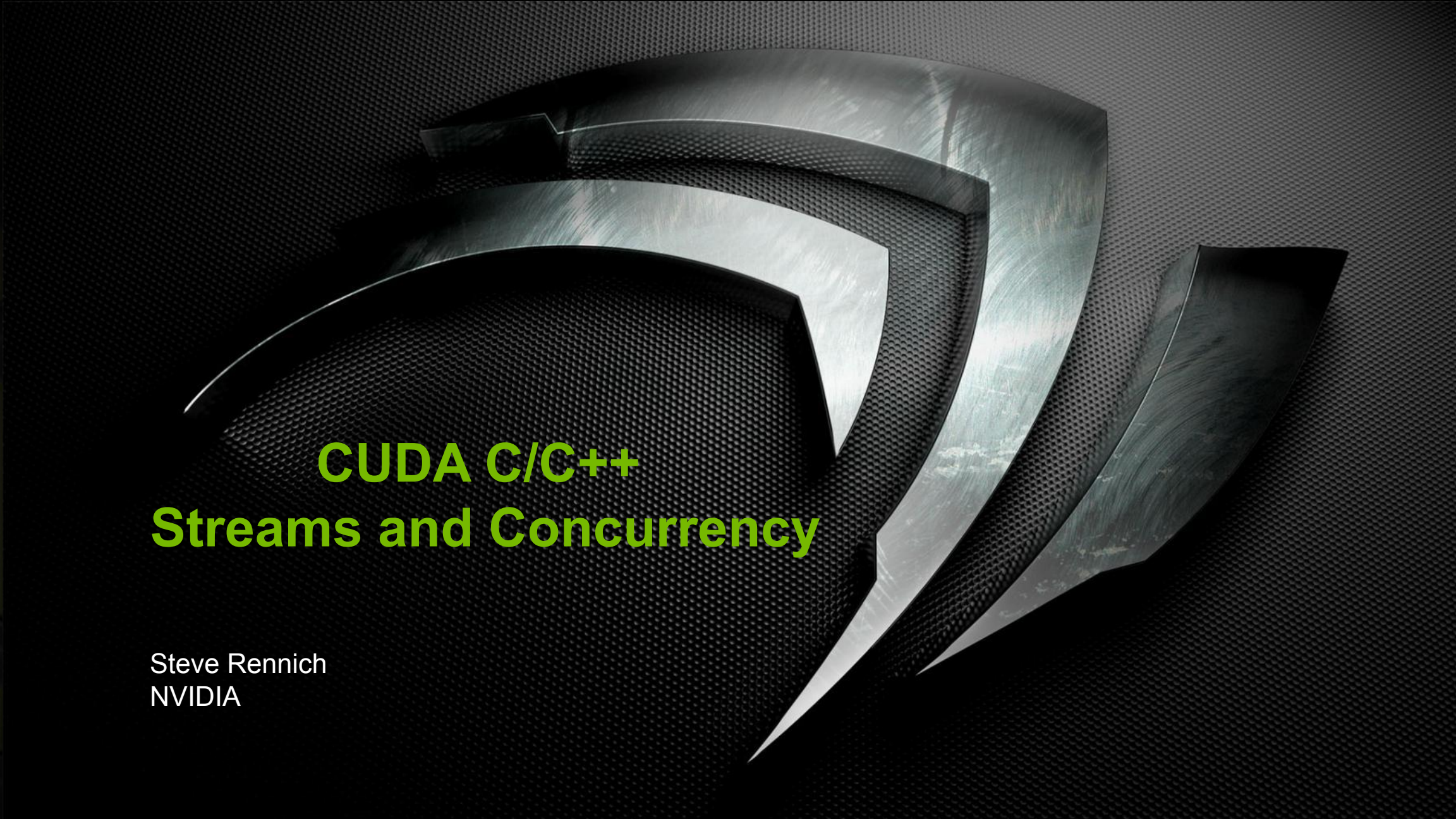


Steve Rennich
NVIDIA

Concurrency

- **The ability to perform multiple CUDA operations simultaneously**
  - (beyond multi-threaded parallelism)
  - CUDA Kernel <<<>>>
  - cudaMemcpyAsync (HostToDevice)
  - cudaMemcpyAsync (DeviceToHost)
  - Operations on the CPU

