# SIMDRAM：

# 使用DRAM进行位串行SIMD处理的框架

# 摘要

有人提议使用DRAM处理有限的一组基本操作（即逻辑操作、加法）。然而，为了能够完全采用使用DRAM的处理，有必要为更复杂的操作提供支持。在本文中，我们提出了SIMDRAM，这是一个灵活的通用处理框架，它使用DRAM框架（1）实现复杂操作的高效实现，（2）提供灵活的机制来支持任意用户定义操作的实现。SIMDRAM框架包括三个关键步骤。第一步构建给定所需操作的有效MAJ/NOT。第二步将保留用于计算的DRAM行分配给操作的输入和输出操作数，并生成所需的DRAM命令序列，以在DRAM中执行所需操作的MAJ/NOT。第三步使用内存控制器内的SIMDRAM控制单元，通过执行框架第二步中生成的DRAM命令，从开始到结束管理操作的计算。我们为SIMDRAM框架设计了硬件和ISA支持，以（1）解决关键的系统集成挑战，（2）允许程序员在不改变硬件的情况下使用新的SIMDRAM操作。

我们使用广泛的操作和七个实际应用来评估SIMDRAM的可靠性、区域开销、吞吐量和能效，以证明SIMDRAM的通用性。我们使用单个DRAM库进行的评估表明：（1）在16次操作中，SIMDRAM提供了2.0倍的吞吐量和2.6倍的能量效率，这是一种使用DRAM机制的最先进处理技术；（2） 在七个实际应用程序中，SIMDRAM提供了2.5倍的范围性能。SIMDRAM使用16个DRAM存储组，在16次操作中，分别为CPU和高端GPU提供（1）88倍和5.8倍的吞吐量，257倍和31倍的能效；（2） CPU和GPU的性能分别为21倍和2.1倍，超过7个实际应用程序。SIMDRAM在高端CPU中的区域开销仅为0.2%。

关键字：大容量按位操作、内存处理、DRAM

# 1介绍

现代应用中数据的日益普及和规模的不断扩大导致了传统计算机体系结构中计算的高能耗和延迟成本。在内存（如DRAM）和CPU之间跨带宽有限的内存通道移动大量数据，会消耗现代系统总能量的60%以上[17,98]。为了降低这种成本，**内存处理（PIM）范式**将计算移到更靠近数据驻留的位置，减少（在某些情况下消除）在内存和处理器之间移动数据的需要。

PIM有两种主要方法[40,99]：（1）在内存附近处理，其中PIM逻辑被添加到与内存相同的芯片或3D堆叠内存的逻辑层[3-5,14,17-19,25,27,29,38,39,41,46,53-55,61,64,68,77,82,90,103,107,108,113,116,119,120,144,145]；以及（2）使用存储器的处理，其利用存储器单元本身的操作原理，通过使单元之间能够相互作用来执行计算[1、23、24、28、35、37、86、123–125、127、129、132、133、141]。由于使用内存的处理直接在内存阵列中运行，因此它受益于内存阵列中可用的大内部带宽和并行性，而这是处理近内存解决方案无法利用的。

使用内存体系结构进行处理的一种常见方法是使用块位计算。许多广泛使用的数据密集型应用程序（如数据库、神经网络、图形分析）严重依赖于广泛的简单（如AND、OR、XOR）和复杂（如等式检查、乘法、加法）位运算。Ambit[122，124]，一种使用内存加速器的DRAM内处理，是第一个提出利用DRAM的模拟操作原理来执行大容量按位AND、OR和NOT逻辑操作的工作。受Ambit的启发，许多以前的作品探索了能够执行内存位操作的DRAM（以及NVM）设计[6、8–12、24、37、50、57、87、141]。然而，一个主要的缺点阻碍了这些方案的广泛应用：**它们只支持基本操作（例如布尔运算、加法），不能灵活方便地支持新的和更复杂的操作**。以前的一些工作建议使用支持更复杂操作的DRAM设计进行处理[24,86]。然而，此类设计（1）需要对DRAM子阵列进行重大更改，（2）仅支持有限的特定操作集，缺乏支持新操作的灵活性，无法满足可能从内存计算中受益的各种应用。

我们在本文中的目标是设计一个框架，通过高效地实现复杂的操作并提供支持新的所需操作的灵活性来帮助采用DRAM进行处理。为此，我们提出了SIMDRAM，这是一种使用DRAM框架的端到端处理，它提供编程接口、ISA和硬件支持（1）高效计算复杂操作，以及（2）提供根据需要执行任意操作的能力，所有这些都在DRAM大规模并行SIMD基板中。在其核心，我们围绕DRAM基板构建SIMDRAM框架，该框架支持两种先前提出的技术：（1）DRAM中的垂直数据布局，（2）用于计算的基于多数的逻辑。

垂直数据布局。支持位移位操作对于实现复杂计算（如加法或乘法）至关重要。先前的工作表明，对DRAM中的数据采用垂直布局[6、15、28、35、37、51、52、62、130、138]，这样操作数的所有位都放在单个DRAM列中（即，在单个位线中），无需在DRAM中添加额外逻辑来实现移位[24、86]。因此，SIMDRAM通过在DRAM中以垂直方式存储操作数来支持高效的位移位操作。这为SIMDRAM提供了两个关键好处。首先，可以通过简单地将DRAM行复制到另一行（使用RowClone[123]、LISA[21]、NoM[128]或FIGARO[140]）来执行位移位操作。例如，SIMDRAM可以通过将DRAM行j中的数据复制到DRAM行j+1来执行一次左移位操作。（请注意，虽然SIMDRAM支持位移位，但我们可以通过简单地更改读取移位数据的SIMDRAM命令的行索引来优化许多应用程序，以避免显式移位操作的需要）。其次，SIMDRAM实现了大规模并行，其中每个DRAM列通过将一个操作的源操作数和目标操作数放在同一DRAM列中，作为SIMD通道运行。

基于多数的计算。先前的工作使用多数运算来实现基本逻辑运算[37、86、122、124]（例如，AND、OR）或加法[6、9、24、36、37、86]。然后将这些基本操作用作基本构建块，以实现DRAM计算中的目标。SIMDRAM通过直接使用逻辑上完整的多数集（MAJ）而不是在DRAM计算中实现的操作，扩展了多数操作的使用。与使用基本逻辑操作作为内存计算的构建块相比，这样做使SIMDRAM能够实现更高的性能、吞吐量和更低的能耗。我们发现，与使用基本逻辑运算and、OR和NOT相比，使用MAJ and NOT进行计算通常需要更少的DRAM命令。

为了通过灵活支持新的和更复杂的操作来帮助采用DRAM进行处理，SIMDRAM解决了两个关键挑战：（1）如何在DRAM操作中合成新的任意操作；（2）如何为此类新增操作利用优化的实施和控制流程，同时考虑DRAM内处理的关键限制（例如，破坏输入数据的DRAM操作、能够使用DRAM进行处理的DRAM行数量有限，以及需要避免昂贵的DRAM内拷贝）。因此，SIMDRAM是第一个使用DRAM进行处理的端到端框架。SIMDRAM提供了（1）一种有效的算法，用于生成给定期望操作的MAJ/NOT实现；（2） 将DRAM行适当分配给操作数的算法，以及将计算映射到有效的DRAM命令序列以执行任何基于MAJ的计算的算法；（3）编程接口、ISA支持和所需的硬件组件，以（i）在不修改硬件的情况下计算任何新的用户定义的DRAM操作，以及（ii）编程内存控制器，以向相应的DRAM行发出DRAM命令并正确执行计算。这种端到端支持使SIMDRAM成为一种整体方法，有助于采用使用DRAM的处理（1）通过向用户提供简化的界面来添加所需操作，从而实现支持新的DRAM内操作的灵活性，以及（2）消除向DRAM添加额外逻辑的需要。

SIMDRAM框架有效地支持各种不同类型的操作。在这项工作中，我们使用16个操作的示例集演示SIMDRAM框架的功能，包括（1）N输入逻辑操作（例如，2个以上输入位的AND/OR/XOR）；（2） 关系操作（例如，相等/不相等检查、大于、最大、最小）；（3） 算术运算（例如，加法、减法、乘法、除法）；（4） 预测（例如，if-then-else）；和（5）其他复杂操作，如比特计数和ReLU[45]。SIMDRAM框架不限于这16个操作，并且可以使用DRAM处理其他现有和未来的操作。SIMDRAM非常适合（i）简单友好，（ii）具有常规访问模式，以及（iii）内存受限的应用程序类。这类应用在数据库分析、高性能计算、图像处理和机器学习等领域很常见。

我们比较了SIMDRAM与不同的先进计算平台（CPU、GPU和DRAM计算机制的范围[124]）的优势。我们综合评估了SIMDRAM的可靠性、面积开销、吞吐量和能效。我们利用SIMDRAM框架来加速来自机器学习、数据库和图像处理的七个应用程序内核（VGG-13[131]、VGG-16[131]、LeNET[75]、kNN[83]、TPC-H[137]、比特编织[88]、亮度[44]）。使用单个DRAM存储库，SIMDRAM提供（1）2.0倍的吞吐量和2.6倍的能量效率[124]，在16个实施的操作中取平均值；和（2）2.5×范围的性能，在七个应用程序内核上的平均值。与CPU和高端GPU相比，使用16个DRAM组的SIMDRAM在16次操作中平均提供（1）257倍和31倍的能效，以及88倍和5.8倍的CPU和GPU吞吐量；（2）CPU和GPU的平均性能分别为7个应用程序内核的21倍和2.1倍。SIMDRAM在AMBIT之上并没有额外的区域开销，在高端CPU中总的区域开销仅为0.2%。我们还评估了SIMDRAM在不同程度的制造工艺变化下的可靠性，并观察到当DRAM工艺技术节点缩小到更小的尺寸时，它保证了正确的操作。

