CUDA专家手册

2.硬件架构

- **write-combined memory** / 写结合锁页内存：当CPU对一块内存中的数据进行处理时，会将这块内存上的数据缓存到CPU的L1、L2 Cache中，并且需要监视这块内存中数据的更改，以保证缓存的一致性。 一般情况下，这种机制可以减少CPU对内存的访问，但在“CPU生产数据，GPU消费数据”模型中，CPU只需要写这块内存数据即可。此时不需要维护缓存一致性，对内存的监视反而会降低性能。通过write-combined memory，就可以不使用CPU的L1、L2 Cache对一块pinned memory中的数据进行缓冲，而将Cache资源留给其他程序使用。
- **点对点映射**：可以是GPU读取另一个GPU的内存。仅支持启用UVA的平台，且只对连接到相同I/O集线器的GPU有效。通过显式映射来使GPU能看到对方的内存。

2.5 CPU/GPU交互

- 锁页主机内存：GPU可以通过DMA来访问CPU中的锁页内存。锁页可以使硬件外设直接访问CPU内存，而避免过多的复制操作。被锁定的页面被OS标记为不可换出，所以设备驱动程序给这些外设编程时，可以使用页面的物理地址直接访问。
- 命令缓冲区：由（CPU）CUDA驱动程序写入命令，GPU从此循环缓冲区读取命令并控制其执行。
- CPU受限和GPU受限型应用程序分别代表，命令缓冲区的消费者等待和生产者等待。
- 因为内核程序的所有启动都是异步的，内存复制也能是复制的。所以最粗粒度的CPU/GPU并行以下、还有数据传输和内核执行并行。
- CUDA_LAUNCH_BLOCKING环境变量会迫使所有启动的内核同步。
- **CPU/GPU同步**：由于应用程序不知道给定的CUDA内核程序会运行多久，所以GPU本身必须给CPU汇报工作进度。
    - CUDA驱动程序维护了一个**单调增的整数进度值**。
    - **每个主要GPU操作后都跟着一个将新进度写入共享同步位置的命令**。
    - **同步位置**和命令缓冲区都位于锁页主机内存上。
- 上下文范围的同步通过简单调用cuCtxSynchronize()、cudaThreadSynchronize()等函数来检查GPU请求的最近同步值，并一直等到同步位置获得该值。
- CUDA event也是这个原理。cuEventRecord函数将一个命令加入队列使一个新的同步值写入共享同步位，cuEventQuery和cuEventSynchronize分别用于检查和等待这个事件的同步值。
- CUDA通过 **阻塞同步** 来等待同步值。这是一种中断机制，不满足条件会挂起CPU线程。
- **事件作为 阻塞 的重要手段**，粒度更细且可以与任何类型的CUDA上下文进行无缝互操作。
- GPU有多个引擎，驱动程序写入的命令会被分发到同时运行的不同引擎中。每个引擎都有自己的命令缓冲区和共享同步值。
    - 分发任务由主机接口完成，并实现同步。
- 因为同步位置在主机内存上，所以可以被多GPU访问，从而通过cudaStreamWaitEvent函数实现GPU间同步。

3.软件架构

3.1 软件层

- CUDA运行时（CUDART）是CUDA语言集成环境 使用的库。每个版本的CUDA工具集都有特定的CUDA运行时版本，兼容才能跑。
- CUDA驱动程序向后兼容，提供一个 驱动程序API，在cuda.h中。
    - 驱动版本应该>=运行时版本，否则CUDA应用程序会初始化失败，导致cudaErrorInsufficientDriver。

```
#include<cuda.h>    // 驱动api头文件
#include<cuda_runtime.h>
int main(){
    // 初始化驱动程序API 在调用任何驱动API之前必须初始化
    cuInit(0);
    // 必须创建一个附加到特定设备的CUDA上下文，并使其成为当前调用主机线程
    CUdevice cuDevice;      int device = 0;
    cuDeviceGet(&cuDevice, device);
    CUcontext cuCtx;
    cuCtxCreate(&cuContext, 0, cuDevice);
    // 在CUDA上下文中，内核作为PTX或二进制对象 由主机代码显式加载
    // 因此C++写的内核必须单独编译成PTX或二进制对象
    CUmodule cuModule;
    cuModuleLoad(&cuModule, "VecAdd.ptx");
    // 从module中获得函数的handle
    CUfunction vecAdd;
    cuModuleGetFunction(&vecAdd, cuModule, "VecAdd");
    // 内核使用API入口点启动
    void *args[] = {...};
    cuLaunchKernel(vecAdd, ...);
}
```

