https://crad.ict.ac.cn
62(1): 75−89, 2025

混洗 SRAM: SRAM 中的并行按位数据混洗

张敦博[1] 曾灵灵[1] 王若曦[1] 王耀华[1] 沈 立[1,2]
[1]（国防科技大学计算机学院 长沙 410073）
[2]（先进微处理器芯片与系统重点实验室（国防科技大学） 长沙 410073）
（zhangdunbo14@nudt.edu.cn）

Shuffle-SRAM: In-SRAM Parallel Bitwise Data Shuffle

Zhang Dunbo[1], Zeng Lingling[1], Wang Ruoxi[1], Wang Yaohua[1], and Shen Li[1,2]
[1]（*College of Computer Science and Technology*, *National University of Defense Technology*, *Changsha* 410073）
[2]（*Key Laboratory of Advanced Microprocessors Chips and Systems* (*National University of Defense Technology*), *Changsha* 410073）

Abstract While vector processing unit is widely employed in processors for neural networks, signal processing, and high performance computing, it suffers from expensive shuffle operations dedicated to data alignment. Traditionally, processors handle shuffle operations with its data shuffle unit. However, data shuffle unit will introduce expensive overhead of data movement and only can shuffle data in serial. In fact, shuffle operations only change the layout of data and ideally should be done entirely within memory. Nowadays, SRAM is no longer just a storage component, but also as a computing unit. To this end, we propose Shuffle-SRAM in this paper, and Shuffle-SRAM can shuffle multiple data elements simultaneously bit by bit within an SRAM bank. The key idea is to exploit the bit-line wise data movement ability in SRAM so as to shuffle multiple data in parallel, where all the bits of different data elements on the same bit-line of SRAM can be shuffled simultaneously, achieving a high level of parallelism. Through suitable data layout preparation and the vector shuffle extension instructions, Shuffle-SRAM efficiently supports a wide range of commonly used shuffle operations efficiently. Our evaluation results show that Shuffle-SRAM can achieve a performance gain of 28 times for commonly used shuffle operations and 3.18 times for real world applications including FFT, AlexNet, and VggNet. The SRAM area overhead only increases by 4.4%.
Key words vector SIMD architecture; SRAM; shuffle operations; vector memory; processing in memory

摘 要 向量处理单元（vector processing unit, VPU）已被广泛应用于神经网络、信号处理和高性能计算等处理器设计中，但其总体性能仍受限于专门用于对齐数据的混洗操作. 传统上，处理器使用其数据混洗单元来处理混洗操作. 然而，使用数据混洗单元来处理混洗指令将带来昂贵的数据移动开销，并且数据混洗单元只能串行混洗数据. 事实上，混洗操作只会改变数据的布局，理想情况下混洗操作应在内存中完成. 随着存内计算技术的发展，SRAM 不仅可以作为存储部件，同时还能作为计算单元. 为了实现存内混洗，提出了混洗 SRAM，它可以在 SRAM 体中逐位地并行混洗多个向量. 混洗 SRAM 的关键思想是利用 SRAM 体中位线的数据移动能力来改变数据的布局. 这样 SRAM 体中位于同一位线上不同数据的相同位可以同时被移动，从而使混洗操作拥有高度的并行性. 通过适当的数据布局和向量混洗扩展指令的支持，混洗 SRAM 可以高效地处理常用的混洗操作. 评测结果表明，对于常用的混洗操作，混洗 SRAM 可以实现平均 28 倍的性能增益，对于 FFT, AlexNet, VggNet 等实际的应用，可以实现平均 3.18 倍的性能增益. 混洗

收稿日期：2023-03-10；**修回日期**：2024-01-15
基金项目：国家自然科学基金项目（61972407）
This work was supported by the National Natural Science Foundation of China (61972407).
通信作者：沈立 (lishen@nudt.edu.cn)

SRAM 相较于传统 SRAM 的面积开销仅增加了 4.4%.

关键词 向量单指令多数据体系结构;静态随机访问存储器;混洗操作;向量内存;存内计算

中图法分类号 TP333

DOI: 10.7544/issn1000-1239.202330151 **CSTR:** 32373.14.issn1000-1239.202330151

计算密集型应用的发展使得向量处理单元(vector processing unit, VPU)[1-5]被广泛地应用在当前的高性能处理器设计中,几乎所有处理器中都使用了 VPU 并提供了向量扩展指令. VPU 通过并行处理多个数据通道来高效地开发数据级并行性(data level parallelism, DLP),例如 Intel Xeon, ARM V8, Brainwave[6], Imagine[7], NX27V[8], DaVinci[9], VT[10], MAVEN[11]等高性能通用处理器或专用加速器. SODA[12], FT-Matrix[13], AnySP[14]等数字信号处理器也使用了 VPU. 近年来,神经网络得到了快速的发展,其高度并行的计算特性与向量处理单元的结构完美匹配,使得带有向量功能单元的处理器不断扩展其向量宽度,即提供更强的数据级并行开发能力(如 DaVinci 结构[9]),以更好地支持神经网络中的密集计算.

尽管 VPU 已被广泛应用于各类芯片的设计中,但其总体性能仍受到数据混洗操作的限制. 数据混洗操作是为重新排列向量元素而设计的,其目的是将 VPU 中的计算通道与向量寄存器文件中的数据一一对齐. 这样之后,处理器才能有效地并行处理多个数据. 当读取到的数据与计算通道不对齐时,处理器就必须在计算之前执行额外的数据混洗操作. 神经网络中的卷积层占据了其整体性能开销的 90% 以上[15-16],为了在 VPU 或类似的结构上高效地实现卷积, Chellapilla 等人[17]提出了使用图像重排算法 Img2Col(image to column)搭配通用矩阵乘法(general matrix multiplication, GEMM)算法的优化算法. Img2Col 的关键思想是展开图像并复制图像中部分像素点,重新排列图像像素和内核参数,从而生成对 VPU 友好的矩阵乘法数据布局,这样只需要再进行 GEMM 运算就可以完成卷积. 虽然 Img2Col 搭配 GEMM 可以显著提高卷积的性能,但 Img2Col 占用了大部分的执行时间,导致计算单元的计算效率较低.

图 1 为 Img2Col 和 GEMM 在 1 个卷积层中执行时间的占比. x 轴表示输入图像的大小, y 轴表示执行时间的占比. 在对不同大小数据集的测试中, Img2Col 的执行时间最多是 GEMM 的 28.3 倍,其占总执行时间的比例超过了 85.6%,最多达到 98.9%. 数据混洗操作已成为对 VPU 性能制约最严重的瓶颈之一[18-19].

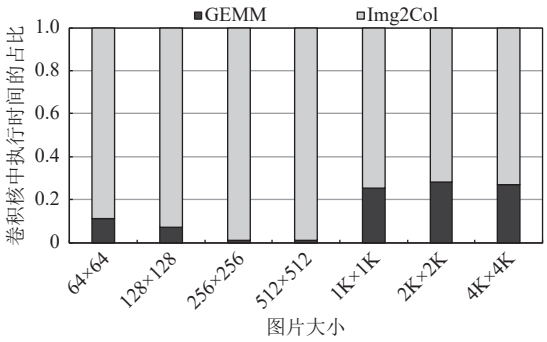


Fig. 1 Execution time proportion of Img2Col and GEMM in the convolutional layer

图 1 Img2Col 和 GEMM 在卷积层中执行时间的占比

为了在处理器中实现数据混洗操作,现有处理器都使用了定制的混洗单元[20]. 定制混洗单元通常采用交叉开关结构,以满足应用中混洗操作多样性的需求. 例如, FT-Matrix 和 DaVinci 都使用了基于交叉开关的硬件混洗单元. 然而,这种结构的混洗单元硬件开销较大,且可伸缩性较差. 随着未来向量宽度的不断增加,定制混洗单元所面临的问题将变得更加严重[21].

与加、减、乘累加等向量算数运算不同,数据混洗操作只会改变数据的布局,而不进行任何运算. 在理想的情况下数据混洗操作应该在内存中完成,从而有效地减少内存访问次数. 而现在的高性能处理器通常使用静态随机访问存储器(static random access memory, SRAM)来保存向量数据,并将其直接与向量寄存器连通,以便为 VPU 提供高带宽低延迟的访问. 这种存储器叫做向量内存, 1.1 节将对其详细介绍. 此外,近年来基于 SRAM 的存内计算技术也得到了快速发展,现有的技术已经能够在 SRAM 中实现复制、逻辑、算数等多种位运算. 基于上述观察,本文提出了混洗 SRAM,一种 In-SRAM 的混洗架构. 其关键思想是利用 SRAM 中的位级数据移动能力作为数据混洗操作的主要数据传输引擎. 通过适当的数据预布局和控制逻辑,混洗 SRAM 可以高效地实现各种常用的混洗算法,包括矩阵转置、蝴蝶混洗、Img2Col 等.

我们在 FT-2000 处理器[22]上对混洗 SRAM 进行了测试,混洗 SRAM 能够带来平均 3.18 倍的性能提

升，同时降低了 12.3% 的能耗.

本文的主要贡献有 3 个方面：

1）利用 SRAM 的存内计算体系结构中的位线数据移动能力作为数据混洗的主要引擎，提出了混洗 SRAM 来实现存内数据混洗操作.

2）设计了数据预布局映射算法和向量扩展指令，使混洗 SRAM 能够支持广泛的混洗操作.

3）对混洗 SRAM 进行了性能评估. 结果表明，对于常用的混洗操作，混洗 SRAM 可以实现 28 倍的性能增益. 使用混洗 SRAM，VPU 的整体性能平均可提高 3.18 倍，功耗降低 12.3%，而混洗 SRAM 仅会给芯片设计带来 4.4% 的额外面积开销.

1 相关背景

1.1 向量内存

向量内存是一种基于 SRAM 实现的高速缓存，已被不少高性能处理器[9,13,23-24]所使用. 向量内存中的数据被直接送往向量寄存器文件，因而可以为 VPU 提供高带宽低延迟的数据访问. 与高速缓存（cache）不同，向量内存由程序员直接管理，以充分发挥其潜力.

