



中山大學  
SUN YAT-SEN UNIVERSITY



# 多核程序设计与实践

## 流与并发

陶钧

[taoj23@mail.sysu.edu.cn](mailto:taoj23@mail.sysu.edu.cn)

中山大学 数据科学与计算机学院  
国家超级计算广州中心

- 第二次作业说明
- 流与并发

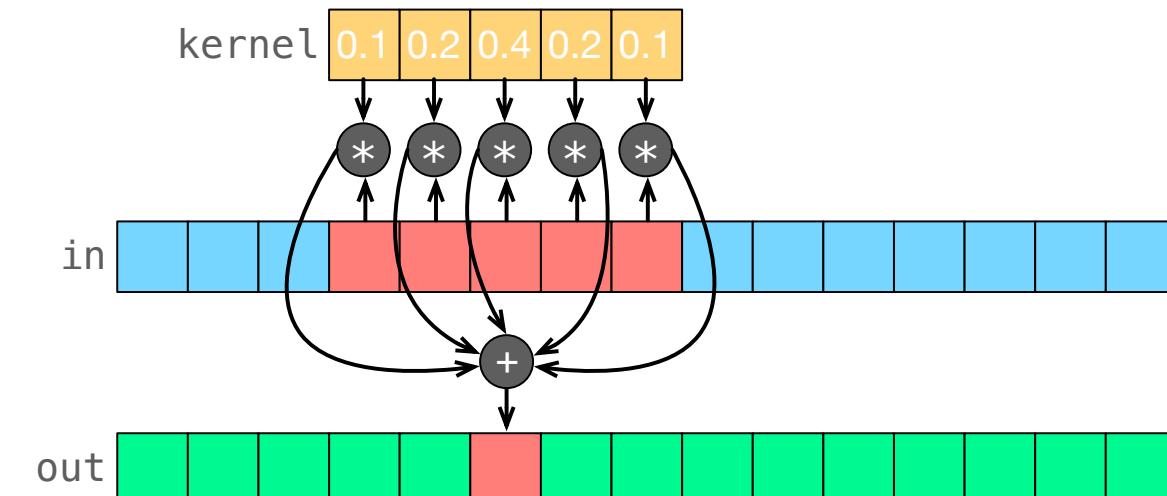


## ● 图像卷积

- 使用第一次作业中计算的二维高斯函数作为核函数对灰度图像进行卷积操作
  - 高斯平滑
- 一维卷积举例
  - 设卷积核 (kernel) 大小为 5
  - 求  $out[5]$  时，使用  $in[5]$  为中心的 5 个数
  - 即  $in[3 \dots 7]$  分别乘 kernel 中相应数字
  - 对积的结果求和
- 二维卷积

$$out[i][j] = \sum_{y=-3s}^{3s} \sum_{x=-3s}^{3s} k[y][x] \cdot in[i+y][j+x]$$

- 注意：这里假设 kernel k 的中心为 [0,0] (实际编程中为 [3s, 3s])



## ● 输入输出

- 详情见课程github页面
  - [https://github.com/multicoresysu/multicore\\_sysu\\_slides/blob/master/%E5%A4%9A%E6%A0%88%E6%A0%BC%E8%A7%A3%E8%A7%A3%E9%A2%9F.md](https://github.com/multicoresysu/multicore_sysu_slides/blob/master/%E5%A4%9A%E6%A0%88%E6%A0%BC%E8%A7%A3%E8%A7%A3%E9%A2%9F.md)
- 输入：灰度图像文件路径 s(卷积核参数，同第一次作业)
  - 灰度图像为二进制文件，其格式为长、宽、浮点数组（一维排列）
  - 读取灰度图像的C代码见github说明
- 输出：卷积后的图像
  - 输出结果的代码见github说明

## ● 编译、文件打包与测试脚本

- 使用统一编译语句编译（模板文件目录见github）
- 使用tar zcvf打包（助教将使用tar zxvf解压）
- 测试脚本见github，请同学们在提交前使用测试脚本在学院集群上进行测试

● 正确性		
– CUDA	40	
● 性能		
– CUDA	30	
– 根据实际搜集的运行时间分布决定		
– 标准将与成绩一并公布		
● 编程规范		
– 初始分	10	
– 缺少文件头	-5	
– 缺少函数头	-5	
– 换行没有正确缩进	-5	
– 函数过长	-5	
● 书面报告		
– 解释程序设计逻辑	10	
– 讨论参数对程序性能的影响	10	
– 讨论性能优化方法	+10	
● 提交作业		
– 邮箱: <b>MulticoreSYSU@163. com</b>		
– 截止时间		
• <b>5月27日晚23:59</b>		
• 如需使用slip days, 请于 <b>截止时间前</b> 将需要使用的天数发送至提交作业邮箱		

- 流与并发概述
- 同步与异步
- 流与并发
- 流同步



## ● 此前：内核级并发

- 单一任务（内核）被多个GPU线程执行
- 性能提升：使单一内核中的不同任务/资源同时执行/利用
  - 提高占用率：利用线程执行掩盖全局内存访问延时
  - Waves and Tails、消除分支分流：提高线程利用率
  - 合并与对齐访问：提高全局内存带宽利用率
  - 消除存储体冲突：提高共享内存带宽利用率



- 此前：内核级并发

- 单一任务（内核）被多个GPU线程执行
- 性能提升：使单一内核中的不同任务/资源同时执行/利用

- 流与并发：多个内核同时执行

- 使多个内核中的不同任务并发，进一步提高利用率
  - 例1：并行归约中，后期使用的线程数量越来越少，空置线程越来越多
    - 是否可以利用空闲线程执行其他内核？
  - 例2：CUDA典型的编程模式：
    - 1. 将输入数据从主机拷贝至设备
    - 2. 在设备上执行内核
    - 3. 将结果从设备拷贝回主机
    - 拷贝与内核执行是否可以同时进行？

- 流与并发概述
- 同步与异步
- 流与并发
- 流同步
- 多GPU编程



- 阻塞 (blocking) 与非阻塞 (non-blocking) 函数调用

- 阻塞调用:

- 同步
    - 函数执行完成后控制权交还主线程 (调用线程)
    - 函数以串行方式调用

- 非阻塞调用

- 异步
    - 函数调用后控制权即交还主线程

- 异步执行的优势

- 使不同设备上的执行及数据拷贝可以同时进行
    - 不仅仅是GPU与CPU, 也可以是更耗时的硬盘访问及网络访问

## ● 异步执行举例

### – CPU超线程

- 编写单线程程序时，我们认为代码是同步操作
- 编译器在编译时可能产生重叠执行的操作以提高资源利用率
- 与非重叠代码产生同样的结果

### – CPU多线程

- 由多个处理器同时执行多个线程
- 需要自行运用同步机制解决竞争条件
  - 例如，在OpenMP中使用临界区（critical section）

### – CUDA线程束执行

- 束内线程指令同步执行
- 多个线程束之间为异步执行（在同一个SM上交替执行）
- 使用 `__syncthreads()` 同步以解决竞争条件

## ● CUDA主机端与设备端

- 大多数CUDA主机端函数为同步（阻塞）
- 主机端的异步调用
  - 核函数调用
  - 设备内的cudaMemcpy (cudaMemcpyDeviceToDevice)
  - 从主机到内存小于64KB的cudaMemcpy
  - 异步内存拷贝与流
- 以下情况异步执行将被阻塞
  - 调用**deviceSynchronize()**同步
  - 新的核函数调用（隐式同步）
  - 主机与内存间的内存拷贝（隐式同步）

