NVIDIA. DEEP LEARNING INSTITUTE

MULTI-GPU PROGRAMMING FOR CUDA C++

COPY/COMPUTE OVERLAP CONSIDERATIONS

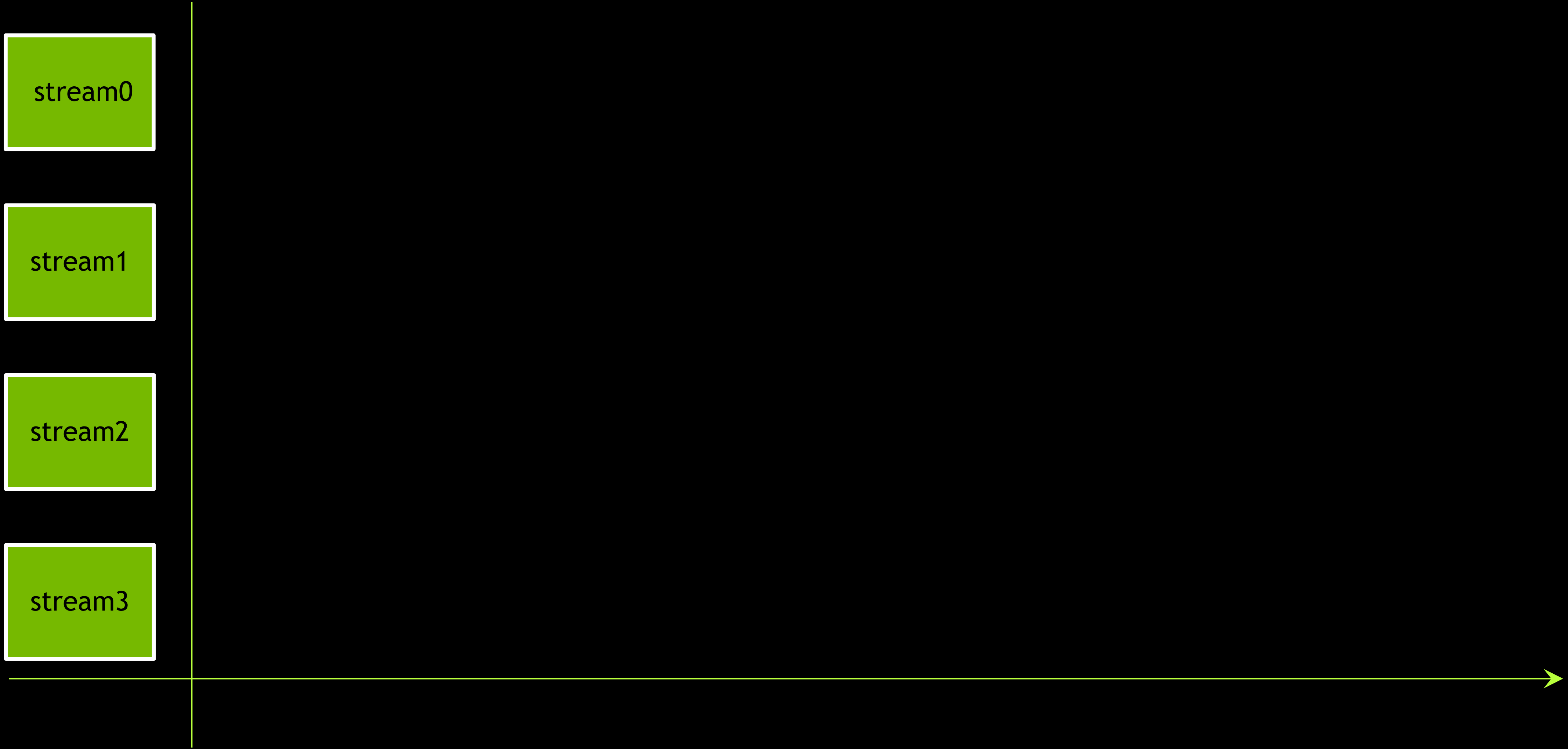
COPY/COMPUTE OVERLAP CONSIDERATIONS

Copy/Compute Overlap with Streams

Copy/Compute Overlap Indexing

COPY/COMPUTE OVERLAP WITH STREAMS

Using the default stream, a typical 3-step CUDA program will perform HtoD copy, compute, and DtoH copy serially

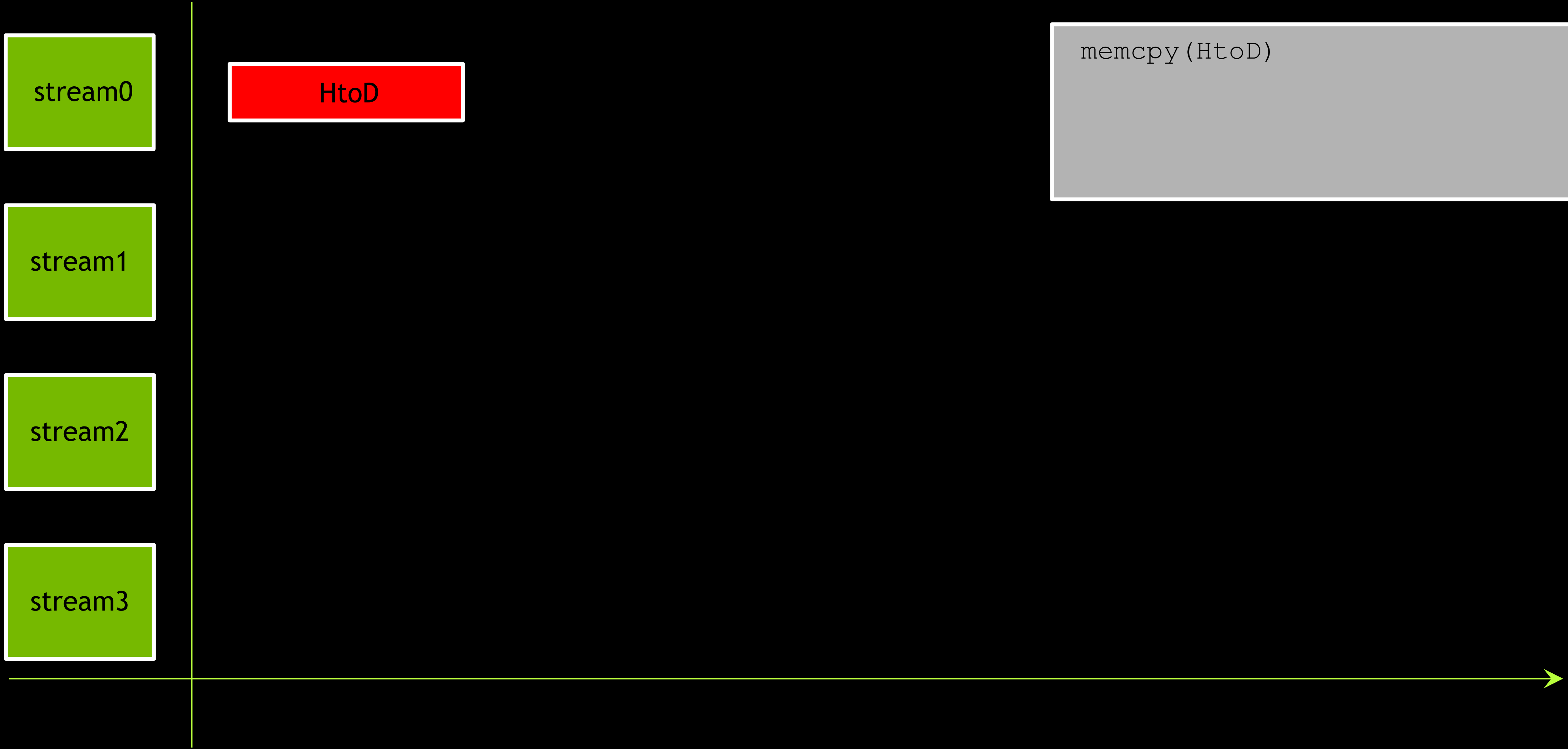


Using the default stream, a typical 3-step CUDA program will perform HtoD copy, compute, and DtoH copy serially

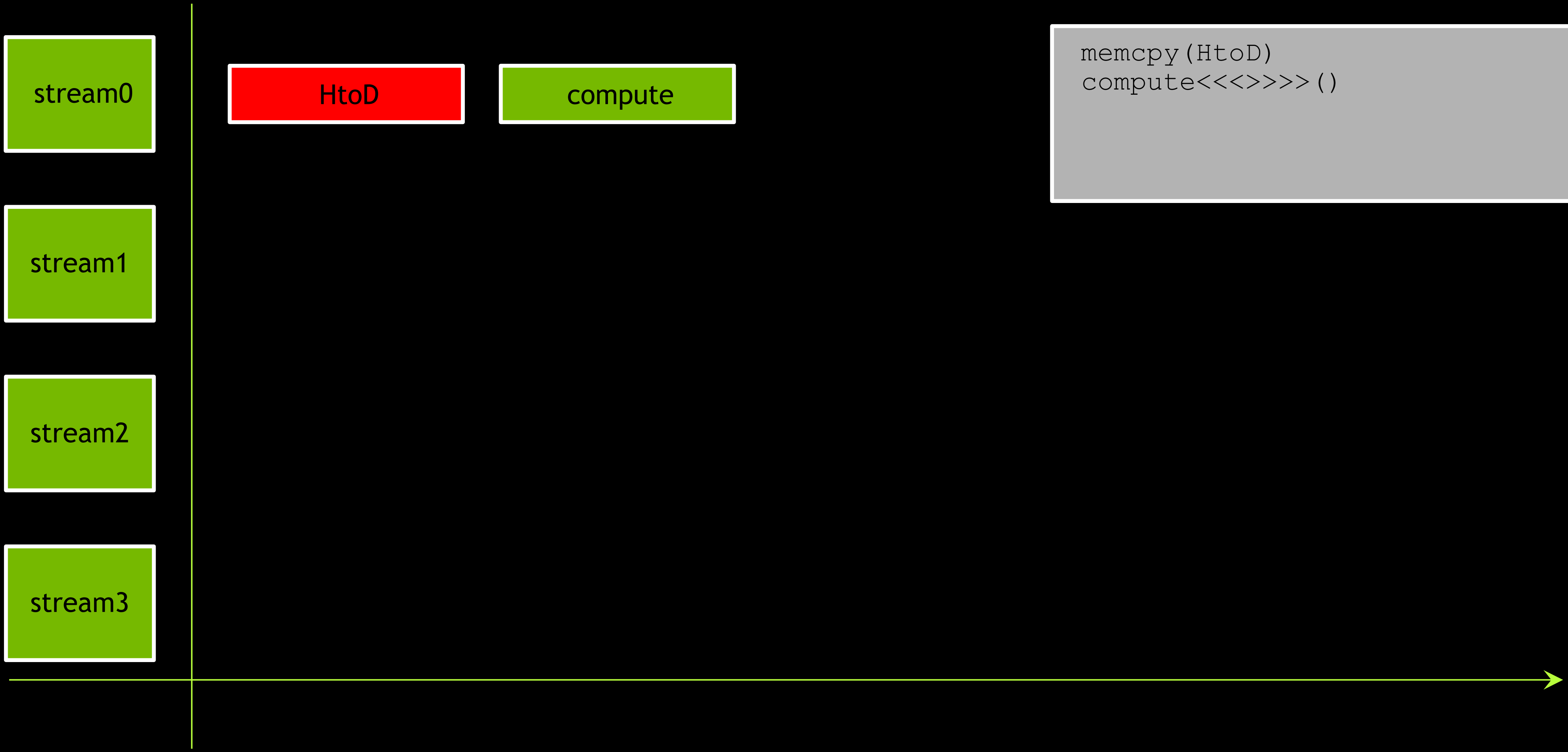

```
memcpy(HtoD)
```

Note: we will be using shorthand code in these slides for ease of presentation

```
memcpy(HtoD)
```

