基于GPU异构计算的关联规则挖掘算法及增量式情况的研究和简单实现

本文将从以下方面进行阐述：

1. 关联规则问题描述
2. 关于关联规则挖掘算法Apriori的说明
3. 关于FUP增量式关联规则挖掘算法的说明
4. 关于在使用CUDA进行关联规则挖掘算法前数据的预处理
5. 使用字典树来产生候选项
6. 候选项集支持度计数的计算方法
7. 使用GPU计算支持度
8. 关于优化的一些简单的说明
9. 将上面谈到的计算支持度计数的思想运用到增量式的关联规则中
10. 性能比较
11. 缺陷总结
12. 关联规则问题描述

购物篮分析是频繁项集挖掘的一个典型例子。该过程通过发现顾客放入他们“购物篮”中的商品之间的关联，分析顾客的购物习惯，这种关联的发现可以帮助零售商了解哪些商品频繁地被顾客同时购买，从而帮助他们指定更好的销售策略。

假设记录“购物篮”信息的数据库称为交易数据库，其中每一条记录代表一个交易，包括唯一的交易标识（TID）和一个数据项的集合，如表1-1中所示。其中商品ID列表中，每一列中的一个数字代表一个单品唯一的ID值，如假设ID值11可以代表牛奶，ID值13可以代表面包。

表1-1 销售事务数据库片段

|  |  |
| --- | --- |
| Trans\_ID（TID） | 商品ID的列表 |
| T100 | 11,13,18,116,… |
| T200 | 12,18,50,… |
| … | … |

如果我们想象全域是商品中商品的集合，则每种商品有一个布尔变量，表示该商品是否出现，则每个购物篮可以用一个布尔向量来表示。通过分析布尔向量，得到反映商品频繁关联或同时购买的模式。这种模式可以用关联规则的形式表示。量化这种规则的标准需要引入支持度和置信度这两个度量。它们分别反映所发现规则的有用性和确定性。

关联规则的挖掘问题的形式化描述如下：设是项的集合。任务相关的数据*D*是数据库事务的集合，其中每个事务是一个非空项集，使得。每个事物有一个标识符，称作*TID*。项的集合称为项集（itemset）。设*A*是一个项集，事务*T*包含*A*当且仅当。如果项集A中包括*k*个项，则称为*k-*项集。关联规则是形如的蕴含式，其中，，，，并且。规则在事务集*D*中成立并且具有支持度*s*，*s*是*D*中包含（即集合*A*和*B*的并或*A*和*B*二者）事务占全部事务的百分比，它的概率表示为*P()*。如果*D*中包含*A*的事务同时也包含*B*的事务的百分比*c*，则规则在事务集*D*中具有的置信度*c*，它是条件概率*P(B|A)*。



*support() = P()* (1-1)



*confidence() = P(B | A)* (1-2)



支持度和置信度两个阈值是描述关联规则的两个重要概念，支持度反映的是关联规则在数据库中的重要性，表示规则出现的频度；置信度衡量关联规则的可信程度，表示规则的强度。

项集的出现频度是包含项集的事务数，简称为项集的频度、支持度计数或技数。（1-1）式定义的项集支持度有时称为相对支持度，而出现频度称为绝对支持度。同时满足最小支持度阈值（min\_sup）和最小置信度阈值的规则称强规则（min\_conf）。最小支持度表示项集在统计意义上的最低重要性，最小置信度表示规则的最低可靠性。

如果项集的出现频度大于或者等于最小支持度阈值min\_sup与D中事务总的乘积，则成项集满足最小支持度。如果一个项集A满足最小支持度（A的支持度大于等于min\_sup），则称它为频繁项集。频繁k-项集的集合通常记作Lk。

由（1-2）式，有

*confidence() = P(B | A) = =*  (1-3)



（1-3）式表明规则A=>B的置信度容易从A和A∪B的支持度计数推出。也就是说，一旦得到A，B和A∪B的支持度计数，则可导出对应的关联规则A=>B和B=>A，并检查它们是否是强规则。因此挖掘关联规则的问题可以归结为挖掘频繁项集。

关联规则的挖掘是一个两步的过程：第一步利用候选项集找出所有的频繁项集，第二步由频繁项集产生强关联规则。步骤二操作开销远低于第一步，因此关联挖掘算法规则的总体性能由第一步决定。

