

CUDA C 编程权威指南

基于CUDA的异构并行计算

硬件提供支持多线程或多进程的平台

对计算机体系结构了解，编写正确且高效的进程

并行性

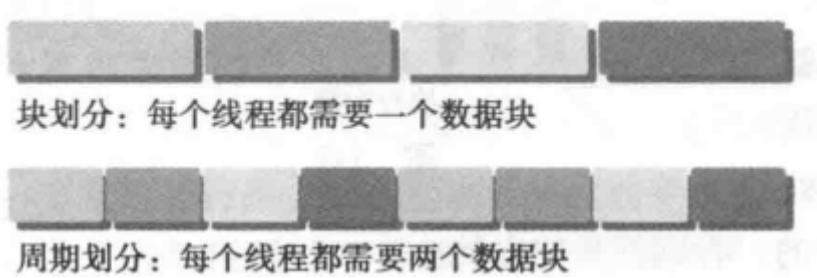
任务并行：多任务可以独立、大规模的并行执行。重点利用多核系统对任务进行分配

数据并行：处理器可以同时处理很多数据。利用多核系统对数据进行分配。

数据划分

块划分：每个线程只处理数据的一部分，通常数据具有相同大小

周期划分：每个线程作用于数据的多部分



计算机架构

指令流和数据流分类

SISD：串行架构

SIMD：所有核心只有一个指令流处理不同的数据流（向量机）

MISD：每个核心使用多个指令流处理同一个数据流

MIMD：多个核心通过多个指令流来处理多个数据流

设计目标：降低延迟、提高带宽、提高吞吐量

内存组织方式分类

分布式内存的多节点系统

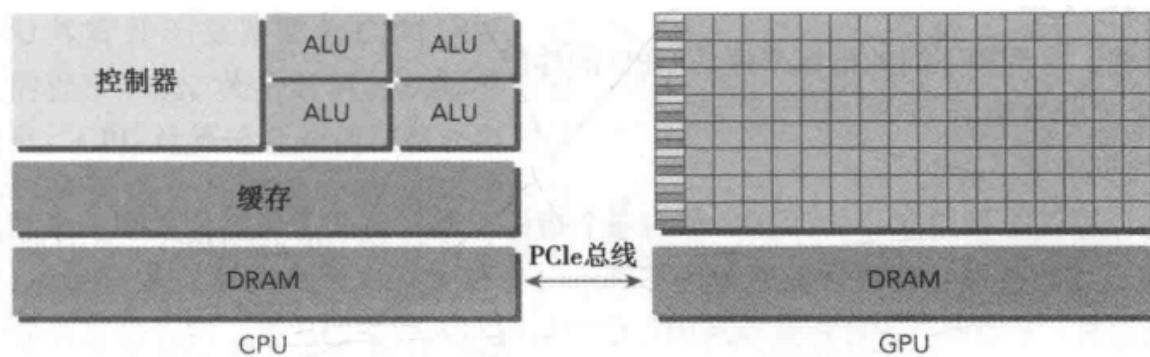
处理器有自己的本地内存，处理器之间通过网络进行通信

共享内存的多处理器系统

多个处理器与同一个物理内存相关联或公用一个低延迟的链路

异构架构

异构计算节点包含两个多核CPU插槽和多个众核GPU，GPU通过总线与基于CPU的主机相连。CPU所在位置被称为主机端，GPU称为设备端。异构平台由CPU初始化，CPU负责管理设备端的环境、代码和数据。



面试题：

CPU和GPU分别适合于什么情况

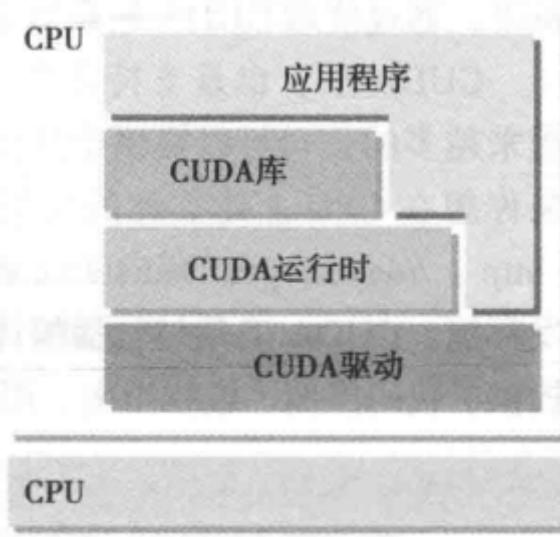
如果一个问题具有较小数据规模、复杂的控制逻辑和或多或少的并行性，最好选择CPU处理，因为CPU有处理复杂逻辑和指令级并行性的能力。包含大量数据并表现出大量的数据并行性，使用GPU。GPU有大量可编程的核心，支持大规模多线程运算，相比CPU有较大的峰值带宽。

CPU线程与GPU线程

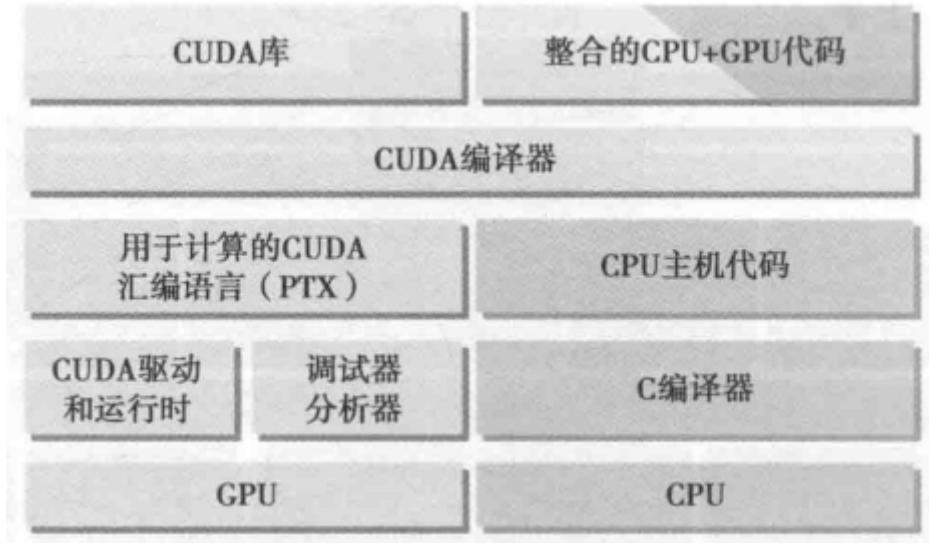
CPU线程通常是重量级的实体，操作系统需要切换线程，上下文切换开销大；

GPU线程是高度轻量级，核用来处理大量并发的、轻量级线程，提高吞吐量。

CUDA



CUDA程序：CPU上运行的主机代码和GPU上运行的设备代码



CUDA编程结构

1. 分配GPU内存
2. 从CPU内存拷贝数据到GPU内存
3. 调用CUDA内核函数完成程序指定的计算
4. 将数据从GPU移到CPU
5. 释放CPU内存空间

```

#include<stdio.h>
__global__ void helloFromGPU(void){
    printf("Hello World from GPU!\n");
}
int main(int argc, char** argv){
    // hello from cpu
    printf("Hello World from CPU!\n");

    helloFromGPU<<<1, 10>>>();
    // 显示释放和清空当前进程中与当前设备有关的所有资源
    cudaDeviceReset();
    return 0;
}

