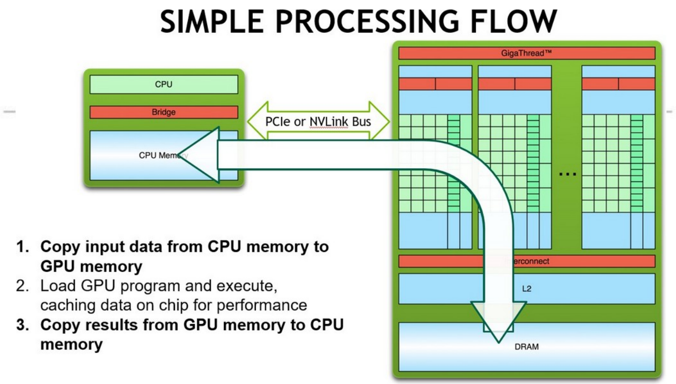
CUDA CONCURRENCY

Bob Crovella, 7/21/2020

AGENDA

- Concurrency - Motivation
- Pinned Memory
- CUDA Streams
- Overlap of Copy and Compute
- Use Case: Vector Math/Video Processing Pipeline
- Additional Stream Considerations
- Copy-Compute Overlap with Managed Memory
- Multi-GPU Concurrency
- Other Concurrency Scenarios: Kernel
- Concurrency, Host/Device Concurrency
- Further Study
- Homework

MOTIVATION



Recall 3 steps from session 1:

Naïve implementation leads to a processing flow like this:

1. Copy data to the GPU

2. Run kernel(s) on GPU

3. Copy results to host

duration

->Wouldn't it be nice if we could do this:

1. Copy data to the GPU

2. Run kernel(s) on GPU

3. Copy results to host

duration

PINNED MEMORY

固定内存

PINNED (NON-PAGEABLE) MEMORY

- Pinned memory enables:
    - faster Host<->Device copies
    - memcopies asynchronous with CPU
    - memcopies asynchronous with GPU
- Usage
    - `cudaHostAlloc` / `cudaFreeHost`
        - instead of malloc / free or new / delete
    - `cudaHostRegister` / `cudaHostUnregister`
        - pin regular memory (e.g. allocated with malloc) after allocation
- Implication:
    - pinned memory is essentially removed from host virtual (pageable) memory

CUDA STREAMS

STREAMS AND ASYNC API OVERVIEW

- Default API:
  - GPU kernel launches are synchronous with CPU
  - cudaMemcpy (D2H, H2D) block CPU thread (if memory not page-locked)
  - CUDA calls are serialized by the driver (legacy default stream)
- Streams and async functions provide:
  - cudaMemcpyAsync (D2H, H2D) asynchronous with CPU
  - Ability to concurrently execute a kernel and a memcopy
  - Concurrent copies in both directions (D2H, H2D) possible on most GPUs
- Stream = sequence of operations that execute in issue-order on GPU, like task queue
  - Operations from different streams may be interleaved
  - A kernel and memcopy from different streams can be overlapped

STREAM SEMANTICS

1. Two operations issued into the same stream will *execute in issue-order in GPU*. Operation B issued after Operation A will not begin to execute until Operation A has completed.

2. Two operations issued into separate streams have *no ordering prescribed by CUDA*.

3. Operation A issued into stream 1 may execute before, during, or after Operation B issued into stream 2.

- **Operation**: Usually, cudaMemcpyAsync or a kernel call.

  More generally, most CUDA API calls that take a stream parameter, as well as stream callbacks.

STREAM CREATION AND COPY/COMPUTE OVERLAP

- Requirements:
  - D2H or H2D memcopy from pinned memory Kernel and memcopy in different, non-0 streams