我们做出了以下主要贡献：

据我们所知，这是第一个提出框架的工作，该框架能够在通过使用DRAM处理构建的大规模并行SIMD基板中高效计算灵活的集合和广泛的操作。

SIMDRAM提供了一个三步框架，用于开发各种操作的高效可靠的基于MAJ/NOT的实现。我们设计了这个框架，并添加了硬件、编程和ISA支持，以（1）解决关键的系统集成难题，（2）允许程序员在不改变硬件的情况下定义和使用新的SIMDRAM操作。

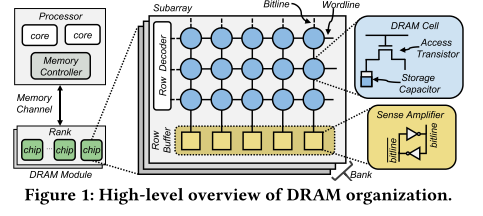
我们提供了SIMDRAM的详细参考实现，包括对应用程序、ISA和硬件所需的更改

我们评估了SIMDRAM在不同程度的过程变化下的可靠性，并观察到当DRAM技术扩展到更小的节点大小时，它保证了正确的操作。

# 2 背景

我们首先简要介绍一个典型DRAM芯片的体系结构。接下来，我们将介绍使用SIMDRAM构建在（RowClone[123]和Ambit[122124127]）之上的DRAM作品进行的先前处理，并解释基于多数的计算的含义。

2.1 DRAM基础

DRAM系统包括组件层次结构，如图1所示，从最高级别的通道开始。通道细分为列组，列组细分为多个列组（例如，8-16）。每个组由多个（例如64-128个）2D单元阵列组成，这些单元阵列称为子阵列。子阵列中的单元被组织成多行（例如512-1024）和多列（例如2-8KB）[65、78、79]。一个单元由一个存取晶体管和一个存储电容组成，存储电容利用其电压电平对一个数据位进行编码。同一列中所有单元的访问晶体管的源节点将单元的存储电容连接到同一位线。类似地，同一行中所有单元的访问晶体管的栅极节点将单元的访问晶体管连接到同一字线。

断言字线时，沿字线的所有单元都连接到它们相应的位线，这会根据存储在每个单元电容器中的值扰动每个位线的电压。连接到每个位线的双端感测放大器感测位线（连接到一个端子）和参考电压（通常为1/2VDD；连接到另一个端子）之间的电压差，并将其放大到CMOS可读值。在这样做时，连接到参考电压的感测放大器端子被放大到相反的（即，否定的）值，其被示为图1中的位线端子。每个子阵列中的一组感测放大器形成一个逻辑行缓冲区，只要行打开（即，字线继续被断言），该缓冲区就保持感测数据。DRAM中的读写操作包括三个步骤：（1）激活。断言目标行的字线，该字线将行中的所有单元格连接到各自的位线。每个位线与其对应的单元电容共享电荷，由此产生的位线电压偏移由位线的检测放大器检测和放大。感测放大器完成放大后，行缓冲区包含最初存储在沿断言字线的单元格中的值。（2） RD/WR。然后，内存控制器向激活行内的列（即行缓冲区内的数据）发出读或写命令。（3） 预充电。通过禁用字线断开电容器与位线的连接，位线电压恢复到其静态状态（例如，通常为1/2VDD）。

2.2使用DRAM进行处理

2.2.1在DRAM行中复制。RowClone[123]是一种利用巨大的内部DRAM带宽在DRAM内高效复制行而无需CPU干预的机制。RowClone通过向同一子阵列中的源行a和目标行B发出两个连续的激活命令，然后再发出预充电命令，可以将源行a复制到目标行B。此命令序列称为AAP[124]。第一个激活命令将源行A的内容复制到行缓冲区中。第二个激活命令将目标行B中的单元格连接到位线。由于感测放大器在激活行B时已经感测并放大了源数据，因此行B的每个单元中的数据（即，电压电平）被存储在行缓冲器中的数据（即，行A的数据）覆盖。最近的工作[37]通过实验证明了在未经修改的现成DRAM芯片中执行DRAM行内复制操作的可行性。

2.2.2在DRAM中按位操作。Ambit[122、124、127]表明，同时激活三个DRAM行（通过称为Triple Row Activation，TRA的DRAM操作）可用于对三行单元格中包含的值执行逐位布尔AND、OR和NOT操作。当激活三行时，连接到每个位线的三个单元同时共享电荷，并导致位线的扰动。感测到扰动后，如果三个DRAM单元的至少两个电容器分别充电或放电，则感测放大器将位线电压放大至VDD或0。因此，TRA在每个位线上的三个DRAM单元之间产生布尔多数运算（MAJ）。多数运算MAJ仅当其一半以上的输入为1（0）时才输出1（0）。根据and（·）和OR（+）运算，三输入多数运算可以表示为MAJ（A，B，C）=A·B+A·C+B·C。

Ambit通过引入自定义行解码器（在§3.1中讨论）来实现MAJ，该解码器可以通过同时寻址三个字行来执行TRA。为了使用该解码器，Ambit定义了一个名为AP的新命令序列，该序列发出（1）一个TRA以计算三行的MAJ，然后（2）一个预充电以关闭所有三行。Ambit使用AP命令序列通过简单地将一个输入（例如C）设置为1或0来实现布尔AND和OR运算。通过将C设置为0（即，MAJ（A，B，0）=A AND B）来计算AND运算。OR运算通过将C设置为1（即，MAJ（A，B，1）=A OR B）来计算。

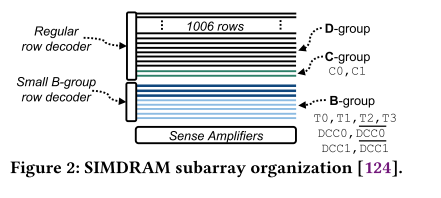
为了实现与AND和OR操作一起的功能完整性，Ambit通过利用DRAM感测放大器的差分设计来实现NOT操作。如§2.1所述，作为激活过程的一部分（图1中的位线），感测放大器已经生成感测值的补码。因此，Ambit只是将位线转发到子阵列中的一个特殊DRAM行，该行由带有两个访问晶体管的DRAM单元组成，称为双接触单元（DCC）。每个存取晶体管连接到感测放大器的一侧，并由单独的字线（d字线或n字线）控制。通过激活d字线或n字线，DCC行可以分别提供存储在行单元格中的真值或反值。

2.2.3基于多数的计算。由于制造工艺变化，同时激活多行会降低检测放大器读取值的可靠性，这会导致电路级电气特性的不均匀性（例如，单元电容水平的变化）[124]。随着（1）同时激活的行数增加，以及（2）技术更先进、尺寸更小的节点，这种影响会恶化。因此，尽管使用DRAM的处理可能支持具有三个以上输入的多数操作（如先前工作[6,9,87]所建议的），但我们使用DRAM的处理实现使用多数操作（N=3）所需的最小输入数来保持计算的可靠性。在论文[49]的最终版本中，我们通过SPICE模拟证明，与每个MAJ操作有三个以上输入的设计相比，使用三输入MAJ操作提供了更高的可靠性。使用3输入MAJ，使用DRAM基板的处理不需要对子阵列组织（图2）进行超出范围（§3.1）建议的修改。最近的工作[37]实验证明了通过激活未经修改的现成DRAM芯片中的三行来执行主要操作的可行性。

# 3SIMDRAM 概述

SIMDRAM是一种使用DRAM框架的处理，其目标是（1）实现复杂操作的高效实现，（2）提供灵活的机制来支持任意用户定义操作的实现。我们介绍SIMDRAM中的子阵列组织，描述SIMDRAM框架的概述，并解释如何将SIMDRAM集成到系统中。

3.1子阵组织

为了使用DRAM执行处理，SIMDRAM利用子阵列组织，该子阵列组织包含执行逻辑原语（即MAJ和NOT）的附加功能。该子阵列组织与Ambit的[124]相同，与DRISA的[86]相似。图2示出了SIMDRAM中类似于传统DRAM子阵列的子阵列的内部组织。SIMDRAM只需要对DRAM子阵列（即，可以同时激活三行的小行解码器）进行最小的修改即可实现计算。与Ambit类似，SIMDRAM将DRAM行分为三组：数据组（D组）、控制组（C组）和按位组（B组）。

D组包含存储程序或系统数据的常规行。C组由两个常量行组成，称为C0和C1，分别包含all-0和all-1值。这些行（1）用作给定SIMDRAM操作（例如，完全加法中的初始进位）的初始输入值，或（2）用于执行自然需要和/或操作（例如，和/或减少）的操作。D组和C组连接到常规行解码器，该解码器一次选择一行。

