中国科学技术大学

学士学位论文

一种关于数据立方体的相关并行算法模型及优化方案

作者姓名：　　邹凯

学科专业：　　计算机专业

导师姓名：　　谢希科 教授

完成时间：二〇一七年六月

University of Science and Technology of China

A dissertation for bachelor's degree

A parallel approach and its optimization for data cubes

Author's Name: Kai Zou

Speciality: Computer Science

Supervisor: Prof. Xike Xie

Finished Time: June, 2017

致　谢

自接触计算机领域以来，我就一直对高性能计算领域抱有浓厚的兴趣。每当看到一个思路敏捷的算法，抑或是一个精妙的实现，我都会认真将其记录下来，在之后细细品读。无论是从算法本身的设计上，还是从如何挖掘新兴的软硬件设施的潜力上，总会有一些厉害的前辈，或者是同辈，甚至可能是后辈们，在不经意间，给予我极大的灵光。

我想，所谓计算机科学的艺术，大概就是这样的吧。

本科的毕业论文，以这个方面的相关研究作为主题，也算是对本科数年，自己寻找的前进方向的一个答复吧。

在论文成文的过程中，我得到了我的导师，中国科学技术大学的谢希科教授的大力支持与帮助。无数次的方向研讨 meeting，无数次的交流，这每一分每一秒都包含着老师的心血，包含着老师的智慧。

而说起为何踏进数据库这个领域，选择在数据库这个领域走高性能研究的路子，那是因为有两位至关重要的前辈，充当了我在这个领域的领路人。一位是谢希科教授，另一位则是新加坡国立大学的何炳胜副教授。这两位杰出的前辈，一同向我展示了数据库领域的博大精深，更重要的是，一同向我展现了，一个合格的学术研究者应有的风貌。

同时，我也要感谢在新加坡国立大学实习时的诸位前辈们。是优秀的你们给予了我奋发向上，与你们并肩作战的动力。

我还要感谢无时无刻不在给予我无微不至的关怀的家人们，以及愿意与我分享生活中点点滴滴的喜怒哀乐的朋友们。是你们，让我的生活变得更加安逸舒适，变得更加多姿多彩。

最后，请容许我用一句话，结束这冗长无味但是真挚的致谢，同时拉开新生活的帷幕

——而那过去了的，就会成为亲切的怀恋

目 录

摘　要

时至今日，得益于并行计算领域取得的包括通用并行计算平台编程（例如GPGPU），新兴的的并行硬件（例如 FPGA）的开发等巨大成就，许多的应用程序抑或是系统，都采用了并行计算的方式来优化他们的性能，这其中也包括了数据库系统。前人在相关领域的工作中，有许多令人瞩目的成果都集中在基于并行计算的查询优化上，例如负载均衡，流水线／并发查询处理模型，等等。得益于这些工作，许多的计算密集型任务的性能来看得到了提升。但是，只有少数的工作将注意力放在数据立方体，以及与之相对应的 OLAP 计算上。而包括数据立方体生成在内的许多数据立方体相关任务，都属于计算密集型任务，很适合使用并行计算方式来进行优化。

为了解决上述问题，本文提出了一套并行计算模型的方案来提高数据立方体任务，例如底层 cuboid 的生成，从 cuboid 到 cuboid 的计算（用以避免每次从原始数据集计算 cuboid 的繁重计算）等的执行效率。在我们的工作中，我们通过将已有的算法根据并行平台的特点重新定制，改进了现存的数据立方体的优化算法。在此基础上，我们还提出了一套新的评估函数，以及对应于这套评估函数的优化方案，使其能够适应于不同的硬件配置与不同的计算任务特性。实验结果表明我们的并行计算方案比起同等情况下的非并行计算方案普遍有了 2-4 倍性能的提升。在此基础之上，我们提出的优化技术又使得原并行算法在数据立方体的计算中加速了 15-30%。

关键词：中国科学技术大学　学位论文　学士　OpenCL　在线分析处理　并行计算

Abstract

Nowadays many applications / systems, including databases, have been used parallel computing technique to optimize their performance because of the great achievements of the parallel computing field such as GPGPU libraries, emerging hardware such as FPGA, GPU and so on. There are many remarkable works which centered on query processing with parallel computing such as workload balancing, pipeline / concurrent query processing model. With their help, as a result, many computation intensive tasks' performance have been improved. However, few of them focus on data cube as well as their corresponding OLAP computation. Actually many data cube-related computation tasks, including data cube generation, are computation intensive that we can use some approaches of parallel computing to optimize them.

To address the problem, this paper provides a parallel computation model to improve the efficiency of data cube tasks such as bottom cuboid generation, inter-cuboid calculation (used to avoid heavy computation which have to face when generating cuboid directly from raw dataset). In our work, we improve existing data cube optimization approaches by customizing them to the parallel settings. Based on that, we provide a new evaluation function and corresponding optimizations, which are adaptable to different hardware configurations and different natures of computational tasks. Experiments show that our prototype is 2-4x better than its non-parallelized counterpart, and our proposed optimization techniques further accelerate the data cube computations efficiency by 15-30%.

Key Words: University of Science and Technology of China (USTC), Thesis, Bachelor, OpenCL, OLAP, Parallel computing

图目录

表目录

算法索引

第一章　简介

在现代计算机系统及其应用中，数据量的增大已经成为了一个必然的趋势。伴随其而来的计算量的增加，使得越来越多高性能相关的新兴硬件／软件技术的出现成为势在必行。

在高性能计算领域之中，并行计算已经成为了一个很重要的工具。各种新兴硬件的出现，以及伴随它们而诞生的各种并行编程模型，都为高性能计算领域的向前发展提供了十足的力量。

而随着数据规模的增大，对于大量数据的管理、分析也在逐渐成为一项重要的需求，不仅仅是因为本身，更是因为这个领域所做的工作将是所有未来计算机领域的基石。其中作为代表的数据库领域，近些年来不断致力于发展新的数据分析技术，改进数据关联的基本操作与算法，以及更多地尝试新的硬件平台，层次结构，在这些方面都取得了长足的进展。

进入 21 世纪，人们也开始在各种方面开始尝试将这两者结合在一起，但是由于现行并行计算编程（譬如大计算量，无相关性，弱控制）与传统数据库领域（譬如 ACID 特性）在某些理念上的冲突，使得在数据库的并行优化方面，还有很多可以开展的工作没有得到人们的充分认识。

在这些工作中，有一个领域是比较重要的：On-Line Analytical Processing(OLAP, 线上分析处理）这个领域的计算中，最主要的部分——数据立方体生成，要求对于用户所需求的数据进行归类、聚合，并且在可控的时间内返回给用户。在短暂的时间内，处理庞大的数据集合，进行复杂的聚合运算，使得这部分成为了一个计算密集的部分。因此，对于这些计算部分，乃至于整个数据立方体聚合操作的性能的优化势在必行。

现有的工作成果中，虽然有很多利用并行计算的方式与特点，对于数据库系统中的其他方面进行优化，从而间接的使得包括数据立方体生成在内的 OLAP 相关计算的性能有了一定程度的提升，但是，却鲜有工作直接对数据聚合立方体的生成进行优化。

基于上述背景，本文尝试从该领域出发，探寻利用并行计算硬件与编程思想，来尝试从源数据生成数据立方体，以及从数据立方体出发如何生成别的数据立方体这两方面进行优化。本文的贡献可以从以下几个角度来阐述：首先，本文给出了一套基本的计算代价估计函数，并在这个基础上提出了生成最底层 cuboid

的时间／空间最优算法，以及结合它们的优点，提出了平衡各自缺点的混合型方法；其次，本文给出了一套如何评估 cuboid 生成的时间的算法，（并且基于并行计算的基础改进了现有的生成预处理 cuboid 的方法）。第三，上述的算法均基于 OpenCL 库实现，为其在多种不同平台间的移植提供了便利。

本文接下来将按照如下方式组织：第 2 章介绍已有的工作成果与本研究开展的动机，第 3 章介绍用并行计算从源数据生成底层 cuboid 的数种方式并分析其优劣，第 4 章给出一套从源 cuboid 到目标 cuboid 的生成方案及其优化，第 5 章对这个工作进行了评估，第 6 章将总结本文。

第二章　背景工作与研究动机

第一节　Parallel-Database 相关

并行计算是一个发展历史悠久的领域，其最早的发源要追寻到上世纪五十年代后期。上世纪六十年代至七十年代，共享内存的多处理器体系结构出现了。在上世纪八十年代，大规模并行处理器的出现开始占领市场，然而旋即又被集群所取代。发展至今天，并行计算硬件的主流已经成为了多核处理器[1]。

在这个主流下，越来越多的硬件被发掘出来参与到并行计算中来。图形处理器（GPU）最初是被开发来实现以电子显示为主的各方面的图形演算与生成的，然而由于其架构满足流水线深，可利用核心多等特点，很快人们就开始研究 GPU 的通用用途（GPGPU），其中一个很重要的领域就是 GPU 参与的并行计算。2006 年，NVIDIA 首先提出了 CUDA，并在 2007 年首次正式将其发布，这也成为了世界上首个在 GPU 上进行通用并行编程的平台[2][3]。在不久之后的 2009 年，苹果公司也由 Khronos Group 进行开发，提出了自己的一套通用并行计算平台 OpenCL[4]。这一套开发平台不仅仅局限于对 GPU 的并行潜能的挖掘，而是在此基础上支持了更多的新兴硬件，例如 FPGA，DSP 等。并行计算在这些优秀平台的支持下开始在各个领域蓬勃发展。

鉴于并行计算其尤其适合对于大量数据的高密度计算的特征，在数据库相关领域，近些年来也有研究者开始了试图将数据库与并行计算硬件（主要是 GPU）结合在一起的研究。B. He 等人所在的研究组作为最早涉足该领域的研究组之一，在 2008-2009 年之间出版了相关论文对于关系型数据库在 GPU 上的可能性进行了探讨。其在 2008 年发表的论文[5] 中提到了在 GPU 上进行的对于关系型数据库 join 操作的并行优化，而在另一篇 2009 年的论文中[6] 最早地提出了一套系统的 GPU 上评估查询执行效率的模型，还有一套完整的在 GPU 上进行的查询计划的生成方案。这两项研究成果至今仍被相关研究领域的人员频繁引用。另外，2011 年发表的另一篇论文[7] 中也对传统关系型数据库中的事务处理进行了适合并行架构上的优化。

在接下来的几年，有许多的研究成果基于前述的成果，在并行计算和数据库的结合方面做出了很多的新的尝试。这其中有很多的工作集中在了对于查询的高效处理的方面。俄亥俄州立大学的研究小组在 2014 年的论文[8] 中首次给出了

一种对于并发处理查询的解决方案，改变了从前至多只能并行处理一个委托的不同部分，从而使得总体硬件资源利用效率低下的状况，使得查询之间可以有机会被并行处理，极大地提高了硬件的利用率，提高了查询的吞吐量。南洋理工大学的另一个研究小组[9] 则尝试从另一个角度挖掘并行硬件的架构特点，通过开发一套对于多个查询的流水线处理系统以及与之相协调的并发内核控制和数据交换方法，来使得并行硬件在单位时间内处理多个查询的性能得到了大幅度的提升。另外，本文也尝试着给出了一套查询处理流水线化的分析评估模型。

总的来说，近些年来在数据库领域，并行计算作为一个新生并且强有力的事物，正在越来越多的方面发挥着重要的作用。

第二节　OLAP 相关

在数据库领域，一个历史悠久的课题就是，对于大规模数据仓库，如何进行有效的管理，分析等一系列数据相关的操作。在这之中不得不提到的就是 OLAP（On-Line Analytical Processing，线上分析处理）。这项技术将大量的原始数据聚合到特定的数据立方体中，在数据立方体中进行各种数据的分析，统计等等操作将比在原始数据集合中来得更加方便，快捷[10]。

