2019 年 6 月 第 6 期
Vol. 40 No. 6 2019

基于 CUDA 的 KNN 算法并行化研究

刘端阳,郑江帆,刘 志

(浙江工业大学 计算机科学与技术学院,杭州 310023)
E-mail: ldy@zjut.edu.cn

摘 要: KNN 分类算法在面对大规模数据集时,计算时间将随着数据集的增大而成倍增长,为了提升算法的运算性能,设计了一种基于 CUDA 模型的并行 KNN 算法,即 GS_KNN 算法. 针对 KNN 算法进行了并行化分析,在距离计算阶段采用通用矩阵乘加速,提高了计算速度; 在距离排序阶段根据 k 值的大小提出两种策略,分别是基于 k 次最小值查找的最近邻选择和基于双调排序的最近邻选择; 在决定分类标号阶段采用 CUDA 内部的原子加法操作,从而提高整体性能. 使用 KDDCUP99 数据集对改进算法进行实验,结果表明,在保证实验结果准确性的情况下,改进算法提高了计算速度,与经典的 BF-CUDA 算法相比加速比达到 2.8 倍.

关 键 词: KNN; CUDA; 并行计算; 大数据
中图分类号: TP391 **文献标识码**: A **文 章 编 号**: 1000-1220(2019)06-1197-06

Research of Parallel KNN Algorithm Based on CUDA

LIU Duan-yang, ZHENG Jiang-fan, LIU Zhi
(College of Computer Science and Technology, Zhejiang University of Technology, Hangzhou 310023, China)

Abstract: When the KNN classification algorithm is confronted with large data sets, the computation time will grow exponentially with the increase of the data set, in order to improve the computational performance of the algorithm, a parallel KNN algorithm based on CUDA programming model called GS_KNN was designed. According to the parallel analysis of KNN algorithm, the calculation speed is improved by using the universal matrix multiplication in distance calculation. According to the size of K value, two strategies are proposed in the distance sorting stage: nearest neighbor selection based on K times minimum search and neighbor selection based on bitonic sorting. The internal atomic addition operation of CUDA is adopted in deciding the classification label stage to improve the overall performance. Using the KDDCUP99 data set to experiment with the improved algorithm, the results show that in the case of the accuracy of the experimental results, the improved algorithm improves the calculation speed and is 2.8 times faster than the classical BF-CUDA algorithm.

Key words: KNN; CUDA; parallel computation; big data

1 引 言

分类技术在许多领域得到广泛应用,比如人工智能,网络入侵检测等. 分类技术是把未知分类的样本数据经过一系列模型之后分到若干已知分类的过程[1]. 它先通过已有的训练数据集进行学习和分析,从这些数据的分布中找出分类的规律,逐步优化分类的模型,从而判断新数据属于哪个类别. K 最近邻(KNN: K-Nearest Neighbor)算法[2]是最为流行的分类算法之一,广泛应用于数据分析、图像处理、文本分类[3]等领域. KNN 算法在距离计算阶段需要计算每个测试数据点到所有训练数据点的距离,距离排序阶段需要对距离向量进行排序. 如果任务的测试数据总量为 m,训练数据总量为 n,数据的维度为 d,则距离计算的时间复杂度为 $O(mnd)$,排序阶段的时间复杂度为 $O(mnlogn)$. 随着测试数据集和训练数据集规模的增加,算法的时间成本也会成倍增长,无法满足大数据时代下的快速计算需求. 因此对于计算密集型的 KNN 算法,运用 CUDA(Compute Unified Device Architecture)并行化该算法,能够节省大量计算时间. CUDA 是 NVIDIA 公司推出的通用并行计算架构,它采用 GPU(Graphics Processing Unit)来解决复杂计算问题. CUDA 提供了硬件的直接访问接口,并不依赖传统的图形 API,实现 GPU 的访问,降低了图形处理器的编程难度.

国内外已经有很多基于 CUDA 架构的 KNN 以及其他算法的并行化研究,Jian 等[4]将输入的数据分组,在每个组内进行并行计算,然后得到每个组的 top-K 元素队列,最后合并每个组得到全局的 K 个近邻,该方法降低了距离排序的时间复杂度,但是在距离的计算阶段并行度不高. Garcia 等[5,6]将 KNN 的暴力搜索算法并行移植到 GPU 中执行,在距离计算阶段使用了分块的矩阵乘法,具有良好并行性,但是在距离排序阶段时间复杂度较高. Dashti 等[7]在面对高维度数据时合理使用线程加速距离计算,并且在 GPU 中并行实现了三种距离排序方法,但是当训练集数据量很大时排序阶段的加速效果有限. Masek 等[8]和王裕民等[9]使用多块 GPU 设备搭建了

收稿日期: 2018-07-17 收修改稿日期: 2018-10-31 基金项目: 国家自然科学基金项目(61771430)资助; 浙江省自然科学基金项目(LY16F020033)资助. 作者简介: **刘端阳**,男,1975 年生,博士,副教授,研究方向为数据挖掘和分布式计算; **郑江帆**,男,1993 年生,硕士研究生,研究方向为数据挖掘; **刘 志**(通讯作者),女,1969 年生,博士,教授,CCF 会员,研究方向为大数据和人工智能.