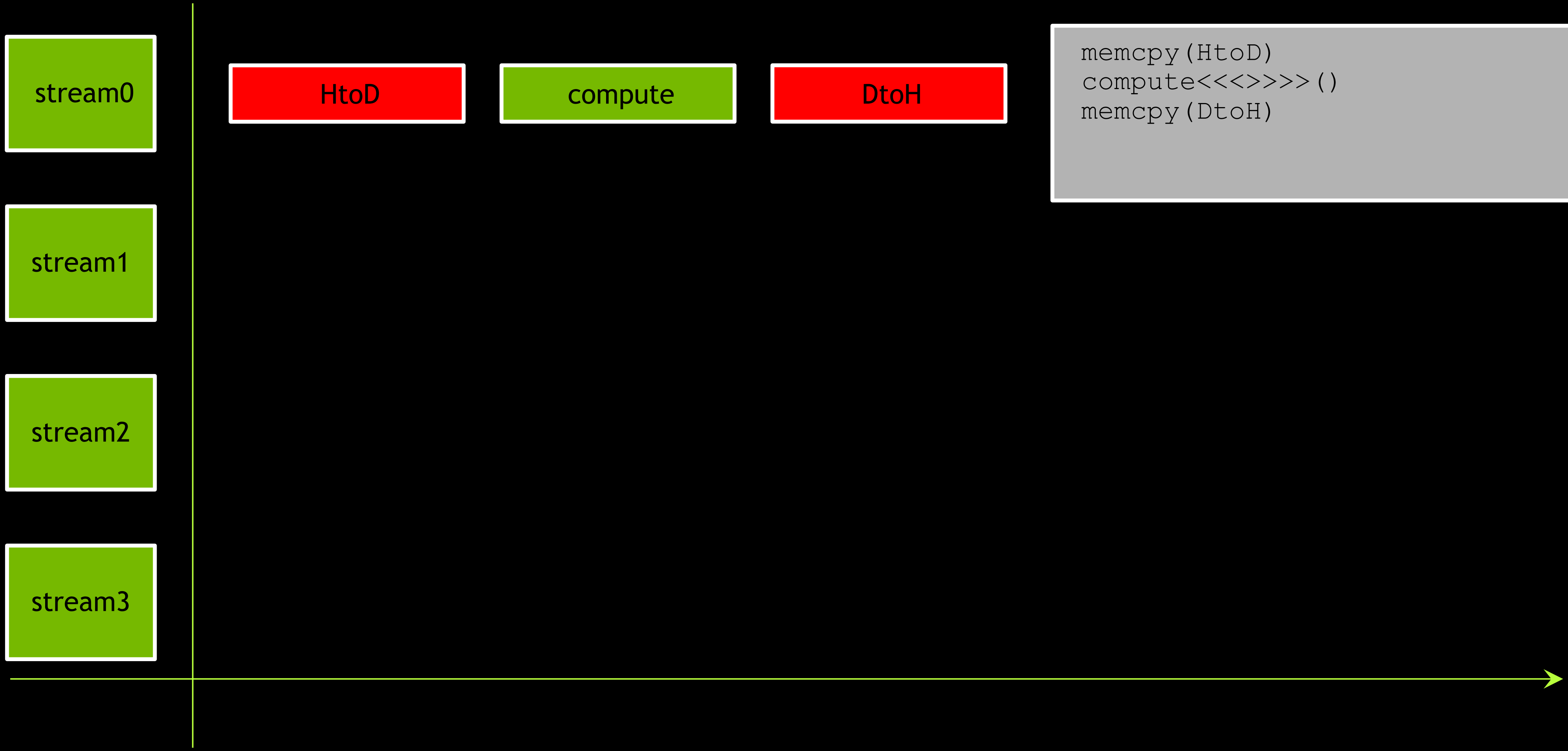


Using the default stream, a typical 3-step CUDA program will perform HtoD copy, compute, and DtoH copy serially



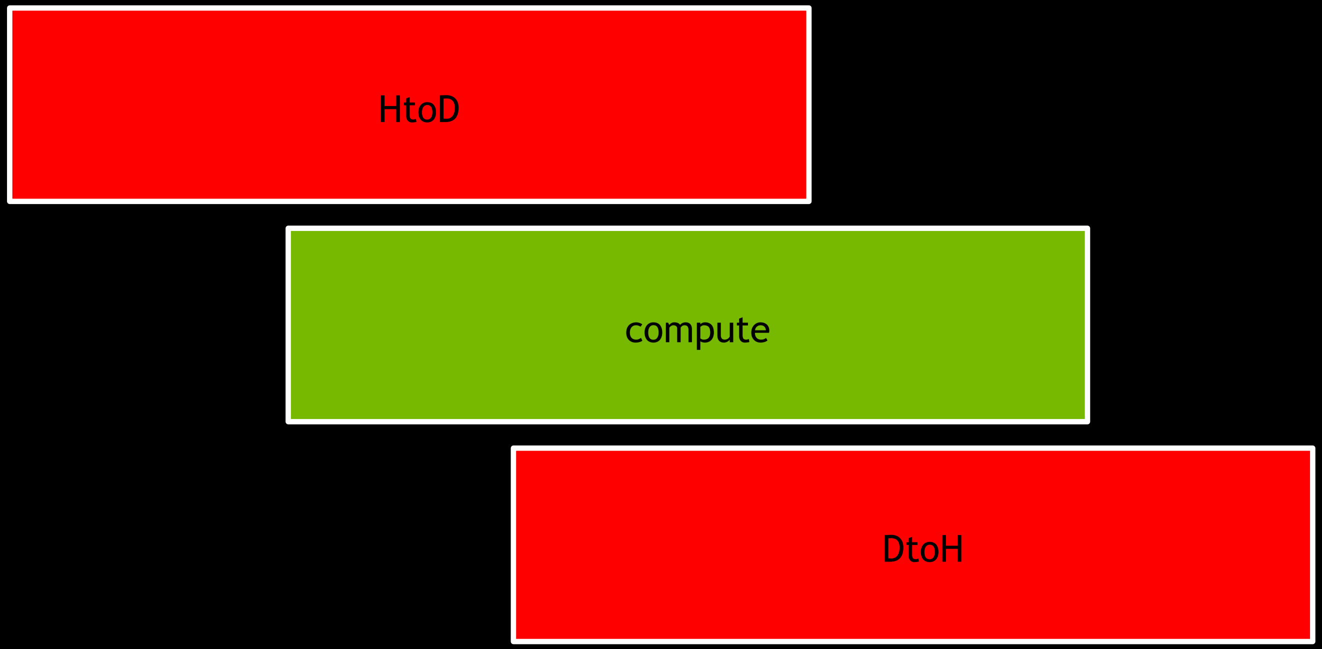

```
memcpy(HtoD)
compute<<<>>>()
```

Using the default stream, a typical 3-step CUDA program will perform HtoD copy, compute, and DtoH copy serially

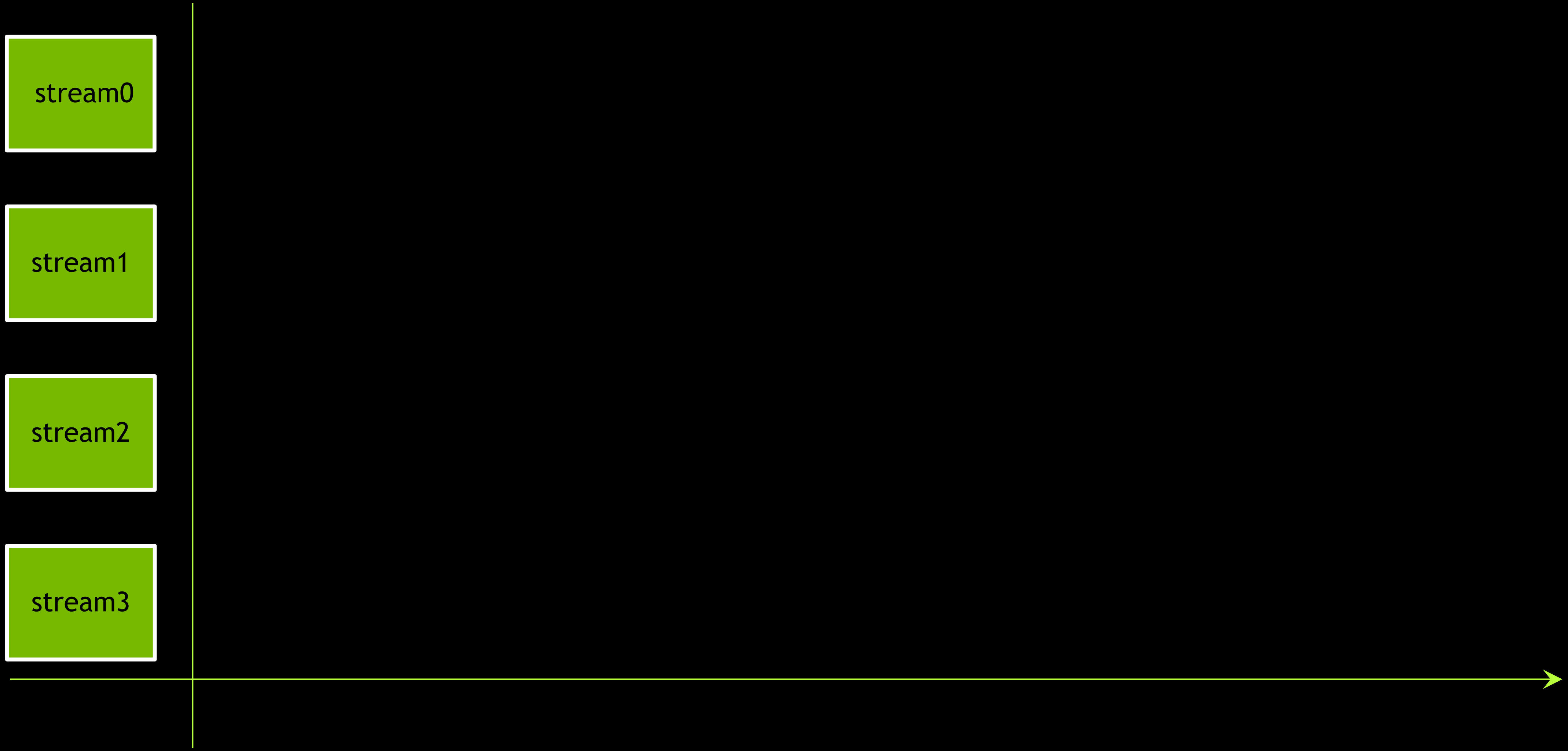



```
memcpy(HtoD)
compute<<<>>>()
memcpy(DtoH)
```

