CUDA编程基础培训

AMAX技术工程师
刘耀卿

资料整理
火龙果软件
荐

CUDA背景介绍

GPU架构

CUDA编程模型

CUDA存储系统

CUDA程序优化

资料整理
火龙果软件
荐

背景介绍

- 单线程串行计算发展遭遇瓶颈
  - 追求最小延迟和单个线程性能，以缓存和控制逻辑减小访存延迟，可以完成各种计算任务
  - 提升频率受材料和热设计功耗限制，频率近几年没有显著提升
  - 依靠大块缓存减小访存延迟，使用大量晶体管增加缓存并不能直接提高计算能力
  - 通过分支预测，乱序执行，超线程等手段挖掘指令级并行控制逻辑开销巨大，提升空间有限
  - 通过SSE，MMX等SIMD指令级提升并行处理能力SIMD指令级宽度有限，带宽不足

资料整理
火龙果软件
荐

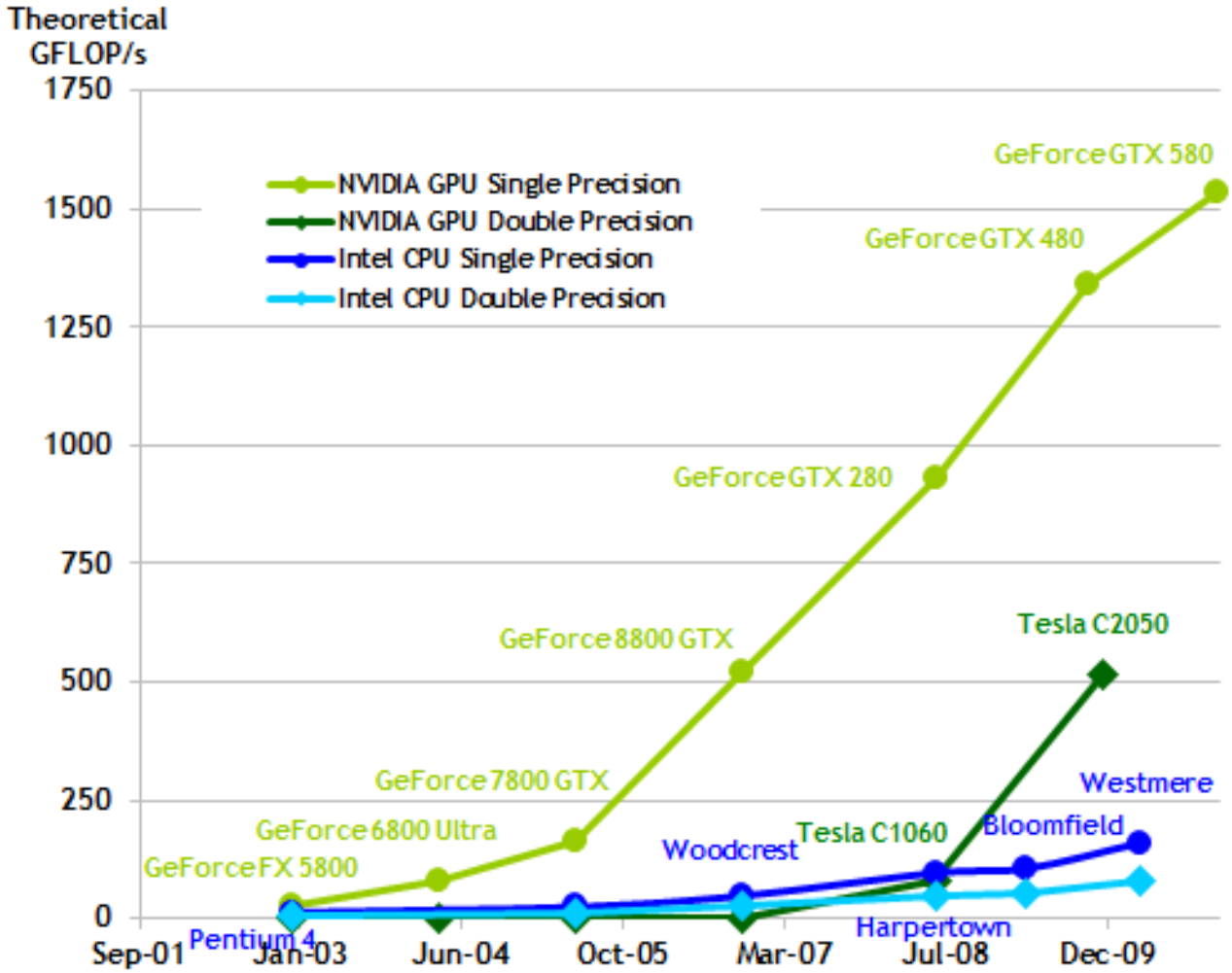
背景介绍

- GPU通用计算已经成熟
  - 追求最大的整体吞吐量，硬件控制线程间切换，以计算量来隐藏访问存储器的延迟，适合数据并行计算
  - 由固定渲染流水线演变为统一架构，适合通用计算
  - 计算精度和可编程性不断提高
  - 大量计算单元，处理能力强
  - 板载显存，电气性能好，位宽大外部存储器带宽大
  - 与专用CPU和向量机相比，GPU供货量巨大易于摊薄成本

资料整理
火龙果软件
荐

背景介绍

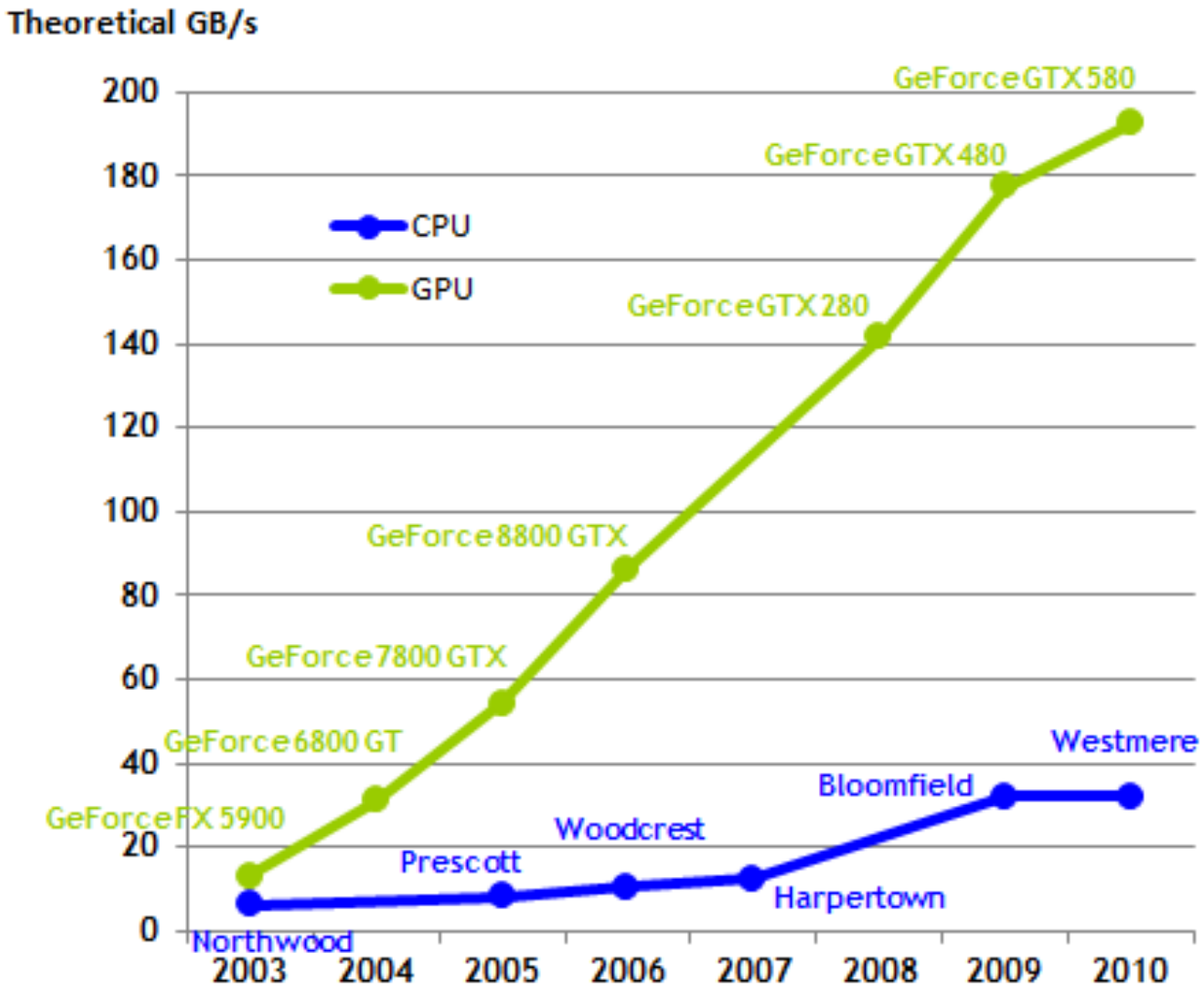
与CPU的浮点运算能力的对比



资料整理
火龙果软件
荐

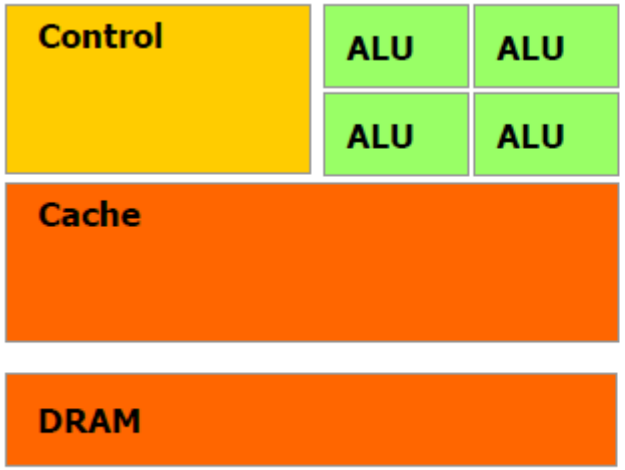
背景介绍

与CPU存储器带宽对比

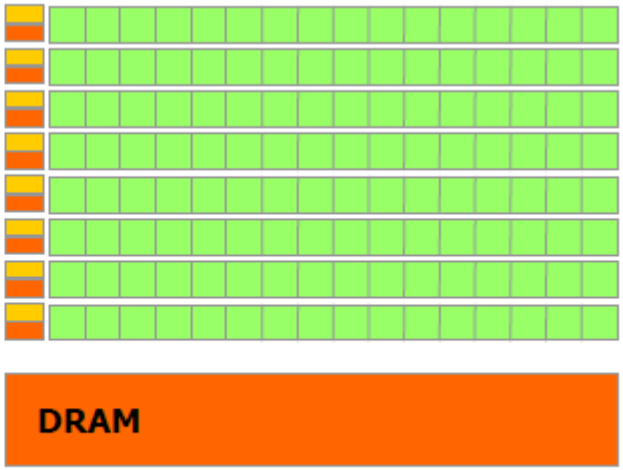


资料整理
火龙果软件
荐

背景介绍



CPU

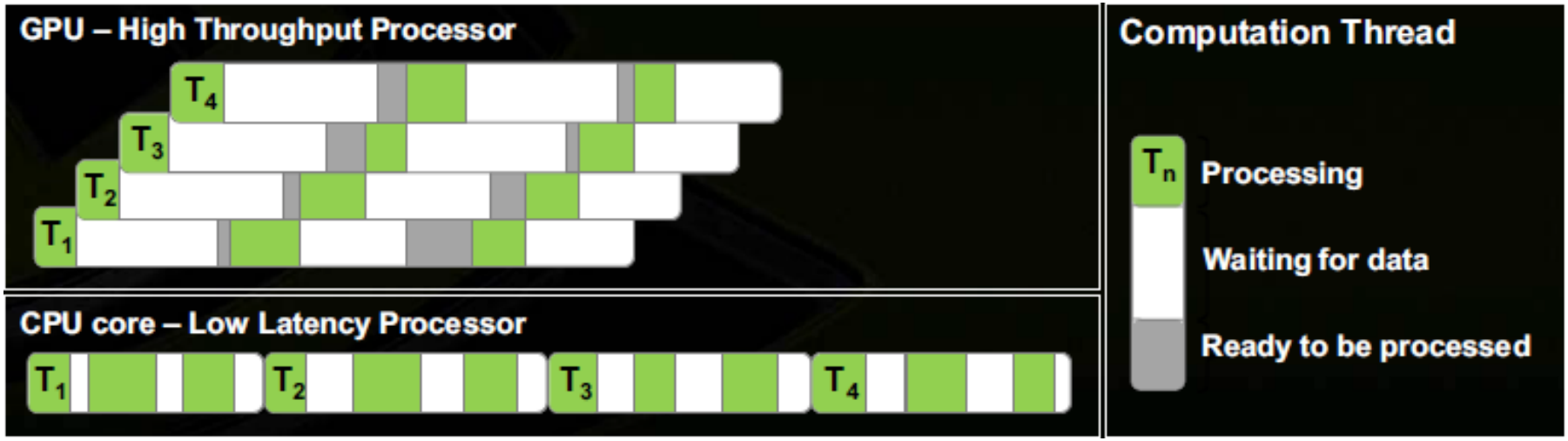


GPU

- CPU
  - 为低延迟访问缓存优化
  - 大量的电路用于乱序执行和分支预测
- GPU
  - 为数据并行、高密度计算优化
  - 比较能够容忍较高的延迟
  - 更多的电路用于计算

资料整理
火龙果软件
荐

背景介绍



- CPU
  - 大的缓存保证线程访问内存的低延迟
  - 内存带宽小，执行单元太少，数据吞吐量小
- GPU
  - 线程多，切换快
  - 访存带宽大、延迟大
  - 大量线程计算掩藏访存延迟

