基于图形处理器的并行方体计算

周国亮[1),2),3)]　陈　红[1),2)]　李翠平[1),2)]　王　珊[1),2)]　郑　涛[1),2)]

[1)](教育部数据工程与知识工程重点实验室(中国人民大学)　北京　100872)

[2)](中国人民大学信息学院　北京　100872)

[3)](保定电力职业技术学院信息系　河北 保定　071051)

摘　要　方体(cube)计算是数据仓库和联机分析处理(Online analytical processing,OLAP)领域的核心问题,如何提高方体计算性能获得了学术界和工业界的广泛关注,但目前大部分方体算法都没有考虑最新的处理器架构.近年来,处理器从单一计算核心进化为多个或许多个计算核心,如多核CPU、图形处理器(Graphic Processing Units,GPU)等.为了充分利用现代处理器的多核资源,该文提出了基于GPU的并行方体算法GPU-Cubing,算法采用自底向上、广度优先的划分策略,每次并行完成一个*cuboid*的计算并输出;在计算*cuboid*过程中多个分区同步处理,分区内多线程并行.GPU-Cubing算法适合GPU体系结构,并行度高.与BUC算法相比,基于真实数据集的完全方体计算可以获得一个数量级以上的加速比,冰山方体获得至少2倍以上的加速.

关键词　图形处理器;并行方体计算;实时数据仓库;联机分析处理

中图法分类号　TP311　　　**DOI号**:10.3724/SP.J.1016.2010.01788

Parallel Data Cube Computation on Graphic Processing Units

ZHOU Guo-Liang[1),2),3)]　CHEN Hong[1),2)]　LI Cui-Ping[1),2)]　WANG Shan[1),2)]　ZHENG Tao[1),2)]

[1)](*Key Laboratory of Data Engineering and Knowledge Engineering (Renmin University of China) of Ministry of Education*　100872)

[2)](*School of Information, Renmin University of China, Beijing*　100872)

[3)](*Department of Information, Baoding Electric Power Vocation & Technology College, Baoding, Hebei*　071051)

Abstract　In the fields of data warehousing and OLAP, data cube computation is a core problem, and how to improve performance of cube computation is constant pursuit of academy and industry, but most of cube algorithms are designed without considering modern processor architecture. In recent years, processor has evolved from one core to multi-core or many-core such as multi-core CPUs and GPU. To take full advantage of modern multi-core processors, this paper proposes parallel data cube computation algorithm called GPU-Cubing. GPU-Cubing adopts bottom-up and breadth-first partitioning order, and one *cuboid* is parallelism computed and output. Multiple partitions are processed simultaneously and multi thread parallel executed in each partition. GPU-Cubing is consistent with GPU architecture and high parallelism. GPU-Cubing achieves a speedup over BUC up to an order of magnitude in full cube computation and at least 2 folds in iceberg cube computation in real dataset.

Keywords　graphic processing units; parallel cube computation; real-time data warehouse; on-line analytical processing

---

收稿日期:2010-08-22.本课题得到国家"八六三"高技术研究发展计划项目基金(2008AA01Z120)、教育部高等学校博士学科点专项科研基金项目基金(20090004110002)资助.**周国亮**,男,1978年生,博士研究生,主要研究方向为基于新硬件的数据仓库和OLAP技术.E-mail:zhouguoliang@ruc.edu.cn.**陈　红**,女,1965生,教授,博士生导师,主要研究领域为高性能计算和无线传感器网络.E-mail:chong@ruc.edu.cn.**李翠平**,女,1971生,博士,博士生导师,主要研究领域为社会网络和图数据挖掘.**王　珊**,女,1944生,教授,博士生导师,主要研究领域为高性能可扩展数据库和视频数据库等.**郑　涛**,男,1987年生,硕士研究生,主要研究方向为基于新硬件的数据挖掘和OLAP.

1 引 言

在联机分析处理(Online Analytical Processing,OLAP)和数据仓库领域,方体计算是一个核心问题.如何提高方体计算效率获得了学术界和工业界的广泛关注[1-8].虽然如此,根据 BI Survey 调查,用户抱怨最多的依然是 OLAP 的性能问题.而且传统数据仓库存储的是具有一定时效性的历史数据,在当今瞬息万变的商业社会中,决策者要把握稍纵即逝的商机,需要对实时数据进行实时战术战略分析.一些学者提出了实时数据仓库的概念(real-time data warehouse),实时数据仓库对方体计算提出了更高的要求.

OLAP 是数据密集和计算密集型应用,面对大数据量大计算量完成实时计算是一项非常有挑战性的工作.文献[5]通过近似计算来完成实时聚集计算,但近似计算在某些情况下不能满足用户精确性的需求.文献[6-7]利用 PC Cluster 或并行机,提出了一系列并行算法,期望实现实时计算.但并行机或 Cluster 造价昂贵且随着节点数增加,管理维护困难.文献[8]基于现代 CPU 的体系结构,提出了缓存敏感的方体算法,获得了较高的加速比,但 CPU 有限的带宽和计算能力,与实时计算的要求存在较大的差距.实现实时计算,设备的计算能力和带宽是一个瓶颈.利用新硬件的高性能,提高方体计算的效率,为实现实时数据仓库提供有益的尝试是本文的出发点.

近年来,处理器体系结构发生重大改变,即由单一计算核心发展为多个计算核心,而且核心的数量在持续增长.现在普通的机器上都配置有多核 CPUs 和图形处理器 GPU(Graphic Processing Units).然而目前大多数方体算法并没有考虑处理器的多核架构.CPU 与 GPU 相比,CPU 作为一种通用计算资源,大部分寄存器用于缓存数据,而不是计算;同时,多个核心共享有限的带宽,进一步阻碍了性能的提高.而 GPU 具有 10 倍于 CPU 的带宽和计算能力,并以每年 2.8 倍的速度增长.本文提出了基于 GPU 的并行方体算法.

基于 GPU 的硬件特征和方体计算的特点,本文提出了基于 GPU 的并行方体算法 GPU-Cubing.算法所需基础数据每列单独存储和处理,减少内存 I/O,提高了 Cache 利用率.算法采用自底向上和广度优先的分区策略,自底向上可以利用剪枝策略,减少不必要的计算,并共享分区的结果;广度优先保证每次并行处理一个 *cuboid*,结果顺序输出,多个分区同步处理,分区内多线程并行.通过详尽的实验分析,GPU-Cubing 算法相比传统的方体算法获得了一个数量级的加速比.

本文第 2 节简单介绍相关工作;第 3 节讨论基于 GPU 的并行操作定义、GPU-Cubing 算法及内存特征;第 4 节讨论算法中主要采取的优化技术及算法的扩展;第 5 节给出算法的实验验证及分析;最后对全文的内容进行总结.

