密级：_____

浙江大學

硕 士 学 位 论 文

论文题目 基于GPU的文本处理相关算法研究

作者姓名 王俊俏

指导教师 寿黎但教授

学科(专业) 计算机应用技术

所在学院 计算机学院

提交日期 2015.01

A Dissertation Submitted to Zhejiang University for the Degree of

Master of Engineering

TITLE: GPU-based Text Processing Related Algorithm

Author: Wang Junqiao

Supervisor: Shou Lidan

Subject: Computer Science and Technology

College: Computer Science and Technology

Submitted Date: 2015-04-29

摘要

基于 GPU 的并行计算是近年来非常热门的技术，在几大显卡制造厂商的不断推动下，GPU 并行计算已经在非常多的领域占据一席之地，比如英伟达公司就推出了为自家显卡编程的语言：CUDA。CUDA 是一种从 C/C++延伸出来的英伟达 GPU 编程语言，完美支持 C/C++的现有语法。

本文根据 GPU 并行加速系统的特点，尝试将两个本文处理相关的算法：单词相似度计算和近似重复文档检测算法应用进去。在单词相似度计算方面，目前较为权威并且适合大规模挖掘计算的方法就是基于单词共生关系的 DISCO 算法，经过详细的分析，本文对 DISCO 算法中从互信息矩阵计算得到相似度矩阵的过程设计了 CUDA 并行加速程序。在近似重复文档检测方面，综合考虑各种重复检测算法之后，本文对基于模糊哈希的近似重复检测进行了深入的研究，并且根据字符串编辑距离的计算方法设计了两种不同的 CUDA 加速策略。

实验方面，本文收集了维基百科英文全文作为单词相似度计算的语料库，抓取了国外的一个在线评测网站的用户提交的代码文件作为近似重复文档检测的数据源，经过大量的参数调整和对比实验，充分验证了本文提出的方法的有效性和高效性。

关键词： 单词相似度，近似文档重复检测，模糊哈希，并行计算，GPU

Abstract

In recent years, Parallel Computing based on GPU is a hot topic, and under the effort of several video card companied Parallel Computing based on GPU has already applied into many fields. For example, the NVIDIA developed a new programming language named CUDA for its video card, which is a C/C++ extended language and perfectly support current C/C++ language.

Based on the characteristics of GPU Parallel Computing, this article managed to put two text processing related algorithm, word similarity calculation and near-duplicate document detection, into test. In the aspect of word similarity calculation, this article accelerate the process of change from mutual information matrix to word similarity matrix using CUDA. In the aspect of near-duplicate document detection, this article researched the detection algorithm based on fuzzy hash and design two different CUDA kernel functions based on the edit distance calculation method.

About experiments, this article uses all page content of English Wikipedia as the corpus of word similarity calculation, and crawled twenty thousand code file from an foreign online programming website as the data source of near-duplicate document detection. Through large amount of experiments, this article demonstrate the effectiveness and efficiency of our proposed methods.

Keywords： Word similarity, near-duplicate document detection, fuzzy hash, Parallel Computing, GPU

目录

图目录

表目录

第1章 绪论

1.1 课题背景和研究现状

随着科学技术的进步和发展，互联网和人的关系变得越来越密切。特别是近几年来，一些新兴的互联网特别是移动互联网应用，很大的冲击了人们原来定的生活方式。比如打车软件“快的打车”、“滴滴打车”，让人们可以足不出户就可以预约想要的车辆。再比如一些电子商务网站“淘宝网”、“京东商城”、“聚美优品”等，让人们以几乎最低的价格买到自己想要的商品，并且可以跨城市、跨省购买甚至是海外代购。传统的 Web 页面的数量也在飞速增长，截至 2014 年 9 月 17 日，“互联网实时统计”(Internet Live Stats)显示，全球互联网网站数量已超过 10.6 亿，并且这个数字还在不断增加。

在互联网如此蓬勃发展的背后，是不断更新的技术。在网上购物人数越来越多的今天，就算是在淘宝“双十一”活动期间每秒钟几百上千万点击量的情况下，阿里巴巴依然能够保持用户流畅地浏览商品、下单、付款等操作。在 Web 网站基数如此巨大并且还在爆发式增长的情况下，全文搜索引擎“谷歌”、“百度”等网站，依然让我们可以快速地检索到我们想要的页面。支撑起这样的效果的是网站背后的优化和加速技术。在互联网如此发达的今天，并行计算、分布式计算等加速方法成了热门的话题。

互联网中的数据是丰富多样的，但其中有一种是我们每天都会见到的，发送短信、浏览微博、看新闻都会接触到它，那就是文本数据。随着文本数据的日益增长，一些朴素的文本处理问题的效率问题成了研究者们关注的焦点，比如文本分类、文本聚类、文本语义分析、自然语言处理和信息检索等问题。在这些问题中，每种课题都有相当多的算法需要研究，比如常见的聚类算法：K-means[1]算法、Single-pass[2]算法、K-medoids[3]算法、CLARANS[4](Clustering Large Application based upon RANdomized Search)算法等，常见的文本分类算法：K 最近邻算法（K-Nearest Neighbor[6]，KNN），支持向量机（Support Vector Machine[7]，SVM），

贝叶斯算法（Na ïve Bayes[9]），神经网络算法（Neural Network[10]）等。这些算法在数据量较少的时候都能够工作的很好，随着数据量的增长，其运行时间也会变得很长，没法适应实际应用的需求。

随着对性能相应速度的需求越来越凸显，并行计算这个概念应运而生。并行计算是指同时调用多种计算资源来解决计算问题的方法，是提高计算机系统的计算速度和处理能力的有效手段。从算法和程序设计人员的角度来看，并行计算又可分为数据并行和任务并行两种。Hadoop[12]是一个非常有代表性的分布式并行计算系统，它实现了一个分布式文件系统 HDFS，并可以利用 MapReduce[11]把集群里面所有的机器结合起来解决一个问题。Hadoop 特别适合解决数据密集型的大数据问题，比如它的例程单词统计问题。另一类并行计算平台就是 GPU，GPU 并行计算是利用一颗图形处理器以及一颗 CPU 来加速科学、工程以及企业级应用程序。 NVIDIA 于 2007 年在这方面迈出了第一步，开放了图形处理器的通用计算能力，并可以用 CUDA 语言进行编程，GPU 现已成就了世界各地政府实验室、大学、企业以及中小企业内的节能数据中心。

在图形处理器（Graphic Processing Unit，GPU）出现的早期，它是一种专门为图像视频解码加速使用的设备。2001 年以后，随着 NVIDIA GeForce 3 图形处理器的出现，顶点可编程（Vertex Program）技术开始普及。2004 年，美国研制的 GPU 集群系统（The Stony Brook Visual Computing Cluster）运用于城市气流模拟。2005 年，NVIDIA 和 ATI 公司相继开发出基于 GPU 的物理计算 API，使得 GPU 能够利用浮点计算单元处理包含大量并行操作的科学计算任务。

在基于 GPU 通用计算的应用系统方面，主要包括几何计算、碰撞检测、运动规划、代数运算、优化计算、偏微分方程、数值求解等。根据 NVIDIA 公司开发的 GPU 工具包 CUDA Toolkit 的测试结果显示，利用 CUDA 实现的 FFT、BLAS、排序及线性方程组求解等科学计算，与单纯依靠 CPU 实现的算法相比，平均性能提高了近 20 倍。在科研领域，非常多的传统算法都有了 GPU 的加速版本，比如：基于 GPU 的支持向量机[8]，基于 GPU 的更快的 K-nearest neighbor 算法[13]，基于 GPU 的卷积神经网络人脸检测[14]等。

1.2 本文工作及贡献

本文主要研究 GPU 并行计算在加速文本处理相关算法方面的应用。并针对两个具体的问题进行深入的研究并实现了相应的加速算法，获得了较好的加速比率。两个问题分别是：

1. 从大文档集合中挖掘英文单词之间的相似度
2. 文档集合的近似重复文档检测

在单词相似度计算方面，本文受 DISCO[5]（extracting DIStributionally related words using CO-occurrences）一文的启发，改进和完善了基于单词共生关系[15]的单词之间相似度的计算方法。单词的共生关系表示单词之间相互搭配的情况，比如 eat an apple 表示 apple 可以作为 eat 的一个宾语。可以想见如果一个单词的上下文搭配情况跟另外一个单词有非常大的相似性，那么可以间接地证明这两个单词本身具有一定的相似性，或者说是某种可替换性，这个相似性就是所谓的分布相似性。根据这个前提，我们下载了整个英文维基百科全文共 40Gbytes 的文本数据作为语料库，提取计算出现次数最多的前 10 万个单词之间的共生搭配关系，并且依据这个搭配关系计算出这些单词对两两之间的相似度。本文分析了上述方法计算单词相似度的整个过程，提出利用 GPU 加速最后一步根据共生关系计算两两单词之间相似度的可行性，将这个计算过程和现有的 GPU 上实现的加速算法进行比较，设计了最适宜单词相似度计算的加速方案。最后通过实际数据的实验，证明了确实有巨大的加速效果。

近似重复文档指的是一篇文档经过少量的修改得到另一篇文档，这个修改的比率我们用编辑距离占文档长度的比例来确定，如果修改率小于我们设定的某个阈值，那么我们说这两篇文档是近似重复的。近似重复的文档不一定是文本文档，也可以是数字的图像文件等。考虑到文档的长度可能非常长，如果采用暴力方法，直接两两计算文档间的编辑距离的方式，将是非常费时的。近似重复文档检测有自己特殊的性质，那就是两篇文档大部分是相同的，根据这个特点我们选择上下文触发的分段哈希作为检测算法的主体。大致过程是先将每篇文档分段哈希[20]

成一个长度小得多的哈希串，再两两计算哈希串之间的编辑率（第四章将详细叙述文档编辑率和哈希串编辑率之间的关系）得到可疑的近似重复文档对，最后验证可疑文档对之间的修改率来确定最终结果。同样的，本文也对上述算法进行了系统地分析，发现其耗时的关键步骤：两两计算哈希串的编辑率非常适合用 GPU 进行加速。根据编辑距离的计算特点，本文设计了两种加速方案，分析了他们的优劣势。抓取了一个在线测评网站上用户提交的代码文件作为我们的文档数据集，选择它作为测试数据集的原因是它数量足够多并且非常容易出现近似重复文档，因为用户往往需要提交多次代码才能通过某一道题的所有测试数据，而两次相邻的提交往往只修改了极少量的代码。本文的第五章实验部分对比分析了 CPU 和 GPU 加速的近似重复文档检测算法的性能差距。

本文的主要贡献包括：

1) 完善和改进了基于共生关系的单词相似度计算算法，详细分析了算法的空间时间消耗，并针对耗时的关键部分采用 GPU 并行计算的方法进行加速，对不同的共生关系矩阵形式设计了不同的加速算法。
2) 分析文档集合近似重复文档检测的特点和 CPU 上的解决方案，提出利用 GPU 加速近似重复文档检测算法的可行性，设计了多种针对编辑距离算法的 GPU 并行实现。
3) 抓取了大量的实验数据，用实际数据做实验。实现了上述算法的 CPU 版本和 GPU 加速版本，通过实验对比，加速效果明显。

1.3 本文组织

本文共分为六章，具体内容如下：

第一章为绪论，主要阐述课题背景、本文的主要工作和贡献以及本文的组织结构。

第二章介绍相关理论和技术，包括 GPU 并行计算的简介，GPU 通用计算的应用，单词相似度计算的几种常见方法，最后是近似重复文档检测的相关理论。

第三章详细介绍基于 GPU 的单词相似度计算算法，先从 CPU 上的计算方法入手，分析时间空间消耗，最后引入基于 GPU 的加速方法。

第四章详细介绍基于 GPU 的近似重复文档检测算法，同样是先从 CPU 上的计算方法入手，将问题慢慢转化成适用于 GPU 的形式，最后讨论了 GPU 上实现的细节。

第五章是实验结果和分析，从数据的获取到对比实验，再到不同算法下参数调整后对效率的影响。

第六章是总结与展望，对本文所做工作的总结和归纳，论述做好的部分和需要继续完善的地方。

1.4 本章小结

本章是本文的绪论部分，主要论述了本课题的研究背景和现状、本文的主要工作及贡献和论文的组织结构。

第2章 相关理论和技术

2.1 GPU 通用计算简介

GPU 是图形处理器（Graphics Processing Unit）的简称，又称为显示核心、视觉处理器、显示芯片，是一种专门在个人计算机、游戏机和一些移动设备（如智能手机、平板电脑等）上负责图像运算工作的微处理器。GPU 和 CPU（Central Processing Unit）的最大区别就是 GPU 拥有比 CPU 多的多的计算核心，如下图所示：

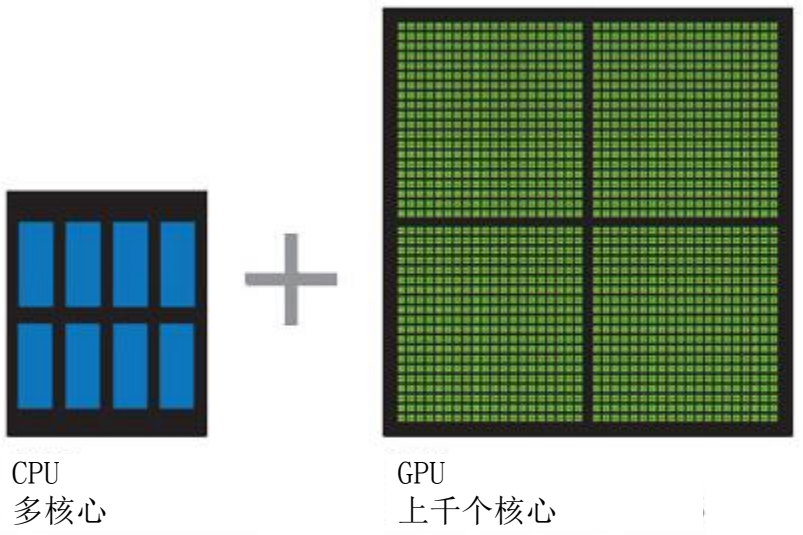


图 2.1 CPU 和 GPU 核心数区别

鉴于本文的 GPU 加速程序都是基于英伟达（NVIDIA）的 CUDA 来设计实现的，因此本文介绍的内容均基于英伟达的 GPU。英伟达的 HPC（High Performance Computing）架构已历经四代：第一代 Tesla（特斯拉）、第二代 Fermi（费米）、第三代 Kepler（开普勒）和最新的第四代 Maxwell（麦克斯韦）。经过一代一代的变革，GPU 的硬件设计也在不断改进，鉴于篇幅限制，下面从程序设计人员角度简要介绍英伟达的 GPU 架构。

首先需要介绍几个基本的概念：

SP（Streaming Processor）：最基本的处理单元，最后具体的指令和任务都是

在 SP 上处理的。

SM（Streaming Multiprocessor）：是多个 SP 加上其他的一些资源（比如共享存储器、调度器、寄存器等）组成的一个多处理器。现在的更先进的流多处理器也叫做 SMX。

Grid：每次调用 GPU 核函数启动的线程阵列。

Block：线程块，是 Grid 的次级单位，Grid 中包含若干个 Block。

Thread：线程，是 Block 的次级单位，每个 Block 包含相同数量的 Thread。

Warp：GPU 执行程序时调度的基本单位，同一个 Block 里面的 Thread 会被分到不同的 Warp 中，目前 CUDA 的 Warp 的大小为 32，也就是同一个 Warp 里面包含 32 个线程。如果一个 Block 里面包含 128 个线程，那么这些线程会被分成 4 个 Warp。

