华南理工大学学报(自然科学版)
第43卷 第1期 Journal of South China University of Technology (Natural Science Edition) Vol. 43 No. 1
2015年1月 January 2015

文章编号:1000-565X(2015)01-0111-07

基于GPU的可扩展哈希方法*

胡学萱[1] 奚建清[2] 林妙[2]
(1. 华南理工大学 计算机科学与工程学院,广东 广州 510006; 2. 华南理工大学 软件学院,广东 广州 510006)

摘 要:为了使用可扩展哈希表进行快速的数据访问,需要高效地更新索引以维护哈希表.文中提出了一种基于GPU的可扩展哈希算法gEHT.该算法充分利用GPU的并行计算能力,并采用表重用、预分裂技术,无锁地扩展和收缩表、插入和删除数据,实现了高并发地创建哈希表、更新索引和检索数据.实验结果表明,该算法的查询数据、维护哈希表和更新索引性能优于其他多核CPU的线性哈希及可扩展哈希算法,尤其是在高负载的情况下.

关键词:可扩展哈希;并行计算;GPU;算法;多核CPU
中图分类号:TP312 doi:10.3969/j.issn.1000-565X.2015.01.018

哈希索引是一种快速的数据检索方法,已广泛应用于文件系统、数据库系统及一些情报检索系统中[1].动态哈希能优雅地扩展存储空间,根据其方法的不同,可分为可扩展哈希[2]和线性哈希[3].如何扩展和收缩哈希表,快速更新索引记录以便有效地使用哈希索引,一直以来受到人们的广泛关注.Ellis[4]提出了一种基于目录锁和桶锁的两级锁模式的可扩展并行哈希算法;Hsu等[5]提出了一种可以并发插入删除的可扩展哈希算法;Kumar[6]提出的EHCW算法使用了两阶段锁和事务回滚的策略,可并发扩展/收缩目录,分裂/合并数据页;Lea[7]提出了一种基于复杂锁模式的Java并发包,允许在改变表大小的同时进行并行查询;Gao等[8]提出的可扩展哈希算法周期性地切换到全局以调整哈希表大小状态,进行多进程协作执行数据项迁移;Shalev等[9]提出的RSO算法通过改变桶地址来扩展或收缩表;Zhang等[10]提出了基于线性哈希和RSO算法的LHlf算法;陈虎等[11]提出了一种利用多核处理器的并行计算能力提升批量插入线性哈希表性能的算法;黄玉龙等[12]提出了一种图形处理器(GPU)加速的线性哈希表的批量插入算法GBLHT.

高性能和可扩展的哈希表是当代大数据应用的需求,故需要提高哈希表的并发处理能力.可编程图形处理器因其众核和高存储器带宽而成为并行计算的超强工具,统一计算设备架构(CUDA)这种基于GPU的计算架构,提供了易用的编程模型和应用程序接口(API),使得将GPU用于通用计算成为可能[13].为充分利用GPU的高并发计算能力,提升可扩展哈希方法的性能,文中提出了一种基于GPU的可扩展哈希方法gEHT.

1 相关的哈希方法

1.1 可扩展哈希方法

可扩展哈希方法分裂和合并桶时,不像线性哈希方法那样按顺序进行,而是按需要进行[2],即当向桶内插入的数据量超过桶的剩余容量时,就分裂该桶,一次分裂产生的两个桶互为对应桶;反之,当桶及其对应桶内数据被删除后不足一个桶的容量时,

收稿日期:2014-05-16
* 基金项目:广东省战略性新兴产业核心技术攻关项目(2011A010801008,2012A010701011,2012A010701003);广州市科技计划项目(201200000034)
Foundation items: Supported by the Guangdong Strategic Emerging Industries Core Technology Key Project (2011A010801008, 2012A010701011, 2012A010701003)
作者简介:胡学萱(1975-),女,博士生,主要从事数据库、并行计算研究.E-mail:wxfhxx@163.com

就合并这两个桶.

图1给出了一个初始桶经过多次分裂形成的分裂树,其中树根为初始桶,实节点为已分裂产生的桶,称为实桶,虚节点为预分裂产生的桶,称为虚桶;实有向边为已产生的分裂,虚有向边为预产生的分裂.桶分裂一次,产生实有向边所指的下一层的两个桶,最底层的所有实桶为现有桶.可见,桶000、001、100、101、10和11为现有桶.

