摘 要

宏基因组主要研究特定环境中比如海洋、土壤和人体肠道中微生物的组成与环境因素的交互关系，任务之一是将基因按照样本特征进行聚类，进而研究相互关系。但因为每个样本中的基因数据量在百万左右，而样本量在几千左右，聚类算法的复杂度是 $O(n^3)$的，普通的实现方式会耗费大量的时间。在肠道宏基因组分析中，需要面对的基因数据量在一千万，样本量是一千，这样的数据量下，计算复杂度达到 $O(10^{21})$，在大概需要 300 年。本文使用 0.1M 即 10^5 的基因数据量和 10^3 的样本量，计算量高达 $O(10^{16})$,大概需要一天。本文通过运用分布式和异构计算技术对宏基因组的聚类过程进行加速，在不同的聚类过程中达到了 40~2000 的加速比，将计算时间减少到两分钟。

关键字：宏基因组、高性能计算、聚类、CUDA

Abstract

Metagenome mainly investigates interactions between microbial composition and environmental factors in specific environments such as oceans, soils and human intestinal. One of the tasks is to cluster and then study the relationships. However, the complexity of clustering algorithm is $O(n^3)$ and the common implementations would spend a lot of time because each sample has millions of genetic data and the sample size is about several thousands. The amount of genetic data that we face is ten million and the sample size is one thousand in intestinal Metagenomic analysis, thus the computational complexity is $O(10^{21})$ and it should take about 300 years[1]. As used herein, the amount of genetic data is 0.1M(i.e. 10^5) and sample size is 10^3, so the computational complexity reach up to $O(10^{16})$ and it needs about one day. The purpose of this study is to accelerate the process of metagenome clustering through the use of distributed and heterogeneous computing technology, the results show that speedups reached 40 to 2000 and the computing time is reduced to two minutes in different clustering process.

Keywords: Metagenomic, High performance computing, clustering, CUDA

目 录

第一章 引言

随着高通量检测技术的发展，我们可以从全基因组水平定量或定性检测基因的表达水平。基因表达产物中蕴含着基因活动的信息，可以反映细胞当前的生理状态，所以通过对基因表达数据的分析可以获取基因功能和基因表达调控的信息。但是由于生物体中的细胞种类繁多，基因在不同阶段有不同的表达能力，因此基因表达产物十分复杂，数据产量十分巨大。

聚类分析是对大规模基因表达数据分析时广泛采用的手段，其目的是将基因根据其数据特征分组归类,进而研究组间的相似性特征。

通过高通量检测技术提供的基因表达量数据，宏基因组聚类根据不同样品间基因表达丰度的相关性将相似的基因组成一个群。通过对一个群内的基因相互比较分析，我们可以加深对特征不明显和新发现的基因的理解。同时，如果这些基因有极强的相关性也能说明他们可能在转录调控网络中共同参与工作。

1.1 研究现状及发展趋势

分层聚类因为其能比较好的处理数据噪音,同时有较好的生物相关性[1]而被广泛运用于生物信息分析中。随着第二代测序技术的普及和进步，基因测序数据的产出量从100P/S达到了1TP/S，并且产出量还在以五个月增长一倍的速度提高[2]。常见分层聚类实现因为其O(n^3)的计算复杂度难以在生产实践过程中良好运用。寻找一种快速高效的分层聚类实现显然是必须的，而并行化和分布式处理则是加快聚类过程的有效途径。

分层聚类的并行化在Li X.[3]中就有研究。

多核GPU(Graphic Processing Unit 图形处理单元)的出现为高通量并行实现开辟了更加快速高效的新途径。近期在生物信息领域发表的众多论文显示出了GPU实现下性能的巨大提升。对比与CPU，在GPU下五到六倍的性能提升很容易达到，在一些例子中可以达到100多倍的加速比。

已经有很多重要的生物信息分析应用有了GPU版本，比如并行序列组装[4,5]，生物序列数据库扫描[6]，蛋白质特征检测[7]等等。

在聚类领域的GPU实现也有大量文献研究，比如常见的K-means的CUDA实现，

能达到40倍的加速比[8, 9]，针对小世界网络模型的MCL聚类算法(在生物信息中有运用于蛋白质互作网络分析)的CUDA实现[10]。分层聚类的CUDA实现在Chang D J[11]中就有实现。

在大数据下,制约聚类过程在生产实践中运用的不仅仅是其高昂的计算复杂度,同时还有较大的分析难度。数据可视化在数据的理解、数据质量、展示数据的意义、处理异常值等方面都遇到难题[12]。

1.2 意义及价值

在本文中，主要根据肠道基因表达丰度的 profile table 数据利用异构计算和分布式计算技术完成高性能聚类的实现。基因个数选在 0.01M 到 10M 之间，样本数为 1267。这些值已经远远大于常见的 Meta 聚类数据量。比如说在出芽酵母基因组有 6300 个基因，人类基因组有 20000 个基因，通常研究员都只会取其中的一部分[1]。本文完成了对 10M 的基因数、1267 的样本个数这样大数据量的宏基因组特征数据的高性能聚类实现。在程序计算性能上提高了 40~2000 倍，同时对空间复杂度也做了优化。这将大幅度缩短基因组聚类分析所需的时间，提高工作效率。

1.3 本文研究范围及内容组织

第二章：回顾 GPU 计算的历史，并简单介绍 CUDA 的编程模型及编程方法以及 OpenAcc 及 cuBlas，并介绍如何通过使用它们简单的提高程序性能。

第三章：分析项目，包括项目中的输入数据，算法选择及其复杂度。根据数据量及程序结构分析寻找合适的优化方案。

第四章：使用 CUDA 及 MPI 对项目程序进行优化，展示优化结果并对其做一定分析。

第五章：对整个项目及其优化做一个总结，并对未来的研究方向做一定展望。同时讲述项目过程中的收获和体会。

第二章 异构编程相关背景介绍

传统的并行编程模型比如 MPI (Message Passing Interface)、多线程等的扩展性受制于线性化和同步阶段会因为核数的增长而更加复杂。当迁移到 GPU 这种众核架构上时，需要一种新的并行计算模型可以解决不同 GPU 间核数不同的问题。

在 2007 年，NVIDIA 发布了一种具有扩展性的并行编程模型。这个模型使用 C 语言来操作 NVIDIA 的 GPU，这种架构被称作统一计算架构(CUDA, Compute Unified Device Architecture)。随后在 2008 年，基于通用 GPU 的 OpenCL (Open Computing Language)并行编程模型被发布。

统一计算架构提供了一系列基于标准 ANSI C 编程语言的扩展功能，这让同时对 CPU 和 GPU 进行异构计算开发成为可能。也正因为 CPU 和 GPU 适合于各自特定的计算任务，最好的模型就是结合他们的优点来共同完成一个任务。比如说，我们可以使用 CPU 来做线性的计算部分，然后用 GPU 做计算密集的且指令间没有相互依赖的计算部分。

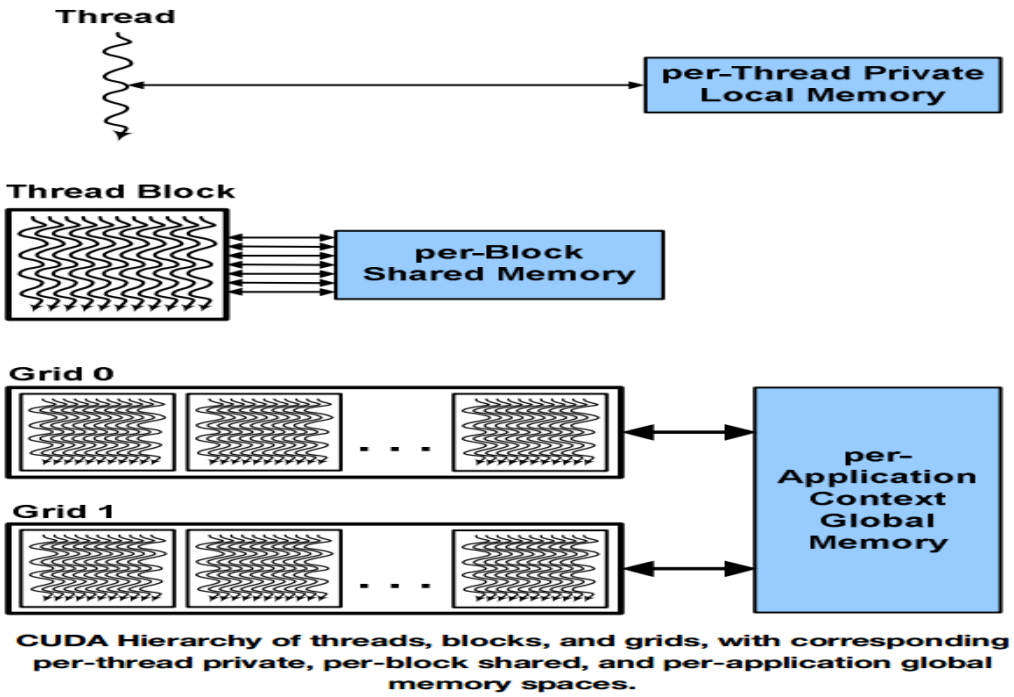


图 2-1 NVIDIA GPU 的层级模型[13]

统一计算架构是一种结合软件和硬件以使 NVIDIA 的 GPU 能够被 C、 C++、

Fortran、 OpenCL、 DirectCompute 和其他语言利用的架构，一个符合统一计算架构的程序调用多个计算内核(Kernel)，一个计算内核会执行多个线程块内的多个线程。

在统一计算架构下，每一个线程都有自己的私有变量，函数和堆栈。每一个线程块有自己的共享存储器用于线程块间交互和数据共享，网络中的所有线程块都可以看到全局显存的数据。

2.1 多核 CPUs 和众核 GPUs 计算的历史

多核一般指 2 到 8 个核，众核一般指几十到几百个核。

1990 年前后，因为单位面积门数(gates-per-die)，时钟频率，指令级并行(ILP, instruction-level parallelism)的提高,处理器性能呈爆炸性增长。但是在 2003 年左右，时钟频率开始因为功耗和热效应达到物理瓶颈，同时复杂的指令级并行实现也成为制约。这种情况下，单位面积门数成为能够提升处理器性能的关键。所以，许多制造商将关注点从时钟频率转移到制造更多的核[14]。

这种多核和众核的发展方向将处理器芯片转变成并行系统。通过放置多个浮动点算数逻辑单元(ALU, arithmetic logic unit)同时对不同的输出做相同的操作来达到更高的计算性能。这种处理方式也叫做单指令多数据流(SIMD, single instruction, multiple data)。普通的单指令多数据实现使用明确的短向量指令，控制四个浮动点

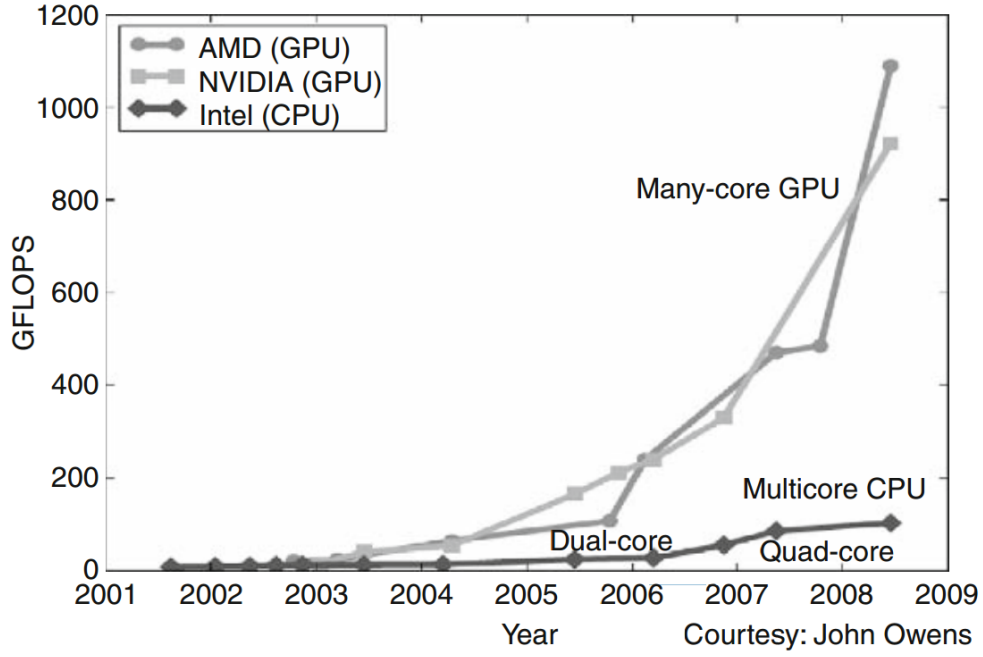


图 2-2 GPU 和 CPU 的性能差距[13]

算数逻辑单元。这种短向量的设计方式常见于多核 CPU，可以在增长带宽的同时保持单线程的计算能力[15]。

现代众核 GPU 的设计采用包括多线程和单指令多数据流方式获得更高的核心利用率。NVIDIA 的 GPU 采用非常宽的 SIMD 处理方式(一般一个 wrap 有 32 个线程).比如 NVIDIA GeForce 的 Tesla M2090 中的核数高达 512，最新的 Kepler 架构下的 Tesla K20 甚至能达到 2880 个单精度核,900MHz 时钟频率和 3.5TFLOPS。相比于高端 CPU Intel Core i5 2500K 的 123.35GFLOPS 的计算峰值，GPU 有接近 30 倍的性能提升空间。从图 2-2 看到，自 2003 年开始，众核 GPU 开始引领计算能力，并且普遍高于同期 CPU 10 倍左右。

2.2 NVIDIA GPU 架构

NVIDIA 从 2008 年的 Tesla 架构开始，以每两年升级一次,一次提升一倍性能的速度发布了 2010 年的 Fermi,2012 年的 Kepler，2014 年的 Maxwell 以及 2016 年将出的 Pascal。接下来将简单描述下各个架构的相似和区别。

2.2.1 Fermi(费米)架构

从 Fermi 开始，NVIDIA 正式使用 CUDA Core 的概念，实际上它就是之前的 Stream Processor（SP，流处理器），为了统一，全文将通称 CUDA Core。Fermi 架构的实现总共有 30 亿个晶体管,超过 512 个 CUDA Core，每个 CUDA Core 在每个时钟下都能执行一个浮点数运算或整数运算。512 个 CUDA Core 被组合成 16 个流处理器(SM, Streaming Multiprocessor)，每个有 32 个核。CPU 和 GPU 间通过 PCI-Express(一种总线和接口的通信标准)通信。

每个 SM 包含 16 个内存 Load/Store（LD/ST，存/取）单元，可以保证源和目标地址在一个周期内同时由 16 个线程来进行操作，支持缓存和 DRAM 的任何位置读取。