Let’s consider how we might perform copy/compute overlap

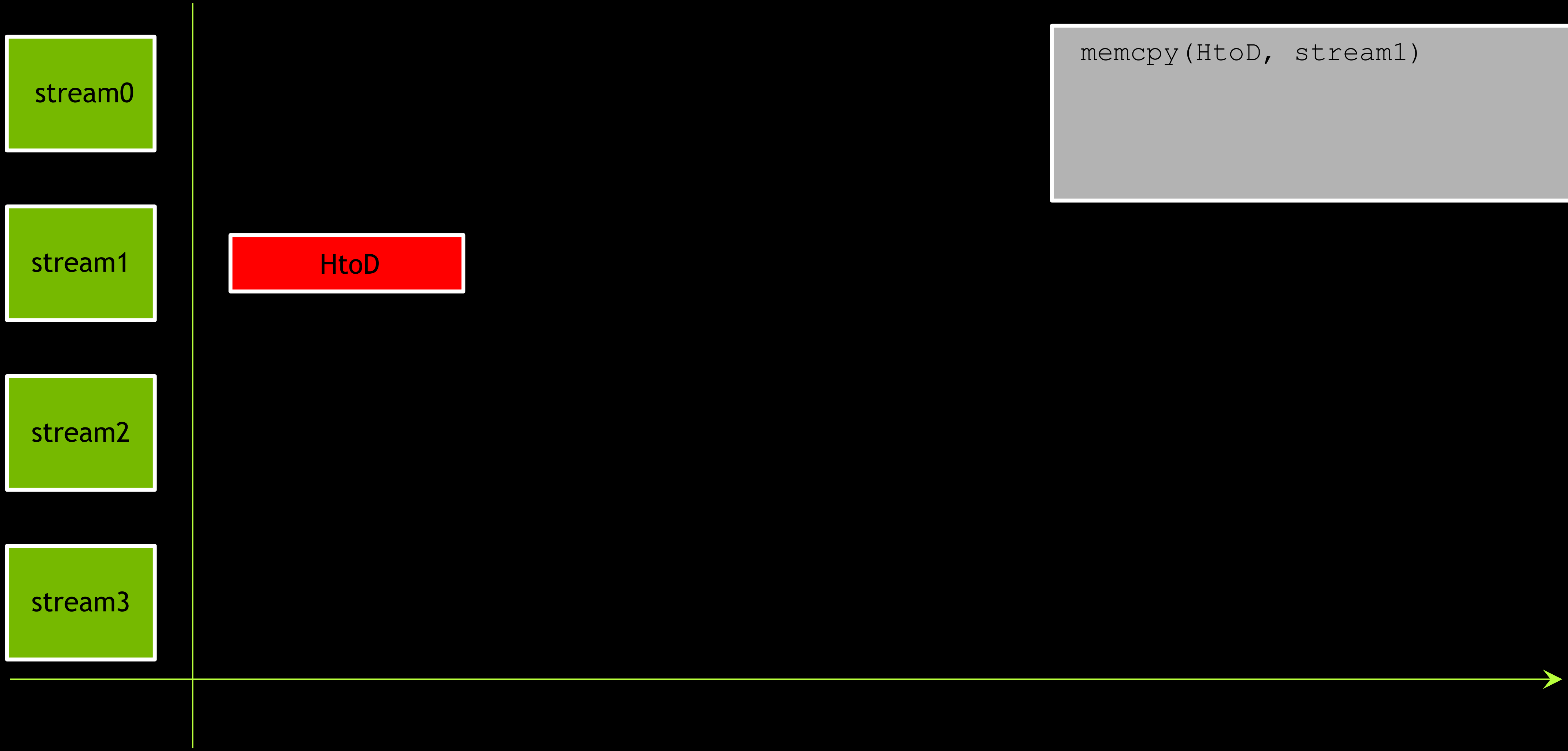


One naïve approach might be to simply issue each of these 3 operations in different non-default streams



One naïve approach might be to simply issue each of these 3 operations in different non-default streams

```
memcpy(HtoD, stream1)
```

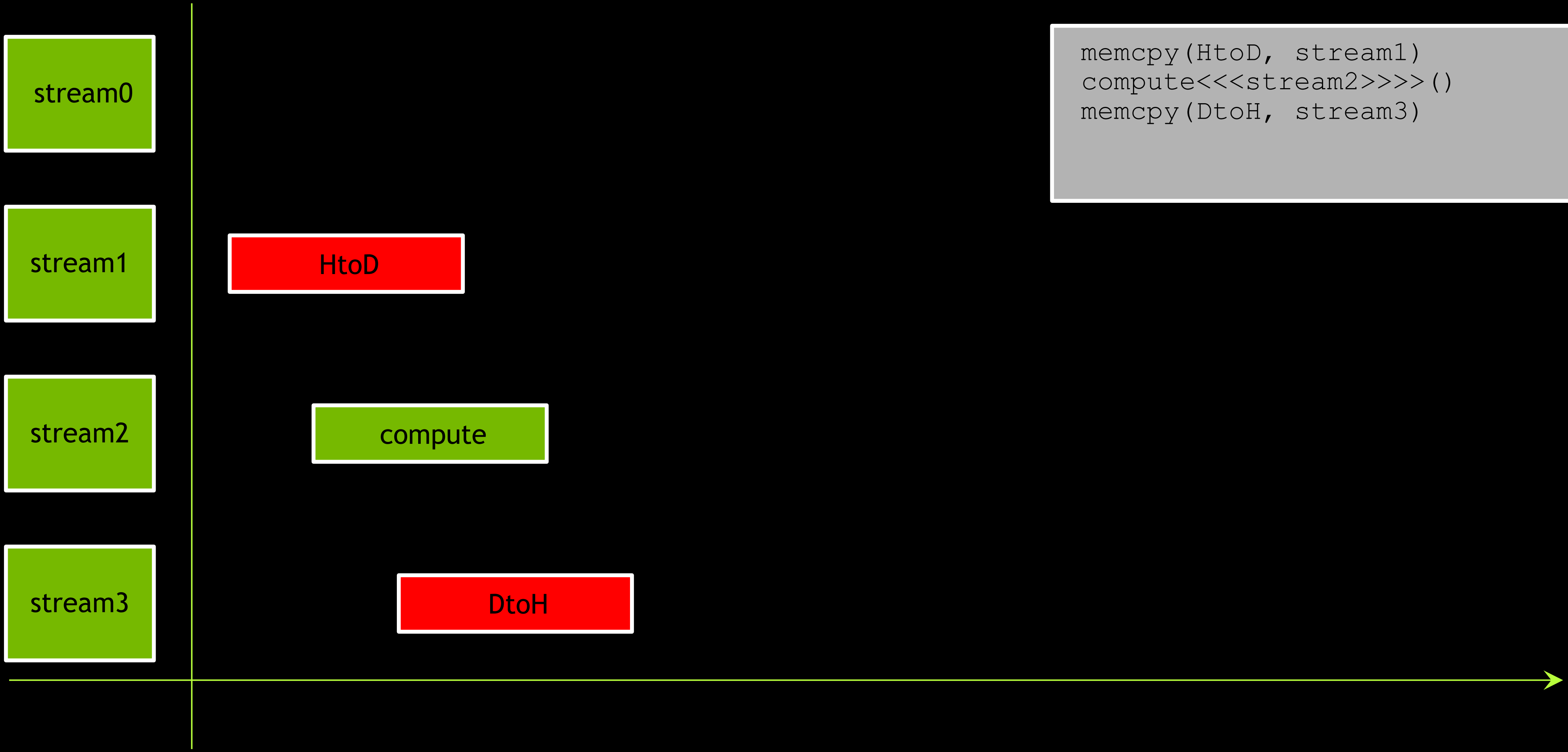


One naïve approach might be to simply issue each of these 3 operations in different non-default streams

```
memcpy(HtoD, stream1)
compute<<<stream2>>>>()
```

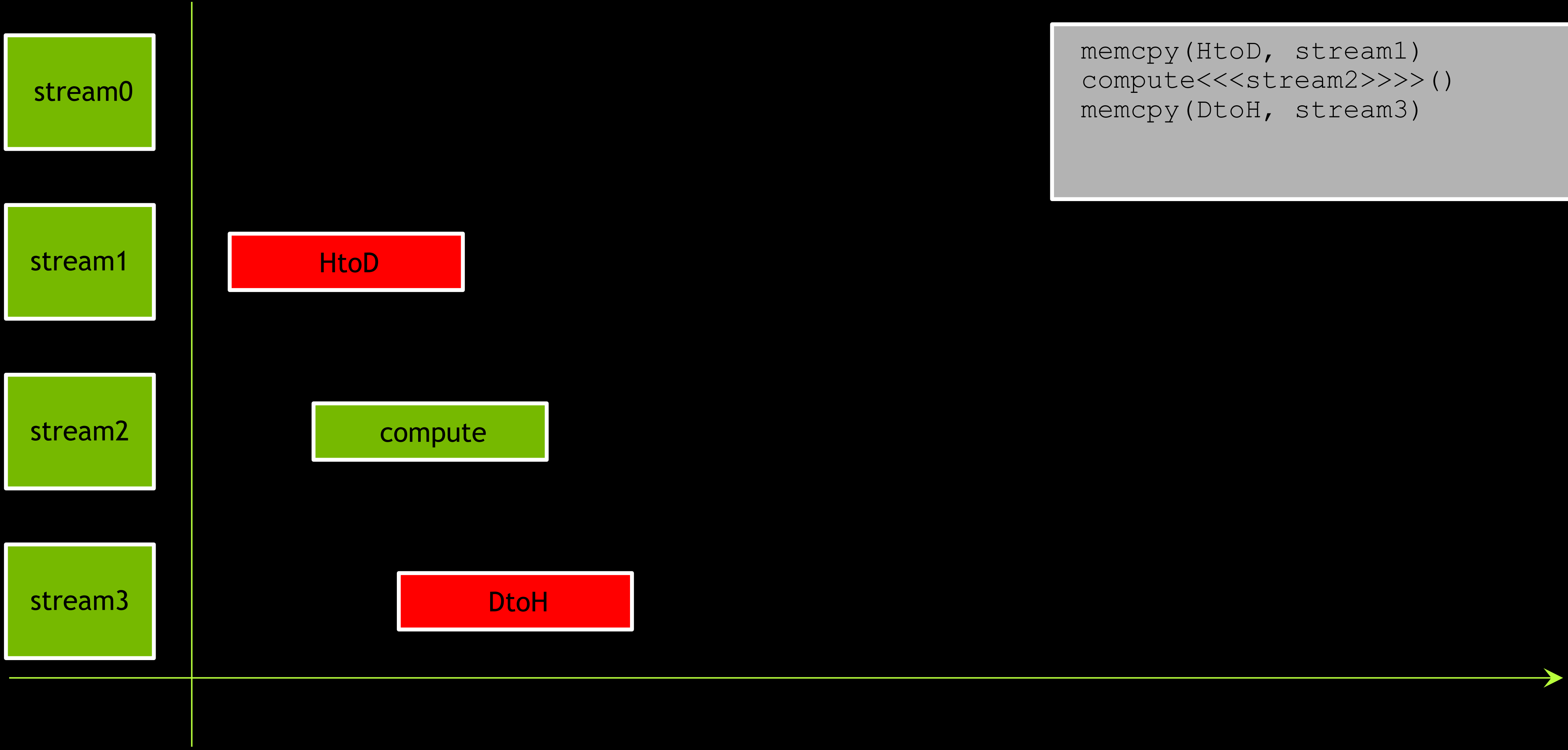
One naïve approach might be to simply issue each of these 3 operations in different non-default streams

```
memcpy(HtoD, stream1)
compute<<<stream2>>>>()
memcpy(DtoH, stream3)
```



