第40卷 第3期
2013年3月
计 算 机 科 学
Computer Science
Vol. 40 No. 3
Mar 2013

基于 Fermi 架构的 Join 算法

李观钊[1] 陈思桐[1] 甄 真[2] 陈 虎[2]

(华南理工大学计算机科学与工程学院 广州 510006)[1] (华南理工大学软件学院 广州 510006)[2]

摘 要 在列数据库中,连接操作依然是最核心和最耗时的操作,GPU 强大的计算能力可为此提供新的优化手段。基于 Fermi 架构,提出了新的 Hash Join 算法和 Sort-merge Join 算法,其基本思想是充分利用该架构新增的缓存结构来减少连接操作的 cache 缺失率。与 CUDA stream 技术相结合,新算法在输出结果较多时可以有效地隐藏主存与显存间数据传输带来的延迟,进一步提升其执行效率。实验结果证实了基于 Fermi 架构的 Hash Join 算法处理偏斜数据的高效性及 Sort-merge Join 算法的稳定性,并且通过比较表明,这两种算法的性能全面优于基于多核 CPU 充分优化的 Join 算法,最大加速 2.4 倍,在外键分布高偏斜时新的 Hash Join 算法的执行速度甚至达到每秒 217M 元组。

关键词 Join 算法,Fermi 架构,缓存,CUDA stream

中图法分类号 TP311.133.2 **文献标识码** A

Join Algorithms Based on Fermi Architecture

LI Guan-zhao[1] CHEN Si-tong[1] ZHEN Zhen[2] CHEN Hu[2]

(School of Computer Science and Engineering, South China University of Technology, Guangzhou 510006, China)[1]

(School of Software, South China University of Technology, Guangzhou 510006, China)[2]

Abstract In column database, the join operation is still the most important and the most time-consuming operation. GPUs' powerful computing capabilities can provide new optimizing solutions. This paper proposed a new Hash Join algorithm and a new Sort-merge Join which are both based on Fermi architecture. These two new implementation approaches take full advantage of the new parallel cache hierarchy of Fermi, and successfully reduce the cache miss rate of the join operation. Combined with CUDA stream technology, the new join algorithms can effectively hide the data transfer delay between the main memory and global memory, when the join operation generates a large number of results. The experimental results show that Hash Join based on Fermi performs better when it deals with data skew, while Sort-merge Join based on Fermi is more stable. Comparing with the implementations based on multi-core CPUs, these two algorithms are faster, speed up 2.4 times at most, and the new Hash Join even achieves 217M tuples per second when foreign key's dataset is high skew.

Keywords Join algorithm, Fermi architecture, Cache, CUDA stream

1 引言

NVIDIA 的 CUDA 计算架构简化了 GPU 编程,其本身也在不断的发展中。其中,相比于较早的版本 G80 和 GT200,Fermi[11]架构发生了革命性的变化。在存储器结构方面,Fermi 增加了可配置的 L1 缓存和读写统一的 L2 缓存。L1 缓存和 shared memory 可以增加临时寄存器的使用,避免寄存器溢出,减少每个线程读写 DRAM 的机会,而读写 DRAM 的延迟为几百个时钟周期。当对大量的数据随机操作时,可透明地应用到 L2 缓存,其起到了极大的加速作用。此外,Fermi 架构中的每个核心增加了一个全速的整型单元,以实现单周期的 32bit 整型运算,而双精度浮点能力是 GT200 的 8 倍。Fermi 架构支持 kernel 的并行执行,相比 G80 和 GT200,任务间的上下文切换速度提升了 10 倍。

与此同时,列数据库作为高效的数据存储技术,广泛应用于海量数据查询、智能数据分析等领域。所谓列存储,指的是按数据库记录的单个属性组织和存储数据,在单独的文件中连续存储同一属性中的所有数据。这为列数据库带来了磁盘读写开销小、复杂查询效率高等优势。在列数据库中,连接操作依然是最常用和最耗时的核心数据库操作,通常做法是对两个单独属性进行连接,得到相应的索引对,其性能直接影响整个列数据库的查询性能。因此,如何充分利用当前 GPU 的结构特点来优化列数据库连接算法是本文主要研究的内容。

本文将着重探讨在 Fermi 架构下的 Hash Join 算法和 Sort-merge Join 算法的实现,分析影响这两种列数据库连接

到稿日期:2012-10-19 返修日期:2012-12-22 本文受广东省科技计划项目(2011A010801008,2011A090200122,2011A090200027)资助。

李观钊(1987-),男,硕士生,主要研究领域为 GPU 并行计算和并行数据库,E-mail:liguanzhao@163.com;陈思桐(1987-),男,硕士生,主要研究领域为多核 CPU 并行计算、并行数据库;甄 真(1987-),男,硕士生,主要研究领域为编译原理、并行数据库;陈 虎(1974-),男,副教授,主要研究领域为面向多核处理器的体系结构、并行软件开发。