在 OLAP 的基本概念中，数据立方体的构建是一项很基本的操作。其通过将海量的原始数据，按照它们之中的一些维度数据来进行划分，并且在此基础上进行聚合操作，从而得到一系列的分类聚合数据集合，用户可以在这些分类的数据聚合上更容易地寻找到自己所需要的对于原始数据集的整体，抑或是各个部分的概况。

在一个数据立方体（下文称 cube）中，存在着许多种不同的聚合：它们都是对原始数据集的聚合，但是它们选择的聚合的维度有所不同。这样的聚合之中的某一个特定的聚合被称作 cuboid。而在一个 cuboid 中，存在着许多 cell，每一个 cell 中存放着在这个 cuboid 在给定的聚合方式（例如 sum，count，histogram 等）前提下，对同一类数据的聚合。这里的同类，指的是未被聚合的所有维度属性值都相同的那些数据。

对于 OLAP 的研究，同样开始于很早以前。1997 年的一篇论文[11] 很早地从各个方面对于 OLAP 技术进行了一个全面的调研，并且指出了 OLAP 技术将来具体发展的形式。斯坦福大学的研究小组[12] 则算是最早的具体研究如何高效快捷的生成 OLAP 的核心——数据立方体的论文之一。这篇论文首次提到“数据立

方体中的 cuboid 并非每次都需要从原始数据生成”，并且基于这个观点给出了一套行之有效的预处理 cuboid 生成的评估方案，以及在此评估方案下的一套次最优的贪心算法。本文的其中一部分工作将基于他们的工作进行扩展与延伸。

近些年来，随着各种更新的系统架构，诸如分布式集群等的出现，OLAP 从最开始的单机处理单机分析开始，渐渐打开了新的发展方向。2009 年的一项研究成果[13] 则具有远见卓识地基于 in-memory database 的视点，对 OLAP，以及 OLTP，在它们运行的基本存储架构上进行了一系列的优化和创新，这也使得因此而诞生的 SAP HANA[14] 很快成为了包括 in-memory database 在内，目前世界上做的最优秀的 in-memory platform 之一。也有很多的文章着眼于并行计算的视点来考虑如何进行数据仓库相关一些操作的优化。俄亥俄州立大学的相关研究小组在 2013 年的论文中[15]，系统地给出了一套评估用并行硬件优化数据仓库相关操作的性能的标准。同一时期也有一系列文章则从不同的方面考虑了对于并行架构中的 OLAP 相关操作的优化。其中[16] 通过适当的控制在 cache 中的查询的调度从而使得 CPU-GPU 架构之间的通信与运行效率提高了，从而使与之相关的 OLAP 相关操作性能也得到了提升。另一个最近的研究成果[17] 则通过调整 query 候选，以及 query 中间结果在各个并行硬件单元上的分配，与[6] 提出的评估模型不同，放弃了评估，转而采取即时地解决这些问题，从而使得其在 OLAP 的一些相关问题的处理上有了性能上的提升。

综上所述，OLAP 是一个由来已久的问题，早些年的研究多是基于 OLAP 算法本身，而近些年来的研究更多专注于如何将其与现代的新兴系统架构，新兴硬件等等结合在一起，来让 OLAP 在现代技术的支持下展现出全新的活力。

第三节　研究动机

上述的研究背景中，我们可以发现：更多的研究要么专注于如何运用并行计算平台对并发／并行查询执行进行优化，要么专注于为在新的系统架构上部署高效的 OLAP 算法而设计具有针对性的软硬件结构的优化，似乎并没有一项研究来针对如何具体的运用并行架构体系来优化 OLAP 中数据立方体的生成算法本身。因此本研究就尝试着从这个领域着手，对这一问题提出自己的见解。

第三章　并行化的数据聚合立方体生成

第一节　一些数学量的符号化设定

本章在接下来的讨论中会涉及到一些计算，其中有一些时间，以及空间的量需要在此列出，以方便计算与讨论。这些量包括：

在系统中并行执行的并行 kernel 数量：K

数据具有 n 个维度属性，其中每个维度的取值数量由（$d_1, d_2, \cdots, d_n$）决定

在数据聚合立方体中一个 cell 的大小：s

一个 kernel 负责生成的 cell 个数：g

数据集的元素个数：d

数据集中一个元素的大小：sd

内存到设备（设备到内存）间拷贝数据集中一个元素所需要的时间：t_{ele_copy}

一个 kernel 处理一个数据集中的 element（包括了扫描，计算，以及将其结果放置到设备内存中的某个位置）所需要的时间：t_{scan}

内存到设备（设备到内存）间拷贝一个 cell 所需要的时间：t_{copy}

将一个已经生成的 cell 聚合进另一个数据聚合立方中所需要的时间：t_{agg}

数据集被划分成的部分数：p

第二节　内存模型

一、并行编程模型与多线程模型的相异点

目前在并行编程模型中常使用的库中，大多拥有“kernel”这一概念，即并行算法执行的单元。所有的并行算法以 kernel 为基础，通过并行硬件对于多个 kernel 的同时计算从而达到并行计算的目的

kernel 比起通常的多线程模型，有如下的相异点：

1、kernel 属于元程序，即 kernel 的执行是不可分割的　⇒　这使得 kernel 在执行之中属于不可控状态：kernel 直到执行完成为止都无法被中止

2、现有的编程模型中，并没有 kernel 对于临界区数据的锁机制　⇒　从而使得 kernel 之间对同一内存区域的协同访问／修改变得困难

3、在任意一个时刻，并行硬件上的一个计算单元中最多存在一个运行中的

kernel ⇒ 从而使得对于 kernel 的任务调度变得更为死板，一旦完成 kernel 的分配，则在运行的过程之中无法再次通过调度来重新均衡负载。

4、然而实际上，由于牺牲了控制相关方面的电路模块，使得并行设备在计算单元的量的容纳上得到了巨大的提升：即通过牺牲控制相关方面的许多性能，使得并行设备在能够同时执行的任务的数量上有了极大的提升。每个计算单元都有能力独自处理一个计算 kernel，而无需关心其余 kernel 的运行情况（事实上也很难关心）。于是，在 kernel 的执行过程中，许多无谓的通信成本，以及同步成本就可以被省去。基于这个特点，如果用户所需要完成的任务是计算密集型而非控制密集型的，那么并行计算模型将是一个很好的选择。

二、基于读写互不冲突原则的内存分配方案

由前所述，在以 kernel 为基础的并行模型中，对共享数据的内存管理是一件不容易的事情，特别当面对读写冲突（例如 RAW）时会变得难以控制。因此，直观的一种解决思路（同时也是现在比较常用的解决思路）便是事先划分每一个 kernel 所使用的内存区域，这样就可以在不造成读写冲突的情况下最大化并行 kernel 之间的并行度。

在下文中，将提到三种不同的由源数据生成底层 cuboid 的方案。这三种方案采用的内存模型都基于上述的读写互不冲突原则，并且将会在方案被具体展示时被具体提出。

第三节 memory 最优方案

首先，对于并行方案的设计，一个最直观的想法就是将计算 kernel 以 cell 作为划分的基准，即每个 kernel 负责计算某些需要生成的数据立方体中的 cell，并且最后设法将所有 kernel 的计算结果统一到一起，即可得到一个完整的 cuboid。

在这种方式的前提下，我们的编程模型生成一系列 kernel，注意这些 kernel 的数量并非一个系统中能够同时运行的 kernel 的数量（后文也将反复强调这个概念）。而后，这些 kernel 在计算时满足以下的几个特点：

· 所有 kernel 共用一个 shared memory 数据聚合立方体，这个数据立方体在所有的 kernel 计算完内部的所有 cell，并且将结果拷贝到该区域之后，即为所求的 cuboid

· 每个 kernel 仅操作被调度方式分配到的那些 cells

下面给出算法运行的步骤：

```
Data: 原始数据集
Result: 对原始数据集生成的 cuboid
1 Generate computation kernel based on system dispatch scenario;
2 Copy the dataset into the parallel device;
3 while kernels to execute exists do
4     Use this kernel to scan the whole dataset;
5     Generate the corresponding cells using the result of scan;
6     Copy the cells to corresponding part in shared memory;
7 end
8 Copy the result cuboid back to the main memory for further use;
```

算法 3.1: 以 cell 为组织形式的并行生成 cuboid 算法

下面我们对这个算法的时空状况进行分析：

第 1 行中，初始化一个并行计算的设备所需要花费的时间是一个与系统相关，并且与具体程序无关的常量 C_T，当然所占用的空间也应该是一个常量 C_S。因此，这一步花费的总时间为：

$$T_{initialize} = C_T \tag{3.1}$$

到这一步为止系统出现过的最大总存储占用量为：

$$S_{initialize} = C_S \tag{3.2}$$

第 2 行中，为了使得并行设备也能够使用 dataset 进行计算，同时为了使得并行计算设备，以及主存都有足够的空间容纳将要计算的 cuboid，因此无论是 dataset 还是 cuboid 的空间都需要准备两份。因此，可以得到这一步花费的总时间为：

$$T_{copy\_in} = t_{copy} \times \prod_{i}^{n} d_i + d \times t_{ele\_copy} \tag{3.3}$$

到这一步为止系统出现过的最大总存储占用量为：

$$S_{copy\_in} = C_S + (d \times sd + \prod_{i}^{n} d_i \times s) \times 2 \tag{3.4}$$

接下来的循环节部分，我们可以算出，在这种前提下系统一共会生成的计算 kernel 数为 $\prod_i^n d_i \div g$，而如同前面的假设，系统在同一时间（同一趟）最多能运行 K 个 kernel。因此系统总共需要运行 $\prod_i^n d_i \div (g \times K)$ 趟才能将这些内核处理完毕。而每一趟中，每个 kernel 都需要为了处理一整个数据集花费时间为 $d \times t_{scan}$。另外观察发现，这一步并没有进行更多的内存分配。于是这个阶段花费的总时间为：

$$T_{cal} = d \times t_{scan} \times \prod_i^n d_i \div (g \times K) \tag{3.5}$$

到这一步为止系统出现过的最大总存储占用量和上一步相同。

最后的拷出部分，显然也没有任何更多的内存分配。所需要的时间为：

$$T_{copy\_out} = \prod_i^n d_i \times t_{copy} \tag{3.6}$$

而整个算法的运行时间为：

$$T_{total\_c} = T_{initialize} + T_{copy\_in} + T_{cal} + T_{copy\_out} \tag{3.7}$$

因此，整个算法的运行时间为：

$$T_{total\_c} = C_T + 2 \times t_{copy} \times \prod_i^n d_i + d \times t_{ele\_copy} + d \times t_{scan} \times \prod_i^n d_i \div (g \times K) \tag{3.8}$$

过程中最大的空间占用为：

$$S_{max_c} = C_S + (d \times sd + \prod_i^n d_i \times s) \times 2 \tag{3.9}$$

注意到此处的 T_{total_c} 中，由于起码需要完成一趟 kernel 的运行，因此必定有 $\prod_i^n d_i \geq g \times K$

从这个结果我们可以看出来，该方案占用的内存仅为数据集所需要占用的空间与生成的 cuboid 所需要占用的空间之和，这是理论上的最小情况。因此我们可以做出结论，该方案是 memory 最优的。

但是另一方面，由于每个 kernel 都需要完整地扫描一遍数据集，这样在累计上，该方案总共需要的扫描与计算量为 $\prod_i^n d_i \div g$，当数据的统计维度较多，生成的 cuboid 比较大的时候，这将是一个非常庞大的数字，在时间上并不现实。因此，我们需要寻找另外的方案。

第四节　速度最优方案

通过上面的讨论，我们可以看到，如何有效地减少对于数据集的扫描次数，即是等同于有效地减少生成 cuboid 所需的总计算量。因此，为了减少用户所需的等待时间，我们必须需要一个合适的方案，使得在能够生成 cuboid 的同时尽可能少的扫描（处理）整个数据集。