1. 关于关联规则挖掘算法Apriori的说明：

Apriori算法是Agrawal和R.Srikant于1994年提出的，为布尔关联规则挖掘频繁项集的原创性算法。算法使用频繁项集性质的先验知识，所以称为Apriori算法。Apriori算法在数据挖掘和机器学习领域被广泛的应用。算法大致思想如下：首先扫描数据库，累计每个项的计数，并收集满足最小支持度的项，找出频繁1-项集的集合，该集合记作L1。之后使用L1找出频繁2-项集的集合L2，使用L2找出L3，依次类推，直到不能再找出频繁k-项集。找出每一个Lk需要一次数据库的完整扫描。

利用先验性质（频繁项集的所有非空子集也一定是频繁的。即如果一个集合不能通过最小支持度阈值测试，则它的所有超集也不能通过最小支持度阈值测试）来提高频繁项集逐层产生的效率。通过Lk-1找出Lk，其中主要包括连接步和剪枝步两个步骤。

连接步：通过将Lk-1与自身连接产生候选k-项集的集合Ck。设*l1*和*l2*是项集*Lk-1*中的项集*，li*[*j*]表示*li*的第*j*项。Apriori算法中假定事务或者项集中的项按字典序排序。对于（k-1）项集*li*，这意味着*li*[1]<*li*[2]<…<*li*[k-1]。执行连接操作，其中*Lk-1*的元素是可连接的，如果他们前（k-2）个项相同。即，*Lk-1*的元素*l1*和*l2*是可连接的，如果（*l1*[1]= *l2*[1]）∧（*l1*[2]= *l2*[2]）∧…∧（*l1*[k-2]= *l2*[k-2]）∧（*l1*[k-1]< *l2*[k-1]）。*l1*和*l*2连接的结果项集是{*l1*[1]，*l1*[2]，…，*l1*[k-2]，*l1*[k-1]，*l2*[k-1]}。

剪枝步：Ck是Lk的超集，即它的成员可以是频繁的也可能是不频繁的，但所有的频繁k-项集都包含在Ck中。通过先验性质来压缩Ck。因为任何非频繁的（k-1）项集都不是频繁k项集的子集。所以，如果一个候选k-项集的（k-1）项子集不在Lk-1中，则该候选项不可能是频繁的，从而可以将其从Ck中删除。

Apriori算法是一种广度优先算法，需要对事务数据库进行多次扫描来发现所有的频繁项集，并且一次扫描只能解决同一长度为k的所有项集。

1. 关于FUP增量式关联规则挖掘算法的说明

Apriori算法是在事务数据库中数据不变的前提下进行的，当数据增加时，若使用Apriori算法则会导致上一次生成的频繁项集不再有用，而重新执行Apriori算法的时间开销太大，增量式关联规则挖掘算法用于这种情况的数据挖掘。FUP算法的基本思想很简单，主要描述如下：

设原事务数据库的数据集为D，新增的数据集为d，则变化后的事务数据库为（D+d）。设L(D)为使用Apriori算法对数据集D进行挖掘得到的全部频繁项集。

1) 利用Apriori算法生成新事物数据集d的频繁项集L(d)，对L(D)和L(d)进行比较，找出公共相同的部分。则相同部分的频繁项集一定属于更新后的事务集（D+d）的频繁项集。

2) 设某一项集为t，若t属于L(d)，但不属于L(D)，则扫描D得到t在D中的支持度计数SupD，再根据d中已经得出的t的支持度Supd，求出t在（D+d）中的支持度SupD+d，如果SupD+d≥min\_sup，则将t放入到变化后的数据集（D+d）中，否则t将不是频繁项集。

3) 若t 属于L(D)，但不属于L(d)，则扫描d得到t在d中的支持度计数Supd，再根据D中已经得出的t的支持度SupD，求出t在（D+d）中的支持度SupD+d，如果SupD+d≥min\_sup，则将t放入到变化后的数据集（D+d）中，否则t将不是频繁项集。

1. 关于在使用CUDA进行关联规则挖掘算法前数据的预处理

如表1-1中所示的事务数据库的格式成为水平数据格式（*TID*是事务标识符，而*itemset*是事务*TID*中项的集合），在使用CUDA进行挖掘前需要将事务数据集变为垂直数据格式，即数据可以使用项-*TID*集格式（即{*item*：*TID\_set*}），其中*item*是项的名称，*TID\_set*是包含该*item*的事务的标识符的集合。如表4-1表示的是将表1-1的数据格式变为垂直数据格式。