在向量内存中，SRAM 被组织成多体的形式，每个单指令多数据（single instruction multiple data, SIMD）通道只能访问其私有存储体，这被称为“混洗限制”. 如果一个存储体中的数据需要在另一个 SIMD 进行通道内计算，就需要引入额外的通道间通信操作，也就是所谓的混洗操作. 为了打破混洗限制，就需要在数据被取到向量寄存器文件后，使用混洗指令来使向量寄存器文件中的数据与 SIMD 计算通道对齐. 在 EEMBC 基准测试[25]中，混洗指令平均占据了总指令数的 23%. 因为在关键循环中总是需要数据混洗指令，而循环是并行应用程序的主要组成部分，占用了大部分的处理器时间. GPU 中提供了 Gather 与 Scatter 指令来共享 SIMD 计算通道之间的内存空间，VPU 则采用混洗单元（shuffle unit, SU）来完成 SIMD 计算通道之间的数据交换.

目前混洗单元主要有 2 种实现方法：一种方法是定制的交叉开关[19]，它的实现开销较小但不能支持具有不同混洗操作的应用. 另一种方法是使用 SRAM 体来替换交叉开关[14]处的触发器，但这种方法的空间开销很大. 后来的一些工作[9,22]也采用了类似的思路，这些工作要么具有更少的内存开销，要么支持更灵活的混洗模式. 然而，这些方案都需要先将数据从内存传输到处理器中，并通过显式的混洗指令来处理，严重限制了处理器的性能. 混洗 SRAM 作为一种新颖的方案，可以将混洗操作交由内存来处理，为处理器带来更高的性能和更低的能耗. 相关研究[26-28]探索了在 SRAM 中实现混洗操作的可能性.

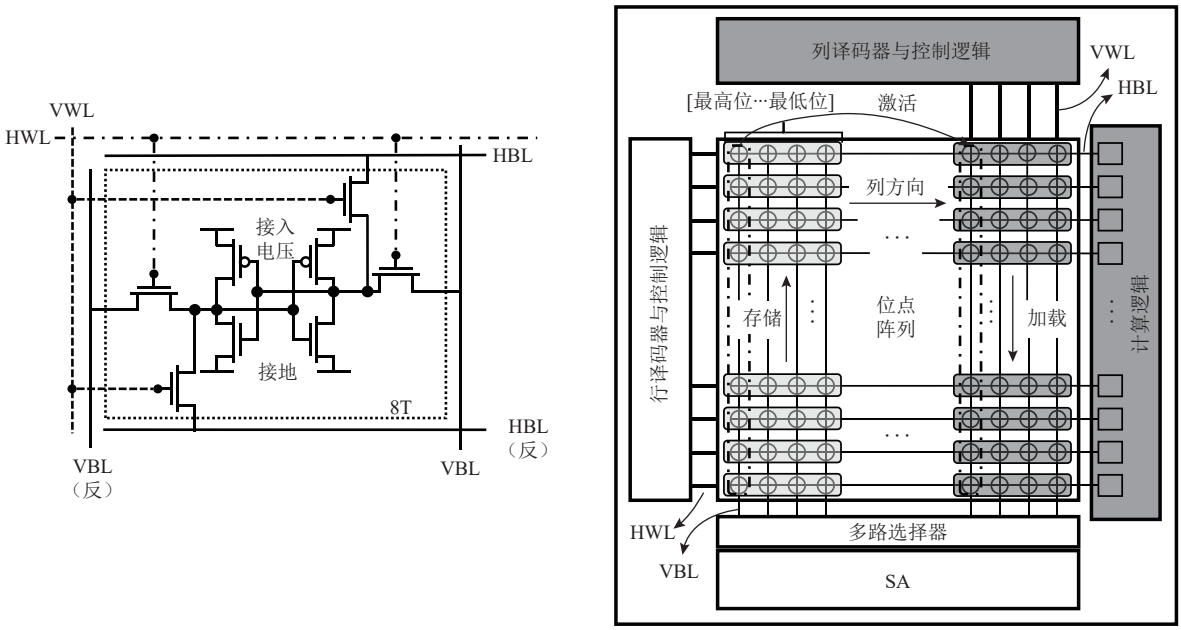
1.2 SRAM 存内计算技术

为了在不影响访存效率的情况下在 SRAM 中引入计算，需要使用 8 个晶体管来构建 SRAM 的位点（bitcell）. 如图 2（a）所示，该位点由 1 对交叉耦合的逆变器和 4 个接入晶体管组成. 交叉耦合的逆变器有 2 个稳定状态，因此可以存储 1b 的数据. 2 个横向的接入晶体管用于数据读写，由水平字线（horizontal word line, HWL）控制. 2 个纵向的接入晶体管用于计算，由垂直字线（vertical word line, VWL）控制. 图 2（b）展示了经典的 8T SRAM 存储体结构. 该存储体由 1 个位点阵列（bitcell array）组成，通常使用水平位线（horizontal bit line, HBL）和垂直位线（vertical bit line, VBL）来加载或存储数据. 在图 2（b）中，存储体内已经预加载了 2 组向量，左侧一组向量用浅色表示，右侧一组用深色表示. 同一向量的数据被放置在存储体不同的行中，而数据的不同位被分散到不同的列中，从最高位到最低位依次排列.

图 3 展示了现有的 In-SRAM 的复制和逻辑操作. 执行复制操作时，首先要激活源 VWL，将列向数据传入到灵敏放大器（sense amplifier, SA）中. 然后，激活写使能信号，将 SA 放大的结果回传到 HBLs 中. 最后，激活目标 VWL，将数据写入. SRAM 中的位线（bit line, BL）由 2 条线组成，这 2 条线保存着相反的数据. 为了区分它们，一条成为正位线，另一条成为反位线. 执行逻辑操作，首先需要同时激活 2 个源 VWL. 经过 SA 放大后，正位线中会产生与（and）的结果，反位线中则会产生或非（nor）的结果. 之后，激活写回使能信号，将逻辑运算的结果回传到 HBLs 中. 最后，激活目标 VWL 以存储结果. 此外，通过在 SA 外部添加更多的逻辑电路，再加上位运算逻辑，就可以支持更实用的复杂计算操作. 本文基于已有的复制和逻辑操作，实现了一个更高效且节能的位交换引擎.

2 In-SRAM 并行按位混洗架构

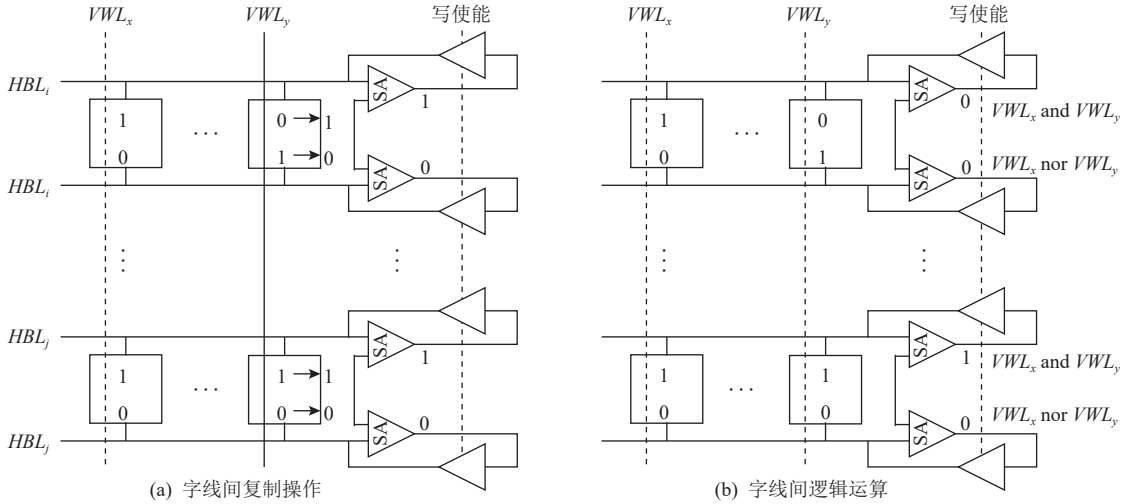
有 2 个观察结果启发了我们的研究. 首先，任何混洗操作都可以表示为 2 个或多个 VWL 之间的交换操作. 其次，对于大多数混洗操作，它们的模式是由应用或算法决定的，通常在一个循环中保持不变，例如对内存访问[29]的重新对齐. 因此，应用程序中往