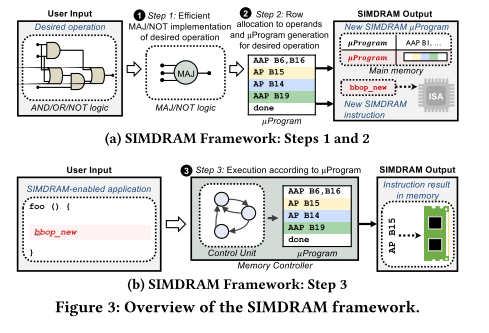
B组包含六个规则行，称为T0、T1、T2和T3；和两排双触点电池（见§2.2.2），其d字线分别称为DCC0和DCC1，其n字线分别称为~DCC0和~DCC1。B组行（称为计算行）被指定执行按位操作。它们都连接到一个特殊的行解码器，该解码器可以使用一个地址同时激活三行（即，执行TRA）

SIMDRAM使用1024行的典型子阵列大小[20、65、67、72、80]，将行寻址拆分为1006个D组行、2个C组行和16个B组行。

3.2框架概述

SIMDRAM是一个端到端的框架，它为用户提供了使用AAP/AP命令序列在DRAM中实现任意操作的能力。该框架包括三个关键步骤，如图3所示。该框架的前两个步骤使用户能够在DRAM中高效地执行任何所需的操作，而第三个步骤则从用户那里透明地控制DRAM中计算的执行流。我们在下面简要描述这些步骤，并在§4中详细讨论每个步骤。

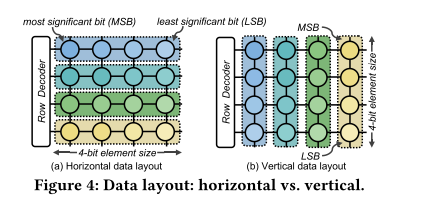
第一步(❶ 在图3a中，§4.1）从基于AND/OR/NOT的实现中构建给定期望操作的有效MAJ/NOT表示。具体而言，此步骤将所需操作作为输入，并使用逻辑优化来最小化所需的逻辑原语数量（因此，计算延迟）。因此，对于用户输入到SIMDRAM框架中的期望操作，第一步导出其基于MAJ/NOT的优化实现。

第二步(❷ 在图3a中，§4.2）将DRAM行分配给操作的输入和输出，并生成所需的DRAM命令序列以执行所需操作。具体而言，该步骤将基于主要/非主要的操作实现转换为AAP/AP。此步骤涉及（1）将DRAM中指定的计算行分配给操作数，（2）确定执行该操作所需的AAP/AP的优化序列。这样做时，SIMDRAM将特定操作所需的AAP/AP数量降至最低。该步骤的输出是一个µ程序，即存储在主内存中的AAP/AP的优化序列，将用于在运行时执行操作。

第三步(❸ 在图3b中，§4.3）执行µ程序以执行操作。具体而言，当用户程序遇到与SIMDRAM操作相关联的bbop指令（§5.2）时，bbop指令通过在内存控制器中执行其µ程序来触发SIMDRAM操作的执行。SIMDRAM使用内存控制器中的控制单元，该控制单元根据µ程序的指示，透明地向DRAM发出AAP/AP序列。一旦µ程序完成，操作结果将保存在DRAM中。

3.3在系统中集成SIMDRAM

正如我们在§1中所讨论的，SIMDRAM使用垂直布局对数据进行操作。图4示出了当采用水平数据布局（图4a）和垂直数据布局（图4b）时，如何在DRAM子阵列内组织数据。我们假设每个数据元素有四位宽，并且有四个数据元素（每个元素用不同的颜色表示）。在传统的水平数据布局中，数据元素存储在不同的DRAM行中，每个数据元素的内容从最高有效位到最低有效位（反之亦然）排列在一行中。相反，在垂直数据布局中，DRAM行仅保存多个数据元素的第i位（其中元素的数量由行的位宽度决定）。因此，当激活垂直数据布局组织中的单个DRAM行时，一次读取来自每个数据元素的单个数据位，从而启用DRAM位内串行并行计算[6、15、46、86、127、130]。



为了保持与传统系统软件的兼容性，我们将常规数据存储在传统的水平布局中，并提供硬件支持（在§5.1中解释）以将水平ID输出数据转置到垂直布局中，用于DRAM内计算。为了简化程序集成，我们提供ISA扩展，向程序员公开SIMDRAM操作（§5.2）。

4 SIMDRAM框架

我们使用完整加法操作作为运行示例，描述§3.2中介绍的SIMDRAM框架的三个步骤。

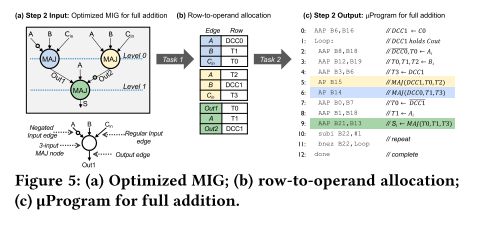
4.1步骤1：高效的MAJ/NOT实现

SIMDRAM使用MAJ和NOT逻辑原语的逻辑完整集在DRAM计算中实现，与使用AND/OR/NOT相比，执行给定操作需要更少的AAP/AP命令序列。因此，SIMDRAM框架中第一步的目标是构建给定操作的优化MAJ/NOT实现，该操作使用尽可能少的AAP/AP命令序列执行操作，从而最小化操作的延迟。为此，步骤1使用先前工作[7]形式化的转换过程，将给定操作的AND/OR/NOT表示转换为优化的MAJ/NOT表示。

转换过程使用基于图形的逻辑原语表示法，AND/OR/NOT逻辑称为AND–OR–Inverter graph（AOIG），MAJ/NOT逻辑称为MARY–Inverter graph（MIG）。AOIG是一种有向无环图形式的逻辑表示结构，其中每个节点表示AND或OR逻辑原语。AOIG中的每条边表示节点之间的输入/输出依赖关系。节点的传入边表示节点的输入操作数，而节点的传出边表示节点的输出。AOIG中的边可以是正则的，也可以是可补的（表示反转的输入操作数；由边上的气泡表示）。边的方向遵循从输入到输出的自然计算方向。类似地，MIG是一个有向无环图，其中每个节点表示一个三输入MAJ逻辑原语，每个规则/补充边表示该节点表示的MAJ原语的一个输入或输出。转换过程由两部分组成，这两部分对输入AOIG进行操作。

转换过程的第一部分简单地用MAJ原语替换AND/OR原语。每个两个输入AND或or原语被简单地替换为一个三输入MAJ原语，其中一个输入分别绑定到逻辑0或逻辑1。这种简单的替代产生了一个MIG，它可以正确地复制输入AOIG的功能，但MIG可能效率低下。

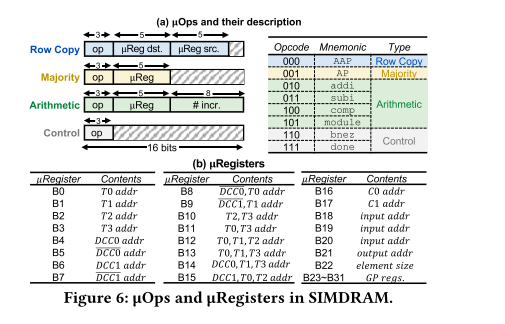
转换过程的第二部分采用低效的MIG，并使用贪婪算法（参见本文的最终版本[49]）应用一系列转换，确定如何将多个主要原语合并为具有相同功能的少量主要原语。这将产生较小的MIG，这反过来需要较少的逻辑原语来执行与未优化MIG（以及因此输入AOIG）执行的相同操作。图5a显示了通过完全添加操作的转换过程产生的优化MIG。



4.2步骤2：µ程序生成

每个SIMDRAM操作都存储为一个µ程序，该程序由SIMDRAM用于在DRAM中执行SIMDRAM操作的一系列微体系结构操作（µ操作）组成。第二步的目标是采用步骤1中生成的优化MIG，并生成一个µ程序，该程序执行MIG代表的SIMDRAM操作。为此，如图5所示，框架的第二步在优化的MIG上执行两个关键任务：（1）将DRAM行分配给操作数，这将MIG中每个主要节点的每个输入操作数（即输入边）分配给DRAM行（图5b）；以及（2）生成µ程序，该程序创建执行MIG中的MAJ和NOT（即节点）的µOp系列，同时保持正确的计算流程（图5c）。在本节中，我们首先描述SIMDRAM中使用的µOps（§4.2.1）。其次，我们解释了将DRAM行分配给MIG to DRAM行中主要节点的输入操作数的过程（§4.2.2）。第三，我们解释生成µ程序的过程（§4.2.3）。

4.2.1 SIMDRAM µOps。图6a显示了我们在SIMDRAM中实现的一组µOps。每个µOp都是（1）SIMDRAM向子阵列发出的命令序列，用于执行部分DRAM内计算，或（2）SIMDRAM控制单元用于管理SIMDRAM操作执行的控制操作（见§4.3）。我们进一步将命令序列µOps分解为三种类型之一：（1）行复制，一种使用AAP命令序列执行从源内存地址到目标内存地址的DRAM内复制的µOps；（2） MAJ，一种µOp，使用AP命令序列在三个DRAM行上执行多数逻辑原语（即，它执行TRA）；和（3）算术运算，四个µOp，对控制操作执行所需的SIMDRAM控制单元寄存器执行简单算术运算（addi、subi、comp、module）。我们提供两个控制操作µOps，以支持SIMDRAM控制流中的循环和终止（bnez，done）。



