**PostgreSQL中基于GPU的排序**

Naju Mancheril

School of Computer Science

Carnegie Mellon University

[naju@cmu.edu](mailto:naju@cmu.edu)

翻译作者朱君鹏简介：华东师范大学博士研究生，个人兴趣主要集中在：新型硬件（GPU、RDMA、FPGA）等在数据库中的应用，架构设计与并行计算。

翻译作者陈河堆简介：中兴通讯数据库平台负责人，曾参与和主导公司自研高性能内存数据库、分布式数据库和分布式缓存等系统的开发和设计，目前致力于PostgreSQL数据库的研究和推广

## 摘要

尽管每个微处理器内核的晶体管数量在过去十年中有所增加，但复杂性的增加仅导致应用性能的适度增加。大多数性能改进可以追溯到技术扩展的更快时钟速率。虽然复杂的、广泛的、超标量核心提供更高的单线程性能，但它们通常不适合具有大量独立线程的应用程序。在后一种情况下，具有多个简单独立内核的多处理器可提供最佳性能[1]。本文不寻求解决哪种复杂程度更适合数据库管理系统的执行。相反，我们探索异构处理环境提供的优势。

具体来说，我们分析了使用GPU来帮助我们的排序操作所带来的性能改进。为此，我们将基于GPU的快速排序算法GPUSort集成到PostgreSQL数据库管理系统中。我们使用GPUSort来实现用于处理SQL语句中的ORDER BY子句的键指针排序。我们发现由于设置基于GPU的排序的通信开销，qsort()仍然是排序小关系的更快的选择。但是，当关系达到中等大小（大约1MB）时，GPUSort表现更好。

## 概述

在过去十年中，每个微处理器内核的晶体管数量有所增加，但复杂性的增加仅导致应用性能的适度增加。大多数性能改进可以追溯到技术扩展的更快时钟速率。虽然更复杂的内核提供更高的单线程性能，但它们通常不适合具有大量独立线程的应用程序。在后一种情况下，具有多个简单独立内核的多处理器可提供最佳性能[1]。本文不寻求解决哪种级别的复杂性更适合数据库管理系统。相反，我们探索异构处理环境提供的优势。特别是，虽然在通用CPU上执行我们的数据库管理系统，但我们分析了使用一个GPU来辅助排序操作可能提供的性能改进。

现代GPU（图形处理单元）的结构与通用CPU非常不同。 GPU针对一系列操作（比如纹理映射）进行了优化，这在CPU中是没有等效硬件支持的。它们的内存子系统也不同，因为除了板载视频内存之外，GPU还必须管理到系统RAM的高速接口。然而，最令人吃惊的区别在于对并行性的硬件支持水平。虽然现代GPU的时钟频率仅为400 MHz，但它可以有多达20个独立的流水线。每个时钟周期，有大量指令可以执行完毕[2]。

由于这种并行性，现代GPUS在原始FLOPS（每秒浮点运算）方面甚至超过了高端CPU。最近，在利用这种处理能力并用它来解决通用计算中出现的复杂问题方面已经做了很多工作[3]。一个这样的问题是排序。排序是现代数据库和数据挖掘系统中主要延迟的原因。这些系统通常简化为使用基于CPU的内存快速排序。在本文中，我们展示如果允许这些系统访问图形处理器，他们可以利用特别适合现代GPU功能的快速并行排序算法。

北卡罗来纳大学教堂山分校的GPUSort项目致力于设计和实现基于GPU的高效排序算法。我们使用的GPUSort实现通过使用OpenGL纹理映射操作将一个bitonic排序网络映射到GPU。排序算法还通过使用高速缓存有效的内存访问模式充分利用了图形卡的内存带宽。图1显示了使用一个第一代旧NVIDIA GPU [4]在3.4 GHz奔腾4处理器系统上运行的GPUSort击败qsort()。

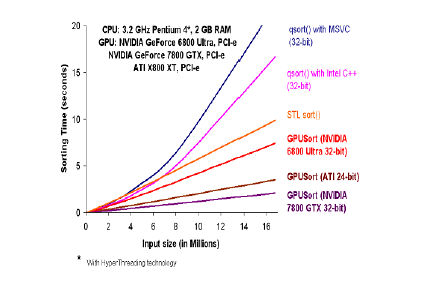


图1 原始GPUSort和原始qsort() [4]

我们通过将其与PostgreSQL数据库管理系统集成来探索GPUSort提供的改进。具体来说，我们使用它来实现用于处理SQL语句中的ORDER BY子句的键指针排序。我们发现由于将元组传递给图形卡的通信开销，qsort()仍然是排序小关系的更快选择。但是，当关系达到中等大小（大约1MB）时，GPUSort表现更好。