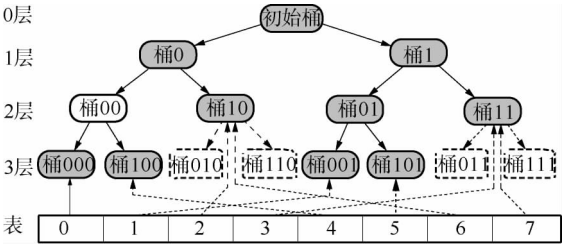

图1 桶的分裂树

Fig.1 Split tree of bucket

由于各桶的层次无序,必然导致有些目录项所指的桶为虚桶,解决的方法是将这样的目录项指向该虚桶的实上级桶,即分裂树上该虚桶到初始桶的分裂路径上最大深度的实桶.这样,在检索数据时,能一次定位到不同局部深度的桶上,而不用层层探索,加快了检索速度.

1.2 基于并行技术的动态哈希

文献[4-7]中提出基于锁的算法,具有死锁、长时间延迟和优先级倒置的缺点,当进行表的扩展或收缩时,这些情况更加严重.文献[8]算法使用了write-all算法,时间复杂度为常数.RSO算法[9]使用链表来存储数据,故其访问效率低,仅适用于将关键字的低位作为最高有效位的哈希函数.文献[10-12]算法使用了溢出桶,降低了检索效率,并且延迟的分裂使得分裂前溢出桶的数据要重新迁移到分裂后的桶中,这种冗余的数据迁移必然会降低插入的效率.

2 可扩展哈希方法在GPU上的实现

2.1 基于GPU的存储结构

为了在GPU上动态伸缩表并克服可扩展哈希表成倍扩展的缺陷,文中采用两层表结构,分别是段表segList和桶表bucketList,段表存放段地址,桶表存放桶地址.以桶表的伸缩实现少量的表扩展,段表的重构实现大量的表扩展[12].当扩展的桶集中在少数的段时,只需扩展少数的桶表,其他未扩展桶表可被重用.另外,段表、桶表和桶内数据都采用数组(即便于GPU并行处理的结构),并且桶内数据采用数组的结构体(SOA)结构而非结构体的数组(AOS),以适应GPU的存储器的优化存取要求.文献[12]的两层存储结构仅用于动态伸缩.文中采用的存储结构如下:

```
struct extendiblehashtable{
    segment **segList;
    int Gd;
    int bucketNum;}
struct segment{
    bucket *bucketList[N];}
struct bucket{
    recorde Data;
    int insertLoc;
    int Ld;
    int is_df[b];}
struct recorde{
    int key[b];
    int value[b];}
```

上述定义中,Gd为全局深度,bucketNum为现有桶的数量,Ld为桶的局部深度,insertLoc为桶内下一个空闲地址,数组is_df是每条数据的删除标记.初始时有M段,每段由N个桶构成,每个桶的容量为b条记录.维护这种结构的挑战在于保证桶分裂/合并和表扩展/收缩后,目录项能指向改变后的桶.表及桶的改变分为以下4种情况:

(1)桶分裂,段表不扩展.如果桶$b[i]$分裂产生新桶$b[i+M\cdot N\cdot 2^{b[i].Ld}]$,分裂后所有桶的最大局部深度未超过全局深度,则段表不扩展.如果没有第$(i+M\cdot N\cdot 2^{b[i].Ld})/N$段桶表则增加它,修改该桶表第$(i+M\cdot N\cdot 2^{b[i].Ld})\%N$项指向新桶,其他项指向相应虚桶的实上级桶.桶$b[i]$和$b[i+M\cdot N\cdot 2^{b[i].Ld}]$的局部深度为$b[i]$.Ld+1.

(2)桶合并,段表不收缩.若桶$b[i]$和$b[i+M\cdot N\cdot 2^{b[i].Ld}]$合并,合并后所有桶的最大局部深度不小于全局深度,则段表不收缩.释放桶$b[i+M\cdot N\cdot 2^{b[i].Ld}]$,修改桶表目录项$i+M\cdot N\cdot 2^{b[i].Ld}$指向桶$b[i]$,桶$b[i]$的局部深度为$b[i]$.Ld–1.若第$(i+M\cdot N\cdot 2^{b[i].Ld})/N$段桶表所有表项指向的桶的局部深度都小于全局深度,则释放该段桶表.

