:

1)一种对子段长度为 16 倍数的序列进行规约,将其中每 16 个元素看作一个基本单位块,如图 7、图 8 所示,以 $16N$ 作为长度间隔,分别取出长度为 16 的基本单位块,将其以列优先存储的方式写入第一次运算的 ReRAM 阵列(即放入第一列与第二列的数据之间间隔为 $16N$),接着取 $16N$ 中的第二块数据并再次以 $16N$ 为间隔,依次取出多个块写入下一次运算 ReRAM 阵列中,并与上一次运算的规约结果向量累加,则可得到每个 $16N$ 分段前两个块的部分规约结果。以此类推,直至做完为止。即取各子段中前 16 个元素依次映射为忆阻器阵列上的一列,与全 1 向量相乘,得到各子段前 16 个元素的规约结果后,对各子段中第二组 16 个元素以同样方式依次映射,并将第一次规约的结果映射至忆阻器阵列的第 $i+1$ 行($i=16$),再次与全 1 向量相乘,则可以得到各子段前 32 个元素规约结果。计算 16 个 $16N$ 长度的分段

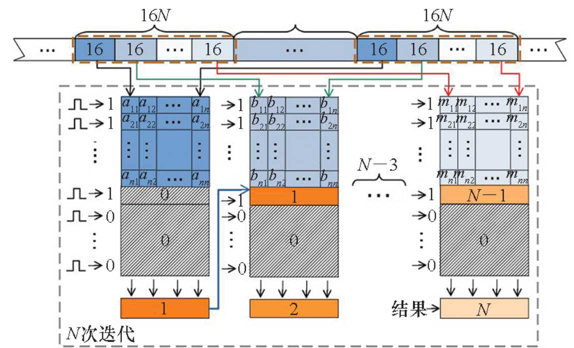


图 7 Reduction16N 的计算方式(矩阵形态)

Fig. 7 Calculation method of reduction16N(in matrix form)

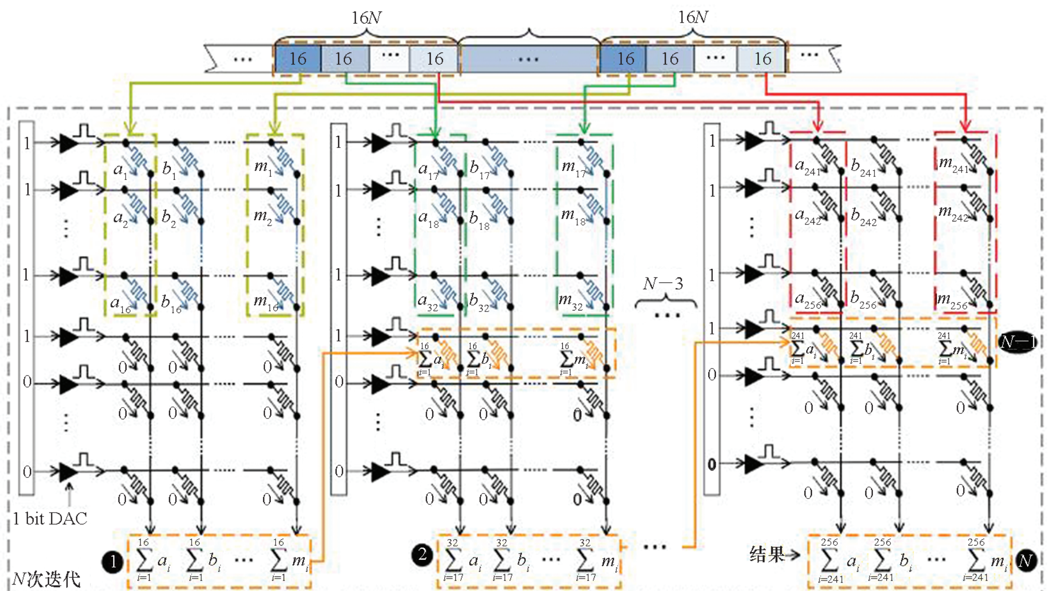


图 8 Reduction16N 的计算方式(忆阻器形态)

Fig. 8 Calculation method of reduction16N(in crossbar form)

规约共需 N 次迭代，其详细计算过程如算法 2 所示。这种方式更适合忆阻器阵列较少，单个忆阻器阵列上需计算数据的总长较长的情况。

算法 2　Reduction16N 的算法

Alg. 2　Reduction16N algorithm

输入：待求解数组 V，长度 V. length，分段 V. segment

输出：Reduction16N 的计算结果

```
for j = 0 → 16 do
    R16N[0] = Reduction(V16N[j][0])
end for
for i = 1→N do
  for j = 0 → 16 do
    R16N[j] + = 1 * Reduction(V16N[j][i])
  end for
end for
```

2）另一种是对子段长度为 256 倍数的输入序列进行规约，对 N 个长度为 256 的序列分别调用上述情况②的计算过程，并将每个 256 序列的规约结果标量累加到下一个 256 序列上。然而，该方法的缺点是共需 $2N$ 次乘加运算，每一个 256 长度的规约都多计算了一次矩阵乘法。如图 9、图 10 与算法 3 所示，将每一个 256 长度规约计算的结果映射至下一个 256 长度运算的忆阻器阵列的第 $i+1$ 行（$i=16$），与全 1 向量相乘后，前一个 256 长度规约计算的结果被直接累加在下一个 256 长度运算结果上，最终对累加后的结果进行一次规约操作即可，该方法进一步优化了计算过程，将运算次数减少为 $N+1$。

