Lecture 20 – CUDA Streams

Based on "CUDA Streams and Concurrency,"
https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf
by Steve Rennich, NVIDIA

Jaejin Lee
Dept. of Computer Science and Engineering, College of Engineering
Dept. of Data Science, Graduate School of Data Science
Seoul National University
http://aces.snu.ac.kr

CHAMP
SNU THUNDER Research Group

CUDA Streams

- Stream
  - A sequence of operations that execute in issue-order on the GPU
- CUDA operations in different streams may run concurrently
- Default stream (a.k.a. stream 0)
  - The stream used when no stream is specified
  - Completely synchronous w.r.t. host and device
    - As if cudaDeviceSynchronize() inserted before and after every CUDA operation
  - Exceptions – asynchronous w.r.t. host
    - Kernel launches in the default stream
    - cudaMemcpy*Async
    - cudaMemset*Async
    - cudaMemcpy within the same device
    - H2D cudaMemcpy of 64kB or less

CUDA Concurrency

- The ability to perform multiple CUDA operations simultaneously
  - CUDA kernel
  - cudaMemcpyAsync (HostToDevice)
  - cudaMemcpyAsync (DeviceToHost)
  - Operations on the CPU
- NVIDIA Fermi architecture can simultaneously support (compute capability 2.0+)
  - Up to 16 CUDA kernels on GPU
  - Two cudaMemcpyAsyncs (must be in different directions)
  - Computation on the CPU

Requirements for Concurrency

- CUDA operations must be in different streams (non-0)
- cudaMemcpyAsync with host from 'pinned' memory
  - Page-locked memory
  - Allocated using cudaMallocHost() or cudaHostAlloc()
- Sufficient resources must be available
  - cudaMemcpyAsyncs in different directions
  - Device resources (shared memory, registers, blocks, etc.)

Pinned Memory

- In contrast to regular pageable host memory, the runtime provides functions to allocate (and free) page-locked (pinned) memory
- Copies between page-locked and device memory can be performed concurrently with kernel execution.
- Bandwidth between page-locked host memory and device may be higher

Synchronous Operations

- All CUDA operations in the default stream are synchronous

```
cudaMalloc ( &dev1, size );
double* host1 = (double*) malloc ( &host1, size ); ...

cudaMemcpy ( dev1, host1, size, H2D );

kernel2 <<< grid, block, 0 >>> ( ..., dev2, ... );
kernel3 <<< grid, block, 0 >>> ( ..., dev3, ... );
cudaMemcpy ( host4, dev4, size, D2H ) ;
```

Completely synchronous!
(default stream)

Asynchronous Operations

- GPU kernels are asynchronous with host by default

```
cudaMalloc ( &dev1, size );
double* host1 = (double*) malloc ( &host1, size ); ...
cudaMemcpy ( dev1, host1, size, H2D );
kernel2 <<< grid, block, 0 >>> ( ..., dev2, ... );
CPU_function_call();
kernel3 <<< grid, block, 0 >>> ( ..., dev3, ... );
cudaMemcpy ( host4, dev4, size, D2H ) ;
```

Potentially overlapped!

Asynchronous with Streams

- Fully asynchronous / concurrent
- Data used by concurrent operations should be independent

```
cudaStream_t stream1, stream2, stream3, stream4;
cudaStreamCreate ( &stream1);
...
cudaMalloc ( &dev1, size );
cudaMallocHost ( &host1, size ); // pinned memory required on host
...
cudaMemcpyAsync ( dev1, host1, size, H2D, stream1 );
kernel2 <<< grid, block, 0, stream2 >>> ( ..., dev2, ... );
kernel3 <<< grid, block, 0, stream3 >>> ( ..., dev3, ... );
cudaMemcpyAsync ( host4, dev4, size, D2H, stream4 );
CPU_function_call();
...
```

Potentially overlapped!

