# 基于GPU的并行化Apriori算法的设计与实现

唐家维 王晓峰
(上海海事大学信息工程学院 上海 200431)

**摘 要** 大数据和高度并行的计算架构的时代已经来临,如何让传统的串行数据挖掘方法在当下获得更高的效率是一个值得探讨的问题。根据现代GPU大规模并行运算架构的特点(单结构多数据),对传统的串行Apriori算法进行并行化处理。使用最新的CUDA技术完成对传统串行Apriori算法中的支持度统计、候选集生成这两个计算的并行化实现,讨论了多种实现方法的差异,并提出改进方案。实验表明:改进后的并行算法使支持度统计在10000条事务的条件下效率提高16%,候选集生成在10000条事务的条件下效率提高25%。

**关键词** 数据挖掘,关联规则,频繁模式,并行算法
**中图法分类号** TP391 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2014.10.050

## Design and Implementation of Apriori on GPU

TANG Jia-wei WANG Xiao-feng
(Department of Information and Engineering, Shanghai Maritime University, Shanghai 200431, China)

**Abstract** Big data and parallel computation era have come, and it is a trend to convert serial data mine algorithm into parallel algorithm to take advantage of cheap machine. In this paper two main steps, namely support counting and candidate set generation in serial apriori algorithm, were rebuilt parallelly on CUDA architecture. Meanwhile the difference between various implements of parallel apriori was compared to find a better solution. Finally, the experiments indicate that the time of support counting and candidate set generation decreases 16% and 25% respectively on a data set containing 10000 items.

**Keywords** Data minint, Association rules, Frequent itemset mining, Parallel agorithm

## 1 引言

自从关联规则挖掘提出以来,频繁模式挖掘算法的研究受到了相当大的关注,关联规则挖掘在商业和学术领域都有广泛的应用。在如今海量数据库的背景下,如何提高频繁模式挖掘的效率是人们关注的核心问题之一。1994年,Rakesh Agrawal和Ramakrishnan Srikant两位博士提出了关联规则经典算法Apriori[1]。可能产生很大的候选集和重复扫描数据库是Apriori算法的两大瓶颈,针对这两个缺陷提出了Partition、Eclat、Fp-growth等算法,但是串行处理的架构限制了其对多核心处理器计算机资源的利用。在分布式计算和并行计算的基础上,Agrawal和Shafer提出了3种并行算法:计数分发(Count Distribution)算法、数据分发(Data Distribution)算法和候选分发(Candidate Distribute)算法[2]。根据这3种思想Ketan D. Shah等人在文献[3]中讨论了使用计数分发的思想提高多个CPU的使用效率,Ning Li在文献[4]中讨论了在多核CPU的计算机上使用MapReduce的思想将Apriori算法并行化,Shintani在文献[5]中讨论了在多节点集群环境中的Apriori算法。Qingmin Cui在文献[6]中从数据并行和任务并行两个角度讨论了并行的关联规则挖掘算法。

在多处理器和集群的环境下频繁项集挖掘的效率得到了很大的提高,但是仍存在很多困难和挑战。困难之一是缺乏成熟的并行算法开发架构,直接调用通用的系统库函数控制多个线程的并行执行在开发和调试上都是非常困难的。再者,搭建高性能集群环境成本过高,不适合个人或者小型企业。基于以上原因,本文针对CUDA架构对Apriori算法的计算过程和数据存储做了优化,提出使用CUDA并行技术统计支持度和生成候选集的并行化改进方案,来提高算法的效率。其中基于MapReduce思想的支持度统计方法比传统串行化方法提高了27倍(在100000条事务上测试),进一步对并行化支持度统计方法优化后避免了不必要的计算开销,使并行支持度统计算法的效率提高16%。在候选集生成算法中,通过GPU的多个运算核并行地对频繁集进行连接测试得到该算法的效率比串行算法提高了22倍(在100000条事务上测试),通过对并行算法进一步优化避免了启动无效的线程,使并行候选集生成算法的效率提高了25%。

## 2 CUDA技术计算Apriori算法的主要思路

### 2.1 使用CUDA技术计算支持度的关键问题

关联规则概述:给定一个交易集D,关联规则的挖掘问题

到稿日期:2013-11-04 返修日期:2014-02-15 本文受国家海洋公益性行业专项(201305026)资助。
唐家维(1989-),男,硕士生,主要研究领域为人工智能,E-mail:billl89@126.com;王晓峰(1958-),男,博士,教授,博士生导师,主要研究领域为人工智能。

