异构平台上多维线性哈希的研究

刘 勇[1,3] 赵秦德[2] 赖正文[3] 黄东平[3] 王璟星[3]
(广西民族大学信息科学与工程学院 南宁 530006)[1] (广西交通科学研究院 南宁 530002)[2]
(华南理工大学软件学院 广州 510006)[3]

摘 要 目前多维数据广泛应用于多个领域,但其复杂性影响了多维数据的操作效率。为提高对多维数据的处理能力,提出一种在 CPU/GPU 异构平台上的多维线性哈希并行计算方案。该方案通过对传统线性哈希表数据结构的扩展,可实现对哈希表的快速创建和查询。同时,在多个处理器平台上进行的实验对提出的方案的有效性进行了验证。实验结果表明,当处理的数据规模较大时,提出的方案由于充分利用了 GPU 强大的并行处理能力,在创建哈希表和查询数据上,比传统的 CPU 方案性能分别提高了约 25 倍和 38 倍,充分显示出提出的方案在处理多维数据时的优势。

关键词 异构平台,图形处理器,多维线性哈希,计算统一设备架构
中图法分类号 TP393 **文献标识码** A

Research for Multidimensional Linear Hashing on Heterogeneous Platforms

LIU Yong[1,3] ZHAO Qin-de[2] LAI Zheng-wen[3] HUANG Dong-ping[3] WANG Jing-xing[3]
(College of Information Science and Engineering, Guangxi University for Nationalities, Nanning 530006, China)[1]
(Guangxi Transportation Research Institute, Nanning 530002, China)[2]
(School of Software Engineering, South China University of Technology, Guangzhou 510006, China)[3]

Abstract Nowadays, multidimensional data is used in plenty of areas, but for the complex of multidimensional data, the efficiency of computing is not perfect. In order to speed up the query and manipulation of multidimensional data, a multidimensional linear hashing parallel computing solution on CPU/GPU heterogeneous platforms was proposed. Based on the extension of traditional hashing table data structure, it is able to achieve quick creating and query. Experiments on different platforms were done. The experimental results show that the proposed solution is about 25 times and 38 times faster than traditional solution on hashing table creating and data query, and it reflects the advantage of the proposed solution.

Keywords Heterogeneous platforms, Graphical processing unit, Multidimensional linear hashing, Compute unified device architecture

1 引言

随着现代计算机技术的发展,多维数据已广泛应用到各种商业领域中,从最初的图像处理、计算机辅助设计制造(CAD/CAM)和地理科学领域扩展到机器人、视知觉、自动导航和医学成像等领域。为加速对多维数据的查询和操作,研究员已提出许多成熟的索引结构和相关算法,例如网格文件、EXCELL、多维可扩展哈希、KD 树和 R 树等[1]。与一维索引数据结构相比,多维索引数据结构更复杂、数据量和计算量更大,因此如何实现多维索引数据结构的并行化处理一直是数据库中极具挑战性的难题。

近年来,各种新型高性能计算体系机构不断涌现。在众多异构混合平台中,由于 GPU 具有强大的并行吞吐量和高可用带宽,因此基于 CPU/GPU 异构协同的计算平台具有很大的发展前景[2]。为此,本文在 CPU/GPU 异构平台上提出一种多维线性哈希的并行计算方案。通过实验对比发现,该方案在批量处理多维数据时,性能明显得到提高。

2 线性哈希索引

2.1 一维线性哈希索引

线性哈希索引[4]是一种非常有效的内存索引结构[5],它可实现索引数据的快速存取和检索,其算法为:假设最初线性哈希表由 N 个桶组成,用 $0,1,\cdots,N-1$ 对它们进行编号。给定一个记录 r 的键值 k,通过哈希函数 $h_0(k)$ 可计算出 r 在哈希表中对应的桶号。把新记录插入到一个已无存储空间的哈希桶时,会产生冲突。线性哈希解决冲突的策略是进行桶分裂,分裂以线性的次序从 0 号桶至 $N-1$ 号桶进行。首先在哈希表中创建一个编号为 N 的桶,然后使用哈希函数 $h_1(k)$