除了存取单元外，每个 SM 中还包含有 4 个 SFU（Special Function Units，特殊功能单元），它的作用是处理超越函数，包括 sin、cosine、求倒数、平方根等。每个 SFU 在一个时钟周期内每个线程可以执行一个指令操作，因此每组 warp 执行需要 8 个周期（每组 warp 有 32 个线程）。指令分发器可以按照当前 SFU 的运行情况来分发指令，当一个 SFU 被占用时，可以将指令分发到其他的 SFU 单元处理。

每一个 CUDA Core 都有完整的整数算数逻辑单元(ALU, Arithmetic Logic Unit)和浮点数单元(FPU, Float Point Unit)处理管线。不同于 Tesla 架构下的 IEEE 754-1985 标准，Fermi 架构使用新的 IEEE754-2008 浮点数标准.其他参数见表 2-1。同时，Fermi 也是第一个支持 PTX2.0 指令集的架构。PTX 是一个底层虚拟机和工业标准.在程序安装时，PTX 指令将翻译成适合于特定 GPU 驱动的机器代码.PTX 2.0 对 CUDA 架构的提升包括利用统一寻址空间达到完整的 C++支持，对 OpenCL 和 DirectCompute 的支持,乘加指令(MAD, Multiply-add)和浮点数精度的支持以及通过分支预测提高带分支程序段的性能。

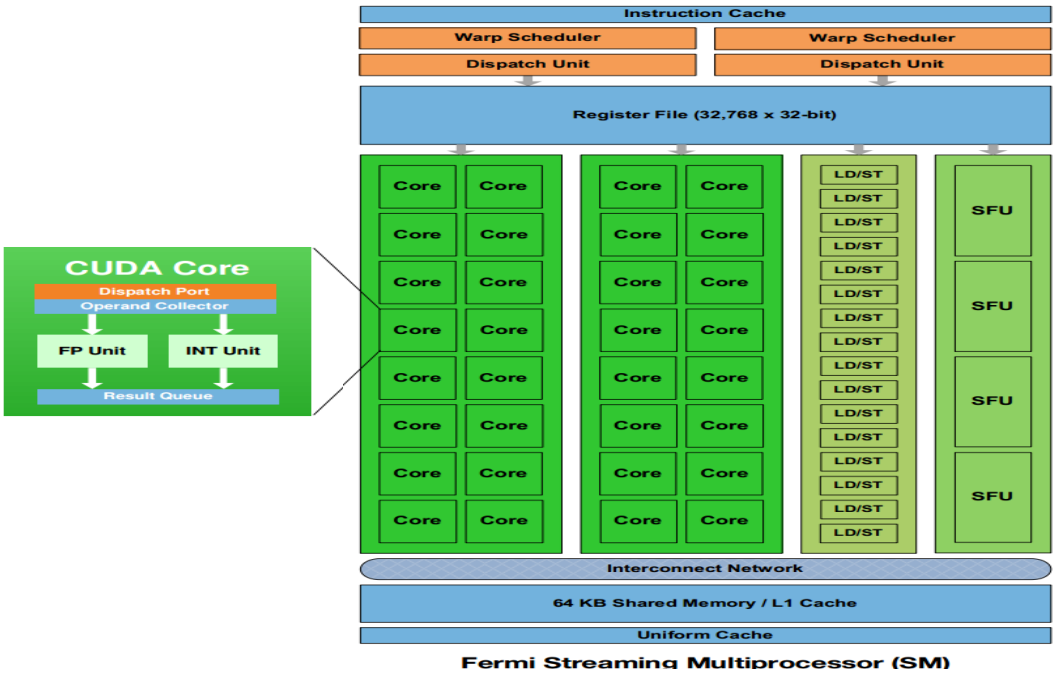


图 2-3 Fermi 架构下的 SM[16]

Fermi 引入了真正的缓存，每组 SM 拥有 64KB 可配置内存(合计 1MB)，可分成 16KB 共享内存加 48KB 一级缓存，或者 48KB 共享内存加 16KB 一级缓存，可灵活满足不同类型程序的需要。

整个芯片拥有一个容量 768KB 的共享二级缓存，执行原子内存操作(AMO)的时候比 GT200 快 5-20 倍。

Fermi 增加了 ECC 功能 ，在大型集群和高可靠性领域中，ECC 是一个重要的特性。这是业界第一款支持 ECC 校验的 GPU。Fermi 的寄存器，共享内存，L1 和 L2 缓存以及显存 DRAM 都支持 ECC 校验，这增加了系统的可靠性。

以前的架构里多种不同载入指令，取决于内存类型：本地(每线程)、共享(每组线程)、全局(每内核)。这就和指针造成了麻烦，程序员不得不费劲清理。

Fermi 统一了寻址空间，简化为一种指令，内存地址取决于存储位置：最低位是本地，然后是共享，剩下的是全局。这种统一寻址空间是支持 C++的必需前提。并且一举支持 64-bit 寻址，即使实际寻址只有 40-bit，支持显存容量最多也可达惊人的 1TB。

2.2.2 Kepler(开普勒)架构

2012 年，NVIDIA 推出 Kepler 的 GK110 标准。Kepler 架构的初衷是利用卓越的电源效率达到计算性能的最大化，就像黄仁勋说的：”数学是免费的，晶体管是免费的。能源是很贵的。效能比才能显示性能。”相对于 Fermi 架构，Kepler 架构有三个最重要的特点。

2.2.2.1 SMX(Streaming Multiprocessor)新一代流式处理器的提出

图 2-4 Kepler 架构下的 SMX[17]

对比图 2-3 和图 2-4，可以看到 SM 和 SMX 无论在计算核心数量还是调度器\缓存大小上的巨大区别。每一个 SMX 单元都集成了 192 个 CUDA Core。为了提高 GPU 计算中的高精度计算性能，SMX 将用于处理快速超越函数计算的特殊功能单

元(SFU，Special Function Unit)的数量提升了八倍。同时，SMX 的计算核心的时钟频率降低为 GPU 的主频频率而不是和 Fermi 一样的两倍主频频率。虽然提高时钟频率可以提高指令吞吐量,但同时会大量提高运行功耗，所以为了达到更好的性能功耗比,在 Kepler 架构中选择了较低的时钟频率。

Warp 调度器也有了较大的变化。在 Kepler 中，每个调度器将发射两个不同的指令到一个 warp 中。同时，为了尽可以减少功耗，NVIDIA 发掘了编译器的能力，将判断数据有效性的处理逻辑从硬件阶段提前到编译期，从而可以将复杂且大功耗的硬件块更换成提取预处理信息的简单电路。

同时,每个线程可利用的最大寄存器数也从 64 个提高到 255 个。这大幅度的提高了 CUDA 的计算性能和应用空间。比如在 QUDA 基于 fp-64 的算法中，仅这项改变就让性能提高了 5.3 倍。

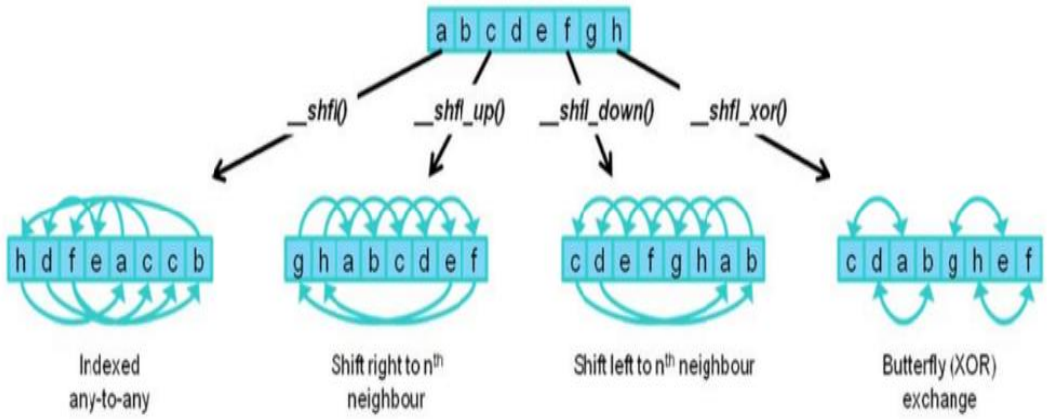


图 2-5 Kepler 架构中的 Shuffle 指令[18]

为了提高性能，Kepler 实现了新的洗牌指令(Shuffle)，如图 2-5。在 Fermi 架构中，为了达到 warp 间通信需要先将数据保存到共享存储器中，然后做同步,最终由各个线程读取。这样不仅耗费指令，而且浪费宝贵的存储器资源。利用 Shuffle 指令集，存储和提取指令都可以一步完成，减少了共享存储器压力。在 FFT 计算中，仅利用 Shuffle 指令就能将计算性能提高 6%。

2.2.2.2 动态并行(Dynamic Parallelism)动态创建工作

在设计 Kepler GK110 架构的总体目标之一是使开发人员更容易更轻松地利用 GPU 的巨大并行处理能力。为此，新的 Dynamic Parallelism 功能，使 Kepler GK110 GPU 能通过应用不返回主机 CPU 的数据而动态创建新线程。

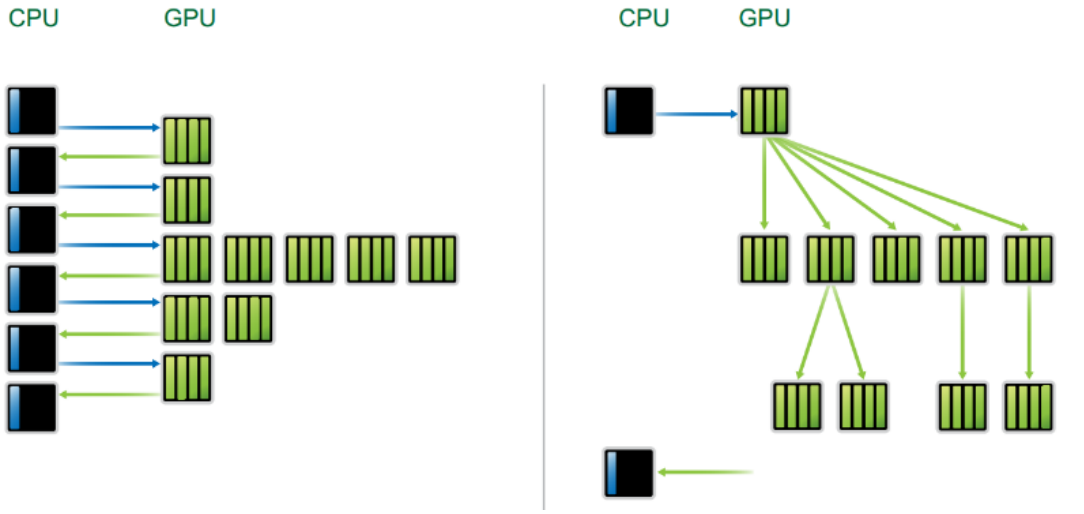


图 2-6 没有 Dynamic Parallelism 的情况下，CPU 启动 GPU 上的每个内核。有了该新功能的情况下，Kepler GK110 GPU 现在可以启动嵌套内核，不需要与 CPU 进行通信[18]。

如图 2-6 所示，任何内核可以启动另一个内核，并创建处理额外的工作所需的必要流程、事件和依赖，而无需主机 CPU 的介入。这种简化的编程模式更易于创建、优化和维护。 它还通过为 GPU 维持与传统 CPU 内核启动工作负载相同的语法，创建了一个程序员友好环境。

2.2.2.3 Hyper-Q 最大化 GPU 资源

Hyper-Q 允许多个 CPU 核同时在单一 GPU 上启动工作，从而大大提高了 GPU 的利用率并削减了 CPU 空闲时间。此功能增加了主机和 Kepler GK110 GPU 之间的连接总数，允许 32 个并发、硬件管理的连接，与 Fermi 相比，Fermi 只允许单个连接。 Hyper-Q 是一种灵活的解决方案，允许 CUDA 流程和消息传递接口（MPI）进程的连接，甚至是进程内的线程的连接。先前被假依赖限制的现有应用程序，可以在不改变任何现有代码的情况下，达到 32 倍的性能提升。

Hyper-Q 在基于 MPI 并行计算机系统中使用会有明显的优势。 通常会在为多核 CPU 系统中运行而创建基于 MPI 的传统算法。由于以 CPU 为基础的系统可以有效处理的工作负载通常比使用的 GPU 处理的较小，所以一般每个 MPI 进程中通过的工作量是不足以完全占据 GPU 处理器。

虽然可以一直发出多个 MPI 进程同时运行在 GPU 上，但是这些进程有可能由于假依赖会成为瓶颈，迫使 GPU 低于最高效率地运行。Hyper-Q 消除了假依赖的瓶颈，并大幅提高了从系统 CPU 将 MPI 进程移动到 GPU 的处理速度。Hyper-Q 必定会是 MPI 应用程序性能提高的驱动。

2.2.3 Maxwell(麦克斯韦)架构

2014 年发布的 Maxwell 架构介绍了一种全新的 SM 架构可以明显提高功效比. 虽然 Kepler 的 SMX 设计已经让这一代 GPU 非常有效率。但是在设计的过程中，NVIDIA 的设计师发现了一个能极大提升功效的机会。

通过对对控制逻辑划分,工作量负载均衡，操控时钟粒度，指令调度，多个时钟周期发射的指令数和其他很多方面的改进，Maxwell 架构下的 SM(官方被称作 SMM)在功效上能够超越 Kepler 架构下的 SMX，如图 2-8。

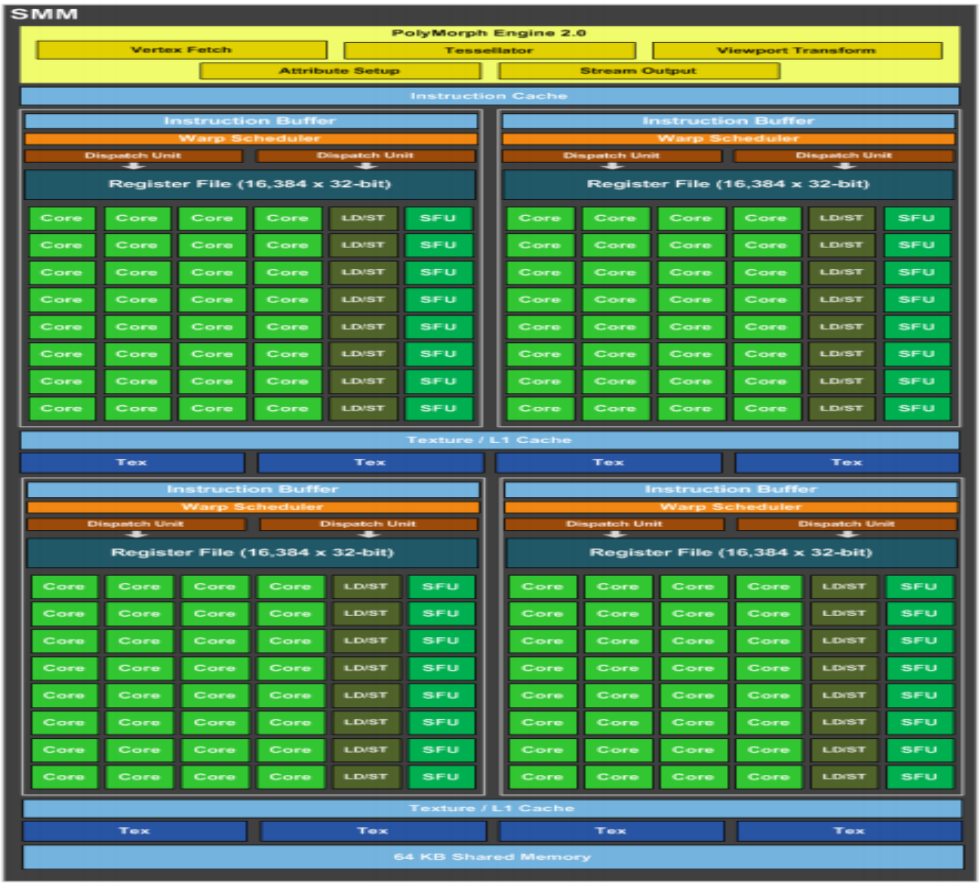


图 2-8 Maxwell 架构下的 SMM[19]

