第四章 基于通用图形处理芯片的大数据挖掘技术

宋秋革

目录

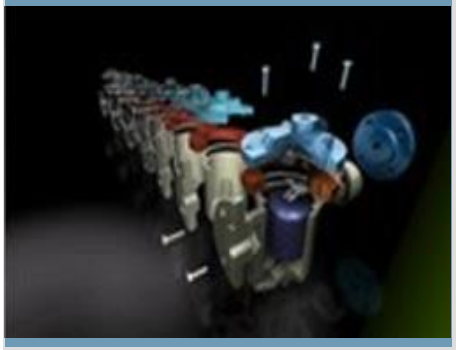
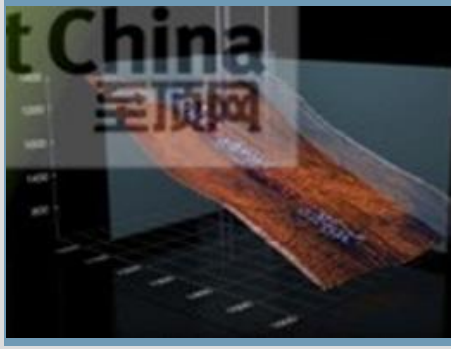
CONTENTS

4.1 图形处理芯片

NVIDIA公司在1999年首先提出GPU的概念。
图形处理器（Graphic Processing Unit，GPU）被定义为“一个集成了几何变换、光照、三角形构造、裁剪和绘制引擎等功能的单芯片处理器，具有每秒至少1千万个多边形的处理能力”。

3D技术发展
⇩
图形细节性和复杂性增强
⇩
图形处理速度的需求增加

4.1 通用图形处理芯片

工业设计	医学研究	物理学	石油勘探
			

- 巨大的计算需求
- 可观的并行性
- 吞吐量比延迟更重要

专用 ⇨

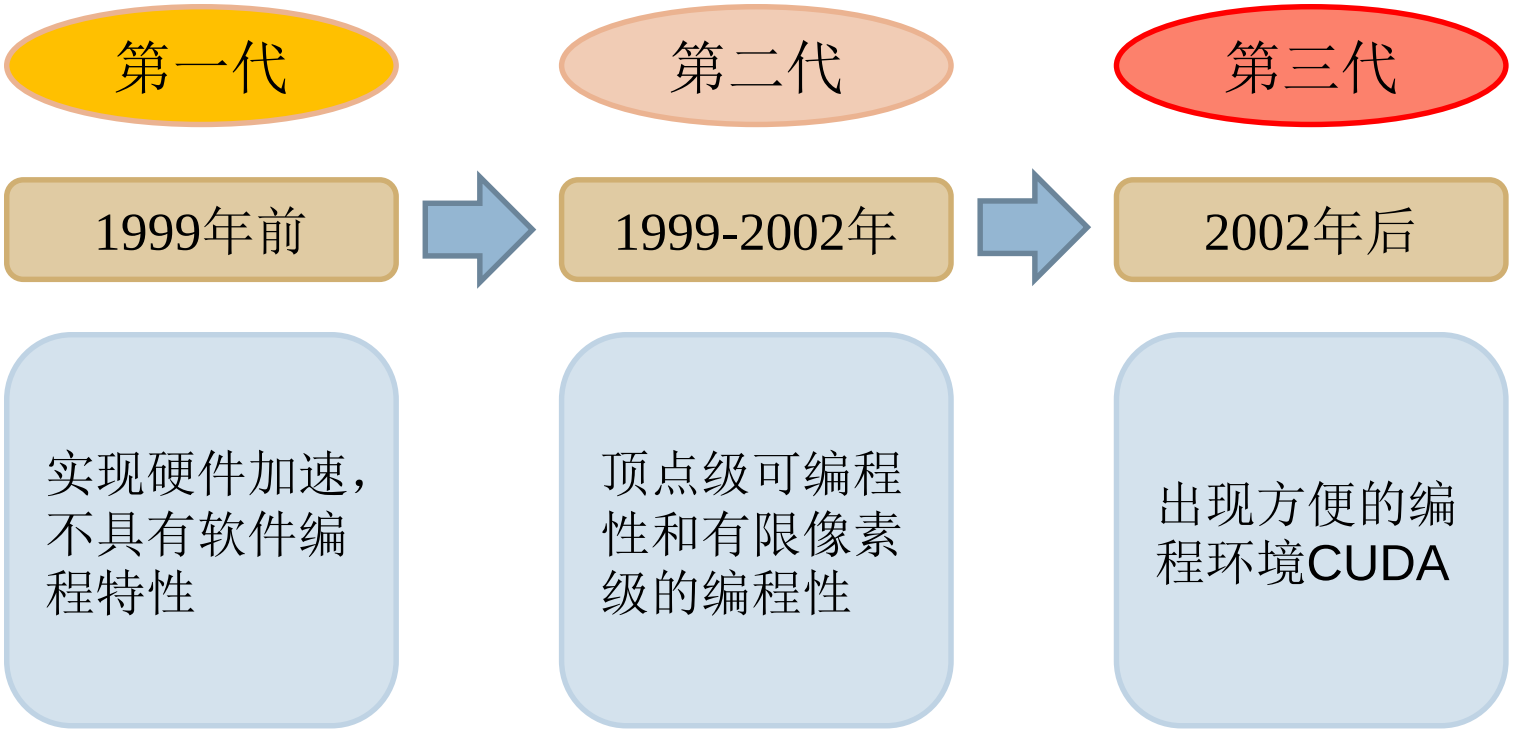
通用图形处理器，就是“一种利用处理图形任务的图形处理器来计算原本由中央处理器处理的通用计算任务”。