- **用户模式客户端驱动程序**：为了避免硬件工作提交带来的过多内核陷入，GPU分配特定的寄存器（如给硬件提交工作的硬件寄存器）到用户模式，确保驱动程序可以以峰值的效率执行。
    - 通过对此类寄存器的保护程序避免用户代码的流氓写入，确保内核代码只会访问可使用的内存。
- nvcc可以通过调用PTX编译器ptxas离线编译PTX，生成特定GPU版本的cubin。cubin可以用cuobjdump反汇编生成sass代码。
    - **P**arallel **T**hread e**X**ecution
- PTX同样可以在线编译(JIT)，--fatbin编译选项（默认）会自启动在线编译。每个内核的 .cubin和ptx都包含在可执行文件里，如果硬件不支持cubin文件里的.cubin，驱动就会转向编译PTX，并将结果缓存在磁盘里。
- PTX可以在运行时通过显式调用cuModuleLoadEx()编译。驱动API不会自动嵌入或加载GPU二进制。.cubin和.ptx均作为参数传递给cuModuleLoadEx函数，如果.cubin不兼容机器架构，会返回错误。

3.2 设备和初始化

- CUDA初始化的两种方式：显式调用cuInit、隐式调用CUDA运行时API

- CUDA驱动程序会在初始化完成后，枚举可用设备并创建一个全局数据结构体（包含设备名称和不可变参数s）。
- CUDA上下文数量：默认模式中多个上下文可被创建，独占模式只有一个CUDA上下文能被设备创建，禁止模式禁止设备创建上下文。
- CUDA运行时程序可以运行在不能运行CUDA的机器上或没CUDA的机器上。但CUDA驱动API，要求驱动程序二进制文件是可用的。

3.3 上下文

- 上下文是管理CUDA程序中所有对象生命周期的容器，类似于CPU进程。包括
  - 内存分配、模块、流、事件、纹理和表面引用、使用localMem的kernel的设备内存
  - 进行调试分析同步操作时用的内部资源、换页内存复制中所使用的锁定中转缓冲区
- CUDA运行时不提供对上下文的直接访问，而通过延迟初始化来创建上下文。
- 对于驱动程序API中每个指定上下文状态的函数，CUDA运行时把上下文和设备同等对待。
- 一个设备可以是多个CPU线程的当前上下文。
- 所有与CUDA上下文相关的分配资源都在上下文被销毁时同时销毁。一个CUDA上下文创建的资源可能不能被其他CUDA上下文使用。
- CUDA尽力避免lazy allocation。如果CUDA不能分配换页内存所需要的锁页中转缓冲区，上下文创建就失败了。
  - 锁页中转缓冲区的分配操作在上下文创建时发生。
  - 少量情况下，CUDA不会预分配一个给定操作所需的全部资源。内核启动所需的本地内存数量可能被限制，所以CUDA不会预分配最大理论数量的内存。当默认分配的少于需要时，内核启动可能失败。
- **当前上下文栈**：多数CUDA入口点并不含有上下文参数。取而代之的，它们在CPU线程的"当前上下文"上进行操作，它存储在线程的本地存储句柄里。在驱动程序API中，每一个CPU线程有一个当前上下文栈，创建一个上下文的同时将这个新上下文压入栈。
  - 栈保证了每个时刻，只有一个漂浮上下文（就是弹出的那个），从而实现多个上下文的驱动
- 当前上下文栈的3个主要功能：
  - 单线程应用可以驱动多个GPU上下文
  - 库可以创建并管理自己的上下文而不需要干涉调用者的上下文。
  - 库不知道调用它的CPU线程信息
- cuCtxSetLimit、cuCtxGetLimit、cuCtxSetCacheConfig、cuFuncSetCacheConfig等函数可以设置上下文状态。

3.4 模块与函数

- 模块是代码和一同加载的数据的集合。模块只在CUDA驱动API中可用。
  - 类似于Windows中的动态链接库和Linux中的动态共享对象。
  - CUDA运行时和CUDA上下文不显式支持模块。
- nvcc直接生成可以被加载为CUDA模块的文件
  - 面向特定SM的.cubin文件
  - 通过驱动程序可以编译为硬件执行代码的.ptx
- CUDA提供加载模块的API，模块能嵌入在可执行文件中。

    - 模块的表示形式：以NULL结尾的字符串
- 一旦模块被加载，应用程序便查询包含在其中的资源。
    - 全局变量
    - 内核
    - 纹理引用

3.5 内核

- cuFuncGetAttribute用来查询指定的属性：
    - 函数每个线程使用的寄存器数量
    - 静态分配共享内存的数量
    - PTX和二进制架构版本
- extern C可以用来禁止C++中默认的改名行为。否则cuModuleGetFunction()必须使用改编的名称
- 运行时不能递增的加载或卸载模块，因为运行时创建的可执行文件被加载时，会在主机内存创建一个全局数据结构体，用来一次性创建CUDA资源，全局数据在CUDA运行时中属于进程级共享。
- 一级缓存和共享内存的配置：cuCtxSetCacheConfig、cudaDeviceSetCacheConfig

