**电 子 科 技 大 学**

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

**学士学位论文**

**BACHELOR THESIS**



论文题目 **C++ 环境的轻量级图数据库存储引擎设计与实现**

专 业 **软件工程（软件技术）**

学 号 **2014220201020**

作者姓名 **李陈扬**

指导教师 **段翰聪**

摘 要

随着Web技术的不断演进与发展，人类先后经历了以文档互联为主要特征的“Web 1.0”与数据互联为特征的“Web 2.0”时代，正在迈向基于知识互联的“Web 3.0”时代[1]。信息技术发展迅猛，数据呈爆炸性增长。传统的基于磁盘的关系型数据库比如 Oracle、MySQL 已经不能够满足日益增长的数据的大规模存储和关系查询需求。传统关系型数据库描述实体之间关系时，需要创建一个关联表以记录这些数据之间的关联关系，数据库需要通过关联表间接地维护实体间的关系，随着实体之间复杂关系数量和关系层次的增加，关联表的数量急剧上升，数据库的执行效能低下性能大大降低。而在图数据库的应用场景下，数据以自然语言方式存储，具有无索引邻接关系的原生图形存储结构可加快数据关系的事务处理速度。

本文基于图数据库的存储场景，研究了基于Java的Neo4j图数据库存储系统，调研其存储引擎的底层数据组织方式和查询模式，提出了一种轻量级的图数据库引擎设计方案。该方案实现了存储引擎的一系列模块：Store，Transaction，TxState，Command，Operations，Lock，FileChannel，ByteBuffer，Record和StoreEngine等。基于这些模块设计，实现了基于 linux 平台的轻量级图数据库存储引擎，最后实施性能验证，分析方案的效率。

**关键词：**图数据库，Neo4j，图形存储结构，无索引邻接

ABSTRACT

With the continuous evolution and development of Web technology, human beings have experienced the "Web 2.0" era characterized by "Web 1.0" and data interconnection, which are characterized by document interconnection, and are now moving towards the "Web 3.0" era based on knowledge interconnection. [1]. The rapid development of information technology and the explosive growth of data. Traditional disk-based relational databases such as Oracle and MySQL are no longer able to meet the growing demand for large-scale storage and relational queries of data. When a traditional relational database describes the relationship between entities, an association table needs to be created to record the association relationship between these data. The database needs to maintain the relationship between the entities indirectly through the association table. With the number of complex relationships between the entities and the relationship level With the increase, the number of related tables has risen sharply, and the performance of the database has been greatly reduced. In the application scenario of the graph database, the data is stored in natural language. The native graphics storage structure with no index adjacency relationship can speed up the transaction processing speed of the data relation.

Based on the storage scenario of graph database, this paper studies the Neo4j graph database storage system based on Java, investigates the underlying data organization and query mode of its storage engine, and proposes a lightweight graph database engine design scheme. The program implements a series of modules of the storage engine: Store, Transaction, TxState, Command, Operations, Lock, FileChannel, ByteBuffer, Record, and StoreEngine. Based on the design of these modules, a lightweight map database storage engine based on the linux platform was implemented. Finally, performance verification was performed and the efficiency of the solution was analyzed.

**Keywords:** graph database, Neo4j, graphical storage structure, no index adjacency

目 录

[摘 要 I](#_Toc484422302)

[ABSTRACT II](#_Toc484422303)

[目 录 III](#_Toc484422304)

[第一章 绪 论 1](#_Toc484422305)

[1.1 研究工作的背景与意义 1](#_Toc484422306)

[1.2 国内外研究历史与现状 1](#_Toc484422307)

[1.3 本文的主要贡献与创新 2](#_Toc484422308)

[1.4 本论文的结构安排 3](#_Toc484422309)

[第二章 相关技术理论基础 4](#_Toc484422310)

[2.1 数据库系统技术 4](#_Toc484422311)

[2.2 内存数据库系统技术 5](#_Toc484422312)

[2.2.1 内存数据库的定义 5](#_Toc484422313)

[2.2.2 内存数据库技术成果 6](#_Toc484422314)

[2.3 内存数据库 vs. 磁盘数据库 7](#_Toc484422315)

[2.3.1 内存 vs. 磁盘 7](#_Toc484422316)

[2.3.1 MMDB 与 DRDB 对比 8](#_Toc484422317)

[2.4 GPU通用计算技术 9](#_Toc484422318)

[2.4.1 GPU并行机制 10](#_Toc484422319)

[2.4.2 CUDA体系结构 13](#_Toc484422320)

[2.4.3 CUDA编程模型 15](#_Toc484422321)

[2.4.2 CUDA执行模式 16](#_Toc484422322)

[2.4.2 CUDA程序优化 17](#_Toc484422323)

[2.5 本章小结 19](#_Toc484422324)

[第三章 GPU计算引擎设计与实现 20](#_Toc484422325)

[3.1 需求分析及设计目标 20](#_Toc484422326)

[3.2 整体架构 20](#_Toc484422327)

[3.3 原语设计与实现 21](#_Toc484422328)

[3.3.1 Map 21](#_Toc484422329)

[3.3.2 Scatter和Gather 21](#_Toc484422330)

[3.3.3 Reduce 23](#_Toc484422331)

[3.3.4 Scan 24](#_Toc484422332)

[3.3.5 Filter 26](#_Toc484422333)

[3.4 原语组合成SQL算子 27](#_Toc484422334)

[3.5 关键技术和改进点 27](#_Toc484422335)

[3.5.1 大数据量处理 28](#_Toc484422336)

[3.5.2 全局内存访问延迟解决 28](#_Toc484422337)

[3.6 本章小结 28](#_Toc484422338)

[第四章 性能测试 29](#_Toc484422339)

[4.1 测试方案 29](#_Toc484422340)

[4.1.1 测试环境 29](#_Toc484422341)

[4.1.2 测试数据生成 30](#_Toc484422342)

[4.1.3 CPU上的实现方式 30](#_Toc484422343)

[4.1.4 正确性检验 30](#_Toc484422344)

[4.2 测试结果与分析 30](#_Toc484422345)

[4.3 本章小结 32](#_Toc484422346)

[第五章 全文总结与展望 33](#_Toc484422347)

[5.1 全文总结 33](#_Toc484422348)

[5.2 后续工作展望 33](#_Toc484422349)

[致 谢 34](#_Toc484422350)

[参考文献 35](#_Toc484422351)

[外文资料原文 36](#_Toc484422352)

[外文资料译文 37](#_Toc484422353)

第一章 绪 论

1.1 研究工作的背景与意义

随着用户、系统和传感器产生的数据量呈指数增长，其增长速度因大部分数据量集中在Amazon、Google和其他云服务的分布式系统上而进一步加快，数据内部依赖和复杂度的增加，这一问题因互联网、Web2.0、社交网络，以及对大量不同系统的数据源开放和标准化的访问而加剧，关系数据库产生了很多的不适应性。从而导致大量解决这些问题中某些特定方面的不同技术出现，它们可以与现有RDBMS相互配合或代替它们——亦被称为混合持久化（Polyglot Persistence）。数据库替代品并不是新鲜事物，它们已经以对象数据库（OODBMS）、层次数据库（如LDAP）等形式存在很长时间了。但是，过去几年间，出现了大量新项目，它们被统称为NoSQL数据库（NoSQL-databases）[2]。其中，图数据库（Graph Database）从最近十年的表现来看已经成为关注度最高，也是发展趋势最明显的数据库类型。

图形数据库每个对象是一个节点，之间的关系是一条边。相对于关系数据库来说，图形数据库善于处理大量复杂、互连接、低结构化的数据，这些数据变化迅速，需要频繁的查询——在关系数据库中，由于这些查询会导致大量的表连接，从而导致性能问题，而且在设计使用上也不方便。图形数据库适合用于社交网络，推荐系统等专注于构建关系图谱的系统。

本文基于以上背景，提出一种基于 C++ 的轻量级图形数据库存储引擎的设计方案。基于 Linux 平台对数据的图形结构存储方式进行设计实现。并对方案的效率进行了验证。

1.2 国内外研究历史与现状

在[众多不同的数据模型](http://en.wikipedia.org/wiki/Data_model)里，关系数据模型自80年代就处于统治地位，而且有不少实现，如[Oracle](http://www.oracle.com/index.html)、[MySQL](http://www.mysql.com/)和[MSSQL](http://www.microsoft.com/sqlserver/2008/en/us/)，它们也被称为关系数据库管理系统（RDBMS）。然而，最近随着关系数据库使用案例的不断增加，一些问题也暴露了出来，这主要是因为两个原因：[数据建模中的一些缺陷和问题](http://www.scribd.com/doc/2670985/SQL-Antipatterns)，以及在大数据量和多服务器之上进行水平伸缩的限制。

在应对这些趋势时，关系数据库产生了更多的问题。这导致大量解决这些问题某些特定方面的不同技术的出现，它们可以与现有RDBMS相互配合或代替它们，亦被称为[混合持久化（Polyglot Persistence）](http://codemonkeyism.com/nosql-polyglott-persistence/)。数据库替代品并不是新鲜事物，它们已经以[对象数据库（OODBMS）](http://en.wikipedia.org/wiki/List_of_object_database_management_systems)、层次数据库（如LDAP）等形式存在很长时间了。但是，过去几年间，出现了大量新项目，它们被统称为NOSQL数据库（NOSQL-databases）。图形便是关系规范化的一种替代技术。领域模型在数据结构上的方案，有两个主流学派：RDBMS采用的关系方法和图，即网络结构，如语义网用到的。

尽管图结构在理论上甚至可以用RDBMS规范化，但由于关系数据库的实现特点，对于像文件树这样的递归结构和像社交图这样的网络结构有严重的查询性能影响。网络关系上的每次操作都会导致RDBMS上的一次“联结”操作，以两个表的主键集合间的集合操作来实现 ，这种操作不仅缓慢并且无法随着这些表中元组数量的增加而伸缩。