目录

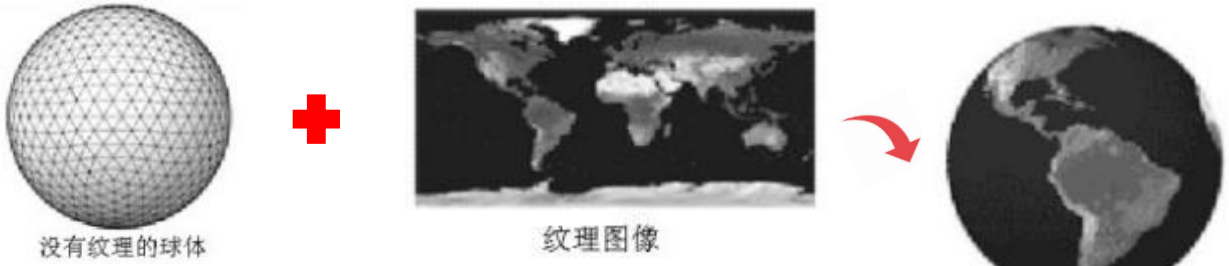
CONTENTS

4.1.1 图形处理芯片的架构

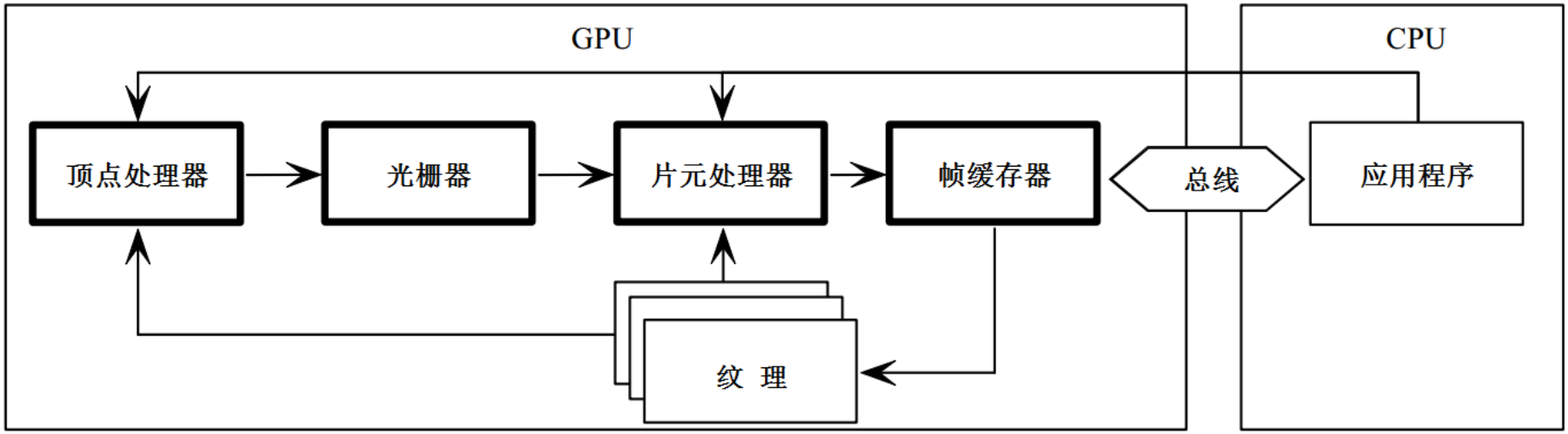
- GPU架构的发展历程



4.1.1 第二代GPU图形流水线



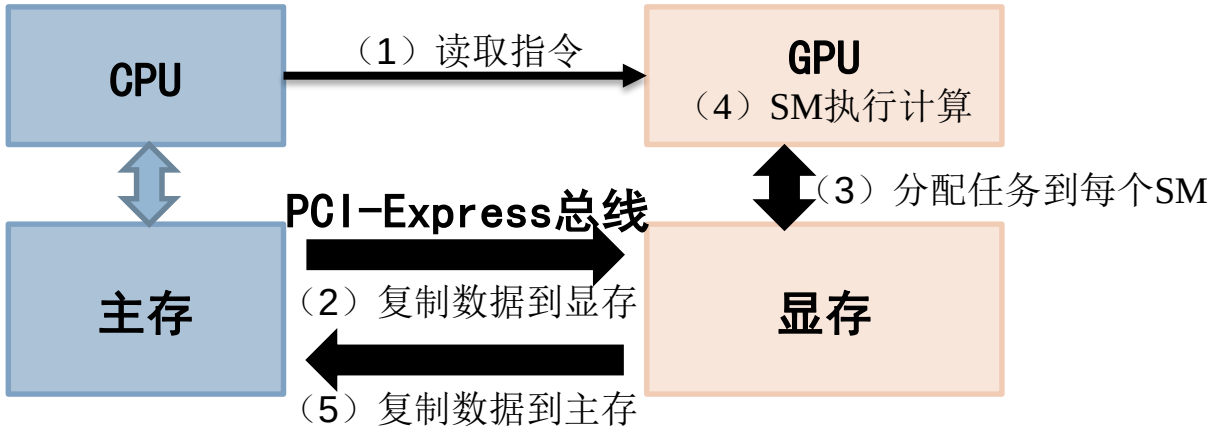
顶点操作 → 光栅化 → 片元操作 → 帧缓存



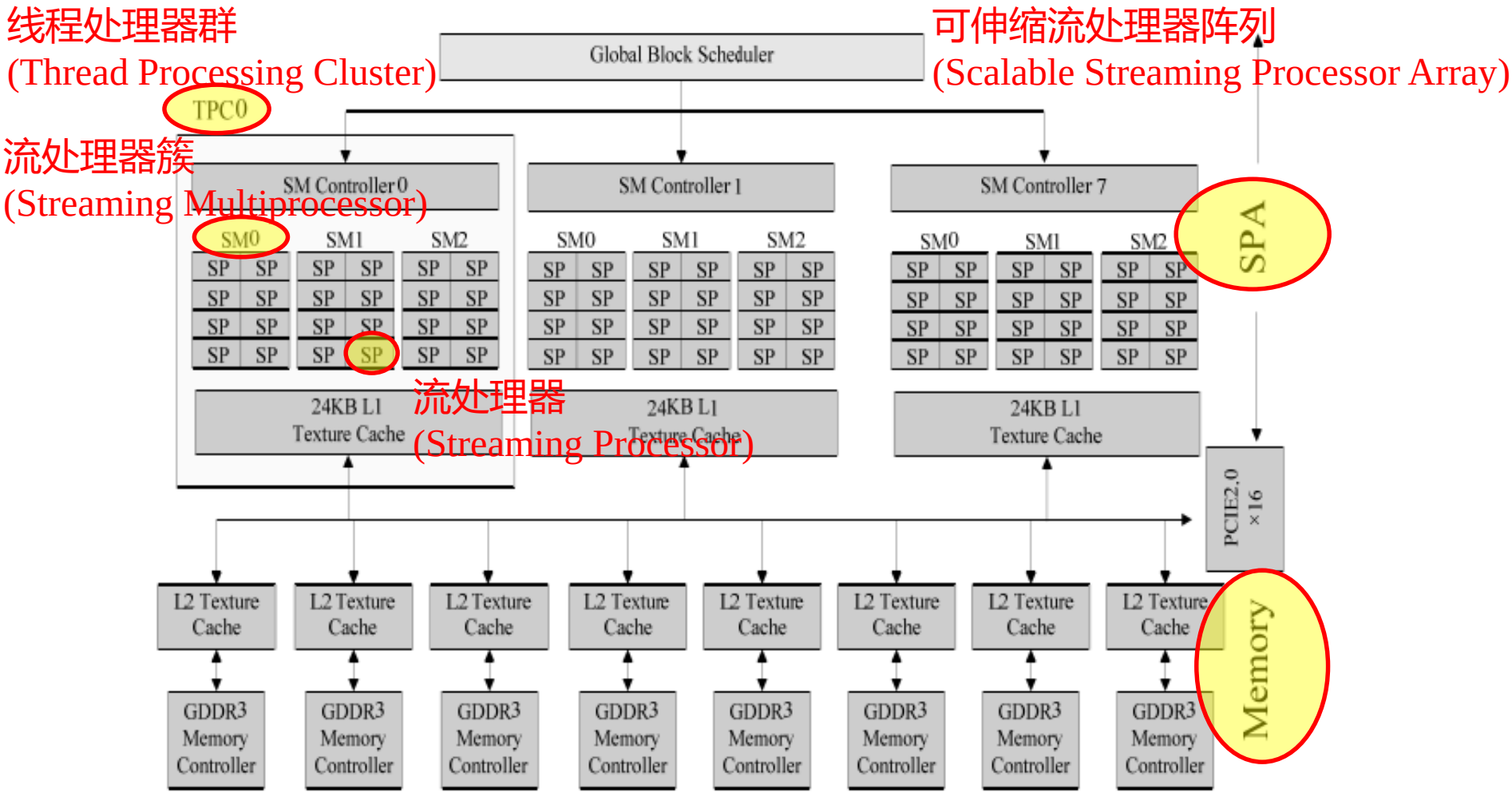
图形渲染流水线简图

4.1.1 第三代GPU的体系结构

- GPU的重要组成部分：
  - 主机接口：它连接了GPU和PCI-Express总线，读取程序指令并分配到对应的硬件单元，负责CPU和GPU不同引擎之间和不同GPU之间的同步；
  - 复制引擎：完成GPU内存和CPU内存之间的数据复制传递；
  - 流处理器簇：GPU最核心的部分，负责并行计算；
  - 内存：有各种不同类型的内存。
- GPU的工作流程：



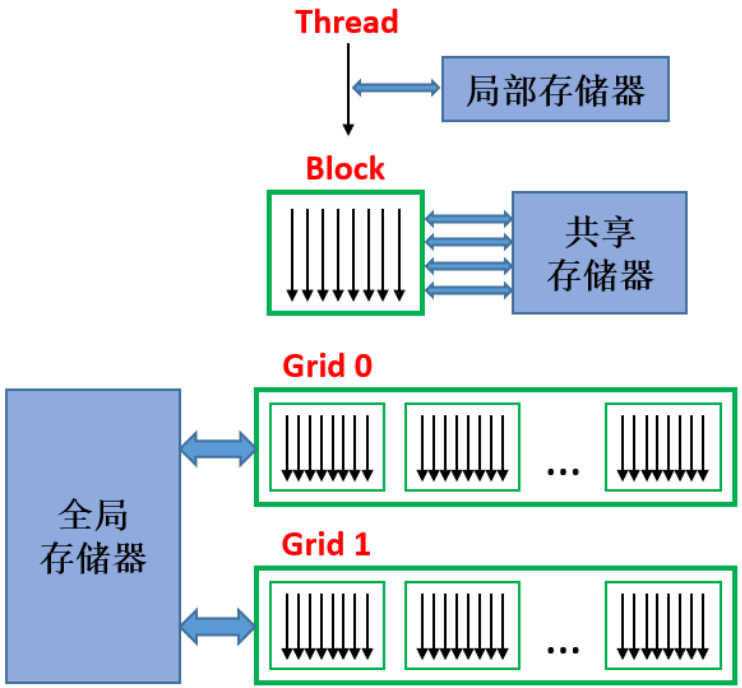
4.1.1 第三代GPU的体系结构



GT200 体系结构

4.1.1 统一计算架构CUDA

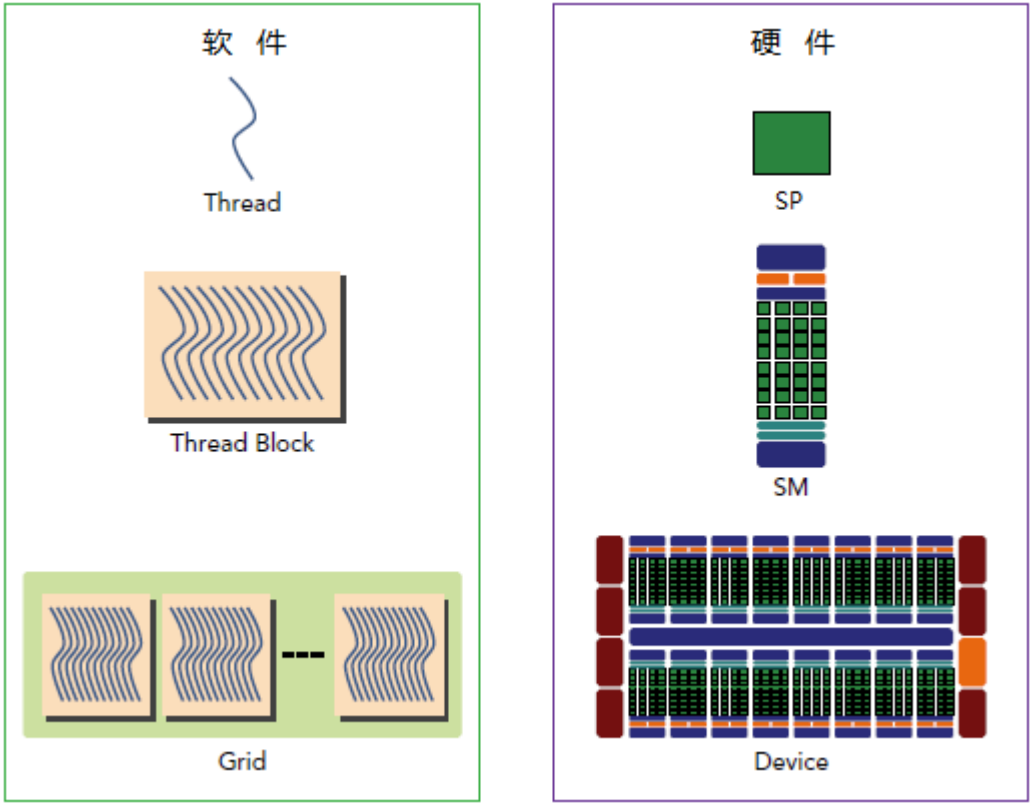
- CUDA（Compute Unified Device Architecture）
  - CPU函数（主函数）：调用GPU函数；
  - GPU函数（内核函数）：由GPU中的流处理器以多线程的形式执行。



