风辰的 CUDA 入门教程

作者：风辰
二零一零年七月二十四日
于中国科学院研究生院青年公寓

目录

第一章、CUDA 的基本内容

第一节、CUDA 及 GPU 简介

GPU 是图形处理单元（Graphic Processing Unit）的简称，最初主要用于图形渲染。自九十年代开始，GPU 的发展产生了较大的变化，NVIDIA、AMD(ATI)等 GPU 生产商敏锐的观察到 GPU 天生的并行性，经过他们对硬件和软件的改进，GPU 的可编程能力不断提高，GPU 通用计算应运而生。由于 GPU 具有比 CPU 强大的计算能力(见图 1-1），为科学计算的应用提供了新的选择。

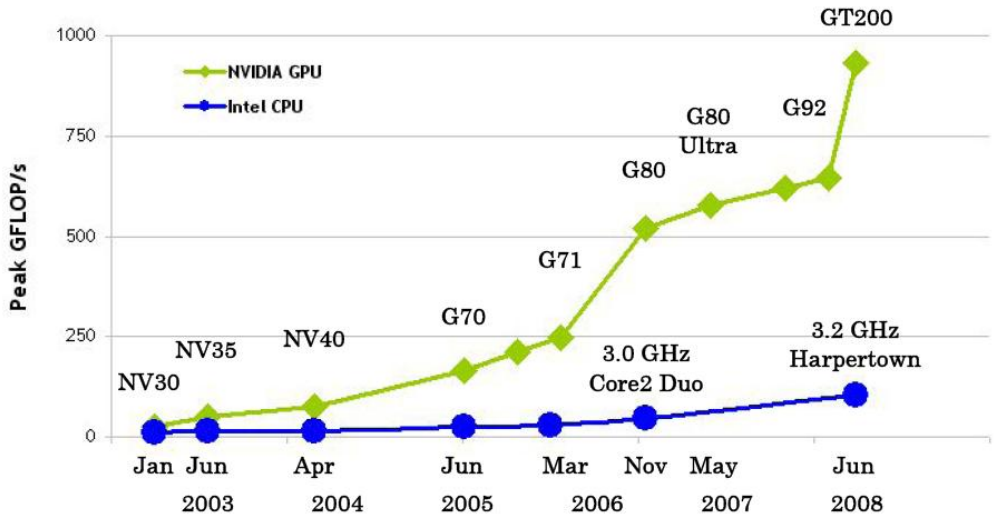

图 1-1 GPU 与 CPU 峰值计算能力对比图

由于 GPU 拥有比 CPU 更强的计算能力，很早就有人想到将 GPU 应用到通用计算上，这就是 GPGPU，所谓 GPGPU 是指直接使用了图形学的 API，将任务映射成纹理的渲染过程，使用汇编或者高级着色器语言 Cg，HLSL 等编写程序，然后通过图形学 API 执行（Direct3D 和 OpenGL），这样的开发不仅难度较大，而且难以优化，对开发人员的要求非常高，因此，传统的 GPGPU 计算并没有广泛应用。

2007 年 6 月，NVIDIA 公司推出了 CUDA（Compute Unified Device Architecture），CUDA 不需要借助图形学 API，而是采用了类 C 语言进行开发。同时，CUDA 采用了统一处理架构，降低了编程的难度，使得 NVIDIA 相比 AMD/ATI 后来居上。相比 AMD 的 GPU，NVIDIA GPU 引入了片内共享存储器，提高了效率。这两项改进使 CUDA 架构更加适合进行 GPU 通用计算。由于这些特性，CUDA 推出后迅速发展，被应用于石油勘测、天文计算、流体力学模拟、分子动力学仿真、生物计算、图像处理、音视频编解码等领域。

CUDA 直接采用 C/C++编译器作为前端（如 Linux 下的 gcc,windows 下的 vs），以 C/C++语法为基础设计，因此对熟悉 C 系列语言的程序员来说，CUDA 的语法比较容易掌握。但是这并不意味着 CUDA 容易，整体来说 CUDA 是相当难的，难在优化，难在开发出健壮、可扩展性的程序；难在没有成熟的库和算法以借用。从语言的角度说，CUDA 只对 ANSI C 进行了最小的必要扩展，以实现其关键特性--线程按照两个层次进行组织、共享存储器和栅栏同步。这些关键特性使得 CUDA 拥有了两个层次的并行：线程级并行实现的细粒度数据并行，和任务级并行实现的粗粒度并行。

第二节、Linux 下 CUDA 开发环境安装

前一节已经简单了说了一下 CUDA,为了能够使用 CUDA 开发，这一节将说明怎样构建 CUDA 开发环境。

本节介绍在 ubuntu9.04 操作系统和 gcc 前端的基础上安装 CUDA 开发环境。

首先，要保证自己机器上的 gcc 能够使用，因为 Ubuntu 缺少 gcc 的一些包和 g++，所以这些得自己安装。安装命令:sudo apt-get install g++，待其完成后，弄个 C 代码试试看:);当然你得保证你的显卡支持 CUDA。

其次，到 nVidia 官方网站(http://www.nvidia.cn/object/cuda_get_cn.html)上下载对应操作系统的驱动(driver)和工具包(toolkit)。

再次，转换到控制台，命令为 Ctrl+Alt+F1/F2/F3/F4，关掉 gdm，命令为：sudo /etc/init.d/gdm stop，要确定已经关闭，否则在安装时会提示你有 x server 程序在运行。

再次，进入 driver 和 toolkit 目录，执行安装命令,为了方便，请一定按照默认安装。命令 sudo sh ./dev……

sudo sh ./nv……

然后，打开个人目录下的.bashrc 文件或者/etc/profile 文件，在其中加入命令：PATH=${PATH}:/usr/local/cuda/bin/
LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/usr/local/cuda/lib，执行 source .bashrc 或者/etc/profile，依据你添加 PATH 和 LD_LIBRARY_PATH 时修改了那个文件确定。

最后执行 nvcc 命令，看看，如果提示你没有输入文件，就安装完成了。

如果你要安装 SDK 的话，还得安装一些包，依据 make 时的提示，Google 和新力得应该可以搞定一切，现在你可以享受 CUDA 了！

第三节、CUDA 与 fork/join 模式

说到 CUDA 就不能不说一下异构并行计算，所谓异构就是指多个计算平台，如 CPU,GPU 和其它一些加速部件。一般而言，各个计算平台有其独立的硬件资源。CPU 和 GPU 拥有各自独立的 ALU、存储器。本文中的内存特指 CPU 内存，而涉及到 GPU 的存储器，则用全称。

CUDA 认为可以用于计算的硬件系统包含两个部分，一个是 CPU,一个是 GPU,CPU 控制/指挥 GPU 工作，GPU 只是 CPU 的协处理器。

熟悉 Linux、java、pthread 或 OpenMP 的人对于 fork/join 一定不会陌生，下图给出了 fork/join 模式的示意图。在 Linux 中，fork 函数会产生子进程，子进程和父进程共同工作，父进程调用 wait/waitpid 函数来等待子进程；在 pthread 中，pthread_create 函数产生子线程，子线程执行指定的工作，在某个位置，父线程调用 pthread_join 来等待子线程完成工作；java Thread 中有一个函数 join 也是等待某个运行至此，在 java 7 中，则直接引入 fork/join 框架；当然，这些语文/库相应的函数功能都远超过这里所说的。在这一点上和 CUDA 最相似的是 OpenMP,其#pragma omp parallel 指令表示下面的一个代码块是由多个线程执行的，到块的结束，所有的线程又都自动消失，只留下主线程。CUDA 也是这样，CUDA 中多线程执行的块称为内核，内核有其特殊的声明和调用函数，这在下一节说明。

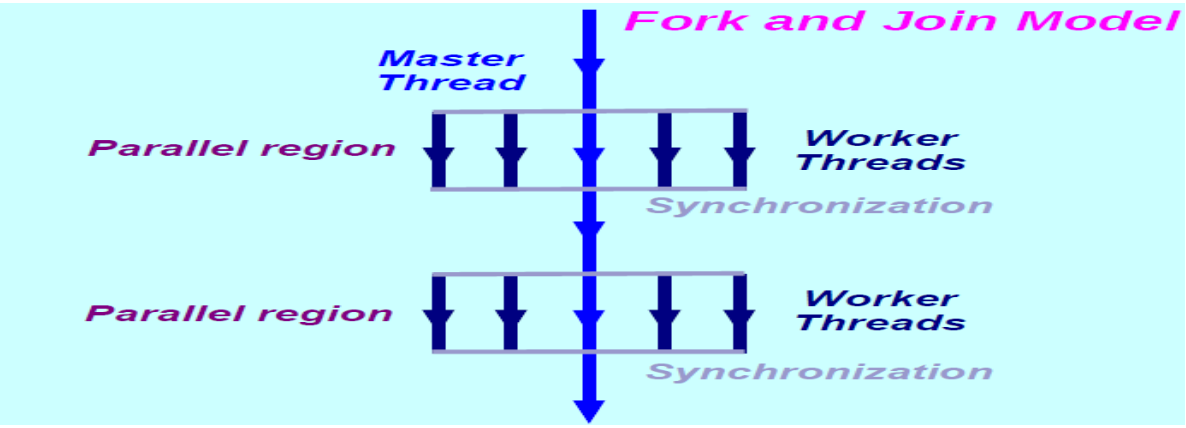

图 1-3.fork/join 模式

第四节、CUDA C 语言

对一个语言来说，最简单的，也是用得最多的当然是它的语法了，下面简单的介绍一下 CUDA 的语法。

CUDA C 不是 C 语言，而是对 C 语言进行扩展。CUDA 对 C 的扩展主要包括以下四个方面：

- 函数类型限定符，用来确定函数是在 CPU 还是在 GPU 上执行，以及这个函数是从 CPU 调用还是从 GPU 调用。
  - __device__，表示从 GPU 上调用，在 GPU 上执行；也就是说其可以被__global__ 或者__device__修饰的函数调用。此限定符修饰的函数使用有限制，比如在 G80/GT200 架构上不能使用递归，不能使用函数指针等，具体可参见我翻译的编程指南。
  - __global__，表示在 CPU 上调用，在 GPU 上执行，也就是所谓的内核(kernel)函数；内核只能够被主机调用，内核并不是一个完整的程序，它只是一个数据并行步骤，其指令流由多个线程执行。
  - __host__，表示在 CPU 上调用，在 CPU 上执行，这是默认时的情况，也就是传统的 C 函数。CUDA 支持__host__和__device__的联用，表示同时为主机和设备编译。

- 变量类型限定符，用来规定变量存储什么位置上。在传统的 CPU 程序上，这个任务由编译器承担。在 CUDA 中，不仅要使用主机端的内存，还要使用设备端的显存和 GPU 片上的寄存器、共享存储器和缓存。在 CUDA 存储器模型中，一共抽象出来了 8 种不同的存储器。复杂的存储器模型使得必须要使用限定符要说明变量的存储位置。
  - __device__，__device__表明声明的数据存放在显存中，所有的线程都可以访问，而且主机也可以通过运行时库访问；
  - __shared__，__shared__表示数据存放在共享存储器在，只有在所在的块内的线程可以访问，其它块内的线程不能访问；
  - __constant__，__constant__表明数据存放在常量存储器中，可以被所有的线程访问，也可以被主机通过运行时库访问；
  - Texture，texture 表明其绑定的数据可以被纹理缓存加速存取，其实数据本身的存放位置并没有改变，纹理是来源于图形学的一介概念，CUDA 使用它的原因一部分在于支持图形处理，另一方面也可以利用它的一些特殊功能。
  - 如果在 GPU 上执行的函数内部的变量没有限定符，那表示它存放在寄存器或者本地存储器中，在寄存器中的数据只归线程所有，其它线程不可见。
  - 如果 SM 的寄存器用完，那么编译器就会将本应放到寄存器中的变量放到本地存储器中。
- 执行配置运算符<<< >>>，用来传递内核函数的执行参数。执行配置有四个参数，第一个参数声明网格的大小，第二个参数声明块的大小，第三个参数声明动态分配的共享存储器大小，默认为 0，最后一个参数声明执行的流，默认为 0。
- 五个内建变量，用于在运行时获得网格和块的尺寸及线程索引等信息
  - gridDim, gridDim 是一个包含三个元素 x,y,z 的结构体，分别表示网格在 x,y,z 三个方向上的尺寸，虽然其有三维，但是目前只能使用二维；
  - blockDim, blockDim 也是一个包含三个元素 x,y,z 的结构体，分别表示块在

x,y,z 三个方向上的尺寸，对应于执行配置中的第一个参数，对应于执行配置的第二个参数；

- blockIdx, blockIdx 也是一个包含三个元素 x,y,z 的结构体，分别表示当前线程所在块在网格中 x,y,z 三个方向上的索引；
- threadIdx, threadIdx 也是一个包含三个元素 x,y,z 的结构体，分别表示当前线程在其所在块中 x,y,z 三个方向上的索引；
- warpSize，warpSize 表明 warp 的尺寸，在计算能力为 1.0 的设备中，这个值是 24，在 1.0 以上的设备中，这个值是 32。

