个分类号 _____ 学号 M200972508

学校代码 10487 密级 _____

华中科技大学

# 硕士学位论文

## 基于 GPU 的稀疏矩阵运算优化研究


学位申请人： 梁 添

学 科 专 业： 计算机应用技术

指 导 教 师： 章 勤 教授


答 辩 日 期： 2012 年 2 月 8 日

A Thesis Submitted in Partial Fulfillment of the R equirements
for the Degree of Master of Engineering

# The Research on Optimizing Sparse Matrix Computation Based on GPU


Candidate ： Tian Liang

Major ： Computer Application

Supervisor： Professor Zhang Qin



Huazhong University of Science and Technology

Wuhan 430074, P.R.China

Feb., 2012

# 独创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除文中已经标明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到，本声明的法律结果由本人承担。

学位论文作者签名：

日期：　　年　月　日

# 学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，即：学校有权保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权华中科技大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

本论文属于 保密□ ，在_____年解密后适用本授权书。

不保密□。

（请在以上方框内打“√”）

学位论文作者签名：　　指导教师签名：

日期：　年　月　日　　日期：　年　月　日

# 华 中 科 技 大 学 硕 士 学 位 论 文

## 摘 要

大规模稀疏矩阵的求解是高性能计算中的一个常见问题，广泛存在于工程实践尤其是计算机仿真领域。用常规方法解稀疏矩阵时，会浪费大量的计算资源。目前，在国内外，在通用计算领域对稀疏矩阵的运算研究较少。已有的研究主要是实现稀疏矩阵和向量之间的乘法运算。

研究 GPU 上的稀疏矩阵向量乘法运算的实现并优化。针对于稀疏矩阵非零元素分布不均造成的空转问题以及同一线程组中线程不能合并访存的问题，提出了一种分段行合并存储策略的稀疏矩阵向量乘方法。针对于一个线程组内的线程间计算量负载不均衡而造成的线程间等待问题以及因线程不满足对全局存储器的合并访问要求而造成的访存延迟问题，提出了一种按行分块存储策略的稀疏矩阵向量乘方法。并针对以上两种方法实现了全局存储器的访存优化并使用纹理存储器和常数存储器对运算进行加速。实现了 GPU 上的稀疏矩阵线性方程求解的雅可比迭代法和广义最小残量法并优化。提出的优化方法可以推广至所有的 GPU 下求解稀疏矩阵线性方程的迭代法上，具有普遍意义。最后给出了主机设备通信优化和共享存储器的访存优化方案。

测试表明，稀疏矩阵方程求解运算相比于获得了 10.3 至 74.0 范围的加速比。

**关键词:** CUDA 架构，GPU 通用计算，稀疏矩阵线性方程运算，迭代法

# Abstract

Large-scale sparse matrix solver is a common problem in high-performance computing, widely exists in engineering practice, particularly in computer simulation field. Using conventional methods for solving sparse matrix will waste a lot of computing resources. At present, both at home and abroad, research on sparse matrix computation in general purpose GPU computing is less. Existing research is major on sparse matrix and vector multiplication.

Sparse matrix vector multiplication on the GPU is achieved and optimized. In order to solve the problem caused by uneven distribution for non-zero elements in sparse matrix and the problem that threads in a same warp can't visit GPU memory in a merged way, the SC-CSR sparse matrix vector multiplication method on GPU is proposed. To solve thread waiting issue caused by load imbalance of thread computing in a warp and the memory access issue as a result of the thread does not meet the merger of the global memory access requirements, a sparse matrix vector multiplication approach based on VAB sparse matrix storage format is proposed. The optimization of global memory access and the use of texture memory and constant memory is proposed for the above two methods. Linear equation solver on the GPU sparse matrix with Jacobi iteration and Generalized Minimum Residual method is achieved and optimized. The optimization method proposed can be extended to all the iterative method on GPU for solving sparse matrix linear equation of universal significance. Finally, the sparse matrix equation solving is accelerated by the host device communication and shared memory access optimization.

Experiments show that the sparse linear equation computation speed increases significantly in relation to serial code on CPU with a speed up ranging from 10.3 to 74.0.

**Keywords:** CUDA architecture, GPGPU, sparse linear equation, iterative method

# 华 中 科 技 大 学 硕 士 学 位 论 文

## 目 录

# 1 绪 论

## 1.1 研究背景及意义

在大型科学和工程计算中出现的计算问题，对计算机运算能力的要求越来越高。稀疏矩阵线性代数运算是高性能计算中的常见问题，广泛地存在于各类工程实践尤其是计算机仿真领域以及科学问题中，如电子工程、航空航天工程、经济工程、力学、电磁学，计算流体力学，流体润滑，船舶与海洋工程，环境水力学，微波与电磁场，电力系统互联电网，集成电路电容分析，图像恢复重建，生物医学工程，水力管网计算、网络分析、遗传理论等领域。

近年来，图形处理器 GPU（Graphics Processing Unit）的发展极其迅猛，GPU 已经从传统的图形图像处理拓展到通用计算领域[1]。现代 GPU 的通用计算单元更加集中和独立，浮点计算能力飞速增长。CPU 和 GPU 的结构对比如图 1.1 所示，GPU 的并行处理单元数量要远高于 CPU。

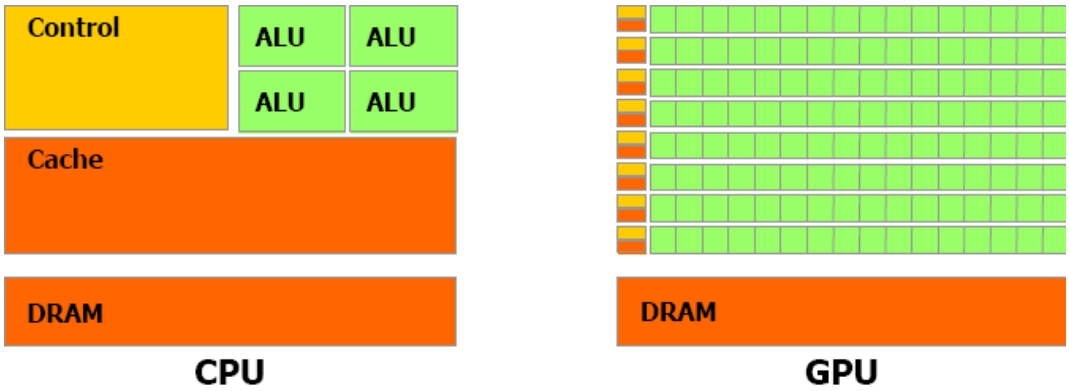


图 1.1 CPU 和 GPU 的结构对比

主流 GPU 的单精度浮点处理能力已经达到了同时期 CPU 的 10 倍左右，同时 GPU 的存储器带宽也是 CPU 带宽的大概 5 倍。NVIDIA 的 GTX280 GPU 达到大约 1TFLOP/s 的浮点运算速度，而同期的 CPU 大约达到 3.2GFLOP/s 的浮点运算速度。NVIDIA 的 G80 Ultra 的显存带宽约为 100GB/s，CPU 的内存带宽却不到 20 GB/s。GPU 的显存带宽明显比 CPU 和内存的带宽高出许多，这样可以通过大量的计算隐藏数据读写的延迟，降低 GPU 读取数据时间，提高并行计算的性能[2]。GPU 的耗电量相对于 CPU 来说要低得多，除了 PCI 接口外不需要任何其它电源为之供电，也正是因为这个原因，GPU 非常节约能耗。GPU 的浮点运算能力远高于 CPU，在处

理同样大小的计算任务时比 CPU 耗电量低，更加绿色环保。GPU 架构的特性使得 GPU 具有超凡的并行计算能力，它具有更低的价格，却提供了更高的并行处理能力，这些特点使得 GPU 在处理计算密集型任务时，有 CPU 所不可比拟的优势[3]。

为了实现通用计算的普及，需要向各行业技术人员提供高性能常用数学运算操作。在 GPU 上实现并优化常用数学运算操作，如 FFT[3] 运算，矩阵乘法运算[3]，LU 分解运算[3]以及 QR 分解运算[3]等，是通用计算研究的热点。目前线性代数的研究工作主要集中在稠密线性代数运算方面。

用常规方法进行稀疏矩阵运算时，存在运算速度慢、计算效率低的问题。在 GPU 平台上实现并优化稀疏矩阵的运算是一个有着重要意义的研究方向。

## 1.2 国内外研究现状

### 1.2.1 基于 GPU 的数值计算

近年来，数值计算已经成为 GPU 通用计算的研究热点，引起了国内外众多研究机构的重视。文献[4]中实现了基于 CUDA 平台的自动调整的 3D-fft 库。Vasily Volkov 等人实现了矩阵乘法等部分基于 GPU 的稠密线性代数运算[3] 。Nico Galoppo 等人实现了基于图像硬件的稠密线性系统的 LU 分解运算[3] 。Andrew Kerr 等人实现了基于 GPU 的 QR 分解运算[3] 。陈一峰等人实现了基于 GPU 的矩阵乘法和快速傅立叶变换的运算并做了优化[3]，还在具有 16 个节点 32 片 GPU 的集群上较好的实现了快速傅立叶变换[3]。这些研究工作主要集中在快速傅立叶变换以及稠密线性代数运算方面。

### 1.2.2 稀疏矩阵运算在高性能计算领域中的研究

稀疏矩阵运算因为在科学和工程计算中的广泛应用已逐渐成为高性能计算领域的研究热点。稀疏矩阵运算的并行化较稠密线性运算难度更大，是高性能计算领域一直以来都具有挑战性研究课题。

部分研究工作主要是对稀疏矩阵的存储结构进行优化。文献[10]中，提出了基于 CUDA 的稀疏矩阵向量乘积运算内核的优化方案：挖掘同步空闲并行化，优化线程映射方式，优化存储访问，挖掘数据重用。文献[11]中实现了基于 GPU 的稀疏矩阵向量乘积运算。文献[12]中，作者在众核计算平台上提出几种并行化方案和快速同步

原函数。文献[13]中，基于 CPU 计算平台优化稀疏矩阵向量乘法性能受限于存储访问和数据重用，讨论了基于 CPU 计算平台优化稀疏矩阵向量乘法运算的许多稀疏矩阵的存储格式。文献[14]提出了一种新的稀疏矩阵的存储格式来减少存储带宽。文献[15]展示了一种稀疏矩阵向量乘法运算的存储优化方法。文献[16]中介绍了一种新的 Pattern-based Representation 方法，这种方法没有 0 元素填充，减少了索引的开销。文献[17]中引入了一种新的压缩稀疏块的存储格式，使得稀疏矩阵和向量的乘法运算以及稀疏矩阵转置和向量的乘法运算都能够高效的并行计算。

部分研究工作是针对于不同高性能计算设备的运算特性对稀疏矩阵内核运算的优化[3]。文献[19]提出了几种优化策略，对于多核环境特别有效，在 AMD 双核，INTEL 四核，异构 STI Cell 以及 Sun Niagara2 运算平台上都有着显著的性能提升。在文献[20]中，解决了 CUDA 架构下稀疏矩阵向量乘法算法并行化的一些问题。Nathan 和 Michael 有效地实现了基于 CUDA 的稀疏矩阵和向量乘积数据结构和算法，因为稀疏矩阵向量乘法运算的带宽受限特性，他们强调存储存储带宽效率和存储格式的压缩[3]。针对 GPU 难以发挥具有存储器瓶颈算法的效率的困难，提出了在 NVIDIA CUDA 架构上进行核心程序并行计算以及优化的主要因素，包括线程映射、合并访问、维度优化和数据复用等[3]。

当稀疏线性方程规模很大的情况下，在CPU上进行迭代法求解时耗时很长[3]。常见的迭代法有雅可比迭代法，高斯-赛德尔迭代法，共轭斜量法，逐次超松弛迭代法和广义最小残量法等。在求解大规模稀疏矩阵线性方程时，迭代法分支转移等操作较少，包含的计算量很大，在迭代过程中对额外存储空间的需求较小，适于GPU上的运算[3]。葛振等人分别在NVIDIA和AMD两种GPU 平台上实现PQMRCGSTAB算法,取得了较好的加速效果[3]。

### 1.2.3 GPU 的工作模式

CPU 具有独立的内存和寄存器，GPU 也具有独立的显存和寄存器。CPU 作为主控制器， CPU 和 GPU 协同处理任务，GPU 主要处理可以高度并行的数据处理任务,CPU 则负责逻辑处理和串行计算相关任务。把逻辑处理和串行计算任务分配给 CPU 处理，并行计算部分则交由 GPU 处理。GPU 上的程序被称为内核函数，也叫 kernel。kernel 是并行执行的程序段。在一段程序中可以有多个内核函数，每个内核函数内部都是并行执行的，但是各个 kernel 之间确是是串行执行的，其中还可以穿

插 CPU 代码段。CUDA 程序执行步骤：CPU 完成初始化工作，将参与并行运算的数据拷贝到显存中，GPU 上启动内核函数，在 GPU 上执行并行运算程序，GPU 运算完成后将数据结果由显存传送回 CPU 内存。CPU 主要负责逻辑控制及辅助运算任务在整个程序执行过程之中。过去 GPU 和主机连接采用过图形加速接口 AGP（Accelerated Graphic Ports）连接方式，现在 GPU 和主机通过 PCI-E 总线进行连接。在程序设计中应尽量少使用分配内存，拷贝数据等涉及到 CPU 和 GPU 数据交换的这些命令。GPU 的硬件特性使得 block 与 block 之间的通信很难，但是 CUDA 中的全局存储器允许多个 GPU 或者同一个 GPU 的多个 block 同时对一块存储空间进行访问。

### 1.2.4 GPU 的编程模型

目前 GPU 通常采用的是 CUDA 编程模型。CUDA 下开发语言是 C 语言但不是标准的 C 语言。开发时需要对函数进行标示，指明函数是在 GPU 上运行，还是在 CPU 上运行；通过设置网格和线程块，控制内核函数的并行执行。nvcc 是 CUDA 专门的编译器。nvcc 通过命令进行分阶段控制编译，其实就是一种编译器驱动。