在µ程序生成过程中，SIMDRAM框架将MIG转换为一系列µOps。请注意，MIG表示操作的1位宽计算。因此，为了实现多位宽的SIMDRAM操作，框架需要将实现MIG的一系列µOps重复n次，其中n是SIMDRAM操作数中的位数。为此，SIMDRAM使用算术和控制µOps将1位宽的计算重复n次，对程序员来说是透明的。

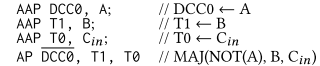
为了支持µOps的执行，SIMDRAM利用位于SIMDRAM控制单元（§4.3）中的一组µ寄存器（图6b）。该框架使用µ寄存器（1）存储子阵列（µ寄存器B0–B17）的B组和C组（图3.1）中DRAM行的内存地址，（2）存储用于计算的输入和输出行的内存地址（µ寄存器B18–B22）和（3）在执行算术和控制操作期间作为通用寄存器（µ寄存器B23–B31）。

4.2.2任务1：将DRAM行分配给操作数。此任务的目标是将DRAM行分配给操作MIG中每个MAJ节点的输入操作数（即传入边），从而使计算操作所需的µOp总数最小化。为此，受线性扫描寄存器分配算法[115]的启发，我们提出了一种新的分配算法。然而，与寄存器分配算法不同，我们的分配算法考虑了特定于使用DRAM处理的两个额外约束：（1）在DRAM中执行MAJ具有破坏性行为，即TRA用MAJ输出覆盖三个输入行的原始值；和（2）指定用于执行逐位运算的计算行（即图2中的B组）的数量是有限的（如§3.1所述，每个子阵列中只有六个计算行）。

SIMDRAM分配算法接收操作的MIG作为输入。该算法假设在开始计算操作之前，操作的输入操作数已使用垂直布局（§3.3）存储在子阵列中D组的单独行中。然后，该算法从MIG最高级别（例如，图5a中的级别0）的最左侧主节点开始进行拓扑遍历，将计算行分配给MIG当前级别中每个主节点的输入操作数，然后移动到图形的下一个较低级别。一旦DRAM行分配给MIG中所有MAJ节点的所有输入操作数，该算法完成。图5b示出了这些分配作为完整添加示例的任务1的输出。然后，在第二步（§4.2.3）的第二个任务中使用生成的行到操作数分配来生成µOp序列，以计算MIG表示的操作。我们在论文的最终版本[49]中描述了我们的分配算法。

4.2.3任务2：生成µ程序。此任务的目标是使用任务1中的MIG和DRAM行分配为SIMDRAM操作生成µ程序的µOps。为此，任务1（1）将MIG转换为一系列行拷贝和多数µOp（即AAP/APs），（2）优化µOp系列以减少AAP/APs的数量，以及（3）利用SIMDRAM的算术和控制µOps将MIG描述的一位串行操作概括为n位操作。

将MIG转换为一系列的Row Copy和Ops。任务1期间产生的分配决定了在µ程序期间如何将DRAM行分配给MIG中的每条边。有了这些信息，框架可以生成适当的行拷贝和操作序列，以反映DRAM中MIG的计算。为此，我们按拓扑顺序遍历输入MIG。对于每个节点，我们首先将行复制µOps（使用AAP命令序列）分配给节点的边缘。然后，在分配给节点的每个边缘的DRAM行分配之后，我们分配一个多数µOp（使用AP命令序列）来执行当前MAJ节点。框架对MIG中的所有节点重复此过程。为了说明，我们假设SIMDRAM分配算法将DRAM行DCC0、T1和T0分别分配给全加MIG中蓝色节点的边A、B和Cin（图5a）。然后，当访问此节点时，我们生成以下一系列µOp：



优化µOps系列。在遍历MIG中的所有节点并生成适当的µOps系列后，我们通过合并AAP/AP命令序列来优化µOps系列，我们可以在两种情况中的一种情况下执行此操作。

案例1：如果所有的µOps都有相同的µ寄存器源作为输入，我们可以合并一系列行拷贝µOps。例如，考虑一系列将数据数组A复制到行T2和T3的两个AAP。我们可以将这一系列AAP合并为一个AAP，该AAP被发布到µ寄存器B10中存储的字线地址（见图6a）。该字线地址利用B组中的特殊行解码器（其是范围子阵列结构[124]的一部分）使用单个激活命令一次激活组中的多个DRAM行。对于我们的示例，激活µ寄存器B10允许AAP命令序列将阵列A同时复制到行T2和T3中。

案例2：当AAP的目的地是AP使用的行之一时，将AP命令序列（即，多数µOp）与AAP命令序列（即，行复制µOp）连接起来。例如，考虑在DRAM行T0、T1和T2上执行MAJ逻辑原语的AP（将结果存储在所有三行中），接着是将登记寄存器B12（指行T0、T1和T2）复制到行T3的AAP。AP后跟AAP将多数值放入所有四行（T0、T1、T2、T3）。这两个命令序列可以合并为一个AAP（AAP T3、B12），因为第一个激活将通过同时激活所有三行自动执行T0、T1和T2行的多数。然后，第二次激活将这些行中的值复制到T3中。

将位串行操作概括为n位操作。一旦所有潜在的µOp合并完成，框架现在就有了优化的1位计算版本。我们将这个1位µOp系列推广到循环体中，循环体重复n次以实现n位操作。我们利用SIMDRAM中可用的算术运算和控制运算来协调n位计算。需要用于下一位计算的一位计算产生的数据（例如，完全相加的进位）在两次计算中保存在B组行中，允许位到位的数据传输，而无需专用移位电路。

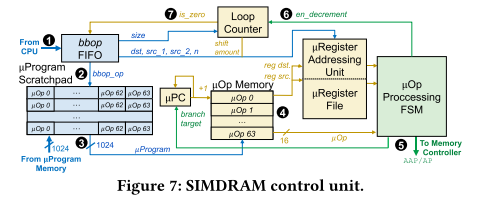
在该步骤之后生成的最终系列µOp随后被打包到µ程序中并存储在DRAM中以供将来使用。2图5c显示了在步骤2结束时为完全添加操作生成的最终µOp程序。该图显示了生成完全加法的1位实现（第2-9行）的优化µOp系列，以及用于启用加法的算术和控制µOpsn-操作的位实现（第10-11行）。

程序抽象的好处。我们用于存储SIMDRAM操作的µ程序抽象为框架提供了三个主要优势。首先，它允许SIMDRAM最小化实现SIMDRAM操作所需的新CPU指令总数，从而减少SIMDRAM对ISA的影响。虽然不同的实现可以使用更多的新CPU指令来表示更细粒度的操作（例如，AAP），但我们认为使用最少的一组新CPU指令可以简化采用和软件设计。其次，µ程序抽象实现了更小的应用程序二进制大小，因为需要放在应用程序二进制中的唯一信息是主内存中µ程序的地址。第三，µ程序提供了一种抽象，使最终用户不必使用与布尔逻辑编程等效的主要/非操作进行低级编程。我们在§5.2中讨论了用户程序如何调用SIMDRAMµ程序。

4.3步骤3：操作执行

一旦框架将生成的SIMDRAM操作µ程序存储在DRAM中，SIMDRAM硬件现在可以接收执行该操作的程序请求。为此，我们将讨论SIMDRAM控制单元，它在运行时处理µ程序的执行。控制单元设计为内存控制器的扩展，对程序员是透明的。程序使用bbop指令（由Ambit[124]引入）发出执行SIMDRAM操作的请求，bbop指令是允许程序与SIMDRAM框架交互的CPU ISA扩展之一（见§5.2）。每个SIMDRAM操作对应于不同的bbop指令。在接收到请求后，控制单元从存储器加载与请求的bbop相对应的µ程序，并在µ程序中执行µOps。由于SIMDRAM操作的所有输入数据元素可能不适合一个DRAM行，因此控制单元重复µ程序i次，其中i是数据元素总数除以单个DRAM行中的元素数。

图7显示了SIMDRAM控制单元的框图，该控制单元由九个主要组件组成：（1）从程序接收bbop的bbop FIFO，（2）在DRAM中分配的µ程序内存（图中未显示），（3）保存常用µ程序的µ程序暂存器，（4）保存当前运行的µ程序的µOp的µOp存储器，（5）生成映射到DRAM行的µ寄存器所使用的物理行地址的µ寄存器寻址单元（基于图6中B0–B17的µ寄存器到行分配），（6）保存非行映射µ寄存器的µ寄存器文件（图6中的B18–B31），（7）跟踪µ程序需要在其上执行的剩余数据元素数量的循环计数器，（8）控制执行流并发出AAP/AP命令序列的µOp处理FSM，以及（9）µ程序计数器（µPC）.SIMDRAM为µ程序内存保留一个DRAM区域，以存储与所有SIMDRAM操作相对应的µ程序。在运行时，控制单元将最常用的µ程序存储在µ程序草稿行中，以减少从DRAM获取µ程序的开销。



在运行时，当运行用户程序的CPU到达bbop指令时，它将bbop转发给SIMDRAM控制单元（图7中的1）。控制单元将bbop放入bbop FIFO中。控制单元通过一个四阶段程序，一次执行一个排队的bbops。

在第一阶段，控制单元获取并解码FIFO（2）头部的bbop。解码bbop涉及（1）将µ程序草稿行的索引设置为bbop操作码；（2） 将对所有元素执行操作所需的循环迭代次数（即，数据元素的数量除以单个DRAM行中的元素数量）写入循环计数器；以及（3）将参与计算的源阵列和目标阵列的基本DRAM地址以及每个数据元素的大小写入µ寄存器寻址单元。

在第二阶段，控制单元将当前在µProgram Scratchpad中索引的µProgram复制到µOp存储器（3）。此时，控制单元准备开始执行µ程序，一次执行一个µOp.

