集成CPU-GPU架构上的列存储连接优化技术研究

丁祥武 李子通
(东华大学计算机科学与技术学院 上海 201620)

摘 要 集成多核CPU-GPU架构已经成为计算机处理器芯片的发展方向。利用这种架构的并行计算能力进行数据处理已经成为了数据库领域的研究热点。为了提高列存储系统的查询性能,首先改进了已有协处理机制中的负载分配策略,通过监测数据库系统CPU占用率,动态地为处理器提供合理的数据划分;然后,针对集成多核CPU-GPU架构上的数据预取机制,提出了一种确定预取数据大小的模型,同时,针对GPU访存的特点,进行了GPU访存优化;最后,使用OpenCL作为编程语言,实现了一种集成多核CPU-GPU架构上的列存储排序归并连接算法,并采用提出的方法对连接处理进行优化。实验证明,所提优化策略可以使列存储系统排序归并连接性能提升33%。

关键词 异构芯片,数据预取,查询优化,排序归并连接,OpenCL
中图法分类号 TP392 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2016.11.052

Column-stores Join Optimization on Coupled CPU-GPU Architecture

DING Xiang-wu LI Zi-tong
(School of Computer Science and Technology, Donghua University, Shanghai 201620, China)

Abstract Heterogeneous architecture is the new trend of the development of computer system central processor unit (CPU). Taking advantage of its powerful computer power has been a new research hotspot in database system field. First, in order to enhance the query performance of column-oriented database, we proposed a data partition model which is environmentally sensitive. The data partition model provides optimal data division for every processing unit dynamically by monitoring the CPU occupancy rate. Then, for GPU memory access optimization, we proposed a DFAT estimate model for prefetching. At the same time, we optimized GPU memory access based on coalesced access. We implemented a sort-merge join algorithm on a PC with an integrated CPU-GPU chip, which adopts the out data partition model and our cost model in prefetching. Our strategy is able to distribute data to different processing units automatically, and can make sort-merge join achieve a performance improvement of 33% on coupled CPU-GPU architecture.

Keywords Heterogeneous chip, Data pre-fetching, Query optimization, Sort-merge join, OpenCL

1 引言

根据硬件特点优化数据库查询一直是数据库领域的一个研究热点。近年来计算机处理器的发展方向之一是集成多核GPU。相比于单纯多核CPU和离散多核CPU-GPU架构,集成多核CPU-GPU架构具有较高性能-能耗比以及较低价格-性能比[2,16]。这种架构为细粒度协同并行处理数据库查询提供了硬件支持。集成多核CPU-GPU架构的片上存储结构复杂而且片上处理核心的计算方式差异较大,为提升查询原语操作在集成多核CPU-GPU平台上的性能,需结合平台以及查询原语操作的双重特点,设计合理的查询原语执行方式。目前,在集成多核CPU-GPU架构上的查询优化研究主要集中在利用细粒度协同处理机制优化哈希连接等原语操作[1]和在查询原语操作中减少GPU访存延迟[3]两个方面。

由于集成多核CPU-GPU架构上不同处理器间的通讯方式由总线通信改为了基于共享内存的通信,因此可以通过细粒度协处理方式进一步提升查询原语操作的性能[1,3]。细粒度协同处理方法需将待处理数据按一定比例划分后分配给CPU和GPU处理,数据划分比例是影响负载均衡的关键。系统计算能力是影响数据划分比例的重要因素之一[1]。在实际应用中,数据库软件享有的系统计算资源(如CPU)是变化的。针对变化的可用计算资源占用率,数据分配模型应该能够动态地调整数据划分比例。现有的数据划分模型[1]没有考虑到数据库系统软件对系统资源占用率的变化,从而无法对数据划分比例进行动态调整。

查询原语操作中减少GPU访存延迟的研究主要集中在数据预取机制上[3]。GPU直接访问内存的高访问延迟导致了较多的内存等待(Memory Stall),严重影响了系统性能。数据预取机制通过采用辅助线程提前读入数据的方法,使得数据在被GPU读取之前已经存储于缓存之中,GPU在访问这部分数据时可以直接从缓存中获取,避免GPU从远端的主存访问,一定程度上减少了GPU等待数据的延迟。预取

到稿日期:2015-10-07 返修日期:2016-03-05
丁祥武(1963-),男,博士,副教授,主要研究方向为数据库、列存储、分布式处理技术,E-mail:dingxw@dhu.edu.cn;李子通(1990-),男,硕士生,主要研究方向为列存储、GPU计算。

机制中，需要预取的数据由 WAS(Working-Ahead Set)来确定。WAS是指内存中一个固定大小的区域，用于存放预取向量[10]，每个预取向量分别指向部分被预取数据。本文将WAS中全部预取向量所指向的数据称为一次预取数据量。一次预取数据量的大小是影响预取性能的关键。过大的一次预取数据量可能导致已经被预取的数据在被 GPU 访问前就被操作系统置换出了缓存，过小的一次预取数据量将导致GPU 经常需要从内存获取数据，使得预取机制的作用减弱甚至完全失去。现有集成多核 CPU-GPU 平台上的预取机制沿用为多核 CPU 设计的 WAS 估计模型[3]，没有针对集成多核CPU-GPU 进行相应的调整。

针对以上所述问题，本文提出了一套集成多核异构CPU-GPU 平台上的列存储系统连接原语优化方案，主要的工作包括：

(1)在协同处理方面，提出了一种环境敏感的数据分配模型，通过监测处理器占用率，动态地调整分配给不同处理单元的数据以及处理器划分。

(2)针对集成多核 CPU-GPU 架构上的数据预取，提出了一种相应的 WAS 估计模型。同时，针对 GPU 访存的特点，进行了 GPU 访存优化。

(3)使用 OpenCL 作为编程语言，在集成多核 CPU-GPU 平台上以排序归并连接为例，验证了本文提出的优化策略的有效性。

2 相关工作

2.1 异构平台上的协处理机制

He Jiong 等[1]针对集成 CPU-GPU 平台提出了一种细粒度的协同处理机制。这套机制的基本思想是：对于在集成CPU-GPU 平台上执行的程序，按照原子性将其分割成不同的执行步骤。在不同执行步骤中，将待处理的数据按不同的比例划分后分配给不同的处理器处理。He 提出的协处理机制可以加速集成 CPU-GPU 平台上的哈希连接处理，但并没有给出一套最优数据分配方案，只是依靠实验来确定合适的数据划分比例。

