论文题目 基于 CUDA 架构的 MD5 并行破解算法设计与实现

专业学位类别	工 程 硕 士
学　　号	201092020424
作 者 姓 名	张　奇
指 导 教 师	赵志钦　　教授

分类号 _____ 密级 _____

UDC [注1] _____

学 位 论 文

基于 CUDA 架构的 MD5 并行破解算法设计与实现

（题名和副题名）

张 奇

（作者姓名）

指导教师 赵志钦 教 授

电子科技大学 成 都

白小霞 副研究员

95879 部队 成 都

（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 2012.11.10 论文答辩日期 2012.11.18

学位授予单位和日期 电子科技大学 年 月 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。

DESIGN AND IMPLEMENTATION OF A MD5 PARALLEL CRACK ALGORITHM BASED ON CUDA

A Thesis Submitted to

University of Electronic Science and Technology of China

Major: **Software Engineering**

Author: **Zhang Qi**

Advisor: **Zhao Zhiqin**

School : **School of Electronic 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

计算机技术的不断发展，使得计算机已经逐渐融入了人们的工作与日常生活中。新型的利用计算机进行违法犯罪的案件也越来越多，大量与计算机相关的犯罪案件的调查取证工作都需要获取计算机中的电子证据。在调查取证过程中，调查取证人员往往需要破译相关计算机中的加密数据信息或系统安全口令，这就需要进行大量的密码破解工作。MD5 算法作为一种常用的加密算法应用于很多计算机安全相关的领域。由于 MD5 算法是一种高强度的加密算法，在以往的破解中，一般只是利用 CPU 的运算能力，破解效率很低。

随着计算机图形处理器 GPU 进入通用计算领域，高性能计算机领域中出现了搭载有多块 GPU 显卡作为运算模块的 SMP 集群。为了提高 MD5 算法的破解效率，自然就想到了能否在基于 GPU 的 SMP 集群上实现一个 MD5 高速并行破解系统，充分利用 SMP 集群和 GPU 的运算能力，实现对 MD5 算法的快速破解。

本文详细分析了高性能并行计算技术和 Fermi 架构新型 GPU 以及 CUDA 计算统一设备架构，详细分析了基于 GPU 的 SMP 集群在高性能计算中的优势。通过仔细研究新型采用 Fermi 架构的 GPU 的硬件特点，总结采用 CUDA 编程的方法和优点，确定在基于 GPU 的 SMP 集群系统中使用 MPI-CUDA 的双层混合并行编程模型开发 MD5 高速并行破解系统。以实际需求为牵引详细研究了 MD5 算法及破解方法，分析了破解算法的可并行性，阐述了如何对密码破解计算任务进行分解，设计了破解系统的程序流程。

在本文的实现部分，具体描述了在基于 GPU 的 SMP 集群系统上如何利用 MPI-CUDA 并行混合编程模型进行编程。最后通过实验对系统进行测试，发现系统中的不足，并对破解系统进行优化，并总结分析了系统采用 MPI-CUDA 编程模型的优缺点和程序优化方法。最终验证结果是采用该系统 MD5 算法破解效率比传统的基于 CPU 的破解算法提高了 35 倍，优化后提高到约 95 倍。

关键字： GPU，SMP 集群，CUDA，MD5 算法，密码破解

ABSTRACT

With the continuous development of computer technology, the computer has been gradually integrated into people's work and daily life. Due to more and more computer crimes cases, a large number of computer-related criminal investigations and evidence gathering are needed to obtain electronic evidence in computer. In the process of investigation and evidence collection, investigators often are required to decipher the encrypted data or system security password, which requires a lot of password cracking work. As a commonly used encryption algorithm, MD5 algorithm isused in many areas of computer security. Because the MD5 algorithm is a high-strength encryption algorithm, the crack in the past which just used the computing power of the CPU, was inefficient.

With the computer graphics processor GPU entering into the field of general-purpose computing, a SMP cluster which equipped with a number of blocks GPU graphics computing module came out in the field of high-performance computers. In order to improve the efficiency of the MD5 algorithm crack,we naturally consider whether the MD5 high-speed parallel crack system can be realized on the GPU-based SMP cluster to take full advantage of SMP clusters and GPU computing power to achieve a rapid crack of the MD5 algorithm.

This article analysizes in detail the high-performance parallel computing technology, the new Fermi architecture GPU and CUDA Compute Unified Device Architecture, then it futhur analysizes the advantages of the GPU-based SMP cluster in high-performance computing. Through a careful study of the new GPU hardware characteristics which adopts the Fermi architecture, we summarizes the methods and the advantages of using CUDA programming to determine the MPI-CUDA GPU-based SMP cluster system using double-hybrid parallel programming model developed the MD5 high-speed parallel crack system. To cater for the actual demand,we study in detail the MD5 algorithm and its crack,then analysize the parallelizability of the cracking algorithm,elaborate how to dismantle the password cracking task, and design

the program flow of the crack system.

In the implementation part,this article specifically described how to take advantage of the MPI-CUDA parallel hybrid programming model to programme on the GPU-based SMP cluster system. Finally, through the experiment we test the system, found the deficiencies in the system, then optimize the cracking system, and summarize and analyze the advantages and disadvantages of the MPI-CUDA programming model and its program optimization method. The final validation results is that the efficiency of using the MD5 algorithm to crack the system is 35 times than traditional CPU-based crack algorithm, and up to 95 times after optimization.

Keywords: GPU, SMP cluster, CUDA, MD5 algorithm, password cracking

目 录

第一章 绪论

1.1 研究背景及重要意义

日新月异的计算机、通信和互联网络等信息技术，使我们正飞速的进入一个全球化、信息化的时代。每天都有海量信息在互联网络上传输，其中含有大量涉及政治、经济等重要领域以及普通人日常生活中的敏感信息[1]。信息安全问题日益突出。现代信息密码技术的发展，成为了解决信息安全问题的基础。利用密码技术可以实现消息的保密传输，保障数据的完整性和真实性以及对发送者的身份进行认证等[2]。目前采用的密码技术主要有：加密算法、安全散列函数（哈希函数 Hash Function）和数字签名等。安全散列函数是密码技术中一类具有重要作用并应用相当广泛的信息安全保密算法。在现代信息安全领域它是实现安全可靠、快速有效进行数字签名和身份认证的工具，也是保障网络安全的众多安全协议中有很多重要的模块中也应用了安全散列函数。现在应用最为广泛的安全散列函数有 MAC 算法、MD4 算法、MD5 算法[3]和 SHA-1 算法[4]。快速破解采用 MD5 算法加密获取的散列值，进而破解加密信息，在网络安全与新型计算机犯罪取证[5,6]中具有重要的现实意义。

近几年，大大超过摩尔定律的高速发展，使得 GPU 已经发展成为内存带宽极高和浮点运算能力巨大的并行众核处理器，其强大的并行运算能力不但促进了图形图像渲染处理等相关应用领域的快速发展，同时 GPU 也为人们提供了良好的硬件平台进行通用计算。最近几年间，GPU 的性能以远超 CPU 的速度快速增长。2003 年 Intel 公司和 Nvidia 公司出品的最顶级 CPU 和 GPU 有着相近的计算峰值，而当前 Intel 公司和 Nvidia 公司出品的最顶级 CPU 和 GPU 的计算峰值却相差了近十倍——GPU 是 CPU 的十倍，并且 GPU 的存储器带宽也已经是 CPU 的五倍以上。从数据层面来说，GPU 在一些可以进行大规模并行处理的计算问题上的处理能力已经超越了 CPU[7]。现在 GPU 除了运用于传统的图形图像的渲染处理等领域以外，已被广泛的应用于一个新的研究领域：基于 GPU 的通用计算（GPGPU，General-Purpose computational on GPU）[8]。目前 GPGPU 已广泛应用于代数计算[9]、数据库处理[10]、信号处理[11]、模式识别[12]、仿真技术等非图形应用领域，一些商

业化应用中也出现了 GPGPU 的身影，诸如：数据深度挖掘工具[13]和信息智能化处理系统[14]等。

为了使得 GPU 的计算性能能在通用计算领域中有更好的应用，Nvidia 公司于 2007 年 6 月正式推出了一种全新的软硬件架构：计算统一设备架构（Compute Unified Device Architecture），简称为 CUDA[7]。这是第一种可以使用 C 语言而不是使用图形学 API 函数进行开发的适用于通用计算领域的开发环境和软件体系。与以往的传统的 GPGPU 开发方式相比，CUDA 有十分显著的改进，它抽象表示了 GPU 的结构和 GPU 中的资源，并且向用户提供了访问 GPU 资源的接口和方法。开发者从而可以依据 GPU 的抽象结构，充分利用 GPU 中的所有资源设计实现通用计算领域中的应用[15]。GPU 的计算能力由于计算统一设备架构技术 CUDA 的出现而得到了充分的释放。

另一方面，近年来提出了一种新的高性能的并行计算方式--集群计算[16]。如果解决一个问题需要的计算能力非常巨大，如何将这个问题分割成若干小的任务，然后把这些小的任务通过网络通信分配给依靠高速网络连接的集群中的多台计算机进行处理，最后综合这些计算结果得到最终的结果，这就是集群计算技术。集群计算也是一种分布式计算。随着普通桌面台式计算机运算能力不断提高，如何将多台桌面台式计算机组建为集群，如何在集群中充分利用桌面台式计算机的运算资源是目前在高性能计算研究领域中的又一热点。

近年来，计算机技术的不断发展，计算机已经走进了千家万户，已经逐渐融入了人们的工作与日常生活中。新型的利用计算机进行违法犯罪的案件也越来越多，在这种类型的案件的调查取证工作中都需要获取罪犯计算机中的电子证据[17]。为了防止相关的电子证据被罪犯销毁，这类案件的调查取证工作往往需要在不惊动罪犯的情况下进行。在调查取证过程中，调查取证人员往往需要破译相关计算机中的加密数据信息或系统安全口令，这就需要进行大量的密码破解工作。MD5 算法作为一种应用广泛的密码算法，在以往的调查取证工作中主要是利用计算工作站对其进行密码穷举攻击。这种计算工作站往往只是单纯地利用 CPU 的计算能力通过串行计算模式来实现。在办案人员的电子取证过程中，将耗费相当长的时间来进行密码破解，这严重阻碍了违法案件的侦破工作。因此，快速密码破解在侦查办案中具有特别重要的意义。目前，在市场上已有的一些密码穷举攻击系统多为国外产品，且大部分密码穷举攻击系统都单纯依赖于传统 CPU 的运算能力，它们所能达到的密码破解速率十分有限，在实际的计算机安全取证过程中并不十分有效[18]。随着现代网络技术和 GPU 技术的快速发展，分布式计算和 GPU

并行计算以其在解决大规模复杂运算问题过程中表现出的强大性能优势引得了人们的广泛关注。为此，我们提出并在基于 GPU 的 SMP 集群上实现了一个并行 MD5 密码穷举攻击系统。我们的系统能够利用 SMP 集群中大量配置有 GPU 芯片的计算机同时对 MD5 散列值进行密码学分析工作，从而快速穷举出经过 MD5 算法加密的密码。在我们系统的运算客户端，我们将主要的 MD5 密码计算分析模块写入到 CUDA 内核程序中，安排在 GPU 上进行并行执行，从而使本系统对 MD5 密码的穷举攻击效率得到显著提高。最终的性能测试结果表明，我们在基于 GPU 的 SMP 集群上实现的并行 MD5 密码分析系统能够有效提高对 MD5 密码的穷举攻击速度，与单纯利用 CPU 计算能力实现的 MD5 密码穷举攻击软件相比在性能上有很大的优势。

本文重点研究了利用 GPU 通用计算技术和集群并行计算技术在穷举遍历破解 MD5 算法散列值中的应用，并通过结合二者的技术，提出并设计实现了一种在基于 GPU 的 SMP 集群上运行的并行 MD5 密码分析系统。该系统充分利用 GPU 的通用并行计算能力和集群并行计算的优点，通过大规模并行密码学分析，快速分析破解 MD5 散列值，其破解速率相比于普通的现有的密码破解产品有了大幅度提升，且通过充分利用计算资源，降低计算成本，使得该系统在网络安全与新型计算机犯罪取证等实际应用领域具有广泛的应用价值。

1.2 国内外研究现状

GPU 已成为目前普通桌面计算机所拥有的高性能、低能耗、高性价比的计算资源，在计算密集型问题求解和并行处理等方面拥有诸多优势。随着各种支持 CUDA 的 GPU 的在普通桌面计算机领域的普及，并行计算领域的研究重点将集中在 GPU 的高速并行计算能力如何应用于通用计算，多台具有支持 CUDA 的 GPU 的对称多处理器计算机（symmetrical multi-processor computer，SMP）如何搭建成计算集群，以及如何有效整合这种 SMP 计算集群的计算能力等方面。

近年来，世界各国的研究人员对利用 GPU 进行通用计算获取性能提升已经开展了很多工作，并取得了丰硕的成果：

2000 年，Hopf 等人首次将 GPU 运用到了通用计算领域，小波变换算法在 GPU 上成功实现[19]。

2001 年，Larsen 在 GPU 上利用多纹理图像处理技术成功实现了矩阵的运算。

Thompson 等人利用 GPU 实现了一些常用的数学应用，包括矢量运算和矩阵乘法等，并对 GPU 和 CPU 的运算效能进行了比较，提出了扩展 GPU 的并行架构以提高 GPU 的通用计算能力[20]。

2002 年，Harris 对在 GPU 上如何实现了各种物理现象的仿真进行了研究，提出了相应的算法模型。Purcell 研究了可以在 GPU 上运行的光线跟踪算法，成功地获得了性能的提升。

2003 年，在 GPGPU 领域是具有重要意义的一年。Kruger 等人利用 GPU 实现了向量和矩阵运算等线性代数操作[21]；Li 实现了 Lattice Boltzmann 的流体仿真[22]；Lefohn 实现了 Level Set 方法；Falcao 等人成功在 GPU 上实现了对 LDPC 纠错码的加速，他们的实验结果表明在 GPU 上的运行速度比在 CPU 上的运行速度提升了 700 倍[23]，性能的大幅提升使得专用的硬件解码器从此淡出 LDPC 码的应用领域。

利用 GPU 的高性能来对密码函数算法进行加速也有一段历史，甚至比将 GPU 用于通用计算领域的时间还要早。最早在 1999 年，成功利用专门定制的“PixelFlow”架构的 GPU 加速了对 DES 和 RC4 密码的穷举遍历破解[24]。Harrison 和 Waldron 先是用 OpenCL 在 GPU 上实现了 AES 加密算法[25]，虽然由于种种原因，AES 算法没有能被成功加速，但是他们证明了 AES 加密算法是可以使用 GPU 来实现的；后来 Harrison 和 Waldron 在 CUDA 平台上利用 GPU 成功实现了对 AES 加密算法的加速，算法的实现速度提升了约 4 倍[26]。Manavski 也在 CUDA 平台上成功实现了 AES 算法，通过不断对算法的优化，他成功将算法进行加密的速度提升了 20 倍[27]。Moss 等人用 OpenCL 在 GPU 上通过将数学算法转化成图形运算，成功实现了 RSA 算法，算法加密的速度提升了 3 倍[28]。

2009 年，美国宣布将采用 Nvidia 公司最新一代的采用 fermi 架构支持 CUDA 的 GPU 来研制超级计算机[29]。我国在 2010 年成功研制的“天河一号 A”，采用了 7168 块 Nvidia Tesla M2050 高性能 GPU 计算卡，成为了当时全球排名第一的超级计算机。

CUDA 是一种充分利用 GPU 的计算能力的软硬件体系，在 CUDA 体系中 GPU 被作为处理数据进行并行计算的设备。迄今为止，CUDA 已经进行了数次版本的提升，现在最新的版本已经开发到了 4.1，新硬件的特性得到了更好地支持，不断地完善了各种功能。

集群计算也可以归纳为一种形式分布式计算。分布式计算是将一个需要大量计算能力进行处理的计算任务和海量数据分割成若干小的任务和小的数据块，然

后分配给由高速网络连接的多台计算机进行分别计算，最后综合这些计算结果得到最终的结果的一种计算模式。与超级计算机相比，分布式计算拥有超高的性价比。

目前 MPI（Message Passing Interface，消息传递接口）是应用最广泛的分布式计算程序开发工具之一。MPI 是一个免费的函数库，函数库中的所有函数都是公开源代码的， C/ C++ / Fortran77 /Fortran90 等几乎所有编程语言中都可以调用函数库中的函数。MPI 是一种基于消息传递的编程模型，在并行计算领域中有着广泛的应用，特别在使用分布式存储模式的并行计算集群中非常适用。这种基于 MPI 的纯消息传递模式的并行程序设计模型虽然能将分布式的计算集群的计算能力整合成一个整体,但 SMP 集群中每个节点内采用共享式内存的优势却得不到充分地发挥。而 CUDA 的程序是在共享内存模型下开发的，在这一方面有着与生俱来的优势。近年来，已经开展了一些适合 SMP 计算集群的程序设计模型和编程方法的研究，一种 SMP 集群混合程序设计模型获得了较多的关注。由于 MPI 是适合采用分布式存储模式的程序设计模型,而 SMP 集群采用的就是节点间分布式存储的模式；OpenMP 是适合共享存储的程序设计模型，而 SMP 集群的节点内就采用的是共享存储模式；故而 MPI+OpenMP 是一种适合的混合并行编程模型。研究认为该模型将消息传递模型的优势和共享存储模型的优势进行了有机的结合，更加贴近于 SMP 集群的体系结构。

MD5 是一种信息摘要算法，全称是 Message-Digest algorithm 5，主要用于对文件和消息进行摘要运算以便检测其完整性以及未被篡改。此外，它还大量运用于口令字网络验证传输等。MD5 摘要算法在 RFC-1321 中有着完整的表述。

近年来，对 MD5 算法的研究主要集中在如何使用 GPU 等高性能的硬件对算法的实现进行加速和对 MD5 的算法攻击方法的研究实现两方面。

在对 MD5 算法的加速实现又分为两方面，利用硬件 FPGA 进行加速和利用 GPU 进行加速两方面。2005 年，Kimmo Jarvinern 在 FPGA 上成功实现了 MD5 算法，利用 10 个并行运算模块，实现了 5.8Gbps 的数据吞吐量[30]。2008 年 ELCOMSOFT 在 GPU 上成功实现了 MD5 算法，在 Nvidia 8800GTX 上实现了 115MBps 的数据吞吐量[31]。

在对 MD5 算法的攻击方法的研究和和实现方面，近年来主要取得了以下这些成果。1993 年，DenBoer 和 Bosselaers 首次发现了 MD5 算法的散列值出现碰撞的情况，两组不同的字符串经过 MD5 算法生成了相同的 MD5 散列值[32]。1996 年，H.Dobbertin 在对 MD5 算法的研究中发生了一个字符串同两个同样定长的字

符串发生了 MD5 散列值得碰撞[33]。2004 年我国著名密码学专家王小云教授宣布成功破解 MD5 算法的消息引起了国际密码学界的广泛关注[34]。然而，迄今为止，王小云教授提出的 MD5 算法破解方案依然未能在实际中得到成功应用。因此，如何真正快速实现 MD5 密码分析仍然是密码分析学中一个很热门的研究课题。正是因为这个原因，现在使用最多的攻击 MD5 的方法主要有：穷举攻击法、彩虹链表攻击法。穷举攻击法是通过指定字符集和口令长度，穷举指定长度内所有指定字符的组合作为 m'，判断 MD5(m')是否等于目标 MD5 散列值，直至二者相等，即得到 MD5 散列值的明文口令 m，MD5 散列值破解成功。彩虹链表攻击法是预先将通过穷举攻击法得到的 MD5 散列值明密对构造成一张有序的一一映射的表，并使用压缩、排序等方法减少表的存储空间，使用特定的搜索、匹配算法减少搜索时间，最终在已知明密对应表中搜索到目标 MD5 散列值对应的原始明文的方法。由于彩虹链表是存储穷举得到所有明文、密文对，故而彩虹链表是非常庞大的，通常需要 TB 级的磁盘容量。彩虹链表的破解效率主要取决于高效性的搜索算法和大容量的彩虹链表。

2007 年，Marc Stevens 等人指出 MD5 算法可被重复性攻击[35]。2010 年，乐德广等人分析了以 GPU 作为运算设备并行攻击 MD5 算法的算法原理，设计并实现了基于 GPU 的 MD5 算法高速密码攻击系统，该系统能有效加快 MD5 安全散列值的破解速度[36]。2011 年，张润梅和王霄在对 CUDA 架构进行深入研究的基础上，设计了一种快速破解 MD5 的方法，使用 VS2005 和 NVCC 进行混合编译，使得 MD5 散列值的破解速度 GPU 上比 CPU 上实现了 100 倍左右的提升，大大缩短了破解耗时[37]。

1.3 主要研究内容

利用 GPU 强大的运算能力和并行处理能力，将多台具有支持 CUDA 的 GPU 的桌面台式计算机建成一个 SMP 集群，建立利用 SMP 集群和 GPU 进行通用计算的框架，进而在此框架下实现 MD5 散列值的高速破解。

本文将首先分析 SMP 集群的体系结构，进而分析 GPU 的体系结构，在此基础上建立基于 GPU 的 SMP 集群，建立 MPI+CUDA 的混合并行编程模型环境；针对现有的 MD5 穷举破解算法，研究其在上述环境下的设计实现与优化方法；分析基于上述环境的 MD5 穷举破解算法的性能，测试 MPI+CUDA 的混合并行编