在第三阶段，从µOp内存中提取当前µOp，该内存由µPC编制索引。µOp处理FSM解码µOp，并确定需要哪些µ寄存器(❹). 对于µ寄存器B0–B17，µ寄存器寻址单元生成与请求的寄存器相对应的DRAM地址（见图6），并将地址发送给µOp处理FSM。对于µ寄存器B18–B31，µ寄存器文件将寄存器值提供给µOp处理FSM。

在第四阶段，µOp处理FSM执行µOp。如果µOp是一个命令序列，则将相应的命令发送到内存控制器的请求队列（5），并递增µPC。如果µOp是一个已完成的控制操作，则表示已为当前迭代执行了所有命令序列µOps。µOp处理FSM然后递减循环计数器（6）。如果递减循环计数器大于零，则µOp处理FSM移动存储在µ寄存器寻址单元中的基本源地址和目标地址，以移动到下一组数据元素3，并将µPC重置为µOp内存中的第一个µOp。如果递减循环计数器等于零，则表示控制单元已完成当前bbop的执行。然后，控制单元从bbop FIFO（7）获取下一个bbop，并为下一个bbop重复所有四个阶段。

4.4支持的操作

我们使用我们的框架有效地支持各种不同类型的操作。在本文中，我们评估了（在§7中）一组针对n位数据元素的五种不同类型的16个SIMDRAM操作：（1）n输入逻辑操作（OR-/AND-/和-/XOR还原）；（2） 关系操作（相等/不相等检查、大于/小于检查、大于或等于检查以及集合中的最大/最小元素）；（3） 算术运算（加法、减法、乘法、除法和绝对值）；（4） 谓词（if-then-else）；和（5）其他复杂操作（位计数和ReLU）。我们支持四种不同的元素大小，它们对应于流行编程语言中的数据类型大小（8位、16位、32位、64位）。

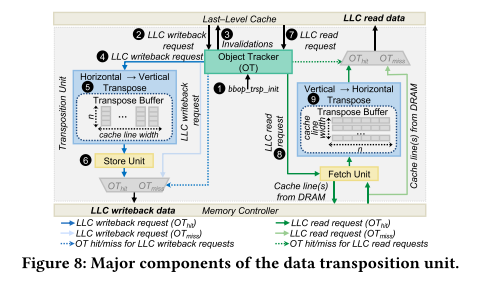
5.SIMDRAM的系统集成

我们讨论了在实际系统中集成SIMDRAM的四个关键挑战，以及我们如何解决这些挑战：（1）数据布局和SIMDRAM如何管理在垂直布局中存储DRAM计算所需的数据（§5.1）；（2） SIMDRAM的ISA扩展和编程接口（§5.2）；（3） SIMDRAM如何管理大量数据的计算（§5.3）；（4） SIMDRAM的安全影响（§5.4）。在论文的最终版本[49]中，我们讨论了我们如何应对其他关键集成挑战，包括：（1）SIMDRAM如何处理一些主要的系统挑战，如页面错误、地址转换、一致性和中断；（2） SIMDRAM当前的局限性。

5.1数据布局

我们设想SIMDRAM是对传统处理元素的补充（而不是替代）。因此，支持SIMDRAMenabled的系统中的程序可以具有CPU指令和SIMDRAM指令的组合，两者之间可能存在数据共享。然而，尽管SIMDRAM操作垂直布局的数据（§3.3），但其他系统组件（包括CPU）希望数据以传统的水平格式布局，这使得SIMDRAM和CPU指令之间共享数据具有挑战性。为了应对这一挑战，SIMDRAM中的内存管理需要（1）同时支持DRAM中的水平和垂直数据布局；（2）将SIMDRAM使用的垂直布局数据转换为CPU使用的水平布局，反之亦然。我们不能依赖软件（如编译器或应用程序支持）来处理数据布局转换，因为这将通过片上内存控制器，并在转换期间在DRAM和CPU之间引入显著的数据移动，从而导致延迟。为了避免转换过程中的数据移动，SIMDRAM使用位于最后一级缓存（LLC）和内存控制器之间的专用硬件单元（称为数据转换单元），将数据从水平数据布局转换为垂直数据布局，反之亦然。转置单元确保对于每个SIMDRAM对象，每当数据在缓存中时，其对应的数据都处于水平布局，每当数据在DRAM中时，其对应的数据都处于垂直布局。

图8示出了转置单元的关键部件。转置单元在称为对象跟踪器的转置单元中的小型缓存中跟踪SIMDRAM操作使用的内存对象。为了在分配SIMDRAM使用的内存对象时向对象跟踪器添加条目，程序员在分配内存对象的malloc之后立即添加一条名为bbop\_trsp\_init（§5.2）的初始化指令(❶ 如图8所示）。假设系统采用延迟分配，bbop\_trsp\_init指令会通知操作系统（OS）内存对象是SIMDRAM对象。这允许操作系统在分配开始之前对对象执行虚拟到物理内存映射优化（例如，将操作的参数映射到物理内存中的同一行/列）。分配SIMDRAM对象的物理内存时，操作系统将基本物理地址、分配数据的总大小以及对象中每个元素的大小（由bbop\_trsp\_init提供）插入对象跟踪器。当最初分配的数据放置在CPU缓存中时，数据以水平布局开始，直到从缓存中移出。

SIMDRAM使用垂直布局将SIMDRAM对象存储在DRAM中，因为这是DRAM计算中使用的布局（§3.3）。由于垂直排列的n位元素跨越DRAM中的n个不同缓存线（每个缓存线位于不同的DRAM行中），SIMDRAM将SIMDRAM对象划分为SIMDRAMobject切片，每个切片的大小为n个缓存线。因此，DRAM中的SIMDRAM对象片包含与缓存线中的位（例如，64B缓存线中的512）一样多的元素的垂直布局位。缓存线i（0≤ 对象切片的i<；n）包含切片中存储的所有元素的位i。每当CPU请求片内的任何一个数据元素时，整个SIMDRAM对象片都会被带入LLC。类似地，每当来自SIMDRAM对象的缓存线从LLC写回DRAM（即被逐出或刷新），所有n− 1同一SIMDRAM对象片的剩余缓存线也被写回。4对象片的使用确保了正确性并简化了转置单元。

每当LLC将缓存线写回DRAM（图8中的2）时，转置单元检查对象跟踪器以查看缓存线是否属于SIMDRAM对象。如果对象跟踪器中的LLC请求未命中，则缓存线不属于任何SIMDRAM对象，并且写回请求会像在传统系统中一样转发到内存控制器。如果LLC请求命中对象跟踪器，则缓存线属于SIMDRAM对象，因此必须从水平布局转换为垂直布局。对象跟踪器命中触发两个动作。

首先，对象跟踪器向所有n个对象发出失效请求− 1同一SIMDRAM对象切片的剩余缓存线（图8中的3）。4我们扩展了LLC以支持一种特殊的失效请求类型，它将脏缓存线和未修改的缓存线发送到转置单元（不同于常规失效请求，它只是使未修改的缓存线失效）。对象跟踪器为其余缓存线发出这些无效请求，确保对象切片的所有缓存线到达转置单元，以正确执行水平到垂直转置。

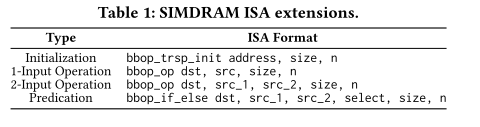
其次，写回请求被转发（图8中的4）到执行逐位转置的水平到垂直转置缓冲器。我们设计了转置缓冲器（5），使其能够在一个周期内转置水平布置的缓存线的所有位。当来自切片的其他缓存线被逐出（由于对象跟踪器的无效请求）并到达转置单元时，它们也被转发到转置缓冲区，并且它们的位被转置。每个水平布局的缓存线映射到垂直缓存线中的一组特定的位列，这是使用水平布局缓存线的物理地址确定的。一旦SIMDRAM对象片中的所有n条缓存线都被转置，存储单元就会为每个垂直布置的缓存线生成DRAM写请求，并将请求发送到内存控制器（6）。

