

4

CUDA Streams and Page-Lock Memory

TOPICS

- What is a stream.
 - How to use one and multiple streams.
 - What the page-lock memory is, its advantages and disadvantages.
 - How to use page-lock memory.
-
- **Key Words:** *Stream, deviceOverlap, cudaStream_t, cudaStreamCreate, cudaStreamDestroy, page-lock, pinned memory, pageable, cudaMalloc(), malloc(), cudaHostAlloc(), cudaMemcpyAsync(), cudaStreamSynchronize(), cudaFreeHost().*



Single Streams

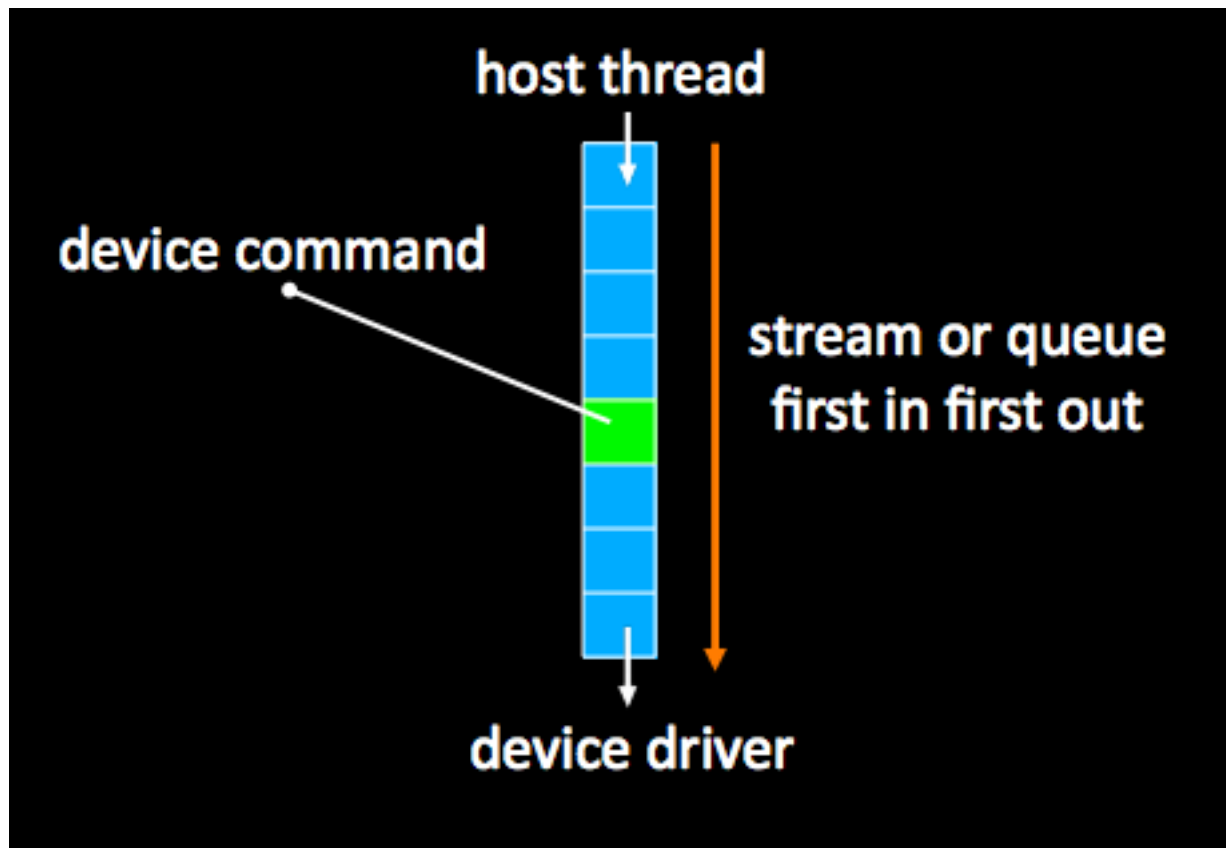
- An option to accelerate our applications is through the use of streams; each stream is a line of execution in our program.
- A CUDA stream represents a queue of GPU operations that get executed in a specific order.
- We can add operations such as kernel launches, memory copies, and event starts and stops into a stream (first in, first out).
- Kernel launches and host \leftrightarrow device memory copies that do not specify any stream parameter, or equivalently that set the stream parameter to zero, are issued to the default stream.