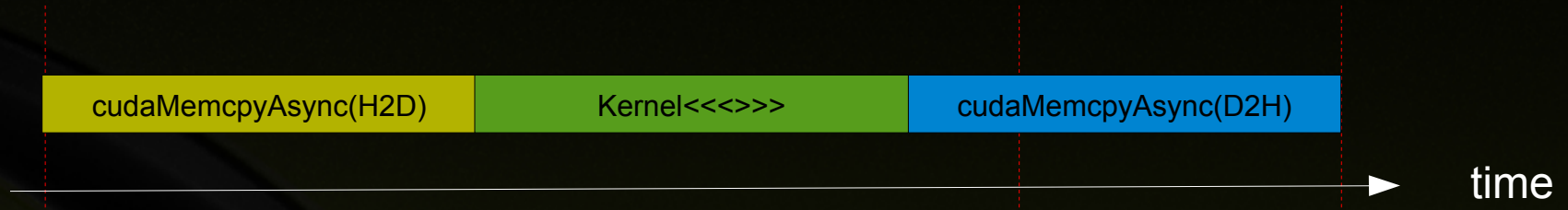
- **Fermi architecture can simultaneously support**
  - (compute capability 2.0+)
  - Up to 16 CUDA kernels on GPU
  - 2 cudaMemcpyAsyncs (must be in different directions)
  - Computation on the CPU

Streams

- Stream
  - A sequence of operations that execute in issue-order on the GPU
- Programming model used to effect concurrency
  - CUDA operations in different streams may run concurrently
  - CUDA operations from different streams may be interleaved

Concurrency Example

- **Serial**



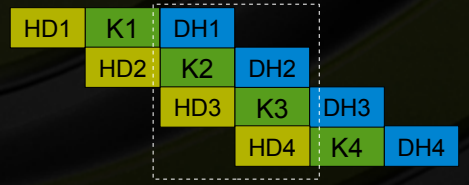
- **Concurrent – overlap kernel and D2H copy**

Amount of Concurrency

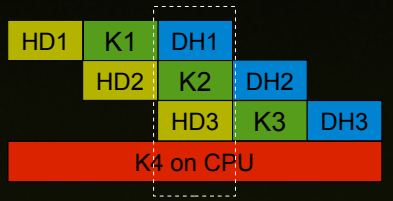
- **Serial** (1x)

- **2-way concurrency** (up to 2x)

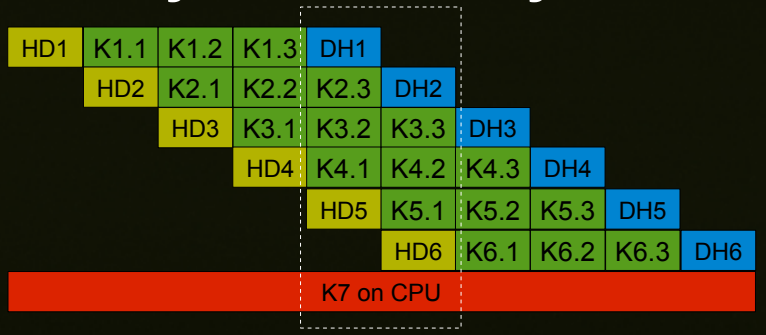
- **3-way concurrency** (up to 3x)



- **4-way concurrency** (3x+)



- **4+ way concurrency**

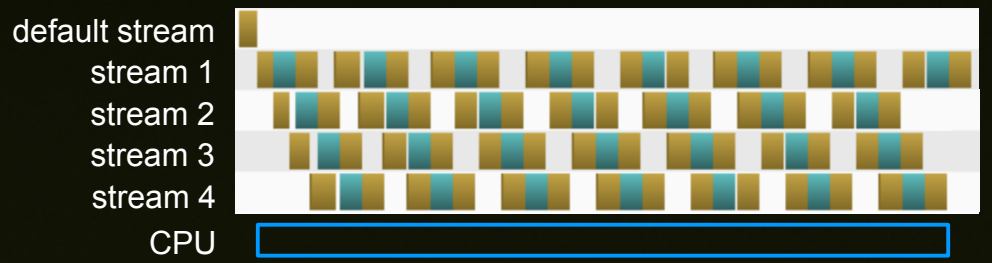


Example – Tiled DGEMM

- **CPU** (4core Westmere x5670 @2.93 GHz, MKL)
  - **43 Gflops**
- **GPU** (C2070)
  - **Serial : 125 Gflops (2.9x)**
  - **2-way : 177 Gflops (4.1x)**
  - **3-way : 262 Gfllops (6.1x)**
- **GPU + CPU**
  - **4-way con.: 282 Gflops (6.6x)**
  - **Up to 330 Gflops for larger rank**
- **Obtain maximum performance by leveraging concurrency**
- **All communication hidden – effectively removes device memory size limitation**