每一个 SMM 中的 CUDA cores 个数变成了 SMX 中的 1/2，Maxwell 的架构提

高了运行效率，单个 SMM 的效率在 SMX 的 90%左右。SMM 同时有相同指令发射能力并且减少了计算延时。

SMM 中的 Warp 调度器个数和 SMX 中一样是 SMM 中的 CUDA core 被分配到对应的 Warp 调度器下。

Maxwell 中一个显著的提升是 SMM 提供了 64KB 的专用的共享存储空间，不是像 Fermi 和 Kepler 一样将 64KB 分配给共享存储器和 L1 Cache。用的存储器能够让 wrap 的占用率得到提升。

同时,Maxwell 提供了原生的共享存储器原子操作支持,而不是像 Fermi 和 Kepler 那样使用加锁/更新/解锁的模式实现。

2.3 架构参数差异

表 2-1 Fermi, Kepler, Maxwell 架构参数差异

GPU	GF116 (Fermi)	GK107 (Kepler)	GM107 (Maxwell)
CUDA Cores	512	384	640
Base Clock	870 MHz	1058 MHz	1020 MHz
GFLOP/s	501.1	812.5	1305.6
共享寄存器	16KB / 48 KB	16KB / 48 KB	64 KB
传输带宽	64GB/s	80 GB/s	86.4 GB/s
L2 缓存	768 KB	256 KB	2048 KB
热设计功耗	105W	64W	60W
晶体管数	11 亿	13 亿	18.7 亿
晶片大小	238 mm^2	118 mm^2	148 mm^2
制程	40 nm	28 nm	28 nm

如表 2-1 所示，在 Fermi、Kepler、Maxwell 的进程中，晶片制程愈来愈小，晶体管数目和每秒浮动点计算量已经翻了一番。传输带宽的增长明显不如计算能力的提升。这也预示着在 GPU 计算中，合理掌控数据传输和指令计算是比较重要的一步。

2.4 CUDA 环境及编程

2.4.1 系统要求

为了在电脑中使用 CUDA,需要满足以下条件:

- CUDA 兼容的 GPU
- Windows XP、Vistal、7 或者 8 或者 linux 系统
- NVIDIA CUDA 开发工具

安装 CUDA 开发工具运行在合适的 Windows 版本上主要由简单的几步组成：

- 验证系统是否有 CUDA-capable GPU
- 下载 the NNIDIA CUDA 工具包
- 安装 NVIDIA CUDA 工具包
- 测试安装的软件运行正确，以及与硬件的通信

目前可用两种接口写 CUDA 程序：CUDA C 和 CUDA 驱动 API。一个应用典型只能使用其中一种，但在遵守限制的情况下，可以同时使用两种。

CUDA C 将 CUDA 编程模型作为 C 的最小扩展集展示出来。任何包含某些扩展的源文件必须使用 nvcc 编译。这些扩展允许程序员像定义 C 函数一样定义内核和每次内核调用时，使用新的语法制定网格和块的尺寸。

CUDA 驱动 API 是一个低层次的 C 接口，它提供了从汇编代码或 CUDA 二进制模块中装在内核，检查内核参数，和发射内核的函数。二进制和汇编代码通常可以通过编译使用 C 写的内核得到。

CUDA C 包含运行时 API，运行时 API 和驱动 API 都提供了分配和释放设备存储器、在主机和内存间传输数据、管理多设备和系统的函数等等。

同时 API 是基于驱动 API 的，初始化、上下文和模块管理都是隐式的，而且代码更简明。CUDA C 也支持设备模拟，这有利于调试。

2.4.2 用 nvcc 编译

内核可以使用 PTX 编写，PTX 就是 CUDA 指令集架构，PTX 参考手册中描述了 PTX。通常 PTX 效率高于像 C 一样的高级语言。无论是使用 PTX 还是高级语言，内核都必须使用 nvcc 编译成二进制代码才能在设备执行。

Nvcc 是一个编译器驱动，简化了 C 或 PTX 的编译流程：它提供了简单熟悉的

命令行选项，同时通过调用一系列实现了不同编译步骤的工具集来执行他们。本节简介了 nvcc 的编译流程和命令选项。完整的描述可在 nvcc 用户手册中找到。

2.4.3 编译流程

nvcc 可编译同时包含主机代码(由主机上执行的代码)和设备代码(在设备上执行的代码)的源文件。Nvcc 的基本流程包括分离主机和设备代码并将设备代码编译成汇编形式(PTX)或/和二进制形式（cubin 对象）。生成的主机代码要么被输出为 C 代码供其它工具编译，要么在编译的最后阶段被 nvcc 调用主机编译器输出为目标代码。

应用能够：

1. 要么在设备上使用 CUDA 驱动 API 装在和执行 PTX 源码或 cubin 对象，同时忽略生成的主机代码。
2. 要么连接到生成的主机代码:生成的主机代码将 PTX 代码和/或 cubin 对象作为已初始化的全局数组导入，将<<<>>>语法转化为必要的函数调用以加载和发射每个已编译的内核。

应用在运行时装载的任何 PTX 代码被设备驱动进一步编译成二进制代码。这称为即时编译。即时编译增加了应用装载时间，但是可以享受编译器的最新改进带来的好处。也是当前应用能够在未来的设备上运行的唯一方式。

2.4.4 PTX 兼容性

一些 PTX 指令只被高计算能力的设备支持。例如，全局存储器上的原子指令只在计算能力 1.1 及以上的设备上支持；双精度指令只在 1.3 及以上的设备上支持。将 C 编译成 PTX 代码时，-arch 编译器选项指定假定的计算能力。因此包含双精度计算的代码，必须使用“-arch=sm_13”（或更高计算能力）编译，否则双精度计算将被降级为单精度计算。

为某些特殊计算能力生成的 PTX 代码始终能够被编译成相等或更高计算能力设备上的二进制代码。

2.4.5 应用兼容性

为了咋子特定计算能力的设备上执行代码，应用加载的二进制或 PTX 代码必须满足如 2.2.节和 3.2.4 节说明的计算能力兼容性。特别的，为了能在将来的更高

计算能力(不能产生二进制代码)的架构上执行，应用必须装载 PTX 代码并为那些设备即时编译。

CUDA C 应用中嵌入的 PTX 和二进制代码，由-arch 和-code 编译器选项或-gencode 编译器选项控制。例如，

```
nvcc x.cu
–gencode arch=compute_10, code=sm_10
–gencode arch=compute_11,code=‘compute_11,sm_11’
```

嵌入与计算能力 1.0 兼容的二进制代码（第一个-gencode 选项）和 PTX 和与计算能力 1.1 兼容的二进制代码。

2.4.6 设备存储器

CUDA 编程模型假定系统包含主机和设备，它们各有自己独立的存储器。内核不能操作设备存储器，所以运行时提供了分配，释放，拷贝设备存储器和在设备和主机间传输数据的函数。

设备存储器可被分配为线性存储器或 CUDA 数组。

CUDA 数组是不透明的存储器层次，为纹理获取做了优化。

计算能力 1.x 的设备，其线性存储器存在于 32 位地址空间内，计算能力 2.0 的设备，其线性存储器存在于 40 位地址空间内，所以独立分配的内存能够通过指针引用，如二叉树。

典型的，线性存储器使用 cudaMalloc()分配，通过 cudaFree()释放，使用 cudaMemcpy()在设备和主机间传输。

线性存储器也可以通过 cudaMallocPitch()和 cudaMalloc3D()分配。在分配 2D 或 3D 数组的时候，推荐使用，因为这些分配增加了合适的填充以满足对齐要求，保证了最佳性能。返回的步长(pitch,stride)必须用于访问数组元素。下面的代码分配了一个尺寸为 width*height 的二维浮点数组，同时演示了怎样在设备代码中遍历数组元素。

2.4.7 共享存储器

共享存储器使用__shared__限定词分配，共享存储器比全局存储器更快。

2.5 例子: CUDA 矩阵乘法的实现

代码由两部分组成，为了尽量简洁，我们省略了与理解程序无关的代码，仅仅罗列出了代码所需要完成的功能。

主机端（Host）代码需要完成以下步骤：

- 分配主存
- 分配显存，初始化 A、B 数组
- 数据传输（将 A,B 数组从主存拷贝到显存）
- 调用 kernel 函数
- 数据传输（将 C 数组从显存拷贝到主存）

设备端（Device）代码要做这些事情：

- 利用 CUDA 提供的内建变量（如 threadIdx）将当前 thread 映射到要操作的数据上
- 矩阵 A 的第 row 行与矩阵 B 的第 col 列进行点积操作，结果保存到矩阵 C 的 (row, col) 位置

2.5.1 显存分配

首先根据矩阵的大小为其分配显存，两个输入矩阵，一个输出矩阵（方便起见，假设矩阵长和宽均为 N）：

```
#define N 15
cudaMalloc((void**)&matA_d, sizeof(int)*N*N);
cudaMalloc((void**)&matB_d, sizeof(int)*N*N);
cudaMalloc((void**)&matC_d, sizeof(int)*N*N);
```

2.5.2 设计 grid 和 block 的维度

假设 N 已定，如何设计 grid 和 block 的维度？（block 和 grid 是 CUDA 编程模型中两个重要的概念，所有 thread 将被组织成 block 块的形式，同一个 block 中的线程可以共享数据、同步；block 又被组织成 grid。）

```
dim3 blocksPerGrid(1, 1);
dim3 threadsPerBlock(16, 16);
```

2.5.3 Kernel 函数

怎么利用 CUDA kernel 函数中提供的内建变量（threadIdx 等）将 thread 映射到矩阵中呢？

我们用以下代码实现：

```
__global__void kernel(int* a, int* b, int* c) {
    int col(0) = threadIdx.x(0) + blockIdx.x(0) * blockDim.x;
    int row(0) = threadIdx.y(0) + blockIdx.y(0) * blockDim.y;
    int sum = 0, i;
    for (i=0; i<N; i(0)++) {
        sum += (a[row*N + i] * b[i*N + col]);
    }
    c[row*N + col] = sum;
}
```

2.6 OpenAcc 及 cuBlas 在 CUDA 中的运用

OpenACC 指令在加速科学代码方面是一种简单而又可移植的方式。利用 OpenACC，只要在自己的 Fortran 或 C 语言代码中插入编译器提示，编译器即可将代码中计算量繁重的部分自动交由 GPU 处理，以实现更高的性能。

现在可以从克雷、CAPS 以及 The Portland Group (PGI) 等行业领军企业处获得 OpenACC 编译器，该编译器具有下列特点：

- 开放：利用这一开放性标准让你的代码永不过时
- 简单：通向并行计算的简单、高级、以编译器驱动 (Compiler driven) 的方法
- 可移植: 非常适合加速传统的 Fortran 或 C 语言程序

NVIDIA CUDA 基本线性代数程序库(cuBlas，Basic Linear Algebra Subroutines)是 GPU 加速的完整的标准 BLAS 库，它相比于最新的 MKL BLAS 实现提供了 6 到 17 倍的加速比。最新的 CUDA 6.0 还支持多 GPU 下的 cuBLAS-XT。特点：

- 对 152 个标准 BLAS 程序的完整支持
- 支持单精度,高精度,复数,高精度复数的数据类型
- 支持 CUDA 流

- Fortran 支持
- 支持多 GPU 和 Kernel 并发执行
- 支持 GEMM 批处理
- 支持 LU 分解批处理
- 支持矩阵逆运算批处理

2.6.1 OpenAcc 介绍

目前有三家厂商提供加速器产品：nVidia GPU、AMD GPU、Intel 至强 Phi 协处理器。三种加速器使用的编程语言分别为 CUDA C/CUDA Fortran、OpenCL 和 MIC 导语。加速器计算有四个技术困难：一是 CUDA/OpenCL 等低级语言编程难度大，且需要深入了解加速器的硬件结构。而大部分的用户不是专业编程人员，学习一门新的编程技术将耗费大量时间、人力、财力。二是加速器的计算模型与 CPU 差别很大，移植旧程序需要几乎完全重写。大量的旧程序在性能优化上已经千锤百炼，稳定性上也久经考验，完全重写是不可完成的任务。三是低级编程语言开发的程序与硬件结构密切相关，硬件升级时必须升级软件，否则将损失性能。而硬件每隔两三年就升级一次，频繁的软件升级将给用户带来巨大负担。四是投资方向难以选择。三种加速器均有自己独特的编程语言，且互不兼容。用户在投资建设硬件平台、选择软件开发语言时就会陷入困境，不知三种设备中哪个会在将来胜出。如果所选加速器将来落败，将会带来巨大损失；而犹豫不决又将错过技术变革的历史机遇。

OpenACC 可以克服这四个困难。OpenACC 应用编程接口的机制是，编译器根据作者的意图自动产生低级语言代码。程序员只需要在原程序中添加少量导语，无须学习新的编程语言和加速器硬件知识，可以迅速掌握。只在原始程序上添加少量导语，不破坏原代码，开发速度快，既可并行执行又可恢复串行执行。在硬件更新时，重新编译一次代码即可，不必手工修改代码。OpenACC 标准制定时就考虑了目前及将来的多种加速器产品，同一份代码可以在多种加速器设备上编译、运行，零代价切换硬件平台。

2.6.1.1 执行模型

OpenACC 编译器的执行模型是主机指导加速器设备(如 GPU)的运行。主机执行用户应用的大部分代码，并将计算密集型区域卸载到加速器上执行。设备上执行

并行区域和内核区域，并行区域通常包含一个或多个工作分担(work-sharing)循环，内核区域通常包含一个或多个被作为内核执行的循环。即使在加速器负责的区域，主机也必须精心安排程序的运行：在加速器设备上分配存储空间、初始化数据传输、将代码发送到加速器上、给并行区域传递参数、为设备端代码排队、等待完成、将结果传回主机、释放存储空间。大多数据时候，主机可以将设备上的所有操作排成一队，一个接一个的顺序执行。

