

Introduction to CUDA 3: Synchronization and Streams

**Programming Massively Parallel Multiprocessors and
Heterogeneous Systems (Understanding and programming the
devices powering AI)**

Jonathan Appavoo

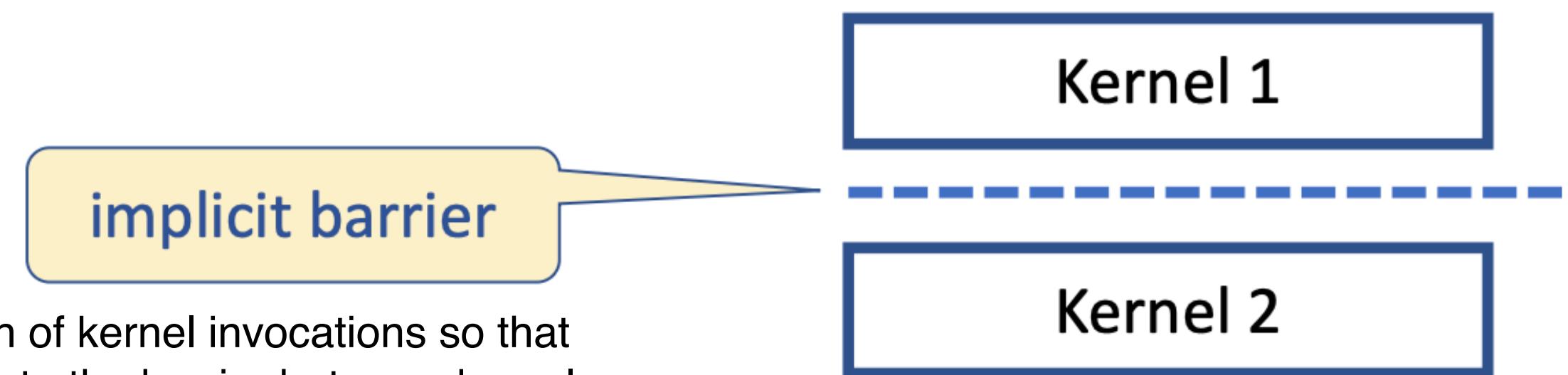
Recall

- GPU hardware organization -- compute capability
- Kernels, threads, warps, blocks, grids
- Kernel invocations, device synchronization
- Occupancy -- calculator and API
- GPU memories, CPU-GPU memory transfers
 - Registers, Shared Memory, Constant Memory, Local Memory and Global Memory
 - Global memory transfer: maximizing coalescing
 - Local memory banked: avoid conflicting access pattern
- Block-level thread barriers: `__syncthreads()`

Recall: `__syncthreads()`

- block-level synchronization barrier
- each thread, when it reaches the statement, blocks until
 - all other threads have reached it as well
 - AND all global and shared memory written by the threads are visible to all threads (includes a memory fence)
- Note: threads in different blocks **can't** synchronize!

Q: What do you do if you want to sync threads across blocks?



NVIDIA now has support for defining a graph of kernel invocations so that you don't have to come back to the host create the barrier between kernels

Aside: "Esoteric" syncthread friends

See CUDA Programming Guide (CPG) for details

<https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/cpp-language-extensions.html#thread-block-synchronization-functions>

```
int __syncthreads_count(int predicate);
```

- returns to all threads of the block the number of threads for which the value passed in was non-zero

```
int __syncthreads_and(int predicate);
```

- returns to all threads of the block non-zero (true) if the add of all values passed in where non-zero (true). eg all passed in 1

```
int __syncthreads_or(int predicate);
```

- returns to all threads of the block non-zero (true) if the or of all values passed in is non-zero (ture). eg. any passed in 1

```
void __syncwarp(unsigned mask=0xffffffff);
```

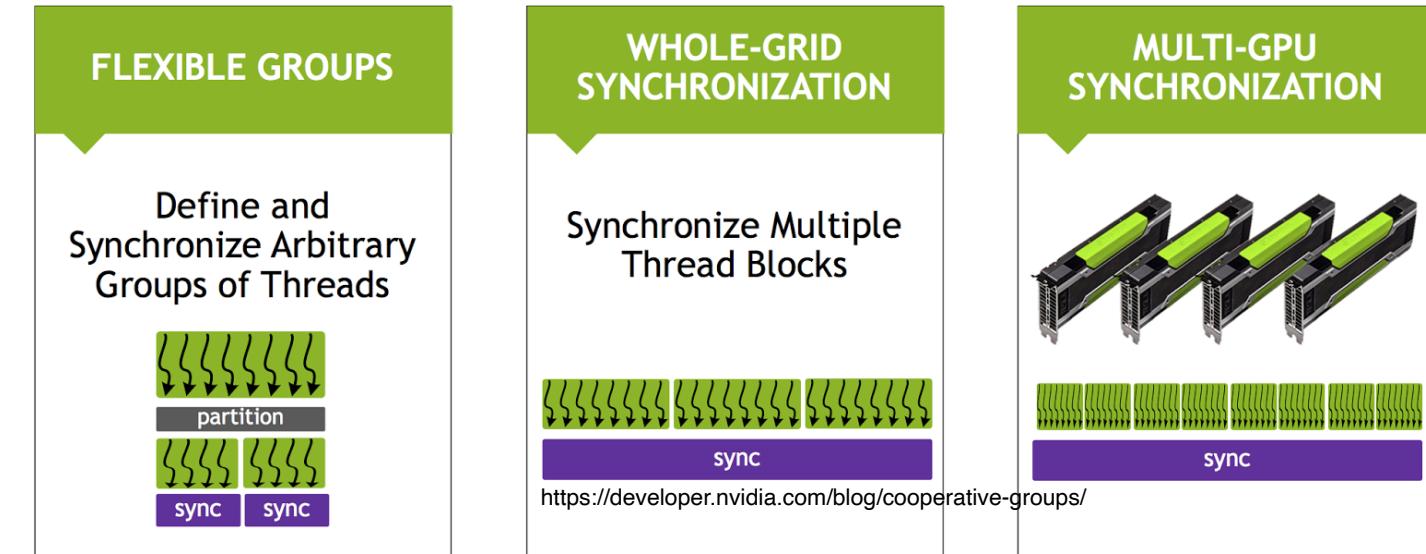
- Like syncthreads but operates on the threads (lanes) of a warp. Can be used with a subset using mask to identify which lanes of the warp are participating

Beyond syncthreads: Co-op Groups and Graphs

1. Cooperative Groups

- flexibly create groups of threads
- allows synchronization within blocks and across blocks
- and even across grids and devices

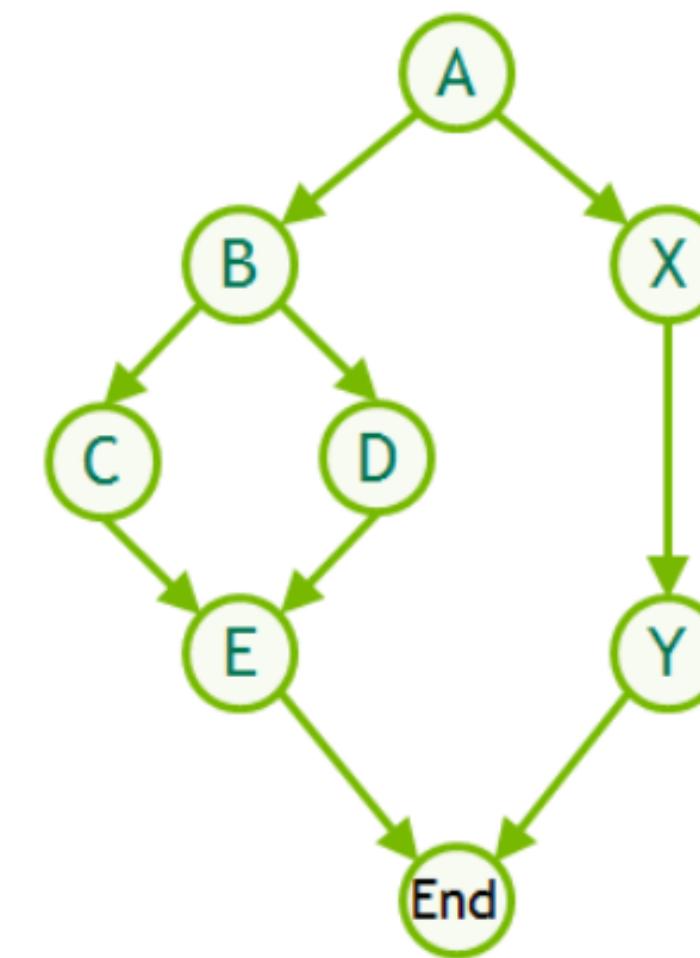
<https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/cooperative-groups.html>



2. CUDA Graphs

- new model for work submission
- a series of kernel launches can be connected by dependencies and handed over
- no need to go back to the CPU
 - eliminate "launch latency" for repeated launch
 - reduce system overheads

<https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/cuda-graphs.html>



<https://developer.nvidia.com/blog/cuda-graphs/>

Atomic Functions

<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/writing-cuda-kernels.html#atomics>

<https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/cpp-language-extensions.html#atomic-functions>

Concurrency Control

Of course we should be creatively trying to eliminate the need (at least on the "hot paths")

Also creative use of sync functions might be better (have to measure)

