POLITECNICO
MILANO 1863

POLITECNICO MILANO 1863
NECST laboratory

CUDA Multi-Stream Programming

Ian Di Dio Lavore - 07/05/2025

WAIT, who am I?

Final year PhD Student in Computer Science **@PoliMi**

→ Worked on the ORACLE Labs | POLITECNICO MILANO 1863 NECST laboratory partnership

→ Ex **HPC Intern** at Pacific Northwest NATIONAL LABORATORY

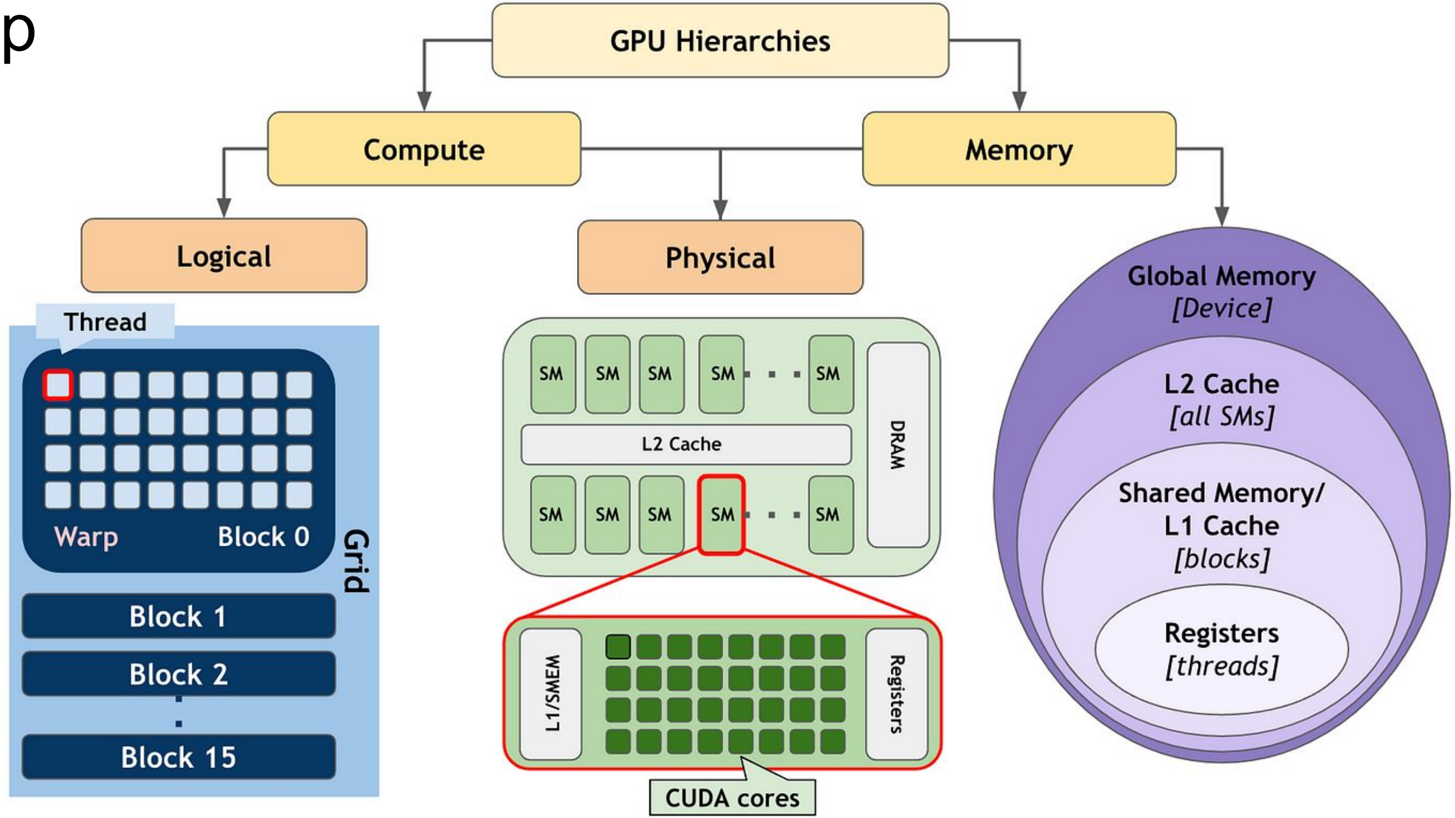
→ Incoming **Research Intern** at Microsoft

Interested in: **HPC** / **Programming Models** / **Distributed Systems**

Example: Handling parallel reductions in CUDA

Based on:
https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf
https://devblogs.nvidia.com/faster-parallel-reductions-kepler/
https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/
https://developer.nvidia.com/blog/faster-parallel-reductions-kepler/

Recap



https://medium.com/@rimikadhara/cuda-1-gpu-v-s-cpu-4802c9113941

It all reduces to a number

Let's compute the **sum of an array**

- Common primitive: extended to dot-product, min, max, etc.

