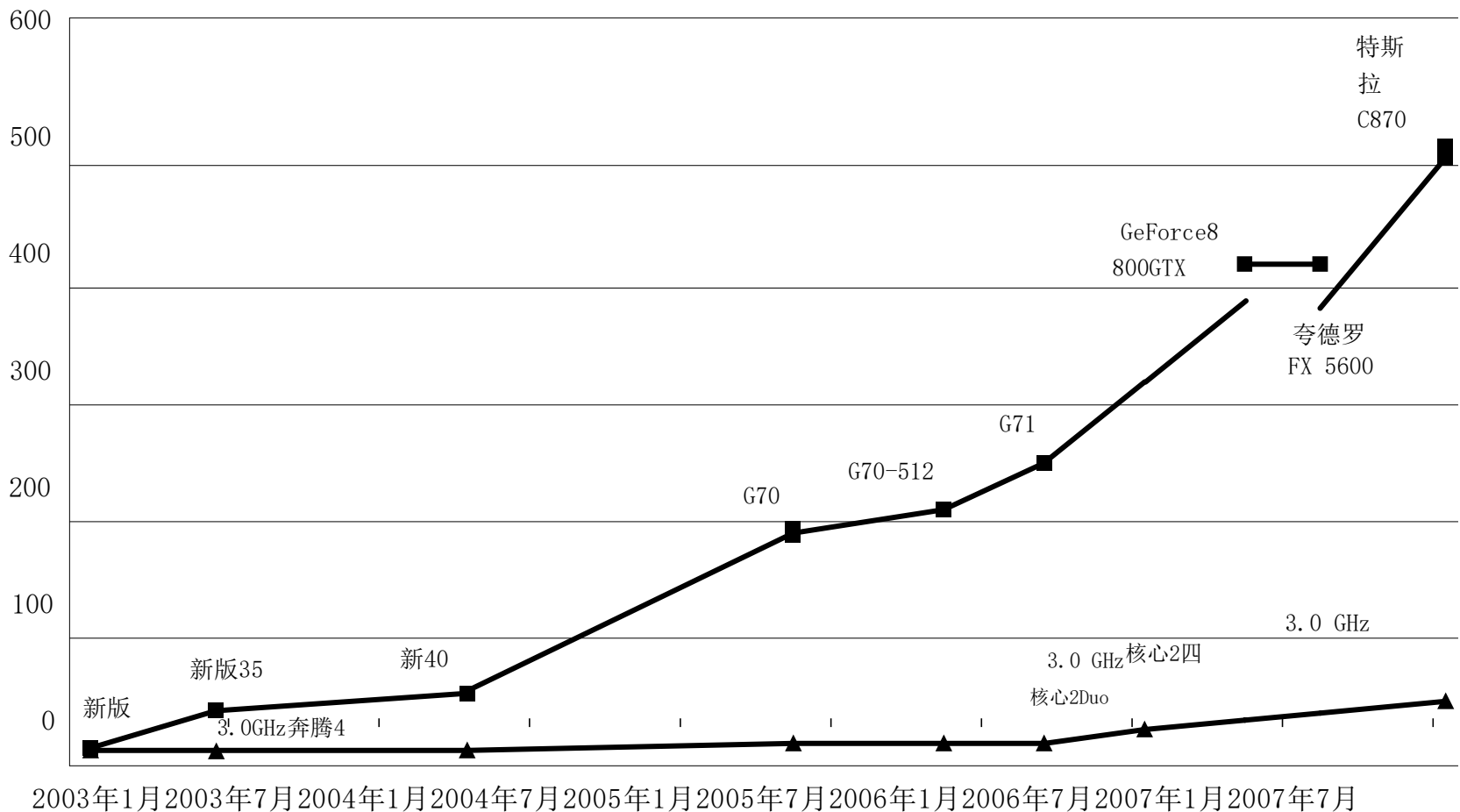


台湾2008CUDA课程

大规模并行处理器编程：CUDA体验

讲座1 介绍和动机

是什么推动了许多人- 果核



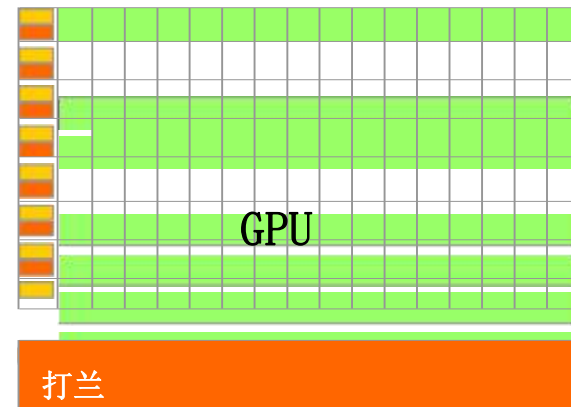
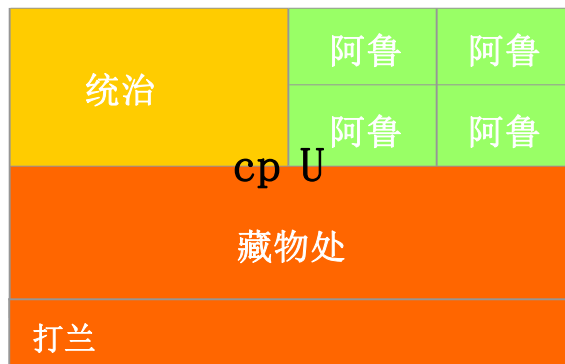
¹根据S的幻灯片7。绿色，“GPU物理”，签名图2007年GPGPU课程。<http://www.gpgpu.org/s2007/slides/15-GPGPU-physics.pdf>

©DavidKirk/NVIDIA和文美W。吴
台湾，2008年6月30日-7月2日

设计理念是不同的。

GPU专门用于计算密集型、大规模数据并行计算（图形渲染的内容）

-因此，更多的晶体管可以用于数据处理，而不是数据缓存和流控制



快速增长的电子游戏产业对不断的创新施加了强大的经济压力

这不是你的顾问的平行关系 计算机

比单处理器执行有显著的应用程序级加速

- 不再有“微杀手”

国桂轻松进场

- 一个初始的，幼稚的代码通常可以得到至少2-3倍的加速

对最终用户的广泛可用性

- 可用于笔记本电脑、台式机、集群、超级电脑

数值精度和精度

- IEEE浮点式和双精度式

©DavidKirk/NVIDIA和文美W。吴

台湾，2008年6月30日-7月2日

国墙强大的规模路线图

GPU计算缩放

笔记本电脑，台式机，工作站，服务器，集群-（手机？ipod吗？）

UIUC已经建立了一个有16个节点的GPU集群

- 峰值性能32.5TFLOPS (SP)
- 为科学和工程应用程序

UIUC正在计划建立一个包含32个节点的GPU集群2008年夏季**特斯拉D870**

- 估计峰值性能为130TFLOPS (SP) 和16TFLOPS (DP)

UIUC将在2010年设计一个拥有1000个节点的GPU集群

- 预计峰值性能为4PFLOPS (SP) 和400TFLOPS (DP)



GeForce8800



特斯拉S870

你计算的能力是多少 足够

计算能力每增加10倍，就会激发出新的计算方法

- 许多应用程序都有由于计算能力的限制而引起的近似值或遗漏
- 性能每提高10倍，就可以让应用程序开发者重新思考他们的基本假设和策略
- 例如：图形、医学成像、物理模拟等。

每个2-3X都允许为应用程序添加新的、创新的功能

历史GPGPU运动

在三维图形以外的应用中使用GPU进行通用计算

- GPU加速了应用程序的关键路径

数据并行算法利用GPU属性

- 大型数据阵列、流媒体吞吐量

- 细粒SIMD平行度

- 低延迟浮点数 (FP) 计算



应用程序-请参见//GPGPU.org

- 游戏效果 (FX) 物理学, 图像处理学

- 物理建模, 计算工程, 矩阵代数, 卷积, 相关性, 排序

历史GPGPU约束

处理图形API

- 使用图形API的角案例

撒布解决模式

- 限制纹理尺寸

沙桂树着色器功能

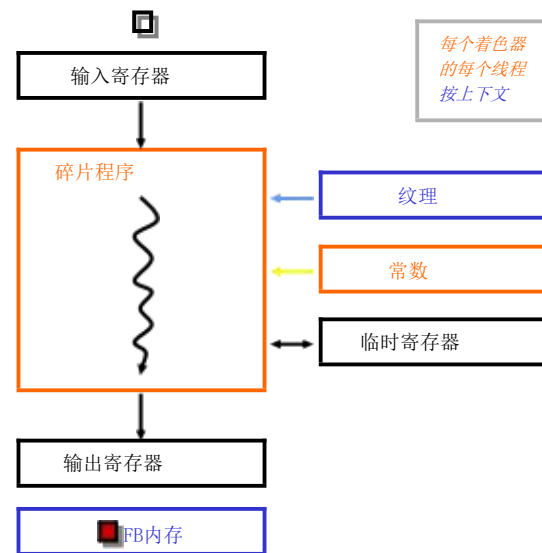
- 有限的输出

《孙子孙女》教学集

- 缺少整数和位操作

国化通信有限公司

- 像素之间没有相互作用
- 无分散存储能力-一个=p



这些都已经改变了，
与CUDA有关！

GPU擅长什么？

GPU擅长于数据并行处理

在许多数据元上并行执行相同的计算-低控制流开销
和高SP浮点运算强度

每次内存访问的多次计算

目前还需要高浮点与整数比

高浮点算术强度和许多数据元素意味着内存访问延迟可以隐藏与计算，而不是大数据缓存-仍然需要避免带宽饱和！

CUDA-不再有着色器功能。

CUDA集成了CPU+GPU应用程序C程序

-串行或适度并行的C代码执行在CPU上

-高度并行的SPMD内核C代码执行在GPU上

CPU串行代码

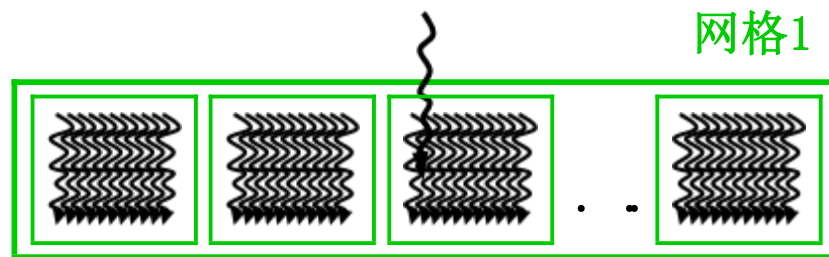
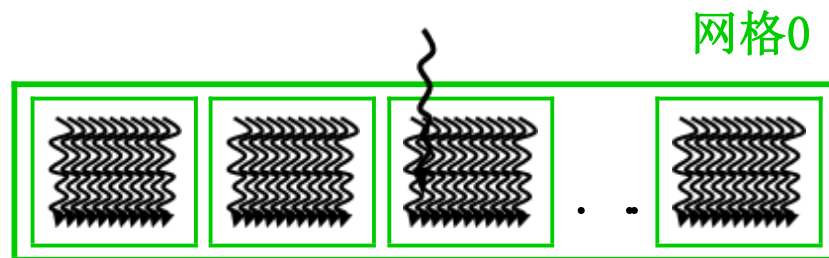
GPU并行内核内核: <<<nBlk

, nTid>>>(args);

CPU串行代码

GPU并行内核

KernelB<<<nBlk, nTid>>>(args);





这是关于 申请

视觉，成像，VACE，HCI，建模和仿真。

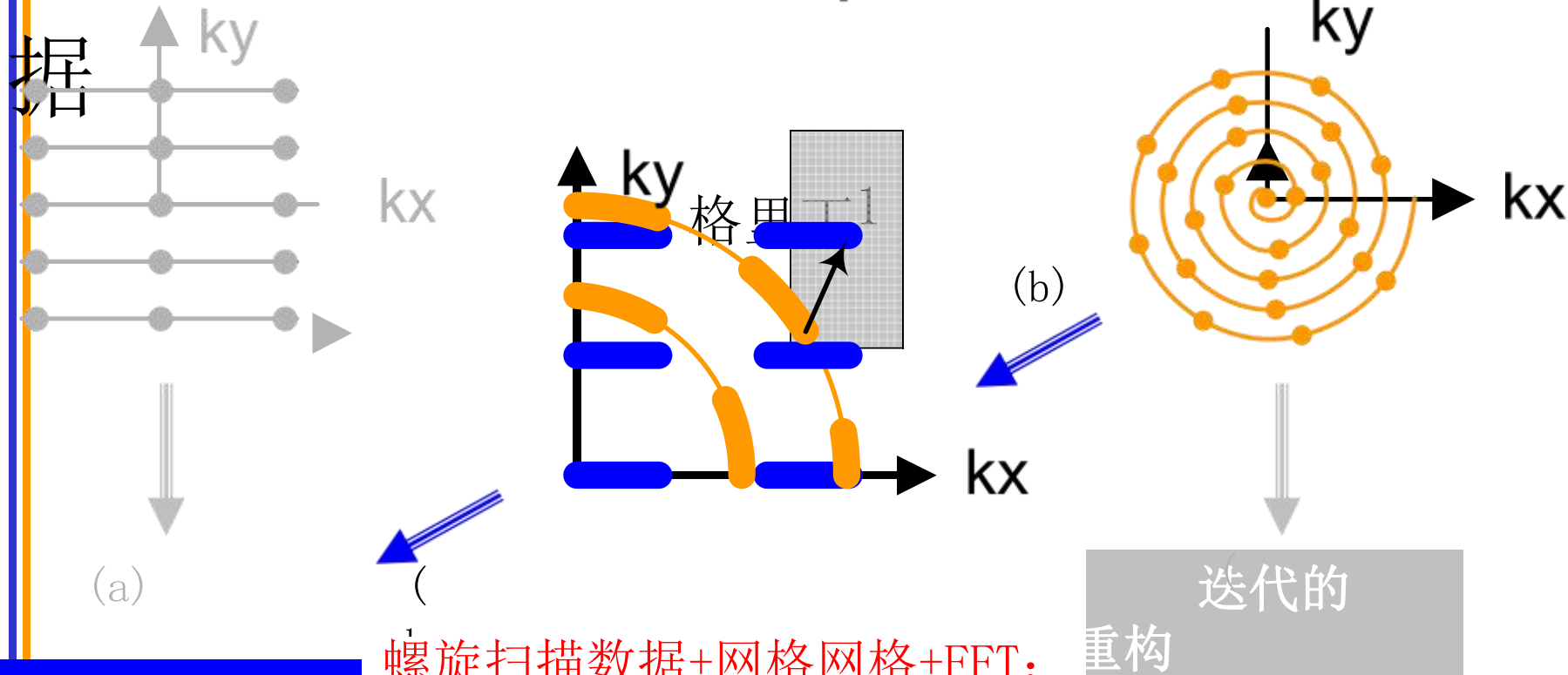
台湾，2008年6月30日-7月2日

科学与工程应用加速

应用程序。	阿奇特。瓶颈，瓶颈	西莫尔特 。 T	内核X	应用程序X
H. 264	寄存器，全局内存延迟	3, 936	20. 2	1. 5
lbm	共享内存容量	3, 200	12. 5	12. 3
RC5-72	寄存器	3, 072	17. 1	11. 0
女子	全局内存带宽	4, 096	11. 0	10. 1
rpe	指令发行率	4, 096	210. 0	79. 4
pns	全局内存容量	2, 048	24. 0	23. 7
线包	全局内存带宽，CPU-GPU数据传输	12, 288	19. 4	11. 8
图谱	共享内存容量	4, 096	60. 2	21. 6
有限差异时 间域	全局内存带宽	1, 365	10. 5	1. 2
核流体动力 学	指令发行率	8, 192	23. 0	23. 0

