CUDA STREAMS

- A stream is a queue of device work
  - The host places work in the queue and continues on immediately
  - Device schedules work from streams when resources are free
- CUDA operations are placed within a stream
  - e.g. Kernel launches, memory copies
- Operations within the same stream are ordered (FIFO) and cannot overlap
- Operations in different streams are unordered and can overlap

MANAGING STREAMS

- `cudaStream_t stream;`
  - Declares a stream handle
- `cudaStreamCreate(&stream);`
  - Allocates a stream
- `cudaStreamDestroy(stream);`
  - Deallocates a stream
  - Synchronizes host until work in stream has completed

PLACING WORK INTO A STREAM

- Stream is the 4th launch parameter
  - `kernel<<< blocks , threads, smem, stream>>>();`
- Stream is passed into some API calls
  - `cudaMemcpyAsync( dst, src, size, dir, stream);`

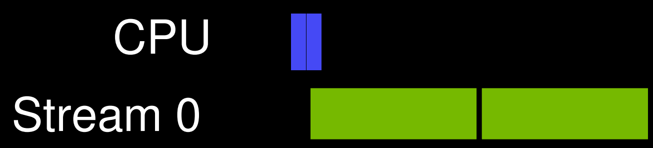
DEFAULT STREAM

- Unless otherwise specified all calls are placed into a default stream
  - Often referred to as "Stream 0"
- Stream 0 has special synchronization rules
  - Synchronous with all streams
    - Operations in stream 0 cannot overlap other streams
- Exception: Streams with non-blocking flag set
  - `cudaStreamCreateWithFlags(&stream,cudaStreamNonBlocking)`
  - Use to get concurrency with libraries out of your control (e.g. MPI)

KERNEL CONCURRENCY

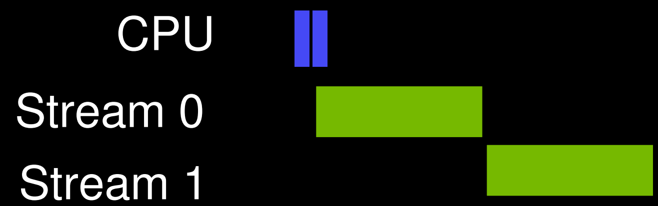
- Assume foo only utilizes 50% of the GPU
- Default stream

```
foo<<<blocks,threads>>>();
foo<<<blocks,threads>>>();
```



- Default & user streams

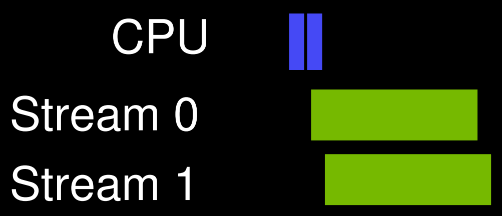
```
cudaStream_t stream1;
cudaStreamCreate(&stream1);
foo<<<blocks,threads>>>();
foo<<<blocks,threads,0,stream1>>>();
cudaStreamDestroy(stream1);
```



KERNEL CONCURRENCY

- Assume foo only utilizes 50% of the GPU
- Default & user streams

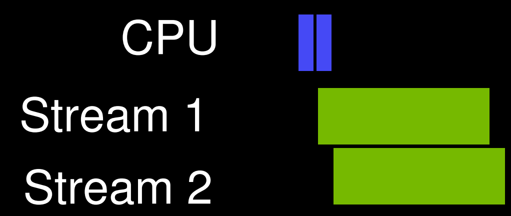
```
cudaStream_t stream1;
cudaStreamCreateWithFlags(&stream1,cudaStreamNonBlocking);
foo<<<blocks,threads>>>();
foo<<<blocks,threads,0,stream1>>>();
cudaStreamDestroy(stream1);
```



KERNEL CONCURRENCY

- Assume foo only utilizes 50% of the GPU
  User streams

```
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
foo<<<blocks,threads,0,stream1>>>();
foo<<<blocks,threads,0,stream2>>>();
cudaStreamDestroy(stream1);
cudaStreamDestroy(stream2);
```



REVIEW

- The host is automatically asynchronous with kernel launches
- Use streams to control asynchronous behavior
  - Ordered within a stream (FIFO)
  - Unordered with other streams
  - Default stream is synchronous with all streams.

