分类号 TP391 学号 14066010

UDC 密级 公开

工学博士学位论文

面向 GPU 异构系统的测评与应用研究

博士生姓名 方民权

学科专业 计算机科学与技术

研究方向 高性能计算

指导教师 张卫民 研究员

国防科学技术大学研究生院

二〇一七年四月

Performance Evaluations and Applications on GPU Systems

Candidate：Fang Minquan

Supervisor：Prof. Zhang Weimin

A dissertation

Submitted in partial fulfillment of the requirements

for the degree of Doctor of Engineering

in Computer Science and Technology

Graduate School of National University of Defense Technology

Changsha，Hunan，P.R.China

April，2017

独 创 性 声 明

本人声明所呈交的学位论文是我本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，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表和撰写过的研究成果，也不包含为获得国防科学技术大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文题目：面向GPU异构系统的测评与应用研究

学位论文作者签名：方民权 日期：2017 年 4 月 5 日

学位论文版权使用授权书

本人完全了解国防科学技术大学有关保留、使用学位论文的规定。本人授权国防科学技术大学可以保留并向国家有关部门或机构送交论文的复印件和电子文档，允许论文被查阅和借阅；可以将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、汇编学位论文。

（保密学位论文在解密后适用本授权书。）

学位论文题目：面向GPU异构系统的测评与应用研究

学位论文作者签名：方民权 日期：2017 年 4 月 5 日

作者指导教师签名： 日期：2017 年 4 月 14 日

目 录

表 目 录

图 目 录

摘 要

由于功耗和散热的限制，提升主频来增加芯片性能的道路已到尽头，多核与众核已成为处理器发展的新方向。众核协处理器因其固有的高性能、低功耗和高性价比等优势，在高性能计算领域表现优异；但也面临异构程序移植困难、众核体系结构性能优化困难、异构系统高效协同困难、数学原理/领域机理理解困难等挑战。本文重点关注面向众核体系结构的性能优化和异构系统协同优化，试图通过微 benchmark 测评以获知感兴趣的体系结构特性，导出相应的优化策略，并将其应用到实际应用获得性能收益。

本文以众核 GPU 为例，重点针对众核体系结构性能优化困难和异构系统高效协同困难两方面的挑战开展研究，关注 GPU 的存储体系和主机端存储选择，设计微 benchmark 探索相关特性与规律，总结优化策略，并将其应用到高光谱影像降维和声呐信号波束形成两类实际应用中，以最大化应用执行性能。本文主要工作和创新包括以下几点：

（1）针对 GPU 存储优化难题，提出了一种面向 GPU 存储系统的 warp 级基准测评方法，设计了 GPU 的访存优化方法，构建了 GPU 的一种访存优化框架，实验结果表明了该优化框架的有效性。面向 GPU 存储层次，测评了各存储单元的 thread 级访存延迟；提出基于 GPU 存储的 warp 级访存测评方法，设计了两个并行访存测评实验，系统地测评了共享存储、常量存储、全局存储和纹理存储的访存特性；探索寄存器替换局部存储的策略、共享存储体冲突及其避免策略、数据类型选用对全局存储访存带宽的影响；设计各存储单元的访存优化策略，构建 GPU 访存优化框架，并阐述其在高光谱影像降维和声呐信号波束形成两类真实应用中的使用和收益，验证了优化框架的实用性和有效性。

（2）面向异构协同优化问题，提出了一种 CPU/GPU 异构系统的分段主机端存储选择模型，利用 zerocopy 技术设计了两种优化方案，并验证了所提模型和方案的有效性。在 CPU/GPU 异构系统中，通过微 benchmark 测评主机端存储的访存带宽、PCI-E 带宽、页锁定存储的注册和解除注册开销，提出分段式的主机端存储选择模型，并以 PCA 降维为例探讨了模型的使用和效果。此外，针对 zerocopy 技术，提出并验证了利用 zerocopy 减少全局存储访问和实现计算通信重叠两项优化方案；实例讨论了计算与通信重叠、（CPU）计算与（GPU）计算重叠两类通用的异构协同优化技术。

（3）将 GPU 存储和异构系统的测评与优化研究成果应用于高光谱影像降维领域，构建了面向众核体系结构的高光谱影像并行降维框架，提出了 PCA、FastICA 和 MNF 三类高光谱影像降维方法的 GPU 并行算法，实验结果表明算法具有良好

的加速效果。针对 3 类主流的降维算法（主成分分析、快速独立成分分析和最大噪声分数变换），分析加速热点，分别基于分布存储、共享存储和 GPU 设计了协方差矩阵计算、PCA 变换、ICA 迭代、噪声估计（滤波）等热点并行方案；面向众核 GPU，研究各热点的性能优化策略及优化效果；提出面向众核体系结构的高光谱影像并行降维框架，并在 CPUs、GPUs 和 Phis 三种平台给予实现。实验结果显示本文并行和优化方案能够显著提升并行降维算法的性能，其中 Gs-PCA 算法最高加速 119.7 倍，Gs-FastICA 算法最高加速 106.6 倍，G-MNF 算法最高加速 86.9 倍；并通过实验分析并行降维算法的可扩展性。

（4）将 GPU 存储和异构系统的测评和优化研究成果应用于声呐信号波束形成领域，提出了 DFT-CBF 和 MVDR 两类宽带波束形成方法的 GPU 并行算法，实验结果显示算法获得了理想的加速效果。重点针对 DFT-CBF 算法中的 DFT 变换、CBF/Lofar 计算和频带能量整合统计，以及 MVDR 算法中的 DFT 变换、双边雅克比迭代（厄尔米特矩阵特征分解）和方位谱统计等加速热点，设计 GPU 并行映射方案，面向 GPU 体系结构探索性能优化策略，量化分析优化效果，实现基于 GPU 的并行宽带波束形成算法。通过实验分析了并行算法加速比和实时性，其中基于 GPU 的 DFT-CBF 算法可实时处理万基元基阵波束形成，最高加速 125.3 倍；同时使用多个 GPU 同时运算的 Gs-MVDR 算法获得最高 30.7 倍加速比。

关键词：GPU；性能优化；warp 级访存测评；主机端存储选择；零拷贝；高光谱影像降维；波束形成

Abstract

Due to the limitations of power consumption and heat dissipation, it is impractical to futher enhance the performance of the chip by increasing frequency. And multi-core and many-core have become the mainstream of building a new processor. Because of its inherent high performance, low power consumption and high performance-price ratio, the many-core processor plays an outstanding role in high performance computing. But at the same time, it is difflicut to port heterogeneous programs, to optimize them on architecture, to use both the host processors and the co-processors, and to understand the mathematical principles and mechanisms. In this paper, we focus on performance optimization of many-core architectures. We derive the corresponding optimization strategies by microbenchmarking the interesting components of many-core architectures, which are applied in two real world applications.

In this paper, we take GPU as an example, and focus on addressing the challenges of performance optimization on many-core architecture and in heterogeneous system. We design microbenchmarks to evaluate the components of the GPU memory system and the interconnecting components, derive the optimization strategies, and then apply them to the real world applications of hyperspectral image dimensionality reduction and sonar signal beamforming to get the maximum performance. Our major contributions include the following:

(1) Focus on the optimization challenge on GPU memory, we propose a GPU memory optimization framework with microbenchmark, and then apply it on two real world applications to obtain excellent performance. Comparing to benchmarking the thread level latencies on different memory hierarchies, we propose a warp-level benchmark method, and design two experiments on parallel memory accessing. Such warp-level measurements are done on shared memory, constant memory, global memory and texture memory. Further we explore the strategies of replacing local memory by registers, avoiding bank conflicts of shared memory, and improving global memory bandwidth with different data types. By summarizing the optimization guidelines for different memory hierarchies, we construct a GPU memory optimization framework, and apply it to hyperspectral image dimensionality reduction and sonar signal beamforming. The results show the practicality and effectiveness of our framework.

(2) Focus on the optimization challenge on CPU/GPU heterogeneous system, we explore the host memory, zerocopy, overlaping calculation and communication, overlaping calculation and calculation, and verify them by case studies. By measuring the memory bandwidth of host memory, the PCI-E bandwidth, the register and

unregister overheads, we propose a piece-wise model for host memory allocation, and demonstrate the model usages and performance effects with a case study of PCA dimensionality reduction. Moreover, we propose and verify two optimization of reducing global memory access and voerlaping calculation and communication with zerocopy, and discuss the coordinated optimization techniques of overlaping calculation and communication, overlaping calculation and calculation by samples study.

(3) Based on the optimization studies on GPU memory and heterogeneous system, we parallelize and optimize hyperspectral image dimensionality reduction. Fousing on three typical algorithms of principal component analysis (PCA), fast independent component analysis (FastICA) and maximum noise fraction (MNF) rotation, we identify the hotspots, and design the parallel schemes on distributed storage, shared memory and GPU for the hotspots of the covariance matrix calculation, PCA transformation, ICA iteration and filtering. Then, we investigate the optimizations and their effects of different hotspots on GPU system. Finally, we propose an efficient and portable parallel framework for hyperspectral image dimensionality reduction on multi-/many-cores platforms, and implement on CPUs, GPUs and Xeon Phis. The experimental results show that our parallel framework can get excellent performance. Gs-PCA, Gs-FastICA and G-MNF can abtain a speedup of upto 119.7X, 106.6X and 86.9X, respectively. And we discuss the scalability of parallel dimension reduction algorithms.

(4) We accelerate the broadband sonar signal beamforming algorithms of DFT-CBF and MVDR with the previous optimization studies on GPU memory and heterogeneous system. We focus on the DFT-CBF hotspots of the DFT, the CBF/Lofar calculation and the band energy statistics, and the MVDR hotspots of the DFT, bilateral Jacobi iterative (hermition matrix decomposition) and the azimuth spectrum statistics. We design GPU mapping schemes, develop optimizations on the GPU system and evaluate the optimization effects, and implement the parallel beamforming algorithms on GPU. We discuss the speedups and real-timeness of parallel beamforming algorithms with experiments. Our parallel DFT-CBF algorithm on GPU can process more than ten thousands elements beamforming in real time, and it can obtain a maximum speedup of 125.3 X. While, the Gs-MVDR algorithm with multiple GPUs can get a best speedup of 30.7 X.

Key words：GPU， performance optimization，warp-level benchmarking，host memory selection，hyperspectral image dimensionality reduction，beamforming

第一章　绪论

1.1　研究背景

1.1.1　多核处理器与众核协处理器的起源与发展

2001 年，IBM 公司的 POWER4 处理器开创了多核 CPU 的历史先河[1]。在 2004 年以前，传统微处理器技术仍然沿片上高性能单处理器的道路发展，提升主频和增加晶体管数量是提高处理器性能的主要手段。由于受功耗和散热问题的影响，继续提升主频将导致功耗和发热量的急剧增长，传统处理器的发展道路已到尽头。此后，处理器技术开始往片上多核、提升向量化位宽的方向发展，从而继续保持了处理器性能的摩尔定律增长速率[2]。

最早的协处理器是 Intel 公司推出的 8087 协处理器[3]，目的是弥补 8086 处理器浮点计算能力差的弱点。其后由于工艺水平的提升，从 486DX 开始，该种协处理器被集成到了 CPU 中，协处理器的概念也随之消失。由于主频提升的末路、功耗和散热的限制及对计算需求的不断增长，协处理器又重新登上历史舞台[4]，例如 GPU（graph processing unit）从专业图形图像处理到科学计算的转型[7]；又如 2012 年，Intel 推出集成众核（many integrated core，MIC）协处理器产品 Xeon Phi[5]。GPU 和 Phi 协处理器已成为高性能计算领域的新宠，2010 年搭载 CPU/GPU 异构系统的天河 1A 超级计算机首次夺得 TOP 500 榜首；2012 年同样搭载 CPU/GPU 异构系统的泰坦超级计算机成为 TOP 500 第一；2013 年 6 月，基于 CPU/Phi 异构系统的天河 2 号超级计算机夺魁 TOP 500 超级计算机榜单，并连续蝉联 6 届[6]。

1999 年 8 月，NVIDIA 公司发布首款 GPU 产品，此后很长的一段时间内，GPU 只用于图形处理，其他领域科学计算问题需要表达为着色器语言才能使用 GPU 计算。2006 年发布支持 CUDA（compute unified device architecture）编程的 GeForce 8 系列 GPU，使得通用计算到 GPU 的移植变得相对简单方便，GPU 开始涉足高性能计算领域[7]。随后 GPU 计算架构得到了跨越式发展，短短十年间，经历了 Tesla、Fermi、Kepler、Maxwell 和 Pascal 五代架构[8-9]，GPU 的浮点计算能力正以超越摩尔定律的速度快速发展。

1.1.2　国内外高性能计算的发展现状

伴随着计算机的产生、发展及应用，高性能计算经历了“计算物理”、“科学计算”、“高性能计算”、“战略计算”等几个阶段，正在朝着“E 级计算”和“量子计算”方向发展。最早的计算机出现是为了计算导弹轨迹，在曼哈顿计划[10]

中用于处理原子弹设计中大量的数值计算，曼哈顿计划解密后出版的“计算物理丛书”中首次出现“计算物理[11]”一词。1983 年，以数学家拉克斯为首的不同学科的专家在向美国政府提出的报告中，强调“科学计算是关系到国家安全、经济发展和科技进步的关键性环节，是事关国家命脉的大事”[126]。1991 年美国提出“高性能计算与通信（HPCC）计划”[12]。1995 年，美国宣布全面禁止核武器条约，为确保核库存的性能、安全性、可靠性和更新需要而实施的“加速战略计算创新（ASCI）计划”中首次出现“战略计算”一词，模拟核试验开始替代真实核试验[13]。2002 年，美国国防部高级研究计划局（DARPA）提出 HPCS 计划[14]，旨在弥补 HPC 计算与未来量子计算之间的空隙。2008 年，美国 DARPA 信息处理技术办公室（IPTO）公布了 E 级计算研究报告，E 级计划将为大气科学、高能物理、核物理、计算化学及生物学等众多领域提供支撑[15]。2015 年，美国发布了新的国家战略计划规划，旨在攻克传统超算难题的同时融合大数据的处理。

伴随上述政策的发布，成果是一台又一台跨量级超级计算机的诞生。1964 年第一台超级计算机 CDC6600 诞生[16]；1985 年 Cray 公司研制首台 Gflops 级运算性能的 Cray-2 超级计算机[17]；1996 年，Intel 公司发布运算性能为 1.453 Teraflop/s 的 ASCI-RED 超级计算机[18]；2008 年 IBM 成功研制“走鹃（RoadRunner）”宣告超级计算机进入 PetaScale 时代[19]；此前蝉联 6 届 TOP500 榜首的天河 2 号超级计算机理论峰值性能高达 54.9Petaflop/s[6]。短短 50 年，超级计算机的计算能力已经提升了 10 个数量级。

天河 1A、天河 2 号、神威太湖之光等超级计算机相继成为 TOP500 榜首[6]，标志着中国目前已进入世界高性能计算机系统研制的前列。目前越来越多的地方政府投入巨资建设超级计算中心。总体而言，我国已在硬件方面取得了长足的发展。然而，距离高性能计算可持续发展仍任重而道远，一个突出问题是对软件的重视不够，直接导致了目前“重硬轻软”的局面。

天河 1A 和天河 2 号都是搭载的多核处理器与众核协处理器异构系统（CPU+GPU、CPU+Phi），其运算性能主要来源于众核协处理器 GPU 和 Phi。但由于异构系统可编程性差、可移植性差、性能优化困难等原因，软件移植需要异常繁重的代码改造以适应全新的众核协处理器。这些代码改造工作包括代码重新编写、近乎无限迭代的性能优化、甚至可能需要对原始 CPU 代码进行重构。这些使得很多用户仅在异构系统上执行 CPU 程序，而忽略使用运算性能更高的众核协处理器，导致异构系统的整机利用率明显偏低。

综上所述，如何有效发挥现有超级计算机众核协处理器资源，进行有效的代码移植和性能优化工作，是当前高性能计算领域的重大问题。由于 Xeon CPU、GPU、Xeon Phi 等计算核心部件均为商用产品，无法获知真正的技术细节，尽管

官方文档给出了很多优化策略，但这种无根源依据的优化策略难以真正在实际应用优化中采纳，也很难真正地基于体系结构深入优化实际应用程序性能。很多并行的应用开发仅仅局限于并行化，或者仅仅是“凑巧”试中了部分优化策略，而未专门进行面向体系结构的性能优化。

本文以 GPU 为例，期望通过 microbenchmark 测评来获知 GPU 体系结构的感兴趣特性，并设计相应的优化策略；再将这些优化策略用于多类科学计算应用的并行和优化研究中，以高效提升实际应用的运算性能。由于优化策略是通过测评结果设计的，对于其原理和应用场景理解更加透彻，能够有效助力高性能计算实际应用程序的开发和优化，最终为扭转高性能计算领域“重硬轻软”的不平衡做出贡献。

1.2 众核体系结构与异构系统

1.2.1 众核体系结构的优势

多核/众核异构系统由传统多核 CPU 和众核协处理器组成。由于多核 CPU 和众核协处理器（GPU 或 Phi）间明确的协作关系，众核协处理器的体系结构设计相对 CPU 有着显著的差异和天然的优势。

（1）超高性价比：在异构系统中，多核 CPU 在提供 256 位宽向量处理的基础上，主要负责复杂的逻辑处理，而众核协处理器的设计目的是超高的浮点计算能力，特别是双精度浮点计算能力。由于多核 CPU 和众核协处理器设计理念的不同，导致其内部结构、元器件成本上的重要区别。众核协处理器在设计时避免或减弱了类似分支处理、逻辑控制等与浮点计算无关的操作功能，专注于浮点运算，因此在制造成本上有着巨大的优势。比如众核 GPU 仅需十分之一的成本即可达到与多核 CPU 同等的浮点运算能力。

（2）绿色能耗比：众核协处理器集成了大量轻量级微处理器单元，这些处理单元功能简单（仅做浮点运算）、时钟频率有限，使得运算产生的功耗较小。其中 GPU 相对 Phi 在能耗比上更具优势，比如 NVIDIA Tesla K20c GPU 在休眠状态仅需 15W 左右功耗，满载运转时功耗约为 150W，能提供超过 2 TFLOPS 单精度浮点运算能力；而 Intel Xeon Phi 31S1P 协处理器在低功耗状态的能耗为 100W，在满载运行提供 2 TFLOPS 单精度浮点运算能力时需要 250W 供能。2015 年 11 月发布的 Green500 榜单前 10 名中，有 9 台超级计算机使用了 CPU/GPU 异构系统，除了第 3 位使用了 AMD FirePro GPU，第 2,4-10 位均使用了 NVIDIA Tesla GPU[20]。

（3）浮点运算能力强：众核协处理器的设计目标就是超高的浮点运算性能，因而众核协处理器相对多核 CPU 在体系结构设计上有天然的区别。比如集成众核

MIC 提供了 512 位宽的向量处理单元（vector processing unit，VPU），而 CPU 仅提供了 256 位宽向量处理单元；又比如，GPU 中一个 warp 的 32 个线程由 1 个指令分发单元控制，处理同一条指令。这些设计都为众核协处理器超强的浮点运算能力提供了保障。由于众核协处理器强大的浮点运算性能，这类多核/众核异构系统广泛应用于超级计算机领域，2010 年，基于 CPU/GPU 异构系统的天河 1A 超级计算机夺得 TOP500 桂冠，2012 年同样是 CPU/GPU 异构系统的泰坦超级计算机在 TOP500 夺魁[6]。2013 年，基于 CPU/MIC 异构系统的天河 2 号超级计算机蝉联了 6 届 TOP500 榜首[6]。

1.2.2 多核/众核异构系统面临的挑战

多核/众核异构系统浮点运算能力强，但异构并行编程非常困难，主要表现在传统的并行程序无法直接移植到异构平台获得高运算性能。由于众核体系结构的设计理念有别于传统的多核 CPU，编程时一方面要用异构并行编程模型重新编写程序，另一方面针对不同的众核体系结构和不同的应用特征设计相应的优化策略，从而充分发挥出众核协处理器的性能。这给多核/众核异构系统的软件开发提出了巨大的挑战，具体表现在以下几个方面：

（1）异构程序移植困难（可编程性差）。众核协处理器与主流 CPU 存在巨大差异，比如体系结构的差异、独立的存储、并行编程模型不兼容等。这些差异导致现有程序无法直接在异构平台上执行，即便能执行（Xeon Phi 与 Xeon CPU 都支持 x86 架构，可将 CPU 程序编译为 native 版本，在 Phi 上执行）也很难获得理想性能。异构系统上的程序开发首先要学习相应的异构并行编程模型，清晰把握宿主机和众核协处理器见的数据交换，还要熟练掌握并行程序调试技术。特别是针对大型的科学计算应用的遗留代码，理清程序结构、熟悉程序中的数据结构、定位移植热点，确保修改时数据结构与代码的全局统一等，这些工作复杂且困难。综上所述，异构系统并行程序的移植非常困难。

（2）众核体系结构性能优化困难。众核协处理器拥有区别于主流 CPU 处理器的特性，比如计算核心众多但单核性能较弱，GPU 线程以 warp 为单位同步执行指令，GPU 中种类丰富的层次式存储体系，Phi 的环状互连（KNL 架构中改为 2D 网格互连）结构，Phi 提供 512 位宽的向量处理单元等。这些特征都将直接影响异构并行程序运算性能，必须在编程时充分考虑到这些体系结构特性，才能优化出高性能的程序。

（3）异构系统高效协同困难。在 CPU/GPU 异构系统、CPU/MIC 异构系统中，仅 CPU 运算或仅 GPU 或 Phi 运算都无法充分发挥多核与众核异构系统性能，在设计异构并行程序时必须充分考虑到异构系统的所有计算资源。异构系统中的协同

优化主要包括计算（CPU）与计算（GPU、MIC）重叠和计算与通信重叠；此外对于 CPU/GPU 异构系统，还需要考虑 zerocopy 优化、主机端存储类型选择等。要设计出优秀的协同优化方案，必须充分熟悉算法流程，寻找可同时计算/通信的步骤；或将大任务有机分割，合理给各计算部件分配子任务；要考虑 PCI-E 带宽和各部件运算性能，最大化地掩盖计算时间或通信时间。寻找最优协同方案的过程是复杂多样和循环迭代的，很难直接得到一个最佳协同优化方案。

（4）数学原理/领域机理理解困难。除了 HPL（high performance linpack）这类基准测试集，高性能计算不会单独存在，必然与应用领域相结合而共存，比如天气预报、计算物理、计算化学、天体物理、量子力学、核模拟等。天气预报需要求解空气动力方程组，量子力学需要处理薛定谔方程，这些物理和数学原理难以理解（特别是对于计算机专业学生而言），求解过程也无比复杂，且书籍内容仅仅是原理的简单抽象，真正在程序开发时应用了大量复杂的近似计算，又没有系统的书面教程供开发人员参考。要充分开发某一算法及其程序的最佳性能，必须首先理解数学原理或领域机理，从根本出发设计出优秀的并行算法，再辅以编程实现和性能优化才能真正发挥异构系统的性能。

针对上述 4 个方面的挑战，异构程序移植困难的问题已有很多研究，提出了如 OpenCL、OpenACC 等通用的异构编程模型。而本文的关注重点是众核体系结构性能优化困难和异构系统高效协同困难这两方面的挑战：第 3 章针对众核体系结构性能优化困难的挑战，通过 microbenchmark 测评获知 GPU 的体系结构特性，从而设计相应的优化策略，并在后文两类实际应用的开发进行运用获得收益；第 4 章针对异构系统高效协同困难的挑战，根据 GPU 特殊的主机端存储类型，提出了主机端存储选择模型，并探索了 zerocopy 等异构协同优化技术，相关研究成果应用到实际的应用开发中。而数学原理/领域机理理解困难的挑战则需要多花时间、多理解，没有捷径。

1.2.3 异构系统的相关研究

本节主要从多核 CPU、众核 GPU 和众核 Phi 三个方面的性能优化研究展开叙述相关工作。此外在后文每一章节均会描述与本章节主题相关的相关工作。

多核 CPU 上有大量的应用开发和性能优化研究。Li 等人[21]基于片上集成 CPU 与 GPU 的 Sandy Bridge 处理器加速 SURF 褶皱人脸识别。Treibig 等人[22]在 4 代 Intel 多核处理器（Harpertown、Westmere、Westmere EX 和 Sandy Bridge）上分析医学图像重建的性能，采用最新 SIMD 指令集 SSE 和 AVX 在底层优化背投（backprojection）算法。Cui 等人[23]基于多核 CPU 计算平台开发了针对蒙特卡罗模拟（MCS）类型应用的通用高性能计算框架，经性能优化与多级并行，MCS 求

解器能达到 CPU 理论峰值性能的 50%。

基于众核 GPU 平台，已有大量性能优化相关研究。Wende 等人[24]针对 GPU 上 kernel 函数并发执行机制，多线程访问 GPU 可能导致性能下降的问题，提出“生产者-消费者理论”方法来管理 kernel。Xiao 等人[25]提出将多个物理 GPU 抽象成一个统一的虚拟 OpenCL 框架（VOCL），支持本地与远程 GPU 的透明使用，通过使用面向数据通信、kernel 发射等优化，可获得 85%的数据写带宽和 90%的数据读带宽。Hoshino 等人[26]利用微测试集和真实 CFD 应用测评 CUDA 和 OpenACC 性能，结果显示 OpenACC 性能普遍低于 CUDA 近 50%，由于缺乏共享存储编程接口，某些 OpenACC 应用损失将近 3 倍性能。Lee 等人[27]提出一个基于 OpenMP 的复杂 CUDA 编程模型抽象的编程接口 OpenMPC，并提供了相关参数与优化的高级控制，开发了一个支撑 OpenMPC 的完全自动编译和用户辅助调试系统，系统能获得手写 CUDA 代码 88%的性能。Dolbeau 等人[28]针对 GPU 程序开发困难，代码重写繁重等问题，提出 HMPP（异构多核并行编程环境），允许异构硬件加速器无缝集成，同时保留遗留代码。Pai 等人[29]开发了软件一致性机制，实现 CPU 与 GPU 间的自动数据传输，该机制使用编译器分析确定潜在的存储访问，必要时使用运行时传输，将自动存储管理器集成到 X10 编译器与运行时，不仅使编程更容易，还减少了冗余存储传输。Yang 等人[30]提出一种 GPGPU 新型优化编译器，输入功能正确且性能未经优化的 kernel 函数，编译器通过代码分析，确定存储访问模式，生成优化的 kernel 及其调用参数，实验测试表明该编译器能达到与 CUBLAS 库 2.2 版本相匹配的性能。Lee 等人[31]提出从 OpenMP 到 CUDA 代码的自动源到源转换的编译器框架，目的是进一步提高可编程性，使现有 OpenMP 应用能在 GPU 上执行，论文定义了几个关键转换技术，使其能有效访问 GPU 存储以获得高性能。

在 Intel 的集成众核协处理器 Xeon Phi 上，亦有很多性能优化的相关研究。Potluri 等人[32]针对 MIC 节点内通信问题，利用共享存储、Infiniband Verbs 和 SCIF 设计了能减少通信延迟近 70%（4MB 信息）的混合解决方案，并在 MVAPICH2 MPI 库中实现。Si 等人[33]在 Phi 集群上实现 DCFA-MPI，提供了 Phi 协处理器间的直接数据通信功能，对比原来的通信方式取得可观的加速效果。Newburn 等人[34]讨论了 offload 软件框架和编译器设计，枚举异构计算关键性能特征，包括初始化、数据移动与访存等，旨在解决 offload 能否获利、如何规定及其条件的问题，为第三方开发者提供指南。Noack 等人[35]提出 HAM（异构活动消息）层来最小化 offload 负载，HAM 提供了类似 LEO 的 offload 基本 API 接口，相对 LEO，HAM 基于 C++ 语言且无需额外编译器支持，测试结果显示 offload 函数耗时比 LEO 降低 18 倍。Ravi 等人[36]提出一个兼容 x86 众核协处理器的优化编译器和生成工具 Apricot，旨在帮助程序员移植既有多核应用或新开发应用到众核协处理器并最大化性能，做

了 3 方面的努力：1）自动在并行区域插入 LEO 语句，2）基于耗时模型选择 offload 区域，3）应用优化集来最小化数据通信负载。

上述绝大多数研究关注利用自动化的手段来优化异构程序，关注的重点有通信、访存、kernel 执行机制、自动代码转换等方面。但这些研究成果很难真正应用到实际应用的优化，主要有以下几个方面原因：（1）研究成果与实际应用结合可能非常复杂；（2）很多成果仅侧重某个方面进行优化，而实际应用优化时会需要很多方面的优化，多个成果之间未必能够交叉结合；（3）从读者角度看，既要熟悉应用程序，又要了解众核体系结构，而这些优化成果自成小系统，需要重新学习掌握才能使用，且仅在某一方面进行了优化，若要优化其他方面则需要再次迭代，这使得应用程序优化变得非常麻烦，也很少有人愿意去做。

本文工作主要通过 microbenchmark 测评体系结构特性、设计优化策略、实例应用研究的方式，来细致剖析众核 GPU 的体系结构特性，设计一系列优化策略，并在两类实际应用中采取并获得可观的优化效果。这些从 microbenchmark 测评结果设计的优化策略能被简易掌握，并与实际应用相结合，获得可观的性能收益。

1.3 研究内容和贡献

1.3.1 本文研究内容

本文以众核 GPU 为例开展研究，通过测评众核 GPU 存储系统中各存储单元的访存特性，设计出相应的优化策略，构建 GPU 访存优化框架，拟解决“众核体系结构优化困难”的挑战；研究 CPU/GPU 异构系统的异构协同优化技术，针对主机端存储选择展开研究，提出分段式的主机端存储选择模型，讨论了零拷贝、计算与通信重叠、计算与计算重叠等优化技术，拟解决“异构系统高效协同困难”的难题；将 GPU 访存优化框架和主机端存储选择模型应用到几类典型的科学计算应用（高光谱遥感影像降维、声呐信号波束形成）的性能优化中，获得可观的性能提升。本文研究内容主要包括以下几个方面：

（1） warp 级 GPU 存储基准测评。对于众核 GPU 而言，thread 级访存延迟意义不大，warp 作为 GPU 执行的基本单位，其访存延迟在一定程度上能够反映 GPU 存储特性。本文提出 warp 级 GPU 存储基准测评方法，设计并利用两个典型并行存储测评实验测评了共享存储、常量存储、全局存储和纹理存储的访存特性。此外，探讨了寄存器替代局部存储策略、共享存储 bank conflict 及其避免策略、数据类型对全局存储访存带宽的影响等。根据测评结果设计了各存储单元的优化策略和建议，并构建 GPU 访存优化框架。将 GPU 访存优化框架和优化策略应用到两类实际应用，均获得了可观的性能提升，实践结果验证了访存优化框架的可行

性和正确性。

（2） CPU/GPU 异构协同优化研究——主机端存储选择模型。从经典矩阵乘法提出可分页存储与页锁定存储的选择问题；量化探索页锁定存储和可分页存储的各项开销，得到一些基本规律；进一步对主机端存储的访存带宽、PCI-Express 带宽、注册及解除注册时间进行建模，并提出主机端存储选择模型；以高光谱遥感影像 PCA 降维算法为例，对比模型预测时间与真实运行耗时，结果验证了主机端存储选择模型的正确性。此外，基于 CPU/GPU 异构系统，利用 zerocopy 技术设计了两项优化方案，并通过实验进行验证；还实例研究了计算与通信重叠、计算与计算重叠两项经典的异构协同优化技术。

（3）将 GPU 访存优化框架和主机端存储选择模型应用于高光谱影像降维算法，开展并行算法和性能优化研究。针对高光谱遥感影像线性降维中经典的主成分分析、快速独立成分分析和最大噪声分数变换 3 类算法开展并行和优化研究。基于分布存储、共享存储和 GPU 三个并行层次设计热点的并行方案；基于前文提出的 GPU 访存优化框架和主机端存储选择模型，结合应用中的具体访存模式，展开性能优化研究，设计一系列详细的优化策略，并量化测评优化效果；提出面向众核体系结构的高光谱影像并行降维框架。通过实验分析了框架中并行算法的加速比和可扩展性。

（4）利用前文提出的 GPU 访存优化框架和主机端存储选择模型，加速 DFT-CBF 和 MVDR 两类经典的声呐信号宽带波束形成算法。重点针对 DFT-CBF 算法中的 DFT（discrete fourier transform）变换、CBF/Lofar 计算和频带能量整合统计，以及 MVDR 算法中的 DFT 变换、双边雅克比迭代（厄尔米特矩阵特征分解）和方位谱统计等热点，设计 GPU 并行方案；基于 GPU 访存优化框架和主机端存储选择模型，结合各热点的具体计算访存模式，设计出一系列适用的性能优化策略，并量化分析优化效果，实现相应的并行算法。最后通过实验探讨波束形成并行算法的加速比和实时性。

1.3.2 本文贡献

本文主要贡献包括：

1）提出了一种面向 GPU 存储系统的 warp 级基准测评方法，设计了 GPU 的访存优化方法，构建了 GPU 的一种访存优化框架，实验结果表明了该优化框架的有效性；

2）提出了一种 CPU/GPU 异构系统的分段主机端存储选择模型，利用 zerocopy 技术设计了两种优化方案，并验证了所提模型和方案的有效性；

3）构建了面向众核体系结构的高光谱影像并行降维框架，提出了 PCA、

FastICA 和 MNF 三类高光谱影像降维方法的 GPU 并行算法，实验结果表明算法具有良好的加速效果；

4）面向声呐信号波束形成领域，提出了 DFT-CBF 和 MVDR 两类波束形成方法的 GPU 并行算法，实验结果表明算法具有良好的性能提升。

1.4 论文结构

本文以 GPU 为例，进行面向众核体系结构的测评与应用研究，与传统的仅在应用领域开展并行研究的思路不同，本文先通过 microbenchmark 测评，设计出 GPU 优化策略，构建 GPU 访存优化框架，提出主机端存储选择模型，接着将这些研究成果应用到两类实际应用中获得性能收益。论文总共分为七个章节：

第一章绪论。从多核与众核的起源与发展、国内外高性能计算发展趋势和现状两个方面介绍课题研究背景，分析众核体系结构的优势和挑战，阐述本文研究的意义、内容和贡献。

第二章 GPU 背景知识。概述了 GPU 硬件架构、GPU 执行核心、GPU 存储系统、CPU/GPU 异构系统等真正影响 GPU 应用程序性能的关键模块；并分析 GPU 硬件架构可扩展性，测评算术运算延迟，探索分支执行顺序，提出关于 GPU 存储优化和 CPU/GPU 异构协同优化的一系列思考。

第三章 warp 级 GPU 存储基准测评。提出了 warp 级 GPU 存储基准测评方法，测评了共享存储、常量存储、全局存储和纹理存储的访存特性，探讨了寄存器替代局部存储策略、共享存储 bank conflict 及其避免策略、数据类型对全局存储访存带宽的影响，根据测评结果设计相应的优化策略，构建 GPU 访存优化框架，并利用实例研究进行验证。

第四章分段式的主机端存储选择模型。从矩阵乘法提出问题，利用基准测评分析主机端存储的访存带宽、PCI-E 通信带宽、页锁定存储注册与解除注册时间，构建分段式的主机端存储选择模型，通过实例研究验证模型的正确性。此外，基于 CPU/GPU 异构系统，提出并验证了两项利用 zerocopy 技术的优化方案；实例研究了计算与通信重叠、计算与计算重叠两类异构协同优化技术。

第五章基于众核 GPU 的高光谱影像降维算法。描述主成分分析、快速独立成分分析和最大噪声分数变换 3 类降维算法并进行热点分析，基于分布存储、共享存储和 GPU 三个并行层次设计热点的并行方案，研究性能优化策略并量化分析优化效果，提出面向众核体系结构的高光谱影像并行降维框架，利用实验分析并行算法的加速比和可扩展性。

第六章基于众核 GPU 的声呐信号波束形成算法。首先介绍波束形成基本概念和阵列流型计算，接着分 2 个部分分别阐述了两类声呐信号波束形成算法

（DFT-CBF 和 MVDR）的并行和优化研究、及实验结果分析。

第七章总结与展望。对本文研究工作进行总结，并阐明未来可行的研究方向。

第二章　GPU 背景知识

图像处理单元（GPU）是经典的众核协处理器之一，在性能和功耗上均有优秀表现。本章主要介绍与 GPU 性能相关的关键部件，包括 GPU 硬件架构、GPU 执行核心、GPU 存储系统、CPU/GPU 异构系统。详细阐述 GPU 的基本硬件架构，包括 SM 基本结构和 GPU 组成，分析 GPU 架构的可扩展性；描述 GPU 的整数运算和浮点运算、分支执行模式，测评算术运算延迟和探索分支执行顺序；介绍寄存器、局部存储、共享存储、常量存储、纹理存储、全局存储及其位置关系，总结常见的 GPU 存储优化方法，提出关于 GPU 存储优化的一系列思考，引出第三章的研究主题；接着介绍 CPU/GPU 异构系统，总结异构协同优化方法，提出关于异构协同优化的思考，引出第四章的研究要点；最后阐明本文实验平台的硬件配置和软件环境。

2.1 GPU 硬件架构

迄今为止，NVIDIA 已发布 5 类 GPU 硬件架构，分别是 Tesla、Fermi、Kepler、Maxwell 和 Pascal。尽管每代架构均不相同，但基本的组成结构是相似的。本节总结这 5 代 GPU 架构的异同点，分析 GPU 硬件架构的可扩展性。关于 5 代 GPU 架构的详细描述可参见成果[15]。

2.1.1 SM 基本结构

流处理器（streaming multiprocessors，SM）是 GPU 的核心运算部件，其组成和结构直接关系到 GPU 的运算性能。总体来说，SM 由指令 Cache、warp 调度器、指令分发单元、流处理器（streaming processor，SP，又称 CUDA 核心）、双精度浮点运算单元（DP）、特殊功能单元（special function unit，SFU）、存取单元（Load/Store Unites，LD/ST）、只读存储、共享存储、纹理存储单元和 L1 Cache 组成，如图 2.1 所示。L1 Cache 不单独配置，而是与只读存储或共享存储共用相同元器件，比如在 Fermi 架构和 Kepler 架构中与共享存储共存，在 Maxwell 架构和 Pascal 架构中与只读存储共生，而 Tesla 架构的 SM 中没有 L1 Cache。其中指令 buffer、warp 调度器、指令分发单元、Core、DP、SFU 和 LD/ST 等部件共同组成一个处理块，一个或多个处理块以及其他部件组成一个 SM。

在 SM 基本结构的基础上，通过有机调节组成 SM 的部件数量和结构，NVIDIA 推出了 5 代 GPU 架构，表 2.1 展示了 5 代架构中经典 GPU 核心的 SM 组成。表中数据显示，历代 GPU 架构 SM 组成的差异性不小，从大体上有以下几点：

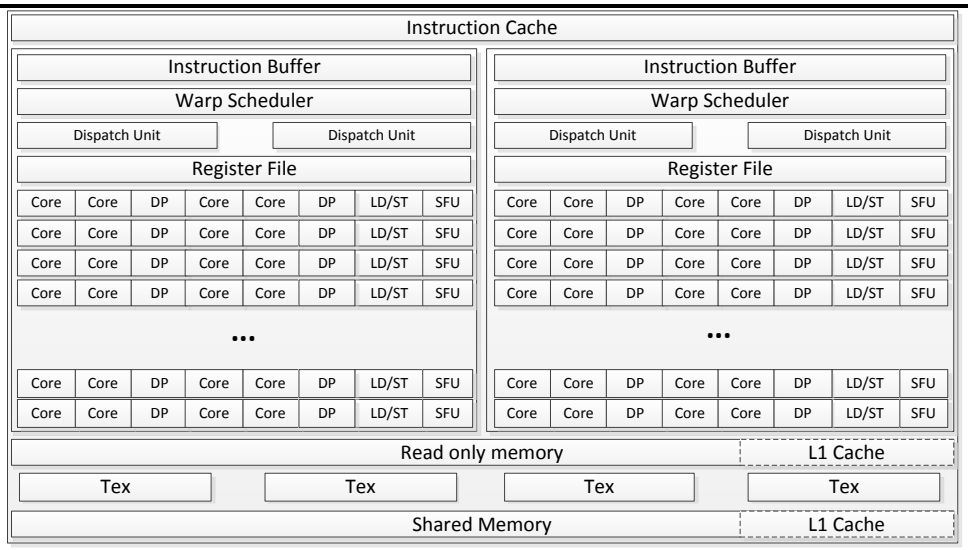


图 2.1 SM 基本结构图

表 2.1 GPU 历代架构典型核心的 SM 组成

架构	Tesla	Fermi	Kepler	Maxwell	Pascal
核心	GT200 a/b	GF100	GK110	GM204	GP100
warp 调度	1	2	4	4	2
指令调度	1	2	8	8	4
Core(SP)	8	32	192	128	64
DP	1	0	64	0	32
SFU	2	4	32	32	16
LD/ST	4	16	32	32	16
Register	16384	32768	65536	65536	65536
共享存储	16KB	16/48KB	16/32/48KB	96KB	64KB
只读缓存	64KB	64KB	64KB	64KB	64KB
L1 Cache	无	共享	共享	只读	只读

（1）寄存器文件数量逐步增加，或单个核心可支配的寄存器文件数量逐步增加；

（2）共享存储资源逐渐增加，或可自由配置的共享存储资源逐渐增加；

（3）L1 Cache 的位置调整优化，在 Tesla 架构位于 SM 外的 TPC 中，到 Fermi 和 Kepler 架构位于共享存储，再到 Maxwell 和 Pascal 架构与只读存储结合；

（4）根据架构定位合理配置 Core、DP、SFU 和 LD/ST 等资源，比如在面向高性能计算的 Kepler 架构和 Pascal 架构中配置了 DP 资源，在面向游戏市场的 Fermi 架构和深度学习市场的 Maxwell 架构未配置 DP 资源。

2.1.2 GPU 组成结构

线程处理器簇（thread processing cluster，TPC）由 SM 控制器及若干 SM 组成，

在 Tesla 架构中还包括 L1 Cache，图 2.2 左图为 TPC 结构示意图。根据产品定位，TPC 中的 SM 数量和组成是可配置的。比如 G80 核心包含 2 个 SM 和 16KB L1 cache；GT200 核心包含 3 个 SM 和 24KB L1 cache；在 GP100 核心中，1 个 TPC 包含 2 个 SM。

图形处理器簇（graph processing cluster，GPC）始于 Fermi 架构，由一个光栅单元、若干 TPC（或 SM）组成，图 2.2 右图是 GPC 结构示意图。GPC 中 TPC 数量并非固定不变的，可以根据产品的市场定位进行调整。比如在 GTX750Ti 中，1 个 GPC 包含 5 个 SM，而在同为 Maxwell 架构的 GTX980 中，1 个 GPC 中包含 4 个 SM。

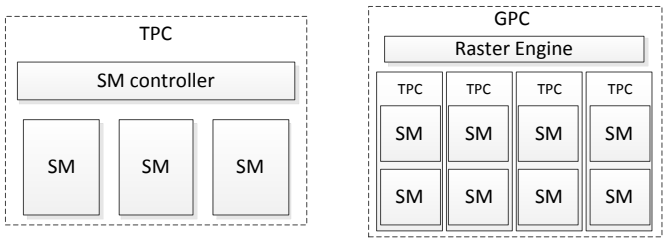


图 2.2 TPC 与 GPC 结构示意图

若干 GPC 结合 L2 Cache 组成流处理器阵列（scalable streaming processor array，SPA），与存储控制器（memory controller，MMC）、PCI-E 接口和线程引擎等元器件共同组成 GPU，如图 2.3 所示。其中每个存储控制器可以支持一定位宽（比如 64 位）的数据合并访存，比如在 GTX480 中，6 个 MMC 可提供 384 bit 访存位宽。图中 high bandwidth memory 2（HBM2）和 NVLink 是 Pascal 架构 GPU 特有的配置。

HBM2 堆叠内存在 Tesla P100 GPU 中首次采用，访存带宽同比增长 3 倍，最高可达 720GB/s。Pascal 架构的访存位宽为 4096 位（8 个 512 位访存位宽的存储控制器），而 Kepler 架构和 Maxwell 架构的 GDDR5 位宽普遍为 384 位，每个存储控制器的访存位宽仅为 64 bit。

NVLink 技术能支持多 GPU 间或 CPU 与 GPU 间的通信，每个 NVLink 连接可以提供 40GB/s 的双向通信带宽，1 个 GPU 至多有 4 条 NVLink 连接。当 NVLink 被用于 CPU 和 GPU 通信时，前提条件是 CPU 和 GPU 都要支持 NVLink 技术，目前只有 Pascal 架构 GPU 和 IBM 的 Power8 CPU 支持 NVLink 技术。当 CPU 和 1 个 GPU 利用 NVLink 技术连接时，如图 2.4 左图所示，有 4 条 NVLink 连接，可提供 160GB/s 的双向通信带宽。而当 CPU 与两个 GPU 互连时，如图 2.4 右图所示，每个 GPU 只能提供 2 条 NVLink 连接与 CPU 相连，GPU 间有两条 NVLink 连接，故双向通信带宽仅为 80GB/s。

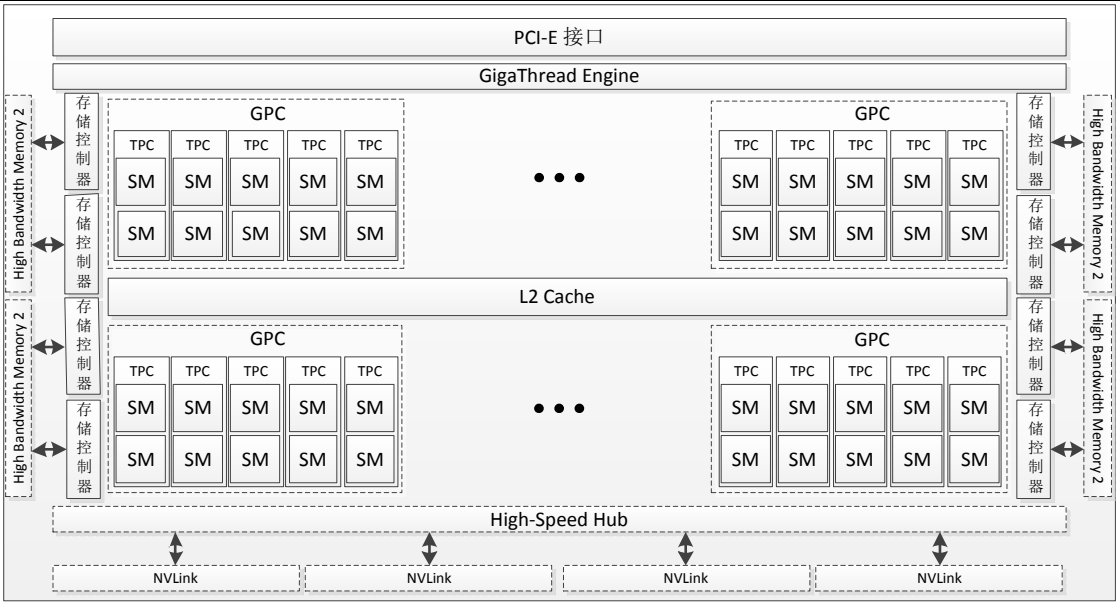


图 2.3 GPU 结构示意图

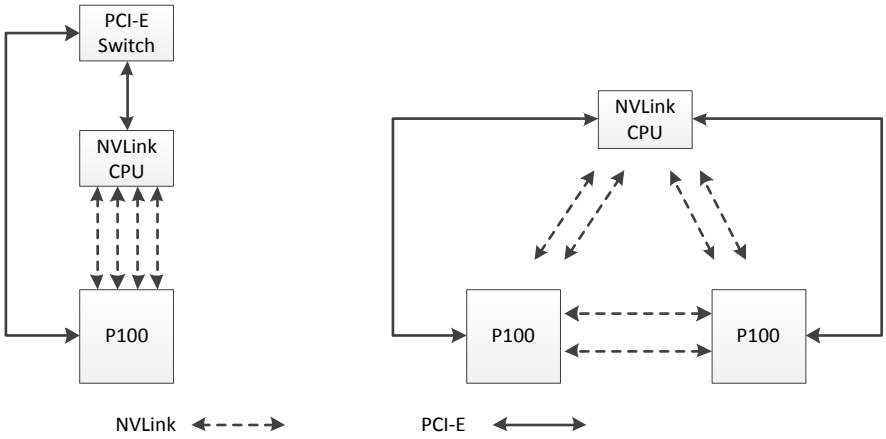


图 2.4 NVLink 连接 CPU 和 GPU

2.1.3 GPU 硬件架构的可扩展性分析

由前文描述可知，GPU 是一个具有多层次可扩展性的体系结构，包括 TPC（GPC）层可扩展性、SM（SMX、SMM）层可扩展性和 SP（Core）层可扩展性。根据产品架构、产品定位、客户需求等不同，可以通过调整这三个层次的可扩展性来确定目标产品的结构。

表 2.2 罗列了一些 GPU 产品的部件组成信息，充分体现了 GPU 优秀的可扩展性：

（1） SM 中 Core 数量随着产品架构不同而变化，且相同架构产品也可调整 CUDA 核心数量，比如 Fermi 架构计算能力 2.0 的 GTX480 和计算能力 2.1 的 GT610 产品核心数量不同，说明 GPU 产品中 SM 的核心数量可扩展性良好。