©[DavKidKir/不v国际刑事法庭伊艾ps-w20-0mci]W. Hwu
台湾，2008年6月30日-7月2日

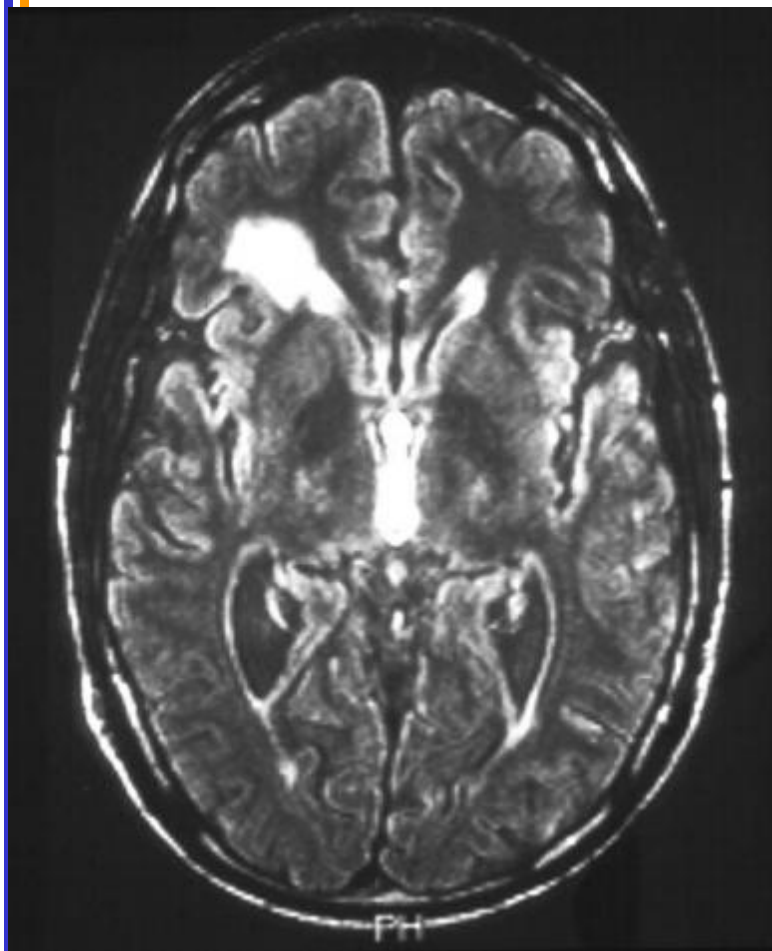
大规模加速可以笛卡尔扫描数据革命性的行动pp西拉1扫描数据



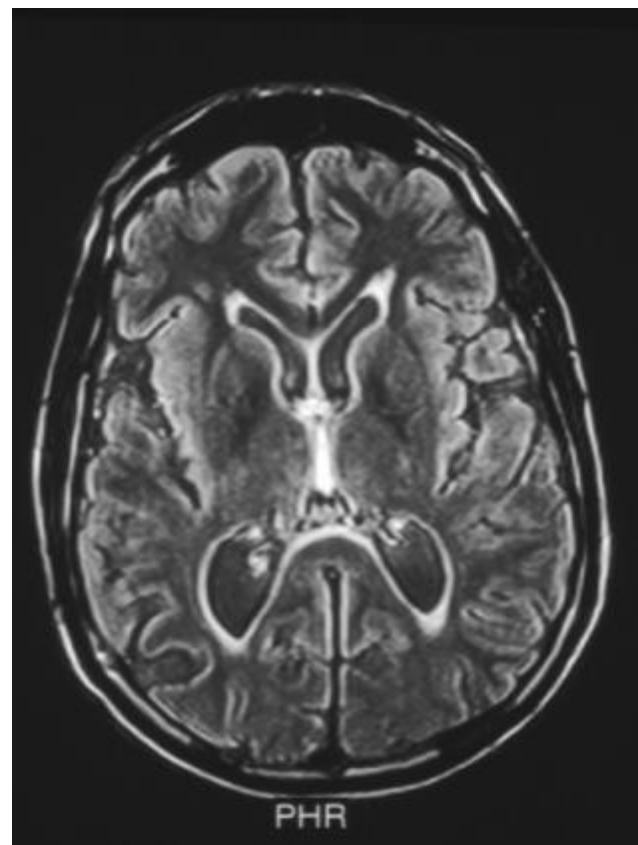
螺旋扫描数据+网格网格+FFT:
更快的扫描减少了伪影, 平均增加信噪比。
重建只需要很少的计算。

¹基于Lustig等图的图1, 紧固螺旋傅里叶变换迭代Mr图像重建, IEEE国际系统。关于生物医学成像, 2004年

化学疗法监测



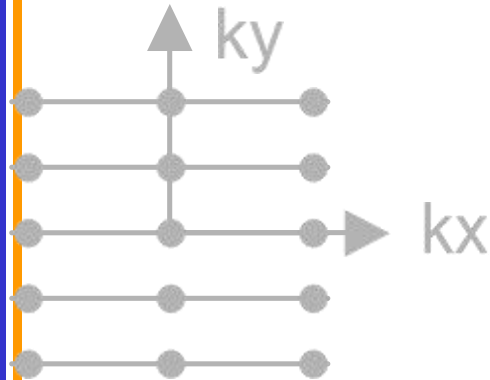
→
6-12周



台湾，2008年6月30日-7月2日

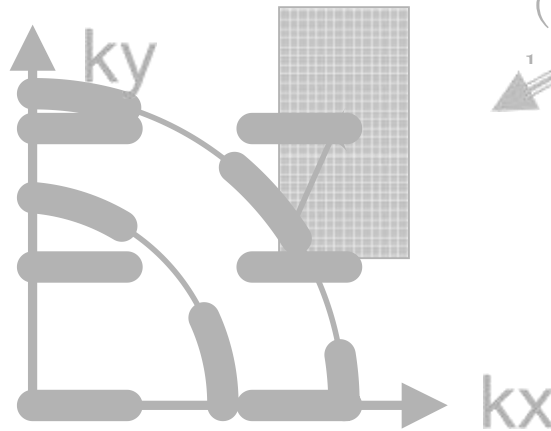
MRI重建

笛卡尔扫描数据

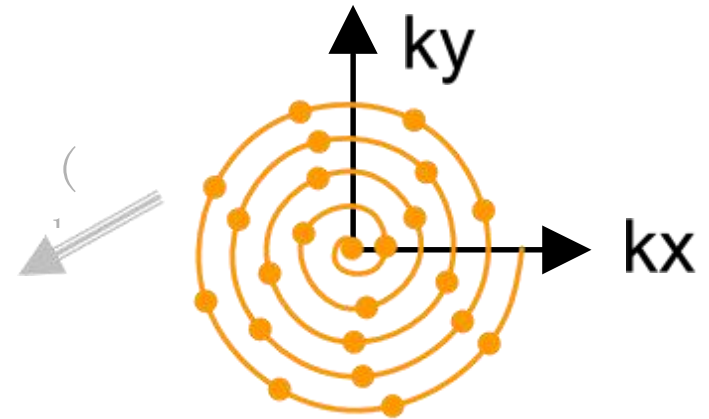


快速傅里叶

格里丁



螺旋扫描数据

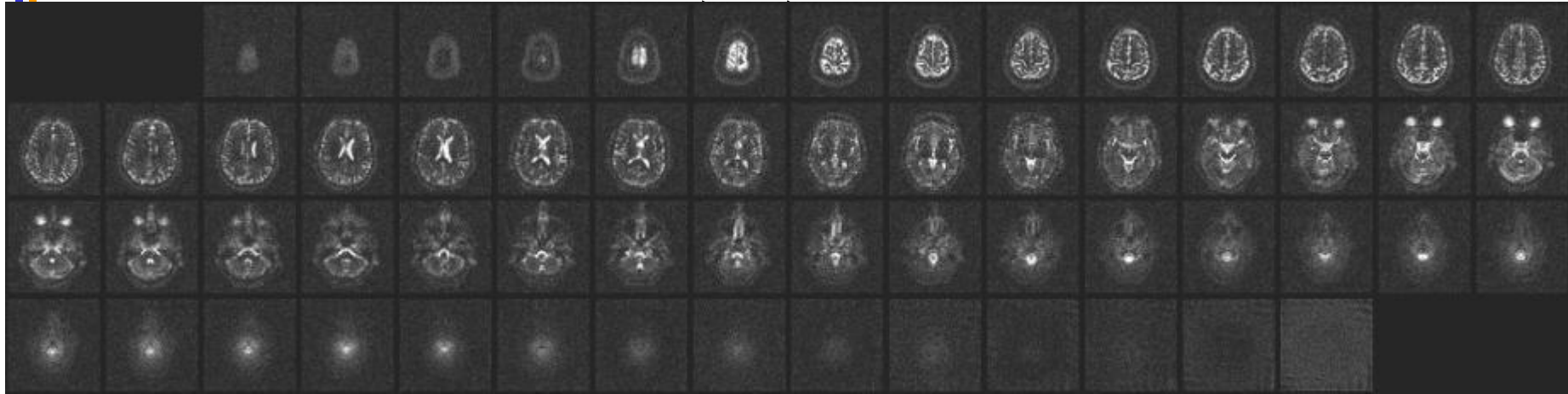


(c)

迭代的
重构

螺旋扫描数据+迭代侦察：
快速扫描减少了伪影，迭代重建增加了信噪比。
重建工作需要大量的计算量。

一个令人兴奋的革命-钠钠的地图



大脑中钠离子的图像

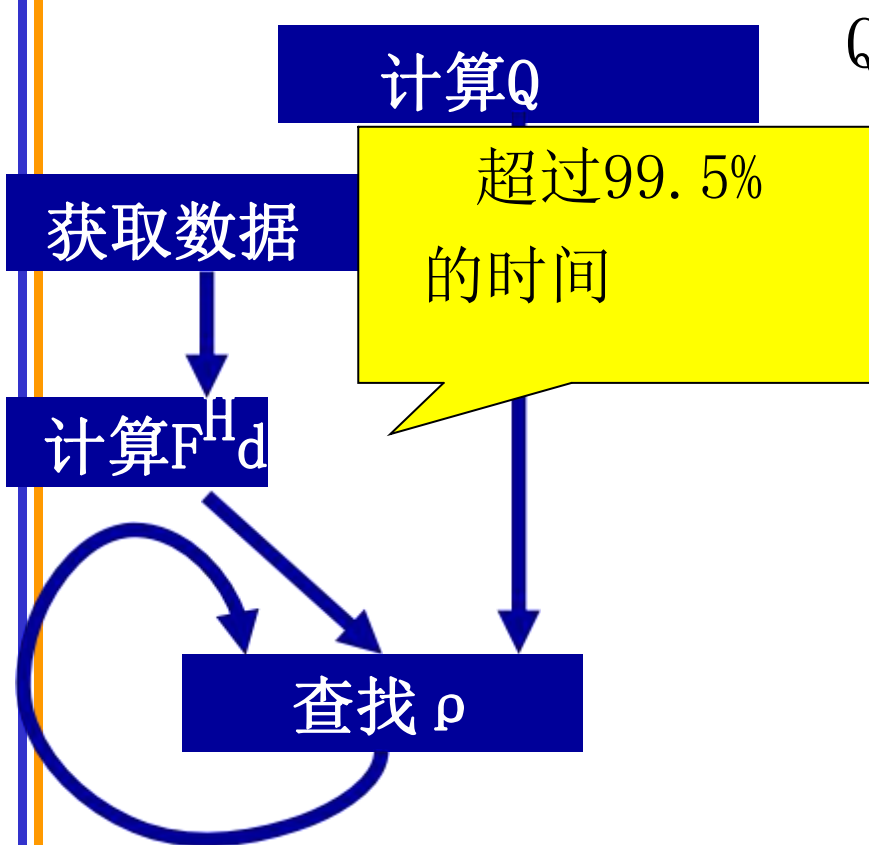
- 需要功能强大的扫描仪（9.4特斯拉）
- 需要大量的样本来增加信噪比
- 需要高质量的重建工作

使在中风和癌症治疗中解剖变化发生前研究脑细胞活力成为可能达古巴 y_{srte} 基思·瑟伯恩和伊恩·阿特金森的sy，伊利诺伊大学芝加哥分校研究中心

©DavidKirk/NVIDIA和文美W。吴
台湾，2008年6月30日-7月2日

高级MRI重建

$$(F^H F + \lambda W^H W) \rho = F^H d$$



Q只依赖于
扫描仪配置

F^H_d依赖于扫描数据

- ρ 发现使用线性求解器
 - F^H_F, 每次迭代计算一次;
取决于Q, F^H_d
 - λ W^HW包含了解剖学上的限制

重建a64³图片过去需要好几天的时间!