如果把 CUDA 的 Grid - Block - Thread 架构对应到实际硬件上的话，会类似对应成 GPU – SM – SP。CUDA 的 Device 在实际执行的时候，会以 Block 为单位，把一个个的 Block 分配给 SM 进行运算，而 Block 中的 Thread，又会以 Warp 为单位，把多个 Thread 结合起来做分组计算。CUDA 的 Warp 大小是 32，也就是说 32 个 Thread 会被组成一个 Warp 来一起执行。同一个 Warp 里的 Thread，会以不同的数据执行相同的指令。因此，我们每次调用 CUDA 核函数的时候都需要指定具体的 Block 数量以及每个 Block 包含的 Thread 数量。

图 2.2 英伟达 Kepler 架构流多处理器结构示意图展示的是英伟达 Kepler 架构的流多处理器结构图。每个 SMX 中包含 192 个单精度 CUDA 核（SP/Core）、64 个双精度处理单元（DP unit），32 个特殊函数单元（SFU）和 32 个存取单元（LS/ST），48Kbytes 的共享存储器以及 65536 个 32 位的寄存器。另外它还包含 4 个 Warp 调度器，也就是说同时会有 4 个 Warp 的 128 个线程在这个 SMX 里面同时运行。

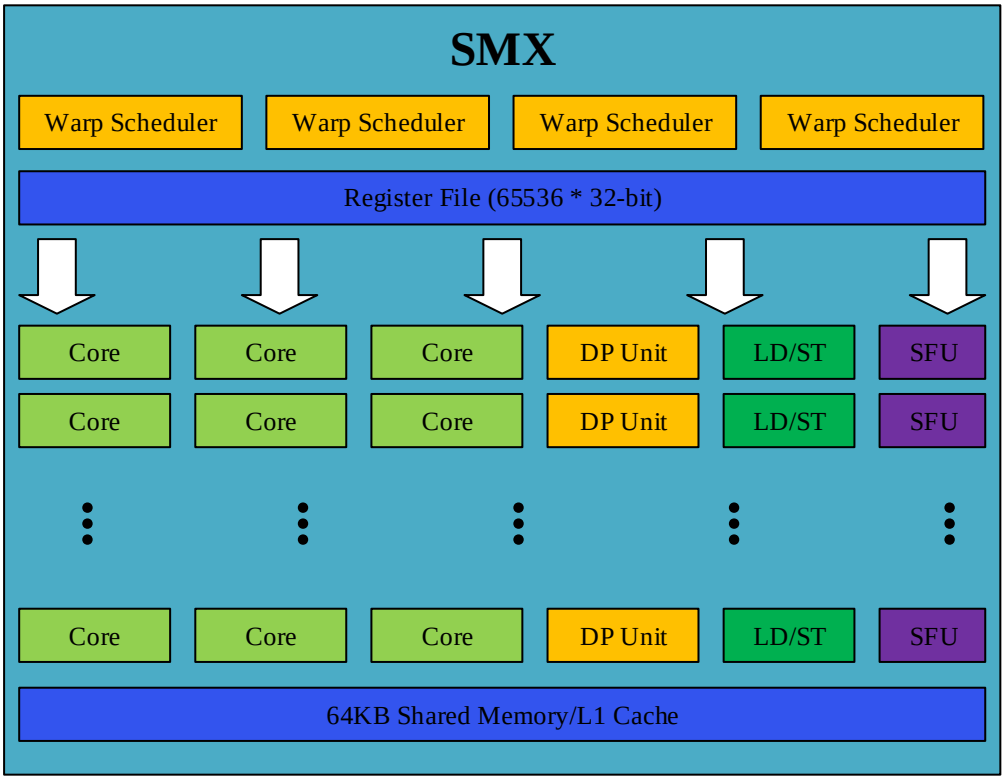


图 2.2 英伟达 Kepler 架构流多处理器结构示意图

GPU 不能主动工作，它只能靠 CPU 指令调用它，它才能启动执行。图 2.3 很好地展示了应用程序是如何利用 GPU 实现加速的。CPU 专门执行顺序串行的代码，而 GPU 则专门针对计算密集型的代码段。这种 CPU-GPU 形式的应用程序就是目前 GPU 通用计算的主流形式。

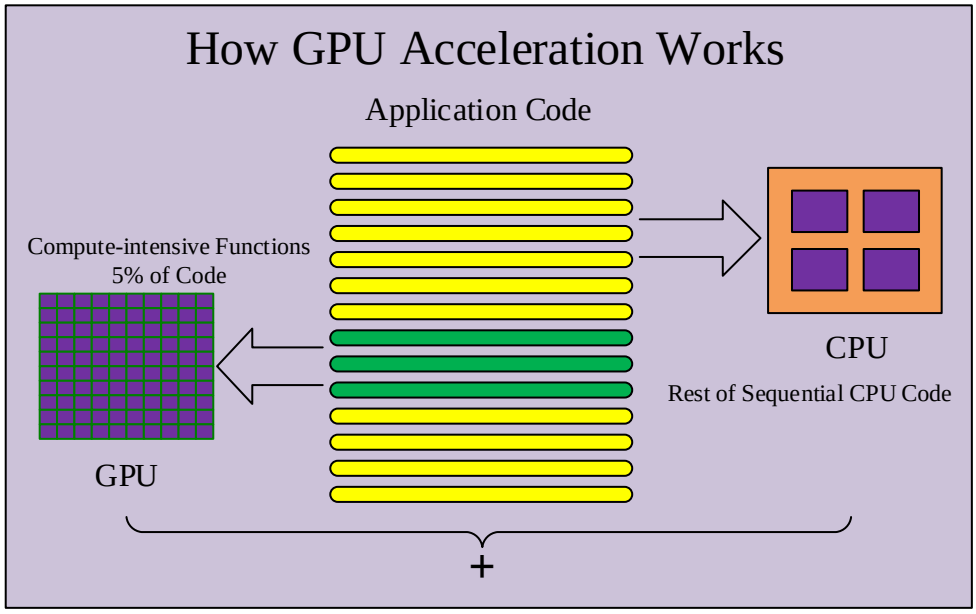


图 2.3 GPU 加速系统示意图

2.2 单词相似性

首先，本文所阐述的单词相似性指的是语义上相似性，而并不是所谓的单词长的相似。比如“peal”是“隆隆声”的意思，而“pear”是“梨”的意思，虽然他们看起来很相似，其实并没有太大的关联。文本讨论的是指像“road”跟“path”这样的意义上比较相似的问题，他们都可以表示“路，道路”的意思。

计算单词的相似度是自然语言处理里面一个比较复杂的问题。一般有一下几种分析方法：潜语义分析法，WordNet 方法和 DISCO 单词相似度计算法。下面就这几种方法做简单介绍。

2.2.1 潜语义分析法

潜语义分析法（Latent Semantic Analysis[16]，LSA）是自然语言处理方面的常用算法，该算法同样也适用于用于单词相似性的计算。这个算法关键的步骤就是奇异值分解（Singular Value Decomposition[17]，SVD），可以说奇异值分解是潜语义分析的数学基础和核心，它能把一个高维空间的矩阵经过复杂的计算转换成一个低维空间的矩阵，并使得均方误差最小。

潜语义分析法计算单词相似度需要一个较大的文档集合，每篇文档中包含一定数量的单词，它的最终目的是要得到单词在给定的文档集合中的真正含义，也就是所谓的“潜在语义”。通俗点来说，就是经过奇异值分解把高维文档集都用映射到一个低维空间。举个例子，比如对于一个包含 2000 篇文档的文档集，出现过 10000 个索引单词，潜语义分析可以做到仅使用 200 维度就将单词和文档都映射到该空间。由此可以看出，降维是潜语义分析中最核心的一步。通过降维，原始文档中的“噪音”（无关信息）都被去除了，剩下的就是比较精炼的语义了。

使用潜语义分析方法计算单词相似度的时候，首先把两个要计算的单词都映射到该低维度空间，也就是把两个单词都表示为该空间中的向量，有了这样的转化关系，那么单词的相似度也就可以用向量的相似度来求了，向量的相似度计算最常用的方法有余弦距离、皮尔森距离等。最终，我们把单词的相似度计算转化成了两个向量的距离。

2.2.2 WordNet 方法

WordNet[19]是一个美国普林斯顿大学提供的一个软件包，它可以用来计算两个概念单词之间的关联性（Relatedness）和语义相似度[18]（Semantic Similarity）。它提供了多种不同的关联性度量方法和相似性度量方法，这些度量方法都是基于 WordNet 人工维护的词汇数据库的，人工维护意味着 WordNet 具有较高的准确性。官方网站提供了这些度量方法对应的 Perl 脚本语言的编程接口。编程接口的输入是两个单词，输出是代表两个单词关联性或者相似度的数值。

WordNet 维护的单词数据库都是一些具有 is-a 层次结构的单词对，单词间的关联性和相似度计算都是基于这个层次结构的。在 WordNet 2.0 中名词具有 9 个层次结构一共包括了 8 万多个单词，动词层次结构较多，有 554 个，一共包括 13000 多个单词。比如 apple 和 pear 间的相似度要高于 apple 和 sky 之间的相似度，这是由于 apple 和 pear 在 is-a 的层次结构中有一个公共祖先 fruit，而 apple 和 sky 之间并没有如此直接的关系。

WordNet 的优点是人工维护准确性高，但是局限性也很明显，那就是 is-a 结

构是一种简单而直接的描述，并不能跨越不同的词性，因此 WordNet 目前只能计算两个名词之间或者两个动词之间的关联性和相似度。对于词性不同的两个单词，WordNet 并没有很好的方法计算它们之间的相似度。

2.2.3 DISCO 方法

DISCO（extracting DIStributionally related words using CO-occurrences）是一种根据单词之间的共生关系来计算单词相似度的方法，它可以用来检索两个单词之间的分布相似度（Distributional Similarity）。Lin 在“自动检索聚类相似单词[21]”一文中提出单词相似度应该考虑单词之间的共生特性的理论，共生特性可以从语句的语法结构中提取，大量重复的单词对的搭配组合说明了特定单词对之间具有某种依赖关系，从而间接地说明了单词之间的相似性，这个方法的难点在于语法分析的模块，Lin 后来又发表了一篇论文[22]来解决这个问题，2004 年 Weeds, J.发布了一个专门用来提取共生单词的框架[23]。DISCO 的作者 Peter Kolb 在前人工作的基础上提出的更加量化并且可以大规模计算的方法，即用单词在滑动窗口中的相对距离来近似代替单词在句子中的语法结构，从而简化了对语句进行语法分析的一部，可以说 DISCO 是一种更加实用的单词相似度计算方法，并且可以计算任意单词对之间的相似度。但是 DISCO 中计算单词间相似度的效率非常低下，本文中将对这一计算过程加入 GPU 并行计算优化。

2.3 近似重复文档检测

近似重复文档检测是文档重复检测里面的一个子类问题。一般来说，文档重复检测分为全文重复（一模一样）、部分重复（语句级别）和近似重复[24]（几乎全文重复）几种。近似重复文档检测是搜索引擎和 Web 爬虫里面的一个重要算法，因为搜索引擎和爬虫算法需要对近似重复的网页进行归类，这样就可以避免用户搜索到很多几乎一样的页面也可以降低搜索引擎的负载，进而改善用户体验。

重复文档的检测已经有很多的算法，下面列举一些。

2.3.1 Shingle 算法

Shingle[25]指的是文档中连续的子序列。Shingle 算法的具体做法是根据给定的常量 w，把文档 Doc 中所有长度为 w 的 Shingle 拿出来，然后去掉重复的子序列，这些子序列形成的集合就是 Doc 的特征集合 S(Doc)。两个文档 A 和 B 的相似度 sim 定义为:

$$sim(A,B)=|S(A)\cap S(B)| / |S(A)\cup S(B)|$$

其中|A|代表集合 A 的大小，因此相似度是一个介于 0 和 1 之间的值。

举个简单的例子，假设文档 Doc 的内容是"a rose is a rose is a rose"。分词后的词汇序列是(a,rose,is,a,rose,is, a, rose)。现在假设 w=4，那么 Doc 的特征集合就是：

{ (a,rose,is,a), (rose,is,a,rose), (is,a,rose,is), (a,rose,is,a), (rose,is,a,rose) }

去掉重复的子集合之后变成：

S(Doc) = { (a,rose,is,a), (rose,is,a,rose), (is,a,rose,is) }

2.3.2 Simhash 算法

Gurmeet 等人在论文“Detecting Near-Duplicates for Web Crawling”中提出了 Simhash[26]算法，专门用于解决数亿级别的网页文档去重问题。Simhash 算法是局部敏感哈希[30]（Locality Sensitive Hash）的一种，它的主要思想是降维，将高维的特征向量映射成低维的特征向量，然后通过计算两个向量的海明距离（Hamming Distance）来确定两篇文档是否重复或者高度近似。Simhash 算法分为 5 步：

1. 哈希。将所有的可能出现的单词赋予一个 n-bit 的哈希值，长度为 n，每一位值是 0 或者 1。这是一个全局操作，可以在处理文档之前做一次预处理即可。
2. 分词。针对每篇文章，首先分词获得所有在文档中出现过的单词，然后赋予每个单词一个范围 1~5 的权值，代表这个单词在文章中的重要程度。
3. 加权。把每个单词的权值加到哈希值上。比如单词 sun 在某文档中的权值是 3，而 sun 的哈希值是 1010，那么 sun 加权后得到序列[3,-3,3,-3]。
4. 合并。把文档内的所有单词的加权合并。假如某文档共包含 2 个单词，

它们的加权序列分别是[3,-3,3,-3]和[2,2,2,-2]，那么合并后得到序列[5,-1,5,-5]。

5. 降维。对于合并后的序列，我们设置一个阈值 P，大于 P 的都变成 1，否则变成 0，这样就又把合并后的序列变成一个 n-bit 的哈希值，这个值就是这篇文档的 simhash 值，完成降维操作。

获得文档的 simhash 值之后，计算文档之间的重复程度只要计算 simhash 串之间的海明距离即可。

2.3.3 I-Match 算法

I-Match[31]算法是搜索引擎判断文章原创的一个重要手段，其实也是一种判断文档重复的方法。I-Match 算法有一个重要的假设：不经常出现的词和经常出现的词不会影响到文档的语义，因此这些词是可以去掉的。可以说 I-Match 算法是基于语义的排重算法。

I-Match 算法的基本思想是这样，将文档中有语义的单词用传统的哈希算法表示成一个数字，数字相同的就表示文档之间的相似性。

算法可以分为以下几个步骤：

1. 将文档进行分词，分解成 token 流。
2. 通过判断单词的阈值（通常是 IDF 值），去掉常见的和不常见的，保留有意义的 tokens。
3. 将所有有意义的 token 排序。
4. 在排序后的 token 序列上应用哈希算法，的到一个哈希值 hash_value。
5. 将元组（doc_id, hash_value) 插入到某一词典中，如果发生冲突，就说明两个文档相似。

2.3.4 模糊哈希算法

模糊哈希（Fuzzy Hashing）算法又叫基于内容触发的分片哈希算法[32]（Context Triggered Piecewise Hashing, CTPH），主要用于文档的相似性比较，尤其对于近似重复的文档具有得天独厚的优势。Carlos[33]等人在 2011 年将模糊哈希算法应用与网页的重复检测。模糊哈希算法是基于哈希算法的，但它又不同于传统的哈希算

法，传统哈希算法是将整个文档作为一个整体来计算一个结果，这样一来即使原始文档有一点点的改动也会使得最终的结果变得完全不同，因此不适合近似重复的文档检测，而模糊哈希算法则需要先对文档进行分块，而且分块的策略并不是固定的，而是根据上下文来触发切分点，然后计算每一块的哈希值，最后将得到的一串哈希值作为文档的签名。利用比较函数与其他文档的签名进行比较，来确定文档之间的相似程度。模糊哈希算法一般由以下几部分组成：

1. 一个滚动哈希算法和一个分片值，用于触发分片。
2. 一个强哈希算法，用于计算每片的哈希值，比如 MD5 等。
3. 一个字符映射算法，将每片的哈希值映射为一个字符，并且把整个文档的哈希值序列映射成一个字符序列。
4. 一个相似度比较算法，用于对两个模糊哈希值计算的相似程度。