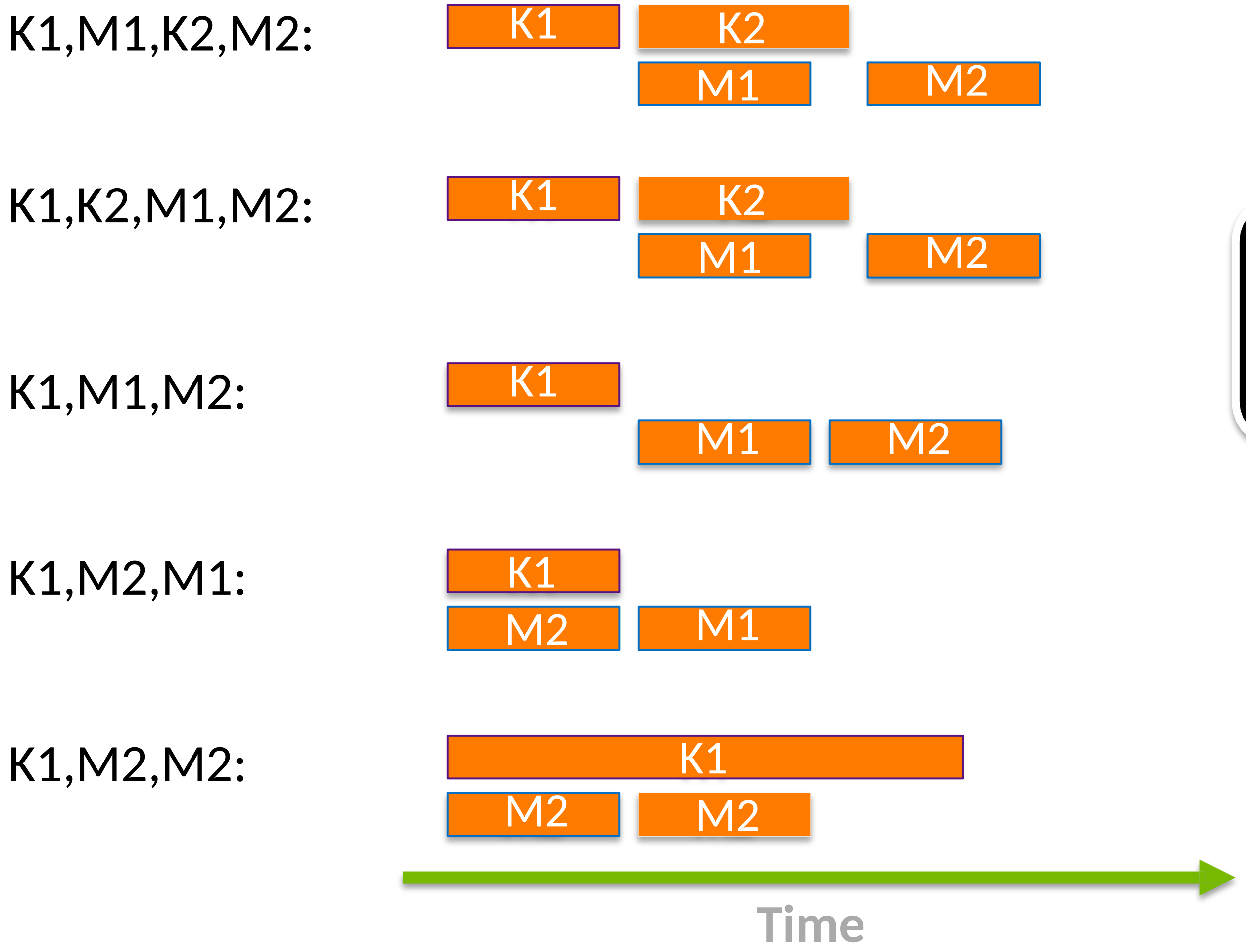
- Code:

```
cudaStream_t   stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

cudaMemcpyAsync( dst, src, size, dir, stream1 );   potentially
kernel<<<grid, block, 0, stream2>>>(…);            overlapped

cudaStreamQuery(stream1);          // test if stream is idle
cudaStreamSynchronize(stream2);    // force CPU thread to wait
cudaStreamDestroy(stream2);
```