(a) 8T SRAM位点结构　　(b) 8T SRAM存储体结构

Fig. 2　Structure of 8T SRAM

图 2　8T SRAM 结构



(a) 字线间复制操作　　(b) 字线间逻辑运算

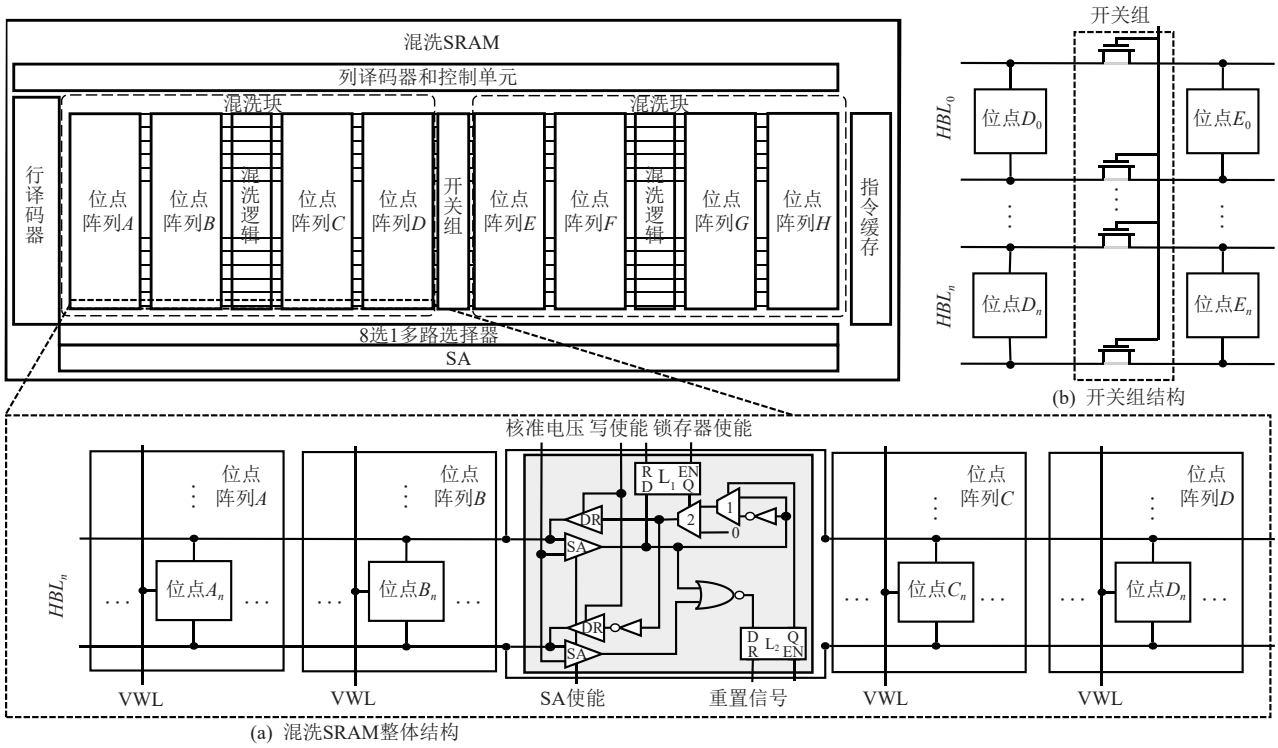
Fig. 3　Copy and logic operations of In-SRAM

图 3　In-SRAM 复制与逻辑操作

往具有足够多且模式相同的混洗操作，只要将向量在 SRAM 体中整齐地排列，通过列向的位交换操作，就可以实现高并行性的混洗操作.

为了在 SRAM 中实现并行的混洗操作，我们重新设计了 SRAM 存储体. 图 4 显示了 In-SRAM 并行按位数据混洗的整体架构. 与传统的 SRAM 存储体相比，1 个混洗 SRAM 融合了 8 个位点阵列，这些阵列共享 HWLs、HBLs、解码器、多路选择器(multiplexer, MUX)和 SA 等模块. 混洗逻辑(shuffle logic)单元是由多个数据交换单元(图 4(a)中的浅灰色模块)组成的，每个 HBL 连接了 1 个数据交换单元.

混洗 SRAM 对传统的 SRAM 进行了 2 处修改，并添加了 2 个新组件. 第 1 处修改是连通了不同位点阵列的水平位线，以支持跨位点阵列的数据传输. 第 2 处则是设计了混洗逻辑单元来支持混洗 SRAM 中的复制和信号位设置指令. 新增的 2 个组件是指令缓存和开关组. 指令缓存用于缓存来自处理器的二进制指令(2.2 节有详细介绍). 开关组由分布在不同 HBLs 上的晶体管组成. 当开关打开时，混洗逻辑单元可以交换 2 个混洗块之间的数据；当开关关闭时，2 个混洗块可以通过各自的混洗逻辑单元独立地进行数据交换.



(a) 混洗SRAM整体结构　(b) 开关组结构

Fig. 4　Shuffle-SRAM architecture overview

图 4　混洗 SRAM 架构总览

随着向量数据中元素个数的增加，位串行混洗操作的延迟也会增加. 为了减少在混洗 SRAM 中按位混洗的延迟，在实际的设计中还可以通过增加混洗逻辑单元和开关组的数量，将混洗 SRAM 分为更多的混洗块（最多 8 个，混洗块的数量需小于等于位点阵列的数量）. 这样，当开关单元关闭时，所有的混洗块都可以独立地交换数据. 图 4 展示了具有 2 个混洗逻辑单元的混洗 SRAM 的结构，这个混洗 SRAM 被分为了 2 个混洗块. 通过 2 个混洗块，混洗 SRAM 可以实现最高的性能面积比. 相关评测结果见 4.5 节.

2.1　数据交换单元

图 4(a)显示了混洗 SRAM 中 1 个混洗块中 1 行的详细视图. 混洗 SRAM 和传统 SRAM 之间最大的区别在于 HBL 连通了所有的位点阵列. 图 4(a)还展示了数据交换单元的详细设计，每条 HBL 都连接着数据交换单元. 混洗 SRAM 支持 4 条机器指令. 如图 5 所示，2 条用于数据传输（读和写），另外 2 条用于数据混洗（传输和信号位设置）. 混洗 SRAM 采用 32 b 二进制来编码这些操作. [31:30] 两位作为操作码. 对于读/写机器指令，[29:19]，[18:11]，[10:0] 三个字段分别代表混洗 SRAM 中的行地址、列地址以及数据地址. 对于传输和信号位设置机器指令，[29:19] 和 [18:8] 两个字段代表了 2 个列地址. [7:4] 字段代表了数据交换模块的使能信号，我们将混洗 SRAM 的 HBL 分为了 4 组，当混洗 SRAM 中需要处理不同的混洗操作时，就可以将数据放在不同的 HBL 组中，进行哪种混洗操作就将哪一组数据混洗单元的使能位置 1，这样就可以在同一个混洗 SRAM 体中执行不同模式的混洗操作. 传输操作中的字段 [3:0] 代表了锁存器（latch，L）L_1 和锁存器 L_2 的复位信号. 信号位设置指令中的字段 [3:0] 代表了 L_1 和 L_2 的启用信号（0 启用 L_1；1 启用 L_2）. 读/写指令在 VBL 中执行，通过 HWL

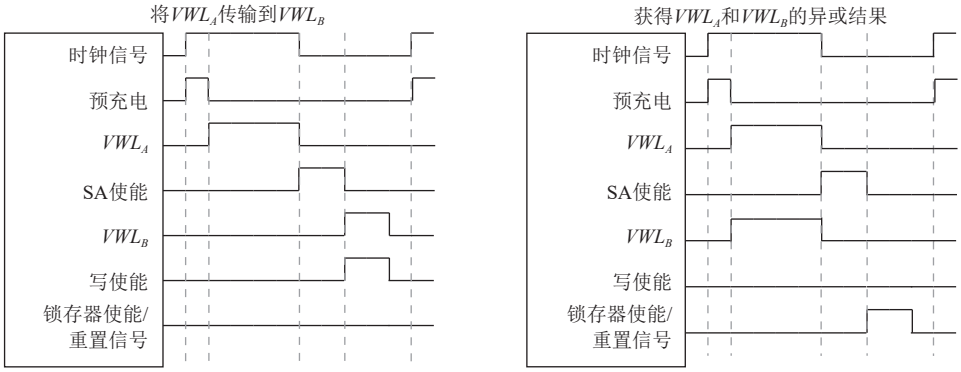
名称	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
读	操作码		行地址											列地址								数据地址										
写	操作码		行地址											列地址								数据地址										
传输	操作码		目的列字线地址											源列字线地址														块使能		重置信号		
信号位设置	操作码		源列字线地址1											源列字线地址2														块使能		锁存器使能		

Fig. 5　Machine instructions of Shuffle-SRAM

图 5　混洗 SRAM 的机器指令

控制. 传输和信号位设置指令, 在 HBL 中执行, 通过 VWL 控制, 由列数据交换单元完成.

图 6 展示了传输和信号位设置机器指令的控制信号时序图. 对于控制指令, 每位需要通过 2 个 2 选 1 的 *MUX*. MUX_1 由 L_1 控制, MUX_2 由 L_2 控制. 如果锁存器接收到控制信号“1”时, 就会输出上方的输入; 收到控制信号“0”时, 则会输出下方的输入. 在经过 *MUX* 之后, 位就会被写回目标位点. 在执行信号位设置指令时, 没有位需要被写回. L_1 存储 2 位的与(and)结果, L_2 存储 2 位的异或(xor)结果. 当某个 VWL 被激活时, 所有垂直方向同列的位点都将被激活, 所有的 HBL 可以并行地执行相同的机器指令.



(a) 传输指令 (b) 信号位设置指令

Fig. 6 Control signal timing diagrams of machine instructions

图 6 机器指令控制信号时序图

图 6(a)展示传输机器指令的控制信号. 首先, 混洗 SRAM 指令译码器接收到传输指令. 传输指令中包含 VWL_A 和 VWL_B 的地址. 在对第 *A* 列所有的 VBL 进行预充电后, 混洗 SRAM 就会先激活 VWL_A 来从位点中输出位数据. 接下来, 位数据经过 SA 放大后, 又会通过 2 个选择器, 再通过写回开关, 位数据又被传输到了原本的位线上. 最后激活 VWL_B, 在这列的位点中写入来自 VWL_A 的位数据. 为了防止同时从 2 个位点同时读取而导致潜在的干扰问题, 我们为 VWL_A 和 VWL_B 的驱动器分别设置了单独的电源电压, 以便在必要时降低字线电压. 我们还使用了伪差分 SA 来代替传统的 SA, 允许在更小的位线电压摆动时更快地感知放大结果.

2.2 向量混洗扩展指令

通过复制和信号位设置指令的灵活组合, 混洗 SRAM 目前可以支持 5 条向量混洗扩展指令: 复制(copy)、数据交换(switch)、条件复制(if-copy)、取反(invert)和绝对值(absolute), 如表 1 所示.