程模型的运行效果。

本文的具体研究及简单分析如下：

（1） 建立适合基于 GPU 的 SMP 集群的通用并行数据处理框架

SMP 集群体系结构也称作 CLUMPS（Cluster of MultiProcessors），其结构特点是节点内基于共享存储，节点间基于分布式存储，有利于集群的扩展和运算性能的提高，且相比于传统的高性能计算机拥有更高的性价比。

GPU 拥有与 CPU 不同的体系架构，拥有比 CPU 更强的数据计算能力。CUDA 是将 GPU 的数据计算能力应用于通用计算的平台，组建基于 GPU 的 SMP 集群就是为了充分发挥整个系统的性能，提供高效、低耗的高性能计算解决方案。

为了实现通用数据并行处理，必须针对集群的体系架构特点，以充分深入理解集群系统中数据流的处理方式为基础，建立适合于集群系统的通用并行数据处理框架，为开发 MD5 散列值大规模并行处理破解程序奠定基础。

（2）MD5 散列值遍历破解算法在基于 GPU 的 SMP 集群的并行数据处理框架下的实现

MD5 散列值穷举破解具有单个破解过程独立，可并行化程度高的特点，而支持 CUDA 的 GPU 具有强大的并行处理能力，因而采用 GPU 为基本破译单元的 SMP 集群进行 MD5 散列值破解具有强大的优势。对于 MD5 穷举破解算法，其并行性是显然的，每个口令都要经过相同的正向 MD5 加密过程处理，最后判断其正确性。为了充分利用基于 GPU 的 SMP 集群的并行数据处理能力，将 MD5 穷举破解算法进行并行性优化很有必要，并针对基于 GPU 的 SMP 集群的物理架构和数据并行处理特点，移植到基于 GPU 的 SMP 集群的并行数据处理框架下。

（3）基于 GPU 的 SMP 集群的并行数据处理框架下 MD5 散列值穷举破解算法的性能分析

针对 MD5 散列值穷举破解算法，测试其在基于 GPU 的 SMP 集群的并行数据处理框架下的破解速率等性能指标，由此分析利用基于 GPU 的 SMP 集群进行 MD5 散列值穷举破解算法的性能。

1.4 论文组织结构

本文内容共分为 6 章，每章的主要内容如下所示：

第一章“绪论”。首先简要介绍了利用计算机图形处理器 GPU 进行通用并行计

算技术的发展历史，并以此为基础阐述了利用基于 GPU 的 SMP 集群高速实现 MD5 散列值密码破解的研究背景和意义。总结回顾了近年来国内外研究人员利用 GPU 进行通用计算的历史，以及利用 GPU 对密码函数的加、解密进行加速所取得的成果，重点介绍了对安全散列函数 MD5 加速和攻击领域的研究和发展现状。最后介绍了本文研究的主要内容和特点，以及本文的结构安排。

第二章“并行处理技术介绍”。简要介绍了现在主流的并行处理技术：SMP 技术、MPP 技术和集群技术，以及利用 SMP 技术和集群技术组建的 SMP 高性能计算集群的优点。

第三章“基于 GPU 的 CUDA 编程技术”。详细介绍了当前 Nvidia 公司的新一代采用 Fermi 架构的高端 GPU 的硬件结构以及 CUDA 的软硬件体系和编程模型。

第四章“利用 MPI 和 CUDA 开发多粒度 MD5 散列值并行破解程序”。首先介绍了 MD5 安全散列算法及其破解原理，提出在利用 GPU 搭建的 SMP 集群上采用 MPI 和 CUDA 混合并行编程模型开发 MD5 快速破解系统的方案，并进行了实现。

第五章“实验结果与分析”。本章是在完成第四章设计的基础上，对所设计的系统在一个实际的基于 GPU 的 SMP 集群上进行测试运行，并分析结果，在此基础上对系统进行优化，最终完成了一个经过优化后运行效率大幅提高的破解系统。并对 MPI-CUDA 的两层混合并行编程模型的优缺点和 CUDA 上实现并行程序优化的方法进行了总结。

第六章“全文总结及工作展望”。对本文的研究和设计内容进行了总结，指出了工作的不足和未来的研究方向。

第二章 并行处理技术介绍

当今世界计算机硬件技术飞速发展,越来越多的晶体管被集成在单个芯片上。但随着集成度的不断提高，单个芯片的耗电量和热效应也越来越大，对整个系统的影响也越来越大，这就使得提高单个 CPU 核心的运算能力越来越困难。因此采用各种并行方式提高计算能力势在必行。目前，主流的并行架构体系技术主要有 SMP 技术、MPP 技术和集群技术等。

目前，按照采用何种存储体系，共享式存储体系还是分布式存储体系，高性能计算机(HPC)被分成了两大类。当前国内外主流的并行计算机类型包括对称共享式存储多处理计算机(SMP)、大规模并行计算机(MPP)和 SMP 集群等。其中 SMP 体系结构基于共享式存储体系，具有延迟低、带宽高的特点，可以使用多线程并行编程机制，但体系的可扩展性较差；MPP 是基于分布式存储的体系结构，处理器间通信主要采用 MPI、PVM 等消息传递机制，系统体系的可扩展性很好，但由于处理器间采用消息传递机制，故而通信开销较大，系统可编程性较差。目前，国内外在并行计算机领域的主流研究方向是组建采用多层次并行的系统体系架构的并行计算机，比如 SMP 集群。SMP 集群的系统体系结构特点是集群中存在节点间和节点内的两级并行处理架构，节点间分布和节点内共享的两级存储体系结构，综合了 SMP 体系和 MPP 体系的优点。此外，单个处理器内部还存在另一级并行性，即指令级并行(Instruction-Level Parallelism,ILP)。单个处理器对数据的处理能力直接影响整个系统并行化执行程序的效能。

2.1 SMP 技术

SMP（Symmetrical MultiProcessing，对称多处理）技术是单处理器技术的简单扩展，在总线上连接多个 CPU，所有 CPU 共享同一个主存（即共享内存）。该体系结构也被称为内存一致性访问的多处理器（UMA）。SMP 技术在并行计算架构中应用十分广泛。在 SMP 体系中，一个操作系统同时管理位于一台计算机内的多个 CPU，这些 CPU 共享内存和 I/O 等其他资源，它们的地位平等，平均分配工作负载。SMP 体系结构的系统对用户是透明的，虽然计算机中多个 CPU 同时工

作，但对用户管理来说，它们就像一个整体和单 CPU 计算机并没有区别。SMP 体系结构的系统将任务队列对称地分配给系统中的多个 CPU，利用多个 CPU 进行并行处理，使得整个系统的数据处理能力得到了极大地提升。SMP 系统体系框架如图 2-1 所示。

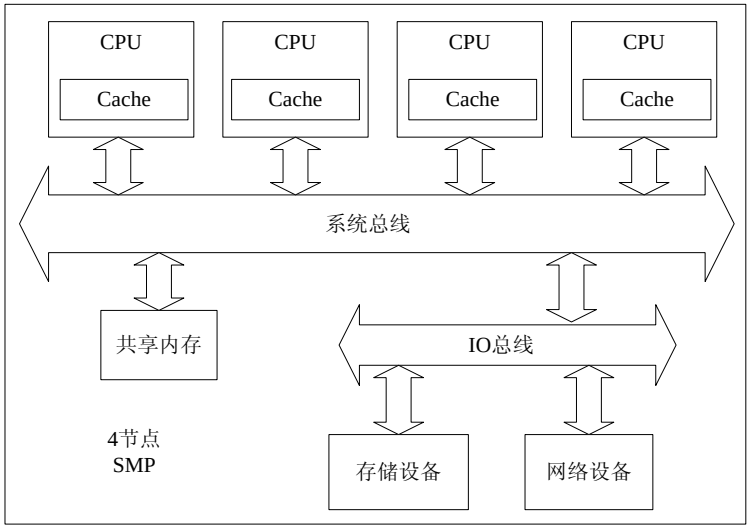

图 2-1 基于 SMP 技术的系统框架

SMP 系统的对称多处理是系统中的 CPU 通过共享高速系统总线来实现的。同时系统总线的带宽也限制了 SMP 系统的可扩展性。两个主要的难点问题限制了在 SMP 系统中增加更多的 CPU，一是内存竞争会消耗大量系统资源，二是 Cache 一致性。

内存竞争是多个 CPU 同时访问内存中同一地址的数据时，它们只能同时读数据，但不能同时修改数据。如果一段内存中的数据正由某一个 CPU 进行修改，这段内存就会由这个 CPU 锁定，其他 CPU 只能等待这段内存解除锁定后才能进行操作。显然，这种等待问题的严重性是随着 CPU 数量的增加而增加的，不仅使系统性能无法得到提升，甚至会导致系统性能的大幅下降。为了能在 SMP 系统中尽可能多的增加 CPU，一般都通过增大 CPU Cache 的容量来缓解内存竞争问题。Cache 是 CPU 私有的高速缓存，数据以远高于内存总线的速度在 CPU 与 Cache 之间进行交换。又由于 Cache 的私有性，不提供共享，多个 CPU 就无法竞争同一个 Cache 资源，在 CPU 内置或外置的 Cache 缓存中就可以顺利完成许多相关的数据操作。

然而，虽然 SMP 系统中的内存竞争问题通过设置 Cache 缓存的方式得到了解决，但另一个较难解决的问题——Cache 的一致性问题，却也伴随着 Cache 的出现而出现。在 SMP 系统中，各 CPU 访问内存中的数据时是通过 Cache 来进行的，

因此内存中的数据必须经常与 Cache 中的数据保持一致，若系统更新了 Cache 中的数据，也必须相应地更新内存中的数据，否则系统中数据的一致性就会被破坏。由于需要通过占用 CPU 来进行系统中数据的更新，并且需要更新的内存字段也必须被锁定，故而过高的更新频率必然会导致整个系统的性能受到影响，但间隔时间过长的更新却有可能会导致因系统交叉读写而造成的数据错误。因此，一个高效率的更新算法对 SMP 系统显得尤为重要。目前 SMP 系统中采用的更新算法通常为侦听算法。Cache 缓存越大，系统中出现内存竞争的概率就越小。同时由于 Cache 缓存与 CPU 之间拥有较高的数据传输速度，增大 Cache 缓存同样也使得 CPU 的运算效率得到了提高，但在整个系统中保持 Cache 一致性的难度也同样加大了。

在硬件方面，SMP 可以在 Alpha、SPARCserver、PowerPC 以及 UltraSPARC 架构上实现，也可以利用 Intel 公司所有 486 以上的芯片来实现，同样也能在基于 GPU 的架构上实现。

2.2 MPP 系统

在诸如科学计算、信号处理、工程模拟和数据仓库等应用中，为了更高的利用并行性，SMP 系统的能力已经不能满足要求，我们需要使用可扩展性更高的并行计算机平台，这可以通过采用分布式存储体系结构的 MPP 来实现。

MPP（Massively Parallel Processors，大规模并行处理机）系统由大量计算节点所构成，各计算节点具有各自独立的局部存储器，系统中并不实际存在全局共享的存储器，整个系统由一个主操作系统进行管理，而各个计算节点内则各有一个子系统与其配合。由于 MPP 系统中的计算节点具有各自独立的局部存储器，因而系统可扩展性高。MPP 系统的最大结构特点是不提供资源共享。

MPP 系统中采用 NORMA（非远程）存储访问体系结构。所有的存储器都是私有的，并且在物理上是分布的，这是 NORMA 体系结构的特点。每个节点中的 CPU 都只能直接访问本节点内的存储器，访问其他节点的存储器是被禁止的。在 MPP 系统上运行的程序由很多相互独立的进程组成，进程间使用消息传递机制进行通信，每个进程都有私有的进程空间。分布式存储的体系架构使得 MPP 系统拥有良好的可扩展性，但采用消息传递机制导致可编程性较差，编程变得困难，。

MPP 系统中为了得到较高的计算处理性能而使用了大量的硬件。MPP 系统中

采用了很多技术来确保随着 CPU 数量的增加整个系统的整体性能能够以近似线性的方式实现增长。使用分布式的存储器体系结构能够提供比集中式个更高的存储带宽。如果 MPP 系统中 CPU 数目很多，影响系统加速比的一个重要因素是通信系统的开销较大。因此，MPP 系统采用了专门的带宽高、延迟低、速度快的互联网络。MPP 系统中包含了大量的 CPU、存储器等硬件，大大提高了系统出现故障的概率。因此，MPP 系统必须采用高可用性技术，系统不会因为部分部件的失效而造成整个系统的故障。同时，失效的 CPU 的任务应该可以被保存，并可以使用其他节点的 CPU 继续进行处理。

由于 MPP 系统中使用了大量的硬件,控制系统成本就成为了另一个重要的问题。目前降低 MPP 系统成本的措施有：使用商品化的微处理器 CPU 以及 Shell 结构；使用物理上分布的存储器；使用 SMP 节点，有效降低节点内部的互连网络规模。使用商品化的微处理器 CPU 有很多益处。一是成本比独立开发低，二是其升级速度快，升级成本也低。Shell 结构是一个专门设计的接口电路。微处理器 CPU 通过 Shell 结构与系统节点内的其他硬件相连。当节点升级时，只需要更换 CPU 和 Shell 结构，系统和节点内的其他硬件部分不用改变。使系统升级变得非常方便，同时节约了升级成本。使用 Shell 结构，MPP 系统可以随着商品化的微处理器 CPU 的升级进行快速升级。

目前的 MPP 系统都使用相同的操作系统，支持多种种类的算法和应用；支持消息传递编程模式 MPI 和 PVM。

总之，MPP 系统与其他系统相区别的最主要的特点是拥有大量的 CPU 处理器。大量存在的 CPU 处理器造就了 MPP 系统巨大的运算能力，但也导致了系统中的许多问题和技术实现的困难，例如高昂的成本、节点间通信的困难。MPP 系统的持续运算速度一般只有峰值运算速度的 3%~10%，大部分都是出于通信开销和算法设计等方面的原因。如何能使 MPP 系统的持续速度提高是一个仍需不断研究的问题。

著名的 MPP 系统早期有 Thinking Machine 的 CM2/CM5，NASA/Goodyear 的 MPP，nCUBE，Cray T3D，Intel Paragon，MasPar MP1 等；当今有美国 ASCI（Accelerated Strategic Computing Initiative）提高战略运算能力计划中的 MPP 系统：Intel 公司与 Sandia 国家实验室联合研制的 Option Red；IBM 公司与 Lawrence Livermore 国家实验室联合研制的 Blue Pacific 和 SGI 公司与 Los Alamos 国家实验室联合研制的 Blue Mountain。IBM SP2、Intel TFLOPS 和我国的曙光-1000 等都是 MPP 系统。

2.3 集群技术

长期以来，数据中心、科学计算等领域长期被高端 RISC 服务器占据，用户只能选择 IBM、SGI、SUN、HP 等公司的产品，不但价格昂贵，而且运行、维护成本高。随着电子商务和 Internet 服务的迅速发展，计算机系统在现实生活中已经越来越重要，要求服务器拥有越来越高的可扩展性和高可用性。RISC 系统高昂的代价和社会旺盛的需求形成了强烈的反差。

集群（Cluster）技术的出现和 IA 架构服务器的快速发展为社会的需求提供了新的选择。采用集群技术可以构造价格低廉、易于使用和维护的超级计算机，其拥有的强大的数据处理能力，成为了替代代价格昂贵的大中型机的一个高性价比的选择。目前，在世界各地正在运行的超级计算机中，有许多都是采用集群技术来实现的。

2.3.1 集群技术概述

集群（Cluster）技术是近几年逐渐兴起的一项新技术，这项技术的提出主要目的是为了发展高性能计算机。集群系统是一组计算机，彼此独立，利用高速通信网络连接在一起，共同组成的一个并行或分布式的计算机系统，并通过单一系统的模式进行管理。一个集群由多台通过高速互联网进行通信的各自拥有私有的共享数据存储器的服务器组成，这些服务器在集群中一起工作，共同运行一些应用程序，并为用户提供单一的系统接口。从宏观上看，它们组成的是一个整体，共同对外提供服务。集群中的所有计算机，在物理上通过一系列高速的物理电缆连接，在逻辑上则通过一系列的集群应用软件连接。通过这些连接，使得集群中的计算机具备负载平衡和应急故障处理功能，而这些功能是无法在单机系统上实现的。应急故障处理功能是当集群中的一台计算机发生故障，无法运行时，集群内的其他计算机将自动接管故障计算机的功能，向外部提供稳定的服务。通常情况下，所有组成集群的计算机是一个整体，拥有一个共同的名称，集群用户可以使用集群中任意一台计算机上提供的服务。一般来说，使用集群系统是为了获得更高的系统稳定性，更高的系统可扩展性，更强大的数据处理能力和服务能力。

上世纪 60 年代，一个新的并行计算系统概念由 IBM 公司提出，这就是集群计算系统。当时集群系统的基本思想是通过高速可靠的物理连接将多台大型计算

机连接成一个整体，使用定制的合理的交互技术，发挥所有计算机的系统效能，共同完成并行计算任务。但在当时由于很多客观条件的限制，尤其是高昂的软硬件成本，使得集群系统很难实现商业化。

直到上世纪 80 年代，随着微处理器技术、网络技术和分布式技术的不断发展，出现了性能高价格低的微处理器，高速的互联网络和具有强大计算能力的分布式并行计算系统，这些共同组成了支持集群系统发展的物质基础。

发展初期的集群系统是单纯的为了获得强大的计算能力。这个时期的集群系统通过不断提高单个处理器的运行、计算速度，增加存储器的容量，连接多个处理器的方式来获得更高的计算性能，这就是早期的并行超级计算机。

早期的并行超级计算机由于受到软硬件资源和成本的限制，未能得到很好的发展，但是集群的理念已经深入人心，众多的研究人员投身于其中。上世纪 90 年代后，随着各种技术的不断发展成熟，集群计算系统也进入了高速发展期，先后研制成功了很多典型的集群系统，其中包括：①Beowulf、②Berkeley NOW、③HPVM、④Solaris MC 四大集群环境。目前，这些集群计算系统已经在科研、工作、学习、生活等各个方面得到了广泛的应用。

集群系统主要有下面这些优点：

高可扩展性：系统的可扩展性强。无需使用专门定制的计算机和网络互连设备，扩展成本低。

高可用性：集群系统拥有应急故障处理机制。集群中的一个节点发生故障，其他节点可以自动继续运行故障节点的任务，保持集群系统的可用性。

高性能：集群系统通过整合多台计算机的处理能力几乎可以获得成线性增长的计算处理能力。在集群系统中，计算机越多性能越高。

高性价比：可以使用高度商业化的普及程度高的计算机软硬件构造高性能的集群系统。

有两种集群架构是最常见的：Web/Internet Cluster System 和平行运算（Parallel Algorithms Cluster System）。前者也叫分布式集群，集群中多台计算机共同负责同一项任务，提供同一项服务，所有资料分布式存放在多台计算机中。后者也叫并行计算集群，集群中的每台计算机都是一个计算节点。一个计算任务提交到集群后，由集群中所有计算节点的 CPU 并行来进行同步运算，完成计算任务。

图 2-2 是典型的集群系统体系结构：

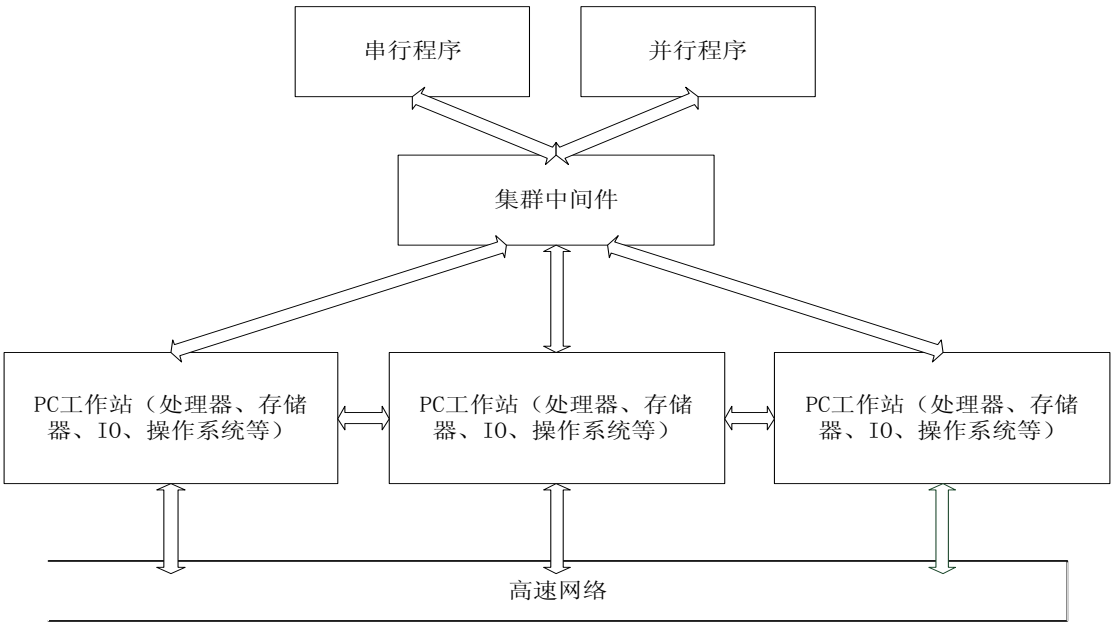

图 2-2 集群系统体系结构

2.3.2 集群硬件平台

2.3.2.1 集群中间件

集群中间件位于集群节点和用户系统环境之间，与集群操作系统共同为集群中所有节点访问系统中的各种资源提供途径，集群系统应该向用户提供完全透明的硬件资源管理和应用系统服务。对用户来说集群系统必须提供良好的可扩展性和可用性。中间件的作用就是在用户不了解系统体系结构的情况下，仍然能够很方便有效地使用系统中的所有资源，而不管这些资源在何处。