Haldar等人，“基于噪声数据的解剖约束重建”，医学博士。©DavidKirk/NVIDIA和文美W

。吴

台湾，2008年6月30日-7月2日

```
为(p=0; p<numP; p++) {  
  为(d=0; d<numD; d++) {  
    exp = 2*PI*(kx[d] * x[p] +  
               ky[d] * y[p] +  
               kz[d] * z[p]);  
    cArg = cos (exp);  
    sArg=sin(exp);  
    rFhD[p] += rRho[d]*cArg -  
              iRho[d]*sArg;  
    iFhD[p] += iRho[d]*cArg +  
              rRho[d]*sArg;  
  }  
}
```

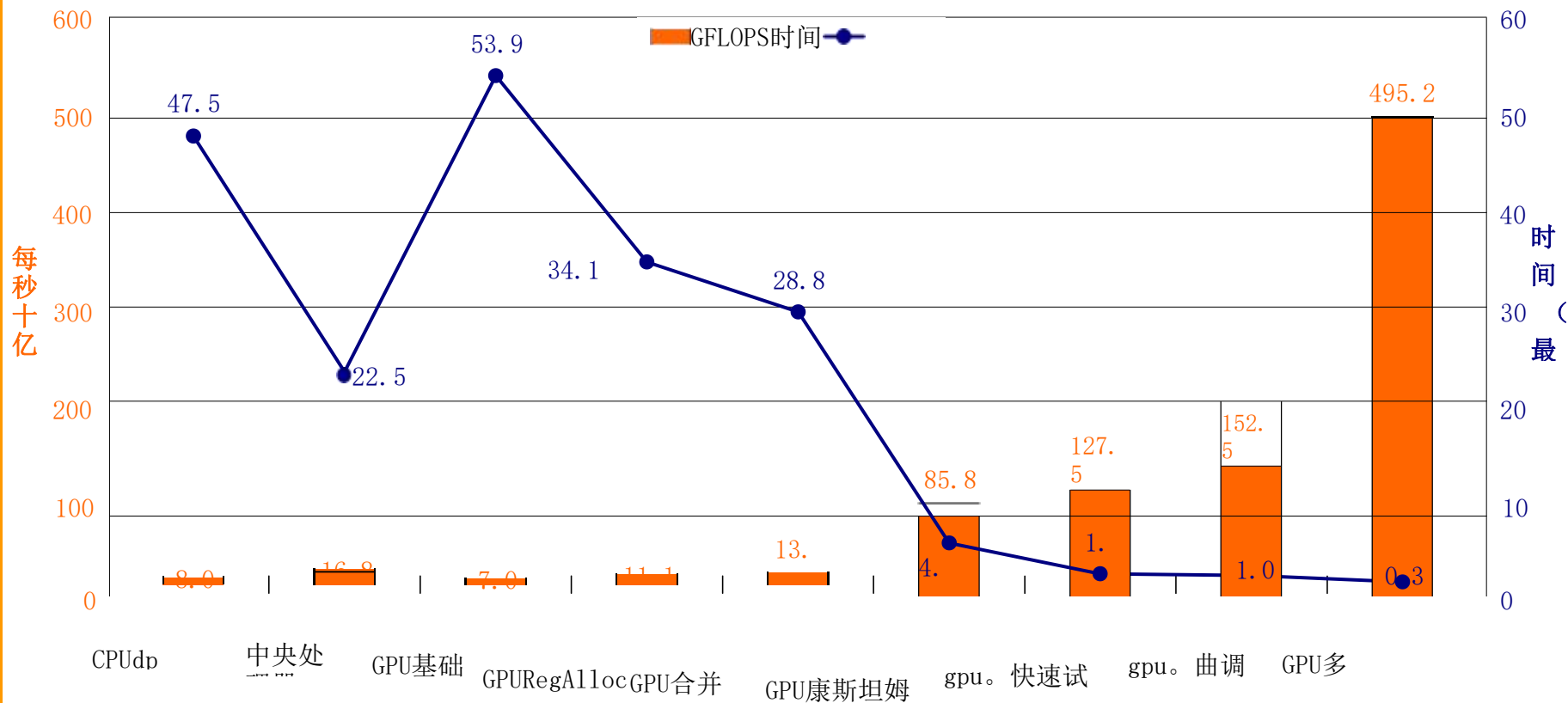

编码

```
__global__ 空白
cmpFhD (float* gx, gy, gz, grFhD, giFhD)
    int p = blockIdx.x*THREADS_PB+线程Idx。

//寄存器分配图像空间的输入和输出
x = gx[p]; y = gy[p]; z = gz[p];
rFhD = grFhD[p]; iFhD = giFhD[p];

为(intd=0; d<SCAN_PTS_PER_TILE; d++) {
    //（扫描数据）保存在恒定的内存中
    浮点式exp=2*PI*(s[d].kx*x+
                                s[d].ky *
                                s[d].kz *
    cArg=公司(exp); sArg=sin(exp);
    rFhD += s[d].rRho*cArg - s[d].iRho*
    iFhD += s[d].iRho*cArg + s[d].rRho*s
}
grFhD[p] = rFhD; giFhD[p] = iFhD;
}
```

FhD计算性能

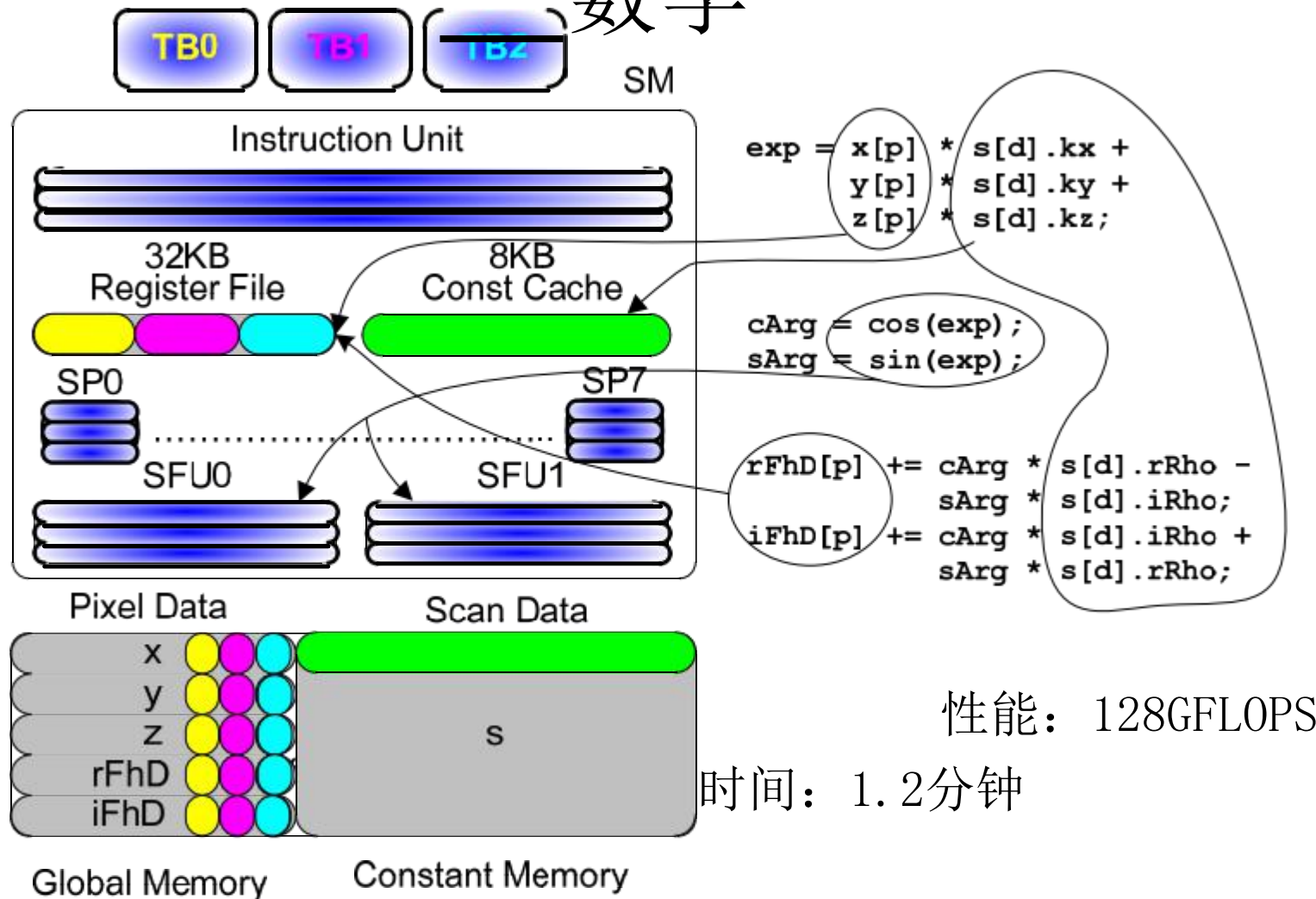


S. S. Stone等人, “使用gpu加速高级MRI重建”, ACM计算前沿会议, 2008年, 意大利, 2008年5月。

©DavidKirk/NVIDIA和文美W。吴
台湾，2008年6月30日-7月2日

最终的数据安排和快速的数据安排

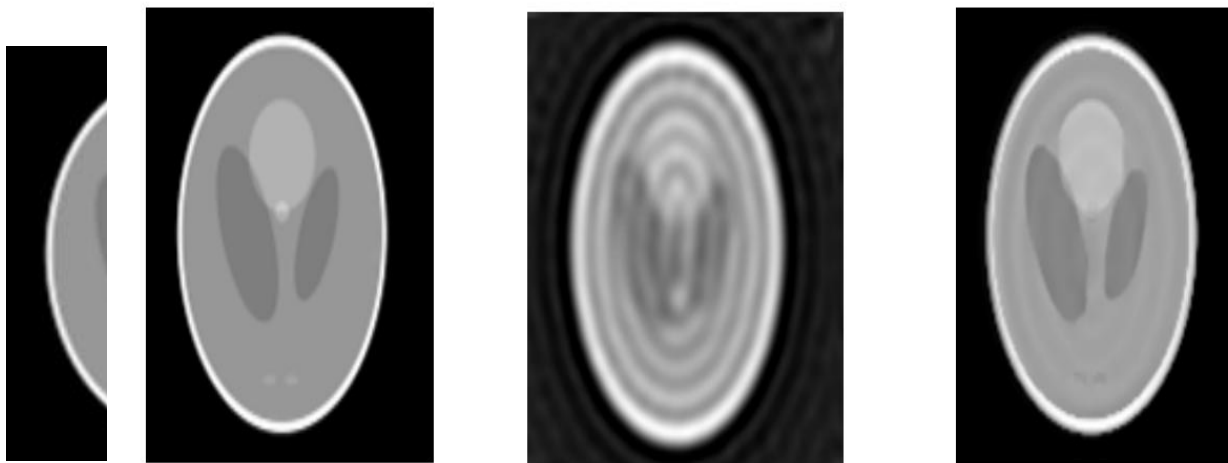
数学



台湾，2008年6月30日-7月2日

结果必须通过域进行验证

CAPERS.



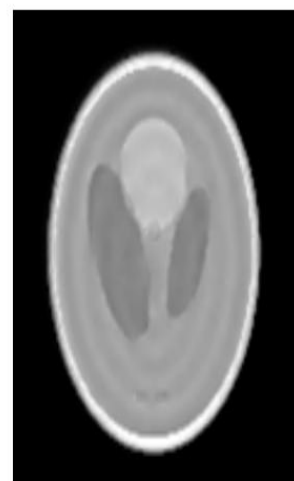
True

Gridded

CPU.DP



CPU.SP



GPU.Tune

台湾，2008年6月30日-7月2日

CUDA用于多核CPU

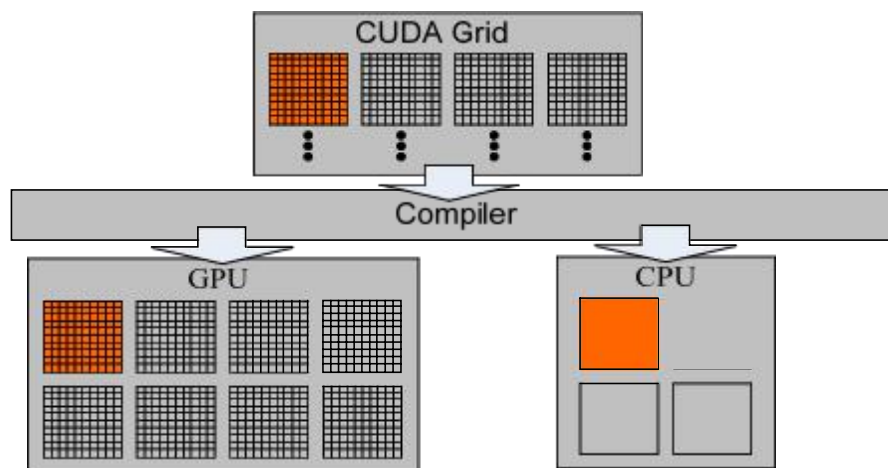
单个GPU线程对于一个CPU线程来说太小了

-CUDA仿真可以做到这一点，但性能很差

CPU核心为ILP设计，SIMD

-优化编译器与迭代循环

将GPU线程块从CUDA转换为迭代的CPU循环



台湾，2008年6月30日-7月2日

更大的图片性能结果

超过手动调优的单线程代码的一致加速
对GPU和CPU的最佳优化并不总是相同的

应用程序	C出现在单 核CPU上 时间	4核CPU上的 CUDA 时间	加速*	G80上的CUDA 时间
核流体动力学	~1000s	230s	~4x	8.5s
cp	180s	45s	4x	.28s
悲哀的	42.5ms	25.6ms	1.66x	4.75ms
MM (4Kx4K)	7.84s**	15.5s	3.69x	1.12s

*通过手工优化的CPU

**英特尔MKL，多核执行

台湾，2008年6月30日-7月2日

对许多人来说，这是一个很好的机会

GPU并行计算允许

- “被发现的时间”的急剧减少
- 1st基于原理的有意义规模上的模拟
- 新，3rd研究的范式：计算实验

要去发现的权力的“民主化”

- \$2000/特拉SPFP在个人电脑
- \$500万/个DPFP在两年内集群

HW成本将不再是大型科学领域的主要障碍

你会有所改变的！

©DavidKirk/NVIDIA和文美W。吴
台湾，2008年6月30日-7月2日

台湾2008CUDA课程

大规模并行处理器编程：CUDA体验

第二讲，CUDA编程

模型

概述

CUDA编程模型-----基本概念和数据类型

CUDA应用程序编程接口-基本功能

用简单的例子来说明基本的概念和功能

稍后将介绍性能特性

CUDA-C, 没有着色器的限制!

集成的主机+设备应用程序C程序

- 主机C代码中的串行或适度并行的部分
- 设备SPMD内核C代码

串行代码（主机）

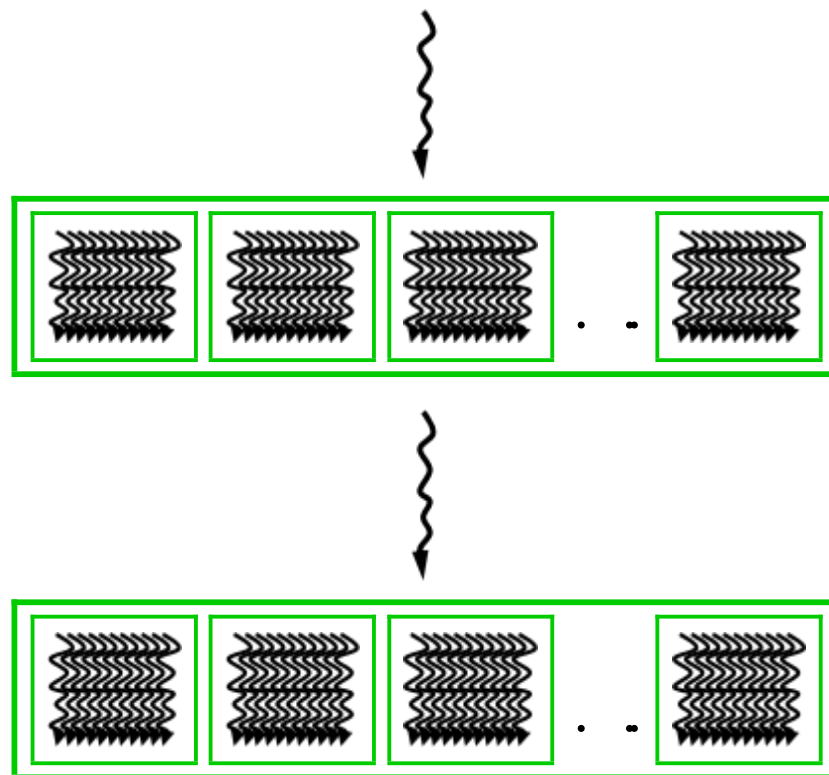
并行内核（设备）

```
KernelA<<<nBlk, nTid>>>(args);
```