表4-1 事务数据库T（垂直数据格式）

|  |  |
| --- | --- |
| 项集 | TID\_set |
| I1 | {T1,T3} |
| I2 | {T4} |
| I3 | {T1,T2,T3,T4} |
| I4 | { T2,T3,T4} |
| I5 | {T1,T2,T3} |

之后将垂直数据格式变形，将*TID\_set*用二进制来表示，其中0表示事务Ti不存在于item中，1表示事务标Ti存在于item中，如表4-2所示。

表3-2 事务数据库T（二进制表示的垂直数据格式）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **项** | **T1** | **T2** | **T3** | **T4** |
| **I1** | 1 | 0 | 1 | 0 |
| **I2** | 0 | 0 | 0 | 1 |
| **I3** | 1 | 1 | 1 | 1 |
| **I4** | 0 | 1 | 1 | 1 |
| **I5** | 1 | 1 | 1 | 0 |

使用二进制的方式在GPU上进行并行计算的交运算，对于使用SIMD-PRAM模型的CUDA更适合，也方便在GPU上对候选项集的支持度计数进行计算。用于测试关联规则算法的实验数据全部来自频繁模式挖掘评测常用数据集，由FIMI（Frequent Itemset Mining Implementations Repository）提供（<http://fimi.ua.ac.be/data/>），本次的实现主要使用其中的三个模拟数据进行测试。以T40I10D100K数据集为例，其中T40表示一个事务中包含项的个数平均为40个，I10表示一个频繁项集中项的个数平均为10个，D100K表示此事务数据库中包含的事务数为10万个，另外此数据集包括999个项。

预处理需要对数据库扫描一遍，生成为用二进制表示的垂直数据格式，并将其存放在二进制文件中。二进制文件中存放的数据格式为：开始的4个字节存放事务的个数D，之后的四个字节存放的是全部项的个数t，之后依次将t个项用二进制表示的对应事务的集合放入后续二进制文件中，并且在每项对应的二进制数据前添加对应项的编号（占用四个字节）并在编号后预留4个字节的位置用于存放对应项的支持度计数值。如图4-1所示。