（2） TPC 中的 SM 数量、GPC 中的 TPC 数量和 GPC/TPC 数量都是可扩展的。

（3） MMC 数量和单个 MMC 的访存位宽都是可扩展的。其中一个 MMC 的访存位宽一般为 64b，但在 GT705 中仅为 32b，而在最新的 Pascal 架构中为 512b，故单个 MMC 的访存位宽是可扩展的；表 2.2 中统计的 MMC 数量信息能有效佐证 MMC 数量的可扩展性。

有了上述三个层次的优秀的可扩展性，可以设计出价格从数百到上万不等的 GPU 产品，从而满足不同类型客户的实际需求。

表 2.2 部分 GPU 产品部件组成信息

GPU 型号	计算能力	Cores/SM	SMs/TPC	TPCs/GPC	GPCs(TPCs)	MMCs
8800GTX	1.0	8	2	0	8	4(256b)
9800GT	1.1	8	2	0	7	4(256b)
GTX285	1.2	8	3	0	10	8(512b)
GTX480	2.0	32	1	3	5	6(384b)
GT705	2.1	48	1	1	1	1(32b)
GT610	2.1	48	1	1	1	1(64b)
K20c	3.5	192	1	1	13	5(320b)
K80m	3.7	192	1	1	13*2	6(384b)
GTX750Ti	5.0	128	1	5	1	2(128b)
GTX980	5.2	128	1	4	4	4(256b)
P100	6.0	64	2	4	7	8(4096b)

2.2 GPU 执行核心

GPU 执行高性能程序的基本操作仍然是算术运算和分支处理。本节将展开阐述 GPU 的整数运算和浮点运算，描述分支执行模式，并设计 microbenchmark 测评 Tesla K20c GPU 中常见的算术运算延迟，探索 GPU 分支执行顺序。

2.2.1 算术运算

2.2.1.1 整数运算

Tesla 架构不支持 32 位整数乘法，仅支持 24 位整数乘法，其中 32 位整数乘法需要 4 条指令才能实现。因此在计算能力 1.X 的设备上最好采用 24 位整数乘法，利用内置函数“__mul24”实现。

自 Fermi 架构开始，GPU 完整支持 32 位整数运算，包括加减乘除、逻辑、条件、类型转换、位操作、窄整型 SIMD 操作等。这些 32 位整数运算基本上都支持标准 C 操作符调用（即普通符号运算）。另外还支持内置函数访问。

GPU 中的位运算除了常见的与运算“&”和或运算“|”运算外，还提供了许

多内置函数，比如__brev()、__clz()、__ffs()、__popc()和_sad()等。这些内置函数在 Fermi（计算能力 2.X）之后的架构中分别对应一条指令，而在 Tesla 架构则被编译为多条指令。另外对于 64 位整型数据（long long），其相应的内置函数改为__brevll()、__clzll()、__ffsll()和__popcll()。

Kepler 架构的 GK110 增加了 64 位漏斗移位指令，用来连接 2 个 32 位整数形成 64 位整数，并左移或右移，返回高 32 位（左移）或低 32 位（右移）。漏斗移位指令最早应用在计算能力 3.5 的产品上。

2.2.1.2 浮点运算

高性能浮点运算是 GPU 相对 CPU 的重大优势之一，特别是其支持的超越函数的精度和性能均强于 CPU。在 Tesla 架构中，SP 采用 IEE754-1985 标准，仅支持单精度运算，在计算能力 1.3 的设备中引入 DP 来支持双精度运算，但性能较差。自 Fermi 架构开始，core（SP）采用 IEE754-2008 标准，能同时支持单精度和双精度运算。

浮点数运算同样支持标准 C 操作符调用，也支持内置函数调用。表 2.3 罗列了一些浮点运算内置函数，其中[rn|rz|ru|rd]指代舍入模式，rn 表示舍入到最近偶数（即最近舍入），rz 表示向 0 舍入（亦称截断），ru 表示向下舍入（向负无穷大舍入），rd 表示向上舍入（向正无穷大舍入）。

表 2.3 浮点运算内置函数

内置函数	运算	内置函数	运算
__fadd_[rn|rz|ru|rd]	加	__dadd_[rn|rz|ru|rd]	加
__fmul_[rn|rz|ru|rd]	乘	__dmul_[rn|rz|ru|rd]	乘
__fmaf_[rn|rz|ru|rd]	乘加	__fma_[rn|rz|ru|rd]	乘加
__frcp_[rn|rz|ru|rd]	倒数	__drcp_[rn|rz|ru|rd]	倒数
__fdiv_[rn|rz|ru|rd]	除	__ddiv_[rn|rz|ru|rd]	除
__fsqrt_[rn|rz|ru|rd]	平方根	__dsqrt_[rn|rz|ru|rd]	平方根

SM 中的特殊功能单元 SFU 实现了 6 种常用超越函数，支持单精度正弦、余弦、对数、指数、倒数和平方根倒数的计算，且性能非常好，相应的内置函数见表 2.4。

2.2.2 分支处理

GPU 采用 SIMT（single instruction multiple threads）执行模式，决定了其分支处理不如 CPU 方便和直接。GPU 中一个 warp 的 32 个线程同时执行相同的指令，因此当遇到条件分支语句时，也要串行逐一执行各个分支。下图 2.5 展示了 warp 遇到的两种分支情况，左图显示 warp 的所有线程处于同一个分支内，此时 warp 直接执行该分支；右图显示 warp 的线程处于不同分支，不同分支需要串行执行，

不符合分支判断的线程空闲。显然图左侧的执行方式效率更高性能更好，在编程时应尽量令 warp 的线程处于同一分支。

表 2.4 SFU 内置函数

内置函数	运算	内置函数	运算
__cosf(x)	cosx	__log10f(x)	$\log_{10}x$
__exp10f(x)	10^x	__powf(x,y)	x^y
__expf(x)	e^x	__sinf(x)	sinx
__fdividef(x,y)	x/y	__sincosf(x,sptr,cptr)	*s=sinx
__logf(x)	lnx		*c=cosx
__log2f(x)	$\log_2x$	__tanf(x)	tanx

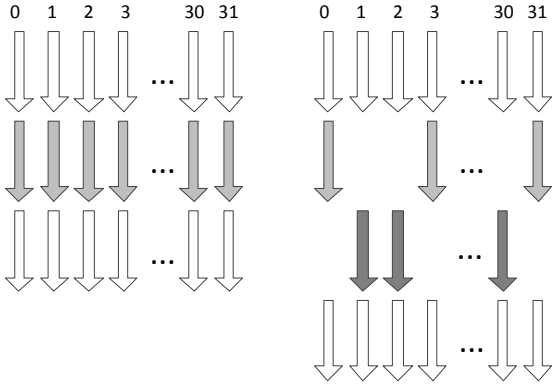


图 2.5 warp 分支处理示意图

2.2.3 算术延迟与分支顺序测评

2.2.3.1 算术运算延迟测评

GPU 的算术运算执行速率在一定程度上决定了 GPU 程序的运算性能，故 GPU 的算术运算延迟是程序员感兴趣的焦点之一。文献[37]对基于 Tesla 架构的 GT280 GPU 进行了算术运算延迟测评，结果显示 uint/int 类型的加减法运算仅需 24 个时钟周期，乘法运算需要 96 个时钟周期，乘加运算需要 120 个时钟周期，其中 uint 的除法仅需 608 个时钟周期，int 的除法需要 684 个时钟周期；float 类型的加法、减法、乘法、乘加运算的时钟延迟是 24 个时钟周期，而 double 类型的加法、减法、乘法、乘加运算的时钟延迟是 48 个时钟周期，float 类型除法的延迟是 137 个时钟周期，而 double 类型的除法延迟是 1366 个时钟周期。

本文针对 K20c GPU 进行算术运算延迟测评，设计了一组 microbenchmark（详细代码参见成果[15]），测定了几种常见的算术运算（加、减、乘、除、乘加）的延迟信息，见表 2.5。表中数据显示，除法运算比较耗时，uint 类型的除法耗时 144 个时钟周期，int 类型的除法耗时 169 个时钟周期，float 类型除法运算需要近 53 个时钟周期，double 类型的除法运算需要近 90 个时钟周期；加法、减法、乘法和乘加运算的延迟均相近，约为 9 个时钟周期，其中 double 类型的运算约为 10 个时

钟周期。

表 2.5 算术运算延迟/clock

	uint	int	float	double
add	9	9	9	10
sub	9	10	9	10
mul	10	9	9	11
div	144	169	53	90
mad	8	9	9	10

2.2.3.2 分支执行顺序探索

当分支处理不可避免时，分支的执行顺序往往是性能优化的另一个关注点。Henry 等人[37]测试了 GT280 GPU 的分支执行顺序，结果显示分支处理顺序与线程 ID 号无关，仅与 if 和 else 顺序相关，其中 else 分支优先执行，if 分支后执行。

本文设计 GPU 分支顺序探索代码（参见成果[15]），用来探索 Tesla K20c GPU 的分支执行顺序。主要思想是给 warp 内的线程的变量赋不同值，利用 if 和 else 分支进行计时和运算，根据时间结果分析 GPU 的分支处理机制。其中根据 warp 内的线程变量赋值的不同，将代码分为两个版本：（1）if_else 版本，赋值语句为“int x=threadIdx.x;”；（2）if_else_1 版本，赋值语句为“int x=31-threadIdx.x;”。根据这两个版本的测评结果，可以分析得出一些有益的结论。

选取其中一组测试结果记录在表 2.6 中。表中数据已按时间顺序重新排列，从结果数据可以总结出以下结论：（1）所有线程执行的分支都不在同一时刻的，说明 warp 内分支处理过程确实是串行执行的；（2）if_else 和 if_else_1 两个版本的分支执行顺序是一致的，说明分支处理顺序与 if 和 else 顺序无关，仅与线程索引相关，且按一定的线程索引顺序执行。结论与文献[37]相悖，可能原因是不同架构的 GPU 产品所采取的分支处理策略不同。

2.3 GPU 存储系统

实际的应用程序执行时，大部分时间消耗在存储访问上，因此提高访存效率是提高程序性能的关键之一。GPU 存储系统提供了种类丰富的可编程存储单元，辅助程序员优化 GPU 程序性能。本节着重介绍 GPU 存储体系、存储优化方法，并提出一些思考以引出后文研究主题。

2.3.1 GPU 存储体系

GPU 包含有种类丰富的可编程存储单元，如寄存器、局部存储、共享存储、常量缓存、全局存储、纹理存储、页锁定存储（Host 端）和可分页存储（Host 端）等。如何合理使用这些存储单元是 GPU 程序性能优化的关键之一。图 2.6 展示了

Kepler 架构 GPU 存储体系基本结构，表 2.7 描述了设备端存储单元的基本信息。

表 2.6 K20c 分支测试结果

if_else		if_else_1	
threadID	clock	threadID	clock
0	18812355	0	19163294
1	18812419	1	19163360
2	18812483	2	19163422
3	18812547	3	19163486
16	18812611	16	19163550
17	18812675	17	19163614
18	18812739	18	19163678
19	18812803	19	19163742
4	18812867	4	19163806
5	18812931	5	19163870
6	18812995	6	19163934
7	18813059	7	19163998
20	18813123	20	19164062
21	18813187	21	19164126
22	18813251	22	19164190
23	18813315	23	19164254
8	18813379	8	19164318
9	18813443	9	19164382
10	18813507	10	19164446
11	18813571	11	19164510
24	18813635	24	19164574
25	18813699	25	19164638
26	18813763	26	19164702
27	18813827	27	19164766
12	18813891	12	19164830
13	18813955	13	19164894
14	18814019	14	19164958
15	18814083	15	19165022
28	18814147	28	19165086
29	18814211	29	19165150
30	18814275	30	19165214
31	18814339	31	19165262

（1）寄存器和局部存储是线程私有的，寄存器资源位于 SM，数量有限，线程运行时动态分配；局部存储在逻辑上等同寄存器，但其物理空间位于显存中，访存较慢。

（2）在 Kepler 架构上，共享存储与 L1 cache 使用相同的硬件资源，总大小为 64KB，其中共享存储可通过编程控制，且线程块内所有线程可访问。

（3）每个 SM 上均有常量缓存，主要支持常量存储的缓存和广播功能。

（4）全局存储和纹理存储位于显存，其中纹理存储是只读存储。

（5）主机端存储包括页锁定存储和可分页存储，可与设备端的常量存储和全局存储进行数据交换。

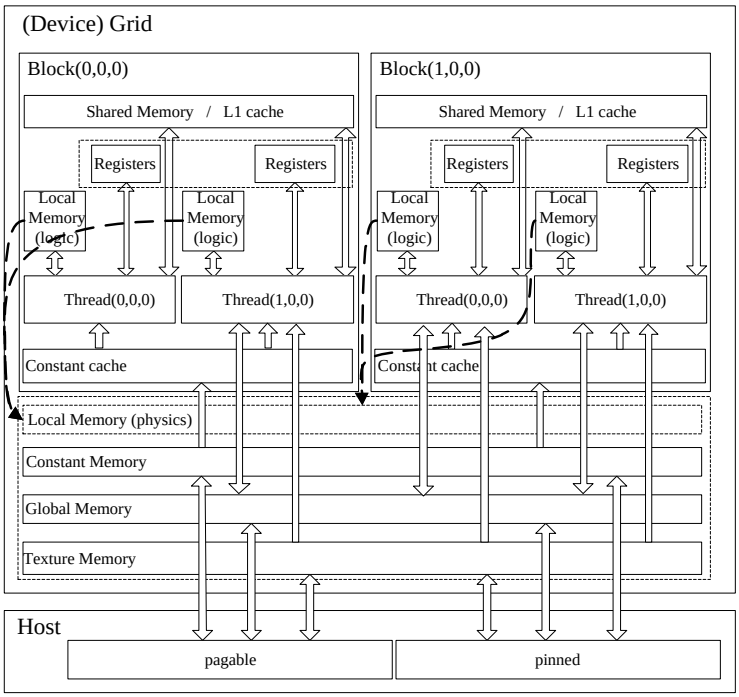


图 2.6 GPU 存储结构图

表 2.7 GPU 存储基本信息

memory	location	cache	access	scope
register	on chip	N/A	R/W	thread
local	off chip	No	R/W	thread
shared	on chip	N/A	R/W	block
constant	off chip	Yes	R	grid
global	off chip	Yes	R/W	grid
texture	off chip	Yes	R	grid

2.3.2 存储优化方法

GPU 存储优化的一个总体原则是：用延迟小的存储单元替换延迟大的存储单元。比如用常量存储来优化全局存储中的不变量，存储空间要小于 48KB；用纹理存储的缓存通道优化全局存储中数据量较大的不变量；用寄存器、共享存储等替代全局存储。

此外，在使用全局存储、共享存储、常量存储时，也需要注意一些访存细则：

1）全局存储需要合并/对齐访存。访问全局存储的本质是通过存储控制器访问显存空间，GPU 一般配置有 6 个存储控制器，每个存储控制器可以访问 64 bit 数据，故 GPU 能够同时访问 384 bit 数据。若全局存储的访问是对齐、可合并的，在

存储控制器访问全局存储时，一个或数个 warp 的线程访问可以合并成一次或数次存储控制器访问。

2）共享存储需要避免 bank conflict。bank conflict 将导致共享存储无法同时访问，产生 bank conflict 的线程只能逐个依次访问共享存储。因此避免 bank conflict 是使用共享存储优化时的基本要求。常见的避免 bank conflict 的方法有数据偏移、矩阵转置等。

3）使用常量存储时，必须广播访问数据。常量存储只能广播访问，若访问无法广播，即不同线程访问不同数据，将导致严重的访存开销。

2.3.3 关于存储优化的思考

从前文的存储优化方法叙述中，要用延迟小的存储单元替代延迟大的存储单元，首先需要知道各存储单元的量化访存延迟，仅凭表 2.7 的位置信息明显还不够直观，此时需要设计 microbenchmark 来量化测定各存储单元的访存延迟。

在访问全局存储时，需要对齐访问、合并访问。那么访问对齐与否是否会对访问性能产生影响？影响多少？什么情况能够合并访问？访问连续与否是否会对合并访问的性能产生影响？影响多少？这些都是程序员在性能优化过程中所关心的热点问题。

在使用共享存储优化程序性能时，需要避免 bank conflict。此时也存在很多问题：bank conflict 对程序性能影响多大？bank conflict 的数量与程序性能的关系如何？怎么才能有效避免 bank conflict？bank conflict 不可避免时，与是否使用共享存储如何取舍？

此外，在判断是否使用共享存储进行性能优化时，更需要考虑存储的复用率，只有将复用次数最多的数据存储在共享存储中，才能最大化发挥共享存储的性能优势。

常量存储只能用来优化广播访问的常量数据。那么为什么常量存储只能广播？如果常量存储访问时，不同的线程并行访问常量存储单元会产生什么影响？影响多大？

此外，局部存储的物理空间位于显存，访存较慢，而现有文档和资料并没有说明寄存器和局部存储的分配策略，是否可以找到影响私有变量在寄存器和局部存储上分配的影响因素？以及这些影响因素的影响边界？

由于 GPU 是商用协处理器，NVIDIA 公司不可能公布底层技术细节，因此上述问题很难找到答案。带着上述思考，本文第三章针对 GPU 存储系统开展一系列研究，通过设计 microbenchmark 来测评 GPU 存储特性，利用存储特性设计真实有效的 GPU 存储优化方法。

2.4 CPU/GPU 异构系统

GPU 作为一类经典的众核协处理器，其本身无法单独工作，需要与 CPU 搭配组成异构系统才能真正执行程序。在一个 CPU/GPU 异构系统中，除了 CPU 与 GPU 自身的浮点运算能力可以挖掘外，还有很多异构协同的优化策略可供采纳和选择。本节将介绍 CPU/GPU 异构系统组成、常见的异构协同优化方法，提出一些关于异构协同优化的思考，引出论文第四章的研究要点。

2.4.1 CPU/GPU 异构系统组成

CPU/GPU 异构系统中，CPU（主机端）与 GPU（设备端）利用 PCI-E（或 NVLink）连接，各自存储（主存和显存）独立，数据通信是必不可少的环节。其中，CPU 擅长逻辑事务处理，GPU 擅长密集的浮点运算，图 2.7 展示了典型的 CPU+GPU 异构系统。在启动 kernel 函数进行 GPU 计算前，必须先通过 PCI-E 将数据传到 GPU 端；GPU 得到计算结果后，需将结果数据传回给主机内存。CPU 与 GPU 间的通信是必不可少的，由于 PCI-E 接口的通信带宽限制，优化数据通信开销是必需考虑的要素之一。

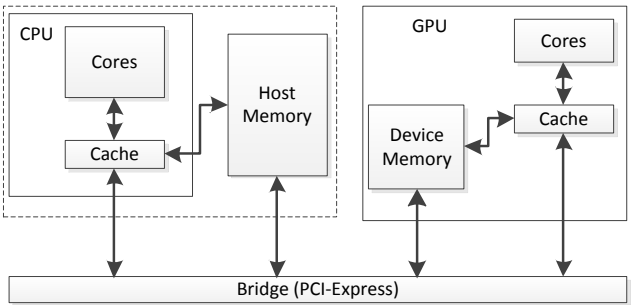


图 2.7 典型的 CPU/GPU 异构系统

2.4.2 异构协同优化方法

在优化多核与众核异构程序时，有一些常见的异构协同优化方法可供选择，比如减少数据通信量、计算与计算重叠、计算与通信重叠等。而在 CPU/GPU 异构系统中，又赋予了一些独特的特性，比如页锁定存储、零拷贝（Zerocopy）等，合理利用这些特性亦是异构程序性能优化的关键。

减少通信量：通过合理组织程序流程，改进算法，缩减 CPU 与 GPU 的交互，进而缩短通信时间，以达到提升性能的目的。

计算与计算重叠：CPU 与 GPU 同样都具有强大的浮点计算能力，充分利用两者同时进行运算，可以有效地重叠计算流程，从而减少计算时间。

计算与通信重叠：利用多流并发技术，GPU 计算与 CPU/GPU 数据通信是可

以同时执行的。计算与通信的重叠进行能够有效减少重叠部分的时间消耗，缩短整体程序的运行时间。

页锁定存储优化：页锁定存储是 CUDA 特有的主机端存储属性，通过将存储页锁定在内存中，以提高与 GPU 通信的速率。此外页锁定存储也是计算与通信重叠的必要条件之一。

零拷贝：零拷贝存储是利用映射锁页内存实现 CPU 和 GPU 的直接存储访问，掩藏数据传输细节。映射锁页内存可以为 CPU 和 GPU 分别提供同一块主机端存储空间的地址，CPU 和 GPU 分别根据各自指针访问相应的数据。

2.4.3 关于异构协同优化的思考

关于页锁定存储的使用，在数据量大时，传输速率确实可以得到提升，而且亦能实现异步通信，但从整体运行时间上看，是否真的能够减少运行时间？页锁定存储的使用是否会引入额外开销？

零拷贝可以实现 CPU 和 GPU 的直接数据访问，但不可避免还是需要通过 PCI-E 进行隐式地数据通信，依然会涉及数据传输。此时零拷贝的访问模式就可能对通信速率产生巨大影响，初步预测零拷贝的访问模式将比普通的全局存储访问模式更加敏感和严格。那么什么样的零拷贝存储的访问模式才能最大限度发挥 PCI-E 带宽？

现有的文档、书籍和资料难以解答上述问题，针对这些问题，在本文第四章，通过设计 microbenchmark 测评 CPU/GPU 异构系统的一些基本特性，构建分段式的主机端存储选择模型，并讨论了使用 zerocopy 技术的两项优化方案。

2.5 本文实验平台

本文实验平台是一个典型的 CPU/GPU 异构系统，其中搭载 2 个 8 核 Intel Xeon E5-2670 CPU、2 个 NVIDIA Tesla K20c GPU，配置了 64GB 运行内存和 PCI-E 2.0 接口。程序编译环境为 Red Hat Enterprise Linux Server release 6.2 (Santiago) 系统，搭载 ICC version 13.0.0.079 编译器、CUDA 5.5 工具包和 MVAPICH2 软件包。

Intel(R) Xeon(R) E5-2670 CPU 拥有 8 个核心，其运行主频为 2.6GHz， 20MB 的 cache 空间，cache line 按 64 位对齐，支持 MMX、SSE 系列和 AVX 的向量运算指令集。

NVIDIA Tesla K20c GPU 的计算能力为 3.5，拥有 13 个 SM，2496 个 CUDA 核心，GPU 的运行主频为 706MHz；配置 5 个存储控制器，提供 320 bit 的存储访问位宽，访存频率为 2600MHz；此外还提供 1280KB 的 L2 Cache，PCI-E3.0 接口。单个 SM 拥有可配置的共享存储/L1 Cache 硬件资源共计 64KB（已配置为 16KB 的

L1 Cache 和 48KB 共享存储），常量存储 64KB，32 位寄存器 65536 个。

2.6 本章小结

本章介绍了 GPU 的核心组成，包括 GPU 硬件架构、GPU 执行核心、GPU 存储系统、CPU/GPU 异构系统；在基础理论描述的基础上，进行一系列分析、测评与思考，比如 GPU 架构的可扩展性分析、算术运算的延迟测评、分支执行顺序探索、关于存储优化和异构协同优化的系列思考。

GPU 体系结构小节总述 GPU 基本硬件架构（SM 和 GPU 的组成），分析 GPU 架构优秀的多层次可扩展性。GPU 执行核心小节对 GPU 运算的两个基本项目（算术运算和分支处理）进行阐述，描述 GPU 的整数运算和浮点运算，测评算术运算延迟，描述了 GPU 的分支执行模式，探索分支执行顺序。GPU 存储系统小节简要阐明 GPU 存储体系中寄存器、局部存储、常量存储、共享存储、纹理存储和全局存储的结构关系及所在位置，阐述一般的 GPU 存储优化方法，提出一些关于 GPU 存储优化的思考，引出论文第三章研究主题。CPU/GPU 异构系统小节描述 CPU/GPU 异构系统的基本组成，总结常见的异构协同优化方法，并基于此提出几点关于异构协同优化的思考，引出论文第四章研究要点。最后说明本文实验平台的基本配置。

本章内容描述了本文研究的基础背景知识，并引发一系列关于性能优化的思考，为后文研究的开展奠定基础和指明方向。

第三章 warp 级 GPU 存储基准测评

GPU 在科学计算中因其得天独厚的性能优势而被广泛使用，但 GPU 编程开发和优化却相当困难，特别是 GPU 存储层次复杂，各存储单元的特性与性能各不相同，是 GPU 应用程序性能优化的重点和难点。对于 GPU 而言，thread 级的访存延迟意义不大，warp 作为 GPU 执行的基本单位，warp 级的访存延迟更能反映 GPU 存储特性。本章旨在用并行的方法测评 GPU 存储单元的 warp 级访存延迟，总结存储特性，设计优化策略并用于实际应用以收获性能收益。

同时本章试图解答 2.3.3 节关于 GPU 存储优化的几点思考。此外，本章研究成果可用于第五章的高光谱影像降维和第六章的声呐信号波束形成，与具体热点的访存模式相结合，设计出具体的优化方法，在热点并行方案基础上大幅度提升性能。

3.1 引言

图像处理单元（GPU）从最初的图像渲染到后来的科学计算，已经在多个领域得到了广泛应用[38-42]。但 CPU/GPU 异构程序开发和优化非常困难，特别是基于 GPU 的存储优化，一直是 GPU 程序开发的瓶颈之一[37,43-44]。

GPU 书籍[9,45]与官方文档[46]提供的信息具有一定的局限性。一方面，定性地介绍了 GPU 各存储单元，没有给出量化测评结果数据；另一方面，给出了一些存储单元特性和优化方法，却没有说明其来源和用处。本章将通过量化测评 GPU 存储单元的访存延迟，总结存储单元特性，导出优化建议，并将之用于实际应用以收获性能收益。

此前，针对 GPU 存储系统的研究仍局限于借鉴传统的指针追逐(Pointer-Chasing)方法，测试 GPU 上单个线程的存储访存延迟，构建 L1/L2 cache 层级等[37,47-51]。由于 GPU 体系结构的特殊性，对于真实应用，GPU 线程不会单独执行，warp 才是 GPU 执行的基本单位。本章创新性地提出通过测评以 warp 线程块为单位的访存延迟来总结 GPU 存储单元特性的方法。

另一方面，GPU 作为众核协处理器，并行度要求非常高，单独的线程访存延迟较难反映存储单元特征，难以导出有意义的优化策略；而本章提出的 warp 级访存延迟可以反映出一些有价值的存储单元特性，进而导出相应的 GPU 优化建议。

一个 warp 内的 32 个线程由同一个指令分发单元控制，同时执行同一条指令，即 SIMT。从某种意义上说，warp 内线程并行执行不像传统 CPU 的线程并行那么自由。针对 warp 内线程并行的特性，设计了两个并行测评实验：广播与并行访存

实验、对齐与连续访存实验。并用来测试共享存储、常量存储、全局存储和纹理存储的 warp 级访存延迟。此外，本文还对寄存器和局部存储的分配策略、共享存储 bank conflict 及其避免、数据类型对全局存储访存带宽的影响等进行了优化探讨。在总结测评结果并设计优化策略和建议的基础上，构建 GPU 访存优化框架，并利用实例研究，探索和验证了访存优化框架的可行性和正确性。

本章主要贡献包括：（1）测定了 Tesla K20c GPU 各存储单元的 thread 级访存延迟；（2）创新性地提出用 warp 级访存延迟来测评存储单元特性，并设计了两个典型并行存储测评实验；（3）测评了共享存储、常量存储、全局存储和纹理存储的 warp 级访存特性；（4）探讨了局部变量数组在寄存器和局部存储上的分配策略、共享存储 bank conflict 及其避免、数据类型对全局存储访存带宽的影响；（5）设计和总结 GPU 各存储单元的优化策略和建议，构建 GPU 访存优化框架；（6）以高光谱影像降维和声呐信号波束形成为例，基于访存优化框架进行优化，获得了理想的性能收益。

3.2 相关工作

本章涉及的相关工作包括以下两个方面：

（1）GPU 存储系统测评研究。

关于 GPU 存储系统测评已有大量研究，主要研究手段是采用指针追逐方法。Volkov 等[47]最早将该方法引入 GPU 测试 8800GTX 的存储层级，揭示了纹理 cache 和一个 TLB 层次；Wong[37]利用 microbenchmark 构建了 GT280 的微架构，揭示了 2 层 TLB 和 cache 层次，讨论了计算核心的执行和分支处理方法。Baghsorkhi 等[48]测评了 C2050 GPU，揭露了 L1 和 L2 数据缓存架构。Meltzer 等[49]研究了 C2070 GPU 的 L1/L2 数据缓存，发现其中 L1 cache 并非使用 LRU 替换策略。Mei 等[50]基于 GTX560Ti 和 GTX780 测评了相应的架构和访存延迟。Mei 等[51]基于三代 GPU 架构（Fermi、Kepler 和 Maxwell）研究 GPU 存储层级，量化测试了全局存储和共享存储的吞吐量和访存延迟。

由于 GPU 特殊的体系结构，上述研究在方法上存在一定的局限性。指针追逐方法最早用于 CPU 的测评，但 GPU 线程与 CPU 线程有着本质区别，且在一个真实应用中，GPU 线程一般不会单独执行。因此这种测试 GPU 单独线程的访存延迟就具有很大的局限性，很难有效反映 GPU 存储的真实特性。相对而言，本文提出的 warp 级存储测评方法能够较好地反映 GPU 存储单元特性。

另一方面，CPU 中线程数量等于 CPU 核数，相应的 cache 层次及大小能够较直观地辅助编程优化。而 GPU 函数执行时都会启动成千上万的线程，其执行的基本单元是 warp 而非线程，并且上万线程的执行性能与单线程执行性能必然差异极

大，故利用单线程访存延迟测定的 cache 层次等信息很难直接导出 GPU 优化策略。而本文的 warp 级存储测评能较好地反映 GPU 存储单元特性，并进而导出相应的 GPU 优化建议。

2）GPU 存储优化研究。

主流的 GPU 存储优化研究关注数据位置布置和访存模式优化两个方面。

数据位置布置是 GPU 存储优化研究热点之一。Ma 等[52]探索了共享存储上的最优数据位置；Chen 等[44]提出了一个便捷的可扩展优化器 PORPLE 来定位 GPU 存储的数据位置。

访存模式优化亦是 GPU 存储优化的热点。Yang 等[53]设计了线下的源到源编译器来促进存储合并和共享存储使用；Zhang 等[54]提出流水线的线上数据改组引擎来减少不规则存储访问；Wu 等[55]通过形式化数据改组，来最小化非合并访存，进行第一复杂性分析并提出几个有效的改组算法。Jang 等[43]探索了各 GPU 存储单元的访存模式，并设计了基于 AMD GPU 的向量化算法和基于 NVIDIA GPU 的存储选择算法。

上述研究主要关注共享存储、常量存储、全局存储和纹理存储等存储单元的研究，暂未涉及寄存器和局部存储的探讨。本文探索了线程私有变量数组在寄存器和局部存储上的分配策略，在一定程度上是对上述研究的补充。

另一方面，上述研究都是基于已有的优化方法和手段（在 GPU 书籍或技术手册已提及），进行前向研究。而本文通过 GPU 存储测评，总结存储单元特性，导出优化策略和建议，构建 GPU 访存优化框架，并用实例研究进行检验，解释了存储优化方法的来源和用处。

3.3 thread 级访存延迟测评

2.3.1 节已简单介绍 GPU 存储体系，描述各个存储单元的位置、访存方式、作用范围等信息。但这些信息只能提供一个感性的认知，无法明确具体哪个存储单元访问快，快多少。因此本节通过 microbenchmark 测定 GPU 各存储单元的访存延迟。

测试数据的构造：除了寄存器外，其他存储单元都支持数组存储，可以采用指针追踪的方法测试访存延迟。由于编译可能包含诸如流水线、运算与访存重叠等优化，需要设计一种依赖关系来回避这些编译器优化对访存延迟测评的影响。本文采用一种固定跳步的指针追逐方法，即声明一个 int 数组，索引为 index 的元素值为（index+step）；访问时用本次读取到的数据作为下一次访问的索引。下面是数组构造和访问的伪代码。

```
for i = 1 to len
    array[i]=(i+step)%len;
end for
```

```
for i = 1 to n
    p=array[p];
end for
```

寄存器访存延迟的测定：寄存器也可以声明数组，但指针追逐方法（p=array[p]）的访存模式将导致任意大小的数组都无法声明在寄存器中（关于私有变量数组在寄存器和局部存储的分配策略将在 3.6.1 节详细探讨），因此寄存器的访问延迟测定方法需要重新构造。简单的访问将会被编译器优化，本文设计了下面的访问过程，每个数据的访问都需要依赖前一个数据的赋值，最后将 r 写到全局存储，若没有输出将被编译器直接优化。

```
__global__ void latency_register
INPUT: NUM
OUTPUT: int *out, double *time
{
    int r, x=2,y=5,z=7,q=1, p=3;
    t0=clock();
    for i = 1 to NUM
        r=p;p=q;q=x;x=y;y=z;z=r;
    end for
    t1=clock();
    out[0]=r;
    time[0] = (t1-t0)/NUM;
}
```

设置数组长度为 2048，循环测试 step 为 1 到 1024 时 Tesla K20c GPU 各存储单元的访存延迟，统计平均值见表 3.1。表中数据显示，寄存器的访存延迟最小，几近于 0；其次是共享存储，访存延迟约为 48 个时钟周期；常量存储和纹理存储拥有片上缓存，平均访存延迟相当，约为 110 ~115 个时钟周期，接近全局存储的一半；局部存储和全局存储位于显存，访存延迟相近，数据显示局部存储的访存延迟比全局存储的访存延迟稍小。

表 3.1 GPU 存储延迟/clock

memory	register	constant	shared	local	global	texture
latency	0.19	110.13	47.69	203.75	218.86	115.60

其中常量存储的访存延迟与 step 数值相关。图 3.1 罗列了 step 为 1 到 32 的常量存储访存延迟，当 step 从 1 到 16 时，访存延迟逐步增加，大于 16 时，访存延迟趋于平稳（而同样采用缓存机制的纹理存储的访存延迟基本稳定）。上述现象也为常量存储优化提供了一条思路，即在构造常量存储时，需要考虑数据的访问顺序，尽量将连续访问的数据邻近存储。

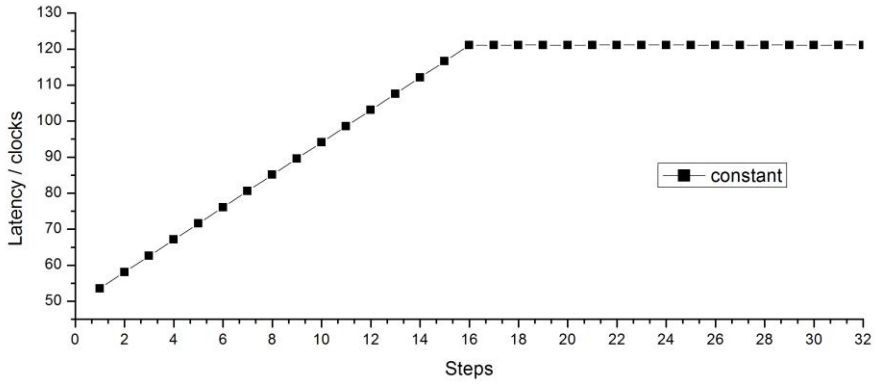


图 3.1 step 对常量存储访存延迟的影响

3.4 warp 级并行测评方法

3.4.1 warp 级延迟测评

GPU 作为众核协处理器，与传统多核 CPU 的一个重要区别是执行模式的不同。CPU 在标量处理时执行模式为 SISD，向量处理时的执行模式为 SIMD；而 GPU 的执行模式为 SIMT。Warp 才是 GPU 执行的基本单位。

Warp 内的所有线程在无分支分歧时执行相同的指令，但访问的数据位置不确定；而在有分支分歧时相同分支的线程执行相同指令，在一个分支执行时其他分支的线程等待。将 warp 内多线程访问数据的模式归结为广播与并行访存两类，多个线程同时访问同一个数据为广播，而多个线程同时访问不同数据则为并行访存。对于并行访存，访问是否对齐、数据是否连续可能会影响并行访存性能。据此，设计两个典型的并行存储测评实验，用于测定 GPU 各存储单元的性能特性。

3.4.2 广播与并行访存实验

广播和并行访存往往直接关系到存储单元的访问性能：广播支持多个线程同时访问同一数据，而并行访存则支持多个线程同时访问不同数据。广播和并行访存带来的性能提升往往是成倍的，且线程越多效果越显著。特别是在 GPU 这类众核协处理器中，大量线程的并行执行，广播与并行访存就变得尤为关键。

设计图 3.2 所示的广播与并行访存实验来测评 GPU 存储单元的 warp 级访存延迟，分别是：①每个线程访问独自的数据；②每两个线程访问相同的数据；③所有线程访问相同的数据。第①组全为并行访问、广播数为零，接着并行访问随之减少、广播数增加，到最后一组全为广播，并行访存度为零。

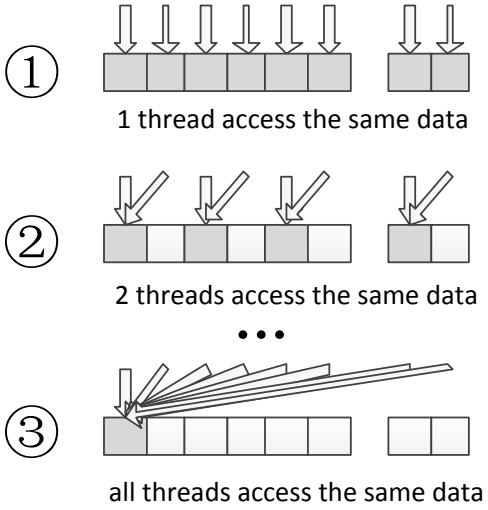


图 3.2 广播与并行访存实验

根据以上广播与并行访存实验描述，设计相应的测试代码。输入数据构造时，令数据值等于其索引号，输入数据构造的伪代码如下。

```
for i = 0 to N
    in[i] = i ;
end for
```

GPU 端访存测试代码如下所示：其中输入的 in1 和 in2 数组均通过上述构造代码生成，step 为广播数（测试时分别设置为 1,2,4,8,16,32）；输出 out 是为了避免编译器优化代码，输出 time 记录测时结果；NUM 为访存测试次数，最终求取时间平均值。

```
__global__ void broadcast_parallel
INPUT: int *in1, int *in2, int step , int NUM
OUTPUT: int *out, double *time
{
    int p,q=(threadIdx.x/step*step);
    int t0=clock();
#pragma unroll
    for i = 0 to NUM
        p=in1[q];
        q=in2[p];
    end for
    int t1=clock();
    out[blockDim.x*blockIdx.x+threadIdx.x]=p+q;
    time[blockDim.x*blockIdx.x+threadIdx.x] =
        (t1-t0)/NUM/2;
}
```

利用上述实验的存储单元访存延迟结果，可以分析得到以下结论：

（1）由于从①到③并行访存度逐步减少，广播数逐步增加，假如①到③访存

延迟逐步减少，说明该存储单元不支持并行访存或并行访存开销过大，而对广播功能的支持较好。

（2）假如①到③访存延迟逐步增加，说明该存储单元支持并行访存，而不支持广播。从①到③随着广播数据量增加，访存延迟逐步增加，故不支持广播；而从③到①并行访存增加，而访存延迟减少，故支持并行访存。

（3）如果访存延迟基本相当，可能存在两种情况，即同时支持广播和并行访存，或者两者都不支持。这种情况下，可以通过对比 warp 级的访存延迟和 thread 级的访存延迟进行区分，若两者相当则同时支持广播和并行访存；若 thread 级访存延迟较小，说明存储单元不支持广播和并行访存。

3.4.3 对齐与连续访存实验

通过广播与并行访存实验确定存储单元支持并行访存后，进一步探讨不同并行访存模式对性能的影响。从 warp 级的 32 个线程角度看，不论何种并行访存模式，都可以归结到两个角度，分别是线程访问是否对齐、访问的数据是否连续。故本节从这两个角度出发，设计了一个对齐与连续访存实验来深入探索 GPU 存储单元的并行访存性能。

在对齐与连续访存实验中，将数据看成一个矩阵，设计图 3.3 所示的 3 种并行访存模式，分别是：①访问的数据是连续存储的，并且所有线程对齐顺序访问；②访问的数据是连续存储的，但线程随机乱序访问数据；③访存的数据不连续存储，所有线程对齐顺序访问。

根据对齐与连续访存实验描述，设计相应的实验测试代码。这里的关键在于构造实验输入数据，访存测试可利用线程指针追逐的方法实现。

对于对齐访问和数据连续的情况①，输入数据只需令数据值等于其索引即可。对于数据连续但访问不对齐的情况②，输入数据生成的伪代码如下。

```
void init_in_32
INPUT: int n
OUTPUT: int *a
{
    int p[32];
    for i=0 to 32
        p[i]=i;
    end for
    for i=0 to n, step=32
        for j=0 to 32
            int jj=rand()%(32-j);
            a[i+j]=p[jj];
            for k=jj to (32-j)
```

```
                    p[k]=p[k+1];
                end for
            end for
            for j=0 to 32
                p[j]=a[i+j];
                a[i+j]+=i;
            end for
        end for
    }
```

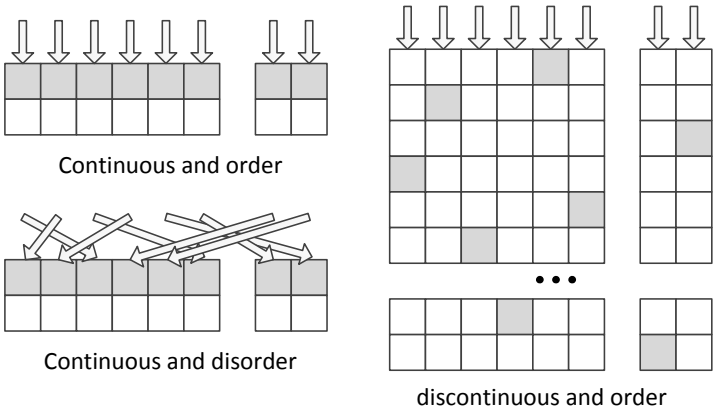


图 3.3 对齐与连续访存实验

对于线程对齐但数据不连续的情况③，需要构造一个数据矩阵，该矩阵每行 32 个元素，对应一个 warp 的 32 个线程，要令每个线程每次访问的数据位于所属数据列，且所有的访问数据均乱序。基于上述目标，设计了下面的数据生成伪代码。

```
    void init_in_512
    INPUT: int n
    OUTPUT: int *a
    {
        const int nn=n/32;
        int q[nn],b[n], p[32];
        for i=0 to nn
            q = i ;
        end for
        for i = 0 to n, step=nn
            for j = 0 to nn
                int jj=rand()%(nn-j);
                b[i+j]=q[jj];
                for k = jj to (nn-j)
                    q[k]=q[k+1];
                end for
            end for
```

```
            for j = 0 to nn
                q[j]=b[i+j];
            end for
        end for
        for i = 0 to 32
            p[i]=i;
        end for
        for i = 0 to 32
            for j = 0 to nn
                a[j*32+i]=b[i*nn+j]*32+p[i];
            end for
        end for
    }
```

利用对齐与连续访存实验结果，可以分析得出以下结论：

（1）若①和②访存延迟相当，两者的区别是线程访问对齐与否，说明访问是否对齐并不影响存储单元的并行访存性能；

（2）若②比①访存延迟高，则说明访存不对齐将导致性能下降，此时需要考虑访问对齐；

（3）若①和③访存延迟相当，两者区别是访问的数据是否连续存储，说明访问的数据不连续不会影响存储单元的并行访存性能；

（4）若③比①访存延迟高，说明不连续访问比连续访问慢，故数据必须连续存储才能发挥存储单元性能。

3.5 warp 级 GPU 存储测评

本节利用 3.4 节提出的两个 warp 级存储测评实验分别对 K20c GPU 中的共享存储、常量存储、全局存储和纹理存储进行测评，总结分析存储访问特性。

3.5.1 共享存储并行测评

3.5.1.1 共享存储的广播与并行访存测评

当 warp 访问共享存储不存在 bank conflict 时，共享存储的访存性能如何？能否支持广播？是否支持并行访存？

带着上述问题，进行共享存储的广播与并行访存实验，分别令 warp 中 1、2、4、8、16 和 32 个线程访问相同数据，测试 warp 级访存延迟。实验结果见图 3.4。图中数据显示，不同数量线程访问相同数据时，访存延迟基本不变，且与 thread 级访存延迟相当。说明共享存储同时支持广播和并行访存。

有意思的是，图中数据显示，当线程数量达到一定程度时，访存延迟反而下降，其中两个关键阈值是 128 和 256，当线程数量大于 128 时，访存延迟开始下降，当

线程数量到达 256 后，访存延迟趋于稳定。

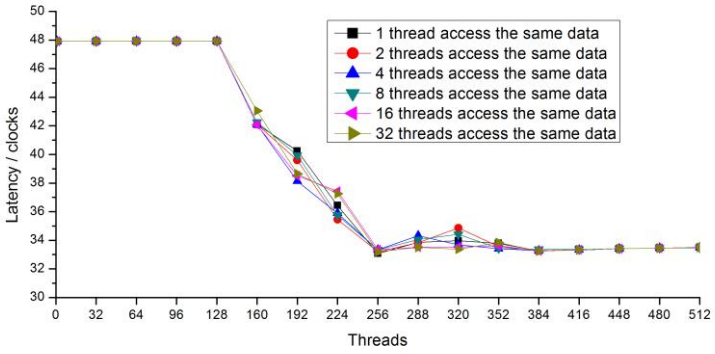


图 3.4 共享存储的广播与并行访存延迟

3.5.1.2 共享存储的对齐与连续访存探索

本节对共享存储进行对齐与连续访存实验，实验结果见图 3.5。图中数据显示，3 种访问模式访问共享存储的时钟延迟几乎相等，说明只要保证不存在 bank conflict，线程访问对齐与否、数据存储连续与否都不会影响共享存储的访存性能。另外从图中也可看出随线程数增加到一定阈值 128，访存延迟开始减少，直至 256 达到稳定。

3.5.2 常量存储并行测评

常量缓存提供了缓存和广播两大主要功能，图 3.1 数据显示了常量存储的缓存特性。本节主要研究常量存储的广播与并行访存机制。试图回答以下几个问题：常量存储是否支持广播？常量存储是否支持并行访存？warp 数量增加时，访存延迟如何变化？block 数量增加时，访存延迟又如何变化？

为了解答上述问题，进行常量存储的广播与并行访存实验：分别令一个 warp 内每 1、2、4、8、16 和 32 个线程访问同一个数据，循环测定 1 个 block 不同 threads 数量时访存延迟的变化，图 3.6（a）统计了最大访存延迟。图 3.6（a）纵向数据显示，一个 warp 中不同数量线程访问相同数据的访存延迟差距巨大，呈现阶梯状，访问同一数据的线程数量越多，访存延迟越小。由此可知：（1）常量存储的访问是串行的，即常量存储不支持并行访存；（2）常量存储支持广播。从图 3.6（a）横向看，当线程数增加且小于某阈值（256）时，访存延迟基本不变；而超过该阈值后，访存延迟少量增加并再次达到平稳。这种现象展示了 warp 数量增加时常量存储广播性能的变化规律。

用一组对比实验来回答最后一个问题：令每个 warp 访问同一数据（不同 warp 访问的数据不相同），分别测试每个 block 内 32,256,512,1024 个线程时，block 数量变化时常量存储访存延迟的变化。测试结果如图 3.6（b）所示，可知：（1）block 数量从小到大变化时，存在一个阈值，阈值之前访存延迟和 block 数量成正比例关

系，其后访存延迟基本恒定；（2）阈值大小与 block 内的 threads 数量相关；（3）唯一影响恒定后访存延迟的因素是 block 内的 threads 数量，且线程越多访存延迟越大；（4）当 block 数量够小（小于阈值）时，访存延迟与 block 数量正相关，当 block 数量大于阈值时，访存延迟不再受 block 数量的影响，仅受限于 block 内的 threads 数量。

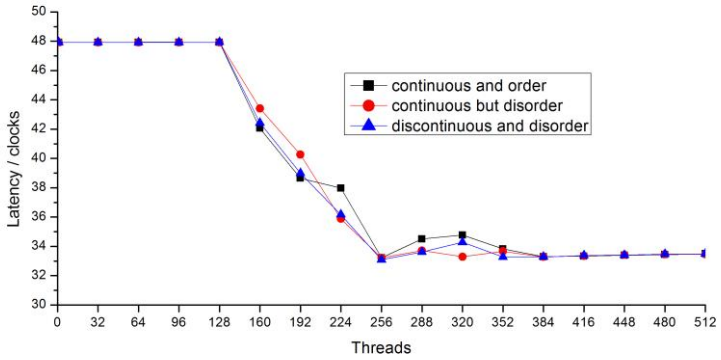
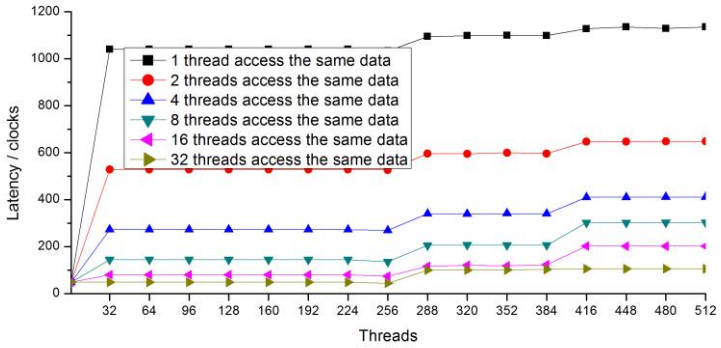
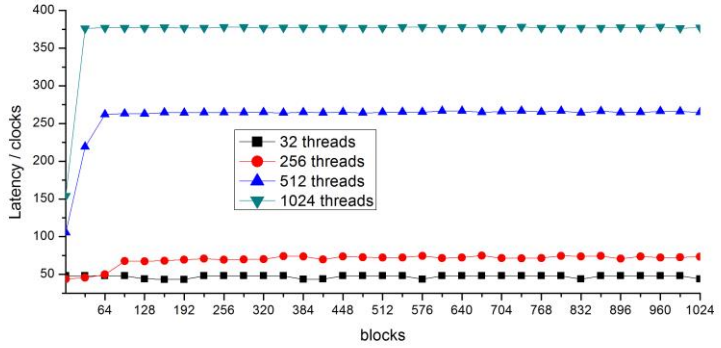


