分类号：_____ 密级：_____

编号：102009328

桂林理工大学

硕士研究生学位论文

基于CUDA并行架构AES算法的研究与实现

专　　业：计算机应用技术

研究方向：并行计算

研 究 生：马梦琦

指导教师：刘羽 教授

论文起止日期：2011年4月至2012年4月

Research and Design of AES Algorithm Based on CUDA Parallel Architecture

Major: Computer Application Technology

Direction of Study: Parallel Computing

Graduate Student: Ma Mengqi

Supervisor: Prof.Liu Yu

College of Information Science and Engineering

GuiLin University of Technology

April,2011 to April,2012

研究生学位论文独创性声明和版权使用授权书

独 创 性 声 明

本人声明：所呈交的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，论文中不包含他人已经发表或撰写过的研究成果，也不包含为获得其它教育机构的学位或证书而使用过的材料。对论文的完成提供过帮助的有关人员已在论文中作了明确的说明并表示谢意。

学位论文作者（签字）：马梦琦

签字日期：2012.6.14

学位论文版权使用授权书

本学位论文作者完全了解(学校)有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的印刷本和电子版本，允许论文被查阅和借阅。本人授权(学校)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。同时授权中国科学技术信息研究所将本学位论文收录到《中国学位论文全文数据库》，并通过网络向社会公众提供信息服务。(保密的学位论文在解密后适用本授权书)

学位论文作者签名：马梦琦　　导师签字：

签字日期：2012年6月14日　　签字日期：2012年6月14日

摘要

随着计算机网络应用的普及和发展，信息安全作为新兴学科日益受到重视。受 CPU 串行体制的限制，传统的串行密码算法已经越来越不能满足信息安全对运算速度和系统安全性的需求。基于硬件实现的 AES 加速加密器，其功能比较单一、运算效率比较低下，实现通信协议还需要额外的硬件资源。这种方法的不足是成本增大，而性能并不理想。随着 GPU 技术的不断发展，GPU 的计算能力和带宽已经获得显著提高，近年来使用 GPU 对计算密集型应用进行加速已成为研究热点之一。CUDA 编程模型的出现使得基于 GPU 的应用开发更加灵活，实现更加高效。正是在这样背景下，选择"基于 CUDA 并行架构 AES 算法的研究与实现"作为研究课题。

详细讨论了 GPU 并行编程平台 CUDA 独特的软硬件架构；在充分研究标准加密算法 AES 原理基础上，分析 AES 算法实现的传统思想，研究了块密码的工作模式并构建 AES 并行加密算法的模型；讨论并分析基于 CUDA 架构优化技术，使用大规模矩阵乘法测试了 CUDA 编程平台，验证了 GPU 拥有强大浮点运算能力。

在此基础上，对基于 CUDA 并行架构 AES 算法采用查找表的方式对轮函数的运算进行优化，简化了加密轮的执行步数，选择合适的块密码工作模式，深入分析并比较了三种不同粒度的并行划分方案；基于 CUDA 平台实现了改进的并行 AES 加密算法，完成了其实现的主机端程序设计及设备端程序设计；对设计实现的并行 AES 加密算法和原有的 CPU 串行 AES 算法进行了测试和分析；实验结果显示，基于 CUDA 架构的 AES 算法较于 CPU 实现的 AES 算法获得较高的加速比。存储器优化后的 GPU 吞吐量最高达到 10.05Gbit/s，加速比最高达到 25。最后，总结影响并行 AES 加密算法在 GPU 上执行效率的主要因素，本课题在保证了运算速度的基础上实现了资源和速度的均衡，加速比和执行效率都取得了理想的效果。本文的研究方法对于在 CUDA 并行平台上实现密码算法具有一定的借鉴意义。

关键字：GPU，CUDA，密码算法，AES

Abstract

With the popularization and development of computer networks, information security as an emerging discipline draws more and more attention. Limited by the CPU serial system, the traditional serial cryptographic algorithm has been increasingly unable to meet the demand of operation speed and system security. AES accelerated encryption based on FPGA hardware. Its function is relatively simple, computational efficiency is low, and communication protocols also require additional hardware resources. The lack of this approach is that the cost increases and the performance is not satisfactory. With the continuous development of GPU technology, computing power and band-width of the GPU has significantly improved. In recent years, accelerating compute-intensive applications using the GPU has become one of research focuses. The CUDA programming model makes GPU-based application development more flexible, more efficient. It is in such a background, this thesis select "research and design of AES algorithm based on CUDA parallel architecture" as a research topic.

This paper has discussed the unique hardware and software architecture of CUDA on GPU parallel programming platform detailedly.This thesis have analyse the traditional thinking of the AES algorithm, researched the work mode of the block cipher, and builded the model of the AES parallel encryption algorithm on the basis of the full study of the standard encryption algorithm AES principle. CUDA –based optimization techniques is discussed and analysed and CUDA programming platform is tested using large-scale matrix multiplication. The experimental results show that the GPU has a powerful floating-point capability.

The AES algorithm proposed in this paper optimizes the operation of the round function using a lookup table, simplifies numbers of implementation steps of the encryption round, selects the appropriate work mode of block cipher, in-depth analyses and compares the parallel partitioning of three different particle, improves parallel AES encryption algorithm based on the CUDA platform, completes host-side and equipment-side programming, and analyses and tests parallel AES encryption algorithm and the original CPU serial AES algorithm. The experimental results show that the ratio between the GPU throughput and CPU throughput is up to 17 , between AES algorithm implemented based on the CUDA parallel architecture and the serial

AES algorithm is up to the highest speedup 25, and GPU throughput is up to 10.05G bit/s after memory being optimized. Finally, we summarize the main factors affecting the execution efficiency of parallel AES encryption algorithm on the GPU, this project ensures the balance of speed and resources .The speedup and efficiency of implementation have achieved desired results. The research method has certain reference significance for cryptographic algorithm on the CUDA parallel platform.

Keywords: GPU,CUDA, encryption algorithm, AES

目录

第 1 章 绪论

传统 CPU 受于其集成电路元器件及功耗等限制，目前是很难提升运算能力的。并行计算已成为突破摩尔定理局限性的重要研究方向[1]。于是，人们逐渐将目光转向了拥有超强浮点运算能力的 GPU(Graphic Processing Unit,图形处理器)上。如今，使用 GPU 解决计算规模大、运算量大的复杂问题已经成为一种主流发展趋势。

在性能、成本、功耗等方面 GPU 较传统 CPU 解决方法都有着显著的优势[2]，因此关于 GPU 的研究及发展已经引起了国内外许多公司及研究机构的普遍关注和重视，目前 GPU 技术已经在很多科学领域得到了广泛应用，并取得了一些骄人的成绩[3]。

1.1 课题研究背景

1.1.1 国内外研究现状、水平和发展趋势

由于计算机技术及网络通信技术研究与应用的迅速发展，由网络通信而带来的网络信息安全问题引起了人们的极大关注[4]。信息安全的有效措施是在电子信息存储、处理、传送以及交换过程中实施加密保护。数据加密标准 DES 于 1977 年向社会公布，它是第一个世界公认的实用分组密码算法标准。但经过 20 年的应用，DES 已被认为不可靠。3DES 作为 DES 的替代，密钥长度为 168bits，可克服穷举攻击问题。同时，3DES 的底层加密算法对密码分析攻击有很强的免疫力。但由于用软件实现该算法的速度慢，使得 3DES 不能成为长期使用的加密算法标准[5]。由于 AES 具有较高的安全性能及实现效率和密钥灵活性，2001 年高级加密标准 AES 替代了 3DES。

目前，AES 已经成为最流行的密码算法，AES 算法主要用于基于私钥数据加密算法（对称密钥加密算法）的各种信息安全技术和安全产品，比如在无线网络，电子商务，音频、视频加密，数据库加密及射频 IC 芯片加密等方面的应用。针对网络通信所面临的安全威胁，密码协议 SSL 协议受到推崇，国内外很多知名企业使用 SSL 协议来保重信息的安全传输。研究表明，非硬件平台实现 SSL 协议的安全传输中，70%的时间是用于处理密码运算[6]，由此可见密码算法的运算速度已经成为影响 SSL 协议是否高效的关键因素。

作为显卡芯片制造行业的领军者，NVIDIA（英伟达）率先在 2007 年 6 月发布了统一设备计算架构 CUDA(Compute Unified Device Architecture) 1.0，在短短

不到 5 年的时间里已经推出了 CUDA 4.0 版。国外很多的企业和研究所在 GPU 理论研究的深度和高度方面领先于中国，毕竟 CUDA 是门新技术，面向公众的时间比较短，因此国内在 CUDA 应用技术方面与国外水平相差并不大。国外很多研究人员对 CUDA 架构研究的常用方法是，首先对 GPU 进行小规模的安装，随后将其移植到基于 CPU 的大规模系统平台。国内的研究机构和企业比较热衷于对 GPU 异构并行架构的研究，而且有越做越大的趋势。2010 年 10 月中国国防科技大学研制的"天河一号 A"正式对外公布，该系统采用了 7168 颗 NVIDIA (英伟达)TeslaM2050 GPU 以及 14,336 颗 CPU[7]，该计算机性能高达 2.507Petaflops (千万亿次)，在 2010 年第 36 届全球超级计算机五百强排行中成为当今中国乃至全世界最快的超级计算机，这对于 GPU 并行计算可以说是一座里程碑[8]。"天河一号 A"和第 36 届全球计算机五百强排行第三的"曙光星云"正是在 NVIDIA 公司的大力协作下，采用其 Tesla 系列产品运用 CUDA 并行架构获得了举世瞩目的成就。NVIDIA 公司加速推动了中国超级计算机产业的发展。目前，全世界范围内有 150 多所大学开设了 CUDA 并行编程与及应用课程[9]，这 150 多所院校包括美国哈佛大学，加州大学伯克利分校，清华大学等。国内外对 CUDA 并行架构的研究热度很高[10]，但现在面临的一个比较尴尬的问题是 CUDA 方面的资料相对匮乏，介绍这门技术的书籍也少之又少。它毕竟是一种新生技术，人们对这一技术的广泛开发及应用还是比较缺乏的。

同时计算机网络已经覆盖了人类生活的每个角落，网络的信息安全和隐私倍受人们关注。俄罗斯的安全软件开发商 Elcomesoft 表示，他们正在借助 NVIDIA 的新一代显卡来完成这一穷举搜索的破解过程。加密解密软件网上有很多，但利用 CUDA 并行架构来加解密的产品国内暂时还没有。传统的数据加密和解密是一项计算密集型的任务，因为它占用了大量的计算和通信资源，并且计算和通信的活动都比较缓慢，因此，人们需求的是更快、更安全的密码算法。目前高级加密标准 AES 算法主要用于私钥数据加密算法的各种安全产品和信息安全技术中，为数据加密应用提供了更强更有利的安全保障。国内外在基于 GPU 高速实现密码算法这一领域的研究早已开始，2003 年，AES 算法首次基于 GPU 的实现是由 D.cook 等人提出[11]，使用图像管道成像子集实现 AES 查找[12]。在 2007 年和 2010 年 IEEE 通信与信号处理国际会议上，有些专家学者已经讨论并分析了基于 CUDA 架构密码算法实现的可行性及高效性。我国中国科学院过程研究所，江南计算机研究所，香港浸会大学，电子科技大学以及解放军信息工程大学等研究机构和高校也在进行基于 CUDA 架构密码算法的研究。

1.1.2 课题研究意义

基于 CUDA 的密码算法的并行计算方面的研究有如下理论成就以及实践意义：

1. 高级加密标准 AES 作为当今密码学的代表之一，拥有非常广阔的发展前景[13]，又因其能适应多种环境、高效、方便等优点，研究 AES 算法有着很好的现实意义。

2.随着计算机运算能力的不断提高，人们对于信息的安全性及加密标准的要求也变得越来越高。人们对计算机的依赖性愈强，因此计算机信息的保密问题就显得愈重要，人们在享受信息资源所带来的巨大的利益的同时，也面临着信息安全的严峻考验[14]。信息安全领域的核心是密码技术[15]，而密码技术领域占据着举足轻重的地位的是加密解密算法。当今比较流行的加解密算法有 AES，RSA，MD-5，SHA-1，ECC 等。

3.在过去十年，企业，政府，医疗等关键领域对互联网络的依赖性越来越高，造成对高效的加密解密解决方案的需求也在不断增长。受 CPU 串行体制的限制，传统的串行 AES 算法的运行速度并不理想。使用 GPU 实现对称密码的实验是由 D. Cook 等提出的，他们的实验基于 OpenGL 使用图形处理流水线对 AES 加密标准进行映射，由于硬件及软件编程环境的限制，系统吞吐量并不尽人意。基于 FPGA 可编程的逻辑器件实现的 AES 加速加密器，虽然获得了 1.02GB/s 的吞吐量，但是其开发难度大、功能比较单一、硬件升级比较麻烦还需要更改代码，还需要为 FPGA 编写额外的驱动程序，实现通信协议需要额外的硬件资源[16]。这样就造成实验的成本增加，性能与成本比比价高。随着 GPU 技术的不断发展，GPU 的计算能力和带宽已经获得显著提高，CUDA 编程模型的出现使得基于 GPU 的应用开发更加灵活，实现更加高效[17]。正是在这样背景下，本文选择“基于 CUDA 并行架构 AES 算法的研究与实现”作为研究课题。

1.2 论文的研究目标及主要工作

1.2.1 研究目标

通过本课题的研究：结合 CUDA 特有的硬件架构，研究 CUDA 的编程模型、存储器模型以及高级加密标准 AES 算法。优化 AES 加密算法，并将优化过的算法移植到 CPU 与 GPU 的异构计算模式上。再根据 CUDA 编程规则对移植后的算法进行存储器访问优化，使其获得更高的加速比和运行效率，最终通过实验进行课题研究验证。

1.2.2 研究内容

1.系统地研究 CUDA 硬件系统架构，SP 流处理器结构，线程层次结构及存储器模型，CUDA 编程技术。

2.配置统一计算环境架构，通过主机端与设备端之间带宽传输速度的实验测试通过 CUDA 并行开发环境。

3.研究基于 CUDA 的一般科学计算方法及相关优化技术，并对所研究的算法进行优化及实现。

4.研究高级数据加密标准 AES，分析其串行程序，并进行相应优化。

5.移植 AES 算法到 CUDA 架构并对程序进行系统优化及架构优化，测试算法的运算性能及吞吐量。分析实验结果，比较 CPU 与 GPU 运行速度、加速比、吞吐量和吞吐量比等，并根据 CUDA 硬件架构进行相应优化。

1.3 论文重难点

本课题将重点从三个方面展开论述：深入理解 CUDA 异构并行的硬件架构和软件编程技术；分析 CUDA 架构的优化技术，并通过测试通用算法体现 GPU 的强大浮点运算能力，根据优化技术对通用算法进行优化；研究 AES 算法，并对 AES 加密算法进行改进，在并行化过程中要考虑到任务的合理划分与调度方式，考虑通信机制、线程层次架构及存储器分配方案。

本课题的难点在于根据 CUDA 的硬件架构特性和编程环境对任务进行合理的粒度划分，尽量避免主机端与设备端进行数据的频繁交换，对存储器分配方案进行优化，使其获得理想的运算时间和吞吐量。

1.4 论文研究成果及创新

1.CUDA 是并行计算的一个新发展方向，出现的时间不长，对其研究并实现一些算法本身就具有创新性。从目前掌握的资料来看，在 CUDA 上实现加解密算法的文献并不多，技术也还不够成熟，从这点来看，本研究也具有一定的创新性。

2.本课题采用一次查找表方法优化了 AES 算法，减少了轮函数的循环操作；基于 CUDA 架构 AES 优化算法的实现采用了三种不同的粒度划分策略，并分析其优劣性，选择适合 CUDA 平台运行的划分方案；根据 CUDA 架构的线程结构设计本课题所需的线程层次方案，为 AES 优化算法涉及到的数据分配存储器。

3.从算法实现性能的高效性出发，合理运用 CUDA 特有的硬件架构及编程环境进行优化，主要采用存储器优化方案对算法进行优化，进而提高本课题的执行效率。

1.5 论文的结构

本文的章节安排如下：

第 1 章绪论详细阐述了课题的研究意义及背景，分析 CUDA 国内外发展现状和基于 CUDA 密码算法的相关研究方法。

第 2 章讨论了 CUDA 并行架构，编程模型，存储器模型，硬件架构。

第 3 章详细分析了 AES 算法传统的实现技术研究。首先介绍了 AES 算法的数学基础，着重讨论了 AES 算法的理论基础和具体实现思路。

第 4 章主要分析和研究了基于 CUDA 架构的优化策略。并针对常用的大数据量矩阵乘法进行了优化和实现。

第 5 章深入分析基于 CUDA 架构 AES 算法的研究。将 AES 算法移植到 CUDA 平台上，并根据优化策略对其进行优化。对实现结果进行对比分析验证了本文研究方法的可行性与解决问题的有效性。

第 6 章对论文工作进行了总结与展望，并指出今后研究的方向及重点。

第 2 章 CUDA 编程模型及体系架构

CPU 主要专注于数据高速缓存(cache)和流处理(flow control)，而 GPU 更多的专注于计算密集型和高度并行的计算。尽管 GPU 的运行频率低于 CPU，但是 GPU 凭着更多的执行单元数量从而获得浮点计算能力上的较大优势。GPU 设计者设计更多的晶体管用做执行单元，通过更多的并行处理单元和更多的存储器控制单元来提高存储器带宽和运算能力。

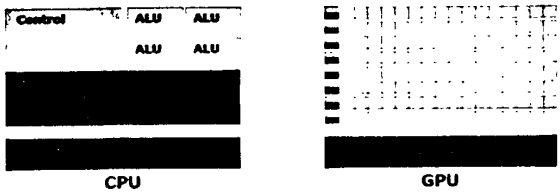

图 2.1 CPU 与 GPU 晶体管的使用数量对比

由于受市场需求的牵引[18]，GPU 的性能提升速度非常的快，完全颠覆了传统 CPU 所遵循的摩尔定律的模式，不再是没 18-24 个月性能提升一倍，最近几年 GPU 的运算性能平均不到一年就会翻倍。相应主流 GPU 的单精度浮点运算能力达到同期主流 CPU 运算能力的 10 倍左右，同时存储器带宽则能达到相应 CPU 存储器带宽的 5 倍，但是其制作成本却要低于相对应的 CPU 制作成本。

2010 年 7 月 NVIDIA 推出了第二代 CUDA 架构，代号为“Fermi” （费米）。Fermi 是有史以来最先进的 GPU 计算架构。Fermi 架构至少拥有 30 亿个晶体管，最多能配置 512 个 CUDA 核心处理器，它的计算能力赶超超级计算机。Fermi 架构的显卡芯片成本仅为基于 CPU 的大型服务器的十分之一，功耗也仅为其的二十分之一。加州大学伯克利分校并行计算研究实验室主任 Dave Patterson 教授盛赞 Fermi 架构将带领 GPU 产业进入更广的领域[19]，坚信 Fermi 会成为 PC 发展史上重要的里程碑。

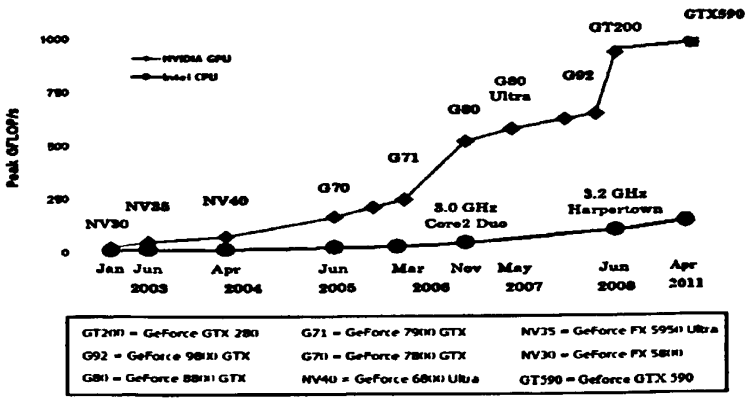

图 2.2 GPU 与 CPU 浮点计算能力的对比

随着科技的不断发展，计算行业变得尤为重要。目前，计算行业正从传统的CPU“中央处理”向CPU与GPU“协同处理”发展[20]。为打造这一全新的计算典范，NVIDIA®（英伟达™）推出了一种革命性的集成技术CUDA（Compute Unified Device Architecture，统一计算设备架构）。这一创新技术颠覆了传统并行计算的形式，充分结合并利用了CPU和GPU各自的优点，尤其是能最大程度地发挥GPU强大的浮点计算能力。需要指出的是，CUDA并不是一种纯粹的编程语言，而是基于GPU通用计算的类似C语言的开发环境[21]，它是一个全新的软硬件架构平台。

2.1 CUDA编程模型

2.1.1 主机与设备及Kernel函数

CUDA程序的执行是由CPU与GPU协同完成的，CPU作为主机(host)端，主要处理逻辑性较强的事务及串行程序的计算。GPU扮演设备(device)端或者协处理器（co_processor）的角色，主要工作是负责密集型大数据的并行计算。可以用公式表示：一个完整的CUDA程序[22]=CPU串行处理 +GPU kernel函数并行处理。

