论文题目 基于 CUDA 架构的高性能图像处理程序设计

专业学位类别 工 程 硕 士

学 号 201192020283

作 者 姓 名 张 琳

指 导 教 师 刘欣刚 副教授

分类号 _____ 密级 _____

UDC [注1] _____

学 位 论 文

基于 CUDA 架构的高性能图像处理程序设计

（题名和副题名）

张琳

（作者姓名）

指导教师 刘欣刚 副教授

电子科技大学 成 都

刘丽君 高 工

沈阳开发军地两用人才培训中心 沈 阳

（姓名、职称、单位名称）

申请学位级别 硕士 专业学位类别 工程硕士

工程领域名称 软 件 工 程

提交论文日期 **2014.9** 论文答辩日期 **2014.11**

学位授予单位和日期 电子科技大学 2014 年 12 月 25 日

答辩委员会主席 _____

评阅人_____

注 1：注明《国际十进分类法 UDC》的类号。

CUDA ARCHITECTURE-BASED HIGH-PERFORMANCE IMAGE PROCESSING PROGRAM DESIGN

A Master Thesis Submitted to

University of Electronic Science and Technology of China

Major: **Master of Engineering**

Author: **Zhang Lin**

Advisor: **Prof. Liu Xingang**

School : **School of Elecrtonic Engineering**

独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，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或撰写过的研究成果，也不包含为获得电子科技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。

作者签名：_____ 日期：　　年　　月　　日

论文使用授权

本学位论文作者完全了解电子科技大学有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权电子科技大学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。

（保密的学位论文在解密后应遵守此规定）

作者签名：_____ 导师签名：_____

日期：　　年　　月　　日

摘　要

高性能计算，又称超级计算，是指在短时间内完成更多的数据计算量，以完成更多的计算任务。云计算是近年来逐渐兴起的一种并行的分布式计算应用的概念与模式，其实现方式是将大量的计算资源通过网络连接到一起，构成一个具有强大的计算能力和并行处理能力的计算资源池，从而向用户提供高性能计算服务。从本质上看，云计算是为了实现高性能计算而产生的一种新的处理模式。由此可见，高性能计算的应用广泛，形式日新月异，对其研究具有重要的理论与现实意义。

由于 GPU 具有强大的计算能力，并且发展迅速，使其应用于通用的高性能计算，成为了一种必然的结果。CUDA 是由美国 NVIDIA 公司为其生产的 GPU 产品开发的通用计算程序与硬件计算资源之间交互的接口。CUDA 架构用于实现 GPU 设备的通用计算应用，具有高并行性、高带宽、性价比高、易于编程等优势。基于这些优势，基于 CUDA 的高性能计算，成为近年来学界研究的重点，在应用上得到了快速的发展。

虽然 CUDA 具有如上所述的诸多优势，但是，其仍然有许多局限性，例如，CUDA 对多分支和并行化低的应用程序执行效率不高，不适合整型数据运算，对处理的任务的细分程度及方式直接影响到 CUDA 应用程序的性能，并且，编程语言语法的特异性也限制了 CUDA 在通用高性能计算领域的应用。

为了克服基于 CUDA 架构的高性能计算在实际应用中存在的问题，从产品应用的角度出发，重点研究了对图像处理任务的并行化分解，并从控制任务与计算任务分离以及存储器优化的角度对应用程序进行了优化，以适应 CUDA 架构实现高性能计算的原理，从而进一步提升系统的性能。

本文首先介绍了高性能计算与云计算的相关概念，以及基于 GPU 的高性能计算的发展历程，在此基础上，介绍了 CUDA 架构下 GPU 编程的特点，总结了优化 CUDA 程序的技术方案，最终使用 CUDA C 语言实现了一个图像快速插值变换程序和图像快速滤波处理程序，开发了一个集成测试平台，并通过实验数据验证了基于 CUDA 架构实现的图像处理系统在计算性能上的提升。

关键词：高性能计算，CUDA 架构，并行任务分解，GPU 编程，高性能图像处理程序

ABSTRACT

High-performance computing, named super-computing, means to finish multiple computing task in as short time as possible. Cloud computing is a newly proposed therapy in recent years, means to apply distributed computing in a new way, and use multiple computing resources connected by network to work together, and so can provide high-performance services to the users. In nature, cloud computing is a new way to fit high-performance computing. So, we can come to a conclusion that, researching high-performance is very important for theory studying and applying.

The powerful computing capability and fast development make people considering using GPU for general-purpose high-performance computing automatically. Compute Unified Device Architecture （CUDA） is the interface of the general computing program and the hardware resources. It is developed by NVIDIA for its GPU product. CUDA has the advantage of high parallelism, high bandwidth, cost-effective and easy programming. Based on these advantages, applying CUDA to general-purpose high-performance computing becomes a hot issue in recent years, and developed very fast in application.

However, CUDA has many merits as described above, it still has many limitations, such as that, CUDA cannot manage multi-branch and low parallelization applications efficiently, and it is not fit for integrate data type. On the other hand, the way and level of dividing task may affect the performance of the application deeply. Difference of programming style also limited its application in general-purpose high-performance computing.

To solve the problems in the application of CUDA in general-purpose high-performance computing, from the view point of meeting demand of practical application, this article researched how to divide the image process task into small ones that can be executed parallel, and how to separate the computing tasks from control tasks, so that, the program can fit the workflow of CUDA, and enhance the performance of the whole system further.

This article introduced the concept of high-performance computing and cloud computing first, and the development of GPU based high-performance computing. On this basis, the feature of GPU programming under CUDA and optimization was

researched. At last, a fast image interpolation procedure and a fast image filter procedure were completed using CUDA C language, developed an integrated test platform. Experiment data show that image processing system using CUDA to realize has a better performance.

Key words: High-performance computing, CUDA, task dividing, GPU programming, High-performance image processing program

目 录

第一章 绪 论

1.1 课题研究的背景意义

1.1.1 高性能计算研究的意义

高性能计算（High-performance computing），又称超级计算，是指通过多个计算资源，并行地、协作完成某种高计算量、高负载率的计算任务。

高性能计算在数据加密、工程计算、DNA 分子测序和空间结构模拟、地质勘探、海洋信息监测、地质灾害预测、天气预报、城市交通调度、模拟实验、图形图像处理、数字水印、网络游戏、电子商务、搜索引擎等众多自然科学、社会科学以及工程应用领域发挥着重要的作用，成为多种学科领域中现代化的实现方法中不可或缺的高端计算工具，其应用研究越来越受到科研工作者和工程技术人员的重视与青睐。

同时，在公安政法领域，对图像处理速度和精度的要求十分突出，例如，在交通管控领域的视频分析与检索应用中，需要对分辨率较低的监控录像当中的图像进行插值放大与分析，以适应检索有效信息的需求；而在视频或者图像资料的图像质量较差时，则需要对图像进行降噪、增强等处理，从而提高视频或图像资料的可用性。

在生产实践中，以高性能的计算机为硬件基础的科学计算方法，在很大程度上，正在取代科学实验的作用，甚至在很多情况下，进行着实验科学所无法完成的研究工作，例如量子力学中的量子运动分析与量子力的分析等，在现有的实验条件下，是无法观察、实验和验证的，只能通过计算机模拟与推倒的方式来开展研究工作。并且，随着信息技术的不断发展，高性能计算已经成为众多产业链中至关重要的一环，成为提高各个学科创新能力与行业生产力的重要工具，带动着科学技术的进步，促进了相关产业的快速发展。因此，研究并解决高性能计算在应用中的问题，具有十分重要的理论与现实意义。

1.1.2 云计算的概念与高性能计算的联系

自 2006 年 8 月 9 日，Google 首席执行官艾瑞克•斯密特（Eric Schmidt）在搜索引擎大会（SES San Jose 2006）上首次提出“云计算”（Cloud Computing）的概念以来，云计算正式进入了科研工作者的视线，近年来，其理论与技术的发展可谓日新月异，应用也越来越广泛，成为目前计算机工程研究的热点问题。

云计算的核心思想是，将大量的计算资源使用网络连接到一起，通过一定的节点进行统一地调度与管理，构成一个具有强大的计算能力和并行处理能力的计算资源池，向用户提供服务，而提供计算资源的互联网络被形象地称为“云”[1]。

从云计算的核心思想与实现途径来看，高性能计算与云计算具有相同的技术发展出发点，这使得高性能计算成为了云计算发展的重要基础和性能提高的突破点，同时使对高性能计算的研究成为了一个古老而又崭新的学术研究方向，更加凸显了对高性能计算研究的现实意义。

1.1.3 使用图形处理器 GPU 实现高性能计算的优势

图形处理器（Graphic Processing Unit，GPU），最早的研制是为了分担中央处理器（Central Process Unit，CPU）的任务量，从而提高计算机处理图像数据的能力而诞生的。因为图像数据的特殊性（需要计算的数据量大、计算任务可以并行处理），使得 GPU 的设计，更加关注于数据处理的并行性，以提高数据处理速率。而这一设计初衷，在现阶段来看，对其在高性能计算领域的应用，是非常具有前瞻性的。

尽管 CPU 在高性能计算领域一直处于统治地位（例如，早期的超级计算机，包括 IBM 系列超级计算机和我国自主研制的银河系列超级计算机，都是采用的 CPU 级的众 CPU 并行运算实现的），但是，CPU 在诞生之初，其设计主要考虑的是通用性，更注重对逻辑分支等控制领域的适应性，从而不可避免的在数据计算方面具有先天的劣势。

由于 GPU 的设计早已更多地考虑了对大数据量的计算能力，所以 GPU 发展到今天，其计算能力已经远远超过 CPU，在计算密集型的高性能计算领域，GPU 所扮演的角色也将越来越重要。GPU 计算能力的提高，已经突破了摩尔定律的限制，基本上是平均每 6 个月就有计算性能翻倍的产品面世，并且相对于传统的大型计算机具有非常明显的成本的优势。这些在高性能计算方面的优势，是传统的 CPU 所无法比拟的。

1.2 CUDA 架构及其优点

CUDA（Compute Unified Device Architecture）的中文译名为“统一计算设备架构”，最早是由 NVIDIA 公司开发的一种全新的处理和管理 GPU 计算资源的硬件和软件架构。

CUDA 将 GPU 视为一个并行数据计算设备来管理，它在对数据进行处理时，不需要把这些数据计算映射为图形图像逻辑，而这种在数据处理逻辑上的简单突

破，极大地提高了 GPU 的计算通用性，也为通用 GPU 设备的研究与发展开拓了新的思路，最终使得 CUDA 架构成为当今高性能计算领域研究的热点方向。

基于 CUDA 架构的 GPU 的高性能计算具有以下优点：

（1）CUDA 架构的 GPU 一般具有更大的内存带宽。例如，NVIDIA 公司的 GeForce 9800GT 显卡，具有超过 57. 6GB/s 的内存带宽，而目前高级的 CPU 的内存带宽也只有 10GB/s 左右；

（2）CUDA 架构的 GPU 具有更多的数据处理核心。同样以 GeForce 9800GT 显卡为例，共有 112 个并行数据处理单元，能够同时完成更多的线程操作，而目前的高级 CPU 最多也只有四个核心处理单元；

（3）CUDA 架构的 GPU 相对于同样性能等级的 CPU 来讲，具有非常明显的价格优势。一般高性能的 GPU 的价格，只是同级别 CPU 价格的三分之二左右。

1.3 基于 CUDA 架构的高性能计算的国内外研究现状

出于以上章节中介绍的优点，基于 CUDA 架构的 GPU 的高性能计算，其研究发展迅速，近些年来成果不断，主要体现在以下几方面：

（1）在图形图像处理方面的应用

正如其名称的含义一样，GPU 最初的设计是为了专门用于图形图像处理，因此，将 CUDA 用于图形图像的处理加速是十分自然的结果。

近些年来，CUDA 的应用研究主要集中在光线追踪和实时渲染等方面。Zhou 等人在文献[2]中利用 CUDA 实现了快速的 kD 树建立和遍历，取得了比 CPU 快一个数量级的求解计算速度；Wang 等人则利用 kD 树结构以及自适应的间接光照收集过程，实现了交互方式的全局光照效果[3]。杨志义，朱娅婷和蒲勇在《基于统一计算设备架构技术的并行图像处理研究》一文中，对统一计算设备架构 CUDA 技术进行研究①，分析了 CUDA GPU 的显著特性，总结了 CUDA 的通用并行程序模式，详细介绍了用 CUDA 实现直方图均衡化的过程，以及 CUDA 在其它图像处理算法中的应用；最后对比 CPU 和 GPU 计算 256 级直方图均衡化的时间，实验结果表明，随着图像像素的增大，CUDA 可以把计算速度提高 40 多倍，在其它的图像算法中，甚至可以上百倍地提高速度。

（2）在数值计算方面的应用

CUDA 在数值计算方面的应用也比较普遍，主要是为了利用 GPU 强大的计算能力来加速经典数学理论问题的求解过程，以提高算法的可应用性能，其典型应

① 杨志义, 朱娅婷, 蒲勇. 基于统一计算设备架构技术的并行图像处理研究[J]. 计算机测量与控制, 2009 (4): 734-737.

用有：

1．矩阵的奇异值分解

Lahabar 等人采用 CUDA 来加速矩阵的奇异值分解速度，其计算效率比在 CPU 中实现的 LAPACK 算法提高了 8 倍[6]。李建江，张磊，李兴钢,提出一种 NVIDIA CUDA 架构下的灰度图像匹配算法①，利用 GPU 加速灰度图像的匹配过程。实际的测试结果表明，在现有实验环境中，对同一图像，在不损失匹配精度的前提下，在 GPU 上使用 CUDA 实现的灰度图像匹配并行算法比在 CPU 上使用 MPI 实现的灰度图像匹配并行算法快了 40 多倍，性能得到了显著提高，从而使灰度图像匹配应用于如交互式系统等实时应用成为可能。

2．模糊神经网络应用

Juang 等人提出了一种并行的模糊神经网络求解算法，并利用 CUDA 予以实现，他们所做的实验数据表明，其相对于利用 CPU 的求解速度，计算效率上有 30 倍的优化 [7]。在 CBIR 研究中，图像低层视觉特征和高层语义特征之间存在的“语义鸿沟”成为语义图像检索的关键问题。为了避免一般映射方法把一幅图像归于一类语义图像的现象，体现自然风景图像中包含的丰富的高层语义信息和多归属类型，石跃祥、文华、龚平等提出了对自然风景彩色图像中颜色较单一的目标区域，重复采用最优阈值化进行一次粗分割来提取最大目标区域，在分割区域的基础上，提取图像的局部颜色和形状特征，最后利用改进的模糊神经网络来建立低层视觉特征和高层语义特征之间的映射，实现了图像属性信息的有效传递和高层语义的自动获取。实验结果表明，该图像分割方法对自然彩色图像能够有效地提取目标物体，并对噪声图像具有一定的鲁棒性，而语义图像的部分类别的检索准确率接近 90％，查全率也达到了 75％，实验结果证明了该方法对自然图像检索的有效性及先进性②。

3．NP 问题的求解

NP 问题。Islam 等人提出了基于 CUDA 的高度可伸缩的 SAT 问题求解方法，求解速度相对于 CPU 处理提高了 188 倍，而且解的数量同 GPU 的内核数量成线性关系[8]。郑伯川构造了一个能量函数来表示旅行商问题，该能量函数的能量最小点对应一条有效的近似最优访问路径，然后，构造一种 LV 神经网络模型来求解该能量函数的能量最小点。实验结果表明，本文提出的 LV 神经网络模型能够收敛到能量最小点，并且与 Hopfield 网络相比，该 LV 神经网络模型具有更好的求解性能

① 李建江, 张磊, 李兴钢, 等. CUDA 架构下的灰度图像匹配并行算法[J]. 电子科技大学学报, 2012, 1: 022.
② 石跃祥, 文华, 龚平, 等. 基于模糊神经网络的语义映射方法及其在自然图像检索中的应用[J]. 计算机科学, 2013, 40(12): 122-126.

[①]。

（3）在物理学领域的应用

CUDA 高性能计算在物理学领域的应用，主要是对理论推到与工程应用问题的计算加速，包括在核物理、分子动力学、流体力学、空气动力学等领域的应用。

赵改善在《地球物理高性能计算的新选择：GPU 计算技术》[②]一文中指出，随着地球物理对高性能计算需求的不断提升，集群系统节点规模不断提高，一方面大大提高了系统建设、运行、维护、管理及应用软件开发的复杂性，另一方面在提高系统总体性能方面也受到越来越大的制约。随着微电子技术的发展，GPU 计算技术与可重构计算技术，将有可能替代集群计算技术成为高性能计算的主流技术。充分利用 GPU 并行处理能力，可以将 GPU 作为计算加速器为基于 CPU 的通用计算平台提供高性能的科学计算能力补充，这样可以在现有通用计算平台的基础上实现高性价比的高性能计算解决方案。GPU 计算平台上的应用软件开发比可重构计算平台上的应用软件开发要容易得多，这一点使得 GPU 计算技术可以更早地广泛应用于地球物理领域。GPU 计算产品已达到很高的性能，相应的软件开发环境也已推出，对于 GPU 计算平台应用软件开发技术的研究将使得 GPU 计算技术在不远的将来广泛地应用于地球物理计算中。

陈曦等人在大规模流体场景模拟过程中，提出了一种基于平滑粒子流体力学（SPH）进行流体场景模拟的算法，通过引入一种三维空间网格划分算法和改进的并行基数排序算法，实现了实时的流体计算和模拟，并能模拟出丰富的细节效果[9]；三维磁流体力学数值模拟是研究日冕和太阳风最常用的方法之一，其中日冕偏振亮度是进行观测对比的重要方法，而江雯倩等人提出的日冕偏振亮度并行计算模型，利用 CUDA 达到了近实时模拟与观测数据比对的计算要求[10]，成绩斐然。

（4）在生物医学方面的应用

随着生物医学工程领域研究的不断广泛与深入，其交叉学科的性质越来越被广泛地认可，其中众多分支学科中所涉及到的计算复杂性日益凸显，直接影响到学科的发展。因此，许多学者将基于 CUDA 的高性能计算技术引入其中，来适应这种发展方向的需求。

李拴强和冯前进[③]在研究了 CUDA 的设计思想和编程方式的基础上，对图割算法进行了并行改造，并在 CUDA 上实现了其并行化。结合肝脏肿瘤的特点，引入感兴趣区域，改进了交互方法，实现了对肝脏肿瘤的分割。实验结果表明，该方

[①] 郑伯川. 一种求解旅行商问题的 LV 回复式神经网络模型[J]. 计算机与现代化, 2013 (8): 204-208.
[②] 赵改善. 地球物理高性能计算的新选择: GPU 计算技术[J]. 勘探地球物理进展, 2007, 30(5): 399-404.
[③] 李拴强, 冯前进. 统一计算设备架构并行图割算法用于肝脏肿瘤图像分割[J]. 中国生物医学工程学报, 2010, 29(5): 641-647.

法分割结果准确，鲁棒性强，执行效率高，易于交互和扩展。Shi 等人将 CUDA 技术用于加速 DNA 序列片段的错误校正，获得了 10~19 倍的加速效果[11]。Zhou 等人采用 CUDA 技术加速交互式场补偿 MR 图像重建，获得两个数量级的加速效果，并保证了场补偿精度[12]。

（5）在通信技术领域的应用

基于 CUDA 的高性能计算在通信技术领域的应用，其本质上主要是利用 GPU 的计算能力来实现通信信号数据的编码、解码与信号分析上，其表现上是对雷达信号和卫星通信信号的高速实时处理上。

兰天，吉庆兵，于飞等人指出，CUDA 是并行计算中重要的研究与应用领域，如何将串行程序重构为并行程序以及如何将并行程序的速度最大化都成为研究的重点。前期搭建了单机单卡和单机多卡的实验环境，并在此平台上重构了一系列的密码算法。为了进一步提高破解平台的破解速度和稳定性，设计并实现了一种基于 GPU 集群（多机多卡）的暴力破解通用平台，并且在此平台上验证了 MD5 暴力破解的高速性和鲁棒性，为未来设计密码分析算法和提升算法性能提供了研究基础①。

Song 等人在文献[13]中，采用 CUDA 框架加速卫星图像的信源信道解码过程中的逆离散小波变换，该系统可以实现 1024×1024 的卫星图像 90 帧/秒的解码率。

程俊仁等人提出了一种由 GPU 完成信号搜索计算的快速实现方法，该方法以基于 FFT 的码相位并行搜索算法为基础，通过 CUDA 编程实现了信号搜索在 GPU 上的并行计算。测试结果表明，该方法的捕获速度有明显的提高，在冷启动条件下搜索全部 32 颗卫星只需 1.653 秒，从而为 GPS 软件接收机的实时运行提供了重要保证[14]。

（6）在其他领域的应用

除了以上高性能计算的常规应用领域外，基于 CUDA 的高性能计算还在众多其他领域发挥着主力军的作用，如在地震信号分析[15]、工业控制[16]、潜艇声纳信号处理[17]等领域，CUDA 同样获得了广泛的应用，并取得了不错的成绩。

从地震勘探资料中提取地震瞬时属性具有十分重要的意义，而基于信号局部特征的经验模态分解为非线性非稳定信号提供了一种全新的瞬时属性提取方法。曹晓初，金弟，王宗仁等对经验模态分解算法在 GPU 架构上的并行处理实现进行了分析和研究②。通过实验对比测试表明，GPU 架构下的算法运行效率较 CPU 具有明显优势。在测试数据中，GPU 加速比最高达到了 8．66 倍。

① 兰天, 吉庆兵, 于飞, 等. 基于 GPU 的 MD5 破解技术研究与实现[J]. 通信技术, 2013, 46(12).