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**

CONCURRENT MEMORY COPIES

- First we must review CUDA memory

THREE TYPES OF MEMORY

- Device Memory
  - Allocated using cudaMalloc
  - Cannot be paged
- Pageable Host Memory
  - Default allocation (e.g. malloc, calloc, new, etc)
  - Can be paged in and out by the OS
- Pinned (Page-Locked) Host Memory
  - Allocated using special allocators
  - Cannot be paged out by the OS

ALLOCATING PINNED MEMORY

- `cudaMallocHost(...) / cudaHostAlloc(...)`
  - Allocate/Free pinned memory on the host
  - Replaces malloc/free/new
- `cudaFreeHost(...)`
  - Frees memory allocated by cudaMallocHost or cudaHostAlloc
- `cudaHostRegister(...) / cudaHostUnregister(...)`
  - Pins/Unpins pagable memory (making it pinned memory)
  - Slow so don’t do often
- Why pin memory?
  - Pagable memory is transferred using the host CPU
  - Pinned memory is transferred using the DMA engines
    - Frees the CPU for asynchronous execution
    - Achieves a higher percent of peak bandwidth

CONCURRENT MEMORY COPIES

- `cudaMemcpy(...)`
  - Places transfer into default stream
  - Synchronous: Must complete prior to returning
- `cudaMemcpyAsync(..., &stream)`
  - Places transfer into stream and returns immediately
- To achieve concurrency
  - Transfers must be in a non-default stream
  - Must use async copies
  - 1 transfer per direction at a time
  - Memory on the host must be pinned

PAGED MEMORY EXAMPLE

```
int *h_ptr, *d_ptr;

h_ptr=malloc(bytes);
cudaMalloc(&d_ptr,bytes);

cudaMemcpy(d_ptr,h_ptr,bytes,cudaMemcpyHostToDevice);

free(h_ptr);
cudaFree(d_ptr);
```

PINNED MEMORY: EXAMPLE 1

```
int *h_ptr, *d_ptr;

cudaMallocHost(&h_ptr,bytes);
cudaMalloc(&d_ptr,bytes);

cudaMemcpy(d_ptr,h_ptr,bytes,cudaMemcpyHostToDevice);

cudaFreeHost(h_ptr);
cudaFree(d_ptr);
```

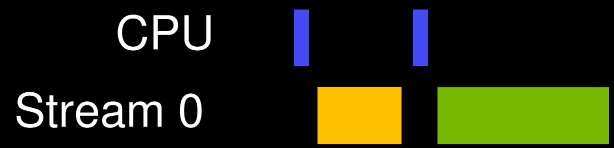
PINNED MEMORY: EXAMPLE 2

```
int *h_ptr, *d_ptr;

h_ptr=malloc(bytes);
cudaHostRegister(h_ptr,bytes,0);
cudaMalloc(&d_ptr,bytes);

cudaMemcpy(d_ptr,h_ptr,bytes,cudaMemcpyHostToDevice);

cudaHostUnregister(h_ptr);
free(h_ptr);
cudaFree(d_ptr);
```

CONCURRENCY EXAMPLES

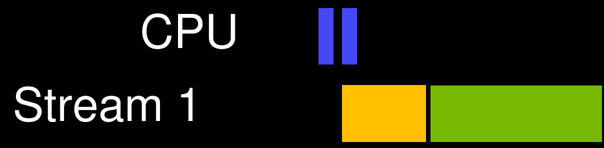
Synchronous

```
cudaMemcpy(...);
foo<<<...>>>();
```



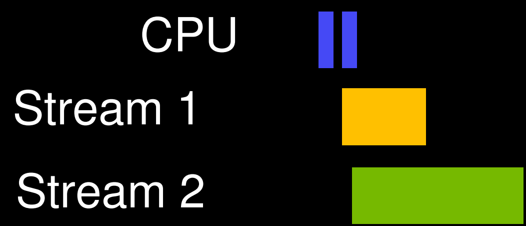
Asynchronous Same Stream

```
cudaMemcpyAsync(...,stream1);
foo<<<...,stream1>>>();
```



Asynchronous Different Streams

```
cudaMemcpyAsync(...,stream1);
foo<<<...,stream2>>>();
```



REVIEW

- Memory copies can execute concurrently if (and only if)
  - The memory copy is in a different non-default stream
  - The copy uses pinned memory on the host
  - The asynchronous API is called
  - There isn't another memory copy occurring in the same direction at the same time.