- In all the other practices we have been using just one stream of execution on the GPU, which is usually device 0.

```
cudaError_t cudaStreamCreate(cudaStream_t * pStream )
```

where:

pStream - Pointer to new stream identifier



- The first step we have to check, is if the GPU has support for device overlap in order to know if the GPU can support the execution of a kernel, and the copy between device and host at the same time.
- In the Device Query practice we saw a property called *deviceOverlap*, this is the property we are going to use to test our GPU.
- The second step is to create the stream. To do this we are going to use the `cudaStream_t`, a data type used by CUDA Runtime. Then, the *cudaStreamCreate()* function is used to create a stream.

```
// create the stream
cudaStream_t stream1;
HANDLER_ERROR_ERR(cudaStreamCreate (&stream1));
```

- Next step is to associate the stream with our kernel.

```
functionKernel1<<< DIMGRID, DIMBLOCK , 0, stream1 >>>(d_a, N);
```

- Now, we need to tell the host that it must wait for the GPU to finish before proceeding.

```
cudaDeviceSynchronize();  
functionKernel2<<< DIMGRID, DIMBLOCK, 0, stream1 >>>(d_b, N);
```

- Finally we have to destroy the stream used.

```
cudaStreamDestroy(stream1);
```



Example

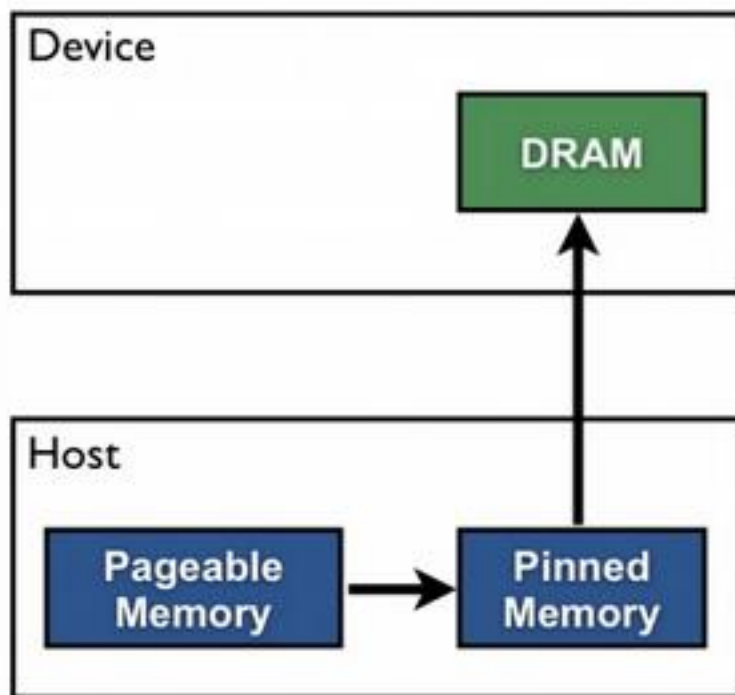
- `01single_stream.cu`: This example shows how to create a single stream to process a job.