一个CUDA架构下的程序分为主机host端和设备device端两个部分。一般情况下CUDA程序的执行流程如图2.3：

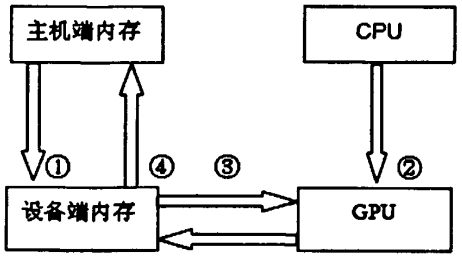

图2.3 cuda程序的处理流程

① 主机端程序先在host分配相应存储器，然后把需要运算的数据传输到显存中。

② 串行程序的运行。

③ 执行GPU核心中的并行计算。

④ 为运算结果分配存储器并把结果从设备端内存传输到主机端内存。

在GPU上运行的程序被称为kernel内核函数。内核函数使用_global_声明符定义，用<<<....>>>执行配置语法执行某一指定内核调用的线程数[23]。每个执行

内核的线程 thread 都有个唯一的索引 threadI.x.x。

以下一段代码为简单的内核函数，所表达的意思是让两个长度为 N 的向量 A 和向量 B 进行加操作，然后把计算的结果存储在向量 C。

```
// kernel 定义
_global_ void VecAdd(float A*,float B*,float C*)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
//主机程序
int main( )
{  ...
//  kernel 函数对 N 个线程的调用
    VecAdd<<< 1,N >>>(A,B,C);
// VecAdd<<< 1,N >>>(A,B,C) 表示 N 个线程中每个都执行一次向量加法。
}
```

2.1.2 线程层次

CUDA 中使用了 dim3[24]类型的内建变量 threadIdx 和 blockIdx，这样方便使用一维、二维及三维的索引来表示和查找线程 thread，同时也构建了一维、二维及三维的线程块[3]，使得程序员对各种域（向量，矩阵，空间）中数据划分更直观，自然，查找更方便，快捷。

1.一维的 block，线程的索引就是 threadIdx.x。

2.大小为(Dx,Dy)的二维 block，线程的索引为 threadIdx.x+ threadIdx.y * Dx。

3.大小为 (Dx,Dy,Dz) 的三维 block，线程的索引为 threadIdx.x+ threadIdx.y * Dx + threadIdx.z *Dx*Dy。

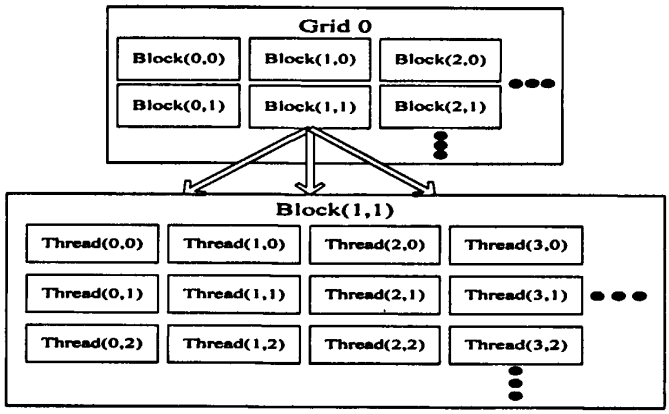

图 2.4 CUDA 的线程组织示意图

从上图 2.4 可以看出，线程[3]是 CUDA 线程层次中的最小单位。一个内核函数以线程网格进行组织，每个网格由多个线程块组成，同时每个线程块又由多个线程组成。因此明显得知，内核函数中存在着两个不同层次的并行，同一个网格内的所有线程块之间是并行的，一个线程块中的所有线程之间也是并行的。并且同一线程块中的线程通过快速共享的片上存储器共享数据,还可以通过同步操作_syncthreads()实现线程间的同步。程序员在 CUDA 架构上通过内存空间可以访问设备的 DRAM 和片上内存。这样的设计是创新的，打破了传统 GPGPU 不能实现线程间通信的规则。

2.1.3 异构编程

CUDA 线程在物理上是独立的 device 设备端上运行，主机运行 C 程序，device 作为协处理器，内核函数在 GPU 上运行。主机端和设备端[3]都拥有自己独立的 DRAM(Dynamic Random Access Memory)动态随机存储存储器空间，主机端被称为主机存储器空间，设备端被称为设备存储器空间。

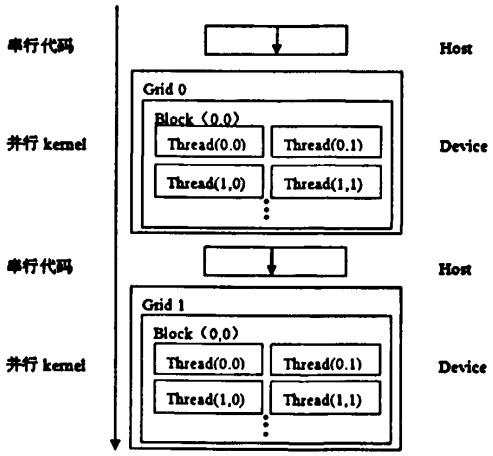

图 2.5 CUDA 编程模型

CUDA runtime API 提供了实现设备管理的一些应用程序接口，并且在CUDA driver API 的基础上进行了封装[25]，隐匿了一些实现细节，使得代码的编写更加方便，快捷。设备存储器的分配及释放，主机与设备间的数据传输和全局、常量和纹理存储器空间的管理都可以通过调用 CUDA runtime 来实现(参看 2.3 章节)。

2.1.4 计算能力

Device 的技术规范及特性取决于计算能力[26]，计算能力高的具有计算能力低得所有特性。

Device 的计算能力由主修订号和次版本号组成。主修订号相同的设备基于相同核心架构的。本论文所用的 NVIDIA GeForce 9800 GT 的主修定号为 1。"Fermi" 是 CUDA 推出的第二代架构，其主修订号为 2。次修订号对应着对核心架构的增量提升，也可能包含着新特性。

2.2 CUDA 软件体系

CUDA 的核心是 CUDA C 语言，CUDA C 为那些熟悉 C 语言的用户提供了一个简便途径[27]，让他们能轻易地写出在 device 端能在设备上执行的程序。CUDA 的软件堆栈由 CUDA Library、CUDA runtime API、CUDA device API 三个层次构成，如下图。

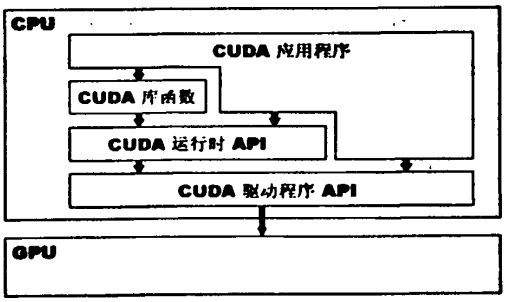

图 2.6 CUDA 软件体系

2.2.1 CUDA C 语言

CUDA C 语言并不是传统的 C 语言，而是在 C 语言的基础上做了相应扩展的一种编程规则。扩展主要包括以下四个方面:

1.函数类型限定符。用来限定引入的函数是在主机端 host 还是在设备端 device 执行，同时还规定了所引入的函数是从主机端进行调用还是从设备端进行

调用。这些限定符是:_device_、_host_和_global_。

2.变量类型限定符。用来规定程序中的变量分配的存储器类型。运行在 CPU 上的传统串行程序，编译器能自动识别并将变量存储在 CPU 的寄存器中或者是主机内存中。CUDA 编程模型就不同了，CUDA 程序不仅要使用主机端的内存，也要使用设备端的显存，总共有八种不同的存储器供 CUDA 编程选择。因此，在 CUDA 编程模型中，为了更好的区分各种存储器，在声明存储器类型的时候需要加上限定符。这些限定符是_device_，_shared_和_constant_。值得注意的是，变量类型限定符_device_和函数类型限定符 _device_表达的含义不同。

3.执行配置。执行配置是在函数名称和用括号括起来的参数之间插入<<<...>>>运算符，用来传递内核函数 kernel 执行时的线程层次设计[28]，包括线程网格和线程块的维度信息。

4.内置变量。用于对不同的线程块和线程进行定位索引，还用来描述网格和线程块的维度设计。

CUDA 还引入了不少函数，比如数学函数、同步函数、纹理函数、测时函数及原子函数等。如果不能按照以上限定进行编写程序，nvcc 会出现警告信息或者报错，甚至使得程序无法运行。

2.2.2 CUDA 函数库

目前 CUDA 的函数库有：CUFFT 以及 CUBLAS[29]。CUFFT(CUDA Fast Fourier Transform，CUDA 快速傅里叶变换)库是一个通过 GPU 进行傅立叶变换的函数库，提供了与 FFTW（the Faster Fourier Transform in the West，快速计算离散傅里叶变换的标准 C 语言程序集）函数库相似的接口，但 CUFFT 操作的数据存储在设备存储器中，不能直接代替 FFTW 操作，需要加入设备存储器与主机存储器之间的数据交换，进行封装后才能代替 FFTW 库。CUBLAS(CUDA Basic Linear Algebra Subprograms)库是一个基本的矩阵与向量的运算库，它是在 CUDA 硬件驱动上实现的 BLAS API，它提供的接口与 BLAS 相似，能用于简单的矩阵计算，也可以作为基础函数包构造复杂度更高的函数包。CUBLAS 操作的数据是存储在设备存储器中的，同样需要封装完毕来替代 BLAS 中的函数。

2.3 CUDA 存储器模型

CUDA 拥有一套特有的存储器模型，并且将存储空间分成多个层次，使得线程执行的时候能访问多个存储器上数据。实际应用中常用到以下几种存储器：

GPU 片内：寄存器(register)，共享存储器(share memory)；

板载显存：局部存储器(local memory)，常量存储器(constant memory)。纹理存储器 (texture memory)， 全局存储器(global memory)；

主机端内存： 主存（host memory），页锁定内存（pinned memory）；

各存储器间的层次关系[3]如图 2.7 所示：

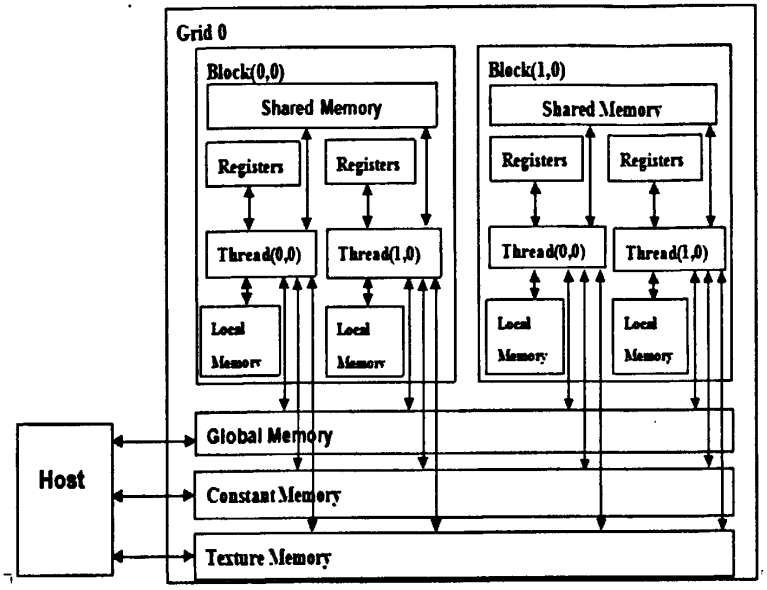

图 2.7 CUDA 存储器层次示意图

由图 2.7 可知，网格所包含的线程都能访问全局存储器上的数据。只读的常数存储器和纹理存储器能被所有的线程访问。每个线程独立拥有寄存器和全局存储器。每个线程块也有自己的共享存储器。全局存储器、常数存储器和纹理存储器还能被主机端内核函数调用。

2.3.1 寄存器

寄存器 register 是位于 GPU 芯片上的高速存储器。跟 CPU 一样，寄存器对每个线程来说都是私有的。寄存器的基本单元是寄存器文件[30]，每个寄存器文件大小为 32 字节。寄存器拥有很高的带宽，每个指令访问一个寄存器是零附加时钟周期。寄存器文件的数量由显卡的计算能力决定，本文使用的显卡计算能力为 1.1，每个 SM（Stream Multiprocess，流多处理器）中有 8192 个寄存器文件，即 8KB。在 GPU 计算能力 1.x 的设备中，每个 thread 可分配的寄存器数量的计算公式为：$\frac{R}{B * ceil(T,32)}$ 表达式中 R 代表流多处理器的寄存器总数，B 是每个流多处理器的活动线程块的数量，T 是每个线程块所拥有的线程数，ceil(T, 32) 是指将 T/32 的结果四舍五入得到的最大正整数。由于并行执行的线程数量比较多，所以平均分给每个线程的寄存器是比较有限的，内核设计时不能为单个线程分配

过多的私有变量。寄存器一旦被使用完，数据将会被存储到局部存储器中。

2.3.2 全局存储器

全局存储器 global memory 位于普通显存中，可用的全局存储器的大小与一个显卡的显存大小相当。全局存储器与执行单元流处理器 SP（Stream Processor）相距最远，所以其访问延迟最长，延迟最长时可达寄存器访问延迟的上百倍。由图 2.7 可知，全局存储器能被网格中的所有线程访问，即可以从主机端访问，也可以从设备端访问。由于目前硬件架构的全局存储器没有缓存机制，所以读取和存储就必须进行对齐操作。如果没有进行合理对齐，读写会被编译器分成多次顺序操作，这样就得不到相对高效的访问效率。因此在程序设计时要合理有效地利用全局存储器带宽，尽量避免读写冲突。

2.3.3 共享存储器

由于共享存储器 shared memory 位于 GPU 芯片上，因而共享存储器比寄存器和全局存储器的速度都要快得多。如图 2.7 所示，共享存储器可以被同一线程块内的所有线程进行读写操作，只要线程之间没有冲突，访问共享存储器与访问寄存器的速度一样快[31]。在 CUDA 的编程模型中，共享存储器可以使得线程间的数据通信延迟降到最低。因此在 CUDA 存储器模型中，共享存储器的地位比较重要。

在数据实际存储的时候，显存中读出的数据不会写到共享存储器而是被写到寄存器中。由于简化设计，CUDA 存储器架构里共享存储器和显存不是直接连接起来的。如果想把显卡内存的数据存储到共享存储器中，首先把显存中的数据写到寄存器，然后由寄存器转存到共享存储器中。实际设计中共享存储器能够用于保存共用的计数器或者线程块的公用中间结果。显卡芯片 Geforce 9800GT 中每个流多处理器的共享存储器大小为 16KB。因此，程序设计时需认真规划资源配置[32]。

2.3.4 纹理存储器

纹理存储器 texture memory 是只读的存储器，它并不是一块专用于存储的存储器，纹理存储器是实现 GPU 纹理渲染功能的图形专用单元。纹理存储器中的数据是以数组的形式存储在显存中，比如一维数组，二维数组甚至三维数组。纹理存储器具有缓存机制，纹理缓存可以很好地节省带宽和功耗。在通用计算中，纹理存储器比较适用于存储查找表，进行图像处理或者对大数据量的随机访问等操作，同时对非对齐的访问也有很好地加速效果。

2.3.5 局部存储器和常量存储器

局部存储器 local memory 位于 GPU 芯片之外的显存中。对于每个线程，l 局部存储器跟寄存器一样是私有的。事实上，局部存储器只是寄存器的备用，仅当寄存器不够使用时才会被用到。当寄存器用完时，如果还有未存储的数据，此时数据会被分配到局部存储器。局部存储器中的数据不是存储在寄存器或者缓存中，而是保存在显存中的，因此对局部存储器的访问速度很慢。所以程序设计的前期最好对线程私有变量的大小进行预算，避免因寄存器大小不够而使用局部存储器所造成延时。

常量存储器 constant memory 是位于显存中的只读地址空间。常量存储器空间仅有 64KB，每个流多处理器平均有 8KB 的常量存储器缓存，通常用于存储那些需要进行频繁访问的只读参数，比如系数矩阵，权重数组等。常量存储器拥有缓存机制，因此对常量存储器的访问可以获得缓存加速，可以加快访问速度从而节约带宽[33]。

上面介绍的六种存储器都是设备端存储器，当然还有主机端的存储器，比如内存。表 2.1 显示了设备上各存储器所处的位置、访问权限、访问延迟以及读写特性等参数。

表 2.1 各种存储器属性比较

存储器	位置	访问权限	大小	缓存	变量生存周期
寄存器	GPU 芯片	device 可读/写	32bit	有	和线程相同
本地内存	显存	device 可读/写	up to global	无	和线程相同
共享存储器	GPU 芯片	device 可读/写	16KB	有	和线程块相同
常量存储器	显存	device 可读 host 可读/写	64KB	有	能在程序中保持
纹理存储器	显存	device 可读 host 可读/写	up to global	有	能在程序中保持
全局存储器	显存	可读/写	768MB	无	能在程序中保持
可分页内存	主机端内存	host 可读/写	/	无	能在程序中保持
页锁定内存	主机端内存	host 可读/写	/	无	能在程序中保持

CUDA 存储器架构中，主机端的内存分为两种:页锁定内存（page-locked memory 或 pinned)和可分页内存（pageable memory)。其中可分页内存是通过操作系统 API 分配的存储空间。页锁定内存一般存在于物理内存中，很难被分配到低速的虚拟内存中。

2.4 CUDA 硬件架构

CUDA 硬件架构是围绕一个可扩展的多线程流多处理器阵列构建的。当主机上的 CUDA 程序调用内核网络，网格内线程块枚举并分发到有可用执行资源的流多处理器上。线程内线程在一个流多处理器上并行执行且可在一个流多处理器上执行。线程块终止时，便在空闲的流多处理器上发射新的线程块。

流多处理器设计是为了能同时并行执行上百个线程，为了管理如此多的线程，流多处理器采用了一种被称为 SIMT（单指令流，多线程流）的独一无二的架构。

2.4.1 SIMT 架构

SIMT 架构类似于 SIMD[34]（单指令流，多数据流）向量组织方法，共同之处是使用单指令来控制多个处理元素。一项主要差别在于 SIMD 向量组织方法会向软件公开 SIMD 宽度，而 SIMT 指令指定单一线程的执行和分支行为。与 SIMD 向量机不同，SIMT 允许程序员为独立、标量线程编写线程级别的并行代码，还允许为协同线程编写数据并行代码。为了确保正确性，程序员可忽略 SIMT 行为，但通过维护很少需要使一个线程束内的线程分支的代码，即可实现显著的性能提升。

另外一个重要不同是 SIMD 中的向量中的元素相互之间可以自由通信，因为它们存在于相同的地址空间（例如，都在 CPU 的同一寄存器中），而 SIMT 中的线程束中每个线程的寄存器都是私有的，线程之间只能通过共享存储器和同步机制进行通信。

2.4.2 流多处理器

在 GPU 中流多处理器才算是真正的完整核心，每个流多处理器又包含 8 个流处理器单元。目前的 CUDA 中各流多处理器必须执行相同的程序，因此这些流多处理器以 SPMD(单程序流,多数据流)的方式工作，而每个流多处理器则是一个 SIMT 处理器，这种架构在可编程性和灵活性与硬件的复杂度和功耗之间取得了很好的折中：相互之间不能通信的粗粒度并行线程块可以被分发到 SPMD

的各个流多处理器上，而每个线程块内的细粒度协作线程数量也因为 SIMT 的自动向量化[35]而可以灵活的调整。

流多处理器拥有独立的完整前端。流多处理器以 32 个为一组创建、管理、调度和执行并行的线程，这 32 个线程被称为 warp(线程束)。线程束内包含的线程从统一程序地址开始，但他们有自己的指令地址计数器和寄存器状态，因此可以自由分支和独立执行。线程束线程束每次执行的指令是相同的，如果线程束中 32 个线程在同一路径上执行时能达到最高效率。

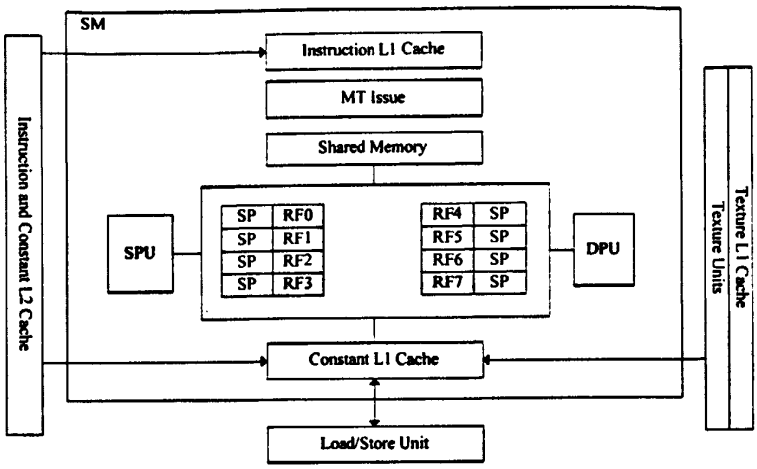

图 2.8 GT200 SM 架构

图 2.8 中，SP 是 Stream Processing Units 流处理单元；MT Issue 是 Multi-threaded Instruction Fetch and issue Unit 多线程取指和发射单元[3]；RF 是 Register File 寄存器文件；SFU 是 Super Function Unit 特殊函数执行单元，殊性就在于 SFU 执行正弦、余弦、平方根、倒数及插值一些运算。DFU 是 Double Precision Unit 双精度处理单元，主要用来处理 Register 中 64 位整型和浮点操作数的 64bit 乘加单元。