对于子段长度在（$256m$，$256(m+1)$）区间的序列还可以结合上述两种原语进一步优化性能，即利用 256 倍数原语对于 $256m$ 长度的数据进行规约，而对剩余序列则利用 16 倍数原语进一步加速。

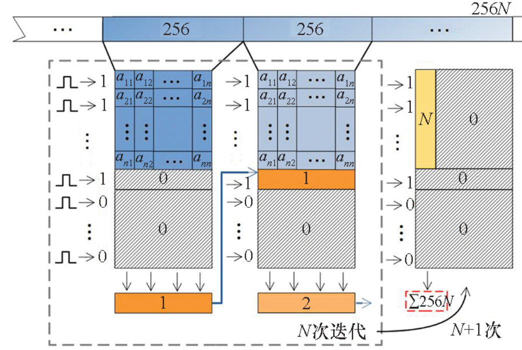


图 9　Reduction256N 的计算方式（矩阵形态）

Fig. 9　Calculation method of reduction256N (in matrix form)

此外，在此基础上，阵列间还可以进行并行规约，以 $256N$ 长度的序列为例，将数据切分成 N 个长度为 256 的序列，调用情况①则可得到 N 个 256 序列中以长度为 16 划分的分段结果，调用情况②将每个长度为 256 序列的部分结果写进忆阻器阵列的每一列，给定输入向量 1，则可得到 N 个以 256 为长度的子段规约结果，将 N 个结果再调用一次情况②，以此类推，则共需两次或多次迭代（$\log_{16}256N$ 次迭代）完成所需运算。可将阵列级规约方法和阵列间并行规约方法相结合，对不同规模的输入序列可自由选择多种不同算法进行组合，以达到最优性能。

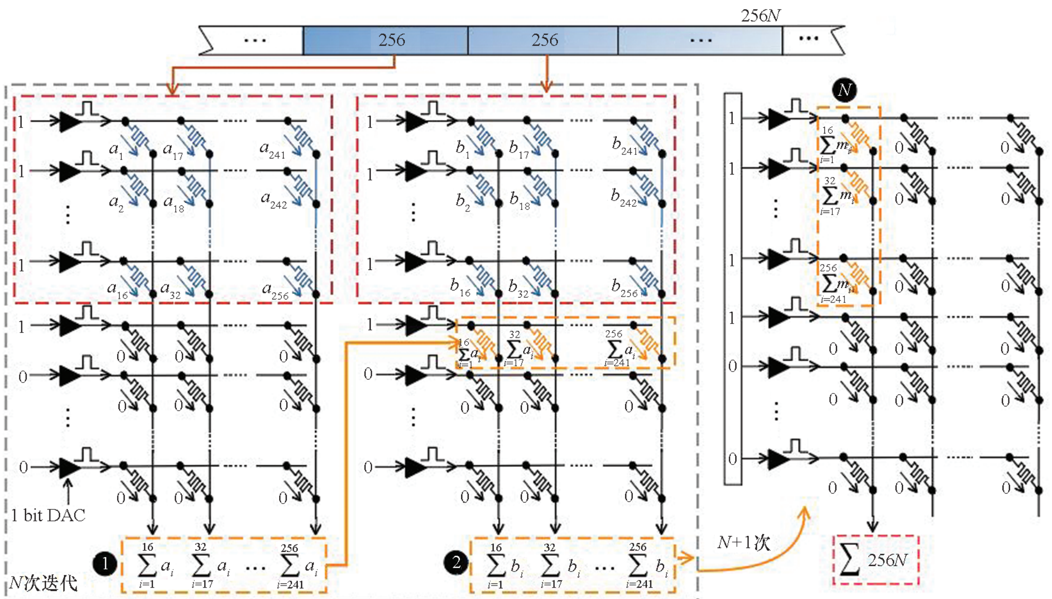


图 10　Reduction256N 的计算方式（忆阻器形态）

Fig. 10　Calculation method of reduction256N (in crossbar form)

算法 3　Reduction256N 算法

Alg. 3　Reduction256N algorithm

输入：待求解数组 V，V. length，V. segment

输出：Reduction256N 的计算结果

```
V1 = V[0] → V[255]
R16 = Reduction16(V1)
for i = 1 → N do
   V[1] = V[i * 256] → V[(i+1) * 256]
  R16_1 = Reduction16(V1)
  for j = 0 → 16 do
    R16[j] + = 1 * R16_1[j]
  end for
end for
R256N = Reduction256(R16)
```

2.2　扫描

忆阻器阵列级的扫描算法如图 11、图 12 和算法 4 所示，与规约算法不同的是，该方法将输入序列以行优先存储的方式映射至忆阻器阵列。首先将以行优先存储方式形成的矩阵 $\boldsymbol{C}$ 与数值均为 1 的上三角矩阵相乘，可得到每行数据的前缀和结果，记为矩阵 $\boldsymbol{CU}$。接着以数值为 1 的下三角矩阵（对角线为 0）与矩阵 $\boldsymbol{C}$ 相乘，得到矩阵 $\boldsymbol{LC}$，其中第一行为 0，第 2 ~ n 行是 $\boldsymbol{C}$ 矩阵的前（$n-1$）行在列上的前缀和结果。最后，将 $\boldsymbol{LC}$ 矩阵与全 1 矩阵相乘，则可在每一行的每个位置上均得到该行的规约结果，即为每一行在其之前的所有数据的总和，将得到的结果加上 $\boldsymbol{CU}$ 矩阵即可得到最终扫描原语的结果。通过两次矩阵乘法与一次矩阵加法共三个步骤，即可完成扫描的计算任务。以 16 × 16 的矩阵运算为例，则该运算可得到分段为 256 的序列前缀和结果，若分段长度为 16，则仅需阵列级扫描算法即可完成 16 个子段长度为 16 的序列的扫描

