Introduction to Streams

Stanford CME, May 2013

Steve Rennich

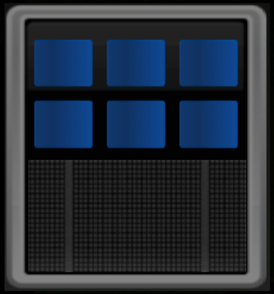
Nvidia - Developer Technology Compute

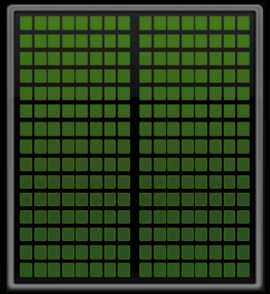
Streams and Concurrency

- Concurrency
  - The ability to perform multiple operations simultaneously
    - Compute kernels on the GPU
    - Data transfer to device (H2D)
    - Data transfer to host (D2H)
    - Operations on the CPU
  - Enables improved performance
- Streams
  - How concurrency is achieved

Simple Accelerator Model

- Tasks are offloaded from CPU to GPU

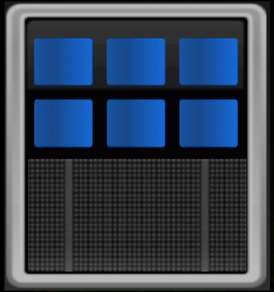
CPU


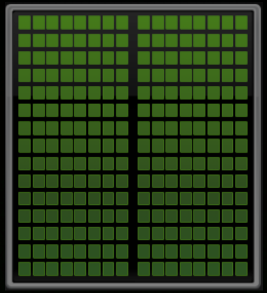

GPU

Simple Accelerator Model

- Tasks are offloaded from CPU to GPU

CPU



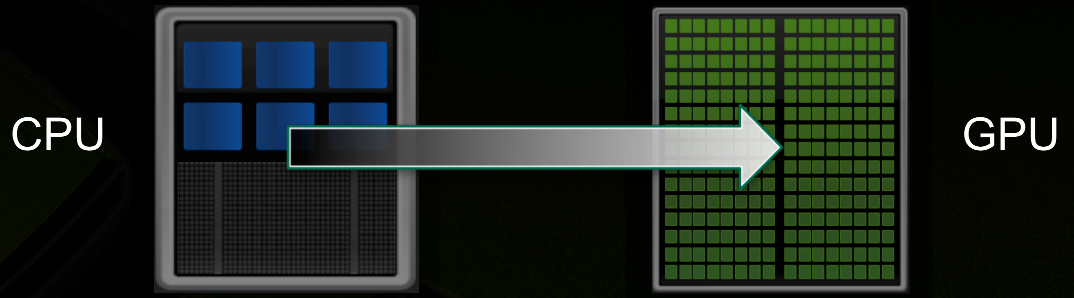


GPU

1. Setup data on CPU
2. Copy data from CPU to GPU (H2D)
3. Launch kernel on GPU
4. Copy result from GPU to CPU (D2H)
5. Repeat …

Simple Accelerator Model

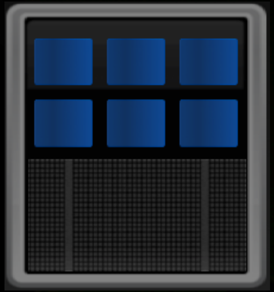
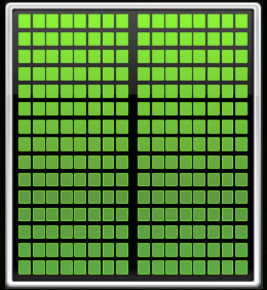
- Tasks are offloaded from CPU to GPU



1. Setup data on CPU
2. Copy data from CPU to GPU (H2D)
3. Launch kernel on GPU
4. Copy result from GPU to CPU (D2H)
5. Repeat …

Simple Accelerator Model

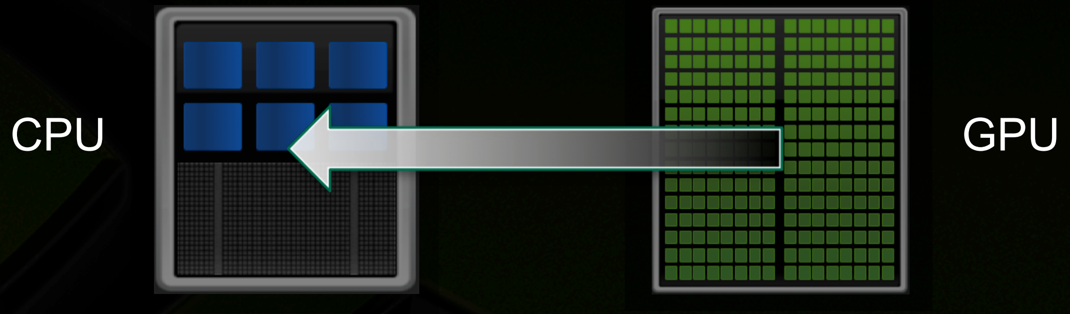
- Tasks are offloaded from CPU to GPU

CPU   GPU

1. Setup data on CPU
2. Copy data from CPU to GPU (H2D)
3. Launch kernel on GPU
4. Copy result from GPU to CPU (D2H)
5. Repeat …

Simple Accelerator Model

- Tasks are offloaded from CPU to GPU

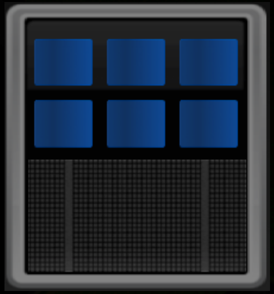


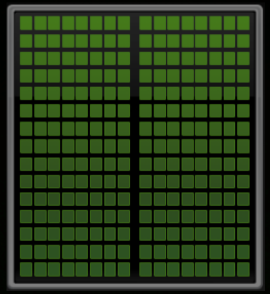
1. Setup data on CPU
2. Copy data from CPU to GPU (H2D)
3. Launch kernel on GPU
4. Copy result from GPU to CPU (D2H)
5. Repeat …

Simple Accelerator Model

- Tasks are offloaded from CPU to GPU

CPU



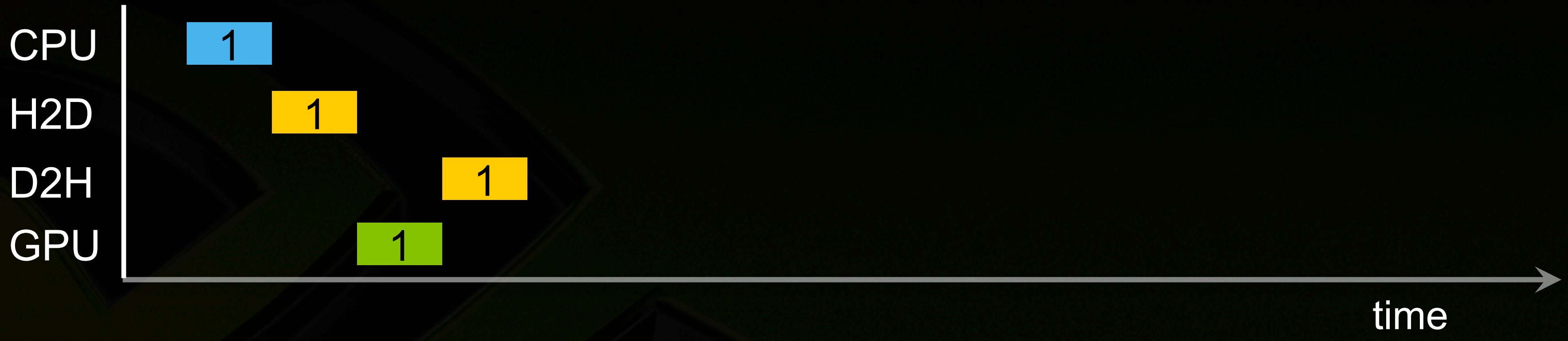


GPU

1. Setup data on CPU
2. Copy data from CPU to GPU (H2D)
3. Launch kernel on GPU
4. Copy result from GPU to CPU (D2H)
5. Repeat …