DGEMM: m=n=8192, k=288

Nvidia Visual Profiler (nvvp)



Default Stream (aka Stream '0')

- 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

Requirements for Concurrency

- CUDA operations must be in different, non-0, streams
- 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 (SMEM, registers, blocks, etc.)

Simple Example: 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

- **All CUDA operations in the default stream are synchronous**

Simple Example: Asynchronous, No Streams


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

cudaMemcpy ( dev1, host1, size, H2D ) ;
kernel2 <<< grid, block >>> ( …, dev2, …  ) ;
some_CPU_method ();
kernel3 <<< grid, block >>> ( …, dev3, …  ) ;
cudaMemcpy ( host4, dev4, size, D2H ) ;
...
```

potentially overlapped

- **GPU kernels are asynchronous with host by default**

Simple Example: Asynchronous with Streams

```
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 ) ;
some_CPU_method ();
...
```

potentially overlapped

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

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 Example

- **Resolve using an event**

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

    cudaMemcpyAsync ( d_in, in, size, H2D, stream1 );  // 1) H2D copy of new input
    cudaEventRecord (event, stream1);                  // record event

    cudaMemcpyAsync ( out, d_out, size, D2H, stream2 );// 2) D2H copy of previous result

    cudaStreamWaitEvent ( stream2, event );            // wait for event in stream1
    kernel <<< , , , stream2 >>> ( d_in, d_out );      // 3) must wait for 1 and 2

    asynchronousCPUmethod ( … )                        // Async GPU method
}
```

Implicit Synchronization

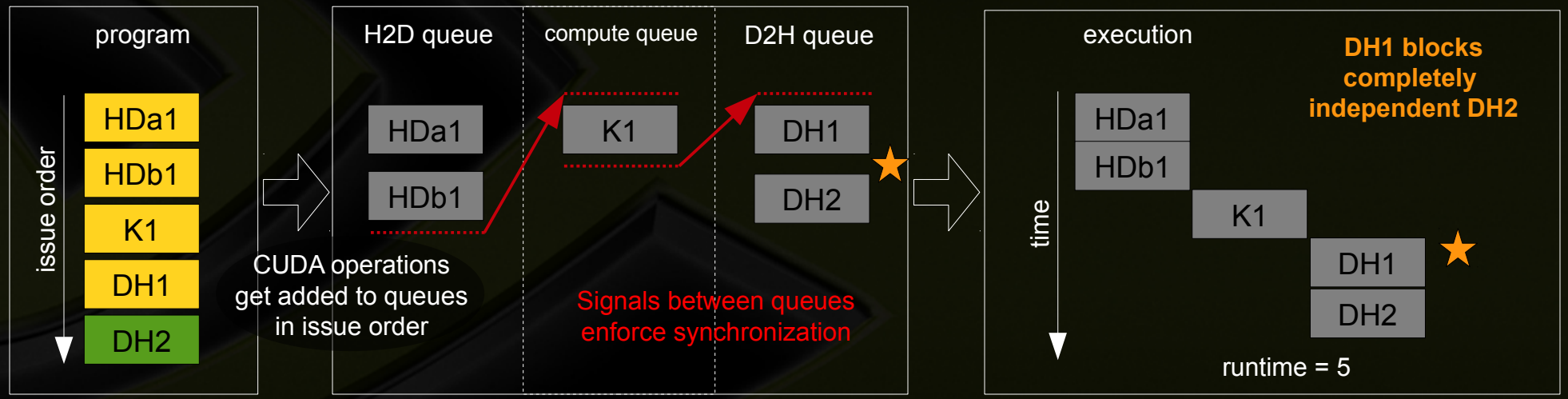
- These operations implicitly synchronize all other CUDA operations
  - Page-locked memory allocation
    - cudaMallocHost
    - cudaHostAlloc
  - Device memory allocation
    - cudaMalloc
  - Non-Async version of memory operations
    - cudaMemcpy* (no Async suffix)
    - cudaMemset* (no Async suffix)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