因为数据库查询处理大多是数据密集型的，对数据读取速度的要求大于对计算的要求，GPU 在处理查询时存在内存访问延迟较高的问题，严重地影响了查询处理的性能[2]。针对这一问题，He Jiong 等[3]提出了一种利用缓存的查询协处理方法。这种方法的主要思想是在数据被 GPU 访问之前，将数据装入缓存中，避免缓存缺失，从而减少 GPU 等待数据的时间。在 He 的实验环境下，这种做法使得 TPC-H 查询性能可最高提升 40%。He 提出的数据预取机制可以较好地解决 GPU 读数据的延迟问题，但对于 GPU 写内存并没有相应的优化。同时，He 没有提出确定预取数据量的理论方法，只是通过实验确定了最优预取数据量。

2.2 异构平台上的连接算法

利用硬件特点加速数据库查询处理一直是数据库领域的一个研究热点，其研究内容涵盖了针对多核多处理器的连接算法[4,5,12]、针对 GPGPU 的查询处理算法[6-8]以及针对异构架构的查询处理[1,3]等。这里主要讨论 GPU 上的连接算法以及集成多核 CPU-GPU 架构上的查询优化。

在使用 GPU 进行并行连接方面，He B. S. 等[6]在 GPU 上实现了循环嵌套、排序归并、哈希、无索引循环嵌套等 4 种连接算法，相比于 CPU 上的连接，最大加速比达到了 7 倍。He B. S. 等[6]还提出了利用 GPU 处理数据时的数据分组算法，该算法首先遍历所有数据，计算分配给每个线程的数据对应的输出位置偏移量，之后再一次遍历所有数据并进行分组。He 提出的数据分组算法有效地解决了由于 GPU 不支持动态分配内存而导致的写内存冲突的问题，但该算法存在大量的内存随机读写，并且没有利用合并访问模式(Coalesced Access)，造成了较多的内存等待，浪费了 GPU 的计算周期。陈虎等[16]实现了在离散 CPU-GPU 异构平台上列存储系统的18 种原语操作，并根据 CPU 和 GPU 的特点，分别对这些原语操作进行了优化。针对连接操作，在 CPU 上采用预取和 cache 优化的方法，在 GPU 上则采用将哈希桶大小设置为可以装入 GPU 全部缓存的方式进行优化，这种优化只针对哈希连接，并没有针对其他链接方式进行优化。

在集成多核 CPU-GPU 上的连接方面，He Jiong 等[1]根据其提出的细粒度协同处理机制实现了集成 CPU-GPU 平台上的哈希连接算法，相比于单纯使用 CPU，其性能提升了53%。He Jiong 的研究验证了在集成多核 CPU-GPU 平台上，采用细粒度的协处理机制可以有效地提升哈希链接的执行性能；但对于其他链接操作如何应用细粒度的协处理机制并未研究。

3 环境敏感的协同处理机制

3.1 协同处理机制

这里的协同处理是指集成 CPU-GPU 平台中的计算资源并行协作处理查询，它分为 Off-loading，Data-dividing，Pipeline Execution 3 种类型[1]。我们采用流水线执行(Pipeline Execution)的协处理方式。假设一个操作的输入数据量为 D，S_1 和 S_2 表示完成这个操作需要的两个细粒度步骤。在每一个细粒度步骤执行之前，输入数据(Input)要经过数据划分模块(Data Partition Model)按一定比例划分为两部分。划分后的数据分别交给不同的处理器处理，处理后的数据存储到中间结果缓冲区(Intermediate Data Buffer)中作为下一个细粒度步骤的输入。通过数据预取模块(Pre-Fetching)，GPU 有更多机会从缓存中获取数据，从而减少 GPU 访问数据的延迟。流水线协处理方式的详细过程如图 1 所示。

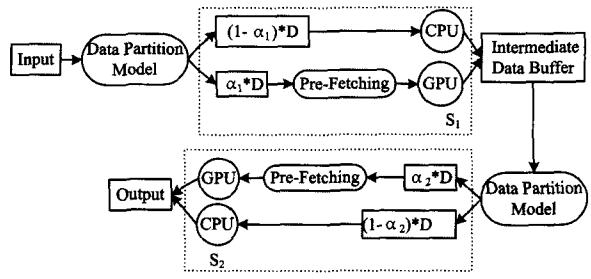

图 1 集成 CPU-GPU 平台上应用协处理

(1)首先，协处理机制将查询原语操作进行细粒度划分。查询原语操作是指数据库中的一些基本操作，如连接、选择、排序、聚集等。对于一个查询原语操作 δ，将其内部完成一定功能的原子操作部分称为一个细粒度步骤 S_i。δ 被划分成 I 个细粒度步骤，这些细粒度步骤按顺序串行执行。对于两个相邻步骤 S_i 和 S_j，使用 $S_i \rightarrow S_j$ 表示两个步骤的执行顺序。

一个查询原语操作的形式化定义为：

$$\delta=\{S_i\rightarrow S_j\mid 0<i<j\leqslant I, j-i=1\} \tag{1}$$

(2)之后，将单个处理器按照计算单元(Compute Unit，CU)划分为多个子处理器，每一个子处理器映射到一个计算单元上。一个异构集成 CPU-GPU 芯片中，CPU 中包含 $|N_{CPU}|$ 个计算单元，GPU 中包含 $|N_{GPU}|$ 个计算单元，记异构集成 CPU-GPU 架构包含的子处理器个数为 $|N_S|$，则有 $|N_S|=|N_{CPU}|+|N_{GPU}|$。划分后，再将子处理器重新组织成两个处理器组，分别记为 R_1，R_2。R_1 和 R_2 分别包含 $|R_1|$ 和 $|R_2|$ ($1\leqslant|R_1|$，$|R_2|\leqslant|NS|$) 个子处理器。同时，数据划分模块将输入数据按一定的比例 α_i 划分，并分别分配给 R_1 和 R_2 处理。协同处理机制中的子处理器分配阶段的示意图如图 2 所示。在实现中，将每一个细粒度步骤中 R_1 与 R_2 的初始分配固定，且初始分配时 R_1 中只包含 CPU 子处理器，R_2 中只包含 GPU 处理器，并通过本文提出的初始数据分配模型来确定每个细粒度步骤中数据分配比例 α_i 的值。

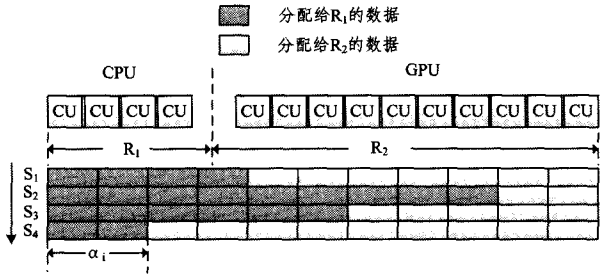

图 2　数据分配示意图

由于真实环境下数据库不可能独享 CPU 资源，在 CPU 占用率发生变化时，原有的负载分配比例可能无法使算法执行性能达到最优。在这种情况下，提出了一种动态数据分配模型，在数据库对 CPU 占用率发生变化时，采用动态调整数据分配比例和子处理器分配的方法，从而优化处理器间的负载平衡，提高数据库性能。