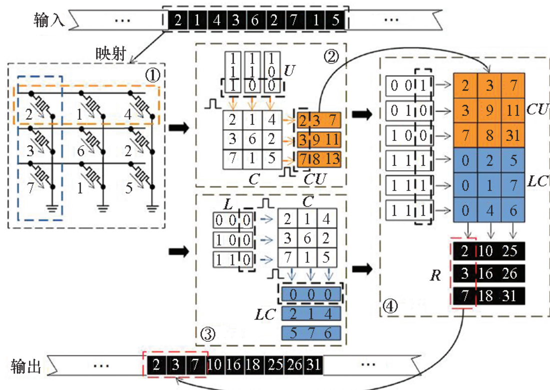


图 11　基于 ReRAM 架构阵列级扫描算法（矩阵形态）

Fig. 11　Array level scan algorithm ReRAM-based architecture (in matrix form)

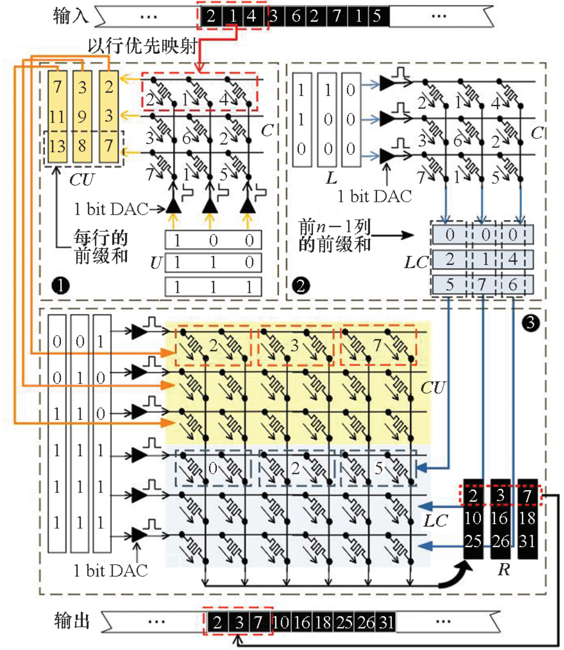


图 12　基于 ReRAM 架构阵列级扫描算法（忆阻器形态）

Fig. 12　Array level scan algorithm ReRAM-based architecture (in crossbar form)

计算。对于子段为 16 倍数或 256 倍数序列的规约与扫描原语的计算思想类似，这里不再赘述。

算法 4　分段扫描算法

Alg. 4　Segment scan algorithm

输入：待求解数组 V，阵列规模 N：V. length

输出：分段扫描计算结果

```
for i = 1→N - 1 do
  for j = 0→N - 1 do
    C[i][j] = V[k + +]
  end for
end for
for i = 1→N - 1 do
  for j = 0→N - 1 do
    CU[i][j] = C[i][j] * U[i][j]
    LC[i][j] = L[i][j] * C[i][j]
  end for
end for
k = 0
for i = 1→N - 1 do
  for j = 0→N - 1 do
    R[i][j] = LC[i][j] * 1[i][j] + CU[i][j]
    V[k + +] = R[i][j]
  end for
end for
```

阵列间的并行扫描方法如图 13 所示（以迭代一次作为示例），将输入序列划分到多个忆阻器阵列上，对每个划分利用阵列级扫描原语进行操作，

收集各划分序列的最大数(最后一个结果)形成新的较短待扫描序列,迭代上述过程,直至最终扫描序列长度≤256(即 1 次可完成扫描操作的序列长度),将扫描结果逐个累加至上一次迭代所扫描序列的各数值上(即将每个位置扫描结果矩阵的值均与上一次迭代相应矩阵进行加法操作),重复回代加法的操作,直至迭代展开至最原始的序列,则得到整个序列的扫描结果。完整的计算流程如算法 5 所示。