Stream Scheduling

- Fermi hardware has 3 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
- Note a blocked operation blocks all other operations in the queue, even in other streams

Example – Blocked Queue

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

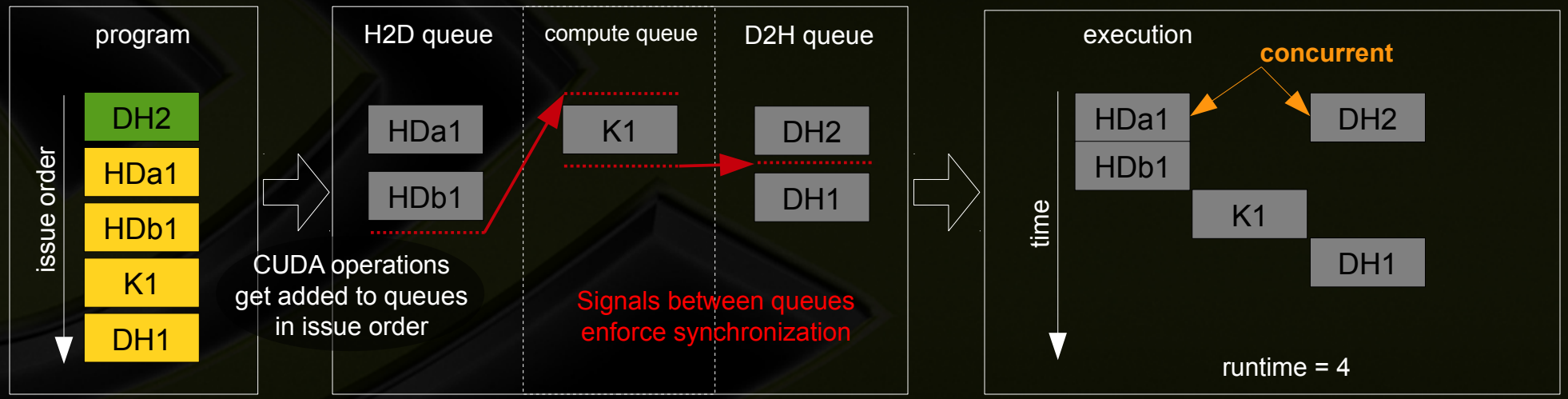


within queues, stream dependencies are lost

Example – Blocked Queue

- Two streams, stream 2 is issued first
  - Stream 1 : HDa1, HDb1, K1, DH1
  - Stream 2 : DH2 (issued first)

issue order matters!



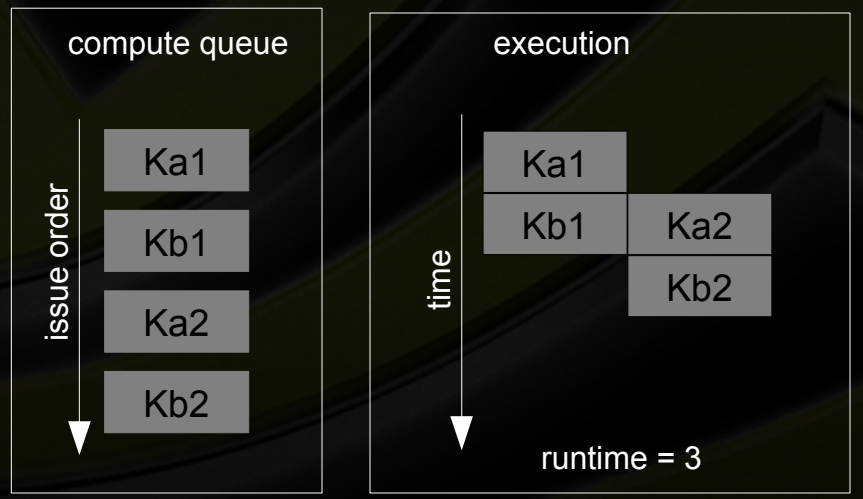
within queues, stream dependencies are lost