考察如下情形：当我们不使用并行编程手段时，我们仅需扫描整个数据集一次，然后对于每个扫描到的元素，我们将其信息加以处理后，放入在内存中某个分配给 cuboid 的内存空间中。当整个数据集扫描完毕后，我们就得到了我们想要的结果 cuboid。考虑到绝大多数基本聚合函数，例如求和，求平均，计数，直方图累积等等，其实现并不依赖于数据集的内在构成。因此，我们自然可以将整个数据集按某种方式分成均等的若干份，然后每一份子数据集就由一个 kernel 来处理。

在这个思路下，我们能得到一个最直观的做法。在这个做法中的 kernel 满足以下几个特点：

· 所有 kernel 独自使用一个独立的 shared memory 区域（类同于 local memory）

· 每个 kernel 负责生成一个 part 的 dataset 的完整数据聚合立方体

这样，当所有的 kernel 执行完之后，我们就能得到这样一些 cuboid：它们是原 dataset 的不同子 dataset 的完整的 cuboid。然后，我们将每个 kernel 生成的 cuboid 再聚合到一个 cuboid 中，就能得到一个属于原数据集的完整的 cuboid。我们的算法就完成了。

下面给出算法运行的具体步骤：

因为在该方案中，数据被划分的块数，也就是 kernel 数变得可控了。因此我们不妨假定此时算法调度将生成 kernel 数为 $k_0 \geq K$。类同上一个算法，下面我们也对这个算法的时空状况进行分析：

第 1 行中，分析同前一个算法，亦有：

$$T_{initialize} = C_T \tag{3.10}$$

$$S_{initialize} = C_S \tag{3.11}$$

Data: 原始数据集
Result: 对原始数据集生成的 cuboid
1 Generate computation kernel based on system dispatch scenario;
2 Copy the dataset into the parallel device;
3 **while** *kernels to execute exist* **do**
4 Use this kernel to scan the specific part of raw dataset;
5 Generate the completed cuboid of this part using the result of scan;
6 Remain the cuboid into its part of device memory;
7 **end**
8 Copy all the sub-cuboids back to the main memory as well as aggregate them into whole cuboid;

算法 3.2: 以 dataset part 为组织形式的并行生成 cuboid 算法

第 2 行中，数据集作为并行设备运行算法时必须读取的部分，我们仍然要将其拷贝一份到设备上。不同的是，这一回每一个 kernel 都需要自己的一份完整的 cuboid 空间来存放其临时结果。因此能得到的该步骤所需要花费的时间为：

$$T_{copy\_in} = k_0 \times t_{copy} \times \prod_{i}^{n} d_i + d \times t_{ele\_copy} \tag{3.12}$$

到这一步为止系统出现过的最大总存储占用量为：

$$S_{copy\_in} = C_S + d \times sd \times 2 + \prod_{i}^{n} d_i \times s \times (k_0 + 1) \tag{3.13}$$

循环节部分，可以发现此时每个 kernel 需要扫描的 dataset 的一部分的大小均为 $d \div k_0$，因此每个 kernel 的运行时间将为 $d \div k_0 \times t_{scan}$。而 k_0 个 kernel 需要执行 $\lceil k_0 \div K \rceil$ 批次，因此最后我们能够得到该步骤所需要花费的时间为：

$$T_{cal} = d \times t_{scan} \times \lceil k_0 \div K \rceil \div k_0 \tag{3.14}$$

同之前一样，这一步没有分配额外的空间

拷出所有的子 cuboid 并且将其聚合成一个 cuboid 的部分，也并没有更多的空间分配。而花费的时间，则主要由两部分组成：将 k_0 个 cuboid 拷贝回主存，以及每拷贝出一个子 cuboid，就将其合并到已经存在于主存之中的完整 cuboid 中

去。因此这部分需要花费的时间为：

$$T_{copy\_out} = \prod_{i}^{n} d_i \times t_{copy} \times k_0 \tag{3.15}$$

$$T_{agg} = \prod_{i}^{n} d_i \times t_{agg} \times k_0 \tag{3.16}$$

而整个算法的运行时间为：

$$T_{total\_d} = T_{initialize} + T_{copy\_in} + T_{cal} + T_{copy\_out} + T_{agg} \tag{3.17}$$

因此，整个算法的运行时间为：

$$T_{total\_d} = C_T + 2 \times k_0 \times t_{copy} \times \prod_{i}^{n} d_i + d \times t_{ele\_copy} + d \times t_{scan} \times \lceil k_0 \div K \rceil \div k_0 + \prod_{i}^{n} d_i \times t_{agg} \times k_0 \tag{3.18}$$

而过程中的最大总空间占用量为：

$$S_{max\_d} = C_S + d \times sd \times 2 + \prod_{i}^{n} d_i \times s \times (k_0 + 1) \tag{3.19}$$

接下来首先考察 k_0 的取值。显然在几乎所有的等式中，k_0 均作为乘积项出现，因此在这些项中，显然是 k_0 具备越小的值越佳。在 T_{cal} 项中，当 k_0 为 K 的整数倍时，该式取得最小值。结合上述两个条件，我们可以令 $k_0 = K$，此时我们就能取得最优解。表达式可以改写为以下形式：

$$T_{total\_d} = C_T + 2 \times K \times t_{copy} \times \prod_{i}^{n} d_i + d \times t_{ele\_copy} + d \times t_{scan} \div K + \prod_{i}^{n} d_i \times t_{agg} \times K \tag{3.20}$$

$$S_{max\_d} = C_S + d \times sd \times 2 + \prod_{i}^{n} d_i \times s \times (K + 1) \tag{3.21}$$

在这个前提下，数据集的总扫描／计算复杂度与仅扫描一遍数据集相当，因此在这部分上可以说是速度最优（时间最优）方案。另外，由于对数据的聚合本身就是一种将大量数据变成少量具有概括性的数据的过程，因此拷出时间，以及子 cuboid 之间聚合的时间对于数据集扫描与计算的过程本身应当是微不足道的。于是结合这个前提，该方案在全局上也基本是速度最优的。

但是另一方面，我们能够看到，每一个 kernel 都需要自己的一块 cuboid 空间以无冲突地计算并生成自己负责的子 cuboid，当并行数很高，又或者是 cuboid

的 cell 数量很多的情况下，现阶段的并行设备存储很难具有足够的空间支持这样的分配。因此，势必要有这么一种方案，通过牺牲一定的性能来使得算法能够执行下去。

第五节　混合方案

基于上述两个小节的讨论，我们发现，如果系统资源足够的话，我们可以直接采用速度最优方案来实现我们的 cuboid 计算。但是有些情况下，系统未必能够满足这样的需求。因此，在速度最优方案的基础上，我们需要加入一些从 memory 最优方案中吸取的思想，在使得算法能够正常运行的前提下能够尽量不牺牲太多的性能。因此，需要考虑混合方案，即 kernel 的处理单元同时以 kernel 和 part of dataset 来进行划分。

该方案主要具有如下的特点：

· 一个 kernel 处理对应 dataset part 的某些 cells 的生成。这些计算完的 cell 会被放入下述的内存分配空间中的对应位置。

· 由于并行计算模型中很难控制具体哪些 kernel 的先后执行顺序，但是大致可以确定一个总体的执行顺序，即位于相近 part 的 kernel 总是倾向于在一起被执行，所以内存分配采用如下方式：分配 $\lceil K \times g \div \prod_i^n d_i \rceil + 1$ 个 cuboid 的共享空间。这样基于 kernel 的运行规律，再根据抽屉原理就可以计算出，在新的 part 的 kernel 开始计算的时候，这个共享空间中一定有至少一个 cuboid 的空间，位于这个空间的老 cuboid 是已经计算完成并且可以拷出的。即，此时我们能保证各个 kernel 之间读写的独立性。

· 从最终的运行结果上看，总共需要经历 p 次全 data cube 上的 aggregation，以及 p 次全 data cube 的拷贝（入与出）。

结合上述三点，我们就可以得出一个最终的时间表达式：

$$T_{total\_m} = C_T + 2 \times p \times t_{copy} \times \prod_i^n d_i + d \times t_{ele\_copy} + d \times t_{scan} \times \prod_i^n d_i \div (g \times K) + \prod_i^n d_i \times t_{agg} \times p \tag{3.22}$$

以及最大空间表达式：

$$S_{max\_m} = C_S + d \times sd \times 2 + \prod_i^n d_i \times (\lceil K \times g \div \prod_i^n d_i \rceil + 1) \times s \tag{3.23}$$

此时可以发现，memory 最优方案即为 $\prod_i^n d_i \geq g \times K$ 且 $p = 1$ 的该方案的特殊情形；速度最优方案则为 $p = K$ 且 $g = \prod_i^n d_i$ 的情形。因此也证明了这个混

合方案的正确性。

根据混合方案的表达式，通过适当控制 g 与 p 两个参数在适合的值，即可在速度最优的方案的基础上，通过牺牲部分性能来减少内存空间占用，从而使得算法能够照常运行下去。

另外，在此方案中，并不能保证某 n 批次的 kernel 处理完毕时正好落在某个 part 的边界，因此需要额外的内存空间来作为其他的 partition data cube 的空间以保证聚合的正确。因此在空间表达式上，混合方案和 memory 最优方案以及速度最优方案给出的表达式有一些微小的偏差。

第四章　从原始数据聚合 cuboid 到目标数据聚合 cuboid 的聚合优化

第一节　目标与评估标准

在本章的开始，我们首先明确该问题的定义与目标。

cuboid 具有 n 个维度，其中每个维度的取值数量由（$d_1, d_2, \cdots, d_n$）决定。假定为了聚合，对聚合前 cuboid 中的一个特定的 cell 的扫描与处理时间时间是一个常数 t_{cs}。

给定一个聚合顺序为（$i_1, i_2, \cdots, i_j$），则第一步聚合的维度为 i_1，聚合所需要扫描的 cell 的数量为 $\prod_i^n d_i$，第一步聚合完成后的 cuboid 的总 cell 数为 $\prod_i^n d_i/(\prod_l^1 d_{i_l})$，以此类推，就可以得到总的 j 步聚合需要在扫描上花费的总时间为：

$$T_{c\_agg\_to\_c} = t_{cs} \times \sum_{m=0}^{j-1}(\prod_{i=1}^{n} d_i/(\prod_{l=1}^{m} d_{i_l})) \tag{4.1}$$

于是我们的总目标就是：尽可能减少平均到每个 cuboid 计算上的 $T_{c\_agg\_to\_c}$，使得在相同的硬件条件下能得到更快的计算速度。

第二节　聚合路线的贪心选择算法

本小节主要介绍一个已有的路线选择算法。在并行计算的背景下，它仍然适用，并且依然能给出在这个步骤下的最优方案。

一、算法的描述

该算法思路非常简单：我们总选择当前剩下来需要被聚合的维度中维度值最大的那一维即可。下面将给出其最优性的简单证明。

二、算法的证明

证明

现在来考察一个聚合过程中的某两步，这两步分别选择聚合第 i 维和第 j 维。

考察这两步聚合发生前的聚合 cuboid 与之后的聚合 cuboid：与这两步聚合的具体顺序无关，结果来看都为后者比前者多聚合了 i，j 两个维度。因此，无论这两步按照如何的顺序发生，对于这两步之前的扫描次数总和，与这两步之后的扫描次数总和都是无关系的。

接下来假定 i，j 两步聚合后的聚合立方体还有 cell 的个数为 D，以及 i 先被聚合。则我们可以计算得出这两步中被扫描的 cell 的总数为：$D \times d_i \times d_j + D \times d_j$

类似的，当假定 j 先被聚合，得到的是：$D \times d_i \times d_j + D \times d_i$

此时可以看出，先聚合 d_i、d_j 中比较大的那一个对应的维度时会得到更小的总扫描次数。

因此，给定任意一个序列，除非其已经是严格按照如下的标准来完成聚合，否则总能找到一次调整，使得调整后的聚合序列拥有更少的总扫描次数。

序列对应的维度值是严格单调非增的。

而满足最少总扫描次数的聚合序列，可以由一个简单的贪心算法得到（前述）。

证毕。

□

第三节　从给定的预处理 cuboid 中的最优起始选择

事实上，对于一个系统而言，存在着大量的闲时，意即存在这样一些时刻，由于没有相应的计算任务，系统计算资源得到空闲。而由上一小节的分析可知，从源 cuboid 到目标 cuboid 是需要计算的，这个计算量与聚合操作的次数，路径上的中间 cuboid 都有关。