到稿日期:2011-12-15 返修日期:2012-2-04 本文受广西壮族自治区 2011 年企业技术改造资金项目计划(软件网页化技术研究与产业化),广西自然科学基金(2012GXNSFBA053162)资助。

刘 勇(1973-),男,博士生,讲师,主要研究领域为数据库与高性能并行计算,E-mail:l. yong07@mail. scut. edu. com;赵秦德(1980-),男,硕士,主要研究领域为互联网和计算机应用技术;赖正文(1974--),男,博士生,讲师,主要研究领域为模式识别与高性能并行计算。

重新计算 0 号桶的记录，并根据计算结果把桶内部分记录移到第 N 号桶中。所有 N 个桶被分裂完后，哈希表就实现了一个完整的扩展，即表空间由 N 扩充为 $2N$ 个桶，同时所有桶的哈希函数从 h_0 变为 h_1。目前，常用的哈希函数为：

$$h_i: k \rightarrow \{0,1,\cdots,2^i * N-1\} \quad (1)$$

传统的线性哈希表还有 2 个变量：Next，指向下一个被分裂的桶；Level，表示地址空间扩大的倍数。

2.2 多维线性哈希索引

多维线性哈希索引[6,7]把 Litwin 的线性哈希索引扩展到多维空间，并保留了一维线性哈希所有的优点。其算法思想为：假设键值 $k=(k_0,k_1,\cdots,k_{d-1})$ 是一个 d 维空间的向量，每一维属性则对应键值空间的一个轴。当把记录插入到哈希表时，对每个轴执行一维线性哈希的分裂方法，且分裂只依赖当前分裂轴的属性值。假定哈希表最初有 N 个主桶，则前 N 个冲突沿着 0 轴进行线性分裂。当完成这个轴的分裂后，哈希表的主桶数扩大一倍；如果再发生冲突，则分裂将沿 1 轴进行，直到哈希表空间再扩大一倍。哈希表以循环的方式选择下一个分裂轴，例如当前分裂轴是 i，则下一个分裂轴是 $(i+1) \bmod d$，当完成一个分裂循环后，哈希表地址空间扩大 $2d$ 倍，分裂过程见图 1。

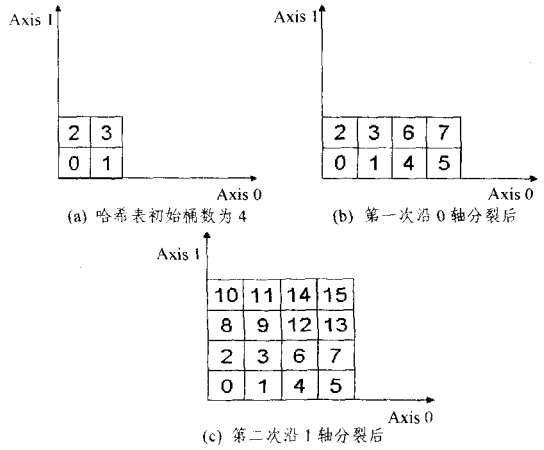

图 1 2 维线性哈希分裂过程

在计算一个多维记录 r 所属的哈希桶号时，通过使用存储映射函数 $M(H_L(k))$ 可把多维地址空间映射到一维地址空间[6]，具体是：首先调用式(1)计算多维键值 k 的每个维 k_j 的哈希值 b_j，然后根据 b_j 算出该记录所属的桶号：

$$M(H_L(k))=\sum_{j=0}^{d-1}\sum_{i=0}^{b_i^j\neq 0} 2^{(d*i+j)} b_i^j \quad (2)$$

