苏州大学学位论文使用授权声明

本人完全了解苏州大学关于收集、保存和使用学位论文的规定，即：学位论文著作权归属苏州大学。本学位论文电子文档的内容和纸质论文的内容相一致。苏州大学有权向国家图书馆、中国社科院文献信息情报中心、中国科学技术信息研究所（含万方数据电子出版社）、中国学术期刊（光盘版）电子杂志社送交本学位论文的复印件和电子文档，允许论文被查阅和借阅，可以采用影印、缩印或其他复制手段保存和汇编学位论文，可以将学位论文的全部或部分内容编入有关数据库进行检索。

涉密论文□

本学位论文属　　在____年___月解密后适用本规定。

非涉密论文□

论文作者签名：_____日　期：2010.6.14

导 师 签 名：_____日　期：2010.6.14

通用GPU计算在分类算法中的研究与应用

摘　要

人们对计算能力的需求是永无止境的。在传统的单核CPU发展遇到瓶颈而朝着多核方向发展的同时，图形处理器以其强大的运算能力，逐渐进入了人们的视野。通用GPU计算成为一个热点的研究方向。然而当前的通用GPU计算所应用的领域，仍以图形图像处理和三维场景模拟为主。

本文关注于将GPU应用在数据挖掘和机器学习领域的分类算法中。在对GPU的体系结构和性能特点进行研究的基础上，提出基于GPU的算法设计原则，并以部分典型算法为例，进行了实现和检验。具体工作包括以下几方面：

首先，探讨将通用GPU计算应用于分类算法的可行性。经分析论证，在大规模和高维度数据集的分类上，应用基于CUDA的GPU运算，如能满足相应的设计约束，则能取到较好效果。

其次，提出了一种基于GPU的K最近邻分类算法（GSNN算法）。该算法在距离计算阶段提出了一种分块策略，在最近邻选择阶段采用了一种评估选择的方法。这两个方法都充分利用了GPU的结构特点，发挥了其运算能力，得到很高的加速比。

第三，针对支持向量机算法复杂度较高，难以应用于大样本分类的问题，提出了GMP-CSVC算法。算法基于序贯最小优化方法，在运算过程中，发挥了GPU的优势，并尽量减少程序分支，取得了较好的效果。

最后，针对SVM分类器的参数寻优过程，提出了基于GPU的GMP-nuSVC算法。算法在参数寻优的训练阶段采用了基于ν-SVM的改进算法、核矩阵缓存调度方案，在标号判定阶段采用了分块计算方法，从而实现对原始SVM分类算法改进的基础上，大幅度提升了性能。

本文成功地将图形处理器应用于数据挖掘和机器学习领域的分类算法上，扩大了算法的可计算边界，大幅度减少了训练时间，对分类算法的应用和通用GPU计算的算法研究，都有一定的参考价值。

关键词： 通用GPU计算、分类算法、CUDA、并行算法

作　　者： 邝泉声

指导教师： 赵　雷

Study on General Purpose GPU Computing in Classification Algorithms

Abstract

Driven by the rapidly growing demand for 3D rendering and graphics processing, the GPU (Graphics Processing Unit) has developed into a kind of micro-processor with tremendous computational horsepower and highly thread parallelism. GPGPU (General Purpose GPU) is becoming a research focus. This paper concentrates on applying GPU to classification algorithms. Based on the study of the GPU architecture, the paper presents principles and methods in designing GPU algorithms using CUDA (Computing Unified Device Architecture). Some typical algorithms are also used to verify these principles and methods. The main research work of this paper is as follows.

Firstly, the paper discusses the feasibility of GPU in classification algorithms. The analysis and appraisal show that better results could be achieved on the condition that some design constraints using CUDA platform is met.

Secondly, the paper proposes GSNN algorithm, which is a GPU-based segmentation nearest neighbor classification method. The algorithm uses a segmentation strategy in distances calculation and an execution assessing method in nearest neighbor selection.

Thirdly, a GPU based massively data parallel C-SVM classification (GMP-CSVC) algorithm is presented to reduce the training time of SVM.

Finally, based on the improvement algorithm of ν-SVM, GMP-nuSVC is proposed to solve the performance issue in cross-validation and parameter selection of SVM. A cache scheduling scheme and a tiling method of calculation are used in the algorithm.

The paper successfully applies GPU to the field of classification algorithms by extending the border of calculation and reducing the training time. It has certain practical significance in the application of classification methods and the research of GPGPU algorithms.

Keywords: GPGPU, Classification, CUDA, Parallel Algorithm

Written by Kuang Quansheng

Supervised by Zhao Lei

目　录

第一章 绪论

1.1 课题背景

当前我们所面对的很多问题，例如国民经济和国防建设领域的核爆炸模拟、卫星数据处理、地理信息和地震预报等，科技前沿领域所涉及的数据仓库和数据挖掘方法、生物信息学、分子动力学模拟等，甚至很多日常应用，诸如视频编码解码和虚拟现实技术，都需要巨大的运算能力。随着科技的发展这种需求也在不断扩大。

为了实现较高的性能，仅靠改进电路工艺，提高CPU的处理速度是远远不够的。高性能计算最终采取的是并行的方式，将很多处理器或计算资源组织成统一的计算系统和环境，以满足对强大运算能力的需求。高性能计算机的结构，包括专门为某些科学计算而设计的超级计算机，也包括兼顾科学计算、事务处理、数据库和网络应用的超级服务器。高性能计算也在朝着网络化的方向发展，分布式计算和网格计算都是当前的应用热点。然而图形硬件的高速发展，使人们逐渐意识到它强大的运算能力，也应当在高性能计算领域有所作为 [1]。

GPU（Graphics Processing Unit，图像处理单元）在现有的个人计算机体系结构中主要负责图形图像的显示生成以及三维场景的渲染处理，是计算机显示卡中最主要的处理芯片，也是个人微型计算机中理论上运算性能最为强劲的处理部件。这是由于GPU和CPU有不同的功能定位，采用不同的设计理念所导致的结果。CPU在发展过程中关注于将浮点运算器、缓存、内存控制器、局部总线控制器甚至显示单元等最初不属于CPU的部件集成进来以大幅度提高系统整体性能，并采用超标量、超长流水线、分支预测和缓存预取等技术不断提升常规串行代码的执行速度。而相比之下，GPU原本专为图形图像的应用而设计。图形图像和三维场景在计算机中均可表示为高阶的二维或三维点阵数组，数据量大且易于划分，故设计成大量相对简单的处理单元并行计算处理，最容易获得性能上的提升。因而在同样的芯片制造工艺水平下，GPU在运算性能指标上得以胜过同时期的CPU。

为了充分利用GPU强劲的运算能力，近年来人们一直探索扩展GPU的应用范围，将GPU的应用扩展到图形图像以外的领域 [2]。通用GPU计算成为当前一个新兴的研究方向。

各种分类算法在机器学习和数据挖掘等很多领域都有广泛应用，但算法复杂度

普遍较高。例如支持向量机(Support Vector Machine) [3,4]，在解决非线性数据及高维模式识别中表现出许多特有的优势，但仅能适用于小样本的学习和分类。*K*最近邻分类(*K*-Nearest Neighbor) [5]、贝叶斯分类(Bayesian Classifier) [6]、决策树分类(Decision Tree) [7] 等方法虽然算法复杂度不是很高，但在很多应用中，数据集或数据维度非常大，性能问题也成为制约应用的瓶颈。并行计算是解决分类器在大样本条件下训练和分类问题的一种思路。

本课题关注于将图形处理器应用于机器学习和数据挖掘领域的各种分类算法中。在GPU特殊的体系结构下，应用共享存储的并行编程模型，研究合适的并行算法，以使用GPU执行分类算法，大幅度减少运算时间，扩大现有分类器的速度和可计算边界。

1.2 研究现状

当前最新GPU单芯片单精度浮点运算性能理论上已经超过1TFLOPS，这已经接近于一个拥有数十个传统运算节点的小型集群的性能。为充分利用其强大的运算能力，近年来众多的研究者都在致力于探索拓展GPU应用领域的方法。

早期的通用GPU计算(GPGPU, General Purpose GPU) [8]的方法是将待处理数据转化和抽象后映射到图形模型，使用GPU中的可编程光栅处理器和顶点处理器，利用图形API完成计算。此方法难度较大，且仅能适用于很少的计算场合，故应用领域受到很大的限制。

2007年CUDA(Compute Unified Device Architecture) [9]的发布，使GPU在通用计算领域的广泛应用变得更为可行。CUDA采用扩展的C语言作为通用GPU计算的编程语言，提供了直接的硬件访问接口，并基于SIMT单指令多线程模型，让GPU通用计算脱离了图形API和图像模型。

然而对于GPU的应用，目前国内外的研究热点主要还集中在图形图像处理、可视化计算和三维场景模拟等方向 [10,11]。这些方向实质上都没有脱离GPU的传统应用领域，只能算作GPU功能的小范围扩展。

将通用GPU并行计算完全应用于图形图像领域之外的场合，是最近才有的一些尝试。最开始的一些工作，主要包括矩阵乘法 [12]、大数组扫描 [13,14]、网络编码 [15,16]、线性链表 [17]以及各种各样的排序算法 [18,19]，都取得了较好的效果，为后来的应用提供了基础。

国内对GPU通用计算的研究当前还处在起步阶段，相关文献还比较少，仅有的一些也主要关注在图形图像领域的应用 [20,21]。部分科研院所也应用GPU强大的运算能力和现有的矩阵算法，将GPU用于地震数据分析 [22]，也取得了较好的效果。

最近部分国外研究学者开始了关于将数据挖掘的部分基本算法在GPU下进行实现的研究 [23]。例如K均值算法 [24,25]和Apriori算法 [26]在GPU下的实现已经有了一些文献报道。

本文正是在这种情况下，进一步尝试将通用GPU运算应用于数据挖掘和机器学习领域的分类算法。当前的常见分类算法基于统计学习理论基础上，学习过程中的数据量和运算量都比较大。例如K最近邻方法，最早是20世纪50年代最早提出的，但由于计算过程属于高度密集型，所以十几年之后计算机的性能大大增强才开始得以真正流行。当前它被广泛应用于数据挖掘和机器学习的各个领域。

决策树算法产生于20世纪70-80年代 [27–29]。算法所构造的分类器是一个树形结构的决策树，从树的根开始各节点表示根据不同属性所采取的判断分支，最终得到叶子节点的判定结果。

贝叶斯分类是一种基于贝叶斯定理的统计学分类方法，在具有高准确率的同时也兼具较高的速度。朴素贝叶斯算法是假设各个属性都是独立的前提下，将标号预测为具有最高后验概率的一个类。

支持向量机(Support Vector Machine)是根据统计学习理论研究成果，由V. Vipnik等在20世纪90年代提出的一种新的机器学习方法，不但可应用于分类，而且可扩展到回归。此方法在解决小样本、非线性及高维模式识别问题中表现出许多特有的优势，已经在模式识别、函数逼近和概率密度估计等方面取得了良好的效果。具有相对优良的性能指标。但缺点是庞大的系统开销导致此方法无法应用于大数据量和过高维度数据的分类。当前也有一些改进方案，例如序贯最小优化(SMO)算法 [30,31] 以及Kernel Cache、Shrinking等优化方法。对于此类大复杂度的分类算法，目前缩短运算时间的有效手段，是传统并行计算的方法。

1.3 研究内容

本文专注于数据挖掘和机器学习领域的各种分类算法，在对GPU的体系结构和性能特点进行研究的基础上，探讨在通用GPU并行计算条件下这些算法的设计原则与实现方法，并对部分算法进行了实现和验证。由CPU与GPU发挥各自优势协同工

作，大幅提升运算性能，扩大了分类器的可计算边界，从而使得大样本数据的学习和分类变得可行。具体工作包括以下几个方面：

（1）在对GPU的软硬件体系结构进行研究，以及对各种分类算法进行分析的基础上，探讨将GPU应用于分类算法的可行性。由此提出基于GPU的算法的设计原则和方法。

（2）*K*最近邻分类是模式识别和数据挖掘中应用广泛的一种分类算法，简单易行且错误率相对较低。但是对大量数据的分类是一个高度密集的运算。本文首先以此为例提出GSNN算法，探讨如何利用GPU的多级存储器体系结构，发挥GPU的运算能力，使得计算过程相比CPU串行算法得到较大的性能提升。

（3）支持向量机分类在解决小样本、非线性及高维模式识别问题中表现出许多特有的优势。但作为一个解决二次规划的问题，算法的复杂度比较高，在大数据集下的训练速度极不理想。为此，提出了适合CUDA体系结构中细粒度多线程模型与多级存储模型的通用GPU运算解决方案，即GMP-CSVC算法。

（4）支持向量机在实际应用中需要经过参数寻优过程，因此构造分类器所耗费的成本被进一步提高了。参数寻优过程涉及训练和标号判定两个主要的运算步骤。本文提出的GMP-nuSVC算法，在训练步骤将原始支持向量机的改进算法ν-SVM移植到GPU上执行，在标号判定步骤应用一种适合GPU运算的分块方案。最终算法取得了较好的效果。

1.4 研究意义

本文将通用GPU计算引入到分类算法这个应用领域，具有一定的创新。同时对分类算法的应用和GPU计算的发展都具有一定的借鉴和参考。主要包括：

首先，扩大分类算法的可计算边界。在传统环境下对大量高维数据的学习分类，效率很多时候都是制约应用的瓶颈。降低数据维度或缩小运算集数量的方法不但难以操作，而且可能导致分类结果准确度的下降。而大规模并行计算成本又非常高。如能利用GPU的优势，在一定的经济和技术条件下获得较高的运算性能，扩大分类算法的可计算边界，对相关的应用会有很大的帮助。

其次，性能的大幅度提升，对分类算法的应用有积极的推动作用。在性能的不断提升中，本来需要长时间等待的工作，可以实现近似实时得到反馈结果。也可以适当提高精度，计算更多的内容，获得更准确的结果。甚至原本在种种约束条件下

不可行的计算工作，提高了效率后变得可行，从而实现从量变到质变的飞跃。

第三，对拓展GPU的应用领域起到借鉴作用。目前GPU强大的运算能力只有在少数图形图像处理场合才得以发挥，其余时间都则在闲置状态。可以预计，刚刚起步的通用GPU计算，必将在高性能计算领域有所作为。而当前CPU+GPU的并行集群已经产生，在2009年11月的TOP 500超级计算机排行榜上位居第5位的，就是我国应用此体系结构的天河一号 [32]。在此结构下对各种算法和应用的研究是一个热点问题。本文研究分类算法在此体系结构下的应用，对其研究发展也有积极促进作用。

1.5 全文组织结构

第一章绪论，主要介绍本文的背景、研究现状、研究内容和意义。

第二章相关技术及研究基础，对数据挖掘和机器学习领域相关的分类算法进行了分析，对并行计算和基于本文研究重点的通用GPU运算做了介绍。

第三章基于CUDA的算法设计思想，在对GPU性能优势进行分析的基础上，提出GPU算法的设计原则，并讨论了算法评估的方法。

第四章K最近邻分类算法在GPU上的实现，描述了本文提出GSNN算法，包括距离计算阶段的分块策略和最近邻选择方法，并对算法进行了测试并分析。

第五章提出基于CUDA的SVM训练算法（GMP-CSVC算法），针对SVM算法复杂度较高、仅能针对小样本训练的问题，使用GPU实现了C-SVM分类器，大幅度扩大了SVM分类器的可计算边界。

第六章GMP-nuSVC算法的实现，针对训练SVM分类器所涉及训练和判定两个阶段的算法，分别进行了改进和实现。在训练阶段，基于ν-SVM的改进算法，提出CUDA下的改进方案。另外的判断算法部分，也充分发挥了GPU的优势所在，提出了具有较高加速比的算法，并给出了实验结果。

第七章总结与展望，对全文进行总结，并展望未来工作。

第二章 相关技术及研究基础

2.1 分类算法

2.1.1 分类算法简介

数据挖掘是基于机器学习、模式识别和统计学等方面的成果，在现有的大量数据中，通过对数据的模式建立和模式识别，从而发现感兴趣信息的过程 [33]。

当前广泛流行的数据挖掘相关的理论和技术，大都是以统计学为基础的。统计学是一门通过收集数据、分析数据，并根据数据进行推断的科学。通过对收集的数据进行描述统计，包括对客观现象的度量和对收集的数据资料进行加工整理和描述，从而进行推断。推断是在搜集、整理样本数据的基础上的，是基于已有数据的一种模式建立和模式识别的过程。在没有明确假设的前提下去挖掘信息、发现知识，得到事物之间的关联，或者进行建模和预测等等，是数据挖掘的最终目的。数据挖掘所得到的信息应具有事先未知、有效和实用三个特征。

通过对数据的处理，将数据用n维的属性向量$X = (x_1, x_2, \cdots, x_n)$来表示，向量元组的$n$个维度分别对应属性$A_1, A_2, \cdots, A_n$的度量值。对像这样已经规范化整理完毕的数据集合进行学习和建模，是数据挖掘的一个重要分支。如果学习的样例由输入输出对给出，也就是向量元组及其类标号都是已知的，称作监督学习(Supervised Learning)。如果数据不包括输出，即分类标号是未知的，则称为无监督学习(Unsupervised Learning)。在监督学习中，如果对数据建立描述模型，从而进行分析预测的结果是产生二元或多元的输出值，分别称作二元分类或多元分类。为此建立的模型就是分类器(Classifier)。如果建模的目的是预测数据未来趋势，所构造的模型预测结果是连续的函数值，则称作回归(Regression)。

2.1.2 分类过程和评估方法

如图2.1所示，数据分类的过程可分为预处理和执行分类两个部分。在预处理部分，需要将原始数据进行分析汇总，填补缺失，除去噪声，也就是数据清理。之后进行数据变换规约，使数据规范化，并进行缩减，除去冗余属性。最终得到接近保持原始数据完整性并大幅度简化压缩的预处理结果。数据预处理部分是分类的基础，

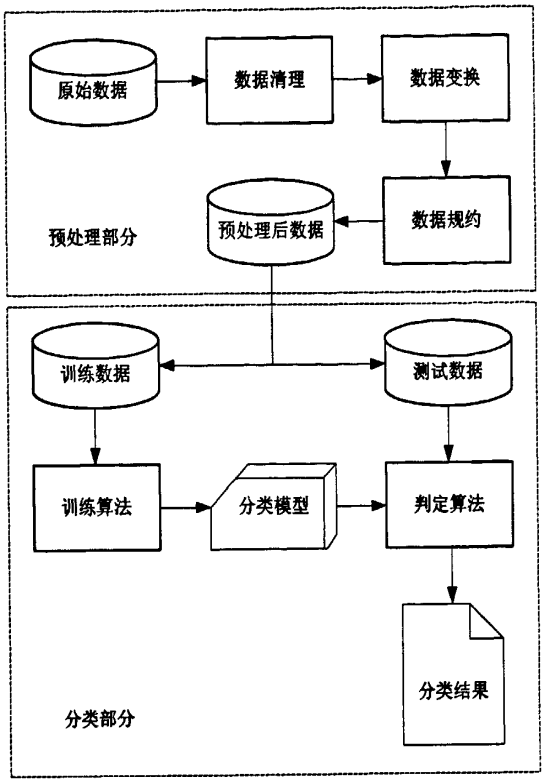

图 2.1 数据分类过程

其结果直接影响到分类的效率和效果。此部分更多地涉及具体领域和应用环境，以及所获得的数据的特性，也有相当多的技巧和方法。本文主要关注分类部分的算法，此阶段相关方法不再赘述。

分类可以分成两个阶段，首先是训练阶段。预处理之后的一部分数据训练数据，执行训练算法。训练算法通过分析，从预先指定分类标号的训练数据中，构造分类器，生成分类模型。此过程也可以看作是学习一个映射函数 $y = f(X)$。对于给定一个元组 X，分类器可以通过计算，将其关联到一个合适的类标号 y上 。这个映射函数，可用分类规则、决策树或数学公式等形式表示。第二个阶段是判定阶段。对测试数据应用分类模型进行判定，即使用训练生成的映射函数，从而得到预测的分类标号。测试数据集已知分类标号，且要求尽量做到独立于训练数据集。对整个测试集进行预测后，最终需要对分类模型进行评估。评估的方法包括如下几个指标：