图 3.5 共享存储的对齐与连续访存延迟



（a）threads 变化导致的访存延迟变化



（b）blocks 变化导致的访存延迟变化

图 3.6 常量缓存的广播与并行访存延迟

3.5.3 全局存储并行测评

3.5.3.1 全局存储的广播与并行访存测评

本节对 K20c GPU 进行全局存储的广播与并行访存实验，分别设置 warp 内访问相同数据的线程数量为 1,2,4,8,16 和 32，测试全局存储的访存延迟，实验结果见图 3.7。图中横向数据显示，线程增加时访存延迟变化不大（5~10 clocks），相对与总体访存延迟（约 215 clocks）可以忽略不计，说明线程配置对全局存储的访存延迟影像较小。从纵向看，warp 内广播的数据数量越多，访存延迟越小，但访存延迟总体变化较小，并且与 thread 级访存延迟相当。说明全局存储同时支持广播和并行访存，且广播比并行访存性能稍好。

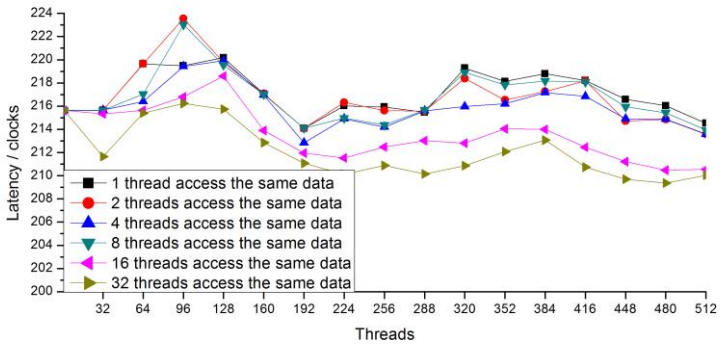


图 3.7 全局存储的广播与并行访存延迟

3.5.3.2 全局存储的对齐与连续访存探索

对全局存储进行对齐与连续访存实验，测得实验结果数据见图 3.8。图中数据显示：（1）数据连续时，对齐访问与乱序访问的访存延迟基本一致，说明对齐访问与否不影响全局存储性能；（2）数据不连续的访存延迟明显比连续的访存延迟大很多，说明连续访存是发挥全局存储性能的必要条件；（3）图 3.8（a）显示访存不连续比访存连续慢了 1 倍左右，访存连续时全局存储的访存延迟不受线程数量的影响，而访存不连续将导致全局存储的访存延迟随线程数量增加而缓慢增加；（4）图 3.8（b）显示访存不连续比访存连续慢了近 4 倍，当 block 数量小于某一阈值（96）时，全局存储的访存延迟呈增长趋势，达到阈值后访存延迟趋于稳定。

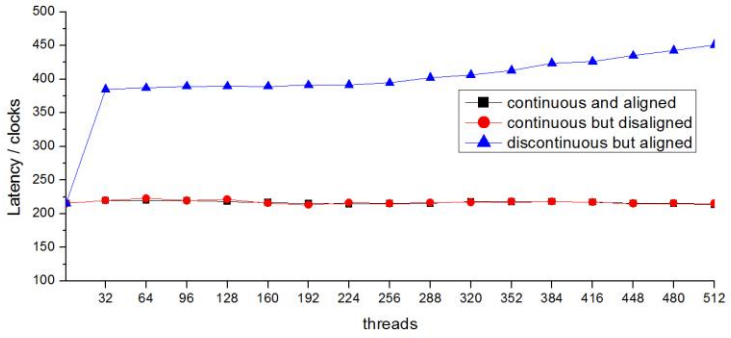
3.5.4 纹理存储并行测评

3.5.4.1 纹理存储的广播与并行访存测评

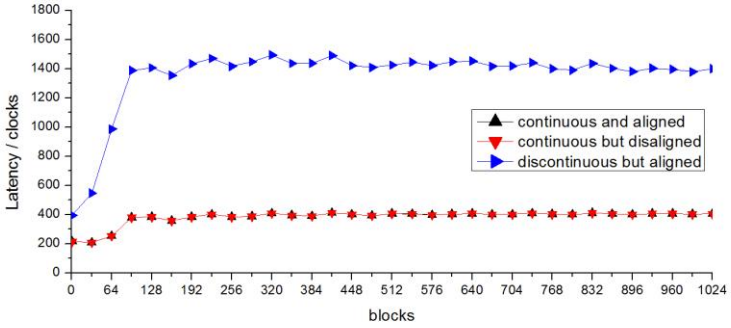
对 K20c GPU 的纹理存储进行广播与并行访存实验，令 warp 中 1、2、4、8、16 和 32 个线程访问相同数据，测试访存延迟。其中 1 个 block 不同 thread 的测试结果显示，不同的线程访问同一数据的访存延迟几乎相等，约为 110 clock。

设定 thread 数量为 256，改变 block 数量时，访存延迟变化情况见图 3.9。图中

不同数量线程访问相同数据的访存延迟几乎相等，且与 thread 级访存延迟相当，说明纹理存储同时支持广播和并行访存。图中数据显示，当 block 数量小于 64 时，纹理存储的访存延迟较小，约为 110 个时钟周期；block 数量从 64 到 128 时，访存延迟增大；超过 128 后访存延迟趋于稳定。



（a）thread 变化对访存延迟的影响



（b）block 变化对访存延迟的影响

图 3.8 全局存储的对齐与连续访存延迟

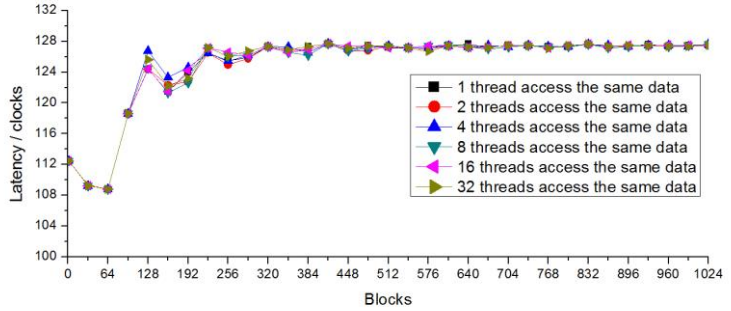
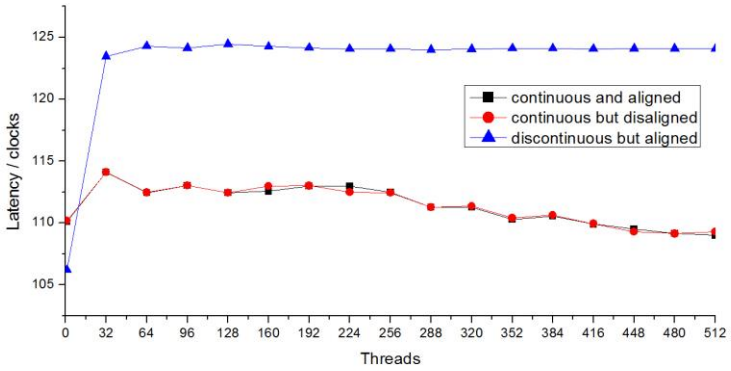


图 3.9 纹理存储的广播特性

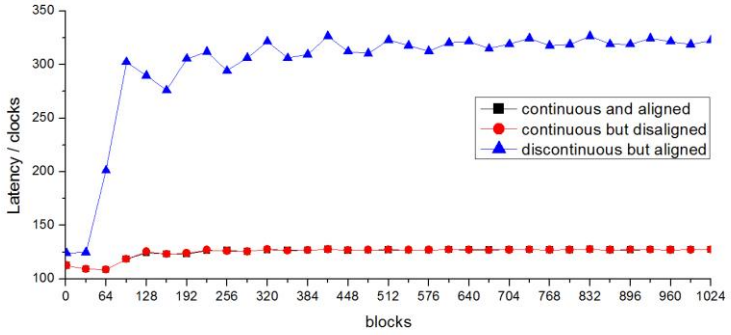
3.5.4.2 纹理存储的对齐与连续访存探索

本节进行纹理存储的对齐与连续访存实验，实验结果见图 3.10。图 3.10 （a）

数据显示，设置 block 数量为 1，改变 thread 数量，当数据连续时，对齐访问与乱序访问两种情况访存延迟几乎完全相等，故对齐访问与否不影响纹理存储访存性能。当数据不连续时，访存延迟明显比连续时大，说明访存不连续会影响纹理存储的访存性能。但是两者差值（~14clocks）相对其访存延迟（~110clocks）较小，说明同一 block 中，连续存储与否对纹理存储性能的影响有限。



（a）



（b）

图 3.10 纹理存储的对齐与连续访存延迟

图 3.10（b）显示，设置线程数量为 256，改变 block 数量，连续存储情况下，访问对齐和访问不对齐的访存延迟基本一致，说明访问对齐与否不影响纹理访问性能。而不连续访问和连续访问的访存延迟差值较大，访存延迟受限于 block 数量，当 block 数量小于 32 时，访存延迟基本不变；大于 32 小于 96 时，访存延迟急剧增加；大于 96 后访存延迟逐步稳定在 320 clocks 左右。Block 数量会影响纹理存储的访存延迟，其中对不连续存储的访存性能影响更大，故优化时需要考虑 block 数量配置。

3.5.5 warp 级存储测评总结

前文用广播与并行访存实验、对齐与连续访存实验分别测试了共享存储、常量

存储、全局存储和纹理存储的 warp 级访存延迟，并根据实验结果分析得出一些存储单元特性，见表 3.2。由表可知，K20c GPU 的存储单元均支持广播访存；除了常量存储不支持并行访存外，共享存储、全局存储和纹理存储均可支持并行访存。在访问各存储单元时，对齐访问与否不会影响访存性能；对于共享存储而言，数据存储连续与否不影响访存性能，数据不连续存储对全局存储的访存性能影响较大，对纹理存储的访存性能影响较小。

表 3.2 GPU 存储单元 warp 级测评结果

	广播	并行访存	对齐访存	连续访存
共享存储	支持	支持	无影响	无影响
常量存储	支持	不支持	N/A	N/A
全局存储	支持	支持	无影响	影响较大
纹理存储	支持	支持	无影响	影响较小

3.6 一些其他 GPU 存储优化研究

3.6.1 寄存器与局部存储的分配策略

3.6.1.1 寄存器与局部存储转换的理论基础

从功能上看，寄存器和局部存储的功能是一致的，局部存储作为寄存器的一种补充和扩展，主要用于寄存器数量不足（数组乘加等运算），或寄存器无法处理的情况，比如排序或指针追逐（p=array[p]）这类访问模式。功能上的一致性提供了两者相互转换的基础。

3.3 节通过实验测定了寄存器和局部存储的访存延迟：寄存器位于片上，访存延迟几乎可以忽略不计；而局部存储的物理空间位于显存，访存延迟与全局存储相当（约 203 clocks）。这种巨大的访存延迟差距为用寄存器替换局部存储提供了巨大的效益基础。

3.6.1.2 局部数组的分配探索

GPU 线程的私有变量包括单独的变量和数组，单独的变量一般声明在寄存器中，可不必考虑；而数组可能分配到寄存器或全局存储，NVIDIA 官方未公布具体的私有数组分配策略。本节试图探索影响数组分配位置的因素，笔者猜测数组的访问模式、数组大小以及 GPU 计算能力是影响数组在寄存器和局部存储上分配的重要因素。

为了探索访问模式、数组大小和 GPU 计算能力对数组在局部存储和寄存器分配的影响，设计了一组对比试验：从简单到复杂的 3 种常见访存模式，在实验中修改数组大小，通过编译开关指定 GPU 计算能力。设计 3 种常见访存模式（详细代码参见成果[15]）：（1）数组加法运算。数组加法是最简单的数组运算之一，在线程内声明一个数组，用该数组进行数组加法运算，整个计算过程仅有一层循

环。（2）矩阵乘法。矩阵乘法比数组加法复杂，需要三层循环来完成运算。（3）数组排序。与数组加法和矩阵乘法运算相比，数组排序涉及数组元素的交换，访存复杂度有了进一步提升。

通过实验分别测试上述三种访存模式在不同计算能力、不同数组大小时，数组声明在寄存器和局部存储的情况。表 3.3 统计了声明在寄存器的最大数组尺寸。从表中数据显示：访问模式越简单，数组能分配在寄存器上的尺寸越大；新的 GPU 计算能力对局部存储的优化更明显，能分配在寄存器上的数组明显增大；类似排序这种复杂的访存模式无法在寄存器上分配数组。

表 3.3 不同访存模式和计算能力的寄存器声明数组尺寸

access pattern	sm_10	sm_35
vector add	29	73
matrix multiplication	2*2	6*6
vector sort	0	0

3.6.2 bank conflict 及其避免

共享存储由交替排列的存储片（bank）组成，每个存储片尺寸为 4 字节，Fermi 架构开始 SM 拥有 32 组 bank，不同存储片组的数据可以并行访问。warp 访问共享存储数据时，不同线程访问同一存储片组的不同数据时发生 bank conflict，此时相同存储片组的数据串行访问。

本文设计了一组对比实验，对比了一个 warp 访问的数据位于同一组 bank 的数量对访存延迟的影响，当同一 bank 组中被访问的数据数量分别为 1、2、4、8、16 和 32 时，通过设置宏定义 conflictnum 值来控制访问的 bank 组中数据的数量，q 是相应的共享存储数组访问索引，计算公式如下。

```
#define conflictnum 1      //2,4,8,16,32
int q=(threadIdx.x%conflictnum)*32+ (threadIdx.x/conflictnum);
```

在 NVIDIA Tesla K20c GPU 上运行上述对比实验，结果见图 3.11。图中数据显示，访存延迟和 bank conflict 数量正相关。因此在 GPU 编程优化时，必须避免 bank conflict。

数据偏移是避免 bank conflict 的有效手段之一。一般情况下，同时访问二维共享存储数组的列数据将导致 bank conflict，因此可在二维数组每行后加一个空数据实现数据偏移，使得同时访问的列数据不再位于同一个 bank 组（图 3.12）。普遍的做法是将[32][32]的数组定义为[32][33]的数组。矩阵转置和 brentkung 方法的前缀求和均可利用数据偏移的方法来避免 bank conflict，获得性能收益（详见成果[15]）；在本文 5.5.1 节对比了矩阵转置和数据偏移两类避免 bank conflict 方法在具体应用热点优化中的性能收益。

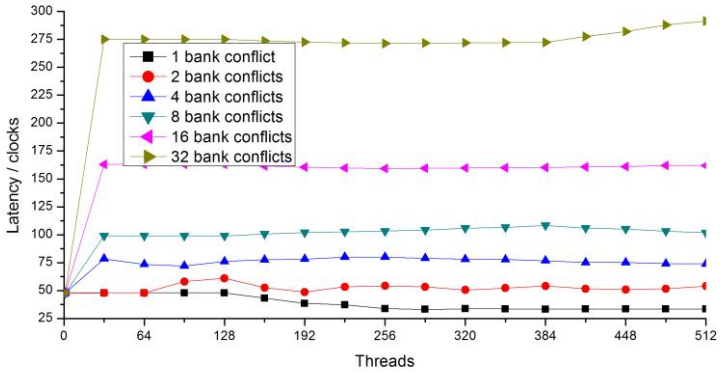


图 3.11 bank conflict 对延迟的影响

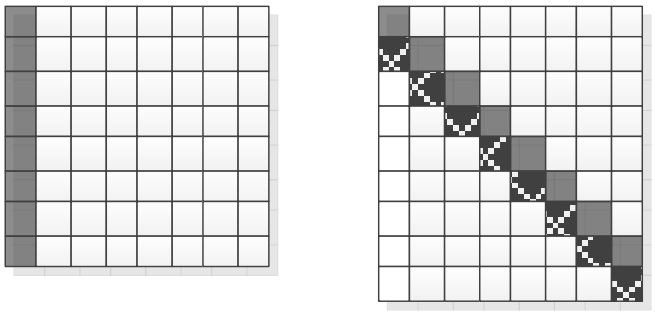


图 3.12 数据偏移示意图

3.6.3 全局存储访存带宽探索

全局存储访存带宽是描述全局存储访存性能的重要指标，也是在实际应用中限制 GPU 程序运行性能的关键因素。本节通过比较不同数据类型获得的最佳访存带宽和 cudaMemcpy()函数的访存带宽，探索发挥全局存储带宽的最佳方式。

本文设计一组实验，分别采用 float、double、int、char 和 char4 等常用的数据类型，通过大量不同维度的 grid 网格（包括 1D 和 2D 维度）测试，获得 kernel 函数访存带宽最大值，并与 cudaMemcpy()函数从设备端到设备端（D2D）拷贝的带宽对比，详细实验结果数据见表 3.4。

表 3.4 主流数据类型的全局存储访存带宽

data type	grid		memory bandwidth/GB/s	
	blocks	threads	kernel	Memcpy(D2D)
float	(64,64)	(32,32)	151.492	159.744
double	(1024,1024)	(16,16)	157.146	158.989
int	(64,64)	(32,32)	151.378	159.757
char	1024	1024	63.02	159.337
char4	(64,64)	(32,32)	148.599	159.286

表中 cudaMemcpy()函数 D2D 的访存带宽数据显示，NVIDIA Tesla K20c GPU 的全局存储访存带宽约为 160GB/s。float、int、char4 等 4 字节数据类型和 8 字节

数据类型 double 都能在合理的 grid 配置下获得接近峰值的访存带宽，其中 8 字节数据类型（double）的访存带宽更接近峰值带宽，而组合数据（char4）会有部分带宽损失。char 数据类型的最佳访存带宽（63 GB/s）远低于峰值访存带宽，而将 4 个 char 组合成 char4 类型后获得的访存带宽可提升到 148 GB/s，说明这类 4 字节合并访问是提高访存带宽的重要手段。

3.7 GPU 访存优化策略

根据前文的 GPU 存储单元测评数据和分析，可以导出以下优化策略。

3.7.1 寄存器优化探讨

根据表 3.1 和 3.6.1 节对私有数组在寄存器和局部存储的分配策略探讨，可以得出以下优化策略和建议：

（R_1）针对简单的访问模式，尽量减少私有变量数组的尺寸，使其小于分配到寄存器的临界值，以达到局部存储到寄存器转换的目的。

（R_2）针对较为复杂的访问模式，尽量简化运算过程，比如合并多层循环。

（R_3）针对排序这类无法分配到寄存器上的访问模式，寻找能够替代该运算过程的简单运算，即可实现寄存器替代局部存储的访问优化。

（R_4）要充分利用 GPU 最新的计算能力，可以在不修改代码的前提下实现局部存储到寄存器的转换。

（R_5）由表 3.1 可知，寄存器访存延迟最小，故可用其替代其他存储单元的访问，比如全局存储、共享存储等。

3.7.2 共享存储优化探讨

结合 3.5.1 节实验结果和 3.6.2 节关于 bank conflict 的讨论，设计得到下列共享存储优化策略：

（S_1）共享存储使用时必须避免 bank conflict，数据偏移是常用的手段。

（S_2）共享存储同时支持广播和并行访存。

（S_3）线程是否对齐和存储是否连续都不影响共享存储的访问效率，故可不必考虑。

（S_4）图 3.4 和图 3.5 显示，线程数量增加到某阈值后，共享存储访存延迟下降，因此可以通过优化线程配置来提升程序性能。

（S_5）共享存储的访存延迟远小于全局存储的访存延迟，故用共享存储替换全局存储是有效的优化策略。

（S_6）共享存储的数据来自全局存储或中间计算结果，并会被再次复用，最大化数据复用是重要的优化策略。

（S_7）少量多次循环使用共享存储。由于 SM 中共享存储资源有限，为了共

享存储使用不影响 SM 中活跃的 warp 数量，笔者在成果[15]提出少量多次循环使用的共享存储使用策略，并实例验证了其效果。

3.7.3 常量存储的优化探讨

根据图 3.1 和 3.5.2 节实验结果，针对常量存储访存特性可以总结出以下几个常量存储优化策略：

（C_1）合理构造数据结构，令连续访问的常量存储数据尽量邻近。

（C_2）常量存储支持广播，广播能直接减少常量存储访存延迟，合理利用常量存储的广播机制是重要的优化方法。

（C_3）常量存储不支持并行访存，这类多个线程同时访问不同数据的访存模式不适合用常量存储优化。

（C_4）合理设计 block 内 threads 配置。在图 3.6（a）中，线程数量达到某一阈值前，常量存储访存延迟基本不变，而超过该阈值访存延迟增加；在图 3.6（b）中，线程数量直接影响常量存储的访存延迟。因此合理配置 block 内 threads 数量，要合理平衡占用率和线程对常量存储访存延迟影响的阈值。

（C_5）block 数量影响常量存储访存延迟的阈值较小，且超过阈值后访存延迟基本恒定，故 block 数量可不必考虑。

（C_6）常量存储访存延迟是全局存储的一半左右，故在合适的条件下，可用常量存储替换全局存储进行优化。

3.7.4 全局存储优化探讨

结合 3.5.3 节全局存储 warp 级测评结果和 3.6.3 节不同数据类型的全局存储访存带宽，设计得到以下 GPU 全局存储优化策略：

（G_1）GPU 全局存储支持广播与并行访存，其中广播性能稍好。

（G_2）当数据类型尺寸小于 4 字节时，构造 4 字节或 8 字节的合并访问可以有效提高全局存储带宽利用率。

（G_3）对齐访问与否不影响全局存储访问性能。不连续存储将导致巨大的访存开销，故在构造数据存储时，同一 warp 访问的存储应尽量连续。

（G_4）当访存不连续时，线程配置会影响访存延迟，需要合理设计；而线程块数量对访存延迟没有影响，可自由配置。

3.7.5 纹理存储的优化探讨

通过 3.5.4 节纹理存储测评，可以总结出以下优化策略：

（T_1）纹理存储提供了良好的广播和并行访存机制，合理利用纹理存储的广播和并行访存可以有效提升程序性能；

（T_2）纹理存储访存延迟与访问对齐与否都无关。不连续存储将导致巨大的

纹理存储访问开销，因此在构造数据结构时应尽量让邻近线程访问的数据连续。

（T_3）thread 数量和 block 数量对纹理存储的访存延迟均有影响，但阈值较小，且超过阈值后访存延迟基本不变，故可不必考虑 thread 和 block 配置优化。

（T_4）不连续存储对纹理存储的访存性能影像较小，当邻近线程访问的存储不连续时，可用纹理存储替代全局存储。

3.7.6 GPU 访存优化框架

结合 GPU 各存储单元的 warp 级测评和分析、私有变量数组在寄存器和局部存储中的分配探索、共享存储的 bank conflict 讨论、不同数据类型的全局存储访存带宽、以及设计的优化策略，提出一个 GPU 访存优化框架（图 3.13）。

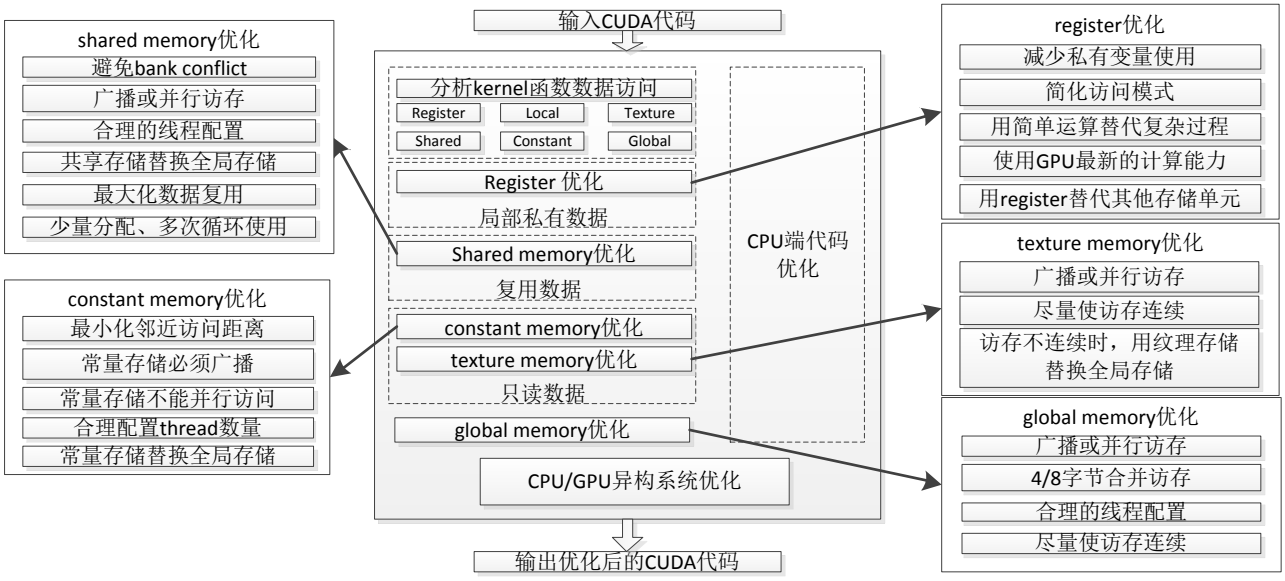


图 3.13 GPU 访存优化框架

该框架以 CUDA 代码为输入，对 kernel 函数进行访存分析，搜索各存储单元使用情况、数据访问类型（中间变量、重用数据、只读数据等）和特殊操作，利用前文设计的访存优化方法对各类存储单元和数据进行优化；同时对 CPU 端代码进行优化；在上述优化基础上基于 CPU/GPU 异构系统进行异构优化，包括主机端内存类型选择、zerocopy 优化、计算和通信重叠、计算和计算重叠等（详见第四章）；框架最终输出优化后的 CUDA 代码。

3.8 优化实例与效果展示

在第五章的高光谱影像降维和第六章的声呐信号波束形成中，阐述了大量的热点性能优化实例，采用本章设计的优化策略，在 K20c GPU 上取得了理想的加速效果。下面简单总述具体热点、采取的优化策略和获得的性能提升。

（1）在 5.5.1 节的协方差矩阵计算性能优化中，采取了 G_3、S_5、S_4+S_6、S_1 等优化策略，与未优化的并行版本相比，获得了 69 倍加速比。

（2）在 5.5.2 节的 PCA 变换性能优化研究中，采取了 C_6 和 S_5，在并行基础上加速了 2 倍左右。

（3）在 5.5.3 节的 ICA 迭代性能优化中，采取了 R_5、S_5、G_3、kernel 函数合并和zerocopy优化等优化策略，最终在并行基础上获得1.8倍左右的加速效果。

（4）在 5.5.4 节的噪声估计性能优化研究中，采取 R_3、S_5、S_4+S_6 等优化策略后，性能可提升 6.9 倍左右。

（5）在 6.4.3.2 节的 CBF 计算优化中，采取了 S_5、R_5、G_3（重构 DFT 结果矩阵）和 G_3（重构声程差矩阵）等优化策略，在并行基础上获得 3 倍左右的加速效果。

（6）在 6.5.3.1 节的雅克比迭代优化中，采取了 R_5 和 G_3，加速近 1.18 倍。

（7）在 6.5.3.2 节的方位谱计算优化中，处理较小规模水听器阵列（小于 1024 基元）信号时，采用了 G_3（特征向量矩阵）、S_5（优化中间变量）、S_5+S_6（优化方向向量）等优化策略，取得了 9.7 倍加速比；在处理较大规模水听器阵列（大于 1024 基元）信号时，在采取上述三种优化策略的基础上，还采取了去除方向向量的共享存储优化以释放 SM 中活跃的 warp 数量、G_3（方向向量）、S_7 等优化策略，最终在较小规模的优化基础上加速了 1.3 倍左右。

3.9 本章小结

本章测定了 GPU 各存储单元的 thread 级访存延迟；提出基于 GPU 存储的 warp 级访存延迟的并行测评方法，并设计两组 warp 级并行测评实验；对共享存储、常量存储、全局存储和纹理存储分别进行 warp 级并行测评和分析；论文就私有临时变量数组在寄存器和局部存储的分配进行探索，探讨了共享存储的 bank conflict 及不同数据类型的全局存储访存带宽；并根据实验分析设计相应的优化策略和建议，构建 GPU 访存优化框架；总结了第五章高光谱影像降维算法和第六章声呐信号波束形成算法中性能优化所采取的本章设计的优化策略，并阐明了优化后获得的性能提升，验证了框架的实用性和有效性。

本章提出的 warp 级并行测评及 GPU 各存储单元的测评结果，可以为开发者设计 GPU 专用算法提供参考；本章导出的访存优化策略可以帮助编程人员优化现有 GPU 程序；此外，本章研究内容还可为 GPU 性能建模提供借鉴。

第四章 分段式的主机端存储选择模型

上一章探讨了 GPU 的片上存储系统，本章则主要围绕 CPU/GPU 异构系统的协同优化（端对端访存）开展研究。

本章重点关注主机端存储的选择问题，通过 microbenchmark 测评和建模，提出分段式的主机端存储选择模型。此外，利用 zerocopy 技术设计了两项优化方案并进行了实验验证；实例研究了常用的计算与通信重叠、（CPU）计算与（GPU）计算重叠两种异构协同优化方法。

本章研究成果能有效助力于 CPU/GPU 异构程序优化和异构系统性能建模。比如利用主机端存储选择模型来指导 PCA 降维等并行算法的主机端存储类型选择，以收获最佳性能；比如利用 zerocopy 技术优化第五章的 ICA 迭代过程中的少量 CPU/GPU 数据通信，减少全局存储访问，获得性能提升（详见 5.5.3 节）；比如利用计算与计算重叠优化第五章的 MNF 降维算法等。

4.1 引言

在 CPU/GPU 异构平台上，CPU 和 GPU 的存储空间是独立且不可直接访问的。在 GPU 执行 kernel 函数前，需要将数据从 CPU 端传输到 GPU 端。其中主机端存储包含可分页存储和页锁定存储两大类，关于主机端存储的正式公开资料较少[9,45-46]。一方面，书籍和文档资料中普遍提到，页锁定存储能提供比可分页存储更高的通信带宽，但在经典矩阵乘法实验（表 4.1）中，可分页存储获得了更好的性能，故需要一个合适的主机端存储选择模型来有效提升异构程序性能。另一方面，一些研究者试图利用简单模型来匹配数据通信带宽[56-58]。根据图 4.2 结果，很难用一个简单的模型来匹配整个通信带宽曲线。

为更好地分配主机端存储和对通信带宽建模，利用微 benchmark 来提取主机端存储的相关特征，得知页锁定存储的分配/释放和注册/解除注册的开销巨大；继而研究主机端存储的访存带宽、PCI-Express 带宽、注册及解除注册时间的性能模型，构建主机端存储选择模型；利用高光谱影像 PCA 降维算法为例，比较模型预测时间与真实运行耗时，结果证明存储选择模型是正确的、可行的。

本章贡献包括：（1）对比了可分页存储和页锁定存储的各项开销，提取主机端存储分配特征；（2）对主机端存储的访存带宽、PCI-E 带宽、页锁定存储的注册及解除注册时间进行建模，并提出分段的主机端存储选择模型；（3）以高光谱遥感影像 PCA 降维算法为例，对比模型预测时间与真实运行耗时，验证存储选择模型的正确性；（4）利用 Zerocopy 技术设计两项优化方案并进行了实验验证；（5）

实例研究了计算与通信重叠、（CPU）计算与（GPU）计算重叠两类常见的异构协同优化方法。

4.2 相关工作

本节从数据通信带宽建模和 CPU/GPU 异构协同优化两个方面展开相关工作研究。

Werkhoven 等[56]对 CPU-GPU 数据传输进行性能建模，分别对计算与通信重叠性能和 PCI-E 带宽进行建模，其中 PCI-E 带宽建模采用 LogGP 模型。Michael 等[57]提出一个能同时预测 kernel 执行时间和数据通信时间的 GPU 性能线性建模框架。Mitesh 等[58]对 CPU 与 FPGA/GPU 间的数据传输带宽进行建模。上述研究利用一个比较简单的模型（如 LogGP、线性模型等）来匹配 PCI-E 通信带宽，难以覆盖全部的带宽曲线。本文提出分段 PCI-E 带宽模型，能较好的契合带宽曲线，其中仅在通信数据量少于 2KB/4KB 的情况下，PCI-E 带宽模型才与 LogGP 和线性模型匹配。此外，本文对页锁定存储的注册/解除注册时间进行建模，并从整体角度进行对比选择合适的主机端存储。

CPU/GPU 异构协同优化在很多实际应用中被广泛采用。Ta 等[59]利用 CPU/GPU 异构系统加速 3 维地球模拟系统 DynEarthSol3D，其中采用的异构协同优化是减少数据通信。Fang 等[60]利用 CPU/GPU 异构系统接近实时处理 ZY-3 卫星图像，提出 CPU/GPU 的负载分配策略。Wu 等[61]在 CPU/GPU 异构平台上开发适用于大规模数据的 FFT，设计了一种分解策略并将计算分配到 CPU 和 GPU 上，减少 CPU 和 GPU 间数据传输次数，并利用 kernel 函数执行和通信重叠来掩盖通信开销。总体而言，上述研究主要关注常见的异构协同优化方法，如减少通信量、计算与通信重叠，暂时还未涉及主机端存储选择。相反的，本文更关注主机端存储选择，从整体上提升 CPU/GPU 异构程序性能。

4.3 主机端存储及问题提出

4.3.1 主机端存储类型

CPU/GPU 异构程序开发时，主机端存储类型包括可分页存储和页锁定存储，其中可分页存储的存储页可被切换至磁盘，而页锁定存储的存储页一直保留在内存中。在 CUDA 中，可分页存储利用 malloc()和 free()函数分配和释放空间，该存储方式不支持异步通信，亦不支持 RDMA 通信。页锁定存储可利用 cudaHostAlloc()和 cudaFreeHost()函数分配和释放空间，或者利用 cudaHostRegister()函数和 cudaHostUnregister()函数将可分页存储注册为页锁定存储和解除注册。页锁定存储

支持 RDMA 访问，支持 CPU 与 GPU 的异步通信，能实现计算与通信重叠，但由于内存空间有限，若页锁定内存分配过多将导致内存空间不足。

4.3.2 从矩阵乘法提出问题

在 CUDA 书籍和 NVIDIA 文档的描述中，页锁定存储的 PCI-E 通信带宽是可分页存储的两倍左右。故理论上，若将可分页存储替换为页锁定存储，可以提高 CPU 与 GPU 间的通信带宽，进而提高程序整体性能。

在经典的 GPU 矩阵乘法程序中，有矩阵空间分配（主机端和设备端）、矩阵初始化、矩阵传输到 GPU、GPU 进行矩阵乘法运算、结果矩阵传输到主机端和释放空间（主机端和设备端）等步骤。分别实现可分页存储和页锁定存储两种方式的矩阵乘法，其中 GPU 矩阵乘法调用 CUBLAS 库 cublasSgemm 函数实现，在 K20c GPU 平台测试两者总时间（见表 4.1），统计时已排除 CUBLAS 库的首次启动开销和 GPU 的首次启动开销。

表 4.1 可分页和页锁定的矩阵乘法耗时/ms

n	pagable	pinned
512	40.38	41.36
1024	156.70	159.35
2048	622.51	636.28
4096	2514.18	2570.30

表中数据显示，页锁定存储的矩阵乘法耗时比可分页存储矩阵乘法耗时长，这与理论推测明显不相符合。实验结果与理论推测不符合的原因是什么？是 GPU 书籍描述错误，还是存在未知的不可控开销？带着上述问题，深入探索页锁定存储和可分页存储的性能特性，试图提出一个分段的主机端存储类型选择模型。

4.4 初探主机端存储

上一节矩阵乘法实验中，页锁定存储版本耗时比可分页存储版本耗时大，那么两者的性能差异到底出在哪儿呢？本节设计了 3 组对比实验来探索可分页存储和页锁定存储的各项开销特性：（a）可分页存储，用 malloc()函数和 free()函数分配和释放内存；（b）页锁定存储，用 cudaHostAlloc()函数和 cudaFreeHost()函数进行内存分配和释放；（c）页锁定存储，用 malloc()函数和 free()函数分配和释放可分页存储，并用 cudaHostRegister()函数将可分页存储注册为页锁定存储，用 cudaHostUnregister()函数解除注册页锁定存储。

经测试发现，malloc()函数执行时没有直接分配存储空间，而是在使用空间时分配。为了测得准确的存储分配时间，在 malloc()函数后加入两个 memset()函数，第 1 个 memset()函数的作用是强制分配空间，第 2 个 memset()函数的作用是测得

单独的 memset()时间并减去两倍的该时间，即为存储分配时间。

设定数据量为 1GB 时，在 K20c GPU 上测试存储分配、注册、主机端存储访问（数据拷贝）、主机端到设备端传输、设备端到主机端传输、解除注册、存储释放和总时间等耗时信息，见表 4.2。表中数据显示：（1）页锁定存储的 PCI-Express 通信带宽比可分页存储高；（2）方案 b 中，cudaHostAlloc()函数和 cudaFreeHost()函数的存储分配和释放非常耗时，若仅有少数次数的数据传输，则传输收益显然无法弥补分配和释放开销；（3）从总时间上看，利用 cudaHostRegister()函数注册的页锁定存储（c 方案）是一个比较好的选择，这种方式的页锁定存储注册时间和解除注册时间较少，线性关系下传输收益大于注册和解除注册开销，总耗时最小。

表 4.2 主机端存储使用时的详细耗时/ms

1GB	malloc	register	cpu_access	H2D	D2H	unregister	free	all_time
a	170.63	0.00	199.44	359.71	335.06	0.00	1.99	1066.82
b	469.98	0.00	193.12	169.67	160.41	0.00	165.10	1158.28
c	170.17	26.65	199.77	169.96	160.31	45.90	1.69	774.44

4.5 主机端存储选择模型

4.4 节实验结果展示了主机端存储的各项开销情况，但不同数据规模和不同访存模式又各有不同，需要进一步展开研究。而由上一节结果可知，使用 cudaHostAlloc()方式的页锁定存储必然比使用 cudaHostRegister()方式耗时长，因此后文的页锁定存储只讨论性能更优的 cudaHostRegister()方式。

可分页存储和页锁定存储均需要分配和释放可分页存储空间，两者耗时相等，可不必考虑。此外需要考虑的比较内容包括可分页存储和页锁定存储的 CPU 访存带宽、两者的 PCI-E 通信带宽、页锁定存储的注册和解除注册耗时，下面分别展开研究。

4.5.1 主机端存储的访存带宽

4.4 节结果显示，1GB 数据拷贝时，可分页存储和页锁定存储的访问性能相当。当数据尺寸和访存模式不同时，两者是否仍然相当呢？本节设计了 3 类主机端存储访问模式：（1） 写操作，对可分页存储或页锁定存储写入相同的数值（比如写入 1.0）。（2） 读操作，对可分页存储或页锁定存储进行向量内积运算，该运算读取数据并累加，主要的访存是读操作。（3） 数据拷贝，分别进行了 4 次数据拷贝操作，包括可分页存储到可分页存储（pa2pa）、可分页存储到页锁定存储(pa2pi)、页锁定存储到页锁定存储(pi2pi)和页锁定存储到可分页存储(pi2pa)。分别测试单线程和多线程访问上述 3 种模式的访存带宽，见表 4.3 和表 4.4。

表 4.3 数据显示，单线程访问主机端存储时，当数据尺寸小于 8MB 时，由于 cache 的存在，访存带宽较大。数据尺寸超过 8MB 时，可分页存储和页锁定存储的访存性能基本相当。而当数据规模达到 2GB 时，可分页存储到可分页存储、页锁定存储到页锁定存储的数据复制带宽开始下降。当数据规模达到 4GB 时，页锁定存储的读操作和写操作带宽开始下降，且可分页存储到可分页存储、可分页存储到页锁定存储的数据复制带宽下降。

表 4.3 主机端存储单线程访存带宽/GB/s

data size/MB	pagable		pinned		pa2pa	pa2pi	pi2pi	pi2pa
	writen	read	writen	read				
4	16.48	12.81	16.55	12.85	23.26	23.31	22.72	23.04
8	15.44	12.26	15.11	12.36	11.48	12.33	11.89	12.29
16	7.84	8.35	8.07	8.47	7.87	7.62	7.56	7.95
32	7.75	8.36	7.62	8.43	6.43	6.61	6.54	6.59
64	7.64	8.39	7.77	8.40	6.01	6.03	6.02	6.14
128	7.72	8.40	7.73	8.41	5.65	5.67	5.62	5.72
256	7.77	8.41	7.81	8.41	5.66	5.64	5.65	5.73
512	7.81	8.41	7.79	8.41	5.66	5.67	5.66	5.72
1024	7.79	8.41	7.79	8.41	5.66	5.66	5.65	5.72
2048	7.82	8.41	7.99	8.41	4.09	5.65	3.44	5.56
4096	7.90	8.41	5.76	6.72	3.44	3.63	3.36	5.56

表 4.4 主机端存储多线程访存带宽/GB/s

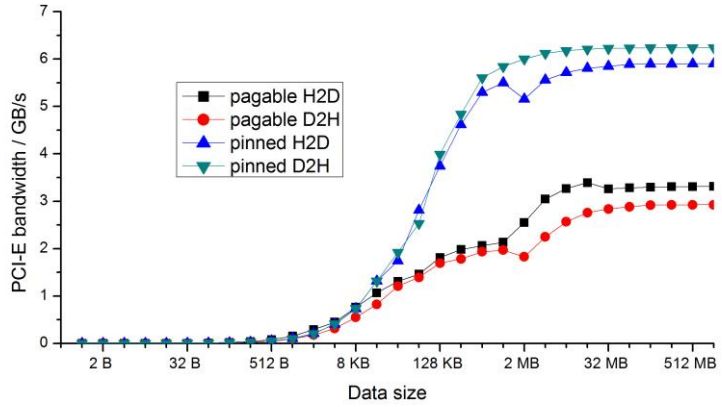
data size/MB	pagable		pinned		pa2pa	pa2pi	pi2pi	pi2pa
	writen	read	writen	read				
4	176.17	156.04	178.09	140.03	287.44	300.62	315.08	312.08
8	211.41	163.02	199.80	169.78	295.21	289.98	306.24	312.08
16	223.67	179.55	214.17	183.57	22.74	19.49	21.58	32.93
32	13.31	41.06	12.92	19.78	24.24	15.64	11.45	16.18
64	11.46	42.15	9.69	18.35	25.22	15.18	10.99	17.13
128	13.84	43.80	9.12	18.85	25.75	15.06	11.43	17.92
256	16.43	44.21	8.76	18.54	28.61	16.30	12.34	19.31
512	17.52	44.67	8.56	18.73	31.83	17.97	13.79	20.42
1024	18.23	44.66	8.55	18.66	33.76	18.74	14.34	21.54
2048	18.97	44.84	8.54	18.88	36.16	19.45	14.99	22.97
4096	19.28	44.85	8.67	17.88	36.15	15.45	14.05	23.31

表 4.4 数据显示，多线程访问主机端存储时，单独的读操作和写操作的 cache 尺寸为 16MB，而数据拷贝适配 cache 大小的数组尺寸为 8MB。从读操作和写操作看，可分页存储的多线程访问带宽远高于页锁定存储。从数据复制的角度看，可分页存储到可分页存储的带宽最大，页锁定存储到可分页存储带宽次之，页锁定存储到页锁定存储带宽最小。

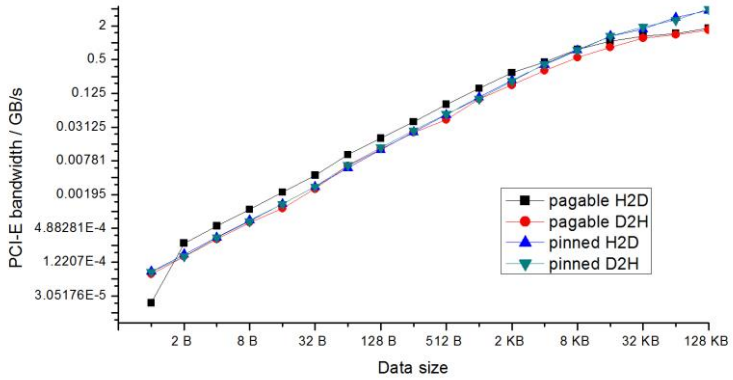
从上述结果和分析中可以得出一个总体结论，即可分页存储的 CPU 访存带宽更佳。

4.5.2 PCI-Express 带宽

设置数据大小从 1B 到 1GB，分别测试页锁定存储和可分页存储的主机端到设备端（H2D）和设备端到主机端（D2H）的通信带宽，见图 4.1。图中数据显示：（1）数据量较小时，H2D 和 D2H、页锁定和可分页的带宽相当；（2）数据量增大后，页锁定存储的通信带宽开始大于可分页存储的带宽，H2D 和 D2H 也出现差异；（3）数据量达到一定规模后，各项带宽趋于稳定，达到最大值。



(a)



(b)

图 4.1 PCI-Express 带宽

文献[56]用 LogGP 模型对 PCI-Express 带宽进行建模。文献[57]用线性模型计算传输时间。显然，LogGP 模型和线性模型均无法完全吻合图 4.1 的带宽曲线，上述 PCI-E 带宽曲线需要进行分段建模。对于每个传输过程，利用 SPSS（Statistical Product and Service Solutions）软件分段拟合 PCI-E 带宽。

仔细观察图 4.1，发现该曲线可大致可分为 3 个阶段（1B~1KB，1KB~1MB，1MB~1GB），其中 1B~1KB 阶段大概服从线性分布（linear），1KB~1MB 阶段有

可能服从对数（logarithmic）、二次多项式（quadratic）或三次多项式（cubic）等分布，1MB~1GB 基本上服从指数分布（exponential）。当然，直接按上述分段进行拟合将无法精确预测所有带宽，特别是数据量较小的情况，此时需要在上述分段的基础上，根据具体情况进行微调，或进一步分段。下面分别对可分页存储的 H2D/D2H 过程和页锁定存储的 H2D/D2H 过程的 PCI-E 带宽进行建模。

4.5.2.1 可分页存储 H2D 带宽模型

利用分段拟合与微调的方法，可以将可分页存储 H2D 的 PCI-E 带宽分为 6 个子部分，分别进行建模（公式 4.1）。其中 1B~2KB 分段服从线性分布，并需分两段（1B~32B，32B~2KB）进行拟合；2KB~256KB 分段服从对数分布；256KB~32MB 分段服从 3 次多项式分布，且需分两段（256KB~4MB 和 4MB~32MB）进行拟合；32MB-1GB 分段服从指数分布。

$$BW_{pagable\_h2d}=\begin{cases}1.365*10^{-4}*S-3.883*10^{-5} & (1B\le S\le 32B)\\ 1.512*10^{-4}*S+1.173*10^{-4} & (32B<S\le 2KB)\\ 0.360*\ln(S)-2.479 & (2KB<S\le 256KB)\\ 1.995-3.838*10^{-8}*S+2.191*10^{-13}*S^2-3.579*10^{-20}*S^3 & (256KB<S\le 4MB)\\ 2.682+1.072*10^{-7}*S-5.043*10^{-15}*S^2+7.050*10^{-23}*S^3 & (4MB<S\le 32MB)\\ e^{\left(1.197-\frac{514828.668}{S}\right)} & (32MB<S\le 1GB)\end{cases} \quad (4.1)$$

图 4.2 展示了各分段的模型曲线和实际观测值，图中观测值与模型曲线基本吻合。计算模型曲线与实测数据的误差率，平均误差率为 4.71%。最大误差率为 76.47%，仅发生在数据量为 1B 的情况，此时误差量级为 10^{-5}，可判断该误差率过大是由于数据量较小时模型基数过小而导致的，故可忽略该误差。

4.5.2.2 可分页存储 D2H 带宽模型

可分页存储 D2H 的 PCI-E 带宽模型见公式 4.2，其中分为 6 段进行建模：1B~2KB 分段符合线性分布，分两段（1B~32B 和 32B~2KB）分别进行建模；2KB~16KB 和 256KB~2MB 两个分段均符合 3 次多项式分布；16KB~256KB 分段符合对数分布；2MB~1GB 分段符合指数分布。

$$BW_{pagable\_d2h}=\begin{cases}7.661*10^{-5}*S-1.020*10^{-5} & (1B\le S\le 32B)\\ 8.703*10^{-5}*S+1.433*10^{-3}e-3 & (32B<S\le 2KB)\\ 1.716*10^{-2}+8.241*10^{-5}*S-2.251*10^{-9}*S^2+1.406*10^{-14}*S^3 & (2KB<S\le 16KB)\\ 0.343*\ln(S)-2.472 & (16KB<S\le 256KB)\\ 1.510+1.326*10^{-6}*S-1.133*10^{-12}*S^2+2.732*10^{-19}*S^3 & (256KB<S\le 2MB)\\ e^{\left(1.073-\frac{1012387.498}{S}\right)} & (2MB<S\le 1GB)\end{cases} \quad (4.2)$$