② 曹晓初, 金弟, 王宗仁, 等. GPU 架构下基于经验模态分解的地震瞬时属性并行提取算法的研究[J]. 计算机科学, 2013, 40(11A): 409-411.

在工业控制方面，胡娅，黄理灿和姚晖[①]提出了在硬件产品上运行 BLAST 算法的方案，认为可以使 BLAST 达到目前为止最快的速度。在 CUDA 编程环境上执行，做了详尽的模拟试验，在一个 3 GHz 英特尔奔腾 IV 处理器上运行，比较了 BLAST 和 SSEARCH 的执行。方案与最新公布的 GPU 执行情况和一个 SIMD 解决方案进行了比较，测试表明，实现了在硬件产品上获得更大速度的目的，也降低了大规模比对的执行成本。

在声呐信号处理方面，李晓敏，侯朝焕，鄢社锋实现了一个基于 CUDA 的声呐信号处理系统[②]，提高了整个主动声纳宽带信号处理系统的实时性。实验结果表明：该系统与 CPU 平台相比，处理速度提高了近一个数量级，与具有同等处理速度的 DSP 阵列信号处理平台相比，克服了开发周期长、成本高和移植性差等缺点。

1.4 基于 CUDA 架构的高性能计算的研究重点

虽然基于 CUDA 架构的高性能计算具有上述的诸多优点，并且应用日趋广泛，技术越来越成熟，但是，不可否认的是，其依然存在着许多固有的缺点，主要体现在以下几个方面[18]-[22]：

（1）一般的 GPU 都不具备能够进行分支预测等复杂的流程控制的逻辑处理单元，因此，对于具有多分支的复杂的应用程序，其执行效率不高；

（2）CUDA 对并行化程度不高的应用程序，其运行效率下降明显，性能上一般不如 CPU 的效率高；

（3）许多产品级的 CUDA 架构的 GPU，多是针对浮点型数据的运算而开发的，并没有加入专门的整型数据运算单元，从而导致 GPU 对整型数据的运算效率较差。

由此，以上所列三点成为基于 CUDA 架构的高性能计算应用的研究重点。

1.5 本文的主要工作

针对以上基于 CUDA 架构的高性能计算在实际应用中存在的问题，从工程应用的角度出发，本文将研究工作的重点放在以下两个方面：

（1）使用自顶向下的逐渐细化的需求分析模型，对计算任务进行分解，将逻辑控制功能与数据计算功能尽可能清晰地剥离开来，以便减少控制流程分支，实现 CPU 与 GPU 的异步处理；

---

① 胡娅, 黄理灿, 姚晖. CUDA 兼容图形卡作为 BLAST 序列比对的有效硬件加速器研究[J]. 工业控制计算机, 2011 (1): 63-64.

② 李晓敏, 侯朝焕, 鄢社锋. 一种基于 GPU 的主动声纳宽带信号处理实时系统[J]. 传感技术学报, 2011, 24(9): 1279-1283.

（2）在此基础上，在实际的任务分析过程中，尽量将数据计算任务分解为可并行执行的子分支进行处理，以适应 CUDA 实现高性能计算的原理架构，将更多的可并行计算的任务交给 GPU 来执行，从而充分发挥 CUDA 的并行运算性能，以进一步提升系统的性能。

（3）除了在算法设计层面上所做的工作外，本文在编写应用系统代码的过程中，充分考虑了 CUDA 的数据类型适应性问题，将丢给 GPU 计算的数据，统一使用浮点型数据，从而充分发挥 CUDA 的计算性能。

（4）本文开发了一个集成测试平台，作为软件开发原型，利用 CUDA 实现了一个图像快速插值变换程序和一个图像快速滤波程序，用于验证本文提出的 CUDA 程序优化方案的可行性与正确性，并使用一般的 C++程序实现了相应的在 CPU 上执行的程序，用于对比不同实现方式下的执行效率，以测试本文提出的算法的有效性，并为以后应用软件的开发奠定基础。

1.6 本文研究的技术路线

本文在研究基于 CUDA 架构的快速图像处理程序的设计时，所采用的技术路线为：

首先，深入研究基于 CUDA 的程序运行时的原理及特点，在此基础上，研究 CUDA 程序的优化问题，包括 CUDA 任务分解、存储器优化与多线程同步等；

在掌握了 CUDA 程序的优化技术之后，重点研究 CUDA C 程序的应用，在本文中，拟对基于 CUDA 架构的快速图像插值程序和快速图像滤波程序的 CUDA C 实现加以研究；

最后，本文预计针对软件使用需求，搭建一个简单的集成测试平台，用以集成测试本文所提的基于 CUDA 架构的快速图像处理程序的性能，在此集成测试平台的帮助下，通过实验数据验证基于 CUDA 架构的图像插值与滤波算法的实现，在图像处理运算中性能的提高。

本文拟采用的技术路线的流程说明如下面的图 1-1 所示：

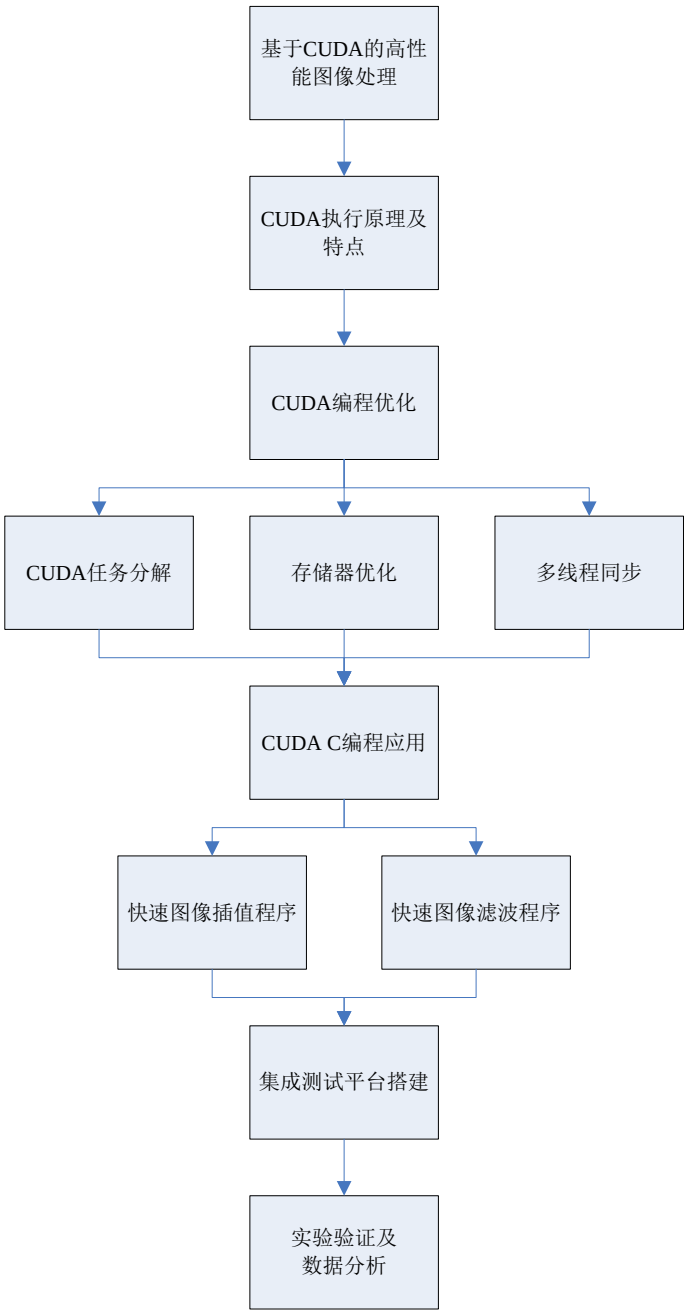

图1-1 本文课题研究采用的技术路线流程说明

1.7 论文的内容安排

本文的内容结构按照如下方式进行组织：

第一章的绪论部分介绍了高性能计算以及云计算的相关概念，并简要介绍了CUDA 架构的特点及其在高性能计算领域应用的研究进展；

在第二章中，本文较为详细地介绍了 GPU 及 CUDA 架构的关系与发展历史，从而对基于 CUDA 架构的高性能计算，从发展的角度进一步有所深入地了解；

在此基础上，本文的第三章介绍了在 CUDA 架构下编程的特点，并总结了优化 CUDA 程序的技术方案；

在具有了以上关于 CUDA 架构的基础知识以及优化编程思想的基础上，本文在第四章介绍了 CUDA C 语言的关键语法，并实现了基于标准 C 语言和 CUDA C 语言的图像处理程序，包括图像快速插值变换程序和图像快速滤波处理程序，并通过实验验证了基于 CUDA 架构的图像处理系统在计算性能上的提高；

第五章内容在以上理论研究与算法设计工作的基础之上，设计并使用.NET 平台开发了一个集成测试平台，用以测试相关软件模块的处理性能。

本文的第六章对全文进行了总结与展望。

第二章 GPU 及 CUDA 架构的发展

2.1 GPU 的发展历史

图形处理器（Graphic Processing Unit，GPU）最早是由全球著名的显卡开发与制造厂商 NVIDIA 公司于 1999 年提出的概念并开始了它的产品化进程，其设计思想与理念被许多计算机设备制造厂商所接受，从而开创了计算机图像与数据处理的崭新篇章[23] - [25]

最早的 GPU 的设计，是用来辅助计算机的中央处理器（Central Processing Unit，CPU）的工作的，其主要功能是对大量的图形图像数据进行并行处理，以减轻 CPU 的工作负担。基于此设计初衷，GPU 的开发与研制一直以来秉承着高速、并行的原则，从而使其出现至今，发展速度大大超越了摩尔定律所预测的 CPU 的发展速度。

自 GPU 出现至今，其功能与架构的发展过程大致经历了以下六个阶段：

2.1.1 1998 年后期的第一代图形处理器

这一阶段的图形处理器还处于雏形阶段，还没有 GPU 这一明确的概念，所有的产品都只是 CUP 的辅助处理芯片，帮助 CPU 减轻工作负担，其主要代表产品为 NVIDIA 公司出品的 TNT2、ATI 公司的 Rage 以及 3DFX 公司的 Voodoo 3 等。这一时期的图形处理器产品能够对图像显示变换前的三角形进行光栅化，并实现了 DirectX 的部分特征集；当执行大部分二维或三维图形处理应用时，这些处理器能够完全独立执行显示像素的更行工作，而不需要 CPU 的参与，从而大大提高了 CUP 的工作效率，其辅助 CPU 运行的功能得到了初步的体现[26] [27]。

但是，这一阶段的 GPU 的计算能力是十分有限的，还不能够完全脱离 CUP 的控制单独工作，例如，它们都缺乏三维物体图形顶点变换的能力，这些顶点变换仍然需要在 CPU 中完成；并且，这些图形处理器只能使用一个有限的数学运算集来映射图像像素光栅化后对应的颜色，效果一般。

2.1.2 1999 年至 2000 年之间的第二代图形处理器

经过第一代产品开发与应用的摸索，GPU 的计算能力有所提高，而 NVIDIA 公司也在 1999 年正式提出了 GPU 的概念。这一代产品在整体上克服了上一代产品的不足，体现在其从 CPU 处接管了顶点变换和光照映射的工作，能够完成更多之前只有 CPU 才能完成的工作，从而使得人计算机的关键有了极大地提高，至少

在显示图形图像方面已经逼近了高端工作站的性能。GPU 从此开始执行原本属于 CPU 的工作，特别是进行 3D 图形处理时[28] [29]。

但是，这一时代的 GPU 所能完成的工作仍然十分有限，其虽然能够进行更多的设置，但仍然只能完成少数预设的计算功能，实际上并不具备编程处理的能力，并且图形图像处理效果差强人意。

这一时期的商用产品主要包括 NVIDIA 公司的 GeForce 256 显卡，以及 ATI 公司的 Radeon 7500 显卡等。

2.1.3 2001 年至 2002 年之间的第三代图形处理器

2001 年 NVIDIA 公司在其 GeForce3 显卡产品中首次提出了“可编程顶点着色器单元”的概念，从而形成了具有 “初级”顶点编程能力的第三代图形处理器。这一代图形处理器能够按照应用程序指定的一系列的指令来处理图形单元的顶点，而不是仅支持 OpenGL 和 DirectX7 中指定的传统的变换和光照模型[30]。

但是，由于本代图形处理器产品在为像素着色的程序中，访问纹理的方式和格式所受限制较多，缺乏真正的像素编程能力，所以，虽然其为应用程序提供了比较丰富的设置像素值的方法，其仍不能被看作具备真正的可编程能力，只是“非可编程”与“可编程”之间的过渡产品[31] [32]。

这一时期的商用产品的主要代表有 NVIDIA 公司的 GeForce3 显卡，Microsoft 公司的 Xbox 以及 ATI 公司的 Radeon 8500 显卡等。

2.1.4 2003 年至 2005 年之间的第四代图形处理器

这一代图形处理器具备了真正意义上的可编程能力，这主要体现在：2003 年，GPU 中引入了具有里程碑意义的 Shader 技术，从而使 GPU 可以支持矩阵计算；同时，GPU 也具备了浮点数据的存储于计算功能，可以读写一般的浮点数；同时，对纹理的依赖方式变得更为灵活，可以使用索引的方式访问数据。这些顶点级和像素级的可编程能力，使得复杂的顶点变换和像素着色操作可以从 CPU 上转移到 GPU 上进行处理[33] [34]。

在 GPU 编程能力有了长足发展的同时，2003 年的 SIGGRAPH 大会首次提出了 GPGPU（General Purpose computing on GPU, 基于 GPU 的通用计算）的概念，意指利用 GPU 的计算能力在非图形处理领域进行更通用、更广泛的科学计算。GPGPU 概念的提出，为 GPU 更为广泛的应用研究开拓了思路，奠定了基础[35]。

第四代图形处理器的商用代表产品包括 NVIDIA 公司使用 Cine FX 体系结构的 GeForce FX 系列显卡以及 ATI 公司的 Radeon 9700/9800 显卡。

2.1.5 2006 年提出的第五代图形处理器

这一代图形处理器相比之前的产品，功能上更为丰富、灵活，其对顶点的处理程序可以访问纹理，并且支持程序的动态条件分支，这已经具备了 CPU 的逻辑控制功能，而在像素层级上的处理程序，不仅支持如循环、分支判断等操作，而且支持子函数的调用，运算精度上达到了 64 位的浮点精度，同时支持多目标渲染，其图像处理速度与效果都非常不错[36] [37]。

第五代图形处理器的主要代表产品为 NVIDIA 公司的 GeForce 6800/7800/7900 系列显卡。

2.1.6 2006 年之后的第六代图形处理器

第六代图形处理器中使用大量的流处理器来完成图形图像显示的加速。所谓的流处理器，是指既可以完成图形的顶点处理和像素处理，又能完成几何变换操作的 GPU 处理核心[38] [39]。

除了使用上述的流处理器作为处理核心外，这一代 GPU 还包括以下性能上的提升，包括：使用统一渲染架构以提高编程的通用性；支持 IEEE 32 位单精度浮点运算，可进行高精度的图形绘制；具有较高的内存传输带宽，内存传输能力与数据吞吐能力得到了大幅度地提高；支持绘制到纹理的功能，以加速中间结果的存储；可以将纹理作为内存来使用，减少交互[40] [41]。

第六代图形处理器以 NVIDIA 公司的 GeForce 8800 系列显卡为代表。

近年来，GPU 在性能上发展迅猛，主要体现在：其内部所有的计算核心都可以直接读写显存，从而减少了数据交互的时间；并且，GPU 内部的处理核心的数量增长飞快，其效果上要强于 CPU 的“多核”的性能，在概念上相当于“众核”处理器。但是，GPU 在本质上，其结构并没有实质性的改变，也没有官方统一的对产品换代的明确划分，所以基本上都属于性能更高的第六代图形处理器[42] [44]。

2.2 CUDA 架构的发展及其与 GPU 的关系

2.2.1 CPU 与 GPU 的区别

首先，正如本章上一小节内容所述，GPU 的设计初衷是为了分担 CPU 的负荷，辅助 CPU 进行逻辑控制与数据处理的。基于这样不同的设计初衷，导致 GPU 和 CPU 的运行机制截然不同[45] - [46]。

（1）从 CPU 的角度来看， CPU 的主要工作是执行不同的指令控制指令，基于这一目的，在设计 CPU 的运行机制时，其首要目标是对指令流的处理效能尽可

能提高。为了提升指令处理的效率，CPU 中设计了如将数据分为整型、浮点型的概念，使用分支预测等技术手段，从而减少指令分析的时间。

而对 CPU 的处理器机制衡量的一个重要标准就是在同一段时间内能够处理更多的指令，也即并行处理的概念，由此也衍生出了流水线式处理器与指令超标量执行的概念。

而后，为了更好地利用 CPU 中的数据运算执行单元，进一步提出了“乱序执行”的概念，即处理器可以打乱原有的指令执行次序，以充分利用核心资源，从而进一步提高指令处理的率。

但是，由于冯.诺依曼式计算机的线性处理架构，以及现实应用中线性处理方式的普遍性，使得计算机中输入的指令流大部分是连续的，这样的问题输入制约了并行处理单元的执行效率，因此，一味地在 CPU 中增加并行运行的处理核心，对 CPU 性能提升的贡献是不明显的，因为它们很有可能在绝大多数的时间内都处于空闲的状态。

（2）从 GPU 的角度来看，其主要的工作的设计初衷是辅助显示图形图像，这一初衷决定了它工作的主要内容是生成一组多边形，同时产生一组像素填充到该多边形当中去。因为这两种操作是相互独立，不存在相互依赖的关系，所以两方面的操作能可以并行处理，在这种模式下，更多的并行处理核心的作用就十分明显了。

其次，CPU 与 GPU 的另一个不同之处体现在数据的存储结构上。

CPU 的存储模型为外存、内存、缓存、寄存器模型，这样的存储模型，使得 CPU 在进行数据转移存储或调度时，数据的存储位置变化较大，从而消耗更多的机器时间；而 GPU 的处理核心与显示存储器之间的联系却是非常紧密，GPU 可以通过共享显存的方式来实现相邻像素点数据的交互与传递，从而使得 GPU 可以更加合理地利用显存带宽[47]。

并且，基于这样的存储模型设计，GPU 可以集成更多的处理核心，而不必担心存储问题，但是，CPU 却需要牺牲较多的硬件资源去降低数据存储问题的效率损耗。

2.2.2 基于 GPU 的通用高性能计算的优势

随着 GPU 的设计理念的不断完善与制作工艺的飞速发展，GPU 的内存带宽与浮点运算能力已大大超过了同时代的 CPU 的性能，在这样的大背景下，2003 年的 SIGGRAPH 大会提出了 GPGPU 的概念，在理论上的突破使得 GPU 逐渐从由专用的固定功能单元组成的并行处理器，向以通用计算单元为主、固定功能单元为辅

的通用型处理器架构的转变。但是，这里需要明确的概念是，这里的通用计算，并不是指使用 GPU 完成 CPU 的工作，二是指突破 GPU 的只适合图形图像显示的计算应用[48]。

