GPU 上的矩阵乘法的设计与实现①

梁娟娟，任开新，郭利财，刘燕君

(中国科学技术大学 计算机科学与技术学院，合肥 230027)

摘 要: 矩阵乘法是科学计算中最基本的操作，高效实现矩阵乘法可以加速许多应用。本文使用 NVIDIA 的 CUDA 在 GPU 上实现了一个高效的矩阵乘法。测试结果表明，在 Geforce GTX 260 上，本文提出的矩阵乘法的速度是理论峰值的 97%，跟 CUBLAS 库中的矩阵乘法相当。

关键词: 矩阵乘法；GPU；CUDA

Design and Implementation of Matrix Multiplication on GPU

LIANG Juan-Juan, REN Kai-Xin, GUO Li-Cai, LIU Yan-Jun

(School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China)

Abstract: Matrix multiplication is a basic operation in scientific computing. Efficient implementation of matrix multiplication can speed up many applications. In this paper, we implement an efficient matrix multiplication on GPU using NVIDIA's CUDA. The experiment shows that our implementation is as fast as the implementation in CUBLAS, and the speed of our implementation can reach the peak speed's 97%, on Geforce GTX260.

Keywords: matrix multiplication; GPU; CUDA

GPU 是一种高性能的众核处理器，可以用来加速许多应用。CUDA 是 NVIDIA 公司为 NVIDIA 的 GPU 开发的一个并行计算架构和一门基于 C 的编程语言。在 CUDA 中程序可以直接操作数据而无需借助于图形系统的 API。现在已经有许多应用和典型算法使用 CUDA 在 GPU 上实现出来。

1 引言

矩阵乘法是科学计算中的最基本的操作，在许多领域中有广泛的应用。对于矩阵乘法的研究有几个方向。一个是研究矩阵乘法的计算复杂度，研究矩阵乘法的时间复杂度的下界，这方面的工作有 strassen 算法[1]等。另外一个方向是根据不同的处理器体系结构，将经典的矩阵乘法高效的实现出来，这方面的结果体现在许多高效的 BLAS 库。许多高效的 BLAS 库都根据体系结构的特点高效的实现了矩阵乘法，比如 GotoBLAS[2], ATLAS[3]等。Fatahalian[4]等人使用着色语言设计了在 GPU 上的矩阵乘法。CUBLAS 库是使用 CUDA 实现的 BLAS 库，里面包含了高性能的矩阵乘法。

本文剩下的部分组织如下，第 2 节介绍了 CUDA 的编程模型，简单描述了 CUDA 上编程的特点。第 3 节讨论了数据已经拷贝到显存上的矩阵乘法，首先根据矩阵分块的公式给出了一个朴素的矩阵乘法实现，分析朴素的矩阵乘法的资源利用情况，然后提出了一种新的高效的矩阵乘法。第 4 节讨论了大规模的矩阵乘法的设计和实现，着重讨论了数据在显存中的调度。第 5 节是实验结果。第 6 节是总结和展望。

2 CUDA编程模型和矩阵乘法回顾

2.1 CUDA 编程模型

NVIDIA 的 GPU 是由 N 个多核处理器和一块显存构成的。每个多核处理器由 M 个处理器核，1 个指令部件，一个非常大的寄存器堆，一小块片上的共享内

① 基金项目:国家自然科学基金(60833004);国家高技术研究发展计划(863)(2008AA010902)
收稿时间:2010-04-26;收到修改稿时间:2010-05-21

存以及一些只读的缓存(包括纹理缓存和常量缓存)。多处理器内所有核共用一个指令部件，在一个时钟周期内，每个处理器核执行同一条指令，但是数据可以是不一样的。

GT200 是 NVIDIA 的第 10 代 GPU，它发布于 2008 年 6 月。GT200 的多核处理器包含 8 个处理器核，16K 字节的片上共享存储器，16384 个 32 位的寄存器，1 个指令部件，1 个特殊功能单元部件，1 个双精度部件以及一些只读的缓存。GT200 是第一代支持双精度计算的显卡。