其它的还有数学函数，原子函数，纹理读取、绑定函数，内建栅栏，内存 fence 函数等。一般而言，知道这些就应该能够写出 CUDA 程序了，当然要写好的话，必须知道很多其它的细节。

第五节、计算π

上一节，已经简单的说了一下 CUDA C 的基本语法；因而在本节，兄弟决定以一个例子为基础说明 CUDA 程序的基本组成部分，不过说实话兄弟选择的例子并不太好，这个例子就是采用积分法计算圆周率/π的值。其计算原理是：在[0,1]范围内积分 1/(1+x*x)

首先，让我们看一下在 CPU 上的计算流程，其计算流程如下

```
/*
串行计算 PI 的程序，基本思想为：将积分区间均分为 num 小块，将每小块的面积加起来。
*/
    float cpuPI(int num){
        float sum=0.0f;
        float temp;
        for(int i=0;i<num;i++){
            temp=(i+0.5f)/num;
        //  printf ("%f\n", temp);
            sum+=4/(1+temp*temp);
        //  printf ("%f\n", sum);
        }

        return sum/num;
    }
```

很明显，我们可以将 for 循环分解，使用 CUDA 处理。

有一个问题就是：for 内部对 sum 变量的更新是互斥的，而 CUDA 中并没有浮点原子函数，对于这个问题的解决方案是：将 for 循环内部的两个语句拆开，分成两个内核函数来做运算。第一个内核计算计算每个小积分块面积，并将每个 block 内所有线程对应的积分块面积加起来，存入全局存储器；第二个内核将前一个内核存入的数据加起来。下面是 kernel 代码：

```
/*
在 GPU 上计算 PI 的程序，要求块数和块内线程数都是 2 的幂
前一部分为计算 block 内归约，最后大小为块数
```

```
后一部分为单个 block 归约，最后存储到*pi 中。
*/
/*
在 GPU 上计算 PI 的程序，要求块数和块内线程数都是 2 的幂
前一部分为计算 block 内归约，最后大小为块数
后一部分为单个 block 归约，最后存储到*pi 中。
*/
    __global__ void reducePI1 (float *d_sum, int num) {
    int id=blockIdx.x*blockDim.x+threadIdx.x;//线程索引
    int gid=id;
    float temp;
    extern float __shared__ s_pi[];//动态分配，长度为 block 线程数
    s_pi[threadIdx.x]=0.0f;

    while(gid<num){
        temp=(gid+0.5f)/num;//当前 x 值
        s_pi[threadIdx.x]+=4.0f/(1+temp*temp);
        gid+=blockDim.x*gridDim.x;
    }

    for(int i=(blockDim.x>>1);i>0;i>>=1){
        if(threadIdx.x<i){
            s_pi[threadIdx.x]+=s_pi[threadIdx.x+i];
        }
        __syncthreads();
    }
    if(threadIdx.x==0)
    d_sum[blockIdx.x]=s_pi[0];
    }

    __global__ void reducePI2(float *d_sum,int num,float *d_pi){
    int id=threadIdx.x;
    extern float __shared__ s_sum[];
    s_sum[id]=d_sum[id];
    __syncthreads();
    for(int i=(blockDim.x>>1);i>0;i>>=1){
        if(id<i)
        s_sum[id]+=s_sum[id+i];
        __syncthreads();
    }
//  printf("%d,%f\n",id,s_sum[id]);
    if(id==0){
    *d_pi=s_sum[0]/num;
//  printf("%d,%f\n",id,*pi);
    }
    }
```

其中__syncthreads()是 CUDA 的内置命令，其作用是保证 block 内的所有线程都已经运行到调用__syncthreads()的位置，这样可以保证各个线程看到的存储器是一样的。

由上面的代码可以看出，使用 CUDA 要写的代码要比串行 C 的多，主要原因在于要避开 GPU 的弱点，或者说相比 CPU,GPU 的可编程性还是不太强。从中亦可以看出，并行化串行程序的主要阻碍在于数据相关性。

一般而言，CUDA 程序的基本模式是：

一、 分配内存空间和显存空间

二、 初始化内存空间

三、 将要计算的数据从内存上复制到显存上

四、 执行 kernel 计算

五、 将计算后显存上的数据复制到内存上

六、 处理复制到内存上的数据

这个程序使用了归约算法，从某种程序上说，我的算法不是最高效的。在我的机器(CPU 2.0GHZ,GPU GTX295)上此程序的加速比超过 100，不知道在你们的机器上能够加速多少？

第六节、编程模式

CUDA 支持大量的线程并行（Thread Level Parallel），并在硬件中动态地创建、调度和执行这些线程，在 CPU 中，这些操作是重量级的，但是在 CUDA 中，这些操作是轻量级的，我们可以忽略线程的创建和调度开销。CUDA 编程模型将 CPU 作为主机（Host），而将 GPU 作为协处理器（Coprocessor），或者设备（Device），以 CPU 来控制程序整体的串行逻辑和任务调度，而让 GPU 来运行一些能够被高度线程化的数据并行部分。即让 GPU 与 CPU 协同工作，更确切的说是 CPU 控制 GPU 工作。GPU 只有在计算高度数据并行任务时才发挥作用。

能够使用 GPU 计算的程序必须具有以下特点：需要处理的数据量比较大，数据以数组或矩阵形式有序存储，并且对这些数据要进行的处理方式基本相同，各个数据之间的依赖性或者说耦合很小，需要复杂数据结构的计算如树，图等，则不适用于使用 GPU 进行计算。找到程序中满足这些要求的部分后，就能将该部分程序移植 GPU 上。运行在 GPU 上的程序被称为内核（Kernel）。内核并不是完整的程序，只是整个程序中的一个可以使用数据并行处理的步骤。一个完整的程序由若干个内核函数以及 CPU 上的串行处理共同组成。一个完整的程序的计算流程如下所示：

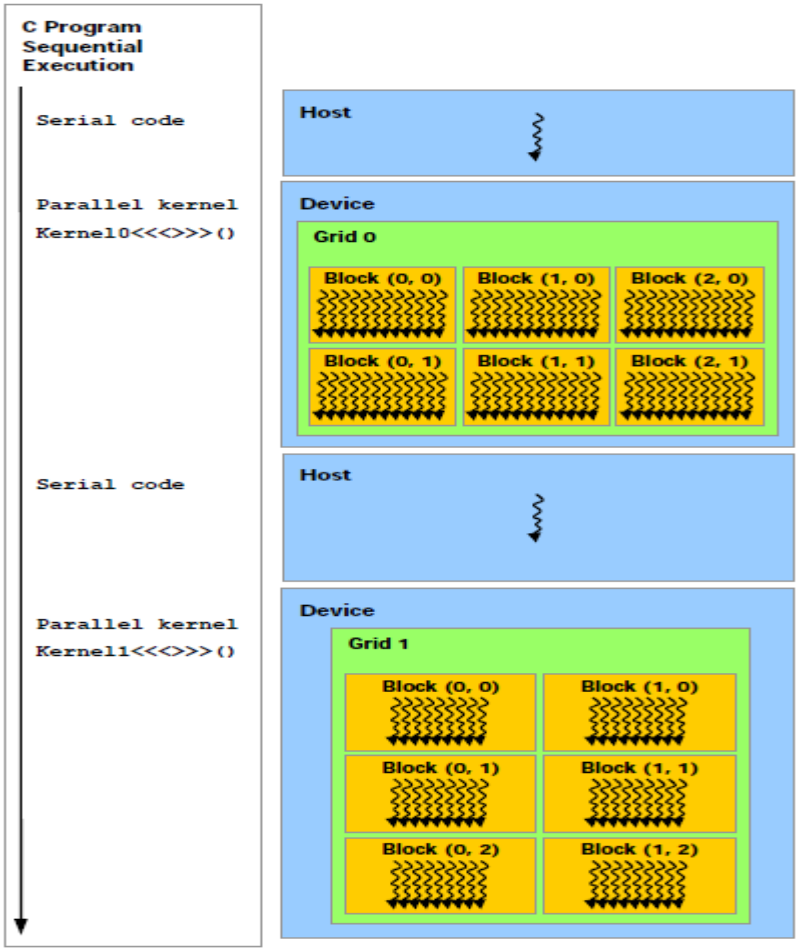

图 1-6 CUDA 程序流程

除了数据并行的内核程序以外，程序中也有标准的串行程序。如上图所示，在两个核函数之间运行的就是串行代码。理想情况下，串行代码的作用应该只是清理上个内核函数，并启动下一个内核函数，但由于目前的 GPU 的功能仍然十分有限，串行部分的工作量仍然十分可观。

第七节、线程层次

GPU 线程以网格（grid）的方式组织，而每个网格中又包含若干个线程块，在 G80/GT200 系列中，每一个线程块最多可包含 512/1024 个线程，Fermi 架构中每个线程块支持高达 1536 个线程。同一线程块中的众多线程不仅能够并行执行，而且能够通过共享存储器（Shared memory）和栅栏（barrier）通信。这样，同一网格内的不同块之间存在不需要通信的粗粒度并行，而一个块内的线程之间又形成了允许通信的细粒度并行。这些就是 CUDA 的关键特性：线程按照粗粒度的线程块和细粒度的线程两个层次进行组织、在细粒度并行的层次通过共享存储器和栅栏同步实现通信，这就是 CUDA 的双层线程模型。

在执行时，GPU 的任务分配单元（global block scheduler）将网格分配到 GPU 芯片上。启动 CUDA 内核时，需要将网格信息从 CPU 传输到 GPU。任务分配单元根据这些信息将块分配到 SM 上。任务分配单元使用的是轮询策略：轮询查看 SM 是否还有足够的资源来执行新的块，如果有则给 SM 分配一个新的块，如果没有则查看下一个 SM。决定能否分配的因素有：每个块使用的共享存储器数量，每个块使用的寄存器数量，以及其它的一些限制条件。任务分配单元在 SM 的任务分配中保持平衡，但是程序员可以通过更改块内线程数，每个线程使用的寄存器数和共享存储器数来隐式的控制，从而保证 SM 之间的任务均衡。任务以这

种方式划分能够使程序获得了可扩展性：由于每个子问题都能在任意一个 SM 上运行，CUDA 程序在核心数量不同的处理器上都能正常运行，这样就隐藏了硬件 SM 数差异。

对于程序员来说，他们需要将任务划分为互不相干的粗粒度子问题(最好是易并行计算)，再将每个子问题划分为能够使用单个 GPU 线程处理的问题。

同一线程块中的线程开始于相同的指令地址，理论上能够以不同的分支执行。但实际上，在块内的分支因为 SM 构架的原因被大大限制了。内核函数实质上是以块为单位调度的。同一线程块中的线程需要 SM 中的共享存储器共享数据，因此它们必须在同一个 SM 中发射。线程块中的每一个线程被映射到一个 SP 上。

任务分配单元可以为每个 SM 分配最多 8 个块。而 SM 中的线程调度单元又将分配到的块进行细分，将其中的线程组织成更小的结构，称为线程束（warp）。在 CUDA 中，warp 对程序员来说是透明的，它的大小可能会随着硬件的发展发生变化，在当前版本的 CUDA 中，每个 warp 是由 32 个线程组成的。每个 SM 中的 8 个 SP 采用了发射一次指令，执行 4 次的流水线结构。所以由 32 个线程组成的 Warp 是 CUDA 程序执行的最小单位，并且同一个 warp 内是严格串行的，因此在 warp 内是无须同步的。SM 划分 warp 的方式是简单一致的，如第 0 warp 包括线程 0-31，第 1 warp 包含线程 32-63，以此类推。在一个 SM 中可能同时有来自不同块的 warp。当一个块中的 warp 在进行访存或者该块进行同步等高延迟操作时，另一个 warp 或块可以占用 SM 中的计算资源，这样，可以尽可能的让 SM 保持忙碌，提高了执行效率。不同块之间的执行没有顺序，完全并行，因此无论是在一次只能处理一个线程块的 GPU 上,还是在一次能处理数十乃至上百个线程块的 GPU 上，这一模型都能很好的适用。