一个 GPU 集群,利用 GPU 的集群优势对 KNN 和卷积神经网络算法进行加速,该方法可以处理更加庞大的数据集,不过提高了硬件成本. Liang 等[10] 提出 cuKNN 算法,采用数据流的方式对输入数据进行计算,利用共享内存加速每个测试数据距离向量的排序,能够批量处理输入数据的同时也提升了性能. Barrientos 等[11] 提出基于 CUDA 的穷举排序算法解决 KNN 问题,在 K 值较高的情况下也有良好的加速比. Mayekar 等[12] 提出一种并行的 KNN 方法快速解决字符识别的问题,并且能保持较高的准确率.

基于上述国内外的研究现状,本文提出了新的 KNN 并行化算法(即 GS_KNN). 新算法充分利用了 CUDA 函数库以及内存体系,将算法的所有计算过程都移植到 GPU 中,发挥其良好的运算能力. 在运算量最大的距离计算阶段,将测试数据集和训练数据集存储在 CUDA 全局内存中,使用 Cublas 函数库的通用矩阵乘计算,提高了运算速度. 在距离排序阶段,针对 k 值的大小提出了两种优化策略,分别是基于 k 次最小值查找的最近邻选择和基于双调排序的最近邻选择,降低了时间复杂度. 在决定分类标号阶段,采用了 CUDA 内部的原子加法操作,GPU 全程参与计算,从而提高了整体性能. 最后通过实验证明,在保证算法鲁棒性的前提下,获得了良好的加速效果.

2 背景知识

2.1 KNN 算法

KNN 算法是目前最为流行的分类算法之一,该算法是一种基于距离的分类,通过计算测试数据与训练数据之间的距离或者相似度来进行分类,同一类别的内部数据之间具有较高的相似度,而不同类别之间数据相似度较低. 现有 $R=\{r_1, r_2,\cdots,r_n\}$ 是一个拥有 n 个点的输入训练集,其中每个点的数据维度为 d,即 $r_j=(r_{j1},r_{j2},\cdots,r_{jd})$,而 $S=(s_1,s_2,\cdots,s_m)$ 是与 R 在同一个维度空间上的输入测试集. KNN 算法的目标是对于每一个 $s_i\in S$,搜索距离 s_i 最近的 k 个 R 数据点,然后根据这 k 个数据点的类别判断 s_i 的类别. 如图 1 所示,表示一个 R 大小为 15,k 等于 4 的查询例子.

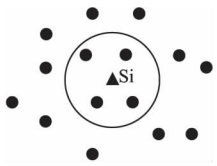

图 1 KNN 查询示例

Fig. 1 KNN query example

求解 KNN 算法最经典的方法就是通过暴力搜索解决,具体算法过程如算法 1 所示.

算法 1. 基于 CPU 的串行 KNN 算法

输入: 包含 n 个点的带有分类标号的训练数据集 R; 包含 m 个点的无分类标号的测试数据集 S; 最近邻个数 k.

输出: m 个测试数据的分类.

1) 计算测试集 S 中点 s_i 到所有 r_j 的距离.

2) 对步骤 1 得到的 n 个距离进行升序排序.

3) 根据排序结果,选择前 k 个最小距离,得到距离点 s_i 最近的 k 个训练数据.

4) 统计这 k 个训练数据的分类标号,出现次数最多的那一个标号即为点 s_i 的分类.

5) 循环步骤 1 至 4,一直到 m 个测试数据的分类计算完毕.

由于数据维度为 d,步骤 1(即距离计算阶段) 的时间复杂度为 O(nmd),步骤 2(即排序阶段) 的时间复杂度为 O($mnlogn$). 步骤 4(即决定分类标号阶段) 的时间复杂度为 O(mk),此值较小,可忽略不计. 当训练数据集 n 很大时或者面对高维度数据时,运用 CUDA 计算架构并行化 KNN 算法通常可以取得很好的加速效率.

2.2 CUDA 编程模型

CUDA 是 NVIDIA 公司推出的一款通用并行计算架构,CUDA 架构改变了传统的 GPU 编程方式,它不需要将程序任务转换为 GPU 图形处理任务,降低了开发难度. CUDA 架构主要包括计算核心和存储器体系. 每个 GPU 计算核心由多个流多处理器(stream multiprocessor,SM) 组成,每个 SM 由多个标量处理器(stream processor,SP) 组成. 同时,GPU 提供了可供线程访问的多级存储器. 每个线程具有私有的本地存储器,每个线程块则有共享存储器,它对块内所有线程共享数据,并且它的生命周期与块相同,读写速度快. 常量存储器和纹理存储器是只读存储器,可以针对不同的用途进行优化,全局存储器则用于接收从 CPU 端传入的数据.