在图的领域，并没有一套被广泛接受的术语，存在着很多不同类型的图模型。但是，后来有人致力于创建一种属性图形模型（Property Graph Model），以期统一大多数不同的图实现。按照该模型，属性图里信息的建模使用3种构造单元：节点（即顶点）、关系（即边）、具有方向和类型（标记和标向）、节点和关系上面的属性（即特性）。更特殊的是，这个模型是一个被标记和标向的属性多重图（multigraph）。被标记的图每条边都有一个标签，它被用来作为那条边的类型。有向图允许边有一个固定的方向，从末或源节点到首或目标节点。属性图允许每个节点和边有一组可变的属性列表，其中的属性是关联某个名字的值，简化了图形结构。多重图允许两个节点之间存在多条边。这意味着两个节点可以由不同边连接多次，即使两条边有相同的尾、头和标记。

图论的巨大用途被得到了认可，它跟不同领域的很多问题都有关联。最常用的图论算法包括各种类型的最短路径计算、测地线（Geodesic Path）、集中度测量（如PageRank、特征向量集中度、亲密度、关系度、HITS等）。然而，在很多情况下，这些算法的应用仅限制于研究，因为实际中没有任何可用于产品环境下的高性能图形数据库实现。幸运的是，近些年情况有所改观。有几个项目已经被开发出来，而且目标直指24/7的产品环境：Neo4j - 开源的Java属性图形模型、AllegroGraph - 闭源，RDF-QuadStore、Sones - 闭源，关注于.NET、Virtuoso - 闭源，关注于RDF、HyergraphDB - 开源的Java超图模型、Others like InfoGrid、Filament、FlockDB等。

目前有多种图形数据库系统，其中Neo4j是一个用Java实现、完全兼容ACID的图形数据库。数据以一种针对图形网络进行过优化的格式保存在磁盘上。Neo4j的内核是一种极快的图形引擎，具有数据库产品期望的所有特性，如恢复、两阶段提交、符合XA等。自2003年起，Neo4j就已经被作为24/7的产品使用。Neo4j既可作为无需任何管理开销的内嵌数据库使用；也可以作为单独的服务器使用，在这种使用场景下，它提供了广泛使用的REST接口，能够方便地集成到基于PHP、.NET和JavaScript的环境里。开发者可以通过Java-API直接与图形模型交互，这个API暴露了非常灵活的数据结构。至于像JRuby/Ruby、Scala、Python、Clojure等其他语言，社区也贡献了优秀的绑定库。Neo4j的典型数据特征：数据结构不是必须的，甚至可以完全没有，这可以简化模式变更和延迟数据迁移；可以方便建模常见的复杂领域数据集，如CMS里的访问控制可被建模成细粒度的访问控制表，类对象数据库的用例、TripleStores以及其他例子；典型使用的领域如语义网和RDF、LinkedData、GIS、基因分析、社交网络数据建模、深度推荐算法以及其他领域。

甚至“传统”RDBMS应用往往也会包含一些具有挑战性、非常适合用图来处理的数据集，如文件夹结构、产品配置、产品组装和分类、媒体元数据、金融领域的语义交易和欺诈检测等。

围绕内核，Neo4j提供了一组可选的组件。其中有支持通过元模型构造图形结构、SAIL，它是一种SparQL兼容的RDF TripleStore实现或一组公共图形算法的实现。要是你想将Neo4j作为单独的服务器运行，还可以找到REST包装器。这非常适合使用LAMP软件搭建的架构。通过memcached、e-tag和基于Apache的缓存和Web层，REST甚至简化了大规模读负荷的伸缩。

自适应规模的Neo4j无需任何额外的工作便可以处理包含数十亿节点、关系和属性的图。它的读性能可以很轻松地实现每毫秒（大约每秒1~2百万个遍历步骤）遍历2000关系，这完全是事务性的，每个线程都有热缓存。使用最短路径计算，Neo4j在处理包含数千个节点的小型图时，甚至比MySQL快1000倍，随着图规模的增加，差距也越来越大。

这其中的原因在于，在Neo4j里，图遍历执行的速度是常数，跟图的规模大小无关。不像在RDBMS里常见的联结操作那样，这里不涉及降低性能的集合操作。Neo4j以一种延迟风格遍历图：节点和关系只有在结果迭代器需要访问它们的时候才会被遍历并返回，对于大规模深度遍历而言，这极大地提高了性能。

如今，图数据库系统已经广泛应用于：社交网络、商品推荐、金融管理、欺诈识别等领域，这大大提高了实体间复杂关系的查找和分析效率。

1.3 本文的主要贡献与创新

本论文主要调研了在本文撰写之际近几年有关全内存数据库计算和GPU加速计算技术进展，了解了结构化数据查询性能存在的新挑战问题。基于全内存计算场景，研究了包括CPU+GPU异构的结构化数据计算优化方法，提出了一种基于GPU的等值、范围查询功能的计算引擎设计方案。方案实现了一系列运行于GPU上的原语：Map，Scatter，Gather，Reduce，Scan和Filter。基于这些原语设计并实现了SELECT，PROJECTION，GROUP和AGGREGATION等SQL算子。基于Linux平台对结构化数据GPU计算引擎方案进行实现，实施性能验证，并分析方案的效率。

1.4 本论文的结构安排

本文的章节结构安排如下：

第一章描述研究工作的背景与意义、国内外研究历史与现状和本文的主要贡献与创新点。

第二章对现有的相关技术理论进行描述分析和总结。

第三章对本文所述方案的架构与实现进行描述分析和总结。

第四章描述对本文所述方案的测试和分析。

第五章是本方案的总结与展望。

第二章 相关技术理论基础

本章对数据库系统中已有的并且和本文相关的技术进行了介绍，最后对现有的技术的优缺点进行分析总结。

2.1 数据库系统技术

数据库系统（Database Management System, DBMS）是存储在计算机内，有组织，可共享的数据集合。数据库管理系统是数据管理的最新技术，是计算机科学发展最快的领域之一，已经成为计算机信息系统的核心和研究的热点。

第一个商用数据库管理系统产生于20世纪60年代末。这些系统是由文件系统演变而来的，提供了对大量数据存储和高效查询的部分支持。但是，文件系统并不能保证数据不丢失，如果没有备份的话；而且它们也不支持对数据项的高效存取，如果不知道它在特定的文件里的存储位置的话。文件系统不直接支持第针对文件中数据的査询语言。它们对数据模式的支持仅限于创建文件的目录结构。持久性需求并不总被文件系统支持，你可能会丢失并没有备份的数据。最后，文件系统也不满足孤立性和原子性。虽然它们允许几个用户或进程并发地访问文件，但文件系统一般并不阻止两个用户同时修改同一个文件，从而导致一个用户的修改不能出现在文件中。早期的DBMS要求程序员直接面对数据的存储格式，并且它们不支持高级查询语言。例如，CODASYL查询语言具有允许用户通过数据元素间的指针从一个数据元素跳到另一个数据元素的语句。即使对于非常简单的查询，也需要相当大的工作量来写这样的程序。

数据模型是数据库系统的核心，根据数据模型数据库的发展可以分为三个阶段。

第一个阶段是以格式化系统统称的层次数据库和网状数据库。层次数据库用树形结构来表示各类实体和实体间的联系，对应于有根定向有序树，其代表系统是IMS(Information Management System)数据库系统；网状模型用有向图表示实体和实体间的联系，对应于有向图，其代表系统DBTG系统。这类系统用存取路径表示数据间的关系，数据库查询和数据操纵是导航式的过程化语言。

第二个阶段是使用关系模型的关系数据库系统。1970 年 IBM 公司 San Jose研究室的研究员 E.F.Codd [12]发表了“A Relational Model of Data for Large Shared Data Banks” 的论文，该论文提出了关系模型，其后又陆续发表了多篇关系数据库研究的论文，为关系数据库奠定了理论基础。以此为基础的典型系统有 System R 和 INGRES。80 年代攻克了关系数据库的一系列难题，涌现出很多优秀的关系数据库系统，如 DB2、Oracle、Sybase、Informix 等。

第三阶段是新一代数据库。关系数据库结构简单，操纵方便，但它只适合管理结构稳定、格式化和结构化的商业事务领域。对现代应用，如实时处理系统、计算机集成制造(CIM)、电话交换机、实时仿真、电信网管、电力调度，GIS，CAD/CAM 等具有很大的局限性。为了满足现代工程应用的需求，把数据库技术和其它的现代信息处理技术，如面向对象、人工智能、多媒体、实时处理、复杂建模等技术完美结合，产生新一代现代数据库已经成为必然。根据不同的应用环境和技术，现代数据库有：内存数据库(MMDB，Main Memory Database)、面向对象数据库(OODB，Object Oriented Database)、实时数据库(RTDB，Real Time Database)、主动数据库(ADB，Active Database)、多媒体数据库(MDB，Multimedia Database)、时态数据库(TDB，Temporal Database)、移动数据库(MDB)、空间数据库、知识库(KBMS，Knowledge Base Management System)等。其中面向对象的方法和技术对数据库的影响最为深远，数据库研究人员借鉴面向对象思想和技术，提出了面向对象的数据库(OODB，Object-Oriented Database)，以及在面向对象技术和在关系模型的基础之上提出了对象关系模型(ORDB，Object-Relational Database)。

2.2 内存数据库系统技术

内存数据库系统（Main Memory Database System, MMDB）的研究始于 20 世纪 80 年代中期，两大因素推动了它的研究和发展，一是一些现代应用对数据库的需求，需要数据库具有实时处理数据的能力，以满足时间限制，如雷达跟踪、证券交易等；工程应用领域如过程控制、CAD/CAM、CIMS、网络(如电信、电力、数据网等)；人工智能领域如自然语言处理、专家系统、知识处理、演绎推理或规则系统、机器人与机器人视觉、数据挖掘等等。另一个是容量大、价格低廉的内存。目前以 G 为量级的内存已经出现，“在十年之内，数据库将使用 T 为量级的内存缓冲区，几乎最大的数据库都可以驻留内存”。