2 相关工作

2.1 方体计算算法

给定一个包括 n 个维(属性)和 m 个测度的关系 $R(A_1,A_2,\cdots,A_n,M_1,M_2,\cdots,M_m)$,方体计算是计算 2^n 个 Group bys.每个 Group by 称为一个 *cuboid*,包含 n 个维的 *cuboid* 称为 *base cuboid*.如果用户对每个 *cuboid* 指定了条件,比如 $count(*)\geq 100$,这样的方体称为冰山方体(Iceberg Cube).目前提出了很多方体计算算法[1-8],其中包括 Multiway[1]、BUC[2]、Star-Cubing[3]、MM-Cubing[4] 和 CC-Cubing[8] 等.

BUC 算法是稀疏数据集上计算方体的有效算法,算法采用自底向上的分区策略,同时利用剪枝操作提高算法效率.

CC-Cubing 是目前最新的方体计算算法,算法充分利用现代 CPU 的硬件特征,提高 Cache 利用率.算法比 BUC 算法获得了 4.1 倍加速比.但算法没有考虑现代处理器的多核特征.

为提高方体计算效率,并行方体计算获得了广泛的关注,文献[6]提出了一系列基于 PC Cluster 的方体算法,希望实现实时计算.在 Cluster 上算法的核心问题是负载均衡,保证每个节点同步工作.文献[7]提出了适合并行机的 Pnp 算法,算法利用自顶向下的流水线技术和自底向上的剪枝技术提高算法效率.无论是 PC Cluster 或者并行机,体系结构和 GPU 都有很大的不同.GPU 整体上是共享内存结构,并且每个处理器组(Multiprocessor,MP)具有更快的共享内存(Shared Memory,SM),处理器之间通信不是通过消息传递机制而是通过不同层次的共享存储器.

2.2 图形处理器 GPU

近年来,GPU 已进化为一种通用图形处理器(General Processing GPU,GPGPU),并在各个领域获得了广泛应用.GPU 由多个 SIMD(Single Instruction Multiple Data)的 MP 组成.同一时刻,

MP 中的任一处理器执行相同的指令，但操作不同的数据. GPU 拥有大容量的显存(Global Memory，GM)，其特点是高带宽高访问延迟. 另外，每个处理器组有一个芯片内 SM，被所有 MP 内处理器共享，特点是存储容量小但访问速度快. 简化后的 GPU 体系结构如图 1 所示.

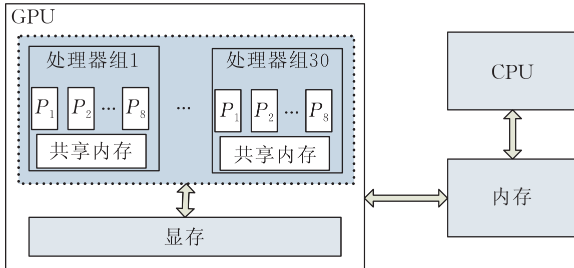


图 1 GPU 体系结构

2006 年，NVIDIA 公司推出了 Compute Unified Device Architecture (CUDA)编程模型，研究人员使用 C 语言就可以完成 GPU 程序的开发. CUDA 线程与 CPU 线程相比，由于有特殊硬件的支持，线程创建和切换代价很低. 另外，与 CPU 通过多级 Cache 缓存数据减少访存代价不同，GPU 可以通过大量多线程并发隐藏访存延迟. CUDA 线程划分为多个线程块(Thread Block，TB)，每个 TB 由一个 MP 执行. CUDA 程序分为在 CPU 上执行的 host 和在 GPU 上执行的 kernel 两部分.

GPU 算法由于大规模多线程并发特点，算法优化对性能影响很大，常用的算法优化技术主要包括以下 3 点：联合访问(coalesced access)、使用共享内存减少访存代价、避免 Bank 冲突.

近年来，通过 GPU 加速数据处理获得了广泛关注. Join 是数据库的核心操作之一，一些学者提出了基于 GPU 的 Join 算法，获得了 2～7 倍的加速比[9]. 在此基础上开发了基于 GPU 的数据库管理系统 GDB，并讨论了 GDB 的代价模型[10]. 数据挖掘算法需要对大量的数据进行复杂计算，通过 GPU 加速数据挖掘算法获得了较高的加速比. DBSCAN 作为一种传统的聚类算法，文献[11]提出了基于 GPU 的 CUDA-DClust 算法，算法通过多个密度链并行执行，链内多线程并行计算点与点之间的距离，获得了很高的加速比；同时传统的频繁相集挖掘算法 Apriori 和 FP-growth 也获得了较高的加速比[12].

3 GPU-Cubing 算法

3.1 相关定义

为了方便描述算法，与文献[9-10]相似，我们首先定义一系列并行操作，这些操作充分利用 GPU 特征，实现了多线程并行执行. 主要操作定义如下.

定义 1. 映射(map). 映射是一个转换过程，使用转换函数处理输入元素，并写入到输出数组中. 形式化描述如下：

$$O[i] = \mathrm{fun}(I[i]) \tag{1}$$

其中：$I[i]$和 $O[i]$分别是输入和输出数组，fun 是映射函数.

GPU 中每个 TB 处理输入数组的一段，每个线程负责段中的一个元素的转换.

定义 2. 扫描(scan). 扫描是一种在并行算法设计中常用的操作，扫描的形式化定义如下.

输入 I：$[a_0, a_1, a_2, \cdots]$，

输出 O：$[a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, \cdots]$.

其中$\oplus$为任意二元操作符，比如＋等. 在 GPU-Cubing 中，使用了前缀和扫描. 我们采用 CUDPP[13] 中的实现.

前缀和扫描分为两个阶段：第一个阶段有 $\log_2|I|$个步骤，$|I|$ 表示输入数组大小，在第 i($0 \le i \le \log_2|I|$)步，线程 j 计算 $I[j \times 2^i]$和 $I[(j+1) \times 2^i]$的部分和，做 $n-1$ 次加法；第 2 个阶段也是 $\log_2|I|$个步骤，使用由第 1 个阶段产生的结果为输入，需要 $n-1$ 次加法和 $n-1$ 次交换. 每个阶段都并行完成.

定义 3. 发散和汇集(scatter and gather)分别对应索引写和索引读操作. 形式化定义如下.

$$O[P[i]] = I[i] \tag{2}$$

将输入数组 I 的第 i 个元素 $I[i]$，写入到输出数组 O 的 $P[i]$位置.

$$O[i] = I[P[i]] \tag{3}$$

将输入数组的 $P[i]$位置的元素读入到输出数组的第 i 个位置.

将输入数组分成多个段，每个 TB 负责一个段，每个线程负责一个元素的发散或汇集.