准确率：准确率是分类器正确分类的检验元组所占的百分比。如果训练集中噪声数据或异常点过多，影响分类效果，就需要重新考虑数据采集和预处理的方法。

也有些时候是训练集的数据量过小，无法得到准确的训练模型。加大数据量来重新训练，就很有可能遇到性能问题。

性能：训练过程中所产生的时间开销。当数据量比较多，数据维度又比较大时，很多分类算法的执行效率是需要考虑的问题。此时并行计算就变得尤为必要。

鲁棒性：在噪声数据异常点存在的情况下，正确构造分类器的能力。

可伸缩性：当数据量和数据维数在较大的范围内变化时，算法是否都能够有效构造分类器。

可解释性：这是分类模型对数据的理解和洞察水平，也就是分类的依据是否有意义并符合逻辑。

2.1.3 常用的分类算法

本文以K最近邻和支持向量机算法为例，介绍基于CUDA的GPU运算在分类算法中的应用。下面详细介绍这两种分类方法。

1. K最近邻分类

K最近邻方法是一种基于实例的学习法，将待判定的数据元组同所有训练元组进行比较，通过与其最相似的一些训练数据的标号来估计该分类标号。算法所存储和计算的数据都是来源于训练集的实例。算法在执行过程中，首先是搜索模式空间，找出对检测元组最近或最相似的K个元组。近似度或相似度也称为样本间的距离，可用欧氏距离表示，即采用如下公式计算：

$$dist(p,q) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2} \tag{2.1}$$

或者采用曼哈顿距离公式：

$$dist'(p,q) = \sum_{i=1}^{n}|p_i - q_i| \tag{2.2}$$

与检测元组距离最近的K个样本的标号是已知的，因此可以用简单的少数服从多数的原则对测试元组的标号进行判定。训练集的数据质量直接影响分类结果，同时参数K的选择也比较重要。例如图2.2，当$K = 3$时，距离圆形待分类点最近的K个元素包括2个三角和1个矩形标号，会判定待分类点为三角形的类。但当$K = 5$时，矩形标号占据三个，待分类点就会判为矩形。也可以采用根据排序结果的顺序进行加权，

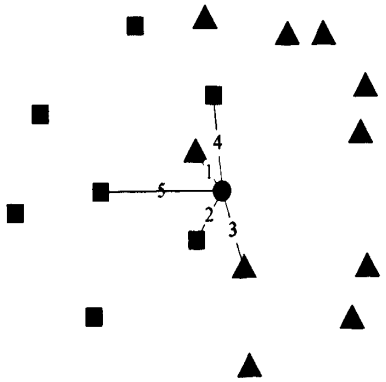

图 2.2 *K*最近邻算法中参数*K*的选择

距离待分类点越近权重越高，最终判断分类标号的方法。参数*K*即影响分类的粒度，需要根据实际应用需要进行调整。

K 最近邻算法虽然简单，但是计算量之大直接影响其应用。随着样本数或者数据维度的增加，其运算量都是呈指数的方式增加的。故很多研究也是使用并行计算或对算法进行一定的改进来进行优化。

2. 支持向量机

支持向量机的分类算法从本质上讲是一种前向神经网络，根据结构风险最小化准则，在使训练样本分类误差极小化的前提下，尽量提高分类器的泛化推广能力。从实施的角度，训练支持向量机的核心思想等价于求解一个线性约束的二次规划问题，从而构造一个超平面作为决策平面，使得特征空间中两类模式之间的距离最大，而且它能保证得到的解为全局最优解。

将两个类的标号y记为+1和-1，属性向量由*X*表示。如图2.3所示的是一个简单的线性可分的数据集，即可用一条直线将两个类分开。但这样的直线可以找到无数条。而最好的一条线无疑是分离两个类的同时使得边界最大的。

对于三维的数据集，分隔两类使边界最大的是一个平面。由此扩展到多维的情景，最大边缘超平面(Maximum Marginal Hyperplane)就是要得到的结果，可以记做：

$$W \cdot X + b = 0$$

对于整个训练集，

$$W \cdot X + b \geq 1, \quad y = 1$$

$$W \cdot X + b \leq -1, \quad y = -1$$

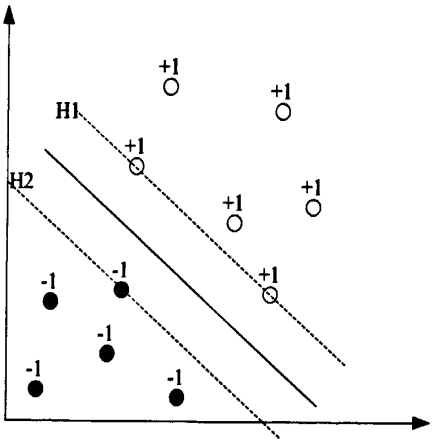

图 2.3 最大超平面

从超平面到 H1 和 H2 的距离均为 $1/\|W\|$，即 $\sqrt{W \cdot W}$。落在边缘 H1 和 H2 上的点，就称作支持向量(Support Vector)。SVM分类器的构造过程就是找到寻找分类间隔$2/\|W\|$最大化条件下的所有支持向量，并将其保存到模型中。

对于线性不可分的情形，可以使用一种非线性变换的方法，把低维数据空间上的点映射到较高维的空间后，实现线性可分。这种数据映射用Φ表示，而$K(x_i, x_j) = \Phi(x_i)^T \Phi(x_j)$就称核函数。常见的核函数有：

线性核：$K(x_i, x_j) = \mathrm{x}_i^T x_j$

多项式核：$K(x_i, x_j) = (\gamma \mathrm{x}_i^T x_j + r)^d, \gamma > 0$

径向基（高斯）核：$K(x_i, x_j) = \exp(-\gamma \|\mathrm{x}_i - x_j\|^2), \gamma > 0$

S型核：$K(x_i, x_j) = \tanh(\gamma \mathrm{x}_i^T x_j + r)$

然而有时在映射到高维的空间后也并非一定线性可分，再尝试向更高维的空间映射会产生较高的成本。很多时候只需要达到近似线性可分即可。故引入松弛变量ξ，使得训练集的点满足如下条件：

$$y_i(W^T x_i) + b) \geq 1 - \xi_i, \qquad (i = 1, 2, \cdots, \ell)$$

其中ℓ是训练样本总数。松弛变量的引入，使得离群点被允许存在。故还需引入惩罚因子C来表示对离群点的重视程度。C越大，说明对离群点越加敏感，期望减少离群点的存在，分类结果也比较精细。C较小，表示分类可以比较粗糙。

最终SVM分类即为求解如下问题：

$$\begin{aligned} \min_{W,b,\xi} \quad & \frac{1}{2}W^T W + C\sum_{i=1}^{\ell}\xi_i \\ s.t. \quad & y_i(W^T\Phi(x_i)+b) \geq 1-\xi_i \\ & \xi_i \geq 0 \\ & i = 1,2,\cdots,\ell \end{aligned}$$

总之，各种分类算法普遍具有一定的复杂度，在大数据集下的学习训练的时间问题经常是制约应用的瓶颈。甚至SVM分类算法直到今天，仍只适用于较小的样本训练。并行计算长期以来也是大规模数据挖掘的重要手段。

2.2 通用GPU计算

2.2.1 并行编程模型

人们对计算机系统提供更强运算能力的需求是不断增长的，传统的单处理器计算机无法达到高性能计算的要求。并行计算机就是由多个处理单元组成的计算机系统，这些处理单元相互通讯和协作，能快速、高效地求解大型复杂问题 [34]。此时需要将整个待求解的问题被分成若干部分，然后每个部分各交给一个处理器，并行地计算。这就是并行编程 [35]。

并行程序设计中遇到的最大问题就是，如何将问题分解为多个能够并行执行的子问题。为了实现问题的分解，必须对所应用的并行计算机体系结构有清晰的认识，应用特定的编程模型。当前最常用的两种并行编程模型，包括共享存储模型和消息传递模型 [36]。

多核和对称多处理器的系统，采用的是共享存储模型。共享存储模型假定所有的数据结构分配在一个公用的区域中，所有的处理机均可访问该区域。图2.4所示的是一个应用共享存储模型的对称多处理器系统。其中，每个处理机都有专用的缓存，通过系统总线连接到一个全局共享存储器上。

对共享数据的同步访问是共享存储模型的一个主要问题。用户必须确保多个处理器在一个共享数据结构上的操作准确无误，即在设计算法时，要人工确保该数据结构始终保持一致性状态。

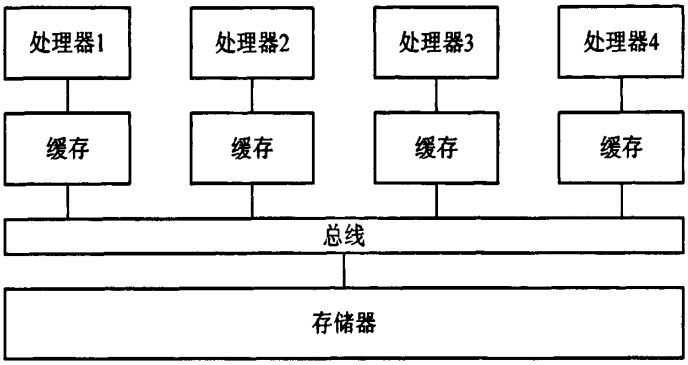

图 2.4 具有一致性访问的共享存储体系结构

图2.5是一个应用消息传递模型的分布式存储结构的系统。消息传递模型假定每个处理机都有自己的私有存储空间，其它处理机不可访问。当某处理机需要访问其它处理机的数据时，必须将数据显式地复制到本地。

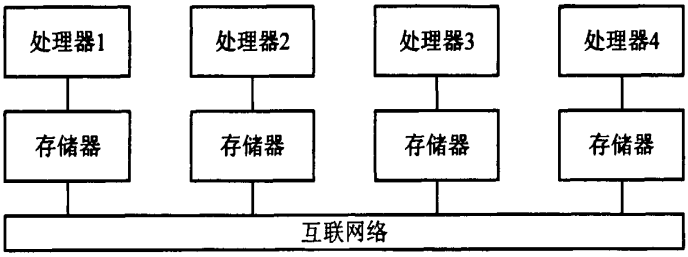

图 2.5 分布存储体系结构

消息传递是目前使用广泛的一种并行计算模型。在消息传递模型中，进程间的通讯和协同行为，通过调用库函数发送和接受消息完成。此模型优势是具有高度的可移植性，理论上不但适用于分布存储体系结构，也可应用于共享存储结构。

2.2.2 GPU硬件结构

图形处理器在近年来得到飞速发展，芯片所采用的制造工艺和晶体管集成度都可以和中央处理器相媲美。然而其独有的特性，使得如果想方设法将其应用扩展到通用计算领域，并采取合适的算法，就可以获得惊人的性能。故研究通用GPU计算，需要首先了解其体系结构。

目前的GPU主要包括ATI和NVIDIA两大制造商。二者的GPU结构和流处理器的组织方式都有所差异。其中NVIDIA GPU的体系结构对通用计算有更多考虑，因而

本文以此系列的GPU为例进行研究。当前的NVIDIA GPU，都采用将每8个流处理器(SP, Streaming Processor)分成一组的形式。这8个流处理器以及其他一些辅助计算部件，组成流多处理器(SM, Streaming Multi-Processor)。流多处理器作为GPU的一个任务执行和调度单元，负责执行GPU分发的线程指令。

流处理器，也称线程处理器(TP, Thread Processor)，是GPU中最基本的指令执行单元。它主要由四个部分组成：寄存器控制器、加法器、乘法器和算术/逻辑器。每个流处理器在每个时钟周期内都可完成一条逻辑运算指令、32位单精度浮点加/乘运算指令、32位整数加或24位整数乘法指令。而32位整数乘法性能是32位的1/4，32位单精度浮点除法是乘法的1/9左右。流处理器不支持双精度浮点小数运算。

流多处理器，也称线程处理器组(TPA, Thread Processor Array)，是GPU中的任务调度单元。它负责控制所下辖的8个流处理的指令执行。也就是说，流处理器的指令部件被去除，其操作由流多处理器来控制。流多处理器执行指令解码等操作后，对这8个流处理器进行指令发射，统一控制操作。因此，8个流处理器所执行代码必须是高度同步的和一致的，才能发挥其性能。流多处理器内还包括2个特殊函数单元(SFU, Special Function Unit)，可以执行一些诸如倒数、三角函数、平方根函数等操作，其位宽也比较大。在GT200核心中，每个流多处理器还配备一个双精度浮点运算器，以支持双精度运算。所以双精度浮点运算的性能在这种结构下，为单精度浮点运算的1/8。GPU的理论运算性能，是非常难以在实际应用中达到的。程序只有在算法合理，且经过了精心细致的优化后，才会有较好的性能结果。

2.2.3 CUDA体系结构

最早将GPU用于通用计算的尝试和探索是针对GPU仅提供图形API的情况，将需要处理的计算任务映射到图形模型，使用图形API运算完毕后，再转换回原始的模型下。这种方法经常可以取得较好的效果，但是更多的运算任务难以使用图形API进行映射。这种模型的转换为解决的问题以及应用的难度设置了很大的障碍。

2007年CUDA(Compute Unified Device Architecture)发布，是GPU运算的一场革命，从此引发将GPU用于通用计算的研究热潮。2008年之后的NVIDIA G92和GT200系列的GPU，在关注图形图像性能的同时，无论是整体架构还是软件驱动，都把通用计算作为处理器改良进步的重要部分。

CUDA是一种与GPU紧密结合的，基于共享存储器的并行体系结构。包括相应

的GPU、驱动程序以及编程环境。CUDA使用类似C语言的编程环境，屏蔽掉了纯粹用于图形图像方面的接口，提供了一种多线程并行环境，采用SIMT(Single Instruction Multiple Threads)单指令多线程模型。从而使得通用GPU运算变得更具有可操作性。

在通用GPU计算的推动下，GPU开始在部分高密集度的运算场合取代CPU执行计算任务。这就推动了一种新的高性能计算机体系结构的产生。使用GPU运算节点组建的高性能集群，也开始在超级计算机中占有一席之地。随着GPU在高性能计算领域的拓展，Tesla通用计算设备随之产生。Tesla设备外观整体上类似于显示卡，但去掉了图像输出功能，同时大幅增加了显示内存的容量，以专注于通用计算领域。本文算法在实验验证部分，也应用了这种设备，取得很好的效果。

2.2.4 CUDA线程模型

CUDA多线程模型如图2.6所示。图中的Host就是传统CPU与内存构成的主机环境。计算机中的所有设备都是CPU进行控制，CPU中通过使用GPU设备(Device)进行运算的一个任务，称为一个内核(Kernel)。将一个内核划分成一个二维网格(Grid)的线程块(Thread Block)，任务或待处理的数据也随之进行了划分。任务以线程块为单位，交付给GPU设备中的基本调度单位，也就是流多处理器(SM)进行计算。线程块内包含大量线程(Thread)，线程在线程块内以三维的方式进行标号。这样对各种应用都具备很好的适用性。

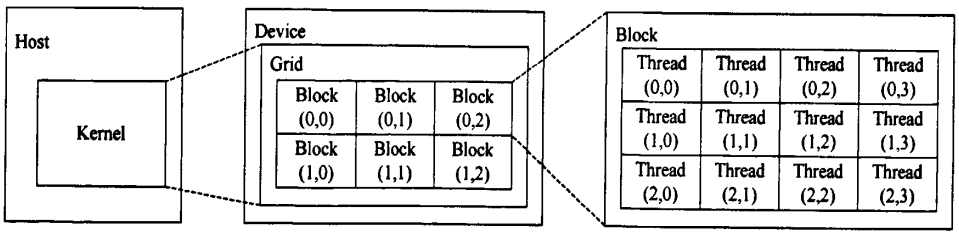

图 2.6 CUDA线程模型

2.2.5 CUDA存储器模型

图2.7显示了CUDA体系结构下的存储模型。图中的主机内存标识为Host Memory，设备上显存主体部分称为全局存储器(Global Memory)。全局存储器无论是速度还是位宽都远远超过内存。因而在高端显卡或专用的Tesla GPU运算设备上，全局存储器的存取带宽要比内存高出几倍。

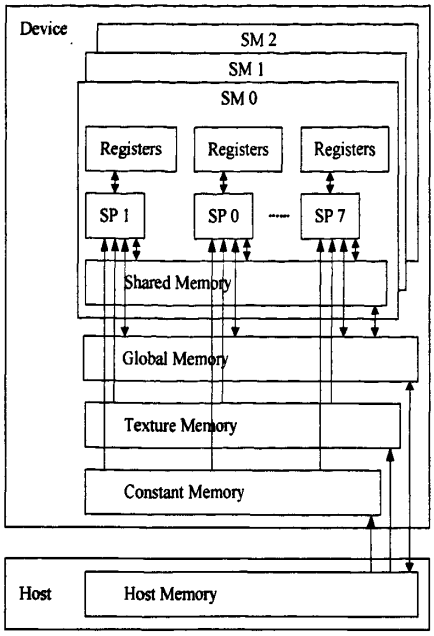

图 2.7 CUDA存储器模型

常量存储器(Constant Memory)和纹理存储器(Texture Memory)也都是显存内的区域，但由于都是只读的，且在流多处理器上有缓存，因而比全局存储器更快。但常量存储器仅有64KB的限制，纹理存储器也属于图形编程领域的内容，因而需要仔细设计算法，才能合理使用。

每个流多处理器包含16KB、被组织成16个bank的共享存储器，以及8192个寄存器。在较新的GT200核心中，寄存器数目得到了加倍。共享存储器的数据只能在线程块间的线程内共享，而寄存器只能由线程独享。其作用域都是不同的。但二者的速度相同。因而合理使用共享存储器是获得更高加速比的关键。

总之，GPU作为图形硬件，具有其特殊的性质，其体系结构也比较复杂。算法的设计要根据体系结构，才能实现较高的性能和较好的效果。

第三章 基于CUDA的算法设计思想

基于CUDA设计高效算法，必须建立在对GPU体系结构充分认知的基础上。本文在对GPU性能优势进行分析的基础上，提出基于CUDA算法设计的一般原则与优化策略，并据此介绍本文分类算法所共同使用的存储方案。最后还讨论了本文在对算法进行评估时所采用的方法和指标。

3.1 GPU算法设计原则

3.1.1 对GPU性能优势的分析

同传统的CPU相比，GPU拥有强大的计算能力。很多高密集度的运算，使用GPU会得到惊人的性能提升。

从硬件设计角度分析，CPU设计和制造中，将重点放在了常规串行代码的执行上。为了提高串行代码的执行速度，采用了复杂的指令解码、分支预测部件，使用了超长的流水线的技术。更重要的是CPU内部集成了多级的高速缓存，包括指令缓存和数据缓存，从而大幅度提高了执行性能。缓存要涉及到数据一致性、提高命中率的问题，以及缓存预取等方面复杂的技术。因此最终CPU中晶体管数目的80%都集中在了缓存及其附属控制部件上。

而GPU自产生以来就是为了图形图像处理领域的应用，此方面应用的特点，首先是数据量非常大。这些大的数据量要求比较高的存储器带宽，而缓存的作用则有所降低。第二，图形数据的格式都非常地统一，无论是表现空间上的点还是色域中的颜色，都是采用数据元组的形式。大量的数据存储的方式就是大规模的矩阵。这些数据非常易于划分。第三，对于图形图像领域的很多运算，对每个点所运算处理的方式都高度一致，且相对比较地简单。第四，三维场景的生成、物体的渲染等快速屏幕显示的内容，速度远远比精度重要。过高的精度也意味着巨大的带宽需求和存储空间，即使在当前，32位的数值精度也是绰绰有余。

正因为如此，GPU的发展走是一条与CPU完全不同的道路。在GPU的设计制造中，是采用的是大规模数据并行的方式，将很多结构简单的处理单元合并在一起，执行和向量计算机类似的并行计算的方法。GPU将最大部分的晶体管用于执行单元

部分，缓存和指令控制被大幅度地削减。这也就是在同样技术条件下，GPU的理论计算性能更加强劲的原因。而GPU运算精度受限等问题，直到当前仍然存在。

从体系结构的角度分析，GPU中的CUDA架构相比CPU X86架构的优势主要体现在以下三点：

（1）寄存器数量。X86 CPU的只有8个寄存器，而当前支持CUDA的GPU中每8个流处理器一组，总共可支配8192个32位寄存器。X86架构的CPU临时变量放在内存中的栈上，虽有缓存，但存在延时。CUDA中，直接使用寄存器存放临时变量，因此其速度有一定优势。

（2）线程管理。X86架构CPU每创建一个线程需要上万个时钟周期。CUDA中线程可批量创建，创建和维护成本几乎可以忽略。同时线程切换的时间，在GPU上能够实现单个周期的切换，而在CPU中需要保存现场等大量操作，开销很大。

（3）指令执行。多核X86架构CPU中，不同线程的指令独立进行指令解释和译码。CUDA中，线程按组管理，同组线程共享指令流。更适合大量数据相同流程的并行处理。

3.1.2　一般原则

为发挥GPU的性能优势，在算法设计时一般需要遵循如下几条原则：

第一，保证线程束内的运算过程高度统一，以避免分支。

GPU中的流处理器虽然号称是标量处理器，但是具有很多向量处理器的特征。流处理器指令的解码和执行在流多处理器的统一控制下进行，并且线程执行时，流多处理器对其所下辖的8个流处理器进行指令发射，以32个线程所组成的线程束(Warp)为单位。因而，每次指令发射，8个流处理器持续执行4个周期对整个线程束进行处理。

这种方法在大大简化控制电路，以增加运算部件的同时，对通用计算的算法和程序设计提出了更高的要求。程序分支需要仔细斟酌，代码执行过程中同一个线程束内不同分支的情况应该尽量避免。这种线程束内的分支并不会使得程序出错，而是大幅度地降低效率。