Simple Processing Flow Timeline

Simple Processing Flow Timeline

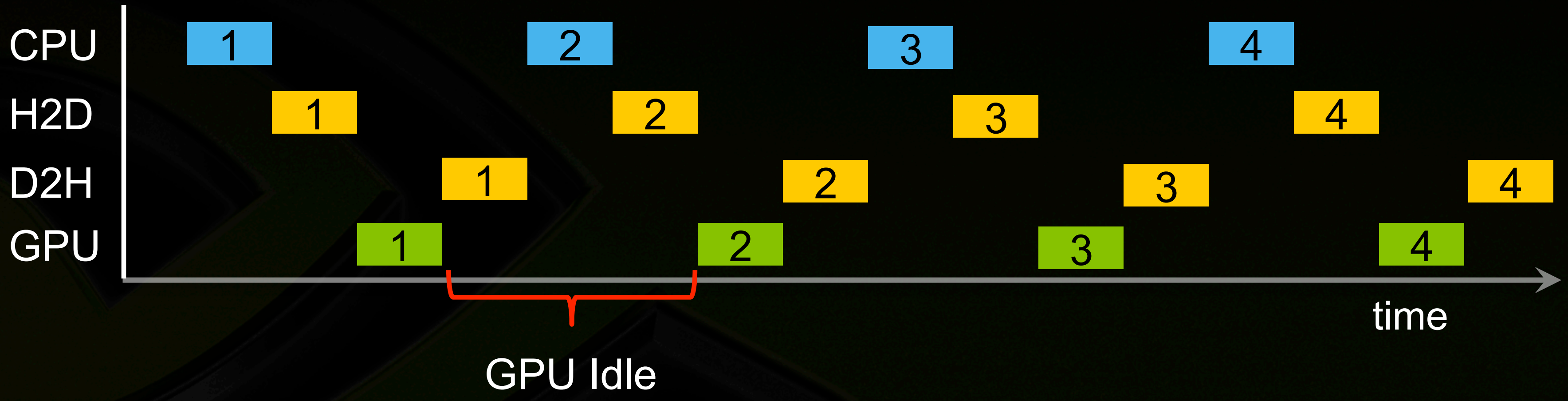


Simple Processing Flow Timeline

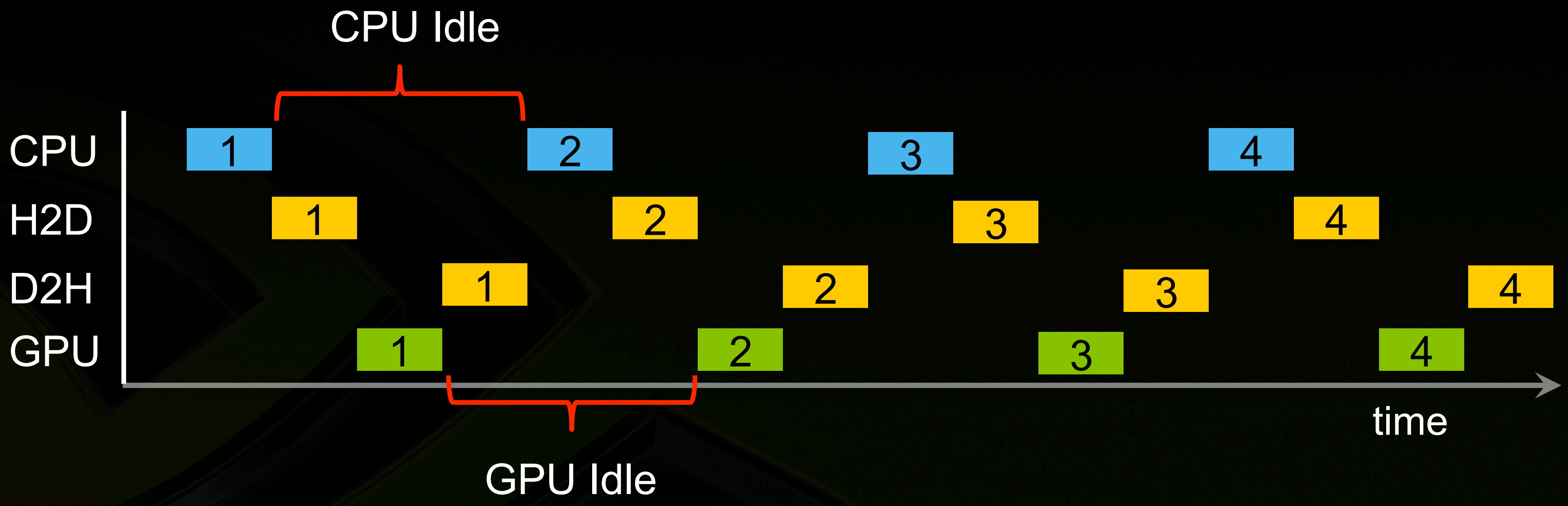
Simple Processing Flow Timeline

Simple Processing Flow Timeline

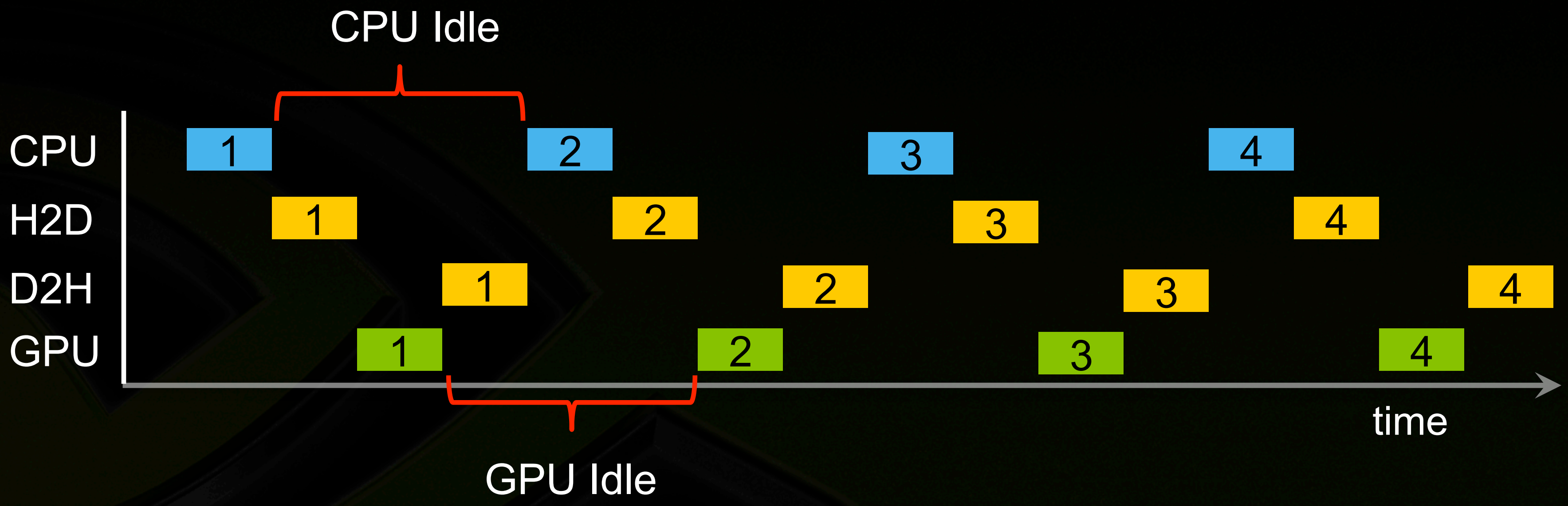
Simple Processing Flow Timeline



Simple Processing Flow Timeline

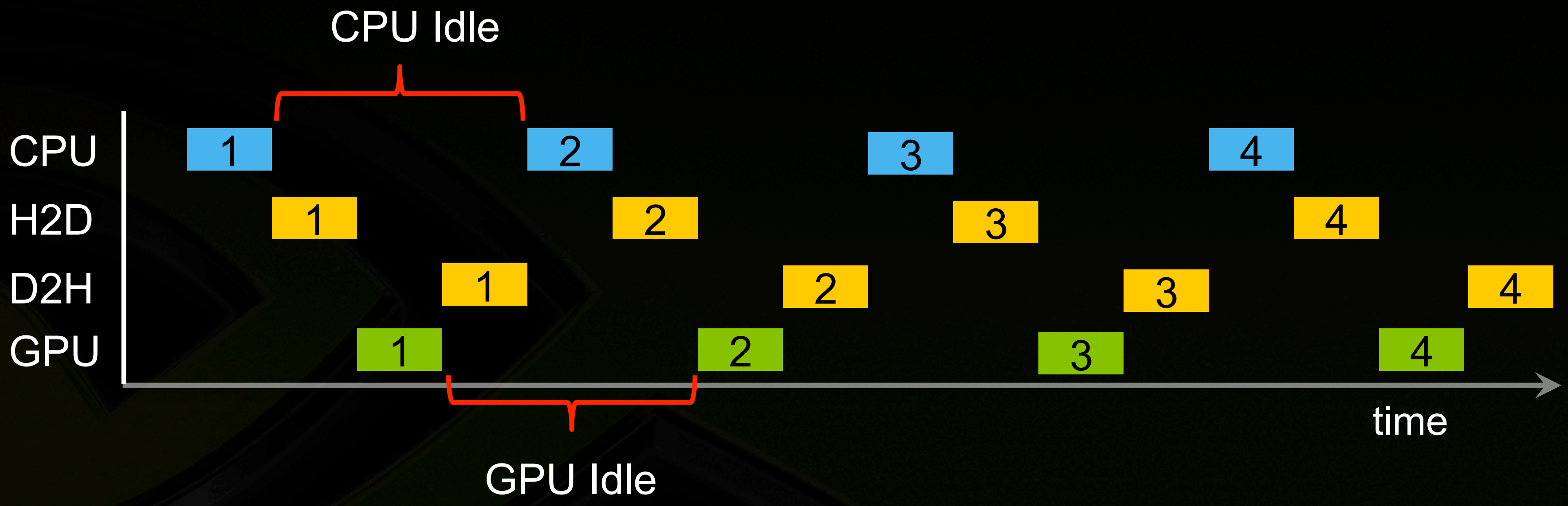


Simple Processing Flow Timeline



- Computing resources are poorly utilized
- We can use concurrency to improve utilization

Simple Processing Flow Timeline



- Computing resources are poorly utilized
- We can use concurrency to improve utilization

Concurrent Processing Flow Timeline

- Efficient utilization of all computing resources

Concurrent Processing Flow Timeline

- Efficient utilization of all computing resources

Concurrent Processing Flow Timeline

- Efficient utilization of all computing resources

Concurrent Processing Flow Timeline

- Efficient utilization of all computing resources

Concurrent Processing Flow Timeline

- Efficient utilization of all computing resources

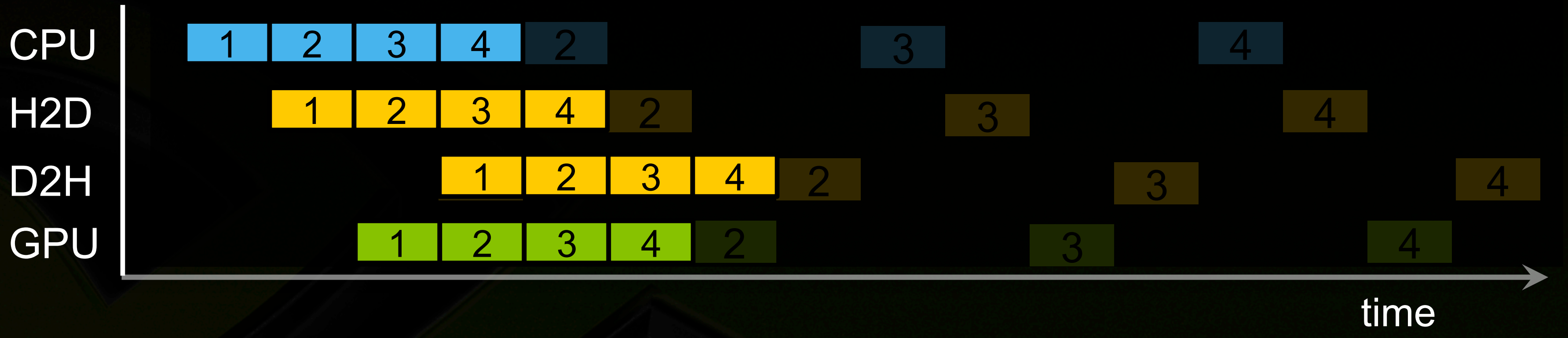
Concurrent Processing Flow Timeline

- Efficient utilization of all computing resources

Concurrent Processing Flow Timeline

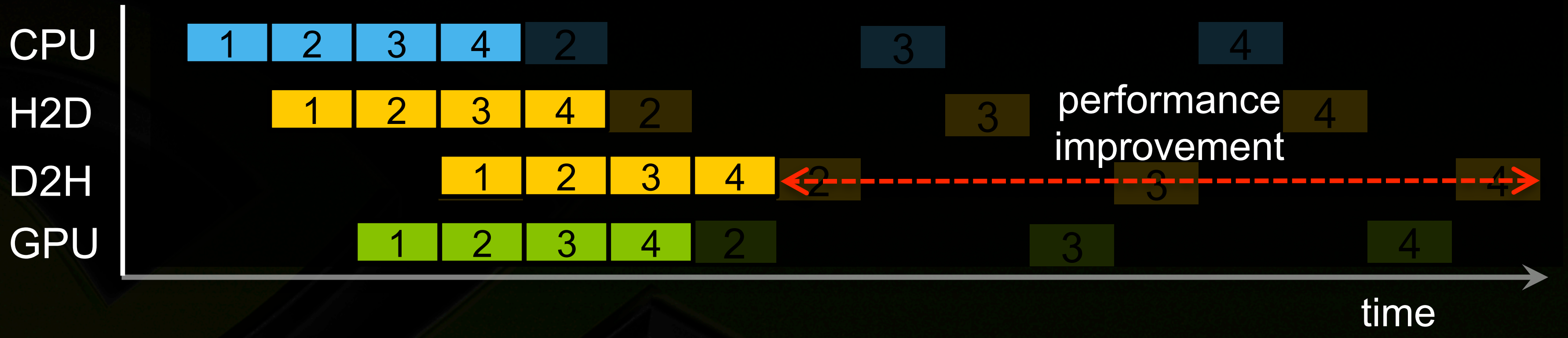
- Efficient utilization of all computing resources

Concurrent Processing Flow Timeline



- Efficient utilization of all computing resources

Concurrent Processing Flow Timeline



- Efficient utilization of all computing resources