图 4.3 展示了可分页存储 D2H 的 PCI-E 带宽中各分段的模型曲线和实际观察值，图中数据显示，观察值与模型曲线基本吻合。计算模型曲线与实验数据的误差百分比，其平均误差率为 3.36%。最大误差率为 12.94%，仅发生在数据量为 1B 的情况，该误差率是由于数据量较小时模型基数过小而导致的，故可忽略。

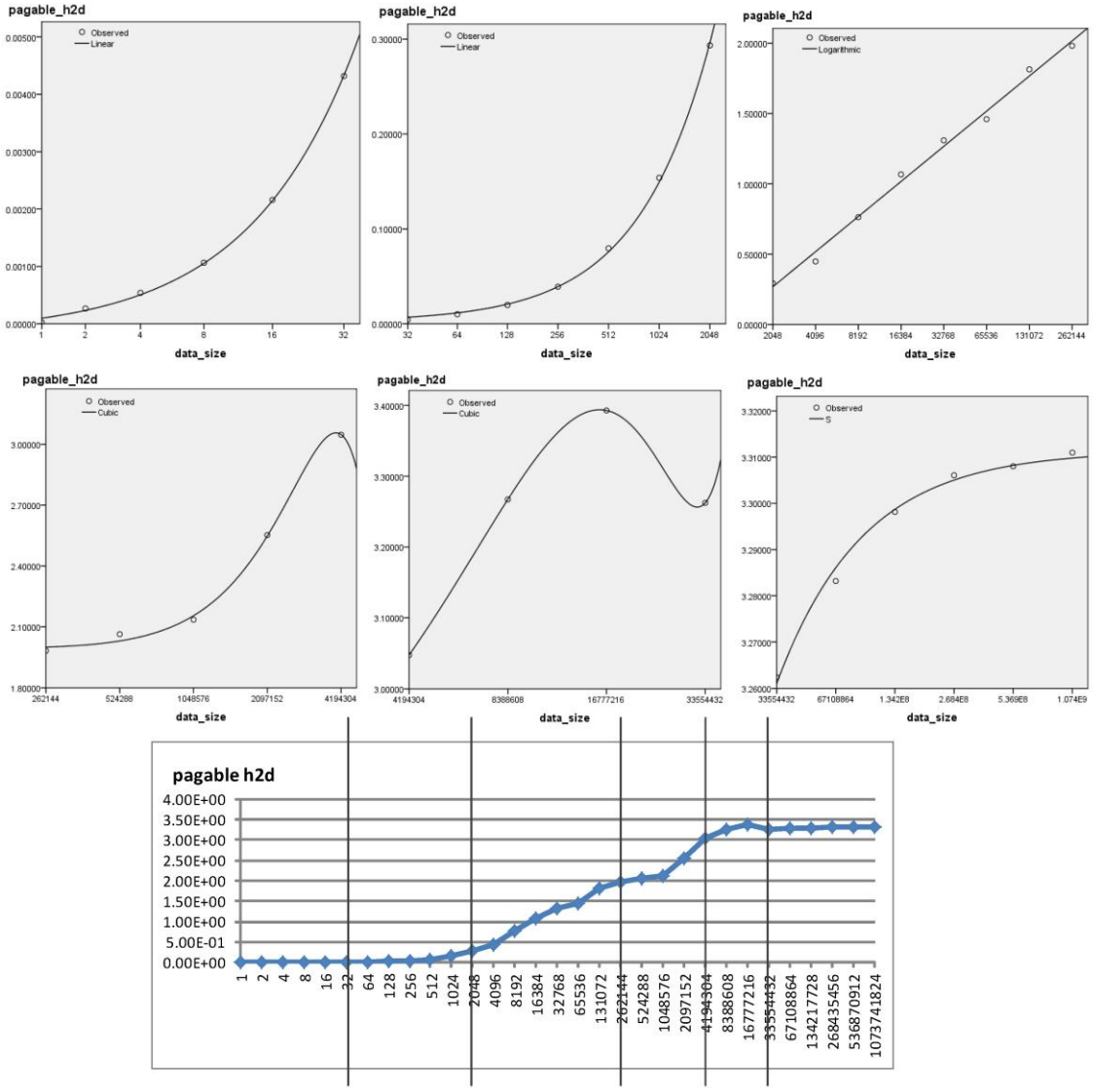


图 4.2 可分页存储 H2D 带宽模型曲线与实测值

4.5.2.3 页锁定存储 H2D 带宽模型

将页锁定存储 H2D 的 PCI-E 带宽分为 5 个分段，建模得到 PCI-E 带宽模型见公式 4.3。其中 1B~2KB 分段符合线性分布，需分两段（1B~32B 和 32B~2KB）进行建模；2KB~2MB 分段符合 3 次多项式分布，需分两段（2KB~256KB 和 256KB~2MB）进行建模；2MB-1GB 分段符合指数分布。

$$BW_{pinned\_h2d} = \begin{cases} 8.462*10^{-5}*S-2.697*10^{-6} & (1B \le S \le 32B) \\ 1.052*10^{-4}*S-1.106*10^{-3} & (32B < S \le 2KB) \\ 0.210+5.970*10^{-5}*S-3.382*10^{-10}*S^2+6.660*10^{-16}*S^3 & (2KB < S \le 256KB) \\ 3.365+6.030*10^{-6}*S-5.147*10^{-12}*S^2+1.278*10^{-18}*S^3 & (256KB < S \le 2MB) \\ e^{(1.776-\frac{278771.750}{S})} & (2MB < S \le 1GB) \end{cases} \quad (4.3)$$

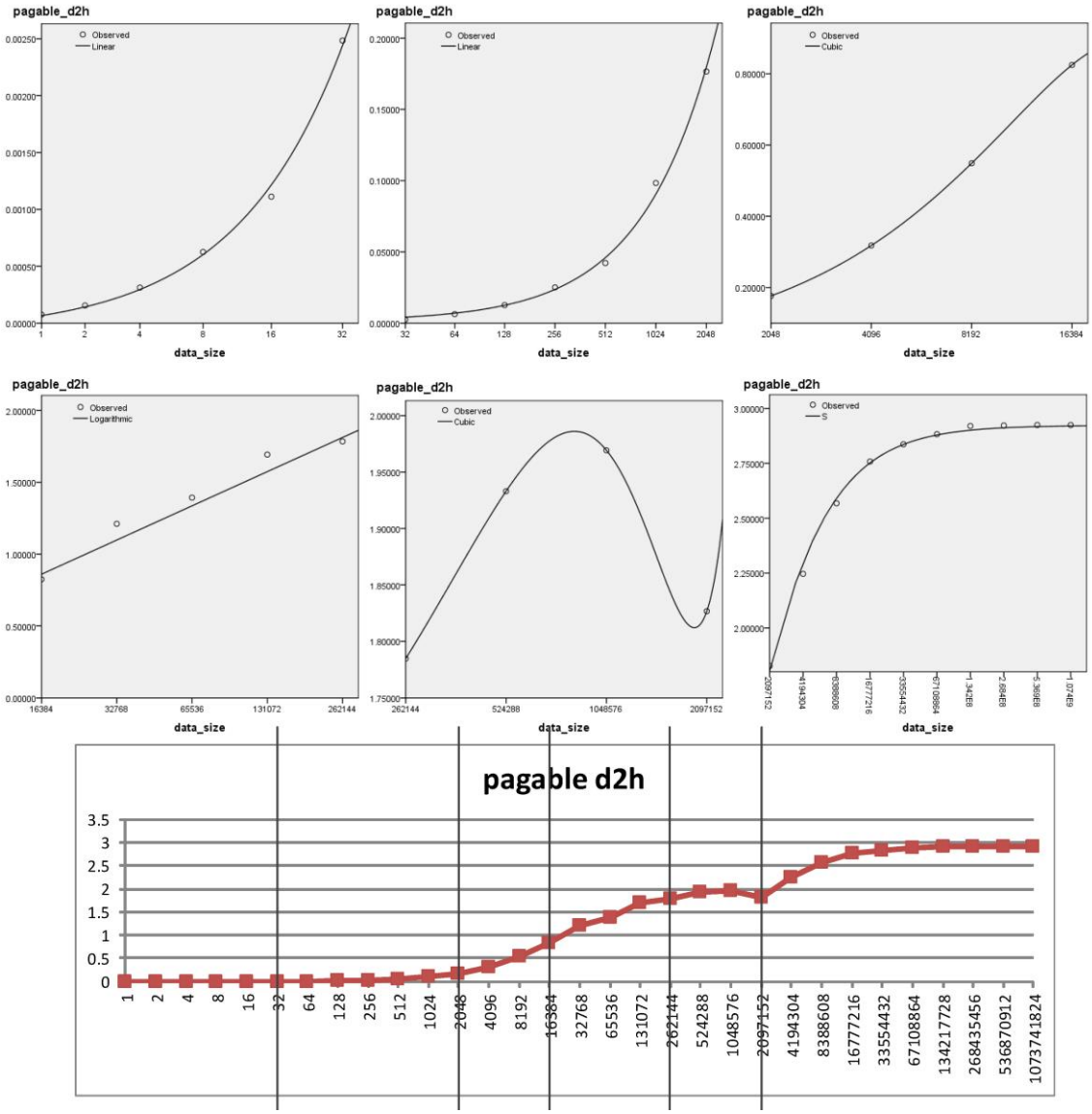


图 4.3 可分页存储 D2H 带宽模型曲线与实测值

图 4.4 展示了页锁定存储 H2D 的 PCI-E 带宽各分段的模型曲线和实际观察值，图中观察值与模式曲线基本吻合。模型曲线与实验数据的平均误差率为 1.82%。最大误差率（19.50%）仅发生在数据量为 16KB 的情况，而其余情况的误差率均较小，可认为这是个特例。

4.5.2.4 页锁定存储 D2H 带宽模型

利用分段拟合与微调的方法，可以将页锁定存储 D2H 的 PCI-E 带宽分为 5 个子部分，分别进行建模（公式 4.4）。其中 1B~4KB 分段符合线性分布，需分两段（1B~32B 和 32B~4KB）进行建模；4KB~256KB 分段符合 3 次多项式分布；256KB~1GB 符合指数分布，需分两段（256KB~8MB 和 8MB~1GB）分别进行建模。

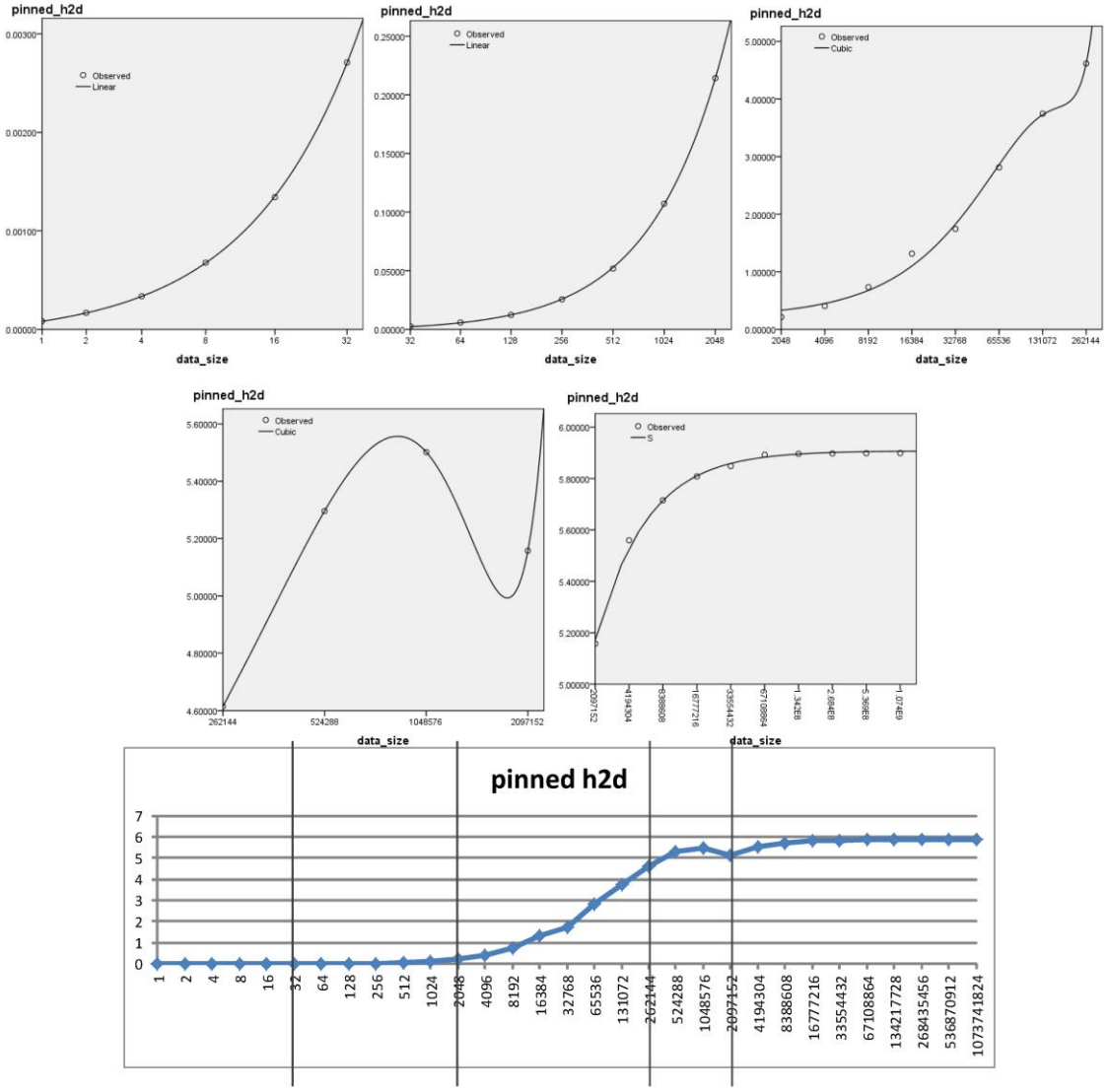


图 4.4 页锁定存储 H2D 带宽模型曲线与实测值

$$BW_{pinned\_d2h}=\begin{cases}8.481*10^{-5}*S-1.332*10^{-5} & (1B\le S\le 32B)\\ 1.011*10^{-4}*S+1.086*10^{-4} & (32B<S\le 4KB)\\ 0.393+4.759*10^{-5}*S-1.974*10^{-10}*S^{2}+3.074*10^{-16}*S^{3} & (4KB<S\le 256KB)\\ e^{\left(1.829-\frac{64448.291}{S}\right)} & (256KB<S\le 8MB)\\ e^{\left(1.831-\frac{83470.782}{S}\right)} & (8MB<S\le 1GB)\end{cases}$$

（4.4）

图 4.5 展示了页锁定存储 D2H 的 PCI-E 带宽各分段的模型曲线和实际观察值，观察值与模型曲线基本吻合。计算模型曲线与实测数据的误差率，平均误差率为 2.54%。最大误差率为 17.35%，且仅发生在数据量为 16KB 的情况，而其余数据量情况的误差率均较小，可以认为是个特例。另外 1B 时的误差率为 14.7%，是由于数据量过小基数太小导致的。

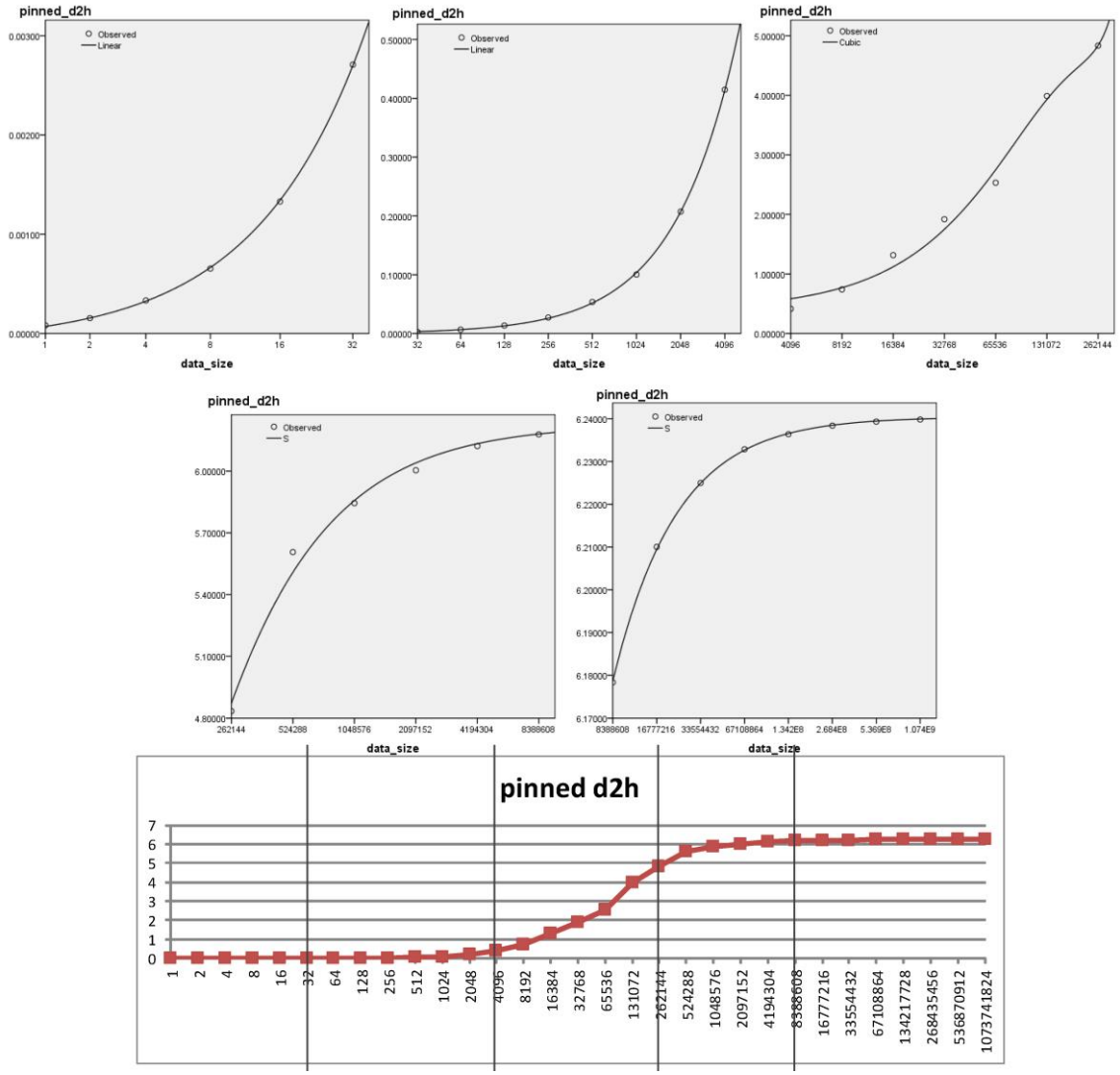


图 4.5 页锁定存储 D2H 带宽模型曲线与实测值

4.5.3 页锁定存储注册与解除注册

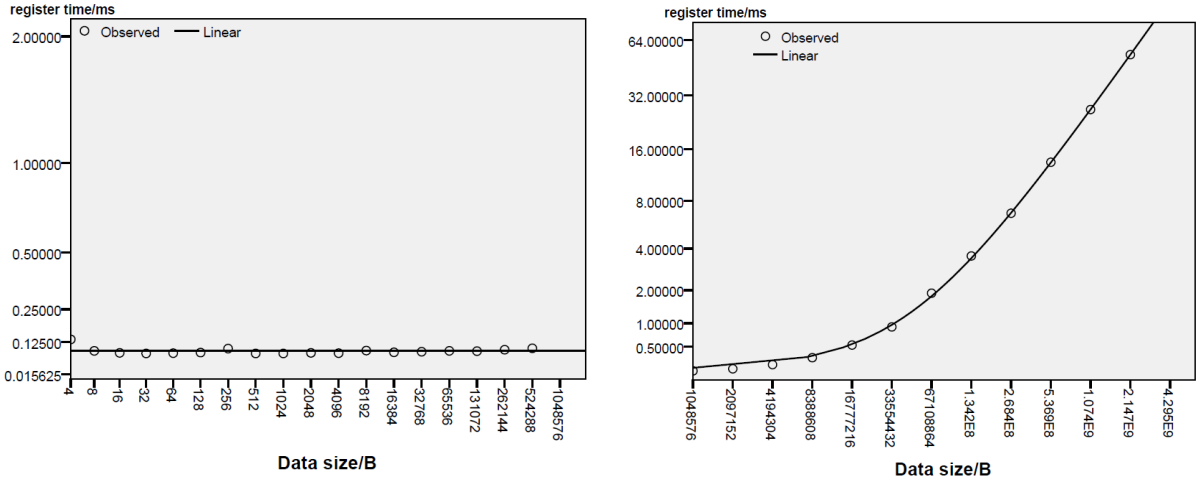
设置注册为页锁定存储的空间大小为 4B 到 2GB，测试相应的注册时间和解除注册时间，拟合得到页锁定存储注册耗时函数和解除注册耗时函数。页锁定存储注册耗时函数（结果单位为 ms）为：

$$T_{register}=\begin{cases}0.0935 & (S<1MB)\\ 2.473e\text{-}8*S+0.1306 & (S\geq 1MB)\end{cases}, \qquad (4.5)$$

页锁定存储的解除注册耗时函数（结果单位为 ms）为：

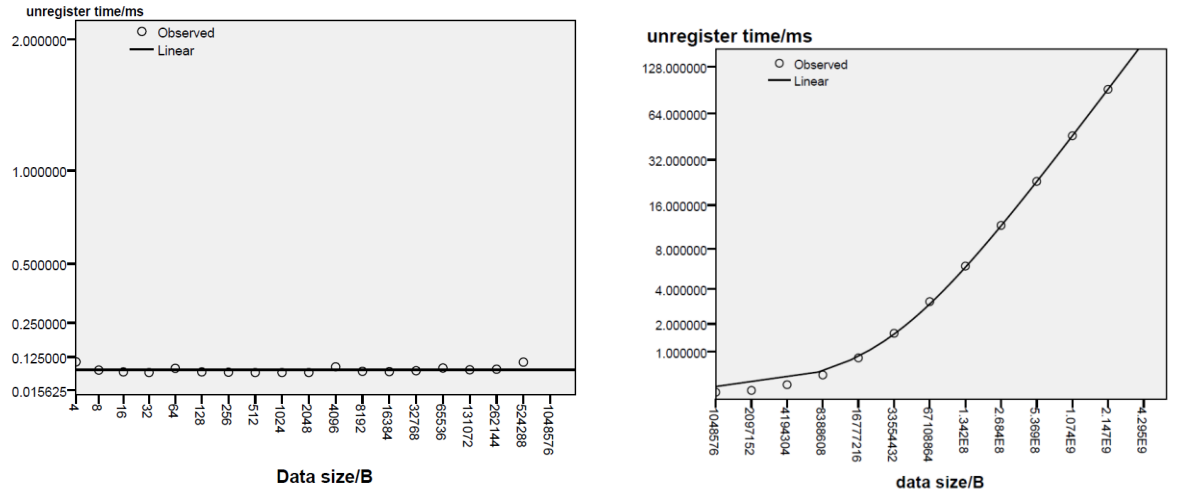
$$T_{unregister}=\begin{cases}8.117e\text{-}2 & (S<1MB)\\ 4.254e\text{-}008*S+0.1579 & (S\geq 1MB)\end{cases}。 \qquad (4.6)$$

绘制页锁定存储注册的实测时间和拟合耗时函数曲线，见图 4.6，绘制页锁定存储解除注册的实测时间和拟合的耗时函数曲线，见图 4.7，其中图（a）的存储空间小于 1MB，图（b）的存储空间大于 1MB。图中结果显示，实测时间与拟合函数曲线基本吻合，说明拟合的耗时函数是正确的。



（a）页锁定存储空间小于 1MB　　（b）页锁定存储空间大于 1MB

图 4.6 页锁定存储注册实测时间与拟合耗时函数曲线



（a）页锁定存储空间小于 1MB　　（b）页锁定存储空间大于 1MB

图 4.7 页锁定存储解除注册实测时间与拟合耗时函数曲线

4.5.4 主机端存储选择模型

可分页存储的相关开销有可分页存储分配开销、CPU 存储访问开销、数据通

信开销（包括主机端到设备端、设备端到主机端）、可分页存储释放开销等。故可分页存储的总耗时计算公式为：

$$T_{pagable} = T_{malloc} + T_{CPU\_access} + T_{pagable\_h2d} + T_{pagable\_d2h} + T_{free}。 \quad (4.7)$$

对于页锁定存储，相关开销包括可分页存储分配开销、页锁定存储注册开销、CPU 存储访问开销、数据通信开销（包括主机端到设备端、设备端到主机端）、页锁定存储解除注册开销、可分页存储释放开销等。页锁定存储的总开销为：

$$T_{pinned} = T_{malloc} + T_{register} + T_{CPU\_access} + T_{pinned\_h2d} + T_{pinned\_d2h} + T_{unregister} + T_{free}。(4.8)$$

其中可分页存储分配开销和释放开销相等，可不必考虑。CPU 存储访问时，应尽量使得主机端存储的类型状态为可分页存储；由于两者的 CPU 存储访问基本相当，故本节亦不考虑。

简化后的可分页存储耗时可表示为：

$$T_{pagable} = T_{pagable\_h2d} + T_{pagable\_d2h} = \sum_i \frac{S^{(i)}_{pagable\_h2d}}{BW^{(i)}_{pagable\_h2d}} + \sum_j \frac{S^{(j)}_{pagable\_d2h}}{BW^{(j)}_{pagable\_d2h}}, \quad (4.9)$$

其中 i 指代从主机端到设备端的多次传输，$S^{(i)}_{pagable\_h2d}$ 表示当次的传输数据大小，$BW^{(i)}_{pagable\_h2d}$ 为相应的 PCI-E 带宽；j 指代设备端到主机端的多次传输，$S^{(j)}_{pagable\_d2h}$ 是当次传输数据的大小，$BW^{(j)}_{pagable\_d2h}$ 是相应的 PCI-E 带宽。PCI-E 带宽根据公式 4.1-4.4 计算。

简化后的页锁定存储耗时表示为：

$$\begin{aligned} T_{pinned} &= T_{register} + T_{pinned\_h2d} + T_{pinned\_d2h} + T_{unregister} \\ &= \sum_i \left(T^{(i)}_{register}(S) + \sum_j \frac{S^{(j)}_{pinned\_h2d}}{BW^{(j)}_{pinned\_h2d}} + \sum_k \frac{S^{(k)}_{pinned\_d2h}}{BW^{(k)}_{pinned\_d2h}} + T^{(i)}_{unregister}(S) \right), \end{aligned} \quad (4.10)$$

其中 i 指代多个不同的主机端存储数组，对于每个主机端存储数组，考虑页锁定存储注册和解除注册开销（公式 4.5 和公式 4.6）；考虑通信开销时，j 指代当前页锁定数组的多次主机端到设备端通信，k 指代当前页锁定数组的多次设备端到主机端通信，PCI-E 带宽根据带宽模型（公式 4.1-4.4）计算。

选择主机端存储类型时，分别计算简化的可分页存储耗时（式 4.9）和页锁定存储耗时（式 4.10）并对比，选择耗时较少的主机端存储类型。

4.6 实例研究：PCA 降维

本节选用高光谱遥感影像 PCA（主成分分析）降维算法为例，验证存储选择模型的正确性和实用性，同时展示存储选择模型在具体应用中的使用方法。

在完成 PCA 降维算法的 GPU 移植和优化（详见第五章）基础上，分析主机端存储的数量、空间尺寸、通信次数及方向等信息。在高光谱遥感影像 PCA 降维算法中，关键步骤包括协方差矩阵计算、特征分解、提取主成分、PCA 变换。经热

点分析及相应的并行及优化，将协方差矩阵的计算、PC 变换两个过程映射到 GPU 进行计算。过程中与 GPU 有数据交互的主机端存储有高光谱影像矩阵、协方差矩阵、变换矩阵以及 PC 变换结果矩阵。

高光谱影像矩阵是输入数据，当取到一组 AVIRIS 高光谱数据（Width=614，Height=1087，Band=224）时，相应的数据尺寸（S）为 224*614*1087*sizeof(uchar)，约为 142.6MB，传输方向为主机端到设备端（H2D）。故取式 4.1 和式 4.3 计算通信带宽，取式 4.5 和式 4.6 计算页锁定注册和解除注册耗时。

协方差矩阵是协方差矩阵计算的结果矩阵，需要从 GPU 传输到 CPU（D2H），然后进行下一步的特征分解。其矩阵维度为 224*224，数据类型是 float，尺寸为 224*224*4=196KB，选择式 4.2 和式 4.4 计算通信带宽，选择公式 4.5 和公式 4.6 计算页锁定注册和解除注册耗时。

变换矩阵是特征分解后得到的特征向量矩阵，根据特征值大小及其贡献率取得主成分后，重排序后的特征向量与原始高光谱影像矩阵进行乘积变换。该矩阵的传输方向是主机端到设备端（H2D），尺寸为 224*224*sizeof(float)=196KB，页锁定存储分配和释放耗时公式选取与协方差矩阵相同，带宽计算选用式 4.1 和式 4.3。

PC 变换结果矩阵尺寸为 m*614*1087*sizeof(float)，其中 m 为取得的主成分数量，设置阈值为 99%时，该高光谱影像数据 PCA 降维取到的主成分数量为 27，尺寸约为 68.7MB，传输方向为 GPU 到 CPU（D2H），故取式 4.2 和式 4.4 计算通信带宽，取式 4.5 和式 4.6 计算页锁定注册和解除注册耗时。

根据上述主机端存储尺寸及相应的公式选择，利用 4.5 节的存储选择模型，计算相应的耗时信息。在 K20c 平台上测试实际耗时数据，见表 4.5。表中数据显示，利用模型预测的时间与真实运行时间基本相等，证明了模型的准确性。

表 4.5 主机端存储的模式预测与真实运行耗时

		hyperspectral image		covariance matrix		Trans matrix		PC result	
		model	run	model	run	model	run	model	run
Size(B)		149501632		200704		200704		72081144	
pagable	bandwidth(GB/s)	3.300	3.052	1.721	1.508	1.920	2.427	2.882	2.944
	transfer(ms)	42.192	45.619	0.109	0.124	0.097	0.077	23.292	22.805
pinned	register(ms)	3.828	4.068	0.094	0.101	0.094	0.092	1.913	1.869
	unregister(ms)	6.517	6.427	0.081	0.094	0.081	0.086	3.224	3.098
	bandwidth(GB/s)	5.896	5.869	4.477	3.063	3.953	3.063	6.233	6.218
	transfer(ms)	23.614	23.722	0.042	0.061	0.047	0.061	10.770	10.797
	all(ms)	33.959	34.217	0.216	0.256	0.222	0.239	15.907	15.764

对比表 4.5 中可分页存储和页锁定存储的模型预测数据，预测的最佳方案是高光谱影像矩阵和 PC 变换结果矩阵采用页锁定存储，而协方差矩阵和变换矩阵采用可分页存储。分别实现可分页存储、页锁定存储、预测方案，实验测试主机端存储相关的总时间，见图 4.8。图中数据显示，最佳方案总耗时最少，说明与传统单纯的采用可分页存储或页锁定存储相比，存储选择模型能在真实应用中获得更好的性能。

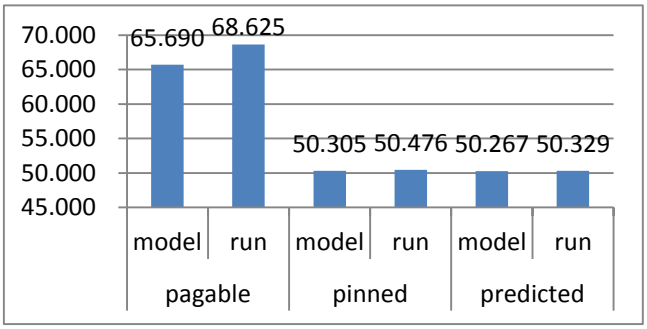


图 4.8 各方案的预测与执行时间/ms

4.7 其他异构协同优化技术与实例研究

在 CPU/GPU 异构系统中，zerocopy（零拷贝）技术是一种重要的远端访存手段，合理使用能够有效提升 CPU/GPU 异构程序性能，本节设计了两项优化方案并利用实例进行验证。此外，计算与通信重叠、（CPU）计算与（GPU）计算重叠是常见的异构协同优化方法，本节将利用实例研究进行探讨。

4.7.1 zerocopy 优化研究

4.7.1.1 zerocopy 带宽测评

零拷贝是 CPU/GPU 异构系统提供的特殊功能，可以实现主机端和设备端的直接数据访问，无需显式的数据通信函数。尽管 zerocopy 直接访问数据，但不可避免仍然要通过 PCI-E 通道，那么零拷贝数据访问的 PCI-E 带宽利用率如何？零拷贝的带宽能否匹配 cudaMemcpy()等通信函数？

针对上述疑问，本节设计一组 microbenchmark 来测评和对比 zerocopy 和 cudaMemcpy()函数的带宽，测试带宽数据见图 4.9。图中数据显示，零拷贝访存带宽与 cudaMemcpy()函数传输带宽基本一致，说明理想化的零拷贝数据访问能够充分发挥 PCI-E 带宽。

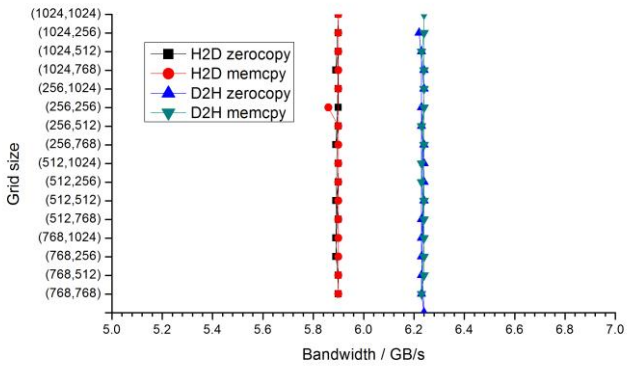


图 4.9 零拷贝的带宽

4.7.1.2 利用 zerocopy 减少全局存储访问

通过上一节测评可知，零拷贝访存带宽与 cudaMemcpy()函数传输带宽相当，这里设计一种 zerocopy 优化方案：对于 kernel 函数中仅访问一次的数据，如果利用零拷贝技术直接访问数据，就可以避免一次全局存储的读和写操作。而传统方法须先将数据复制到全局存储（包含一次全局存储写操作），然后 kernel 函数从全局存储读取数据。

用向量内积实验来验证上述优化方案，分别实现传统方法的 GPU 向量内积、cublas 向量内积和零拷贝向量内积运算，处理 1024*1024 个 float 数据向量内积运算，统计传输时间和计算时间的总和，见表 4.6。K20c GPU 平台结果显示传统 GPU 向量内积与 cublas 库耗时几乎相等，而利用了 zerocopy 技术后耗时减少了，说明利用“zerocopy 访问避免一次全局存储读写”来加速 GPU 程序是可行的。实验结果也为进一步优化 cublas 库提供了一个新的方向，即使用零拷贝技术进行优化。

表 4.6 向量内积的零拷贝优化效果

vision	original	cublas	zerocopy
time/ms	1.552	1.555	1.361

4.7.1.3 利用 zerocopy 实现计算通信重叠

Zerocopy 访问远程数据时，仍然要通过 PCI-E 通道，而计算与通信重叠是常见的异构协同优化方法。基于此，提出另一项 zerocopy 优化方案：zerocopy 访存支持隐式的计算和通信重叠。本节利用 mandelbrot 实验验证该优化方案。

曼德博罗特集(mandelbrot)是一种经典的图像绘制算法，利用一个复平面上的简单迭代公式的迭代次数绘图，获得一个奇异且瑰丽的图案（图 4.10）。曼德博罗特集的迭代公式为:

$$z_{k+1} = (z_k)^2 + c, \tag{4.11}$$

其中最大迭代次数为 255，即 unsigned char 类型上限。像素点的颜色规则为：若迭代次数未达到最大迭代次数，而计算的 Z_n 的模超过最大阈值 2，则停止迭代，以该迭代值为该点颜色值；若迭代次数达到最大迭代次数而 Z_n 的模依然未超过阈值，则以 255 作为其颜色值。

曼德博罗特集计算是一个特殊的实例，无输入数据，计算和访存相对独立，且计算时间和通信时间相当，适合采用计算与通信重叠优化。对于 RGB 结果图像数据（结构体中的顺序为 BGR）保存时的访存方式，有多种不同的可能性；经实践测试，这些图像结构访存方式在全局存储中性能相当，而在 zerocopy 中却性能差异巨大。

从曼德博罗特集的计算实验切入，在 K20c 实验平台上，分别统计曼德博罗特集的普通 CUDA 版本、数个零拷贝版本以及计算通信重叠版本的总时间，见表 4.7。根据对曼德博罗特集结果图像中 RGB 三个波段图像访存方式的不同，设计了 4 个 zerocopy 版本。下面阐述各版本基本信息：

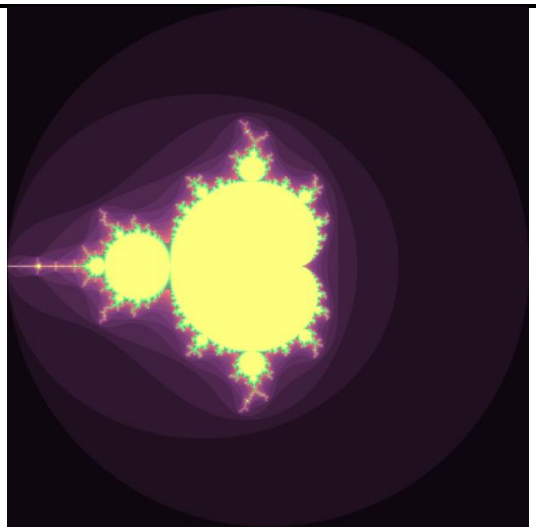


图 4.10 Mandelbrot 结果图像

（cuda）曼德博罗特集的基础 CUDA 版本；

（cuda2）计算和通信重叠版本（4.7.2 节详细讨论）；

（zerocopy1）zerocopy 单独访问 RGB 结构体中的单个元素；

（zerocopy2）将 RGB 结构体数组拆分成 3 个 uchar 数组，并对齐访存；

（zerocopy3）zerocopy 整体访存 RGB 结构体；

（zerocopy4）采用 uchar4 内置数据类型，并整体访问 uchar4 结构体。

表 4.7 Mandelbrot 零拷贝性能/ms

n	1024	2048	4096	8192	16384	32768
cuda	1.38	4.83	17.29	67.68	267.57	1062.54
zerocopy1	73.32	290.73	1162.41	4653.66	18637.97	74596.48
zerocopy2	1.00	3.75	15.07	64.40	285.40	1170.30
zerocopy3	72.65	290.69	1162.41	4653.75	18638.93	74593.46
zerocopy4	0.95	3.07	12.04	52.87	204.64	846.09
cuda2	1.07	3.78	13.06	52.56	208.44	827.15

表中数据显示：

（1）零拷贝技术不能盲目使用，否则将导致巨大的不可控开销，比如表中 zerocopy1 和 zerocopy3；

（2）char3 数据类型（RGB 结构体）不适用于零拷贝访问，无论是单独访问其中元素（zerocopy1）还是整体访存（zerocopy3）。笔者曾测试过用 char3 类型（RGB 结构体）数据对全局存储访存，与 cuda 版本性能相当。说明零拷贝访问更加严格，而全局存储访问相对宽松。

（3）char 类型数据访存也不太适合采用零拷贝技术。zerocopy2 在数据量小时比 cuda 版本快，而数据量增大后比 cuda 版本慢，两者性能差距并不大。本文将其原因归结为 char 类型数据访存时存在带宽浪费现象，浪费的带宽与零拷贝的收益大约一致，最终导致该版本没有获得明显的性能收益。

（4）char4 类型数据（长度为 4 字节的数据）适合使用零拷贝技术并能获得显

著优化效果。Zerocopy4 版本性能表现良好，对比 cuda 版本加速效果明显。说明长度为 4 字节的数据利用零拷贝技术才能充分利用 PCI-E 带宽，并获得性能收益。

（5）零拷贝技术隐式地实现了计算通信重叠。对比 zerocopy4 和 cuda2 版本的时间数据，发现尽管两者在不同数据尺寸时各有千秋，但整体的性能相差不大。故可判定零拷贝技术在实现时采用了隐式的计算通信重叠技术，GPU 执行过程中，部分线程计算得到结果后利用零拷贝技术将结果写回主机端内存，而于此同时其他线程继续计算曼德博罗特集。

4.7.2 计算与通信重叠

计算与通信重叠是最常见的 CPU/GPU 异构系统协同优化方法之一。这里的计算主要指代 GPU 的 kernel 函数执行，通信指代主机端和设备端的数据传输。计算与通信重叠主要通过 CUDA 多流并发机制实现，不同 CUDA 流上的计算和通信可以同时执行，参与通信的主机端存储必须为页锁定存储。本节以曼德博罗特集计算为例验证计算与通信重叠的异构协同优化效果。

Mandelbrot 的 kernel 函数执行时间与结果图像的传输时间相当，若重叠计算和通信，性能收益是可以预期的。曼德博罗特集的结果图像矩阵中，故可将其按行划分为 n 等分，每个等分图像的计算和另一等分图像的通信（已完成计算）可以同时进行，实现计算与通信重叠。实现曼德博罗特集的计算与通信重叠版本（cuda2），通过实验与原始版本（cuda）进行对比，实验结果见表 4.8。表中数据显示，计算和通信重叠优化后，曼德博罗特集的运算耗时明显减少了，说明了计算与通信重叠的优化效果。

表 4.8 Mandelbrot 的计算与通信重叠/ms

n	1024	2048	4096	8192	16384	32768
cuda	1.38	4.83	17.29	67.68	267.57	1062.54
cuda2	1.07	3.78	13.06	52.56	208.44	827.15

4.7.3 计算与计算重叠

CPU/GPU 异构系统中，CPU 与 GPU 相对独立，两者可以同时执行不同的计算任务。在 CUDA 中，kernel 函数是以非阻塞的方式启动的，在 GPU 执行 kernel 函数的同时，CPU 仍继续执行 CPU 端代码，直到下一个与 GPU 相关的 kernel 函数或通信函数。本节选择第五章的 MNF 降维算法为例，研究计算与计算重叠的异构协同优化效果。

最大噪声分数变换（Maximum Noise Fraction Rotation，MNF Rotation）是一种经典的高光谱遥感影像线性降维方法，加速热点包括噪声估计、协方差矩阵计算和 MNF 变换（5.3.3 节详细介绍）。Wu 等[62]利用 CUDA 加速噪声估计，利用 CUBLAS 库加速协方差矩阵计算和 MNF 变换，实现准实时的 MNF 降维。

本节分别实现计算与计算重叠优化前（mnf_0）后（mnf_1）的高光谱 MNF 降

维算法，并利用 CUBLAS 库实现 MNF 降维算法（mnf_cublas，其中协方差计算过程采用 cublasSasum 函数和 cublasSsyrk 函数，相比论文[62]所使用的 cublasSegemm 函数性能更优），执行并统计 MNF 降维时间，见表 4.9。表中数据显示，经计算与计算重叠优化后，MNF 算法的耗时明显减少了，说明计算与计算重叠在 MNF 降维算法中切实有效。并且最终 MNF 算法的性能优于 CUBLAS 库实现的版本。

表 4.9 MNF 的计算与计算重叠/ms

input	data1	data2	data3
mnf_0	1186.12	2004.77	6013.53
mnf_1	949.77	1785.14	5719.42
mnf_cublas	2923.20	3969.39	8888.73

4.8 异构协同优化总结

在 CPU/GPU 异构程序开过过程中，分别完成 CPU 代码和 GPU 代码的性能优化，还需进行 CPU/GPU 异构协同优化研究。根据本章研究以及实验结果，可以总结出一些 CPU/GPU 异构系统协同优化方法及其适用范围，详见表 4.10。利用这些异构协同优化方法，能有效辅助程序员优化 CPU/GPU 异构并行程序，获得可观的性能收益。

表 4.10 CPU/GPU 异构协同优化方法及其适用范围

优化方法	适用范围	实例
减少数据通信	更倾向于对算法流程本身的改进，而不再是纯粹的异构协同优化（不在本章研究范围内）	5.4.1.1 节的式 5.16 公式变形
计算与通信重叠	（1）算法流程中存在无依赖的、可以同时执行的计算和通信； （2）单一且运算可分的 kernel 函数，kernel 执行时间与通信时间相当或同一量级，可通过拆分 kernel 函数来实现计算与通信重叠。	Mandelbrot 计算
计算与计算重叠	从算法流程中寻找可以同时执行的 CPU 端计算与 GPU 端计算	MNF 降维算法
主机端存储类型选择	分段的主机端存储选择模型（4.5 节）	PCA 降维算法； FastICA 降维算分； MNF 降维算法； CBF 波束形成； MVDR 波束形成；

优化方法	适用范围	实例
Zerocopy 优化	（1）零拷贝技术能实现主机端和设备端数据通信功能，且零拷贝技术能充分发挥 PCI-E 带宽； （2）合理利用零拷贝技术可以减少对全局存储的访存，并从中获得性能收益； （3）零拷贝技术不能盲目使用，char3 这类不规整的数据使用零拷贝技术将导致不可控的巨额开销； （4）char 类型数据访问不建议采用零拷贝技术； （5）char4 这类 4 字节规整数据适合采用零拷贝优化； （6）零拷贝技术采用了隐式的计算通信重叠。	向量内积； Mandelbrot 计算； 5.5.3 节中的 ICA 迭代优化；

4.9 本章小结

本章从经典矩阵乘法提出主机端存储的选择问题；量化探索了页锁定存储和可分页存储的各项相关开销，得到一些基本规律；进一步对主机端存储的 CPU 访存带宽、PCI-Express 带宽、页锁定存储的注册及解除注册时间进行建模，并提出分段的主机端存储选择模型；以高光谱遥感影像 PCA 降维算法为例，对比模型预测时间与真实运行耗时，结果验证了存储选择模型的正确性。

此外，针对 CPU/GPU 异构系统中特殊的零拷贝技术，测试了其 PCI-E 带宽利用率，设计两项 zerocopy 优化方案并利用实验进行验证；给出了计算与通信重叠、计算与计算重叠的实例优化研究。最后总结了异构协同优化方法、适用范围和优化实例。

本章研究内容很好地解答了 2.4.3 节关于异构协同优化的几点思考，研究成果能在后文的高光谱影像降维和声呐信号波束形成两类领域有效应用获得性能收益。

第五章 基于众核 GPU 的高光谱影像降维算法

前两章通过 microbenchmark 测评设计出大量真实可靠的性能优化策略和建议，构建了 GPU 访存优化框架和主机端存储选择模型，本章开始尝试将这些研究成果应用到具体的真实应用中，以收获实际的性能收益。

高光谱遥感影像降维过程计算复杂、运算量大、运算时间长，普通串行程序无法满足其计算需求，故需要并行计算。本章针对经典的高光谱遥感影像线性降维中的主成分分析、快速独立成分分析和最大噪声分数变换 3 类算法开展并行和性能优化研究，基于分布存储、共享存储和 GPU 三个并行层次设计热点的并行方案，进行性能优化研究并量化分析优化效果，提出面向众核体系结构的高光谱影像并行降维框架。实验结果显示并行降维框架中的并行算法可获得良好的加速效果，其中 Gs-PCA 算法最高加速 119.7 倍，Gs-FastICA 算法最高加速 106.6 倍，G-MNF 算法最高加速 86.9 倍。最后通过实验分析并行降维算法的可扩展性。

本章工作可用助力高光谱遥感领域科学家快速处理高光谱数据，更高效地进行科学研究。

5.1 引言

高光谱遥感技术广泛应用于军事、农业、环境科学、地质、海洋学等领域，这些领域大都要求及时处理[63]。高光谱影像数据有波段多、数据量大、相关性强、冗余多等特点，直接处理将导致样本类别训练困难、维数灾难、空空间现象等严重的计算问题[64-66]。因此，需要引入降维对高光谱影像数据进行预处理。降维是通过特定的映射，将高维影像数据转换成低维影像数据，并保持信息量基本不变。

高光谱影像降维方法可分为线性和非线性两类，线性降维有主成分分析（principal component analysis, PCA）[67]、独立成分分析（independent component analysis, ICA）[68]等方法，非线性降维包括基于核的方法和基于流行学习的方法，如等距映射法（isometric feature mapping, Isomap）[69]、局部线性嵌入（locally linear embedding, LLE）[70]等。由于高光谱影像波段多、数据量大等特征，降维处理过程复杂且耗时。提高降维速度是重要研究热点，设计新的快速降维方法和在现有高性能计算平台上开发高效的并行降维算法是加快降维过程的两个重要研究方向。

本章将从并行降维的角度来加速高光谱影像降维，主要基于主流的 CPU/GPU 异构系统，对 3 大主流线性降维算法开展并行和优化研究，提出相应的并行降维算法，并构建了面向众核体系结构的高光谱并行降维框架，以满足高光谱领域科学家对高光谱影像降维的快速计算和及时处理的需求。