目前，一个 SM 上，某一时刻只能有一个内核函数正在执行，但是在 Fermi 架构中，这一限制已被解除。通过流，Fermi 允许多个内核同时占用一个 SM 的计算资源，这样在一个内核访问数据时，另一个内核能够进行计算，可以有效的提高设备的利用率。

第八节、存储器组织

CUDA 的存储器模型如下图所示，由一系列不同的地址空间组成。其中，shared memory 和 register 位于 GPU 片内，Texture memory 和 Constant memory 可以由 GPU 片内缓存加速对片外显存的访问，而 Local memory 和 Device memory 位于 GPU 片外的显存中。

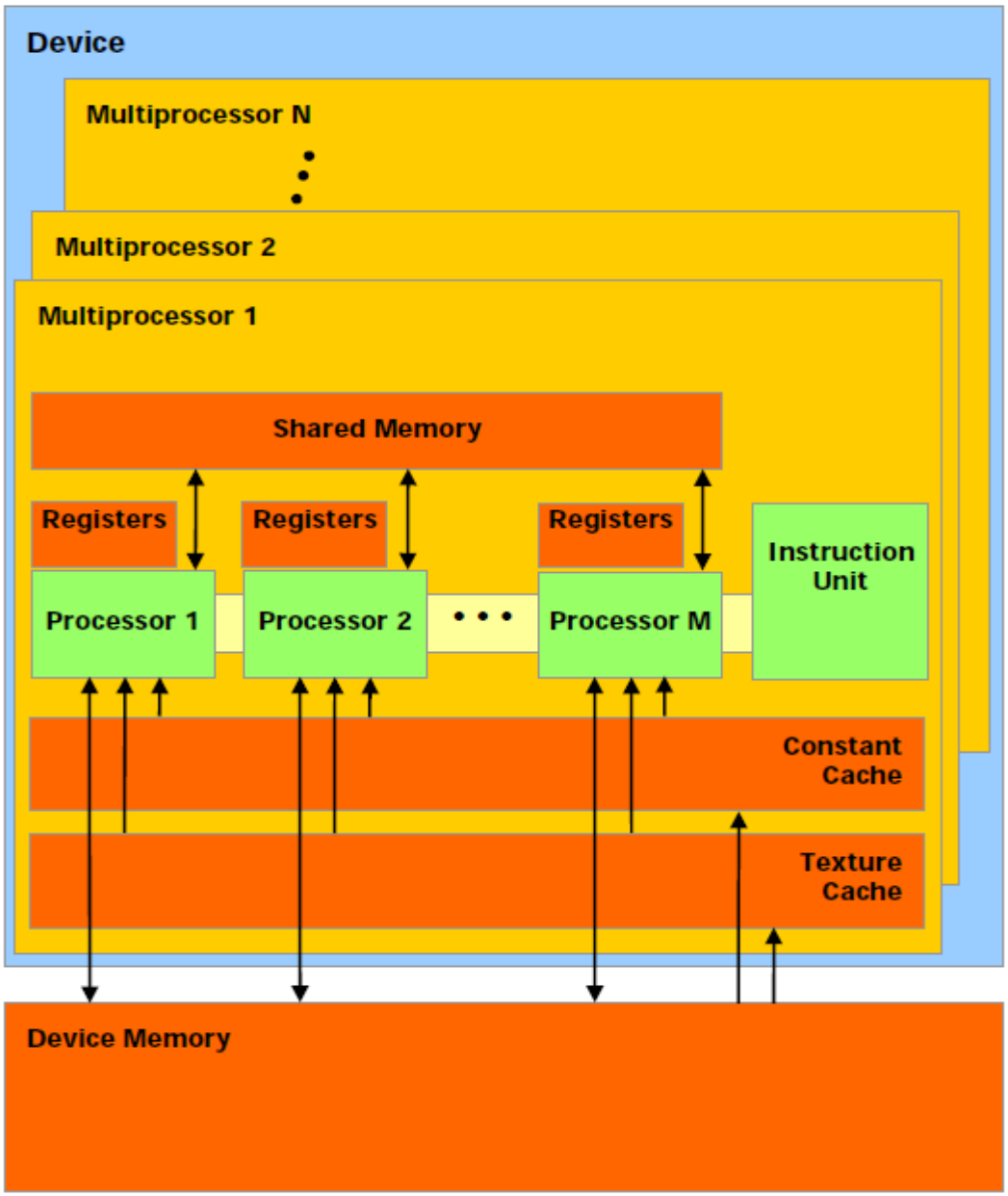

图 1-8-1　CUDA 存储器层次结构

最靠近流处理器的是寄存器文件（register file），每个寄存器文件是 32bit。对线程来说，寄存器都是私有的，不允许其它线程染指。由于更靠近流处理器，寄存器具有最快的速度，GT200 的每个 SM 拥有 64KB 的寄存器文件（Register Files），故一个块内最多可分配 16K 个寄存器，而 G80 中每个 SM 只有 32KB，故一个块最多可分配 8K 个寄存器，Fermi 每个ＳＭ有 32K 个寄存器。最新加入的 64bit 数据类型（双精度浮点和 64 位整数型）将占用两个相邻的寄存器单元。如果寄存器被消耗完，数据将被存储在本地存储器（local memory）。对每个线程来说，本地存储器也是私有的，但是本地存储器是显存中的一个分区，对于 G80/GT200,没有缓存，速度极慢，由于 Fermi 拥有一级和二级缓存，速度不会有太大的影响。内核寄存器的用量可以通过修改块大小，使用共享存储器等方法来增加或减少。

共享存储器是可以被同一块中的所有线程访问的可读写存储器，它的生存期就是块的生命期。在没有冲突的情况下，访问共享存储器只要一两个时钟，几乎与访问寄存器一样快，是实现线程间通信的最好方法。共享存储器可以实现许多不同的功能，如用于保存块共用的计数器或者块内的公用结果（例如 reduction）。在同一个块内，所有的线程都能够读写共享存储器中的数据，相比于 AMD 的显卡来说，共享存储器是 NVIDIA 显卡的一项特色。一般

而言，在 kernel 运行时，要先将数据从全局存储器写入共享存储器；计算完成后要将共享存储器中的结果转存入全局存储器。

Tesla 的每个 SM 拥有 16KB 共享存储器，用于同一个线程块内的线程间通信。为了使一个 half-warp 内的线程能够在一个内核周期中并行访问，共享存储器被组织成 16 个 bank，每个 bank 拥有 32bit 的宽度，故每个 bank 可保存 256 个整形或单精度浮点数，或者说目前的 bank 组织成了 256 行 16 列的矩阵。如果一个 half-warp 中有一部分线程访问属于同一 bank 的数据，则会产生 bank conflict，降低访存效率，在冲突最严重的情况下，速度会比全局显存还慢，但是如果 half-warp 的线程访问同一地址的时候，会产生一次广播，其速度反而没有下降。在不发生 bank conflict 时，访问共享存储器的速度与寄存器相同。在不同的块之间，共享存储器是毫不相关的。

在共享存储器方面，Fermi 采用了更为灵活的方式，Fermi 一共有 64KB 可配置为一级缓存和共享存储器的片上存储器，默认情况下，是 16KB 共享存储器，48KB 一级缓存，但是你也可以设置成 48KB 共享存储器和 16KB 一级缓存，具体的函数是 cudaFuncSetCacheConfig(xx)。Fermi 的 bank 数量不再是 16，而是 32，但是每次依旧最多只能读取 16 个。

下图说明了两种不会发生 bank conflict 的访问模式。Half warp 中的线程与 16 个 bank 一一对应，不会发生 bank conflict。