Streams and Concurrency

- Concurrency
  - The ability to perform multiple operations simultaneously
    - Compute kernels on the GPU
    - Data transfer to device (H2D)
    - Data transfer to host (D2H)
    - Operations on the CPU
  - Enables improved performance
- Streams
  - How concurrency is achieved

Enabling Concurrency with Streams

CUDA Streaming Model

- All transfers and kernels are placed into a *stream*
  - *Stream*: A sequence of operations that execute in issue-order
  - Operations within a stream will not overlap
  - Operations in different streams may run concurrently
  - Operations from different streams may be interleaved
- Stream is the 4th launch parameter
  - kernel <<< blocks, threads, smem , stream >>> ()

Default Stream

- Stream used when no stream is specified
  - a.k.a. Stream ‘0’
  - a.k.a. ‘Null Stream’
- Completely synchronous w.r.t. host and device
  - As if a cudaDeviceSynchronize() was inserted before and after every operation
- Exceptions – asynchronous w.r.t. host
  - Kernel launches
  - cudaMemcpy*Async
  - cudaMemset*Async
  - cudaMemcpy within the same device

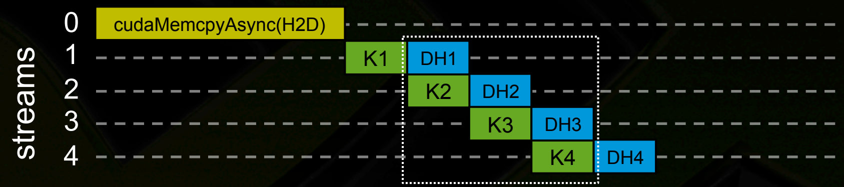
Requirements for concurrency

- Operations must be in different, non-default, streams
- cudaMemcpyAsync with host from ‘pinned’ memory
  - page-locked memory
  - Allocated usng cudaMallocHost() or cudaHostAlloc()
- Sufficient resources must be available
  - cudaMemcpyAsyncs in different directions
  - Device resources (SMEM, registers, etc.)