- 举例：下列语句中，哪些调用是异步执行的？

```
//copy data to device
cudaMemcpy(d_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size * sizeof(int), cudaMemcpyHostToDevice);

//execute kernels on device
kernel_1<<<blocks, threads>>>(d_a, d_b);
kernel_2<<<blocks, threads>>>(d_b, d_c);

//copy back result data
cudaMemcpy(c, d_c, size * sizeof(int), cudaMemcpyDeviceToHost);
```



## ● 举例：下列语句中，哪些调用是异步执行的？

```
//copy data to device
cudaMemcpy(d_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size * sizeof(int), cudaMemcpyHostToDevice);

//execute kernels on device
kernel_1<<<blocks, threads>>>(d_a, d_b);
kernel_2<<<blocks, threads>>>(d_b, d_c);

//copy back result data
cudaMemcpy(c, d_c, size * sizeof(int), cudaMemcpyDeviceToHost);
```

完全同步：  
所有函数串行执行



- 举例：下列语句中，哪些调用是异步执行的？

```
//copy data to device
cudaMemcpy(d_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size * sizeof(int), cudaMemcpyHostToDevice);

//execute kernels on device
kernel<<<blocks, threads>>>(d_c, d_a, d_b);

//host execution
host_func();

//copy back result data
cudaMemcpy(c, d_c, size * sizeof(int), cudaMemcpyDeviceToHost);
```



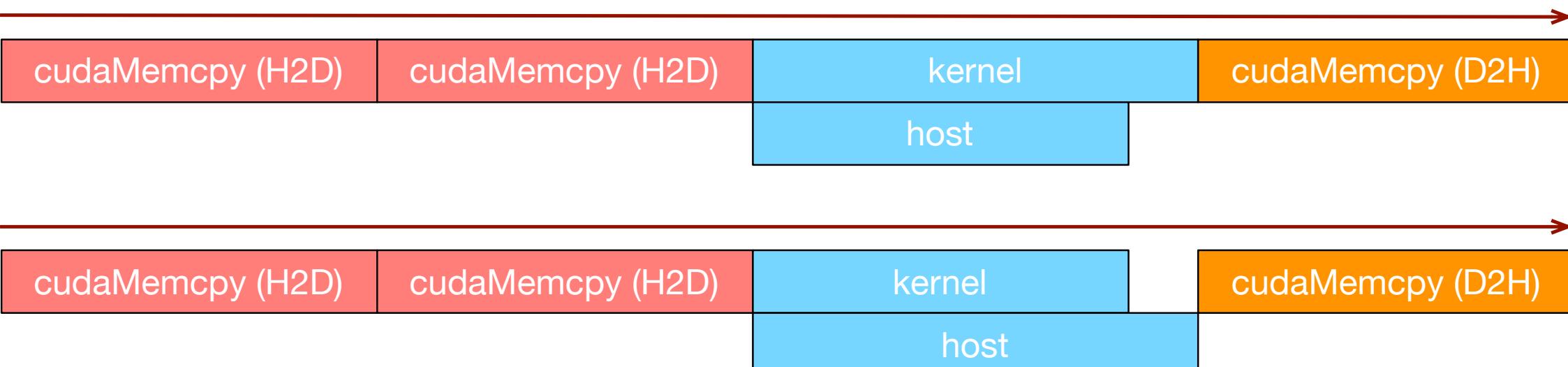
● 举例：下列语句中，哪些调用是异步执行的？

```
//copy data to device
cudaMemcpy(d_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size * sizeof(int), cudaMemcpyHostToDevice);

//execute kernels on device
kernel<<<blocks, threads>>>(d_c, d_a, d_b); 核函数为异步执行！

//host execution
host_func();

//copy back result data
cudaMemcpy(c, d_c, size * sizeof(int), cudaMemcpyDeviceToHost);
```



- 流与并发概述
- 同步与异步
- 流与并发
- 流同步
- 多GPU编程



- CUDA设备上通常具有异步核函数执行与内存拷贝引擎
  - 允许核函数执行的同时拷贝数据
  - 具有双工PCIe总线的设备可同时执行双向数据拷贝
    - 如Kepler和Maxwell核心的显卡
    - PCIe上行 (D2H)
    - PCIe下行 (H2D)
- 所有计算能力2.0+的设备都支持多个核函数同时调用
  - GPU上的多任务并行
  - 每个核函数完成一个任务
  - 存在多个规模较小的任务时，能显著提升性能

## ● CUDA流

- 所有CUDA操作都在流中显式或隐式运行
  - 内核执行和内存操作
  - 隐式声明的流（空流/默认流）
  - 显式声明的流（非空流）
    - 阻塞流与非阻塞流
- CUDA流指明了操作在设备上进入调度队列的方式
- 在同一流中的操作将按顺序执行，执行时间之间没有重叠（FIFO）
- 不同流中的操作可同时执行，执行顺序互不影响（非空流）

```
// create a handle for the stream
cudaStream_t stream;
//create the stream
cudaStreamCreate(&stream);

//do some work in the stream ...

//destroy the stream (blocks host until stream is complete)
cudaStreamDestroy(stream);
```

- 内核执行所对应的流可由执行配置中的第4个参数指明
  - `kernel<<<grid, block, 0, stream>>>();`
- 默认流为唯一的同步流
  - 将block其他流的执行

```
kernel_1<<<grid, block, 0>>>();
```