5.2 相关工作

本节主要介绍多核 CPU 平台、FPGA 平台和 GPU 平台这 3 类平台上的高光谱影像并行处理的相关研究，以此来论述本章的相关工作。

在传统的多核 CPU 集群中，Valencia 等人[73]基于异构 MPI 在网络集群系统研究了高光谱影像并行处理技术。 Plaza 等人[74]提出基于神经网络的高光谱影像并行分类算法。 Plaza 等人[75]基于商用集群讨论了高光谱影像并行数据分析方法，并实现了并行的无监督多通道形态学变换来集成空间和谱间信息。Dong 等人[76]基于高光谱影像提出并行的高斯马尔可夫随机场（Para-GMRF）异常检测算法，将 3D 的高光谱影像立方体在空间域进行分解，并映射到多节点进行并行处理。Dong 等人[77]用线性代数库和 MPI 库实现了并行的高光谱影像相关向量机（RVM）分类方法。Plaza 等人[78]在异构网络工作站上，基于数据压缩技术提出新框架，用来提升高光谱标准光谱解混并行处理的可扩展性。Tzeng 等人[79]提出并实现基于多核 CPU 的并行 DBC，用来统计纹理信息，在动态学习神经网络（DLNN）中通过提取和结合光谱信息来获得更好的分类精度。Plaza 等人[80]介绍了高光谱数据信息提取的图像信号处理技术开发和应用，以及采用并行和分布处理技术有效处理时间受限应用的新趋势。Bernabe 等人[81]开发了并行的高光谱影像自动目标生成处理（ATGP）算法。Plaza 等人[82]设计了一个并行形态学算法来集成高光谱影像的空间和谱间信息。

有些学者基于 FPGA（field programmable gate arrays）研究高光谱影像快速处理方法。Du 等人[83]从 FastICA 中提出并行 ICA（pICA）算法，并讨论了 pICA 的 FPGA 实现在分析高光谱影像时的能力限制。Plaza 等人[84]利用商用集群、异构的分布式工作站网络、基于硬件的计算架构等高性能计算平台加速了自动目标识别、光谱混合分析和数据压缩。Gonzalez 等人[85]在 FPGA 和 GPU 上探索了高光谱遥感处理的全波段解浑并行实现。Plaza 等人[86]利用 MPI、FPGA 和 CUDA 分别实现了 ENVI 软件包中的 PPI 算法。

基于 CPU/GPU 异构系统，亦有大量的高光谱影像并行处理相关研究。2006 年，CUDA 发布之前，Setoain 等人[87]提出了基于 GPU 的高光谱影像处理框架，充分利用了 GPU 的多级并行。Sánchez 等人[88]基于 GPU 实现了高光谱影像解浑。Platoš 等人[89]致力于用 GPU 加速非负矩阵分解（NMF）。Ramalho 等人[90]基于 GPUs 实现 FastICA 并行算法。Du 等人[91]研究用随机选择（RS）和随机规划（RP）分析高光谱影像，并用 GPUs 和集群分别实现，其中在 GPU 上加速 10 倍和 32 个处理器中加速 31 倍。Rosario 等人[92]采用 MATLAB Jacket 工具包研究高光谱影像分析工具包的 GPU 实现。Heras 等人[93]提出并用 GPU 实现了两个用于救援搜寻任务中

目标识别的人工智能算法，第一个算法将神经网络用于图像个别像素级，第二个算法采用分层人工神经网络框架。Bernabe 等人[94]采用两项优化来加速 ATDCA 的计算性能，首先用 Gram–Schmidt 正交化方法替代分类算法中的正交规划处理，接着在商用 GPU 中进行并行实现。Sanchez 和 Plaza 等人[95]基于 GPU 开发了有损高光谱影像压缩的迭代错误分析（IEA）算法。Santos 等人[96]描述了在线有损高光谱影像压缩算法的 GPU 实现，并提出基于 GPU 的并行压缩框架。Wu 等人[97]基于 GPU 开发了有监督的用于处理高光谱影像的谱间空间分类器的并行实现。Wu 等人[98]给出高光谱影像分类中支撑向量机（SVM）复合核心的并行实现。Tan 等人[99]用两级并行（CUDA 和 OpenMP）计算框架来加速基于支撑向量机（SVM）的高光谱遥感影像分类。Tarabalka 等人[100]基于 GPU 研究高光谱异常检测算法。Hossam 等人[101]用 GPU、CPU/GPU 节点和 CPU/GPU 集群加速了递归分层分割（RHSEG）分析算法。Setoain 等人[102]研究形态学端元提取算法的 GPU 实现。Paz 等人[103]在集群和 GPU 上开发了自动目标和异常识别算法的并行实现。

上述研究主要基于 CPU、FPGA、GPU 三个计算平台，更倾向于应用的并行化，而较少进行专门的面向体系结构的性能优化研究。此外基于较新的 MIC 计算平台，暂时还未有相关研究。

本章将探索 3 类高光谱影像线性降维算法的并行方案，并试图根据第三、四章设计的优化策略对其进行细粒度优化。最终构建一个面向众核体系结构的高光谱影像并行降维框架，使其适用于 CPUs、GPUs、MICs（Phis）等多种高性能计算平台并能获得良好的性能。下文将重点介绍基于 CPU/GPU 异构平台的并行和性能优化，与 CPU 和 MIC 平台相关的并行和优化研究请参见成果[1,2,3,5,7]。

5.3 高光谱影像线性降维算法

5.3.1 主成分分析

5.3.1.1 高光谱影像 PCA 降维算法

主成分分析利用统计特征进行多维正交线性变换，将原始特征空间的特征轴旋转到平行于混合集群结构轴的方向去，得到新的特征轴[127]。在高光谱遥感影像降维中，通过 PCA 变换，把高维图像变换到低维空间，原始数据的信息被压缩到较少的波段中，实现高光谱影像降维。

用 $\boldsymbol{X}=(\boldsymbol{X}_1,\boldsymbol{X}_2,\cdots,\boldsymbol{X}_S)=(\boldsymbol{X}_1,\boldsymbol{X}_2,\cdots,\boldsymbol{X}_B)^T$ 表示高光谱遥感图像，其中 S 表示高光谱遥感图像的象元数目，B 表示波段数目。下面对 PCA 降维算法进行简要描述：

Step1. 首先计算高光谱影像 $\boldsymbol{X}$ 中各波段间的协方差，并构建协方差矩阵 $\boldsymbol{\Sigma}$；

$$\boldsymbol{\Sigma}^{ij}=\frac{1}{S-1}[\sum_{k=1}^{S}(\boldsymbol{X}_k^i-\overline{\boldsymbol{X}^i})(\boldsymbol{X}_k^j-\overline{\boldsymbol{X}^j})] \tag{5.1}$$

Step2. 对协方差矩阵 $\boldsymbol{\Sigma}$进行特征分解，求得特征值 $\boldsymbol{D}$ 和特征向量 $\boldsymbol{V}$；

$$\boldsymbol{\Sigma}=\boldsymbol{V}\boldsymbol{D}\boldsymbol{V}^{\mathrm{T}} \tag{5.2}$$

Step3. 根据特征值计算贡献率 v，并依据给定的阈值 R（本章取 99%），按特征值从大到小选取主成份个数 m，使得累计贡献率∑v≥R；

Step4. 对选中的主成分进行 PCA 变换。令 $\boldsymbol{A}=\boldsymbol{V}^{\mathrm{T}}$，PCA 变换结果矩阵：

$$\boldsymbol{Y}=(\boldsymbol{Y}_1,\boldsymbol{Y}_2\cdots\boldsymbol{Y}_{\mathrm{m}})^{\mathrm{T}}=\boldsymbol{A}\boldsymbol{X} \tag{5.3}$$

通过 PCA 降维，高光谱遥感影像数据从 B 个波段降为 m 个波段，可以大幅减少后续高光谱影像处理的运算量。

5.3.1.2 PCA 算法热点分析

实现高光谱影像 PCA 降维算法（已进行向量化优化和连续访存优化），用一组高光谱图像数据（data1，详细参数见表 5.7）测试串行 PCA 降维的各步骤时间消耗，计算其占总时间的百分比（表 5.1）。表中数据显示，PCA 降维的时间主要消耗在协方差矩阵计算和 PCA 变换，这两个步骤是本文并行和优化研究的加速热点；由于本文采用的 AVIRIS 高光谱数据的波段数恒为 224，协方差矩阵特征值求解时间消耗恒定且较小，可不必考虑并行。

表 5.1 各步骤耗时百分比

Step	time	percent
Input	95.239	0.30%
Step1	25689.01	81.86%
Step2	310.835	0.99%
Step3	2.341	0.01%
Step4	4862.563	15.49%
Output	423.323	1.35%

5.3.2 独立成分分析

5.3.2.1 基于负熵最大的快速独立成分分析

FastICA 算法有基于负熵最大、基于似然最大、基于峭度等 3 种形式，本章研究基于负熵最大的 FastICA 算法[71-72]。该算法以负熵最大作为搜寻方向，实现独立源的顺序提取，充分体现传统线性变换中的投影追踪（Projection Pursuit）思想，该算法还采用定点迭代的优化算法，使收敛更加快速、稳健[71-72]。

对于高光谱影像数据 $\boldsymbol{X}$（宽 W、高 H、波段 B），$\boldsymbol{X}$ 可看成是 B 行 S 列（S=W×H）的 2 维矩阵，矩阵的 1 行表示 1 个波段的高光谱影像数据。通过独立成分分析方法降维，可以获得 m 个独立成分，m<B，从而实现降维效果。

step1. 计算高光谱影像矩阵 $\boldsymbol{X}$ 的协方差矩阵 $\boldsymbol{\Sigma}$（公式 5.1）；

step2. 对协方差矩阵 $\boldsymbol{\Sigma}$进行特征分解，得到特征值 $\boldsymbol{D}$ 及特征向量 $\boldsymbol{V}$，并根据特征值 $\boldsymbol{D}$ 和阈值 T（本文选 99%）选取最终 IC 图像数量 m；

step3. 计算白化矩阵 M：

$$\boldsymbol{M} = \boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{V}^{\mathrm{T}} \tag{5.4}$$

step4. 高光谱影像数据白化处理，其中 $\boldsymbol{X}$`为 $\boldsymbol{X}$ 的零均值处理；

$$\boldsymbol{Z}=\boldsymbol{M}\boldsymbol{X}` \tag{5.5}$$

step5. ICA 迭代计算：

Step5.1 初始化 $\boldsymbol{W}_{\mathrm{i}}$；

Step5.2 迭代计算 $\boldsymbol{W}_{\mathrm{i}} = \sum\{\boldsymbol{Z}\mathrm{g}(\boldsymbol{W}_{\mathrm{i}}^{\mathrm{T}}\boldsymbol{Z})\} - \sum\{\mathrm{g}`(\boldsymbol{W}_{\mathrm{i}}^{\mathrm{T}}\boldsymbol{Z})\}\boldsymbol{W}$， （5.6）

其中 g 为非线性函数，经测试，取 $\mathrm{g}(\mathrm{y}) = \mathrm{y}^3$时可更稳定、更快地使 ICA 迭代收敛；

Step5.3 正交化 $\boldsymbol{W}_{\mathrm{i}} = \boldsymbol{W}_{\mathrm{i}} - \sum\left(\boldsymbol{W}_{\mathrm{i}}^{\mathrm{T}}\boldsymbol{W}_{\mathrm{j}}\right)\boldsymbol{W}_{\mathrm{j}}$； （5.7）

Step5.4 归一化 $\boldsymbol{W}_{\mathrm{i}} = \dfrac{\boldsymbol{W}_{\mathrm{i}}}{\|\boldsymbol{W}_{\mathrm{i}}\|}$； （5.8）

Step5.5 判断是否收敛，不收敛则返回到 Step5.2，若收敛，如果所有的 $\boldsymbol{W}_{\mathrm{i}}$ 已经计算结束（即 i=m），则退出，否则令 i=i+1，返回到 Step5.1 计算新的 $\boldsymbol{W}_{\mathrm{i}}$；

Step6. 计算独立成分变换 $\boldsymbol{Y}=\boldsymbol{W}\boldsymbol{Z}$。 （5.9）

5.3.2.2 FastICA 算法热点分析

实现串行的高光谱遥感影像 FastICA 降维算法，进行向量化优化和连续访存优化，选用高光谱影像数据（data1）进行降维，测试并统计各步骤占总计算时间的比率（表 5.2）。表中数据显示，Step1 的协方差矩阵计算、Step4 的白化处理、Step5 的 ICA 迭代和 Step6 的 ICA 变换等过程占总计算时间的比重超过 99%，是并行和性能优化研究的主要热点。

5.3.3 最大噪声分数变换

5.3.3.1 最大噪声分数变换

最大噪声分数变换（MNF）将原始数据中的噪声分离，提取影像数据的主要特征，从而表征主要信息[104]。该算法由两层主成分变换构成，在主成分分析基础上考虑了噪声对图像的影响，具有更好的效果，是目前主流的线性降维算法，实现其并行化在高光谱降维处理领域具有重要意义。

最大噪声分数变换根据图像质量进行数据降维和排列，其采用 SNR 和噪声比例对图像质量进行评价。MNF 变换通过寻找变换矩阵，使得变换后的图像按信噪

比递减顺序排列，从而达到降维的目的。对于高光谱影像数据 X，通过 MNF 方法降维，可以获得 m 个成分，其中 m<B，从而实现降维的目的。

表 5.2 FastICA 各步骤时间比率

Step	Time/ms	precent
Input	101.372	0.06%
Step1	25690.41	15.78%
Step2	313.341	0.19%
Step3	209.467	0.13%
Step4	39768.04	24.43%
Step5	94133.92	57.82%
Step6	2209.261	1.36%
Output	368.977	0.23%

Step1. 利用滤波方法对高光谱矩阵 $\boldsymbol{X}$ 进行噪声估计；

Step2. 计算滤波后的噪声协方差矩阵 C_N；

Step3. 对 C_N 进行特征分解，$D_N = U^T C_N U$. （5.10）

其中 D_N 为降序排列的特征值，U 为所对应的特征向量，求解得到变换矩阵

$P = UD_N^{-\frac{1}{2}}$； （5.11）

Step4. 求原始高光谱 X 的协方差矩阵 C_D；

Step5. 对 C_D 变换：$C_{D-adj} = P^T C_D P$； （5.12）

Step6. 对 D_{D-adj} 特征值分解，$D_{D-adj} = V^T C_{D-adj} V$. （5.13）

其中 D_{D-adj} 为降序排列的特征值，V 为所对应的特征向量；

Step7. 计算变换矩阵 T=PV； （5.14）

Step8. 进行 MNF 变换 $Z = T^T X$。 （5.15）

5.3.3.2 MNF 算法热点分析

实现串行的高光谱影像MNF降维算法,进行向量化优化和连续存储访问优化，对高光谱影像数据（data1）进行降维，测试并统计各步骤占总计算时间的比重（图 5.1）。图中数据显示， Step2 和 Step4 的协方差矩阵计算占比最大超过 80%，Step1 的噪声估计（滤波）和 Step8 的 MNF 变换耗时明显。因此，协方差矩阵计算、噪声估计和 MNF 变换是 MNF 算法并行移植和性能优化的研究热点。

5.4 降维热点并行化方案

由热点分析可知，需要考虑进行并行和优化研究的热点包括协方差矩阵计算、PCA 变换、白化处理、ICA 迭代、ICA 变换、噪声估计（滤波）和 MNF 变换等，其中 PCA 变换、白化处理、ICA 变换和 MNF 变换的计算过程类似，并行方案和

优化策略亦可通用，故仅需考虑其中之一。下面分别展开讨论这些热点的并行化方案。

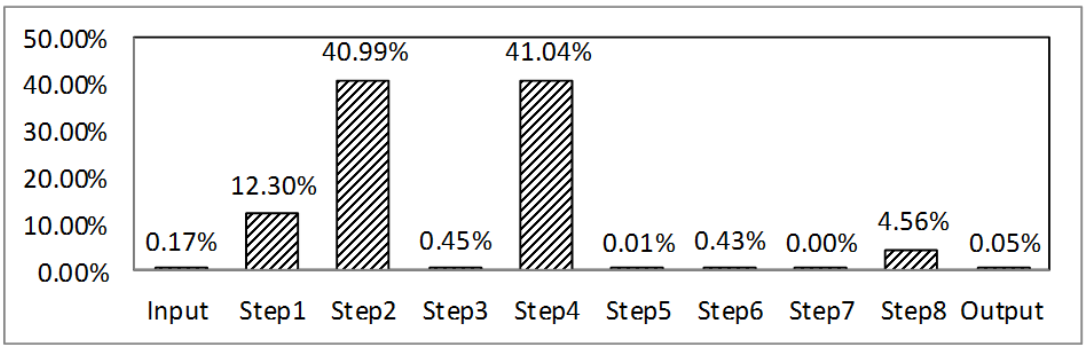


图 5.1 串行 MNF 时间

除了热点的 GPU 映射方案，还设计了基于分布存储和共享存储的热点并行方案，在 5.6 节构建的面向众核体系结构的高光谱影像并行降维框架中，包含本节设计的所有并行方案。

5.4.1 协方差矩阵的并行计算

5.4.1.1 分布存储协方差矩阵并行计算

在分布存储系统中，不同进程间的存储空间是独立的，故需要利用消息通信来协同多进程间的计算任务。当多个进程计算协方差矩阵时，需要 3 次消息通信，分别是各波段像素中间结果的收集、root 进程算得波段像素平均值后广播给所有进程、收集各进程的协方差矩阵计算子结果并完成计算。经过下面的公式变换，可以将消息通信次数降为 1 次（契合 4.8 节中减少数据通信的优化方法）。

$$\boldsymbol{\Sigma}^{ij}=\frac{1}{S-1}[\sum_{k=1}^{S}(\boldsymbol{X}_k^i-\overline{\boldsymbol{X}^i})(\boldsymbol{X}_k^j-\overline{\boldsymbol{X}^j})]$$

$$=\frac{1}{S-1}\left[\sum_{k=1}^{S}\left(\boldsymbol{X}_k^i\boldsymbol{X}_k^j+\overline{\boldsymbol{X}_k^i}\,\overline{\boldsymbol{X}_k^j}-\boldsymbol{X}_k^i\overline{\boldsymbol{X}_k^j}-\overline{\boldsymbol{X}_k^i}\boldsymbol{X}_k^j\right)\right]$$

$$=\frac{1}{S-1}\left[\sum_{k=1}^{S}\boldsymbol{X}_k^i\boldsymbol{X}_k^j+S\,\overline{\boldsymbol{X}^i}\,\overline{\boldsymbol{X}^j}-\overline{\boldsymbol{X}^j}\sum_{k=1}^{S}\boldsymbol{X}_k^i-\overline{\boldsymbol{X}^i}\sum_{k=1}^{S}\boldsymbol{X}_k^j\right]$$

$$=\frac{1}{S-1}\left[\sum_{k=1}^{S}\boldsymbol{X}_k^i\boldsymbol{X}_k^j+S\,\overline{\boldsymbol{X}^i}\,\overline{\boldsymbol{X}^j}-\overline{\boldsymbol{X}^j}S\,\overline{\boldsymbol{X}^i}-\overline{\boldsymbol{X}^i}S\,\overline{\boldsymbol{X}^j}\right]$$

$$=\frac{1}{S-1}\left[\sum_{k=1}^{S}\boldsymbol{X}_k^i\boldsymbol{X}_k^j-\frac{1}{S}\sum_{k=1}^{S}\boldsymbol{X}_k^i\sum_{k=1}^{S}\boldsymbol{X}_k^j\right]$$

$$=\frac{1}{S-1}\left[\begin{array}{l}\left(\sum_{k=1}^{s/N}\boldsymbol{X}_k^i\boldsymbol{X}_k^j+\sum_{k=s/N+1}^{2s/N}\boldsymbol{X}_k^i\boldsymbol{X}_k^j+\cdots+\sum_{k=(N-1)s/N+1}^{s}\boldsymbol{X}_k^i\boldsymbol{X}_k^j\right)\\ -\frac{1}{S}\left(\sum_{k=1}^{s/N}\boldsymbol{X}_k^i+\sum_{k=s/N+1}^{2s/N}\boldsymbol{X}_k^i+\cdots+\sum_{k=(N-1)s/N+1}^{s}\boldsymbol{X}_k^i\right)\\ *\left(\sum_{k=1}^{s/N}\boldsymbol{X}_k^j+\sum_{k=s/N+1}^{2s/N}\boldsymbol{X}_k^j+\cdots+\sum_{k=(N-1)s/N+1}^{s}\boldsymbol{X}_k^j\right)\end{array}\right] \tag{5.16}$$

该变形将协方差计算转换为向量加法规约运算（∑X）和向量内积运算（∑XY），向量加法规约和向量内积运算存在天然的可分割性。利用这种可分割性，可以设计出分布式协方差计算方法，如图 5.2 所示，各进程分别计算向量加法规约和向量内积子任务，接着将子结果发送给 root 进程，root 进程汇总所有子结果并计算得到协方差值。整个过程仅需一次通信，且能使用 MPI_Reduce 集合通信减少通信开销。

本节设计了一种分布式的高光谱影像矩阵的协方差矩阵并行计算方法：1）将高光谱影像矩阵按像元数方向进行均匀分割，子数据分布存储在各进程；2）各进程并行计算向量加和下三角矩阵乘（协方差矩阵是对称阵，只需要计算下三角矩阵），计算得到子结果：和向量（B）、乘积矩阵（B×B）；3）利用集合通信函数，将所有进程的子结果累加到 root 进程；4）最后由 root 进程完成最后的协方差矩阵运算。

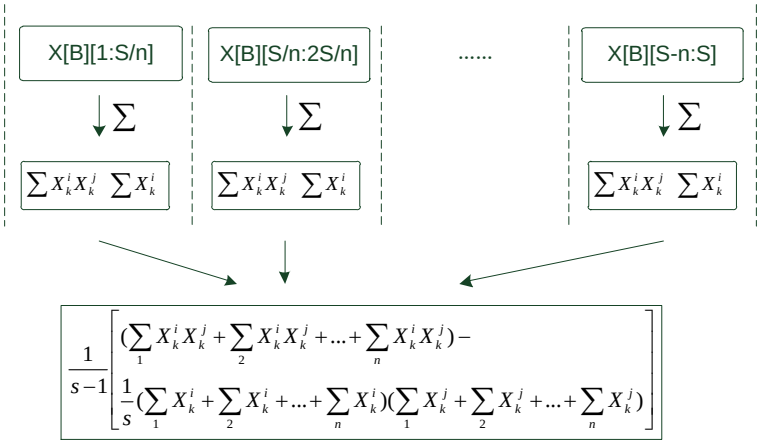


图 5.2 分布式协方差计算

5.4.1.2 共享存储协方差矩阵并行计算

在分布存储协方差矩阵并行计算方案的基础上，节点内计算向量加和下三角矩阵乘法的过程中仍存在两个并行层次：子结果（和向量、乘积矩阵）中每个元素的计算都是独立的，可以并行计算；子结果向量/矩阵中单个元素的计算量较大，亦可以并行计算。此时为了更好地开发 CPU 和 MIC 的向量化功能，预留出第二个并行层次给编译器进行向量化优化（即 SIMD 指令级并行），将粒度较粗的第一层均匀划分任务后绑定到不同处理器核。利用这种方法，可以有效地实现共享存

储的协方差矩阵并行计算。

5.4.1.3 协方差矩阵计算的 GPU 映射

协方差矩阵是对称阵，仅需计算其上三角阵元素，再将值填入对应的下三角阵。针对协方差矩阵计算过程，笔者在成果[6]中设计了 3 种 GPU 映射方案，并对比了加速效果。本文在其中第 I 种 GPU 映射方案的基础上进行了深入的性能优化研究，并获得了最佳优化效果（见 5.5.1 节），下面详细描述该协方差矩阵计算 GPU 映射方案。

令 GPU 上的一个 block 完成协方差矩阵的一个协方差计算任务，如图 5.3，启动 B×B 个 block 用于计算协方差，其中仅上三角（或下三角）的 block 进行运算。每个 block 中的 thread 数量可根据 occupancy 选取最佳的 thread 配置。核心思想是将协方差矩阵中单个协方差的计算任务映射到对应的 block 中，由一个 block 完成一个协方差元素的计算。从实现的角度，有两种实现策略：（1）每次启动维度为<<<i,blockDim>>>（i 为循环变量）的 kernel 函数，循环 B 次完成所有协方差矩阵元素的计算任务；（2）一次性启动所有（B×B）的 block，即启动<<<(B,B,1) , blockDim>>>的 kernel 函数，完成协方差矩阵计算。

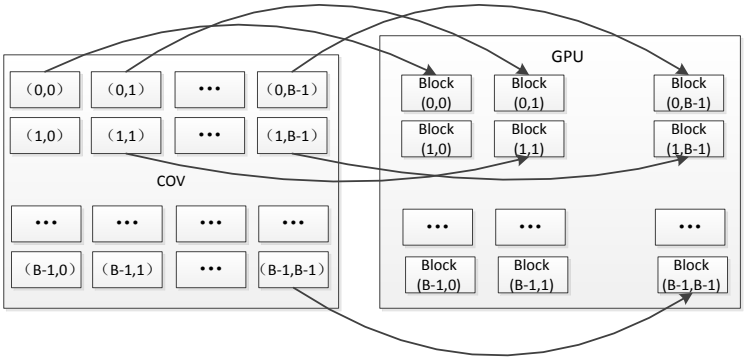


图 5.3 协方差矩阵的 GPU 映射方案

5.4.2 并行 PCA/ICA/MNF 变换与白化处理

由于 PCA 变换、白化处理、ICA 变换和 MNF 变换的计算过程类似，且参与运算的矩阵和数据形式也类似，故可归为同一类问题统一研究相应的并行方案和优化策略。下面以 PCA 变换为例，进行并行方案设计。

5.4.2.1 分布/共享存储的并行 PCA 变换

PCA 变换的计算公式为公式 5.3，其计算过程与矩阵乘法类似，即 $\boldsymbol{Y}=\boldsymbol{A}\boldsymbol{X}=\boldsymbol{V}^{\mathrm{T}}\boldsymbol{X}$。这里有别于一般的矩阵乘法，PCA 变换过程中，排序后的特征向量矩阵 $\boldsymbol{V}$ 的大小为 B×m，其中 B 为高光谱影像数据的波段数，一般为数百左右，m 为选取的主成分数量，一般为十几到几十不等；$\boldsymbol{X}$ 为 B 行 S 列的高光谱影像数

据矩阵，其中 S 为每个波段的像元个数，可达成百上千万。

根据 $\boldsymbol{V}$ 矩阵（m×B）和 $\boldsymbol{X}$ 矩阵（B×S）的维度特点，m<B<S，提出本文的并行 PCA 变换方案，如图 5.4 所示。此时存在两个层次的并行可待挖掘，首先是特征向量矩阵中的 m 行可均匀切分，其次是高光谱影像数据 $\boldsymbol{X}$ 的 S 列可分割。

在分布式并行层次，选择高光谱影像数据 $\boldsymbol{X}$ 的 S 列进行等分，将计算任务分配给不同的进程，各进程完成 $\boldsymbol{V}^{\mathrm{T}}\boldsymbol{X}_{\mathrm{i}}$ 的计算。最终的结果传输到 root 进程统一处理。

在共享存储并行层次，从特征向量矩阵的 m 行进行分割，每个线程计算 $\frac{\mathrm{m}}{\mathrm{n}}$ 行的特征向量矩阵一行与高光谱影像所有列的向量内积，由于 m 一般为十几到几十不等，相对于当前的 Xeon CPU 核数，亦能提供足够的并行度。

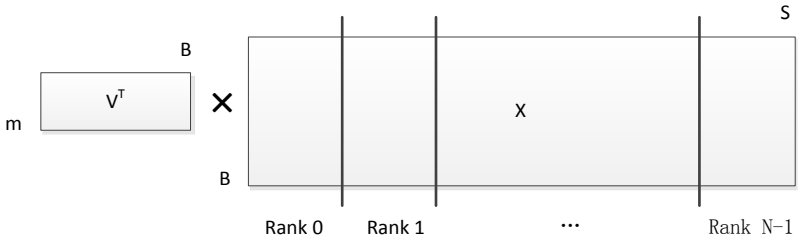


图 5.4 PCA 并行变换方案

5.4.2.2 GPU 上并行 PCA 变换

PCA 变换本质上是矩阵乘法运算，本节设计其 GPU 映射方案，而不直接调用 CUBLAS 库。一方面由于 PCA 变换运算变量的数据类型为 uchar，CUBLAS 不支持；另一方面若转换为 float 将引入额外开销，而且导致数据量的几何增长（是原来的 4 倍），甚至可能导致显存空间不足。

在 PCA 变换（$\boldsymbol{Y}=\boldsymbol{AX}$）中，$\boldsymbol{A}$ 是 B×m 的矩阵，$\boldsymbol{X}$ 的尺寸为 S×B，结果矩阵 $\boldsymbol{Y}$ 大小为 S×m。结果矩阵的每个元素的计算任务均为 $\boldsymbol{A}$ 矩阵一行和 $\boldsymbol{X}$ 矩阵一列的向量内积，相互间是相对独立的。高光谱影像的波段数 B 一般为数百，即向量内积的运算量（B）有限，是细粒度任务，故可将结果矩阵的单个元素的计算视为单位任务，将其映射到 GPU 的 thread 上。

整个结果矩阵共有 S×m 个单位任务，需要 S×m 个线程参与计算；由于高光谱影像的宽 W 和高 H 较大（成百上千），其乘积 S=W×H 可达成百上千万，而 S×m 的数量级更大，超过 GPU 一次可启动的最大线程数量上限，无法一次性直接运算。此时，采用循环计算的方法，即启动适当数量的 block 和 thread，多次循环完成计算任务。该方案中，gridDim 和 blockDim 不受算法自身约束，可根据 occupancy 自由设定，以最大化 GPU 设备占用率，从而获得最佳性能。

5.4.3 ICA 并行迭代

5.4.3.1 分布/共享存储的 ICA 并行迭代

ICA 迭代过程中，受数据依赖的限制，不同的迭代次数之间无法同时运算，只能在单次迭代中开发并行。ICA 迭代的 5 个步骤中，计算量主要集中在 Step5.2 式（5.6）。该运算比较复杂，先抽象得到宏观模型，然后针对该模型进行并行任务分割，得到 ICA 迭代并行方案（图 5.5）。其中中像元数目 S 值最大，其他参数值均较小，故分布存储和共享存储的并行方案类似，都是对像元数 S 进行均匀分割，其中共享存储的分割层次可嵌套在分布存储的并行层次中。

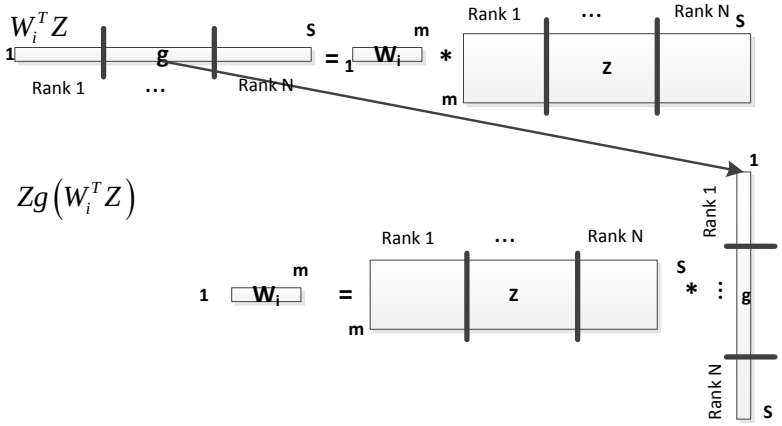


图 5.5 ICA 迭代并行方案

分布存储 ICA 迭代并行方案中，包含 3 次消息通信：root 进程初始化 $\boldsymbol{W}_{\mathrm{i}}$ 后广播给所有进程；各进程计算得到 $\boldsymbol{Zg}\left(\boldsymbol{W}_i^{\mathrm{T}}\boldsymbol{Z}\right)$ 中间结果后收集到 root 进程进行归约运算；root 进程算得归约结果后将其广播给所有进程进行下一步运算。其中第二和第三次消息通信可以用 MPI_Allreduce()函数进行合并通信来提升通信性能。

5.4.3.2 ICA 迭代的 GPU 移植

对比单次 ICA 迭代中各个步骤的运算量，初始化、正交化、归一化和判断操作的运算量较小，主要的计算量集中在 Step5.2 式（5.6），重点考虑该式的 GPU 移植。其中，$\mathrm{W_i^TZ}$ 运算类似于白化处理/PCA 变换，在 GPU 任务映射时，可借鉴 PCA 变换的 GPU 映射方案，即启动<<<gridDim, blockDim>>>的 kernel 函数进行循环计算。$\mathrm{Zg\left(W_i^TZ\right)}$ 运算的特点是向量内积中的向量元素较多，单次计算量较大，向量运算次数较少，故可将一个向量内积运算视为大任务，通过 kernel 函数映射到 GPU 的 grid，循环处理并结合归约，完成 $\mathrm{Zg\left(W_i^TZ\right)}$ 的计算。线程 threadIdx 需循环处理索引为((blockIdx×blockDim+threadIdx)+gridDim×blockDim×i)的元素乘加运算，其中 i 为循环次数，共可得到 gridDim×blockDim 个中间结果；接着在

block 内进行归约，得到 gridDim 个中间结果；由于 GPU 的 block 间无法同步，此时还需要启动一个<<<1, gridDim >>>的 kernel 函数来完成后续的归约计算，最终得到长向量的内积结果。

5.4.4 并行噪声估计

5.4.4.1 分布/共享存储的并行噪声估计

噪声估计是对图像像素点进行滤波处理以得到相应的噪声估计值，该处理需要目标点及周围 8 个像元点的数据。图像各点的噪声估计是相互独立的，不存在相互依赖，因此图像中每个像元点能并行计算，且高光谱影像不同波段的噪声估计也能并行计算。丰富的并行度使其不仅适合多核 CPU 上并行计算，更适合在众核协处理器 GPU 或 MIC 上进行并行运算。

在分布存储系统中，由于噪声估计计算时需要目标点及其周围 8 个像元点数据，故每个进程需要持有本进程的像元点数据外，还需要持有边界之外一圈的像元数据。为了方便，将高光谱影像数据按其高（H）或宽（W）进行均匀划分，每个节点除了本进程像元数据外，还包含分割线外一行或一列的数据。这样可避免进程间的数据通信开销，在噪声估计过程中，各进程无需通信即可完成滤波计算。

在共享存储系统中开发噪声估计的并行计算相对简单，根据前文分析，将图像各点的噪声估计计算均匀分配给各处理器核，即能实现共享存储的并行噪声估计。

5.4.4.2 噪声估计的 GPU 并行映射方案

由于图像各点的噪声估计相互独立，无相互依赖，故高光谱影像的所有像元点都能并行计算，包括不同波段。GPU 的每个 thread 可对应一个像元，计算其噪声估计值。GPU 上每个线程完成高光谱矩阵中一个像元的噪声估计任务，边界噪声估计值事先置零，不参与映射和计算。该映射方案在实现时有两种线程配置策略，见图 5.6 和图 5.7，下面分别展开叙述。

图 5.6 将高光谱影像一行的噪声估计任务映射到一个 block 中，而 block 中的 thread 处理该行中对应像元的噪声估计。此时需要启动 H-2 个 block，每个 block 启动 W-2 个 thread，将噪声矩阵非边界元素的滤波任务映射到相应的线程中进行计算，通过循环处理的方式，迭代 B 次完成所有波段的噪声估计任务。

图 5.7 展示了另一种映射的线程配置策略，其中高光谱影像的噪声估计任务被均匀划分为一系列的小矩阵，每个矩阵的噪声估计任务映射到 GPU 上的 block，此时每个 block 需要的数据是该矩阵区域及其边界外围一圈的数据。该线程配置策略的优势是具有更好的空间局部性，在使用共享存储进行优化时能获得更好的性能收益。

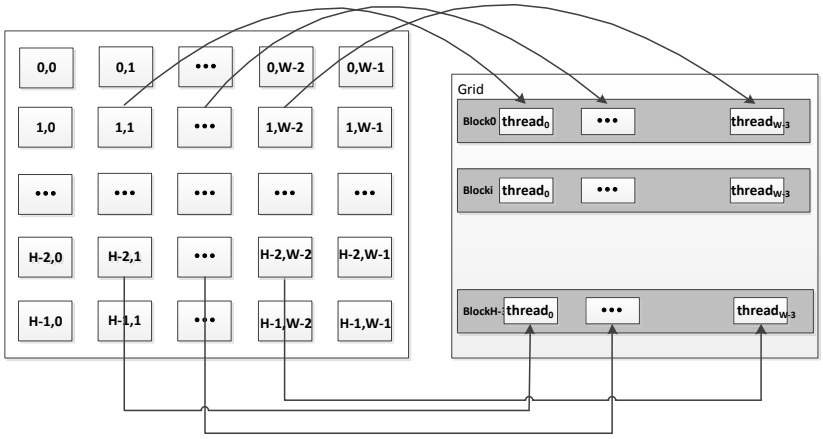


图 5.6 GPU 滤波映射方案 1

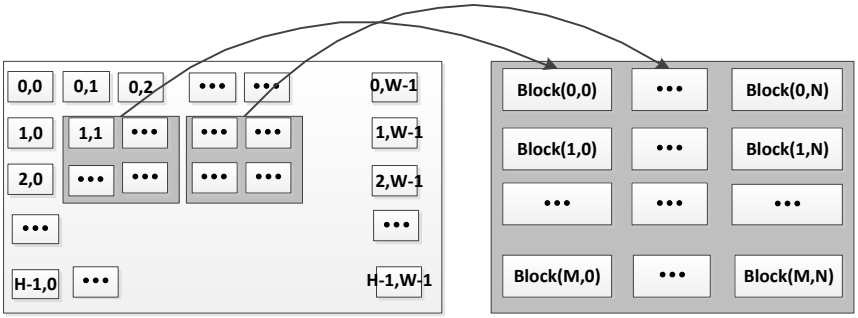


图 5.7 GPU 滤波映射方案 2

5.5 面向众核 GPU 的性能优化策略及效果

本节在前文热点并行方案的基础上，结合第三章的 GPU 访存优化和第四章的异构协同优化，开展一系列性能优化研究，并量化分析优化效果。结果显示，本文第三章和第四章提出的优化方法和策略均正确、有效，能够在并行所获得性能提升的基础上，继续加速数倍乃至数十倍不等。

5.5.1 协方差矩阵计算优化

本节在 5.4.1.3 节的协方差矩阵计算的 GPU 映射方案基础上展开优化研究。针对协方差矩阵运算，根据 3.7 节设计的 GPU 存储优化策略，结合协方差矩阵计算问题及其在 5.4.1.3 节设计的 GPU 映射方案，采取了包括 G_3、S_5、S_4+S_6 和 S_1 等优化策略（优化策略描述详见 3.7 节）。

（1） 全局存储的合并访存优化（G_3）。从单个线程计算一个协方差的角度看，线程对高光谱影像数据矩阵的访问是按行访问的，即连续访问。但 GPU 优化时需要从 warp 内所有线程的角度看，相邻线程对高光谱影像数据的访问是广播和按列访问的，从 3.5.3 节的测评结果中可知，全局存储支持广播，但按列访问是不

连续的，无法合并访问，将导致巨大的访存开销。此时为了能够让全局存储（高光谱影像数据）的访问能够合并（连续），再额外构建一个矩阵用于存储高光谱影像矩阵的转置，将 warp 线程列为主的数据访问转换到对转置矩阵的按行访问。

（2） 利用共享存储优化复用数据（S_5）。一个 block 的所有 thread 共同计算高光谱影像的一个波段与所有波段的协方差，此时该波段数据是复用的，可以考虑用共享存储优化该波段数据访问。

（3） 调整线程配置策略，以达到最大化共享存储复用的目的（S_4+S_6）。从一个宏观的角度看，协方差矩阵的运算与矩阵乘法类似，其高光谱影像数据存在复用，可利用共享存储进行优化。为了最大化共享存储复用，设计了图 5.8 的并行方案。每次从高光谱影像数据中读取一个小窗口的数据到共享存储进行运算，循环滑动小窗口读取其他数据完成所有运算任务。

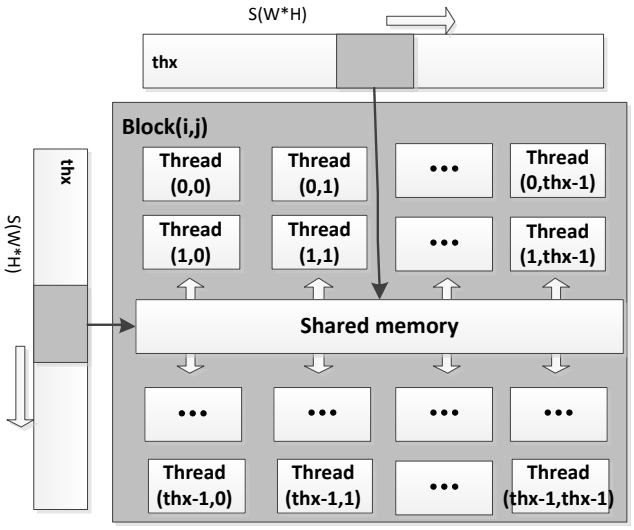


图 5.8 共享存储优化协方差矩阵计算

（4） 避免共享存储的 bank conflict（S_1）。在共享存储的数据读取和访问中，小窗口数据的读取和访问存在 bank conflict，通过转置或数据偏移的方式可以有效减少或避免 bank conflict。这里分别实现了矩阵转置和数据偏移两种方式来避免 bank conflict，其中由于高光谱影像数据的数据类型是 uchar，4 个数据才组成一个 bank，故数据偏移时需要偏移 4 个。

通过实验测试优化前后的 kernel 函数执行时间，见表 5.3。其中 V0 表示未优化，V1~V4 逐步采取了上述 4 种优化策略，V4 采用了矩阵转置的方式避免 bank conflict，V4_1 采用偏移 1 个数据的方式避免 bank conflict，V4_4 采用偏移 4 个数据的方式避免 bank conflict。表中结果显示，从 V0 到 V4 协方差矩阵计算的执行时间不短缩减，说明上述几种优化策略均获得了可观的加速效果。经性能优化后，矩阵转置避免 bank conflict 的方法获得了最佳性能，对比并行未优化版本最高获得 69 倍的加速效果。其中加速效果最显著的是全局存储的合并访存优化，亦说明若

全局存储无法合并访存将导致巨额的访存开销，该结论与 3.5.3 节的测评结果相吻合。

对比表中避免 bank conflict 的 3 种实现方式，其中偏移 1 个 uchar 数据时基本没有性能提升，偏移 4 个 uchar 数据时获得少量性能提升，达到最佳性能的是通过转置来避免 bank conflict 的方式。说明访问对于 uchar 类型数据组成的二维矩阵时，矩阵转置是避免 bank conflict 的有效方式。

表 5.3 协方差矩阵计算优化效果/ms

optimization	V0	V1	V2	V3	V4	V4_1	V4_4
data1	8262.08	774.98	750.53	432.08	269.42	431.86	429.54
data2	24163.80	1692.44	1649.07	938.11	626.61	937.67	934.90
data3	163899.69	6377.08	6181.60	3537.05	2366.40	3540.84	3384.11

5.5.2 PCA/ICA/MNF 变换与白化处理的性能优化

PCA/ICA/MNF 变换&白化处理的实质是矩阵乘法运算，基于 5.4.2 节设计的 GPU 映射方案，结合第三章和第四章的研究成果，选用了 C_6 和 S_5 两项访存优化策略。

（1） 利用常量存储优化变换矩阵的访问（C_6）。变换矩阵的大小为 m*B，其中 m 为选取的主成分数量，一般为十几或几十，本文选用的高光谱影像数据的波段数 B 为 224，故变换矩阵的数据量有限，且访问时均为广播访问。根据第 3 章常量存储访存特性及优化策略，可利用常量存储来优化变换矩阵的访问。

（2） 利用共享存储优化高光谱影像数据的读取（S_5）。PCA 变换过程中，对高光谱影像数据矩阵的访问存在数据复用，故可利用共享存储进行优化。

实现并测试上述优化前后 kernel 函数执行时间，见表 5.4。其中 V0 表示未优化，V1~V2 逐步采取上述两种优化策略。表中数据显示，V0~V2 的 kernel 执行时间逐步下降，说明上述两种优化策略均取得了理想的加速效果，在并行的基础上获得了 2 倍左右加速比。

表 5.4 MNF 变换优化效果/ms

optimization	V0	V1	V2
data1	137.38	88.84	67.99
data2	298.27	200.20	158.67
data3	599.02	413.20	273.15

5.5.3 ICA 迭代的性能优化研究

针对复杂的 ICA 迭代过程，核心步骤的计算占几乎所有的时间，故优先考虑核心计算步骤 Step5.2 式（5.6）的性能优化，接着从整体角度展开优化。分析 ICA 迭代过程中所涉及的数组（矩阵）存储访问，结合第三章提出的 GPU 访存优化框

架和第四章的 Zerocopy 优化方案，设计了以下 5 种优化策略。

（1） 用寄存器优化累计运算的变量访问（R_5）。在 ICA 迭代的核心计算步骤 Step5.2 式（5.6）中，涉及数处累计运算，此时可以不直接累计到结果数组中，而是采用高精度中间变量进行累计，根据 3.6.1 探索得出的私有变量分配策略，该高精度中间变量可以分配在寄存器中快速访问。

（2） 利用共享存储优化复用数据访问（S_5）。核心计算步骤的一些数组被多次访问，并且访问时有读有写，故可采用共享存储优化这些复用数组访问。

（3） 全局存储合并访存优化（G_3）。分析各类数据的访问情况，发现访问 $\boldsymbol{W}$ 矩阵时，warp 内线程访问的是 $\boldsymbol{W}$ 矩阵的列元素（即 W_i 是列向量），此时从 warp 角度看，$\boldsymbol{W}$ 矩阵的访问是不连续的，根据 3.5.3 节的测评结果，该访问非常耗时，故需要进行全局存储的合并访存优化。通过对 $\boldsymbol{W}$ 矩阵的转置处理，使得 warp 内线程访问的数据元素连续存储，即能令全局存储的访存合并。

（4） 合并循环启动的 kernel 函数。ICA 迭代过程包含大量的 kernel 启动函数，特别是由于 block 间无法同步的原因，导致需要 2 个或多个 kernel 函数来完成某一运算，此外还需循环启动 kernel 函数来完成运算，这些将导致可观的 kernel 函数启动开销。通过分析 kernel 函数间的依赖关系，将循环启动的 kernel 函数合并，以减少大量 kernel 函数的启动开销，同时增加单个 kernel 函数的并行度和 GPU 占用率。

（5） 利用 zerocopy 优化少量数据通信，减少全局存储访问。ICA 迭代过程中，CPU 控制迭代流程，GPU 负责计算。GPU 进行迭代计算的起始数据 W_i 是随机生成的，需要在 CPU 上完成，然后传输到 GPU 上进行运算；CPU 判断是否迭代收敛时，需要 GPU 计算结果作为依据，故需要将结果数据从 GPU 传输到 CPU。该过程中有两次必不可少的数据传输，并且传输的数据量较小。此时根据 4.7.1.2 节设计的 zerocopy 优化方案，利用 zerocopy 技术来优化这两个数据通信过程，可以在避免数据通信函数的前提下，减少全局存储的访问。

实现并测试上述优化前后 ICA 迭代过程的单次迭代执行时间，见表 5.5。其中 V0 表示未优化，V1~V5 逐步采取上述 5 种优化策略。表中结果显示，V0~V5 的单次 ICA 迭代执行时间逐步下降，说明上述 5 种优化策略均取得了一定的加速效果。通过一系列优化，在 GPU 并行的基础上再次加速了近 1.8 倍的执行速率。

5.5.4 噪声估计的性能优化研究

噪声估计利用滤波实现，是一类经典的并行运算，在噪声估计的 GPU 映射方案基础上，根据第 3 章 GPU 存储特性及优化策略开展优化研究，采取了 R_3、S_5 和 S_4+S_6 等优化策略。

表 5.5 ICA 迭代优化效果/ms

optimization	V0	V1	V2	V3	V4	V5
data1	4.54	3.86	3.05	2.92	2.53	2.47
data2	7.33	5.63	4.79	4.67	4.01	4.01
data3	10.37	6.97	6.48	6.29	5.91	5.91

（1）利用寄存器替代局部存储访问（R_3）。原始噪声估计使用中值滤波，其中包含排序过程，在 3.6.1 节测定排序是无法分配在寄存器上的；利用均值滤波替代中值滤波，此时复杂的排序运算就被累加和除法等简单运算取代，此时存储可分配在寄存器中，在不影响输出结果的前提下，实现了寄存器替代局部存储的目标。

（2）使用共享存储优化复用的全局存储数据访问（S_5）。滤波的 1 行数据分别被其本行与上下两行的噪声估计运算使用，存在 3 次数据复用，可利用共享存储进行优化。图 5.9 展示了共享存储数据复用优化策略，每个 block 处理一行的噪声估计任务，加载本行与上下两行的数据到共享存储，计算噪声估计时到共享存储访问相应的数据。根据噪声估计的访存特性，共享存储访问无 bank conflict。