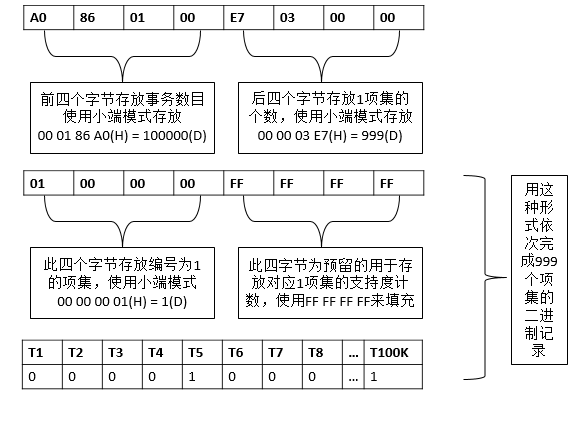


图4-1

使用OpenMp尽可能的进行并行优化，减少预处理的时间，对于处理事务数据为100K的数据量来说，大约有3s左右的减小，效果并非很理想。

1. 使用字典树来产生候选项

这里使用的字典树的方法产生候选项集，将在增量式关联规则挖掘算法中使用。字典树可以帮助快速存取候选项集（减少无谓的字符串比较），并对其进行剪枝。

字典树中每个节点保存三个数据分别是：项的编号值，从根节点到此项所构成的项集对应的支持度计数值，以及此节点指向的所有孩子的节点，如图5-1所示。

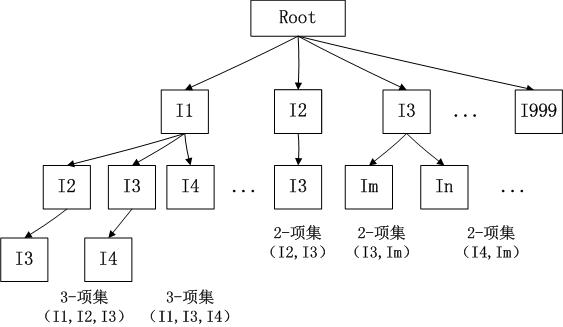


图5-1

生成候选k-项集：k=1时，直接计算每个1-项集的所对应的的支持度大小，删去不满足的支持度的项集，将剩下的项集放入字典树中。k=2时，将每一个深度为1的节点的全部右兄弟节点作为该节点的孩子节点，如5-1所示。k>2时，逐层构建候选项集，当字典树增长到第k层，则将生成全部的k-项集，生成候选k+1项集的方式和k=2一样，将当前叶节点的全部右兄弟节点作为该节点的孩子节点。之后对生成的候选k+1项集进行压缩剪枝，即验证生成的候选k+1项集的全部k项子集都在这棵树的前缀中，在图5-1中，需要删除{I1,I3,I4}候选分支中的I4节点。为了提高之后生成候选项集中项集压缩的效率，可以在删除死节点（如果当前字典数的最大深度为*k*，存在一个深度为*k-1*的节点，而该节点没有子节点，则该节点即为死节点），并且不会影响之后候选项集的子集验证的正确性。

在生成候选k-项集之后，就需要验证候选项集，找出满足最小支持度的节点。

1. 候选项集支持度计数的计算方法

当k=1时，即求1-项集的支持度计数值。在一个*m\*n*的事务数据位图中，*m*表示事务中所包含的项的个数，*n*表示事务的总个数，如果*bit（i，j）=*1，则项i存在于事务j中，否则如果*bit（i，j）=*0，表示项i不存在于事务j中。求得项目i的支持度计数值只需计算出的值，即项目i中*bit（i，j）*为1的数目。为了提高计算支持度计数效率，将二进制数据以16位一组来读入，将读取的无符号短整型的数值在查询表（*lookup table*）中查出该数值所对应二进制数中所包含的1的数目，并将其相加。另外查询表是一个一维数组，存储的是整型数的二进制表示中的1的个数。

当k≥2时,因为在计算支持度计算前，已经生成了候选k-项集。将每个k-项集中每个项对应的二进制事务位图做交运算，即可得到k-项集在全部事务中存在关系。如下图6-1所示当k=3时，候选3-项集所对应的支持度计数。

表6-1 项对应的事务的二进制数据

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | T1 | T2 | T3 | T4 |
| I1 | 1 | 0 | 0 | 1 |
| I2 | 1 | 1 | 0 | 1 |
| I3 | 1 | 0 | 1 | 1 |
| I4 | 1 | 1 | 1 | 0 |

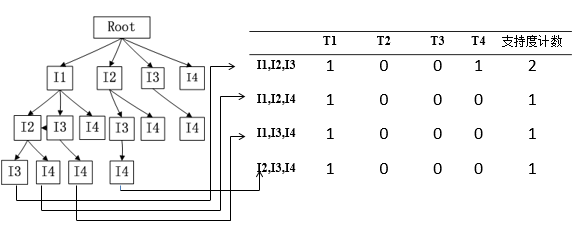


图6.1当k=3时，候选3-项集支持度计数值的计算

1. 使用GPU计算支持度

在说明CUDA部分的计算支持度程序设计之前，简要说明一下GPU的硬件结构和CUDA编程的一些问题。

GPU硬件由以下几个关键模块组成：内存（包括全局内存，常量内存，共享内存，这些概念在后续程序设计过程中会说明）；流处理器簇（Streaming Multiprocessor，SM）：一个GPU内有许多流处理器簇，他们就相当于CPU中的核。且一个SM有多个其他资源（包括存储资源，共享内存，常量内存，和成百上千的寄存器）组成；流处理器（Streaming Processors，SP）：最基本的处理单元。GPU进行并行计算，也就是很多个SP同时做处理。CUDA并行编程的核心就是线程的概念，一个线程就是程序中一个单一的执行流，一个个线程组合在一起形成了并行程序。CUDA的编程模型就是将线程组合在一起，形成线程束、线程块以及线程网格。GPU采用的数据并行模式，其需要成千上万的线程，来实现高效的工作。指令队列中的每条指令都会被分配到SM中的每个SP中。每个SM就相当于嵌入了N个计算核心（SP）的处理器。在CUDA中，CUDA可以创建很多线程，每个线程都有一个特殊的变量----线程索引来进行标识，并且每个线程都会执行同一个内核函数。线程都是每32个一组，当所有32个线程都在等待诸如内存读取这样的操作时，他们就会挂起。这组线程叫做线程束（warp），是SM调度和执行的基本单位，同样同一个warp中以不同数据资源执行相同的指令。当连续的线程发出读取内存指令时，读取操作会被合并在一起执行，由于硬件在管理请求时会产生一定开销，因此这样做将减少延迟。