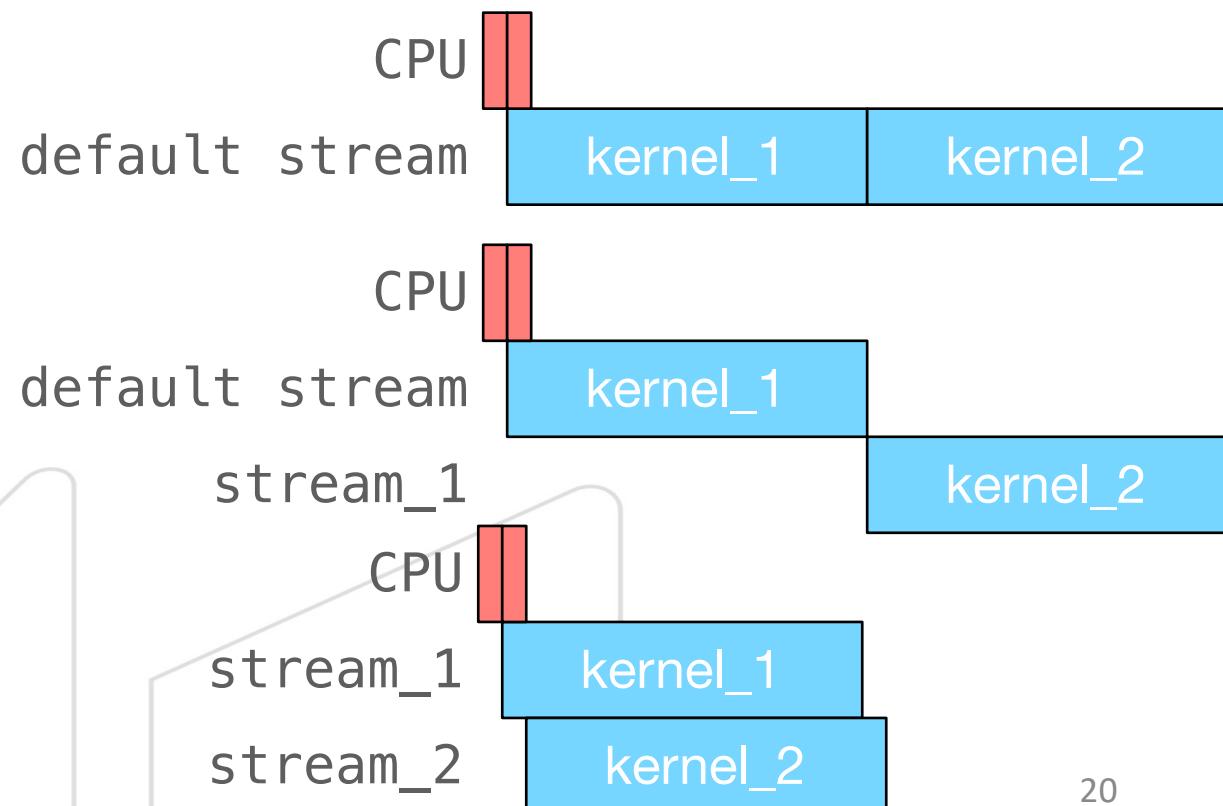
```
kernel_2<<<grid, block, 0>>>();
```

```
kernel_1<<<grid, block, 0>>>();
```

```
kernel_2<<<grid, block, 0, stream_1>>>();
```

```
kernel_1<<<grid, block, 0, stream_1>>>();
```

```
kernel_2<<<grid, block, 0, stream_2>>>();
```



## ● 异步内存分配

- CUDA只允许对**锁页内存**进行异步操作
- 可分页内存
  - 使用**malloc()**分配与**free()**释放
- 锁页内存
  - 不能被交换到磁盘上
  - 分配开销更高，但在传输大量数据时通常能达到的带宽更大
  - 使用**cudaMallocHost()**分配与**cudaFreeHost()**释放
  - 也可以使用**cudaHostRegister()**将普通内存注册为锁页内存
    - 使用**cudaHostUnregister()**取消注册
    - 非常慢

## ● 异步内存拷贝

### – 使用 **cudaMemcpyAsync()** 进行拷贝

- 需要在参数中指定其对应的CUDA流
- 将拷贝操作置于相应的流中随即将控制权交回主机
- 只能对锁页内存使用
- 只能在非空流中使用

```
cudaStreamCreate(&stream_1);
cudaMallocHost(&h_a, size);
cudaMalloc(&d_a, size);

cudaMemcpyAsync(d_a, h_a, size, cudaMemcpyHostToDevice, stream_1);
//work in other streams ...

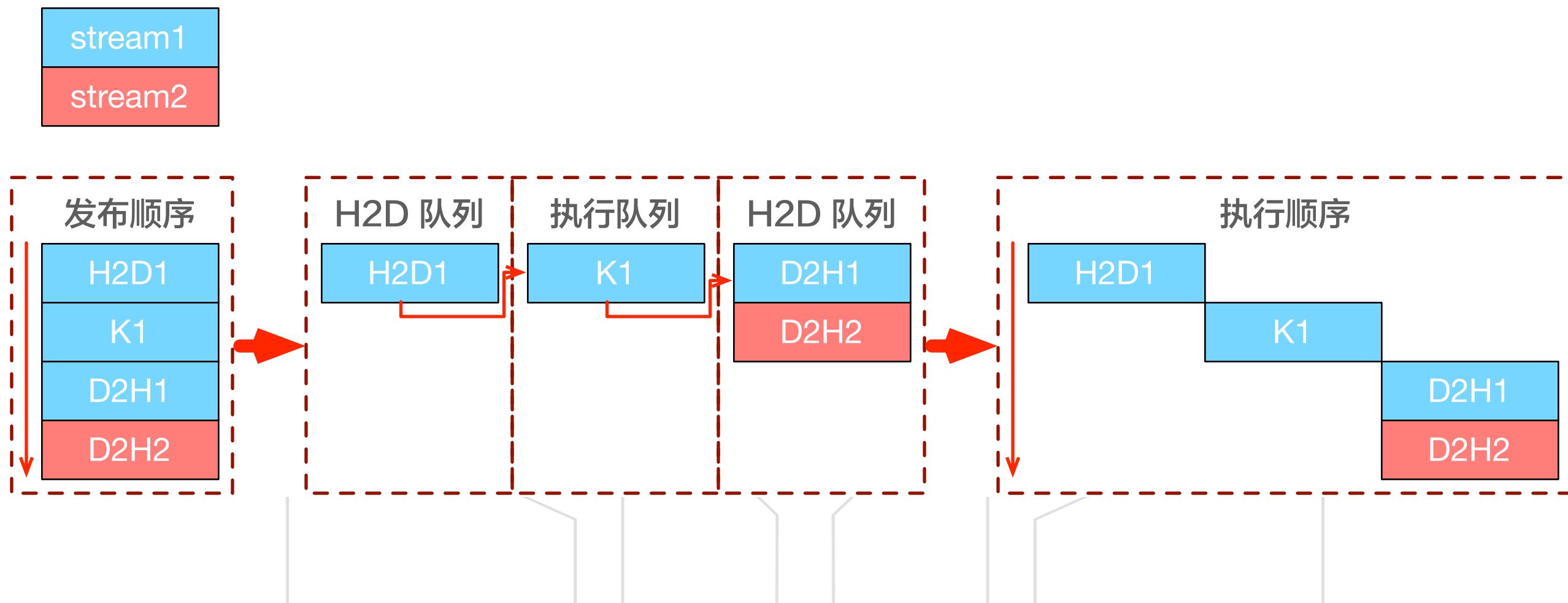
cudaStreamDestroy(stream_1);
```

## ● 流调度

- CUDA操作根据其发布到流中的顺序进入硬件工作队列
  - 发布顺序很重要 (FIFO)
  - 核函数调用、上行、下行内存操作被发布到不同队列
- 当满足以下情况时，操作将出列执行：
  - 同一个流中的前序操作已完成
  - 同一个队列中的前序操作已完成
  - 具备执行所需要的资源
    - 位于不同流中的不同核函数可同时执行！
- 阻塞 (Blocking) 操作
  - 同步流 (非空流) 、 cudaMemcpy