以下是中间件必须提供的主要服务：

唯一地址空间：集群系统内所有节点的内存作为整个系统的共享内存，统一进行组织和分配（包括输入/输出设备）。

唯一用户界面：集群用户进行集群操作使用的操作界面是统一的并且是唯一的。

唯一的进程空间：集群系统中的每一个进程可以在系统中任意节点内生成子进程（包括自身所在的节点），并且可以与任意节点上的其他进程进行通信。

唯一作业管理：集群系统的作业提交机制对用户来说是透明的。用户可以在集群中的任意节点向集群提交作业，作业可以采用批处理模式、交互式模式或者并行计算模式在系统上自动调度执行。

检查点设置：设置检查点可以定期保存集群系统中的进程状态和应用程序的中间结果，当有节点发生故障时，故障节点的进程可以很容易地在另一个节点上继续执行，而并不会出错。

进程迁移：采用动态地方式对集群中各个节点间的负载进行平衡。

2.3.2.2 集群中的通信网络

集群中的所有计算机都是依靠通信网络进行连接，故而通信网络是集群中的关键部分。通信网络传输速率、带宽和稳定性直接影响整个集群系统，是集群系统性能提高的主要瓶颈。

故而在集群系统中必须使用开关网络而不是共享网络，即集群中必须使用交换机而不是集线器进行数据交换。集群中通常使用的开关网络有快速以太网、Myrinet、千兆以太网、Server NET 、ATM 等。

网络传输协议在通信网络中是起最主要的作用的。为了说明通信网络在集群中的作用，下面以 Xpress 传输协议为例。

在集群系统中，一个应用程序在运行时可能需要将数据流从一个节点分配到多个节点上。在数据流的分配过程中不同的传输协议使用的数据传输方式也不尽相同，当然会有传输效率的差别。例如传输控制协议 TCP 会在数据发送端与每一个数据接收端建立连接，单独处理每一个数据流；用户数据报协议 UDP 则可以使用报文组播的方式进行数据流的传输，虽然这种方式提高了传输效率，但 UDP 提供的是不可靠的数据传输服务。故而上述两种传输协议都无法满足集群系统低延迟、高可靠的通信需求。而 Xpress（XTP）传输协议支持组播组管理，多播传输服务，连接快速建立和释放，支持高带宽、低延迟的通信服务，因此采用 Xpress 协议进行数据流的传输正好可以满足集群系统的通信要求。

2.3.2.3 集群中的内存管理

集群中内存的组织方式确定了集群的互联体系结构，也确定了集群采用的编程模式。一种是共享内存模式，一种是分布式内存模式。

共享内存模式中集群中所有节点的内存作为一个整体进行全局内存地址编制，这种模式中集群网络对于集群上运行的进程来说是透明的；由虚拟内存管理的硬件和软件将全局内存中的虚拟内存地址映射到本地或远程物理内存中去，即本地的应用程序可以通过虚拟内存映射的方式调用远程的物理内存。

分布式内存模式中，集群通信网络对应用程序来说是可见的，节点间通信使

用消息传递模式。与共享内存模式相比，本地的应用程序无法调用远程的物理内存，系统各节点间只能通过消息传递的方式进行通信。应用程序必须有明确的通信模块执行数据的发送和接受任务。

2.4 SMP 高性能计算集群

目前，高性能计算集群的应用范围越来越广，除了一些特殊的应用需要高性能计算外，一些比如：数据挖掘应用、图像处理业务、基因测序比对处理等领域也越来越需要高性能计算。

但是，想要得到一个高性能解决方案，往往意味着要投入大量的金钱。例如：国内曙光 4000A（10 万亿次运算能力），市场价格约为 1 亿元人民币；曙光 5000A（200 万亿次运算能力），市场价格约为 2 亿元人民币。国外能达到相同计算规模的超级计算机，价格大约是国内的 5~6 倍。一个小规模的高性能解决方案，也需要花上百万乃至上千万的成本投入。

表 2-1 是对多种并行处理技术的比较。SMP 集群提供了一个低成本、高性能的解决方案。

表 2-1 多种并行处理技术比较

	SMP	MPP	Cluster
可扩展性	最差	强	很强
易管理性	很强	一般	较差
网络能力	无需网络	很强	强
并行编程方式	共享变量	消息传递	消息传递
价格	贵	昂贵	较低

通过对大量集群环境配置的经验进行研究，显示小型 SMP 平台最适合构建高性能计算集群，并且现在的多核计算已经达到平民化的程度，更有 GPU 这种众核处理器在通用计算中的成功应用，因此使用多路处理器作为系统节点成为首选方案。SMP 集群有效综合了 SMP 技术和集群技术两种并行技术的优点，同时具备节点内共享式存储和节点间分布式存储的两级内存层次结构，支持节点内采用单地址空间的共享变量和节点间采用分布式存储的消息传递的两级混合并行编程模型。具体的在单节点内采用 GPU 作为并行运算硬件的小型 SMP 集群中，可以采用节点间消息传递编程模型 MPI 和节点内 GPU 共享内存编程模型 CUDA 的两级

混合并行编程模型。

2.5 本章小结

本章主要介绍了当前主流的并行架构体系技术：SMP 技术、MPP 技术和集群技术，并以此为基础介绍了 SMP 高性能计算集群系统。通过对多种并行处理技术的比较，总结了 SMP 集群技术的优点和可以采用的并行编程模型。

第三章　基于 GPU 的 CUDA 编程技术

3.1 CUDA 原理简介

近年来，随着计算机技术的不断发展，桌面台式计算机中的图形处理设备(GPU)已经逐渐演化成高并行度、多线程，具有的内存传输带宽极高和浮点运算能力巨大的并行众核处理器。如下图 3-1 与图 3-2 所示：

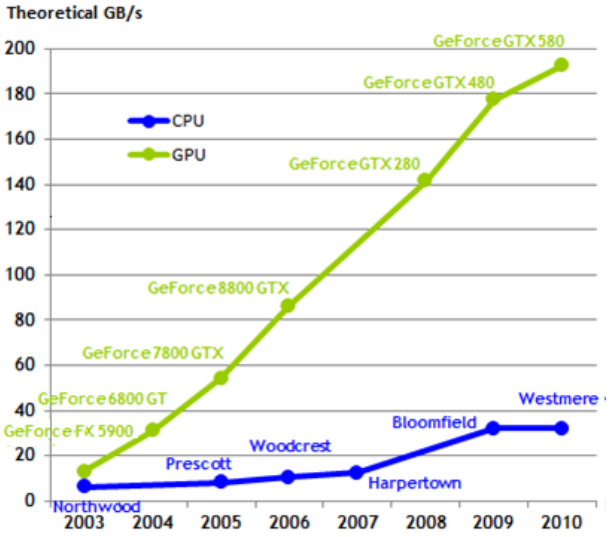

图 3-1 GPU 的内存带宽

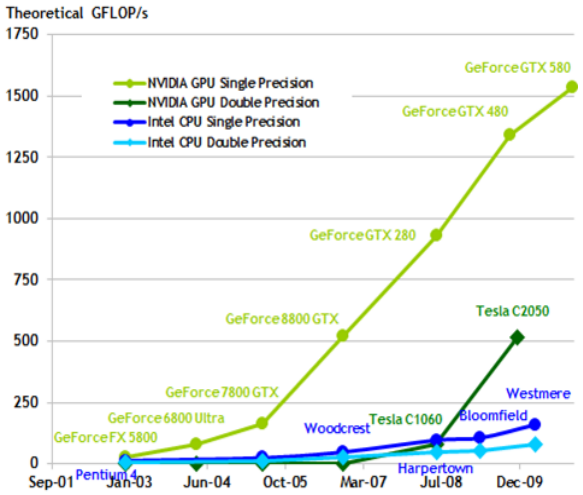

图 3-2 GPU 的浮点运算能力

CPU 和 GPU 的硬件结构的差异造成了它们之间运算能力的差异，如图 3-3 所示，GPU 将芯片中更多的晶体管用来进行数据处理，而 CPU 是用很多晶体管作为数据缓存和进行流控制。因而 GPU 是特别为计算密集度高，具有高并行度的计算（如同图像渲染）设计的。

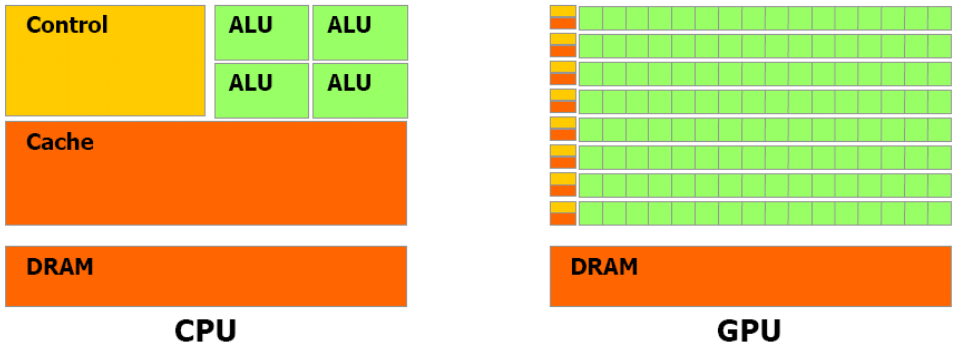

图 3-3 CPU 和 GPU 功能性单元对比

GPU 非常适合处理具有低相互依存度，高可并行度的问题，计算密度非常高。由于同一程序在每个计算部分上执行，因此对负载流控制的要求非常少，更因为在多个计算部分上同时执行并且计算密度高，访存延迟可以被计算隐藏，因此用不着大的数据缓存。

CUDA 全称是 Compute Unified DeviceArchitecture，是 Nvidia 公司在 2007 年推出的一种在 GPU 上进行通用计算的架构。这是一种可以使用类 C 语言而不再需要进行图形的 API 映射进行开发的通用计算开发环境和软件体系。

CUDA 是一个完整的以 GPU 作为运算工具的并行计算架构，包含由底层的硬件部分和基于硬件之上的软件部分。硬件部分是支持 CUDA 计算架构的具有多个图形处理流处理核心的 GPU，软件部分包括编译工具、驱动 Driver API 库、运行时 runtime API 库和高层数学函数库。

在 CUDA 架构下，开发者为了充分利用 GPU 中的硬件资源的并行计算能力，通常采取使用大量的并发线程进行并行计算的方式。这些并发线程的创建和管理在 CUDA 中是依靠硬件来实现的，软件的执行时间并不会被占用。访问 GPU 中硬件资源在 CUDA 中是通过调用接口函数来实现访问的，这些接口函数包含在运行时 runtime API 库中。CUDA 开发环境中提供了 DRMA 内存寻址方式，这是一种在 C 语言中通用的寻址方式。故而 CUDA C 是一种扩展的 C 语言，拥有极强的编程灵活性。操作系统负责管理所有访问 GPU 的程序，不论是基础的图形应用程序，还是利用 GPU 进行通用计算的多个并发运行的 CUDA 程序。

在 CUDA 的编程模型中，CPU 是主机（Host）或称为宿主，GPU 是计算设

备（Compute Device），是 CPU 的协处理器。在 GPU 上可以并行运行多个线程。CUDA 程序由专门的编译器 nvcc 编译后，分成了两个部分：主机部分和设备部分。主机部分运行在 CPU 上，设备部分运行在 GPU 上。主机部分一般为控制程序执行流程的串行程序，设备部分则是负责计算的密集性并行程序。

设备（Device）端程序实际上是由大量的并发线程组成，这些线程由 GPU 硬件调度，在不同的流处理器分布执行。在 CUDA 中，线程（Thread）、线程块（Block）和线程网格（Grid）构成了线程组织的三个层次。由线程块组成线程网格，线程块可以是三种维度的一维、二维或者三维，在不同的流处理器簇上执行不同的线程块；线程块是一维或者二维的，由流处理器簇内不同的流处理器执行同一线程块内的线程。大量的并发线程能够在 GPU 上同时执行是 CUDA 的优势所在，提高 CUDA 应用性能的基本手段是增加并发线程的数目。

3.2 CUDA 硬件架构

3.2.1 Fermi 架构的 GPU

2010 年，Nvidia 公司在原有 GT200 架构 GPU 的基础上推出了最新一代的 Fermi 架构的 GPU 产品，在保持图形处理性能的同时，通用计算性能得到了前所未有的提升。

Fermi 架构的 GPU 是一块单管芯封装芯片，集成度高达 30 亿个晶体管，规模远远超过了当前主流的 CPU。

对 CPU 来说多个核心之间的通信代价并不太大。CPU 的缓存本身就有数据一致性（Coherency&Consistency，简称 CC），即 CPU 内缓存的数据与 CPU 外部任何存储单元的数据保持一致，因此实现几个 CPU 之间的数据一致性也没有什么难度。GPU 的缓存具有只读的特性，因而并不存在数据一致性问题。在一个 GPU 内都不保证数据一致性，多个 GPU 之间的数据一致性就更加无从谈起了。因此，多个 GPU 之间很难实现高带宽低延迟的通信，只有通过系统内存和 PCI-E 总线交换数据，开销非常高昂。单个 GPU 的计算能力较强，就可以使用较少的 GPU 完成同样的任务，有利于减少控制盒通信方面的开销。Fermi 上集成的大量的晶体管体现了 Nvidia 公司将 GPU 用于通用计算领域的决心。

Fermi 架构的 GPU 包含了 4 个 GPC（图形处理团簇），因此是有 4 个核心的

GPU。MC（Memory Controller 内存控制器），SM（可扩展流阵列多处理器）和GPC（图形处理团簇）组成了 Fermi 的三层体系架构。一个完整的 Fermi 架构包括 6 个 MC，16 个 SM 和 4 个 GPC，因此 Fermi 架构的 GPU 可以称为一个 4GPC 核心的 GPU。如图 3-4 所示。

从图 3-4 中可以看到 Fermi 架构核心包括 Host Interface 总线接口、GigaThread 线程调度引擎、4 个 GPC 单元、6 个 MC 内存控制器、6 个 ROP 簇和 768KB 二级缓存。每个 GPC 单元包含 4 个 SM 单元，每个 SM 单元包含两个线程束 warp 调度器。

CPU 的命令通过 Host Interface 总线接口传输到 GPU。GigaThread 线程调度引擎将指定数据从系统内存拷贝到指定的显存。 为获得高带宽和低延迟，Fermi 架构集成了 6 个 64 位（共计 384 位）GDDR5 内存控制器。由 GigaThread 线程调度引擎创建并将线程块 block 分配到各个 SM 单元，在每个 SM 单元中再分配到线程束 warp（32 个线程 thread 组成一个线程束 warp）由 CUDA 计算核心（CUDA Core）执行。当 GigaThread 线程调度引擎重新分配工作时，图形流水线上的各个单元如顶点拾取和细分曲面之类的单元也会继续工作。

图 3-4 Fermi 架构核心示意图

Fermi 架构拥有 512 个 CUDA 计算核心，分别属于 16 个 SM 单元，每个 SM 单元包括 32 个 CUDA 计算核心。每个 SM 单元是一个高度平行处理器，最多支持同时完成对 48 个线程束 warp 的处理 。每个 CUDA 计算核心是一个统一的处理器核心，执行顶点，像素，几何等图形操作和通用计算的 kernel 函数。线程的加载、数据的存取和纹理操作由芯片上大小为 768KB 的二级缓存统一负责。每个 SM 单元包含 4 个纹理处理单元，共享使用大小为 12KB 的一级纹理缓存，并和整个芯片共享大小为 768KB 的二级缓存。每个纹理处理单元均支持新的 DirectX 11 的压缩纹理格式，每个周期可以拾取四个纹理采样、计算一个纹理寻址。

Fermi 架构拥有 6 个 ROP 簇共 48 个 ROP 单元，每个 ROP 簇拥有 8 个 ROP 单元，每个 ROP 簇配备一个内存控制器。每个 ROP 单元执行原子内存和抗锯齿操作。ROP 单元和二级高速缓存、内存控制器是紧密耦合的。整个芯片和所有 ROP 单元共享大小为 768KB 的二级缓存。

Fermi 架构以 GPC 的运行频率作为“核心频率”，几乎所有硬件单元的运行频率都和“核心频率”相关。一级缓存等于“核心频率”，其他包括纹理处理单元、光栅引擎、多形体引擎等都为“核心频率”的一半。

3.2.2 Fermi 架构 GPU 中的 GPC 架构

Fermi 架构的 GPU 的核心是称为图形处理团簇的 GPC。GPC 是 Fermi 架构中的核心硬件模块。每个 GPC 由 4 个 SM 单元和 1 个光栅引擎（Raster Engine）组成。除了计算单元，每个 SM 单元还包括一个负责细分曲面（Tessellation）和属性设定（Attribute Setup）的多形体引擎（Polymorph Engine）。如图 3-5 所示。

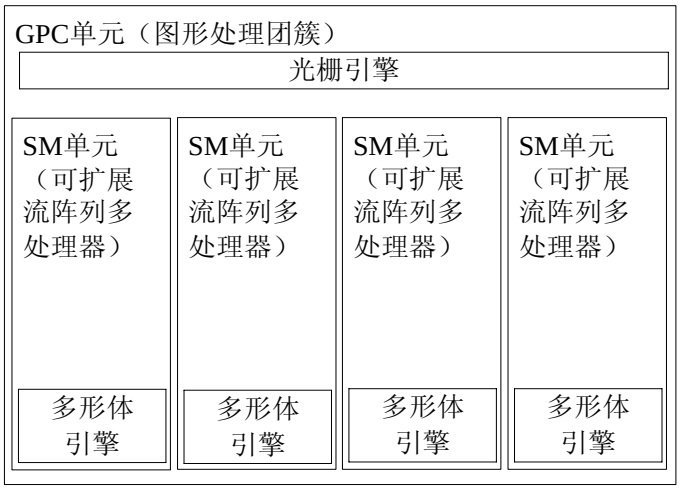

图 3-5 Fermi 架构 GPC 架构图

所有的 GPC 都集成了包括顶点、几何、光栅、纹理均衡设置和像素处理等资源在内的关键的图形处理单元。随着 ROP 单元功能的不断增强，一个 GPC 单元可以被看作是一个配置齐全的 GPU，而 Fermi 拥有 4 个 GPC 核心。

3.2.3 可读写缓存 Cache 和共享缓存 Shared memory

为了充分降低访问存储器时的时间延迟，为计算核心提供更好的服务，有效提高整个计算单元的工作效率，在 CPU 的硬件架构中建立了拥有 L1、L2 和 L3 三级缓存的架构体系。但是在 GPU 的硬件架构中一直没有提出明确的缓存概念，自然也没有建立合适的缓存架构体系。探究这其中的原因主要分为两个方面，其一是因为在 GPU 中存在大量的计算核心，这就要求提供相配套的足够数量的缓存；其二是因为以前是采用图形学 API 进行通用计算开发，这就导致程序中并不一定需要使用到缓存，尤其是传统意义上同 CPU 缓存一样的通用可进行读写操作的缓存。

在 Fermi 架构的 GPU 中，引入了一套灵活实用的二级缓存体系 L1 和 L2。在应用程序开发中，不同的需求需要配置不同的共享缓存 shared memory 和可读写缓存 cache。Fermi 架构通过对存储器资源配置的变化，提供了一种优化的内存设计方案，既提供共享缓存 shared memory 也提供可读写缓存 cache。如图 3-6 所示。

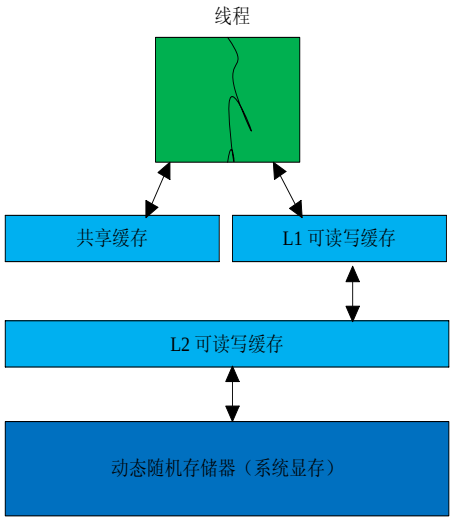

图 3-6 Fermi 架构可配置的二级缓存体系

Fermi 架构中为每个 SM 单元提供了大小为 64KB 的可配置缓存，并提供了两种配置方案，将 64KB 的缓存分成了大小分别为 16KB 和 48KB 的两部分，均可以分别配置为共享缓存 shared memory 或者是一级可读写缓存 L1 cache。

堆栈操作和寄存器溢出均可以使用一级可读写缓存 L1 cache 进行处理。在没有一级可读写缓存 L1 cache 时，如果发生 GPU 的寄存器溢出，则会大幅增加存储器访问时延。有了一级可读写缓存 L1 cache 后，即使应用程序使用大量的临时寄存器，程序的性能表现依然很稳定，能更加准确地控制包括双精度浮点运算在内的多种运算中的精度衰减问题。由于 Fermi 架构中一级可读写缓存 L1 cache 的设计存在，使得物理仿真、光线追踪等不能事先准确预知数据存储访问地址的算法，也都能从中受益。