作为上一小节中提到的算法的前提条件，“源 cuboid”与“目标 cuboid”是明确的，即从源 cuboid 到目标 cuboid 的聚合次数是一定的，而上一小节中的算法在这个前提下给出了最小代价的聚合方案。然而在实际操作中，用户只需求目标 cuboid，意即其并不关心这个目标 cuboid 是从哪个地方来的（可能是直接从源数据生成，也可能是由其他一些已经生成好的 cuboid 生成）。于是，一个直观的想法就是我们或许可以不需要每次都从最底层的 cuboid 来现场计算出目标 cuboid，而是从某些预先计算好的 cuboid 来计算目标 cuboid，从而减少聚合次数，进而减少总的计算代价。同时，因为此时需要读写的 cuboid 较之底层 cuboid，或者是原数据集，将小得多，这样也使得与之相关的读写开销有了明显的改善，从而

在总的处理时间上更占优势。这些预处理的 cuboid 可以在系统空闲时间被计算出来，从而不占用任何的实际运行时间。

一、选择标准

如前所述，假定我们现在已经有了一个预处理过后的 cuboid 集合。可知这个集合中势必包含一个最底层的 cuboid：

定义 4.1 任意给定一个目标 cuboid，一个预处理的 cuboid 能生成这个目标 cuboid 的充要条件是：预处理 cuboid 中不存在这样的已聚合维度，使得目标 cuboid 中该维度未聚合。

沿用上一小节给出的时间计算式，我们可以简单地得出我们问题的定义：从满足上述定义的 cuboid 中寻找一个使得由该计算式能给出最小值的 cuboid，此 cuboid 即为所求。

二、选择算法的实现

这个问题可以由一个并行算法实现，每个 kernel 简单地使用一个位于共享内存中的内存单元以存放最后的计算结果。下面给出算法运行的具体步骤：

事实上，由于预处理的 cuboid 未必很多（每个 cuboid 本身是会占用不少存储空间的），而其中维度数据占的比例又非常小，因此这个算法即使全部交由 CPU 来执行也是可行的。

第四节 最优化预处理 cuboid 生成

如前所述，在考虑 cuboid 的生成问题时，没有必要每次都从最底层的 cuboid 来进行生成，而是可以从预先预处理好的 cuboid 中寻找一个较为合适的，在被选择的 cuboid 的基础上更快更好地生成所需的目标 cuboid。这一小节从经典的 cuboid 预处理选择生成算法出发，在新的硬件模型上对此进行了一定的拓展。

一、古典方法（Stanford，1996）

在[12] 中，一种以硬盘 I/O 时间作为主要衡量因素的评估标准，以及由此而生的一种生成次最优解的贪心算法被提出。具体的算法描述如下：

```
Data: 预处理 cuboid 集合，目标 cuboid
Result: 最优的 cuboid
Generate computation kernel based on system dispatch scenario;
Each kernel read the dimensional information of one of the pre-processed
  cuboids;
while kernels to execute exists do
    Use this kernel to test if the target cuboid can be reached from this cuboid;
    if true then
        Calculate the total cost of this route;
        Copy the result back to the shared memory;
    end
end
Check all the result and choose the minimum one;
```

算法 4.1: 在已有的预处理 cuboid 中寻找最优的预处理 cuboid

假定每个 cuboid（下文称为 v）在进行处理时拥有一个花费函数 $C(v)$，同时假定在除了最底层的 cuboid 之外，预先处理好的 cuboid 的个数为 k。集合 S 作为当前已选择的 cuboid 的集合。定义一个收益函数 $B(v,S)$，具体定义如下：

定义 4.2 对于每一个能从 v 聚合得来的 w，定义 B_w 为

1. 令 u 是在 S 中的能聚合生成 w 的且拥有最小的花费函数值的 cuboid
2. 如果 $C(u) > C(v)$，则 $B_w = C(u) - C(v)$, 否则 $B_w = 0$

$B(v,S) = \sum B_w$

由此我们可以看出，收益函数的定义实质是“选择当前 cuboid 比起选择由它而生成的所有 cuboid 相比能多得到的收益”。

基于这个收益函数，为了找到一个全局最优的方案，我们当然可以通过枚举所有的 k 个 cuboid，在其中找到一个最优的方案。但是这种做法是 NP-hard 的。为了解决这个问题，我们退而求其次，求一个次优的方案。此时有一个对于所需生成的 cuboid 进行选择的贪心算法如下：

Data: Nothing

Result: 最优的预处理集合

Initial state: $S = \{bottomcuboid\}$, $i = 0$;

while $i < k$ **do**

  Select the cuboid v that isn't in the set S and can make the $B(v, S)$ be the maximum.;

  $S = S \bigcup \{v\}$;

end

Return set S;

算法 4.2: 寻找最优的预处理集合

二、对花费函数的修正

上一小节中提到的贪心算法，可以看到其最终执行的效果与 $C(v)$ 的选择息息相关。在前述论文发表的时代背景中，可以认为"从硬盘到内存中的对于预处理 cuboid 的存取"占用了绝大多数的时间，意即存储之间的 I/O 成为了整个系统的绝对瓶颈；另一方面，I/O 时间在系统硬件配置恒定的情况下基本只与存取的数据的大小有关，具体到每个数据聚合 cuboid 上，就是和 cuboid 作为一张 table 所具有的行数（cell 的数目）有关。因此，在原论文中，这个花费函数被简单的设定成为了该 cuboid 的行数。所有的计算都通过"I/O 数据量最优"来间接地指向"时间最优"。

然而，上述的假定并不周全。我们再来详细考察从一个事先预处理好的 cuboid 到目标 cuboid 的过程：

因此，对于一个完整的过程，我们能得到每一步的计算时间表达式为：（假定目标 cuboid 有 D 个 cell，从预处理的 cuboid 到目标 cuboid 一共聚合了维度 q 个，它们的维度值分别为 $(d_1, d_2, , d_q)$，一个 cell 的大小为 s）

第 1 行：从硬盘到主存的拷贝：

$$T_{copy\_hd\_to\_m} = C_{copy\_hd\_to\_m} \times D \times \prod_{i=1}^{q} d_i \times s \tag{4.2}$$

循环节内部的 3 行代码将如下考虑：每一步执行的 cuboid 都比上一步少了一个维度，具体少的维度可以由前述的方法给出，这里将 q 次的时间总和写在一起：

Data: 起始 cuboid
Result: 目的 cuboid
1 (*)Copy the corresponding cuboid(s) into main memory;
2 **while** *The target cuboid doesn't exist* **do**
3 Copy the corresponding cuboid(s) to the parallel device when a computation mission exists;
4 Parallel device calculates the result cuboid;
5 Copy the result cuboid back to main memory;
6 **end**

算法 4.3: 从起始 cuboid 到目的 cuboid 的计算过程

第 3 行：设备初始化与 cuboid 的拷入。总共的拷贝时间为：

$$T_{copy\_m\_to\_d} = C_{init\_time} + C_{copy\_m\_to\_d} \times \sum_{m=0}^{q-1}(D \times \prod_{i=1}^{q} d_i \div (\prod_{l=1}^{m} d_{i_l})) \tag{4.3}$$

第 4 行：计算所有的 cuboid。总共的计算时间是：

$$T_{cal} = t_{scan\_per\_cell} \div K \times \sum_{m=0}^{q-1}((D \times \prod_{i=1}^{q} d_i) \div (\prod_{l=1}^{m} d_{i_l})) \tag{4.4}$$

第 5 行：cuboid 的拷出。总时间为：

$$T_{copy\_d\_to\_m} = C_{copy\_d\_to\_m} \times \sum_{m=1}^{q}(D \times \prod_{i=1}^{q} d_i \div (\prod_{l=1}^{m} d_{i_l})) \tag{4.5}$$

那么这个 $C(v)$ 函数究竟应该如何构造呢？我们不妨这么考虑：假定现在有两个 cuboid，分别为 a 与 b，其中 a 可以生成 b（意即 a 经由聚合能达到 b）。我们现在暂且不知道 $C(a)$ 与 $C(b)$ 如何取值，但有一点可以明确的是，选择 $C(b)$ 和选择 $C(a)$，在以后生成所有可以由 b 生成的 cuboid 时，可以有差值：$C(a) - C(b) = \Delta read\_time(a,b) + generate\_time(a->b)$。于是，顺着这个思路，定义最底层 cuboid 的 $C(bottom)$ 为 0，则有：

定义 4.3 $C(bottom) - C(v) = -C(v) = C_{copy\_hd\_to\_m} \times sizeof(source\,cuboid\,file - target\,cuboid\,file) + C_{copy\_m\_to\_d} \times count_A \times s + C_{copy\_d\_to\_m} \times count_B \times s + t_{scan\_per\_cell} \div K \times count_A + C_{init\_time}$

即 $C(v)$ 是上述式子的相反数。

我们可以发现，由于我们的定义中，$C(v)$ 指的是“相对于最底层 cuboid 的花费的差值”，因此如果选择对应的 cuboid 比选择底层的 cuboid 要更优，则该值应该是负的。

在这个式子中，$count_A$ 与 $count_B$ 分别代表在具体的计算中，相应的部分需要处理的 cell 的个数。

我们可以发现，出现了一些新的常量，而这些常量与文中提到的聚合方法，以及系统的一些具体硬件参数是有关的：$C_{copy\_hd\_to\_m}$：指从硬盘向内存中拷贝单位大小的数据所需要的时间。C_{init_time}：初始化指定的 cuboid 聚合函数所需要的时间。$C_{copy\_m\_to\_d}$：从内存向并行设备拷贝单位大小的数据所需要的时间。$t_{scan\_per\_cell}$：聚合方法中，单个 kernel 对于每一个聚合前的 cell 进行扫描以及聚合进聚合后的 cuboid 的平均时间。$C_{copy\_d\_to\_m}$：从并行设备向内存拷贝单位大小的数据所需要的时间。

关于这一小节留下来的这些常量，它们如何取值很大程度上决定了我们该如何在前人论文的基础上修正我们对于 cuboid 的选择方式。它们的具体取值将会在第五章通过实验来推定，从而在此基础上再来回顾修正前后的花费函数对于我们最终生成预处理 cuboid 的决策的影响。

最后，注意到上述算法的（*）。在当今的计算机系统中，内存的成本已经变得越来越低，以至于很多 in-memory database 有了很大的发展空间。作为一个必须长时间保证稳定运行的系统，数据库系统也有很大的机会通过收集用户的使用数据，从而调整自己缓存在内存之中的数据。这些预处理的 cuboid 也是其中之一。因此，在这个前提下，我们可以将之前归纳出的公式进行修改，从而得到在 in-memory database system 中的花费函数为：

定义 4.4 $-C(v) = C_{copy\_m\_to\_d} \times count_A \times s + C_{copy\_d\_to\_m} \times count_B \times s + t_{scan\_per\_cell} \div K \times count_A + C_{init\_time}$

第五节　聚合的并行化实现

前述的章节中，我们阐明了如何选择聚合的各个因素，从而使得聚合的平均性能尽可能达到最优。本小节将讨论如何具体实现从源 cuboid 到目标 cuboid 的聚合。

由上一小节，我们计算出了一个使得平均聚合代价最优的预处理集合；由上上小节，我们找到了一个能最优化对于给定目标 cuboid 的生成的预处理 cuboid，

并且在此同时由上上上小节得到了一条从选择出的预处理 cuboid 到目标 cuboid 的生成路线。因此，这一步要做的就是按照这个路线，逐个逐个地由预处理 cuboid 生成到目标 cuboid。可以发现这个问题每个步骤其实是相似的：聚合当前 cuboid 的某个给定的维度。因此下文的讨论中仅讨论一步的聚合。