CUDA 的程序经过编译后得到的是 GPU 上的代码，需要借助 CUDA 的 API 来完成启动 kernel 函数、在 GPU 配置显存等操作，这些 API 分为运行时 API 以及驱动 API 两种。

采用 CUDA 编程模型的 GPU 程序编译及运行的基本流程：一）分离源程序中的 GPU 代码和 CPU 代码；二）调用各自的编译器分别编译，CPU 上的代码由 C 编译器编译，GPU 上代码由 nvcc 进行编译，编译后输出二进制代码；三）链接 CPU 已经生成的代码与 GPU 上的代码；四）加载已经完成编译的 kernel 函数，将 GPU 的配置代码转化成 CUDA 的启动代码。

在 NVIDIA 公司 Tesla 系列的 GPU 上，流处理器阵列处于硬件架构的最高一层。流处理器阵列被分为两层，一层结构是由流式处理器 SM 组成，GPU 内部具有众多流处理器是其具备强大的计算能力的重要原因。另一层由线程处理器群组成，不同核心的 GPU 所具有的线程处理器群数量不同。流处理器 SM 具有 8 个线程处理器。线程处理器也叫做流处理器 SP，在同一个 SM 中的 8 个 SP 又被分成了两组，4 个 SP 处理单元为一组。流处理器 SP 并不是真正意义上的一个完整的处理器，它拥有自己独立的内部寄存器和指令指针，它却缺少调度单元和取地址单元。流处理器 SM

采用的是单指令多线程的调度模型。单指令多线程的调度模型不同于以往的单指令多数据模型。单指令多数据模型是基于向量的操作，需要将数据转化成向量格式后才能运算。由同类型数据所组成的数组可以由向量处理单元进行并行计算。所以单指令多数据模型处理数据时硬件结构会制约向量长度，最后会按照若干周期输出最终的运算结果。GT200 系列之后的 GPU 都具有了对双精度浮点数运算的支持。

每个流多处理器都有自己私有的寄存器，这是为了尽可能提高运算的速度。计算能力为 1.0 的 GPU 上，每个 SM 就只具有 8KB 的寄存器空间。CUDA 架构计算能力为 1.2 的 GPU 上，每个 SM 都有 16KB 大小的寄存器，所以每个流处理器 SP 就能分得 2KB 大小的寄存器空间。线程在运算时对局部存储器访问的速度很慢。一般在运算过程中产生的中间变量将会被保存在寄存器中。只有在寄存器的存储空间被用完时，才会把这些中间变量放入局部存储器中。共享存储器也是一种可以被高速读取的存储器。在同一个块内的线程都能访问共享存储器。常数存储器是位于片外的显存上的，它是只读的且具有缓存，访问速度较快。有关 GPU 的组织结构与存储模型如图 1.2 所示。

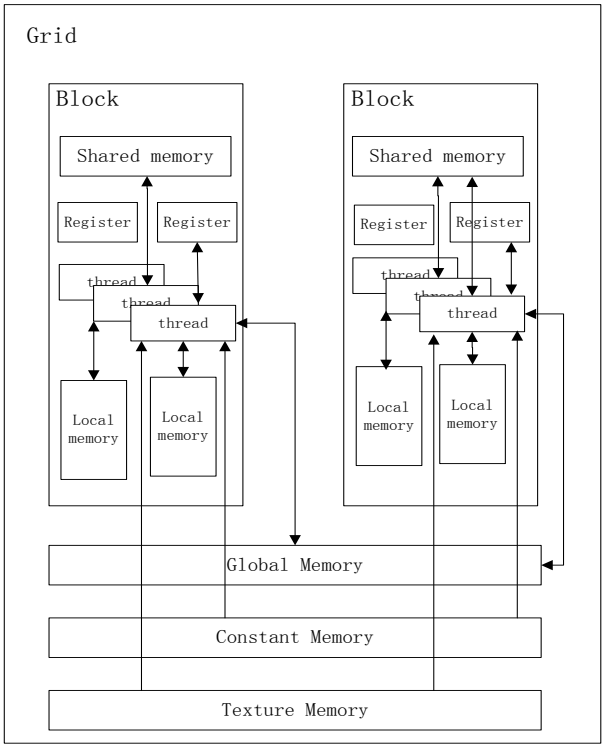


图 1.2 GPU 组织结构和存储模型

CUDA 中的基本逻辑执行单位：网格（grid）、线程块（block）、线程（thread）和线程组（warp）。一个内核函数对应一个网格。一个网格中有多个线程块，线程块

是内核函数执行的基本单位，线程块之间是无序并行执行的，并且不同的线程块之间是没办法相互通信的。一个线程块可最多由 512 个线程构成。线程是 CUDA 编程模型中可以分配的最小单位，也是资源最终的持有者。每个线程都有独立的 register 和 local memory，同一个线程块的线程之间可以相互进行细粒度通信。CUDA 中存在 grid、block 以及 thread，这些都是从硬件架构一一映射而来的。warp 是 CUDA 程序运行时的实际执行单位。因为 warp 是和硬件结构紧密相关，程序员无需知道 warp 的执行情况。CUDA 架构中一个 warp 由 32 个线程组成。

## 1.3 研究内容

本文主要以在 GPU 上稀疏矩阵运算为研究对象，在 GPU 平台下研究稀疏矩阵数值计算的优化方法，分析了算法在移植过程中遇到的问题，探讨了提高运算性能的多种方法，从核心运算稀疏矩阵向量乘和内积操作入手，在 NVIDIA GPU 平台上使用 CUDA 编程模型实现稀疏线性方程的求解运算的优化。

介绍了目前高性能计算 GPU 通用计算领域数值计算特别是稀疏线性矩阵运算的国内外研究情况以及面临的挑战。接着对 NVIDIA 公司 CUDA 架构的 GPU 通用计算进行分析，分别介绍了 GPU 的工作模式，GPU 的编程模型，GPU 的硬件架构和 GPU 的存储器结构。

针对 GPU 运算平台下稀疏矩阵向量乘法运算面临的挑战以及目前存在的影响该运算性能的主要问题，提出了两种基于不同稀疏矩阵存储格式的稀疏矩阵向量乘法的运算方法：分段行合并 CSR（compressed sparse row）的 SpMV（sparse matrix-vector multiplication）方法和按行分块的 SpMV 方法。分段行合并 CSR 的 SpMV 方法，采用行长度升序排序、根据行长度分段后合并相邻行等方式，解决负载不平衡和计算资源浪费的问题，从而提高 SpMV 的计算性能。按行分块的 SpMV 方法以一个 Block 为单位对齐并改变了矩阵元素在内存中的排布，一个线程可以读取连续的矩阵元素，同时一个线程组中的线程可以以合并访问（coalesced）的方式读取矩阵元素，减少因线程间不同步带来的性能损失。按行分块的方法使得每个 thread 的计算量均衡，避免了各个 thread 之间因计算任务不均衡而造成的等待。还对 CUDA 平台上稀疏矩阵向量乘法常用的全局存储器的访存操作进行优化，并使用纹理存储器和常数存储器对运算进行加速。

研究在 GPU 平台求解稀疏矩阵线性方程的雅可比和广义最小残量法两种迭代法

的实现方式和优化策略。GPU 上雅可比迭代法求解稀疏矩阵线性方程时通过采用在GPU 上分配两个解向量空间交替运算的方法，减少主存和 GPU 之间的数据传输次数并充分发挥GPU的运算能力进行精度判断;从而提高大规模稀疏矩阵线性方程在GPU平台求解的计算性能。GPU 上广义最小残量法求解稀疏矩阵线性方程时给出了向量内积和范数运算的 GPU 实现方法。并对 CUDA 平台上迭代法求解稀疏矩阵方程时的主机设备通信进行优化，此外还针对共享存储器的访存特性对共享存储器的访存进行优化。

## 1.4 文章组织结构

第一章首先介绍了目前高性能计算 GPU 通用计算领域数值计算特别是稀疏矩阵运算的研究情况以及面临的挑战。同时简述了 GPU 工作模式，GPU 的编程模型以及 GPU 的组织结构和存储模型。

第二章研究 GPU 上的稀疏矩阵向量乘法运算的实现机制和优化策略。针对于稀疏矩阵非零元素分布不均造成的空转问题以及同一 warp 中线程不能合并访存的问题，提出了一种分段行合并存储策略的稀疏矩阵向量乘方法。针对于一个 warp 内的线程间计算量负载不均衡而造成的线程间等待问题以及因线程不满足对全局存储器的合并访问要求而造成的访存延迟问题，提出了一种按行分块存储策略的稀疏矩阵向量乘方法。针对以上两种方法实现了全局存储器访存优化并使用纹理存储器和常数存储器对运算进行加速。

第三章研究 GPU 上的稀疏矩阵线性方程求解运算的实现机制并优化。实现了基于 GPU 的雅可比迭代法求解稀疏矩阵方程以及 GPU 上广义最小残量法求解稀疏矩阵线性方程的运算。给出了主机设备通信优化方案以及共享存储器访存优化方案来对 GPU 上的稀疏矩阵方程求解运算进行加速。

第四章测试了 GPU 稀疏矩阵向量乘法性能和 GPU 稀疏矩阵线性方程求解的性能，并对结果做出分析。

第五章对工作做出总结，明确后续的研究内容和研究方向。

# 2 稀疏矩阵向量乘法在 GPU 上的实现与优化

稀疏矩阵向量乘法运算是科学计算中的核心计算问题，特别是在使用迭代法求解线性方程时，稀疏矩阵向量乘法运算占据了极大的比重。在 GPU 通用计算平台上实现稀疏矩阵向量乘法运算较稠密矩阵运算更具有挑战性。这主要是因为稀疏矩阵存储格式对 GPU 访存的影响，非零元素的随机分布可以导致线程之间的计算负载不均衡以及通用计算编程优化方面的挑战。针对于以上这些问题，本章提出了两种基于 GPU 的稀疏矩阵向量乘的实现方法，并进行了相关优化。

## 2.1 相关工作

NVIDIA研究团队09年在基于CUDA编程模型的GPU上实现了多种格式稀疏矩阵向量的乘法运算，他们的并行化工作获得了非常好的加速效果，对于CSR格式存储的的稀疏矩阵向量乘法运算，采用的是一个warp对稀疏矩阵的一行进行乘法运算然后规约求和的方法[30]，该方法最高可获得16.6 GFLOP/s的浮点运算速度。后来上海复旦大学的马超等人在GPU上对非零元素最多的行与最少的行的非零元素个数相差很大的情况以及大多数行的非零元素个数都很少的情况进行了优化[30]，首先对行分割进行了优化，使得每一个流式处理单元的计算负载均衡；然后对float4数据类型进行了优化，使得每个线程的计算量提升为原来的4倍，获得了2倍左右的加速比。武汉大学的研究团队也从优化存储格式，优化线程映射以及避免分支判断三个方面对CUDA架构下的稀疏矩阵向量乘法运算进行了优化[30]，也取得了很好的优化效果。

### 2.1.1 稀疏矩阵的存储格式

稀疏矩阵的存储格式有许多种。通常针对于不同特性的稀疏矩阵会采用不同的存储结构，比如协调存储格式 COO（coordinate format），对角存储格式 DIA（diagonal format），行压缩存储格式 CSR 等。

COO 格式是一种简单的存储方案，采用三个数组存储行标识，列标识和非 0 元素的值。COO 是一种通用的存储格式，缺点是存储的效率不高。稀疏矩阵的 COO 存储格式如图 2.1 所示。

$$\mathtt{A} = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

$$\mathtt{row} = \begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 2 & 3 & 3 \end{bmatrix}$$
$$\mathtt{col} = \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix}$$
$$\mathtt{data} = \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix}$$

图 2.1　稀疏矩阵的 COO 存储格式

尽管不是一种很通用的存储格式，DIA 对于特定的对角矩阵的存储效率是非常高的。DIA 由两个数组组成，存储非 0 元素的 data 数组和存储每一个对角行相对于主对角行偏移量的 offsets 数组。DIA 存储格式的优势在于节省了存储空间，减少了运算时的数据传输量。并且对于该存储格式稀疏矩阵的所有访存都是连续的，保证了数据传输的性能。稀疏矩阵的 DIA 存储格式如图 2.2 所示。

$$\mathtt{A} = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

$$\mathtt{data} = \begin{bmatrix} * & 1 & 7 \\ * & 2 & 8 \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix} \qquad \mathtt{offsets} = \begin{bmatrix} -2 & 0 & 1 \end{bmatrix}$$

图 2.2　稀疏矩阵的 DIA 存储格式

使用的最为广泛的是 CSR 行压缩存储。CSR 格式的稀疏矩阵使用 3 个数组存储。。CSR 格式适合于一般的稀疏矩阵，可找到矩阵中任意元素的值，并很快得到每一行中的非零元素个数。稀疏矩阵 CSR 存储格式如图 2.3 所示，其中 data 数组用于按行存储非零元素，数组 indices 用于标识非零元素的列索引，数组 ptr 用于存储每一行的位置，其包含指向每一行开始元素的指针，因此 ptr[i]的内容为对应数组 data 和 indices 在第 i 行开始的位置。

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

$$\begin{aligned} \text{ptr} &= \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix} \\ \text{indices} &= \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix} \\ \text{data} &= \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix} \end{aligned}$$