```

CUDA核中的3个关键抽象：线程组的层次结构，内存的层次结构以及障碍同步

CUDA编程模型

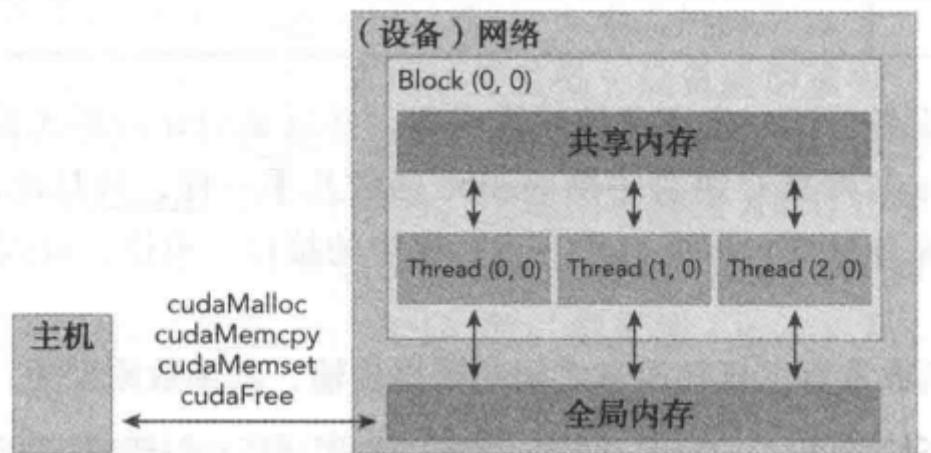
目标

主机和设备分配内存空间

在CPU和GPU之间拷贝共享内存

内存管理

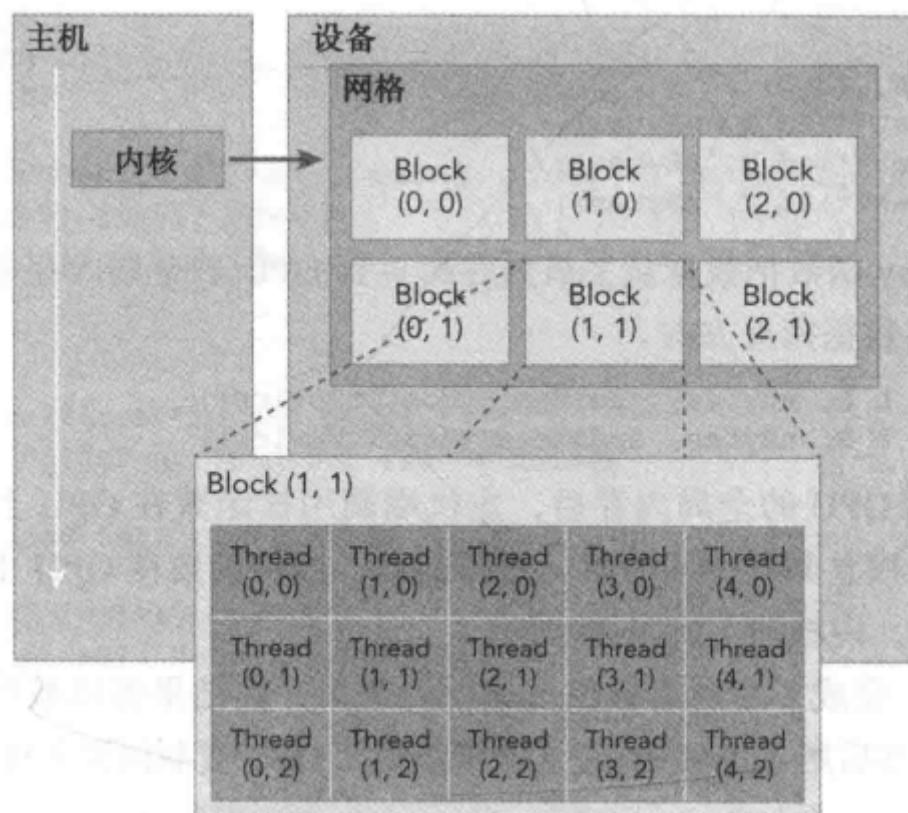
用于执行GPU内存分配的函数：向设备分配一定字节的线性内存，并以devPtr的形式返回指向所分配内存的指针



```
cudaMalloc //与C语言malloc函数几乎一样
```

```
cudaError_t cudaMemcpy(void* dst, const void* src,  
size_t count, cudaMemcpyKind kind);  
// kind : 1.cudaMemcpyHostToHost  
//         2.cudaMemcpyHostToDevice  
//         3.cudaMemcpyDeviceToHost  
//         4.cudaMemcpyDeviceToDevice  
//CPU分配成功：cudaSuccess 否则：  
cudaErrorMemcpyAllocation  
//CudaMemcpy的调用会导致主机运行阻塞  
char* cudaGetErrorString(cudaError_t error); //将错误代码转化为可读信息
```

线程层次结构：线程块（block）和线程块网格（grid）



由一个内核启动所产生的所有线程统称为一个网格；同一网格中所有线程共享相同的全局内存空间。一个线程块包含一组线程，统一线程块内的线程可以通过**同步和共享内存**协作。不同块内线程不能协作。

blockIdx(线程块在线程格内的索引)

threadIdx(块内线程的索引)

坐标基于uint3,可以通过x,y,z三个字段来指定。

blockDim(线程块的维度，每个线程块中的线程数量)

gridDim (线程格的维度，每个线程网格中的线程块数来表示)

块大小的限制因素是可利用的计算资源（如寄存器，共享内存）

核函数的调用与主机线程是异步的，核函数调用结束后，控制权立刻返回给主极端。可利用 cudaDeviceSynchronize() 强制主机端程序等待所有核函数执行结束

```
cudaError_t cudaDeviceSynchronize(void);
```

cudaMemcpy函数在主机与设备间拷贝数据时，主机端隐士同步。

核函数编写

限定符	执行	调用	备注
-----	----	----	----

限定符	执行	调用	备注
global	设备端	可以主极端调用，也可以计算能力为3的设备中调用	必须void返回类型
device	设备端	仅设备端调用	
host	主机端	仅主极端调用	可以省略

device 和 host 可一起使用，这样函数可以同时在主机和设备端进行编译。

cuda核函数的限制：

- 1.只能访问设备内存
- 2.必须具有void返回类型
- 3.不支持可变数量的参数
- 4.不支持静态变量
- 5.显示异步行为

处理错误

```
#define CHECK(call)
{
    const cudaError_t error = call;
    if(error != cudaSuccess) {
        printf("Error: %s:%d, ", __FILE__, __LINE__);
        printf("code:%d, reason:%s\n", error, cudaGetErrorString(error));
        exit(1);
    }
}
```

CHECK(cudaDeviceSynchronize()) 核函数启动后添加这个检查点会阻塞主极端线程，是该检查点成为全局屏障。

给核函数计时

```
#include<sys/time.h>
double cpuSecond(){
    struct timeval tp;
    gettimeofday(&tp, NULL);
    return ((double)tp.tv_sec +
(double)tp.tv_usec*1.e-6);
}

double iStart = cpuSecond();
kernel_name<<<grid, block>>>(argument list);
cudaDeviceSynchronize();
double iElaps = cpuSecond() - iStart;

//nvprof --help
```

组织并行线程

使用合适的网格和块大小来正确组织线程，可以对内核性能产生很大的影响。

在一个矩阵加法函数中，一个线程通常被分配一个数据元素来处理。

1. 使用块和线程索引从全局内存中访问指定的数据

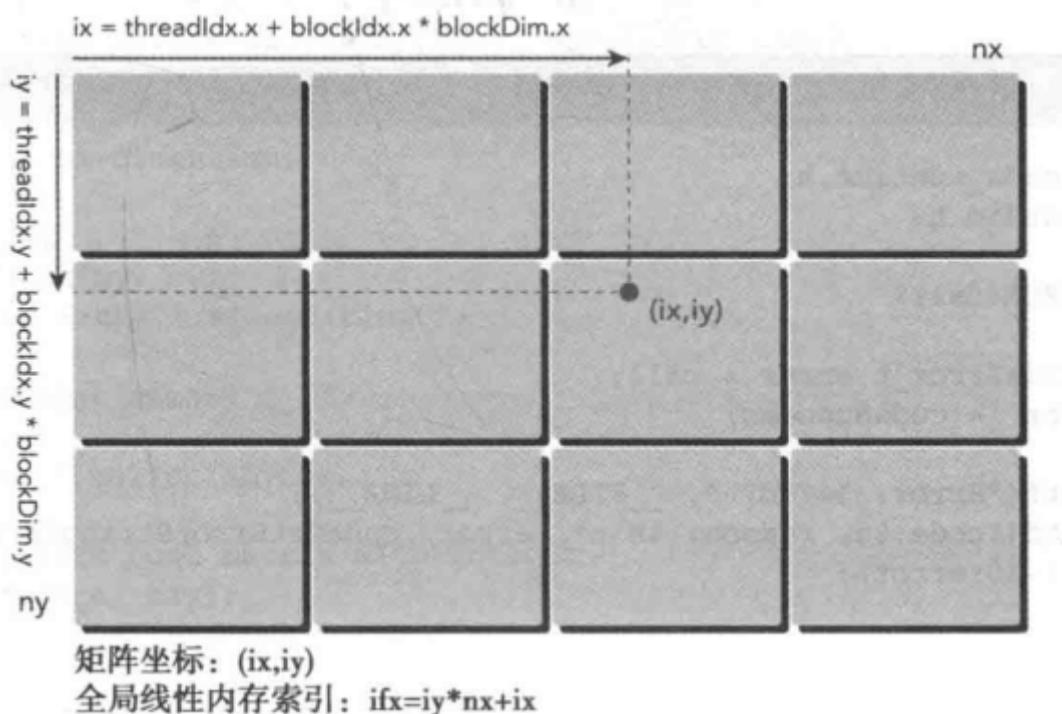
线程和块索引

矩阵中给定点的坐标

```
ix = threadIdx.x + blockIdx.x * blockDim.x;  
iy = threadIdx.y + blockIdx.y * blockDim.y;
```

全局线性内存中的偏移量

```
idx = iy * nx + ix;
```



```

//set up execution configuration
dim3 block(4, 2);
dim3 grid((nx+block.x-1)/block.x, (ny+block.y-1)/block.y);

__global__ void printThreadIndex(int *A, const int nx, const int ny){
    int ix = threadIdx.x + blockIdx.x * blockDim.x;
    int iy = threadIdx.y + blockIdx.y * blockDim.y;
    unsigned int idx = iy * nx + ix;

    printf("thread_id (%d,%d) block_id (%d,%d)
coordinate (%d,%d)"
           "global index %2d ival %2d\n",
           threadIdx.x, threadIdx.y,
           blockIdx.x, blockIdx.y, ix, iy, idx,
           A[idx]);
}

```

设备管理

```

//查询关于gpu设备的信息
cudaError_t cudaGetDeviceProperties(cudaDeviceProp* prop, int device);

//确定最优GPU
int numDevice = 0;
cudaGetDeviceCount(&numDevice);
if(numDevice > 1){
    int maxMultiprocessors = 0, maxDevice = 0;
    for(int device = 0; device<numDevice;device++){
        cudaDeviceProp props;
        cudaGetDeviceProperties(&props, device);

```

```
    if(maxMultiprocessors <
props.multiProcessorCount){
    maxMultiprocessors =
props.multiProcessorCount;
    maxDevice = device;
}
cudaSetDevice(maxDevice);
}
```