"Easy" to implement in CUDA, hard to get it right

- In C, just loop over all values
- But on GPUs, we can sum chunks of the array in parallel
- Then, sum those chunks. Then, repeat → **Tree-reduction**
- With infinite paralellism, O(log(N)) instead of O(N)

We need to use multiple thread blocks

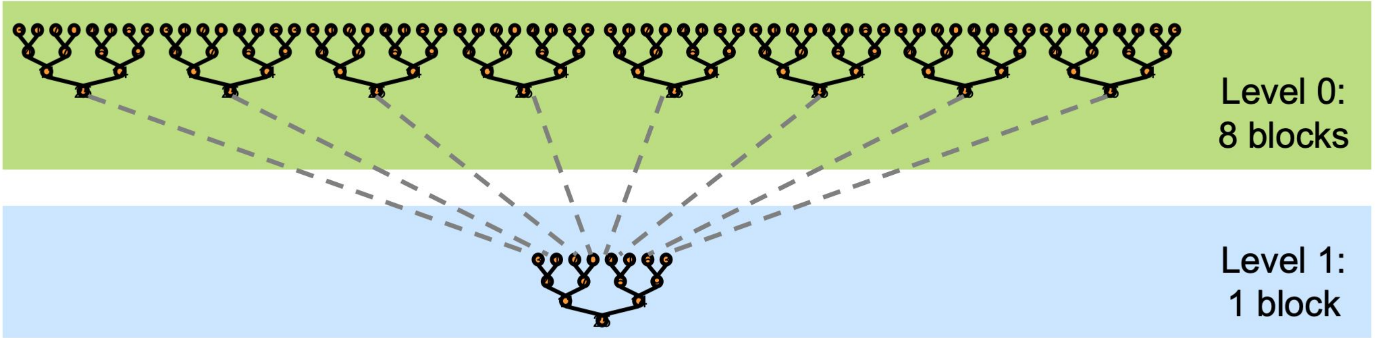
- Each thread block reduces a portion of the array

1. Old approach (monolithic, 2007) developer.download.nvidia.com/assets/cuda/files/reduction.pdf
2. Better approach (grid-stride, 2013, 2014) developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/, https://developer.nvidia.com/blog/faster-parallel-reductions-kepler/

Kernel decomposition

Avoid global sync by decomposing computation into multiple (2) kernel invocations, or just do the second step on CPU

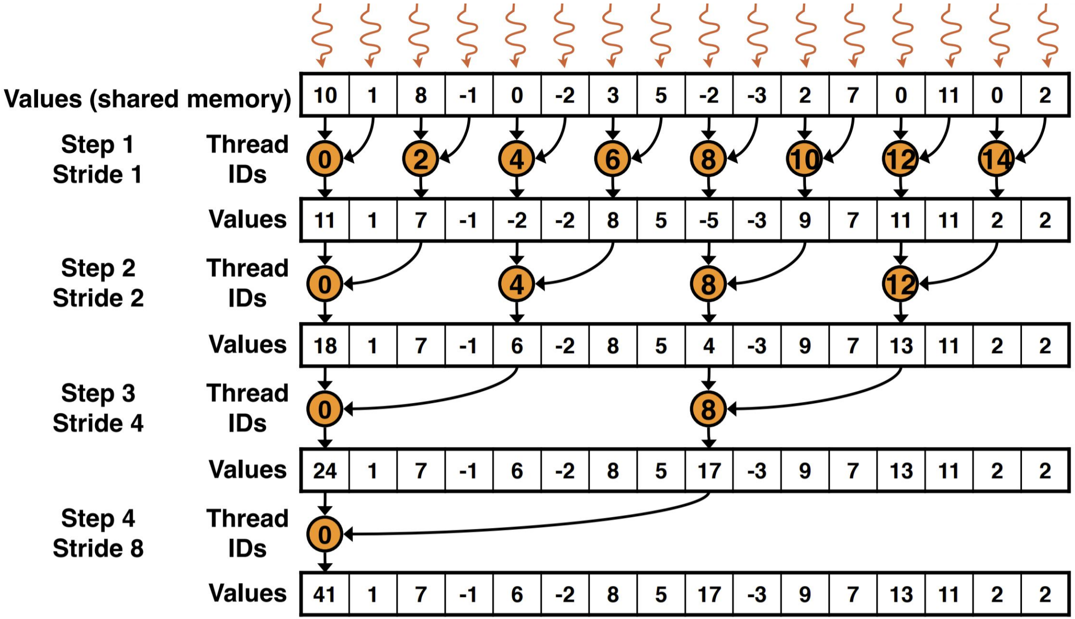
- In this case the kernel to call is the same, the grid size changes



Reductions have low arithmetic intensity (math operation for each byte read)

- We need to work on peak bandwidth

1. Interleaved Adderessing



Values (shared memory)		10	1	8	-1	0	-2	3	5	-2	-3	2	7	0	11	0	2
Step 1 Stride 1	Thread IDs	0		2		4		6		8		10		12		14	
	Values	11	1	7	-1	-2	-2	8	5	-5	-3	9	7	11	11	2	2
Step 2 Stride 2	Thread IDs	0				4				8				12			
	Values	18	1	7	-1	6	-2	8	5	4	-3	9	7	13	11	2	2
Step 3 Stride 4	Thread IDs	0								8							
	Values	24	1	7	-1	6	-2	8	5	17	-3	9	7	13	11	2	2
Step 4 Stride 8	Thread IDs	0															
	Values	41	1	7	-1	6	-2	8	5	17	-3	9	7	13	11	2	2

Idea 1: tree-reduction inside blocks (on the left), then sum the «partial sums» on the CPU (as `num_blocks << N`)

O(Log(T)) steps.

```
__global__ void reduce0(int *g_idata, int *g_odata) {
}
```

Shared memory with size defined when the kernel is called

```
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];









}
```

Shared memory with size defined when the kernel is called

```
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];
    // each thread loads one element from global to shared mem (warning: no boundary checks!)
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;






}
```

Shared memory with size defined when the kernel is called

```
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];
    // each thread loads one element from global to shared mem (warning: no boundary checks!)
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];



}
```

All warp’s threads loads in parallel the shared memory

Shared memory with size defined when the kernel is called

```
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];
    // each thread loads one element from global to shared mem (warning: no boundary checks!)
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();









}
```

All warp’s threads loads in parallel the shared memory

Ensure that all threads in the block have loaded the value

Shared memory with size defined when the kernel is called

```
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];
    // each thread loads one element from global to shared mem (warning: no boundary checks!)
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();
    // do reduction in shared mem
    for(unsigned int s = 1; s < blockDim.x; s *= 2) {




    }


}
```

All warp’s threads loads in parallel the shared memory

Ensure that all threads in the block have loaded the value

Sum the array chunk loaded in the current block.
log2(blockDim) iterations.