图 2.3 稀疏矩阵 CSR 存储格式

### 2.1.2 稀疏矩阵向量乘法运算分析

基于 GPU 平台的稀疏矩阵向量乘法存在的挑战有：线程映射和存储层次的管理问题[30],数据访问和数据复用[30]，非零元素分布不均[30] 。

常用的基于CSR格式、利用CUDA加速的SpMV有两种实现方法：1）scalar kernel 方法：使用一个线程来处理一行，于是一个 warp 可处理 32 行；但由于每行长度的不同，造成空转问题非常严重，且无法联合访问 GPU 显存，访存效率低下；2）vector kernel 方法：使用一个 warp 来处理一行，从一定程度上减轻了空转问题，warp 内可联合访问，效率有所提高；但空转问题依然严重，访存效率尚存提高的空间。基于 CSR 存储格式的 GPU 平台下的稀疏矩阵和向量的乘法运算还存在着同一个 warp 内的线程间计算量负载不均衡而造成的线程间等待问题以及因线程不满足对全局存储器的合并访问要求而造成的访存延迟问题，影响了稀疏矩阵方程求解的性能。

## 2.2 GPU 上分段行合并存储策略的稀疏矩阵向量乘方法

针对 CUDA 架构上 SpMV 方法的负载不平衡和计算资源未充分利用的问题，提出一种基于分段行合并 SC-CSR（segmentation combined）的 SpMV 方法，采用行长度升序排序、根据行长度分段后合并相邻行等方式，解决负载不平衡和计算资源浪费的问题，从而提高 SpMV 的计算性能。

### 2.2.1 SC-CSR 存储格式

SC-CSR 存储格式由五个一维数组组成，分别为 length、row_index、val、col_ind 和 row_ptr。length 存储矩阵每行的行长度，row_index 存储每行实际对应的行索引，

val 存储非零元素（包括补齐的 0 元素），col_ind 存储非零元素的列索引（补齐的 0 元素列索引为-1），row_ptr 数组存储 SC-CSR 每行第一个非零元素的索引，最后一个值为非零元素总数。

SC-CSR 存储格式示意图如图 2.4 所示，数据存储格式 SC-CSR 产生步骤如下：

步骤 1：利用快速排序算法对每行的长度值数组 length 进行升序排序，并更新每行的实际行索引值 row_index。

步骤 2：根据行长度区间进行分段，每段合并不同的行数，长度值越大的合并的行数 c 值越小，对每 c 行元素进行按列读取操作，即 c 行所有第一个非零元素为 SC-CSR 新的一行的前 c 个元素，依次类推，行长度小于最长行的则补零，最后一段中行数不是 c 的整数倍时则添加长度为 0 的行。

步骤 3：val 存储合并后的非零元素（包括补齐的 0 元素），col_ind 存储非零元素的列索引（补齐的 0 元素列索引为-1），row_ptr 数组存储 SC-CSR 每行第一个非零元素的索引，最后一个值为非零元素总数。

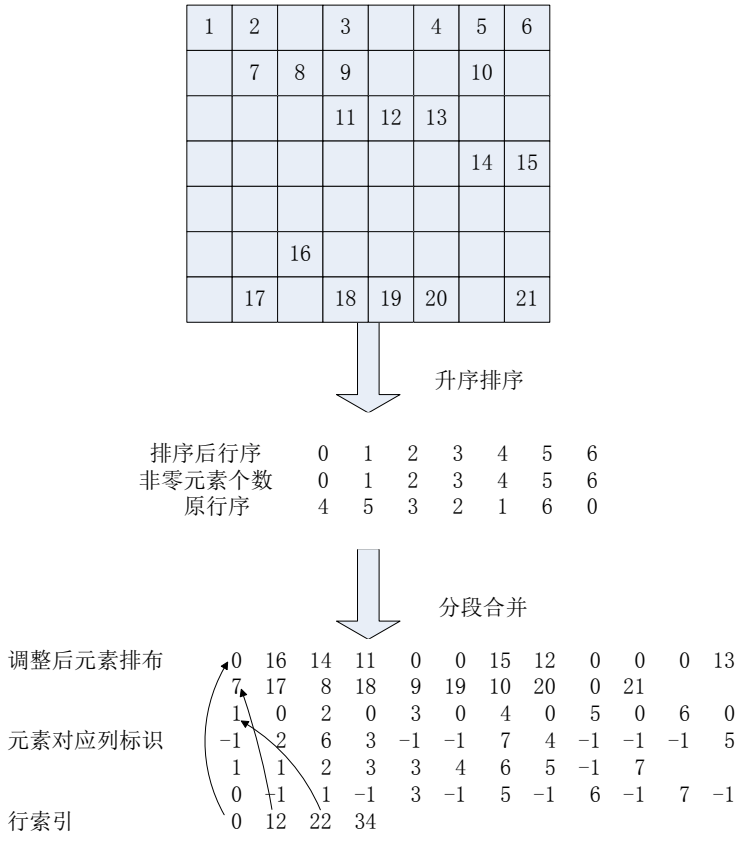


图 2.4 SC-CSR 存储格式示意图

### 2.2.2 基于 GPU 平台 SC-CSR 存储格式的 SpMV 方法

首先对行长度做升序排序，并根据排序结果来对行索引进行更新，然后利用补零方法对长度接近的相邻行进行分段合并，并分别调用不同的 kernel，最后对结果进行顺序还原，采用这种方法有效地提高了稀疏矩阵向量乘的运算性能。

基于 SC-CSR 存储格式的 SpMV 方法的流程如图 2.5 所示。

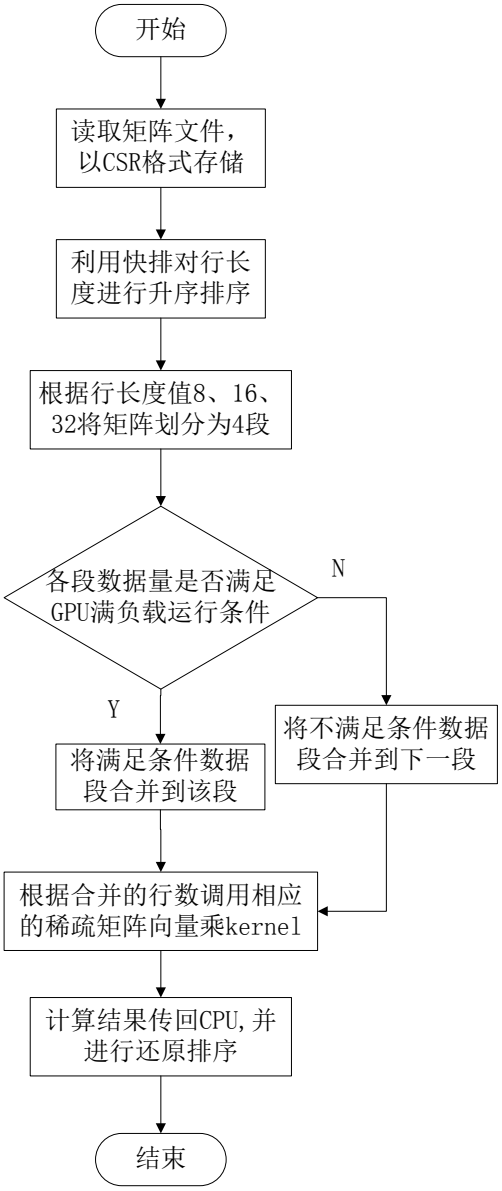


图 2.5 基于 SC-CSR 存储格式的 SpMV 方法的流程

基于 SC-CSR 存储格式的 SpMV 方法的具体步骤如下：

步骤 1：读取矩阵文件，按 CSR 格式存储，将存储 CSR 每行长度值的数组 length

采用快速排序方法进行升序排序，并更新每行的实际行索引值 row_index。

步骤 2：排序后的数组按长度值区间[0, 8)，[8, 16)，[16, 32)，[32, +∞)划分为 4 段，根据各段长度值的不同，使用不同的 c 值。长度值越小的段，使用越大的 c 值，从而减少合并后行长度的变化幅度，来达到 warp 间的负载平衡。合并是为了避免行长度小于 32 时 warp 内计算资源的空闲，因此，第一段的 c 为 32，第二段的 c 为 16，第三段的 c 为 8，第四段的 c 为 4。

步骤 3：根据各段处理的数据量是否达到 GPU 满负荷运行的条件。如公式 2.1 所示，$N_w$ 为 GPU 满载需要的 warp 数目，NSM 为 GPU 的多处理器数目，$N_b$ 为活动线程块数目，$S_b$ 为线程块的大小，$S_w$ 为 warp 的大小。GTX295 的 $N_w$ 为 480 个 warp，决定是否启动一个 GPU kernel（内核）。即第一段的行数除以第一段的 c 值 32，若小于 $N_w$，则划分为下一段处理，以此类推。

$$N_w = 0.5 \times NSM \times N_b \times (S_b / S_w) \tag{2.1}$$

步骤 4：根据划分的各段，更新 val、col_ind 和 row_ptr 的值，由此产生 SC-CSR 存储格式。如第一段，CSR 中每 32 行合并产生 SC-CSR 的一行，以列顺序存储在 val 中，即 SC-CSR 的前 32 个元素为 CSR 中每行的第一个非零元素，32 行中长度小于最长行的则补零，以此类推。各段中行数不足 c 的整数倍时，在段后添加零元素行来补齐，0 元素对应的 col_ind 都是-1。

步骤 5：CPU 将产成的 SC-CSR 格式数据传输到 GPU 显存，并根据各段合并的行数，分别启动不同的 kernel，对 col_ind 值为正的元素进行处理，计算完后每段 kernel 的归约次数为 log(32/c)。

步骤 6：GPU 将计算结果传回 CPU。CPU 根据 row_index 的实际行索引值，将 GPU 的计算结果进行还原排序，从而得到最终的计算结果。

## 2.3 GPU 上按行分块存储策略的稀疏矩阵向量乘方法

针对于 GPU 平台的运算特性对稀疏矩阵的存储进行改进，采用按行分块 VAB（vertical aligned blocks）存储策略以一个 Block 为单位对齐并改变了矩阵在内存中的排布，一个线程可以读取连续的矩阵元素，同时一个 warp 中的线程可以以 coalesced 的方式读取矩阵元素，减少因线程间不同步带来的性能损失。按行分块的方法使得每个 thread 的计算量均衡，避免了各个 thread 之间计算任务不均衡而造成的等待。

### 2.3.1 按行分块（VAB）存储策略

对 CSR 稀疏矩阵的存储进行改进。稀疏矩阵 CSR 存储结构的示意图如图 2.6 所示，elems 数组存储的是稀疏矩阵按行排列的非 0 元素，rowptr 数组存储的是稀疏矩阵每一行第一个非零元素在 elems 数组中的位置。定义每个块的大小为 4。将稀疏矩阵中每行的元素按照逻辑顺序依次填充至各个线程的块中。若当前行元素未填满该行最后一个待填充的块则补 0 元素填充该块，然后开始下一行的填充。在填充块的同时，记录各行第一个块中间结果在所有块中的顺序号并存入到行指针数组中，采用 VAB 方法调整稀疏矩阵存储结构的原理示意图如图 2.7 所示，黄色方格中即为各行第一个块中间结果在所有块中的顺序号。 求出每一个 Block 中具有的块数，每一个 Block 可以计算的元素个数，实际运算中计算的块个数，实际运算中计算的元素个数，并将所有元素初值赋为零。对于每一行中的各个块，计算其所在 block 序号，以及 block 内的排序情况，按照物理横向存储，逻辑纵向存储的方式计算其存储地址，并将相应元素的值存入对应的地址中。蓝色的方格表示的是稀疏矩阵中的非 0 元素。白色方格表示的是当前行元素未填满该行最后一个待填充的块补充的 0 元素。一个 block 中有 BS（Block Size）个线程，每一个线程按照纵向矩阵存储元素的逻辑顺序计算各行每一个块中元素的累加和。但是实际上稀疏矩阵在物理存储中的排布方式是横向的，每一个方格中中的标号就是该方格元素在物理存储中的相对位置，该相对位置以物理存储中的第一个元素为基准。thread0 线程计算相对位置为 0 的元素，thread1 线程计算相对位置为 1 的元素，直到 block 的最后一个线程 thread（BS-1）计算相对位置为 BS-1 的元素，紧接着 thread0 计算相对位置为 BS 的元素，后面的线程与对应计算元素的关系同上。BS 的大小为 64 的整数倍。这样做的好处是满足 warp 中线程对显存合并访问的条件，同时各个线程的计算任务基本均衡，从而避免了线程等待现象，使运算加速。