nvidia-smi查询GPU信息

nvidia-smi -L //系统有多少个GPU

nvidia-smi -q -i 0 // 查询0号GPU详细信息

相关参数

MEMORY

UTILIZATION

ECC

POWER

CLOCK

COMPUTE

PIDS

PERFORMANCE

SUPPORTED_CLOCKS

PAGE_RETIREMENT

ACCOUNTING

```
nvidia-smi -q -i 0 -d UTILIZATION | tail -n 5
```

使用环境变量CUDA_VISIBLE_DEVICES可以在运行时指定所选的GPU且无须更改应用程序

CUDA执行模型

CUDA执行模型概述

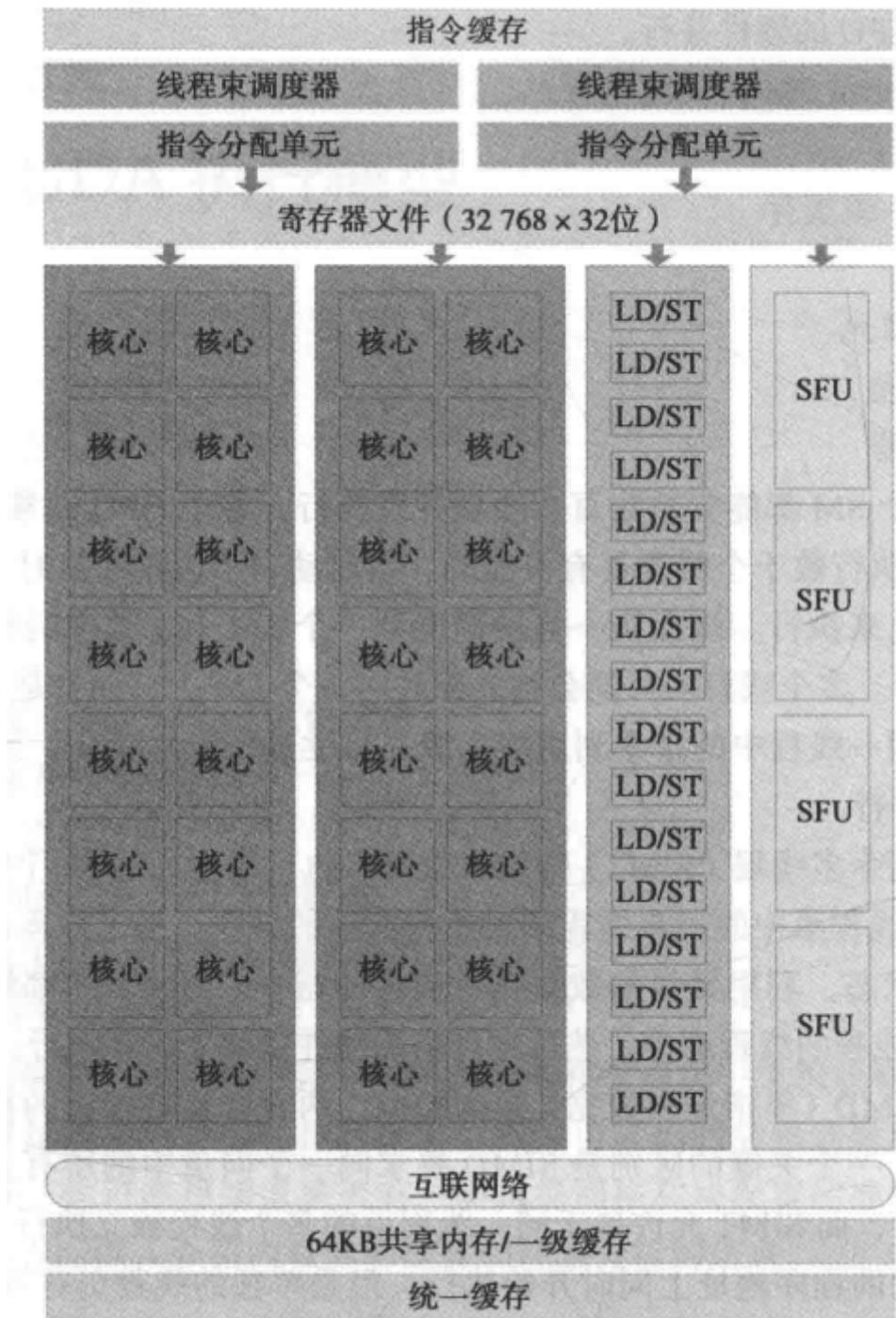
GPU架构SM处理器

- 1.CUDA核心
- 2.共享内存/一级缓存
- 3.寄存器文件
- 4.加载/存储单元
- 5.特殊功能单元
- 6.线程束调度器

cuda使用单指令多线程（SIMT）架构来管理和执行线程，每32个线程为一组，称为线程束（warp），线程束中所有线程执行相同的指令，每个线程有自己的指令地址计数器和寄存器状态，利用自身的数据执行当前指令。

SIMT与SIMD区别： SIMD要求同一个向量中的所有元素要在一个统一的同步组中一起执行，而SIMT允许统一线程束中的多个线程独立执行，

SIMT每个线程都有自己的指令地址计数器，寄存器状态和独立的执行路径



16个Load和Store，允许每个时钟周期内有16个线程计算源地址和目的地址

SFU执行固有指令，正弦，余弦，平方和插值等

一个线程块只能在一个SM上被调度

Kepler架构

1.强化的SM

2. 动态并行

3. Hyper-Q 技术

nvvp nvprof 使用

线程束执行的本质

一个线程块中线程束的数量 = ceil (一个线程块中线程的数量 / 线程束大小)

线程束分化：CPU 拥有复杂的硬件以执行分支预测；GPU 中一个线程执行一条指令，那么同一线程束中的线程执行该指令，如果一个线程束中的线程产生分化，线程束将连续执行每一个分支。

```
/* 线程束分化 */
__global__ vlid mathKernel1(float *c)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    float a, b;
    a = b = 0.0f;
    if(tid % 2 == 0){
        a = 100.0f;
    }else{
        b = 200.0f;
    }
    c[tid] = a + b;
}

/* 交叉存取数据，防止线程束分化 */
__global__ void mathKernel2(void){
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    float a, b;
```

```
a = b = 0.0f;
if((tid / warpSize) % 2 == 0){
    a = 100.0f;
}else{
    b = 200.0f;
}
c[tid] = a + b;
}

/* nvcc -g -G a.cu -o output_file 编译器不利用分支预测优化*/
/* nvprof --metrics branch_efficiency ./outputfile 查看分支效率*/
```

SM处理器每个线程上下文，整个线程束的生存期是保存在芯片内的，从一个上下文切换到另一个执行上下文没有损失

同一个SM中线程块和线程束的数量取决于在SM中可用的且内核所需的寄存器的共享内存的数量

同步：

cudaDeviceSynchronize函数可以用来阻塞主机应用程序，直到所有的CUDA操作完成

_syncthreads在同一线程块中每个线程都必须等待直至该线程块中所有其他线程都已经达到这个同步点。

可扩展性

可实现占用率 (achieved_occupancy)：每个周期内活跃线程束的平均数与最大支持线程束的比值

```
nvprof --metrics achieved_occupancy ./output_file /* 可实现占用  
率 */
```

```
nvprof --metrics gld_throughput ./output_file /* 内存读取效率  
*/
```

```
nvprof --metrics gld_efficiency ./output_file /* 全局加载效率 */
```

并行性

增大并行性：一个方法调整blockDim.x

1. 领域并行

```
__global__ void reduceInterleaved(int *g_idata, int  
*odata, int n)  
{  
    unsigned int tid = threadIdx.x;  
    unsigned int idx = blockIdx.x * blockDim.x +  
threadIdx.x;  
  
    int *idata = g_idata + blockIdx.x * blockDim.x;  
    if(idx >= n) return;  
  
    for(int stride = 1; stride < blockDim.x; stride  
*= 2){  
        if((tid%(2 * stride * tid))==0){  
            idata[tid] += idata[tid + stride];  
        }  
        __syncthreads();  
    }  
    if(tid==0) g_odata[threadIdx.x] = idata[0];  
}
```

2.间域并行

```
__global__ void reduceInterleaved(int *g_idata, int
* g_odata, unsigned int n){
    unsigned int tid = threadIdx.x;
    unsigned int idx = blockIdx.x * blockDim.x +
threadIdx.x;
    int *idata = g_idata + blockIdx.x * blockDim.x;
    if(idx >= n) return;
    for(int stride =
blockDim.x/2;stride>0;stride>>=1){
        if(tid<stride){
            idata[tid] += idata[tid + stride];
        }
        __syncthreads();
    }
    if(tid==0) g_odata[blockIdx.x]=idata[0];
}
```

3.循环展开

```
__global__ void reduceUnrolling (int *g_idata, int
*g_odata, unsigned int n){
    unsigned int tid = threadIdx.x;
    unsigned int idx = blockIdx.x * blockDim.x * 2 +
threadIdx.x;
    int *idata = g_idata + blockIdx.x * blockDim.x *
2;
    if(idx + blockDim.x < n) g_idata[idx] +=
g_idata[idx + blockDim.x];
    __syncthreads();
    for(int stride=blockDim.x/2;stride>0;stride>>=1)
{
```

```

        if(tid < stride){
            idata[tid] += idata[tid+stride];
        }
        __syncthreads();
    }
    if(tid==0) g_odata[blockIdx.x] = idata[0];
}

```

4. 模板函数

```

template<unsigned int iBlockSize>
__global__ void reduceCompleteUnrollWarps8(int
*g_idata, int *g_odata, unsigned int n){
    unsigned int tid = threadIdx.x;
    unsigned int idx = blockIdx.x * blockDim.x * 8 +
threadIdx.x;
    int *idata = g_idata + blockIdx.x * blockDim.x *
8;
    if(idx + blockDim.x * 7 < n){
        int a1 = g_idata[idx];
        int a2 = g_idata[idx+blockDim.x];
        int a3 = g_idata[idx+blockDim.x*2];
        int a4 = g_idata[idx+blockDim.x*3];
        int b1 = g_idata[idx+blockDim.x*4];
        int b2 = g_idata[idx+blockDim.x*5];
        int b3 = g_idata[idx+blockDim.x*6];
        int b4 = g_idata[idx+blockDim.x*7];
        g_idata[idx] = a1+a2+a3+a4+b1+b2+b3+b4;
    }
    __syncthreads();
    if(iBlockSize>=1024&&tid<512) idata[tid] +=
idata[tid+512];
    __syncthreads();
}

```

```

        if(iBlockSize>=512&&tid<256) idata[tid] +=  

idata[tid+256];  

__syncthreads();  

if(iBlockSize>=256&&tid<128) idata[tid] +=  

idata[tid+128];  

__syncthreads();  

if(iBlockSize>=1128&&tid<64) idata[tid] +=  

idata[tid+64];  

__syncthreads();  

if(tid<32){  

    volatile int *vsmem = idata;  

    vsmem[tid] += vsmem[tid + 32];  

    vsmem[tid] += vsmem[tid + 16];  

    vsmem[tid] += vsmem[tid + 8];  

    vsmem[tid] += vsmem[tid + 4];  

    vsmem[tid] += vsmem[tid + 2];  

    vsmem[tid] += vsmem[tid + 1];  

}  

if(tid == 0) g_odata[blockIdx.x] = idata[0];
}