CUDA的逻辑结构

4.1.1 统一计算架构CUDA

- GPU是CUDA的硬件基础；
- CUDA是GPU的实现工具。

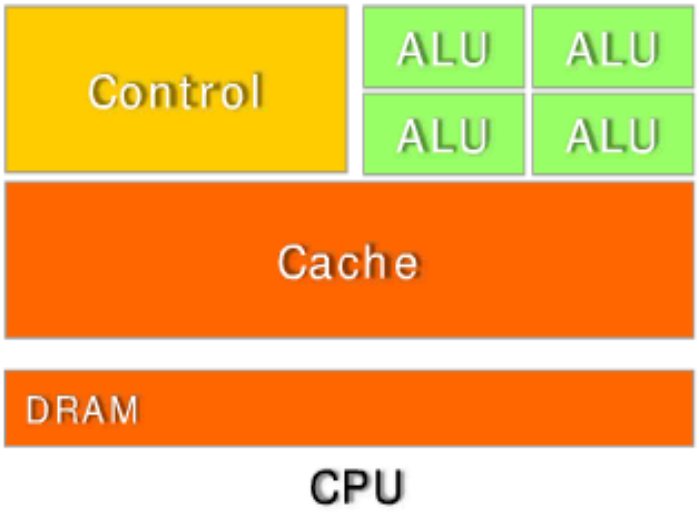


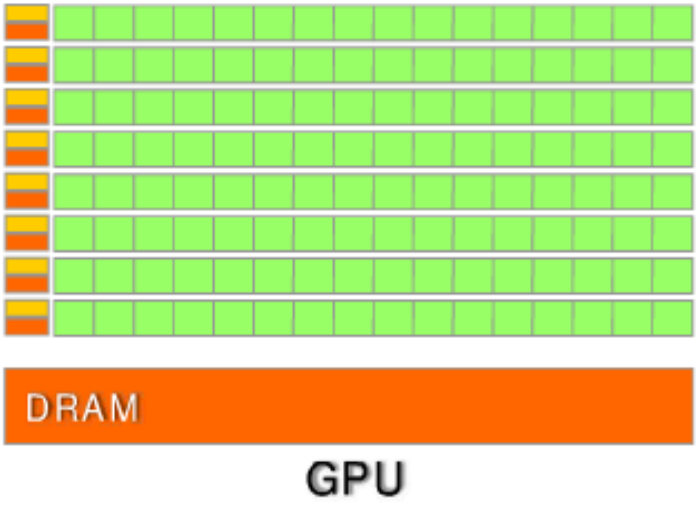
线程逻辑结构和GPU物理结构的对应关系

目录 CONTENTS

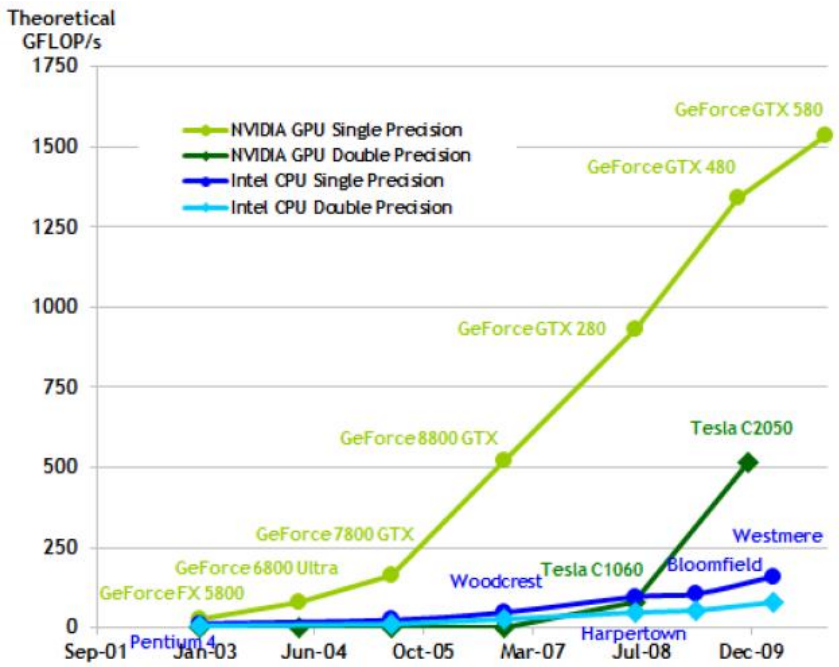
4.1.2 GPU与CPU硬件架构的对比

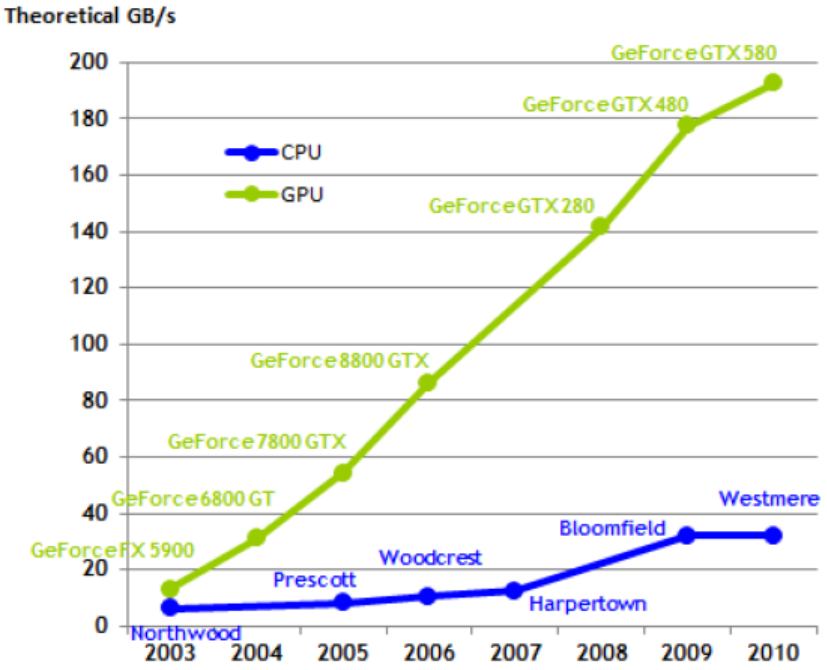
- CPU：面向通用计算；
  大量的晶体管用于Cache和控制电路
- GPU：面向计算密集型和大量数据并行化的计算；
  大量的晶体管用于计算单元





4.1.2 通用图形处理芯片的优点





强大的处理能力

高显存带宽

4.1.2 GPU与多核CPU的对比

- CPU的MIMD多核与GPU的SIMT众核
  - CPU的每个核心具有取指和调度单元构成的完整前端，因而其核心是**多指令流多数据流**(Multiple Instruction Multiple Data, MIMD)的，但这种设计增加了芯片的面积，限制了单块芯片集成的核心数量。
  - GPU的每个流多处理器才能被看作类似于CPU的单个核心，每个流多处理器以**单指令流多线程**方式工作，只能执行相同的程序。尽管GPU运行频率低于CPU，但其流处理器数目远远多于CPU的核心数（众核），其单精度浮点处理能力达到了同期CPU的十倍之多。
- 多核CPU针对的是指令级并行和任务级并行，而GPU则是数据级并行；

目录

CONTENTS

4.2 分类算法

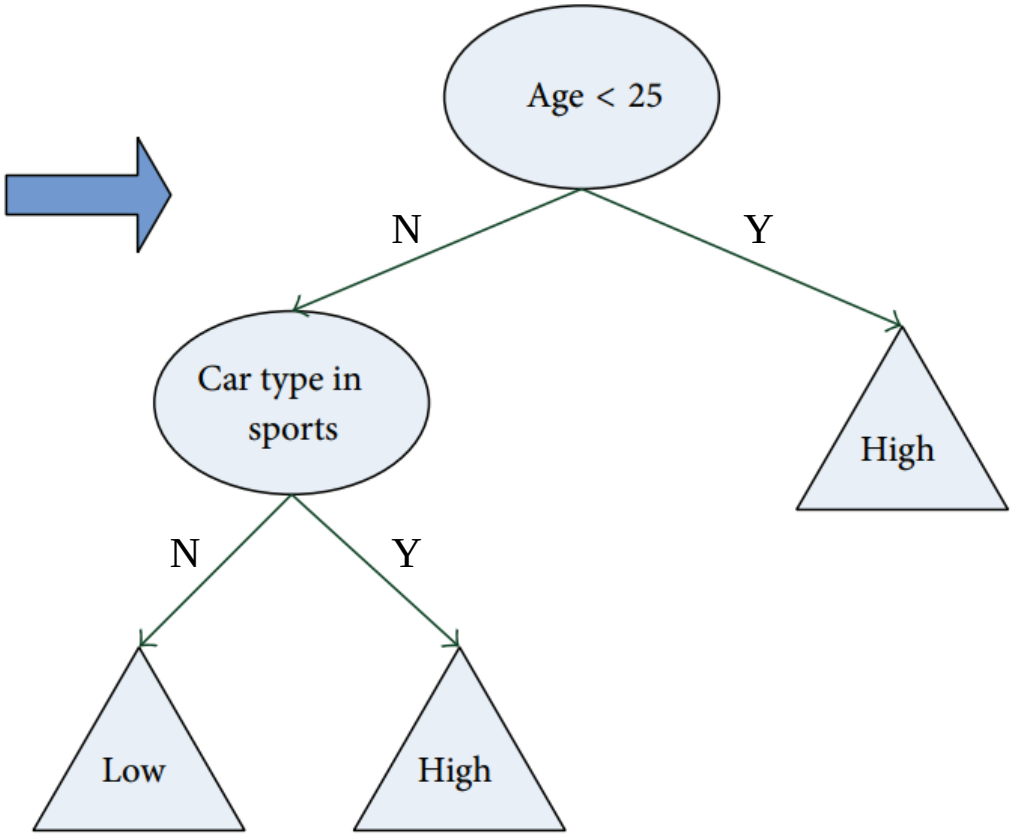
- 分类的概念：
  - 在一定的有监督的学习前提下，将物体或抽象对象的集合分成多个类的过程。
- 分类的应用：
  - 医生诊断人患有某病/无病
  - 银行判断某人可以授予贷款权限/不能贷款
- 常用的分类算法：
  - 决策树(ID3算法，C4.5算法，CART算法)
  - 支持向量机
  - 贝叶斯分类
  - K-近邻

4.2.1 决策树算法

- 决策树（Decision Tree）

Rid	Age	Car type	Risk
0	23	Family	High
1	17	Sports	High
2	43	Sports	High
3	68	Family	Low
4	32	Truck	Low
5	20	Family	High

Training data



决策树示例

4.2.1 决策树算法

- **定义：**分类决策树模型是一种描述对实例进行分类的树形结构。
  - 决策树由结点和有向边组成。
  - 结点有两种类型：内部结点和叶结点。内部结点表示一个特征或属性，叶结点表示一个类。
- **决策树学习通常包括3个步骤：**
  - 特征选择：信息增益，信息增益比，基尼系数
  - 决策树的生成：ID3，C4.5，CART算法，SPRINT算法
  - 决策树的修剪
- **缺点：**
  - 缺乏伸缩性：计算复杂性从$O(AN\log N)$到$O(AN(\log N)^2)$，其中，A是属性数目，N是记录数。在非常大的数据集上构造决策树的计算时间会很长，且受内存大小限制。

4.2.1 SPRINT算法

- SPRINT（Scalable PaRallelizable INduction of decision Trees）
  - 破除了主存限制
  - 引入了并行算法
  - 利用数据结构：
    属性列表
    - 属性值
    - 类标签
    - id号

Age	Class	Rid
17	High	1
20	High	5
23	High	0
32	Low	4
43	High	2
68	Low	3

（a）连续属性

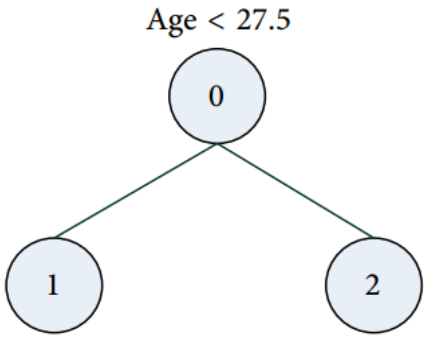
Car type	Class	Rid
Family	High	0
Sports	High	1
Sports	High	2
Family	Low	3
Truck	Low	4
Family	High	5

（b）分类属性

SPRINT属性列表示例

4.2.1 SPRINT算法

- SPRINT算法中属性列表消除了重复排序的开销
  - 减少了排序开销
  - 增加了分裂开销



Age	Class	Rid
17	High	1
20	High	5
23	High	0

Age	Class	Rid
32	Low	4
43	High	2
68	Low	3

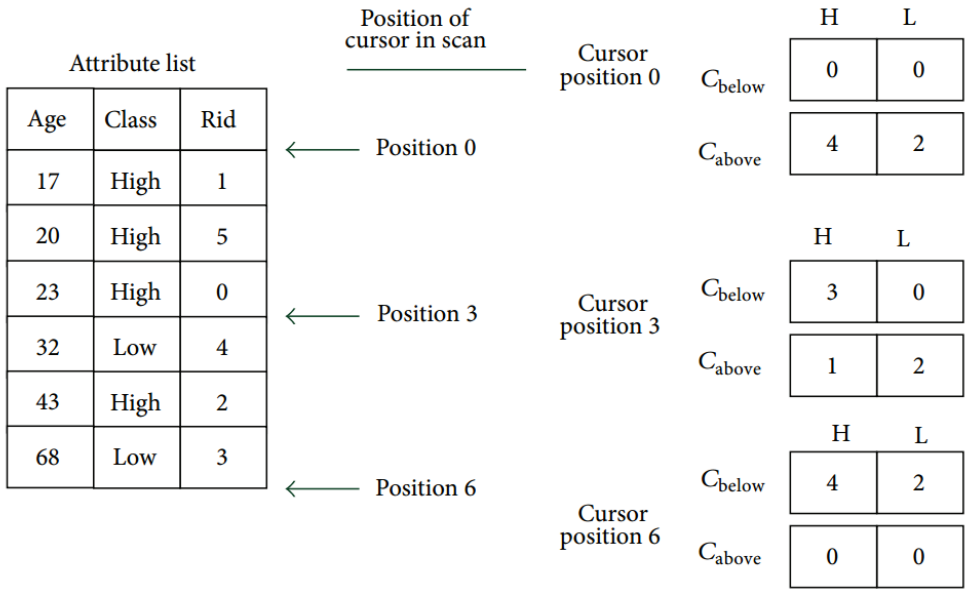
Car type	Class	Rid
Family	High	0
Sports	High	1
Family	High	5