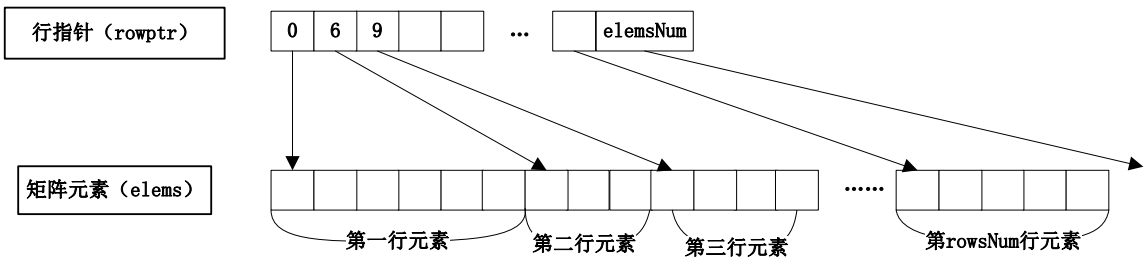


图 2.6　稀疏矩阵 CSR 存储结构示意图

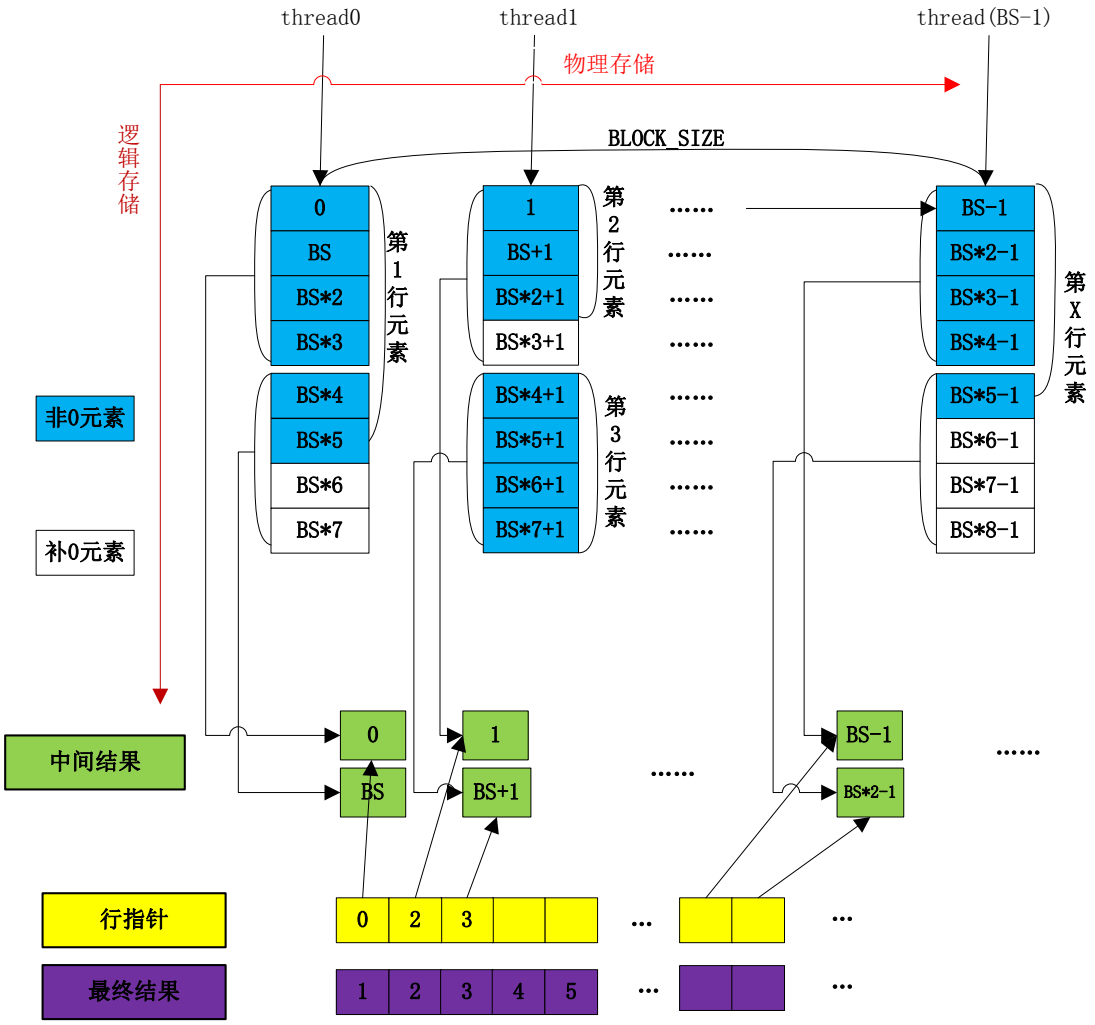


图 2.7 采用 VAB 方法调整稀疏矩阵存储结构的原理示意图

## 2.3.2 基于 GPU 平台的 VAB 存储格式的 SpMV 方法

采用 VAB 存储格式的优势是满足了对 GPU 的全局存储器合并访问条件以及使得同一线程组中的线程计算任务均衡，整个计算过程主要分为两大步骤。首先是对各个块中的元素及其对应向量元素的乘积进行求和运算，获得中间结果。然后根据行指针数组对中间结果进行合并，得到最终的结果。在 GPU 平台采用按行分块的方法计算稀疏矩阵和向量乘法的运算步骤如下：

(1) 对稀疏矩阵的存储进行改进。定义每个块的大小为 4。将稀疏矩阵中每行的元素按照逻辑顺序依次填充至各个线程的块中。若当前行元素未填满该行最后一个待填充的块则补 0 元素填充该块，然后开始下一行的填充。在填充块的同时，记录

各行第一个块中间结果在所有块中的顺序号并存入到行指针数组中。

(2) 对于每一行中的各个块，计算其所在 block 序号，以及该块在该 block 内的排序情况，按照物理横向存储，逻辑纵向存储的方式计算其存储地址，并将相应元素的值存入对应地址的内存中。在 GPU 的全局存储器中分配稀疏矩阵，待乘向量，中间结果和最终结果需要的显存空间，并把其中所有元素的初值赋值为 0。将调整存储结构后的矩阵和待乘向量由内存拷贝至 GPU 存储器中，显存中的稀疏矩阵元素的排布与内存一致。

(3) 将存储在内存中的矩阵元素拷贝到显存的全局存储器中。

(4) GPU 的每个线程计算每一个块中的 4 个元素与其对应向量元素乘积的累加和，即块中间结果，并将结果写入显存全局存储器，绿色方格中即为中间结果。

(5) GPU 根据块中间结果和在(4)中计算出的块中间结果对应的行指针，计算出每一行的所有块中间结果的累加和，此累加和为稀疏矩阵的该行与该行对应向量乘积的最终结果，并将结果写入显存。紫色方格中的编号即为结果向量中的元素编号。

该方法使得一个 warp 中的线程可以按照合并访问的方式访问全局存储器，减少了线程间因访问全局存储器不同步带来的性能损失。按行分块的方法使得线程间的计算量均衡，避免了各个线程之间因计算任务不均衡而造成的等待。

## 2.4 基于 GPU 的稀疏矩阵向量乘的优化方法

### 2.4.1 全局存储器的访问优化

对 CUDA 程序性能影响最明显的因素之一就是对全局存储器的访问是否满足合并访问条件，满足全局存储器的合并访问条件与否有时候会严重的影响 GPU 程序的性能。当来自一个 half-warp 的 16 个线程对全局内存进行装载或者存储访问时，满足访问条件时只需要一次传输就可以处理这些线程的访存请求；合并访问要求同一 half-warp 中的线程要按照一定的字长来访问经过对齐的段[30]。

全局存储器上通过合并访问机制可以屏蔽访存指令的访存延时，而由于 GPU 有不同层次的存储器模型，这些存储器又有不同的访问带宽和访问延时，所以合理的对存储器访问进行规划，通过合并访问机制和足够的运算指令来隐藏访存延时，才可以获得较高的性能[30]。

half-warp 的任何内存请求模式都将得到合并，包含多个线程存取同一个地址的

模式。如果 half-warp 存取的字在不同的 N 个内存段中，将有 N 个内存传输被执行，特别的，如果线程存取 16 字节的字，至少要进行两次内存传输。全局存储器的访问示例如图 2.8 所示，给出了数据对齐和非对齐的访问模式，当出现两个 128Byte 段内的非对齐访问时，会产生两次数据传输[30]。

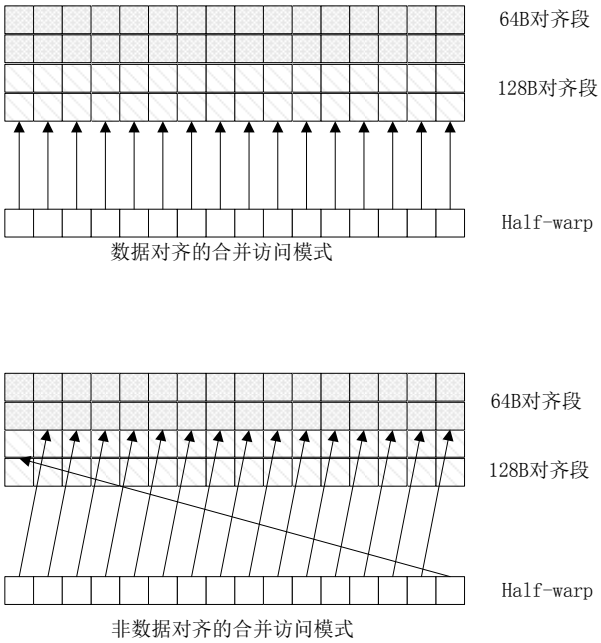


图 2.8　全局存储器的访问示例

在基于 CUDA 架构的 GPU 平台上，将 kernel 函数的 block 大小设定为 64 的倍数并对于稀疏矩阵的存储格式进行调整。GPU 上 SpMV 运算的全局存储器访问优化如图 2.9 所示，线程在对 VAB 存储结构的稀疏矩阵访问时始终满足对于全局存储器的合并访问条件，避免了访存的延迟，提高了运算的性能。

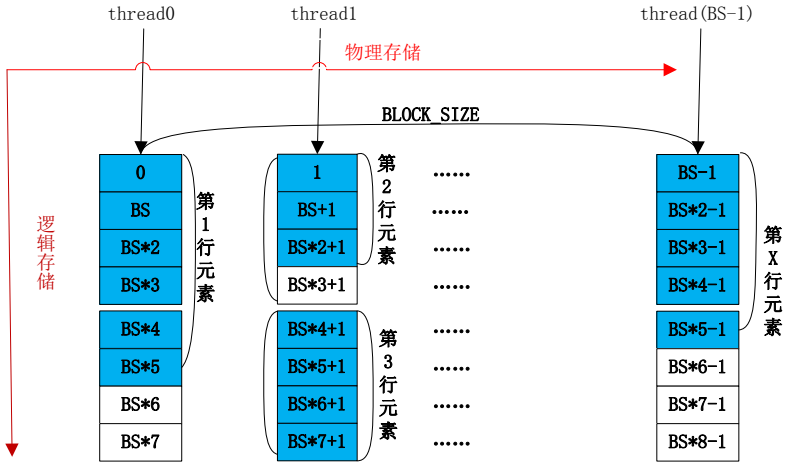


图 2.9　GPU 上 SpMV 运算的全局存储器访问优化

### 2.4.2 使用纹理存储器和常数存储器加速

纹理存储器能够通过缓存利用数据的局部性，提高效率。使用纹理存储器时的好处有：不用严格遵守合并访问条件，也能获得很高带宽。对于随机访问，如果要访问的数据并不是很多，效率也不会特别差。可以使用线性滤波和自动类型转换等功能调用硬件的不可编程计算资源，而不必占用可编程计算单元。常数存储器主要用于存放指令中的常数。对当前的硬件来说，如果一个 half-warp 的线程访问常数存储器中相同的一个数据，可能只需要一个周期就可以获得这个数据。实际使用常数存储器时速度一般低于立即数或者共享存储器，但还是明显高于将数据存放在显存中的情况。有因为尺寸太大无法存放在寄存器或者共享存储器的常数数组，可以将其放在常数存储器中获得加速。GPU 上 SpMV 运算的纹理内存绑定加速如图 2.10 所示，结果向量可以实现 coalesced 方式的写入，对于中间结果数组的访问不能实现 coalesced 方式的读取，对于中间结果数组使用纹理内存绑定可以一定程度上提高访存效率。

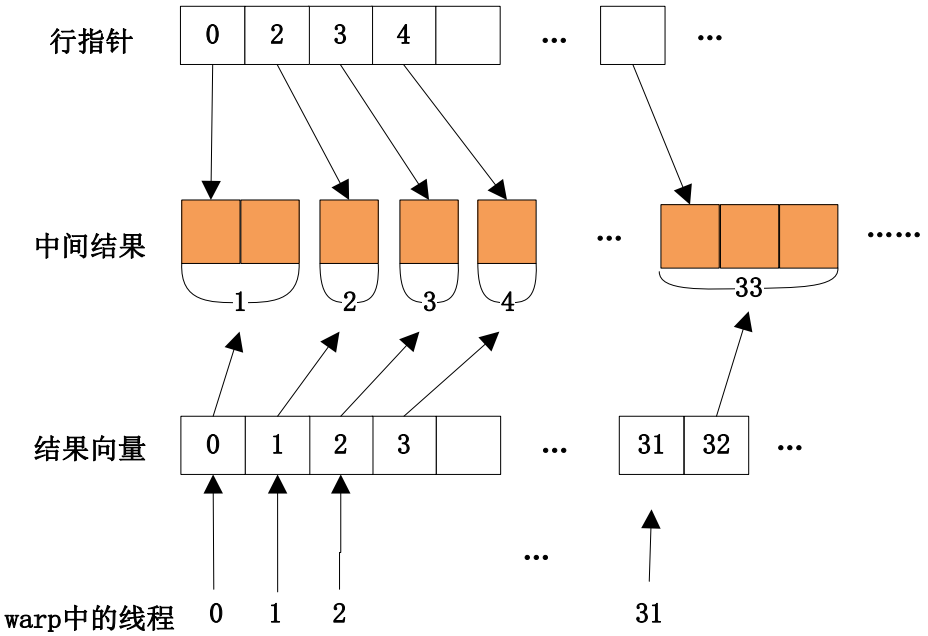


图 2.10 GPU 上 SpMV 运算的纹理内存绑定加速

在 GPU 上进行稀疏矩阵运算时，将参与运算的常量置入常量存储器，加快访问速度。将不能满足全局存储器合并访问条件运算的中间运算结果置入纹理存储器，节省了存储空间并使运算得到加速。

## 2.5 本章小结

本章首先分析了基于 GPU 运算平台的稀疏矩阵向量乘法运算面临的挑战以及目前存在的影响该运算性能的主要问题。并提出了两种基于不同稀疏矩阵存储格式的稀疏矩阵向量乘法的运算方法。

分段行合并 CSR 的 SpMV 方法，采用行长度升序排序、根据行长度分段后合并相邻行等方式，解决负载不平衡和计算资源浪费的问题，从而提高 SpMV 的计算性能。

按行分块的 SpMV 方法以一个 Block 为单位对齐并改变了矩阵在内存中的排布，一个线程可以读取连续的矩阵元素，同时一个 warp 中的线程可以以合并的方式读取矩阵元素，减少因线程间不同步带来的性能损失。按行分块的方法使得每个 thread 的计算量均衡，避免了各个 thread 之间计算任务不均衡而造成的等待。