Atomic Operations: Motivation

Need for Mutexs/Locks

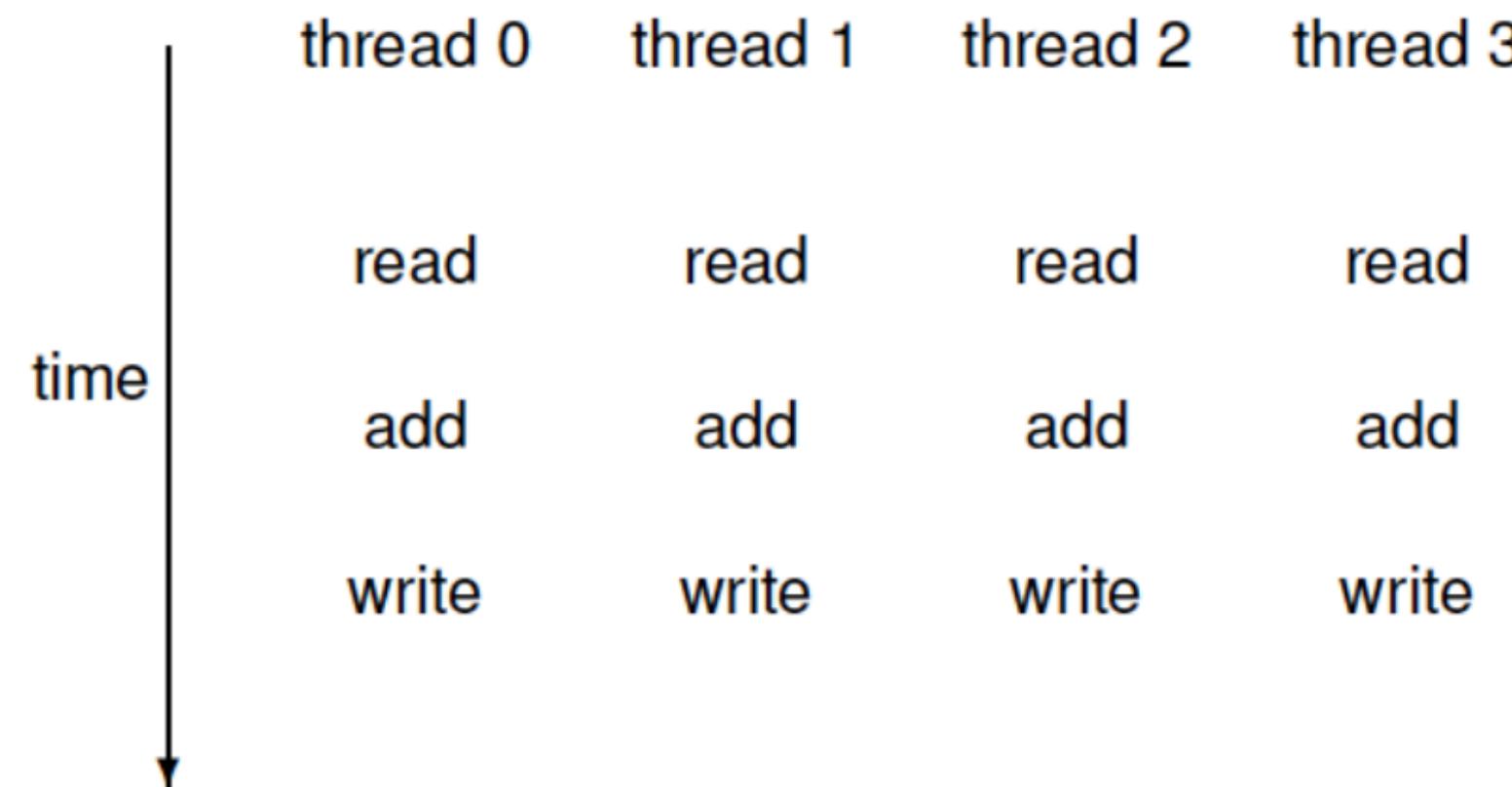
- Need threads to update a shared value (eg. a counter in shared memory or global)

```
__shared__ int count;
```

```
...
```

```
if (...) count++
```

- Problem if two (or more) threads do it at the same time



Atomic Operations: CAS

Compare & Swap : Common hardware provided atomic primitive

```
int atomicCAS(int* address, int compare, int val);
```

<https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/cpp-language-extensions.html#atomiccas>

```
atomically {
    int old = *address; // load copy into register
    if (old == compare) /
        *address = val;
}
return old;
```

"...any atomic operation can be implemented
based on **atomicCAS()**"

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=__syncthreads#atomic-functions

Atomic Operations: CAS

Compare & Swap : eg. Lock/Mutex

```
int atomicCAS(int* address, int compare, int val);
```

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=__syncthreads#atomiccas

```
atomically {
    int old = *address;
    if (old == compare)
        *address = val;
}
return old;
```

"...any atomic operation
based on **atomicCAS**"

```
--device__ void mutex_lock(unsigned int *mutex) {
    unsigned int ns = 8;
    while (atomicCAS(mutex, 0, 1) == 1) {
        __nanosleep(ns);
        if (ns < 256) {
            ns *= 2;
        }
    }
}

--device__ void mutex_unlock(unsigned int *mutex) {
    atomicExch(mutex, 0);
}
```

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=__syncthreads#nanosleep-example

Atomic Operations: CAS

Compare & Swap : eg. Lock/Mutex

```
int atomicCAS(int* address, int compare, int val);
```

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=__syncthreads#atomiccas

```
atomically {
    int old = *address;
    if (old == compare)
        *address = val;
}
return old;
```

"...any atomic operation
based on **atomicCAS**"

```
__device__ void mutex_lock(unsigned int *mutex) {
    unsigned int ns = 8;
    while (atomicCAS(mutex, 0, 1) == 1) {
        __nanosleep(ns);
        if (ns < 256) {
            ns *= 2;
        }
    }
}
```

```
__device__ void mutex_unlock(unsigned int *mutex) {
    atomicExch(mutex, 0);
}
```

This is not a "usefully" correct version!

Requires memory fences
(discussed later)

More useful Atomic Operation provided

Often won't need lock

- Arithmetic Functions
 - atomic[Add|Sub|Exch|Min|Max|Incl|Decl|CAS]()
- Bitwise Functions
 - atomic[And|Or|Xor]()
- Others
 - CUDA >12.8 added some new one that follow GNU atomic built-in function signatures
 - __nv_atomic[load|load_n|store|store_n|thread_fence]
 - Also introduced similar ones for the Arithmetic and Bitwise
 - Above for various word types (int, long long, float, double, etc)

BUT BE CAREFUL: Avoid and when needed try to limit frequency of updating a Global Memory word -- Perf will suck especially under contention

Atomic Operation: Memory order and scope

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=__syncthreads#atomic-functions

- In general assume relaxed memory order "there are no synchronization or ordering constraints imposed on other reads or writes, only this operations atomicity is guaranteed"

https://en.cppreference.com/w/cpp/atomic/memory_order.html

```
enum memory_order
{
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst
};
```

C++ standard does NOT expose heterogenous costs -- CUDA does

Default for CUDA API

- In general atomic functions that take an address can be used with data located in Global or Shared memory

```
namespace cuda {

enum thread_scope {
    thread_scope_system,
    thread_scope_device, →
    thread_scope_block,
    thread_scope_thread
};

} // namespace cuda
```

https://nvidia.github.io/cccl/libcudacxx/extended_api/memory_model.html#thread-scopes

<https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/memory-sync-domains.html#memory-synchronization-domains>

Atomic Operation: Thread Fences

<https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/cpp-language-extensions.html#memory-fence-functions>

CUDA Programming Model assumes device with a weakly-ordered memory model.

1. Order of **writes** to shared memory, global memory, page-locked host memory, and memory of a peer device by **a thread**
2. Is **NOT** necessarily **the order** in which the data is observed being written by another CUDA or host thread

"It is undefined behavior for two threads to read the same memory location without [memory] synchronization"

Atomic Operation: Thread Fences

<https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/cpp-language-extensions.html#memory-fence-functions>

```
__device__ int X = 1, Y = 2;  
  
__device__ void writeXY()  
{  
    X = 10;  
    Y = 20;  
}  
  
__device__ void readXY()  
{  
    int B = Y;  
    int A = X;  
}
```

Assume

1. Thread 1 executes writeXY()
2. Thread 2 executes readXY()

Possible outcomes:

A	1	10	10	1
B	2	2	20	20

Fix:

`__threadfence_block();`

wait until all memory writes are visible to all thread
in **block**

`__threadfence();`

wait until all memory writes are visible to all threads

NOTE: `__syncthreads()` ensures both threads and memory are "synced"

Atomic Operation: Thread Fences

<https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/cpp-language-extensions.html#memory-fence-functions>

Assume

```
__device__ int X = 1, Y = 2;  
  
__device__ void writeXY()  
{  
    X = 10;  
    Y = 20;  
}  
  
__device__ void readXY()  
{  
    int B = Y;  
    int A = X;  
}
```

1. Thread 1 executes writeXY()
2. Thread 2 executes readXY()

Don't forget that you may need to use **__volatile__** to avoid compiler optimizations that can reorder your instructions (eg loads and stores) or optimize them away.

__threadfence_block();

wait until all memory writes are visible to all thread in **block**

__threadfence();

wait until all memory writes are visible to all threads

NOTE: __syncthreads() ensures both threads and memory are "synched"

Warp Synchronous Operations

Threads of a warp are called lanes, with lane ids (0 -- 31)

- **Warp Shuffle Functions**

<https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/cpp-language-extensions.html#warp-shuffle-functions>

- `__shfl_sync, __shfl_[updownlxor]_sync`
 - **synchronous** (all threads of warp must execute)
 - exchange variables/data between threads (lanes) of a warp **without shared memory (using registers)**
 - **no implied memory fence (no memory order)**
 - can implement bcsts, scans, etc.

- **Warp Vote functions**

<https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/cpp-language-extensions.html#warp-vote-functions>

- `__[allanylbballot]_sync, __active_mask`
 - all eval pred true, any one eval true, ballot exactly who was true
- who is active with right now

- **Warp Reduce Functions**

<https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/cpp-language-extensions.html#warp-reduce-functions>

- `__reduce_[addlminlmaxlandlorlxor]_sync`
 - eg. sum a value across threads $\text{sum} = \text{__reduce_add_sync}(0xFFFFFFFF, \text{value})$

- **Warp Match Functions**

<https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/cpp-language-extensions.html#warp-match-functions>

- `__match_[anyllall]_sync`
 - any: returns mask of lanes that have the same value of a variable with respect to the calling lane (who else has my value)
 - all: mask of lanes that have the same value as that of lane 0

Warp Synchronous Operations

Threads of a warp are called lanes, with lane ids (0 -- 31)

- **Warp Shuffle Functions**

- `__shfl_sync`, `__shfl_[updownlxor]_sync`
 - **synchronous** (all threads of warp must execute)
 - exchange variables/data between threads (lanes) of a warp **without shared memory (using registers)**
 - **no implicit synchronization**
 - can implement complex logic

These seem like they could be fun to geek out on.