相比于 CPU 的更适合对复杂的运算指令、多路的分支与判断的逻辑控制程序的处理，GPU 则更适合处理具有大数据量、高并行性、低耦合性、高计算密度的数据计算程序。因此，CPU 与 GPU 各自拥有自己的优势和擅长的领域，可以预见，未来处理器发展趋势是，CPU 和 GPU 充分协同合作，在各自擅长的领域完成相关的处理工作，并在技术上不断吸取对方的优势，以提升系统的整体性能[49]。

对于 CPU 的设计架构，如果要进一步提高运算性能，现实的作法，一是提高单个处理核心的运算性能，二是增加处理核心的数目。目前的多核 CPU、众核 CPU 以及超级计算机的设计理论，都是基于第二种方式考虑的。但是，正如本章前面部分所介绍的那样，单纯对增加逻辑控制单元的处理核心数目，对实际的应用提速来讲，意义并不是很大。相较之下，出于 GPU 的主要功能与设计架构考虑，GPU 用于通用高性能计算的优势就十分明显了。

归纳起来，GPU 的通用高性能计算上的优势主要体现在以下几个方面[50] [51]：

（1）性价比高。由于 GPU 将绝大多数的资源用在了计算单元而非控制单元上，使得 GPU 的浮点数据处理能力能够达到同级别的 CPU 的 10 倍左右，存储器带宽达到 CPU 的 5 倍左右，而价格却仅是 CPU 的 2 至 3 倍。

（2）功耗低。一般的中、低端 GPU 只需要 PCI-E 接口提供的电能就能正常工作，而具有更高处理能力的高端 GPU 也只需要通过普通计算机电源接口提供 200w 左右的电能供应即可满足需求，相比其计算能力，功能损耗已经很小了。

（3）体积小。一般体积的 GPU 芯片即可集成众多处理核心，完成通用计算需求。

基于以上的设计结构特点和优势，GPU 成为通用高性能计算应用领域研究的热点之一。

2.2.3 CUDA 架构与 GPU 的关系及优势

计算统一设备架构（Compute Unified Device Architecture, CUDA），是 NVIDIA 公司于 2007 年 6 月推出的针对其 G80 处理核心以后的 GPU 的并行计算架构，如下面的图 2-1 所示，其实质上是应用程序和 GPU 硬件资源之间的接口，是 C 语言的一个扩展集，程序的执行效率仍然取决于 GPU 的硬件资源[52]。

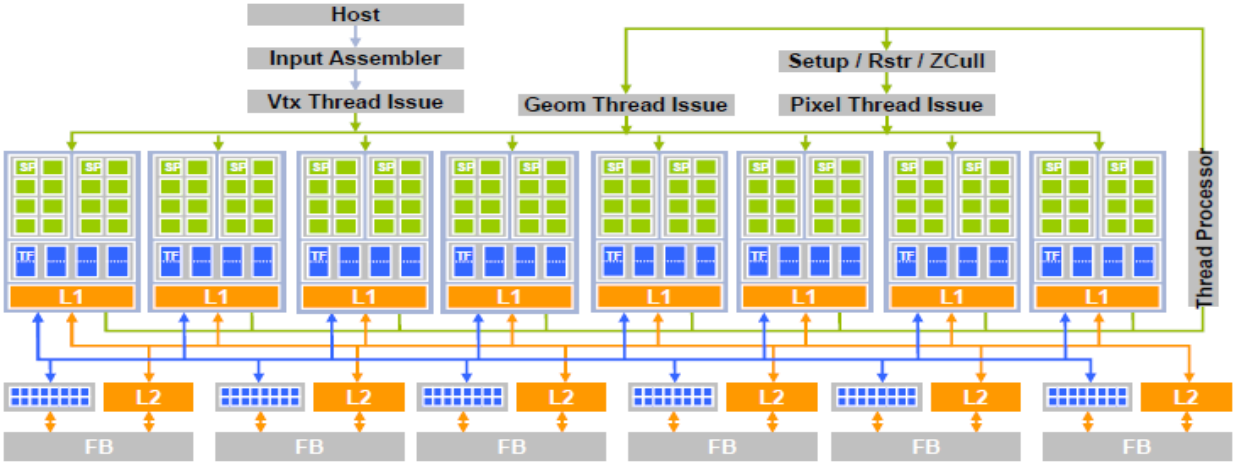

图2-1 G80 GPU的软件平台架构

但是，如下面的图 2-2 所示，CUDA 编程结构模型提供了良好的层次结构，对 C 语言具有良好的兼容性，利用 CUDA 进行 GPU 编程，不需要借助传统的图形学 API，虽然在编程思想上与 C 语言有着根本的不同，但仍然容易被熟悉 C 编程的程序员所接受，从而提高了 CUDA 的编程效率和普及程度，这是其能够在近些年迅速发展壮大的优势所在[53]。

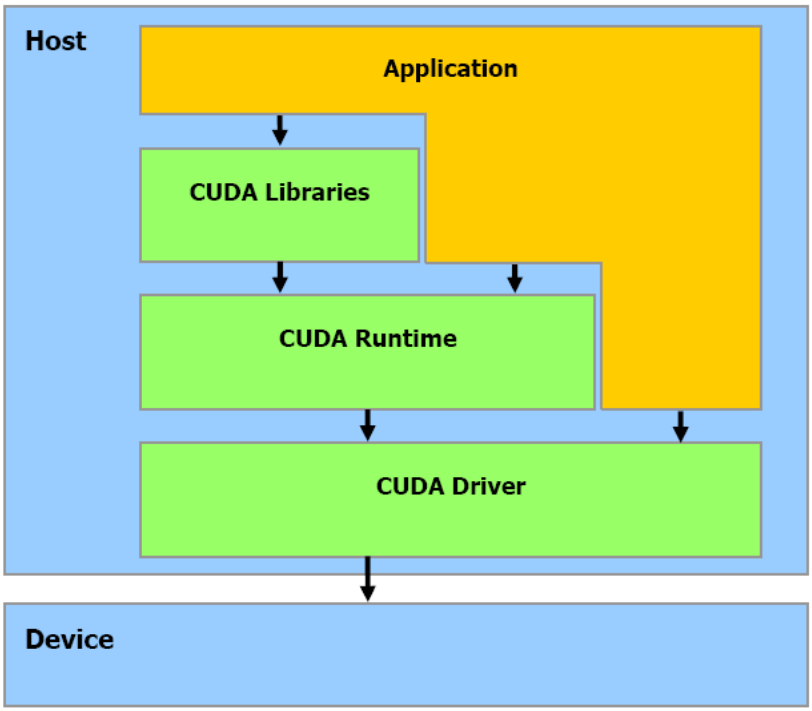

图2-2 CUDA的软件栈模型

2.3 本章小结

本章内容对 GPU 及 CUDA 架构的发展进行了研究。

首先对 GPU 的发展历史进行了简要的回顾，对 CUDA 架构的发展进行了阐述，分析了 CUDA 架构与 GPU 的关系，以及 CUDA 架构相对于传统的图像显示处理的 GPU 的优势所在。

另一方面，本章内容对 CPU 与 GPU 的架构进行了对比，分析了各自架构的特点，本文的研究认为，由于设计初衷的不同，传统的 CUP 相对于传统的 GPU，其对通用高性能计算的能力更强，但是，同时，通过对比分析，得出了基于 CUDA 架构所实现的 GPU 的通用高性能计算，相对于在传统的 CPU 上实现的高性能计算的优势所在，即性价比高、功耗低、体积小，同时，基于 CUDA 架构的通用高性能计算，性能更强。通过这样的分析结论，奠定了本文课题研究的技术先进性的基础，从而为后续的课题研究提供了保障。

第三章 CUDA 架构编程特点及优化

3.1 CUDA 概览

由于 CUDA 是 NVIDIA 公司为其生产的 GPU 产品设计的应用程序和硬件之间的接口，因此，CUDA 的架构是有其针对性和特殊的应用范畴的。

首先，从硬件资源的构成上来看，NVIDIA 的 GPU 具有以下的硬件结构层次：

（1）每个 GPU 是由若干个纹理处理器集群（Texture Processor Cluster, TPC）组成的，不同型号之间的 GPU 的 TPC 的数量不同；

（2）每个纹理处理器集群（TPC）由一个纹理存储单元和两个流式多处理器（Streaming Multiprocessor, SM）组成；

（3）每个流式多处理器（SM）包括一个用于读取、解码和发送命令的前端，和一个包括八个流处理器（Streaming Processor, SP）、两个超级函数单元的后端组成。

NVIDIA 的 GPU 的硬件结构层次如下面的图 3 - 1 所示：

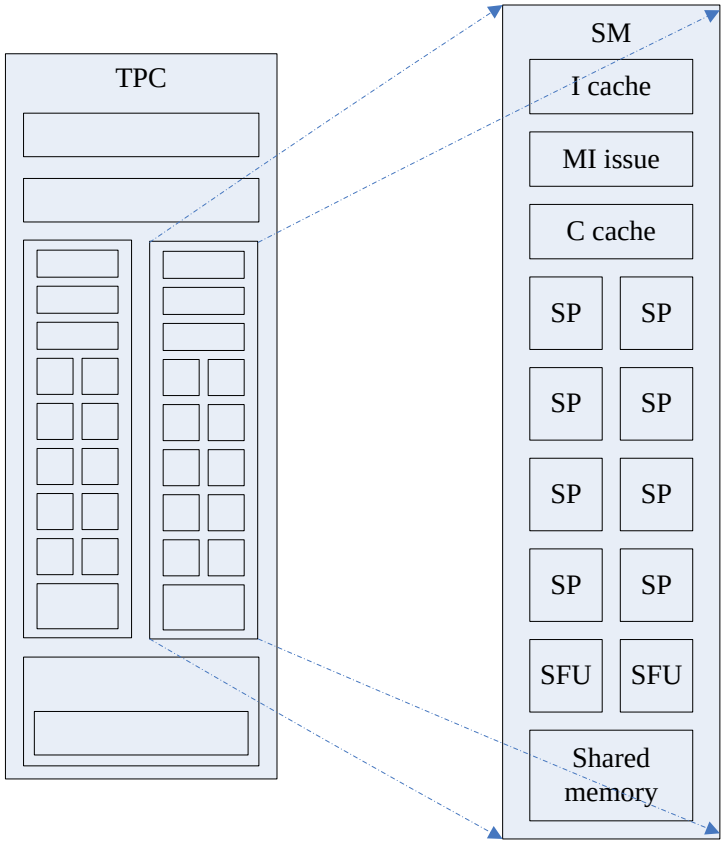

图3 - 1 NVIDIA的GPU的结构示意图

在如上的硬件架构上，CUDA 应用程序在 GPU 上运行时，采用“单指令-多线程”的执行方式，也即在一个指令周期内，可以并行执行多个线程。

其次，从软件角度来看，CUDA 的出现带来了许多应用上的优势。在 CUDA 出现之前，GPU 应用于通用计算时存在着以下缺点：

（1）要想在 GPU 上运行通用计算程序，必须把通用计算的过程转化为 GPU 图形学的概念，并且只能使用图形学专用的 API（如 Open GL 或 Direct）对 GPU 编程进行控制；

（2）以前的 GPU 应用程序，虽然可以读取显存中任意位置的数据，但不能对任意位置的数据进行写入操作；

（3）计算机系统的内存与 GPU 显存之间的数据交换的速度很慢，导致 GPU 不能充分发挥其强大的计算能力。

CUDA 作为实现 GPU 通用计算的强大的开发平台，克服了以上的缺点。其将 GPU 看作一个支持并行计算的设备，对分配到其上的计算进行统一的管理，而不再需要使用图形处理 API 来控制；在 CUDA 的管理下，GPU 应用程序可以按照自身的处理需要访问显存中的数据；并且，CUDA 的硬件架构的存储器带宽也完全能够满足显存与计算机的系统内存之间的处理数据的需要。

3.2 CUDA 编程思想及程序优化

3.2.1 CUDA 程序结构

在 CUDA 架构下，一个应用程序被分为两个部分分别执行，即设备端程序和主机端程序。

（1）设备端程序（也称为 Device 端程序），是指运行在 GPU 上的程序部分，又称为核函数（Kernel）；

（2）主机端程序（也称为 Host 端程序），是指运行在 CPU 上的程序部分。

CUDA 源程序经过 NVIDIA 公司开发的 C 编译器 NVCC 编译，一部分代码会被编译成标准的 C/C++程序，在 CPU 上执行，即成为主机端程序；另一部分代码首先转换成 PTX 文件，然后再转换成可由 GPU 执行的指令，即成为设备端程序。

一般，Host 端程序负责准备计算用的数据，并将数据复制到 GPU 的显存中，再由 GPU 执行 Device 端程序，进行实质的数据运算。计算任务完成后，Host 端程序再将结果从 GPU 的显存中取回，返回给调用程序使用。

CUDA 程序的结构和运行顺序如下面的图 3 - 2 所示：

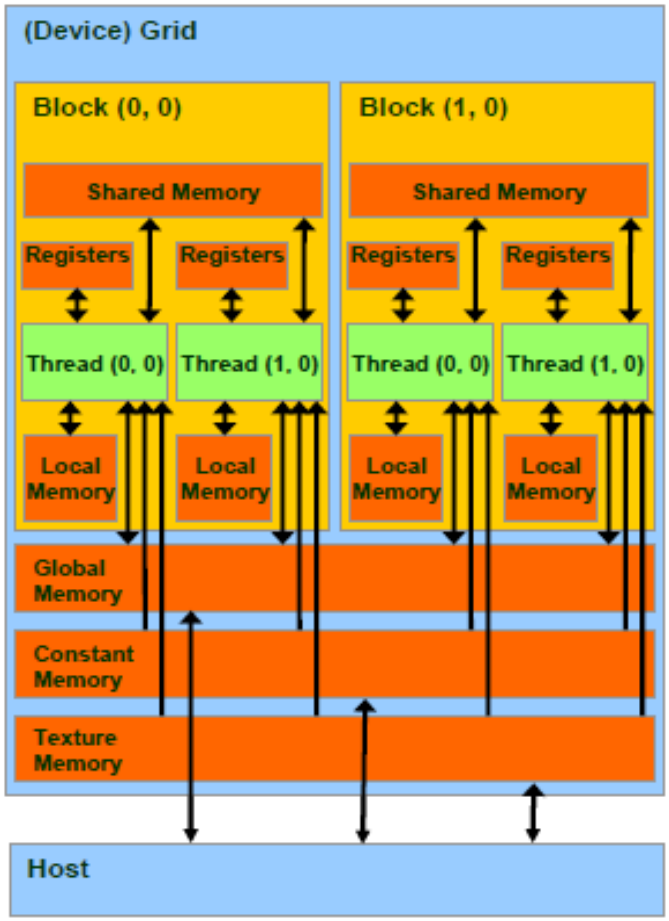

图 3 - 2 CUDA程序的结构和运行层序

3.2.2 CUDA 编程中的任务分解思想

通过 3.2.1 小节所述的 CUDA 程序的结构和执行层序可知，利用 CUDA 实现高性能计算，实质上是通过 CPU 与 GPU 的分工合作、并行运行来完成的。CUDA 充分利用了 GPU 的卓越的并行计算 （Parallel Computing） 能力，将局部计算能力强的任务分配给 GPU 执行，并使用 CPU 对这些并行计算加以控制、整合，从而提高了程序的整体运行性能。

但是，为大家所熟知的是，CPU 对不同处理任务之间的切换是十分耗费系统资源的，这样的情况在 CPU 于 GPU 之间的任务切换过程中尤为明显；另一方面，对于 GPU 的数据运算架构，也不适合执行并行性不高的程序，相反，这类程序反倒会降低 GPU 的运行效率。

因此，将 CUDA 程序执行的任务，尽可能地分解为可并行执行的部分以及必须的逻辑控制部分，并且尽量减少 CPU 与 GPU 之间的数据与控制交互，是提高 CUDA 程序执行效率的有效方案，是优化 CUDA 程序的重要方式。

3.2.3 CUDA 编程中的存储器优化思想

CPU 与 GPU 之间的数据交换的速度虽然也比较快，但是，相对于 GPU 高速的计算能力来讲，仍显缓慢。同时，数据的传输不仅发生在 CPU 系统内存和 GPU 显存之间，在 GPU 的内部同样存在着数据的传输问题。如下面的图 3-3 所示，CUDA 架构下的 GPU 的存储器具有以下的层次结构：

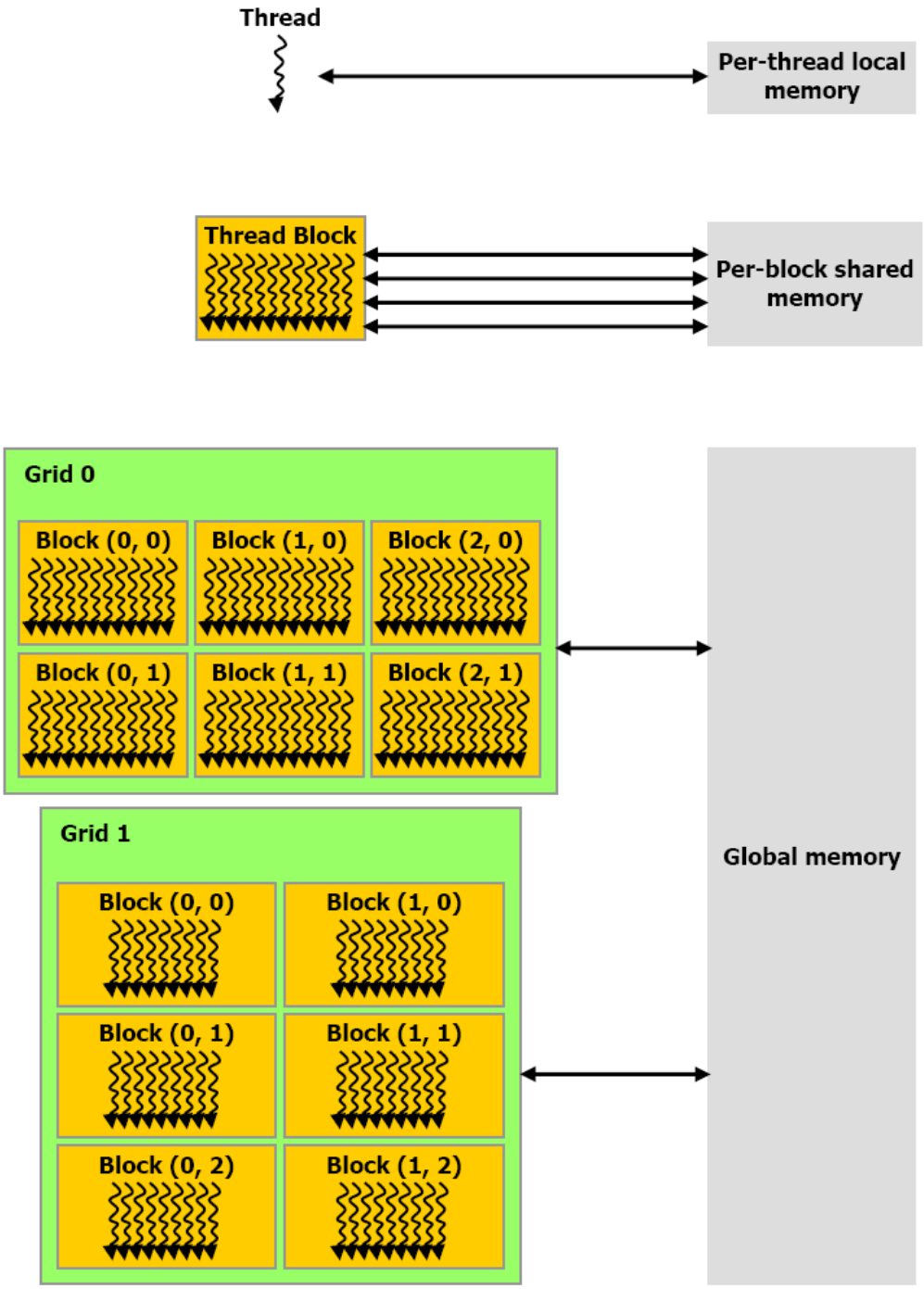

图3 - 3 CUDA的存储器层次结构

（1）每个线程拥有自己的寄存器（Register）和本地存储器（Local Memory）；

（2）同一个线程块中的所有线程共享一块共享存储器（Shared Memory）；

（3）所有线程（包括不同线程块之间的线程）可以共享一块全局存储器（Global Memory）、常量存储器（Constant Memory）以及纹理存储器（Texture Memory）；