当程序想要读取属于SIMDRAM对象的数据，并且数据不在CPU缓存中时，LLC向DRAM发出读取请求（图8中的7）。如果读取请求的地址在对象跟踪器中没有命中，则请求被转发到内存控制器，就像在传统系统中一样。如果读取请求的地址在对象跟踪器中命中，则读取请求是SIMDRAM对象的一部分，并且对象跟踪器向提取单元发送信号（8）。Fetch单元为所有垂直布置的缓存线生成读取请求，这些缓存线属于与请求数据相同的SIMDRAM对象片，并将这些请求发送到内存控制器。当对象片缓存线的请求响应到达时，提取单元将缓存线发送到垂直到水平转置缓冲器（9），该缓冲器可以在一个周期内将一条垂直布置的缓存线的所有位转置到水平布置的缓存线中。然后将水平布置的缓存线插入LLC− 1不属于原始内存请求一部分但属于同一对象片的缓存线以类似于传统预取请求的方式插入LLC[134]。

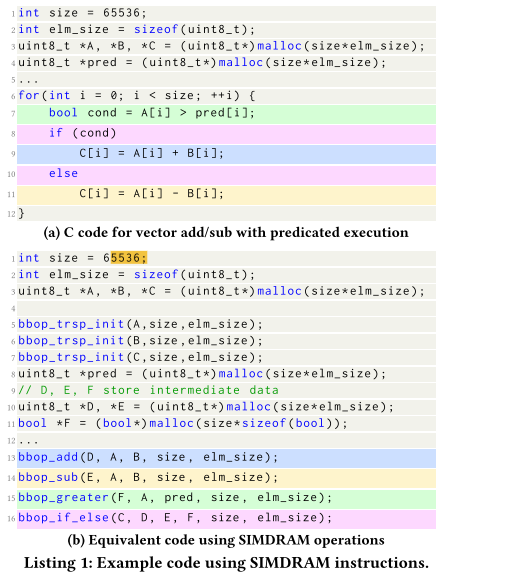
5.2 ISA扩展和编程接口

缺乏高效且富有表现力的程序员/系统界面会对SIMDRAM框架的性能和可用性产生负面影响。这将使数据转移到SIMDRAM计算的关键路径上，这将导致巨大的性能开销。为了解决这些问题并使程序员/系统能够有效地与SIMDRAM通信，我们使用专门的SIMDRAM指令扩展ISA。SIMDRAM ISA扩展的主要目标是让SIMDRAM控制单元知道（1）需要执行哪些SIMDRAM操作以及何时执行，以及（2）SIMDRAM内存对象是什么以及何时转置它们。

表1显示了SIMDRAM框架向程序员公开的CPU ISA扩展。有三种类型的指令：（1）SIMDRAM对象初始化指令，（2）执行不同SIMDRAM操作的指令，以及（3）谓词指令。我们在§5.1中讨论了bbop\_trsp\_init，这是我们唯一的SIMDRAM对象初始化指令。用于执行SIMDRAM操作的CPU ISA扩展可进一步分为两类：（1）具有一个输入操作数的操作（例如，位计数、ReLU）和（2）具有两个输入操作数的操作（例如，加法、除法、相等、最大值）。SIMDRAM使用基于数组的计算模型，这些指令中的src（即单输入操作中的src和双输入操作中的src\_1、src\_2）和dst表示源和目标数组。bbop\_op表示SIMDRAM操作的操作码，而size和n分别表示源阵列和目标阵列中的元素数以及每个阵列元素中的位数。为了启用谓词，SIMDRAM使用bbop\_if\_else指令，其中除了两个源数组和一个目标数组外，select还表示谓词数组（即谓词或掩码位）。



清单1显示了如何使用SIMDRAM的CPU ISA扩展来执行DRAM计算，其中的示例代码根据A的每个元素与第三个数组（pred）的对应元素的比较，对两个数组（A和B）执行按元素的加法或减法。清单1a显示了用于计算的原始C代码，而清单1b显示了使用SIMDRAM操作的等效代码。执行相同操作的行在C代码和SIMDRAM代码中使用相同的颜色高亮显示。C代码中的if-then-else条件在SIMDRAM中使用谓词指令执行（即清单1b第16行的bbop\_-if\_-else）。SIMDRAM将if-then-else条件视为多路复用器。因此，bbop\_if\_else将两个源数组和一个谓词数组作为输入，其中谓词用于选择应在相应索引处选择哪个源数组作为输出。为此，我们首先分配两个数组来保存加法和减法结果（即清单1b中第10行的数组D和e），然后分别使用bbop\_add和bbop\_sub（清单1b中的第13行和第14行）填充它们。然后，我们分配谓词数组（即清单1b第11行的数组F）并使用bbop\_morer（清单1b第15行）填充它。加法、减法和谓词数组构成bbop\_if\_else指令的三个输入（数组D、E、F）（清单1b中的第16行），该指令将谓词执行的结果存储到目标数组（即清单1b中的数组C）。



在这项工作中，我们假设程序员手动重写代码以使用SIMDRAM操作。在§7.3中评估实际应用程序时，我们遵循此方法。我们为SIMDRAM设想了两种编程模型。在第一种编程模型中，SIMDRAM操作被封装在用户空间库例程中，以简化可编程性。通过这种方法，程序员可以优化基于SIMDRAM的代码，以充分利用底层的内存计算机制。在第二种编程模型中，SIMDRAM操作使用编译器辅助透明地插入到应用程序的二进制文件中。由于SIMDRAM是一个类似SIMD的计算引擎，我们希望编译器至少可以通过两种方式生成SIMDRAM代码，而无需程序员干预。首先，它可以利用现代编译器[32，89]中已经存在的自动矢量化例程，通过设置等同于DRAM行的SIMD通道宽度来生成SIMDRAM代码。例如，在LLVM[74]中，可以使用“-force vector width”标志[89]定义SIMD单元的宽度。基于SIMDRAM的编译器后端可以将LLVM中间表示指令转换为bbop指令。其次，编译器可以将编译器生成的现有SIMD指令组（例如AVX2指令[31]）组合成与DRAM行大小匹配的块，然后将这些指令转换为单个SIMDRAM操作。先前的工作[2]对三维叠层PIM使用了类似的方法。我们将SIMDRAM编译器的设计留给以后的工作。

SIMDRAM指令可以通过扩展主机CPU的ISA来实现。这是可能的，因为有足够的未使用的操作码空间来支持SIMDRAM所需的额外操作码。为了说明这一点，之前的工作[91,92]表明，仅考虑x86 ISA的AVX和SSE扩展，就有389个未使用的操作代码。扩展指令集是CPU与PIM体系结构接口的常用方法[3，124]。

5.3处理有限子阵列大小

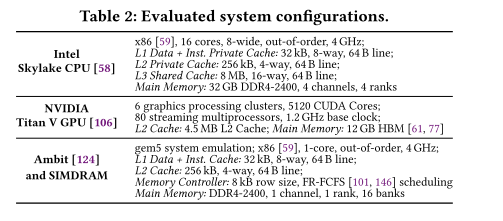
SIMDRAM对放置在同一子阵列中的数据进行操作。但是，单个子阵列仅存储数兆字节的数据。例如，具有1024行且行大小为8KB的子阵列只能存储8MB的数据。因此，SIMDRAM需要使用一种能够在DRAM内高效移动数据的机制（例如，跨DRAM组和子阵列）。SIMDRAM可以利用（1）行克隆流水线串行模式（PSM）[123]通过使用内部DRAM总线在两个存储组之间复制数据，或（2）低成本互连子阵列（LISA）[21]在同一存储组内的两个子阵列之间复制行。我们在[49]中评估了使用这两种机制的性能开销。其他DRAM内数据快速移动机制[128140]也可以增强SIMDRAM的能力。

5.4安全影响

SIMDRAM和其他类似的使用专用DRAM行执行计算的DRAM内计算机制可能会增加对行锤攻击的脆弱性[33、66、70、97、100]。我们相信，而且文献表明，RowHammer应该有稳健且可扩展的解决方案，与我们的工作正交（例如，BlockHammer[142]，第[69]段，Twow[81]，Graphene[110]）。结合SIMDRAM（或其他PIM方法）探索RowHammer预防和缓解机制需要特别关注和研究，我们将其留给未来的工作。

6方法

我们使用gem5模拟器实现SIMDRAM[16]，并将其与真正的多核CPU（Intel Skylake[58]）、真正的高端GPU（NVIDIA Titan V[106]）和使用DRAM机制的最先进处理（Ambit[124]）进行比较。在我们所有的评估中，CPU代码都经过了优化，以利用AVX-512指令[31]。表2显示了我们在评估中使用的系统参数。为了测量CPU性能，我们在sys/time.h[136]中实现了一组计时器。为了测量CPU能耗，我们使用Intel RAPL[48]。为了测量GPU性能，我们使用cudaEvents API实现了一组计时器[22]。我们捕获不包括数据初始化/传输时间的GPU内核执行时间。为了测量GPU能耗，我们使用nvml API[105]。我们报告每个CPU/GPU数据点平均运行五次，每次都有一个预热阶段，以避免冷缓存效应。我们在gem5上实施了范围，并根据[124]中报告的结果严格验证了我们的实施。我们在范围和SIMDRAM实现中使用相同的垂直数据布局，这使我们能够（1）使用其等效和/或/或基于非的实现评估范围内的所有16个SIMDRAM操作，以及（2）强调SIMDRAM框架中步骤1的好处（即，使用优化的基于主要/非主要操作的实现）。我们的综合吞吐量分析（§7.1）使用64M元素输入阵列。