SYNCHRONIZATION APIS

- Synchronize everything
  - cudaDeviceSynchronize()
    - Blocks host until all issued CUDA calls are complete
- Synchronize host w.r.t. a specific stream
  - cudaStreamSynchronize (stream)
    - Blocks host until all issued CUDA calls in stream are complete
- Synchronize host or devices using events

More Synchronization

Less Synchronization

CUDA EVENTS

- Provide a mechanism to signal when operations have occurred in a stream
  - Useful for profiling and synchronization
- Events have a boolean state:
  - Occurred
  - Not Occurred
  - Important: Default state = occurred

MANAGING EVENTS

- `cudaEventCreate(&event)`
  - Creates an event
- `cudaEventDestroy(&event)`
  - Destroys an event
- `cudaEventCreateWithFlags(&ev, cudaEventDisableTiming)`
  - Disables timing to increase performance and avoid synchronization issues
- `cudaEventRecord(&event, stream)`
  - Set the event state to not occurred
  - Enqueue the event into a stream
  - Event state is set to occurred when it reaches the front of the stream

SYNCHRONIZATION USING EVENTS

- Synchronize using events
  - cudaEventQuery (event)
    - Returns CUDA_SUCCESS if an event has occurred
  - cudaEventSynchronize (event)
    - Blocks host until stream completes all outstanding calls
  - cudaStreamWaitEvent (stream, event)
    - Blocks stream until event occurs
    - Only blocks launches after this call
    - Does not block the host!
- Common multi-threading mistake:
  - Calling cudaEventSynchronize before cudaEventRecord

CUDA_LAUNCH_BLOCKING

- Environment variable which forces sychronization
  - export CUDA_LAUNCH_BLOCKING=1
  - All CUDA operations are synchronous w.r.t the host
- Useful for debugging race conditions
  - If it runs successfully with CUDA_LAUNCH_BLOCKING set but doesn’t without you have a race condition.

Explicit Synchronization Example

- **Resolve using an event**

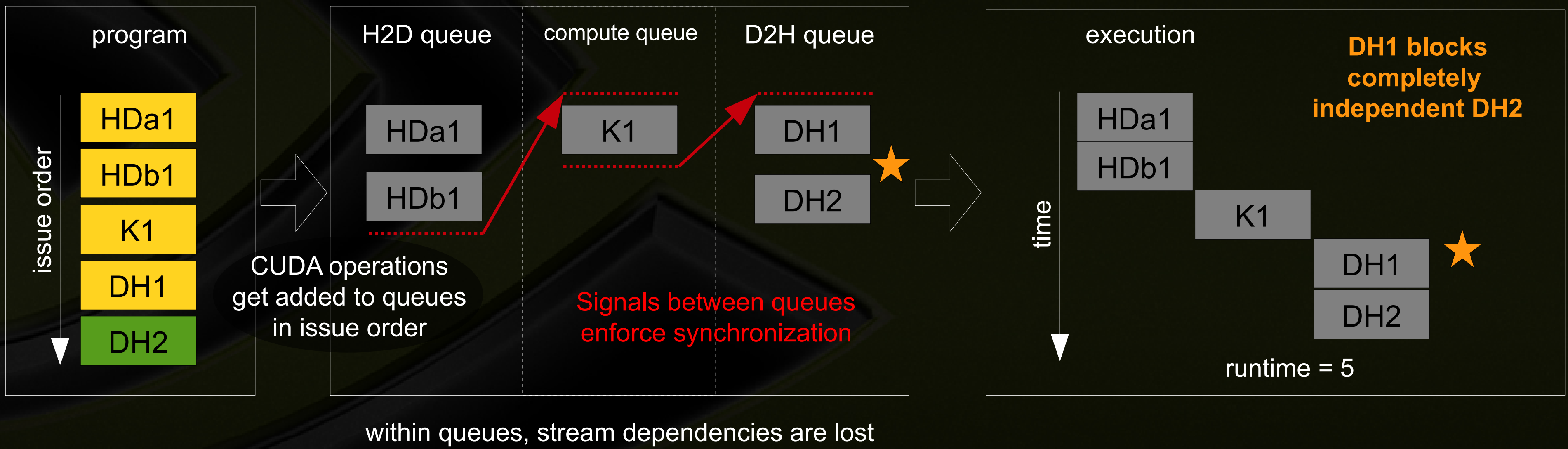
```
{
    cudaEvent_t event;
    cudaEventCreate (&event);                                  // create event

    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
}
```