图 7 展示了 1 b 数据交换指令和条件复制指令的控制信号时序图. 为了实现 2 列数据的交换, 我们使用了 1 个信号位设置指令和 2 个复制指令的组合来实现最基本的 1 b 数据交换指令. 1 b 数据交换将经过以下过程. 首先, 同时激活需要交换的 2 列(VWL_A 和 VWL_B), 此时位经过 SA 放大后, 在 HBL 上就会产生“*A* and *B*”的结果, 在水平反位线上产生“*A* nor *B*”的结果. 接下来, 将 HBL 和水平反位线上产生的结果输入 NOR 门就会产生“*A* xor *B*”的结果. 之后, 打开 L_2 的写使能信号, 使 L_2 能够保存“*A* xor *B*”的结果. 接下来, 再激活 VWL_A, 将输出写回 VWL_A. 激活 VWL_B, 将输出写回到 VWL_B. 根据 L_2 中保存的结果, 如果 VWL_A 和 VWL_B 的位不同, 那么“*A* xor *B*”的结果就为“1”, MUX_1 就会输出取反的结果, 这样 VWL_A 和 VWL_B 分别取反后就相当于交换了 2 列中的位数据. 最后, 重置 L_2, 这样 1 个多位数据的交换可以通过逐位的 1 b 交换操作来实现. 我们使用 1 个 4 b 数据交换操作的例子来说明. 假设有 2 个 4 b 数据“0110”和“0101”, 按位异或的结果是“0011”. 根据异或的结果, 这 2 个数据的最后 2 位将被取反. 这 2 个数据最终将变为“0101”和“0110”, 取反的结果就是这 2 个数据被交换了位置.

Table 1 Vector Shuffle Extended Instructions Supported by Shuffle-SRAM

表 1 混洗 SRAM 支持的向量混洗扩展指令

操作码	源 *A*	源 *B*	目的 *C*	数据位宽	描述
copy	√		√	*N*	*C*=*A*
switch	√	√		*N*	*A*=*B*, *B*=*A*
if-copy	√		√	*N*	if (*A*>0)*C*=*A*
invert	√			*N*	*A*=*A*
absolute	√		√	*N*	*C*=|*A*|

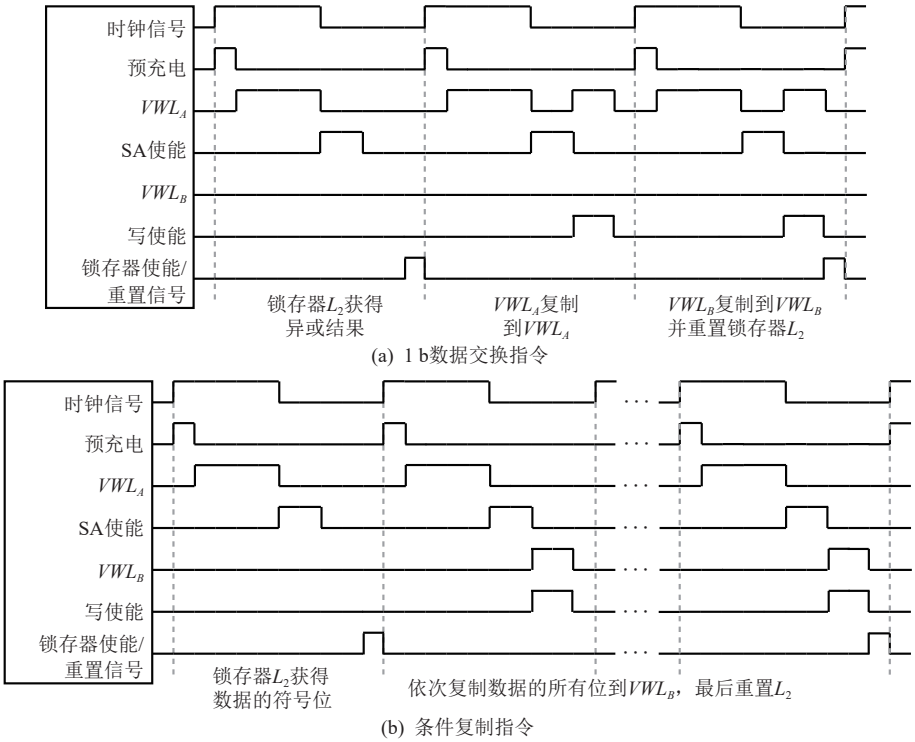


Fig. 7 Control signal timing diagrams of vector shuffle extended instructions

图 7 向量混洗扩展指令的控制信号时序图

条件复制、取反和绝对值这 3 条向量扩展指令可以通过组合 1 个信号位设置指令和多个复制指令来实现(复制指令的具体数量由数据长度决定). 以条件复制扩展指令为例, 使用信号位设置指令在 L_1 中存储条件位. 通过 *MUX*, 如果 L_1 保存的信号为“1”, 则该位将被置为“0”. L_1 需要在最后一条复制指令结束后被复位. 而复制向量扩展指令是通过组合多个复制机器指令来实现的.

为了提高混洗 SRAM 的可编程性, 我们又向原始指令集中添加了 2 个额外的指令, 分别是 copy to SRAM 和 start execution. copy to SRAM 指令用于将二进制指令复制到指令缓冲区, 被复制的指令将在混洗 SRAM 中执行. 当混洗 SRAM 接收到 start execution 指令时, 它将开始执行存储在指令缓冲区中的二进制代码. 在所有二进制指令被执行后, 指令缓冲区将被清空.

2.3 混洗 SRAM 的优势

In-SRAM 并行按位混洗架构具有提升数据并行性的潜力, 同时也可以显著地减少处理器需要执行的指令数和数据移动的开销. 图 8 通过比较数据混洗单元和混洗 SRAM 的工作流程, 来展示混洗 SRAM 的优势. 如图 8(a)所示, 数据混洗单元采用位并行的模式来执行混洗操作, 混洗操作只有在向量被加载到向量寄存器后才能执行. 使用数据混洗单元, 1 次混洗操作最多只能重排 2 个向量之间的元素. 如图 8(b)所示, 混洗 SRAM 使用位串行模式来执行混洗操作. 通过 12 条指令就可以完成图中所有向量的混洗, 且所有的混洗操作都是在混洗 SRAM 中完成的. 向量寄存器可以直接加载被混洗后的向量. 1 个混洗 SRAM 通常都具有成百上千个 HWL, 这些 HWL 都可以并行地混洗数据, 从而在 SRAM 中开发了数据级并行性. 此外, In-SRAM 并行按位混洗架构通过将混洗操作移动到内存中, 不仅能减少了芯片上数据的混洗开销, 还能减少数据在片上和内存之间的移动开销.

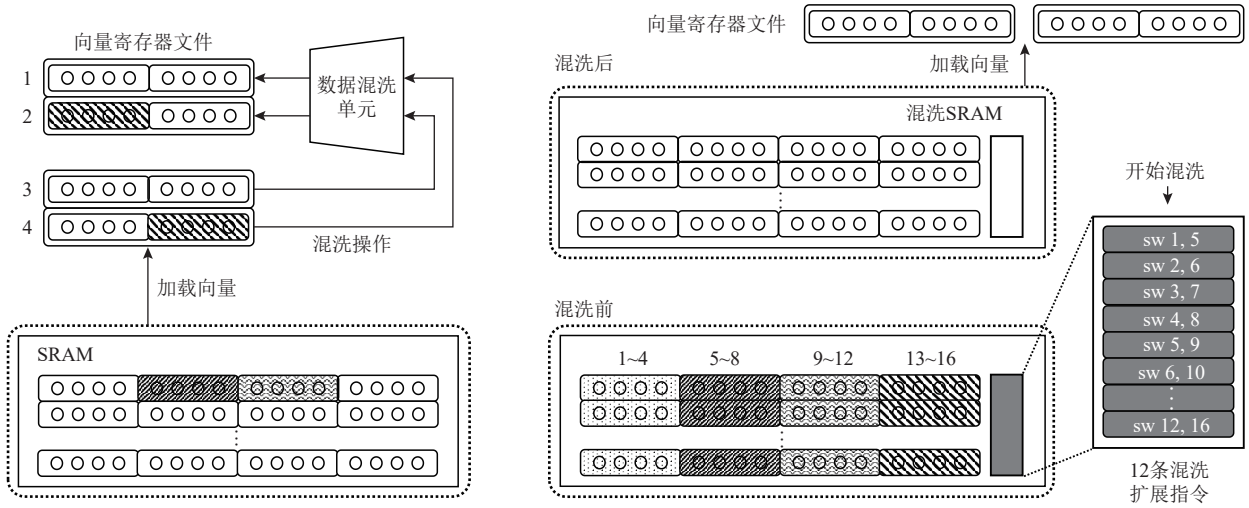
2.4 讨 论

1)混洗逻辑单元数量的选择

根据前面的介绍, 虽然数据交换引擎可以加速按位混洗操作. 但在位串行混洗模式下, 单次混洗操作的延迟仍然较大、无法忽略. 随着向量长度与数据长度的增加, 并行混洗操作的延迟也会增加. 增加混洗逻辑和开关单元的数量可以进一步提升混洗 SRAM 的并行性. 4.5 节将详细讨论如何选择混洗 SRAM 参数, 以获得最佳的性能面积比.

2)多种混洗模式(MSM)的处理

在一些特殊的循环中, 同时会有多组数据需要



(a) 使用数据混洗单元进行混洗 (b) 使用混洗SRAM进行混洗

Fig. 8 Compare data shuffle unit with shuffle-SRAM

图 8 数据混洗单元对比混洗 SRAM

同时进行混洗,且不同组的数据具有不同的混洗模式. 我们将这种循环称为具有“多混洗模式”的循环. 图 4 只显示了 1 个混洗 SRAM 体的结构. 在向量内存的设计中,混洗 SRAM 体是构建向量内存的基本单元. 整个向量内存由多个混洗 SRAM 体组成. 这些混洗 SRAM 体是串联的,它们的地址是连续的. 想要使用混洗 SRAM 来优化具有多种混洗模式的循环,程序员需要直接控制数据的传输,以将不同的数组分布到不同的混洗 SRAM 体中. 通过给每个混洗 SRAM 体发送不同的指令,不同的数据组就可以并行地被混洗.