Example - Blocked Kernel

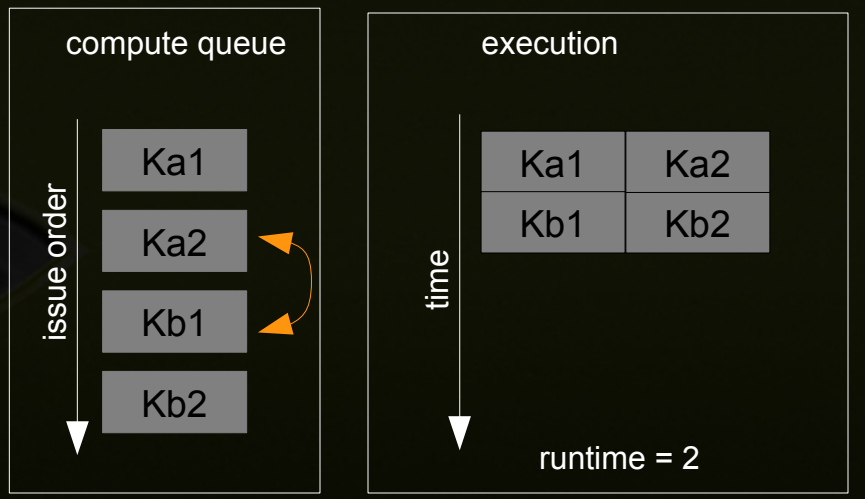
- Two streams – just issuing CUDA kernels
  - Stream 1 : Ka1, Kb1
  - Stream 2 : Ka2, Kb2
  - Kernels are similar size, fill ½ of the SM resources

issue order matters!

- Issue depth first



- Issue breadth first

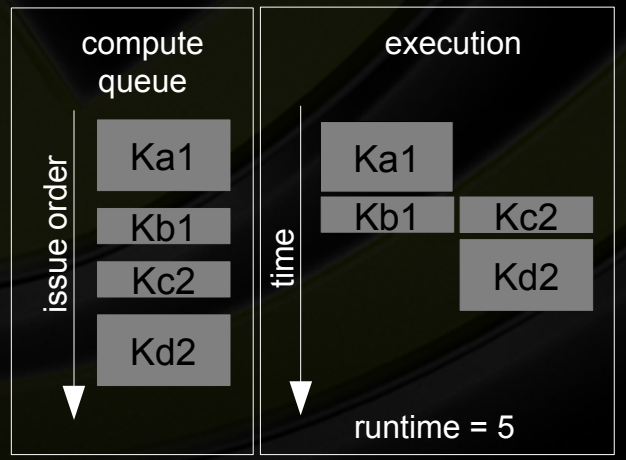


Example - Optimal Concurrency can Depend on Kernel Execution Time

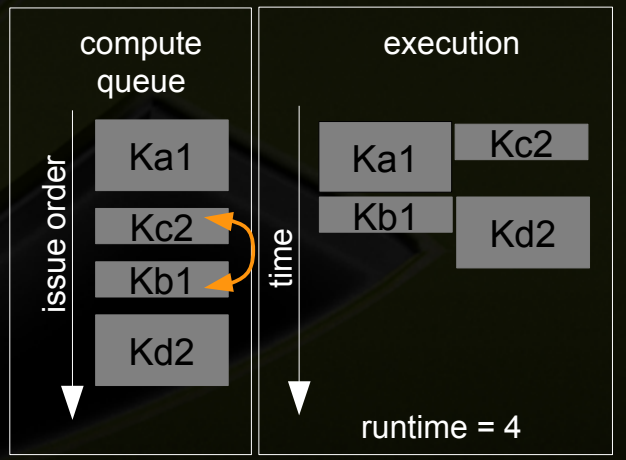
- Two streams – just issuing CUDA kernels – but kernels are different 'sizes'
  - Stream 1 : Ka1 {2}, Kb1 {1}
  - Stream 2 : Kc2 {1}, Kd2 {2}
  - Kernels fill ½ of the SM resources

issue order matters!
execution time matters!