进行如下的假定：对于当前 cuboid，未聚合维度的维度值集合为 $(d_1, d_2, , d_j)$。假定我们需要聚合维度 i，考察聚合后的 cuboid，我们可以发现，所有满足如下条件的聚合前的 cuboid 的 cell 将会被聚合成聚合后 cuboid 的一个 cell：

$(ddk_1, ddk_2, , ddk_{i-1}, d_i, ddk_{i+1}, , ddk_j)$，其中 ddk_m（m 从 1 到 j，m 代表维度）是明确的值，d_i

$(ddk_1, ddk_2, , ddk_{i-1}, ddk_{i+1}, , ddk_j)$

这个 cell 中。

举个例子来说，假如是 5 维的，我们要聚合第 4 维，并且将其聚合到则聚合前的 cell 的 $(3, 7, 2, 6)$ 这个 cell 之中，则聚合之前的 cell 一定有属性值为 $(3, 7, 2, d_i, 6)$。

因此，我们可以令每个 kernel 生成一个聚合后的 cell。注意到所有的 kernel 合起来将完成一次对于聚合前 cuboid 的扫描，因此计算代价上是最优的，无需像第 3 章中一般特地使用多个 cuboid 的存储空间，而仅需要一个共享的 cuboid 空间即可。同时这个定义可以直接避免读写冲突，因此这是一个可行的方法。

下面给出该算法的具体过程：

Data: 原始 cuboid
Result: 一步聚合 cuboid
Generate computation kernel based on system dispatch scenario;
while *kernels to execute exists* **do**
    Use this kernel to scan the corresponding cells in raw cuboid;
    Aggregate them into the next cuboid;
end

算法 4.4: 并行化的一步 cuboid 聚合

由此可以得到时间开销与空间开销为：

时间：$d_1 \times d_2 \times \cdots \times d_j \times t_{scan\_per\_cell} \div K$

空间：$d_1 \times d_2 \times \cdots \times d_j \times s + d_1 \times d_2 \times \ \times d_{i-1} \times d_{i+1} \times \cdots \times d_j \times s$

注意到在该聚合方法中，对于每个聚合前 cuboid 的每个 cell，我们简单地

取出内部的数据，并将其累计进入对应的聚合后的 cell，那么基于前文所假定的 cell 结构相似性，我们可以认为 $t_{scan\_per\_cell}$ 在任何阶段都是一定的。

第五章　实验与评估

在本章中，由于 OpenCL 的多平台性，我们将在两个不同的平台上进行：一个是 CPU 上的并行计算，另一个是 GPU 上的并行计算。通过同样的一个 OpenCL 程序在两个不同的硬件架构上的运行来确定在不同的情况下各个在上一章中提到的常数，并用这些常数具体评估在不同硬件平台上的运行性能，并对这些结果进行对比。

第一节　实验环境

本次实验采取了两种并行硬件架构来完成实验：

1. CPU 架构运行在如下的硬件上：Intel Core i5 Dual Core Processor@2.9GHz，8GB 2133MHz LPDDR3 memory，512GB SSD。

2. GPU 架构运行在如下的硬件上：ATI Radeon HD 8570，Intel Core i7-6700@3.4GHz（8 cores），8GB 2133MHz DDR3 memory，128GB SSD

由于条件所限，实验时选择的 GPU 是相对较老的款式，性能上较为不足。因此实验数据的考察更偏向于对于相对并行性能的考察。

另外，在第四章中我们也提到，在花费函数的估计式中出现的常数的大小，以及它们之间的比例关系，会对最终的花费函数的计算结果产生影响。选择这块 GPU 的理由中也包括了基于“创造出一种条件，使得花费函数估计式中系数的比重有所不同”的考量。

另外，实验中所采用的 OpenCL 的库版本均为 OpenCL 2.0，在每一个 cell 所采取的聚合函数包括以下几种：对于数据项个数的统计；对于测量值的最大值，最小值，求和，以及根据测量值的 10 个分段的直方图归类。

一、由实验环境进行的参数选择测试

在进行实验之前，首先需要对上一章中提到的诸常数进行考察，从而明确在算法运行的全过程中哪方面将成为性能瓶颈。

对于 $C_{copy\_hd\_to\_m}$：

对于 $C_{copy\_hd\_to\_m}$ 的考察可以通过读取的文件的大小来确定。我们可以采用一个“纯粹的读取程序”——即只将数据读入内存，在读取的过程之中不进行任何

多余的操作——来完成求出这个常量的实验。

因此，我们直接简单地利用现有程序中的“从硬盘中读入数据集”这一步来进行系数的计算。对不同大小的数据集进行读入所需要花费的时间为（均为 10 次的平均数值）：

在 CPU 架构上：

表 5.1　CPU 架构上的硬盘读写测试

数据条目数	总的数据集文件大小（byte）	平均读取用时（s）
262144	3479769	0.3520096
2097152	27841394	2.741692
16777216	222741273	21.11312
134217728	1781907696	166.6692

取这四个数据点进行拟合，能得到直线的斜率，即我们所需的 $C_{copy\_hd\_to\_m}$ 的估计值为 9.34613×10^{-8}s / byte

在 GPU 并行架构上：

表 5.2　GPU 架构上的硬盘读写测试

数据条目数	总的数据集文件大小（byte）	平均读取用时（s）
262144	3479769	0.0813523
2097152	27841394	0.3875493
16777216	222741273	0.2.85776
134217728	1781907696	22.504198

取这四个数据点进行拟合，能得到直线的斜率，即我们所需的 $C_{copy\_hd\_to\_m}$ 的估计值为 1.26069×10^{-8} s / byte

对于 C_{init_time} 在 CPU 架构上：通过对不同大小的 dataset，以及分配的 kernel 数不同等等变量的对照实验，我们可以发现这个时间基本都落在 0.002s 到 0.0025s 内，因此我们可以取定 $C_{init_time} = 0.00225$s 作为我们的实验用常数

在 GPU 架构上：类同于上述的观察方式，可以得到在该 GPU 平台上这个值均匀落在 0.19s 到 0.20s 之间，因此我们这里取 $C_{init_time} = 0.195$s

对于 $C_{copy\_m\_to\_d}$ 与 $C_{copy\_d\_to\_m}$

这两个变量其实我们可以统一成同一个量，因为决定设备之间拷贝速度的最大因素是设备之间的总线传输速率，而这个速率并不因数据的流向而改变。

对这部分的测量，我们采用对于数据拷出这个部分的时间来作为标准。在已经实现的代码中，拷出的数据仅为一个完成聚合的最底层 cuboid，而如前文所述，每个 cell 的结构是相同的，以至于他们的大小也是相同的，于是拷出的数据量完全由拷出的 cell 个数决定。

我们通过修改数据集中每个维度数据的可能的取值的数目，来控制最初生成的 cuboid 中 cell 的个数（因为这个 cuboid 中对于每个维度的每个取值的所有组合，都有且仅有一个 cell 与之对应，即此时 cell 数 = 维度值的总乘积）。cell 的大小：136 bytes。测试时为了简便都使用的三维聚合。单个 kernel 中分配的内存，除了 cell 的空间以外，还有一些对于 kernel 本身信息的标记，所以在表格中看到的实际的内存占用，与理论计算出来的 cell 个数 × cell 的值将有所差距。

在 CPU 架构上：

表 5.3　CPU 架构上的并行设备读写测试

X	Y	Z	cell 总数	单 kernel 占用	kernel 数	总拷贝量	均时间
31	79	11	26939	3663708	16	58619328	0.02738936
7	11	13	1001	136140	16	2178240	0.00081016

注：时间的单位：s，占用内存的单位：byte

由此可以作差估算出 $C_{copy\_d\_to\_m}$ 与 $C_{copy\_m\_to\_d}$ 为：4.70919×10^{-10}s / byte

在 GPU 架构上：

实际上，基于上述测量准则的测试中，该部分时间过于短暂以至于无法测出，我们姑且认为在最小 OpenCL 的测量时间跨度（1×10^{-6}s）内就已经完成，因此此处可以得到系数 $C_{copy\_d\_to\_m}$ 与 $C_{copy\_m\_to\_d}$ 不大于：1.7×10^{-14}s / byte

对于 $t_{scan\_per\_cell}$ 这个系数将在具体求出不同平台的花费函数估计式的时候被计算。

第二节　实验结果

在本小节中，本文提到的数个算法的运行性能将在两种并行平台上得到评估。

一、对于从源数据生成底层 cuboid 生成算法的评估

为了检验在不同的数据集上并行与否对性能的影响程度，在实验中我们将采用数据条数为 262144，2097152，16777216，134217728 这四个规模的数据集来进行。根据前文的算法设计，数据聚合立方体中 cell 的个数并不会影响到算法的时间复杂度，因此我们在每个测试之中都将数据维度值控制在 $(19, 101, 13)$（即 cell 总数为 24947）。另外，这里仅对速度最优方案进行评估。

在 CPU 架构上：

为了检验不同的 kernel 数对于实际运行速度的影响，我们在 134217728 条目的数据集上分别进行了不同的 kernel 数（指生成的并行 kernel 数，并不是指系统中实际同时运行的 kernel 数）运行的测试。这里所有的运行时间都是五次运行的平均。结果如下：

表 5.4　CPU：Kernel 数-底层 cuboid 生成时间（单位：s）表

kernel 数	计算部分的运行时间（s）
1	2.810336
2	1.798154
4	1.701672
8	1.718508
16	1.62508

另外，对于不同规模的数据集的测试有结果如下（基于 kernel 数 =16）

表 5.5　CPU：数据集大小-底层 cuboid 生成时间（单位：s）表

数据集条目数	计算部分的运行时间（s）
262144	0.0260788
2097152	0.06733256
16777216	0.3991288
134217728	1.62508

在不同的 kernel 运行时，从系统管理中能够查询到的内存空间占用如下表：
由上面这些测试数据和图表，可以得到如下结论：

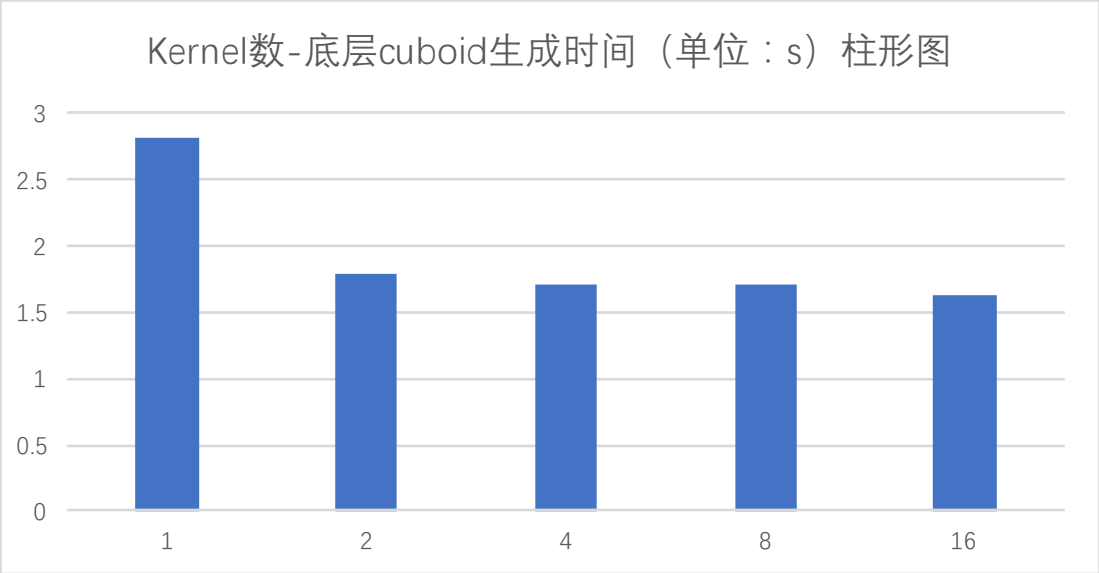

图 5.1　CPU：Kernel 数-底层 cuboid 生成时间（单位：s）柱形图

表 5.6　CPU：数据集大小-kernel 数-内存占用（单位：MB）表

数据集条目数	kernel 数	总内存占用
262144	1	18.5
262144	2	25.0
262144	4	37.9
262144	8	63.8
262144	16	115.6
2097512	16	171.6
16777216	16	619.6

