计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2018.20170842
55(2):289-304,2018

CPU 和 DRAM 加速任务划分方法:大数据处理中 Hash Joins 的加速实例

吴林阳　罗　蓉　郭雪婷　郭　崎
(中国科学院计算技术研究所　北京　100190)
(wulinyang@ict.ac.cn)

Partitioning Acceleration Between CPU and DRAM: A Case Study on Accelerating Hash Joins in the Big Data Era

Wu Linyang, Luo Rong, Guo Xueting, and Guo Qi
(*Institute of Computing Technology, Chinese Academy of Sciences, Beijing* 100190)

Abstract　Hardware acceleration has been very effective in improving energy efficiency of existing computer systems. As traditional hardware accelerator designs (e.g. GPU, FPGA and customized accelerators) remain decoupled from main memory systems, reducing the energy cost of data movement remains a challenging problem, especially in the big data era. The emergence of near-data processing enables acceleration within the 3D-stacked DRAM to greatly reduce the data movement cost. However, due to the stringent area, power and thermal constraints on the 3D-stacked DRAM, it is nearly impossible to integrate all computation units required for a sufficiently complex functionality into the DRAM. Therefore, there is a need to design the memory side accelerator with this partitioning between CPU and accelerator in mind. In this paper, we describe our experience with partitioning the acceleration of hash joins, a key functionality for databases and big data systems, using a data-movement driven approach on a hybrid system, containing both memory-side customized accelerators and processor-side SIMD units. The memory-side accelerators are designed for accelerating execution phases that are bounded by data movements, while the processor-side SIMD units are employed for accelerating execution phases with negligible data movement cost. Experimental results show that the hybrid accelerated system improves energy efficiency up to 47.52x and 19.81x, compared with the Intel Has well and Xeon Phi processor, respectively. Moreover, our data-movement driven design approach can be easily extended to guide the design decisions of accelerating other emerging applications.

Key words　3D-stacked DRAM; accelerator; big data; hash joins; optimized version of radix joins algorithm (PRO); hash partition accelerator (HPA)

摘　要　硬件加速器能够有效地提高当前计算机系统的能效.然而,传统的硬件加速器(如 GPU,FPGA 和定制的加速器)和内存是相互分离的,加速器和内存之间的数据移动难以避免,这使得如何降低加速器和内存之间数据移动的开销成为极具挑战性的问题.随着靠近数据的处理技术(near-data

收稿日期:2017-11-07;**修回日期**:2017-12-20

基金项目:国家重点研发计划项目(2017YFB1003101);国家自然科学基金项目(61472396,61432016,61473275,61522211,61532016,61521092,61502446,61672491,61602441,61602446,61732002,61702478);北京市科技计划项目(Z151100000915072);中科院 STS 计划项目;国家"九七三"重点基础研究发展计划基金项目(2015CB358800)

This work was supported by the National Key Research and Development Program of China (2017YFB1003101), the National Natural Science Foundation of China (61472396, 61432016, 61473275, 61522211, 61532016, 61521092, 61502446, 61672491, 61602441, 61602446, 61732002, 61702478), Beijing Science and Technology Project (Z151100000915072), the STS Project of Chinese Academy of Sciences, and the National Basic Research Program of China (973 Program) (2015CB358800).

processing)和 3D 堆叠 DRAM 的出现,我们能够将硬件加速器集成到 3D 堆叠 DRAM 中,使得数据移动的开销大大降低.然而,由于 3D 堆叠 DRAM 对面积、功耗和散热具有严格的限制,所以不可能将一个功能复杂的硬件加速器完整地集成到 DRAM 中.因此,在设计内存端的硬件加速器时,应该考虑将加速任务在 CPU 和加速器之间合理地进行划分.以加速大数据系统中的一个关键操作 hash joins 为例子,阐述了 CPU 和内存端加速任务划分的设计思想.以减少数据移动为出发点,设计了一个包含内存端定制加速器和处理器端 SIMD 加速单元的混合加速系统,并对应用进行分析,将加速任务划分到不同的加速器.其中,内存端的加速器用于加速数据移动受限的执行阶段,而处理器端 SIMD 加速单元则用于加速数据移动开销较低成本的执行阶段.实验结果表明:与英特尔的 Haswell 处理器和 Xeon Phi 相比,设计的混合加速系统的能效分别提升了 47.52 倍和 19.81 倍.此外,提出的以数据移动为驱动的方法很容易扩展于指导其他应用的加速设计.

关键词 3D 堆叠内存;加速器;大数据;hash joins; radix joins 算法的优化版本; hash 分区加速器

中图法分类号 TP302

在设计现代计算机系统时首先要考虑的因素是能效.为了提高系统能效,诸如 FPGA,GPU 和定制加速器等硬件加速器已被广泛应用于工业领域.使用硬件加速器的典型案例有:微软组建 FPGA 服务器用于加速大规模的数据中心服务[1];亚马逊于 2010 年开始部署 GPU 云服务器;惠普实验室为 Memcached 定制加速器[2].通常,硬件加速器集成在处理器芯片中(例如 SIMD 单元、卷积引擎[3]、IBM 的 PowerEN[4]),或部署为一个独立的芯片(例如 GPU 显卡、FPGA 板卡和 Xeon Phi 协处理器),通过 PCI-E 与主处理器进行通信.

随着靠近数据的处理(near-data processing)技术的出现,将硬件加速器集成到 DRAM 堆栈中以降低数据移动的成本成为一种新的系统设计思路.其基本思想是利用 3D 堆叠技术,将一些包含加速器的逻辑 die 和多个 DRAM die 垂直集成到一个芯片中[5-7].然而,由于 3D 堆叠 DRAM 在面积、功耗、散热和制造等方面的限制,能够集成到 DRAM 中的加速器的数量和类型是有限的.因此,给定一个需要加速的目标应用,确定其中哪些部分最适合在 DRAM 中加速是至关重要的.

理想的加速目标应该是内存带宽受限的,因为 3D 堆叠 DRAM 具有比现有系统高一个数量级的高内存带宽.一个最典型的例子是数据重组操作,它是许多应用领域的关键组成部分,如高性能计算、信号处理和生物信息学.数据重组只包含 CPU 和 DRAM 之间的往返数据移动,它不消耗浮点运算周期.因此,在 DRAM 中加速数据重组是很自然的想法,文献[8]中论证了这项工作.但是对于大多数真实应用程序来说,它们混合了计算和数据移动,情况更为复杂.

在本文中,我们进行了一个 hash joins 的案例研究,探索如何将它在 CPU 和 DRAM 之间进行合适的加速任务划分.选择 hash joins 有 3 个原因:1)hash joins 不仅是传统数据库系统的基本操作,还是诸如 Spark[9]等主流大数据系统的关键操作;2)很多文献都对 hash joins 做了充分研究[10-15],因此加速的 baseline 已经高度优化;3)最先进的 hash joins: optimized version of radix join algorithm (PRO)[11,15]是带宽资源受限的,它可能会从内存加速中受益.我们对 PRO 以及 PARSEC benchmark suite 中具有代表性的多线程应用程序分别进行了带宽分析,如图 1 所示.平均来看,PRO 对带宽的需求比使用 8 线程的 PARSEC benchmark 高 3.6 倍.与 PARSEC benchmark suite 中带宽资源受限最严重的 streamcluster 相比,在 2,4 和 8 线程下,PRO 仍多消耗 143%,117%和 18%的带宽资源.

为了研究在 PRO 中内存访问对总能耗的贡献,我们对 PRO 的 3 个不同阶段:partition,build 和 probe,进行了详细的能耗分析.虽然所有阶段的内存带宽都趋于饱和,但各个阶段中计算所消耗的能量差异是很大的.在 partition 阶段,计算消耗的能量还不到该阶段总能量的 50%;而在 build 和 probe 阶段,计算消耗的能量占总能量的 85%以上.换句话说,数据移动消耗的能量是 partition 阶段能耗的主要部分.主要的原因是在 partition 阶段有许多高代价的非规则内存访问,而其他 2 个阶段只有常规的内存访问.因此,仅在 DRAM 端加速 partition 阶段,在 CPU 端加速其余阶段是有意义的.为了加速 partition 阶段,我们构建了适合于 3D 堆叠结构的 hash 分区加速器(hash partition accelerator, HPA).为了加快 build 和 probe 阶段,我们采用传统的 SIMD 加速单元来提高计算效率.

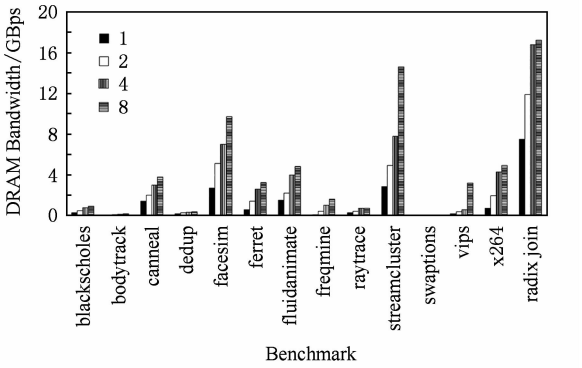


Fig. 1 Memory bandwidth of PARSEC benchmarks and Radix hash join

图 1 PARSEC benchmarks 和 Radix hash join 的内存带宽分析

本文的创新点和贡献可以概括如下:

1) 以数据移动为驱动的加速任务划分方法. 我们提出了一种以数据移动为驱动的方法来划分 CPU 和 DRAM 之间的加速任务,使得我们能够在满足 3D 堆叠 DRAM 面积、功耗、发热的严格限制条件下,充分挖掘其加速性能.

2) 最先进的 hash joins 的详细分析. 我们对最先进的 hash joins 算法进行了详细的性能和能耗分析. 性能分析结果显示,最先进的 hash joins(PRO)本质上是内存受限的. 我们的能耗分析结果显示,在 partition 阶段,总能量的 50%以上被数据移动和流水线阻塞所消耗,这可以通过在内存端进行加速得到显著缓解. 在 build 和 probe 阶段,只有大约 15%的能量被数据移动和流水线阻塞所消耗,这仍然可以利用现有的 CPU 端加速单元(如 SIMD 单元)进行加速.

