DOI:10.19304/j.cnki.issn1000-7180.2014.12.001

基于图形处理器的高性能跳表(Skiplist)数据结构

李怀明,邓仰东
(清华大学 微电子学研究所,北京 100084)

摘 要: 提出了一种高效率、适合 GPU 的跳表结构及其相应例程,核心思想是将包含指针的操作转化为数组操作,从而充分发挥 GPU 的计算能力. 实验结果证明,该数据结构的插入和删除操作相对目前最好的 GPU 结果分别改善 6.8 倍和 9.6 倍.

关键词: 跳表;图形处理器;并行数据结构;无锁
中图分类号: TN402 **文献标识码**: A **文章编号**: 1000-7180(2014)12-0001-05

High Performance Skiplist for GPU Computing

LI Huai-ming, DENG Yang-dong
(Institute of Microelectronics, Tsinghua University, Beijing 100084, China)

Abstract: This work proposes a high performance skiplist data structure that can be efficiently implemented on graphics processors. The key idea is to convert the pointer operations into arrayed manipulations so as to unleash the computing power of GPUs. Experimental results prove that the proposed data structure outperform the best previous work by 6.8X and 9.6X in insertion and delete operations, respectively.

Key words: skiplist; GPU; parallel data structure; lock-free

1 引言

目前 GPU 通用计算亟待解决的问题是通过硬件体系结构、数据结构和算法的改进提升不规则程序的运算性能[1],而其中一个重要问题是如何改进数据结构及其操作例程的设计使之适合 GPU 的海量多线程执行方式.

跳表(skiplist)[2]是替代平衡二叉树的数据结构,核心思想是通过随机化算法在概率上保证数据访问时间的平衡. 跳表不需要数据以树形结构存储,并行访问时也不需要额外操作来保持平衡,相比平衡树更适合多线程执行. 因此,一些研究工作猜测跳表可以成为 GPU 并行计算的基本数据结构[3]. 本文基于 GPU 的高性能跳表数据结构,针对 GPU 特点在对跳表数据结构基础上,构造了支持高度并行操作的插入和删除算法.

2 跳表

跳表结构及其串行操作算法最早由 Pugh[2]提出. 如图 1 所示,跳表以数组或链表形式按序存放数据. 即每个节点存放一个数据作为键值(key). 节点具有随机高度,某一具体高度上的节点个数随高度的增加而指数衰减,即 $h+1$ 层的元素数量约为 h 层的 $1/p$. 每一节点配备指针指向同样高度上相邻的下一节点,这种连接实际上是加速搜索过程的快速通道.

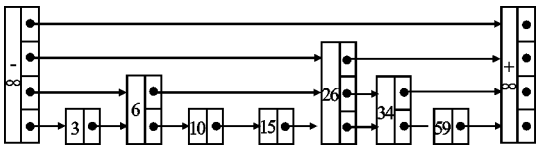

图 1 基本跳表结构

Pugh 也提出了基于锁的并行跳表处理算法[4].

收稿日期: 2014-01-23; 修回日期: 2014-03-18
基金项目: 国家自然科学基金项目"基于光线追踪机制的三维集成图形处理器体系结构研究"(61272085)

基于锁的跳表有两个主要缺点，首先，当获得锁的线程在关键区因某种原因出现错误，则其他线程不能再获取锁，可能导致整个多线程程序停滞[5]；其次，在可以正常运行的情况下，程序的可扩展性较差，海量线程的资源竞争会降低程序性能. 因此，一系列研究工作致力于改进跳表并行操作的性能[6-7].

3 基于 GPU 的跳表数据结构及相应算法

跳表的基本操作为插入和删除，操作结束后需保证跳表中元素的顺序没有被破坏. 由于每层新插入或删除的指针链接将破坏原有的链接，因此插入或删除过程中需要修改相关指针. 图 2 以插入操作为例说明[8].