目前大多数加速器支持二到三层并行。大多数加速器支持粗粒度并：执行单元完全并行地执行。加速器可能有限支持粗粒度并行操作间的同步。许多加速器也支持细粒度并行：在单个执行单元上执行多个线程，这些线程可以快速地切换，从而可以忍受长时间的存储操作延时。最后，大多数加速器也支持每个执行单元内的单指令多数据(SIMD)操作或向量操作。既然设备端上的执行模型包含多个并行层次，程序员就需要理解它们之间的区别。例如，一个完全并行的循环和一个可向量化但要求语句间同步的循环之间的区别。一个完全并行的循环可以用粗粒度并行执行。有依赖关系的循环要么适当分割以允许粗粒度并行，要么在单个执行单元上以细粒度并行、向量并行或串行执行。

2.6.1.2 存储模型

一个仅在主机上运行的程序与一个在主机+加速器上运行的程序，它们最大的区别在于加速器上的内存可能与主机内存完全分离。例如目前大多数 GPU 就是这样。这种情况下，设备内存可能无法被主机直接读写，因为它没有被映射到主机的虚拟存储空间。主机内存与设备内存间的所有数据移动必须由主机完成，主机调用运行时库在相互分离的内存间显式地移动数据。数据移动通常采用直接内存访问(Direct Memory Access, DMA)技术。类似地，不能假定加速器能读写主机内存，尽管有些加速器设备支持这样的操作。

在 CUDA C 和 OpenCL 等低层级加速器编程语言中，主机和加速器存储器分离的概念非常明确，内存间移动数据的语句甚至占据大部分用户代码。在 OpenACC 模型中，内存间的数据移动是隐式的，编译器根据程序员的导语管理这些数据移动。然而程序员必须了解背后这些相互分离的内存，有包含但不限于以下理由：

- 有效加速一个区域的代码需要较高的计算密度，而计算密度的高低取决于主机内存与设备内存的存储带宽；
- 设备内存空间有限，因此操作大量数据的代码不能卸载到设备上。

在加速器一端，一些加速器(例如目前的 GPU)使用一个较弱的存储模型。特别地，它们不支持不同执行单元上操作的内存一致性；甚至，在同一个执行单元上，只有在存储操作语句之间显式地同步才能保证内存一致性。否则，如果一个操作更新一个内存地址而另一个操作读取同一个地址，或者两个操作向同一个位置存入数据，那硬件可能不保证每次运行都能得到相同的结果。尽管编译器可以检测到一些这样的潜在错误，但仍有可能编写出一个产生不一致数值结果的加速器并行区域或内核区域。

目前，一些加速器有一块软件管理的缓存，一些加速器有多块硬件管理的缓存。大多数加速器具有仅在特定情形下使用的硬件缓存，并且仅限于存放只读数据。在 CUDA C 和 OpenCL 等低层级语言的编程模型中，这些缓存交由程序员管理。在 OpenACC 模型中，编译器会根据程序员的导语管理这些缓存。

2.6.2 cuBlas 介绍

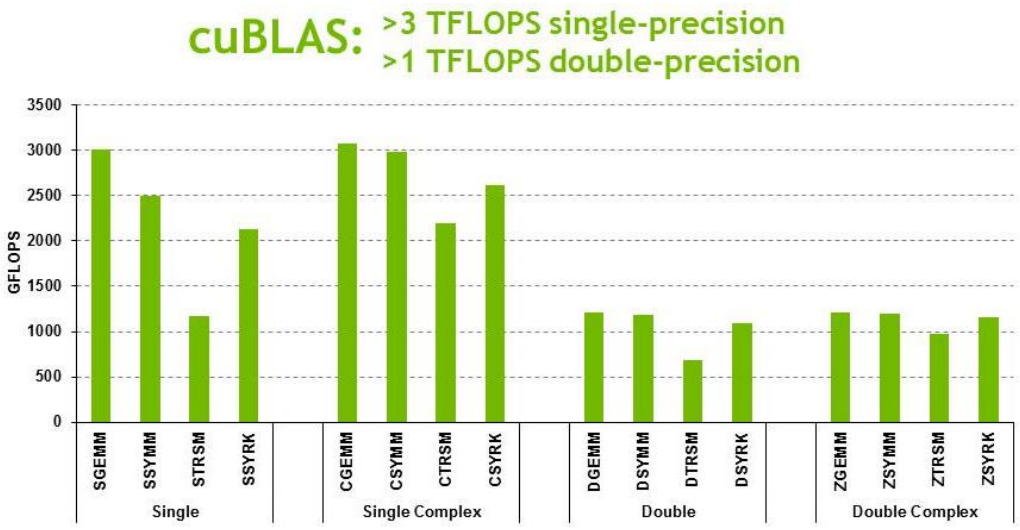


图 2-9 cuBlas 的性能参数[20]

Cublas Library 就是在 NVIDIA CUDA 中实现 Blas(基本线性代数程序)。它允许用户访问 NVIDIA 中 GPU（图形处理单元）的计算资源，但不能同时对多个 GPU 进行自动并行访问.在使用 Cublas library 时，应用程序必须在 GPU 内存空间中分配所需的矩阵向量空间，并将其填充数据，再顺序调用所需的 Cublas 函数，最后将计算结果从 GPU 内存空间上传到主机中。Cublas library 同时还提供了从 GPU 中书写和检索数据的功能。

2.6.2.1 数据布局

对于现有的具有最大兼容性的 Fortran 环境，Cublas library 使用 column-major storage(列主序存储)和 1-based indexing(以 1 开始索引)。由于 C 和 C++使用 row-major storage, 使得用这些语言编写的应用程序对于二维数组不能使用本地数组语义。相反，宏或内联函数应该被定义为实现以为数组矩阵。将 Fortran 代码机械的移植到 C 中，你可以选择保留 1-based indexing 以避免转换循环。在这种情况下，矩阵中位于 i 行 j 列的数组元素可通过下面的宏计算：

```
#define IDX2F(i,j,ld) ((((j)-1)*(ld))+((i)-1))
```

这里，ld 代表矩阵的主要维度，而在列主序存储矩阵中代表行数。在本地写 C 或 C++代码，我们喜欢用 0-based indexing, 那么上述矩阵中元素可通过下面宏计算：

```
#define IDX2C(i,j,ld) (((j)*(ld))+(i))
```

2.6.2.2 应用程序接口

从第四版开始，cuBlas Library 提供了一个升级的 API，另外还有已有的 legacy API。原先的 API 的区别:

- 利用函数初始化了对 Cublas Library 内容的处理并明确的传递给每个后续的库函数调用，这使得用户在使用多主机线程和多个 GPU 时能对库的设置有更多的控制。这也使得 CUBLAS API 成为可重入的。
- 标量 α 和 β 可通过引用传递给主机或设备，替代原先只能通过主机数值进行传递.这种改变使得库函数在使用流时可以异步地并行执行，即使当 α 和 β 由前一个内核所产生。

第三章 项目分析

本节通过对程序的输入数据、所使用的算法及其复杂度做基于性能优化的分析，为之后的具体优化过程做铺垫。

3.1 输入数据分析

3.1.1 输入数据格式

	158256496-stool1					158337416-stool1					158337416-stool2					158458797-stool1				
1	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	6	0
2	290	0	0	0	0	54	322	0	0	0	0	0	0	0	0	0	0	97	623	0
3	72	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	21	216	0
4	104	0	0	0	9	0	0	0	2	4	0	0	28	0	11	0	0	13	43	1
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	192	235	136	24	109	12	66	178	775	1354	0	0	290	5	135	71	803	9	183	
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	14	450	0	0	0	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	4	0	0	0	0	0	0	0	0	0	1	10	16	121	0	
19	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0	0	0	

图 3-1 宏基因组聚类输入数据

如图 3-1 所示，聚类程序的输入数据是 N*M 的矩阵。其中第一列是基因序号，第一行是样本名字，其对应的一列为各个基因在这个样本中的丰度估计。比如第 6 行第 3 列中的数字 136，表示在 158337416-stool2 这个样本中序号为 6 的基因的丰度估计是 136。如果将每一行作为一个聚类分析中的一个测试点，就可以分析出不同基因的性质异同。如果将每一列作为聚类分析中的一个测试点，就可以分析出不同基因采集地(即样本)的性质异同。

3.1.2 数据特性及需求分析

假设输入的数据中的样本数位 M，每个样本中的基因数位 N，那么 N*M 个数据组成了如图 3-1 的矩阵。为了更有针对性的选择聚类算法并对聚类程序做优化，我们需要对输入的数据做特性分析。

观察输入数据，我们可以得到以下这些特征：

1. 无法轻易找到符合这个数据集的聚类模型。在数据分布情况事先完全不清楚的情况下，一个好的聚类算法需要尽可能少的依赖先验知识[7]。
2. 基因数据是由复杂的实验流程产生的，其中有大量的噪声[8]。一些聚类算法对于这样的数据敏感，可能导致低质量的聚类结果。所以应该选择适当的聚类算法。
3. 数据矩阵非常稀疏。 这可能预示着数据量的大小会导致存储空间的瓶颈。
4. 输入的数据是基因的丰度估计而不是一般情况下的相关性系数。在这种情况下，我们需要准确选择合适的相关性系数计算方式。
5. 基因的丰度分布非常不均匀，且有较大的数值变化。这可能预示着在计算或传输过程中会有严重的负载不均衡现象。

同时，实际生产实习过程中，我们发现为了达到理想的结果，还有以下需求：

1. 一种基因可能会同时在多个类中，即这种基因可能对多个类都有较强的相关性。因此需要选择有密度模型的聚类算法。

3.2 聚类算法分析

根据 3.1.2 的数据特性分析可以剔除一些与分析的结果冲突的聚类算法，并最终选择合适的算法。

对宏基因组的聚类算法研究较多的是 K-means，层次聚类，SOM 和运用广泛的高斯混合聚类模型。因为不知道符合数据集的聚类模型，所以 K-means 系列的算法因为需要事先知道聚类个数而不能被使用。同样，类似马尔可夫聚类算法(MCL, Markov Cluster Algorithm)等需要满足小世界模型的聚类算法也不适合在这里被应用。

因为输入数据可能会有较大的噪声，为了解决这一点，使用层次聚类的全链聚类方式(两个簇的相似性定义为基于这两个簇的最小相似性)可以让聚类算法对噪音和离群点不敏感。

同时，为了解决一种基因属于多个类的问题，需要引入一种基于密度的聚类方式。高斯混合模型的聚类属于软聚类方法（一个观测量按概率属于各个类，而不是完全属于某个类），各点的后验概率提示了各数据点属于各个类的可能性。

聚类分析的数学描述是根据给定的数据集$T = \{x_i | x_i \in X, i = 1, \cdots, N\}$,要求寻找 T 上的一个“好”的划分(划分成 m 个类； m 可以是已知的，也可以是未知的)，满足约束条件：

1) $T = \bigcup_{i=1}^{m} C_i$
2) $C_i \neq \emptyset, i = 1, \cdots, m$
3) $C_i \cap C_j = \emptyset, i \neq j, i, j = 1, \cdots, m$

聚类分析的算法可以分为划分法（Partitioning Methods）、层次法（Hierarchical Methods）、基于密度的方法（density-based methods）、基于网格的方法（grid-based methods）、基于模型的方法（Model-Based Methods）。

3.2.1 层次聚类算法

层次化聚类方法就是将数据对象组成一棵聚类的树。根据层次分解是自底向上生成还是自顶向下生成,层次的聚类方法可以细分为凝聚的(agglomerative)和分裂的(divisive)层次聚类。

凝聚的层次聚类：凝聚的层次聚类是自底向上的策略。首先将每个对象作为一个类，然后合并这些原子类为越来越大的类，直到所有的对象都在一个类中，或者某个终结条件被满足。分裂的层次聚类是种自顶向下的策略与凝聚的层次聚类相反，它首先将所有对象置于一个类中,然后逐渐细分为越来越小的类，直到每个对

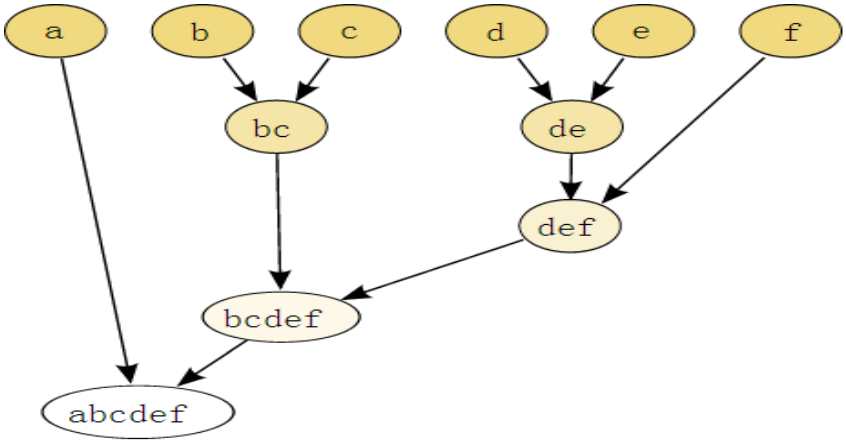


图 3-1 凝聚的层次聚类方式. 如图所示, 第一次将(b, c), (d, e)聚类, 第二次将(de, f)聚类, 第三次将(bc, def)聚类, 最终将(a, bcdef)聚类。

象自成一类，或者达到了某个终结条件，例如达到了某个希望的类数目,或者两个最近的类之间的距离超过了某个闽值。绝大多数聚类方法属于这一类，它们只是在簇间相似度的定义上有所不同。

分裂的层次聚类：这种自顶向下的策略与凝聚的层次聚类相反，它首先将所有对象置于一个簇中，然后逐渐细分为越来越小的簇,直到每个对象自成一簇，或者达到了某个终止条件，例如达到了某个希望的簇数目,或者两个最近的簇之间的距离超过了某个闽值。

3.2.2 高斯混合模型聚类

高斯混合模型[21]（Gaussian mixture model，简称 GMM）是单一高斯密度函数的延伸，由于 GMM 能够平滑地近似任意形状的密度分布，因此近年来常被用在数据挖掘中。

3.2.2.1 单一高斯概率密度函数的参数估计

假设我们有一组在高维空间(维度为 d)的点$x_i, i = 1 \cdots n$,若这些点的分布近似于椭球状,则我们可用高斯密度函数$g(x; \mu, \sum)$来描述产生这些点的概率密度函数:

$$g(x; \mu, \sum) = \frac{1}{\sqrt{(2\pi)^d |\sum|}} \exp\left[-\frac{1}{2}(x-\mu)^T \sum^{-1} (x-\mu)\right] \qquad (3-1)$$

其中μ代表此密度函数的中心点，$\sum$则代表此密度函数的协方差矩阵(Covariance Matrix)，这些参数决定了此密度函数的特性，如函数形状的中心点\宽窄及走向等。