```

5. 动态并行

在内核函数中执行递归调用

```

if(tid == 0){  

    gpuRecursiveReduce<<<1, istride>>>(idata, odata,  

istride);  

    // sync all child grids launched in this block  

    cudaDeviceSynchronize();  

}  

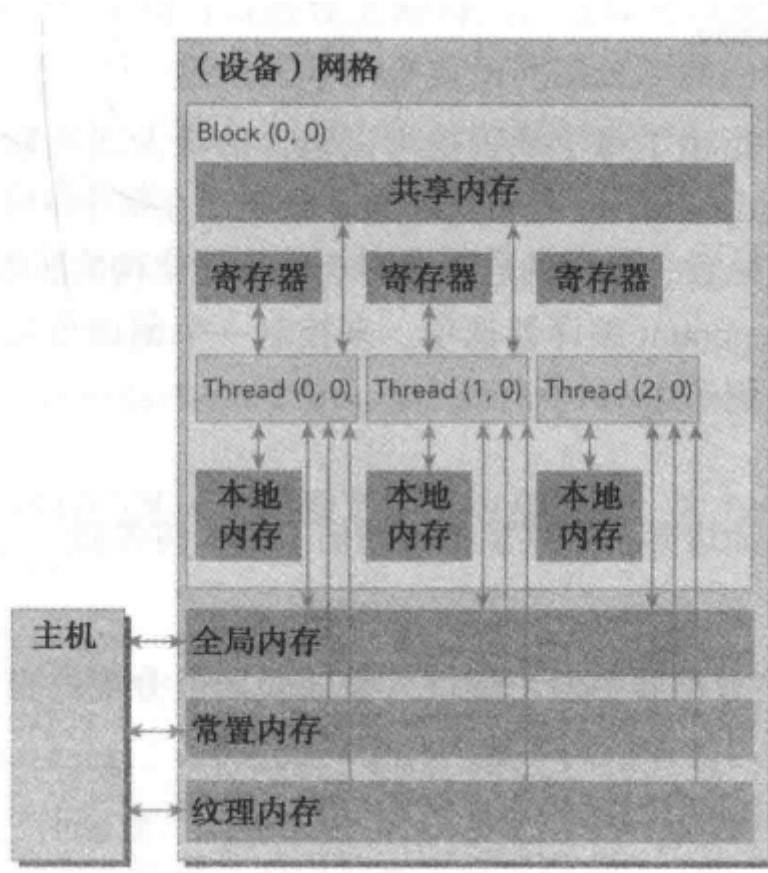
//sync at block level again  

__syncthreads();

```

全局内存

CUDA内存模型



1. 寄存器

`ncv -Xptxas -v, -abi=no //` 输出寄存器的数量， 共享内存字节数， 常量内存字节数

2. 共享内存

`share` 片上内存，在核函数的范围内声明

SM中的一级缓存和共享内存都使用64KB的片上内存划分，它通过静态划分，但在运行时可以通过指令进行动态配置

```
cudaError_t cudaFuncSetCacheConfig(const void* func,  
enum cudaFuncCache cacheConfig);  
/* func参数  
   cudaFuncCacheNone: 没有参考值  
   cudaFuncCachePreferShared: 建议48KB的共享内存和  
16KB的一级缓存  
   cudaFuncCachePreferL1: 建议48KB的一级缓存和  
16KB的共享内存  
   cudaFuncCachePreferEqual: 建议相同尺寸的一级缓存核  
共享内存， 都是32KB
```

3.本地内存

本地内存由每个线程独有，延迟比寄存器大；寄存器溢出时数据会被保存到本地内存；

内核编译时无法确定索引的数组保存在本地内存；结构体和大数组保存在本地内存

4.常量内存

——constant——

驻留在设备内存中，必须在全局空间内和所有核函数之外进行声明

核函数只能从常量内存中读取数据，常量内存必须在主极端使用下面函数初始化：

```
cudaError_t cudaMemcpyToSymbol(const void* symbol,  
const void* src, size_t count);  
/* 线程束中所有线程从相同的内存地址中读取数据时，常量内存表现  
最好  
如果线程束中每个线程都从不同的地址空间读取数据，并且只读一  
次，常量内存不是最佳选择，因为每从一个常量内存中读取一次数据，  
都会广播给线程束里的所有线程 */
```

5.纹理内存

纹理内存驻留在设备内存中，并在每个SM的只读缓存中缓存；纹理内存是一种通过指定的只读缓存访问的全局内存。

二维空间线程束使用纹理内存访问二维数据的线程可以达最好的性能

对应用程序，纹理内存比全局内存慢

6.全局内存

声明可以在任何SM上被访问到，贯穿应用程序的整个生命周期

静态声明一个变量：——device——

动态：cudaMalloc 使用cudaFree释放

7.GPU缓存

一级缓存

二级缓存

只读常量缓存

只读纹理缓存

每个SM都有一个一级缓存，所有的SM共享一个二级缓存；每个SM只有一个只读常量/纹理缓存，在设备内存中提高来自于各自内存空间内的读取性能。

表 4-1 CUDA 变量和类型修饰符

修 饰 符	变 量 名 称	存 储 器	作 用 域	生 命 周 期
	float var	寄存器	线程	线程
	float var[100]	本地	线程	线程
<code>_shared_</code>	float var ⁺	共享	块	块
<code>_device_</code>	float var ⁺	全局	全局	应用程序
<code>_constant_</code>	float var ⁺	常量	全局	应用程序

⁺既可以表明标量也可以表示数组。

表 4-2 总结了各类存储器的主要特征。

表 4-2 设备存储器的重要特征

存 储 器	片上 / 片外	缓 存	存 取	范 围	生 命 周 期
寄存器	片上	n/a	R/W	一个线程	线程
本地	片外	⁺	R/W	一个线程	线程
共享	片上	n/a	R/W	块内所有线程	块
全局	片外	⁺	R/W	所有线程 + 主机	主机配置
常量	片外	Yes	R	所有线程 + 主机	主机配置
纹理	片外	Yes	R	所有线程 + 主机	主机配置

```
cudaMemcpyToSymbol(devData, &value, sizeof(float));
cudaMemcpyFromSymbol(&value, devData,
sizeof(float));
/* 获取全局变量地址 */
cudaError_t cudaGetSymbolAddress(void** devPtr,
const void* symbol);
```

内存管理

核函数在设备内存空间运行，CUDA运行时提供函数以分配和释放设备内存，在主机上使用

```
cudaError_t cudaMalloc(void **devPtr, size_t count);
```

可分配全局内存，分配count字节的（任何变量类型）的全局变量，失败返回cudaErrorMemoryAllocation; 用devptr指针返回该内存的地址。

```
/* Initialization */
cudaError_t cudaMemset(void *devPtr, int value,
size_t count);
/* Free */
cudaError_t cudaFree (void *devPtr);
```

设备内存的分配和释放操作成本较高，应用程序应重利用设备内存。

寄存器溢出

```
__global__ void
__launch_bounds__(maxThreadsPerBlock,
minBlocksPerMultiprocessor)
kernel(...){
    // your kernel body
}
/* nvcc编译参数-maxregcount(设置__launch_bounds__时该参数被忽略)
```

本地内存由每个线程独有，延迟比寄存器大；寄存器溢出时数据会被保存到本地内存；

内核编译时无法确定索引的数组保存在本地内存；结构体和大数组保存在本地内存

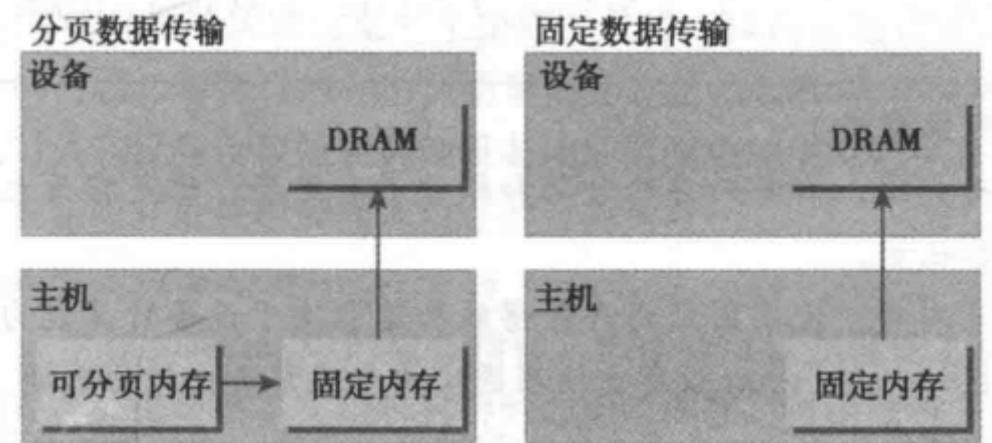
内存传输

```
cudaError_t cudaMemcpy(void *dst, void *src, size_t count, enum cudaMemcpyKind kind);
***** kind *****
***** cudaMemcpyHostToHost *****
***** cudaMemcpyHostToDevice *****
***** cudaMemcpyDeviceToHost *****
***** cudaMemcpyDeviceToDevice *****/
```

减少设备和主机间的数据传输，提高程序的整体性能。

固定内存

GPU不能在可分页主机内存上安全的访问数据，当主机操作系统在物理位置上移动数据时，它无法控制。从可分页主机内存传输数据到设备内存时，CUDA驱动程序首先分配临时页面锁定的或固定的主机内存，将主机源数据复制到固定内存



```
/* CUDA分配固定内存 */
cudaError_t cudaMallocHost(void **devPtr, size_t count);
/* 释放固定内存 */
cudaError_t cudaFreeHost(void *ptr);
```