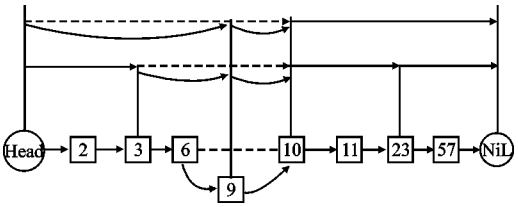

图 2 跳表操作及对指针的维护

图 2 中，当前表结构的最高高度为 3，新插入拥有键值 9 的元素. 任何结点的任意指针都可由结点键值和指针高度确定组成的二元组(key，level)表示，其中 key 为结点上的数据，level 为指针所在层数. 对于新插入的元素 9 来说，其各层前驱元素的信息分别为(6，1)、(3，2)、(head，3)，其中 head 是整个跳表的头(不存储有效数据)；同样各层后继元素的信息也可表示为(10，1)、(10，2)、(10，3). 图中虚线代表因被修改而无效的指针，弧线代表修改后形成的指针链接.

使用 GPU 进行计算时，大量线程可能同时修改跳表数据结构，此时需要解决两个主要问题.

(1)在任意层，当多个数据由多个线程同时插入时，由于线程之间的执行顺序无法确定，因此需要快速确定同层的链接对象.

(2)当多个元素同时插入表中时，可能在某层有连续两个新插入的结点，此时两节点的前驱相同，只需修改一次，简单的重复修改将导致效率的降低. 在删除操作时，重复的修改甚至会因为操作次序的不同而产生错误，所以不得不引入原子操作的比较交换(Compare & Swap)机制，同样会降低效率.

课题研究了新的方法解决以上问题，核心思路为将原本按指针索引方式定位链接对象的处理方式转化为对数组处理.

3.1 数据结构

跳表结构中结点的定义如图 3 所示，其中 key 为键值，forward 为向后连接的指针数组，value 为结点存储值，randomlevel 为此结点高度，图 3 中也给出了产生随机高度的方法，从其实现方式可知，跳表的高度满足概率为 p 的离散型几何分布. 随机高度的产生保证跳表概率意义下的平衡性. 图 3 中将用(key，level)二元组表示的节点指针定义为为 node_ref 类型.

```
struct node
{
  keytype key;      node** forward;
  valuetype value; int randomlevel;
}


函数 random_level()
begin
   level ← 1
   while(概率为P的随机事件发生)level← level+1
   return level
end
struct node_ref{keytype key，int level}
```

图 3 基本结构

插入和删除算法的基本操作结构为数组，主要涉及的数组参见表 1.

表 1 数组

数组名称	类型	说明
data[]	keytype	待插入或删除的数据
node_level[]	int	待插结点高度
prefix_sum[]	keytype	node_level 数组前缀和
mix_data[]	node_ref	各层待插数据指针信息
node_ptr[]	node*	各待插结点物理地址
index[]	int	存储 node_ptr[]数组的索引
pre[][]	node*	各层前驱指针
next[][]	node*	各层后继指针
pre_eq[]	node_ref	各层前驱指针二元组类型
next_eq[]	node_ref	各层后继指针二元组类型

pre[][]和 next[][]所存储指针同样可以用(key，level)二元组表示，为清晰表示算法，用两个类型为 node_ref 的虚拟的数组 pre_eq[]和 next_eq[]分别对应表示. 算法描述的对 pre_eq[]和 next_eq[]的操作可以通过 index[]数组很方便地转化为对 pre[][]和 next[][]物理指针数组的操作.

3.2 插入算法

本文提出的插入算法的核心思想是将无序的指针修改操作数组化，将插入结点的键值和其高度一同考虑，按照新插入元素的键值和当前层高度进行联合排序，根据排序后的数组信息进行链接，这样将

无序的指针寻址操作转化为对数组一次扫描. 实现高效并行的插入操作.

