基于CUDA架构的MD5破解方法研究

张润梅 王 霄
(安徽建筑工业学院电子与信息工程学院 合肥 230022)

摘 要 由于内存、运算速度以及磁盘空间的限制,暴力破解MD5几乎无法在PC机上实现。CUDA意在使GPU的超高计算性能在数据处理和科学计算等通用计算领域发挥优势。主要研究基于CUDA架构的MD5破解方法,并使用VS2005与NVCC进行混合编译。实验选择在GeForce9600GT显卡和四核CPUQ6600上分别运行所提程序和标准C语言版程序。结果表明,在高计算负荷与巨量数据情况下,中低端显卡的计算速度比高端CPU高30～50倍。CUDA使GPU流处理器阵列的性能得到充分发挥,极大地提高了并行计算程序的效率。

关键词 MD5破解,CUDA,并行计算
中图法分类号 TP181 **文献标识码** A

MD5 Crack Method Based on Compute Unified Device Architecture

ZHANG Run-mei WANG Xiao
(School of Electronics and Information Engineering, Anhui University of Architecture, Hefei 230022, China)

Abstract CUDA is intended to give full play to the advantages of ultra-high computing performance of GPU in data processing, scientific computing and other fields of general purpose. This paper studied MD5 crack method based on Compute Unified Device Architecture and carried on hybrid compilation by using VS-2005 and NVCC. The experiment runs CUDA programs on GeForce-9600GT graphics card and runs Standard-C programs on Quad-Core CPU-Q6600. The results shows that, under the environment of high computational load and huge amounts of data, the computing speed of mid and low end graphics card is 30 to 50 times higher than that of high end CPU. CUDA gives full play to the advantages of GPU Streaming Multiprocessors Array and greatly improves the efficiency of the parallel computation programs.

Keywords MD5 crack, CUDA, Parallel computation

随着科学技术的飞速发展,人们对高速并行计算的要求与日俱增。并行计算技术由于需要大数量的PC机集联,对网络的速度和稳定性要求较高,算法编写难度较大,且数据错误率也较高,因此难以大规模应用。为了应对这一局面,全球各大公司都推出了自己的单机并行计算架构,并设计出与之相应的计算芯片,如IBM的CELL(单CPU+多协处理器)和Sun公司的Niagara(多核心共用I/O)以及Intel的多核心多总线处理器。

从微架构上看,CPU擅长处理操作系统、系统软件和通用应用程序这类拥有复杂指令调度、循环、分支、逻辑判断以及执行等程序任务。程序逻辑的复杂度限定了程序执行的指令并行性,在CPU上难以完成上百个线程并行执行程序。而GPU(Graphic Processing Unit)擅长图形类兼非图形类的高度并行数值计算,可以容纳上千个没有逻辑关系的数值计算线程。GPU在数值计算方面的优势主要是浮点运算,运算速度快主要是靠大量并行,但这种数值运算的并行性在逻辑判断执行方面却发挥不了优势。因此,GPU特别适合于并行数据运算的问题,即一个程序处理大量并行数据元素,且具有高运算密度(算数运算与内存操作的比例)。

但传统的GPU通用计算,还存在以下问题:

(1)GPU只能通过图形API来编程,复杂且难以学习、使用,所以在非图形领域的应用很不充分;

(2)GPU编程缺乏灵活性,对GPU性能发挥有很大限制;

(3)存在带宽瓶颈,不能充分发挥GPU的计算能力。

CUDA(Compute Unified Device Architecture,统一计算设备架构)由NVidia公司在2007年推出,意在使GPU的超高计算性能在数据处理和科学计算等通用计算领域发挥优势。

本文分别对256MB,512MB,1G数据量的明码进行MD5化,并分别在NVDIA GF9600GT芯片与Intel CoreQuand芯片上进行计算,根据计算时间来评估两者在大规模并行计算环境中计算能力的差距。

1 CUDA编程模型

CUDA的基本思想是尽量使任务线程级并行(Thread

到稿日期:2010-03-22 返修日期:2010-06-28 本文受建设部科研开发项目(2009-K9-11),安徽省自然科学基金项目(090412057),安徽省教育厅自然科学研究重点项目(KJ2009A020Z)资助。

张润梅(1971－),女,教授,主要研究方向为多Agent系统、机器人技术、并行计算。

Level Parallel)，这些线程能够在硬件中被动态地调度和执行。CUDA 编程模型是将 CPU 作主机(Host)，而 GPU 作为服务器(Server)进行并行性计算。