3) hash 分区加速器的设计. 我们设计了一个 hash 分区加速器(HPA),并将其集成到 3D 堆叠 DRAM 的逻辑层中. HPA 包含 3 个主要的部件:hash unit, histogram unit 和 shuffle unit. 此外,我们探索了设计空间,使得我们能够在多种 3D 堆叠 DRAM 配置中找到 HPA 的最佳设计方案. 实验结果表明,与现有系统相比,HPA 提高了 3 个数量级的能效.

4) 针对 hash joins 的混合加速系统. 我们建立了一个混合加速系统,该系统包含内存端定制的加速器和处理器端 SIMD 加速器,这个系统能提高 hash joins 的整体能效. 实验结果表明,平均而言,该系统在英特尔 Haswell 和 Xeon Phi 处理器上分别提高了 48 倍和 20 倍的能效.

1 背景介绍

1.1 hash joins

join 操作是将 2 个或多个关系表中的元组组合起来的基本数据库操作,hash joins 被认为是内存数据库管理系统中最流行的 join 算法之一. 此外,hash joins 也被广泛应用在 Spark 等大数据平台中. 现有的 hash joins 算法大致可以分为 2 类,即不考虑硬件结构的 hash joins 和考虑硬件结构的 hash joins[11].

1) 不考虑硬件结构的 join. 该类 join 算法不关心运行算法的底层硬件结构. 简单 hash joins 算法(SHJ)就是一种典型的不考虑硬件结构 join 算法,它包括 2 个阶段:build 和 probe. 在 build 阶段,扫描较小的关系表 R,建立一个 hash 表;然后在 probe 阶段,遍历 S 关系表中的每个元组,在 hash 表中找到与之匹配的 R 关系表中的元组. 文献[10]提出了 SHJ 的并行版本:no partition algorithm (NPO),用以挖掘多核架构的并行性. NPO 的关键是把输入关系表 R 和 S 分成相等大小的部分,每个部分被分配到一个工作线程进行处理. 然而,由于 NPO 的 hash 过程中存在大量的随机内存访问,可能导致大量的 cache miss,使得能效下降. 这就是不考虑硬件结构的 join 算法的缺点.

2) 考虑硬件结构的 join. 为了更好地利用高速缓存的层次结构,考虑硬件结构的 join 算法被大量研究. 文献[16]提出了一种新的可以充分利用高速缓存的磁盘连接算法 DBCC-join. 它将执行过程分成了 2 个阶段:JPIPT 构建阶段和结果输出阶段,并在 JPIPT 构建阶段构建索引时充分考虑了高速缓存的特性. 此外,为了充分利用高速缓存和 TLB 的局部性,radix partition 被广泛采用[17]. PRO[11,15] 是目前最先进的一种 hash joins 算法. PRO 算法分为 3 个阶段:partition,build 和 probe.

① partition. 图 2 给出了 PRO 的 partition 阶段. 该阶段可以进一步分为 4 个阶段:local histogram, prefix sum, output addressing, data shuffling. 其中,local histogram 和 data shuffling 阶段占总执行时间的 99%以上. 在 local histogram 阶段,扫描分块的输入关系表(R 或 S)*rel*,建立一个 histogram 数组 *my_hist*;在 data shuffling 阶段,*rel* 中的元组根据存储在 *dst* 数组中的位置被复制到目标分组 *tmp*,完成对关系表 *rel* 的划分.

```
① //Step 1: Local Histogram
② for (i=0; i<num_tuples; i++) {
③     uint32_t idx=HASH(rel[i].key);
④     my_hist[idx]++;
⑤ }
⑥ //Step 2: Prefix Sum
⑦ //Step 3: Output Addressing
⑧ //Step 4: Data Shuffling
⑨ for (i=0; i<num_tuples; i++) {
⑩     uint32_t idx=HASH(rel[i].key);
⑪     tmp[dst[idx]]=rel[i];
⑫     ++dst[idx];
⑬ }
```

Fig. 2 The partition phase of PRO

图 2 PRO 的 partition 阶段

② build. 图 3 给出了 PRO 的 build 阶段. 在这一阶段中扫描较小的关系表 R, 建立 *next* 和 *bucket* 数组. *next* 数组用于追踪被 hash 到同一个簇中的上一个 R 元组; 而 *bucket* 数组用于追踪被 hash 到当前簇的最后一个 R 元组.

```
① //R is an input relation
② for (i=0; i<R->num_tuples;) {
③     uint32_t idx=HASH(R->tuples[i].key);
④     next[i]=bucket[idx];
⑤     bucket[idx]=++i;
⑥ }
```

Fig. 3 The build phase of PRO

图 3 PRO 的 build 阶段

③ probe. 图 4 给出了 PRO 的 probe 阶段. 关系表 S 中的每一个元组 hash 成一个 *bucket* 数组的索引. 然后检索 *bucket* 和 *next*, 尝试找到与该 S 元组匹配的 R 元组.

```
① //R and S are input relations
② for (i=0; i<S->num_tuples; i++) {
③     uint32_t idx=HASH(S->tuples[i].key);
④     hit=bucket[idx];
⑤     for (; hit>0; hit=next[hit-1])
⑥         if (s->tuples[i].key==R->tuples[hit-1].key)
⑦             Output a match.
⑧ }
```

Fig. 4 The probe phase of PRO

图 4 PRO 的 probe 阶段

1.2 内存端加速

作为减少计算单元与内存之间通信开销的一种有前景的技术, 内存中处理(PIM)在过去几十年受到了广泛关注. PIM 的基本思想是将简单的处理逻辑集成到传统的 DRAM 或嵌入式 DRAM(eDRAM)阵列中[18]. 比如, 可以将 SIMD 加速单元集成到 DRAM die 中[19-20]; 还可以将可重构计算逻辑集成到 DRAM die 中, 构建 ActivePage[21]. 然而, PIM 技术的实现面临一个主要障碍: 内存加工工艺与逻辑单元加工工艺有很大的不同. 用同样的工艺制造出单个芯片不能充分发挥出 PIM 的优势.

垂直 3D die 堆叠技术允许将多个存储器 die 直接叠加在处理器 die 上, 从而提高内存带宽[22-25]. 这些 die 与短、快速、密集的 TSV 集成在一起, 提供了非常高的内部带宽. 虽然发热问题是异构 die 堆叠的一个主要问题, 但是仍然出现了一些 3D 堆叠内存产品, 比如三星、Tezzaron、Micron 生产的 3D 堆叠内存. 以 Micro 的 Hybrid Memory Cube(HMC)为例, 它在逻辑层之上堆叠多个 DRAM die, 逻辑层中包含能够访问垂直 bank 的 vault controller, 以及与其他处理单元和 stack 通信的 link controller. 通过在 HMC 的逻辑层直接集成加速器来获得极高的内部带宽, 可以避免在计算单元和内存之间进行来回的数据搬运.

虽然有一些关于集成加速单元的文献, 如高能效处理器核[26]、可编程 GPU[5]、SIMD 单元[7]和定制的加速器[6,27], 但当给定一个应用程序时, CPU 和 DRAM 之间的执行任务的划分仍旧是一个值得研究的问题, 因为 CPU 在这些系统中仍然是不可或缺的.

2 hash joins 分析

这一部分, 我们在商用多核处理器上对 hash joins 操作的访存和能耗进行了详尽的分析.

2.1 分析方法

我们对最先进的 hash joins 算法 PRO 进行了详细的性能分析和能耗分析, 该算法针对现代多核系统进行了特殊的优化. 我们建立了如表 1 所示的多核评测系统. 为了收集如访存带宽、cache miss 和功耗等执行细节, 我们使用了 Intel 的 Performance Counter Monitor(PCM)2.8 提供的编程接口. 在该评测平台上, 我们可以收集 L1/L2 缓存未命中率、内存控制器的读/写字节数等关键信息. 此外, 通过查看 Intel 的 RAPL(Runtime Average Power Limit)[28] 提供的计数器, 我们也可以获得处理器和 DRAM 的能耗信息.

Table 1 The Hardware System Used for Evaluation

表 1 用作评估的硬件系统

Hardware	Intel Haswell System
CPU	Intel i7-4770k 3.5 GHz
Core/Thread	4/8
L1 cache	32 KB data cache /32 KB instruction cache
L2 cache	256 KB
L3 cache	8 MB
Memory	32 GB DDR3 1 333 MHz

2.2 访存分析

虽然 hash joins 算法的性能瓶颈已经得到了很好的研究,但是在分析 hash joins 访存行为细节方面仍然缺乏研究工作.

如图 1 所示,虽然 PRO 整体的内存带宽要求高于传统的多线程应用程序(比如 PARSEC benchmark),但我们将进一步研究 PRO 是否受限于可用的内存带宽.如表 2 所示,根据内存设备的物理特性计算,系统的理论最大带宽为 21.2 GBps,但是在实际应用中理论带宽是难以达到的.对于实际应用来说,一种充分挖掘内存带宽的方法是采用流式(stream)访存.在高性能计算领域中,STREAM benchmark[29] 被广泛用于测量系统的可持续内存带宽.使用 STREAM benchmark 对我们的评测平台进行测试,发现平台的可持续内存带宽约为 18.49 GBps.在运行 8 线程的 PRO 中,我们测得实际内存带宽为 17.24 GBps,约为可持续带宽的 93.2%.换句话说,PRO 几乎使评测平台的可持续内存带宽达到饱和.