式中，b^{ij} 表示 k_j 的哈希值中第 i 个 2 进制位值。例如：一个 2 维线性哈希索引每个维初始的主桶数均为 $N=4$，当前哈希表分裂级 $Level=5$，哈希桶分裂指针 $Next=300$，则键值为 $k=(110,250)$ 的记录所属哈希桶号 m 的计算步骤如下：

1) $H_5(k)=(h3(110),h_2(250))=(14,10)$

2) $M(14,10)=M(14,0)+M(0,10)$

$M(14,0)=84, M(0,10)=136$

记录 r 的哈希桶号：$m=220$。

3) 由于 $m<Next$，因此需重新计算哈希值：

$H_6(k)=(h_3(110),h_3(250))=(14,26)$

$M(14,26)=M(14,0)+M(0,26)$

$M(14,0)=84, M(0,26)=648$

最终记录 r 的哈希桶号：$m=732$。

3 异构平台的多维线性哈希方案

3.1 多维线性哈希表的数据结构

NVIDIA 提出的 CUDA-C 编程框架[9]不支持在 GPU 上动态分配空间，为此，本文设计一种适合 CPU/GPU 异构平台上并行处理的多维线性哈希表数据结构，见图 2。

```
struct MLH_TABLE //多维线性哈希表结构
 {
   bucket * primary;//哈希表主桶地址
   bucket * overflow;//溢出桶地址
   int * Level;//哈希表分裂的级数
   int * NextSplit;//指向哈希表下一个分裂桶
   int * records;//每个桶链的记录数
   int * OverflowBucketindex;//指向溢出桶下一个分配位置
   int * sum_overflowBuck;//统计需要的溢出桶总数
   int * insert_count;//每个桶链要插入的记录数
};
```

图 2 基于异构平台的多维线性哈希表数据结构

与传统的多维线性哈希表数据结构相比，新的结构增加了两个整型指针变量：insert_count，用于统计每一个哈希桶链将要插入的记录数；sum_overflowBuck，根据 insert_count 汇总哈希表需要分配的溢出桶总数。

3.2 哈希表构建算法

基于 CPU/GPU 异构平台的哈希表批量构建算法包括以下 3 个步骤。

1)多维哈希表初始化。这是构建哈希表的第一步，首先设置多维线性哈希表每个维的初始桶数 DIM_BUCKET 和每个哈希桶的容量 CAPACITY，然后从 host 端[9]为 device 端上的哈希表的 primary、records、sum_overflowBuck 和 insert_count 等变量分配存储空间，并初始化哈希表部分变量值。

2)溢出桶分配。根据式(2)，首先计算每个待插入记录所属的哈希桶号，再调用 CUDA 提供的原子函数 atomicAdd 把计算结果累加至与该桶链对应的变量 insert_count，由此统计出每个桶链需要新增的溢出桶数；然后把这些新增的溢出桶数汇总至 sum_overflowBuck，据此可为哈希表分配新的溢出桶存储空间，最后在 GPU 上为每个需要增加溢出桶的桶链分配溢出桶。溢出桶分配的部分算法伪代码如下。

算法 1 bucket_assign (m_table) on GPU

```
1. add_buckets=insert_count[tid]/CAPACITY-1;
   //tid 是 GPU 线程号，在此表示哈希表的一个桶链
2. while add_buckets>0 do
3. location←atomicAdd(m_table. OverflowBucketindex,1);
   //计算溢出桶下一个可分配的地址
4.   c_bucket←m_table. Overflow[location];
   //从溢出桶地址数组中取一个桶地址
5.   p_bucket->next=c_bucket;
     // p_bucket 为指向桶链的最后一个桶地址
6.   p_bucket=c_bucket;
7.   add_bucknumber--;
8. end while
```