在 CUDA 编程模型中,GPU 被认为是能够并行执行大量线程的协处理器. 一个简单的源程序包括运行在 CPU 上的主机端代码和运行在 GPU 上的内核(kernel) 代码. Kernel 函数通过__global__类型限定符定义,并通过 *Kernel*<<<*blocks*, *threads*>>>(*parameters*) 调用,*blocks* 代表线程块数量,*threads* 代表每个线程块中包含的线程数,*parameters* 代表函数调用时的参数列表. 计算密集型和数据并行的内核函数代码运行在 GPU 上. 所有线程被组成许多线程块,每个线程块中的线程会在同一个 SM 中并行执行. 同一个线程块的线程可以通过块内的共享内存共享数据,并且可以通过__*syncthreads*() 函数同步数据. 在 CUDA 编程模型中另一个重要的概念就是线程束,它由 32 个并行的线程组成并且是每个 SM 处理器的基本调度单元. 当一个线程束执行结束之后,SM 处理器会接着调度另一个线程束继续执行,线程束一次执行同一条指令,所以当线程束中的 32 条线程有相同的执行路径时可以达到最大的加速效率. 当遇到条件分支时线程束中的线程产生不同的执行路径,这将会增加该线程束执行指令的总时间. 为了让资源不浪费,即 warp 内的线程被充分利用,线程的数量经常设置为 32 的整数倍. 只有合理地分配线程块以及线程的数量,才可以使 GPU 的利用率最大化.

3 基于 CUDA 的 KNN 算法并行优化

分析串行 KNN 算法的流程可知,算法的时间开销大部分都在距离计算阶段和排序阶段. 因此本文着重在这两个步骤上进行并行优化,并且在 GPU 中进行所有计算. 以下四个小节详细介绍了 GS_KNN 算法的优化过程.

3.1 基于 Cublas 的距离矩阵计算

Cublas Library 是基于 NVIDIA 的通用函数库,它实现了 Blas(基本线性代数子程序),它允许用户访问 GPU 中的计算单元,在使用时在 GPU 中分配所需的矩阵向量空间并填充数据,然后调用其中的函数可以加速向量和矩阵等线性运算. 对于 KNN 算法中庞大的距离计算量,本文利用 Cublas 库中的矩阵乘法函数 *cublasSgemm*() 来加速距离计算,该函数能实现向量与矩阵,矩阵与矩阵之间的运算,并且拥有很好的加速比.

cublasSgemm( ) 函数调用时有许多参数,调用方法为 *cublasSgemm*(*handle*, *transa*, *transb*, *m*, *n*, *k*, * *alpha*, * *A*, *lda*, * *B*, *ldb*, * *beta*, * *C*, *ldc*) ,在 Cublas 里面所有的矩阵都是按照列优先方式存储,*lda*,*ldb*,*ldc* 代表矩阵 *A*,*B*,*C* 的行数,矩阵 *A* 的维度为 *m* * *k*,*B* 的维度为 *k* * *n*,*C* 的维度为 *m* * *n*,*transa* 和 *transb* 代表矩阵 *A* 和 *B* 是否转置,函数实现的矩阵运算为:

$$C = alpha * OP(A) * OP(B) + beta * C \tag{1}$$

为了满足公式(1) ,GS_KNN 算法的距离度量选择欧几里得距离的平方. 这样,GS_KNN 算法的距离计算可以转化为:

$$|x-y|^2 = x^2 + y^2 - 2x * y = alpha(x * y) + beta(x^2 + y^2) \tag{2}$$

其中,*alpha* 取值为-2,*beta* 取值为 1. 在设计 CUDA 核函数时,如果有 *m* 个测试数据点和 *n* 个训练数据点,数据维度为 *d*,则矩阵 *A* 为 *x*,代表 *m* * *d* 维的测试数据点矩阵,矩阵 *B* 为 *y*,代表 *n* * *d* 维的训练数据点矩阵,由于 *x* 和 *y* 都是按照列优先方式存储,实际上在函数运算时矩阵 A 的维度为 *d* * *m*,矩阵 B 的维度为 *d* * *n*,为了得到 *m* * *n* 维的距离结果矩阵,需要将矩阵 A 进行转置,矩阵 B 不需要转置,所以将 *transa* 这个参数的值设为 CUBLAS_OP_T 代表转置,*transb* 这个参数的值设为 CUBLAS_OP_N 代表不需转置. 由于矩阵 A 的行数为 *m*,矩阵 B 的列数为 *n*,矩阵 A 的列数为 *d*,所以分别将函数参数列表中的 *m* 赋值为 *m*,*n* 赋值为 *n*,*k* 赋值为 *d*. 在调用函数之前先要计算矩阵 C 也就是 $x^2 + y^2$ 的值,对于 x^2 算法开启 *m* 个线程,若矩阵 A 在 CUDA 内存中的地址为 *p*,线程编号为 *threadId*,每个线程计算地址在 *p* + *d* * *threadId* 至 *p* + *d* * (*threadId* + 1) 上的数据平方并求和,即一个测试数据点在 *d* 个维度上的平方和. 对于 y^2 开启 *n* 个线程,若矩阵 B 在 CUDA 内存中的地址为 *q*,线程编号为 *threadId*,每个线程计算地址在 *q* + *d* * *threadId* 至 *q* + *d* * (*threadId* + 1) 上的数据平方并求和,即一个训练数据点在 *d* 个维度上的平方和. 得到以上两个结果之后,在 CUDA 中继续开启 *n* 个线程,每个线程将一个训练数据点的平方和分别与 *m* 个测试数据点的平方和相加从而得到维度为 *m* 的向量,则 *n* 个线程计算得到 *m* * *n* 维的矩阵,即表达式为 $x^2 + y^2$ 的输入的矩阵 C. 最后将矩阵 A,B 和 C 在 CUDA 全局内存中的地址指针传入 *cublasSgemm* ( ) 函数,函数返回 *m* * *n* 维的距离结果矩阵,每一行向量代表一个测试数据点分别与 *n* 个训练数据点的距离.