Table 2 Bandwidth Measurement

表 2 带宽评估

Theoretical Peak/GBps	Sustainable Bandwidth/GBps	PRO /GBps	Utilization /%
21.2	18.49	17.24	93.2

PRO 包含 partition,build 和 probe 这 3 个主要执行阶段,我们进一步测量每个阶段的访存带宽.如图 5(a)所示,在 8 个线程下,partition 阶段的内存带宽为 17.2 GBps,其余 2 个阶段的内存带宽约为 15.6 GBps.这一观察结果表明,在 PRO 的整个执行过程中,内存带宽利用率始终很高.但是,如表 3 所示,partition 阶段的最后一级 cache 未命中率远远高于其他 2 个阶段.从内存带宽利用率和 cache miss 两个不同角度观察,我们发现导致 PRO 不同阶段都有如此高的带宽利用率的根本原因是不同的.由于 partition 阶段中 data shuffling 过程(参见图 2)存在大量随机访存行为,导致大量 cache miss,从而使得带宽利用率看起来很高,但实际的数据移动效率却较低;而 build 和 probe 阶段是流式访存,所以带宽利用率和数据移动效率都较高.

我们进一步研究 partition 阶段,其中 local histogram(LH)和 data shuffling(DS)的执行时间占总执行时间的 99%以上.如图 5(b)显示,当使用 4 个线程运行时,LH 阶段受到可用内存带宽的限制,原因是关系表的顺序访存容易使内存带宽达到饱和.对于 DS 的阶段,在运行 8 个线程时内存带宽达到峰值.我们可以看到,DS 阶段内存带宽低于 LH 阶段.究其原因,是由于键(key)在各自位置上的分布性,在 DS 阶段存在很多不规则的访存.

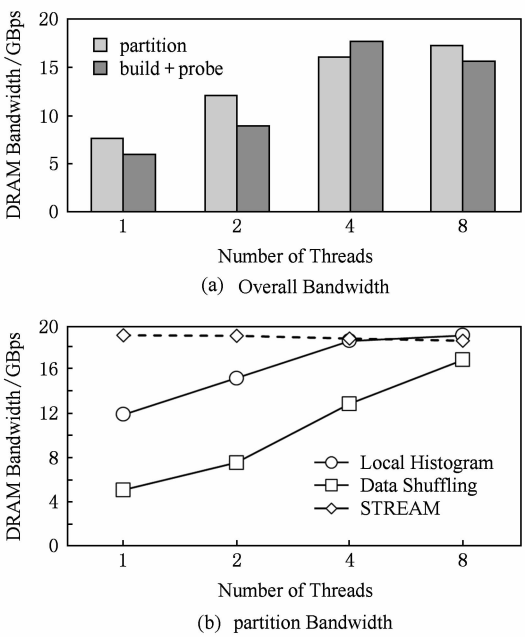


Fig. 5 Detailed bandwidth analysis of PRO

图 5 PRO 的详细带宽分析

Table 3 L3 Cache Miss Ratio of PRO

表 3 PRO 的 cache miss 率

# Threads	partition	build+probe
1	0.986	0.652
2	0.935	0.668
4	0.939	0.615
8	0.986	0.209

基于以上分析,我们认为由于密集的访存行为(规则的和不规则的),最先进的 hash joins 算法在商用多核结构上被可用带宽所限制.这种密集的访存行为不仅会导致性能下降,而且具有较高的能耗.

在 hash joins 算法中,计算部分相对简单,比如位操作、hash(如图 2 中的行③)、数组索引(如图 2 中的行⑪)和整数增量(如图 2 中的行⑫). 因此,预计移动数据产生的能耗比计算本身更为严重. 为了验证这一假设,我们进行进一步的实验来确定 hash joins 算法的能耗瓶颈.

2.3 能耗分析

我们从 2 个级别来进行能耗分析:粗粒度级别和细粒度级别. 在粗粒度级别,我们分析了 partition, build+probe 阶段的能耗. 在细粒度级别,我们将能消耗分为 3 类:数据移动、流水线阻塞和计算能耗,我们采用了文献[30-31]中提出的评测方法.

1) 粗粒度级别分析结果. 图 6 给出了在不同输入大小(32～256 M)和线程数(1～4)情况下,partition 和 build+probe 阶段的能耗比较. 平均来看,partition 阶段能耗约占总能耗的 86.4%. 因此,对于 CPU 和 DRAM 来说,有必要降低 partition 阶段的能耗.

2) 细粒度级别分析结果. 图 7 展示了整个 hash joins 过程的细粒度能耗分解. 我们测量由数据移动和流水线阻塞带来的能耗,流水线阻塞主要是由数据移动操作和功能单元的争抢引起的,剩余的能耗是由计算和其他原因导致的.

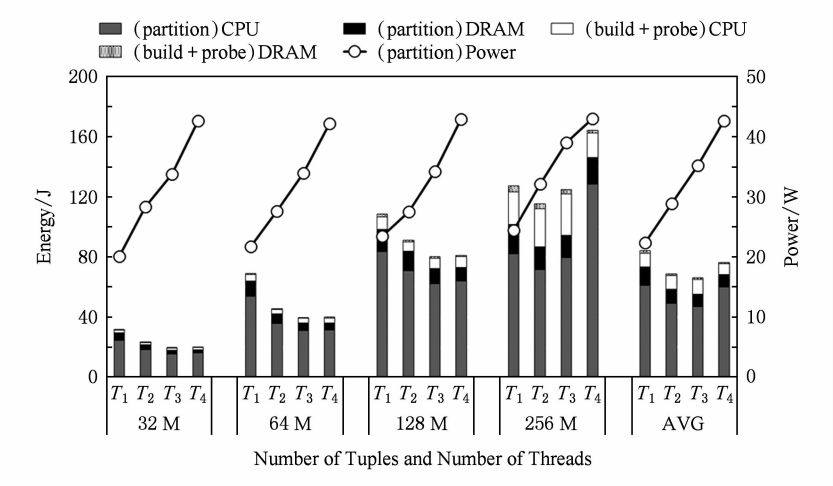


Fig. 6 Energy of the partition and build+probe

图 6 partition 阶段和 build+probe 阶段的能耗

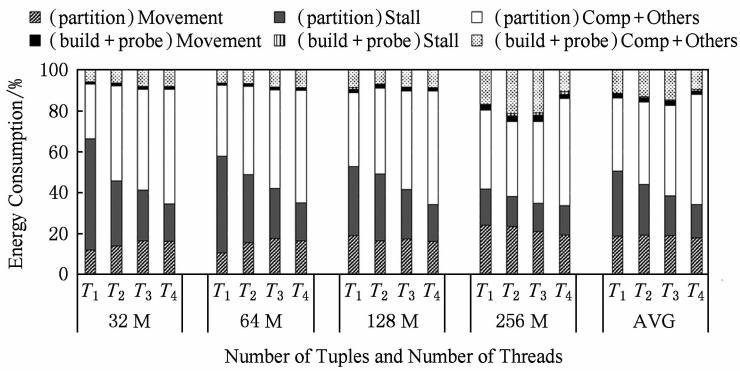


Fig. 7 Energy breakdown of the hash joins operation

图 7 hash joins 操作的能耗分解

对于 build+probe 阶段,由数据移动引起的能耗相对较小,仅占总能耗的 1.8%. 相比之下,计算引起的能耗约占总能耗的 11%,是数据移动能耗的 6.4 倍. 与 build+probe 阶段相比,partition 阶段能耗情况则完全不同.

在运行 1 个工作线程时,在数据移动、流水线阻塞和计算+其他方面消耗的能量分别为 18.6%, 31.9%和 35.8%. 可以看出,数据移动和流水线阻塞对整体能耗有很大的贡献.

虽然 hash joins 的所有阶段几乎使内存带宽饱和(参考 3.2 节),但 partition 和 build+probe 阶段的数据移动能耗有着显著的不同. 在 partition 阶段,数据 shuffle 过程存在大量高代价的不规则的访存行为,导致数据移动开销大. 而在 build+probe 阶段,只有常规的流式访存,使得数据移动成本可以忽略不计. 因此,设计 DRAM 中加速器的首要考虑

的因素是数据移动的成本,而不是内存带宽.

现代处理器为了降低内存访问延迟,设计了多个层次的存储结构,因此数据移动操作可以进一步分为 4 类:L1 到寄存器、L2 到寄存器、L3 到寄存器和内存到寄存器.这 4 类操作的能耗分布如图 8 所示.从 DRAM 到寄存器数据移动的能耗约占总能耗的 60%,而处理器内的移动数据(包括 L1 到寄存器、L2 到寄存器和 L3 到寄存器)的能耗约占总能耗的 40%.因此,有必要同时从 CPU 和 DRAM 两端降低数据移动的能耗.

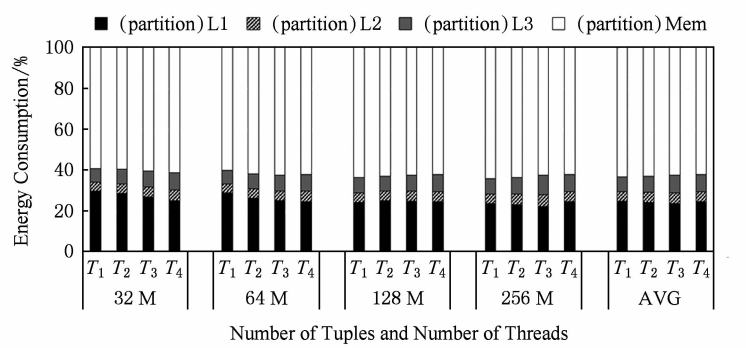


Fig. 8 Breakdown of energy caused by data movement in the partition phase

图 8 partition 阶段数据移动带来的能耗分解

综上,虽然最好的 hash joins 算法是内存受限的,但由于 partition 阶段有着不规则的内存访问,大部分的能耗是由流水线阻塞和数据移动产生的;而对于访存有规律的 build+probe 阶段,大部分能量消耗在计算上.为了在面积、功耗、散热受限的 DRAM 逻辑层上提高整体的能效,我们建议只在 DRAM 中加速 partition 阶段,而把 build+probe 阶段的加速留给 CPU.

2.4 在 DRAM 中加速的收益

通过将加速器集成到 DRAM 中,可以从各个方面减少 partition 阶段的能耗,从而减少总体能耗.

1) 数据移动的能耗.通过将加速器集成到 DRAM 中,可以避免 CPU 和 DRAM 之间高代价的往返数据传输,从而直接降低数据移动能耗.