(3)段表扩展.若分裂后所有桶的最大局部深度大于全局深度,则段表扩展.扩展后的第$(i+M\cdot N\cdot 2^{b[i].Ld})/N$项指向新增的桶表,其他表项指向其实上级桶所在的桶表.

(4)段表收缩.合并后所有桶的最大局部深度小于全局深度,则段表收缩.

图2为桶分裂/合并和表扩展/收缩的一个例子.

图 2(a)所示的桶经过收缩和扩展，产生图 2(b)、2(c)所示的桶和表.其中，图 2(a)的桶 00 和 10、01 和 11 分别合并为图 2(b)的两个桶 0 和 1，则原指向桶 00、10 的表项合并后指向其实上级桶 0，指向 01、11 的表项合并后指向其实上级桶 1，桶的最大局部深度小于合并前的全局深度，段表和桶表收缩为原表的一半.图 2(a)的桶 00 扩展为图 2(c)的桶 000 和 100，桶的最大局部深度大于原全局深度，段表扩展一倍，桶表扩展第 2 段，使段表的第 2 项指向新增的桶表，第 3 项指向第 1 段桶表.新增的桶表中，第 4 项指向新增的桶 100，其他扩展的表项 5 指向虚桶 101 的实上级桶 01.可见，采用这种目录结构虽然扩展了指向新增桶的桶表，但能寻址倍增的桶(包括虚桶).

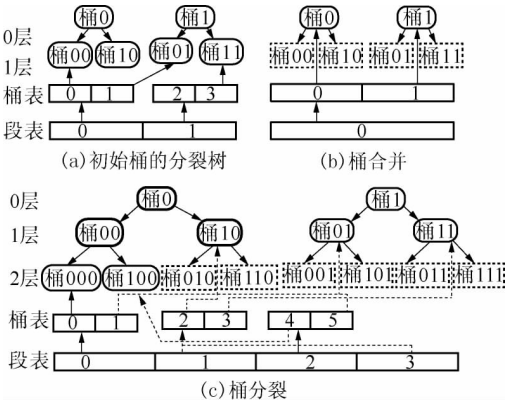

图 2　表的收缩和扩展

Fig. 2　Shrinking and growing of table

2.2　创建哈希表算法

哈希索引的创建包括创建段表、桶表和桶，并使段表和桶表的各项指向新创建的桶表和桶.算法并行创建段内的桶，伪代码如下：

```
createEHTable(eHTable){
CR1 Allocate an array segList of length M;
CR2 For each i∈[0,M) do
CR3   Allocate an array bucketList of length N;
CR4   segList[i] ← &bucketList;
CR5   For each tid∈[0,N) parallel do //tid 为线程号
CR6     Allocate a bucket and initialize it;
CR7     bucketList[tid] ← &bucket;
CR8   End for
CR9 End for}
```

2.3　查询算法

检索数据先计算该数据的散列值，然后定位到桶中并在桶中搜索该数据.在 GPU 中，可使每个数据用一个线程处理，高并发地执行.文献[12]的查询算法是基于线性哈希的，因而，对每个数据的查询，延迟比可扩展哈希大.文中查询算法的伪代码如下：

```
queryEHTable(eHTable,eData,qResult){
Q1 Allocate a variable bucketNo and an array qResult;
Q2 For each tid∈[0,len(eData)) parallel do
Q3   bucketNo← hash(eData[tid].key);
Q4   For each i∈[0,eHTable.segList[bucketNo/N]->
       bucketList[bucketNo%N]. insertLoc) do
Q5   if(eHTable.segList[bucketNo/N]->bucketList [bucketNo%
       N].eData[i].key == eData[tid]. key ){
Q6       qResult[tid] = 1;
Q7       return;}
Q8   End for
Q9   qResult[tid] = 0;
Q10  End for}
```

2.4　删除算法

本算法采用延迟删除的策略，即删除并不立即执行，只对删除数据做标记.在批量插入数据时，用插入数据覆盖有删除标记的数据.该算法只需要在查询算法 queryEHTable 中加入对要删除的索引记录做标记，文中不再赘述.