本文的其余部分的结构如下。第2节简要介绍了Bitonic排序网络。第3节讨论了我们修改过的PostgreSQL系统的体​​系结构。第4节包含我们实验评估的设置。第5节包含我们的实验结果和分析。最后，第6节包含我们的结论和未来工作的可能领域。

## Bitonic排序

Bitonic排序（或bitonic归并排序）[5]是一种利用排序网络的快速排序算法。排序网络[6]是一个特殊的基于比较的并行排序过程，其中比较序列不依赖于数据[7] [8] [9]。这使得排序网络成为理想的并行化功能，甚至可能在硬件中实现。

假设有一个序列a = a0，....，a n−1，其中ai∈{0,1}， i = 0，...，n−1，它称为0-1序列。 如果在0和1比特系列之间最多包含两次变化，则该0-1序列称为bitonic。即如果存在子序列长度k，m∈{1，... ，n}，使得

a0, . . . , ak−1 = 0, ak, . . . , am−1 = 1, am, . . . , an−1 = 0

或

a0, . . . , ak−1 = 1, ak, . . . , am−1 = 0, am, . . . , an−1 = 1

Bitonic排序网络有两个主要组成部分：比较器网络和归并网络。这些组件是递归定义的。如果Bn是一个能对n位数进行排序的比较器网络，则有：

Bn(a) = b0, . . . , bn/2−1 c0, . . . , cn/2−1

其中，对于i，j∈{1 ，...，n / 2 - 1}，有bi≤cj，而且，b0 ，... ，bn/2-1和c0 ，... ，cn/2-1，每个都是bitonic。归并网络将这两个有序的bitonic序列组合成一个。现在对n位进行排序成为一个标准的分而治之算法，包括拆分和排序阶段，然后跟随一个归并阶段。

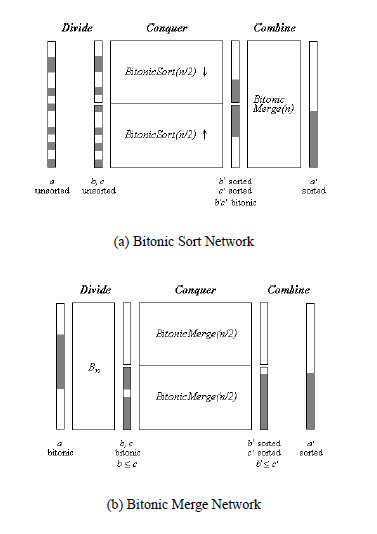


图2 Bitonic 排序[10]

每个排序和归并迭代称为一个阶段。通过使用n个独立处理器的阵列，bitonic排序允许我们在O（log n）阶段内对n个元素进行排序，且每个阶段只需O（log n）个步骤。总运行时间为O（log2 n），远远优于quicksort [11]提供的O（n log n）时间。

GPUSort使用OpenGL纹理映射函数将32位浮点比较器网络映射到我们图形处理器的20条流水线上。因此，我们可以在每个400 MHz周期中执行完大约20个比较指令，与之鲜明对比是，每个周期3.0GHz CPU只能执行完2-3个指令。

## 3．数据库体系结构

PostgreSQL中的排序操作是使用tuplesort模块实现的。尽管所有数据库操作符都具有迭代器风格的接口（open，next，close），但正确排序需要访问所有输入元组。因此，tuplesort接口要求sort操作符代码从其输入流中对每个元组调用puttuple（），然后调用performsort（）进行实际排序，最后在每次next（）调用时调用gettuple（），直到每个元组都被有序获取。

peformsort（）函数可以以两种方式之一工作。对于小数据集（小于1MB），它构造一个元组指针的纯内存数组，定义一个比较器函数来对两个元组指针进行排序，然后调用libC 的qsort（）函数。对于较大的数据集，它使用带堆排序的外部排序来构建有序的执行系列。

图3显示了我们所做修改的高层次描述。一旦我们能够从元组指针中抽取浮点键，我们就能够构造自己的浮点数偶对数组，并将这个纯内存数组直接传递给GPUSort代码。完成之后，使用有序的键指针数组将原始的元组数组置换为有序顺序。