就是产生支持度和可信度分别大于用户给定的最小支持度(minsupp)和最小可信度(minconf)的关联规则。关联规则的发掘分为两个步骤:(1)找出所有支持度大于最小支持度的频繁集;(2)从频繁集中产生期望的规则。其中第二步相对简单,所以人们在事务数据库中寻找大项集这一问题上做了大量研究。由 Rakesh Agrawal 和 Ramakrishnan Srikant 两位博士提出的 Apriori 算法是最经典的频繁模式挖掘算法,其核心思想是连接测试并利用频繁集的向下封闭性保证算法的正确性。

Apriori 算法的第一步是扫描事务数据库,从中统计出所有物品的支持度,通过删除支持度低于最小支持度(minsupp)的物品生成长度为 1 的频繁项集。Apriori 算法的第 $K$ 步分为两个阶段:第一个阶段通过连接两个长度为 $K-1$ 的频繁项集生成长度为 $K$ 的候选集;二个阶段通过扫描整个事务数据库统计出每个长度为 $K$ 的候选集的支持度,并删除支持度小于最小支持度的候选集生成长度为 $K$ 的频繁集,直到所生成的候选项集为空的时候 Apriori 算法终止。

重复扫描数据库和规模巨大的候选集是 Apriori 算法的两大瓶颈,如何在内存中表达数据集以减少扫描数据库次数,以及利用 CUDA 并行架构提高算法的效率是加速 Apriori 算法的关键。将 Apriori 算法并行化主要存在以下难点:1)在传统的事务集表达方式中(横向表达详见第 3 节)与一个候选集相关的事务分散在整个事务集中,统计每个候选集的支持度都需要扫描事务集的所有行,这种数据之间相互依赖的特性使得支持度统计难以并行化并且给系统数据总线带来压力;2)在串行的 Apriori 中有大量的循环操作,例如支持度统计、候选集生成,这些循环操作的循环次数可能达到上万次,由于 GPU 运算核频率低(相对于 CPU),因此它不适合做大规模的循环操作。

### 2.2 利用 CUDA 计算的主要思路

CUDA 并行计算架构:CUDA 是由 NVIDIA 推出的一种通用并行计算架构,该架构使得 GPU 能够解决复杂的计算问题。在硬件上,支持 CUDA 技术的 GPU 是一个包含多个流处理器单元的集合,每一个流处理器单元包含 8 流处理器。例如,在 NVIDIA Tesla C2075 显卡中有 448 个流处理器,NVIDIA Tesla K20 显卡中包含 2048 个流处理器。每一个流处理器单元中都有一块共享内存,用于流处理器模块中各个处理器之间的通信。除了共享内存,每个流处理器模块中还有一块常量内存和纹理内存用来加速特殊的应用。而各个流处理器单元通过全局内存通信。全局内存是对所有的流处理器单元共享的。在软件上,CUDA 并行架构是以线程为最小单位并行执行的。多个线程组成一个线程块,并行程序通过 CUDA 驱动程序随机地分发到流处理器模块上进行执行,并且多个线程块之间是并行执行的。CUDA 通过线程格的形式来管理线程块,例如一个二维的线程格可以通过 x 和 y 两个坐标来唯一确定线程格中的一个线程块。在 CUDA 并行运算架构中每个并行执行的线程都有相同的结构,每个线程通过其内部的 Id 号来区分彼此,并通过 Id 读取输入数据的不同部分。

针对 2.1 节中提出的多次遍历事务集的问题,我们通过重新组织事务集的格式来消除数据之间的依赖。在事务集的纵向表达方式中,与候选集相关的所有事物被保存在事务集的同一行中,在统计一个候选集的支持度时只涉及到整个事务集的一行。但是事务集的纵向表达大大增加了事务集的列数(列数=事物的数量),在事物数量相当大的条件下用循环的方式统计支持度是非常耗时的。改进的方法是将统计支持度的操作分摊到多个 CUDA 核上并行执行,最后将所有的结果并行求和(详见第 4 节)。并行 Apriori 算法流程如图 1 所示。

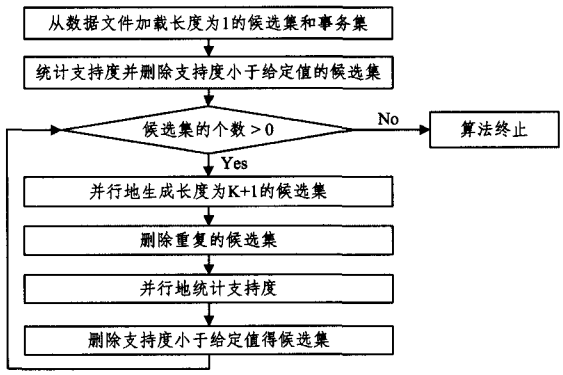


图 1 Apriori 算法流程

## 3 数据集

在 Apriori 算法中有两个关键的数据集:候选集、事务集。候选集和事务集都是一张二维二进制表,其中候选集中的每一行表示一个物品(item)的集合,事务集的每一行表示一个事物所包含的所有物品。