基本步骤如下：

E1. 搜索待插结点，记录路径信息；

E2. 对记录随机高度的数组，计算 prefix-sum[9] 值；

E3. 产生包含键值和层数关系的数组；

E4. 对键值和层数数组排序；

E5. 根据排序结果，链接各层指针.

图 4 中描述了插入操作算法流程. 首先，暂不考虑结点是否已存于表中，在搜索之前为每个待插入结点产生一个随机高度. 之后，搜索待插入结点的键值，记录其高度范围内各层的前驱和后继. 若结点已存在，则将 level 数组其对应高度设为 0，这样后续计算时即可忽略此结点. Data 数组中存有待插入的数据，共六项，其中 key 值 6 已存在于当前表中，所以在搜索后在对应的存储各点随机高度 level 数组中将其高度置为数值 0. 算法在搜索之前产生随机高度，在搜索过程中只需记录对应高度的信息，无需考虑搜索所经过的所有层的信息，从而减少了对全局内存的访问次数，有利于提高性能.

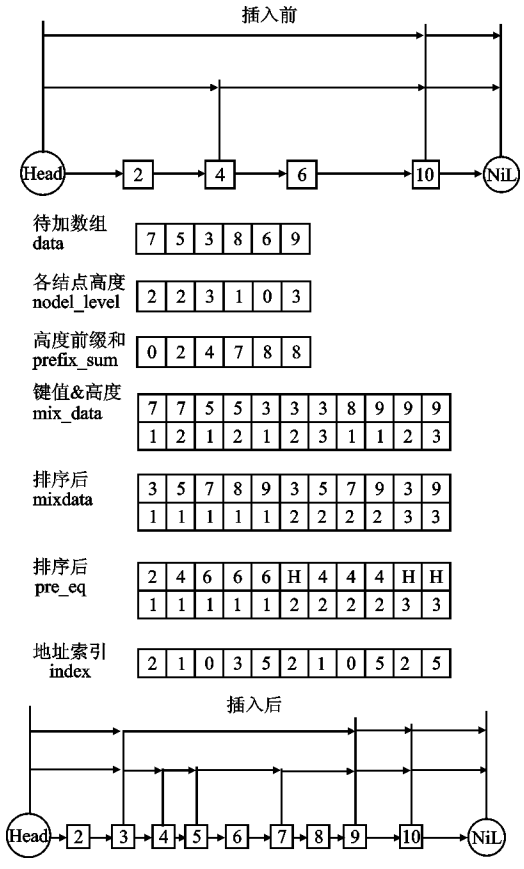

图 4　插入算法流程图

对 level 数组计算 prefix-sum，在 E2 步骤中，prefix_sum[]数组中存储 level[]数组的累加和. 在 E3 中根据 prefix_sum[]数组值定位各线程操作起点，将 data[]数组中键值和分解了的 level[]数组结点层数写入数组 mix_data[]. 由于跳表的每一层都需要维护有序的指针链接，各层之间独立，所以若待插结点高度为 h，就将其分解为 h 层，并分别同 key 组合，写入新数组 mix_data[]中，所以数组所存储元素包含键值和所在层数.

因为跳表结构在各层是有序的，在任一层对于结点、前驱、后继都有如下性质.

性质 1：若 k_1、k_2、k_3 表示待插入结点的键值，p_1、p_2、p_3 为对应前驱结点键值，n_1、n_2、n_3 为对应后继结点键值. 若满足 $k_1<k_2<k_3$，则有 $p_1\leqslant p_2\leqslant p_3$ $n_1\leqslant n_2\leqslant n_3$.

所以只要待插结点有序则对应前驱后继有序，又由于不同层彼此独立，为使同层 key 放在一起且有序，需对 mix_data[]数组排序，先比较层数，层数相同则比较 key 值. 在 E4 部分，将 mix_data[]数组同 index[]数组相关联，对 mix_data[]排序的过程中，同样也交换 index[]的相应元素. index[]数组是待插结点的地址索引，由于前驱、待插结点、后继的在有序性方面一致，index[]也可作为前驱、后继的索引，获得对应顺序. 为使算法更加明晰，在伪代码中认为排序后 pre_eq[]和 next_eq[]有序.