欲求的最佳的参数来描述所观察到的资料点，可由最佳可能性估测法的概念来求得。在上述高斯密度函数的假设下，当$x = x_i$时，其概率密度为$g(x; \mu, \sum)$，若我们假设$x_i, i = 1 \cdots n$之间为相互独立事件，则发生$X = \{x_1, x_2 \cdots x_n\}$的几率密度为

$$p(X; \mu, \sum) = \prod_{i=1}^{n} g(x; \mu, \sum) \qquad (3-2)$$

由于 X 是已经发生的事件，因此我们希望找出$\mu, \sum$的值，使得$p(X; \mu, \sum)$能有最大值，此种估测参数$(\mu, \sum)$的方式，即称为最佳可能性估测法(MLE，maximum Likelihood Estimation)

欲求得$p(X;\mu,\sum)$的最大值，我们通常将之转化为求下列$J(\mu,\sum)$的最大值：

$$\begin{aligned} J(\mu,\sum) &= \ln p(X;\mu,\sum) \\ &= \ln[\prod_{i=1}^{n} g(x;\mu,\sum)] \\ &= \sum_{i=1}^{n} lng(x_i;\mu,\sum) \\ &= \sum_{i=1}^{n}[-\frac{d}{2}\ln(2\pi)-\frac{1}{2}ln|\sum|-\frac{1}{2}(x_i-\mu)^T\sum^{-1}(x_i-\mu)\,] \\ &= -\frac{nd}{2}\ln(2\pi)-\frac{n}{2}ln|\sum|-\frac{1}{2}\sum_{i=1}^{n}[(x_i-\mu)^T\sum^{-1}(x_i-\mu)] \end{aligned} \quad (3-3)$$

欲求最佳的μ值，直接求$J(\mu,\sum)$对μ的微分即可：

$$\begin{aligned} \nabla_\mu J(\mu,\sum) &= -\frac{1}{2}\sum_{i=1}^{n}\left[-2\sum(x_i-\mu)^{-1}\right] \\ &= \sum(\sum_{i=1}^{n}x_i-n\mu)^{-1} \end{aligned} \quad (3-4)$$

令上式等于零，我们就可以得到：

$$\hat{\mu}=\frac{1}{2}\sum_{i=1}^{n}x_i \quad (3-5)$$

欲求最佳的$\sum$值，就不是那么容易了，需要经过较为繁杂的运算，最终可以得到：

$$\hat{\sum}=\frac{1}{n-1}\sum_{i-1}^{n}(x_i-\hat{\mu})(x_i-\hat{\mu})^T \quad (3-6)$$

3.2.2.2 混合高斯概率密度函数的参数估计

如果我们的样本$X=\{x_1,x_2\cdots x_n\}$在 d 维空间中的分布不是椭球状，那么就不适合以一个简单的高斯密度函数来描述这些样本点的概率密度函数。此时的变通方案，就是采用数个高斯函数的加权平均(Weighted Average)来表示。若以三个搞死函数来表示，则可表示成：

$$p(x)=\alpha_1 g(x;\mu_1,\sum_1)+\alpha_2 g(x;\mu_2,\sum_2)+\alpha_3 g(x;\mu_3,\sum_3) \quad (3-7)$$

这个概率密度函数的参数是$(\alpha_1,\alpha_2,\alpha_3,\mu_1,\mu_2,\mu_3,\sum_1,\sum_2,\sum_3)$,而且$\alpha_1,\alpha_2,\alpha_3$要满足下列条件：

$$\alpha_1+\alpha_2+\alpha_3=1 \quad (3-8)$$

以此种方式表示的概率密度函数被称为高斯混合密度函数，或者高斯混合模型(Gaussian Mixture Model)，简称 GMM。

通过复杂的数学运算，我们可以得到计算概率密度函数参数的迭代算法:

1. 设定一个起始参数值$\theta = [\alpha_1, \alpha_2, \alpha_3, \mu_1, \mu_2, \mu_3, \sum_1, \sum_2, \sum_3]$。
2. 使用θ来计算$\beta_2(x_i)$、$\beta_2(x_i)$、$\beta_3(x_i)$，$i = 1 \sim n$
3. 计算新的μ_j值：$\tilde{\mu}_j = \frac{\sum_{i=1}^{n} \beta(x) x}{\sum_{i=1}^{n} \beta(x)}$
4. 计算新的σ值：$\tilde{\sigma} = \frac{1}{d} \frac{\sum_{i=1}^{n} \beta(x)(x_i - \hat{\mu})^T (x_i - \hat{\mu})}{\sum_{i=1}^{n} \beta(x) x}$
5. 计算新的$\tilde{\alpha_j} = \frac{1}{n} \sum_{i=1}^{n} \beta(x)$
6. 令$\tilde{\theta} = [\alpha_1, \alpha_2, \alpha_3, \mu_1, \mu_2, \mu_3, \sum_1, \sum_2, \sum_3]$，若$\|\theta - \tilde{\theta}\|$小于一个极小的容忍值，则停止。否则令$\theta = \tilde{\theta}$并跳回步骤 2。

上述迭代方法一定会让$J(\mu, \sum)$逐步增长，并收敛至一个局部最大值(Local Maximum)。

3.3 基因聚类的相似度计算方式分析

每条基因特征数据都是一个 M 维向量，计算基因特征数据的相似度就是计算两个向量之间的相似度。可以使用欧几里距离，Pearson correlation，Kendall 相关等来作为相似度的度量方法。

3.3.1 欧氏距离

欧氏距离[5]将向量看作 M 维空间的点，两点的相似度为两向量的空间距离。但在宏基因组特征数据聚类中更看重两基因间样本的模式关系。

$$d = \sqrt{\sum_{i=1}^{M} (x_{1i} - x_{2i})^2} \qquad (3-9)$$

3.3.2 皮尔森(Pearson)相关系数

Pearson correlation 假设样本数据分布满足高斯模型，那么知道计算样本的均值就计算两两基因在该均值分布下的相似度[6]。但是宏基因组聚类的特征数据可能并不满足高斯分布，使用 Pearson correlation 也不具备鲁棒性。

$$r = \frac{1}{n-1}\sum_{i=1}^{n}\left(\frac{X_i - \bar{X}}{s_X}\right)\left(\frac{Y_i - \bar{Y}}{s_Y}\right) \quad (3-10)$$

3.3.3 肯德尔(Kendall)和谐系数

为了能够尽量减少实验噪音对聚类的影响，课题使用 Kendall 和谐系数作为两基因相关性的依据[7]。肯德尔和谐系数是计算多个等级变量相关程度的一种相关量。对于 X，Y 两个基因的两对样本值 X_i,Y_i 和 X_j,Y_j,如果 $X_i<Y_i$ 并且 $X_j<Y_j$,或者 $X_i>Y_i$ 并且 $X_j>Y_j$，则称这两对观察值是和谐的，否则就是不和谐的。

Kendall 相关系数的计算公式如下：

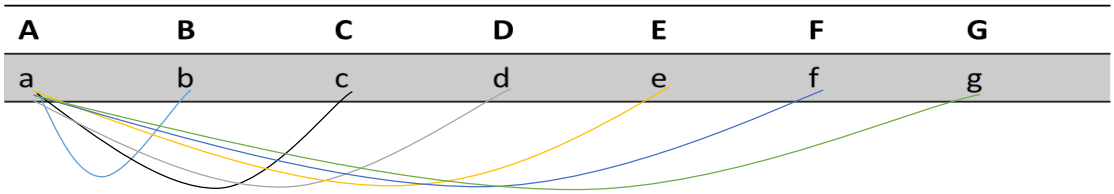


图 3-2 当且仅当 a>b 且 A>B 或者 a<b 且 A<B，那么称(ab, AB)是和谐的。反之，则是不和谐的。

$$\tau = \frac{(\text{和谐数对数} - \text{不和谐数对数}) * 2}{m*(m-1)} \quad (3-11)$$

如图 3-1 中的输入数据，两个基因丰度估计的大小关系并不能简单通过数值来比较。参照 Audic S. 等人发表在 Genome Research 上的数字化基因表达谱差异基因检测方法，我们使用差异检验来判断两个基因丰度的大小关系。

设 N 为样本中总的数值，x，y 为比较的两个基因丰度估计，这个 p 值小于 1%时，证明两个样本的大小关系是显著的，否则则认为 x，y 两个值相等。验证公式为：

$$P(y|x) = \left(\frac{N_2}{N_1}\right)^{y} \frac{(x+y)!}{x!\,y!\left(1+\frac{N_2}{N_1}\right)^{x+y+1}} \quad (3-12)$$

3.4 时间复杂度分析

现在整个程序结构被分为三大块。

1) 相关系数计算
2) 层次聚类

3) 高斯混合模型聚类

我们将分别对这三大块做时间复杂度分析以确定接下来优化的重心。假设有 M 个样本，每个样本有 N 个基因数据。

3.4.1 相关系数计算

如图 3-2 所示，肯德尔相关系数的计算需要基因丰度数据两两比较，N*(N-1)步。每一步计算需要有 M*(M-1)次大小关系比较，每一步比较又需要一个复杂度依赖于丰度估计数值C的有效性验证。在实际数据中，这个数值最大有 20000 多，但多数为 0。

发现这里可以争取到一次优化的机会，即大小关系的有效性在一种基因数据中被多次计算，因此我们只要提前计算任意两两基因之间的大小关系并保存起来就能减小总的复杂度。

我们假设每一步比较的计算复杂度为 C，那么计算相关系数的计算复杂度为：

$$\begin{aligned} \mathrm{P} &= \mathrm{N} * (\mathrm{N}-1) * M * (M-1) + N * C * M * (M-1) \\ &\approx \mathrm{N}^2 * M^2 + \mathrm{N} * \mathrm{M}^2 * \mathrm{C} \end{aligned} \tag{3-13}$$

3.4.2 层次聚类过程

层次聚类依次将最相似的两组基因合并，同时更新相似性关系。这个过程中需要迭代 N 次，选择相似的组的复杂度为 O(N)，更新单连接的复杂度为 O(N)，更新完全连接的复杂度为 O(N^2)，所以层次聚类的实现在 O(N^2)到 O(N^3)之间，依实际情况而言。

3.4.3 高斯混合模型聚类过程

可以发现，根据 3.3.3 所使用的 Kendall 和谐系数作为相似度判断依据后，会损失很多数据。因此，需要将从层次聚类得到的结果结合宏基因组特征数据做高斯混合模型聚类优化之前的聚类结果，将层次聚类未完成的聚类再聚在一起。

高斯混合模型聚类算法需要使用 EM 迭代算法。

在 EM 算法 E 步骤需要计算一个 M*M 协方差矩阵的行列式，这个计算过程的复杂度是$\mathrm{O}(\mathrm{M}^3)$。假设有 K 个簇和 N 种基因参与计算，那么整个 E 步骤的复杂度高达$\mathrm{O}(\mathrm{M}^3 * \mathrm{N} * \mathrm{K})$。在 M 步骤的更新参数过程中需要的计算量分别为$\mathrm{O}(\mathrm{K} * \mathrm{N}), \mathrm{O}(\mathrm{K} * \mathrm{N})$和 $\mathrm{O}(\mathrm{K} * \mathrm{M} * \mathrm{N})$。当 N=10^6，M=10^3 时，复杂度约为 O(K*10^15)。因此，高斯混合模型聚类算法的复杂度是：

$$P = M^3 * N * K + K * N * 2 + K * M * N \approx M^3 * N * K \quad (3-14)$$

3.5 空间复杂度分析

在 3.4.1 节中，我们通过保存大小关系将计算复杂度降低为原来的$\frac{1}{10^3}$左右，但同时我们需要申请 M*M*N 个字节的空间。假设 N=10^6，M=1000，所需要的空间为：

$$S = M * M * N = 10^{12} byte \approx 900\ Gigabyte \quad (3-15)$$

第四章 项目优化

本章通过性能分析工具对原始程序的计算性能做评估，并使用包括 OpenAcc 和 MPI 技术完成了对聚类项目的性能优化。

4.1 热点分析

通过 6.4 的时间复杂度分析和 6.5 空间复杂度分析，我们发现相关性系数计算和高斯混合模型聚类这块的时间复杂度较高，空间复杂度不太理想。因此，这次课题的主要任务是对这两个步骤做性能优化。

CUDA 程序的 profilers 到 cuda5.0 为止有四种[22]：

- ntegrated into NVIDIA® Nsight™ Eclipse Edition (nsight)
- NVIDIA® Nsight™ Visual Studio Edition
- nvprof（Command-line）
- Command-line, controlled by environment variables

前三种都只在 cuda5.0 所支持的驱动下能够完整运行。其中 Nsight Eclipse 为 windows 和 linux 平台，nsight visual stdio 为 windoows 平台，nvprof 和 command line 为 linux 平台。

Command line profiler 可以用 COMPUTE_PROFILE = 1 的环境变量打开，使用 COMPUTE_PROFILE_CSV = 1 控制输出格式，用 COMPUTE_PROFILE_CONFIG = filename.config 来控制输出内容。

输出的内容有函数名，gpu 时间，cpu 时间，SM 占有率等，输出格式如下：

```
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 Tesla C2075
# CUDA_CONTEXT 1
# TIMESTAMPFACTOR fffff6de60e24570
method, gputime, cputime, occupancy
method=[ memcpyHtoD ] gputime=[ 80.640 ] cputime=[ 278.000 ]
method=[ memcpyHtoD ] gputime=[ 79.552 ] cputime=[ 237.000 ]
method=[ _Z6VecAddPKfS0_Pfi ] gputime=[ 5.760 ] cputime=[ 18.000 ] occupancy=[ 1.000 ]
method=[ memcpyDtoH ] gputime=[ 97.472 ] cputime=[ 647.000 ]
```

4.2 优化方式分析

4.2.1 相似度计算

发现相似度计算步骤在时间和空间复杂度上都存在瓶颈。

程序主体是一个三重循环框架，计算密度很大，计算内容上相互独立，很容易做到计算并行。所以考虑使用 OpenACC 对其加速。同时使用 CUDA 作为参照。

但是检验过程的中间结果需要非常的存储空间，发现其实只要存储差异是否显著以及大小关系就行，因此考虑使用位压缩技术。

考虑到检验过程的高次幂运算必然会导致精度问题，使用对其求导的方式将幂运算转变成乘法运算，大大降低了运算难度。

同时考虑到计算大循环的独立性，因此使用 MPI+CUDA 的计算架构使用分布式异构计算技术达到更大的加速能力。

4.2.2 高斯混合模型计算

高斯混合模型的性能瓶颈在协方差矩阵的计算上，简单使用CUDA配合cuBlas高性能线性代数库就能起到非常好的加速效果。因为所在试验中样本数为常数1267，远小于 n。考虑 n*m 矩阵有$10^6 * 10^3 = 10^9$个元素，整个计算大约需要 10G 左右的存储空间。所以在特殊情况下只能根据矩阵性质，将矩阵乘法分解,有以下结论：

$$\begin{matrix} A1 \\ A*B = A2*B \\ A3 \end{matrix} = \begin{matrix} A1*B \\ A2*B \\ A3*B \end{matrix} \qquad (4-1)$$