首先，在用GPU做运算前，要将之前生成的二进制数据从CPU中放入到GPU的全局内存中，并在GPU全局内存中申请变量空间，用于之后做计算使用（之后会对申请的GPU空间进行说明）。

使用GPU进行运算，就要保证计算的数据量要足够大，能够尽可能充分利用GPU的计算能力。因此采取的方法就是同时计算很多个k-项集的交运算。在这里我们设一个数值为bound（取1000的原因会在3.2.6节进行说明），并令bound的值为1000。bound的取值关系到每次调用GPU的内核函数时，能够计算的候选项集的个数或者是能够进行交运算的项的次数。

当k≥3时，因为每一个候选项集中包含的项的个数超过2个，所以在求项集支持度计数之前需要对项集进行k-1次的交运算。在具体求bound个候选k项集的过程中（k≥3），主要包括了以下3种情况种情况，并且分别对应了内核函数的3种情况，设候选k项集为{*I1，I2，I3，...，Ik*}：

1) 项I1的二进制事务数据和项I2的二进制事务数据进行交运算。

通过循环遍历已生成的字典树，将目前的1000个2项集的信息存储在3个，整型数组中。这三个数组分别记录了两个项集中处于父项的项的id（parentIdList）和这些父项所有子项的id（childrenIdList），以及每个父项所包含的子项的数目（parent\_Children\_Lenlist），并且在进行GPU运算前，要为这些数组申请GPU内存空间，用于传递内核函数的参数。在调用内核函数时，要指定使用的线程块的个数（numBlock）和每个线程块中的线程数（numThread），以及为每个线程块分配的静态共享内存的大小，且静态共享内存的大小为3个数组所占的总字节大小。这里分配共享内存是要将上述的三个整型数组放入其中，来提高CUDA部分的运算效率。内核函数中的每一个线程处理交运算中的16位数据。在做交运算时，同样也是先读取存在GPU中的二进制数据，然后每个线程对16位二进制进行交运算。最后将中间结果放到实现定义好的GPU的全局变量中。这个中间结果的大小可以看成一个数组midRes，只是这个数组的长度为level（level指的是频繁k项集中k可能的最大值，可以设为15），数组中每一项为一个大小为numTranInByte\*overBound，其中numTranInByte指每一项所对应的二进制事务的字节大小，如有100000个事务则numTranInByte为100000/8=12500字节，overBound指比bound值大一些的值，可以是1500。第一次1000多个项集做完交运算会将中间结果放在数组midRes中索引为1的项中，为之后的交运算提供交运算的左值。

2) 中间结果Imid和项Ii（i={3,4,…k-1}）的二进制事务数据进行交运算。

和第一种情况很相似，第一种情况是父项和子项的二进制数据做交运算，第二种是上一次得到的中间结果（midRes[i]，i={1,2,…14}）和后续项的二进制数据做交运算，并将做完交运算的中间结果放到midRes[i+1]中，以此类推。当然中间还有很多细节需要考虑。

3) 最后一次，将得到的中间结果Imid和Ik的二进制事务数据进行交运算。

将最后得到的中间结果midRes[i]和Ik的二进制事务数据进行交运算，并将得到的结果直接进行支持度的计算，即求二进制结果中1的个数，如图7-1。每次读取16位数据并在查询表（lookup table）中查询对应无符号短整型的二进制数中包含的1的个数，不同的是要将查询表放入常量内存中进行查询。

常量内存是一块只读内存，它具有高速缓存的特点，需要在编译时特殊声明，大小只有64KB。因为在计算支持度计数时，需要频繁的访问查询表，与从全局内存中读取数据相比，从常量内存中读取数据更加节省内存带宽，所需的周期更少，提高计算支持度计数的速度。