3.6 设备内存

- CUDA硬件不支持请求式换页，所以，所有的内存分配都被真实的物理内存支持着。

3.7 流与事件

- 流支持GPU上的多内核并发执行，支持多GPU并发执行。
- CUDA流对多个处理单元并发执行的粗粒度管理：
    - CPU与GPU
    - 在SM处理时，用复制引擎执行DMA操作
    - SM
    - 并发内核
    - 多GPU并发
- 一个CUDA流中的操作按顺序执行。
- 复制引擎只有2个，为了防止多流并发中，每个流不同引擎间的同步导致的复制引擎并发中断，应用软件必须对多流执行软件流水线操作。HyperQ实际上消除了软件流水线需求。
- 流回调：CPU/GPU同步机制。
- NULL流：任何一个异步的内存复制函数都可以使用NULL流作为参数，而且内存复制会等到GPU之前的操作全部完成才开始。所有流的内存复制函数都是异步的。
- 一旦流操作使用NULL流初始化，应用程序必须使用同步函数（sync）来确保操作在执行下一步之前完成。
- 事件表示了另一个同步机制。CUDA event计时比CPU计时更准确，因为不会受限于伪相关事件。最好在NULL流中使用。
- 如果为计时提供足量的工作，计时本身的时间就可以忽略。

3.8 主机内存

- DMA的三个优势：

    - 避免了数据复制
    - 硬件与CPU并发操作
    - DMA可以使硬件获得更好的总线表现
- 一旦锁页，驱动程序就能使用DMA硬件引用内存的物理地址。
    - cuMemHostAlloc、cudaHostAlloc
- 异步内存复制只在锁页内存上工作
- if UVA -> 所有锁页内存 全部可分享
- 映射锁页内存被映射到CUDA的上下文地址空间中，所有内核可读写。由于主机和设备共享该内存池，所有不用显式复制来交换数据。
- if UVA -> 所有锁页内存 都被映射
- cuMemHostRegister、cudaHostRegister

3.9 CUDA数组和纹理操作

- 数组为2D和3D局部性做了优化。对于稀疏访问模式的应用，特别是维度局部性的程序，CUDA数据有明显优势。
- CUDA数组和设备内存从相同的物理内存池分配，但CUDA数组不消耗地址空间
- 数据传输有在主机内存上保持一个等宽形式内存的需求时，CUDA数组是最好的处理办法。
- 纹理引用是CUDA对象，用来设置纹理硬件 解释实际内存内容的。
- 纹理引用是个中间层，这使得多个纹理引用使用不同的属性引用相同的内存有效。

```
texture<Type, Dimension, ReadMode> Name;
```

- 纹理在使用前需要绑定在实际内存上，设备内存或CUDA数组。绑CUDA数组会更有优势
- 下述情况，绑设备内存会获利：
    - 作为带宽聚合器支持纹理缓存
    - 使应用程序满足合并访问限制
- 运行时：cudaBindTexture、cudaBindTexture2D、cudaBindTextureToArray
- 驱动：cuTexRefSetAddress、cuTexRefSetAddress2D、cuTexRefSetArray
- 纹理和表面是俩东西。。。

4.内存

4.1 主机内存

- CUDA为了能复制可换页内存中的数据，使用了一对中转缓冲区，它们是锁页的，在CUDA上下文创建时由驱动程序分配。

4.2 全局内存

- CUDA程序中的主机代码可以在设备指针上执行指针算数运算，但不能解引用这些指针。
- CUdeviceptr宽度同主机指针，uintptr_t（<stdint.h>）可以用来做主机和设备指针间的中间类型，用来切换。

动态内存分配

- cuMemAlloc、cuMemFree
- 分配全局内存成本非常高昂！因此，在性能要求很高的代码中避免分配或释放全局内存是一个较好的做法。
- 等步长(pitch)内存分配：数组宽度会补到最近的2^X幂的倍数，元素地址计算方式：
  - cudaMallocPitch / cuMemAllocPitch
  - cudaMalloc3D

```
inline T* getElement(T* base, size_t Pitch, int row, int col){
    return (T*)((char*)base + row * Pitch) + col;
}
```

- 费米架构能在内核里调用malloc和free动态的申请释放全局内存，这可能很慢。

静态内存分配

- 即 **device** 标记的内存, 由CUDA驱动程序在模块加载时分配。
- 静态内存复制：cudaMemcpyToSymbol 、 cudaMemcpyFromSymbol
- 获得静态内存分配的指针：
  - 运行时：cudaGetSymbolAddress
  - 驱动：cuModuleGetGlobal

