NVIDIA

# CUDA CONCURRENCY

Guillaume Barnier, Solutions Architect, Energy
gbarnier@nvidia.com

# 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

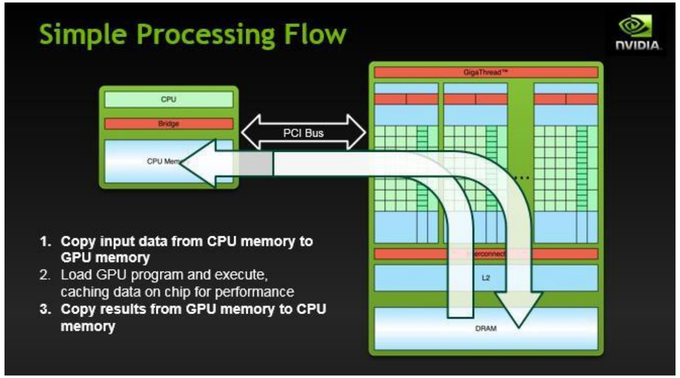
Other concurrency scenarios: kernel concurrency, host/device concurrency

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

# 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:**
  - Kernel launches are asynchronous with CPU
  - cudaMemcpy (D2H, H2D) block CPU thread
  - 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 devices with at least 2 copy engines

- **Stream: sequence of operatations that execute in issue-order on GPU**
  - Operations from different streams may be interleaved
  - A kernel and memcopy from different streams can be overlapped

# Stream Semantics

- **Two operations issued into the same stream will execute in issue-order.**
  - Operation B issued after operation A will not begin to execute until operation A has completed
- **Two operations issued into separate streams have no ordering prescribed by CUDA**
  - Operation A issued into stream 1 may execute before, during, or after operation B issued into stream 2
- **What do we mean by “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**

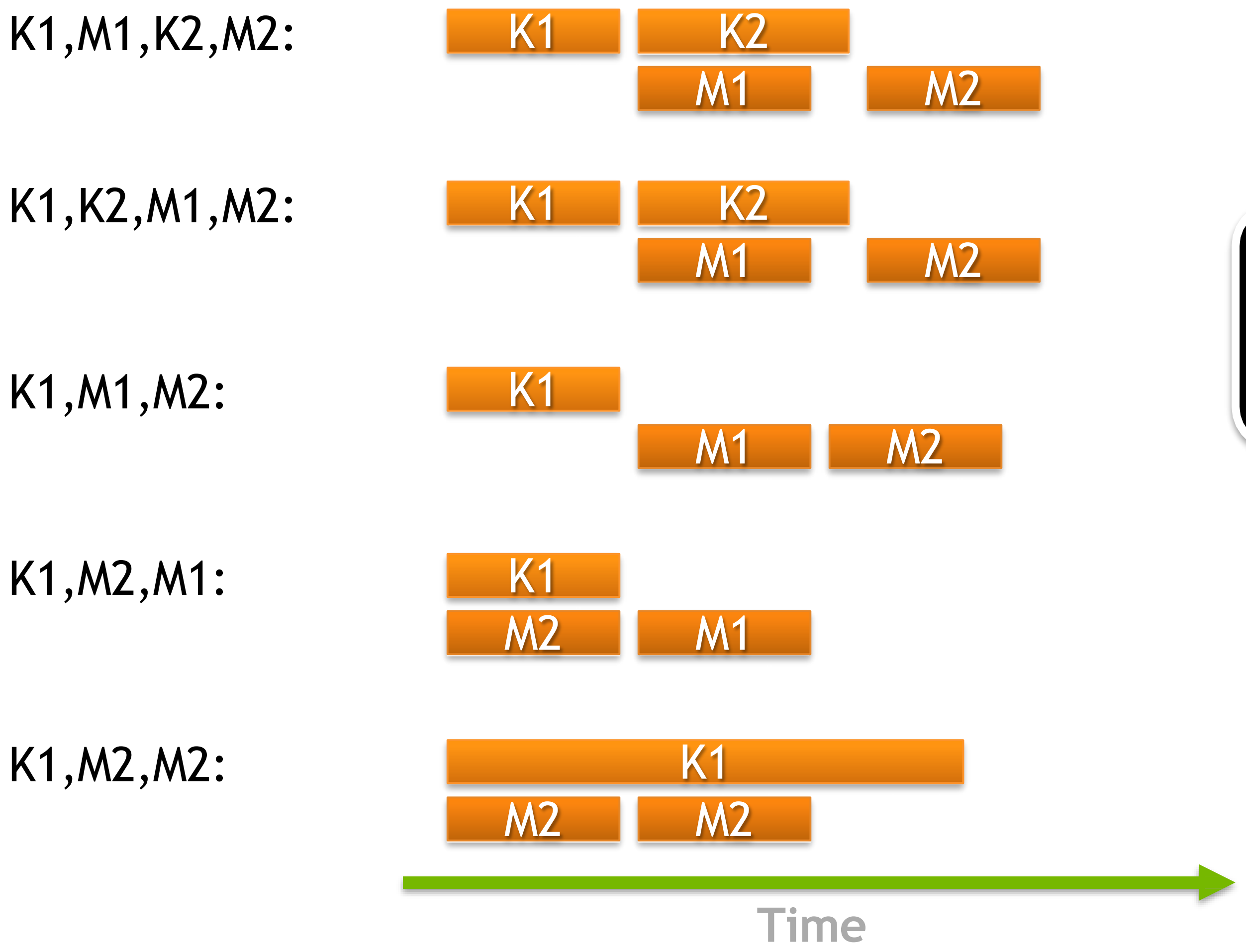
```
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

cudaMemcpyAsync(dst, src, size, dir, stream1);
Kernel<<<grid, block, 0, stream2>>>(…);
```

