



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

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

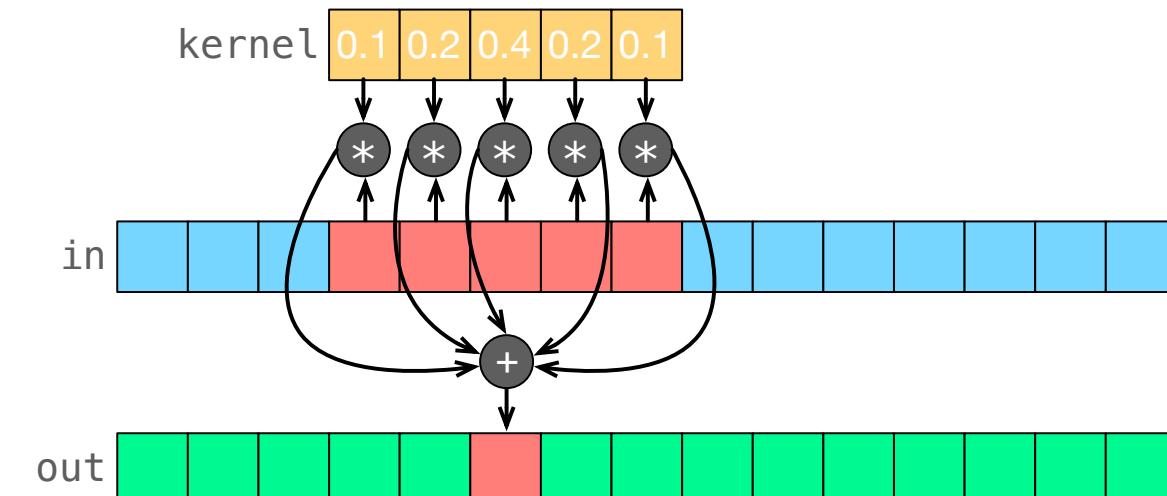


● 图像卷积

- 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%E8%A7%A3%E8%A7%A3.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: MulticoreSYSU@163. com		
– 截止时间		
• 5月27日晚23:59		
• 如需使用slip days, 请于 截止时间前 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



- 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);
```



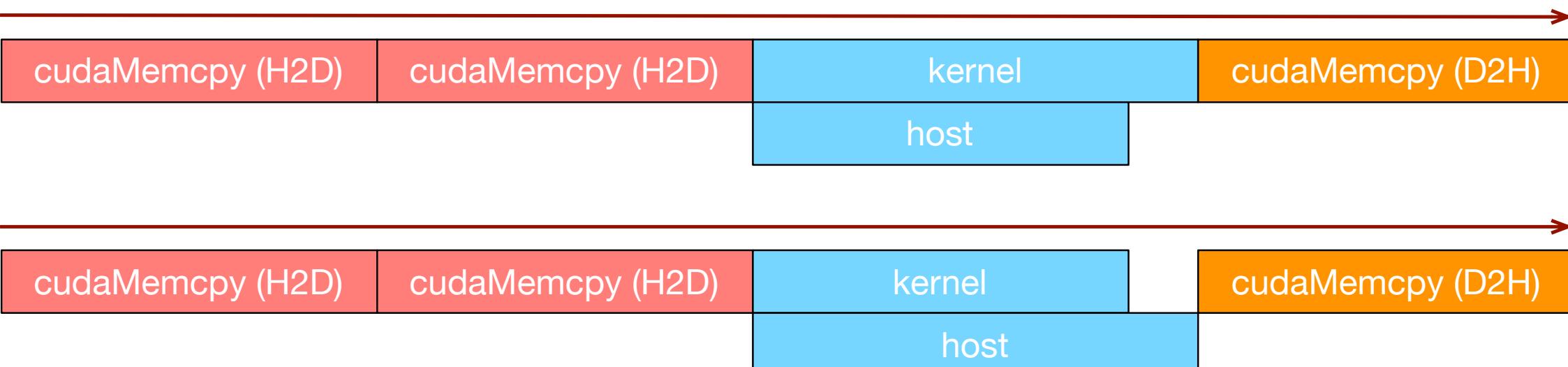
● 举例：下列语句中，哪些调用是异步执行的？