Car type	Class	Rid
Sports	High	2
Family	Low	3
Truck	Low	4

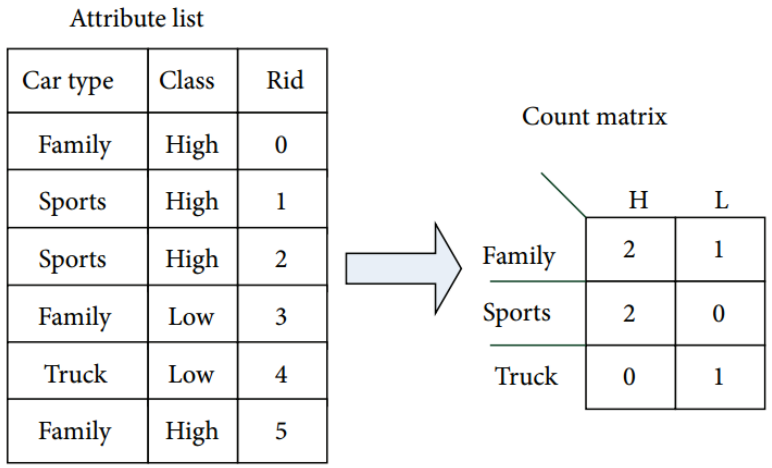
SPRINT算法中属性列表分裂示例

4.2.1 SPRINT算法

- SPRINT算法：寻找最佳分裂点
  - 连续属性：类直方图C_{below}，C_{above}
  - 分类属性：计数矩阵（Count matrix）



（a）C_{below}，C_{above}



（b）计数矩阵

4.2.1 SPRINT算法

基尼指数（Gini Index）

若集合T包含n个类别的m条记录，则其基尼指数是：

$$gini(T) = 1 - \sum_{j=1}^{n} P_j^2$$

其中P_j为类j出现的频率。若集合T分成两部分T_1和T_2，分别对应m_1和m_2条记录，则这个分割的基尼指数就是：

$$gini_{分割}(T) = \frac{m_1}{m} gini(T_1) + \frac{m_2}{m} gini(T_2)$$

提供最小基尼指数的分割就被选择作为集合T的最佳分割。

SPRINT算法的并行研究：

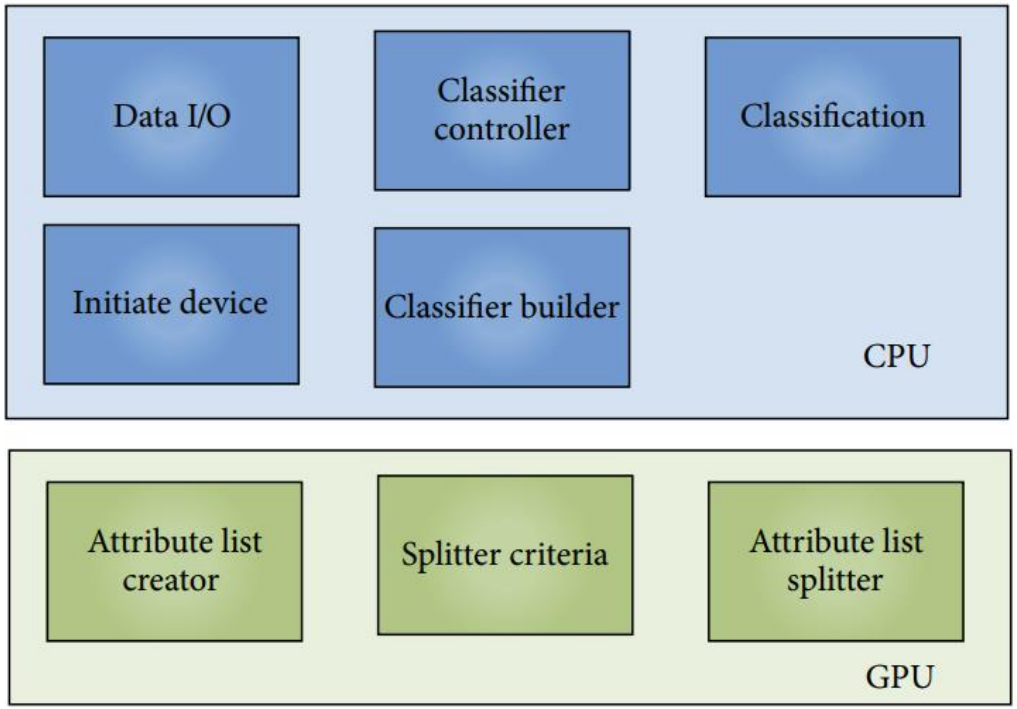
- 训练数据集在多处理器中的分布；
- 并行确定节点的最佳分割属性及确定最佳分裂点；
- 并行分割节点的属性列表到相应子节点。

4.2.1 CUDT算法

- 参考文献

Lo W T, Chang Y S, Sheu R K, et al. ***CUDT: a CUDA based decision tree algorithm***[J]. The Scientific World Journal, 2014, 2014.

- CUDT系统概括



4.2.1 CUDT算法

- **CUDT算法流程步骤：**

（1）把数据从磁盘加载到主存中

（2）初始化显存：查询显存信息，分配内存空间，复制训练数据到显存，分配内部缓存

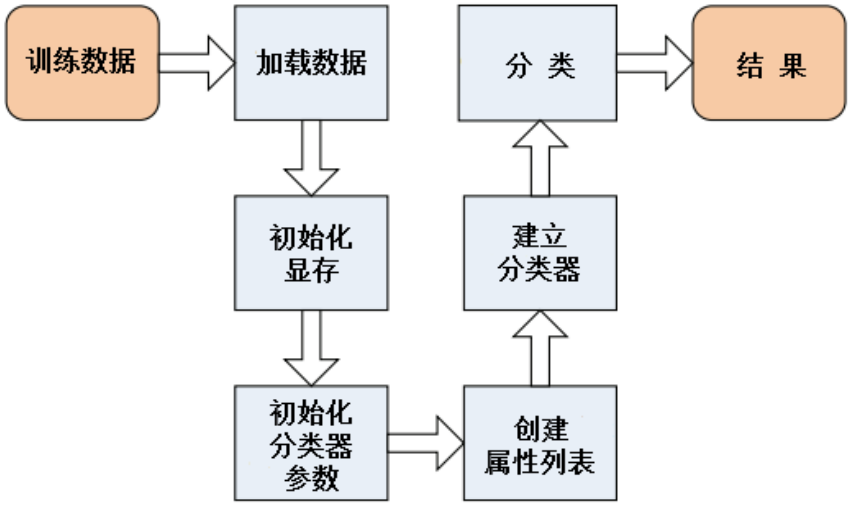
（3）根据用户需求设置分类器参数

（4）在设备存储器上创建属性列表并排序（GPU）

（5）建立分类器算法（CPU/GPU）

- 寻找分裂点
- 属性列表分裂

（6）在CPU上串行分类



CUDT算法流程图

4.2.1 寻找最佳分裂点

- 压缩（Compact）操作：去除冗余分裂点记录；
- 归约（Reduction）操作：基尼系数最小化。

4.2.1 属性列表分裂

属性列表

ID	0	1	2	3	4	5	6	7
属性值	0.1	0.2	0.2	0.4	0.5	0.5	0.5	0.5
类标签	0	0	0	1	1	1	0	0
标记数组	F	T	F	F	F	T	T	T
假值数组	0	1	1	2	3	4	4	4
地址	0	4	1	2	3	5	6	7

并行

地址	0	1	2	3	4	5	6	7
ID	0	2	3	4	1	5	6	7
属性值	0.1	0.2	0.4	0.5	0.2	0.5	0.5	0.5
类标签	0	0	1	1	0	1	0	0

分裂属性列表示例

- 分裂步骤：
  - 判断标记数组$FlagArray[i]$
  - 计算假值数组
  - 按照公式计算新地址

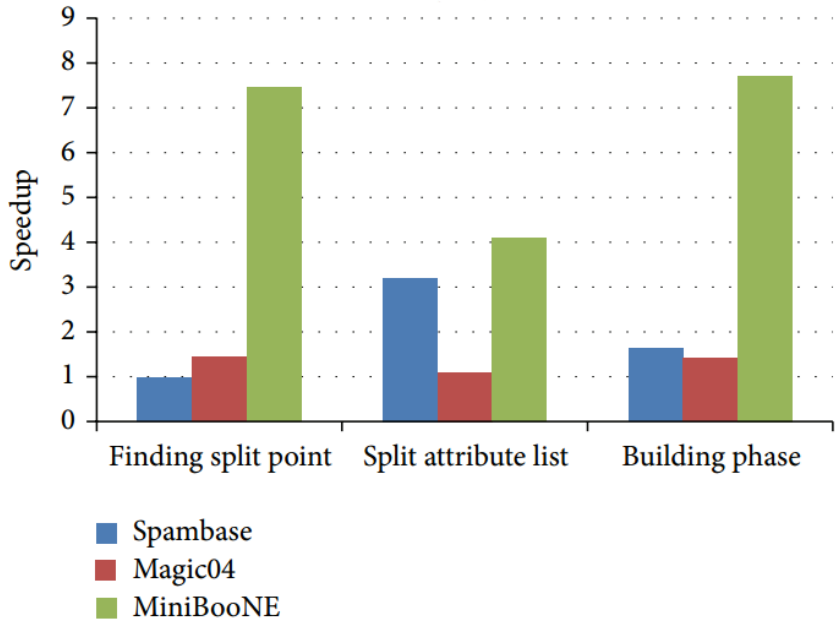
$$Address[i] = \begin{cases} i - FalseArray[i] + TotalFalse, 若FalseArray[i] = T \\ FalseArray[i], 若FlagArray[i] = F \end{cases}$$

  - 移动记录到相应的位置。

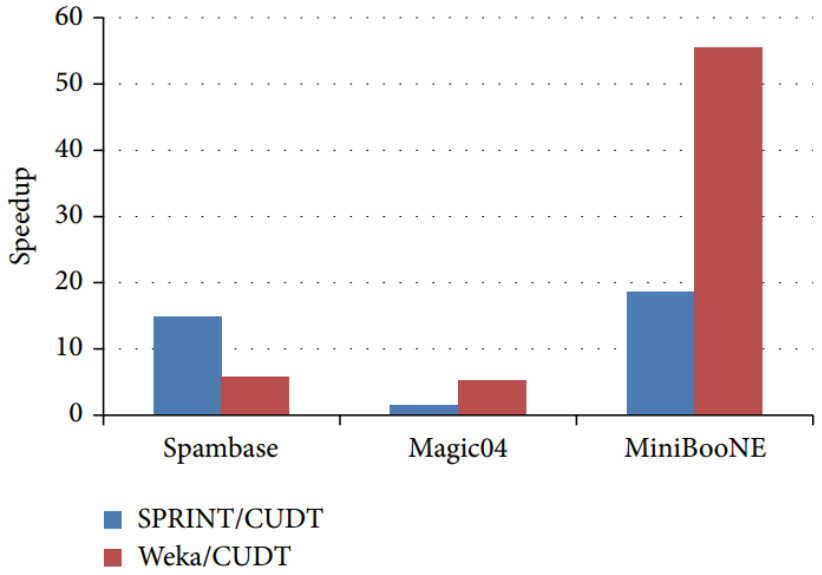
4.2.1 CUDT算法效率

Table 1: Test data set.

