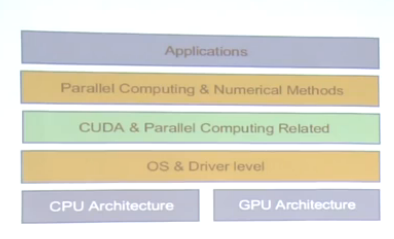
参考书籍：

CUDA C Programming Guide, CUDA官方手册.

CUDA Best Pracice Guide, 优化经验、方法和手段.

Programing Massively Parallel Processors.



1. CPU体系架构概述

CPU是执行指令，处理数据的器件->完成逻辑和算术指令。

并包括内存接口，外部设备接口。

对于编译好的程序，最优化目标：，

cycles是周期数、instruction是指令、cycle是周期、seconds是秒。

第一部分是一个指令需要多少个时钟周期。.

第二部分是一个时钟周期需要多少秒。

桌面应用：

1）轻量级进程，少量线程

2）大量分支和交互

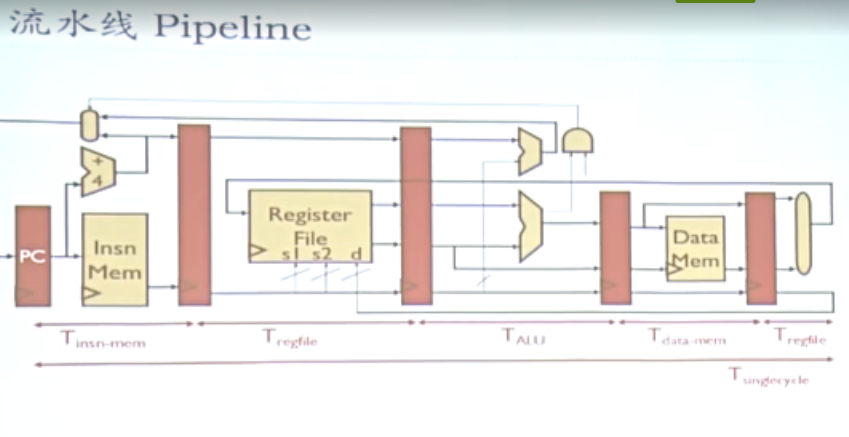
3）大量的存储器访问

4）真正用于数值运算的指令很少

1,流水线

CPU处理过程：

取址🡪译码🡪执行🡪访存🡪写回



利用指令级并行🡪极大的减少时钟周期，增加一些延迟和芯片面积。

带来的问题：具有依赖关系的指令怎么办？（后一条指令要利用前一条指令的结果，但前一条指令还没处理完），分支如何处理，流水线长度。

2.分支预测

基于过去的分支记录，猜测下一条指令。

现代分支预测期准确度>90%

带来面积增加、延迟增加。

分支断定：就是(statement ? x1 : x2)，x1和x2都计算，然后从中选择。

3.超标量

增加流水线宽度，提升IPC(instructions/cycle，一个周期的指令数)。常规的IPC=1。

峰值IPC为N（N路超标量）

会增加面积，N倍资源使用，旁路网络N^2, 需要更多的寄存器和存储器宽度。

4.乱序执行

Out-of-Order(OoO) Execution

重排指令，获得最大的吞吐率。减少不合理的开销，减少不合理。

重拍缓冲区。

发射队列/调度器。

5.存储器层次

存储器越大越慢。

SRAM(L1 L2 L3)

DRAM(memory)

Flash(disk)

HDD(disk)

缓存，将数据尽可能解决的位置。

利用：时间的临近新、空间的临近性。

硬件管理： L1 Instruction/Data caches

L2 unifield cache

L3 share unifield cache

软件管理: Memory

Disk

6.矢量操作

For(int i=0; i<N; i++)

A[i] = B[i] + C[i];

🡪

For(int i=0; i<N; i+=4)

{

A[i+0] = B[i+0] + C[i+\0];

A[i+1] = B[i+1] + C[i+1];