2.4.3 GeForce 9800 GT 硬件架构

虽然 PCI-Express 总线已经取代 PCI 和 AGP，但 PCI-E 总线的带宽还是无法完全应对大量高宽带并行读写的要求，PCI-E 总线成为系统提升的一个瓶颈。因此在 GPU 运行程序中要尽量减少主存与显存的多次数据传输。下表 2.2 是基于 CUDA 架构使用 testHostDeviceTransfer（）和 testDeviceToHostTransfer（）及 test Device To Device Transfer（）语句测试得到的 GPU 带宽传输速度。比较数据传输过程中的速度差异。

2008年2月，NVIDIA公司发布了GeForce 9800 GT。本实验平台采用的是GeForce9800 GT的显卡芯片。通过实验平台测试，测试的具体硬件属性见图2.9。

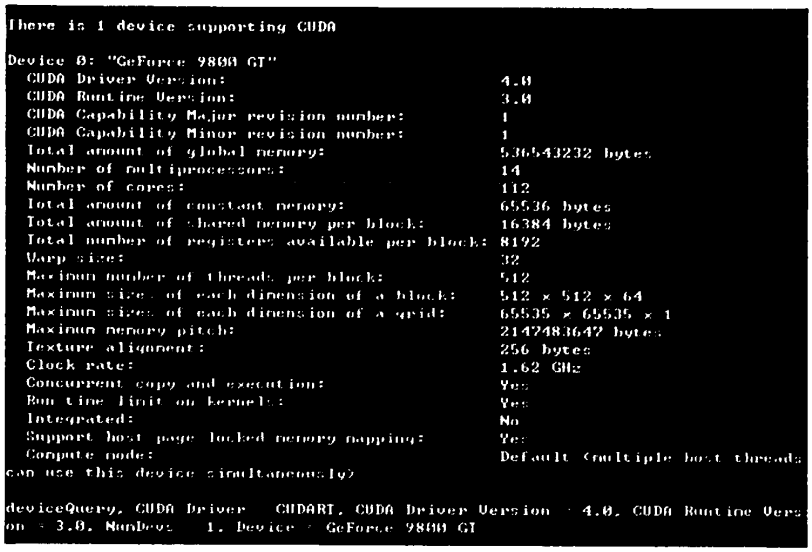

图2.9 Geforce 9800GT 硬件指标

图2.9显示GeForce 9800 GT支持CUDA计算架构，计算能力为1.1，流多处理器及流处理器个数，线程束大小，线程层次组织等显卡芯片属性。

表2.2 GPU带宽传输速度

GPU(单位GB/S)	HOST_TO_DEVICE	DEVICE_TO_HOST	DEVICE_TO_DEVICE
9600GT	1.358	1.223	36.094
9800GT	1.707	1.527	56.032
FX 460	1.824	1.743	47.657
Tesla C1060	2.488	1.971	73.453

表2.2显示的是四种不同的显卡芯片的GPU带宽传输速度，明显可以看到设备端与设备端之间的传输速度远大于主存与显存间的数据传输速度。

2.5 本章小结

本章主要讨论了CUDA的编程模型、CUDA软件体系结构、CUDA的存储器模型及硬件架构。通过本章的详细阐述可以对CUDA的软硬件架构及程序设计过程及方法有一个深入的理解，为后续章节提供强有力的理论基础。

第 3 章 AES 算法研究

随着计算机运算能力的不断提高，对加密标准也提出了更高的要求。1997 年 4 月 15 日，美国国家标准技术研究所（NIST）向全球公开征集高级加密标准（Advanced Encryption Standard，AES），旨在取代 21 世纪的数据加密标准（Data Encryption Standard，DES）。AES 算法历时 3 年的全面遴选，经过多次大会讨论分析，最终比利时的密码学专家 Vincent Rijmen 和 Joan Daemen 成为获胜者，同时结合两位作者的名字法以 Rijndael 命名[36]。高级加密标准 AES 在 2002 年 5 月 26 日成为有效的标准。AES 自从被确立为标准后已成为对称密钥加密中最流行的算法之一。

3.1 AES 数学基础

3.1.1 伽罗华域(Galois Field，GF，有限域)

有限域 $GF(2^8)$[37]是由一个 GF(2)上的 8 次既约多项式生成。$GF(2^8)$的全体元素组成一个线性空间即加法交换群。$GF(2^8)$的非零元素组成的是乘法循环群。$GF(2^8)$中元素的集中表示形式如下：

指数形式：$GF(2^8)^*=\{\alpha^0,\alpha^1 \ldots \alpha^{254}\}$；

对数形式：$GF(2^8)^*=\{0,1 \ldots 254\}$；

多项式形式：$GF(2^8)=\{a_7x^7+a_6x^6+\ldots+a_1x^1+a_0\}$；

字节表示形式：$GF(2^8)=\{a_7,a_6,\ldots a_1,a_0\}$。

3.1.2 AES 的 GF(28)表示

AES采用$GF(2^8)$的多项式形式进行表示。字节A＝$b_7b_6b_5b_4b_3b_2b_1b_0$可表示为GF(2)上的多项式：$b_7x^7+b_6x^6+b_5x^5+b_4x^4+b_3x^3+b_2x^2+b_1x^1+b_0$。

字节 65＝01100101的多项式表示：01100101 $\longleftrightarrow x^6+x^5+x^2+1$。

AES的理论基础在$GF(2^8)$，基本的运算有加法，乘法，乘法逆元和x 乘法四种。

1. 加法就是两元素多项式的系数按位进行异或即模2加操作，（⊕ 表示异或运算）。

例如：65＋87＝E2

$$\begin{array}{r} 0\ 1\ 1\ 0\ 0\ 1\ 0\ 1 \\ \oplus\ 1\ 0\ 0\ 0\ 0\ 1\ 1\ 1 \\ \hline 1\ 1\ 1\ 0\ 0\ 0\ 1\ 0 \end{array}$$

也可以用多项式表示法：

$(x^6+x^5+x^2+1)\oplus(x^7+x^2+x+1)=x^7+x^6+x^5+x$；

2. 有限域GF(2^8)中两元素的乘法：

两元素在有限域GF(2^8)中相乘的方法是首先把两个输入转化为多项式形式，然后让这两个多项式进行乘法运算，把x的次幂系数大于2的舍去，若运算所得的多项式有超过x的8次方的项，则必须对此结果对一个多项式 m(x) 进行模运算。AES 算法中定义 m(x) 多项式（不可约多项式）为：$m(x)=x^8+x^4+x^3+x+1$(十六进制的0x 011b)。同时m(x)是AES算法中30个8次不可约多项式中第一个不可约多项式（⊗表示乘法运算）。

例如：65×87=D0

$$(x^6+x^5+x^2+1)\otimes(x^7+x^2+x+1)=x^{13}+x^{12}+x^9+x^8+x^5+x^4+x^3+x+1$$

$$(x^{13}+x^{12}+x^9+x^8+x^5+x^4+x^3+x^1+1)\bmod m(x)=x^7+x^6+x^5$$

3. 乘法逆元：

如果多项式 $a(x)$ 的次数小于 $m(x)$，假设 $a(x)$ 的逆元是 $b(x)$，则 $a(x)\otimes b(x)\bmod m(x)=1$。可根据 Euclid（欧几里德）扩展算法求出输入元素的逆元。在按位异或运算⊕和乘法运算⊗下，有限域GF(2^8)是由256个字节值的集合组成。

4. x 乘法 xtime：用x乘GF(2^8)的元素。

$$xtime(65)=x(x^6+x^5+x^2+1)=x^7+x^6+x^3+x$$

$$xtime(87)=x(x^7+x^2+x+1)=(x^8+x^3+x^2+x)\bmod m(x)=x^4+x^2+1$$

若 x^7 的系数=0，则 $xtime()$ 为简单元素相乘：进行系数左移。

若 x^7 的系数=1，则取模 $m(x)$。

3.1.3 AES 字的表示与运算

AES数据处理的单位是字节和字，显然的是4个字节等于1个字。一个字表示为系数取自GF(2^8)上的次数低于4次的多项式

字：57 83 4A D1 $\longleftrightarrow$ $57x^3+83x^2+4Ax+D1$

字加法就是两多项式系数进行异或也可以说是按位模2加。

字乘法：设a和c是两个字，$a(x)$和$b(x)$是其字多项式，AES定义a和b的乘积为$c(x)=a(x)b(x)\bmod(x^4+1)$

$a(x)=a_3x^3+a_2x^2+a_1x^1+a_0$；

$b(x)=b_3x^3+b_2x^2+b_1x^1+b_0$；

$c(x)=c_3x^3+c_2x^2+c_1x^1+c_0$；

由于$c(x)=a(x)b(x)\bmod(x^4+1)$，则

$c_0=a_0b_0+a_3b_1+a_2b_2+a_1b_3$；　$c_1=a_1b_0+a_0b_1+a_3b_2+a_2b_3$；

$c_2=a_2b_0+a_1b_1+a_0b_2+a_3b_3$；　$c_3=a_3b_0+a_2b_1+a_1b_2+a_0b_3$；

写成矩阵形式:

$$\begin{bmatrix}c_0\\c_1\\c_2\\c_3\end{bmatrix}=\begin{bmatrix}a_0a_3a_2a_1\\a_1a_0a_3a_2\\a_2a_1a_0a_3\\a_3a_2a_1c_0\end{bmatrix}\begin{bmatrix}b_0\\b_1\\b_2\\b_3\end{bmatrix}$$ 公式(3.1)

注意：

1.x^4+1是可约多项式，字$b(x)$不一定有逆；

2.但在AES中选择$c(x)$固定，且有逆。

字x乘法：$d(x)=xb(x)\bmod(x^4+1)$,写成矩阵形式：

$$\begin{bmatrix}d_0\\d_1\\d_2\\d_3\end{bmatrix}=\begin{bmatrix}00&00&00&01\\01&00&00&00\\00&01&00&00\\00&00&01&00\end{bmatrix}\begin{bmatrix}b_0\\b_1\\b_2\\b_3\end{bmatrix}$$ 公式(3.2)

注意：因为模x^4+1，字x乘法相当于字节循环移位；

3.2 AES 算法过程详述

虽然高级加密标准 AES[38]算法和 Rijnadel 算法在实际应用中可以互相转换，但是严格意义来讲，这两种算法并不完全一样。AES 的密钥长度标准可以是 128bit、192bit、256bit，并且其区块长度必须是这三个固定值中的一个。AES 运算是在特别的有限域内完成。而 Rijnadel 算法使用的密钥长度和区块长度可以是

[128bit, 256bit]区间内任一 32 的整数倍。

表 3.1 循环轮数与分组长度和密钥长度之间关系

标准	密钥长度（Nk）	分组长度（Nb）	循环轮数（Nr）
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

由表 3.1 可知，不管 AES 的密钥标准是 128 或 192 或 256，分组长度都为 4，密钥长度和循环轮数随着改变，对于不同的分组长度，相应的循环轮次数也是不同的。

3.2.1 算法描述

AES 算法由以下步骤完成:1、密钥扩展(KeyExpansion)，2、初始轮(InitialRound)，3、重复轮（Rounds)，4、最终轮（FinalRound)。运算过程会出现很多轮的重复和变换。

初始状态 state,第 i 轮的密钥为 ExpandedKey[i]，KeyExpansion 表示 CipherKey 导出 expandedKey 的过程。

伪代码描述如下：

```
AES(state，cipherkey)
{
    KeyExpansion(cipherkey，expandedkey);
    InitialRound (state，expandedkey[0]);
    for(i=1;i<=Nr;i++)
    Round(state，expandedkey[i] );
    FinalRound(state，expandedkey[Nr]）;
}
```

3.2.2 重复轮

AES 重复轮的每一轮循环（除最终轮以外）均包含以下 4 个步骤，下面做简要介绍，3.3 节会给出详细的实现流程。

1.AddRoundKey（密钥加）— 矩阵中的每一个字节都与此轮的密钥（round key）进行异或（xor）运算。

2.SubBytes（字节替换变换）— 使用一个非线性的替换函数，通过查找表的方式把每个字节替换成查找表中对应的字节。将 S 盒里的每一个字节映射成它在有限域 $GF(2^8)$中的乘法逆。

3.ShiftRows（行移位变换）— 将状态矩阵 State 中各行按照不同的偏移量进行循环移位，其中偏移量是根据分组长度的不同而选择的。

4.MixColumns（列混合变换）— 对状态矩阵中的每一列进行逐个混合变换。

Round（重复轮）的伪代码描述如下：

```
Round(state，expandedkey[i])
{
    SubBytes(state);
    ShiftRows(state);
    MixColumns(state):
    AddRoundKey(state，expandedkey[i]);
}
```

加密操作中最后一个循环中没有 MixColumn，而是由 AddRoundKey 取代。

FinalRound（最终轮）的伪代码描述如下:

```
FinalRound(state，expandedkey[Nr])
{
    SubBytes(state);
    ShitfRows(state):
    AddRoundKey(state，expandedkey[Nr]);
}
```

由上面 AES 算法描述，算法实现流程表示如下图：

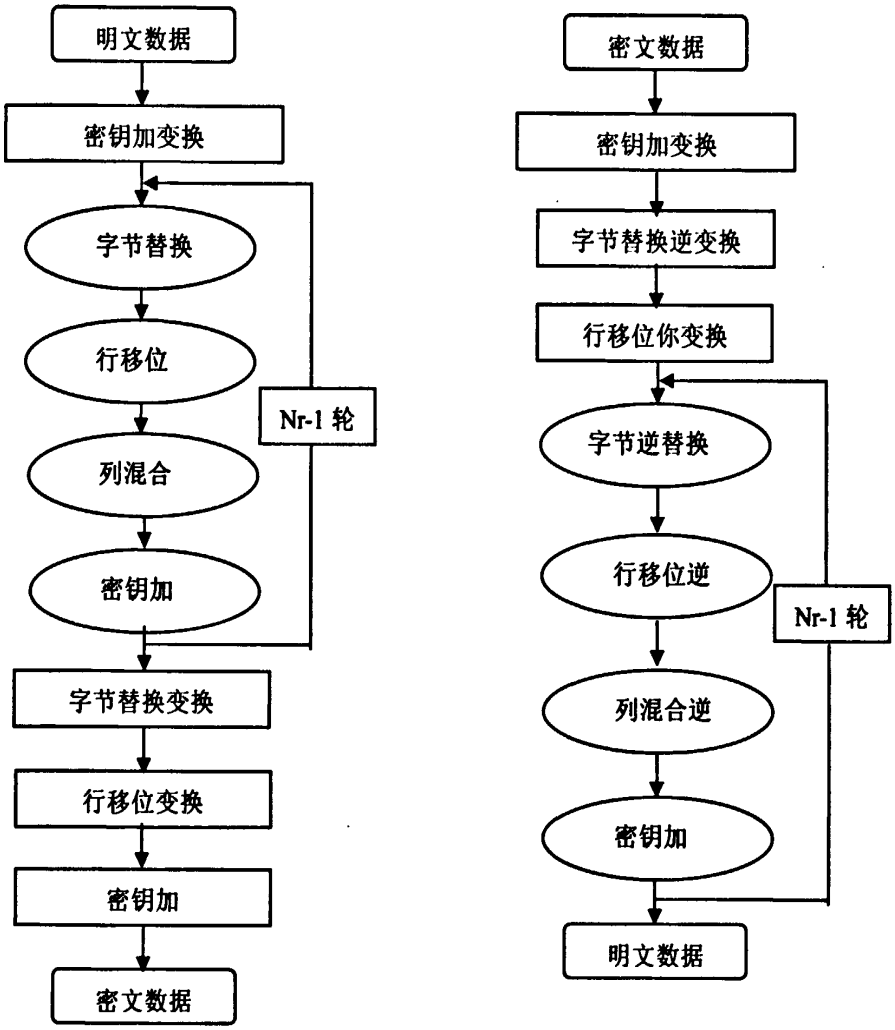

图 3.1 AES 算法的加密/解密流程图

3.3 AES 算法的实现流程

3.3.1 字节替换变换

字节替换变换是 AES 算法中唯一的非线性变换[39]，它是一个基于 S 盒把状态字节映射为新的字节的置换操作，映射方法是：将每个值作为 sbox 的下标值。S 盒的设计原理如下：

1.把每个字节都映射为乘法逆运算并进行替换。'00'的逆是自身。

2.将上一步得到的字节值进行仿射变换，写成矩阵形式表示如下：

$$\begin{bmatrix} b0' \\ b1' \\ b2' \\ b3' \\ b4' \\ b5' \\ b6' \\ b7' \end{bmatrix} = \begin{bmatrix} 10001111 \\ 11000111 \\ 11100011 \\ 11110001 \\ 11111000 \\ 01111100 \\ 00111110 \\ 00011111 \end{bmatrix} \bullet \begin{bmatrix} b0 \\ b1 \\ b2 \\ b3 \\ b4 \\ b5 \\ b6 \\ b7 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \quad \text{公式(3.3)}$$

下图是 AES 定义的一个使用十六进制表示的映射表 SBox[40]，S 盒是一个 16 * 16 字节的矩阵，包含了 8 位数值能表达的最多 256 种可能的变换。总共有 16 行（0～F）*16 列（0～F）。

```
unsigned SBox[] = {
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16
};
```

图 3.2 S 盒的数组表示

字节替换变换可以采取查找表格的形式表示。参照上图，如果 $S_{1,1}$={D5}，那么替换值便是上图中第“D”行和第“5”列共同指向的值。所以字节变换的替换值 $S'_{1,1}$ ={0x03}。

伪代码表示如下：

```
static void SubBytes(unsigned char *state)
{
    int i,j;
    for(i =0;i<16;i++)
    {
        state[i] = sbox[state[i]];
    }
}
```

3.3.2 行移位变换

行移位变换是基于行的循环移位操作，对状态中的字节进行移位变换，行移位遵循的规则是第一行保持不变，即偏移量为 0，第二行的偏移量为C_1，第三行的偏移量为C_2，第四行偏移量为C_3。偏移量与数据块长度有关系如表 3.2 所示。

表 3.2 偏移量与数据块长度的关系

偏移量 \ 数据块长度	128	192	256
C_1	1	1	1
C_2	2	2	3
C_3	3	3	4

在行移位变换中，矩阵中的每一行的各个字节循环向左方进行移位操作。偏移量则随着行数递增而递增。

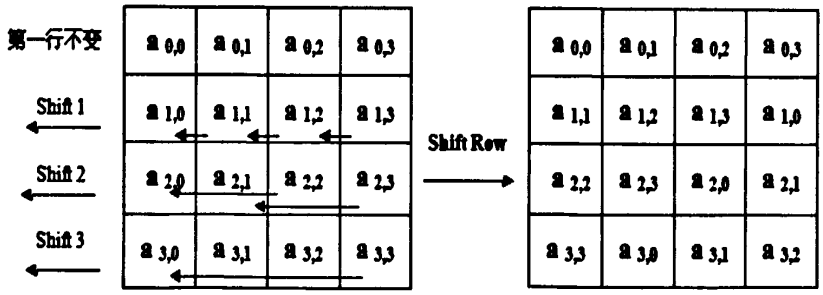

图 3.3 数据块长度为 128bit 的行移位变换

矩阵形式可表示为：

$$\begin{bmatrix} c0,j \\ c1,j \\ c2,j \\ c3,j \end{bmatrix} = \begin{bmatrix} d[a0,j] \\ d[a1,j-c_1] \\ d[a2,j-c_2] \\ d[a3,j-c_3] \end{bmatrix} \quad \text{公式(3.4)}$$

3.3.3 列混合变换

列混合变换是对状态矩阵的每一列进行独立的操作。把每一列当做有限域 GF(2^8)上的一个四项多项式，并且每列的四项当做x^0即是 1，x^1，x^2，x^3的系

数。这个四项多项式与一个固定多项式 a(x)相乘再对（x^4+1）进行取模运算。

其中 $a(x)=\{03\}x^3+\{01\}x^2+\{01\}x^1+\{02\}1$。

写成矩阵形式为：$d(x)=a(x)c(x)$，展开为

$$\begin{bmatrix} d_{0,1} \\ d_{1,1} \\ d_{2,1} \\ d_{3,1} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} c_{0,1} \\ c_{1,1} \\ c_{2,1} \\ c_{3,1} \end{bmatrix} \quad \text{公式(3.5)}$$