3) 记录批量插入哈希表。算法根据式(2)首先计算出每个记录所属的哈希桶号，然后在哈希表中根据桶号取出对应

的主桶地址，接着计算该桶链中下一个空的存储位置，最后把记录插入到哈希表中。算法部分伪代码如下。

算法 2 insert_table(m_table, int records[ ][DIM])

```
input: m_table, 多维线性哈希表
       records, 本次批量插入的多维记录数组
1. BucketNumber←M_hashfun(records[tid], * (m_table. Level));
   //计算键值其所属桶号
2. c_bucket←&(m_table. primary[BucketNumber]);
   //取出桶号对应的主桶地址
3. insert_location←atomicAdd(&(m_table. records[BucketNumber]),1);
   //计算该桶链下一个空的存储位置
4. bucket_index←insert_location/CAPACITY;
   // 计算桶链的桶索引号
5. location←insert_location% CAPACITY;
   // 计算目标桶中的下一个空的存储位置
6. for i←0 to bucket_index do
7.    c_bucket←c_bucket->next; //移动桶链
8. end for
9. c_bucket->records[location]←record_values[tid];
   // 记录插入到哈希表中
```

3.3 哈希表查询算法

算法首先根据查找键值在表中找到对应的哈希桶号，然后根据桶号取出对应的桶链地址，接着在桶链中遍历哈希桶，判断该键值是否已存在哈希表中，最后返回查询的结果。算法部分伪代码如下：

算法 3 Search(m_table, key[ ][DIM], result)

```
input: m_table, 哈希表
       key, 多维键值数组
output: result, 保存查询的结果
1. BucketNumber←M_hash(key[tid], * (m_table. Level));
   //计算查询键所属桶号
2. c_bucket←&(m_table. primary[BucketNumber]);
3. while c_bucket ! = NULL do
4.    for j←0 to CAPACITY do //遍历哈希桶
5.       if (Compare(key[tid], c_buket->records[j]) then
            //判断查找键是否在哈希桶中
6.          result[tid]=1; break;
7.       end if
8.    end for
9.    c_bucket←c_bucket->next;
10. end while
```

4 实验测试及分析

为验证提出方案的有效性，在 CPU 的单线程、两线程(DUAL)、四线程(FOUR)以及 GPU 等平台上进行哈希表的批量构建和查询实验。实验的硬件平台：GPU：NVIDIA GTX 550Ti；CPU：intel core i5 2.53G Hz。软件环境：操作系统 WIN7；多核 CPU 算法采用 OpenMP3.0 规范[8]；GPU 编程使用 CUDA 4.0 开发包。

实验的多维数据采用随机函数产生，每条记录由一个 d 维键值向量空间构成。由于篇幅限制，本文只对 2 维和 3 维的键值向量空间进行实验。实验中设置的主要参数：哈希桶的容量 CAPACITY 为 128，每个维度初始主桶数 DIM_BUCKET 为 64 等。

4.1 实验结果和分析

在图 3 创建的 2 维线性哈希表中，当键值数量从 5M 增大到 30M 时，基于 CPU 的传统方案创建哈希表所需时间急剧增长，而在 CPU/GPU 异构平台上创建哈希表所需的时间则缓慢增加，其相对各 CPU 线程的处理方案的加速比也在不断提高。

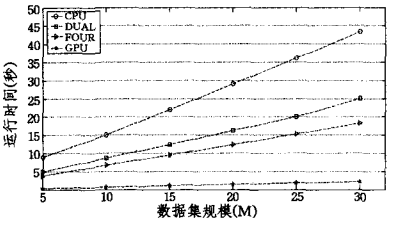

图 3 多个平台上创建 2D 哈希表性能对比

图 4 是创建 3 维线性哈希表的性能对比。当键值数从 5M 增大到 30M 时，基于单线程 CPU 的方案创建哈希表所需时间从 8 秒增长到 48 秒多，即使采用 4 线程的 CPU 方案，创建 30M 的哈希表时间也需要 21 秒。而在 CPU/GPU 异构平台上的方案则需 2 秒就能完成键值数为 30M 的哈希表的创建。