在 CUDA 中的核心概念是内核函数。内核函数是从 CPU 端调用，运行在 GPU 上的函数。CUDA 的运行管理系统产生一个 grid 来运行一个内核函数。1 个 grid 包含一组 block，每个 block 是有一组线程组成。grid 维度和 block 的维度都是在调用内核函数的时候指定的。在一个 grid 中，block 之间没有同步和通信机制，因此 block 之间必须是独立的。在一个 block 里面，线程之间可以用同步原语_syncthreads()来同步，每个 block 可以分配一定量的所有线程都可以访问的共享内存。一个 block 只能映射到一个多核处理器上执行，而不能将线程分布在不同的处理器中。一个多核处理器上可以同时运行多个 block。在运行一个 block 的线程时，CUDA 运行管理系统将 block 中的线程分成一个个 warp，每次取出一个 warp 来运行。一个 warp 分成前半个 warp 和后半个 warp。运行的最小单位是半个 warp，一个周期里面，半个 warp 的所有线程执行相同的指令。在 CUDA 2.3 的版本中，一个 warp 中含有 32 个线程。

一个内核函数可以使用的存储器资源包括全局内存，共享内存，纹理内存，常量内存以及寄存器。全局内存，纹理内存以及常量内存都是分配在显存中的，可以被所有的 block 访问。纹理内存和常量内存都是只读的，如果重复访问的话，它们会被缓存在片上的缓存上。全局内存是不会被缓存的。GPU 访问显存是使用内存事务来完成的。完成一个内存事务的代价是非常高的，通常需要 400-600 个周期。但是，如果半个 warp 的所有线程的访问满足：(1)所有线程访问 1 个字节，并且访问的位置在一个 32 字节对齐的同一个 32 字节的段里面。(2)所有的线程访问 2 个字节，并且访问的位置在一个 64 字节对齐的同一个 64 字节的段里面。(3)所有的线程访问 4 个字节，并且访问的位置在一个 128 字节对齐的同一个 128 字节的段里面。三个条件中的一个，则半个 warp 的访存会被合并成一个内存事务。

2.2 矩阵乘法回顾

给定矩阵乘法 C=AB，其中 $C=(c)_{ij}$ 是 $M\times N$ 的矩阵，$A=(a)_{ij}$ 是 $M\times K$ 的矩阵，$B=(b)_{ij}$ 是 $K\times N$ 的矩阵。则 c_{ij} 的定义为

$$c_{ij}=\sum_{k=1}^{K}a_{ik}b_{kj} \tag{1}$$

假定 $M=wp,N=hq,K=kr$，将矩阵 C 分解为 $p\times q$ 的分块矩阵 $(c)_{ij}$，每个 c_{ij} 是一个 $w\times h$ 的小矩阵，A 分解成为 $p\times k$ 的分块矩阵 $(a)_{ij}$，每个 a_{ij} 是一个 $w\times r$ 的矩阵，B 分解成为 $k\times q$ 的分块矩阵 $(b)_{ij}$，每个 b_{ij} 是一个 $r\times h$ 的元素。则分块矩阵乘法的定义为

$$c_{ij}=\sum_{s=1}^{k}a_{is}b_{sj} \tag{2}$$

在等式中核心的操作是

$$c_{ij}=a_{is}b_{sj}+c_{ij} \tag{3}$$

假定缓存中同时可以放下分块后的 c_{ij}，a_{is}，b_{sj}，则等式需要的访存数量是 $wh+wK+Kh$，计算量是 $2whK$，平均一次访存的计算量是 $2whK/(wh+wK+Kh)$，略去分母中的小项 wh 得到 $2/(1/w+1/h)$，从而知道 $1/w+1/h$ 越小，平均一次访存做的计算量是越多的。

3 算法设计和实现

本节假定矩阵 A，B，C 已经存放在显卡的显存中，矩阵是以行主序的方式存储的。假定 $M=wp,N=hq,K=kr$，A，B，C 分别是 $M\times K$，$K\times N$，$M\times N$ 的矩阵。

3.1 普通实现

按照等式，每个 c_{ij} 的计算是相互独立的。将矩阵 C 分成许多 $w\times h$ 的子矩阵，每个子矩阵使用不同的 block 来计算。假定一个 block 的线程数目是 T，这样每个线程需要计算 C 中 wh/T 个元素。将 C 中的元素固定分配给某个线程，这样可以将 C 中的元素存储在每个线程的寄存器中。为了充分利用 GPU 上的带宽，将矩阵 A 和 B 读取进来的数据先缓存在片上的共享存储器中。每次读取 A 的一个 $w\times r$ 的子矩阵和 B 的一个 $r\times h$ 的子矩阵到共享存储里，在共享存储器中完成一个小块的矩阵乘法。这样得到的算法如图 1 所示。