A[i+2] = B[i+2] + C[i+2];

A[i+3] = B[i+3] + C[i+3];

}

7.多核处理

流水线完整复制

完整的核，除了最后一级缓存外，不共享其他资源

问题: 1.读写同一块数据，加锁

2.谁的数据是正确的，缓存一致性协议

3.什么样的数据是正确的，存储器一致性

能量墙、存储器墙。

CPU内部的并行性：

指令级并行：超标量、乱序执行、流水线

数据级并行：矢量计算

线程级并行：同步多线程、多核

1. 并行程序设计概述

并行计算是同时应用多个计算资源解决一个计算问题。

问题呗分解为多个离散的部分，可以同时处理。

每个部分由一些指令完成。

矩阵

SISD， 单指令但数据-----串行模式

SIMD， 单指令多数据-----同一条指令用在多个数据

MISD， 多指令单数据-----多个指令操作在同一个数据

MIMD， 多指令多数据

常见名词:

Task(任务)，完整得到结果的一段程序。

Parallel Task(任务)，任务由并行计算解决。

Serial Execution(串行执行)，单一流解决任务。

Parallel Executioin(并行执行)，多个流解决任务。

Shared Memory(共享存储)，多个流的数据共享。

Distributed Memory(分布式存储)，数据是放在不同的位置。

Communicatioins(通信)，任务之间的。

Synchronizatioin(同步)，并行指令段，要保证一致性，但是会破坏并行性。

Granularity(粒度)，每个任务的大小

Observed Speedup(加速比)，并行系统和串行系统相比的性能提升倍数。

Parallel overhead(并行开销)，为了完成并行计算的成本，比如同步，通信。

Scalability(可扩展性)

存储器架构

Shared Memory

Distributed Memory，数据分布在若干位置

Hybrid Distributed-shared Memory

并行编程模型

共享存储模型

线程模型

消息传递模型

数据并行模型

设计并行处理程序和系统

自动和手动并行

理解问题和程序

分块分割。数据的分块、任务的分割。

通信。不同的分块和分割间，进行交互。

同步

数据依赖

负载均衡，多个独立任务间的负载是否平衡。

粒度

I/O

成本

性能分析和优化

加速比

,p是程序可以被并行化的部分(百分比)，若程序每条指令都可以并行化，p=1.0, speedup=无穷，若都不能并行化,p=0,speedup=1。

这个指标也指明了并行化可扩展的上限。

1. CUDA开发环境搭建和工具配置

常用的GPU开发，一般有两个独立GPU，一个作显示，一个作运算，这样系统流畅些。

1. GPU体系结构概述

CUDA Sample，提供多个CUDA开发的例程。

CUDA Tookit，开发工具、编译器。

1. GPU体系架构概述

FLOPS ---每秒进行的浮点数的运算

GFLOPS ----每秒的十亿条浮点运算

TFLOPS -----1k GFLOPS

为什么需要GPU？

应用需求越来越高。计算机技术由应用驱动。

GPU是一个异构多处理器芯片，为图形图像处理优化。

CPU型的内核，内部存在太多和数据处理无关的部分，比如分支预测，存储部分。

🡪优化1, 精简，减肥。找到最核心的部分。可以将节约的空间用来设计更多的核，这多个核应该共享指令流，若做的非共享指令流，则又会让芯片臃肿。

🡪优化2，增加ALU宽度，SIMD， 一般用8个ALU。

最后会形成128矩阵，有16个核，每个核有个ALU。

对于if-else的分支情况，其中3个ALU的结果是True，剩余的5个的ALU是false，由于指令共享，因此在执行True的时候，那5个False会一直等待。