计算完1000个左右的候选项集后，将对应的支持度计数放入字典树的对应叶节点中，之后对是否满足最小支持度计数阈值进行判断，删去不满足的节点。

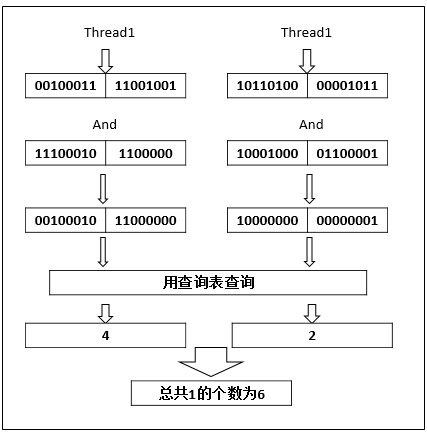


图7-1 在一个线程块中计算支持度

1. 关于优化的一些简单的说明

1) bound取值为1000的说明：上一节说明了需要使用共享内存，是因为很多线程需要同时访问同一个地址的存储体，来取出对应项的id值。如果使用共享内存来存取，会极大的提高运行效率。共享内存同常量内存很相似，当所有线程访问同一地址的存储单元时，会触发一个广播机制到线程束的的每一个线程中，不同的是常量内存是只读的，而共享内存是可读可写的，所以在此处使用共享内存更合适。

因为每个线程块可以配置的共享内存有16KB，所以最多可以存储4000个int型数据。且放入共享内存的三个整型数组的长度和最大为1000\*3=3000。另外当候选*k*项集中*k*的值大于3后，经过实验发现，放入共享内存的三个整型数组的长度都不会超过1000，所以在线程块中分配过多静态共享内存也不会对运行效率有任何提升。另外如果bound值设置的过大，当k=2时，有可能使得内核函数的执行的时间过长，当执行时间超过5s，操作系统会中断GPU端的计算。因为实验机器只有一个GPU，同时兼顾计算和图像显示功能，因此当GPU端计算时间过长时，会被中断，来保证图像显示。

2)线程块(numBlocks)和每个线程块中线程数(numThreads)的取值

根据分析本机GPU的一些基本属性，得出numBlocks=8，numThread=192。本实验所用GPU的计算能力为2.1，通过查询CUDA编程手册，得到了以下属性：

多处理器最大常驻线程数量：1536（一个SM中最多可以驻留1536个线程数）；

每个线程块中最大线程数量：1024；

多处理器最大常驻块数量：8（一个SM中最多可以驻留8个线程块）；

一般情况，一个线程块由128-512个线程组成，多个线程块会被加载到SM上运行，且一个线程块里面的线程数最好是32的倍数，这样可以更好的隐藏延迟。另外在CUDA的编程手册中，也建议一个block中至少要有64、192或者256个线程。

由以上的分析可知，当numBlocks=8时，numThread = 1536/8=192，满足上述说明的条件。实验也证明，当线程块和每个线程块中包含的线程数分别为8和192时，运行的效率更好。

1. 将上面谈到的计算支持度计数的思想运用到增量式的关联规则中

将使用CUDA计算GPU的思想应用到FUP的增量式关联规则挖掘过程中。当前只假设数据库事务增加的情况，并且暂且称之前使用CUDA计算支持度的方法叫做GPU\_compute\_minsup.

1) 先使用GPU\_compute\_minsup方法对增加部分的数据集db进行挖掘得到频繁项集L(db)。

2) 将原先对数据集DB挖掘的结果L(DB)与L(db)进行比较，求出同时存在于L(DB)和L(db)的项集，记为L(DB)∩L(db)，并将每一个项集在数据集DB和db中的支持度计数值相加，得到新的支持度计数值。

3) 对L(DB)和L(db)进行对比，求出属于L(db)但不属于L(DB)的项集集合，记为L(db)-L(DB)，这些项集可以视为数据集DB中的候选项集，并将这些项集根据每个项集中包含项的个数，逐层构建为字典树，类似第五节构造的候选项集树，之后使用GPU\_compute\_minsup对候选项集计算其对应的支持度计数值。

4) 求出属于L(DB)但不属于L(db)的项集集合，记为L(DB)-L(db)，这些项集视为数据集db中的候选项集，并将这些项集根据每个项集中包含项的个数，逐层构建为字典树，使用GPU\_compute\_minsup对每一层候选项集计算其对应的支持度计数值。

1. 性能比较

测试的使用的计算机带有英伟达的计算能力为2.1的显卡——GeForce 610M，GPU的全局存储大小为2GB，处理器的最大频率为900MHz，显存的最大位宽为14.4GB/s。本实验所用计算机的CPU为Core i5-2450M，双核四线程，CPU主频为2.5GHz，机器主存为4GB。测试数据如下表所示