	Spambase	Magic Gamma Telescope	MiniBooNE particle identification
Number of attributes	58	11	51
Number of data	4601	19020	130065
Attribute type	Continuous	Continuous	Continuous
Number of classes	2	2	2
Source	UCI	UCI	UCI



(a) 算法每一部分的加速对比



(b) 不同数据集上的加速对比

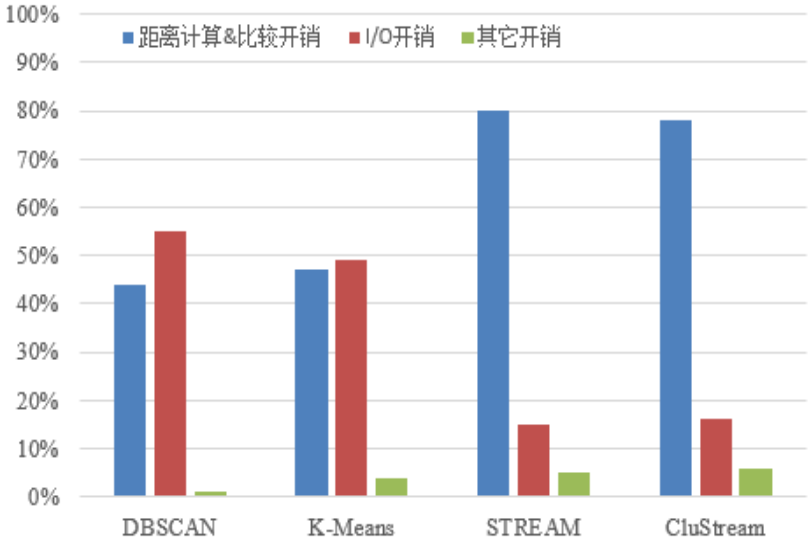
目录

CONTENTS

4.3 聚类算法

- **聚类**：将数据分为多个簇 (Cluster)，在同一个簇中的对象之间有较高的相似度，而不同簇的对象差别较大。

- **方法种类**：
  - 基于密度的方法(DBSCAN)
  - 基于划分的方法(K-Means)
  - 基于层次的方法(层次聚类)
  - 基于网格的方法

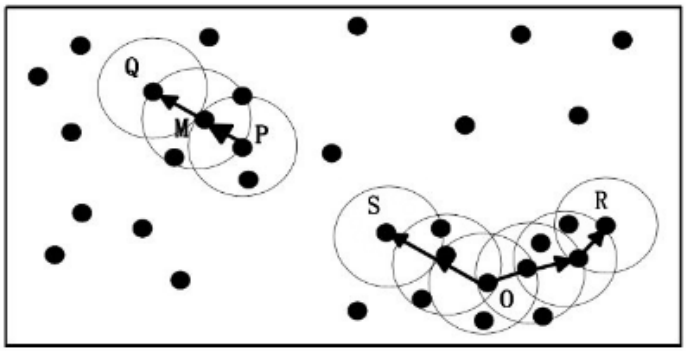


各种聚类算法的相对开销

- **CPU开销**成为制约可伸缩性提升的主要因素。

4.3.1 DBSCAN算法

- **DBSCAN算法**需要二个参数：
  - 扫描半径 （*eps*）
  - 最小包含点数（*minPts*）
- **DBSCAN算法原理：**
  - DBSCAN通过检查数据集中每点的*eps*邻域来搜索簇，如果点*p*的*eps*邻域包含的点多于个，则创建一个以*p*为**核心对象**的簇；
  - 然后， DBSCAN迭代地聚集从这些核心对象**直接密度可达**的对象，这个过程可能涉及一些**密度可达**簇的合并；
  - 当没有新的点添加到任何簇时，该过程结束。



概念示意描述（ *minPts* = 3）

核心对象：点*M*，*P*，*O*，*R*
直接密度可达：*P*→*M*，*M*→*Q*
密度可达：*P*→*Q*，*O*→*S*，*O*→*R*
密度相连：*R*和*S*

4.3.1 DBSCAN算法

Algorithm 1. DBSCAN

Input: a set of objects $\mathcal{D}$
Output: a set of clusters $\mathcal{C}$
1: Set the state of each object o in $\mathcal{D}$ as *unprocessed*;
2: $\mathcal{C} \leftarrow \emptyset$;
3: **while** there exist *unprocessed* objects in $\mathcal{D}$ **do**
4: Choose any *unprocessed* object $p \in \mathcal{D}$ and compute $N_\epsilon(p)$;
5: **if** $|N_\epsilon(p)| \geqslant MinPts$ **then**
6: Create a new cluster C containing only p and add C into $\mathcal{C}$;
7: $Seed \leftarrow N_\epsilon(p) - p$;
8: **while** there are *unprocessed* or *noise* objects in *Seed* **do**
9: **for** each *unprocessed* or *noise* object $q \in Seed$ **do**
10: Insert q into C and compute $N_\epsilon(q)$;
11: **if** $|N_\epsilon(q)| \geqslant MinPts$ **then**
12: $Seed \leftarrow Seed \cup N_\epsilon(q) - q$;
13: **end if**
14: **end for**
15: **end while**
16: **else**
17: Set the state of p as *noise*;
18: **end if**
19: **end while**

- 需要计算每两个点之间的距离，则时间复杂度为

$$O(n^2)$$

- 使用空间索引，则时间复杂度为

$$O(n\log n)$$

4.3.1 CudaSCAN算法

- **CudaSCAN算法包含三个步骤：**
  - **划分阶段：**把整个数据集划分成多个子区域，数量是GPU芯片上共享存储器大小的整数倍；
  - **子聚类阶段：**每个子区域并行地局部聚类，只需要计算每个子区域内对象间的距离，而不需要计算所有对象间的距离
  - **合并阶段：**合并局部聚类结果，可证：CudaSCAN算法的结果等于DBSCAN的结果。

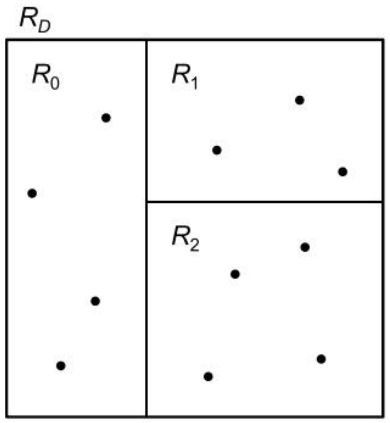
- **参考文献：**

Loh W K, Yu H. ***Fast density-based clustering through dataset partition using graphics processing units***[J]. Information Sciences, 2015, 308: 94-112.

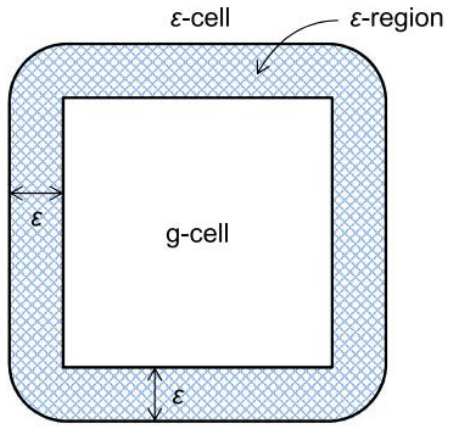
4.3.1 CudaSCAN算法——划分阶段

- **建立子区域的条件：**
  - 每个子区域的大小（对象数目×对象大小）应不超过预定义的大小S；
  - 在划分的时候，对象应尽可能地均匀分布，使得每个子区域包含相似数量的对象。

① $\forall R_i, R_j, i \neq j, R_i \cap R_j = \emptyset$
② $\cup_i R_i = R_D \ (0 \leq i < r)$
③ ε-cell之间可以有重叠



(a) 数据区域被分为3个g-cell



(b) g-cell与其对应的ε-cell

4.3.1 CudaSCAN算法——划分阶段

- **CudaSCAN划分算法描述：**

（1）整个区域被均匀分配成固定数目的子区域，即初始g-cell；

（2）为每个g-cell都计算出初始ε-cell；

（3）并行计算每个ε-cell区域中的对象数目，记为$Count[i]$；

（4）如果一个ε-cell的大小（对象数目×对象的大小）超过S，那么对应的g-cell必须要被继续划分。具体地讲，如果$Count[i]$大于$n = \left\lfloor \frac{S}{d \cdot f} \right\rfloor$，其中$d$是数据维度，$f$是每维属性的字节大小，那么该子区域必须要继续划分。

Input: a set of ϵ-cells $\mathcal{E}$
Output: the number of objects in each ϵ-cell in $\mathcal{E}$, i.e., $Count[i]$
1: Let $tID = blockID \cdot blockSize + threadID$;
2: **for** $j = tID$ **to** N **step** $gridSize \cdot blockSize$ **do**
3: **for all** $E_i \in \mathcal{E}$ **do**
4: **if** E_i contains $o_j \in \mathcal{D}$ **then** atomicInc($Count[i]$);
5: **end for**
6: **end for**

CudaSCAN: 计算$Count[i]$的内核函数

4.3.1 CudaSCAN算法——子聚类阶段

- 与**DBSCAN**算法的不同：
  - 子聚类算法只在一个ε-cell中的对象集合D_S上运行；
  - 集合D_S中每个对象也是被并行处理的。

- 对象数据和局部聚类的中间结果只能被相应的块访问，因此它们可以被存在共享存储器中，这可以显著地提高**CudaSCAN**算法的效率。

Input: a set of objects $\mathcal{D}_S$ within an ϵ-cell
Output: an array of local cluster IDs *cid*[] within an ϵ-cell
1: Set the state of each object o in $\mathcal{D}_S$ as *unprocessed*;
2: **while** there exist *unprocessed* objects in $\mathcal{D}_S$ **do**
3: Choose any *unprocessed* object $p \in \mathcal{D}_S$ and compute $N_\epsilon(p)$;
4: **if** $|N_\epsilon(p)| \geqslant MinPts$ **then**
5: Create a new cluster C and set $cid[p] = C$;
6: $Seed \leftarrow N_\epsilon(p) - p$;
7: **while** there are *unprocessed* or *noise* objects in *Seed* **do**
8: $q \leftarrow Seed[threadID]$ such that q is *unprocessed* or *noise*;
9: Set $cid[q] = C$ and compute $N_\epsilon(q)$;
10: **if** $|N_\epsilon(q)| \geqslant MinPts$ **then**
11: $Seed \leftarrow Seed \cup N_\epsilon(q) - q$;
12: **end if**
13: **end while**
14: syncThreads();
15: **else**
16: Set $cid[p] = noise$;
17: **end if**
18: **end while**

CudaSCAN：子聚类阶段内核函数

4.3.1 CudaSCAN算法——合并阶段

- 合并算法过程：
  - 判断哪些局部簇需要被合并，$MM[C][cid[j]]$（$MM[cid[j]][C]$也可）被设置为$true$；
  - 一旦MM[][]指示出哪些簇需要被合并，这些簇应该被分配到相同的簇编号。

示例：

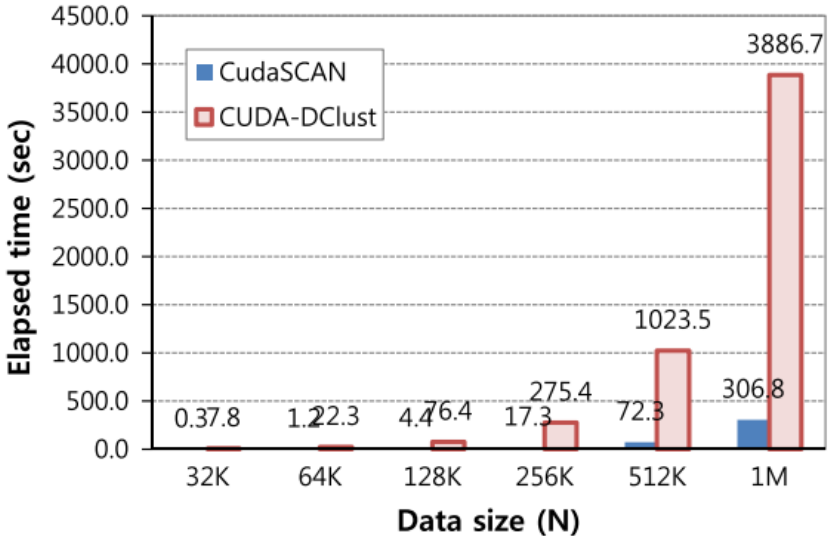
若$MM[3][23]$和$MM[23][34]$是$true$，簇3,23,34需要被合并，分配到相同的编号。

Input: *cid*[], local cluster IDs returned from Algorithm 4
Output: *clusterID*[], global cluster IDs, and *MM*[][], merge indicator
1: **for** $j = threadID$ **to** n **step** *blockSize* **do**
2: $i = \mathcal{D}_S[j]$; // $\mathcal{D}_S[j] = i$ when j-th object in $\mathcal{D}_S$ is i-th object in $\mathcal{D}$
3: C = atomicCAS(*clusterID*[i], *noise*, *cid*[j]);
4: **if** $C \neq noise$ **and** $C \neq cid[j]$ **then**
5: atomicExch(*MM*[C][*cid*[j]], *true*);
6: **end if**
7: **end for**