假设当程序执行时，遇到了形如“if(A) section1; else section2;”的代码，指令分发单元将会以线程束为单位发射指令，各个线程步调一致地从条件判断语句开始统一执行。假如各个线程分别对条件A进行判断，且不同线程得到两种不同的

结果：满足和不满足。之后的执行将采取如下的方式：首先指令发射单元统一分发section1程序段的指令，满足条件A的线程接受并执行，不满足的线程空闲等待；之后发射section2程序段的指令，执行的线程正好相反，不满足条件A的线程同步执行，另一部分则空闲等待。

当出现分支更多的情况时，性能问题更加明显。最极端的情况是线程束内的32个线程有完全不同的分支，此时程序实际上成为了串行执行。GPU结构的这种指令发射方式，对算法的设计制造了很大的障碍。很多原本经典的并行算法在这种结构下无法发挥出应有的性能。基于CUDA的很多高效算法采用的都是独特的数据划分方式和执行策略。

第二，充分利用CUDA结构下细粒度多线程模型与多级存储模型。

并行算法所涉及的一个重要问题就是对任务或数据进行划分。在CUDA中，对任务或数据进行划分的粒度，与CPU并行完全不同。GPU采用的是一种细粒度的划分方式。将整个网格的线程划分为线程块，分配给流多处理器进行处理。每个流多处理器再将线程块以每32个为单位分成线程束(Warp)执行。在算法执行时，如果GPU中的线程数目在不足上万个的规模，根本无法发挥出应有的计算性能。

GPU中存储器的分层结构包括寄存器、共享内存、本地内存、常量内存、全局内存和系统内存多个层次。每个层次的存储器都有不同的作用域和速度时延。因此，在算法设计中，充分利用各种类型的存储器，尤其是共享内存，将会有很好的效果。

第三，运算精度也是通用GPU运算必须考虑到的问题。

在当前GPU的流处理器，最擅长的是32位浮点小数的运算。而32位整数的乘法性能，都相对24位的有大幅度下降。另外，双精度的浮点运算在GT200之后的GPU中才得到支持，而这种支持也是在增加专门的双精度运算单元基础上实现，而并非由最主要的运算单元，也就是流处理器来完成，故性能打了很大的折扣。

GPU执行单精度浮点运算时，存放中间结果的寄存器位宽是32位，与CPU中的浮点运算器的80位相比小了很多。另外，IEEE-754标准中，对于0附近的小数，采用了非规范化(Denomalization)的表示方式。也就是说，0附近的数字精度，在完整实现IEEE-754标准的CPU运算中，要比其它数字略高。而GPU对非规范化数字暂时无法支持。

总之图形图像领域对运算精度要求不高，使得GPU以牺牲精度为代价实现了高

速度。在GPU中执行通用计算，结果的精度需要经过评估。算法设计也要避免由于精度而产生严重后果。

本文将GPU应用于分类算法，精度问题经过评估测试，发现对结果影响在可控制的范围内。因为分类算法是对向量标号的预测，本身就有一定的错误概率的容忍程度。而在分类的过程中，在很多应用中所判断的依据是由结果的数量级来完成的。因而小幅度的误差对分类影响很小。这也是本文算法比较成功的原因之一。

3.1.3 性能优化策略

GPU算法的设计，还需考虑存储器的访问指令，对此针对硬件约束做出优化。

首先，对于全局存储器，隐藏访问延迟和实现协同访问是两种高效优化方法。

全局存储器是位于显示内存上的，供整个任务共享的数据部分，其容量依显卡配置而定。不同规格的显卡或GPU计算设备，其容量区别很大。因此算法如果有可移植性和可扩放性的要求，就要充分考虑各种情况，保证在各种环境下的正常执行。然而如果走向另一个极端，不能充分利用GPU上的存储器资源，也是对资源的浪费。

全局存储器虽然频率和带宽都远远大于内存，但在GPU能用到的各级存储器中，仍然是最慢的。线程对全局存储器的访问有200-600个时钟周期的延时，且没有缓存。访问全局存储器经常是算法的性能瓶颈。但当从线程提出访问全局存储器的指令请求，直到其实际获得数据的这数百个时钟周期里，流多处理器并非处于阻塞状态。前面已经述及，GPU中的线程切换仅需要一个时钟周期，并且有自动调度的机制。因此，算法只要保证流多处理器内的线程束足够多，且指令具有足够的运算密度，也就是计算/通讯比，是可以完全实现隐藏访问延迟的效果的。

全局存储器的访问模式也非常重要。如能达到相关的条件，实现协同访问(Coalesced Access)，则对性能会有一个很大的提升。全局存储器的协同访问需要满足如下四个条件：

（1）各个线程访问的数据长度为4、8或16字节；

（2）被访问地址构成一片连续内存空间；

（3）第N个线程访问第N个全局存储器地址（允许部分线程不参加访问）；

（4）起始地址对齐到访问的数据长度的16倍。

其次，对于共享存储器，防止bank冲突是其性能优化的要点。

共享存储器同寄存器一样，位于流多处理器内部，并且都仅有一个时钟周期的访问时延。但其特点是数据可在线程块内的所有线程中共享。因此，利用共享存储器，可实现线程块内线程的通讯，且具有极高的性能。每个流多处理器内的共享存储器容量为16KB。值得注意的是被分成了16个独立且可并行工作的bank来控制。具体划分方法为地址从0开始，每连续32bit为一个bank，顺序排列并循环划分到bank0-bank15。同一个bank每时钟周期只能执行一条对其下辖存储单元的32位访问指令。对共享存储器的访问，也是以半个线程束(Half-Warp)，也就是16个线程为一个单位的。

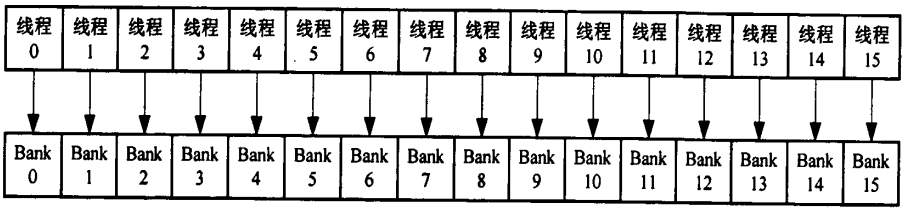

图 3.1 一种无bank冲突的访问模式实例

如图3.1所示，是一种未产生bank冲突的访问实例。此时流多处理器内的半个线程束所执行的指令包括对共享存储器的访问，并且请求的地址分别在不同的bank中，这将是零等待的一次访问，完全体现了共享存储器的高性能。然而，假如图中线程1此时需要访问的数据与线程0所请求的数据，恰好都位于bank0中，则此时产生bank冲突。这种访问需要在2个周期内，由bank0串行地执行，效率被无形中降低了。图3.1仅是无bank冲突的一个例子。事实上，线程号与bank号无需有对应关系，只要地址满足相应条件即可。

另外，常量存储器(Constant Memory)和纹理存储器(Texture Memory)都是位于显存内的区域。但是由于这两者在流多处理器内部有缓存的存在，如能合理使用，也能达到非常好的效果。

3.1.4 存储方案分析

设计基于CUDA的分类算法，首先要考虑向量的存储方案。如果线程间在向量的读取指令上存在程序分支，将会对整个程序的性能产生巨大的负面影响。向量集合在内存中的存放，主要有两种主要方式。

一种是采用链表的稀疏式存储方案，如图3.2所示。在这种方案中，在向量指针所指向的数据节点数组中，每个元素结构体存放维度数和相应维度上的值。此方案在数据挖掘领域中应用最为广泛，原因在于大部分经过预处理的训练数据普遍维度很高且具有稀疏性。只要数据依概率50%以上的维度值为0或是空值，此方法就有优越性。

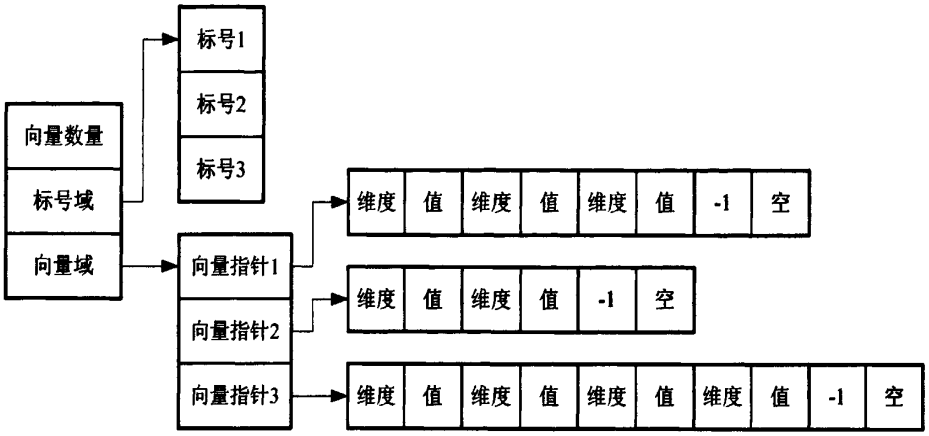

图 3.2 稀疏式向量集存储方案

另外一种是矩阵式。在矩阵中，每行存储一个向量或元组，各个偏移量的位置存放相应维上的度量值。此方法的特点是简单高效，但如果向量过于稀疏，则浪费的存储空间过大。

考虑到GPU的特性，流处理器指令控制和发射的硬件比较简单，更接近向量处理器，线程以线程束为单位进行统一调度，不适合出现程序分支，故本文实现的各种算法，均采用矩阵式的存储策略。测试结果表明，即使在较稀疏的数据集上，也得到了很高的加速比性能。当然数据集的密度越高，优势也就更加明显。

进一步考虑到设备存储器，也就是显存的协同访问，能够大幅度地提高访问性能，甚至在流多处理器较高负载的情况下通过快速线程切换，能够实现隐藏存储器访问时延。所以矩阵式存储的具体方案也需要进一步分析。

首先，向量究竟是按行还是按列来存放是第一个问题。如果每个线程处理一个向量，数据根据向量进行划分，算法迭代读取和计算每一个维度的数据，按列存放是实现协同访问的必要条件。但如果对高维度的数据集进行计算时，采用线程块内每个线程读取一个维度信息的方式，则将向量按行顺序存放是更为高效的方案。故存储方案要结合算法和数据分割方法，统一考虑。

另外，矩阵式存放还需注意到矩阵宽度的问题。如果矩阵的宽度没有对齐到数据长度的16倍，也就是在32位浮点数的矩阵中，宽度不是64的倍数，则无法保证每一次的读写操作都满足协同访问的要求。因此矩阵的分配和数据的转换也是算法设计要考虑的问题。例如在CUDA中所提供的cudaMallocPitch()函数就是分配二维数组时考虑行的宽度，以保证每一行的首地址满足对齐条件。

3.2 GPU算法评估

3.2.1 加速比

加速比(Speedup)是衡量并行算法好坏的标准之一。加速比的定义为，对给定的一个应用，串行算法（或串行程序）的执行时间除以并行算法（或并行程序）的执行时间所得的比值。对p个处理器的系统而言，可获得的加速比一般小于p。原因在于程序总是有可并行部分以及剩下的串行部分。串行部分的比例越大，对加速比的负面影响也就越大。

在传统的CPU并行中，测量加速比的一个重要前提，是串行和并行程序运行在同样的处理器上。不同的性能的处理器上，程序的执行时间有很大差异，不能作为加速比测试的参考依据。

然而对于通用GPU计算，目前为止尚未有严谨的加速比测量方法。无论是采用绝对加速比的GPU单线程串行算法与并行算法的执行时间相比，还是使用相对加速比的单个流处理器与整个GPU执行时间对比，都非常不合适。首先，GPU的体系结构，决定它根本不适合做串行计算。如偏要单单使用一个流处理器串行执行，将会得到非常糟糕的测试结果。其次，GPU是一个整体，仅使用其中的一个运算单元不具实际应用价值。

同时，不同等级的GPU中流处理器的频率、数目都不相同，各种产品配置差别很大，不同的CPU性能也有差别。在这两种异构的环境下，很难找到最为科学的评测依据。衡量计算机的绝对运算性能可以采用系统每秒执行的浮点运算次数(FLOPS)指标。但对于特定的算法，此数据难以精确测量；对于通用GPU计算，此指标忽略初始显存分配和拷贝时间开销，也不是完全合理。

最终本文在数据分析阶段采取的方法，是在配置与CPU价格类似的GPU运算设备（包括GPU芯片、显示内存及相关器件的板卡）的系统上分别测试CPU执行时间

和GPU执行时间，将加速比定义为：

$$S = \frac{t_c}{t_g}$$

其中t_c是不使用GPU的环境下CPU串行程序执行时间，t_g是执行同样任务的程序采用了GPU加速后所耗费的整体时间。本文所涉及的算法会在不同的系统中分别测试评估，并列示详细配置。加速比只能作为衡量算法优劣的参考指标之一，还需结合其它指标综合考虑。

3.2.2 计算/通讯比

所有的并行算法都涉及到将数据和任务进行分解，分配到不同的处理器上执行，执行当中也会有消息传递或是数据共享同步，这都是很大的开销。将并行算法中，处理器全速运行执行计算的时间定义为计算时间(t_{comp})，将数据分配、等待同步、进程间通信的时间统称为通讯时间(t_{comm})。并行算法执行时间是二者之和：

$$t_p = t_{comp} + t_{comm}$$

而在串行求解的过程中不存在通讯时间，因而这是一种并行后的开销。计算/通讯比就是衡量并行算法优劣的一种指标，定义为：

$$F = \frac{t_{comp}}{t_{comm}}$$

计算/通讯比越高，说明算法的额外开销比例较少，算法也就更为优秀。

CUDA采用一种共享存储的体系结构。其中线程块内的数据共享采用流多处理器内的共享存储器，其时延为一个时钟周期。线程块间的数据共享采用全局存储器，时延虽有数百个时钟周期，但在流多处理器拥有较高负载时，借助快速线程切换来隐藏时延。此时通讯时间实际上主要集中在设备存储器分配和数据传输等操作。存储器的分配由用户在程序中制定分配容量，具体操作交由操作系统，由设备驱动层执行并返回相应指针。不同的操作系统和设备下，此时间都有所差异，驱动程序的逐步更新完善，也会带来更好的性能。数据传输操作，包括数据初始化阶段的主机内存到设备显存的拷贝传输，以及结果的设备显存到主机内存传输。此操作也由驱动层完成，内存和显存的速度、内存控制器以PCI-Express通道带宽都对此有影响。整体实测带宽一般都在数个GB/s数量级。总之，好的GPU算法应当尽可能减少线程同步次数、存储器的分配数量和传输容量，以提高计算/通讯比。

第四章 K最近邻分类在GPU上的实现

4.1 引言

K 最近邻分类 [37]是模式识别和数据挖掘中应用广泛的一种分类算法，其特点是算法简单易行，错误率相对较低。这是一种典型的基于实例的惰性学习方法。

将 d 维的属性向量表示为 $X = (x_1, x_2, \cdots, x_d)$，训练数据集总共包含 n 个向量，则对于 K 最近邻算法，计算待分类的向量与所有训练集中向量的相似度或距离后，根据最近的 K 个邻居判断其类标号。在训练集比较大时，对向量的分类时间一般也能够接受。但大部分的应用中，例如图像处理和文字识别等领域，都需要实时或准实时地对大量数据预测分类结果，这就成为一种具有一定密集度的运算。

本文所提出的基于 GPU 的分块最近邻算法，即 GSNN(GPU Based Segmentation Nearest Neighbor) 算法，在距离计算步骤采用了一种适合 GPU 的分块策略，同时对多个向量进行计算；在最近邻的选择步骤使用了一种评估选择方法，从而实现了高维大数据量的快速分类。

4.2 GSNN算法框架

4.2.1 K最近邻算法分析

算法的基本思想为，如果一个样本在特征空间中的 K 个最相似样本中的大多数属于某一个类别，则判定该样本也属于这个类别。相似度可用空间距离来衡量，最相似即在特征空间中的距离最近，故称 K 最近邻算法。该算法首先要已知一个已经准确分类的训练数据集，再针对一条测试数据，计算在特征空间中与训练数据集里所有点的距离，最终排序后根据 K 个距离最近点的类标号，决定该测试点的标号。

K 最近邻算法的计算步骤简单，并可适用于高维数据集。但当测试集、训练集和数据维度都比较大时，运算量十分庞大。当数据维度为 d，测试集和训练集分别包含 m 和 n 条数据时，时间复杂度为 $O(m \times n \times d)$。当前也有一些为提高效率而进行优化方法，例如使用 KD-Tree 提高效率 [38]，或者采用降低精度而提高效率的方法。也有文献 [5]提出，训练集中部分数据对最终结果影响不大或没有影响，可以采取精简训练集的方法减少运算量。这些方法在某些情况下最多可使得执行时间减半。

4.2.2 整体计算流程

K 最近邻算法的基本过程为：首先是数据预处理，得到已标号的训练数据集train，以及待分类的测试数据集test；其次针对一个测试数据，计算训练集中所有的点与该点之间的距离；对结果进行排序后，根据参数 *K*，找到与这个测试点距离最近的 *K* 个点；然后根据这 *K* 个点的类标号，由一定的规则判定测试点的类标号；最终循环迭代，判断测试集中所有的点的类标号。算法流程图如图4.1所示。

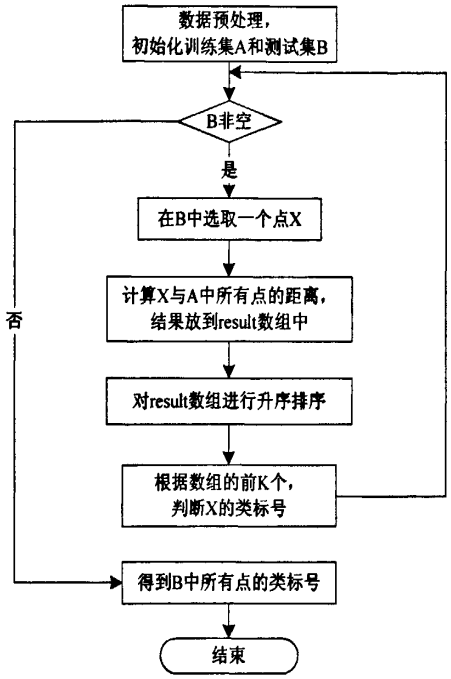

图 4.1 KNN原始算法流程图

数据预处理一步，需要将原始数据处理成数值型的向量集合，并进行数据放缩。此步骤涉及的具体方法与应用领域以及原始数据特点紧密相关，当前也有大量成熟方法，本文不再讨论。距离计算一步本文使用公式2.1计算欧式距离。但最后的求平方根运算对排序结果没有影响，为简化计算可将其省略。最终两点间的距离表示为欧氏距离的平方。

经分析，计算测试集中和训练集中点与点之间的距离和排序一步可采用 GPU 并行完成，其余如判断类标号一步难以在 GPU 上高效实现，由 CPU 完成。GSNN 算法整体过程如算法1所示。

算法 1 GSNN算法整体过程

```
1: init train, test, results_value
2: compute results_value[i][j] as dist(train[i], test[i])
3: for all i in test do
4:     decide the label of test[i]
5: end for
6: output results_value
```

4.3 算法实现

4.3.1 距离计算中的并行分块策略

在计算距离这一步，需要计算训练集里的所有样本与待测试样本的距离。

出于尽量减少程序分支和易于操作的考虑，本文采用矩阵的方式表示高维数据集。训练数据集 A 和测试数据集 B 的数据维度均为 d。用 n 行 d 列的矩阵表示样本数为 n 的训练集，用 m 行 d 列的矩阵表示样本数为 m 的测试集。而结果集 C 存放所有训练集和测试集中两点之间的距离，采用 m 行 n 列的矩阵表示。C 中第 x 行 y 列的元素即表示 B 中下标为 x 的点与 A 中下标为 y 的点之间的距离，即欧氏距离的平方。整体运算复杂度为 $O(m \times n \times d)$。

常规的方式是每次计算两个样本点的的距离。对于测试集B中的每个点，分别计算与训练集A中每个点的距离，A 共要载入 m 次。为了减少数据载入次数，提高运算/通信比，本文采取了一种分块方案。基本思想如图4.2所示。