Consider something as simple as a shared incrementing counter

- **Warp Vote functions**

- `__[all|any|lany|lanyt|anyt]_sync`
 - all eval prob
 - who is active

- **Warp Reduce Functions**

- `__reduce_[add|min|max|land|lor|lxor]_sync`
 - eg. sum a value across threads $\text{sum} = \text{__reduce_add_sync}(0xFFFFFFFF, \text{value})$

- **Warp Match Functions**

- `__match_[any|all]_sync`
 - any: returns mask of lanes that have the same value of a variable with respect to the calling lane (who else has my value)
 - all: mask of lanes that have the same value as that of lane 0

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=__syncthreads#warp-shuffle-functions

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=__syncthreads#warp-match-functions

Asynchrony

Overlapping I/O with other work is a critical strategy

- 1. Use hardware that can do things without tying up computational resources (execution units and registers)**
- 2. Maximize parallel I/O channels -- get them all busy**

Asynchronous Programming Model

<https://docs.nvidia.com/cuda/cuda-programming-guide/03-advanced/advanced-kernel-programming.html#asynchronous-data-copies>

**NVIDIA has been adding more and more support for fine grained
asynchronous operations**

Asynchronous SIMD Programming Model

Provides acceleration to memory operations

1. memcpy_async

- move data asynchronously
- while continuing to compute

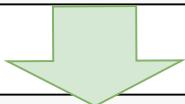
2. builds on memcpy and barrier abstractions (with hw acceleration)

- a copy from src to destination by a pretend help thread ("as-if-thread")
- whose completion can be synchronized with by: cuda::pipeline, cuda::barrier or cooperative_groups::wait
 - I assume there must be some underlying C API as well

3. See Programming Guide for examples and details

<https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/device-callable-apis.html#cooperative-groups-async-h>

```
shared[local_idx] = global_in[global_idx];
```



```
cooperative_groups::memcpy_async(group, shared, global_in + batch_idx, sizeof(int) * block.size());
```

Modern GPU architectures provide multiple hardware mechanisms for asynchronous data movement.

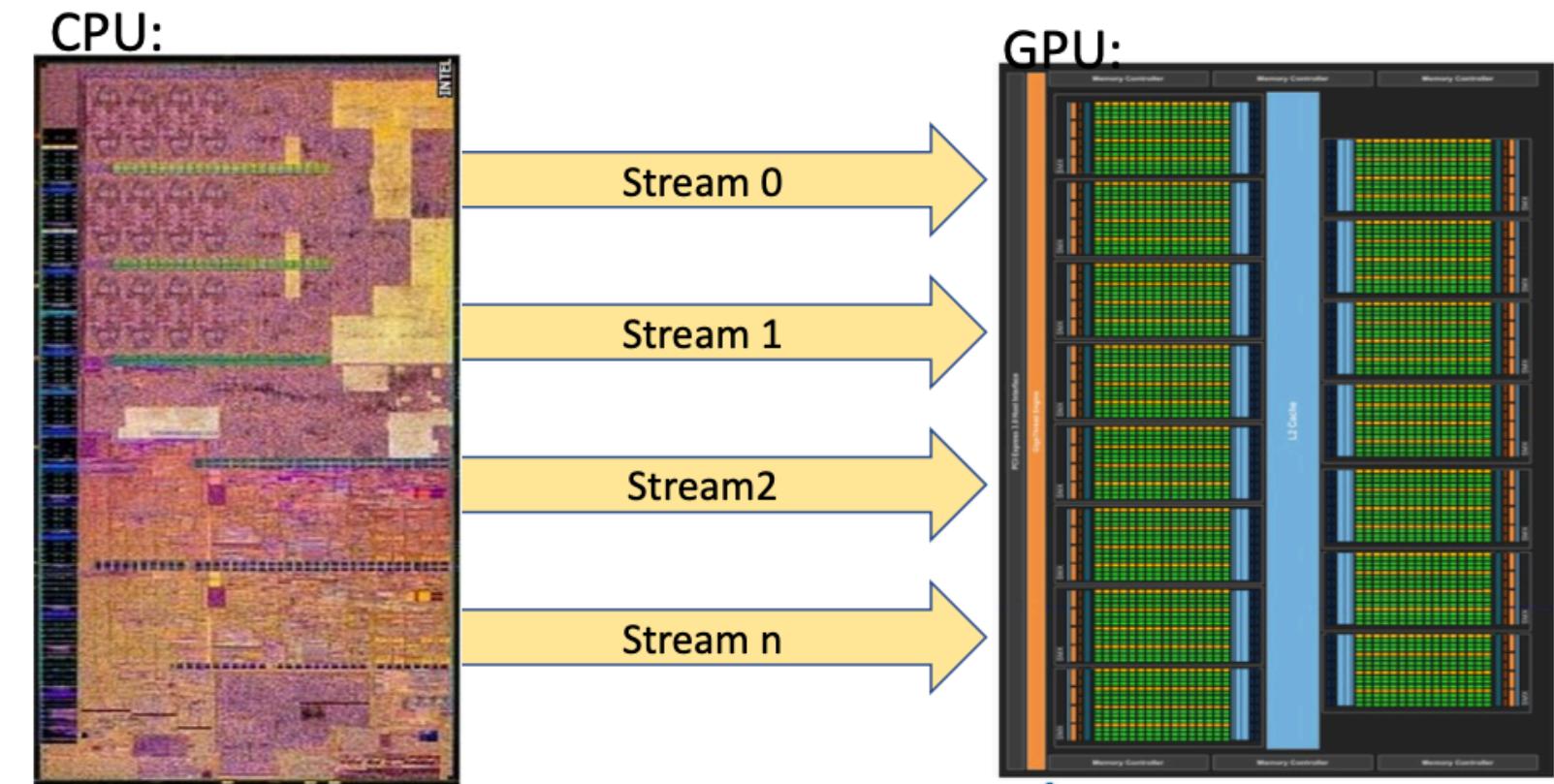
Asynchronous Concurrent Execution (Overlapping I/O and Kernel Execution)

**Coarse grain: Overlapping Host and Device data movement using
multiple CUDA streams**

Streams to launch multiple kernels conc.

<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/asynchronous-execution.html#asynchronous-execution>

- All CUDA operations run in a "stream"
 - executed in order
 - by default: NULL stream, declared implicitly
- For more concurrent operation (eg 2 concurrent kernels) use multiple streams
 - must be declare explicitly
 - handle (pStream) used to identify steam in other calls
- "The actual ability to carry out various operations concurrently will depend on the version of CUDA and the compute capability of the hardware being used"



<https://docs.nvidia.com/cuda/cuda-c-programming-guide/#overlap-of-data-transfer-and-kernel-execution>

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/#overlapping-behavior>

Stream creation

<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/asynchronous-execution.html#creating-and-destroying-cuda-streams>

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
```

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/#creation-and-destruction-of-streams>

- Do not use stream 0 (default)
 - Synchronizing on stream 0 waits until ALL streams completed

Stream: cudaMemcpyAsync

<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/asynchronous-execution.html#launching-memory-transfers-in-cuda-streams>

<https://docs.nvidia.com/cuda/cuda-programming-guide/03-advanced/advanced-kernel-programming.html#asynchronous-data-copies>

```
cudaError_t cudaMemcpyAsync( to, from,  
                           {h2d/d2h}, stream )
```

- returns immediately host side
- careful:
 - error returned may be from an earlier call
 - host memory being accessed must be pinned

Stream use

```
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                   size, cudaMemcpyHostToDevice, stream[i]);
    MyKernel <<<100, 512, 0, stream[i]>>>
        (outputDevPtr + i * size, inputDevPtr + i * size, size);
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                   size, cudaMemcpyDeviceToHost, stream[i]);
}
```

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/#creation-and-destruction-of-streams>

- Queue up operations to schedule for device to schedule
- cudaMemcpyAsync(...)
 - returns immediately
 - error maybe from earlier call
 - host memory must be pagelocked (pinned)
- Above may not result in maximum overlap (see later)

Stream destruction