Explicit Synchronization

- Synchronize everything
  - cudaDeviceSynchronize ()
  - Blocks host until all issued CUDA calls are complete
- Synchronize w.r.t. a specific stream
  - cudaStreamSynchronize (streamid)
  - Blocks host until all CUDA calls in streamid are complete
- Synchronize using Events
  - Create specific 'Events,' within streams, to use for synchronization
  - cudaEventRecord (event, streamid)
  - cudaEventSynchronize (event)
  - cudaStreamWaitEvent (stream, event)
  - cudaEventQuery (event)

Explicit Synchronization (cont'd)

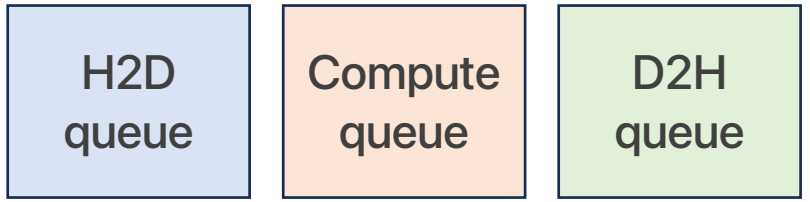
```
{
    cudaEvent_t event;
    cudaEventCreate (&event);    // create event

    cudaMemcpyAsync(d_in, in, size, H2D, stream1);    // 1) H2D copy of new input
    cudaEventRecord(event, stream1);                  // Captures in event the contents of stream
    cudaMemcpyAsync(out, d_out, size, D2H, stream2); // 2) D2H copy of previous result
    cudaStreamWaitEvent(stream2, event);              // Makes all future work submitted to
                                                      // stream2 wait for all work captured in event
    Kernel<<<,,,stream2>>>(d_in, d_out);              // 3) must wait for 1) and 2)
    asynchronousCPUmethod(…) // Async CPU method
}
```

Implicit Synchronization

- The following operations implicitly synchronize all other CUDA operations
  - Page-locked memory allocation
    - cudaMallocHost
    - cudaHostAlloc
  - Device memory allocation
    - cudaMalloc
  - Non-Async version of memory operations
    - cudaMemcpy* (no Async suffix)
    - cudaMemset* (no Async suffix)
  - Change to L1/shared memory configuration cudaDeviceSetCacheConfig

Stream Scheduling



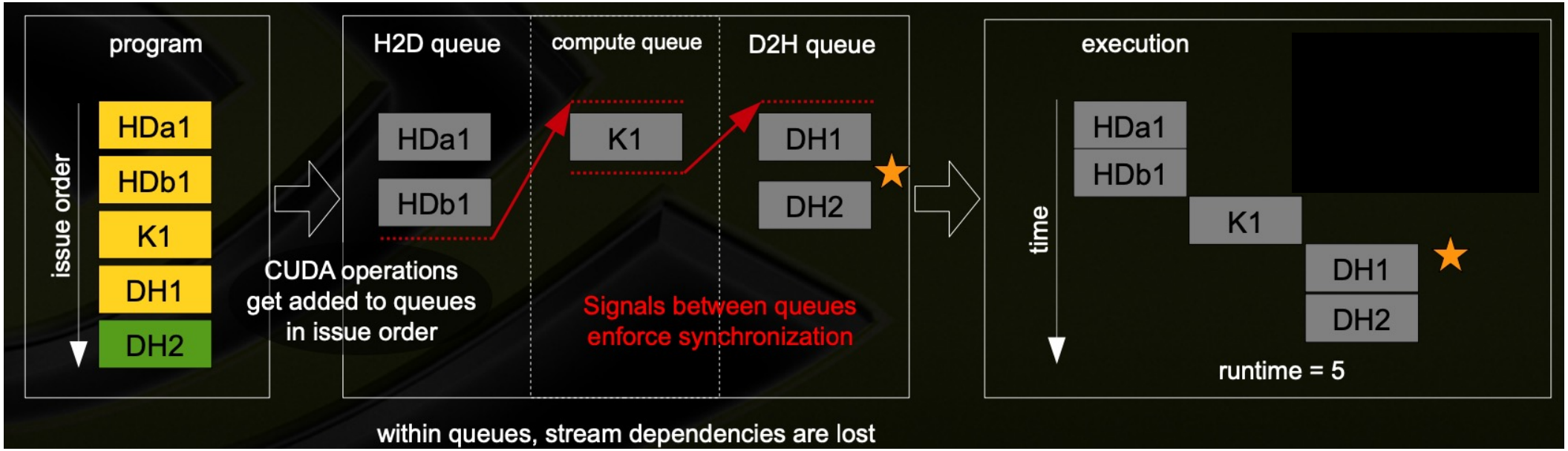
- Fermi hardware has three engine queues
  - 1 Compute Engine queue
  - 2 Copy Engine queues – one for H2D and one for D2H
- CUDA operations are dispatched to HW in the sequence they were issued
  - Placed in the relevant queue
  - Stream dependencies between engine queues are maintained but lost within an engine queue
- A CUDA operation is dispatched from the engine queue if:
  - Preceding calls in the same stream have completed,
  - Preceding calls in the same queue have been dispatched, and
  - Resources are available
- *CUDA kernels may be executed concurrently if they are in different streams*
  - Threadblocks for a given kernel are scheduled if all threadblocks for preceding kernels have been scheduled and there still are SM resources available