3.2　基于 *k* 次最小值查找的最近邻选择

通过计算获得每个测试数据点的距离向量后,需要进行排序. 虽然有不少研究把排序算法移植到 CUDA 中执行,如 Sintorn 等[13] 和 Satish 等[14] 的研究,但算法的时间复杂度为 O(*nlogn*) ,并行度不高. 通过分析 KNN 算法可知,参数 *k* 往往远小于训练集数据量 *n*,所以对于每一个测试数据来说,没有必要将它到训练集的所有距离都进行排序,只需要获得前 *k* 个最小距离即可. 冒泡排序算法和选择排序算法,每次排序都可以选出一个最小值,*k* 次排序就可以获得 *k* 个最小距离. 虽然这两种算法的时间复杂度只有 O(*kn*) ,但是它们却无法并行执行,因为每次排序都要依赖前面元素的排序结果. 因此,冒泡排序算法和选择排序算法都不适合 CUDA 平台.

本节提出了一个基于 *k* 次最小值查找的最近邻选择方法,适合 GPU 的多线程环境,并且是一个基于比较网络的模型,可以在同一时刻执行多个不相关的比较操作. 算法定义的一个比较器如图 2 所示,比较器的输入为两个任意未经过排序的数据,经过比较器之后两个输出被排序,并且上端线路的元素大于或者等于下端线路的元素,一个比较器所执行的时间为一个单位时间,时间复杂度为 O(1) .

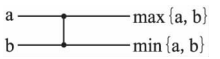

图 2　K 次最小值查找的比较器

Fig. 2　Comparator of K-min search

对于一个测试数据,经过距离计算之后得到它对 *n* 个训练数据的距离向量{ $d_0, d_1, \cdots d_{n-2}, d_{n-1}$} ,需要从该距离向量中选择出 *k*(*k*≤*n*) 个最小距离,根据排序网络的比较器需要两个输入,算法将该向量以 $d_{n/2}$ 处的位置一分为二,然后 d_0 和 $d_{0+\lfloor n/2 \rfloor}$(符号⌊ ⌋表示向下取整) 输入比较器做判断之后得到 max{ $d_0, d_{0+\lfloor n/2 \rfloor}$} 和 min{ $d_0, d_{0+\lfloor n/2 \rfloor}$} . 与此同时,$d_1$ 和 $d_{1+\lfloor n/2 \rfloor}$ 做比较得到 max{ $d_1, d_{1+\lfloor n/2 \rfloor}$} 和 min{ $d_1, d_{1+\lfloor n/2 \rfloor}$} ,$d_{n/2}$ 之前的一个元素 $d_{\lfloor n/2 \rfloor - 1}$ 和 $d_{\lfloor n/2 \rfloor - 1+\lfloor n/2 \rfloor}$ 比较得到 max{ $d_{\lfloor n/2 \rfloor - 1}, d_{\lfloor n/2 \rfloor - 1+\lfloor n/2 \rfloor}$} 和 min{ $d_{\lfloor n/2 \rfloor - 1}, d_{\lfloor n/2 \rfloor - 1+\lfloor n/2 \rfloor}$} ,当 *n* 为奇数时,最后一个数 d_{n-1} 无须经过比较自动进入下一轮迭代. 这些两两元素之间的比较是互不相关的,符合 CUDA 并行执行的特性. 在经过第一轮比较器之后会产生两个子向量 l_1 和 l_2,如果 *n* 为偶数,则向量如下所示:

$$l_1 = \{\max\{d_0, d_{0+\lfloor n/2 \rfloor}\}, \max\{d_1, d_{1+\lfloor n/2 \rfloor}\}, \cdots \max\{d_{\lfloor n/2 \rfloor - 1}, d_{\lfloor n/2 \rfloor - 1+\lfloor n/2 \rfloor}\}\} \tag{3}$$

$$l_2 = \{\min\{d_0, d_{0+\lfloor n/2 \rfloor}\}, \min\{d_1, d_{1+\lfloor n/2 \rfloor}\}, \cdots \min\{d_{\lfloor n/2 \rfloor - 1}, d_{\lfloor n/2 \rfloor - 1+\lfloor n/2 \rfloor}\}\} \tag{4}$$

如果 *n* 为奇数,则子向量 l_2 为:

$$l_2 = \{\min\{d_0, d_{0+\lfloor n/2 \rfloor}\}, \min\{d_1, d_{1+\lfloor n/2 \rfloor}\}, \cdots, \min\{d_{\lfloor n/2 \rfloor - 1}, d_{\lfloor n/2 \rfloor - 1+\lfloor n/2 \rfloor}\}, \min\{d_{n-1}\}\} \tag{5}$$