资料整理
火龙果软件
荐

背景介绍

- 处理器需要巨大开销才能变得更快，但比较容易变得更宽
- CPU向多核化，GPU向众核化发展
- GPU-CPU融合，CPU矢量单元将经历一次更新
- 无论是CPU还是GPU，都需要更新编程思考方式，目前，并行是共同的方向

资料整理
火龙果软件
荐

CUDA背景介绍

GPU架构

CUDA编程模型

CUDA存储系统

CUDA程序优化

资料整理
火龙果软件
荐

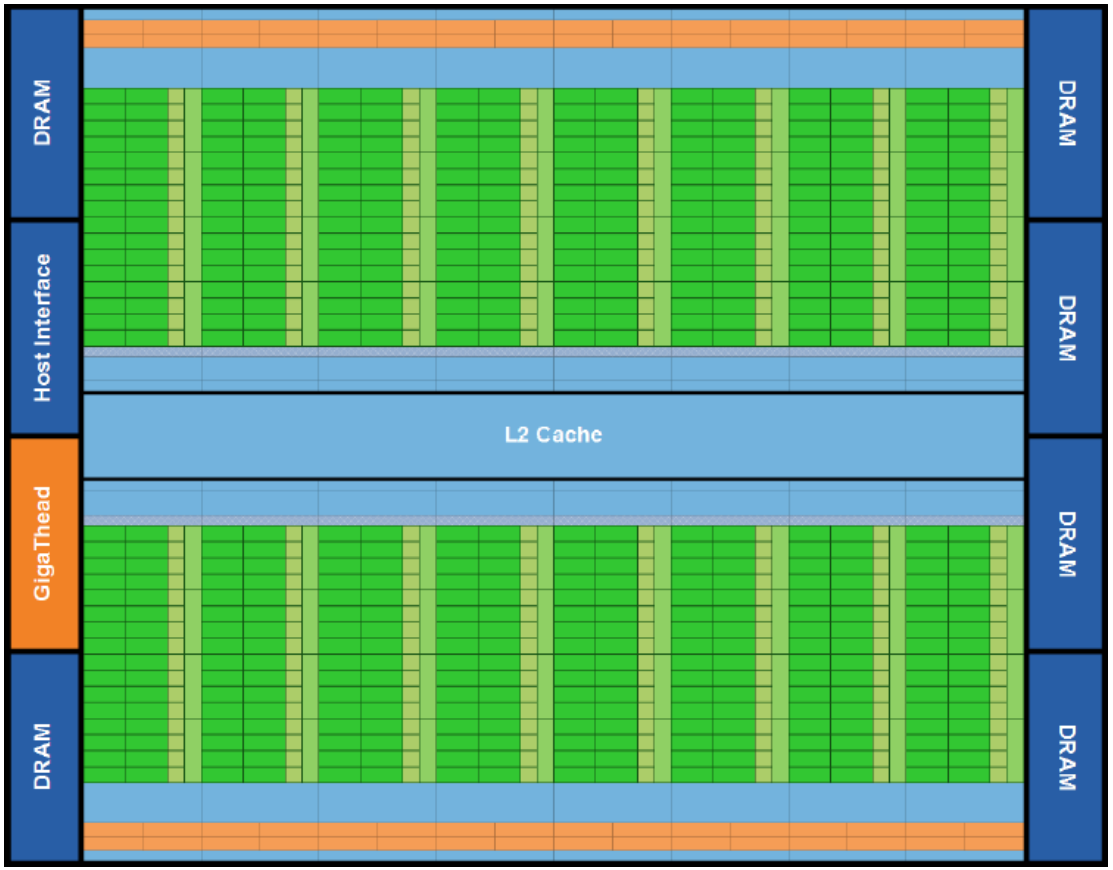
GPU架构

Global memory（全局内存）

- 类似CPU服务器中的内存
- 6个64位内存控制器

Streaming multiprocessors (SMs，流多处理器)

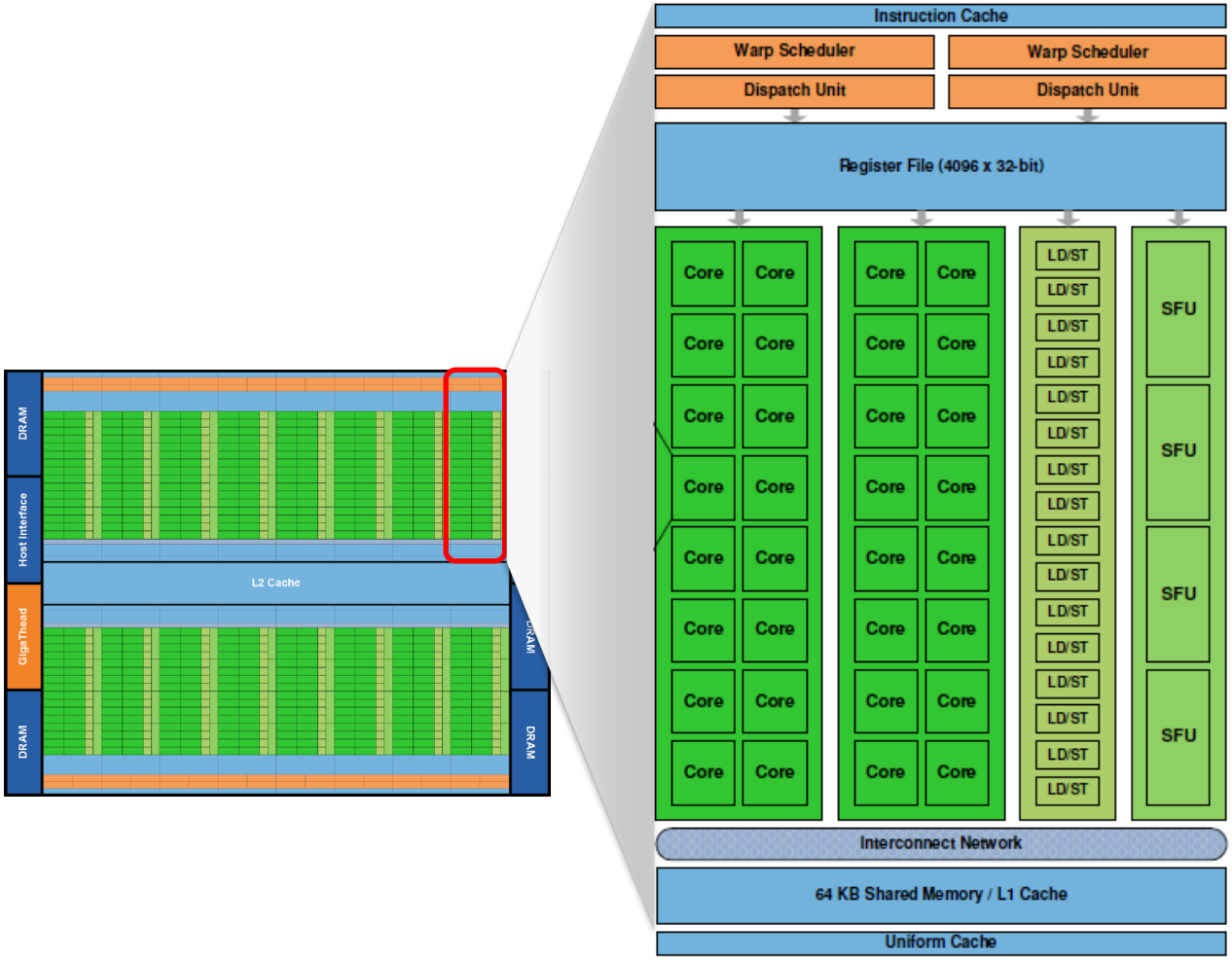
- 16个SM
- 真正的计算单元
- 每个SM拥有自己的控制单元，寄存器，执行流水线，缓存



资料整理
火龙果软件
荐

Streaming Multiprocessor (SM)

- 32 CUDA Cores per SM
  - 32 fp32 ops/clock
  - 16 fp64 ops/clock
  - 32 int32 ops/clock
- 2 warp 调度器
  - 最高同时支持 1536 线程
- 4 special-function units（SFU）
- 64KB shared mem + L1 cache
- 32K 32-bit 寄存器

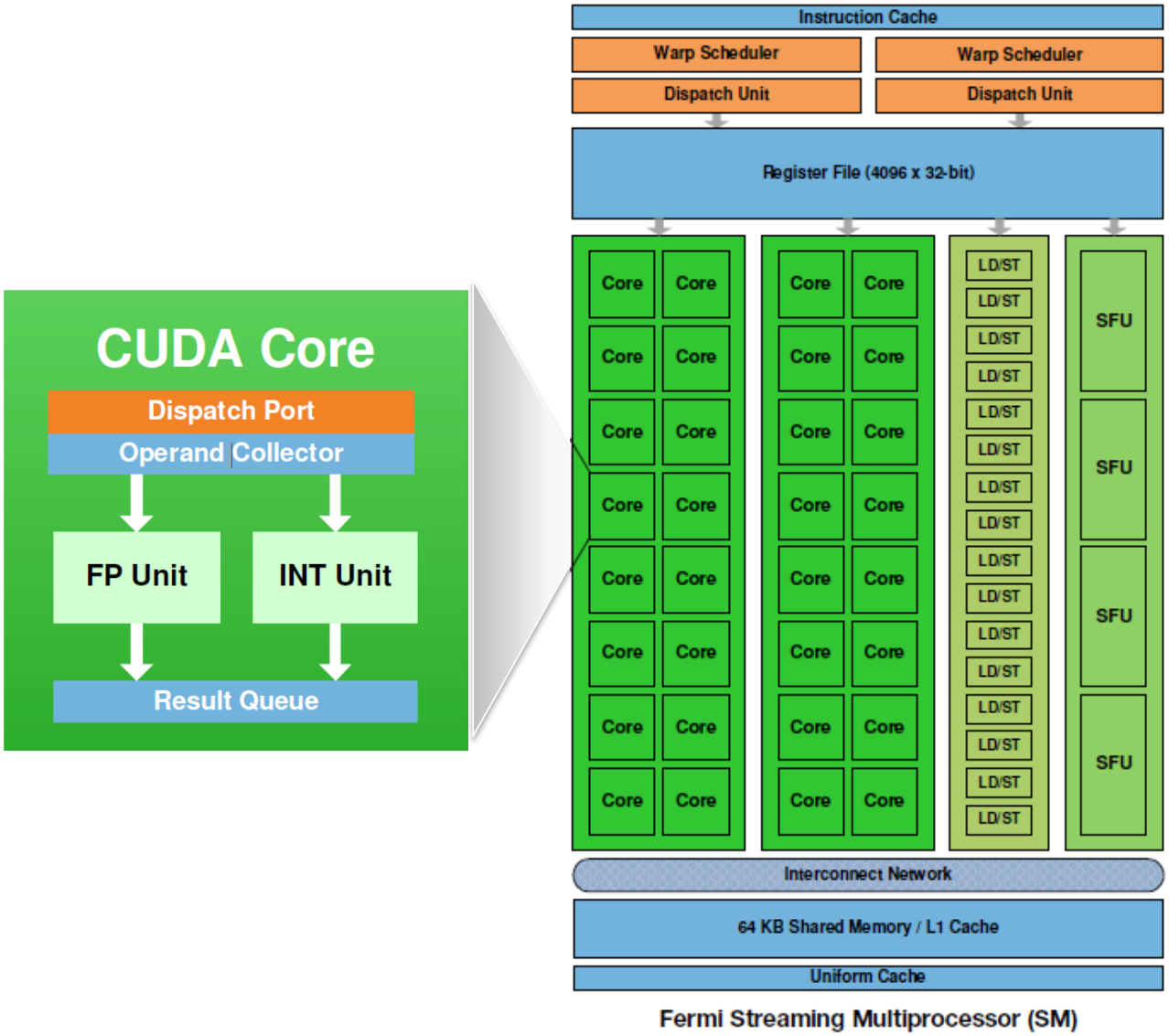


Fermi Streaming Multiprocessor (SM)

资料整理
火龙果软件
荐

CUDA Core

- 浮点单元
  - IEEE 754-2008 浮点标准
  - 支持Fused multiply-add (FMA)指令
- 整数单元
  - 逻辑指令，移动、比较指令等
  - 所有指令都支持32位
  - 为64位及扩展精度优化