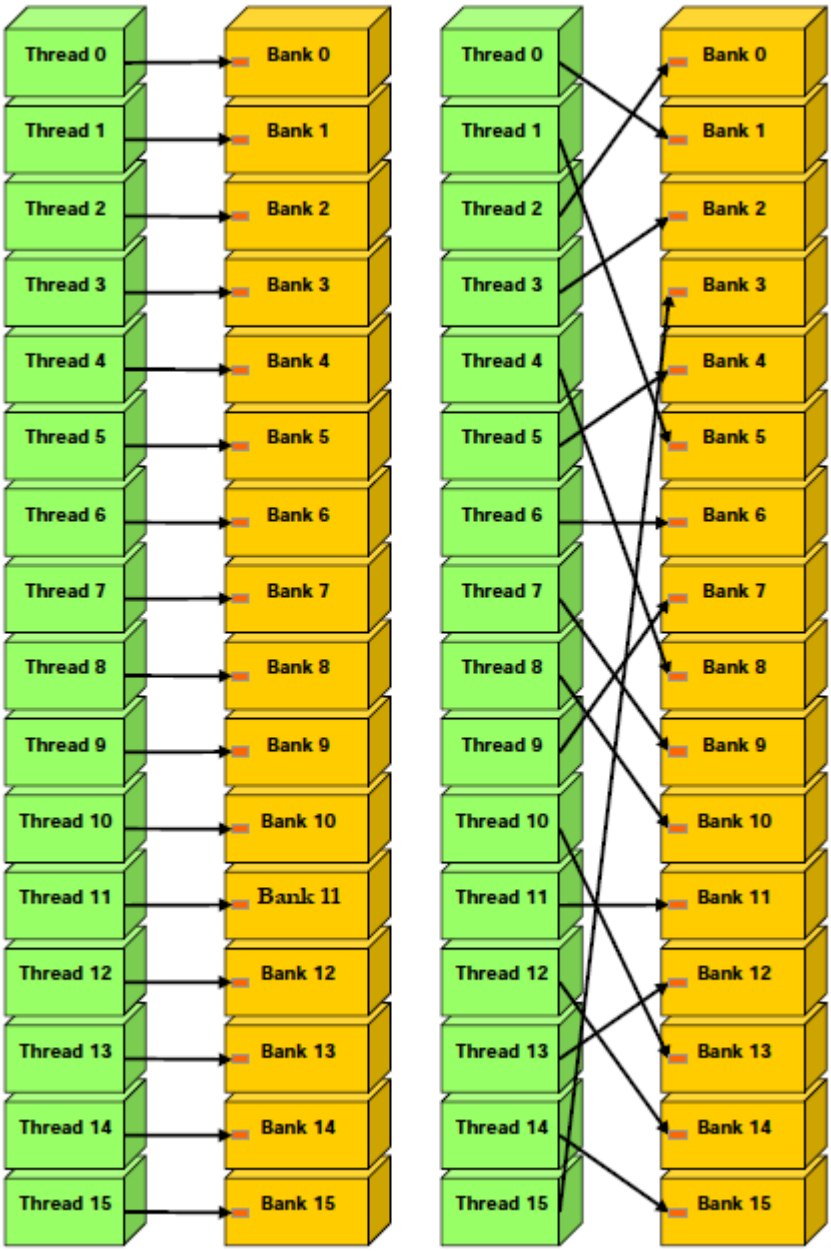

图 1-8-2 没有冲突的共享存储访问

下图中的两种访问模式则分别发生了 2 路 bank conflict 和 8 路 bank conflict。

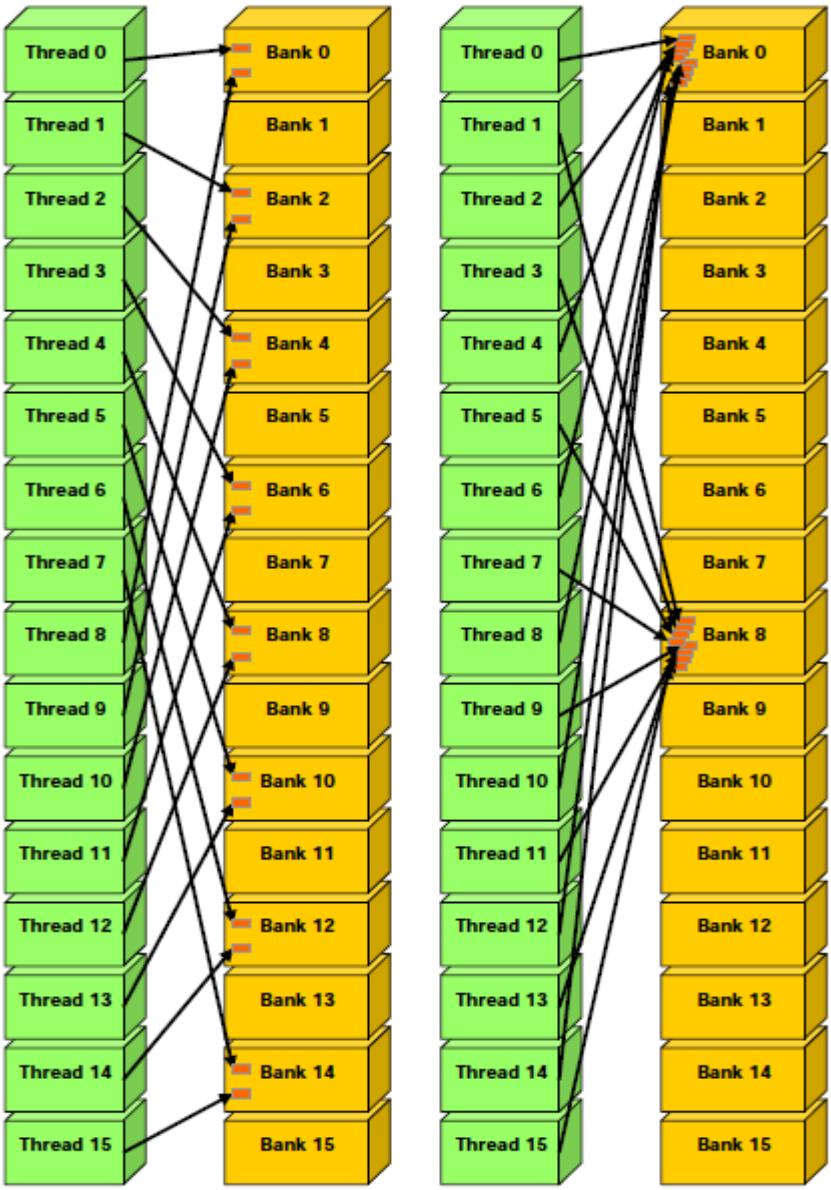

图 1-8-3　有冲突的共享存储访问

在实现中，GPU 要把显存中的数据写到共享存储器中，必须先把数据写到寄存器里，再转移到共享存储器中，在编程时，这是隐式实现的。所以如果没有块内的数据共享千万不能用共享存储器，否则会降低速度。但是如果由于寄存器使用过量，那么我们可以使用共享存储器来当寄存器使用，此时比纯使用寄存器慢一点，但是是值得的。

Tesla 能够在共享存储器内进行高速的原子操作。这里的原子操作是指保证每个线程能够独占的访问存储器，即只有当一个线程完成对存储器的某个位置的操作以后，其他线程才能访问这一位置。G80 只支持对 global memory 的原子操作。访问 global memory 需要很长的访存延迟（长达数百个时钟周期），性能很低。在 GT200 及以后的 GPU 上，可以支持对 shared memory 中的原子操作指令（其中包括 CAS 指令，并且支持 64 位）。但是，CUDA 并不提供对浮点数的原子操作(只有一个赋值的浮点原子指令)，而在科学计算中，浮点数的使用远比整数要多，在 Fermi 中，已经加入了浮点原子加指令。由于 Fermi 拥有一级和二级缓存，其原子函数效率要快得多，据官方数据大约 5x-20x。

除此以外，多处理器上，还有两种只读的存储器：常量存储器（constant memory）和纹理存储器（texture memory），它们是利用 GPU 用于图形计算的专用单元实现的。常数存储器空间较小（只有 64KB），属于片外存储器，其速度比 shared 要慢，但是它具有缓存，并且无须考虑冲突问题，主要用来加速对常数的访问。

从物理上说，纹理存储器不是存储器，它只是利用了纹理缓存而已。纹理缓存与 CPU 的缓存有很大的不同。首先，CPU 的缓存往往是一维的，因为大多数的架构中的存储器地址是线性的。当访问一个只有 4-8Byte 的数据字时，会取出一个缓存单元中所有的 64B 数据。根据局部性原理，CPU 处理的数据往往有很强的时空相关性，因此多取出的相邻的数据极有可能会被用到。CPU 处理的数据只有一维，因而其缓存也只是在一个维度上是连续的；GPU 需要处理的纹理则是连续的二维图像，因此纹理缓存也必须是在两个维度上连续分布的。典型的存储器控制器会将二维的纹理存储器空间映射为一维。其次，纹理缓存是只读的，也不满足数据一致性。当纹理被修改以后，必须更新整个纹理缓存，而不是纹理缓存中被修改的一小部分。第三，纹理缓存的主要功能是为了节省带宽和功耗，而 CPU 的缓存则是为了实现较低的延迟。第四，纹理可以实现对数据的特殊处理，比如怎样处理越界数据，自动实现插值等。

最后是全局存储器（global memory），使用的是普通的显存。整个网格中的任意线程都能读写全局存储器的任意位置。目前对 Global memory 的访问没有缓存，因此显存的性能对 GPU 至关重要。为了能够高效的访问显存，读取和存储必须对齐，宽度为 4Byte。如果没有正确的对齐，读写将被编译器拆分为多次操作，极大的影响效率。此外，多个 half-warp 的读写操作如果能够满足合并访问（coalesced access），那么多次访存操作会被合并成一次完成，从而提高访问效率。

G80 的合并访存条件十分严格。首先，访存的开始地址必须对齐：16x32bit 的合并必须对齐到 64Byte（即访存起始地址必须是 64Byte 的整数倍）；16x64bit 的合并访存起始必须对齐到 128Byte；16x128bit 合并访存的起始地址必须对齐到 128Byte，但是必须横跨连续的两个 128Byte 区域。其次，只有当第 K 个线程访问的就是第 K 个数据字时，才能实现合并访问，否则 half warp 中的 16 个访存指令就会被发射成 16 次单独的访存。

GT200 不仅放宽了合并访问条件，而且还能支持对 8bit 和 16bit 数据字的合并访问（分别使用 32Byte 和 64Byte 传输）。在一次合并传输的数据中，并不要求线程编号和访问的数据字编号相同。其次，当访问 128Byte 数据时如果地址没有对齐到 128Byte，在 G80 中会产生 16 次访存指令发射，而在 GT200 中只会产生两次合并访存。而且，这两次合并访存并不是两次 128Byte 的。例如，一次 128Byte 访存中有 32Byte 在一个区域中，另外一个区域中有 96Byte，那么只会产生一次 32Byte 合并访存（对有 32Byte 数据的区域）和一次 128Byte（对有 96Byte 数据的区域）。

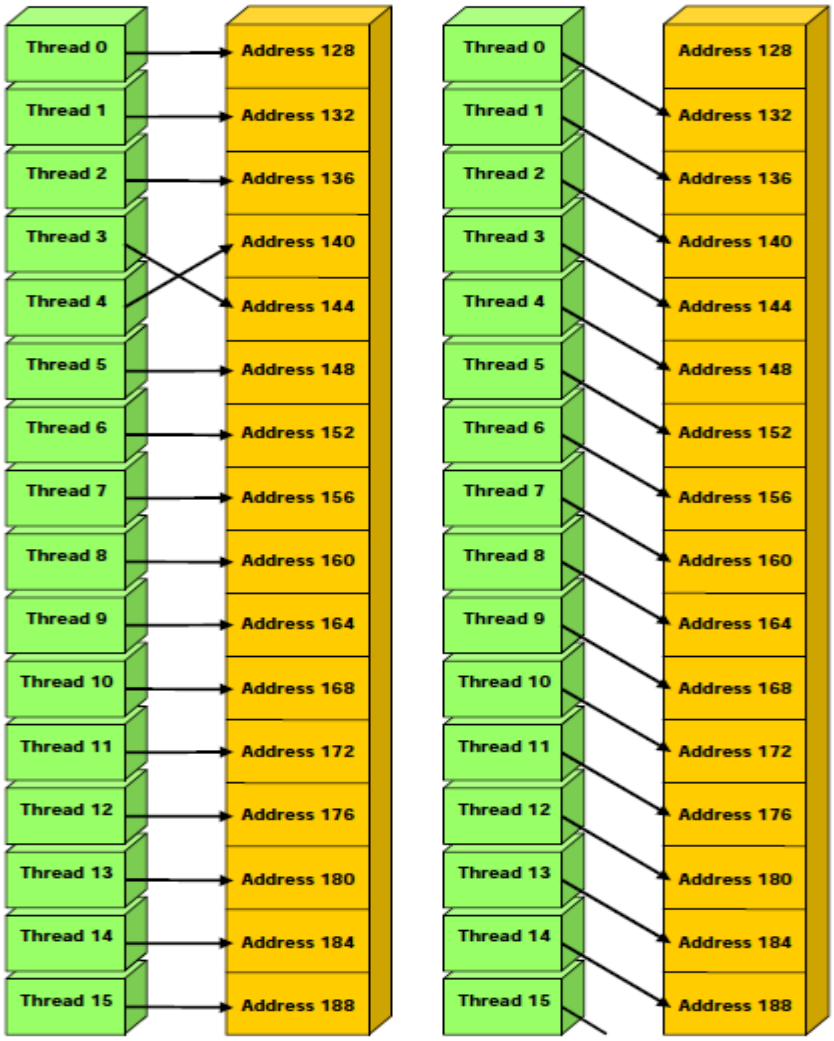

图 1-8-4　全局存储器访问

如上图所示，在 G80 硬件中，左边的访问模式会产生 16 次访问，只是因为 Thread 3 和 Thread 4 访问出现了交叉；而右边的模式也会产生 16 次访存请求，因为开始地址没有对齐；但在 GT200 硬件中，左边的只会产生一次访问，而右边的会产生再次。

在 Fermi 上，全局存储器会被缓存，每次读取的大小是 128 字节，这进一步降低了合并访问的影响，减小了优化难度。

Fermi 中，还有 768KB 的二级缓存，用来加速全局存储器和纹理存储器的读取。

在 G80 / GT200 的硬件上，由于全局存储器没有缓存，为了节约随机读取的带宽，使用常量存储器和纹理存储器是必不可少的，但是常量存储器和纹理存储器必须声明成全局变量，这会破坏程序的可扩展性和可维护性。

除了 device 端存储器外，还有存在于 host 端的存储器，即内存。在 CUDA 中，主机端内存分为两种：Pageable host memory 和 Page-locked host memory，其中 Page-locked host memory 保证位在于物理内存中，并且能够通过 DMA 加速与显卡的通信，提高数据传输速度，但是如果主机的内存不够用的话，会减弱系统的性能，但是一般不会出现这种情况。

第九节、执行模式

Tesla 架构的构建以一个可伸缩的流多处理器（SM）阵列为中心。当主机 CPU 上的 CUDA 程序调用内核网格时，网格的块将被枚举并分发到多处理器上。一个线程块的线程在一个多处理器上并发执行。在线程块终止时，调度单元将决定是否启动新块和启动那一个块。

为了管理、调度内核的成千上万个线程，多处理器利用了一种称为 SIMT（单指令、多线程）的新架构。多处理器会将各线程映射到一个标量处理器核心，各线程有自己的指令地址和寄存器状态，能够独立执行。多处理器以 warp 为单位来创建、管理、调度和执行线程，构成 warp 的各个线程在同一个程序地址一起启动，严格串行。

为一个多处理器指定了一个或多个要执行的线程块时，它会将其分成 warp，并由 SIMT 单元进行调度。将块分割为 warp 的方法总是相同的，每个 warp 都包含连续的线程，递增线程索引，第一个 warp 中包含全局线程过索引 0-31。每发出一条指令时，SIMT 单元都会选择一个已准备好执行的 warp，并将指令发送到该 warp。Warp 中线程每次都执行同一条指令，因此在 warp 内的全部 32 个线程执行同一条路径时，可达到最高效率。如果一个 warp 的线程因为条件分支而有不同的执行路径时，warp 将连续执行所使用的各分支路径，而禁用未在此路径上的线程，完成所有路径时，线程重新汇聚到同一执行路径下，其执行时间为各时间总和。分支仅在 warp 块内出现，不同的 warp 块总是独立执行的--无论它们执行的是通用的代码路径还是彼此无关的代码路径。