```
//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



- 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>>>();
```

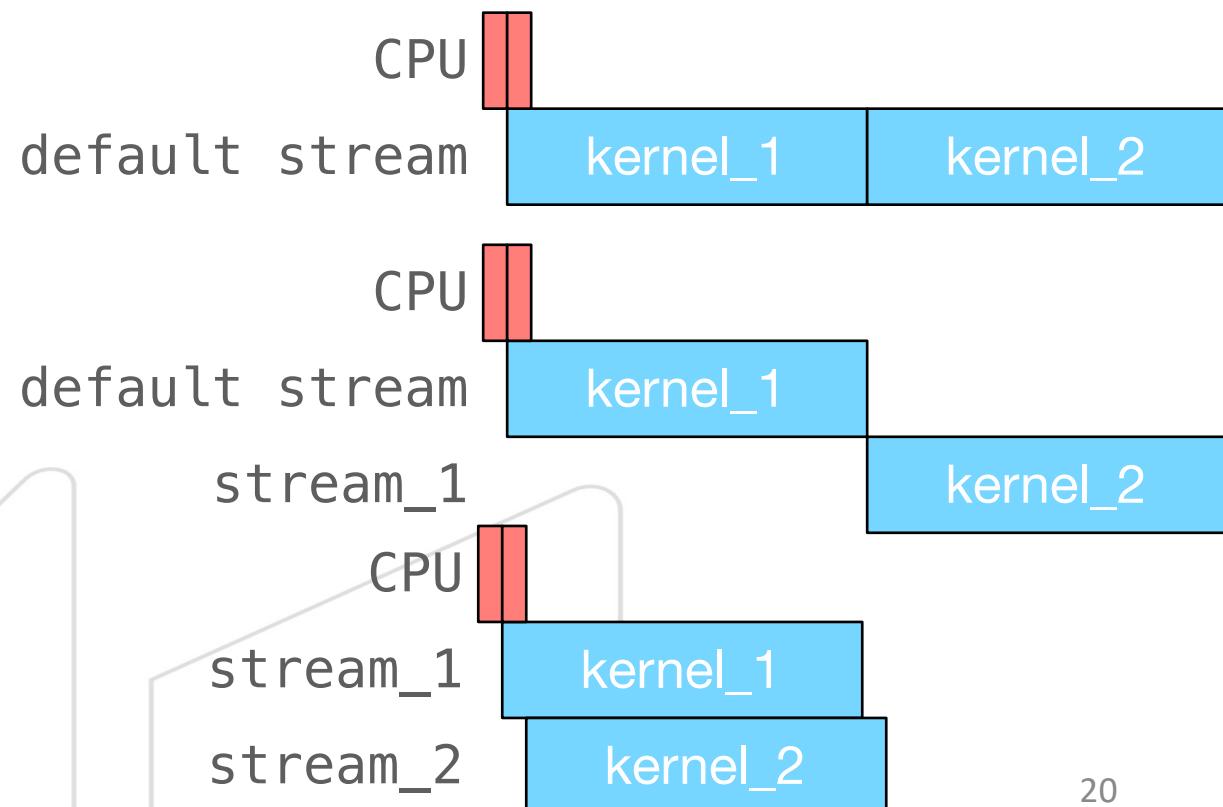
```
kernel_2<<<grid, block, 0>>>();
```

```
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 ...