模糊哈希算法是本文研究的一个重点，关于它的具体讨论会在第四章展开。

2.4 本章小结

本章主要介绍了一些跟本文相关的一些技术和算法，包括 GPU 通用计算方面的介绍、单词相似度计算相关的方法和近似重复文档检测的相关算法。

第3章 基于 GPU 的单词相似度计算

3.1 问题定义

在某些应用场景下，我们可能需要知道不同单词之间的相似程度。在此文[27]介绍的音乐推荐系统 Pictune[28]的实现过程中就牵涉到了相似单词之间转化的问题，它需要把若干个用户评论的单词转换成歌词中常用的单词。虽然都是英文单词，但是普通人写的词汇跟歌曲创作者使用的词汇，还是有较大差别的，因此需要建立这两个“词汇域”之间的映射关系，这个就是单词的相似度。

下面给出本文所述的单词相似度计算的具体问题：从一个大的文档集合挖掘出一个单词相似度矩阵，如果我们考虑常见的 N 个单词，那么这个矩阵应该是一个 N*N 大小的浮点数矩阵，第 i 行第 j 列的值表示第 i 个单词和第 j 个单词之间的相似度。

在《基于移动用户地理信息的音乐推荐研究》这篇文章中，作者简要论述了其计算单词之间相似度的方法，笔者认为他的方法还有很大的改进空间。其一是他在计算二阶相似度矩阵的时候只考虑了一阶矩阵中的列项不为 0 的单词，更精确的方法应该是考虑所有单词对之间的相似度。其二就是计算的效率，63 小时的计算还是可以继续优化的，因此笔者期望通过 GPU 的加速效果进一步优化计算的时间。

3.2 原始计算方法

Lin 在 1998 年提出一个理论：单词的相似度计算需要考虑词汇之间的搭配情况，也就是一个单词的意义在很大程度上取决于这个单词的跟其他单词的搭配模式。词汇搭配是英文语言中非常常见的形式，比如“eat an apple”（吃苹果），“pull the trigger”（扣动扳机），“a drunk man”（醉酒男子）等。而单词之间的搭配可以从大量的已有的文本文档中获取，这给单词相似度计算的大规模大批量进行提供了可能。在深入一层，如果两个单词 A 和 B 出现大量的一起搭配使用的情况，并

且 A 和 C 也出现了大量的一起搭配的情况，那么我们推测 B 和 C 具有一定的相似性，或者说 B 和 C 具有某种可替换性，比如“a cup of water”和“a glass of water”，cup 和 glass 在这里存在一定的可替换性，倘若在很多情况下都具有这样的可替换性，那么我们可以猜测 glass 和 cup 具有很高的单词相似性。

词汇的搭配可以简化为单词依赖三元组<word1, relation，word2>的模型。比如句子“I have a black cat”中可以提取的词汇搭配有：

<I, N:subj:V, have>

<have, V:compl:N, cat>

<black, A:jnab:N, cat>

<a, D:det:N, cat>

各种不同的依赖关系及其他们的含义如表 3.1 所示：

表 3.1 三元组依赖关系及其含义

依赖关系	含义
N:subj:V	主语和它的动词
V:compl:N	动词和它的宾语
N:nn:N	名次修饰词和被修饰的名词
A:jnab:N	定语和它修饰的名词
D:det:N	限定词和名词
A:jvab:V	状语和它的动词

从一个非常大的语料库中把这样的搭配三元组提取出来，然后我们用互信息[29]（Mutual Information）来衡量单词之间的搭配关系。

一个单词搭配三元组<W_1,rel,W_2>被定义为以下三个事件的共生事件：

A：随机地选取一个单词 W_1；

B：随机地选取一种依赖关系 rel；

C：随机地选取一个单词 W_2。

在 Lin 的文章[21]中互信息被定义为：

$$\log \frac{P(A,B,C)}{P(A)*P(B)*P(C)} \qquad \text{公式（3.1）}$$

这个定义有一个假设，那就是 A，B，C 三个事件是相互独立的。但事实上，他们并不是独立的，A 和 C 是会受到 B 的限制的，根据表 3.1 所示，比如当依赖关系是“D:det:N”时，那么 W_1 的词性必须是限定词，而 W_2 的词性必须是名词。

因此我们必须再做一次假设，在 B 给定的情况下，A 和 C 是条件独立的。文章中采取的假设可以用贝叶斯网络（Bayesian Network）图模型的方式表示如下：

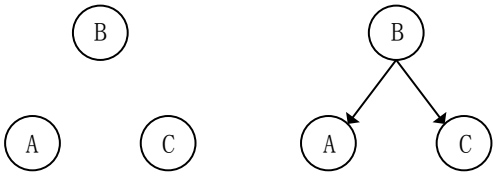


图 3.1 依赖三元组的图模型

在这样的前提下，三元组<W_1,rel,W_2>的互信息可以表示为：

$$\log \frac{P(A,B,C)}{P(A|B)\times P(B)\times P(C|B)} \qquad \text{公式（3.2）}$$

上述公式中的概率函数 P 可以用元组在整个语料库中出现的次数来近似估计。因此我们可以得到如下的概率公式：

$$P(A,B,C)=\frac{\|W_1,rel,W_2\|}{\|*,*,*\|} \qquad \text{公式（3.3）}$$

$$P(B)=\frac{\|*,rel,*\|}{\|*,*,*\|} \qquad \text{公式（3.4）}$$

$$P(A|B)=\frac{\|W_1,rel,*\|}{\|*,rel,*\|} \qquad \text{公式（3.5）}$$

$$P(C|B)=\frac{\|*,rel,W_2\|}{\|*,rel,*\|} \qquad \text{公式（3.6）}$$

其中$\|W_1,rel,W_2\|$表示三元组<W_1,rel,W_2>在解析整个语料库后出现的总次数。其中*是通配符，表示在该维度下将所有可能的单词和关系进行次数求和。

比如$\|*,rel,*\|$代表所有关系三元组中单词之间的关系类型是 rel 的三元组出现的总次数。将公式（3.3）、公式（3.4）、公式（3.5）和公式（3.6）带入到公式（3.2）得到三元组<W_1,rel,W_2>的互信息计算公式：

$$I(W_1,rel,W_2)=\log\frac{\|W_1,rel,W_2\|\times\|*,rel,*\|}{\|W_1,rel,*\|\times\|*,rel,W_2\|} \qquad \text{公式（3.7）}$$

至此，我们可以把所有三元组之间的互信息值计算出来，这样我们就得到一个 v*v*r 大小的矩阵，其中 v 表示词汇总数，r 表示关系总数。

接下来，我们定义 T(W_1)表示所有使得 I(W_1,rel, W_2)>0。然后单词的关系单词对<r, W_2>的集合。那么的单词对<W_1，W_2>的相似度可以定义为：

$$sim(W_1,W_2)=\frac{\sum_{(r,w)\in T(W_1)\cap T(W_2)}(I(W_1,r,w)+I(W_2,r,w))}{\sum_{(r,w)\in T(W_1)}I(W_1,r,w)+\sum_{(r,w)\in T(W_2)}I(W_2,r,w)} \qquad \text{公式（3.8）}$$

至此，我们已经基本得到了从一个很大的语料库获得两个单词相似度的计算方法。总体上可以粗略地分为以下三步：

1. 运用语法分析、词法分析从大语料库 C 中解析所有的单词搭配三元组<W_1,rel,W_2>，统计相同的三元组<W_1,rel,W_2>的出现次数。
2. 根据公式（3.7）计算所有单词对之间的搭配信息（互信息）。
3. 根据公式（3.8）和第二步已经计算好的互信息来计算所有单词对之间的相似度。

3.3 DISCO 计算方法

从上面的分析中，我们看到，原始的单词相似度计算方法非常依赖一开始的搭配三元组的解析提取。句子的语法分析作为一个自然语言处理里面非常麻烦的问题，它在原始算法里的表现将直接影响到最后单词相似度计算的精确度。而现实中的文本文档大多由人们撰写，这就引发了一个很重要的问题，各个地方都有不同的语言风格，句子的结构可谓是千变万化，人类可以根据上下文的逻辑、内容等理解文章的内容，但是计算机却很难做到这一点。因此，语料库的第一步预

处理就成了原始单词相似度算法的大难题。

DISCO 算法的作者提出：既然句子的语法结构如此地难以处理，那么我们是不是可以用另一种方式来定义两个单词的依赖关系呢，这种方式就是“相对位置”。具体做法就是设置一个滑动窗口（假设大小为 4），那么句子里的每个单词会跟它出现位置的前 3 个和后 3 个单词产生一个搭配关系三元组。举个例子，句子“I have a black cat”可以被分解为以下三元组（a 是一个停止词，所以忽略它）：

<I, 1, have>, <have, 2, black>, <have, 3, cat>

<have, -1, I>, <black, -2, have>, <cat, -3, have>

即< W_1, position, W_2>的形式，W_1 和 W_2 表示两个单词，position 表示两个单词的相对位置。这样的处理方式是语法结构分析的一种简化，是一种简易的替代品，是对计算精度和算法复杂性之间的一种妥协。但实验结果表明，它依然可以得到非常满意的单词相似度的计算结果。

1. 对语料库进行分词，去掉停止词，并用一个滑动窗口扫描所有的句子，把所有的< W_1, position, W_2>统计起来。
2. 根据公式（3.7）计算所有单词对之间的搭配信息（互信息）。
3. 根据公式（3.8）和第二步已经计算好的互信息来计算所有单词对之间的相似度。

可以发现，上述处理过程已经没有词法分析、语法分析等复杂的自然语言处理过程，只剩下分词、去除停止词等简单的处理。也正是基于此，DISCO 算法计算单词相似度是一种语言依赖性非常低的算法，只要有一个足够大的语料库，DISCO 可以计算任何一种语言里面两个单词的相似度，这也是 DISCO 单词相似度计算一个非常大的优点。

3.3.1 DISCO 算法复杂度分析

下面我们把 DISCO 的单词相似度计算量化到具体的细节，并考虑计算过程中的性能问题。为了便于表述，以下的分析过程先忽略三元组<W_1, position, W_2>中的 position，因为 position 在最终的相似度计算公式（3.8）后就不存在了。那么

<W_1, W_2>其实就表示 W_1 和 W_2 在同一个滑动窗口中出现过一次。

现在以英文的维基百科（Wikipedia）所有词条作为语料库为例，来讨论 DISCO 单词相似度计算的具体细节。假设我们考虑的单词集合为 V，||V||表示词汇集合的大小（根据实验结果，大致为 20 万左右）。

处理的第一步，经过分词、去除停止词，再用滑动窗口扫描，我们可以得到一个$\|V\|^2$ 大小的整数矩阵，整数值表示两个单词共生出现的次数，图 3.2 是一个例子：

	w_1	w_2	w_3	...	$w_{\|V\|}$
w_1	0	5	0	...	2
w_2	5	0	3	...	0
w_3	0	3	0	...	0
⋮	⋮	⋮	⋮	⋱	⋮
$w_{\|V\|}$	2	0	0	...	0

图 3.2 单词共生次数矩阵

$w_1, w_2, w_3, ..., w_{\|V\|}$ 表示所有的单词，对角线上的数值没有意义，全部标记为 0。值得注意的是，这个共生次数矩阵中有大量的 0，因为可以预料的是，并不是所有的单词对之间都会出现共生搭配关系，有些单词对之间没有搭配的可能性，所以就算是在整个维基百科里面，都没有出现搭配也是可以理解的。总结第一步的处理过程，它的计算复杂度为$\Theta(\|V\|^2 + C)$，其中 C 是文档集合的大小。

下一步就是根据公式（3.7）来计算单词对之间的互信息，得到一个互信息矩阵。考虑公式（3.7）中的几个项目，$\|W_1, rel, W_2\|$这一项为 0 的（也就是图 3.2 中矩阵中数值为 0 的）单词对(W_1, W_2)都不用计算互信息，因为这个值始终为 0。从互信息的含义中我们也可以理解这一点，两个单词在整个文档集合中都没有搭配出现过，互信息就等于没有，也就是 0 了。$\|*, rel, *\|$、$\|W_1, rel, *\|$和$\|*, rel, W_2\|$这三项都可以一次性预处理统计出来，因此这一步的计算复杂度为$\Theta(\|V\|^2)$。

经过计算得到的互信息矩阵类似图 3.3 所示（图中的数值仅作示意，并不是根据图 3.2 计算得到）：

	w_1	w_2	w_3	...	$w_{\|\|V\|\|}$
w_1	0	0.3	0	...	0.79
w_2	0.3	0	1.2	...	0
w_3	0	1.2	0	...	0
⋮	⋮	⋮	⋮	⋱	⋮
$w_{\|\|V\|\|}$	0.79	0	0	...	0

图 3.3 单词互信息矩阵

最后一步是根据互信息矩阵计算得到所有单词对之间的相似性矩阵。研究公式（3.8）可以发现，分母里面的两项 $\sum_{(r,w)\in T(W_1)} I(W_1,r,w)$ 和 $\sum_{(r,w)\in T(W_2)} I(W_2,r,w)$ 可以通过预处理得到，但是分子项 $\sum_{(r,w)\in T(W_1)\cap T(W_2)} (I(W_1,r,w)+I(W_2,r,w))$ 随着两个单词 W_1 和 W_2 的不同计算结果是不同的，因此这一项每次都需要 $\Theta(\|V\|)$ 的是时间来做，这导致整个第三步的计算复杂度达到了 $\Theta(\|V\|^3)$。计算得到的单词相似性矩阵如图 3.4 所示：

	w_1	w_2	w_3	...	$w_{\|\|V\|\|}$
w_1	0	0.12	0.07	...	0.15
w_2	0.12	0	0.04	...	0.02
w_3	0.07	0.04	0	...	0.09
⋮	⋮	⋮	⋮	⋱	⋮
$w_{\|\|V\|\|}$	0.15	0.02	0.09	...	0

图 3.4 单词相似度矩阵

因此整个 DISCO 算法计算单词相似性矩阵的时间复杂度是 $\Theta(\|V\|^3+2\|V\|+C)$，以维基百科全文大约 40G 的文档集合作为语料库和词汇量 20 万来算，一台单 CPU（假设这个 CPU 每秒能处理 10^9 次的基本操作）的计算机大概需要处理 2000 个小时才能完成。其中大部分的时间花在了第三步从互信息矩阵计算得到相似度矩阵上。

3.3.2 DISCO 的初步优化方法

前文已经提到了互信息矩阵里面有大量的 0 数值，这代表着非常多的单词对并没有在语料库中出现过搭配在一起的情况。因此，互信息矩阵可以被认为是一个稀疏矩阵。对于稀疏矩阵，我们可以采取压缩存储的方式来减少存储的空间，并且这也会降低第三步相似性矩阵计算的复杂度。经过笔者的测试，对于整个维基百科的物料库来说，每个单词的共生单词数量大概在 6000 个左右。这样一来，整个的计算时间大大降低，只需要大概 70 个小时的时间。具体的压缩存储方式如图 3.5 所示：

$$\begin{pmatrix} & x_1 & x_2 & x_3 & x_4 \\ x_1 & 0 & 4 & 0 & 0 \\ x_2 & 4 & 0 & 6 & 0 \\ x_3 & 0 & 6 & 0 & 0 \\ x_4 & 0 & 0 & 0 & 0 \end{pmatrix} \rightarrow \begin{bmatrix} x_1 & 1 & (2,4) & \\ x_2 & 2 & (1,4) & (3,6) \\ x_3 & 1 & (2,6) & \\ x_4 & 0 & & \end{bmatrix}$$