产生两个子向量 l_1 和 l_2 需要经过⌈*n*/2⌉ (符号⌈ ⌉表示向上取整) 次比较器的使用,在 CUDA 中这⌈*n*/2⌉ 次比较可以同时执行,对于要寻找到本轮迭代中的最小值来说,它必定存在于子向量 l_2 中,循环迭代上一步的过程,继续将向量 l_2 一分为二输入比较器,直至 l_2 中只存在一个元素时结束迭代,此时最后的这个元素就是本轮迭代中得到的最小值. 经过上述的第一轮最小值选择操作之后,距离向量变为{ $e_0, e_1, \cdots e_{n-2}, e_{n-1}$} ,并且 e_{n-1} 这个元素是该向量中的最小值,也是算法选择出的第一个最近邻元素,然后继续对{ $e_0, e_1, \cdots e_{n-2}$} 向量做如上操作,就能得到第二个最近邻元素. 经过 *k* 次最小值查找之后,就可以查找出初始距离向量中的 *k* 个最近邻元素,由于每轮迭代中可以在 CUDA 中开启多线程并行查找,将原本时间复杂度为 O(*n*) 的比较阶段降为 O(1) ,所以一次最小值查找的时间复杂度就等于迭代次数 *O*(⌈$\log_2 n$⌉) ,那么整体 *k* 次最小值查找的时间复杂度就是 *O*(*k*⌈$\log_2 n$⌉) ,相比于其它排序算法拥有更低的时间复杂度.

综上所述,基于 *k* 次最小值查找的最近邻选择算法的详细步骤如下:

1) 将全局存储器中的 *m* * *n* 的距离矩阵拷贝至共享内存.

2) 当 *n* 的值小于 1024 时,在 CUDA 中开启 *m* 个线程块,每一个线程块负责一个测试数据点距离向量的排序,每个线程块内部分配⌈*n*/2⌉ 个线程,每个线程负责两个对应位置上

数据的比较器选择,留下值较小的数据并更新到共享内存上.

3) 每一轮迭代通过 *syncthreads*( ) 函数同步块内的所有线程,同步完成之后继续对子向量 l_2 进行比较器选择,在完成 $O(\lceil \log_2 n \rceil)$ 次迭代之后,每个线程块零号线程对应位置上的共享内存存储的数据就是本轮最小值.

4) 经过 k 次选择就得到属于该测试数据的 k 个最近邻.

当 n 的值较大时,由于 CUDA 架构的硬件限制,则每个向量需要使用 $\lceil n/1024 \rceil$ 个线程块负责计算,通过找出这些线程块输出数据中的最小值从而得到最近邻数据.

3.3 基于双调排序的最近邻选择

通知分析可知: 基于 k 次最小值查找的最近邻选择算法,当 k 值比较大时,经过 k 次查找所需要的时间也越大. 从时间复杂度上分析,如果 k 值接近于训练集数据量 n 时,该方法的时间复杂度就变为 $O(n\log n)$,与快速排序的时间复杂度相同,相当于将距离向量完全排序,因此它不适应 k 值较大的情况. 本节介绍了一种基于 CUDA 的双调排序算法,它同样能够利用多线程特性来并行执行,适合于 k 值比较大的排序场合.

本算法定义一个序列 $s=(a_0,a_1,\cdots,a_{n-1})$,如果满足条件 $a_0 \leqslant a_1 \leqslant \cdots \leqslant a_{n/2-1}$ 并且 $a_{n/2} \geqslant a_{n/2+1} \geqslant \cdots \geqslant a_{n-1}$,那么称序列 s 就是一个双调序列. 算法将任意一个长度为 n(n 为偶数) 的双调序列 s 分为相等长度的两段 s_1 和 s_2 ,将 s_1 中的元素和 s_2 中的元素一一比较,即 $a[i]$ 和 $a[i+n/2]$ $(i<n/2)$ 比较,将两者中的较大者放入 max 序列,较小者放入 min 序列,那么得到的 max 序列和 min 序列仍然是双调序列,并且 max 序列中任意一个元素不小于 min 序列中的任意一个元素. 算法将双调序列划分为两个子双调序列,然后对每个子双调序列递归划分,直到得到的子序列的长度为 1 时停止,此时输出的序列就是一个单调递增或者单调递减的序列. 对于每一个测试数据的距离向量,刚开始的时候并不是一个双调序列,要通过双调排序得到结果必须先将无序距离序列转换成一个双调序列. 转换过程如下: 将两个相邻的并且单调性相反的单调序列看成一个双调序列,将这两个单调序列合并成一个新的双调序列,然后对它做双调排序. 这样只要每次两个相邻长度为 n 的序列单调性相反,就可以合并成一个长度为 $2n$ 的双调序列,然后对其进行双调排序变成有序,直至最后合并成的双调序列长度为输入序列的长度时,将其进行排序即可得到结果. 构建双调序列示意图如图 3 所示,图中圆圈内加号代表按递增排序,减号代表按照递减排序.