图 9 展示了混洗 SRAM 构成的向量内存如何处理具有多种混洗模式的循环.

```
for (i,j, k=0; i<M, j<N, k<O; i+=3, j+=2, k+=4)
{
    C[i]=A[j]+B[k];
}
```

Fig. 9 Code example of multiple shuffle modes

图 9 多种混洗模式代码示例

在这个循环中,数组 A, B, C 都有自己的混洗模式. 在计算之前,程序员应将数组数据复制到不同的混洗 SRAM 体中(假设数组 A, B, C 分别保存在混洗 SRAM 体 1, 2, 3 中). 然后,混洗 SRAM 体 1 聚合数组 A 中步长为 1 的元素,混洗 SRAM 体 2 聚合数组 B 中步长为 3 的元素,混洗 SRAM 体 3 聚合数组 C 中步长为 2 的元素. 在处理过程中,向量处理器只需要执行“取数、计算、写回”的指令循环. 在所有计算结束后,混洗 SRAM 体 3 再以 3 为间隔离散元素,就可以完成整个循环. 我们在实验中测试了这类循环的性能,结果见 4.2 节.

3 数据重组的实现

3.1 数据映射

在混洗 SRAM 执行操作之前,向量数据应该按一定规律整齐地放置在混洗 SRAM 中. 对于 1 维数组,只需要将向量按混洗 SRAM 的地址顺序对齐放置在混洗 SRAM 的不同行中. 对于 2 维数组,在将向量存入混洗 SRAM 之前,需要根据混洗 SRAM 的行大小将向量分块. 直接存储器存取(DMA)单元可以支持不同存储器之间的高速传输,因此实现数据在混洗 SRAM 布局的主要挑战是如何为每个向量生成对应的地址.

混洗 SRAM 中数据映射的本质是要找出具有相同混洗模式的混洗操作,并将相应的向量对齐地放在混洗 SRAM 的每一行上. 这样,所有的向量都就可以通过同一条指令同时被混洗,这是利用混洗 SRAM 高并行性的关键.

3.2 向量指令集中的混洗操作

向量指令集中的混洗操作很简单,所有的混洗指令只涉及到在 1 个向量内或在多个向量间对元素进行重新排列. Gather2 是 ARM SVE 向量扩展指令集中的 1 条,用于收集元素索引模 2 为 0 的元素. 类似地,Gather3 收集元素索引模 3 为 0 的元素. 快速傅里叶变换(FFT)中的蝴蝶混洗就是主要通过 Gather2 指令实现的. 要利用混洗 SRAM 来实现这些 Gather 指

令，就需要使用式(1)来生成向量在混洗 SRAM 中的地址. 式(1)中 R_s 代表混洗 SRAM 的行大小，E_s 代表数组的元素大小，$Addr$ 代表混洗 SRAM 的地址，i 代表数组元素的索引.

$$Addr = i \times E_s / R_s. \tag{1}$$

混洗 SRAM 实现数据混洗操作需要在多个源到多个目的 VBL 之间执行位交换，从而为不同的 SIMD 通道提供所需的数据，激活哪些源 VWL 与目的 VWL 则是由混洗模式决定的. 对于传统的数据混洗单元，由于对多粒度混洗操作的需求，可伸缩性是一个关键问题. 随着 SIMD 宽度的增加，数据混洗单元的硬件开销也会不断变大. 在混洗 SRAM 中，混洗操作是逐位进行的，它的可伸缩性范围由硬件宽度决定. 因此，混洗 SRAM 的可伸缩性要远优于数据混洗单元.

在混洗 SRAM 的支持下，混洗操作在 3 个方面得到了优化. 首先，使用混洗 SRAM 可以支持任意粒度的混洗操作，这使得混洗操作能够摆脱硬件宽度的限制. 其次，混洗 SRAM 支持统一混洗模式下的并行处理，这提高了 SIMD 器件的性能和能效. 最后，混洗 SRAM 支持具有不同控制流路径的混洗操作，这扩展了混洗操作的功能，使其能够处理更多的有数据重组需求的应用.

3.3 矩阵转置

假设 R_s 代表混洗 SRAM 的行大小，E_s 代表数组的元素大小，$M \times N$ 代表矩阵大小，$Addr$ 代表混洗 SRAM 的地址. 对于矩阵转置中的数据混洗，向量地址可以用式(2)生成：

$$\begin{aligned} Addr &= \frac{i \times N}{X \times Y} + \frac{j}{Y}, \\ X &= \sqrt{R_s / E_s}, \\ Y &= \sqrt{R_s / E_s}. \end{aligned} \tag{2}$$

许多重要的数字信号处理算法(包括线性代数、音频处理和图像处理等应用)的性能在很大程度上都取决于矩阵转置的效率. 传统的矩阵转置实现方案都需要处理器的参与来完成矩阵转置：首先从内存中先加载源矩阵中的每个向量；然后使用混洗操作交换多个向量之间的数据；最后再将结果向量写回内存. 虽然矩阵转置的混洗模式很简单，但由于混洗的并行性非常差，极大地影响了矩阵转置的性能，大型矩阵因转置带来的性能延迟更为突出. 传统的矩阵转置方案不仅占用了访存单元和带宽，还影响了计算任务的处理，这是因为矩阵转置和计算任务只能串行执行. 使用混洗 SRAM，经过分块与数据映射，每个 HBL 上的矩阵块可以同时被转置，不需要处理器的参与，也没有额外的数据传输开销. 此外，混洗 SRAM 中的矩阵转置和处理器的后续计算也可以通过灵活的编程来实现流水化执行.

图 10 显示了一个矩阵转置的例子. 该矩阵大小为 6×6，$R_s = 16$，$E_s = 4$. 矩阵被分为 9 个块，每个块包含 4 个元素. 根据式(2)，每个块都可以被映射到混洗 SRAM 的每一行中. 通过 4 个交换操作，整个矩阵的转置操作就可以被完成.

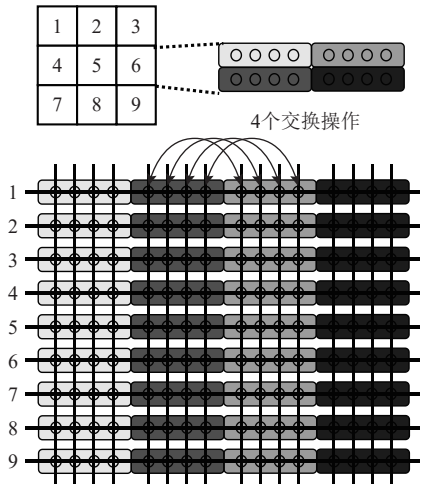


Fig. 10 Implementation of matrix transposition by using Shuffle-SRAM

图 10 使用混洗 SRAM 实现矩阵转置

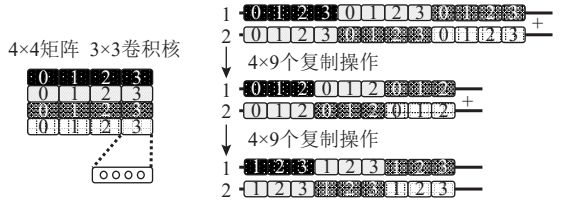
3.4 Img2Col

假设 K_s 表示卷积核(kernel)大小，S 表示步长，矩阵的大小为 $M \times N$，$Addr$ 表示为混洗 SRAM 的地址. 在 Img2Col 中，元素 $[i][j]$ 在混洗 SRAM 中的地址可以用式(3)生成. 与矩阵转置不同，Img2Col 需要根据核大小多次输入数据.

$$\begin{aligned} Addr &= \frac{(i - z + 1) \times (N + S - \sqrt{K_s})}{X^2} + \frac{j + S - \sqrt{K_s}}{Y}, \\ & z - 1 < i < M + z\sqrt{K_s}, \\ X &= 2\sqrt{K_s} - S, \\ Y &= 2\sqrt{K_s} + S \times \left(\frac{R_s}{E_s \times K_s} - 2 \right). \end{aligned} \tag{3}$$

图 11 显示了一个 4×4 大小的矩阵，它需要由一个 3×3 大小的内核进行卷积，每个元素的长度为 4 b. 根据式(3)，当“$z = 1$”时，矩阵的前 2 行输入到混洗 SRAM 的“1”行和“2”行(源矩阵第 1 行的元素被放在“1”行中，源矩阵第 2 行的元素被放在“2”行中). 由于页面大小的原因，图 11 中的“1”行和“2”行应当更长，图 11 中“1”行和“2”行被分为了 3 段. 之后的输入并不会替换之前的输入，而是紧接着之前的输

入存放. 当“$z=2$”时, 源矩阵第 2 行的元素被放在“1”行中, 源矩阵第 3 行的元素被放在“2”行中. 以此类推, 当“$z=3$”时, 源矩阵第 4 行的元素被放在“1”行中, 源矩阵第 4 行的元素被放在“2”行中. 到此为止, 数据映射过程完成. 然后, 可以通过执行 2 次 4×9 个复制操作就可以完成算法 Img2Col. 随着矩阵尺寸的增大, 使用混洗 SRAM 的性能优势也将不断提升.



(a) 数据在原矩阵中的位置 (b) Img2Col在混洗SRAM中的混洗步骤

Fig. 11 Implementation of Img2Col by using Shuffle-SRAM

图 11 使用混洗 SRAM 实现 Img2Col

3.5 ReLU

ReLU 中数据的地址也可以通过式(1)生成. ReLU 函数是人工神经网络中常见的激活函数. ReLU 激活函数只需要简单的计算: 如果输入大于 0, 则返回输入值; 如果输入为 0, 则返回值为 0. 它可以用式(4)来表示.

$$f(x)=\max(0,x). \tag{4}$$

使用混洗 SRAM 中的条件复制操作就可以容易且高效地实现 ReLU. 图 12 展示了如何对位长为 4 的数据实现 ReLU 变换, 从而来解释元素的每 1 位如何从最高位逐位执行到最低位的. 这 4 个元素占据混洗 SRAM 的 4 列. 首先使用信号位设置操作将 4 个元素的符号位输入到数据交换单元的 L_1 中. 接下来使用 4 个复制操作将这 4 列都转移到目标位置. L_1 锁存器保存的符号位是“1”, 则 *MUX* 将输出“0”. 图 12 中, 混洗 SRAM 并行完成了 4 个数据的 ReLU 变换.