串行代码（主机）

并行内核（设备）

```
KernelB<<<nBlk, nTid>>>(args);
```



CUDA设备和线程

Subaraga计算装置

- 是CPU或主机的协处理器
- 有自己的DRAM（设备内存）
- 并行运行多个线程
- 是一个典型的GPU，但也可以是另一种类型的并行处理设备

应用程序的数据并行部分被表示为在许多线程上运行的设备内核

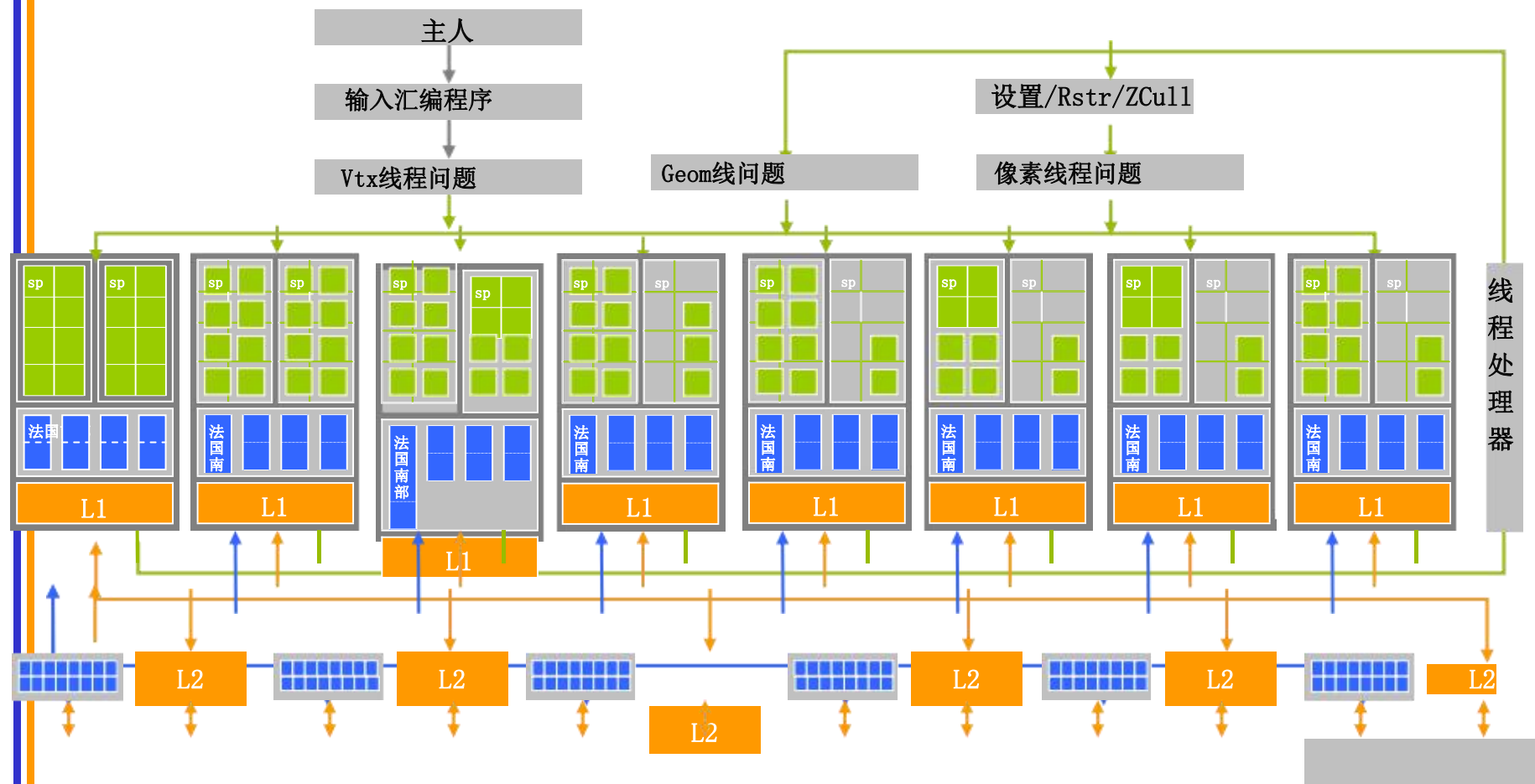
GPU和CPU线程之间的区别

- GPU线程非常轻
 - 创建开销非常小
- GPU需要1000个线程来完全提高效率
 - 多核CPU只需要少数

G80图形模式

gpu的未来是可编程处理

因此，请围绕处理器构建体系结构

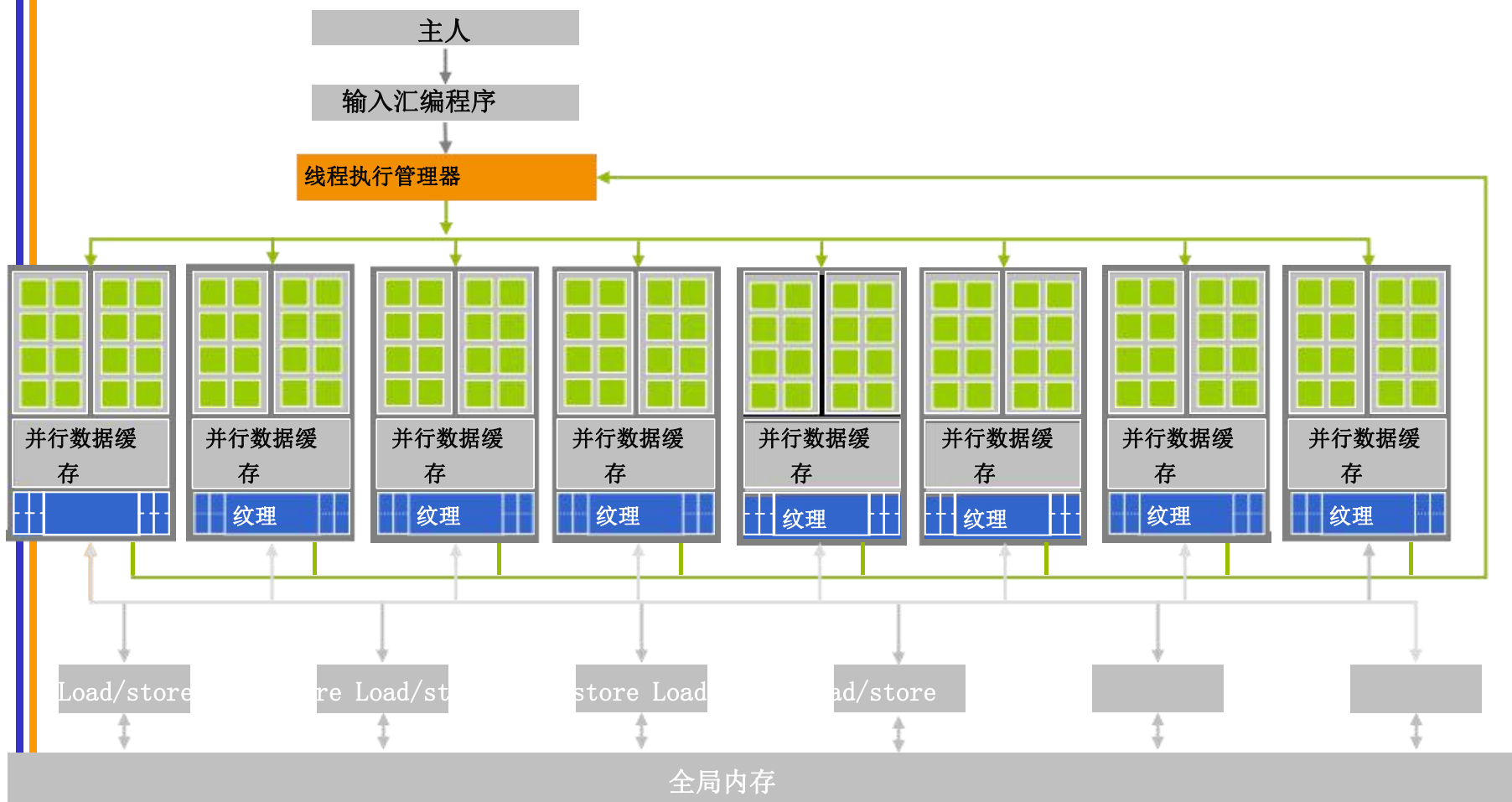


台湾，2008年6月30日-7月2日

G80CUDA模式-一个设备示例

处理器执行计算线程

新的操作模式/硬件计算界面



CUDA扩展C

- Declspecs

- 全局、设备、共享、本地、不变

©DavidKirk/NVIDIA和文美W. Hwu台湾，2008年6月30日-7月2日

《囊桂树》关键词

- threadIdx, blockIdx

沙痂病病

- __syncthreads

沙桂油运行时API

- 内存、符号、执行管理

董事长函数启动

__device__ 浮子滤波器

```
__global__ 空卷积（浮动*图像）{  
  
    __shared__ 浮动区域[M];  
    ...  
  
    region[threadIdx] = image[i];  
  
    __syncthreads()  
  
    ...  
  
    图像[j]=结果;  
}
```

//分配GPU内存

空白*我的图像=cudaMalloc（字节）

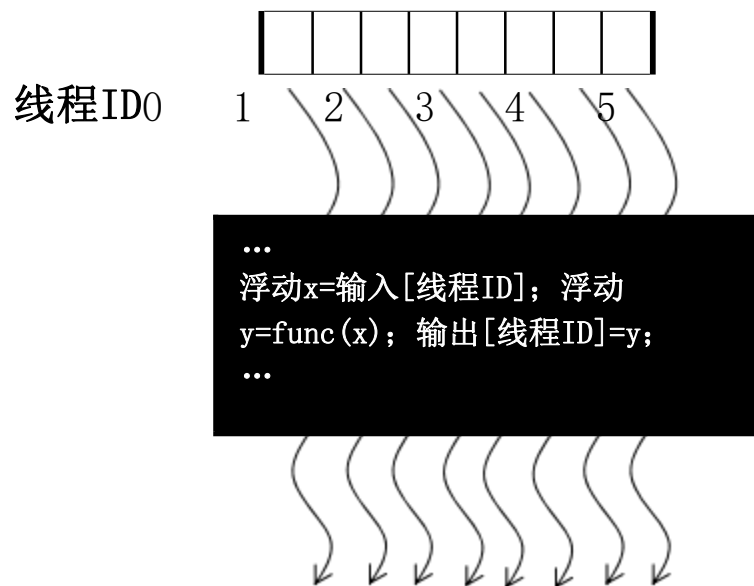
//100个块，每个块10个线程

convolve<<<100, 10>>> (myimage);

并行线程数组

一个CUDA内核是由一个线程数组来执行的

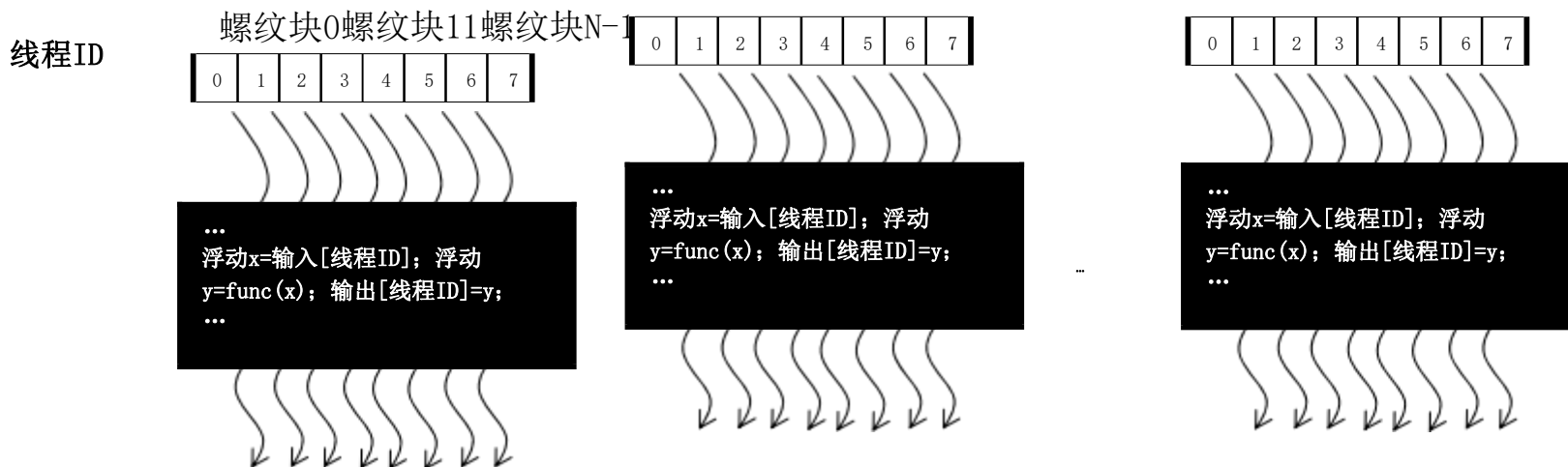
- 所有的线程都运行相同的代码 (SPMD)
- 每个线程都有一个ID，它用于计算内存地址和做出控制决策



线程块：可扩展的协作

将单片线程阵列划分为多个块

- 块内的线程通过共享内存、原子操作和屏障同步进行协作
- 不同块中的线程不能协作



©DavidKirk/NVIDIA和文美W。吴
台湾，2008年6月30日-7月2日

块ID和线程ID

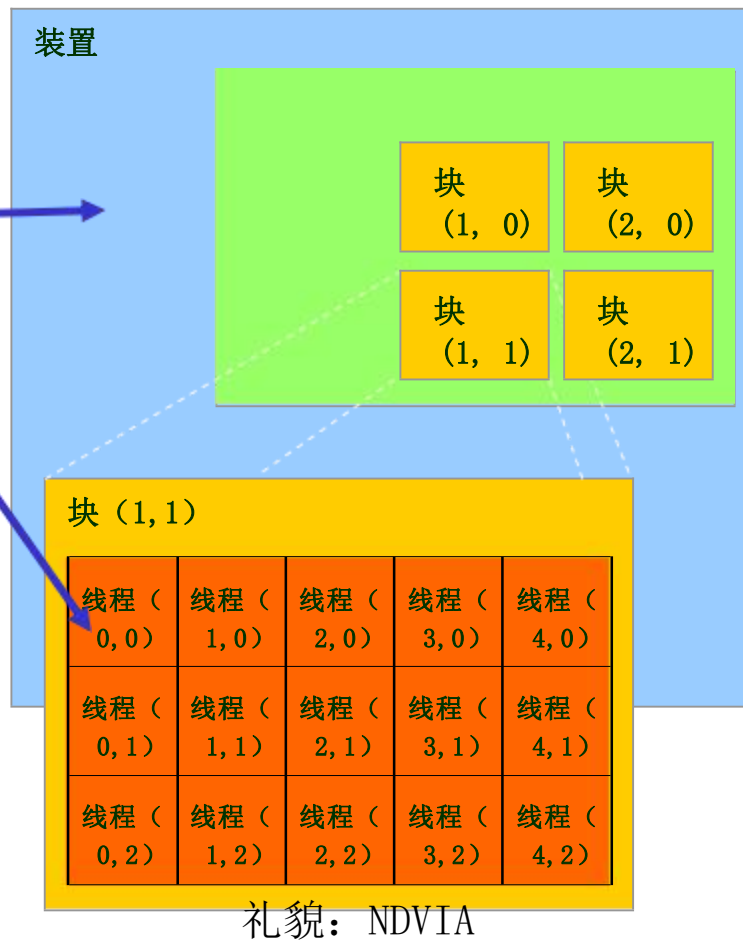
每个线程都使用id来决定要处理什么数据

- 方块ID: 1D或2D
- 螺纹线程ID: 1D、2D或3D

简化了在处理多维数据时的内存寻址

- 图像处理
- 解决体积上的偏微分方程

- ...