Shared memory with size defined when the kernel is called

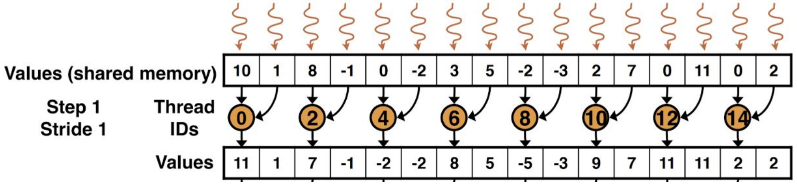
```
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];
    // each thread loads one element from global to shared mem (warning: no boundary checks!)
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();
    // do reduction in shared mem
    for(unsigned int s = 1; s < blockDim.x; s *= 2) {
        if (tid % (2 * s) == 0) {
            sdata[tid] += sdata[tid + s];
        }

    }


}
```

All warp's threads loads in parallel the shared memory

Ensure that all threads in the block have loaded the value

Sum the array chunk loaded in the current block.
log2(blockDim) iterations.

Only the «left» thread does the accumulation
Step 1: on every even thread
Step 2: every 4 thread
...



Values (shared memory)	10	1	8	-1	0	-2	3	5	-2	-3	2	7	0	11	0	2
Step 1 Stride 1 — Thread IDs	0		2		4		6		8		10		12		14	
Values	11	1	7	-1	-2	-2	8	5	-5	-3	9	7	11	11	2	2

```
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];
    // each thread loads one element from global to shared mem (warning: no boundary checks!)
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();
    // do reduction in shared mem
    for(unsigned int s = 1; s < blockDim.x; s *= 2) {
        if (tid % (2 * s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }


}
```

Shared memory with size defined when the kernel is called

All warp's threads loads in parallel the shared memory

Ensure that all threads in the block have loaded the value

Sum the array chunk loaded in the current block.
log2(blockDim) iterations.

Only the «left» thread does the accumulation
Step 1: on every even thread
Step 2: every 4 thread
…

Ensure current step is done

```
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];
    // each thread loads one element from global to shared mem (warning: no boundary checks!)
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();
    // do reduction in shared mem
    for(unsigned int s = 1; s < blockDim.x; s *= 2) {
        if (tid % (2 * s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }
    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Shared memory with size defined when the kernel is called

All warp’s threads loads in parallel the shared memory

Ensure that all threads in the block have loaded the value

Sum the array chunk loaded in the current block.
log2(blockDim) iterations.

Only the «left» thread does the accumulation
Step 1: on every even thread
Step 2: every 4 thread
…

Ensure current step is done

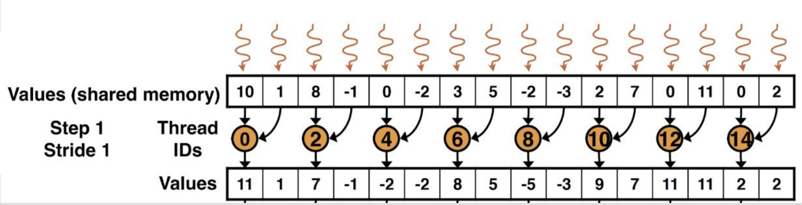
The first thread in each block store its sum to global memory,
Then we obtain the final sum on CPU

Shared memory with size defined when the kernel is called

```
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];
    // each thread loads one element from global to shared mem (warning: no boundary checks!)
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x
    sdata[tid] = g_idata[i];
    __syncthreads();
    // do reduction in shared mem
    for(unsigned int s = 1; s < blockDim.x; s *= 2) {
        if (tid % (2 * s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }
    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

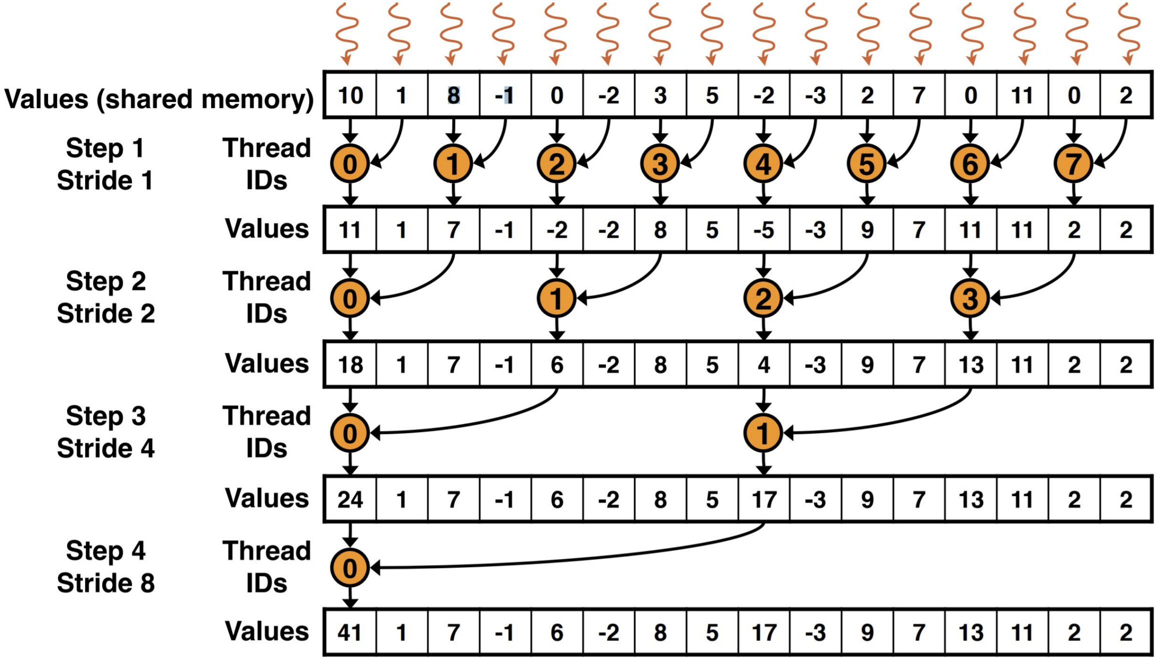
Ensure th

Warp divergence! (consecutive threads do different operations)
And % is slow!