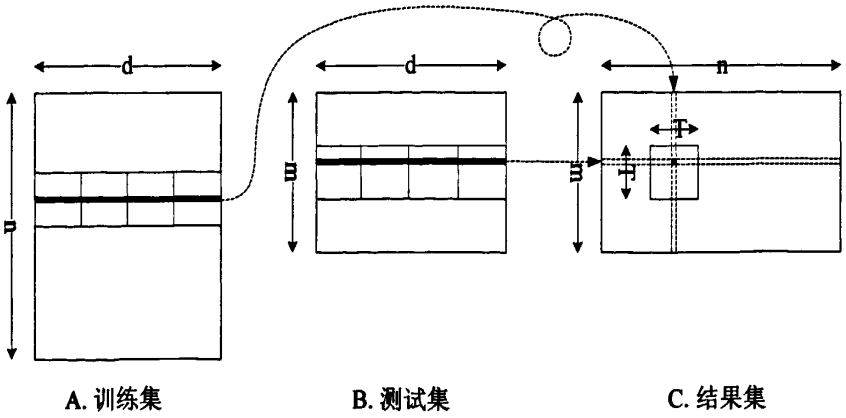

图 4.2 距离计算的分块策略

将数据集C划分成大量宽和高均为T的方块。线程块也据此划分，每个方块由一个线程块进行处理，方块中的每个点由一个线程处理。这样，任务网格中总共包含$n \times m$个线程，x方向和y方向各含有(n/T)个和(m/T)个的线程块，线程块的总数为$(n/T) \times (m/T)$。线程块内，也以二维的方式划分线程，x和y的方向均含有T个线程。每个线程处理C中的一个元素，也就是计算A与B中对应两点的距离。

此方法的优势是可以充分利用流多处理器中的共享存储器。在运算前，线程块首先载入所需数据，在后面一轮的计算中，线程只从共享存储器中读取数据。共享存储器具有和寄存器同样的时延即一个时钟周期。相比全局存储器数百个周期的时延，具有极大的优势。

定义两个位于共享存储器上长度均为$T \times T$的数组，分别命名为shared_train和shared_test。由线程块中的线程x方向的标号，可算得其在A中对应的向量。同理，有该线程y方向的线程号，可算得其在B中所对应的向量。shared_train和shared_test数组的载入就是通过线程对全局存储器的访问完成，每个线程读取一个数值，每次读取T个向量中的T个维度的所有数据。在数据维度d大于T的情形下，采取分步骤载入的模式。将向量在维度d上以T划分成多个块，算法迭代执行，每次载入和计算其中的一个块，训练集和测试集所对应的两个块同步载入。距离计算中，来自训练集和测试集的两个向量的相同维度进行直接运算的，当计算进行到某一轮时，前一轮载入的shared_train和shared_test数组数据已经无用，因而完全可以将其覆盖成新值。计算结果累加到结果集C中的相应位置即可。经过(d/T)次迭代，C中的结果即为所求。各个线程执行的算法如算法2所示。

这种方法当测试集和训练集都比较大时，能够有效减少重复的显存访问。与不采用此分块算法相比，线程总共对显存的访问次数降为$1/T$。

参数T的选择经过了慎重考虑。当前版本的CUDA对线程的一些限制包括，每个线程块最多包含512个线程，每个流多处理器包含768个线程，最多分配8个线程块，以及16KB的共享存储器等等。综合考虑各种约束条件，将T的大小设置为16比较合适。这样每个线程块256个线程，每个流多处理器可同时执行3个线程块，GPU中同时并行执行的线程块数量即为3倍的流多处理器的数目。

算法2计算一个向量距离所需的迭代次数是$\lceil d/T \rceil$。最后一次迭代可能会造成一定的额外开销，这是不能整除所导致的。因此在不能整除情况下数据维度d与T的比值越高，造成额外开销的概率就越小。考虑到在大量的应用中，数据维度都在数

算法 2 distances_computation_in_GSNN

```
1: Each block is given the 2-Dimensional identifier bx, by, and tx, ty for each thread.
2: sub_result = 0
3: temp = 0
4: for all sub_tile in dimension/T do
5:    loads shared_train in train set to shared memory
6:    loads shared_test in test set to shared memory
7:    syncthreads
8:    for k=0 to T do
9:       temp = shared_test[ty][k] - shared_prob[tx][k]
10:      sub_result += temp × temp
11:   end for
12:   syncthreads
13:   sub_result to the corresponding position in result set
14: end for
```

十到数百的数量级，因此采用 16 × 16 的分块也是比较合理的。对于维度远小于 16 的数据集，或是固定二维的数据，则最好采用其它的优化方法了。

此分块策略充分地利用了流多处理器中共享存储器的速度，减少了对显示内存的读写次数。算法在低端的 GPU 上即可达到 CPU 的九十余倍。具体测试数据见实验结果及分析部分。

4.3.2 最近邻的选择方法

测试集中的每个向量与训练集中的向量距离全部计算完毕之后，下面的工作就是从中选出最近的 K 个，以便进行标号预测。

在大量元素中选择最小的 K 个元素，本文采用的第一个方案就是在这些元素中，先找到最小值，取出后以能表示的最大数替换，继续如此重复 K 次寻找最小值。对于长度为 n 的数组，运算复杂度为 $O(Kn)$。

每次迭代中的寻找最小值，实际上是一种并行规约算法。所谓并行规约，就是在一个大数组中执行一次遍历和运算后，最终以一个单值来表示汇总的结果。运算可以包括求和、求最值或是利用其它二元运算符所进行的运算。整体而言，并行规约是一种低密集度的计算，通讯时间占据主要比重。文献 [39–41]提出了在多核和集

群下的并行算法实现。

在传统并行计算机体系结构下，最常见的方法就是采用一种倒二叉树的方式，将数据两两汇总合并，最终到根节点就是全局的汇总结果。无论在什么样的体系结构下，程序的并行度在后期的运算中，肯定是以指数的梯度下降的。节点的利用率，也就是系统整体的负载随着运算的进行越来越低。

显存带宽远高过内存，在执行这种高数据吞吐量的任务时，GPU具有一定的优势。针对并行规约这个问题，文献[13]提出了一个高效的解决方案。该方案充分利用GPU的体系结构特点，使用了一种独特的线程组织方式。最终算法的实测性能可以接近显存的理论峰值带宽，达到数十GB/s的数量级。该算法根据问题的规模，将整体分成多个步骤。每个步骤分成两个阶段。第一阶段如图4.3所示。

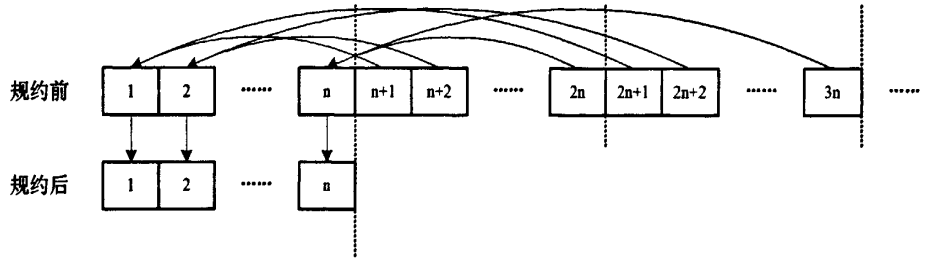

图 4.3 并行规约第一阶段

此阶段将数据划分成多个部分，每个线程块处理其中之一。设线程块内的线程数为 n，则算法初始化时，读取数据块的前 n 个数据到共享存储器中。在第一轮迭代，数据中偏移量为 $(n+1)$ 到第 $2n$ 的数据分别在 n 个线程的控制下同第 1 到第 n 个数据做运算。第二轮迭代，归并偏移量为 $(2n+1)$ 到 $3n$ 的数据。直到所有数据处理完毕为止。这种划分方法，每轮迭代对全局存储器的访问都符合协同访问的要求。

第二阶段的操作是在线程块的共享存储器中进行的，是将这 n 个存放在共享存储器中的数据归并成一个值。方案如图4.4所示。

为了避免共享存储器的 bank 冲突，并且尽量避免在线程束内出现程序分支，每一轮迭代中的前一半数据同后一半中相应位置的元素做计算。在总共参与运算的线程数减少到32个以前，所有的线程数都是满负载运行。直到最后5步才会出现空闲进程。这种方法也极大地避免了后期处理器资源闲置的问题。

在实验中，发现当 K 的值比较大时，这种选择 K 个最小值的算法耗时较大。从算法复杂度上分析，当 K 趋向于数值长度 n 时，复杂度成为 $O(n^2)$。事实上，这演变

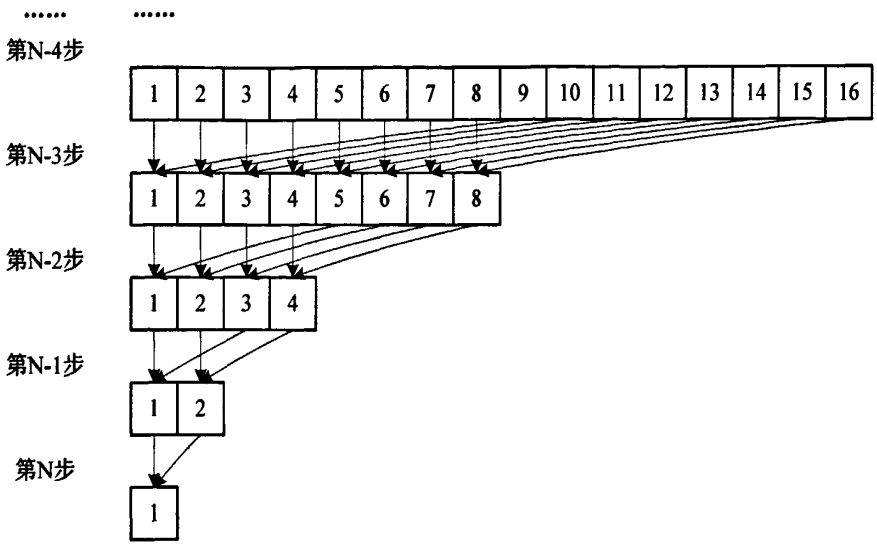

图 4.4 并行规约第二阶段

成为一种选择排序算法。因此，采用复杂度为 $O(n \log n)$ 的排序算法，此时能够提高算法性能。基于 GPU 的并行排序算法也是当前的一个研究重点 [19,42,43]。本文所采用的排序算法，参考了文献 [18]所提出的基数排序算法。

算法整体思路如图4.5所示。将待排序的每个数字以宽度为 w 的二进制存放。例如整形和单精度浮点小数，w 等于 32。这 w 位的数字，每 d 位作为一个单位，划分成多个不同的部分。算法从最右侧，也就是对数值影响最小的 d 位开始依次处理，迭代 w/d 次后完成排序。

每次迭代，首先将数组划分给 p 个线程块，线程块的 t 个线程中，每个处理一个数据。在各个线程块内部根据相应的 d 位数值采用稳定的排序算法进行排序后，回写到数组中该块所属的部分。经过此次运算，图4.5中的每一行，是以这 d 位数值为排序依据的有序数列。第二步，以前一步同不同线程块间线程号的数据为单位，对上一步处理的 d 位前面的 d 位数值求和，从而得到长度为 t 的求和数组。在图4.5中，竖向向下的箭头即表示相同的求和分组。第三步，根据此求和数组值的大小顺序，将相应分组的数据以竖向次序串接成新数组。这个新数组进入下一轮迭代计算。

此排序算法性能较 CPU 快速排序在某些条件下可达到数倍的性能。整体而言，排序算法的计算密集度比较低，线程块内分支较多。相比距离计算步骤，采用 GPU 排序得到的性能提升十分有限。

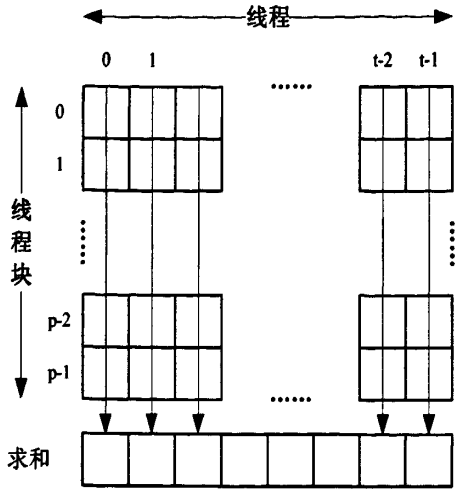

图 4.5 GPU基数排序算法示意

并行规约和排序两种方法孰优孰劣，要看具体应用中 K 的大小。对于排序算法，前 K 个元素的排序才有意义，后面元素的结果是多余和浪费。而对于规约，在 K 比较大的情况下复杂度更高。算法复杂度仅能够衡量随着数据量的增长，运算量的增长变化趋势。运算量的绝对值，也就是执行的时间，才是具体应用更关注的焦点。因此，本文最终采取的策略是选取这二者中运算量的绝对值较少的一个。具体方法是执行初期，分别测试这两种方法在少量同等数目集合上的执行时间，选择时间较短的方法用以后面主体部分的运算。由于测试所用的数据量很小，只要超过计时误差可供评估即可，其开销仅需数十毫秒。所以算法整体效果良好。

4.3.3 决定分类标号

此步骤统计排序结果最前面的 K 个元素的类标号，即训练集A中与测试点 X 距离最近的 K 个点的类标号，以此判断 X 的类标号。

比较简单的方法是对这 K 个标号值进行分类统计，数量较多的一个类标号就是预测结果。另外也可使用一种针对排序次序进行加权的方法，事先定义每个排序位置的权重。位置越靠前，权重越高。最终权重和的值最高者所属的类作为预测结果。总之此步骤运算量很低，程序分支较多，直接由CPU完成即可。

本文采用的是简单数量统计的方法，如算法3所示。

算法 3 lable_decision_in_GSNN

```
1: classNum = 0
2: for i=0 to K do
3:   for j=0 to i do
4:     if j == classNum then
5:       classCount[j].value = label[ results_index[i] ]
6:       classCount[j].index = 1
7:       classNum ++
8:       break
9:     end if
10:    if classCount[j].value == label[ results_index[i] ] then
11:      classCount[j].index ++
12:      break
13:    end if
14:  end for
15: end for
16: maxCount = 0
17: for i=1 to classNum do
18:   if classCount[i].index > classCount[maxCount].index then
19:     maxCount = i
20:   end if
21: end for
22: output classCount[maxCount].value
```

4.4 实验与结果分析

本文采用的测试平台为Pentium D 820 2.8GHz双核CPU，1.5GB内存。作为对比所使用的GPU为nVidia G92核心的9600GSO，包含96个流处理器，搭配的显存规格为192bit/384MB。

执行分类的第一步是数据预处理。由于本文对预处理一步不予涉及，同时为了方便与LIBSVM [44]比较，以及方便在不同的分类算法之间进行对比，所以本文4-6章中的测试数据都是采用LIBSVM提供的预处理后数据。

本章使用的测试数据集类标号2个，数据维度为123。其中a1a训练集大小为30956，测试集为1605个；a2a训练集大小为30296，测试集为2265个。原始数据

集来自于UCI Adult 数据集。该数据集为人口统计数据，根据年薪是否大于五万美元，分成两类。原始的数据元组共14个属性，其中6个连续的数值型，8个枚举类型。

预处理的方法是：对于连续值的属性，将其区间划分成5等分，分割成5个新属性。原属性值落在哪个区间，相应新属性的值就是1。对于枚举型属性，每个取值都作为一个新属性。1和0分别表示满足和不满足该属性。最终Adult数据集在预处理后，包含123个属性，每个属性的值均为0或1。而对于每个向量，最多有14个属性值为1（少量原始数据中有属性为空）。这是一个比较稀疏的向量集合。

本文所提出的GSNN算法的CPU对照程序采用快速排序，标识为CPU，采用文献 [38]中KD-Tree优化的对比算法标识为ANN-Brute。三种方法的程序初始化和输入输出部分基本相同，其中文件读写部分考验的是磁盘性能，在计时中都已去除。

当K = 5时，算法整体的执行时间如图4.6所示。从图中可以看到，GPU算法整体运行效率的优势较为明显。此时 K 较小，GSNN算法经过评估，在第二步使用了并行规约的方式。a1a数据集下，同CPU对照算法相比，加速比为51.5；同ANN-Brute算法相比，加速比为24.4。在a2a数据集下，同CPU算法相比，加速比为45.6，同ANN-Brute算法相比，加速比为20.3。

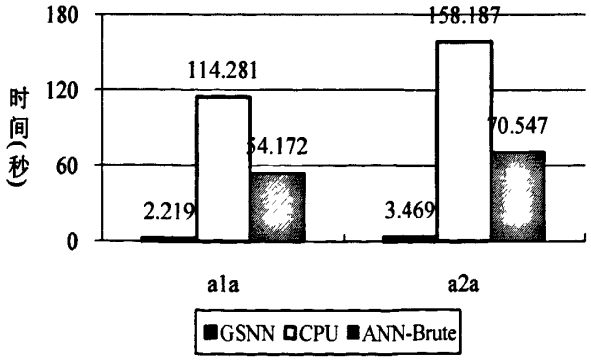

图 4.6 算法整体执行时间(K = 5)

下面分析对算法各个步骤的执行时间进行对比。由于ANN-Brute算法整体过程与其余两种略有不同，现将GSNN和CPU两种算法进行比较。其中距离计算部分的执行时间记为 t_1，最近邻的选择记为 t_2，预测分类标号时间记为 t_3。表4.1展示了各步骤的耗时明细。

表 4.1 各步执行时间(K = 5)

	t_1 (秒)	t_2 (秒)	t_3 (秒)	$t_1/(t_1+t_2+t_3)$ (%)
a1a GSNN	0.828	1.375	0.016	37.31%
a1a CPU	81.954	32.312	0.015	71.71%
a2a GSNN	1.156	2.297	0.016	33.32%
a2a CPU	113.639	44.532	0.016	71.84%

经由上表可知，KNN算法的决定分类标号的第三部分比较简单，执行时间极短，几乎可忽略不计。前两部分占据了绝大部分时间。本文所提出的GSNN算法在第一步采用了分块策略，分块的边长为16，在如此高维的数据集下得到了较好的性能表现。此部分的加速比如单独计算，达到了九十余倍，足以证明算法的有效性。

第二步的性能提升则较之略为逊色。此处实际上是5次并行规约和CPU快速排序的性能比较。此时后者的执行时间是前者的二十倍左右，GSNN算法性能基本还是令人满意的。但并行规约的次数为 K，可以预测，t_2 随着 K 的增大呈线性增长。

整体来看，第一步比第二步的算法复杂度更高。在CPU串行算法中，第一步占据大部分执行时间。本文算法在GPU上执行，第二步由于复杂度较低且程序分支较多，而转化为性能瓶颈。

当 K 逐渐增大时，GSNN 算法在第二步将会采用并行基数排序的方式。图4.7是K = 20的情况下三种算法的比较。

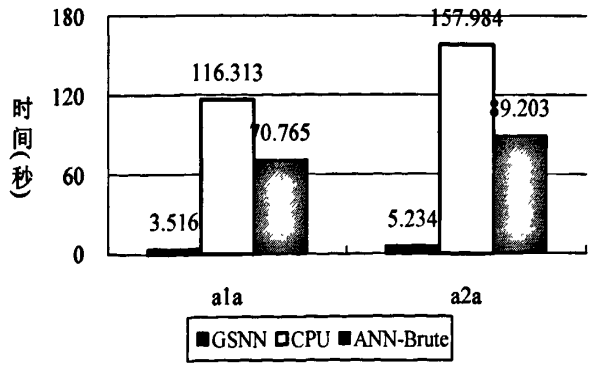

图 4.7 算法整体执行时间(K = 20)

此时算法在第二步经性能评估后使用了排序的方法。a1a数据集下，同CPU对照算法相比，加速比为33.08；同ANN-Brute算法相比，加速比为20.12。在a2a数据集下，同CPU算法相比，加速比为30.18，同ANN-Brute算法相比，加速比为17.04。各步执行时间如表4.2所示。CPU对照算法对 K 值不敏感，故此处同表4.1相比差距极小。GSNN算法也仅在第二步的执行时间上同表4.1有所差异。这个步骤如果采用并行规约方法，时间将会是 $K = 5$ 时的四倍，因此基数排序能起到更好的效果。

表 4.2 各步执行时间($K = 20$)

	t_1 (秒)	t_2 (秒)	t_3 (秒)	$t_1/(t_1 + t_2 + t_3)$ (%)
a1a GSNN	0.812	2.688	0.016	23.09%
a1a CPU	83.327	32.969	0.017	71.64%
a2a GSNN	1.125	4.093	0.015	21.49%
a2a CPU	113.649	44.317	0.018	71.94%

算法的分类准确率比较如图4.8所示。各种方法的准确率区别不大，主要在于KNN算法本身优良的特性。结果的略微差异，仅是由于在规约或排序一步，离测试点距离完全相同而类标号有差异的各点被参数 K 的不同分割所导致。

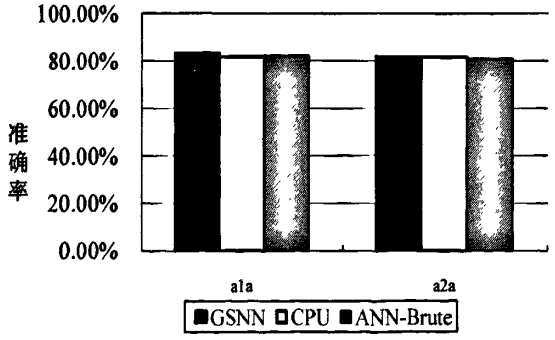

图 4.8 分类准确率

总之，本章所提出的基于GPU的分块最近邻GSNN算法，充分利用了GPU多级存储器体系结构，发挥了GPU的运算处理能力，使得计算过程相比CPU串行算法得到了很大的性能提升，对 K 最近邻算法在高维度、大数据量下的应用具有参考价值。