SIMT 架构类似于 SIMD（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，但通过维护很少使一个 warp 内的线程产生分支的代码，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 中的每个线程的寄存器都是私有的，线程之间只能通过 shared memory 和同步机制进行通信。

在 SIMT 编程模型中如果需要控制单个线程的行为，必须使用分支，这会大大的降低效率。例如，如果一个 warp 中需要进行分支（即 warp 内的线程执行的指令指针指向不同的位置），性能将急剧的下降。如果一个 warp 内需要执行 N 个分支，那么 SM 就需要把每一个分支的指令发射到每一个 SP 上，再由 SP 根据线程的逻辑决定需不需要执行。这是一个串行过程，因此 SIMT 完成分支的时间是多个分支时间之和。

第十节、NVIDIA GPU 结构

目前市场上的 NVIDIA 显卡基于 Tesla / Fermi 架构的，分为 G80、GT200、gf100 三个系列。Tesla 体系架构是一块具有可扩展处器数量的处理器阵列。图 2-1 是 GT200 图形处理器的示意图，每个 GT200 GPU 包含 240 个流处理器（streaming processor,SP），每 8 个流处理器又组成了一个流多处理器(streaming multiprocessors)，因此共有 30 个流多处理器。全规格的 Fermi 具有 512 个 SP，且对 SM 中 SP 的数量作了极大的提升，达到了 32 个，这潜在的提升了性能，因为不会对线程的数量提出过高的要求，但是相应的对取数的延迟提出了挑战。GPU 在工作时，工作负载由 PCI-E 总线从 CPU 传入 GPU 显存，按照体系架构的层次自顶向下分发。PCI-E 2.0 规范中，每个通道上下行的数据传输速度达到了 5.0Gbit/s，这样 PCI-E2.0×16 插槽能够为上下行数据各提供了 5.0*16Gbit/s=10GB/s 的带宽，故有效带宽为 8GB/s,而 PCI-E 3.0 规范的上下行数据带宽各为 20GB/s。但是由于 PCI-E 数据封包的影响，实际可用的带宽大约在 5-6GB/s（PCI-E 2.0 ×16）。

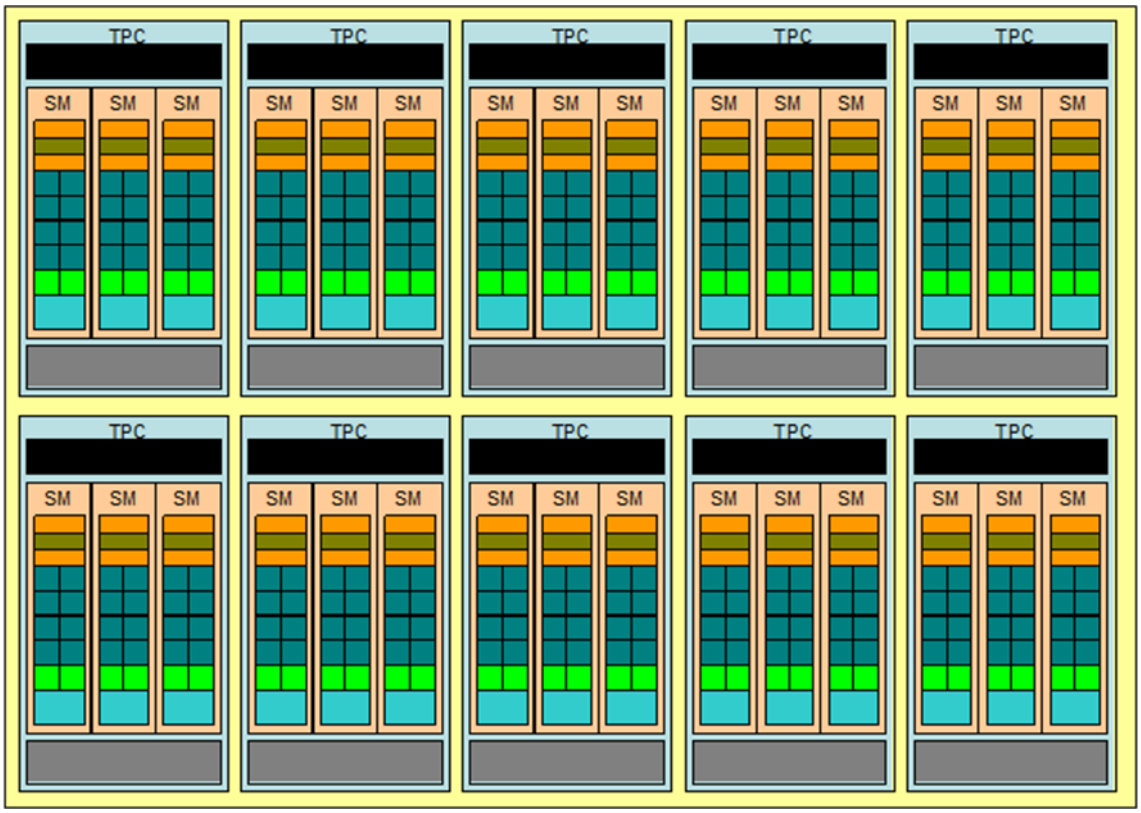

图 **错误！文档中没有指定样式的文字。**-1 GT200 图形处理器的示意图

在 GT200 架构中，每 3 个 SM 组成一个 TPC（Thread Processing Cluster，线程处理器集群），而在 G80 架构中，是两个 SM 组成一个 TPC，G80 里面有 8 个 TPC，因为 G80 有 128(2*8*8)个流处理器，而 GT200 中 TPC 增加到了 10(3*10*8)个，其中，每个 TPC 内部还有一个纹理流水线。Fermi 将 TPC 的概念替换成 GPC，每个 GPC 包含四个 SM,这样全规格的 Fermi 卡具有四个 GPC,

SM(Stream Multiprocessor)大致相当于一个 8/16 路的 SIMD 处理器，但指令宽度并不是 8，而是 32。和当代的 CPU 核一样，SM 也拥有独立完整的前端，包括取指、译码、发射、执行和调度单元等，但是一个 SM 的存储器流水线是与一个线程处理器群中的其它 SM 共享。SM 通过向量机技术增强计算性能，减少控制开销。大多数情况下，控制开销被 8/16 个 SP 分摊，但也有一部分的控制逻辑是直接控制单个线程的，无法共享。

GT200 和 G80 的每个 SM 包含 8 个流处理器。流处理器也有其他的名称，如线程处理器，“核”等，而最新的 Fermi 架构中，给了它一个新的名称:CUDA Core。 SP 并不是独立的处理器核，它有独立的寄存器和程序计数器(PC)，但没有取指和调度单元来构成完整的前端（由 SM 提供）。因此，SP 更加类似于当代的多线程 CPU 中的一条流水线。SM 每发射一条指令，8 个 SP 将各执行 4 遍。因此由 32 个线程组成的线程束（warp）是 Tesla 架构的最小执行单位。由于 GPU 中 SP 的频率略高于 SM 中其他单元的两倍，因此每两个 SP 周期 SP 才能对片内存储器进行一次访问，所以一个 warp 中的 32 个线程又可以分为两个 half-warp，这也是为什么取数会成为运算的瓶颈原因。Warp 的大小对操作延迟和访存延迟会产生影响，取 Warp 大小为 32 是 NVIDIA 综合权衡的结果。

SM 最主要的执行资源是 8 个 32bit ALU 和 MAD（multiply-add units，乘加器）。它们能够对符合 IEEE 标准的单精度浮点数（对应 float 型）和 32-bit 整数（对应 int 型，或者 unsigned int 型）进行运算。每次运算需要 4 个时钟周期（SP 周期，并非核心周期）。因为使用了四级流水线，因此在每个时钟周期，ALU 或 MAD 都能取出一个 warp 的 32 个线程中的 8 个操作数，在随后的 3 个时钟周期内进行运算并写回结果。

每个 SM 中，还有一个共享存储器(Shared memory),共享存储器用于通用并行计算时的共享数据和块内线程通信，但是由于它采用的是片上存储器，其速度极快，因此也被用于优

化程序性能。

每个 SM 通过使用两个特殊函数(Special Function Unit,SFU)单元进行超越函数和属性插值函数（根据顶点属性来对像素进行插值）计算。SFU 用来执行超越函数、插值以及其他特殊运算。SFU 执行的指令大多数有 16 个时钟周期的延迟，而一些由多个指令构成的复杂运算，如平方根或者指数运算则需要 32 甚至更多的时钟周期。SFU 中用于插值的部分拥有若干个 32-bit 浮点乘法单元，可以用来进行独立于浮点处理单元(Float Processing Unit,FPU)的乘法运算。SFU 实际上有两个执行单元，每个执行单元为 SM 中 8 条流水线中的 4 条服务。向 SFU 发射的乘法指令也只需要 4 个时钟周期。

在 GT200 中，每个 SM 还有一个双精度单元，用于双精度计算，但是其计算能力不到单精度的 1/8。

控制流指令（CMP,比较指令）是由分支单元执行的。GPU 没有分支预测机制，因此在分支得到机会执行之前，它将被挂起，直到所有的分支路径都执行完成，这会极大的降低性能。和算术指令一样，一个分支 warp 指令也需要 4 个时钟周期来执行。

第二章、CUDA 程序优化

每一节、CUDA 总体优化策略

一、选择程序中最耗时的部分，对它进行并行化，道理就是如果你选择了消耗运行时间 10%的部分来并行化，就算你达到了 10 倍的加速，现在的运行时间还是以前的 91%，但是如果你并行的是消耗运行时间 90%的部分，则现在的时间是以前的 19%。

二、最大化并行，就是在你选择的部分，使用尽可能多的线程来处理它，并且让每个线程尽可能多的计算，因为如果数据传输的时间太大的话，无论你提高多少倍也没多大作用，对于数据传输，首先要保证传输次数少，其次传输量要小，可以使用 cudaMallocHost 来加速传输。当然，这和问题本身，算法，硬件都有关系。

三、尽量保证全局内存融合，尽量使用 cuda 已经定义好的向量，他们往往已经对齐了，而且 cutil_math.h 中还有相关的操作符重载，对自己定义的结构体要使用对齐，如果实在无法实现内存融合的话，使用 texture

四、使用 share，constant 存储器，同时保证 share 不出现 memory bank,就算有严重的 bank conflict，shared 通常也快于 global。

五、优化 register 的使用，可以查看 cubin 文件，如果使用了 lmem，那么你就得考虑这个问题了，解决这个问题的方法，使用 share，减小 block 尺寸，后一条可能更实用。

六、注意条件分支语句，尽量少用分支；展开小循环，使用 inline 减少函数调用。

七、优化指令使用，尽量使用吞吐量大的指令，而不是相反，比如-use_fast_math 编译选项。

第二节、计时器的设计

判断程序优劣的最简单方式就是计算程序的运行时间，在同一台机器上，运行时间短的程序一般来说是更优的，当然不能一概而论，毕竟决定程序运行速度的因素很多，比如算法，机器的指令集，使用的语言等等。

我们的目的是要设计一个既能够在 windows 上运行，又能够在 Linux 上运行，既可以用于C，又可以用于 CUDA 的计时器，同时最好便于扩展。计时的方法很多，如标准C库的 time、clock 系列函数，CUDA 的事件，linux 下的 gettimeofday,windows 下的 GetTickCount。

本计时器系列采用C＋＋的面向对象设计方法，基本的思路是设计一个父类，然后再从父类创造子类。这样要建造一个新计时器，只要继承父类就行了。

父类代码如下：

```
class TimeCounter{
        protected :
            clock_t startp,endp;
        public :
            TimeCounter():startp(-1),endp(-1){
            }
            void start(){//设置计时起点
            #ifdef __CUDACC__
                cudaThreadSynchronize();
            #endif
                startp=clock();
            }
            void stop(){//设置计时终点
                if(-1==startp){
                    perror("you must set start point at first");
                }else{
                #ifdef __CUDACC__
                    cudaThreadSynchronize();
                #endif
                    endp=clock();
                }
            }
            virtual long getTimeDiff()=0;//返回时间差滴答数
            virtual void printTimeDiff()=0;//打印出时间差
    };
```