Potentially overlapped

```
cudaStreamQuery(stream1);       // Check if stream is idle
cudaStreamSynchronize(stream2); // CPU waits until all operations on stream2 are completed
cudaStreamDestroy(stream2);
```

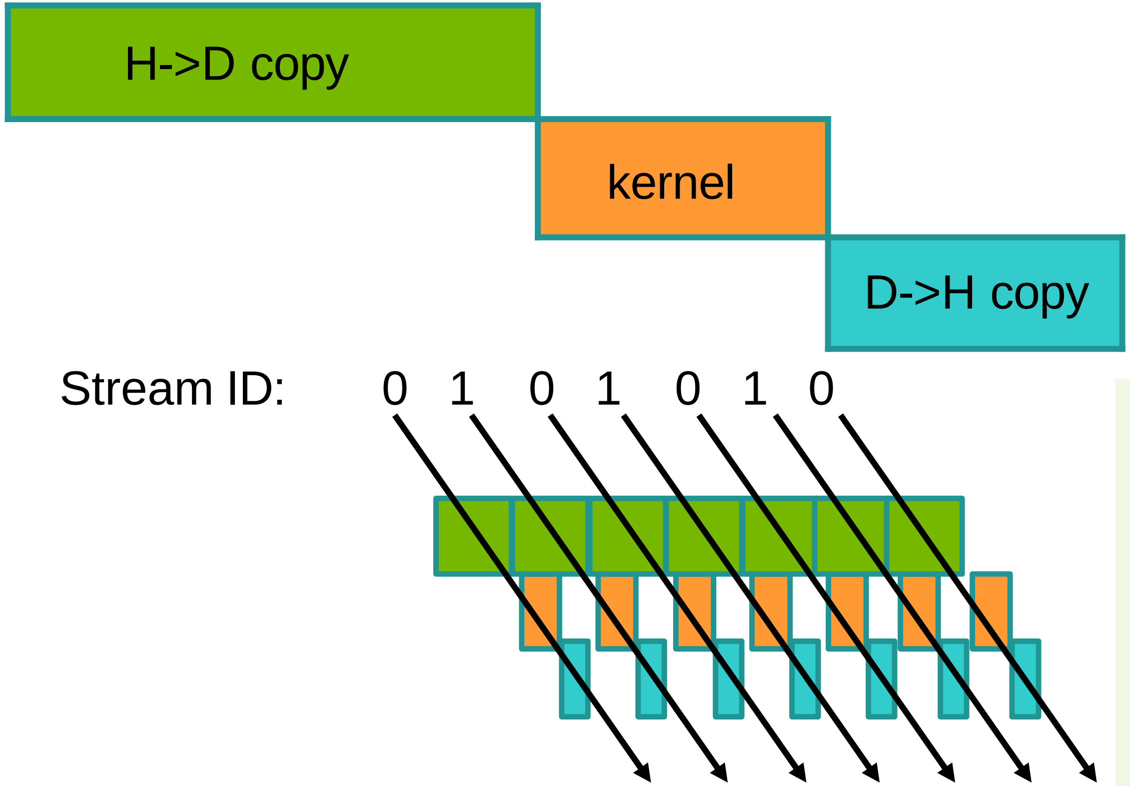
# Stream Examples




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);
```

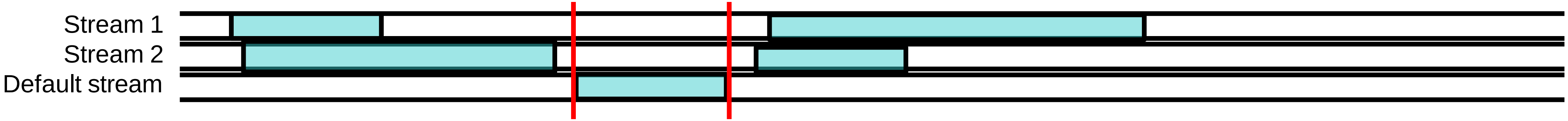
**streamed**

```
for (int i = 0, i<c; i++){
  size_t offx = (size_x/c)*i;
  size_t offy = (size_y/c)*i;
  cudaMemcpyAsync(d_x+offx, h_x+offx,
size_x/c, cudaMemcpyHostToDevice,
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):