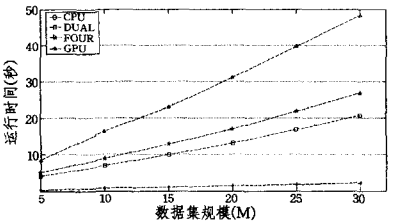

图 4 多个平台上创建 3D 哈希表性能对比

图 5 是 3 维线性哈希表的数据查询性能对比，实验设置哈希表的键值总量是 20M，查询的键值规模从 2M 增大到 10M。在图 5 中使用传统的单线程 CPU 查询 2M 数据需要 6 秒，查询 10M 数据需要 33 秒；使用双线程查询 10M 数据需要 17 秒，而使用四线程查询 10M 数据也需要 11 秒多。但在 CPU/GPU 异构平台上查询 2M 数据仅需要 170 毫秒，查询 10M 数据所需时间为 900 毫秒。由此可见，利用 GPU 强大的并行计算能力和高可用带宽，可大大提高通用计算的效率。

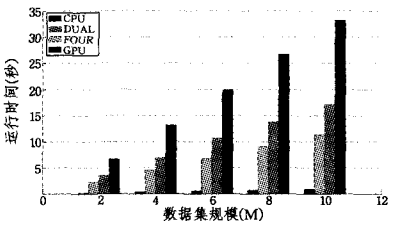

图 5 3D 哈希表多维数据查询性能对比

结束语 提出一个 CPU/GPU 异构平台上多维数据的并行处理方案，并在多个处理器平台上对方案的有效性进行了充分的实验论证。结果表明，在异构平台上创建哈希表相比传统的单线程 CPU 方案获得 25 倍以上的加速比，相比四线程的 CPU 方案也获得 10 倍的加速比；在查询方面，异构平台查询多维数据相比传统的四线程 CPU 方案达到 13 倍的加速比。在以后的工作中将研究 CPU/GPU 异构平台上并行

(下转第 163 页)

表中的第一列代表数据集的名称，是实验中选取的4个数据集，第二列时间差的含义是改进算法在挖掘频繁项集耗费的总时间减去原始FP-Stream算法耗费的总时间，单位是ms。表中的时间差是相对于整个数据集而言的，当平均到每批数据上，其值是很小的，几乎可以忽略，以pumb_star为例，每批数据的处理时间差值是8ms左右，这个时间差值完全在可接受的范围之内，因此算法在时间上是可行的。

通过结果集可以看出，改进算法在事务集读入内存后占用的存储空间较原始算法有很大程度上的提高，且稳定性并没有受到影响。显然改进算法可以提高数据处理能力，扩展FP-Stream算法的应用。

结束语 通过实验结果分析可知，改进的FP-Stream算法具有更高的数据处理能力，但是仍然存在需要改进的地方，对于FP-Tree和Pattern-tree这两个主要的数据结构来说，对内存的需求仍然很大，因此，提出针对该数据结构的进一步压缩算法，将会进一步改善算法的效率。前面提出的对倾斜时间窗口存储的压缩是一个方面。此外，在FP-Tree产生频繁项集的算法中，大量的中间结果也会造成内存紧张，这方面需要进一步改进。也有很多学者提出了相关的改进算法，如文献[11]的LR-Trie和文献[10]的NBFP-Tree等。提出更高效的算法将是我们下一步的工作。

参 考 文 献

[1] Agrawal R, Srikant R. Fast Alogoritm for mining Association Rules[C]// Proc 20th Int Conf Very Large Data Bases VLDB (1994). Volume 1215, Citeseer, 1994: 487-499

[2] Han Jia-wei, Pei Jian, Yin Yi-wen. Mining Frequent Patterns without Candidate Generation[C]// SIGMOD '00 Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data. ACM, New York, NY, USA, 2000

[3] Pei Jian, Han Jia-wei. CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets[Z]. DMKD. May 2000

[4] Wang Jian-yong, Han Jia-wei, Pei Jian. CLOSET+: searching for the best strategies for mining frequent closed itemsets[Z]. ACM SIGKDD, August 2003