```
for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
```

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/#creation-and-destruction-of-streams>

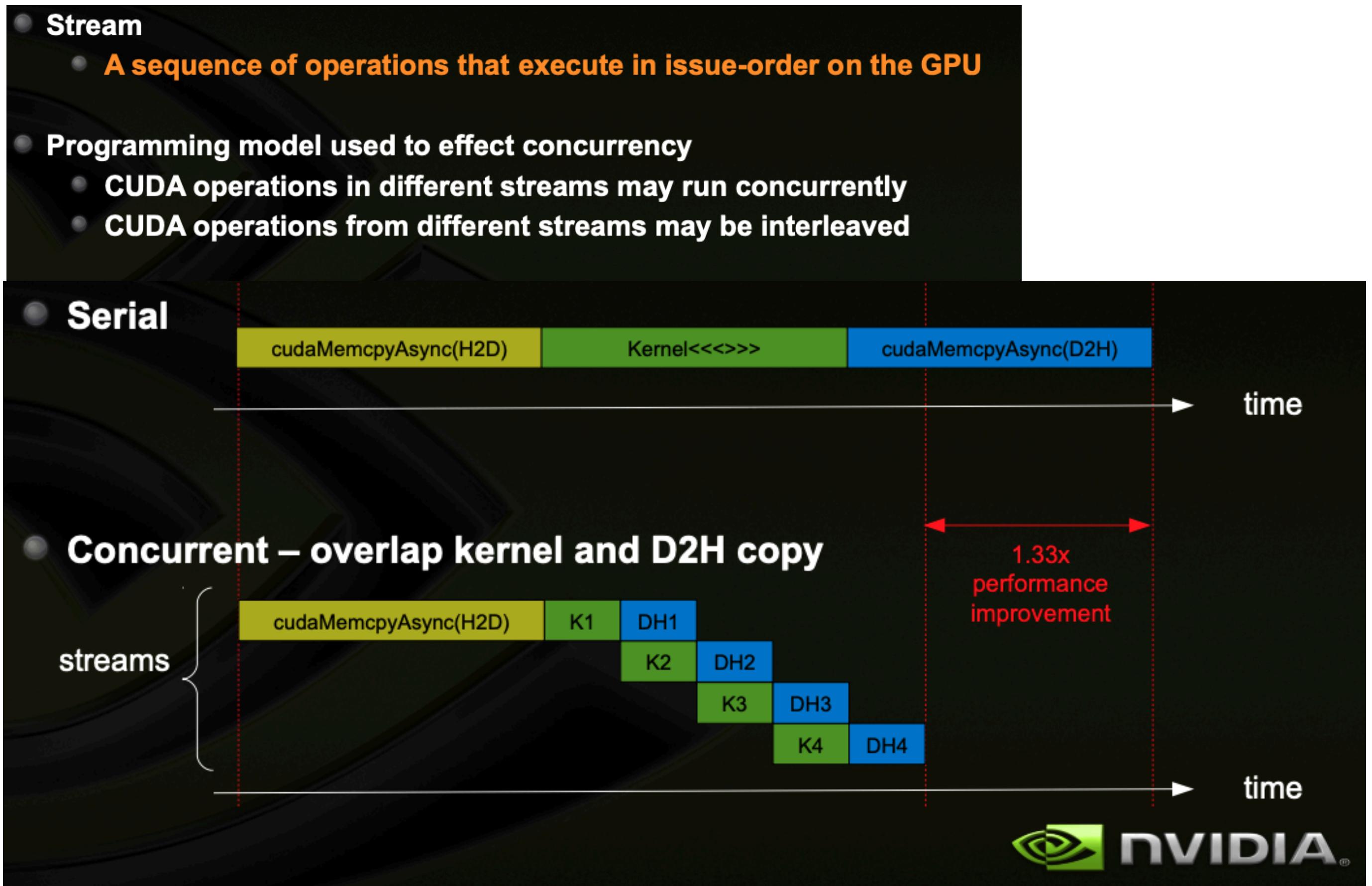
- Async
- if called before stream is complete and resources will be released when stream on device is complete

Stream Concurrency example

Serial Execution:



Stream Concurrency example



Stream Concurrency example



Stream concurrency requirements

- CUDA operations must be in different, non-zero, streams
- cudaMemcpyAsync with host 'pinned' memory
 - Page-locked memory
 - cudaHostMalloc() or cudaHostAlloc()
- Sufficient resources must be available
 - cudaMemcpyAsyncs in different directions
 - device resources (SMEM, registers, blocks, etc)

Stream concurrency requirements

- CUDA operations must be in different, non-zero, streams
 - cudaMemcpyAsync with host 'pinned' memory
 - Page-locked memory
 - cudaHostMalloc() or cudaHostAlloc()
 - Sufficient resources must be available
 - cudaMemcpyAsyncs in different directions
 - device resources (SMEM, registers, blocks, etc)
- Enough DMA Engines
for concurrent I/O
- Enough compute resources for
concurrent kernel execution

Stream: Overlap of Data Transfers and Kernel Execution

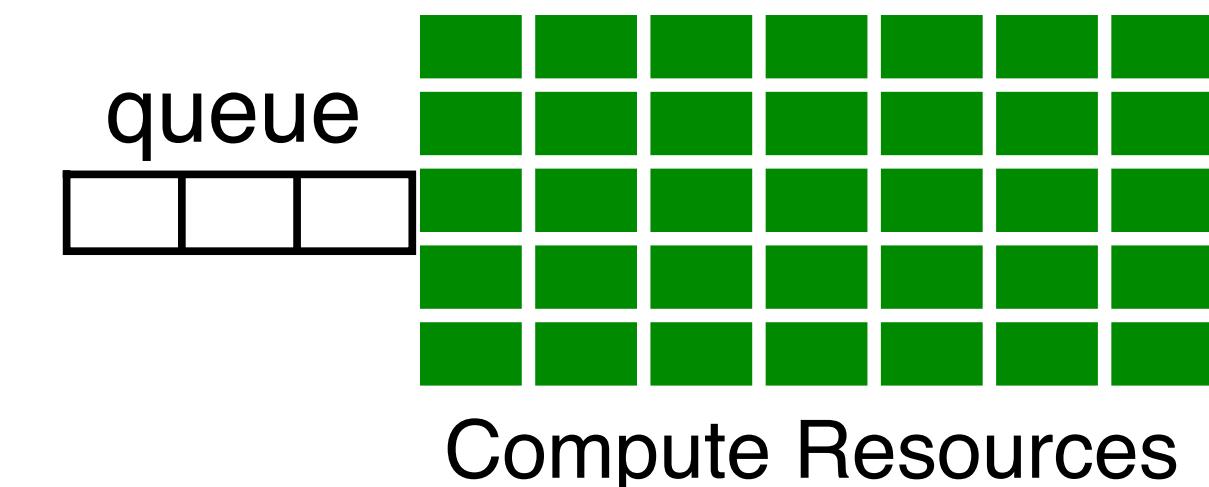
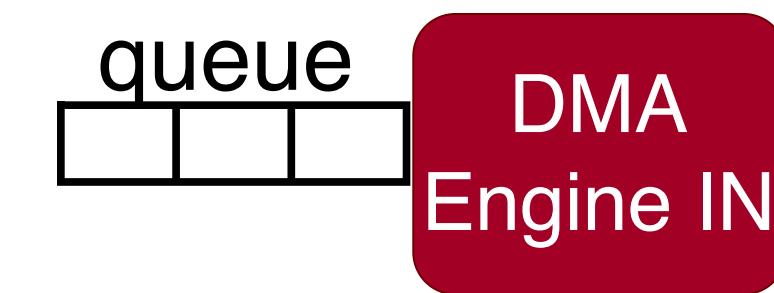
<https://docs.nvidia.com/cuda/cuda-c-programming-guide/#overlapping-behavior>

```
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                   size, cudaMemcpyHostToDevice, stream[i]);
for (int i = 0; i < 2; ++i)
    MyKernel<<<100, 512, 0, stream[i]>>>
        (outputDevPtr + i * size, inputDevPtr + i * size, size);
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                   size, cudaMemcpyDeviceToHost, stream[i]);
```

- "amount of execution overlap between two streams depends on the order in which the commands are issued to each stream"

Stream: Overlap of Data Transfers and Kernel Execution

- Beware of Head-of-line (HOL) blocking
 - Resource Contention/Camping
 - DMA engines
 - Or compute resources



Event Streams

<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/asynchronous-execution.html#cuda-events>

```
cudaEvent_t event ;  
cudaEventCreate( &event ) ;
```

```
cudaEventRecord( event, stream[i] ) ;
```

Like an operation added to the stream that sets a flag host-side when it reaches head of work queue GPU-side

```
cudaStreamWaitEvent( event ) ;
```

```
cudaQueryEvent( event ) ;
```

- Marker in a stream
 - synchronize stream execution
 - monitor device progress
 - Useful for synchronizing concurrent streams

Blocks until
event occurs

CUDA_SUCCESS if
event occurred

Stream: Host functions (Callbacks)

<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/asynchronous-execution.html#callback-functions-from-streams>

```
void CUDART_CB MyCallback(void *data){  
    printf("Inside callback %d\n", (size_t)data);  
}  
...  
for (size_t i = 0; i < 2; ++i) {  
    cudaMemcpyAsync(devPtrIn[i], hostPtr[i], size, cudaMemcpyHostToDevice, stream[i]);  
    MyKernel<<<100, 512, 0, stream[i]>>>(devPtrOut[i], devPtrIn[i], size);  
    cudaMemcpyAsync(hostPtr[i], devPtrOut[i], size, cudaMemcpyDeviceToHost, stream[i]);  
    cudaLaunchHostFunc(stream[i], MyCallback, (void*)i);  
}
```

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/#host-functions-callbacks>

- Callback occurs after all previously queue operations completed
- Restrictions
 - No CUDA function can be in call back: directly or indirectly

Streams and Concurrency

- Be aware of the issue order
- Default stream (0) serializes everything (note it seems like this behavior is now configurable) <https://docs.nvidia.com/cuda/cuda-runtime-api/stream-sync-behavior.html#stream-sync-behavior>
- Use profilers to explore gaps and if the overlap is working

Management operations on streams:

- **cudaStreamDestroy** https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html#group__CUDART__STREAM_1gfda584f1788ca983cb21c5f4d2033a62
- **cudaStreamSynchronize** https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html#group__CUDART__STREAM_1g82b5784f674c17c6df64affe618bf45e
- **cudaStreamWaitEvent** https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html#group__CUDART__STREAM_1g7840e3984799941a61839de40413d1d9
- **cudaStreamQuery** https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html#group__CUDART__STREAM_1g2021adeb17905c7ec2a3c1bf125c5435
 - async check if completed
- **priorities**
 - **cudaDeviceGetStreamPriorityRange** https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__DEVICE.html#group__CUDART__DEVICE_1gfdb79818f7c0ee7bc585648c91770275
 - **cudaStreamCreateWithPriority** https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html#group__CUDART__STREAM_1ge2be9e9858849bf62ba4a8b66d1c3540

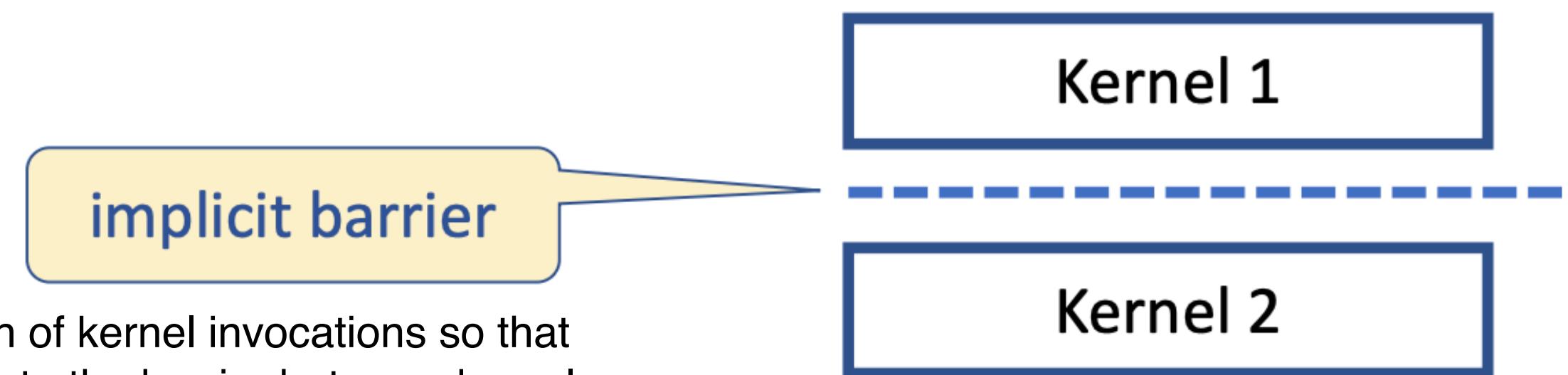
Recall

- GPU hardware organization -- compute capability
- Kernels, threads, warps, blocks, grids
- Kernel invocations, device synchronization
- Occupancy -- calculator and API
- GPU memories, CPU-GPU memory transfers
 - Registers, Shared Memory, Constant Memory, Local Memory and Global Memory
 - Global memory transfer: maximizing coalescing
 - Local memory banked: avoid conflicting access pattern
- Block-level thread barriers: `__syncthreads()`

Recall: `__syncthreads()`

- block-level synchronization barrier
- each thread, when it reaches the statement, blocks until
 - all other threads have reached it as well
 - AND all global and shared memory written by the threads are visible to all threads (includes a memory fence)
- Note: threads in different blocks **can't** synchronize!

Q: What do you do if you want to sync threads across blocks?



NVIDIA now has support for defining a graph of kernel invocations so that you don't have to come back to the host create the barrier between kernels

Aside: "Esoteric" syncthread friends

See CUDA Programming Guide (CPG) for details

<https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/cpp-language-extensions.html#thread-block-synchronization-functions>