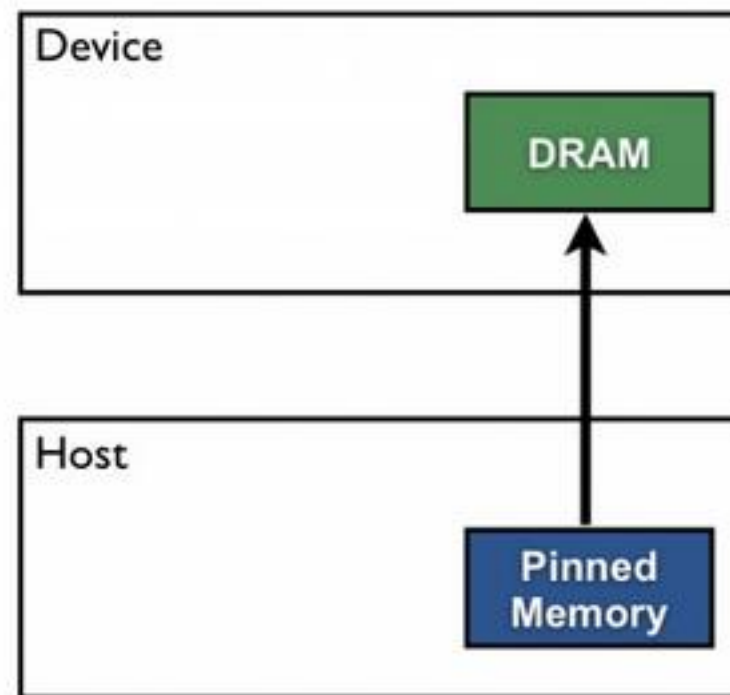
Page-Lock Host Memory

- To allocate memory in the host we have been using the *malloc()* function. When this function is invoked, it allocates pageable host memory, which means that the operating system could page this memory into the disk.
- To avoid this, we can use *cudaHostAlloc()*. This function allocates a buffer of page-locked (pinned memory).
- The memory stored in this way has its residence in physical memory, and it will not be relocated.
- Using *cudaHostAlloc()* to allocate memory, it is necessary to use *cudaFreeHost()* instead of just the common *free()* function.

Pageable Data Transfer



Pinned Data Transfer




■ Advantages

- The operating system will never page this memory out to disk.
- If the physical address is known, it is possible for the GPU to use direct memory access (DMA) to copy data to or from the host, the CPU is not involved in this operation.
- Copies between page-locked host memory and device memory can be performed concurrently with kernel execution.
- On systems with a front-side bus, bandwidth between host memory and device memory is higher if host memory is allocated as page-locked.

- Using page-locked host memory can support executing more than one device kernel concurrently for compute capability 2.0.
- There is no need to allocate a block in device memory and copy data between this block and a block in host memory, the data transfer is implicitly performed by the kernel.
- Mapped memory is able to exploit the full duplex of the PCI express bus by reading and writing at the same time, since memory copy only move data in one direction, it is half duplex.
- Therefore, the copy occurs twice, first to pageable system buffer to a page-locked buffer and the from page-locked system to the GPU.

■ Disadvantages:

- The application always needs enough physical memory for each page-locked buffer, since the buffers can never be swapped out to disk.
- Your application can affect the performance of other applications that are running in the system.
- Your application could fail in systems with small physical memory.
- The page-locked memory is shared with host and device, any application must avoid its use simultaneously.



Example:

- **02page-lock**: This example shows a benchmark comparing the performance of using pageable and page-locked memory.



CUDA Asynchronous Memory Copy

- Some functions are supported asynchronous launching in order to facilitate concurrent execution between host and device resource control.
- Control is returned to the host thread before the work is finished.
- To use asynchronous memory copies, it is necessary to use `cudaMemcpyAsync()` function.
- When the call returns, there is no guarantee that the copy has even started yet, much less that it has finished.

- The guarantee that we have is that the copy will definitely be performed before the next operation placed into the same stream.
- It is required that any host memory pointers passed to `cudaMemcpyAsync()` have been located by `cudaHostAlloc()`.
- You are only allowed to shedule asynchronous copies to or from page-lock memory.

```
cudaError_t cudaMemcpyAsync (void *dst, const void *src, size_t count,  
enum cudaMemcpyKind kind, cudaStream_t stream = 0)
```

<code>dst</code>	- Destination memory address
<code>src</code>	- Source memory address
<code>count</code>	- Size in bytes to copy
<code>kind</code>	- Type of transfer
<code>stream</code>	- Stream identifier

- If we would like to guarantee that the GPU is done with its computations and memory copies, we need to synchronize it with the host.
- Using `cudaStreamSynchronize()` we can ask to the host to wait for the device to finish before proceeding.
- Some devices of compute capability 1.1 and higher can perform copies between page-locked host memory and device memory concurrently with kernel execution. Applications may query this capability by checking the `asyncEngineCount` device property.
- Some devices of compute capability 2.x and higher can execute multiple kernels concurrently. Applications may query this capability by checking the `concurrentKernels` device property.