由于两大因素的作用，数据库研究者考虑将数据库全部或部分驻留内存以获得高性能数据访问。通过将数据库的“工作版本”驻留内存可以极大地提高数据库的性能，如减少了磁盘 I/O、提高事务的执行速度、减少 CPU 缓存替换次数、提高系统的吞吐率。

2.2.1 内存数据库的定义

对数据库系统使用大内存，采用两种主要途径。一种是增大缓冲区，以至每一事务所需的大部分甚至全部数据都保存在缓冲池中。这种方法与磁盘数据库(DRDB，Disk Resident Database)无任何区别，只是缓冲区很大而己，数据的存取要通过缓冲区，最小化磁盘 I/O 仍然是算法设计的主要目标。

另一种途径是使用大内存作为数据库存储介质，数据库常驻内存，即数据库的内存版本是“主版本”或“工作版本”，而磁盘只做后备。这种方式 I/O 不再是系统瓶颈，其数据结构、查询处理、并发控制、恢复等都必须重设计，以高效地使用 CPU 和内存空间。

一般认为，内存数据库的定义不应涉及内存大小、I/O 次数、数据何时进入及怎样留驻内存等具体的实现技术，而只包含数据库永久留驻内存，事务的数据存取只涉及内存的意思。

就一般的理解，数据库的“工作版本”，即数据库运行期间所需的数据环境不应有 I/O，常驻内存是内存数据库充分必要的条件。内存数据库的定义如下：

设有数据库系统 DBS，DB 为 DBS 中的数据库，DBM(t)为在时刻 t，DB 在内存的数据集，DBM⊆DB。TS 为 DBS 中所有可能事务的集合，AT(t)为在时刻 t 处于活动状态的事务集，AT(t)⊆TS。(T)为事务 T 在 t 时刻所操作的数据集，(T)⊆DB。若任意时刻 t，均有：

∀TAT(t)，(T) ⊆ DBM(t)

成立，则称 DBS 为一个 MMDBS，DB 为一个内存数据库，记为 MMDB。

此定义说明 MMDB 运行的任何时刻，活动的数据集都包括在内存中，不涉及内存大小和磁盘 I/O 的次数问题。由于内存是易失性介质，还应把数据库在磁盘中保留备份，称为 SDB(Secondary Database，外存数据库版本)。

MMDBS 就是数据库的“工作版本”(当然也不排除整个数据库)常驻内存的DBS，显然它使用较大的内存，至少能存储当前事务“活动”的数据集，但并不要求任何时刻整个数据库都驻留内存，所以它还要用外存来存储内存放不下数据和用于恢复的备份，即 MMDB 还是要处理 I/O。

2.2.2 内存数据库技术成果

内存数据库自出现以来就得到数据库研究者的广泛关注，现在人们对它的体系结构、数据结构、存取方法、查询优化、事务处理、并发控制、数据恢复等技术取得了丰富的成果。

* 在体系结构方面，为了支持内存数据库的数据恢复，从软硬件出发，开发了多种结构形式，如多处理机结构、专用非易失内存存储活动日志等；
* 在数据组织方面，提出了基于关系模型的区-段式结构、影子内存等；
* 在存取方法上，研究多种适合内存数据库的索引及存取策略，如多种形式的 Hash，T 树索引等；
* 在事务和查询处理上，主要在事务提交和与之关联日志记录、查询优化，尤其是连接（join）优化，进行了多方面的研究，开发了“提前提交”策略；
* 在并发控制方面，提出了二层封锁方案、乐观并发控制等方法。由于内存速度较快，锁的粒度较大，这样可以减少死锁检测开销，提高吞吐率，降低系统的响应时间；
* 内存数据库安全性更为脆弱，恢复技术显得尤为重要。这方面的研究集中在检验点操作、日志记录、数据库装入和重装技术，如 Fuzzy 检验点策略、Black/White 策略、Copy on Update 策略等；有序装入、带优先级的装入、按存取频率的装入、以及优先级和频率组合等装入策略。

2.3 内存数据库 vs. 磁盘数据库

内存数据库系统的数据永久驻留在内存中。而常规的磁盘数据库系统（Disk Resident Database System, DRDB）的数据是驻留在磁盘中。因而，在访问数据时，内存数据库系统拥有比磁盘数据库系统更高的访问效率。随着内存价格不断走低且存储芯片的集成度越来越高，在内存中存储越来越大的数据库变得可行。这使得变得更加实用。内存数据库因为其快速的数据访问能力，使其能比磁盘数据库更适合于需要快速响应和高事务吞吐量的应用环境。对于那些需要在严格要求的时间内段完成事务请求的实时应用系统，和需要支持大数据量并发访问的高性能事务处理平台来讲，内存数据库都是一个理想的选择。正因为如此内存数据库技术越来越受到人们的关注。

2.3.1 内存 vs. 磁盘

计算机上的内存和磁盘在物理机制上有很大的不同,这些不同之处影响了数据库的设计、实现和性能。简要说来，内存和磁盘有如下一些不同之处：

1. 磁盘属于永久存储介质，在断电后磁盘中的数据将继续保持。而内存属于易失性存储介质，断电之后内存中的数据就将全部丢失。但目前，也有非易失性内存，它们在断电之后仍将保持原有的数据。非易失性内存的访问速度在内存和磁盘之间。
2. 磁盘为机械结构，每次访问数据时都需要有磁头寻道和定位的过程。这使得磁盘访问的代价很高。但是磁盘的访问代价又相对固定，与访问的数据量无关。即便是访问一块数据中的若干数据，也会将整块数据读取一页数据，因此磁盘是块设备。而内存不是块设备，它的最小数据访问单位是字节。
3. 由于磁盘的物理机械方面的原因，使得磁盘的访问速度比内存的访问速度漫数十倍。
4. 磁盘中的数据存储方式与内存中的数据存放方式有着显著的不同。磁盘是块设备，访问磁盘中连续存储的数据的效率明显高于访问随机存储的数据，这需要所访问的数据要尽可能的连续才能获得较高的访问性能。而在内存中，对被访问数据的连续存放没有特别的要求。总体上讲，访问内存中连续存放的数据和随机存放的数据,在性能上只有少量的差别。考虑到处理器缓存的利用和缓存的利用，访问内存中连续存放的数据具有一定的性能优势。
5. 内存中的数据能被处理器直接访问，而磁盘中的数据不能被处理器直接访问。因此,当软件出错时，软件可能会任意破坏内存中的数据，却并不能直接破坏磁盘中的数据。

可以说,磁盘和内存的这些差异，影响了数据库管理系统从并发控制、数据存储到数据访问、程序访问的各个方面。

2.3.1 MMDB 与 DRDB 对比

MMDB 与 DRDB 之间主要区别在于 MMDB 的主数据库常驻内存，体系结构设计的优化目标是提高内存和 CPU 使用效率。与 DRDB 相比，MMDB 的优点如下：

1. 完成同样的功能，所需机器指令大大降低；
2. 事务处理无需 I/O，极大提高了系统性能；
3. 不再需要缓冲区管理器，消除了磁盘和内存之间数据拷贝开销；
4. 在数据组织与管理中，广泛使用指针，简化了内存管理，降低了空间开销。

内存数据库的工作版本存放在内存中，可以直接的存取数据，因而不需要对缓冲区进行管理，这决定了其体系结构的差异。

一般的磁盘数据库和内存数据库结构分别如图2-1和图2-3。

图2-1 磁盘数据库结构

图2-2 内存数据库结构

2.4 GPU通用计算技术

计算机图形处理器(Graphics Processing Unit，GPU)被定义为“一个单芯片的处理器，集成了几何变换、光照、三角形构造、裁剪和绘制引擎等功能，并具有每秒至少 1 千万个多边形的处理能力”。GPU 极大提升了计算机图形处理的速度、增强了图形的质量，并促进了与计算机图形相关其他应用领域的快速发展。与中央处理器(Central Processing Unit，CPU)的串行设计模式不同，GPU 为图形处理设计，具有天然的并行特性。

2.4.1 GPU并行机制

自中央处理器（CPU）诞生以来，主频一直不断地在提高，运算能力也越来越强，并于 2002 年达到了 3GHz。然而，随着频率的提升，也带来处理器的功耗、晶体管的散热、访存延迟和设计复杂度等工艺制造问题，单纯依靠频率已经无法继续提升处理器的性能。为此，工程师们设计出了多核 CPU，即在一个处理器中集成两个或多个完整的计算引擎(内核)。多核技术不仅给用户带来更强大的计算性能，并且还可在多任务的计算环境用进行多任务处理。