在算法的设计中数据集的格式影响着算法的时间和空间复杂度。合理的数据集格式既要保证数据集的体积在可接受的范围内,又要保证数据集可以被高效地检索。

### 3.1 候选集的格式

候选集是多个物品集的集合,它的每一行记录了一个物品的集合和这个集合在事务集中的支持度。假设有 ABCD 4 种物品和 AB、AC、AD、BC 4 个候选集,则数据集的表达如表 1 所列。

表 1 候选集格式

| | 物品 A | 物品 B | 物品 C | 物品 D | 支持度 |
|---|---|---|---|---|---|
| 候选集 AB | 1 | 1 | 0 | 0 | 0 |
| 候选集 AC | 1 | 0 | 1 | 0 | 1 |
| 候选集 AD | 1 | 0 | 0 | 1 | 2 |
| 候选集 BC | 0 | 1 | 1 | 0 | 3 |

物品集的第一行为 11000,表示第一个候选集包含了第一种和第二种物品且它的支持度为 0。

由于本次实验所使用的 GPU 卡不支持 64 位的数据类型,本文将数据以 32 位无符号整数的形式存储。数据类型的长度会影响并行算法划分任务的粒度,过长的数据类型会使得划分的粒度过粗,导致多核 GPU 的负载不均衡;过短的数据类型(如 4 位 int 类型)会使得每一个任务过小,导致大量的时间用于 GPU 线程的启动和关闭。

在机器内存中物品集存储在一个长度为 8 的一维的无符号整型(32 位 int 型)数组中。如果物品的种类不是 32 的倍数,则以 0 填充,如表 2 所列。

表 2　候选集格式

| | 物品 | 支持度 |
|---|---|---|
| 候选集 AB | 1100 填充 28 个 0 | 0000…0000 共 32 位 |
| 候选集 AC | 1010 填充 28 个 0 | 0000…0001 共 32 位 |
| 候选集 AD | 1001 填充 28 个 0 | 0000…0010 共 32 位 |
| 候选集 BC | 0110 填充 28 个 0 | 0000…0011 共 32 位 |

### 3.2　事务集的格式

事务集是多个事务的列表，其中每一个事务都是物品的集合，例如客户一次购买的所有物品。事务集的格式常用的有两种：纵向表示和横向表示。

#### 3.2.1　事务集的横向表达

如果事务集采用横向表示，那么事务集的每一行表示一条事务所包含的物品。若有 $D_1$、$D_2$、$D_3$、$D_4$ 4 个事务，那么它的数据表达如表 3 所列，事务集的第一行 1100 表示事务 D1 中包含物品 A 和 B。

表 3　事务集横向格式

| | 物品 A | 物品 B | 物品 C | 物品 D |
|---|---|---|---|---|
| 事务 D1 | 1 | 1 | 0 | 0 |
| 事务 D2 | 1 | 0 | 0 | 0 |
| 事务 D3 | 0 | 1 | 0 | 0 |
| 事务 D4 | 0 | 0 | 1 | 0 |

一个经典的基于 CPU 的 Apriori 算法通常采用横向数据表达，但是在统计每一个候选集的支持度时需要遍历整个事务集。在一个基于 GPU 的算法中多次对整个事务集进行扫描将大大增加数据通道的通信量，这降低了 GPU 的利用率。基于以上原因，在基于 GPU 的 Apriori 算法中采用纵向数据表达。

#### 3.2.2　事务集的纵向表达

如果事务集采用纵向表示，那么事务集的每一行都是一个事务的集合且事务集中的每一行都和候选集一一对应。例如有 AB、AC、AD、BC 4 个候选集，以及 $D_1$、$D_2$、$D_3$、$D_4$ 4 个事务，那么事务集的纵向表达如表 4 所列，事务集的第一行 1100 表示事务 D1、D2 包含了候选集 AB。

表 4　事务集纵向格式

| | 事务 D1 | 事务 D2 | 事务 D3 | 事务 D4 |
|---|---|---|---|---|
| 候选集 AB | 1 | 1 | 0 | 0 |
| 候选集 AC | 1 | 0 | 1 | 0 |
| 候选集 AD | 0 | 0 | 0 | 1 |
| 候选集 BC | 0 | 0 | 1 | 0 |