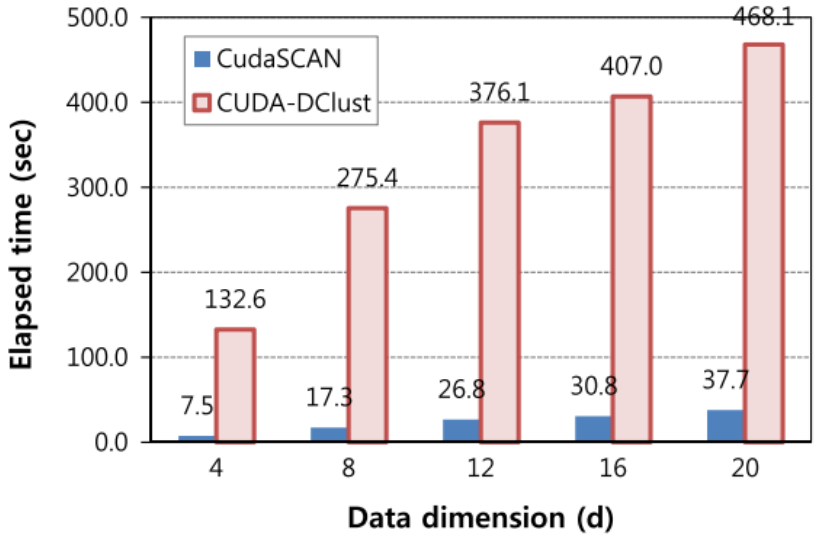
CudaSCAN：判断局部簇是否需要合并

4.3.1 CudaSCAN算法

- **CudaSCAN算法实验验证：**
  - 聚类结果与DBSCAN算法的结果是相同的；
  - CudaSCAN算法的效率远远高于普通的基于CPU的聚类算法的效率，适用于大规模数据集上的聚类。



(a) 效率随数据集规模的变化



(b) 效率随数据维度的变化

目录

CONTENTS

4.3.2 K-Means算法

- **K-Means算法描述**
  - 输入参数k：簇数目；S：点集合。
  1. 为每个簇选择一个初始点
  2. for each S中的点p
  3. 　将p加入到最近的簇中；
  4. 重新计算每个簇的中心点；
  5. if (有数据元素被重新分配) then Goto 第2行；

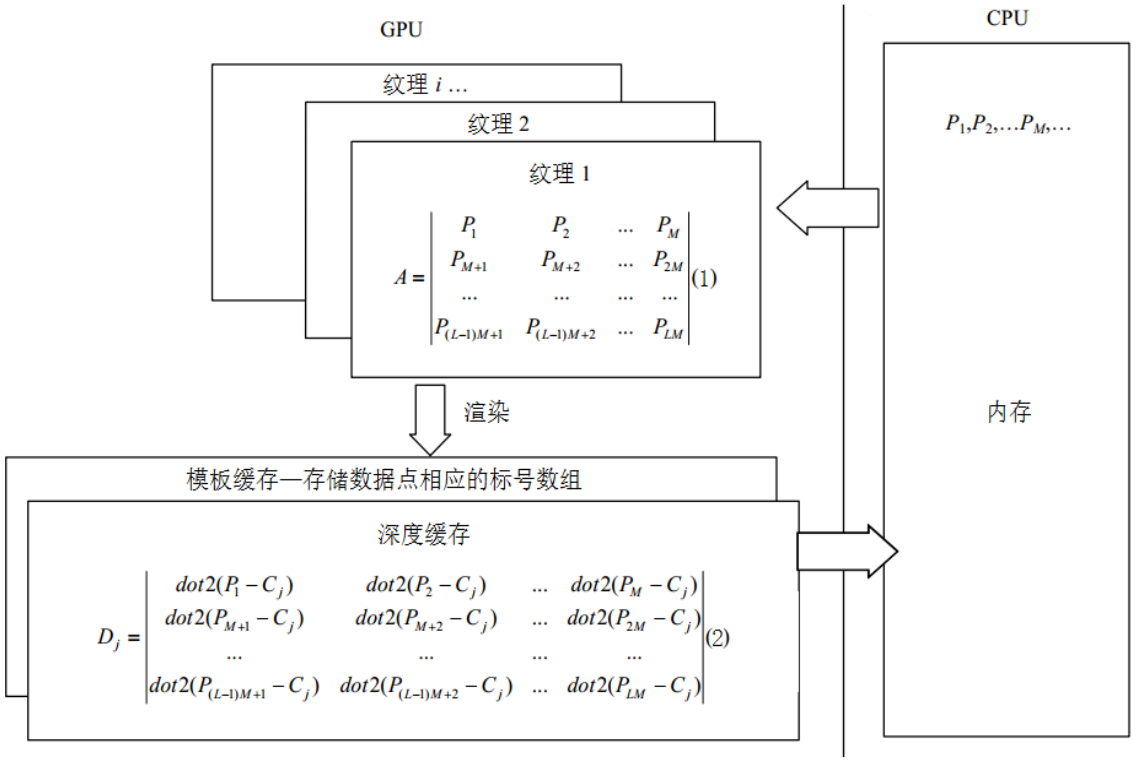
KRN

- **备注：**
  - k值的设置：过大/过小均不可行。
  - 初始点选择：所选取的每个点都尽量远离选过的点。
  - 算法终止条件：迭代次数达到上限时也可终止。
  - 距离计算和比较步骤往往是最耗时的部分。

4.3.2 并行K-Means算法

- 数据点的各个属性存储在一个纹理点的各个通道内，被组织成图中的矩阵(1)形式；
- 数据点矩阵对K个中心点$C_j(1 \le j \le K)$分别计算距离矩阵D_j，如图中的矩阵(2)。
- 由于模板缓存与各像素一一对应，并且与用于进行比较操作的深度测试具有内在关联，因而成为存储类标号的良好载体。

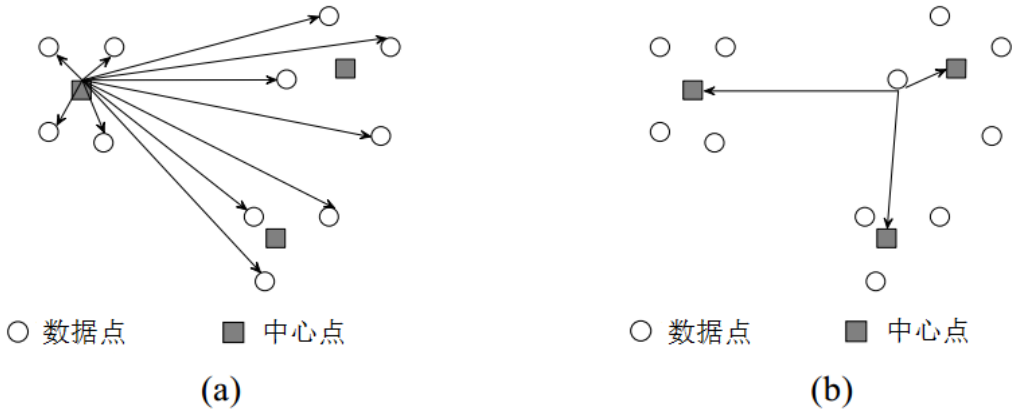
- Cao F, Tung A K H, Zhou A. ***Scalable clustering using graphics processors***[M]//Advances in Web-Age Information Management. Springer Berlin Heidelberg, 2006: 372-384.
- 曹锋，周傲英. 基于图形处理器的数据流快速聚类[J]. 软件学报, 2007, 18(2):291-302.



数据转换流程

4.3.2 并行K-Means算法——顺序调整

- 传统的基于CPU的K-Means聚类是**以数据点为中心的计算**，即一次计算从某个数据点到K个中心点的距离，并比较这K个距离值，从而获得该数据点最近的中心点，如图(b)所示。
- 基于GPU的并行K-Means聚类算法中，**以中心点为中心进行计算**，即一次计算从某个中心点到各个数据点间的距离，然后并行地与上个中心点的计算结果进行比较，如图(a)所示。这种计算方式可以通过**硬件并行**执行。



基于中心点和基于数据点的距离计算

4.3.2 并行K-Means算法——距离比较

- 距离矩阵中的距离比较通过**深度测试**来实现。
- **模板缓存**与各像素一一对应，并且与用于进行比较操作的深度测试具有内在关联，成为**存储类标号**的良好载体。
- **类标号计算过程：**

（1）计算距离矩阵D_1并直接存入深度缓存中；

（2）将模板缓存区中的模板值初始化为1；

（3）允许深度测试，实现距离矩阵之间距离的比较；

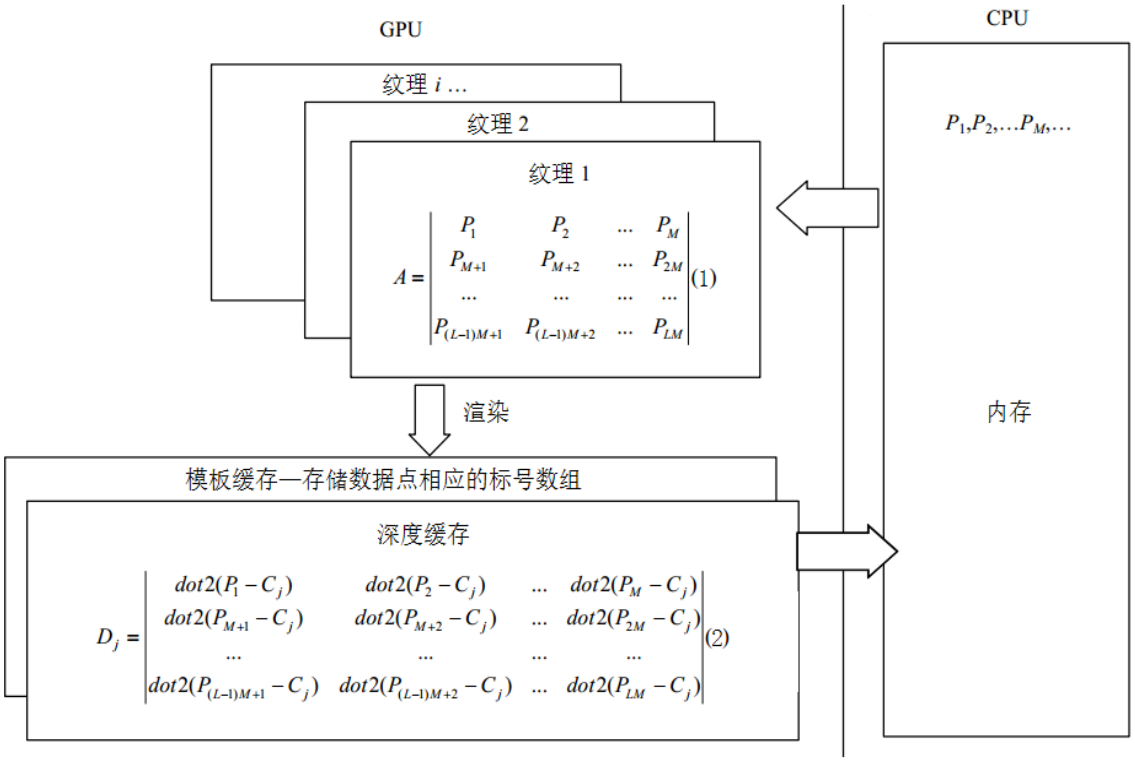
（4）依次计算其它距离矩阵D_{i+1}（$1 \le i < K$），若D_i中新产生的片元通过深度测试，则更新深度值，同时将对应像素的模板值更新为i+1；否则，对应像素的深度值和模板值保持不变；