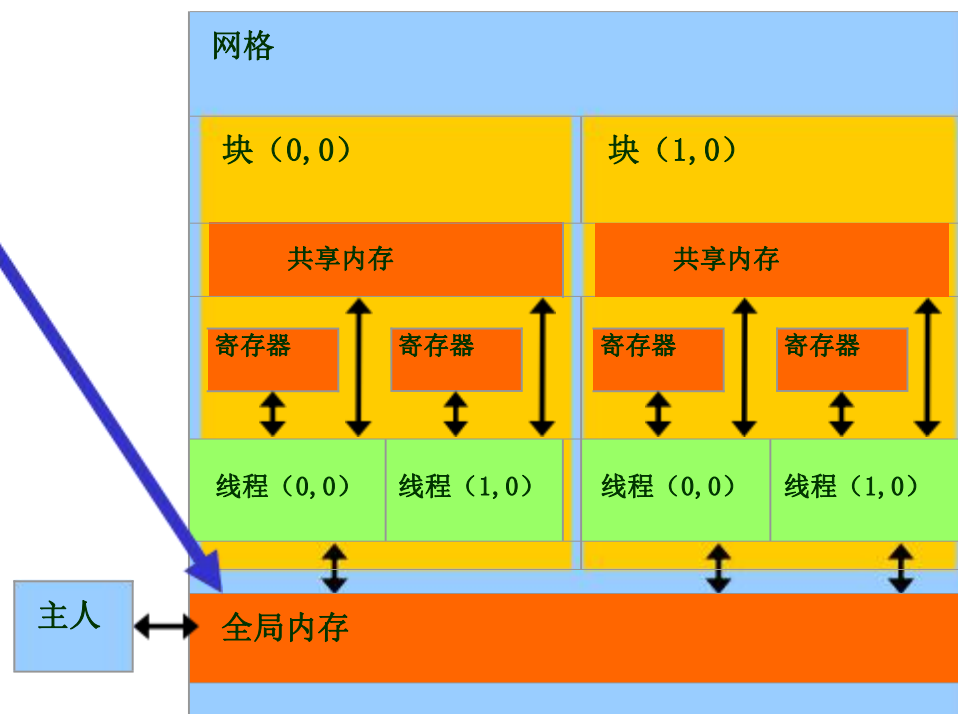


CUDA内存模型概述

沙漏全球存储器

- 主机与设备之间收发数据的主要方式
- 对所有线程都可见的内容
- 长时间延迟访问

我们将关注全局内存



台湾，2008年6月30日-7月2日

CUDA API的亮点：简单而轻量级

该API是对ANSIC编程语言的扩展

→ 低学习曲线

硬件设计支持轻量级运行时和驱动程序

→ 高性能

CUDA设备内存分配

- `cudaMalloc()`

- 在设备的全局设置中分配对象 记忆力

- 需要两个参数

- 指向已分配对象的指针的地址

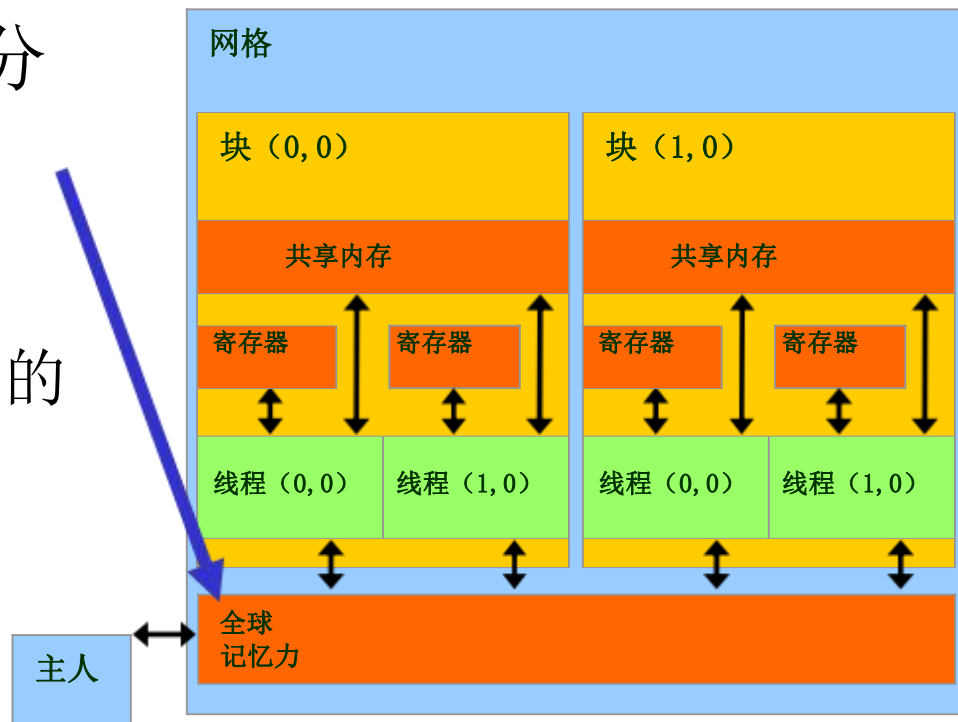
- 已分配的对象的大小

-

- 从设备中释放对象

- 全局内存

- 指向已释放的对象的指针



©DavidKirk/NVIDIA和文美W。吴
台湾，2008年6月30日-7月2日

CUDA设备内存分配（续）

Subarg代码示例：

- 分配一个64*64个的单精度浮点阵列
- 将已分配的存储空间附加到Md. 元素上
- “ - “d” 通常用于表示设备的数据结构

```
tile_width = 64;
```

矩阵Md

```
int大小=TILE_WIDTH*TILE_WIDTH*尺寸（浮动）；
```

`cudaMalloc((void**)&Md. 元素, 大小);` 免
费的(Md. 成分

CUDA主机设备数据传输

- `cudaMemcpy()`
存储器数据传输

-需要四个参数

隔墙指针至目的地

沙桂油指针源

已复制的字节数

沙进行转移类型

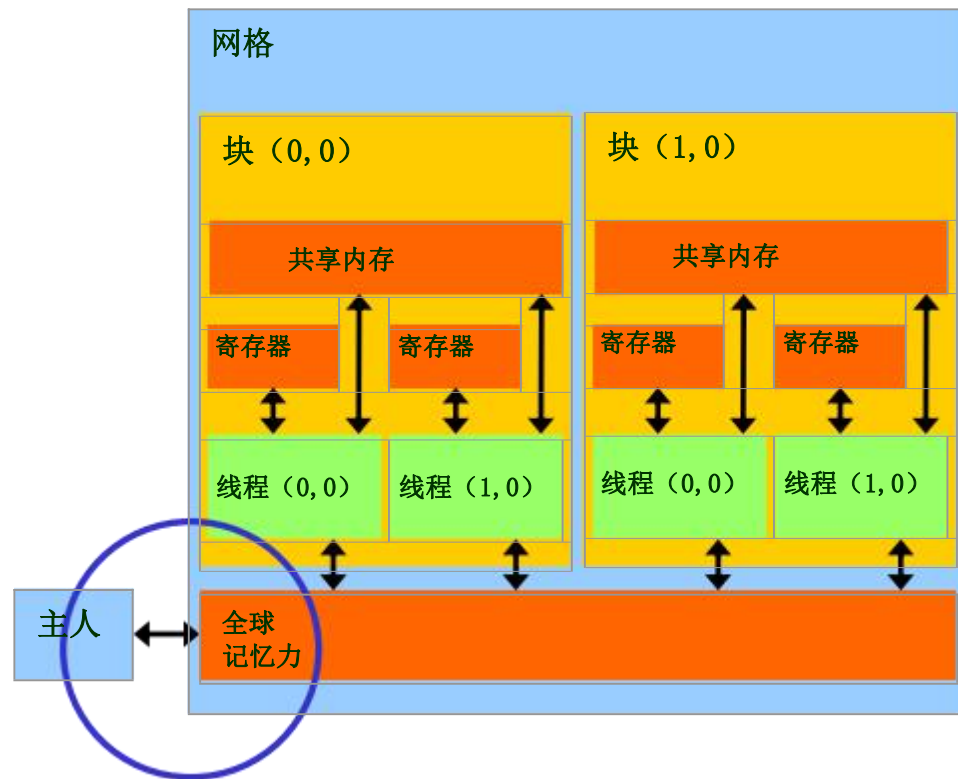
-主机到主机

-主机到设备

-设备到主机

设备到设备

隔墙异步传输



台湾，2008年6月30日-7月2日

CUDA主机设备数据传输 (续)

Subarg代码示例:

- 传输一个为64*64英寸的单精度浮子阵列
- M在主机内存中, Md在设备内存中
- 设备设备和设备主机是符号常数

Md. 元素, M. 元素、尺寸、虚拟设备);

cudaMemcpy (M. elements, Md. 元素, 大小, 到
主机);

CUDA关键字

台湾，2008年6月30日-7月2日

CUDA函数声明

	已执行 在:	只能从以下 位置调用 :
<code>__device__</code> 浮动设备Func()	设备	设备
<code>__global__</code> 空白角Func()	设备	主人
<code>__host__</code> floatHostFunc()	主人	主人

- `__global__` 定义了一个内核函数
 - 必须返回无效
- `__device__` 和 `__host__` 可以一起使用

台湾，2008年6月30日-7月2日

CUDA函数声明（续。）

- `__device__` 函数不能被获取它们的地址
对于在设备上执行的功能：

无递归

- 在函数内部没有静态变量声明
- 没有变量数量的参数

调用一个内核函数-线程 创世

必须使用以下执行配置调用内核函数：

```
__global__ void KernelFunc (...);  
dim3 DimGrid (100, 50); //5000个线程块  
dim3 二块 (4, 8, 8); //256线程  
块  
size_t 共享内存字节=64; //64字节的共享  
记忆力
```

```
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes  
>>> (...);
```

任何对内核函数的调用都是从CUDA1.0开始异步的，
阻塞需要显式同步

一个简单的运行示例矩阵乘法

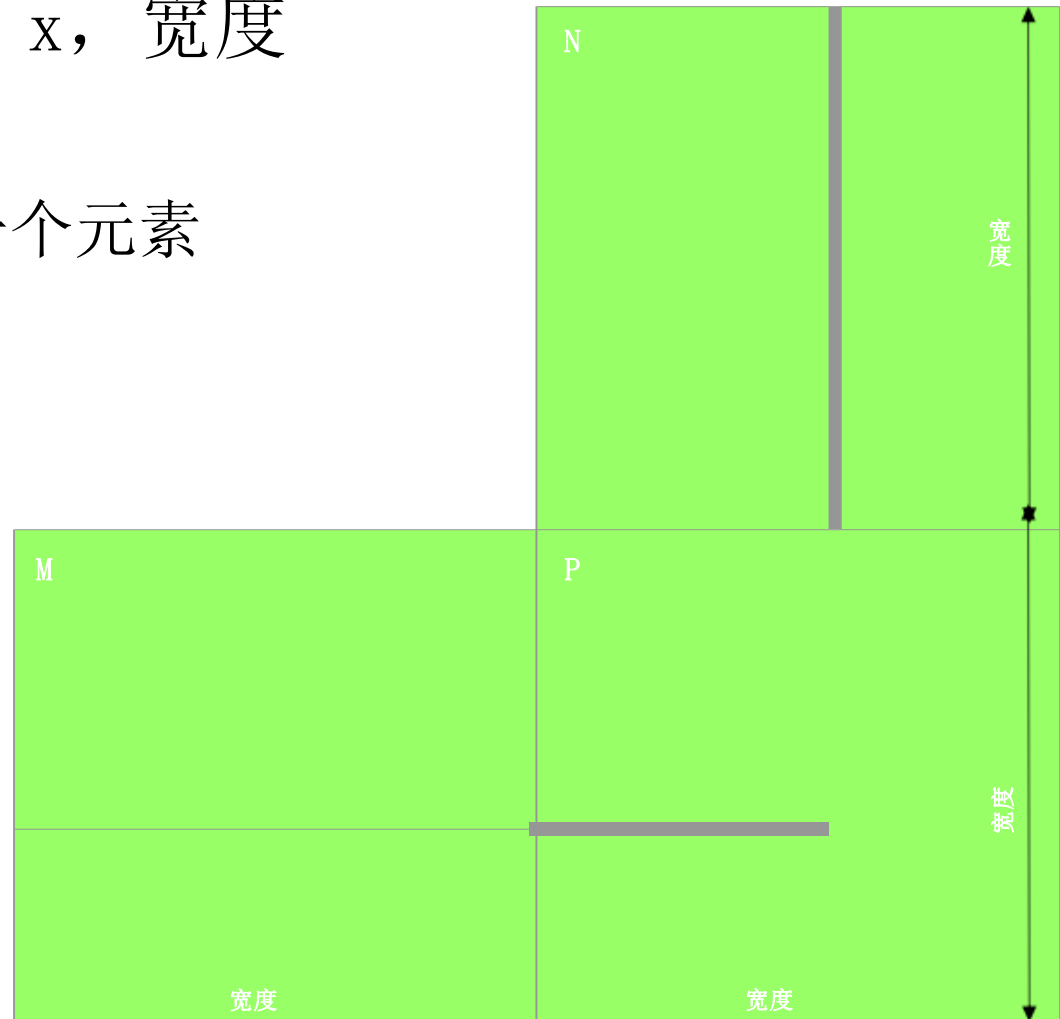
一个简单的矩阵乘法示例，说明了CUDA程序中内存和线程管理的基本特性

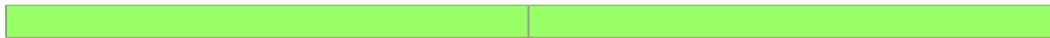
- 保留共享内存的使用情况，直到以后再使用
- 本地，注册使用
线程ID使用情况
- 主机和设备之间的内存数据传输API
- 为简单起见，假设平方矩阵为方阵

编程模型：平方矩阵乘法示例

$P=M*N$ 的尺寸，宽度， x ，宽度
不带瓷砖的隔墙：

- 一个线程计算 P 的一个元素
- M 和 N 为加载宽度时间
来自全局内存





步骤1：矩阵乘法，一个简单的主机版本在C中

//矩阵乘法在(CPU)主机上的双精度空白矩阵多主机(浮动

, 浮动, N, 浮动, P, intW idth)

{

为(int i=0; i<宽度; ++i)

为(int j=0; j<宽度; ++j) {

双和=0;

为(int k=0; k<宽度; ++k) {

双倍=M, 宽度+];

双b=N[k*宽度+j];