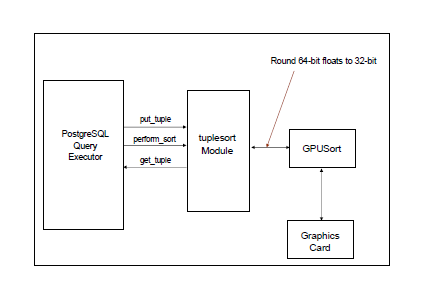


图3 PostgreSQL排序体系架构

尽管SQL将FLOAT定义为4字节浮点值，但PostgreSQL内部使用64位（双）数据类型来表示它。只可惜GPUSort当前只支持16位和32位浮点数，否则这不算什么问题。出于这个原因，在应用GPUSort之前，我们必须将每个double值舍入到最接近的float值。此舍入过程不会更改结果元组的集合，因为在我们的工作负载中使用的元组不会使用超过几位数的精度。反而，在评估最终实现时，应牢记舍入的代价。完全采用GPUSort的任何系统要么使用32位浮点数，要么获取支持64位键值的GPUSort版本。

另一个问题是我们的GPUSort版本似乎是专门为32位机器设计的。因此，即使我们将FLOAT32指针偶对的数组传递给排序例程，OpenGL代码也假设每个偶对的总大小为8个字节。我们通过传入FLOAT32偏移偶对数组来解决这个问题：我们传递了每个元素在元组数组中的偏移量，而不是每个元素的绝对地址。由于偏移量是32位数，我们仍然可以对最多40亿个元组的数组进行排序，但我们希望在将来的工作中消除这种限制。

## 4．实验设置

### 4.1 硬件

所有的开发和评估都是在具有2048KB缓存和超线程的3.0GHz Intel（R）Pentium（R）4 CPU上完成的。系统配备1GB主内存。使用的显卡是eVGA Geforce 7800GT，配备256MB RAM。核心GPU速度为400 MHz，板载内存运行速度为500 MHz。该卡使用x16 PCI-Express接口，最大额定内存带宽为54.4GB /秒[2]。

### 4.2 软件

我们的排序是来自北卡罗来纳大学教堂山分校的GPUSort项目的GPUSort 1.0。我们将它集成到PostgreSQL 8.2devel中。所有评估都在Fedora Core第4版上完成，Linux内核版本为2.6.11-1.1369\_FC4smp，针对x86\_64进行了优化。

### 4.3 工作负载

我们使用一个微基准测试风格的评估，因为我们只关心测量排序操作符的性能。实验使用如下表模式：

TABLE vals(x FLOAT, y FLOAT)

我们在这个表中填充了一百万个元组，这些元组在0.0到100.0之间随机均匀生成。我们所有的查询都来自以下模板：

SELECT \* FROM vals

WHERE y < y0

ORDER BY y

这个简单的查询总是产生如图4所示的双运算符查询计划。

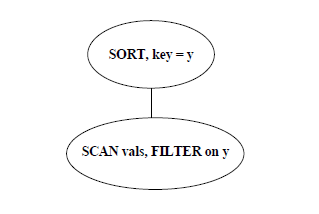


图4 微基准测试查询计划

由于基表中的元组是随机均匀生成的，我们可以通过改变y0的值来控制SCAN运算符的选择性。例如，通过将y0设置为10.0，我们可以执行一次对（10.0 / 100.0）·100万= 100,000个元组的排序。

在我们的实验中，我们改变了关系大小以及图形卡上GPUSort代码使用的视频内存量。另一个可能调节的参数是元组大小。但是，这不会影响排序成本，因为PostgreSQL实现了键指针排序。无论每个元组的大小如何，当我们执行比较和交换时，我们只会移动浮点数和字大小的指针（或者对于GPUSort来说是一个32位的偏移）。

## 5. 实验结果

### 5.1 大输入

我们比较了三种不同排序算法的执行时间：默认的PostgreSQL外部排序，PostgreSQL内部排序和GPUSort。图5显示了这三种算法在大输入规模下的性能。 “堆排序”是PostgreSQL外部排序算法。它使用堆排序来创建每个1MB的有序执行系列，然后将它们归并在一起。 “qsort”表示内部排序算法，它实际上只是对libC qsort（）函数的调用。出于测量目的，我们将qsort使用的内存量增加到1GB。 GPUSort 256MB显示使用256 MB视频卡的GPUSort运行时间。

需要注意的一点是，“堆排序”的执行时间不随输入大小单调增加。由于该算法基于分而治之思想，因此当输入可以被均分多次时，它将表现得更好。虽然在图5中很难看得出来，但GPUSort也表现出这种行为特征。这在图6中应该看得更清楚，它显示了GPUSort如何随着更大输入而按比例增长1。