子类秒计时器代码如下：

```
  class SecondCounter:public TimeCounter{
        public :
             long getTimeDiff(){
                 if(-1==endp){
                    perror("you must set stop point before invoke this function");
                    exit(1);
                }else{
                    return (endp-startp)/CLOCKS_PER_SEC;
                }
            }
            void printTimeDiff(){
                long temp=getTimeDiff();
                printf("use time :%lds\n",temp);
            }
    };
```

子类毫秒计时器代码如下：

```
    class MillisecondCounter:public TimeCounter{
        public :
            long getTimeDiff(){

                if(-1==endp){
                    perror("you must set stop point before invoke this function");
                    exit(1);
                }else{
                    return 1.0f*(endp-startp)/CLOCKS_PER_SEC*1000;
                }
            }
            void printTimeDiff(){
                long temp=getTimeDiff();
                printf("use time :%ldms\n",temp);
            }
    };
```

子类微秒计时器代码如下：

```
#ifdef __CUDACC__
    class MicrosecondCounter:public TimeCounter{
        public:
            long getTimeDiff(){
                if(-1==endp){
                    printf("please set start point or end point\n");
                    exit(1);
                }else{
                return 1.0f*(endp-startp)/CLOCKS_PER_SEC*1000000;
                }
            }
            void printTimeDiff(){
                long temp=getTimeDiff();
                printf("use time:%ld us\n",temp);
            }
    };
#endif
```

这种设计使得我们可以声明一个计时器父类引用，而让其实际指向子类，这样，如果精度小了的话，就可以只更改一处，而不用更改其它的内容了。在我的实践中，这种基于对象的多态设计的计时器工作得很好。在开始计时处调用 start 方法，在计时终点调用 stop 方法，调用 getTimeDiff 方法返回经过的时间数，但是这个方法有个问题就是如果两个不同的子类的计时器，其用此函数获得的值不能直接相加减，由于这个问题我本来想将其设置 private,但是最终还是让其保持 public，因为这符合 C/C++的设计思想：程序员是最聪明的，他们应当知道会发生什么，一切由他们来决定。函数 printTimeDiff 打印计时时间。

如果有一天C语言的 clock 函数精度达到了微秒，我们只用将包围 MicrosecondCounter 类的条件编译语句去掉就行了。如果有一天其精度达到了纳秒，我们只用再写一个类继承 TimeCounter 类就行了，不用改写任何代码。

第三节、错误处理

标准C提供了错误输出函数perror,其功能是将在调用其处最近之前出错的错误输出到的标准错误流，当然利用操作系统的功能，你也可以将它们重定向到文件。

由于CUDA函数设计为都返回错误码，而且CUDA本身也提供了两个函数cudaGetLastError和cudaGetErrorString，因此其处理可充分利用这两个函数，另外我们在写CUDA程序时，经常要验证结果和CPU上的结果是否一致，因此我也提供了这个功能。

程序代码如下：

```
class ErrorHander{//file:file name;line:lineNo
    const static float EPSFLOAT = 0.000001;
    const static double EPSDOUBLE = 0.0000000001;
    public :
#ifdef __CUDACC__
    //can't use it at asynchronize situation
        void static printError(cudaError_t cet){
            if(cudaSuccess != cet){
                printf("error:%s\n",cudaGetErrorString(cet));
                fflush(stdout);
                exit(1);
            }
        }
    //can use it at anywhere
        void static printLastError(){
            cudaThreadSynchronize();
            cudaError_t cet = cudaGetLastError();
            if(cudaSuccess != cet){
                printf("error:%s!\n",cudaGetErrorString(cet));
                fflush(stdout);
                exit(1);
            }
        }
        void static printError(int line,cudaError_t cet){
            if(cudaSuccess !=cet){
                printf("line:%d,error:%s\n",line,cudaGetErrorString(cet));
                fflush(stdout);
                exit(1);
            }
        }
```

```
        void static printLastError(int line){
            cudaThreadSynchronize();
            cudaError_t cet = cudaGetLastError();
            if(cudaSuccess != cet){
                printf("line:%d,error:%s!\n",line,cudaGetErrorString(cet));
                fflush(stdout);
                exit(1);
            }
        }
        void static printError(char *file,int line,cudaError_t cet){
            if(cudaSuccess != cet){
                printf("%s,line:%d,error:%s\n",file,line,cudaGetErrorString(cet));
                fflush(stdout);
                exit(1);
            }
        }
        void static printLastError(char *file,int line){
            cudaThreadSynchronize();
            cudaError_t cet = cudaGetLastError();
            if(cudaSuccess != cet){
                printf("%s,line:%d,error:%s!\n",file,line,cudaGetErrorString(cet));
                fflush(stdout);
                exit(1);
            }
        }
#endif
        void static valid(const int *first,const int *second, int num, int firstSpan = 1,
                int secondSpan = 1, int firstOffset = 0, int secondOffset = 0){
            for(int i=0;i<num;i++){
                assert(first[firstOffset+i*firstSpan]==second[secondOffset+i*secondSpan]);
            }
        }
        void static valid(const float *first, const float *second, int num, int firstSpan = 1,
            int secondSpan = 1, int firstOffset = 0, int secondOffset = 0){
            for(int i=0;i<num;i++){
    assert(abs(1-(first[firstOffset +
i*firstSpan]/second[secondOffset+i*secondSpan]))<EPSFLOAT);
    }
             }
            void static valid(const double *first, const double *second, int num, int firstSpan = 1,
              int secondSpan = 1, int firstOffset = 0, int secondOffset = 0){
                  for(int i=0;i<num;i++){
    assert(abs(1-(first[firstOffset +
i*firstSpan]/second[secondOffset+i*secondSpan]))<EPSDOUBLE);
              }
         }
};
```

printError 函数打印传入的 cuda 函数返回的错误码所代表的错误，printLastError 打印最后一次错误的错误字符串。file 代表文件名，调用时可用__FILE__宏代替，line 代表行数，调用时可传入__LINE__宏，当然由于设计上的缺陷，你也可以传入其它的信息。

valid 函数用于确定两个数组的数据是不是一样的，本来想使用模板，但是还是使用了重载。由于全部采用静态函数，所以我们没有必要建立对象，直接使用类名调用就行了，这样可以方便的使用条件编译来启用 / 关闭调试信息，这一点比 cutil 库的 cudaSafeCall 好用得多。另外这个类的功能也足够使用了。如果你想改变精度限的话，改变 EPS 变量的值就可以了。非常的方便。

另，由于能力所限或者说设计缺陷，printError 各函数都无法处理异步调用错误。另外这些工具函数只是拿来调试用的，真正的出错处理不能这样做。

当然我个人的库包含的类远比这要多，比如还有文件输入输出，随机数初始化等。这些内部就不包括在本节了。

整体而言，这两节都是一些准备工作。从下一节起才真正涉及到优化的内容。

第四节、串行 C 程序的优化

相信大家看到这个题目会有所疑问，毕竟要说的ＣＵＤＡ程序优化，而这里说的是串行程序的优化，我要说的是串行程序优化的能力对于并行程序优化同等的重要，因为大多数时候我们是从串行程序到并行程序，而且并行线程的内部依旧是串行的。

一般而言，算法级别的优化是最有效的，但是本文的优化并不涉及，本文假设你已经有了一个可以运行并能够得到正确结果的程序，在此基础上进行优化。

1 编译器选项

这是最简单的一种优化方法，一般而言使用高级的优化选项总比使用低优化选项的速度要快一些，如 gcc 就有ｏＯ，ｏ１，ｏ２，ｏ３三个优化选项，一般使用ｏ２就已经足够，注意有时Ｏ３优化选项会更慢甚至产生错误的结果。

2 缓存优化

程序访问的数据在空间和时间上都有其局部性，也就是说一个数据被访问了，它附近的数据也很有可能被访问，因此，如果我们在访问下一个数据的时候，其已经在缓存中也就是说缓存命中，此时就用不着从内存中取数据，可加速访问。

比如有一个二维数组ａ［Ｍ］［Ｎ］，大家觉得下面的代码，那个快呢

```
代码一：
for(int I = 0; I < M; i++)
      for(int j = 0; j < N; j++)
            a[i][j] *= 3;

代码二、
for(int I = 0; I < M; i++)
      for(int j = 0; j < M; j++)
            a[j][i] *= 3;
```

当然还有其它一些优化缓存的方法，比如分块等。缓存的优化，往往对程序的性能有决定性的改变。

3 选用尽量小的数据类型

一般而言，小数据类型的计算速度比大数据类型要快

4 结构体声明

声明结构体时，尽量大数据类型在前，小数据类型在后。这样会节省一些空间，你可以自己验证一下，只要 sizeof 运算符就可以了。在作用 CUDA 时可以使用__align__(xx)来确定对齐的字节数。

5 使用位运算

因为位运算要比算术运算快，因此使用位运算取代算术运算是有利的。但是要注意的是由于ＧＰＵ

没有ＥＣＣ，使用位运算的时候出错的概率有点大。

如乘以一个数可以转变成左移操作，如果乘数是常量，编译器会自动转换；除以２的幂可换成右移；模２的幂可换成位与。

6 使用复合运算符

在Ｃ中使用＋＝，－＝，＋＋，－－等复合运算符会比较快。

7 展开小循环

展开小循环不但省略了每次的判断，更使用编译器使用向量化和并行指令成为可能。如下面的代码，代码一效率就比代码二高

```
代码一
float sum = 0.0f;
for(int I = 0; I <100000; I++){
    Sum += a[I];

float sum = 0.0f, sum1 = 0.0f, sum2 = 0.0f, sum3 = 0.0f;
for(int I = 0; I < 25000; I +=4){
    Sum1 +=   a[I];
    Sum2 +=   a[I+1;
    Sum3 +=   a[I+2];
    Sum +=   a[I+3];
}
Sum += sum1+sum2+sum3;
```

8 表达式移除

假设有一向量ｖ长为Ｎ；下面的代码一就比代码二快

```
代码一
Vector<int>   vi;
Int len = vi.size();
For(int I = 0; I < len; i++)
…
代码二
vector<int>   vi;
…
for(int i = 0; I < vi.size(); i++)
….
```

9 判断式

在Ｃ中判断式是短路的，也就是说如果现在的信息已经能够决定整体的结果，后面的就不用算了。如 if(a && b)，如果ａ为假，那么 if 里语句就一定不能执行，故ｂ不用再求值，这样如果ａ，ｂ中有一个是计算量相当大的，就应当将它放在后面，如果计算量差不多，就把ａ，ｂ中为假概率大的放在前面。对于||可以类推。

10 查表

查表在图形学非常常用。比如有一批数据，我们要知道每个数据出现的次数，我们就没有必要每次都去统计，只要统计一次，然后记录结果，在每次查的时候，只要查记录的结果就行了。

其它

如声明 float 时加 f 后缀，使用 const，static，少用虚函数等。

最后要说的是，很多时候编译器能够帮助我们优化，因此，这里说的优化方法在某些编译器下，或者某些编译器开启了某些选项的情况下，并不有效。

第五节、ＣＵＤＡ程序的优化

对于一个问题来说，往往有很多办法可以解决它。选择那个方法，这是一个问题，而且是一个与政策、技术，甚至阴谋相关的问题，具体怎么做，往往考量很多。或者说条件不一样，评判的标准不一样，选择的方法也就不一样。编程也类似，一般来说，不是为了特地使用 GPU,就没有必要没有原因就将原来的ＣＰＵ代码修改为ＧＰＵ。

说这段话，是因为发现有很多人在不适合 GPU 的情况下，却使用它，结果当然很坏具了。

使用ＣＵＤＡ可分两种情况，一种是从头到尾从新设计；一种是已经有了串行的版本，但不能满足速度要求，想通过使用ＣＵＤＡ来加快速度，此时可以考虑将代码中最耗时的部分使用 CUDA 重写。

1 选择合适的工具

由于历史的原因，我们目前所使用的串行编程模式是完全一致的，说白了就是以控制为中心，虽然面向对象说是以数据为中心，但是我认为其本质上还是以控制为中心，大家想想看你写面向对象的代码时，其整个中心是不是处理逻辑？是不是系统要做什么？而目前的并行编程模式却分为数据并行和任务并行，或者分布式存储并行和共享存储并行，ＣＵＤＡ是基于数据并行和共享存储器的并行，如果你有 OpenMP 或者 MPI 的编程经验，相信你也许也会有这种看法。