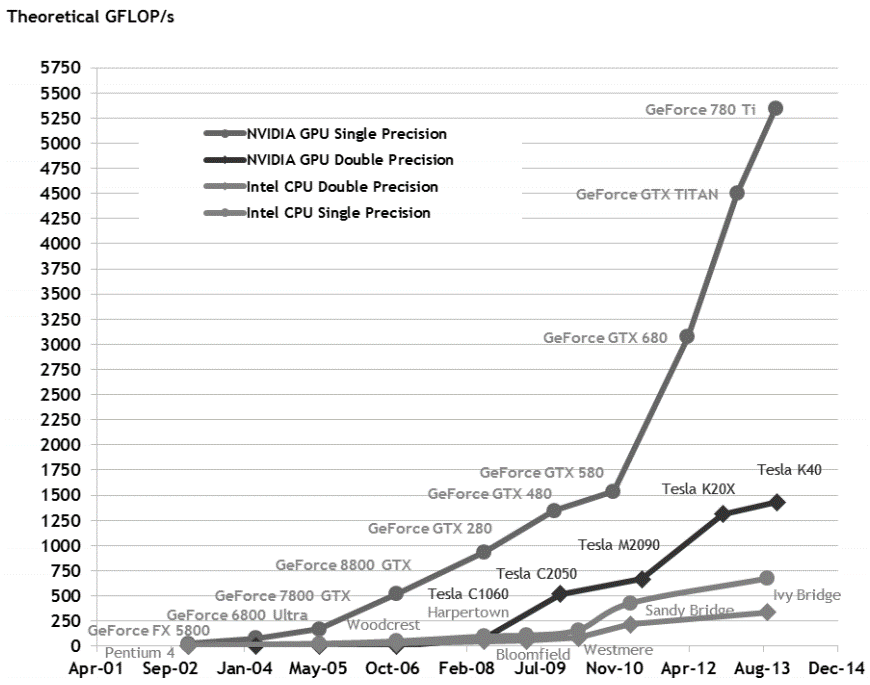
GPU 是个人计算机、工作站、游戏机等硬件设备普遍采用的图形图像加速处理器，具有 SIMD 并行处理能力。近年来，随着 GPU 硬件技术的发展，可编程的 GPU已逐渐发展成为一种众核、多线程和高度并行化处理器，并可处理传统图形领域之外的通用计算领域。目前，GPU 正以每12个月提高1倍性能的速度向前发展，远远超过了经典的摩尔定律；而同期的 CPU 由于核心体系架构没有新的突破，因此受传统工艺的限制已经减缓了发展的步伐。以下是当前主流CPU 与NVIDIA 的GPU的浮点处理能力的对比图。

图2-4 CPU与GPU单浮点计算能力对比图

从图2-4可以看出，无论从处理器的发展速度或者当前处理器的性能上，GPU 的浮点计算能力已经远超过CPU的计算能力。GPU和CPU的浮点计算能力差异的原因是：GPU是特别为计算密集，高并行度计算（如同图像渲染）设计的，因此将更多的晶体管用于数据处理而不是数据缓存和流控。

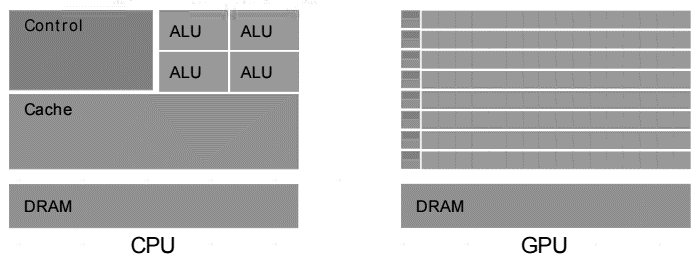
GPU 独特的硬件架构设计，决定了它的计算性能上比 CPU 具有明显的优势。最初，GPU 是专门为计算密集型和高度并行化的的图形图像处理（如颜色渲染和顶点变换等）而诞生，它大量的晶体管被用于算术逻辑单元(Arithmetic Logic Unit, ALU)，即运算的流处理器，因此不需要太多的控制单元和缓存。传统的 CPU 设计目标是处理逻辑性较复杂的事务计算，它把更多的执行时间用于控制指令的流程而不是指令的运算上，使执行单元能够以最低的延迟获得数据和指令，通过提高分支预测的准确度和缓存的命中率，提高指令执行的效率；因此在CPU的芯片内部，大多数的晶体管用于分支预测、数据缓存、指令流控制等硬件的实现。由此可见，在相同的有限芯片空间上，GPU比CPU 有更多的计算资源，因此它的计算能力比 CPU 更强大，性能更容易扩展。下图是CPU 与 GPU 中晶体管的数量及用途比较图。

图2-5 CPU与GPU结构对比图

由图2-5可见，由于逻辑控制和缓存的晶体管占据了 CPU 的芯片大量空间，限制了其上集成的内核数量，因此目前主流的 CPU 内核数不超过8个，并且每个核执行的流水线也不超过6个；而当前 NVIDIA 的 基于 Fermi 的 GPU 架构中，可有多达 16 个以上的流处理器(Stream Multiprocessor，SM)，每个流处理器有可以有多达 32 个标量流处理器(Stream Processor，SP)，即常常被人称为“CUDA 核”，则并行计算的总核数可达到 512 个。所以，尽管 GPU 的主频低于 CPU，由于它有大量的可并行计算的 CUDA核，使其在单精度或双精度的浮点运算和并行处理能力上比 CPU 的性能更优。

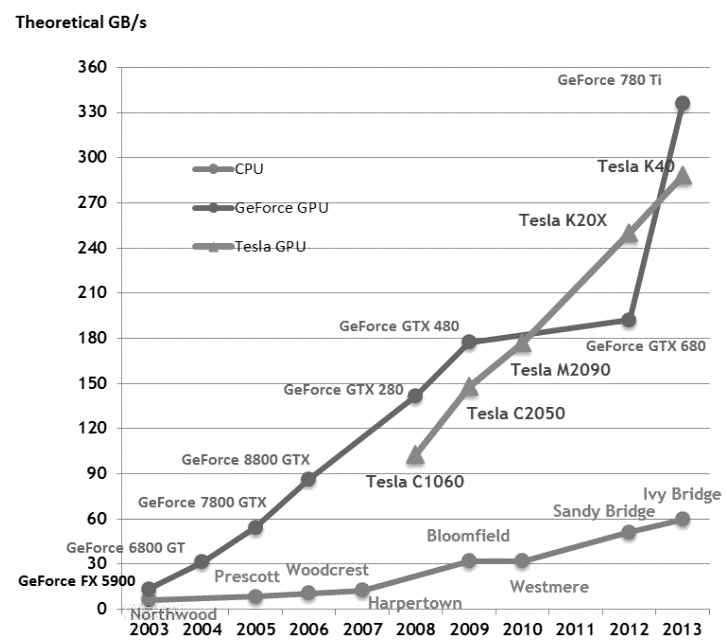
在存储带宽上，如下图2-6所示，GPU的显存带宽比CPU的内存带宽高很多。

图2-6 CPU与GPU存储带宽对比图

由图2-6可见，如今GPU的显存带宽已经远大于CPU的存储带宽，并且GPU的存储带宽发展速度比CPU更快，造成二者如此大差异的原因主要有以下两个方面：

一方面，尽管从芯片技术的角度上看，GPU和CPU使用的存储器颗粒的材料和制造原理相同，但内存使用的DDR为了硬件的灵活性和可扩展性而通过DIMM插槽与主板相连，显存中使用的 GDDR 则直接固化在PCB板上，这种方式能很好地解决数据信号的完整性问题，并且显存的工作频率通常也比内存高。

另一方面，当前的CPU存储器主要是基于双通道或三通道技术，每个通道的位宽是64位；而在GPU中则同时并存着多个存储器控制单元。例如英伟达GTX 480 中共有6个存储控制器，每个控制位宽64位的显存芯片，则总的存储位宽为384位；如果GDDR5的频率1.848GHZ，则GTX480的带宽是： 6\*64\*1.848\*2/8 = 177.408 GB/s（传统的 DDR\GDDR 接口的速率为每个时钟传输2个数据）。由此可见，尽管传统内存的DDR有较大的存储空间，但实际的访存带宽反而比GPU低。

通常CPU的每个处理核在某个时刻只能运行一个线程，并且需要调用操作系统底层的API实现多线程的调度机制；当进行线程切换时，操作系统首先采用栈来保存当前线程的上下文，然后再装载另一个线程的上下文，为此要付出高达数百个时钟周期的代价，因此它属于软件粗粒度的多线程方式。在线程切换过程中，还会导致缓存的缺失，由此还需要进行复杂的逻辑控制操作来保证数据的一致性。目前GPU采用的是由硬件管理的轻量级线程，它能够管理比实际计算单元数更多的线程数并且线程切换的开销几乎为零；通过执行大量的线程并高效地进行线程切换，能够有效地隐藏访存延迟并使计算单元的满负荷运行,从而提升并行处理能力。

综上所述，CPU适合解决具有复杂的控制逻辑和能够适应各种情况的串行计算问题，特别是复杂的逻辑运算。GPU的设计目标是针对面向吞吐量的数据并行计算问题，适合执行逻辑分支简单、计算密集度高且可采用大量线程并行机制的大规模数据任务。

2.4.2 CUDA体系结构

自在当今众多的 GPU生产厂商中，能引导市场潮流的无疑是AMD/ATI、NVIDIA和Intel三家公司，他们不仅都向市场推出了各自产品，还提出相应的编程模式。AMD 公司的 R700 芯片主要由数据并行处理器阵列、命令处理器、存储控制器和超线程管理调度器等部分组成，它通过SIMD 流水线调度数以百万计的线程并发执行，并以重叠访存和计算的方式隐藏访存的延迟。为解决CPU和 GPU间数据传输的瓶颈问题，AMD推出了新一代的产品APU，它将多核 X86 CPU芯片、GPU处理器核心和高速总线集成到同一芯片上，可实现不同处理核心间的数据高速传输。Intel 公司的Larrabee体系结构的基本计算部件为顺序CPU核，它基于传统的 x86 指令集结构设计，面向高端的通用计算平台，并把 L2 Cache 设计成分布式且每个核心对应其中一块, 从而提高了缓存的命中率, 有利于数据的同步和共享。NVIDIA公司于2006发布基于统一架构的G80，随后又推出相应的CUDA编程平台。由于它采用扩展的C语言作为基本编程语言，并封装了图形编程的底层转化算法，不仅降低了GPU的编程难度，同时还提供灵活的并发线程管理机制，因此成为深得众多GPGPU领域研究者的喜爱。本文是基于NVIDIA 的CUDA编程模型进行内存数据库索引技术的研究工作，以下简单介绍NVIDIA 公司在 GPU 产品市场的发展历史。