操作算法的性能的各种因素，并与对应的多核 CPU 实现方法进行性能对比。

2 相关工作

目前，数据库领域出现了各种各样的连接算法，其中包括 Nest-loop Join、Hash Join 和 Sort -merge Join 等，并且在不同的计算平台上实现了这些算法。

He 等[1]学者在 GPU 上实现了 Map、Prefix Scan、Scatter and Gather、Split、Sort 等原语，并使用这些原语实现了基于 GPU 的 Nested-loop Join、Sort-Merge Join 和 Hash Join 算法。与 CPU 的实现方法相比，其最大可以达到 7 倍加速。但文章的 GPU 实现方法并没有结合 Fermi 结构的特点，充分利用其中的 L1 和 L2 缓存。

Kim 等[2]充分利用现代多核 CPU 的结构特点和 SIMD 指令实现了 Radix-cluster Hash Join 算法和 Sort-merge Join 算法，通过比较，其中 Hash Join 算法每秒连接多于 100M 元组，相比所报告的 GPU 实现方法快 8 倍。本文还指出在未来的处理器结构上，Sort-merge Join 将比 Radix-cluster Hash Join 快。

Blanas 等[3]提出了一种简单、高效的基于多核 CPU 的 Hash Join 算法，该算法并不对内关系表进行分区，而是构建一个共享式的 hash 表，探寻过程中可以充分利用 SMT 优势来隐藏内存时延，其性能与 Radix-cluster Join 算法相接近，甚至在数据偏斜度较高的情况下，该算法性能更好。Chen 等[4]采用组预取和软件流水方法优化了 NSM 模型下的 join 操作，性能相应提高了 9 倍。

文献[5]指出连接操作是大规模的随机操作，并不具有良好的时间局部性，而且构建的 hash 表和探测 hash 表也不具备良好的空间局部性，而在列存储化的情况下，元组步长较小，可以获得较好的高速缓存利用率，提升空间局部性，因此提出了一种新的高速缓存敏感的磁盘连接算法 DBCC-Join。

3 基于 Fermi 架构的 Join 算法

本文假设在列数据库中连接操作要处理的主键属性向量为 $R=\{RKey_1, RKey_2, \cdots, RKey_r\}$，外键属性向量为 $S=\{SKey_1, SKey_2, \cdots, SKey_s\}$，其中 r 和 s 分别为 R 和 S 的元组个数，需要得到的结果是属性索引对向量 $\langle Sid, Rid\rangle=\{\langle Sid_1, Rid_1\rangle, \langle Sid_2, Rid_2\rangle, \cdots, \langle Sid_n, Rid_n\rangle\}$，$n$ 为结果个数。r、s 和 n 3 者不一定相等，并且 n 小于或等于 r。

Hash Join 算法是最常用的连接算法，包括 partition 和 probe 两个阶段。partition 阶段利用 hash 函数 H，按照 $RKey$ 的 hash 值将 R 中各元组散列到相应的桶，构建 Hash 表；probe 阶段则读取 S 中的各个元组，根据相同的 hash 函数 H，将其跳转到 hash 表中相应的桶，然后顺序扫描该桶，如果匹配到相同的元组，则输出相应的索引对，如此循环，直至得到最终的 $\langle Sid, Rid\rangle$。

传统的 Sort-merge Join 算法同样包括两个阶段，分别是 sort 和 merge。sort 阶段对 S 和 R 进行排序，可以使用快速排序、基数排序等常见的排序算法；merge 阶段则读取 S 中的元组，逐步扫描 R，匹配到相同的元组后，记录当前位置，因为 S 的下一个元组大于或等于当前元组，所以下一趟扫描只需从 R 的该位置开始，如此循环，直至输出最终的 $\langle Rid, Sid\rangle$。这样，整个连接过程只需要对 S 和 R 进行一趟扫描。

3.1 基于 Fermi 架构的 Hash Join

基于上述 Hash Join 原理，根据 Fermi 的体系结构特点，GPU 上的 Hash Join 对两个阶段进行了详细的优化。算法描述如下。

算法 1 基于 Fermi 架构的 Hash Join

输入：外键向量 S，主键向量 R

输出：外键和主键索引对向量〈Rid，Sid〉

第 1 步 首先在 global memory 中分配全局的 hash 表、直方图 Histogram、hash 桶的起始位置表 StartAddrTable 及全局的结果向量 Result，并进行初始化，将 R 和 S 从主存传输到显存；

第 2 步 每个线程读取 R 的一部分元组，利用 hash 函数得到每个 RKey 的 hash 值，统计散列到每个 hash 桶的元组个数，得到相应的 Histogram；同样使用多线程对 Histogram 求前 n 项和，获取每个 hash 桶的起始位置表 StartAddrTable；