3.3.4 密钥加

密钥加是输入数据或者中间状态都与相应轮密钥做异或（⊕或者 xor）运算，表达式为 $a_{i,j}=b_{i,j}\oplus k_{i,j}$。可表示为：

$$\begin{bmatrix} b_{0,j} \\ b_{1,j} \\ b_{2,j} \\ b_{3,j} \end{bmatrix} = \begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix} \quad \text{公式(3.6)}$$

轮密钥用 Expandedkey [i]表示，0≤i 〈Nr,Nr 指循环轮数。轮密钥是初始密钥通过密钥生成算法生成，3.4 节有详细叙述。图 3.5 给出了密钥加变换示意图。

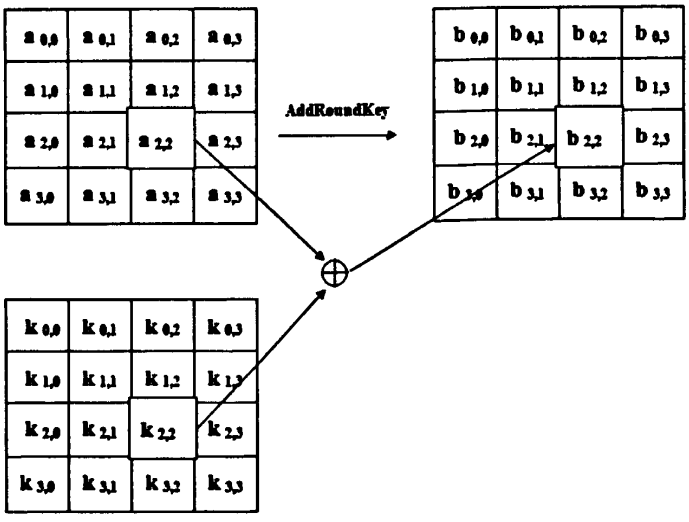

图 3.4 密钥加变换示意图

密钥加伪代码：

```
static void AddRoundKey(unsigned char *state,unsigned char *expandedkey,int Nr)
{

    int x;
    for(x=0;x<16;x++)
    {
        state[x]^=expandedkey[(Nr*16)+x];
    }
}
```

3.4 密钥生成算法

轮密钥（Expandedkey）是由原始密钥通过密钥生成算法来生成的。因为每轮都需要一个轮密钥，所以密钥生成算法是很重要的。密钥生成算法包括密钥扩展算法和轮密钥选择两个部分。

3.4.1 密钥扩展

在 AES 加密和解密运算中都需要 Nr 个子密钥，由于每轮都需要一个轮密钥，轮密钥中全部的比特数可表示为 Nb*(Nr+1)，Nb 表示分组长度，Nr 表示对应循环轮数。可用一维数组 W [Nb * (Nr+1)]来表示扩展密钥，当然初始密钥就是数组 W 中初始的 Nk 个字节(Nk = 密钥长度 / 32)。后面的字节由前面的字节按递归方式计算确定，轮密钥 Expandedkey[i]由 W[Nb*(Nr+1)] 中的第 Nb * i 列到第 Nb * (i+1)-1 列给出。其中

$0 \le i < Nb * (Nr+1)$。

密钥扩展方法[4]与 Nk 的值有关。密钥扩展可从 Nk≤6 和 Nk>6 两种情况进行考虑：

1. $Nk \le 6$ 时，密钥扩展的方法如下：

在扩展密钥数组中，W[i]依赖于 W[i-1]和 W[i-Nk];需从两方面进行考虑：

（1）如果数组 W 的下标 i 不是 Nk 的倍数时，密钥扩展只进行简单的异或 xor，逻辑表达式是：W[i]=W[i-1] xor W[i-Nk] 。

（2）如果数组 W 的下标为 Nk 的倍数时，用下面的计方法进行计算：

字节移位：循环左移前一个字的每一个字节，一个字包含四个字节。例如以字节[a_0，a_1，a_2，a_3]作为输入，则输出是[a_1，a_2，a_3，a_0]。

字节替代：使用 S 盒对初始密钥的每个字节进行字节 S 替代。

将步骤字节替换的结果与轮常量 Rcon[i/Nk]进行异或 xor。

此过程为：

W[i] = (SubByte(RotByte(W[i-1]))) xor Rcon[i/Nk]

其中：

Rcon[i] = (RC[i], 00, 00 ,00)

RC[1] = 1（即'01'）

RC[i] = 02^{i-1}；i=2,3,.....

算法伪代码表示如下：

```
KeyExpansion(byteKey[4*Nk] word W [ Nb*(Nr+1) ] )
{
    for ( i=0; i<Nk; i++ )
        W[i] = ( Key[4*i],Key[4*i+l],Key[4*i+2],Key[4*i+3] );
    for(i=Nk; i<Nb*(Nr+1); i++)
    {
        temp=W[i-1];
        If (i mod Nk == 0)
          temp=SubByte(RotByte(temp)) xor Rcon[i/Nk] ;
        else
        W[i]=W[i-Nk] xor temp;
     }
}
```

2. Nk >6 与 Nk≤6 时的密钥扩展方案略微不同，差别就在于当 i 是 4 的整数倍时，W[I-1]必须进行 SubByte 变换。操作后扩展密钥中便增加了部分字的 SubByte 变换，同时也提高了扩展密钥的安全度。

使用伪代码描述密钥扩展算法，如下:

```
KeyExpansion(byteKey[4*Nk] word W[ Nb*(Nr+1) ] )
{   for ( i=0; i<Nk; i++ )
```

```
        W[i] = ( Key[4*i],Key[4*i+1],Key[4*i+2],Key[4*i+3] ) ;
    for(i=Nk; i<Nb*(Nr+1); i++)
    {
        temp=W[i-1] ;
        If (i mod Nk == 0)
            temp=SubByte(RotByte(temp)) xor Rcon[i/Nk] ;
        else if (i%Nk=4)
            temp = SubByte(temp) ;
        W[i]=W[i-Nk] xor temp ;
    }
}
```

3.4.2 轮密钥的选取

轮密钥 i 由缓冲区 W[Nb*i]到 W[Nb*(i+1)-1]间的字组成[41]。AES 的密钥长度为 128bit 时，Nb=4，Nk=4，此时轮密钥的选取如图 3.5。

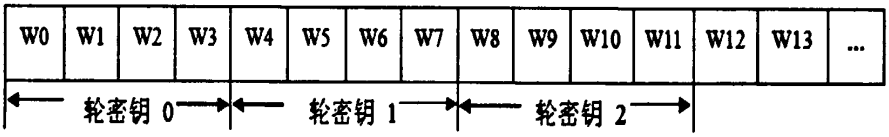

图 3.5 密钥长度为 128bit 时轮密钥的选取

可参照 3.2 节内容，有关密钥长度与 Nb，Nk 关系图。

3.5 工作模式

2000 年 3 月 NIST（美国国家标准技术研究所）为 AES 公开征集工作模式，2001 年 12 月公布了 5 种适用于保密的工作模式[39,40]，他们分别是 ECB（Electronic codebook，电子密码本模式）、CBC（Cipher-block chaining，密码分组链接模式）、CFB（Cipher feedback，密文反馈）、OFB（Output feedback，输出反馈）和 CTR（Counter Mode，计数模式）。

本章所用到的一些参数：P_i代表明文块C_i是指密文块，，E_i是指分组密码的加密运算，D_i表示分组密码的解密操作。IV 是初始化向量 Initialization Vector 的缩写。

3.5.1 电子编码本工作模式

电子编码本（ECB）工作模式很多时候被简称为电码本模式，ECB 是最简单的一种加密模式，一次仅处理一组明文数据，每次使用相同的密钥加密，每个块都独立的进行。因此 ECB 工作模式可以并行的计算密文。

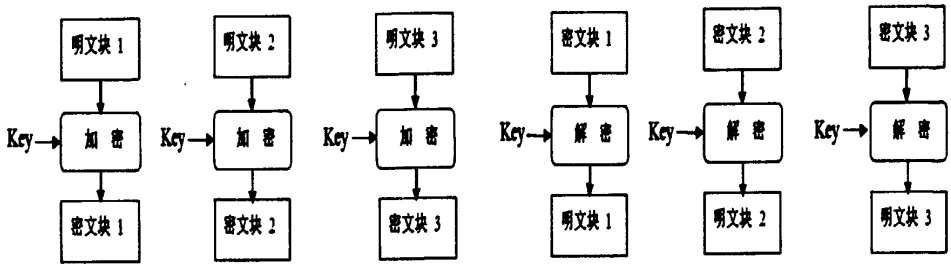

图 3.6 电子编码本工作模式的加密和解密操作

由于每次使用相同的密钥进行加密，明文数据中又包含有固定长度的重复数据，这块明文数据的长度正好等于加密区块的大小，可能会产生相同的密文，从而对明文产生主动地攻击。因此使用 ECB 工作模式存在着一定的安全隐患，密码协议中不推荐使用。

3.5.2 密码分组链接工作模式

密码分组链接（CBC）工作模式在加密当前分组之前，先让前一次加密所得的结果与当前明文进行异或（xor）再加密,直到最后一个密文块，这样的操作会产生密文链。为了保证每条消息的唯一性，在第一个明文分组进行操作时，需与一个初始向量（IV）组进行异或（xor）运算。

CBC 模式的加密过程可表示为：

$C_i=E_K(P_i \oplus C_{i-1})$ ， $C_0=IV$；

而解密方式为：

$P_i=D_K(C_i) \oplus C_{i-1}$， $C_0=IV$；

CCB 模式的优点是使用初始化向量以后，即使相同的明文数据，也会被加密成不同的密文，并且密文上下块是关联的，不容易主动攻击。安全性要高于 ECB 模式。但是 CCB 模式的主要缺点是只能串行运行，加密过程不能并行化。

3.5.3 密文反馈工作模式

密文反馈模式（CFB）与 CBC 的工作过程类似，需要初始化向量（IV）和密钥两个内容，首先通过密钥对初始向量进行加密，得到结果，再使用分组加密得到的结果与明文进行移位异或（xor）运算后得到密文。然后把前一次的密文

作为初始向量再对后面的明文进行加密。

相对简单的密文反馈 CFB 模式的加密过程可表示为：

$C_i = E_k(C_{i-1}) \oplus P_i$；

$P_i = E_k(C_{i-1}) \oplus C_i$；

$C_0 = IV$；

CFB 模式的优点是可以隐匿明文，缺点不适合并行计算，存在误差传递，一个明文块被破坏会影响多个块。

3.5.4 输出反馈工作模式

输出反馈模式（OFB）可以用分组密码产生一个随机密钥流。前提是初始化向量和密钥两个内容，首先运用密钥对初始化向量进行加密，其加密所得结果有两个用处：

1.与明文块进行异或（xor）运算生成密文块。

2.作为下一个初始化向量，对下个明文块进行加密操作。

由于异或（xor）操作的对称性，OFB 模式下的加密和解密操作完全相同：

$C_i = P_i \oplus O_i$；

$P_i = C_i \oplus O_i$；

$O_i = E_k(O_{i-1})$；

$O_0 = IV$；

其中 O_i 是状态，它独立于明文和密文。

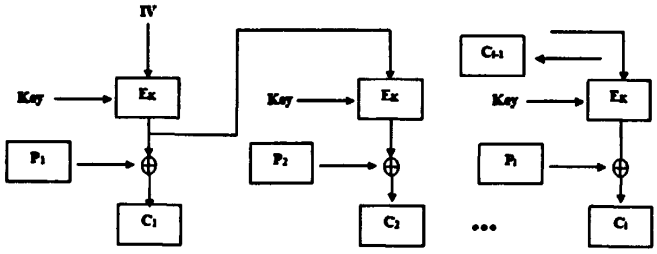

图 3.7 输出反馈加密工作模式

输出反馈模式 OFB 的密文块与前面所有的明文数据块都相关，因此此模式不能进行并行操作。对明文的主动攻击是可能的，安全性不高。

3.5.5 计数器工作模式

计算器模式（CTR）特别的地方在于每次运行使用的初始化向量都是通过计

数器产生的。这个计数设为 ctr（也就是一个初始化向量）。计数器的长度与明文分组长度相同。加密时，首先初始化计数器，然后每增加一个明文组，计数器的值便加1.，再与明文组进行异或得到密文分组。解密时，使用具有相同值的计数器序列，用加密后的值与密文分组进行异或来恢复明文组。CTR 模式的加密解密可表示为：$C_i=E_k(ctr+i)\oplus P_i$ ；　$P_i=E_k(ctr+i)\oplus C_i$ ；

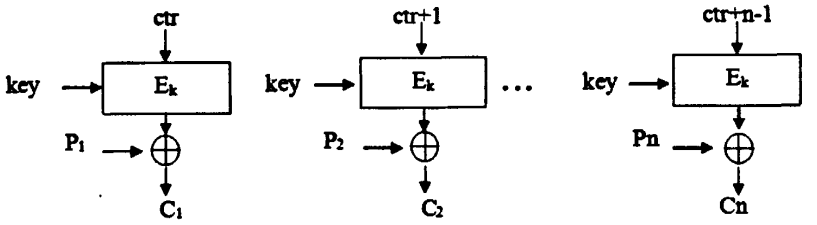

图3.8　计数器模式

3.6 本章小结

本章首先回顾了 AES 高级加密标准的产生背景，介绍了 AES 算法涉及到的数学基础，然后详细阐述了 AES 算法的执行流程，并给出相关步骤的转换状态图及伪代码，最后对密钥生成算法进行了分析，对密码算法的工作模式进行了讨论，分析了各工作模式的优缺点及范围。为后面章节 AES 算法的优化提供了理论基础。

第 4 章 CUDA 并行架构优化技术

对一般算法进行性能的优化[43]，都需要考虑以下三点：最大化并行性以获得最大利用率；优化对指令的选择及使用以获得最大指令吞吐量；优化存储器的选择及使用以获得最大存储器吞吐量；

4.1 并行性优化

最大化并行性就是尽量使得需要并行处理的程序能够有效地工作。在算法中，有些线程为了跟其他线程共享数据而使用同步函数_syncthreads()会导致并行程序执行出错。这种情况下，如果那些线程同属于一个线程块的话，这里仅需使用同步操作_syncthreads()和在同一个内核函数调用中使用共享存储器来共享数据即可解决问题。如果那些线程处在不同的线程块内，此时必须使用共享存储器来共享两个不同内核函数调用的数据，细节是其中一个内核将数据写入全局存储器，而另一个从全局存储器读取数据。第二种优化方法会增加全局存储器通信量。因此设计程序时要尽量将需要互相通信的线程划分到一个线程块。

在流多处理器层次应当最大化利用其内部功能单元使得线程间并行。每个线程块内的线程数都应该是线程束的整数倍，这样可以避免因为线程束占用不足而使得资源浪费。对于给定的内核，设备的计算能力、流多处理器的数量、存储器带宽等因素都对性能产生影响。

4.2 指令优化

指令吞吐量是衡量指令优化程度的一个标准。在实际操作中，指令的吞吐量不仅取决于名义上的指令吞吐量还取决于带宽大小及内存延迟。指令吞吐量是每个流多处理器在单个时钟周期内执行的操作数量。CUDA 指令吞吐量如表 4.1 所示。

if、for、swith、while、do 等控制指令也会导致同一线程束内的线程进行分支，就是形成不同的执行路径，从而影响指令的吞吐量。指令分支会增加为此线程束执行的指令数量。控制条件仅依赖（threadID.x/warp.size）时，控制指令的条件与线程束满足对齐要求，此时不会出现指令分支。

表 4.1 各种算术指令在不同计算能力设备上的指令吞吐量

CUDA 指令的相应吞吐量	计算能力 1.x	计算能力 2.x
32 位浮点运算加，乘，乘加	8	32
64 位浮点运算加，乘，乘加，32 位正余弦	1	16
32 位整数加，逻辑运算，位运算，比较	8	32
24 位整数乘	8	多条指令
32 位整数乘，乘加，累加	多条指令	32
32 位浮点平方根，倒数，指数，对数，正余弦	2	4
类型转换，最大，最小	8	32

由表 4.1 可知：计算能力 1.x 与计算能力 2.x 的设备在本地指令实现的过程不完全一样，不同的编译器版本也会造成本地指令操作数量有小量的变动。

4.3 存储器的访问优化技术

由于 GPU 的存储器数目较多，每种存储器都有自己的限制及适用性，基于 CUDA 架构算法的优化中，存储器的访问优化是一个必备的步骤。本节主要分析全局存储器和共享存储器的访问优化。

4.3.1 host-device 间通信的优化

由表 2.2 测试的 GPU 带宽传输速度可知，本文使用的显卡芯片设备存储器之间的带宽比主机端存储器与设备端存储器之间的带宽要高 30 多倍。可用以下方法来优化主机端与设备端通信优化：

1.最小化带宽的数据传输可以获得更高的存储器吞吐量，也就是说尽量减少主机端与设备端之间数据的频繁传输。 在设备端分配、操作和释放算法执行过程可能创建的中间数据结构；

2.整合主机端与设备端传输的数据，把需要多次进行的传输数据量小的操作让其一次传输，这样一次传输大数据量的操作总比多次传输小数据量的操作性能高。

3.使用主机端的页锁定内存可以获得更高的 host-device 数据传输带宽。若页锁定内存通过合并写的方式进行分配，会获得更高的带宽。页锁定存储器是“稀缺资源”，分配过多的话会导致用于分页的内存减少，进而影响整体性能。因此分配的时候要合理。

4.3.2 全局存储器访问优化

由图 2.8 可知全局存储器的时间延迟为 400-600 个时钟周期，因此全局存储

器经常成为算法实现性能的瓶颈，但如果来自同一 half-warp 的 16 个线程对全局存储器访问满足一定的条件[44]，那么多次存储器访问可以合并为一次完成，从而提高全局存储访问的效率。另一方面可以通过增加线程数，提高计算密度来隐藏全局存储器的时延。对于目标平台 Geforce 9800GT 而言，为了合并全局存储器访问必须满足如下条件：

1.访问的起始地址必须对齐：如果线程访问 32 位字必须对齐到 64 字节；如果线程访问 64 位字必须对齐到 128 字节；如果线程访问 128 位字必须对齐到 128 字节，但必须横跨两个连续的 128 字节区域；

2.half-warp 中的第 k 个线程必须访问第 k 个字。

4.3.3 共享存储器访问优化

在 2.3.3 章节已经对共享存储器做了简单介绍。这里讲解的是共享存储器的访问优化。为了能在访问 CUDA 并行架构时获得更高的带宽，把共享存储器平均划分成小的的存储器块，这样的小存储器块被成为 bank。对应于 a 个 bank 上 a 个地址的访问能同时进行。有 a 个 bank 时的有效带宽是仅有一个 bank 时的 a 倍，因此这种划分方式能提高带宽。如果一个存储器请求的两个逻辑地址在同一个 bank 内，此时会造成 bank conflict（存储器冲突），访问务必序列化。必要时，cuda 架构管理机制会将存在存储器冲突的存储器划分为多个不冲突的请求。假如划分后的存储器请求数量是 n，则可以认为存储器的请求产生了 n 路存储器冲突，此时的有效带宽会大幅下降，其值等于原带宽比上 n。