所以如果你要解决的问题并不是数据并行的或者共享存储器的，使用 CUDA 就绝不是一个好的选择。

2 优化最应该优化的地方－优化准则

相信很多人都知道 80%-20%定律，也就是说程序的 80%时间花在执行 20%的代码上，如果我们将那 20%的代码加速了１０倍，最终也才加速一点点，如果我们将那８０％加速了一倍，最终的结果将加速近一倍。

根据公式 s = 1/(f + (1-f)/p),其中ｆ是程序中串行代码的比重，ｐ是处理器数目。可知ｆ越大，ｓ就越小。这也许解释了ｎｖ的说法（尽量让更多的代码在 GPU 上执行，就算没有加速比）。也就是尽量提高并行代码的比重。这潜在的能够提升程序的并行效率，Fermi 的某些功能显然是向前迈进了一些，但是ｎｖ是不是真的能够达到他们的设想，那就是另一回事了。

3 全局存储器优化

在 CUDA 的存储器结构中，全局存储器是最慢的，其延迟在几百个时钟，但是其容量是最大的，我们使用 cudaMalloc 函数分配的指针都是指向全局存储器空间的，我们不可能避开它们的使用。这两个因素使得全局存储器的使用对 CUDA 程序的性能至关重要。

3.1 合并访问

为了提高访问全局存储器的效率，CUDA 提供了一种称为合并访问的机制来加速全局存储器的访问。在计算能力 1.0 和 1.1 的设备上，合并访问能够在一次存储器访问中访问最多达 16 个数据；在计算能力 1.2 和 1.3 的设备上，合并访问一次最多能够访问 32 个数据。另外线程间的切换也能够隐藏全局存储器的访问延迟。

所谓合并访问就是尽量要求相邻的线程访问相邻的地址空间，当然在不同计算能力的设

备上，其具体要求有所差别，在１．０和１．１的设备上，合并访问要求第ｋ个线程访问对齐的第ｋ个存储地址，必须对齐，不能交叉，而１．２以上的设备没有此要求。

对于不同的数据类型，合并访问的带宽不相同，一般而言使用四字节的数据类型比较好，但是并不绝对，在某些情况下，使用８字节或１６字节的数据性能可能会更好，尤其是在满足１．０和１．１设备的合并访问条件下。

在设计的时候要考虑到全局存储器的合并访问，并以此为要求来设计，比如在必要的时候对矩阵进行转置。对数据的组织方式加以改变。

4 共享存储器优化

共享存储器比较靠近ＳＭ的，因此其速度相当的快，一般而言可以在一到二个时钟内读写，因此使用共享存储器取代全局存储器会极大的节约带宽，但是共享存储器的使用是有要求，它要求数据有局部性（一个ｂｌｏｃｋ内共享）重用。当然有时也可用共享存储器取代寄存器，这会在后面说到。

一般的使用共享存储器的方式是：s_x[tid] = x[id];其中 tid 是块内线程的块内索引，也就是类似 threadIdx，id 是线程的网格内索引。然后可以访问ｓ__ｘ取代访问ｘ。这样一般可以提升程序的效率。

要注意的是，使用共享存储器往往伴随着存储器的一致性问题，此时可求助于__syncthreads()和 memory fence 大神。

4.1 存储体冲突

类似于合并访问，共享存储器一次也能满足　半束线程的要求，条件是这半束线程访问的数据在不同的存储体中。CUDA 将共享存储器组织成 16 列，每一列称为一个存储体。如果有两个或以上的线程访问的数据在同一个存储体中，此时不能在一次存储体访问中，满足半束线程的要求，这称为存储体冲突。一般而言，存储体冲突对性能的影响还不是太大，当然具体的影响要看半束内最多有多少线程落入同一存储体内。要提到的

编程时要注意的是对于一个ＳＭ来说，共享存储器的数量是有限的，如果超出其使用限制，其结果未知，我的经验告诉我，有时没有问题，有时会有大问题。另外，使用共享存储器有时可能会影响程序的可扩展性，这个在编写库代码的时候要特别注意。

5 常量存储器优化

CUDA 允许分配最多 64ＫＢ的常量存储器，常量存储器顾名思义内容是不变的，所以也有人称其为不变存储器。每个ＳＭ有６－８ＫＢ的常量缓存，一般而言一到两个周期可读取常量存储器。如果半束内的线程访问的不是同一个地址，那么各个线程的访问将会串行化。

常量存储器的设计个人认为不是太好，有点像鸡肋。因为要求常量存储器是全局的，因此对程序的可读性和可扩展性都有影响。

对于常量存储器大小的限制问题，有些情况下，可以使用多次导入并多次执行内核的方式解决。

6 纹理存储器

纹理存储器是来自图形学的一个概念。由于硬件的支持，提供了很多额外处理功能，如边界处理、滤波等。访问纹理存储器要通过纹理参考和纹理获取。

ＣＵＤＡ对于随机存取的性能极其的悲剧，而纹理存储器可减弱这种悲剧的效果，尤其是访问的数据之间具有极大的局部性的时候。

另外在某些情况下可以利用纹理的插值功能来加速计算。此时要注意的纹理执行的是低精度的线性插值。

7 寄存器

寄存器是NVIDIA GPU存储器空间中，最快的硬件，读取它几乎不用耗费时间。因此合理的使用它是至关重要的。

7.1 寄存器溢出和本地存储器

由于ＧＰＵ上寄存器的数目是有限的，在计算能力1.0,1.1的设备上，其数目是8 K，在计算能力1.2,1.3的设备上，其值是16 K。如果我们使用的寄存器数目超过了系统的最大数目，编译器便会将其转入设备存储器上的一个区域内，这个区域称为本地存储器。本地存储器的速度和全局存储器一样。

使用2.3及以前的toolkit,可以使用-keep指令输出中间文件，然后直接查看cubin文件就可以知道内核使用的寄存器数目，同时也能看到共享存储器、常量存储器和本地存储器的使用量。如果使用的是3.0的toolkit，此时cubin文件已经不是文本格式，因此不能直接查看，此时可以在用nvcc 编译的时候，使用—ptxas-options=-v选项，编译器会报告存储器使用信息。

8 执行配置和占用率

使用<<<grid, block>>>语法指定执行线程配置的时候，grid 和block大小也影响程序的效率。一般而言，grid要大于多处理器的数目，这样才能让多处理器不至于空闲，但是这样也会导致一些问题，比如负载均衡，如果grid大小不能比sm 数目整除的话，就会有ＳＭ计算的时候，另外一些ＳＭ空闲，如果grid远大于ｓｍ数目的话，可忽略，但是如果ＳＭ数目与ｇｒｉｄ大小相差不大的话，性能损耗就很可观了。一般而言，ｇｒｉｄ大小至少是ｓｍ数目的三倍，如果有块内同步的话，ｇｒｉｄ要大于ｓｍ的四倍以上。一般而言，ｂｌｏｃｋ大小要是束大小的四倍以上，此时基本上可隐藏访存延迟。如果数据量比较小的话，ｇｒｉｄ大小和ｂｌｏｃｋ大小可能相互牵制，此时要综合考虑。在数据量比较大的时候，只要考虑ｂｌｏｃｋ大小就行了。

另外，CUDA还有一个占用率问题，所谓占用率就是ＳＭ上活跃块数目和最大允许数目的比例。使用ｃｕｄａ　ｖｉｓｕａｌ　ｐｒｏｆｉｌｅｒ的时候会有这个值。这个值的影响因素有内核内寄存器、共享存储器的用量。由于CUDA线程在切换的时候并不转储线程的状态，这是ＣＵＤＡ线程极轻的原因，但是这也使得为了保证线程能够切换，内核内使用的资源最多只能是最大资源的一半，此时占用率是0.25，一般而言占用率要在0.5左右。

9 指令优化

各种指令有不同的执行时间，这样要更快的速度可尽量使用占用时间少的指令，比如x *= 2; 最好写成x += x;因为乘法的占用的时间比加法多，但是这个并不明显，因为在现代处理器上乘法的效率已经和加法差不多了。要注意的是除法和模余，它们占用的时间远大于加法，要尽量少使用。另外一些超越函数占用的时间更长，为此CUDA提供了超越函数的更快版本，更快的代价是精度损失。所以在速度远比精度重要的情况下，可以使用，但是如果精度比较重要的话，就要注意了。

9.1 寄存器依赖

在程序的一些部分，我们经常要存取数据，如果在后面的比较近的指令中要读取这个数据，此时就要等到这个数据存取完成，这就是寄存器依赖，其后果就是由于要等待导致的性能损耗。这种损耗往往可以通过重新安排程序的语句减弱。

10 分支优化

在 CUDA 中，分支会极大的减弱性能，因为ＳＭ没有分支预测，因此只能让束内线程在每个分支上都执行一遍，当然如果某个分支没有线程执行，就可以忽略，因此要减少分支的数目。但是在实践中很多时候分支是没有办法减少的。此时目前还没有好的解决方案。

第三章、一些例子

第一节、 两向量的距离

本程序中，我以 a[N]和 b[N]代表两个向量，其欧氏距离计算的串行Ｃ代码如下：

```
dis = 0;
for(int i = 0; i < N; i++){
      dis += (a[i] - b[i]) * (a[i - b[i]);
}

dis = sqrt(dis);
```

其中 dis 表示两个向量的距离，sqrt 表示求平方根函数。从代码可以看出，我们可以使用Ｎ个线程，每个线程计算 a,b 一维上数据差的平方，但是由于这个计算量太小，为了加大计算量，我采用了一个线程计算多个数据差的平方和，在这步后，我们要将所有线程数据加起来以得到最终的结果，考虑到ＣＵＤＡ不支持全局同步而支持块内同步，因此我们可以先得到每个块的和，然后再将每个块的和保存到全局存储器 d_temp 中。最后再重新建立一个内核来将ｄ__ｔｅｍｐ的数据求和，由于此时数据量小，因此只要一个块就够了。

考虑到分支会极大的影响到 CUDA 程序的性能，因此我们将块内各线程数据加和的方式是折半相加，此时可以利用高速的共享存储器。折半相加的思想是：假设有 n ＝ 2^k 个线程，那么第一次相加在 0 和 n/2, 1 和 n/2+1,……，n/2-1 和 n-1 之间进行。此后线程 0 至 n/2-1 拥有的数据和等于之前块内所有数据和，然后可以进行下一次折半相加。在再次折半相加之间为了保证数据一致性，必须使用__syncthreads()函数在块内所有线程间同步。具体代码如下：

```
template<typename T> static __global__ void reduceMultiBlock(const T*
__restrict__ d_a, const T* __restrict__ d_b, const int dim, T* __restrict__
d_temp){
//global thread id
1.    unsigned int id = blockDim.x*blockIdx.x + threadIdx.x;
//iterate length
2.    unsigned int iter = blockDim.x*gridDim.x;
//for temp distance, length = blockDim.x
3.    extern __shared__ T s_dis[];
//use for when dim is shorter than iter
4.    s_dis[threadIdx.x] = 0;

5.    T temp;
6.    T tempDis = (T)0;

7.    for(int i = id; i < dim; i += iter){
8.          temp = d_a[i] - d_b[i];
9.          tempDis += temp*temp;
10.   }

11.   s_dis[threadIdx.x] = tempDis;

12.   __syncthreads();

13.   for(int i = (blockDim.x>>1); i > 0; i >>= 1){
14.         if(threadIdx.x < i)
15.               s_dis[threadIdx.x] += s_dis[i + threadIdx.x];
16.         __syncthreads();
17.   }

18.   d_temp[blockIdx.x] = s_dis[0];
}
```

为了使代码能够同时应用于整形和浮点型，使用了模板机制。下面来具体分析：

1.取得当前线程的索引，CUDA 线程有两种组织，一是网格，一是块。

2.取得整个网格内的线程数。

3.动态声明共享存储器，其大小由内核调用的第三个参数决定，单位是字节。

4.初始化共享存储器为 0。

5.-6.声明两个寄存器，并初始化一个为 0。

7.-10.遍历两个向量的所有维并将向量对应维差的平方加到对应线程的寄存器上，为了保证全局存储器的合并访问，采用了循环读取模式，循环读取是指，假设一个有 5 个线程， 1 0 个数，那个线程 0 就读第 0 和第 5 个数，线程 1 读第 1 个和第六个数，以此类推。