第 3 步 再一次扫描 R，每个线程读取一部分元组，使用与第 2 步同样的 hash 函数，以每个 RKey 的 hash 值作为 StartAddrTable 的下标，获取桶的起始位置，将〈RKey，Rid〉对划分到 hash 表相应的桶中，并对该桶的起始位置原子加 1，每个线程如此循环直至所有主键元组都划分到 hash 桶中，完成构建 hash 表；

第 4 步 每个线程里面申请一块临时空间，存储该线程探寻到的结果，然后读取 S 的一部分数据，根据每个 SKey 的 hash 值，将其跳转到 hash 表相应的桶，与桶中的〈RKey，Rid〉逐个比较，如果 SKey 与 RKey 相等，则输出〈Sid，Rid〉到临时空间里面，每个线程如此循环直到所有外键元组探寻完毕，记录每个线程的结果数 SumPerThread；

第 5 步 分配全局变量 Sum，初始化为 0，每个线程竞争获取写入结果至 Result 的起始位置，将临时空间的〈Rid，Sid〉复制到 Result 中，并对 Sum 原子加 SumPerThread，所有线程复制完毕后，Sum 即为结果总数。

第 6 步 将 Result 向量从显存传输到主存，并释放所有的显存空间。

本文将第 1 步称为 memHtoD 阶段，第 2、3 步称为 partition 阶段，第 4、5 步称为 probe 阶段，第 6 步称为 memDtoH 阶段，这样便于后面的实验分析。在每个步骤中，为了充分利用 GPU 流处理器的执行单元，每个 Block 的线程数一般设置为 256。同一个 warp 内的线程访问的地址是连续的，同时也与 global memory 的传输段对齐，这将获得更高的访存带宽。在第 2、3、5 步中则使用了 CUDA 的原子操作 atomicAdd，由于在 Fermi 架构中大量的原子操作单元和 L2 缓存的使用，增强了原子操作的能力，因此其对算法性能影响不大。同时因为 Fermi 架构增加了 L2 Cache，Cache 行大小为 32byte，故 S 所构建的 hash 表中每个桶大小维持在 32byte，以便提升 Cache 的命中率。此外在第 2 步求前 n 项和时，本文使用了 CUDA Thrust 库中的 scan[9]函数，该函数在对 1M 个整数求前 n 项和时最少仅耗时 0.79ms，并且时间成本随着数据量呈线性增长，因此这部分的时间开销非常小。

3.2 基于 Fermi 结构的 Sort-merge Join

借助 CUDA 的 Thrust 库，GPU 的 Sort-merge Join 算法实现比较简单，整个过程描述如下。

算法 2 基于 Fermi 架构的 Sort-merge Join

输入：外键向量 S，主键向量 R

输出：外键和主键索引对向量〈Sid，Rid〉

第 1 步　在 global memory 分配全局的结果向量 Result，排序后的中间向量〈SKey，Sid〉=｛〈$SKey_1$，Sid_1〉，〈$SKey_2$，Sid_2〉，…，〈$SKey_s$，Sid_s〉｝和〈RKey，Rid〉=｛〈$RKey_1$，Rid_1〉，〈$RKey_2$，Rid_2〉，…，〈$RKey_r$，Rid_r〉｝，并将 S 和 R 从主存传输到显存；

第 2 步　分别对 S 和 R 按 Key 值排序，得到相应的〈SKey，Sid〉向量和〈RKey，Rid〉向量；

第 3 步　每个线程里面申请一块临时空间，存储该线程探寻到的结果，然后读取〈SKey，Sid〉向量的一部分元组，因为 SKey 值有序，所以使用二分查找法在〈RKey，Rid〉向量中探寻 SKey，如果 SKey 与 Rkey 相等，则输出〈Sid，Rid〉到临时空间里面，每个线程如此循环直到所有外键元组探寻完毕，记录每个线程的结果数 SumPerThread；

第 4 步　与算法 1 的第 5 步的方法一样，把所有线程中缓冲区的结果写回到 Result 向量中；

第 5 步　将 Result 向量从显存传输到主存，并释放所有的显存空间。

同样地，为了简化实验分析，将第 1 步称为 memHtoD 阶段，第 2 步称为 sort 阶段，第 3、4 步称为 merge 阶段，第 5 步称为 memDtoH 阶段。其中，sort 阶段的并行排序采用了 CUDA Thrust 库中的 sort 函数，该函数基于文献[6]的思想在 GPU 上实现基数排序，充分利用 shared memory，最小化 global memory 的非合并访问次数，比其他 GPU 排序加速 4 倍，相比在多核 CPU 上充分优化的 CPU 排序算法加速 23%，所以在算法 2 中排序耗费的时间非常少。并且，将 S 排序可以使第 3 步相邻线程所访问的元组的 Key 值在数值上相邻，这样，在〈$RKey$，Rid〉向量中进行二分查找时，相邻线程访问同一数据项集合 cache 的概率将大大增加，从而减少 merge 阶段的 cache 缺失率，所节省的时间可大于排序带来的开销。