[5] Zaki M J, Hsiao C-J. CHARM: An efficient algorithm for closed association rule mining[C]// 2nd SIAM International Conf. on Data Mining. 1999

[6] Giannella C, Han Jia-wei, Pei Jian, et al. Mining frequent Patterns in Data STream at Multiple Time Granularities[J]. Next generation data mining, 2006, 35(1)

[7] Xenfontas, Hurley P, Kind A. Probabilistic lossy counting: an efficient algorithm for finding heavy hitters[J]. ACM SIGCOMM Computer Communication, 2008, 38(1)

[8] 周莉，张吉赞. 用垂直格式构建FP增长树的算法[J]. 计算机科学与工程，2009，45(8)：161-164

[9] Ashrafi M, Taniar D, Smith K. An efficient compression Technique for vertical Mining Methods[M]. Research and Trends in Data Mining technologies and applications, 2007: 151-163

[10] 吕橙，郝莹，张翰韬. 基于垂直二进制位图的频繁模式挖掘算法[J]. 山东大学学报，2007(5)：24-29

[11] 徐艳红. 基于倾斜时间窗口的频繁项集挖掘算法研究[D]. 哈尔滨：哈尔滨工程大学，2010

[12] 孟彩霞. 面向数据流的频繁项集挖掘研究[J]. 计算机工程与应用，2010，46(24)

[13] Manku G S, Motwani R. Approximate frequency counts over data streams[C] // Proceedings of the 28th VLDB Conference. HongKong, China, 2002: 346-357.

[14] Carmona J M, Baena-García M, Campo-Avila J, et al. Using GNUsmail to Compare Data Stream Mining Methods for On-line Email Classification[C] // Journal of Machine Learning Research-Proceeding Track. 2011: 12-18

[15] 孙志长，冯祖洪，王沛栋. 一种高效的混合压缩数据挖掘算法[J]. 计算机应用研究，2009(10)：3738-3742

[16] Han Jia-wei, Kamber M. Data Mining Concept and Techniques (Second Edition)[M]. Morgan Kaufmann, 2006

[17] Ren J, Li K. Online data stream mining of recent frequent itemset based on sliding window model[J]. Proceedings of 2008 International Conference on machine and Cybernetics, 2008, 1(7)

[18] Leung C K-S, Brajczuk D A. Efficient Mining of frequent itemsets from data streams[J]. Information and Knowledge, LNCS, 2008, 5071: 2-14

[19] Jea K-F, Li Chao-wei, Chang T-P. An efficient approximate approach to mining frequent itemsets over high speed transactional data streams[C]// ISPA 2008: English International Conference on intelligent Systems Design and Applications. 2008, 3: 275-280

(上接第159页)

算法的时空分析模型。

参 考 文 献

[1] Gaede V, Gunther O. Multidimensional Access Methods [J]. ACM Computing Surveys, 1998, 30(2)

[2] 卢凤顺，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] 杨珂等. 图形处理器在数据库技术中的应用[J]. 浙江大学学报：工学版，2009，43(8)：1349-1360

[4] Litwin W. Linear hashing: a new tool for file and table addres-sing [C]// VLDB1980. Montreal, 1980: 212-223

[5] Lehman T J, Michael J. A Study of Index Structures for Main Memory Database Management Systems[C] // Proceedings of the 12th International Conference on Very Large Data Bases. August 1986: 294-303

[6] Ouksel A M, Scheuermann P. Storage Mappings for Multidimensional Linear Dynamic Hashing[C]// Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems. 1983: 90-105

[7] Ouksel A M, Abdul-Ghaffar J. Concurrency In Multidimensional Linear Hashing[C]// Proceedings of FODO. 1989: 233-240

[8] 周伟明. 多核计算与程序设计[M]. 武汉：华中科技大学出版社，2009：19-25

[9] NVIDIA CUDA (Compute Unified Device Architecture)[OL]. http://developer.nvidia.com/object/cuda.html