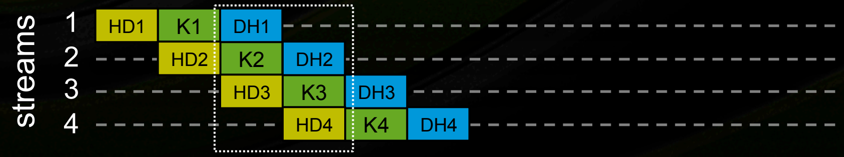
Concurrency Examples

- Serial (1x)

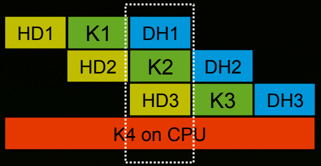
- 2-way concurrency (up to 2x)



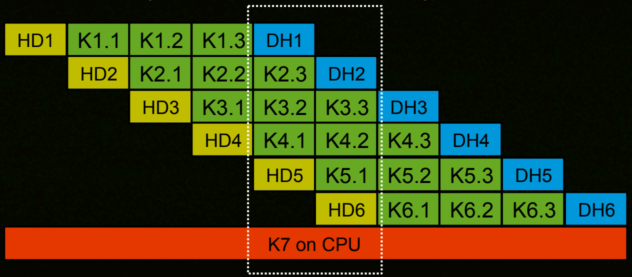
- 3-way concurrency (up to 3x)



- 4-way concurrency (3x+)



- 4+ way concurrency

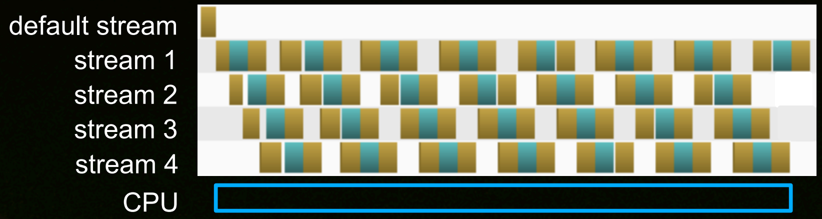


Example – Tiled DGEMM

- CPU (dual 6 core SandyBridge E5-2667 @2.9 Ghz, MKL)
  - 222 Gflop/s
- GPU (K20X)
  - Serial: 519 Gflop/s (2.3x)
  - 2-way: 663 Gflop/s (3x)
  - 3-way: 990 Gflop/s (4x)
- GPU + CPU
  - 4-way con.: 1180 Gflop/s (5.3x)

DGEMM: m=n=16384, k=1408

Nvidia Visual Profiler (nvvp)



- Obtain maximum performance by leveraging concurrency
- Removes impact of PCIe bandwidth
- Removes device memory size limitations

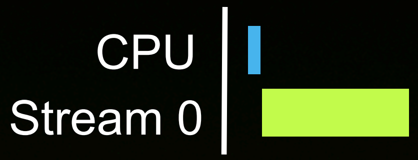
Managing Streams

- `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
- `cudaStreamCreateWithFlags(&stream,cudaStreamNonBlocking);`
  - Allocates a stream that is asynchronous with stream 0
  - Cuda 5.0 and newer only

Kernel Concurrency

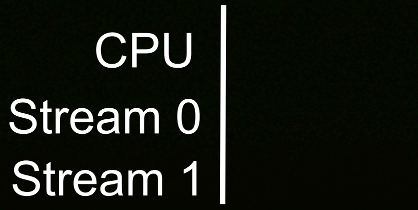
- Assume foo only utilizes 50% of the GPU
- Default stream

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



- Default & user streams

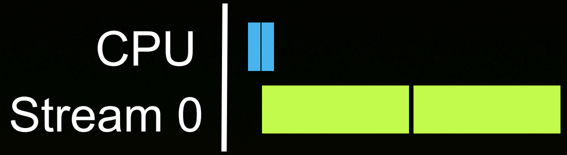
```
cudaStream_t stream1;
cudaStreamCreate(&stream1);
foo<<<blocks,threads>>>();
foo<<<blocks,threads,0,stream1>>>();
cudaStreamDestroy(stream1);
```



Kernel Concurrency

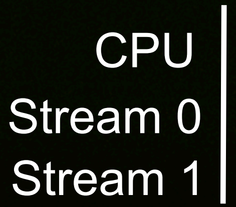
- Assume foo only utilizes 50% of the GPU
- Default stream

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



- Default & user streams

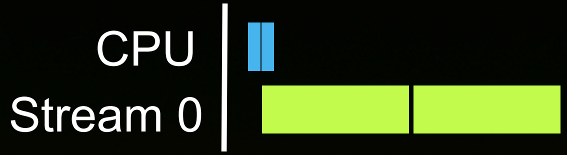
```
cudaStream_t stream1;
cudaStreamCreate(&stream1);
foo<<<blocks,threads>>>();
foo<<<blocks,threads,0,stream1>>>();
cudaStreamDestroy(stream1);
```



Kernel Concurrency

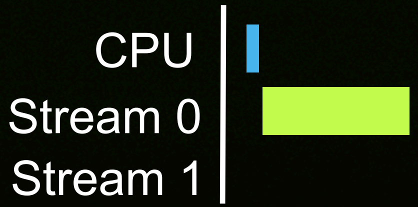
- Assume foo only utilizes 50% of the GPU
- Default stream

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



- Default & user streams

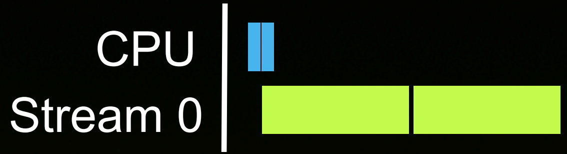
```
cudaStream_t stream1;
cudaStreamCreate(&stream1);
foo<<<blocks,threads>>>();
foo<<<blocks,threads,0,stream1>>>();
cudaStreamDestroy(stream1);
```



Kernel Concurrency

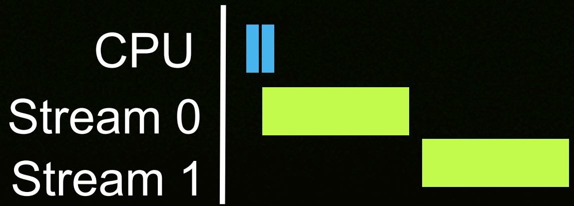
- Assume foo only utilizes 50% of the GPU
- Default stream

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



- Default & user streams

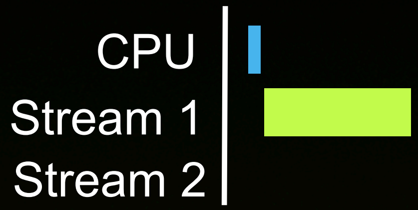
```
cudaStream_t stream1;
cudaStreamCreate(&stream1);
foo<<<blocks,threads>>>();
foo<<<blocks,threads,0,stream1>>>();
cudaStreamDestroy(stream1);
```



Kernel Concurrency

- User streams

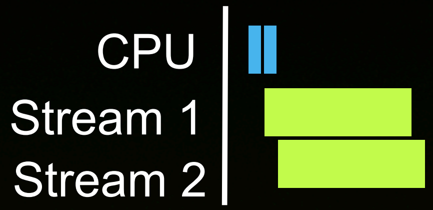
```
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
foo<<<blocks,threads,0,stream1>>>();
foo<<<blocks,threads,0,stream2>>>();
cudaStreamDestroy(stream1);
cudaStreamDestroy(stream2);
```



Kernel Concurrency

- User streams

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



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
  - Only 1 transfer per direction at a time
  - Memory on the host must be pinned