在 Fermi 架构中设计共享缓存 shared memory，使得程序员可以便捷的实现多线程间的数据重用。程序员可以通过程序实现将共享缓存 shared memory 当成缓存 cache 来使用，并实现数据的读写和一致性管理。而对那些没有使用共享缓存 shared memory 的 CUDA 应用程序来说，也可以直接从使用一级可读写缓存 L1 cache 中受益，程序的运行效率得到大幅提升，运行时间显著减少。GT200 和 Fermi 架构缓存构成形式与容量对比如图 3-7 所示。

	GT200	Fermi	Benefits
L1 Texture Cache (per quad)	12KB	12 KB	Fast texture filtering
Dedicated L1 LD/ST Cache	None	16 or 48 KB	Efficient physics and ray tracing
Total Shared Memory	16 KB	16 or 48 KB	More data reuse among threads
L2 Cache	256 KB (Texture read only)	768 KB (all clients read/write)	Greater texture coverage, robust compute performance

图 3-7 GT200 和 Fermi 架构缓存构成形式与容量对比

Fermi 架构还设计了 768KB 的二级可读写缓存 L2 cache，线程的加载、数据的存取和纹理操作皆由此二级可读写缓存 L2 cache 统一负责。二级可读写缓存 L2 cache 由所有的 SM 单元所共享。二级可读写缓存 L2 cache 支持线程进行高速有效地数据存取。当 CUDA 程序需要所有的 SM 单元去读取相同的数据时，可读写缓存就能提供极大的帮助。

3.3 CUDA 编程模型

3.3.1 主机与设备

在 CUDA 的编程模型中，CPU 是主机（Host）或称为宿主，GPU 是计算设备（Compute Device），是 CPU 的协处理器。一个 CPU 和多个 CUDA 设备可以共同组成一个系统，即单个多核 CPU 加多 GPU 的模式，这样可以充分发挥多核 CPU 的逻辑处理能力和多 GPU 的计算处理能力。

在 CUDA 编程模型中，作为主机的 CPU 与作为设备的 GPU 分工合作，协同工作。CPU 负责执行一些不适合进行并行计算的部分，比如复杂逻辑事务处理和管理等；GPU 负责执行需要大规模并行计算的密集型计算部分。主机端的系统内存作为 CPU 处理器的存储器地址空间，设备端的显卡显存作为 GPU 协处理器的存储器地址空间，它们相互独立。CUDA 和 C 语言程序以同样的方式对存储空间进行操作。显存作为 CUDA 的存储空间，对它进行操作是通过调用存储器管理函数来实现的，这些存储器管理函数包含在 CUDA 的驱动 API 函数库中。显存操作和内存操作一样，同样包括开辟、释放和初始化存储空间等一般操作，但显存操作还包含了进行 CPU-GPU 之间的数据传输等特殊操作。

GPU 主要完成程序中的并行计算部分。Kernel（内核函数）是在 GPU 上运行的 CUDA 并行计算函数。一个完整的 CUDA 程序并不仅仅只包含一个 Kernel 函数，Kernel 函数只是其中的一部分，是整个 CUDA 程序其中的一个模块，一个能够并行执行的模块。如图 3-8 所示，在主机端运行的串行处理代码模块和在设备端运行的一系列并行 Kernel 函数模块共同组成了一个完整的 CUDA 程序。CUDA 程序会按照程序执行流程中的顺序来依次顺序执行这些模块。

从图 3-8 中可以看出，一个 Kernel 并行函数模块包括两层并行，底层是线程块 block 中的线程 thread 之间的并行，上层是线程网格 Grid 中的线程块 block 之间的并行。CUDA 编程模型一个最重要的创新就是引入了上述的包含两个层次并行的程序模型。

新一代的 GPU 通用开发环境如 CUDA、OpenCL 和 DirectX 11 的 Computing Shader 都采用了这种编程模型，由 CPU 主机端（Host）和 GPU 设备端（Device）共同组成了一个计算系统，并将两个层次并行的程序模型引入了 GPU 设备端。

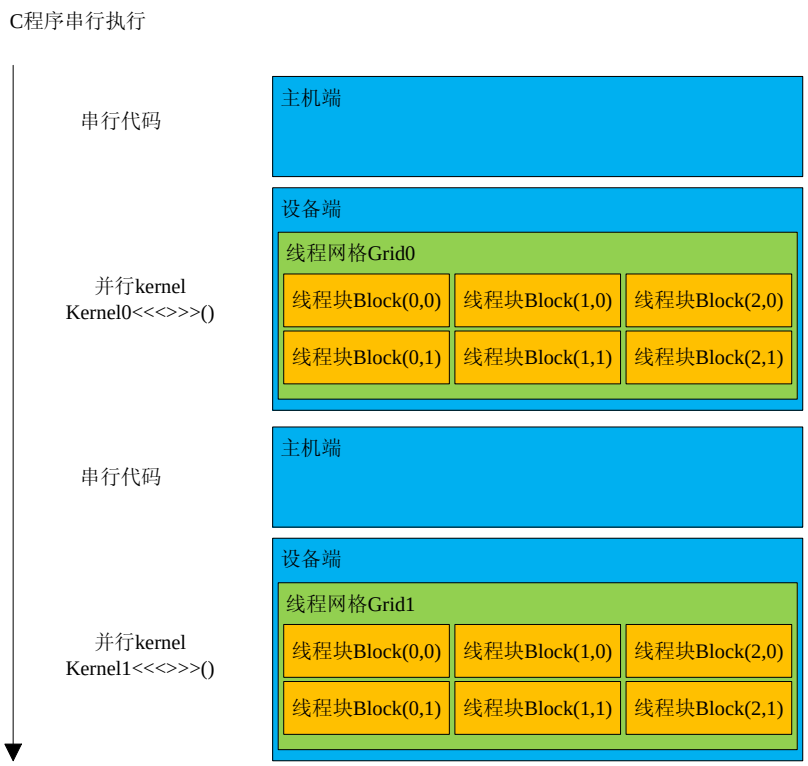

图 3-8 CUDA 编程模型

3.3.2 线程结构

CUDA 中的 Kernel 并行函数模块由大量线程同时执行，这些线程在逻辑上被组织成线程块（block）和线程网格（Grid），由若干线程（thread）组成一个线程块（block），若干线程块（block）组成一个线程网格（Grid）。实际上，线程块（block）是 Kernel 并行函数模块的基本执行单元。定义线程网格（Grid）的概念只是为了在逻辑上便于表述多个能够并行执行的线程块（block）的集合。各个线程块（block）之间相互无法通信，地位相等，并行执行，且没有固定的执行顺序。

目前新的支持 DirectX 11、采用 Fermi 架构的 GPU 使用了 MIMD（多指令多数据）架构，一个 Kernel 并行函数模块中可以支持同时存在多个不同的线程网格（Grid）。Kernel 内核中的线程结构关系如图 3-9 所示。

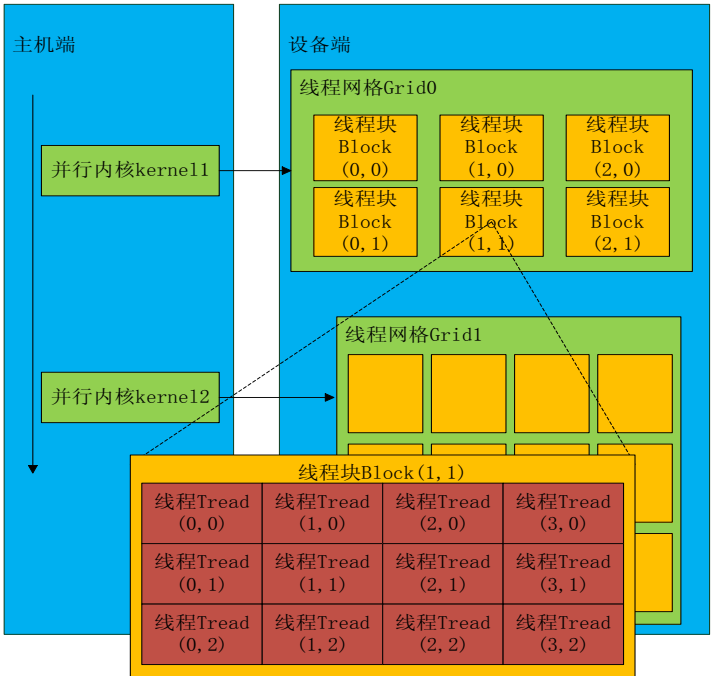

图 3-9 kernel 内核中的线程结构关系

首先介绍两个概念：线程块（block）和线程束（warp）。

线程块：CUDA 中的 Kernel 函数实质上的执行单位是线程块（block）。由于数据需要在线程块内共享，所以同一个线程块中的所有线程必须在同一个硬件执行模块 SM 单元中执行，而 SM 单元中的每一个计算核心（CUDA Core）负责执行线程块中的一个线程。一个 SM 单元中可以同时有多个线程块在等待执行，这些线程块的上下文也存在于同一个 SM 单元中，但一个线程块只能在一个 SM 单元中执行。当一个线程块执行高延迟操作时，比如同步或者访问显存等，SM 单元就可以执行另一个在等待队列中的线程块，充分发挥 GPU 的运算能力，最大限度地使用 GPU 中的运算资源。

线程束：在实际中，线程块会被划分为更小的线程组织单元来执行，这就是线程束（warp）。GPU 硬件的计算能力决定了线程束的大小。在 Fermi 架构的 GPU 中，以 32 个线程 ID（thread ID）号连续的线程为一个线程束。例如，每个线程块中线程 ID 为 0~31 的线程为一个线程束，32~63 的线程为另一个线程束。实际上，应用程序在 GPU 硬件上实际执行时，最小的执行单位是线程束。线程束并不存在于抽象的 CUDA 编程模型中，它只是一个被硬件的实际计算能力所决定的概念，但其拥有巨大的影响力。

Fermi 架构的每一个 SM 单元中都包含有 2 个指令发送单元，使得两个线程

束 warp 可以相互独立的在不同的计算核心上并发运行。Fermi 架构中的每一个 SM 单元包含 32 个 CUDA 计算核心 CUDA core，这些计算核心按照 16 个一组被分为两组。两个线程束 warp 的一条指令可以分别同时在这两组计算核心上并发执行，或者同时在 16 个存储/载入单元上执行存/取数据操作，这就是 Fermi 架构的双重线程束 Dual warp 调度机制。而且执行线程束 warp 调度任务的调度器对指令流并不进行附属检查。正是由于 Fermi 架构的这种线程束的双重调度机制，如图 3-10 所示，使得硬件的计算能力可以发挥到极致。

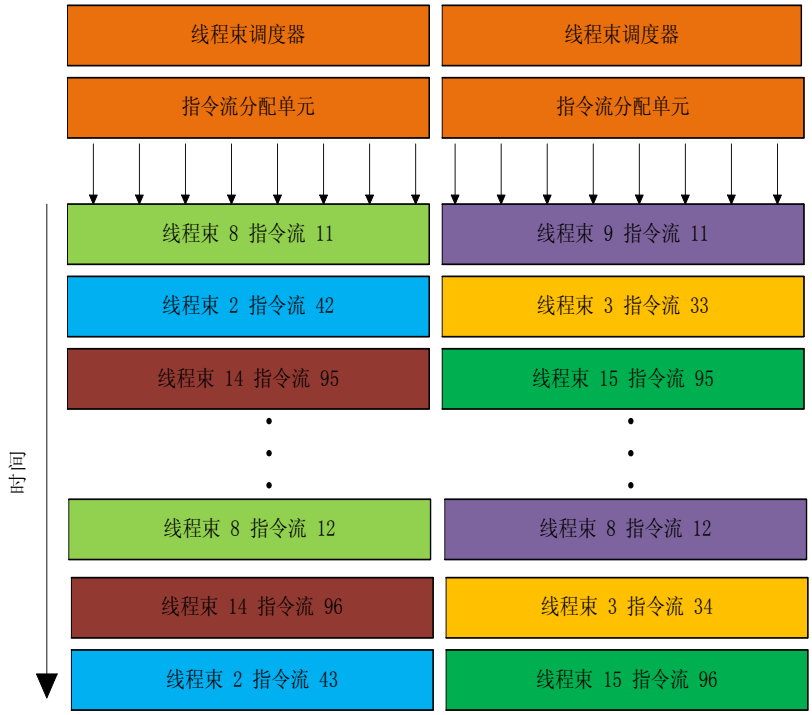

图 3-10 Dual warp scheduling（双重线程束调度）

在 Fermi 架构中，有很多的指令可以使用双重线程束调度（Dual warp）机制进行并发执行，例如整数运算指令，浮点数运算指令，整数浮点数的混合运算指令，单、双精度混合运算指令、数据存取操作指令等都可以并发执行。

为了更好的隐藏延迟，Nvidia 公司和 AMD 公司采用了不同的 GPU 发展策略。Nvidia 公司的策略是不断改进 GPU 架构，增加 GPU 运算核心的周边资源；AMD 公司的策略是不断扩大运算核心即流处理器的规模。事实上，Nvidia 公司的 GT200 架构已经可以控制 SIMT 活用跳转来在实现线程在不同的 SM 单元之间进行跳跃。命令单元为 multi-thread 模式，能够执行 Out-of-Order 指令，而当处理 warp 命令流时则是 In-Order，而根据 Nvidia 公司的架构设计师 John Nickolls 的介绍，GT200 架构实际 warp 中的线程也能够支持 Out-of-Order。

Fermi架构的每个SM单元都有两个独立分配单元和两个线程束warp调度器，完全独立于 SM 单元内的其它部分。两个独立分配单元在一个时钟周期里均可以选择发送 half-warp，而且这些 half-warp 可以来自不同的 warp。独立分配单元和运算核心之间有一个完整的 Crossbar 交叉开关。每个独立分配单元都可以在一定的限制条件下向 SM 单元内的任何运算核心分配线程。

Fermi 架构的双层分布式调度机制是其另一个重要特性。在芯片层面由全局分布式线程调度引擎向每个 SM 单元分发线程块，在 SM 单元层面由线程束分布式调度引擎以线程束为单位在硬件上并发执行。

Fermi 架构在 SM 单元层面实现了双指令的同时调度执行，如图 3-11 所示，意味着在单个线程内实现了多任务。每个 SM 单元内实现的单线程多任务综合起来就形成了显示芯片核心的多线程多任务，这实际上就是 Fermi 架构硬件的并行原理。Fermi 架构的这种设计已经同 CPU 非常类似，而且可以预见的是随着未来 GPU 架构的不断发展，GPU 会和 CPU 越来越相似。

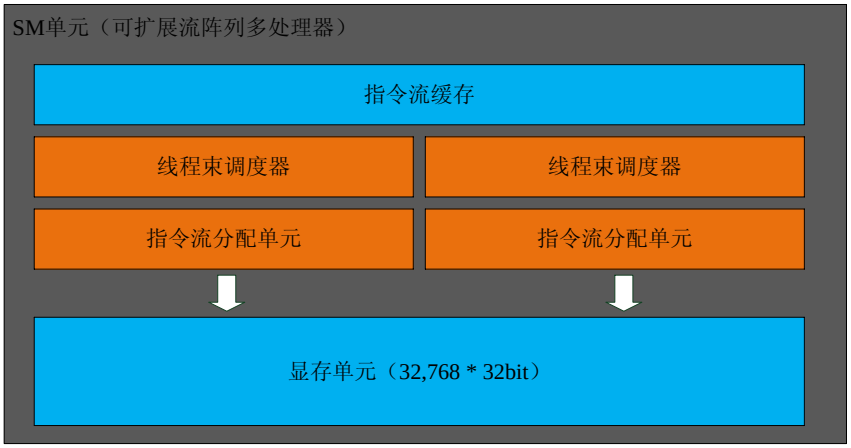

图 3-11 Fermi 架构中 SM 单元内的双指令调度

G80 架构中的第一代线程调度引擎，实时实现了对上万个线程的调度管理。Fermi 架构不仅仅只是对原有的线程调度机制进行了改进，提高了线程调度效率，还增加了线程的上下文交换机制和 kernel 核函数的并发执行机制，进一步增强了调度线程块的能力。总的说来，Fermi 架构的线程调度能力比 G80 架构提高了 10 倍。图 3-12 为并行执行内核和串行执行内核的比较。

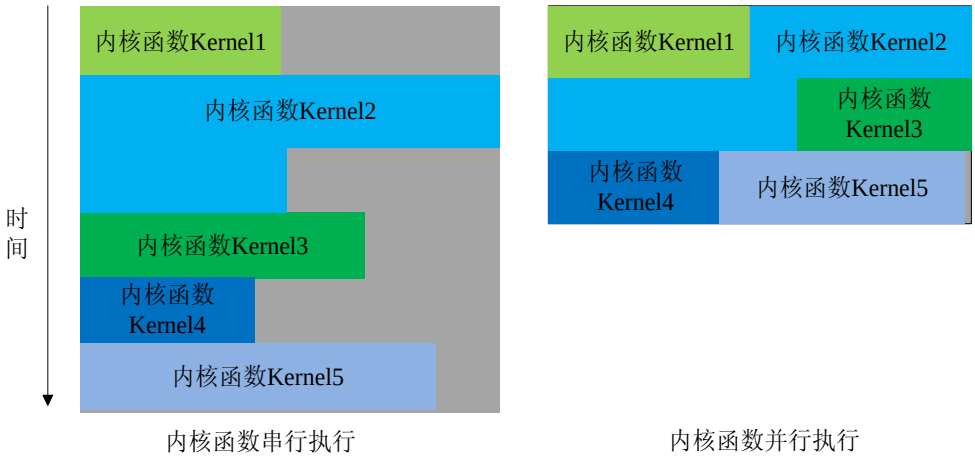

图 3-12 并行执行内核与串行执行内核的比较

Fermi 架构中多个 kernel 内核函数能够同时并行执行，即同一个应用程序中多个不同的内核函数能够在 GPU 上同时运行。内核函数的并发执行机制可以充分发挥 GPU 的运算能力，让应用程序执行更多的内核函数。Fermi 架构中同一个 GPU 上能够同时运行多个拥有相同的线程上下文的内核函数，使得 GPU 的硬件资源能够得到更加有效地利用。而快捷的上下文切换机制使得拥有不同的线程上下文的内核函数也能够在 GPU 上更快速的运行。

3.3.3 CUDA 软件体系

如图 3-13 所示， CUDA 高层数学库、CUDA runtime API、CUDA 驱动 API 共同组成了 CUDA 的三层软件体系。CUDA C 语言是 CUDA 的核心，它包含 C 语言的扩展集和一个 CUDA 运行时库（Runtime API），使用 C 语言扩展集和 CUDA 运行时库中函数编写的应用程序源文件必须通过 nvcc 编译器进行编译。

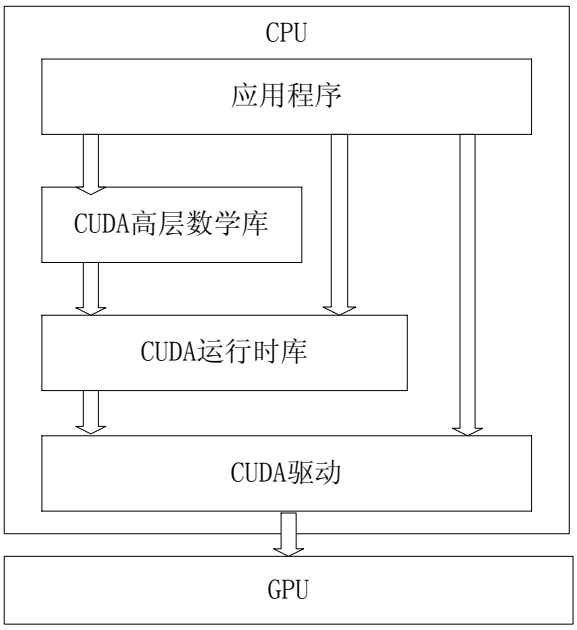

图 3-13 CUDA 体系架构

CUDA C 语言通过 nvcc 编译器编译得到的只是设备端代码，而要执行内核函数的启动，进行 GPU 硬件资源的管理，就只能通过调用 CUDA 运行时函数库或者 CUDA 驱动函数库中的函数来实现。在一个程序中只能选择使用两个函数库中的一个，而不能将两个函数库中的函数进行混合使用。

CUDA C 代码根据配置通过 nvcc 编译器编译，可以生成三种不同类型的输出：标准 C、CUDA 二进制代码和 PTX 代码。nvcc 是一种驱动式的编译器，在命令行模式中，通过使用不同的选项，可以启动不同的编译工具在不同的编译阶段完成编译工作。

nvcc 编译器工作的一般流程是：首先将源程序代码进行分割，把运行在主机端的代码和运行在设备端的代码分别做成两个代码集合，然后再使用不同的编译工具对这两个源程序代码集合分别进行编译。编译后两个集合的输出文件格式也不相同，设备端代码的输出为 CUDA 二进制代码或者 PTX 代码，主机端代码的输出则为标准 C。也可以在编译的最后阶段，由其他编译器将主机端代码直接生成.obj 或者.o 文件。图 3-14 为 nvcc 的编译流程。