3.2　环境敏感的协处理机制数据分配策略

在协同处理机制中，数据量的分配是影响程序执行效率的关键因素之一。本节将给出一种环境敏感的数据分配模型。该数据分配模型包含两个阶段：初始数据分配阶段和动态数据调整阶段。在初始数据分配阶段，初始化处理器的数据分配。动态数据调整在系统运行时进行，根据监测得到的 CPU 占用率，动态地调整子处理器的分配以及数据分配比例。

3.2.1　初始数据量分配模型

在系统初始化阶段，记一个细粒度步骤中所要处理的数据总量为 M_C，数据初始分配比例因子为 α，分配给 R_1 处理器的数据量为 $\alpha\times M_C$，分配给 R_2 处理器的数据量为 $(1-\alpha)\times M_C$，数据处理阶段所要执行的指令数记为 $|I_{XPU}|$ ($XPU\in\{CPU,GPU\}$)，处理器处理速度峰值记为 P_{XPU} ($XPU\in\{CPU,GPU\}$)，U_C 为 CPU 中一个 OpenCL 本地工作组中的工作项数，U_G 为 GPU 中一个 OpenCL 本地工作组中的工作项数，记处理阶段 R_1 的执行时间为 T_{E1}，则有：

$$T_{E1}=\frac{\alpha\times M_C}{B_{CCPU}}+\frac{|I_{CPU}|}{P_{CPU}}\times\frac{NDRange}{|R_1|\times U_C} \tag{2}$$

若处理阶段 R_2 的执行时间记为 T_{E2}，则有：

$$T_{E2}=\frac{(1-\alpha)M_C}{B_{CGPU}}+\frac{|I_{GPU}|}{P_{GPU}}\times\frac{NDRange}{|R_2|\times U_G} \tag{3}$$

GPU 写回输出缓冲区 O 的时间记为 T_{OG}，则有：

$$T_{OG}=\frac{(1-\alpha)\times M_C}{B_{CGPU}} \tag{4}$$

CPU 写回输出缓冲区的时间记为 T_{OC}，则有：

$$T_{OC}=\frac{\alpha\times M_C}{B_{CCPU}} \tag{5}$$

记处理阶段的总时间为 T_E，则有：

$$T_E=MAX(T_{E1}+T_{OC},T_{E2}+T_{OG}) \tag{6}$$

理想状态下，希望负载在 R_1 与 R_2 上的执行时间相等，则有：

$$T_{E1}+T_{OC}=T_{E2}+T_{OG} \tag{7}$$

通过式(6)、式(7)，可以得到数据初始分配比例 α 的估计值。为简化记为：

$$K_1=\frac{|I_{CPU}|}{P_{CPU}}\times\frac{NDRange}{|R_1|\times U_C} \tag{8}$$

$$K_2=\frac{|I_{GPU}|}{P_{GPU}}\times\frac{NDRange}{|R_2|\times U_G} \tag{9}$$

则数据初始分配比例 α 的估计值为：

$$\alpha=\frac{(B_{CCPU}-B_{CPU})B_{CGPU}}{(B_{CGPU}+B_{CCPU})B_{CPU}}\times\frac{K_2-K_1}{K_1} \tag{10}$$

3.2.2　动态数据调整模型

动态数据调整模型主要用于 CPU 占用率发生变化时改变子处理器的分配以及数据分配比例。其基本思想是：选取 R_2 中的部分 GPU 子处理器分配给 R_1，令其处理 R_1 的部分数据。在本文实验环境中，GPU 共有 16 个子处理器，并将其中 4 个子处理器分配给 R_1。记数据库系统对 CPU 的占用率为 γ，并假定在 R_1 和 R_2 中，只有 R_1 会包含两种不同类型的子处理器，R_2 只包含 GPU 子处理器。在 R_1 中，子处理器的数量记为 $|CU_{1XPU}|$ ($XPU\in\{CPU,GPU\}$)，分配给 R_1 中 CPU 子处理器的数据量记为 $\beta\times\alpha\times M_C$，分配给 R_1 中 GPU 子处理器的数据量记为 $(1-\beta)\times\alpha\times M_C$；将 R_2 所包含的处理器数记做 $|R_2|-4$。记处理阶段 R_1 中 CPU 的执行时间为 T_{ECPU}，则有：

$$T_{ECPU}=\frac{\beta\times\alpha\times M_C}{B_{CCPU}}+\frac{|I_{CPU}|}{\gamma P_{CPU}}\times\frac{NDRange}{|R_1|\times U_C} \tag{11}$$

处理阶段 R_1 中 GPU 的执行时间记为 T_{EGPU}，则有：

$$T_{EGPU}=\frac{(1-\beta)\times\alpha\times M_C}{B_{CGPU}}+\frac{|I_{GPU}|}{P_{GPU}}\times\frac{NDRange}{4\times U_G} \tag{12}$$

R_1 处理阶段的总时间记做 T_{E1}，则有：

$$T_{E1}=MAX(T_{ECPU},T_{EGPU}) \tag{13}$$

处理阶段，记 R_2 的处理时间为 T_{E2}，则有：

$$T_{E2}=\frac{(1-\alpha)M_C}{B_{CGPU}}+\frac{|I_{GPU}|}{P_{GPU}}\times\frac{NDRange}{(|R_2|-4)\times U_G} \tag{14}$$

记处理阶段的总时间为 T_E，则有：

$$T_E=MAX(T_{E1},T_{E2}) \tag{15}$$

根据式(13)、式(15)，可以得到 β 与分配给 R_1 的 GPU 子处理器的关系为：

$$\beta=\frac{(B_{CCPU}-B_{CPU})B_{CGPU}}{(B_{CGPU}+B_{CCPU})B_{CPU}}\times\left(\frac{\gamma K_2\times|R_2|/4}{K_1}-1\right) \tag{16}$$

3.2.3　动态数据分配模型的应用

若将初始分配数据全部存放于同一内存对象中交给 kernel 进行处理，则在数据再分配后，必须暂停 kernel 并重新分配数据，生成新的内存对象。由于无法预知 kernel 中断前的执行位置，中断 kernel 执行可能造成数据错误。为避免这种情况，提出了一种 kernel 执行算法，采用数据分块执行的方式