```
int __syncthreads_count(int predicate);
```

- returns to all threads of the block the number of threads for which the value passed in was non-zero

```
int __syncthreads_and(int predicate);
```

- returns to all threads of the block non-zero (true) if the add of all values passed in where non-zero (true). eg all passed in 1

```
int __syncthreads_or(int predicate);
```

- returns to all threads of the block non-zero (true) if the or of all values passed in is non-zero (ture). eg. any passed in 1

```
void __syncwarp(unsigned mask=0xffffffff);
```

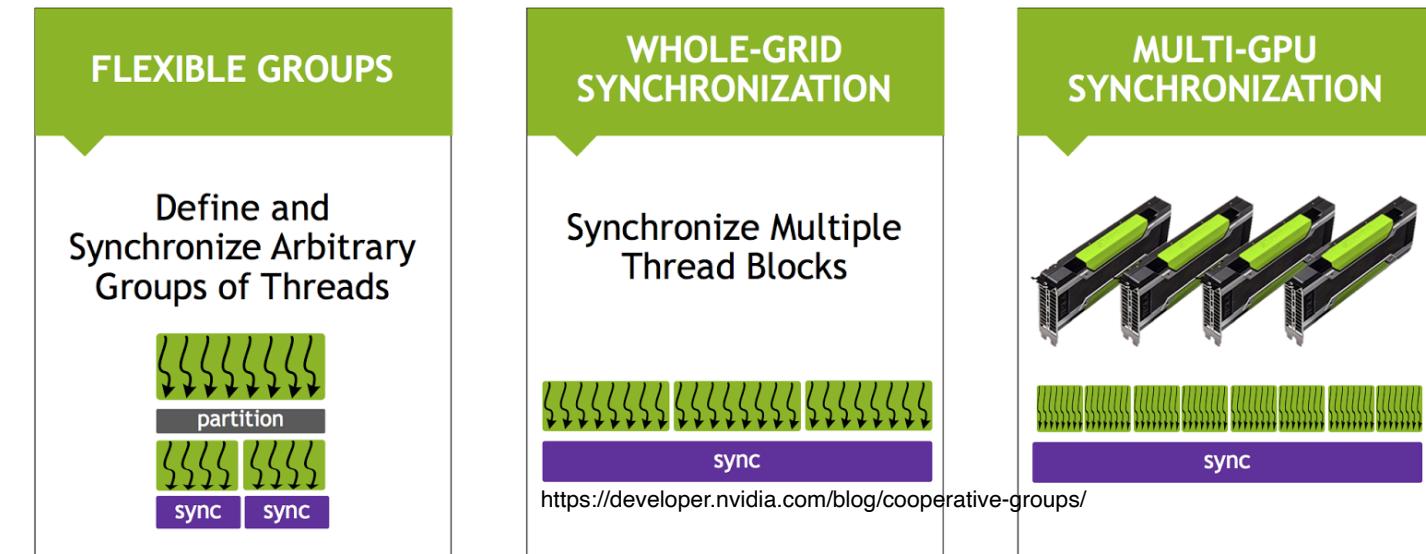
- Like syncthreads but operates on the threads (lanes) of a warp. Can be used with a subset using mask to identify which lanes of the warp are participating

Beyond syncthreads: Co-op Groups and Graphs

1. Cooperative Groups

- flexibly create groups of threads
- allows synchronization within blocks and across blocks
- and even across grids and devices

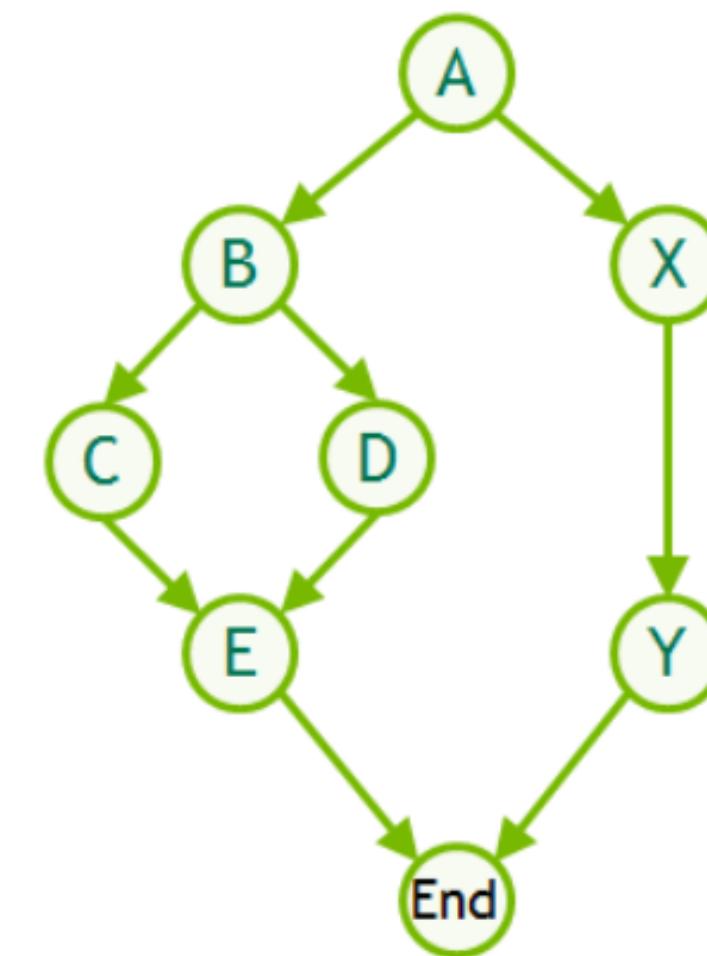
<https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/cooperative-groups.html>



2. CUDA Graphs

- new model for work submission
- a series of kernel launches can be connected by dependencies and handed over
- no need to go back to the CPU
 - eliminate "launch latency" for repeated launch
 - reduce system overheads

<https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/cuda-graphs.html>



<https://developer.nvidia.com/blog/cuda-graphs/>

Atomic Functions

<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/writing-cuda-kernels.html#atomics>

<https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/cpp-language-extensions.html#atomic-functions>

Concurrency Control

Of course we should be creatively trying to eliminate the need (at least on the "hot paths")

Also creative use of sync functions might be better (have to measure)

Atomic Operations: Motivation

Need for Mutexs/Locks

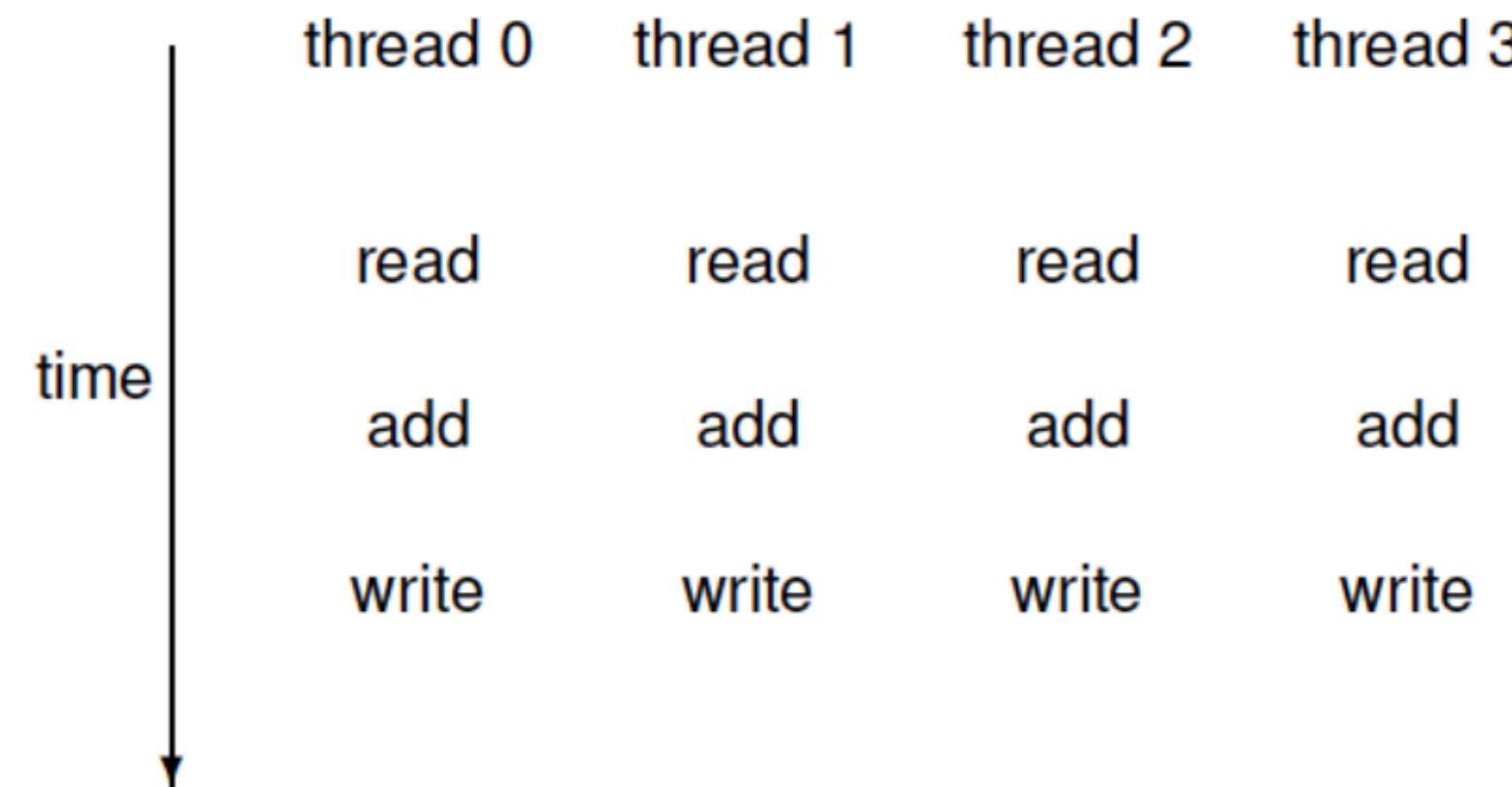
- Need threads to update a shared value (eg. a counter in shared memory or global)