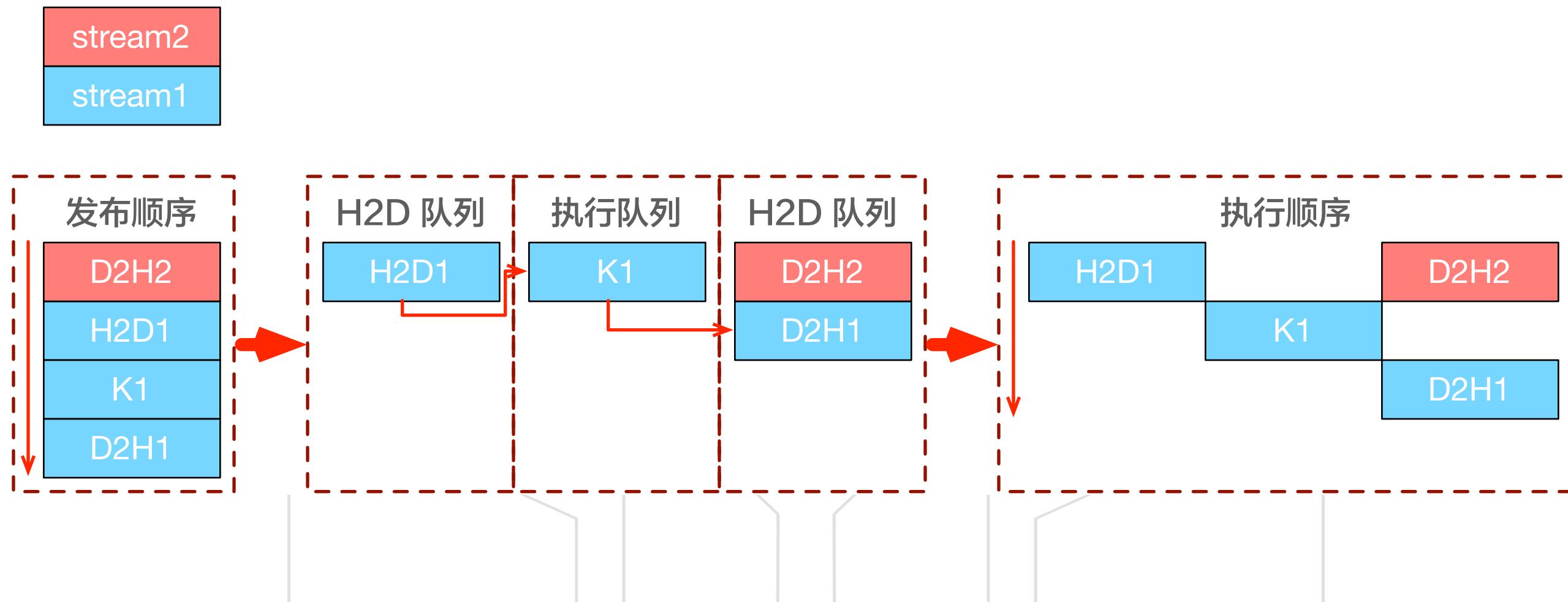
## ● 流调度举例

- stream2中的操作D2H2 (device to host 2) 没有并发
  - 被D2H1所阻塞



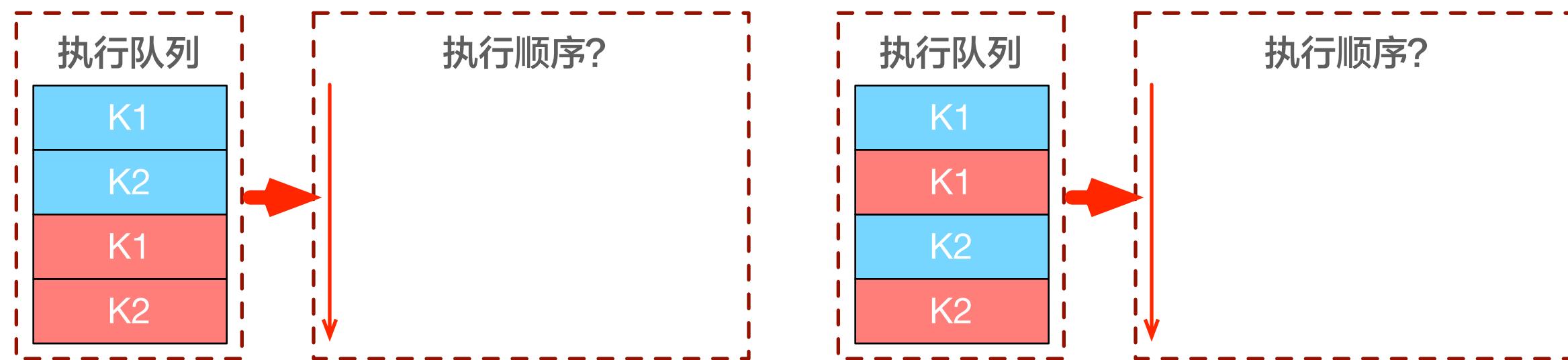
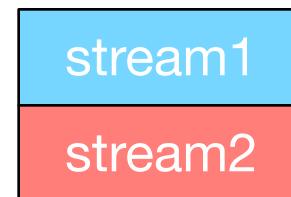
## ● 流调度举例