最后，对 CUDA 平台上稀疏矩阵向量乘法常用的全局存储器的访存操作进行优化，并使用纹理存储器和常数存储器对运算进行加速。

# 3 稀疏矩阵线性方程求解在 GPU 上的实现与优化

大规模稀疏线性方程通常采用迭代法求解。本章重点研究迭代法求解稀疏矩阵线性方程在 GPU 上的实现，选取了两种具有代表性的雅可比迭代法和广义最小残量法在 CUDA 架构上实现并进行优化。其中雅可比迭代法是求解大型线性方程组的基本方法也是科学计算领域常用的计算核心。求解稀疏线性方程组的广义最小残量迭代法是也许多大型应用问题的核心运算，是求解稀疏矩阵线性方程非常有效的方法之一，一直以来它都是在数值求解中研究的重点。

## 3.1 相关工作

文献[25]中在两种主流GPU 平台上实现了一个经典的迭代法PQMRCGSTAB，并且针对不同的GPU平台特点提出了具体的优化方法。文献[33] 基于AMD的流处理GPU平台设计并实现了Jacobi算法，相对于CPU平台取得了很好的加速效果。文献[34]和文献[35]改进了雅可比迭代法和高斯—塞德尔迭代法的实现过程，从而提高了求解线性方程组的速度，并研究了对于不同规模的方程，GPU对这两种迭代算法的加速效果。

## 3.2 基于 GPU 的雅可比迭代法

### 3. 2. 1 算法原理和算法分析

设线性方程组 $Ax = b$ 的系数矩阵 $A$ 可逆且主对角元素均不为零，令 $D = diag(a_{11}, a_{22}, \dots, a_{nn})$ ，并将 $A$ 分解成 $A = (A-D)+D$，从而得到 $Dx = (D-A)x+b$，令 $x = B_1x+f_1$ 其中 $B_1 = I-D^{-1}A$，$f_1 = D^{-1}b$。故以 $B_1$ 为迭代矩阵的迭代法公式为 $x^{(k+1)} = B_1x^{(k)}+f_1$，该公示又可称为称为雅可比迭代公式，用向量的分量可表示为：

$$x_i^{(k+1)} = \frac{1}{a_{ii}}[b_i = \sum_{\substack{j=1 \\ j\neq i}} a_{ij}x_j^{(k)}], i = 1,2,...n, k = 0,1,2... \tag{3.1}$$

根据上述推导过程，可以明确雅可比迭代法的处理步骤如下：

步骤 1：读取稀疏矩阵和向量文件；

步骤 2：调整稀疏矩阵的存储结构；

步骤 3：设求解的稀疏线性方程为 $Ax = b$。令 $D = diag(a_{11}, a_{22}, \ldots, a_{nn})$，求出 $B_1 = I\text{-}D^{-1}A$，$f_1 = D^{-1}b$；

步骤 4：以 $B_1$ 为迭代矩阵的迭代运算 $x^{(k+1)} = B_1x^{(k)}+f_1$；

步骤 5：检验 $\max\limits_{1\le i\le n}\left|x_i^{(k+1)} - x_i^{(k)}\right| < \varepsilon$ 是否成立来确定迭代是否终止。其中 $\varepsilon$ 为精度值。若条件成立则迭代结束 $x^{(k+1)}$ 即为方程的解向量，否则转置步骤（4）。

计算过程中的步骤（3）求解迭代矩阵 $B_1$ 和 $f_1$ 运算一次即可，故在 CPU 上执行该步骤运算。步骤（4）（5）涉及到多次迭代过程，是整个迭代运算的核心，这两个步骤需要进行 GPU 的并行化操作是优化的重点。步骤（4）中的矩阵和向量乘法运算使整个方程求解中运算量最大的部分，该步骤在 GPU 上进行。

### 3.2.2 判断收敛的 GPU 加速方法

每次迭代后，都需要判断结果是否达到收敛要求，而最终判断需要在 CPU 主机上完成，因此需要将迭代后的结果由 GPU 设备传回 CPU 主机，传输和 CPU 主机端的线性计算过程比较耗时，需要想办法加速。在 GPU 设备上存储 2 个解，设为 p_d1、p_d2，交替作为临时解。在 GPU 上计算 p_d1 与 p_d2 的差，将多个差值的绝对值最大者存储在数组 pSub_d 中传回 CPU 主机。这样可以有效地减少主机设备间传输和主机端进行判断所花费的时间，同时减少 CPU 主机判断收敛条件的时间。在比较多个差值绝对值的最大值的过程中，也实现了 GPU 并行计算，并且实现了对于全局存储器的合并访存方式，判断方程求解收敛的 GPU 加速方法如图 3.1 所示。

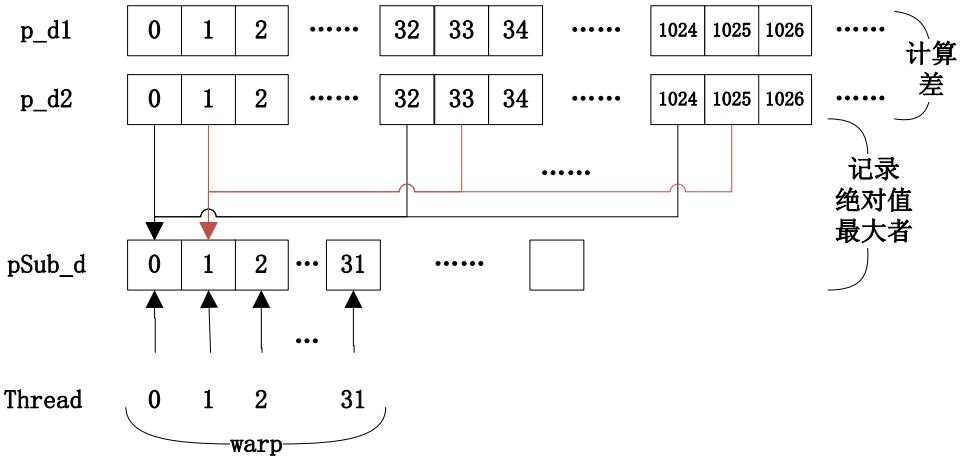


图 3.1 判断方程求解收敛的 GPU 加速方法

在 GPU 上分配两个解向量空间，这两个解向量空间分别作为本次的运算结果和上一次的运算结果交替进行步骤以 $B_1$ 为迭代矩阵的迭代运算 $x^{(k+1)} = B1x^{(k)} + f_1$ 的迭代运算，减少了主存和 GPU 存储器之间的数据传输。收敛可以分为两步完成，其中 $x_i^{(k+1)} - x_i^{(k)}\ (1 \le i \le n)$ 的运算具有很好的并行性，且两个解向量空间已直接保存在显存中，故可直接在 GPU 上执行；判断过程涉及到条件判断和分支计算故在 CPU 上进行。在 GPU 上分配两个解向量空间，这两个解向量交替参与迭代矩阵的迭代运算，显著减少了主存和 GPU 存储器之间的数据传输量；并且可以在 GPU 上直接计算两个解向量的差值，使计算获得加速。

### 3.2.3 雅可比迭代法的 GPU 实现

稀疏矩阵雅可比迭代法的核心运算是以 $B_1$ 为迭代矩阵的迭代运算和收敛的判断计算。经过分析要把这两部分运算并行化，可以得到 GPU 上稀疏矩阵方程雅可比迭代法求解流程如图 3.2 所示。

（1）读取矩阵和向量文件。

（2）矩阵文件是按照 CSR 格式存储的，对其进行按行分块的格式存储转换。

（3）在 CPU 运算平台求出迭代矩阵 $B_1 = I - D^{-1}A,\ f_1 = D^{-1}b$。

（4）在 GPU 运算平台显存中分配两个临时解数组空间 *tempx1* 和 *tempx2*，并将初始解数组拷贝至 *tempx1* 中。

（5）设最大迭代次数为 *max_iter*，迭代次数 *iteration* 初值置为 0。

（6）标志值 *flag* 置为 0。GPU 运算平台以 *tempx1* 作为临时解进行迭代运算 $x^{(k+1)} = B_1x^{(k)} + f_1$。迭代结果存入 *tempx2* 中，*ieration* 值加一。在 GPU 上计算临时解数组空间 *tempx1* 和 *tempx2* 的差值 *sub* 并将结果传至 CPU 内存。CPU 检验 sub 向量是否满足 $\max\limits_{1\le i\le n}\left|x_i^{(k+1)} - x_i^{(k)}\right| < \varepsilon$ 的条件，若满足跳至步骤（8），否则跳至步骤（7）。

（7）标志值 *flag* 置为 1。GPU 运算平台以 *tempx2* 作为临时解进行迭代运算 $x^{(k+1)} = B_1x^{(k)} + f_1$。迭代结果存入 *tempx1* 中，*iteration* 值加一。在 GPU 上计算临时解数组空间 *tempx1* 和 *tempx2* 的差值 *sub* 并将结果传至 CPU 内存。CPU 检验 *sub* 向量是否满足 $\max\limits_{1\le i\le n}\left|x_i^{(k+1)} - x_i^{(k)}\right| < \varepsilon$ 的条件，若满足跳至步骤（8），否则继续进行。判断 *iteration* 的值是否大于等于 *max_iter*，若是则跳至步骤（8），否则跳至步骤（6）。

（8）若标志值 *flag* 值为 0，则将 *tempx2* 由显存拷贝至内存中，作为解向量。若标志

值 *flag* 值为 1，则将 *tempx1* 由显存拷贝至内存中，作为解向量。最后将获得的解向量由 GPU 传送至主机内存中。

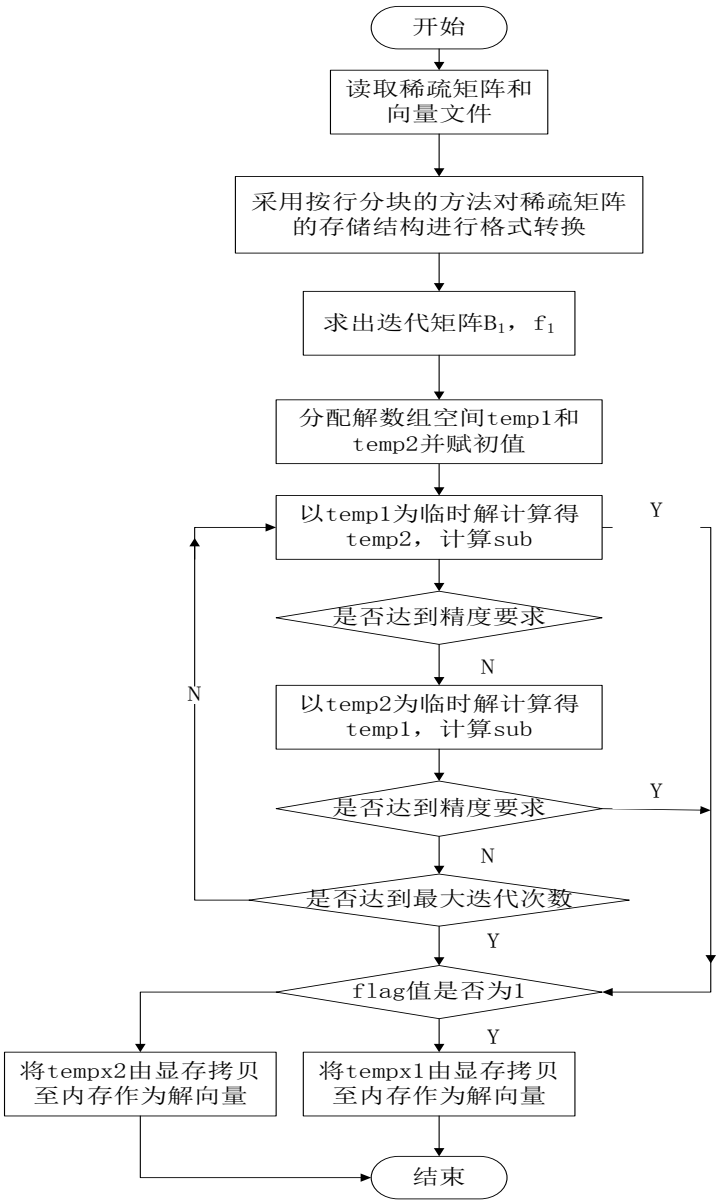


图 3.2 GPU 上稀疏矩阵方程雅可比迭代法求解流程

## 3.3 基于 GPU 的广义最小残量法

### 3.3.1 算法介绍和分析

在求解大型稀疏线性系统的迭代法中，广义最小残量法是十分高效的算法。广义

最小残量法是 *Krylov* 子空间方法的一种[30] 。文献[37]中给出了 *Krylov* 子空间方法的定义，并指出这是一种基于投影的迭代法。由文献[38][39]可以得到广义最小残量法的算法描述，广义最小残量法的算法描述如图 3.3 所示。

1. $Compute\ r_0 = b - Ax_0, \beta := \|r_0\|_2, and, v_1 := r_0 / \beta$
2. $Define\ the\ (m+1)\times m\ matrix\ \overline{H}_m = \{h_{ij}\}_{1\le i\le m+1, 1\le j\le m}, set\ \overline{H}_m = 0$
3. $For\ \ j = 1,2,\cdots,m\ \ do:$
4. $\quad Compute\ \omega_j := Av_j$
5. $\quad For\ \ i = 1,\cdots j\ \ do:$
6. $\quad\quad h_{ij} := (\omega_j, v_j)$
7. $\quad\quad \omega_j := \omega_j - h_{ij} v_j$
8. $\quad EndDo$
9. $\quad h_{j+1,j} = \|\omega_j\|_2. If\ \ h_{j+1,j} = 0\ \ set\ \ m := j\ \ and\ \ goto\ 12$
10. $\quad v_{j+1} = \omega_j / h_{j+1,j}$
11. $EndDo$
12. $Compute\ y_m\ \ the\ \ minimizer\ \ of\ \ \|\beta e_1 - \overline{H}_m y\|_2\ \ and\ \ x_m = x_0 + V_m y_m$