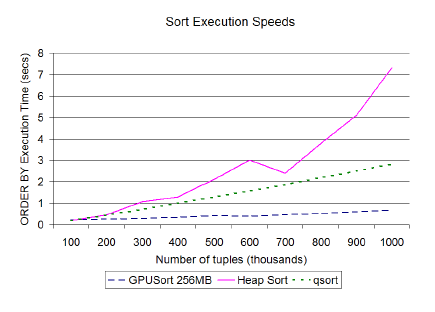


图5 ORDER BY子句执行时间

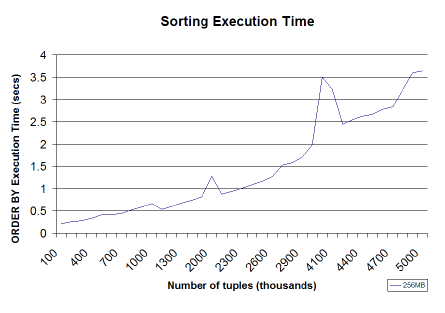


图6 GPUSort ORDER BY子句执行时间

每当元组数量跨过一个2的幂次方时，我们就会看到排序时间的一次下降。我们在50万、100万、200万和400万个元组时分别看到了这个现象。对420万个元组进行排序所花费的时间比对400万个元组进行排序所花费的时间要少。此行为实际上是GPUSort库的实现方式。在应用纹理映射例程之前，GPUSort会将元组编组为内部矩阵。当前实现不会对完全在GPU上对数组进行排序，它将产生非2的幂次方大小的矩阵。相反，它采用比输入数组尺寸小的2的最大幂，在GPU上对结果矩阵进行排序，使用qsort（）对剩余数据进行排序，最后将两个流归并在一起。由于涉及到额外的CPU使用开销，代码运行在非2的幂次方大小的数组上执行得更慢[4]。

好消息是即使我们可能会看到执行时间的这种变化，但是，当数据集变大时，GPUSort总是比qsort（）快。即使查询优化器必须决定使用哪种排序算法，也无需专门处理这些2的幂次方问题。事实上，对于大型（大于1MB）数据集，GPUSort的执行速度始终是内部快速排序的4倍。

## 5.2 小输入

我们已经看到GPUSort的性能取决于它需要多少CPU支持。尽管图5显示GPUSort的性能优于基于CPU的替代方案，但这些测量是通过大输入来完成的，这里的执行时间主要由排序时间来决定。情况并非总是如此。我们可以将应用程序用于执行GPUSort的总时间分为四个部分：

Tsort = Tarray\_build + Tnetwork\_build + Ttransfer + TGPU + TCPU

Tsort是总排序时间，Tarray\_build是使用CPU构建键偏移数组所需的时间，Tnetwork\_build是在GPU上构建一个bitonic排序网络所需的时间，Ttransf er是将数据传输到GPU所需的时间。TGPU是对显卡进行排序所需的时间，而TCPU是qsort（）剩余数据和归并两个有序流所需的任何额外时间。

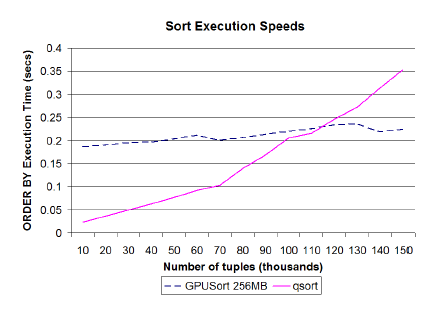


图7 ORDER BY子句处理时间

创建单独的键指针数组+Tarray\_build+创建排序网络+Tnetwork\_build+将元组传递到图形卡Ttransfer的开销对于排序小关系而言可能太高。图7展示了这种情况。当要排序的元组数量小于10万时，qsort（）实际上是更快的选项。

## 5.3 显存大小

GPUSort必须在显卡的视频内存中保留空间，以存储将要被排序的矩阵。通过改变GPUSort对总视频内存大小的估算，我们可以控制它生成的矩阵的大小。我们很想知道当GPUSort拥有处理功能较弱的图形处理器时的性能表现。图8显示了此功能伸缩时的结果。

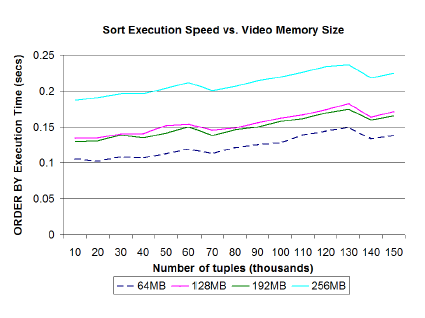


图8 改变视频内存

该图表中的奇怪结果是256MB显卡表现最差。256MB是我们使用的视频图形卡的总内存容量。最初，我们认为这种糟糕的性能可能是视频内存和系统RAM之间抖动的结果。毕竟，PostgreSQL必须与XWindows和其他可能正在运行的应用程序共享显卡。然而，通过运行更多测试，我们可以找到更简单的解释。回想一下GPUSort的执行时间分解：