The first thread in each block store its sum to global memory,
Then we obtain the final sum on CPU

2. Fixing warp divergence



Values (shared memory)	10	1	8	-1	0	-2	3	5	-2	-3	2	7	0	11	0	2
Values	11	1	7	-1	-2	-2	8	5	-5	-3	9	7	11	11	2	2
Values	18	1	7	-1	6	-2	8	5	4	-3	9	7	13	11	2	2
Values	24	1	7	-1	6	-2	8	5	17	-3	9	7	13	11	2	2
Values	41	1	7	-1	6	-2	8	5	17	-3	9	7	13	11	2	2

Idea 2: remove % operator by having consecute threads do the processing (cmp. Slide 42, we accumulated on {0, 2, 4, ...})

Still do the last sums on CPU

```
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];
    // each thread loads one element from global to shared mem (warning: no boundary checks!)
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();
    // do reduction in shared mem
    for(unsigned int s = 1; s < blockDim.x; s *= 2) {
        int index = 2 * s * tid;
        if (index < blockDim.x) {
           sdata[index] += sdata[index + s];
        }
        _syncthreads();
    }
    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

s=1, tid=0 → sum indices 0 and 1
s=1, tid=1 → sum indices 2 and 3
...
s=2, tid=0 → sum indices 0 and 2
s=2, tid=1 → sum indices 4 and 6
...
We no longer interleave threads that do different operations

Grid-stride loops

Avoid having the number of blocks be a function of input size, e.g. `blocks = (N+T-1)/T`

That’s ugly and limiting!

1. Creating and switching between blocks has overhead
2. **Rule of thumb:** best performance is obtained with num. blocks ≈ 1-8x number of SM
3. Using 1 block is useful for debugging

Instead, use **grid-stride loops**, to have kernels with arbitrary num. of blocks

```
__global__
void add(int n, int *x, int *y, int *z)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) z[i] = x[i] + y[i];
}
```

→

```
__global__
void add(int n, int *x, int *y, int *z)
{
    int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
    for(int i = thread_id; i < n; i += blockDim.x * gridDim.x) {
        z[i] = x[i] + y[i];
    }
}
```

3. Grid-stride loops

So far, number of blocks is a function of input size, e.g. blocks = (N+B-1)/B

That's ugly and limiting!

1. Creating and switching between blocks has overhe
2. **Rule c** ith
3. Using

Instead,

Why jumping by blockDim.x * gridDim.x instead of 1?
So that contiguous threads access contiguous memory indices (coalesced accesses)

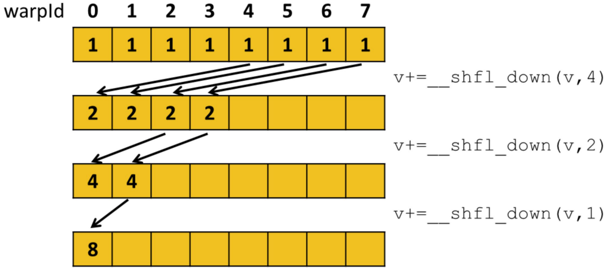
blockDim.x * gridDim.x
is the total number of threads.
Each thread processes N / total_threads elements

```
__global__
void add(int n, int *x, int *y, int *z)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) z[i] = x[i] + y[i];
}
```

```
__global__
void add(int n, int *x, int *y, int *z)
{
    int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
    for(int i = thread_id; i < n; i += blockDim.x * gridDim.x) {
        z[i] = x[i] + y[i];
    }
}
```

Shuffle down

A better way to do tree-reduction in a warp, since Kepler (2013)



```
__device__ float warp_reduce(float val) {
    int warp_size = 32;
    for (int offset = warp_size / 2; offset > 0; offset /= 2)
        val += __shfl_down_sync(0xFFFFFFFF, val, offset);
    return val;
}
```

- Each thread has a different value of «val»
- `__shfl_down_sync` passes this «val» to a thread with index equal to `threadIdx.x - offset`
- For example, thread with `threadIdx.x = 4` passes 1 to thread 0
- For thread 0, `__shfl_down_sync` returns the value passed by thread 4
- For thread 4, `__shfl_down_sync` does not return anything

Note: `__shfl_down_sync(0xFFFFFFFF,…` is the same as `__shfl_down(…,` it's the new syntax since CUDA 9

3. A better reduction

A better way to do tree-reduction in a warp, since Kepler (2013).

No need to aggregate data on the CPU