1999年，NVIDIA首次向个人 3D 电脑市场推出具有标志意义的图形处理器Ge Force 256，受游戏行业和图像处理需求的驱动，GPU 开始进入高速发展时期；

2001年，NVIDIA 先后推出 Ge Force3和 Ge Force4 Ti产品，将图形硬件的流水线解释为流处理器，允许应用程序对顶点数据的操作，GPU首次具有可编程性，由此开启了GPGPU的时代；

2003年，NVIDIA 新的Ge Force FX架构的象素级和顶点级操作的可编程性得到进一步地扩展，不仅提供对浮点格式的纹理的支持，还可在纹理上实现索引的查找；

2004年，NVIDIA Ge Force6800集成2亿多的晶体管，具有16条超标量的管线架构，使其功能更强大、更灵活，顶点程序不仅可以直接访问纹理，还支持动态分支；

2006年，NVIDIA 的 Ge Force 7800 内建了Cine FX 4.0引擎，它将许多突破性绘图技术融入顶点着色器、像素着色器、以及材质引擎的核心，极大地提高了3D的绘图效能与视觉质量。同年，NVIDIA 还发布第一款基于统一架构 G80的Ge Force 8800 产品，它的核心拥有空前规模的 6.81 亿晶体管，并有 12 颗高频率的显存，使 8800GTX拥有超强的性能,将 GPGPU 推向一个更广阔的应用领域。

此后 NVIDIA 在G80 体系结构的基础上，分别提出了 GT200 和 Fermi 两种新的体系结构，使GPU的性能得到更进一步的提升。以下将对 NVIDIA 公司的三种统一架构GPU体系结构进行阐述。

NVIDIA 的第一代统一架构 G80[17]的体系结构主要由存储系统和流处理器阵列（Stream Processor Array，SPA）两部分组成，并且通过一个片上的互连网络连接。G80 采用一种灵活的扩展机制，可使存储系统和流处理器根据客户的需求进行独立扩展。在此结构中，SM 相当于一个 SIMD 处理单元，它是G80的基本计算单元并具有独立的硬件结构，SM拥有独立的完整前端，包括取指、译码、发射和执行单元等，每个SM中包含 8个标量处理器SP和两个特定函 数 计 算 单元（Special Function Unit ，SFU）；SM上还包含16K的共享存储器和8192个32位寄存器，供执行线程的使用。此外，每两个SM 组成一个线程处理器群（Thread Processing Cluster，TPC）。在 G80的体系架构，负责执行任务的是128个SP，最大可并行的线程数达到4096，内存带宽达到86 GB/s。

NVIDIA于2010年推出的基于Fermi架构的GTX480，它是采用一种全新的 GPU硬件架构设计，堪称GPGPU革命的一次质的飞跃。Fermi架构采用更新的制造工艺技术，片上可集成数十亿个晶体管，Fermi的主要新特征包括：拥有16个 SM 且每个SM 可包含32个CUDA 核，总数可达512个，每个SM的寄存器数增至 32K；引入数据存储的 ECC 保护功能和 64-bit 寻址空间，最大可支持1TB 的显存；双精度浮点计算性能达到768 GFLOPS；编程语言完全支持C++；支持全局存储器的 L2 缓存机制，容量达到768K。

2.4.3 CUDA编程模型

CUDA 编程模型在形式上类似单程序多数据（SPMD）软件模型，而GPU 被认为是一个执行并行计算的核函数(Kernel function)的众核处理器。CUDA提供三种关键的抽象概念：线程组的层次结构，共享的内存和障栅同步（barrier synchro- -nization）。CUDA 的线程组分为3个层次，其中网格（grid）是执行CUDA核函数的线程块的集合，每个网格含有多个线程块；每个线程块由数百个线程组成，在同一个线程块内的线程可以使用共享的存储器共享数据，并可在一个障栅实现同步，同一个线程块内的所有的线程在一个多线程架构中可并发执行。CUDA程序语言中的线程比传统操作系统的线程更轻量级，CUDA的每个线程通常每次处理一个数据元素。 在执行 CUDA 核函数时，程序员需要指定每个线程块的线程数和每个网格中的线程块的数量；CUDA程序模型提供2个共享的可读写内存空间：共享存储器和全局存储器。对于线程块而言，共享存储器属于局部的内存，而全局存储器则可被所有的线程块访问。此外，CUDA还提供两个只读的存储器：常数存储器和纹理存储器，这两个存储器驻留在外部 DRAM中，需要通过只读缓存才能访问。

GPU 是一个单指令多数据（SIMD）的框架，它包括一组数量可伸缩的流多处理器（SM），且每个SM 有多个可共享控制逻辑和指令缓存的流处理单元（SP）；此外，SM 还有一个多线程指令的提取和发布单元以及一个只读常量缓存和一个 16KB 可读写共享内存。每一代 GPU 所含的 SM 数量和每个 SM 的 SP 数量是不同的。目前 GPU 的图形存储器（GDDR）即全局存储器的容量可高达 4GB，这些 GDDR 存储器与 CPU 主板上的系统内存有所不同，它们是图形的帧缓冲存储器。对于图形应用，它们可为 3D 渲染保存视频图像和纹理信息，如果没有通用计算的功能它们就是一个高带宽的片外存储器，甚至比传统的系统内存延迟大。全局存储器（GM）是一个可随机访问、可读写并可被线程块所有线程访问的片外全局存储器，由于没有高速缓存，GM是GPU设备里访问速度最慢的存储器，通常需要耗费数百个时钟周期；在每个 SM里有一个片上的可读写的共享存储器（Sh M），它可被SM 上运行的所有线程访问，并且访问的速度与寄存器的访问速度一样快，但是 Sh M 的容量比GM小很多。

GM通过512-bit的数据带宽连接到GPU 设备，凭此巨大的总线带宽，GPU只需一个时钟周期就可从全局内存中提取 16个连续的32位（4字节）字，GPU的片上内存资源还包括寄存器文件（每个SM有 16K 或更多）。为了隐藏片外存储器的访问延迟，GPU并发地执行大量的线程，这些线程被组成多个线程块，并根据每个SM的资源被动态调度到 SM中执行。这些线程遵循单程序多数据 SPMD 的程序执行模式：在一个线程块内，32个编号连续的线程集成到一个指令组称之为warp ；SM执行一个warp指令时，把指令发布到32个线程的方式类似于在 X86 系统执行单指令行多数据（SIMD），然而与执行 SIMD 指令不同的是：warp的概念对于编程者是透明的，程序员只需为每个线程编写相应代码，然后指定每个线程块中并行的线程数和Kernel函数网格里的线程块数即可。

2.4.2 CUDA执行模式

CUDA 程序代码是主机和设备代码的结合，它包括执行在主机（CPU）或设备（GPU）上的一个或者多个阶段，其中大规模量的数据并行阶段在设备代码中执行；设备代码使用扩展 ANSIC的一些关键字来标记数据并行的函数（Kernel 函数）以及和它们相关的数据结构。

当调用设备上的 Kernel函数时，每个 Kernel 将启动一组由程序员编写的含有大量线程的线程块。此时，线程块调度机制将根据每个 SM 的可用资源把线程块动态地调度到 SM 中执行。

在GPU里，一个线程块里所有的线程被执行在同一个SM里，并且每个SM可能并发执行的线程块数由线程块中的warp数，寄存器数和共享存储器的使用量所决定。SM的有限资源如 warp调度、寄存器和共享存储器将被分配到被每个 SM 上调度执行的线程块中。在SM上某一时刻运行的线程块称为活动线程块，由于一个线程块通常由数个warp (warp 的数量等于块内的线程数/32) 组成，因此每个 SM总的活动 warp 数量等于每个块中的warp数\*当前的活动线程块数。warp是 SM 处理器执行的基本单元，它以分时的模式调度多个warp。SM有足够的功能部件和寄存器端口同时执行32个线程，对于一个只有8个功能部件的SM，一个SM 最多可同时处理8个线程块，执行32个线程则需要耗费4 个SM处理器时钟才能完成指令计算。对于Fermi架构的GPU，SM能并发执行32个warp。

当线程块在SM执行时，SM 管理每个线程的id并且同一个块内的线程使用 Sh M可在SM 内进行相互通信，但不同线程块内的线程不能相互交流。由于线程块被动态地调度到不同的SM执行，因此即使线程利用GM也不能相互通信。在具有高维度的Kernel 中，warp可根据行号从线程块中提取。由于warp以SIMD的方式，如果SM发现warp内有一个或多个线程执行类似访问GM的较长延迟的数据传输操作，则它将发起一个 DMA操作，挂起该warp的执行并保存它的上下文，然后调度那些已准备好数据的warp到SP中执行，这种通过调用其他warp的执行来掩盖代价极高的延迟操作的机制，称为延迟隐藏。

GPU把GM按32，64或128个字节划分成多个段并依此对齐，因此通过一个内存事务处理就可访问一个段内的所有数据。假如段的长度为128个字节、数据总线带宽为 512-bit，则编译器分发一个装载或存储指令到16个线程（半个 warp）就可获取 16个连续的数据，从而减少GM的事务处理数。因此，理想的情况是一个warp内所有线程的内存访问可以合并到一个内存事务处理中完成，这与warp访问内存的方式相关。如果访问的是连续内存地址，则warp内所有的内存请求可合并到一个内存事务中；否则，每个内存地址将产生一个不同的分支。如果一个 warp的内存请求是非合并访问的方式，则只有warp内所有的内存事务处理完成后，warp才能继续执行，这也意味着非合并访存比合并访存需要更多的时间等待。

综上所述，用全局存储器的合并访问可显著地提高内存数据的传输性能，并且通过 warp 调度机制可使 SP 始终处于繁忙的工作状态。此时，并行程序的运算时间将不考虑程序与 GM 间的数据传输代价。