4　基于 CUDA stream 的优化

在 GPU 程序中，主机端与 GPU 端通过 PCI-E 总线传输数据，而 PCI-E 带宽远小于显存和 GPU 片内存储器带宽，当 GPU 上实现的 Join 算法传输的数据较多时，PCI-E 总线带宽将成为瓶颈。并且在算法 1 中，memHotD 阶段和 Probe 阶段、memDtoH 阶段 3 者之间虽串行执行，但仍存在并行性。这时可以考虑数据传输与内核之间的并行执行。

本文采用 CUDA 的 stream 技术对算法 1 进行进一步优化。每个 stream 是一系列按顺序执行的操作，不同 stream 之间则是乱序执行或并行执行。首先使用 S 向量构建 hash 表，然后将 S 向量数据按 stream 的个数 n 平均分为 S_1，S_2，…，S_n 段，第 i 个 stream 负责第 i 段数据 S_i，stream 之间异步执行。其中，每个 stream 都可以分为以下 3 个子步骤：

Step1　将 S_i 段从主机端传输至 GPU 端；

Step2　S_i 段在内核中使用整个 *hash* 表进行探寻，得到相应的结果〈Sid，Rid〉向量段；

Step3　将〈Sid，Rid〉向量段从 GPU 端传输至主机端的结果向量中的相应位置。

如图 1 所示，每个 stream 的 Step2 可以与另一个 stream 的 Step1 或 Step3 重叠执行，隐藏大部分主机端与 GPU 端的数据传输时间，提高 GPU 中资源的利用效率。但是，每个 stream 所得到的结果数据段的每个位置并非都有效，因此在 CPU 端需要把每段结果的有效部分连接起来，这样会增加一些开销。同理，可将 stream 技术应用于 Sort-merge Join 算法，但 merge 阶段之前，必须将 S 的所有数据从主机端传输到 GPU 端进行排序，所以 memHtoD 阶段与 merge 阶段并不存在并行性，这样只能将 merge 阶段与 memDtoH 阶段重叠执行。

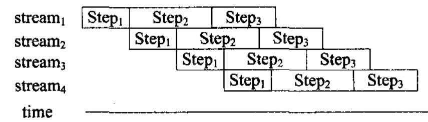


图 1　基于 CUDA stream 的算法 1 的执行流程

5　实验结果与分析

5.1　实验环境

本文采用 Intel Core i7 990x CPU 和 NVIDIA GTX480 显卡作为实验平台，主要的硬件参数如表 1 所列。实验程序在操作系统 Ubuntu 10.04 上运行，CUDA SDK 版本是 NVIDIA GPU Computing SDK 4.0。

表 1　实验平台主要参数

参数	GTX480	Intel Core i7 990x
核数	480	6
Cache	64kB 可配置 L1 cache 和 shared memory，768kB L2 cache	6 * 64kB L1 cache，6 * 256 kB L2 cache，12MB L3 cache
显存/主存容量	1.5GB	24GB
显存/主存带宽	177GB/s	32GB/s

基于文献[8，10]的方法，我们生成海量测试数据集，数据类型为整型。其中主键向量 R 包含 16M 个元组，每个元组取值唯一。外键向量 S 的元组个数分为 4M、8M、16M、32M 和 64M 5 种大小，这里的 $1M=10^6$。并且每种大小的外键数据集均满足均匀分布、低偏斜分布和高偏斜分布 3 种类型，分别标记为 Uniform、ZipfLow、ZipfHigh。在均匀分布中，每个元组以相等的概率出现在外键向量中。在数据偏斜分布中，元组出现的频率会以一定的程度递减。我们将低、高偏斜分布的偏斜值分别取为 $s=1.05$ 和 $s=1.25$。直观地看，当 $s=1.05$ 时，最常出现的值的频率为 8%，前 10 个最常出现的值的频率之和为 24%；当 $s=1.25$ 时，这两个频率值分别为 22% 和 52%。此外，连接操作的选择度也有所不同。假设外键向量 S 的元组总数为 n，能在主键向量 R 中连接成功的元组总数为 n'，则选择度为 n'/n。这里，选取的选择度分别为 100%、75%、50% 和 25%。下面将通过比较实验结果来分析外键偏斜度和选择度对性能的影响以及使用 CUDA stream 的效果，最后比较 GPU 和多核 CPU 上各种 Join 实现方法的性能。

5.2　外键偏斜度的影响

在测试外键偏斜度对算法性能的影响时，外键数据集按照 Uniform、ZipfLow、ZipfHigh 3 种情况分布，选择度为 100%。由于外键的分布主要对算法 1 的 probe 阶段和算法 2 的 sort、merge 阶段有影响，故本节给出在不同外键分布情况下这 3 个阶段的运行时间变化。