单独一个 bank 的固定宽度为 32 比特,连续临近的 32 比特字会被组织在临近的 bank 里，一个 bank 在每个时钟周期能提供 32 比特带宽。计算能力为 1.x 的显卡芯片中规定每个线程束包含 32 个线程，每个流多处理器中的共享存储器被画封面成编号为 0 到 15 的 16 个 bank。一个线程束中的线程对共享存储器的访问请求会被划分为两个 half-warp 的访问请求，位于同一个 half-warp 中的线程才有可能发生存储器冲突，线程束中位于第一个 half-warp 的线程与位于后一个 half-warp 的线程之间不会发生存储器冲突[45]。

half-warp 中的所有线程都读取同一个 32 位字内的地址时不会出现存储器冲突。图 4.1 和图 4.2 表示的是没有存储器冲突的共享存储器的访问示例图。

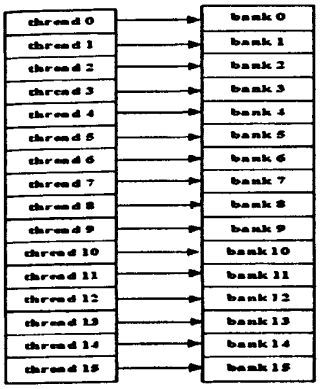

图 4.1 步幅为 32 位字的线性寻址

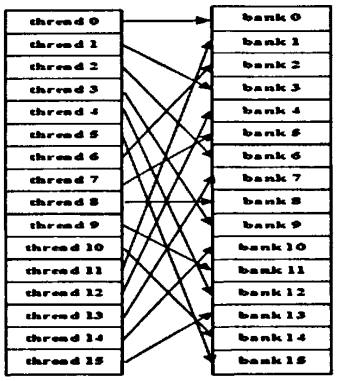

图 4.2 步幅为 3*32 位字的线性寻址

图 4.3 和图 4.4 表示的是有存储器冲突的共享存储器访问模式示例图。

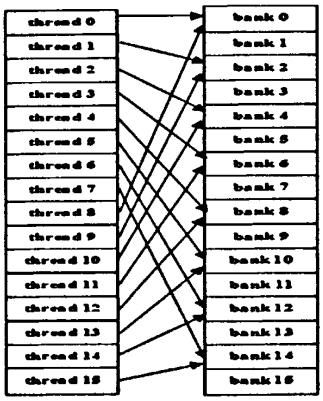

图 4.3 2 路 bank conflict

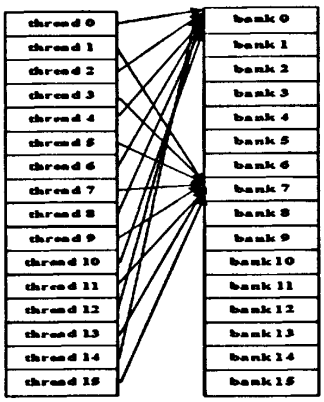

图 4.4 8 路 bank conflict

响应对同一地址的读请求时，可以使用共享存储器的广播机制将 32 比特字读取并广播给不同的线程。图 4.5 和 4.6 显示的是共享存储器广播机制表示图。

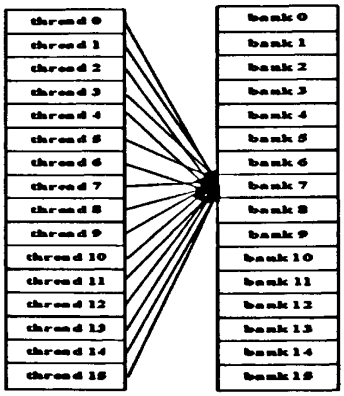

图 4.5 half-warp 读取同一 32 比特地址

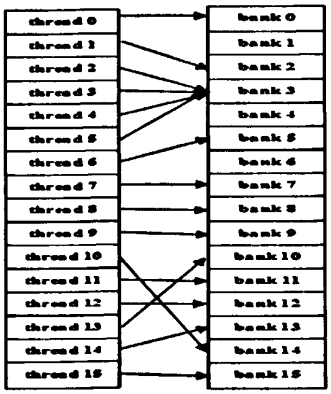

图 4.6 共享存储器广播模式

图 4.5 显示的是一个 half-warp 中的线程对同一地址进行读取时，不会发生存储器冲突。在图 4.6 中，如果 bank 3 的 32 比特字在第一个步骤时被选为广播字，则不会发生存储器冲突，否则会导致 2 路存储器冲突。

4.4 基于 CUDA 并行架构矩阵乘法的研究与优化

矩阵乘法是密集度较高且计算量大的算法，同时矩阵对应的元素之间的计算没有依赖性，又因为在传统 CPU 上执行大矩阵乘法是很耗时的。因此使用 GPU 强大的浮点运算能力来实现大矩阵的乘法是很合适的。

4.4.1 矩阵乘法实现的详细设计

本文实现的详细设计如下：

1.分析两种不同的矩阵优化方法。

2.基于 CUDA 实现两种不同优化方案的矩阵乘法。

3.分配矩阵数组，传输数据，执行内核函数，测试实验并分析结果，对比基于 CUDA 大矩阵乘法的加速效果。

4.4.2 矩阵乘法的实现与优化

传统矩阵乘法的实现首先是通过两个 for()循环遍历两个目标矩阵中各个位置，其次使用两次循环进行两个目标矩阵的乘法运算，然后将运算所得结果进行整合，最后将整合结果存储到结果矩阵中。如果矩阵的规模是 K 阶乘上 K 阶（K>1000），那运算量就很大了，是相当耗时且消耗 CPU 内存的。

若给定矩阵 A 的维度是 i*k，矩阵 B 的维度是 k*j，那么矩阵 A 与矩阵 B 相乘所得的结果矩阵 C 的维度就是 i*j。表达式为：$A_{i,k} * B_{k*j} = C_{i,j}$[46]。

4.4.2.1 优化方案 1

优化方案 1 是每次读取和计算目标矩阵中的一行或者一列。在 CUDA 并行架构中，使用一个线程计算结果矩阵 C 的一个元素，每个线程负责读取和计算矩阵 A 中的每一行和矩阵 B 中的每一列，并将计算的结果矩阵 C 存储在全局存储器。执行优化方案 1，全局存储器需要对 A 矩阵读取 i 回，读取 B 矩阵 j 回。

下面的内核函数表示的是，矩阵 C 中的每个元素由一个线程负责，使用单独的线程负责 A 中的第 x 行与矩阵 B 中第 x 列的对应元素，使用 for 循环进行乘加运算，并将运算结果累加到 Cstate。

```
for( int e=0;e < i;++e)
Cstate+=A.elements[row*A.width+e] * B.elements[e*B.width+col];
C.elements[row*width+col]=Cstate;
```

优化方案 1 的计算量是 2*i*j*k flop，对全局存储器的访问量是 2*i*j*k / B

bit[47]，则内存的计算率为 B/4(flop/bit)。显而易见的是当矩阵维数达到 1000 时，这个运算量就相当大。在矩阵乘法的 CUDA 实现中，优化方案 1 的运算效果并没不理想，效率低下的原因主要是对内存的频繁读取。

4.4.2.2 优化方案 2

优化方案 2 充分利用了共享存储器的高速存储能力。由于共享存储器能够在线程间进行通信，并且其工作延迟要小于全局存储器。优化方案 2 是对矩阵进行分块处理读取并运算，进行优化方案 2 的目的是为了避免矩阵 A 的每一行和矩阵 B 的每一列在内核函数执行时被多次读取。优化方案 2 中矩阵 A 被读取 j / BLOCK_SIZE 次，矩阵 B 也会被读取 i / BLOCK_SIZE 次（BLOCK_SIZE 是指线程块的数量）。优化方案 2 比优化方案 1 节省了全局存储器的带宽。

优化方案 2 就是把目标矩阵按照一定的维度划分成一个个的小块[48]，结果矩阵 C 中的(0,0)～(15,15) = A (0～15,0～15) * B (0～15,0～15) + A (0～15,16～31) * B (16～31,0～15) + A (0～15,32～47) * B (32～47,0～15)+ ... A (0～15,(16*(n-1)-1)～(16*n-1)) * B ((16*(n-1)-1)～(16*n-1),0～15)。

内核函数 kernel 的伪代码表示如下：

```
// 对线程维度的设计如下
dim3block(BLOCK_SIZE,BLOCK_SIZE,1);
dim3grid(((k+BLOCK_SIZE-1)/BLOCK_SIZE),(k+BLOCK_SIZE-1)/BLOCK_SIZE
,1);
```

假设目标矩阵的维度都是 n * n ，BLOCK_SIZE 设置为 x，由上面的线程维度的设计可知，每个线程块含有 x * x 个线程，每个网格由((n+x-1)/x)*((n+x-1)/x)个线程块组成。

```
// 为矩阵 A，B 分配共享存储器存储空间
  for (int x=0;x<i;x+=BLOCK_SIZE)
 {      __shared__ float A[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ floatB[BLOCK_SIZE][BLOCK_SIZE];
 }...
// 执行目标矩阵中的两个子块的乘加操作，每个线程读取和计算结果矩阵 C 中的一个元素值

    for (int k = 0; k < BLOCK_SIZE; ++k)
    {
```

```
            float t;
            C sub + = As [ ty ] [ k ] * Bs [ k ] [ tx ];
            Cs [ ty ] [ tx ] = C sub;
        }
    // 执行同步操作，使得每个线程同步运算结果
    __syncthreads();
    ....
```

一个线程块中线程数量的选择对程序的执行效率是有很大影响的。NVIDIA CUDA 编程手册规定：计算能力为 1.1 的显卡芯片，每个流多处理器最多可以拥有 768 个活动线程(active thread)，而每个线程块最多能分配 512 个线程。NVIDIA CUDA 编程手册建议在每个线程块中分配 256 个线程，究其原因是每个线程块包含有 256 个线程时，流多处理器能尽量满负载的工作，并且每个流多处理器中有足够多的活动线程束来有效地隐藏延迟。线程块的数量也不是设置的越大越好，当线程块的数量较大时，每个流多处理器中的线程自然会多，虽说线程数量越大越能隐藏延迟。毕竟计算能力为 1.X 的 CUDA 架构上每个流多处理器的共享存储器大小仅为 16KB，由于资源平均分配原则，线程数量越多则单个线程分配到的计算资源就越少，因此执行效率会降低。

4.4.3 测试环境参数及实验结果分析

实验所用的软硬件配置的参数及编程环境如下：

CPU：AMD Athlon II X2 245 处理器，核心数为 2，主频是 2.9 0GHZ，一级缓存是 2×64K，浮点运算能力为 17.4 2Gflops。

显卡芯片：NVIDIA Deforce 9800 GT，显卡芯片的计算能力是 1.1。14 个多核处理器，112 个流处理器单元，核心频率是 0.60GHZ，显存带宽是 57.60 GB/s,，单精度浮点计算能力为 336 Gflops。

编程环境：Windows XP 系统，CUDA toolkit 3.0，Visual Studio 2008，。

为了确保实验数据的准确性，实验结果是由 20 次相同测试计算的平均值。实验所用的矩阵维度由 256 * 256 到 2048 * 2048 持续增加。加速比是衡量并行算法性能的一个重要标准。$S_p=T_1 / T_p$，其中 S_p 表示加速比，T_1 是指在单个处理器上的执行时间，T_p 表示的是 P 个处理器的并行系统中的执行时间。峰值比就等于运算速度比上 GPU 单精度浮点运算能力。实验测试所得结果如表 4.3 所示：

表 4.3 不同矩阵维度下矩阵乘法在 CPU 与 CUDA 上的性能比较

矩阵规模	256*256	320*320	512*512	1024*1024	2048*2048
CPU 运算时间/s	0.093	0.213	1.359	29.261	233.203
优化方案 1 运算时间/s	0.156	0.212	0.297	0.564	1.075
优化方案 2 运算时间/s	0.059	0.083	0.117	0.268	0.417

由表 4.3 及加速比计算公式可得图 4.1。

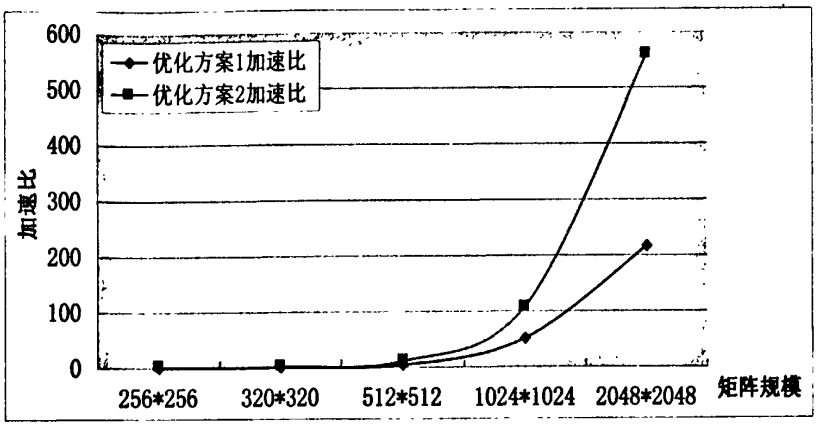

图 4.7 各矩阵规模下优化方案 1 和 2 的加速比

由上表 4.3 及上图 4.1 分析如下：当矩阵规模较小时，GPU 优化方案 1 所测运行时间要大于 CPU 串行执行时间，也就是说此时加速比 $S_p<1$，也说明主存与显存间进行数据传输消耗了一定的时间，数据传输时间占运算总时间的比例较高，因此 CUDA 加速并不明显，这也验证了 GPU 适合做大规模密集型的数据处理。当输入的矩阵规模增大，计算量随着增加，GPU 强大的浮点运算能力就凸显出来，加速比也呈级数增长。矩阵维度为 2048*2048 时执行优化方案 2，加速比最大达到 559.24。GPU 凭借多核处理器的强大浮点运算能力，充分利用其高速的共享存储器获得了较高的加速比，CUDA 架构为运算量密集型计算提高了很好的研究方法。

4.5 本章小结

本章节首先对 CUDA 并行架构的优化技术进行了深入的探讨，着重分析了存储器访问优化技术，特别对全局存储器和共享存储器做了相关优化研究。然后在 CUDA 并行架构上研究了矩阵乘法，并做了相应优化，最后测试实验数据并评价性能。

第 5 章 基于 CUDA 并行架构 AES 算法的研究与实现

5.1 AES 算法优化

AES 算法执行过程中需要进行 10 轮，12 轮或者 14 轮不等的重复轮变换，每个重复轮变换又包括非线性的字节替换变换，线性的行移位变换和列混合变换及轮密钥加变换。因为移位变换操作不能与其他的指令进行配对形成流操作，同时移位变换也是比较费时的操作，这就将很大程度上影响算法的执行速度和执行效率。其次，算法的执行过程包括了多次相同的循环操作，循环和变量操作数会造成指令流水线的阻断和指令预取失败。

根据 AES 算法特性，本文采用一次查找表的方法对高级加密标准 AES 轮函数中的 4 个操作进行了合并优化。

下面的优化过程的初始定义如下：每轮轮密钥的输入状态矩阵为 a ，输出状态矩阵为 b，中间状态矩阵是 k。同时 $a_{i,j}$,$b_{i,j}$ 及 k_{ij} 分别表示各状态矩阵中第 i 行和第 j 列的元素。

对于密钥加变换,可表示如下：

$$\begin{bmatrix} b_{0j} \\ b_{1j} \\ b_{2j} \\ b_{3j} \end{bmatrix} = \begin{bmatrix} d_{0j} \\ d_{1j} \\ d_{2j} \\ d_{3j} \end{bmatrix} \oplus \begin{bmatrix} k_{0j} \\ k_{1j} \\ k_{2j} \\ k_{3j} \end{bmatrix} \qquad \text{公式 (5.1)}$$

在字节替换 SubBytes 变换中，$s_{[i,j]}$ 需要进行字节替换变换，表示式为：

$$c_{i,j} = s_{[i,j]} \qquad \text{公式 (5.2)}$$

对于行移位变换，可表示如下：

$$\begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix} = \begin{bmatrix} s[a(0,j)] \\ s[a(1,j-c_1)] \\ s[a(2,j-c_2)] \\ s[a(3,j-c_3)] \end{bmatrix} \qquad \text{公式 (5.3)}$$

其中，c_1,c_2,c_3 表示行移位变换的偏移量，可参考章节 3.3.2。

列混合变换,可表示如下：

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix} \qquad \text{公式 (5.4)}$$

综合 5.1，5.2，5.3，5.4 四个公式，AES 轮函数可由公式 5.5 表示：

$$\begin{bmatrix} b_{0,j} \\ b_{1,j} \\ b_{2,j} \\ b_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s[a(0,j)] \\ s[a(1,j-c_1)] \\ s[a(2,j-c_2)] \\ s[a(3,j-c_3)] \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix} \quad \text{公式 (5.5)}$$

这样，每一轮循环的每一行仅进行异或操作就可完成，减少了运算时间。

观察公式 5.5 会发现，它虽然减少了运行次数，但还是要多次执行运算效率相对低下的异或操作，何况是与常数矩阵进行运算，还可以作进一步的函数变换。

$$\begin{bmatrix} b(0,j) \\ b(1,j) \\ b(2,j) \\ b(3,j) \end{bmatrix} = s[a(0,j)] \begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} + s[a(1,j-c_1)] \begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} + s[a(2,j-c_2)] \begin{bmatrix} 01 \\ 03 \\ 02 \\ 01 \end{bmatrix} + s[a(3,j-c_3)] \begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix} \otimes \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix} \quad \text{公式 (5.6)}$$

公式 5.6 可以由四个独立的矩阵进行表示，每个矩阵相当于一张表，每个表由 256（16*16）个元素组成。查找表 $T_0[\,]$、$T_1[\,]$、$T_2[\,]$及 $T_3[\,]$都是 8bit 的输入，32bit 的输出函数，每个矩阵占用 1KB(1024 字节)的内存空间。

$$T_0[x] = \begin{bmatrix} s[a(0,j)].02 \\ s[a(0,j)].01 \\ s[a(0,j)].01 \\ s[a(0,j)].03 \end{bmatrix} \quad T_1[x] = \begin{bmatrix} s[a(1,j-c_1)].03 \\ s[a(1,j-c_1)].02 \\ s[a(1,j-c_1)].01 \\ s[a(1,j-c_1)].01 \end{bmatrix}$$

$$T_2[x] = \begin{bmatrix} s[a(2,j-c_2)].01 \\ s[a(2,j-c_2)].03 \\ s[a(2,j-c_2)].02 \\ s[a(2,j-c_2)].01 \end{bmatrix} \quad T_3[x] = \begin{bmatrix} s[a(3,j-c_3)].01 \\ s[a(3,j-c_3)].01 \\ s[a(3,j-c_3)].03 \\ s[a(3,j-c_3)].02 \end{bmatrix} \quad \text{公式 (5.7)}$$

下图是关于查找表 $T_0[\,]$的数组存储形式，$T_1[\,]$,$T_2[\,]$,$T_3[\,]$类似。

```
__constant__ unsigned int T0[] = {
0xa5633c6, 0x84f7c6ff, 0x99ee, 0xd6f7bdf6, 0xdf2f2ff, 0xbd6b6bd6, 0xb1de6fde, 0x54e5c591,
0x92503060, 0x3010102,  0xe96767ce, 0x7d2b2b56, 0x1fefece7, 0x62d7d7b5, 0xe6abab4d, 0x9a7676ec,
0x45caca8f, 0x9d82821f, 0x40c9c989, 0xf07d7dfa, 0x15fafaef, 0xeb5959b2, 0xc947478e, 0xbf0f0efb,
0xeca3a3d41, 0x67d4d4b3, 0xfda2a25f, 0xeaafaf45, 0xbf9c9c23, 0xf7a4a453, 0x967272e4, 0x1bc0c09b,
0xc2b7b775, 0x1cfdfde1, 0xae93933d, 0x6a26264c, 0x5a36366c, 0x413f3f7e, 0x2f7f7f5f,  0x4fcccc83,
0x5c343468, 0xf4a5a551, 0x34e5e5d1, 0x8f1f1f9,  0x937171e2, 0x73d8d8ab, 0x53131162, 0x3f15152a,
0xc7240428,  0x52c7c795, 0x65232346, 0x5ec3c39d, 0x28181830, 0xa1969637, 0xf0705090,  0xb59a9a2f,
0x97072e0e,  0x36121224, 0x9b801b1b, 0x3de2e2df, 0x26ebebcd, 0x692727e1e, 0xcdb2b27f, 0x9f7575ea,
0x1b091209, 0x839e3831d, 0x7c2c2e58, 0x2e1a1a34, 0x2d1b1b36, 0xb26e6edc, 0xee5a5ab4, 0xfba0a05b,
0xf65252a4, 0x4d3b3b76, 0x61d6d6b7, 0xceb3b37d, 0x7b292952, 0x3ee3e3dd, 0x712f2f5e, 0x97141413,
0xf55353a6, 0x68d1d1b9, 0x0,  0x2cededc1, 0x6720205, 0x1ffcfce3, 0xced1b1c7, 0xed5b5bb6,
0xbe6a6ad4, 0x46cbcb8d, 0xd9beba67, 0x4b3939721, 0xde4a4a94, 0xd44c4c98, 0xe8585db0, 0x4acfcf85,
0x6bd0d0bb, 0x2aefefc5, 0xe5aaaa4f, 0x16fbfbed, 0xc5434386, 0xd74d4d9a, 0x85333366, 0x94585511,
0xcf454a8a, 0x10f9f9e9, 0x6020204,  0x817f7ffe, 0xf3505ca0, 0x443c3c78, 0xbae9f9f25, 0xe3a3a48b,
0xf351512a2, 0xfea3a35d, 0xc0404050, 0x8aefffd5, 0xad92923f, 0xbc9d9d21, 0x43535179, 0x4f5f5f1,
0xdfbcbc63, 0xc1b6b677, 0x7d8dadaaf, 0x63212142, 0x3101102f, 0x1effffe5, 0xef3f3fd,  0x6dd2d2bf,
0x4ecdcd81, 0x1c0c0c1d, 0x351111326, 0x2fecece3, 0xe15f5fbe, 0xa2979735, 0xcc444455, 0x391717e2e,
0x57c4c493, 0xf2a7a785, 0xf27e7efc, 0x4f3d3d7a, 0xac6464c8, 0xe75d5dba, 0x2b191932, 0x95737e36,
0xa06060c0, 0x98181119, 0xd14f4f9e, 0x7fdcdca3, 0x662222244, 0x7e2a2a54, 0xab909038, 0x388888cb,
0xca4646fc, 0x29eeeec7, 0xd3b5b56b, 0x3c141428, 0x79dedea7, 0xe25e5ebc, 0x1d0b0b16, 0x756dbdad,
0x3be0e0db, 0x56323264, 0x4e3a3a74, 0x1e0a0a14, 0xdb494992, 0xa06060c,  0x6c242448, 0xe45c5cb8,
0x5dc2c29f, 0x6ed3d3bd, 0xefacac43, 0xa662262c4, 0xa8919139, 0xa4959531, 0x37e4e4d3, 0x8b797f2,
0x32e7e7d5, 0x43c7c782, 0x59373760, 0xb76d6dda, 0x8c8d8d01, 0x64d5d5b1, 0xd24e4e9c, 0xe0a9a949,
0xb46c6cd8, 0xfa5656ac, 0x7f4f4f3,  0x25eaeacf, 0xaf6565ca, 0x8e7a7af4, 0xe9eaea47, 0x1808081,
0xd5baba6f, 0x887878f0, 0x6f2f2544a, 0x722e2e5c, 0x241c1c38, 0xf1a6a657, 0xc7b4b473, 0x51c6c697,
0x23e8e8cb, 0x7cdddda1, 0x9c7474e8, 0x211f1f3e, 0xdd4b4b96, 0xdcbdbd61, 0x868b8b0d, 0x858a8a0f,
0x907070e0, 0x423e3e7c, 0xc4b5b571, 0xaa6666cc, 0xd8484890, 0x5030306,  0x1f6f6f7,  0x120e0e1c,
0xa36161c2, 0x5f35356a, 0xf9575fae, 0xd0b9b969, 0x91e1e17, 0x5f5c1c199, 0x2711d1d3a, 0xb99e9e27,
0x38e1e1d9, 0x13f8f8eb, 0xb3598f2b, 0x33111122, 0xbbb6969d2, 0x70d9d9a9, 0x8e8e8e07, 0xa1949433,

0xb69b9b2d, 0x221e1e3c, 0x926787154, 0x20e9e9c9, 0x49ceced7, 0xff8585aa, 0x78282850, 0x7edfdfa5,
0x8ffcfc03, 0x8fa1a129, 0x80989909, 0x17d0d1a, 0xdabfbf65, 0x31e6e6d7, 0xc6424284, 0xb8686d0,
0xc3414182, 0xb0999929, 0x772d2d5a, 0x110f0f1e, 0xcbb5b07b, 0xfc5454a8, 0xd6bbbb6d, 0x3a16162c
};
```