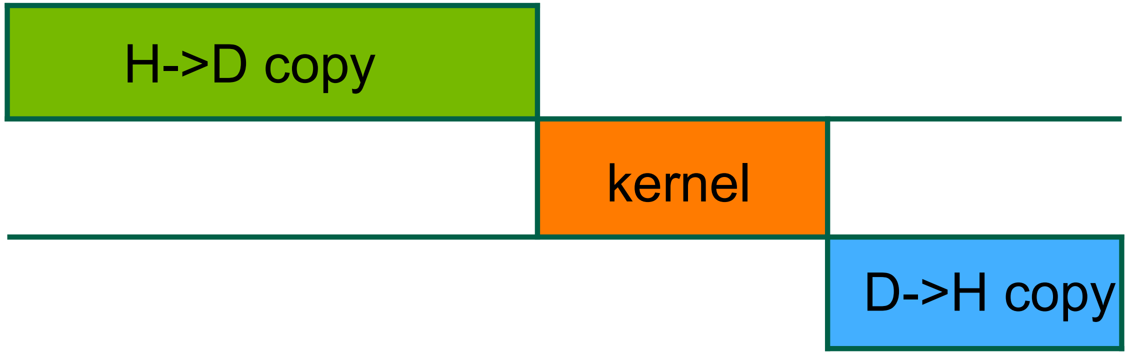
STREAM EXAMPLES



K: Kernel
M: Memcopy
Integer: Stream ID

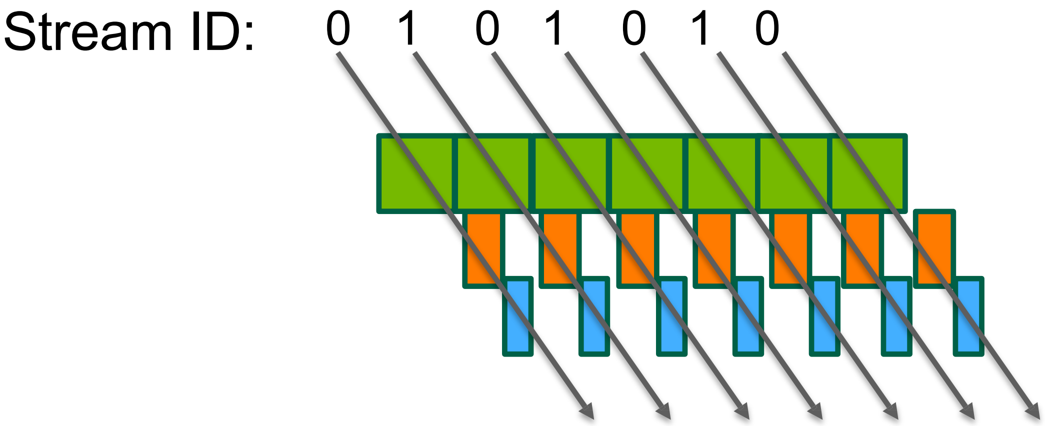
EXAMPLE STREAM BEHAVIOR FOR VECTOR MATH

(assumes algorithm decomposability)



non-streamed

```
cudaMemcpy(d_x, h_x, size_x,
cudaMemcpyHostToDevice);
Kernel<<<b, t>>>(d_x, d_y, N);
 cudaMemcpy(h_y, d_y, size_y,
cudaMemcpyDeviceToHost);
```



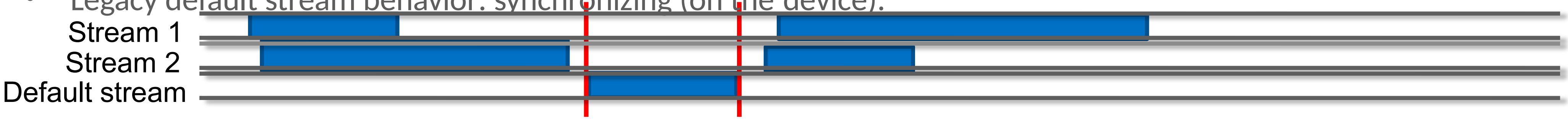
streamed

```
for (int i = 0, i<c; i++)
         offx = {
  size_t  offy  (size_x/c)*i;
  cudaMemcpyAsync(d_x+offx, h_x+offx,
 size_x/c, cudaMemcpyHostToDevice,
 size_t   (size_y/c)*i;
stream[i%ns]);
  Kernel<<<b/c, t, 0,  stream[i
%ns]>>>(d_x+offx, d_y+offy,  N/c);
  cudaMemcpyAsync(h_y+offy,
  d_y+offy,
size_y/c, cudaMemcpyDeviceToHost,
 stream[i%ns]);}
```