在机器内存中该事务集存储在一个长度为 4 的一维的无符号整型数组中，如果事务的总量不是 32 的倍数，则以 0 填充，如表 5 所列。

表 5　候选集纵向表达

| | |
|---|---|
| 候选集 A | 1100 填充 28 个 0 |
| 候选集 B | 1010 填充 28 个 0 |
| 候选集 C | 0001 填充 28 个 0 |
| 候选集 D | 0010 填充 28 个 0 |

在纵向数据表达中，统计候选集的支持度仅涉及到事务集的一行，每个候选集的支持度的统计与其他的候选集独立，这为算法的并行执行提供了可能。同理，在生成长度为 $K+1$ 的候选集时也仅涉及事务集中的两行，这提高了并行执行的速度，详见第 5 节。

## 4　统计支持度

支持度的统计是指统计候选集在事务集上的支持度，具体来说一个候选集的支持度为这个候选集中的所有物品被事务集包含的次数。在基于 CPU 的 Apriori 算法中统计候选集的支持度需要扫描整个事务集，它是整个算法中耗时最长的步骤。假设有 $m$ 个事务以及 $n$ 个物品，在基于 CPU 的算法中每一个候选集做一次支持度统计总共要查询 $m*n$ 次。而在基于 GPU 的算法中，由于事务集中的每一行相对其他行都是独立的，每一个候选集做一次支持度统计只要查询 $m$ 次，即事务集中的一行。

在基于 GPU 的支持度统计中，每一个候选集的统计工作由一个线程块负责，其中线程块的每一线程负责统计一个 32 位无符号整数(以二进制表示)中 1 的个数，统计的结果存储在设备共享内存中，最后并行地对它们求和。假设事务集中共有 4 * 32 个事务，支持数统计的流程如图 2 所示。如果有更多的事务，例如有 $n*32$ 个事务，那么事务集的每一行用 $n$ 个无符号整数保存，每一个线程块启动 $n$ 个线程。

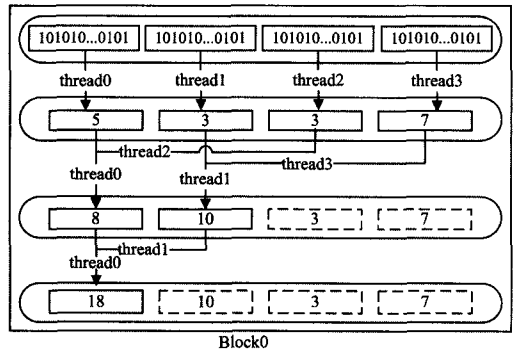


图 2　支持度统计流程

图 2 所示的支持度的统计分为两步：第一步是线程 0－3 统计出每一个无符号数中 1 的个数；第二步是用 reduce 的方法对中间结果求和。下面详细展开这两个步骤。

### 4.1　统计 32 位二进制串中 1 的个数

Fang Wenbin 等人曾在文献 Frequent Itemset Mining on Graphics Processors 中提到，可以在设备的常量内存中存储一个映射，从一个 32 位无符号整数映射到其中 1 的个数，这样通过查表的方式就能确定一个整数中 1 的个数。例如把 00000001 11000000 00000000 00000000 映射为 3。这种方法的常量内存开销为 $2*2^{32}*4$ byte，即 32GB 的常量内存，这是硬件难以满足的，并且需要二叉查找才能降低算法的复杂度。基于以上原因，本文不采用查表的方式而采用求余数的方法确定 1 的个数。具体如算法 1 所示。

**算法 1**　利用余数确定二进制串中 1 的个数

```
int count ← 0 // count 存放 1 的个数,初始化为 0
while (paramInt ≠ 0)
do
{
  If paramInt % 2=1
    count ← count+1
  end if
  paramInt >>=1 // paramInt 右移一位,相当于除以 2
```

}
Return count

求余法将一个 32 位无符号整数进行 32 次右移，每次右移后对 2 求余数，如果余数为一则计数器增加 1。相比查表法而言，右移求余法没有使用到常量内存，节约了内存开销，且时间复杂度上与查表法相同(对于一个长度为 $n$ 的整数，复杂度为 $O(n)$)。

### 4.2 Reduce 求和

在单 CPU 系统中多个数的求和只能串行执行，例如 16 个数求和需要执行 15 次加法。而 GPU 中有上百个运算核，利用分而治之的思想可以用多个运算核并行实现求和算法。具体方法如图 3 所示。