通过 CUDA 编程时，将 GPU 看作可以并行执行多个线程的计算设备(Compute Device)，用于处理应用程序中数据并行的、计算密集的部分。对不同数据执行相同操作的应用程序部分可以独立放在此设备上作为许多不同线程执行的函数。GPU 将这样一个函数的指令集中编译，把得到的程序加载到设备。主机和设备都保留自己的 DRAM(Dynamic Random Access Memories，动态随机存储器)，分别称为主机内存(host memory)和设备内存(device memory)。用户可以通过优化的 API 使用设备的高性能直接存储器存取引擎将数据从一个 DRAM 复制到其他 DRAM 中。

如图 1 所示，执行内核的线程(thread)被组织成线程块(block)，而线程块又组成了栅格(grid)。线程块是可以一起协作的线程批(batch)，它们通过一些快速的共享内存有效地共享数据并同步执行，以协调内存访问。用户可以在内核中指定同步点，线程块中的线程在到达此同步点时挂起。每个线程由线程 ID 标识，这是线程块中的线程号。为了帮助基于线程 ID 的复杂寻址，应用程序还可以将线程块指定为任意大小的一维、二维或三维线程阵列，并使用 1 个、2 个或 3 个索引分量来标识每个线程。对于大小为 Dx 的一维线程块，索引为 x 线程的线程 ID 为 x；对于大小为 (Bx,By) 的二维线程块，索引为 (x,y) 线程的线程 ID 为 $(x+yBx)$；对于大小为 (Bx,By,Bz) 的三维线程块，索引为 (x,y,z) 线程的线程 ID 为 $(x+yBx+zBxBy)$。

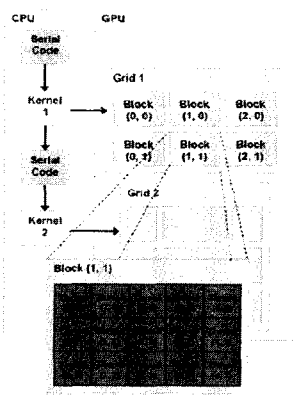

图 1　线程组织结构示意图

一个线程块仅由一个流式多处理器(Streaming Multiprocessors)处理，共享内存空间驻留在芯片共享内存中。SM 的寄存器和共享内存划分给线程块批的所有线程，每个多处理器一批可以处理的线程块的数量取决于内核中每个线程块需要的寄存器和共享内存的数量。如果每个多处理器没有足够的可用寄存器或共享内存来处理至少一个块，则内核将无法启动。在一个批次内并被一个多处理器处理的线程块被称为活动(active)线程块。每个活动线程块划分到被称为线程束(warp)的 SIMD(Single Instruction Multiple Data，单指令多数据)线程组中。其中，每个线程束包含相同数量的线程(此数量被称为线程束尺寸)，并以 SIMD 方式由多处理器执行。活动线程结束(比如所有活动线程块中的所有活动线程束)是分时的；线程调度器(thread scheduler)定期从一个线程束切换到另一个线程束，以便最充分地使用多处理器的计算资源。

2　使用 CUDA 进行 MD5 破解

2.1　MD5 破解原理

MD5(Message-digest Algorithm 5，信息-摘要算法)是在 20 世纪 90 年代初由 MIT Laboratory for Computer Science 和 RSA Data Security Inc 的 Ronald L. Rivest 开发，经 MD2，MD3 和 MD4 发展而来的。它的作用是让大容量信息在用数字签名软件签署私人密钥前被“压缩”成一种保密的格式，即把一个任意长度的字节串变换成一定长的大整数。MD5 的典型应用是对一段 Message(字节串)产生 fingerprint(指纹)，以防止被“篡改”，这就是所谓的数字签名应用。

MD5 以 512 位分组作为输入信息，且每一分组又被划分为 16 个 32 位子分组，经过了一系列的处理后，算法的输出由 4 个 32 位分组组成，将这 4 个 32 位分组级联后将生成一个 128 位散列值。

在 MD5 算法中，首先需要对信息进行填充，使其字节长度对 512 求余数的结果等于 448。因此，信息的字节长度(Bits Length)将被扩展至 $N*512+448$，即 $N*64+56$ 个字节(Bytes)，N 为一个正整数。填充的方法如下：在信息的后面填充一个 1 和无数个 0，直到满足上面的条件时才停止用 0 对信息的填充。然后再在这个结果后面附加一个以 64 位二进制表示的填充前的信息长度。经过这两步的处理，现在的信息字节长度 $=N*512+448+64=(N+1)*512$，即长度恰好是 512 的整数倍数。这样做的原因是满足后面处理中对信息长度的要求。MD5 中有 4 个 32 位被称作链接变量(Chaining Variable)的整数参数，它们分别为 $A=0x01234567$，$B=0x89abcdef$，$C=0xfedcba98$，$D=0x76543210$。当设置好这 4 个链接变量后，就开始进入算法的 4 循环运算，循环的次数是信息中 512 位信息分组的数目。