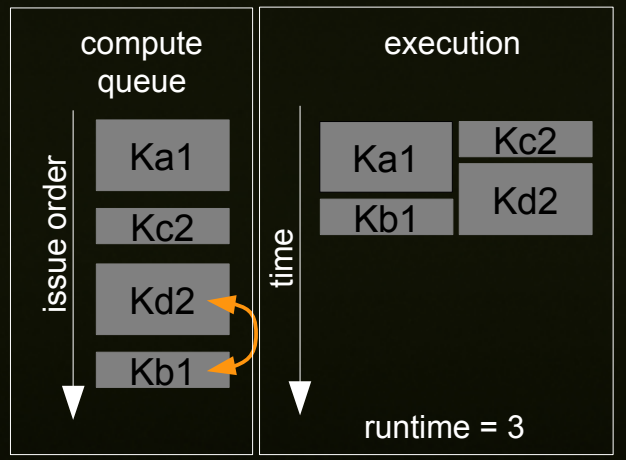
- Depth first



- Breadth first



- Custom

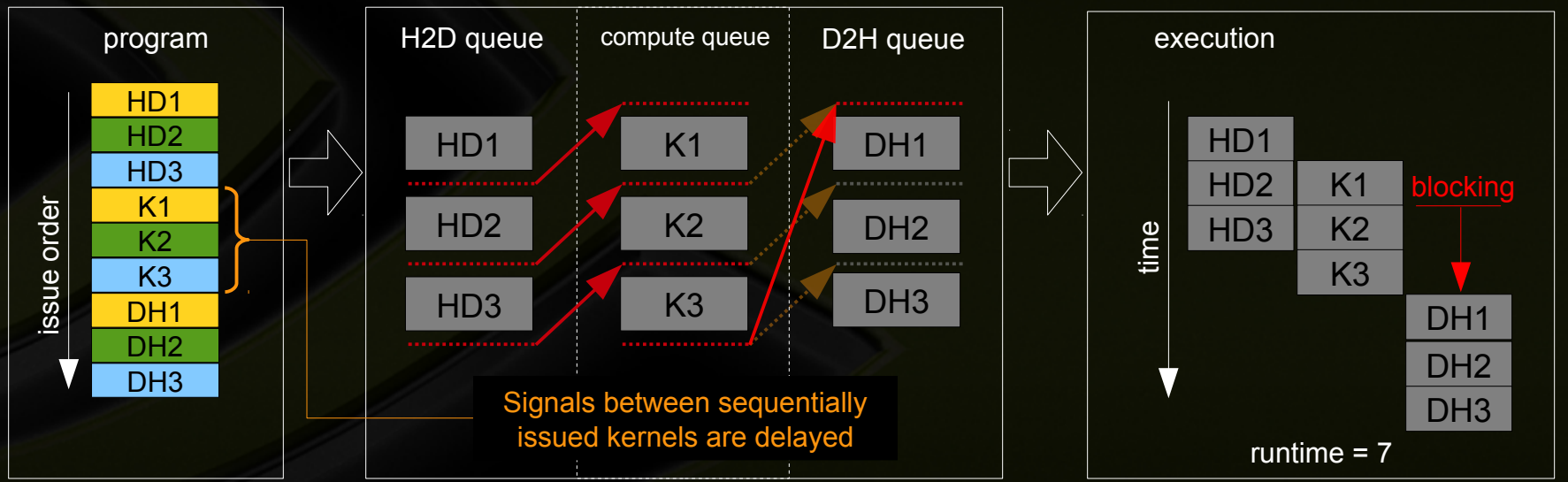


Concurrent Kernel Scheduling

- Concurrent kernel scheduling is special
- 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