图 3.3　广义最小残量法的算法描述

设系数矩阵为 n 阶可逆矩阵，每行非零元数为 k，*krylov* 子空间维度为 m(常数)，则广义最小残量法所需计算的情况如表 3.1 所示：

表 3.1 广义最小残量法所需计算的情况

| 名称 | 复杂度 | 常系数 | 计算次数 |
|---|---|---|---|
| 稀疏矩阵与向量乘积 | O(kn) | 2 | m+1 |
| 向量内积与范数 | O(n) | 2 | m(m+3)/2+1 |
| 稠密矩阵与向量乘积 | O(n) | 2m | 1 |
| 向量加减、向量乘标量 | O(n) | 1 | 6 |
| 最小二乘求解 | O(1) | 4m^2 | 1 |

很显然无论从计算量来说还是从 GPU 程序的优化难度来说，表中的前两项都是整个算法中最耗时的部分。稀疏矩阵向量的乘积运算在上一章已讨论过，稠密矩阵与向量乘积，向量加减、向量乘标量运算简单在这里不再赘述。下面重点讨论向量

内积和范数运算的 GPU 实现。

### 3.3.2 向量内积和范数的 GPU 加速方法

算法中向量范数采用欧几里得范数，即 $\|\mathbf{a}\|_2=\sqrt{a_1^2+a_2^2+...+a_n^2}$，和向量内积计算是相似的，因此本文只介绍向量内积的计算。

内积计算的困难在于如何将每个标量乘积累加起来。我们使用规约的思想，将两个输入向量划分成若干对小向量，每个 block 负责计算一对小向量的内积，这些小向量的内积被写到映射存储中，由 CPU 负责将这些小向量内积加起来得到最终结果。

为了对不同问题规模都能得到最佳的 warp 块装载量，本文并未固定小向量的长度，而是在限制 block 的个数不大于 512 的情况下动态决定小向量的长度，因此得到的小向量不多于 512 对。设输入向量长度为 n，block 大小为 b，block 个数为 g，小向量长度为 m，向量内积和范数 GPU 实现的判定树 3.4 所示：

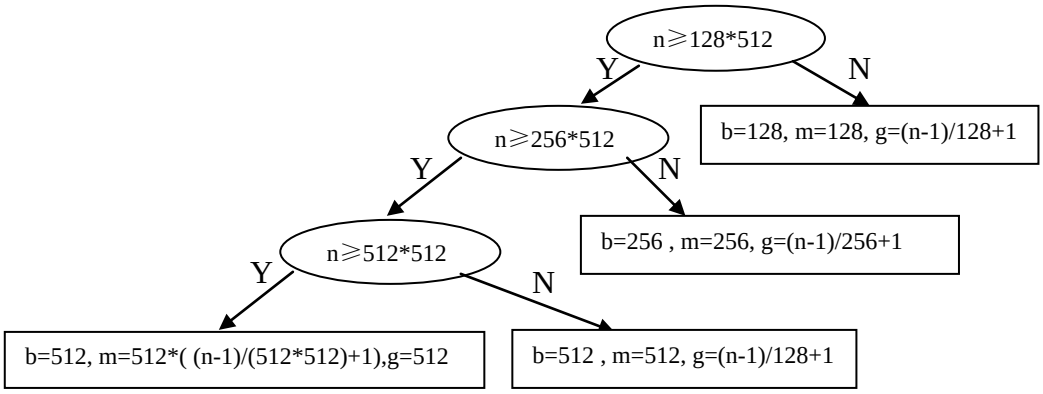


图 3.4　向量内积和范数 GPU 实现的判定树

容易看出，利用该判定树得到的线程和数据划分在任何时候都不会对 warp 块装载量产生影响。当 n≥512*512 时，尽管此时 block 的数量被限制为 512，但由于每个 SM 只能同时装载 2 个大小为 512 的 block，所以这个限制是合理的不会对性能产生影响。

以 block 的大小为 128 为例，kernel 部分的函数代码如下：

```
__global__ void innerPro_128( float *v, float *w, float *mres ){
    int tid = threadIdx.x ;
    int tid_in_grid  = blockIdx.x * blockDim.x + threadIdx.x ;
    __shared__ float r_s[128] ;
    r_s[ tid ] = v[tid_in_grid] * w[tid_in_grid] ;
```

```
    __syncthreads();
    if( tid < 64 ) r_s[tid] += r_s[tid + 64] ;     __syncthreads() ;
    if( tid < 32 ) {
        r_s[tid] += r_s[tid + 32] ;
        if( tid < 16 ) r_s[tid] += r_s[tid + 16] ;
        if( tid <  8 ) r_s[tid] += r_s[tid + 8 ] ;
        if( tid <  4 ) r_s[tid] += r_s[tid + 4 ] ;
        if( tid <  2 ) r_s[tid] += r_s[tid + 2 ] ;
        if( tid < 1 ) mres[ blockIdx.x ] = r_s[0] + r_s[1];
    }
}
```

### 3.3.3 广义最小残量法的 GPU 实现

根据上一节的讨论为了在求解时充分发挥 CPU/GPU 的运算能力，从而提高大规模稀疏矩阵方程求解的计算性能。算法处理具体步骤如下：

步骤 1：读取矩阵和向量文件。

步骤 2：选择 $x_0 \in R^n$ 并在 GPU 上分别计算 $r_0=b-Ax_0$，$v_1=r_0/||r_0||$，$\beta=||r_0||$。其中 A，b 是求解方程 Ax=b 中的矩阵和向量。$A \in R^{n*n}$ 非奇异，b 是 n 维列向量。$x_0$ 是初始解向量。

步骤 3：选择适当大小的 m，在 GPU 上完成 *Arnoldi* 运算过程得到 $\{v_i\}_{i=1}^{m}$ 和 $H_m$。通常情况下，我们可以采用以下方法来估计参数：$m=(1+\delta)(\log\varepsilon/\log\tau-1)-3-\delta$，其中 $\tau=||r_m||/||r_0||$，$\varepsilon=||r_{最终}||/||r_0|| < \tau$，$\log\varepsilon/\log\tau-1$ 为迭代的次数，δ 是求解矩阵中各行非零元素的平均数。$v_1, v_2, \ldots, v_m$ 是方程 m 维 *Krylov* 子空间的一组标准正交基。$H_m$ 是 *Arnoldi* 过程产生的一个 $(m+1)\times m$ 阶上 *Hessenberg* 矩阵。$H_m$ 满足 $AV_m=V_{m+1}H_m$。

*Arnoldi* 过程：

Step1：取 $v_1=r_0/||r_0||$。

Step2：对于 $k=1,2,\ldots,m$，计算 $\sum v_{k+1}=Av_k-\sum h_{i,k}v_i$（$i=1, \ldots k$）其中，$h_{i,k}=(Av_k, v_i)$，$h_{k+1,k}=||v_{k+1}||$，$v_{k+1}=\sim v_{k+1}/h_{k+1,k}$ 在此过程中如果 $h_{k+1,k} \neq 0$，当 $k=m$ 时，得到了一组向量 $\{v_i\}_{i=1}^{m}$ 和上 *Hessenberg* 矩阵 $H_m$。

步骤 4：在 CPU 上极小化 $||\beta e_1-H_m y||$ 得到 $y_m$。GMRES 算法求 $x_m$ 满足 $x_m=x_0+z_m$。其中 $z_m \in K_m$，满足最小二乘条件：

$$||r_m||=||b-Ax_m||=||r_0-Az_m||=min||r_0-Az|| \tag{3.2}$$

该式等价于 $r_m \perp Ak_m$ 令 $z=V_m y$，其中 $y \in R^m$，则（3.2）式右端极小化可表示为

$J(y)=||r_0-AV_my||=||r0-v_{m+1}H_my||=||V_{m+1}(||r_0||e_1-H_my)||$ (3.3)

由于 $V^T_{m+1}V_{m+1}=I$，所以$||r_0-Az||=||||r_0||e_1-H_my||$,由此（1）式的解 $x_m=x_0+V_my_m$ 其中 $y_m$ 极小化(3.3)式的 $J(y)$。

步骤 5：在 GPU 上计算 $x_m=x_0+V_my_m$。其中 $x_m$ 是使残量 $\beta e_1-H_my_m$ 的范数最小。

步骤 6：在 CPU 上计算$||r_m||=||b-Ax_m||$。如果精度满意，则转至步骤 8；否则转至步骤 7。其中精度要求根据实际应用需求确定。

步骤 7：令 $x_0=x_m,v_1=r_m/||r_m||$跳转至步骤 3。

步骤 8：将运算结果 $x_m$ 由 GPU 显存传至内存中。

## 3.4 基于 GPU 的稀疏矩阵方程求解优化

### 3.4.1 主机设备通信优化

目前大多数的显卡通过 PCI 总线与主机连接。当传输的数据量很大时，有限的带宽就成为了瓶颈。锁定存储是分配出一部分的内存专门用于与 GPU 存储器之间的数据传输而不参与和 CPU 及内存的数据传输工作，因而速度比较快。但是声明这些物理内存只会被分配给对应的 GPU 设备使用，占了操作系统的可用内存。这可能会影响到 CPU 运行需要的物理内存，所以在考虑整个系统的优化时，需要合理规划 CPU 和 GPU 各自使用的内存。在稀疏矩阵方程求解时通过分配锁定存储来加速主存和显存之间的数据传输，使运算得到加速。

异步执行是指 GPU 进行的操作从主机启动后，GPU 未完成时，CPU 就可以得到计算返回值继续下一步操作。CPU 可以在 GPU 数据传输或者运算的时候进行计算，从而更高效地利用计算资源。通过不同流之间的异步执行，使流之间的传输和运算能够同时执行，更好的利用 GPU 资源。不使用异步执行时的程序运行时间为（执行时间+传输时间）；而使用流和异步后时间降为（执行时间+传输时间/流的数量）。此时，CPU 和 GPU 间的数据传输时间被有效的隐藏了，程序的性能得到了改善。在 GPU 上进行稀疏矩阵方程求解时会有多个 kernel 函数参与计算，将这些函数设定为异步执行的方式可以隐藏主存和 GPU 存储器之间的数据传输时间，使计算得到优化。

### 3.4.2 共享存储器的访存优化

为了能以高带宽来进行并行访问，共享存储器分为大小相等的 bank 模块。不同

的模块可以同时互不干扰的工作，故对 m 个 bank 可以同时进行访问，这时的访存带宽是只有一个 bank 时的 m 倍。如果多个线程同时请求访问的地址在同一个 bank 里，就产生了冲突。存储器只能串行的完成这些请求。共享存储器会将造成冲突的请求划分成几次来完成，每一次都不产生访存冲突。这个时候存储传输带宽会降低数倍，降低的倍数就是不产生访存冲突的传输次数。bank 的宽度固定为 32bit，每个时钟周期内一个 bank 提供 32bit 的带宽，相邻的 32bit 字存储在相邻的 bank 中[30]。可以通过优化调度访存请求来减少共享存储器的访存冲突[30]。共享存储器的访问模式如图 3.5 所示，给出了无 Bank Conflict 和有 Bank Conflict 访问模式。

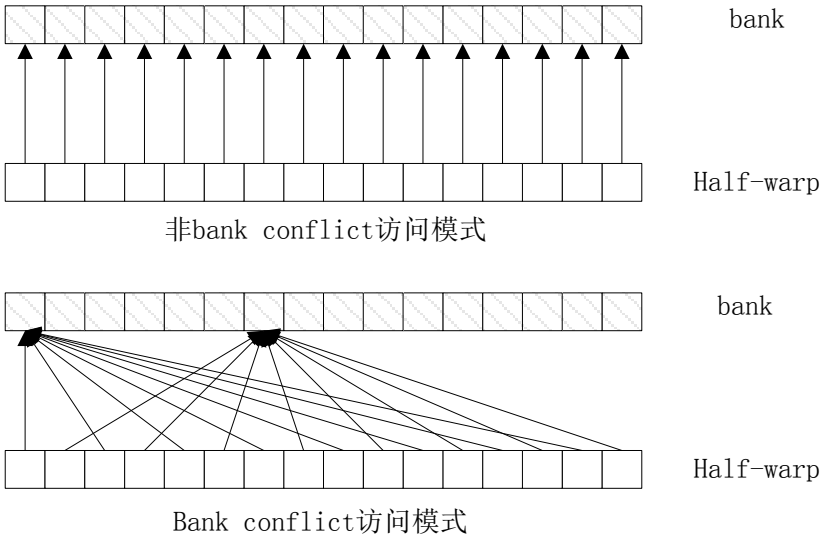


图 3.5 共享存储器的访问模式

为了避免 Bank Conflict，要尽两让同一个 half-warp 中的线程的访存操作不在同一 Bank 内。在基于 CUDA 架构的 GPU 平台上，对于稀疏矩阵的存储格式进行调整，在 kernel 函数的设计以及数据从全局存储器到共享存储器拷贝的过程中确保同一个 half-warp 中的线程不在同一个 bank 中进行操作，避免了 Bank Conflict 现象的出现，提高了运算的性能。

## 3.5 本章小结

本章首先介绍了一些目前基于 GPU 通用计算平台的稀疏矩阵线性方程求解的研究工作，接着论述了两种迭代法求解稀疏矩阵线性方程在 GPU 平台的实现与优化。

GPU 上雅可比迭代法求解稀疏矩阵线性方程时通过采用在 GPU 上分配两个解向量空间交替运算的方法，减少主存和 GPU 之间的数据传输次数并充分发挥 GPU 的

运算能力进行精度判断；从而提高大规模稀疏矩阵线性方程在 GPU 平台求解的计算性能。

GPU 上广义最小残量法求解稀疏矩阵线性方程时最主要的运算是稀疏矩阵向量乘法运算以及向量内积和范数运算。前者在上一章已经实现并优化。本章给出了向量内积和范数运算的 GPU 实现和优化的方法。

最后，对 CUDA 平台上迭代法求解稀疏矩阵方程时的主机设备通信进行优化，并针对共享存储器的访存特点对共享存储器的访存进行优化。

# 4 测试和分析

## 4.1 实验环境和方法

实验的 GPU 服务器的硬件配置为 Inter(R) Core(TM) i7 2.67GHz 的 CPU，DDR3 的 6GB 内存，1TB 的 SATA 硬盘，使用的显卡为 NVIDIA 公司的 GTX 260+专业显卡，通过 PCI-E2.0 和主存相连，GTX260+的设备显存高达 896MB。