三种方法提升GPU的处理速度

1. 使用大量精简的核心。
2. 每个核心里面塞满ALU。
3. 不停的干活。

GPU的存储设计

GPU内部放的全是计算能力的核，存储器没有在GPU内部，而是在外部。带宽是非常宝贵的资源，为了让数据处理流畅。否则数据处理快，但获取数据速度慢，没有意义。

尽量减少不必要的数据请求。

将读取的频率减少，将零散的数据打包在一起。

1. GPU编程模型

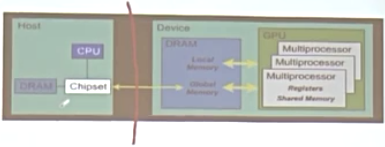
SM流式多处理器，是SMIT的。

CPU和GPU的交互

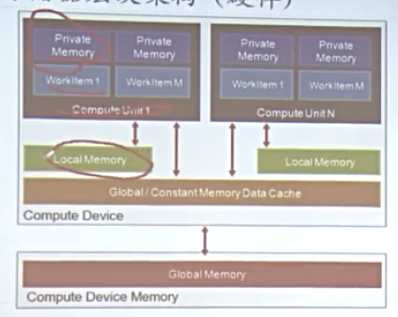
各自的物理内存空间，相互独立。

通过PCIE总线(8GB/s~16GB/s)

交互开销较大



最左边的块是CPU及DRAM，右边是GPU。其中中间的块是GPU的显存。



Compute Unit的Privvate Memory主要是寄存器。

访存速度：

Register，寄存器，专门的硬件支撑，单周期，线程私有。

Shared Memory，有专门的硬件支撑，单周期。

Local Memory，显存的一部分，寄存器放不了，是线程私有的。Slow

Global Memory，显存的一部分。Slow

Constant Memory, 显存的一部分，比较快，GPU不能修改，CPU可以修改。

Texture Memory,线程的一部分。

Instruction Memory,显存的一部分，指令的存储单元，是不可见。

GPU线程组织模型

多个thread组织为block，多个block组织为Grid。

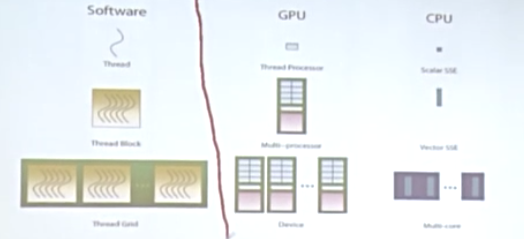
一个kernel具有大量的线程，

线程被划分成线程块”blocks”，一个block内部的线程共享shared Memory，可以同步。

Kernel启动一个grid，包含若干线程块，由用户设定。

线程和线程块具有唯一标识。

GPU线程映射关系



GPU存储模型

CPU内存和线程关系：

Thread，线程私有的内存是Local Memroy和寄存器

Block,共享Shared Memory。

Kernel，共享Global Memory。

编程模型

常规意义的gpu用于处理图形图像

操作与像素，每个像素操作都类似

可以应用于SIMD，并且这也是非常天然的，设计初衷。

SIMD，可以认为是数据并行分割。

SIMT

GPU版本的SIMD

是一种线程模型，大量想出模型获得高度并行。

线程切换获得延迟掩藏。

多个线程执行相同指令流。

GPU上大量线程承载和调度。

CUDA编程模式：Extended C

GPU变量需要加修饰词。（没有加修饰词，认为是放在CPU上的。）

关键词

运行期API

函数调用

|  |  |  |
| --- | --- | --- |
| Cuda函数声明 | 执行位置 | 调用位置 |
| \_\_device\_\_ float DeviceFunc() | Device | Device |
| \_\_global\_\_ void KernelFunc() | Device | Host |
| \_\_host\_\_ float HostFunc() | Host | Host |

\_\_global\_\_定义一个kernel函数，入口函数，cpu上执行，必须返回void

\_\_device\_\_ and \_\_host\_\_可以同时使用，在设备端和主机端代码一样。也是这个原因，才额外弄了一个\_\_host\_\_的声明。