1、由于该实验平台的 CPU 是 2 core 的，因此在多核实验中，生成的计算核心在超过 2 之后运行速度并没有实质性的变化（因为系统同时只能执行两个计算核心）。

2、另一方面，通过数据集大小和生成底层 cuboid 的时间的图表，可以看到这个算法运行的时间确实和原始数据集大小基本呈线性关系，与最初的算法分析一致。

在 GPU 架构上：

为了检验算法在更多核的平台上是否具有更好的可拓展性，因此该实验在 GPU 上重新部署了一遍。采用的数据集是 16777216 条目的数据集。具体的实验结果如下所示，其中所有的测量值都是五次测量的平均数。

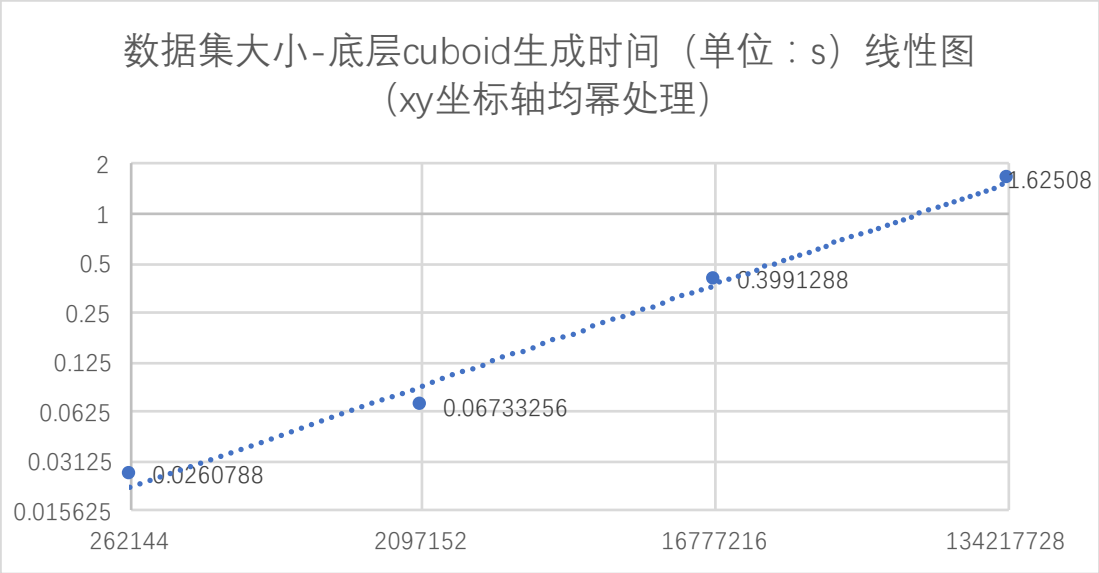

图 5.2　CPU：数据集大小-底层 cuboid 生成时间（单位：s）线性图

注：x, y 轴均经过了幂处理

表 5.7　GPU：Kernel 数-底层 cuboid 生成时间（单位：s）表

kernel 数	计算部分的运行时间（s）
12	7.503698
16	6.260434
32	3.76185
48	2.971528
64	2.967586

从这个实验中我们也可以看出来，在这个设备上的 K 值大概介于 48 到 64 之间。

由此可见，虽然实验结果的数值受限于硬件，仍然可以看出该并行算法具有很好的可拓展性，在具有更多（流）处理核心的并行计算平台上将获得更大的收益。

二、对于由 cuboid 到 cuboid 的聚合算法的评估

这一小节中将进行有无采用并行方式对于聚合性能的评估

在 CPU 架构上：

在本部分的测试中，我们将固定在 cuboid 聚合中使用聚合路线选择算法。采

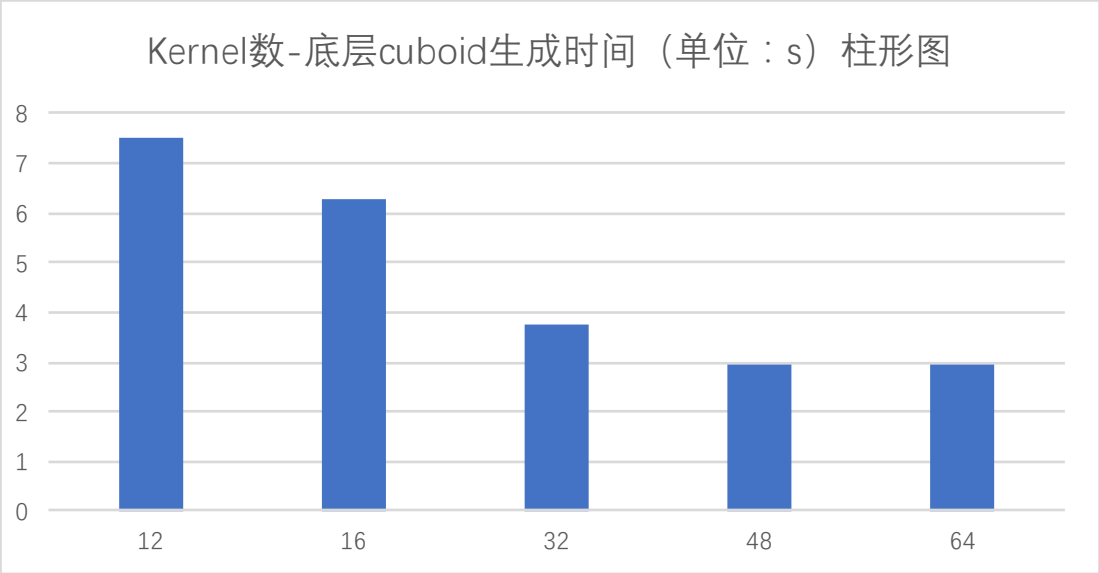

图 5.3　GPU：Kernel 数-底层 cuboid 生成时间（单位：s）柱形图

用的原始 cuboid 是基于 262144 个元素的数据集生成的 cell 总数为 24947(19, 101, 13) 的 cuboid，目标 cuboid 是三步聚合后的顶层 cuboid。另外，所有测试均采用下文将要测试的聚合路径优化算法。测试结果如下：

表 5.8　CPU：Kernel 数-从 cuboid 到 cuboid 聚合时间（单位：s）表

kernel 数	计算部分的运行时间（s）
1	0.001860436
2	0.001092724
4	0.00081602
8	0.000845582

在 GPU 架构上：相同于之前的测试，这里我们仍然使用如下测试方法：原始 cuboid 是基于 262144 个元素的数据集生成的 cell 总数为 24947(19, 101, 13) 的 cuboid，目标 cuboid 是三步聚合后的顶层 cuboid。另外，所有测试均采用下文将要测试的聚合路径优化算法。测试结果如下：

从上述两组测量数据中也能够看出，并行之后的算法效率比起未并行之时有了很大的提升。另外，并行算法的良好的可扩展性也在实验数据中有所体现：其他条件相同的前提下，更高的并行数会带来更高的执行效率。

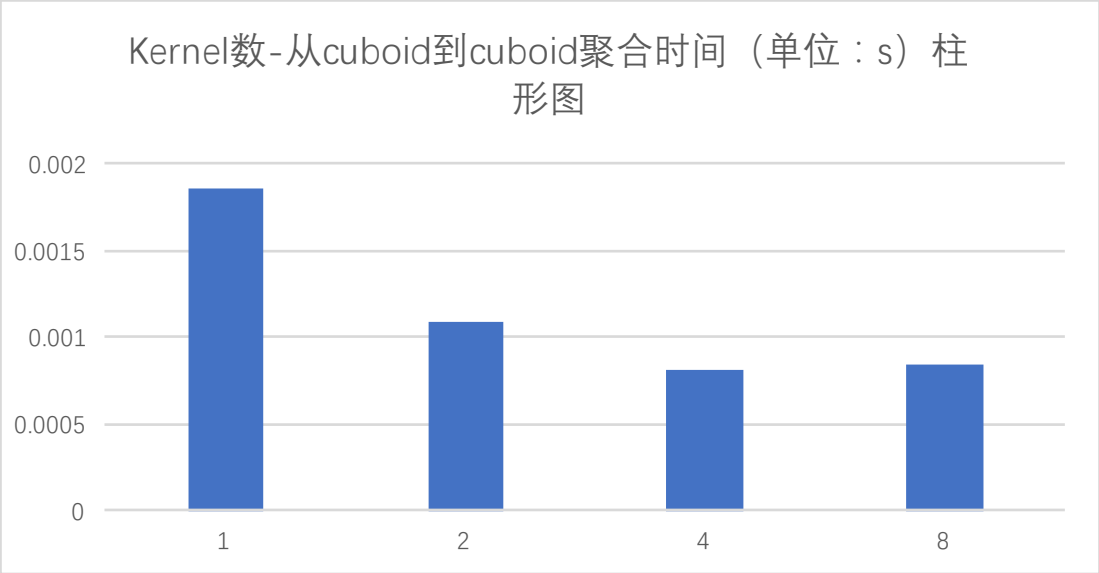

图 5.4　CPU：Kernel 数-从 cuboid 到 cuboid 聚合时间（单位：s）柱形图

表 5.9　GPU：Kernel 数-从 cuboid 到 cuboid 聚合时间（单位：s）表

kernel 数	计算部分的运行时间（s）
12	0.044490438
16	0.036521843
32	0.024351568
64	0.015527524

三、对于聚合路线选择算法的评估

在本小节中，我们将测试不同的聚合路线选择对于从一个较原始的 cuboid 到另一个上游 cuboid 的聚合性能的影响。

下文将列出参加测试的原始 cuboid 的各项参数，以及实际程序运行的结果。为了控制变量的原则，这里固定采用的硬件是 CPU，OpenCL 生成的计算 kernel 数为 16。这些原始 cuboid 都是在之前的实验中生成的。为了简单起见，我们假定所有的聚合的最终目标都是一个“全维度聚合”cuboid，即一个顶层 cuboid，从它将无法再进行任何聚合。另外，由于我们只考察计算部分的时间开销，因此我们不妨将底层 cuboid 放在内存中，以节省进行实验测试的时间。

根据测试所得数据，可以做出柱形图来表现出各种不同的对比方式之间的性能差异。

可以从数据汇总表格和柱形图中看到如下规律：

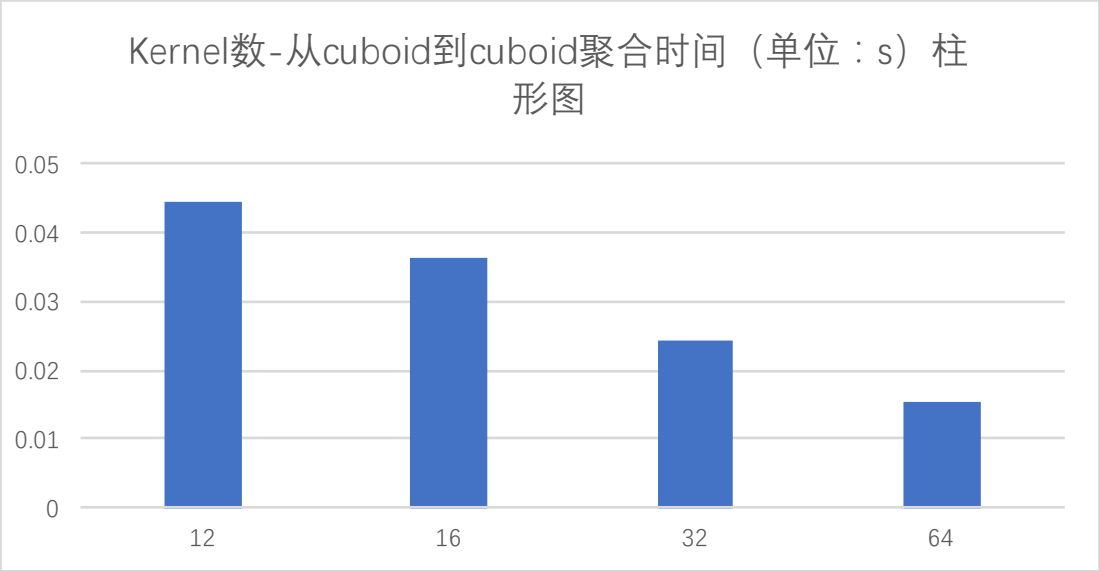

图 5.5　GPU：Kernel 数-从 cuboid 到 cuboid 聚合时间（单位：s）柱形图

表 5.10　CPU：聚合路径的选择与否系列图表（单位：s）表

原数据集条目数	cell 个数	路径优化	第一步	第二步	第三步
134217728	24947	T	0.00085184	0.00002840	0.00000615
134217728	24947	F	0.00082885	0.00010397	0.00001860
262144	24947	T	0.00085994	0.00002619	0.00000640
262144	24947	F	0.00083391	0.00016831	0.00001782
262144	3333	T	0.00027702	0.00000758	0.00000573
262144	3333	F	0.00026012	0.00004609	0.00002158