软件环境操作系统为 Red Hat Enterprise Linux Server release 5.4 操作系统，使用编程平台为 CUDA2.3 版本和 SDK3.0 版本，编译平台为 NVCC2.0 和 GCC 两个编译器。

测试数据集：本测试选用的均为佛罗里达大学稀疏矩阵选集中的测试矩阵[30]，这些测试矩阵选自非线性优化，晶体自由振动有限元分析，计算机图形学等应用场景，具有不同大小的矩阵规模，因而具有代表性。

求解稀疏矩阵向量乘法功能测试：CPU 计算结果 $V_{CPU}$ ，GPU 计算结果 $V_{GPU}$，绝对误差 $error=||V_{CPU}-V_{GPU}||/||V_{CPU}||$; 其中范式的计算为向量各元素的坐标平方和再开方。通过绝对误差的计算来检验稀疏矩阵向量乘法功能的正确性。

稀疏矩阵方程求解功能测试:在文献[41]中，绝对误差界$||x-x_s||<=||r||/||b||$，其中 x 是采用迭代法求出的解向量，$x_s$ 是准确的方程解向量，$r=b-Ax_s$。通过绝对误差的计算来检验稀疏矩阵方程求解功能的正确性。

CUDA 的内核程序运行时间可以在设备端测量，也可以在主机端测量，无论是哪一种测量方式，最好都测量内核函数多次运行的时间，然后再除以运行次数已获得更加准确的结果。要从主机端准确地测量一个或者一系列 CUDA 调用需要的时间，就要先调用 cudaThreadSynchronize() 函数同步 GPU 线程与 CPU 线程之后，才能完成对时间的测量。cudaThreadSynchronize()函数的功能是阻塞 CPU 线程，直到 cudaThreadSynchronize() 函数之前的所有 CUDA 调用都已经完成。

## 4.2GPU 稀疏矩阵向量乘法性能测试

### 4.2.1 GPU 上分段行合并存储策略的稀疏矩阵向量乘方法性能测试

分别在基于 CSR 存储格式的 CPU 平台，基于 CSR 存储格式的 GPU 平台以及基于 SC-CSR 格式的 GPU 平台上进行稀疏矩阵向量的乘法运算，并记录每一种情况的运行时间。为了使性能比较更具有说服力，其中基于 CSR 存储格式的 GPU 平台上的稀疏矩阵向量乘法运算采用的是 NVIDIA 公司基于 CSR vector kernel 的 SpMV 。三种不同规模的矩阵基于 SC-CSR 的 SpMV 性能对比如图 4.1 所示，把 CPU 运算的运行时间，基于 CSR 存储结构的 SpMV 运算在 GPU 上的运行时间以及采用分段行合并存储策略的稀疏矩阵向量乘运算的运行时间做比较，实验结果表明，CUDA 架构下，相对于 NVIDIA 公司基于 CSR vector kernel 的 SpMV，基于 SC-CSR 的 SpMV 有较好的加速，可见本方法具有显著的性能提升效果。

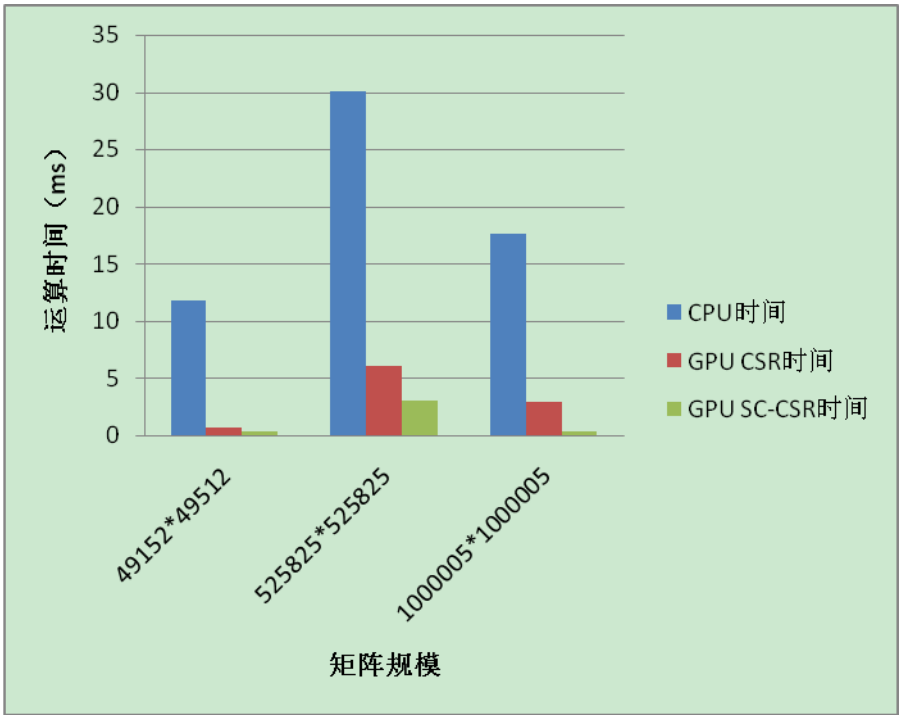


图 4.1 基于 SC-CSR 的 SpMV 性能对比

### 4.2.2 GPU 上按行分块存储策略的稀疏矩阵向量乘方法性能测试

分别在 GPU 平台上对基于 CSR 存储结构和按行分块存储结构的稀疏矩阵向量乘法运算进行测试。基于 CSR 和按行分块存储结构的稀疏矩阵向量乘法性能测试结果如表 4.1 所示，其中 NVGPU 运行时间是采用 NVIDIA 的研究人员的稀疏矩阵向量乘法的 GPU 程序。改进 GPU 时间是采用按行分块的方法计算稀疏矩阵向量乘法运算的执行时间。

表 4.1 基于按行分块存储结构的稀疏矩阵向量乘法性能测试结果

| 矩阵规模 | 非零元素 | NVGPU 时间(us) | 改进 GPU 时间(us) |
|---|---|---|---|
| 4307*4307 | 19422 | 51 | 55 |
| 13965*13965 | 491274 | 180 | 57 |
| 60000*60000 | 410077 | 434 | 78 |
| 102158*102158 | 406858 | 632 | 115 |

基于 VAB 的 SpMV 相对于基于 CSR 的 SpMV 的加速比如图 4.2 所示，可以看出随着矩阵规模的增加，加速比越来越大，性能的提升更加明显。

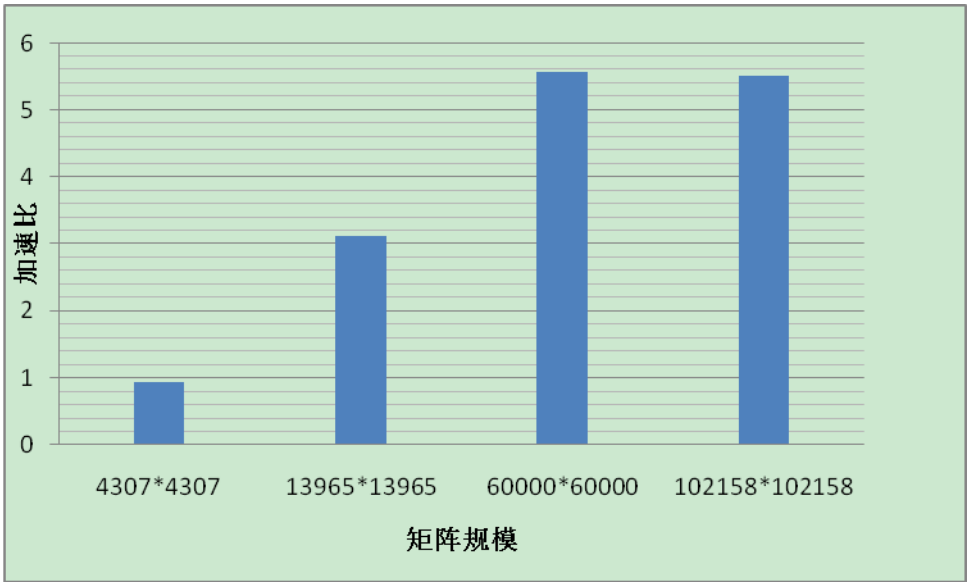


图 4.2 基于 VAB 的 SpMV 的加速比

## 4.3GPU 稀疏矩阵线性方程求解运算性能测试

### 4.3.1 GPU 上雅可比迭代法求解稀疏矩阵线性方程性能测试

对 GPU 上雅可比法求解稀疏矩阵方程的测试，选用了 7 种在工程计算和科学计算常用的 7 个不同规模的矩阵。这些矩阵的规模由 4307*4307 至 1585478*1585478，有着很大的跨度。测试矩阵分别选自非线性优化，晶体自由振动有限元分析，计算机图形学，热能计算，生物学计算，环境科学等工程计算和科学计算的应用场景。这些矩阵具有不同的非零元素个数，GPU 上雅可比法求解稀疏矩阵方程的测试矩阵

如表 4.2 所示。

表 4.2GPU 上雅可比法求解稀疏矩阵方程的测试矩阵

| 矩阵名称 | 矩阵规模 | 非零元素个数 |
|---|---|---|
| ecology2 | 999999*999999 | 4995991 |
| thermal2 | 1228045*1228045 | 8580313 |
| G3_circuit | 1585478*1585478 | 7660826 |
| c-26 | 4307*4307 | 19422 |
| crystk02 | 13965*13965 | 491274 |
| thermomech_TC | 102158*102158 | 406858 |
| Andrews | 60000*60000 | 410077 |

GPU 上雅可比法相对于 CPU 的加速比如图 4.3 所示，文献[33]中不同矩阵的加速比均不超过 20 倍，而本方法的加速比均在 20 倍以上，显然本方法在性能上有了很大的提升。

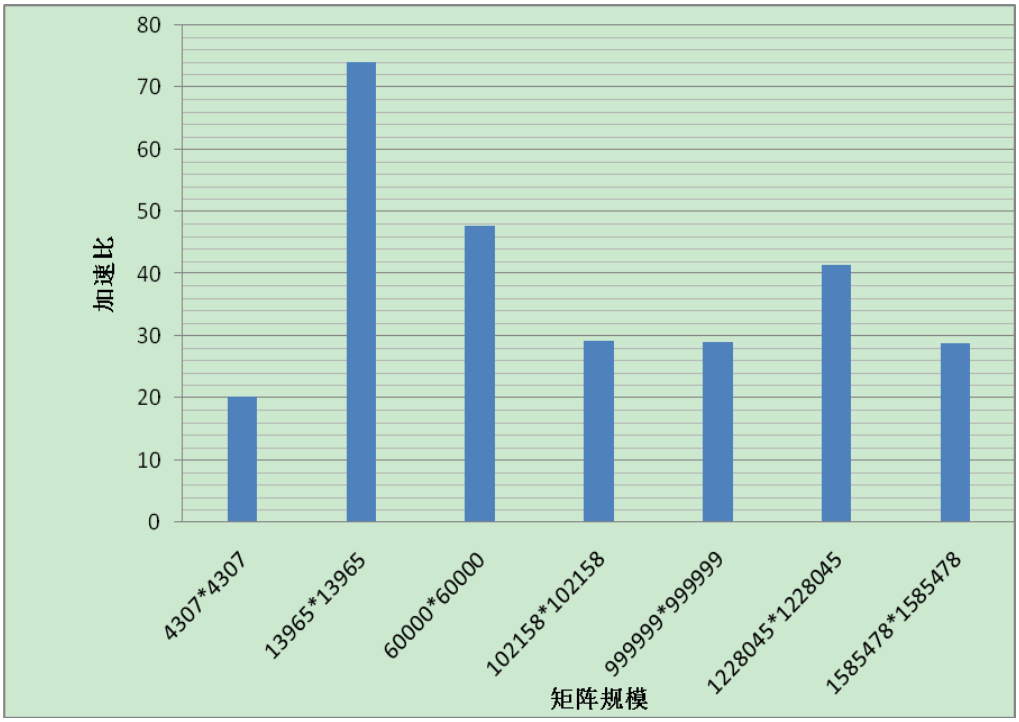


图 4.3 GPU 上雅可比法相对于 CPU 的加速比

### 4.3.2 GPU 上广义最小残量法法求解稀疏矩阵线性方程性能测试

对 GPU 上广义最小残量法求解稀疏矩阵方程的测试，选用了 7 种在工程计算和科学计算常用的 7 个不同规模的矩阵。这些矩阵的规模由 217918*217918 至 1585478*1585478，有着很大的跨度。测试的矩阵具有不同的非零元素个数。非零元

素的个数从 4995991 至 27130349。GPU 上广义最小残量法求解稀疏矩阵方程的测试矩阵如表 4.3 所示。

表 4.3 GPU 上广义最小残量法求解稀疏矩阵方程的测试矩阵

| 矩阵名称 | 矩阵规模 | 非零元素个数 |
|---|---|---|
| af_0_k101 | 503265*503265 | 9027150 |
| af_shell9 | 504855*504855 | 17588845 |
| cage14 | 1505785*1505785 | 27130349 |
| ecology2 | 999999*999999 | 4995991 |
| G3_circuit | 1585478*1585478 | 7660826 |
| pwtk | 217918*217918 | 11524432 |
| thermal2 | 1228045*1228045 | 8580313 |

GPU 上广义最小残量法求解稀疏矩阵方程加速比如图 4.4 所示，其中的加速比是广义最小残量法 GPU 加速后相对于 CPU 串行代码的加速比。可以看到整体加速比比前面的雅可比迭代法求解时候的略低，这是因为广义最小残量法的串行计算比重占整体运算的比例更大。