2.5　插入算法

当批量插入的数据量大于该桶的剩余容量时，需要分裂桶，如果分裂后仍不能满足插入需要，则要继续分裂直到满足需求为止.分裂会使部分数据迁移到新桶中，这些中间过程的数据迁移是不必要的，会降低索引更新速度，成为插入过程的性能瓶颈[11].因此，文中采用预分裂技术，先循环预测桶的分裂情况，再分裂桶或扩展表并插入数据；充分利用 GPU 的计算能力，并行处理预测过程和实际插入过程以提高插入效率.

批量插入算法流程如图 3 所示，预测部分和插入部分的算法主要由子算法 A、B、C、D 构成.插入算

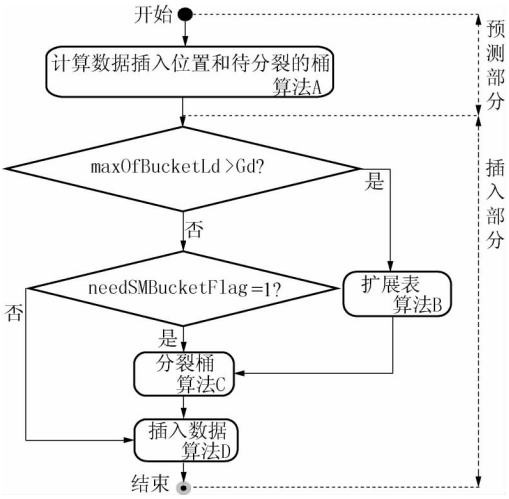

图 3　批量插入算法流程图

Fig. 3　Flowchart of batch insertion algorithm

法的步骤如下:

(1)计算数据插入的桶号 insert_bucketNo、各桶插入数据量 insert_bucketNum、待分裂的桶号 needSMBucketNo、待分裂的段号 needSMSegNo、桶预计分裂到的局部深度 SMBucketLd、待分裂桶的数量 needSMBucketNum、所有待分裂桶预计分裂到的最大局部深度 maxOfBucketLd.若待分裂桶的数量 needSMBucketNum>0,则置待分裂桶标记 needSMBucketFlag=1,并根据上轮循环预计的桶分裂到的局部深度 SMBucketLd,循环计算以上数据,直到 needSMBucketNum=0,如算法 A(countInsert).

(2)若所有待分裂桶预计分裂到的最大局部深度 maxOfBucketLd>Gd,则转步骤(3),否则转步骤(4).

(3)扩展段表和桶表,将扩展的段表表项指向新建的桶表,如算法 B(growsTable).

(4)若有需要分裂的桶,即 needSMBucketFlag=1,则转步骤(5),否则转步骤(6).

(5)分裂桶,修改桶表指针,如算法 C(splitBucket).

(6)插入数据,如算法 D(insertData).

如 2.4 节所述,插入数据将覆盖有删除标记的数据,因此批量插入数据除了上述情况外,也会有合并桶和收缩表的情况.限于篇幅,以下伪代码仅为分裂桶和扩展表的情况:

```
countInsert(eHTable, eData, insert_bucketNo,insert_bucketNum){
A1   while (needSMBucketNum > 0){
A2    For each tid∈[0,len(eData)) parallel do
A3      bucketNo← hash(eData[tid].key);
A4      insert_bucketNo[tid]←bucketNo;
A5      atomicAdd(insert_bucketNum[bucketNo],1);
A6    End for
A7   For each tid∈[0,M·N) parallel do
A8   needSMBucketNo[tid] =(insert_bucketNum[tid]>b–
       eHTable.segList[tid/N]->bucketList[tid%N].insertLoc);
A9   SMBucketLd[tid] = SMBucketLd[tid]+
         needSMBucketNo[tid];
A10  needSMSegNo[tid/N]= needSMBucketNo[tid];
A11  atomicMax(maxOfBucketLd,SMBucketLevel[tid]);
A12  atomicAdd(needSMBucketNum,needSMBucketNo[tid]);
A13  End for}}
growsTable(eHTable,needSMSegNo,needSMBucketNo,
            maxOfBucketLd,SMBucketLd){
B1 Allocate an array nSegListof of length M·2^maxOfBucketLd;
B2 For each tid∈[0,M) parallel do
B3  For each i∈[0,maxOfBucketLd–eHTable.Gd) do
B4    nSegList[tid + i·M·2^eHTable.Gd] ←segList[tid];
B5  End for
B6  if(needSMSegNo[tid] > 0){
B7  For each j∈[0,SMBucketLd[tid]–eHTable.Gd)do
B8      Allocate an array nBucketList of length N;
B9      nSegList[tid + j·M·2^eHTable.Gd]←&nBucketList;
B10 End for}
B11 End for
B12  eHTable.segList ← &nSegList;
B13  eHTable.Gd = maxOfBucketLd;}
splitBucket(eHTable,needSMBucketNo,SMBucketLd){
C1 For each tid∈[0,M·N) parallel do
C2   if (needSMBucketNo[tid] > 0){
C3   For each i∈[0,SMBucketLd[tid]–eHTable.Gd)do
C4      Allocate a bucket and initialize it;
C5      bucketList[tid + i·M·2^Gd] ← &bucket;
C6   End for}
C7 End for}
insertData(eHTable,eData,insert_bucket No){
D1   sort eData by thrust::sort_by_key;
D2   For each tid∈[0,len(eData)) parallel do
D3      insert eData into eHTable;
D4   End for}
```