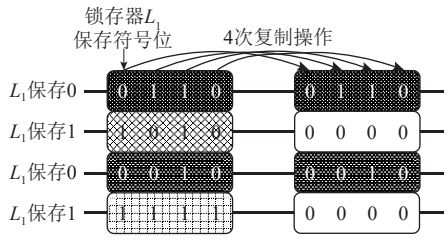


Fig. 12 Implementation of ReLU by using Shuffle-SRAM

图 12 使用混洗 SRAM 实现 ReLU

4 实验与评估

本节展示了对混洗 SRAM 的评估方法与结果. 评估以 1 GHz 的 FT-2000 处理器[22] 为基准. FT-2000 使用 ARM V8 指令集, 支持 ARM V8 的向量扩展指令, 以其作为基准具有一定的代表性.

4.1 模拟器配置

表 2 展示了 FT-2000 模拟器的一些体系结构参数. 在我们的模拟中, 设置的 SIMD 指令的延迟和吞吐量都与 FT-2000 文档中的数据相同.

Table 2 Parameters of Vector Processor

表 2 向量处理器参数

配置	描述
处理器频率/GHz	1
向量长度/b	2 048
向量单元	16 × 8
SRAM 容量/MB	8
混洗 SRAM	8 MB; 8 个位点阵列; 位点阵列大小: 2 048×256; 2 个混洗块

4.2 内核测试

表 3 展示了我们在实验中使用的 6 个数据重组内核, 这些内核经常出现在多媒体、CNN 和其他应用程序中. 这些内核都需要大量的混洗操作来实现, 包括不对齐的内存访问、数据重组和控制分支流. 它们可以用来显示 In-SRAM 并行按位混洗架构的高效性和广泛适用性. Gather2(G2)和 Gather3(G3)内核广泛应用于 FFT、线性预测编码和图像处理. 矩阵转置(matrix transpose, MT)、Img2Col 和 ReLU 被广泛应用于 GEMM 中, 它们都有相对简单且统一的数据混洗模式. 在测试中, 假设需要混洗的数据已经按照第 3 节中所介绍的方式在混洗 SRAM 中摆放好. 基准结构和混洗 SRAM 之间有 2 个主要的区别: 一是基准结构需要先将每个数据移动到一个特定的混洗单元来完成数据重组, 而混洗 SRAM 可以在不移动数据

Table 3 Performance Comparison Between Shuffle-SRAM and Baseline When Shuffling Test Kernels

表 3 混洗 SRAM 和基准在处理测试内核时的性能比较

测试内核	数据大小/MB		数据长度/b	延迟/μs		加速比
	输入	输出		基准	混洗 SRAM	
G2	2	2	32	3.67	0.061	60.1
G3	3	3	8	10.62	0.018	590.0
MT	1	1	32	2.88	0.011	361.8
Img2Col	3	27	32	14.16	0.590	24.0
ReLU	3	3	32	1.77	0.004	442.5
MSMs	6	3	32	6.23	1.380	4.5

注: 基准的时钟频率比混洗 SRAM 的快 4 倍.

的情况下完成数据混洗. 另一个区别是，模拟的混洗 SRAM 可以同时对 2 048 个向量进行混洗，而基准结构 1 次只能对 1 个向量进行混洗.

图 13 展示了在使用 1 个混洗 SRAM(512 KB)时，采用不同大小的数据集时的实验结果. 当增加输入数据的大小时，混洗 SRAM 有不同的加速比. 当数据大小为无限大时，相比于基准结构，混洗 SRAM 可以被平均加速 4.22 倍. 我们发现了一个现象，随着数据大小的增加，加速比持续减小. 导致这个现象的原因是：当数据大小较小时，所有数据都可以提前以适当的布局放置在混洗 SRAM 中. 当数据大小增加时，1 个混洗 SRAM 容量消耗完时，就需要交换混洗数据. 即在完成一批数据的混洗后，再将下一批数据放入混洗 SRAM 中，再对这一批数据进行混洗，依此类推. 正是数据移动开销的占比不断增大才导致了加速比的持续减小.

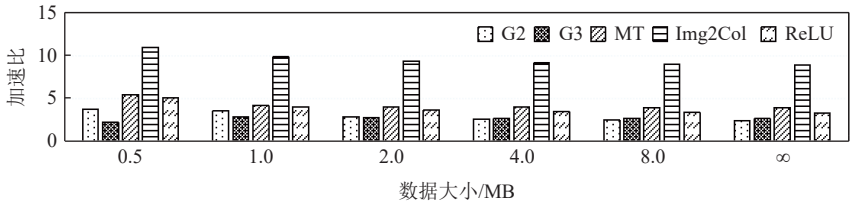


Fig. 13 Speedup of different shuffle kernels with different input data size

图 13 输入数据大小的不同时不同混洗内核的加速比

图 14 展示了基于混洗 SRAM 构建的向量内存在不同数据大小时内核的加速效果. 我们在这个测试中使用的输入数据大小为 2 MB. 如图 14 所示，随着向量内存数据大小的增加，加速比也会不断增加. 当向量数据大小为 8 MB 时，加速比为 373. 我们发现一个现象，随着混洗 SRAM 数据大小的增加，加速比在一定范围内显著增加(G2：1~4 MB；G3：2~8 MB；MT：4~8 MB；Img2Col：4~8 MB；ReLU：2~4 MB). 导致这个现象的原因是在这个范围内，所有数据都可以预先按照适当布局被放置在混洗 SRAM 中，因此消除了数据移动的开销. 对于 G2，数据大小在 1~2 MB 下的加速比增加的原因是消除了数据移动的开销，而在 2~4 MB 下的加速比增加的原因是用 2 个复制操作替换了 1 个交换操作. 对于 MT，加速比在数据大小 2~4 MB 下不会增加. 原因是当混洗 SRAM 大小为 2 MB 时，混洗 SRAM 可以保存所有数据. 但是，当混洗 SRAM 大小为 4 MB 时，复制操作不能代替交换操作，从而减少了性能收益.

由于多混洗模式内核是一个混合了计算和混洗操作的内核，因此它的加速主要来源于减少的内存访问. 此外，它的加速比不会随着不同的混洗 SRAM 大小或不同的输入数据大小而变化. 因此，我们没有在图 13 和图 14 中展示多混洗模式内核的性能收益，相关数据被放在了表 3 中.

4.3 实际应用测试

表 4 展示了我们选取的 3 个实际测试应用. 图 15 显示了运行 FFT, VggNet16, AlexNet 的总加速比. 与

Table 4 Test Applications

表 4 测试应用

应用	描述
FFT[30]	快速傅里叶变换是用计算机计算离散傅里叶变换（DFT）的高效快速计算方法的通术语，称为 FFT，由 Cooley 等人在 1967 年提出.
VggNet16[31]	VGGNet16 是一个深度卷积神经网络，由牛津大学计算机视觉小组和谷歌 DeepMind 的研究人员开发.
AlexNet[32]	AlexNet 神经网络有 6 000 万个参数和 65 万个神经元，由 5 个卷积层组成，其中一些卷积层之后是最大池化层以及 3 个全连接层，最后是 1 000 路 softmax 层.

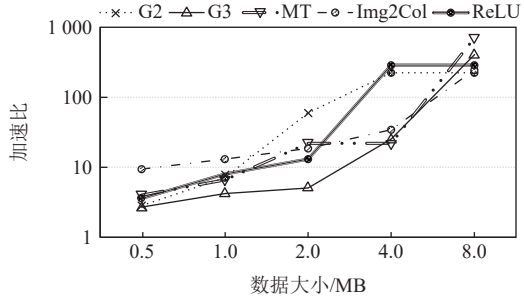


Fig. 14 Speedup with different vector memory sizes

图 14 不同向量内存大小时的加速比

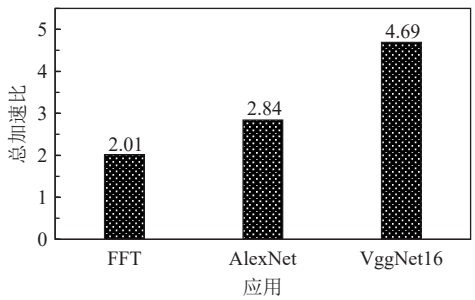


Fig. 15 Total speedup with 8 MB Shuffle-SRAM

图 15 8 MB 混洗 SRRAM 的总加速比

基准结构性能相比, FFT 加速了 2.01 倍, AlexNet 加速了 2.84 倍, VggNet16 加速了 4.69 倍. 如此显著的加速比可以归因于减少的额外访存开销以及消失的处理器中的数据混洗开销.

图 16 显示了另一个性能加速的来源. 宏流水线是一种多指令、多数据流并行算法, 它有序且易于在多处理器或超长指令字结构中实现, 适用于批量处理. 使用混洗 SRAM 后, 向量计算和数据重组可以在不同的设备中处理. 同时, 混洗 SRAM 支持并行的读/写和混洗操作. 因此, 处理器将数据进行分批, 并不需要将所有数据都混洗后再进行计算. 可以先混洗一部分就开始计算, 这样就可以使处理器与混洗 SRAM 并行工作.