将上面 4 个链接变量复制到另外 4 个变量中：A 到 a，B 到 b，C 到 c，D 到 d。主循环有 4 轮(MD4 只有 3 轮)，每轮循环都很相似。第一轮进行 16 次操作。每次操作对 a，b，c 和 d 中的其中 3 个做 1 次非线性函数运算，然后将所得结果加上第 4 个变量(文本中的 1 个子分组和 1 个常数)。

再将所得结果向右环移 1 个不定的数，并加上 a，b，c 或 d 中之一。最后用该结果取代 a，b，c 或 d 中之一。以下是每次操作中用到的 4 个非线性函数(每轮 1 个)：

$$F(X,Y,Z)=(X\&Y)|((\sim X)\&Z)$$

$$G(X,Y,Z)=(X\&Z)|(Y\&(\sim Z))$$

$$H(X,Y,Z)=X\wedge Y\wedge Z$$

$$I(X,Y,Z)=Y\wedge(X|(\sim Z))$$

式中，& 是与，| 是或，～是非，^ 是异或。

如果 X，Y 和 Z 的对应位是独立和均匀的，那么结果的每一位也应是独立和均匀的。F 是一个逐位运算的函数，即如果 X，那么 Y，否则 Z。函数 H 是逐位奇偶操作符。所有这些完成之后，将 A，B，C，D 分别加上 a，b，c，d。然后用下一分组数据继续运行算法，最后的输出是 A，B，C 和 D 的级联。最后得到的 A，B，C，D 就是输出结果，A 是低位，D 为高位，$DCBA$ 组成 128 位输出结果。

由于原算法的复杂性，破译 MD5 密码的方法不同于传统密码破解使用的穷举法，而是一个先穷举生成随机原文，再用

MD5 程序计算出原文的字典项的 MD5 值，然后用目标的 MD5 值在字典中检索的过程。假设密码的最大长度为 8 位字节(8 Bytes)，同时密码只能是字母和数字，共 26＋26＋10＝62 个字符，排列组合出的字典的项数则是 $P(62,1)+P(62,2)\cdots+P(62,8)$，这是一个巨大的数字，存储这个字典就需要 *TB* 级的磁盘阵列。

由以上分析可知，暴力破解 MD5 的方法有如下特点：

• 计算量巨大。每一次都是对按照特定规则顺序生成的原文进行填充，然后开始进入算法的 4 轮循环运算(共 64 步)，每步都是非线性运算；

• 数据上下文依赖性几乎为零。对于每个原文的操作独立进行，计算过程独立，执行顺序对结果没有影响。

2.2 使用 CUDA 破解 MD5 的程序架构

从暴力破解 MD5 的特征可以发现，MD5 的暴力破解非常适合由大规模并行运算来完成，并提高了计算速度。本文设计的 CUDA MD5 破解程序核心算法是在 Ronald L. Rivest 于 1991 年开发的 MD5 码生成标准算法的基础上，使用带有 CUDA 扩展的 C 语言进行改写的。计算使用 Gforce9600gt 显卡的多计算核心，轻量级计算线程在多核心上大量开启，进行并行运算。计算开始后把包含字符“$A\sim Z, a\sim z$，^ -{\\}”且长度为 5 位的原文按照逐位变化的次序自动生成。

将生成的原文分段，依次交给 GPU 的多个流处理器进行计算，得到计算结果之后，与原始密码生成的 MD5 码进行比较，找出与其相一致的计算原文，由此得到原始密码。

MD5 破解程序流程如图 2 所示，代码分为 3 部分：main. cu，include. cu，md5. cu。

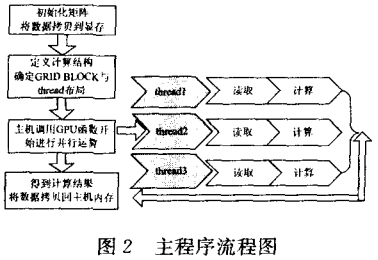

图 2　主程序流程图

main. cu 作为程序的文字界面部分，负责 CUDA 的初始化，完成数据文件的拷贝，设置 GPU 计算任务的分配工作(设置 gird，block，threads 等数值)，并激发显卡上的计算函数进行运算。之后比较由随机原文计算产生的 MD5 码值，返回和原始密码相同的原文并输出结果。最后注销程序在主内存和显存以及 GPU 内部的存储空间。