如前所述，在链接过程需要修改前驱和当前待插入结点的各层指针，虽然根据待插入结点的排序结果，便可知其前驱和后继结点的大致顺序，但无法确定是否相等，对有序的前驱数组做扫描判断便可链接各个指针. 如算法 1 中 E5 部分所示，若两点前驱相同则前一个待插结点指向后一个待插结点，否则前一个待插结点指向自己的后继，后一个带插结点的前驱指向后一个待插结点.

算法的中 E3、E4、E5 操作的数据量大小为 mix_data[]数组的大小. 假设待插入结点总数为 N，把 mix_data[]数组大小记做 S，可通过如下方式计算得到：

$$S=\sum_{1}^{N}h_i,E_{(S)}=\sum_{1}^{N}E(h_i)=N/p,$$

式中，h_i 是为第 i 个待插入结点的随机高度，其值服从概率为 p 的几何分布，期望为 $1/p$. 通常 $p=1/2$，所以 mix_data 数组大小在概率上为 $2N$. 虽然单个结点的指针最高可达 32 层，但高度平均值为 2，整体排序整体扫描链接的方法避免了 GPU 多线程实现

可能出现的负载不均衡问题.

3.3 删除算法

基本步骤：

E1. 搜索待删除结点，记录路径信息；

E2. 对记录随机高度的数组，求 prefix-sum 值；

E3. 产生包含键值和层数关系的数组；

E4. 对键值层数数组排序；

E5. 根据排序结果，对地址索引做预处理；

E6. 链接各层指针.

跳表的删除操作同插入操作类似. 但是，在 E1 搜索过程中所记录的信息除各层的前驱和后继外还包括结点的高度. 图 5 中 mix_data 已经过 E1～E4 步骤，排序完毕. 在删除操作中，只需修改前驱结点的指针，若待删除的几个结点相邻，则只需修改第一个待删除元素的前驱结点的指针. 当执行完 E1～E4 步骤，获得各层待删结点、前驱、后继的顺序. 在 E5 预处理部分，前驱 pre_eq[]数组中包含待删除元素，通过对 pre_eq[]数组和待删除的 mix_data[]数组扫描，满足 pre_eq[i+1]等于 mix_data[i]的前驱结点都因本身也是待删除结点不需修改，同样对应在 next_eq[]数组中的后继结点也因被删除而不会被任何前驱链接. 其对应关系图 6 所示，pre_eq[]和 next_eq[]数组中虚线表示的结点都将在本部中被去除. 在算法实现上可通过修改 index[]数组来实现. 去除多余信息的操作并行性很高，以本算法的实现为例，根据判断结果将 index[]数组中对应的索引值设为特定值 0xffffffff，然后调用相应的 thrust[9] 库函数 stable_partiton()便可高效实现.

```
Algorithm1 insert(data[],S)
Input:待插入数组data[],跳表S
output: 插入新元素后的跳表S
begin
 for i ∈[0,data.size()] do in parallel    /*E1*/
   searchkey(S,data[],node_level[],
             node_ptr[],pre,next[])
pre_fixsum(node_level[]) in parellel      /*E2*/
for i∈[0,data.size()] do in parallel      /*E3*/
  start ← prefix_sum[i]
  for j∈[0, nodelevel[i]] do
    mix_data[start+j] ← key,j
sort(mix_data[],index[]) in parallel      /*E4*/
for i∈[0,pre.size()] do                   /*E5*/
 if pre_eq[i]=pre_eq[i+1]
   mix_data[i]指向mix__data[i+1]
else
   mix_data[i]指向next[i]
   pre_eq[i+1]指向mix_data[i+1]
end
```

图 5 插入算法

对前驱数组和后继数组做过预处理，将冗余的待删除结点信息去除后，两个数组呈现一一对应的关系，可直接并行链接. 如图 7 算法 2 中 E6 部分所示.