2) 流水线阻塞的能耗.由于相当缓慢的内存访问导致了大量的流水线阻塞,因此数据访问的改进(通过将计算移动到更接近数据位置)也有助于减少流水线阻塞带来的能耗.

3) 计算的能耗.使用自定义 hash 分区加速器(HPA)可以显著降低计算能耗.

3 系统概览

图 9 展示了 hash joins 混合加速系统的总体结构,包括多/众核的主机处理器和一个或多个 3D DRAM stack.在每个 DRAM stack 的逻辑层中,我们将 hash 分区加速器(HPA)集成到每个 vault controllor 上,使 HPA 可以充分利用 3D DRAM 中极高的内部带宽.

1) 主机处理器.主机处理器是一个传统的多/众核处理器,例如 Intel 的 Haswell 或 Xeon Phi 处理器.它通过高速连接(用于传统的 HMC 系统中)或者 interposer(用于 Intel 的 Knights Landing 中)与 3D DRAM stack 进行通信.主机处理器可以通过引入 SIMD 单元、定制的加速器或 GPU 等加速器增强自身性能,从而提高大多数应用的性能.在我们的混合系统中,主机处理器主要侧重于利用 SIMD 单元来加速 build 和 probe 阶段.

2) 集成加速器的 3D 堆叠 DRAM.针对每个 3D DRAM stack,我们集成多个 HPA 来充分利用

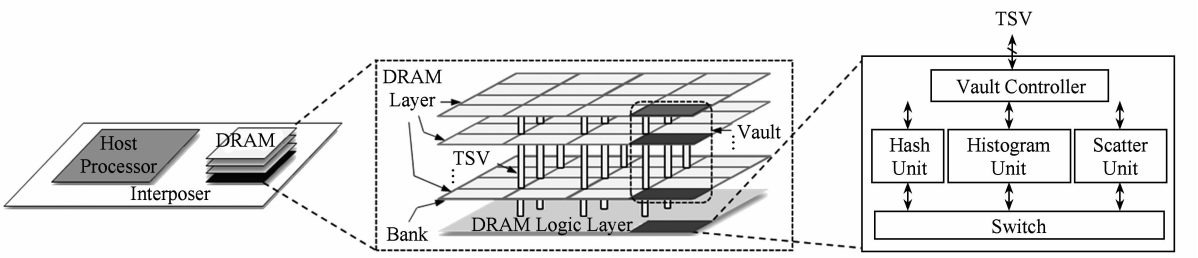


Fig. 9 The overview of proposed hybrid accelerated system

图 9 混合加速系统的总体架构

其高内存带宽,从而降低整体的能耗. 由于每个 3D DRAM stack 包含多个 vault,并且每个 vault 都由逻辑层的 vault controller 访问,所以我们将一个 HPA 集成到 vault controller 上.

为加速 partition 阶段而设计的 hash 分区加速器(HPA)包含 3 个主要部件:hash unit,histogram unit 和 shuffle unit. 这些单元通过 vault controller 访问 DRAM 层,并且与开关电路相连.

4 hash 分区加速器

在这一部分,我们首先介绍 HPA 的设计架构. 然后,探索可能的设计空间来找到最优的设计方案.

4.1 加速器设计

每个 HPA 的详细架构如图 10 所示. HPA 通过 vault controller 访问 DRAM.

1) hash unit. 如图 10(a)所示,我们以一个处理并行度为 4 的 hash unit 为例,介绍 hash unit 的架构. 它以流式访问的方式从 DRAM 中读取关系表中的多个元组,然后并行地处理它们的 keys,以位移或者掩码的方式来产生 hash 索引. 由于 hash unit 会被 histogram unit 和 shuffle unit 复用,所以加入了多路选择器(MUX)用于决定 hash 索引的输出目标.

2) histogram unit. 图 2 中的原始 histogram 算法应该串行执行. 为了增加 histogram 操作的并行度,我们采用本地存储器(local memory, LM)来保存 *my_hist* 的多个副本,每个副本都由单独的处理单元处理. 如图 10(b)所示,并行的 histogram 操作被分解为 2 个阶段:并行增量阶段(INC)和串行归约阶段(RED). 在 INC 阶段,hash unit 产生的 hash 索引 idx 用于更新对应的 LM 当前的 histogram 值. 在处理完从 DRAM 中读取的所有键后,最后的 RED 阶段进行所有 LM 的整合,以获得一个完整且保持数据一致性的 *my_hist*.

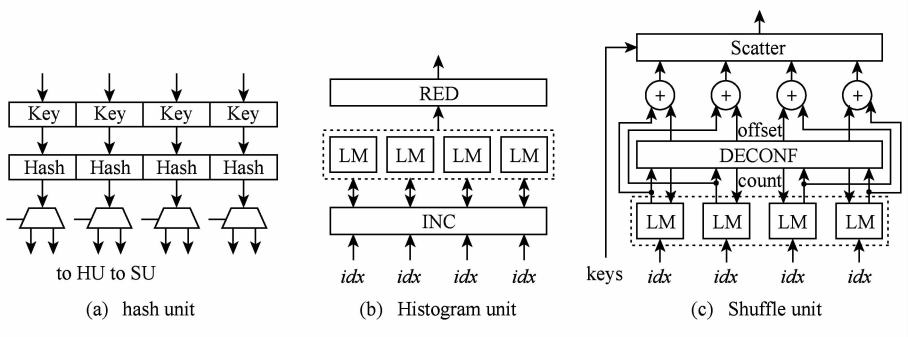


Fig. 10 The architecture of HPA

图 10 HPA 的架构

histogram unit 设计中需要关心的是 LM 的功耗和面积. LM 的数量越多,处理 *my_hist* 的并行度越高,但是相应地会增加面积和功耗. 同时为了减少 TLB miss,LM 的大小还受到 TLB 项数量的限制[11]. 具体来说,每个 *my_hist* 副本大小最好与 TLB 项数量相等. 在现代处理器中针对 L1 设计的数据 TLB 通常是 64 或 128 项. 因此,给定一个很高的并行度,比如 512,LM 的总大小为 256 KB(512×128×4). 当然,我们应该谨慎选择并行度,在性能、功耗和面积之间做设计权衡.

3) shuffle unit. 与 histogram unit 的并行设计思路类似,shuffle unit 的设计的挑战是如何将多个元组同时写入到各自在 *tmp* 数组中的目标地址($tmp[dst[idx]]$),然后更新 *dst* 数组中的目标地址(如图 2 所示). 如果多个处理路径中具有相同目的地址的元组,还需要处理目标地址冲突问题.

我们提出的并行 shuffle unit 由 4 个阶段组成:第 1 阶段(DIST)是根据 hash 索引(idx)从 *dst* 数组中并行地读取多个目标地址(图 2 的第⑪行). 第 2 阶段(DECONF)用于检测多个处理路径之间有冲突的目的地址. DECONF 阶段产生基于原始目标地址的偏移,从而确定在 *tmp* 数组中正确的目的地址. 同时 DECONF 阶段也产生相同目的地址的计数值来更新 idx 数组. 第 3 阶段(SCATTER)将元组移动到正确的位置,正确的位置根据 DECONF 阶段产生的偏移和 DIST 阶段产生的存储在 *tmp* 数组的原始目的地址联合计算得出. 最后一个阶段(UPDATE)根据 DECONF 阶段产生的计数值更新目的地址.

DECONF 本质上是一个在原始目的地址上执行 XNOR 操作的 XNOR 网络. 图 11 给出了 XNOR 网络的结构示例,在此示例中,4 个目的地址的数值

d_0,d_1,d_2 和 d_3 从 dst 数组中并行的读出.为了计算 d_0 的总数量 count(d_0),首先 d_0 分别与 d_1,d_2 和 d_3 进行同或操作,然后所有同或的数值进行求和.同样地,通过对 xnor(d_1,d_0), xnor(d_1,d_2)和 xnor(d_1,d_3)求和计算得出 count(d_1).计算目标地址偏移也可以通过复用 XNOR 网络来实现.例如,offset(d_1)是 xnor(d_1,d_2)和 xnor(d_1,d_3)的和.

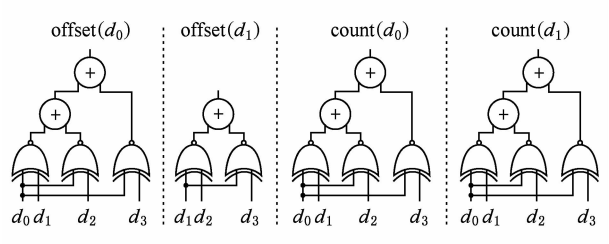


Fig. 11 XNOR network

图 11 XNOR 网络图示

4.2 设计空间探索

在 HPA 的设计过程中,在并行度和频率上存在多种选择,同时 3D 堆叠 DRAM 也可以选择不同的配置.表 4 列出了 3 种可能的 3D 堆叠 DRAM 配置:HI,MD 和 LO,它们的内部带宽在 860 GBps~360 GBps 之间.我们探索了由每个 vault 的并行度(1~512)、操作频率(0.4~2.0 GHz)和 DRAM 配置组成的设计空间,尝试在在并行度和功耗之间进行设计权衡.更多的细节和评估方法论可以在第 6 节中找到.