```
__shared__ int count;
```

```
...
```

```
if (...) count++
```

- Problem if two (or more) threads do it at the same time



Atomic Operations: CAS

Compare & Swap : Common hardware provided atomic primitive

```
int atomicCAS(int* address, int compare, int val);
```

<https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/cpp-language-extensions.html#atomiccas>

```
atomically {
    int old = *address; // load copy into register
    if (old == compare) /
        *address = val;
}
return old;
```

"...any atomic operation can be implemented
based on **atomicCAS()**"

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=__syncthreads#atomic-functions

Atomic Operations: CAS

Compare & Swap : eg. Lock/Mutex

```
int atomicCAS(int* address, int compare, int val);
```

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=__syncthreads#atomiccas

```
atomically {
    int old = *address;
    if (old == compare)
        *address = val
}
return old;
```

"...any atomic operation
based on **atomicCAS**"

```
--device__ void mutex_lock(unsigned int *mutex) {
    unsigned int ns = 8;
    while (atomicCAS(mutex, 0, 1) == 1) {
        __nanosleep(ns);
        if (ns < 256) {
            ns *= 2;
        }
    }
}

--device__ void mutex_unlock(unsigned int *mutex) {
    atomicExch(mutex, 0);
}
```

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=__syncthreads#nanosleep-example

Atomic Operations: CAS

Compare & Swap : eg. Lock/Mutex

```
int atomicCAS(int* address, int compare, int val);
```

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=__syncthreads#atomiccas

```
atomically {
    int old = *address;
    if (old == compare)
        *address = val;
}
return old;
```

"...any atomic operation
based on **atomicCAS**"

```
__device__ void mutex_lock(unsigned int *mutex) {
    unsigned int ns = 8;
    while (atomicCAS(mutex, 0, 1) == 1) {
        __nanosleep(ns);
        if (ns < 256) {
            ns *= 2;
        }
    }
}
```

```
__device__ void mutex_unlock(unsigned int *mutex) {
    atomicExch(mutex, 0);
}
```

This is not a "usefully" correct version!

Requires memory fences
(discussed later)

More useful Atomic Operation provided

Often won't need lock

- Arithmetic Functions
 - atomic[Add|Sub|Exch|Min|Max|Incl|Decl|CAS]()
- Bitwise Functions
 - atomic[And|Or|Xor]()
- Others
 - CUDA >12.8 added some new one that follow GNU atomic built-in function signatures
 - __nv_atomic[load|load_n|store|store_n|thread_fence]
 - Also introduced similar ones for the Arithmetic and Bitwise
 - Above for various word types (int, long long, float, double, etc)

BUT BE CAREFUL: Avoid and when needed try to limit frequency of updating a Global Memory word -- Perf will suck especially under contention

Atomic Operation: Memory order and scope

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=__syncthreads#atomic-functions

- In general assume relaxed memory order "there are no synchronization or ordering constraints imposed on other reads or writes, only this operations atomicity is guaranteed"

https://en.cppreference.com/w/cpp/atomic/memory_order.html

```
enum memory_order
{
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst
};
```

C++ standard does NOT expose heterogenous costs -- CUDA does

Default for CUDA API

- In general atomic functions that take an address can be used with data located in Global or Shared memory

```
namespace cuda {

enum thread_scope {
    thread_scope_system,
    thread_scope_device, →
    thread_scope_block,
    thread_scope_thread
};

} // namespace cuda
```

https://nvidia.github.io/cccl/libcudacxx/extended_api/memory_model.html#thread-scopes

<https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/memory-sync-domains.html#memory-synchronization-domains>

Atomic Operation: Thread Fences

<https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/cpp-language-extensions.html#memory-fence-functions>

CUDA Programming Model assumes device with a weakly-ordered memory model.

1. Order of **writes** to shared memory, global memory, page-locked host memory, and memory of a peer device by **a thread**
2. Is **NOT** necessarily **the order** in which the data is observed being written by another CUDA or host thread

"It is undefined behavior for two threads to read the same memory location without [memory] synchronization"

Atomic Operation: Thread Fences

<https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/cpp-language-extensions.html#memory-fence-functions>

```
__device__ int X = 1, Y = 2;  
  
__device__ void writeXY()  
{  
    X = 10;  
    Y = 20;  
}  
  
__device__ void readXY()  
{  
    int B = Y;  
    int A = X;  
}
```

Assume

1. Thread 1 executes writeXY()
2. Thread 2 executes readXY()

Possible outcomes:

A	1	10	10	1
B	2	2	20	20

Fix:

`__threadfence_block();`

wait until all memory writes are visible to all thread
in **block**

`__threadfence();`

wait until all memory writes are visible to all threads

NOTE: `__syncthreads()` ensures both threads and memory are "synced"

Atomic Operation: Thread Fences

<https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/cpp-language-extensions.html#memory-fence-functions>

Assume

```
__device__ int X = 1, Y = 2;  
  
__device__ void writeXY()  
{  
    X = 10;  
    Y = 20;  
}  
  
__device__ void readXY()  
{  
    int B = Y;  
    int A = X;  
}
```

1. Thread 1 executes writeXY()
2. Thread 2 executes readXY()

Don't forget that you may need to use **__volatile__** to avoid compiler optimizations that can reorder your instructions (eg loads and stores) or optimize them away.

__threadfence_block();

wait until all memory writes are visible to all thread in **block**

__threadfence();

wait until all memory writes are visible to all threads

NOTE: __syncthreads() ensures both threads and memory are "synched"

Warp Synchronous Operations

Threads of a warp are called lanes, with lane ids (0 -- 31)

- **Warp Shuffle Functions**

<https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/cpp-language-extensions.html#warp-shuffle-functions>

- `__shfl_sync, __shfl_[updownlxor]_sync`
 - **synchronous** (all threads of warp must execute)
 - exchange variables/data between threads (lanes) of a warp **without shared memory (using registers)**
 - **no implied memory fence (no memory order)**
 - can implement bcsts, scans, etc.

- **Warp Vote functions**

<https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/cpp-language-extensions.html#warp-vote-functions>

- `__[allanylbballot]_sync, __active_mask`
 - all eval pred true, any one eval true, ballot exactly who was true
- who is active with right now

- **Warp Reduce Functions**

<https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/cpp-language-extensions.html#warp-reduce-functions>

- `__reduce_[addlminlmaxlandlorlxor]_sync`
 - eg. sum a value across threads $\text{sum} = \text{__reduce_add_sync}(0xFFFFFFFF, \text{value})$

- **Warp Match Functions**

<https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/cpp-language-extensions.html#warp-match-functions>

- `__match_[anyllall]_sync`
 - any: returns mask of lanes that have the same value of a variable with respect to the calling lane (who else has my value)
 - all: mask of lanes that have the same value as that of lane 0

Warp Synchronous Operations

Threads of a warp are called lanes, with lane ids (0 -- 31)

- **Warp Shuffle Functions**

- `__shfl_sync`, `__shfl_[updownlxor]_sync`
 - **synchronous** (all threads of warp must execute)
 - exchange variables/data between threads (lanes) of a warp **without shared memory (using registers)**
 - **no implicit synchronization**
 - can implement complex logic

These seem like they could be fun to geek out on.

Consider something as simple as a shared incrementing counter

- **Warp Vote functions**

- `__[all|any|lany|lanyt|anyt]_sync`
 - all eval prob
 - who is active

- **Warp Reduce Functions**

- `__reduce_[add|min|max|land|lor|lxor]_sync`
 - eg. sum a value across threads $\text{sum} = \text{__reduce_add_sync}(0xFFFFFFFF, \text{value})$

- **Warp Match Functions**

- `__match_[any|all]_sync`
 - any: returns mask of lanes that have the same value of a variable with respect to the calling lane (who else has my value)
 - all: mask of lanes that have the same value as that of lane 0

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=__syncthreads#warp-shuffle-functions

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=__syncthreads#warp-match-functions

Asynchrony

Overlapping I/O with other work is a critical strategy

- 1. Use hardware that can do things without tying up computational resources (execution units and registers)**
- 2. Maximize parallel I/O channels -- get them all busy**

Asynchronous Programming Model

<https://docs.nvidia.com/cuda/cuda-programming-guide/03-advanced/advanced-kernel-programming.html#asynchronous-data-copies>

**NVIDIA has been adding more and more support for fine grained
asynchronous operations**

Asynchronous SIMD Programming Model

Provides acceleration to memory operations

1. memcpy_async

- move data asynchronously
- while continuing to compute

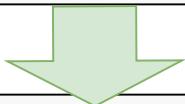
2. builds on memcpy and barrier abstractions (with hw acceleration)

- a copy from src to destination by a pretend help thread ("as-if-thread")
- whose completion can be synchronized with by: cuda::pipeline, cuda::barrier or cooperative_groups::wait
 - I assume there must be some underlying C API as well

3. See Programming Guide for examples and details

<https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/device-callable-apis.html#cooperative-groups-async-h>