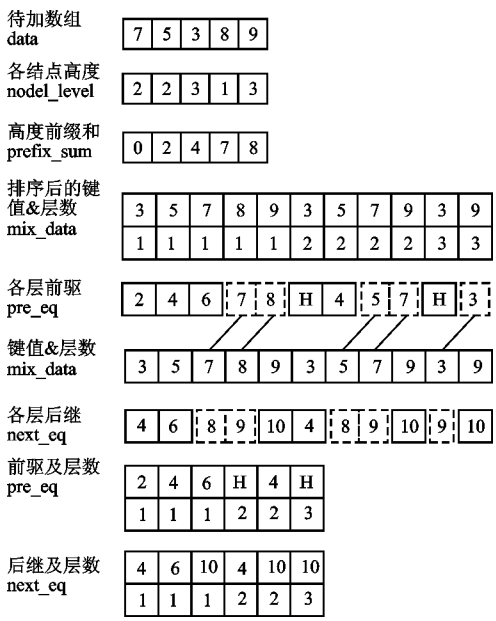

图 6 删除算法流程图

```
Algorithm2 delete(data[])
Input:待插入数组data[],跳表S
output: 插入新元素后的跳表S
begin
 for i ∈[0,data.size()] do in parallel      /*E1*/
   searchkey(S,data[],node_level[],node_ptr[],pre[][],next[][])
pre _fixsum(data[]) in parellel             /*E2*/
for i∈[0,data.size()] do in parallel        /*E3*/
  start ← prefix_sum[i]
  for j∈[0, nodeleve[i]] do
    mix_data[start+j] ← key,j
    index[start+i] ← i
sort(mix_data[],index[]) in parallel        /*E4*/
for i∈[0,mix_data.size()] in parallel       /*E5*/
 if (pre_eq[i].level=mix_data[i+1].level \
  &&pre_eq[i].key= =mix_data[i].key
 pre_eq[i+1] ← 0xffffffff,0xffffffff
 next_eq[i] ← 0xffffffff,0xffffffff
stable_partition(pre[][],next[][])
for i∈[0,pre_eq.size()] do in parallel      /*E6*/
  pre_eq[i]指向next[i]
end
```

图 7 删除算法

4 实验结果及性能分析

实验在 NVIDIA GeForce GTX 670 运算平台上进行，源代码由 CUDA 5.0 编译器编译. 现有文献中 Misra 和 Chaudhuri[7]（简为 MC）提出的跳表性能最好，因此本文使用该工作提供的源代码，在完全相同的软硬件环境下比较 MC 和新算法的 CUDA 实现.

实验比较了操作 32～250 000 个数据的性能，插入和删除的数据集服从均匀分布. 每个 block 内的最大线程数为 512，更大的数据通过增加 block 个数实现处理.