图 5.1 查找表$T_0[]$的数组表示

根据公式 5.5，5.6，5.7 三个式子，四个查找表可将轮函数进一步变换

$$b_j = T_0[a(0,j)] \oplus T_1[a(1,j-c_1)] \oplus T_2[a(2,j-c_2)] \oplus T_3[a(3,j-c_3)] \oplus k_j \quad \text{公式(5.8)}$$

公式 5.8 中 T[ ]是查找表，⊕表示异或，k_j是轮密钥状态矩阵的 j 列,其中 $1 \leq j \leq 4$。该解决方案在轮函数的每一列只需进行 4 次表查找和 4 次异或操作即可。

5.2 并行 AES 算法在 CUDA 平台实现的可行性分析

过去二十年，随着企业、政府、医疗等关键性领域对互联网络的依赖性越来越高，造成对高效的加密解密解决方案的需求也在不断增长。因此，许多基于 SSL（Secure Sockets Layer 安全套接层）硬件加速的解决方案在科研单位及工业领域被提出并进行了研究，加密算法的运行效率实际上并不理想。随着 GPU 强大的浮点运算能力被认可，基于 GPU 的加密解密解决方案被提出，然而实验的传输速率仅达到 1.53Mbps[50]。主要的原因有以下几点：

1.当时的 GPU 缺少内部数据和位运算的整数表示。

2.图形 API 的编程模型比较有限，缺少一些基本的操作。

3.图形 API 的内存模型受限制。

4.图形处理器流水线的开销是固定的。

2007 年 NVIDIA 公司推出的 CUDA 通用并行计算架构和 2006 年 AMD 公司提出的 CTM（Close To Metal）新型开放式接口都分别为 GPGPU（通用计算图形处理器）扩展了编程模型和内存模型，扩展相关面向图形编程的接口。此外，

CUDA 还直接提供了访问底层硬件的环境。

由于 AES 算法对 RAM（随即存取内存）的需求不大，GPU 中大量的晶体管主要是用于计算单元的，而少量的缓存和控制器就能支撑计算单元的高速运行，同时也没有多个功能部件间的指令缓存及复杂的预测分支。因此 AES 算法能在 GPU 上各线程间进行并行运算。又因为 CUDA 编程模型是市场上第一个使用内部整数来表示本地可编程逻辑的数据运算的 GPU。进行 CUDA 编程时，GPU 被看做是能并行执行大量线程的设备端。因此 AES 算法在 CUDA 并行计算架构上执行是可行的。

5.3 AES 优化算法 CUDA 并行化设计

本章节研究基于 CUDA 并行架构的 AES 算法的设计与实现，具体实现思想如下：首先选择适合于 AES 算法实现的工作模式，深入分析 AES 算法的并行策略，为算法高效的实现做铺垫。然后在 CUDA 异构并行平台上设计线程层次和存储器层次，最后是合理划分串行和并行任务，设计主机端函数和设备端函数。

5.3.1 并行 AES 算法工作模式的选择

根据 3.6.5 章节关于工作模式的介绍，分析了五种工作模式的优缺点，本论文选用计时器模式 CTR 作为目标算法的工作模式。与其它四种工作模式进行比较，CRT 具有以下特点。

CRT 计数器模式能够同时处理多块明文数据和密文数据。并且 CTR 计数器模式能很好地利用 CPU 流水线等并行技术，适用大规模数据的并行计算。CTR 算法首先通过索引当前块进而得到输入块，称之为计数块，再使用对称密钥对计数块进行加密，并和原始的明文块进行 xor 操作[51]。由此可见，算法并不依靠明文和密文的输入，加密操作也仅仅是一系列的异或运算，能提高吞吐量。安全性至少和其他四种模式一样。CTR 模式仅要求实现加密算法，不要求实现解密算法。对于 AES 等加/解密本质上不同的算法来说，这种简化是巨大的。

5.3.2 并行模式的选择

并行模式的选择前还需将处理的任务进行划分，也就是把任务划分为 CPU 主机端程序和 GPU 设备端程序。首先要根据所处理问题的规模选择恰当的算法，将任务中逻辑性强的事务及串行计算交由 CPU 进行处理，计算密度高及费时的大规模并行计算由 GPU 进行处理。

根据第三章的详细介绍，AES 算法是一种串行运算与并行处理都比较显著的算法。下面四部分是由 CPU 主机端进行串行处理的：(1)主机端到设备端的数据传输；(2). 密钥扩展 Key_expansion( )；(3).存储查找表 T；(4).设备端到主机端数据的传输。内核函数包括轮函数的加密执行是由 GPU 设备端进行并行处理的了。设计并行处理的程序时得考虑怎样提高算法的并行性，尽量细化并行粒度，以此来提高执行单元的利用率。还需要合理设计线程间层次关系，减少线程间所处理数据的相互依赖关系，避免线程之间不必要的通信花销。分析数据与处理器的映射关系，安排好算法所涉及数据的存储器选择，使得算法在 CUDA 架构上获得更好的优化。（本论文设计 AES 加密算法的数据分组长度为 128bit）

根据并行粒度大小的划分，有以下三种不同的并行实现方案进行比较及分析。

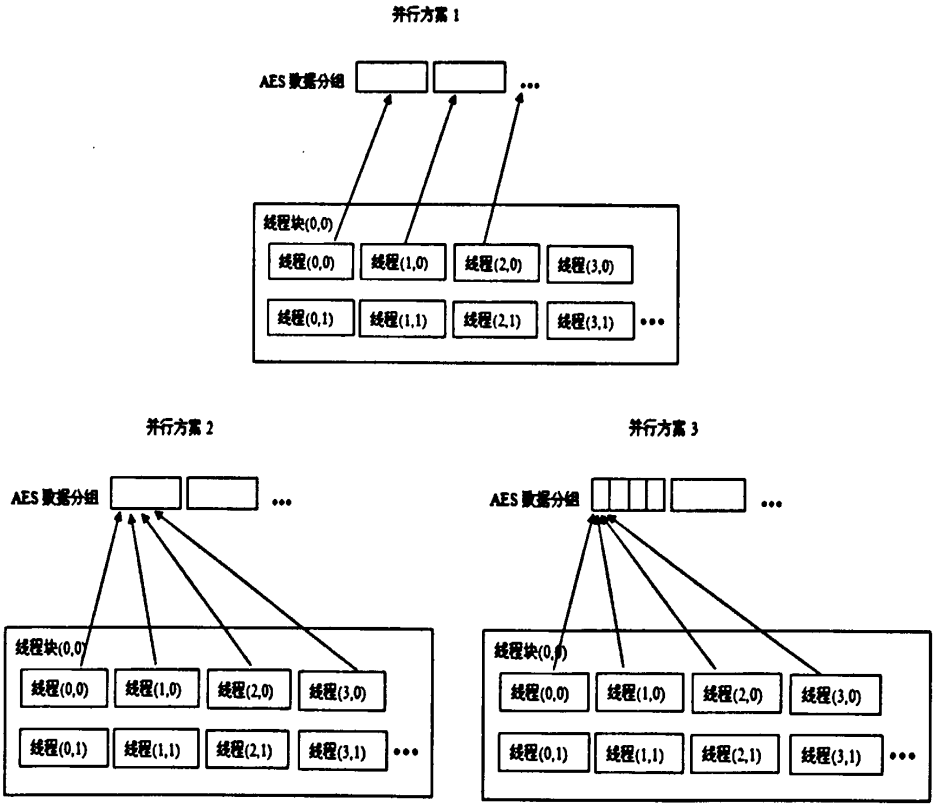

图 5.2 不同并行粒度的三种设计方案

第一种方案是粗粒度设计方案：GPU 上 1 个线程负责处理一个 AES 明文分组，运算的最重要任务便是每个线程与一个 AES 明文分组进行映射。每个线程都是互相独立的，因此线程间数据处理的相关性比较低，线程间并不需要通信，也就是说通信开销为零。由于每个线程需要独自传输和下载负责处理的数据，就达不到合并访问的要求。为了更好的利用带宽，线程块中的线程之间需要相互协

作将数据传输到共享存储器，线程块同步之后各线程才能各自执行任务，最后还需要在线程块内进行合并操作，共同将运算结果传输回全局存储器。这种粗粒度设计方案在，每个线程在计算开始之前都会访问全局存储器 4 次，连续的 4 次访存操作会带来 200*4 个时钟周期的延迟，这么长时间的延迟需要大量活动线程来掩盖这么多延迟。计算能力为 1.1 的 CUDA 架构规定每个流多处理器上最多只能有 24 个活动线程，因此这种粗粒度方案并不能获得相对较好的执行效率。

第二种是细粒度设计方案：4 个线程负责处理一个 AES 明文分组，线程粒度相对第一种方案更细，也就是说单个线程负责处理 32 bit 状态字的一系列操作，四个线程合作完成每一轮状态字的计算。这种设计方案中，每个线程在运算之前仅需要下载 1 次 32bit 字，也就是说只进行 1 次全局访问的读操作。由于每个 32bit 状态字的状态更新之前，都要先读取其他三个线程字运算的上一轮的结果状态字，每个线程要和相邻的其他 3 个线程进行通信，为了减小通信开销，在循环轮函数之间强制插入一个同步栅_syncthreads 操作。

线程束是 CUDA 架构最基本的调度和执行单元，由 32 个连续的线程构成。而 half-warp 是 CUDA 中存储器操作的基本单元，它由线程束中的前 16 个线程或者后 16 个线程构成。分支运算的时候线程束起到很大的作用，而在合并访问和带宽冲突的时候 half-warp 能大显身手。每个线程束中的所有线程不需要同步，因此通信开销为零。相邻的 4 个线程也同处一个 half-warp 中，具有完全同步特性。每一循环轮运算结束后，都将运算结果存储在共享存储器中。于是，每个状态字可同时得到状态更新，然后被 4 个线程同时读取。这种方案能很好的隐匿线程间通信，提高了执行效率。

第三种方案是相对更细粒度设计：每 16 个线程负责处理一个 AES 明文分组，可参照第二种方案，即是每 4 个线程组成的线程组负责处理一个 32bit 字。每个线程在一轮循环中进行一次查找表操作和三次异或 xor 操作，每单个线程的一次异或操作需要将自己的运算结果传输回到共享存储器，再在共享存储器中读取所需计算的数据[52]。因此每个循环相当于要进行 32 次读写操作，这样频繁的读取数据会成为并行执行的瓶颈。此方案实现了更细粒度的划分，在增加并行性的同时也使得通信开销变大。

经过上面的讨论，本文选择第二种细粒度划分方案进行试验，细粒度划分方案的实现流程图如下图所示。

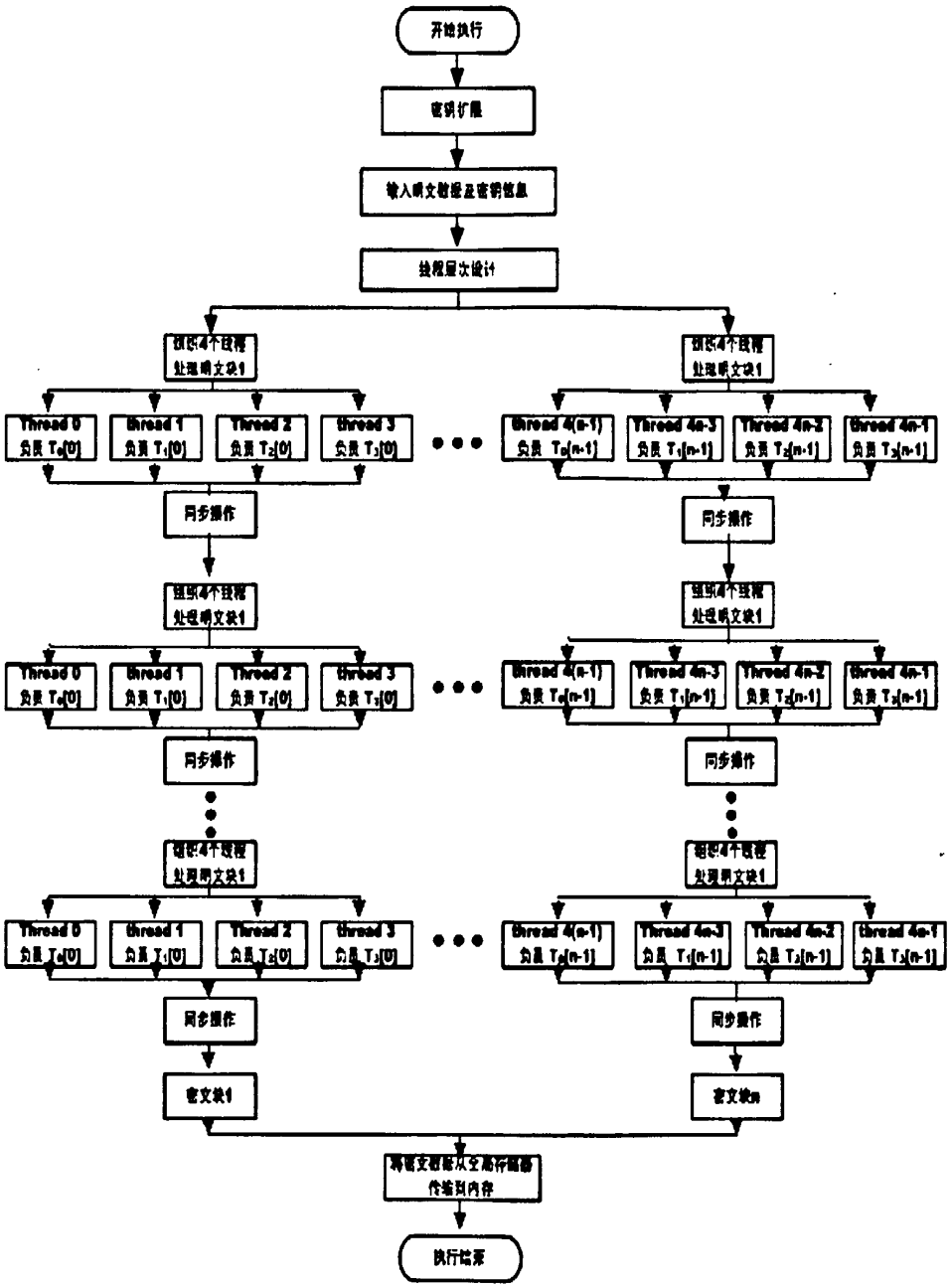

图 5.3 细粒度并行划分实现流程图

5.3.3 线程层次设计

根据 CUDA 编程模型，开发人员拥有很大的自由空间，例如，内存空间的类型，线程之间共享的数据类型，线程块的尺寸大小，等等。但是，与通用编程模型 CPU 相比，在设计高效应用程序时，CUDA 编程模型还是相当复杂的。

目前的 CUDA 架构，每个内核函数只有一个网格，并且在网格中一个维度最多只可以定义 65535 个线程块。实验所用的显卡芯片每个流多处理器最多能处

理 768 个活动线程[21]。考虑到流多处理器满负荷工作时效率最高。每个线程块有 16*16=256 个线程，768/256=3,就是说一个流多处理器同时处理 3 个线程块，又因为每个流多处理器最多仅包括 8 个线程块。因此每个线程块包含 256 个线程时流多处理器工作效率最高。同时最好使每个流多处理器上至少拥有 2 个活动线程块。线程块的尺寸是 32 的整数倍时能更好的利用执行单元。

在细粒度划分方案中，4 个线程负责处理完成一个 AES 明文分组的加解密运算，即每个线程负责 32 bit 字的加解密运算。一个线程块所处理的明文长度的大小为线程块中的线程总数量乘以 32bit。线程索引即是线程所负责的状态字在整个明文分组中的存储地址，本文设计如下线程层次方案：

```
dim3 threads ( Bsize,1 );
dim3 grid( ( InputSize / Bsize ),1 );
...
unsigned t x = threadIdx;
unsigned bx = blockIdx;
unsigned mod 4tx = tx%4;
unsigned int 4tx = tx/4;
thread_index = tx + ( Bsize * bx );
date_index = ( thread_index /4) + ( thread_index mod 4 );
state = pt ( thread_index );
ct ( thread_index ) = state;
```

其中，pt 是 plain text 的缩写，是指明文数据。ct 是 cipher text 密文数据的缩写。state 表示当前线程所负责的明文分组。InputSize 代表输入的明文数据大小，Bsize 是每个线程块中的线程数，InputSize / Bsize 表示一个网格中的线程块数，threadIdx 代表线程在当前线程块中的索引；thread_index 是指当前线程在全部线程序列里的索引；date _index 表示当前线程负责处理的状态字在明文数据中的索引。

5.3.4 存储器层次设计方案

CUDA 并行架构有多种不同类型的存储器结构可供选择。各种存储器的特性差别较大，因此，如何灵活的使用各种存储器的特性将直接关系到算法的执行效率。在 CUDA 并行架构中，要尽量避免让存储器的访问和通信成为限制性能提升的关键因素。为了在高性能并行程序的设计过程中获得更好的性能，就必须

对存储器访问进行详细的规划。

本章节详细分析了AES 算法所涉及到的数据如何在CUDA并行架构上选择存储器。在对存储器进行规划前我们先来考虑程序设计中需要进行存储的数据：明文数据、密文数据、循环轮密钥、CTR 计数器的初始值、查找表 T。图 5.4 描述了数据对相应存储器的选择。

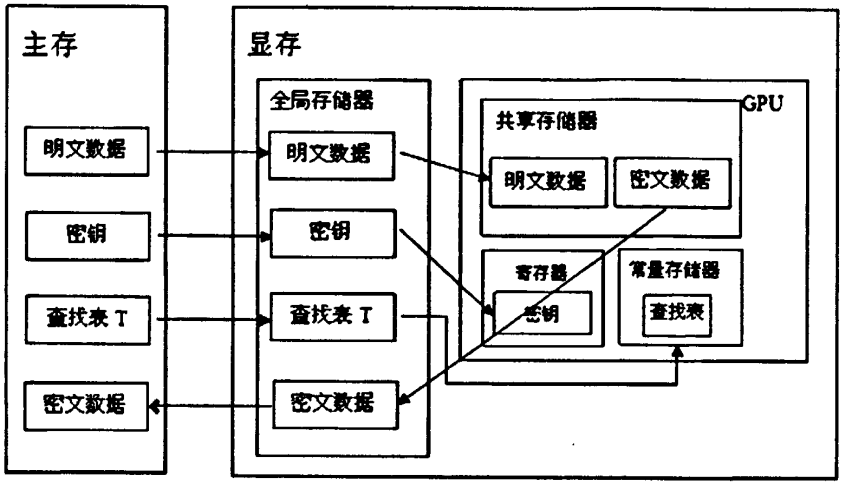

图 5.4 AES 并行算法实现所需数据的存储器选择

本文所用的显卡芯片 NVIDIA Geforce 9800GT 是通过 PCI-E 总线与主机端进行数据传输的。PCI Express 2.0 x16 通道的理论带宽能达到双向每向 8GB/s，这个数值远小于 Geforce 9800GT 的存储带宽 57.6GB/s。