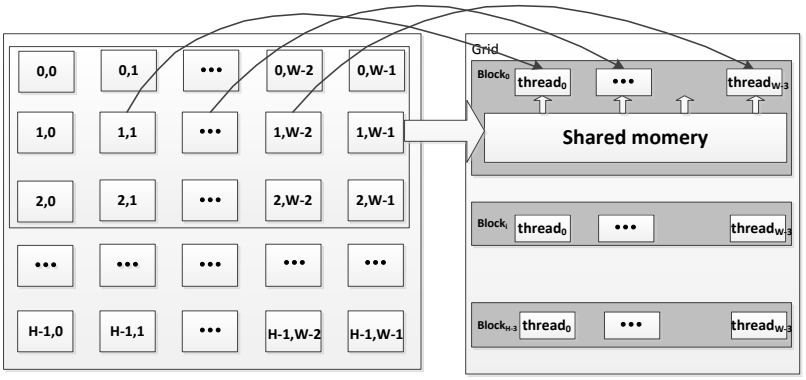


图 5.9 滤波的共享存储优化

（3）调整线程配置策略，以最大化共享存储复用率（S_4+S_6）。对于单个元素，可以被相邻的 8 个元素和其自身的噪声估计使用，最多时可有 9 次复用，而优化方案 2 中仅发挥了 3 此复用，故可进一步优化共享存储复用率。图 5.10 利用分块线程配置的策略来进一步优化噪声估计，此时的 grid 和 block 都按小矩阵分块，每个 block 需要的数据是相应的数据块及其周围一圈数据，此时的中间的数据可被复用 9 次，能够大幅度提升共享存储的数据复用率。

通过实验测试上述优化前后 kernel 函数的执行时间，见表 5.6，其中 V0 表示未优化，V1~V3 逐步采用上述 3 种优化策略。表中数据显示，从 V0 到 V3 的 kernel 函数执行时间逐步减少，说明这几种优化策略均取得了理想的优化效果。经上述优化，相比原始并行映射版本可最高加速 6.9 倍。

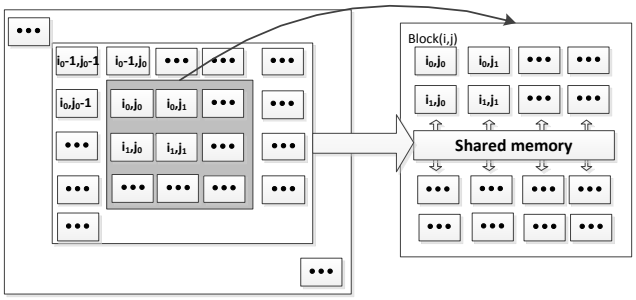


图 5.10 最大化共享存储复用

表 5.6 噪声估计优化效果/ms

optimization	V0	V1	V2	V3
data1	123.38	25.96	22.00	20.88
data2	254.08	57.89	55.55	44.97
data3	1011.15	221.36	206.18	168.59

5.6 面向众核体系结构的高光谱影像并行降维框架

基于当前主流的多核（CPU）与众核（GPU、MIC）异构平台，提出面向众核体系结构的高光谱影像并行降维框架，见图 5.11。该框架以高光谱遥感影像数据及其基本参数（W、H、B）为输入，选定降维算法、执行平台和实现方式，以降维结果为输出。

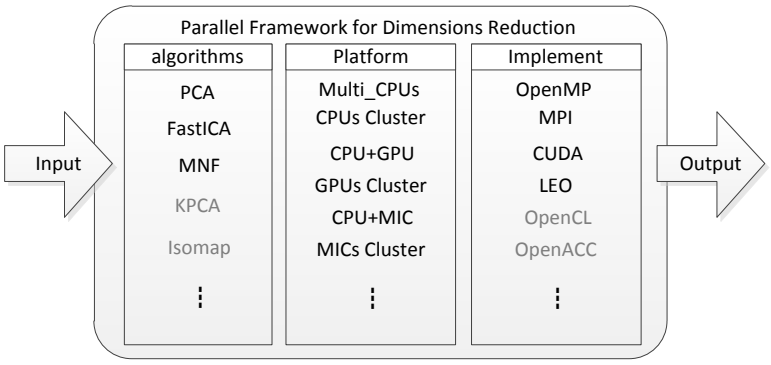


图 5.11 面向众核体系结构的高光谱影像并行降维框架

该框架目前支持的降维算法包括 PCA、FastICA 和 MNF，执行平台包括多核 CPU 节点、CPU 集群、CPU+GPU 异构节点、GPU 集群、CPU+MIC 异构节点、MIC 集群等，实现并行算法采用的并行编程模型有 OpenMP、MPI、CUDA 和 LEO。该并行降维框架具有良好的可扩展性，未来可进一步扩充，比如增加非线性降维算法如 KPCA、Isomap，增加其他执行平台如 DSP、FPGA，亦可进一步利用 OpenCL、OpenACC 等并行编程模型进行实现和优化。

截止本文撰写，并行降维框架利用 OpenMP、MPI、CUDA、LEO 等编程模型，交叉混合，实现了多种不同的并行降维算法：其中 PCA 和 FastICA 降维算法分别在框架所述的 6 种执行平台上均有实现；MNF 降维算法分别在多核 CPU 节点、

CPU+GPU 异构节点和 CPU+MIC 异构节点实现了并行算法。MNF 算法的多节点扩展、其它降维算法和高性能计算平台的移植和优化有待进一步研究。

并行降维框架针对多种经典线性降维算法实现了多种并行版本，可以有效地在各个主流商用高性能计算平台上高速执行降维任务。并行降维框架的高效执行需要硬件层和软件层的支撑，硬件层主要包含多核 CPU、GPU 和 Intel Xeon Phi 等硬件平台，软件层又可分为系统软件、驱动软件、编译器（GCC、ICC、NVCC 等）、编程模型（OpenMP、LEO、CUDA 等）和应用代码。

并行降维框架的并行代码实现时，采用了 5.4 节所述的并行方案和 5.5 节的性能优化策略。框架中与 MIC 相关的并行方案、优化策略及并行降维算法描述参见成果[1,2,3,5,7]。此外，除了 5.5 节所述的热点优化策略外，从整个并行降维算法的层面上看，可结合具体算法流程和第 4 章研究成果选用合适的异构协同优化策略，在此不一一展开论述。下面选取其中 3 个众核并行降维算法展开描述。

5.6.1 基于 GPU 的 PCA 降维算法

针对 PCA 降维算法的加速热点（协方差矩阵计算和 PCA 变换），采用 5.4.1 节的协方差矩阵并行计算方案和 5.4.2 节的并行 PCA 变换方案，并采取 5.5.1 节和 5.5.2 节的 GPU 性能优化策略，结合 PCA 算法流程，利用分段主机端存储选择模型选定主机端存储类型，采取合适的异构协同优化策略，实现基于 GPU 的 PCA 降维算法（G-PCA），见图 5.12。

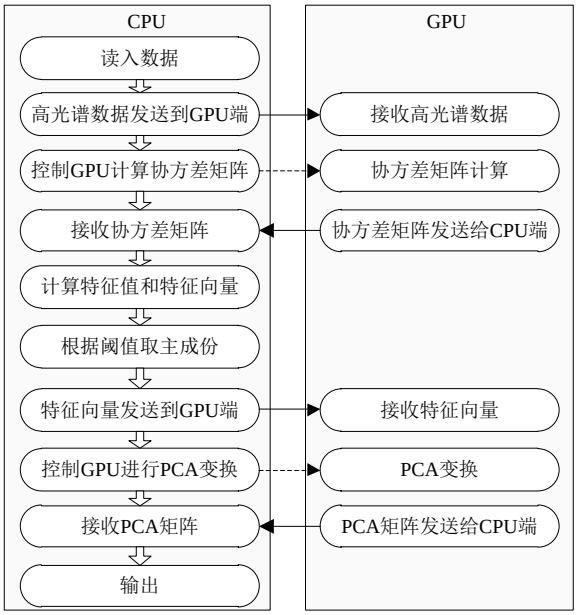


图 5.12 G-PCA 算法流程图

在 G-PCA 并行降维算法中，CPU 从磁盘读取高光谱影像数据到主机端内存，通过 PCI-E 通信将高光谱影像数据传输到 GPU 端（H2D），GPU 完成协方差矩阵

计算，后将协方差矩阵传回 CPU（D2H），CPU 对协方差矩阵进行特征分解，选取主成分，再将特征向量矩阵传到 GPU（H2D），在 GPU 上进行 PCA 变换，变换结果传回 CPU（D2H）并输出到磁盘。上述过程中，涉及通信的数据分配空间选择主机端存储类型时采取分段主机端存储选择模型进行计算，4.6 节详细阐述了 PCA 算法中的主机端存储选择模型计算和实验验证内容。

5.6.2 基于 GPUs 集群的 FastICA 降维算法

结合 5.4.1.1 节的分布存储协方差矩阵并行计算方案、5.4.1.3 节协方差矩阵计算的 GPU 映射方案，5.4.2.1 节分布存储的并行白化处理/ICA 变换方案、5.4.2.2 节 GPU 上白化处理/ICA 变换并行映射方案，5.4.3.1 节分布存储 ICA 并行迭代方案、5.4.3.2 节 ICA 迭代的 GPU 移植方案，利用 MPI 和 CUDA 实现基于 GPUs 集群的 FastICA 并行降维算法（Gs-FastICA），见图 5.13。

Gs-FastICA 并行算法采取了 5.5.1~5.5.3 节所提出的性能优化策略，大幅度优化并行算法性能。此外，对 Gs-FastICA 算法中涉及的 CPU 与 GPU 通信的数据进行量化分析，利用分段主机端存储选择模型进行分析，选择合适的主机端存储类型。对于节点间的数据通信，尽量采用集合通信来优化 MPI 传输开销。

在 Gs-FastICA 并行降维算法中，每个节点读取本节点计算所需要的高光谱影像数据；将高光谱数据传输到 GPU（H2D），在 GPU 上计算ΣX 和ΣXY（协方差计算的分解，并行方案与优化策略与协方差矩阵计算类似），结果传回 CPU（D2H），再通过 MPI 通信将数据收集到 root 节点计算协方差矩阵；接着，root 节点对协方差矩阵进行特征分解，计算白化矩阵，并将相关数据通过 MPI 广播到所有进程；各进程收到广播数据并传输到 GPU（H2D），由 GPU 完成白化处理；ICA 迭代开始后，root 节点初始化 W_i 并广播给所有进程，各进程接收到 W_i 后利用 GPU 开始本轮迭代，直到符合收敛条件，迭代过程中需要两次节点间通信；完成 ICA 迭代后，各节点利用 GPU 完成 ICA 变换，并将变换结果传输回 CPU 端（D2H），并输出至磁盘。

5.6.3 基于 GPU 的 MNF 降维算法

本节设计了基于 GPU 的 MNF 降维算法（G-MNF），如图 5.14 所示。该并行降维算法集成了 5.4.1.3 节的协方差矩阵计算 GPU 映射方案、5.4.2.2 节 GPU 上并行 MNF 变换方案和 5.4.4.2 节噪声估计（滤波）的 GPU 映射方案。算法采纳了 5.5.1 节协方差矩阵计算的性能优化策略、5.5.2 节 MNF 变换的性能优化策略和 5.5.4 节噪声估计的性能优化策略。此外，利用分段主机端存储选择模型计算 MNF 算法中涉及通信的数据的预测时间，选择合适的主机端存储类型。

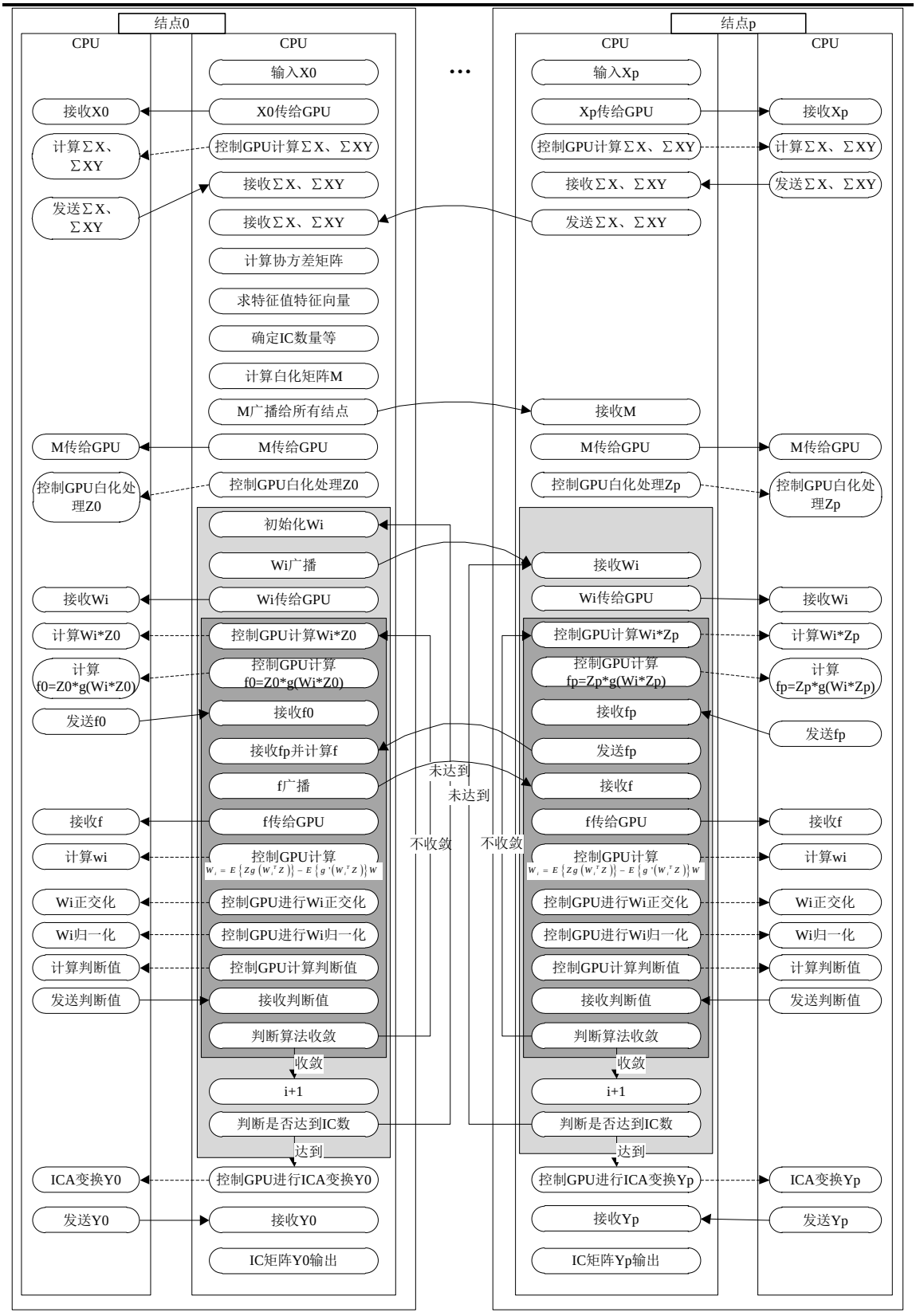


图 5.13 Gs-FastICA 算法流程图

G-MNF 并行降维算法中，CPU 读入高光谱影像数据并发送到 GPU（H2D），

GPU 计算噪声估计（滤波）及噪声协方差矩阵，发送噪声协方差矩阵到 CPU 端（D2H）；CPU 对噪声协方差矩阵进行特征分解，并计算变换矩阵 $\boldsymbol{P}$，于此同时，GPU 计算高光谱影像的协方差矩阵，CPU 与 GPU 的计算过程可同时执行，见图中虚线框部分；接着 GPU 将原始高光谱协方差矩阵传回 CPU 端（D2H），由 CPU 完成变换并进行特征分解，计算 MNF 的变换矩阵 $\boldsymbol{T}$；发送变换矩阵 $\boldsymbol{T}$ 到 GPU 端（H2D），由 GPU 进行 MNF 变换，将变换结果传回 CPU 端（D2H），并输出到磁盘。算法流程中包含了计算（CPU）与计算（GPU）重叠的思想，可有效地重叠耗时较少的部件的开销。

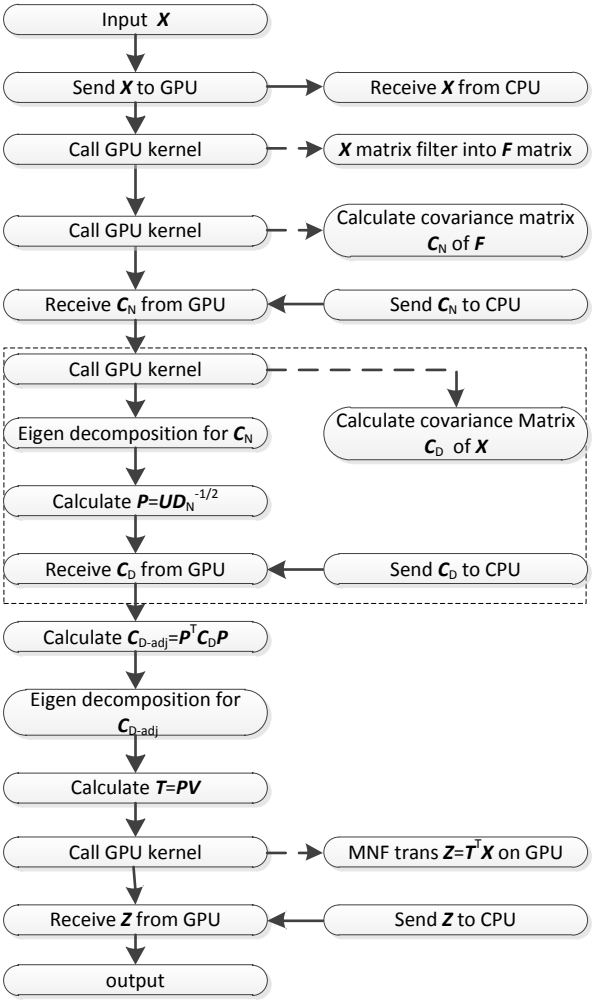


图 5.14 G-MNF 算法流程图

5.7 实验结果与分析

5.7.1 实验准备

本文实验平台的软硬件参数已在 2.4.2 节详细介绍，不再赘述。需要声明的是实验平台中仅有 1 个 GPU 节点（搭载两个 K20c GPU）和 4 个普通节点（其他配

置与 GPU 节点一致），故本章的可扩展性分析仅能扩展到 4 个节点 64 核心。

本节实验准备部分主要介绍本文降维所用到的高光谱影像数据、ICA 迭代次数的声明、以及本节分析的面向众核体系结构的高光谱影像降维框架中的并行降维算法。

高光谱影像数据：表 5.7 罗列了本文实验采用的 3 组 AVIRIS 高光谱影像数据信息。原始高光谱影像数据经过预处理，作为面向众核体系结构的高光谱影像降维并行框架的输入数据。预处理主要包括两项关键内容：光谱信息转换为像素信息；重新组织高光谱影像矩阵的数据存储位置，使其适应并行降维算法，以获得更高处理性能。

表 5.7 高光谱遥感图像数据信息

Data	Width	Height	Band
data1	614	1087	224
data2	753	1924	224
data3	781	6955	224

ICA 迭代次数声明：由于高光谱影像数据和 ICA 迭代过程中初始化的随机数不同，导致 ICA 迭代次数不同，不同的迭代次数将导致时间消耗不相等。为了更准确地对比并行算法性能，本文约定：同一数据取固定的迭代次数。通过多次实验求平均值的方法，确定各组数据的迭代次数（表 5.8）。

表 5.8 高光谱影像数据迭代次数

Data	data1	data2	data3
Iterations	468	581	295

并行降维框架中的并行算法：面向众核体系结构的高光谱影像降维框架中实现了数量众多的并行算法，在此选取与本章主题紧密相关的十数种并行算法作为代表（详见表 5.9），进行实验测试，分析其加速比和可扩展性。其他并行算法的实验和分析可参见成果[1,2,3,5,7]。

表 5.9 实验对比的并行降维算法

降维算法	PCA	FastICA	MNF
单节点	O-PCA	O-FastICA	O-MNF
	G-PCA	G-FastICA	G-MNF
多节点	M-PCA	M-FastICA	
	MO-PCA	MO-FastICA	
	Gs-PCA	Gs-FastICA	

表 5.9 中，“O”表示单节点内利用 OpenMP 实现的多核 CPU 并行降维算法，例如 O-PCA 是指共享存储的 PCA 并行降维算法；“G”表示基于 CPU+GPU 异构节点的并行降维算法，例如 G-MNF 就是基于 CPU+GPU 异构平台的 MNF 降维算法；“M”指利用 MPI 实现的分布存储并行降维算法；“MO”表示利用 MPI 和 OpenMP 实现的混合并行降维算法，利用 MPI 处理进程级分布存储并行层次，

OpenMP 处理线程级共享存储并行层次，结合 SIMD 指令级并行实现。“Gs”表示基于 GPUs 集群的并行降维算法，利用 MPI、OpenMP 和 CUDA 混合实现。

5.7.2 并行算法加速比分析

5.7.2.1 PCA 并行降维算法加速比

在实验平台上运行面向众核体系结构的高性能影像并行降维框架中的数种 PCA 降维算法，统计时间并计算并行算法相对串行算法的加速比，见表 5.10。表中数据显示，这些并行算法均获得了较为理想的加速比，其中 Gs-PCA 并行算法利用 2 个 K20c GPU 获得最高 119.7 倍加速比。

表 5.10 PCA 并行降维算法加速比

version	data1	data2	data3
O-PCA	7.5	6.2	7.0
M-PCA(1 node)	18.5	20.1	22.8
M-PCA(4 nodes)	46.3	61.0	81.9
MO-PCA(4 nodes)	31.0	38.8	35.6
G-PCA	40.8	50.7	68.3
Gs-PCA(2 GPU)	57.1	79.2	119.7

在单个节点上，O-PCA 可加速 6.2~7.5 倍，一个节点上 M-PCA 可加速 18.5~22.8 倍。相对 O-PCA，M-PCA 性能更好，可能原因有：（1）5.4.1.1 节的分布存储协方差矩阵并行方案中，通过协方差计算公式变换，一定程度上起到了减少运算量和简化运算过程的效果；（2）通过多进程划分后，单个进程中的数据访问 cache 命中率更好；（3）M-PCA 算法对通信过程进行了大幅优化，通信开销表现不明显；（4）为了令串行程序高效执行，做了一些优化改造，使其可向量化、访存连续，但同时也增加了访存总量，在串行算法中访存带宽可以支撑单线程访存，而在多线程并行算法中，存储带宽无法满足所有 GPU 核心的访存需求，导致访存带宽资源竞争，性能无法达到理想水平。同理，从 4 个节点分析，M-PCA 比 MO-PCA 性能更佳，可能原因亦是如此。

另外，G-PCA 和 2 个 GPU 上执行的 Gs-PCA 算法亦运行在单个节点上，两者的加速比均远高于 CPU 并行版本。对比多核 CPU 并行，众核 GPU 协处理器的性能优势更加显著。

5.7.2.2 FastICA 并行降维算法加速比

在本文实验平台上执行高光谱影像并行降维框架中的数种 FastICA 并行降维算法，统计运行时间并计算并行算法的加速比，见表 5.11。表中数据显示，6 类并行算法均获得了良好的性能提升，其中对于高光谱影像数据 data1 和 data2，在 4 节点上的 M-FastICA 降维算法取得最佳性能；对于高光谱影像数据 data3，

Gs-FastICA 并行算法在搭载 2 个 K20c GPU 的节点上最高加速 106.6 倍。

表 5.11 FastICA 并行降维算法加速比

version	data1	data2	data3
O-FastICA	5.5	6.1	6.9
M-FastICA(1 node)	11.9	15.8	15.8
M-FastICA(4 nodes)	49.6	66.8	69.6
MO-fastICA(4 nodes)	26.9	30.8	30.6
G-FastICA	22.6	40.2	59.3
Gs-FastICA	28.9	58.7	106.6

在单个节点上，O-FastICA 加速 5.5~6.9 倍，而 M-FastICA 加速 11.9~15.8 倍。基于分布存储的并行算法比共享存储的算法性能更好，除了在上一节中分析的原因外，还有一个原因可能是：在迭代过程中，每次迭代的运算量不是很大且运算比较复杂，在利用 OpenMP 实现共享存储并行算法时，需要多次启动 fork-join 并行区域，这在一定程度上降低了并行效率。同理，4 个节点上的 M-FastICA 的加速比比 MO-FastICA 的加速比更高。

在异构平台上，G-FastICA 算法与 Gs-FastICA 算法的性能表现均不错，与单节点的 CPUs 并行算法相比，基于 GPU 的并行算法的性能更加优秀。Gs-FastICA 在 2 个 K20c GPU 上亦获得比 G-FastICA 算法更好的性能，且在处理高光谱数据 data3 时，获得了最高 106.6 倍加速比。

5.7.2.3 MNF 并行降维算法加速比

并行降维框架中的 MNF 降维算法仅实现了 O-MNF 和 G-MNF 两种并行算法，在本文实验平台上执行并对比这两种并行算法加速比（表 5.12）。表中数据显示，两种并行算法均获得了理想的的加速比。其中 O-MNF 算法相对串行算法加速 13.2~14.1 倍，而 G-MNF 算法利用 K20c GPU 可获得 64.1~86.9 倍加速比。

表 5.12 MNF 并行降维算法加速比

version	data1	data2	data3
O-MNF	13.2	13.8	14.1
G-MNF	64.1	76.4	86.9

5.7.3 可扩展性分析

本节的可扩展性分析主要分析算法的强可扩展性，由于高光谱影像降维算法的特性，不适合分析算法的弱可扩展性。主要原因有：（1）若利用某一高光谱数据成比例放大，则其协方差矩阵和最终取得的主成分数量均不相等，导致运算量不可对比，特别是 ICA 迭代过程的运算量更为复杂多变；（2）若采用不同的成比例数据，数据不同取到的主成分数量不相同，迭代次数不相等，总运算量亦无法比较。

本节可扩展性分析主要从 3 个方向展开：（1）多 GPU 平台的可扩展性，分别分析 Gs-PCA 和 Gs-FastICA 两种算法的可扩展性；（2）基于 MPI 的 M-PCA 算法和 M-FastICA 算法的可扩展性；（3）基于 MPI+OpenMP 的 MO-PCA 和 MO-FastICA 算法的可扩展性。

此外，在成果[1,2]中，笔者在天河 2 号超级计算机平台上，对 MO-FastICA（基于 CPU 集群的 FastICA 降维算法）、Ps-FastICA（基于 Phi/MIC 集群的 FastICA 降维算法）进行了可扩展性分析，其中 MO-FastICA 的可扩展至 16 节点，而 Ps-FastICA 可在 64 节点 192 个 Phi 上获得最佳性能。

5.7.3.1 Gs-PCA/Gs-FastICA 的可扩展性

由于实验平台 GPU 资源的限制，仅有 2 个 GPU 且安装在同一节点上，故 Gs-PCA 和 Gs-FastICA 两种并行算法仅能扩展到 2 个 GPU。实验测试两个 GPU 的“Gs-”算法和单个 GPU 的“G-”算法执行时间，并计算“Gs-”算法相对于“G-”算法的加速比（表 5.13）。表中数据显示，“Gs-”算法性能均优于“G-”算法，且随着数据规模扩大，其加速比亦逐步增长，说明并行算法的多 GPU 可扩展性良好。对高光谱数据 data3 降维时，Gs-PCA 和 Gs-FastICA 对比单 GPU 版本均获得了 1.8 倍加速。

表 5.13 Gs-PCA/Gs-FastICA 的可扩展性

data	data1	data2	data3
G-PCA/Gs-PCA	1.4	1.6	1.8
G-FastICA/Gs-FastICA	1.3	1.5	1.8

5.7.3.2 M-PCA/M-FastICA 的可扩展性

本文实验平台仅有 4 个相同配置的 CPU 节点，故本节可扩展性分析仅能达到 4 节点 64 核心。在实验平台上，设定进程数量从 1 到 64，分别测定 M-PCA 和 M-FastICA 的执行时间（图 5.15 和图 5.16）。图中数据显示，当进程数量增加时，执行时间呈现近线性下降趋势，说明 M-PCA 和 M-FastICA 算法的可扩展性非常好。

5.7.3.3 MO-PCA/MO-FastICA 的可扩展性

分别在 1~4 个节点上执行 MO-PCA/MO-FastICA 算法，统计相应的执行时间（图 5.17 和图 5.18）。图中数据显示，随着节点数量增加，执行时间总体呈下降趋势。但下降趋势随节点数量增加而变得相对平缓，当数据量较小时变平缓的过程更加迅速，比如图 5.17 的 data1 曲线。可能原因是，节点数量增加后，单个节点分配的运算量减少，此时节点内的 OpenMP 共享存储级并行所能发挥的效率下降，而串行执行时间和并行开销并没有减少，导致程序的整体耗时下降趋势变得相对平缓。

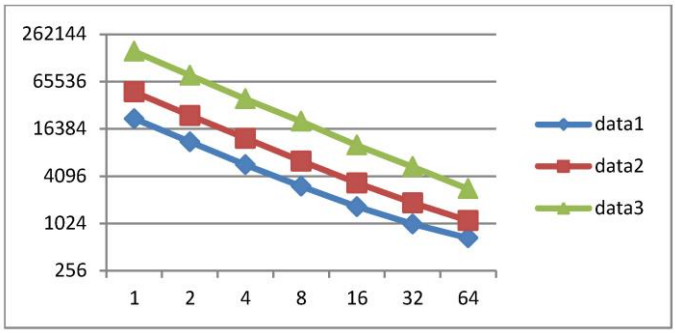


图 5.15 M-PCA 算法可扩展性

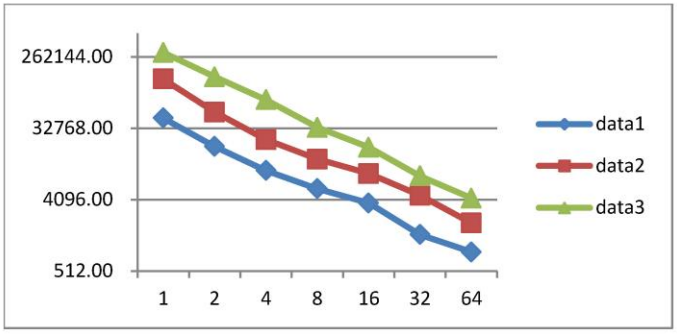


图 5.16 M-FastICA 算法可扩展性

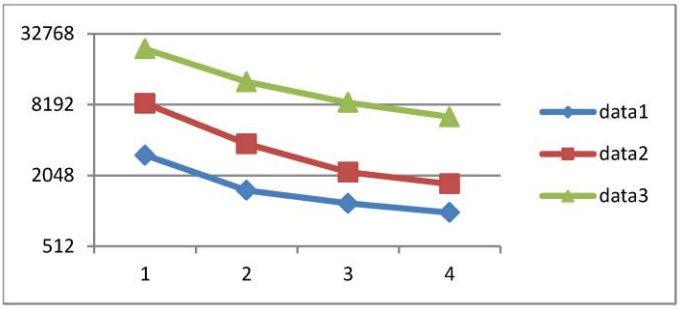


图 5.17 MO-PCA 算法可扩展性

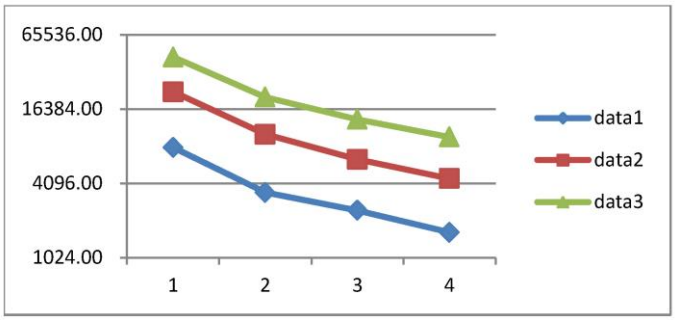


图 5.18 M-FastICA 可扩展性

5.7.4 实验讨论

（1）并行降维框架的可扩展性

面向众核体系结构的高光谱影像并行降维框架具有较好的可扩展性，主要表现在 3 个方面：降维算法的可扩展性、执行平台的可扩展性和实现方式的可扩展

性。

降维算法的可扩展：并行降维框架目前主要支持 PCA、FastICA 和 MNF 三类降维算法，未来可以增加其他降维算法，如 KPCA、Isomap 等。

执行平台的可扩展性：并行降维框架目前适用于 CPU 同构、CPU+GPU 异构和 CPU+MIC 异构等平台，实验结果显示框架中的并行算法均获得了良好的加速效果。如果需要，可在其他高性能计算平台上实现这些降维算法，令其高效执行。

实现方式的可扩展性：并行降维框架中并行算法实现所采用的并行编程模型主要有 OpenMP、MPI、CUDA 和 LEO。未来可以采用其他并行编程实现，比如 OpenCL、OpenACC 等。

（2）并行框架中的算法选择

由加速比结果可知，众核 GPU 相对多核 CPU 有着显著的性能优势，因此在并行降维框架中选择并行算法时，若运行平台允许，应优先选择众核异构并行算法。在并行框架中选择算法时，首先根据实验平台进行选择，若平台搭载 GPU 或 Phi(MIC)等众核协处理器，优先选择相应的异构并行算法；若实验平台为集群时，先根据平台结构选择并行算法，再根据高光谱数据规模搜寻最佳处理节点数量。

5.8 本章小结

本章描述 PCA、FastICA、MNF 三类降维算法，分析算法热点，针对热点提出 3 个并行层次（分布存储、共享存储和 GPU）的并行方案。基于第三章和第四章的研究成果，结合热点并行方案，开展性能优化研究，并量化分析性能优化效果。提出面向众核体系结构的高光谱影像并行降维框架，讨论了框架在降维算法、执行平台和实现方式等方面良好的可扩展性，且框架中实现了大量的并行降维算法。实验结果显示框架中并行降维算法的加速效果良好，其中 Gs-PCA 算法最高加速 119.7 倍，Gs-FastICA 算法最高加速 106.6 倍，G-MNF 算法最高加速 86.9 倍。最后通过实验分析了并行算法良好的可扩展性。

第六章 基于众核 GPU 的声呐信号波束形成算法

上一章将 3~4 章的研究成果应用于高光谱影像降维领域，获得了非常可观的性能收益，本章试图将其应用到另外一个领域——声呐信号波束形成，以获得理想的性能优化效果。

波束形成是声呐信号处理系统的关键组成部分，运算量巨大，实时性要求高。本章利用众核 GPU 加速处理 DFT-CBF 和 MVDR 两类经典的宽频波束形成算法。重点针对 DFT-CBF 算法中的 DFT（discrete fourier transform）变换、CBF/Lofar 计算和频带能量整合统计，以及 MVDR 算法中的 DFT 变换、双边雅克比迭代（特征分解）和方位谱统计等热点，设计并行方案，基于第 3~4 章研究成果设计一系列优化策略并量化分析优化效果，提出基于 GPU 的 DFT-CBF 算法和 3 种并行 MVDR 自适应波束形成算法。实验结果显示基于 GPU 的并行 DFT-CBF 算法能实时处理 16384 基元的声呐信号波束形成，3 种并行 MVDR 自适应波束形成算法亦可获得较理想的加速效果。

6.1 引言

由于水是电的良导体，电磁能极易以热的形式耗散；而声波在水中衰减缓慢，传播距离较远，是水下/海底信号处理的主要信号[105]。声呐是利用水下声波判断海洋中物体的存在、位置及类型的方法和设备[106]。二战时期一些国家的舰艇就开始装备有声呐设备，此后声呐技术在海军中得到了广泛应用[105]。

在军事战争和国家防卫中，必须争分夺秒，对声呐信号处理实时性要求极高。波束形成作为声呐系统的关键部分，涉及大量数据运算，实时的波束形成是实现声呐信号实时处理的关键组成之一。由于众核体系结构的高性能运算优势，利用该类高性能计算平台加速声呐信号处理是实现声呐信号实时处理的重要途径之一。本章将利用 CPU+GPU 异构系统加速处理声呐信号的波束形成算法。

当前舰艇上装备的水听器基阵一般为数百基元，并正朝着上千基元迈进。在石油勘探领域，甚至可能将会使用万基元水听器基阵。随着声呐技术发展，计算能力提升，基阵的基元数量正逐步增长。基元的增长将获得更大的空间增益、更广的监测范围和更高的精细度。预先研究更大规模基阵的声呐波束形成对未来声呐系统的发展有着重要价值。本章在评价并行波束形成算法性能时，通过模拟生成不同规模的声呐信号，来预测并行波束形成算法在未来大规模水听器基阵的处理性能。

本章基于众核 GPU 平台，开发频域常规波束形成（DFT-CBF）算法和最小方

差无畸变响应（MVDR）波束形成算法等两类经典的声呐信号波束形成算法的并行和优化。分析波束形成算法热点，设计并行方案，开展性能优化研究，实现并行波束形成算法，利用实际声呐信号和模拟信号分析并行算法加速比和实时性。

6.2 波束形成概述

6.2.1 基本概念

波束形成对基阵接收到的声呐信号进行加权，形成空间指向性。波束形成可视为空间滤波器，可以令指定的方向信号通过，滤除空间中其他方向的信号干扰和噪声，进而收获较高的空间增益，提高感兴趣信号的信噪比。波束形成技术能有效克服水下、海底复杂环境引发的水声信道的多径干扰和同信道干扰等不利影响。

波束形成的理论基础是不同基元接收的声呐信号相对于基准基元的延迟差。利用波束形成形成空间指向后，可以将基阵接收的阵元域信号转换为波束域信号，进行下一步水下目标定位处理。波束形成的作用是获得良好的信噪比和较高的目标分辨力。

波束形成是现代声呐信号处理的核心组成部分之一，主动声呐和被动声呐都装备有波束形成系统，图 6.1 为声呐接收机架构图，展示了波束形成在声呐信号处理中的位置[105]。由于水声环境中水听器接收的信号信噪比低，导致水声定位精度不高，利用波束形成能达到足够大的信噪比，抵消干扰，从而收获高精度的目标分辨力。

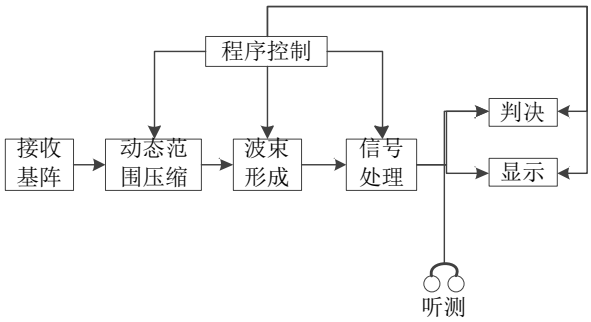


图 6.1 声呐接收机结构图

6.2.2 阵列流型计算

阵列流型是各基元之间的声程差，远场和近场的阵列流型计算各不相同。首先讨论远场的阵列流型计算，远场的声波信号可视为平面波（见图 6.2），对于水听器基阵上的各基元而言，其入射角度 θ 是相等的。选择最中间的基元作为基准

点，此时远场声程差的计算公式为

$$\Delta l_k(\theta) = (k - mid) \times d \times \cos(\theta)，\qquad (6.1)$$

其中 d 为基元间距，θ 为入射信号和基阵的夹角。cos(θ)具有方向性，信号后于基准点达到基元的声程差为负，先于基准点到达的声程差为正。

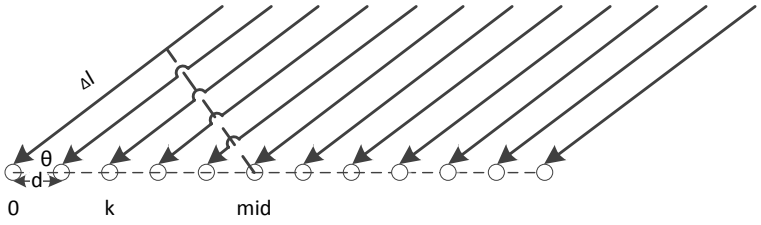


图 6.2 远场阵列流型

近场情况下，由近及远，取 7 组距离，分别为 50 米、100 米、200 米、400 米、800 米、1600 米和 3200 米，分别计算各基元各角度各距离的近场声程，然后与基准点的声程做差计算得到近场声程差。近场声程的计算如图 6.3 所示，对于高度 h 上的任一信号点，其到基准点的声波入射方向和基阵方向的夹角为 θ，其与基准点的水平距离 L。第 k 个基元到基准基元的距离为 D，此时该信号点到第 k 个基元的声程为 l_k，计算公式如下

$$\begin{cases} l_k(\theta) = \sqrt{(D_k - L(\theta))^2 + h^2} & (\theta \neq 90°) \\ l_k(\theta) = \sqrt{(D_k)^2 + h^2} & (\theta = 90°) \end{cases}，\qquad (6.2)$$

$$D_k = (k - mid) \times d，\qquad (6.3)$$

$$L(\theta) = \frac{h}{\tan(\theta)}。\qquad (6.4)$$

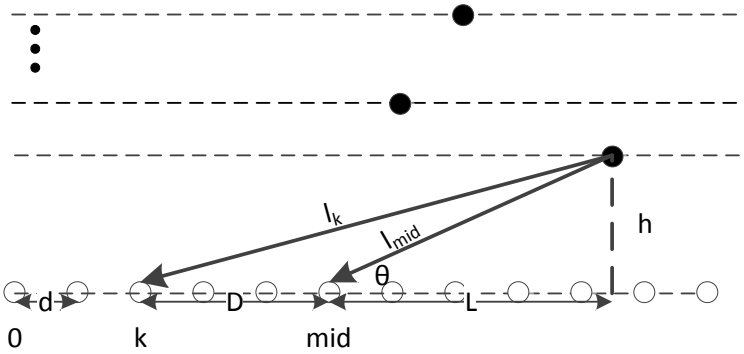


图 6.3 近场阵列流型

其中(k-mid)和 tan(θ)都具有方向性：（1）当 θ<90°，k<mid 时，(k-mid)为负，tan(θ)为正；（2）当 θ>90°，k<mid 时，(k-mid)为负，tan(θ)为负；（3）当 θ<90°，k>mid 时，(k-mid)为正，tan(θ)为正；（4）当 θ>90°，k>mid 时，(k-mid)为正，tan(θ)为负。此时近场声程差的计算公式为

$$\Delta l_k(\theta) = l_{mid}(\theta) - l_k(\theta)。 \quad (6.5)$$

6.3 相关工作

本节从波束形成算法和波束形成算法的并行处理两个方面展开讨论本章的相关工作。

6.3.1 波束形成算法发展与现状

二战时期，常规波束形成（conventional beamforming，CBF）已被广泛应用，CBF 算法具有运算量少、稳健性好等优点，但同时也存在阵增益有限、抗干扰能力差和受“瑞利（Rayleigh）限”限制的缺点[124-125]。Schmit 提出多重信号分类（multiple signal classification, Music）算法，该算法将信号数据分解为信号子空间和噪声子空间，两者相互正交，并利用该特性构造方位谱峰，获得较高的方位分辨力[107]。Roy 等人[108]提出旋转不变子空间（estimation of signal parameters via rotational invariance techniques，ESPRIT）算法，对比 Music 算法大幅降低了计算量。Cox 等人[109]提出对角加载稳定自适应波束形成算法，提高自适应波束形成对基阵误差的稳健性。Capon 提出最小方差无畸变响应（minimum variance distortionless response，MVDR）波束形成算法，在保持波束指向方向的输出信号不变的前提下，最小化阵列总输出功率，意味着最小化阵列输出的噪声和干扰，进而提升干扰抑制能力和方位分辨力[110]。Owsley 等人[111]提出基于子阵划分波束形成算法，对自适应基阵进行降维，降低运算量的同时提高自适应收敛速度。Krolik 等人[112]提出导向最小方差（steered minimum variance，STMV）算法，在保持 MVDR 算法性能的同时，能有效地降低运算量和加快收敛速度。

随着波束形成算法的不断改进和发展，算法的计算量和计算复杂度在逐步增大；且随着声呐技术的发展，基阵规模也在逐步扩大，同时伴随着巨大的计算量提升。因此波束形成的快速处理刻不容缓，基于高性能计算平台加速处理波束形成可以在保持波束形成算法原有输出结果不变的前提下，大幅缩短波束形成处理时间。本章将基于 CPU/GPU 异构平台加速处理两类经典的波束形成算法。

6.3.2 波束形成并行处理发展和现状

波束形成在通信基站、超声波成像和声呐信号处理等领域有广泛应用，本节分别讨论这些领域中波束形成算法的并行处理相关研究工作。基于基站天线模块，Ahmed 等人[113]用 CUDA 加速智能天线系统中自适应波束形成，加速 16.5 倍；Yun 等人[114]在 B4G 基站系统中，用 GPU 加速天线模块的波束形成。超声波成像领域

的波束形成并行化研究较为成熟：Phuong 等人[115]实现实时的 B 超波束形成。Chen 等人[116]基于 GPU 实现了用于超声波成像的实时自适应 MV 波束形成器。Chen 等人[117]利用 FPGA 和 GPU 加速医学超声波系统中的自适应波束形成。Fathi 等人[118]基于 GPU 实现医学超声波成像中的自适应波束形成。Asen 等人[119]基于 GPU 实现 Capan 波束形成，用于 64 分相阵列的心脏超声波实时成像。Kjeldsen 等人[120]在多核平台（CPU、GPU）上加速合成孔径连续波束形成（SASB），实现实时超声波成像。声呐信号处理方面的波束形成并行化研究相对较少：Jin 等人[121]基于多 DSP 系统实现并行的宽频 MVDR 波束形成； Han 等人[122]利用 FPGA 加速 2D 平面声呐波束形成算法，并扩展利用多 FPGA 加速水下 3D 成像。Buskenes 等人[123]在 32 基元的主动声呐系统中利用 GPU 加速 MVDR 波束形成。

上述研究显示，当前声呐系统中波束形成实时处理仍以 DSP 和 FPGA 为主，而这类嵌入式高性能计算平台的应用程序开发周期长、成本过高、兼容性差、可靠性低。利用 GPU 能有效避免上述缺陷并获得超高性能，而相关的研究较少，本文基于 GPU 研究声呐信号波束形成算法的快速处理。

由于大规模基阵强大的空间增益、广阔的监测范围和更高的精细度，未来基阵规模将逐步扩大。上述基于 GPU 的波束形成研究局限于当前或过去的声呐基阵规模，而没有对未来大规模声呐基阵的波束形成进行研究和讨论，本文将扩展讨论大规模基阵波束形成的快速处理，是对现有研究的重要补充。

6.4 基于 GPU 的万基元实时频域常规波束形成

6.4.1 常规波束形成

常规波束形成分为时域常规波束形成和频域常规波束形成，分别从时域和频域两个角度求解常规波束形成算法[124-125]。

假设基阵规模（基元数量）为 B，对每个基元接收的声呐信号进行离散傅里叶变换，第 i 个基元的傅里叶变换结果为 $X_k(Pt)$，

$$\boldsymbol{X}_{\mathrm{k}}(\mathrm{Pt}) = \boldsymbol{A}_{\mathrm{k}}(\mathrm{Pt})\mathrm{e}^{\mathrm{j}\boldsymbol{\varphi}_{\mathrm{k}}(\mathrm{Pt})} \quad , \tag{6.6}$$

其中 $\boldsymbol{\varphi}_{\mathrm{k}}(\mathrm{Pt})$ 是相位分量， $\boldsymbol{A}_{\mathrm{k}}(\mathrm{Pt})$ 是幅度分量。

常规波束形成又称为时延求和波束形成，通过对声呐基阵的所有基元接收信号的时延求和获得输出结果，其输出为

$$Y(\theta, Pt) = \sum_{k=0}^{N-1} w_k(\theta)^{\mathrm{H}} X_k(Pt) = w(\theta)^{\mathrm{H}} X(Pt)\text{。} \tag{6.7}$$

其中，θ 为波束指向角度， $w_k(\theta)$ 是 θ 角度 k 基元的权重。常规波束形成的加权向

量 $w(\theta)$ 与接收信号不相关，与角度 θ 和阵列流型相关，是恒定不变的。θ 角度 k 基元的权重 $w_k(\theta)$ 是复数，其实部为 cos(Φ)，虚部为 sin(Φ)，其中 Φ 计算公式为

$$\phi = -\frac{2\pi f \Delta d}{c}, \tag{6.8}$$

$$\Delta d = \frac{Fs}{2(Fs\_bat - 1)} \times \Delta l_k(\theta), \tag{6.9}$$

其中 c 是声速，f 是频率，Fs 是采样率，Fs_bat 为一次处理的采样点数，$\Delta l_k(\theta)$ 为 k 基元在 θ 角度的阵列流型（见 6.3.2 节）。

此时 θ 角度 Pt 频点的（窄带）常规波束形成的输出功率为

$$\boldsymbol{P}_{\mathrm{CBF}}(\theta,\mathrm{Pt}) = \mathrm{E}[\left|\boldsymbol{Y}(\theta,\mathrm{Pt})\right|^2] = \boldsymbol{w}(\theta)^{\mathrm{H}}\mathrm{E}[\boldsymbol{X}(\mathrm{Pt})\boldsymbol{X}^{\mathrm{H}}(\mathrm{Pt})]\boldsymbol{w}(\theta) = \boldsymbol{w}(\theta)^{\mathrm{H}}\boldsymbol{R}_{\mathrm{Pt}}\boldsymbol{w}(\theta)。\tag{6.10}$$