图 3.5 稀疏矩阵及其压缩存储

另一种加速方法简单而暴力，根据上面的论述，既然互信息矩阵里面有大量的 0 元素，那么我们也可以做出一个合理的推断：仅有一部分的单词会经常和别的单词出现搭配情况。这也很好理解，生僻单词往往会跟一些常用单词一起出现，两个生僻单词一起出现的概率是比较低的。因此可以把共生单词的数量作一个限定，并不是考虑整个的词汇集合 V。比如我们只取其中的 10000 个单词作为共生单词的数目，以这些单词作为媒介来计算两两单词之间的分布相似度。这样的方法同样可以大大降低整个计算过程的时间复杂度，大概需要 100 个小时左右。只选取常用词作为共生词的方法如图 3.6 所示：

$$\begin{pmatrix} & x_1 & x_2 & x_3 & x_4 \\ x_1 & 0 & 1 & 0 & 0 \\ x_2 & 1 & 0 & 6 & 1 \\ x_3 & 0 & 6 & 0 & 0 \\ x_4 & 0 & 1 & 0 & 0 \end{pmatrix} \rightarrow \begin{pmatrix} & x_2 & x_3 \\ x_1 & 1 & 0 \\ x_2 & 0 & 6 \\ x_3 & 6 & 0 \\ x_4 & 1 & 0 \end{pmatrix}$$

图 3.6 原矩阵的精简化

可以发现，这种方式会导致原矩阵中部分信息的丢失，因为虽然生僻单词不常出现，但对于计算结果还是有一些影响的。

经过上面的论述，我们现在已经可以画出整个 DISCO 单词相似度计算的全部过程，流程如图 3.7 所示：

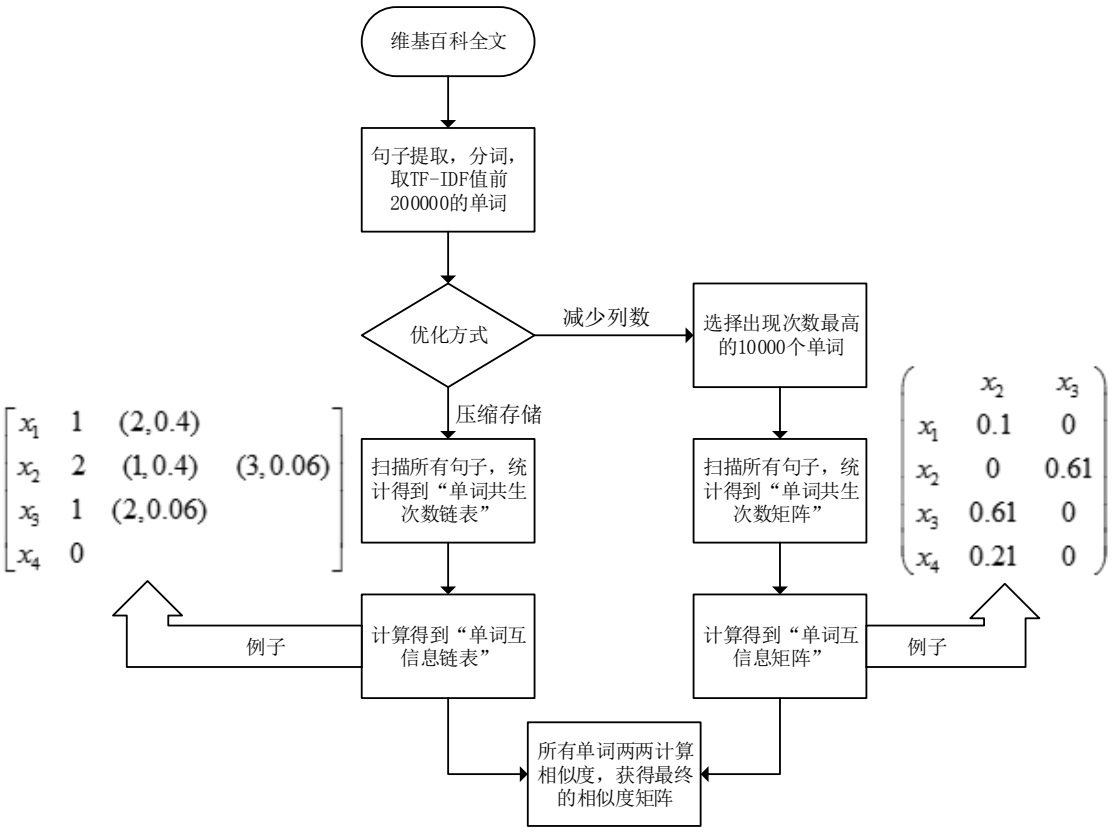


图 3.7 DISCO 单词相似度计算流程

上面已经说明了互信息矩阵中存在大量的 0，直接全部存储会非常的浪费空间，并且也会极大地降低最后一步计算的效率。下面介绍压缩存储的具体存储形式：

假设我们需要考虑的单词是$(w_1, w_2, w_3, ..., w_N)$共 N 个，下面列出的几个量是我们压缩存储互信息矩阵时用到的：

1. 数组 num[N]，num[i]表示第 i 个单词跟多少个单词存在共生关系。
2. N 个整型向量 index[N],index[i]包含 num[i]个元素，元素值表示跟第 i 个单词共生的单词的编号。

3. N 个浮点型向量 value[N],value[i]包含 num[i]个元素，元素值表示跟第 i 个单词共生单词的互信息值。

在 CPU 上计算单词相似度矩阵的过程如下：首先枚举两个单词（设为 i 和 j），然后根据公式（3.8）我们要获取 sumi（第 i 个单词的所有互信息的和），sumj（第 j 个单词的所有互信息的和）和 sum（第 i 个单词的互信息向量和第 j 个单词的互信息向量共同出现的项的和），求出这三个值后 sum/（sumi+sumj）便是单词 i 和 j 的相似度。在 CPU 上压缩存储方式的相似度计算算法大致如下所示：

Algorithm：CompressedWordSimCPU
Input：word number *N*；array *num*[]，*index*[][]，*value*[][]
Output：similarity matrix：*sim*[][]

```
for i from 0 to N – 1 do
    for j from i + 1 to N – 1 do
        sumi <- 0, sumj <- 0, sum <- 0
        for x from 0 to num[i] – 1 do
            sumi <- sumi + value[i][x]
        for x from 0 to sum[i] – 1 do
            sumj <- sumj + value[j][x]
        x <- 0, y <- 0
        while x < num[i] && y < num[j] do
            if index[i][x] < index[j][y] do
                x <- x + 1
            if index[i][x] > index[j][y] do
                y <- y + 1
            if index[i][x] == index[j][y] do
                sum <- sum + value[i][x] + value[j][y]
                x <- x + 1
                y <- y + 1
        sim[i][j] <- sum / (sumi + sumj)
```

相对于压缩矩阵的存储方式，减少列数的存储方式就显得简单而优美的多了，假设删减后的列数为 F 个，那么我们只需要一个 $N \times F$ 的二维数组 mi[N][F] 就可以完整保存整个互信息的矩阵了。

在 CPU 上删减列数形式下的相似度计算非常简单，代码结构大致如算法 ColReducedWordSimCPU 所示。除了最外部两个简单的循环之外，第 5~13 行就是具体的计算两个单词相似度的代码，可以看到每次计算都需要迭代 $\Theta(F)$ 次，因此整个的计算过程耗时非常清晰，就是 $\Theta(N^2 \times F)$。

Algorithm：ColReducedWordSimCPU
Input：word number：*N*；col number：*F*；array *value*[][]
Output：similarity matrix：*sim*[][]

for i from 0 to *N* – 1 do
  for j from i + 1 to *N* – 1 do
  sumi <- 0, sumj <- 0, sum <- 0
  for x from 0 to F - 1 do
  sumi <- sumi + *value*[i][x]
  for x from 0 to F - 1 do
  sumj <- sumj + *value*[j][x]
  for x from 0 to F – 1 do
  sum <- sum + *value*[i][x] + *value*[j][x]
  sim[i][j] <- sum / (sumi + sumj)
  end for
end for

现在还剩最后一个问题：输出结果太大。我们计算一下输出结果的大小，假设我们需要计算 100000 个单词之间的相似度矩阵，用一个 float 值存储相似度的值，因此我们需要的空间就是 $100000*100000*4=4*10^{10} \approx 40$GBytes，而且这还是连续紧凑存储的容量，这显然太大了。

那么计算结果 sim 矩阵到底如何存储呢？分两种情况，如果我们分析的单词数量不多，那么可以直接存储二维相似度矩阵。如果分析的单词数量很多，只要

保存 TopK 个结果。保存 TopK 个结果可以这样来处理：假设有 N 个单词，对于每个单词我们维护一个按照相似度排序的最小堆的优先队列 pq，这样每次计算得到一个单词对的相似度（W_1，W_2，sv）时，我们只需要把他们添加到 W_1，W_2 的队列中，具体过程如下伪代码所示：

Algorithm：UpdateTopK
Input：Two word: W_1，W_2；Similarity value：sv；K
Output：NULL
get the priority queue of W_1: pq if pq.size < K then 　　pq.push() else if pq.top.sim_value > sv then 　　pq.removeTop() 　　pq.push({W2, sv})

因为优先队列是按照相似度（sim_value）排序的最小堆，所以我们获得的堆顶元素就是队列中和 W_1 相似度最小的单词，只需要比较 sv 是否比这个值更大就可以做出判断了。鉴于 Priority Queue 的 push 是一个 O(log(K))复杂度的操作，可以发现每次调用 UpdateTopK 的复杂度也是 O(log(K))。

3.4 利用 GPU 加速的 DISCO 计算

本节将详细讨论利用 GPU 加速的单词相似度矩阵计算问题。由于对语料库的预处理、分词和计算互信息矩阵并不是单词相似度计算的主要耗时部分，我们也并没有对其进行特殊处理，因此我们在这一节将着重讨论由互信息矩阵计算得到相似度矩阵的过程，其他环节不予考虑。

在给出具体的 GPU 加速方案之前，有几个问题是我们需要考虑的：

1. GPU 存储容量有限。目前的 GPU 的存储容量还远远比不上内存的容量，N 个单词的相似度矩阵就是 N*N 的大小，如果我们需要计算的 N=100000 或者更大的情况，一次性将数据全部导入到显存中这种方法显然就不太

现实，也不符合可扩展性的要求。

2. GPU 全局存储器不带缓存机制，访问缓慢。为了提高效率，必须尽量减少全局存储器的访问次数。

出于 GPU 存储容量有限考虑，我们先要对单词互信息矩阵分块。分块的方案有几种，可以是固定模式的分块，比如可以这样：第 1~1000 号单词为一块，第 1001~2000 号单词为一块，以此类推。也可以是自适应地分块，比如我设定每个块的数据是 100Mbytes，那么实际的分块情况可能是第 1~936 号单词为一块，第 937~2019 号单词为第二块，以此类推。由于我们需要计算所有单词对之间的相似度，在分块之后我们就需要对所有块进行两两计算了。计算完了之后整理数据并存储下来。

至此我们可以大致给出 GPU 加速的具体流程，如图 3.8 所示：

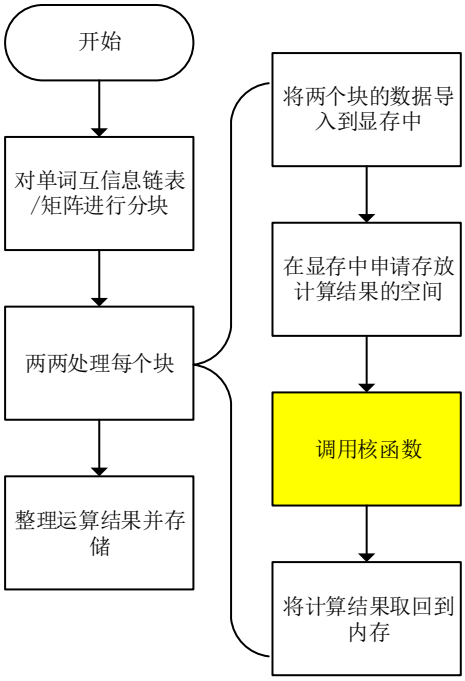


图 3.8 GPU 加速的总体流程

上图中调用核函数一步是加速的关键，而这一步恰恰就是实现的难点。我们对互信息矩阵的压缩存储和删减列数两种方式分别作介绍。

3.4.1 删减列数形式下的核函数

首先考虑如何分块。删减列数的情况下，互信息矩阵是一个规则的 N*F 的矩阵，因此我们采用固定单词数量的分块方案是最好的。

现在假设我们对互信息矩阵分块之后，每个块包含 B 个单词，也就是说每次我们会把 2 个 B*F 大小的矩阵导入到显存中，并且计算得到一个 B*B 大小的结果出来，每一对的计算需要 F 次的迭代。这样的处理过程跟我们熟悉的矩阵乘法非常的相似，他们的对比图如图 3.9 所示：

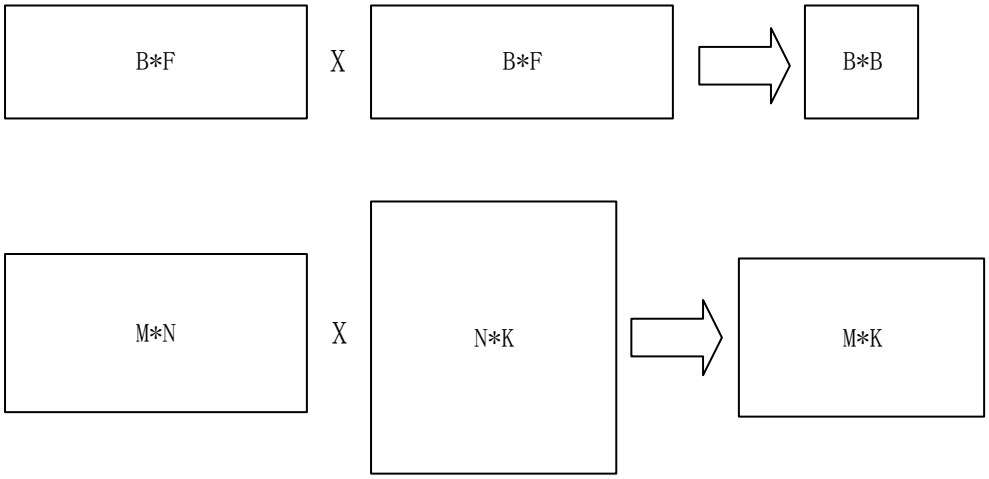


图 3.9 相似度计算和矩阵乘法对比

因此我们可以借鉴 GPU 上矩阵乘法的优化方式来类比单词相似度计算。找到 CUDA5.0 的例程 MatrixMul，其示意了一个 160*320 的浮点数矩阵和另一个 320*320 的浮点数矩阵相乘的 GPU 并行实现。它实现的大致思路是分成 10*10 的线程块（Block），每个块的内部是 16*16 的线程阵列（Thread），这样一来，每个大小为 16*16=256 的线程阵列可以使用一个大小为 48KBytes 的共享存储器，然后通过这 256 个线程的协同工作一起计算 256 个结果矩阵里面的值。

回到本文讨论的问题上来，从图 3.9 相似度计算和矩阵乘法对比可以看出，其实两个问题具有非常高的相似性，只不过我们面临的情况可能要再复杂一些，比如 B 和 F 的值并不是一个 2 的幂次的值。根据本问题的特殊性，我们可以挖掘一些性质：首先我们可以确定 F 是一个 5000 到 10000 大小的值，根据我们分块的特性，B 的值大概在 2000 到 4000 不等，另外还有一个更特殊的性质，那就是

结果矩阵是一个 B*B 大小的方阵。因此，我们完全可以直接启动一个 B*B 大小的线程阵列，每个线程负责计算一个结果。然后我们把这 B*B 个线程分成 B/T * B/T 大小的 Block 阵，每个 Block 包含 T*T 个线程。由于每个 block 拥有的共享存储器大小是有限的，只有 48KBytes，因此必须控制好 T 值的大小，一般 T 的大小设置为 64 以下。删减列数情形下核函数线程分配情况如下图所示。