资料整理
火龙果软件
荐

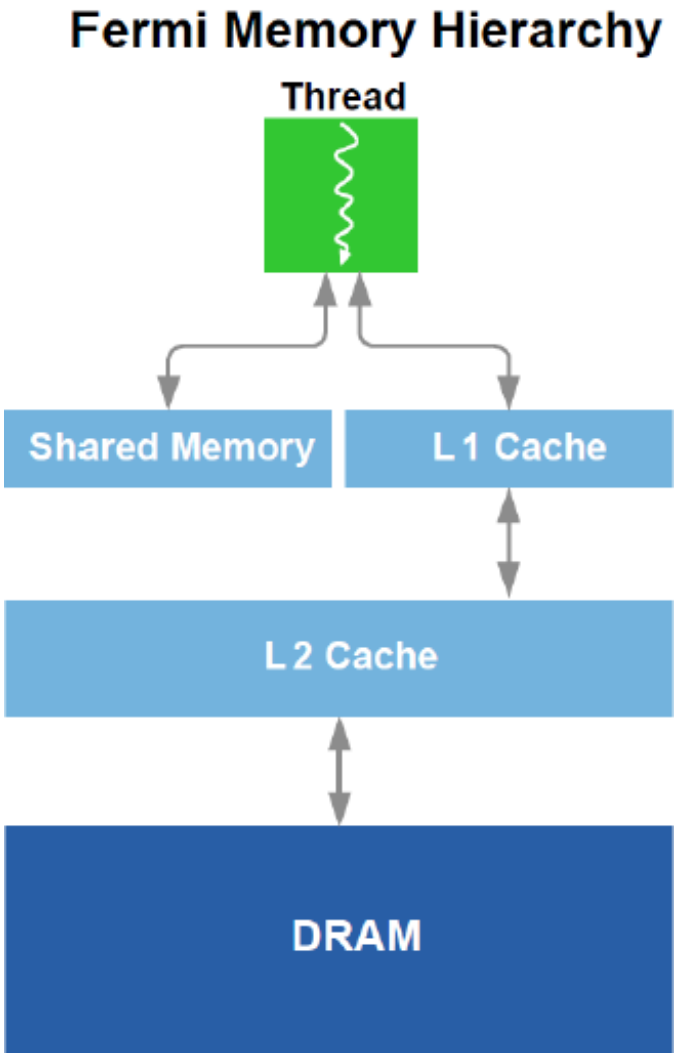
存储子系统

- Global Memory(全局内存）
  - 可以被CPU和GPU访问
  - 当前最大支持6 GB
  - 带宽最大支持150GB/s（GDDR5）
  - 支持ECC的开关（仅限于Quadro 和 Tesla 产品）
- ECC 保护
  - DRAM
  - 支持GDDR5 显存
  - 所有主要的内部存储器都支持ECC
  - 寄存器，L1 cache, L2 cache

资料整理
火龙果软件
荐

存储子系统

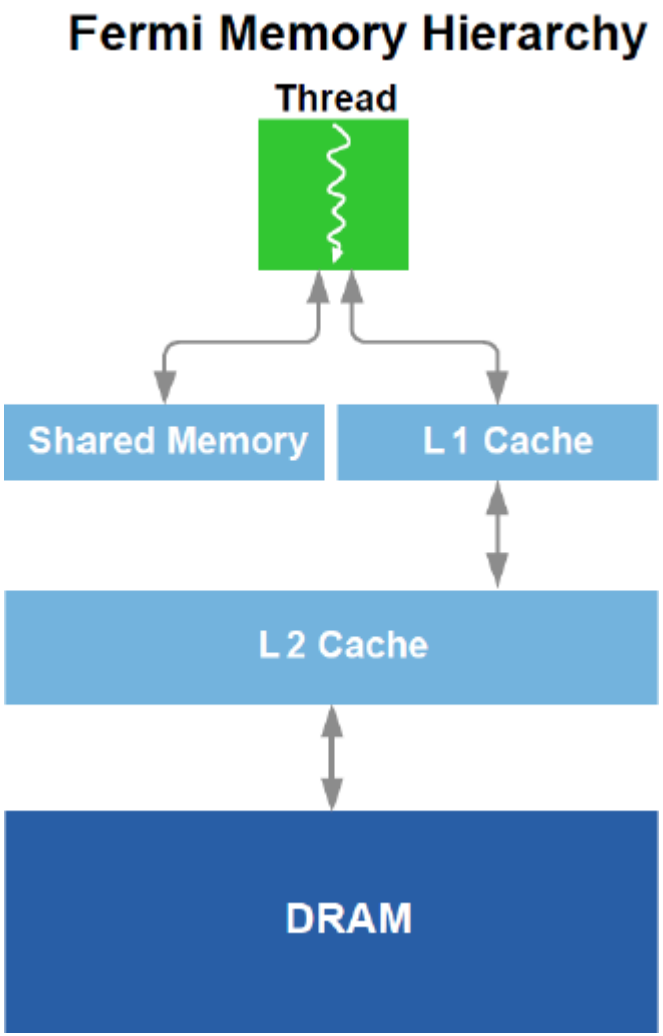
- Shared memory（共享内存）
  - 用户管理便签
    - 硬件不会修改，直到线程覆盖
  - 16 或 48KB / SM (L1 和共享内存共享 64KB)
  - 总带宽: 1.03 TB/s



资料整理
火龙果软件
荐

存储子系统

- L1（一级缓存）
  - 单个SM共享
  - 16 或 48KB / SM, 可以通过程序选择
  - 硬件管理
  - 总带宽: 1.03 TB/s
- L2（二级缓存）
  - 768KB
  - 所有SM共享



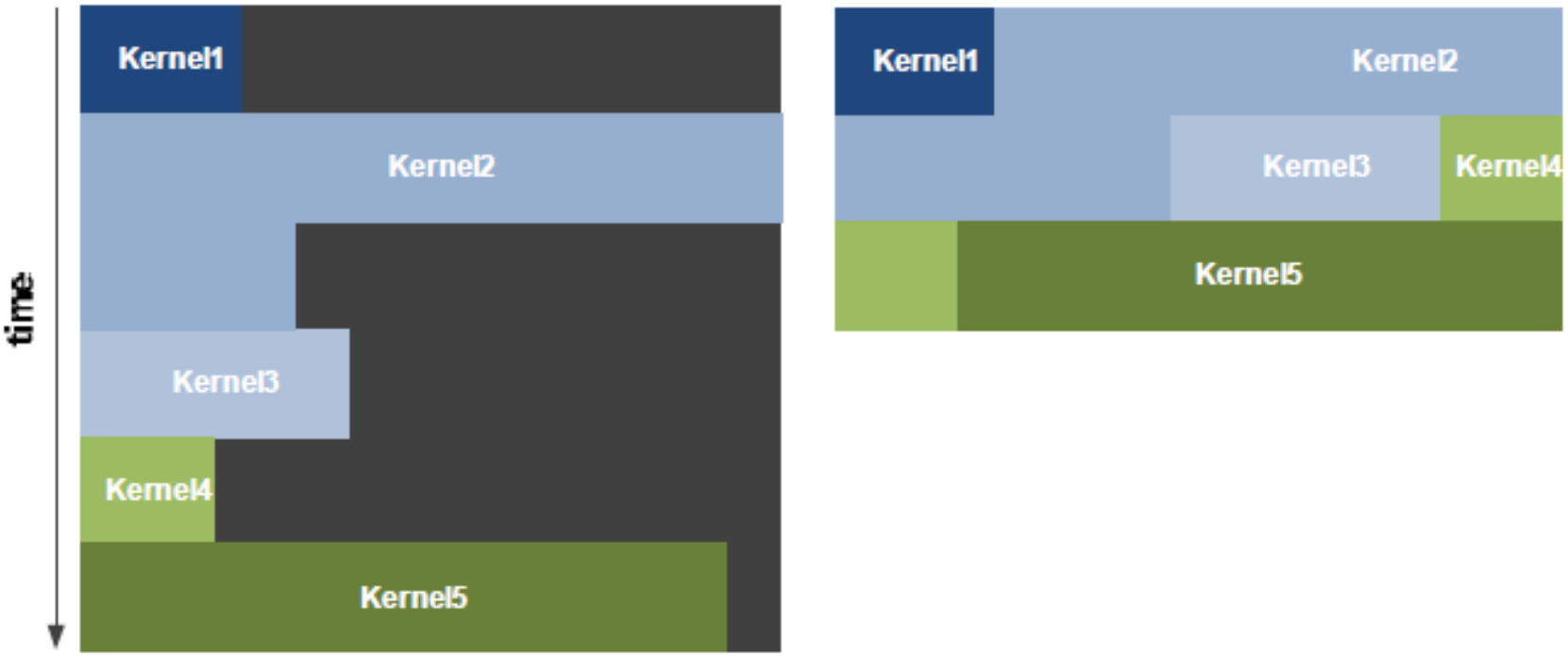
资料整理
火龙果软件
荐

Fermi架构改进

- 32 SP/SM
- 双warp-schduler
- 64KB shared memory/L1 cache,可以配置为48KB shared/16KB L1或者16KB shared/48KB L1
- 支持ECC
- 增加二级缓存（L2 Cache）
- 支持多kernel并发执行
- 统一地址空间，完全支持C++
- 支持64位寻址空间，最大支持1TB显存

资料整理
火龙果软件
荐

Fermi架构改进

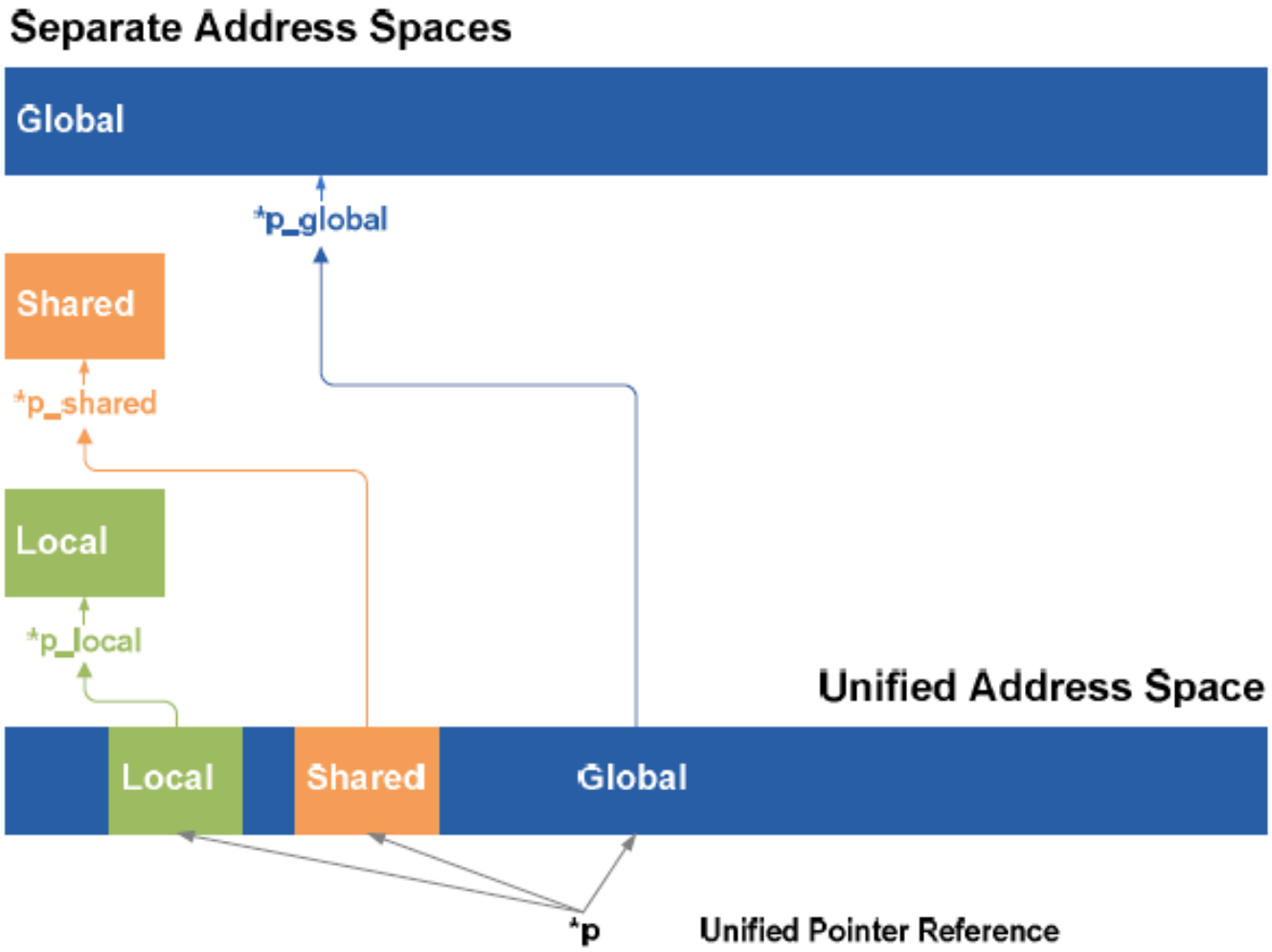


Serial Kernel Execution

Concurrent Kernel Execution

资料整理
火龙果软件
荐

Fermi架构改进



资料整理
火龙果软件
荐

三代Tesla产品的对比

GPU	G80	GT200	Fermi
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Double Precision Floating Point Capability	None	30 FMA ops / clock	256 FMA ops /clock
Single Precision Floating Point Capability	128 MAD ops/clock	240 MAD ops / clock	512 FMA ops /clock
Warp schedulers (per SM)	1	1	2
Special Function Units (SFUs) / SM	2	2	4
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB
L2 Cache (per SM)	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit

资料整理
火龙果软件
荐

CUDA背景介绍

GPU架构

CUDA编程模型

CUDA存储系统

CUDA程序优化

资料整理
火龙果软件
荐

早期GPGPU的编程模式

- 早期GPGPU主要应用于信号成像和图像处理
- 使用图形学API进行计算
- 将数据打包为纹理，将对数据的计算编写为渲染程序，“欺骗”GPU进行计算
- 程序员必须精通图像处理和专业领域内的算法，开发难度很大
- 编程模型灵活性不足，不能使用更有效率的算法
- API成熟可靠，通用性较好，仍然适合消费级应用程序开发