第五章　基于CUDA的SVM训练算法

5.1　引言

支持向量机的分类是一个解决二次规划问题，算法的复杂度比较高，在大数据集下的训练效果不甚理想。SMO序贯最小优化算法等方法对支持向量机进行了一定的改进，也使得大数据量的学习成为可能，但算法的执行性能问题仍非常突出。本文所提出的基于GPU的大规模数据并行SVM分类算法GMP-CSVC(GPU Based Massively Data Parallel C-SVC)算法，基于CUDA架构，在较低的成本与功耗条件下使得SVM算法性能的得到高倍提升。

5.2　算法思想

5.2.1　求解方法

基本的SVM分类算法所要求解的问题为：

$$\begin{aligned} \min_{W,b,\xi} \quad & \frac{1}{2}W^TW + C\sum_{i=1}^{\ell}\xi_i \\ s.t. \quad & y_i(W^T\Phi(x_i)+b) \geq 1-\xi_i \\ & \xi_i \geq 0 \\ & i = 1,\cdots,\ell \end{aligned}$$

这是一个最优化问题。为求解此问题，可根据拉格朗日理论将其转化为相应的对偶问题 [45]。其拉格朗日函数为：

$$L(W,b,\alpha) = \frac{1}{2}W^TW - \sum_{i=1}^{\ell}\alpha_i[y_i(W^T\Phi(x_i)+b)-1] \tag{5.1}$$

其中$\alpha_i \geq 0$是拉格朗日乘子。对此拉格朗日函数分别求W和b偏导，然后置零，得：

$$\frac{\partial L(W,b,\alpha)}{\partial W} = W - \sum_{i=1}^{\ell}y_i\alpha_i\Phi(x_i) = 0 \tag{5.2}$$

$$\frac{\partial L(W,b,\alpha)}{\partial b} = \sum_{i=1}^{\ell}y_i\alpha_i = 0 \tag{5.3}$$

将式子5.2和5.3分别代入到原始的拉格朗日函数（式子5.1），得：

$$\begin{aligned} L(W,b,\alpha) &= \frac{1}{2}W^TW - \sum_{i=1}^{\ell}\alpha_i[y_i(W^T\Phi(x_i)+b)-1] \\ &= \frac{1}{2}\sum_{j=1}^{\ell}\sum_{i=1}^{\ell}y_iy_j\alpha_i\alpha_jK(x_i,x_j) - \sum_{j=1}^{\ell}\sum_{i=1}^{\ell}y_iy_j\alpha_i\alpha_jK(x_i,x_j) + \sum_{i=1}^{\ell}\alpha_i \\ &= \sum_{i=1}^{\ell}\alpha_i - \frac{1}{2}\sum_{j=1}^{\ell}\sum_{i=1}^{\ell}y_iy_j\alpha_i\alpha_jK(x_i,x_j) \end{aligned}$$

故求最大间隔超平面，实际上就是将问题转换为：

$$\begin{aligned} \max \quad & W(\alpha) = \sum_{i=1}^{\ell}\alpha_i - \frac{1}{2}\sum_{j=1}^{\ell}\sum_{i=1}^{\ell}y_iy_j\alpha_i\alpha_jK(x_i,x_j) \\ s.t. \quad & \sum_{i=1}^{\ell}y_i\alpha_i = 0 \\ & \alpha_i \geq 0 \\ & i = 1,2,\cdots,\ell \end{aligned}$$

根据Karush-Kuhn-Tucker条件 [45]，此问题的解α^*满足：

$$\alpha^*[y_i(W^{*T}\Phi(x_i)+b^*)-1] = 0 \qquad i = 1,\cdots,\ell$$

结论就是落在分类边缘上，同超平面距离为1，也就是支持向量的拉格朗日乘子可以不为0，其它点对应的α^*都为0。根据原始约束条件：

$$W \cdot X + b = 0$$

相应的b^*也通过如下方法求得：

$$b^* = -\frac{\max_{y_i=-1}(W^{*T}\Phi(x_i)) + \min_{y_i=1}(W^{*T}\Phi(x_i))}{2}$$

对于一个未知标号的向量X，可以通过判定函数来预测其类标号：

$$y = \mathrm{sgn}(\sum_{i=1}^{\ell}y_i\alpha_i^*K(x_i,x)+b^*)$$

为了计算方便，定义半正定矩阵Q，$Q_{ij} = y_i y_j K(x_i, x_j) = y_i y_j \Phi(x_i)^T \Phi(x_j)$。这也是一个海森矩阵(Hessian Matrix)。另有$e = (1, 1, \cdots, 1)^T$，原问题的对偶问题可表示为：

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2}\alpha^T Q\alpha - e^T\alpha \\ s.t. \quad & y^T\alpha = 0 \\ & 0 \le \alpha_i \le C \\ & i = 1, 2, \cdots, \ell \end{aligned}$$

求解此凸二次规划问题，最简单的数值解方法是梯度法。这个算法从一个满足条件的初始估计值α_0开始算起，沿着最快上升的路径迭代更新。每次更新的元素由算法决定，更新的步长则是固定的。在第$t+1$次迭代，更新的量可表示为：

$$\Delta\alpha_i^t = \eta\frac{\partial W(\alpha^t)}{\partial\alpha_i}$$

其中参数η也叫学习率。如果选择合适的参数，将会使得目标函数值单调下降，直到算法收敛。η过大会使得出现震荡，过小又会使得迭代次数过多。

除此之外还有其它一些方法。总之，早期的简单方法需在内存中完整保存$l \times l$大小矩阵的Q，且存在收敛性能不好的问题，无法实现高效的求解，因而SVM算法的优化是自其产生以来长期的热点。

5.2.2 算法优化策略

SVM的分解算法 [46,47]是将大规模的问题拆分成小的部分，每次迭代执行仅仅更新小规模的数据，即活动工作集中的元素，因而无需保存完整核矩阵，实现较大规模的分类。

1999年文献 [30]提出的一种序贯最小优化(SMO, Sequential Minimal Optimization)算法，利用了分解的思想，并将这种思想推向极致。此方法每次迭代仅仅更新由两个元素所组成的子集，每更新一个乘子，必有另一个得到相应调整，约束得到最大限度地保证。

SMO算法采用启发式的工作集选择策略，在迭代之初选择待更新的两个元素。第一个点的选择方法是遍历所有参数满足的点，选择最违反Karush-Kuhn-Tucker条件的点作为候选。第二个点的选取依据是尽量使得乘子的更新能使目标函数朝着优化的目标产生更大的变化，从而达到更快收敛的目的。

算法每次迭代仅仅求两个元素的解析解，对相应乘子进行优化更新，操作较少。虽然迭代次数较多，但整体上的速度还是比传统方法有了革命性的提升。

算法的大体框架如算法所示：

算法 4 SMO算法

Require: 训练数据集X，标号$y_i, i \in 1,2,\cdots,\ell$

```
读取文件
初始化α_i = 0, f_i = −y_i
loop
    计算bHing, bLow, iUp, iLow
    更新α_i, α_j
    更新f
    if 满足KKT条件 then
        break
    end if
end loop
计算b
保存模型
```

SMO算法在计算的过程中，矩阵 Q 无需完整保存在内存里，这个特性对大数据量的训练起到至关重要的作用。其贡献不知使算法成为可行，而且根据启发性的选择策略，大大降低了算法的运算量。对SMO算法也有一些进一步的优化和改进 [48,49]。

SMO方法把问题的分解发挥到极致，每次只求解两个问题的子集。进而更新 α 的时候，仅需要将少数使用到的矩阵 Q 中的元素进行计算，也避免了巨大的内部存储器开销，大数据集下是的算法可行。计算核向量总之是有一定的开销，所以如果在存储空间允许的情况下，能够对矩阵 Q 的一部分进行缓存，则能够进一步降低运算复杂度。为此可根据实际情况维护一块存储空间，如果待计算核向量的点乘结果已经在缓存中命中，则省却了一些计算过程，仅当未命中时才需要进行计算。缓存的维护方法也需要考虑。

在计算过程中，候选的支持向量就是对应的拉格朗日乘子 α_i 不为0的样本，这只在样本集中占据较小的部分。Shrinking优化就是对样本集做缩减，尽量减少参加运算的样本总数。Shrinking压缩算法每间隔一定的迭代次数后运行一次。实现的具体

方法，简而言之就是在根据上一轮迭代更新的 α_i 值，认为当 $0 < \alpha_i < C$ 时，该样本可以不参加后面的运算。也就是说，使用启发式的算法进行两个候选待处理向量的选择并计算的全过程，只在非支持向量和对应拉格朗日乘子的值取上界的支持向量中下进行，从而在一定程度上减少了运算的复杂度。

另外，原始的SVM算法不支持增量学习，文献 [50–52]提出了增量学习的支持向量机改进算法。

5.2.3 并行SVM算法

采用并行的策略，将 SVM 的训练算法的开销分散到多个处理节点进行计算是解决其速度问题的有效方法 [53,54]。

其中一种并行化的方法是从串行算法演化而来，在每一步迭代中将计算分发到多个处理器中进行。例如文献 [55]提出将 SMO 算法中的 KKT 条件判定过程所涉及到的运算分派到多个并行计算节点完成。文献 [56]采取更适于并行化的变梯度投影法对工作集进行优化。

这类算法在整体上还是基于串行算法的框架，在传统的并行体系结构上通讯开销过大，效果并不理想。

另一类的方法是将训练样本进行分割，在每个计算节点上训练本地 SVM 分类器，再按照特殊设计的并行学习结构获得 SVM 的全局最优解。其中Graf在2005年提出的Cascade SVM算法 [57]，是一种基于二叉级联的结构。将数据分割后每个节点训练所生成的支持向量集两两合并，再作为新的训练集进行训练，从而逐步提高本地分类器的数据泛化能力，反复进行直到数据完全合并为止。文献 [58,59]针对二叉级联结构反馈速度比较慢的问题进行了改进，提出一种带交叉反馈的二叉级联结构，提高了 Cascade SVM 算法的效率。

此类方法的问题是，训练过程的整体运算量被大幅度提高了，同时在数据划分时，如果高度不均匀或不对称的会对训练结果产生很大的不利。更重要的是，随着运算的逐步合并迭代执行，节点的利用率呈迅速下降的趋势。

5.3 GMP-CSVC训练算法

本文提出的基于 CUDA 的大规模数据并行 SVM 训练算法 GMP-CSVC，是在序贯最小优化策略的基础上，针对 GPU 的体系结构，对传统的训练算法进行了改进。

算法采用了基于数据并行的细粒度的划分方法，充分利用了 GPU 的多层存储器模型，并采用了一定的优化策略，使得算法完全符合了 GPU 运行的要求。

5.3.1 算法流程

GPU 的特点是为了在当前的半导体技术下实现更高的运算性能，处理器的运算单元数量庞大，而指令解码、分支预测和缓存等部分被最大限度地简化或取消，所以在执行并行运算的全过程中，需要仔细考虑每一个部分，应用 CUDA 的线程模型合理的进行数据分割。

同 LIBSVM 等分类器采用的用链表存储向量的方式不同，本文采用了矩阵的方式存储各个向量。向量矩阵每行存储一个向量元组，各个偏移量的位置存放相应维上的度量值。这种方法应用在稀疏的数据集上，会保存大量的 0 元素，占用一定的存储空间并增加运算量。但在相对密集数据集上的优势不言而喻。同时在 GPU 运算中，为了实现流多处理器中各个运算单元所执行指令的高度一致，矩阵式存储也是最优的策略。最终的测试结果表明，本文算法即使在较稀疏的数据集上，也得到了很高的加速比性能。数据集的密度越高，算法的优势也就更加明显。

为了减少同一个流处理器内线程的分支，运算过程也都是仔细考虑和优化的。很多计算步骤经过了演算，采用了不同于传统串行的运算方法。共享锁在GPU上的开销尤为巨大，算法也尽量地避免了多个线程同时对同一数据读写操作。算法的整体流程如算法5所示。

5.3.2 算法说明

1. 初始化部分，首先是初始化长度为 ℓ 的拉格朗日乘子数组 alpha。其初始值首先要满足约束，故显而易见，最简单的是全为 0 的组合，此时没有支持向量。

根据KKT最优化条件，定义长度为 ℓ 的数组f，每个元素的计算采用如下方法：

$$f_i = \sum_{j=1}^{\ell} \alpha_j y_j K(x_i, x_j) - y_i$$

由于 alpha 初始为全为 0 的数组，故 f 的初始值为训练集中个向量标号取反。

参加每一轮计算的两个向量元素记为 iUp，iLow，分别来自于如下两个集合：

$$I_{up}(\alpha) = \{i | \alpha_i < C, y_i = 1 \text{ or } \alpha_i > 0, y_i = -1\}$$

$$I_{low}(\alpha) = \{i | \alpha_i < C, y_i = -1 \text{ or } \alpha_i > 0, y_i = 1\}$$

算法 5 GMP-CSVC-Train

1: 变量初始化$\alpha_i = 0, f_i = -y_i$
2: iLow为第一个-1标号的点下标，iUp为第一个+1标号的下标
3: 初始化bUp=-1, bLow=1
4: **loop**
5: 计算eta = $\phi(x_{iUp}, x_{iUp}) + \phi(x_{iLow}, x_{iLow}) - 2 \times \phi(x_{iUp}, x_{iLow})$
6: alpha2new[iLow]= alpha[iLow] + y[iLow] × (bUp - bLow) / eta
7: alpha1new[iUp]= alpha[iUp]- y[iUp] × (bUp - bLow) / eta
8: 将alpha值规约到[0, C] 区间
9: **if** bLow <= bUp + 2t **then**
10: break
11: **end if**
12: 更新f[i]
13: 寻找新的iUp, iLow
14: **end loop**
15: rho = (bLow + bUp)/2
16: 输出模型。

初始化时可以令 iLow 为第一个-1标号的点下标，iUp 为第一个+1标号的下标。bUp 和 bLow 变量是帮助对KKT条件进行判断的变量。初始化 bUp=-1, bLow=1

2. 进入迭代过程，首先对α进行更新，具体方法为：

$$\alpha_{i_{low}}^{new} = \alpha_{i_{low}}^{old} + y_{i_{low}} \frac{(b_{high} - b_{low})}{K(x_{i_{high}}, x_{i_{high}}) + K(x_{i_{low}}, x_{i_{low}}) - 2K(x_{i_{high}}, x_{i_{low}})}$$

$$\alpha_{i_{high}}^{new} = \alpha_{i_{high}}^{old} - y_{i_{low}} y_{i_{high}} \frac{(b_{high} - b_{low})}{K(x_{i_{high}}, x_{i_{high}}) + K(x_{i_{low}}, x_{i_{low}}) - 2K(x_{i_{high}}, x_{i_{low}})}$$

这个过程占据了最多的运算时间。其中核函数的计算是整个算法中耗费运算时间最多的子问题之一。

3. 根据 KKT 条件对 bUp 和 bLow 变量进行判断，如果满足 KKT 条件，则意味着寻找到了满足约束条件的最优解，即 α 组合，跳出循环。否则继续进行计算。

4. 对 f 进行更新。在 SMO 优化算法下，每次迭代只是计算包含两个元素的子问题，所以 f 并非每次都需要从头重新计算，而是仅需要对 iUp 和 iLow，两个元素的优化而产生的变化量增加到原值中即可：

$$f_i^{new} = f_i^{old} + \Delta\alpha_{i_{high}} y_i K(x_i, x_j) + \Delta\alpha_{i_{low}} y_i K(x_i, x_j)$$

其中：

$$\Delta\alpha_{i_{high}} = \alpha_{i_{high}}^{new} - \alpha_{i_{high}}^{old}$$
$$\Delta\alpha_{i_{low}} = \alpha_{i_{low}}^{new} - \alpha_{i_{low}}^{old}$$

5. 选择参加下一轮计算的两个向量。采用启发式的向量选择方法，即 iUp 和 iLow 的选取依据是使得对下一轮运算所产生的变化最大的两个元素。即

$$i_{up} = \arg\min_t\{f(\alpha)_t | t \in I_{up}(\alpha)\}$$
$$i_{low} = \arg\max_t\{f(\alpha)_t | t \in I_{low}(\alpha)\}$$

6. 计算两个元素对应的b值：

$$b_{up} = \min\{f(\alpha)_t | t \in I_{up}(\alpha)\}$$
$$b_{low} = \max\{f(\alpha)_t | t \in I_{low}(\alpha)\}$$

7. 重复循环，直到满足 KKT 条件为止。

8. 计算 b，方法为：

$$b = -\rho = \frac{b_{low} + b_{up}}{2}$$

5.3.3 核函数的计算

在算法5的第(5)步，需要计算两个向量的点乘结果。也就是计算式子：

$$K(x_{i_{high}}, x_{i_{high}}) + K(x_{i_{low}}, x_{i_{low}}) - 2K(x_{i_{high}}, x_{i_{low}})$$

的函数值。其中当$K(x_i, x_j) = \mathrm{x}_i^T x_j$，称为线性核函数。

为了将原本线性不可分的数据映射到线性可分或近似线性可分的高维数据空间，从而实现分类器更好的数据泛化能力，引入了更为复杂的核函数，常见的包括以下几种：

多项式核函数：$K(x_i, x_j) = (\gamma \mathrm{x}_i^T x_j + r)^d, \gamma > 0$

径向基（高斯）核函数：$K(x_i, x_j) = \exp(-\gamma\|\mathrm{x}_i - x_j\|^2), \gamma > 0$

S型核函数：$K(x_i, x_j) = \tanh(\gamma \mathrm{x}_i^T x_j + r)$

算法5中，这四种核函数都得到了实现。核函数的引入使得运算量进一步增加。将此计算交由GPU执行可以得到很好的性能。

算法 6 K1-Polynomial-Kernel

Require: 核函数参数γ，r，d，向量起始指针p，向量结束指针pEnd，向量矩阵宽度pitch

```
accumulant = 0
result = 0
while p < pEnd do
    value = *p
    accumulant += value × value
    p += pitch
end while
accumulant = accumulant ×γ + r
result = accumulant
for degree = 2 to d do
    result = result × accumulant
end for
写入结果result到相应的位置
```

首先讨论最简单的 $K(x_i, x_i)$ 计算。在径向基核函数中，此值恒等于1。多项式核函数下的每个线程的算法如算法6所示。

数据划分的方式是根据向量来划分的，每个线程处理一个向量，在循环迭代的过程中依次读取各个维度的数据进行计算。算法和程序在执行中的临时变量数目，是任务网格的划分依据。此算法非常简单，涉及的临时变量不多，将流多处理器同时运行的线程数设置成最大值，有利于隐藏存储器的访问时延，提高整体性能。故将线程块的大小设置成256，这样每个流多处理器将同时执行3个线程块，总共768个线程同时由指令调度器来调度。这是当前支持的最大值。

为了达到协同访问的要求，向量矩阵的存储采用如图5.1所示的存储方案，矩阵的每一列是一个向量，矩阵的宽度在向量数量的基础上也经过了针对64字节的对齐处理。因而在数据读取的过程中，一个线程块所同时请求的数据每次都保证都在一行中，且符合协同访问的条件。

$K(x_i, x_i)$ 计算完成后，可以保存在一个长度为 ℓ 的数组中，在后面的运算中此计算可以不在进行，直接读取结果。而 $K(x_i, x_j)$ 的运算结果也就是核函数矩阵，完整保存需要 $O(\ell^2)$ 空间复杂度。ℓ 的数量级在几万个的情况下，完整保存核矩阵所需的

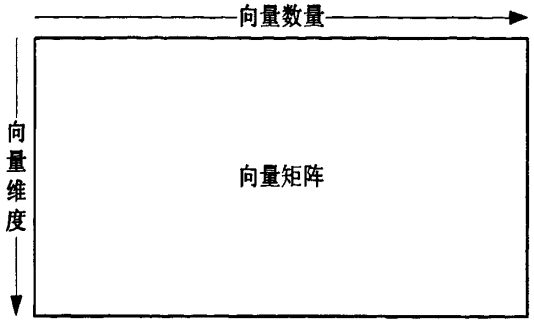

图 5.1 向量矩阵的存储方式

空间就已经达到GB的数量级。因此大数据集的训练，核函数每次都需要重新计算。

算法7是采用径向基核函数计算 $K(x_i, x_j)$ 时每个线程的算法。算法数据划分的方式是根据数据维度而进行的。每个线程处理一个维度上的数据。对于数据维度大于 blockDim.x 的情形，则所有线程计算的维度递增 blockDim.x，循环多次直到完成。最终在线程块的共享存储器中运行一次并行规约算法求和后，由 sharedTemps[0] 计算出最终结果。线程块的大小仍设置成256。此算法的特点是使用了共享存储器，且尽量避免了 bank 冲突，在计算高维数据时算法有很好的性能。如果数据维度低于256，则运算时间无差异。