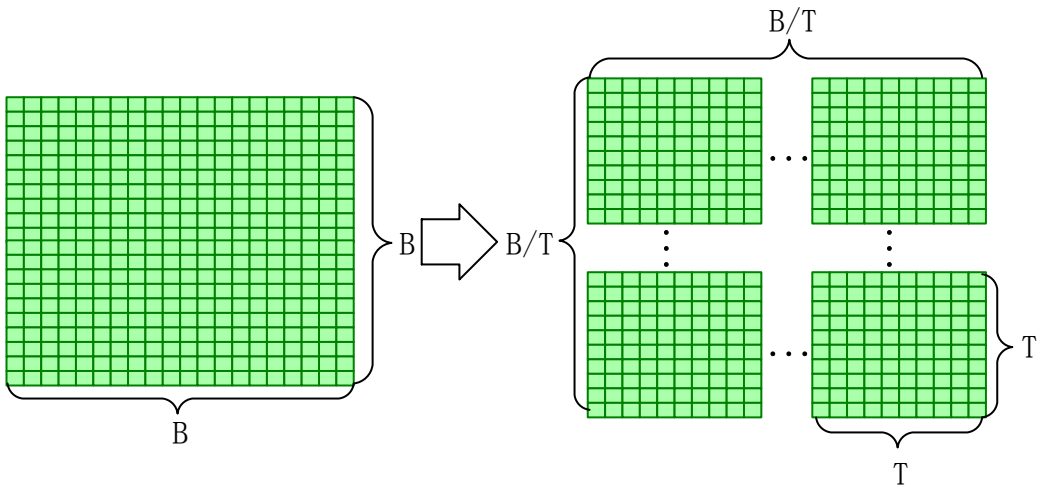


图 3.10 删减列数情形下核函数线程分配情况

下面考虑每个线程的执行情况，首先是每个 Block 里面的 T*T 个线程协同工作，一起计算 T 个单词和另外 T 个单词之间的相似度。这样我们就可以每次把 2 个 T*T 的矩阵数据先导入到共享存储器，然后计算 T*T 个单词对之间的相似度，虽然总体复杂度是 O(T*T*T)，但是由于我们有 T*T 个线程同时工作，所以总体复杂度只有 O(T)。下面给出核函数的伪代码：

Algorithm：ColReducedWordSimGPUKernel
Input：column number *F*，*data1*，*data2*
Output：the word similarity result between data1 and data2

(bx, by) <- get block id in the 2-dimensional thread block matrix
(tx, ty) <- get thread id in block
T <- get thread block size // there are T*T threads in each block
sum1 <- 0, sum2 <- 0

```
For c from 0 to F, step = T:
    Allocate shared memory a[T][T] and b[T][T]
    a[tx][ty] <- data1[bx * T + tx][c + ty]
    b[tx][ty] <- data2[by * T + tx][c + ty]
    synchronize all T*T threads
    For k from 0 to T
        if (ty == 0) sum1 <- sum1 + a[tx][k]
        if (ty == 0) sum2 <- sum2 + b[tx][k]
        if (a[tx][k] > 0 && b[ty][k] > 0)
            sum <- sum + a[tx][k] + b[ty][k]
    end for
    synchronize all T*T threads
    put tuple {bx * T + tx, by * T + ty, sum/(sum1 + sum2)} to result buffer
end for
```

3.4.2 压缩存储形式下的核函数

同样地，我们要先确定采用什么样的分块策略。由于在压缩存储形式下，每个单词的互信息向量其实是不一样长的，采用固定的分块策略可能会导致有些块的数据量很小，而有些块的数据量很大。因此，我们采用自适应的分块策略，保证每个块的数据量在 300MBytes 左右。

一个很大的互信息矩阵 mi_matrix 被分割成了很多的块，依据图 3.8 GPU 加速的总体流程所述，我们两两地把每个块倒入到显存中，并计算他们包含的单词之间的相似度矩阵。假设第一块数据包含 X 个单词的互信息向量，第二块数据包含 Y 个单词的互信息向量，这样我们会得到 X*Y 个计算结果。那么另一个可能的问题又来了，那就是如果某一块数据包含的向量都是非常断的，那么 X 或者 Y 就可能非常大，他们的乘积也就是结果数组就可能非常大，因此在分块的时候需要再考虑一点就是最大块内单词数 X 满足 X*X 小于某个值，保证显存能够存下计算结果。

下面讨论如何分配每个线程的计算任务使得计算效率尽可能高。根据算法

CompressedWordSimCPU 中描述的，压缩存储情况下计算相似度是比较复杂的，并不是非常优美的迭代过程，因此我们不能照搬 ColReducedWordSimGPUKernel 算法的过程。既然没法像删减列数那样完美地利用共享存储器带来的高访问速度，那我们就必须改变核函数的设计策略。由于在压缩存储形式下，计算两个单词的相似度需要遍历两个单词的整个互信息向量，因此我们可以尝试把其中一个向量加载进共享存储器，并且每个 Block 里面的线程协同计算这个单词和其他单词之间的相似度计算，这样至少把一半的显存访问加速了。

经过上面的论述，我们可以这样设计我们的核函数：启动 B*T 个线程，分成 B 个 block，每个 Block 包含 T 个线程。每个 Block 负责计算第一个互信息矩阵 mi_data1 里面的一个单词和另一个互信息矩阵 mi_data2 里面所有单词之间的相似度计算。这样每个 Block 可能会计算 mi_data1 里面多个单词和 mi_data2 中单词的相似度计算，这样我们就需要一个循环来迭代这个过程。在计算 mi_data1 中的一个单词对 mi_data2 中所有单词相似度的时候同样需要用循环来迭代。再有就是每个 Block 的共享存储起来保存一个 mi_data1 单词的互信息向量（太长的话只能保存部分）。下面给出压缩存储形式下的核函数伪代码：

Algorithm: CompressedWordSimGPUKernel
Input: two part of mutual information matrix: mi_data1, mi_data2
Output: the word similarity result between mi_data1 and mi_data2

num1 <- get word number in mi_data1
num2 <- get word number in mi_data2
tot_blocks <- get total thread blocks
threads_in_block <- get thread number in every block
block_id <- get block id that current thread belongs to
thread_id <- get thread id in current thread block
for i from 0 to num1/tot_blocks + 1:
  w1 <- i*tot_blocks + block_id
  if w1 >= num1 then
  break
  load w1-th data to block shared memory, named sw1
  sum1 <- get sum of all elements in sw1

```
    for j from 0 to num2/threads_in_block
        w2 <- j*threads_in_block + thread_id
        if w2 >= num2 then
            break
        sum2 <- get sum of all elements in w2-th array in mi_data2
        sum <- get sum of common part of sw1 and w2-th array in mi_data2
        put the value tuple{w1, w2, sum/(sum1 + sum2)} to result
    end for
end for
```

3.5 本章小结

本章主要论述基于 GPU 的单词相似度计算加速算法。首先给出了问题的定义，为什么需要计算单词之间的相似度，有什么用处。然后从原始的计算方法开始，慢慢引出不用关系语法分析的 DISCO 的计算方法，之后分析了 DISCO 方法的一些局限性和优化的方法，最后针对 CPU 版本的 DISCO 计算方法提出利用 GPU 加速的方案，根据不同的互信息矩阵形式，设计了不同的 GPU 加速方案。

第4章 基于 GPU 的近似重复文档检测算法

4.1 问题定义

本章讨论的问题是文档集合的近似重复检测算法。

首先给出近似重复文档的定义，本文所讨论的近似重复文档指的是文档 A 经过少量的修改（增加字符，修改字符和删除字符）得到文档 B，那么我们说文档 A 和 B 在一定程度上是近似重复的，这个修改的幅度我们用“编辑率”来计算，下面会做详细的介绍。

假设我们有一个文档集合，其中包含 N 篇文本文档，我们希望找出其中的近似重复文档对。形式化地定义这个问题：给定 N 篇文本文档$\{doc1, doc2, ..., docN\}$和一个阈值 P，我们要计算编辑率小于 P 的结果集合：

$$\{(doci, docj) \mid edit\_rate(doci, doj) < P\}$$

其中编辑率定义如下：

$$edit\_rate(doci, docj) = edit\_distance(doci, docj) / (length(doci) + length(docj))$$

上面式子中$edit_distance(doci, docj)$表示文档 doci 和文档 docj 转化到字符串之后的编辑距离。$length(doci)$表示文档 doci 转化成字符串之后的长度。

可以发现，如果 doci 和 docj 完全一样，那么他们的编辑率 edit_rate 等于 0。如果 doci 和 docj 文本几乎都不一样，那么他们的编辑率是一个大于等于 0.5 的值。因此，我们想要近似重复文档对的时候，只需要把 P 设置成一个相对较小的值（比如 0.05）就可以了。

4.2 原始计算方法

首先考虑这个问题的直接计算方法。根据 4.1 节的问题定义，最直接的计算方法就是两两枚举文档，然后计算他们的编辑率，其中计算编辑率的一步需要计算两个字符串的编辑距离。计算两个单词的编辑距离是一个动态规划问题并且已

经被广泛的研究过，下面直接给出其计算的过程:

Algorithm：EditDistance
Input：string A，B
Output：edit distance of A and B

```
lenA <- get length of A
lenB <- get length of B
allocate 2-dimension buffer array dis[lenA][lenB] filled with INF
for i from 0 to lenA do
    dis[i][0] = i
for j from 0 to lenB do
    dis[0][j] = j
for 1 from 1 to lenA do
    for j from 1 to lenB do
        cost <- 1
        if A[i - 1] == B[j – 1] then
            cost <- 0
        dis[i][j] = minimal(
            dis[i – 1][j] + 1,
            dis[i][j – 1] + 1,
            dis[i – 1][j – 1] + cost
        )
    end for
end for
return dis[lenA][lenB]
```

可以发现上面计算两个字符串编辑距离的算法复杂度是 O(lenA*lenB)。也就是说，如果 N 篇文本文档的平均长度是 S 的话，整个的计算过程的复杂度将达到 $O(N^2*S^2)$。

4.3 基于 CTPH 的近似重复文档检测算法

经过上面的分析，原始的近似重复文档检测算法太过耗时，并不具有实用价值。这节将介绍一种大大优化暴力算法的方法。CTPH（Context Triggered Piecewise Hash）是指根据上下文触发的分段哈希算法。分段哈希算法是相对全文哈希算法而来的，全文哈希算法每一篇文档只对应一个哈希值，而分段哈希算法把文档切分成很多段，比如第 0~1023 字节为一段，下一个 1024 字节为一段，每一段传统哈希方法（比如 MD5）计算得到一个哈希值，这样一篇文档就对应了一个哈希值序列。上面说的是分段哈希算法采用的是固定的分段方式，而 CTPH 采用的是根据上下文来分段。

CTPH 的根据上下文分段方法依赖的是滚动哈希算法。滚动哈希算法就是根据当前输入的最后若干个字节来产生一个随机哈希值得算法。假设我们的输入是 n 个字符，第 i 个字符用b_i表示，那么整个输入就可以表示成$b_1,b_2,..b_i..,b_n$。在输入串的任何位置 p，我们考虑其之前的 s+1 个字符，将这些字符作为滚动哈希函数的参数，得到一个滚动哈希值：

$$r_p = F(b_p, b_{p-1}, ..., b_s)$$

由于我们需要循环处理整个输入串，这个哈希函数要能够以非常少的计算代价得到r_{p+1}，也就是：

$$r_{p+1} = F(b_{p+1}, b_p, ..., b_{s+1}) = G(r_p, b_s, b_{p+1})$$

满足 G 是一个常数计算复杂度的函数。这样以来就可以在 O(n)的时间内计算得到所有的滚动哈希值。

举一个非常简单的滚动哈希函数的例子：

$$r_p = (10^0 \times b_p + 10^1 \times b_{p-1} + ... + 10^{p-s} \times b_s)\%97$$

式子中的%是整数取模运算，那么它的迭代计算就是这样：

$$r_{p+1} = G(r_p, b_s, b_{p+1}) = ((r_p - 10^{p-s} \times b_s) \times 10 + b_{p+1})\%97$$

它可以进一步化简成：

$$r_{p+1} = (r_p \times 10 - 10^{p-s+1}\%97 \times b_s + b_{p+1})\%97$$

其中$10^{p-s+1}\%97$可以预计算得到。

下面回到 CTPH 算法中，在处理文档数据时，依次计算每个位置的滚动哈希值，当滚动哈希值正好满足某个条件的时候就会发生分段，在整个文档的处理过程中，可能会有多次滚动哈希值满足这个条件，因此文档就被分成了很多段，每一段采用传统哈希算法计算得到一个哈希值，这样一篇文档就可以对应一个哈希值序列了。举个例子，我们令 s=6，分段的条件可以是当滚动哈希值 r 模 B 的值为 B-1 的时候发生分段。再进一步，假设滚动哈希值是完全随机的，那么 r%B=B-1 的期望段长度是 B，因此我们还可以通过调整 B 的值来控制文档被分割的段数，进而控制输出的哈希序列长度。

现在我们已经知道可以利用滚动哈希来对文档进行分段，并且能够通过修改 B 的值来控制输出哈希序列的长度，那么我们如果把所有文档都用 CTPH 算法进行分段哈希得到一个长度短的多的签名，然后再应用我们在上一节提出的暴力算法就快多了。为了便于后面编辑距离的计算，我们把输出的 signature 的字符集定为大小写拉丁字母和数字共 64 个字符。控制输出序列长度的方法就是：如果长度太长就倍增 B 的值，这样 signature 的长度就会急速降低。下面直接给出它的伪代码：

Algorithm：CTPH
Input：document sequence：*input*，max signature length *S*
Output：hash sequence：*signature*

B <- 1
while TURE do
r <- 0 //rolling hash value
tr <- 0 //traditional hash value
signature <- “”
foreach byte d in *input* do
r <- update_r (r, d)
tr <- update_tr(tr, d)

```
        if r%B == B-1 then
            signature <- signature + tr%64
            tr <- 0
    end foreach
    if length(signature) <= S then
        break
    else
        B *= 2
end while
return signature
```

下面证明为什么原始串的编辑率可以转化成分段哈希签名串的编辑率。为了便于说明，我们把两篇文档分别记为 docA 和 docB，docA 经过若干修改变成 docB。我们分三种情况来讨论：

1. docA 和 docB 完全一样
2. docB 的修改点不在分段触发点上
3. docB 的修改点在分段触发点上

第一种情况，由于两篇文档完全一样，而我们的 CTPH 算法是一个完全确定的算法，因此他们得到的 signature 也应该是完全一样的，编辑率为 0。

第二种情况，修改点没有在触发点上，那么这个修改只影响到了前后两个触发点之间的这一段，这样在 signature 中相当于之修改了一个字符。当然也会有 1/B 的概率会多出一个触发点，就算这样也同样不会影响到前一个触发点之前和后一个触发点之后的分段情况，然后这种情况下，signature 的编辑距离会多 1。

第三种情况，如果修改点在触发点上，那么这个修改点最多影响 docA 的 signature 中的两个字符，与第二种情况类似，第 i 个触发点改动了，但是第 i-1 个和第 i+1 个触发点还是依然会触发使得发生分段。

还有一个问题就是两个非近似重复文档被错误判定为近似重复的可能性。根据 CTPH 算法的描述，如果两个本来不相同的文档偶然地因为哈希函数的碰撞使得他们的 signature 的编辑率很低了，这样的概率是非常低的。假设 signature 的长

度是 60，编辑率是 0.1，那么至少 60*(1-0.1)=54 个字符发生了偶然哈希碰撞，而发生一次碰撞的概率是 1/B，一般文档长度达到 1000bytes 的情况 B 就会达到 30，那么发生这种误判事件的概率是 $1/30^{54}$，这是一个极小极小的概率，几乎是不可能的。