下面讨论算法的参数。为了提高访问全局内存的效率，r 和 h 必须是 16 的倍数。同时，一个 block 需要的共享内存是 $8(wr+rh)$ 字节。每个 block 需要的共享内存的大小必须小于一个 SM 上的共享存储器的大小，从而得到 $8(wr+rh)<16384$，同时所有线程需要的寄存器数目要小于一个 SM 上的寄存器数目。每个双精度的数需要占用 2 个 32 位的寄存器，w 和 h 必须满足不等式 $2wh\le 16384$。为了让 w 和 h 足够大，我们取 $r=16$，这样有 $w+h<128$。根据矩阵分块乘法，$1/w+1/h$ 越小越好，而 h 必须是 16 的倍数，在算法中取 $w=32$，$h=64$。于是有 $wh=2048$。而当前一个 block 中的线程数目最多是 512，于是可以取 T=512，每个线程计算 4 个数据。使用 cuda 编译器编译以后，每个线程需要使用 24 个寄存器，一个 block 总共需要 12288 个寄存器，需要 12728 个共享存储器。

```
在共享存储器中分配 32×16 的数组 As
在共享存储器中分配 16×64 的数组 Bs
根据 block 的 id 和线程的 id 计算指针 A，B 和 C 的指针位置
do
   读取 A 中 32×16 的子矩阵到 As
   读取 B 中 16×64 的子矩阵到 Bs
   block 内同步
   计算 Cs+=AsBs
   更新指针 A 和指针 B
until 指针 B 越界
将 Cs 中的值写入 C 中的位置
```

图 1　矩阵乘法的普通实现

3.2 优化实现

在 3.1 节的算法中，矩阵 A 和 B 的子块都是保存在共享存储器中，而 C 的子块是保存在寄存器中。主要是根据共享存储器的限制来得到 w 和 h 的值，寄存器的使用数目远远没有达到处理器的上限，处理器的寄存器资源没有被充分利用。为了充分利用处理器上的寄存器资源，我们改变数据映射方式。仍然假设每个 block 计算 $w\times h$ 的一个子矩阵 Cs，使用一个线程来计算 Cs 中的一列，也就是要计算 w 个元素。注意到一个线程只会访问矩阵 B 的一列，也就是要计算 w 个元素。注意到一个线程只会访问矩阵 B 的一列，不同的线程访问 B 的元素是不相交的。这样 B 的元素可以放在线程中的寄存器中，而不用存放在共享存储器中。在共享存储器中分配一个 $w\times r$ 的数组 As 存放矩阵 A 的子块。这样得到的算法如图 2 所示。

```
在共享存储器中分配 w×r 的数组 As
根据 block 的 id 和线程的 id 计算指针 A，B 和 C 的指针位置
do
   读取 A 中 w×r 的子矩阵到 As
   更新指针 A
   block 内同步
   for i=1 to r do
      读取 B 中一行 Bs[1:h]
      更新 B 的指针
      Cs+=As[:i]Bs[1:h]
   until 指针 B 越界
将 Cs 中的值写入 C 中的位置
```

图 2　矩阵乘法的优化实现

下面讨论算法的参数。主要考虑的是两个条件，一个是充分利用片上的存储资源，包括寄存器和共享存储器；另外一个是要有足够的线程在运行。一个 block 需要的共享存储器是保存 A 的子矩阵的数组 As，大小是 $8wr$ 字节；需要的寄存器数目包括矩阵 B 和矩阵 C 的元素，需要 $2(wh+h)$ 个寄存器。为了节省共享内存，同时充分利用 GPU 上的带宽，取 r=16。另外一个方面，线程数目应该是 32 的倍数，为了保证足够的线程来计算，至少需要 2 个 warp 来计算，从而 $h\ge 64$。当 h=64 时，取 w=16，用 cuda 编译编译好了以后，每个线程需要 50 个寄存器，一个 block 需要的总寄存器数目是 3328，共享内存是 2248 字节。这样一个处理器的资源可以容纳的 block 数目为