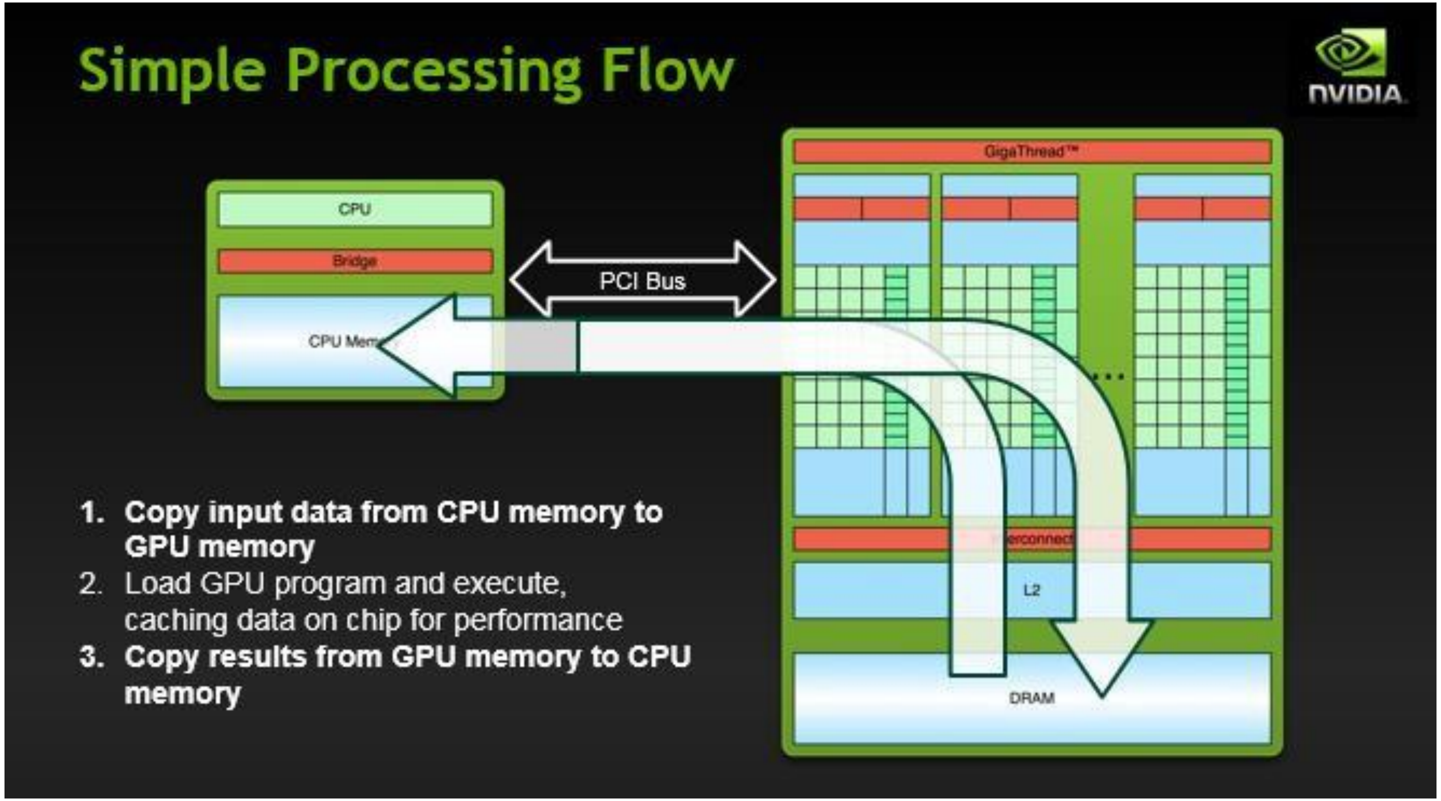

  - All device activity issued prior to the item in the default stream must complete before default stream item begins
  - All device activity issued after the item in the default stream will wait for the default stream item to finish
  - All host threads share the same default stream for legacy behavior
  - Consider avoiding use of default stream during complex concurrency scenarios
- Behavior can be modified to convert it to an “ordinary” stream
  - nvcc --default-stream per-thread …
  - Each host thread will get its own “ordinary” default stream