下面介绍基于 CTPH 算法的近似重复文档检测是如何实现的。首先对于给定的 N 篇文档和编辑率阈值 P，我们选择一个合适的 S（一般设置为 100），使得所有的文档经过 CTPH 算法之后得到一个长度刚好小于 S 的 signature 字符串，然后对于 N 个 signature 串，我们两两计算他们之间的编辑率，由于从文档转变成 signature 之后编辑率可能会有轻微的提高，我们适当放大 P 的值（比如 3 倍）作为 signature 之间的编辑率的阈值。这样计算之后，我们会得到一些 candidate 的近似重复文档对，这些文档对不一定满足编辑率小于 P 的条件，需要做进一步的验证，也就是再计算一遍原文档之间的编辑率。具体的计算过程如下伪代码所示：

Algorithm：CTPHBasedHomoDocsDetectCPU
Input：documents *docs*，max signature length *S*，edit rate threshold *P*
Ouput：homologous document pairs *homos*

N <- get document number from docs
signatures <- {}
for i from 0 to N-1 do
　　signature.put(CTPH(*docs*[i], S))
candidates <- {}
for i from 0 to N-1 do
　　for j from i + 1 to N – 1 do
　　　　if calc_edit_rate(signatures[i], signatures[j]) < 3*P
　　　　　　candidates.put((i, j))
　　end for
end for
homos <- {}
foreach (i, j) in candidates do

```
    if calc_edit_rate(docs[i], docs[j]) < P
        homos.put((i, j))
end for
return homos
```

分析一下 CTPHBasedHomoDocsDetectCPU 算法的复杂度，假设所有文档的平均长度是 A，那么每次 CTPH 算法的执行复杂度约为 $\Theta(A*\log_2\frac{A}{S})$，其中 $\log_2\frac{A}{S}$ 是 B 的迭代次数。计算 signature 之间的编辑率的复杂度为 $\Theta(N^2*S^2)$，最后一步是跟输出结果大小相关的一个复杂度，我们记为 $\Theta(output*A^2)$，那么 CPU 上的近似重复文档对检测算法的总时间复杂度为 $\Theta(N*A*\log_2\frac{A}{S}+N^2*S^2+output*A^2)$。可以发现，中间的一项是整个复杂度的主要部分，因此优化这个部分就能够极大地提升整个算法的效率。

4.4 利用 GPU 加速的近似重复文档检测算法

根据上一节的分析，基于 CTPH 的近似重复文档检测算法还是有很大的优化余地的。其中耗时最大的部分是两两计算 signature 的编辑率，这部分的计算跟第三章讨论的单词相似度计算有类似之处，只不过单词相似度计算的时候两两枚举完了只需要 O(N)的遍历就行，而这里需要一个 O(S*S)的动态规划子过程，可以说近似重复文档检测比单词度相似性计算更加难以放到 GPU 上进行加速计算。

下面就文档结合包含 N=100000 篇文档，并且文档平均长度为 10KBytes 的情况进行讨论。

首先考虑空间问题，第一个是源文档集合的大小，由于文档平均长度为 10kbytes，那么全部文档大小大概是 10k*100000=1Gbytes，并不算大，所以就算把整个文档集合导入到显存中也是可以接受的，不过出于扩展性的考虑，我们在真正实现过程中并没有这么做。第二个是经过 CTPH 算法得到 signature 序列的大小问题，如果我们取 S 的值为 100 的话，那么也就是说所有的 signature 占用的空

间大概就是 100*100000=10Mbytes，是一个非常小的数值。因此，经过处理得到的 signature 数据还是完全可以导入到显存中的。

下面再来考虑如何将第二部分放到 GPU 中加速，由于字符串编辑距离的计算是一个动态规划算法，后面的值的计算需要依赖前面的值，因此计算一对字符串之间的编辑距离不适合多个线程协同工作完成，只能交由一个线程单独迭代地完成。

4.2 节中介绍了字符串的编辑距离的算法，假设两个串的长度是 lenA 和 lenB 的话，那么 EditDistance 算法其实需要占用 O(lenA*lenB)大小的存储空间，而我们每启动一个线程就需要申请一个 lenA*lenB 大的空间，由于显存的大小是有限的，每个线程占用的空间越多，就会使得我们同时启动的线程数量越少，这对于充分利用 GPU 的核心计算资源是不利的。因此需要对 EditDistance 算法进行适当的改进，使得它的空间需求降低一些。具体做法就是把 dis 数组替换成一个滚动的数组，这样我们只需要 O(lenB)的空间就够了，具体的流程如下所示：

Algorithm：EditDistancePlus
Input：string A，B
Output：edit distance of A and B

lenA <- get length of A
lenB <- get length of B
allocate 2-dimension buffer array dis[2][lenB] filled with INF
for j from 0 to lenB do
　　dis[0][j] = j
now <- 0
for i from 1 to lenA do
　　dis[1-now][0] <- i
　　for j from 1 to lenB do
　　　　cost <- 1
　　　　if A[i - 1] == B[j – 1] then
　　　　　　cost <- 0

```
            dis[1-now][j] = minimal(
                dis[now][j] + 1,
                dis[1-now][j – 1] + 1,
                dis[now][j – 1] + cost
            )
        end for
        now = 1-now
    end for
    return dis[now][lenB]
```

现在已经确定了单个的编辑距离计算必须由单个线程来做，并且这个计算过程需要消耗 O(2*S)的空间。那么可以考虑的 GPU 计算方式已经不是很多，可能的方式有两种：

1. 把 O(2*S)的空间放在共享存储器中。
2. 把 O(2*S)的空间放在全局存储器中。

第一种方式把中间数组放在共享存储器中，好处是访问速度快，可以大大加快编辑距离的计算，缺点也很明显，那就是共享存储器其实是非常稀缺的资源，每个线程块（block）只有 48KBytes，因此如果是第一种方式的话，极大地制约单个 block 里面线程的数量，降低并行度。

第二种方式跟第一种恰好相反，把中间数组放到全局存储器中，虽然降低了数据读写的速度，但是单个线程块里面的线程可以非常多，这样 GPU 频繁的切换执行线程束可以掩盖数据访问的延迟，进而提高整体的加速比率。

可以说，上面的两种方式各有利弊，具体的执行效果会在后面的实验章节做具体的论述。

讨论完了计算的过程，最后剩下的就是计算结果如何存储。由于在 GPU 内部，高级的较复杂的数据结构都不会太好的运行效率，因此我们还是要把注意力集中在最朴素的数据结构：数组上。那么假设我们一次性调用 GPU 计算了 N=100000 个字符串之间的编辑距离，这将会产生 N*N=10G 个计算结果，这么大的计算结果并不好处理。因此我们依然采用分批次计算的方式，不过这次我们的分批方式

和单词相似度计算里面的有所不同，我们的过程类似这样：首先计算第 1~100 号字符串和其它全部 N=100000 个字符串的编辑距离，然后计算第 101~200 号跟 N 个字符串的编辑距离，依此类推。这样一来，我们每次产生 $100*N=10^7$ 个结果，并不是很大。

到此为止，我们已经把整个 GPU 加速近似重复文档检测算法的过程论述完毕，他的总体流程如下图所示：

Algorithm：CTPHBasedHomoDocsDetectGPU
Input：documents *docs*，max signature length *S*，edit rate threshold *P*
Ouput：homologous document pairs *homos*

N <- get document number from docs
signatures <- {}
for i from 0 to N-1 do
　　signature.put(CTPH(*docs*[i], S))
copy *signatures* into GPU global memory
STEP <- 10^7/N
candidates <- {}
for block_start from 0 to N-1 step=STEP do
　　call CTPHBasedHomoDocsDetectGPUKernel(block_start,STEP,N,S,P)
　　copy result from GPU to main memory
　　collect candidate pair and put into *candidates*
end for
homos <- {}
foreach (i, j) in candidates do
　　if calc_edit_rate(*docs*[i], *docs*[j]) < P
　　　　homos.put((i, j))
end for
return homos

4.5 本章小结

本章详细论述了基于 GPU 的近似重复文档检测算法的加速研究。首先给出了需要解决的问题的定义，并从暴力的方法开始一步一步将方向引到模糊哈希算法。最后根据模糊哈希最费时的一步的编辑距离的算法过程，研究它在 GPU 加速的可行性，最后设计了两种不同的 GPU 加速方案。

第5章 实验结果

本章将对本文涉及的两个主要算法基于 GPU 的单词相似度计算和基于 GPU 的近似重复文档检测算法的实际运行效果和相对 CPU 的加速比率进行评估。

5.1 实验环境

本实验机器的配置如下：处理器为 16 核心 32 线程的 Genuine Intel(R) CPU，主频 2.60GHz。内存为 64GBytes，频率 1600MHz。硬盘为希捷 2TBytes。显卡为 NVIDIA GeForce Titan X。操作系统为 64 位 Ubuntu，Linux 内核版本 3.13.0-24-generic。

5.2 数据的获取

首先是单词相似度计算的语料库。在本文的第 3 章我们已经对单词相似度计算的方法做了详细介绍，但是有一个很重要的问题并没有触及到，那就是语料库的获取。经过综合考虑，我们选择英文维基百科全文作为我们的语料库。这个语料库有以下几个特点：

1. 文档数量足够多，这是我们能够计算得到较为精准结果的保障。
2. 容易获取，维基百科把压缩的全文（包括历史修改信息）放在其官方网站上[1]，供开放下载。

从 http://dumps.wikimedia.org/enwiki/latest/下载最新的维基百科全文，解压缩之后得到一个 xml 格式的文件 enwiki-latest-pages-articles.xml，其中的主要结构如下图所示：

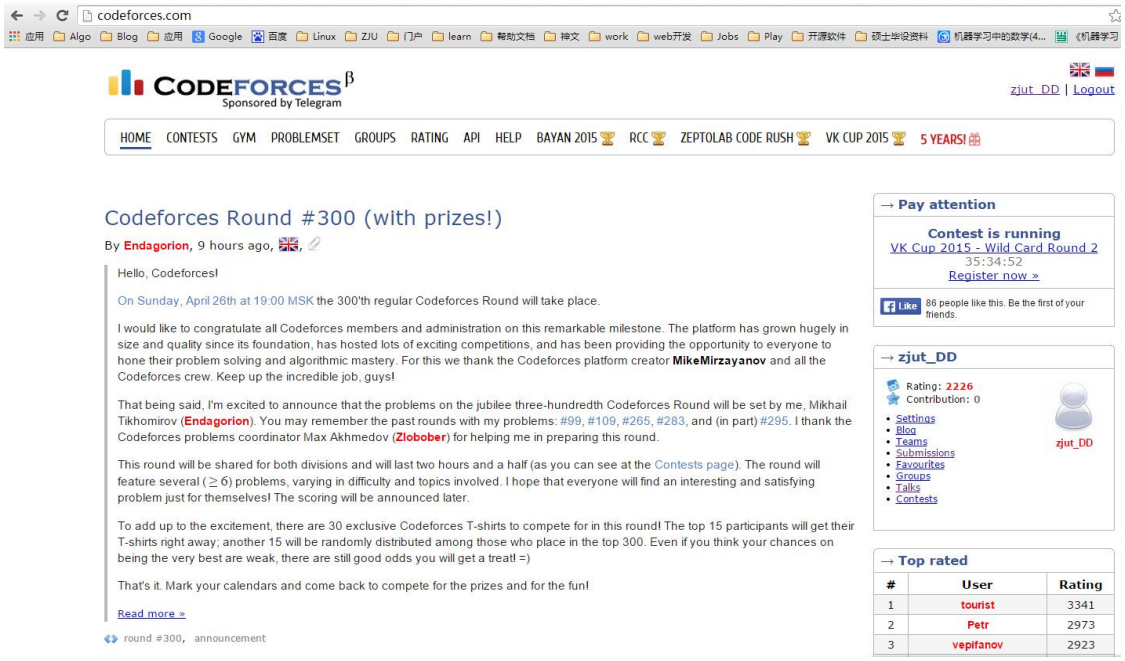
[1] http://dumps.wikimedia.org/enwiki/latest/

```
1   <page>
2     <title>Computer accessibility</title>
3     <revision>
4       <id>381202555</id>
5       <timestamp>2010-08-26T22:38:36Z</timestamp>
6       <comment>......</comment>
7       <text>.......</text>
8       <sha1>lo15ponaybcg2sf49sstw9gdjmdetnk</sha1>
9       ...
10    </revision>
11    ...
12  </page>
```

图 5.1 维基百科数据源主要结构

它描述了每个维基页面的内容，其中第 7 行的<text> </text>就是我们需要的维基百科正文所在的位置。其中的内容以行为单位，每一行就是一个维基百科里面的段落。段落里面还会包含很多的标记符号，比如图片标记、超链接标记等等，这些标记需要我们做特殊的处理，最方便的做法就是把这些位置的字符直接替换成空格，这样就只剩下常规的英文句子。顺序处理整个 enwiki-latest-pages-articles.xml，把所有的正文单独取出来，得到 1800 多万个段落，每个段落几十个单词到 1000 多个单词不等，保存成一个纯文本文档。

近似重复文档检测的数据来自一个在线程序评测网站 http://codeforces.com/。这个平台每周都会举办一次在线算法比赛，限时 2 小时，在两小时内参赛选手需要解出给定的 5 到算法逻辑题目，如果参赛选手提交的代码程序能够通过后台的所有测试数据，那么就算他解出了这道题。解出的题数越多排名越靠前，相同题数情况下，用时越少排名越靠前。由于一个选手在一场比赛里面可以对同一道题目多次提交代码，因此在很多情况下，参赛选手这次提交的代码跟同一题上一次提交的代码只改动了非常少的地方，这两个代码文件是近似重复的。更重要的一点，这个网站上用户提交的代码是所有人可见的。所以，如果我们能抓取到这个网站上的用户提交的代码文件的话，是非常适合作为我们近似重复文档检测的数据集的。



由于 CODEFORCES 的开放 API 中并没有提供直接下载用户提交的代码的接口，因此我们必须自己寻找方法来抓取代码。抓取方法如下：

1. 依次抓取足够数量的 Status 页面[2]，并从页面的 HTML 数据中提取出用户的提交（Submission）的元信息。
2. 根据用户的提交的元信息，直接获得代码查看页面[3]，并从页面 HTML 数据中提取代码。

上述过程中用户的提交的元信息主要包含 Contest 编号和 Submission 编号。迭代 Status 的 URL 的 page 参数，由于每一页包含约 30 条记录，为了满足实验的需要，我们把 page 号 10000 以内的 Status 页面都抓取下来，页面源码中<tr data-submission-id="817236">…</tr>的内容描述的是提交记录，从中找到相应的 Contest 编号和 Submission 编号。然后把抓取下来的 Contest 编号和 Submission 编号替换到源代码查看页面 URL 的相应位置，获得源码页面内容后，把<pre class="prettyprint program-source" style="padding: 0.5em;">...</pre>标签中间的用户提交的真正代码提取出来，直接把它们以单个文件的形式保存到磁盘目录中备用。

[2] http://codeforces.com/problemset/status/page/1?order=BY_ARRIVED_DESC
[3] http://codeforces.com/contest/443/submission/10844467

5.3 单词相似度计算实验

单词相似度计算的实验分为两部分，分别是互信息矩阵的删减列数形式和压缩存储形式。每种形式我们都做了 CPU 和 GPU 的对比实验，本章将对其中更具代表性的压缩存储形式做着重分析。并且在 GPU 的实现中，调用核函数时启动的 Block 数量、Thread 数量以及分块的大小都有一定程度的影响。在所有的实验中，单词数量分几个数量级，下面对这些情况分别作介绍。