如图 2 所示，在数据量相同的情况下，外键的偏斜度越高，算法 1 的 probe 阶段运行的时间越少，并且时间降幅很大。我们知道，在数据均匀分布的情况下，连接操作的行为是

随机的，在处理完一个元组后立刻处理下一个元组，并不具备良好的时间局部性。但数据偏斜度越高，相邻线程处理的外键元组相同的概率越大，重复利用 hash 表中相同的数据集合的次数也越多，这样可以充分利用 Fermi 架构中新增的高速缓存，降低 Cache 缺失率，所以在偏斜度高的情况下，probe 阶段运行得更快。

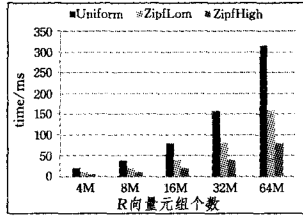


图 2　probe 阶段在不同外键分布下的运行时间

而算法 2 的 merge 阶段性能受外键偏斜度的影响不大，如图 3 所示，在数据偏斜度高的情况下，merge 阶段的运行时间有所减少，但降幅很小。这是由于算法 2 原本对外键向量和主键向量进行了排序，即使数据均匀分布时进行二分查找也具备良好的空间局部性，即相邻线程处理的外键元组虽然不同，但在数值上相邻，每次查找同一主键元组集合的概率却非常大，这样可以减少在 merge 阶段的 Cache 替换次数。在外键数据集偏斜度较高时，merge 阶段处理数据具备良好的时间局部性，但失去了原本的空间局部性，并且两者之间性能差异不大。

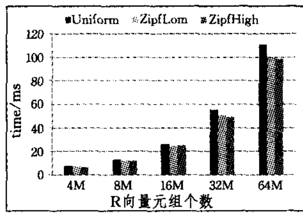


图 3　merge 阶段在不同外键分布下的运行时间

假设从 L1 Cache 向 L2 Cache 请求读的次数为 n，当 L2 Cache 读缺失的次数为 m，每次读的大小为 Cache 行大小时，L2 Cache 缺失率为 m/n。通过使用 NVIDIA 的 Compute Visual Profiler，测出算法 1 的 probe 阶段和算法 2 的 merge 阶段在不同的外键元组个数和不同的外键分布情况下的 n 值和 m 值，通过计算，得到 probe 阶段和 merge 阶段的 L2 cache 缺失率，如表 2 所列。由表 2 可以看出，probe 阶段的 L2 Cache 缺失率在外键均匀分布时约为 96%，在外键偏斜度较高时仅为 49%左右，而 merge 阶段的 L2 cache 缺失率受外键分布影响极小，一直稳定在 45%～65%，这正好与图 2 和图 3 所得到的结果相吻合。

表 2　两个阶段在不同的外键分布下的 L2 Cache 缺失率

外键元组个数	Uniform		ZipfLow		ZipfHgh	
	probe	merge	probe	merge	probe	merge
4M	96.3%	45.1%	63.8%	50.8%	48.8%	57.1%
8M	96.7%	53.3%	64.2%	50.4%	49.5%	55.9%
16M	96.7%	59.7%	64.0%	52.8%	49.7%	55.5%
32M	96.8%	62.5%	64.1%	57.2%	49.7%	56.5%
64M	96.6%	63.1%	64.5%	57.9%	49.3%	55.7%

在算法 2 的 sort 阶段对外键排序时，划分数据到每个 Block 的 shared memory 中进行排序，这样将 global memory 的随机分散读写转换成 shared memory 的分散读写，使用片上存储器对齐数据，无论数据偏斜与否，硬件的作用都可使数据具有良好的局部性，所以外键的分布对 sort 阶段的性能影响较小，如图 4 所示。

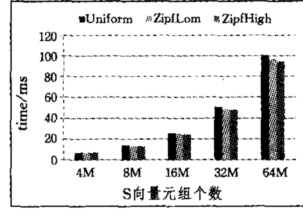
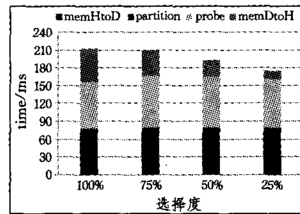


图 4　sort 阶段在不同外键分布情况下的运行时间

5.3　选择度的影响

本节测试算法 1 和算法 2 在不同选择度下各阶段性能的变化，这里选取的选择度分别为 100%、75%、50%和 25%，S 和 R 的元组个数均为 16M，数据均匀分布。

从图 5 和图 6 可以看出，算法 1 和算法 2 的 memDtoH 阶段的运行时间均随选择度的降低而减少。这是因为选择度较低时，连接得到的结果数相应地缩减，从显存传输到主存的结果数据量变小，传输的时间也就减少了。