2.6 算法优化

根据 CUDA 的编程模型,将计算密集的任务交给 GPU 线程网格处理,如核函数 A、B、C、D,而 CPU 执行逻辑复杂的处理.在 GPU 上的并行优化主要从以下几个方面考虑:

1)任务粒度划分.粒度越细,越能充分利用 GPU 大量的轻量级线程,算法的并行度越高.文中的创建算法中,桶间并行执行;查询算法中,各数据可并行查询;插入算法的预测部分中,对各数据的插入位置、各桶分裂情况的计算相互独立;在扩展表、分裂桶和插入数据算法中,段间、桶间和数据间的操作相互独立,蕴含大量并行操作,如 For each tid… parallel do 与 End for 之间的代码所示.

2)程序重构.GPU 的体系结构使其适于进行逻辑简单和数据并行的密集计算.文中合并多个循环以增强计算密度,隐藏访存延迟;重构分支以避免 warp 内分支转移造成性能下降;将不同粒度的并行操作合并为一个核函数,以增强计算密度、减少核函数启动开销和数据传输开销.

计算每个桶是否分裂、分裂到的层次等,其迭代空间是所有的桶,因而可合并循环,如 A7–A13 行;计算每个数据的插入位置与计算每个桶是否需要分裂,其任务粒度不同,故不能合并循环,但可以合并为一个核函数,以便将待插入数据 eData 放入共享存储器重用,如算法 A.计算桶是否分裂以及桶的层数,需要根据插入数据量与桶的剩余容量大小比较进行分支选择,文中重构该分支语句,将比较插

入数据量与桶的剩余容量大小的逻辑值赋给记录桶是否分裂的数组 needSMBucketNo，并将该数组作为桶层数的增量，如 A8、A9 行.

3）数据的组织与存储访问.适合并行处理的数据结构、对 GPU 上各存储器的充分利用以及规则的访问模式是存储优化的主要方法.

（1）数组能体现数据的并行性，适于 GPU 上的密集计算.文中对哈希表结构、算法的输入数据 eData、中间变量（如 insert_bucketNo、insert_bucketNum、needSMBucketNo、SMBucketLd 等）及输出结果 qResult 都使用数组.为了全局存储器的合并访问，尽量将数据由 AOS 转为 SOA 结构[14]，如桶内数据 Data 和输入数据 eData.文献[12]采用数组结构，但没有将 AOS 转为 SOA 结构.

（2）GPU 的多层次存储器可满足不同需求，共享存储器访问速度快但容量有限，为块内线程所共有，适合存放重用的局部性数据；全局存储器容量大但速度慢.高效利用片上局部存储器，可提高计算全局访存比 CGMA [13]，且采用规则访问的模式能显著改善程序的存储墙问题.如算法 A 中，待插入数据 eData 循环重用，宜用共享存储器存储.从上述算法可以看出，各数组的访存模式中，除 insert_bucketNum 为随机访存，needSMSegNo 为跳步访存外，其他都是规则访问，使得全局存储器能合并访问，不产生局部访存冲突，而这两个数组的非规则访存，并不适宜进行循环重构和数组重构[15]来改变其访存模式.文献[12]未提到共享存储器的使用以及数据的规则访问.