Tsort = Tarray\_build + Tnetwork\_build + Ttransfer + TGPU + TCPU

事实证明，Tnetwork\_build根据GPUSort将使用的视频内存量成比例进行缩放。为了构建我们的排序网络，我们必须在显卡上分配内存，决定在哪里放置我们的矩阵，并承担在系统RAM和显卡之间建立未来传输所涉及的任何开销。事实证明，没有必要为每次排序构建一个新的排序网络。我们可以“预先计算”卡上的排序网络，并在每次排序中重复使用它。其结果，Tnetwork\_build被摊销了，而我们看到了更合理的执行时间：

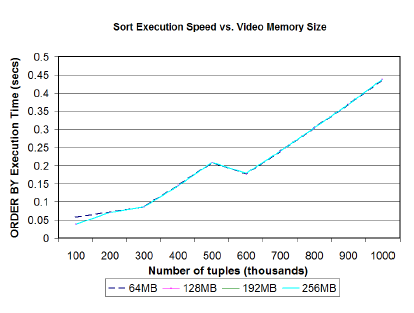


图9 改变视频内存与重用排序网络

图9显示，对于大型数据集，GPUSort非常之快，以至于它实际并在乎GPU可以访问多少视频内存。其性能仅受GPU速度和可用于将数据从系统RAM传输到视频RAM的带宽的限制。根据这些参数测量性能是未来工作的主题。

## 6. 结论

本文通过将其与PostgreSQL数据库管理系统集成，探讨了GPUSort（一种基于GPU的快速排序算法）所提供的改进。具体来说，它被用来实现用于处理SQL语法ORDER BY子句的键指针排序。我们发现由于将元组传输到显卡所涉及的开销，qsort（）仍然是较小关系的更快选择。然而，当关系的大小变为1 MB级别时，GPUSort会更快。

现代GPU不会取代通用CPU，但其独特的并行流水线设计给予它们巨大潜力，用于提升计算和内存受限程序的性能。即使是64MB显卡（其售价低于100美元），也可以比传统的基于CPU的qsort（）提供很大改进。

GPU提供的好处只会随着每张卡内流水线的速度和数量而增加。随着PCI Express的出现，总线传输变得越来越不成为瓶颈，这种技术提供了两倍于旧AGP-8x接口的独立传输通道。这些进步促成了许多项目的创建，特别是GPGPU项目，他们完全致力于设计和实现用于通用应用软件编码的GPU协处理器库。虽然基于图形库的应用程序可能更难开发，但它们最终会胜出：这些库提供的性能提升将超过复杂性的增加。

数据库系统不是唯一在不同执行阶段具有非常不同资源需求的应用程序。在“正常”执行期间，存在大量指令级并行性。为了执行这样的程序，一个多发的超标量CPU提供了最佳性能。但是，当应用程序需要执行一些计算密集型操作（如排序）时，大量独立流水线提供了最佳性能。最后，内核复杂性的最佳选择实际上可能是“以上全部”：同时具有高和低复杂度内核的异构芯片多处理器，可以在程序执行的不同阶段分别激活[1]。

## 引用

[1] R. Kumar, D. Tullsen, N. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. IEEE Computer, 38, No. 11:32–38, 2005.

[2] NVIDIA Corporation. NVIDIA GeForce 7800 GPUs Specifications.

[3] GPGPU Project. <http://www.gpgpu.org>.

[4] GPUSort Project. GPUSort Documentation.

[5] K.E. Batcher. Sorting networks and their applications. volume 32, pages 307–314, 1968.

[6] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. Introduction to Algorithms. MIT Press / McGraw-Hill, Cambridge, Massachusetts, 1990.

[7] C.D. Thompson and H.T. Kung. Sorting on a meshconnected parallel computer. Communications of the ACM, 20 4:263–271, 1977.

[8] H.W. Lang, M. Schimmler, H. Schmeck, and H. Schroder. Systolic sorting on a mesh-connected network. IEEE Transactions on Computers, C-34, 7:652–658, 1985.

[9] C.P. Schnorr and A. Shamir. An optimal sorting algorithm for mesh-connected computers. pages 255–261, 1986.

[10] Institut fur Medien Informatik und Technische Informatik. Sequentielle und parallele Sortierverfahren: Bitonic Sort.

[11] Randima Fernando. GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics. Pearson Education, Boston, Massachusetts, 2003.

文章链接：https://pdfs.semanticscholar.org/869f/beb4db447174d4c6a475e942a2607465a410.pdf