5.3.4 避免线程内分支

由于 GPU 中指令预测的部件被彻底精简，流多处理器以线程束为单位堆线程进行指令控制，因而程序分支将会导致性能的下降。在 CUDA 体系结构下，避免程序分支有很多方法和技巧。

IF 和 SWITCH 语句分支最常见，也是对性能影响最大的。在算法设计时，应尽一切可能避免。在任务分割和个线程工作定义阶段，就应对程序分支的问题进行考虑。对于实在不可避免的分支，应考虑将条件判断转化为计算指令，将不同的分支统一到同样的计算过程中去。

另外，充分利用模板(Template)也是 CUDA 下常见的技巧。模板是 C++ 的一个重要特性，使用参数化的类型或变量实现泛型化的编程。在算法5的具体实现中，大量应用到参数化变量的模板函数。这样，由不同参数值导致的程序分支在编译器的帮助下消除。编译器为不同的模板参数值生成不同的函数示例，程序执行时就不再

算法 7 K2-RBF-Kernel

Require: 核函数参数γ，向量A起始指针pa，向量结束指针paEnd，向量B其实指针pb，向量矩阵宽度pitch，

```
pa += threadIdx.x
pb += threadIdx.x
初始化共享存储器sharedTemps[threadIdx.x] = 0
while pa < paEnd do
    diff = (*pa) - (*pb) ..
    sharedTemps[threadIdx.x] += diff × diff
    pa += blockDim.x
    pb += blockDim.x
end while
syncthreads
对sharedTemps数组并行规约求和
if threadIdx.x = 0 then
    sharedTemps[0] = exp(sharedTemps[0] × gamma)
end if
输出sharedTemps[0]
```

包含分支判断，大大提高了效率。

例如四种核函数的计算方式完全不同，将调用何种核函数的条件判断语句写入每个线程的行为中，是极为低效的。而此条件作为模板参数，就在编译阶段将不同类型核函数的代码进行了分离。再例如，在不同条件下，流多处理器中的共享内存会有不同的分配方式。此条件在可经分析归并后，由不同的模板参数值来调用函数实例，取得了很好的效果。

另外一种分支判断循环结束条件的判断。线程进行循环条件判断的开销也非常大，将循环次数可预知循环体展开是减少迭代次数，进而减少分支的有效方法。例如线程块内的求和中的最后几步，线程数目仅有几种组合。将其展开会得到更好的性能。另外，CUDA 也提供了 #pragma unroll 命令，由编译器协助循环展开的操作。

5.3.5 并行规约在求解依赖问题中的应用

算法5的第11步，是选取下一轮迭代过程中所要优化的两个元素，下标分别为

iUp 和 iLow。其中 iUp 为：

$$i_{up} = \arg\min_t\{f(\alpha)_t | t \in I_{up}(\alpha)$$

也就是在 UP 集合当中，寻找 f 值最小的一个元素，其下标就是所求的结果。而这个最小值，记为 bUp：

$$b_{up} = \min\{f(\alpha)_t | t \in I_{up}(\alpha)\}$$

bUp作 为循环中止条件的判断依据，并在可能的下一轮迭代中参与更新 α 的计算。与此对应的 iLow 和 bLow 为：

$$i_{low} = \arg\max_t\{f(\alpha)_t | t \in I_{low}(\alpha)\}$$
$$b_{low} = \max\{f(\alpha)_t | t \in I_{low}(\alpha)\}$$

这种求最值的计算，可以通过转换，采用并行规约算法求解。

以 iUp 和 bUp 的计算为例。在之前求解 f 数组的计算过程中，采取的线程划分方法是每个线程计算一个 f 数组的值。为了提高计算效率，减少重复的数据拷贝和线程分割，可就地将新计算好的f值放入共享存储器中的数组 sharedF 中。同时定义另一个同样大小的位于共享存储器上的数组 sharedI，存放对应的位置下标。

随后，各个线程判断对应的元素是否属于 UP 集合。如果不属于，则将相应的 f 值替换为 +INF，从而在保证在执行最小值的规约中不将其包含进来。此步骤会导致线程束内的分支，但为了分支总体上最少，此处是为不得已而为之。

下一步，在线程块内执行一次最小值规约算法。在以 sharedF 数组的值为依据进行规约的过程中，同时对 sharedI 数组进行同步交换调整，保证二者对应关系一致。

再下一步，定义位于全局存储器上的，长度与线程块数目一致的两个数组，globalF 和 globalI。每个线程块拷贝 sharedF[0] 及 sharedI[0] 到其相应的位置。

最后，对 globalF 和 globalI 进行一次最小值规约，即得到 bUp 和 iUp 的值。

iLow 和 bLow 的也同理计算。区别在于不属于 LOW 集合的元素对应的f值以-INF替换，且执行的是最大值规约算法。

5.4 实验结果与分析

本文将算法在两种不同的 GPU 环境下分别作了测试，以验证算法的有效性。主机1的配置环境为：

- CPU：Intel Pentium D 820 2.8GHz
- 内存：64bit 1.5GB DDRII 667MHz
- GPU：GeForce 9600GSO 96SPs 1.625GHz
- 显存：192bit 384MB GDDRIII 1GHz
- OS：Windows XP 32bit

主机2的配置环境为：

- CPU：Intel Xeon 5520 2.26GHz
- 内存：192bit 12GB DDRIII 1333MHz
- GPU：Tesla C1060 240SPs 1.3GHz
- 显存：512bit 4GB GDDRIII 800MHz
- OS：CentOS 5.3 64bit

为了使测试具备合理性与可信性，本文采用的数据集都来自于LIBSVM的预处理数据，分成+1和-1的两类。其中 Adult 数据集与第四章所用相同，原始数据集来自 UCI Machine Learning Repository，预处理成32561个123维数据。IJCNN1 原始数据集来自 Ford Research Laboratory，作为 IJCNN 2001 神经网络竞赛的数据集，包含49990个22维数据。Web数据集原始集合来自文献 [30]对SMO算法的测试数据，包含49749个300维数据。

CPU 对比测试采用 LIBSVM 的算法。对三个数据集采用算法5和 LIBSVM 算法在主机1上进行测试，结果如图5.2所示。

在测试中统一采用 RBF 核函数，核函数参数 γ 为数据维度的倒数，惩罚因子 $C = 10$。图5.2显示了 GMP-CSVC 算法比较出众的性能。在这三个不同数据集下，都取得了比较高的加速比。衡量算法执行结果的几个指标如表5.1所示：

表 5.1 不同数据集下算法的几个指标

数据集	LIBSVM		GMP-CSVC		加速比
	迭代次数	支持向量数	迭代次数	支持向量数	
Adult	21807	11465	31237	11466	19.46488
IJCNN1	6503	7082	19138	7082	17.44208
Web	7733	2446	12388	2383	7.42767

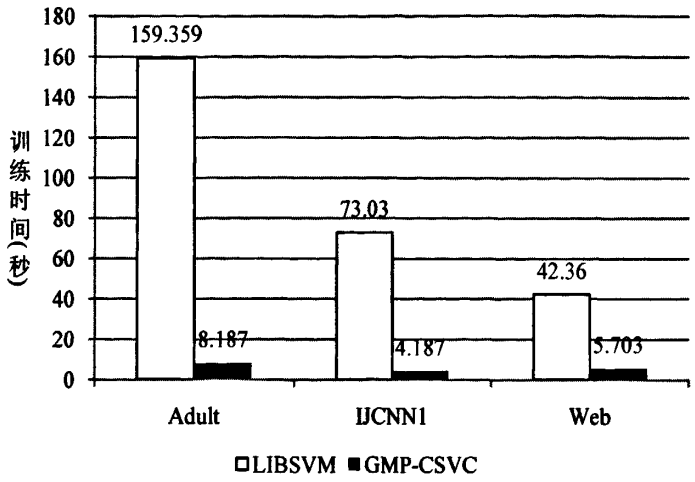

图 5.2 不同数据集下的训练时间

由表5.1可以看出，GMP-CSVC 算法与 LIBSVM 算法最终所计算得到的支持向量数目非常相似。对于迭代次数的差异，是由于二者的对 KKT 终止条件容忍偏差的掌握略有不同，以及运算具有一定的随机性所造成的，也在可接受的范围内。

GMP-CSVC 算法对四种核函数均予以实现，可由用户根据需要进行选择。四种核函数在 $C = 1$ 时的执行时间对比如图5.3所示。

综合运算效率和分类效果，一般认为在大部分情况下，径向基核函数在这四者中占优。

以上数据均是算法的整体执行时间。对并行算法进行评估，计算/通讯比也是很重要的一个方面。故在测试中，对计算和通讯分别计时，以更进一步地分析。同时，为了验证本文算法的可扩展性，在配置不同数目流处理器的 GPU 上进行测试，以及在同数据集下不同数目的样本上测试也是必要的。

表5.2给出了 GMP-CSVC 算法在两种主机环境下的测试结果。其中测试数据集来自 Adult 集合，a1a至a9a是由小到大对原始测试集和的拆分。主机1的 GPU 上包含有96个运行频率为 1.6GHz 的流处理器。主机2的 GPU 上配置了主频为 1.3GHz 的240个流处理器。训练时间是算法整体的执行时间，由计算时间和通讯时间两部分组成。

由表5.2可以看出，各个样本集下，两个平台的计算时间基本上都是随着样本的

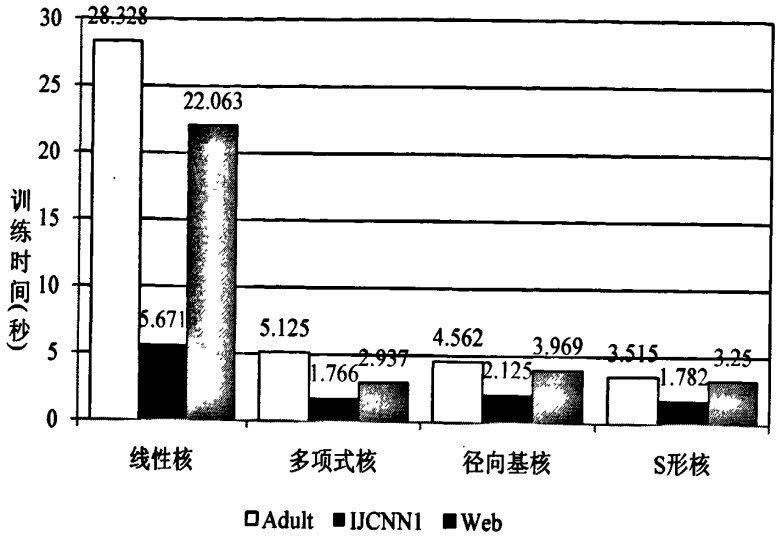

图 5.3 不同核函数下的训练时间

数目呈线性递增的趋势。这反映了算法的复杂度特性，以及具备较好的收敛性能。

其次，两个平台上计算时间的差异，原因在于是两个 GPU 运算性能的不同。性能与流处理器的频率、数目之积成正比。表中最后一列显示后一个平台相对于前者性能的提升，波动很小，反映了本文算法较好的可扩放性。

另外，表中训练时间减去计算时间，即是通讯时间。从表中可以得出，无论样本集数量多少，主机1平台通讯时间基本保持在0.3秒左右，主机2平台则在1.6秒左右。对此情况，本文也经过了仔细调校和分析。GPU 环境下的通讯时间主要包括显存的分配，以及内存与显存间数据的拷贝时间。对于后者，实际上是内存与显存间带宽的一种体现。此带宽经过仔细测试，可保持在数个GB/s的数量级。因此，在 Adult 数据集的测试中，数个到数百个MB数量级的数据拷贝所产生的微小时间差，被淹没到其它的时间中未得以体现。显存分配在通讯时间中占据主要比重。显存的分配由 cudaMalloc() 和 cudaMallocPitch() 等函数执行，由驱动程序根据用户指定的容量分配显存空间，并返回指针。经仔细测试，分配一块显存的时间比较固定，主机1都在20毫秒左右，主机2维持在100毫秒左右。本文算法总共需要执行十几次显存分配的操作，因而有了表5.2中的测试差异。

造成主机2环境下显存分配时间较长的原因，首先在于存储器调度算法上。在主机2中4GB的显存下，寻找与用户指定长度最适合的未分配空间，显然产生的开销要

表 5.2 两种GPU在各种数量样本上的结果

数据集	训练集数目	主机1		主机2		计算性能提升
		训练时间 T1(秒)	计算时间 T1'(秒)	训练时间 T2(秒)	计算时间 T2'(秒)	T1'/T2'
a1a	1605	0.484	0.203	1.76	0.16	1.27
a2a	2265	0.594	0.313	1.85	0.24	1.30
a3a	3185	0.672	0.359	1.88	0.28	1.28
a4a	4781	0.875	0.547	1.92	0.40	1.37
a5a	6414	1.109	0.813	2.15	0.54	1.51
a6a	11220	1.766	1.453	2.58	1.02	1.42
a7a	16100	2.718	2.422	3.46	1.80	1.35
a8a	22696	4.657	4.359	4.48	2.76	1.58
a9a	32561	8.187	7.859	7.09	5.39	1.46

比主机2中384MB的环境下大。其次，驱动的更新和进一步完善，对此时间的缩短有较大的帮助。最新硬件在64位Linux下驱动的完善度、支持度和更新速度，普遍要落后于Win32平台。而这已超出本文算法的讨论范围。

总之，本章基于SMO序贯最优策略，提出了CUDA平台下适合GPU体系结构与线程模型的GMP-CSVC分类算法。算法在核函数的计算、拉格朗日乘子的更新、KKT条件的判断等步骤中，充分发挥了GPU的优势，并尽量避免了程序分支，取得了较好的效果。测试数据也证明了算法的有效性和通用性。

第六章 GMP-nuSVC分类器的实现

6.1 引言

SVM分类算法在实际应用中，需要在训练阶段进行交叉验证和参数寻优。通过对每个参数在其约束范围内以一定间隔进行尝试，寻找到最优的参数组合，从而生成分类模型。需要寻优的参数除了核函数参数外，还包括惩罚因子 C。参数寻优一般采取网格搜索的方式，n 个参数对应 n 维空间的网格，每个点执行一次SVM训练算法生成模型，并用判定算法评估模型的准确率，找到准确率最高的一种参数组合，用以对全部训练样本进行训练，生成最终模型。

因此，SVM分类器的构造所耗费的成本被进一步提高了。在参数寻优的过程中，每个参数的在其范围内尝试的间隔，也要从计算量方面进行权衡。对于惩罚因子 C 等限制较小的参数，为了使得搜索空间尽量全面，搜索间隔只得以指数的梯度递增。同时，随着 C 的增大算法收敛的时间也越来越长。

为了解决参数C的这些问题，文献 [60]提出了一种新的SVM模型，并从理论角度证明其优势所在。然而新模型目前只有近似解法而没有精确解法。本章在这些理论和工作的基础之上，提出了GMP-nuSVC(GPU Based Massively Data Parallel ν-SVC)算法。算法训练阶段，是一种ν-SVM在GPU上的近似解法。对于参数寻优所需的另外一个标号判定阶段，GMP-nuSVC采用了一种合理的分块解法，得到了较好的性能。

6.2 GMP-nuSVC训练算法

6.2.1 算法思想

ν-SVM是 Scholkopf 等人在文献 [60]首次提出的一种新的支持向量机理论。这种理论不但可以应用在分类，也可推广到回归方法中，文献 [61]和 [62]分别对此作了理论分析。LIBSVM中也实现了一种串行近似求解的方法。从表面上看，ν-SVM只是将原始的C-SVM中的惩罚因子 C 换为 ν，但由此却带来巨大的深层变化。

首先，C-SVM中的惩罚因子 C，在 $C > 0$ 的范围内均可取值。惩罚因子表示对离群点的容忍程度，C 越大表示越重视，越需要寻找一个尽量使得离群点更少的超平面。随着 C 的增大，问题求解所得的超平面会更加弯曲，从而最大限度地将两个

类分开。参数 C 并不是越大越好，过大的 C 表示对错误分类或噪声数据的惩罚力度过大，在训练样本中取得很好的效果，但不一定反映了实际情况，也就是可能会对测试数据预测的正确率有负面影响。这就是过度拟合了训练数据，从而导致分类器的泛化能力下降。因而参数 C 需要在其范围内尽量以小的间隔进行搜索，寻找到最优值。而 ν-SVM中的参数 ν，约束条件是 $0 < \nu < 1$，而在测试集中的两个类数目不均衡时，其上界还要减小。所以对其最优值搜索所需的训练/评估数，大大减小了。

其次，参数 ν 可对支持向量的数目进行控制。从整体上而言，随着 ν 的增大，ν-SVM分类器训练所生成的支持向量数也随之增加。假设训练样本的数目为 ℓ，则 $\nu\ell$ 是支持向量数目的下界，同时也是错误分类向量数目的上界。因此，对生成的分类模型的规模可以有所控制。控制支持向量的数目会使得后面分类标号判定步骤的运算量更少，对训练时间的减少也有较大帮助。

另外，如果 ν-SVM中的变量 $\rho > 0$ 的情况下，与C-SVM中 $C = 1/(\rho\ell)$ 可推得相同的决策函数。同时，在C-SVM求解完毕之后，可以推得等效的 ν-SVM问题。根据优化后的拉格朗日乘子 α，可以计算

$$\nu = (\sum_{i=0}^{\ell} \alpha_i)/C\ell$$

以此参数 ν 训练 ν-SVM分类器，将会得到基本同样的结果。

根据文献 [61]的讨论，ν-SVM分类算法所要求解的问题：

$$\begin{aligned} \min_{W,b,\xi,\rho} \quad & \frac{1}{2}W^T W - \nu\rho + \frac{1}{\ell}\sum_{i=1}^{\ell}\xi_i \\ s.t. \quad & y_i(W^T\Phi(x_i) + b) \geq \rho - \xi_i \\ & \xi_i \geq 0 \\ & \rho \geq 0 \\ & i = 1,2,\cdots,\ell \end{aligned}$$

将此问题转换为对偶型后，成为：

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2}\alpha^T Q\alpha \\ s.t. \quad & y^T\alpha = 0 \\ & e^T\alpha \geq \nu \\ & 0 \leq \alpha_i \leq \frac{1}{\ell}, (i = 1,2,\cdots,\ell) \end{aligned} \tag{6.1}$$

决策函数与C-SVM问题一致：

$$y = \mathrm{sgn}(\sum_{i=1}^{\ell} y_i \alpha_i^* K(x_i, x) + b^*)$$

对比C-SVM算法所要求解的问题为：

$$\begin{aligned} \min_{W,b,\xi} \quad & \frac{1}{2} W^T W + C \sum_{i=1}^{\ell} \xi_i \\ s.t. \quad & y_i(W^T \Phi(x_i) + b) \geq 1 - \xi_i \\ & \xi_i \geq 0 \\ & i = 1, 2, \cdots, \ell \end{aligned}$$

其对偶型为：

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2} \alpha^T Q \alpha - e^T \alpha \\ s.t. \quad & y^T \alpha = 0 \\ & 0 \leq \alpha_i \leq C, (i = 1, 2, \cdots, \ell) \end{aligned} \tag{6.2}$$

通过对比式子6.1和6.2可看出，ν-SVM 所要求解的问题多了一个不等式约束 $e^T \alpha \geq \nu$。而对这个包含两条不等式约束下的二次规划问题，目前尚未有可操作的精确解法。当前现有的方法，例如 LIBSVM，SVM-Light 等，都是采用近似的解法。本文在实际求解的过程中，也将其转化为 $e^T \alpha = \nu$，将不等式约束转化为等式，然后再行求解。

另外由于大部分情况下，问题的规模普遍比较大，因而第三个不等式约束$0 \leq \alpha_i \leq 1/\ell$中，拉格朗日乘子 α 的上界，也就是 $1/\ell$ 将会比较小。而当前GPU对于IEEE-754浮点小数标准的支持仍是非常完善，非规范化浮点数(Denormalized Number)暂未得到支持。这意味着，0附近的小数的表示精度比CPU要差。因此，为了避免产生精度问题，本文将原始的问题，式子6.2所有约束两边均乘以 ℓ，再对问题的拉格朗日函数的各个变量求偏导，最终本文求解的对偶问题为：

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2} \alpha^T Q \alpha \\ s.t. \quad & y^T \alpha = 0 \\ & e^T \alpha = \nu \ell \\ & 0 \leq \alpha_i \leq 1, (i = 1, 2, \cdots, \ell) \end{aligned} \tag{6.3}$$

问题求解过程得到简化，并提高了GPU上执行的可行性。但是，此时决策函数却相应转变为：

$$y = \mathrm{sgn}(\sum_{i=1}^{\ell} y_i \frac{\alpha_i^*}{\rho}(K(x_i, x) + b^*))$$

为了将GMP-CSVC与GMP-nuSVC两种算法的决策函数统一起来，采用一致的标号判定算法，并尽量将所生成的分类模型合理归并，使得在输出文件格式上尽量保持一致，GMP-nuSVC算法在问题求解完毕之后，对上面式子中 ρ 和 b 两个参数的值进行了转换，重新进行计算，使得GMP-nuSVC在标号判定阶段采用了同GMP-CSVC相同的决策函数。