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

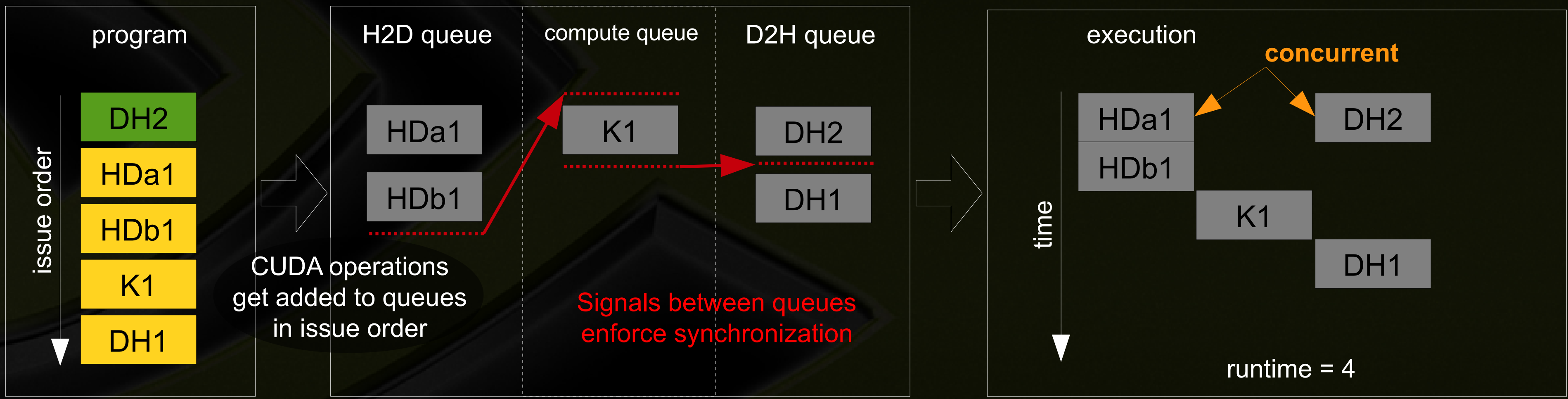
- **Two streams, stream 1 is issued first**
  - **Stream 1 : HDa1, HDb1, K1, DH1 (issued first)**
  - **Stream 2 : DH2 (completely independent of stream 1)**