定义 4. 约简(reduce). 约简是从一个输入数组中计算一组输出值，类似于 Group by. 描述如下.

$$\{O[k], k \in [1,m]\} = \{\odot_{i=P[k]}^{P[k+1]} I[i] \mid k \in [1,m], \quad P[1]=0, P[m+1]=n-1\} \tag{4}$$

其中$\odot$表示约简函数，比如 sum、max 等. $P[k]$表示分位点，将输入数组分成 m 段，然后每段计算一个结果，并写入到输出数组的 k 位置. 算法描述如下(以求和为例)：

Kernel. *Reduce* (I,P)

Inputs：

```
    I: input array
    P[0,…,m]: quantile array
  Globals:
    bid: index of TB(0,…,m-1)
    tid: index of thread in TB(0,…,|TB|-1)
    |TB|: thread number in TB
  Output:
    O[0,…,m-1]: output array
  begin
   1. int offset=1, mask=1; int_shared_ S[|TB|];
   2. S[tid]=0;
   3. for i=P[bid]; i <P[bid+1];i+=|TB| do
   4.    S[tid]+=I[i];
   5. end for
   6. _syncthreads(); //同步语句
   7. while (offset<|TB|)
   8. if ((tid & mask)==0)
   9.    S[tid]+=S[tid+offset];
  10. end if
  11. offset+=offset;
  12. mask=offset+mask;
  13. _syncthreads();
  14. end while
  15. O[bid]=S[0];
  End
```

1～5 行每个 TB 将自己负责的段累加到 SM 中，利用 SM，提高了 Cache 的本地性. 第 6 行是同步语句，保证每个 TB 中的线程都将自己负责的数据累加到了 SM 中. 7～14 行完成树状求和，每个 TB 负责 SM 中的 $|SM|$ 个元素，需要执行 $\log_2|SM|$ 步骤，在第 i $(0\le i\le \log_2|SM|)$ 步，线程 j 计算部分约简结果 $S[j\times 2i]$ 和 $S[(j+1)\times 2i]$. 每个 TB 计算输出结果中的一个元素. 15 行获取输出结果.

定义 5. 排序(Sort). 排序是将一组无序的数按升序或降序排成一组有序的数. 对任意输入数组 I，输出数组 O 满足：

$$O[i]\le O[j],\quad i\le j \tag{5}$$

在输出数组中，i 和 j 位置的元素值，如果 $i\le j$，则 $O[i]\le O[j]$. GPU-Cubing 主要用到如下排序算法.

基数排序(radix sort). 基数排序是非比较排序算法，算法的时间复杂度是 $O(n)$. 算法源于 CUDPP[14].

将输入数组划分为多个段，每个 TB 在 SM 中对某段进行 b 次 1 位分裂；每个 TB 将 2^b 个计数器和排序后的段写入 GM；对 $p\times 2^b$ 计数器前缀和扫描，获取全局偏移量；最后每个 TB 负责将本段数据复制到正确的输出位置. 上述过程需要执行 d/b 次. 其中 d 表示每个元素占用的位数，b 表示每次处理的位数，p 表示 TB 个数.

双调排序(bitonic sort). 双调排序是一种与数据无关的排序方法，时间复杂度为 $O(n\log^2 n)$，当元素个数较少时，效率非常高.

双调排序有 $\log_2|I|$ 个阶段，阶段 j 有 j 个步骤，在步骤 i，构建一个大小为 2^i 的双调序列. 当元素个数较少时，双调排序在 SM 中完成，从而有较少访存延迟.

定义 6. 过滤(filter). 过滤是从一组元素中选取某些满足一定条件的元素. 描述如下

$$\{O[i],i\in[1,m]\}=\{I[i]\mid \mathrm{fun}(I[i])=1,\\ i\in[1,n]\},m\le n \tag{6}$$

将输入数组 I 中满足条件的元素存储到输出数组 O 中.

对输入数组中每个元素应用过滤条件，生成位向量，满足过滤条件，对应位置 1，否则置 0(映射操作)；对位相量前缀和扫描；发散操作，将输入数组对应位相量为 1 位置的值写入到输出数组的前缀和扫描位置. 在 GPU-Cubing 算法中，通过过滤操作获取满足最小支持度的分区.

3.2　算法实现

GPU-Cubing 采用自底向上、广度优先的方式完成方体计算. 首先对 A 属性进行划分，并将结果 *cuboid* A 输出；接着基于满足最小支持度的 A 分区，对 B 属性进行划分，得到 *cuboid* AB；依次类推，直到所有的 *cuboid* 处理完成. 算法在 GPU 上完成主要分为 3 个阶段：

(1) 当需要处理的分区足够大时，整个 GPU 完成对分区的划分，并计算当前的 *cuboid*；

(2) 当分区足够多时，每个 TB 处理一个分区；

(3) 当分区小并且多时，每个线程处理一个分区.

算法生成树如图 2 所示.

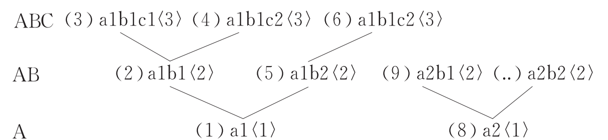


图 2　GPU-Cubing 算法的生成树

圆括号中的数字表示 BUC 算法的处理顺序，而尖括号中的数字表示 GPU-Cubing 处理的顺序. BUC 算法结果的写入方式是深度优先，生成的结果属于不同的 *cuboid*，需要较大的内存空间保存所有

的 *cuboid*，同时 cache 的空间本地性(spatial locality)差. 而 GPU-Cubing 以宽度优先的方式输出结果，一次处理一个 *cuboid*，结果顺序输出. GPU-Cubing 算法详细描述如下.

```
GPU-Cubing (dim, data)
Inputs:
  dim: beginning dimension
  data: input data
Globals:
  D: number of dimensions 0…D−1
  minsup: minimum support
  counter⟨pos,count⟩[D][]: partition information
  counter[0][0].pos=0; counter[0][0].count=T
  counter[0].size()=1(T: number of tuples)
  |TB|: thread number in TB
  bid: index of TB
  tid: index of thread in TB(0,…, |TB|−1)
  bit[T], prefix[T]
Output:
  Cube or Iceberg Cube
Begin
 1. for d=dim; d<D; d++ do
 2.    |TB|=counter[d].count();
 3.    for each TB parallel do
 4.       int pos=counter[d][bid|tid].pos;
 5.       int count=counter[d][bid|tid].count;
 6.       partitions=partition(d,data[pos,pos+count]);
 7.    end for
 8.    counter[d+1]=filter(partitions,minsup); //filter
 9.    sort (counter[d+1]);
10.    cuboid=getcuboid (counter [d+1]); //gather
11.    copyCuboidToCpu(cuboid);
12.    GPU-Cubing(dim+1,data);
13. end for
End
kernel partition(dim,data)
Begin
14. sort (data);
15. bit[0]=1;
16. if (data[dim][tid]==data[dim][tid+1])
17.    bit[tid+1]=0;
18. else
19.    bit[tid+1]=1;
20. end if
21. prefix=prefixsum(bit);
22. position[0]=0;
23. if (prefix[tid+1]<prefix[tid])
24   counter[position[prefix[tid]]].pos=tid+1;
25. end if
26. counter[tid].count=position[tid+1]−position[tid];
End
```