- 交换stream1与stream2的发布顺序后
  - D2H2可以与H2D1同时执行



## ● 流调度举例

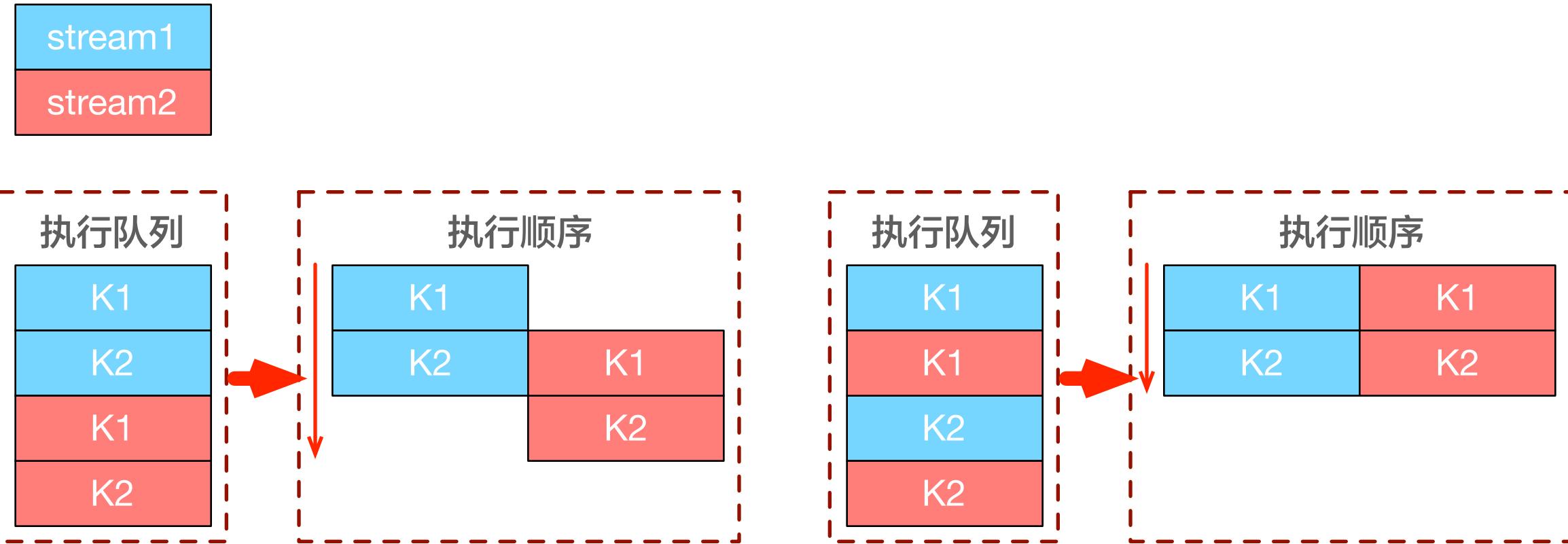
- 同一队列并不意味着一定串行执行
  - Fermi支持16路并发，但只有一个硬件队列
    - 最多可同时执行16个网格（核函数调用）
  - 以下哪个发布顺序效率更高？



## ● 流调度举例

– 同一队列并不意味着一定串行执行

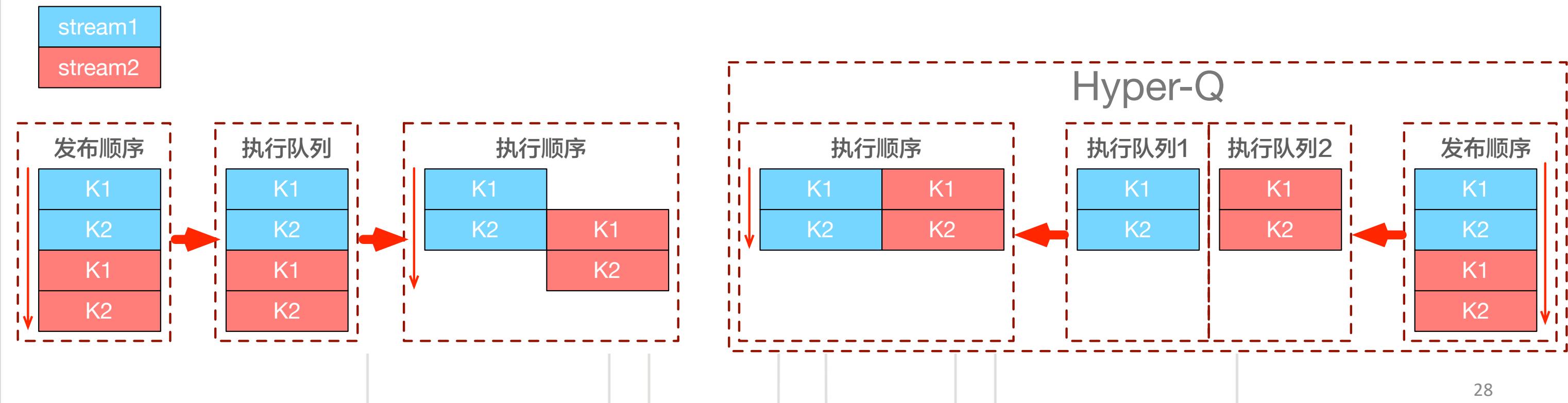
- Fermi支持16路并发，但只有一个硬件队列
  - 最多可同时执行16个网格（核函数调用）
- 以下哪个发布顺序效率更高？



## ● 流调度

### – Hyper-Q技术

- 每个流分配一个工作队列
- Kepler GPU使用32个硬件工作队列
  - 若流超过32个，则多个流将共享一个硬件队列



## ● 流调度

### – 阻塞流与非阻塞流

- 显式创建的流为异步执行，但仍可能被阻塞
  - 使用 `cudaStreamCreate()` 创建的为阻塞流
- 创建非阻塞流
  - 使用 `cudaStreamCreateWithFlags(&stream, flags);`
  - `cudaStreamDefault` 为阻塞流
  - `cudaStreamNonBlocking` 为非阻塞流

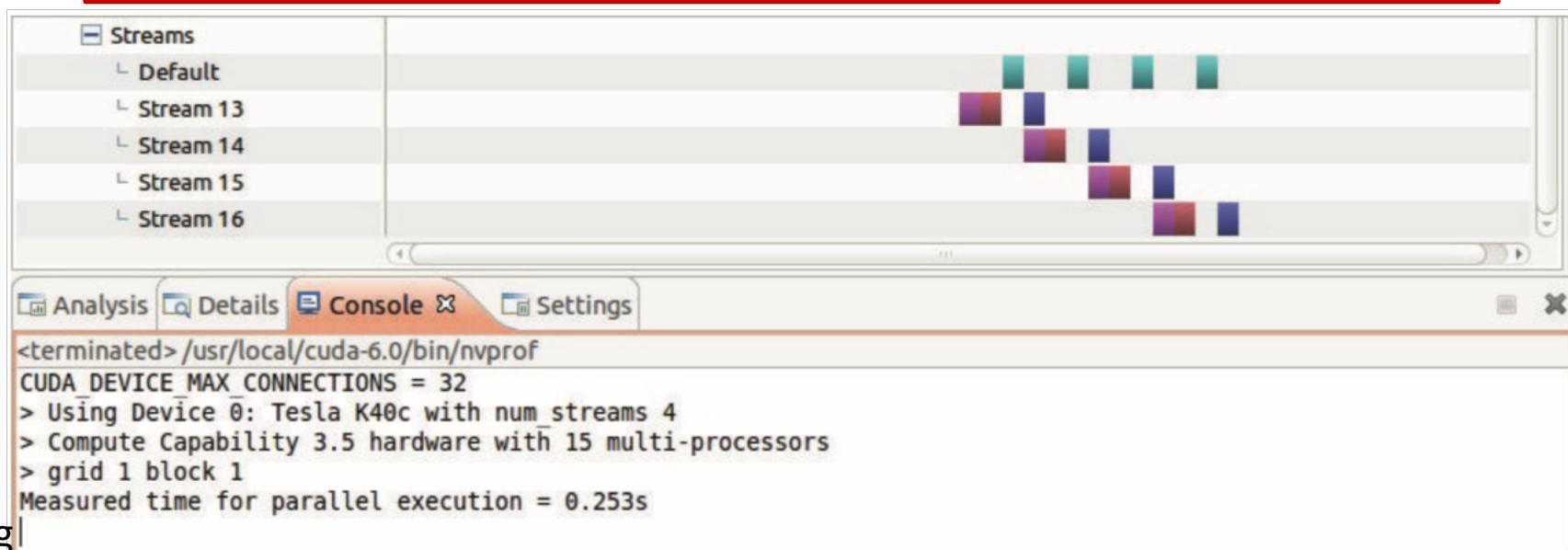


## ● 流调度

### – 阻塞流执行举例

- 空流中的核函数执行 (kernel\_3) 将阻塞其他流中核函数执行

```
// dispatch job with depth first ordering
for (int i = 0; i < n_streams; i++) {
    kernel_1<<<grid, block, 0, streams[i]>>>();
    kernel_2<<<grid, block, 0, streams[i]>>>();
    kernel_3<<<grid, block>>>();
    kernel_4<<<grid, block, 0, streams[i]>>>();
}
```