```
__global__ void reduce(int *x, int *res, int n) {
    int warp_size = 32;
    float sum = float(0); // Initialize partial sum for this thread;
    for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x) {
        sum += x[i]; // Each thread accumulates N / total_threads values;
    }
    sum = warp_reduce(sum);                    // Obtain the sum of values in the current warp;
    if ((threadIdx.x & (warp_size - 1)) == 0)  // Same as (threadIdx.x % warp_size) == 0 but faster
        atomicAdd(res, sum);                   // The first thread in the warp updates the output;
}
```

Values have been accumulated in the first thread of each block, so it's the only one we consider

`atomicAdd` is synchronized for the whole GPU. Use with care, it can be very slow!

4. Even better

A better way to do tree-reduction in a warp, since Kepler (2013).
No need to aggregate data on the CPU

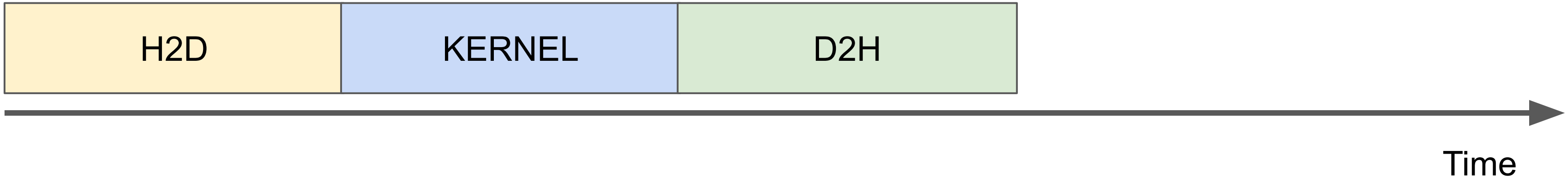
```
__global__ void reduce(int *x, int *res, int n) {
    int warp_size = 32;
    float sum = float(0); // Initialize partial sum for this thread;
    for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x) {
        sum += x[i]; // Each thread accumulates N / total_threads values;
    }
    sum = __reduce_add_sync(0xFFFFFFFF, sum);
    if ((threadIdx.x & (warp_size - 1)) == 0)  // Same as (threadIdx.x % warp_size) == 0 but faster
        atomicAdd(res                                            warp updates the output;
}
```

This does the warp reduction for us!
It adds the value of «sum» for all threads in the current warp, and return the total sum.
Only on Ampere, on integers

CUDA Multi-Stream Programming

Ian Di Dio Lavore - 07/05/2025

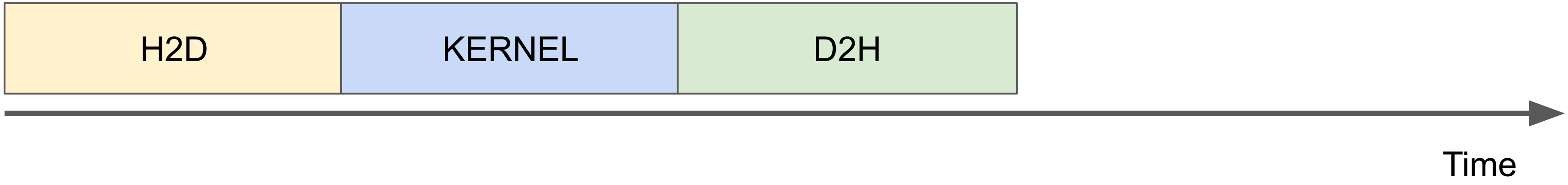
The usual way



→ Data-Level parallelism

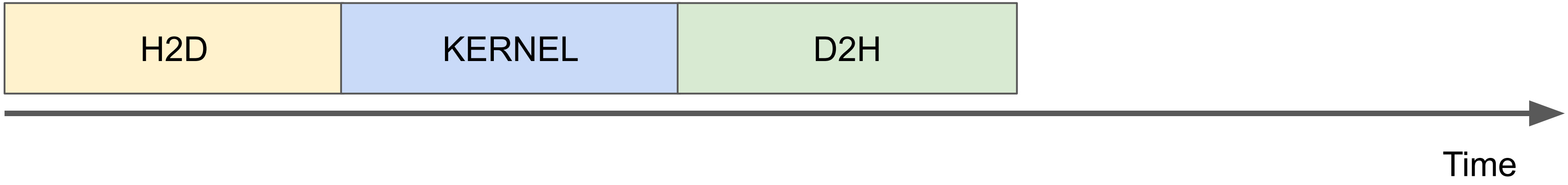
Our kernel is being executed by (possibly) thousand of threads in parallel
→ Same logical operation applied to different portion of data
→ Single Instruction Multiple Data (SIMD) paradigm

The usual way



```
float *a, *d_a;

a = (float*)malloc(sizeof(float) * N);
cudaMalloc(&d_a, sizeof(float) * N);

do_stuff(a);

cudaMemcpy(d_a, a, sizeof(float) * N, cudaMemcpyHostToDevice);

kernel<<<GridDim,BlockDim>>>(out, d_a, b, N);

cudaMemcpy(a, d_a, sizeof(float) * N, cudaMemcpyDeviceToHost);
```

The usual way



```
float *a, *d_a;

a = (float*)malloc(sizeof(float) * N);
cudaMalloc(&d_a, sizeof(float) * N);

do_stuff(a);

cudaMemcpy(d_a, a, sizeof(float) * N, cudaMemcpyHostToDevice);

kernel<<<GridDim,BlockDim>>>(out, d_a, b, N);