执行 kernel。所谓数据分块执行，是指在一个细粒度步骤中将待处理数据划分为大小相等的若干块，每块数据分别生成一个独立的 OpenCL 内存对象，并依次交给 kernel 处理。kernel 每次执行只处理一个块的数据，在两次 kernel 执行间隙，根据前一次 kernel 执行时的 CPU 占用率对块内数据分配比例做出调整。kernel 执行算法如算法 1 所示，其流程图如图 3 所示。

算法 1 基于动态数据分配模型的数据调度算法

输入：数据库进程对 CPU 的占用率变化 gama；等待 CPU 处理数据队列头指针 wl _cpu_h、尾指针 wl_cpu_t；等待 GPU 处理的数据队列头指针 wl _gpu_h、尾指针 wl_gpu_t；CPU 内存对象大小 CPU_ MEMOBJ _ LENGTH、GPU 内存对象大小 GPU_ MEMOBJ_LENGTH；GPU 片上存储器大小为 LDS_SIZE；再分配给 GPU 的数据总量为 gpu_redis_data，分配给 CPU 的数据总量为 cpu_redis_data

```
1. while(wl _cpu_h!=NULL&&wl _gpu_h!=NULL){
2.     calculate gama;
3.   if(gama <= α){
4.       calculate new data partition ratio R;
5.       Calculate wl_cpu_t moving OFFSET based on R;
6.       gpu_redis_data ← gpu_redis_data+OFFSET;
7.       tmp ← wl_cpu_t;
8.       move wl_cpu_t forward based on OFFSET;
9.       add data from wl_cpu_t to tmp to GPU data list;}
10.   else if(gama>=α){
11.     calculate new data partition ratio R;
12.     calculate wl_gpu_t moving OFFSET based on R;
13.     cpu_redis_data ← cpu_redis_data+OFFSET;
14.     tmp ← wl_gpu_t;
15.     move wl_gpu_t forward based on OFFSET;
16.     add data from wl_cpu_t to tmp to CPU data list;}
17. build_cpu_memobj();
18. wl_cpu_h ← wl_cpu_h+CPU_MEMOBJ_LENGTH;
19. build_gpu_memobl();
20. wl_gpu_h ← wl_gpu_h+GPU_MEMOBJ_LENGTH−4×LDS_
    SIZE;
21. adjust gpu_redis_data and gpu_redis_data;
22. exe_kernel();
23. wait();}
```

第 1 行，判断数据队列尾部指针是否为空，若为空表明所有数据已处理完毕；第 4－14 行，根据 CPU 占用率变化调整数据划分比例，并将重新划分后的数据加入相应的数据队列；第 16－18 行生成内存对象，调整数据队列，并执行 kernel。

除此之外，还有几点需要说明：

1）生成 GPU kernel 内存对象时，将单一线程多次访问的数据存放在同一内存对象中。由于这部分数据会被拷贝到 GPU 片上存储器中，出于充分利用片上存储器的考虑，将内存对象的大小 *GPU_MEMOBJ_LENGTH* 设置为 OpenCL 本地工作组数×*LDS_SIZE*。数据再分配后，再分配给 GPU 的数据首地址偏移量为 *wl_gpu_−gpu_redis_data*。在生成内存对象时，取大小为 4×*LDS_SIZE* 的再分配数据（对应 4 个 GPU 子处理器）和大小为（*GPU_MEMOBJ_LENGTH*−4）×*LDS_SIZE* 的初始分配数据生成同一内存对象。

2）CPU 占用率的统计将在 kernel 执行期间进行，由单一 CPU 线程统计 kernel 运行期间数据库进程对 CPU 的占用率并计算其平均值。

3）实现中通过线程组 ID 区分 GPU 子处理器。OpenCL 线程组数量被设置为与 GPU 计算单元数相等，每一个线程组将被映射到一个 GPU 计算单元上。在执行时，ID 小于 4 的线程组将被用来处理再分配的数据，从而实现子处理器的划分。

算法中，数据待执行队列用来管理待处理数据，其头指针指向下一次 kernel 需要处理的数据首地址，尾指针指向其管理的数据尾部。在实现过程中，宿主机程序共维护 4 个数据队列，用来管理初始化时分配的数据。其中，每个处理器对应两个待执行队列，每个队列分别管理参与连接的两个连接关键字的数据。

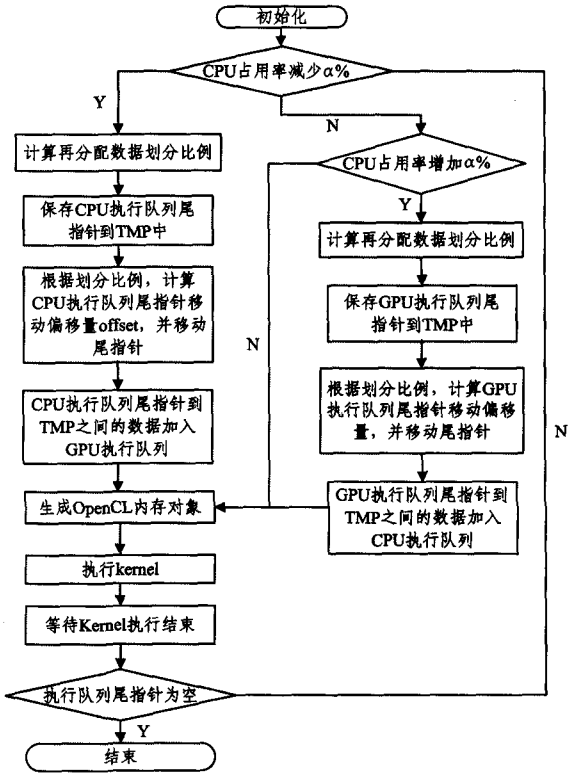

图 3 执行算法流程图

4 预取机制

4.1 预取

GPU 访问内存存在较高的延迟，出现缓存缺失（Cache Miss）将会极大地影响 GPU 性能[7]。根据现有的研究[9,10]，本文采取一种数据预取策略以减少 GPU 的访存延迟，预取算法如算法 2 所示。

符号说明 1：*temp* 为预取缓冲变量；p_i' 为指针，指向当前需要被预取的数据；*cache_line_size* 为当前系统中一个缓存行的大小；⟨*pi*，*size*⟩表示 WAS 中的一个预取向量，p_i 为预取数据首地址，*size* 为其大小。

算法 2 Pre-fetching

输入：预取向量集合 WAS，WAS={⟨p_i，size⟩|1≤i≤|T_W|}

输出：被预取的元组

```
# define CACHE_LINE cache_line_size
1. for every ⟨p_i, size⟩ in P
2. do
3.       p_i' = p_i
4. end
5. while(P!=NULL)
```

```
6. do
7.   for every ⟨p_i, size⟩ in p
8.   do
9.     temp+=*(byte*)p_i';
10.    p_i'+=CACHE_LINE;
11.    if p_i'==p_i+size+1 //⟨p_i, size⟩已被全部预取
12.    do void ⟨p_i, size⟩
13.        refill ⟨p_i, size⟩
14.    end
15.  end
16. end
```