4.3 实现方式

本节展示如何使用 OpenACC 对分层聚类中的相关系数过程，使用 cuBlas 对及高斯混合模型聚类过程进行加速。

4.3.1 OpenAcc 编程方法

4.3.1.1 导语

在 C 和 C++中，用语言本身提供的#pragma 机制来引入 OpenACC 导语。在

Fortran 中，用带有独特前导符的注释来引入 OpenACC 导语。如果不支持或者关闭了 OpenACC 功能，编译器将会忽略 OpenACC 导语。

在 C/C++中，使用#pragma 机制指定 OpenACC 导语，语法是：

```
#pragma acc  导语名字[子语[[,]子语]...]
```

每个导语都以#pragma acc 开始。导语的其它部分都遵守 C/C++中 pragma 的使用规范。#pragma 的前后都可以使用空白字符；导语中使用空白字符来分隔各字段。#pragma 后面的预处理标记使用宏替换。导语区分大小写。一个 OpenACC 导语作用于紧接着的语句、结构块和循环。

4.3.1.2 加速器计算构件

1. parallel 构件

在 C 和 C++中，OpenACC parallel 导语的语法为

```
#pragma acc parallel 子语, [[,]子语]...]
```

当遇到一个加速器 parallel 构件，程序就创建多个 gang 来执行这个加速器并行区域，而每个 gang 又包含多个 worker。这些 gang 一旦创建，gang 的数量和每个 gang 包含的 worker 数量在整个并行区域内都保持不变。接着，每个 gang 中的所有 worker 都开始执行构件中结构块的代码。

如果没有使用 async 子句，加速器并行区域结束处将会有一个隐式屏障，此时主机处于等待状态，直至所有的 gang 都执行完毕。

如果一个聚合数据类型的变量(或数组)在 parallel 构件中被引用，并且没有出现在该 parallel 构件的数据子语中，也没有包含在任何 data 构件中，那么它等同于出现在 parallel 构件的 present_or_copy 子语中 1。如果一个标量变量在 parallel 构件中被引用，并且没有出现在该 parallel 构件的数据子语中，也没有包含在任何 data 构件中，那么它等同于出现在 parallel 构件的 private 子语(如果没有参数传入也没有结果返回)中或 copy 子语中。

2. kernels 构件

C 和 C++中，OpenACC kernels 导语的语法是：

```
#pragma acc kernels[子语[[,]  子语]…]换行
```

编译器将 kernels 区域内的代码分割为一系列的加速器 kernel。通常，每个循环成为一个单独的 kernel。当程序遇到一个 kernels 构件时，它在设备上按顺序启动这一系列 kernel。对不同的 kernel, gang 的数量、每个 gang 包含多少个 worker、

vector 的长度以及三者的组织方式都可能不相同。

如果没有使用 async 子语，kernels 区域结束时将有一个隐式屏障，主机端程序会一直处于等待状态，直至所有的 kernel 都执行完毕。

如果一个聚合数据类型的变量(或数组)在 kernels 构件中被引用，并且没有出现在该 kernels 构件的数据子语中，也没有包含在任何 data 构件中，那么它等同于出现在 parallel 构件的 present_or_copy 子语中。如果一个标量变量在 kernels 构件中被引用，并且没有出现在该 kernels 构件的数据子语中，也没有包含在任何 data 构件中，那么它等同于出现在 kernels 构件的 private 子语(如果不带值进也不带值出)中或 copy 子语中。

3. kernels 构件和 parallel 构件的异同

Kernels 构件源于 PGI Accelerator 模型的 region 构件。嵌套 kernels 构件里的循环可能会被编译器转换成能在 GPU 上高效并行的部分。在这个过程中有三步：

- 判断并行中遇到的循环。
- 把抽象的并行转换成硬件上的并行。对于 NVIDIA CUDA GPU， 它会把并行的循环映射到 grid 层次(blockIdx) 或 thread 层次(threadIdx)。OpenACC 申明， gang 对应 grid， vector 对应 thread。编译器可能会通过 strip-mining(一种拆分循环利用缓存的技术)把一层的循环映射到多层。
- 编译器生成并优化代码。

在 kernels 构件中，编译器用自动并行技术识别并行的循环。这个识别能被指令(directives)和条款(clauses)增强，比如说 loop independent。使用-Minfo 标志可以让 PGI 编译器显示编译信息。你能看到类似 Loop is parallelizable 如果编译器认为这个循环可以被并行化。

第二步中，PGI 编译器使用目标硬件的模型去选择循环被映射成 vector 并行还是 gang。比如说，循环中有多个步进为 1 的数组时，更多的会被映射成 vector(thread)并行。在 NVIDIA GPU 上，这种映射更倾向于生成能同时运行的代码。

第三步是底层代码生成和优化。

Parallel 构件源于 OpenMp 的 parallel 构件。OpenMP 会立即产生很多多余的线程，当运行到一个循环是，一个线程运行一部分。

4.3.2 相似度计算

将相似度计算分成三个部分：

4.3.2.1 预处理

程序伪代码如下:

```
Pre_Fact();
#pragma acc kernels copyin(sum[0:n_sample]), copy(temp1[0:total*n_array1], temp2[0:total*n_array2]) , copyin(num[0:n_sample*total])
{
#pragma acc loop independent
for(int i = 0; i < n_sample; ++i) {
    #pragma acc loop independent
    for(int j = i + 1; j < n_sample; ++j) {
        double p_value_0_0 = P_value_cacul(sum[i], sum[j], 0, 0);
        for (int index = 0; index < total; ++index) {
            double p_value = P_value_cacul(sum[i], sum[j], x, y);
            if(p_value <= pvalue) {
                if(up_down == '+') {
                    t1[k] |= 1,t2[l] |= 1;
                } else {
                    t1[k] |= 2,t2[l] |= 1;
                }
            } else {
                t1[k] |= 0, t2[l] |= 0;
            }
        }
    }
}
}
```

调用 P_value_cacul(x,y)函数计算 x 和 y 的差异显著性，为简化计算,我们将原来的：

$$P = \left(\frac{\mathrm{N2}}{\mathrm{N1}}\right)^{\mathrm{y}} * \left(\frac{(\mathrm{x}+\mathrm{y})!}{\mathrm{x}!} * \mathrm{y}!\right)\left(1+\frac{\mathrm{N2}}{\mathrm{N1}}\right)^{\mathrm{x+y+1}} \quad (4-2)$$

求对数后得到：

$$\log P = y * \log\left(\frac{N2}{N1}\right) + \log\big((x+y)!\big) - \log(x!) - \log(y!)$$
$$-(x+y+1) * \log\left[1+\left(\frac{N2}{N1}\right)\right] \quad (4-3)$$

注意到其中需要计算 log(x!)，而这个值计算过程较为繁琐。因此我们通过 Pre_Fact()函数预处理出所需要的级数的对数。为了在不影响结果的条件下尽量减少计算时间，这里使用了斯特灵级数的一个近似值：

$$\ln n! = n\ln n - n + \frac{1}{2}\ln 2\pi n + \frac{1}{12n} - \frac{1}{360n^2} \quad (4-4)$$

然后根据 p_value 值的大小将 x, y 的大小关系和显著性关系用位运算填充到 t1 和 t2 中。

整个代码片段通过 OpenACC 导语包围，通过 PGI 编译器自动的达到异构计算的目的。

4.3.2.2 通过大小关系得出和谐对数

伪代码如下：

```
#pragma acc kernels pcopy(vec_denom[0:total], temp2[0:total*n_array2])
for (int i = 0; i < total; ++i) {
    unsigned long long pattern;
    int denom = 0;
        #pragma acc loop private(pattern, denom) reduction(+:denom)
        for(int j = 0; j < n_array2; j++){
            pattern = temp2[i*total + j];
            denom += bit_count2(pattern);
        }
        vec_denom[i] = denom;
    }
```

为了得到一种基因中的显著不同的和谐对数总数，程序归约了相应的值。这里为了能够快速统计比特位数，运用了 pop_cnt 计算方法中的 hacker_popcnt 算法[22]。算法的简单描述:

```
int hacker_popcnt(unsigned int n)
{
        n -= (n>>1) & 0x55555555;
        n    = (n & 0x33333333) + ((n>>2) & 0x33333333);
        n    = ((n>>4) + n) & 0x0F0F0F0F;
        n += n>>8;
        n += n>>16;
        return n&0x0000003F;
}
```

整个代码通过 kernels 导语做到并行，并通过 reduction 导语提示编译器做到更好的优化。

4.3.2.3 计算相关性系数

伪代码如下：

```
#pragma acc kernels loop independent pcopyin(temp1[0:total*n_array1],\
                temp2[0:total*n_array2], vec_denom[0:total]), \
                pcopy(p_tau[0:total*total], p_name_ij[0:total*total*2])
for (int i = 0; i < total; ++i) {
#pragma acc loop independent private(denomx)
    for (int j = i+1; j < total; j++) {
        for (int k = 0; k < n_array1; ++k) {
            pattern1 = temp1[i*n_array1 + k];
            pattern2 = temp1[j*n_array1 + k];
            con_dis += bit_count2(pattern1 & pattern2);
        }
}
}
```

程序通过与位运算将两个基因的大小关系联系起来，并通过 bit_count2 计算其中 1 的比特位数，即两个基因的和谐对数。有了和谐对数就能很方便的计算出相关性系数。

4.3.3 cuBlas 编程方法

4.3.3.1 cuBlas context

所有应用程序必须通过调用 cublasCreate()函数对 Cublas Library 内容进行初始化处理，然后明确地传递给后面库函数调用，一旦应用程序结束了对库的使用，必须调用函数 cublasDestory()以释放与 Cublas 库内容相关的资源。

这种方式允许用户在使用多主机线程和多 GPU 时可明确地掌握库的启动.例如，应用程序可使用 cudaSetDevice()连接不同设备和不同的主机线程，其中的每一个主机线程初始化一个独特的处理，这可使一个独特的设备连接那个主机线程，然后 cublas 库函数调用不同处理去自动调度不同的计算到设备中。只有在假设相应的 cublasCreate() 和 cublasDestory()之间内容保持不变的情况下才能使设备与某个特定的 CUBLAS 内容相连。为了使 CUBLAS 库在同一主机线程中使用不同设备，应用程序必须通过 cudaSetDevice()设置所需要使用的新设备然后创建另一个 CUBLAS 内容，通过调用 cublasCreate()连接一个新设备。

4.3.3.2 Scalar Parameters（标量参数）

在 CUBLAS 中，标量参数 α 和 β 可以通过主机或设备中的参数传递。 一些函数例如 amax()，amin，asum()，rotg()，rotmg()，dot() 和 nrm2()可以返回标量结果，通过主机或设备的参数返回标量数值。尽管这些函数可以像矩阵和向量一样立即返回值，标量结果只有当常规的 GPU 执行完成后才能准备好，这些都需要适当的同步以从主机上读取结果。这些变化允许库函数使用流执行完全异步，即使标量 α 和 β 由前一个内核产生。

4.3.3.3 Parallelism with Streams（流的并行性）

如果应用程序使用多个独立任务来得到计算结果，CUDA 流可用于在重叠计算中执行这些任务。应用程序可以在概念上将每个流与任务相联系，为了实现任务的重叠计算，用户应当使用 cudaStreamCreate()函数创建 CUDA 流，在调用实际的 cublas 例程以前，调用 cublasSetStream()让这些流被 cublas 库中每个单独的例程所引用，然后再每个单独流中的计算结果将会在 GPU 上自动重叠计算。当一个任务的计算规模相对较小时，不足以填满 GPU 工作。

4.3.3.4 Batching Kernels(批处理内核)

例如我们有个应用程序需要使用密集矩阵实现多个独立小矩阵的相乘运算。

很明显即使有数以百万的独立小矩阵也不能实现像一个大矩阵实现 GFLOPS（每秒千兆次浮点运算）。例如对于一个 n×n 规模的大矩阵乘法对于的输入需要执行 n 三次方的操作。然而对于 1024 个$\left(\frac{n}{32}\right)*\left(\frac{n}{32}\right)$小矩阵对于相同的输入需要执行$\frac{n3}{32}$次操作。由此可见我们可以通过使用很多独立的小矩阵实现更好的计算效率相比于一整个大矩阵。 GPU 体系结构允许我们同时执行多个内核。因此，为了批处理这些独立内核，我们可以将其放在每个单独的流中执行。尤其，在上述事例中我们通过 cudaStreamCreate()创建了 1024 个 CUDA 流。

4.3.4 利用 MPI 实现分布式计算

伪代码如下：

```
int NUM_SEND = (numproc-1)/2;
if (NUM_SEND > 0) {
    int *mpi_total_arr = (int*)malloc(sizeof(int)*numproc);
    MPI_Allgather(&total, *);
    MPI_Isend(temp1, *);
    MPI_Irecv(mpi_temp1, *);
    for (int i = 0; i < NUM_SEND; ++i) {
        MPI_Waitall(2, request, status);
        if (i < NUM_SEND -1) {
            MPI_Isend(temp1, n_array1*total, *);
        }
    MPI_Isend(vec_denom, total, *);
    MPI_Recv(mpi_name_len, *);
    compute_kendall_tau_mpi(total,*);
}
```

通过调用 MPI 的函数发送接受多台服务器的数据，并通过 compute_kendall 计算当前服务器所需要做的任务。

4.3.5 高斯混合模型计算

高斯混合模型计算是迭代做最大似然估计的过程。令

$$\mathrm{p(x;\mu)} = \frac{1}{2\pi^{\frac{l}{2}}\varepsilon^{\frac{1}{2}}}\exp\left(-\frac{1}{2}(\mathrm{x}-\mu)\varepsilon(\mathrm{x}-\mu)\right) \qquad (4-5)$$

表示数据 x 在μ 为均值的高斯分布下的概率。为简化计算过程，取它的对数似然函数：

$$\mathrm{Log}(\mu) = \ln\prod_{k=1}^{N} p(x;\mu)$$

$$= -\frac{N}{2} * \ln((2\pi)^{l} * \varepsilon) - \frac{1}{2} * \sum_{k=1}^{N}(x-\mu)^{T} * \varepsilon^{-1} * (x-\mu) \qquad (4-6)$$

可以看到，等式右边的第一项是常数项，整体的计算复杂度主要在第二项上 (O(n*m*m)) (4.3)，所以文中只讨论对这一步的加速方法。

我们使用 cuBlas 高效的矩阵计算库来完成性能提升，伪代码如下：

```
cublasHandle_t handle;
cublasCreate(&handle);
for (int i = 0; i < cluster_num; ++i) {
    caculatepm<<<256, 256>>>( d_pm, d_profile, d_mean+i*n_sample, gene_num, n_sample);
    err = cublasDgemm( handle, CUBLAS_OP_N, CUBLAS_OP_N, *);
    matrixDot<<<256, 256>>>( d_tmp, d_pm, d_result, gene_num, n_sample, cluster_num, i);
}
cuda_synchronize();
cuda_memcpy_DTH(result, d_result, gene_num * cluster_num * sizeof(double));
```