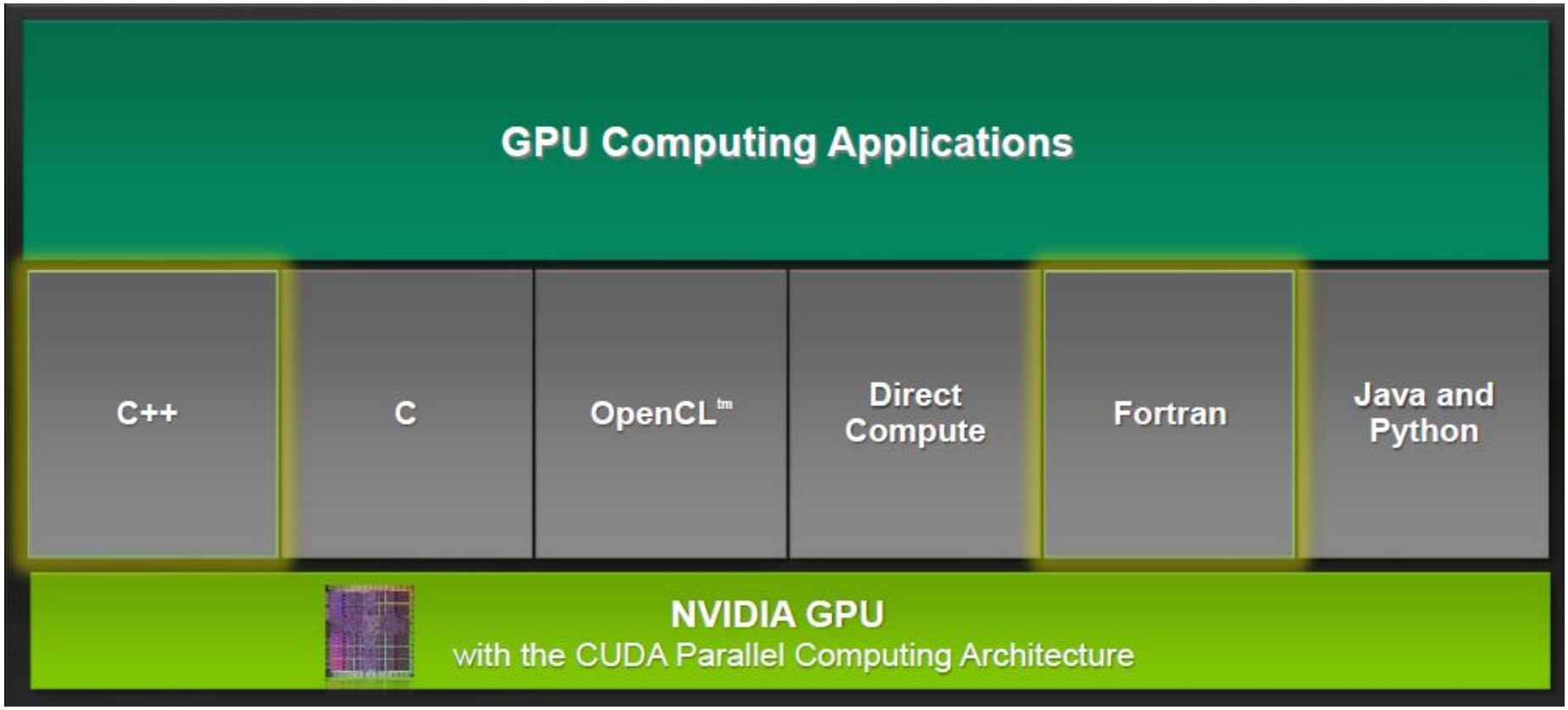
资料整理
火龙果软件
荐

CUDA架构

- Computing Unified Device Architecture，采用统一处理架构，不需要借助图形学API进行GPU通用计算的软硬件架构
- 引入了片内共享存储器，支持随机写入和线程间通信
- 软件经历了1.x, 2.x, 3.x,4.x四次版本更新
- 硬件从Tesla架构（G80/G8x/G92/G9x GPU, 计算能力1.0/1.1），改进到Tesla II架构(GT21x/GT200,计算能力1.2、1.3)，再革新为Fermi架构（GF100、GF10x等，计算能力2.x）
- 最新的软件提供了对最新硬件特性的支持，同时也对老版本硬件的功能和性能进行了提升

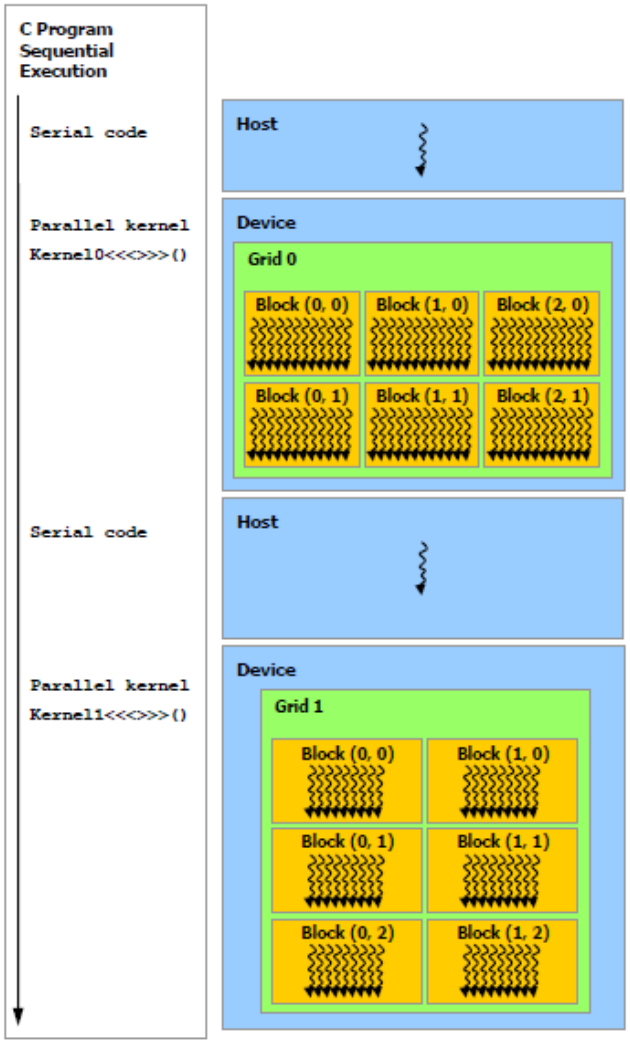
资料整理
火龙果软件
荐

CUDA并行计算架构



资料整理
火龙果软件
荐

CUDA的执行模型



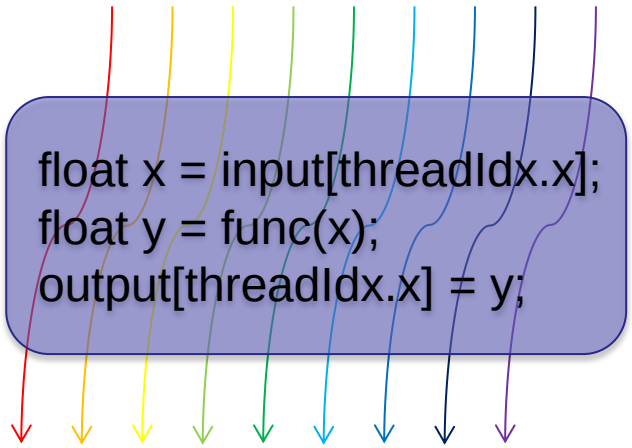
Serial code executes on the host while parallel code executes on the device.

- 将CPU作为主机端(Host)，而GPU作为协处理器(Coprocessor)，或者设备（Device）
- 程序的串行部分在主机端（CPU）运行
- 并行部分在设备端（GPU）运行，叫做Kernel

资料整理
火龙果软件
荐

CUDA的执行模型--kernel

- Kernel 是一个由cpu调用在gpu上执行的函数，通常有许多线程并行执行
- 所有的线程执行相同的代码，可以走不同的路径
- 每个线程有一个线程ID，和一个block ID



资料整理
火龙果软件
荐

一个简单的例子

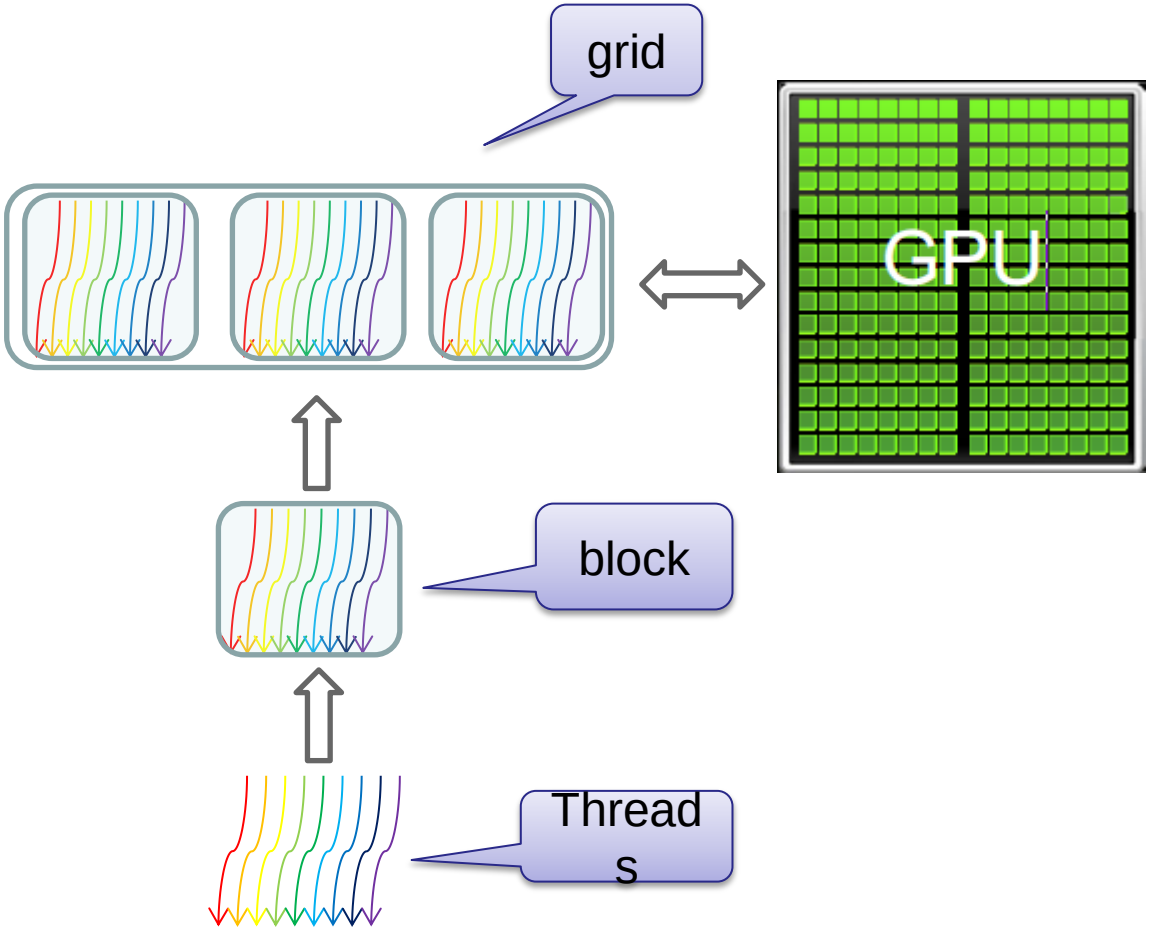
```
// kernel 定义
__global__ void vecAdd(float* A, float* B, float* C){
……
}

int main (){
//Kernel 调用
vecAdd<<<1,N>>>(A, B, C);
}
```