（4）不同的网格（Grid）拥有各自的全局存储器、常量存储器以及纹理存储器。

CUDA 计算程序的执行过程如下面的图 3 - 4 所示：

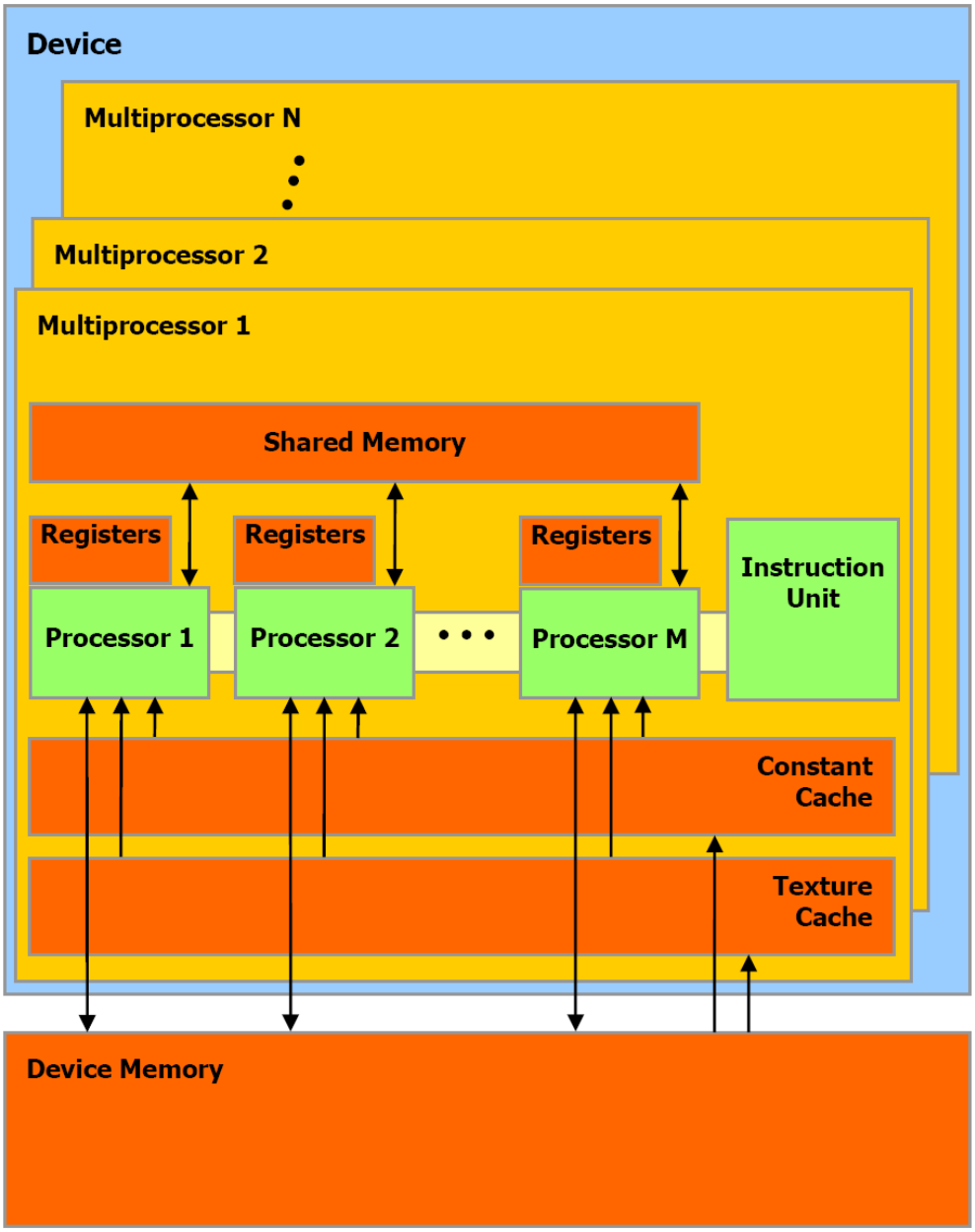

图3-4 CUDA计算程序的执行过程

如上面的示意，GPU 在执行计算任务时，首先将存储在二维数组中的数据映射到纹理内存中，然后再从纹理内存中提取数据并保存到共享内存中，以供 GPU 运算时使用。由此可见，CUDA 中的线程在执行时，可能会访问到处于不同存储器层级中的数据。因此，优化 CUDA 程序的另一条重要途径便是减少这种 CPU 系统内存与 GPU 显存之间以及 GPU 内部的各个存储单元之间的数据交互。

对于减少 CPU 系统内存与 GPU 显存之间的数据交互，即本章 3.2.2 小节提到的任务分解的思想与原则；对于减少 GPU 内部各存储器之间的交互，有效的方案是对全局存储器和共享存储器访问的优化。

对于全局存储器访问的优化，如果多个线程对全局存储器的访问满足各线程访问的起始地址对齐，并且每个线程与各自要访问的存储地址对齐这两个条件，那么，多次全局存储器访问就可以一次完成，从而提高全局存储器访问的效率。另外，还可以通过增加一次在 GPU 上执行的线程数量，从而提高数据计算密度的方式来弱化全局存储器访问的时间延迟。

而对于共享存储器访问的优化，为了提高共享存储器的访问效率，基于 CUDA 的计算程序，在算法设计时，应尽量避免计算任务对共享数据的访问，从而避免对共享存储器的访问冲突。

3.2.4 CUDA 编程中的多线程及多线程同步的思想

如前文提到的线程的概念，CUDA 中最重要的思想就是多线程处理与同步的思想。

CUDA 将整个 GPU 抽象成一个包含任意数量的多处理器的虚拟机，而每个多处理器则包含 16 个 SIMD（Single Instruction Multiple Data，单指令多数据流）形式的流处理器。在这样的抽象模型下，CUDA 将分配给 GPU 完成的计算任务，通过线程的形式在流处理器之间进行调度。在调度细节上，CUDA 将各线程通过一定的分配原则进行组合执行，所有执行同一任务的线程被分配到一个逻辑上的二维的网格（Grid）中，而只有当一个网格中的所有线程都执行完成之后，GPU 才能执行下一个网格中的线程；同一个网格中的线程又被细分为多个一维、二维或三维的线程块（Block），每个线程块中的线程被分配在同一个多处理器上运行；更进一步地，每个线程块被分解为以 32 个线程为单位的 warp（暂无统一的翻译词汇），warp 即为线程在 GPU 上执行调度的最小单位，而线程则是最基本的运行单元。

CUDA 的多线程的组织方式如下面的图 3-5 所示：

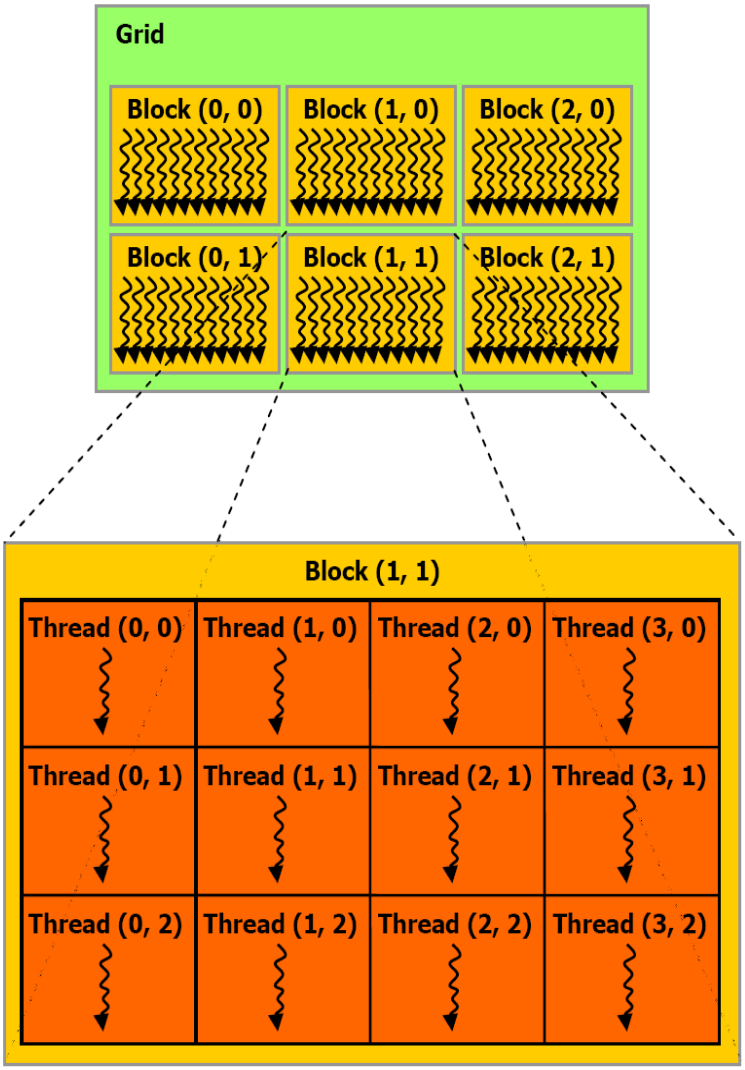

图3-5　CUDA中线程的组织方式

虽然 Grid 的执行在硬件上做了同步限制，但是，各线程之间的执行顺序是不确定的，而这种线程执行顺序的不确定性，导致 CUDA 程序的正确性受限于执行相关任务的各线程的同步，这一点在逻辑上是显而易见的。

然而，多线程的同步等待，意味着将有处理器时钟周期被闲置，因此，优化 CUDA 程序的另一个思路便是减少线程同步的频率。但是，如上所述，由于多线程运行时的不确定性，不可能对单个的线程加以限制，所以，尽可能均衡地分配各线程块之间的工作任务，是一个比较实际的减少多线程同步频率的方式。

需要再次说明的是，3.2.2 小节中的任务分解思想，是将通用计算任务分解为控制任务和数据运算任务，而本小节中提到的合理的任务分配思想，是指将通用计算任务均衡地分配出去，二者针对的任务分解阶段是不同的。

3.3 本章小结

本章内容对 CUDA 架构编程的特点及其优化问题进行了研究。

首先，本章对 CUDA 架构进行了总体的分析，认为 CUDA 架构作为实现 GPU 通用计算的强大的开发平台，克服了传统的使用 GPU 实现通用高性能计算时的弊端和缺点。其使用 GPU 对分配到其上的计算进行统一的管理的思想，能够使 G PU 应用程序可以按照自身的处理需要访问显存中的数据，这在一定程度上提高了 GPU 不间断处理通用计算程序的能力。

其次，本章对于基于 CUDA 架构的通用高性能计算程序的优化方法的研究认为，要实现在 CUDA 架构下的通用高性能计算程序的优化，应在编写程序的过程当中，着重做好以下的工作：

（1）由于在 CUDA 架构下的应用程序被分为设备端程序和主机端程序，而 CPU 对不同处理任务之间的切换是十分耗费系统资源的，因此，需要将 CUDA 程序执行的任务，尽可能地分解为可并行执行的部分以及必须的逻辑控制部分，并且尽量减少 CPU 与 GPU 之间的数据与控制交互。

（2）对于减少 CPU 系统内存与 GPU 显存之间的数据交互应使用任务分解的思想与原则；对于减少 GPU 内部各存储器之间的交互，有效的方案是对全局存储器和共享存储器访问的优化。

（3）对于全局存储器访问的优化，应尽量满足多个线程对全局存储器的访问的起始地址对齐，并且每个线程与各自要访问的存储地址对齐；其次，还可以通过增加一次在 GPU 上执行的线程数量，从而提高数据计算密度的方式来弱化全局存储器访问的时间延迟。

（4）对共享存储器访问的优化，应尽量避免计算任务对共享数据的访问，从而避免对共享存储器的访问冲突。

（5）优化 CUDA 程序的另一个思路便是减少线程同步的频率。

本章内容从 CUDA 程序结构、CUDA 编程中的任务分解思想、存储器优化思想、多线程处理及多线程同步的思想的角度，来分析和总结的 CUDA 架构程序的优化问题的解决方案，从技术的角度，奠定了本文课题研究的技术基础。

第四章 基于 CUDA 架构的快速图像处理系统

4.1 CUDA 编程的软件体系与关键语法

4.1.1 CUDA 编程的软件体系

NVIDIA 公司为 CUDA 应用程序开发人员提供了一整套程序开发与调试环境组件，包括 NVCC 编译器、GDB 调试器、CUDA 库函数、CUDA 运行时库（CUDA Runtime API）以及 CUDA 驱动程序（CUDA Driver API）。

在上述的 CUDA 编程的软件体系中，其核心部分是 CUDA C 语言，它实质上是标准 C 语言的一个极小扩展集和一个运行时库，源文件可以通过 NVCC 编译器编译得到调用这些扩展和运行时库的目标程序；使用 NVCC 编译器得到的目标程序只是在 GPU 设备上运行的代码，而要管理 GPU 的计算资源，就需要借助 CUDA 运行时库和 CUDA 驱动程序来实现。

4.1.2 CUDA 编程的关键语法

虽然 CUDA C 语言是标准 C 语言的一个极小扩展集，但同样具有一些自己独特的语法结构和编程准则。

在 CUDA 应用程序开发的过程中，一个十分重要的概念便是核函数（kernel）的概念。核函数是直接指定线程需要完成的计算任务，也即一组指令集，其中制定的计算功能由线程在 GPU 上实现。

CUDA C 在实现核函数时，针对 GPU 的特性做了以下的扩展：

（1）引入了函数调用类型限定符。共有三种函数调用类型限定符，用来规定该函数是由 GPU 设备调用还是由主机 CPU 调用。这三种限定符以双下划线开头，并以双下划线结尾，具体为：

__device__：指定该函数只能被 GPU 设备调用；

__host__：指定该函数只能被 CPU 调用；

__global__：指定该函数既可以被 GPU 设备调用，也可以被 CPU 调用。

（2） 引入了变量类型限定符。共有三种变量类型限定符，用来规定变量的存储位置。这三种限定符同样以双下划线开头，并以双下划线结尾，具体为：

__device__：指定数据存储在 GPU 的显存中；

__constant__：指定数据存储在 GPU 的常量存储器中；

__shared__：指定数据存储在 GPU 的共享内存中。

（3）引入了多种内置的矢量类型。如 double2、dim3、char4 等，它们是由基本的整型或浮点型数据构成的组合型的向量类型，具有特定的维数，CUDA 中规定通过 x、y、z、w 这四个固定符号来访问每一维分量。

（4）引入了四个内建的索引变量。CUDA 中使用固定的四个符号的组合来索引要操作的线程，其由外到内依次为：

gridDim：用来描述网格的维度；

blockDim：用来描述线程块的维度；

blockIdx：用来索引线程块；

threadIdx：用来索引具体的线程。

（5）引入了“<<<>>>”（三组嵌套的尖括号）运算符，用来指定网格和线程块的维度，并传递执行参数（具体用例见后文的实验部分）。

（6）其他辅助函数。为了保证 CUDA 程序能够快速、正确地完成，在 CUDA C 语言中还引入了其他一些辅助功能的函数，如内置数学函数、纹理函数、原子函数、同步函数等。

4.2 图像处理需求的特点及几种主要需求

我们在实际应用中，对图像处理的需求多集中在对图像视觉效果的提升上，这方面的需求在现阶段主流的软件应用产品上体现得十分明显，如看图软件、图像修改软件、图像美化软件等。

以上的图像应用软件，其实际的实现技术，无外乎图像的显示变换（图像的翻转、旋转，放大显示等）与图像滤波（图像的噪点去除、光滑处理等）两大技术范畴。

因此，研究基于 CUDA 架构的快速图像显示变换与滤波处理系统的实现，对实际的应用性需求具有十分重要的现实意义。本章的后续部分就专注于使用 CUDA C 来实现这两类图像处理系统，具体地实现了一个快速图像插值程序与一个快速图像滤波程序。

4.3 快速图像插值程序

4.3.1 任务概述

在进行图像的显示时，很多情况下需要对图像进行放大显示，当图像的像素数量不足以满足放大显示的需求时，也即图像信息量不够时，就需要对缺失的图像像素，利用存在的像素值进行估计，尽量满足放大显示图像的要求。

4.3.2 图像插值的原理及常用算法

所谓图像的插值，也即是通过已有的图像像素值，来估计图像被放大显示后，所缺少的像素所具有的像素值。

常用的插值算法包括最近邻插值、双线性插值、三次样条插值算法等。这三种典型的插值算法，算法复杂度依次提高，但算法的插值效果也越来越好，一般，双线性插值的时间复杂度和插值效果可以达到普遍的要求。本文据此，将利用双线性插值法来验证 CUDA 的性能。

双线性插值算法的原理是，首先，根据图像的放大比例，来计算放大后的图像各像素值在原图上的坐标（浮点数坐标），再根据此浮点型坐标位置临近的四个原图上的整数型坐标位置处的像素值，分别计算权重来计算此位置处的最终像素灰度值，如下面的图 4-1 所示。

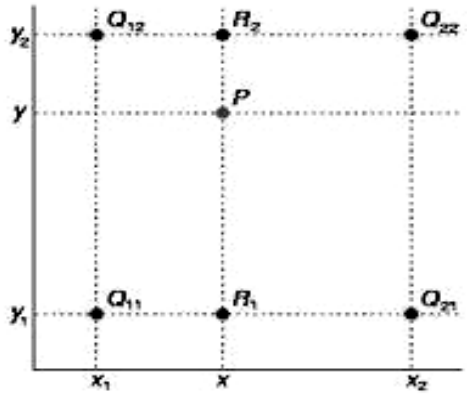

图4-1 双线性插值原理图

4.3.3 算法实现

在此，使用 C++语言和 CUDA 分别实现图像的双线性插值算法，其关键代码如下所示：

（1）C++实现的双线性插值算法的关键代码

```
int BiLinearInsert (int* pImageData, int nImageWidth, int nImageHeight,
                    int* pDestImage, int nDestWidth, int nDestHeight)
{   pDestImage = new unsigned int(nDestWidth*nDestHeight);
    if (!pDestImage) return -1;
    float fWidthGradient = (float)(nImageWidth-1) / (nDestWidth -1);
    float fHeightGradient = (float)(nImageHeight -1) / (nDestHeight -1);
    int x_dest, y_dest;
    float x_center, y_center;
```

```
float wx0, wx1, wy0, wy1;
int x_org_left, x_org_right;
int y_org_ceil, y_org_floor;
for(y_dest =0; y_dest < nDestHeight; y_dest ++)
{
    for(x_dest =0; x_dest < nDestWidth; x_dest ++)
    {
        x_center = x_dest* fWidthGradient;
        y_center = y_dest* fHeightGradient;
        x_org_left = (int) x_center;
        x_org_right = 1+ x_org_left;
        y_org_ceil = (int) y_center;
        y_org_floor = 1+ y_org_ceil;
        wx0 = x_center - x_org_left
        wx1 = 1f – wx0;
        wy0 = y_center - y_org_ceil
        wy1 = 1f – wy0;
        pDestImage [(y_dest-1)* nDestWidth + x_dest] =
            (int)(pImageData[(y_org_ceil -1)* nImageWidth +
        x_org_left]*wx0*wy0) +
            (int)(pImageData[(y_org_ceil -1)* nImageWidth +
        x_org_right]*wx1*wy0) +
            (int)(pImageData[(y_org_ floor -1)* nImageWidth +
        y_org_ceil]*wx0*wy1) +
            (int)(pImageData[(y_org_ floor -1)* nImageWidth +
        y_org_floor]*wx1*wy1);
    }
}
return 0;}
```

在上面的代码中，x_dest 和 y_dest 为插值后的图像像素的索引；fWidthGradient 和 fHeightGradient 为计算得到图像宽和高的插值比例；x_center 和 y_center 为插值点对应于原图像中的浮点型坐标；而 x_org_left、x_org_right、y_org_ceil、y_org_floor 分别对应插值点四周的四个像素的索引； wx0、wx1、wy0、wy1 则为四个点在插

值计算时的权重。

（2）使用 CUDA 实现图像双线性插值的核函数

首先，使用 CUDA 进行图像处理时，需要预定义纹理寄存器，用于作为 CUDA 运算时的工作区，2 维纹理寄存器的定义如下：

texture <float, 2 , cudaReadModeElementType> ImageTex;

其中，texture 关键字用于声明定义纹理寄存器，尖括号中的第一个参数为寄存器存储数据的类型（在这里声明为浮点型），第二个参数确定纹理寄存器的维数（在这里声明为二维的纹理寄存器），第三个参数确定纹理寄存器的读取方式（在这里声明为只读类型）。