基础数据每列单独存储在 GM. 依次处理每个维开头的 *cuboid*. 第 2 行根据待处理分区的个数确定 TB，3～7 行每个 TB 处理一个分区，第 8 行使用过滤操作得到满足 minsup 的分区，第 9 行使用排序操作对过滤后的计数器排序，第 10 和 11 行根据计数器获取最终结果并传回 CPU. 第 12 行递归处理以当前 *cuboid* 开头的下一个 *cuboid*.

算法核心是分区操作，根据待处理分区的大小，分区操作可以由整个 GPU、TB 或一个线程完成. 一个 TB 处理一个分区的实现过程如图 3 所示.

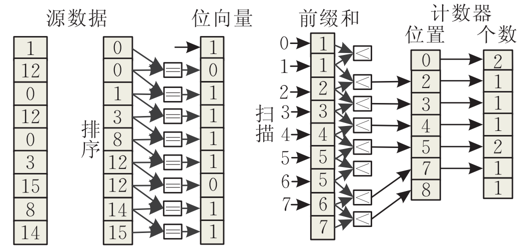


图 3 分区执行过程

首先对原始数据排序(14 行)；然后对排序后数据应用映射操作，得到一个位相量(15～20 行)；通过位相量作前缀和扫描，从而确定分区中不同值的个数和位置(21 行)，然后应用发散和汇集操作，得到不同值的开始位置(22～25 行)，如果是求 *count*，则直接获取个数(26 行)，如果是其它函数，比如 sum，则应用约简操作. 所有操作均并行完成，实现了 TB 内并行；同时，多个分区并行执行，实现了 TB 间并行. GPU-Cubing 算法适应了 GPU 并行模型，保证了算法的高性能.

下面我们通过一个例子来说明 GPU-Cubing 算法执行过程. 比如有两个 TB(T_0，T_1)，每个 TB 有 2 个线程(T_{00}，T_{01} 和 T_{10}，T_{11}). 计算最小支持度为 2 的三维冰山方体的过程如图 4 所示. 在步骤(0)所有线程完成对 A 属性的划分，得到 *cuboid* A；步骤(1)在 A 属性划分的基础上，对 B 属性进行划分，得到 *cuboid* AB，这时每个 TB 处理一个分区；在步骤(2)中对 C 属性进行划分，由于这时分区很多，且每个分区很小，这时每个线程负责一个分区的划分，得到 *cuboid* ABC. 然后依次处理 B、BC 和 C. 从而完成整个方体的计算.

GPU-Cubing 中采用多属性排序算法：算法通过一个位置数组记录元素排序后的位置，然后其他维的元素根据位置数组移动，这里需要 Ping-pong 操作，所以需要两倍的列存储空间. 通过多属性排序保证了处理下一个属性时，多线程的联合访问. 同

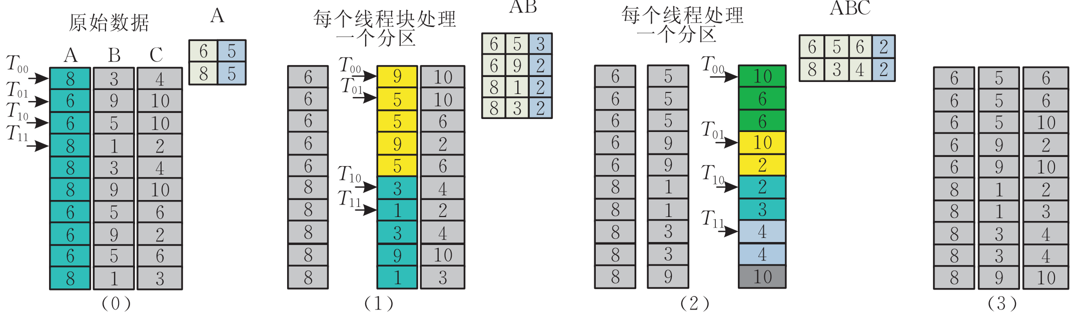


图 4　GPU-Cubing 执行过程

时，通过多属性排序算法可以方便计算任一 *cuboid*.

3.3　内存需求

假设关系中有 N 个元组，每个元组包含 D 个维，每个维的势为 $C_i(0\le i\le D-1)$. 则算法中的中间数据结构 *counter*[i]满足如下要求.

定理 1. GPU-Cubing 算法中计数器 *counter*[i]满足条件：$|counter[i]|\le\min\left(\prod_{i=0}^{D-1}C_i, N\right)$，计数器中最大元素个数为 N.

证明. 开始时只有一个分区，$|counter[0]|=1$. 对属性 A 划分后，计数器大小满足 $|counter[1]|\le C_0$，$|counter[1]|\le N$，所以有 $|counter[1]|\le\min(C_0, N)$. 假设生成 $|counter[1]|=n$ 个分区，每个分区的元素个数为 $|P_i|$，则满足条件 $\sum_{i=1}^{n}|P|_i\le N, n\le C_0$；对属性 B 划分后，对每个分区 P_i 生成的计数器满足 $|counter[2]_i|\le\min(C_1, |P|_i)$，则 $|counter[2]|=\sum_{i=1}^{n}|counter[2]_i|\le\sum_{i=1}^{n}|\min(C_1, |P|_i)|\le\min(C_1\times C_0, N)$，依次类推，则计数器大小满足 $|counter[i]|\le\min\left(\prod_{i=0}^{D-1}C_i, N\right)$，最大包含元素个数为 N.

计数器中每个元素需要 8 个字节，算法需要 D 个计数器. 位向量每个元素占用 1 位，在排序中需要使用位置数组记录排序后其它维元素移动的位置，位置数组和前缀和数组及原始数据中每个元素占用 4 个字节. 多属性排序，Ping-pong 操作需要一列作为辅助空间，GPU-Cubing 算法总的最大内存需求为

$$N(4\times D+8+1/8)+\sum_{j=1}^{D-1}\min\left(\prod_{i=0}^{j-1}C_i, N\right)\times(16+4+1/8)+N\times 4 \tag{7}$$

另外，只要 GM 可以存储一列，算法就可以正常运行，当一列也不能存储时，可以采用文献[15]的基于硬盘、内存、GPU 三级流水排序算法完成数据的划分.