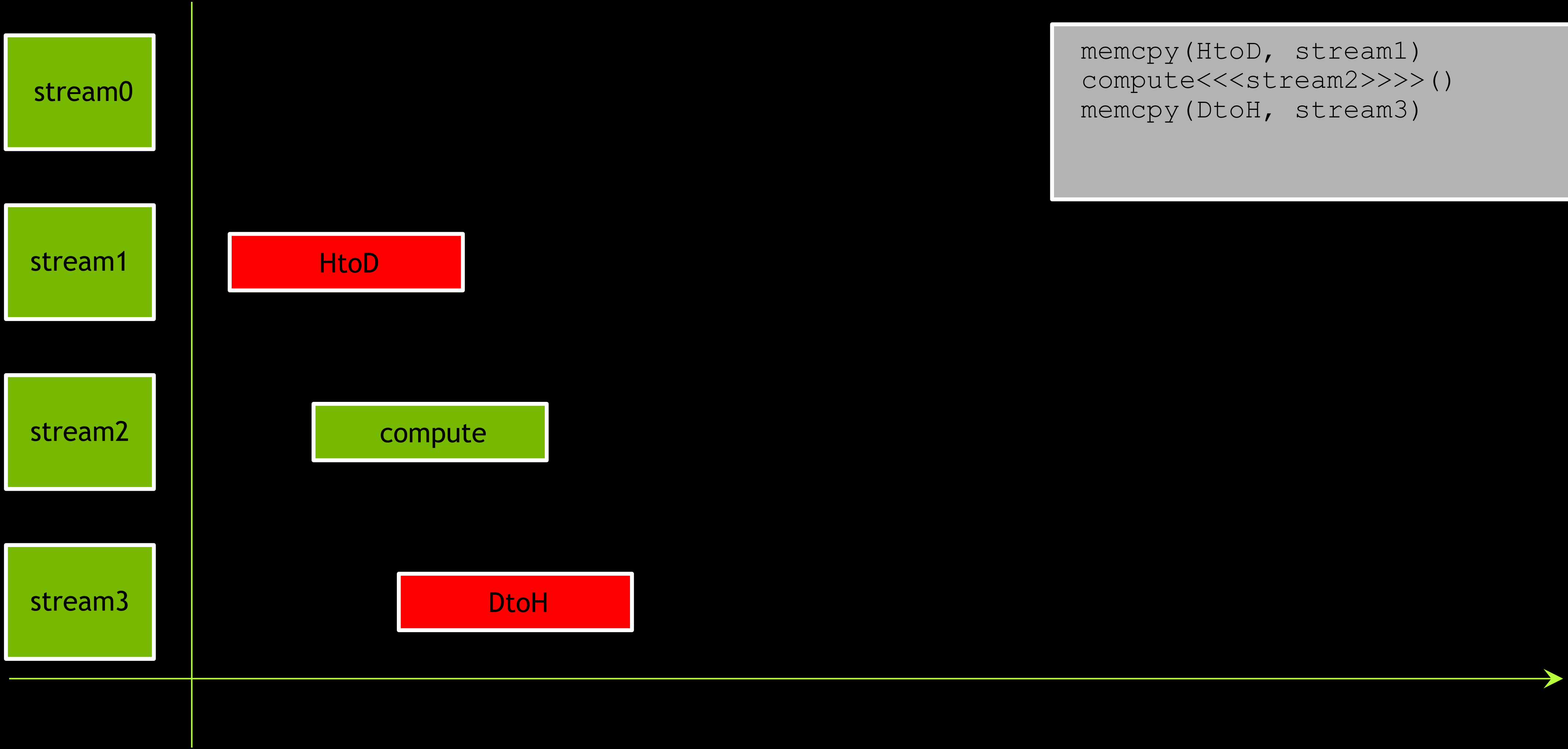
Would this work?

```
memcpy(HtoD, stream1)
compute<<<stream2>>>>()
memcpy(DtoH, stream3)
```



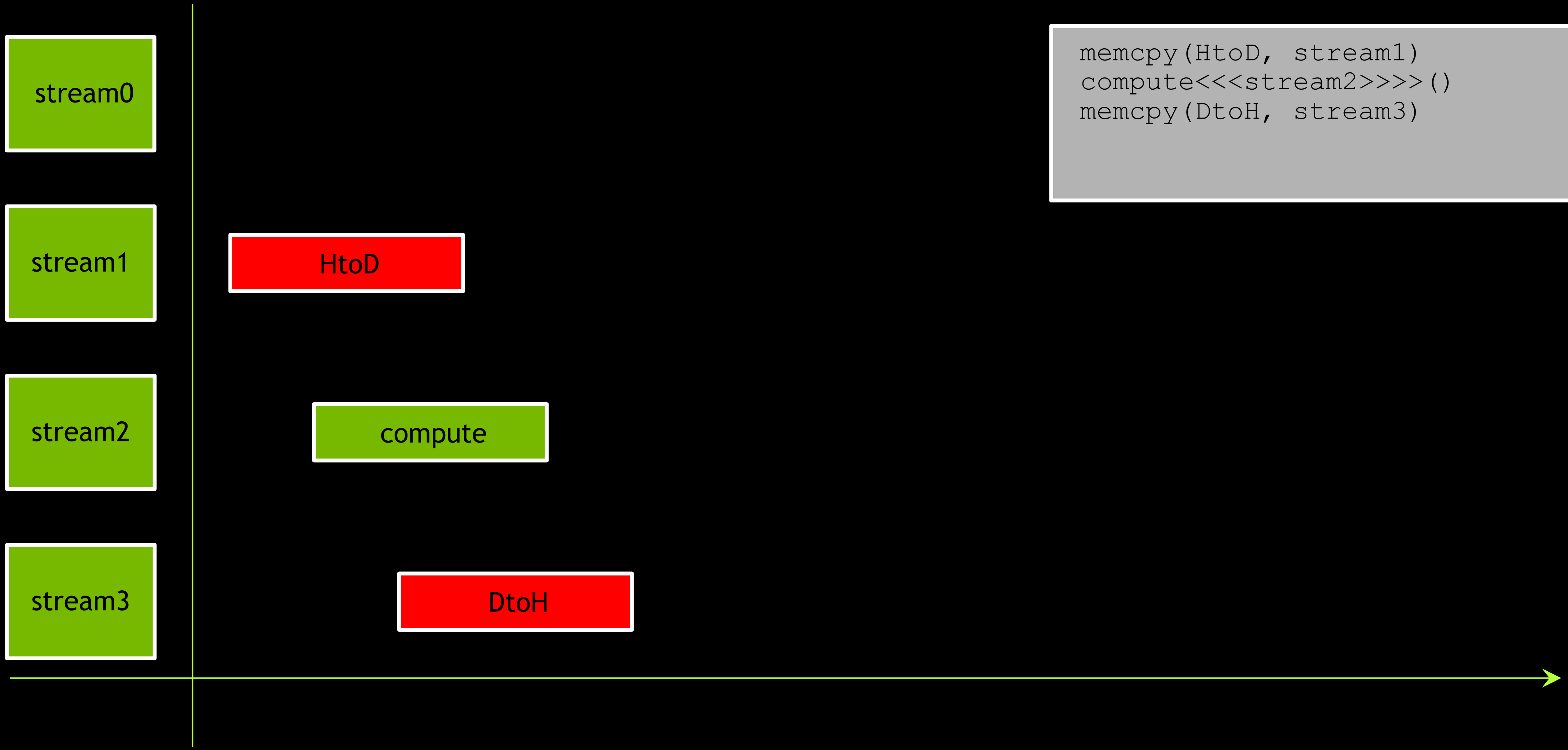
Would this work?
No

```
memcpy(HtoD, stream1)
compute<<<stream2>>>>()
memcpy(DtoH, stream3)
```



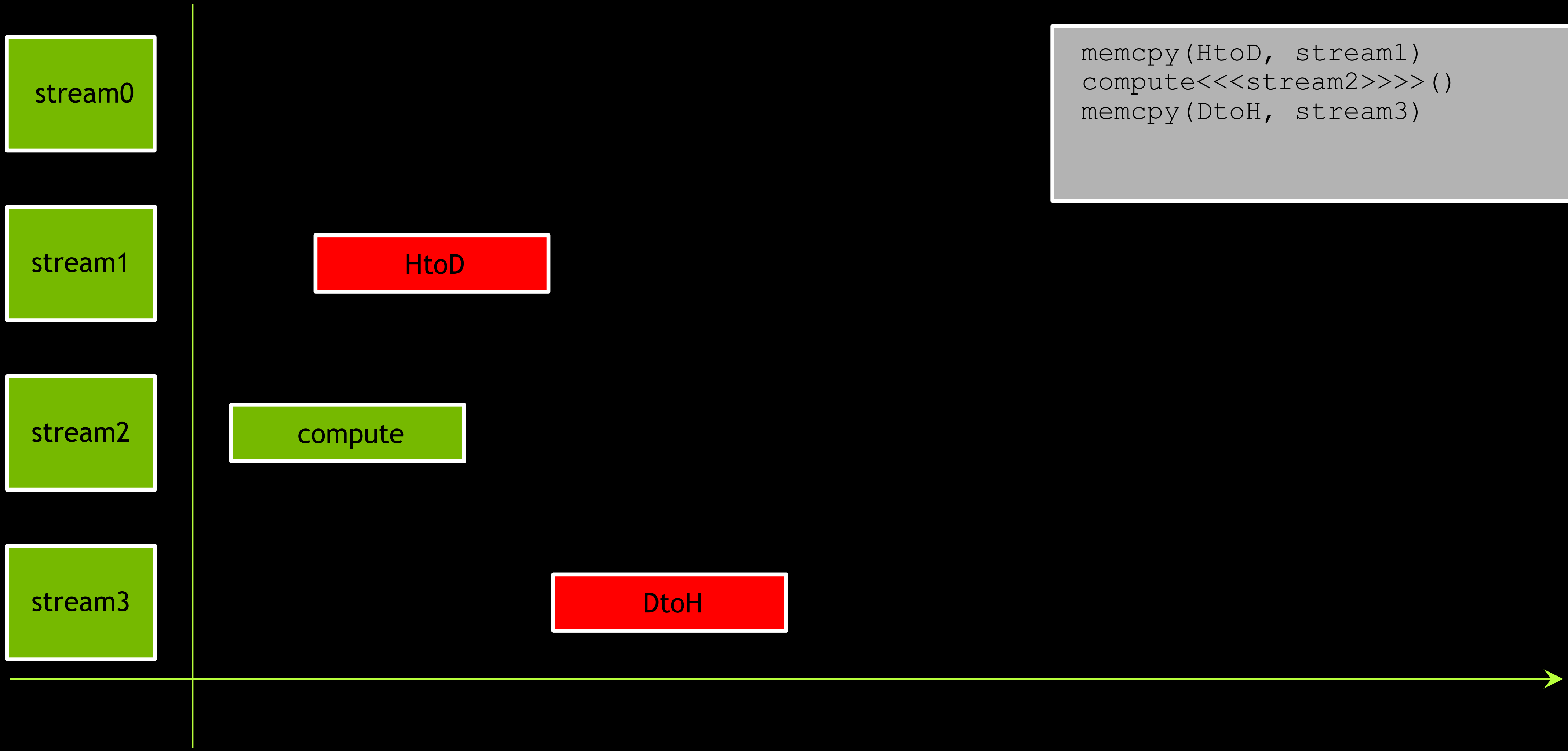
Recall that operations in non-default streams have no guaranteed order, therefore…

```
memcpy(HtoD, stream1)
compute<<<stream2>>>>()
memcpy(DtoH, stream3)
```