在上述的图像数据纹理寄存器上运行的实现图像双线性插值的核函数如下：

```
__global__ void
BiLinearInsert _kernel ( float * pImageData, int nImageWidth, int nImageHeight,
                         float a0, float a1, float a2, float a3)
{
int x=blockDim.x*blockIdx.x+threadIdx.x;
int y=blockDim.y*blockIdx.y+threadIdx.y;
int offset_x;
int offset_y;
float m , n;
if (x<w&&y<h)
{
    offset_x = (x&1)>0?1:-1;
    offset_y = (y&1)>0?1:-1;
    m=float(x>>1);
    n=float(y>>1);
    float Pixel00=tex2D(ImageTex, m, n);
    float Pixel01=tex2D(ImageTex, m+offset_x , n);
    float Pixel10=tex2D(ImageTex, m, n+offset_y);
    float Pixel11=tex2D(ImageTex, m+offset_x , n+offset_y);
    pImageData [y*w+x] = a0* Pixel00 + a1 * Pixel01 + a2 * Pixel10 + a3 * Pixel11;
}
}
```

其中，x、y 为像素点在纹理寄存器中的索引；offset_x、offset_y 为像素点的

偏移量；m、n 为插值点在纹理寄存器中的索引，使用位运算（右位移>>和左位移<<）能够进一步提高运算速度；a0~a3 为根据放大倍数算得的插值点附近各点灰度的相应权值，这些值在函数调用时确定，能够减少这种非并行性运算对 CUDA 性能的影响。

4.3.4 实验数据

如下面的图 4-2 所示，为原始的 256×256 的 Lena 图（灰度图像，灰度区间为 0~255）；图 4-3 插值为 515×512 的 Lena 图和原始图像的组合对比图。

分别使用上述两类方式实现的图像双线性插值处理的效果相同，只是运行时间有所差异，得到的实验图均如下面的图 4-3 所示。

图4-2 256×256，灰度区间0~255的原始Lena图

图4-3 插值为512×512的Lena图和原始图像

4.3.5 数据分析与对比

使用如下的 C++语言记录时间，在每次执行 C++程序之前和之后，分别记录当前时刻，以及在调用 CUDA 核函数之前和之后时间的数据如下所示（分别执行 10 次运算），计算机硬件为 Intel i3 处理器，4G DDR3 内存，NVIDIA GFORCE 9800 显卡。

```
#include <time.h>
#include<stdio.h>
FILE *pFile;
time_t t;
pFile = fopen（"C:\\timelog.txt ","a+"）;
fprintf（pFile,"Function Name: %s\n", ctime（&t））;
fclose（pFile）;
```

其中，Function Name 为具体调用的 C++函数或者 CUDA 函数的名称，后面显示执行时的时刻，开始的时刻与结束的时刻做差，即为函数的执行时间。

本次实验的实验结果数据如下面的表 4 - 1 和表 4 - 2 所示。其中，表 4 - 1 为 C++函数执行的数据，表 4 - 2 为 CUDA 函数执行的数据。

表4-1 图像差值运算执行时间表——C++函数执行数据

CPP	Begin time	Duration（ms）
1	14:52:37.343	--
2	14:52:37.396	53
3	14:52:37.451	55
4	14:52:37.501	59
5	14:52:37.551	50
6	14:52:37.601	50
7	14:52:37.653	52
8	14:52:37.705	55
9	14:52:37.758	53
10	14:52:37.809	51
11	14:52:37.864	55

下面的表 4 - 2 为 CUDA 函数执行的数据：

表4 - 2 图像差值运算执行时间表——CUDA函数执行数据

CUDA	Begin time	Duration（ms）
1	13:07:32.399	--
2	13:07:32.415	16
3	13:07:32.430	15
4	13:07:32.446	16
5	13:07:32.462	16
6	13:07:32.478	15
7	13:07:32.493	15
8	13:07:32.508	16
9	13:07:32.524	17
10	13:07:32.541	15
11	13:07:32.566	15

由上面的表 4 - 1 和表 4 - 2 的函数执行的数据可见，执行同样的图像双线性差值算法进行图像差值处理，使用 CUDA 的执行时间，在上述实验条件下，相比于使用 CPU 执行所耗费的时间，性能提高在 70%左右，显示了 CUDA 的较高的执行能力。

4.4 快速图像滤波程序设计

4.4.1 任务概述

图像数据在获得的过程中，很容易受到噪声的污染，例如，数码相机在拍照的过程中，容易受到不稳定电流的影响；数字 X 射线成像设备则对电磁辐射的波动非常敏感，从而生成含噪图像。这些含噪图像中的噪点会影响人们的视觉效果，在某些场合，甚至会导致极其严重的后果（例如在数字影像用于临床诊断的过程中）。

对于这种不能准确知道噪声在图像中出现位置和强度的图像降噪问题，只能通过图像的滤波来处理。

4.4.2 图像滤波原理及算法

所谓图像滤波，根据实施的区域不同，可以分为空间域和频域滤波两大类。虽然有如此的划分，但其本质上的原理都是相同的，即通过某个像素周边的其他相关像素的值，来确定该像素的终值。由此可见，对图像的滤波，其计算量是非常大的，根据图像滤波时所选择的邻域的大小，其运算量会达到图像原始尺寸的 9 倍以上（因为通常最小的滤波是在 3×3 的邻域上进行的）。

常用的图像滤波算法有高斯滤波、拉普拉斯滤波、均值滤波等，其所使用的公式为：

设灰度图像 Image 的大小为 m×n，其中每个像素的像素值为 $x_{i,j}$，其中，i∈[0, m-1]，j∈[0, n-1]分别为像素的横纵坐标，则对图像滤波的定义为：

$$\begin{aligned} x_{i,j} = &\frac{1}{a_{i-1,j-1}} x_{i-1,j-1} + \frac{1}{a_{i,j-1}} x_{i,j-1} + \frac{1}{a_{i+1,j-1}} x_{i+1,j-1} \\ &+ \frac{1}{a_{i-1,j}} x_{i-1,j} + \frac{1}{a_{i+1,j}} x_{i+1,j} \\ &+ \frac{1}{a_{i-1,j+1}} x_{i-1,j+1} + \frac{1}{a_{i,j+1}} x_{i,j+1} + \frac{1}{a_{i+1,j+1}} x_{i+1,j+1} \end{aligned} \tag{1}$$

其中，$a_{i,j}$ 为滤波系数。根据 $a_{i,j}$ 系数的不同，也即形成了不同的滤波算法。由此可见，图像滤波算法可以抽象出如下公式：

$$x_{i,j} = \sum_{i\in[1,m],\, j\in[1,n]} k_{ij} x_{ij} \tag{2}$$

对于图像边缘上的点，通常做拓扑来解决边缘点的邻域像素值不全的问题。常用的图像拓扑算法包括复制法、镜像法和填 0 法等。拓扑区域的大小根据所使用的滤波器邻域大小来确定。

4.4.3 算法实现

在此，使用 C++语言和 CUDA 分别实现图像滤波算法，其关键代码如下所示：

（1）C++实现图像滤波的关键代码

实现图像滤波的 C++函数如下，算法中假设已经做了相应的拓扑。

```
int ImageFilter（int* pImageData, int nImageWidth, int nImageHeight,
                    int* pMaskCoeficient, int nMaskLength, int* pDestImageData）
{
    int nOrgImageWidth=nImageWidth-nMaskLength+1;
    int nOrgImageHeight=nImageHeight-nMaskLength+1;
```

```
    pDestImageData=new unsigned int（nOrgImageWidth*nOrgImageHeight）;
    if （!pDestImageData） return -1;
    int i,j,k;
    for（i=0;i<nOrgImageHeight;i++）
    {
        for（j=0;j<nOrgImageWidth;j++）
        {
            pDestImageData[i*nOrgImageWidth+j]=0;
            for（k=0;k<nMaskLength*nMaskLength;k++）
            {
                pDestImageData[i*nOrgImageWidth+j]+= pMaskCoeficient[k]*
                                pImageData[（i+nMaskLength/2）* nImageWidth+k];
            }
        }
    }
    return 0;
}
```

在上面的代码中，最内层循环的 pImageData 的下标的全部计算公式为：pImageData[（i+nMaskLength/2）* nImageWidth+nMaskLength/2-nMaskLength/2+k]一加一减，表示数据的偏移。

pImageData 为输入的需要滤波的图像数据的指针；nImageWidth 和 nImage Height 标明了输入图像数据的尺寸；pMaskCoeficient 为滤波系数的数组；nMask Length 为正方形滤波器的边长；pDestImageData 为输出的滤波后的图像数据的指针。

（2）使用 CUDA 实现图像滤波的核函数

类似 4.3.3 小节中第(2)部分内容所述，预定义了 2 维的纹理寄存器 ImageTex，则在此图像数据纹理寄存器上运行的实现图像 3×3 邻域的高斯滤波的 CUDA 核函数如下：

```
__global__ void
Gauss3X3Filter _kernel（float* pImageData, int nImageWidth, int nImageHeight）
{
    float * pDestImageData = （float *）（((float *） pImageData） + blockIdx.x）;
    for （ int i = threadIdx.x; i < nImageWidth; i += blockDim.x ）
```

```
    {
        float Pixel00 = tex2D（ImageTex, （float） i-1, （float） blockIdx.x-1 ）;
        float Pixel01 = tex2D（ImageTex, （float） i, （float） blockIdx.x-1 ）;
        float Pixel02 = tex2D（ImageTex, （float） i+1, （float） blockIdx.x-1 ）;
        float Pixel10 = tex2D（ImageTex, （float） i-1, （float） blockIdx.x+0 ）;
        float Pixel11 = tex2D（ImageTex, （float） i, （float） blockIdx.x+0 ）;
        float Pixel12 = tex2D（ImageTex, （float） i+1, （float） blockIdx.x+0 ）;
        float Pixel20 = tex2D（ImageTex, （float） i-1, （float） blockIdx.x+1 ）;
        float Pixel21 = tex2D（ImageTex, （float） i, （float） blockIdx.x+1 ）;
        float Pixel22 = tex2D（ImageTex, （float） i+1, （float） blockIdx.x+1 ）;
        pDestImageData [i] = 0.0278 * Pixel00 + 0.1111* Pixel01 + 0.0278 * Pixel02
                             + 0.1111* Pixel10 + 0.4444 * Pixel11 + 0.1111*
Pixel12
                             + 0.0278 * Pixel20 + 0.1111* Pixel21 + 0.0278 *
Pixel22;
    }
}
```

其中，pImageData 为输入的要被滤波的图像数据的指针；pDestImageData 为一个中间变量，记录了每个线程计算时需要使用的数据的地址；最后的结果仍然通过 pImageData 返回。

需要说明的是，由于 GPU 的特殊性，在这里只有针对性地实现了对图像 3×3 邻域的高斯滤波，而没有编写如本小节第（1）部分所述的 C++程序的通用型图像滤波函数，是为了减少 GPU 在进行通用高性能计算时的逻辑分支，从而提高系统的整体运算性能，而其他种类的滤波器系数以及滤波函数的原理与编程方法与此类似。

4.4.4 实验数据

下面以数字图像处理的去噪试验过程中经常使用的电路板 X 射线图像为例，验证 CUDA 对图像滤波处理的效率。

如图 4 - 4 电路板 X 光原图所示，为 464×448 的电路板 X 射线图（灰度图像，灰度区间为 0~255）；图 4 - 5 为加入强度为 3 的高斯噪声后的电路板图像加入强度为 10 分贝的高斯白噪声后的图像；图 4 - 6 为使用 3×3 邻域的高斯滤波器进行滤波处理后的图像，在滤波时，图像的边缘使用复制的拓扑方式。

下面的图 4 - 4 为电路板 X 光原图，从图中可见，原始图像的效果较为清晰，线条明确，噪声较少。

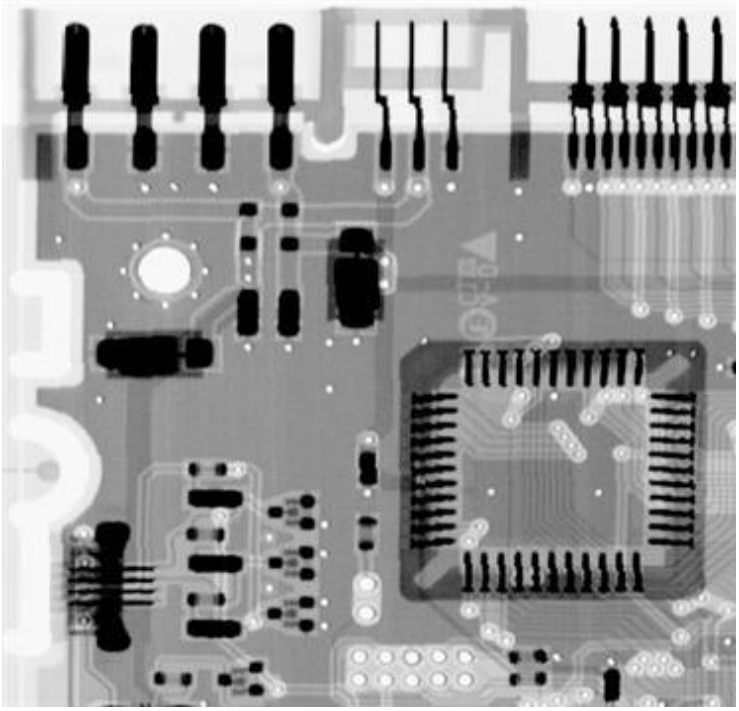

图4-4 电路板X光原图

下面的图 4 - 5 为上述的图 4 - 4 所示的实验图像，加入强度为 3 的高斯噪声后的效果图，与图 4 - 4 对比，图 4 - 5 已经显得较为模糊，噪声偏大。

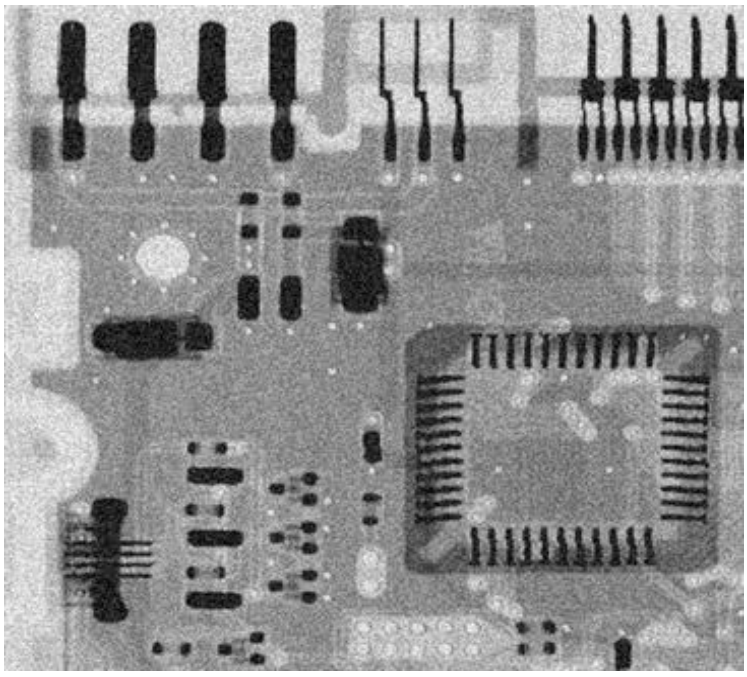

图4-5 加入强度为3的高斯噪声后的电路板图像

下面的图 4 - 6 为上述的图 4 - 5 所示的实验图像，使用 3×3 的高斯滤波器进行滤波降噪处理之后的图像。与图 4 - 4 和图 4 - 5 进行对比，图 4 - 6 的噪声程度已经大大降低，但是相比于原始图像，还是较为模糊。

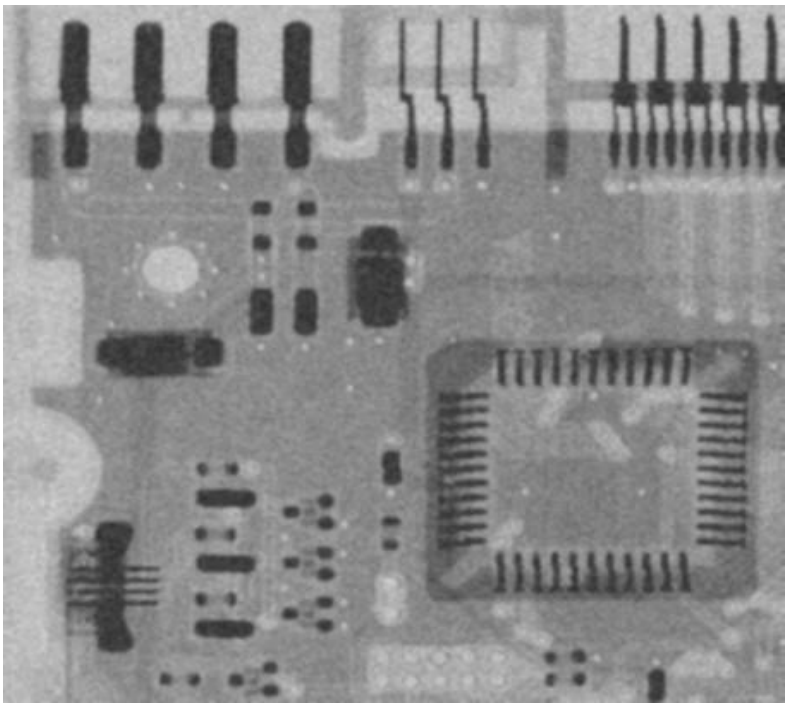

图4-6 经过3×3高斯滤波后的图像

4.4.5 数据分析与对比

在 4.3.5 所述的实验环境下和时间记录的方法下，分别执行 10 次运算的实验数据如分别如下面的表 4 - 3 和表 4 - 4 所示，其中，表 4 - 3 为 C++函数执行的数据，表 4 - 4 为 CUDA 函数执行的数据。

表4 - 3 图像高斯滤波处理执行时间表——C++函数执行数据

CPP	Begin time	Duration（ms）
1	10:53:29.785	--
2	10:53:30.010	225
3	10:53:30.226	216
4	10:53:30.446	220
5	10:53:30.678	232
6	10:53:30.887	219

续表4 - 3 图像高斯滤波处理执行时间表——C++函数执行数据

7	10:53:31.109	222
8	10:53:31.341	232
9	10:53:31.576	235
10	10:53:31.806	230
11	10:53:32.034	228

下面的表 4 - 4 所示为 CUDA 函数执行的数据：

表4 - 4 图像高斯滤波处理执行时间表——CUDA函数执行数据

CUDA	Begin time	Duration（ms）
1	09:32:11.086	--
2	09:32:11.118	32
3	09:32:11.146	28
4	09:32:11.175	29
5	09:32:11.206	31
6	09:32:11.234	28
7	09:32:11.263	29
8	09:32:11.295	32
9	09:32:11.326	31
10	09:32:11.357	31
11	09:32:11.385	28

由上面的表 4 - 2 和表 4 - 3 的数据可见，执行同样的图像高斯滤波运算处理，使用 CUDA 的执行时间，在上述实验条件下，相比于使用 CPU 执行所耗费的时间，性能提高在 85%左右，显示了 CUDA 的较高的执行能力。

4.5 本章小结

本章内容是本文课题研究的核心内容，对基于 CUDA 架构的快速图像处理系统的实现问题进行了研究。

首先，本章对 CUDA 编程的软件体系与关键语法进行了介绍；其次，本章内容分析了图像处理的需求的特点，并介绍了几种主要的图像处理的需求；在上述工作的基础之上，本章使用传统的 C 语言和 CUDA C 语言，实现了快速的图像插值程序和快速的图像滤波程序，介绍了这两个程序的处理任务、算法实现，并进行了相应的实验，获得了实验数据，对两种方式实现的程序的实验数据进行了对比分析。

本章的研究结果表明，使用 CUDA 实现的图像处理，比使用普通的 CUP 所实现的图像处理，其性能得到了较大幅度的提升。执行同样的图像双线性差值算法进行图像差值处理，使用 CUDA 的执行时间，在本文的实验条件下，相比于使用 CPU 执行所耗费的时间，性能提高在 70%左右；而执行同样的图像高斯滤波运算处理，相比于使用 CPU 执行所耗费的时间，性能提高在 85%左右，显示了 CUDA 的较高的执行能力。