3.4　维序与维的个数

维序和维的个数对 CPU 算法的性能影响较大，把区分度高的维放在前面，可以实现更快的剪枝，提高算法效率；随着维个数增长，算法效率急剧下降. 但 GPU-Cubing 算法对维序和个数并不敏感. 剪枝操作对性能影响不大，因为 GPU 可以同时处理大量的分区，而且分区越多并行度越高，可以更有效的隐藏内存延迟，提高效率. 相反，通过剪枝虽然可以减少一些运算，但对算法的整体运行效率影响不大. 另外 GPU-Cubing 算法采用一种层层递进的方式依次处理每个 *cuboid*，新加入的维可以利用前面的分区结果. 实验结果也验证了上述论述.

定理 2. 随着维个数增长，GPU-Cubing 所需内存线性增长.

证明. GPU-Cubing 算法需要固定上限的中间数据结构，每次完成一个 *cuboid* 计算并输出. 所以算法所需内存主要由基础数据决定. 而基础数据随维个数增长线性增长.

4　算法优化技术及扩展

4.1　动态调整排序算法

在 GPU-Cubing 算法中，数据划分通过排序算法完成，这里主要用到基数排序和双调排序. 当分区中包含足够多的元素时，采用基数排序；当算法进入到每个 TB 处理一个分区时，由于 TB 内线程可以同步，算法采用双调排序.

算法没有使用计数排序(counting sort)，因为计数排序中某些操作需要原子操作，会影响性能；另外计数排序会使算法中 *counter* 数据结构过大.

4.2　TB 内线程数动态调整

当算法进入到每个 TB 处理一个分区时，为了提高排序和前缀和效率. 首先将数据从 GM 取到 SM，然后在 SM 中完成排序和前缀和处理. 排序和

前缀和操作与 TB 内线程个数相关. CUDA 要求所有 TB 内线程数相同,并且每个 TB 可使用的 SM 大小相等,我们通过在 SM 中补 0 来使每个分区大小相同. 但实际上每个分区内元素个数不等,如果设置线程数为每个 TB 可支持的最大值,会造成大量不必要的运算. 于是根据每次 kernel 启动时分区中最大分区元素个数动态确定 TB 内线程数,考虑双调排序要求待排序个数为 2 整数次幂,所以线程数为

$$Threads=2^k\geq\max(|P_1|,|P_2|,\cdots,|P_n|)\,|\,2^{k-1}<\max(|P_1|,|P_2|,\cdots,|P_n|) \quad (8)$$

其中 $|P_i|$ 表示分区中元素个数,k 为正整数.

4.3 负载平衡

通常情况下,分区大小不等,而且在数据集倾斜分布时,问题更加严重. 当多个 TB 或线程并行处理分区的时候,会出现负载不均问题. 在这种情况下,我们将分区划分为 3 组:第一组是较大的分区,每个分区由整个 GPU 处理;第二组是较小的分区,每个分区由一个 TB 处理;第三组是非常小的分区,一个线程处理一个分区. 分组考虑硬件特征并保证并行度.

由于目前 GPU 硬件限制,上述分组在某些情况下并不能充分发挥 GPU 的能力. 比如当分区大于 TB 可以处理而小于整个 GPU 处理能力时,目前只能使用整个 GPU 来运行,因为 GPU 不支持 kernel 级的并行. 但在 NVIDIA 的 Fermi 架构显卡中,支持此项功能,相信在新硬件上算法效率会进一步提升. Kernel 级并行和串行运行比较如图 5 所示.

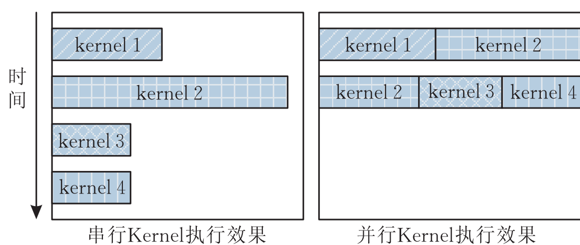


图 5 并行 Kernel 执行效果

4.4 Lock-free 结果输出

目前,在 kernel 程序执行过程中,GPU 不支持增量 GM 内存分配. 所以 GPU-Cubing 算法的结果输出需要解决两个问题:确定结果 *cuboid* 的大小;避免写冲突. 我们把结果输出分为 3 个阶段:

首先,每个 TB 计算所处理分区的结果大小,并保存到数组 $O[i]\,|\,0\leq i<|TB|$,作为临时计数器;

然后,计算临时计数器 O 的前缀和 P,从而确定每个 TB 产生的结果写出的位置和输出结果 *cuboid* 大小;

最后,通过 host 程序分配 GM 空间,第 i 个 TB 将结果写到对应 $P[i]$ 位置,从而避免了写冲突. 同时也避免了原子操作,提高了效率.

4.5 扩展的 Multi-GPU 和 Multi-core CPU

算法可以扩展到多 GPU. 算法中的关键部分排序和前缀和扫描都可以扩展到多 GPU,首先将数据分段,每段在 GPU 上单独处理,然后在 CPU 端合并. 当算法进入第 2 和第 3 阶段时,产生了足够多的分区,这时将分区均匀的分布的不同的 GPU,多个 GPU 并行工作,CPU 负责合并结果.

通过每个核心计算一个分区,算法也可以运行在多核 CPU 上.

5 实验分析

为了验证 GPU-Cubing 算法的有效性,我们进行了详细的实验验证. 实验平台采用 Windows XP 操作系统,开发工具为 Visual Studio 2005,语言为 C++. GPU 型号为 GTX275,CPU 为 Core2 Quad CPU 2.66GHz. GPU 通过 PCI-E 总线与主存交换数据. 实验硬件配置如表 1 所示.

表 1 硬件配置

	处理器频率	缓存	内存	带宽
GPU	1440MHz×30	16KB×30	1.8GB	127GB/s
CPU	2.66GHz	L2:3MB×2, L1:32KB×4×2	4GB	10.67GB/s

实验数据包括合成和真实数据集,合成数据集 zipf 分布,随着倾斜度 S 增大数据集越倾斜. 用 D 表示维的个数,用 C 表示维的势,用 T 表示元组个数,用 M 表示最小支持度.

真实数据是天气数据集 May95L.DAT[16],包含 28 个维,共有 1195149 个元组. 我们使用了 9 个维,维的名称和势如下: sky brightness indicator(2), longitude×100(36000), total cloud cover(9), middle cloud type(12), high cloud type(12), middle cloud amount×100(800), high cloud amount×100 (800), wind speed(999) and dew point depression (C×10)(700).

实验中用到的 BUC 算法来源于网络并根据需要对算法进行了一些修改.