经过距离计算得到的 $m*\ n$ 的距离矩阵,使用基于双调排序的最近邻选择算法的详细步骤如下:

1) 对于每一个测试数据对应的距离向量 l,其长度为 n,算法中双调排序输入的序列长度为 2^a($a>0$) ,当 n 的大小不足 2^a 时,补足最少个数的元素使 $n=2^a$,补足的元素统一为能够表示的最大数.

2) 将数据拷贝至共享内存,同时开启若干个线程块负责距离序列的双调排序,块内总共进行 a 轮迭代,前 $a-1$ 轮进行相邻两个单调性相反的序列合并,并分别按相反单调性递归进行双调排序.

3) 第 a 轮时合并前面两个长度为 $n/2$ 的单调序列,开启 $n/2$ 个线程做两两元素的比较,按此方式迭代 a 轮,每轮都会开启 $n/2$ 个线程参与计算,直至最后一轮迭代时是长度为 2 的序列相比较,即可得出最后单调递增的距离序列.

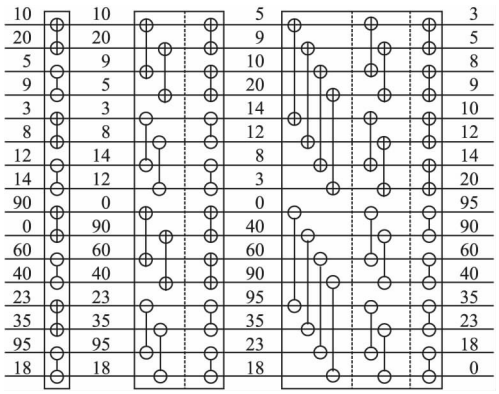

图 3 构建双调序列示意图

Fig. 3 Diagram of a bitonic sequence

基于双调排序的最近邻选择算法的时间复杂度为 $O((\log n)^2)$,所以当 k 较大时,该方法的时间复杂度是优于基于 k 次最小值查找的最近邻选择算法.

3.4 基于 CUDA 的决定分类标号

基于 CUDA 的决定分类标号的具体步骤如下: 在 CUDA 中开启 m 个线程块,每个线程块负责一个测试数据分类标号的统计,在共享内存上开辟一个数组,数组长度为训练数据集的类别个数. 块内开启 k 个线程,每个线程统计自己对应位置上数据的类别标号,采用 CUDA 内部的原子加法操作统计,即 *atomicAdd*( ) 操作,在类别数组上取数据并执行加法操作. 最后同步线程块内线程,得到最终记录数组,数组当中数字最大的类别即为该测试集最终的分类.

3.5 GS_KNN 算法的详细步骤

综合前面各节所述,GS_KNN 算法的详细步骤如算法 2 所示,算法流程如图 4 所示.

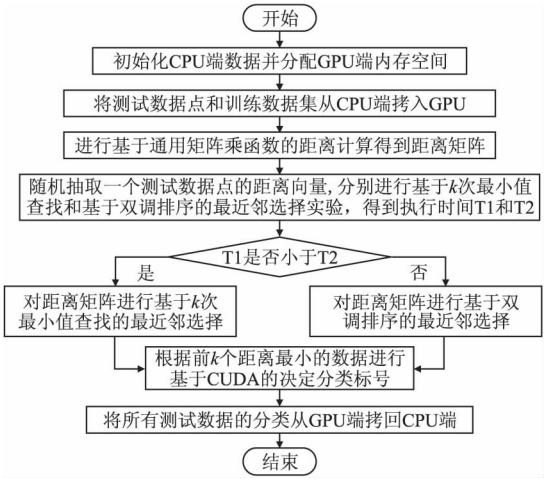

图 4 GS_KNN 算法流程图

Fig. 4 Flow chart of GS_KNN algorithm

算法 2. 基于 CUDA 的 GS_KNN 算法

输入: m 个点的测试数据集,n 个点的训练数据集,以及最近邻个数 k.

输出: m 个测试数据的分类结果

1) 在 CPU 主机端初始化测试数据集,训练数据集和最近邻个数 k 的内存空间.
2) 在 GPU 设备端分配测试数据集,训练数据集以及最近邻个数 k 的内存空间.
3) 将主机端数据拷贝至设备端.
4) 利用 CUDA 调用基于 Cublas 通用矩阵乘的距离计算内核函数,得到所有测试数据集数据到训练数据集的距离矩阵.
5) 随机抽取一个测试数据的距离向量,分别进行基于 k 次最小值查找和基于双调排序的最近邻选择排序算法的实验,得到时间 T1 和 T2.
6) 如果 T1 小于 T2,则选择基于 k 次最小值查找的最近邻选择算法,进行距离排序. 否则,选择基于双调排序的最近邻选择算法来排序.
7) 进行基于 CUDA 的决定分类标号,得到每个测试数据的分类结果.
8) 将最后的分类结果从 GPU 设备端返回拷贝至 CPU 主机端并输出.

4　实验与分析

4.1　实验环境

本文所使用的实验平台为 Intel Core i5-4590 CPU,主频 3.30GHz,内存 8GB,GPU 为 NVIDIA GTX750 Ti,计算能力为 5.0,拥有 5 个流多处理器,640 个 SP. 软件环境为 Windows 10,Visual studio 2010 和 CUDA7.0.