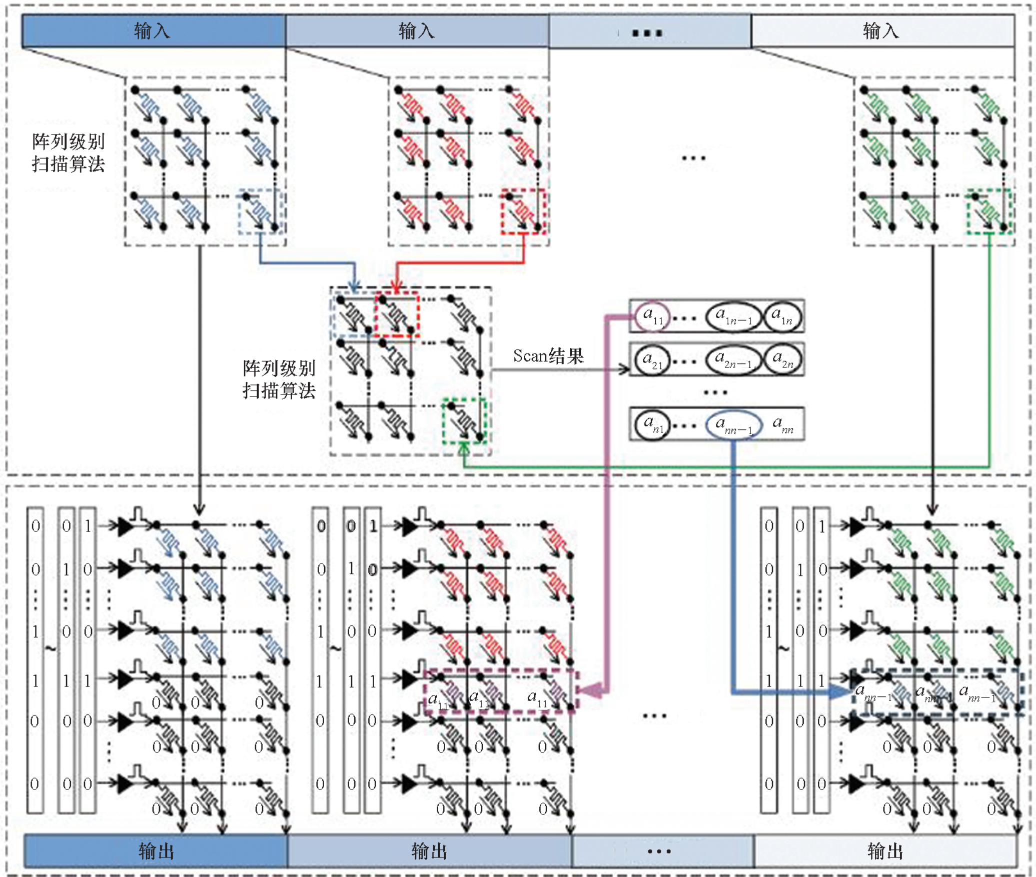


图 13　阵列间的并行扫描算法

Fig. 13　Bank level parallel scan algorithm

算法 5　阵列间并行扫描算法

Alg. 5　Bank_level parallel scan algorithm

输入:待求解数组 V, 阵列规模 N: V. length
输出:扫描结果

```
k = 0
V_max[k] = GetMax(Segment scan(V))
While(V_max[k]).length > 256 do
    V_max[++k] = GetMax(Segment scan(V_max[k]))
end while
for i = k→1 do
    len = V_max[i].length
    for j = 0→len do
        for l = j * len→(j + 1) * len do
            V_max[i-1][l] += V_max[i][j]
        end for
    end for
end for
for i = 0→V_max[0].length do
    V[i] = V_max[0][i]
end for
```

2.3　忆阻器架构及映射

上述算法涉及多个如 $\boldsymbol{R}=\boldsymbol{A}\times\boldsymbol{B}+\boldsymbol{C}$ 模式的矩阵乘加运算,对于该类型运算可如图 14 所示,将矩阵 $\boldsymbol{B}$ 映射到 ReRAM 交叉阵列的上半部分,其中每两个 ReRAM 单元表示一个数据,矩阵 $\boldsymbol{C}$ 以相同方式被映射到 ReRAM 交叉阵列的下半部分,将 $\boldsymbol{A}$ 矩阵作为输入向量的上半部分,下半部分以对角为 1 的矩阵补齐,通过如上操作,可一步实现矩阵的乘加操作。值得注意的是,本文所有的矩阵乘法、加法、乘加运算均可以这种方式映射到 ReRAM 交叉阵列上一步实现,即只需控制不同的输入向量即可,如纯粹的矩阵乘法将 $\boldsymbol{C}$ 矩阵或 $\boldsymbol{C}$ 矩阵对应的输入向量设为 0 即可。

此外,为满足科学计算等应用中对规约与扫描原语较高精度的需求,可将数据通过比特切片的方式映射到多个忆阻器阵列上,在单个交叉阵列上则利用两个相邻单元来存储相同数据(如

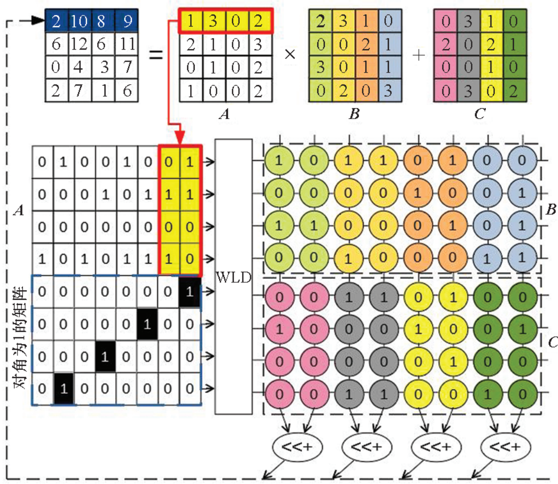


图 14　矩阵到 ReRAM 阵列的映射

Fig. 14　Mapping matrix to ReRAM crossbar

16×16 的矩阵可被映射到 16×32 的忆阻器阵列上），并通过移位与加法实现完整的运算得到最终结果。以 32 bit 精度的数据为例，该数据切片方式与上述映射方式相结合，则 16×16 矩阵乘加运算可在 8 个 ReRAM 阵列上一步实现（单个 ReRAM 单元存储 2 bit）。

图 15 展示了基于上述电路设计的加速规约与扫描原语的存算一体系统架构，与上文所述的计算流程、映射方法以及数据切片相结合，达到软硬件协同设计的目标。该架构包含多个 bank，每个 bank 包含多个计算单元，每个计算单元包含多个 ReRAM 阵列，此外，还包含控制单元、输入输出 buffer、数据 buffer 和互联，以及数字模拟转换器（digital-to-analog converter, DAC）、模拟数字转换器（analog-to-digital converter, ADC）外围电路与移加器等多个组成部分。单个 ReRAM 阵列可选择从字线或位线输入，多个 ReRAM 阵列之间可并行地进行运算。