4）使用 cuda 提供的原子函数和并行算法库以提高代码的性能，如 A5、A11、A12 和 D1 行所示.

3　算法性能分析

文中从时间和空间两方面讨论 gEHT 的性能.假设哈希表初始段数为 M，每段的桶数为 N，每个桶的容量为 b，现有桶的数量为 bucketNum，现有记录数为 m，批处理的数据规模为 batchNum.

3.1　时间开销

（1）查询时间开销.由 queryEHTable 算法可知，其时间开销 t_q 主要由计算搜索键对应散列值的时间 t_c、定位包含数据的桶以及在桶中检索目标数据的时间 t_s 构成.由 Q3 和 Q4–Q8 行知，t_c 的复杂度为 $O(1)$，t_s 的复杂度为 $O(b)$，因此，查询时间 t_q 的复杂度为 $O(b)+O(1)$，即为常数时间，且数据规模 batchNum 越大，越能充分利用 GPU 的并行计算能力，从而获得更高的吞吐量.

（2）插入时间开销.由 insertEHTable 算法可知，其总时间开销 t_i 主要由预测部分和实际操作部分各子算法的执行时间 t_A、t_B、t_C 和 t_D 组成.

子算法 A 循环执行，在最好情况下数据平均插入所有桶中，循环次数为 batchNum/(bucketNum·b)；在最坏情况下数据都插入同一个桶中，循环次数为 batchNum/b，其循环内部的 A2–A6、B7–B13 行并行执行，时间开销为 $O(1)$.算法 B、C 中 B2–C11、C1–C7 行也是并行执行，其并行内部循环次数与预测部分的循环次数相同，循环内部的时间开销为 $O(1)$.算法 D 中 D1 行的时间代价为 $O(\log_r \text{batchNum})$[14]（$r$ 为基数），D2–D4 行并行执行的时间开销为 $O(1)$.因此，总时间开销 t_i 在最好情况下为 $O(\text{batchNum}/(\text{bucketNum}\cdot b)+\log_r \text{batchNum})$，在最坏情况下为 $O(\text{batchNum}/b+\log_r \text{batchNum})$.

3.2　空间开销

文中从总空间开销和表扩展的开销两个方面来分析.

1）总空间开销.在 GPU 上建立的哈希索引，如 2.1 节所述，其显存的开销 S 主要包括段表空间 S_{segList}、桶表空间 $S_{\text{bucketList}}$ 和桶空间 S_{bucket}.

若 m 条记录占有桶数 $L(m)\approx m/(b\ln 2)$[16]，则占有空间 $S_{\text{bucket}}=bL(m)\approx m/\ln 2$，桶表所需空间 $S_{\text{bucketList}}$ 为 $O(m^{(1+1/b)}/b)$[17]，段表空间 S_{segList} 为 $O(m^{(1+1/b)}/(Nb))$，总空间 S 为 $O(m/\ln 2+m^{(1+1/b)}/b+m^{(1+1/b)}/(Nb))$.

2）表扩展的开销.假设分裂桶产生 a 个桶段的扩展，$0\leqslant a\leqslant M$（M 为扩展前桶的段数，即段表长度），则段表增加 M 项，桶表增加 aN 项（N 为每段桶表的长度），共扩展 $M+aN$ 项.若无段表仅有桶表，则表项要增加 MN 项.

（1）当数据倾斜，分裂的桶不多，即 $a\ll M$ 时，双层表结构中表的扩展是线性的，而单层表结构中表的扩展是二次的，更加剧烈；

（2）当数据均匀分布，每段都有桶分裂，即 $a\approx M$ 时，双层结构表的扩展接近 $M+MN$ 项，单层结构表的扩展接近 MN 项，而 $M\ll MN$，因此这种情况下，双层结构比单层结构多的表项也是有限的.

4　实验

实验在 CPU+GPU 的异构平台上进行，CPU 为 Intel Core i7-4770k，四核 3.50 GHz 主频，GPU 为 NVIDIA GeForce GTX 770，1536 CUDA Cores，每个核的频率为 1.19 GHz，显存容量为 4 GB.集成开发环境为 VisioStudio2010，GPU 开发工具包为 CUDA 5.5.