与可分页内存相比，固定内存的分配和释放成本更高，但它为大规模数据传输提供了更高的传输吞吐量。

零拷贝内存

零拷贝内存主机和设备都可以访问；固定不可分页的

优势:1.设备内存不足时可以利用主机内存

2.避免主机和设备间的显示数据传输

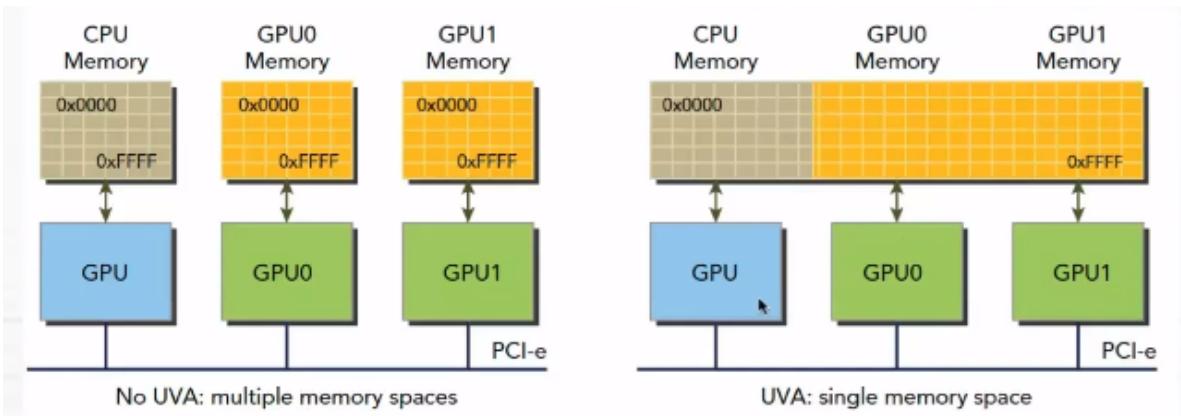
3.提高PCIe传输率

```
cudaError_t cudaHostAlloc(void **pHost, size_t
count, unsigned int flags);
/* flag:
   cudaHostAllocDefault 与cudaMallocHost行为一致
   cudaHostAllocPortable 返回被所有CUDA上下文使用的
固定内存
   cudaHostAllocWriteCombined 返回写结合内存
   cudaHostAllocMapped 主机写入与设备读取被映射到设
备地址空间中的主机内存 */
cudaFreeHost(void *pHost) // 内存释放

// 获取映射到固定内存的设备指针
cudaError_t cudaHostGetPointer(void **pDevice, void
*pHost, unsigned int flags);
```

统一虚拟寻址

主机内存和设备内存可以共享同一个虚拟地址空间;主机使用
CUDA API分配的内存（页锁定内存、零拷贝内存）与GPU上分配的
内存在同一个虚拟地址空间。



```
__host__ cudaError_t
cudaPointerGetAttributes(cudaPointerAttributes
*attributes, const void *ptr) // 返回指针属性
```

统一内存寻址

将内存和执行空间分离，可以根据需要将数据透明地传输到主机或设备上，以提升局部性和性能。

托管内存由底层系统自动分配的统一内存，与特定于设备分配内存可以互操作。

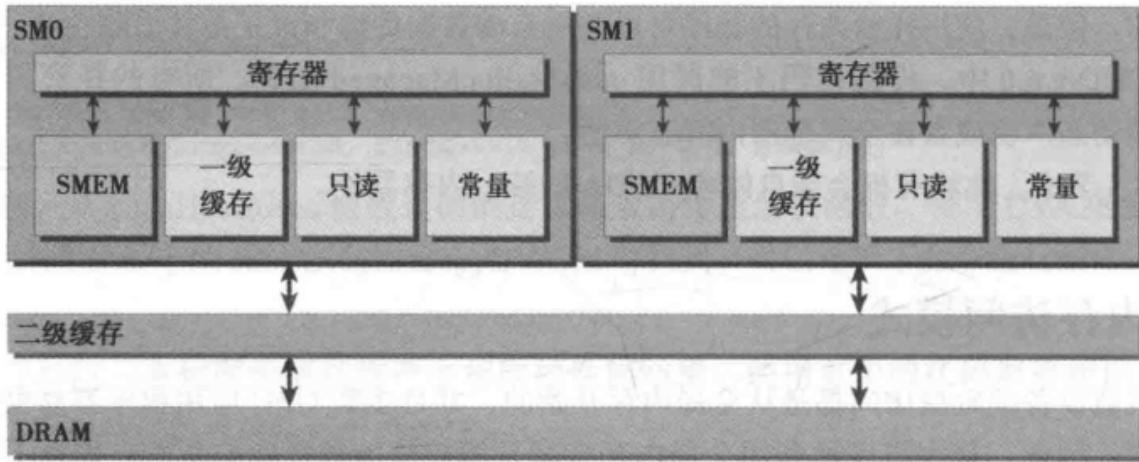
```
__device__ __managed__ int y; // 静态声明
cudaError_t cudaMallocManaged(void **devPtr, size_t
size, unsigned int flags=0); // 动态获取
cudaFree(); // 释放
```

使用托管内存的程序可以利用自动数据传输和重复指针消除功能。

内存访问模式

1. 对齐余合并访问

全局内存通过缓存来实现加载/存储；一个逻辑内存空间，通过核函数访问它。



内存加载访问模式：

缓存加载（启用一级缓存）

没有缓存的加载（禁用一级缓存）

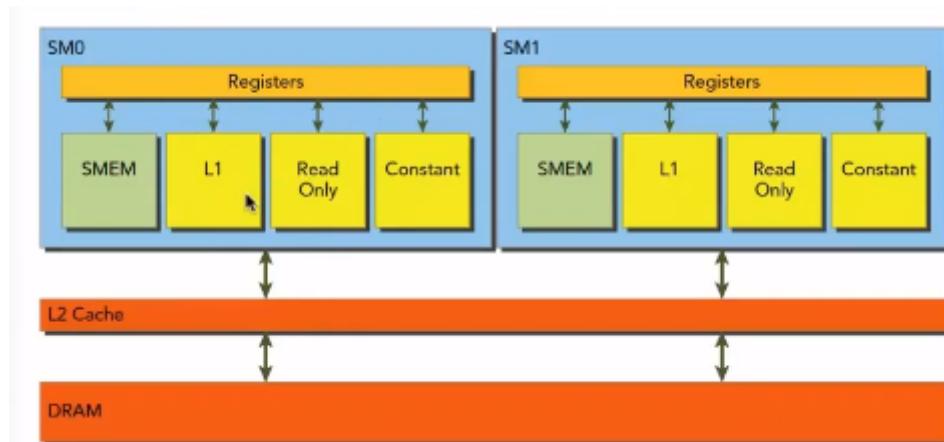
内存加载的访问模式的特点：

有缓存与没有缓存：如有启用一级缓存，则内存被缓存加载

对齐与非对齐：如果内存访问的第一个地址是32字节的倍数，则对齐加载

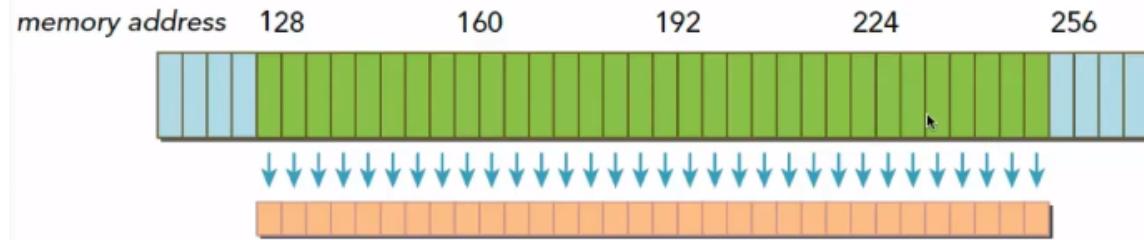
合并与未合并：如果线程束访问一个连续的数据块，则加载合并

缓存加载



数据存储在GPU的DRAM中，数据加载会经过L2缓存，部分还经过L1缓存；经过L1缓存时，内存事务大小为128字节，未经过则为32字节。

内存加载对齐和合并的特点



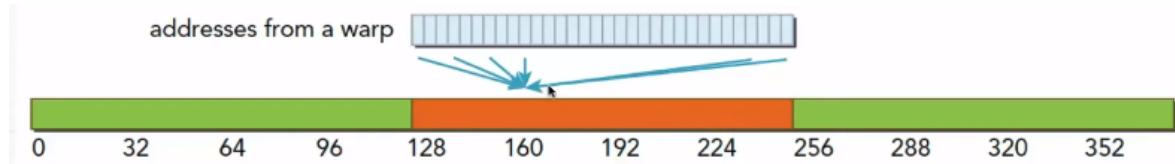
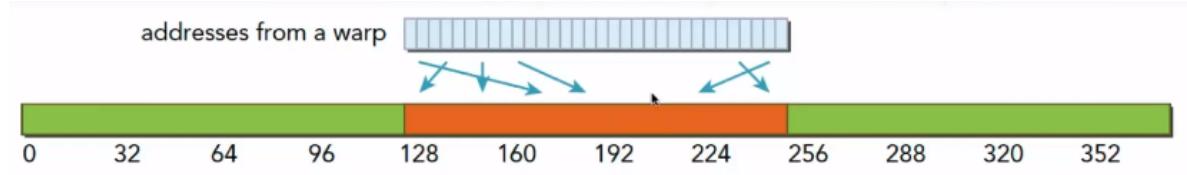
启用L1缓存：-Xptxas -dlcm=ca 关闭：-Xptxas -dlcm=cg