cudaMemcpy(a, d_a, sizeof(float) * N, cudaMemcpyDeviceToHost);
```

Can we do better?

CUDA Streams

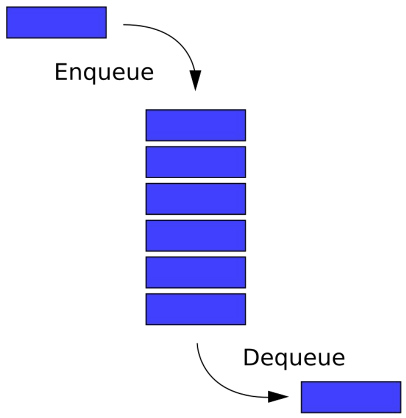
Modern GPUs empower programmers with multiple streams, i.e., independent work queues where the host can quickly submit tasks without waiting.

CUDA Streams

Modern GPUs empower programmers with multiple streams, i.e., independent work queues where the host can quickly submit tasks without waiting.

Tasks in the same stream are automatically synchronized:

➔ they execute in order (FIFO - First In, First Out).



https://commons.wikimedia.org/wiki/File:Fifo_queue.svg

CUDA Streams

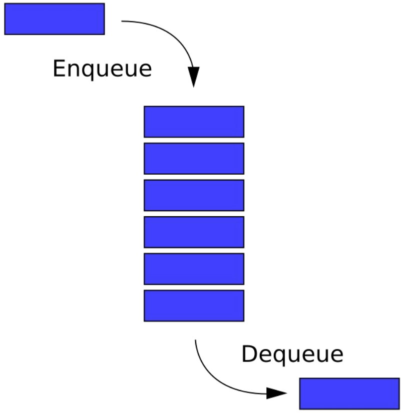
Modern GPUs empower programmers with multiple streams, i.e., independent work queues where the host can quickly submit tasks without waiting.

Tasks in the same stream are automatically synchronized:

➔ they execute in order (FIFO - First In, First Out).

Tasks in different streams are asynchronous:

➔ no guaranteed order between them.



https://commons.wikimedia.org/wiki/File:Fifo_queue.svg

CUDA Streams

Modern GPUs empower programmers with multiple streams, i.e., independent work queues where the host can quickly submit tasks without waiting.

Tasks in the same stream are automatically synchronized:

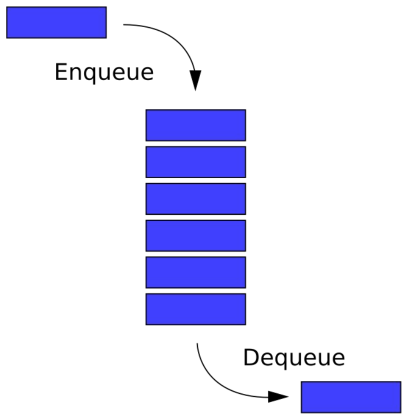
➔ they execute in order (FIFO - First In, First Out).

Tasks in different streams are asynchronous:

➔ no guaranteed order between them.

N.B. The Default Stream acts synchronously with all others:

➔ It waits for all tasks in other streams to finish first.



https://commons.wikimedia.org/wiki/File:Fifo_queue.svg

How can we handle streams?

How can we handle streams?

```
cudaStream_t stream;
              → Creates a stream handle
```

How can we handle streams?

```
cudaStream_t stream;
        → Creates a stream handle


cudaStreamCreate(&stream);
        → Allocates a stream
```

How can we handle streams?

```
cudaStream_t stream;
        → Creates a stream handle


cudaStreamCreate(&stream);
        → Allocates a stream


cudaStreamDestroy(stream);
        → Deallocates a stream
        → Synchronizes host until work in stream has completed
```

How to place work into a stream?

```
kernel<<<GridDim, BlockDim, SharedMem, Stream>>>(...);
        → 4th launch parameter for kernels
```

```
cudaMemcpyAsync(dest, src, size, direction, stream);
        → some API calls will take the stream handle as input
```

The Default Stream

In all API calls that accept a stream as input parameter, if the stream is not passed, the call will be executed on the "Default Stream" (aka. Stream 0).

The default stream has a particular property:

- Synchronized with respect to all other streams
  → Operations on other streams cannot overlap with the default one

- Only exception: `cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking)`
  → Used by libraries out of user control (like MPI)

Kernel Concurrency

Let’s assume that we have a kernel “foo” that uses 50% of the GPU resources.

Host call

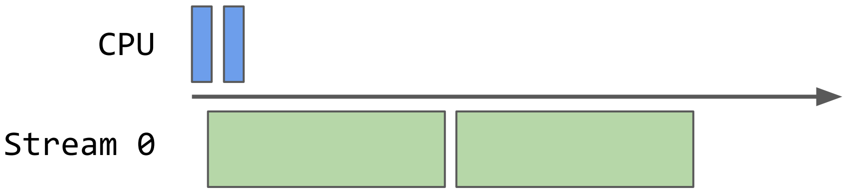
kernel

Kernel Concurrency

Let’s assume that we have a kernel “foo” that uses 50% of the GPU resources.

DEFAULT STREAM:

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



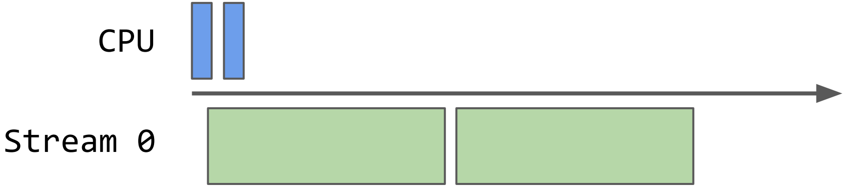
Host call · kernel

Kernel Concurrency

Let’s assume that we have a kernel “foo” that uses 50% of the GPU resources.

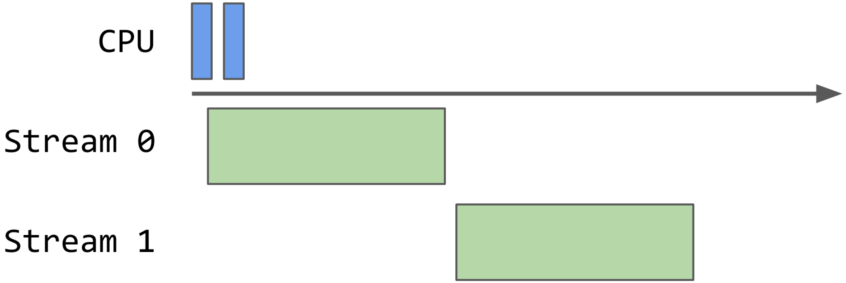
DEFAULT STREAM:

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



DEFAULT + User managed streams:

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



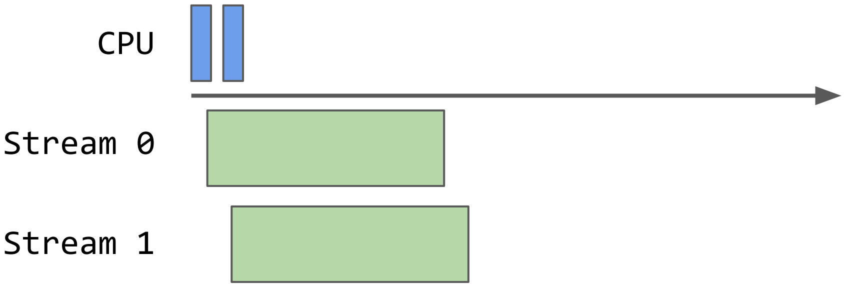
Host call | kernel

Kernel Concurrency

Let’s assume that we have a kernel “foo” that uses 50% of the GPU resources.