资料整理
火龙果软件
荐

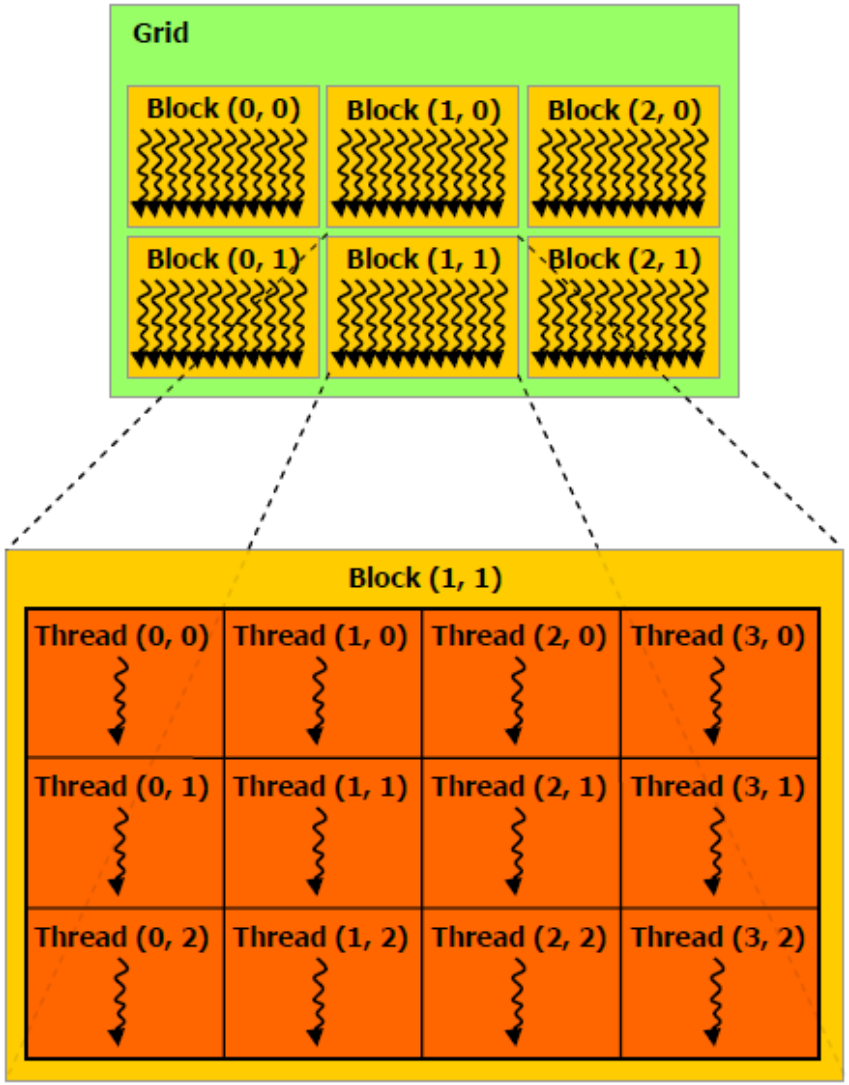
CUDA kernel

- Kernel对应grid
- 很多block组成grid
- 很多的thread组成block



资料整理
火龙果软件
荐

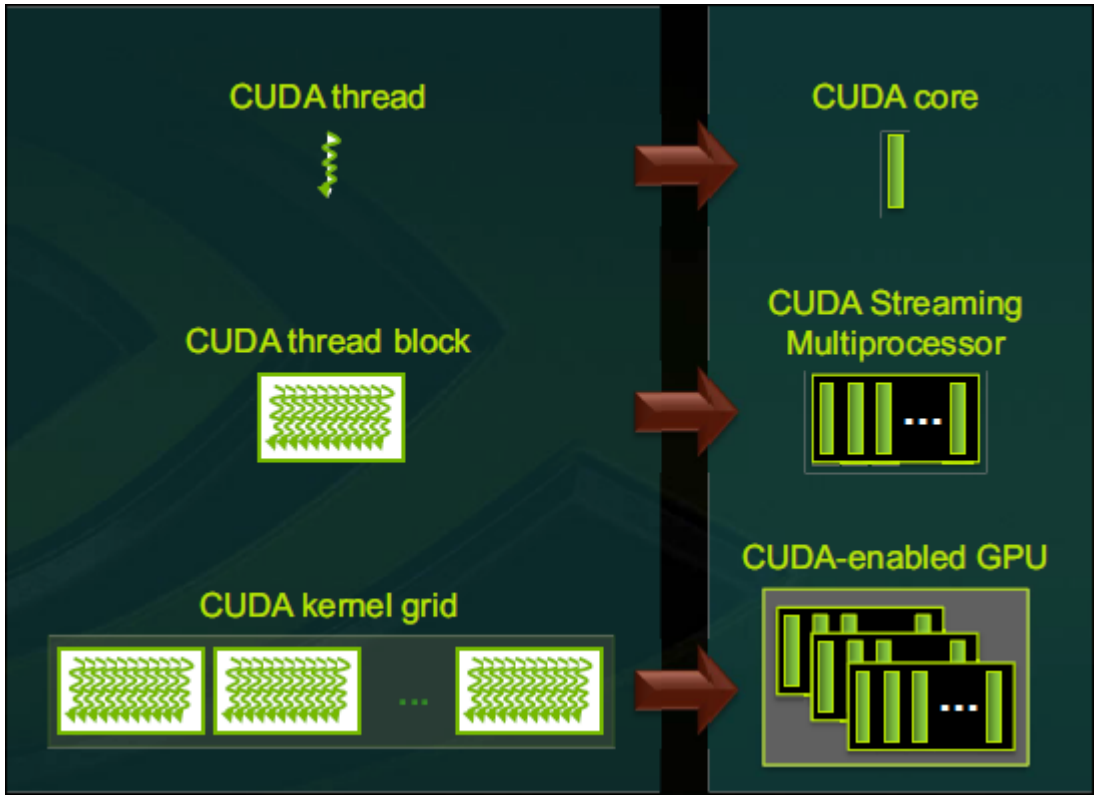
CUDA的执行模型



- Kernel以包含大量的线程的Grid形式 存在，每一个线程执行相同的程序
- Grid由block组成，block内的线程间可以高效通信
- Block和thread拥有唯一的ID，用于控制thread操作不同的数据，或者执行不同分支

资料整理
火龙果软件
荐

CUDA的执行模型



- 每个线程在一个核心上执行
- 每个block在一个SM上执行，不能迁移到其它SM上
  - 如果SM的内存资源允许，几个block可以在一个SM上执行
- Grid运行在SPA上对应gpu

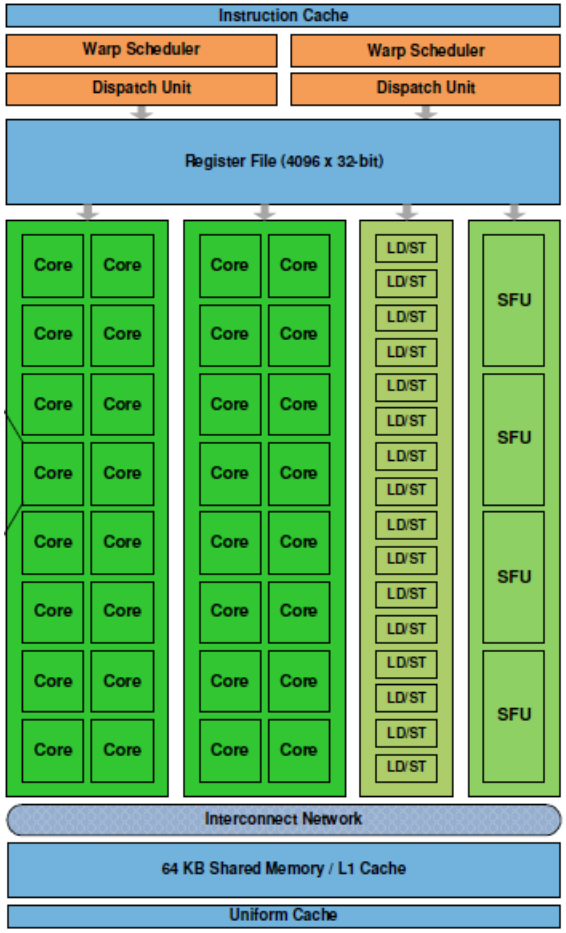
资料整理
火龙果软件
荐

CUDA的执行模型

- 对grid和block的限制
  - Block和grid以dim3格式定义，有x y z三个维度
  - Grid的z维度必须为1，x, y维度上不能超过65535
  - 对1.x架构硬件，block的x和y维度不能超过512，z维度不能超过64，同时x*y*z不能超过512
  - 对2.0架构硬件，block的x和y维度不能超过1024，z维度不能超过64，同时x*y*z不能超过1024

资料整理
火龙果软件
荐

Block

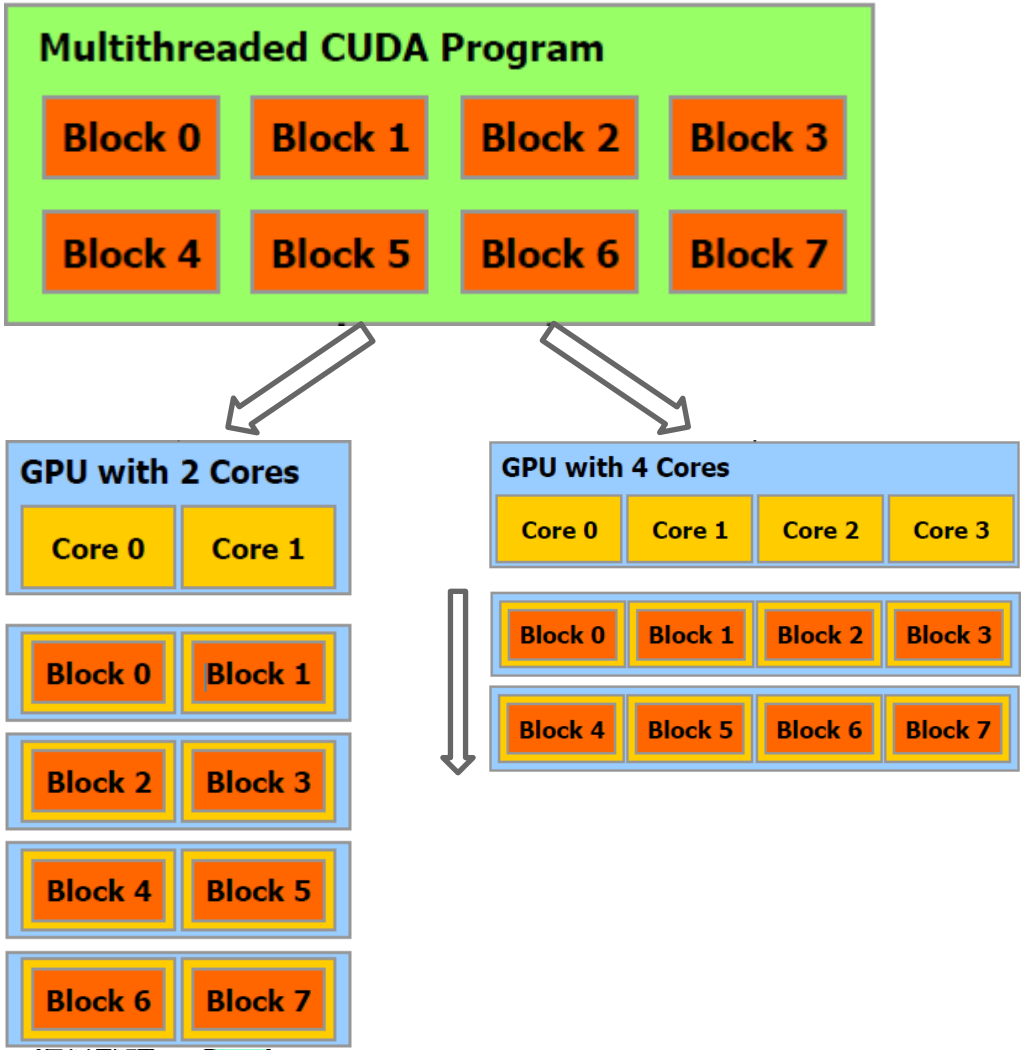


Fermi Streaming Multiprocessor (SM)

- 线程block允许合作
  - 相互协作，加载/存储显存里所有线程都要用到的连续数据
  - Block内的线程共享结果或共同产生单一的结果
  - 可以互相同步

资料整理
火龙果软件
荐

Block



- 线程block有很好的扩展性
  - Block可以任何顺序执行，同时执行或顺序执行
  - Block间的独立性提供了扩展性：
    - 内核函数可以扩展到任意数目的SM

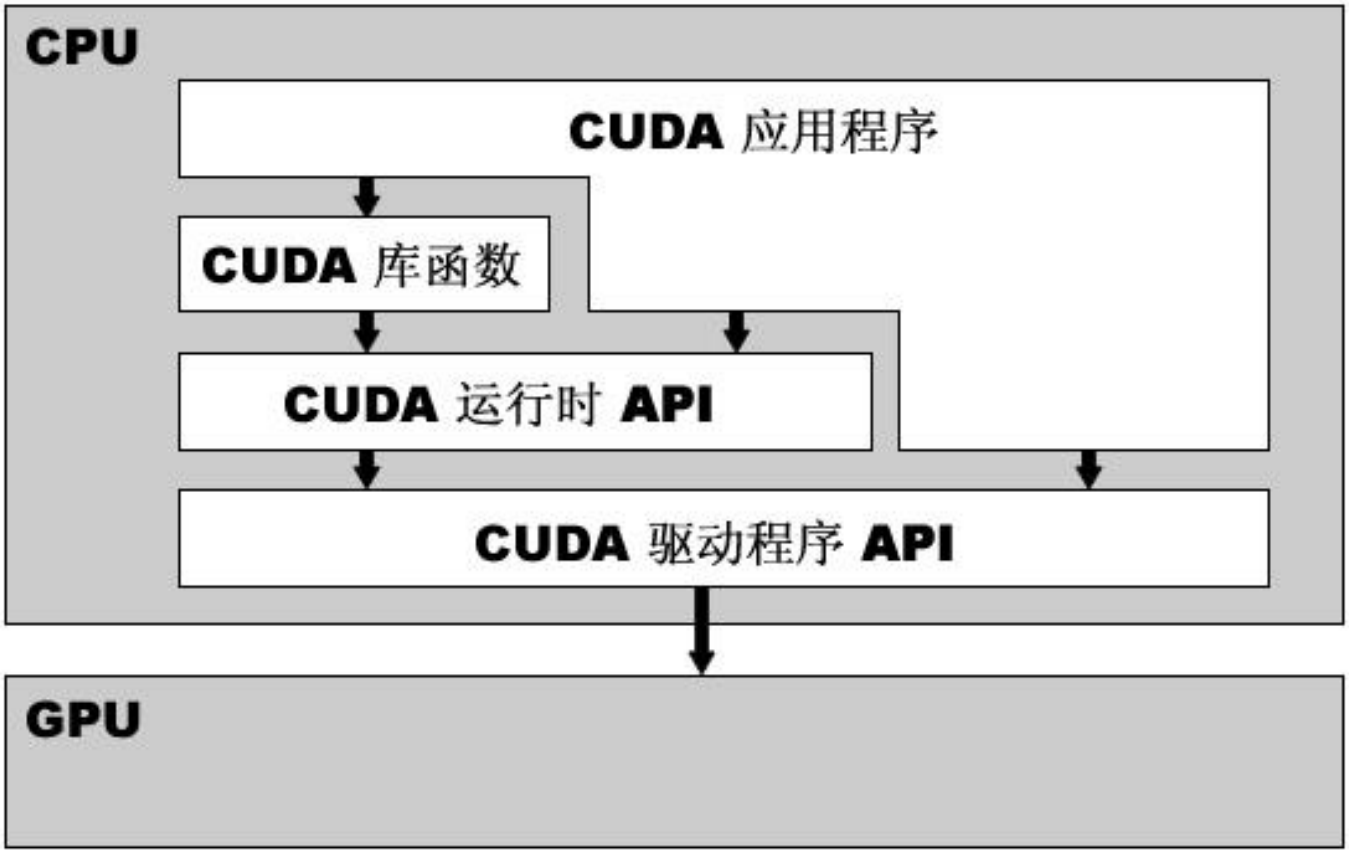
资料整理
火龙果软件
荐

warp

- Warp并不存在于抽象的CUDA编程模型，而是由硬件决定的，对性能影响较大
- 在1.x和2.x架构硬件中，一个warp由同一block内相邻的32个线程组成。如0，1，2，……31为1 个warp。
- 同一warp内的线程可以认为是“同时”执行的，不需要进行同步也能通过shared memory进行通信
- 分支以warp为单位进行
- 以warp为单位考虑对存储器的访问的优化，可以获得更高性能