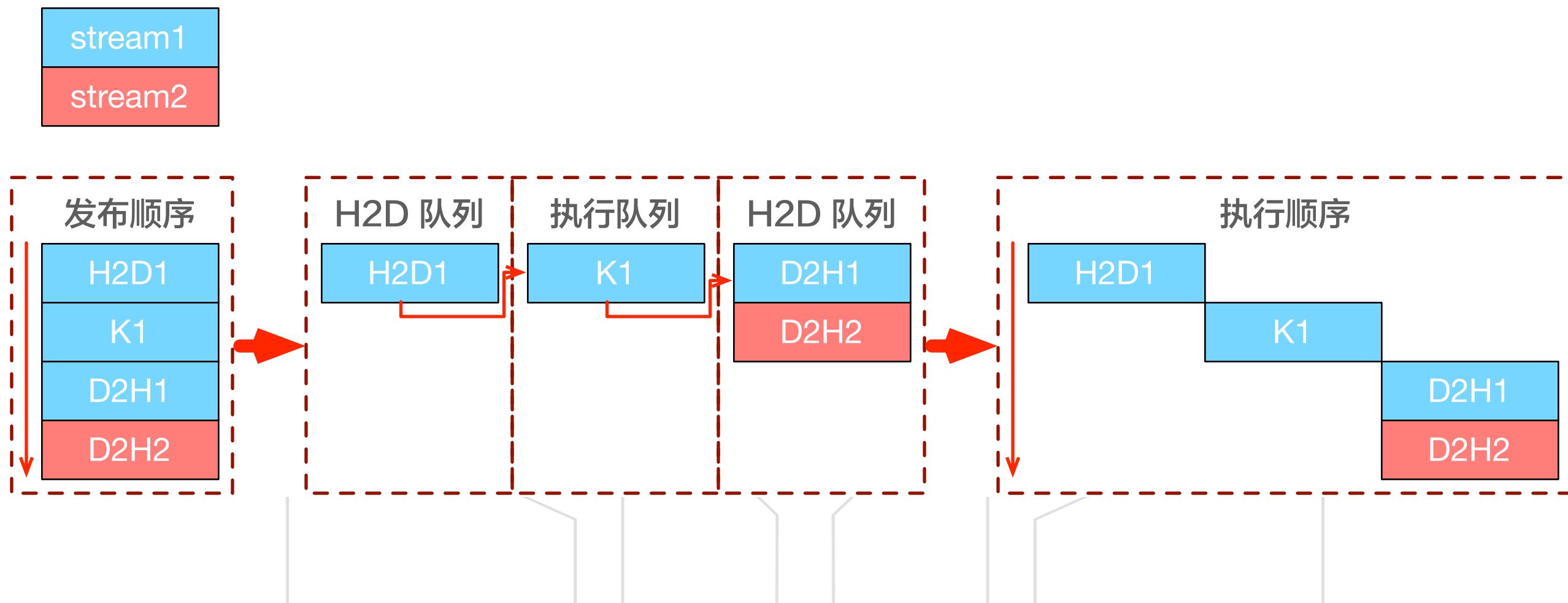
cudaStreamDestroy(stream_1);
```

● 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

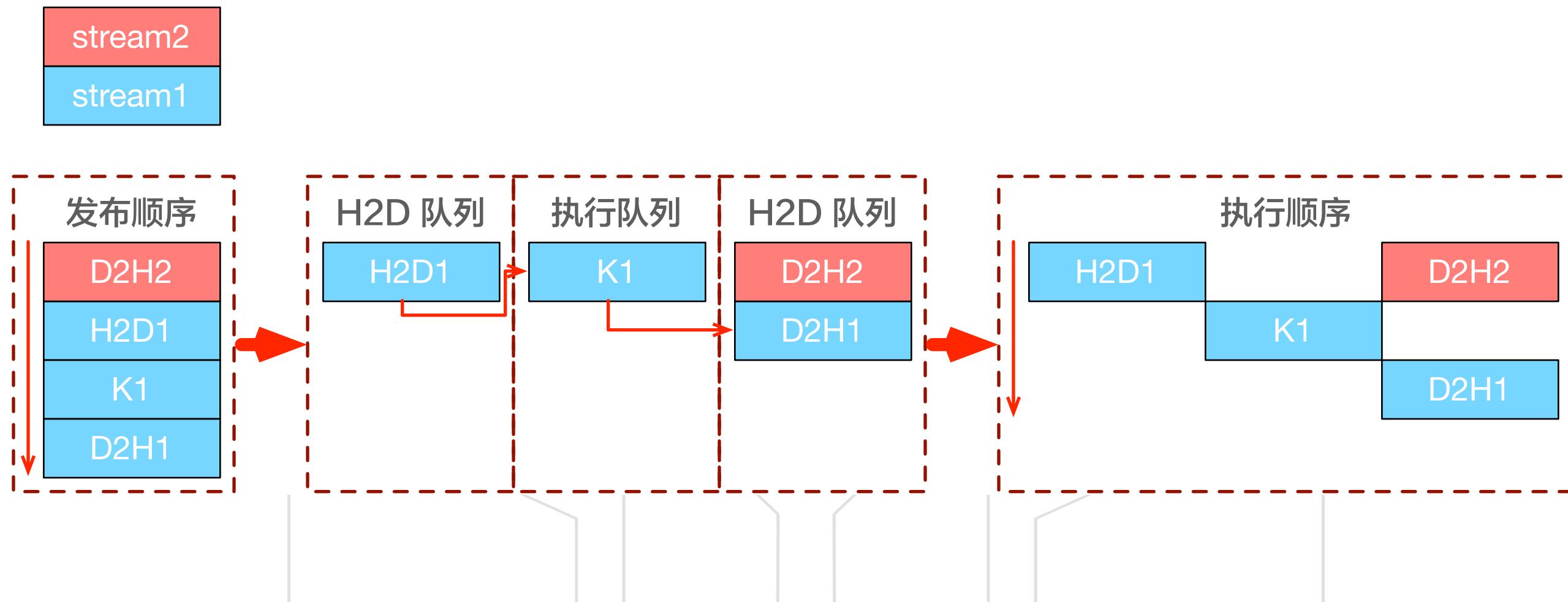
● 流调度举例

- stream2中的操作D2H2没有并发
 - 被D2H1所阻塞



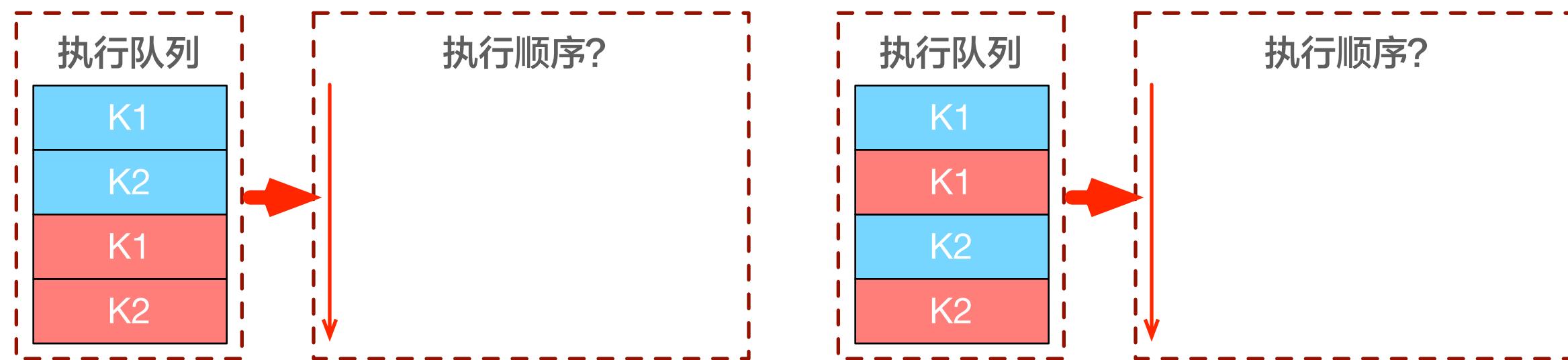