Pinned Memory

- Pageable Memory (malloc, new, etc)
  - Can be paged in and out
  - Achieves a low % of peak bandwidth
- Pinned Memory
  - Cannot be paged in and out
  - Achieves a high % of peak bandwidth
- `cudaMallocHost(), cudaHostAlloc(), cudaFreeHost()`
  - Allocate/Free pinned memory on the host
  - Replaces malloc/free
- `cudaHostRegister(), cudaHostUnregister()`
  - Pins/Unpins existing memory

Paged Memory Example

```
int *h_ptr, *d_ptr;

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

cudaMemcpyAsync(d_ptr, h_ptr, bytes, cudaMemcpyHostToDevice, stream);

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

(synchronous)

Pinned Memory Example

```
int *h_ptr, *d_ptr;

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

cudaMemcpyAsync(d_ptr, h_ptr, bytes, cudaMemcpyHostToDevice, stream);

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

(asynchronous)

Pinned Memory Example 2

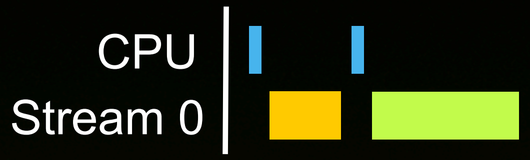
```
int *h_ptr, *d_ptr;

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

cudaMemcpyAsync(d_ptr,h_ptr, bytes, cudaMemcpyHostToDevice, stream);

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

(asynchronous)

Concurrency Examples

Synchronous

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



Asynchronous Same Stream

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

Asynchronous Different Streams

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

Concurrency Examples

Synchronous

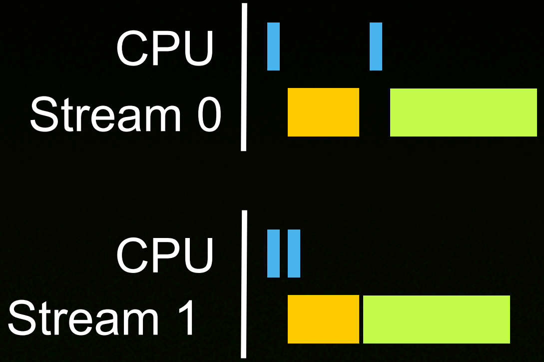
```
cudaMemcpy(...);
foo<<<...>>>();
```

Asynchronous Same Stream

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

Asynchronous Different Streams

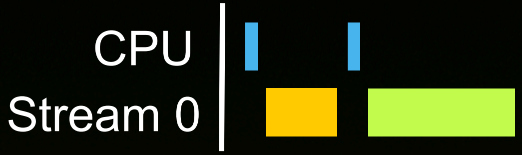
```
cudaMemcpyAsync(...,stream1);
foo<<<...,stream2>>>();
```



Concurrency Examples

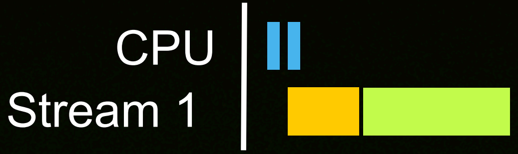
Synchronous

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



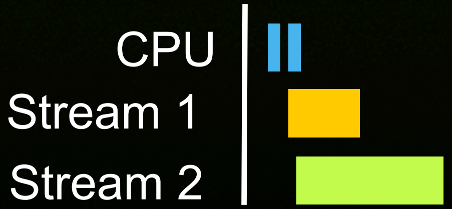
Asynchronous Same Stream

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



Asynchronous Different Streams

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



Implicit Synchronization

- These functions *implicitly* synchronize the device
  - cudaMalloc, cudaFree
  - cudaEventCreate, cudaEventDestroy,
  - cudaStreamCreate, cudaStreamDestroy
  - cudaHostRegister, cudaHostUnregister
  - cudaFuncSetCacheConfig
- Avoid by reusing memory and data structures as much as possible

Explicit Synchronization

- CUDA provides mechanisms for expressing synchronization between the host, device, and streams.

- Synchronize everything
  - cudaDeviceSynchronize()
  - Blocks host until all issued CUDA calls are complete

- Synchronize host w.r.t. a specific stream
  - cudaStreamSynchronize (streamid)
  - Blocks host until all issued CUDA calls in streamid are complete

Explicit Synchronization using Events

- Mechanism for arbitrary synchronization
  - Create ‘events’ at specific points within streams

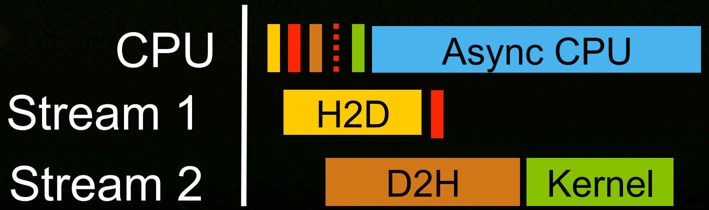
- Synchronize using events
  - cudaEventRecord (event, streamid)
  - cudaEventQuery (event)
  - cudaEventSynchronize (event)
  - cudaStreamWaitEvent (stream, event)

Explicit Synchronization Example

```
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
```

Explicit Synchronization Example

```
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
```



Advanced: Stream Callbacks

- Cuda 5.0 now allows you to add stream callbacks (K20 or newer)
  - Useful for launching work on the host when something has completed