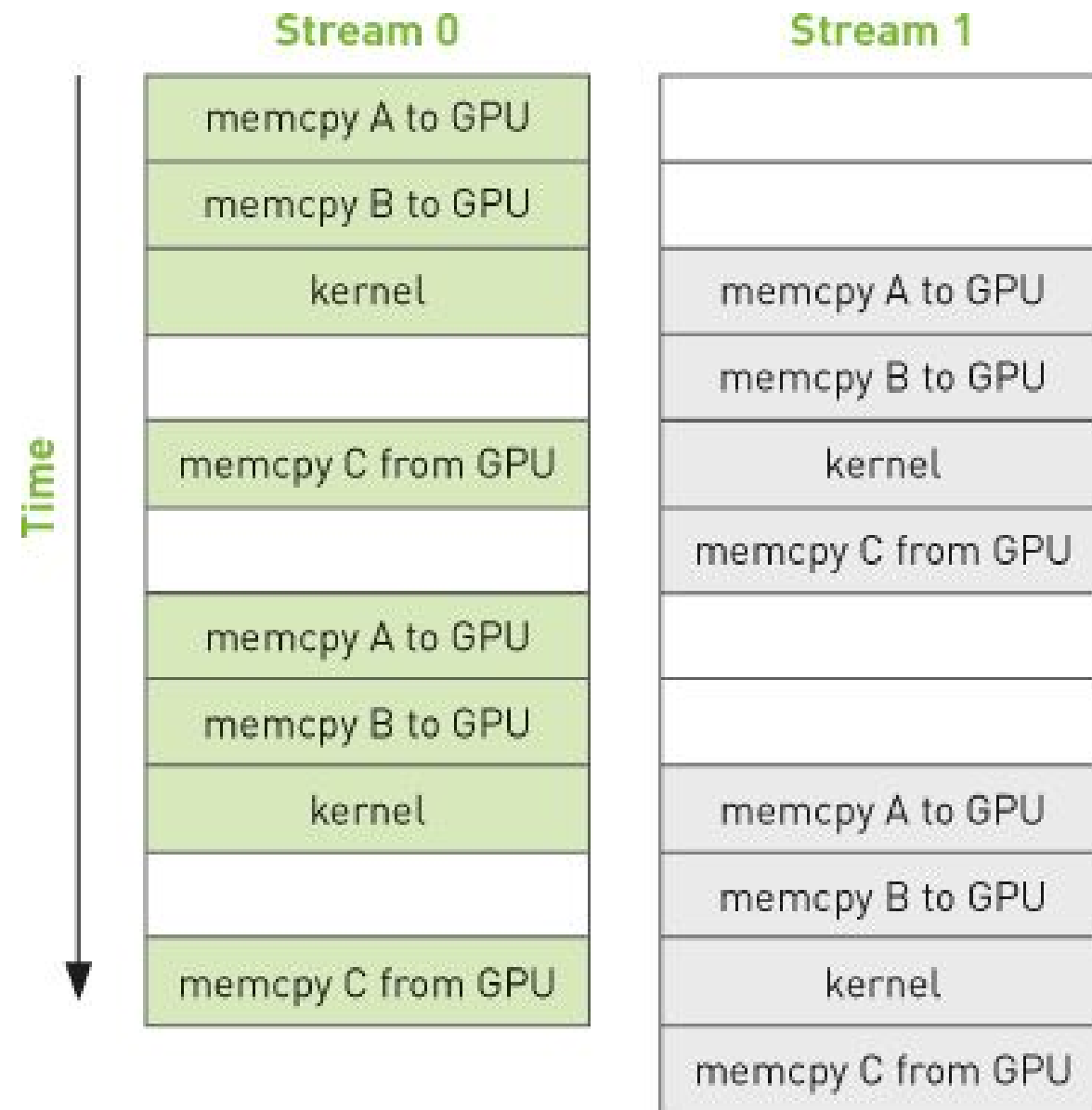
Example

- `03single_stream_chunks`: This example makes use of the `cudaMemcpyAsync()` function in order to take advantage of the asynchronous memory copies (overlaps).



Multiple Streams

- Different streams may execute their commands or host requests out of order with respect to another one, but the same stream is still a sequence of commands that are executed in order.
- Some devices support simultaneous kernel execution and two memory copies, one to the device, one from the device.