Table 4 3D-Stacked DRAM Configurations

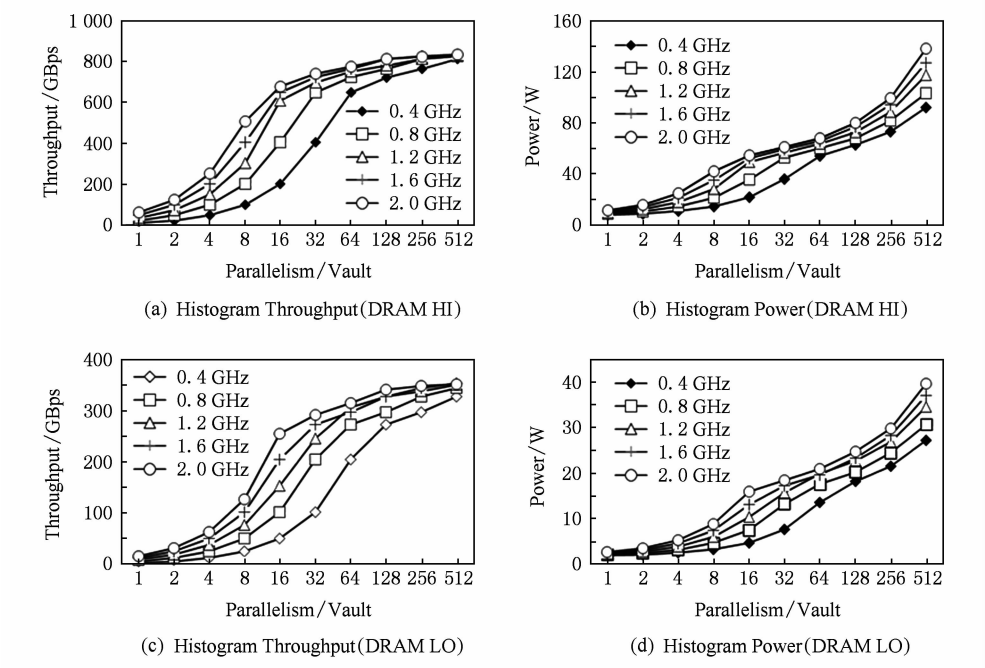
表 4 3D 堆叠内存配置

Parameter	HI	MD	LO
# Vault	16	8	4
# Layer	8	4	4
Total # TSV	2048	2048	1024
Internal BW/GBps	860	710	360
Power/W	45	30	25

4.2.1 histogram 操作

图 12 显示了使用 HI 和 LO DRAM 配置,在不同的频率和并行度下 histogram 操作的吞吐率和功耗.第 1 个观察结果是吞吐率随着并行度和频率的增加而增加,这充分说明了我们设计的可伸缩性.针对 HI 配置,在 2 GHz 下当并行度从 8 增加到 16,相应的吞吐率也增加了约 2 倍.然而,随着频率和并行度的不断增加,所需的内存带宽也会增加,从而使吞吐率受到可用内存带宽的限制.在另一个例子中,给定 0.4 GHz 的频率,虽然并行度从 32 增加到 64 时,吞吐率增加了约 2 倍,但是当并行度从 64 增加到 128 时,吞吐率只有 1.6 倍的增加.另外,当并行度从 256 增加到 512 时,吞吐率的增益仅为 1.06.针对不同的频率,在不同并行度下,HI 和 LO 配置的吞吐率最终会分别收敛到 840 GBps 和 450 GBps.

至于功耗,除了直观地观察到功耗随并行度和频率增加外,我们还观察到不同频率之间的功耗差距会随着并行度的变化而变化.对 HI 配置,当并行



(a) Histogram Throughput(DRAM HI)

(b) Histogram Power(DRAM HI)

(c) Histogram Throughput(DRAM LO)

(d) Histogram Power(DRAM LO)

Fig. 12 The throughput and power of the histogram operation

图 12 histogram 操作的吞吐率和功耗

度是 16 时,最大功耗(在 2 GHz 时)比最小功耗(在 0.4 GHz 时)高 2.54 倍,而当并行度是 64 时,功耗差距只有 1.26 倍. 这是由于 DRAM 在相对较小的并行度的情况下,不同频率之间的功耗有一个较大的差距.

为了详细说明功耗,图 13 显示了 HI 配置下的 histogram 操作的功耗分解. 我们可以看到,随着并行度的增加,DRAM 的功耗所占比例(例如 DRAM_RD)也随之增加. 然而,当并行度大于 32 时,DRAM 功耗所占的比例减小,因为内存带宽几乎饱和,同时 HASH 和 INC 功耗显著增加. 在 2 GHz 下,并行度是 512 时,HASH 和 INC 功耗甚至高于 DRAM 功耗.

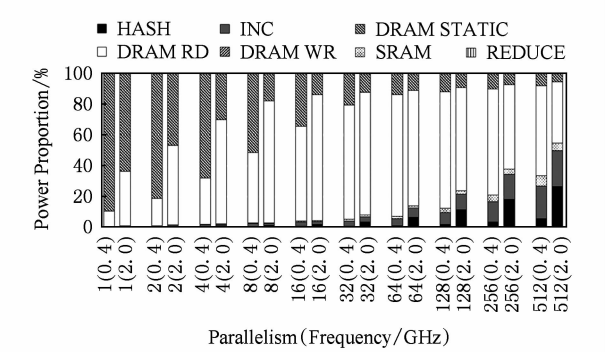


Fig. 13 The power breakdown of the histogram operation

图 13 histogram 操作功耗分解

我们还在图 14 中,以能量-延迟积(EDP)为标准,展示了在不同设计方案下的每个 vault 的能效. 在所有的 DRAM 配置下,随着并行度的增加,EDP 并没有显示出单调下降的趋势. 特别值得注意的是,在并行度在 32 左右时,所有 DRAM 配置的 EDP 达到了最优值. 举例来说,使用 HI 配置,在 1.2 GHz 操作频率下,达到最优的 EDP 需要 64 的并行度. 一般来说,相比 HI 配置,MD 和 LO DRAM 配置具有更好的能效(每个 vault),随着并行度的增大变得尤为明显. 图 14 还显示了不同设计选择下的面积. 可以看出,随着并行度的增加面积会显著增加. 在每个 vault 中每一个 HPA 的最大面积大约为 1.4 mm^2,这是因为 HPA 需要更多的处理单元.

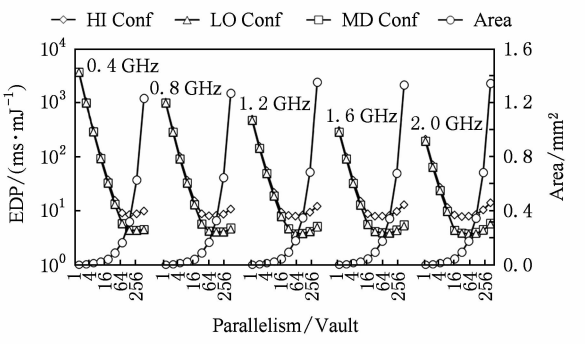


Fig. 14 Energy-delay product (EDP) of different design options for the histogram operation

图 14 不同设计下 histogram 操作的能量-延迟积

4.2.2 shuffle 操作

图 15 展示了使用 HI 和 LO DRAM 配置,在不同的频率和并行度下 shuffle 操作的吞吐率和功耗.

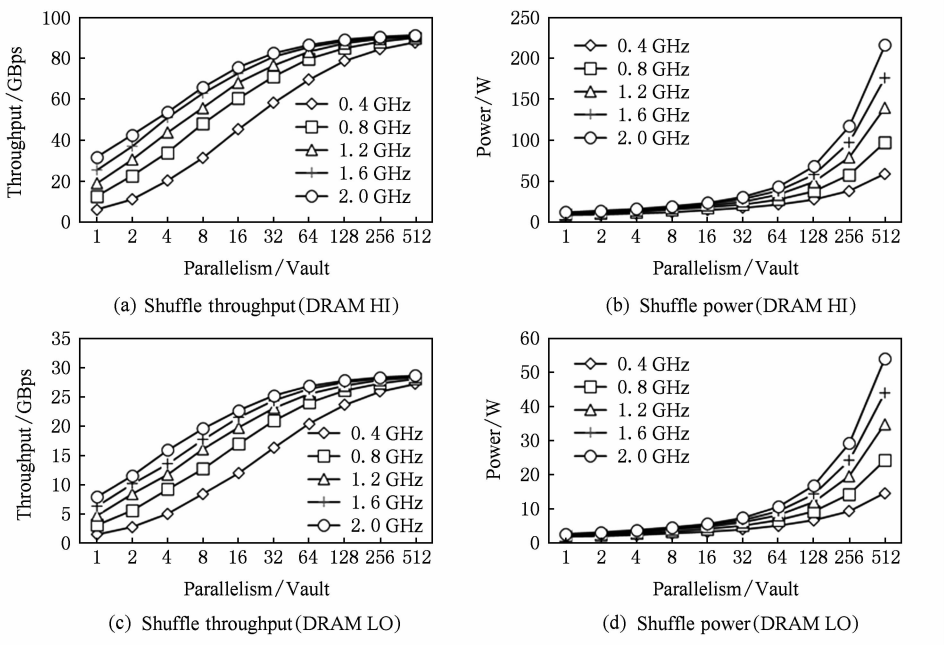


Fig. 15 The throughput and power of the shuffle operation

图 15 shuffle 操作的吞吐率和功耗

一般来说,吞吐率会随着并行度和频率的提高而增加,在 HI 和 LO 的配置下,它最终分别收敛到 91 GBps 和 29 GBps. 最大吞吐率主要受 SCATTER 阶段的限制,在该阶段中是以相对随机的方式进行内存访问的. 随着并行度和频率的增加,功耗也会随之增长. 在 2 GHz 下,在 HI 和 LO 配置中最大功耗分别达到了 216 W 和 54 W.

图 16 展示了 shuffle 操作下的功耗分解. 当并行度很小时,比如 1～16,DRAM 的静态功耗占主要部分. 随着并行度的增加,HASH 和 SCATTER 的功耗显著增加,最终超过了 DRAM 的静态功耗.