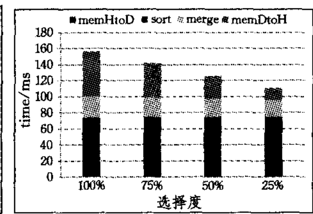


图 5　算法 1 各阶段在不同选择度下的运行时间　图 6　算法 2 各阶段在不同选择度下的运行时间

算法 1 的 partition 阶段和算法 2 的 sort 阶段基本不受选择度变化的影响。算法 1 的 probe 阶段的运行时间在选择度降低的情况下略微增加，算法 2 的 merge 阶段则相反。选择度降低时，由于产生的结果数减少，从 cache 到 global memory 缓冲区的写操作次数也减少，以致整个 probe 阶段或 merge 阶段的效率提升。然而，在 hash 表每个桶中探寻时，算法 1 找到匹配的元组后立即停止探寻，相反则直到扫描完整个 hash 桶为止。因此选择度为 100%时，算法 1 的 probe 阶段部分线程探寻元组时不必扫描完相应的整个 hash 桶，若选择度降低，意味着该部分线程需要扫描完整个 hash 桶，若这带来的开销大于写结果所节省的时间，算法 1 的 probe 阶段就会变慢。算法 2 的 merge 阶段使用的是二分查找法，每个线程查找一个外键元组时的平均比较次数不随选择度的降低而增加，依然是 $\log_2(N)$，这里 N 为 R 向量的元组个数。所以整体上 merge 阶段在选择度低的情况下运行时间反而有所减少。

5.4　CUDA Stream 的效果

图 7 和图 8 给出了算法 1 和算法 2 在不使用和使用 CUDA Stream 时的运行时间对比，分别标记为 no_stream 和 stream。其中，外键均匀分布，选择度为 100%。

由图 7 可以看出，算法 1 使用 CUDA stream 后，运行时间比不使用 CUDA stream 时平均减少 17%～20%，相当于 memDtoH 阶段时间的 60%～80%，这说明使用 stream 可以有效地隐藏大部分的数据传输时间，同时后面连接结果数据段的有效部分也耗费了小部分时间。图 8 同样说明算法 2 使

用 stream 后性能有所提升，只是提升的幅度较小，这主要因为算法 2 的 merge 阶段仅能与 memDtoH 阶段并行执行，而没有隐藏 memHtoD 阶段的执行时间，所以性能提升不大。

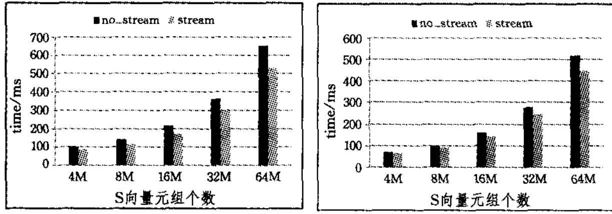


图 7 算法 1 使用与不使用 CUDA Stream 的运行时间对比　图 8 算法 2 使用与不使用 CUDA Stream 的运行时间对比

5.5 GPU 与多核 CPU 的性能比较

本文基于文献[3]的思想，在多核 CPU 上充分优化了 Hash Join 算法，主要实现方法是维护一个全局的共享 hash 表，并使用锁保护每个表项，多个线程同时构建和探寻 hash 表。另外，我们采用文献[6,8]中的方法实现了多核 CPU 的并行排序，并在此基础上实现基于多核 CPU 的 Sort-merge Join 算法。在两种算法的实现过程中，我们使用了原子锁、组预取指令、任务窃取等多种优化机制，并且最大限度地减少了 Cache 和 TLB 的缺失。

这样，本文针对 Hash Join 算法和 Sort Join 算法分别实现了基于 GPU 但没有使用 CUDA stream、基于 GPU 并且使用了 CUDA stream 以及基于多核 CPU 的 3 种优化方法，在图 9—图 12 中依次标记为 GHJ、GSJ、GHSJ、GSSJ、CHJ 和 CSJ，并给出了各种 Join 实现方法的性能对比。假设外键元组个数为 s，算法整体时间为 t，则执行速度为 s/t，单位为兆元组每秒，记为 MT/s。其中 GPU 算法的整体时间为内核时间与数据传输时间的总和，多核 CPU 算法的整体时间则为算法核心执行时间。

由图 9—图 11 可以看出，每种优化方法的执行效率均随外键数据量的增大而提升。这主要是因为要处理的数据越多，GPU 或多核 CPU 的资源利用越充分，执行速度也越快，直到处理器负载达到饱和。如图 9 所示，在外键均匀分布时，基于 GPU 并且使用 CUDA stream 的 Sort-merge Join 算法执行速度最快，在外键元组个数为 64M 时超过 140MT/s，相当于基于多核 CPU 的 Sort-merge Join 算法的 1.73 倍，后者是本文在外键均匀分布时基于多核 CPU 最快的一种实现方法。而当外键存在数据偏斜时，执行最快的是基于 GPU 并且使用 CUDA stream 的 Hash Join 算法，如图 10 和图 11 所示，其原因与 5.2 节所阐述的一致。当外键偏斜度较低或较高时，该算法的执行速度分别超过 170MT/s 和 210MT/s，最大达到基于多核 CPU 的 Hash Join 算法的 2.4 倍。