2.4.2 CUDA程序优化

为尽可能提高GPGPU程序的加速效率，编写程序的过程中应考虑并行粒度划分、指令吞吐量、存储器带宽等方面的性能影响因素。通常执行一个GPGPU程序需要经历：预处理、主机到设备端的数据传输、数据处理、把设备端运算的结果传回主机和后期处理等共5个阶段，在对程序进行优化之前，必须先了解程序执行时每个阶段的操作内容，从而可对其采用相应的优化措施，以下将详述各个阶段操作的细节。

1. 预处理阶段（）

程序在这个阶段，首先对 GPU 硬件进行相应的初始化工作（），通过调用 Runtime API中的cudaSetDevice()函数设置程序需要使用的GPU卡，假如当前系统只有一块网卡，此步骤可以略去；其次，根据问题的规模为主机端和设备端分配相应的内存（），由于目前 NVIDIA 的图形显卡大多不支持动态分配显存，因此需要在主机端调用 cudaMalloc()函数分配问题所需的显存空间；第三，在 CPU 端对待解决问题的应用数据格式做相应的预处理（）。

1. 原始数据传输阶段（）

由于GPU不能直接读取硬盘的数据，因此需要通过调用 cudaMemcpy()函数，把待解决问题的数据从主机端的内存传送到GPU的全局存储器。

1. 处理阶段（）

在此阶段，首先把数据从 GPU 的全局内存复制到 GPU 的共享存储器中()，然后执行“核函数”的处理（），最后把运算结果从共享存储器复制回全局存储器（）。

1. 运算结果传输阶段（）

当 GPU 的核函数对数据进行并行处理结束后，由于在 GPU 上不支持运算结果的直接打印输出，为此需要继续调用 cudaMemcpy() 函数把运算结果传回主机端的内存。

1. 后续处理阶段（）

在执行程序的最后阶段操作内容包括对运算结果进行相应的格式处理、释放之前分配的主机端的内存空间和设备端的显存空间等。

对于一个数据并行程序，通常还需多次执行“核函数”直到所有的输入数据全处理完毕。在整个处理过程中，部分数据还会从GPU的全局存储器复制到每个 SM的共享内存，然后由“核函数”执行，最后把结果复制回全局存储器。由此，整个GPGPU的程序运行时间为：

=

其中，DataS是应用问题的数据规模，SMsize是每个SM可用的共享内存。

由上式程序员可根据实际问题对GPGPU程序分阶段地进行评估，找出影响程序性能的瓶颈，从而对其进行相应的优化工作。

在执行GPGPU程序的预处理阶段，首先根据待处理任务的计算特征按执行过程划分为主机端和设备端部分，尽可能减少CPU的串行运算时间，而把大规模的数据并行计算任务交给GPU执行；其次对输入和输出的原始数据格式进行相应处理，尽可能满足GPU合并访存的要求；接着要考虑执行Kernel函数的配置优化问题，GPU的线程是按“grid ->block->thread”三个层次结构来组织，为提高SM的占用率，必须仔细设计线程块的维度和尺寸。由于一个SM内可同时执行多个不同block的warp，并且warp的指令吞吐量和延迟各不相同，为使SM中的一个 warp的延迟能够被 block 中其他warp的执行完全隐藏，在一个SM中执行的线程数至少为192个（2.0以下的架构）或384个。对于较大的block则有更多线程能够利用共享存储器通信获得更高效率，而较小的block则获得更多的活动线程块，并且也能减少因block内线程同步导致的性能损失。通常情况，block内的线程数量设置为64的整数倍。

使用CUDA编程时，影响程序性能的瓶颈主要是内存带宽和延迟，为此，GPU提供一些具有不同特性和性能的存储器，用于提高CUDA程序的性能。例如，程序员可在编程时使用ShM以减少频繁到GM获取数据的访存延迟等。常用的优化策略包括：

1. 由于访问片外的显存延迟高，而片上存储器带宽远高于片外的显存带宽，因此设法使用访问Sh M和寄存器以取代对GM的访问；
2. 对于Fermi架构的GPU，根据数据的局部性，选择支持缓存的显存访问方式，提高缓存的命中率；
3. 尽量使用内建的数据类型，并对数据类型使用\_\_align 进行对齐，对存储器的访问对齐到分段的起始位置；
4. 尽量避免共享存储器出现bank conflict的现象；
5. 在进行主机-设备端的数据传输时，可通过mapped memory, portable memory, write-combined memory 和 zero-copy 提高数据的传输速率；
6. 使用移位指令代替除法和求余可以获得提高指令的吞吐量。

帮助程序员分析CUDA程序的性能，NVIDIA在CUDA toolkit中提供了一个基于图形界面的分析工具：Computer Visual profiler，它通过在芯片上的硬件计数器跟踪每个事件，以图形化的形式表示每个计数器的度量值，提供基于profiling 数据的性能分析，并提示提高程序性能的关键因素。Computer Visual Profiler提供的分析图表包括：GPU时间摘要图（GPU time summary plot），设备级摘要图(Device level summary plot)，会话级摘要图（Session level summary plot）和 CUDA API追踪图等等。此外，它还为CUDA应用程序提供基于上下文级、内核级、会话级和设备级等的性能分析。

2.5 本章小结

本章首先从介绍数据库技术入手引入内存数据库技术，从而展现当前利用GPU进行全内存数据库的底层查询的必要性和优缺点。接下来介绍了本文所述方案所用到的GPU通用计算技术，并分析如何进行优化。

第三章 GPU计算引擎设计与实现

3.1 需求分析及设计目标

随着互联网的发展，越来越多的数据出现在了人们的生活当中。而数据中的价值也开始被人们越来越广泛的重视。传统的关系数据库比如 Oracle、MySQL 已经不能够满足日益增长的数据的存储和查询功能。无论是政府还是公司，都需要能够快速对业务进行响应，从而在最短的时间内拿出决策方案。数据库在整个决策过程中扮演着重要的角色。

本设计方案应用在对查询响应速度要求特别高的场景下，提出了一种基于GPU 的等值、范围查询功能的计算引擎设计方案。方案实现了一系列运行于GPU上的原语：Map，Scatter，Gather，Reduce，Scan和Filter。基于这些原语设计并实现了SELECT，PROJECTION，GROUP和AGGREGATION等SQL算子。

3.2 整体架构

数据库在经过此法解析和语法解析后生成一系列逻辑计划，再生成一个物理计划DAG图，而这些这个DAG图就是一个的节点就是一些基本的算子，算子的结构组成如图 3-1所示。自底向上分别是存储层，计算原语和算子层。

作为一个GPU计算引擎，其实现的主要就是最底层的这些原语，另外，本文也对这些原语如何组成基本的数据库算子做了描述及实现，并测试了相应的性能。



图 3-1 数据库执行引擎架构

存储层即底层数据，本方案为方便比较，源数据直接由CPU随机生成供上层原语进行计算。关于底层原语如何组合成为SQL算子见下一节介绍原语时对于其如何组合成SQL算子的说明。

3.3 原语设计与实现

本节主要介绍原语的具体定义与实现方式，以及这些原语如何对应实现SQL算子。

3.3.1 Map

Map原语即映射操作，其定义与Lisp中有关映射函数的定义相同：

原语：Map

输入：，映射函数

输出：

函数：，

给定一组数据元组和一个函数，Map对每个元组执行该函数对应的操作。本文实现的Map采用GPU上多线程扫描处理的方式，每个线程负责关系元组中的一部分数据进行处理，从而达到并行。

具体CUDA内核代码如下：

\_\_global\_\_ void map\_kernel(Tuple \*d\_data\_in, int len, int delta,

int \*d\_data\_out)

{

int idx = blockIdx.x \* blockDim.x + threadIdx.x;

for(; idx < len; idx += delta)

d\_data\_out[idx] = func(d\_data\_in[idx].y);

}

其中Tuple为输入元组的格式。每个线程通过线程块的ID和线程块的维度以及自身的块内ID计算出自己的全局ID，如上所示。为实现当数据量大于线程总数时仍可以支持Map原语，设置一个delta值，即线程总数，线程id不断叠加delta来访问相应目标数据，从而访问到所有的输入数据。

3.3.2 Scatter和Gather

Scatter即散列操作，Gather即聚集操作。这两个原语主要用来实现Projection和Join算子。Scatter负责对关系元组的顺序读取和随机写入，相应的，Gather负责对关系元组的顺序写入和随机读取。当指定的访问位置是顺序的，Scatter和Gather执行的操作与Map相同。具体执行方式和Map类似，也是每个线程负责关系元组中的一段数据进行处理。

Scatter的定义如下：

原语：Scatter

输入：，

输出：

函数：，

Scatter的CUDA内核执行代码如下：

\_\_global\_\_ void scatter\_kernel(Tuple \*d\_data\_in, int len,

int \*d\_location, int delta, int \*d\_data\_out)

{ int idx = blockIdx.x \* blockDim.x + threadIdx.x;

for(; idx < len; idx += delta)

d\_data\_out[d\_location[idx]] = d\_data\_in[idx].y;

}

Gather的定义如下：

原语：Gather

输入：，

输出：

函数：，

Gather的CUDA内核执行代码如下：

\_\_global\_\_ void gather \_kernel(Tuple \*d\_data\_in, int len,

int \*d\_location, int delta, int \*d\_data\_out)

{ int idx = blockIdx.x \* blockDim.x + threadIdx.x;

for(; idx < len; idx += delta)

d\_data\_out[idx] = d\_data\_in[d\_location[idx]].y;

}

从比较可以发现，两者在读取访问和写入访问时刚好相反，Scatter负责顺序读取、随机写入，而Gather负责随机读取，顺序写入。

3.3.3 Reduce

Reduce原语根据输入关系计算输出一个值，这个值可以是SUM、MAX和MIN等。Reduce原语主要实现Group和Aggregation算子。

原语：Reduce

输入：

输出：

函数：

其具体实现方式如下图3-2所示，以求SUM为例，先分别每个块计算出块内的SUM结果，之后将这些结果汇总到一个块内进行计算，得到最终的SUM结果。根据输入数据的数量级可动态增加或减少递归的层数。