$$\min\left(\frac{512}{64},\frac{16384}{3328},\frac{16384}{2248}\right)=4$$

4　大规模矩阵乘法的算法设计

在本节中，我们假定显卡的显存已经无法同时容

纳矩阵 A，B 和 C。显存只能存放部分数据，因此把矩阵 A，B 以及 C 分块。为了充分利用显卡的资源，可以考虑使用数据拷贝和计算进行重叠的技术。按照 2.2 节的分析，w 和 h 越大，在主机和显卡之间传送的数据就越小。将矩阵 C 分成 $w\times h$ 的子块，在显存中分配 5 块内存的空间：C_0,A_0,A_1,B_0,B_1。其中 C_0 用来存放 $w\times h$ 的矩阵，A_0,A_1 用来存放 $w\times r$ 的矩阵，而 B_0,B_1 用来存放 $r\times h$ 的矩阵。将 A，B 和 C 分成形状 $w\times r$，$r\times h$，$w\times h$ 的子块。假定计算 C 中一个子块需要执行等式 K 次。这样得到的算法如图 3 所示。

for C 中的每个 $w\times h$ 的子块 C_{sub}， do
  将 C_{sub} 拷贝到 C_0 中
  将 a_{i0} 和 b_{0j} 拷贝到 A_0， B_0 中
  $p=1$
  for $q=0$ to $K-2$ do
    $C_0=A_{1-P}B_{1-P}+C_0$
    将 $a_{i(q+1)}$ 和 $b_{(q+1)j}$ 拷贝到 A_p， B_p 中
    $p=1-p$
  done
  $C_0=A_{1-P}B_{1-P}+C_0$
  将 C_0 拷贝回 C_{sub} 中
done

图 3 大规模矩阵乘法的算法实现

5 实验结果

实验平台是 Intel core 2 duo E8400，4GB RAM，Geforce GTX 260，操作系统是 32 位的 Ubuntu 9.04。CUDA 工具包的版本是 2.3。

首先考虑小规模的矩阵乘法，所有数据都已经拷贝到显存上，没有考虑主机和显卡的通信。我们记 3.1 节的算法为 normal，3.2 节的算法为 opt，CUDA 工具包中 CUBLAS 库的矩阵乘法记做 cublas。

从图 4 中可以看出，随着输入的规模的增大，三个实现的速度都是慢慢上升的。当输入规模大于 2000 时，三个算法的速度达到了峰值，且基本保持不变。normal 的峰值是 60GFLOPS，而 opt 和 cublas 的峰值是 65GFLOPS。而 GTX 260 的双精度峰值是 67.068GFLOPS，从而 opt 的效率达到了 97%。

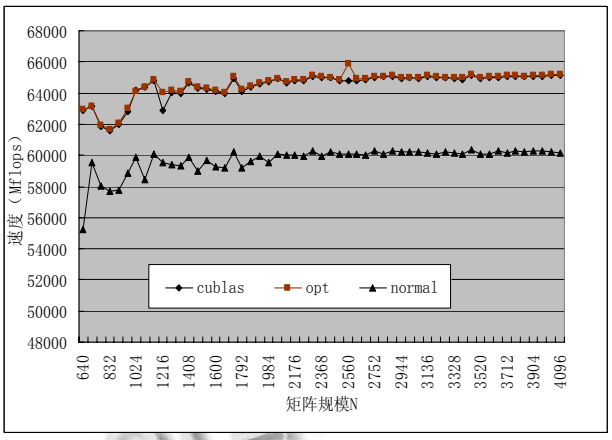

图 4 normal，opt 以及 cublas 的速度比较

我们分析 opt 的性能，在小规模输入的情况下，由于数据不多，从显存读取数据到 GPU 核心的时间没有被覆盖，所以速度比较慢。而当输入数据规模超过 2000 以后，计算量比较大，从而可以掩盖大部分的访问时间。但是仍然有少量的时间没有被计算重叠，这是由于片上的资源支持的 block 数目太少，没有足够的线程切换来掩盖访存的时间。