其中 R_{Pt} 为基阵接收的声呐信号的协方差矩阵。对感兴趣频带（感兴趣频点的合集）的常规波束形成功率进行整合，得到宽带 CBF 功率谱为

$$\mathrm{P}_{\mathrm{CBF}}(\theta) = \sum_{\mathrm{Pt}} \boldsymbol{P}_{\mathrm{CBF}}(\theta,\mathrm{Pt})。\tag{6.11}$$

频域常规波束形成是从时域常规波束形成推导而来的，但时域常规波束形成处理无论如何提高采样率都会存在误差，而频域常规波束形成在满足一定条件的前提下可以避免计算误差，故频域常规波束形成优于时域常规波束形成，本文基于频域常规波束形成开展并行和优化研究。

宽带频域常规波束形成首先对各基元接收的声压信号进行 DFT 变换，得到各基元信号在不同频点处的幅度和相位信息，接着计算各角度各频点的 CBF 输出功率，统计整合频带能量得到宽带 CBF 功率谱。其中远场 CBF 计算时，需要对选定频点中除 CBF 感兴趣频点外的频点进行 Lofar 计算，Lofar 计算与 CBF 计算类似。Lofar 计算的目的是计算水声信号在低频频段不同频点的能量分布。

6.4.2 GPU 并行方案设计

6.4.2.1 DFT 并行方案

傅里叶变换提供一种变换到频率域的手段，是经典的数学变换之一。目前已有很多数学函数库实现了该变换，常见的有 FFTW 库、MKL 库和 CUFFT 库等。利用这些数学函数库可以简单方便地实现 DFT 变换，避免复杂的代码书写和调试工作。其中 CUFFT 库函数可以在 GPU 上实现并行 DFT 变换。

一般情况下，DFT 变换时间主要与基元数量和信号采样率相关。当基元数量较少、采样率较低时，DFT 运算量较少，GPU 并行计算可能无法获得理想的性能收益，此时需要通过训练得到相应的阈值，再根据阈值和声呐信号的参数来判断适合的数学函数库。

6.4.2.2 并行 CBF/Lofar 计算

CBF 和 Lofar 计算中，包含以下 3 个层次的并行计算：大量感兴趣频点的 CBF 能量可以同时计算（频点级并行）；单个感兴趣频点中，不同搜索角度的 CBF 能量可以并发计算（角度级并行）；在单个频点单个方向的 CBF 能量计算中，不同基元进行加权乘积时，亦可并行计算（基元级并行）。基于 GPU 协处理器，可以将这 3 个并行层次同时映射到一个 Grid：最内层的基元级并行映射到 block 内的 threads，搜索角度级并行映射到 blocks 并行中的 blockIdx.x，感兴趣频点级并行映射到 blocks 并行中的 blockIdx.y，三个并行层次的三维映射关系如图 6.4 所示。利用这种映射方式，可以将不同方向不同频点的 CBF 能量计算合理地映射到 GPU 上进行并行运算。

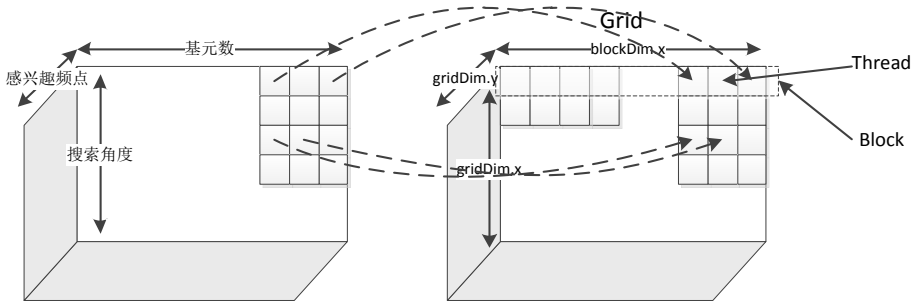


图 6.4 CBF/Lofer 计算的 GPU 映射方案

图 6.4 映射方案的好处之一是在 CBF 计算过程中，感兴趣频点和搜索角度可能会随具体情况改变，而 GPU 的 grid 网格中可分配的 block 维度非常大，能够满足感兴趣频点和搜索角度变化导致的不同资源需求。若基元数量增加超过 1024，超出了 block 中的 thread 数量限制，此时需要引入循环处理的思想，即所有 thread 通过固定跳步来循环处理所有基元的计算任务。

该映射方案的另一个好处是契合 block 内快速归约的计算需求。算法中每个感兴趣频点和搜索角度都需要累加不同基元计算的 CBF 能量，这是一个典型的归约运算。归约过程伴随同步，GPU 中只有 block 内部才能同步，而 block 间无法同步，上述并行映射方案恰能契合这一点，并能充分发挥 GPU 快速归约的能力。

6.4.2.3 并行频带能量整合

在计算得到不同方向不同频点上的 CBF 能量后，还需要对其进行频带能量整合。频带能量整合是将相同角度的不同感兴趣频点的 CBF 能量进行整合，所有的角度能量构成频带能量。在频带能量整合过程中，存在两个并行层次，分别是搜索角度并行和频点能量统计并行。根据 GPU 架构特性，这两层并行可分别映射到 block 和 thread，如图 6.5 所示。其中单 block 内所有 thread 整合单一角度的频点能量，过程中需要 thread 间数据交互，能较好地契合 block 间无法同步和交互，而 block 内可以同步和交互的 GPU 基本特征；同时这种 GPU 映射方案可以采取共享

存储来优化能量整合过程中的数据访问。

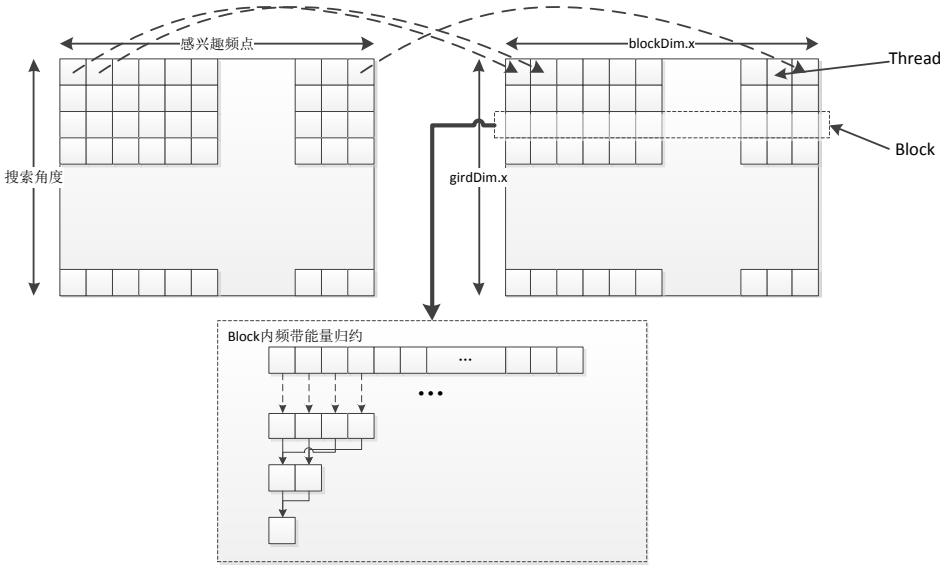


图 6.5 频带能量整合的 GPU 映射方案

6.4.3 面向众核 GPU 的性能优化策略与效果

6.4.3.1 DFT 合并优化

FFTW 库提供了种类丰富的 DFT 变换函数，将接收到的声压信号转换为频域信号。对各个基元接收的声压信号分别进行 DFT 变换，共 B 个基元信号需进行 1 维 DFT 变换，实现时选用 fftw_plan_dft_r2c_1d()函数循环 B 次（记为 fftw_1d）。此外，FFTW 库还提供一些合并的 DFT 变换函数，一个函数可以实现多次 DFT 变换，比如 fftw_plan_many_dft_r2c()函数可以一次启动完成多次 DFT 变换。选用该函数可以更加充分发挥 CPU 运算性能，进而优化 DFT 变换（记为 fftw_many）。同理可实现 MKL 库的 DFT 变换的优化（分别记为 mkl_1d 和 mkl_many）。

在 CUFFT 库中，cufftPlan1d()函数和 cufftPlanMany()函数均支持同时多次 DFT 变换，且经测试两者性能相当（记为 cufft_many）。为了与 CPU 的两种 DFT 变换方案对比，将 cufftPlan1d()函数同时多次 DFT 变换拆分为 B 次 1 维 DFT 变换（记为 cufft_1d）。

下面分别实现了 FFTW 库、MKL 库和 CUFFT 库三个版本优化前后的两种 DFT 变换，处理不同基阵规模的声呐信号（采样率为 8000），统计 DFT 变换时间，见表 6.1。表中数据显示：（1）分别对比 fftw_1d 和 fftw_many、mkl_1d 和 mkl_many、cufft_1d 和 cufft_many，经过 DFT 合并优化后性能得到了不同程度的提升；（2）对比 fftw_1d 和 mkl_1d，发现 FFTW 库的 1 维 DFT 变换性能比 MKL 库性能更好；（3）对比 fftw_many 和 mkl_many，MKL 的合并 DFT 变换性能更佳；（4）CUFFT 库的 DFT 变换函数的性能明显更优于 FFTW 库和 MKL 库。

表 6.1 DFT 合并优化效果/ms

B	128	256	512	1024	2048
fftw_1d	56.78	113.95	227.74	454.27	909.34
fftw_many	50.24	101.03	201.56	402.64	805.45
mkl_1d	74.89	149.53	299.87	598.32	1204.30
mkl_many	17.95	38.77	71.60	153.16	305.97
cufft_1d	5.16	9.50	17.71	36.39	71.31
cufft_many	2.27	3.62	6.36	12.58	24.14

测试时还发现，首次 DFT 变换耗时比平均 DFT 变换耗时长。表 6.2 统计了首次 DFT 变换额外耗时。表中数据显示，FFTW 库和 MKL 库的首次 DFT 变换的额外耗时随 DFT 变换次数的增多而增加，而 CUFFT 库的首次 DFT 变换（已排除 GPU 的首次 kernel 启动开销）额外耗时是个定值，约为 233ms。

表 6.2 首次 DFT 额外耗时/ms

B	128	256	512	1024	2048
fftw_1d	7.08	9.61	15.95	27.58	49.92
fftw_many	7.21	10.37	16.27	28.34	51.61
mkl_1d	7.71	10.41	14.45	34.21	57.24
mkl_many	6.44	11.26	13.86	26.10	46.50
cufft_1d	234.05	233.28	231.01	235.09	233.61
cufft_many	231.52	231.75	234.21	233.17	235.25

由于 FFT 函数库的首次启动开销的存在，若 DFT 库函数仅被使用 1 次或数次，需要综合考虑不同 FFT 库的时间消耗。比如若仅需要一次 DFT 变换，基阵规模小于 1024 时，mkl_many 版本执行的速度比 cufft_many 版本快，此时选用 MKL 库的 DFT 变换函数能获得更好的性能；而当基阵规模达到 2048 后，cufft_many 版本的性能最好，此时选择 CUFFT 库的相关 DFT 函数。

6.4.3.2 CBF 计算的性能优化

根据 CBF 计算的特点及映射到 GPU 的策略，结合第 3 章访存优化框架，分析 kernel 函数中各数据的访存模式，选用 S_5、R_5 和 G_3 等访存优化策略来优化 CBF 并行计算过程。

（1） 利用共享存储优化归约运算中的中间数据访存（S_5）。CBF 计算过程中，不同基元的能量累加需要进行归约运算。GPU 中的归约运算已有大量优化研究，这里不再赘述，在初写 kernel 函数时已充分考虑了该优化。（记为 V0）

（2）以传参的方式优化全局存储中参数数组的访问（R_5）。CBF 运算涉及一个参数数组，存储基元数、采样率、频率、CBF 感兴趣频点的最大最小值、Lofar 感兴趣频点的最大最小值、搜索角度、近场搜索距离数量和频点数等信息。C_0 中，该参数数组存储于全局存储，访存开销较大；通过传参的方式，将 kernel 函数需要的参数数据以传参的形式直接传值给 kernel 函数，此时参数数据位于寄存

器，避免了开销较大的全局存储参数数组访问。（记为 V1）

（3）重构 DFT 结果矩阵，保证 warp 线程连续访存，令全局存储访存合并（G_3）。分析 kernel 函数的数据访存模式时，发现相邻线程对 DFT 结果矩阵的访问是不连续的，这将导致全局存储访问不可合并。根据 DFT 结果矩阵的访问规律，对 DFT 结果的数据结构进行重新组织，令 warp 线程访问的数据连续存储，从而达到合并访问全局存储的目的。（记为 V2）

（4）重构声程差矩阵，使得相邻线程访问连续的声程差数据（G_3）。声程差矩阵是一个 3 维矩阵，包括距离维度、基元维度和角度维度，其中角度维度的存储是连续的。根据图 6.4 的映射策略，warp 相邻线程按基元维度访问声程差矩阵，而声程差矩阵是按角度维度存储的，这就导致了 warp 线程访问不连续的问题。通过重构声程差矩阵，使得 warp 线程访问的声程差矩阵连续，此时声程差矩阵的访问是可合并的。（记为 V3）

分别实现上述优化，处理 1024 个基元的声呐信号波束形成，测试 CBF 计算时间，见表 6.3。表中数据显示，从 V0 到 V3，CBF 计算时间逐步下降，说明上述几种优化策略均取得了理想的性能提升，优化后 CBF 计算性能提升了近 3 倍。

表 6.3 CBF 优化效果

optimization	V0	V1	V2	V3
CBF time/ms	186.05	170.72	120.45	61.88

6.4.3.3 可扩展性优化

在 DFT 变换结果标准化、CBF/Lofar 计算、频带能量整合等过程中，若水听器阵列的基元数量持续增长，原始的 GPU 映射策略会出现漏洞，导致无法运算大规模基阵的波束形成。比如在 DFT 变换结果标准化时，当水听器阵列的基元数量过万时，将导致 x 维度的 block 数量超出设备上限。又比如在 CBF/Lofar 计算中，若基元数量大于 1024 时，映射到 thread 的基元层并行数量超过 CUDA 的最大 thread 数量限制，导致 kernel 函数无法执行。

为了令 DFT-CBF 的代码具有良好的可扩展性，使其能够处理万基元基阵的声呐信号波束形成，进行了以下两个方面的可扩展性优化：（1）对于因 thread 数量和 block 数量限制，导致无法处理大规模基阵声呐信号波束形成的情况，引入循环处理的思想，在 kernel 函数中循环处理数据。（2）当 thread 数量增长、数据规模扩大导致共享存储容量不足或性能下降的情况，利用“少量多次复用共享存储优化（S_7）”的方式进行优化。经优化后，理论上本文的并行波束形成代码可以处理任意基阵规模的声呐信号数据。

6.4.4 基于 GPU 的频域常规波束形成算法

在 DFT-CBF 串行算法基础上，利用 6.4.2 节的 DFT 变换、CBF 能量计算、Lofar 能量计算、频带能量整合等 GPU 并行映射方案替代串行实现，采取 6.4.3 节描述的优化策略，提出基于众核 GPU 的宽带 DFT-CBF 算法，见图 6.6。算法中有两个过程可以并发执行：输入声呐信号并传输到 GPU（H2D），GPU 进行 DFT 变换；配置参数，计算阵列流型并传输声程差矩阵到 GPU（H2D）。在计算 CBF 能量场时，CPU 控制能量场计算流程，GPU 完成 CBF/Lofar 计算、频带能量整合等。GPU 完成 CBF 能量场的计算后，将 CBF 能量场传回 CPU 端（D2H）并输出。

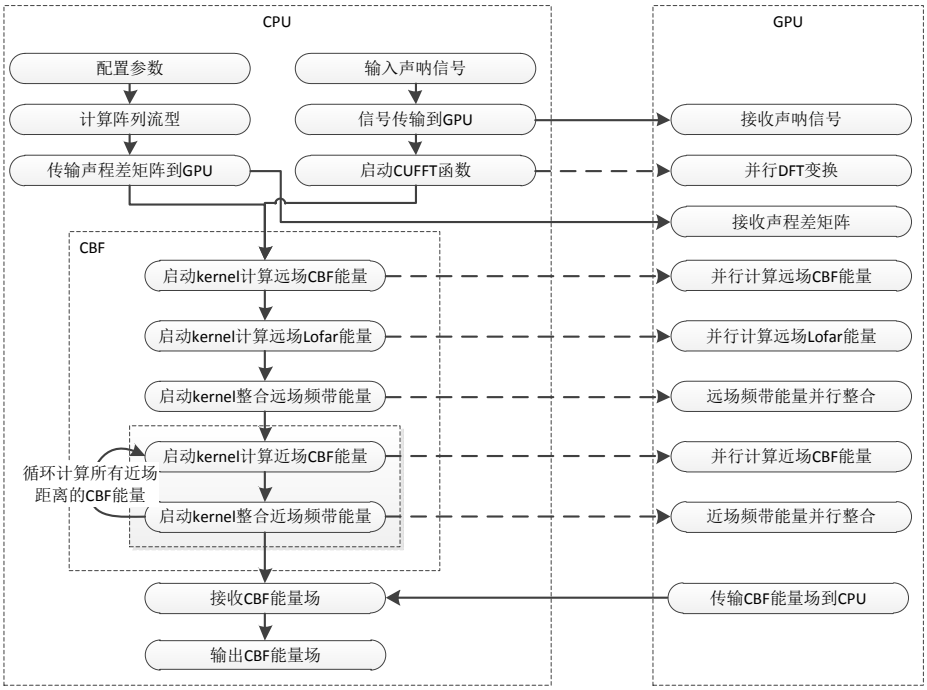


图 6.6 基于 GPU 的宽带 DFT-CBF 算法

6.4.5 实验结果分析

6.4.5.1 CBF 能量场及到达角度分析

在 K20c GPU 平台上，利用基于 GPU 的宽带 DFT-CBF 算法对一组真实的水声信号（基元数为 184，采样率为 20000）进行波束形成运算，统计不同距离不同角度的 CBF 能量场数据（见图 6.7）和远场不同角度的 CBF 频带能量信息（见图 6.8）。根据图中数据，目标位于远场，20°角方向。

6.4.5.2 参数配置与加速比变化分析

声呐信号波束形成处理时，一些参数配置将直接影响波束形成的执行时间，比如基阵规模、感兴趣频带宽度（频点数）、搜索角度和声波信号采样率等。本节主要分析感兴趣频点数、搜索角度和声波信号采样率改变时，串行/并行的声呐

信号常规波束形成的执行时间和加速比的变化。

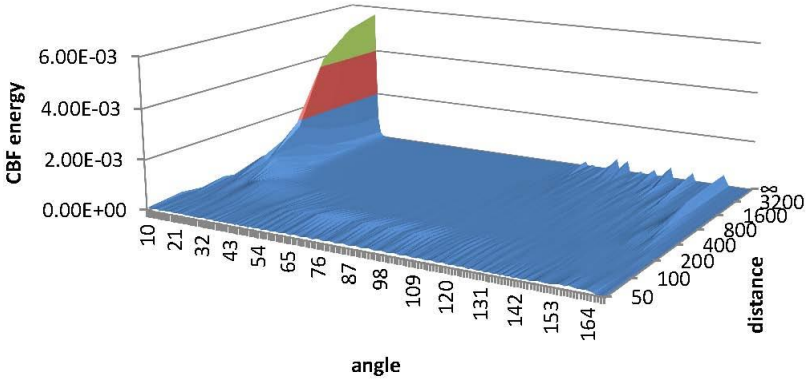


图 6.7 CBF 能量场

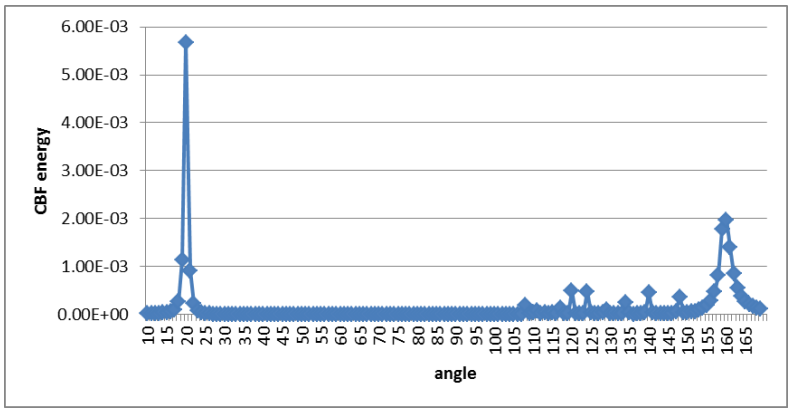


图 6.8 远场 CBF 频带能量

设定声呐基阵的基元数为 512，搜索角度为 160°（从 10°搜索到 170°），声波信号采样率为 8000Hz，改变感兴趣频点数量（Lofar 频带设定为较 CBF 频带多 100 个频点），测试频域常规波束形成的执行时间，并计算加速比，见表 6.4。表中数据显示，串行和并行的常规波束形成耗时均随着频点数增加而线性增长，其中串行 CBF 时间增加更快，表现为加速比随频点数增加而逐步增长。

表 6.4 频点数变化导致的性能改变

频点数	serial/ms	cuda/ms	speedup
100	959.73	36.03	26.6
200	1756.74	58.81	29.9
300	2593.27	81.64	31.8
400	3372.75	104.41	32.3
500	4203.13	127.23	33.0
600	5009.35	150.00	33.4
700	5886.76	172.82	34.1
800	6786.82	195.76	34.7
900	7620.06	218.60	34.9
1000	8533.20	241.52	35.3

设定声呐基阵的基元数为 512，声波信号采样率为 8000Hz，CBF 感兴趣频带

为 327~492，Lofar 感兴趣频带为 34~492，逐步调整搜索角度，测试 DFT-CBF 执行时间，计算加速比，见表 6.5。表中数据显示，DFT-CBF 耗时与搜索角度正相关，加速比随搜索角度增加先逐步增加后稳定。分析加速比先增加后平稳的原因，CBF 计算（图 6.4）和频带能量整合（图 6.5）两个步骤中均包含角度维度，当搜索角度较少时，K20c GPU 的计算资源未能完全使用，而随着搜索角度增多图 6.4 和图 6.5 中的角度维度的并行度增加，能够更加充分地发挥 GPU 计算性能，直到完全使用计算资源，加速比达到稳定状态。

表 6.5 搜索角度变化导致的性能改变

角度	Serial/ms	Cuda/ms	speedup
45	551.00	24.38	22.6
90	979.92	35.25	27.8
135	1471.73	49.08	30.0
180	2003.63	63.14	31.7
225	2530.97	76.78	33.0
270	2958.87	90.63	32.6
315	3500.29	103.95	33.7
360	3870.21	118.36	32.7

设定基元数为 512，搜索角度为 160°，CBF 感兴趣频带为 327~492，Lofar 感兴趣频带为 34~492，改变声波信号的采样率，测得 CBF 耗时及加速比见表 6.6。表中数据显示，当采样率增加时，DFT-CBF 执行时间有少量增加，加速比却逐步下降。且加速比下降呈阶梯状，采样率 2000~4000 的加速比为 30 左右,采样率 6000~8000 的加速比为 29 左右，采样率 10000~16000 的加速比约为 26, 采样率 18000~20000 的加速比约为 22。分析 CBF 执行时间少量增加的原因，采样率直接影响 DFT 执行时间，而 DFT 占整体程序的比重有限，所以采样率的变化不会导致执行时间的大幅改变。分析加速比阶梯下降的原因，随着采样率增加，DFT 执行时间增加，而 DFT 的并行加速比远低于整体加速比，故 DFT 比重增加后导致整体加速比的下降。阶梯下降的原因在于，真正影响 DFT 执行时间的参数是采样点数，而采样点数是比采样率大的最小的 2 的幂次，当不同采样率的采样点数相同时，DFT 执行时间一致，故不影响整体加速比，此时为同一个阶梯。表 6.7 展示了采样率、采样点数及其导致的 DFT 性能变化，表中数据验证了上述分析。

6.4.5.3 加速比与实时性分析

利用模拟生成的一系列不同基阵规模的声呐信号数据作为输入，将 1 分钟采样得到的声呐信号数据输入串行和 GPU 并行的频域常规波束形成算法，设定声波信号采样率为 8000Hz，搜索角度为 160°，CBF 感兴趣频带为 327~492，Lofar 感兴趣频带为 34~492，统计连续处理 1 分钟采样信号的执行时间，见表 6.8。表中数据显示，基于 GPU 的并行 DFT-CBF 算法的执行时间比串行 DFT-CBF 算法的执行

时间短很多，加速 24~125 倍不等，且并行算法加速比随着水听器阵列规模增大而增加。

表 6.6 采样率变化导致的 CBF 性能改变

采样率	serial/ms	cuda/ms	speedup
2000	1591.747	52.190	30.5
4000	1632.054	53.520	30.5
6000	1644.052	56.586	29.1
8000	1641.685	56.646	29.0
10000	1639.181	63.022	26.0
12000	1640.944	62.969	26.1
14000	1658.441	63.008	26.3
16000	1676.230	63.020	26.6
18000	1697.948	75.067	22.6
20000	1672.627	75.041	22.3

表 6.7 采样率变化导致的 DFT 性能改变

采样率	采样点数	serial/ms	cuda/ms	speedup
2000	2048	2.764	2.289	1.2
4000	4096	7.055	3.624	1.9
6000	8192	15.639	6.581	2.4
8000	8192	14.735	6.564	2.2
10000	16384	33.238	12.895	2.6
12000	16384	31.255	12.859	2.4
14000	16384	33.242	12.897	2.6
16000	16384	31.377	12.912	2.4
18000	32768	76.791	24.987	3.1
20000	32768	71.368	24.945	2.9

从频域常规波束形成实时性的角度分析，串行的 DFT-CBF 仅在处理少于 256 个基元的声呐信号数据时才能保证实时性，若基阵规模达到 512 后则无法实时处理。在 K20c GPU 上的并行 DFT-CBF 算法处理 16384 个基元 1 分钟采样的声呐信号耗时仅需 36.14 s；处理 32768 个基元 1 分钟采样的声呐信号耗时 76.97 s；说明基于 GPU 的并行 DFT-CBF 算法具有极佳的实时性，能够满足万基元声呐信号的实时处理需求。若要实时处理更大规模基阵声呐信号的频域常规波束形成，主要有两条思路：选用性能更高的 GPU 型号，利用多节点多 GPU 进行算法扩展。

6.5 基于 GPU 的最小方差无畸变响应自适应波束形成

6.5.1 最小方差无畸变响应波束形成及热点分析

6.5.1.1 MVDR 自适应波束形成

MVDR 方法是一种主瓣约束自适应方法，通过调节阵列加权向量的方式，允

许主波束方向的信号正常通过，而抑制来着其他方向的信号。MVDR 在保证主波束方向信号无畸变的前提下，使得整体输出功率最小。在常规波束形成中，阵列加权向量 w 是固定的，而在 MVDR 算法中，阵列加权向量 w 是根据实际接收数据、环境噪声和实际空间目标强弱分布自适应求得的。

表 6.8 1 分钟水声波束形成耗时与加速比

B	Serial/s	Cuda/s	speedup
128	23.96	1.00	24.0
256	49.44	1.58	31.3
512	101.71	3.40	30.0
1024	214.76	4.35	49.4
2048	485.24	6.32	76.8
4096	993.03	10.50	94.6
8192	2071.60	18.73	110.6
16384	4241.82	36.14	117.4
32768	9646.53	76.97	125.3

声呐在期望方向上的波束输出包含该方向的目标信号和旁瓣接收的其他方向的干扰信号。在 MVDR 算法中，为了使来自某个期望方向上的信号 θ_s 能完全接收，抑制干扰信号，构造一个有约束的最优化命题。令阵列在目标方向的输出（式 6.7）是一个常系数，例如 1；同时使得基阵的整体输出功率最小，即式 6.10 最小。数学表达式为

$$\begin{cases} \min_{\omega} w^H R w \\ w^H a_v(\theta_s) = 1 \end{cases}。 \tag{6.12}$$

利用用拉格朗日常数算法进行运算。令得目标函数为

$$L(w) = \frac{1}{2} w^H R w - \lambda\left[w^H a_v(\theta_s) - 1 \right] \ , \tag{6.13}$$

其中 λ 为常数。目标函数对 w 求导，令导函数结果为零，可以得到 MVDR 算法最优权矢量 w_{opt}，为

$$w_{opt} = \frac{R^{-1} a_v(\theta_s)}{a_v{}^H(\theta_s) R^{-1} a_v(\theta_s)}。 \tag{6.14}$$

此时基阵的（窄带）MVDR 输出功率为

$$P_{MVDR} = w_{opt}{}^H R w_{opt} = \frac{a_v{}^H(\theta_s)(R^{-1})^H}{a_v{}^H(\theta_s)(R^{-1})^H a_v(\theta_s)} R \frac{R^{-1} a_v(\theta_s)}{a_v{}^H(\theta_s) R^{-1} a_v(\theta_s)} = \frac{1}{a_v{}^H(\theta_s) R^{-1} a_v(\theta_s)}。 \tag{6.15}$$

对于每个感兴趣频点，扫描所有角度来计算能量方位谱，故窄带 MVDR 方位谱计算公式为：

$$P_{MVDR}(\theta, Pt) = \frac{1}{a_v{}^H(\theta) R^{-1}(Pt) a_v(\theta)}。 \tag{6.16}$$

由于求逆运算比较麻烦，利用特征值和特征向量进行替换。对 R 进行 SVD 分解 $R = VDV^H$，其中 V 为酉矩阵，D 为对角矩阵，故 $R^{-1} = VD^{-1}V^H$，此时 $a_v^H(\theta)R^{-1}a_v(\theta) = a_v^H(\theta)V\sqrt{D^{-1}}$，故窄带 MVDR 方位谱计算公式为：

$$P_{MVDR}(\theta, Pt) = \frac{1}{a_v^H(\theta)V(Pt)\sqrt{D^{-1}(Pt)}}。 \quad (6.17)$$

将所有感兴趣频点（频带）的窄带 MVDR 方位谱进行整合，可以得到最终的宽带 MVDR 方位谱：

$$\mathrm{P_{MVDR}}(\theta) = \sum_{\mathrm{Pt}} \mathrm{P_{MVDR}}(\theta, \mathrm{Pt})。 \quad (6.18)$$

宽带 MVDR 波束形成的算法流程见图 6.9，首先利用离散傅里叶变换将基阵接收到的声呐信号转换为频域信号，根据频域信号计算自相关矩阵，对自相关矩阵进行对角加载以确保其可以解得特征值和特征向量，通过特征分解求得特征值和特征向量；求取方向向量，计算角度能量，算得每个感兴趣频点的所有角度方位谱，统计各角度的能量方位谱，最终确定信号到达角度方向。

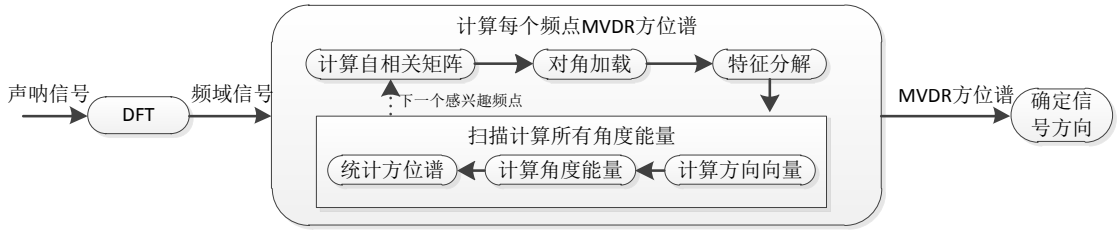


图 6.9 MVDR 波束形成算法流程图

6.5.1.2 MVDR 热点分析

实现宽频 MVDR 自适应波束形成的串行算法，分别输入基阵规模为 64、128、256、512、1024 和 2048 的声呐信号数据，进行 MVDR 波束形成运算，统计不同规模基阵信号的分步处理时间，见表 6.9。表中数据显示，随着基阵规模的增加，FFT 时间和 MVDR 时间均明显增加，两者都是并行和优化研究需要考虑的热点。其中 6.4.2.1 节和 6.4.3.1 节已研究了 FFT 变换的并行和优化，不再赘述。故本节重点研究 MVDR 的并行和性能优化。

表 6.9 MVDR 波束形成耗时分析/ms

基阵规模	64	128	256	512	1024	2048
FFT	8.344	12.143	19.906	41.837	115.476	239.649
MVDR	166.138	609.552	2362.765	9525.697	37038.274	174741.975

针对 MVDR 过程，测试和分析各个步骤的耗时，见表 6.10，表中记录了的是单个感兴趣频点的 MVDR 方位谱计算的分步时间。其中，特征分解和计算方位谱过程耗时最长且增长最快，是并行和优化研究的重点。此外当基阵规模扩大到一定程度后，计算自相关矩阵的开销逐步增加，亦可纳入考虑范围。

表 6.10 MVDR 精细耗时分析/ms

MVDR	计算自相关矩阵	对角加载	特征分解	计算方位谱
64	0.004	0.000	0.373	1.340
128	0.014	0.000	1.466	4.802
256	0.162	0.001	6.170	18.174
512	0.786	0.003	27.730	71.325
1024	1.754	0.005	104.185	280.906
2048	8.307	0.018	676.588	1223.901

6.5.2 MVDR 并行方案设计

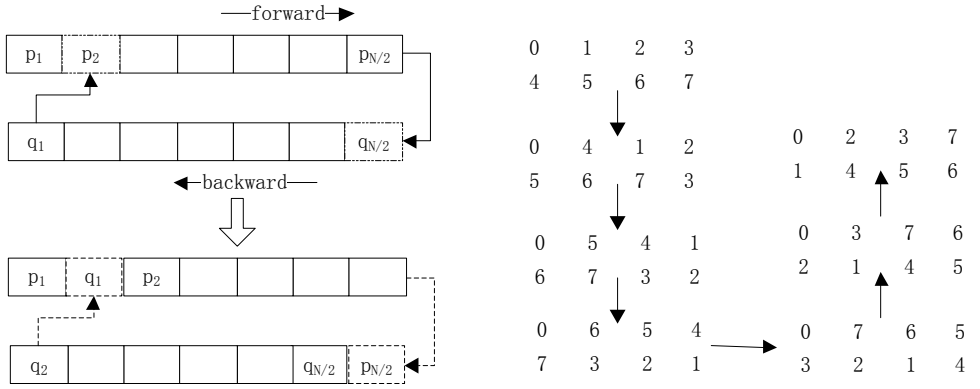
6.5.2.1 并行的厄尔米特矩阵特征分解（雅克比迭代法）

在 MVDR 算法中，频域声呐信号的自相关矩阵是厄尔米特矩阵（复数），采用雅克比迭代算法进行特征分解，求解特征值和特征向量。在双边雅克比迭代算法中，关键计算量为左右两次 Givens 变换，即矩阵的行更新和列更新。由于存在数据依赖，行更新和列更新无法同时进行；但当矩阵规模较大时，行更新和列更新的计算量也是比较可观的，可以进行并行计算。

此外，对于雅克比迭代，每次 Givens 变换的行/列更新仅更新两行和两列，通过循环的方式更新矩阵的所有行列。串行雅克比迭代算法一般采用轮询的方式实现，对于对角线上的交合点而言，每个点与其他所有点进行组合，分别进行 Givens 变换。但这种轮询的方式由于存在依赖，不同的 Givens 变换（行列更新）无法同时进行。利用预先的坐标调度方式，可将互不相关的两行和两列更新提前取出，此时互不相关的行列更新可以并行运算。图 6.10 描述了这种预先的坐标调度方式，图 6.10(a)为坐标调度策略，图 6.10(b)展示了一个简单的小实例。坐标调度时，先构造一个两两对应的坐标序列，基于该坐标序列，保持首个坐标不变，依次移动其他坐标，直到首个坐标与其他所有坐标都组成两两对应关系。通过一轮坐标调度后，记录每次调度的坐标序列，此时每一组坐标序列中对应的坐标都是不相关的，可以同时进行 Givens 变换。

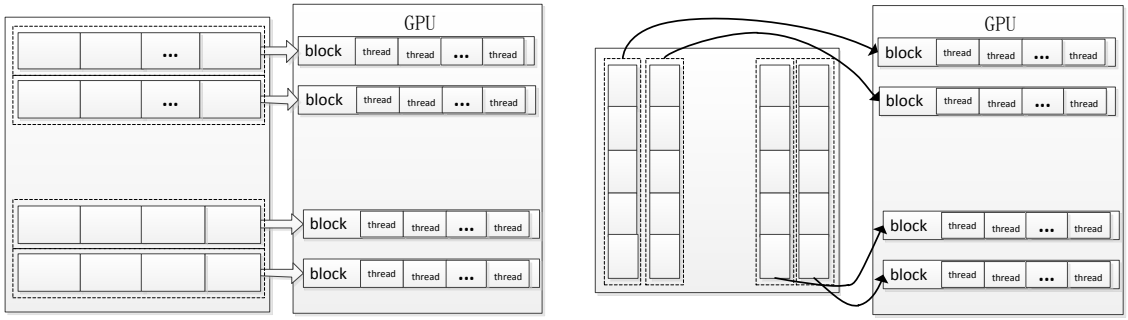
通过预先的坐标调度方式，构造了两个层次的雅克比迭代并行计算方案，分别是 Givens 变换（行/列更新）内的并行和不同坐标对应关系的 Givens 变换间的并行。故可将较粗粒度的 Givens 变换间的并行层次映射到 GPU 的 block，而细粒度的 Givens 变换内的并行层次映射到 block 内的 thread。具体实现时，对于维度为 n×n 的厄尔米特矩阵，通过预先的坐标调度方案构造 n/2 个不同的坐标序列，其中每个坐标序列包含 n/2 个 Givens 左变换（行更新）和 Givens 右变换（列更新），且相互间不相关。故将各个坐标序列的 Givens 左变换（行更新）合集和 Givens 右变换（列更新）合集映射到 GPU，其中一组坐标序列的行更新合集映射到 1 个 kernel

函数，列更新合集映射到另一个 kernel 函数，每个行更新映射到 1 个 block，每个列更新映射到 1 个 block，而行/列内的单个元素更新则映射到 thread，见图 6.11。



(a) (b)

图 6.10 雅克比迭代的坐标调度方案



(a)Givens 左变换（行更新）映射 (b) Givens 右变换（列更新）映射

图 6.11 雅克比迭代的 GPU 映射方案

6.5.2.2 方位谱并行计算

MVDR 方位谱计算过程包括三个步骤：计算方向向量、计算各角度各基元的能量、统计能量方位谱。下面分别展开描述这三个步骤的 GPU 并行映射策略。

方向向量的计算公式为

$$a_v^H(\theta,k)=\left\{\cos\left(\frac{2\times\pi\times Fs\times d\times k\times\sin(\theta)}{c}\right),-\sin\left(\frac{2\times\pi\times Fs\times d\times k\times\sin(\theta)}{c}\right)\right\},\quad(6.19)$$

其中 k 为基元索引号，Fs 为感兴趣频点，d 为基元间距，c 为声速。公式中，不同的角度 θ 和不同的基元索引 k 的方向向量的计算相互独立，可映射到 GPU 进行并行计算。在 GPU 映射时，将角度 θ 的计算映射到 block 级，即每个角度的方向向量计算任务映射到 1 个 block；同时将基元映射到 thread 级，即每个线程计算 1 个基元的方向向量。

各角度各基元的能量计算公式为

$$S(\theta,k)=\frac{\left(a_v^H(\theta,k)V(k)\right)^2}{D(k)},\quad(6.20)$$

其中 a_v^H 是方向向量矩阵，其维度是角度数和基元数，V 是特征向量矩阵，其维度是基元数和基元数。上述运算与矩阵乘法类似，GPU 并行映射时，将角度 θ 的能量计算映射到 block 层次，将基元 k 的能量计算映射到 thread 级。

统计能量方位谱时，需要统计各个角度上的所有基元接收的信号能量总和，不同角度的能量谱统计可以并行计算，一个角度的所有基元能量统计为归约运算。GPU 的 block 内 thread 间可以同步，适合用于基元间的能量归约运算，而 block 间无法同步，适合不同角度的能量统计。故 GPU 并行映射时，将角度层的能量统计映射到 block 级，将基元层的归约运算映射到 thread 级。

MVDR 方位谱计算的三个步骤均包含有角度层和基元层，GPU 映射时都将角度层映射到 block，将基元层映射到 thread。故实现 kernel 函数时，可以将三者合并为同一个 kernel，以避免中间结果的的全局存储读写，同时可以避免多个 kernel 函数的启动开销。

6.5.3 面向众核 GPU 的性能优化策略与效果

6.5.3.1 雅克比迭代的优化与效果

GPU 并行雅克比迭代包含大量的 kernel 函数迭代计算，每次迭代的时间较少，且需要三个 kernel 函数来分别计算 Givens 变换矩阵、Givens 左变换（行更新）和 Givens 列变换（列更新），故单个 kernel 函数的耗时非常小。此时 kernel 函数对 thread 配置就变得非常敏感。分别处理多组不同规模基阵的声呐信号，配置不同的 thread 数量，统计雅克比迭代计算时间，见表 6.11，其中横行表示 thread 配置，竖列为输入声呐信号的基元数量。表中数据显示，当 thread 配置为 128 时，并行雅克比迭代的执行时间最少，故配置雅克比迭代的 kernel 启动函数的线程数量为 128。

表 6.11 thread 配置对雅克比迭代的影响/ms

	64	128	256	512	1024
512	14.51	12.34	12.34	14.24	14.27
1024	36.81	31.23	32.56	41.41	64.38
2048	110.65	91.25	97.32	132.55	227.64

根据 6.5.2.1 节的并行雅克比迭代方案，结合 3.7 节 R_5 和 G_3 两种访存优化策略，设计了以下两种雅克比迭代优化方案。

（1）利用私有变量（寄存器）优化 Givens 变换矩阵（全局存储）的访问（R_5）。在 GPU 并行雅克比迭代中，Givens 变换矩阵存储于全局存储，被行变换和列变换访问，访问时为广播访问且每个 block 内所有 thread 的 Givens 变换矩阵均相同，在一次 Givens 左/右变换时，Givens 变换矩阵多次重复访问。基于上述特点，可以利用私有变量矩阵来存储 Givens 变换矩阵，由于该矩阵尺寸较小且运算简单，结合 3.6.1 节测评结果可知，该私有变量矩阵将被分配在寄存器中，从而达到极低的

访存延迟。

（2）特征向量矩阵转置，令其在雅克比迭代过程中按行访问（G_3）。在雅克比迭代过程中，除了输入的厄尔米特自相关矩阵需要进行行更新和列更新外，特征向量矩阵也需要进行列更新。结合 3.5.3 节测评的全局存储访存特性，行更新时厄尔米特自相关矩阵的 warp 线程访问是连续可合并的，而列更新时厄尔米特自相关矩阵和特征向量矩阵都按列访问，warp 级线程访存不连续且不可合并。对于厄尔米特自相关矩阵而言，必有一次行更新和一次列更新，不可避免一次不可合并的全局存储访问。而对于特征向量矩阵，通过矩阵转置，可以将列更新转换为行更新，使得其 warp 级访存连续，特征向量矩阵可合并访存，进而提升雅克比迭代性能。

依次实现上述两种性能优化方案，分别输入声呐基阵规模为 1024 和 2048 的两组声呐信号，在 K20c GPU 平台上，统计雅克比迭代的时间见表 6.12。其中 V0 表示未采取优化版本，V1~V2 分别依次采取了优化策略 1~2。表中数据显示，从 V0 到 V2 两组信号的雅克比迭代时间均逐步下降，说明上述两种优化策略均能有效加速 GPU 上雅克比并行迭代过程。

表 6.12 雅克比迭代优化效果/ms

	V0	V1	V2
1024	31.30	29.93	27.40
2048	91.72	85.48	77.83

6.5.3.2 方位谱计算优化与效果

针对 6.5.2.2 节方位谱计算的并行方案，结合 3.7 节总结的 GPU 访存优化策略，采取了 G_3、S_5 和 S5+S6 三种优化策略。

（1）通过矩阵转置来避免 warp 级访存不连续，确保全局存储的合并访问（G_3）。访问特征向量矩阵时，相邻 thread 按列访问数据，warp 级访存不连续，导致了特征向量矩阵访问无法合并。通过矩阵转置的方式，可以将按列访问的矩阵转变为按行访问，确保 warp 内 thread 访问的数据是连续的，从而保证全局存储的合并访存。

（2）利用共享存储优化归约过程的中间变量访问（S_5）。不同基元的 MVDR 能量归约运算时，待归约的数组（中间变量）存在重复访问现象，可以利用共享存储优化归约过程中的中间变量数组。

（3）利用共享存储优化方向向量的存储与访问（S_5+S_6）。计算能量方位谱时，先计算方向向量，然后利用方向向量、特征向量和特征值计算各角度个基元的能量，对于方向向量而言，其访问为先写后读，特别是读取访问涉及多次重复读取（即复用），可以利用共享存储优化方向向量的访存。

依次实现上述优化策略，在 K20c GPU 平台上，执行一组 1024 个基元的声呐

信号，统计单个频点的方位谱计算时间，见表 6.13。其中 V0 表示未优化版本，V1~V3 分别依次采取上述优化策略 1~3。表中数据显示，随着优化策略的依次采纳，执行时间逐步减少，说明上述优化策略均取得了有效的正优化效果。经过上述 3 个优化策略，优化后的方位谱计算相对未优化的并行版本加速了 9.7 倍。

表 6.13 方位谱计算优化效果/ms

optimization	V0	V1	V2	V3
run time	51.92	7.33	7.07	5.35

当输入信号的基阵规模扩大后，特别是超过 block 内的 thread 限制（1024）后，额定 thread 数量无法处理所有的基阵，需要引入循环处理机制进行扩展。此外，当基阵规模数量超过 1024 后，由于共享存储的过多使用，将导致两个后果：（1）若规模持续增大，导致共享存储空间无法满足方向向量的存储需求；（2）单个 block 使用过多共享存储，导致一个 SM 中活跃的 warp 数量骤降，使得 GPU 的 occupancy 下降，导致整体性能损失。针对这些问题，接下来在上述 3 种优化策略的基础上，面向大规模基阵声呐信号的方位谱计算进行进一步优化，采取了 G_3 和 S_7 等优化策略。

（4）减少共享存储使用，释放 SM 中活跃 warp 数量。由于过多的共享存储使用导致 SM 中活跃的 warp 数量下降，导致 kernel 函数性能下降。解决思路就是通过减少共享存储使用，来释放 SM 中活跃的 warp 数量。具体解决方案是将方向向量从共享存储中释放，回归到采用全局存储进行访存。

（5）重组织全局存储中的方向向量，增加访存局部性，使其能够合并访存（G_3）。根据 6.5.2.2 节的并行映射方案，方向向量存储时 warp 级线程访问是连续的，故可合并访存；而访问方向向量时 block 内是广播访问，block 间按列访问，不可合并访存。根据 3.5.3 节的测评结果，全局存储支持广播访问并且性能较优，而 block 间的按列访问可能导致访存不连续。由于方向向量仅写入 1 次而读取很多次（基阵规模），故读取开销比写入开销更敏感。从这方面考虑，将方向向量矩阵转置，令不同 block 访问的数据连续存储，增加访存局部性，使得其有合并访存的可能性。

（6）少量多次循环使用共享存储优化（S_7）。根据大规模基阵声呐信号处理的特点，归约运算中中间变量的共享存储优化中，可采取“少量多次循环使用共享存储优化（S_7）”的策略。将原来根据基阵规模设定的中间变量长度固定下来，通过循环处理的方式完成所有计算和共享存储访问。利用这种方法，可以减少 block 内的共享存储使用，增加 SM 中活跃的 warp 数量，进而提高 kernel 函数运算性能。

分别依次实现上述几种优化策略，输入规模为 2048 的水听器阵列声呐信号，在 K20c GPU 平台上执行并统计方位谱计算时间，见表 6.14。表中 V3 表示采取了

1~3 优化策略，V4~V6 在 V3 的基础上依次采纳 4~6 三种优化策略。表中数据显示，从 V3 到 V6 方位谱的计算时间依次下降，说明针对大规模基阵方位谱计算提出的这 3 种优化策略均能获得有效的性能提升。

表 6.14 大规模基阵的方位谱计算优化/ms

optimization	V3	V4	V5	V6
run time	31.67	27.26	25.75	24.19

6.5.4 并行 MVDR 自适应波束形成算法

根据 MVDR 自适应波束形成和前文提出的热点并行方案和优化策略，本节基于不同平台提出基于分布存储的 MVDR 波束形成算法、基于 GPU 的 MVDR 波束形成算法和基于 GPU 集群的 MVDR 波束形成算法。

6.5.4.1 基于分布存储的 MVDR 波束形成算法

本节提出一种基于分布存储的 MVDR 自适应波束形成算法（M-MVDR），见图 6.12。其中多进程间的任务划分以感兴趣的频点划分为主，将各频点的方位谱计算任务均匀地分配给所有进程。图中所示，root 进程负责读入声呐信号、进行 DFT 变换，接着将频域信号数据广播给所有进程；各进程接收到频域信号后开始计算本进程分配到的频点 MVDR 方位谱，接着将方位谱传回 root 进程；root 进程收集所有进程的方位谱并进行统计算出最终 MVDR 能量访问谱结果并输出。