在单词相似度计算的实验中，我们对单词数量为 5000，10000，20000，50000 和 100000 分别作了实验。

在压缩存储形式下，CPU 的执行时间随着单词数量的增加执行时间成倍地增长，这从它的执行复杂度 $O(N^2*L)$可以看出，其中 N 代表单词的数量，L 代表所有单词互信息向量的平均长度。而在 GPU 加速的情况下，算法的运行时间也出现了随着单词数量增长成倍增加的情况，但是由于 GPU 的高效性，它始终维持在比 CPU 快两个档次的水平上。下面给出它们的具体运行时间：

表 5.1 单词相似度计算运行时间（秒）

单词数量	5000	10000	20000	50000	100000
CPU	80.4	649.4	3327.9	10166.7	16731.3
GPU	3.1	15.8	69.0	200.7	322.6

上表给出了在不同的单词数量情况下，CPU 单机程序和 GPU 并行加速程序计算单词相似度矩阵的时间。可以看到 CPU 单机程序在单词量只有 5000 的时候运行时间已经达到了 80 秒，而 GPU 加速的程序在单词量达到 20000 时才花了 69 秒，可见他们的运行效率差别非常大。单纯看数字并不能明显看出差别，下面我们给出他们的曲线图：

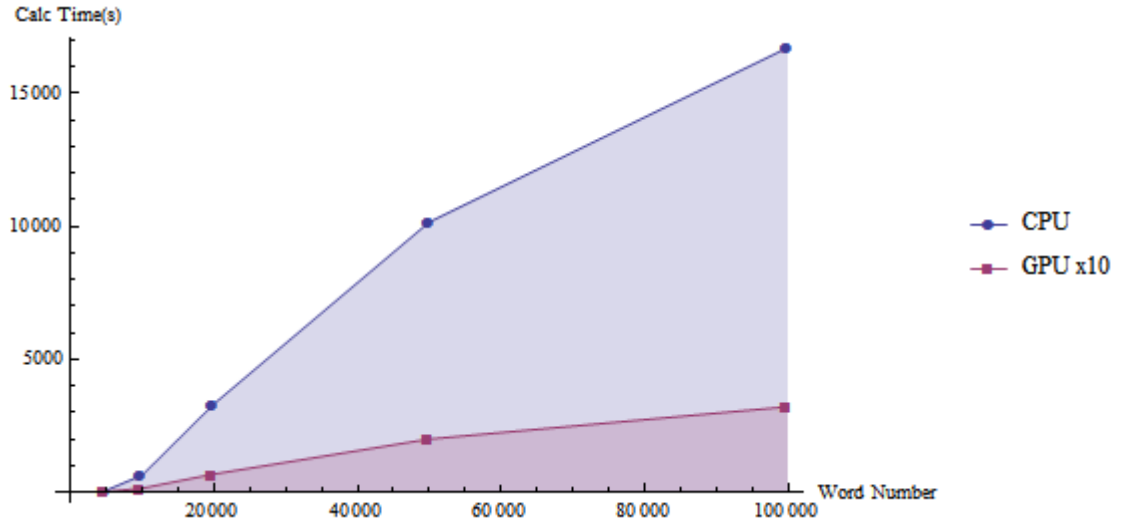


图 5.2 单词相似度计算运行时间对比图

上图中两条曲线非常明显地显示了 CPU 程序和 GPU 加速程序在运行时间上的差距，由于 GPU 加速程序相对 CPU 程序运行时间少很多，为了很好的分辨两条曲线的位置，所有我们只有把红色曲线的值乘以 10 之后加以标注。可以看到就算是乘以 10 之后的曲线还是可以看到非常大的改进，下图给出了 GPU 加速程序相对 CPU 单机程序的加速比率，这个比率的计算我们采用了一个非常简单的公式：

$$speed\_up = \frac{cpu\_exe\_time}{gpu\_exe\_time} \qquad \text{公式（5.1）}$$

就是 CPU 单机程序的执行时间除以 GPU 加速程序的执行时间，当然这个时间是不包含文档预处理。

图 5.3 展示了 GPU 加速程序相对于 CPU 单机程序的加速比率，单词量为 5000 的时候有 25 倍的加速，而单词量到达 10000 的时候加速比跃升到 41，之后逐渐趋于平缓，当单词量为 100000 的时候，加速比率为 51。

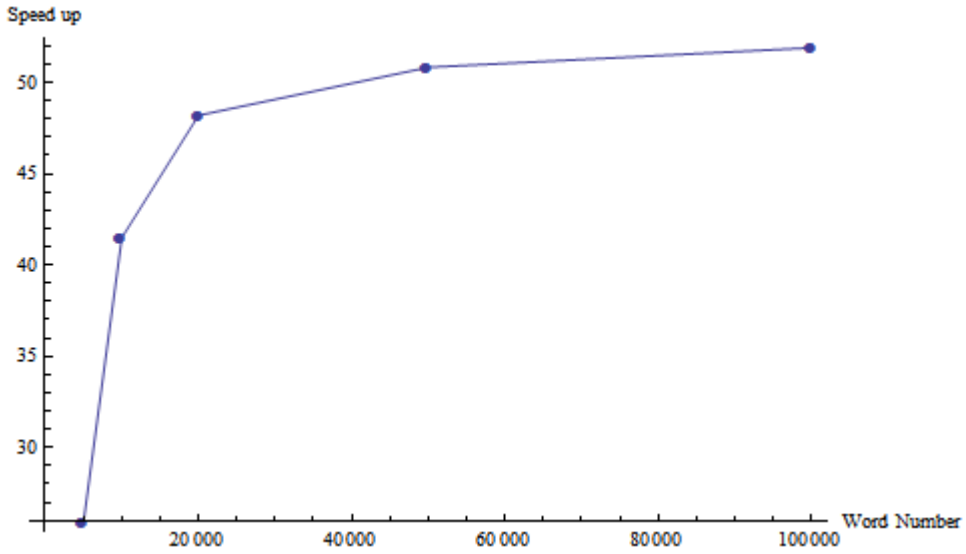


图 5.3 单词相似度计算加速比

以上是 CPU 单机程序和 GPU 加速程序的对比实验，下面论述 GPU 加速的内部参数对运行时间的影响。根据第三章的论述，在压缩存储形式下，我们采用了自适应的分段策略，也就是固定每一段的互信息响亮长度总和，这样一来每一段的单词数量是不一样的，这个参数对实验结果有一定的影响。此外核函数固定的情况下，调用核函数时启动的线程块 Block 大小和每个线程块的内部线程 Thread 的大小对运行结果也有影响，下面介绍 Block 和 Thread 数量对加速效果的影响。

前面的章节已经论述过，GPU 在实际执行的时候会对每个线程块 Block 内的所有线程分 Warp，每个 Warp32 个线程，这 32 个线程就同时运行同时停止，用不同的数据执行相同的指令。因此，我们实验的时候直接选取 Block 内线程数量是 32 的倍数，而对于 Block 的数量并没有明显的限制条件，为了便于研究我们也将它设为 32 的倍数。在实际的单词数量为 5000 的实验中，我们迭代了 32 到 320 的所有 32 的倍数作为 Block 和 Thread 的参数，得到的结果如下所示：

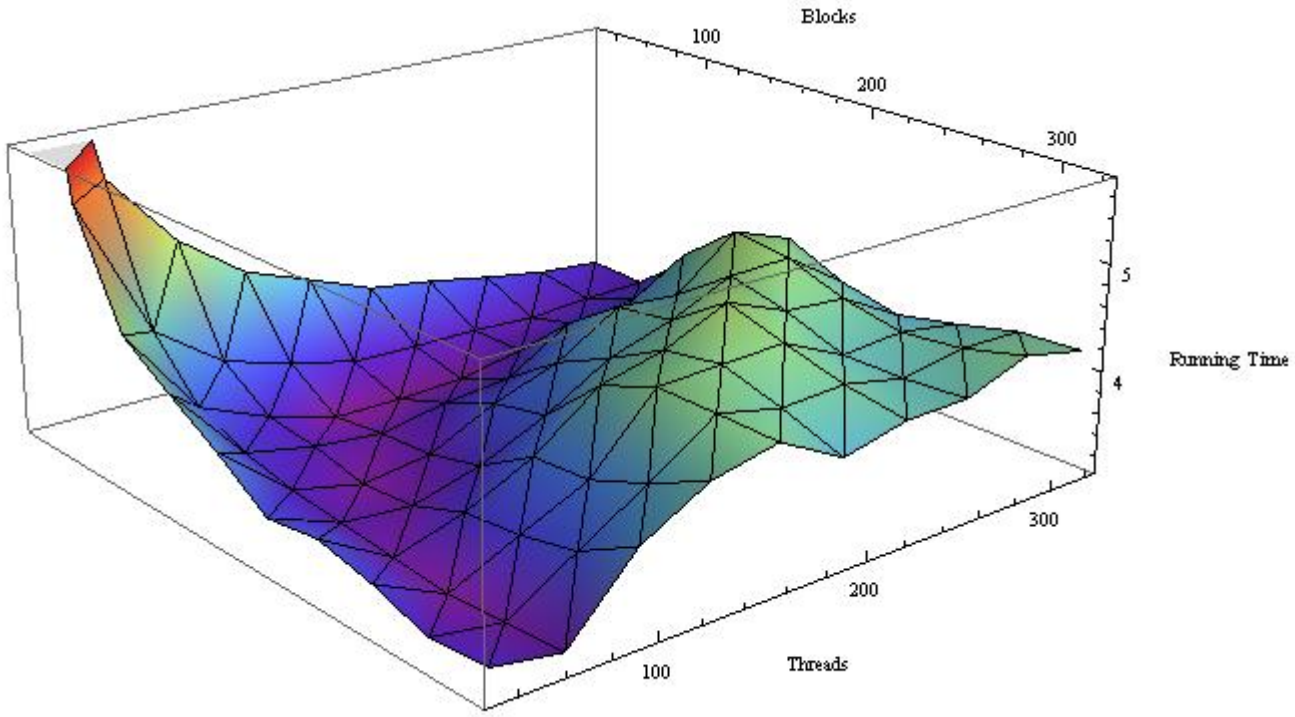


图 5.4 Block 和 Thread 对加速效果的影响

图 5.4 显示的是选用不同的 Block 和 Thread 参数时 GPU 加速算法运行时间的关系。图中蓝的发红的区域为耗时最短的区域，可以看到这一片区域基本符合一条双曲线的形状，在 Block 较小 Thread 较大、Block 较大 Thread 较小和 Block 跟 Thread 一般大的情况下都能得到最好的运行时间，因此我们可以做出合理的猜测，在单词相似度实验中 Block*Thread 约为某一个常量的时候我们能得到最好的运行时间，而当单词数量为 5000 时，这个乘积值约为 128*128=16384。这个结果也非常符合 GPU 本身的体系结构，我们设计的 GPU 程序需要大量地访问全局的存储器，总线程数量（也就是 Block*Thread）远超 16384 则引发访问冲突，远少于 16384 则不能充分发挥 GPU 的多核优势。

5.4 近似重复文档检测实验

在近似重复文档检测的实验中，由于数据计算量较大，我们选择的文档数分别为 1000，2000，5000，10000 和 20000。根据本文之前的分析，CPU 单机程序

近似重复文档的计算复杂度为 $O(N^2*S^2)$，其中 N 是文档集合的大小，S 是我们既定的模糊哈希算法的输出串长度。可以看到这是一个复杂度非常高的算法，在实际的实验中也得到了充分的体现。而 GPU 的加速实现有两种情况：

第一种是充分利用线程块 Block 里的共享存储器，将字符串和中间计算结果都放在共享存储器内，这样可以加速单词计算数据存取的时间。

第二种是充分利用 GPU 多线程的优势，尽量多地启动线程，将中间计算结果放全局存储器，利用 Block 内部多个 Warp 的切换来抵消对全局存储器存取的延迟。

我们实现了两种不同的核函数，并对他们的实验结果做了记录。下表展示了 CPU 和 GPU 在不同文档集合大小情况下的平均计算时间：

表 5.2 近似重复文档检测运行时间（秒）

文档数	1000	2000	5000	10000	20000
CPU	64.5	258.2	1614.6	6558.7	26293.5
GPU-1	1.34	4.65	30.87	126.18	508.38
GPU-2	1.10	3.82	23.67	92.68	360.37

从上表可以发现，各项计算时间基本都是按照文档数量增长率的平方倍数在增长，GPU 加速虽然大大缩减了总的计算时间，但是它毕竟没有减少整体的算法复杂度，只是充分利用了 GPU 的多核优势。下面给出两种 GPU 加速算法的具体加速比率，加速比率的计算依旧是按照公式 5.1。

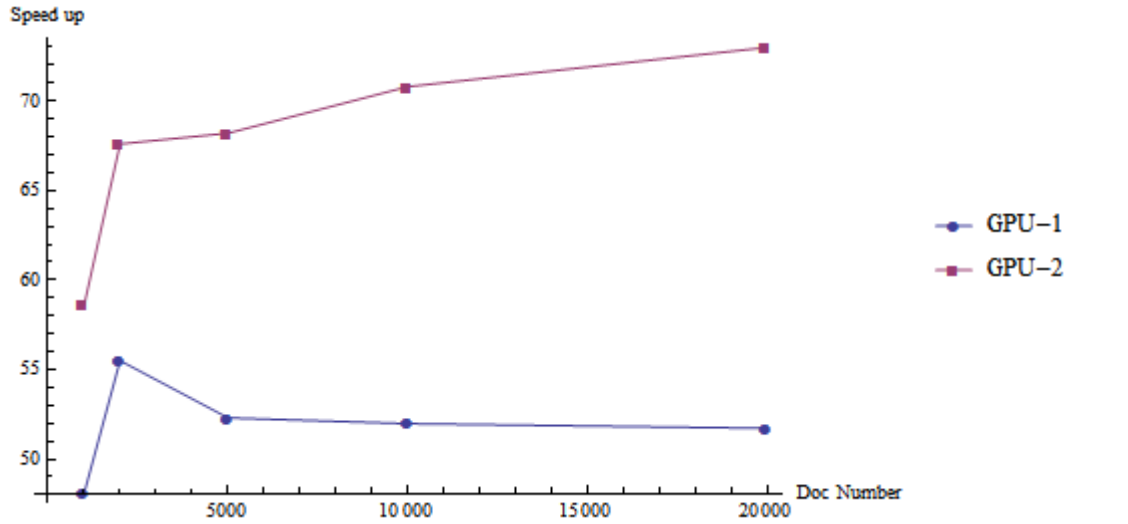


图 5.5 近似重复文档检测加速比

从上图可以看出，第一种加速算法（下面那条线）并没有想象中那么高的加速比，反而在文档数量增加的时候，出现了加速比不及文档数为 2000 的时候，究其原因是第一种加速算法过分依赖 Block 内部的共享存储器的缘故，由于每个 Block 内部共享存储器总共就 48Kbytes，因此它极大地缩减了每个 Block 内部线程的数量，随着文档数量的增加，加速比立即趋向于平稳。而第二种加速算法在最新的 GPU 体系结构下显得出色的多，在文档数增加时依然有微小的加速比提升，而且总体上比第一种加速算法高很多。

再来看不同 Block 和 Thread 参数对 GPU 加速程序的性能影响，首先看看第一种加速算法的情况，我们在 2000 篇文档的数据量下枚举了 32 到 412 的 Block 值和 16 到 96 的 Thread 值，Thread 值没法再次提升了是因为块内共享存储器的大小限制，最多只能启动 96 个块内线程。