# Stream callbacks

- Allows definition of a host-code function that will be issued into a CUDA stream
- Follows stream semantics: function will not be called until stream execution reaches that point
- Uses a thread spawned by the GPU driver to perform the callback work
- Has limitations: do not use any CUDA runtime API calls (or kernel launches) in the callback
- Useful for deferring CPU work until GPU results are ready

# Managed Memory

# THE CUDA 3-STEP PROCESSING SEQUENCE

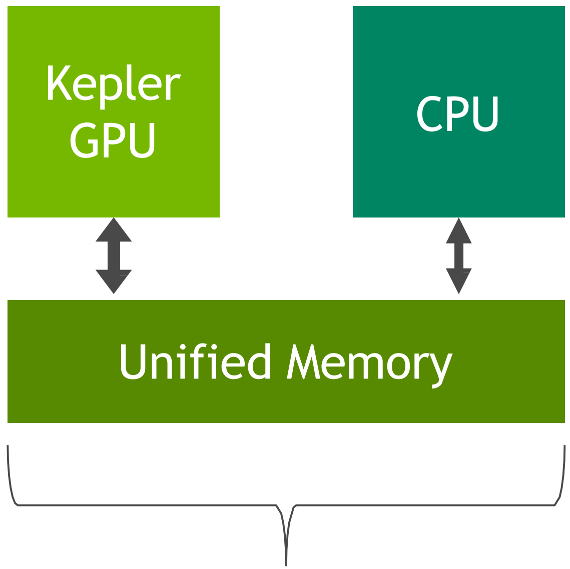



->Wouldn’t it be nice if we didn’t have to do (i.e. write the code for) steps 1 and 3?