假设需要预取的数据总量为 M，使用 CPU 线程 0（记为 T_{C0}）作为数据预取线程，其他 CPU 线程和 GPU 线程作为工作线程，记为 T_W。本文用 $|T_W|$ 表示 T_W 中包含的线程组的数量（即 OpenCL 本地工作组的数量）。T_{C0} 将为每一个线程组 i 维护一个对应的预取向量 $\langle p_i, size\rangle (1 \leqslant i \leqslant |T_W|)$，$p_i$ 是一个指针变量，指向预取数据的首地址，$size$ 表示一次预取中，需要预取的数据量大小。所有预取向量的大小 $size$ 均被设置成同一值，这是出于以下两个考虑：1）GPU 线程在执行时，一个线程会独享一个 SIMD 核，故所有线程的计算速度是相近的；2）所有 GPU 线程将处理相同的负载且处理的数据量大小基本相同。预取向量的集合称为 WAS，一次预取数据量 $|M_P|$ 是指 WAS 中所有预取向量指向的内存区域大小的总和，其值为 $|M_P| = |T_W| \times size$。$T_{C0}$ 分多次进行数据预取，每次预取数据量为一次预取数据量 $|M_P|$，预取次数为 $M/|M_P|$。设置一个 $Temp$ 变量用于临时存储 T_{C0} 读取的数据，每一次数据预取中，T_{C0} 依次读取每个预取向量指向的数据，每次读取一个字节的数据到 Temp 中。这里需要指出，CPU 访存采取内存对齐的方式，T_{C0} 访问同一预取向量指向的内存区域时，连续两次访问之间的步长应为内存存取粒度（Memory Access Granularity）[14]。这样做避免了对同一缓存行中数据的反复读取，减少了 T_{C0} 读取内存的次数。第一次预取完成后，T_W 开始工作，T_{C0} 则开始第二次预取。从第二次预取开始，T_{C0} 将会在完成预取后进入等待状态，等待 T_w 完成其负载后，再开始下一次预取。T_{C0} 和 T_W 将反复进行以上步骤，直到所有数据被处理完成。

4.2 WAS 估计模型

一次预取数据量是影响预取性能的关键，理想情况下，希望数据在被 GPU 读取时还没有被置换出缓存。设 T_{C0} 一次读取的数据量为 $|M_P|$。T_{C0} 读取大小为 $|M_P|$ 的数据到 WAS 中的时间应当接近于工作线程组 T_W 处理数据的速率，否则将会造成 T_W 等待，浪费计算时间周期。为确定合理的预取数据量，提出了一种预取数据估计模型，具体内容如下。

在数据预取阶段，用 CPU 的一个固定的线程执行数据预取，一次预读取的数据量记为 $|M_P|$，一个 CPU 线程访问内存的最大带宽记为 B_{CPU}，则一次预取所消耗的时间 T_P 为：

$$T_P = M_P / B_{CPU} \tag{17}$$

理想状态下，希望处理阶段总时间 T_E 大于或等于预取时间，则有：

$$T_E \geqslant T_P \tag{18}$$

根据式(6)－式(9)、式(17)、式(18)，得到 $|M_P|$ 的估计模型为：

$$|M_P| = \frac{K_1 * B_{CPU} * B_{CCPU}}{B_{CCPU} - \frac{B_{CCPU} - B_{CPU}}{B_{CGPU} + B_{CCPU}} * B_{CGPU} * \frac{K_2 - K_1}{K_1}} \tag{19}$$

5 排序归并连接算法

为应用协处理机制，首先需要将原有的算法分解为不同的细粒度步骤。在实现中，将并行排序归并连接算法按照协处理机制的要求进行细粒度分解，分解后的算法在 5.1 节与 5.2 节中给出。分解后，算法的各个细粒度步骤在 CPU 和 GPU 两个处理器上并行执行，每个处理器处理不同的数据。每个处理器处理的数据量将根据本文提出的数据分配模型确定。并行的排序归并连接算法分为 3 个步骤，分别是数据的排序、分组与归并连接。为保证排序性能，排序部分采用双调排序算法[12]，下面着重介绍分组部分和归并连接部分。

5.1 数据分组算法

本文提出一种用于 GPU 的数据分组算法，并将此算法按照协处理机制的要求进行细粒度的划分，划分后的算法如算法 3 所示。算法 3 中涉及到的前缀和运算如算法 4 所示。分组算法的时间复杂度为 $O(n)$。算法 3 中，$\{S_i | i=1,2,3,4\}$ 表示分组部分的 4 个细粒度步骤。

符号说明 2：k 为线程在一个线程组内的序号；$local\_size$ 为线程组的大小；j 为线程组的序号；g 为线程的全局序号；$p[H]$ 为一个线程的输出空间，用来存储分组后数据在输出数组中的偏移量。

算法 3 Partition(int * Input, int * Output, int * L, int Size_L, int size)

输入：属性 Input，L 的大小 Size_L，Input 的大小 size

输出：分组后的数据 Output，每个组起始位置在 output 中起始位置的偏移量数组 L

步骤：

S_1 p←each thread counts it's input data amount

S_2 L[g * i+j * local_size+k]=p;

S_3 L←prifix_sum(L);

S_4 Do partition, each thread's offset in output space is L[g * i+j * local_size+k];

实现中，对于要进行连接的表 R 和 S，将连接关键字 $R.a$ 和 $S.b$ 作为输入。分组后，$R.a$ 和 $S.b$ 中相对应的一个或几个分组将被调度到同一个处理单元上。$R.a$ 与 $S.b$ 对应分组的计算方法为：1）取 $S.b$ 中一个分组的第一个元组 $first$ 和最后一个元组作 $last$；2）探查 $R.a$，若 $R.a$ 的分组中至少存在一个元素位于区间 $[first, last]$ 内，则这两个分组即为对应分组。对于已排序的元组，分组算法可以保证分组后组内数据仍保持原有顺序，据此，本文在 5.3 节中提出了一种 GPU 访存优化方法，此方法中，GPU 可以采用合并访问模式[12]顺序读取组内元素，避免随机读取内存带来的延迟问题。

算法 4 PrefixSum(int[] S, int S_length)

```
输入：数组 s，数组长度 S_Length
输出：数组 s
1. while(i<S_Length)
2.     if i equals 0
3.         s[i]←0
4.     else
5.         tmp_1←S[i]
6.         s[i]←Tmp_2+s[i−1]
7.         tmp_2=tmp_1
```