6.2.2 算法流程

本文所提出的GMP-nuSVC算法如算法8所示。

首先，算法第(1)步是对拉格朗日乘子 α 的初始化。同C-SVC算法不同，ν-SVC增加的一个约束条件 $e^T\alpha = \nu\ell$必 须得到满足。因此，本文采取的初始化方法是对标号为+1的前 $\nu\ell/2$ 个元素所对应的 α 初始为1。同样，标号为-1的前 $\nu\ell/2$ 个元素也初始为1。在这种情况下，如果两个类的标号数目不对称时，要求参数 ν 小于1。具体是 $\nu \le 2m/\ell$，其中 m 为数目较少的一个类的数量。

在算法8第(4)步，是对f的第一轮计算。公式为：

$$f_i = \sum_{j=1}^{\ell} \alpha_j y_j K(x_i, x_j) - y_i$$

此处由于α初始化时有大量的元素不为0，所以要有首轮的计算。具体方法是循环ℓ次对每个元素分别计算，每个f值的计算中，将任务划分成ℓ份，每个线程计算一个核函数，然后累加。

算法8的整个初始化过程将涉及到较多的运算。同时在初始化阶段大量计算核函数的之时，如能将核函数矩阵尽可能地保存起来，对减少后面的运算量会有一定的帮助。因此，算法实现并维护一个核函数矩阵缓存，以达到此目的。

算法第(9)步进入循环后，对每轮迭代所要计算的两个元素的选择以及KKT条件的判断过程，与C-SVC算法完全不同。其中第(10)步选择ipUp、ipLow的方法为：

$$i_{pUp} = \arg\min_t \{f(\alpha)_t | y_t = 1, \alpha_t < 1\}$$

$$i_{pLow} = \arg\max_t \{f(\alpha)_t | y_t = 1, \alpha_t > 0\}$$

算法 8 GMP-nuSVC_Train

初始化拉格朗日乘子数组alpha
初始化优化函数数组f[i]=-y[i]
初始化核函数缓存，将剩余的全部显存空间用于存放核矩阵
for i=0 **to** ℓ **do**
  计算f[i]
  更新核函数缓存
end for
loop
  在y=1的向量中计算bpUp、bpLow
  选取ipUp、ipLow
  在y=-1的向量中计算bnUp、bnLow
  选取inUp、inLow
  if bnHigh - bnLow < bpHigh-bpLow **then**
    inUp、inLow作为本轮迭代所要优化的两个元素
  else
    选取ipUp、ipLow为本轮迭代所要优化的两个元素
  end if
  if 满足KKT条件 **then**
    break
  end if
  计算eta=$\phi(x_{iUp}, x_{iUp}) + \phi(x_{iLow}, x_{iLow}) - 2 \times \phi(x_{iUp}, x_{iLow})$
  更新核函数缓存
  alpha2new[iLow]= alpha[iLow] + y[iLow] × (bUp - bLow) / eta
  alpha1new[iUp]= alpha[iUp]- y[iUp] × (bUp - bLow) / eta
  将alpha值规约到[0, 1] 区间
  更新f[i]
end loop
计算ρ与b
输出模型

第(12)步选取inUp、inLow的方法为：

$$i_{nUp} = \arg\min_t\{f(\alpha)_t | y_t = -1, \alpha_t < 1\}$$
$$i_{nLow} = \arg\max_t\{f(\alpha)_t | y_t = -1, \alpha_t > 0\}$$

与这四个元素所对应的f分别为bpUp、bpLow、bnUp、bnLow。bpUp-bpLow与bnUp-bnLow中较大者所对应的两个元素，即是本轮迭代所要选取并加以优化的目标。

算法第(21)步之后是对选取的两个元素计算核函数，以更新f和alpha数组。由于每一轮迭代仅更新两个元素的α值，所以f数组的更新不必重新计算，仅需要加上增量即可。

在计算最后，求解到最优问题所需要的alpha数组后，为了实现与GMP-CSVC算法同样的决策函数，需要对ρ和b重新计算。计算方法为：

$$r_1 = (\max_{\alpha_i=C, y_i=1} f_i + \min_{\alpha_i=0, y_i=1} f_i)/2$$
$$r_2 = (\max_{\alpha_i=C, y_i=-1} f_i + \min_{\alpha_i=0, y_i=-1} f_i)/2$$

由此可以计算ρ和b的值：

$$\rho = \frac{r_1 + r_2}{2}$$
$$-b = \frac{r_1 - r_2}{2}$$

6.2.3 实现策略

算法8在计算过程中包含大量的核函数计算步骤。经分析可得，这种运算都是针对训练集中的两个向量而进行的。因此假如能够有足够的存储空间，将核函数矩阵保存在内存中，则能减少大量的重复计算。

为了保存完整的核矩阵，所需要的存储空间为：sizeof(float) $\times \ell^2$。这意味着所需空间随着训练集的增大，呈平方规模的趋势递增。当训练集比较大时，这根本不可行。因此，本文算法只能采取保存部分核矩阵的方法。

在GPU计算的过程中，GPU如果同时肩负显示输出的功能，则需耗费一定的时间片处理输出到显示器的图形。此开销对现代GPU而言非常小，为此所耗用的显示内存一般也都在数十MB以下，所以未经使用的系统资源将处在闲置状态。而对于

专门的Tesla GPU运算设备，则已去掉显示功能，专注于计算领域。因此，GPU算法设计的宗旨，是最大限度地利用显示内存。在保证不同配置设备上算法的可伸缩性基础上，如能尽最大可能利用全部显示内存，通过存放中间结果来减低部分计算量，从而提高速度，则达到了相关目的。

本着这个原则，GMP-nuSVC算法在初始化和分配显存的最后阶段，将所有剩余的显示内存用于存放核函数矩阵。核函数矩阵最为一个对称矩阵，实际上有一半的冗余。但由于涉及到缓存调度，并且为了尽可能地实现全局存储器的协同访问，本文最终采取了按行存储的方式。核函数矩阵初始化时，调用CUDA的Driver API，获取当前的剩余存储器容量。根据容量和训练集数量 ℓ，计算能够保存的核矩阵中的行数，从而分配空间。在初始第一轮f的计算，即开始将核函数的计算结果按行保存。事实上，此阶段的迭代过程，每次都是计算核矩阵的一行数据。而在主要的运算迭代过程中，每一轮是针对两个元素的优化，也是计算核矩阵中的两行。因此，核矩阵缓存按行保存，不但有利于存放和调度，更主要的也是减少了线程间的分支，是非常高效的算法。

而对于缓存的调度，GMP-nuSVC算法采用了最近最少使用的原则。此方法是否合理还有待于进一步的研究。事实上，采用启发式的算法，在每一轮迭代之初对所需要优化的两个元素进行选取，选取时具有一定的随机性，难以事先预测，并指导缓存的调度。本文方法中，缓存的命中的概率基本上相当于缓存所占核函数矩阵的比例。

并行规约在GMP-nuSVC算法中也得到了大量应用。在采用启发式的两点选择策略中，每一轮迭代都需要四次规约，已针对ipHigh、ipLow、inHigh、inLow四个元素。在最后阶段 ρ 与 b 的计算中，也需要单独根据f数组的值，对四个集合进行规约，最终计算 r_1 与 r_2 的值。

6.3 GMP-nuSVC标号判定算法

6.3.1 算法思想

无论是传统的 GMP-CSVC 还是改进的 GMP-nuSVC 算法，最终训练结果都是找到支持向量，将其保存在分类模型中。每个支持向量都对应一个 E 值。GMP-CSVC中，$E = y_i\alpha_i^*$。对于 GMP-nuSVC，$E = (y_i\alpha_i^*)/\rho$。除此之外，分类模型还需要保存的数据包括各种训练参数、核函数参数以及判定函数中所需要的参数值。有

了分类模型，对于一个未知标号的向量 X，可以通过判定函数来预测其类标号。在GMP-CSVC 中：

$$y = \mathrm{sgn}(\sum_{i=1}^{\ell} EK(x_i, x) + b^*) \tag{6.4}$$

GMP-nuSVC中的原始形式为：

$$y = \mathrm{sgn}(\sum_{i=1}^{\ell} E(K(x_i, x) + b_1^*)) \tag{6.5}$$

这里的参数 b_1^* 在训练阶段也以转换成了 b^*。故二者在分类阶段实际执行的运算统一到了公式6.4。这部分的运算由此演变成一种高密度的核函数运算过程。本文所实现的四种核函数虽然有所差异，但对整体过程都很类似。在 LIBSVM 的实现中，流程大体上是对一个待分类的向量，计算它与分类模型中的每个支持向量之间的核函数值，然后乘以相应的 E 值后累加求和。最终加上 b^*，其符号就是预测的结果。这个过程迭代进行，直到测试集中的所有向量都计算完毕为止。

在 GPU 上实现此算法，核函数的计算同训练部分没有差异。分析此计算过程，发现支持向量的重复读取非常严重。将分类模型中的支持向量数目记为 nSV，测试集中向量的数目记为 ℓ，算法在整体执行的过程中，共要完成 $nSV \times \ell$ 个核函数的计算。而每个支持向量总共需要的 ℓ 次读取，如充分利用流多处理器中的共享存储器，则能大幅度减少。

6.3.2 算法流程

本文算法在通过判定函数对测试集中向量进行判断标号的过程中，核函数计算一步使用了共享存储器，减少了线程对全局存储器的访问次数；求和一步采用了并行规约的算法，适应了CUDA的线程模型。算法如算法9所示：

6.3.3 算法说明

GMP-nuSVC-Classify算法将数据划分成块，在载入时按块载入到流多处理器中的共享存储器中后，供线程块内的多个线程共享使用，因而减少了重复载入。数据划分策略如图6.1所示。

为了尽量达到显存的协同访问，支持向量矩阵和测试集矩阵在显存中都采用竖向存放向量的方式。支持向量的数目为 nSV，全部存放于显存中。核函数

算法 9 GMP-nuSVC-Classify

```
1: read model
2: device initialization
3: init kernel matrix KM
4: for each ℓ' sub set in ℓ do
5:    devide KM into a T × T 2-D grid of threads, each thread computes a kernel value
6:    each block init sub_result = 0
7:    for each sub-tile in d/T do
8:       loads shared_sv in SV matrix to shared memory
9:       loads shared_test in test matrix to shared memory
10:      syncthreads
11:      for k = 0 to T do
12:         compute kernel value in one dimension and add to sub_result
13:      end for
14:      syncthreads
15:      add sub_result to the corresponding position in KM
16:   end for
17:   KM[i][j] = E × KM[i][j]
18:   do sum reduction in each row of KM
19:   KM[i][0] += b
20:   sgn(KM[i][0]) is the predict result of vector test[i]
21: end for
22: output predict results
```

矩阵和应用到的测试集也需置于显存中。因而该算法对存储器的需求是巨大的。而GPU的显存容量很多时候比较有限，在一般的显示卡上一般配置数百MB，专用的Tesla GPU运算设备也仅有4GB的容量。为了使得大数据集下的运算成为可能，GMP-nuSVC-Classify算法中的第3步采用了分步骤的数据切分方法。由于对每一个测试向量的分类，均需计算与每个支持向量的核函数，所以支持向量需要全部装载，总数为 ℓ 的测试集可以进行切分。根据系统剩余的显示内存容量，以及向量维数和支持向量数量，计算系统能够存放的测试向量数目 ℓ'。之后将 ℓ' 个测试向量拷贝到显存中，并初始化 $nSV \times \ell'$ 大小的核函数矩阵。经过这种分配之后显存资源的利用率一般会达到90%以上。将存放于显存中的 ℓ' 个测试向量计算完毕后，继续装载接

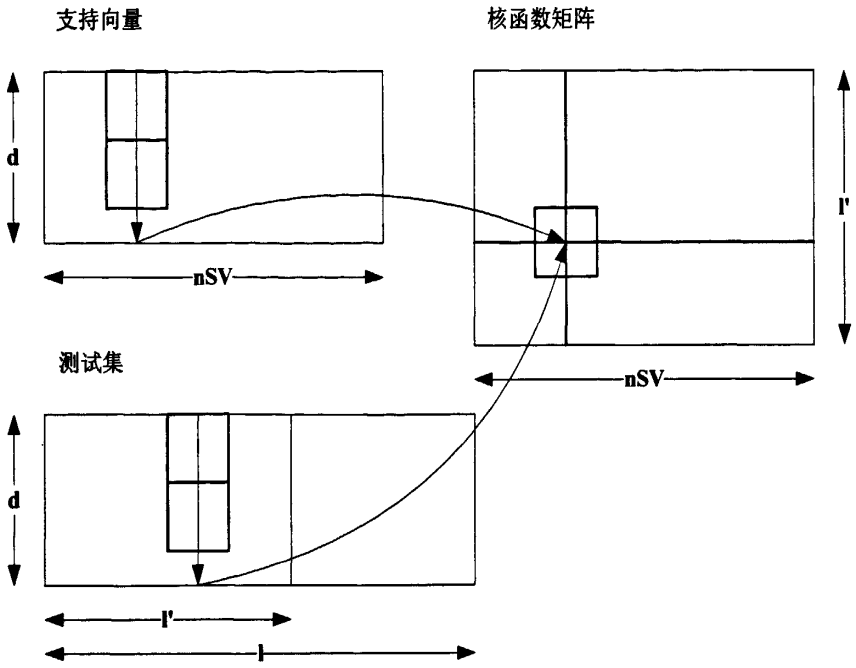

图 6.1 标号判定部分核函数的计算

下来的 ℓ' 个执行计算，直到完成为止。

接下来进行核函数的计算。核函数矩阵中的每个元素由一个线程完成，划分方式是每个宽度为 T 的方块为一个线程块，包含 $T \times T$ 个线程。线程的 x 维度方向的编号对应所计算的支持向量编号，y 维度方向的编号对应所处理的测试集中向量的编号。核函数的计算需要 d/T 次迭代完成，每次迭代载入支持向量矩阵和测试集矩阵中各一个方块的数据，供 $T \times T$ 个线程共享使用。

GMP-nuSVC-Classify算法第18步的并行规约求和，是在核函数矩阵KM的行内各自求和。求和在此处可分为两个阶段。第一阶段的数据划分方式与核函数的计算相同，当 d/T 次迭代完成后即执行。规约是在线程块内 y 方向维度相同的线程间进行的，每一行得到一个结果，保存在宽度为 nSV/T、高度为 ℓ' 的二维临时数组tempReduce中。第二次规约采用常规的方式，针对tempReduce中的每一行而进行。最终的规约结果加上 b，其符号即为预测的分类标号。

GMP-nuSVC-Classify算法符合GPU算法的设计原则，在500元等级的低端显卡上即可达到相比CPU算法30倍左右的加速比，是比较成功的。

6.4 实验结果与分析

本节测试环境和应用的数据集与第五章所述的完全相同。主机1为G92核心的9600GSO，主机2采用了Tesla C1060设备。测试所用数据集与第五章相同。

CPU对比测试也是采用LIBSVM中的算法。对三个数据集采用GMP-nuSVC训练算法和LIBSVM中的ν-SVC算法在主机1上进行测试，结果如图6.2所示。

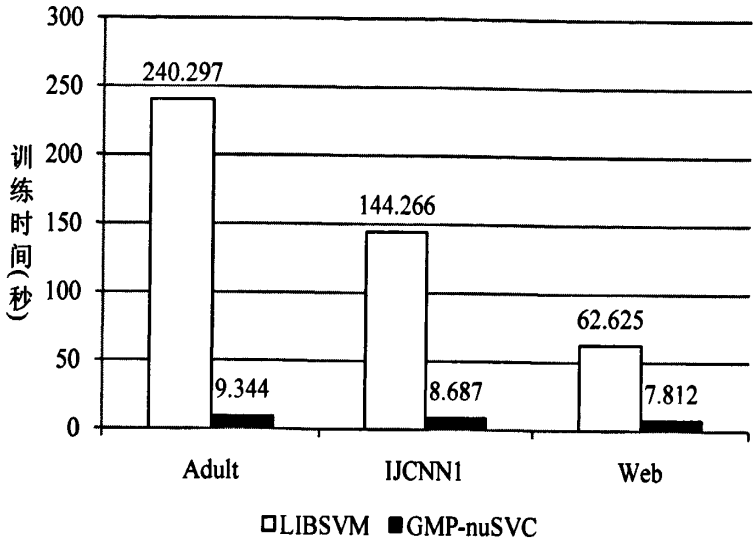

图 6.2 GMP-nuSVC算法训练时间

对Adult、IJCNN1、Web三个数据集均采用RBF核函数，参数γ为数据维度的倒数。三个数据集中的两个类别的比例都不相同，采用的参数ν分别为0.4、0.1和0.05。由图6.2可以看出本文算法显著的性能提升，取得了较大的加速比。表6.1列出了此实验结果的其它指标。迭代次数和支持向量数二者基本相同。

表 6.1 不同数据集下算法的几个指标

数据集	LIBSVM		GMP-nuSVC		加速比
	迭代次数	支持向量数	迭代次数	支持向量数	
Adult	6196	13049	6325	13052	25.71672
IJCNN1	16815	5098	25731	5129	16.60711
Web	3209	2638	2549	3041	8.01651

表6.2给出了GMP-nuSVC算法在两种主机环境下，Adult数据集不同样本数子集

下的测试结果比较。a1a至a9a是将样本数由小到大对原始测试集的拆分。采用RBF核函数，$\gamma = 0.0081$，$\nu = 0.4$。

表 6.2 算法在各种样本量和两种主机环境下的结果

数据集	训练集数目	主机1		主机2		计算步性能提升
		训练时间 T1(秒)	计算时间 T1'(秒)	训练时间 T2(秒)	计算时间 T2'(秒)	T1'/T2'
a1a	1605	0.391	0.125	1.70	0.12	1.04
a2a	2265	0.500	0.187	1.80	0.20	0.94
a3a	3185	0.500	0.219	1.88	0.24	0.91
a4a	4781	0.688	0.360	1.97	0.32	1.13
a5a	6414	0.891	0.578	2.01	0.43	1.34
a6a	11220	1.703	1.390	2.68	1.03	1.35
a7a	16100	2.812	2.516	3.77	2.06	1.22
a8a	22696	5.985	5.656	4.77	3.07	1.84
a9a	32561	9.344	9.000	8.70	6.98	1.29

在表6.2中，比较衡量计算性能的计算时间列，可以发现在原始集合或者较大样本数的情况下，拥有30个流多处理器，总共240个流处理器的主机2环境具有一定的性能优势。而在样本数目比较小时，二者差距不大。原因就在于算法8采用针对训练集中向量划分线程的方式，向量数越多，程序的并行程度也就越高。如果表中上面的 a1a 至 a4a 训练样本过小，不足以让GPU中的流多处理器满负荷运转，故在主机2中运行并没有优势。在实际应用中，万条以下的样本的训练所耗时间原本就不高，且如此小样本的训练没有太多的实际价值，所以，GMP-nuSVC算法针对大样本数据集训练的定位是准确的，效果也是显著的。

如本章第二节所述，GMP-nuSVC算法对比GMP-CSVC算法的优势在于使用小区间参数ν替代大区间参数C，同时通过参数ν控制支持向量数目，并解决C比较大时算法运算量过大的问题。下面对GMP-nuSVC算法的这些特点进行测试。

表6.3显示GMP-nuSVC训练算法和GMP-CSVC算法在adult数据集的不同子集下，采用$C = 10$或等效的参数ν训练的结果对比。

测试的方法是，对于每个数据集，首先执行GMP-CSVC算法。完毕后，计算此问题等效的GMP-nuSVC训练算法中的参数ν，$\nu = (\sum_{i=0}^{\ell} \alpha_i)/C\ell$。之后以此参数执行GMP-nuSVC训练算法。两种方式最终生成模型的效果是相似的。

表 6.3 GMP-CSVC和GMP-nuSVC算法比较(C = 10)

数据集	训练集数目	GMP-CSVC			GMP-nuSVC			
		训练时间(秒)	迭代次数	支持向量数	等价ν值	训练时间(秒)	迭代次数	支持向量数
a1a	1605	0.484	1743	645	0.3725	0.422	754	646
a2a	2265	0.594	2698	932	0.3927	0.516	1274	935
a3a	3185	0.672	3160	1228	0.3671	0.563	1522	1226
a4a	4781	0.875	4589	1792	0.3624	0.735	2151	1792
a5a	6414	1.109	5930	2361	0.3568	1.016	2927	2369
a6a	11220	1.766	9942	4070	0.3538	2.000	5223	4072
a7a	16100	2.718	13889	5799	0.3530	3.359	7111	5809
a8a	22696	4.657	21648	8142	0.3521	5.750	9892	8153
a9a	32561	8.187	31237	11466	0.3463	10.797	14812	11482

表6.3中，由于C = 10较小，二者整体运算量，也就是训练时间差别不大，而GMP-nuSVC训练算法迭代次数要略少。原因在于GMP-nuSVC训练算法在初始化阶段，计算f数组值的运算量超过前者。