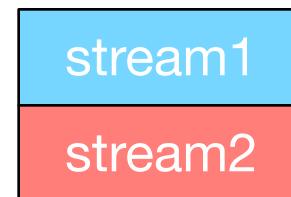
● 流调度举例

- 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

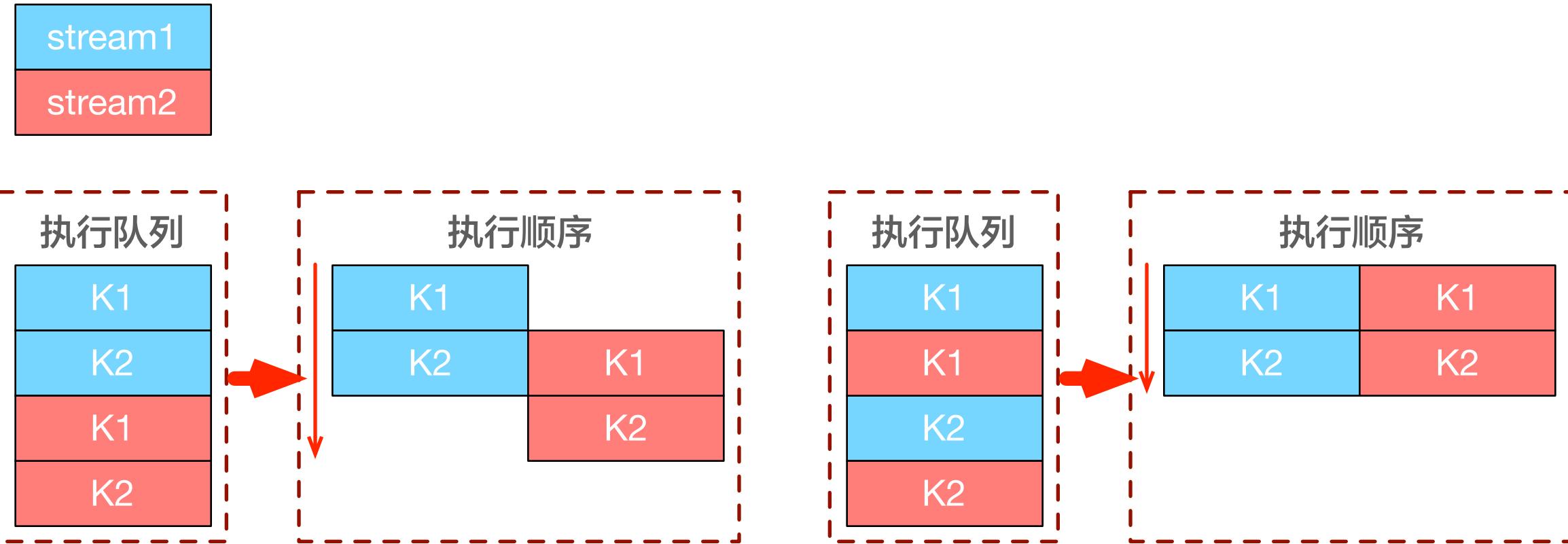
- 同一队列并不意味着一定串行执行
 - Fermi支持16路并发，但只有一个硬件队列
 - 最多可同时执行16个网格（核函数调用）
 - 以下哪个发布顺序效率更高？