图3-2 Reduce流程图

块内SUM的计算方式如下图3-2所示，每个线程处理两个或多个数据（由线程数量与数据量大小之比决定），求出这几个数据的SUM值，之后进行同步，等所有线程都执行完此步骤之后再进行下一层迭代。直到迭代到最后一步为止，回到主机函数，判断块的数量以及返回的结果数，再调用设备函数进行最终结果的计算。

为每个线程能处理多个数据，在调用内核函数时，向其传入一个delta参数，即为所有线程总数，之后每个线程使用全局线程ID对其进行叠加从而获取目标数据进行计算。

下面分别给出线程块内计算SUM的样例流程图和简化的内核函数实现。

图3-3 块内并行Reduce操作过程

\_\_global\_\_ void reduce\_kernel(int \*d\_data\_in, T \*d\_data\_out, unsigned int n)

{

\_\_shared\_\_ int sdata[];

unsigned int tid = threadIdx.x;

unsigned int i = blockIdx.x\*blockDim.x + threadIdx.x;

sdata[tid] = (i < n) ? d\_data\_in[i] : 0;

\_\_syncthreads();

for (unsigned int s=1; s < blockDim.x; s \*= 2)

{

int index = 2 \* s \* tid;

if (index < blockDim.x)

sdata[index] += sdata[index + s];

\_\_syncthreads();

}

if (tid == 0) d\_data\_out[blockIdx.x] = sdata[0];

}

3.3.4 Scan

Scan原语主要用来实现前缀和（Prefix Sum），也称前缀扫描（Prefix Scatan）。在数据库中主要用来支持select和filter算子的执行。它是并行数据库中的重要操作：给定一个输入关系（或数组），输出数组的值为，即为的和，其中。

Scan的实现主要分为两个步骤，首先利用Reduce原语对输入数据进行一遍前缀和计算操作（向上扫描），然后再对这个结果数组进行向下扫描，计算出其余部分和。

向上扫描的算法描述为：

**for** d:=0 **to** **do**

**for** k **from** 0 **to** n-1 **by**  **in parallel do**

x[k+] := x[k+] + x[k+]

其具体执行流程如下：

图3-4 Reduce向上扫描执行流程

当进行完第一次向上扫描后，部分位置已经计算出了前缀和或与相邻位置的元素之和，之后对这个结果数组再进行向下扫描。

向下扫描的算法描述为：

x[n-1]:=0

**for** d:= **down to** **do**

**for** k **from** 0 **to** n-1 **by**  **in parallel do**

t :**=** x[k+]

x[k+] := x[k+]

x[k+] := t + x[k+]

从根开始，先将之前计算出末位前缀和置0，之后与位于其前面距离处的元素叠加存入当前位置，并将原来的值存入与之叠加的元素的位置，所有线程执行相同的操作，最后将上图所示最后一步所得的结果与输入源数据相叠加，即得到最终的前缀和数组。其具体执行流程如下：

图3-5 Reduce向下扫描执行流程

3.3.5 Filter

Filter原语根据给定的条件找出输入数据中的目标数据并输出，用于SELECT算子。其具体实现通过Map和Scan相结合，先通过Map找出输入数据中符合条件的目标元组，并在标志数组中将其位置对应位置1，非目标值置0，之后通过Scan原语计算前缀和，从而找出目标元组。

实现方式如下图所示：

图3-6 Filter原语找出值为2的元素所在位置

3.4 原语组合成SQL算子

如下图3-7所示，有上节实现的基本原语可组合成各类SQL算子。



图3-7 原语组合成SQL算子

3.5 关键技术和改进点

方案基于全内存计算场景，利用GPU进行底层查询计算，从而达到加速数据库查询的目的。主要实现了Map，Scatter，Gather，Reduce，Scan和Filter等六个原语，通过这6个原语组合形成一系列SQL语句。在实现这些原语的过程中，研究了一些面向硬件结构的优化方法，如数据量太大以至于不能为每个数据单独分配一个线程并行处理，全局内存访问速度较慢导致性能下降等问题。方案都给出了相应的解决办法。具体细节在下面分别介绍。

3.5.1 大数据量处理

本文所提出的设计方案基于全内存计算，目前TITAN X 系列GPU的显存有12G左右，若采用Zero Copy机制则数据供全内存计算的数据会随着主存的增大而增大，而目前主流的服务器主存已经在128G以上。在面大如此大的数据量并行计算，CUDA内核的线程总数有限，不能做到为每一个数据都分配一个单独的线程，因此方案在实现内核函数时，向内核函数传递一个线程总数delta参数，从而美俄个线程处理以delta叠加直到超过输入数据总数的数据，这样每个数据都能被处理，在对一些同步要求比较多的原语如Reduce和Scan，方案实现根据数据量大小计算合适的线程数，使得每个线程处理的数据量差不多，从而达到减少同步延迟的问题。

3.5.2 全局内存访问延迟解决

当原语如Scan和Reduce需要多次访问全局内存时，带来的延滞会比较高，为此，我们采用先将数据缓存到线程块块共享内存中的方式，从而使得访问速度大大提高，并且也减少了线程块间的访问冲突问题。

3.6 本章小结

本章首先介绍了计算引擎的需求分析及设计目标，然后介绍了方案的整体架构以及具体实现方式，介绍了Map，Scatter，Gather，Reduce，Scan和Filter等六个原语的设计与实现方式，以及它们如何组合形成SQL底层算子，最后对方案中的关键技术和改进点进行了介绍并作出总结。

第四章 性能测试

4.1 测试方案

测试环境如表4-1，本计算引擎实现了一系列运行于GPU上的原语：Map，Scatter，Gather，Reduce，Scan和Filter。基于这些原语设计并实现了SELECT，PROJECTION，GROUP和AGGREGATION等SQL算子。由于这些SQL算子都是基于以上六个原语进行实现。对于等值和范围查询即SELECT算子是基于Filter原语进行实现的，而Filter原语是基于Map和Scan的组合进行实现，所以此处只比较CPU与GPU实现的Map，Scatter，Gather，Reduce，Scan五个原语的运行时间差异，计算查询加速比。

4.1.1 测试环境

表4-1 测试环境

|  |  |  |  |
| --- | --- | --- | --- |
| 操作系统 | CPU | GPU | 内存 |
| Ubuntu 14.04 LTS | Intel(R) Xeon(R) E5-2650 v4 | NVIDIA TITAN X | 64G |

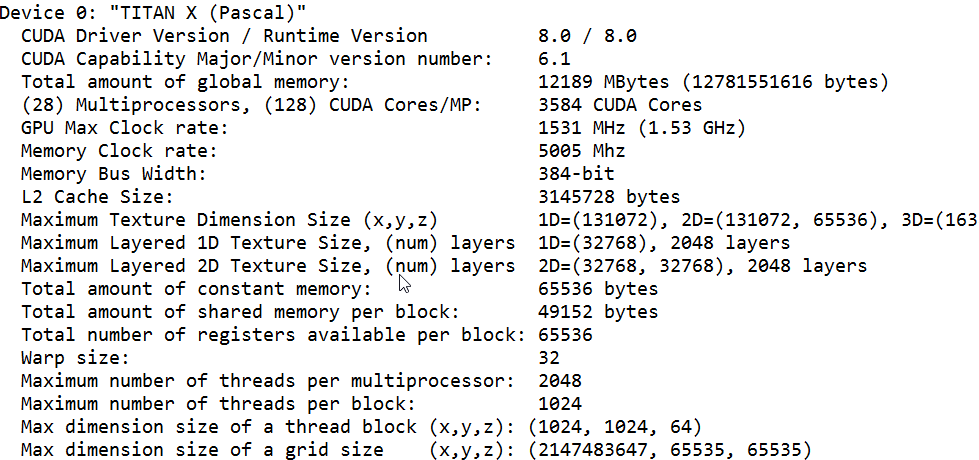
由于目前数据库的应用场景大部分在linux服务器环境下，故测试环境如表4-1所示，其中GPU采用NVIDIA公司最新的TITAN X系列GPU，其计算能力为6.1，与其进行比较的是目前主流的服务器CPU，即Intel Xeon系列的E5-2650，这样在做CPU与GPU的查询开销对比时相对公平一致，都为当前较先进主流的水平。内存采用64G可以更好的利于全内存计算的环境，当大量数据加载进内存不至于经常发生内存换入换出而导致开销增大。

图4-1 GPU性能详情

4.1.2 测试数据生成

所有测试源数据都是使用rand()随机生成的，为了保证统一性，三种查询方式使用的数据都应是一样的。所以在初次生成数据后会拷贝给GPU显存一份以用来进行拷贝方式查询，拷贝时计算拷贝开销。

4.1.3 CPU上的实现方式

为便于比较，按照同样的算法定义要求，实现了基于CPU的六个原语，与GPU上的实现进行比较。其中Map、Scatter和Gather均采用顺序遍历的方式对输入元数据进行要求的读写操作。Reduce 和Scan原语的实现即为传统的复杂度的扫面方式计算。Filter的实现是基于Map和Scan，实现方式与GPU类似。

4.1.4 正确性检验

由于数据量较大，而且随机生成，故我们每次随机生成完数据后，固定某一行或几行的值，这样在计算之前就已经知道某些行会是什么结果，然后在计算完后对这些行进行检验是否查询到，以及查询结果是否准确。

4.2 测试结果与分析

由于Map、Scatter和Gather基本都为顺序遍历，测试出来的数据相差不大，故这三个原语只以Map为例给出计算时间比较。另外还有两个原语Reduce和Scan则分别给出运行时间对比。