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

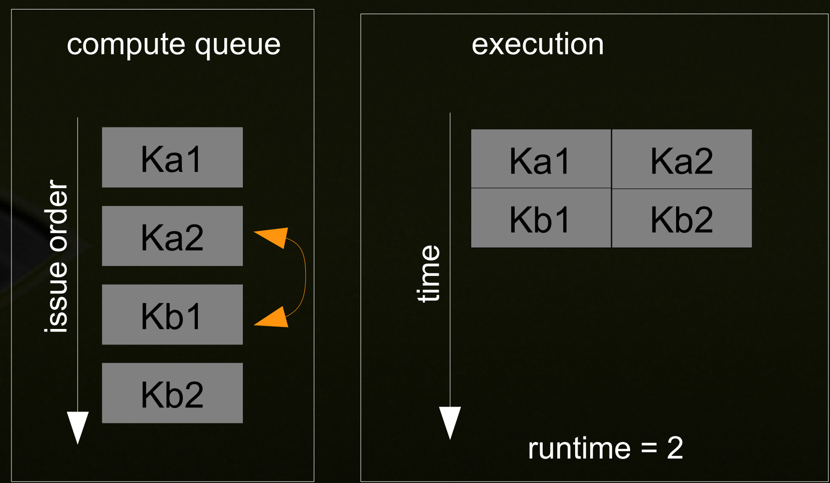
Example - Blocked Kernel

- 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

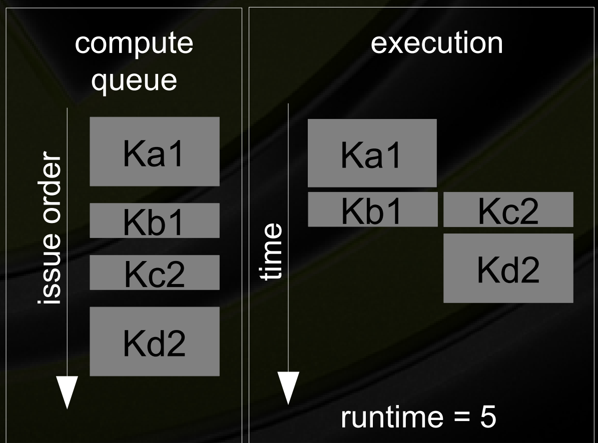


Example - Optimal Concurrency can Depend on Kernel Execution Time

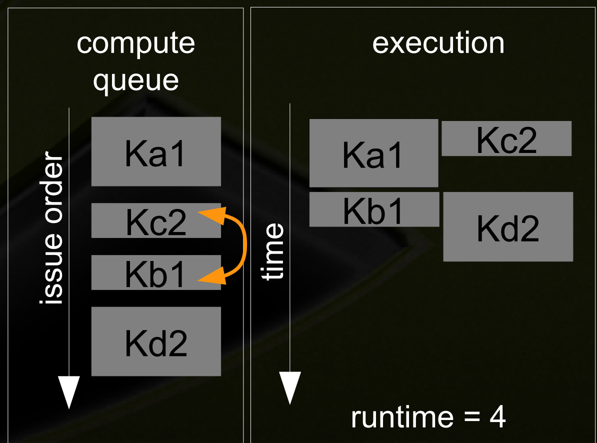
- Two streams – just issuing CUDA kernels – but kernels are different 'sizes'
  - Stream 1 : Ka1 {2}, Kb1 {1}
  - Stream 2 : Kc2 {1}, Kd2 {2}
  - Kernels fill ½ of the SM resources

issue order matters!
execution time matters!