```
DEFAULT + managed streams with flags:
```

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



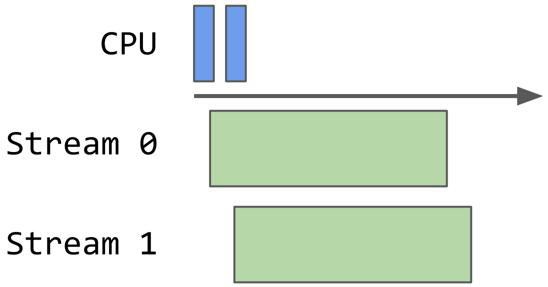
🟦 Host call 🟩 kernel

Kernel Concurrency

Let’s assume that we have a kernel “foo” that uses 50% of the GPU resources.

`Only managed streams:`

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



Data transfer concurrency

What about memory movements?

Three main types of memories:

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

Pinned Host Memory

Why pinned memory?

- Pageable memory is transferred passing through the CPU
- Pinned memory enables the usage of DMA engines
    - Frees the CPU for asynchronous execution
    - Achieves a higher percent of peak bandwidth

How to **manage it**?

- cudaMallocHost(...) or cudaHostAlloc(...)
    - Allocate pinned memory on the host
    - Replaces the functions of malloc/new
- cudaFreeHost(...)
    - Frees memory allocated by the previous functions
- cudaHostRegister(...) or cudaHostUnregister(...)
    - Pins/Unpins pageable memory regions (making it pinned memory)
    - Slow → don’t do it often

Concurrent memory copies

- cudaMemcpy(...)
  - Places the copy command in the default stream
  - Synchronous with respect to the host → must complete before returning
- cudaMemcpyAsync(... , &stream)
  - Places the data transfer into the stream and returns immediately

Conditions to achieve concurrency:

- Transfers must be in a non-default stream
- Must be async copies
- 1 transfer per direction (PCIe design limitation)
- Memory on the host must be pinned

Example: Paged memory

```
int *h_ptr, *d_ptr;

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

cudaMemcpy(d_ptr, h_ptr, bytes, cudaMemcpyHostToDevice);

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

Example: Pinned memory - 1

```
int *h_ptr, *d_ptr;

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

cudaMemcpy(d_ptr, h_ptr, bytes, cudaMemcpyHostToDevice);

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

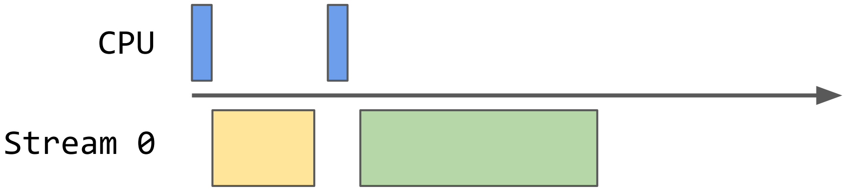
Example: Pinned memory - 2

```
int *h_ptr, *d_ptr;

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

cudaMemcpy(d_ptr, h_ptr, bytes, cudaMemcpyHostToDevice);

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

Data movements concurrency

SYNCHRONOUS:

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



Host call

Mem. op.

kernel

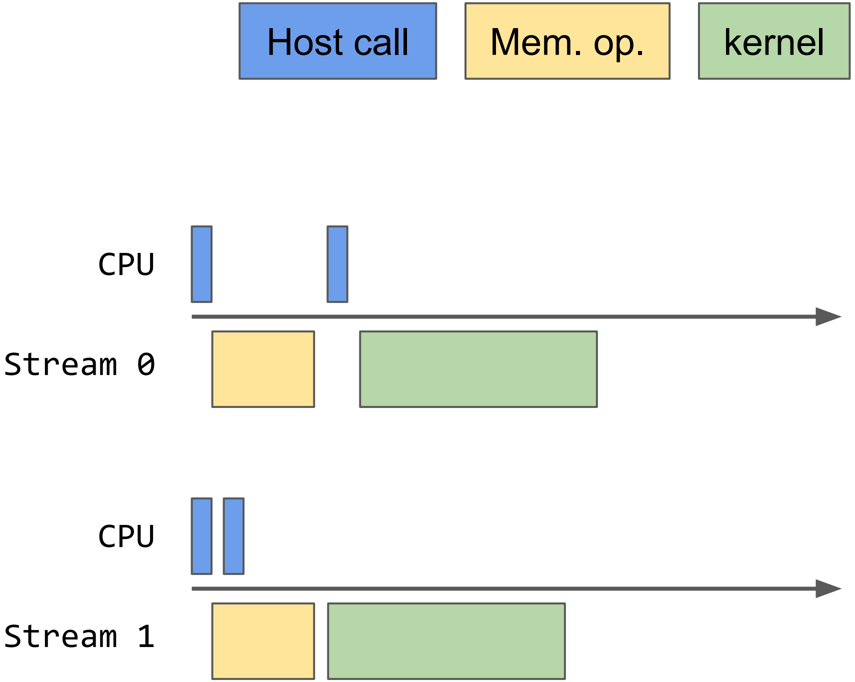
Data movements concurrency

SYNCHRONOUS:

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

ASYNC same stream:

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



Data movements concurrency

SYNCHRONOUS:

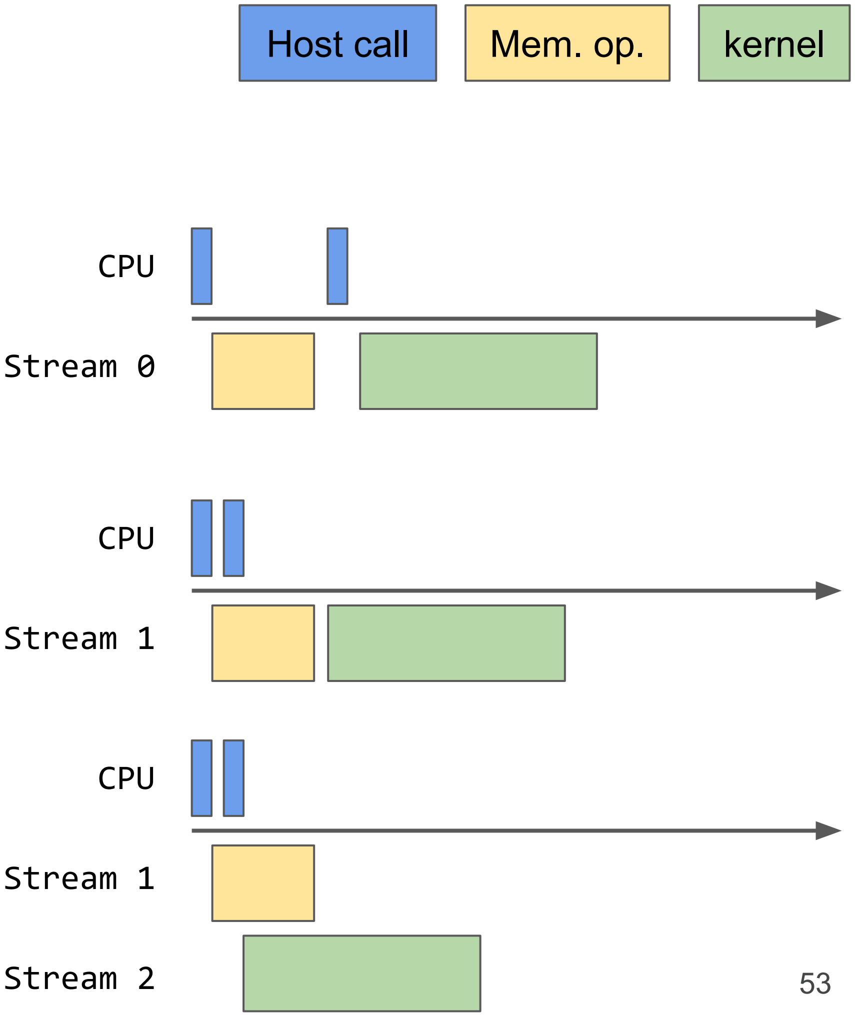
```
cudaMemcpy(...);
foo<<<...>>>();
```

ASYNC same stream:

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

ASYNC different streams:

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



“All that glisters is not gold” - W. Shakespeare

No free parallelism!

- In order to have asynchronous execution between a kernel and a data movement, the kernel needs to work on a different memory region
  - The source code needs to be refactored (if possible) in order to work on partitions of the original data

- GPUs starting from the Kepler architecture support a maximum of 32 HW queues dedicated to streams.

- Too much pinned memory can hurt host performance!

Host-Device synchronization

Common APIs to synchronize

- **cudaDeviceSynchronize()**
  - The one command to rule (block) them all!
  - Blocks host until all issued CUDA calls are completed

- **cudaStreamSynchronize(stream)**
  - Synchronize with respect to a single stream
  - Block host until all issued CUDA calls in that specific stream are completed

- **CUDA Events**
  - The most customizable way
  - Allows you to extract the maximum parallelism between tasks

CUDA Events

Provides a mechanism to signal when operations have occurred in a stream.

→ use them for profiling
→ add a more fine grained way of handle synchronization

Boolean state:

→ occurred / not occurred
→ default state: occurred

Manage CUDA Events

```
cudaEventCreate(&event);
    → create an event

cudaEventDestroy(event);
    → destroy an event

cudaEventCreateWithFlags(&event, cudaEventDisableTiming);
    → disable timings to increase performance

cudaEventRecord(&event, stream);
    → sets the event state to not-occurred
    → enqueue the event into the stream (streams are FIFO queues)
    → event state is set to occurred once it reaches the front of the stream
```

How to synchronize using CUDA Events

cudaEventQuery(event)
→ returns CUDA_SUCCESS if the event has occurred

cudaEventSynchronize(event)
→ blocks host until the stream arrives at the event

cudaStreamWaitEvent(**stream**, event)
→ blocks the stream until event occurs
→ blocks only launches after this call
→ does not block the host!

If you want to check just the correctness of your program:
→ CUDA_LAUNCH_BLOCKING=1 → CUDA calls will be synchronous wrt host

Credits

Slides adapted from:

CUDA STREAMS - Best Practices and Common Pitfalls (Justin Luitjens - NVIDIA)

*Unless stated otherwise, all images, trademarks and examples are property of their respective owners

Additional material:

– Programming Massively Parallel Processors, 3rd Edition

– Steve Rennich - NVIDIA
→ GTC: CUDA C/C++ Streams and Concurrency

– Abbas Mazloumi - University of California, Riverside
→ CS/EE 147 – GPU Computing and Programming

- Sergio Orlandini - CINECA
→ Introduction to GPU Accelerators and CUDA Programming