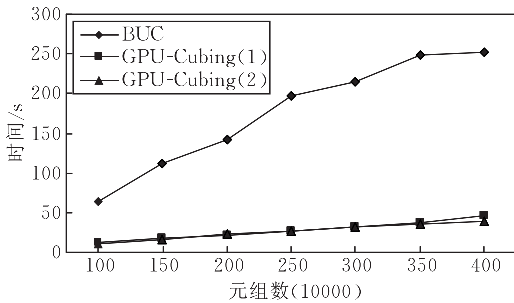
GPU-Cubing 中原始数据存储在 GM 中,算法计时包含 *cuboid* 结果从 GPU 传输到 CPU 的时间. BUC 算法完全使用 CPU 端运行.

在目前硬件条件下,GPU 设备的频繁启动会耗

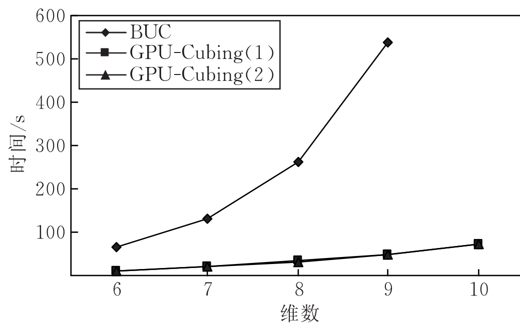
费很大的资源，所以在记录 GPU-Cubing 算法时间时，我们记录了两个时间，一个是包含启动 GPU 的时间(1)，一个是不包括启动 GPU 的时间，只包括计算和数据传输时间(2).

5.1　合成数据集上完全方体计算

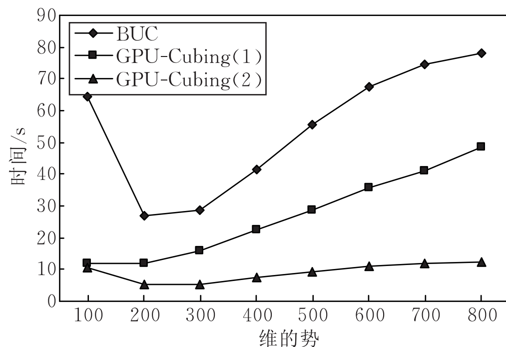
在这一节中，我们比较 GPU-Cubing 与 BUC 算法在完全方体计算上的性能，我们分别改变元组数、维的个数和维的势来验证算法的效率. 所有维的势相同，数据均匀分布($S=0$). 实验结果如图 6 所示.



(a) $C=100$, $D=6$



(b) $T=1\text{M}$, $C=100$



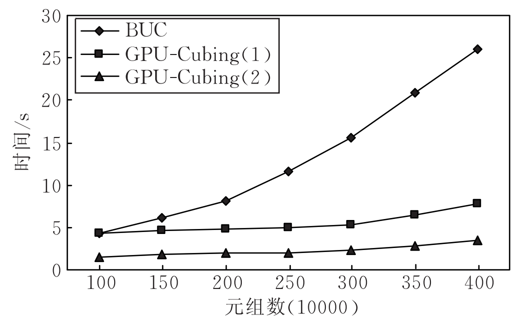
(c) $T=1\text{M}$, $D=6$

图 6　方体计算效率对比

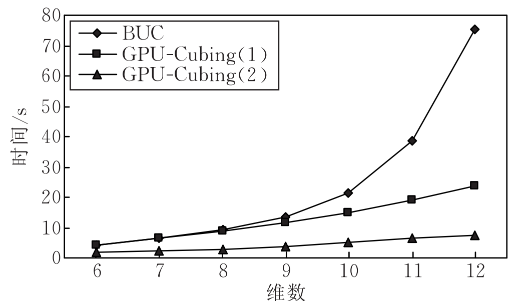
随着数据量图 6(a)和维个数图 6(b)增长，GPU-Cubing 算法线性增长，与 BUC 算法加速比逐渐提高，算法性能提高一个数量级以上，在图 6(b)中，当维的个数增长到 10 时，BUC 算法内存溢出，而 GPU-Cubing 可以在短时间内完成. 在图 6(c)中 GPU-Cubing 算法随维势的增长而线性增长，与 BUC 算法趋势相似，但加速比逐渐下降，最后趋于一个较稳定值. 主要原因是随着维势增长，算法推迟进入第 2 个阶段，不能充分发挥 GPU 的计算能力；同时，随着维势增长，kernel 程序启动的次数明显增加，GPU-Cubing(1)和 GPU-Cubing(2)差距逐渐增大.

5.2　合成数据集上冰山方体计算

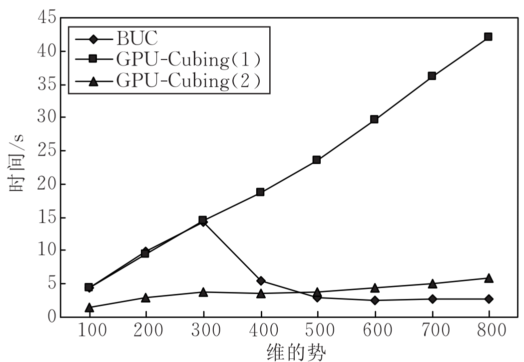
在本节我们测试 GPU-Cubing 与 BUC 算法计算冰山方体的效率，所有实验选择 $M=10$，分别测试了随着数据量增长、维个数增长及维的势增长的算法效率对比. 实验结果如图 7 所示.



(a) $C=100$, $D=6$



(b) $T=1\text{M}$, $C=100$



(c) $T=4\text{M}$, $D=6$

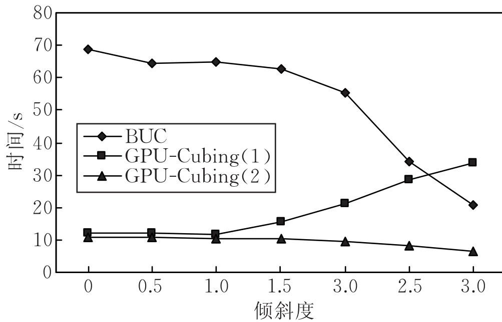
图 7　冰山方体计算效率对比

实验数据表明，冰山方体计算的加速比要小于完全方体计算. 主要原因是剪枝操作对 GPU-Cubing 算法影响较小，但对 CPU 算法影响较大. 而 GPU-Cubing 获得高性能的关键是：在处理某一个 *cuboid* 时，有足够多的分区，保证 GPU 能够最大限度的并行，而这一点正适合完全方体计算. 在图 7

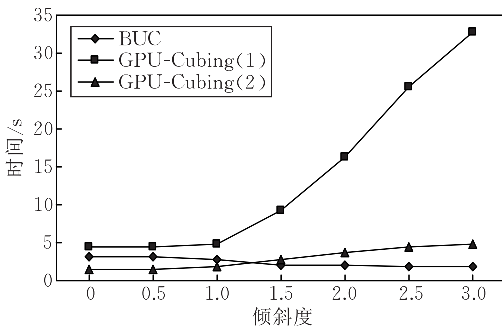
(c)中随着维势的增长，BUC 算法逐渐优于 GPU-Cubing，因为随着维势增长，数据变得更加稀疏，BUC 的剪枝策略可以发挥更大的作用. 如果在处理某些维时只有有限个数的分区，GPU-Cubing 算法并行度低，很难获得高性能.