## ● 流的优先级

- 高优先级流中的网格队列可以优先占有低优先级的工作
  - **cudaStreamCreateWithPriority**(&stream, flags, priority)
  - priority数值越小，优先级越高
  - 如果priority超出定义的优先级数值范围，将被自动限制为最高/低值
  - 只对计算内核产生影响（不影响数据传输操作）
- 查询优先级范围
  - **cudaDeviceGetStreamPriorityRange**(&leastPriority, &greatestPriority)
  - leastPriority返回最低优先级（数字大）
  - greatestPriority返回最高优先级（数字小）

- 流与并发概述
- 同步与异步
- 流与并发
- 流同步
- 多GPU编程



## ● 隐式同步

- 许多与内存相关的操作都会隐式同步主机和设备函数
  - cudaMemcpy、锁页主机内存分配、设备内存分配，等

## ● 显式同步

- **cudaDeviceSynchronize()**
  - 阻塞主机执行，直到所有异步执行的设备操作都已经完成
- **cudaStreamSynchronize(stream)**
  - 阻塞主机执行，直到stream内所有操作都已经完成
- 使用CUDA事件（event）同步机制



## ● CUDA事件 (event)

- 标记操作流中特定点
  - 同步流的执行
  - 监控设备的进展
- 监控常用语句
  - **cudaEventCreate(&event)**
    - 在空流中创建事件
  - **cudaEventDestroy(event)**
    - 销毁事件
  - **cudaEventElapsedTime(&time, event1, event2);**
    - 获取两次事件之间的间隔时间

```
cudaEvent_t start, stop;  
cudaEventCreate(&start);  
cudaEventCreate(&stop);  
  
cudaEventRecord(start);  
kernel <<< grid, block >>>();  
cudaEventRecord(stop);  
  
cudaEventSynchronize(stop);  
float milliseconds = 0;  
cudaEventElapsedTime(&milliseconds, start, stop);  
cudaEventDestroy(start);  
cudaEventDestroy(stop);
```

## ● 事件同步机制

- **cudaEventRecord**(event, stream)
  - 在非空流中插入event
- **cudaEventSynchronize**(event)
  - 在event发生前，阻塞主机执行
  - 需要在event插入stream后调用
- **cudaStreamWaitEvent**(event)
  - 在event发生前，阻塞stream执行
  - 用于两个流之间同步
- **cudaEventQuery**(event, stream)
  - 查询event是否已经发生

```
cudaMemcpyAsync(d_in, in, size, H2D, stream1);
cudaEventRecord(event, stream1); // record event

cudaStreamWaitEvent(stream2, event); // wait for event in stream1
kernel << <BLOCKS, TPB, 0, stream2 >> > (d_in, d_out);
```

## ● 流回调 (callback)

– 在CUDA流中调用主机函数的方式

- 流中回调前的所有操作完成时，由CUDA运行时调用流回调指定的主机端函数
- 同步流中的操作与只有主机端能执行的操作
  - 硬盘或网络读写
  - 系统调用

– **cudaStreamAddCallback**(stream, callback, user\_data, flag)

- flag目前没有意义，必须设置为0
- 回调函数中不可以调用CUDA的API函数
- 回调函数中不可以执行同步

## ● 流回调 (callback)

- 在CUDA流中调用主机函数的方式
- 流回调举例

```
for (int i = 0; i < n_streams; i++){  
    stream_ids[i] = i;  
    kernel_1<<<grid, block, 0, streams[i]>>>();  
    kernel_2<<<grid, block, 0, streams[i]>>>();  
    kernel_3<<<grid, block, 0, streams[i]>>>();  
    kernel_4<<<grid, block, 0, streams[i]>>>();  
    cudaStreamAddCallback(streams[i], my_callback, (void *)(stream_ids + i), 0);  
}
```



- 流与并发概述
- 同步与异步
- 流与并发
- 流同步
- 多GPU编程



## 为什么要使用多GPU编程？

- 单GPU无法处理的超大数据集问题
- 通过多GPU系统提升吞吐量和效率
  - 例如，将主机开销分散到多GPU上（价格开销也是如此）
- 需要解决的问题
  - GPU间通信与同步（大体分两种情况）
    - 单主机多GPU
    - 多主机通过网络互连



## ● 多GPU编程模型

		单主机	多主机
单进程	单线程	■	■
	多线程	■	■
多进程		■	■



GPU通过P2P或共享主机内存通信



GPU通过主机间的消息传递通信



## ● 单线程多GPU

- CUDA默认使用系统中的第一个设备 (`device_id=0`)
  - 不一定是最适合计算的设备
- 可使用**`cudaSetDevice(device_id)`**切换设备
  - **`cudaGetDeviceProperties(&device_prop, device_id)`**查询设备参数
- 举例：

```
cudaSetDevice(0);  
kernel<<<...>>>( ... ); //executed on device-0  
  
cudaSetDevice(1);  
kernel<<<...>>>( ... ); //executed on device-1
```

## ● 单线程多GPU

### – 使用流与事件

- 流与事件属于创建时正在使用的设备 (current device)
- 可通过事件同步不同设备上的流

```
cudaSetDevice(0);
cudaStreamCreate(&streamA); // streamA and eventA belong to device-0
cudaEventCreate(&eventA);
```

```
cudaSetDevice(1);
cudaStreamCreate(&streamB); // streamB and eventB belong to device-1
cudaEventCreate(&eventB);
kernel << <..., streamB >> >(...);
cudaEventRecord(eventB, streamB);
```

```
cudaSetDevice(0);
cudaEventSynchronize(eventB);
```