资料整理
火龙果软件
荐

CUDA软件体系



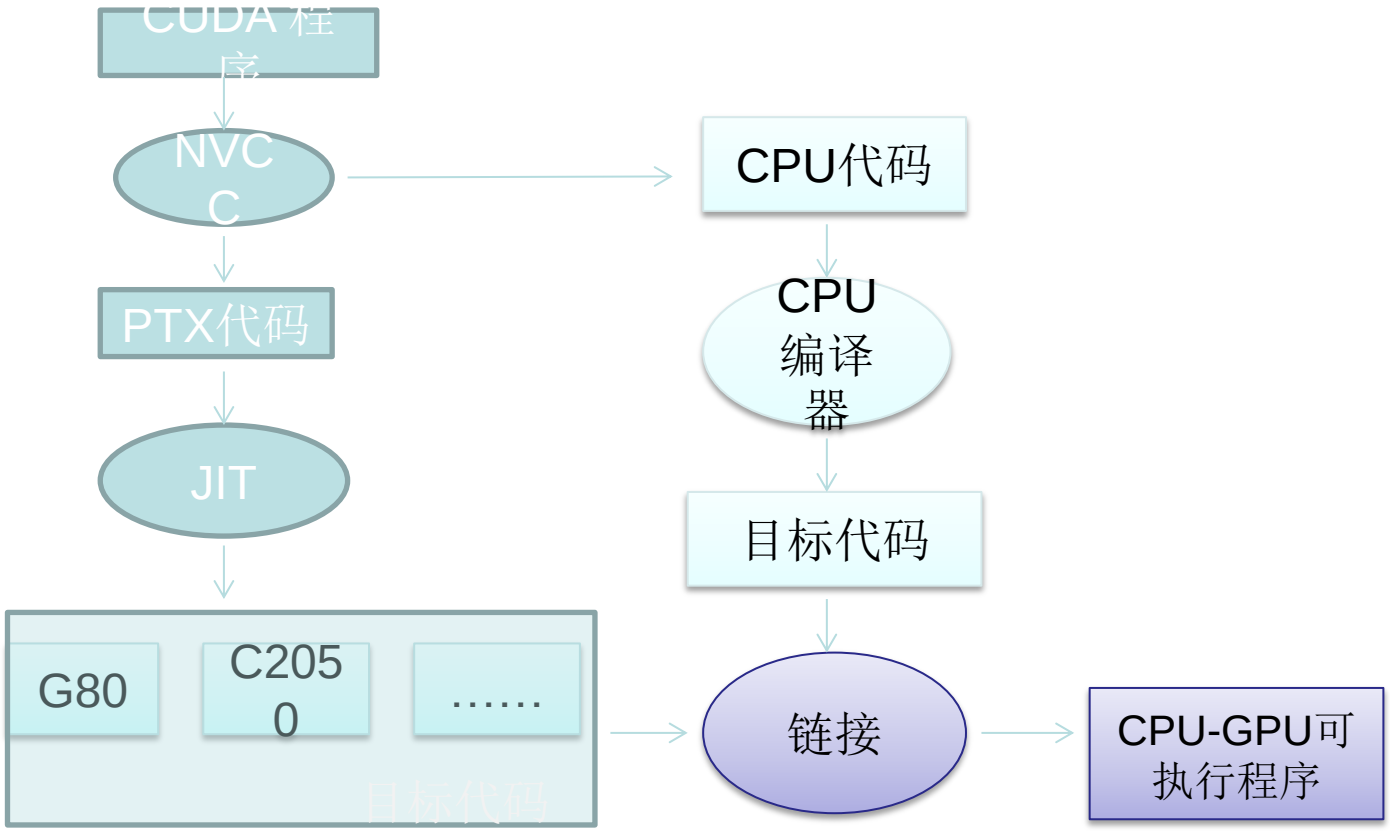
资料整理
火龙果软件
荐

CUDA编译流程

- 首先将CPU端代码与GPU端分离（cudafe）
- 将CPU端代码交给其他编译器编译
- 将GPU端代码使用nvcc编译器编译为类似汇编的ptx代码或者二进制的cubin代码
- 再将编译好的CPU端代码和GPU端代码链接（runtime API）到同一个程序中
- Driver API在运行时显式加载另一个文件中的GPU端代码

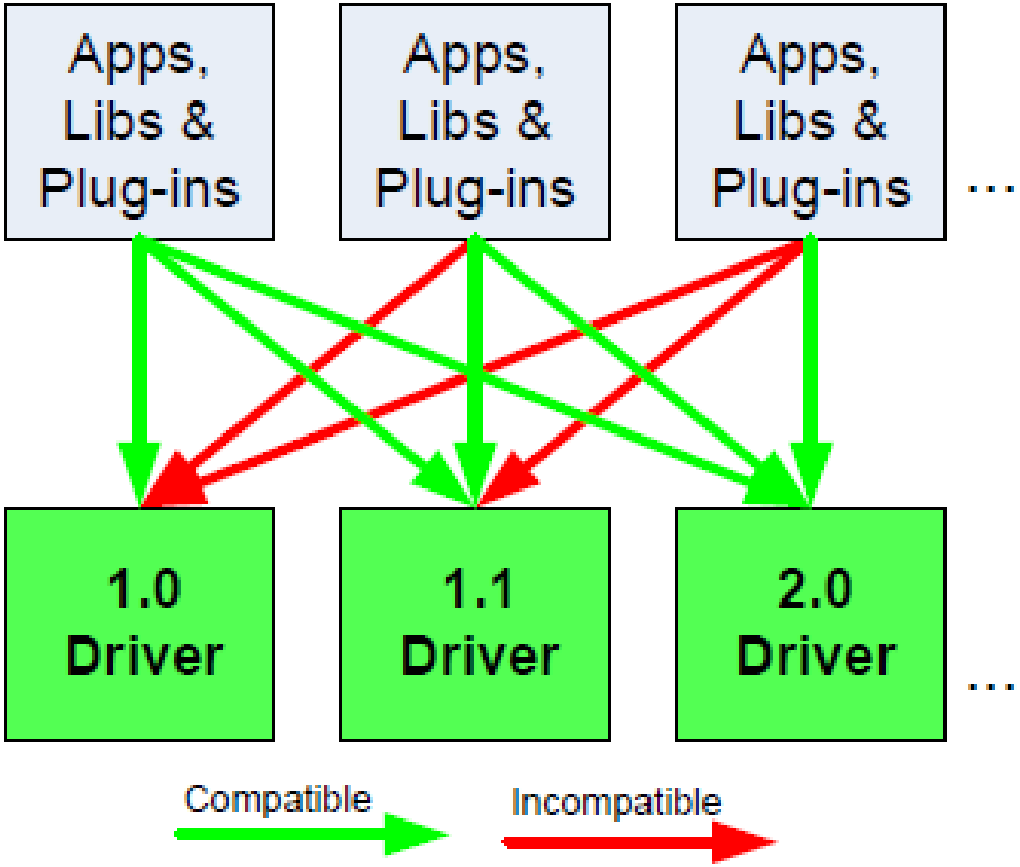
资料整理
火龙果软件
荐

CUDA编译流程



资料整理
火龙果软件
荐

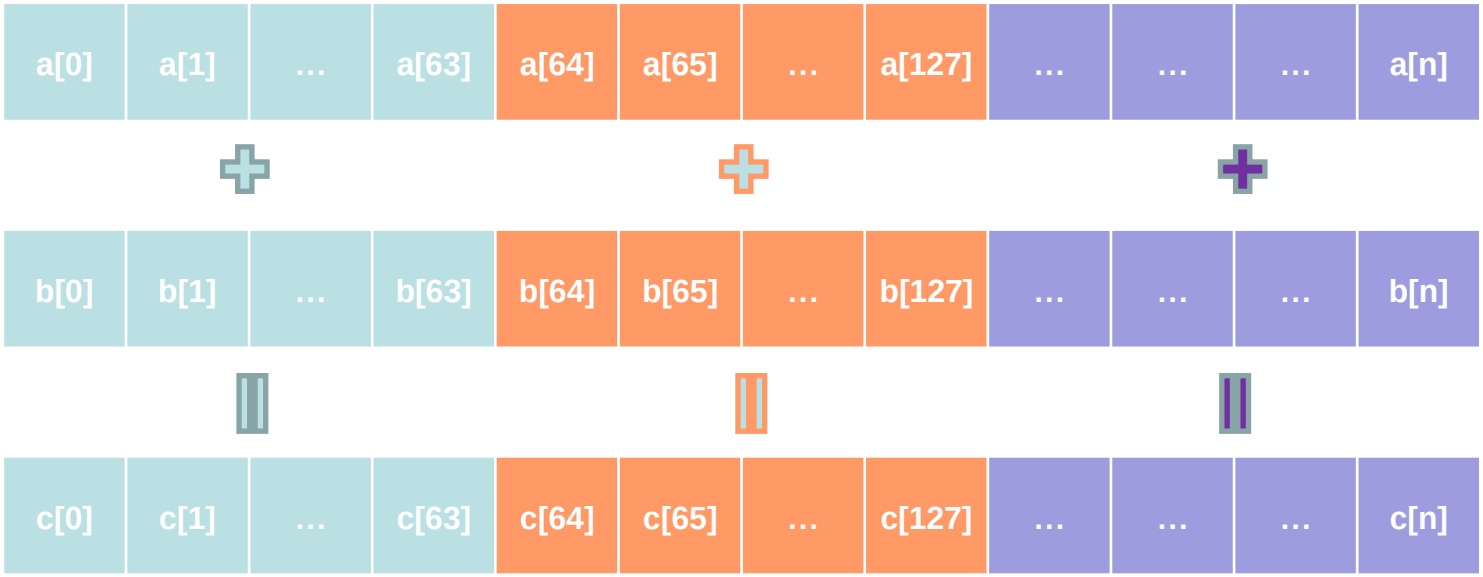
CUDA程序版本兼容性



- 新的驱动支持旧的cuda程序
- 旧的驱动不支持在新驱动上编译的程序

资料整理
火龙果软件
荐

例子：向量加



c[i] = a[i] + b[i]

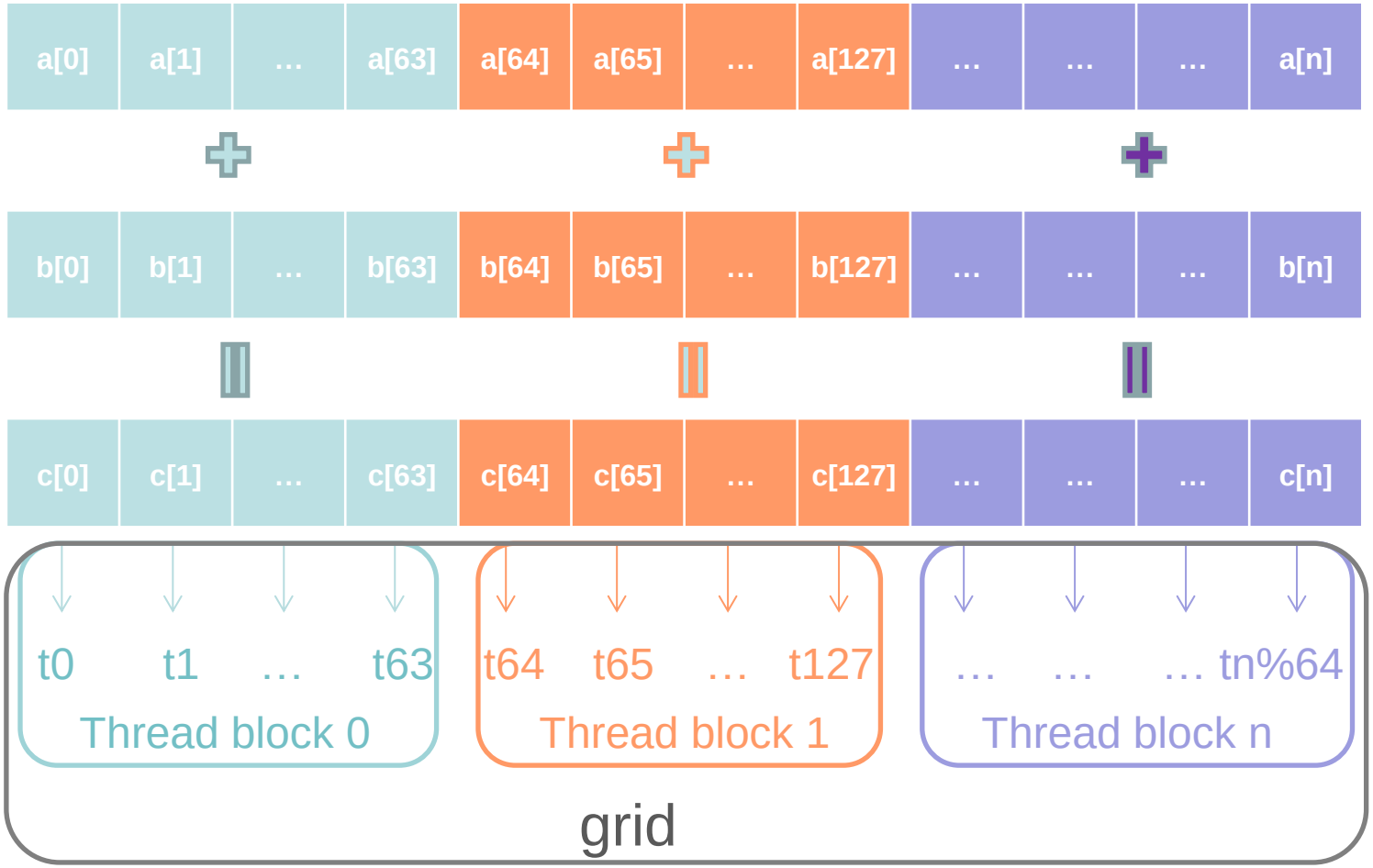
资料整理
火龙果软件
荐

CPU代码

```
//CPU program
//sum of two vectors a and b
void vecAdd(float *a, float *b, float *c, int
N)
{
for (int idx = 0; idx<N; idx++)
c[idx] = a[idx] + b[idx];
}
void main()
{
...
vecAdd(a, b,c, N);
…
}
```