1. CUDA编程 1

适用于：

GPU特别适用于密集计算、高度可并行计算，图形学。而不是缓存数据，控制指令流。

CPU在计算方面不如GPU，因为CPU在缓存和控制逻辑上占了很大的面积，GPU刚好相反。

历史：

2001/2002-研究人员把GPU当作数据并行协处理器。

2007年-NVIDIA发布CUDA(Compute Uniform Device Architecture)即统一计算设备架构。把gpu可以进行通用计算的平台。发展成为GPU computing。

2008 发布opencl规范。

CUDA术语:

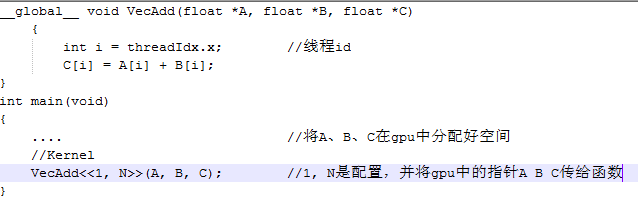
Host – 即主机端，通常指cpu，采用c编程。

Device-即设备端，通常指gpu（设备端的代码是数据并行化的代码采用c扩展编程，就是cuda c）

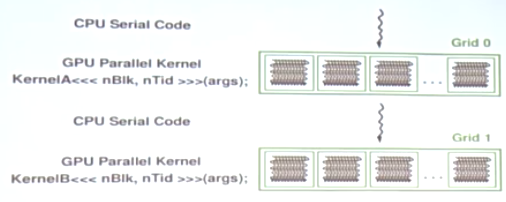
Host Device拥有各自的存储器

CUDA编程，包括主机端和设备端两部分的代码。

Kernel-数据并行处理函数，通过调用kernel函数，在设备端创建轻量级线程，线程由硬件负责创建并调度。



CUDA程序的执行：



首先是cpu串行代码，接着遇到kernel时候启动gpu并行完成工作，然后返回结果到cpu，cpu接着串行运行。

CUDA线程的层次

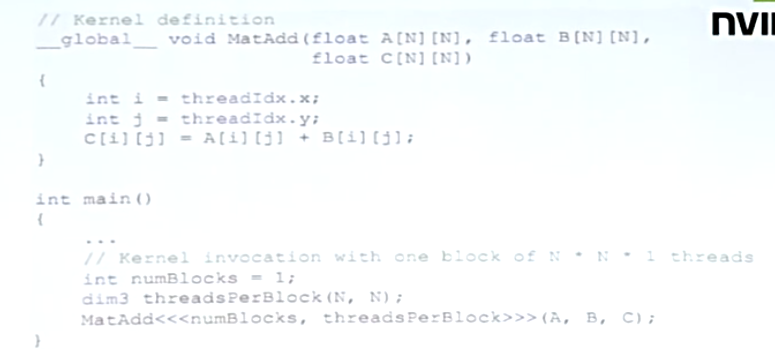
1. Grid, 一维或二维线程块(Blcok)。不同的组的线程不能合作。
2. Block, 一组线程，线程和线程间可以相互协作
   1. 一维，二维或三维
   2. 一个grid里面的每个block的线程数是一样的
   3. Block内部的每个线程可以
      1. 同步 synchronize。
      2. 访问共享存储器shared memory。

线程索引：threadIdx，在blocks中的编号以及grid中的编号。

一维block: Thread ID == Thread Index

二维block (Dx, Dy): (x, y)的索引是== x + y\*Dx;

三维block (Dx, Dy, Dz): (x, y, z)的索引是 == x + y\*Dx + z\*DxDy;



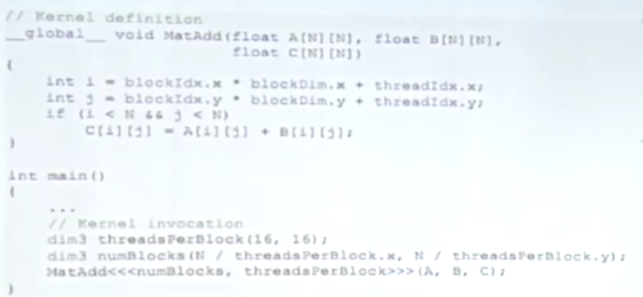
每个block中的线程数量是特定的:比如G80的 block中最多512个线程 。Fermi的block中有最多1024线程