第五章 集成测试平台的设计与实现

5.1 集成测试平台的开发与测试环境

本文课题研究实现的用以测试相关软件模块的处理性能的集成测试平台，使用 Microsoft Visual Studio 2010 集成开发环境，基于.NET Framework 4.0 框架进行开发。平台界面开发使用 C#语言进行，其中的图像处理程序，分别使用 C++语言和 CUDA C 语言进行开发，用以对比测试 CUDA 架构的性能提升。

集成测试平台运行的主要的计算机硬件环境为：CPU 使用 Intel i3 双核处理器，系统内存为容量 4 G 的 DDR 3 内存，显卡即 CUDA 运行硬件，使用 NVIDIA 公司的 GFORCE 9800 显卡。

5.2 集成测试平台的界面原型效果

本文课题研究设计开发的集成测试平台的界面原型如下面的图 5-1 所示：

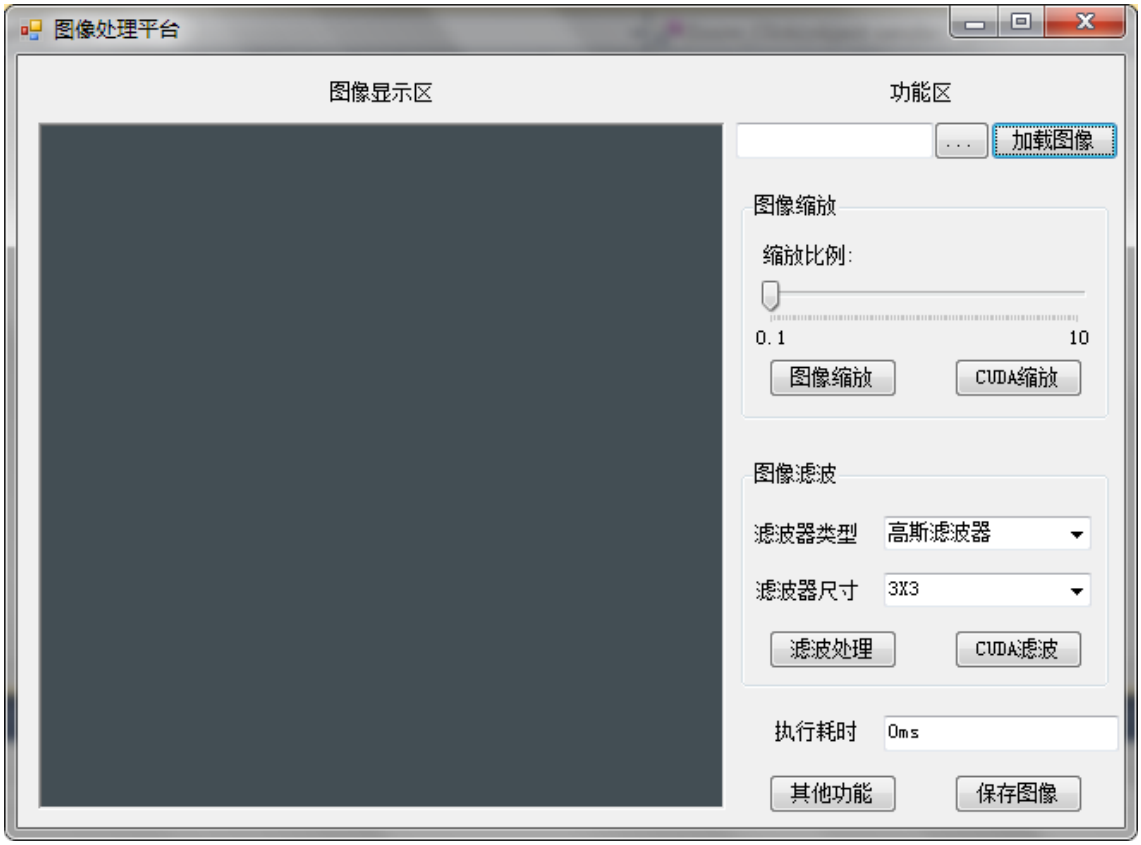

图5-1 集成测试平台的界面原型

5.3 集成测试平台的功能实现

5.3.1 引用的命名空间

本文课题研究开发的集成测试平台，以及用以实现图像处理功能并显示图像处理结果的图像显示控件，是基于以下命名空间提供的类库实现的：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Linq;
using System.Text;
using System.Windows.Forms;
```

图像显示控件使用的命名空间为：

```
namespace ImageShow;
```

5.3.2 图像显示控件的类设计

用以提供图像处理功能和显示图像处理结果的图像显示控件 ImageShow 的类设计如下所示：

```
public partial class ImageShow : UserControl
{
    public Image image;
    public int[] nPara = new int[10];
    public float[] fPara = new float[10];
    public FilterType eFilterType = FilterType.Gaussian;
    public ProcessType eProcessType = ProcessType.CUDA;
    public float fZoomRatio = 1.0f;
    private double dStartTime = 0;
    private double dEndTime = 0;
    private double dDuration = 0;
}
```

其中，成员变量 image 用于存储需要处理的图像数据，图像处理默认使用 image 变量中存储的数据进行；成员变量 nPara 和 fPara 用于存储一般性的图像处理需要使用的 int 类型和 float 类型的数组变量；成员变量 eFilterType 用于指示图像控件

用于图像滤波的方法，默认使用 Gaussian 滤波。枚举类型 FilterType 的定义为：

```
public enum FilterType
{
    Gaussian,
    Butterworth,
    Average
}
```

成员变量 eProcessType 用于指示图像控件用于图像处理所使用的方法，默认使用 CUDA 方式进行图像处理。枚举类型 ProcessType 的定义为：

```
public enum ProcessType
{
    CPP,
    CUDA
}
```

ImageShow 类中，包括成员函数 ZoomImage 和 FilterImage，其函数声明如下所示：

public int ZoomImage（ProcessType eType, int[] npara, float[] fpara）;

public int FilterImage（ProcessType eType, int[] npara, float[] fpara）;

成员函数 ZoomImage 用于实现图像的缩放功能，其中，形参 eType 表示图像处理使用的类型（C++方法或者 CUDA 方法）；形参 npara 用以传递图像处理过程所需要的整型参数；形参 fpara 用以传递图像处理过程所需要的浮点型参数；函数的返回类型为整型值，用以返回函数执行成功与否或者函数执行错误时的错误码。函数默认对类成员变量 image 当中存储的数据进行处理。

成员函数 FilterImage 用于实现图像的缩放功能，其参数定义和返回值类型与成员函数 ZoomImage 的定义与返回值类型类似。

成员变量 dStartTime 记录本次调用函数的起始时间；成员变量 dEndTime 记录本次调用函数的结束时间；成员变量 dDuration 记录本次调用函数的耗时，计算方式为：dDuration = dEndTime - dStartTime。

5.3.3 集成测试平台界面的按钮功能实现

集成测试平台界面的按钮功能的实现，通过界面上按钮的 click 事件来实现。在集成测试平台的设计器的“工具箱”中，添加入自定义的 ImageShow 控件后，可以实现 ImageShow 控件功能的使用。

对于集成测试平台中的文本框控件内容的使用，直接读取其中的 String 类型的字符数据，并将数据转换成 float 类型，用以计算；对于集成测试平台中的组合框控件内容的使用，需要在其 SelectedValueChanged 事件中，对图像处理所需要的参数值，根据组合框的选项，进行写入处理。

在集成测试平台中，图像显示控件在集成测试平台中的 Name 属性为 imageshow；“图像缩放”按钮的 Name 属性为 CPP_ZoomImage；“CUDA 缩放”按钮的 Name 属性为 CUDA_ZoomImage；“滤波处理”按钮的 Name 属性为 CPP_ FilterImage；“CUDA 滤波”按钮的 Name 属性为 CUDA_ FilterImage；文本框控件的 Name 属性为 ZoomRatio；滤波器类型组合框的 Name 属性为 FileteType；滤波器尺寸组合框的 Name 属性为 FileteSize.

以图像滤波处理功能的 CUDA 滤波功能的按钮实现为例，其实现方式及过程如下：

（1）滤波器类型组合框的 SelectedValueChanged 事件的代码为：

```
private void FileteType_SelectedValueChanged（object sender, System.EventArgs e）
{
    switch （this.FileteType.Text）
    {
        case "高斯滤波器":
            this.imageshow.eFilterType = ImageShow.FilterType.Gaussian;
            break;
        case "巴特沃斯滤波器":
            this.imageshow.eFilterType = ImageShow.FilterType.Butterworth;
            break;
        case "均值滤波器":
            this.imageshow.eFilterType = ImageShow.FilterType.Average;
            break;
        default:
            this.imageshow.eFilterType = ImageShow.FilterType.Gaussian;
            break;
    }
}
```

其中，在 default 分支出现时，表示程序出现了错误，则使用高斯滤波器来进行图像处理，以保证程序的稳定性。

（2）滤波器尺寸组合框的 SelectedValueChanged 事件的代码与滤波器类型组合框的 SelectedValueChanged 事件的代码内容类似，即通过比较选择的内容，来决定用于图像滤波的滤波器尺寸，同时，需要根据选择的滤波器尺寸，将图像处理的滤波器参数，写入到成员变量 nPara 和 fPara 当中。

（3）“CUDA 滤波”按钮的 click 事件的代码为：