1、由于这个算法是从 cuboid 出发的计算算法，因此与原始数据集的大小无关。

2、如同分析的那样，我们的路径优化算法在第一步聚合时是不会起到任何作用的，因此第一步的执行时间基本相同，可能由于系统架构或者是代码书写的问题，使用路径优化的方案甚至略慢于不使用路径优化的方案。而从第二步开始，经过了聚合路径优化的算法开始凌驾于未经过优化的算法。但是由于第一步聚合占用的时间占了整个聚合步骤的最主要部分，因此总体时间上来看优化的比例大概在 10%-15% 之间。

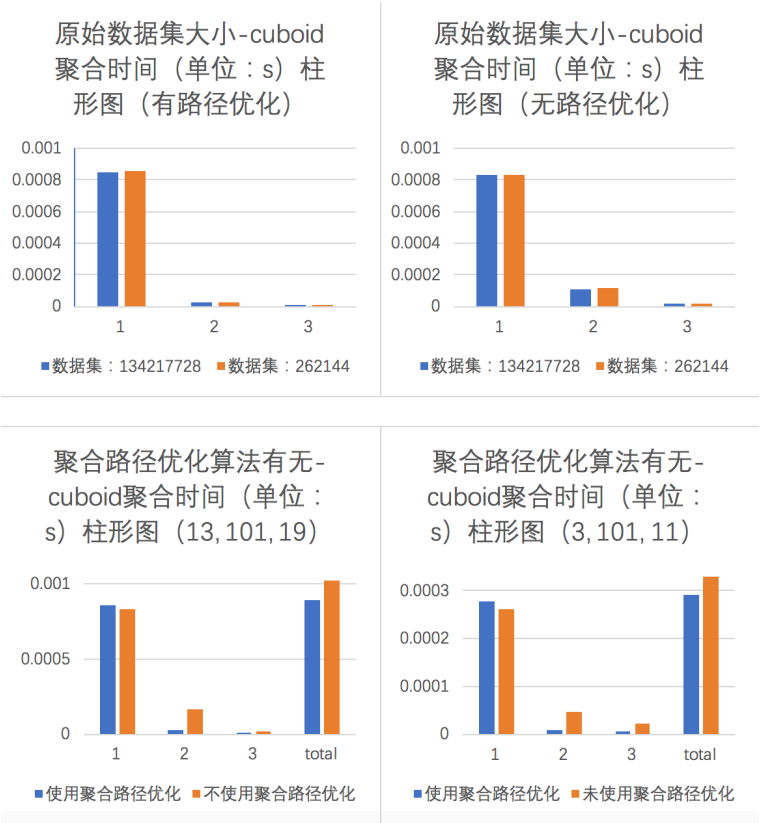

图 5.6　CPU：聚合路径的选择与否系列图表（单位：s）柱形图

四、花费函数在不同实验平台上的实际计算

根据前文测试所得结果，下面我们分别在不同的平台上计算出一个花费函数，并且用这个花费函数与 Stanford 的方案下的花费函数来计算在不同情形下的预处理集合，以及评估他们之间的性能差距。

这个测试中，由于本身维度空间比较小（共 3 个维度），所以可能的 cuboid 总数 $2^3 = 8$ 也并不是太多，因此在我们的测试中固定预处理的 cuboid 数量为 2。评估最后的性能时，我们将以“除去底层 cuboid 以外的所有 7 个 cuboid 生成的总时间”作为评估标准。我们采用的底层 cuboid 的总 cell 数为 24947，其中三个维度的值分别为 $(19, 101, 13)$。另外，我们此处固定采用聚合路径优化算法。

在 CPU 架构上：

根据之前小节中估计的结果，我们已经有了花费函数估计式中的三个系数如下：

$$C_{copy\_hd\_to\_m} = 9.34613 \times 10^{-8}\text{s / byte}$$

$$C_{init\_time} = 0.00225\text{s}$$

$$C_{copy\_d\_to\_m} = C_{copy\_m\_to\_d} = 4.70919 \times 10^{-10}\text{s / byte}$$

考察 $t_{scan\_per\_cell}$，其指的是在从 cuboid 到 cuboid 聚合时“单个 kernel”对于“聚合前的单个 cell”的处理时间。因此，这个地方要进行计算时，首先要选择的数据集应该也是“单 kernel 前提下”进行聚合的时间测量结果。因此我们将采用之前小节中聚合 cuboid 的测量数据进行计算。

测量表格中，给出的是“三步聚合的时间总和”，实际上我们在测量时也测量了第一步聚合的时间。我们在这里选择第一步聚合的时间作为我们的计算基准，是因为这样能尽可能平摊可能产生的系统与测量误差到每一个 cell 上，从而让我们的估计更为精确。

从实验数据中我们可以直接得到，在该平台上，对于 24947 个 cell 的扫描／聚合操作的总时间为 0.00184109s，即 $t_{scan_per_cell} = 0.00184109 \div 24947 = 7.38 \times 10^{-8}$s / cell

在 GPU 架构上：

根据之前小节中估计的结果，我们已经有了花费函数估计式中的三个系数如下：

$C_{copy_hd_to_m} = 1.26069 \times 10^{-8}$s / byte

$C_{init_time} = 0.195$s

$C_{copy_d_to_m} = C_{copy_m_to_d} = 1.7 \times 10^{-14}$s / byte

利用与之前类似的方法，也能计算出 $t_{scan_per_cell} = 1.36165 \times 10^{-5}$s / cell

考察上一章中花费函数中提到的 $count_A$ 与 $count_B$，结合我们此处具体设置的实验条件，我们可以直接给出关于这两个量在所有情况下的具体取值。此时“cell 的处理个数”指的是由原始 cuboid 到目标 cuboid 的路径上总共需要处理的 cell 个数。

于是，根据以上这些数据，我们可以计算出在两个不同的平台上的所有 cuboid 的 $C(v)$ 了。

首先针对通常的 database system，这样的 system 将预处理的 cuboid 缓存在硬盘等外部存储设备中。下面两个表格给出了我们计算出的 $C(v)$ 值。第一列如同前述两个 count 的表格，均指的目标 cuboid 中的 cell 个数；第二列则直接列出根据前述花费函数公式，以及本小节中给出的常数值来计算的每个 cuboid 的花费值。

在 CPU 平台上：

在 GPU 平台上：

可以看出来，在我们的估计方案的基础上估计出来的 $C(v)$ 和传统 Stanford

表 5.11 $count_A$ 表

目标 cuboid 的 cell 个数	$count_A$（总共需处理的 cell 个数）
1919	24947
1313	24947
247	24947
101	24947 + 1313 = 26260
19	24947 + 247 = 25194
13	24947 + 247 = 25194
1	24947 + 247 + 13 = 25207

表 5.12 $count_B$ 表

目标 cuboid 的 cell 个数	$count_B$（总共需处理的 cell 个数）
1919	1919
1313	1313
247	247
101	1313 + 101 = 1414
19	247 + 19 = 266
13	247 + 13 = 260
1	247 + 13 + 1 = 261

表 5.13 CPU + HD 平台上的 cell 个数-$C(v)$ 表

将预处理 cuboid 的 cell 个数	$C(v)$
1919	-0.234726
1313	-0.240735
247	-0.251306
101	-0.252971
19	-0.253608
13	-0.253666
1	-0.253789

表 5.14　GPU + HD 平台上的 cell 个数-$C(v)$ 表

将预处理 cuboid 的 cell 个数	$C(v)$
1919	-0.247233
1313	-0.248049
247	-0.249484
101	-0.250798
19	-0.250001
13	-0.250009
1	-0.250036

方案估计出来的不一样：并非按照 cell 个数而单调递增。另外，所有的 $C(v)$ 值都是负的，这和直观上的观念“预处理比不处理要有更大的收益”是相符合的。

然后，在计算出来的 $C(v)$ 值的基础上，我们会发现，实际上我们得到的 S 集合是一样的。这证明了我们的方案至少不会比 Stanford 原有的方案更差。具体关于这部分的原因会在接下来的小节中提到。

我们使用预处理的方案与未预处理的方案进行比对，能得到如下结果（运行在 CPU 平台上）：

表 5.15　CPU：预处理-全 cuboid 生成时间（单位：s）表

处理方式	预处理	未预处理
时间总和（读取文件 + 处理）	0.289799	0.901132

可以看到，进行了两次预处理之后，最占用运行时间的次低层 cuboid 的生成都被免去了，从而使得运行效率大幅度提高了。

接下来针对 in-memory database 模型，这样的 database system 会将尽可能多的有用信息缓存在 memory 中，从而使得对这部分数据的读写效率大幅提升。

首先计算这个前提下的 $C(v)$。由于假定预处理完毕的 cuboid 现在存放在内存中，因此我们采用 4.4.2. 后半段提出来的适用于该情形的花费函数。该花费函数中没有 $C_{copy\_hd\_to\_m}$ 一项。

以下是在该前提下计算出来的新的 $C(v)$

在 CPU 平台上：

在 GPU 平台上：

表 5.16　CPU + in-memory 平台上的 cell 个数-$C(v)$ 表

将预处理 cuboid 的 cell 个数	$C(v)$
1919	-0.00489118
1313	-0.00485236
247	-0.00478409
101	-0.00499137
19	-0.00481024
13	-0.00480986
1	-0.00481124

表 5.17　GPU + in-memory 平台上的 cell 个数-$C(v)$ 表

将预处理 cuboid 的 cell 个数	$C(v)$
1919	-0.2162307387
1313	-0.2162307373
247	-0.2162307348
101	-0.2173481446
19	-0.21644094018
13	-0.21644094016
1	-0.2164520036