即使切换至device-0，仍然能同步device-1上创建的事件B。

```
kernel << <..., streamA >> >(...);
```

## ● 单线程多GPU

### – 使用流与事件

- 不能将事件插入不同设备创建的流中

```
cudaSetDevice(0);
cudaStreamCreate(&streamA); // streamA and eventA belong to device-0
cudaEventCreate(&eventA);

cudaSetDevice(1);
cudaStreamCreate(&streamB); // streamB and eventB belong to device-1
cudaEventCreate(&eventB);
kernel << <..., streamB >> >(...);
cudaEventRecord(eventA, streamB);

cudaSetDevice(0);
cudaEventSynchronize(eventB);
kernel << <..., streamA >> >(...);
```

eventA 在 device-0 上创建，不能插入到在 device-1 上创建的 streamB 中。

## ● 多线程多GPU

- 使用多线程在多GPU上同时创建任务
  - 例如，使用OpenMP线程将操作调度到不同设备的不同流上
    - 也可以调度到不同设备的默认流上

```
omp_set_num_threads(num_dev);
#pragma omp parallel
{
    int i = omp_get_thread_num();
    cudaSetDevice(i);

    cudaStreamCreate(&streams[i]);
    kernel_1<<<..., streams[i]>>>(....);
    kernel_2<<<..., streams[i]>>>(....);
    kernel_3<<<..., streams[i]>>>(....);
    kernel_4<<<..., streams[i]>>>(....);
}
```

此处是否存在竞争条件？

## ● 多线程多GPU

- 使用多线程在多GPU上同时创建任务
  - 例如，使用OpenMP线程将操作调度到不同设备的不同流上
    - 也可以调度到不同设备的默认流上

```
omp_set_num_threads(num_dev);
#pragma omp parallel
{
    int i = omp_get_thread_num();
    cudaSetDevice(i);

    cudaStreamCreate(&streams[i]);
    kernel_1<<<..., streams[i]>>>(....);
    kernel_2<<<..., streams[i]>>>(....);
    kernel_3<<<..., streams[i]>>>(....);
    kernel_4<<<..., streams[i]>>>(....);
}
```

不存在竞争条件，CUDA运行时API为thread-safe，每个线程单独维护关于current device的状态。

## ● 多线程多GPU

- 然而，由于每个线程有独立的current device状态，也造成某些情况下的安全隐患
  - 以下代码的问题是？

```
cudaSetDevice(1);
cudaMalloc(&a,bytes);

#pragma omp parallel
{
    kernel<<<...>>>(a);
}
```

## ● 多线程多GPU

- 然而，由于每个线程有独立的current device状态，也造成某些情况下的安全隐患
  - 以下代码的问题是？

```
cudaSetDevice(1);  
cudaMalloc(&a, bytes);  
  
#pragma omp parallel  
{  
    kernel<<<...>>>(a);  
}
```

在device-1上分配内存

创建新的主机线程，初始化current device为device-0

device-0无法访问device-1上的内存（无UVA时）

## ● 多线程多GPU

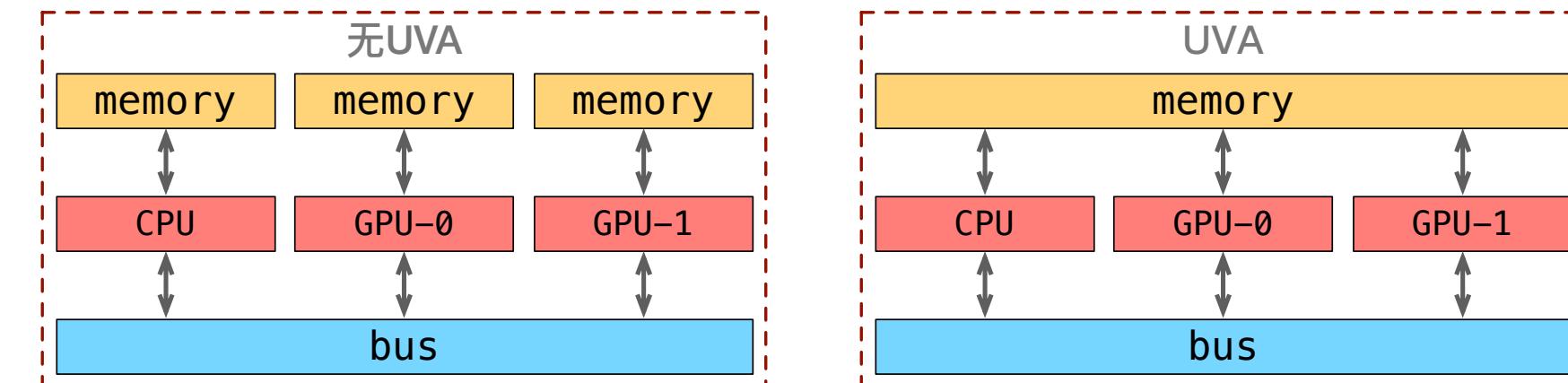
- 然而，由于每个线程有独立的current device状态，也造成某些情况下的安全隐患
  - 修正代码（使用单GPU：device-1）

```
cudaSetDevice(1);
cudaMalloc(&a,bytes);

#pragma omp parallel
{
    cudaSetDevice(1);
    kernel<<<...>>>(a);
}
```

## ● 单主机上的数据传输

- 统一虚拟寻址 (Unified Virtual Address, UVA)
  - 主机内存及多个设备内存共享同一个虚拟地址空间
    - 分配的内存依然在某个单独的设备上 (一个数组不会跨GPU)
    - 在根据地址访问数据时, 驱动/设备会决定地址所处的设备
    - 可访问另一个GPU内存上的地址或主机内存上的地址
  - 过于依赖UVA将对性能产生负面影响
    - 如, 跨PCIe总线的许多小规模传输
  - 需要CUDA 4.0及计算能力 2.0以上的64位架构



## ● 单主机上的数据传输

### – 使用Peer-to-Peer (P2P) 内存拷贝

- 数据将通过最短的PCIe路径进行传输
- 无需通过CPU内存
- 需要启用peer-access

– **cudaDeviceEnablePeerAccess**(*peer\_device*, 0)

» 允许当前设备通过P2P访问*peer\_device* GPU的内存

– **cudaDeviceCanAccessPeer**(&*accessible*, *dev\_X*, *dev\_Y*)