```
void CUDART_CB MyCallback(void *data){
  ...
}
...
  MyKernel<<<100, 512, 0, stream>>>();
  cudaStreamAddCallback(stream, MyCallback, (void*)i, 0);
```

- Callbacks are processed by a driver thread
  - The same thread processes all callbacks
  - You can use this thread to signal other threads

Advanced: Multiple GPUs

- cudaDeviceSynchronize syncs with current device only
- Streams are associated with a particular device
  - Current device when stream was created
  - Error if stream is referenced which its device is not current
  - Each device has its own default stream
- Events are associated with a particular device
  - cudaEventRecord will fail if event and stream associate with different devices
  - cudaEventElapsedTime must take two events associated with the same device
- Synchronization works between devices with any event
  - cudaEventQuery, cudaEventSynchronize, cudaStreamWaitEvent

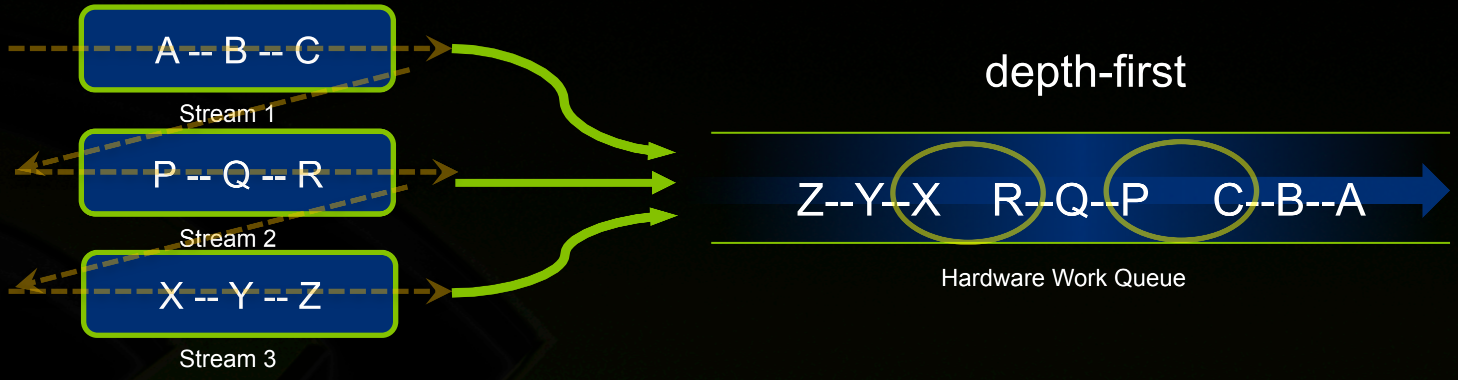
Common Streaming Issues

- Using the default stream
- Not using asynchronous version of memcpy
- Not using pinned host memory for memcpy
- Implicit synchronization
- Concurrency can be disabled for debugging