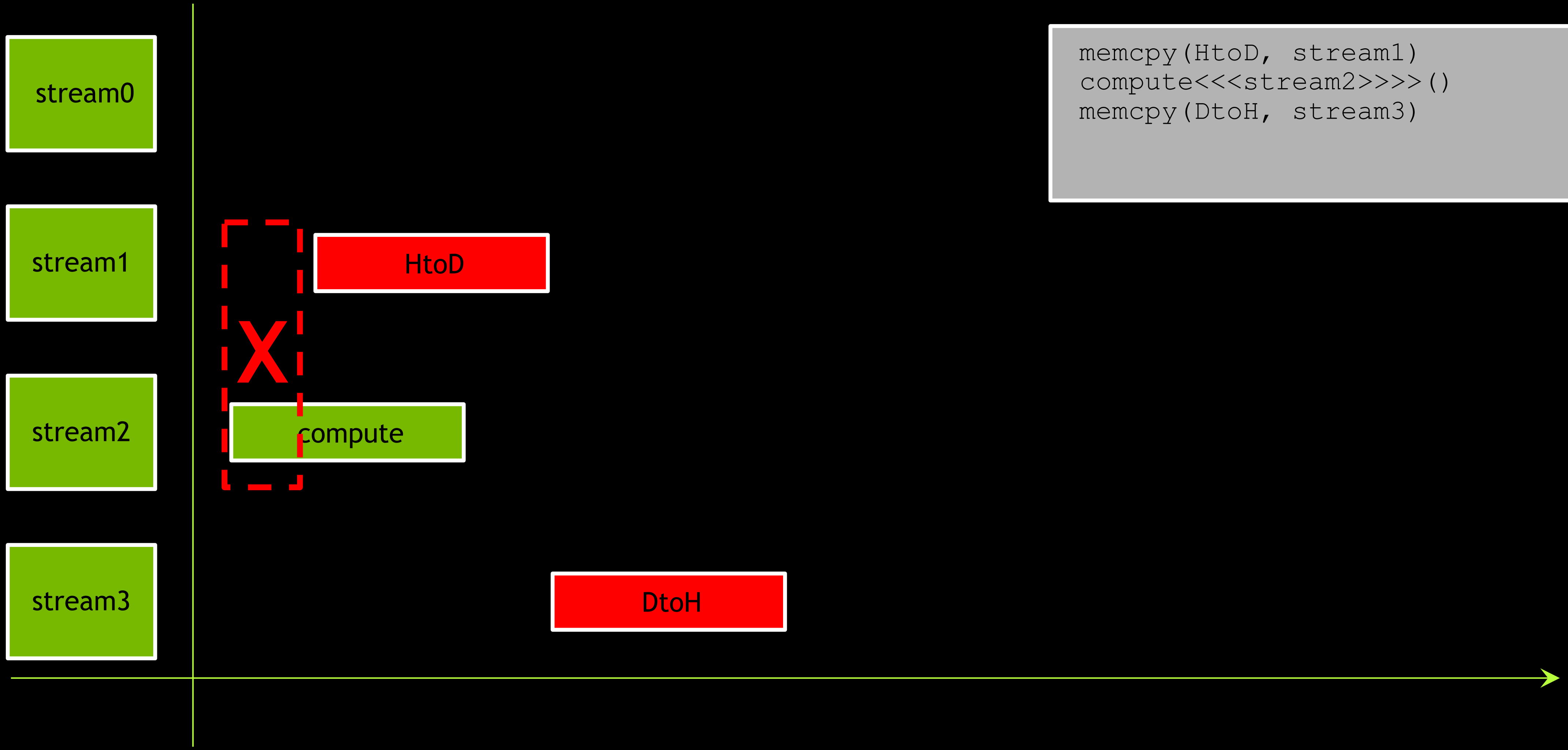
…something like this could occur

```
memcpy(HtoD, stream1)
compute<<<stream2>>>>()
memcpy(DtoH, stream3)
```

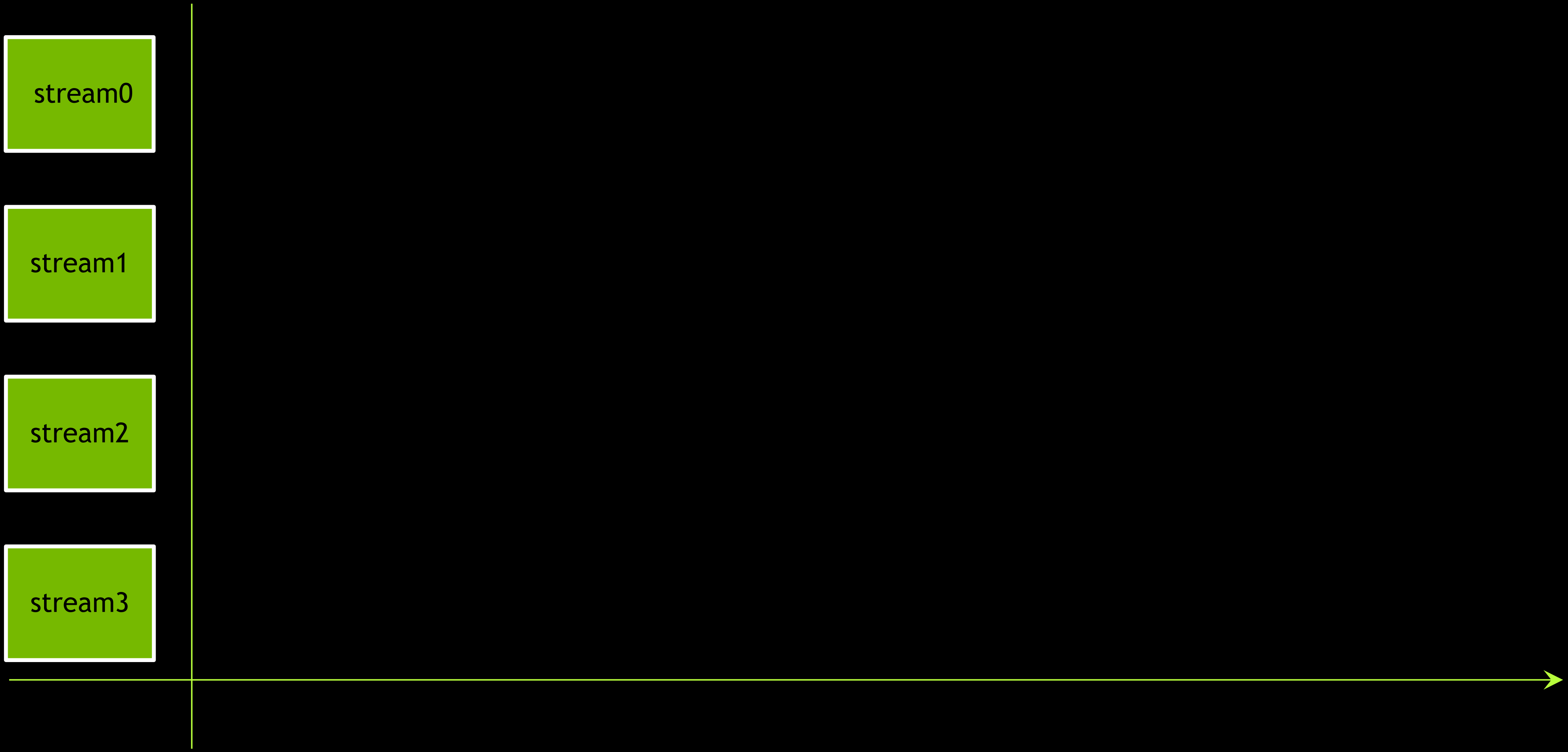


…and compute might begin before the data it needs is present on the GPU

```
memcpy(HtoD, stream1)
compute<<<stream2>>>>()
memcpy(DtoH, stream3)
```

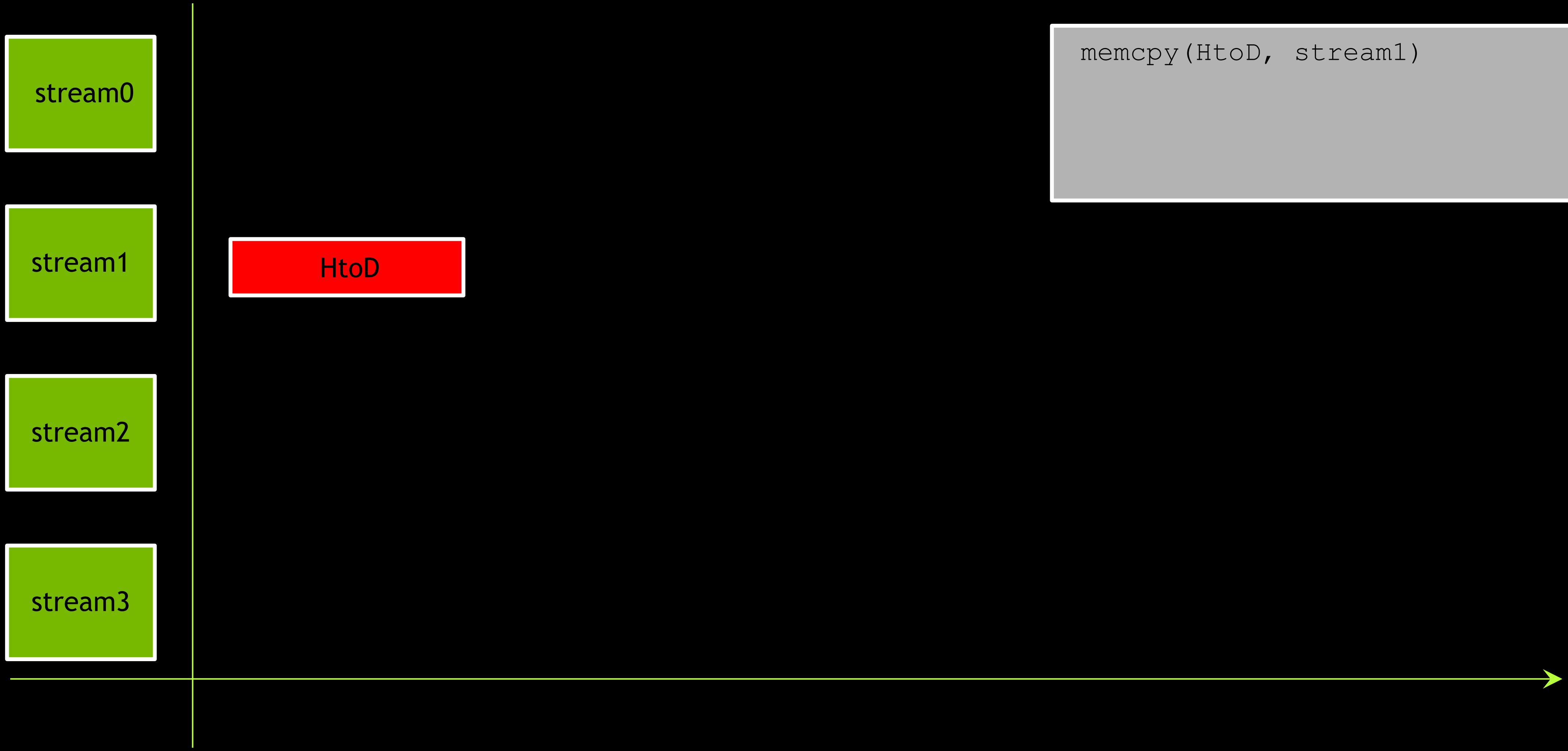


Another naïve approach might be to issue them all in the same non-default stream, to guarantee data/compute order