6.5.4.2 基于 GPU 的 MVDR 波束形成算法

结合 DFT 并行方案、并行的厄尔米特矩阵特征分解方案和并行方位谱计算方案，采取 DFT 合并优化、雅克比迭代的 3 项优化策略、方位谱计算的 6 项优化策略，提出一种基于 GPU 的 MVDR 自适应波束形成算法（G-MVDR，见图 6.13）。CPU 端负责声呐信号输入，传输声呐信号到 GPU 端（H2D），启动 kernel 函数调用 GPU 进行并行的 DFT 变换，接着将频域信号传回 CPU 端（D2H）；CPU 端控制 GPU 循环计算各感兴趣频点的方位谱，首先 CPU 计算自相关矩阵并进行对角加载，将自相关矩阵传输到 GPU 端（H2D），启动 kernel 函数调用 GPU 进行并行雅克比迭代求解厄尔米特自相关矩阵的特征值和特征向量，接着启动 kernel 函数计算 MVDR 方位谱，然后循环计算下一个感兴趣频点的 MVDR 方位谱；完成 MVDR 方位谱计算后，将 MVDR 方位谱从 GPU 传回 CPU（D2H），并由 CPU 计算到达角度并输出。

6.5.4.3 基于 GPU 集群的 MVDR 波束形成算法

结合分布存储的 MVDR 波束形成算法（M-MVDR）和基于 GPU 的 MVDR 波束形成算法（G-MVDR），利用 GPU 加速计算分布存储 MVDR 算法中的 DFT 变

换、雅克比迭代特征分解和方位谱计算，提出基于 GPU 集群的 MVDR 波束形成算法（Gs-MVDR）。算法中 GPU 节点间的并行度主要依赖于感兴趣的频点数量，一般为几十到数百不等。

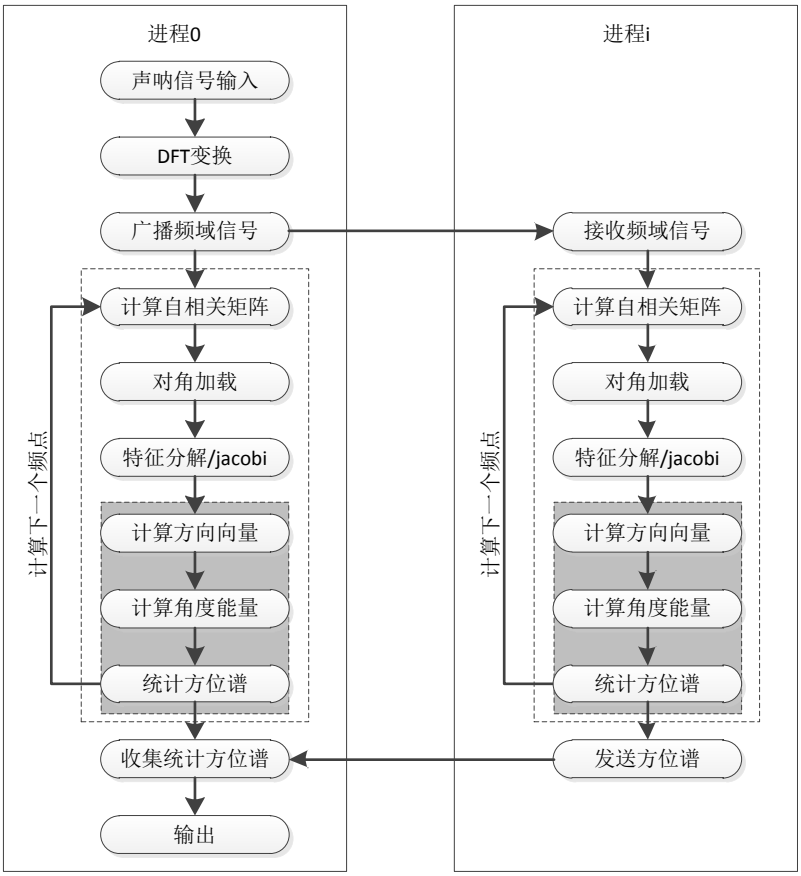


图 6.12 基于分布存储的 MVDR 波束形成算法（M-MVDR）

6.5.4 实验结果分析

6.5.4.1 MVDR 到达角度分析

实现基于 GPU 的宽频 MVDR 自适应波束形成算法，输入一组模拟的基元数为 1024 的声呐信号，统计方位谱见图 6.14。分析该图，在角度为 123°时，方位谱能量最高，故可判定目标位于到达角度为 123°的方向上。

6.5.4.2 MVDR 加速比分析

利用模拟生成的 6 组声呐信号数据（基元数分别为 64、128、256、512、1024 和 2048）作为输入，设定感兴趣频点数为 100，搜索角度为 180°，在本文实验平台（Xeon E5-2670 CPU 和 Tesla K20c GPU）上分别执行串行和 3 类并行的宽频 MVDR 自适应波束形成算法（M-MVDR、G-MVDR 和 Gs-MVDR），统计执行时间见表 6.15。表中数据显示，本章设计的这 3 类并行 MVDR 波束形成算法相对串行算法的执行时间明显减少。其中在单个节点内，当基阵规模小于 256 时，

M-MVDR 算法性能更佳；而当基阵规模大于 512 后，Gs-MVDR 算法的执行时间最短。

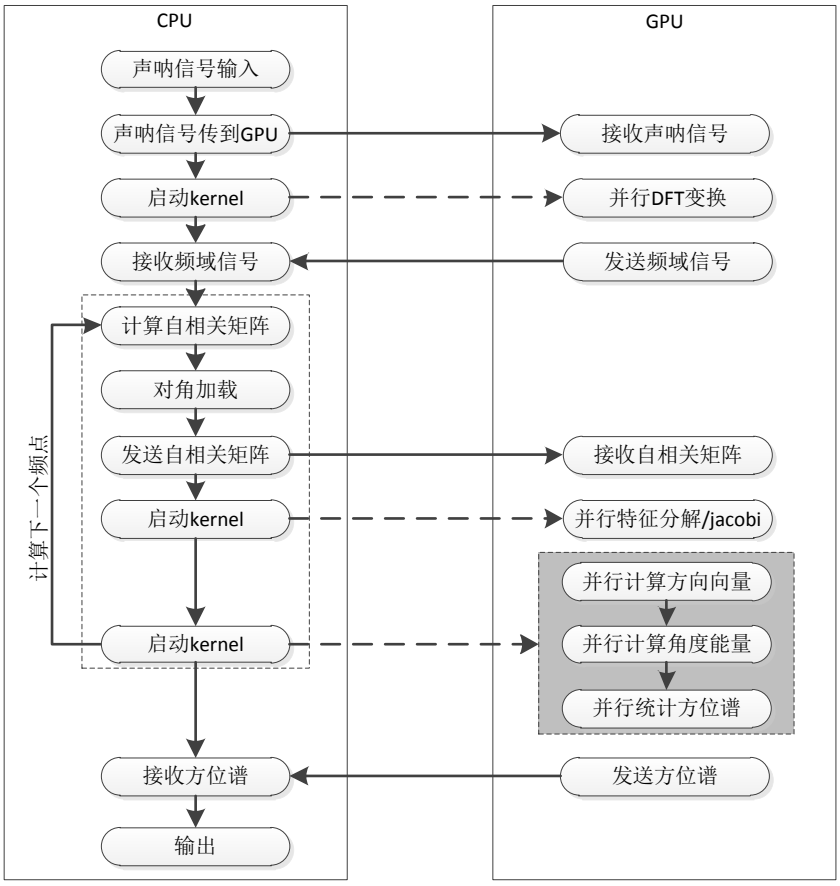


图 6.13 基于 GPU 的 MVDR 波束形成算法（G-MVDR）

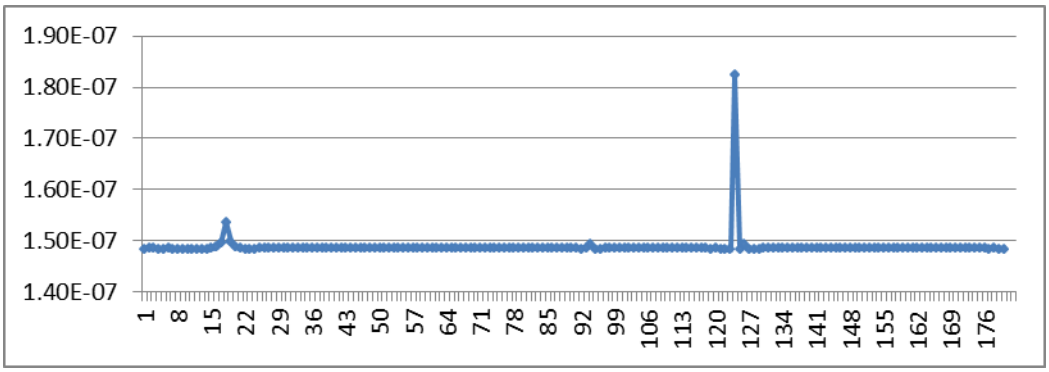


图 6.14 MVDR 方位谱

从实时性的角度分析，M-MVDR 算法和 G-MVDR 算法仅能支持 256 个基元、100 个感兴趣频点、180° 角的 MVDR 方位谱计算，这恰好能够适应当前舰艇的声呐基阵规模；对于未来数年内要搭载的 512 基元的声呐系统，则需要使用性能更强的 GPU，或者采用多个 GPU（Gs-MVDR 算法）来实现实时 MVDR 波束形成。

计算各并行 MVDR 波束形成算法相对串行 MVDR 算法的加速比，见表 6.16。表中数据显示，2 进程的 M-MVDR 算法加速 1.7~1.9 倍，16 进程的 M-MVDR 算

法加速 5.9~9.5 倍；在 K20c GPU 上运行 G-MVDR 算法加速 1.5~17.3 倍，Gs-MVDR 算法利用 2 个 K20c GPU 加速 2.5~30.7 倍。其中 Gs-MVDR 算法在处理 2048 个基元的声呐信号数据时获得最高 30.7 倍加速比。

表 6.15 MVDR 波束形成执行时间/ms

基元数	64	128	256	512	1024	2048
MVDR	174.48	621.70	2382.67	9567.53	37153.75	174981.62
M-MVDR(2)	94.09	329.38	1261.48	5162.34	20013.50	100723.97
M-MVDR(16)	29.37	76.80	250.37	1021.71	5681.78	28046.03
G-MVDR	112.88	227.02	533.58	1263.33	3283.07	10142.59
Gs-MVDR(2GPU)	71.21	124.99	301.39	712.30	1853.40	5707.23

表 6.16 并行 MVDR 加速比

基元数	64	128	256	512	1024	2048
M-MVDR(2)	1.9	1.9	1.9	1.9	1.9	1.7
M-MVDR(16)	5.9	8.1	9.5	9.4	6.5	6.2
G-MVDR	1.5	2.7	4.5	7.6	11.3	17.3
Gs-MVDR(2GPU)	2.5	5.0	7.9	13.4	20.0	30.7

6.6 本章小结

本章基于众核 GPU 平台，对声呐信号处理中的两类波束形成算法（DFT-CBF 和 MVDR）进行并行方案设计和性能优化研究。针对 DFT-CBF 算法的加速热点（DFT、CBF/Lofar 计算和频带能量整合）和 MVDR 算法的加速热点（DFT、厄尔米特矩阵特征分解、方位谱计算），设计并行方案，结合第 3~4 章的优化策略提出一系列的优化策略，并量化分析优化效果，提出了基于 GPU 的频域常规波束形成算法和 3 种并行的 MVDR 自适应波束形成算法。实验结果显示，基于 GPU 的 DFT-CBF 算法可以实时处理万基元基阵的声呐信号波束形成，最高获得 125.3 倍加速比；本章提出的 3 类并行 MVDR 波束形成算法均获得了理想的性能提升，当基阵规模小于 256 时基于分布存储的 M-MVDR 算法性能更佳；而当基阵规模大于 512 后，能够发挥多 GPU 性能的 Gs-MVDR 算法性能最高，最高加速 30.7 倍。

第七章　结论与展望

由于众核协处理器在浮点运算性能、性价比、能效比等方面的优势，多核与众核异构系统已在高性能计算领域广泛采用，大到超级计算机，小到工作站。但在具体应用研究和开发时，仍面临异构程序移植困难、众核体系结构性能优化困难、异构系统高效协同困难、数学原理/领域机理理解困难等挑战。本文针对众核体系结构性能优化困难和异构系统高效协同困难两方面的挑战开展研究，以 GPU 为例开展研究，重点关注 GPU 的访存优化和异构协同优化，设计 microbenchmark 测评感兴趣的特性，根据测评结果设计优化策略，并应用到高光谱影像降维和声呐信号处理两类实际应用中有效提升程序性能。

本文研究成果可以为异构程序性能建模、异构并行算法设计和性能优化提供有效参考；基于高光谱影像降维和声呐信号处理两个领域的研究能辅助领域科学家快速进行科学研究，也能为这两个领域应用的多核与众核的并行和优化研究和实现提供借鉴。

7.1 工作总结

本文概述了 GPU 硬件架构、执行核心、存储系统和 CPU/GPU 异构系统，分析了硬件架构可扩展性，测评了算术运算延迟，探索了分支执行顺序，提出了关于 GPU 存储优化和异构协同优化的一些思考。基于 GPU 存储优化和异构协同优化的思考展开研究工作，取得了一些研究成果，包括：

（1）基于微 benchmark 设计和构建 GPU 访存优化框架，应用于真实应用获得可观的性能收益。测评 GPU 存储单元的 thread 级访存延迟；提出 warp 级访存测评方法，设计并行访存测评实验并测评共享存储、常量存储、全局存储和纹理存储；探索私有变量数组在寄存器和局部存储的分配策略、共享存储 bank conflict 及避免、数据类型对全局存储访存带宽的影响；设计并总结 GPU 存储优化策略，构建 GPU 访存优化框架，并结合实例研究验证了优化策略和框架的实用性。

（2）在异构系统中探索主机端存储、zerocopy、计算与通信重叠和计算与计算重叠等优化技术，并用实例研究进行验证。通过微 benchmark 测评主机端存储的访存带宽、PCI-E 带宽、页锁定存储的注册和解除注册开销，提出主机端存储选择模型，并用实例研究验证模型的正确性。此外，针对 zerocopy 设计并验证两项优化方案，讨论计算与通信重叠、计算与计算重叠两类通用的异构协同优化技术。

（3） 提出并实现面向众核体系结构的高光谱影像降维框架。针对主成分分析、快速独立成分分析和最大噪声分数变换 3 类主流的高光谱影像降维算法，分

别在分布存储、共享存储和 GPU 三个层次设计协方差矩阵计算、PCA 变换、ICA 迭代、噪声估计（滤波）等热点并行方案；结合 GPU 访存优化和异构协同优化的研究成果，研究了性能优化策略及优化效果；提出面向众核体系结构的高光谱影像并行降维框架，基于 CPUs、GPUs 和 Phis 平台均有实现。实验结果显示框架中的并行降维算法的优秀性能和可扩展性，其中 Gs-PCA 算法最高加速 119.7 倍，Gs-FastICA 算法最高加速 106.6 倍，G-MNF 算法最高加速 86.9 倍。

（4） 提出并实现基于 GPU 的两类并行波束形成算法。针对 DFT-CBF 算法的 DFT 变换、CBF/Lofar 计算和频带能量整合统计，以及 MVDR 算法的 DFT 变换、双边雅克比迭代（厄尔米特矩阵特征分解）和方位谱统计等加速热点，设计了 GPU 并行映射方案，结合 GPU 访存优化和异构协同优化的研究成果研究了性能优化策略，并量化分析优化效果，实现基于 GPU 的 DFT-CBF 算法和三类并行 MVDR 自适应波束形成算法。实验分析并行算法加速比和实时性，其中，基于 GPU 的 DFT-CBF 算法可实时处理万基元基阵的声呐信号波束形成，最高加速 125.3 倍；Gs-MVDR 算法获得最高 30.7 倍加速比。

7.2 研究展望

基于众核体系结构的发展和本文的研究成果，未来可从以下几个方面开展相关研究工作：

（1） 进一步完善 GPU 访存优化框架，实现自动化或半自动化的 kernel 代码的访存分析和优化。本文从 microbenchmark 测评总结 GPU 存储特性，设计存储优化策略，构建了 GPU 访存优化框架。但该框架仍不够完善，部分优化策略可能还未考虑到，在实际应用开发时仍需人工分析代码，定位优化策略并手动优化。因此下一步将继续完善 GPU 访存优化框架，实现自动化或半自动化的代码转换。

（2） 利用 microbenchmark 测评其他众核体系结构产品，总结体系结构特性，设计优化策略，并在实际应用中提升程序性能。除了本文研究的 GPU 外，还有其他众核体系结构产品，比如 Xeon Phi。可以将本文研究方法平移到其他众核体系结构产品开展相关研究工作。

（3） 第 3 章 GPU 访存优化和第 4 章异构协同优化研究成果可应用到其他实际应用中，收获可观的性能加速。本文将第 3~4 章研究成果应用到了高光谱影像降维和声呐信号波束形成两个领域，获得了理想的加速效果。同理，未来也能够应用到其他很多领域的相关应用中，结合第 3~4 章研究成果和具体应用特性，设计出有针对性的优化方案，从而获得有益的性能提升。目前已有计划基于本文研究成果，研究间断 Galerkin 有限元方法（DG-FEM）全球地震波模拟的高效并行和优化。

致　谢

攻读博士学位期间，伴随着探索研究方向的迷茫、努力科研的忙碌、面临难题的焦躁、论文拒稿的沮丧和录用的喜悦，也离不开很多人的帮助、支持和鼓励。在博士论文完成之际，我衷心感谢所有给予过我指导、帮助和鼓励的老师、同学和亲友。

首先，向我的导师张卫民研究员致以崇高的敬意和深深的感谢！张老师高深的学术造诣、宽厚的待人态度、平易近人的长者风范以及敬业的工作态度，都值得我敬仰和学习。在读博之初，张老师就为我提供了一个广阔的平台，让我有机会在当时TOP500榜首的天河2号超级计算机上开展课题研究。在课题研究过程中，张老师作为海洋科学与工程研究院总工，也常在百忙之中抽空关心我的课题进展，指导我的研究方向。同时，张老师还怀着一颗宽阔的包容之心，包容我擅自更改研究课题、撰写对博士毕业“无用”的专著。在生活上，张老师考虑到我作为一名地方生的困难，给我安排了助研岗位，缓解了我拮据的生活。我在博士期间的每篇论文、每项成果都离不开张老师的支持和帮助，在此谨向他表示衷心的感谢和诚挚的敬意！

衷心感谢方建滨师兄对我无私的指导和帮助！方建滨师兄严谨的科研态度、开阔的视野和温润的待人方式，对我的博士期间学习、科研和生活帮助非常之大。我的大多数论文都经过方建滨师兄的修改和润色，特别是英文论文，限于我的英文水平，方建滨师兄常帮我一轮又一轮地修改。当我遇到迷茫和困难时，每次找方建滨师兄谈完心都能有所收获，找到方向，令我茅塞顿开。若没有方建滨师兄的指导和帮助，我的博士生涯将面临很多弯路和坎坷。再次衷心感谢方建滨师兄！

感谢我的硕士导师周海芳教授的指导和鼓励！我读博之初的很多研究方法和思想都来自周海芳老师，让我在读博初期取得了不少研究成果。周老师擅长教书育人，擅长调动学生积极性，每当我犹豫不决是否要做某项工作、或者情绪低落无法专心科研的适合，周老师都会给我正确的方向性指导和切实有效的鼓励，每次跟周老师谈完心里都有一种笃定。我的专著《GPU 编程与优化》的撰写和出版离不开周老师的鼓励和修订，当时我还在纠结课题研究和书籍撰写的时候，周老师的鼓励让我正式下定决心要写完该书并出版，并且在书籍修订过程中，周老师严谨精准的语言表达，帮我修正了很多不当的描述。在此，再次感谢周海芳老师！

特别感谢同门师妹高畅！我的专著能够正式出版，离不开高畅长达一个多月的代码和结果验证工作。高畅办事认真负责、工作谨慎细致，帮我处理了很多琐碎而复杂的工作。感谢同门小伙伴申小龙，帮我调过代码、借过服务器资源等，申小龙办事负责可靠，每次我需要他帮助的时候都能尽心尽力完成，甚至比我自

己办得还要靠谱。

感谢张理论老师帮我牵线搭桥，让我能够与多个应用领域交叉，比如第六章的声呐信号波束形成应用、北京大学的 3D 蒙特卡罗器件模拟和中国地质大学的间断 Galerkin 有限元方法地震波模拟。感谢王勇献老师和车永刚老师带领我们参加 PAC 竞赛。感谢包长春老师、赵军老师、杜云飞老师、吴建平老师、赵娟老师、赵延来老师和冷洪泽老师等人的帮助。

感谢同门的余意师姐、朱孟斌师兄、黄群博、刘柏年、段博恒、孙敬哲、张泽、吴应昂、邢威、朱祥茹、邢德、林士伟、陈妍、王品强、李松、吴茂永、邢翔。正是有你们，师门大家庭才有浓厚的学术氛围，有丰富多彩的生活，特别感谢余意师姐的关心和照顾。

感谢我的同学和朋友在读博期间的陪伴和帮助，他们是马俊波、邓明翥、明月伟、张军阳、于松平、陈呈、康文杰、李荣振、杨茂森、郝培培、王传立、黄辉、曾皓、陈洪义、王斌峰、汤振森等。

感谢欧阳登轶政委、王艳丰政委、寻兵斌队长和孙靖副政委对我的关心和照顾，特别感谢欧阳登轶政委对我的教导和关照，引导我向党靠拢，帮助我入党。

感谢我的父母，是你们含辛茹苦把我养育成人，为我付出毕生心血而不求回报，你们的嘱咐和唠叨是我学习和生活中最强有力的支柱，你们的殷切期盼是我分发拼搏的动力，言语已不能表达我对你们的感激之情，谢谢！

最后，感谢百忙之中抽出时间对我的论文进行评审的各位专家和教授！

参考文献

[1] Tendler J M, Dodson J S, Fields J S, et al. POWER4 system microarchitecture[J]. IBM Journal of Research and Development, 2002, 46(1): 5-25.

[2] Wikipedia: https://zh.wikipedia.org/wiki/多核心處理器

[3] Palmer J F. The Intel® 8087 numeric data processor[C]//Proceedings of the May 19-22, 1980, national computer conference. ACM, 1980: 887-893.

[4] 王恩东, 张清, 沈柏, 等. MIC 高性能计算编程指南[M]. 中国水利水电出版社, 2012.

[5] Jeffers J, Reinders J. Intel Xeon Phi coprocessor high-performance programming[M]. Newnes, 2013.

[6] Meuer H, Strohmaier E, Dongarra J, et al. Top500 supercomputing sites[J]. http://www.top500.org/

[7] Wikipedia: https://en.wikipedia.org/wiki/Graphics_processing_unit

[8] nvidia. Whitepaper of nvidia's next generation CUDATM compute architecture KeplerTM GK110/210. http://international.download.nvidia.com/pdf/kepler/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf

[9] Wilt N. The cuda handbook: A comprehensive guide to gpu programming[M]. Pearson Education, 2013.

[10] Reed B C. the Physics of the Manhattan Project[M]. Springer, 2014.

[11] Methods in computational physics[M]. Academic Press, 1972.

[12] Challenges G. High performance computing and communications[J]. The FY, 1992: 7.

[13] Seager M. Operational machines: ASCI white[C]//7th Workshop on Distributed Supercomputing, Durango, CO. 2003.

[14] Dan M, Zhihong Z, Mingyu C. High Productivity Computing Systems[J]. Journal of Computer Research and Development, 2005, 4: 004.

[15] Kogge P, Bergman K, Borkar S, et al. Exascale computing study: Technology challenges in achieving exascale systems[J]. 2008.

[16] Thornton J E. The cdc 6600 project[J]. Annals of the History of Computing, 1980, 2(4): 338-348.

[17] Bailey D H. Extra high speed matrix multiplication on the Cray-2[J]. SIAM Journal on Scientific and Statistical Computing, 1988, 9(3): 603-607.

[18] Warren M S, Salmon J K, Becker D J, et al. Pentium pro inside: I. a treecode at 430 gigaflops on asci red, ii. price/performance of $50/mflop on loki and hyglac[C]//Supercomputing, ACM/IEEE 1997 Conference. IEEE, 1997: 61-61.

[19] Wikipedia: https://en.wikipedia.org/wiki/Roadrunner_(disambiguation)

[20] green500: http://green500.org/lists/green201506

[21] Li E, Yang L, Wang B, et al. SURF cascade face detection acceleration on Sandy Bridge processor[C]//Computer Vision and Pattern Recognition Workshops (CVPRW), 2012 IEEE Computer Society Conference on. IEEE, 2012: 41-47.

[22] Treibig J, Hager G, Hofmann H G, et al. Pushing the limits for medical image reconstruction on recent standard multicore processors[J]. International Journal of High Performance Computing Applications, 2012: 162-177.

[23] Cui T, Franchetti F. A multi-core high performance computing framework for probabilistic solutions of distribution systems[C]. IEEE Power and Energy Society General Meeting, 2012.

[24] Wende F, Cordes F, Steinke T. On improving the performance of multi-threaded CUDA applications with concurrent kernel execution by kernel reordering[C]. Symposium on Application Accelerators in High-Performance Computing, 2012: 74-83.

[25] Xiao S, Balaji P, Zhu Q, et al. VOCL: An optimized environment for transparent virtualization of graphics processing units[C]// 2012 Innovative Parallel Computing, 2012.

[26] Hoshino T, Maruyama N, Matsuoka S, et al. CUDA vs OpenACC: Performance case studies with Kernel benchmarks and a memory-bound CFD application[C]// 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid , 2013: 136-143.

[27] Lee S, Eigenmann R. OpenMPC: Extended OpenMP for efficient programming and tuning on GPUs[J]. International Journal of Computational Science and Engineering, 2013,8(1): 4-20.

[28] Dolbeau R, Bihan S, Bodin F. HMPP: A hybrid multi-core parallel programming environment[C]//Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007). 2007.

[29] Pai S, Govindarajan R, Thazhuthaveetil M J. Fast and efficient automatic memory management for GPUs using compiler-assisted runtime coherence scheme[C]. Parallel Architectures and Compilation Techniques - Conference Proceedings, 2012: 33-42.

[30] Yang Y, Xiang P, Kong J, et al. A GPGPU compiler for memory optimization and parallelism management[J]. ACM SIGPLAN Notices, 2010,45(6):86-97.

[31] Lee S, Min S J, Eigenmann R. OpenMP to GPGPU: A compiler framework for automatic translation and optimization[C]// Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2009:101-110.

[32] Potluri S, Venkatesh A, Bureddy D, et al. Efficient intra-node communication on Intel-MIC clusters[C]// 13th IEEE/ACM International Symposium on Cluster, Cloud,

and Grid Computing, 2013: 128-135.

[33] Si M, Ishikawa Y, Tatagi M. Direct MPI library for intel xeon phi co-processors[C]// IEEE 27th International Parallel and Distributed Processing Symposium Workshops and PhD Forum, 2013: 816-824.

[34] Newburn C J, Dmitriev S, Narayanaswamy R, et al. Offload compiler runtime for the intel Xeon Phi coprocessor[C]// IEEE 27th International Parallel and Distributed Processing Symposium Workshops and PhD Forum, 2013: 1213-1224.

[35] Noack M. HAM-Heterogenous Active Messages for Efficient Offloading on the Intel Xeon Phi[J]. ZIB, Takustr, 2014, 7: 14195.

[36] Ravi N, Yang Y, Bao T, et al. Apricot: An optimizing compiler and productivity tool for x86-compatible many-core coprocessors[C]// Proceedings of the International Conference on Supercomputing, 2012: 47-57.

[37] Wong H, Papadopoulou M M, Sadooghi-Alvandi M, et al. Demystifying GPU microarchitecture through microbenchmarking[C]//Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on. IEEE, 2010: 235-246.

[38] Keckler S W, Dally W J, Khailany B, et al. GPUs and the future of parallel computing[J]. IEEE Micro, 2011 (5): 7-17.

[39] Li Y, Chi H, Xia L, et al. Accelerating the scoring module of mass spectrometry-based peptide identification using GPUs[J]. BMC bioinformatics, 2014, 15(1): 1.

[40] Ryoo S, Rodrigues C I, Baghsorkhi S S, et al. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA[C]//Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. ACM, 2008: 73-82.

[41] Micikevicius P. 3D finite difference computation on GPUs using CUDA[C]//Proceedings of 2nd workshop on general purpose processing on graphics processing units. ACM, 2009: 79-84.

[42] Zhao K, Chu X. G-BLASTN: accelerating nucleotide alignment by graphics processors[J]. Bioinformatics, 2014, 30(10): 1384-1391.

[43] Jang B, Schaa D, Mistry P, et al. Exploiting memory access pat-terns to improve memory performance in data-parallel architectures[J]. Parallel and Distributed Systems, IEEE Transactions on, 2011, 22(1): 105-118.

[44] Chen G, Wu B, Li D, et al. Porple: An extensible optimizer for portable data placement on gpu[C]//Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, 2014: 88-100.

[45] David B.Kirk, Wen-mei W.Hwu. Programing massively parallel processors:A Hands-on Approach, Second Edition[M]. Beijing:Tsinghua University Press. 2013.

[46] CUDA C programming guide(v7.5). NVIDIA Corporation . 2016. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[47] Volkov V, Demmel J W. Benchmarking GPUs to tune dense linear algebra[C]// High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for. IEEE, 2008: 1-11.

[48] Baghsorkhi S S, Gelado I, Delahaye M, et al. Efficient perfor-mance evaluation of memory hierarchy for highly multithreaded graphics processors[C]//ACM SIGPLAN Notices. ACM, 2012, 47(8): 23-34.

[49] Meltzer R, Zeng C, Cecka C. Micro-benchmarking the C2070[C]//GPU Technology Conference. 2013.

[50] Mei X, Zhao K, Liu C, et al. Benchmarking the memory hierarchy of modern GPUs[M]//Network and Parallel Computing. Springer Berlin Heidelberg, 2014: 144-156.

[51] Mei X, Chu X. Dissecting GPU Memory Hierarchy through Microbenchmarking[J]. arXiv preprint arXiv:1509.02308, 2015.

[52] Ma W, Agrawal G. An integer programming framework for optimizing shared memory use on GPUs[C]// High Performance Computing (HiPC), 2010 International Conference on. IEEE, 2010: 1-10.

[53] Yang Y, Xiang P, Kong J, et al. A GPGPU compiler for memory optimization and parallelism management[C]// ACM Sigplan Notices. ACM, 2010, 45(6): 86-97.

[54] Zhang E Z, Jiang Y, Guo Z, et al. On-the-fly elimination of dynamic irregularities for GPU computing[C]//ACM SIGARCH Computer Architecture News. ACM, 2011, 39(1): 369-380.

[55] Wu B, Zhao Z, Zhang E Z, et al. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu[C]// ACM SIGPLAN Notices. ACM, 2013, 48(8): 57-68.

[56] Van W B, Maassen J, Seinstra F J, et al. Performance models for CPU-GPU data transfers[C]//Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on. IEEE, 2014: 11-20.

[57] Boyer M, Meng J, Kumaran K. Improving GPU performance prediction with data transfer modeling[C]//Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International. IEEE, 2013: 1097-1106.

[58] Meswani M R, Carrington L, Unat D, et al. Modeling and predicting performance of high performance computing applications on hardware accelerators[J]. International Journal of High Performance Computing Applications, 2013, 27(2): 89-108.

[59]Ta T, Choo K, Tan E, et al. Accelerating DynEarthSol3D on tightly coupled CPU–GPU heterogeneous processors[J]. Computers & Geosciences, 2015, 79: 27-37.

[60] Fang L, Wang M, Li D, et al. CPU/GPU near real-time preprocessing for ZY-3 satellite images: Relative radiometric correction, MTF compensation, and geocorrection[J]. ISPRS Journal of Photogrammetry and Remote Sensing, 2014, 87:

229-240.

[61] Wu J, JaJa J. High Performance FFT Based Poisson Solver on a CPU-GPU Heterogeneous Platform[C]// Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on. IEEE, 2013: 115-125.

[62] Wu Y, Gao L, Zhang H, et al. Real-time implementation of optimized maximum noise fraction transform for feature extraction of hyperspectral images[J]. Journal of Applied Remote Sensing. 2014,8: (084797-1-084797-16)

[63] Green RO, Eastwood ML, Sarture CM, et al. Imaging spectroscopy and the airborne visible/infrared imaging spectrometer(AVIRIS)[J]. Remote Sensing of Environment, 1998, 65(3): 227-248

[64] Green AA, Berman M, Switzer P, et al. A transformation for ordering multispectral data in terms of image quality with implications for noise removal[J]. IEEE Trans on Geoscience and Remote Sensing, 2000, 26(1): 65-74

[65] Kaarna A, Zemcik P, Kalviainen H, et al. Compression of multispectral remote sensing images using clustering and spectral reduction[J]. IEEE Trans on Geoscience and Remote Sensing, 2000, 38(2): 1073-1082

[66] Scott DW, Thompson JR. Probability density estimation in higher dimensions[C]// Computer Science and Statistics: Proc of the 15th Symp on the Interface. Amsterdam: North-Holland, 1983: 173–179

[67] Jolliffe IT. Principal Component Analysis[M]. New York: Sprinner, 2002

[68] Hyvärinen A, Oja E. Independent component analysis: Algorithm and applications[J]. Neural Networks, 2000, 13(4/5): 411-430

[69] Tenenbaum J, De Silva V, Langford JC. A global geometric framework for nonlinear dimensionality reduction[J]. Science, 2000, 290(5500): 2319-2323

[70] Roweis ST, Saul LK. Nonlinear dimensionality reduction by locally linear embedding[J]. Science, 2000, 290(5500): 2323-2326

[71] Hyvärinen A. Fast and robust fixed-point algorithms for independent component analysis[J]. IEEE Trans on Neural Networks, 1999, 10(3): 626-634

[72] Hyvärinen A. The fixed-point algorithm and maximum likelihood estimation for independent component analysis[J]. Neural Processing Letters, 1999, 10(1): 1-5

[73] Valencia D, Lastovetsky A, O'Flynn M, et al. Parallel processing of remotely sensed hyperspectral images on heterogeneous networks of workstations using HeteroMPI[J]. International Journal of High Performance Computing Applications, 2008, 22(4): 386-407

[74] Plaza J, Plaza A, Pérez R, et al. Parallel classification of hyperspectral images using neural networks[J]. Computational Intelligence for Remote Sensing Studies in Computational Intelligence, 2008, 133: 193-216

[75] Plaza A, Valencia D, Plaza J, et al. Commodity cluster-based parallel processing of hyperspectral imagery[J]. Journal of Parallel and Distributed Computing,

2006, 66(3): 345-358.

[76] Dong C, Zhao H, Li N, et al. Para-GMRF: parallel algorithm for anomaly detection of hyperspectral image[C]//International Symposium on Multispectral Image Processing and Pattern Recognition. International Society for Optics and Photonics, 2007: 67891V-67891V-7.

[77] Dong C, Tian L. Accelerating Relevance-Vector-Machine-Based Classification of Hyperspectral Image with Parallel Computing[J]. Mathematical Problems in Engineering, 2012, 2012.

[78] Plaza A, Plaza J, Paz A. Improving the scalability of parallel algorithms for hyperspectral image analysis using adaptive message compression[C]//Geoscience and Remote Sensing Symposium, 2009 IEEE International, IGARSS 2009. IEEE, 2009, 4: IV-196-IV-199.

[79] Tzeng Y C, Fan K T, Chen K S. A parallel differential box-counting algorithm applied to hyperspectral image classification[J]. Geoscience and Remote Sensing Letters, IEEE, 2012, 9(2): 272-276.

[80] Plaza A, Plaza J, Paz A, et al. Parallel hyperspectral image and signal processing [applications corner][J]. Signal Processing Magazine, IEEE, 2011, 28(3): 119-126.

[81] Bernabé S, Plaza A. Commodity Cluster-Based Parallel Implementation of an Automatic Target Generation Process for Hyperspectral Image Analysis[C]//Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on. IEEE, 2011: 1038-1043.

[82] Plaza A, Plaza J, Valencia D. AMEEPAR: Parallel morphological algorithm for hyperspectral image classification on heterogeneous networks of workstations[M]//Computational Science–ICCS 2006. Springer Berlin Heidelberg, 2006: 24-31.

[83] Du H, Qi H, Peterson G D. Parallel ICA and its hardware implementation in hyperspectral image analysis[C]//Defense and Security. International Society for Optics and Photonics, 2004: 74-83.

[84] Plaza A, Valencia D, Plaza J, et al. Parallel Implementation of Hyperspectral Image Processing Algorithms[C]//Geoscience and Remote Sensing Symposium, 2006. IGARSS 2006. IEEE International Conference on. IEEE, 2006: 940-943.

[85] Gonzalez C, Sánchez S, Paz A, et al. Use of FPGA or GPU-based architectures for remotely sensed hyperspectral image processing[J]. INTEGRATION, the VLSI journal, 2013, 46(2): 89-103.

[86] Plaza A, Plaza J, Vegas H. Improving the performance of hyperspectral image and signal processing algorithms using parallel, distributed and specialized hardware-based systems[J]. Journal of Signal Processing Systems, 2010, 61(3): 293-315.

[87] Setoain J, Tenllado C, Prieto M, et al. Parallel hyperspectral image processing on commodity graphics hardware[C]//Parallel Processing Workshops, 2006. ICPP 2006 Workshops. 2006 International Conference on. IEEE, 2006: 8 pp.-472.

[88] Sánchez S, Ramalho R, Sousa L, et al. Real-time implementation of remotely sensed hyperspectral image unmixing on GPUs[J]. Journal of Real-Time Image Processing, 2012: 1-15.

[89] Platoš J, Gajdoš P, Krömer P, et al. Non-negative matrix factorization on GPU[C]// Proc of the 2nd Int Conf on Networked Digital Technologies. Heidelberg: Springer, 2010: 21-30

[90] Ramalho R, Tomas P, Sousa L. Efficient independent component analysis on a GPU[C]// Proc of the 10th IEEE Int Conf on Computer and Information Technology. Piscataway, NJ: IEEE Computer Society, 2010: 1128-1133

[91] Du Q, Li X. Parallel random selection and projection for hyperspectral image analysis[C]//SPIE Remote Sensing. International Society for Optics and Photonics, 2014: 924702-924702-6.

[92] Rosario-Torres S, Vélez-Reyes M. Speeding up the MATLAB™ hyperspectral image analysis toolbox using GPUs and the Jacket toolbox[C]//Hyperspectral Image and Signal Processing: Evolution in Remote Sensing, 2009. WHISPERS'09. First Workshop on. IEEE, 2009: 1-4.

[93] Heras D B, Argüello F, Gómez J L, et al. Towards real-time hyperspectral image processing, a GP-GPU implementation of target identification[C]//Intelligent Data Acquisition and Advanced Computing Systems (IDAACS), 2011 IEEE 6th International Conference on. IEEE, 2011, 1: 316-321.

[94] Bernabe S, López S, Plaza A, et al. GPU implementation of an automatic target detection and classification algorithm for hyperspectral image analysis[J]. Geoscience and Remote Sensing Letters, IEEE, 2013, 10(2): 221-225.

[95] Sánchez S, Plaza A. Parallel hyperspectral image compression using iterative error analysis on graphics processing units[C]//Geoscience and Remote Sensing Symposium (IGARSS), 2012 IEEE International. IEEE, 2012: 3474-3477.

[96] Santos L, Magli E, Vitulli R, et al. Highly-parallel GPU architecture for lossy hyperspectral image compression[J]. Selected Topics in Applied Earth Observations and Remote Sensing, IEEE Journal of, 2013, 6(2): 670-681.

[97] Wu Z, Wang Q, Plaza A, et al. Parallel Spatial–Spectral Hyperspectral Image Classification With Sparse Representation and Markov Random Fields on GPUs[J].

[98] Wu Z, Liu J, Plaza A, et al. GPU Implementation of Composite Kernels for Hyperspectral Image Classification[J]. 2015.

[99] Mercier G, Lennon M. Support vector machines for hyperspectral image classification with spectral-based kernels[C]//Geoscience and Remote Sensing Symposium, 2003. IGARSS'03. Proceedings. 2003 IEEE International. IEEE, 2003, 1:

288-290.

[100] Tarabalka Y, Haavardsholm T V, Kåsen I, et al. Real-time anomaly detection in hyperspectral images using multivariate normal mixture models and GPU processing[J]. Journal of Real-Time Image Processing, 2009, 4(3): 287-300.

[101] Hossam M A, Ebied H M, Abdel-Aziz M H, et al. Accelerated hyperspectral image recursive hierarchical segmentation using GPUs, multicore CPUs, and hybrid CPU/GPU cluster[J]. Journal of Real-Time Image Processing, 2014: 1-20.

[102] Setoain J, Prieto M, Tenllado C, et al. GPU for parallel on-board hyperspectral image processing[J]. International Journal of High Performance Computing Applications, 2008, 22(4): 424-437.

[103] Paz A, Plaza A. Clusters versus GPUs for parallel target and anomaly detection in hyperspectral images[J]. EURASIP Journal on Advances in Signal Processing, 2010: 35.

[104] Gordon, C. a generalization of the maximum noise fraction transform[J]. IEEE Transactions on Geoscience and Remote Sensing, 2000, 38(1): 608-610.

[105] 李启虎. 声呐信号处理引论[M]. 北京:科学出版社. 2012.

[106] Winder A A. Sonar system technology. IEEE Trans., 1975, SU-22.

[107] Schmidt R. Multiple emitter location and signal parameter estimation[J]. IEEE transactions on antennas and propagation, 1986, 34(3): 276-280.

[108] Roy R, Kailath T. ESPRIT-estimation of signal parameters via rotational invariance techniques[J]. IEEE Transactions on Acoustics, Speech, and Signal Processing, 1989, 37(7): 984-995.

[109] Cox H, Zeskind R, Owen M. Robust adaptive beamforming[J]. IEEE Transactions on Acoustics, Speech, and Signal Processing, 1987, 35(10): 1365-1376.

[110] Capon J. High-resolution frequency-wavenumber spectrum analysis[J]. Proceedings of the IEEE, 1969, 57(8): 1408-1418.

[111] Owsley N L. Systolic array adaptive beamforming[R]. naval underwater systems center new London CT new London lab, 1987.

[112] Kim J S, Lee J H. MVDR method using subband decomposition for high frequency resolution in passive sonar system[J]. UDT Korea, 2002.

[113] Ahmed E, Mahmoud K R, Hamad S, et al. Using Parallel Computing for Adaptive Beamforming Applications[C]//Progress In Electromagnetism Research Symposium Proceedings, Cambridge, USA. 2010: 296-299.

[114] Yun S W, Jin Y, Hyeon S, et al. Implementation of 4-element beamforming antenna module for B4G base station system using GPU[C]//The 18th IEEE International Symposium on Consumer Electronics (ISCE 2014). IEEE, 2014: 1-2.

[115] Phuong T Y, Lee J G. Design space exploration of SW beamformer on GPU[J]. Concurrency and Computation: Practice and Experience, 2015, 27(7):

1718-1733.

[116] Chen J, Yiu B Y S, So H K H, et al. Real-time GPU-based adaptive beamformer for high quality ultrasound imaging[C]//2011 IEEE International Ultrasonics Symposium. IEEE, 2011: 474-477.

[117] Chen J, Alfred C H, So H K H. Design considerations of real-time adaptive beamformer for medical ultrasound research using FPGA and GPU[C]//Field-Programmable Technology (FPT), 2012 International Conference on. IEEE, 2012: 198-205.

[118] Fathi Y, Mahloojifar A, Asl B M. GPU-based adaptive beamformer for medical ultrasound imaging[C]//2012 19th Iranian Conference of Biomedical Engineering (ICBME). 2012.

[119] Asen J P, Buskenes J I, Nilsen C I C, et al. Implementing capon beamforming on a GPU for real-time cardiac ultrasound imaging[J]. IEEE transactions on ultrasonics, ferroelectrics, and frequency control, 2014, 61(1): 76-85.

[120] Kjeldsen T, Lassen L, Hemmsen M C, et al. Synthetic Aperture Sequential Beamforming implemented on multi-core platforms[C]//2014 IEEE International Ultrasonics Symposium. IEEE, 2014: 2181-2184.

[121] Jin X L, Yang Y X, Yang S, et al. Parallel implementation of wideband MVDR beamforming on multi-DSP for sonar system[C]//Signal Processing, Communications and Computing (ICSPCC), 2011 IEEE International Conference on. IEEE, 2011: 1-4.

[122] Han Y, Fu J, Zhou F, et al. Distributed and parallel subarray beamforming for near-field 3D sonar imaging[C]//Computer Science and Network Technology (ICCSNT), 2013 3rd International Conference on. IEEE, 2013: 1021-1025.

[123] Buskenes J I, Åsen J P, Nilsen C I C, et al. An optimized GPU implementation of the MVDR beamformer for active sonar imaging[J]. IEEE Journal of Oceanic Engineering, 2015, 40(2): 441-451.

[124] Martins N E, Jesus S M. Blind Estimation of the Ocean Acoustic Channel by Time–Frequency Processing[J]. IEEE Journal of Oceanic Engineering, 2006, 31(3): 646-656.

[125] Pan Y X, Li D W, Duan Y, et al. Predicting protein subcellular location using digital signal processing[J]. Acta biochimica et biophysica Sinica, 2005, 37(2): 88-96.

[126] Lax P. Mathematics and its application[J]. The Mathematical Intelligenzer. 1986,(8):14-17.

[127] 胡冰. 遥感图像融合并行算法的研究及实现[D]. 国防科学技术大学. 2006.

作者在学期间取得的学术成果

一 学术论文

[1] **Minquan Fang**, Jianbin Fang and Weimin Zhang. Efficient and portable parallel framework for hyperspectral image dimensionality reduction on heterogeneous platforms[J]. Journal of Applied Remote Sensing. 2017, 11(1): 015022 (SCI 源刊，doi: 10.1117/1.JRS.11.015022)

[2] **Minquan Fang**, Yi Yu, Weimin Zhang, Heng Wu, Mingzhu Deng, Jianbin Fang. High Performance Computing of Fast Independent Component Analysis for Hyperspectral Image Dimensionality Reduction on MIC-based Clusters[C]. The Eighth International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2) on ICPP. 2015.(EI: 20160401838015)

[3] **方民权**, 张卫民, 周海芳. 集成众核上快速独立成分分析降维并行算法[J].计算机研究与发展. 2016, 53(5):1136-1146.（EI: 20162302456186）

[4] **方民权**, 周海芳, 张卫民, 申小龙. GPU 上高光谱快速 ICA 降维并行算法[J]. 国防科大学报. 2015,37(4):65-70.（EI: 20154101350231）

[5] **方民权**,张卫民,高畅,方建滨. 多核与众核上 MNF 并行算法与性能优化[J]. 软件学报.2015.12, 26(S2):247-256.（EI: 20160501878092）

[6] **方民权**, 周海芳, 申小龙. 基于GPU的高光谱PCA降维多级并行算法[J]. 东北大学学报（自然科学版）. 2014.10, 35(S1): 238-243. （EI: 20152000840923）

[7] **方民权**, 张卫民, 张理论, 伍恒, 方建滨. 面向 MIC 的高光谱影像降维多级并行算法及性能优化[J]. 东北大学学报（自然科学版）. 2014.10, 35(S1):25-30.（EI: 20152000840886）

[8] **方民权**, 张卫民, 张理伦, 曾琅, 刘晓彦, 尹龙祥. 基于集成众核的 3D 蒙特卡罗半导体器件模拟[J]. 计算机工程与科学. 2015.4, 37(4): 621-627.

[9] **方民权**, 张卫民, 方建滨. 海洋数值模式并行化研究综述[C]. 第 33 届中国气象学会年会. 2016.

[10] Yin L, **Fang M**, Zeng L, et al. Accelerated 3D full band self-consistent ensemble Monte Carlo device simulation utilizing intel MIC co-processors on tianhe II[C]// Computational Electronics (IWCE), 2015 International Workshop on. IEEE, 2015: 1-4. （EI: 20160801971593）

[11] 高畅, 周海芳, **方民权**. 基于GPU 的MNF 并行算法与性能优化[C]. 全国高性能计算学术年会. 2015: 575-578.

[12] Gao Chang, Zhou Haifang, **Fang Minquan**. Parallel Algorithm and Performance Optimization of Kernel Principal Component Analysis on GPUs for

Dimensionality Reduction of HSI[C]. CCF HPC China 2016, 611-614.

[13] 周海芳, 高畅, **方民权**. 基于 CUBLAS 和 CUDA 的 MNF 并行算法设计与优化[J]. 湖南大学学报.（EI 源刊，已录用，待出版）

[14] **Minquan Fang**, Jianbin Fang, Weimin Zhang and Haifang Zhou. Benchmarking the GPU memory at the warp-level[J]. Journal of Parallel Computing. (SCI 源刊，CCF B 类期刊，二审中)

二 学术专著

[15] **方民权**, 张卫民, 方建滨, 周海芳, 高畅. GPU 编程与优化——大众高性能计算[M]. 北京:清华大学出版社. 2016.（ISBN 9787302446422）

三 软件著作权

[16] 周海芳, **方民权**. 软件著作权：基于 CPU/GPU 异构系统的高光谱遥感影像降维并行处理软件[HIDRPPS]. 2015. (登记号: 2015SR225820)

[17] 周海芳, **方民权**, 高畅. 软件著作权：基于集成众核的高光谱影像并行降维软件[HIPDR-MIC]. 2017. (登记号: 2017SR027552)

[18] 周海芳, 高畅, **方民权**. 软件著作权：基于众核 GPU 的高光谱非线性降维并行处理软件[HINLDRPPS]. 2017. (登记号: 2017SR027547)