11.-12.将块内线程取得的对应维差的平方和存入共享存储器并同步，同步的目的是防止有些线程可能先执行 15.语句而取得了不正确结果。

13.-17.将块内共享存储器的数据全加到共享存储器数组的第 0 个元素上。

18. 将整个块的数据和存入全局存储器。为什么要做这一步的原因在于：共享存储器并不持久，当内核执行完后，其就会被回收，而全局存储器是持久的。这类似于ＣＰＵ中内存和栈。

在获得每个块的数据和并存入全局存储器后，下一步要做的就是将上一内核存入全局存储器的数据加起来得到最终的结果。由于内容和上一内核函数差不多，因此就不详细说了。

代码如下：

```
template<typename T>
static __global__ void reduceOneBlock(T* __restrict__ d_temp, const int len){
    unsigned int tid = threadIdx.x;
    T temp = (T)0;
//length == blockDim.x
    extern __shared__ T s_dis[];
//use for when len is shorter than d_temp's length
    s_dis[tid] = (T)0;

    for(int i = tid; i < len; i += blockDim.x){
        temp += d_temp[i];
    }

    s_dis[tid] = temp;

    __syncthreads();

    for(int i = (blockDim.x>>1); i > 0; i >>= 1){
        if(tid < i)
            s_dis[tid] += s_dis[i + tid];
        __syncthreads();
    }
    if(0 == tid)
        *d_temp = s_dis[0];
}
```

经过这两个内核的处理，d_temp[0]就是向量所有维差的平方和了。我们只要将其从显存上复制到内存并求其平方根就得到了这两个向量的距离。

第二节、 矩阵与向量乘积

本文使用一个线程块计算矩阵的一行和向量的乘积的模式。主要代码如下：

```
void __global__ matrixMultiVectorBlock(const int rowSize, const int columnSize,
const int pitchItem, const float* __restrict__ d_matrix,   const float* __restrict__
d_vec, float* __restrict__ d_r){
    unsigned int tid = threadIdx.x;
    extern __shared__ float s_r[];
    float temp = 0.0f;
    for(int i = tid; i < columnSize; i += blockDim.x){
        temp += d_matrix[blockIdx.x*pitchItem+i]*d_vec[i];
    }

    s_r[tid] = temp;
     __syncthreads();
    for(int i = (blockDim.x>>1); i > 32; i >>= 1){
        if(tid < i){
            s_r[tid] += s_r[tid+i];
        }
        __syncthreads();
    }

    if(tid < 32){
        s_r[tid] += s_r[tid+32];
    }
    if(tid < 16){
        s_r[tid] += s_r[tid+16];
    }

    if(tid < 8){
        s_r[tid] += s_r[tid+8];
    }

    if(tid < 4){
        s_r[tid] += s_r[tid+4];
    }

    if(tid < 2){
        s_r[tid] += s_r[tid+2];
    }
    if(tid < 1){
        s_r[tid] += s_r[tid+1];
        d_r[blockIdx.x] = s_r[0];
    }
}
```

这种做法的主要思想还是 reduction，在矩阵的列数和行数都比较大的时候，这种方法的性能很好，但是如果矩阵是那种行比较多，列比较少的话，应该怎么做？我们可以使用一个 warp 计算一行（有人称之为 warp 模式），代码如下所示：

```
void __global__ matrixMultiVectorWarp(const int rowSize, const int columnSize, const int pitchItem,    const float* __restrict__ d_matrix, const float* __restrict__ d_vec, float* __restrict__ d_r ){
	unsigned int warpid = threadIdx.y + blockIdx.x*blockDim.y;

//length equal to two block size
	extern __shared__ float s_s[];
	float *s_v = s_s;
	unsigned int blockSize = blockDim.y*blockDim.x;
	float *s_r = s_s + blockSize;
	unsigned int tid = threadIdx.x + threadIdx.y*blockDim.x;

	s_r[tid] = 0.0f;
	 int iterTimes = (columnSize+blockSize-1)/blockSize;
	for(int i = 0; i < iterTimes ; i++){
		if(i*blockSize+tid < columnSize){
			s_v[tid] = d_vec[i*blockSize+tid];
		}
		__syncthreads();

		int temp = min(blockSize, columnSize-i*blockSize);
		for(int j = threadIdx.x; j < temp; j += blockDim.x){
			s_r[tid] += d_matrix[i*blockSize+j+pitchItem*warpid]*s_v[j];
		}
		__syncthreads();
	}

	if(warpid >= rowSize){
		return;
	}

	if(threadIdx.x < 16){
		s_r[tid] += s_r[tid+16];
	}

	if(threadIdx.x < 8){
		s_r[tid] += s_r[tid+8];
	}

	if(threadIdx.x < 4){
		s_r[tid] += s_r[tid+4];
	}

	if(threadIdx.x < 2){
		s_r[tid] += s_r[tid+2];
	}

	if(threadIdx.x < 1){
		s_r[tid] += s_r[tid+1];
	}

	if(threadIdx.x == 0){
		d_r[warpid] = s_r[threadIdx.y<<5];
	}
}
```

由于如果没有 host 端代码的话，上面的代码非常难以理解，因此本文提供 host 端的代码以供参考。

```
void runMatrixMultiVectorWarpGPU(const int rowSize, const int columnSize, const float *matrix,
const float *v, float *r){
    float *d_matrix;
    size_t pitch;
    cutilSafeCall(cudaMallocPitch((void**)&d_matrix, &pitch, columnSize*sizeof(float),
rowSize));
    cutilSafeCall(cudaMemcpy2D(d_matrix, pitch, matrix, columnSize*sizeof(float),
columnSize*sizeof(float), rowSize, cudaMemcpyHostToDevice));

    float *d_v;
    cutilSafeCall(cudaMalloc((void**)&d_v, columnSize*sizeof(float)));
    cutilSafeCall(cudaMemcpy(d_v, v, columnSize*sizeof(float), cudaMemcpyHostToDevice));

    float *d_r;
    cutilSafeCall(cudaMalloc((void**)&d_r, rowSize*sizeof(float)));
for(int i = 0; i < 1000; i++){
    dim3 threads(32, 8);
    matrixMultiVectorWarp<<<(rowSize+threads.y-1)/threads.y, threads,
2*sizeof(float)*threads.x*threads.y>>>(rowSize, columnSize, pitch/sizeof(float), d_matrix, d_v,
d_r);
    cutilSafeCall(cudaThreadSynchronize());
}
    cutilSafeCall(cudaMemcpy(r, d_r, rowSize*sizeof(float), cudaMemcpyDeviceToHost));

    cutilSafeCall(cudaFree(d_r));
    cutilSafeCall(cudaFree(d_v));
    cutilSafeCall(cudaFree(d_matrix));
}
```

不知道这些代码给了你灵感没有，我第一次设计这些的代码的时候，非常的兴奋，在 GTX295 上的测试表明在每行元素比较多的话，warp 模式没有优势，而在每行元素比较少的时候，warp 可比 block 模式快一倍以上。如果你知道更好的计算方式，请你联系我，我的 qq：304128534

第三节、 线性方程组的求解

本文的线性方程组求解主要采用主元素高斯消元法。其分为三个步骤：一、选主元；二、交换主元素行与当前行；三、消元，消元时也消除对角线上面的元素。

一、选主元：选主元的做法就是对于当前列，先用一个内核将每一个线程块中的绝对值最大的数据求出来并保存其下标，将这两数据存到全局存储器。然后再另开一内核求得最大值所在的位置并保存其下标。

```
//find the main row
static __global__ void mainRowMultiBlock(const int dim, const int currentRow, float *d_matrix, float
*d_matrixTemp, int *d_mainRowTemp){
	const unsigned int id = blockDim.x*blockIdx.x + threadIdx.x;
	extern __shared__ float s_s[];//len 2*blockDim
	float *s_m = s_s;    int      *s_id = (int*)(s_s + blockDim.x);
	float r = 0.0f;  int index = id + currentRow;
	int      rId = 0;

	for(int i = index; i < dim; i += gridDim.x * blockDim.x){
		if( fabsf(d_matrix[i*(dim+1) + currentRow]) > r){
			r = fabsf(d_matrix[i*(dim+1) + currentRow]);
			rId = i;
		}
	}

	s_m[threadIdx.x] = r;
	s_id[threadIdx.x] = rId;
	__syncthreads();

	for(int i = (blockDim.x/2); i > 0; i /= 2){
		if((threadIdx.x < i) && (s_m[threadIdx.x + i] > s_m[threadIdx.x])){
			s_m[threadIdx.x] = s_m[threadIdx.x + i];
			s_id[threadIdx.x] = s_id[i + threadIdx.x];
		}
		__syncthreads();
	}
	if(0 == threadIdx.x){
		d_matrixTemp[blockIdx.x] = s_m[0];
		d_mainRowTemp[blockIdx.x] = s_id[0];
	}
}
static __global__ void mainRowOneBlock(int len, float *d_matrixTemp, int *d_mainRowTemp){
	const unsigned int tid = threadIdx.x;
	extern __shared__ float s_s[];
	float *s_m = s_s;
	int      *s_id = (int*)(s_s + blockDim.x);
	float r = 0.0f;
	int rId = 0;
	for(int i = tid; i < len; i += blockDim.x){
		if(r < d_matrixTemp[i]){
			r = d_matrixTemp[i];
			rId = d_mainRowTemp[i];
		}
	}
	s_m[tid] = r;
	s_id[tid] = rId;
//	  __syncthreads();
	for(int i = (blockDim.x>>1); i > 0; i >>= 1){
		if(tid < i && s_m[tid + i] > s_m[tid]){
			s_m[tid] = s_m[tid + i];
			s_id[tid] = s_id[tid + i];
		}
//		  __syncthreads();
	}
	if(0 == tid){
		d_mainRowTemp[0] = s_id[0];
	}
}
```

二、交换主元素行，代码如下：

```
//swap the main row and current row
static __global__ void swapMain(const int dim, const int currentRow, const int *mainRow, float
*d_matrix){
	const unsigned int id = blockDim.x*blockIdx.x + threadIdx.x;
	int row = *mainRow;
	for(int i = id + currentRow; i < dim+1; i += blockDim.x*gridDim.x){
		float temp = d_matrix[currentRow*(dim + 1) + i];
		d_matrix[currentRow*(dim + 1) + i] = d_matrix[row*(dim + 1) + i];
		d_matrix[row*(dim + 1) + i] = temp;
	}
}
```

三、消元，代码如下：

```
static __global__ void reduceRow(const int dim, const int currentRow, float *d_matrix){
	const unsigned bid = blockIdx.x;
	const unsigned tid = threadIdx.x;
	if(bid == currentRow)
		return;
	__shared__ float fac;
	fac = d_matrix[bid*(dim+1) + currentRow]/d_matrix[currentRow*(dim+1) + currentRow];
	__syncthreads();

	for(int k = tid + currentRow; k < dim+1; k += blockDim.x){
		d_matrix[bid*(dim+1) + k] -= d_matrix[currentRow*(dim+1) + k]*fac;
	}
}
```

主机端代码如下：

```
void solveFuncGPU(const int dim, float *d_matrix){
	unsigned int grid = 30*4;	unsigned int block;
	int *d_mainRowTemp;
	cutilSafeCall(cudaMalloc((void**)&d_mainRowTemp, grid*sizeof(int)));
	float *d_matrixTemp;
	cutilSafeCall(cudaMalloc((void**)&d_matrixTemp, grid*sizeof(float)));

	for(int i = 0; i < dim; i++){
		block = 256;
		if(dim - i < 256)
			block = 32;

		mainRowMultiBlock<<<grid, block, 2*block*sizeof(int)>>>(dim, i, d_matrix,
d_matrixTemp, d_mainRowTemp);
		mainRowOneBlock<<<1, 32, 64*sizeof(int)>>>(grid, d_matrixTemp,
d_mainRowTemp);
		swapMain<<<grid, block>>>(dim, i, d_mainRowTemp, d_matrix);
		reduceRow<<<dim, block>>>(dim , i, d_matrix);
	}

	cutilSafeCall(cudaFree(d_matrixTemp));
	cutilSafeCall(cudaFree(d_mainRowTemp));
}
```

很明显的可以看出来，这个程序中最消耗时间的应该是消元，而消元的内核消耗依旧有优化的余地，原因在于对全局存储器的访问是合并的，但是并不完美（每次访问的首地址可能没有对齐），想到这一点，你应该能够想到优化的办法了，优化后性能提高了差不多一倍。