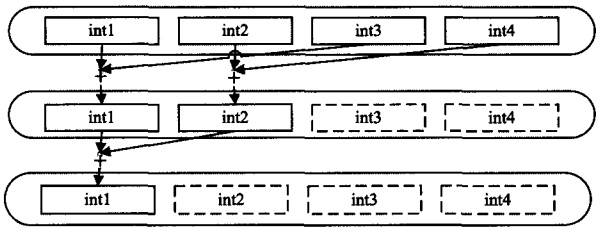


图 3 MapReduce 求和

例如对 4 个数求和，第一次并行执行 int1＝int1＋int3 和 int2＝int2＋int4，第二次执行 int1＝int1＋int2。4 个数的和保存在 int1 中。

相对于时间复杂度为 $O(n)$的串行求和算法，并行求和算法的时间复杂度为 $O(\lg n)$，时间复杂度大大降低。但是 Reduce 求和算法要求输入的整数个数必须为 2 的幂，如果不满足要求就用 0 补足。这不仅浪费了宝贵的设备共享内存，而且增加了算法执行的时间。

如果 Reduce 算法不需要满足输入个数为 2 的幂的要求，算法的执行时间将大大缩短，在输入个数非常大的条件下最多可以提高 50%的效率。解决该问题的关键是加入一个标志来确定当前迭代步中的输入个数是奇数还是偶数。假设有 5 个数相加，且标志位 Flag＝0 表示输入个数为偶数；flag＝1 表示输入个数为奇数，改进后的 reduce 算法执行过程如图 4 所示。

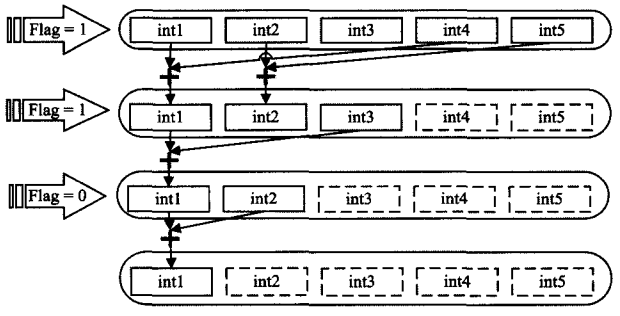


图 4 优化后的 MapReduce 求和

第一轮迭代是参与运算的整数为 5 个，所以 Flag＝false，执行 int1＝int1＋int4 和 int2＝int2＋int5。在第一轮运算中 int3 没有参与计算，int3 将在下轮计算中被累加的 int1 中。

Flag 的初始化以及经过改进的 Reduce 算法详细过程如算法 2 所示。假设需要求和的数组存放在名为 cache[]的共享内存中，flag 为标志位。

**算法 2** 加入 Flag 的 Reduce 求和算法

```
Algorithm 2  reduceKernal ()
int Flag ← 线程数量 % 2
int j ← 线程数量 / 2
while j ≠ 0
  {
  if (Flag=0)
    if (线程号 < j)
      cache[线程号]=cache[线程号]+cache[线程号+ j]
end if
    线程同步
    Flag ← j % 2
    j ← j / 2
  else if (Flag=1)
    if (线程号< j)
      cache[线程号]=cache[线程号]+cache[线程号+j+1]
    线程同步
    j ← j+1
    Flag ← j % 2
    j ← j / 2
  end if
  }
  if (线程号 =0)
    retrun cache[0]
end if
```

### 4.3 时间复杂度分析

假设候选集个数为 $M$，事务集个数为 $N$。使用 CPU 进行串行计算的时间复杂度为 $O(M*N)$。如果使用 GPU 进行并行计算，假设 GPU 含有 $K$ 个流处理器，那么可以得到总的复杂度为 $O(M/K * \log(N))$。

## 5 候选集的生成

在 Apriori 算法中长度为 $K+1$ 阶的候选集由长度为 $K$ 阶的平凡项集连接生成。例如 $I_1$ 和 $I_2$ 是长度为 $K$ 的平凡项集，将 $I_1$ 和 $I_2$ 做并操作生成 $I_3$(即 $I_3=I_1 \cup I_2$)，如果 $I_3$ 中物品的数量为 $K+1$，则 $I_3$ 属于长度为 $K+1$ 阶的候选集。