（5）当K个距离矩阵都计算完毕后，模板缓存区中保存了与各个数据点最近的中心点标号，而深度缓存区中则保存了各个数据点到最近的中心点间的距离。

4.3.2 并行K-Means算法—中心点计算

- CPU通过向GPU输入新的中心点来控制聚类计算中的每一次循环。

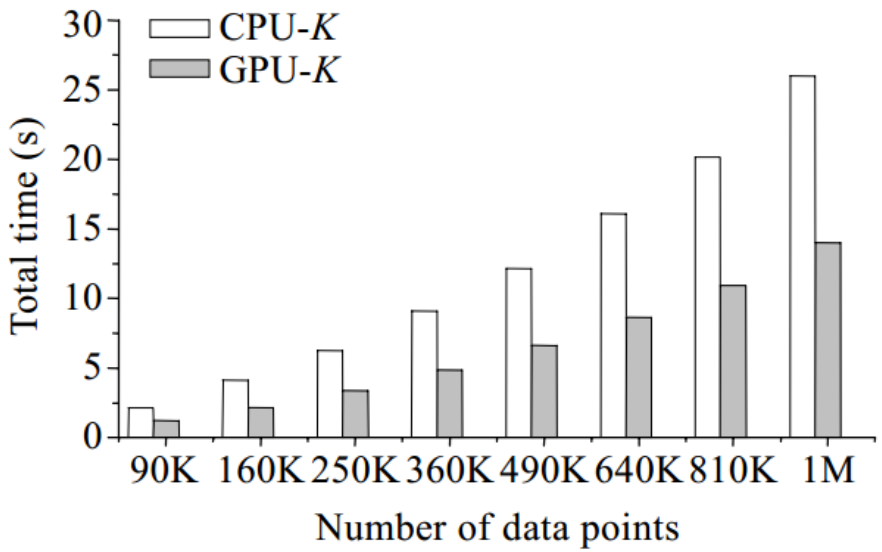
- GPU不适合处理累加运算，因而中心点的计算在**CPU**中进行。直接从模板缓存取回标号的数组，然后在CPU中将标号相同的数据点累加并生成新的中心点。



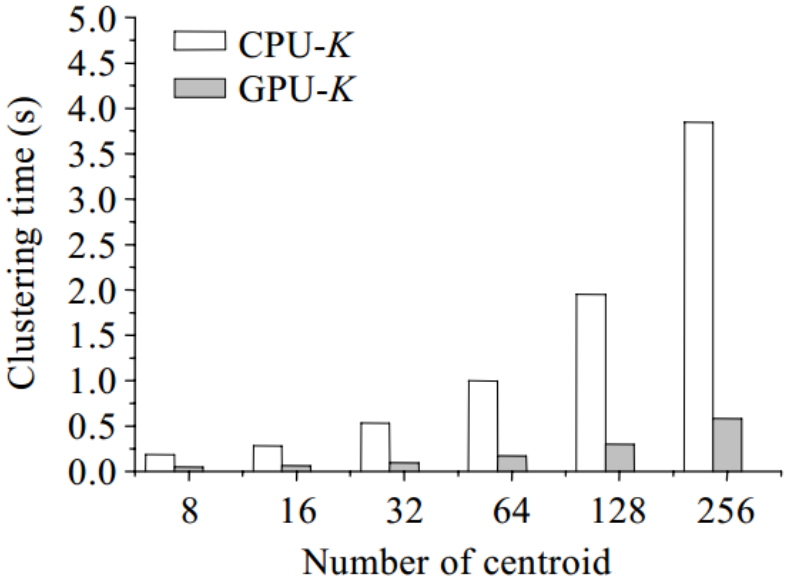
数据转换流程

4.3.2 并行K-Means算法

- **并行K-Means算法实验验证：**
  - GPU-*K* 和 CPU-*K* 随着数据点个数的增加，总体代价呈线性增长。GPU-*K* 的总体代价约为 CPU-K 的 60%。
  - 聚类代价随 *K* 而增加，GPU-*K* 相对于 CPU-*K* 的性能优势越来越明显。



(a) 效率随数据集规模的变化



(b) 效率随*K*值的变化

目录

CONTENTS

4.4 关联规则

- 也称**频繁模式挖掘**，是从大量数据集中找出有价值的项目间的关系。
- 典型关联规则分析案例：“啤酒和尿布”，“面包和牛奶”
- **基本概念定义：**
  - **事务数据库*D*：** $D = \{T_1, T_2, \dots, T_k, \dots, T_n\}$，$T_k$称为事务(Transaction)，$Tk = \{a1, a2, \dots, a_m, \dots, a_p\}$，am称为项 (Item)。
  - **项集(Itemset)：** 设$I = \{a_1, a_2, \dots, a_m\}$是事务数据库$D$中全体项组成的集合，$I$的任何子集$X$称为$D$的项集。
  - **项集支持度(Support)：** 项集X在事务数据库D中所出现的概率。
  - **频繁项集：** 事先给定一个最小支持度ζ，如果$support(X) \geq \zeta$，那么项集X称为频繁项集。
  - **规则置信度(Confidence)：** 规则$X \rightarrow a$的可信度等于集合$X \cup \{a\}$的支持度与X的支持度的比值。

4.4 关联规则

- 示例：

TID	Items
1	面包, 牛奶
2	面包, 尿布, 啤酒, 鸡蛋
3	牛奶, 尿布, 啤酒, 可乐
4	面包, 牛奶, 尿布, 啤酒
5	面包, 牛奶, 尿布, 可乐

support(啤酒，尿布) = 3/5

confidence(尿布→啤酒) = 3/4

support(面包，牛奶，尿布) = 2/5

confidence(面包→牛奶，尿布) = 2/4

若最小支持度minsupport=0.4，最小置信度minconfidence=0.75，

则会有一条强关联规则：尿布⟹啤酒

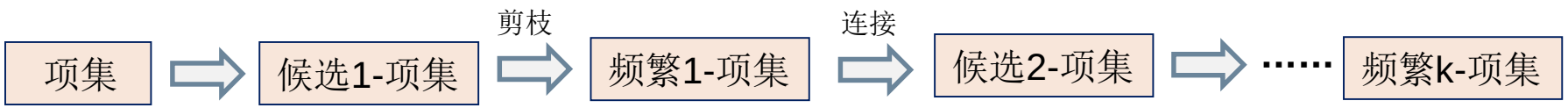
4.4.1 Apriori算法

- 最经典的关联规则算法，由Agrawal等人于1993年提出

Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. *Mining association rules between sets of items in large databases*. In Proc. of the ACM SIGMOD Conference on Management of Data, pages 207-216, Washington, D.C., May 1993.

- 两条定理
  - 定理1：若一个集合是频繁项集，则它的所有子集都是频繁项集。
  - 定理2：若一个集合不是频繁项集，则它的所有超集都不是频繁项集。
- 算法基本思路

逐层迭代，通过连接和剪枝来生成频繁项集



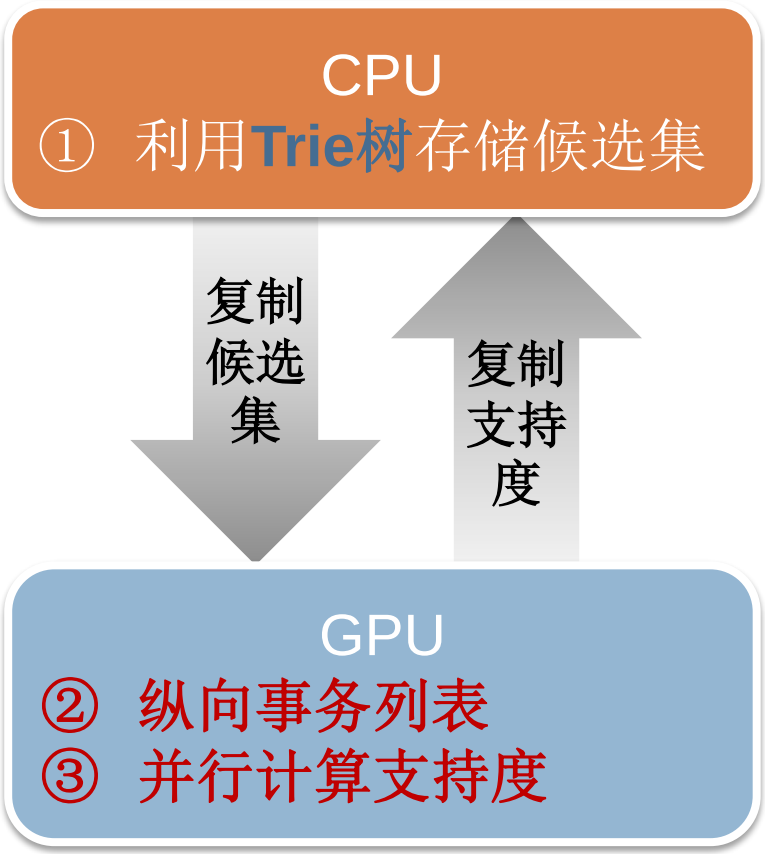
4.4.1 Apriori算法

- 随着数据集规模的不断增长，逐渐显现出一定的局限性：
  - 需多次扫描数据库，很大的I/O负载，算法的执行效率较低
  - 产生大量的候选项目集，会消耗大量的内存
- 针对这些问题，多种改进方法被提出：
  - 在分布式系统上并行执行（Hadoop，Spark）
  - 基于海量内存
  - 基于多核芯片
  - **GPApriori算法**

Zhang F, Zhang Y, Bakos J. ***Gpapriori: Gpu-accelerated frequent itemset mining***[C]//Cluster Computing (CLUSTER), 2011 IEEE International Conference on. IEEE, 2011: 590-594.

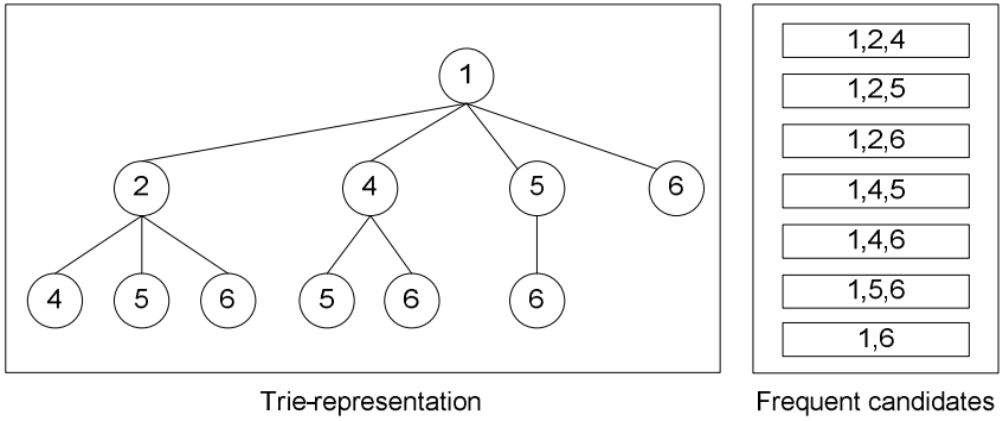
4.4.1 GPApriori算法

- GPApriori算法的创新
  - Trie树结构
  - 纵向事务列表
  - 并行支持度计算



4.4.1 GPApriori算法

- Trie树结构
  - **主要思想：**k^{th}代的候选集和$k+1^{th}$代的候选集共享相同的k个前缀，因此来自于不同代的候选集可以被存储在一个层次树结构中。
  - 算法通过把一个叶子结点和它的兄弟结点分别合并来产生新的叶子结点，从而产生新一代的候选集。



Trie树结构示例

4.4.1 GPApriori算法

- 事务表示方法
  - **横向表示：**只是存储一个由事务组成的项的列表，比较直观。
  - **纵向表示：**存储一个对应于每个候选集的事务id列表（tidset），每个列表也可以被表示为一个位掩码（bitset）。