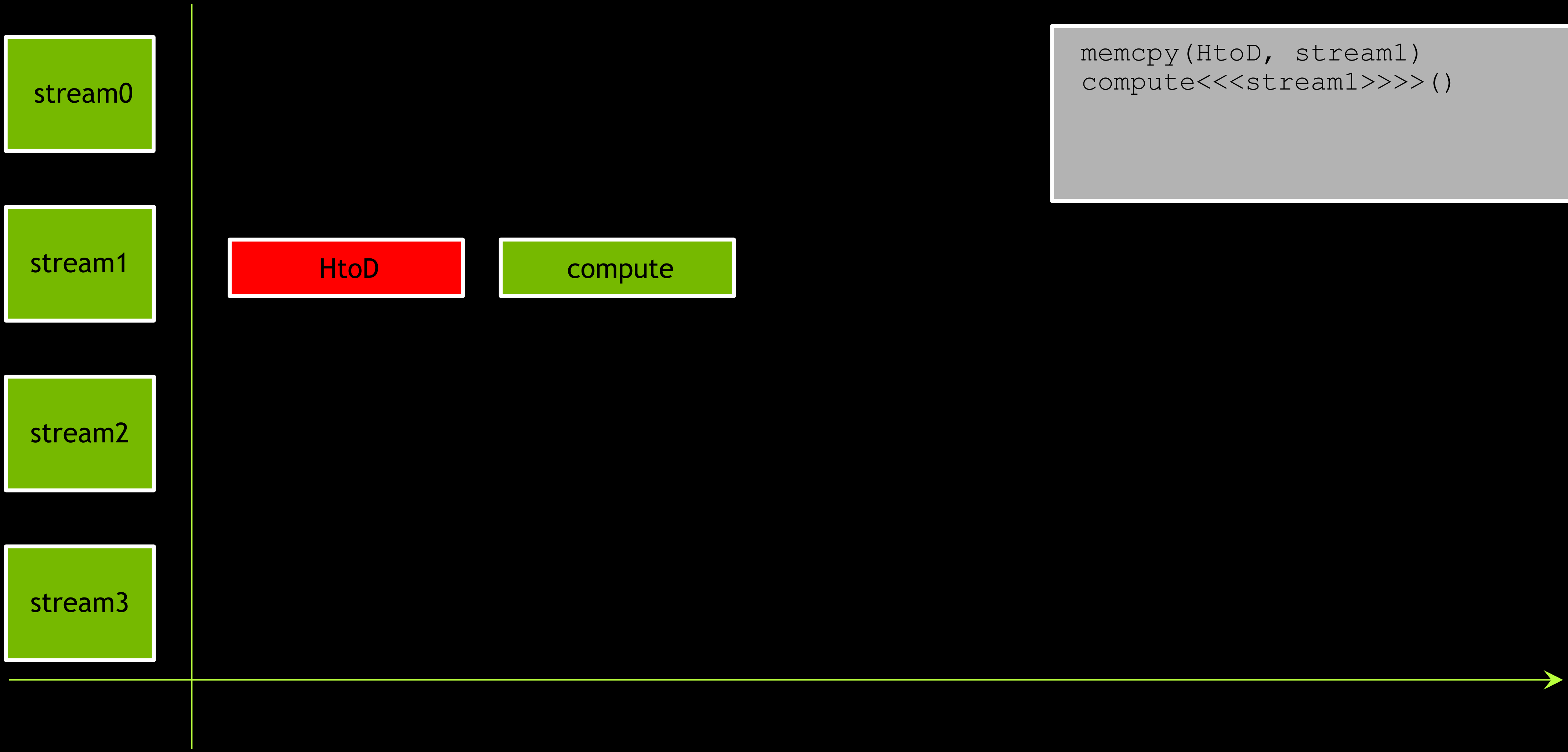
Another naïve approach might be to issue them all in the same non-default stream, to guarantee data/compute order

```
memcpy(HtoD, stream1)
```



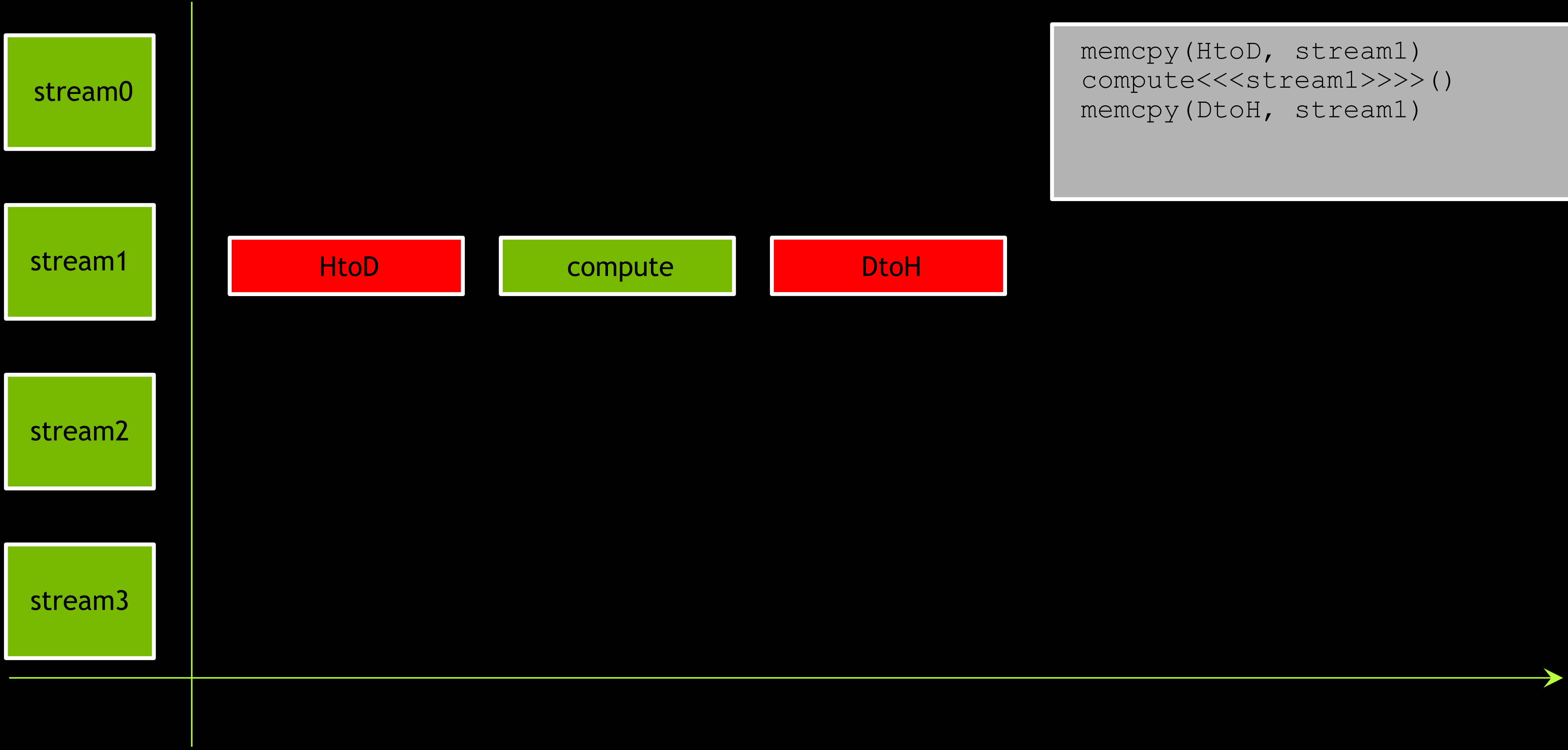
Another naïve approach might be to issue them all in the same non-default stream, to guarantee data/compute order

```
memcpy(HtoD, stream1)
compute<<<stream1>>>>()
```



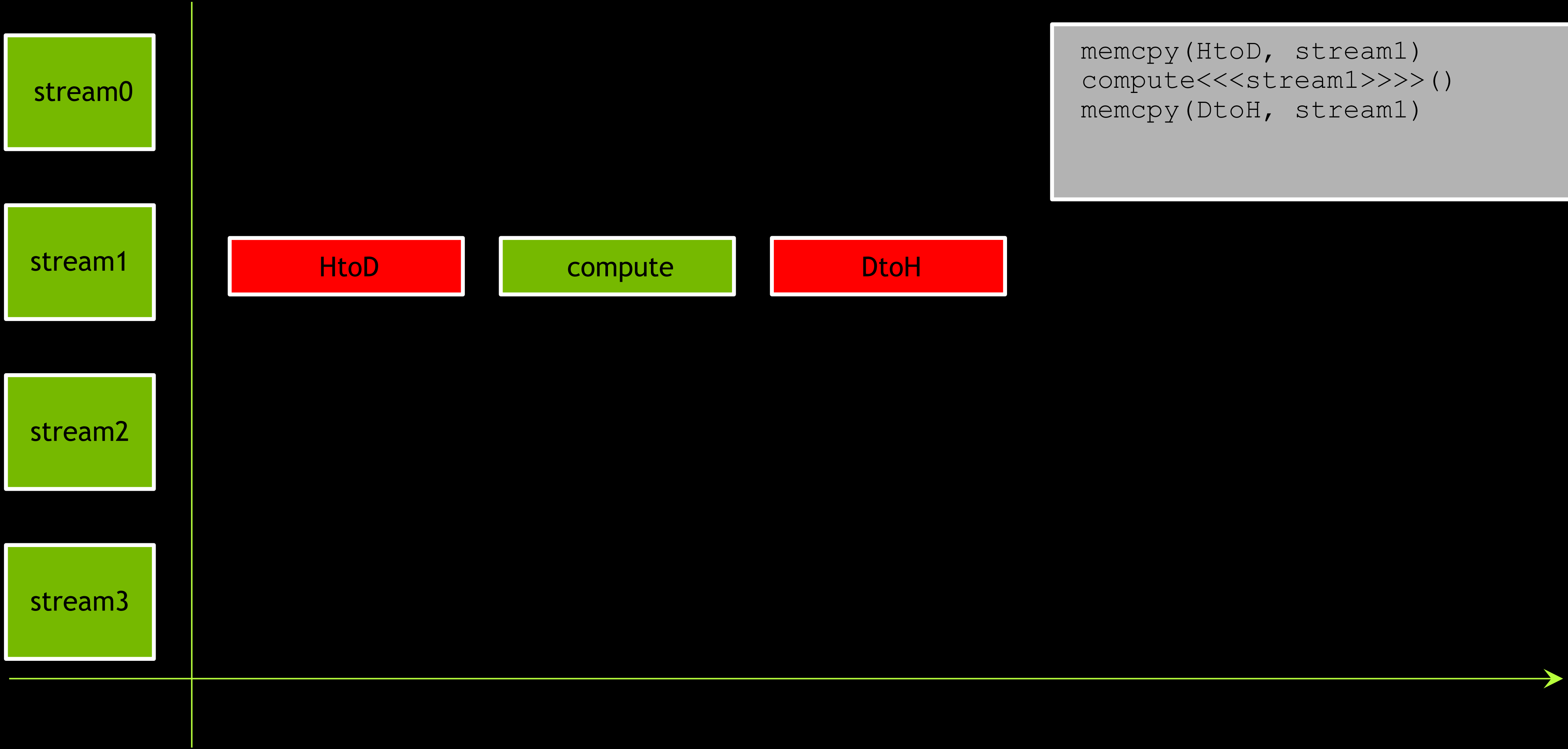
Another naïve approach might be to issue them all in the same non-default stream, to guarantee data/compute order

```
memcpy(HtoD, stream1)
compute<<<stream1>>>>()
memcpy(DtoH, stream1)
```