文中设计两部分实验，分别从时间和空间两个

方面检验 gEHT 算法的有效性.测试数据集是随机产生的键值对构成的记录,每条记录平均长度为 10 B,每桶的记录容量 b 为 64 条,初始桶的总数 $M\times N$ 为 8×1 024.

实验 1 对比 Lea 算法、RSO 算法、GBLHT 算法和 gEHT 算法在数据操作上的时间性能,其中 Lea 算法和 RSO 算法均采用 4 线程.

在包含 90%的查询数据、6%的插入数据和 4%的删除数据的负载下,4 种算法的吞吐量如图 4 所示.在 4 线程算法中,锁冲突对 Lea 算法带来的影响不大,其性能略高于 RSO 算法.RSO 算法和 gEHT 算法的性能都会随着数据量的增加而有所波动,这是因为表收缩或扩展的时间损耗使其吞吐量下降.GBLHT 算法采用的线性哈希及前述优化问题,使其性能略低于 gEHT 算法.总的来说,随着负载的增加,GBLHT 和 gEHT 算法充分利用了 GPU 的计算能力,显著地提高了吞吐量.

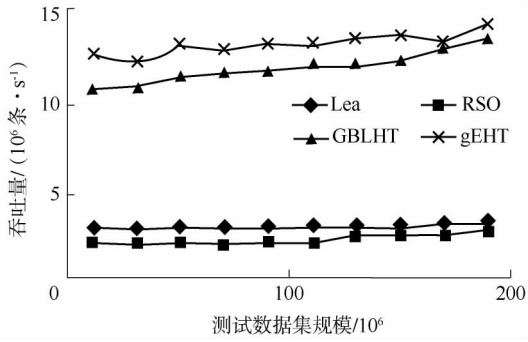

图 4 不同负载下 4 种算法的吞吐量

Fig. 4 Throughputs of four algorithms under different loads

实验 2 测试 Fagin 的 EH(Extendible Hashing)算法和 gEHT 算法随着数据的插入表扩展所占用的空间情况.

如图 5 所示,EH 算法在扩展表时,表长成倍增长,而 gEHT 算法接近线性的增长,并且随着插入数据的增加,表空间扩展的速度放缓,甚至小于 EH 算法的表空间大小.这是因为随着表空间的扩大,更多的段表项指向相同的桶表,越来越多的桶表被重用,以致尽管 gEHT 算法比 EH 算法多出段表,其总的表空间大小也不会超过 EH 算法.

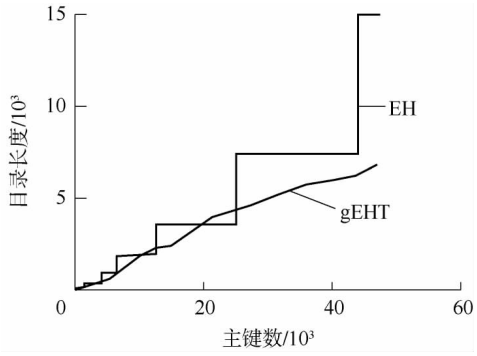

图 5 表的大小与插入数据量的关系

Fig. 5 Directory size versus number of insertions

5 结论

可扩展哈希是一种具有最快检索速度的动态哈希,文中利用 GPU 实现了可扩展哈希算法 gEHT.首先利用 GPU 的计算能力和 CUDA 的编程模型,设计并实现了哈希表的创建、索引的更新以及数据检索的高并发算法;为了克服表长增长剧烈的缺陷,文中采用二级表结构,使得段表能重用部分桶表,大大地节省了表结构对空间的需求.最后,通过实验验证了 gEHT 算法的性能.

显存空间十分有限,文中讨论的算法都是在待处理数据能够全部放在显存中的前提下进行处理的.大数据的特征使得批量处理的数据量更大,当其大于显存容量时,又应该如何在 GPU 上使用哈希索引,是下一步要考虑的问题.

参考文献:

[1] Du H C,Ghanta S,Maly K J,et al.An efficient file structure for document retrieval in the automated office environment [J].IEEE Transactions on Knowledge and Data Engineering,1989,1(2):258-273.

[2] Fagin Ronald,Nievergelt Jurg,Pippenger Nicholas,et al. Extendible hashing:a fast access method for dynamic files [J].ACM Transactions on Database Systems,1979,4(3):315-344.