```
shared[local_idx] = global_in[global_idx];
```



```
cooperative_groups::memcpy_async(group, shared, global_in + batch_idx, sizeof(int) * block.size());
```

Modern GPU architectures provide multiple hardware mechanisms for asynchronous data movement.

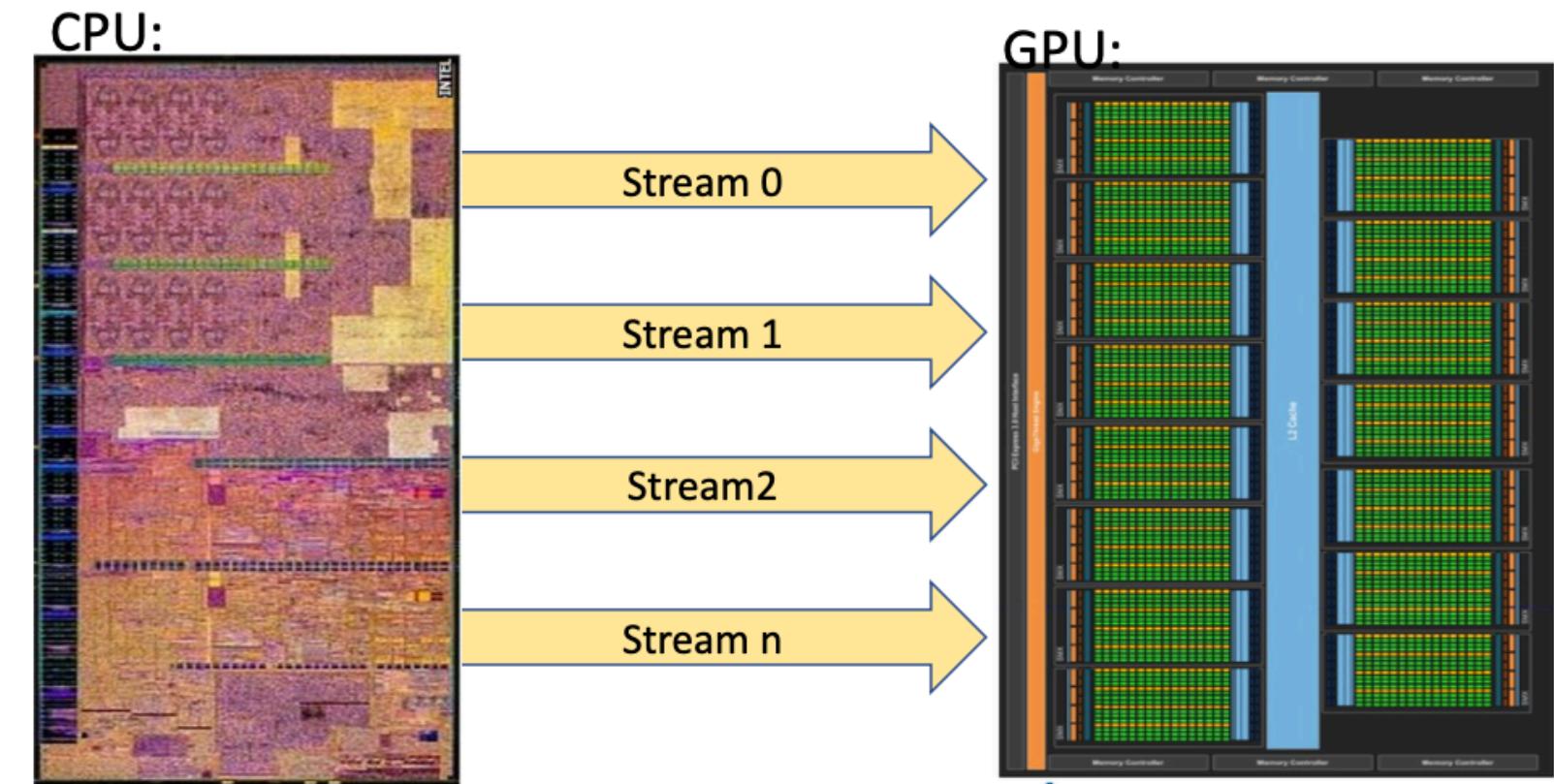
Asynchronous Concurrent Execution (Overlapping I/O and Kernel Execution)

**Coarse grain: Overlapping Host and Device data movement using
multiple CUDA streams**

Streams to launch multiple kernels conc.

<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/asynchronous-execution.html#asynchronous-execution>

- All CUDA operations run in a "stream"
 - executed in order
 - by default: NULL stream, declared implicitly
- For more concurrent operation (eg 2 concurrent kernels) use multiple streams
 - must be declare explicitly
 - handle (pStream) used to identify steam in other calls
- "The actual ability to carry out various operations concurrently will depend on the version of CUDA and the compute capability of the hardware being used"



<https://docs.nvidia.com/cuda/cuda-c-programming-guide/#overlap-of-data-transfer-and-kernel-execution>

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/#overlapping-behavior>

Stream creation

<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/asynchronous-execution.html#creating-and-destroying-cuda-streams>

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
```

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/#creation-and-destruction-of-streams>

- Do not use stream 0 (default)
 - Synchronizing on stream 0 waits until ALL streams completed

Stream: cudaMemcpyAsync

<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/asynchronous-execution.html#launching-memory-transfers-in-cuda-streams>

<https://docs.nvidia.com/cuda/cuda-programming-guide/03-advanced/advanced-kernel-programming.html#asynchronous-data-copies>

```
cudaError_t cudaMemcpyAsync( to, from,  
                           {h2d/d2h}, stream )
```

- returns immediately host side
- careful:
 - error returned may be from an earlier call
 - host memory being accessed must be pinned

Stream use

```
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                   size, cudaMemcpyHostToDevice, stream[i]);
    MyKernel <<<100, 512, 0, stream[i]>>>
        (outputDevPtr + i * size, inputDevPtr + i * size, size);
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                   size, cudaMemcpyDeviceToHost, stream[i]);
}
```

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/#creation-and-destruction-of-streams>

- Queue up operations to schedule for device to schedule
- cudaMemcpyAsync(...)
 - returns immediately
 - error maybe from earlier call
 - host memory must be pagelocked (pinned)
- Above may not result in maximum overlap (see later)

Stream destruction

```
for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
```

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/#creation-and-destruction-of-streams>

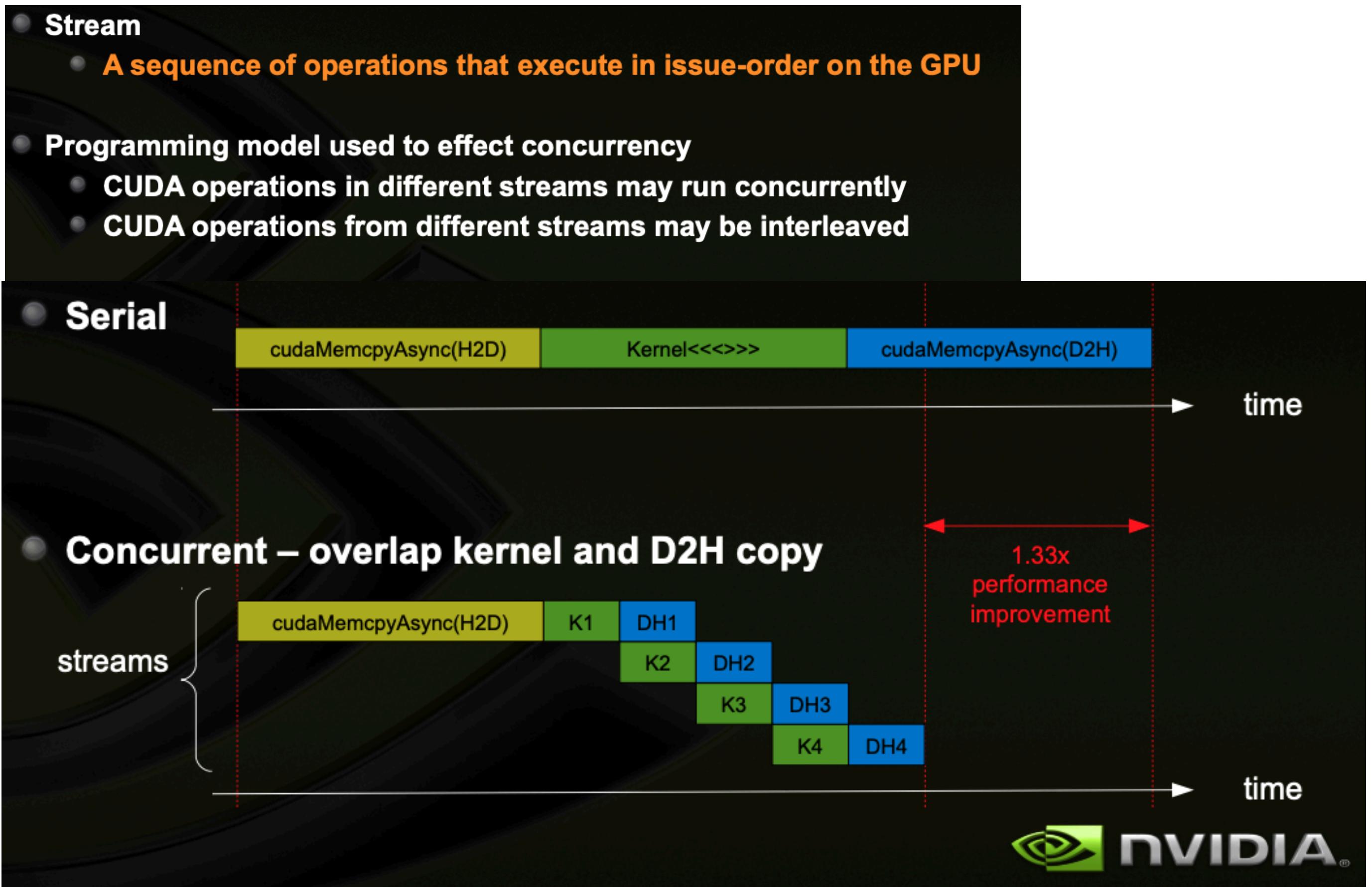
- Async
- if called before stream is complete and resources will be released when stream on device is complete