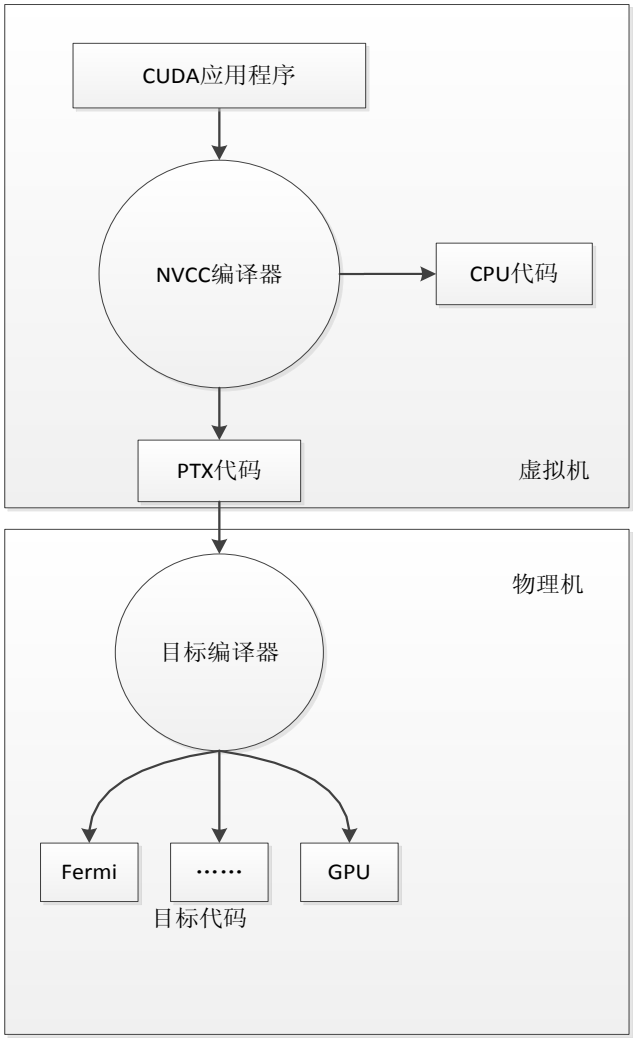

图 3-14 nvcc 编译流程

3.3.4 CUDA 存储器模型

CUDA 中规定了存储器模型，共分为 register 存储寄存器、shared memory 共享存储器、local memory 局部存储器、texture memory 纹理存储器、constant memory 常数存储器以及 global memory 全局存储器等 6 种存储器。如图 3-15 所示，线程块中的线程在运行时将需要调用不同存储器中的数据，而且每种存储器设备的访问权限均有所区别。

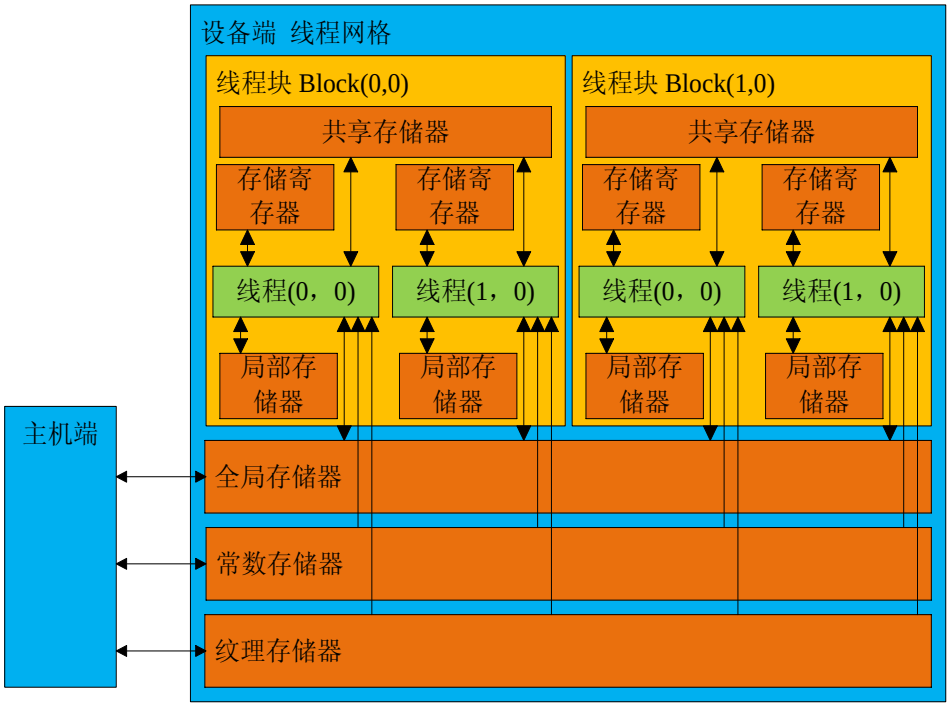

图 3-15 存储器层次结构

存储寄存器和局部存储器对每个线程来说是私有的；共享存储器由一个线程块内的全部线程共同使用；全局存储器由一个线程网格内的全部线程块共同使用，即被网格内全部线程共享；常数存储器和纹理存储器则由全部线程网格共同使用，是真正意义上的被全部线程共享。global memory 全局存储器、constant memory 常数存储器和 texture memory 纹理存储器中的数据在整个程序运行周期中都将被保持，程序中的所有 kernel 内核函数都可以调用。

表 3-1 给出了 6 种存储器的位置、缓存情况、访问权限及生存域。

表 3-1 各种存储器比较

存储器	位置	拥有缓存	访问权限	变量生存周期
存储寄存器	GPU 片内	N/A	设备端可读/写	与线程相同
局部存储器	板载显存	无	设备端可读/写	与线程相同
共享存储器	GPU 片内	N/A	设备端可读/写	与线程块相同
全局存储器	板载显存	无	设备端可读/写，主机端可读/写	可在程序中保持
常数存储器	板载显存	有	设备端可读，主机端可读/写	可在程序中保持
纹理存储器	板载显存	有	设备端可读，主机端可读/写	可在程序中保持

register 存储寄存器是 GPU 片内的高速缓存器，指令访问存储寄存器的延迟极低。register file 存储寄存器文件是存储寄存器的基本单元，每个 register file 大

小为 32bit。Fermi 架构的 GPU 中每个 SM 单元拥有 16384 个 register file，即共有 524288bit，表面看来似乎存储寄存器占用的空间巨大，但是实际上由于 GPU 中要运行上万甚至更多的线程，而把这些看起来数量巨大的 register file 平均分配给全部并发执行的线程，那么分给每个线程的就只有很少量的 register file。故而，编程时只能分配少量的私有变量给每个线程。

local memory 局部存储器也是每个线程的私有存储器。如果程序消耗完 register 存储寄存器，数据就将被存储在 local memory 局部存储器中。如果每个线程使用了过多的 register 存储寄存器，比如大型数组或结构体的定义，local memory 局部存储器中可能就会分配线程的私有数据。一个线程的输入和中间变量将被保存在 register 存储寄存器或者 local memory 局部存储器中。local memory 局部存储器是位于显存上的，因此访问 local memory 局部存储器会有较高的时间延迟。

shared memory 共享存储器是由一个线程块内的全部线程共同使用，是一种在 SM 单元内的高速的可读写存储器。访问 shared memory 共享存储器和访问 register 存储寄存器的时间延迟几乎一样，是访问延迟最小的实现线程间通信的方法。每个 SM 单元中的共享存储器被组织为 16 个 bank，总大小为 16KB。

global memory 全局存储器也是一种可读写存储器，它被安排在 GPU 硬件的显存部分，显存上主要都是 global memory 全局存储器。CPU、GPU 都可以访问 global memory 全局存储器并进行读写操作。程序中的任意线程都能任意地读写 global memory 全局存储器中的所有位置。由于 global memory 全局存储器的特点是访问带宽较高，但同时会消耗较多的访问时间，即访问延迟也较高。必须采取合并访问的方式，有效避免分区访问冲突，才能合理高效地利用全局存储器访问带宽。

constant memory 常数存储器是一种只读的存储器，虽然也在显存上，但是拥有用于访问加速的缓存。常数存储器 constant memory 只有 64KB，主要用于存储 CUDA 程序中需要被频繁访问的只读参数。当常数存储器 constant memory 中的同一数据被来自同一 half-warp 的线程访问时，如果发生缓存命中，那么只需要一个时钟周期就可以获得数据。每个 SM 单元中拥有的常数存储器 constant memory 缓存的大小为 8KB。由于常数存储器 constant memory 是只读的，因此不存在缓存一致性问题。

texture memory 纹理存储器也是位于显存，拥有缓存加速的只读存储器。数据以三维以下数组的形式存储在纹理存储器 texture memory 中，访问通过缓存加速。在通用计算中，纹理存储器 texture memory 非常适合实现对表的查找，也可

以加速对大量数据的非对齐或者随机访问。

3.4 本章小结

本章首先对 CUDA 原理进行了简介，其次详细介绍了 Fermi 架构的 GPU，以及基于 Fermi 架构的 CUDA 编程模型。通过对 GPU 架构和 CUDA 模型的深入研究分析，充分论证了通过 CUDA 实现大规模并行计算的可行性。

第四章　利用 MPI 和 CUDA 开发多粒度 MD5 散列值并行破解程序

4.1 安全散列算法介绍

安全散列算法是广泛应用于现实生活中的一类加密算法。在信息安全领域中，作为一类重要的加密算法，它可以快速有效、安全可靠地实现数字签名和身份认证，同时也是众多信息安全协议中的一个重要组成模块。由于安全散列算法有着极其广泛的应用，其本身的特点又非常鲜明，因而有很多的名字，其含义也各个不同；如压缩函数、数据认证码、消息摘要、数字指纹、数据完整性检测、密码校验和消息认证码、篡改检测码等。安全散列算法按照加密时是否有密钥参与控制分为两大类：一类有密钥参与控制，称之为密码安全散列函数。有密钥参与控制的安全散列函数要满足各种安全性要求，如能防止伪造，抗击诸如生日攻击，中间相遇攻击等各种攻击。与此类安全散列值相关的元素有两类，一类是输入的字符串，另一类是参与算法控制的密钥。因此，只有密钥的拥有者才能根据算法计算出相应的安全散列值，故而可以进行有身份认证，如用作消息认证码（MAC）。另一类没有密钥参与控制的安全散列函数，又称为一般安全散列函数，其安全散列值只与输入的字符串相关，任何人都可以用输入的字符串通过算法运算得到，因而不能进行身份认证，而只能用于数据完整性检测，如做篡改检测码（MDC）。安全散列算法构造方法与要求与分组密码相类似，但又有很多差异。

安全散列算法是一种单向函数，算法实现的基本原理是：将一个任意长度的字符串通过一系列函数运算映射成一个定长的、较短的输出字符串。设计一个完好的安全散列算法很难，其输出的安全散列码必须是与输入字符串的所有比特都相关的函数值；输入字符串的任意一个比特的改变都将导致输出的安全散列码发生变化；算法要具备差错检测能力；同时算法要求运算速度快、效率高，能在软硬件上快速实现。

常用的安全散列算法有 MD2、MD4、MD5、SHA、RIPEMD-160、GOST 等，下面介绍本文研究的一种重要的安全散列算法：MD5 算法。

4.2 MD5 算法简介及破解原理

4.2.1 MD5 算法简介

MD5 即 Message-Digest Algorithm 5（信息-摘要算法 5），是在信息安全领域中使用非常广泛的一种安全散列算法。

美国麻省理工学院 Ronald L. Rivest 于 1990 年设计了一个称为 MD4 的安全散列算法，为了弥补 MD4 算法的缺陷，增强安全性，Rivest 于 1991 年将 MD4 算法改进为 MD5 算法。MD5 算法比 MD4 算法更加复杂，但它们拥有类似的设计思想。1992 年 8 月 Rivest 向 IETF 提交了一份描述 MD5 算法原理的重要文件，后公布于 RFC1321。由于 MD5 算法的公开性和安全性，90 年代在各种程序语言中被广泛应用，都有相应的应用实例，主要作用是用来确保信息传输的完整性和一致性。

该算法对任意长度的报文按 512 比特进行分组，对每个分组利用 4 个原始逻辑函数进行每个 16 次的迭代循环运算，最终得到一个长度为 128 比特的报文信息摘要散列值。

具体算法如下：

（1）数据预处理

给定一个字符串 x，首先生成一个如同以下这种形式的消息字符串：

$m = m_0, m_1, \cdots, m_{n-1}$

这里每个 $m_i(0 \le i \le n-1)$ 是长为 4 字节的串，$n \equiv 0(\mathrm{mod}16)$. 称每个 $m_i(0 \le i \le n-1)$ 为一个字。这样，m 的长度始终为 512 的倍数。由 x 产生 m 的算法如下：

1）$d = 447 - (|x| \mathrm{mod} 512)$。　(当 $d < 0$ 时，按模 512 处理)

2）设 l 表示 $|x| (\mathrm{mod} 2^{64})$ 的二进制表示，则 $|l| = 64bit$。

3）$m = x \| 1 \| 0^d \| l$　$\|$ 表示级联。

在 m 的构造中，首先将 x 的右边填上一个 1，然后级联足够多（d 个）的 0，使得整个消息的长度模 512 等于 448。然后再级联 $|x| (\mathrm{mod} 2^{64})$ 的二进制表示长度为 $64bit$。产生的 m 的长度恰为 512 的倍数，所以可将 m 分成 $32bit$ 字的个数是 16 的倍数。

（2）构造消息摘要

从 m 开始可构造一个 $128bit$ 的消息摘要，其构造过程如下：

给出四个 $32bit$ 初始值（用 16 进制表示）为：

$A=0x67452301$

$B=0xefcdab89$

$C=0x98badcfe$

$D=0x10325476$

执行算法的主循环（见图 4-1）

先将四个初始值置入四个寄存器：$A\to a$，$B\to b$，$C\to c$，$D\to d$。

$for\ i=0\ to\ (n/16)-1\ do$　　（即作 $n/16$ 次主循环）

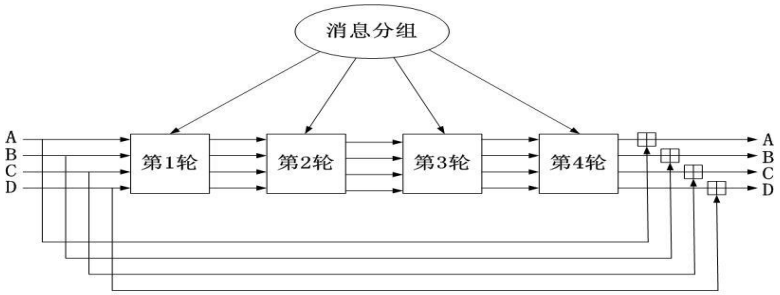

图 4-1 MD5 的主循环

每次主循环的输入时消息字符串的一个 $512bit$ 的分组（这样的分组共有 $n/16$ 个，故要执行 $n/16$ 次主循环），它被分成 16 个 $32bit$ 的字，记其中一个分组的 16 个字为 $M_0,M_1\cdots,M_{15}$。主循环有四轮，每轮都很相似，但并不相同。每轮 16 次操作，这 16 次操作的每一次使用一个字 M_j 作输入，那次操作用哪个字，后面将给出。每一次操作的过程如图 4-2 所示。

$for\ \ j=0\ to\ 15\ do$

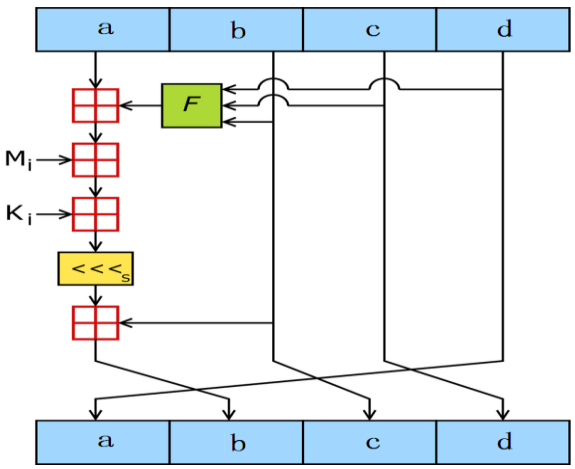

图 4-2 MD5 的一轮一次操作过程

图中的非线性函数是以下图 4-3 中四个函数之一：

```
F(X,Y,Z)=(X&Y)|((~X)&Z)
G(X,Y,Z)=(X&Z)|(Y&(~Z))
H(X,Y,Z)=X^Y^Z
I(X,Y,Z)=Y^(X|(~Z))
(^是异或，&是与，|是或，~是反)
```

图 4-3 MD5 算法中的非线性函数

这四个函数分别给第一至第四轮运算使用，图中四个 32 位寄存器中的 b,c,d 寄存器的 32 比特输出作为非线性函数的输入，其输出的 $32bit$ 与 a 寄存器 $32bit \bmod 2^{32}$ 相加后再与 $M_j(32bit) \bmod 2^{32}$ 相加，然后再和常量 $K_i(32bit) \bmod 2^{32}$ 相加，其结果循环左移 s_i 位再与 b 寄存器的 $32bit \bmod 2^{32}$ 相加后送入 a 寄存器。这一过程可记为如图 4-4 所示：

```
FF(a,b,c,d,Mj,si,Ki)  表示 a=b+((a+F(b,c,d)+Mj+Ki) <<<si
GG(a,b,c,d,Mj,si,Ki) 表示 a=b+((a+G(b,c,d)+Mj+Ki) <<<si
HH(a,b,c,d,Mj,si,Ki) 表示 a=b+((a+H(b,c,d)+Mj+Ki) <<<si
II(a,b,c,d,Mj,si,Ki)  表示 a=b+((a+I(b,c,d)+Mj+Ki) <<<si
```

图 4-4 函数形式表示

各轮的 16 次操作分别是：

第一轮如图 4-5 所示：

FF(a,b,c,d,M_0, 7,0xd76aa478)
FF(d,a,b,c,M_1,12,0xe8c7b756)
FF(c,d,a,b,M_2,17,0x242070db)
FF(b,c,d,a,M_3,22,0xc1bdceee)
FF(a,b,c,d,M_4, 7,0xf57c0faf)
FF(d,a,b,c,M_5,12,0x4787c62a)
FF(c,d,a,b,M_6,17,0xa8304613)
FF(b,c,d,a,M_7,22,0xfd469501)
FF(a,b,c,d,M_8, 7,0x698098d8)
FF(d,a,b,c,M_9,12,0x8b44f7af)
FF(c,d,a,b,M_{10},17,0xffff5bb1)
FF(b,c,d,a,M_{11},22,0x895cd7be)
FF(a,b,c,d,M_{12}, 7,0x6b901122)
FF(d,a,b,c,M_{13},12,0xfd987193)
FF(c,d,a,b,M_{14},17,0xa679438e)
FF(b,c,d,a,M_{15},22,0x49b40821)

图 4-5 第一轮操作

第二轮如图 4-6 所示：

GG(a,b,c,d,M_1, 5,0xf61e2562)
GG(d,a,b,c,M_6, 9,0xc040b340)
GG(c,d,a,b,M_{11},14,0x265e5a51)
GG(b,c,d,a,M_0,20,0xe9b6c7aa)
GG(a,b,c,d,M_5, 5,0xd62f105d)
GG(d,a,b,c,M_{10}, 9,0x02441453)
GG(c,d,a,b,M_{15},14,0xd8a1e681)
GG(b,c,d,a,M_4,20,0xe7d3fbc8)
GG(a,b,c,d,M_9, 5,0x21e1cde6)
GG(d,a,b,c,M_{14}, 9,0xc33707d6)
GG(c,d,a,b,M_3,14,0xf4d50d87)
GG(b,c,d,a,M_8,20,0x455a14ed)
GG(a,b,c,d,M_{13}, 5,0xa9e3e905)
GG(d,a,b,c,M_2, 9,0xfcefa3f8)
GG(c,d,a,b,M_7,14,0x676f02d9)
GG(b,c,d,a,M_{12},20,0x8d2a4c8a)

图 4-6 第二轮操作

第三轮如图 4-7 所示：

HH(a,b,c,d,M_5 , 4,0xfffa3942)
HH(d,a,b,c,M_8 ,11,0x8771f681)
HH(c,d,a,b,M_{11},16,0x6d9d6122)
HH(b,c,d,a,M_{14},23,0xfde5380c)
HH(a,b,c,d,M_1 , 4,0xa4beea44)
HH(d,a,b,c,M_4 ,11,0x4bdecfa9)
HH(c,d,a,b,M_7 ,16,0xf6bb4b60)
HH(b,c,d,a,M_{10},23,0xbebfbc70)
HH(a,b,c,d,M_{13}, 4,0x289b7ec6)
HH(d,a,b,c,M_0 ,11,0xeaa127fa)
HH(c,d,a,b,M_3 ,16,0xd4ef3085)
HH(b,c,d,a,M_6 ,23,0x04881d05)
HH(a,b,c,d,M_9 , 4,0xd9d4d039)
HH(d,a,b,c,M_{12},11,0xe6db99e5)
HH(c,d,a,b,M_{15},16,0x1fa27cf8)
HH(b,c,d,a,M_2 ,23,0xc4ac5665)

图 4-7 第三轮操作

第四轮如图 4-8 所示：

II(a,b,c,d,M_0 , 6,0xf4292244)
II(d,a,b,c,M_7 ,10,0x432aff97)
II(c,d,a,b,M_{14},15,0xab9423a7)
II(b,c,d,a,M_5 ,21,0xfc93a039)
II(a,b,c,d,M_{12}, 6,0x655b59c3)
II(d,a,b,c,M_3 ,10,0x8f0ccc92)
II(c,d,a,b,M_{10},15,0xffeff47d)
II(b,c,d,a,M_1 ,21,0x85845dd1)
II(a,b,c,d,M_8 , 6,0x6fa87e4f)
II(d,a,b,c,M_{15},10,0xfe2ce6e0)
II(c,d,a,b,M_6 ,15,0xa3014314)
II(b,c,d,a,M_{13},21,0x4e0811a1)
II(a,b,c,d,M_4 , 6,0xf7537e82)
II(d,a,b,c,M_{11},10,0xbd3af235)
II(c,d,a,b,M_2 ,15,0x2ad7d2bb)
II(b,c,d,a,M_9 ,21,0xeb86d391)