在基于 GPU 的 Apriori 算法中，所有的数据被存储在物品集和事务集中，这两个集合在逻辑上是两个由二进制位组成的二维表，并且它们的行数相等，两个表中每一行是一一对应的。例如候选集中的第一行表示第一个项集所包含的物品，事务集的第一行表示第一个项集被那些事务包含。现在假设总共有 A、B、C、D 4 种物品，一个长度为 1 的候选集共有 A、B、C、D 4 个项集，该候选集所对应的事务集共有 $D_1$、$D_2$、$D_3$、$D_4$ 4 个事务，则候选集和事务集的表示如表 6 所列。

表 6 物品集(上)、事务集(下)

| | 物品 A | 物品 B | 物品 C | 物品 D |
|---|---|---|---|---|
| 项集 A | 1 | 0 | 0 | 0 |
| 项集 B | 0 | 1 | 0 | 0 |
| 项集 C | 0 | 0 | 1 | 0 |
| 项集 D | 0 | 0 | 0 | 1 |

| | 事务 D1 | 事务 D2 | 事务 D3 | 事务 D4 |
|---|---|---|---|---|
| 项集 A | 1 | 1 | 0 | 0 |
| 项集 B | 1 | 0 | 1 | 0 |
| 项集 C | 0 | 0 | 0 | 1 |
| 项集 D | 0 | 0 | 1 | 0 |

在 Apriori 算法中一个长度为 $K$ 的候选集是由长度为 $K-1$ 的候选集做连接操作生成的，在 GPU 上实现时分为两步。第一步将长度为 $K-1$ 的物品集中的两个项集做按位或操作生成一个长度为 $K$ 的项集。第二步将长度为 $K-1$ 的事务集中对应的两个项集做按位与操作生成一个长度为 $K$ 的项集。例如有项集 A:{1000} B:{0100}做按位或生成项集 AB:{1100}，以及所对应的事务{1000}。

在 GPU 上实现时每一个候选集的生成由一个线程块完成，线程块中每一个线程完成 32 个位的位运算，即两个 32 位无符号整数的按位与和按位或。CUDA 核函数的调用需要一个二维的线程块，维数都为长度为 $K-1$ 的候选集的项集数。生成候选集的算法如算法 3 所示。

**算法 3** 候选集生成

```
Algorithm 3  generateItemsetKernal ()
If 线程块号.x >=线程块号.y
  Return
int  indexX ← 线程块号.x * 候选集列数 * 线程号
int  indexY ← 线程块号.y * 候选集列数 * 线程号
// 将二维的线程块号转换为一维
int  indexOutput ← (线程块号.x * (线程块号.x-1) / 2 +线程块号.y) * 候选集列数+线程号
长为 K 的候选集[indexOutput] ← 长度为 K-1 的频繁集[indexX] | 长度为 K-1 的频繁集[indexY]
```

例如 $K-1$ 的候选集如表 6 所列，其需要 4 * 4 的线程块，每一个线程块包含一个线程。如果 blockIdx.x > blockIdx.y，则该线程块中的线程执行第 x 个项集和第 y 个项集的位运算；如果 blockIdx.x <= blockIdx.y，则该线程块中的所有线程不做任何操作直接返回。候选集生成的执行过程如图 5 所示。

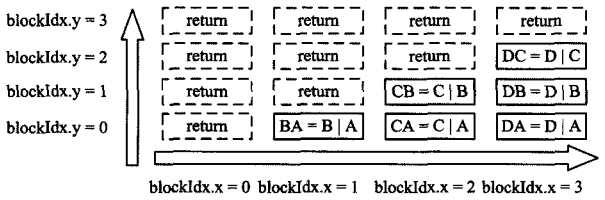


图 5 由长度为 1 的候选集生成长度为 2 的候选集

可以看到核函数一共启动了 16 个线程块，其中只有 6 个真正进行计算而剩下的 10 个线程块中的线程直接返回，不难从图 5 中发现，有超过 50%的线程块没有真正执行计算。然而无论是线程还是线程块的启动都是需要消耗时间和资源的，这就造成了设备利用率的下降。下面从这个角度去提升算法的效率。

提高生成候选集的效率的关键是让每一个线程块都参与实际计算，可以通过坐标转换的方法去除不必要的线程块。具体分两种情况讨论。第一种：长度为 $K-1$ 的项集总数为偶数。如果项集总数为偶数那么启动的线程块的 $x$ 轴维数＝项集数－1，$y$ 轴维数＝项集数/2。线程块的启动和坐标转换过程图 6 所示。