  - CUDA_LAUNCH_BLOCKING=1

Potential Hazards

- Concurrency is broken if there are more than 62 outstanding operations
  - Kernels, memory copies, recorded events
  - That is, in 'issue order' concurrent operations must not be separated by more than 62 other issues
  - Further operations are serialized
  - Avoid by changing issue order
- Kernels using more than 8 textures cannot run concurrently
- Switching L1/Shared configuration may break concurrency
- Hardware older than SM3.5 suffers from false serialization

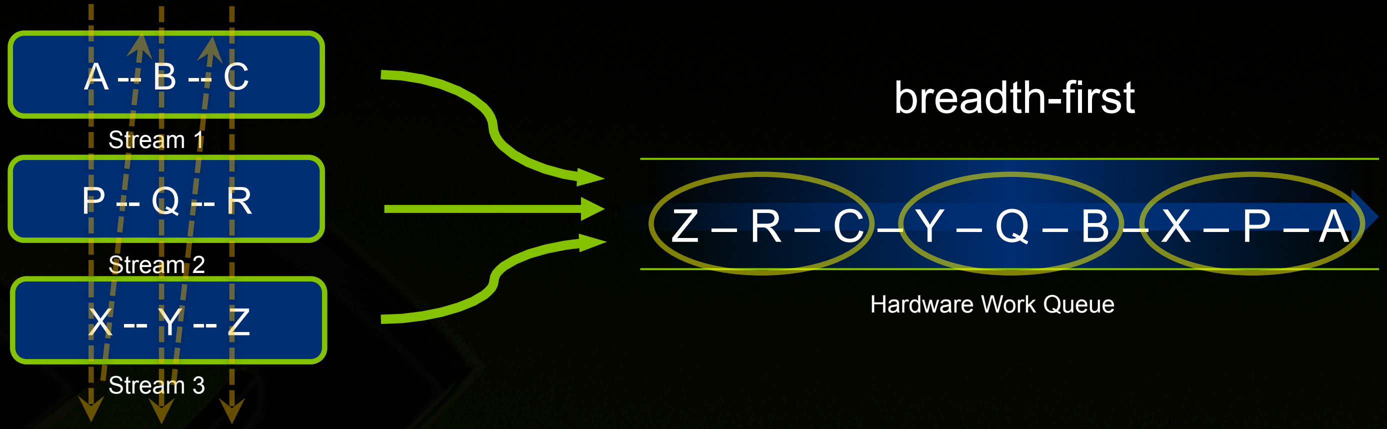
Fermi – Concurrent Kernels



Fermi allows 16-way concurrency

- But CUDA kernels multiplex into a single queue
- Issue order matters for concurrency
- https://developer.nvidia.com/gpu-computing-webinars
- http://www.stanford.edu/group/ttsdocs/cgi-bin/techbriefingvideos/2013/01/18/cuda-programming-your-gpu

Fermi – Concurrent Kernels



Fermi allows 16-way concurrency

- But CUDA kernels multiplex into a single queue
- Issue order matters for concurrency
- https://developer.nvidia.com/gpu-computing-webinars