图 4-8 第四轮操作

常数 K_i 可以如下选择：

K_i 是 $2^{32} \times \sin(i)$ 的整数部分，其中 i 的单位是弧度。

在完成了上述全部操作后，将四个初始值 A,B,C,D 与寄存器 a,b,c,d 中的数值分别相加，然后继续运行主循环输入数据为下一个 $512bit$ 分组的数据。结果最后输出的是:

$A \| B \| C \| D$ ‖表示级联

信息安全领域广泛应用了 MD5 安全散列算法。但是由于计算机运算能力的不断提高和对 MD5 算法的不断深入分析，MD5 算法的弱点被不断发现，现在已经可以构造出两个信息使它们具有相同 MD5 散列值，这使得 MD5 算法不再适合在当前安全需求越来越高的环境中使用。

MD5 安全散列算法广泛应用于信息安全领域，例如计算机操作系统用户的认证登陆、信息完整性检测、数字签名、数字水印等诸多方面。如在 Linux 操作系统中就是以 MD5 散列值的形式将用户的登陆密码存储在文件系统中。当用户登录系统的时候，系统首先调用 MD5 算法，以用户输入的明文登陆密码作为输入，计算 MD5 散列值，然后从文件系统中提取出正确登录密码的 MD5 散列值，将两个散列值进行比较，从而判断用户是否输入的是正确的登陆密码。通过这样的校验方式，用户登录操作系统的合法性得到了验证的同时也保护了用户登录密码的私密性。有效避免了具有系统管理员权限的用户知道普通用户的登陆密码。在一些下载软件中，也使用通过计算 MD5 散列值的方式检测下载到的碎片的完整性。

4.2.2 MD5 破解原理

满足以下 3 个条件之一，MD5 安全算法即被破解：

⑴对于任意给定的消息 m，找一个 m'，$m' \neq m$，使得 $MD5(m') = MD5(m)$

⑵找到任意一对消息 m，m'；$m' \neq m$ 使得 $MD5(m') = MD5(m)$

⑶对于给定的散列值 x，找一个 m，使得 $MD5(m) = x$

MD5 安全散列算法将任意长度的字符串映射成一个固定长度为 128bit 的字符串，显然这是一个多对一的映射，故而这个映射也显然是单向的。即对于 MD5 算法来说已知一个任意长度的字符串 m，由 m 计算 $MD5(m)$ 是容易的，但要找一个 m'，使得 $MD5(m') = MD5(m)$ 是很困难的，理论上所需的操作量级是 2^{128}。正是因为这个原因，现在使用最多的攻击 MD5 的方法主要有：穷举攻击法、彩虹链表攻击法。

穷举攻击法是通过指定字符集和口令长度，穷举指定长度内所有指定字符的

组合作为 m'，判断 $MD5(m')$ 是否等于需要破解的 MD5 散列值，直至二者相等，即得到 MD5 散列值的明文口令 m，MD5 散列值破解成功。彩虹链表攻击法是预先将通过穷举攻击法得到的 MD5 散列值明密对构造成一张有序的一一映射的表，并使用压缩、排序等方法减少表的存储空间，使用特定的搜索、匹配算法减少搜索时间，最终在已知明密对应表中搜索到需要破解的 MD5 散列值对应的原始明文的方法。由于彩虹链表是存储穷举得到所有明文、密文对，故而彩虹表是非常庞大的，通常需要 TB 级的磁盘阵列。

4.3 基于 GPU 的 SMP 集群 MD5 散列值穷举破解程序的设计与实现

4.3.1 MPI-CUDA 混合并行编程模型

在本项目实际实现中，我们根据具体的基于 GPU 的 SMP 集群，运用 MPI-CUDA 混合并行编程模型，设计 MD5 散列值并行破解算法。MPI-CUDA 的混合并行编程模型，由消息传递模型（MPI）和线程模型（CUDA）组合而成。

MPI-CUDA 的混合并行编程模型，很好地映射了基于 GPU 的 SMP 集群的体系结构，在两级存储结构的基础上，提供了两级并行，进程级的粗粒度并行和进程内线程级的细粒度并行，能充分利用 MPI 消息传递编程模型和 CUDA 线程编程模型的优点。

混合并行编程模型 MPI-CUDA 具有如下的层次结构：MPI 位于上层，用来表示节点间和节点内 CPU 内核间的进程级并行；CUDA 位于下层，用来表示进程内运行于 GPU 上的线程级并行。

GPU 采用的是轻量级的线程，这些线程由硬件进行管理，线程之间进行切换几乎是零开销的。线程切换在这里成为一件好事：当一个线程由于进行片外存储器访问开始等待以后，可以立即进行线程切换，另一个处于就绪状态的线程调入硬件开始执行，通过计算来隐藏访问延迟。当线程中执行计算指令需要的时间较多，而执行存储器访问指令的时间相对较少，即计算密集度比较高时，访问延迟就可以被计算隐藏，而且线程越多，访问延迟隐藏得越好。

MPI-CUDA 混合并行编程模型中，每个 CPU 核只能绑定一个 GPU，CPU 核与核之间的通信通过消息传递（MPI）完成，GPU 与 CPU 之间不能直接通信。MPI 实现进程间通信，CUDA 实现 GPU 大规模数据并行任务，MPI-CUDA 混合

并行编程模型，提供成本更低、体积和功耗更小、性能更强的高性能计算解决方案，实现基于 GPU 的 SMP 集群中的多 GPU 的并行计算。

实现的基本原理是：按照通信的密集程度，首先将任务分解为几个通信不密集的部分，在每个 SMP 节点上可以运行等于 CPU 内核数量的若干部分，每个 CPU 内核创建一个进程，节点内 CPU 内核间和节点间的进程间通过 MPI 消息传递来通信。其次在每个 CPU 内核进程内，根据每个 CPU 内核绑定的 GPU 协处理器的运算能力，使用 CUDA 指令创建若干并行执行的线程。

为了充分利用 CPU 和 GPU 的运算能力，我们设计了一种 MPI-CUDA 的异步模型，使得 CPU 在 GPU 进行计算或数据传输时不必等待 GPU 运行完毕就可以继续执行相关计算程序，从而 CPU 和 GPU 可以并行工作，节省了 CPU 的等待时间，有效提高了破解程序的并行执行效率，更加有效地利用了系统中的计算资源。见图 4-9。

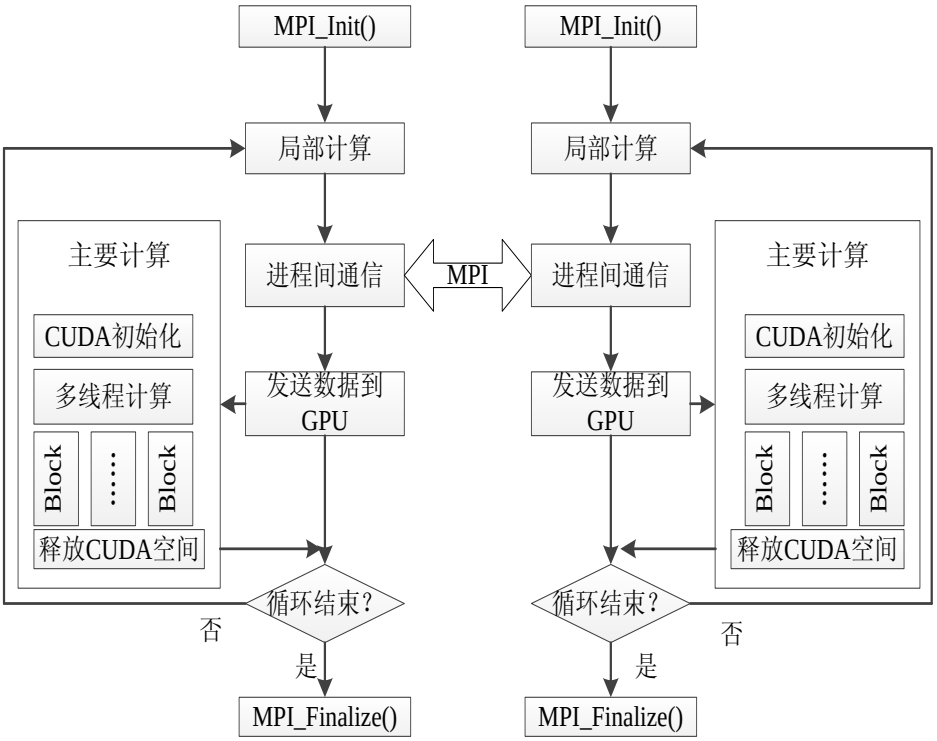

图 4-9 MPI-CUDA 异步模型

在 MPI-CUDA 异步模型中，CPU 则利用 MPI 消息传递函数与其他进程进行数据交换的同时，CUDA 则利用 GPU 进行多线程计算并验证已知口令 MD5 散列值，在通信开销中有效隐藏了计算开销，显著地提高了并行计算的执行效率。使用异步模型，程序的执行时间将只由初始化数据时间和计算时间相加得到。由于有效隐藏了主机和设备之间的通信和数据传输时间，因此程序的性能得到了显著

改善。见图 4-10。

图 4-10 MPI-CUDA 异步执行方式

4.3.2 串行 MD5 破解算法及基于 CPU 并行的 MD5 破解算法

传统的串行 MD5 破解算法，是基于 CPU 的密码口令遍历，其一般解密算法如下：

串行解密算法：

Step1：读取 MD5 散列值存储文件，获取 MD5 散列值 m；

Step2：生成待验证口令，针对每一个口令，执行；

Step2.1：利用核心加密算法 MD5 对口令进行加密，得到密文 hash；

Step2.2：判断 m==hash，相同 return，否则转到 Step2 继续；

Step3: 口令验证完毕，结束；

从上述解密算法可以看出，数据处理流程相同，口令产生、口令验证等操作相同；从处理流程可以看出，各个待验证口令之间没有相关性，非常适合并行。运算量巨大；当字符集和穷举长度增加时，运算量成指数级增长。口令字具有人为性，规律性，可设置性，因此好的口令库和口令规则可提高口令破译的攻击效率，可以基于口令设置规律进行研究，从而实现口令的特征控制。

随着 CPU 运算能力的提高，设计的传统的基于 CPU 的并行程序，执行流程如图 4-11 所示：

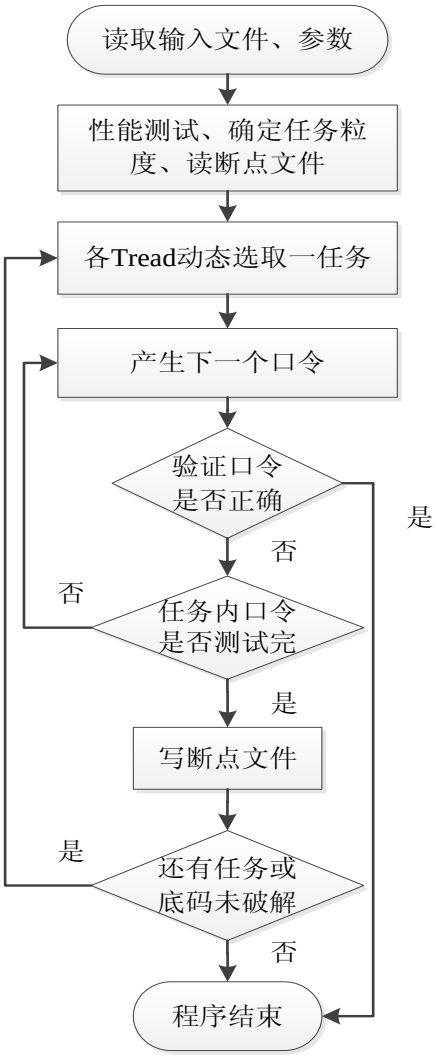

图 4-11 CPU 程序执行流程

4.3.3 基于 GPU 的 MD5 破解算法

4.3.3.1 任务分割

MD5 是计算密集型密码算法，适合用 GPU 进行并行计算，提高运算效率。通过对 MD5 算法和 GPU 模型的分析，我们将整个破解任务进行了分割，将有复杂控制机构和极少计算量的部分在 CPU（主机端、Host）实现，在 GPU（设备端、Device）实现计算密集部分。图 4-12 为任务分割示意图。

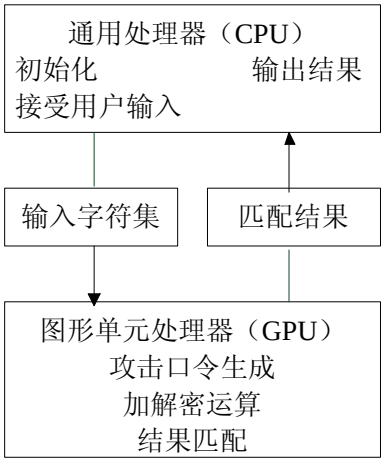

图 4-12 任务分割示意

4.3.3.2 并行算法设计

使用 MPI-CUDA 编程模式，采用两级任务划分，CPU 进程间并行采用粗粒度模式，任务划分，任务分配和传统 MPI 编程无异，静态或动态模式均可；CPU 划分的任务要交给绑定的 GPU 卡执行，GPU 线程并行采用细粒度模式，协同完成一个 CPU 任务，每完成一个 CPU 任务就和 CPU 进程交互一次，进行一些写断点和打印结果的操作，然后继续取下一个任务；以此类推，直至所有的任务完成。这里需要考虑 CPU 和 GPU 两级并行的协同。

首先由控制 CPU 核向其他 CPU 核发出 MPI 消息，轮询各 CPU 核绑定的 GPU 卡的运算能力，计算核心数等 GPU 设备信息；根据这些反馈回的信息，在控制 CPU 核上将欲测试的密钥空间划分为若干个子密钥空间，子密钥空间不重不漏地组成了整个密钥空间。测试完一个子密钥空间中的每一密钥即为完成一个破解子任务，完成所有子任务就完成了整个破解任务。各 CPU 核根据自身绑定的 GPU 卡的计算能力通过 MPI 与控制 CPU 核进行通信，自行向控制 CPU 核申请一个破解子任务，获得子密钥空间的起点，通过 GPU 线程产生子密钥空间中的所有密钥，并进行验证，返回验证结果。完成一个子任务后再继续申请下一个子任务，如果已经破解出密钥，则向控制 CPU 核发送任务完成，请求终止消息，控制 CPU 核响应后终止破解或启动下一条散列值的破解新任务，并输出破解结果。

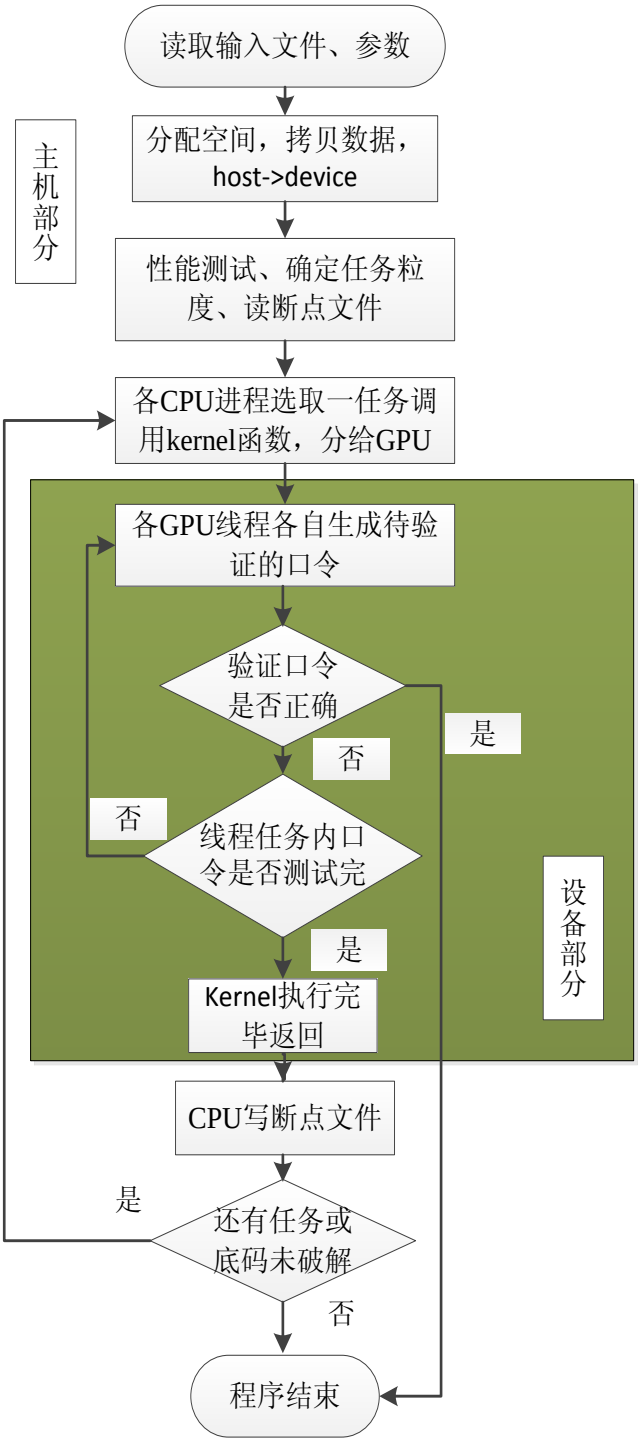

图 4-13 GPU 并行程序执行流程

GPU 并行程序执行流程如图 4-13 所示。各个 CPU 核获得破解子任务后调用 GPU 内核程序进行破解。GPU 将子密钥空间的起点读入显存中，以一个线程控制密钥生成，从起点向后一次生成数个不同的密钥，将这些密钥分配给多个不同的

线程进行 MD5 算法计算，结果与已知的散列值比较验证，返回验证结果。一般情况下每个 GPU 都可以同时处理上万个这样的密钥验证线程，因此采用这种并行破解方法可以获得很高的并行破解计算效率。

图 4-14 为破解程序执行流程：

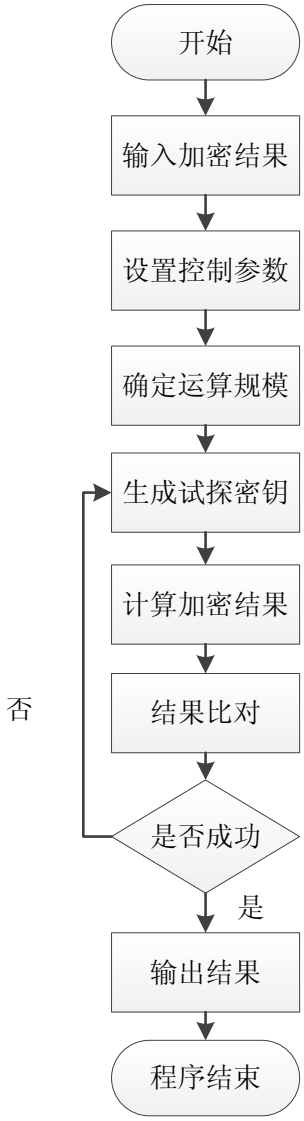

图 4-14 破解程序执行流程

1) 输入一个加密结果，指定密钥口令长度和密钥口令组合类型。
2) 根据对算法运行环境中 GPU 能力查询，确定并行运算的规模，即线程块数量和线程块的大小，并分配所需 CPU 端存储和 GPU 端全局存储。由于算法运行的设备的计算资源可能差距较大，因此需要按照设备的能力来设计并行运算的规模，而不是固定并行规模。

3) 按照密钥口令要求，由数据准备核函数在 GPU 上生成试探密钥数据并写到全局存储，目的是利用 GPU 的海量并行能力来减少准备试探密钥数据的时间，同时减少 CPU 准备数据传输到 GPU 花费的传输时间，利用了 GPU 全局存储读写速度较快的特性。
4) 由计算核函数在 GPU 上开始一轮运算，按照线程号并行读取密钥口令数据计算加密结果，通过与输入加密结果比对，若比对成功，将线程号写入结果中。该核函数中将原加密算法中大量的判断语句，转化为计算或是通过分析确定去掉，利用了 GPU 并行计算的优势，此外，每个线程分别比对结果，利用了 GPU 海量并行的优势，降低 CPU 计算的开销。
5) 在 CPU 中对核函数结果进行检查，根据分析是否产生结果确定是否需要输出密钥（由结果中线程号所对应，第 3）步中生成存储在设备的全局存储上）停止运算清理存储或继续下一轮密钥尝试。

实现流程，主机端代码：

Step1：启动 MPI，初始化 CUDA，轮询各卡的计算能力，多卡时加上设备号。

Step2：读取并分析输入参数，指定密钥长度和密钥组合类型；

Step3：读输入数据文件 GPM_readfile，读取至少一个加密结果；

Step4：在 CPU(host)端和 GPU(device)端分别分配内存和显存空间，存放输入或输出数据；

Step5：将 CPU 主机端(host)内存中的数据拷贝到 GPU 设备端(device)的显存；

Step6：性能测试，计算任务粒度、任务数

Step7：任务分配，CPU 并行执行各任务，针对每个任务，CPU 进程执行；

Step7.1：调用 GPU 设备端(device)的 kernel 核函数进行计算，并将结果写入显存

Step7.2：将 GPU 设备端(device)显存的结果回传到 CPU 主机端(host)内存中对应位置

Step7.3：使用 CPU 进程判断内核函数返回的结果，并进行结果处理

Step8：释放在 CPU 主机端(host)申请的内存空间和 GPU 设备端(device) kernel 核函数申请的显存空间，打印破解结果，程序结束。

设备端代码：

输入参数：任务起点地址，线程块数，每块线程数，块内线程 ID 号等

返回值：0 破解失败；非 0 破解出的密文编号

Step1：确定本线程穷举口令的起始位置；

Step2：由起始位置，产生第一个口令 pwd；

Step3：针对线程粒度大小，执行操作；

Step3.1：调用验证函数，对 pwd 是否真口令进行验证 GMP_pwdCheck，正确结果写入结果空间 out

Step3.2：产生下一个口令 next，继续 Step3.1

Step4：返回破解出来的口令个数 matched；

CPU 端函数，读密文数据文件

功能：从指定文件中读取破解所需要的数据

函数定义：int GMP_readfile(char *infile)