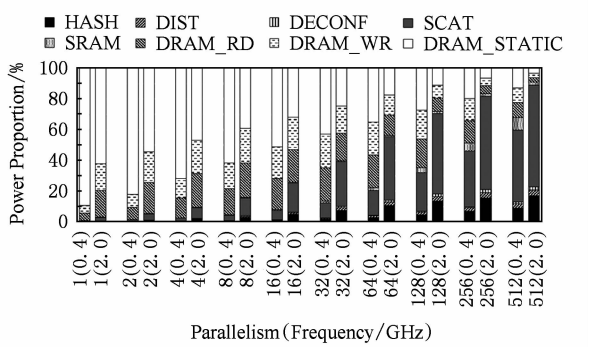


Fig. 16 The power breakdown of the shuffle operation

图 16 shuffle 操作的功耗分解

图 17 展示了每个 vault 的 shuffle 操作能效(EDP)和面积. 达到最优 EDP 的并行度是在 32～128 之间. 例如,在 2 GHz 下,并行度为 32 的 EDP 比并行度为 512 的 EDP 好 1.77 倍. 在所有的这些配置中,获得最优 EDP 的配置为并行度=64、频率=1.2 GHz 和 DRAM 配置=LO. 最优配置所需要的面积为 0.18 mm^2(每 vault).

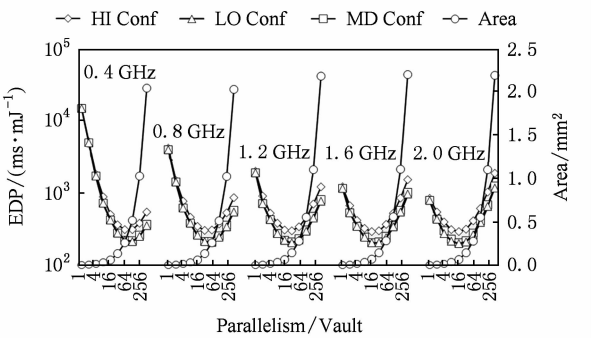


Fig. 17 Energy-delay product (EDP) of different design options for the shuffle operation

图 17 不同设计下 shuffle 操作的能量-延迟积

4.2.3 设计决策

要做出 HPA 的最优设计决策,我们必须综合考虑 histogram 和 shuffle 两个操作的性能、功耗和面积. 我们通过对比不同配置下这 2 个操作的总 EDP 决定最优的 HPA 配置. 我们得出结论是:当输入的大小是 128 M 时,HPA 的最优配置是并行度=16、频率=2.0 GHz、DRAM 配置=HI. 相应的面积只有 1.78mm^2,加速器的功耗(不含 DRAM 功耗)只有 7.52 W.

5 软件设计

软件设计包括用于调用 HPA(hash 分区加速器)的编程接口和用于加速 build 和 probe 阶段的 SIMD 指令.

5.1 HPA 编程接口

我们采用的编程接口是在文献[6]中提出的编程库上进行搭建的. 所使用的库函数包括内存管理库函数 malloc 和 free,以及加速器控制库 acc_plan(用于配置加速器)和 acc_execute(用于调用已配置的加速器). 我们对 acc_plan 函数进行扩展,使它支持 2 个新的操作:HIST 和 SHUF,分别用来控制 HPA 执行 histogram 和 shuffle 操作. 在这些库函数的帮助下,程序员可以很容易地在程序中使用 HPA. 此外,我们使用的编程接口本质上是一个顺序编程模型,所以可以减轻异构系统的编程负担.

然而,加速代码(如 histogram 和 shuffle 操作)是一个并行执行范式,其中每个工作线程都需要处理自己的数据分块. 为了缩小顺序编程模和并行编程模型之间的差距,我们在每个工作线程调用 HPA 之前添加 barrier 操作,然后只有 1 个线程通过 acc_plan 和 acc_execute 调用 HPA,同时所有其他线程都在等待 HPA 执行(histogram 或 shuffle 操作)完成. 在 HPA 完成执行之后,每个工作线程恢复各自的执行流,进行后续操作.

5.2 针对 build+probe 阶段的 SIMD 加速

近年来,为提高 hash joins 的执行效率,针对 SIMD 和向量化有很好的研究. 我们采用了在文献[13,15]中提出的 SIMD 算法,在多核(例如 Intel Haswell)和众核(如 Intel Xeon Phi)处理器上加速 build 和 probe 阶段.

图 18 展示了在支持 512-bit SIMD 的 Xeon Phi 处理器上用 SIMD 执行 build 阶段的基本思想. 直观来看,多个 hash 操作可以并行执行(参考图 3 中的行③). 在图 18 行④中,gather 操作从元组中选择键. 在行⑤中,基于 SIMD 的 hash 操作(包括 SIMD 按位与和移位)并行地进行多个键的 hash 处理. 在行⑥中,将hash处理的结果写回*extVector*. *extVector*

中包含 16 个 idx(参考图 3),用于顺序更新 bucket 数组. probe 阶段的 SIMD 执行类似于 build 阶段.

```
① for (i=0; i<numR-(numR%16);) {
②     …
③     //SIMD operations
④     key=_mm512_i32gather_epi32(offset, lRel, 4);
⑤     key=simd_hash(key, MASK, NR);
⑥     _mm512_store_epi32((void*)extVector, key);
⑦     //vector-based build phase
⑧     for (int j=0; j<16; j++) {
⑨         next[i]=bucket[extVector[j]];
⑩         bucket[extVector[j]]=++i;
⑪     }
⑫     lRel+=32;
⑬ }
```

Fig. 18 The basic idea of 512-bit SIMD execution of the build phase

图 18 512-bit SIMD 执行 build 阶段的基本思想

6 实 验

在本节中,我们展示了混合加速系统的能效,并将其与支持 SIMD 的商用多/众核处理器进行比较.

6.1 评测方法

1) 评测基础设施. 我们采用了文献[6]中提出的混合评价方法论来评估我们的系统. 这种评估方法的主要优点是真正的执行行为可以被如实地模拟.

2) 主机处理器. 表 5 列出了用于评估的主处理器,其中 Intel Haswell 和 Intel Xeon Phi 处理器分别是商用多核和众核处理器. 我们在主机处理器上运行 PRO 算法,并测量不包括加速部分(partition 阶段的 histogram 和 shuffle 操作)程序的性能和功耗. 我们还测量了使用主处理器把加速部分任务分派到 HPA 加速器的开销(包括加速器控制带来的开销和刷新缓存以保证数据一致性带来的开销).

3) 加速器仿真. 通过各种仿真工具对 hash 分区加速器(HPA)的性能、功耗和面积行评估. 该加速器的性能是由一个内部的、C-level、周期精准的模拟器估计的. DRAM 的访存延迟是由一个周期精准的 3D 堆叠 DRAM 模拟器获得的[8]. 我们采用 32 nm 的库和编译器来评估 HPA 的功耗和面积.

4) Baseline. 为了展现混合加速系统的优势,我们对比了 Intel Haswell 和 Xeon Phi 处理器和我们的系统. 如表 5 所示,它们的内存带宽的峰值分别为 21.2 GBps 和 320 GBps. 我们在这 2 个平台上全面的测量了 hash joins 算法的性能和功耗.

5) HPA 配置和输入数据集. 如第 4 节所探讨的,HPA 的最佳频率和并行度分别为 2.0 GHz 和 16. 3D 堆叠 DRAM 是 HI 配置,它包含 16 个 vault,最大的内部带宽为 860 GBps(如表 4 所示). 关于输入数据集,我们将输入关系表的元组数目作为可调整的变量,从 32 M~256 M① 进行了测试.

Table 5 The Baseline Host Processors

表 5 基准主处理器

Hardware	Haswell i7-4770K	Xeon Phi 5110P
Frequency/GHz	3.5	1.053
Core/Thread	4/8	60/240
L1 cache/KB	32/32 I/D-Cache	32/32 I/D-Cache
L2 cache/KB	256	512
L3 cache/MB	8	—
Bandwidth/GBps	21.2	320

6.2 实验结果

我们用不同大小的输入数据集,在性能、能耗和 EDP 这 3 个方面将我们提出的混合系统和 Haswell 和 Xeon Phi 处理器进行了比较,结果如图 19 所示. 与 Haswell 处理器相比,混合系统在性能、功耗和 EDP 上分别优化了 6.70,7.08 和 47.52 倍. 与 Xeon Phi 处理器相比,混合系统在性能、功耗和 EDP 上分别优化了 4.17,4.68 和 19.81 倍. 我们观察到,Xeon Phi 处理器的收益比 Haswell 处理器的收益要小. 原因是在 Xeon Phi 上,build 和 probe 阶段执行所占的比例比 Haswell 更大.

如图 20 所示,针对 hash partition 操作,我们比较了 HPA 与基准平台的性能、能耗和 EDP. 平均而言,与 Intel Haswell 处理器相比,HPA 的性能、功耗和 EDP 分别优化了 30,90 和 2 725 倍. 与 Xeon Phi 处理器相比,EDP 优化甚至超过了 6 000 倍. 性能和能效的显著提高主要来源于定制加速器和由 3D 堆叠 DRAM 提供的高内存带宽.

为了更详细地分析实验结果,我们在图 21 中给出了 Intel Haswell 处理器的执行过程分解. 根据图 21(a)中执行时间的分解,HPA 只占整个执行时间的 19.2%. 相比之下,加速前的 partition 阶段的执行时间大约是整个执行时间的 90%,这表明 partition 阶段已经被加速了大约 30 倍. 此外,当输入数据集

① 单位为元组的个数,数据的总大小是 0.5 GB~4 GB (256 M×8×2 B=4 GB)

很小时(比如 32 M),HPA 的调用成本是巨大的,占总执行时间的 8.6%.当输入数据的大小增加时,调用成本逐渐变得可以忽略,例如,当输入数据大小是 256 M 时,调用成本只是总执行时间的 1%.