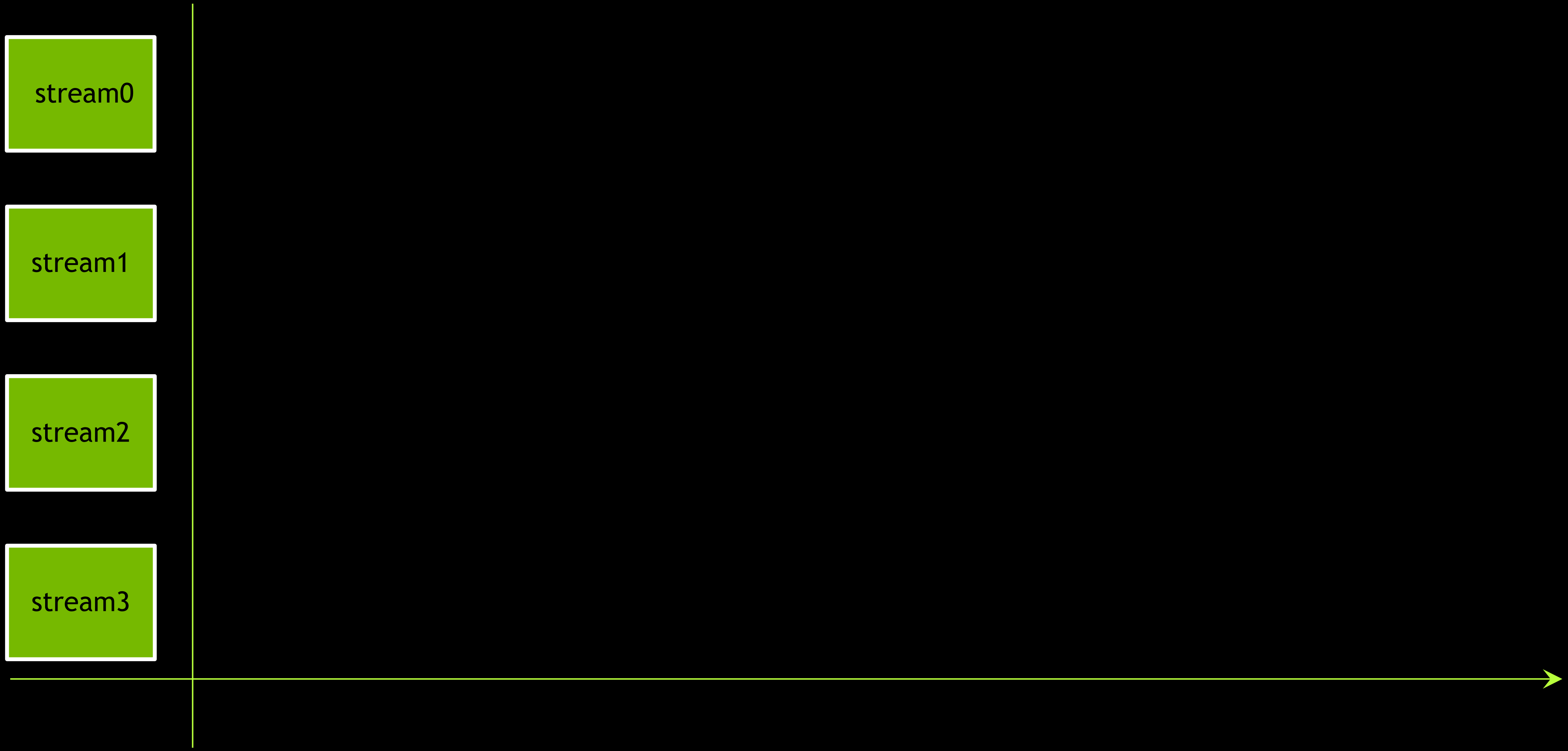


However, this results in the same behavior as using the default stream: no overlap

```
memcpy(HtoD, stream1)
compute<<<stream1>>>>()
memcpy(DtoH, stream1)
```

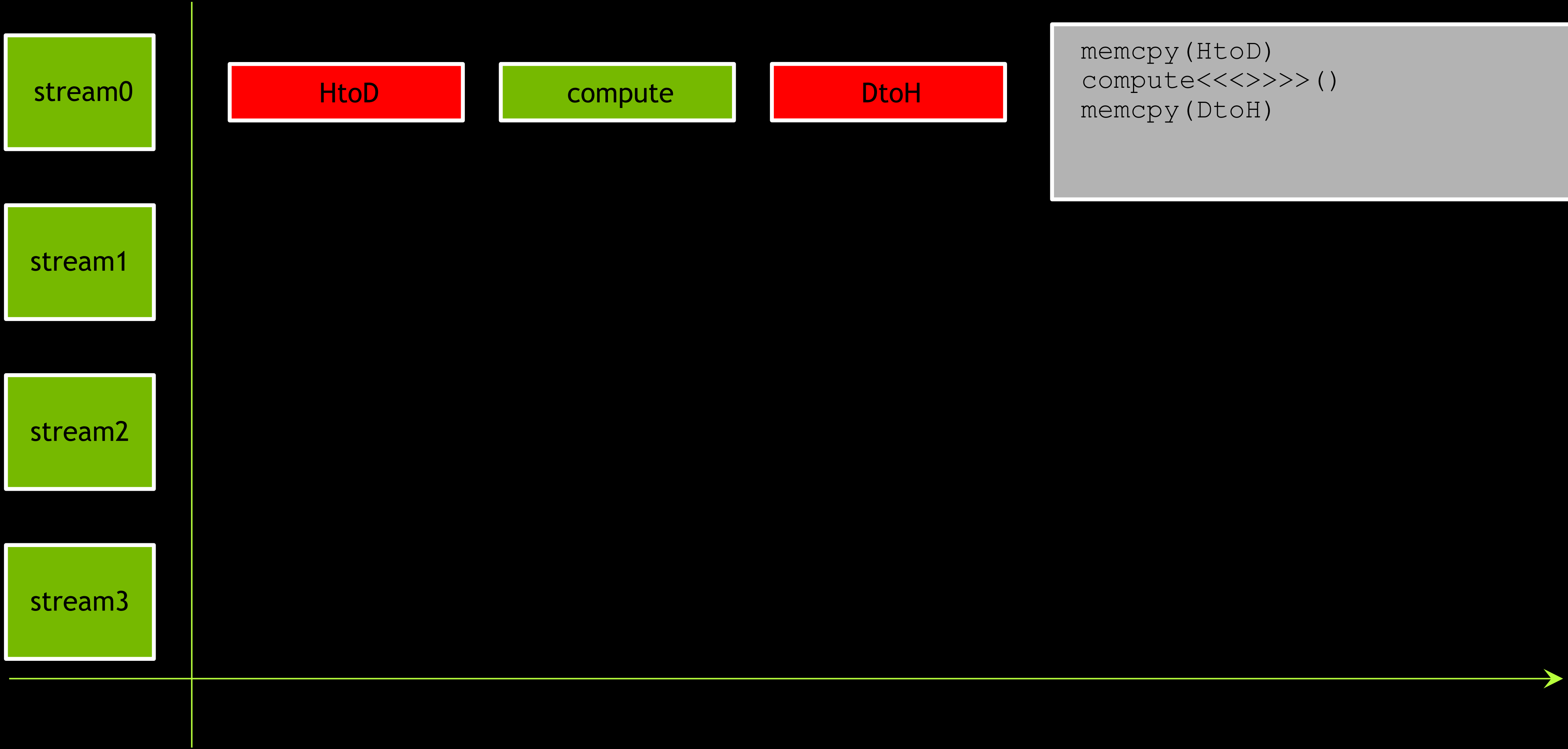


Consider if we were to take the existing program...

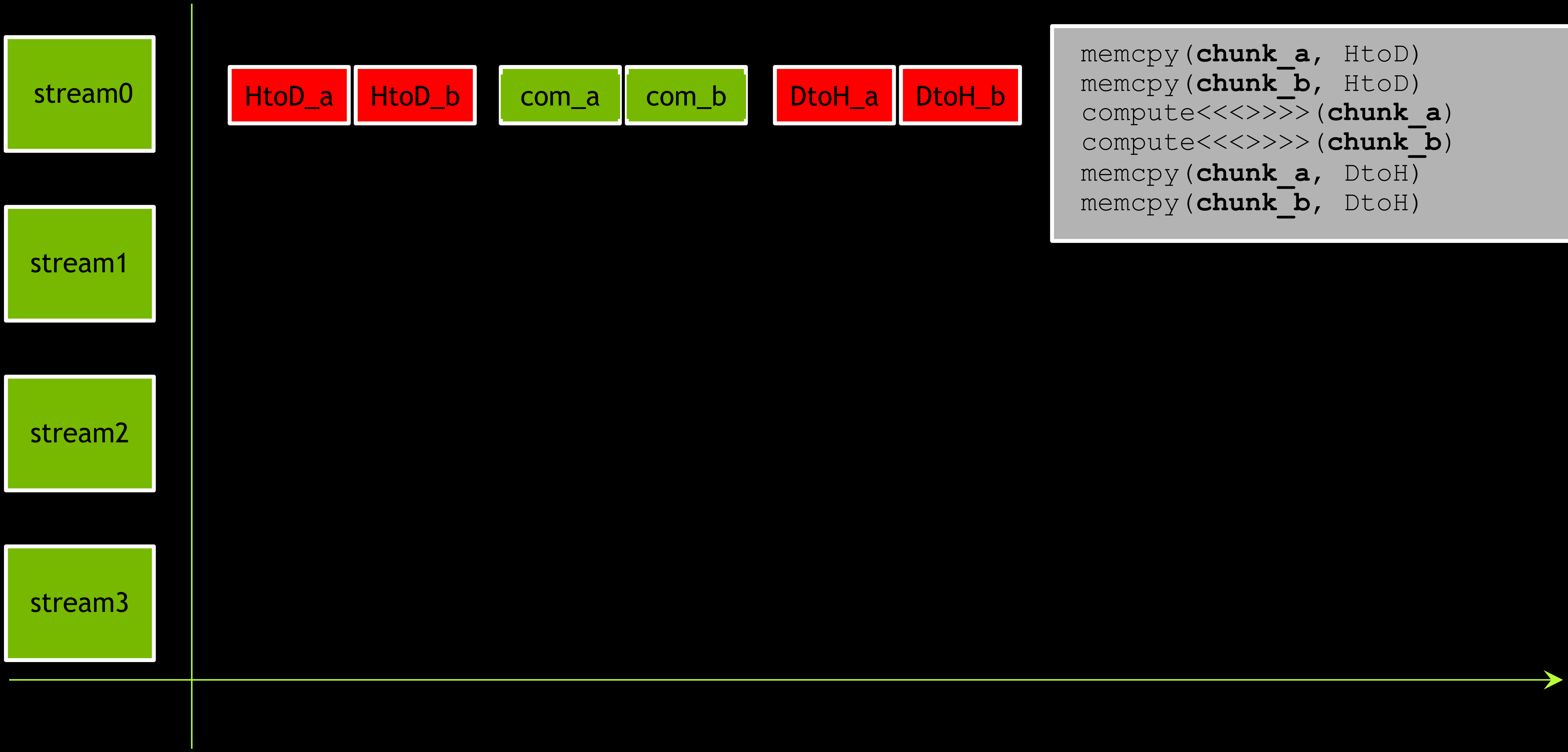


Consider if we were to take the existing program…

```
memcpy(HtoD)
compute<<<>>>()
memcpy(DtoH)
```