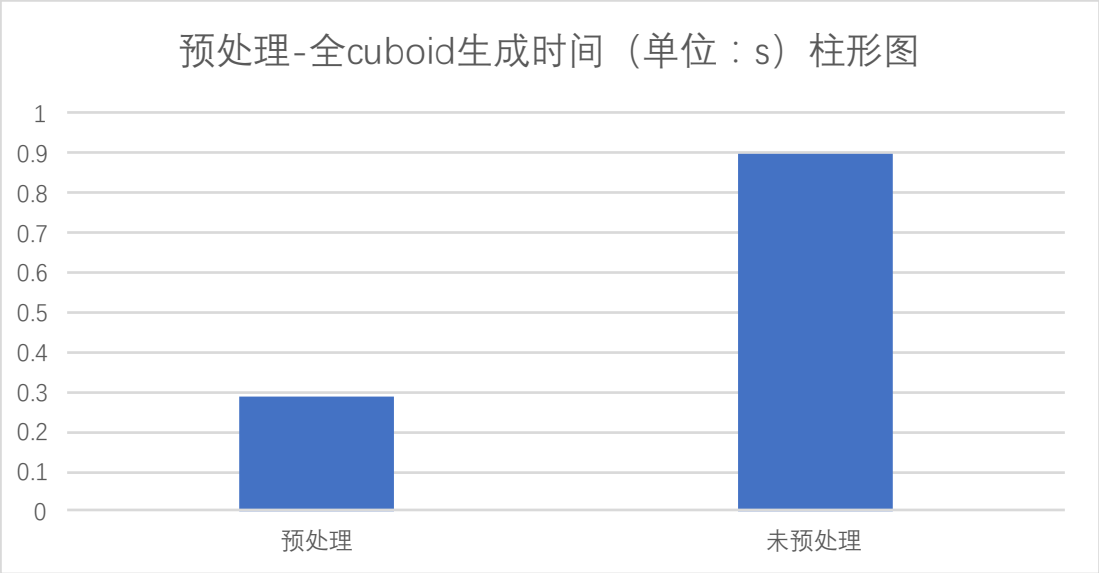

图 5.7　CPU：预处理-全 cuboid 生成时间（单位：s）柱形图

此时我们可以发现，$C(v)$ 的生成方式已经和 Stanford 有了很大的区别。在这个基础上计算出来的 S 集合如下：

Stanford：$\{24947, 247, 1313\}$

我们的方案 _CPU：$\{24947, 1919, 1313\}$

我们的方案 _GPU：（同上）

这里也能看出来，在 GPU 平台上，去掉了 $C_{copy\_hd\_to\_m}$ 项之后，由于 $C_{copy\_m\_to\_d}$ 项对 $t_{scan\_per\_cell}$ 项影响实在微乎其微，因此整个式子的相对大小关系基本全部由扫描一项决定，即扫描的 cuboid 数（$count_A$ 表）成为了 $C(v)$ 差异的关键。而事实上由 $count_A$ 表中也能看出来，由于聚合路径选择算法，使得后续步数的聚合远比前面一步的扫描要耗时更少，因此扫描的 cuboid 数量几乎没有差距，所以最终体现出来估计值几乎完全相同的情况。

最后，我们使用算法给出的预处理集合，分别测试两种预处理方案的性能差距。结果如下（运行在 CPU 平台上）：

表 5.18　CPU：预处理方式-全 cuboid 生成时间（单位：s）表

处理方式	我们的方式	Stanford
时间总和（仅处理）	0.00081504	0.00087807

注：在我们对 in-memory 模型的假定中，不存在硬盘读写

可以看到，虽然差距不很明显，但是在这个场景设置下我们的预处理方案略

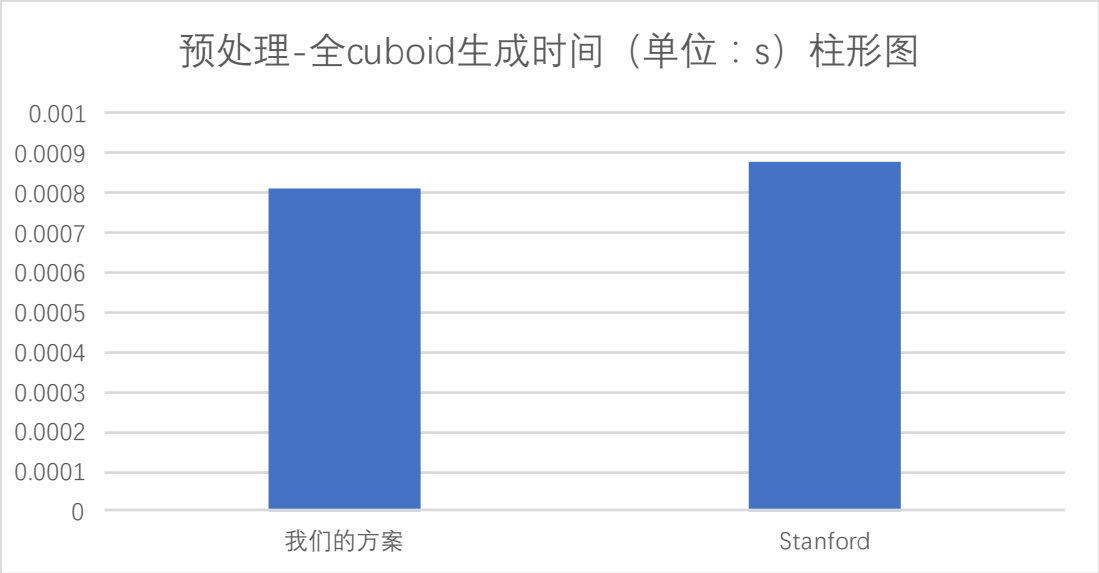

图 5.8　CPU：预处理方式-全 cuboid 生成时间（单位：s）柱形图

优于 Stanford 提出的方案。

第三节　实验分析与讨论

一、何时使用生成并行化的数据立方体生成的混合方案的考量

在本小节中，我们将考察何时必须采用混合方案。

显然，若是条件允许，速度最优方案易于内存管理，易于程序流程控制并且拥有最高的运行速度，应该是我们的首选。而也如前所述，速度最优方案中的最大的问题在于潜在的占用内存空间过多问题。因此考察混合方案何时必须被使用，其实另一方面来说也是考察速度最优方案使用的边界条件。

在讨论内存空间问题的大前提下，我们可以发现，在整个算法运行之中，占用内存最多的主要是两块数据：直接读入的原始数据集，以及在运行中产生的临时 subcuboid。下面就从这两个方面来分析其对于方案选择的影响。

在前面的的实验数据中，可以计算得出，在本实验所给定的 cell 的结构基础上，一个拥有 24947 个 cell 的 cuboid 所占用的空间约为 6.5MB。因此仅从设备空间因素考虑，假定我们有 2GB 内存空间能够分配给这些临时 subcuboid 的前提下，能够同时生成（注意！生成了那么多也未必会同时参加计算。这个受限于系统的 K 值）的最大 kernel 数和一个 cuboid 的总 cell 数的关系大概如下：

基本可以得到如下结论：在维度总乘积小于 100000 的时候，基本上可以不

表 5.19　cell 数目-最大可同时生成 kernel 数表

cuboid 的 cell 的个数	最大可同时生成 kernel 数	CPU or GPU
1000	7692	GPU
5000	1538	GPU
10000	769	GPU
25000	307	GPU
50000	153	GPU
100000	76	maybe GPU
250000	30	maybe very good CPU
1000000	15	CPU
2500000	6	CPU
5000000	3	CPU
7500000	1	x

做任何调整将其部署在 GPU 上进行并行计算，正如本论文中实验所涉及到的数据集那样。在 100000 ～ 2500000 的区间，如果为了保证在 GPU 上的并行度，则应该转而采取混合方案来进行计算；而在 CPU 上仍可以采用直接计算的方法而不影响到并行度。在更大的区间上，则由于存储空间已经不足以支持数个，甚至是一个完整的 cuboid，则无论 CPU 抑或是 GPU 都应该采取混合方案来进行计算。

然后，我们再来考察一下数据集大小对于是否采用混合方案的影响。然而实际上，由于本来就存在“原始数据集过大以至于无法一次全部读取”的问题，所以在内存的分配上，我们可以主要以 cuboid 占用的总空间作为衡量标准，然后对原 dataset 进行一定程度的划分，使得每个 part 的内存占用与临时 subcuboid 占用的空间总和不超过允许的最大限度。在此基础上，每次都在其中的一个 part 上运行我们的算法（然后我们的算法在运行过程中把这个 part 再分成一些更小的 part），并将这个 part 的聚合结果暂时存储在一些其他存储介质中，最后再收集起来形成一个完整的数据集的底层 cuboid。

综合以上两部分的分析，我们可以得知，采取何种方案其实直接相关于问题的规模，即我们需要得到的 cuboid 具体的 cell 总数的多少。然后在这个基础上，我们可以适当的在编程上对一次读入的数据集的大小进行调整，使得程序能尽

可能更快地运行。

二、对于修正后的花费函数的评估

在本小节中，我们主要评估在不同的实验平台，也就是在不同的系数修正下得到的我们的估计花费函数和 Stanford 方案的差异点。

可以发现，实际上在硬盘性能不足，抑或是并行执行的算法计算量本身并非太大的话，对应项的系数修正之后，我们的方案的花费函数的最终表达式基本上和 Stanford 的式子是统一的：在这种前提下，读写仍然是系统运行的主要瓶颈，从修正系数中就可以看出来，该项在整个花费函数中的权重是很大的。

而当我们在更宽的主板数据总线／更快的硬盘，或者直接忽略从外部存储到主存的时间（in-memory mode），抑或是运行更复杂的，计算量更大的并行算法的时候，系数的修正有时就会使得修正后的花费函数开始偏向于花费函数中的计算项或者是设备之间的拷贝项。此时两种花费函数所选择出来的预处理 cuboid 将有可能有所区别。而因为我们的花费函数直接以时间作为考量因素，因此在以时间作为衡量标准的最终运行效率上比以空间作为运行效率的间接考量因素的 Stanford 方案不会更差。

但无论怎样，我们的方案指出了一个事实：在不同的硬件平台，不同的算法构成的前提下，花费函数都是需要重新计算的。

第六章　结论

本文提供了基于 OpenCL 的利用并行计算平台来对 OLAP 中数据聚合立方体的生成／计算进行优化的基本动机，设计方案，具体实现和评估结果。事实证明，采用了并行计算的方案在各方面都强于非并行方案，并且基于并行计算平台的特点而进行的一些优化也体现出了一定的效果。

未来该项研究的扩展方向将会往数个方向进行：重新审视花费函数，更加紧密结合并行计算平台的特点，提出更为通用的模型；将其扩展到 CPU-GPU 结合，或者是其他并行计算硬件的组合的异构计算平台；以及将以上这些技术整合成为一个更为完整的数据库系统。

参考文献

[1] Microsoft. The Manycore Shift: Microsoft Parallel Computing Initiative Ushers Computing into the Next Era[M]. USA: Microsoft Research, 11 2007.

[2] Cuda officical website[EB/OL]. http://www.nvidia.com/object/cuda_home_new.html.

[3] Wikipedia: Cuda[EB/OL]. https://en.wikipedia.org/wiki/CUDA.

[4] Wikipedia: Opencl[EB/OL]. https://en.wikipedia.org/wiki/OpenCL.

[5] He B, Yang K, Fang R, et al. Relational joins on graphics processors[J]. *SIGMOD*. 2008: 511-524.

[6] He B, Liu M, Yang K, et al. Relational query coprocessing on graphics processors[J]. *ACM Transactions on Database Systems*. December 2009.

[7] He B, Yu J X. High-throughput transaction executions on graphics processors[J]. *Proc. VLDB Endow*. 2011, 4 (5): 314–325.

[8] K.Wang, K.Zhang, Y.Yuan, et al. Concurrent analytical query processing with gpus[J]. *Proc. VLDB Endow*. 2014, 7 (11): 1011–1022.

[9] P.Johns, J.He, B.He. Gpl: A gpu-based pipelined query processing engine[J]. *SIGMOD*. 2016.

[10] Wikipedia: Olap[EB/OL]. https://en.wikipedia.org/wiki/Online_analytical_processing.

[11] S.Chaudhuri, U.Dayal. An overview of data warehousing and olap technology[J]. *SIGMOD*. 1997.

[12] V.Harinarayan, A.Rajaraman, J.D.Ullman. Implementing data cube efficiently[J]. *SIGMOD*. 1996.

[13] Plattner H. A common database approach for oltp and olap using an in-memory column database[J]. *SIGMOD*. 2009.

[14] Sap-hana official website[EB/OL]. https://www.sap.com/product/technology-platform/hana.html.

[15] Yuan Y, Lee R, Zhang X. The yin and yang of processing data warehousing queries on gpu devices[J]. *Proc. VLDB Endow*. August 2013, 6 (10): 817–828.

[16] He J, Zhang S, He B. In-cache query co-processing on coupled cpu-gpu architectures[J]. *Proc. VLDB Endow*. December 2014, 8 (4): 329–340.

[17] T.Karnagel, D.Habich, W.Lehner. Adaptive work placement for query processing on heterogeneous computing resources[J]. *Proc. VLDB Endow.* 2017, 10 (7).