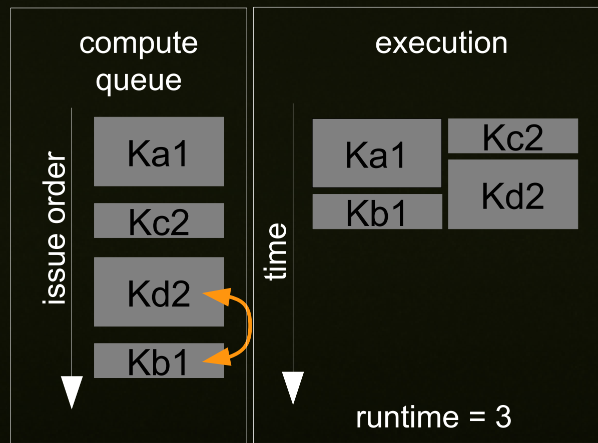
- Depth first



- Breadth first



- Custom

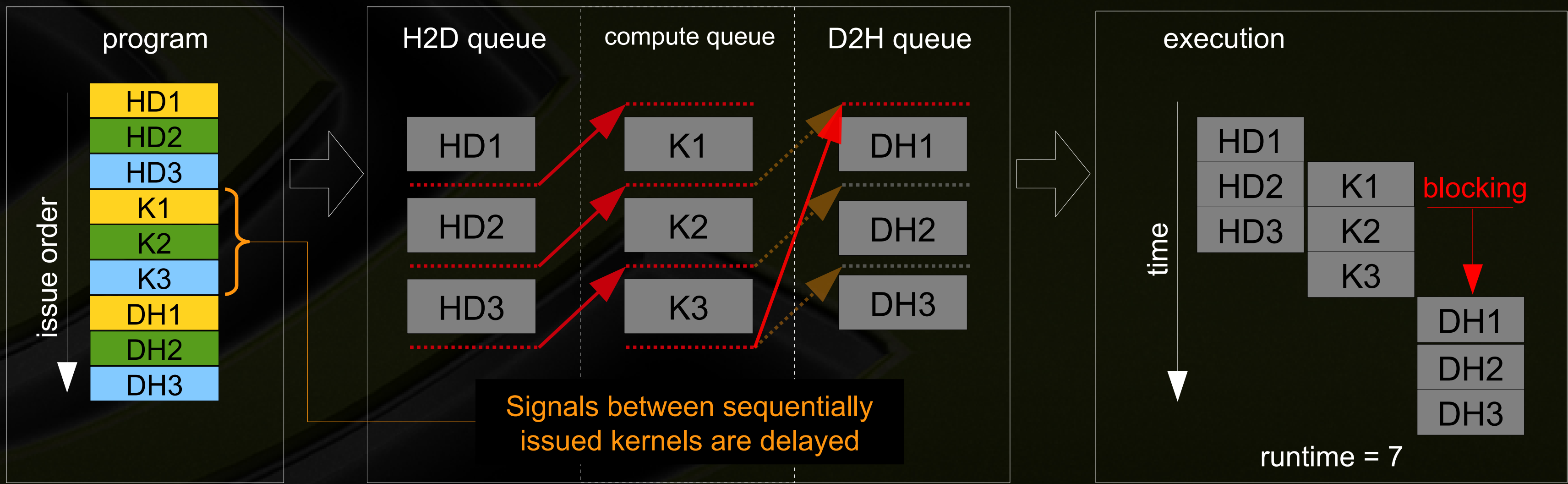


Concurrent Kernel Scheduling

- 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

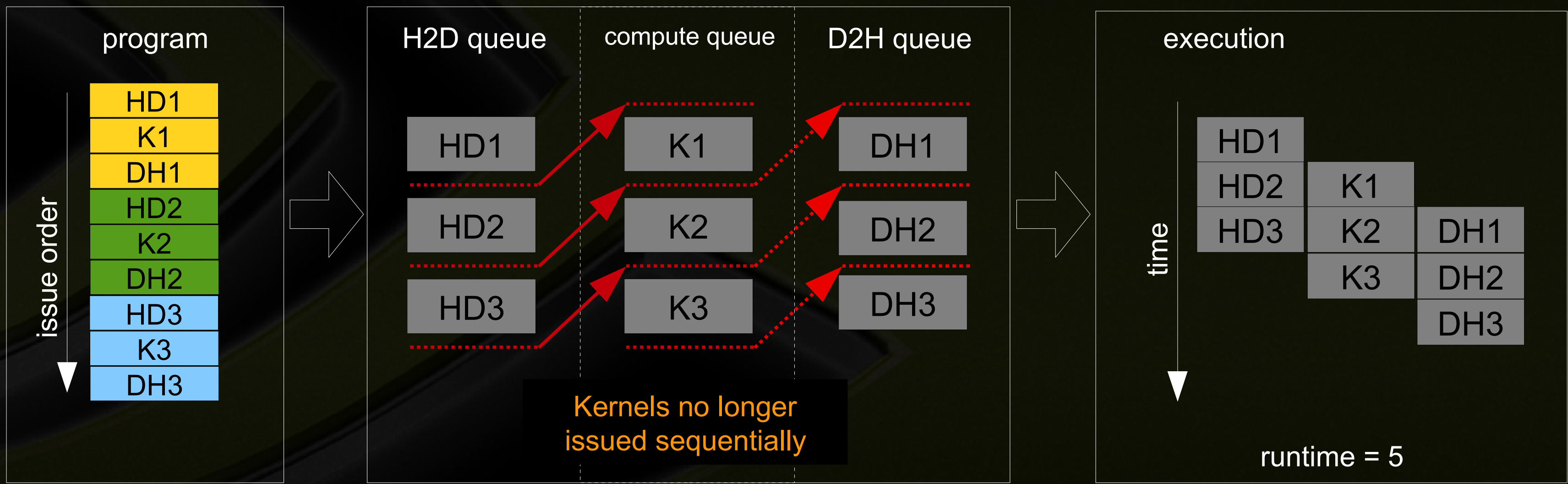
Example – Concurrent Kernels and Blocking

- 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



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)