```
private void CUDA_ FilterImage_Click（object sender, System.EventArgs e）
{
this.imageshow.FilterImage （ ProcessType.CUDA imageshow.nPara, imageshow.
fPara）;
}
```

其中，图像处理所使用的参数，为（1）和（2）中写入的处理参数。

以上为图像滤波处理功能的 CUDA 滤波功能的按钮实现过程。图像处理集成测试平台的其他按钮功能的实现过程与此类似。

5.4 集成测试平台的功能效果

在如 5.2 小节介绍的如图 5 - 1 所示的测试平台中，主要分为两个功能模块区域，即左侧的“图像显示区”，和右侧的“功能区”。在右侧的功能区中，主要提供了五组功能，即“加载图像”、“图像缩放”、“图像滤波”、“其他功能”和“保存图像”。

图像显示区的 Control 控件用于显示图像，并对外提供图像处理的功能接口；图像处理功能按钮区的功能按钮，用于选择所需的图像处理功能和所需的图像处理参数，以及其他有关图像处理的功能。

在功能区的按钮中，“加载图像”功能用于打开要处理的图像，浏览按钮“…”用于打开“打开文件”对话框，以灵活地选择要处理的图像。加载图像的流程实现效果如下面的图 5 - 2 至图 5 - 3 所示：

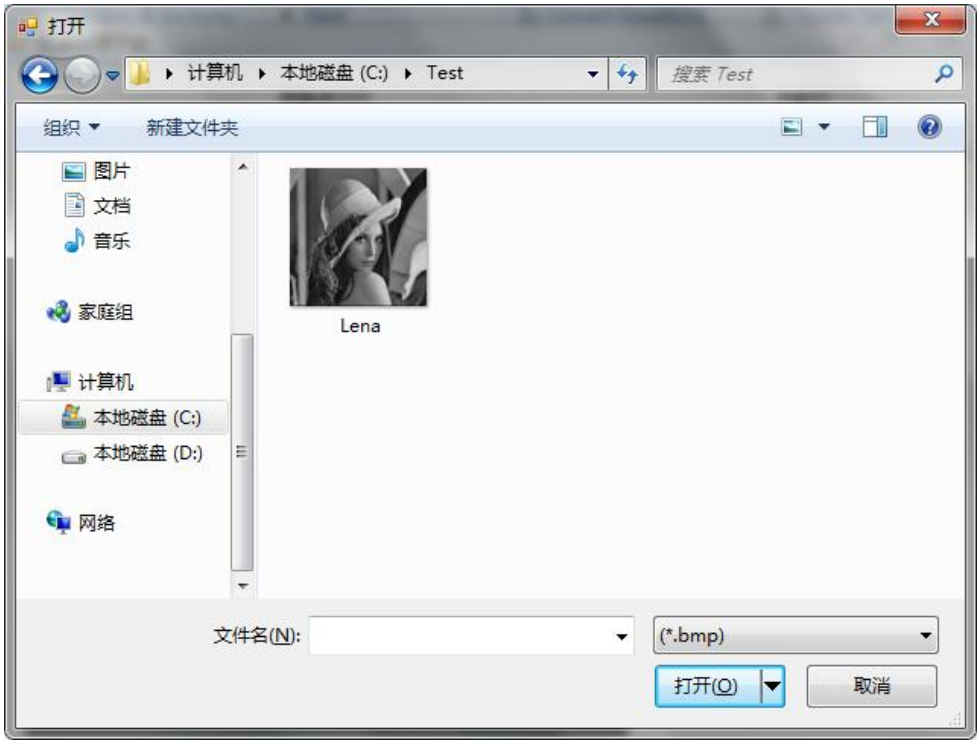

图5-2 浏览按钮打开的“打开文件”对话框效果图

下面的图 5 - 3 和图 5 - 4 所示过程，为测试平台打开实验图像，开始图像处理对比实验的过程示意。

图 5 - 3 为在打开图像对话框关闭之后的效果，其中，打开文件的路劲显示在了路径文本框中，点击“加载图像”按钮，就可以加载其中的图像到图像显示区（效果如图 5 - 4 所示）。在这里，图像滤波功能区的功能属于独立环节，因此不受影响。

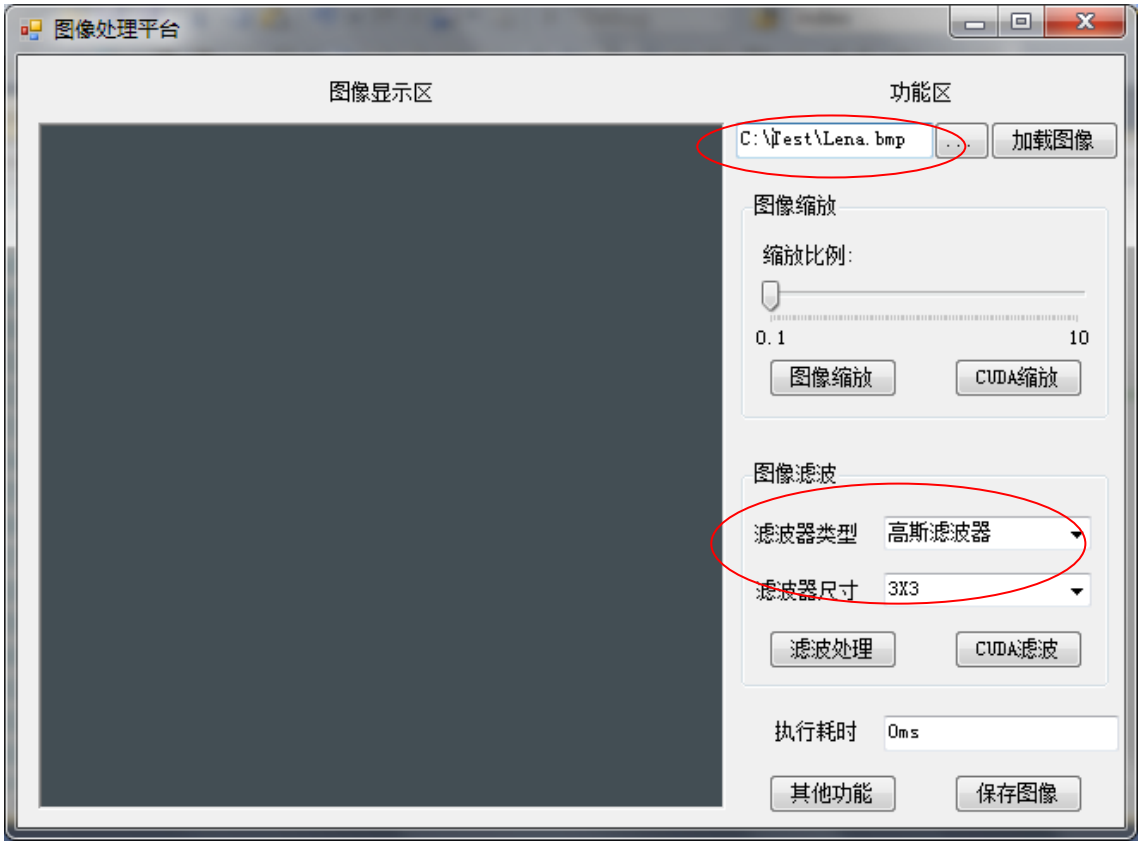

图5-3 选择了测试图片之后的文本框现实文件路径的效果图

下面的图 5 - 4 即为加载了 Lena 图之后，测试平台的显示情况，从图中可以看出，实验图片被加载到了图像显示区域，各功能调用按钮可以随时调用。

图5-4 加载实验图像之后的效果图

下面的图 5 - 5 和图 5 - 6 所示过程，为使用测试平台打进行图像缩放功能实验的过程示意。

在“图像缩放”子功能区，可以通过滑块，来调节要实现的图像缩放的比例，可以实现放大比例从 0.1 至 10.0 不等的缩放比例，并且可以选择图像处理的实现模式，包括普通模式的“图像缩放”和 CUDA 模式的“CUDA 缩放”，从而可以通过处理日志（前文已经介绍）来对比两种处理方式的性能。

图像缩放功能的效果图，如下面的图 5 - 5 和图 5 - 6 的效果所示。

图 5 - 5 为使用 0.5 的缩放比进行的 Lena 图的缩小显示的处理效果；图 5 - 6 为使用 3.6 的缩放比进行的 Lena 图的放大显示的处理效果。图上的“执行耗时”文本框中显示了本次调用所耗费的时间，其计算方法为用记录的函数执行的结束时间，减去函数执行的开始时间。从时间上来看，使用 CUDA 方法实现的函数的执行时间，较之使用普通的 C++方式实现的时间，算法耗时缩减很多，算法的执行效率的提升效果明显。

从图 5 - 6 的处理效果可见，由于算法的限制，图像出现了“马赛克”效果。

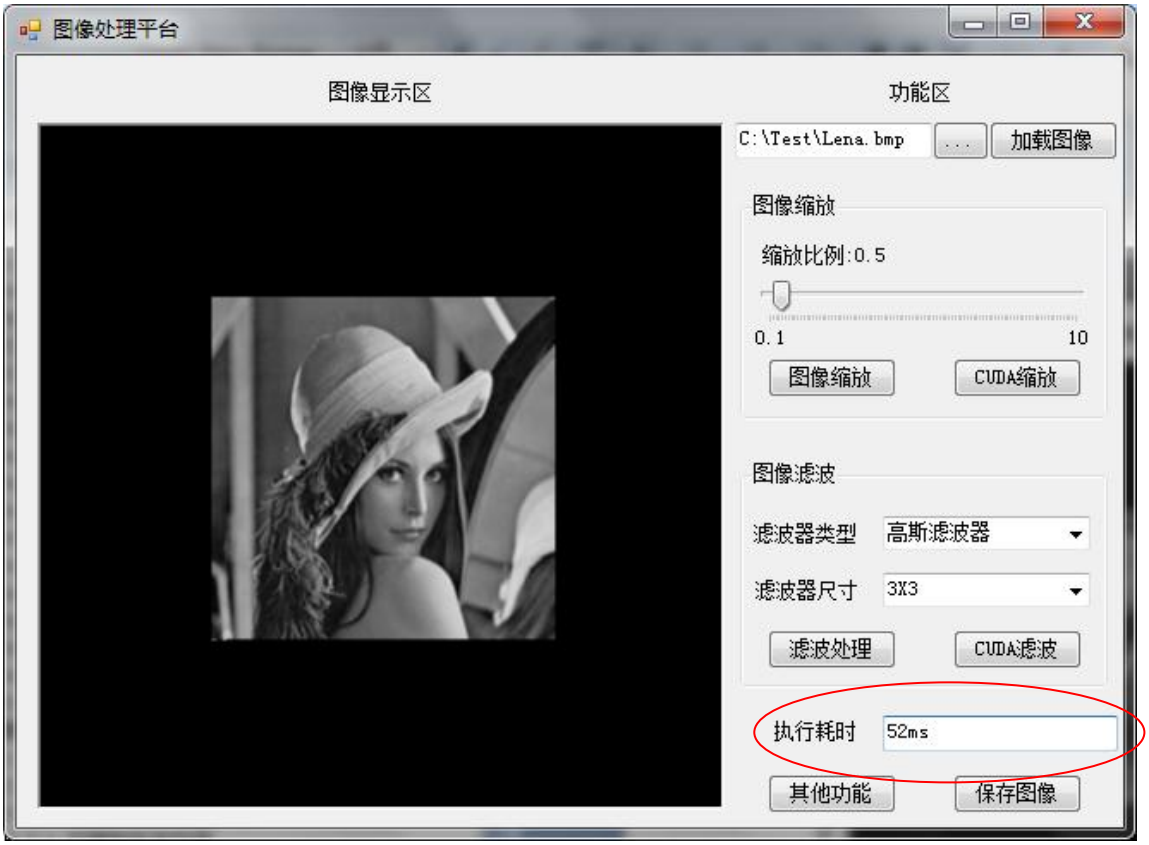

图5-5 Lena图普通缩放0.5倍的效果图

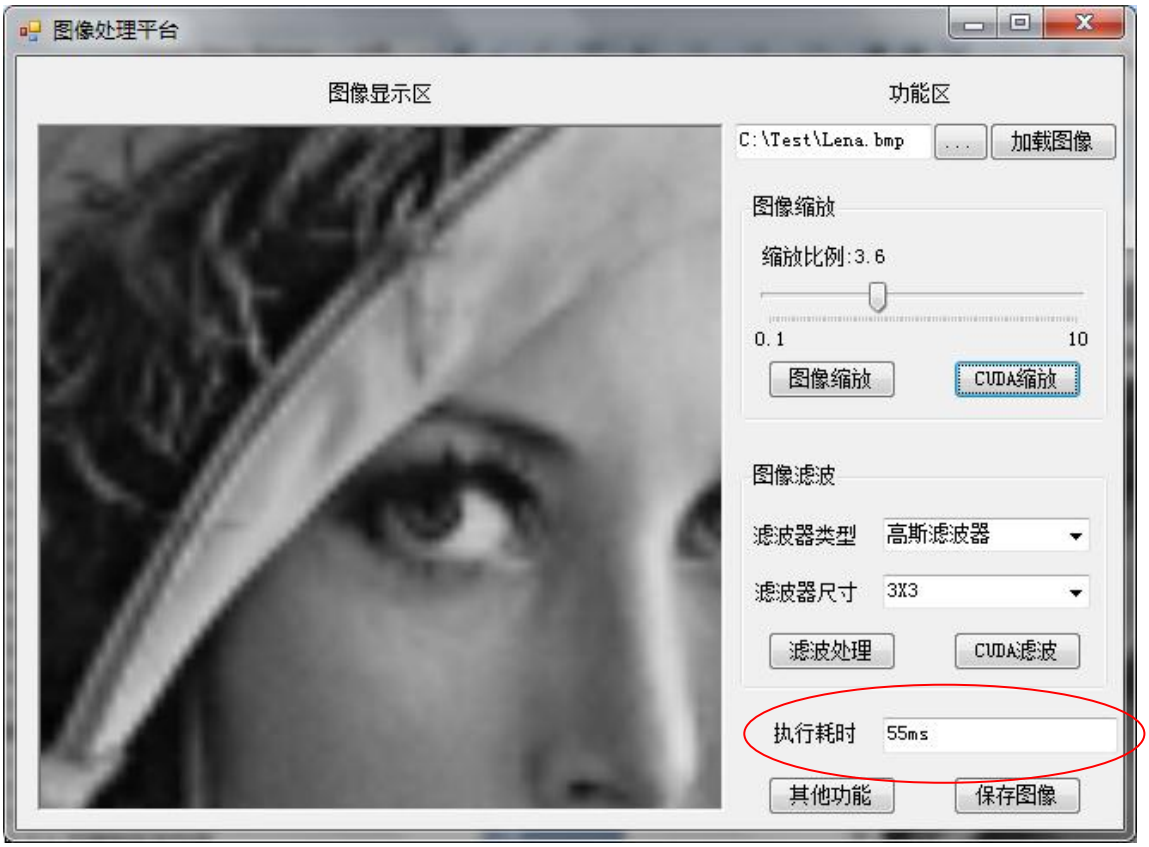

图5-6 Lena图使用CUDA模式放大3.6倍的效果图

下面的图 5 - 7 和图 5 - 9 所示过程，为使用测试平台打进行图像的滤波处理功能实验的过程示意。同样，图上的“执行耗时”文本框中显示了本次调用所耗费的时间，其计算方法为用记录的函数执行的结束时间，减去函数执行的开始时间。从时间上来看，使用 CUDA 方法实现的函数的执行时间，较之使用普通的 C++方式实现的时间，算法耗时缩减很多，算法的执行效率的提升效果明显。

在“图像滤波”子功能区，可以以用户选择的滤波器类型和滤波器尺寸，并使用普通模式和 CUDA 模式两种模式，来实现图像的滤波处理功能，默认使用的滤波器为 3×3 的高斯滤波器。

加载打开实验用的电路板 X 光图像，加入强度为 3 的高斯噪声后的电路板图像的效果，如下面的图 5-7 所示（图像缩放功能区是一个独立的功能模块区域，由于本次实验流程的连续性，其状态没有改变，仍保留了在进行图像缩放实验时所最后出于的状态）：

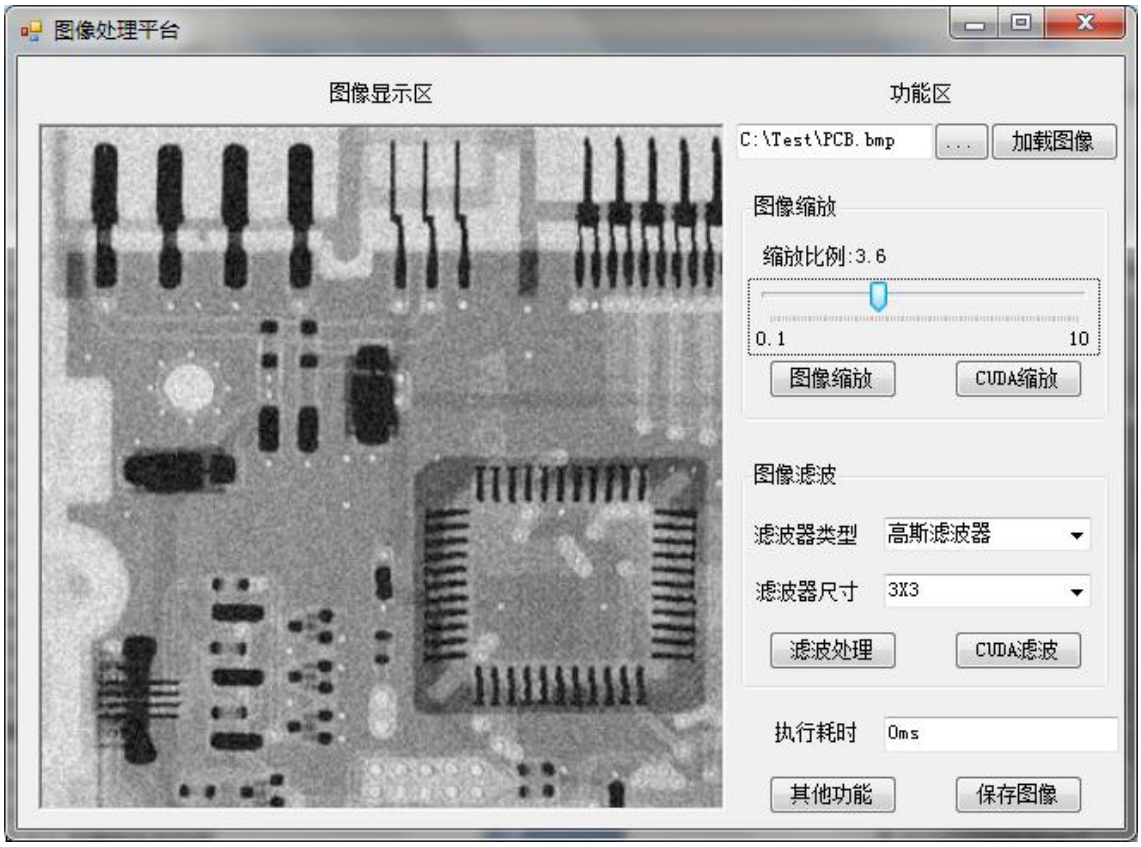

图5-7 加载了加入强度为3的高斯噪声后的电路板图像的效果

使用图像滤波功能区提供的滤波器类型选项和滤波器尺寸选项，选择合适的处理参数效果，在此，使用默认的 3×3 的高斯滤波器，以普通方式进行滤波处理

后的效果，如下面的图 5-8 所示：

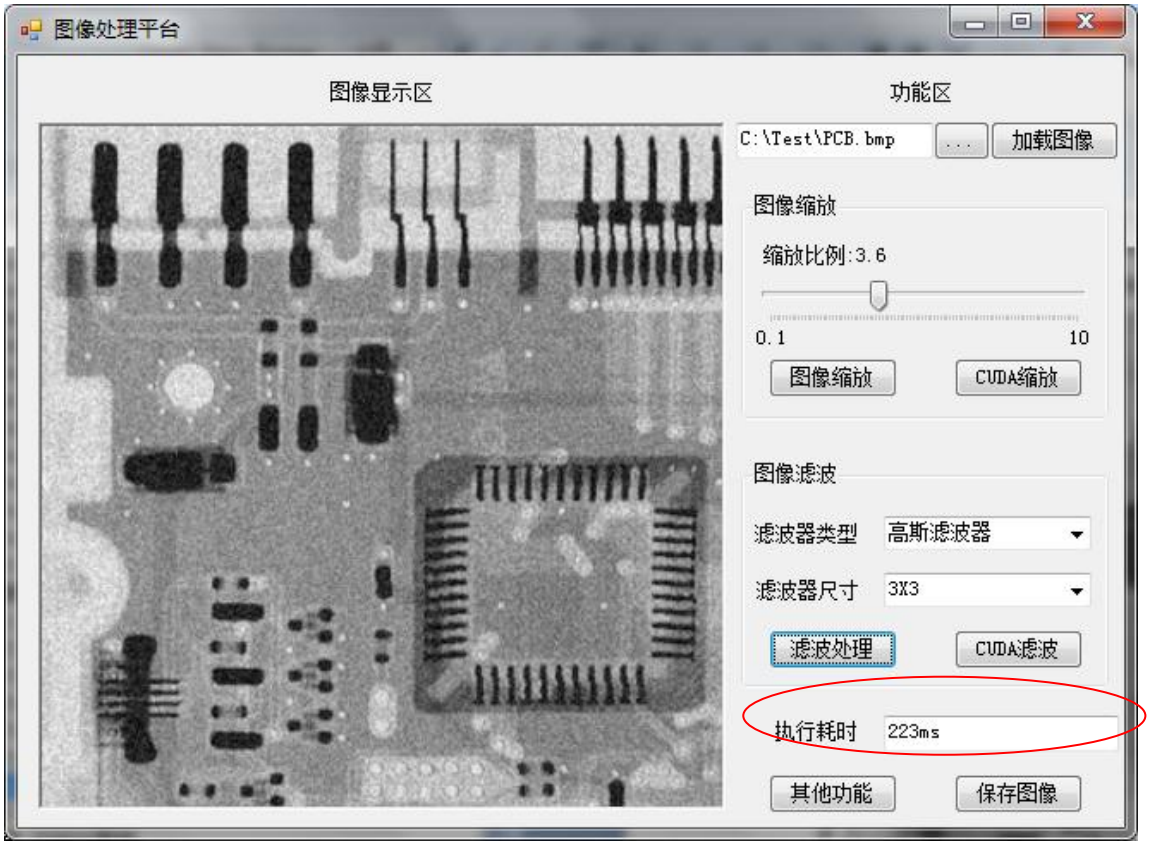

图5-8 使用3×3的高斯滤波器，以普通方式进行滤波处理后的效果

使用 5×5 的巴特沃斯滤波器，以 CUDA 方式进行滤波处理后的效果如下面的图 5-9 所示：

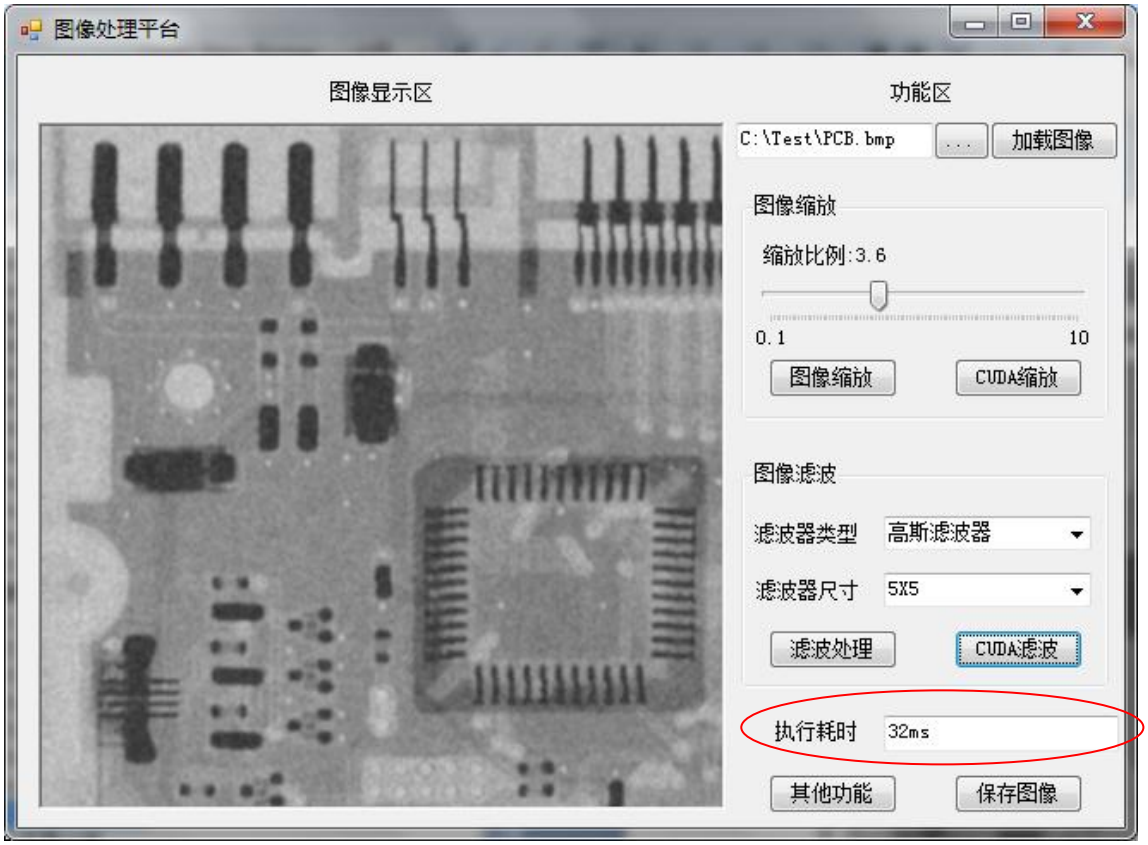

图5-9 使用5×5的巴特沃斯滤波器，以CUDA方式进行滤波处理后的效果

集成测试平台中的“其他功能”选项，是为了后续的集成测试平台的扩展预留的接口，在本版本的集成测试平台中，并没有实现相关的功能。

集成测试平台中的“保存图像”功能，可以打开“保存图像”对话框，用以灵活地将图像处理的结果进行保存。保存处理结果的过程效果如下面的图 5-10、图 5-11 所示：

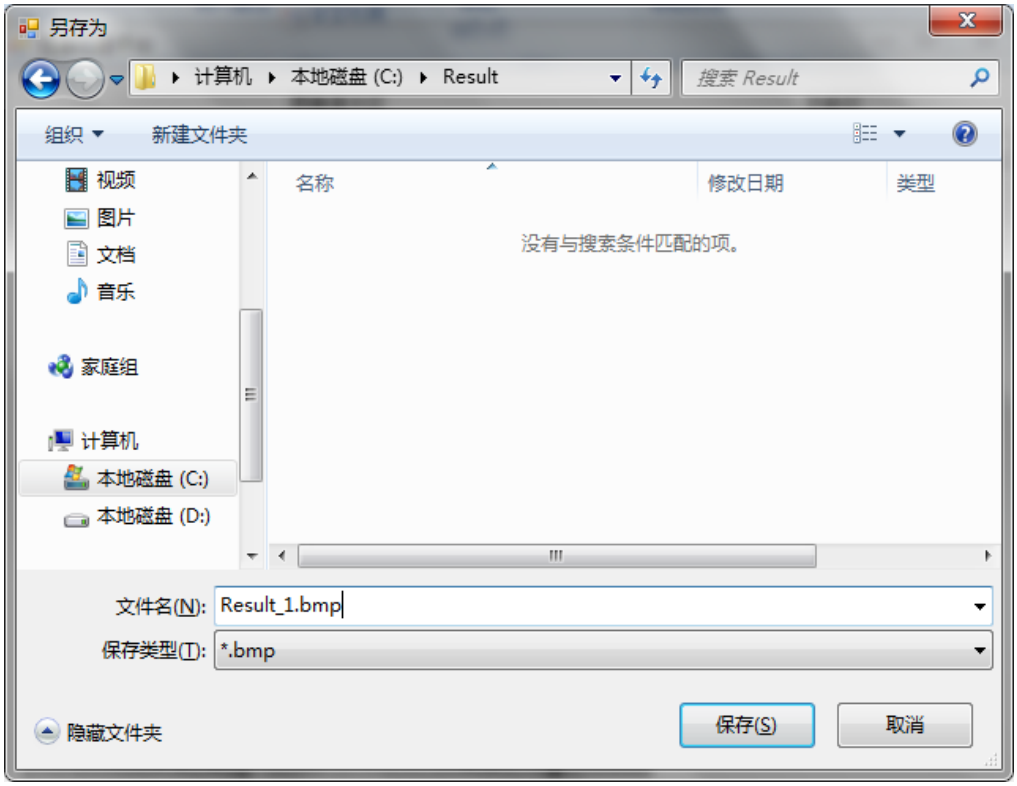

图5-10 “保存图像”按钮打开的“保存文件”对话框效果图

至此，本集成测试平台所实现的功能举例说明完毕。

5.5 其他功能展示

本测试平台，除了提供上述的测试功能外，还对进行图像滤波的方法进行了其他实验，如下面的图 5 - 11 所示，其中，除了提供高斯滤波器外，还提供了巴特沃斯滤波器和均值滤波器，其实现的效果和高斯滤波器的效果相似，在此不再演示。

图 5-11 测试平台提供的其他图像滤波程序示意图

5.6 本章小结

本章内容实现了一个图像处理算法的集成测试平台，从而为测试 CUDA 架构实现的快速图像处理程序与使用传统 C 语音实现的图像处理程序的性能对比提供技术支持。

首先，本章对集成测试平台的开发与测试环境进行了介绍，通过确定集成测试平台的开发环境，从而保证了本测试程序对本文研究的目的——测试 CUDA 架构在通用计算方面的性能提升——的可测性、测量条件的一致性与测量结果的有效性。

其次，本章设计并实现了集成测试平台的界面原型。需要说明的是，本次设计的测试平台的界面原型，只是针对本文的研究目的所设计的一种解决方案，不具有唯一可行性，在未来的课题相关的更加深入的研究工作中，需要适当地调整相应的解决方案的设计。

第三，本文对所开发的集成测试平台的实现进行了简要的介绍，包括命名空间引用、控件类的设计，以及按钮功能的实现等，从而实现了整个集成测试平台的功能。

最后，本章对使用本集成测试平台所进行的实验进行了介绍，包括具体的实验步骤，并截取了相应的界面处理效果，从而使本章对集成测试平台的设计与实现的效果更加直观、明确。

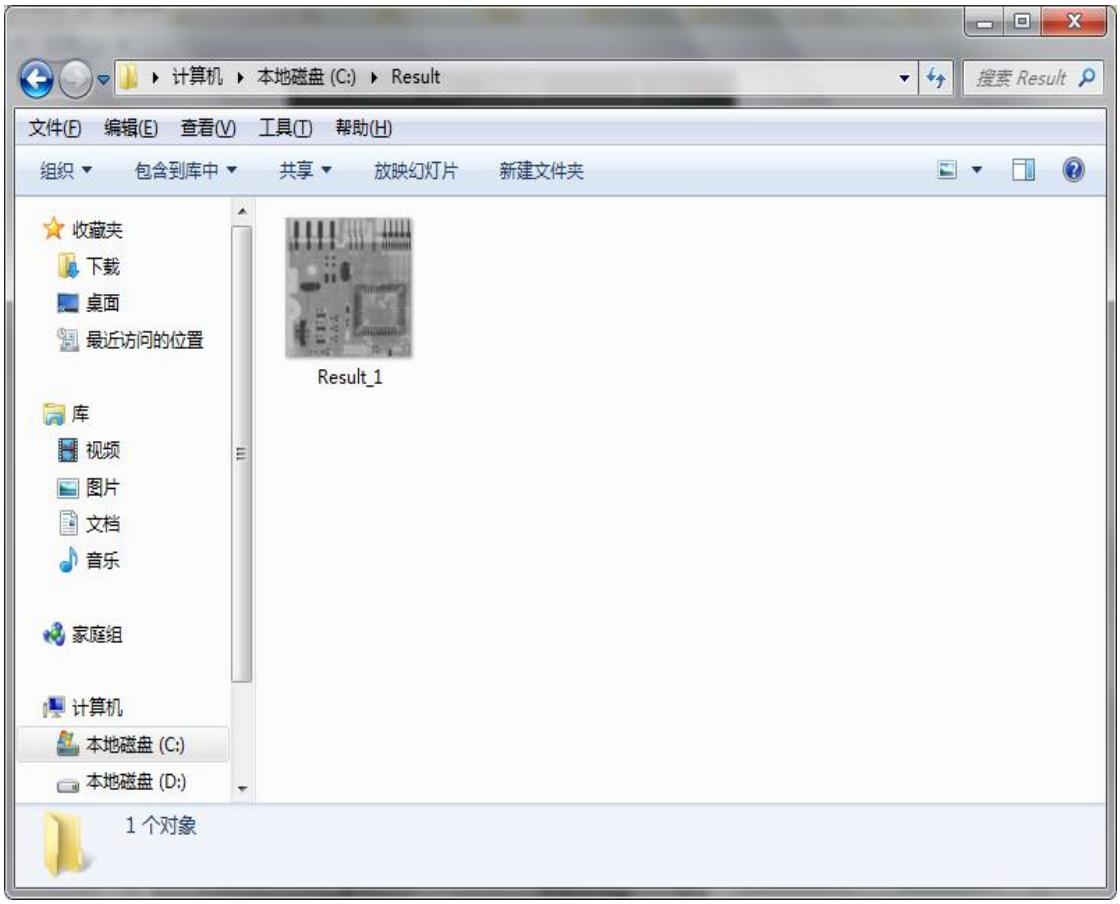

图5-11 文件保存的结果示意图

第六章 总结与展望

6.1 全文总结

高性能计算在数据加密、工程计算、DNA 分子测序和空间结构模拟、地质勘探、海洋信息监测、地质灾害预测、天气预报、城市交通调度、模拟实验、图形图像处理、数字水印、网络游戏、电子商务、搜索引擎等众多自然科学、社会科学以及工程应用领域发挥着重要的作用，同时，在公安政法领域，对图像处理速度和精度的要求十分突出，因此，通用高性能计算成为多种学科领域中现代化的实现方法中不可或缺的高端计算工具，其应用研究越来越受到科研工作者和工程技术人员的重视与青睐，研究并解决高性能计算在应用中的问题，具有十分重要的理论与现实意义。

另一方面，从云计算的核心思想与实现途径来看，高性能计算与云计算具有相同的技术发展出发点，这使得高性能计算成为了云计算发展的重要基础和性能提高的突破点，同时使对高性能计算的研究成为了一个古老而又崭新的学术研究方向，更加凸显了对高性能计算研究的现实意义。

随着大数据分析在金融行业、气象灾害预报、军事科技、生物医学等领域的应用越来越广泛与深入，高性能计算的研究越来越凸显了其必要性，而近年来兴起的云计算应用，在本质上也属于高性能计算。

GPU 在高性能计算领域的应用，是非常具有前瞻性的，由于 GPU 的设计早已更多地考虑了对大数据量的计算能力，所以 GPU 发展到今天，其计算能力已经远远超过 CPU，在计算密集型的高性能计算领域，GPU 所扮演的角色也将越来越重要。

GPU 编程采用的 CUDA 架构，作为一种先进的高性能计算应用的实现载体，其应用研究的意义是显而易见的。

本文在上述的背景下，深入研究了 GPU 并行计算与 CUDA 发展史，并在深入学习了 GPU 编程的基础上，实现了基于 CUDA 架构的高性能计算的几个应用子系统，实验数据显示，本套图像处理子系统是可行的，并且是快速的。

本文围绕基于 CUDA 架构的高性能图像处理程序设计课题的研究，主要进行了以下工作：

（1）本文对 GPU 及 CUDA 架构的发展历程进行了回顾，通过对比分析，总结得出了基于 CUDA 架构所实现的 GPU 的通用高性能计算，相对于在传统的 CPU 上实现的高性能计算的优势所在，即性价比高、功耗低、体积小，同时，基于 CUDA

架构的通用高性能计算，性能更强。通过这样的分析结论，奠定了本文课题研究的技术先进性的基础，从而为后续的课题研究提供了保障。

（2）从 CUDA 的程序结构，CUDA 编程中的任务分解思想、存储器优化思想、多线程同步思想的角度，介绍了 CUDA 架构的编程思想及程序的优化方式，为本文课题研究的程序开发提供编程与优化原则，主要包括以下几个方面：一是将 CUDA 程序执行的任务，尽可能地分解为可并行执行的部分以及必须的逻辑控制部分，并且尽量减少 CPU 与 GPU 之间的数据与控制交互；二是使用任务分解的思想与原则减少 CPU 系统内存与 GPU 显存之间的数据交互；三是对全局存储器和共享存储器访问进行优化；四是尽量满足多个线程对全局存储器的访问的起始地址对齐，并且每个线程与各自要访问的存储地址对齐；五是通过增加一次在 GPU 上执行的线程数量，从而提高数据计算密度的方式来弱化全局存储器访问的时间延迟；减少线程同步的频率。

（3）本文分析了图像处理的需求特点，以及在日常应用中的主要的几种图像处理需求，包括图像的插值需求和滤波处理需求，针对这两种图像处理需求，分别使用 C++语言和 CUDA C 语言，实现了快速的图像处理程序，并对两种编程语言实现的图像处理程序的处理性能进行了实验验证，进行了实验数据分析，从而得出，CUDA 架构对快速图像处理程序能力的显著的提升作用。实验数据表明，使用 CUDA 实现的图像处理，比使用普通的 CUP 所实现的图像处理，其性能得到了较大幅度的提升。执行同样的图像双线性差值算法进行图像差值处理，使用 CUDA 的执行时间，在本文的实验条件下，相比于使用 CPU 执行所耗费的时间，性能提高在 70%左右；而执行同样的图像高斯滤波运算处理，相比于使用 CPU 执行所耗费的时间，性能提高在 85%左右，显示了 CUDA 的较高的执行能力。

（4）本文同时开发了一个简单的集成测试平台，用以对 CUDA 架构的通用高性能处理程序的开发的实验验证提供支持。对集成测试平台的开发与测试环境进行了介绍，设计并实现了集成测试平台的界面原型，控件类的设计，以及按钮功能的实现等，并对使用本集成测试平台所进行的实验进行了介绍，包括具体的实验步骤，并截取了相应的界面处理效果，从而使本章对集成测试平台的设计与实现的效果更加直观、明确。

6.2 展望

CUDA 高性能计算是未来高性能计算领域应用的主流，从硬件发展的角度来看，提高 GPU 的硬件能力，针对特定的应用，设计开发具有专门处理功能的硬件芯片，同时简化 CUDA 架构的编程，是发展基于 CUDA 架构的高性能计算的基础，

是研发人员需要努力的重点方向。

从软件的角度来看，围绕基于 CUDA 架构的高性能编程尚有许多问题需要解决，一是并行计算任务的分解方法，二是针对 CUDA 架构的特殊性，对应用人员的编程思想和方式进行系统化、科学化的培训等，这些都是我们在产品级上应用 CUDA 架构时，所面临的首要的，并且是无法回避的问题，是需要我们继续深入研究的。

致 谢

本论文的课题研究以及学位论文的写作过程，是在我的导师的亲切关怀和悉心指导下完成的。导师严肃的科学态度，严谨的治学精神，精益求精的工作作风，深深地感染和激励着我。从课题的选择到项目的最终完成，老师都始终给予了我细心的指导和不懈的支持。

两年多来，老师不仅在学业上给我以精心指导，同时还在思想上、生活上给我以无微不至的关怀，在此谨向老师致以诚挚的谢意和崇高的敬意。

在此，我还要感谢在一起愉快的度过研究生生活的同学们，正是由于你们的帮助和支持，我才能够克服一个一个的困难和疑惑，直至本文的顺利完成。

在论文即将完成之际，我的心情无法平静，从开始进入课题到论文的顺利完成，有多少可敬的师长、同学、朋友给了我无言的帮助，在这里请接受我诚挚的谢意！最后我还要感谢培养我长大含辛茹苦的父母，谢谢你们！

参考文献

[1] 史佩昌，王怀民，蒋杰，卢凯. 面向云计算的网络化平台研究与实现[J]. 计算机工程与科学, 2009, 31(A1): 249~252.

[2] K. Zhou, Q. Hou, R. Wang, et al. Real-time KD - tree Construction on Graphics Hardware [J]. ACM Trans on Graphics, 2008, 27 (5): Article No.126.

[3] R. Wang, R. Wang, K. Zhou, et al. An efficient GPU-based Approach for Interactive Global Illumination [C] //ACM SIGGRAPH, New York, ACM Press, 2009: Article No.91.

[4] D. Goddeke, R. Strzodka. Cyclic Reduction Tridiagonal Solvers on GPUs Applied to Mixed-Precision Multigrid [J]. IEEE Trans on Parallel and Distributed Systems, 2011, 22(1): 22-32.

[5] M. Chao, C. Chu, C. Chao, et al. Efficient Parallelized Particle Filtering Design on CUDA [C] //Proc of IEEE Workshop on Signal Processing Systems, 2010, CA, USA, 2010:299-304.

[6] S. Lahabar, P. J. Narayanan. Singular Value Decomposition on GPU using CUDA [C]//Proc of IEEE Int Sym on Parallel & Distributed Processing, 2009, Rome, Italy.

[7] C. Juang, T. Chen, W. Cheng. Speedup of Implementing Fuzzy Neural Networks With High - Dimensional Inputs Through Parallel Processing on Graphic Processing Units [J]. IEEE Trans on Fuzzy Systems, 2011, 19 (4): 717-728.