开启L1缓存，数据大小128字节，数据对其要求128整数倍

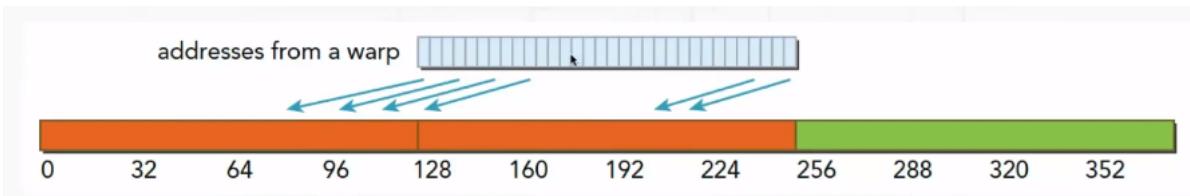
对齐：数据起始地址是128的整数倍

联合：线程束中线程访问的是连续的内存地址，内存带宽利用率100%

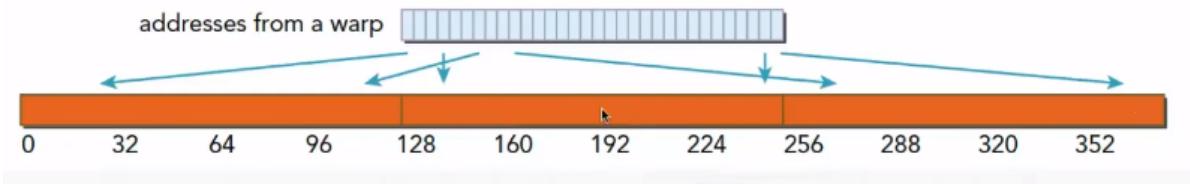
对齐非联合（带宽利用率分别为100%和1/32）



非对齐联合（内存带宽利用率50%）



非对齐非联合（内存带宽利用率极底）



内存带宽

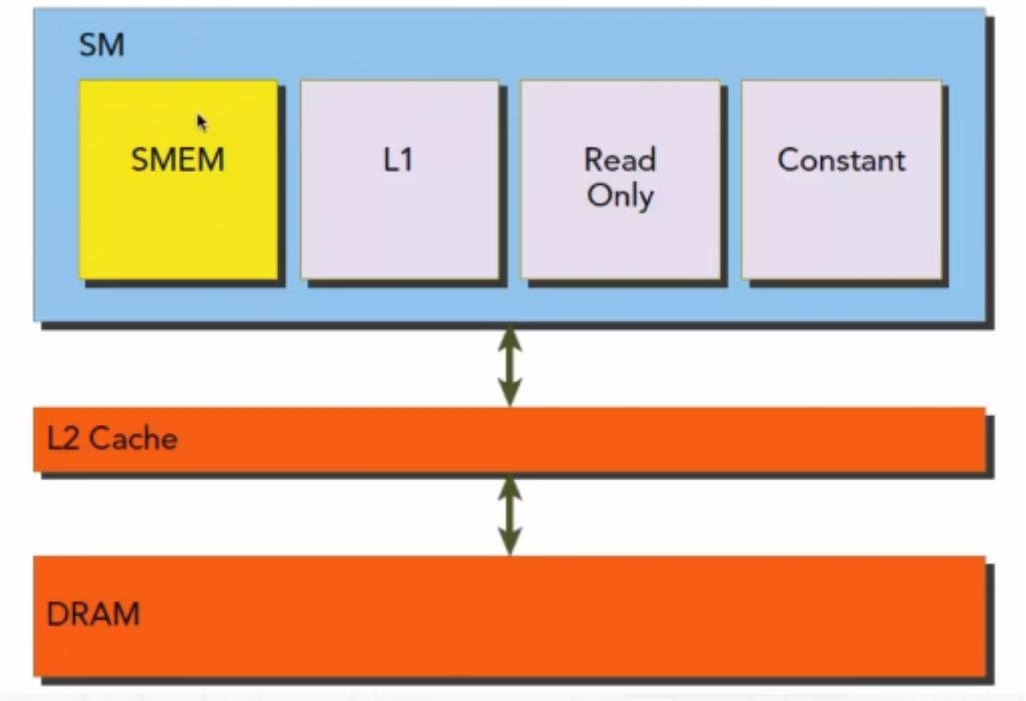
全局内存中数据的安排方式，线程束访问该数据的方式对带宽有显著影响

无缓存加载

关闭L1缓存，数据大小32字节，数据对齐要求32字节的整数

共享内存和常量内存

共享内存是较小的片上内存，具有相对较低的延迟，并且带宽比全局内存高；在SM上所有线程块间分配；在线程块启动时分配给线程。



局部静态分配：在核函数中定义

全局静态分配：在全局区域中定义

```
__share__ float = 0.0f;
```

动态分配共享内存

1. extern声明动态共享内存变量

2. 调用内核时指定动态共享内存大小

```
extern __shared__ var
kernel<<<grid, block, isize * sizeof(int)>>>( ... )
```

共享内存用途：

块内线程通信的通道

用于全局内存数据的可编程管理的缓存

告诉暂存存储器，用于转换数据以优化全局内存访问模式

共享内存bank

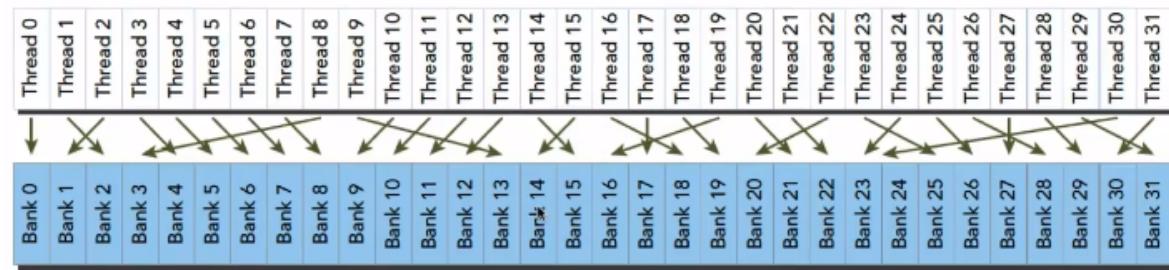
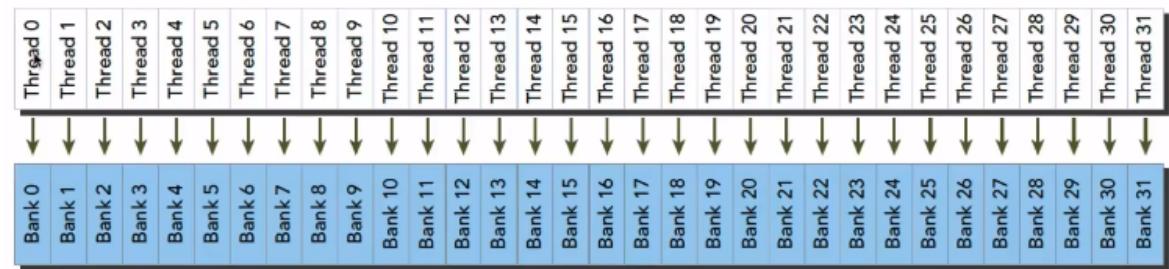
1.bank概念

共享内存被划分为32个连续的、相同大小的模块，这些模块称作banks；banks具有一维地址空间，可以被并行访问；共享内存地址与bank间的映射与GPU计算力和共享内存访问模式有关；共享内存的读写操作使每个bank只访问一个内存地址，那么这样的读写操作可以通过一次内存事务完成，否则需要多个内存事务。

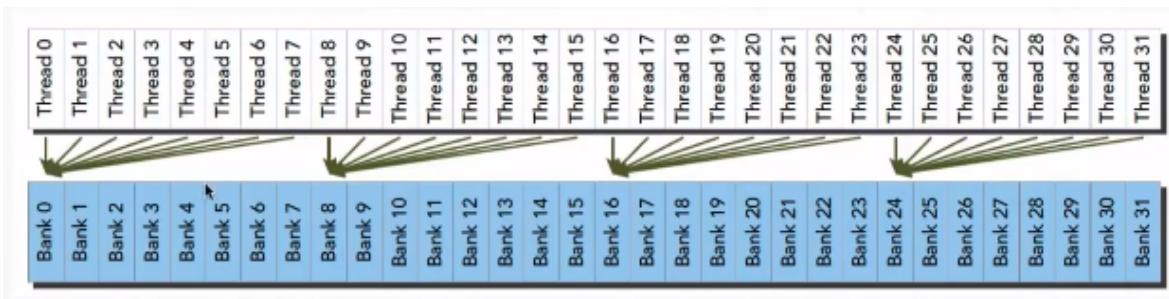
2.bank冲突

多个共享内存地址的读写请求落在同一个bank的现象称为bank冲突；bank冲突会导致读写请求的重复执行；出现bank冲突时，GPU硬件会将这样的请求分割为多个无冲突的内存事务

并行访问：多个地址访问映射到多个bank



串行访问：多个地址访问映射到一个bank



广播访问：线程束中多个线程访问相同的内存地址，此内存地址映射到一个bank，此时需要一次内存事务，数据会被广播到所有请求该数据的线程。

共享内存访问

共享内存有32个banks，具有64位，32位两种模式

64位模式

1. 连续的64位映射到一个bank

2. bank宽度为64位

3. 共享内存地址到banks映射

$$\text{bank index} = (\text{byte address} / 8 \text{ bytes/bank}) \% 32 \text{ banks}$$

32位模式

1. 连续的32位映射到一个bank

2. bank宽度为32位

共享内存访问模式

```
// 查询访问模式
cudaDeviceGetSharedMemConfig(cudaSharedMemConfig* );
// cudaSharedMemBankEightByte
// cudaSharedMemBankFourByte
// 模式设置
cudaDeviceSetSharedMemConfig(cudaSharedMemConfig);
```

共享内存大小配置

每个SM处理器有64K的片上存储，64K的存储空间用于L1缓存和共享内存

```
// cudaDeviceSetCacheConfig 按GPU设备配置
// cudaFuncSetCacheConfig 按内核函数配置
__host__ cudaError_t
cudaDeviceSetCacheConfig(cudaFuncCache cacheConfig);
__host__ cudaError_t cudaFuncSetCacheConfig(const
void*func, cudaFuncCache cacheConfig);
// cacheConfig: cudaFuncCachePREFER_SHARED
//                               cudaFuncCachePREFER_EQUAL
```