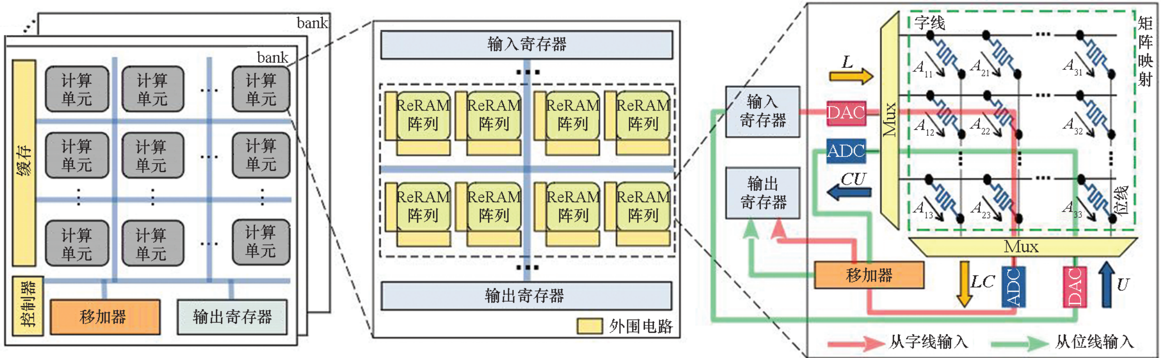


图 15　ReRAM 加速器的架构

Fig. 15　Architecture of ReRAM accelerator

在扫描原语中，步骤②和步骤③（见图 11）两步中涉及对相同矩阵 ***C*** 的操作，可将 ***C*** 矩阵在 ReRAM 阵列上复用，而无须进行两次较为耗时的 ReRAM 写操作，其对应的电路设计如图 15 所示，通过片选决定输入与输出的端口。将 ***C*** 矩阵编程写入 ReRAM 阵列，步骤②需对矩阵 ***C*** 的行进行点积操作，而步骤③需对 ***C*** 矩阵的列进行点积操作。在步骤②中将 ***U*** 矩阵中的列向量以电压的形式从位线输入，在字线收集电流即为计算结果的一列；在步骤③中，将 ***L*** 矩阵的行向量以电压的方式从字线输入，在位线获得输出结果的一行。通过该方式无须多次擦写即可实现对 ***C*** 矩阵的行或列有选择地进行点积。该方式显著增加数据的复用，减少写操作的次数，提高性能减少功耗。

3　实验结果与分析

3.1　实验环境

本文将所提方法实现在了基于 ReRAM 的忆阻器阵列上，利用 HSPICE 仿真电路行为，NvSim 仿真忆阻器阵列架构的延迟与功耗，以及高级语言 C++ 模拟了规约与扫描原语的性能与功耗。如表 1 所示，本文中的忆阻器架构包含 128 个 bank，每个 bank 包含 128 个计算单元，每个计算单元包含 64 个 ReRAM 阵列，ReRAM 阵列尺寸为 32×32，每个 ReRAM 单元可存储 2 bit，DAC 精度为 2 bit。ReRAM 的读延迟为 1.332 ns，写延迟为 20.362 ns，每个 ReRAM 阵列的功耗为 15.153 mW。此外，本文还对比了十核 CPU 和 GPU 上的规约与扫描算法，在 Inter Core i7 和 GeForce RTX 2080 硬件环境下，对 thrust 库[25]进行了性能测试。输入数据为 32 位 int 型。忆阻器架构上的并行算法可根据输入序列长度在文中展示的多个优化算法间进行调优，即将多大长度的规约与扫描任务分配到多少忆阻器阵列上，如何划分单个忆阻器阵列上需处理的子段长度，如何选择合适的忆阻器阵列级的算法等，均可根据不同情况进行选择，这

里展示不同输入参数下观测到的最佳性能。

表 1　ReRAM 架构配置

Tab. 1　Configuration of ReRAM architecture

组成结构	个数
bank	128
计算单元/bank	128
ReRAM 阵列/计算单元	64
ReRAM 阵列尺寸	32 × 32

3.2　算法性能分析

本文将规约和扫描算法分别在 ReRAM 加速架构和 GPU、CPU 上进行性能对比,共分为两组对比实验:第一组对比总长变化的规约和扫描算法;第二组对比总长不变情况下,子段长度变化的分段规约和扫描算法。通过两组对比实验,可以看到本文中所提出的算法在 ReRAM 加速架构上所获得的良好加速效果。

理论上,对 256 长度的数据在 GPU 上实现 warp 级规约,基于 shuffle 指令的操作通常需要进行 8 次 32 个元素规约的迭代,共需要 256 个时钟周期,因为每个 shuffle 指令和加法需要 4 个周期,而在 ReRAM 存算一体架构上则只需要 2 个时钟周期。此外,基于 ReRAM 的存算一体架构单次时钟周期的时间与功耗也较低。

如图 16 和表 2 所示,对于总长改变的扫描原语,CPU、GPU 和 ReRAM 架构的性能都随着数据段的增长而缓慢增长,而 ReRAM 架构在数据段相同的情况下,最高比 CPU 加速 75 302.82 倍,比 GPU 加速 906.66 倍, CPU 平均加速 25 075.76 倍,GPU 平均加速 302.49 倍(图中横坐标为取 log2 对数处理的数据总长,纵坐标为取 log10 对数的每秒通量数据)。同样地,如图 17和表 3 所示,对于总长改变的规约算法,ReRAM 架构最高比 CPU 加速 81 258.97 倍,比 GPU 加速 454.81 倍, CPU 平均加速 30 090.39 倍, GPU 平均加速 152.38 倍。