Transactions	ID
1,2,3,4,5	1
2,3,4,5,6	2
3,4,6,7	3
1,3,4,5,6	4

(a) 横向表示

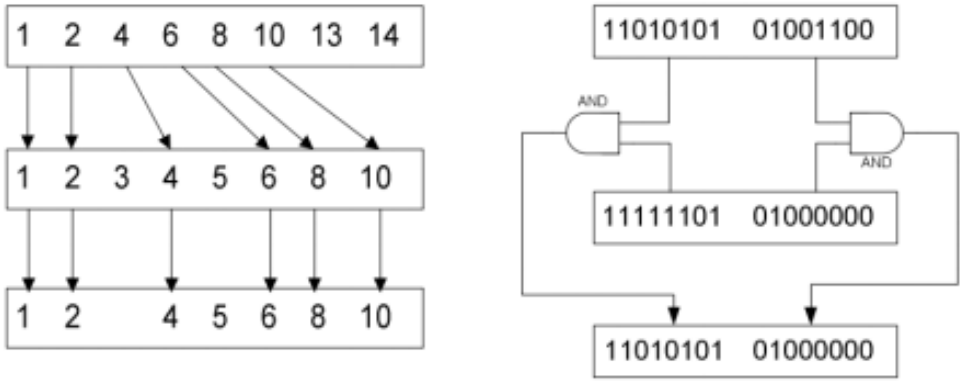
Candidate	tidset	bitset
1	1,4	1001
2	1,2	1100
3	1,2,3,4	1111
4	1,2,3,4	1111
5	1,2,4	1101
6	2,3,4	0111
7	3	0010
1,2	1	1000
1,3	1,4	1001
1,4	1,4	1001

(b) 纵向表示

纵向表示方法被许多最新的Apriori算法采用，实验证明纵向表示在大多数测试集上通常能将算法效率提高一个数量级。

4.4.1 GPApriori算法

- 新候选集的支持度通过统计纵向列表中的元素数得到。
  - **tidset连接：** tidset被存储在线性有序数组中，tidset虽然是紧密的，但其连接操作高度依赖于数据，是难以并行化的；
  - **bitset连接：**bitset虽然需要更多的内存空间，但其连接操作可以执行为两个位向量之间的“按位与”操作，这更适合设计为并行的集合连接操作，因此bitset更适合于GPU。



(a) tidset连接 **(b) bitset连接**

4.4.1 GPApriori算法

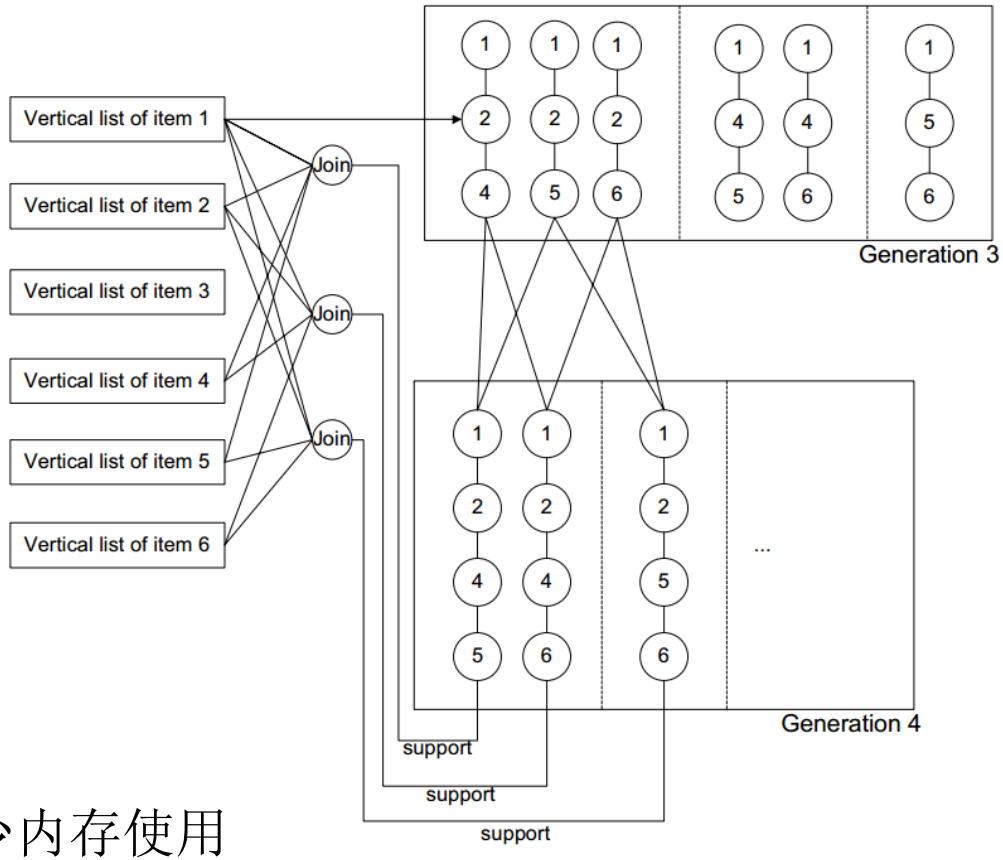
- 支持度计算

CPU：利用Trie树存储候选集

复制候选集

复制支持度

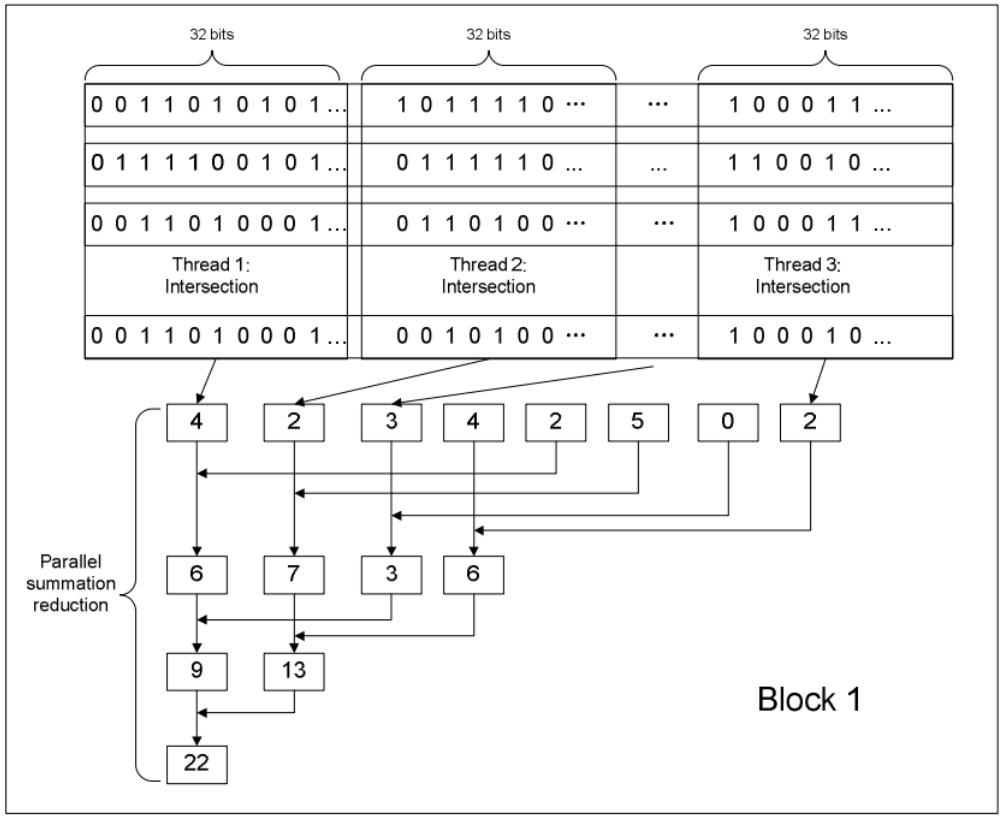
GPU：通过在纵向事务列表上并行bitset交运算计算支持度



- 增加了计算复杂度，以减少内存使用率和内存操作。

4.4.1 GPApriori算法

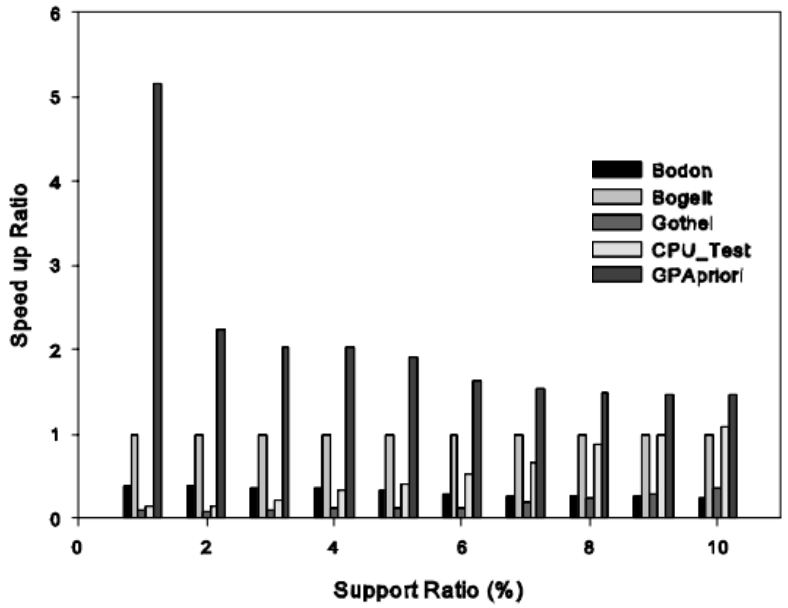
- GPU上线程部署
  - 每个列表的交运算被置于一个块上进行计算。
  - 每个线程的交运算的结果被存储为一个32比特的整数。
  - CUDA内置的popcount函数统计“1”的数量，并存储在共享存储器的一个整数数组中。
  - 并行的求和归约算法。



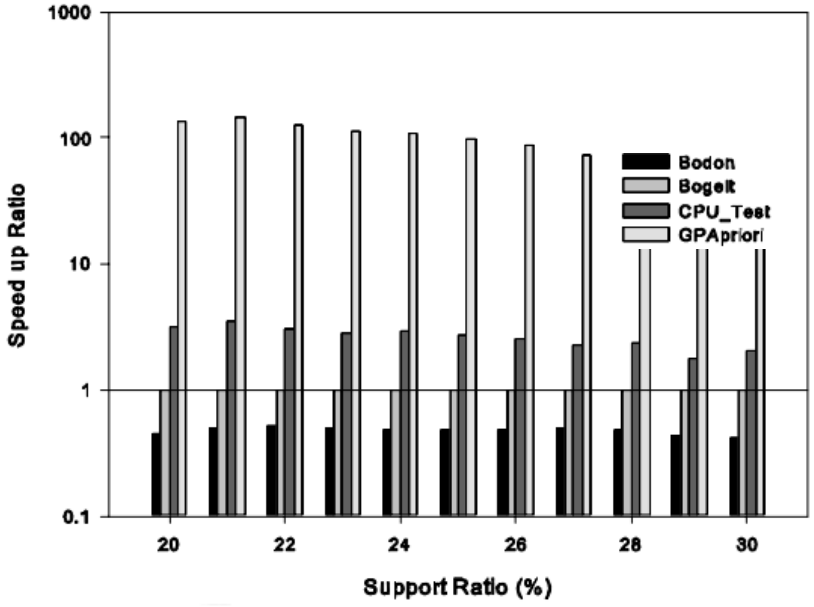
计算块上的线程部署

4.4.1 GPApriori算法

- **GPApriori算法实验验证：**
  - 在较小的数据集上，GPApriori算法可以加快至10倍；
  - 在较大的数据集上，可以加速50倍到80倍。



(a) 较小数据集



(b) 较大数据集

总结

CONCLUSION

- 图形处理芯片的基础理论
- 基于通用图形处理芯片的大数据挖掘技术
  - 并行分类算法（**CUDT**）
  - 并行聚类算法（**CudaSCAN**，并行**K-Means**）
  - 并行关联规则（**GPApriori**）


THANKS