图 8 中比较插入数据的性能，在整个范围内采用新算法的跳表操作性能平均是 MC 的 143%，最

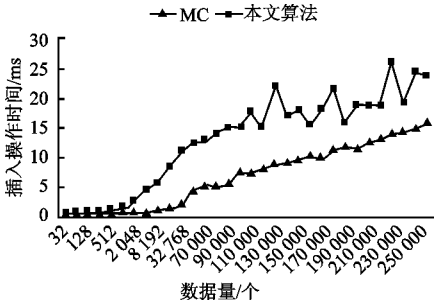

图 8　插入操作比较

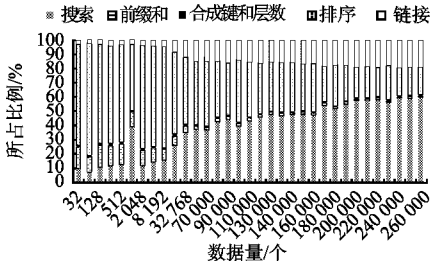

图 9　插入操作各步骤时间分配

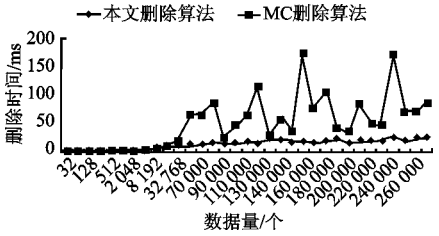

图 10　删除操作比较

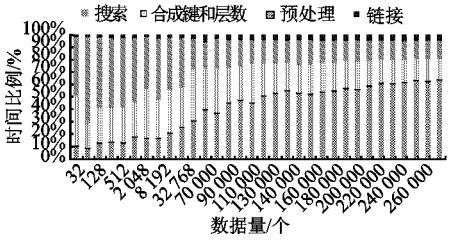

图 11　删除操作各步骤时间比例

高吞吐率能够达到 MC 的 6.82 倍，在插入 150 000 以上数据时吞吐率趋于饱和，但仍比 MC 高 62%.

图 9 中显示了在处理不同大小数据时，插入操作的各个步骤所占用时间的比例.

图 10 中的实验比较两种算法进行不同数据量的删除操作所用的时间. 实验显示，在较小数据量(32～8 192)情况下，MC 的算法性能略好，但当数据量增大后，MC 的性能表现极不稳定，整体呈现下降趋势. 相比 MC 算法的不稳定性，新的删除算法具有较好的扩展性，程序的性能同数据量大小和所用线程资源直接相关. 在 8 192～250 000 的数据量范围内，本文算法平均性能为 MC 的 3.8 倍，最高可达 9.6 倍.

图 11 显示了删除算法在整体数据范围内各步骤的时间比例，同插入算法相比，算法在最终链接前多出一个预处理的步骤.

5　结束语

本文提出并实现了适用于 GPU 的跳表数据结构和相应操作算法. 实验表明，新方法在跳表的插入和删除操作均比前人工作有大幅度提升.

参考文献：

[1] Deng Y, Mu S. A survey on GPU based electronic design automation computing[J]. Foundation and Trends in Electronics Design Automation, 2013(特邀单行本综述论文): 1-180.

[2] Pugh W. Skip lists: a probabilistic alternative to balanced trees[J]. Communications of the ACM, 1990, 33(6): 668-676.

[3] Lars Nyland, Stephen Jones. Understanding and using atomic memory operations[C]// GPU Technical Conference. San Jose, 2013.

[4] Pugh W. Concurrent maintenance of skip lists[D]. Marylard: University of Maryland, College Park: 1998.

[5] Fomitchev M, Ruppert E. Lock-free linked lists and skip lists[C]// Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing. Washington, DC, ACM, 2004: 50-59.

[6] Sundell H, Tsigas P. Scalable and lock-free concurrent dictionaries[C]// Proceedings of the 2004 ACM symposium on Applied computing. Washington, ACM, 2004: 1438-1445.

[7] Misra P, Chaudhuri M. Performance evaluation of concurrent Lock-free data structures on GPUs[C]// ICPADS. Singapore, 2012: 53-60.

[8] Harris M, Sengupta S, Owens J D. Parallel prefix sum (scan) with CUDA[J]. GPU gems, 2007, 39(3): 851-876.

[9] Bell N, Hoberock J. Thrust: A productivity-oriented library for CUDA[J]. GPU Computing Gems, 2011(8): 359-364.

作者简介：

李怀明　男，(1989-)，硕士. 研究方向为基于图形处理器的并行计算及并行计算结构. E-mail: dream_bingo@163.com.

邓仰东　男，(1973-)，副教授，博士生导师. 研究方向为并行计算和仿真、众核体系架构.