[8] S. Islam, R. Tandon, S. Singh, et al. A Highly Scalable Solution of an NP-Complete Problem using CUDA [C]//Proc of IEEE Int Sym on PARELEC, 2011, Luton, UK, 2011:93-98.

[9] 陈曦，王章野，何戡等. GPU 中的流体场景实时模拟算法[J]. 计算机辅助设计与图形学学报, 2010, 22(3):396~405.

[10] 江雯倩，钟鼎坤，解张鹏等. 基于 CPU/GPU 的日冕偏振亮度并行计算模型[J]. 空间科学学报, 2011 31(1):51-56.

[11] H. Shi, B. Schmidt, W. Liu, et al. Accelerating Error Correction in High-Throughput Short-Read DNA Sequencing Data with CUDA [C] //Proc of IEEE Int Sym on Parallel & Distributed Processing, 2009, Rome, Italy.

[12] Y. Zhou, X. Wu, J. Haldar, et al. Accelerating Iterative Field-compensated MR Image Reconstruction on GPUs [C]//Proc of IEEE Int Sym on Biomedical Imaging, 2010, Rotterdam, Netherlands, 2010:820-823.

[13] C. Song, Y. Li, B. Huang. A GPU-Accelerated Wavelet Decompression System with SPIHT and Reed-Solomon Decoding for Satellite Images[J]. IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, 2011,4(3):683-690.

[14] 程俊仁, 刘光斌, 张博. 基于 CUDA 的 GPS 信号快速捕获[J]. 宇航学报, 2010, 31(10): 2407-2410.

[15] 张兵, 赵改善, 黄骏等. 地震叠前深度偏移在 CUDA 平台上的实现[J]. 勘探地球物理进展, 2008, 31(6): 427-432.

[16] G. Cena, M. Cereia, S. Scanzio, et al. A High-Performance CUDA-Based Computing Platform for Industrial Control Systems [c]//Proc of IEEE Int Sym on Industrial Electronics, 2011, Gdansk, Poland, 2011:1169-1 174.

[17] 李晓敏, 侯朝焕, 鄢社锋. 一种基于 GPU 的主动声纳带宽信号处理实时系统[J]. 传感技术学报, 2011,24(9):1279-1284.

[18] NVIDIA CUDA Programming Guide [EB/OL]. http://developer. download. nvidia. com/compute/cuda/22/toolkit/docs/NVIDIA CUDA Programming Guide 2.2.1. pdf, 2009-5-26.

[19] NVIDIA CUDA Programming Guide [M]. Version 2.1. NVIDIA Corporation, 2008.

[20] General-purpose computation using graphics hardware [EB/OL]. [2010-05-23]. http: //www. gpgpu. org/

[21] KENNETH M, EDWARD A. The FFT on a GPU [C]//Proc of the ACM SIGGRAPH/ EUROGRAPHICS Conference on Graphics Hard-ware. San Diego, California: Euro graphics Association, 2003:112-119.

[22] 张舒, 褚艳利, 赵开勇等. GPU 高性能运算之 CUDA [M]. 北京: 中国水利水电出版社, 2009.

[23] NVIDIA CUDA Reference Manual [EB/OL]. http://developer. download. nvidia. com/ compute/cuda/22 /toolkit/docs/CUDA Reference Manual 2. 2. pdf 2009-4.

[24] 许建平, 统一计算设备架构技术的应用研究进展[J]. 舰船电子工程, 2011, 31(12): 1 - 5, 9.

[25] NVIDIA Corporation. CUDA Programming Guide 1. 0 [EB / OL]. http://wwwnvidia. com, 2007

[26] Tom R. Halthil. Parallel Processing With CUDA. Microprocessor Report[R], Scottsdale, Arizona, Jan 28, 2008

[27] 吴恩华. 图形处理器用于通用计算的技术、现状及其挑战[J]. 软件学报. 2004,15(10): 1493-1504

[28] John D. Owens, Mike Houston, David Luebke, etal. . GPU Computing [J]. Proceedings of the IEEE, 2008,96 (5): 879 - 897

[29] NVIDIA CUDA Compute Unified Device Architecture Programming Guide. V2. 0. 2008

[30] M. Pharr. GPU 精粹 2: 高性能图形芯片和通用计算编程技巧[M]. 龚敏敏. 清华大学出版社, 2007: 357 – 376

[31] K. Moreland, E. Angel. The FFT on a GPU [J]. Proceedings of the ACM SIGGRAPH / EURO GRAPHICS Conference on Graphics Hardware. 2003:112-119

[32] J. Fung, S. Mann. Computer Vision Signal Processing on Graphics Processing Units [J]. Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing. 2004, 5: 93-96

[33] K. Bjorke. Image Processing on Parallel GPU Pixel Units [J]. Computational Imaging IV. 2006, 6065: 342-353

[34] C. Bruyns, B. Feldman. Image Processing on the GPU: A Canonical Example. http: //www. cs. berkeley. Edu/-kubitron/courses/cs 252-F03/projects/reports/project 12_report_ver 2. pdf, 2003

[35] H. Scherl, B. Keck, M. Kowarschik and J. Hornegger. Fast GPU-Based CTR e construction Using the Common Unified Device Architecture (CUDA)[J]. IEEE Nuclear Science Symposium Conference Record. 2007, 6: 4464-4466

[36] J. Michalakes, M. Vachharajani. GPU Acceleration of Numerical Weather Prediction [J]. Parallel and Distributed Processing. 2008:1-7

[37] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos and S. Ioannidis. Gnort: High Performance Network Intrusion Detection Using Graphics Processors [J]. 11 th International Symposium, Recent Advances in Intrusion Detection. 2008: 116-134

[38] Y. C. Luo, R. Duraiswami. Canny Edge Detection on NVIDIA CUDA [J]. Computer Vision and Pattern Recognition Workshops. 2008:1-8

[39] G. B. Shen, G. P. Gao. Accelerate Video Decoding With Generic GPU [J]. IEEE Transactions on Circuits and Systems for Video Technology. 2004, 15(5):685-693

[40] Graphics Processing Unit (GPU). http: //www. nvidia. Com/object/gpu. Html

[41] 章毓晋. 图像工程第二版[M]. 清华大学出版社, 2007

[42] 阮秋琦. 数字图像处理学第二版[M]. 电子工业出版社, 2007: 199-203

[43] 王文豪. 图像边缘检测中边界闭合性的分析与探讨[J]. 计算机与信息技术, 2005, 12:90-91

[44] 田娟, 郑郁正. 模板匹配技术在图像识别中的应用[J]. 传感器与微系统. 2008, 27(1): 112-113

[45] Y. H. Lin, C. H. Chen, C. C. Wei. New Method for Sub pixel Image Matching with Rotation Invariance by Combining the Parametric Template Method and the Ring Projection Transform Process [J]. Optical Engineering, 2006, 45(6):1-9

[46] J. Flusser, J. Boldys, B. Zitova. Moment Forms Invariant to Rotation and Blurin Arbitrary Number of Dimensions [J]. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2003, 25(2):234-246

[47] Y. Y. Tang, B. F. Li, H. MaandJ. Liu. Ring-Projection-Wavelet-Fractal Signatures: A Novel Approach to Feature Extraction [J]. IEEE Transaction on Circuits and Systems. 1998, 45: 1130-1134

[48] 曹炬, 马杰, 谭毅华, 田金文. 基于像素抽样的快速互相关性图像匹配算法[J]. 宇航学报, 2004, 25(2):174-179

[49] K. Fukunaga, L. Hostetler. The Estimation of the Gradient of a Density Function, with Applications in Pattern Recognition [J]. IEEE Transactions Information Theory, 1975, 21(1): 32-40

[50] D. Comaniciu, P. Meer. Mean Shift: A Robust Application Toward Feature Space Analysis [J]. IEEE Transactions on Pattern Analysis and Machine Intelligence. 2002, 24 (5): 603-619

[51] S. L. Zhu, S. A. Zhu, X. C. Li. Algorithm for Tracking of Fast Motion Objects with Mean Shift [J]. Opto-Electronic Engineering. 2006, 33(5): 66-70

[52] J. Wang, B. Thiesson, Y. Xu and M. Cohen. Image and Video Segmentation by Anisotropic Kernel Mean Shift[J]. Proceedings of European Conference on Computer Vision. 2004, 238-249

[53] K. Zhang, J. T. Kwok, M. Tang. Accelerated Convergence Using Dynamic Mean Shift[J]. European Conference on Computer Vision, 2006, 2: 257-268

[54] 李波, 赵华成, 张敏芳. CUDA 高性能计算并行编程 [J]. 微型电脑应用, 2009, 25(9): 55-64.

[55] 张军华, 臧胜涛, 单联瑜, 等. 高性能计算的发展现状及趋势[J]. 石油地球物理勘探, 2010, 45(6): 918-925.

[56] 陈波, 韩永国, 刘志勤. 高性能并行计算的研究与分析[J]. 四川师范学院学报: 自然科学版, 2003, 24(2): 200-202.

[57] 倪颖杰, 王律科, 张军. 基于高性能数据挖掘的网络海量信息处理平台 [J]. 计算机工程与科学, 2009, 31(1): 129-131.

[58] 王国明. 高性能集群计算系统的结构与设计探讨[J]. 电脑知识与技术: 学术交流, 2006 (12): 158-159.

[59] 郭境峰，蔡伟涛. 新一代高性能运算技术 —— CUDA 简介[J]. 现代科技，2009，8(6): 29-30.

[60] 吴长茂，张聪品，张慧云，等. CUDA 平台下多核 GPU 高性能并行编程研究[J]. 河南机电高等专科学校学报, 2011, 19(1): 19-21.

[61] 曹宗雁. 高性能计算集群运行时环境的配置优化[J]. 科研信息化技术与应用，2011，2(6): 52-60.

[62] 郑启龙，刘术娟，胡晨光，等. 基于 Eclipse/Web Services 的高性能计算机远程服务与开发环境[J]. 微电子学与计算机, 2007, 24(9): 151-154.

[63] 林茂，董玉敏，邹杰，等. GPGPU 编程技术初探 [J]. 电脑编程技巧与维护，2010 (2): 15-17.

[64] 张艳，代巧莲，黄奇珊，等. 高性能运算之 CUDA 应用分析[J]. 电信技术研究，2012 (3): 42-46.

[65] 赵华成，李波. CUDA 高性能计算并行编程 [J]. 高性能计算技术, 2009 (006): 17-21.