线程块中的线程:

1.位于相同的处理器核中。(GPU里面是多核的，每个核里面又有多个ALU，一个ALU就可以干一个线程)

2.共享所在核的存储器。

块索引: blockIdx



🡺线程索引threadIdx: (threadIdx.x, threadIdx.y, threadIdx.z)

🡺块索引bolckIdx: blockIdx.x, blockIdx.y,blockIdx.z)

线程块之间彼此独立执行：任意顺序，被任意数量的处理器以任意顺序调度。具体来说，由于核的总数有限，块数量往往大于核数，因此，会以选择任意两个块在核心中执行代码，执行完成以后，再换成另外两个块。

一个块内部的线程：

共享容量有限的低延迟存储器

同步执行

合并访存

\_\_syncThreads()🡪块内线程一起等待所有线程执行到该处。轻量级。

内存传输：

Device code can:

R/W 线程自己的寄存器。

R/W 线程自己的local memory

R/W 所在块的shared memory

R/W grid的global memory

Read only grid的constant memory

Host code can

R/W gird的global memory以及constant memory.，这两部分都是在gpu芯里的，是显存。

Host可以从device往返传输数据:

Global memory and constant memory

cudaMalloc()，在设备端分配global memory。

cudaFree()，释放存储空间。

cudaMalloc((void \*\*) &Md, size); Md的地址是在设备端的，只能在设备端引用，不能在主机端直接引用。

CudaMemcpy(),

Host to host

Host to device

Device to host

Device to device

如 cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

CUDA算法框架：

1.在gpu上分配空间，将数据从cpu拷贝到gpu中。

2.在gpu上启动并行kernel。

3.将结果拷贝回到host。

八、CUDA编程 2

内置类型和函数

\_\_global\_\_, 声明kernel，返回值必须是void。

\_\_device\_\_,曾经默认是内联，现在不是。

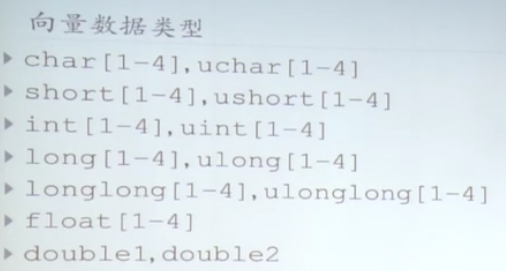
Global和device函数

尽量少用递归(不鼓励)。

不要用静态变量。

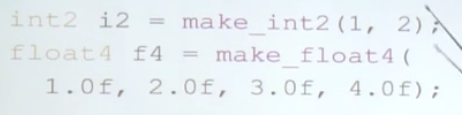
少用malloc(运行不鼓励)。

小心通过指针实现的函数调用。



同时使用host device代码

通过函数make\_<type name>构造



通过.x .y .z .w访问

部分函数列表

Sqrt rsqrt

Exp

内建函数

仅面向Device设备端

更快，但精度降低

以\_\_为前缀:\_\_log, \_\_sin, \_\_pow

线程同步

块内线程可以同步

1).调用\_\_syncthreads();创建一个barrier栅栏。

2).每个线程在调用点等等块内所有线程执行到这个地方，然后所有线程继续执行后续指令

3)要求线程的执行时间尽量接近，如果有拖后腿的线程，就会效率低下。

4)全局同步消耗太大，因此没有这么设计。

5)同步会导致线程的暂停，甚至死锁(都挂起，但是挂起的位置不同)。

线程调度

线程数远远大于硬件的执行单元数

G80

16个SM (流式多核处理器)

每个sm含8个sp(流式处理器)