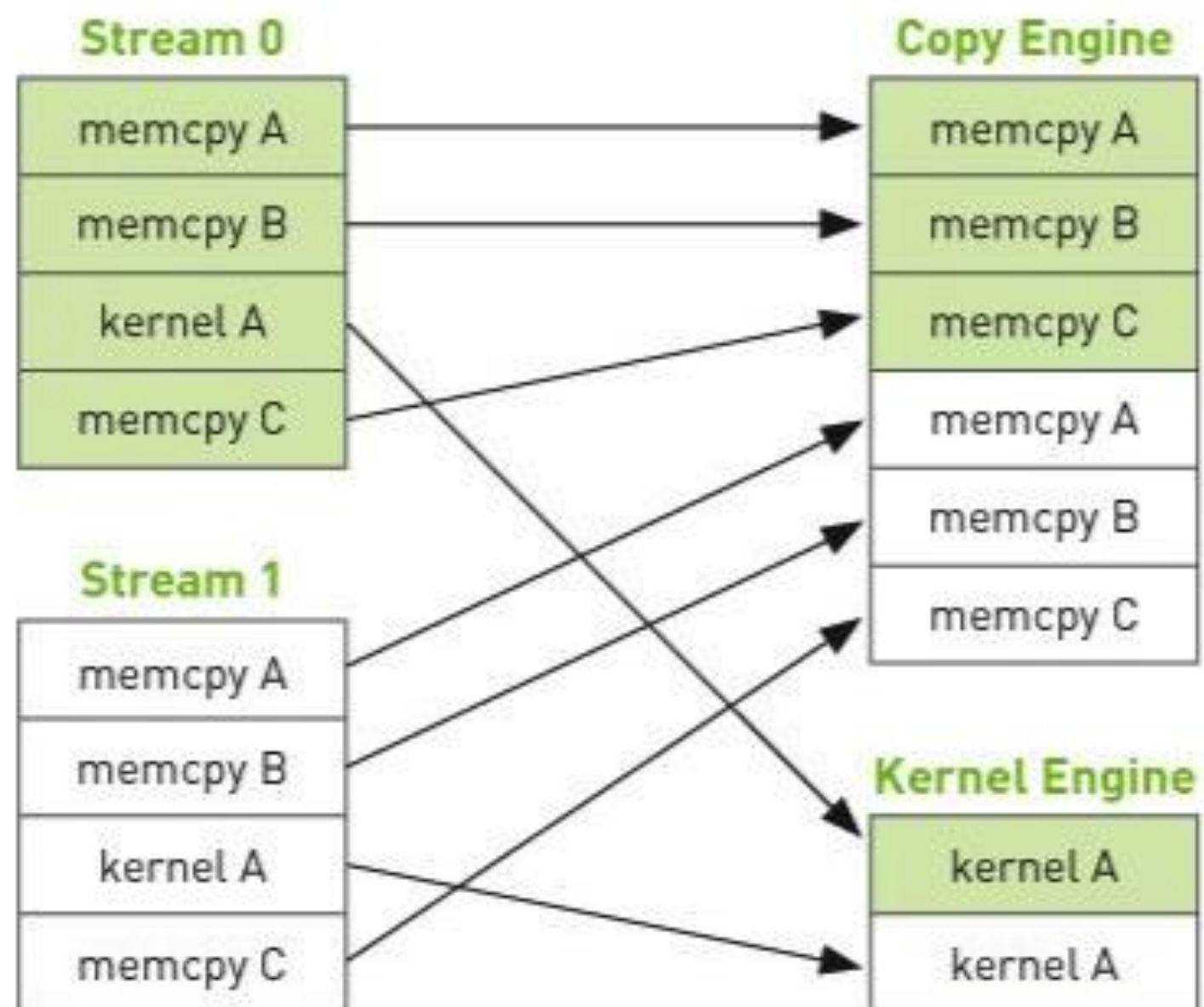
Example

- `04multiple_stream`: This example executes two streams to process two different jobs.