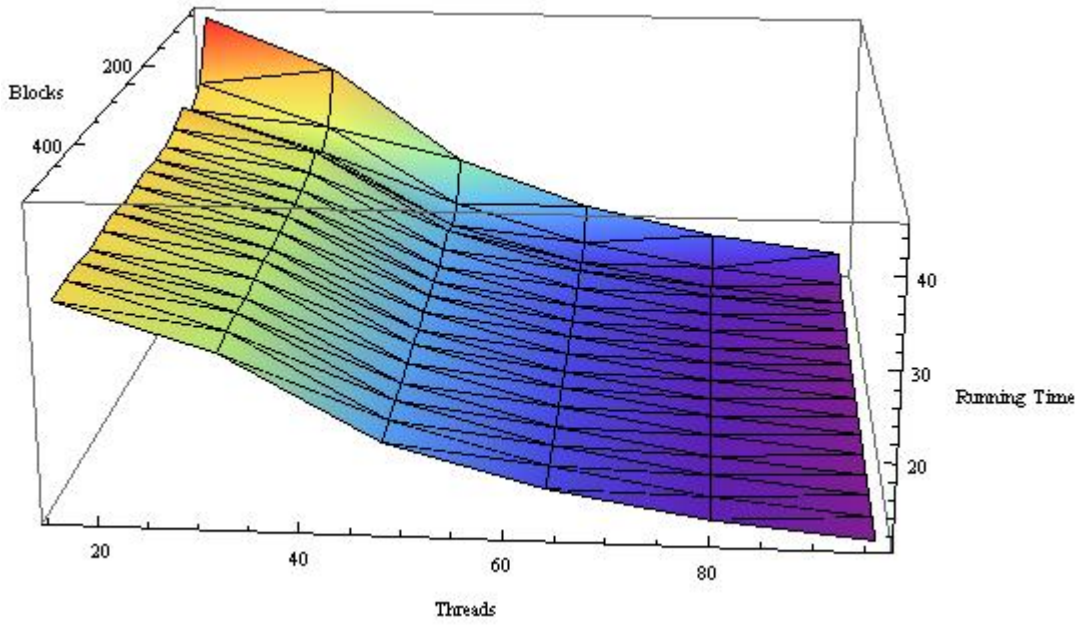


图 5.6 Block 和 Thread 对第一种核函数的影响

从上图来看，随着 Thread 个数的增加，运行时间急剧减少，这是因为每个 Block 拥有固定数量的共享存储器，当线程数量太少的时候，共享存储器并没有完全利用起来，当线程数量多到共享存储器完全利用，性能到达极限。再看 Block 方面，当 Block 达到 128 的时候出现了性能最优的情况，随着 Block 继续增大，运行时间并没有因此减少，可见 Block 的参数值并不是影响第一种加速方案效率的核心因素。从总体上来说，第一种加速算法太过依赖块内共享存储器的容量，在现有的 GPU 体系结构下很难再有提升的空间。而第二种加速方案就没有这方面的限制，下面给出其运行时间和 Block、Thread 的关系：

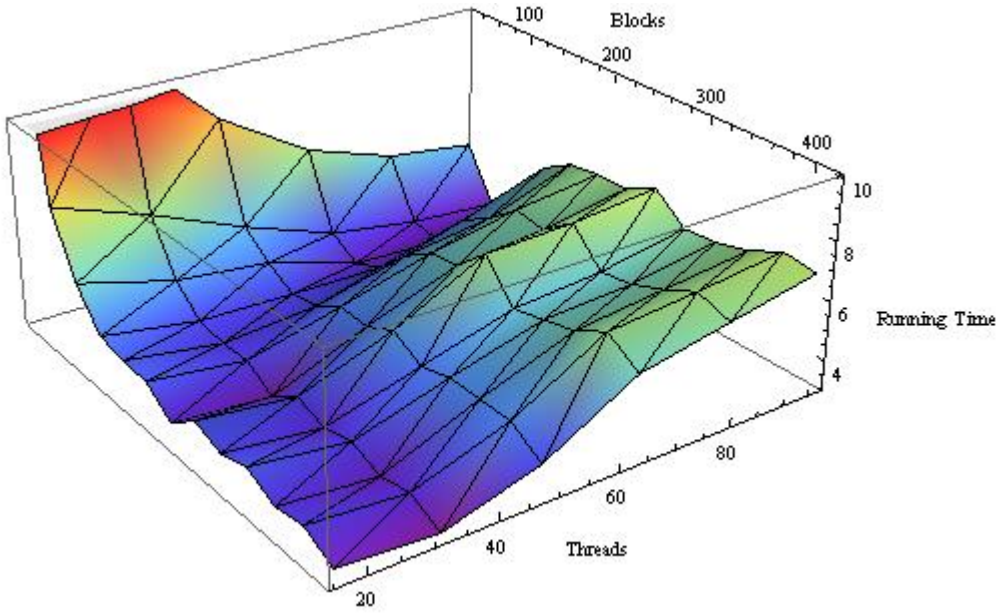


图 5.7 Block 和 Thread 对第二种核函数的影响

上图是第二种加速方案中 Block 和 Thread 参数对整体加速效果的影响，它跟 5.3 节中单词相似度计算对 Block 和 Thread 的依赖性非常相似。观察颜色最深的区域，可以看到当执行时间最少的时候，Block 和 Thread 大致成双曲线的关系，也就是 C=Block*Thread，其中 C 是一个常量。具体的原因是这样：当总共的线程数量（也就是 Block*Thread）远小于 C 的时候，并不能充分发挥 GPU 大规模核心的优势，造成大量的核心空闲浪费资源，导致执行时间延长；而当总线程数量远大于 C 的时候，由于这些线程都需要存取全局存储器，大量的并发访问引起数据访问冲突延迟增高，最终导致执行时间延长。

5.5 本章小结

本章主要是对本文研究问题的实践部分的讨论。5.1 节介绍所有实验进行的环境，5.2 节给出了实验数据的获取途径和方法，后面两节针对本文具体讨论的两个文本处理算法做的对比实验进行论述，给出了相对 CPU 单机程序的加速比率，讨论了 GPU 加速程序的中参数的影响以及他们背后的原因。

第6章 工作总结与展望

6.1 本文主要工作和贡献

本文主要就当前相对火热的 GPU 并行计算在文本处理相关算法方面的实践作了一定的研究，集中主要精力研究单词相似度计算和近似重复文档检测两个算法在加入GPU并行加速之后运行效率的变化，并且针对这两个不同的算法和GPU本身特有的性质设计了不同的核函数以期达到最佳的加速效果。

本文的主要贡献包括：

1. 完善和改进了基于共生关系的单词相似度计算算法，详细分析了算法的空间时间消耗，并针对耗时的关键部分采用 GPU 并行计算的方法进行加速，对不同的共生关系矩阵形式设计了不同的加速算法。
2. 分析文档集合近似重复文档检测的特点和 CPU 上的解决方案，提出利用 GPU 加速近似重复文档检测算法的可能性，设计了多种针对编辑距离算法的 GPU 并行实现。
3. 抓取了大量的实验数据，用实际数据做实验。实现了上述算法的 CPU 版本和 GPU 加速版本，通过实验对比，加速效果明显。

6.2 未来展望

现在的GPU技术可以说是日新月异，很多新技术正在不断出现，比如NVIDIA的 GPUDirect[4]技术。利用 GPUDirect，CPU 可以直接读写 CUDA 主存储器，消除不必要的系统存储器开销，GPUDirect 还支持 GPU 之间直接对等(P2P)DMA 传输，为创建 GPU 集群建立了非常好的条件。文本处理算法多种多样，其中大部分都可以采用 GPU 进行加速，比如现今火热的深度学习（Deep Learning）也可以和 GPU 完美结合，NVIDIA 已经发布了基于 CUDA 的深度学习框架，相信在

[4] https://cudazone.nvidia.cn/gpudirect/

不久的将来，GPU 必将大展拳脚。

回到本文讨论的问题，虽然有一定的成果，但还是有一些可以继续改进的地方的，比如：

1. 本文设计的算法均是针对的单机单 GPU 的情况，随着数据量的继续扩大，现有的系统和算法是不能够满足要求的，如何设计多个 GPU 组成集群甚至多 CPU 带多 GPU 系统协同工作，来满足数据两的持续扩张。
2. 本文涉及的近似重复文档检测为文档集合的离线计算，如果改进算法使得系统可以处理文档流的近似重复检测问题，处理流式数据是一个更加实用化的问题。
3. 本文的单词相似度计算针对的是英文的单词，能否扩展到其他的语种，因为维基百科的语种也是多种多样的。

参考文献

[1]Jiawei H, Kamber M. Data mining: concepts and techniques[J]. San Francisco, CA,itd: Morgan Kaufmann, 2001, 5.

[2]Kowalski G. Information retrieval systems: theory and implementation [M]. Kluwer Academic Publishers, 1997.

[3]Ding C, He X, Zha H, et al. Spectral min-max cut for graph partitioning and data clustering[C]. Proceedings of the First IEEE International Conference on Data Mining. 2001: 107-114.

[4]Han E H, Karypis G, Kumar V, et al. Hypergraph based clustering in high-dimensional data sets: A summary of results[J]. IEEE Data Eng. Bull., 1998, 21(1): 15-22

[5]Peter Kolb, DISCO: A Multilingual Database of Distributionally Similar Words [J]. Textressourcen und lexikalisches Wissen, Berlin 2008.

[6]Trevor Hastie and Rolbert Tibshirani: Discriminant Adaptive Nearest Neighbor Classification. [J] Manuscript received Jan. 13,1995, revised Ian. 30,1996. Recommended for acceptance by D.M. Titterington.

[7]S. S. Keerthi, S. K. Shevade: Improvements to Platt's SMO Algorithm for SVM Classifier Design. Neural Computation. March 1, 2001, Vol. 13, No. 3, Pages 637-649.

[8]Sergio Herrero-Lopez, John R. Williams, Abel Sanchez: Parallel multiclass classification using SVMs on GPUs. [J] GPGPU '10 Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units Pages 2-11.

[9]John O'Kane: Interpretable Boosted Naïve Bayes Classification. [J] KDD-98 Proceedings. 1998, Seattle, WA 98195-4322.

[10]Hepner, George F, Logan, Thomas, Ritter, Niles, Bryant, Nevin: Artificial neural network classification using a minimal training set - Comparison to conventional supervised classification. Photogrammetric Engineering and Remote Sensing

(ISSN 0099-1112); 56; 469-473. 1990.

[11] Jeffrey Dean, Sanjay Ghemawat: MapReduce: simplified data processing on large clusters. Communications of the ACM, Volume 51 Issue 1, January 2008.

[12] Shvachko, K, Hairong Kuang ; Radia, S. Chansler, R: The Hadoop Distributed File System. Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium. 2010, Page(s):1 – 10.

[13] Vincent Garcia, Eric Debreuve: Fast k Nearest Neighbor Search using GPU. [J] Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference.

[14] Fabian Nasse, Christian Thurau, Gernot A: Face Detection Using GPU-Based Convolutional Neural Networks. [J] Computer Analysis of Images and Patterns Lecture Notes in Computer Science Volume 5702, 2009, pp 83-90.

[15] Chin-Yew Lin, Eduard Hovy: Automatic evaluation of summaries using N-gram co-occurrence statistics. [J] NAACL '03 Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1,Pages 71-78, 2003.

[16] Landauer, T. K. and S. T. Dumais. A Solution to Plato's Problem: The Latent Semantic Analysis Theory of Acquisition, Induction and Representation of Knowledge. Psychological Review, 1997, 104(2):211–240.

[17] AMINUL ISLAM, DIANA INKPEN, Semantic Text Similarity Using Corpus-Based Word Similarity and String Similarity. [J] ACM Transactions on Knowledge Discovery from Data, July 2008, Vol. 2, No. 2, Article 10.

[18] Jay J. Jiang, David W. Conrath: Semantic Similarity Based on Corpus Statistics and Lexical Taxonomy. [J] In the Proceedings of ROCLING X, Taiwan, 1997.

[19] Pedersen, T., S. Patwardhan, and J. Michelizzi. WordNet::Similarity - Measuring the Relatedness of Concepts. In Proceedings of Fifth Annual Meeting of the North American Chapter of the Associationfor Computational Linguistics,2004, 38–41, Boston, MA.

[20] Long Chen, Wang Guoyin: An Efficient Piecewise Hashing Method for Computer Forensics. [J] Knowledge Discovery and Data Mining, 2008.

Page(s):635 – 638.

[21] Lin, D. Automatic Retrieval and Clustering of Similar Words. In Proceedings of COLING-ACL, 1998, Montreal.

[22] Lin, D. Extracting Collocations from Text Corpora. In Workshop on Computational Terminology, 1998, 57–63, Montreal.

[23] Weeds, J. and D. Weir. Co-occurrence Retrieval: A Flexible Framework for Lexical Distributional Similarity. Computational Linguistics 2005, 31(4):439–475.

[24] 魏诗云、杨家骏，网页近似重复检测算法研究. 计算机软件与应用，2012，1007-9599 08-0135-02.

[25] A. Broder, S. Glassman, M. Manasse, and G. Zweig.Syntactic Clustering of the Web. Computer Networks and ISDN Systems, 29(8-13):1157–1166, Sept. 1997.

[26] Gurmeet、Arvind and Anish. Detecting Near-Duplicates for Web Crawling. [J] WWW 2007, May 8–12, 2007, Banff, Alberta, Canada.

[27] 夏飞. 基于移动用户地理信息的音乐推荐研究.硕士学位论文.浙江大学 2013.

[28] Ke Chen, Gang Chen, Lidan Shou, Fei Xia: Pictune: situational music recommendation from geotagged pictures. SIGIR 2012:1011

[29] Turney, P. D. (2001). Mining the Web for Synonyms: PMI-IR versus LSA on TOEFL. In Proc. of the Twelth European Conference on Machine Learning, 491–502.

[30] A. Dasgupta, R. Kumar and T. Sarlós. Fast Locality-Sensitive Hashing. [J] KDD’11, August 21–24, 2011, San Diego, California, USA.

[31] Chowdhury A, Frieder O, Grossman DA, McCabe MC Collection statistics for fast duplicate document detection. [J] ACM Trans Inf Syst 20(2):171–191 2002.

[32] Identifying almost identical files using context triggered piecewise hashing [J]. DFRWS. Published by Elsevier Ltd, 2006.

[33] Carlos G. Figuerola, Raquel G´omez D´ıaz, Jos´e L. Alonso Berrocal. Web Document Duplicate Detection Using Fuzzy Hashing. [J] Trends in PAAMS, AISC 90, pp. 117–125, 2011.

攻读硕士学位期间主要的研究成果

致谢

时光荏苒，白驹过隙，三年的研究生生涯马上就要结束，在这即将毕业之际，我心里真的是感慨万千。三年前我刚踏入浙大的时候，心里想着一定要好好把握这来之不易的机会，不浪费一分一秒的时间，争取学到有用的专业知识为之后的人生道路做好铺垫。在实验室三年时间，我学到了很多，不只是专业科研知识，更有生活的道理。我非常感谢我的导师寿黎但教授、陈刚教授、陈珂副教授、胡天磊副教授、伍赛老师对我的科研工作上的指导，在我有困惑的时候在他们那里总是能够找到我想要的答案。

除了导师们，还有很多的师兄师姐，在这里感谢实验室的周显稞师兄、骆歆远师兄、汪戎师兄、王振华师兄、夏飞师兄、张超师兄、澎湃师兄、朱珠师姐、张冰冰师姐，在我三年的学习、科研和生活中给我的支持和帮助。感谢同窗刘博文、何平、胡乔楠、祁雅萍、周宇、吴逸、叶茂伟、姚雨程、王振杰、赵王军、罗妙辉、唐钦、李梦雯、邝昌浪、林秋霞、唐思、李环，跟你们在一起学生生活的三年是我人生中最快乐的时光，跟你们讨论学习生活中的问题让我成熟进步很多，因为有了你们的陪伴，让我三年的研究生生活变得愉快充实。还有实验室的师弟师妹们，特别感谢俞骋超师弟、李幸超师弟和庞志飞师弟，感谢你们曾经跟我一起学习研究。最后感谢实验室管理唐颖红老师和郑茜老师，正是因为你们的不辞辛劳，才有我们的这么好的学习工作环境。

即将毕业，希望老师们、师兄师姐们、跟我一起入学的同学们还有师弟师妹们都有一个美满的人生。

王俊俏

2015 年 4 月