弱序内存

CPU线程将数据写入内存的顺序和其对应代码出现的顺序不一定完全一致

写入内存中的数据对其他线程可见的顺序和数据写入的顺序不一定完全一致

GPU线程从内存中读出数据的顺序和其对应代码出现的顺序不一定完全一致

线程屏障

线程屏障保证线程块中所有线程必须都执行到某个特定点才能继续执行

建立线程屏障：`_syncthreads`

保证线程此前对全局内存、共享内存所做的任何改动对线程块中的所有线程可见

如果要在条件分支中使用线程屏障，那么此条件分支对于线程块中所有线程束具有一致的结果

线程屏障不会等待已经结束的线程

线程块栅栏

内存栅栏保证在此之前写入内存的数据被所有线程可见

线程块栅栏作用域同一线程块，保证对**共享内存、全局内存**数据同步

线程块栅栏：`_threadfence_block()`

线程网格栅栏

网格栅栏作用域为相同线程网格，保证写入**全局内存**中的数据对网格中所有线程可见

`threadfence ()`

shuffle指令

1.shuffle指令使同一线程束中的线程可以直接交换数据

2.shuffle指令的数据交换不依赖于共享内存和全局内存，延迟极低

3.shuffle指令分为两类：用于整形数据、用于浮点数数据，每一类又包含四个指令

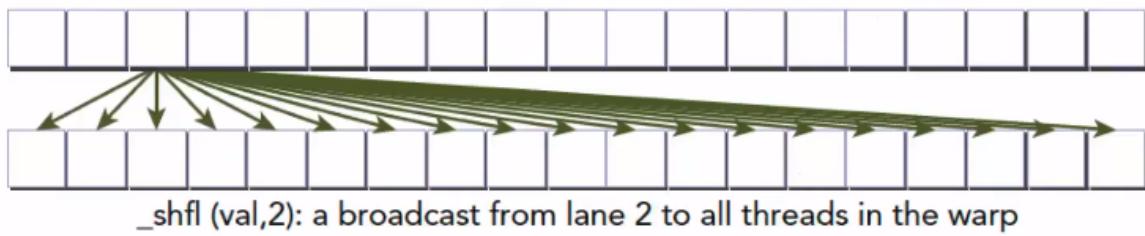
4.lane代表线程束中的一个线程

5.lane的索引范围为0~31

6. 对一维线程块，lane id = threadIdx.x % 32

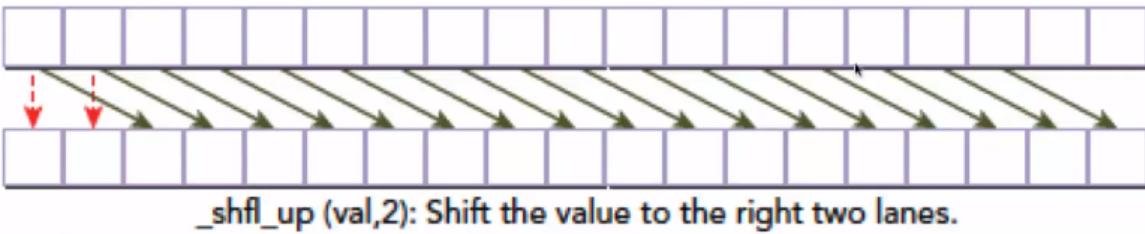
广播shuffle

```
int __shfl(int var, int srcLane, int width = warpSize);  
// 从srcLane所指定的线程并行读取var到相同线程束中的其它线程  
// width只能是2~32之间，2的n次方  
// 如果width不是32, shfl操作在两个子线程束中独立操作
```



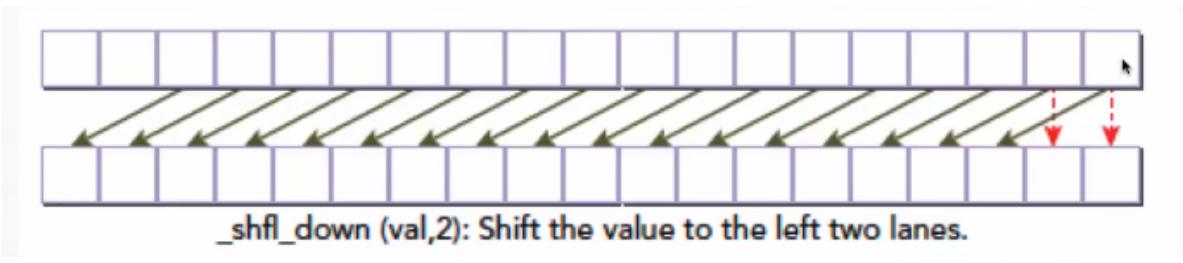
向上shuffle

```
int __shfl_up_sync(int var, unsigned int delta, int width=warpSize);
```



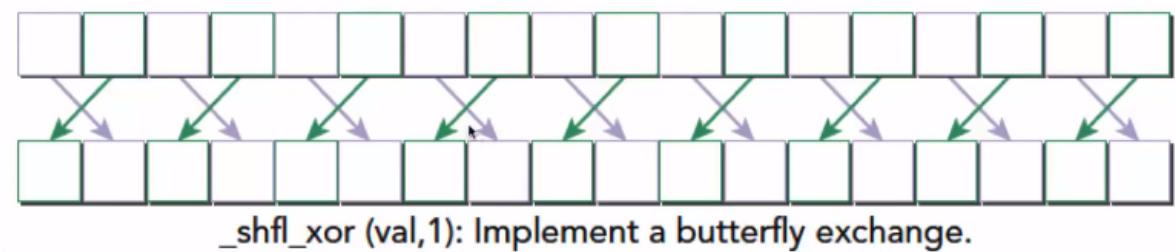
向后shuffle

```
int __shfl_down_sync(int var, unsigned int delta,  
int width = warpSize);
```



异或shuffle

```
shfl_xor(int val, int laneMask, int width=warpSize);
// 将当前线程id与laneMask做位异或操作
```



流与并发

CUDA流

从主机代码中发起，在同一个GPU设备上执行的一系列异步
CUDA操作

CUDA流封装CUDA操作，维持操作在队列中的顺序，查询操作状态

CUDA流中的操作与主机程序异步执行

同一CUDA流中的操作具有严格执行顺序

使用多个CUDA流执行不同内核可以实现网格层面的并发执行

CUDA操作包括：

主机和设备间数据传输

内核执行

其它由主机代码发起，由GPU执行的操作

CUDA流类型

内核执行、数据传递都是在CUDA流中执行

CUDA流分为显示流、隐式流

如果没有显示定义流，在启动内核或传递数据时都使用隐式流

如果需要CUDA操作的重叠，就必须使用显示流

主机和GPU计算重叠

主机极端和GPU数据传递重叠

GPU计算和数据传递重叠

GPU计算重叠

流对象

CUDA平台流对象：cudaStream_t

typedef struct CUstream_st *cudaStream_t

流对象创建：cudaStreamCreate

流对象释放：cudaStreamDestroy

```
cudaStream stream;
cudaStreamCreate(&stream);
kernel<<<grid, block, 0, stream>>>();
cudaStreamDestroy(stream);
```

异步拷贝

异步数据拷贝在显示创建CUDA流中完成

异步数据拷贝过程与主机线程异步执行

用于异步数据拷贝的主机内存必须时页锁定内存

异步拷贝：cudaMemcpyAsync

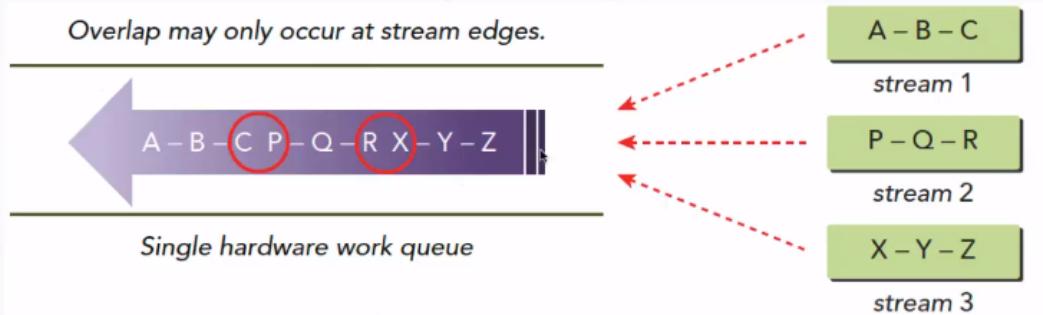
与主机同步：cudaStreamSynchronize

```
float *pinned_A;  
cudaHostAlloc((void **) &pinned_A, nBytes,  
cudaHostAllocDefault);  
cudaStream stream;  
cudaStreamCreate(&stream);  
float *d_A;  
cudaMalloc((void **) &d_A, nBytes);  
cudaMemcpyAsync(d_A, pinned_A,  
cudaMemcpyHostToDevice, stream);  
cudaStreamSynchronize(stream);
```

流调度

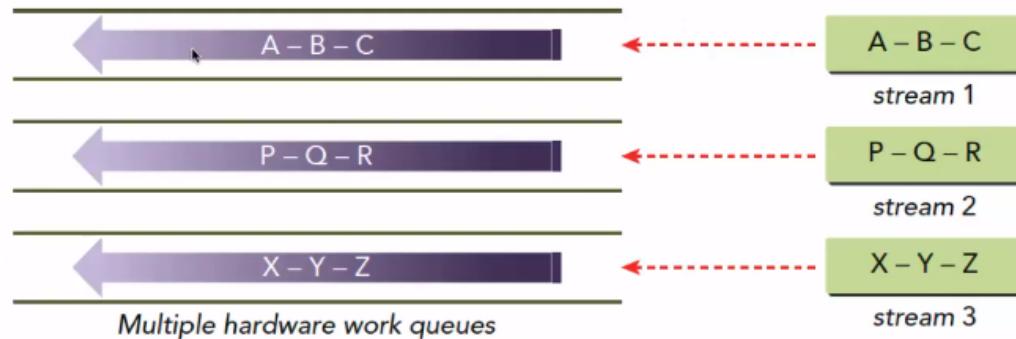
硬件队列

- ❖ Fermi架构中，GPU支持最高16路并发
- ❖ 硬件工作队列只有一个，实际的并发不能达到理论值



硬件队列

- ❖ Kepler架构中引入Hyper-Q，避免了单一工作队列问题
- ❖ Kepler架构支持32个工作队列



流优先级

优先级设置：cudaStreamCreateWithPriority

流优先级范围查询：cudaDeviceGetStreamPriorityRange

流优先级只影响内核执行，不影响数据拷贝的执行

```
int lowPriority, highPriority;
cudaDeviceStreamPriorityRange(&lowPriority,
&highPriority);
cudaStreamCreateWithPriority(&kernel_stream,
cudaStreamDefault, highPriority);
```

流事件

事件是CUDA流中与某个特定操作相关的标记

事件作用：同步流的执行，监控设备执行进度

CUDA平台提供了事件插入和查询功能

```
/*  cudaEvent_t 事件对象
    cudaEventCreate 事件创建
    cudaEventDestroy 事件销毁
*/
```