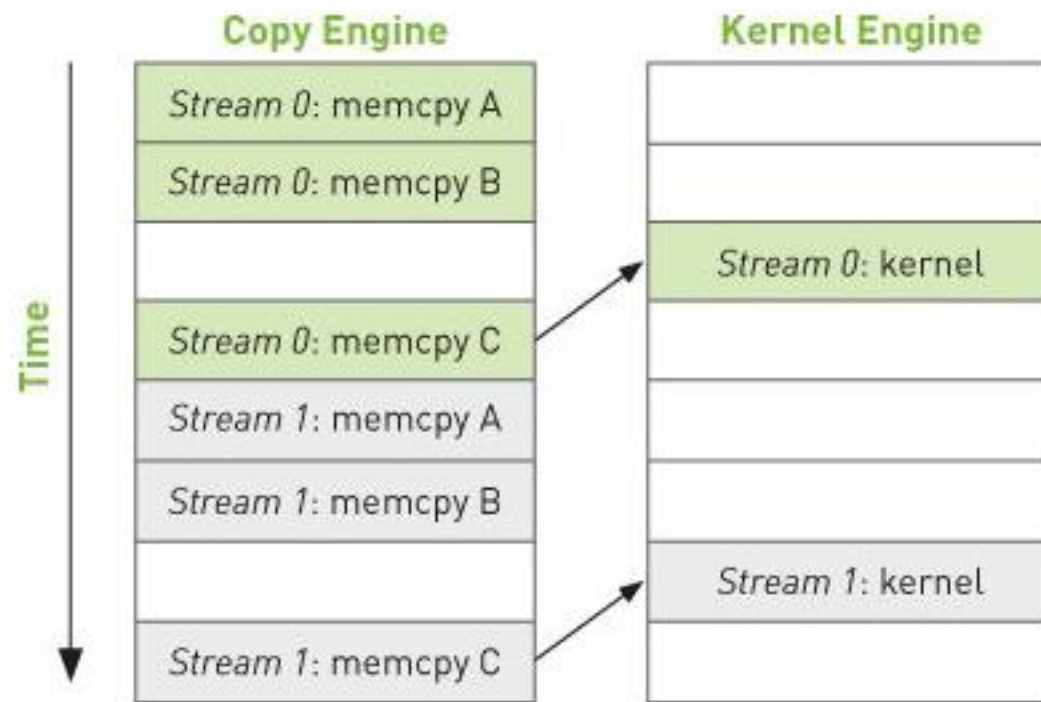


Work Scheduling

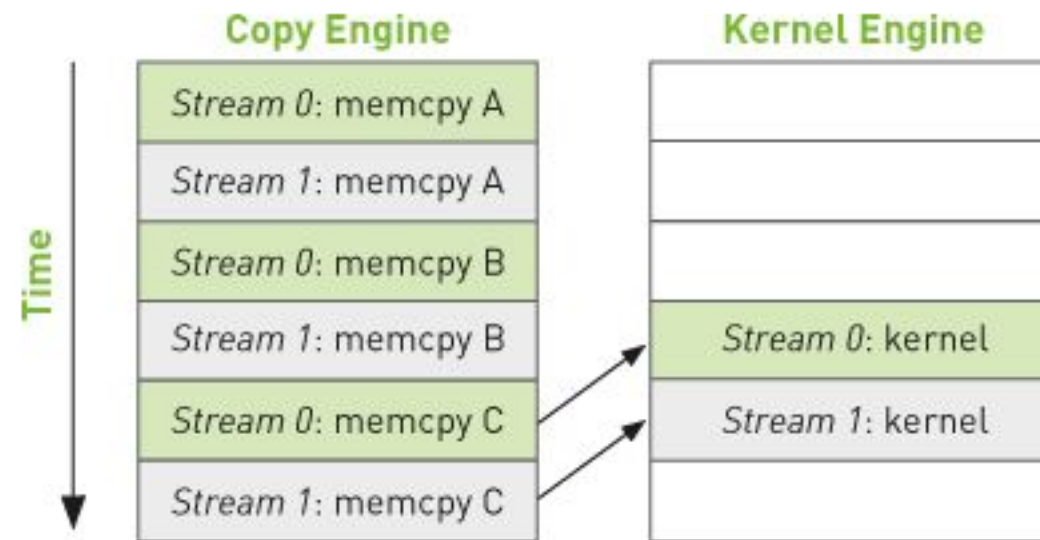
- On one hand, we know that the stream is a sequence of operations, memory copies and kernel executions.
- On the other hand, the device just has one or more engines to perform copies and an engine to execute kernels.
- However, there are dependencies specified by the order in which operations are added to the stream.