表10-1 两个测试数据集的信息

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 数据集 | 水平数据格式  存取数据集大小 | 二进制存储的  数据集大小 | 数据生成（转  换）平均时间 | 最小支持度  取值 |
| T10I4D1K | 41KB | 104KB | 65ms | 1%，1.5%，2% |
| T40I10D100K | 14.7MB | 11.9M | 11.6s | 1%，1.5%，2% |

1) 对比使用CPU和GPU计算原始数据的性能

图10-1 T40I10D100K数据集测试结果 图10-2 T10I4D1K数据集测试结果

由以上两个图标可以分析出，当事务数据量非常多的时候，使用GPU相比于使用CPU计算有较大的性能提升，且随着支持度阈值减小，GPU的运行速度回更快，当支持度为1.0%时，运行速度会有5倍的提升。而当数据集中事务数据较少时，GPU并不会比CPU运行的更快，反而更慢。这是因为数据量过小无法充分利用GPU的计算能力，并行度过低，且CPU和GPU之间交换数据就会花费比较多的时间。

2) 对比使用CUDA增量式挖掘和直接使用CUDA对更新后的数据库挖掘

在实验数据集的选择中，以数据集T40I10D100K作为原始数据集DB，以数据集T10I4D1K作为增加的数据集db，我们暂且称使用CUDA增量式挖掘的方法为FUP\_GPU。图10-3显示了在支持度阈值为1.0%的情况下，两种方法的挖掘所用时间。

图10-3 比较GPU\_Apriori和FUP\_GPU算法挖掘增量式数据库

由上图可以发现增量式挖掘反而比直接挖掘的速度慢一些，通过对增量式算法运行时间的分析，发现当候选项集集合为L(DB)-L(db)，在增加的数据集T10I4D1K上进行挖掘所需的时间最多，进一步分析，可以将这部分时间分成树生成时间，GPU计算支持度耗费时间，可以发现使用GPU计算支持度所用时间所占比重只有17.9%，即并行计算所花费时间在整个挖掘过程中所占比例不大，大部分时间都花费在树的维护。

另外由运行结果分析发现，L(DB)和L(db)公共集合中的频繁项集数比较小，L(db)-L(DB)中候选项集数也较小，而L(DB)-L(db)集合中候选项集数目过多，数量是L(db)-L(DB)项集数的100倍以上，这使得对L(db)-L(DB)集合中的候选项集在数据集DB中使用CUDA来计算项集的支持度计数，无法充分利用GPU的计算能力，甚至没有使用CPU来计算支持度计数块。L(DB)-L(db)集合中虽然项集数比较多，能较好的利用GPU的计算能力，但是花费在树的维护上的时间过大，也使得计算效率偏低。

如果L(DB)∩L(db)中项集比较多，而L(DB)-L(db)和 L(db)-L(DB)中项集偏小也会导致无法充分使用GPU计算能力，因此不适合于使用CUDA进行并行就按。比如在使用数据集T10I4D10K和数据集T10I4D100K做测试时，其中前者为增加的数据集，后者为原数据集，因为这两个数据集中代表L(DB)∩L(db)公共部分的集合很多（373个），而代表L(DB)-L(db)和L(db)-L(DB)部分的集合中包含的项集数目很少，仅有10个左右，很明显使用CUDA来对这10个项集进行支持度计算，不会提高算法的效率

1. 全文总结

使用GPU来对Apriori算法和FUP算法进行改进的思想比较简单，没有过多算法上的考虑，主要是要做到充分利用利用GPU的计算性能，这是使用CUDA并行思想提升效率的主要目标。以下我将对本文中存在的缺陷和未来改进列举出来。

1. 测试数据集不够大，测试数据的种类不够多，对更新的数据集的选择不够严谨
2. 在数据预处理方面可以考虑使用MPI进行预处理，尽力加快预处理的时间
3. 将CUDA应用到增量式挖掘的处理方法有些简单，之后将不使用树来存储维护信息，减少对树的维护所花费的时间，并且没有考虑很多细节性的优化并且还没有考虑当数据库事务数据减少的情况
4. 在对支持度进行规约计算上，思想有些简单，没有能利用CUDA进行规约加法计算。
5. CUDA的优化思想还需要更进一步学习，优化思想是CUDA的精髓所在。