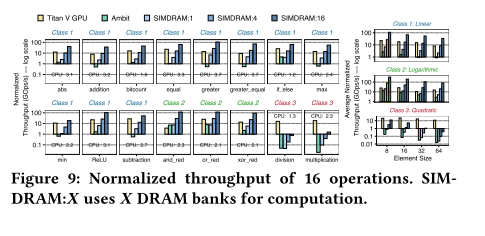
我们评估了三种不同的SIMDRAM配置，其中1（SIMDRAM:1）、4（SIMDRAM:4）和16（SIMDRAM:16）个组（在我们的评估中为16个组）具有SIMDRAM计算能力。在SIMDRAM 1-bank配置中，我们的机制利用65536（即8 kB行缓冲区大小）SIMD通道。传统的DRAM体系结构利用银行级并行（BLP）来最大化DRAM吞吐量[71–73,76,102]。内存控制器可以向同一通道上的不同组（每个周期一个）发出命令，以便组可以并行运行。在SIMDRAM中，同一渠道中的银行可以并行运行，就像传统银行一样。因此，为了实现所需的并行性，SIMDRAM不需要更多的修改。因此，通过利用SIMDRAM配置中的BLP（即16列组配置中的可用SIMD通道数量为16×65536），可用SIMD通道的数量（即SIMDRAM的计算能力）增加。

7评价

通过评估，我们展示了SIMDRAM框架的优势：（1）SIMDRAM在各种操作中的吞吐量和能耗；（2） SIMDRAM在实际应用中的性能优势；（3）与使用高速缓存体系结构的密切相关处理相比，SIMDRAM的性能和能源优势[35]。最后，我们评估了SIMDRAM的面积成本。论文的最终版本[49]演示了（1）SIMDRAM的三行激活（TRA）操作如何比以前的工作[6,9]使用的五行激活（QRA）操作更具可扩展性和容错性；（2）SIMDRAM的巨大性能优势，即使存在最坏情况下的数据移动和数据转置开销。

7.1吞吐量分析

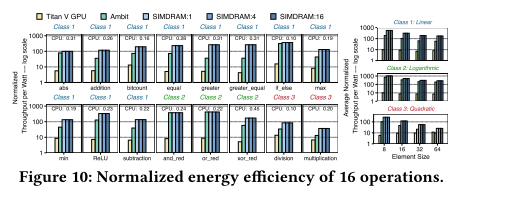
图9（左）显示了所有16个SIMDRAM操作（§4.4）的归一化吞吐量，与CPU、GPU和Ambit（归一化为多核CPU吞吐量）上的吞吐量相比，元素大小为32位。我们在每个图中提供基准CPU的绝对吞吐量（以GOps/s为单位）。我们根据操作延迟与元素大小n的比例对每个操作进行分类。第1类、第2类和第3类操作分别以n线性、对数和二次缩放。图9（右）显示了同一类的所有操作的平均吞吐量如何相对于元素大小进行缩放。我们评估8、16、32、64位的元素大小。我们将这个数字标准化为CPU上的平均吞吐量。



我们从图9中进行了四次观察。首先，我们观察到SIMDRAM优于三种最先进的基线系统，即CPU/GPU/Ambit。与CPU/GPU相比，SIMDRAM的吞吐量分别为CPU/GPU的5.5×/0.4×、22.0×/1.5×、88.0×/5.8×1、4和16个银行的所有16个SIMDRAM操作的平均吞吐量。为了确保公平性，我们仅将在评估中使用单个DRAM库的Ambit与SIMDRAM:1进行了比较。7我们的评估表明，SIMDRAM:1的性能比Ambit高出2.0倍，这是所有16个SIMDRAM操作的平均值。其次，当我们对所有线性和对数运算使用四个以上的DRAM组时，SIMDRAM的性能优于GPU基线。SIMDRAM:16平均为所有线性（对数）操作提供5.7×（9.3×）的GPU吞吐量。SIMDRAM:16的吞吐量在所有线性（对数）操作中的平均值分别为CPU和Ambit的83倍（189倍）和45.2倍（19.9倍）。第三，我们观察到，多核CPU基线和GPU仅在除法和乘法运算方面优于SIMDRAM:1、SIMDRAM:4和SIMDRAM:16。这是由于这两个操作的位串行实现的二次性。第四，正如预期的那样，我们观察到随着元素大小的增加，所有操作的吞吐量都会下降，因为每个操作的延迟都会随着元素大小的增加而增加。我们得出结论，SIMDRAM在广泛的操作范围内显著优于所有三个最先进的基线。

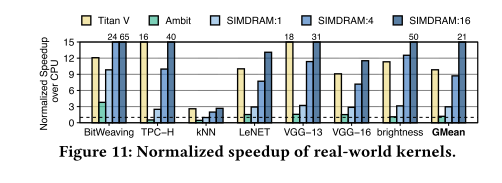
**7.2能量分析**

我们使用仙人掌[96]来评估SIMDRAM的能量消耗。先前的工作[124]表明，每增加一个同时行激活，能耗就会增加22%。我们利用这一观察结果来评估SIMDRAM的能耗，这需要TRAs。图10将SIMDRAM的能效（每瓦特吞吐量）与GPU和范围基线进行了比较，标准化为CPU基线。我们在每个图中提供基准CPU每瓦特的绝对吞吐量。我们做了四次六点的服务。首先，SIMDRAM显著提高了所有三条基线上所有操作的能源效率。SIMDRAM的能量效率分别是CPU、GPU和Ambit的257倍、31倍和2.6倍，在所有16个操作中的平均值。SIMDRAM的节能直接源于（1）避免了从内存加载/存储数据的昂贵片外往返，（2）利用内存设备内丰富的内存带宽，减少执行时间，以及（3）通过优化基于多数的操作实现，减少计算给定操作所需的TRA数量。其次，与我们关于吞吐量的结果类似（§7.1），SIMDRAM的能量效率随着元件尺寸的增加而降低。然而，CPU或GPU的能源效率并不高。这是因为（1）对于所有SIMDRAM操作，TRA的数量随着元素大小的增加而增加；（2）CPU和GPU可以充分利用其更宽的运算单元和更大（即32位和64位）的元素大小。第三，尽管SIMDRAM乘法和除法运算在元素大小方面的扩展性很差，但与CPU和GPU基线相比，这些运算的SIMDRAM实现明显更节能，这使得SIMDRAM成为乘法和除法运算的竞争对手。第四，由于SIMDRAM的吞吐量和功耗都随存储组的数量成比例增加，因此SIMDRAM 1、4和16存储组配置的每瓦吞吐量是相同的。我们得出结论，SIMDRAM比所有三种最先进的基线在广泛的操作中都更节能。



7.3对真实内核的影响

我们使用一组内核来评估SIMDRAM，这些内核代表来自不同领域的选定重要现实世界应用程序的行为。评估的核来自数据库（TPC-H查询1[137]、比特编织[88]）、卷积神经网络（LeNET-5[75]、VGG-13[131]、VGG-16[131]）、分类算法（k近邻[83]）和图像处理（亮度[44]）。这些内核依赖于我们在§7.1中评估的许多基本操作。我们在[49]中简要介绍了每个内核及其使用的SIMDRAM操作。图11显示了SIMDRAM的性能和每个内核的基线配置，标准化为多核CPU的性能。我们有四个观察。首先，SIMDRAM:16大大优于CPU和GPU基线，在所有七个内核中，CPU和GPU的平均性能分别为21倍和2.1倍。SIMDRAM的最大性能分别为CPU和GPU的65倍和5.4倍（对于这两种情况下的比特编织内核）。类似地，SIMDRAM:1在所有七个内核上平均提供了2.5倍的Ambit性能（它也使用单个库进行内存计算），TPC-H内核的最大性能为4.8倍。第二，即使只有一个DRAM库，SIMDRAM的性能也始终优于CPU基线，在所有内核中平均提供2.9倍的CPU性能。第三，SIMDRAM:4分别为比特编织和亮度内核提供2倍和1.1倍的GPU基线性能。第四，尽管GPU的乘法吞吐量高于SIMDRAM（§7.1），但SIMDRAM:16即使在严重依赖乘法的内核上也优于GPU基线[49]（例如，kNN和TPC-H内核分别为1.03倍和2.5倍）。这种加速是利用SIMDRAM中的高内存带宽来避免GPU中因此类内核中生成大量中间数据而导致的内存瓶颈的直接结果。我们得出结论，SIMDRAM是一种有效的基板，可以加速许多常用的实际应用。



7.4与DualityCache的比较

我们将SIMDRAM与DualityCache[35]进行比较，DualityCache是一种使用缓存架构的密切相关的处理。DualityCache是一种缓存内计算框架，它使用添加到SRAM外围电路中的离散逻辑元件（如逻辑门、锁存器、多路复用器）执行计算。与内存计算方法（如SIMDRAM）相比，缓存内计算方法（如DualityCache）需要先将数据放入缓存，这需要额外的数据移动（如果应用程序的工作集不适合缓存，则需要更大的移动）。