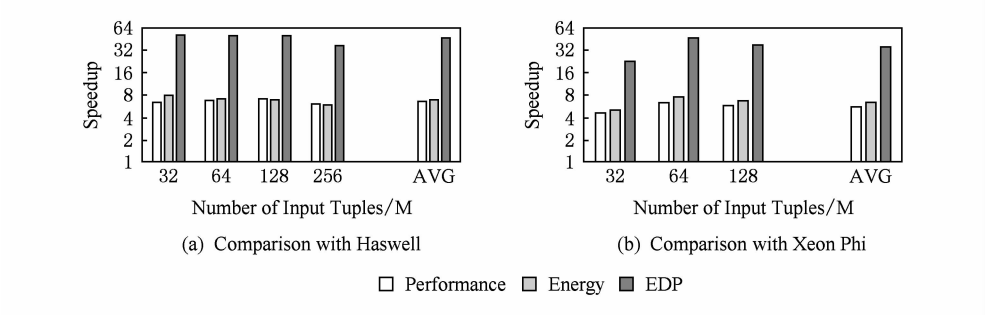


Fig. 19 Overall comparison with the Intel Haswell and Xeon Phi processor

图 19 与 Intel Haswell 和 Xeon Phi 处理器总体对比

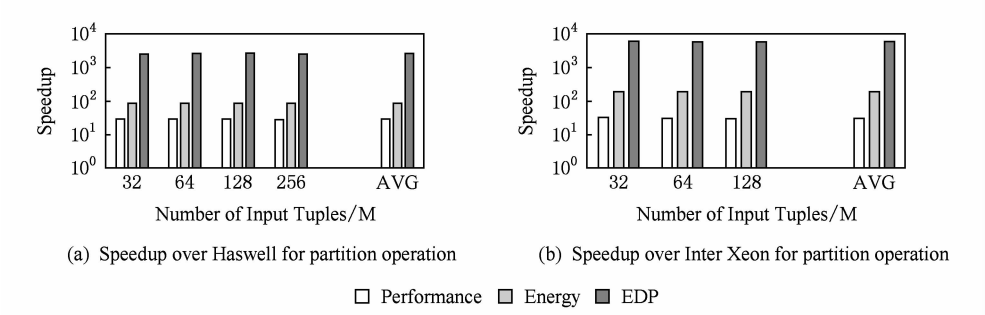


Fig. 20 Comparison with the Intel Haswell and Xeon Phi processor for the hash partition operation

图 20 与 Intel Haswell 和 Xeon Phi 处理器在 partition 操作上的对比

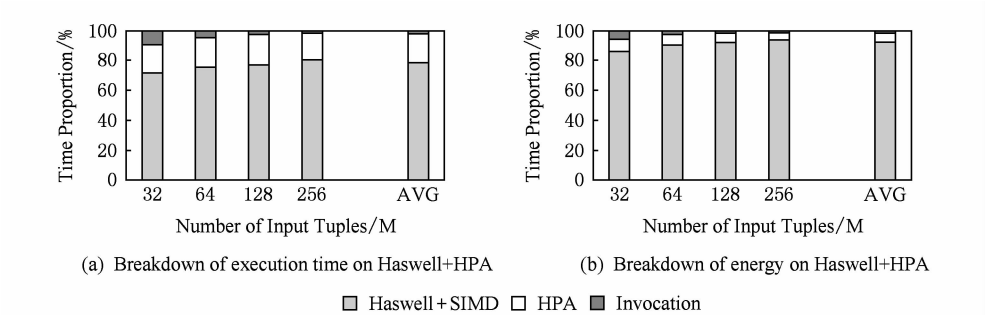


Fig. 21 Breakdown of execution time and energy on the hybrid system and Intel Haswell processor

图 21 混合系统以及 Intel Haswell 处理器的执行时间和功耗分解

在图 21(b)中,功耗的分解进一步证明了 HPA 的优势.平均来说,HPA 功耗只占总功耗的 6.2%.同时,在 Haswell 上实际计算的功耗远远高于调用 HPA 的功耗.平均而言,Haswell 计算和调用 HPA 的功耗比率分别为 92.7%和 1.1%.

6.3 功耗和面积总结

我们在表 6 中展示了混合系统的功耗和面积,该系统由 HPA,DRAM 和主机处理器(Haswell)组成.就面积而言,HPA(由 hash、histogram 和 shuffle unit 组成)的总面积仅为 Micron 的 Hybrid Memory Cube (HMC)总尺寸的 2.6%.因此,HMC 的逻辑层很容易容纳 HPA.此外,HPA 的面积只占由 DRAM 和 CPU 组成的整个系统的 0.72%.

在功耗方面,HMC 只给 3D 堆叠 DRAM 增加了 7.52 W 的功耗,在 DRAM 中使用 HPA 的总功耗为 28.43W.由于 DRAM 也被主机处理器(如 Haswell)访问,我们还测量了由 Haswell 处理器访问引起的 DRAM 功耗,相应的 DRAM 功耗是 9.79 W.由于 DRAM 不能被 HPA 和 Haswell 同时访问,所以 DRAM 的总功耗不会超过 18.64 W.对于主要用来执行 build 和 probe 阶段 Haswell 处理器,测得其总功耗为 89.75 W.

Table 6 Power and Area Consumption of the Entire System

表 6 整个系统的功耗和面积消耗

Hardware	Area/mm²	Average Power/W
hash Unit	0.20	1.16
Histogram Unit	0.58	1.41
Shuffle Unit	0.99	4.95
DRAM	68	20.91 (w/HPA) 9.79 (w/CPU)
CPU (Haswell)	177	89.75
Total	245	

7 相关工作

在本节中,我们列出了针对数据库操作、算法、应用而使用新型硬件的相关工作.

1) SIMD 扩展. 在过去 10 年中,为了加速数据库的原语,已经广泛部署 SIMD 扩展. Zhou 等人在文献[32]中首先用 SIMD 指令来加速基本的数据库操作,如索引扫描和嵌套循环联接. Willhalm 等人在文献[33]中使用向量处理单元在数据扫描中进行解压缩. Schlegel 等人在文献[34]中使用 SIMD 支持在 Xeon Phi 处理器上实现了一个可扩展的频繁项集挖掘算法. Polychroniou 等人在文献[14]中使用 SIMD 指令重写了内存数据库操作,其中包含扫描、hash、筛选和排序.

2) FPGAs. 已经有很多用 FPGA 加速数据库查询操作的工作. 比如 IBM Netezza,它是一个结合了 Xeon 服务器和 FPGAs 的商用系统,可以提高数据库查询的性能. Sukhwani 等人在文献[35]中利用 FPGA 对传统 DBMS 进行数据解压和谓词评价. Chung 等人在文献[36]中提出了一种框架,将查询语言 LINQ 映射到硬件模板,该模板可以由 FPGA 或 ASIC 实现. 由 ARM A9 处理器和 FPGA 组成的 SoC 系统可以提高 30.6 倍的能效.

3) GPUs. 图处理单元也被广泛用于加速通用数据查询[37]和数据库操作原语,如索引[38]、排序[39]和连接[40-41].

4) 自定义的逻辑单元. 在数据库领域自定义加速器也被广泛地使用. Kocberber 等人在文献[42]中为 hash 索引查找建立了 on-chip 加速器. Wu 等人在文献[43]中首先提出了一种用于 range partitioning 的硬件加速器:HARP. 为了充分利用 HARP 的计算能力,他们还提出了一个硬件-软件流式框架,用于 CPU 和 HARP 之间的通信. 与他们的工作相比,我们更关注 hash 分区,并且我们在最先进的 hash joins(即 PRO)中发现 hash 分区是内存受限的. 因此,将加速过程集成到 DRAM 中可以使 hash 分区获得巨大的收益. 为了实现这个目标,我们设计了一个 hash 分区加速器(HPA),它可以适应 3D 堆叠 DRAM 架构. Wu 等人在文献[44]中进一步提出了 Q100 数据库处理单元(DPU),以有效地处理数据库原语,如聚合、连接和排序.

随着靠近数据的处理技术出现,也有一些将自定义逻辑集成到 DRAM 中进行数据库处理的研究. Xi 等人在文献[45]中设计了 JAFAR 加速器,他们将选择操作附加到 DRAM 中,获得了 9 倍的加速. Seshadri 等人在文献[46]中提出在 DRAM 中实现位操作(AND 和 OR),进而加速各种数据库操作(例如索引).

8 总 结

尽管 3D-die 的堆叠技术使得计算单元可以集成到 3D 堆叠 DRAM 中,但由于其严格的面积、功耗、散热的限制,使得我们在架构设计时应该谨慎地考虑如何在 3D 堆叠 DRAM 中加入最合适的逻辑单元. 在本文中,我们以数据移动为驱动,对 CPU 和 DRAM 之间的加速任务进行划分. 基于这种方法,我们设计了一种混合加速系统,用于加速数据库和大数据系统中的基本操作 hash joins. 该系统包含一个内存端定制的 hash 分区加速器(HPA)和处理器端的 SIMD 单元. 内存端的加速器用于加速数据移动受限的执行阶段,而处理器端 SIMD 加速单元则用于加速数据移动开销较低成本的执行阶段. 实验结果表明,该系统与 Intel Haswell 和 Xeon Phi 处理器相比,仅仅用 7.52W 额外的功耗,将能效分别提升了 47.52 和 19.81 倍. 同时我们的设计方法可以很容易地被应用到其他应用程序的加速设计中.