部分代码如下：

```
dim3 threads,blocks;// 核心载入的运行架构(如何分块，划分任务在核心中的分布)
//对要使用的空间进行填 0 处理
memset(g_input_md5,0,sizeof(g_input_md5));
//在主机上分配空间
unsigned char *a=0;// 指向在主机内存上的数组空间的指针
unsigned char *out=0;
CUDA_SAFE_CALL( cudaMallocHost((void **)&a,nbytes));//在主机上为 a 申请大小为 nbytes 的内存空间(作为输入)
memset(a,0,nbytes);//将 a 中的内容全部置 0
CUDA_SAFE_CALL( cudaMallocHost((void **)&out,nbytes));//在主机上为 out 申请大小为 nbytes 的内存空间(作为输出)
memset(out,0,nbytes);
// 为数据流申请创建空间
cudaStream_t *streams =(cudaStream_t*) malloc(nstreams * sizeof(cudaStream_t));
for(int i= 0;i<nstreams;i++)
CUDA_SAFE_CALL( cudaStreamCreate(&(streams[i])));
//创建显存上的计算空间
unsigned char *d_a=0,*d_o=0;// 定义指向数据在设备上位置的指针
CUDA_SAFE_CALL( cudaMalloc((void **)&d_a,nbytes));//在设备(显卡)上申请输入空间
CUDA_SAFE_CALL( cudaMalloc((void **)&d_o,nbytes));//在设备(显卡)上申请输出空间
//将设备上的内容置 0
cudaMemset(d_a,0,nbytes);
cudaMemset(d_o,0,nbytes);
//将数据从主机拷贝到设备(显卡)上
cpy_const_data_from_host_to_device();
// 对线程块和栅格进行设定(为了数据的连续性，在此分配为一维)
threads=dim3(THREAD_NUM,1);
blocks=dim3(THREAD_BLK_NUM,1);
```

include. cu 包含了 THREAD_NUM(每块的线程数量)、THREAD_BLK_NUM(运行块数)、NUM_PER_THREAD(每个线程的任务数量)等值，以方便修改参数，使得程序在 GPU 内部逻辑网格与块数量不同的情况下运行，同时还有 MD5_PASSWORD(原始密码)，以方便修改原始密码，从而验证程序运行结果的正确性。为了提高效率，必须让尽可能多的流处理器参与计算，以避免出现流处理器闲置的情况。同时要防止过多流处理器同时运算，造成处理速度减慢，甚至出现错误结果的现象。

include. cu 部分代码如下：

```
#define THREAD_NUM(256) //(每块的线程数量)
#define THREAD_BLK_NUM 16 //运行块数
#define NUM_PER_THREAD(1024 * 8 * 2) // 每个线程的任务数量
//定义初始密码(由明码经在线 MD5 计算系统生成)
#define MD5_PASSWORD {0x46,0xa0,0xe2,0x40,0xb5,0xe9,0x6a,0x95,0x56,0xfa,0xea,0x8c,0xd9,0xb2,0xcc,0x26}
```

注：定义块的数量(THREAD_BLK_NUM)为 16，使得每一个 SM 负责 2 个块，每个块(block)中则开启 256 个线程，最终同时有 4096 个线程参与到 MD5 编码计算当中。

md5. cu 文件包含了所有以_global_(仅在 GPU 上运行)，_device_(仅在设备上运行)，_host_(仅可通过主机调用)声明的函数，其中_host_ MDString 是整个程序的主体核心部分。

其中 MD5 转码步骤中所有的相关函数(MD5Init，MD5Update，MD5Final，Transform，MDString)均由 MD5 计算 C 语言版代码提取，再转化为 CUDA 版本。在转化和提取的过程中，严格按照原版代码的标准，每一步骤都独立测试并与原生版本做比对。

3　运行结果及分析

试验平台参数：

CPU：intel Q6600 2.4GHhz

主板 P35-TA

内存 DDR2 800 2GB

显卡 NVIDIA GF9600gt(核心代号:G92,流处理器数量:64,核心频率:650MHz,流处理器频率:1870MHz,理论峰值:208GFLOPS)

系统:windowsXP SP3

编译环境:VS2005+NVCC(nv 专用编译器)混合编译

注:最终计算时间采用记录 CPU 时钟次来最终统计,以确保准确。

程序编译调试后,使用 Nvidia 公司提供的 CUDA visual profiler 工具来分析程序的运行效率,得到程序运行过程中显卡端主函数 MDstring 对 GPU 的使用率(见图 3)以及 16 个进程流的流水线分布情况(见图 4)。