FFT 性能的提高来自 3 个方面: 1)使用混洗 SRAM 执行混洗操作, 消除了处理器中蝴蝶混洗操作的延迟; 2)通过混洗 SRAM 混洗后的数据, 向量中有效元素的占比更高, 从而提高了向量功能单元的利用率; 3)通过宏流水编程技术将 FFT 的各数据点分为实部和虚部, 每个数据点都需要对另一个数据点进行蝴蝶计算, 包括复数的加、减、乘法. 为了利用 SIMD 体系结构的性能, 数据需要在计算过程中连续地进行蝴蝶混洗. 如图 16(a)所示. 值得注意的是, 当采用混洗 SRAM 时, 蝴蝶混洗的计算时间开销更低. 因为在基准结构下蝴蝶计算前必须做混洗操作, 而最后 3 级的蝴蝶混洗需要更多额外的混洗操作. 使用混洗 SRAM, 大部分延迟可以通过宏管道编程来隐藏.

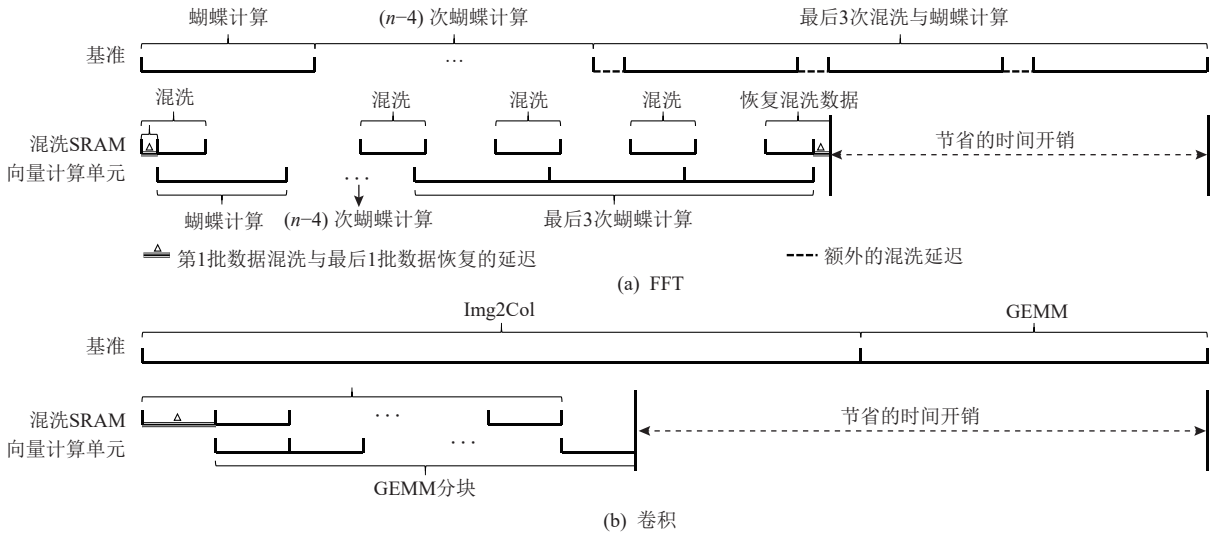


Fig. 16 Macropipelining

图 16 宏流水线

表 5 显示了 AlexNet 和 VggNet16 延迟的分解. 无论哪个网络, 卷积层几乎花费了所有的执行时间. 在基准结构下, 卷积层花费的时间占 AlexNet 总执行时间的 99.1%, 而在 ReLU 层上花费了 2.63 s, 池化层需要 1.23 s. 卷积层是最耗时的部分; 其余的部分是全连接层(full connect, FC)FC4096 和 FC1000 的时间花费, 分别占 1.57 s 和 0.19 s. 对于 VggNet16, 卷积层花费的时间占 VggNet16 总执行时间的 99.3%, 在 ReLU 层上花费 0.20 s, 池化层需要 15.35 s. 卷积层同样是最耗时的部分.

Table 5 Latency of VggNet16 and AlexNet

表 5 VggNet16 和 AlexNet 的延迟 s

测试用例	网络	卷积层	ReLU	池化层	FC4096	FC1000	总计
基准	VggNet16	615.43	2.63	1.23	1.57	0.19	621.05
	AlexNet	2 441.57	0.20	15.35	1.57	0.19	2 458.89
混洗 SRAM	VggNet16	215.63	0.05	1.23	1.57	0.19	218.67
	AlexNet	507.16	0.01	15.35	1.57	0.19	524.28

使用混洗 SRAM, 我们可以用 GEMM 来划分卷积从而并行化 Img2Col, 如图 16(b)所示, 除了初始的混洗时间外, 所有的 Img2Col 开销都可以被 GEMM 的计算延迟隐藏. 总的来说, AlexNet 的卷积比例下降了 65.0%, ReLU 下降了 98.1%; VggNet16 的卷积比例下降了 79.2%, ReLU 下降了 95.0%.

4.4 能耗与面积的影响

为了获得能耗与处理器面积, 我们模拟了基于 40 nm SOI CMOS 工艺下的位点阵列, 该阵列使用标准 8T 位点结构. 1 个 8T 位点的面积为 0.589 μm^2, 1 个数据交换单元的面积为 246.903 2 μm^2. 每个读/写指令的延迟为 3.497 ns, 每个复制/信号位设置指令的延迟

为 3.968 ns. 与传统的 8T SRAM 相比，总面积增加了 4.4%. 在能耗评估中，我们对上述参数与每个指令的能量进行相加，来确定混洗 SRAM 的总能耗，具体数据如表 6 所示.

Table 6 Energy Consumption per Machine Instruction

表 6 单条机器指令的能耗

能耗	read	write	trans	signal
Energy/nW	0.815	0.856	10.282	6.519

图 17 展示了混洗 SRAM 与具有 32 B 宽度 SIMD 单元的处理器的能耗对比. 结果表明，混洗 SRAM 可以为这些应用平均节省 1.14 倍的能耗. 降低能耗有 3 个方法：一是通过多个水平位线并行混洗提升了的数据并行性；二是减少内存和向量寄存器文件之间的数据移动；三是消除了处理器所需处理的混洗指令.

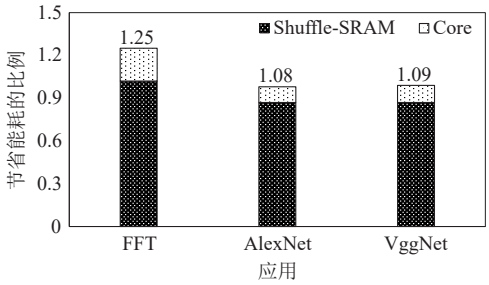


Fig. 17 Total energy benefit of Shuffle-SRAM vector memory for applications

图 17 混洗 SRAM 向量内存为测试应用带来的总能耗收益

4.5 性能面积比

图 18 显示了当混洗 SRAM 具有不同数量的混洗逻辑单元时的性能面积比. x 轴表示混洗逻辑单元的数量. 横坐标“0”处表示 8T SRAM 传统结构的性能面积比，其面积和加速比都设为 1. 随着混洗逻辑单元数量的增加，加速比和面积开销都在不断增加.

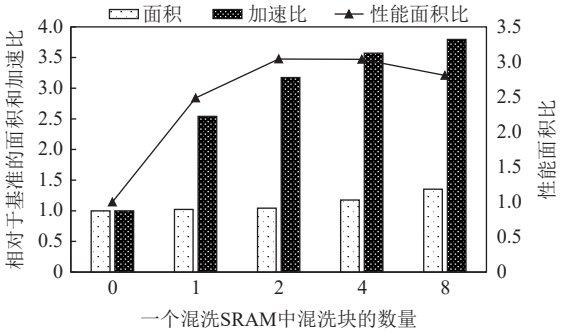


Fig. 18 Performance to area ratio under different shuffle block numbers

图 18 不同混洗块数量时处理器性能面积比

性能面积比显示了加速比与面积的比率. 当混洗 SRAM 中混洗块数量为 2 时，性能面积比最高，为 3.04. 在图 4 和表 3 中，我们评估的结果是基于具有 2 个混洗逻辑单元的混洗 SRAM 所构成的向量内存.

4.6 讨 论

In-SRAM 并行按位数据混洗架构的主要优势是将数据混洗操作从向量处理器中解耦到向量内存中，混洗操作可以在 1 个混洗 SRAM 中并行处理所有水平位线上的数据. 将数据混洗操作集成到内存中需要做到：1)用混洗 SRAM 替换传统的 SRAM，并添加相关的控制逻辑来执行特定的指令；2)将混洗 SRAM 微指令集成到现有的向量 ISA 中. 混洗 SRAM 可适配于任何 SIMD 架构. 虽然在实验中只评估了 1 个 SIMD 架构，但混洗 SRAM 是通用的，当数据混洗操作和 SIMD 算术指令可以并行处理时，SIMD 设备的性能优势将进一步凸显. 因为在消除了数据混洗开销之后，SIMD 通道可以专注于执行算术指令. 随着数据大小的增加，通过宏流水编程，使用混洗 SRAM 带来的加速比也会更高.

混洗 SRAM 对现有算法的实现带来了一定的影响. 以卷积层的实现为例. 目前所采用的方法是将整个图像转换为矩阵模式，然后调用 GEMM 高性能计算库. 使用混洗 SRAM 时，一种方法需要将图像进行更细粒度的划分，这样使用混洗 SRAM 进行数据重排的时间就可以被 GEMM 计算的时间隐藏起来. 另一种方法是设计一种新的卷积实现方案，专用于 In-SRAM 的处理模式. 之所以目前高速卷积的实现方案都不采用传统的卷积串行计算方法，是因为列方向上的数据局部性很差，且高速缓存的容量有限，难以缓存大规模的数据，从而导致处理器性能下降. 对于 3×3 卷积内核，计算每个数据点时使用的数据将跨越 3 个不同的列. 使用混洗存储器就可以提前将这些数据排列在一起，这意味着卷积的实现有进一步研究的空间，来判断传统的卷积实现方案在混洗向量内存的支持下是否能进一步地提升.