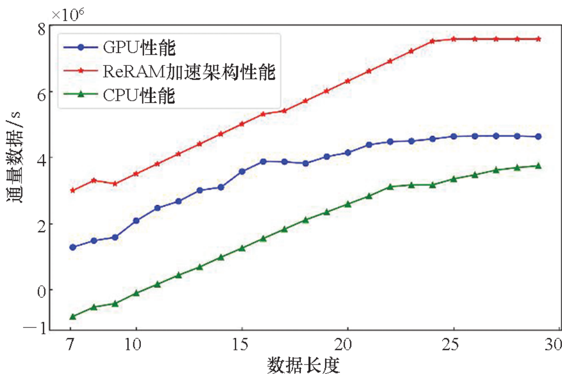


图 16　总长可变的扫描算法在 CPU、GPU 和 ReRAM 架构的性能对比

Fig. 16　Performance comparison of scan algorithm with variable total length on CPU、GPU and ReRAM architectures

表 2　扫描算法在 GPU 和 ReRAM 架构上性能对比

Tab. 2　Performance comparison of the scan algorithm on GPU and ReRAM architectures

数据长度	GPU/s	ReRAM/s	加速比
2^{7}	1.95×10^{7}	1.02×10^{9}	52.17
2^{8}	3.09×10^{7}	2.03×10^{9}	65.77
2^{9}	3.90×10^{7}	1.63×10^{9}	41.73
2^{10}	1.24×10^{8}	3.25×10^{9}	26.31
2^{11}	2.99×10^{8}	6.50×10^{9}	21.77
2^{12}	4.78×10^{8}	1.30×10^{10}	27.22
2^{13}	1.02×10^{9}	2.60×10^{10}	25.40
2^{14}	1.27×10^{9}	5.20×10^{10}	40.83
2^{15}	3.82×10^{9}	1.04×10^{11}	27.22
2^{16}	7.65×10^{9}	2.08×10^{11}	27.22
2^{17}	7.52×10^{9}	2.60×10^{11}	34.59
2^{18}	6.70×10^{9}	5.20×10^{11}	77.68
2^{19}	1.07×10^{10}	1.04×10^{12}	96.96
2^{20}	1.42×10^{10}	2.08×10^{12}	146.86
2^{21}	2.45×10^{10}	4.16×10^{12}	169.54
2^{22}	3.05×10^{10}	8.32×10^{12}	273.30
2^{23}	3.17×10^{10}	1.66×10^{13}	524.49
2^{24}	3.67×10^{10}	3.33×10^{13}	**906.66**
2^{25}	4.40×10^{10}	3.89×10^{13}	883.61
2^{26}	4.47×10^{10}	3.89×10^{13}	869.88
2^{27}	4.53×10^{10}	3.89×10^{13}	857.13
2^{28}	4.47×10^{10}	3.89×10^{13}	868.84
2^{29}	4.36×10^{10}	3.89×10^{13}	892.05

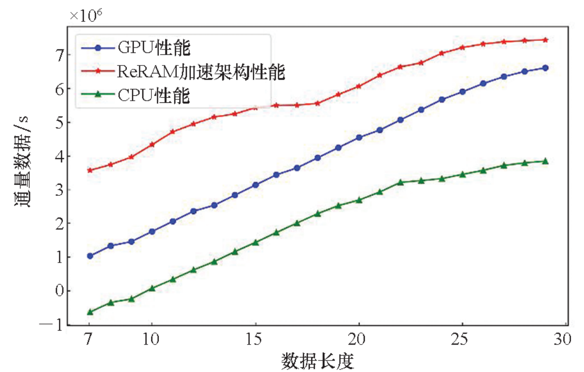


图 17　总长可变的规约算法在 CPU、GPU 和 ReRAM 架构的性能对比

Fig. 17　Performance comparison of reduction algorithm with variable total length on CPU、GPU and ReRAM architectures

表 3　规约算法在 GPU 和 ReRAM 架构上性能对比

Tab. 3　Performance comparison of the reduction algorithm on GPU and ReRAM architectures

数据长度	GPU/s	ReRAM/s	加速比
2^7	1.08×10^7	3.77×10^9	347.80
2^8	2.17×10^7	5.57×10^9	257.07
2^9	2.89×10^7	9.31×10^9	322.50
2^{10}	5.78×10^7	2.18×10^{10}	377.40
2^{11}	1.15×10^8	5.25×10^{10}	**454.81**
2^{12}	2.31×10^8	8.90×10^{10}	385.60
2^{13}	3.46×10^8	1.44×10^{11}	414.92
2^{14}	6.93×10^8	1.78×10^{11}	257.07
2^{15}	1.39×10^9	2.75×10^{11}	198.74
2^{16}	2.77×10^9	3.18×10^{11}	114.81
2^{17}	4.44×10^9	3.22×10^{11}	72.64
2^{18}	8.87×10^9	3.66×10^{11}	41.29
2^{19}	1.77×10^{10}	6.70×10^{11}	37.76
2^{20}	3.55×10^{10}	1.18×10^{12}	33.14
2^{21}	5.91×10^{10}	2.49×10^{12}	42.13
2^{22}	1.18×10^{11}	4.42×10^{12}	37.38
2^{23}	2.36×10^{11}	5.83×10^{12}	24.64
2^{24}	4.73×10^{11}	1.12×10^{13}	23.76
2^{25}	8.11×10^{11}	1.66×10^{13}	20.49
2^{26}	1.42×10^{12}	2.12×10^{13}	14.93
2^{27}	2.27×10^{12}	2.47×10^{13}	10.86
2^{28}	3.24×10^{12}	2.67×10^{13}	8.22
2^{29}	4.13×10^{12}	2.78×10^{13}	**6.73**