流事件同步

事件插入

CUDA流事件用于检测流的执行是否达到指定的操作点

流事件本身需要作为流操作插入流中

cudaEventRecord

流事件插入流后，当其关联的操作完成时，流事件会在主机线程中产生一个完成标志

事件等待

插入流中的事件可用于在主机线程中等待此事件完成

主机线程同步：cudaEventSynchronize

流操作时长计算

流操作时长是CUDA操作时长指CUDA流中完成某个操作所需时间

计算起、止事件的事件间隔：cudaEventElapsedTime

隐式和显式流顺序

除了内核执行外，隐式流中操作与主机线程同步

显式流中的操作与主机线程异步

显式流可分为阻塞流和非阻塞流

阻塞流中的操作可以被隐式流中的操作阻塞

非阻塞流中的操作不会被隐式流中的操作阻塞

显式流控制

使用cudaStreamCreate创建的显式流都是阻塞流

非阻塞显式流创建：cudaStreamCreateWithFlags

隐式同步

CUDA平台部分API接口内部对主机和设备线程执行做了同步

许多与内存操作相关的接口都包含了隐式同步机制

主机页锁定内存分配

设备内存分配

设备内存初始化

在设备上拷贝数据

修改一级缓存和共享内存配置

显式同步

CUDA平台提供不同层次的显式同步

设备同步：cudaDeviceSynchronize

流同步：cudaStreamSynchronize

流中的事件同步：cudaEventSynchronize

不同流同步：cudaStreamWaitEvent

可配置事件创建

创建：cudaEventCreateWithFlags

可以在主机代码和内核函数中调用此接口

可配置事件属性：

cudaEventDefault

cudaEventBlockingSync

cudaEventDisableTiming

cudaEventInterprocess

内核并发

在多个CUDA流中执行多个内核函数

执行内核的流可以是隐式流也可以是显式流

检测是否支持多内核并发执行：

cudaDeviceProp::concurrentKernels

多流多内核执行

方法：创建多个显式流，将不同内核函数分发到不同的显式流执行

多线程并发CUDA流

适用场景

在内核执行前、执行中、执行后由额外工作需要在主机代码中完成

通过OpenMP多线程可以充分的利用CPU、GPU的事件片

分发方法

建立OpenMP并行代码片段

获取线程id : `omp_get_thread_num()`

使用openmp 编译时 `nvcc -Xcompiler -fopenmp`

工作队列

概念

工作队列是主机和GPU间的硬件队列

通过此队列主机线程可以在GPU上启动流执行

修改工作队列数量

`CUDA_DEVICE_MAX_CONNECTIONS`

内核和数据拷贝并行

并行原理：

不同流可以在硬件层面并行

主机到设备和设备到主机的数据拷贝分属于不同的硬件队列

如果内核A执行时不会使用数据块B中的数据，那么内核A和数据B可以并行

并行时需要内核和数据在不同的流中执行

流回调函数

定义：

流回调函数是在主机嗲吗中提供的函数

流回调函数可以被加入CUDA流中

当流回调函数之前的所有流操作完成后，流回调函数就会被CUDA运行库调用

流回调函数时CPU与GPU同步的一种机制

流回调函数会阻塞后面的流操作

使用方法：

流回调函数注册：cudaStreamAddCallback

```
__host__ cudaError_t  
cudaStreamAddCallback(cudaStream_t stream,  
cudaStreamCallback_t callback, void *userData,  
unsigned int flags);
```

使用限制：

流回调函数内部不能调用CUDA API

流回调函数内部不能使用任何同步机制

指令级原语

底层指令优化

概念：

CUDA程序分为两类，IO限制和计算限制

计算能力以的单位时间内完成计算的次数衡量

CUDA内核函数代码在执行时被转换成底层指令

理解GPU底层指令在性能、数值精度、线程安全性方面的优缺点对于优化CUDA程序非常重要

内容：

浮点数：浮点数用于非整形计算，影响计算性能和精度

内置与标准函数：实现相同的数学运算，但内置函数保证了计算精度和计算性能

原子操作：保证同一变量在多个线程访问时的正确性

浮点数二进制编码

IEEE754标准

内置和标准函数

标准函数：可以在主机代码，也可以在设备代码中使用（如含在C语言中的数学库）

内置函数：只能在设备代码中调用

内置函数是经过编译器优化并具有与标准函数不同的指令

由于图形计算的需求，许多内置三角函数是直接在硬件层面实现

与标准函数比，由更高的运算速度，但数值精度更低

原子操作

CUDA平台的原子操作用于完成基本数学运算

当线程在某一变量上执行完原子操作，变量的状态就一定能达到线程所期望的状态

CUDA平台为32位、64位的全局内存和共享内存提供原子操作

OPERATION	FUNCTION	SUPPORTED TYPES
Addition	atomicAdd	int, unsigned int, unsigned long long int, float
Subtraction	atomicSub	int, unsigned int
Unconditional Swap	atomicExch	int, unsigned int, unsigned long long int, float
Minimum	atomicMin	int, unsigned int, unsigned long long int
Maximum	atomicMax	int, unsigned int, unsigned long long int
Increment	atomicInc	unsigned int
Decrement	atomicDec	unsigned int
Compare-And-Swap	atomicCAS	int, unsigned int, unsigned long long int
And	atomicAnd	int, unsigned int, unsigned long long int
Or	atomicOr	int, unsigned int, unsigned long long int
Xor	atomicXor	int, unsigned int, unsigned long long int

PTX代码

PTX代码生成命令：nvcc --ptx -o (target ptx file) (source file)

CAS原子操作

CAS是一种比较交换操作

CUDA提供的其他原子操作都可以通过CAS实现

CAS还可以用于自定义原子操作

CAS步骤

输入：内存地址、期望值、期望存储值

将当前内存地址中的值与期望值比较，如果一致，将期望存储值写入内存地址

返回未替换时内存中的值，如果返回值与期望值一致，就保证了替换是成功的

```
int atomicCAS(int *address, int compare, int val);
```

MAD指令优化

编译器优化MAD运算

优化方法：

编译器指令-fmad控制对MAD运算的优化

--fmad=true优化，--fmad=false 不优化，默认优化

原子操作性能损失

对全局内存、共享内存的原子操作结果在原子操作结束后对其余线程可见

原子操作指令没有缓存，直接读写全局内存、共享内存

多条线程同时执行原子操作时，冲突的线程会尝试多次执行原子操作

线程束中的线程在同一内存地址的原子操作必须串行执行

自定义原子操作

CUDA程序调试

CUDA程序调试可分为内核调试、内存调试

内核调试是在程序运行时探查内核执行流程，执行状态

内存调试目的在发现异常的程序行为，如非法内存访问，内存访问冲突

内核调试工具：cuda-gdb

内存调试：cuda-memcheck

焦点定义

- ❖ CUDA程序中包含多条线程，cuda-gdb调试会话一次只能调试一条线程
- ❖ cuda-gdb当前调试的线程称为调试焦点
- ❖ 如果当前焦点是主机线程，cuda-gdb所有调试命令只作用于主机线程
- ❖ 对GPU而言，调试焦点是设备上运行的线程
- ❖ GPU调试焦点由(kernel,block,thread)定义时，称为软件坐标
- ❖ GPU调试焦点由(device,sm,warp, lane)定义时，称为硬件坐标

单步执行

- ❖ 内核程序单步执行时以线程束为单位
- ❖ 如果要单步执行多个线程束，必须在期望的代码处加断点，然后继续执行程序
- ❖ 当内核中有线程屏障(thread barrier)时，系统会在线程屏障后一句代码处设置隐式断点，然后执行所有线程

断点类型

- ❖ 主机代码、内核代码设置断点方法一致
 - 符号断点(symbolic breakpoints)
 - 代码行断点(line breakpoints)
 - 内存地址断点(address breakpoints)
 - 内核入口断点(kernel breakpoints)
 - 条件断点(conditional breakpoints)
 - 监视点(watch points)

❖ 符号断点是在指定的函数入口设置断点

- ❖ 代码行断点使线程在指定的代码行处中断执行
- ❖ 内核入口断点在每个执行的内核函数的第一行代码处中断
- ❖ `set cuda break_on_launch [type]`

类型

- ❖ none
- ❖ application
- ❖ system
- ❖ all

- ❖ 当条件满足时，程序在所设定的断点处中断
- ❖ 条件表达式中可以使用程序中的任何变量包括CUDA内置变量
- ❖ 条件表达式中不能使用任何函数

用法

- ❖ if条件表达式
- ❖ cond条件表达式

print命令使用

- ❖ print命令可用于查看程序中任何变量值
 - cudaMalloc分配的全局内存
 - GPU中各种内存区域中的变量值
 - GPU内置变量值
- ❖ 查询GPU信息和CUDA程序状态
 - devices sms
 - warps lanes
 - kernels blocks
 - threads launch trace
 - launch children contexts

条件设置

- ❖ 设置过滤条件以查看指定内容

- device n sm n
- warp n lane n
- kernel n grid n
- block x[,y] 或者 block (x[,y])
- thread x[,y[,z]] 或者 thread (x[,y[,z]])
- breakpoint all 或者 breakpoint n

GPU寄存器

- ❖ 寄存器访问\$R<regnum>
- ❖ 寄存器内容查看 : info registers [\$R<regnum>]

上下文事件

- ❖ CUDA上下文(CUDA context)在创建、入栈、出栈、销毁时会触发上下文事件
- ❖ 上下文事件控制 : set cuda context_events

内核事件

- ❖ 内核启动、内核终止时会触发内核事件
- ❖ 内核事件包含内核ID，内核名，内核运行的设备
- ❖ set cuda kernel_events [options]
 - none
 - application
 - system
 - all
- ❖ 内核事件深度控制：set cuda kernel_events_depth [num]

CUDA内存调试

CUDA程序中线程数量众多，容易出现内存访问错误，线程竞争等问题

- ❖ cuda-memcheck工具可用于检测CUDA程序中各种内存访问错误
- ❖ cuda-memcheck是一个程序框架结构
 - Memcheck用于检测内存访问错误和内存泄漏
 - Racecheck用于共享内存访问错误检测
 - Initcheck用于检测全局内存未初始化错误
 - Synccheck用于检测线程同步错误

程序编译

- ❖ -lineinfo：包含文件名和代码行号信息
- ❖ 符号信息，用于对主机程序的堆栈追踪
 - Linux -Xcompiler -rdynamic
 - Windows -Xcompiler /Zi

- ❖ cuda-memcheck能够定位到出现内存错误的代码行数、访问线程信息、错误原因

线程竞争检测

- ❖ -lineinfo : 包含文件名和代码行号信息
- ❖ 符号信息，用于对主机程序的堆栈追踪
 - Linux -Xcompiler -rdynamic
 - Windows -Xcompiler /Zi
- ❖ 运行cuda-memcheck --tool racecheck

检测模式

- ❖ cuda-gdb提供了对驱动API和运行时API调用结果的自动检测
- ❖ 检测模式设置：set cuda api_failures
 - hide
 - ignore
 - stop