Simultaneous Execution of Kernels

- Simultaneous execution of small kernels utilizes the whole GPU
- However, in practice,
  - Concurrent kernel execution on the same device is hard to witness
  - Requires kernels with relatively low resource utilization and long execution time
  - There are hardware limits to the number of concurrent kernels per device
    - The maximum number of kernel launches that a device can execute concurrently is four (CUDA compute capability 2.0+)
  - Less efficient than saturating the device with a single kernel

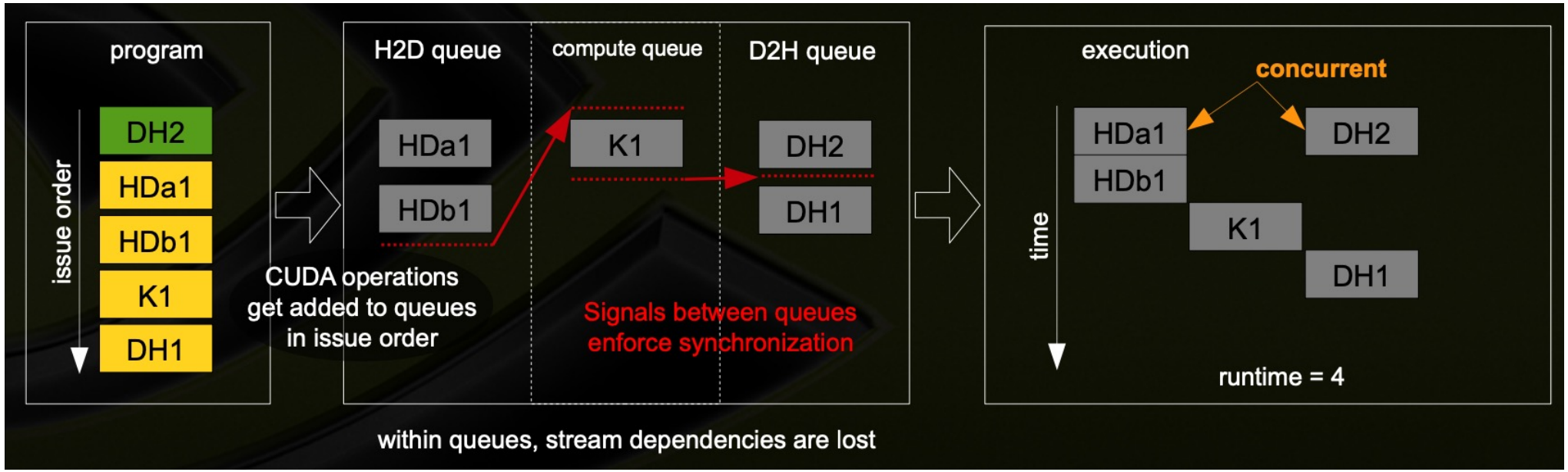
Blocked Queue

- Stream 1: HDa1, HDb1, K1, DH1 (issued first)
- Stream 2: DH2 (completely independent of stream 1)