每个sm可以驻扎768个线程

在同一个时钟周期执行128个，但是有16\*768个线程可以被调度。

Warp 块内的一组线程

1).32个线程为1个warp

2).运行于同一个SM

3).线程调度的基本单位

4).threadIdx连续

5).调度0开销

6).在一个sm上，一个时刻，只会有一个warp在执行。

7).Warp是一直同步的！指令的位置必须一样，对于同一个warp内有分支情况，就会出现等待情况

8).每个warp有32个线程，但每个sm只有8个sp，那就轮流执行，8个线程先执行1次，再换下一个8个执行一次，因此wrap中所有线程执行一条指令，需要4个周期。现代情况，sm的sp个数以及比wrap多了。

存储模型

全都是在gpu上的，不是显存就是寄存器。

寄存器

每个线程专用。

快速，片上，可读写。

对于G80，每个SM上多大768线程，有8k寄存器，没给线程8k/768

Local Memory

线程私有

存储于global memory

用于存储局部变量数组

Shared Memory

块内线程共享，位于cache

快速，可读写

Global Memory

长延时

片外显存

随机访问影响性能

Host可读写

Constant Memory

短延时，高带宽，只读

存储于global memory，但是有cache

Host主机可读写

容量 64kB

|  |  |  |  |
| --- | --- | --- | --- |
| 变量声明 | 存储器 | 作用域 | 生命期 |
| 非数组的自动变量 | 寄存器 | Thread | Kernel |
| 数组的自动变量 | Local | Thread | Kernel |
| \_\_shared\_\_ int sharedVar; | Shared | Block | Kernel |
| \_\_device\_\_ int globalVar; | Global | Grid | Application |
| \_\_constant\_\_ int constantVar | Constant | Grid | Application |

Global和constant

Host可以通过以下函数访问

cudaGetSymbolAddress

cudaGetSymbolSize

cudaMemcpyToSymbol

cudaMemcpyFromSymbol

Constants必须在函数外声明

重访

原子函数

1. CUDA编程 3
2. 去除长度限制
   1. 这也是引入grid(多个block)的原因，每个block可以申请的线程数是有长度限制的，而block的数量是不会首限的，gpu会自动调度block。
   2. 若要申请的线程数会很大，可以将整个任务划分为多个block。
   3. dim3 是一个三维数组。用了多少个就表示的多少维。

dim3 dimGrid(size\_x, size\_y, size\_z); //每个grid的block数，可三维

dim3 dimBlock(size\_x, size\_y, size\_z); //每个block的线程数，可三维

kernelFunc<<<dimGrid, dimBlock>>>(param);

1. global memory的读写较慢问题
   1. gpu的速度将受限于global memory带宽。

若GFLOPS:346.5(1秒处理356.5G的float数据，即1386GB/s的数据处理速度)，而存储器实际宽度只有86.4GB/s

* 1. 用shared memory来减轻读写慢问题。

1. 原子操作

针对于某个特定的操作数，通过原子操作，只有一个线程获得操作，当这个操作完成后其他线程才能操作。尽量少用原子操作，太过于耗时，因为一旦用了，就会容易让很多线程排队。

十、CUDA程序分析和调试工具

十一、CUDA基本优化

有效的数据并行算法 + 针对gpu架构特性的优化 = 最优性能

1. 并行规约(Palallel Reduction)

规约，就是把一组很大的数据，进行综合性的运算，得到较少的数据。

1. Warp分割

块内线程划分warp

1).Block被划分为32为单位的线程组，叫做war。相当于部门里面的小组。是连续的32个线程为一组。

2).Warp是最基本的调度单元。

3).Warp是一直执行相同的指令。

4).设备切换没有时间代价

5).许多warp在一起可以隐藏访存延时。

减少分支发散。

分支发散是指的，由于一个warp的线程要求指令相同，因此遇到分支语句，不同的线程将处理不同的指令，处理不同指令的时候，线程将等待。

1. 访存合并

相邻的线程操作相邻的存储位置。

1. Sm资源动态分割
2. 数据预读
3. 指令混合
4. 循环展开

十二、CUDA程序深入优化

十三、最新NVIDIA GUP和CPU特性