» 检查两个设备之间是否可以通过P2P访问

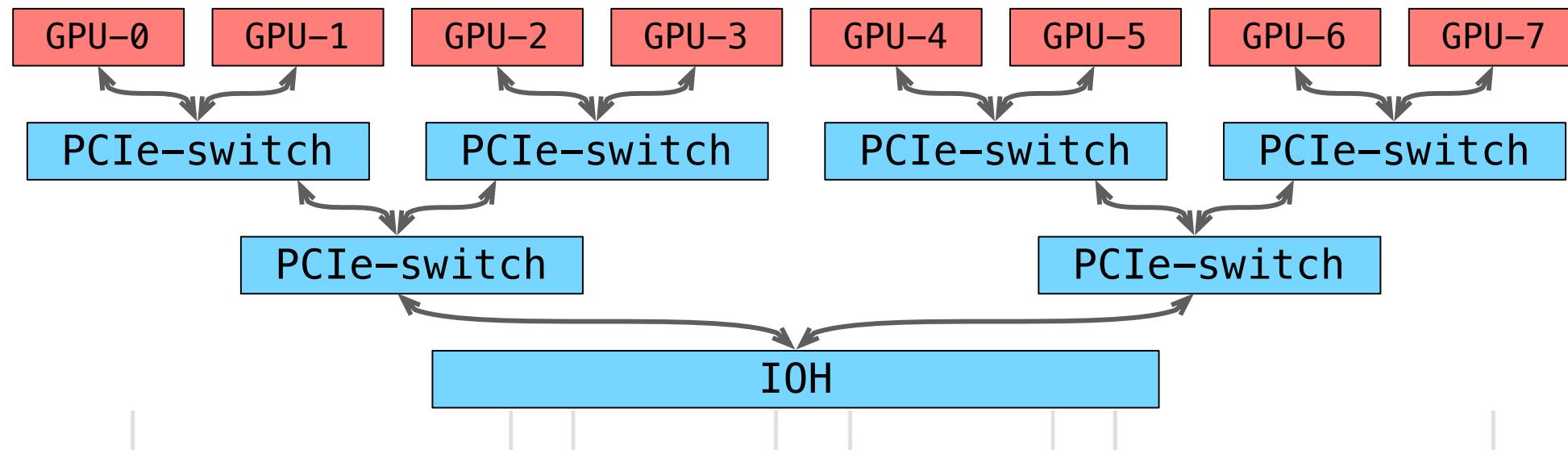
» 其中一个GPU为Fermi前产品则无法访问

» 两个GPU不处于同一个IOH芯片上时也无法访问

## ● 单主机上的数据传输

### – 使用Peer-to-Peer (P2P) 内存拷贝

- 通过**cudaMemcpyPeerAsync**(dst\_addr, dst\_dev, src\_addr, src\_dev, num\_bytes, stream) 进行拷贝
  - 拷贝为异步进行，需要指明其所处的CUDA流
  - 也可使用与其对应的blocking版本**cudaMemcpyPeer**(...)
- 大幅提高吞吐量（尤其是当最短路径不需要经过IOH芯片时）



## ● 多主机多GPU编程

- 通过MPI (message passing interface) 跨主机在进程间通信
  - 传统MPI
    - MPI函数只能用于传输主机内存
    - 在传递前必须在CPU与GPU内存间进行拷贝
  - CUDA-aware MPI
    - 可以将GPU内存中的内容直接传递到MPI函数上，不需要通过主机内存中转数据
    - 开源的CUDA-aware MPI (参考Cheng et al., Professional CUDA C Programming)
      - » MVAPICH2 2.0rc 2
      - » MVAPICH2-GDR 2.0b
      - » OpenMPI 1.7
      - » CRAY MPI (MPT 5.6.2)
      - » IBM Platform MPI (8.3)

## ● MPI+CUDA

### – MPI程序基本框架

```
#include <mpi.h>

int main(int argc, char *argv[]) {
    int rank, size;
    /* Initialize the MPI library */
    MPI_Init(&argc, &argv);

    /* Determine the calling process rank and total number of ranks */
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size); MPI_COMM_WORLD指明通信域为所有进程

    /* Call MPI routines like MPI_Send, MPI_Recv, ... */
    ...

    /* Shutdown MPI library */
    MPI_Finalize();
    return 0;
}
```

## ● MPI+CUDA

– MPI通过消息（message）交换数据

- 常见的MPI数据交换API

- 点到点: **MPI\_Send**(...), **MPI\_Recv**(...), **MPI\_Sendrecv**(...)

- 集合交换: **MPI\_Reduce**(...)

- **MPI\_Recv**(buf, counter, datatype, dest, tag, comm , &status)

- buf: 发送缓冲区地址

- counter: 发送数据个数

- datatype: 发送数据类型

- dest: 目的地进程号

- tag: 消息标志

- comm: MPI进程组所在的通信域

- status: MPI\_Status结构指针, 返回状态信息

## ● MPI+CUDA

- MPI通过消息（message）交换数据
  - 使用**MPI\_Send**(...)与**MPI\_Recv**(...)交换数据
    - blocking通信机制

代码问题？

```
dest = (rank==0)?1:0;  
  
MPI_Send(sbuf, size, MPI_CHAR, dest, 100, MPI_COMM_WORLD);  
MPI_Recv(rbuf, size, MPI_CHAR, dest, 100, MPI_COMM_WORLD, &reqstat);
```



## ● MPI+CUDA

- MPI通过消息（message）交换数据
  - 使用**MPI\_Send**(...)与**MPI\_Recv**(...)交换数据
    - blocking通信机制

```
dest = (rank==0)?1:0;  
  
MPI_Send(sbuf, size, MPI_CHAR, dest, 100, MPI_COMM_WORLD);  
MPI_Recv(rbuf, size, MPI_CHAR, dest, 100, MPI_COMM_WORLD, &reqstat);
```

```
if (rank==0){  
    MPI_Send(sbuf, size, MPI_CHAR, 1, 100, MPI_COMM_WORLD);  
    MPI_Recv(rbuf, size, MPI_CHAR, 1, 100, MPI_COMM_WORLD, &reqstat);  
} else if (rank==1){  
    MPI_Recv(rbuf, size, MPI_CHAR, 0, 100, MPI_COMM_WORLD, &reqstat);  
    MPI_Send(sbuf, size, MPI_CHAR, 0, 100, MPI_COMM_WORLD);  
}
```

更正

## ● MPI+CUDA

- MPI通过消息（message）交换数据
  - 使用non-blocking API交换数据
    - **MPI\_Irecv**(...)与**MPI\_Isend**(...)
    - I: immediate
    - 使用CUDA-aware MPI实现时，可以直接把设备内存指针传给MPI函数
    - 使用传统MPI时，需要先将设备内存拷贝到相应的主机内存中

```
MPI_Irecv(r_buf, size, MPI_CHAR, other_proc, 100,  
MPI_COMM_WORLD, &recv_request);  
MPI_Isend(s_buf, size, MPI_CHAR, other_proc, 10, MPI_COMM_WORLD, &send_request);  
MPI_Waitall(1, &send_request, &reqstat);  
MPI_Waitall(1, &recv_request, &reqstat);
```

- GPU操作既可以是同步，也可以是异步的
  - 同步操作将会阻塞主机进程
- 流操作可以异步调用核函数及拷贝数据
  - 可以重叠核函数调用及数据拷贝以获得更大的性能提升
  - 也可以同时调用多个核函数以保证GPU满载
  - 在某些情况下，操作插入流中的顺序将影响效率
  - 流也可以用于管理多个GPU上代码的执行以及同步



<https://devblogs.nvidia.com/cuda-pro-tip-always-set-current-device-avoid-multithreading-bugs/>

<https://www.sie.es/wp-content/uploads/2015/12/cuda-streams-best-practices-common-pitfalls.pdf>

<https://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf>

<https://www.nvidia.com/docs/IO/116711/sc11-multi-gpu.pdf>

<https://devblogs.nvidia.com/parallelforall/gpu-pro-tip-cuda-7-streams-simplify-concurrency/>

<https://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/>



# Questions?