对比表6.4，在C = 1000比较大的情况下，GMP-nuSVC算法相对于GMP-CSVC算法的优势则得以凸显。无论是算法的迭代次数还是训练时间，或者支持向量的数目，GMP-nuSVC均大幅度领先。

图6.3显示了LIBSVM和GMP-nuSVC-Classify算法标号判定的执行时间对比。从图中可以看到，GMP-nuSVC-Classify算法对Adult和IJCNN1数据集的分类标号预测，达到了四五十倍的加速比，而Web数据集仅有十余倍。表6.5对这三个数据集下的执行情况差异进行了对比展示。GMP-nuSVC-Classify算法最适合支持向量集和测试集都比较大的情况下的分类，Web数据集这两个数目都比较小。从计算/通讯比的角度分析，对Web数据集的分类整体执行时间仅有数百毫秒，其中接近一半是显存分配的开销。因而其加速比相对前两者略差是必然的。

总之，本章提出的GMP-nuSVC训练算法，充分利用了GPU的特性，得到了较高的加速比。而用于标号判定的GMP-nuSVC-Classify算法，采用分块的策略，利用GPU的共享存储器，达到了很高的速度。测试数据也从各个方面证明了算法的有效性。

表 6.4 GMP-CSVC和GMP-nuSVC算法比较(C = 1000)

数据集	训练集数目	GMP-CSVC			GMP-nuSVC			
		训练时间(秒)	迭代次数	支持向量数	等价 ν值	训练时间(秒)	迭代次数	支持向量数
a1a	1605	0.953	5764	848	0.1055	0.672	2661	532
a2a	2265	1.641	11620	1232	0.1493	0.953	4747	798
a3a	3185	1.906	13255	1685	0.1215	1.031	5121	948
a4a	4781	3.547	25694	2582	0.1587	1.500	8081	1774
a5a	6414	5.125	37156	3439	0.1721	1.891	9837	1946
a6a	11220	8.016	51356	5942	0.1258	2.328	10746	2617
a7a	16100	17.656	87633	8616	0.1501	5.406	23801	3976
a8a	22696	32.141	98248	11805	0.1158	10.875	45033	4482
a9a	32561	93.672	171528	16695	0.1432	26.360	88013	6229

表 6.5 三个数据集下GMP-nuSVC-Classify算法的执行统计

数据集	向量维数	测试集数	支持向量数	准确率
Adult	123	16281	13049	84.56%
IJCNN1	22	91701	5098	97.72%
Web	300	14951	2638	98.27%

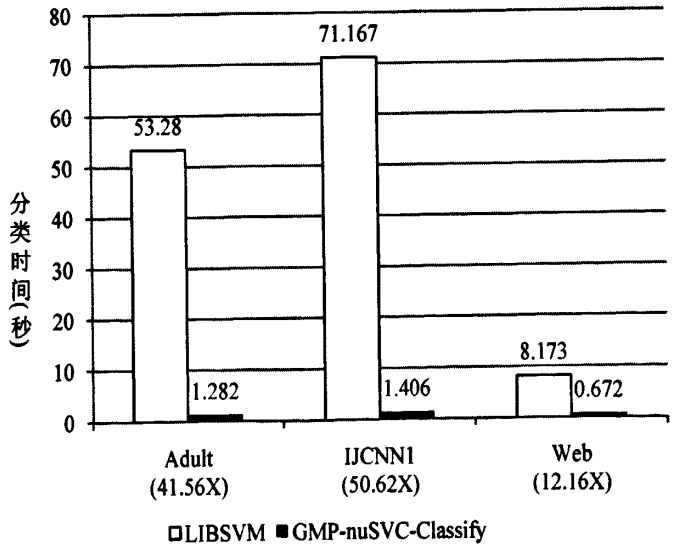

图 6.3 分类算法性能

第七章 总结与展望

7.1 全文总结

本文专注于将GPU应用于数据挖掘和机器学习领域，对各种分类方法进行了研究，提出基于GPU的算法设计原则，并对部分算法进行了实现和验证。具体工作包括以下几个方面：

（1）对GPU的软硬件体系结构进行了研究，对CUDA体系结构和编程模型进行了测试和讨论，也对数据挖掘和机器学习领域的各种分类算法进行了研究分析。很多分类算法具有优良的分类特性和广泛的应用前景，但是性能问题经常是制约实际应用的瓶颈。对此，本文提出了基于GPU的算法设计的一些原则和方法。

（2）提出了一种基于GPU的K最近邻分类算法（GSNN算法）。K最近邻分类是模式识别和数据挖掘中应用广泛的一种分类算法，简单易行且错误率相对较低。但是对大量数据的分类是一个高度密集的运算。本文算法在距离计算阶段提出了一种分块策略，在最近邻选择阶段使用了一种评估选择方法。这两个方法都充分利用了GPU的多级存储器体系结构，发挥了GPU的运算处理能力，使得计算过程相比CPU串行算法得到了很大的性能提升，对K最近邻算法在高维度、大数据量下的应用具有参考价值。

（3）支持向量机分类在解决小样本、非线性及高维模式识别问题中表现出许多特有的优势。但作为一个解决二次规划的问题，算法的复杂度比较高，在大数据集下的训练速度极不理想。为此，本文提出了一种GPU上的大规模数据并行SVM算法，即GMP-CSVC算法。该算法适合CUDA体系结构与线程模型，采用SMO序贯最优策略，在核函数的计算、拉格朗日乘子的更新、KKT条件的判断等步骤中，充分发挥了GPU的优势，并尽量减少程序分支，取得了较好的效果。测试数据也证明了算法的有效性和通用性。

（4）在实际应用中，训练SVM分类器需要经过参数寻优的过程，通过对每个参数在其约束范围内以一定间隔进行尝试，寻找到最优的参数组合，从而生成分类模型。在参数寻优中，构造SVM分类器所耗费的成本被进一步提高了。针对参数寻优过程所遇到的问题，本文提出了GMP-nuSVC改进算法。其中训练部分采用了基于的ν-SVM的改进算法，并提出核矩阵缓存调度方案，提高了参数寻优的成本并提高

了算法性能。另外在标号判定部分，通过采用分块的策略，充分利用GPU的共享存储器，达到了很高的速度。本文算法扩大了SVM算法的可计算边界，大幅度减少了训练时间，对SVM分类器的应用和GPU算法的研究发展，都有一定的参考价值。

7.2 工作展望

本文对GPU的体系结构进行了研究，对各种分类方法进行了分析，提出在GPU上设计各种分类算法的原则和方法，并对部分算法进行了实现和验证。但是，研究中还一些不够完善或不够深入的地方，还需要在以后的工作中逐步改进。主要体现在以下几个方面：

（1）将GPU用于通用计算是一个新兴的方向，尤其是将其应用于图形图像和离散模拟之外的领域。同时，这方面的研究也需要研究者具备深厚的基础，在软件和硬件方面相结合进行研究分析。本文由于作者水平有限，很多地方分析不够透彻全面，部分算法设计也不一定合理，需要在今后不断改良。另外本文只以KNN、C-SVC、ν-SVC算法为例进行了具体实现，还需在今后的工作中将其它一些算法的具体实现扩充进来。

（2）本文算法都是基于单GPU的结构的。在多GPU甚至是GPU集群下，算法的设计是有所不同的。采用MPI+CUDA是其中的一种形式。在这种环境下需要先将任务或数据划分到多个CPU进程中，进程间采用MPI接口进行通讯。之后再根据GPU的结构进行划分。本文的单GPU环境下，虽然在一定程度上提高了算法的运行速度，获得了很好的加速比，但对于更复杂或者更密集的运算，GPU集群才能更高效地予以解决。对多GPU或GPU集群下算法的设计和改进，也是需要继续研究的方面。

（3）通用GPU计算作为一个正在发展中的方向，GPU的体系结构也在不断地改进中。在本文即将完成之际，新一代的GPU刚刚发布，其结构也有所改进和变化。而本文未将这些新的发展变化包含进来，在新的GPU下算法的设计原则和方法也可能会涉及到些改良和简化，这些都是需要继续跟进研究的内容。

参考文献

[1] T. Dokken, T. Hagen and J. Hjelmervik. The GPU as a High Performance Computational Resource[C]. SCCG '05: Proceedings of the 21st Spring Conference on Computer Graphics, 2005: 21–26.

[2] 吴恩华. 图形处理器用于通用计算的技术、现状及其挑战[J]. 软件学报, 2004, 15(10): 1493–1504.

[3] B. Boser, I. Guyon and V. Vapnik. An Training Algorithm for Optimal Margin Classifiers[C]. Proceedings of the 5th ACM AnnualWorkshop on Computational Learning Theory, 1992: 144–152.

[4] C. Cortes and V. Vapnik. Support-Vector Networks[J]. Machine Learning, 1995, 20(3): 273–297.

[5] S. Arya, D. Mount, N. Netanyahu, R. Silverman and A. Wu. An Optimal Algorithm for Approximate Nearest Neighbor Searching Fixed Dimensions[J]. Journal of the ACM, 1998, 45(6): 891–923.

[6] R. Duda, P. Hart and D. Stork. Pattern Classification[M]. John Wiley and Sons, 2001.

[7] 鲁为，王枞. 决策树算法的优化与比较[J]. 计算机工程, 2007, 33(16): 189–190.

[8] D. Luebke, M. Harris, J. Krüger, T. Purcell, N. Govindaraju, I. Buck, C. Woolley and A. Lefohn. GPGPU: General Purpose Computation on Graphics Hardware[C]. SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes, 2004: 33.

[9] NVIDIA. CUDA Compute Unified Device Architecture: Programming Guide, Version 2.3[EB/OL]. http://www.nvidia.com/object/cuda_home_new.html, 2009.

[10] L. Nyland, M. Harris and J. Prins. Fast N-Body Simulation with CUDA[M]. NVIDIA GPU GEMS3, Addison Wesley Professional, 2007.

[11] T. Harada. Real-Time Rigid Body Simulation on GPUs[M]. NVIDIA.GPU GEMS3, Addison Wesley Professional, 2007.

[12] 肖江，胡柯良，邓元勇. 基于CUDA的矩阵乘法和FFT性能测试[J]. 计算机工程, 2009, 35(10): 7–10.

[13] M. Harris. Optimizing Parallel Reduction in CUDA[EB/OL]. http://www.nvidia.com/object/cuda_home.html, 2007.

[14] M. Harris. Parallel Prefix Sum (Scan) with CUDA[EB/OL]. http://www.nvidia.com/object/cuda_home.html, 2009.

[15] X. Chu, K. Zhao and M. Wang. Massively Parallel Network Coding on GPUs[C]. IEEE International Performance Computing and Communications Conference, IPCCC, 2008: 144–151.

[16] X. Chu, K. Zhao and M. Wang. Practical Random Linear Network Coding on GPUs[C]. NETWORKING 2009 - 8th International IFIP-TC 6 Networking Conference, 2009: 573–585.

[17] M. Rehman, K. Kothapalli and P. Narayanan. Fast and Scalable List Ranking on the GPU[C]. ICS'09 - Proceedings of the 23rd International Conference on Supercomputing, 2009: 235–243.

[18] N. Satish, M. Harris and M. Garland. Designing Efficient Sorting Algorithms for Manycore GPUs[C]. Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium, 2009.

[19] A. Greb and G. Zachmann. GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures[C]. Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International, 2006.

[20] 桂叶晨，冯前进，刘磊，陈武凡. 基于CUDA的双三次B样条缩放方法[J]. 计算机工程与应用, 2009, 45(1): 183–185.

[21] 李军，李艳辉，陈双平. CUDA架构下的快速图像去噪[J]. 计算机工程与应用, 2009, 45(11): 183–185.

[22] 吴连贵，易瑜，李肯立. 基于CUDA的地震数据相干体并行算法[J]. 计算机应用, 2009, 29(3): 912–914.

[23] W. Fang. Parallel Data Mining on Graphics Processors[R]. HKUST, Technical Report CS08-07, 2008.

[24] H. Bai, L. He, D. Ouyang, Z. Li and H. Li. K-Means on Commodity GPUs with CUDA[C]. CSIE '09: Proceedings of the 2009 WRI World Congress on Computer Science and Information Engineering, 2009: 651–655.

[25] M. Zechner and M. Granitzer. Accelerating K-Means on the Graphics Processor via CUDA[C]. INTENSIVE '09: Proceedings of the 2009 First International Conference on Intensive Applications and Services, 2009: 7–15.

[26] W. Fang, M. Lu, X. Xiao, B. He and Q. Luo. Frequent Itemset Mining on Graphics Processors[C]. DaMoN '09: Proceedings of the Fifth International Workshop on Data Management on New Hardware, 2009: 34–42.

[27] Utgoff and Paul E. Incremental Induction of Decision Trees[J]. Machine Learning, 1989, 4(2): 161–186.

[28] Quinlan and J. Ross. C4.5: Programs for Machine Learning[M]. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[29] L. Breiman, J. Friedman, R. Olshen and C. Stone. Classification and Regression Trees[M]. Wadsworth and Brooks, Pacific Grove, San Francisco, CA, USA, 1984.

[30] J. Platt. Fast Training of Support Vector Machines Using Sequential Minimal Optimization[M]. Advances in Kernel Methods: Support Vector Learning, MIT Press, 1999.

[31] 李建民，张钹，林福宗. 序贯最小优化的改进算法[J]. 软件学报, 2003, 14(5): 918–924.

[32] TOP500.Org. TOP500 List of Supercomputers[EB/OL]. http://www.top500.org/, 2009.

[33] J. Han and M. Kamber. 数据挖掘-概念与技术[M]. 机械工业出版社, 2007.

[34] 陈国良，吴俊敏，章锋，章隆兵. 并行计算机体系结构[M]. 高等教育出版社，2002.

[35] J. Dongarra, I. Foster and G. Fox. 并行计算综论[M]. 电子工业出版社, 2005.

[36] B. Wilkinson and M. Allen. 并行程序设计[M]. 机械工业出版社, 2005.

[37] 潘丽芳，杨炳儒. 基于簇的K最近邻(KNN)分类算法研究[J]. 计算机工程与设计，2009, 30(18): 4260–4262.

[38] D. Mount and S. Arya. ANN: A Library for Approximate Nearest Neighbor Searching[EB/OL]. http://www.cs.umd.edu/ mount/ANN/.

[39] J. Dean. Experiences with MapReduce, an Abstraction for Large-Scale Computation[C]. PACT '06: Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques, 2006: 1–1.

[40] C. Chu, S. Kim and Y. Lin. Map Reduce for Machine Learning on Multicore[C]. Proceedings of the 19th Neural Information Processing Systems, 2007: 281–288.

[41] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters[J]. Commun. ACM, 2008, 51(1): 107–113.

[42] E. Sintorn and U. Assarsson. Fast Parallel GPU-Sorting Using a Hybrid Algorithm[J]. J. Parallel Distrib. Comput., 2008, 68(10): 1381–1388.

[43] D. Cederman and P. Tsigas. GPU-Quicksort: A Practical Quicksort Slgorithm for Graphics Processors[J]. J. Exp. Algorithmics, 2009, 14: 1.4–1.24.

[44] C. Chang and C. Lin. LIBSVM: a Library for Support Vector Machines[EB/OL]. http://www.csie.ntu.edu.tw/ cjlin/libsvm, 2001.

[45] N. Cristianini，J. Taylor著，李国正等译. 支持向量机导论[M]. 电子工业出版社，2004.

[46] E. Osuna, R. Freund and F. Girosi. An Improved Training Algorithm for Support Vector Machines[C]. Neural Networks for Signal Processing VII. Proceedings of the 1997 IEEE Workshop, 1997: 276–285.

[47] C. Chang, C. Hsu and C. Lin. The Analysis of Decomposition Methods for Support Vector Machines[J]. IEEE Transactions on Neural Networks, 2000, 11(4): 1003–1008.

[48] T. Joachims. Making Large-Scale Support Vector Machine Learning Practical[M]. MIT Press, 1999.

[49] 曾志强，吴群，廖备水，朱顺痣. 改进工作集选择策略的序贯最小优化算法[J]. 计算机研究与发展, 2009, 46(11): 1925–1933.

[50] Y. Zhao and Q. He. An Incremental Learning Algorithm Based on Support Vector Domain Classifier[C]. ICCI '06: Proceedings of the 2006 5th IEEE International Conference on Cognitive Informatics, 2006: 805–809.

[51] G. Wu, E. Chang, Y. Chen and C. Hughes. Incremental Approximate Matrix Factorization for Speeding Up Support Vector Machines[C]. KDD '06: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2006: 760–766.

[52] K. Boukharouba, L. Bako and S. Lecoeuche. Incremental and Decremental Multi-category Classification by Support Vector Machines[C]. ICMLA '09: Proceedings of the 2009 International Conference on Machine Learning and Applications, 2009: 294–300.

[53] L. Zanni, T. Serafini and G. Zanghirati. Parallel Software for Training Large Scale Support Vector Machines on Multiprocessor Systems[J]. Journal of Machine Learning Research, 2006, 7: 1467–1492.

[54] T. Do and F. Poulet. Classifying One Billion Data with a New Distributed SVM Algorithm[C]. Proceedings of the 4th IEEE International Conference on Computer Science, Research, Innovation and Vision for the Future, 2006: 59–66.

[55] L. Cao, S. Keerthi, C. Ong, J. Zhang, U. Periyathamby, X. Fu and H. Lee. Parallel Sequential Minimal Optimization for the Training of Support Vector Machines[J]. IEEE Transactions on Neural Networks, 2006, 17(4): 1039–1049.

[56] L. Zanni. An Improved Gradient Projection Based Decomposition Technique for Support Vector Machines[J]. Computational Management Science, 2006, 3(2): 131–145.

[57] H. Graf, E. Cosatto, L. Bottou, I. Dourdanovic and V. Vapnik. Parallel Support Vector Machines: The Cascade Svm[J]. Advances in Neural Information Processing Systems, 2005, 17: 521–528.

[58] L. Wang and H. Jia. A Parallel Training Algorithm of Support Vector Machines Based on the MTC Architecture[C]. IEEE International Conference on Cybernetics and Intelligent Systems, CIS 2008, 2008: 274–278.

[59] 贾华丁，游志胜，王磊. 基于MTC结构的支持向量机并行训练算法[J]. 四川大学学报(工程科学版), 2007, 39(6): 123–128.

[60] B. Schoikopf and R. Williamson. New Support Vector Algorithms[J]. Neural Computation, 2000, 12(5): 1207–1245.

[61] C. Chang and C. Lin. Training ν-Support Vector Classifiers: Theory and Algorithms[J]. Neural Computation, 2001, 13(9): 2119–2147.

[62] C. Chang and C. Lin. Training ν-Support Vector Regression: Theory and Algorithms[J]. Neural Computation, 2002, 14(8): 1959–1977.

攻读硕士学位期间发表的论文

已发表论文

[1] Quansheng Kuang and Lei Zhao. A Practical GPU Based KNN Algorithm. In ISCSCT2009: Proceedings of the 2nd International Symposium on Computer Science and Computational Technology, 2009: 151-155.

致　谢

三年的研究生生活即将结束。回首往事，我觉得这三年是充实和颇有收获的三年。作为充满活力与激情的年轻人，我相信，所做出努力都会得到回报，所有的付出都能收获结果。这个阶段的学习将会让我终身受益，对我的一生产生深远的影响。三年的学习生活，紧凑充实，丰富多彩，让人难忘。而曾经帮助过我的老师，共同学习的同学和朋友们，我也将永远铭记在心，并表示由衷的感谢。

首先要感谢我的父母。父母的爱是世界上最无私的爱，父母的关爱一直温暖着远方游子的心怀。父母的鼓励与支持，是我不断前进的动力。

感谢我的导师赵雷副教授。这三年的时间里，您在学习、科研上对我的关心、指导和帮助，将会让我受益终身。您渊博的学识和严谨的治学态度深深地影响了我，将是我今后学习的楷模。而对工作认真负责的态度和一丝不苟的精神，令我非常尊敬。我这三年的学业和进步，无不倾注了您的大量心血，凝聚者您的智慧和辛劳。

感谢杨季文教授对我的指导和关心。您在繁忙的工作中一直记得关心我们的学习与生活，无论是做学术，还是做人做事，您对我们的指导和教诲，对我们思想上的启发，以及对我们的成长都有深远的影响。

我要向在做项目的过程中指导我的程宝雷老师、王岩老师和韩月娟老师，表达由衷的感谢。你们的帮助和指导，使我学到了很多东西，提高了动手能力，增长了技术水平。在项目实施的过程中，程老师所表现出来的认真负责的态度与精神，给我留下了深刻的印象，也是我今后工作中学习的榜样。

还要感谢实验室的同学们。三年的共同学习与生活，我们建立了深厚的友谊，在彼此的关心与互助中，共同进步。共同关注学术领域的进展，一起讨论技术前沿的问题，共同分享成功的喜悦。在这样的氛围中，我学到了很多东西，取得了很大的进步。

最后祝愿我的所有老师和同学们工作顺利，学习进步，事业有成。

邝泉声

二〇一〇年四月十二日