第一步：计算线程块的维数，如例子中所示长度为 $K-1$ 的项集总数为 4，在优化后的候选集生成算法中一共生成 6 个线程块，其中 $x$ 轴维数＝4－1＝3，$y$ 轴维数＝4/2＝2。第二步：坐标映射，如果 $x>y$，那么 $x=4-x-2$，$y=4-y-1$。第三步：坐标平移，将所有线程中的 $x$ 加 1，$y$ 保持不变。

可以从图 6 中发现，通过对 blockIdx.x 和 blockIdx.y 的两次变换后 $x$ 和 $y$ 的值和优化之前图 5 中的 blockIdx.x 和 blockIdx.y 的值相同，但是优化后线程块的数量从 16 个下降为 6 个，去除了不必要的线程块。通过以上算法生成的候选集可能包含重复的项集和长度不为 $K$ 的项集，在算法结束后要删除这两种项集。

第二种情况：对于长度为 $K-1$ 的项集总数为奇数的情况也有类似的处理。

时间复杂度分析：假设候选集个数为 $M$，事务集个数为 $N$。使用 CPU 进行串行计算的时间复杂度为 $O(0.5NM^2-0.5NM)$。如果使用 GPU 进行并行计算，假设 GPU 含有 $K$ 个流处理器，每个流处理器有 $L$ 个处理单元，那么可以得到总的复杂度为 $O((0.5NM^2-0.5NM)/(KL))$。

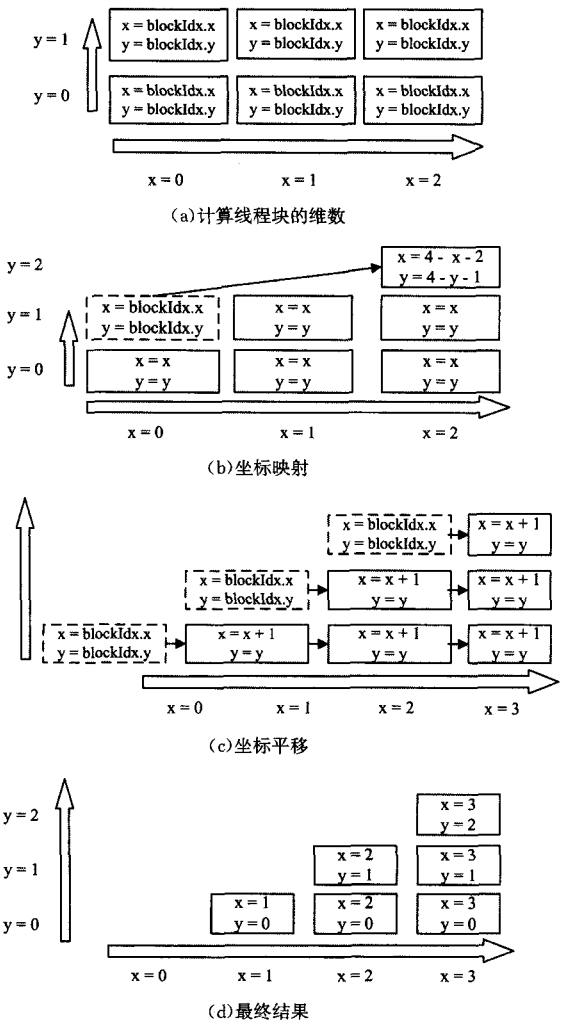


图 6

## 6 实验结果

在本节中我们将通过 IBM 的 T40I10D100K 数据集对上述算法的 CPU 和 GPU 版本的性能进行对比。其中 CPU 和 GPU 版本的算法所使用的数据类型相同，唯一的不同是 CPU 版本的 Apriori 算法用循环语句模拟了 GPU 上的并行算法并且 CPU 不需要将数据集在主机内存和设备内存间拷贝。

为了简洁，我们对统计支持度和生成候选集分开测试，并对比了同一个实现在不同数据量下的区别。

### 6.1 实验环境

操作系统：Windows 7 旗舰版 32 位

主机内存：6GB DDR3 (1333MHz)

CPU 型号：英特尔 Xeon(至强) W3530 4 核

CPU 主频：2800MHz 最大 Turbo 频率：3060MHz

CPU 缓存：8MB

GPU 型号：Nvidia Tesla C2075

GPU 显存：5375 MB

CUDA 核数量：448 个

CUDA 核频率：1.15GHz

显存频率：1.5GHz

显存带宽：144GB/Sec

CUDA 并行运算性能与以上列表中最后 4 个指标有关，指标越高，性能越强。

### 6.2 支持度统计实验结果

在统计候选集的支持度时最关键的步骤是用 Reduce 的思想对多个无符号整数求和。在 6.2 节中我们对基本的 Reduce 算法进行了优化，使得需要启动的线程数不需要正好等于 2 的幂。在图 7 中我们对比了支持度统计算法的 3 种实现在不同数据量下的运行时间(单位毫秒)。