● 流调度举例

– 同一队列并不意味着一定串行执行

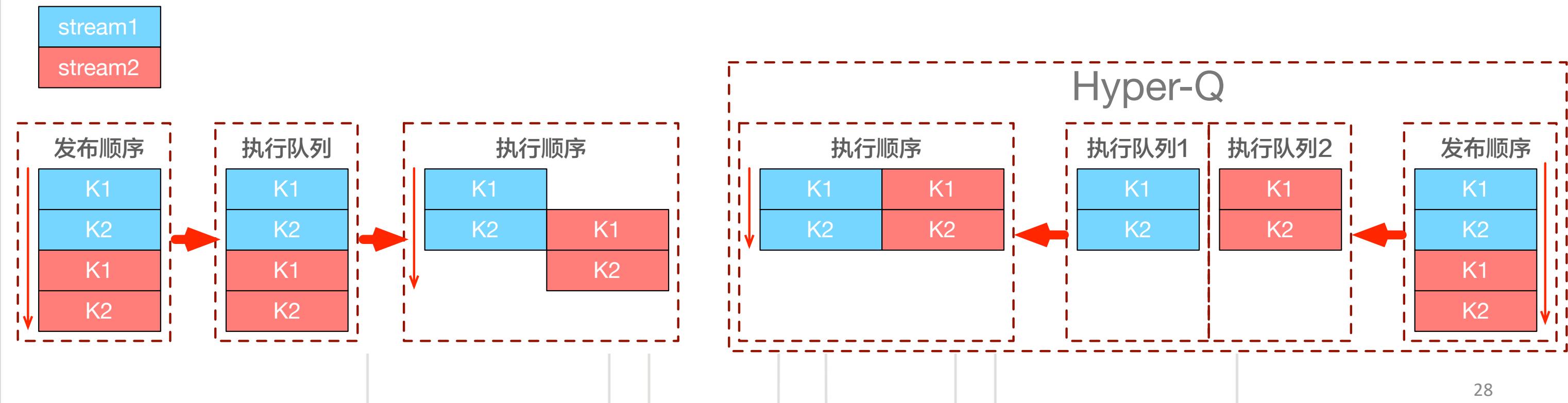
- Fermi支持16路并发，但只有一个硬件队列
 - 最多可同时执行16个网格（核函数调用）
- 以下哪个发布顺序效率更高？



● 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

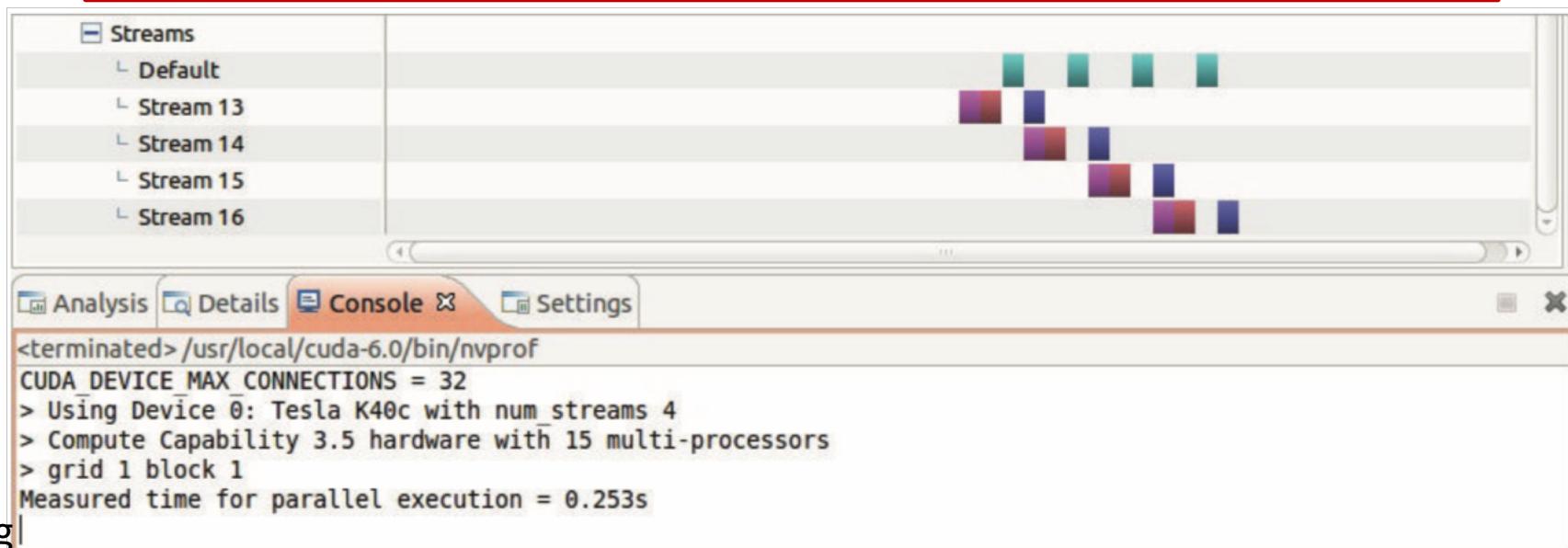


● 流调度

– 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]>>>();
}
```



- 流与并发概述
- 同步与异步
- 流与并发
- 流同步



● 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



● 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);
```

Questions?