程序使用 cublasDgemm 函数批量完成矩阵操作。在 cuBlas 中使用列为主的存储方式，为了能够高效的将 C 中的数组直接完成操作，我们利用矩阵的转置性质：

$$\mathrm{C} = \mathrm{A} * \mathrm{B} \rightarrow \mathrm{C}' = \mathrm{B}' * \mathrm{A}' \qquad (4-7)$$

4.4 测试环境

表 4-1 记录了程序运行的硬件环境，使用 putty 软件经过 ssh 通道连接天河计算集群。本文使用了 13 个带有 Tesla M2050 显卡的计算节点。

表 4-1 测试硬件环境

集群	天河 1A
CPU	Intel Xeon X5670 @ 2.93GHz (6 cores) x 2
GPU	NVIDIA Tesla M2050 3G memory
内存	24GB
操作系统	Red Hat Enterprise Linux Server release 5 (Tikanga)
CUDA	Version 4.0

表 4-2 记录了所使用的输入数据。本文使用的测试数据是华大基因提供的人体肠道基因组测序数据。

表 4-2 测试数据

样本数	基因数	输入文件大小
1267	100	270KB
1267	500	1.3MB
1267	1000	2.5MB
1267	5000	13MB

4.5 优化效果

4.5.1 相关性系数优化效果

将测试过程分为预处理、计算和谐对数、计算相关系数部分。分别测试了有表4-2 说明的测试数据。

4.5.1.1 预处理

如表 4-3 所示，通过对算法上并行量的估计和优化、对算法流程的改进。在5000 的数据量下，计算时间从 CPU 的 39284s 降低到 482s。计算时间加速比达到81 倍。

表 4-3 预处理部分加速比数据

基因数	100	500	1000	5000
CPU(s)	1057	4170	11051	39284
GPU(s)	27	71	223	482
加速比	39.1	58.7	49.5	81.5

4.5.1.2 计算和谐对数部分

如表 4-4 所示，因为这一计算过程的计算量并不大、计算时间并不多，导致计算加速比不大。

表 4-4 计算和谐对数部分加速比数据

基因数	100	500	1000	5000
CPU(s)	0.006	0.029	0.058	0.291
GPU(s)	0.013	0.014	0.015	0.048
加速比	0.46	2.07	3.86	6.06

4.5.1.3 计算相关性系数部分

如表 4-5 所示，最终通过和谐对数预处理数据计算相关性系数时，在 5000 的

数据量下的计算时间从 1625s 下降到 40s，计算加速比达到 40 倍。

表 4-5 计算相关性系数部分加速比数据

基因数	100	500	1000	5000
CPU(s)	0.643	16.209	64.876	1625.211
GPU(s)	0.035	0.474	1.738	40.615
加速比	18.3	34.1	37.3	40.0

4.5.1.4 相关系数过程结果

如表 4-6 所示，计算相关系数的总体计算时间从 40909s(11 小时)下降到 523s (8 分钟)，总的计算加速比达到 78 倍。

表 4-6 计算相关系数部分整体加速比数据

基因数	100	500	1000	5000
CPU(s)	1057	4186	11116	40909
GPU(s)	27	71	225	523
加速比	39.1	58.9	49.4	78.2

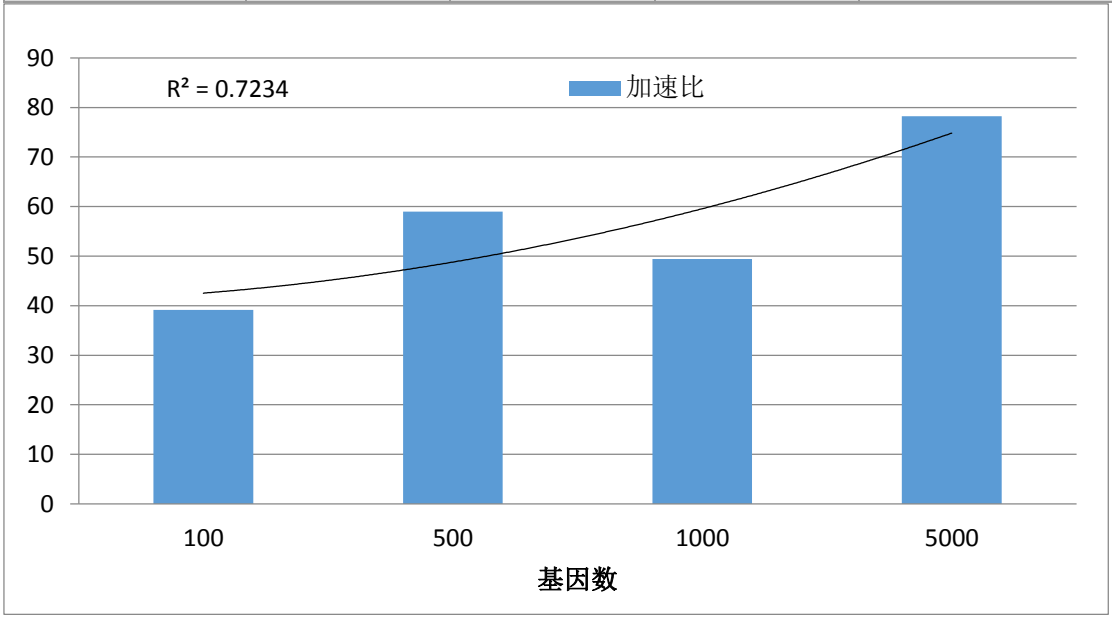


图 4-1 计算相关系数部分整体加速比数据

如图 4-1 所示，在基因数从 100 到 5000 的过程中，加速比随着数据量的增大而增大，这说明程序具有不错的扩展性。

4.5.2 MPI 分布式计算测试结果

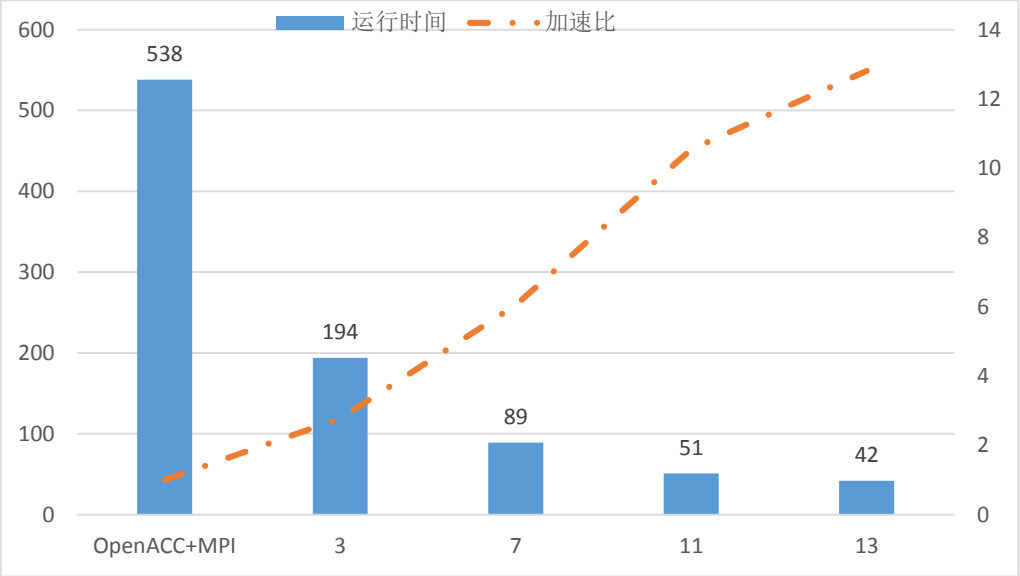


图 4-2 多 GPU 与单 GPU 的计算时间比较，最左侧是单 GPU 下的计算时间。

如图 4-2 所示，在 5000 基因，1267 个样本情况下使用 MPI 做分布式处理有的测试结果为：

在使用 1、3、7、11、13 个计算节点时，计算时间分别从 40909s 下降到 538s、194s、89s、51s、42s。使用 11 个节点计算时，相比 CPU 计算有了 770 倍的加速比，只需 51 秒就能完成计算。随着计算节点数增长，加速比基本呈线性，表明有较为优秀的扩展性。

4.5.3 高斯混合模型计算优化结果

高斯混合模型计算只测试了在 20000 个基因，1267 个样本情况下进行一次迭代的结果。其中 CPU 计算用时 74304s，GPU 用时 30.8s，加速比为 2412.5 倍。按迭代 20 次分析，计算时间从原来的 17 天减少到 10 分钟。矩阵计算使用了 CuBlas 库，高效解决了线性代数问题。

4.6 数据可视化

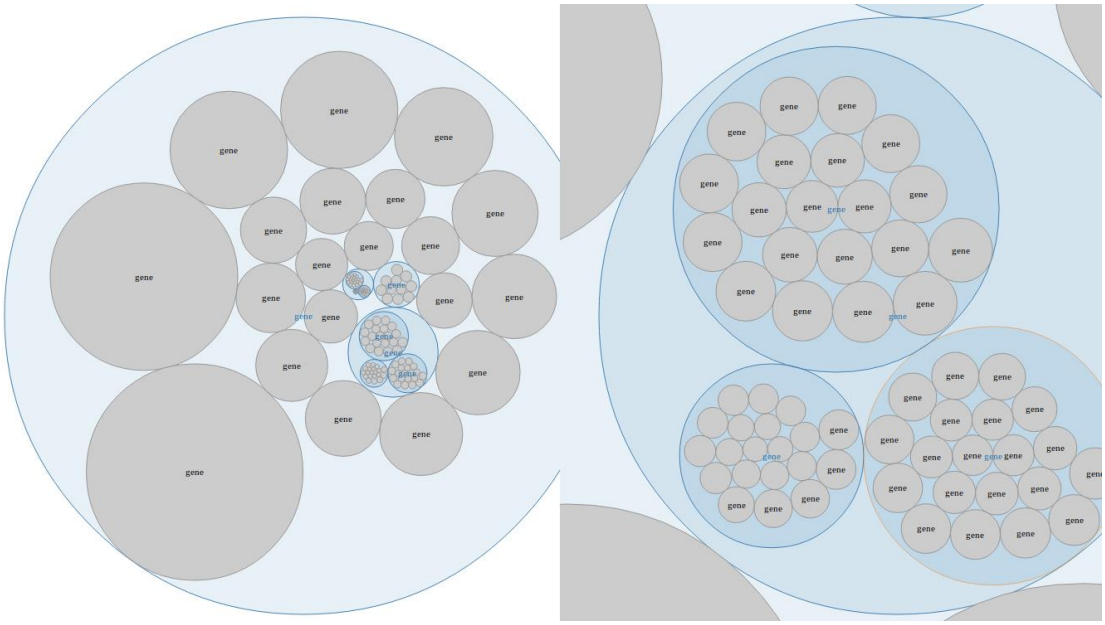


图 4-3 将宏基因组数据可视化后的结果，左图是整体，有图是点击后放大效果

如果 4-3 所示，通过考虑层次聚类的特性，我们将树的结构重新解构成包围圆的结构，将同一个类中的基因用圆包围起来。左图是解析完数据后的整体图像，可以看到最终生成的多个类。右图是点击其中一个需要关注的类后经过自动缩放生成的图像。

第五章 总结与展望

在这次课题工作中，通过利用 NVIDIA 的 GPU 实现异构计算技术，并利用多台服务器通过 MPI 实现分布式计算技术。选择了合适的技术解决方案，简单完成了数据的可视化实现。

将层次聚类过程中的计算相似度计算步骤的计算复杂度从$O(N^2 * M^2 * C)$下降到$O(N * M^2 * C + N^2 * M^2)$，将计算时间从 11 个小时下降到 51 秒，实现了 770 倍左右的加速比，获得了较好的性能优化效果。

在高斯混合模型计算中，在找到性能瓶颈的情况下，通过利用 cuBlas 高性能基本线性代数库完成性能优化。将计算时间从 17 天下降到 10 分钟，获得了 2412 倍左右的加速比。

通过利用 HTML5 技术，将分层聚类的计算结果通过可视化技术展示了出来。跨平台的实现减轻了分析工具的负担，加快了实验流程。

通过实现高性能的聚类过程和可视化过程，大大减少了等待时间，加速了宏基因组聚类分析过程。并对数量级数做了一定预估，通过一定的算法是程序能够处理更大的数据量。

5.1 本文的研究工作

本文通过 GPU 和 MPI 两种高性能计算技术结合多种并行计算算法及优化手段，将宏基因组聚类的过程所需要的计算时间降低到一个可以接受的范围。同时使用简洁的可视化解决方案完成了对聚类后数据可视化的需求。较为完整的完成了课题要求。

5.2 未来的研究方向

在基因组聚类这个领域，准确快速的解决问题是必要的。而当数据量不停增长，数据额外的噪声会越来越成为聚类过程中不可避免要解决的难题。同时随着下一代测序技术的到来，这个问题会越来越急切。同时，本文完成的可视化方案还很不成熟。因此，本文接下来的研究方向将是加强对数据的理解，提取数据模型，减少

噪音对聚类准确度的影响，并改进可视化方案以能更好的表达数据的特性。

5.3 收获与体会

在这次课题中，我不断的学习包括 GPU 架构，并行算法，异构计算技术，分布式计算技术以及各种性能分析方法，同时不断更新知识，获取最新的技术手段。

为了完成这次可以，我阅读了大量的文献资料和读物，对高性能计算领域有了初步的了解和分析，并完成了初步的成果。

同时在论文写作过程中又复习了之前学习的理论，对知识有了更深刻的认识。李永乐图书馆、互联网等渠道搜索大量资料，并学会了如何整理和分类，提高了文献的检索能力和分析整理能力。在写好毕业论文的过程中，我不仅学习了专业知识，而且学会了论文语言的组织和结构安排。

对我而言，这次毕业设计最大的收获就是意志的磨练，对我实际能力的一次提升，也会对我未来的学习和工作有很大的帮助。

参考文献

[1] Shannon W, Culverhouse R, Duncan J. Analyzing microarray data using cluster analysis[J]. Pharmacogenomics, 2003, 4(1): 41-52.

[2] Stein L D. The case for cloud computing in genome informatics [J] Genome Biology, 2010, 11: 207

[3] Li X. Parallel algorithms for hierarchical clustering and cluster validity[J]. Pattern Analysis and Machine Intelligence, IEEE Transactions on, 1990, 12(11): 1088-1092.

[4] Manavski S A, Valle G. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment[J]. BMC bioinformatics, 2008, 9(Suppl 2): S10.

[5] Jung S. Parallelized pairwise sequence alignment using CUDA on multiple GPUs[J]. BMC Bioinformatics, 2009, 10(Suppl 7): A3.

[6] Liu W, Schmidt B, Voss G, et al. Bio-sequence database scanning on a GPU[C] Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International. IEEE, 2006: 8 pp.