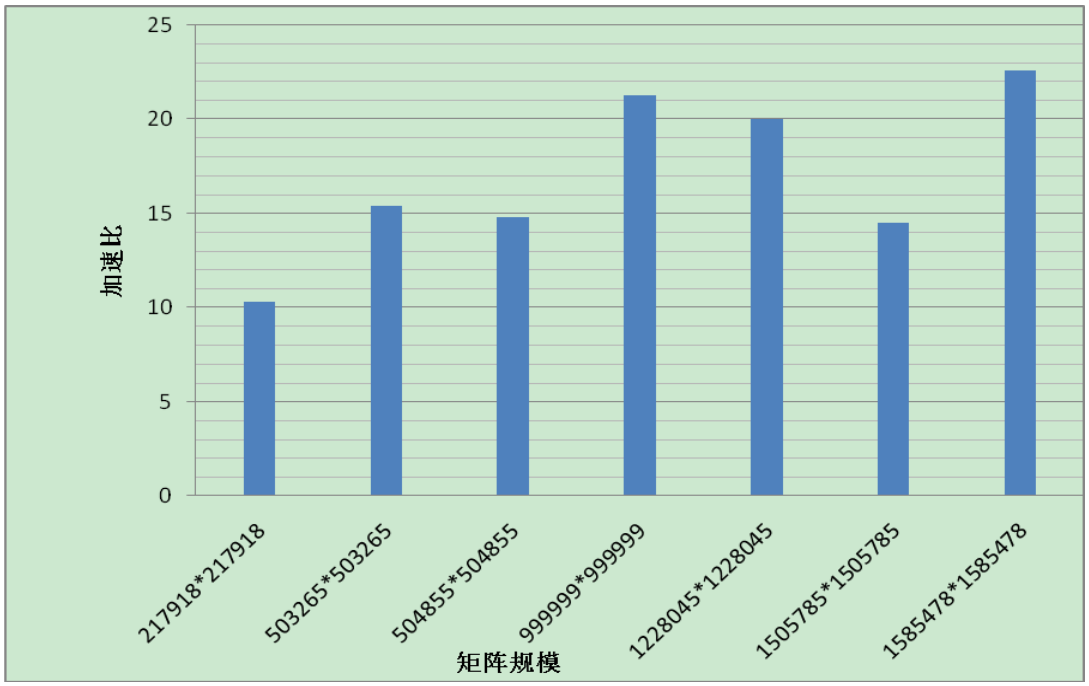


图 4.4 GPU 上广义最小残量法求解稀疏矩阵方程加速比

## 4.4 本章小结

本章首先介绍了系统的实验的硬件配置，软件环境，测试数据集的选取原则，功能测试的方法以及测量程序运行时间的注意事项。然后测试了 GPU 稀疏矩阵向量乘法性能。其中基于 SC-CSR 的 SpMV 最高能达到 8.5 倍的加速，基于 VAB 的 SpMV 能达到 5.5 倍的加速。然后测试了 GPU 稀疏矩阵线性方程求解的性能，雅可比迭代法 GPU 加速后相对于 CPU 的加速比均超过了 20 倍，广义最小残量法 GPU 加速后相对于 CPU 串行代码的加速比也普遍在 10 倍至 20 倍之间。稀疏矩阵的运算总体上都获得了很好的加速效果。

# 5 总结与展望

## 5.1 基于本文的主要工作

稀疏矩阵运算因为在科学和工程计算中的广泛应用已逐渐成为GPU通用计算下一轮的研究热点。稀疏矩阵运算的并行化较稠密线性运算难度更大，是高性能计算领域一直以来都具有挑战性研究课题。

本文主要研究了基于CUDA架构的稀疏矩阵运算在GPU上的实现。以下为本文的主要工作：

1.介绍了目前高性能计算GPU通用计算领域数值计算特别是稀疏线性系统运算的国内外研究情况以及面临的挑战。接着对NVIDIA公司的CUDA架构的GPU通用计算进行分析，分别介绍了GPU的工作模式，GPU的编程模型，GPU的硬件架构和GPU的存储器结构。

2.研究GPU上的稀疏矩阵向量乘法运算实现和优化策略。针对于稀疏矩阵非零元素分布不均造成的空转问题以及同一warp中线程不能合并访存的问题，提出了一种分段行合并存储策略的稀疏矩阵向量乘方法。针对于一个warp内的线程间计算量负载不均衡而造成的线程间等待问题以及因线程不满足对全局存储器的合并访问要求而造成的访存延迟问题，提出了一种按行分块存储策略的稀疏矩阵向量乘方法。

3.研究GPU上的稀疏矩阵线性方程求解运算实现和优化策略。基于GPU的雅可比迭代法求解稀疏矩阵方程时在GPU上分配两个解向量空间，这两个解向量交替参与迭代矩阵的迭代运算，显著减少了主存和GPU存储器之间的数据传输量；并且可以在GPU上直接计算两个解向量的差值来进行精度判断，使计算获得加速。GPU上广义最小残量法求解稀疏矩阵线性方程时给出了向量内积和范数运算的GPU实现方法。这两种优化方法可以推广至所有的GPU下求解稀疏矩阵线性方程的迭代法上，具有普遍意义。

4.针对GPU上稀疏矩阵运算，提出了了全局存储器的访存优化策略，使用纹理存储器和常量存储器对运算进行加速。给出了一种主机设备通信优化方案并针对共享存储器的访存特性进行访存优化。

## 5.2 将来需要做的工作

接下来的工作主要致力于以下方面：

1. 在 GPU 上完善稀疏矩阵的其他数值计算方法，争取为用户提供完整的稀疏矩阵线性运算库。

2.在多 GPU 或者 GPU 集群上实现稀疏矩阵和向量的乘法运算以及稀疏矩阵方程运算，提供更为强大的运算能力。

3.简化 GPU 通用计算编程方法，使 GPU 编程尽可能的贴近 CPU 上的编程。提供自动在 CPU 和 GPU 之间任务自动分配的架构，充分发挥 CPU 和 GPU 的协同计算能力。

# 致 谢

论文的编写已经接近尾声，硕士生活也即将要结束。在此论文即将编写完成之际，衷心的向所有支持过我和帮助过我的亲人、老师、同学和朋友们致以衷心的感谢和诚挚的祝福!

首先要感谢我的导师章勤教授。感谢她给予我学习上的指导和生活上的关怀。章老师平易近人的性格和严谨求实的科研精神让我铭记于心。章老师对学生强烈的责任心让我倍感温暖。在此，向我的导师章勤教授表示由衷的感谢。

感谢课题组的郑然老师长期以来对我的学习、工作和生活所给予的无私关怀、悉心指导。她在学术研究中一丝不苟的严谨作风和乐观向上生活态度会使我终身受益。郑老师敏锐的思维、认真负责的工作态度令我十分敬仰，是我终生学习的楷模。在此，向我的指导老师郑然副教授表示诚挚的敬意。

感谢实验室主任金海教授，金老师严谨的治学态度、渊博的知识、积极的创新精神、开阔的思路、勤勉刻苦的工作作风、与人为善的性格和宽厚待人的胸怀，都让我敬佩不已和受益终生。他对事业的执著追求给我留下了深刻印象，并深深感染和影响了我，这将是我人生中不可多得的精神财富。在此，我谨向金老师表示最真挚的祝福和最崇高的敬意。

感谢韩宗芬老师，吴松老师，刘英书老师，王小兰老师对我生活上的关怀，感谢邵志远老师，胡侃老师对我学业上的指导和帮助。

感谢冯晓文博士，作为师兄，他在学术上给过我提供过很多好的思路，他孜孜不倦的科研精神和积极向上的生活作风一直是我的榜样。感谢已经毕业的李波博士，文石磊师兄，江武师兄，汪聪师兄。感谢同组的周挺，郭明瑞，曾敬翔和黎帅同学两年来在生活中给我的关怀和学业上给我的帮助。

感谢实验室的全体老师同学在 2010 年夏天为我组织的捐款，让我在人生中非常困难的一个阶段感受到了实验室家一般的温暖。在此，衷心的祝愿实验室全体老师同学平安幸福！

最后要特别感谢我的父母和亲人们。感谢父母这些年来对我的养育之恩，感谢我的亲人们对我的支持和鼓励！

# 参考文献

[1] 杨珂. 基于图形处理器的数据管理技术研究： [博士学位论文]. 杭州： 浙江大学图书馆. 2008

[2] B.Nathan, G.Michael. Efficient sparse matrix-vector multiplication on CUDA. NVIDA Technical Report NVR-2008-004. 2008. 1~25

[3] NVIDIA Corp. NVIDIA CUDA Reference Manual, Version 2.3. 2009. 1~372

[4] A.Nukada, S.Matsuoka. Auto-Tuning 3-D FFT Library for CUDA GPUs. SC'09. 2009. 5~12

[5] V.Volkov, W. Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. SC'08. 2008. 13~18

[6] N.Galoppo, K.Govindaraju, M.Henson et al. LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics. SC'05. 2005. 2~5

[7] A.Kerr, D.Campbell, M.Richards. QR Decomposition on GPUs. GPGPU'09. 2009. 71~78

[8] X.Cui, C.Yifeng, H.Mei. Improving Performance of Matrix Multiplication and FFT on GPU. ICPADS'09. 2009. 42~48

[9] C.Yifeng, X.Cui, H.Mei. Large-Scale FFT on GPU Clusters. ICS'10. 2010. 315~324

[10] M.M. BASKARAN, R.WEKAR. Optimizing Sparse MatrixVector Multiplication on GPUs. IBM Research Reprot RC24704 . 2009. 23~34

[11] URI:http://www.mendeley.com/research/the-sparse-matrix-vector-product-on-gpus/.

[12] H.Courtecuisse, J.Allard. Parallel Dense Gauss-Seidel Algorithm on Many-Core Processors. HPCC09. 2009. 139~147

[13] URL: http://bebop.cs.berkeley.edu/pubs/ perftune-2003-01-24.pdf.

[14] A.Monakov, A.Avetisyan. Implementing Blocked Sparse Matrix Vector Multiplication on NVIDIA GPUs. Heidelberg. 2009. 56(57): 289~297.

[15] J.Willcock, A.Lumsdaine. Accelerating Sparse Matrix Computations via Data Compression. ICS'06. 2006. 307~316

[16] M.Belgin, G.Back, J. Ribbens. Pattern based Sparse Matrix Representation for Memory-Efficient SMVM Kernels. ICS'09. 2009. 100~109

[17] A.Buluc, T. Fineman. Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks. SPAA'09. 2009. 233~244.

[18] J. C.Bik, A.G.Wijshoff. Automatic data structure selection and transformation for sparse matrix computations. IEEE Transactions on Parallel and Distributed Systems. 1996. 109~126

[19] S.Williams, L.Oliker. Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms. SC'07. 2007. 23~45

[20] M.Garland. Sparse Matrix Computations on Manycore GPU's. DAC'08. 2008. 2~6

[21] N.Bell. Implementing Sparse Matrix-Vector Multiplication implementation on Throughput-Oriented Processors. SC'09. 2009. 32~35

[22] 白洪涛. 基于 GPU 的高性能并行算法研究：[博士学位论文]. 吉林：吉林大学图书馆. 2010

[23] X.Liu, T.Gu, X.Hang et al. A parallel version of QMRCGSTAB method for large linear systems in distributed parallel environments. Applied Mathematics and Computation. 2006, 172(2):744~752

[24] 林成森. 数值分析. 北京：科学出版社. 2006. 90~107

[25] 葛振, 杨灿群, 吴强等. 线性系统求解中迭代算法的 GPU 加速方法. 计算机工程与科学. 2009，31 (A1)：179~182

[26] W.Zhuowei, X.Xianbin. Optimizing Sparse Matrix-Vector Multiplication on CUDA. 2010 2nd International Conference on Education Technology and Computer (ICETC). 2010. 109~113

[27] 白洪涛，欧阳丹彤，李熙铭等. 基于 GPU 的稀疏矩阵向量乘优化. 计算机科学. 2010，37(8)：168~171

[28] 马超，韦刚，裴颂文等. GPU 上稀疏矩阵与矢量乘积运算的一种改进. 计算机系统应用. 2010，19(5)：116~120

[29] 张舒，褚艳利，赵开勇等. GPU 高性能计算之 CUDA. 北京：中国水利水电出版社. 2009. 155~163

[30] D.B. Lloyd, C.Boyd, N.Govindaraju. Fast computation of general fourier transforms on GPUs[C]. IEEE International Conference on Multimedia and Expo. ICME. 2008. 5~8

[31] NVIDIA Corporation. NVIDIA CUDA C Programming Best Practices Guide CUDA Toolkit 2.3. 2009. 32~34

[32] NVIDIA Corporation, NVIDIA CUDA Programming Guide version 2.3. 2009. 43~46

[33] 唐滔，林一松. Jacobi 和 Laplace 算法在 GPU 平台上的设计与实现. 计算机工程与科学. 2009，31 (A1) ：93~96

[34] 张健. 方程组的迭代法求解在GPU 上的实现. 电子器件. 2010，33(6)：767~771

[35] 张健，涂永明，涂晓明. 雅可比迭代法在图形处理器上实现的研究. 计算机工程与应用. 2009，45(34)：53~55

[36] 刘华磊. 解线性方程组的简单 GMRES 算法研究：[硕士学位论文]. 南京：南京航空航天大学图书馆. 2007

[37] 蔡大用，白峰杉. 高等数值分析. 北京：清华大学出版社. 1997. 71~93

[38] Y. Saad, M.H. Schultz. GMRES: A generalized minimal residual algorithm for solving non symmetric linear systems. SIAM Journal on Scientific and Statistical Computing. 1986, 7(3) : 856~869

[39] M. Habu, T. Nodera, GMRE algorithm with changing the restart cycle adaptively. Proceedings of ALORITMY ‘00 Conference on Scientific Computing. 2000. 254~263

[40] URL: http://www.cise.urf.edulresearch/sparse/matrices/

[41] 曹玉平. 略论线性方程组解的误差估计. 甘肃联合大学学报(自然科学版). 2010，24(3)：26~29