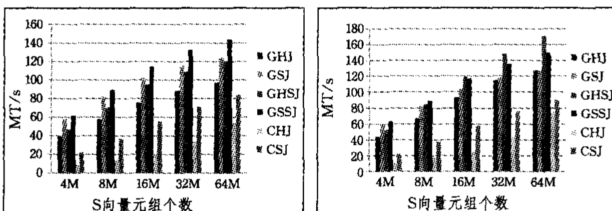


图 9 外键均匀分布时各种 Join 实现方法的性能比较　图 10 外键偏斜度较低时各种 Join 实现方法的性能比较

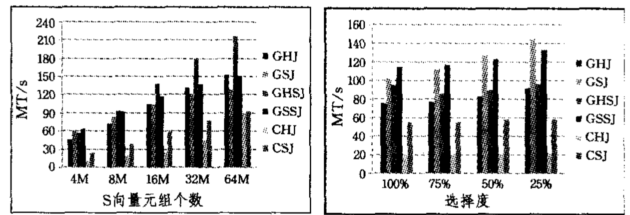


图 11 外键偏斜度较高时各种 Join 实现方法的性能比较　图 12 外键分布均匀时不同选择度下各种 Join 实现方法的性能比较

如图 12 所示，在外键均匀分布时，GHJ、GSJ、GSSJ、CHJ 和 CSJ 的执行效率均随选择度的降低而提升，而 GHSJ 则上下波动，并且基于 GPU 的 Hash Join 算法和 Sort-merge Join 算法均出现了使用 CUDA stream 时的实现效率比未使用 CUDA stream 时的实现效率低的情况。除了 5.3 节阐述的原因之外，还因为在选择度较低时，从显存向主存传输的结果数据量减少，使用 stream 隐藏的延迟变小，连接结果数据段的有效部分的开销却不变，这样算法整体的效率反而降低了。选择度低的情况下执行最快的是 GSJ，执行速度在选择度为 25%时达到 144MT/s。

结束语　Fermi 架构的出现，使列数据库的连接操作的实现更加简单高效。本文提出了基于 Fermi 架构的 Hash Join 算法和 Sort-merge Join 算法，并分析外键偏斜度、选择度等因素对算法性能的影响。实验表明，基于 Fermi 架构的 Sort-merge Join 算法执行效率比较稳定，Hash Join 算法则在外键偏斜时性能更佳，这在很大程度上得益于 Fermi 架构中全新的缓存结构。通过使用 CUDA stream，在选择度较高时可以有效地隐藏这两种算法数据传输带来的延迟。并且，与在多核 CPU 充分优化的 Join 算法相比，基于 Fermi 架构的 Join 算法最大获得 2.4 倍的加速比，执行效率在外键分布高偏斜时可达到 217MT/s。

参考文献

[1] He B, Lu M, Yang K, et al. Relational Query Coprocessing on Graphics Processors [J]. ACM Transactions on Database Systems, 2009, 34(4): 23-32

[2] Kim C, Kaldewey T, Lee V W, et al. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-core CPUs[C]// Proceedings of the VLDB Endowment. US: VLDB Endowment, 2009: 1378-1389

[3] Blanas S, Li Y, Patel J M. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs[C]// Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data. NY USA: ACM, 2011: 37-48

[4] 陈虎，欧彦麟，陈海波. 面向多核处理器平台的 Hash Join 算法设计与实现[J]. 计算机研究与发展，2010，47(z1)：171-175

[5] 韩希先，杨东华，李建中. DBCC-Join：一种新的高速缓存敏感的磁盘连接算法[J]. 计算机学报，2010，33(8)：1500-1511

[6] Satish N, Kim C, Chhugani J, et al. Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort[C]// Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data. NY USA: ACM, 2010: 351-362

[7] Satish N, Harris M, Garland M. Designing Efficient Sorting Al-

gorithms for Manycore GPUs[C]// Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium. US: IEEE Computer Society, 2009: 1-10

[8] Chhugani J, Nguyen A D, Lee, V W, et al. Efficient Implementation of Sorting on Multi-core SIMD CPU Architecture[C]// Proceedings of the VLDB Endowment. US: VLDB Endowment, 2008: 1313-1324

[9] Senggupta P, Harris P, Zhang P, et al. Scan Primitives for GPU Computing[C]// Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics hardware. Switzerland: Eurographics Association, 2007: 97-106

[10] Gray J, Sundaresan P, Englert S, et al. Quickly Generating Billion-record Synthetic Databases[C]// Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data. NY USA: ACM Press, 1994: 243-252