5.2 归并连接部分

归并连接部分的实现同样使用了协同处理机制和数据预取机制，连接算法如算法 5 所示。数据在经过分组后，被分配

到每个处理单元上，每个处理单元并行执行连接算法。将连接算法分成4个细粒度步骤。S_1 的作用是计算一个分组中，属性 $S.b$ 中与 ts 等值的元组的数量。S_2 的作用是在对应的 $R.a$ 的分组中，将取值与 ts 相等的元组 tr 与属性 $S.b$ 的对应分组中的所有等值元组进行连接。

符号说明3：tr 为指向 R 的指针；ts 为指向 S 的指针；$temp\_s\_offset$ 为临时指针变量。

算法5 Join(int * R.a,int * R.b,int[] h,int s_offset, int size_h)

输入：属性 R.a,S.b,记录分组偏移量的数组 h,h 中 S 分组偏移量的起始位置为 s_offset,h 的大小为 size_h

输出：连接后的元组 t

步骤：

```
S1  (1)ts←s_offset;
    (2)temp_s_offset=&ts;
    (3)c ←calculate tuples which value equals ts in S.b;
S2  (4)  while tr<=ts
    (5)    if(tr==ts)
    (6)        join ts and tr;
    (7)    else
    (8)        &tr++;
S3  (9)  while ts<=s_offset+size_h
    (10)  repeat S1、S2;
```

5.3 其他优化

GPU线程对内存的访问是以 wavefront[1] 为单位统一访问的，当线程组中的所有线程顺序访问一段连续的内存单元时，GPU会把所有线程的访存请求合并成一次访存操作，这种访问方式被称为合并访问模式[1]。合理使用合并访问模式可以极大地减少GPU访存操作次数，提高GPU访存效率[1]。本文采用合并访问模式，将分组算法在GPU上的实现进行了优化。首先，规定每个线程组的每次内存访问都保证组内线程ID按顺序依次访问相邻内存单元；其次，为避免多个线程组之间出现访存冲突，设置次数指示变量 E_i 来记录线程组的访存次数，每个线程组维护一个私有指示变量，线程组内线程共享这个变量。假设数据总量为 M，线程组数为 N，线程组的序号是 K，则线程组的下一次访存位置偏移量是 $(M/N)\times(E_i-1)+(K-1)\times N$。应用合并访问的示意图如图4所示。这种方式实现了合并访问模式，同时避免了各个线程组之间的访存冲突。

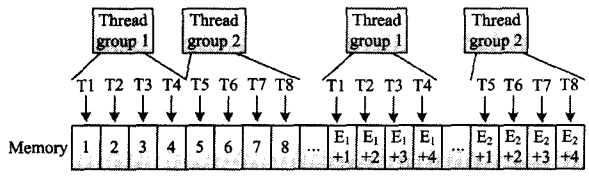

图4 采用合并访问模式时GPU访存优化

6 实验

6.1 实验环境

实验软件环境使用 OpenCL 1.0、AMD APP SDK2.9、Intel OpenCL RunTime、实验室开发的列存储系统 DWMS 1.0。硬件环境使用 Intel(R) Core(TM) i5-3210M CPU@2.5 GHz，计算核心数为2，最大线程数为4，缓存大小为3MB。集成GPU为 Intel HD Graphics 4000 GPU@650MHz，计算核心数为16，最大线程数为512。8GB DDR3@1600MHz 内存。CPU的内存带宽为12.27GB/s，缓存带宽为17.65GB/s(单通道)。GPU的内存带宽为8.51GB/s，缓存带宽为12.24GB/s。操作系统为 Windows 7，开发环境为 Microsoft(R) Visual Studio 2010。实验使用SSB数据集，大小为300M。

6.2 实验结果与分析

本文共完成了4组实验：1)在连接算法的分组阶段，改变数据分配比例 α，得出不同数据分配比例时排序归并连接的实际运行时间，并将其与本文提出的初始数据分配模型得到的估计时间进行对比，验证初始数据分配模型的有效性。2)通过在实际运行时选取不同大小的一次预取数据量，将实际运行时间与本文提出的数据分配模型得出的估计时间作对比，验证开销模型的准确性。3)通过对比采用预取机制和未采用预取机制的排序归并连接算法，验证预取机制是否可以提高排序归并连接性能。4)改变CPU占用率，验证动态数据分配模型的有效性。

6.2.1 初始数据分配模型

本文选取分组部分细粒度步骤 S_1 作为实验对象，选取不同的数据分配比例 α，将 S_1 的实际执行时间与我们的分配模型的执行时间进行对比，实验结果如图5所示。从图5中可以看出，所提的数据分配模型的运行情况接近于实际运行的情况，且变化趋势保持一致。当 $\alpha=37\%$ 时，S_1 的执行时间最短。这是因为GPU相比于CPU更加适合处理算数运算，同时 S_1 涉及到的算数运算较多，逻辑运算较少。

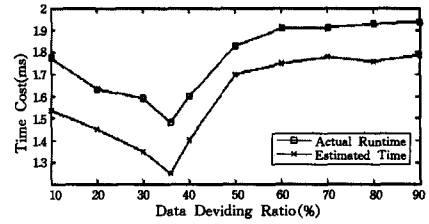

图5 数据分配模型

6.2.2 每次预取数据量分配模型

选取分组部分细粒度步骤 S_1 作为实验对象，固定数据分配比例为 $\alpha=37\%$，改变一次预取数据量 $|M_P|$，实验结果如图6所示。从图6中可以看出，所提数据分配模型的结果接近于实际的运行时间，且具有相同的趋势。根据式(19)，通过计算可以知道每次预取数据量大小设置为0.8M时 S_1 的执行时间最短。实验结果表明，一次预取数据量并非越大越好。出现这种情况是由于以下原因：当每次预取数据量的大小设置过大时(大于0.8M)，预取数据的时间会较长，工作线程组需要等待预取结束才可以处理数据。当每次预取数据量设置过小(小于0.8M)时，工作线程组需要从内存中直接读取部分数据，造成了较长的内存等待时间。