5.3 倾斜数据集上方体计算

本节我们测试了 GPU-Cubing 算法在倾斜数据集上与 BUC 算法的效率对比，实验结果如图 8 所示.



(a) $T=1\mathrm{M}, D=6, M=1$



(b) $T=1\mathrm{M}, C=100, M=10$

图 8 倾斜数据集上方体计算效率对比

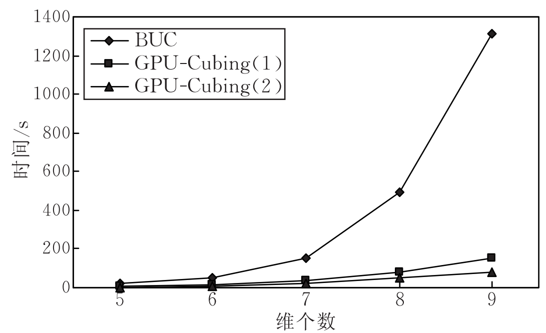
图 8(a)测试了随着数据倾斜度 $S(0\text{-}3)$的变化，完全方体计算的效率对比，GPU-Cubing 虽然优于 BUC，但加速比逐渐下降. 主要原因是随着倾斜度增大，GPU 中出现负载不均，效率降低. 图 8(b)测试了 $M=10$ 时在倾斜数据集上冰山方体计算的效率. 在数据分布倾斜度很大时，BUC 算法优于 GPU-Cubing 算法，效果与图 7(c)类似.

我们也测试了在 $T=4\mathrm{M}$ 时在不同倾斜度上算法的效率，GPU-Cubing 获得了更高的加速比.

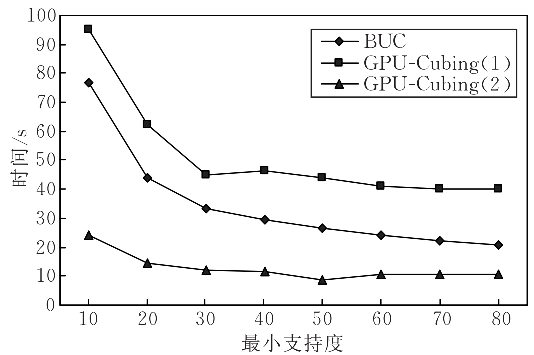
5.4 真实数据集上方体计算

我们测试了在真实数据集上 GPU-Cubing 算法的效率，测试结果如图 9 所示.

在图 9(a)中比较了随着维个数的增长，GPU-Cubing 和 BUC 算法的效率及加速比. 随着维数增长，BUC 效率急剧下降，而 GPU-Cubing 近似线性增长，在维数为 9 时，GPU-Cubing 算法比 BUC 快一个数量级以上. 图 9(b)测试了随着 M 的增长，两个算法的效率，加速比逐渐下降，但最终趋于一个相对稳定范围. GPU-Cubing 算法虽然对剪枝效果不明显，但当剪枝达到一定阈值时，如果可以有效地减少计算量，剪枝策略也可以提高效率. GPU-Cubing 算法表现出随着 M 的增长分段递减的趋势，当 $M=10^3$时，加速比大约为 1 倍多，加速比达到最小值，但当 M 继续增长为 10^4 和 10^5时，加速比为 6～7 倍.



(a) $M=1$



(b) $D=9$

图 9 真实数据集上方体计算效率对比

5.5 聚集计算的性能

在本节我们给出聚集(group by)计算的性能测试，聚集计算通过 filter＋partition＋reduce 完成. 首先根据 where 条件对原始数据过滤；然后对过滤后的数据进行划分；最后对每个分区进行约减. 数据集采用基于 OLAP 的星型模式数据集 SSB[17]，事实表包含 600 万条记录，测试了 SSB 包含的 13 个查询. SSB 的这 13 个查询分为 4 组(Q1，Q2，Q3 和 Q4)，分别用 Q11，Q12，Q21 等表示每个查询. 测试结果和加速比如图 10 所示.

GPU 算法明显优于 CPU 算法，而且 GPU 算法的加速比与数据量成正比，当数据量较大时，更能发挥 GPU 高带宽和大规模并行的优势，从而获得高加速比.

6 结 论

随着 GPU 硬件和可编程性的不断发展，基于

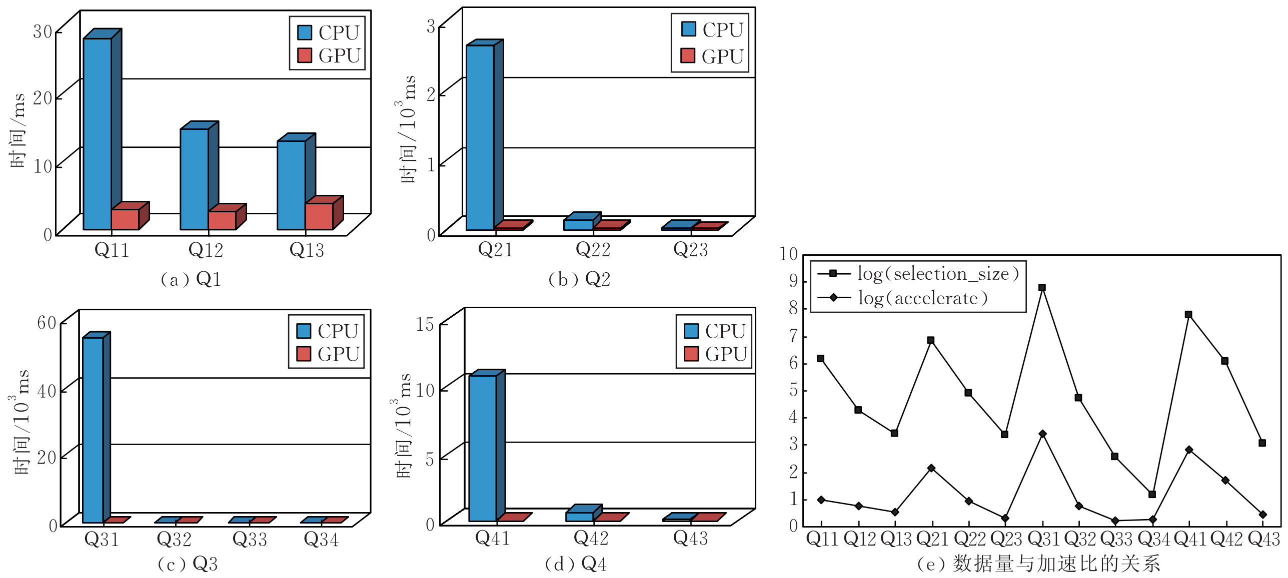


图 10　基于 GPU 的聚集操作

GPU 的应用会越来越广泛. 本文探讨了通过 GPU 加速数据仓库和 OLAP 领域的方体计算问题，提出了 GPU-Cubing 算法，算法同时对多个分区进行划分，分区内多线程并行，适合 GPU 的体系结构，获得了较高的加速比.