5 结束语

传统的 SRAM 仅作为中间低延迟的存储单元使用. 我们的工作直接挑战了这种传统的设计范式，并提出将数据混洗操作从向量处理器中解耦到向量内存中. 本文中，我们提出了带有 In-SRAM 并行按位混洗架构的混洗 SRAM，以减少片上数据重组的开销. 混洗 SRAM 实现了 4 个基本操作，同时我们设计了 5

条向量扩展指令来支持存内灵活的混洗编程. 我们提出的混洗 SRAM 位交换引擎和数据布局方面的方案使我们在与先进向量处理器性能相比时具有竞争力. 混洗 SRAM 的评估结果证明, 通过精心设计的硬件机制和编程策略, 可以消除大部分数据的混洗延迟, 从而实现高效的 SIMD 计算. 总的来说, In-SRAM 并行按位混洗架构的引入为开发更有效的并行算法开辟了一条新的道路, 这些算法可以用于进一步优化向量 SIMD 架构的性能.

作者贡献声明: 张敦博提出了架构思路和实验方案, 并且完成实验和撰写论文; 曾灵灵参与了架构的设计; 王若曦协助修改论文; 王耀华和沈立提出指导意见并修改论文.

参 考 文 献

[1] Chen Weihao, Li Kaixiang, Lin Weiyu, et al. A 65nm 1MB nonvolatile computing-in-memory reram macro with sub−16ns multiply-and-accumulate for binary DNN AI edge processors[C]//Proc of IEEE Int Solid-State Circuits Conf. Piscataway, NJ: IEEE, 2018: 494−496

[2] Domingos J, Neves N, Roma N, et al. Unlimited vector extension with data streaming support[C]//Proc of the 48th ACM/IEEE Annual Int Symp on Computer Architecture. Piscataway, NJ: IEEE, 2021: 209−222

[3] Malkowsky S, Prabhu H, Liu Liang, et al. A programmable 16-lane SIMD ASIP for massive MIMO[C/OL]//Proc of IEEE Int Symp on Circuits and Systems. Piscataway, NJ: IEEE, 2019[2023-01-13]. https://ieeexplore-ieee-org-s.libyc.nudt.edu.cn/stamp/stamp.jsp?tp=&arnumber=8702770

[4] Stephens N, Biles S, Boettcher M, et al. The ARM scalable vector extension[J]. IEEE Micro, 2017, 37(2): 26−39

[5] Tagliavini G, Mach S, Rossi D, et al. Design and evaluation of smallfloat SIMD extensions to the RISC-V ISA[C]//Proc of Design, Automation & Test in Europe Conf & Exhibition. Piscataway, NJ: IEEE, 2019: 654−657

[6] Fowers J, Ovtcharov K, Papamichael M T, et al. A configurable cloud-scale DNN processor for real-time AI[C/OL]//Proc of the 45th ACM/IEEE Annual Int Symp on Computer Architecture. Piscataway, NJ: IEEE, 2018[2023-01-13]. https://web.tecnico.ulisboa.pt/~joaomiguelvieira/public/PTDC/EEI-HAC/4583/2021/JF18.pdf

[7] Khailany B, Dally W, Kapasi U, et al. Imagine: Media processing with streams[J]. IEEE Micro, 2001, 21(2): 35−46

[8] ANDES Technology, Andescore™ NX27V processor[EB/OL]. 2020[2023-01-13].https://www.andestech.com/en/products-solutions/andescore-processors/riscv-nx27v/

[9] Liao Heng, Tu Jiajin, Xia Jing, et al. DaVinci: A scalable architecture for neural network computing[C/OL]//Proc of the 31st IEEE Hot Chips Symp. Piscataway, NJ: IEEE, 2019[2023-01-13]. https://pdfs.semanticscholar.org/4ca3/69ee20343433bd50c288c01ebcba2d6a03b2.pdf

[10] Krashinsky R, Batten C, Hampton M, et al. The vector-thread architecture[J]. IEEE Micro, 2004, 24(6): 84−90

[11] Lee Y, Avizienis R, Bishara A, et al. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators[C]//Proc of the 38th Annual Int Symp on Computer Architecture. Piscataway, NJ: IEEE, 2011: 129−140

[12] Lin Yuan, Lee H, Woh M, et al. SODA: A low-power architecture for software radio[C]//Proc of the 33rd Int Symp on Computer Architecture. Piscataway, NJ: IEEE, 2006: 89−101

[13] Chen Shuming, Wang Yaohua, Liu Sheng, et al. FT-Matrix: A coordination-aware architecture for signal processing[J]. IEEE Micro, 2014, 34(6): 64−73

[14] Woh M, Seo S, Mahlke S, et al. AnySP: Anytime anywhere anyway signal processing[J]. IEEE Micro, 2010, 30(1): 81−91

[15] Chen Y, Emer J, Sze V. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks[C]//Proc of the 43rd ACM/IEEE Annual Int Symp on Computer Architecture. Piscataway, NJ: IEEE, 2016: 367−379

[16] Cong J, Xiao Bingjun. Minimizing computation in convolutional neural networks[C]//Proc of the 24th Int Conf on Artificial Neural Networks. Berlin: Springer, 2014: 281−290

[17] Chellapilla K, Puri S, Simard P. High performance convolutional neural networks for document processing[C/OL]//Proc of the 10th Int Workshop on Frontiers in Handwriting Recognition. La Baule: Université de Rennes 1, 2006[2023-01-13]. https://inria.hal.science/file/index/docid/112631/filename/p1038112283956.pdf

[18] Hsu S, Agarwal A, Anders M, et al. A 280 MV-to-1.1 V 256b reconfigurable SIMD vector permutation engine with 2-dimensional shuffle in 22 nm tri-gate CMOS[J]. IEEE Journal of Solid-State Circuits, 2013, 48(1): 118−127

[19] Raghavan P, Munaga S, Ramos E, et al. A customized cross-bar for data-shuffling in domain-specific SIMD processors[C]//Proc of the 20th Int Conf on Architecture of Computer System. New York: ACM, 2007: 57−68

[20] National Center for Biotechnology Information. PubChem patent summary for US−7631025-B2, method and apparatus for rearranging data between multiple registers[EB/OL].[2023-01-13]. https://pubchem.ncbi.nlm.nih.gov/patent/US-7631025-B2

[21] Veluri H, Li Yida, Niu Xuhua, et al. High-throughput, area-efficient, and variation-tolerant 3-D in-memory compute system for deep convolutional neural networks[J]. IEEE Internet of Things Journal, 2021, 8(11): 9219−9232

[22] Wang Yaohua, Li Chen, Liu Chang, et al. Advancing DSP into HPC, AI, and beyond: Challenges, mechanisms, and future directions[J]. Transactions on High Performance Computing, 2021, 3(1): 114−125

[23] Wang Yaohua, Wang Dong, Chen Shuming, et al. Iteration interleaving-based SIMD lane partition[J]. ACM Transactions on Architecture and Code Optimization, 2016, 12(4): 1−18

[24] Yang Xuejun, Yan Xiaobo, Xing Zuocheng, et al. A 64-bit stream processor architecture for scientific applications[J]. ACM SIGARCH

Computer Architecture News, 2007, 35(2): 210−219

[25] Hennessy J, Patterson. Computer Architecture: A Quantitative Approach[M]. Amsterdam: Elsevier, 2017

[26] Eckert C, Wang Xiaowei, Wang Jingcheng, et al. Neural cache: Bit-serial in-cache acceleration of deep neural networks[C]//Proc of the 45th ACM/IEEE Annual Int Symp on Computer Architecture. Piscataway, NJ: IEEE, 2018: 383−396

[27] Wang Jingcheng, Wang Xiaowei, Eckert C, et al. A 28-nm compute SRAM with bit-serial logic/arithmetic operations for programmable in-memory vector computing[J]. IEEE Journal of Solid-State Circuits, 2020, 55(1): 76−86

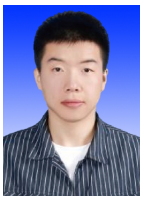
[28] Aga S, Jeloka S, Subramaniyan A, et al. Compute caches[C]//Proc of IEEE Int Symp on High Performance Computer Architecture. Piscataway, NJ: IEEE, 2017: 481−492

[29] Huang Libo, Shen Li, Wang Zhiying, et al. SIF: Overcoming the limitations of SIMD devices via implicit permutation[C]//Proc of the 16th IEEE Int Symp on High Performance Computer Architecture. Piscataway, NJ: IEEE, 2010: 303−314

[30] Cochran W, Cooley J, Favin D, et al. What is the fast Fourier transform[J]. IEEE Transactions on Audio and Electroacoustics, 1967, 15(2): 45−55

[31] Karen S, Andrew Z. Very deep convolutional networks for large-scale image recognition[J]. arXiv preprint, arXiv: 1409.1556, 2015

[32] Alex K, Ilya S, Geoffery E. ImageNet classification with deep convolutional neural networks[J]. Communications of the ACM, 2017, 60(6): 84−90



Zhang Dunbo, born in 1996. PhD candidate. His main research interests include computer architecture, processing-in-memory and SRAM.

张敦博, 1996 年生. 博士研究生. 主要研究方向为计算机体系结构、存内计算、静态随机存储器.



Zeng Lingling, born in 1996. Master candidate. His main research interests include computer architecture, processing-in-memory and SRAM.

曾灵灵, 1996 年生. 硕士研究生. 主要研究方向为计算机体系结构、存内计算、静态随机存储器.



Wang Ruoxi, born in 2000. Master. Her main research interest includes computer architecture.

王若曦, 2000 年生. 硕士. 主要研究方向为计算机体系结构.



Wang Yaohua, born in 1985. PhD, professor, PhD supervisor. His main research interests include high performance computing, accelerator architecture, and memory system optimization.

王耀华, 1985 年生. 博士, 研究员, 博士生导师. 主要研究方向为高性能计算、加速器体系结构、存储系统优化.



Shen Li, born in 1976. PhD, professor, PhD supervisor. His main research interests include multi/many-core architecture, runtime & compilation optimization, processing-in-memory, and high performance computing.

沈　立, 1976 年生. 博士, 教授, 博士生导师. 主要研究方向为多核/众核体系结构、运行时编译优化、存内计算、高性能计算.