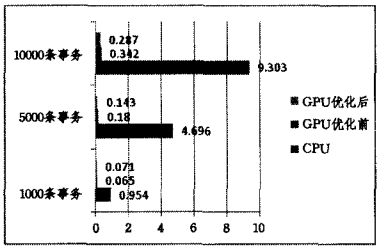


图 7 支持度统计实验结果

值得注意的是在 1000 条事务的条件下优化后的 Reduce 算法比基本 Reduce 算法慢，当数据量增加到 5000 和 10000 条时优化后的算法才体现出优势，分别比基本 Reduce 算法快 20.1%和 16.1%。

### 6.3 候选集生成实验结果

对候选集生成算法优化的关键在于启动最少数量的线程块，从而避免了启动无用的线程造成资源的浪费。在图 8 中我们对比了候选集生成算法的 3 种实现在不同数据量下的运行时间(单位毫秒)。

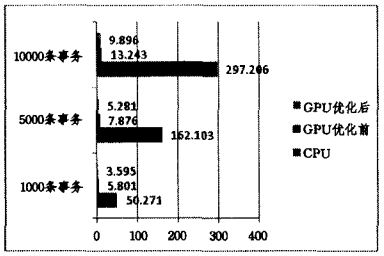


图 8 候选集生成实验结果

如图 8 所示，优化后的候选集生成算法在 3 种数据量的情况下分别提升了 38.0%、32%、25.3%。值得注意的是当数据量从 1000 增加到 5000 时 GPU 算法的运行时间只增加了大约 50%，同时 CPU 算法的运行时间增加了 100%，这是由于当数据量为 1000 时每个线程块启动 33 个线程相对于数据量为 5000 时每个线程块启动 167 个线程流处理器的利用偏低所造成的。

**结束语** 利用数据挖掘算法的并行性，在多个运算核上并行执行任务可以大大提高算法的效率，但是算法的并行方式各异，选择最合适的方式才能使资源得到合理的利用。本文对 Apriori 算法的并行化着眼于支持度统计和候选集生成两个步骤。通过纵向的事务集存储方式，避免了在统计候选集的支持度时反复扫描整个事务集，同时使得每一个候选集支持度的统计在数据上是相互独立的，这为并行计算提供了条件。在统计支持度的过程中改进的 reduce 求和算法，使得在输入数据数量不等于 2 的幂的情况下无需对数据进行填充从而降低了运算量。最后通过实验对比了在不同数据量下算法性能的改变，实验证明，GPU 上的支持度统计算法和 CPU 相比较提高了 32 倍，候选集生成算法提高了 30 倍。

## 参 考 文 献


[1] Agrawal R, Srikant R. Fast algorithms for mining association rules[C]// Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94). 1994:487-499

[2] Agrawal R, Shafer J C. Parallel mining of association rules [J]. IEEE Transactions on Knowledge and Data Engineering, 1996,8(6):962-969

[3] Shah K D, Mahajan S. Maximizing the Efficiency of Parallel Apriori Algorithm[C]// International Conference on Advances in Recent Technologies in Communication and Computing. IEEE,2009:107-109

[4] Li Ning, Zeng Li, He Qing, et al. Parallel Implementation of Apriori Algorithm Based on MapReduce[C]// Software Engineering, Artificial Intelligence, Networking and Parallel & Distributed Computing (SNPD). 2012:236-241

[5] Shintani T, Kitsuregawa M. Hash based parallel algorithms for mining association rules[C]// Fourth International Comperence on Parallel and Distributed Information Systems. IEEE, 1996: 19-30

[6] Cui Qing-min, Guo Xiao - bo. Research on Parallel Association Rules Mining on GPU[C]// Proceedings of the 2nd International Conference on Green Communications and Networks. 2013:215-222

[7] Yang Yuan-sen, Yang Chung-ming, Hsieh T J. GPU parallelization of an object-oriented nonlinear dynamic structural analysis platform[J]. Simulation Modelling Practice and Theory, 2014, 40:112-121

[8] Shan Feng, Hart John C. Parallel computing on geostatistical data using CUDA[C]// IDEALS. 2014

[9] Smirnov V. Parallel Integration Using OpenMP and GPU to Solve Engineering Problems[J]. Applied Mechanics and Materials, 2014, 475:1190-1194