由于主机端页锁定内存能通过 DMA 加速主机端与设备端的通信，使其获得较高的数据传输带宽。因此可以先将输入的明文数据存储在主机端页锁定内存中，然后再将数据传输到全局存储器内，使用 CUDA_SAFE_CALL(cudaMalloc()) 语句在全局存储器内分配线性空间，将密钥扩展存储在 GPU 全局内存。要充分使用设备端共享存储器，这样能减少存储器访问延迟，提高算法的执行效率。密文数据由共享存储器传输到全局存储器中，最后传到主机内存。

轮密钥先存储在全局存储器，然后将其存储到寄存器内。

CTR 计数器的初始值存储在常量存储器，通过常量存储器将计数器初始值发送到每个单独的线程。

对 AES 算法的优化是采用一次查找表的方法对 AES 轮函数中的 4 个操作进行了合并优化。由此可见对查找表的访问速度快慢将影响到 AES 算法在 CUDA 架构的实现速度，因此需要使用速度较快的存储器来存储查找表；4 个查找表的

大小都是 16*16=256Byte。片内存储器的存储速度最可观，所以想到使用片内存储器来存储查找表。由 2.3 节对各种存储器结构的分析，可使用的存储器有纹理存储器、常量存储器。又因为查找表具有一定的随机性，它和明文分组的状态关联紧密，不适合存储在纹理存储器上。常数存储器是拥有片上缓存的只读存储器，通常用于存储那些需要进行频繁访问的只读参数。常量存储器拥有缓存机制，对常量存储器的访问可以获得缓存加速，能够加快访问速度从而节约带宽。因此查找表存储在设备特定的常数存储器比较合适。

5.4 主机端及设备端函数设计

前面已经对 CUDA 线程层次和存储器层次进行了详细的分析及分配，本章节主要是在 CUDA 并行架构上实现优化过的 AES 算法，设计主机端函数和设备端内核函数，并给出相应的分析。

5.4.1 主机端函数设计

主机端处理的大多是顺序执行的串行算法。概括的说，主机端完成的工作为内核函数启动准备数据、执行算法中的并行部分及处理内核函数运算结果三部分组成。当然还有一些设备初始化工作。主机端运行的函数和只从主机端调用的函数是使用_host_限定符进行声明的。如果函数没有_host_限定符修饰的话，默认是用_host_限定符修饰的函数。主机端函数设计如下：

```
_host_ int aesHost ( unsigned char*ct,const unsigned char*pt,int inputSize, const unsigned char*key,int keySize);
```

主机端函数的明文数据和密文结果数据是通过指针参数传递和地址进行实现的。使用的返回类型是整型 int 。const 修饰的是指针的内容，一定程度上提高程序的安全性也节省了内存空间。对应的参数解释：pt 表示明文输入数据地址，ct 代表的是密文输出结果的地址，inputSize 是指明文数据大小，key 是密钥的初始地址。

主机端函数的执行过程由下图表示。

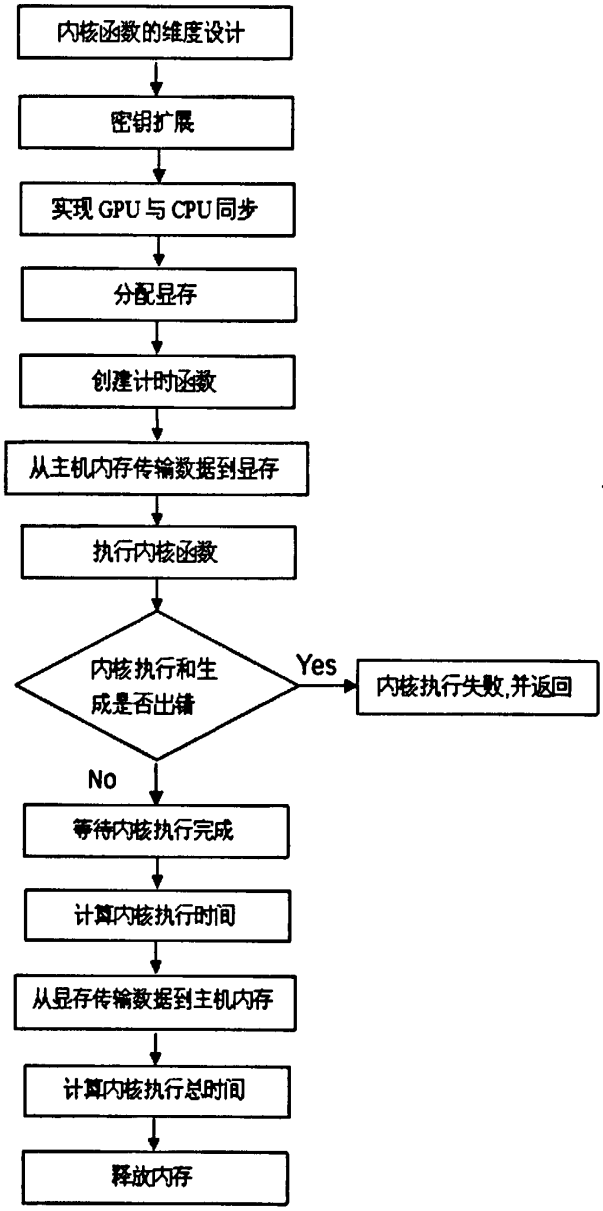

图 5.5 主机端函数执行流程

1.内核函数的维度设计在线程层次设计的时候已经做了分析，由执行参数配置 dim3 threads (Bsize,1); dim3 grid((InputSize / Bsize),1) 可知每个线程块里面有 Bsize 个线程，每个网格包括了 InputSize / Bsize 个线程块。根据本论文算法实现的粒度划分，四个线程处理一个明文分组，所以明文分组个数在数值上等于加密运算所需线程数量的四分之一，由此可知明文分组个数与 inputSize 的大小关系密切。考虑到每个线程块内含有 16*16=256 个线程时，流多处理器的利用率最高。因此在程序设计时 Bsize 被声明为 256，也就是说执行程序时一个线程块内有 256 个线程。如果每个线程块对线程的需求量大于 256 时，线程块分配的线程

数也只有 256。

2.密钥扩展是由主机端函数进行处理的。密钥长度标准为 128bit 时，初始密钥的大小为 128/32=4 字节，即密钥长度 Nk 为 4。密钥扩展算法在 3.4 节有详细的讲解。密钥扩展使用到 S 盒进行字节替换变换,S 盒在很多地方都被访问，所以使用 extern unsigned Sbox [] 对其进行声明，作用范围为主机端和设备端函数都可见。轮常数 Rcon[]定义为 unsigned Rcon[]={ 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36 }

3.想要实现主机端与设备端的同步，需要在主机端代码中使用线程同步函数 cudaThreadSynchronize()，启动同步函数可以确保所有设备端线程均已运行完毕，可以使得主机端计时函数测时更准确。

```
aesEncrypt128<<< grid,threads >>> (d_ct,d_pt, inputSize);
CUDA_SAFE_CALL(cudaThreadSynchronize() );
```

4.分配设备端显存，并将数据从主机端传输到设备端，创建并启动计时函数。

```
unsigned * d_pt;                    //定义指针，指针指向显存，前缀 d_表示设备端
CUDA_SAFE_CALL( cudaMalloc((void**) &d_pt, inputSize) );
//分配设备端存储器
unsigned * d_Key;
CUDA_SAFE_CALL( cudaMalloc((void**) &d_Key, keySize) );
// keySize 等于密钥的内存大小，每个线程传输一个密钥块到共享存储器
unsigned int ext_timer = 0;                                    //创建计时函数
CUT_SAFE_CALL(cutCreateTimer(&ext_timer));
CUT_SAFE_CALL(cutStartTimer(ext_timer));                       //启动计时函数
CUDA_SAFE_CALL(cudaMemcpy(d_pt,h_pt,inputSize,cudaMemcpyHostToDevice
));
CUDA_SAFE_CALL(cudaMemcpy( d_Key,   h_key, keySize, cudaMemcpyHostTo
```

Device)); //从主机端到设备端传输数据，前缀 h_表示 host

主机端到设备端传输的数据还有查找表 T0，T1，T2，T3。查找表 T 首先被传输到设备端的全局存储器，然后再传输到适合存储查找表的常量存储器中。

```
_constant_ unsigned int T0[]={...};                    //声明查找表 T0
CUDA_SAFE_CALL ( cudaMemcpyToSymbol ( "d_T0",T0,256 ) ) ;
//将查找表从主机端传输到设备端的常量储存器
```

5.数据传输完成后，下面将进行的是内核函数的调用及内核函数的状态检查。

```
aesEncryptHandler128( d_ct, d_pt, inputSize);  ...
CUT_CHECK_ERROR("Kernel execution failed");
 //检查 kernel 函数执行是否生成错误
```

6.内核函数执行完成，首先测试内核执行时间，其次为密文数据分配全局存储器，然后将密文数据从设备端的全局存储器传输到主机内存，最后关掉计时函数，测试内核执行的总时间（包括密文数据传输时间）。

7.主机端程序执行的最后工作是删除计时函数，释放内存空间及显存空间。若不及时释放会造成内存及显存溢出，多次运行后会造成显卡无法正常工作。

5.4.2 设备端内核函数设计

在 GPU 上运行的并行程序被称为内核函数。内核函数只能在主机端代码中被调用，并且必须使用函数类型限定符_global_进行定义。在调用时，得首先声明内核函数的执行参数。

4 个线程负责处理一个 AES 明文分组，首先定义两个循环块保存密文数据的中间结果，因为下一轮循环调用的是上一轮循环的计算结果。函数体外使用整数纹理坐标将获取的线性内存区域绑定到纹理参考中。定义线程执行配置，为轮密钥状态矩阵分配内存，将密文中间结果存储在共享存储器中，然后对应查找表 T 的数组元素进行查表操作。执行线程同步，并行执行 10 轮循环。最后将共享存储器中的密文数据结果传输到全局存储器。伪代码的流程如图 5.6 所示：

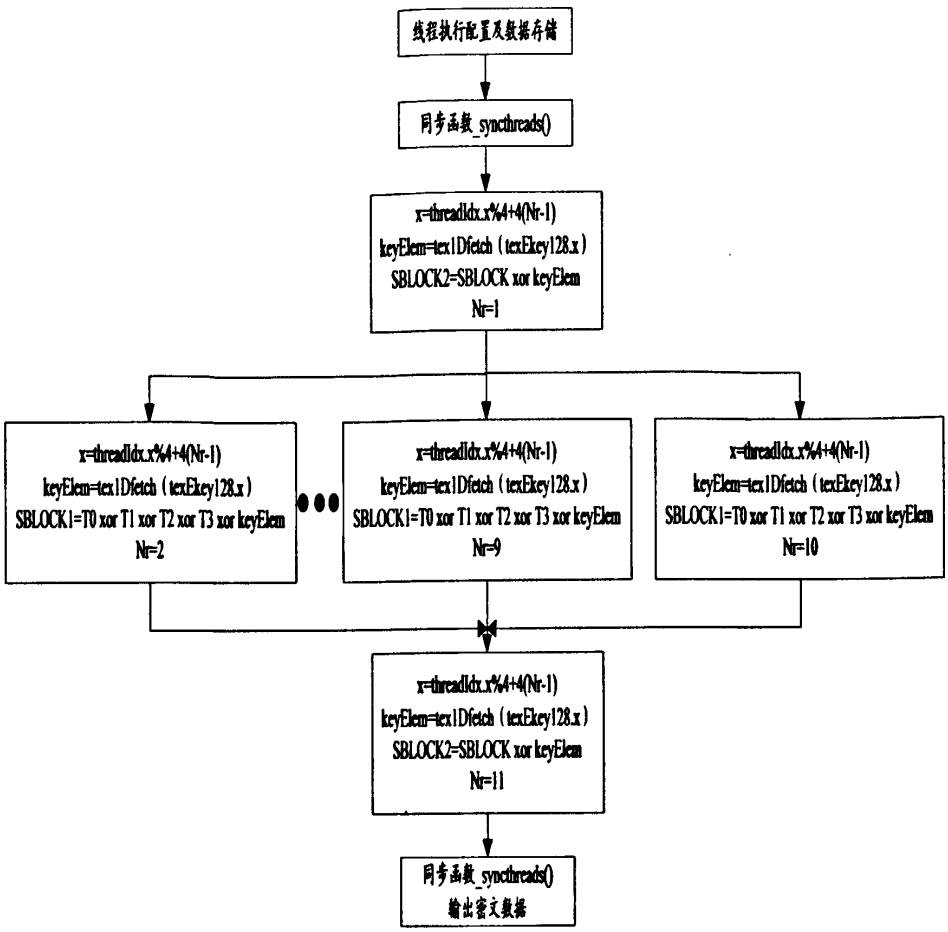

图 5.6 设备端伪代码实现流程图

内核函数的执行在某种意思上可以看做是线程的执行，_syncthreads（）实现了线程块内的线程同步,此函数能保证同一线程块内的线程都执行到同一个位置。同时也能避免程序执行出现写后读，读后写，写后写等安全问题[53]。将明文数据传输到共享存储器的数组时，使用同步操作确保共享存储器中的数据全部传输完毕。计算完密文数据后，同样要保证共享存储器的数据已经全部传输到全局存储器。每四个线程共同完成每一轮加密操作，因此需要调用同步函数进行同步操作。一般情况下使用一次_syncthreads()操作至少要花费四个时钟周期，甚至更多，因此要尽量减少对_syncthreads()函数的调用。

5.5 实验测试及结果的性能分析

5.5.1 实验的软硬件及编程环境配置

本论文实验所用到的软硬件环境及编程环境的相关配置如下：

CPU：AMD Athlon II X2 245;时钟频率:2.90GHz;一级数据:16KB;二级缓存:1024KB;核心数：2。

主机内存：DDR 2;内存：1024MB; DRAM 频率: 199. 5MHz。

GPU 芯片：NVIDIA Geforce 9800GT：14 个流多处理器 SM，112 个流处理器 SP; GPU 时钟频率:600MHz;显存位宽: 256bit，存储器容量 1024MB;存储器带宽: 57.6GB/s; 存储器频率:1800MHz; 显卡总线接口：PCI Express 2.0 x16。

编程环境：cudatoolkit_4.0，gpucomputingsdk_4.0；ubuntu 10.10；gcc 4.4。

5.5.2 实验结果及性能分析

主机端函数使用测时函数 cutCreateTimer()、cutStartTimer()、cutStopTimer()和 cutDeleteTimer()来测试内核运行时间和内核运行总时间，其中内核运行总时间包括内核运行时间和数据传输时间，测时函数返回的时间以毫秒（ms）为单位。为了保证实验的准确性，实验中所测数据是通过 20 次反复实验所得的平均值。

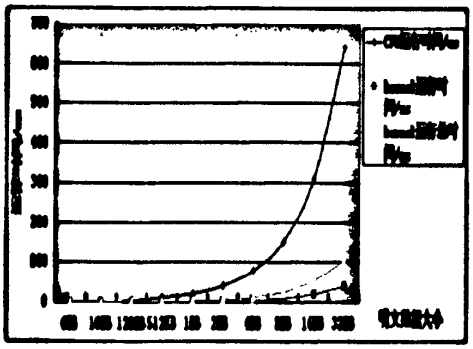

图 5.7 CPU 与 GPU 运行时间对比

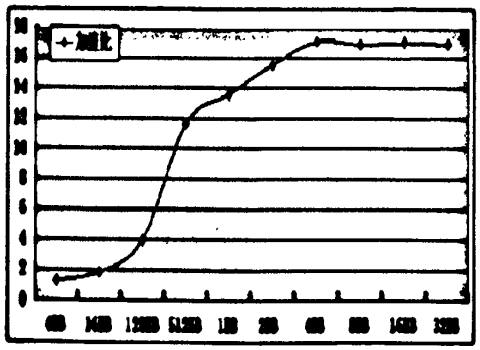

图 5.8 AES 算法加速比

由上面实验结果可知：当明文数据小于 128KB 时，基于 CUDA 架构 AES 并行算法的运行时间相对于 CPU 串行运算时间并没有什么优势，加速比也仅仅在 4 以下。由于当前 CUDA 架构不支持对某个核心处理单元独立分配任务，需要先缓存一定量的计算数据然后把数据传输到 GPU 运算，最后拷贝到内存，这样造成的延迟就会比 CPU 串行处理要多。运算量很小的情况下，GPU 并不能彰显其强大的浮点运算能力。

运行时间是衡量算法运行效率高低的指标之一。随着明文数据的增大，由上图可以明显看出内核运行时间及内核运行总时间都要远小于 CPU 运行时间，加速比在增大，这说明随着运算量的成倍增加，GPU 高效的并行处理能力得以体现。同时我们会发现内核运行时间与内核运行总时间的差距在线性增加，很显然的是随着明文数据的增加 IO 数据传输的时间也在增加，PCI-E 带宽会成为程序执行的瓶颈[54],加速比最大停留在 17 左右。

吞吐量是有效衡量算法在 CPU 与 GPU 运行环境下效率的一个重要指标。这里所指的吞吐量在数值上等于数据大小比上运算时间。实验所用 CPU 的时钟频率为 2.9GHZ，GPU 的时钟频率为 1.12GHZ。

由图 5.9 可知，CPU 的吞吐量一直提升不大，趋于直线，这说明 CPU 运算在一开始就达到了 CPU 相对较高的吞吐量。即使数据量不断增大，但是其运算单元处理数据的能力毕竟有限，吞吐量在 0.40Gbit/s 左右波动。相对于 CPU 吞吐量曲线的平缓，GPU 吞吐量的变化是线性的。数据量小于 8MB 时，吞吐量在成倍增大。当图 5.9 中的 GPU 总体吞吐量稳定在 2.43Gbit/s，GPU 吞吐量在 7.00Gbit/s 左右波动时，明文数据量虽然在不断增大，但是吞吐量并没有随其增大，甚至有所下降，这就侧面说明数据传输成为了程序性能提升的瓶颈。

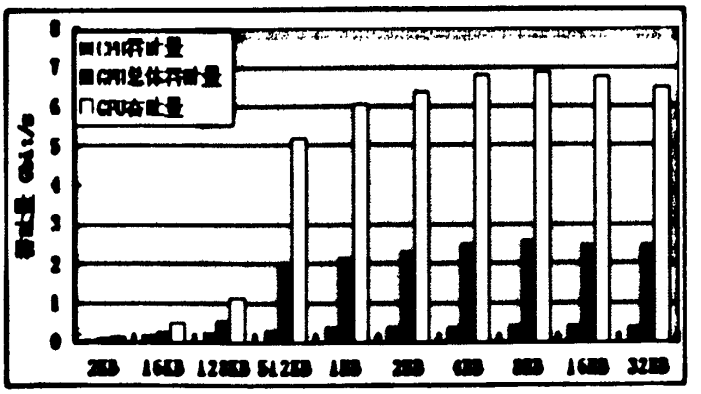

图 5.9 CPU 吞吐量与 GPU 吞吐量比较

当然随着明文数据量的增大，CUDA 并行架构内 thread 数量也会随之增加，毕竟每个流多处理器中线程数量是有限的。如果线程数量达到饱和，而数据量还在增加，所需的计算资源也会增加，此时会造成带宽冲突或者延迟，随之吞吐量会下降。

吞吐量比值是 GPU 吞吐量与 CPU 吞吐量之间的比值，直观的说明了 GPU 的计算单元得以很好的利用。吞吐量比值最高可达到 17，也就是说基于 CUDA 并行架构的 AES 优化算法的吞吐量是串行 AES 优化算法吞吐量的 17 倍。

5.5.3 存储器优化

在 CUDA 特有的并行架构下，能合理利用硬件资源就能获得较好的算法性能。如图 5.9 所示，当明文数据 512KB 时 GPU 吞吐量就获得了比较理想的数值，但是当数据规模持续增大并没有使得吞吐量有相应提升。本课题利用 CUDA 硬件资源，对数据进行存储器优化（基于 5.3.3 章节存储器分配方案）。

本课题讨论两种存储器优化方案：

1.轮密钥，查找表 T 和明文数据块先被加载到全局存储器，然后加载到共享存储器。

2.首先把查找表 T 存储在常量存储器，然后传输到共享内存。轮密钥和明文数据块先被存储在全局存储器，然后再传输到共享存储器存储。

在数据量一定的时候，多次读取小数据，会因为数据的频繁的读写会造成很高的通信延迟，不如一次读取多个数据，这样更加安全并且效率相对高。

随着明文数据的不断增大，GPU 分配的线程数量也随着增加，因为线程在执行的时候会占用常量存储器的一级缓存，线程量比较大导致常量存储器的缓存不足的情况发生。共享存储器能被线程块里的所有线程访问，并且访问速度也可观，线程间的通信延迟也非常的小。同时在每个流多处理器中有 16KB 的共享存储器，而常量存储器为 8KB。又因为每个查找表的大小为 256Byte，可以把查找表 T 存储在共享存储器中，并且把查找表中的值分配到每个流多处理器中的线程中。这样就使得查找表 T 并行的加载到共享存储器，从而节省了数据频繁读取所用时间。由于明文数据块需要同时进行写入和读取操作，处理结果应该传输 DRAM，因此应该把数据先暂时存储在全局存储器，再加载到共享存储器。由于轮密钥数据量比较小，因此可以把它加载到常量存储器或者全局存储器，然后传输到共享存储器进行处理。在内核函数中的每轮计算需要顺序的执行 4 个查找表，毕竟频繁的读取查找表 T 会消耗大量的时间。内核函数的每轮计算都得执行 4 个查找表，在各个线程对查找表的加载更新后要加上同步操作 _syncthreads()，这样是为了确保查找表的每个元素都被更新完毕和数据安全。