参 考 文 献

[1] Putnam A, Caulfield A M, Chung E S, et al. A reconfigurable fabric for accelerating large-scale datacenter services [C] //Proc of the 41st Annual Int Symp on Computer Architecture. Los Alamitos, CA: IEEE Computer Society, 2014: 13-24

[2] Lim K, Meisner D, Saidi A G, et al. Thin servers with smart pipes: Designing SoC accelerators for Memcached [C] //Proc of the 40th Annual Int Symp on Computer Architecture. New York: ACM, 2013: 36-47

[3] Qadeer W, Hameed R, Shacham O, et al. Convolution engine: Balancing efficiency & flexibility in specialized computing [C] //Proc of the 40th Annual Int Symp on Computer Architecture. New York: ACM, 2013: 24-35

[4] Krishna A, Heil T, Lindberg N, et al. Hardware acceleration in the IBM PowerEN processor: Architecture and performance [C] //Proc of the 21st Int Conf on Parallel Architectures and Compilation Techniques. New York: ACM, 2012: 389-400

[5] Zhang D, Jayasena N, Lyashevsky A, et al. TOP-PIM: Throughput-oriented programmable processing in memory [C] //Proc of the 23rd Int Symp on High-performance Parallel and Distributed Computing. New York: ACM, 2014: 85-98

[6] Guo Q, Alachiotis N, Akin B, et al. 3D-stacked memory-side acceleration: Accelerator and system design [C] //Proc of the 2nd Workshop on Near-Data Processing in Conjunction with the 47th IEEE/ACM Int Symp on Microarchitecture. Workshop on Near-data Processing. Piscataway, NJ: IEEE, 2014

[7] Nair R, Antao S, Bertolli C, et al. Active memory cube: A processing-in-memory architecture for exascale systems [J]. IBM Journal of Research and Development, 2015, 59(2/3): No. 17

[8] Akin B, Franchetti F, Hoe J C. Data reorganization in memory using 3D-stacked DRAM [C] //Proc of the 42nd Annual Int Symp on Computer Architecture. New York: ACM, 2015: 131-143

[9] Zaharia M, Chowdhury M, Das T, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing [J]. USENIX Conf on Networked Systems Design and Implementation, 2012, 70(2): 2-2

[10] Blanas S, Li Y, Patel J M. Design and evaluation of main memory hash join algorithms for multi-core CPUs [C] //Proc of the 2011 ACM SIGMOD Int Conf on Management of Data. New York: ACM, 2011: 37-48

[11] Teubner J, Alonso G, Balkesen C, et al. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware [C] //Proc of the 2013 IEEE Int Conf on Data Engineering. Piscataway, NJ: IEEE, 2013: 362-373

[12] He J, Lu M, He B. Revisiting co-processing for hash joins on the coupled CPU-GPU architecture [J]. VLDB Endowment, 2013, 6(10): 889-900

[13] Jha S, He B, Lu M, et al. Improving main memory hash joins on intel Xeon Phi processors: An experimental approach [J]. VLDB Endowment, 2015, 8(6): 642-653

[14] Polychroniou O, Raghavan A, Ross K A. Rethinking SIMD vectorization for in-memory databases [C] //Proc of the 2015 ACM SIGMOD Int Conf on Management of Data. New York: ACM, 2015: 1493-1508

[15] Balkesen C, Teubner J, Alonso G, et al. Main-memory hash joins on modern processor architectures [J]. IEEE Trans on Knowledge & Data Engineering, 2015, 27(7): 1754-1766

[16] Han Xixian, Yang donghua, Li jianzhong. DBCC-Join a novel cache-conscious disk-based join algorithm [J]. Chinese Journal of Computers, 2010, 33(8): 1501-1511 (in Chinese) (韩希先, 杨东华, 李建中. DBCC-Join:一种新的高速缓存敏感的磁盘连接算法[J]. 计算机学报, 2010, 33(8): 1501-1511)

[17] Manegold S, Boncz P, Kersten M. Optimizing main-memory join on modern hardware [J]. IEEE Trans on Knowledge & Data Engineering, 2002, 14(4): 709-730

[18] Balasubramonian R, Chang J, Manning T, et al. Near-data processing: Insights from a micro-46 workshop [J]. IEEE MICRO, 2014, 34(4): 36-42

[19] Elliott D, Snelgrove W, Stumm M. Computational RAM: A memory-SIMD hybrid and its application to DSP [C] //Proc of the IEEE Custom Integrated Circuits Conf. Piscataway, NJ: IEEE, 1992: 30.6.1-30.6.4

[20] Gokhale M, Holmes B, Iobst K. Processing in memory: The terasys massively parallel PIM array [J]. Computer, 1995, 28(4): 23-31

[21] Oskin M, Chong F T, Sherwood T. Active pages: A computation model for intelligent memory [C] //Proc of the Int Symp on Computer Architecture. Los Alamitos, CA: IEEE Computer Society, 1998: 192-203

[22] Loi G L, Agrawal B, Srivastava N, et al. A thermally-aware performance analysis of vertically integrated (3-D) processor-memory hierarchy [C] //Proc of the 43rd Annual Design Automation Conf. New York: ACM, 2006: 991-996

[23] Loh G H. 3D-stacked memory architectures for multi-core processors [C] //Proc of the 35th Annual Int Symp on Computer Architecture. Los Alamitos, CA: IEEE Computer Society, 2008: 453-464

[24] Woo D H, Seong N H, Lewis D, et al. An optimized 3D-stacked memory architecture by exploiting excessive, high-density TSV bandwidth [C] //Proc of the 16th IEEE Int Symp on High Performance Computer Architecture. Piscataway, NJ: IEEE, 2010: 1-12

[25] Kim D H, Athikulwongse K, Healy M, et al. 3D-maps: 3D massively parallel processor with stacked memory [C] //Proc of Solid-State Circuits Conf Digest of Technical Papers. ISSCC, 2012: 188-190

[26] Pugsley S H, Jestes J, Zhang H, et al. NDC: Analyzing the impact of 3D-stacked memory+logic devices on mapreduce workloads [C] //Proc of the Int Symp on Performance Analysis of Systems and Software. Piscataway, NJ: IEEE, 2014: 190-200

[27] Low T M, Guo Q, Franchetti F. Optimizing space time adaptive processing through accelerating memory-bounded operations [C] //Proc of IEEE High Performance Extreme Computing Conf. Piscataway, NJ: IEEE, 2015: 1-6

[28] Hähnel M, Döbel B, Völp M, et al. Measuring energy consumption for short code paths using RAPL [J]. ACM Sigmetrics Performance Evaluation Review, 2012, 40(3): 13-17

[29] McCalpin J D. Memory bandwidth and machine balancein current high performance computers [N]. IEEE Technical Committee on Computer Architecture Newsletter, 1995: 19-25

[30] Kestor G, Gioiosa R, Kerbyson D, et al. Quantifying the energy cost of data movement in scientific applications [C] // Proc of IEEE Int Symp on Workload Characterization. Piscataway, NJ: IEEE, 2013: 56-65

[31] Pandiyan D, Wu C J. Quantifying the energy cost of data movement for emerging smart phone workloads on mobile platforms [C] //Proc of IEEE Int Symp on Workload Characterization. Piscataway, NJ: IEEE, 2014: 171-180

[32] Zhou J, Ross K A. Implementing database operations using SIMD instructions [C] //Proc of the 2002 ACM SIGMOD Int Conf on Management of Data. New York: ACM, 2002: 145-156

[33] Willhalm T, Popovici N, Boshmaf Y, et al. SIMD-scan: Ultra fast in-memory table scan using on-chip vector processing units [J]. VLDB Endowment, 2009, 2(1): 385-394

[34] Schlegel B, Karnagel T, Kiefer T, et al. Scalable frequent itemset mining on many-core processors [C] //Proc of the 9th Int Workshop on Data Management on New Hardware. New York: ACM, 2013: 3:1-3:8

[35] Sukhwani B, Min H, Thoennes M, et al. Database analytics acceleration using FPGAs [C] //Proc of the 21st Int Conf on Parallel Architectures and Compilation Techniques. New York: ACM, 2012: 411-420

[36] Chung E S, Davis J D, Lee J. Linqits: Big data on little clients [C] //Proc of the 40th Annual Int Symp on Computer Architecture. New York: ACM, 2013: 261-272

[37] Bakkum P, Skadron K. Accelerating SQL database operations on a GPU with CUDA [C] //Proc of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units. New York: ACM, 2010: 94-103

[38] Kim C, Chhugani J, Satish N, et al. Fast: Fast architecture sensitive tree search on modern CPUs and GPUs [C] //Proc of the 2010 ACM SIGMOD Int Conf on Management of Data. New York: ACM, 2010: 339-350

[39] Satish N, Kim C, Chhugani J, et al. Fast sort on CPUs and GPUs: A case for bandwidth oblivious SIMD sort [C] //Proc of the 2010 ACM SIGMOD Int Conf on Management of Data. New York: ACM, 2010: 351-362

[40] Pirk H, Manegold S, Kersten M L. Accelerating foreign-key joins using asymmetric memory channels [C] //Proc of Int Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures. New York: ACM, 2011: 27-35

[41] Kaldewey T, Lohman G, Mueller R, et al. GPU join processing revisited [C] //Proc of the 8th Int Workshop on Data Management on New Hardware. New York: ACM, 2012: 55-62

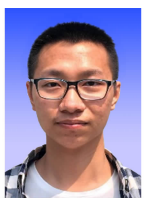
[42] Kocberber O, Grot B, Picorel J, et al. Meet the walkers: Accelerating index traversals for in-memory databases [C] // Proc of the 46th Annual IEEE/ACM Int Symp on Microarchitecture. New York: ACM, 2013: 468-479

[43] Wu L, Barker R J, Kim M A, et al. Navigating big datawith high-throughput, energy-efficient data partitioning [C] // Proc of the 40th Annual Int Symp on Computer Architecture. New York: ACM, 2013: 249-260

[44] Wu L, Lottarini A, Paine T K, et al. Q100: The architecture and design of a database processing unit [C] // Proc of the 19th Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2014: 255-268

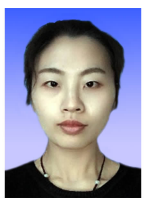
[45] Xi S L, Babarinsa O, Athanassoulis M, et al. Beyond the wall: Near-data processing for databases [C] //Proc of the Int Workshop on Data Management on New Hardware. New York: ACM, 2015: 2-2

[46] Seshadri V, Hsieh K, Boroumand A, et al. Fast bulk bitwise AND and OR in DRAM [J]. IEEE Computer Architecture Letters, 2015, 14(2): 127-131



Wu Linyang, born in 1991. PhD candidate at the Institute of Computing Technology, Chinese Academy of Sciences. His main research interests include computer architecture and computational intelligence.

Luo Rong, born in 1994. MSc candidate at the Institute of Computing Technology, Chinese Academy of Sciences. Her main research interests include computer architecture and computer application.



Guo Xueting, born in 1995. MSc candidate at the Institute of Computing Technology, Chinese Academy of Sciences. Her main research interests include computer architecture and computational intelligence.

Guo Qi, born in 1985. PhD, associate professor with the Institute of Computing Technology, Chinese Academy of Sciences. His main research interests include computer architecture and intelligent computing.