- http://www.stanford.edu/group/ttsdocs/cgi-bin/techbriefingvideos/2013/01/18/cuda-programming-your-gpu

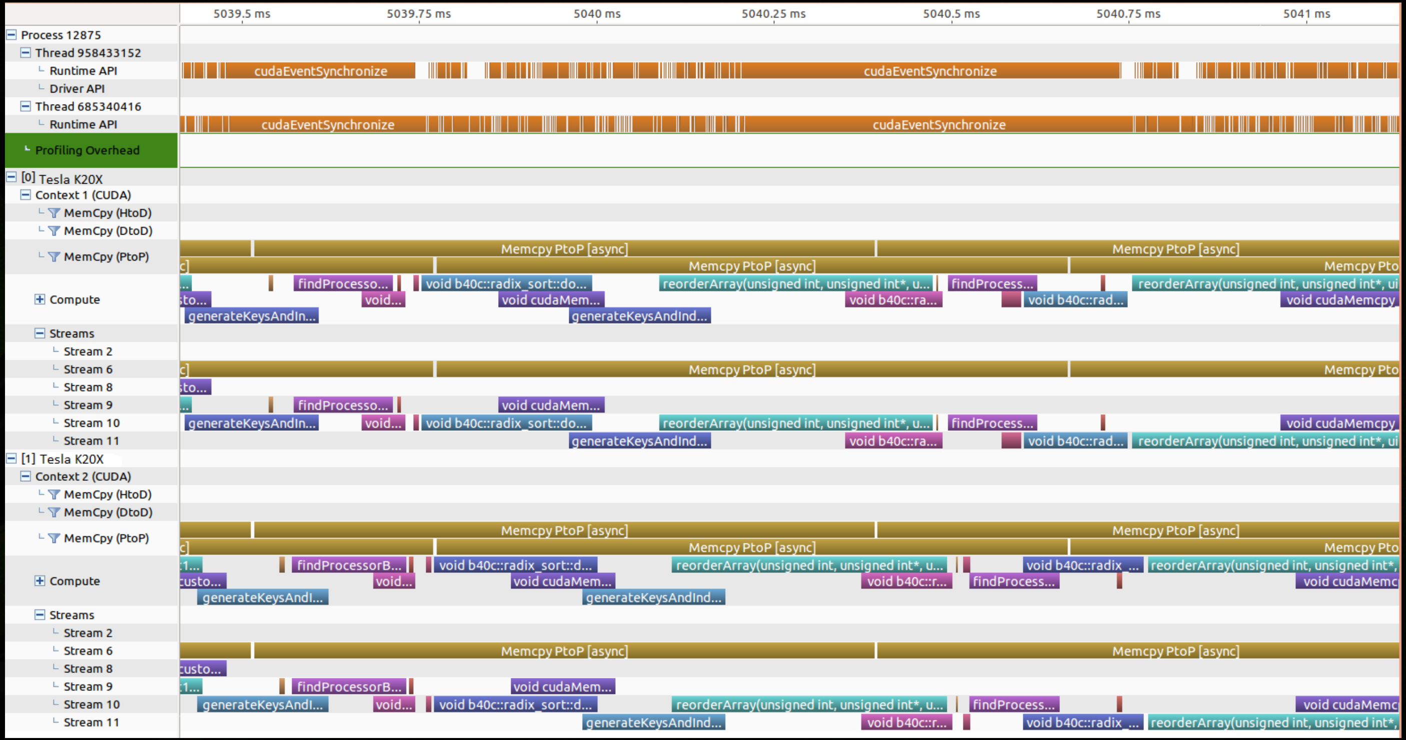
K20 Improved Concurrency

depth-first or
breadth-first

Kepler allows 32-way concurrency

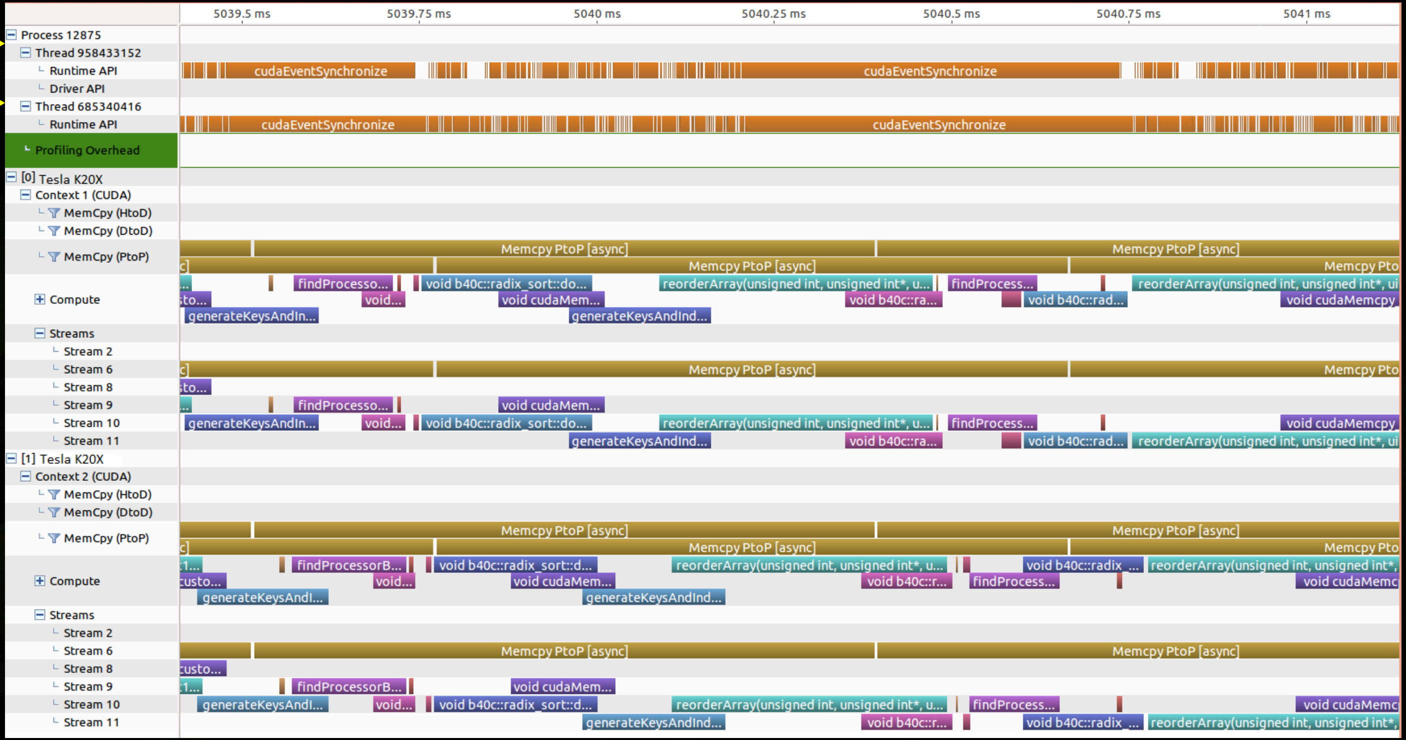
- One kernel queue per stream
- No inter-stream dependencies

Nvidia Visual Profiler (nvvp)



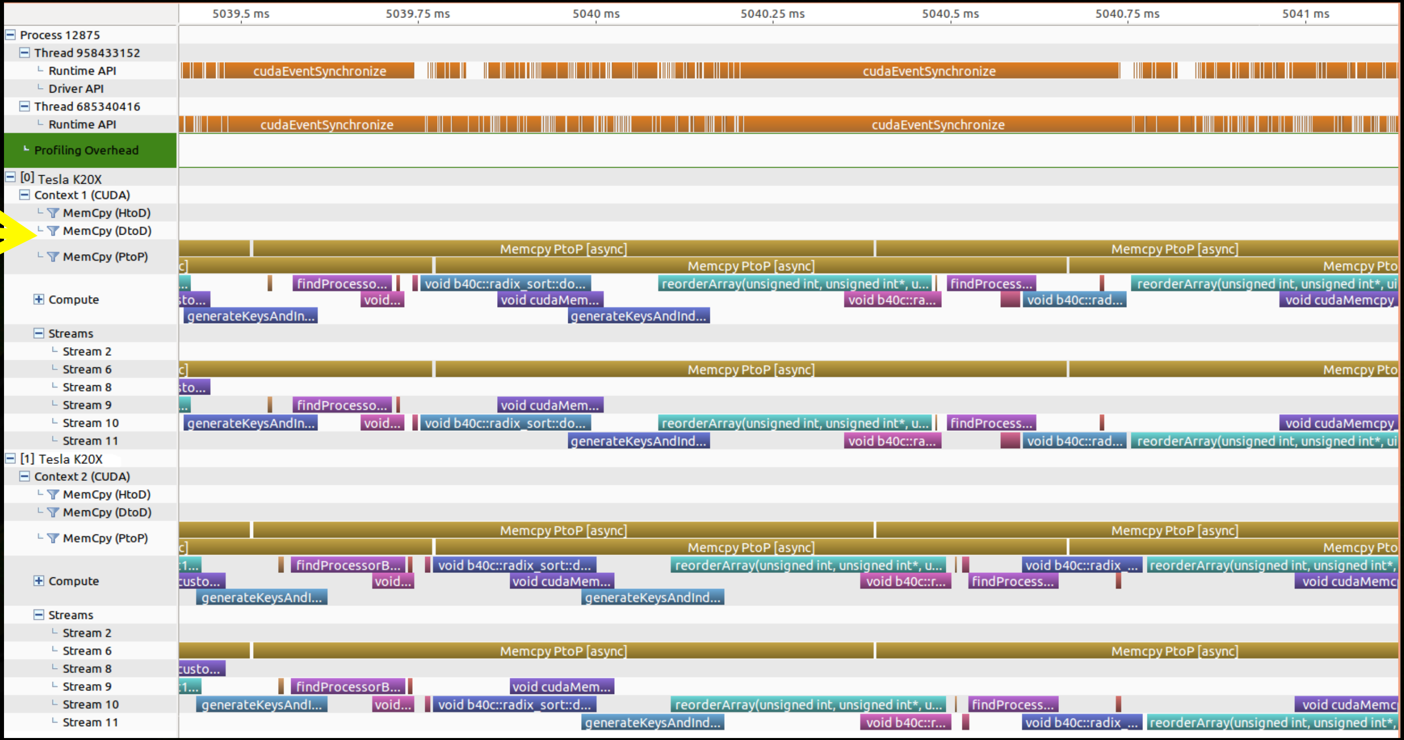
Nvidia Visual Profiler (nvvp)

What is the host doing?



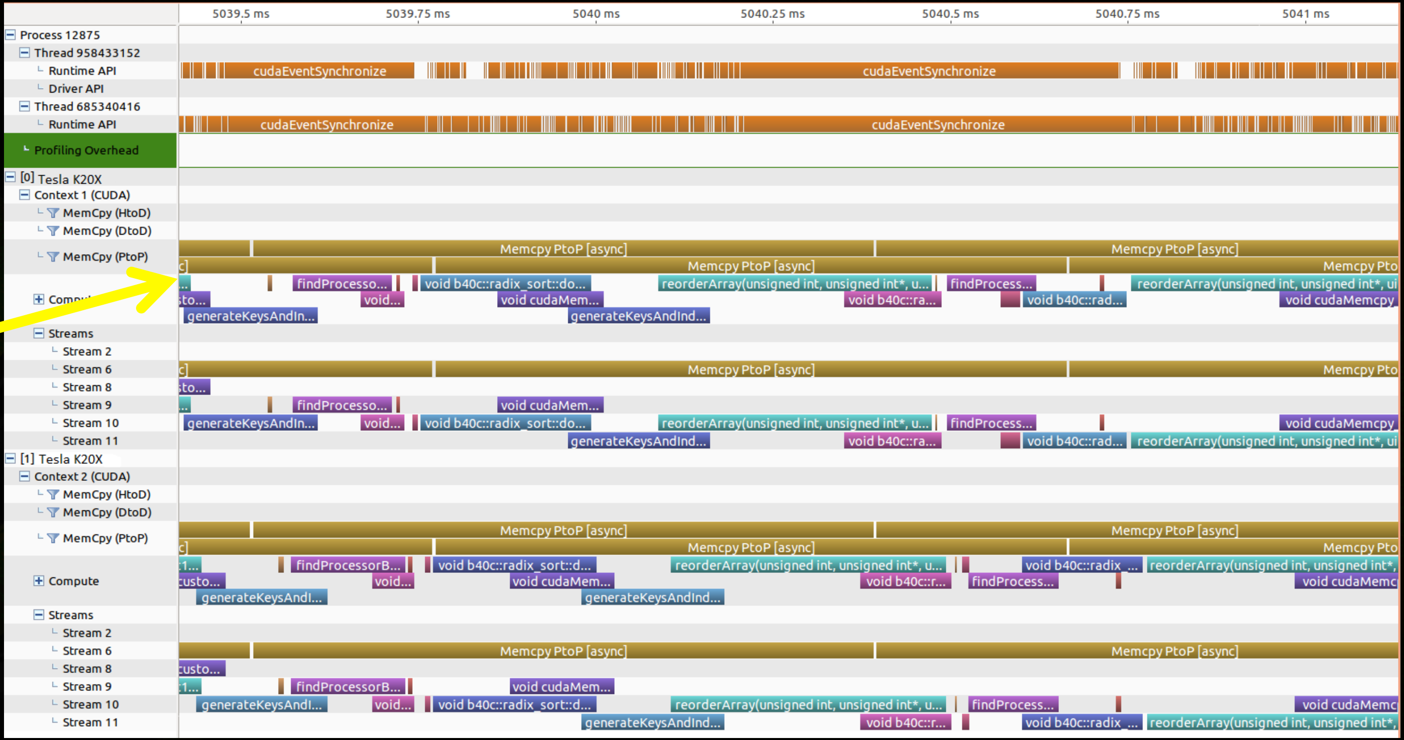
Nvidia Visual Profiler (nvvp)

What copies
are happening?



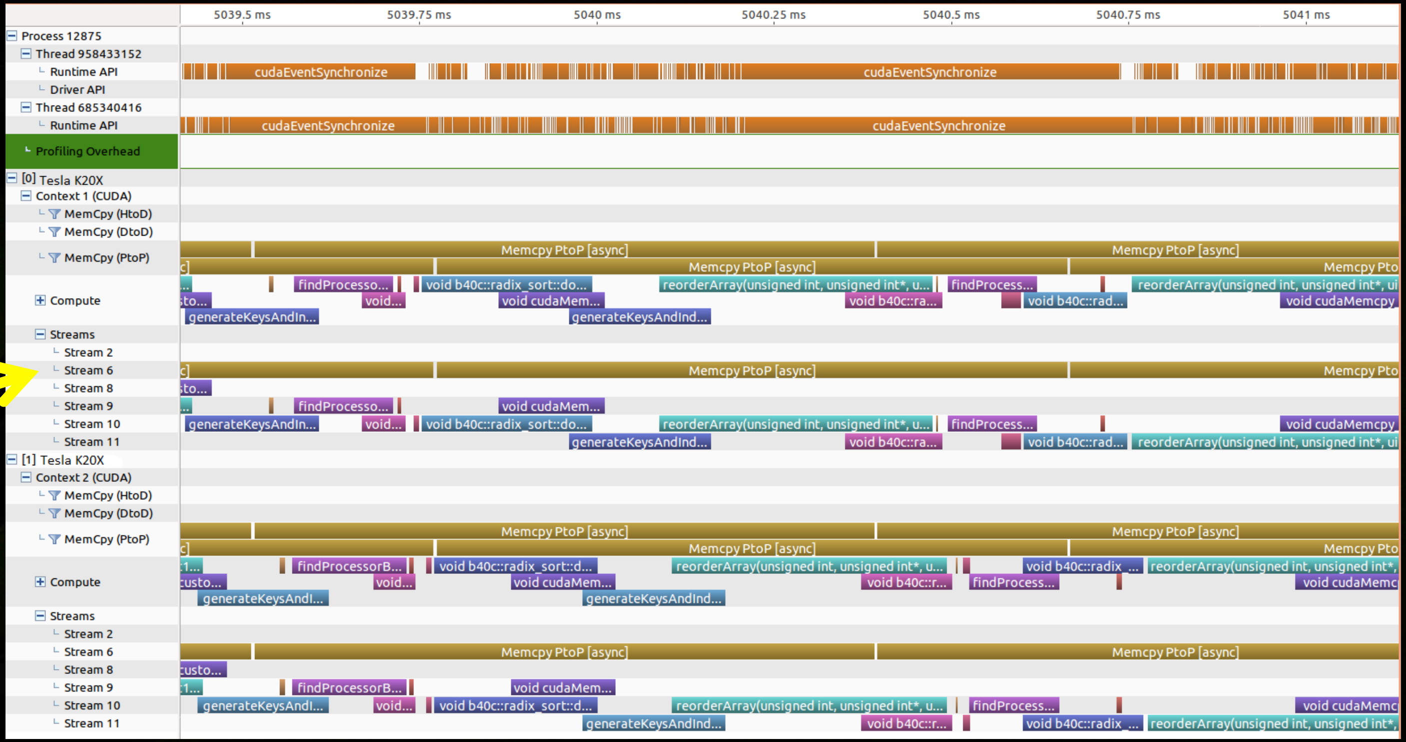
Nvidia Visual Profiler (nvvp)

How much overlap?

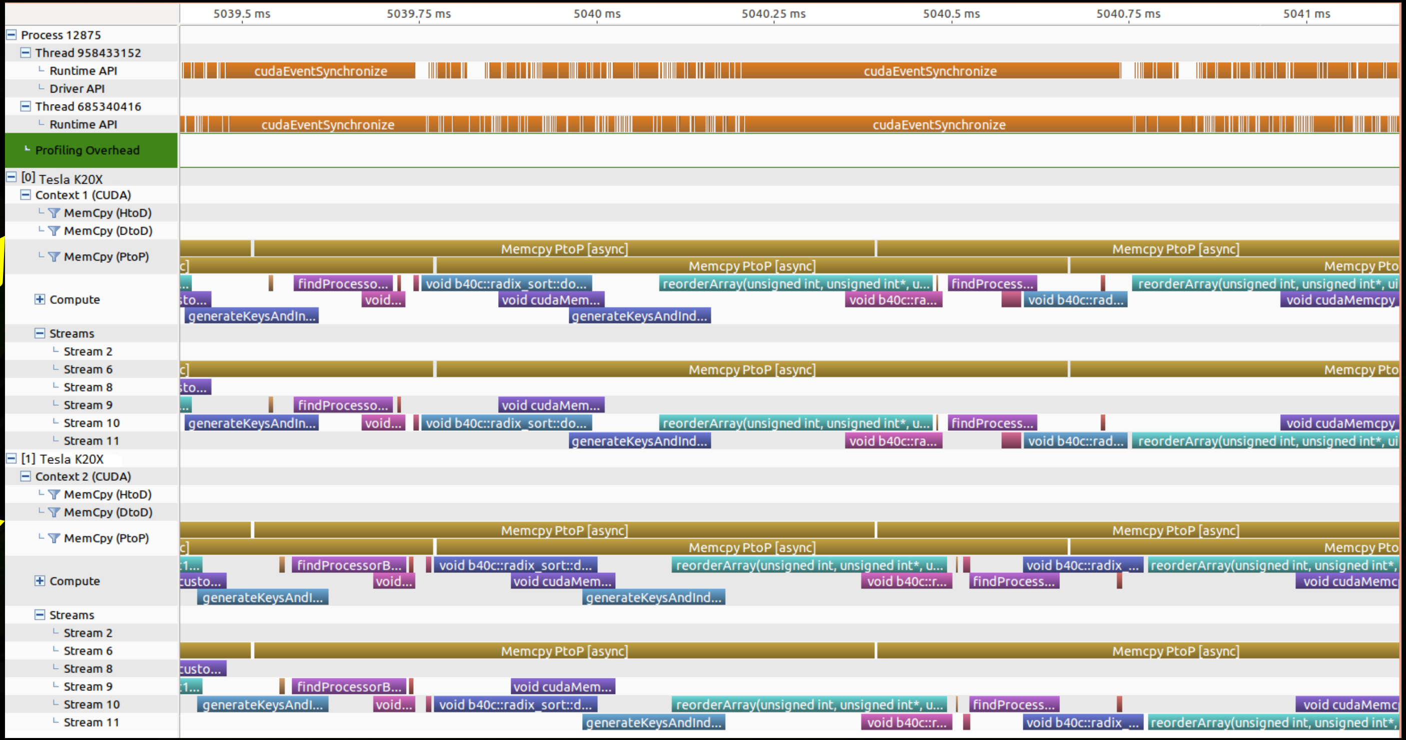


Nvidia Visual Profiler (nvvp)

Streams



Nvidia Visual Profiler (nvvp)



Multi GPU

Concurrency Guidelines

- Code to the programming model – streams
  - Future devices will continually improve HW rep. of programming model
- Pay attention to operations which can break concurrency
  - Use of default stream
  - Implicit synchronization
- Synchronize only when required
  - Excessive synchronization imparts overhead, limits scheduler
- **Use profilers! (nvvp, Nsight, …)**

Questions?