参数：从命令行传递进来的输入文件名

返回值：返回值大于 0，表示要破解密信息个数；返回值小于等于 0，表示读取数据无效或失败，程序退出。

此段是 CPU 端代码，用标准 C 实现。

GPU 端函数，数据拷贝

函数定义：__host__void User__CopyConstantDatatoDevice()

功能：拷贝数据，公共 host 数据拷贝到 device 空间中

参数：无

返回值：无

核心验证：

功能：验证给定的单个口令字的正确性

函数定义：

```
__device__int GMP_pwdCheck(
    unsigned char *passwd,   //口令
    int len,                 //口令长度
    int matched,             //已经破译口令个数
    unsigned char *d_out)    //破译结果存放空间
```

返回值：大于 0 的整数 n，表示口令满足 n 个密文；

等于 0，该口令字不满足条件；

小于 0，验证过程出错。

GPU 端程序

```
__host__ void User_CopyConstantDatatoDevice()
{

CUDA_SAFE_CALL(cudaMemcpyToSymbol("d_mbSum",&mbSum,sizeof(int)));
CUDA_SAFE_CALL(cudaMemcpyToSymbol(d_HASH,HASH,16*Max));

}
__device__ int GMP_pwdCheck(unsigned char *passwd,int len,int matched,unsigned char *d_out)
{
int i,j;
unsigned int flag;
int threadID;
int ismatch=0;
unsigned char state[16];

threaded = (blockDim.x*blockIdx.x) + threadIdx.x;

MD5(passwd,len,state);

for(j=0;j<d_mbSum;j++)
{
if(!d_memcmp(state,d_HASH[j],16))
{
    i=(threadID*OUTPUT_INT_NUM*OUTPUT_BYTE_NUM) + (matched+ismatch)*OUTPUT_BYTE_NUM;
d_out[i] = (j+1)<<8;
d_out[i+1] = (j+1)&0xff;
d_out[i+2] = passwd[0];
d_out[i+3] = passwd[1];
 •••
```

```
d_out[i+17]= passwd[15];
ismatch++;
}

}
return ismatch;
}
```

4.4 本章小结

本章首先简要介绍了 MD5 算法，并详细分析了 MD5 算法的破解原理。其次在基于 GPU 的 SMP 集群上采用 MPI-CUDA 混合编程方法，通过任务分割、并行算法设计将传统的串行 MD5 破解算法改造为可以进行大规模并行运算的破解算法。最后列出了部分核心函数流程和源代码。

第五章　实验结果与分析

5.1 实验环境及配置

本项目的实验环境为双节点的 SMP 集群，分别为节点 1 和节点 2，每个节点为具有支持 CUDA 的 4 块 GeForce GTX 480 显卡的对称多核处理器计算机（SMP），通过千兆网线和交换机连接成集群系统。每个节点计算机配置如下：Intel I7 930 四核高性能处理器 ，2.00GB DDR3 1333MHz 内存，64 位 Microsoft Windows 7 操作系统，Microsoft Visual Studio 2008，MPICH2 函数库，CUDA 4.0。

5.1.1 MPI 环境配置

MPICH 是 MPI 标准的一种最重要的具体实现，可以免费从网上下载。MPICH 目前的通用版本是针对 1997 年在 MPI 论坛上公布的 MPI-2 标准的 MPICH2，这是在原来的 MPI-1 基础上扩充了一些功能后得到的，这些扩充的功能包括：动态任务管理、单边通讯等。

安装 MPICH2 需要注意以下问题：

(1) 每台计算机都必须拥有网络设备，并能够建立 TCP/IP 网络连接，

(2) 每台计算机都用管理员身份登录进行安装。

(3) 在每台计算机上建立一个相同用户名的账户。

(4) 在所有计算机上安装好 MPICH2 之后，在每台计算机上还必须进行注册，在作为系统控制的计算机上还必须进行配置后，整个系统才能使用。注册是将每台计算机上申请的账户与密码注册到 MPICH2 中去，使得 MPICH2 可以通过网络访问系统中的每台计算机。

5.1.2 CUDA 环境配置

首先从 http://developer.nvidia.com/cuda-downloads 下载 CUDA 相关软件，包括 CUDA TOOLKIT、Drivers 和 CUDA SDK。

其次安装 CUDA 软件需要以下步骤：

1、安装 CUDA TOOLKIT

运行 Toolkit 的安装程序包，并按照屏幕上的提示，一步步的安装 CUDA Toolkit。CUDA Toolkit 默认安装在 C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v#.#，其中 v#.#是版本的名称，如 v4.0。这个目录里包括以下的内容，如图 5-1 所示：

图 5-1 CUDA Toolkit 文件目录

1）.bin 文件夹：包括编译器的可执行文件和运行时库;

2）.include 文件夹：包括编译 CUDA 程序所需的头文件;

3）.lib 文件夹：包括链接 CUDA 程序所需的库文件;

4）.doc 文件夹：包括 CUDA C Programming Guide (CUDA C 语言编程指南)、CUDA C Best Practices Guide (CUDA C 语言的最佳实践指南)、CUDA 库文件文档，以及其他 CUDA Toolkit 相关的文档。

注意：CUDA Toolkit 3.1 或者更早的版本，默认的安装路径为 C:\CUDA，所以需要在安装新版本 CUDA Toolkit 前将旧版本卸载。CUDA Toolkit 3.2 开始，多个 CUDA Toolkit 版本可同时安装。

2、安装 GPU Computing SDK

运行 GPU Computing SDK 安装程序包，并按照屏幕上的提示进行安装。GPU Computing SDK 默认的安装路径是 C:\Documents and Settings\All Users\Application Data\NVIDIA Corporation\NVIDIA GPU Computing SDK 4.0(或

者 Windows Vista 及以上系统，默认安装路径为 C:\ProgramData\NVIDIA Corporation\NVIDIA GPU Computing SDK 4.0)，该文件夹包含许多示例问题的源代码和微软 Visual Studio 的模板。

5.1.3 MPI,CUDA 与 Visual Studio 编译器的整合

MPI（Message Passing Interface）是一种进程间通信方式为消息传递的并行程序设计的标准，是并行程序设计事实上的工业标准。为了在新建的 VC++程序中可以正常调用 MPI 函数和 CUDA 函数，需要对编译环境做如下修改设置：

(1) “我的电脑”path 中设置 MPICH2 的 bin 目录（%MPICH2%/bin）以便运行 mpiexec 程序。我的具体设置如下：右键“计算机”——“属性”——点击窗口右侧“高级系统设置”——选择“高级”标签——点击“环境变量”按钮——在系统变量栏找到变量 path,在其后添加你的 mpich 安装路径，我的是 D:/Program Files (x86)/MPICH2/bin。注意和已有的系统变量之间用分号隔开。

(2) 在 VS 的 IDE 菜单栏中选择：工具—>选项—>项目和解决方案—>VC++目录，在其中添加包含文件和库，就是在 VS2008 或其它 VS 版本（2010 除外）的菜单栏找到“工具”菜单然后按提示包含 mpich 安装目录中的 include 文件夹和 lib 文件夹。

(3) 在 VS 的工程设置对话框中选择：项目—>属性—>配置属性—>连接器—>输入—>附加依赖项，在其中添加“cudart.lib cutil32D.lib mpi.lib”。

(4) 高亮显示：复制 C:/ProgramData/NVIDIA Corporation/NVIDIA GPU ComputingSDK 3.2/C/doc/syntax_highlighting/visual_studio_8 目录下的文件 usertype.dat 到 D:/Program Files (x86)/Microsoft Visual Studio 9.0/Common7/IDE 目录下；在 VS2008 中，工具〉选项〉文本编辑器〉文件扩展名，添加扩展名为 cu，编辑器为 Microsoft Visual C++的项目，添加扩展名为 cuh，编辑器为 Microsoft Visual C++的项目；重启 VS，高亮显示出现了。

5.2 实验过程、代码优化及结果

我们的实验项目如下：

穷举破解事先生成的 3 个文件 hash1.dat、hash2.dat、hash3.dat 中存储的 MD5 散列值，分别存储有 10 个、50 个和 100 个 MD5 散列值。选取口令长度为小于等于 8 个字符，字符选取空间 62 个字符（即所有大写字母、小写字母和数字）。

因此口令空间为 $62^8+62^7+62^6+ \bullet\bullet\bullet +62^1$，大约为 $2*10^{14}$。

在 Intel I7 930 4*2.8GHz CPU 上执行 8 线程并行破解程序，速度每秒大约 2400 万个口令（24million/s），运行完整个口令空间大约需要 107 天，即最坏情况下需要 107 天破解完全部 MD5 散列值。

实际运行过程中，鉴于口令密钥的实际复杂性的不同 CPU 程序穷举破解 hash1.dat、hash2.dat、hash3.dat 中 10 个、50 个和 100 个 MD5 散列值的破解分别耗时 183 秒，652 秒和 3780 秒。

5.2.1 CUDA 并行的实现

线程结构如图 5-2 所示：

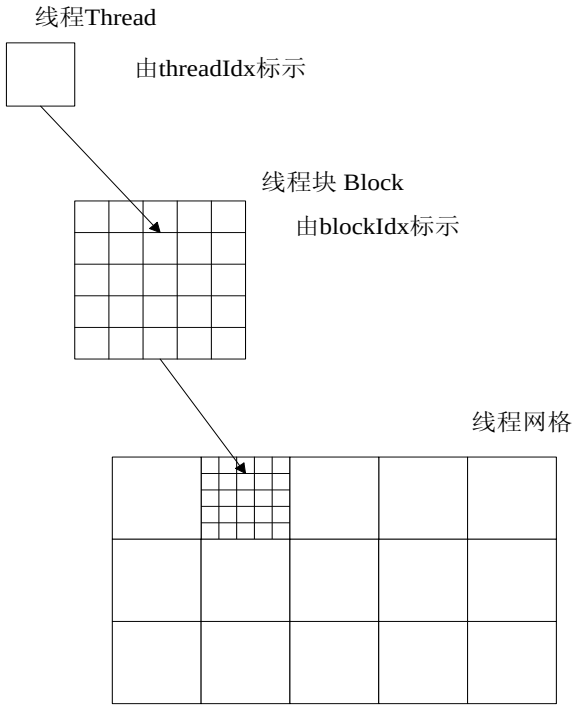

图 5-2 线程结构

Thread Block（线程块）：

– 3-Dimensional array of threads 三维线程

– Up to 512 threads 多达 512 个线程

– Can communicate through shared memory 可通过共享内存进行通信

– threadIdx

Grid of blocks（块网格）

– 2-Dimensional array of blocks 二维块

– blockIdx

一个并行执行内核 kernel 由任意数量的并行线程组成。

```
__global__void kernel(float a,float b){}
int main(){
        kernel<<<A,B>>>(1.0,1.0);
}
```

“A”的类型是 dim3，指定的是网格的尺寸。

“B”的类型是 dim3，指定的是块的尺寸。

可选的第三个参数：每块的共享内存的使用。

CUDA 实现过程：

-复制字母和目标散列到 device 设备端

-为解密的口令密钥分配内存

-计算网格尺寸和每个块中线程数的合理值

-调用并行内核 kernel，每个线程计算一个可能的口令密钥的 MD5 哈希值，并与目标散列进行比较。

Memory initialization 内存初始化

```
__device__ __constant__ uint target_d[4];
int main(){
        …

cudaMemcpyToSymbol(target_d,(void*)target_h,4*sizeof(uint),0,cudaMemcpyHostToDevice);
        …
        cudaMalloc((void **)&pass_d,32*sizeof(unsigned char));
        cudaMemset((void *) pass_d,0x00,32* sizeof(unsigned char));
        …
}
```

字母和目标散列值用常量内存。

全局内存用于解密的明文口令。

网格尺寸：

```
Void calcDims(dim3 &grid,dim3 &tpb,unit alphabet_size)
{
    grid.x=alphabet_size*alphabet_size*alphabet_size;//Max:65535
    grid.y=alphabet_size*alphabet_size;//Max:65535
    tpb.x = alphabet_size;//Max:512
}
```

在一块 GTX480 显卡上利用 CUDA 编程实现，速度大约每秒 1 亿 2 千万个口令（120million/s），运行完整个口令密钥空间大约需要 23 天，效率是 CPU 并行程序的 4.5 倍。在采用双节点，每个节点 4 块显卡的 SMP 集群上的效率大约是一块显卡的 8 倍，速度大约每秒 9 亿 6 千万个口令（960million/s），运行完整个口令密钥空间大约只需要不到 3 天，效率提高到 CPU 并行程序的 35 倍。

实际破解 hash1.dat、hash2.dat、hash3.dat 中 10 个、50 个和 100 个 MD5 散列值的破解分别耗时 5.2 秒、18.7 秒和 108 秒。

5.2.2 程序优化

在实验的基础上分析程序流程，优化程序，主要是优化 CUDA 部分：

由于 GTX480 中有 16384 字节的共享内存，而每个线程需要 512bits（64 字节），每个 block 块需要 32 字节用于存放函数参数，故而每个 block 块最多可以执行 255 个并行线程（255*64+32=16352）。图 5-3 为线程块中线程个数对程序性能的影响。

由图 5-3 可以看出并不是 block 块越多，程序的运行效率越高。随着每个 GPU 处理内核运行的 block 块数量增多，每个 block 块中的线程数必然减少，程序运行效率显著下降。在显存条件允许的情况下，当每个 GPU 处理内核运行的 block 块数为 1，每个 block 块中线程数为 255 时，密钥口令破解速度达到峰值 120million。

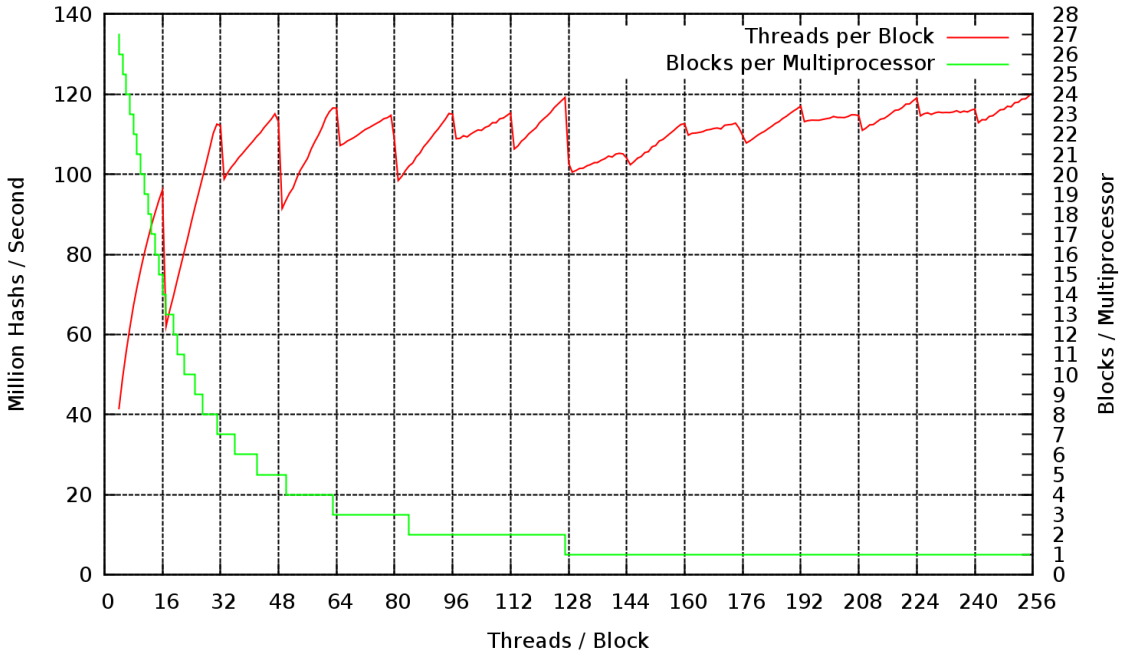

图 5-3 原始程序中线程块中线程个数对性能的影响

共享内存分为 16 组，不同的组可以同时访问，如果两次内存请求落到了同一组，则必须排队执行，连续的 32bit 字在不同的组，如图 5-4 所示。

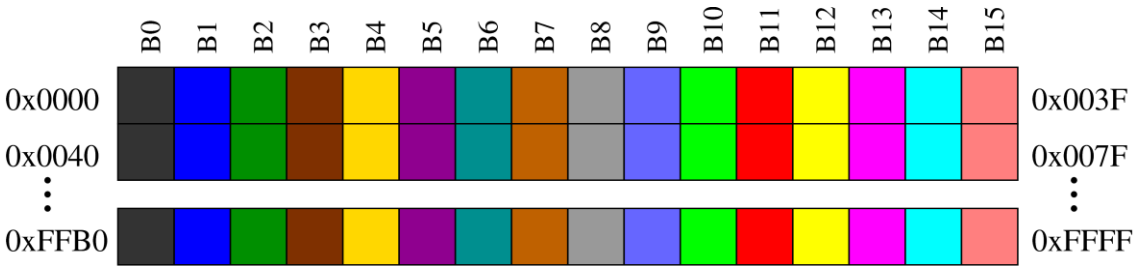

图 5-4 共享内存示意

下面两段源代码是对共享内存的两种操作，读取内存的情况分别如图 5-5，图 5-6 所示，效率显而易见。

```
void __global__ entry(uint *hashed,unsigned char* pass)
{
extern __shared__ uint memory[];
extern __shared__ char mem[];

uint *word = &memory[threadIdx.x * 16];
char *wword = (char *)word;
…
```

}

word[i]读取是一次读取 16 个分组，正好每次读取可以分配给共享内存的一个组，在一个 half-warp 中的每个线程只需要访问一次共享内存即可，并且每个线程可以同时访问。

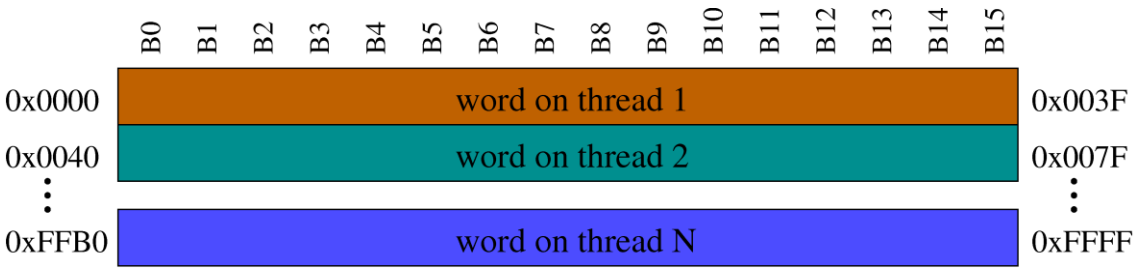

图 5-5 共享内存读取情况示意一

```
void __global__ entry(uint *hashed,unsigned char* pass)
{
extern __shared__ uint memory[];
extern __shared__ char mem[];

uint *word = &memory[threadIdx.x * 17];
char *wword = (char *)word;
…
}
```

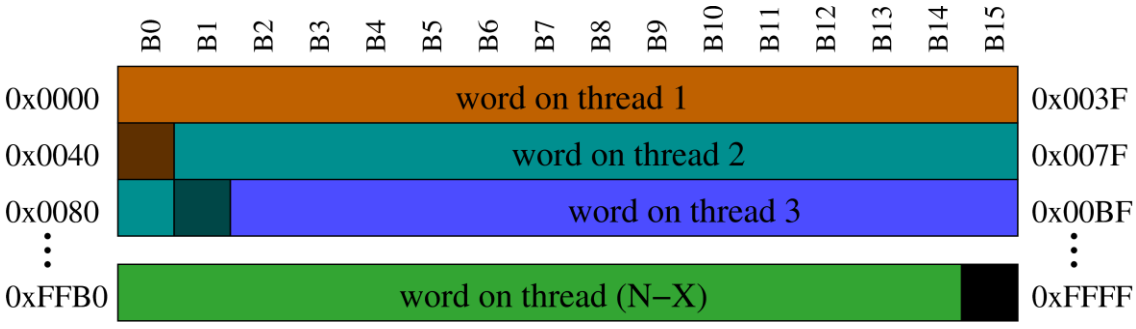

图 5-6 共享内存读取情况示意二

word[i]读取一次读取 17 个分组，在一个 half-warp 中必然存在两个线程访问同一个共享内存分组。线程之间存在访问冲突，线程访问每个 word[i]都要访问两次，且相邻的线程不能同时访问 word[i]，程序运行效率肯定会显著降低。