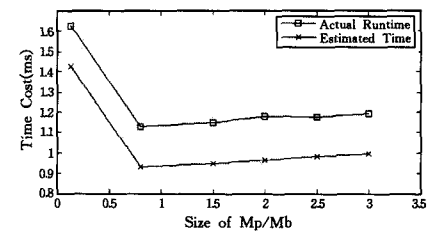

图6 一次预取数据量大小对执行时间的影响

6.2.3 预取机制

在3种情况下实现了排序归并连接算法，分别是：1)在多

核 CPU 平台上，未使用预取机制；2)在集成多核 CPU-GPU 平台上，采用协处理机制，未使用预取机制；3)在集成多核 CPU-GPU 平台上，采用协处理机制，使用预取机制。实验结果如图 7 所示，与纯 CPU 平台上的排序归并连接相比，集成 CPU-GPU 环境下的协处理排序归并连接可以达到 1.2 倍的加速比，采用预取机制后，加速比达到了 1.3 倍。

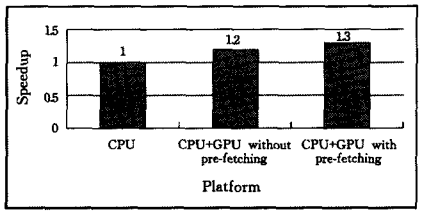

图 7　3 个平台下选择排序连接执行时间的加速比

6.2.4　环境敏感的动态数据分配模型

选取分组部分细粒度步骤 S_1 作为实验对象，选取固定的数据分配比例为 37%，在需要重新分配子处理器时，将分配给 R_1 的子处理器数固定为 4。通过采用开启额外进程占用 CPU 的方式，改变数据库进程的 CPU 占用率，对比采用动态数据分配模型和未采用动态数据分配模型时执行时间的差异，以此说明动态数据分配模型的有效性。执行时间对比如图 8 所示，图中的横坐标表示额外进程的 CPU 占用率，加速比如图 9 所示。可以看出，采用动态数据分配模型的平均加速比为 1.11 倍，最大加速比可以达到 1.23 倍。从图中可以看出当额外进程 CPU 占用率大于 77%后，采用动态数据分配模型的执行时间会大于未采用时动态数据分配模型的执行时间。这是因为再分配给 GPU 的数据过多，导致 GPU 的 kernel 执行时间远大于 CPU 的 kernel 执行时间。因此，动态数据分配模型只在除数据库进程外的其他进程对 CPU 的占用率小于 75%时有效。

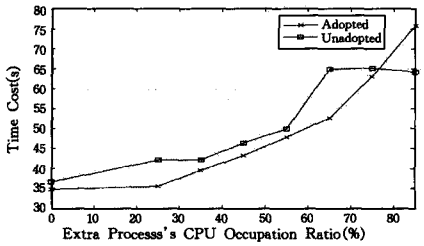

图 8　采用和不采用动态数据分配模型 S1 的执行时间

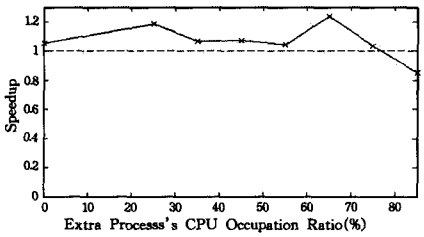

图 9　采用动态数据分配模型时执行的加速比

6.2.5　环境敏感的动态数据分配模型对比论证

本节将文献[1]提出的数据分配模型与本文提出的环境敏感的动态数据分配模型进行对比，用来论证本文提出的数据分配策略的有效性。本文将文献[1]提出的数据分配模型称作方法 A。本文选取哈希连接中的构建阶段作为实验对象，分别采用方法 A 与本文提出的数据分配模型确定每个细粒度步骤中的数据分配比例，实验结果如表 1、表 2 所列。由表 1 可知，在数据库占用率不发生较大变化时，采用本文提出的数据分配模型计算出的数据分配比例与采用方法 A 得出的数据分配比例相近；由表 2 可知，在 CPU 占用率较低时(占用比为 50%)，通过本文提出的数据分配模型计算得出的数据分配比例发生了变化。

表 1　CPU 占用率为 100%时数据分配比例

	方法 A		环境敏感的动态数据分配模型	
	CPU	GPU	CPU	GPU
b1	0%	100%	0%	100%
b2	4%	96%	4%	96%
b3	60%	40%	61%	39%
b4	22%	78%	22%	78%

表 2　CPU 占用率为 50%时数据分配比例

	方法 A		环境敏感的动态数据分配模型	
	CPU	GPU	CPU	GPU
b1	0%	100%	0%	100%
b2	4%	96%	0%	100%
b3	60%	40%	34%	66%
b4	22%	78%	11%	89%

哈希连接构建阶段的运行时间如图 10 所示，由图可知，当 CPU 占用率不发生变化时，采用方法 A 和采用本文提出的数据分配模型得到的运行时间相近；当 CPU 占用率变化时，采用本文提出的方法运行时间较短，表明本文提出的方法可以有效地缩短 CPU 占用率减少时构建阶段的运行时间。

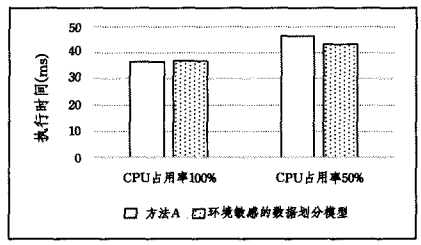

图 10　构建阶段执行时间

结束语　本文首先提出了一种集成多核 CPU-GPU 架构上环境敏感的协处理机制负载分配策略，通过监测系统处理器占用率动态地为协处理机制提供合理的数据和处理器划分。同时，针对集成多核 CPU-GPU 平台上的数据预取机制，提出了一种确定预取数据大小的模型；并利用我们的策略，实现了列存储系统上的排序归并连接算法。实验结果表明，本文实现的列存储系统中的排序归并连接算法的性能相比于原系统提升了 33%；同时，在 CPU 占用率发生变化时，环境敏感的负载分配策略可以有效地调整子处理器负载分配比例，提高了系统性能。未来的工作将继续探究集成多核 CPU-GPU 平台上协处理机制在查询原语操作上的应用，将重点放在如何将协处理机制与聚集操作及排序操作相结合上。同时，在 GPU 访存优化方面，还将研究如何通过软件方式隐藏或避免 GPU 存在的多线程访问片上的存储器冲突问题。

参 考 文 献

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

(下转第 308 页)

参 考 文 献

[1] Gonzalez R C. Digital image processing using MATLAB(3rd Edition)[M]. Beijing: Tsinghua University Press, 2013: 211(in Chinese)
冈萨雷斯. 数字图像处理 MATLAB 版(第三版)[M]. 北京: 清华大学出版社, 2013: 211-213

[2] Flusser J, Zitova B, Suk T. Moments and moment invariants in pattern recognition[M]. John Wiley & Sons, 2009

[3] Mukundan R, Ong S, Lee H P A. Image analysis by Tchebichef-moments[J]. IEEE Trans. Image Process., 2001, 10(9): 1357-1364