资料整理
火龙果软件
荐

例子：向量加



资料整理
火龙果软件
荐

CUDA代码

```
//CUDA program
//sum of two vectors a and b
__global__ void add_gpu(float *a, float *b, int N)
{
int idx =blockIdx.x* blockDim.x+ threadIdx.x;
if (idx < N) //Avoid overflow!
a[idx] += b[idx];
}
void main()
{
…..
dim3 dimBlock (64);
dim3 dimGrid( ceil( N / 64));
fun_add<<<dimGrid, dimBlock>>>(a, b, N);
}
```

资料整理
火龙果软件
荐

CUDA的C扩展

- 函数限定符
- __global__
  - GPU Kernel代码，由CPU发起，返回void，不能由其它GPU Kernel调用
- __device__
  - 由GPU Kernel调用的函数，不能由CPU发起
- __host__
  - 在CPU上执行的函数
- __host__和__device__能同时使用
- 引入一些内置函数
- 数学函数，同步函数，测时函数，原子函数等

资料整理
火龙果软件
荐

CUDA的C扩展

- 变量限定符
  - __constant__
    - 常量内存 float my_constant_array[32];
  - __shared__
    - 共享内存 float my_shared_array[32];
- 内置矢量类型
  - char4, ushort3, double2, dim3
    - 由基本的整型或浮点型构成，通过x, y, z, w访问每个分量，在设备端有不同对齐要求
    - 内建变量
      - dim3 gridDim; // Grid尺寸
      - dim3 blockDim; // Block尺寸
      - dim3 blockIdx; // Block ID
      - dim3 threadIdx; // Thread 点

资料整理
火龙果软件
荐

CUDA的C扩展

- 执行配置
  - 引入<<<>>>运算符
    - 指定grid和block的维度
    - 完整的参数配置形式：<<<Dg, Db, Ns, S>>>
      - Dg, 指定grid的维度和尺寸，dim3类型，Dg(Dg.x, Dg.y, 1)
      - Db, 指定block的维度和尺寸，dim3类型，Db(Db.x, Db.y, Db.z)
      - Ns, 可选，指定动态分配的shared memory的大小
      - S, 可选，cudaStream_t类型，初值为0.
  - dim3 grid_dim(100, 50); // 5000个block
  - dim3 block_dim(4, 8, 8); // 每个block包含256个线程

资料整理
火龙果软件
荐

例子

```
__global__ void kernel( int*a )
{
int idx= blockIdx.x*blockDim.x+ threadIdx.x; // Assume blockDim.x = 4
a[idx] = 7;
}
Output: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
__global__ void kernel( int*a )
{
int idx= blockIdx.x*blockDim.x+ threadIdx.x; // Assume blockDim.x = 4
a[idx] = blockIdx.x;
}
Output: 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3
__global__ void kernel( int*a )
{
int idx= blockIdx.x*blockDim.x+ threadIdx.x; // Assume blockDim.x= 4
a[idx] = threadIdx.x;
}
Output: 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3
```

资料整理
火龙果软件
荐

CUDA背景介绍

GPU架构

CUDA编程模型

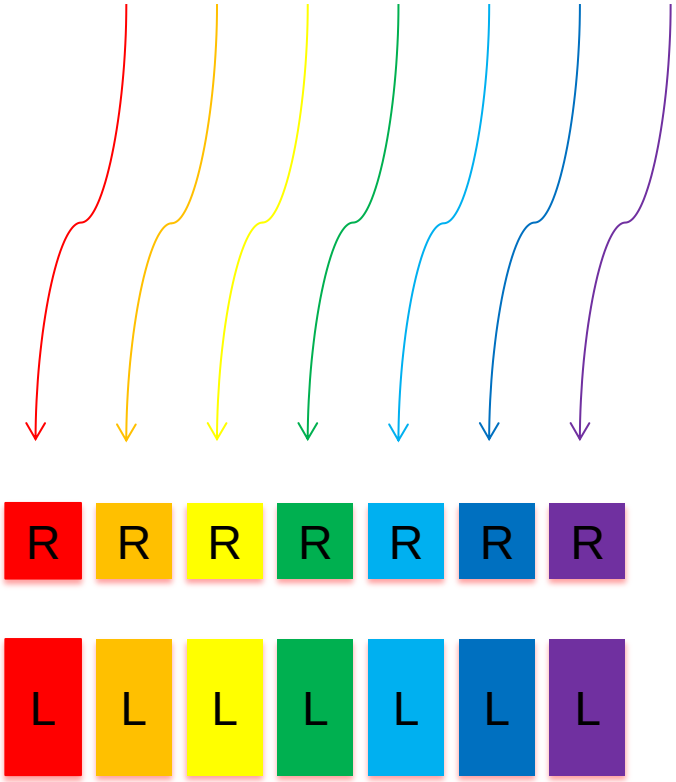
CUDA存储系统

CUDA程序优化

资料整理
火龙果软件
荐

存储器模型

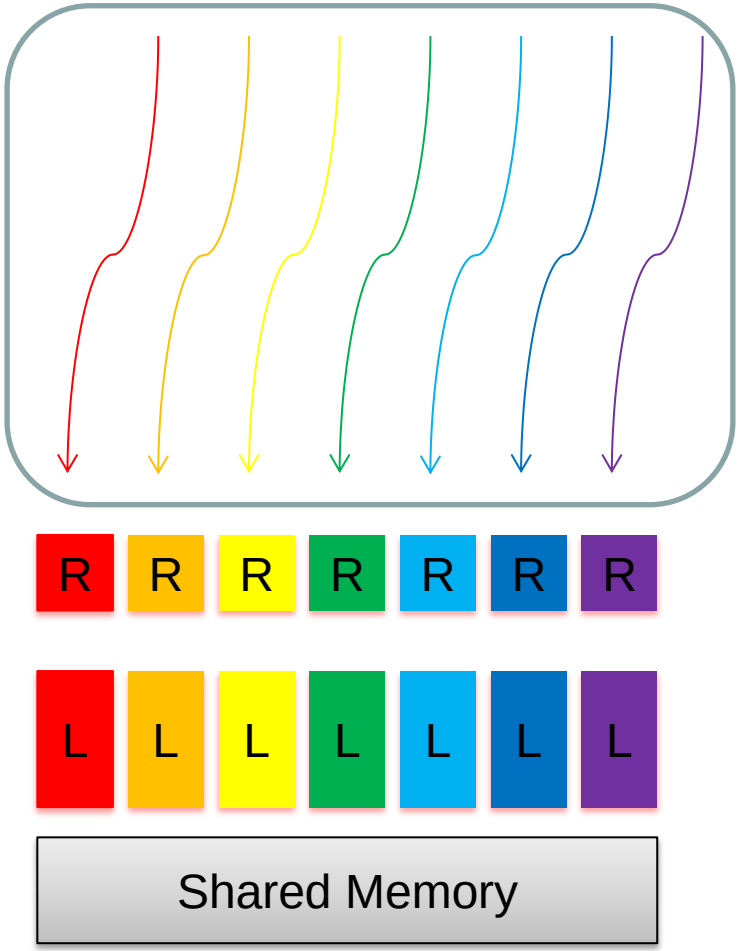
- Registers/Local memory
  - 每个线程私有
  - 能够使用的Register受thread数量和资源问题的限制
  - 2.0硬件每SM有32768个寄存器文件，每SP平均1K
  - 通常情况下私有变量存放在寄存器（register），当寄存器资源不足时自动使用显存/Cache保存私有变量
  - 寄存器通常无延迟



资料整理
火龙果软件
荐

存储器模型

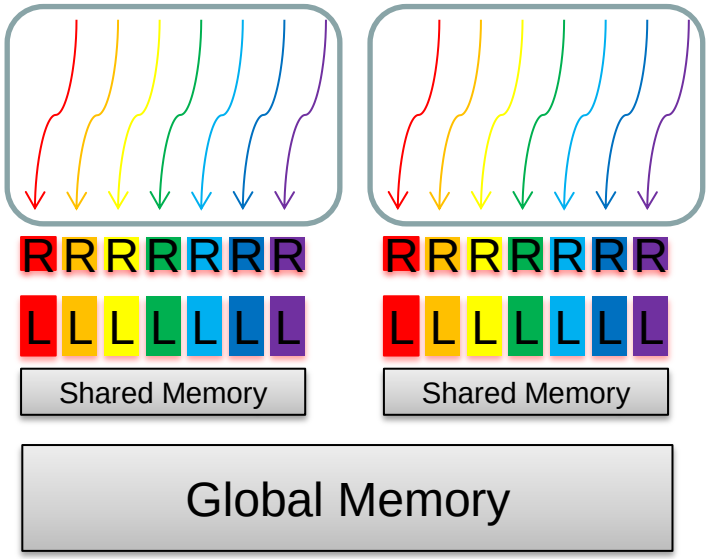
- Shared Memroy
  - Block私有，block线程内不同线程通过shared memory进行通信
  - 通过__shared__前缀定义
  - 带宽较大，延迟很小（无bank conflict时1-2周期）
  - 1.x硬件每SM 16KB shared memory，分为16 bank，bank宽度32bit，全速
  - 2.x硬件每SM 16kB/48kB shared memory, 分为32 bank, bank宽度32bit，半速



资料整理
火龙果软件
荐

存储器模型

- Global memory
  - 通过__device__前缀定义，或者通过API分配
  - 存在于显存中，访问延迟大（400-800周期）
  - 受显存和存储器控制器设计影响，按照一定方式访问时才能可以获得最大带宽
  - 带宽：156GB/s
  - 2.x硬件上有对global memory的cache机制



资料整理
火龙果软件
荐

存储器模型

- Texture memory
  - 定义纹理参考后与CUDA array或者global memory绑定
  - 存在于显存中，有texture cache
  - 利用二维三维数据的数据局部性，访问相邻元素性能较好
  - 可以利用GPU中图形硬件单元进行一些处理，不占用可编程计算性能
  - 在1.x硬件上为只读，而2.x硬件可写

资料整理
火龙果软件
荐

存储器模型

- Constant memory
  - 通过__constant__前缀定义
  - 存在于显存中，访问延迟大（400-800周期）
  - 在1.x硬件上，同一warp访问同一数据才能得到缓存
  - 在2.x硬件上，cache机制与global memory相同
  - 在1.x和2.x硬件上尺寸限制均为64KB

资料整理
火龙果软件
荐

存储器模型

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	†	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

†Cached only on devices of compute capability 2.x.

资料整理
火龙果软件
荐

存储器模型

- Host memory
  - 通过通常的new/malloc定义，或者通过CUDA API提供的函数定义
  - 通过CUDA API(cudaHostAlloc)定义的host memory为page-locked memory，不参与操作系统维护的页交换，具有固定的物理地址，只有page-locked才能实现异步和流
  - Page-locked memory可以通过置标志位，声明为mapped memory, portable memory, write-combined memory（可以同时置多个标志位）
    - Portable memory可以被多个主机端线程共享，可以实现控制不同GPU的CPU线程间的通信
    - Mapped memory物理上存在于内存中，但同时拥有设备端和主机端地址，可以被GPU直接访问，实现zero-copy
    - Write-combined memory可以实现更高速度的主机端到设备端数据传输，但不会被CPU缓存，从CPU端进行操作时速度较低。

资料整理
火龙果软件
荐

GPU存储器操作

- GPU存储器空间分配/释放API
  - cudaMalloc(void ** pointer, size_tnbytes)
  - cudaMemset(void * pointer, int value, size_t count)
  - cudaFree(void* pointer)
- 示例
  - int n = 1024;
  - int nbytes = 1024*sizeof(int);
  - int* d_a = 0;
  - cudaMalloc((void**)&d_a, nbytes);
  - cudaMemset(d_a, 0, nbytes);
  - cudaFree(d_a);

资料整理
火龙果软件
荐

GPU存储器操作

- GPU存储器复制API
- cudaMemcpy(void *dst, void *src, size_tnbytes, enum cudaMemcpyKinddirection);
  - 同步执行（能阻塞CPU线程）
  - 若前一kernel未执行完毕，则等待
  - enum cudaMemcpy*Kind*
    - cudaMemcpyHostToDevice
    - cudaMemcpyDeviceToHost
    - cudaMemcpyDeviceToDevice