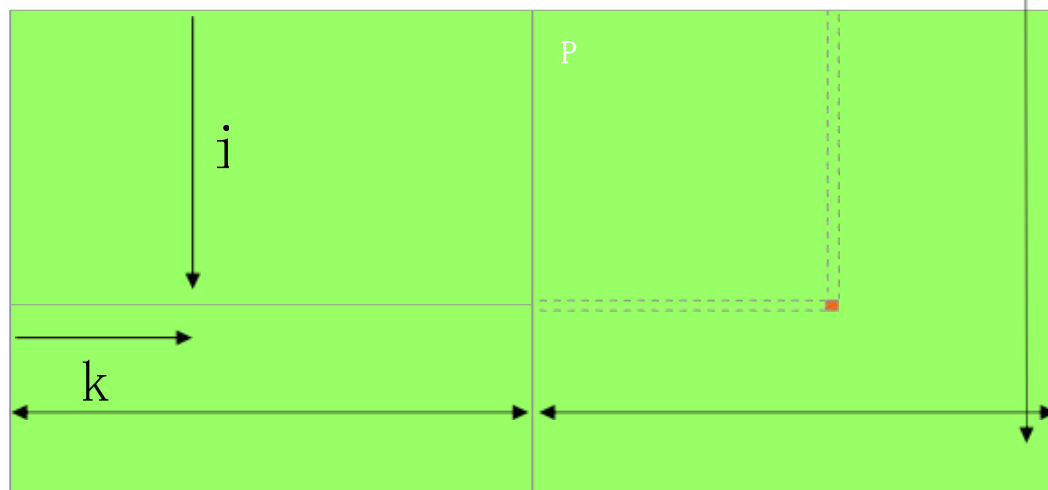
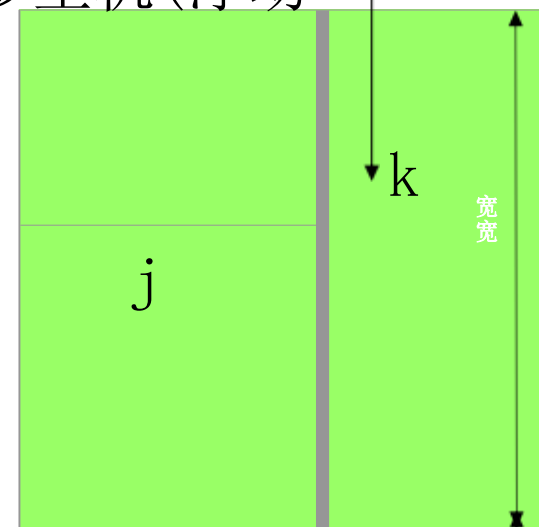
总和+=a*b;

}

P宽度+j]=和;

}

}



©DavidKirk/NVIDIA和文美W。吴
台湾，2008年6月30日-7月2日

步骤2： 输入矩阵数据传输 (主机端代码)

```
(浮动, 浮动, 浮动, int宽度){  
    int尺寸=宽度*宽度*尺寸(浮动);  
    float* Md, Nd, Pd;  
    ...  
1. //分配并加载M, N到设备内存卡达洛(&Md, 大小);  
    (Md, M, 大小, 虚拟主机设备);  
  
    CudaMalloc(大小);  
    cudaMemcpy(编号, 大小, 数字主机设备);  
  
    //在设备上分配P  
    CudaMalloc(&Pd, 大小);
```

步骤3：输出矩阵数据传输 (主机端代码)

2. //内核调用代码-稍后将显示

...

3. //从设备中读取P
(P, Pd, 大小, 到主机);

//免费设备矩阵

```
cudaFree (Md); cudaFree (Nd); cudaFree (Pd);  
}
```


台湾，2008年6月30日-7月2日

步骤4： 内核函数

//矩阵乘法内核-每个线程代码

```
__global__ void 矩阵多内核(浮动*Md, 浮动*Nd, 浮动*Pd, int宽度) {  
    //2D线程ID  
    int tx = threadIdx.x;  
    int ty = threadIdx.y;  
  
    //Pvalue用于存储矩阵的元素  
    由线程计算出的//  
    浮动Pvalue=0;
```

第4步：内核函数（续）

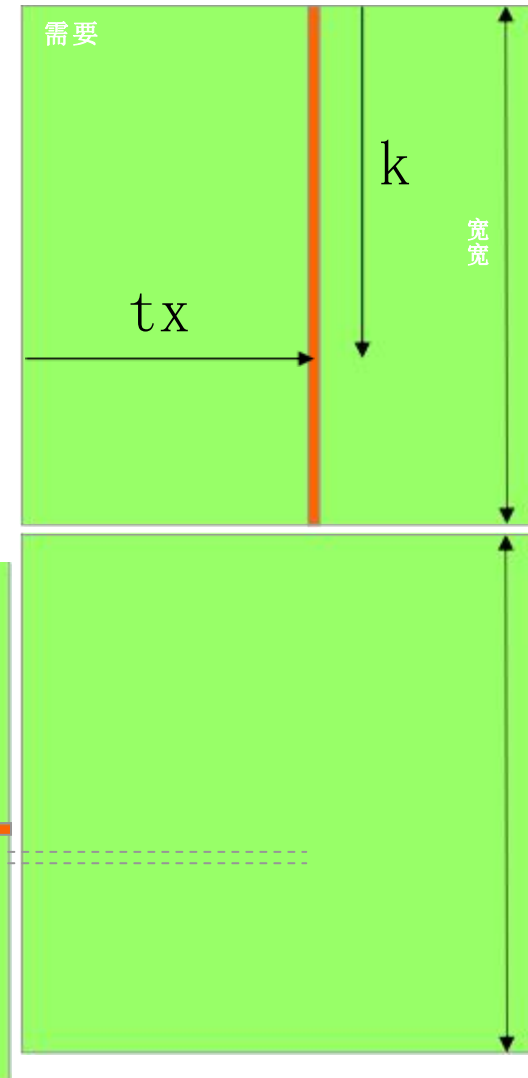
用于 (int k=0; k<宽度; ++k)

```
{
    浮点元素=Md[ty*宽度+k];
    浮点Nelement=Nd[k*宽度+tx];
    Pvalue += Melement * Nelement;
}
```

//写出这个矩阵来设计 e内存; 每行
//each thread writes one element

```
Pd[ty*Width+tx] = Pvalue;
```

```
}
```



台湾，2008年6月30日-7月2日

步骤5：内核调用 (主机端代码)

```
//设置执行配置
```

```
dim3dimblock (宽度、宽度) ;
```

```
dim3 dimGrid (1, 1);
```

```
//启动设备计算线程！
```

```
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd) ;
```

©DavidKirk/NVIDIA和文美W。吴
台湾，2008年6月30日-7月2日

仅使用了一个线程块

一个线程块计算矩阵Pd

- 每个线程计算Pd的一个元素

散布每根线

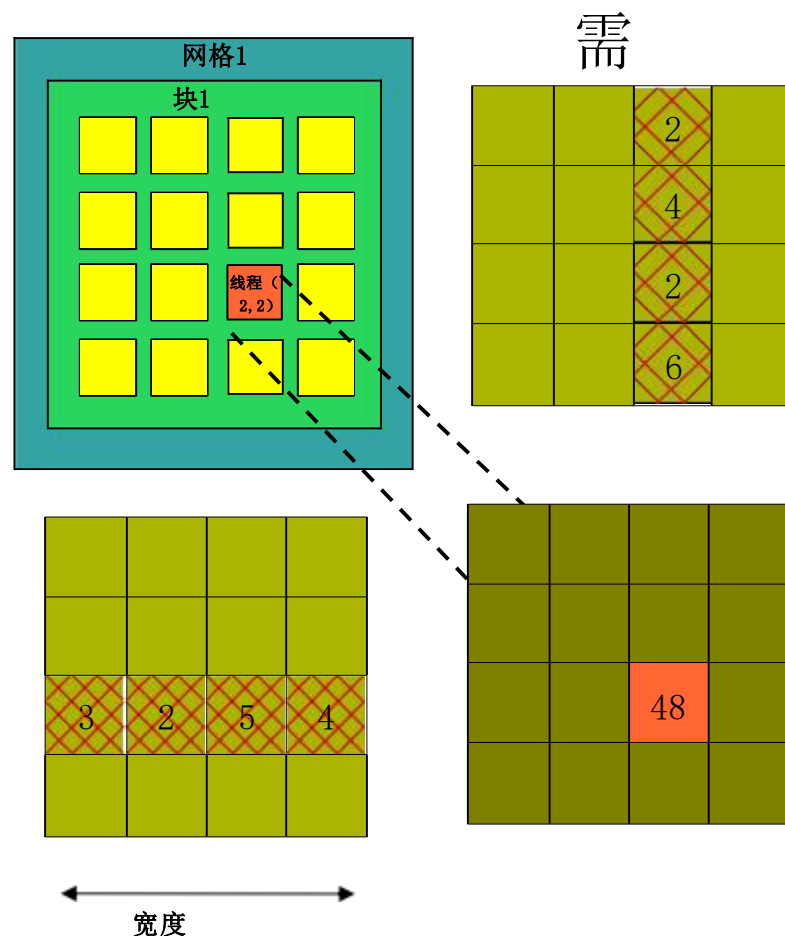
- 加载一行矩阵Md

- 加载一列矩阵Nd

- 对每一对Md和Nd元素执行一次
乘法 and 加法

- 计算到芯片外内存存取率接近1
: 1 (不是很高)

受线程块中允许的线程数量限制的
矩阵的大小



第7步：处理任意大小的方阵

让每个二维线程块来计算

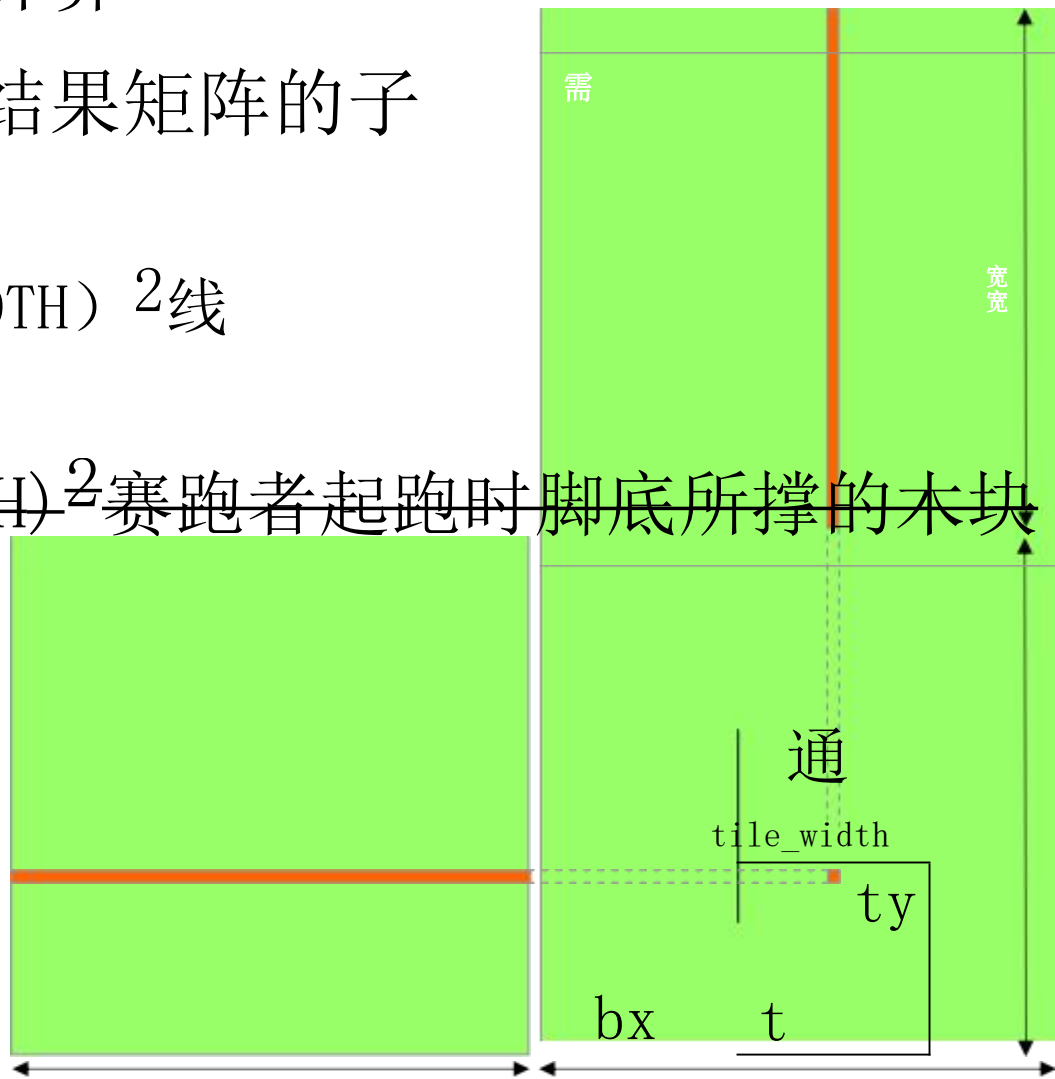
$a(\text{TILE_WIDTH})^2$ 结果矩阵的子
矩阵（瓷砖）

-每个都有 $(\text{TILE_WIDTH})^2$ 线


生成的二维网格

$(\text{WIDTH}/\text{TILE_WIDTH})^2$ 赛跑者起跑时脚底所撑的木块

你仍然需要放一个循环
围绕着内核调用的案例
其中宽度/ TILE_WIDTH
是否大于最大网格大小
(64K)！



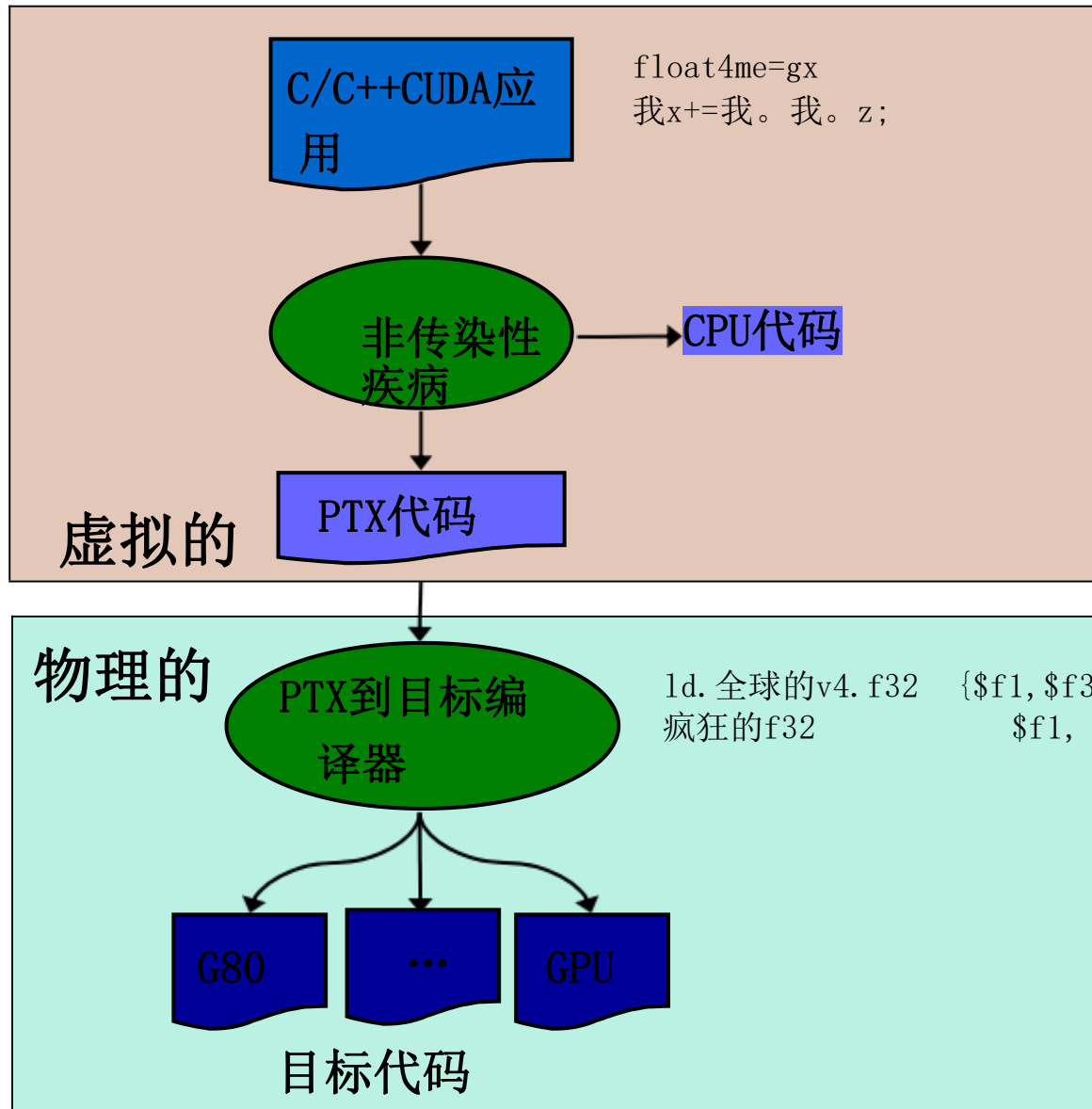
©DavidKirk/NVIDIA和文美W。吴
台湾，2008年6月30日-7月2日



一些有用的信息 工具

台湾，2008年6月30日-7月2日

编译CUDA程序



平行螺纹提取 (PTX)

- 虚拟机和ISA
- 编程模型
- 执行资源和

编译

任何包含CUDA语言扩展名的源文件都必须使用
NVCC进行编译

NVCC是一个编译器驱动程序

- 通过调用所有必要的工具和编译器，如`cudacc`，
`g++`，`cl`，...