图12（上图）比较了SIMDRAM和DualityCache支持的操作子集（即加法、减法、乘法和除法）与DualityCache[35]的延迟。在这个实验中，我们研究了三种不同的配置。首先，DualityCache:Ideal在缓存中拥有DualityCache所需的所有数据。因此，DualityCache:Ideal的结果不包括将数据从DRAM移动到缓存的开销，这使得它成为一种不切实际的配置，需要数据已经驻留并适合缓存。其次，DualityCache:Reality包括从DRAM到缓存的数据移动开销。两种DualityCache配置都在45MB的输入阵列上计算。第三，辛德拉姆：16。对于所有三种配置，我们使用与原始DualityCache工作相同的缓存大小（35MB）[35]，以提供公平的比较。如图所示，当实际考虑数据移动时，SIMDRAM的性能大大优于DualityCache。SIMDRAM:16在所有四种运算中都优于DualityCache:Reality（在所有元素大小中，加法、减法、乘法和除法的平均值分别为52.9倍、52.4倍、1.8倍和2.1倍）。与DualityCache相比，SIMDRAM的性能改进的区域开销要低得多。DualityCache（包括其外围设备、转置存储单元、控制器、未命中状态保持寄存器和交叉开关网络）在高端CPU中的面积开销为3.5%，而SIMDRAM的面积开销仅为0.2%（§7.5）。因此，与DualityCache相比，SIMDRAM实际上可以在给定区域中安装数量明显更多的SIMD车道。因此，SIMDRAM的单位面积性能改进将比我们在图12中观察到的要大得多。我们得出结论，当DRAM缓存数据移动时，SIMDRAM在更低的区域开销上实现了比DualyCache更高的性能。

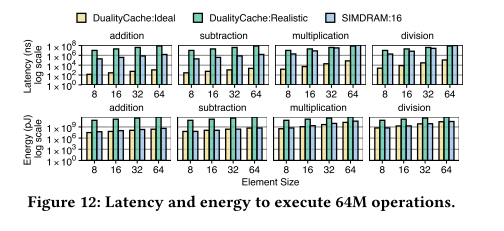


图12（底部）显示了在执行64M加法、减法、乘法和除法运算时，DualityCache:reality、DualityCache:Ideal和SIMDRAM:16的能耗。我们有两个观察。首先，与DualityCache:Ideal相比，SIMDRAM:16将平均能耗提高了60%。这是因为，虽然在DRAM中执行计算的每比特能量（13.3 nJ/比特[95，139]）小于在缓存中执行计算的每比特能量（60.1 nJ/比特[28]），但每个操作的DualityCache实现需要的迭代次数少于其等效的SIMDRAM实现。其次，SIMDRAM:16比DualityCache:Reality平均能耗降低了600倍，因为DualityCache:Reality需要从DRAM加载所有输入数据，导致了高能耗开销（DRAM访问消耗的能耗是DualityCache操作每位能耗的650倍[28，35]）。相反，SIMDRAM操作DRAM中已经存在的数据，消除了任何数据移动开销。我们的结论是，当实际考虑缓存到DRAM的数据移动时，SIMDRAM比DualityCache效率更高。

7.5区域开销

我们使用CACTI[96]评估使用22nm技术节点的SIMDRAM设计中主要组件的面积开销。SIMDRAM不会对DRAM电路进行任何修改，但Ambit提出的修改除外，其面积开销为<；1%在商品DRAM芯片中[124]。因此，SIMDRAM在范围上的区域开销只有内存控制器中的两种结构：控制单元和换位单元。控制单元区域开销。SIMDRAM控制单元中的主要部件是（1）bbop FIFO、（2）µ程序暂存器、（3）µOp存储器。我们将bbop FIFO和µ程序草稿行的大小分别调整为2 kB。bbop FIFO的大小足以容纳多达1024条bbop指令，我们观察到这对于我们的实际应用来说已经足够了。µProgram Scratchpad的大小足以存储我们在本文中评估的所有16个SIMDRAM操作的µPrograms（16µPrograms×128 B max perµProgram）。我们对µOp内存使用128 B草稿行。2我们估计SIMDRAM控制单元面积为0.04mm2。换位单位面积开销。转置单元中的主要组件是（1）对象跟踪器和（2）两个转置缓冲器。对于对象跟踪器，我们使用一个8 kB的完全关联缓存，其缓存线大小为64位。这足以在对象跟踪器中存储1024个条目，其中每个条目保存SIMDRAM对象的基本物理地址（19位）、分配数据的总大小（32位）和对象中每个元素的大小（6位）。每个转置缓冲区为4KB，可转置到64位SIMDRAM对象（64位×64B）。我们估计换位单位面积为0.06mm2。考虑到控制和换位单元的面积，与Intel Xeon E5-2697 v3 CPU的芯片面积相比，SIMDRAM的面积开销仅为0.2%[35]。我们得出结论，SIMDRAM具有较低的面积成本。

8相关工作

据我们所知，SIMDRAM是第一个支持DRAM计算的端到端框架，对用户来说是灵活和透明的。我们将SIMDRAM与最先进的内存处理设计进行对比，突出SIMDRAM的主要贡献。在3D堆叠内存中处理近内存（PnM）。许多最近的工作（例如，[3,4,17-19,25,27,29,30,38,39,42,43,47,53,54,64,68,84,90,103,107,114,118-120,144]）探索将逻辑直接添加到三维堆叠存储器的逻辑层（例如，高带宽存储器[61,77]，混合存储立方体[55]）。SIMDRAM的实现相当简单，并且依赖于对商品DRAM芯片的最小修改。使用内存（PuM）进行处理。先前的工作提出了一些机制，其中内存阵列本身执行各种批量操作[6、9、24、26、85、86、122–127、135、141]。SIMDRAM支持范围更广的操作（与[6,9,24,85,86,123,124,141]相比），计算成本更低（与[124,141]相比），区域开销更低（与[86]相比），执行更可靠（与[6,9]相比）。在缓存中处理。最近的工作[1,28,35]提出了利用SRAM位线结构在缓存中执行位串行计算的SRAM加速器。SIMDRAM与这些方法有相似之处，但通过利用高密度和低成本的RAM技术，每比特的成本显著降低。与§7.4中的DualityCache[35]相比，我们展示了SIMDRAM的巨大性能和能源优势。PIM框架。以前很少有工作能够解决为PIM提供端到端支持的挑战。我们描述了这些框架及其在DRAM计算中的局限性。DualityCache[35]是一种用于缓存内计算的端到端框架。DualityCache利用CUDA/OpenAcc编程语言[22109]为缓存内机制生成代码，该机制以单指令多线程（SIMT）方式执行一组固定的操作。与SIMDRAM一样，DualityCache通过SRAM阵列的位线以垂直布局存储数据。它将每个位线视为一个独立的执行线程，并利用纵横制网络允许跨位线的线程间通信。尽管有好处，但在DRAM中使用DualityCache并不简单，原因有二。首先，使用SRAM中DualityCache所使用的纵横制网络扩展DRAM子阵列，以允许线程间通信，这将在DRAM中施加禁止的区域开销（9×DRAM子阵列区域）。其次，作为一种缓存内计算解决方案，DualityCache没有考虑到内存内计算的局限性，即破坏输入数据的DRAM操作、能够处理DRAM的有限内存行数以及避免昂贵的内存内拷贝的需要。我们已经证明，当实际考虑DRAM到缓存的数据移动时，SIMDRAM在较低的区域开销下实现了比DualityCache更高的性能（§7.4）。之前的两项工作提出了针对ReRAM设备的框架。Hyper AP[143]是一个使用ReRAM进行关联处理的框架。由于Hyper-AP以关联处理为目标，因此所提出的框架与SIMDRAM有根本不同。IMP[34]是一个用于现场ReRAM操作的框架。与DualityCache类似，IMP框架依赖于ReRAM阵列的特定结构（例如模拟-数字/数字-模拟转换器）来执行计算，因此不适用于执行批量位操作的DRAM衬底。此外，DualityCache、Hyper-AP和IMP都有一个严格的ISA，只支持有限的一组内存内操作（DualityCache支持16个内存内操作，而Hyper-AP和IMP都支持12个）。相比之下，SIMDRAM是PuM的第一个灵活的框架，它提供了一种方法，允许根据需要在内存中集成和计算新的操作。总之，SIMDRAM填补了以使用DRAM进行处理为目标的灵活端到端框架的空白。

9结论

我们介绍了SIMDRAM，这是一种使用DRAM框架的大规模并行通用处理，它（1）能够以SIMD方式在DRAM中高效地执行各种操作，（2）提供了一种灵活的机制来支持执行任意用户定义的操作。SIMDRAM引入了一个新的三步框架，使基于MAJ/NOT的高效DRAM实现能够实现不同类别（例如，算术、关系、预测）的复杂操作，并适用于广泛的实际应用。我们为SIMDRAM框架设计了硬件和ISA支持，以（1）解决关键的系统集成挑战，（2）允许程序员在不改变硬件的情况下使用新的SIMDRAM操作。我们通过实验证明，与最先进的CPU、GPU和PuM系统相比，SIMDRAM具有显著的性能和能源优势。我们希望未来的工作建立在我们的框架之上，以进一步简化采用，并提高使用DRAM体系结构和应用程序进行处理的性能和效率。