对于总长不变，即总长度为 2^{29} 的数据段，不同分段时规约和扫描算法的性能，ReRAM 加速器都比传统 GPU 快 3～5 个数量级，尤其在小分段的情况下，ReRAM 架构可以达到 4～5 个数量级的加速效果。图 18 为总长不变，分段 size 为 2^{15} 到 2^{28} 扫描算法的性能比较，相比 GPU，最高加速比为 92 342.39倍，平均性能加速比为 13 261.39 倍。相比 CPU，最高加速比为 34 759.34 倍，平均加速比为 15 925.73倍。图 19 为总长不变，分段 size 为 2^{15} 到 2^{28} 规约算法性能比较，相比 GPU 最高加速比为 367 733.57倍，平均性能加速比为 55 680.80 倍。相比 CPU，最高加速比为 372 421.71 倍，平均性能加速比为 173 853.25 倍。综上，ReRAM 架构展现了明显的加速优势，尤其是对于分段规模较小的问题可达到较大的性能提升效果，这类规模较小的分段规约与扫描原语在数值计算、神经网络中具有极为广泛的应用场景。此外，相比 GPU，ReRAM 加速架构上的功耗减少了 79%。

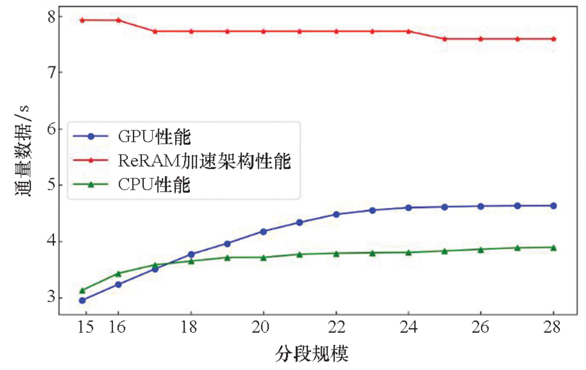


图 18　总长不变、分段长变化的扫描算法在 CPU、GPU 和 ReRAM 架构的性能对比

Fig. 18　Performance comparison of scan algorithm with constant total length and variable segment length on CPU、GPU and ReRAM architectures

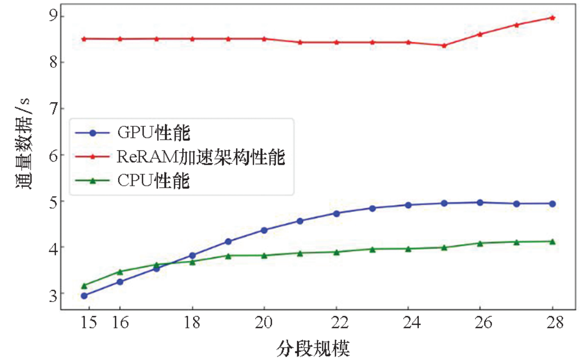


图 19　总长不变、分段长变化的规约算法在 CPU、GPU 和 ReRAM 架构的性能对比

Fig. 19　Performance comparison of reduction algorithm with constant total length and variable segment length on CPU、GPU and ReRAM architectures

本文针对不同规模的输入序列均给出了较为适用的方法。对于不分段的规约与扫描来说，尽可能填满所有 ReRAM 阵列并进行迭代统筹的方式最为高效，其所需时钟周期是 16 为底的对数函数（16 为单次可运算的矩阵阶数）。当分段规模大于等于 256 时，分段尺寸和交叉阵列规模也是影响计算结果的因素。以规约为例，假设对数据总长为 2^x 的序列做分段规约，分段长度为 2^k，ReRAM 阵列共 M 个，字段长度为 16 倍数的规约原语需要计算 $(2^{x-k-4}\% M+1)\times 2^{k-4}$ 次时钟周期，而字段长度为 256 倍数的规约原语则需计算 $(2^{x-k}\% M+1)\times(2^{k-8}+1)$ 次时钟周期。根据所给定字段的长度和个数，可利用公式快速定位最优算法。对于分段尺寸较小的情况，例如子段长为 32、64 等，采用字段长度为 16 倍数的规约原语会获得较好的性能，而对于较大的分段规模，如 512、1 024 等，则是利用 256 倍数的计算方式较为高效。此外，还可根据情

况对文中的不同算法进行组合。

4　结论

规约与扫描是并行计算中的核心原语,对科学计算、机器学习等诸多应用均具有显著的性能影响。本文面向忆阻器阵列的存算一体架构,首次将规约与扫描以矩阵向量乘的形式实现并映射到忆阻器阵列上,提出将任意长度的输入序列映射到固定尺寸的忆阻器阵列上的多种高效算法,以及与之相适应的电路设计与存算一体架构,尽可能地实现数据重用,避免写操作,实现了软硬件协同设计。此外,还对不同尺寸的输入序列选择何种算法可达到最优性能进行了仿真与分析。与 GPU 上的实现相比,所提算法实现了多个数量级的性能提高,对于总长不变的分段规约与扫描,性能最高可加速 5 个数量级;总长改变的情况下,规约最高可加速 454. 81 倍,平均加速可达 152. 38 倍,扫描最高可加速 906. 66 倍,平均加速 302. 49 倍,同时降低了 79% 的功耗。

参考文献(References)

[1] NVIDIA Corporation & Affiliates. CUDA toolkit documentation[EB/OL]. [2021 – 11 – 14]. https://docs.nvidia.com/cuda/cub/index.html.