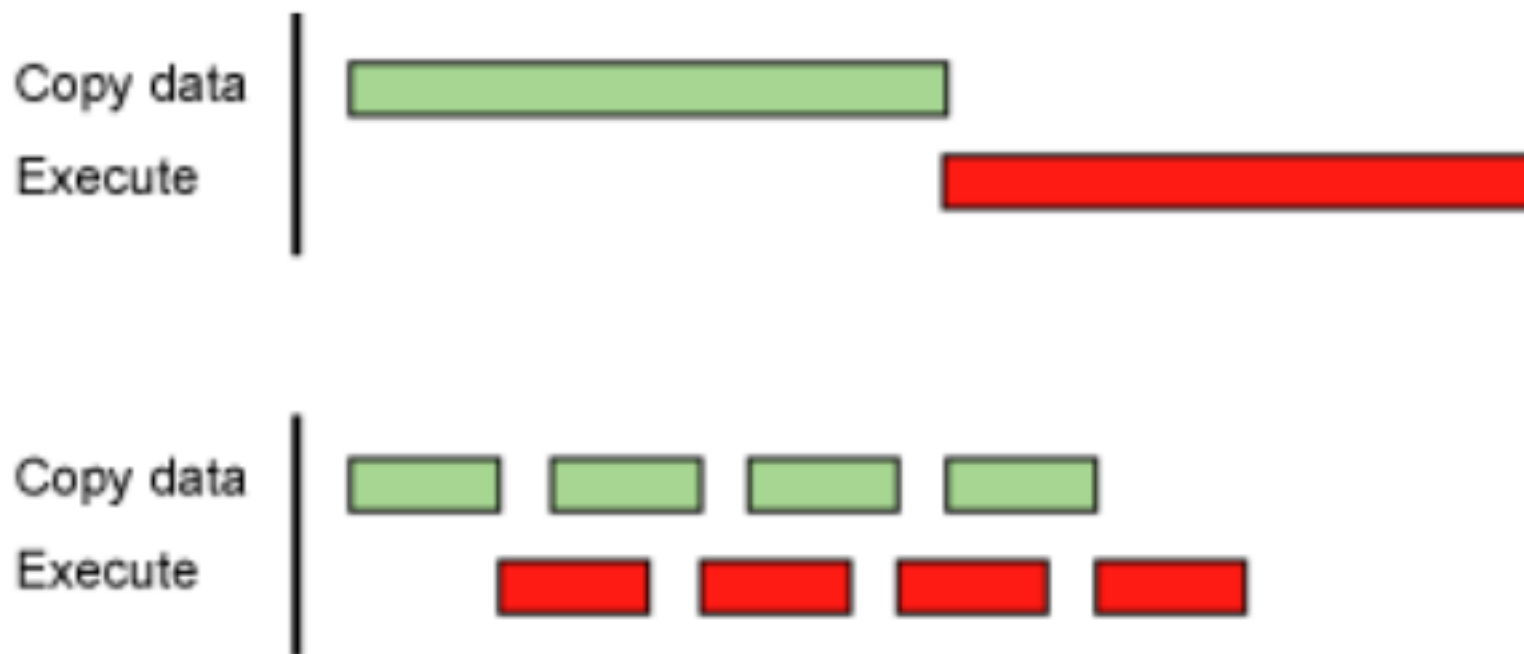
■ Wrong way



■ Correct way



- The amount of execution overlap between two streams depends on the order in which the commands are issued to each stream and whether or not the device supports overlap of data transfer and kernel execution.





Example

- `05multiple_stream_correct`: This example shows the correct use of streams.



Practice

- [06sieve_of_Eratosthenes.cu](#): This algorithm is used to find all the prime numbers up to a given boundary.

- To find all the prime numbers less than or equal to a given integer n :
 1. Create a list of consecutive integers from 2 to n : (2, 3, 4, ..., n).
 2. Initially, let p equal 2, the first prime number.
 3. Starting from p , count up in increments of p and mark each of these numbers greater than p itself in the list. These numbers will be $2p$, $3p$, $4p$, etc.; note that some of them may have already been marked.
 4. Find the first number greater than p in the list that is not marked. If there was no such number, stop. Otherwise, let p now equal this number (which is the next prime), and repeat from step 3.
- When the algorithm terminates, all the numbers in the list that are not marked are prime.