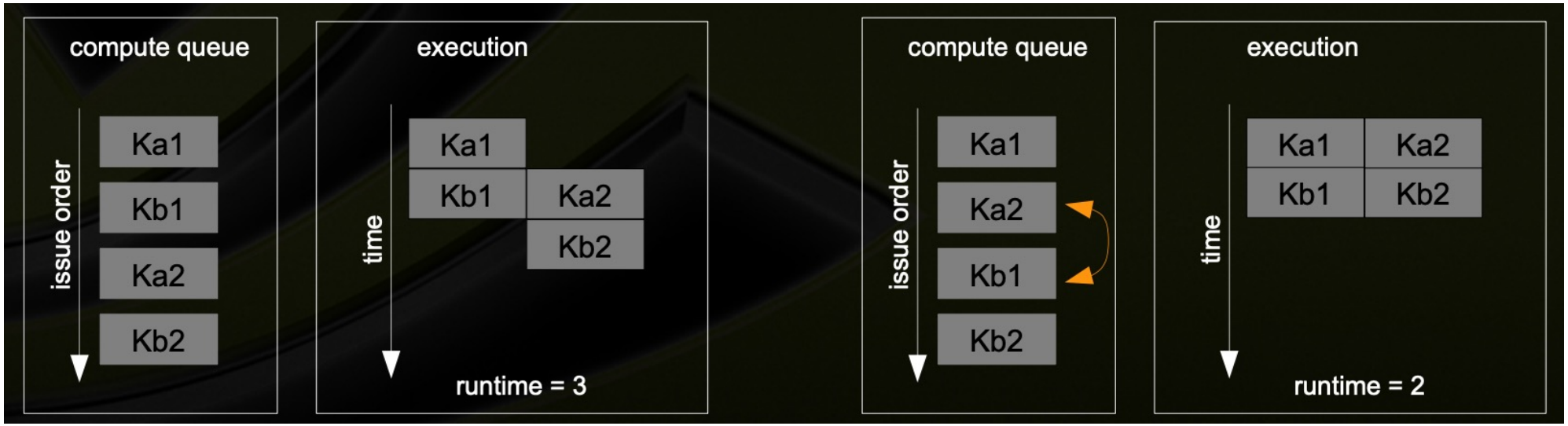
Blocked Queue (cont'd)

- Issue order matters!
- Stream 1: HDa1, HDb1, K1, DH1
- Stream 2: DH2 (issued first)