[2] HARRIS M, SENGUPTA S, OWENS J D. Parallel prefix sum (scan) with CUDA[M]//Nguyen H. GPU Gems 3. New Jersey: Addison Wesley, 2007: 851 – 876.

[3] YAN S G, LONG G P, ZHANG Y Q. StreamScan: fast scan algorithms for GPUs without global barrier synchronization[C]//Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2013: 229 – 238.

[4] WANG X Y, YANG J L, ZHAO Y L, et al. TCIM: triangle counting acceleration with processing-in-MRAM architecture[C]//Proceedings of 57th ACM/IEEE Design Automation Conference, 2020: 1 – 6.

[5] DOTSENKO Y, GOVINDARAJU N K, SLOAN P P, et al. Fast scan algorithms on graphics processors[C]//Proceedings of the 22nd Annual International Conference on Supercomputing, 2008: 205 – 213.

[6] SENGUPTA S, HARRIS M, ZHANG Y, et al. Scan primitives for GPU computing[C]//Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, 2007: 97 – 106.

[7] LONG Y, NA T, MUKHOPADHYAY S. ReRAM-based processing-in-memory architecture for recurrent neural network acceleration[J]. IEEE Transactions on Very Large Scale Integration Systems, 2018, 26(12): 2781 – 2794.

[8] YANG X X, YAN B N, LI H, et al. ReTransformer: ReRAM-based processing-in-memory architecture for transformer acceleration[C]//Proceedings of IEEE/ACM International Conference on Computer Aided Design, 2020: 1 – 9.

[9] 陈怡然, 李海, 陈逸中, 等. 神经形态计算发展现状与展望[J]. 人工智能, 2018, 5(2): 46 – 58.
CHEN Y R, LI H, CHEN Y Z, et al. Current status and prospects of neuromorphic computing[J]. AI-View, 2018, 5(2): 46 – 58. (in Chinese)

[10] 季宇, 张悠慧, 郑纬民. 基于忆阻器的近似计算方法[J]. 清华大学学报(自然科学版), 2021, 61(6): 610 – 617.
JI Y, ZHANG Y H, ZHENG W M. Approximate computing method based on memristors [J]. Journal of Tsinghua University (Science and Technology), 2021, 61(6): 610 – 617. (in Chinese)

[11] LI S C, XU C, ZOU Q S, et al. Pinatubo: a processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories [C]//Proceedings of 53nd ACM/EDAC/IEEE Design Automation Conference, 2016: 1 – 6.

[12] ANKIT A, HAJJ I E, CHALAMALASETTI S R, et al. PANTHER: a programmable architecture for neural network training harnessing energy-efficient ReRAM [J]. IEEE Transactions on Computers, 2020, 69(8): 1128 – 1142.

[13] JI H N, LU S B, HOU K X, et al. Segmented merge: a new primitive for parallel sparse matrix computations [J]. International Journal of Parallel Programming, 2021, 49(5): 732 – 744.

[14] DAKKAK A, LI C, XIONG J J, et al. Accelerating reduction and scan using tensor core units[C]//Proceedings of the ACM International Conference on Supercomputing, 2019: 46 – 57.

[15] SONG L H, ZHUO Y W, QIAN X H, et al. GraphR: accelerating graph processing using ReRAM [C]//Proceedings of IEEE International Symposium on High Performance Computer Architecture, 2018: 531 – 543.

[16] IPEK E. Memristive accelerators for dense and sparse linear algebra: from machine learning to high-performance scientific computing[J]. IEEE Micro, 2019, 39(1): 58 – 61.

[17] CHI P, LI S C, XU C, et al. PRIME: a novel processing-in-memory architecture for neural network computation in ReRAM-based main memory [C]//Proceedings of ACM/IEEE 43rd Annual International Symposium on Computer Architecture, 2016: 27 – 39.

[18] SONG L H, QIAN X H, LI H, et al. PipeLayer: a pipelined ReRAM-based accelerator for deep learning[C]//Proceedings of IEEE International Symposium on High Performance Computer Architecture, 2017: 541 – 552.

[19] HE Y T, WANG Y, ZHAO X D, et al. Towards state-aware computation in ReRAM neural networks[C]//Proceedings of 57th ACM/IEEE Design Automation Conference, 2020: 1 – 6.

[20] LOU Q, WEN W J, JIANG L. 3DICT: a reliable and QoS capable mobile process-in-memory architecture for lookup-based CNNs in 3D XPoint ReRAMs[C]//Proceedings of IEEE/ACM International Conference on Computer-Aided Design, 2018: 1 – 8.

[21] DAI G H, HUANG T H, CHI Y Z, et al. GraphH: a processing-in-memory architecture for large-scale graph processing[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2019, 38(4): 640 – 653.

[22] HUANG Y, ZHENG L, LIAO X F, et al. RAGra: leveraging monolithic 3D ReRAM for massively-parallel graph processing[C]//Proceedings of Design, Automation & Test in Europe Conference & Exhibition, 2019: 1273 – 1276.

[23] NAI L F, HADIDI R, SIM J, et al. GraphPIM: enabling instruction-level PIM offloading in graph computing frameworks [C]// Proceedings of IEEE International Symposium on High Performance Computer Architecture, 2017: 457 – 468.

[24] QIN Y F, BAO H, WANG F, et al. Recent progress on memristive convolutional neural networks for edge intelligence[J]. Advanced Intelligent Systems, 2020, 2(11): 202000114.

[25] BELL N, HOEROCK J. Thrust: a productivity-oriented library for CUDA[M]//HWU W M W. GPU Computing Gems Jade Edition. San Francisco: Morgan Kaufmann Publishers, 2011: 359 – 371.