4.2　实验结果

本文实验所使用的第一个数据集为程序生成的模拟数据,可以指定测试集大小、训练集大小、每个样本的类别以及它的属性维度. 第二个数据集为 KDDCUP1999 提供的网络入侵检测数据集,该数据集包含大量真实的网络入侵数据,数据集中每个元素包含 41 个维度的属性,每个维度属性为浮点类型的数据,总共有 24 种攻击类型. 由于每个维度上的度量单位不同,需要对输入数据进行标准化和归一化,本文使用 Wang 等[15] 的方法进行数据的预处理.

4.2.1　固定 m,k,改变 n,d 对加速的影响

本实验采用模拟数据,设定测试集的大小 m 为 1200,训练集 n 的大小为 2^{10} 到 2^{15},这样可以方便线程的访问,测试集与训练集的维度 d 相同,k 的值设为 25. 由于 k 值较小,所以经过评估后算法采用基于 k 次最小值查找的最近邻选择. 选择了两种算法来与 GS_KNN 算法作对比. 第一种算法选用了基于 CPU 的超快近似最近邻搜索算法库,即 ANN-C++[16],另一种选用了 BF-CUDA 算法. 改变 n 和 d 的大小,实验结果如表 1 所示,可以看出 GS_KNN 算法在相同数据集上的性能上要优于 ANN-C++算法和 BF-CUDA 算法,并且随着训练集大小和维度的增加,GS_KNN 算法相较于这两个算法的加速比也越来越高,当 n 等于 2^{15},d 等于 256 时,GS_KNN 算法相对于 ANN-C++算法的加速比为 145,相对于 BF-CUDA 算法的加速比为 2.8.

4.2.2　固定 m,n,d,改变 k 对排序的影响

实验(1)是在固定 $k=25$ 的情况下进行的,为了比较 k 值的大小对算法距离排序阶段的影响,选取测试数据集大小为 1200,测试数据集为 16384,数据维度为 32,第一次实验中 GS_KNN算法的距离排序采用基于 k 次最小值查找的最近邻选择,第二次采用基于双调排序的最近邻选择,然后每次实验都不断增加 k 的值,观察记录距离排序阶段的执行时间,实验结果如图 5 所示,可以看出当 k 的值为 46 时,基于双调排序和基于 k 次最小值查找的距离排序所用时间相同,当 k 的值小于 46 时,k 次最小值查找所需的时间要少于双调排序. 随着 k 的不断增加,双调排序所花费的时间不变,因为不管 k 值如何变化,算法都会得到一个完整的排序结果,而 k 次最小值查找的时间开销呈线性增长趋势,因为算法的迭代次数会随着 k 的增加而线性增加,此时双调排序的优势会越来越明显.

表 1　改变 n 和 d 时各算法运行时间比较(单位: s)
Table 1　Comparison of running time of each algorithm when changing n and d (unit: s)

d	算法	$n=1024$	$n=2048$	$n=4096$	$n=8192$	$n=16384$	$n=32768$
16	ANN-C++	0.172	2.025	4.496	17.306	40.748	92.761
	BF-CUDA	0.072	0.105	0.153	0.452	1.406	2.791
	GS_KNN	0.069	0.081	0.102	0.251	0.853	2.329
32	ANN-C++	0.309	2.523	7.134	26.319	99.735	197.592
	BF-CUDA	0.075	0.115	0.191	0.509	2.176	4.033
	GS_KNN	0.070	0.087	0.123	0.283	0.989	2.804
64	ANN-C++	0.471	2.784	8.968	30.132	130.47	347.166
	BF-CUDA	0.080	0.132	0.236	0.599	2.621	6.210
	GS_KNN	0.071	0.096	0.152	0.324	1.165	3.426
128	ANN-C++	0.687	3.131	16.142	40.375	212.643	638.049
	BF-CUDA	0.086	0.141	0.269	0.808	4.076	11.924
	GS_KNN	0.073	0.101	0.168	0.425	1.772	5.317
256	ANN-C++	0.951	3.584	11.834	54.624	327.063	1167.886
	BF-CUDA	0.096	0.163	0.327	1.127	6.298	22.479
	GS_KNN	0.076	0.112	0.194	0.569	2.703	8.031

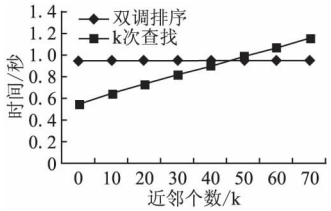

图 5　改变 k 对距离排序时间的影响
Fig. 5　Effect of changing k on distance sorting time

4.2.3　算法准确性验证

从 KDDCUP99 数据集中选取 5000 条数据,其中 4900 条正常数据,100 条包含攻击的入侵数据作为训练集,再选 500 条包含正常数据以及训练集中未知的入侵数据集作为测试集,分别运用经典的基于 CPU 的 K 最近邻入侵检测算法和 GS_KNN 算法,进行 10 次实验取检测率和误报率的平均值,每次调整 k 值的大小,统计结果如表 2 所示.