其次考虑数据没有拷贝到显存中的情况。由于测试平台的限制，本文选取的测试矩阵的情况是，A 的规模是，B 的规模是，C 的规模是。对 C 的分块大小是，对 A 和 B 的分块大小分别为和，测试结果如图 5 所示。在图 5 中，dgemm_kernel_NN_10 是计算的内核，而 memcpyDtoHAsync 和 memcpyHtoDAsync 分别是从设备到主机和从主机到设备进行数据拷贝的原语。Stream_3 将 C 矩阵从主机端拷贝到设备端，Stream_2 将 A 矩阵和 B 矩阵从主机端拷贝到设备端。Stream_1 进行矩阵乘的计算。Stream_4 将计算好的 C 矩阵从设备端拷贝到主机端。从图 5 可以看出，GPU 计算使用的时间占到 90%以上，而数据拷贝使用的时间不到 10%。

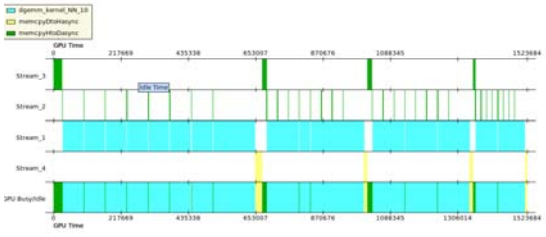

图 5 GPU 的运行时序图

(下转第 149 页)

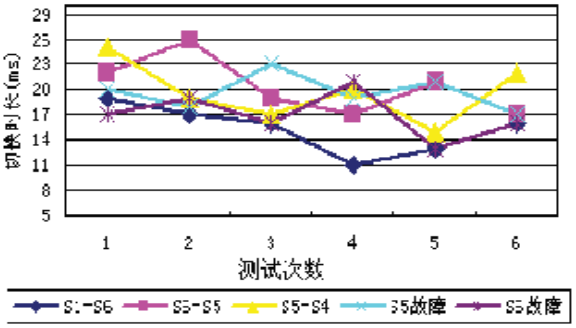

图 5 不同故障条件下 EAPS 切换时长

4 结束语

文中针对校园网用户对网络可靠性的电信级要求的现状，将目前校园网中所采用生成树协议来解决可靠性的方法与 EAPS 技术作了对比，描述了 EAPS 的工作机制。结合我校设备情况，引入利用 EAPS 技术来解决校园网的可靠性，对我校网络拓朴进行了重设置。事实证明，无论是设备宕机还是线路意外断开，环路的切换时间均能满足电信级要求，能保障网络业务的正常开展。进一步工作是关于 EAPS 多环在大规模校园网、大型园区网中的应用及可靠性研究。

参考文献

1 胡中栋.图书馆网络的安全可靠性分析.信息安全,2007(3):57－59.
2 庞雄琦.ESR 的研究与实现.光通信技术,2005(11):39－41.
3 田晓宏,赵方.城域以太网发展综述.光通信技术,2008 (6):28－30.
4 何强,张祖平.基于 MSTP 协议的一致性测试设计.计算机应用,2007(27):27－29.
5 郭彦伟,郑建德.生成树协议与交换网络环路研究.厦门大学学报(自然科学版),2006(5):301－304.
6 Etrame Network Ethernet automatic protection switching version1. 2003.

(上接第 181 页)

6 结论

本文在 GPU 上设计和实现了一个高效的矩阵乘法。在 Geforce GTX 260 上，在数据已经拷贝到显存的前提下，双精度的实现的效率达到了 97%。如果数据不能够一次全部的拷贝到显存中，通过数据调度，本文的算法拷贝数据的时间不超过总时间的 10%。

参考文献

1 Volker Strassen. Gaussian Elimination is not Optimal. Numerische Mathe-matik, 1969,13(4):354－356.
2 Goto K. Anatomy of High-Performance Matrix Multiplication. ACM Trans. on Mathematical Software, 2007,34(3):1－24.
3 Whaley RC, Petitet A and Dongarra J. Automated Empirical Optimization of Software and the Atlas Project. Parallel Computing, 2001,27(1-2):3－35.
4 Fatahalian K, Sugerman J, Hanrahan P. Understanding the Efficiency of GPU Algorithms for Matrix-matrix Multiplication. Proc. of the ACM SIGGRAPH/ EUROGRAPHICS Conference on Graphics Hardware (HWWS'04). New York: ACM Press, 2004. 133－137.