SubarNVCC输出：

- C代码（主机CPU代码）

然后必须使用其他工具与应用程序的其余部分一起进行编译吗

- PTX

Subab对象代码直接

或者，PTX源代码，在运行时进行解释

©DavidKirk/NVIDIA和文美W。吴
台湾，2008年6月30日-7月2日³³

耦合

任何具有CUDA代码的可执行文件都需要两个动态库：

- CUDA运行时库(cudart)
- CUDA核心文库(cuda)

©DavidKirk/NVIDIA和文美W。吴
台湾，2008年6月30日-7月2日

调试使用

设备仿真模式

在设备仿真模式(nvcc-设备模式)下编译的可执行文件完全使用CUDA运行时在主机上运行

- 不需要任何设备和CUDA驱动程序
- 每个设备线程都模拟了一个主机线程

在设备模拟模式下运行时，您可以：

- 使用主机本地调试支持（断点、检查等。）
- 从主机代码中访问任何特定于设备的数据，反之亦然
- 从设备代码调用任何主机函数(e. g. **Printf**)，反之亦然
- 检测因使用不当而导致的死锁情况

__sync threads

设备仿真模式陷阱

模拟的设备线程按顺序执行，因此，多个线程同时访问相同的内存位置可能会产生不同的结果。

解除引用主机上的设备指针或设备上的主机指针可以产生正确的结果

结果为设备仿真模式，但在设备执行模式下会产生一个错误

浮点

浮点计算的结果会略有不同，因为：

- 不同的编译器输出，指令集
- 对中间结果使用扩展的精度

有多种选项可以强制对主机进行严格的单一精度操作

台湾2008CUDA课程

大规模并行处理器编程：CUDA体验

第三讲：CUDA记忆

CUDA变量类型限定符

变量声明				记忆力	范围	寿命
<code>__device__</code>	<code>__local__</code>	<code>int</code>	<code>LocalVar;</code>	本地的	线	线
<code>__device__</code>	<code>__shared__</code>	<code>int</code>	<code>SharedVar;</code>	共享的	块	块
<code>__device__</code>		<code>int</code>	<code>GlobalVar;</code>	全球的	网格	申请
<code>__device__</code>	<code>__constant__</code>	<code>int</code>	<code>ConstantVar;</code>	持续不断的	网格	申请

- 当__device__与__local__、__shared__或__constant__一起使用时，它是可选的
- 没有任何限定符的自动变量位于寄存器中
 - 除了位于本地内存中的数组之外

在哪里声明变量？

主机可以访问它吗？

是的没有

全局常
量

寄存器（自动）共
享
本地的

外
任何功能

在内核中

台湾，2008年6月30日-7月2日

G80实现了CUDA内存

Subark每个线程可以:

-按线程读取/写入

登记

-按线程读取/写入

局部存储器

-每块读写

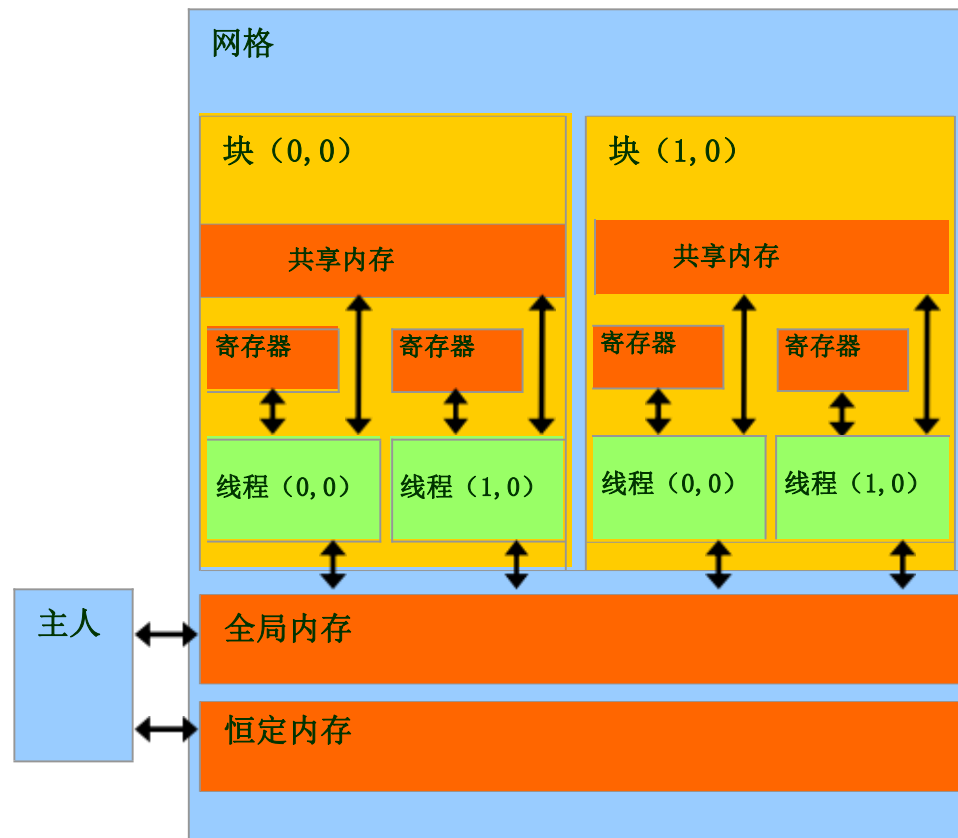
共用存储器

-按网格读取/写入

全局存储器

-只读每网格

常数存储器



台湾，2008年6月30日-7月2日

通用编程策略

全局内存位于设备内存 (DRAM) 中

- 访问速度比共享内存要慢得多

因此，在设备上执行计算的一种有利可图的方法是平铺数据，以利用快速共享内存：

- 将数据划分为适合于共享内存的子集

- 使用一个线程块处理每个数据子集：

 - 正在将该子集从全局内存加载到共享内存中，

 - 使用多个线程来利用内存级的并行性

 - 从共享内存对子集执行计算；每个线程都可以有效地多次传递任何数据元素

 - 正在将结果从共享内存复制到全局内存

通用编程策略

(续)

固定内存还驻留在设备内存 (DRAM) 中——访问速度比共享内存要慢得多

但是... 缓存!

-对只读数据的高效访问

根据访问模式仔细划分数据

-R/Only>固定内存 (进入高速缓存时非常快)

-R/W在Block>共享内存中共享 (非常快)

-每个线程>寄存器内的记录 (非常快)

-收发输入/结果>全局内存 (非常慢)

有关纹理内存的使用情况, 请参见课程。 ece.uiuc.edu/ece498/al.

变量类型限制

指针只能指向在全局内存中已分配或已声明的内存：

- 已在主机中分配，并已传递给内核：

 - `__global__ 空白内核基金(浮动*ptr)`

- 作为一个全局变量的地址获得：

 - `float* ptr = &GlobalVar;`

GPU原子整数运算

对全局内存中的整数执行的原子操作：

- 在有符号/无符号int上的关联操作
- 添加、子、最小、最大、...
- 和，或，xor
- 增加，减少
- 交换、比较和交换

需要具有计算能力的硬件1.1

台湾，2008年6月30日-7月2日

使用共享内存的矩阵乘法

第4步：内核函数（续）

```

用于(int k=0; k<宽度; ++k)
{
    浮点元素=Md[ty*宽度+k];
    浮点Nelement=Nd[k*宽度+tx];
    Pvalue += Melement * Nelement;
}

```

//写出这个矩阵来设计 e内存; 每日

```

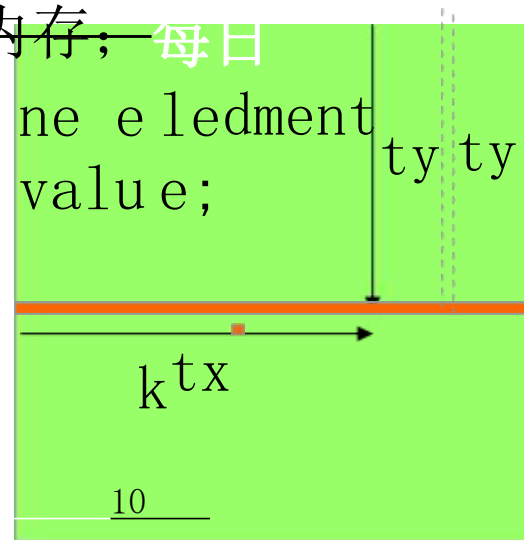
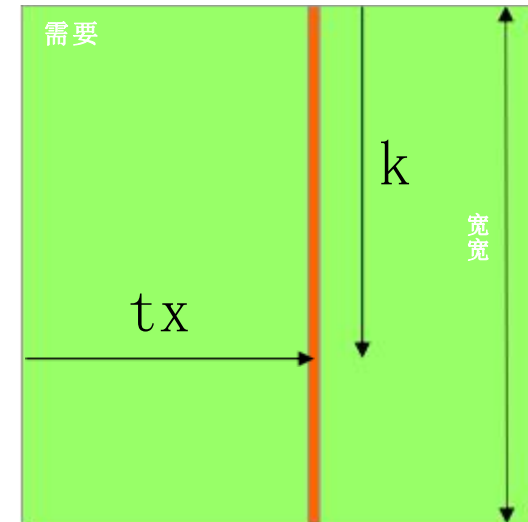
//each thread writes one element
Pd[ty*Width+tx] = Pvalue;