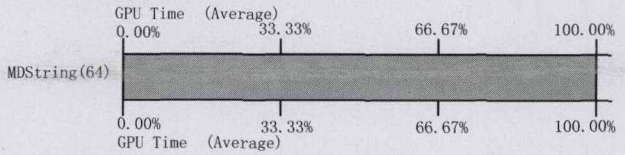

图 3 主函数 GPU 使用率

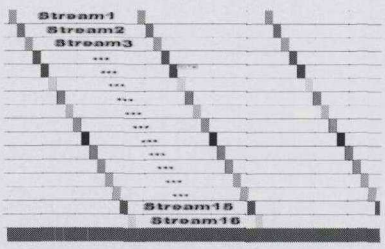

图 4 并行分布图(计算 512M 数据量)

由 CUDA visual profiler 分析图解得出,程序运行期间显卡端主函数 MDstring 对 GPU 的使用率为 100%,同时 16 组 SM 分时轮流参与到计算当中,共计 4 轮,成功避开了读取延时,形成了无缝链接的计算流水线,使得 GPU 流处理器阵列的性能得到充分发挥,极大地提高了程序计算效率。

如图 5 所示,程序完成后,使用 GPU 计算总量为 1024M 数据的结果,并顺利破解密码(计算开始不久已经破解,密码为 AAAAA),耗时约 37.328s,而完成 256M 与 512M 数据量的计算时间为 9.344s 和 18.373s。

图 5 计算时间结果图

计算数据量为 256M 的 CPU 计算耗时为 1679s,而完成 512M 与 1024M 数据量的计算时间为 3365s 和 6720s。C 语言原生版本程序运行期间 CPU 的使用率始终停留在 25%(传统单线程计算程序只能使用多核处理器的一个核心,本次使用的处理器为 Intel 四核处理器 Q6600,主频 2.4G),造成了对计算资源的极大浪费。对比图表如图 6 与图 7 所示。

	256M	512M	1024M
GPU	9.344	18.373	37.328
CPU	1679	3365	6720

图 6 计算时间对比表(单位为 s)

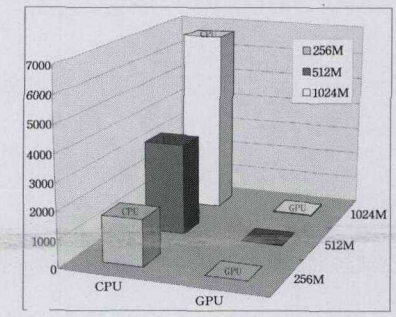

图 7 计算时间图形对比(单位为 s)

显卡流处理器频率仅为 1.8G,且不支持分支处理指令,却依靠较多的数量(64 个)和对数据计算的芯片级优化,在 MD5 破解速度胜过高端处理器。这使得相同数据量的计算时间缩短了 170 倍,即使完全发挥 Q6600 的四核心计算能力,流处理器阵列的计算速度也仍然胜出其 40 倍左右。

从上面分析得出,流处理器虽然主频较低,且不支持复杂指令,但其设计上专门为数值计算做出优化,加上在其上运行线程的支援开销较低且较容易实现线程级并行,使得流处理器阵列在数据计算上有着天生的优势。相比而言,支持各种复杂指令集的 CPU,纵然有着较高的主频,而其中也不乏多核产品,但是始终缺乏大众化的多核心编译工具,浪费了大部分计算资源。高昂的价格也使得普通用户难以接受。

结束语 本实验结果体现出 CUDA 在并行运算方面的巨大优势,其不仅计算速度非常高,且能最大化使用当前硬件资源。CUDA 程序编写最重要的是将算法中的并行性抽象提取,再将其进行转化。CUDA 的精髓在于大规模计算,更在于并行计算的单机化。使用具有精确数据计算能力的流处理器阵列,以较低成本在有限的资源和空间内完成了尽可能多的任务,达到了节约计算成本的目的。

参 考 文 献

[1] Corporation N. CUDA Programming Guide[EB/OL]. www.nvdia.com. 2009.2

[2] CUDA——走向 GPGPU 新时代[J]. 程序员,2008,10(3):36-39

[3] Nvidia. The CUDA Compiler Driver NVCC. 2008

[4] Tehver M. General Purpose GPU Programming with CUDA(演讲稿). 2007

[5] Portegies Z S F, Belleman R G, Geldof P M. High-performance direct gravitational N-body simulations on graphics processing units[J]. NewAstron, 2007, 12: 641-650

[6] Dongarra J, Foster I, Fox G, et al. Sourcebook of Parallel Computing[M]. Elsevier Science, 2003

[7] Kider Jr J T. GPU as a Parallel Machine: Sorting on theGPU[C]// Proc. of CIS'05. 2005