[7] Hussong R, Gregorius B, Tholey A, et al. Highly accelerated feature detection in proteomics data sets using modern graphics processing units[J]. Bioinformatics, 2009, 25(15): 1937-1943.

[8] Chang D J, Desoky A H, Ouyang M, et al. Compute pairwise manhattan distance and pearson correlation coefficient of data points with gpu[C]. 10th ACIS International Conference on. IEEE, 2009: 501-506.

[9] Bustamam A, Burrage K, Hamilton N A. Fast parallel Markov clustering in bioinformatics using massively parallel computing on GPU with CUDA and ELLPACK-R sparse format[J]. IEEE ACM Transactions on Computational Biology and Bioinformatics (TCBB), 2012, 9(3): 679-692.

[10] Chang D J, Kantardzic M M, Ouyang M. Hierarchical Clustering with CUDA/GPU[C].ISCA PDCCS. 2009: 7-12.

[11] World Headquarters SAS Institute Inc., Data Visualization: Making Big Data Approachable and Valuable[OL]. http://www.sas.com/content/dam/SAS/en_us/doc/other1/, 2013

[12] Kirk D B, Wen-mei W H. Programming massively parallel processors: a hands-on approach[M]. Newnes, 2012.

[13] Chen T P, Chen Y K. Challenges and opportunities of obtaining performance from multi-core CPUs and many-core GPUs[C].. IEEE International Conference on. IEEE, 2009: 613-616.

[14] K. Fatahalian K, Houston M. GPUs a closer look[C]. ACM, 2008: 11.

[15] NVIDIA Corporation. Whitepaper NVIDIA’s Next Generation CUDA Compute Architecture: Fermi[OL]. http://www.nvidia.com/content/PDF/Fermi. 2010.

[16] Patterson D. The top 10 innovations in the new NVIDIA Fermi architecture, and the top 3 next challenges[J]. NVIDIA Whitepaper, 2009.

[17] NVIDIA Corporation. Whitepaper NVIDIA’s Next Generation CUDA Compute Architecture: Kepler GK110[OL]. http://www.nvidia.com/content/PDF/kepler. 2012

[18] NVIDIA Corporation. Whitepaper NVIDIA GeForce GTX 750 Ti Featuring First-Generation Maxwell GPU Technology[OL]. http://www.nvidia.com/content/PDF/Maxwell. 2014

[19] NVIDIA Corporation, Cublas API user guid [OL], http://docs.nvidia.com/cuda, 2013

[20] 边肇祺, 张学工. 模式识别[M]. 清华大学出版社有限公司, 2000.

[21] Warren H S. Hacker's delight[M]. Addison-Wesley Professional, 2003.

[22] Wang H, Wang W, Yang J, et al. Clustering by pattern similarity in large datasets[C]. ACM, 2002: 394-405.

[23] D’haeseleer P, Wen X, Fuhrman S, et al. Mining the gene expression matrix: Inferring gene relationships from large scale gene expression data[C]. Springer US, 1998: 203-212.

[24] Jiang D, Tang C, Zhang A. Cluster analysis for gene expression data: A survey[J]. Knowledge and Data Engineering, IEEE Transactions on, 2004, 16(11): 1370-1386.

[25] Quackenbush J. Microarray data normalization and transformation[J]. Nature genetics, 2002, 32: 496-501.

致 谢

本科生四年的工作和学习即将画上句号，感叹时光飞逝。四年时光虽然短暂，却收获良多。

首先我要由衷的感谢深圳华大基因研究院的王丙强王老师以及郭贵鑫，叶志强，邱爽，林哲，李璇等同事，没有他们的鞭策、引导和帮助，我无法完成这个项目。

还要感谢电子科技大学生命科学与技术学院的王玲王老师及教研室的师兄师姐，没有他们的关心和指导，我无法完成这篇论文。

感谢电子科技大学生命科学与技术学院给予我们良好的科研和生活环境。

最后还要感谢同学和家人的理解和帮助。

外文资料原文

Performance of Kepler GTX Titan GPUs and Xeon Phi System

Hwancheol Jeong*, Weonjong Lee, and Jeonghwan Pak

1. Introduction

On 2013, NVIDIA launched a new Kepler GPU, GTX Titan, named after the fastest supercomputer, a GPU cluster of NVIDIA Tesla K20X at Oak Ridge National Laboratory [1]. GeForce GPUs are designed for gaming. However, GTX Titan is good for parallel computing with CUDA, too. From the standpoint of computing, GTX Titan is as great as Tesla K20X. Nevertheless, the price of the former is about 3 times cheaper than of the latter.

Meanwhile, in 2012, Intel announced the Xeon Phi system with Intel many integrated core architecture (MIC) [2]. A Xeon Phi coprocessor integrates many CPU cores on a PCI express card like GPU, so that it could, in principle, provide similar theoretical performance with GTX Titan. The merit is that most of usual C codes which runs on CPUs can run on Xeon Phi system without much modification, because it is a CPU-based platform. However, it turns out that its performance is so low that it is very hard to obtain the high performance from Xeon Phi.

2. GTX Titan & Kepler Architecture

Table 1 presents chip and memory specification of NVIDIA Kepler GPUs compared with Fermi GTX 580. GTX Titan inherits most of important features of the Kepler architecture such as new streaming multiprocessor SMX, increased memory bandwidth and dynamic parallelism. In addition, it supports the features provided only for Tesla or Quadro GPUs such as large memory size and high performance in double precision floating point calculation.

	Fermi	Kepler
simultaneous blocks / SM(X)	8	16
warp schedulers / SM(X)	2	4
registers / thread	63	255
bandwidth (GB/s)	192 (GTX 580)	288 (GTX Titan)

Table 2: changed properties related with thread and block scheduling

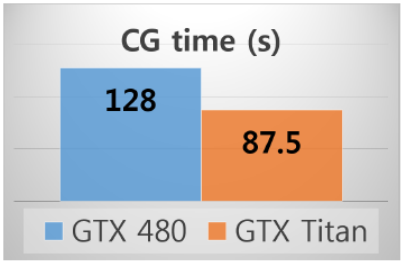
Fig. 1 shows the performance of conjugate gradient (CG) solver by Fermi GTX 480 and

Kepler GTX Titan without applying any optimization to Kepler GPUs. Although GTX Titan's performance are, in principle, much better than that of GTX 480, our CG code is optimized only for Fermi GPUs and not for Kepler GPUs. Hence, it is necessary to tune the code such that it achieves the highest performance for GTX Titan.

There are several optimization schemes possible for GTX Titan. Table 2 shows those changes in GPUs regarding thread and block scheduling. A SMX (Kepler) has 6 times more cores than SM (Fermi). To deal with these cores, the SMX has twice number of blocks run simultaneously and twice of warp schedulers than SM. The number of registers per thread is also increased to 255, so that a thread can store more variables to registers and reuse them quickly. Therefore, we might obtain significantly better performance by simply adjusting thread and block numbers.

The change of the memory bandwidth is also very important. Unfortunately in general, the main bottle neck in GPUs are the limitation in data transfer speed between GPU registers and

memories. The performance of a CUDA program is usually determined by the product of CGMA (compute to global memory access) ratio and the amount of data transfer per time [3].



Device	CG time (ratio)
GTX 480	128 s (1)
GTX Titan	87.5 s (0.68)

Figure 1: CG performance by GTX 480 and GTX Titan. Here, CG time means the time (in the unit of second) which it takes to run the CG code 10 times in two GPUs.

Here, CGMA ratio is the number of floating point operations per single data transfer.

Kepler architecture has new features to improve the memory usage as follows.

• 8 bytes shared memory bank mode is added. Fermi GPUs provide only 4 bytes (32 bits) mode. But Kepler GPUs provide 8 bytes (64 bits) mode, too. When this mode is turned on,

one gets about twice the effective bandwidth for double precision floating point numbers.

Architecture	Fermi	Kepler (GK104)	Kepler (GK110)		
GPU Device	GTX 580	GTX 680	Tesla K20X	GTX TITAN	GTX 780
# of CUDA Cores	512	1536	2688	2688	2304
Core Clock (MHz)	772	1006	732	837	863
SP GFLOPS	1581	3090	3950	4500	3977
DP GFLOPS	197	128	1312	1300	166
Memory Size (GB)	1.5	2	6	6.1	3
Memory Bandwidth (GB/sec)	192.4	192.26	250	288.4	288.4
L1 cache + shared memory (KB)	64	64	64	64	64
read-only data cache (KB)	0	0	48	48	48
L2 cache (KB)	768	512	1536	1536	1536

Table 1: chip and memory specifications of recent NVIDIA GPUs

• It is possible to adjust the ratio of shared memory and L1 cache to get the better performance with the total memory size fixed. Fermi GPUs only support 16 Kbytes (shared memory) + 48 Kbytes (L1 cache) and 48 + 16 modes. Kepler GPUs can allocate 32 K to shared mem and 32 K to L1.

• 48 KB Read-only data cache is added. The texture memory can be used as an additional read-only cache memory for Kepler GPUs.

• Warp shuffle is introduced. By warp shuffle, data between threads in a warp can be exchanged without using shared memory. Thus we can reduce redundant use of the shared memory. Moreover, its latency is lower than shared memory access

外文资料译文

Kepler GTX 显卡与 Xeon Phi 协处理器性能

Hwancheol Jeong_, Weonjong Lee, and Jeonghwan Pak

1. 介绍

在 2013 年，NVIDIA 推出了新的 Kepler GPU，GTX Titan。它是以 NVIDIA 在橡树岭国家实验室的 GPU 集群[1]中最快的超级计算机在而得名。GeForce GPU 是专为游戏设计的，然而，GTX Titan 也同时善于利用 CUDA 进行并行计算。从计算的角度来看，GTX Titan 和 Tesla K20X 一样强。然而，前者的价格比后者便宜三分之一左右。

同时，在 2012 年，英特尔发布众核架构（MIC）[2]。至强协处理器集成了 PCI Express 以及和 GPU 一样多 CPU 内核，因此，它在原则上可以提供类似于 GTX Titan 的理论性能。这样设计的优点是大部分运行在 CPU 的通常的 C 代码可以在至强系统运行而不用太多的修改，因为它是一个基于 CPU 的平台。然而，事实证明，它的性能是如此之低，很难达到至强的理论性能峰值。

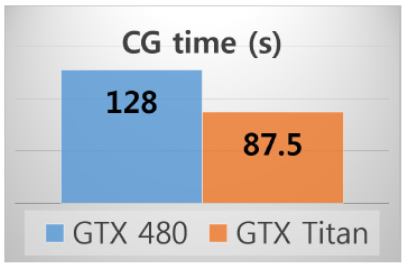
2. GTX Titan 和 Kepler 架构

表 1 近期 NVIDIA 显卡的芯片和存储器参数

Architecture	Fermi	Kepler (GK104)	Kepler (GK110)		
GPU Device	GTX 580	GTX 680	Tesla K20X	GTX TITAN	GTX 780
# of CUDA Cores	512	1536	2688	2688	2304
Core Clock (MHz)	772	1006	732	837	863
SP GFLOPS	1581	3090	3950	4500	3977
DP GFLOPS	197	128	1312	1300	166
Memory Size (GB)	1.5	2	6	6.1	3
Memory Bandwidth (GB/sec)	192.4	192.26	250	288.4	288.4
L1 cache + shared memory (KB)	64	64	64	64	64
read-only data cache (KB) [#]	0	0	48	48	48
L2 cache (KB)	768	512	1536	1536	1536

表 1 给出了 NVIDIA 系列 GPU 芯片和显存规格的比较，包括费米的 GTX580，开普勒的 GTX 680 等。GTX Titan 继承了大部分的开普勒架构的重要功

能，如新的流式多处理器 SMX，增加了内存带宽和动态并行性功能。此外，它提供了只提供给 Tesla 和 Quadro GPU 的大容量存储器尺寸和高性能双精度浮点计算的特点。



Device	CG time (ratio)
GTX 480	128 s (1)
GTX Titan	87.5 s (0.68)

图 1 GTX 480 和 GTX Titan 的 CG 性能。图中，CG 时间表示在这两个 GPU 中运行十次的总时间。

图 1 显示由 Fermi GTX480 和不应用任何优化的 GTX Kepler Titan 计算共轭梯度（CG）求解器的性能。虽然 GTX Titan 的表现在比 GTX480 的要好得多，但是我们只对 Fermi GPU 做过了程序特殊优化。因此，有必要调整代码，使得其实现 GTX Titan 最高性能。

表 2 关系到线程和线程块调度的参数变化

	Fermi	Kepler
simultaneous blocks / SM(X)	8	16
warp schedulers / SM(X)	2	4
registers / thread	63	255
bandwidth (GB/s)	192 (GTX 580)	288 (GTX Titan)

对 GTX Titan 有几种可能的优化方案。表 2 列出了那些关于线程块调度变化的信息。一个 SMX（开普勒）比 SM（Fermi）有 6 倍以上的内核。为了处理利用核心， SMX 有两倍数量的线程块和两倍的调度器数量。每个线程的寄存器的数量也增加至 255，这样一个线程可以有更多的变量存储到寄存器并快速重用它们。因此，我们可以通过简单地调整线程块数取得显著更好的性能。

内存带宽的变化也是非常重要的。不幸的是在一般情况下， GPU 的主要瓶颈是在 GPU 寄存器和存储器之间的数据传输速度上。CUDA 程序的性能通常通过计算全局存储器访问的比例(CGMA)和单位时间的数据传输量的积求出[3]。这里，CGMA 比率是每单次数据传输的浮点操作的数量。

Kepler 架构有提高内存的使用能力的新功能，如下。

•8 字节的共享内存的 bank 模式被添加。Fermi 的 GPU 提供 4 字节（32 位）

模式。但开普勒图形处理器提供 8 个字节（64 比特）模式。当这个模式被开启时，可以得到两倍左右的双精度浮点数有效带宽。

•虽然具有固定的总存储器大小可以，但是可以通过调整共享存储器和 L1 高速缓存的比率以获得更好的性能。费米的 GPU 只支持 16 字节（共享内存）+ 48 字节（L1 高速缓存）或 48 +16 的模式。开普勒 GPU 的可分配 32 K 到共享存储器和 32 K 到 L1。

•48 KB 只读数据缓存被添加。纹理存储器可以在开普勒 GPU 用作额外的只读高速缓存存储器的。

•新的洗牌技术被提出。由经洗牌，在一个 warp 内的数据可以不使用共享内存进行交换。因此，我们可以减少使用的共享内存。此外，它的通信延迟比共享内存访问更低。

引用：

[1] “NVIDIA GTX Titan.” http://nvidianews.nvidia.com/Releases/NVIDIAIntroduces-GeForce-GTX-TITAN-DNA-of-the-World-s-Fastest-Supercomputer-Powered-by-World-s-Fa-925.aspx.

[2] “Wikipedia - Intel MIC.” http://en.wikipedia.org/wiki/Intel_MIC.

[3] D. B. Kirk andW. mi W. Hwu, Programming Massively Parallel Processors : A Hands-on Approach. Elsevier Science, 2010. ISBN:01238147