内存初始化

- cu（da）Memset*由内核实现，所以默认是异步的。
- 可以用Async()形式的函数，使之流内顺序执行。

4.2.1 指针

- 查询指针驻留的地址空间：cuMemGetAddressRange()
- cuPointerGetAttribute查询指针信息
- 点对点内存访问：映射其他GPU中的内存
  - 需要相同的架构/硬件
  - 连接在相同的I/O集线器上
  - 启用UVA

4.2.9 事务合并

- 事务在线程束的基础上被合并。
- 要求基址对齐，字数为32的倍数，访问地址连续。
- 一级缓存的缓存行为128字节宽，所以开启一级缓存的事务是128字节的。
- 仅有二级缓存的时候，最小可以是32字节事务。
- -Xptxas-dlcm=cg只使用二级缓存
- volatile指针访问内存会导致所有缓存失效，并重新为数据做缓存。
- 内存峰值带宽：

```
template<typename T, const int n>
__global__ void GlobalWrites(T* out, T value, size_t N){
    size_t i;
    for(i = n*blockDim.x*blockIdx.x+threadIdx.x; i < N-n*blockDim.x*gridDim.x; i
+= n*blockDim.x*gridDim.x){
        for(size_t j = 0; j < n; j++){
            size_t index = i+j*blockDim.x;
            out[index] = value;
        }
    }
    for(size_t j = 0; j < n; j++){
        size_t index = i+j*blockDim.x;
        if(index < N){
            out[index] = value;
        }
    }
}
```

```
#include <cuda_runtime.h>
#include <iostream>

#define GRID_SIZE 4800
#define BLOCK_SIZE 512  // [128, 256, 512]
#define UNROLL 8
#define DATA_NUM (GRID_SIZE*BLOCK_SIZE*UNROLL*8)    // 8是外层循环的次数

int main(){
    GlobalWrites<float, UNROLL><<<GRID_SIZE, BLOCK_SIZE>>>(d_out, 3.14,
DATA_NUM);
}
```

- 原子操作：CUDA支持int atomicCAS(int *address, int expected, int value);

```
void enter_spinlock(int * addr){
    while atomicCAS(addr, 0, 1);
}
void leave_spinlock(int * addr){
    atomicExch(m_p, 0);
}
```

- 开发者不能假定一次内核启动的所有线程都是活跃的，因为放不下那么多。
    - 同样类似的情况，是分支的线程。
- CUDA里实现同步，一般不会用上面的spinlock，容易死锁。可以用syncthreads和共享内存。
- 对于不能很好满足合并内存访问限制的应用程序，纹理映射硬件是一个令人满意的选择。
    - 它虽然不能提供比对齐合并更快的性能，但对不规则访问的性能很好。
    - 纹理缓存和其他缓存资源相独立。
- const restrict 和 __ldg() 可以通过纹理缓存层读取全局内存，无需绑定纹理引用。

4.3 常量内存

4.4 本地内存

- 本地内存包括CUDA内核中每一个线程中的栈。它容纳寄存器溢出的数据，保存编译器不能解析其索引的数组，实现ABI。
    - 寄存器溢出会导致两项开销：指令数增加和内存传输数量增加
- 内核启动使用超过默认的本地内存分配数，会导致CUDA驱动必须分配一块新的本地内存缓冲区。这可能需要额外的时间，CPU/GPU同步，甚至导致启动失败。

4.6 共享内存

- 共享内存的访问比寄存器访问速度慢十倍，但比全局内存快十倍。
- 共享内存能静态和动态分配。
    - extern **shared** char arr[];
- 在束同步编程中共享内存变量必须声明为volatile，避免编译器优化造成的错误。

4.7 内存复制

- 涉及主机内存复制的操作默认都是同步的。
- 任何不涉及主机内存的复制都是异步的（D2D/D2array）。

5.流与事件

- 在一个给定流中，操作顺序进行。在不同流上，操作能并行。
- 事件能提供GPU多个引擎间的同步，多个GPU间的同步。

5.1 异步内存复制

- 启动双临时缓冲区，CPU写入一个缓冲区，GPU通过DMA读取另一个缓冲区。

5.2 NULL流和并发中断

- 流中的操作对于主机来说都是异步的。
- null流中的操作需要等到之前的所有GPU操作完成后才会启动。
- 非空流包含阻塞流和非阻塞流。cuStreamCreate默认的是阻塞流。如果需要非阻塞流，需要制定cuStreamCreate的CUDA_STREAM_NON_BLOCKING或cudaStreamCreatewithFlags的cudaStreamNonBlocking标记。
- 非阻塞流不会阻塞空流，所有要创建一个不需要与NULL流同步的流，可以创建非空非阻塞流。

5.3 事件：CPU/GPU同步

-