# Managed Memory

## Reduce Developer Effort

**CUDA 6+**




Allocate Up To
GPU Memory Size

**Simpler Programming & Memory Model**

- Single allocation, single pointer, accessible anywhere
- Eliminate need for *explicit* copy
- Simplifies code porting

**Maintain Performance through Data Locality**

- Migrate data to accessing processor
- Guarantee global coherence
- Still allows explicit hand tuning

# SIMPLIFIED MEMORY MANAGEMENT CODE

## CPU Code

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

## Ordinary CUDA Code

```
void sortfile(FILE *fp, int N) {
  char *data, *d_data;
  data = (char *)malloc(N);
  cudaMalloc(&d_data, N);
  fread(data, 1, N, fp);
  cudaMemcpy(d_data, data, N, …); // 1
  qsort<<<...>>>(data,N,1,compare); // 2
  cudaMemcpy(data, d_data, N, …); // 3

  use_data(data);
  cudaFree(d_data);
  free(data);
}
```

# SIMPLIFIED MEMORY MANAGEMENT CODE

## CPU Code

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

## CUDA Code with Unified Memory

```
void sortfile(FILE *fp, int N) {
  char *data;
  cudaMallocManaged(&data, N);

  fread(data, 1, N, fp);

  qsort<<<...>>>(data,N,1,compare);
  cudaDeviceSynchronize();

  use_data(data);

  cudaFree(data);
}
```

# Copy-compute overlap with managed memory

## In particular, with demand-paging

- Follow same pattern, except use cudaMemPrefetchAsync() instead of cudaMemcpyAsync()
- Stream semantics will guarantee that any needed migrations are performed in proper order
- However, cudaMemPrefetchAsync() has more work to do than cudaMemcpyAsync() (updating of page tables in CPU and GPU)
- This means the call can take substantially more time to return than an “ordinary” async call – can introduce unexpected gaps in timeline
- Behavior varies for “busy” streams vs. idle streams. Counterintuitively, “busy” streams may result in better throughput

# Aside: cudaEvent

- cudaEvent is an entity that can be placed as a “marker” in a stream
- A cudaEvent is said to be “recorded” when it is issued
- A cudaEvent is said to be “completed” when stream execution reaches the point where it was recorded
- Most common use: timing

```
cudaEvent_t start, stop;          // cudaEvent has its own type
cudaEventCreate(&start);          // cudaEvent must be created
cudaEventCreate(&stop);           // before use
cudaEventRecord(start);           // “recorded” (issued) into default stream
Kernel<<<b, t>>>(…);              // could be any set of CUDA device activity
cudaEventRecord(stop);
cudaEventSynchronize(stop);       // wait for stream execution to reach “stop” event
cudaEventElapsedTime(&float_var, start, stop);    // measure Kernel duration
```

- Also useful for arranging complex concurrency scenarios
- Event-based timing may give unexpected results for host activity or complex concurrency scenarios

# Multi-GPU – Device Management

- **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)

```
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”

# Other concurrency scenarios

- **Host/Device execution concurrency:**

```
Kernel<<<b, t>>>(…);    // this kernel execution can overlap with
cpuFunction(…);         // this host code
```

- **Concurrent kernels:**

```
Kernel<<<b, t, 0, stream0>>>(…);    // these kernels have the possibility
Kernel<<<b, t, 0, stream1>>>(…);    // to execute concurrently
```

  - **In practice, concurrent kernel execution on the same device is hard to witness**
  - **Requires kernels with relatively low resource utilization and relatively long execution time**
  - **There are hardware limits to the number of concurrent kernels per device**
  - **Less efficient than saturating the device with a single kernel**

# 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 implementation only has 2 priorities
- Current 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);
```

# CUDA Graphs (overview)

- New feature in CUDA 10
- Allows for the definition of a sequence of stream(s) work (kernels, memory copy operations, callbacks, host functions, graphs)
- Each work item is a *node* in the graph
- Allows for the definition of *dependencies* (e.g. these 3 nodes must finish before this one can begin)
- Dependencies are effectively graph edges
- Once defined, a graph may be executed by launching it into a stream
- Once defined, a graph may be re-used
- Has both a manual definition method and a “capture” method

# 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

# Stream Examples

K: kernels
M: memcopy
Number: stream ID

Issue order

K1,M1,K2,M2

Execution order