[11] NVIDIA Corp. NVIDIA's Next Generation CUDA™ Compute Architecture: Fermi™ [EB/OL]. http://www.nvidia.com, 2009-05-15

[12] NVIDIA Corp. Tuning CUDA Applications for Fermi[EB/OL]. http://developer.nvidia.com, 2011-05-15

(上接第 61 页)

格划分步长的 2 倍，即 $H=2h$；最细一级网格是原始输入图像的尺寸。

图 5、图 6 示出用本文所提出的并行多重网格算法求解泊松方程实现的梯度域图像拼接结果。图像拼接前先采用文献[7]的配准算法对所有图像进行配准。图 5 用 5 张 1080×720 的图像拼接生成 8100×3680 的全景图，图 6 用 5 张 1024×768的图像拼接生成 7963×3580 的全景图。并行多重网格算法在图 5、图 6 实验中的内存占用、内存/外存通讯量、迭代次数和运算时间如表 2 所列。

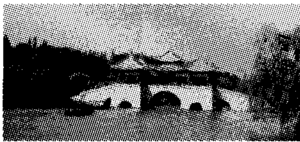


图 5　图像拼接实验 1(数据 1)

图 6　图像拼接实验 2(数据 2)

表 2　不同计算方法比较

算法		内存占用(兆)	I/O通讯(兆)	迭代次数	时间(秒)
超松弛迭代 SOR	数据 1	56	7.53	676	9.656
	数据 2	51	7.3	615	8.433
伽可比迭代	数据 1	56	7.52	1365	19.499
	数据 2	51	7.34	1127	15.451
多重网格	数据 1	224	5.95	232	3.315
	数据 2	204	5.53	209	2.866
并行多重网格	数据 1	184	3.53	165	2.357
	数据 2	180	3.36	160	2.194

从表 2 可知，并行多重网格算法在内存占用方面低于传统多重网格算法，在内存/外存通讯量、运算时间及迭代次数等方面都明显优于超松弛迭代、伽可比迭代和传统多重网格算法。

结束语　泊松方程是工程中常用的偏微分方程，其快速、精确的求解对于问题解决非常重要。基于泊松方程的梯度域图像编辑是目前图像处理和图形学的研究热点，大尺度图像的梯度域编辑操作需要求解超大规模未知数的泊松方程，传统多重网格算法在 PC 机上运算效率低。提出了一种面向大尺度图像梯度域编辑的并行多重网格求解泊松方程的算法，利用多重网格的迭代、约束和插值操作之间的内存局部性和更新相关性并以并行方式运行多重网格算法。该方法能够在 PC 机上高效地实现大尺度图像的梯度域编辑操作。

传统多重网格数值求解算法单独完成迭代、约束和插值操作，没有很好地利用内存数据的局部性，算法存在改善空间。文中所提出的泊松方程并行多重网格求解算法充分利用内存空间数据的局部性和更新相关性来并行完成迭代、约束和插值操作，提高了泊松方程的求解效率。

所提出的并行多重网格求解算法适合于求解纽曼边界条件、结构化数据的泊松方程。扩展本文工作，使之适合大尺度图像的复杂梯度域编辑操作，即适合求解狄雷克利边界的泊松方程，是未来的工作方向。

参考文献

[1] Pérez P, Gangnet M, Blake A. Poisson Image Editing[J]. ACM Transactions On Graphics, 2003, 22(3): 313-318

[2] Levin A, Zomet A, Peleg S. Seamless Image Stitching in the Gradient Domain[C]// Tomas P, Jiri M, eds. Proceedings of 8th European Conference on Computer Vision (ECCV'2004). Springer Verlag Publishing House, 2004, 1: 377-389

[3] Agarwala A, Dontacheva M, Agarwala M, et al. Interactive Digital Photomontage[J]. ACM Transaction on Graphics, 2004, 23(3): 294-302

[4] Press W H, Teukolsky S A, Vetterling W T, et al. Numerical Recipes in C[M]. US: Cambridge University Press, 2002: 871

[5] Kazhdan M, Hoppe H. Streaming Multigrid for Gradient-Domain Operations on Large Images[J]. ACM Transaction on Graphics, 2008, 27(3)

[6] Chow E, Falgout R D, Hu J J, et al. A Survey of Parallelization Techniques for Multigrid Solvers[C]// Frontiers of Parallel Processing for Scientific Computing. US: the Society for Industrial and Applied Mathematics, 2005

[7] Szeliski R. Image Alignment and Stitching: A Tutorial[J]. Foundations and Trends in Computer Graphics and Computer Vision, 2006, 2(1): 1-104

[8] 廖臣，祝大军，刘盛纲. 五点差分格式求解泊松方程并行算法的研究[J]. 电子科技大学学报, 2008, 37(1): 81-83