由表4-2以及图4-2可以知道，在Map原语的计算情况下，随着数据规模的增大，计算加速比加速到一定的程度之后出现下降的趋势，这是由于数据规模增大，拷贝开销也越来越大，而且拷贝开销占整个GPU运算过程开销的主要部分。而CPU是直接对内存寻址，其运行时间基本为线性。

对于Reduce原语，其加速比一直呈增长的趋势，这是因为方案在实现Reduce原语时进行了第三章所述优化，可以处理大量的数据，而且性能上相对CPU几乎不会下降许多。

对于Scan原语，其性能相对Map和Reduce较低，但也一直呈增长状态，这是因为对于CPU的Scan实现其时间复杂度为，基本就等于CPU上Map和Scatter等原语的时间复杂度。而在GPU上的Scan操作比GPU上的其他原语操作要复杂一些，不过相对于CPU而言，由于其并行性比较高，还是起到了一定的加速作用。

表4-2 计算时间比较

图4-2 三种原语计算时间加速比对比

|  |  |  |  |
| --- | --- | --- | --- |
| 数据量() | Map（ms） | Reduce（ms） | Scan（ms） |
| 100,000（CPU） | 0.442 | 0.566 | 0.794 |
| 100,000（GPU） | 0.071 | 0.120 | 1.265 |
| 500,000（CPU） | 2.206 | 2.735 | 3.413 |
| 500,000（GPU） | 0.234 | 0.253 | 1.751 |
| 1,000,000（CPU） | 4.396 | 5.470 | 6.852 |
| 1,000,000（GPU） | 0.432 | 0.428 | 1.801 |
| 5,000,000（CPU） | 22.364 | 27.314 | 36.647 |
| 5,000,000（GPU） | 1.389 | 1.804 | 4.909 |
| 10,000,000（CPU） | 44.781 | 54.880 | 73.624 |
| 10,000,000（GPU） | 3.695 | 3.519 | 8.613 |

综上，就传统CPU计算方式而言，GPU在一定程度上起到了加速作用，尤其是数据量较大的情况下，约在10倍左右。

4.3 本章小结

本章首先介绍了采用的测试方案以及测试环境，然后介绍了CPU的实现方式以及对比因素和正确性检验方式，最后比较了性能测试的结果，进行了分析从而验证了方案的可行性与优点。

第五章 全文总结与展望

5.1 全文总结

在科技不断发展的时代下，本计算引擎融合了时下流行的GPU计算技术提出了一种基于GPU 的等值、范围查询功能的计算引擎设计方案。方案实现了一系列运行于GPU上的原语：Map，Scatter，Gather，Reduce，Scan和Filter。基于这些原语设计并实现了SELECT，PROJECTION，GROUP和AGGREGATION等SQL算子。并且通过实施性能测试，验证了方案的效率。

5.2 后续工作展望

本方案虽然已经实现，但是还是有许多日臻完善的地方：

1、SQL算子还有很多，本计算引擎只是实现了一部分，另外对于JOIN等算子的实现还有待研究。

2、非结构化数据的查询问题，目前非结构化数据库也越来越热门，对于这些数据库系统的支持需要进一步研究。

3、对与并行计算的研究已经有几十年的历史了，基类下了许多宝贵的研究成果，在利用GPU进行并行计算实现原语的过程中还有待对更多资料的研究，不断改进完善。

要达到一个更加完善的计算引擎，等待解决和丰富的内容还有很多，这些都有待自己在接下来的时间里一一完成。

致 谢

大学四年转眼就要结束了，非常感谢在大学里认识的老师们、同学们。四年里，我总是不断的感到迷茫，老师们就像指路的明灯，一次又一次的为我指明了前进的方向。感谢身边的同学们一直以来支持和帮助，让我不断成长，同时也在四年的大学生活里收获了许多快乐而美好的回忆。也要感谢实验室的师兄师姐们一直依赖的帮助，是你们让我少走了许多的弯路。最后，我要感谢我的导师段翰聪老师，在本文完成的过程中总是耐心的指导，不断的从思路上进行点，从行为上进行督促，让我能更好的完成本篇论文。

再一次诚挚的对你们致谢，也祝各位老师工作顺利、身体健康。也祝自己给这四年的学习生涯画一个圆满的句号。

参考文献

1. De Witt D. J., Katz R. H., Olken F., et al. Implementation techniques for main memory database systems[A].In Proceedings of the 1984 ACM International Conference on Management of Data[C].Boston,USA:ACM,1984:1-8
2. Boncz P A, Zukowski M, Nes N. MonetDB/X100: Hyper-Pipelining Query Execution[C].CIDR. 2005, 5: 225-237.
3. Färber F, Cha S K, Primsch J, et al. SAP HANA database: data management for modern business applications[J]. ACM Sigmod Record, 2012, 40(4): 45-51.
4. Oukid I, Lehner W, Kissinger T, et al. Instant Recovery for Main Memory Databases[C]. 2015 7th Biennial Conference on Innovative Data Systems Research, Asilomar, California, USA. CIDR. 2015.
5. Yuan Y, Lee R, Zhang X. The Yin and Yang of processing data warehousing queries on GPU devices[J]. Proceedings of the VLDB Endowment, 2013, 6(10): 817-828.
6. Heimel M, Saecker M, Pirk H, et al. Hardware-oblivious parallelism for in-memory column-stores[J]. Proceedings of the VLDB Endowment, 2013, 6(9): 709-720.
7. Breß S, Beier F, Rauhe H, et al. Efficient co-processor utilization in database query processing[J]. Information Systems, 2013, 38(8): 1084-1096.
8. Breß S. The design and implementation of CoGaDB: A column-oriented GPU-accelerated DBMS[J]. Datenbank-Spektrum, 2014, 14(3): 199-209.
9. Bakkum P, Chakradhar S. Efficient data management for GPU databases[J]. High Performance Computing on Graphics Processing Units, 2012.
10. Li P, Luo Y, Zhang N, et al. HeteroSpark: A heterogeneous CPU/GPU Spark platform for machine learning algorithms[C]. 2015 IEEE International Conference on Networking, Architecture and Storage (NAS). IEEE, 2015: 347-348.
11. Jia Y, Shelhamer E, Donahue J, et al. Caffe: Convolutional architecture for fast feature embedding[C]. Proceedings of the 22nd ACM international conference on Multimedia. ACM, 2014: 675-678.
12. Codd E F. A relational model of data for large shared data banks[J]. Communications of the ACM, 1970, 13(6): 377-387.

外文资料原文



外文资料译文

通过CUDA使用GPU加速SQL数据库算子

Peter Bakkum and Kevin Skadron

Department of Computer Science University of Virginia,

Charlottesville, VA 22904 {pbb7c, skadron}@virginia.edu

摘要

以前的工作已经显示出对GPU上各种数据库操作的显着加速，但是仅使用不属于常规数据库语言（如SQL）一部分的原语。 本文直接在GPU上实现了SQLite命令处理程序的一个子集。 这大大减少了实现GPU加速所需的努力，避免了数据库程序员需要使用新的编程语言（如CUDA）或修改其程序以使用非SQL库。

本文着重于加速SELECT查询，并在SQLite命令处理器的高效GPU实现中描述考虑因素。 根据结果集的大小，NVIDIA特斯拉C1060的结果可实现20-70倍的加速。

1.介绍

GPU通常被称为视频卡，是计算机在屏幕上呈现图形信息的手段。现GPU的并行架构在某些问题上提供了非常高的吞吐量，并且其在台式计算机中几乎普遍使用意味着它是一种便宜且无所不在的处理能力来源。通过诸如NVIDIA CUDA（一种应用程序编程接口）开发的框架，将这种功能应用于更一般的非图形化问题越来越受到人们的关注，为程序员提供了一种在NVIDIA GPU上执行通用逻辑的简单而标准的方法。程序员经常使用CUDA和类似的接口来加速计算密集的数据处理操作，通常执行速度提高了五十倍，程序员必须了解六个不同的存储空间，CUDA线程和线程块如何映射到GPU硬件，理解CUDA的线程内部通信等.CUDA已经使GPU开发更接近于主流，但程序员仍然必须为GPU执行的每个数据处理操作编写一个低级别的CUDA内核，这是一个频繁重复工作的时间任务。

SQL是用于操纵和查询数据库的行业标准通用声明性语言。能够执行非常复杂的数据集连接和聚合，SQL被用作过程程序和数据结构化表之间的桥梁。SQL查询的加速将使程序员能够通过很少或根本不改变其源代码来提高数据处理操作的速度。尽管需要GPU程序加速，但是即使在GPU上严格地仿真了SQL查询来证明并行架构对这种执行模式的适应性，SQL也不能自动访问GPU。

目前的GPU技术存在限制，这些技术影响了这样一种GPU SQL实现的潜在用户。两个最相关的技术限制是GPU内存大小和主机到GPU设备内存传输时间。虽然未来的显卡几乎肯定会有更大的内存，但目前的NVIDIA卡最多可以达到4 GB，是许多数据库大小的一小部分。在CPU和GPU之间传输内存块仍然是昂贵的。因此，将数据行分段到GPU和分段结果行返回需要很大的开销。尽管有这些限制，实际的查询执行可以通过GPU的高度并行组织并发运行，因此优于执行CPU查询。

尽管有上述限制，但仍有许多适用于该项目领域的应用程序。许多数据库，如用于研究的数据库，不频繁地修改数据，并在读取查询期间遇到最大的负载。另一组应用程序更关心特定查询的延迟，而不是严格遵守最新的数据，一个例子是互联网搜索引擎。对大尺寸数据集的许多查询仅处理总数据的一个子集，从而邀请将该子集分为GPU存储器。另外，虽然GPU的有限存储器大小是一个很大的限制，但只分配了Tesla C1060的4 GB的一半来存储数据集，为用户提供了超过1.34亿行4个整数的空间。