对存储器分配方案进行优化后，所测数据如图 5.11 所示：

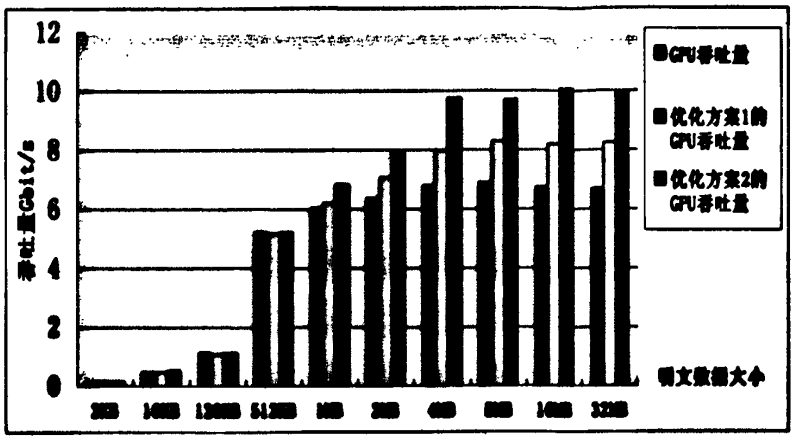

图 5.10 存储器优化后 GPU 吞吐量的比较

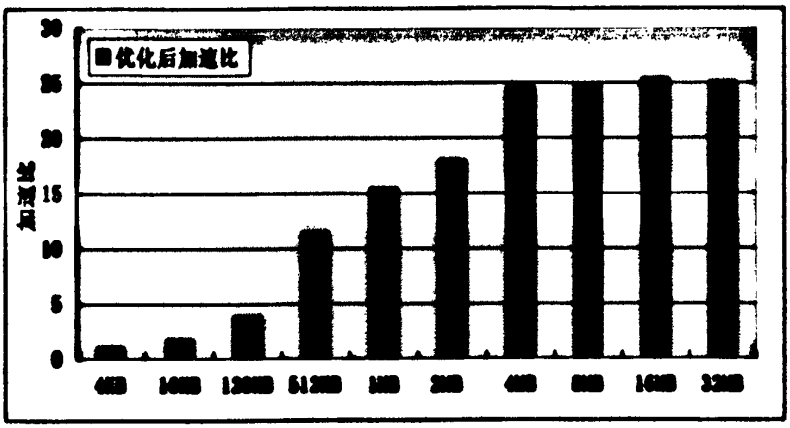

图 5.11 优化后的 GPU 并行实现 AES 算法的加速比

图 5.10 和图 5.11 所测得的数据不包含明文数据加载与密文结果输出部分耗费的时间。这样所得的实验数据更能凸显 GPU 高效的计算能力。图 5.10 所示，优化后的 GPU 吞吐量在明文数据大于 512KB 时才有性能的提高，究其原因是当明文数据小于 512KB 时，算法的优化并没有在吞吐量上得以体现。执行的过程中加入了同步操作_syncthreads(),尽管同步操作能保证数据的安全性，但随着同步操作的增多，会造成执行单元的闲置，毕竟计算资源是有限的，每使用一次同步操作_syncthreads()至少得花费 4 个时钟周期。在数据量大的情况下，算法的优化就隐匿了同步操作所带来的延迟。虽然优化方案 2 对算法性能有所提升，但是提高的幅度仅有 1～2 倍，GPU 的吞吐量最高可达到 10Gbit/s，此时，加速比最高达到 25。10Gbit/s 也就相当于 1.25GB/s。由表 2.1 可知 Geforce 9800GT 主机内存到显存的带宽传输速度为 1.707GB/s,而显存到主机端内存传输的带宽传输速度为 1.527GB/s,由此说明存储器优化分配方案已经达到峰值带宽传输速度。

图 5.12 所示，基于 CUDA 架构存储器优化后 AES 算法，在线程块中设置不同数量的线程时 GPU 的吞吐量。

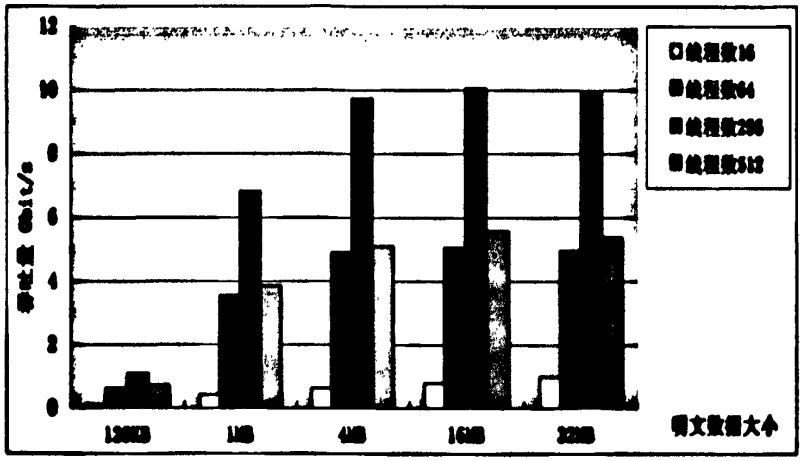

图 5.12 不同线程时 GPU 吞吐量比较

在目前的 CUDA 架构中，线程束的数量规定是 32。线程是以线程束为单位进行分组计算的，每 32 个线程为一组执行同样的指令。所以线程块里的线程最好为 32 的倍数，不然会造成线程计算能力的浪费。

1.当线程数小于 32 时，吞吐量比较小，是因为每个线程块里只有一个线程束，因为一个流多处理器一次只会执行一个线程束，只有 1 个线程束的情况下，流多处理器没有足够多的活动线程束来隐匿执行过程出现的等待时间。

2.计算能力 1.x 的显卡芯片，每个流多处理器一个时刻最多能处理 768 个活动线程束,每个线程块只能容纳 512 个线程。又因为每个流多处理器最多能容纳 8 个线程块，同时最好让每个流多处理器上至少拥有 2 个活动线程块。当每个线程块内线程数量为 256 时，线程被分为 16 个线程束，此时每个流多处理器使用 3 个线程块。实验结果如图 5.11,在线程数为 256 时 GPU 获得了最高的吞吐量，正验证了上述分析。

3.当线程数量等于 512 时，吞吐量没有增大反而减小，这说明线程数量与吞吐量并不成正比。在一定情况下，一个流多处理器中的线程越多就能隐藏更多的延迟，但是当线程数量增大时，每个线程所分的计算资源有限，毕竟计算能力 1.x 的显卡芯片上一个流多处理器仅有 8194 个寄存器和 16KB 的共享存储器。因此在优化的时候对线程数量的选择要折中。

终上所述：线程数量为 256 时，流多处理器能够满载执行，能很好的隐藏延迟，此时 GPU 能获得最高吞吐量。

5.6 本章小结

本章首先对 AES 加密算法进行了优化，然后选择合适的工作模式及并行模式。其次根据 CUDA 架构的异构并行特点设计了 AES 算法的线程层次和存储器层次，再次对如何设计主机端函数和设备端函数做了详细叙述，使得 AES 算法在 CUDA 平台高速实现。然后分析对比了实验测试各项结果，比如运行时间、加速比、CPU 与 GPU 的吞吐量。再然后根据 CUDA 硬件架构优化了存储器分配方案，最后分析了不同线程数量的选择对吞吐量的影响。

第6章 结论

6.1 总结

随着人们对信息的安全需求越来越高，传统 AES 加密算法的执行速度和效率已经不能很好的满足人们的要求，而基于硬件设计的 AES 算法加速器的运算速度不是很理想，并且成本比较高。考虑到显卡芯片拥有相对低廉的价格，较高的计算能力和存储能力等优点，CUDA 编程是一个很好的并行发展方向。本文以高级加密算法 AES 为研究对象，利用异构并行的 CUDA 平台对 AES 加密算法进行研究。本文主要研究工作可总结如下：

1.结合现有的实验条件，选择在 CUDA 并行异构平台下采用 CUDA C 语言对 AES 加密算法进行并行优化，硬件方面只需要一台配置 NVIDIA Geforce 8 以上级别的计算机，而传统 CPU 或硬件实现 AES 算法的片外存储速度较低。又因为 GPU 拥有强大浮点运算能力，相对而言在 CUDA 并行架构实现 AES 所需的软硬件资源少，开销小。

2.根据 CUDA 架构特性，搭建基于 CUDA 并行计算实验平台，并以主机内存与显存间的带宽传输速度来测试平台的可用性。将运算量大、耗时的矩阵乘法移植到 CUDA 并行平台，并对其进行优化，通过实验证明，基于 CUDA 平台优化后的矩阵乘法获得了很高的加速比，程序执行效率显著提高。

3.采用查找表的方式对轮函数进行了优化，AES 算法移植到 CUDA 平台上，合理划分串行及并行任务，设计线程层次及存储器层次，编写主机端函数及设备端内核函数。并根据 CUDA 优化策略对其进行存储器优化。CUDA 并行处理的优化过的 AES 算法的运行时间明显减少，得到理想的吞吐量。实验结果验证了本文研究方法的可行性与解决问题的有效性。

6.2 展望

随着研究工作的不断深入，发现本论文还存在一些值得继续改进和完善的地方。我们可以考虑从以下几点进行深入的研究：

1.文中在并行过程中所采用的任务划分和存储器分配方式与国外一些优秀的算法相比还存在一定的差距，影响了并行优化性能，能更好的应用 CUDA 架构编程特性来获得更好的执行效率，这是有待进一步改进的地方。

2.随着 NVIDIA 公司推出新的 Fermi“费米”和 Kepler“开普勒”并行架构，费米和开普勒提供了更多的计算资源，并且更加适用于通用计算。因此，下一步工作将继续深入地研究基于CUDA架构密码算法的实现技术，例如研究如何在CUDA并行架构上实现哈希算法和椭圆曲线密码算法 ECC 等。

3.GPU 拥有超强浮点运算能力，可以尝试多显卡 CUDA 编程技术来提高密码算法的运算效率，或者整合并行编程模型 CUDA，MPI 及 OpenCL 的优点，设计出一种高效、统一并且易编程的并行计算模型将会成为今后研究的热点方向。

参考文献

[1] Jason Sanders,Edward Kandrot.CUDA by Example:An introduction to General-Purpose GPU Programming 1st edition[M].Addison-Wesley Professional,2010.6 :sequence

[2] 年华.GPU 通用计算与基于 SIFI 特征的图像匹配并行算法研究：[硕士学位论文].西安：西安电子科技大学，2010：2～4

[3] 张舒，褚艳利，赵开勇，张钰勃.GPU 高性能运算之 CUDA[M].北京：中国水利水电工业出版社，2009.10：1～2，44～47

[4] 贾旭.AES 算法的安全性分析及其优化改进：[硕士学位论文].长春：吉林大学，2010.4：2～3，15～17

[5] 张金辉，郭晓彪，符鑫.AES 加密算法分析及其在信息安全中的应用[J].信息网络安全，2011.5：31

[6] Zhao Li，R.Iyer，S.Makineni，L.Bhuyan.Anatomy and Performance of SSL Processing [A]. In：Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software，IEEE Computer Society[C] ，Washington，2005：197-202

[7] 百度百科.天河一号[EB/OL].http://baike.baidu.com/view/2932264.htm，2010

[8] 孙敏杰.天河一号A称霸!超级计算未来属于GPU[EB/OL].http://www.pcpop.com,2010.11

[9] 杨琳桦.NVIDIA 详解 CUDA 攻略："麦当劳"式推广之道[EB/OL].http://news.csdn.net，2009.6.2

[10] John D.Owens,Mike Houston,David Luebke,Simon Green，.GPU Computing [J].Proceeding of the IEEE,2008.96(5):879～899

[11] Chonglei Mei,Hai Jiang,Jeff Jenness.CUDA-based AES Parallelization with Fine-Tuned GPU Memory Utilization[J].2010 IEEE,2010.3:3～7

[12] D.Cook,J.Loannidis,A. Keromytis,and J. Luck.Cryptographics:Secret key crptography using graphics cards,RSA Conference.Cryptographer's Track (CT-RSA),2005:3～5

[13] 张鹏.基于GPU的并行AES加密算法研究：[硕士学位论文].长春：吉林大学,2011:36～39

[14] 周学广.信息安全学第2版[M].北京：机械工业出版社，2008：35～38

[15] 刘丹，赵广辉，钟珞.GPU 加速希尔加解密方法的研究[J].计算机工程与应用，2010：54～58

[16] 张舒.基于Nvidia GPU的通用计算开发[EB/OL]. http://wenku.it168.com，2010：12～17

[17] 苏华友.基于CUDA的H.264并行编码器研究与实现：[硕士学位论文].长沙：国防科技大学，2010：2～5

[18] 方旭东.面向大规模科学计算的CPU-GPU异构并行技术研究：[硕士学位论文].长沙：国防科技大学，2009：16～20

[19] David Kirk,Wen-mei W.Hwu.Programming Massively Parallel Processors:A Hands-on Approach[R],2010.2:34～38

[20] 张武生，李建江.MPI并行程序设计实例教程[M].北京：清华大学出版社，2009：54～57

[21] 中关村在线,濮元恺.提高多GPU编程与执行效率CUDA4.0初探[EB/OL]. http://vga.zol.com.cn/219/2194832_all.html,2011.3.7

[22] 邓仰东.NVIDIA CUDA 超大规模并行程序设计训练课程[R].清华大学微电子学研究所，2010：21～26

[23] NVIDIA C CUDA Programming Guide.version 3.2[Z],2010.2:7-8,82～84

[24] NVIDIA CUDA C Best Practices Guide.version 3.2[Z],2010.2:15～17

[25] David Kirk,Wen-mei W.Hwu. Taiwan 2008 CUDA Course Programming Massively Parallel Processors :the CUDA experience[R].Taiwan, 2008.2:36～40

[26] 赵开勇.CUDA 学习入门资料整理[H].CSDN 论坛，2009：3～12

[27] 徐建.基于 CUDA 的 2D-3D 医学图像配准技术研究：[硕士学位论文].广州：南方医科大学，2010.4：15～18

[28] Erilk Wynters.Parallel processing on NVIDIA graphics processing units using CUDA[J]. Consortium for Computing Sciences in Colleges,2011.2:3～6

[29] NVIDIA Corporation.NVIDIA CUDA CUBLAS Library PG-00000-002_V3.0, 2012.2:23～27

[30] 姚平.CUDA 平台上的 CPU/GPU 异步计算模式：[硕士学位论文].合肥：中国科学科技大学，2010：32～38

[31] 高卓.基于岛的遗传算法在 CUDA 上的优化实现：[硕士学位论文].长春：吉林大学，2011：44～46

[32] 多相复杂系统国家实验室，多尺度离散模拟项目组.基于GPU的多尺度离散模拟并行计算[M].北京：科学出版社，2010.4：112～117

[33] Chunyang Gou Georgi N. Gaydadjiev.Elastic Pipeline: Addressing GPU On-chip Shared Memory Bank Conflicts[J].2011 ACM 978-1-4503-0698-0/11/05,2011.5:1～7

[34] 陈国良.并行计算：结构·算法·编程[M].北京：高等计算出版社，2003.8：89～96

[35] 胡慧丽，陈庆奎，庄松林.基于CUDA 的3G视频清晰度评估方法[J].计算机工程，2011.9：33～35

[36] 韦宝典.高级加密标准AES中若干问题的研究：[博士学位论文].西安：西安电子科技大学，2003.12：12～15

[37] 曹志刚.高级加密标准(AES)算法-Rijndael的研究及FPGA上的实现:[硕士学位论文].哈尔滨：哈尔滨工程大学，2006：7～10.

[38] 戴尔蒙瑞蒙，谷大武，徐胜波译.高级加密标准 AES 算法 Rijndaesl 的设计[M].北京：清华大学出版社，2003：98

[39] 何明星，林昊. AES 算法原理及其实现[J].计算机应用研究，2002.12：26

[40] Federal Information Processing Standards Publication 197.Announcing the ADVANCED ENCRYPTION STANDARD (AES).November 26, 2001:16～17

[41] 黄敦锋.AES 加密解密算法的高速 ASIC 设计:[硕士学位论文].成都:电子科技大学 2010.4: 23～25

[42] 中华人民共和国国家标准化管理委员会发布.GB/T 17964-2008 信息安全技术 分组密码算法的工作模式[M].北京: 中国标准出版社, 2009: 2～13

[43] 风辰翻译.CUDA 编程指南 3.0[Z], 2010.2: 67～76

[44] Tianyi David Han,Tarek s.Abdelrahman.hiCUDA: a high-level directive-based language for GPU programming[R].ACM,2009.3:32～35

[45] A. D. Biagio, A. Barenghi, G. Agosta, and G. Pelosi,.Design of a parallel aes for graphics hardware using the cuda framework[J].2009 IEEE Internaltional Symposium on Parallel and Distributed Processing,2009:5～8

[46] 梁娟娟, 任开新, 郭利财.GPU 上的矩阵乘法的设计与实现[J].计算机系统应用, 2011: 179～180

[47] zhenyu ye. GPU Assignment 5KK70 version 1[H], 2009.11:3～4

[48] 刘进峰, 郭雷.CPU 与 GPU 上几种矩阵乘法的比较与分析[J].计算机工程与应用, 2011, 47(19): 9～11

[49] 肖江, 胡柯良, 邓元勇.基于 CUDA 的矩阵乘法和 FFT 性能测试[J].计算机工程, 2009.35(10): 7～10

[50] Svetlin A. Manavski.CUDA COMPATIBLE GPU AS AN EFFICIENT HARDWARE ACCELERATOR FOR AES CRYPTOGRAPHY[J].2007 IEEE International Conference on Signa Processing and Communications (ICSPC 2007),2007.11:65～68

[51] Hubert Nguyen.GPU Gems 3[M].Addison-Wesley Professional,2007.8:608～610

[52] Gu Q'Gao N'Bao Z Z'et al.Implementation and optimization of hign speed AES algorithm based on GPGPU and CUDA[J],2011.28(6):776～785

[53] Ping Yao,Hong An,Mu Xu,Gu Liu,Yaobin Wang,Wenting Han,CuHMMer:A Load Balanced CPU-GPU Cooperative Bioinformatics Application[R],Proceedings of the 2010International Conference onHigh Performance Computing&Simulatio(HPCS2010).IEEE ComputerSociety Press Caen France,2010.2:32～34

[54] 曾宗雁.高性能计算集群运行时环境的配置优化.科研信息化技术与应用, 2011.11

个人简历、申请学位期间的研究成果及发表的学术论文

个人简历：

马梦琦，男，1988年6月19日出生，2009年7月毕业于桂林工学院计算机科学与技术专业。即将于2012年7月毕业于桂林理工大学信息科学与工程学院计算机应用技术专业。

参与的科研项目：

1.团委创新基金项目《CUDA与Openmp混合并行编程研究》；

2.第十届全国多媒体课件大赛，参与制作《计算机英语》，获得全国高教文科组一等奖。

发表的学术论文：

[1] 马梦琦，刘羽，曾胜田.基于CUDA架构矩阵乘法的研究[J].微型机与应用，2011，30(24)：62～65

[2] 曾胜田，刘羽，马梦琦.基于CUDA的Prewitt算子并行实现[J].微计算机应用，2011，32(11)：71～74

致谢

在这篇论文完成之际，我的研究生生涯即将结束，在此我想对那些给予我关心和帮助的老师、同学、朋友致以最诚挚的谢意！

首先要感谢我的导师刘羽教授，是他把我带入并行计算这个神奇的领域。在这三年中，我深刻体会到了刘老师对大家学习上和生活上的指导和关怀。他对教学工作的热心负责，对科研工作的兢兢业业，无论现在还是将来都是我学习的榜样。在此我要向我最敬爱的导师致以最崇高的谢意！

感谢信息科学与工程学院的老师们，他们为我传道授业解惑，给我提供了良好的学习和生活环境。感谢实验室的兄弟姐妹们，一直以来对我工作和学习上孜孜不倦的指导和帮助，衷心的祝愿实验室的同学们在学院老师的带领下收获更丰硕的成果。感谢我宿舍的好兄弟们，感谢一起打球的兄弟们，是你们的帮助让我变得更加坚强和勇敢。祝愿大家都能实现自身的价值。

我也要感谢分别远在北京、南宁和中山的三位挚友，是你们的鼓励和帮助使我更好的完善论文。

最后，感谢父母这么多年给予我的最无私的关爱，为我创造了一个温馨的家，我的每一点进步都来自于他们背后默默地支持和鼓励。