Stream Concurrency example

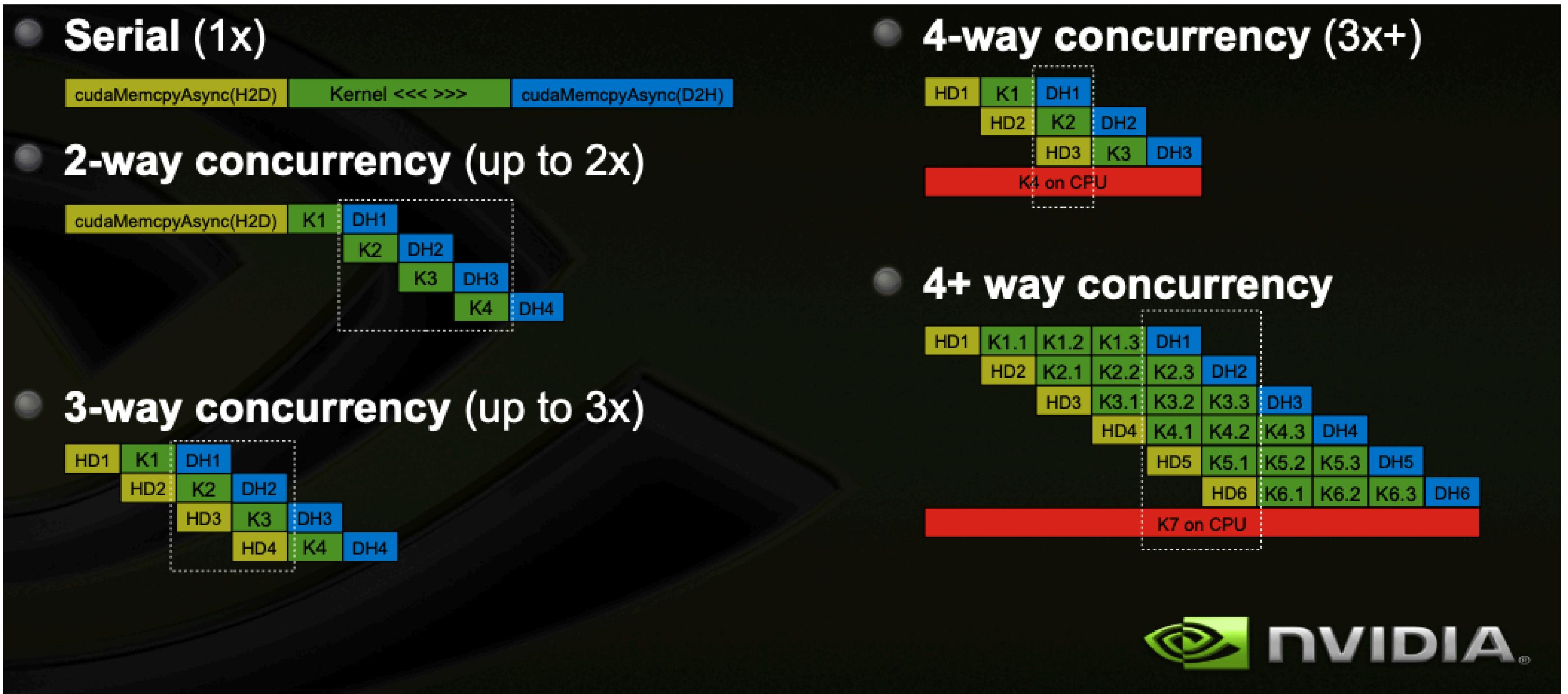
Serial Execution:



Stream Concurrency example



Stream Concurrency example



Stream concurrency requirements

- CUDA operations must be in different, non-zero, streams
- cudaMemcpyAsync with host 'pinned' memory
 - Page-locked memory
 - cudaHostMalloc() or cudaHostAlloc()
- Sufficient resources must be available
 - cudaMemcpyAsyncs in different directions
 - device resources (SMEM, registers, blocks, etc)

Stream concurrency requirements

- CUDA operations must be in different, non-zero, streams
 - cudaMemcpyAsync with host 'pinned' memory
 - Page-locked memory
 - cudaHostMalloc() or cudaHostAlloc()
 - Sufficient resources must be available
 - cudaMemcpyAsyncs in different directions
 - device resources (SMEM, registers, blocks, etc)
- Enough DMA Engines
for concurrent I/O
- Enough compute resources for
concurrent kernel execution

Stream: Overlap of Data Transfers and Kernel Execution

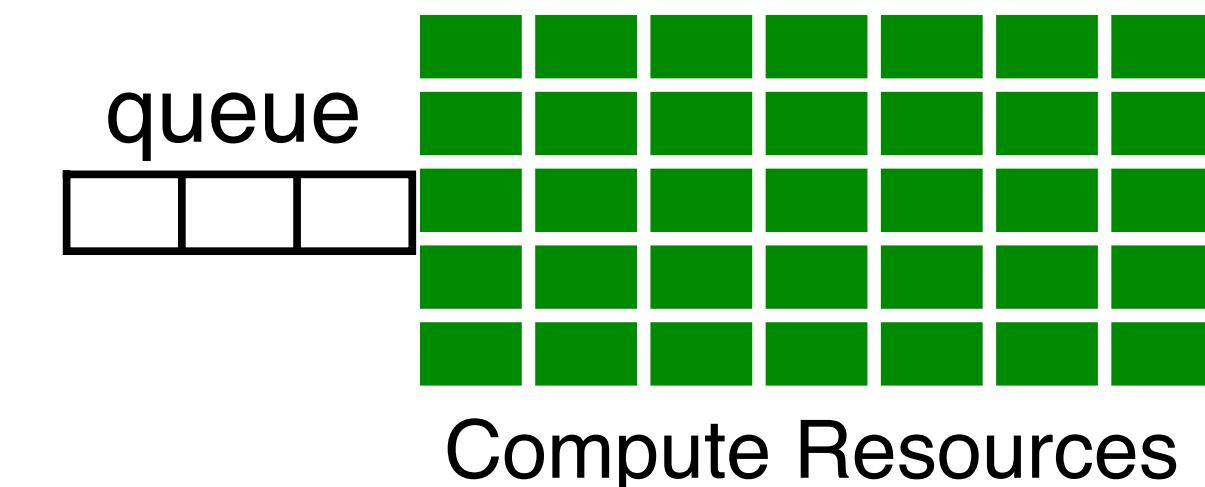
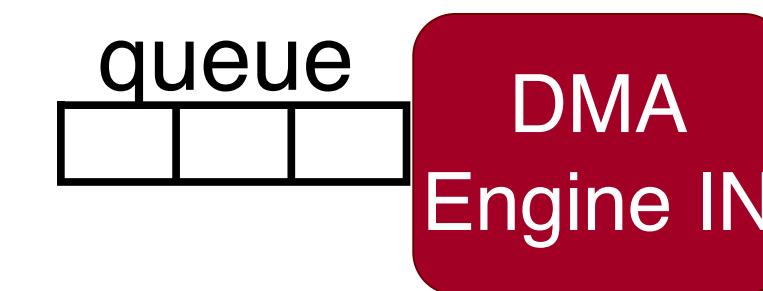
<https://docs.nvidia.com/cuda/cuda-c-programming-guide/#overlapping-behavior>

```
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                   size, cudaMemcpyHostToDevice, stream[i]);
for (int i = 0; i < 2; ++i)
    MyKernel<<<100, 512, 0, stream[i]>>>
        (outputDevPtr + i * size, inputDevPtr + i * size, size);
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                   size, cudaMemcpyDeviceToHost, stream[i]);
```

- "amount of execution overlap between two streams depends on the order in which the commands are issued to each stream"

Stream: Overlap of Data Transfers and Kernel Execution

- Beware of Head-of-line (HOL) blocking
 - Resource Contention/Camping
 - DMA engines
 - Or compute resources



Event Streams

<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/asynchronous-execution.html#cuda-events>

```
cudaEvent_t event ;  
cudaEventCreate( &event ) ;
```

```
cudaEventRecord( event, stream[i] ) ;
```

Like an operation added to the stream that sets a flag host-side when it reaches head of work queue GPU-side

```
cudaStreamWaitEvent( event ) ;
```

```
cudaQueryEvent( event ) ;
```

- Marker in a stream
 - synchronize stream execution
 - monitor device progress
 - Useful for synchronizing concurrent streams

Blocks until
event occurs

CUDA_SUCCESS if
event occurred

Stream: Host functions (Callbacks)

<https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/asynchronous-execution.html#callback-functions-from-streams>

```
void CUDART_CB MyCallback(void *data){  
    printf("Inside callback %d\n", (size_t)data);  
}  
...  
for (size_t i = 0; i < 2; ++i) {  
    cudaMemcpyAsync(devPtrIn[i], hostPtr[i], size, cudaMemcpyHostToDevice, stream[i]);  
    MyKernel<<<100, 512, 0, stream[i]>>>(devPtrOut[i], devPtrIn[i], size);  
    cudaMemcpyAsync(hostPtr[i], devPtrOut[i], size, cudaMemcpyDeviceToHost, stream[i]);  
    cudaLaunchHostFunc(stream[i], MyCallback, (void*)i);  
}
```

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/#host-functions-callbacks>

- Callback occurs after all previously queue operations completed
- Restrictions
 - No CUDA function can be in call back: directly or indirectly