[4] Yap P T, Paramesran R, Seng-Huat O. Image analysis by Krawtchoukmoments[J]. IEEE Trans. Image Process., 2003, 12(11): 1367-1377

[5] Zhu Hong-qing, Shu Hua-zhong, Liang Jun, et al. Image analysis by discrete orthogonal Racahmoments[J]. IEEE Trans. Signal Process., 2007, 87(4): 687-708

[6] Pew-Thian Y, Paramesran R, Seng-Huat O. Image Analysis Using Hahn Moments[J]. IEEE Trans. Pattern Analysis and Machine Intelligence, 2007, 29(11): 2057-2062

[7] Zhu Hong-qing, Shu Hua-zhong, Liang Jun, et al. Image analysis by discrete orthogonal dual Hahn moments[J]. Pattern Recognition Letters, 2007, 28(13): 1688-1704

[8] Karakasis E G, Papakostas G A, Koulouriotis D E, et al. Generalized dual Hahn moment invariants[J]. Pattern Recognition, 2013, 46(7): 1998-2014

[9] Priyal S P, Bora P K. A robust static hand gesture recognition system using geometry based normalizations and Krawtchoukmoments[J]. Pattern Recognition, 2013, 46(8): 2202-2219

[10] Zhang Li, Qian Gong-bin, Xiao Wei-wei, et al. Geometric invariant blind image watermarking by invariant Tchebichefmoments[J]. Optics Express, 2007, 15(5): 2251-2261

[11] Prattipati S, Ishwar S, Swamy M N S, et al. A fast 8× 8 integer Tchebichef transform and comparison with integer cosine transform for image compression[C]// 2013 IEEE 56th International Midwest Symposium on Circuits and Systems(MWSCAS). Columbus: IEEE Press, 2013: 1294-1297

[12] Hung A C, Meng H Y. Optimal quantizer step sizes for transform coders[C]// 1991 International Conference on Acoustics, Speech, and Signal Processing. IEEE, 1991: 2621-2624

[13] Wu S W, Gersho A. Rate-constrained picture-adaptive quantization for JPEG baseline coders[C]// IEEE International Conference on Acoustics, Speech, and Signal Processing. IEEE, 1993: 389-392

[14] Ratnakar V, Livny M. An efficient algorithm for optimizing DCT quantization[J]. IEEE Transactions on Image Processing, 2000, 9(2): 267-270

[15] Reininger R, Gibson J D. Distributions of the Two-Dimensional DCT Coefficients for Images[J]. IEEE Transactions on Communications, 1983, 31(6): 835-839

[16] Yang En-hui, Sun Chang, Meng Jin. Quantization table design revisited for image/video coding[J]. IEEE Transactions on Image Processing A Publication of the IEEE Signal Processing Society, 2014, 23(11): 4799-4811

[17] Xiao Bin, Lu Gang, Wang Guo-yin, et al. Image Compression Based on Discrete HermitePolynomials[J]. Computer Science, 2015, 42(11A): 140-141, 154(in Chinese)
肖斌, 陆刚, 王国胤, 等. 基于离散 Hermite 多项式的图像压缩[J]. 计算机科学, 2015, 42(11A): 140-141, 154

[18] Kodak: 24 PNG images[OL]. http://r0k.us/graphics/kodak

[19] CVG-UGR image database[OL]. http://decsai.ugr.es/cvg/dbimagenes/g256.php

(上接第 271 页)

[2] Lu Feng-shun, Song Jun-qiang, Yin Fu-kang, et al. Survey of CPU/GPU Synergetic Parallel Computing[J]. Computer Science 2011, 38(3): 5-9(in Chinese)
卢风顺, 宋君强, 银福康, 等. CPU/GPU 协同并行计算研究综述[J]. 计算机科学, 2011, 38(3): 5-9

[3] He J, Zhang S, He B. In-Cache Query Co-Processing on Coupled CPU-GPU Architectures[J]. Proceedings of the VLDB Endowment, 2014, 8(4): 329-340

[4] Albutiu M C, Kemper A, Neumann T. Massively parallel sort-merge joins in main memory multi-core database systems[J]. Proceedings of the VLDB Endowment, 2012, 5(10): 1064-1075

[5] Balkesen C, Alonso G, Teubner J, et al. Multi-core, main-memory joins: Sort vs. hash revisited[J]. Proceedings of the VLDB Endowment, 2013, 7(1): 85-96

[6] He B, Lu M, Yang K, et al. Relational query coprocessing on graphics processors[J]. ACM Transactions on Database Systems (TODS), 2009, 34(4): 2939-2965

[7] Fang R, He B, Lu M, et al. GPUQP: query co-processing using graphics processors[C]// Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data. ACM, 2007: 1061-1063

[8] Heimel M, Saecker M, Pirk H, et al. Hardware-oblivious parallelism for in-memory column-stores[J]. Proceedings of the VLDB Endowment, 2013, 6(9): 709-720

[9] Chen S, Ailamaki A, Gibbons P B, et al. Improving hash join performance through prefetching[J]. ACM Transactions on Database Systems(TODS), 2007, 32(3): 116-127

[10] Zhou J, Cieslewicz J, Ross K A, et al. Improving database performance on simultaneous multithreading processors[C]// Proceedings of the 31st international conference on Very large data bases. VLDB Endowment, 2005: 49-60

[11] Peters H, Schulz-Hildebrandt O, Luttenberger N. Parallel Processing and Applied Mathematics[M]. Springer Berlin Heidelberg, 2010: 403-410

[12] Hardavellas N, Pandis I, Johnson R, et al. Database servers on chip multiprocessors: Limitations and opportunities[C]// Proceedings of the Biennial Conference on Innovative Data Systems Research. 2007: 79-87

[13] He B, Yang K, Fang R, et al. Relational joins on graphics processors[C]// Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. ACM, 2008: 511-524

[14] Rentzsch J. Data alignment: Straighten up and fly right[R/OL]. (2005-02-08). https://www.Ibm.com/developerworks/library/pa-dalign/pa-dalign-pdf.pdf

[15] Chen Hu, Chen Si-tong, Li Guan-zhao, et al. Column-Storage database parallel query on heterogeneous computing platforms[J]. Journal of Computer Research and Development, 2012, 49(Suppl.): 65-71(in Chinese)
陈虎, 陈思桐, 李观钊, 等. 异构计算平台上列存储数据库的并行查询技术研究[J]. 计算机研究与发展, 2012, 49(Suppl.): 65-71

[16] Gaster B R, Howes L, Kaeli D R, et al. OpenCL 异构计算[M]. 张云泉, 张先轶, 龙国平, 等译. 北京: 清华大学出版社, 2012: 42-44