Similar: video processing pipeline

DEFAULT STREAM

- Kernels or cudaMemcpy… that do not specify stream (or use 0 for stream) are using the default stream
- Legacy default stream behavior: synchronizing (on the device):



  - All device activity issued prior to the item in the default stream must complete before default stream item begins
  - All device activity issued after the item in the default stream will wait for the default stream item to finish
  - All host threads share the same default stream for legacy behavior
  - Consider avoiding use of default stream during complex concurrency scenarios

MULTI-GPU

MULTI-GPU – DEVICE MANAGEMENT

- Not a replacement for OpenMP, MPI, etc.
- Application can query and select GPUs

  `cudaGetDeviceCount(int *count)`

  `cudaSetDevice(int device)`

  `cudaGetDevice(int *device)`

  `cudaGetDeviceProperties(cudaDeviceProp *prop, int device)`
- Multiple host threads can share a device
- A single host thread can manage multiple devices

  `cudaSetDevice(i)` to select current device

  `cudaMemcpyPeerAsync(...)` for peer-to-peer copies

MULTI-GPU – STREAMS

- Streams (and cudaEvent) have implicit/automatic *device association*
- Each device also has its own unique default stream
- Kernel launches will fail if issued into a stream not associated with current device
- cudaStreamWaitEvent() can synchronize streams belonging to separate devices, cudaEventQuery() can test if an event is "complete"
- Simple device concurrency:

```
cudaSetDevice(0);
cudaStreamCreate(&stream0);        //associated with device 0
cudaSetDevice(1);
cudaStreamCreate(&stream1);        //associated with device 1
Kernel<<<b, t, 0, stream1>>>(…);   // these kernels have the possibility
 cudaSetDevice(0);
Kernel<<<b, t, 0, stream0>>>(…);   // to execute concurrently
```

MULTI-GPU – DEVICE-TO-DEVICE DATA COPYING

- If system topology supports it, data can be copied directly from one device to another over a fabric (PCIE, or NVLink)
- Device must first be explicitly placed into a peer relationship (“clique”)
- Must enable “peering” for both directions of transfer (if needed)
- Thereafter, memory copies between those two devices will not “stage” through a system memory buffer (GPUDirect P2P transfer)

cudaDeviceEnablePeerAccess (int peerDevice, unsigned int flags), peerDevice Peer device to enable direct access to from the current device, flags Reserved for future use and must be set to 0

```
cudaSetDevice(0);
cudaDeviceCanAccessPeer(&canPeer, 0, 1); // test for 0, 1 peerable
cudaDeviceEnablePeerAccess(1, 0);        // device 0 sees device 1 as a “peer”
cudaSetDevice(1);
cudaDeviceEnablePeerAccess(0, 0);        // device 1 sees device 0 as a “peer”
cudaMemcpyPeerAsync(dst_ptr, 0, src_ptr, 1, size, stream0); //dev 1 to dev 0 copy
 cudaDeviceDisablePeerAccess(0);          // dev 0 is no longer a peer of dev 1
```

- Limit to the number of peers in your “clique”

STREAM PRIORITY

- CUDA streams allow an optional definition of a *priority*
- This affects execution of concurrent kernels (only).
- The GPU block scheduler will attempt to schedule blocks from high priority (stream) kernels before blocks from low priority (stream) kernels
- Current CUDA implementation only has 2 priorities
- Current CUDA implementation does not cause preemption of blocks

```
// get the range of stream priorities for this device
 int priority_high, priority_low;
cudaDeviceGetStreamPriorityRange(&priority_low,
&priority_high);
// create streams with highest and lowest available priorities
 cudaStream_t st_high, st_low;
cudaStreamCreateWithPriority(&st_high, cudaStreamNonBlocking, priority_high);
 cudaStreamCreateWithPriority(&st_low, cudaStreamNonBlocking, priority_low);
```

FURTHER STUDY

- Concurrency with Unified Memory:
  - https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/
- Programming Guide:
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#asynchronous-concurrent- execution
- CUDA Sample Codes: concurrentKernels, simpleStreams, asyncAPI, simpleCallbacks, simpleP2P
- Video processing pipeline with callbacks:
  - https://stackoverflow.com/questions/31186926/multithreading-for-image-processing-at-gpu- using-cuda/31188999#31188999