随着人们对数据仓库实时性要求越来越高，本文试图通过新一代硬件提高数据仓库的性能，为实现实时数据仓库提供有益的尝试. 我们下一步的工作首先完成基于多 GPU 的算法扩展，从而适应海量数据，然后探索开发基于新一代硬件的数据仓库和 OLAP 原型系统.

参　考　文　献

[1] Zhao Y, Deshpande P M, Naughton J F. An array-based algorithm for simultaneous multidimensional aggregates//Proceedings of the ACM SIGMOD International Conference on Management of Data. New York, 1997: 159-170

[2] Beyer K, Ramakrishnan R. Bottom-up computation of sparse and iceberg CUBEs//Proceedings of the ACM SIGMOD International Conference on Management of Data. New York, 1999: 359-370

[3] Xin D, Han J W, Li X L, Wah B W. Star-Cubing: Computing iceberg cubes by top-down and bottom-up integration//Proceedings of the 29th International Conference on Very Large Data Bases (VLDB). San Francisco: Morgan Kaufmann Publishers, 2003: 476-487

[4] Shao Z, Han J W, Xin D. MM-Cubing: Computing iceberg cubes by factorizing the lattice space//Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM). Washington: IEEE Computer Society, 2004: 213-222

[5] Hellerstein J, Haas J, Wang H. Online aggregation//Proceedings of the ACM SIGMOD International Conference on Management of Data. New York, 1997: 171-182

[6] Ng R, Wagner A, Yin Y. Iceberg-cube computation with PC clusters//Proceedings of the ACM SIGMOD Conference on Management of Data. Santa Barbara, California, 2001: 25-36

[7] Chen Y, Dehne F, Eavis T, Rau-Chaplin A. PnP: Sequential, external memory and parallel iceberg cube computation. Distributed and Parallel Databases, 2008, 23(2): 99-126

[8] Luan Hua, Du Xiao-Yong, Wang Shan. Cache-Conscious data cube computation on a modern processor. Journal of Computer Science and Technology (JCST), 2009, 24(4): 708-722

[9] He Bing-Sheng, Yang Ke, Fang Rui et al. Relational joins on graphics processors//Proceedings of the ACM SIGMOD International Conference on Management of Data. Vancouver, 2008: 511-524

[10] He B, Lu M, Yang K et al. Relational query co-processing on graphics processors. ACM Transaction on Database System, 2009, 34(4): 1-39

[11] Böhm C, Noll R, Plant C, Wackersreuther B. Density-based clustering using graphics processors//Proceeding of the 18th ACM Conference on Information and Knowledge Management (CIKM). Hong Kong, 2009: 661-670

[12] Fang W, Lu M, Xiao X et al. Frequent itemset mining on graphics processors//Proceedings of the 5th International Workshop on Data Management on New Hardware (DaMoN). New York, 2009: 34-42

[13] Shubhabrata Sengupta, Mark Harris, Michael Garland. Efficient parallel scan algorithms for GPUs. Santa Clara: NVIDIA, Technical Report: NVR-2008-003, 2008

[14] Nadathur Satish, Mark Harris, Michael Garland. Designing efficient sorting algorithms for many-core GPUs//Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium. Aurelio, 2009: 1-10

[15] Govindaraju N, Gray J, Kumar R et al. GPUTeraSort: High performance graphics coprocessor sorting for large database management//Proceedings of the ACM SIGMOD International Conference on Management of Data. Chicago, 2006: 325-336

[16] Hahn C, Warren S. Extended edited synoptic cloud reports from ships and land stations over the globe. 1952-1996. http://cdiac.esd.ornl.gov/ftp/ndp026c/ndp026c.txt

[17] Patrick O'Neil, Elizabeth (Betty) O'Neil, Chen Xue-Dong. The star schema benchmark (SSB). Boston: UMB, 2009. http://www.cs.umb.edu/～poneil/StarSchemaB.PDF

ZHOU Guo-Liang, born in 1978, Ph. D. candidate. His current research interests include new hardware based data warehouse and OLAP technology.

CHEN Hong, born in 1965, professor, Ph. D. supervisor. Her current research interests include high performance computing and sensor networks.

LI Cui-Ping, born in 1971, Ph. D., Ph. D. supervisor. Her current research interests include social network and graph data mining.

WANG Shan, born in 1944, professor, Ph. D. supervisor. Her research interests include high performance scalable database, data warehouse, database information retrieval and video database.

ZHENG Tao, born in 1989, master candidate. His current research interests include new hardware based high performance data mining and OLAP system.

Background

Cube computation is a core and time consuming problem in OLAP fields and significantly affects the performance of OLAP. Many algorithms and technologies are proposed for cube computation, but there is still a large gap between user requirements based on BI Survey. In recent years, computer architecture has evolved with large memory and multi-core or many-core processors. Using the features of new generation hardware to improve the performance of algorithms and design new algorithms are the focus of research. GPU-based algorithms are a hot spot. Modern GPU have more computing power and higher memory bandwidth than CPU. The latest GPU like NVIDIA GTX275 contains 240 processing cores and theoretical performance beyond 1 TFLOP. GPU based joins acquire about 2～7 times speedup. GPU based sort and data mining also are proposed and high performance. But there are few works considering cube computation utilizing GPU.

In this paper a GPU based parallel cube computation (GPU-Cubing) is proposed. GPU-Cubing achieves a speedup over BUC up to an order of magnitude in full cube computation and at least 2 folds in iceberg cube computation in real dataset. To the best of our knowledge, this technology is given firstly. The work is supported by the National High-Tech Research and Development Plan of China under Grant No. 2008AA01Z120 and Ph. D. Programs Foundation of Ministry of Education of China under grant No. 20090004110002. The target of research group is to establish a GPU based OLAP system which significantly improve performance of OLAP system and support complex applications such as what-if analysis function.