表 2　入侵检测鲁棒性检测结果
Table 2　Intrusion detection robustness test results

攻击类别	CPU 检测率	GS_KNN 检测率	CPU 误报率	GS_KNN 误报率
Dos	72.3%	72.4%	2.49%	2.49%
Probe	89.6%	89.8%	2.12%	2.11%
R2L	78.7%	78.7%	2.34%	2.34%
U2R	64.9%	64.8%	3.01%	3.03%

从结果来看,GS_KNN 算法在检测 4 类攻击上与标准 CPU 版本的 K 最近邻算法相比,检测率和误报率几乎相同. 造成略微差异的主要是由于在距离排序阶段,离测试数据距离完全相等而类别标号不同的训练数据被 k 的不同切分所导致.

5 结 论

本文在经典串行 KNN 算法的基础上,利用 GPU 的并行计算能力,提出了一种基于 CUDA 的并行 KNN 算法,即 GS_KNN 算法,该算法能有效处理大规模数据情况下的数据分类问题.算法利用矩阵乘的思想加速了距离矩阵的运算,提出基于 k 次最小值查找和双调排序两种策略优化距离排序,并在 CUDA 中完成分类标号的统计.它与经典的 BF-CUDA 算法相比获得 2.8 倍的加速比.然而本文在 GPU 和 CPU 之间的数据传输以及多 GPU 协同合作还有待研究,因此下一步尽可能优化数据的存储,利用 GPU 集群提升算法的性能.

References:

[1] Mitchell T M. Machine learning [M]. Beijing: China Machine Press,2008.

[2] Keller J M,Gray M R,Givens J A. A fuzzy K-nearest neighbor algorithm [J]. IEEE Transactions on Systems Man & Cybernetics, 2012,15(4):580-585.

[3] Yang Shuai-hua,Zhang Qing-hua. Research on KNN text classification algorithm based on approximate set of rough sets [J]. Journal of Chinese Computer Systems,2017,38(10):2192-2196.

[4] Jian L,Wang C,Liu Y,et al. Parallel data mining techniques on graphics processing unit with compute unified device architecture (CUDA) [J]. Journal of Supercomputing,2013,64(3):942-967.

[5] Garcia V,Debreuve E,Barlaud M. Fast k nearest neighbor search using GPU [C]//Proceedings of Computer Society Conference on Computer Vision and Pattern Recognition Workshops,IEEE,2008: 1-6.

[6] Garcia V,Nielsen F. Searching high-dimensional neighbours: CPU-based tailored4 data-structures versus GPU-based brute-force method [C]//Proceedings of International Conference on Computer Vision/Computer Graphics Collaboration Techniques,Springer-Verlag,2009:425-436.

[7] Dashti A,Komarov I,D´Souza R M. Efficient computation of k-nearest neighbour graphs for large high-dimensional data sets on GPU clusters [J]. PLOS ONE,2013,8(9):e74113.

[8] Masek J,Burget R,Karasek J,et al. Multi-GPU implementation of k-nearest neighbor algorithm [C]//Proceedings of International Conference on Telecommunications and Signal Processing,IEEE, 2015:764-767.

[9] Wang Yu-min,Gu Nai-jie,Zhang Xiao-ci. Parallel algorithm of convolutional neural network in multi-GPU environment [J]. Journal of Chinese Computer Systems,2017,38(3):536-539.

[10] Liang S,Wang C,Liu Y,et al. CUKNN: a parallel implementation of k-nearest neighbor on CUDA-enabled GPU [C]//Proceedings of Youth Conference on Information,Computing and Telecommunication,IEEE,2009:415-418.

[11] Barrientos R J,Millaguir F,Sánchez J L,et al. GPU-based exhaustive algorithms processing kNN queries [J]. Journal of Supercomputing,2017,73(1):1-24.

[12] Mayekar M M N,Kuwelkar M S. Implementation of machine learning algorithm for character recognition on GPU [C]//Proceedings of International Conference on Computing Methodologies and Communication,2017:470-474.

[13] Sintorn E,Assarsson U. Fast parallel GPU-sorting using a hybrid algorithm [J]. Journal of Parallel and Distributed Computing,2008, 68(10):1381-1388.

[14] Satish N,Harris M,Garland M. Designing efficient sorting algorithms for manycore GPUs [C]//Proceedings of IEEE International Symposium on Parallel and Distributed Processing(SPDP),IEEE, 2009:1-10.

[15] Wang Q,Megalooikonomou V. A clustering algorithm for intrusion detection [C]//Proceedings of SPIE-The International Society for Optical Engineering,2005:31-38.

[16] Arya S,Mount D M,Netanyahu N S,et al. An optimal algorithm for approximate nearest neighbor searching fixed dimensions [J]. Journal of the ACM,1998,45(6):891-923.

附中文参考文献:

[1] 米歇尔.机器学习 [M].北京:机械工业出版社,2008.

[3] 杨帅华,张清华.粗糙集近似集的 KNN 文本分类算法研究 [J].小型微型计算机系统,2017,38(10):2192-2196.

[9] 王裕民,顾乃杰,张孝慈.多 GPU 环境下的卷积神经网络并行算法 [J].小型微型计算机系统,2017,38(3):536-539.