[3] Lirwzn W.linear hashing:a new tool for files and table addressing [C]//Proceedings of the 6th Conference on VLDB.Montreal:VLDB Endowment,1980:212-223.

[4] Ellis C S.Concurrency in linear hashing [J].ACM Transactions on Database Systems,1987,12(2):195-217.

[5] Hsu M,Yang W.Concurrent operations in extendible hashing [C]//Proceedings of the 12th Conference on VLDB.Kyoto:Morgan Kaufmann Publishers Inc,1986:241-247.

[6] Kumar Vijay.Concurrent operations on extendible hashing and its performance [J].Communications of the ACM,1990,33(6):681-694.

[7] Lea D.Hash table util.con current.concurrent hash map,revision 1.25,in JSR-166,the proposed Java Concurrency package [EB/OL].(2013-12-01)[2014-03-10].http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/main/java/util/concurrent/.

[8] Gao H，Groote J F，Hesselink W H.Almost wait-free resizable hashtables [C]//Proceedings of the 18th International Parallel and Distributed Processing Symposium.Santa Fe：IEEE，2004：681-689.

[9] Shalev Ori，Shavit Nir.Split-ordered lists：lock-free extensiblehash tables (poster paper) [J].Journal of the ACM，2006，53(3)：379-405.

[10] Zhang D，Larson P-A.LHlf：lock-free linear hashing[C]// Proceedings of the 17th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming.New York：ACM，2012：307-308.

[11] 陈虎，唐海浩，廖江苗，等.面向批量插入优化的并行存储引擎 MTPower [J].计算机学报，2010，33(8)：1492-1499.
Chen Hu，Tang Hai-hao，Liao Jiang-miao，et al. MTPower：a parallel database storage engine for batch insertion [J].Chinese Journal of Computers，2010，33(8)：1492-1499.

[12] 黄玉龙，奚建清，张平健，等.GBLHT：一种 GPU 加速的批量插入线性哈希表 [J].华南理工大学学报：自然科学版，2012，40(4)：49-56.
Huang Yu-long，Xi Jian-qing，Zhang Ping-jian，et al. GBLHT：a GPU-accelerated linear Hash table with batch insertion [J].Journal of South China University of Technology：Natural Science Edition，2012，40(4)：49-56.

[13] NVIDIA Corporation.NVIDIA CUDA programming guide version 4.2 [EB/OL].(2012-04-16) [2014-03-10].http：//docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#ixzz3I5UFqJtq.

[14] Bell N，Hoberock J.Thrust：a productivity-oriented library for CUDA [EB/OL].(2012-12-10) [2014-03-10].http：//cloud.github.com/downloads/thrust/thrust/Thrust% 3A% 20A% 20Productivity-Oriented% 20Library% 20for% 20CUDA.pdf.

[15] Leung S-T.Array restructuring for cache locality [R]. Seattle：Department of Computer Science and Engineering，University of Washington，1996.

[16] Enbody R J，Du H C.Dynamic hashing schemes [J]. ACM Computing Surveys，1988，20(2)：85-113.

[17] Gonnct G H.算法和数据结构手册 [M].张子让，周晓东，译，北京：人民邮电出版社，1988.

Extendible Hashing Method Based on GPU

Hu Xue-xuan[1]　*Xi Jian-qing*[2]　*Lin Miao*[2]

(1. School of Computer Science and Engineering，South China University of Technology，Guangzhou 510006，Guangdong，China；
2. School of Software Engineering，South China University of Technology，Guangzhou 510006，Guangdong，China)

Abstract: In order to use an extended hash table for fast data accessing，an efficient index updating to maintain the hash table is necessary. Proposed in this his paper is a GPU-based extendible hashing algorithm named gEHT，which takes full advantage of GPU's parallel computing power，and adopts list reuse as well as pre-split technology to extend and merge lists and to insert and delete data in a lock-free manner. Thus，high levels of concurrency for hash table creation，index updating and data retrieval are realized. Experimental results show that gEHT is superior to some other linear hashing and extendible hashing algorithms on the basis of multi-core CPU in terms of querying data，maintaining hash table and updating index，especially in the case of heavy load.

Key words: extendible hashing；parallel computing；GPU；algorithm；multi-core CPU