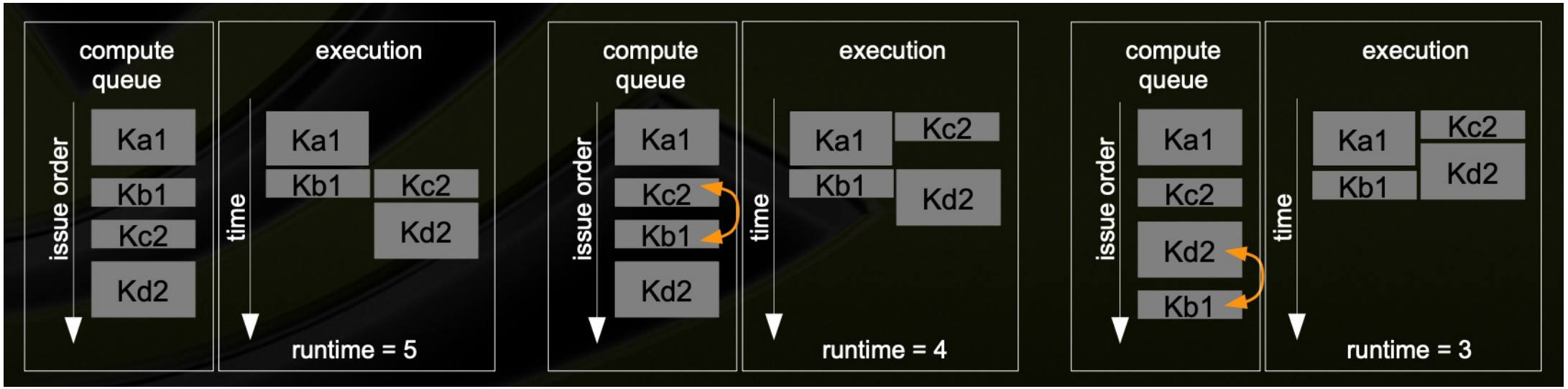
Blocked Kernel

- Issue order matters!
- Kernels are similar in size, fill 1/2 of the SM resources
- Stream 1: Ka1, Kb1
- Stream 2: Ka2, Kb2



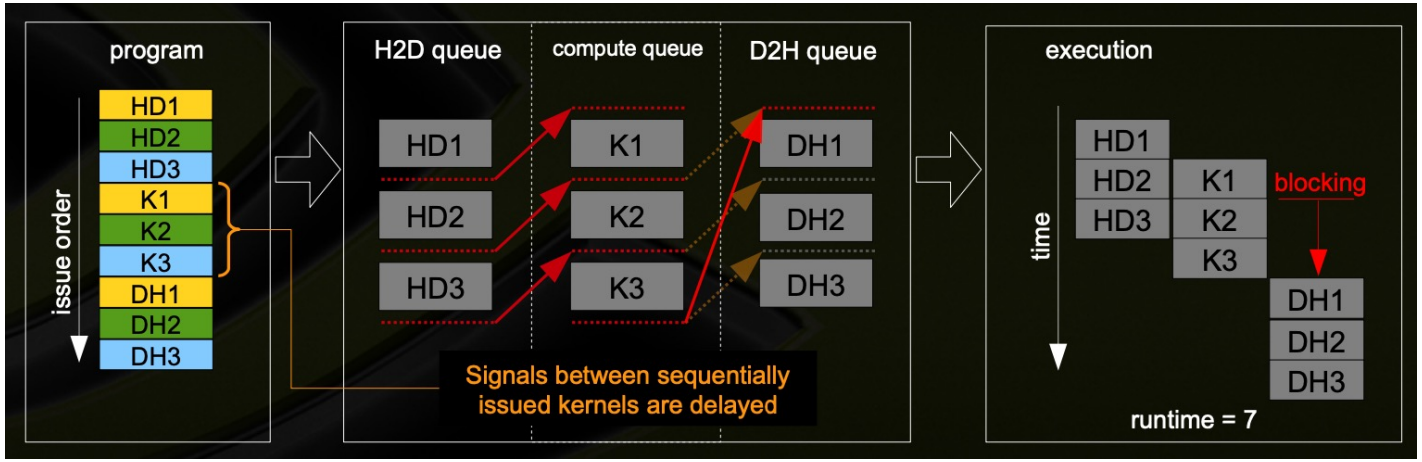
Optimal Concurrency Depends on Kernel Execution Time

- Kernels are different 'sizes'
- Issue order and execution time matters!
- Stream 1: Ka1 {size 2}, Kb1 {size 1}
- Stream 2: Kc2 {size 1}, Kd2 {size 2}



Concurrent Kernels and Blocking

- Normally, a signal is inserted into the queues, after the operation, to launch the next operation in the same stream
- For the compute engine queue, to enable concurrent kernels, when compute kernels are issued sequentially, this signal is delayed until after the last sequential compute kernel
- In some situations, this delay of signals can block other queues
- Example
  - Three streams, each performing (HD, K, DH)
  - Sequentially issued kernels delay signals and block cudaMemcpy(D2H)



Concurrent Kernels and Blocking (cont'd)

- Three streams, each performing (HD, K, DH)