```

```

}

```



台湾，2008年6月30日-7月2日

在G80上的性能怎么样？

所有线程都为其输入矩阵元素访问全局内存

- 每个浮点数增加两次内存访问（8字节）

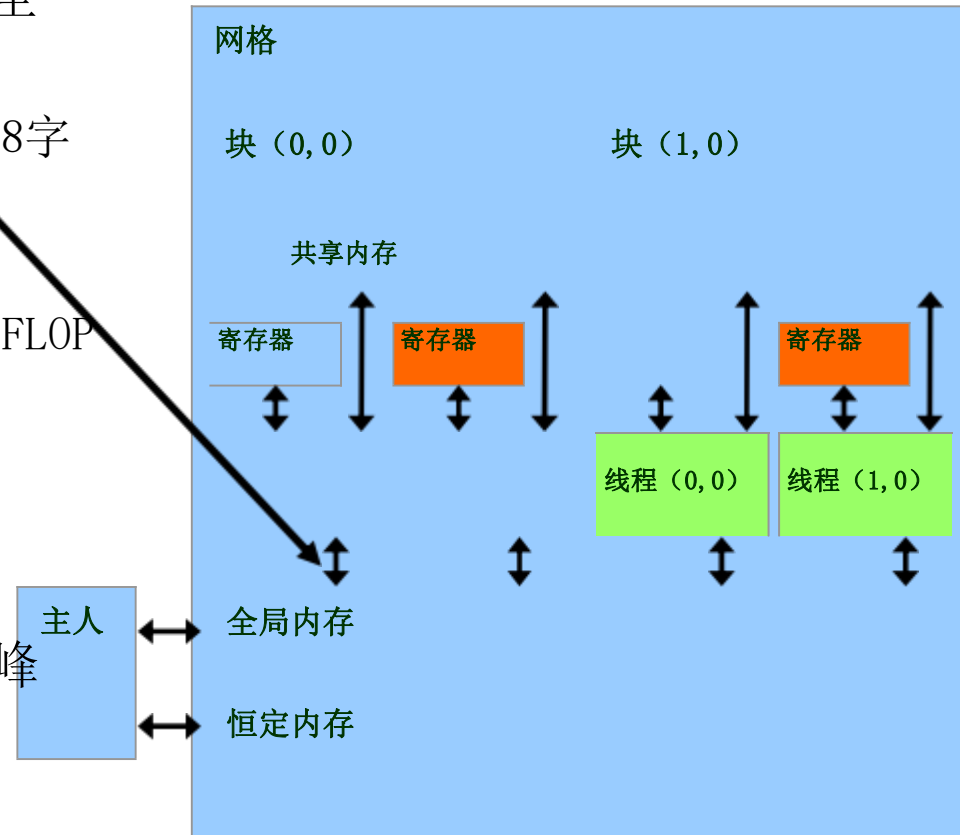
4B/秒的内存带宽

-4*346.5=1386GB/s需要达到峰值FLOP等级

-86. 4GB/s限制代码为
21.6条

实际的代码运行在15GFL0PS左右

需要大幅减少内存访问，以更接近峰值346.5GFLOPS



分层乘法

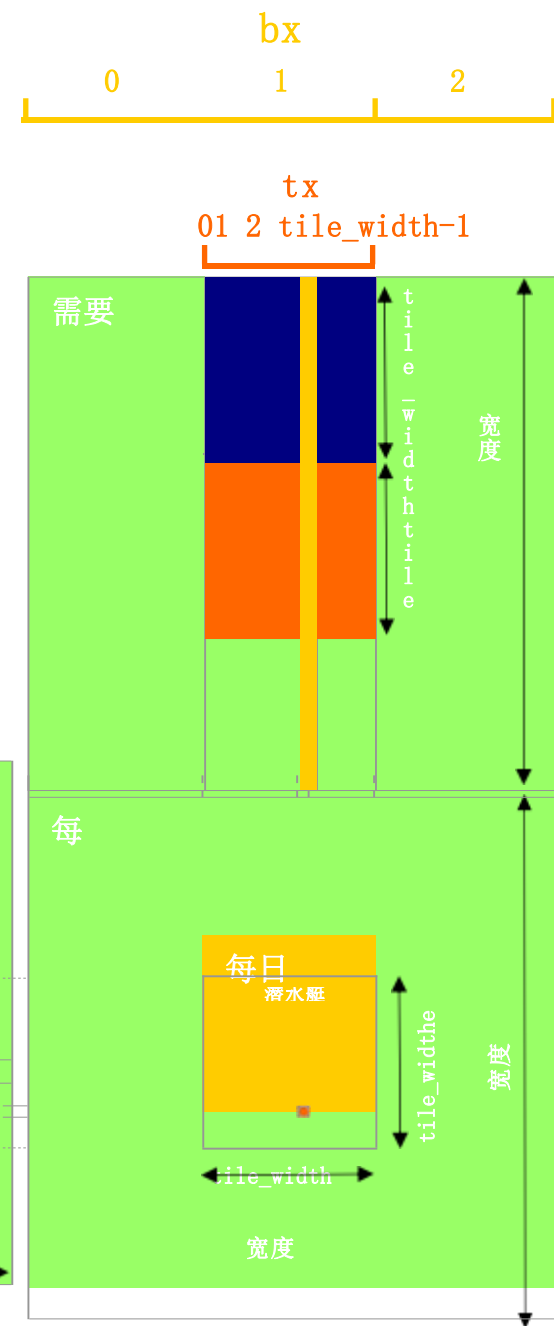
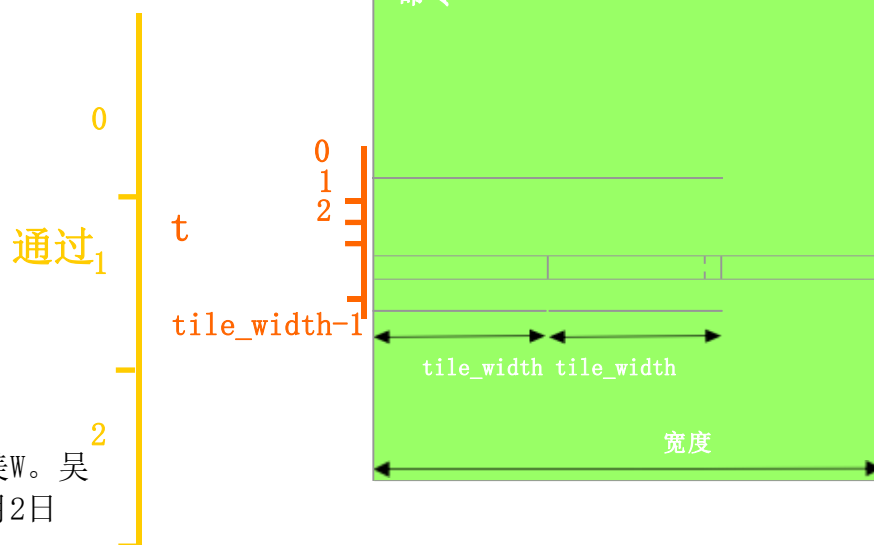
每个块计算一个平方的子矩阵Pd潜

水艇大小TILE_WIDTH

每个线程计算一个Pd元素潜水

艇

假设Md和Nd的维数是TILE_WIDTH的
倍数



G80中的一级尺寸考虑因素

每个线程块应该有许多线程

-TILE_WIDTH的16给出 $16*16=256$ 个线程

应该有许多线程块

-A1024*1024Pd可给出 $64*64$ 个=4096个线程块

每个线程块为 $256 * (2*16) = 8,192$ mul/add操作

从全局内存执行 $2*256=512$ 浮动加载。

-内存带宽不再是一个限制因素

CUDA代码-内核执行 配置

//设置执行配置

```
dim3暗块(TILE_WIDTH, TILE_WIDTH);  
昏暗的3个昏暗的网格(宽度/TILE_WIDTH,  
                        宽度/TILE_WIDTH);
```

台湾，2008年6月30日-7月2日

CUDA代码-内核概述

//块索引

```
int bx = blockIdx .x;
```

```
int由=bloxkIdx提供。 y;
```

//线程索引

```
int tx = threadIdx .x;
```

```
int ty = threadIdx.y;
```

//预值存储块子矩阵的元素

由线程计算出的//-自动变量!

```
浮动Pvalue=0;
```

//循环覆盖了M和N的所有子矩阵

//计算块子矩阵需要//

```
用于(intm=0; m<宽度/TILE_WIDTH; ++m) {
```

下一张小幻灯片上的代码;

台湾，2008年6月30日-7月2日

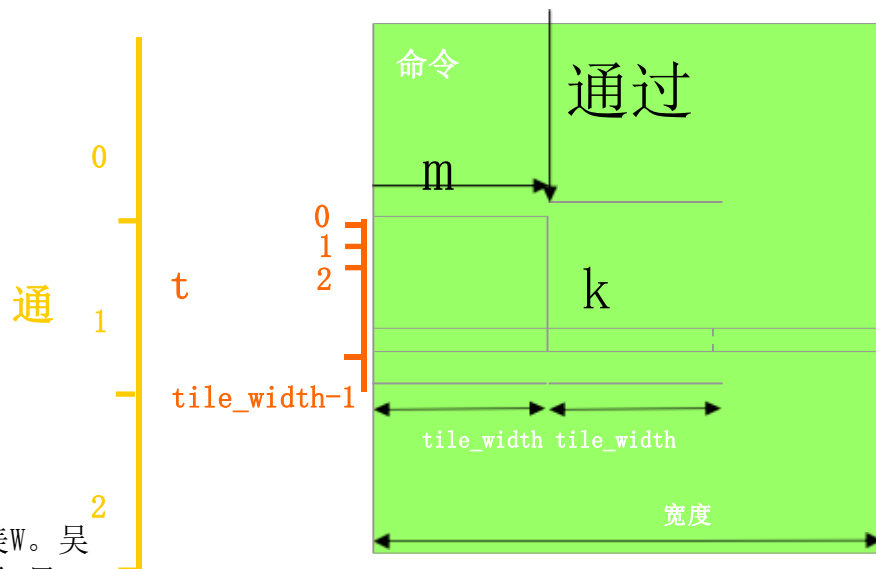
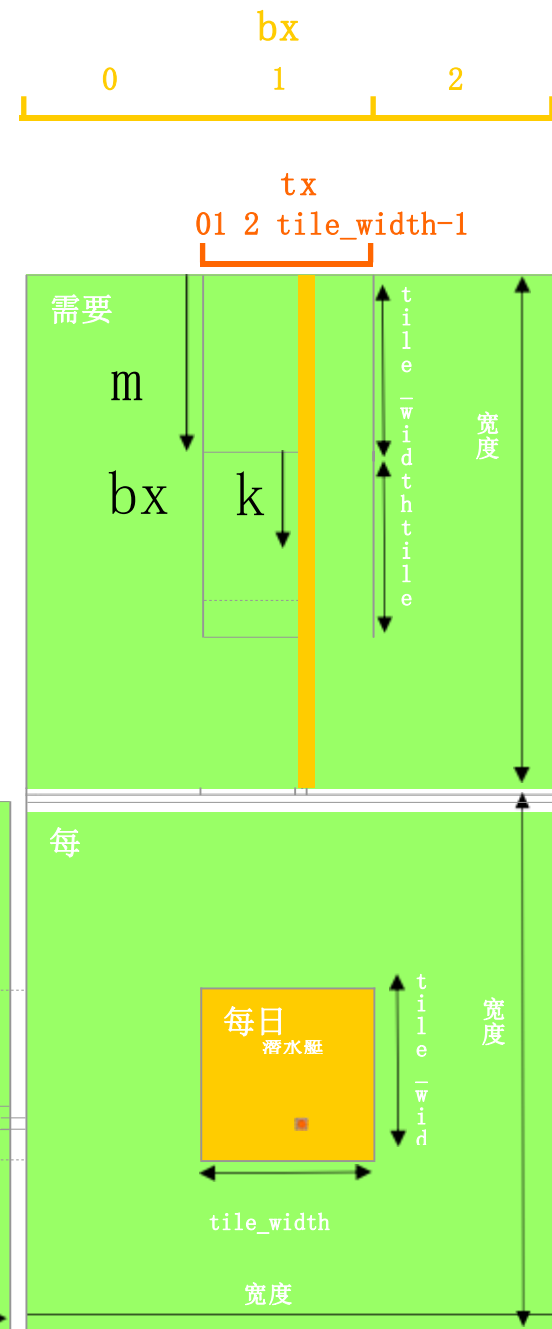
分层乘法

每个块计算一个平方的子矩阵Pd潜

水艇大小TILE_WIDTH

每个线程计算一个Pd元素潜水

艇



CUDA代码-将数据加载到共享内存中

//获取一个指向M的当前子矩阵Msub的指针

浮动*Mdsub=获取子矩阵（Md、m、by、宽度）；

//得到一个指向N的当前子矩阵Nsub的指针

浮动*Ndsub=Get子矩阵（Nd、bx、m、宽度）；

__shared__ float Mds[TILE_WIDTH][TILE_WIDTH];

__shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

//每个线程加载子矩阵中的一个元素

Mds[ty][tx] = GetMatrixElement (Mdsub, tx, ty);

//每个线程加载子矩阵中的一个元素

Nds[ty][tx] = GetMatrixElement (Ndsub, tx, ty);

©DavidKirk/NVIDIA和文美W。吴¹⁸
台湾，2008年6月30日-7月2日

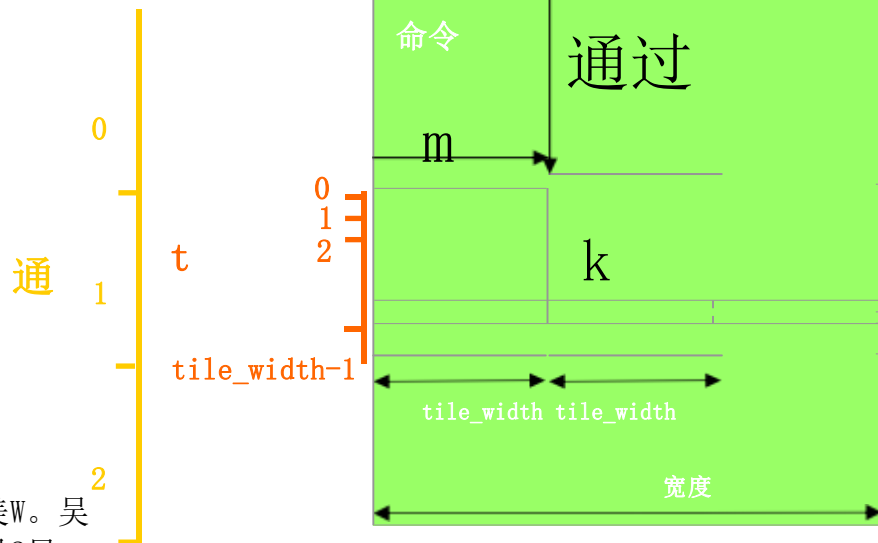
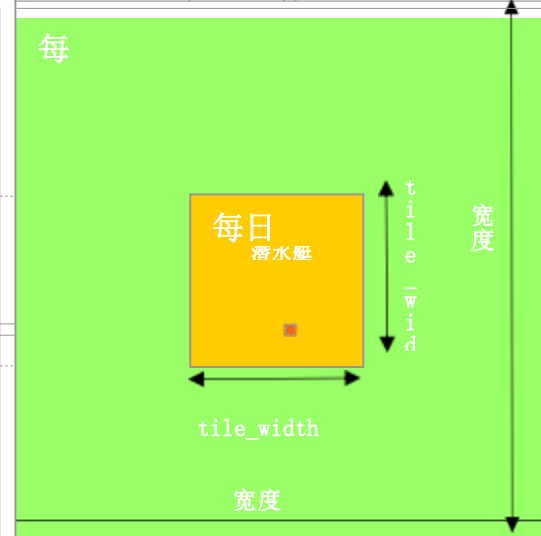
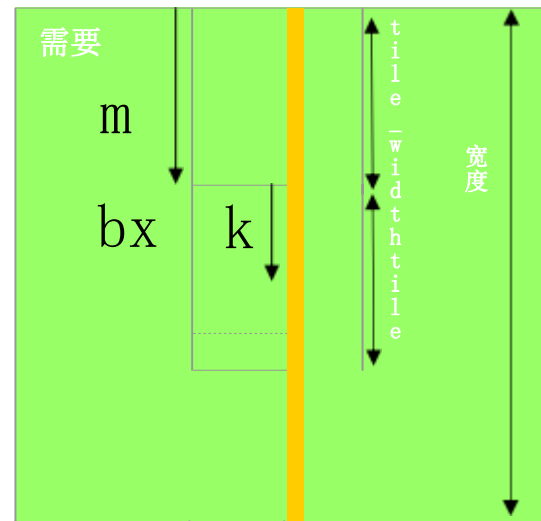
分层乘法

每个块计算一个平方的子矩阵Pd潜

水艇大小TILE_WIDTH

每个线程计算一个Pd元素潜水

艇



CUDA代码-计算结果

```
//正在进行同步，以确保子矩阵已被加载  
//之前开始计算  
__syncthreads();
```

```
//每个线程计算块子矩阵中的一个元素  
为(int k=0; k<TILE_WIDTH; ++k)
```

```
    Pvalue += Mds[ty][k] * Nds[k][tx];
```

```
//同步，以确保在下一次迭代__syncthreads()中加载M和N的///  
两个新的//子矩阵之前完成前面的//计算;
```

CUDA代码-保存结果

//得到一个指向P的块子矩阵的指针

矩阵子=得到子矩阵 (P, bx, by);

//将块子矩阵写入设备存储器;

//: 每个线程写入一个元素

SetMatrixElement (Psub, tx, ty, Pvalue);

这段代码在G80上的运行速度约为
45GFL0PS。

©DavidKirk/NVIDIA和文美W。吴
台湾，2008年6月30日-7月2日

缝合一些松散的结局

获取子矩阵 (Md、x、y、宽度
-Md+y*TILE_WIDTH*宽度+x*TILE_WIDTH);

Get矩阵元素 (Mdsb、tx、ty、宽度)
- *(Mdsb+ty*Width+tx));

总结了一个CUDA程序的典型结构

全局变量声明

- `__host__`
- `__device__...__global__`, `__constant__`, `__texture__`

沙包函数原型

- `__global__` 空腔内核一个 (...)
- 浮动手动功能 (...)

沙痂主()

- 在设备上分配内存空间 - `cudaMalloc(&d_GlblVarPtr, 字节)`
- 将数据从主机传输到设备 - `cudaMemcpy(d_GlblVarPtr, h_Gl...`
- 执行配置设置
- 内核调用 - 内核一个 `<<<执行配置>>>(args...);` 重复
从设备到主机的结果 - `cudaMemcpy(h_GlblVarPtr, ...)`
- 可选: 与黄金 (主机计算) 解决方案进行比较

内核空白内核1 (类型args,)

- 变量声明 - `__local__`, `__shared__`

透明地分配给寄存器或本地寄存器的自动变量

- `syncthreads() ...`

隔墙公司 有或起作用

- 浮动手持功能 (`intinVar...`);