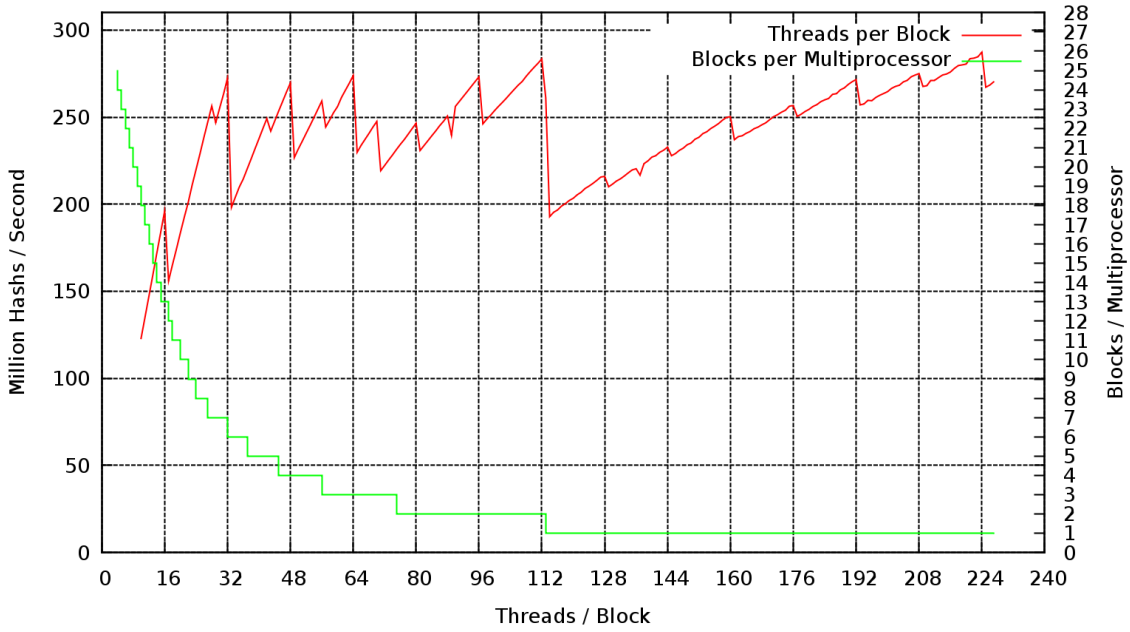

图 5-7 程序优化后线程块中线程个数对性能的影响

通过对破解程序中 CUDA 部分访问共享内存的优化，如图 5-7 所示，优化后在一块 GTX480 显卡上，口令密钥破解速度提高到每秒大约 2 亿 8 千万个口令(280million/s)，运行完长度为 8 的口令的整个口令密钥空间大约需要 9 天，效率是 CPU 并行程序的 11 倍。在实验用 SMP 集群上，只需要大约 1.1 天即可完成全部可能口令的验证，效率提高到 CPU 并行程序的约 95 倍。

实际破解 hash1.dat、hash2.dat、hash3.dat 中 10 个、50 个和 100 个 MD5 散列值的破解分别耗时 2 秒、6.1 秒和 36 秒。

表 5-1 和图 5-8 分别以表和图的方式给出了本文实验中所采用的三种破解方法所花费的破解时间的对比。

表 5-1 MD5 口令密钥文件破解时间对比

	hash1.dat （10 个 MD5）	hash2.dat （50 个 MD5）	hash3.dat （100 个 MD5）
CPU 破解	183 秒	652 秒	3780 秒
GPU 破解优化前	5.2 秒	18.7 秒	108 秒
GPU 破解优化后	2 秒	6.1 秒	36 秒

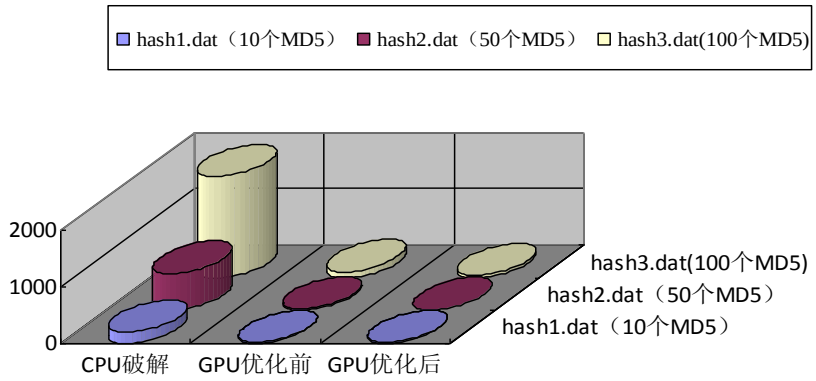

图 5-8 MD5 口令密钥文件破解时间对比（单位秒）

图 5-9 为 CPU 并行破解程序和优化前的 GPU 并行破解程序以及优化后的 GPU 并行破解程序的口令密钥破解速度比较。

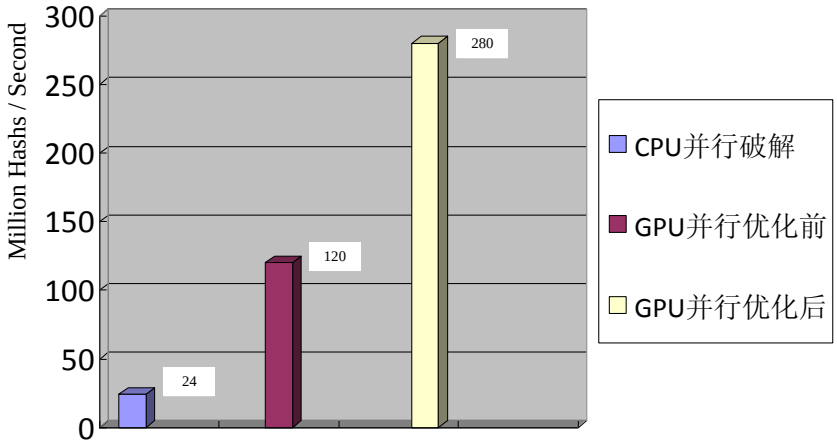

图 5-9 MD5 口令密钥破解速度比较

5.2.3 程序优化方法分析

通常情况下，MPI-CUDA 混合并行编程方法的程序优化需要采用下面的步骤：

（1）确定任务中只能串行执行的部分和能够采用大规模并行执行的部分，进行适合算法的选择。

（2）按照算法确定划分方式，对数据和任务进行划分，将每个需要并行实现的步骤准备为一个 MPI-CUDA 并行模型，先进行 MPI 层的并行，然后进行 CUDA

层的并行。

（3）编写一个能够正确运行的程序，作为优化的起点。

（4）优化显存访问，如果不能完全优化显存的访问带宽，那么其他部分的优化将不会产生明显效果。如：使用共享内存、合并访问等

（5）优化指令流。如：分支、循环展开、原子函数、CUDA 指令集、避免 banck conflict。

（6）资源平衡。调整 shared memory 和 register 的使用量。

（7）与主机通信优化。如：多个较小的传输可整合为一次较大的传输等。

5.3 算法实现方式的优缺点分析

在本文研究实现的 MD5 散列值快速并行破解系统中，进程之间通过 MPI 消息传递模型实现通信，进程内部通过 CUDA 并行编程模型实现 CPU-GPU 异构多核编程，从而屏蔽了系统的体系结构，实现了 CPU-GPU 混合体系结构系统的混合并行编程。

MPI 编程模型的优点是用户可以自行根据算法需要将数据进行分块划分并控制程序流程中进程间的同步时机，进而可以较为准确的控制数据的局部性和程序流程，优化程序执行效率。

MPI 在不同的工作模式下可以适合 SPMD（单线程多任务）或 MPMD（多线程多任务）等体系结构。MPI 对远程数据的访问是通过调用库函数进行进程间的消息传递来完成的，这个过程是显示的。这种显式的并行一般来说可以获得更好的性能和更强的可移植性。

在一个计算节点内可以存在多个 CUDA 设备，一个 CPU 核管理一个 CUDA 设备，从而使这些设备可以并行工作。采用这种方式建立的 CPU-GPU 混合体系结构系统可以提高机器的性能，节约空间和成本。

采用 MPI-CUDA 混合并行编程方式的优点分析

（1）负载不平衡得到了有效的改善。MPI 程序可扩展性差的主要原因是负载分配的不平衡和程序运行中的细粒度并行。采用混合并行编程模型，MPI 仅需要实现节点间通信，建立进程间的粗粒度并行；而节点内使用 CUDA 实现细粒度并行，基本不存在负载不平衡的问题，故而，能够有效提升程序性能。

（2）减少了进程间通信。通过在进程间进行粗粒度并行，进程内部进行细粒

度并行，这种多层次混合粒度编程的实现，有效减少了进程间相互通信的次数，更有利于性能的提高。

（3）实现了通信和计算的重叠。使用单线程方式实现的 MPI 比如 MPICH，虽然可以减少同步时的开销，但是由于必须等待前面计算的结果，所以程序的执行效率仍会显著降低。进程内的并行（线程并行）可以通过 GPU 片上共享内存、线程进行计算，实现通信和计算的重叠，有效提高了效率。

（4）在采用 CPU-GPU 混合体系结构的系统中，MPI 程序并不能实现对 GPU 的操作，这时，使用混合并行编程模型，在 CPU 上使用 MPI 进程，在 GPU 上使用 CUDA 线程，从而使所有 CPU 和 GPU 都得以高效运行。

（5）对于大规模的并行应用来说，使用基于 CPU-GPU 混合体系结构的系统（服务器、集群系统）可以在缩减硬件规模、能耗、系统调度开销等的同时达到同样的并行性能，甚至超越很多大规模并行硬件系统的性能。

缺点分析：

（1）虽然在 CPU-GPU 混合体系结构中，使用 MPI-CUDA 混合并行编程方式编写的程序执行效率更高。但是它也存在着一些不足，比如对于消息通信比较频繁的课题，CUDA 内不能实现所有线程的相互通信。比如对于内存要求大的课题，显存的大小也可能是一个限制。

（2）采用混合并行编程模型编写的程序能否取得更高的执行效率取决于以下几种因素：节点内是否有着更小的通信开销；是否可以用轻量级的线程并行来取代重量级的 MPI 进程并行；显存的使用和访问是否优化等，故而对程序员的要求较高。

5.4 本章小结

本章在双节点的 SMP 集群，每个节点为具有支持 CUDA 的 4 块 GeForce GTX 480 显卡的对称多核处理器计算机（SMP），通过千兆网线和交换机连接成集群系统的实验环境中对第四章的设计进行了实验，验证了设计的正确性。通过实验发现了原始程序实现的问题，并进行了优化，有效提高了破解系统的执行效率。最终将程序破解效率提高到传统 CPU 并行破解程序的约 95 倍。最后本章分析了本文采用的编程方式的优缺点并总结了算法实现的优化方法。

第六章　全文总结及工作展望

随着 GPU 计算能力的不断提高，CPU 多核化和 GPU 众核化的计算机体系架构将是未来计算机的主流发展趋势。在单台计算机上搭载多张 GPU 组成小型的 SMP 计算集群具有低耗、高效、易维护、易扩展的先天优势，将自然成为高密度并行计算的首选解决方案。Nvidia 公司不断推出适合通用计算领域的 GPU 软硬件架构 CUDA 的新的改进版本，极大的降低了传统 C 语言程序员编写适合 GPU 进行通用计算的并行程序的难度，有力的推动了 GPU 在通用并行计算领域的发展。本文围绕着 MD5 散列值并行破解程序和在基于 GPU 的 SMP 集群上的实现这一研究主题，进行了以下几个方面的工作：

（1）学习并研究了现有的几种主流并行技术 SMP 技术、MPP 技术、NUMA 技术和集群技术，确定以基于 GPU 的 SMP 集群来实现 MD5 散列值的并行破解程序。根据基于 GPU 的 SMP 集群的体系结构特点，决定采用节点间消息传递编程模型 MPI 和节点内 GPU 共享存储编程模型 CUDA 的两级混合并行编程模型进行程序开发。

（2）简要介绍了 SMP 集群系统，分析了 Fermi 架构的 GPU 和 CUDA 体系，较详细的介绍了基于 GPU 的 CUDA 编程技术，包括 CUDA 的硬件架构和 CUDA 的编程模型等，其中具体的有 Fermi 架构的 GPU 的 GPC 运算架构、CUDA 线程结构、CUDA 存储器模型、CUDA 软件体系等。

（3）简单介绍了 MD5 哈希算法，利用 MPI 和 CUDA 开发多粒度 MD5 散列值并行破解程序的原理和实现方法。对 MD5 破解算法的并行实现进行了可行性分析，并详细阐述了在基于 GPU 的 SMP 集群上 MD5 散列值破解程序算法的设计与实现。

（4）在基于 GPU 的 SMP 集群上实现了采用 MPI-CUDA 混合并行编程模式实现的 MD5 散列值破解程序，并与仅在 CPU 上实现的多线程并行破解程序进行了破解效率的比较。通过对 MPI-CUDA 混合并行编程模式下 CUDA 编程部分的优化，进一步有效提高了破解效率，并以此为基础初步分析了 MPI-CUDA 混合并行编程模式的优缺点和算法实现优化的方法。

时值论文交付之际，由于文章篇幅和时间所限，作者完成的工作尚有欠缺，

有待于进一步完善：

（1）目前采用的 MPI-CUDA 的两级混合编程模型在单节点多显卡 GPU 的 SMP 计算集群上的运算效率还没有达到理论峰值，究其原因，是单节点内控制 GPU 的多核 CPU 之间的通信存在延迟和消息冲突。未来可以研究 MPI-OpenMP-CUDA 的三级混合并行编程模型，节点间采用消息传递模型的 MPI 并行编程，节点内的 CPU 多核间采用共享内存模式的 OpenMP 编程模型，每个 CPU 内核控制的 GPU 内采用 CUDA 这种符合 GPU 结构特点的编程方式。

（2）目前在基于 GPU 的 SMP 计算集群上还只实现了 MD5 散列值并行破解算法，下一步还可以进一步实现 AES、WPAWPA2PSK 等计算密集度高的并行破解算法。

时至今日,采用 GPU 进行通用计算大行其道，GPU 的硬件架构还在不断发展中，适合 GPU 编程的 CUDA 也在不断改进中，因此，算法和实现方案也需要随之进行不断改进。随着基于 GPU 的 SMP 集群的不断发展，如何优化改写原有基于 CPU 的并行程序、如何合理利用二级共享缓存、如何有效控制传输带宽限制造成的延迟、如何将基于 CPU 和基于 GPU 的混合 SMP 集群捏合成一个整体发挥最大的效能等都是亟待解决的问题。本文的研究只涉及了其中很小的一部分，在今后的工作中可以对上述问题进行进一步的深入研究和实现。

致谢

首先要衷心感谢我在电子科大的老师们和我在单位的同事们，特别是我的导师赵志钦老师和白小霞副研究员，是他们在我这几年的研究生学习生活中给我的无私帮助，我将终生难忘。在平时的学习生活中，各位老师不辞辛劳的工作，使我在许多方面都达到了一个较高的层次。给我以后的工作与生活都有着非常有益的帮助！在论文的研究写作中，老师们无论在学习还是在生活上都给予了本人莫大的帮助和热情关怀，并为同学们提供了良好的设计环境，让大家学到了知识，掌握了学习的方法，也获得了实践锻炼的机会。老师学识渊博，治学态度严谨，勤恳敬业，自进行论文写作任务以来，在赵老师和白副研究员的悉心指导下，我查阅了大量资料，从不断的学习过程中，积累了许多宝贵的知识和经验。导师渊博的知识、严谨的作风和认真的研究态度，也使我受益匪浅，这一切将对我今后的工作和生活产生深远的影响。

其次，还要感谢的是与我朝夕相处的各位同学以及一直默默支持我的家人。能够同你们这样一群渴求知识，渴望进步的同学一起挑灯夜读、讨论课题，共同进步是我最大的荣幸。

最后，祝愿所有关心帮助我的人身体永远健康！

参考文献

[1] 吴文玲，冯登国，张文涛.分组密码的设计与分析[M].北京：清华大学出版社，2009.

[2] Stinson D.密码学原理与实践[M].冯登国,译.北京:电子工业出版社,2003:78.

[3] R.L.Rivest. The MD5 message-digest algorithm, Request for Comments (RFC1320)[P]，Internet Activities Board, Internet Privaey Task Force, 1992.

[4] IONES P.US Secure Hash Algorithm 1(SHA1) RFC3174[P], 2001.

[5] 曾剑平，郭东辉.一种有效支持计算机取证的审计机制研究[J].计算机工程，2006，32(6)：148-150.

[6] 张建伟，李鑫，张梅峰.基于MD5算法的身份鉴别技术的研究与实现[J].计算机工程，2003，29(4)：118-119.

[7] MorelandK.,Angel E. The FFT on a GPU[J]. In: proeeedings of Graphies Hardware，SanDiego，2003:112-119.

[8] NVIDIA CUDA Compute Unified Device Architecture Programming Guide[J].6/23/2007.

[9] Markus Unger, Thomas Pock, Horst Bischof. Continuous Globally Optimal Image Segmentation with Local Constraints[J]. Computer Vision Winter Workshop 2008,2008.

[10] Bryan Catanzaro, Narayanan Sundaram, Kurt Keutzer. Fast Support Vector Machine Training and Classification on Graphics Processors[J]. In: Proceedings of the 25th International Confer-ence on Machine Learning, Helsinki, Finland, 2008: 104-111.

[11] B. Pieters, D. Van Rijsselbergen, W. De Neve, et al. Performance Evaluation of H.264/AVC Decoding and Visualization using the GPU[J]. In: Proceeding of SPIE on Applications of Digital Image Processing, 2007: 669606.1-669606.13.

[12] Randy Smith, Neelam Goyal, Justin Ormont, et al. Evaluating GPUs for Network Packet Signature Matching[J]. In: Proceedings of the International Symposium on Performance Analysis of Systems and Software, 2009: 175-184.

[13] Michael M. The GPU enters computing's mainstream[J].IEEE Computer society，2003.36(10): 106-108.

[14] S.Ryoo, C.I.Rodrigues, S.S.Stone, et al. Program optimization study on a 128-core GPU[J]. In: The First Workshop on General Purpose Processing on Graphics Processing Units, 2007:

137-145.

[15] D.Tarditi, S.Puri, J.Oglesby. Accelerator: Using data parallelism to program GPUs for general-purpose uses[J]. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, 2006: 325-335.

[16] Buck. Brook Specification v0.2[R], 2003.

[17] K. Fatahalian, J.Sugerman, P.Hanrahan. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication[J]. In: Proceedings of the 2004 ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, 2004: 133-137.

[18] 钱睿智. 基于GPU加速的MD5哈希函数加密算法研究[D].武汉：华中科技大学,2009.

[19] Randima F.GPU Gems:Programming techniques,Tips&Trieks for Real-Time Graphics[R]. Addison-Wesley Professional.2004.

[20] C.J.Thompson, S.Hahn, M.Oskin. Using modern graphics architectures for general-purpose computing: a framework and analysis[J]. In: ACM/IEEE International Symposium on Microarchitecture, 2002: 306-317.

[21] J.Kruger, R.Westermann. Linear algebra operators for GPU implementation of numerical algorithms[J]. In: ACM SIGGRAPH International Conference on Computer Graphics and Interactive Techniques, 2003: 908-916.

[22] M.L.Curry, A.Skjellum, H.L.Ward, et al. Accelerating Reed–Solomon coding in RAID systems with GPUs[J]. IPDPS, 2008, 14(18): 1-6.

[23] G.Falcao, L.Sousa, V.Silva. Massive parallel LDPC decoding on GPU[J]. In: ACM SIGPLAN Symposium on Principles and practice of Parallel Programming, 2008: 83-90.

[24] G. Kedem, Y. Ishihara. Brute Force Attack on UNIX Passwords with SIMD Computer[J]. In: Proceedings of the 8th conference on USENIX Security Symposium, 1999: 93-98.

[25] Harrison, J. Waldron. AES encryption implementation and analysis on commodity graphics processing units[J]. In: Workshop on Cryptographic Hardware and Embedded Systems, 2007:209-226.

[26] Harrison, J. Waldron. Practical symmetric key cryptography on modern graphics hardware[J]. In: USENIX Security Symposium, San Jose, CA, 2008: 195-209.

[27] S. A. Manavski. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography[J]. In: IEEE International Conference on Signal Processing and Communications, 2007: 65-68.

[28] Moss, D. Page, N. Smart. Toward acceleration of RSA using 3D graphics hardware[J]. In:

Cryptography and Coding, 2007: 364-383.

[29] S.Sinha, J.M.Frahm, M.Pollefeys. GPU-based Video Feature Tracking and Matching[R]. Tech. Rep. TR06-012, University of North Carolina at Chapel Hill. 2006.

[30] Kimmo Jarvinen, MattiTommiska, Jorma Skytta. Hardware Implementation Analysis of the MD5 Hash Algorithm[R]. In: Proceedings of the 38th Hawaii International Conference on System Sciences, Hawaii, United States, 2005 .

[31] Bert den Boer, Antoon Bosselaers. Collisions for the Compression Function of MD5[J]. In: Proceedings of EUROCRYPT, 1993: 293-304.

[32] Andrey Belenko. Faster Password Recovery with Modern GPUs[R]. TROOPERS 08,2008.

[33] H. Dobbertin. Cryptanalysis of MD5 compress[M]. In: Rump Session of EuroCript, 1996.

[34] Xiaoyun Wang, Hongbo Yu. How to Break MD5 and Other Hash Functions[J]. EUROCRYPT, 2005: 19-35.

[35] Marc Stevens, Arjen K. Lenstra, Benne de Weger. Vulnerability of software integrity and code signing applications to chosen-prefix collisions for MD5[R].2007.

[36] 乐德广，常晋义，刘祥南等. 基于GPU 的MD5 高速解密算法的实现[J].计算机工程,2010,36(11),154-158.

[37] 张润梅，王霄. 基于 CUDA 架构的 MD5 破解方法研究[J].计算机科学.2011,38(2):302-305.