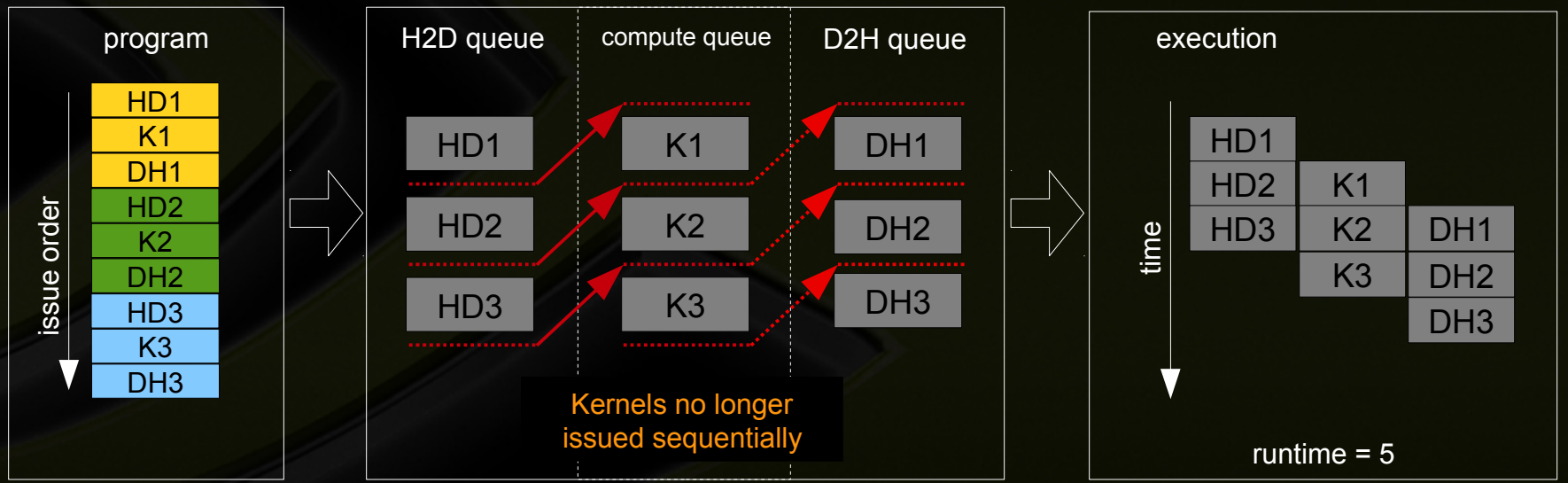
Example – Concurrent Kernels and Blocking

- Three streams, each performing (HD, K, DH)
- Breadth first
  - Sequentially issued kernels delay signals and block cudaMemcpy(D2H)



Example – Concurrent Kernels and Blocking

- Three streams, each performing (HD, K, DH)
- Depth first
  - 'usually' best for Fermi



Previous Architectures

- Compute Capability 1.0+
  - Support for GPU / CPU concurrency
- Compute Capability 1.1+ (i.e. C1060)
  - Adds support for asynchronous memcopies (single engine)
    - (some exceptions – check using asyncEngineCount device property)
- Compute Capability 2.0+ (i.e. C2050)
  - Add support for concurrent GPU kernels
    - (some exceptions – check using concurrentKernels device property)
  - Adds second copy engine to support bidirectional memcopies
    - (some exceptions – check using asyncEngineCount device property)

Additional Details

- It is difficult to get more than 4 kernels to run concurrently
- Concurrency can be disabled with environment variable
  - CUDA_LAUNCH_BLOCKING
- cudaStreamQuery can be used to separate sequential kernels and prevent delaying signals
- Kernels using more than 8 textures cannot run concurrently
- Switching L1/Shared configuration will break concurrency
- To run concurrently, CUDA operations must have no more than 62 intervening CUDA operations
  - That is, in 'issue order' they must not be separated by more than 62 other issues
  - Further operations are serialized
- cudaEvent_t is useful for timing, but for performance use
  - cudaEventCreateWithFlags (&event, cudaEventDisableTiming)

Concurrency Guidelines

- Code to programming model – Streams
  - Future devices will continually improve HW representation of streams model
- Pay attention to issue order
  - Can make a difference
- Pay attention to resources and operations which can break concurrency
  - Anything in the default stream
  - Events & synchronization
  - Stream queries
  - L1/Shared configuration changes
  - 8+ textures
- Use tools (Visual Profiler, Parallel Ensight) to visualize concurrency
  - (but these don't currently show concurrent kernels)


Thank
You

Questions

- 1. Verify that cudaMemcpyAsync() followed by Kernel<<<>>> in Stream-0, the memcpy will block the kernel, but neither will block the host.
- 2. Do the following operations (or similar) contribute to the 64 'out-of-issue-order' limit?
  - CudaStreamQuery
  - cudaWaitEvent
- 3. I understand that the 'query' operations cudaStraemQuery() can be placed in either the engine or the copy queues, that which queue any query actually goes in is difficult (?) to determine, and that this can result in some blocking.