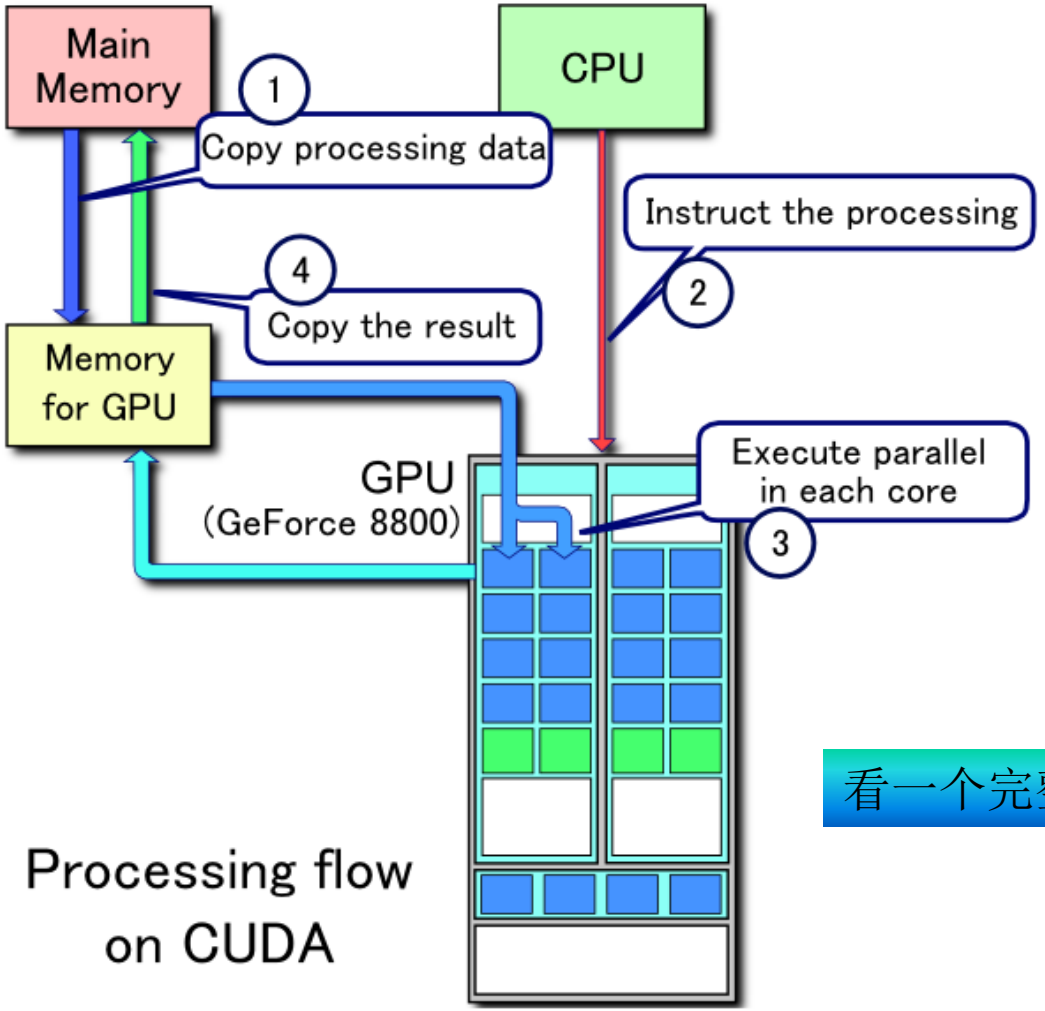
资料整理
火龙果软件
荐

cuda程序数据传输

```
// allocate and initialize host (CPU) memory
float *h_A = …, *h_B = …, *h_C = …;
// allocate device (GPU) memory
float *d_A, *d_B, *d_C;
cudaMalloc( (void**) &d_A, N * sizeof(float));
cudaMalloc( (void**) &d_B, N * sizeof(float));
cudaMalloc( (void**) &d_C, N * sizeof(float));
// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float),
cudaMemcpyHostToDevice) );
cudaMemcpy( d_B, h_B, N * sizeof(float),
cudaMemcpyHostToDevice) );
// launch N/256 blocks of 256 threads each
vector_add<<<N/256, 256>>>(d_A, d_B, d_C);
// copy result back to host memory
cudaMemcpy( h_C, d_C, N * sizeof(float),
cudaMemcpyDeviceToHost) );
// free device (GPU) memory
cudaFree(d_A);
cudaFree(d_B);
```

资料整理
火龙果软件

cuda程序数据传输



看一个完整的例子：vedAdd.txt

资料整理
火龙果软件
荐

CUDA背景介绍

GPU架构

CUDA编程模型

CUDA存储模型

CUDA程序优化

资料整理
火龙果软件
荐

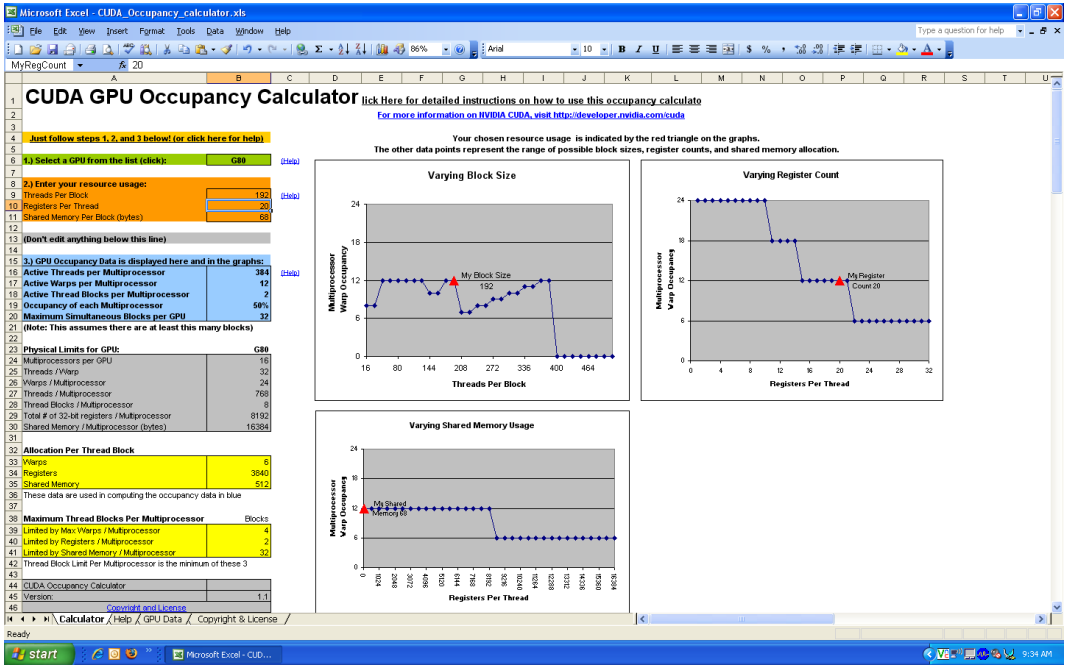
程序优化

- 首先优化算法
  - 复杂度小，高效的算法
  - 更加适合GPU特点
    - 相关数据访问
    - 数据并行处理
- 最大化GPU利用率
  - 程序级
  - 设备级
  - SM级
- 优化指令吞吐量
- 优化内存吞吐量

资料整理
火龙果软件
荐

提高计算资源利用率

- 尽可能增大一个SM中active block的数量
  - 可以有多个block等待处理，在一个warp需要访问存储器或者同步时，另外一个warp可以使用执行单元的资源
- active block的数量受到资源的限制，如每thread寄存器使用量，每block shared memory使用量，SM支持的最大warp数量，最大block数量等



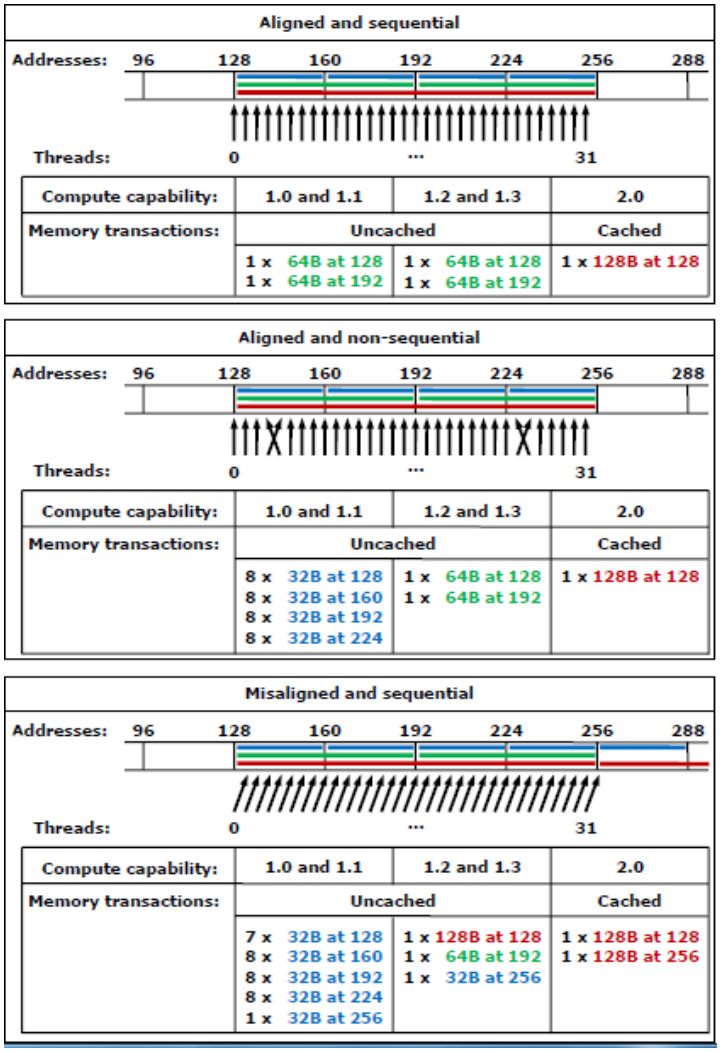
资料整理
火龙果软件
荐

存储器优化

- 访问片外的显存有很高的延迟
- 片上存储器带宽远高于显存带宽
- 显存控制器设计允许一次传输请求中传输很多数据（一个分段）
  - 使用对shared memory和register的访问代替对显存的访问
  - 利用数据局部性，选择合适的带cache的显存访问
  - 按照合并访问条件访问显存，了解Fermi的cache，提高cache命中率
- 物理设计决定各种存储器都是分组的，端口是有限的，每一个存储器单元的位宽是固定的
  - 避免bank conflic
  - 理解数据类型对齐

资料整理
火龙果软件
荐

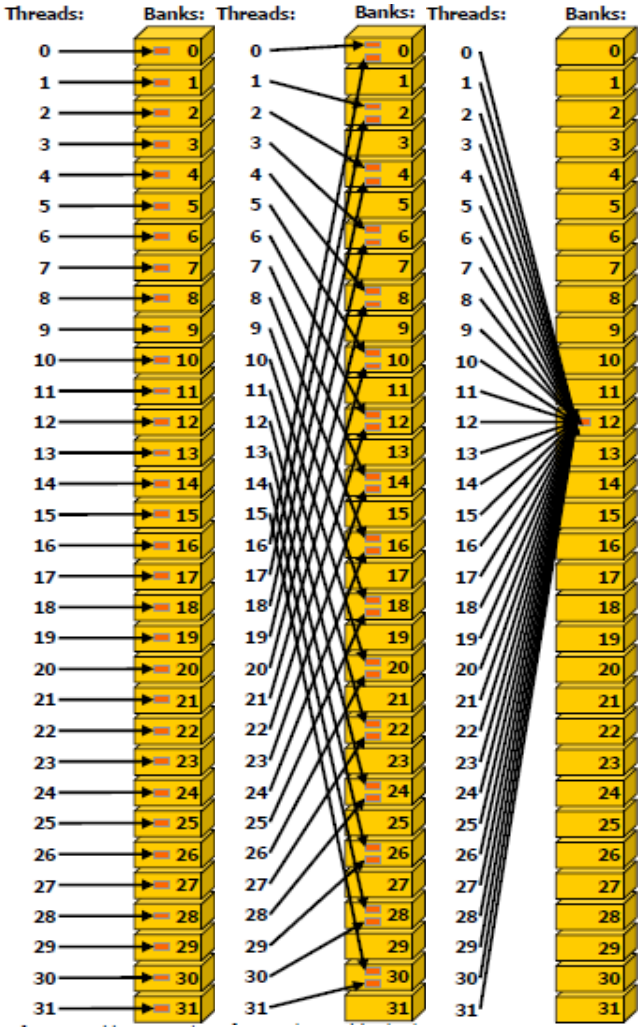
显存的合并访问



- 1.0/1.1硬件-只支持对32bit 64bit 128bit数据类型合并访问-对线程与数据次序，对齐，分段等非常敏感，一旦不满足合并访问，惩罚极高
- 1.2/1.3硬件-增加对8bit 16bit数据类型合并访问支持-有一定cache机制，可以缓冲不同长度的分段
- 2.0硬件-完整的L1/L2 cache机制-每次缓冲一个128Byte分段

资料整理
火龙果软件
荐

共享内存的bank conflict



- Shared memory被分为多个bank，每个bank只有一个32bit端口
- 相邻的32bit存储在相邻的bank中
- Bank conflict只受stride影响，不受offset影响
- 如果同一half-warp(1.x)或者warp(2.x)中的线程访问了同一bank中的数据，就会发生bank conflict
- 广播可以避免一部分读取时的bank conflict
- 1.x硬件只有16个bank，2.x硬件有32个bank

资料整理
火龙果软件
荐

优化小技巧

- 选用计算复杂度较小的算法
- 并行度高，粗粒度并行多，细粒度并行有局部性
- 分支映射成固定运算，展开代码避免循环
- 在精度允许的前提下使用带有__前缀的快速算法，只在必要的部分使用双精度和64-bit int
- 使用移位运算代替整数除法和求余
- 只在线程间通信前进行同步
- 对显存的I/O成本很高，提高两次对显存访问之间的计算量，通过同时计算与访存隐藏延时
- 把适合的数据放入纹理和常数缓存等缓解带宽压力，提高读取速度
- 避免bank conflict，非合并访问或cpu-gpu数据传输 使用数组的结构体,而不是结构体数组
- 使用对齐，类型转换等手段实现合并访问

资料整理
火龙果软件
荐

Q&A

资料整理
火龙果软件
荐