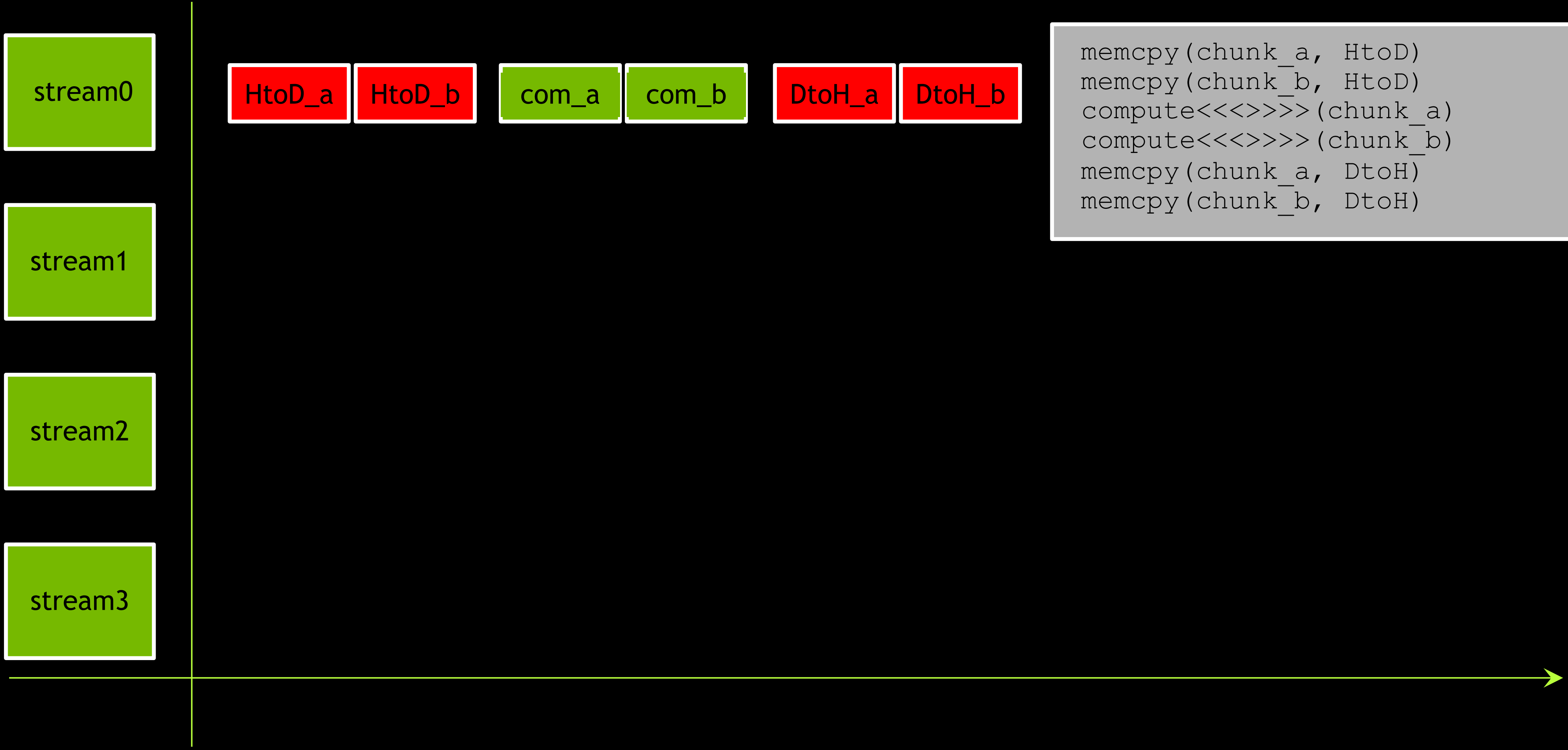


...and split the data into 2 **chunks**



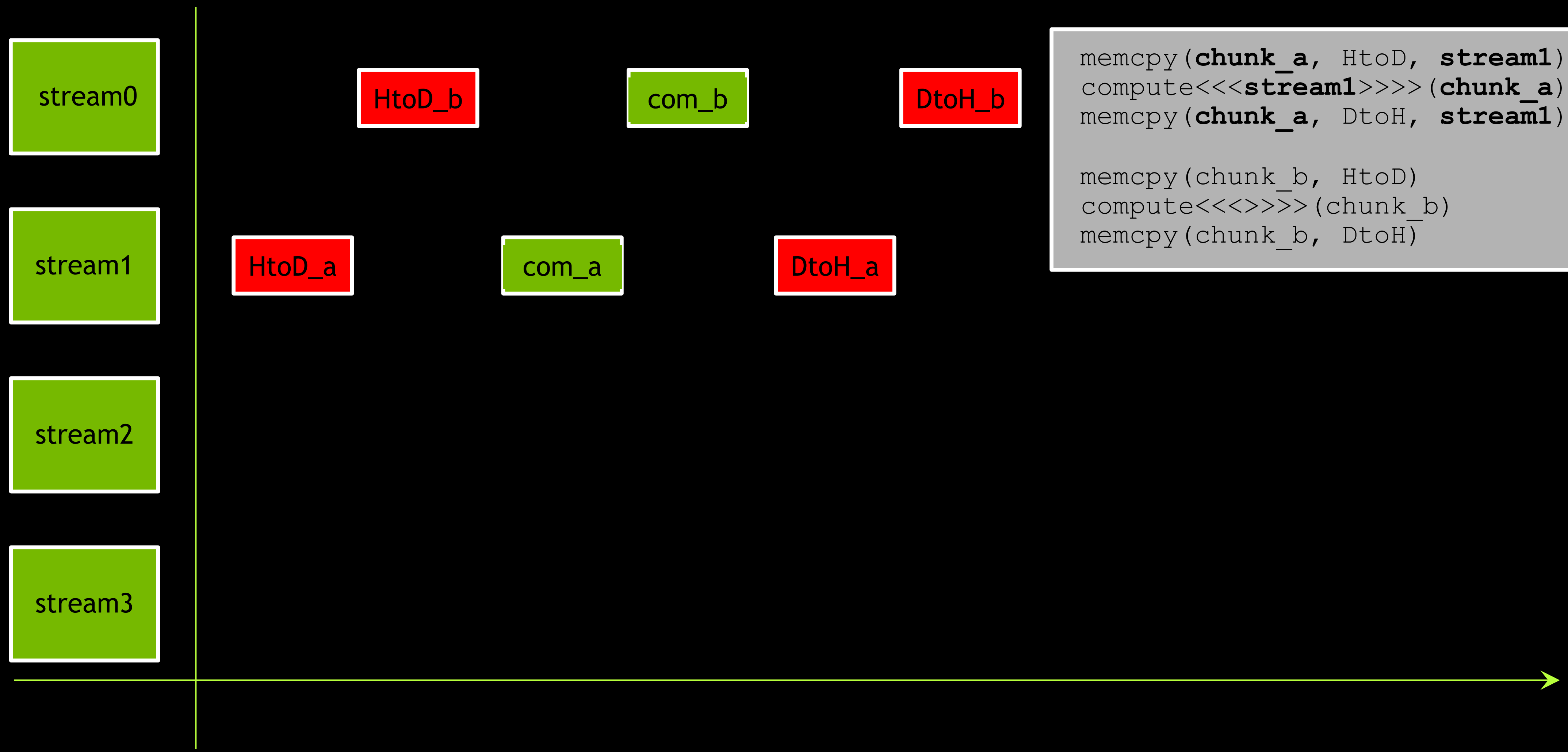

```
memcpy(chunk_a, HtoD)
memcpy(chunk_b, HtoD)
compute<<<>>>(chunk_a)
compute<<<>>>(chunk_b)
memcpy(chunk_a, DtoH)
memcpy(chunk_b, DtoH)
```

If we now move all operations for each chunk into their own separate non-default stream…




```
memcpy(chunk_a, HtoD)
memcpy(chunk_b, HtoD)
compute<<<>>>(chunk_a)
compute<<<>>>(chunk_b)
memcpy(chunk_a, DtoH)
memcpy(chunk_b, DtoH)
```

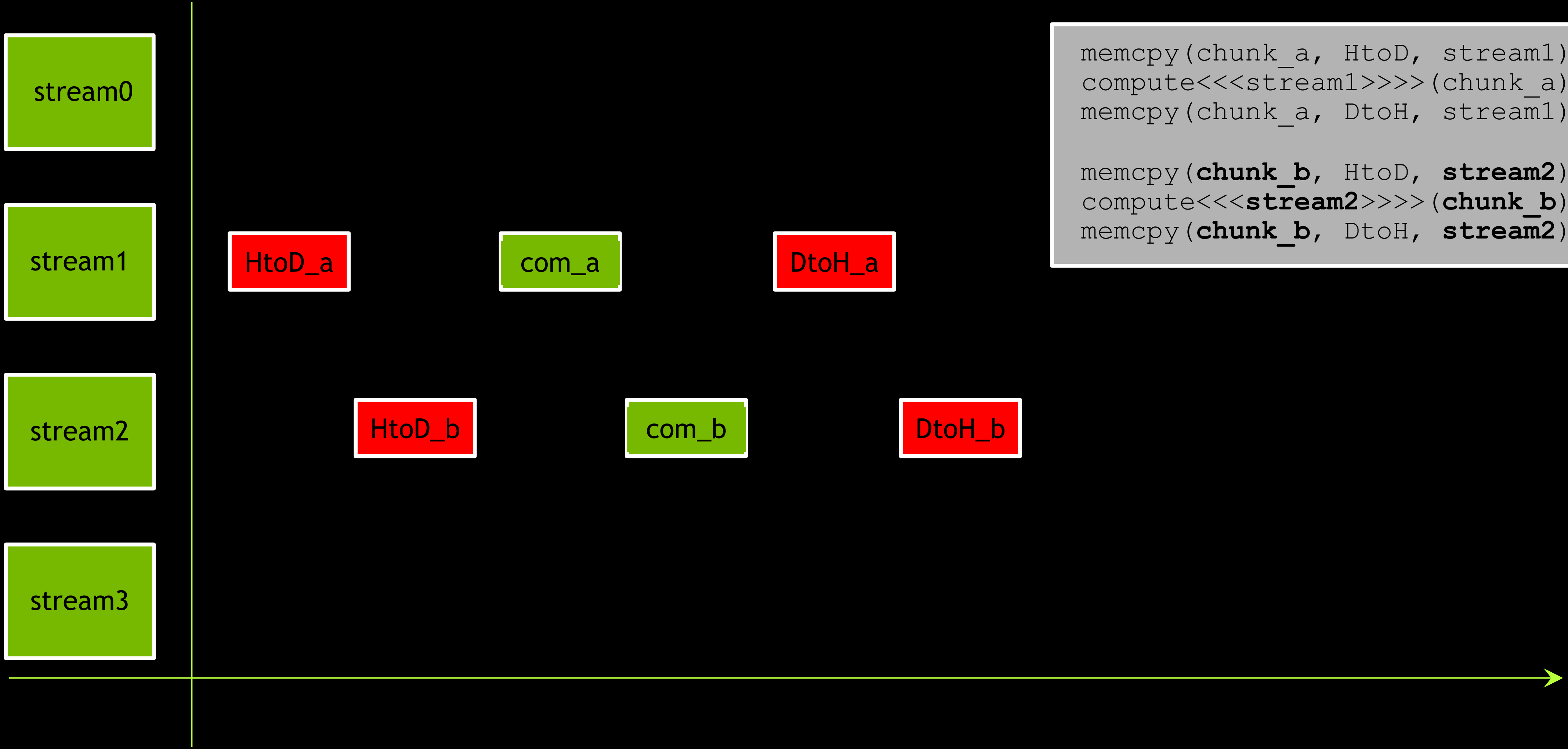
If we now move all operations for each chunk into their own separate non-default stream...




```
memcpy(chunk_a, HtoD, stream1)
compute<<<stream1>>>>(chunk_a)
memcpy(chunk_a, DtoH, stream1)

memcpy(chunk_b, HtoD)
compute<<<>>>>(chunk_b)
memcpy(chunk_b, DtoH)
```

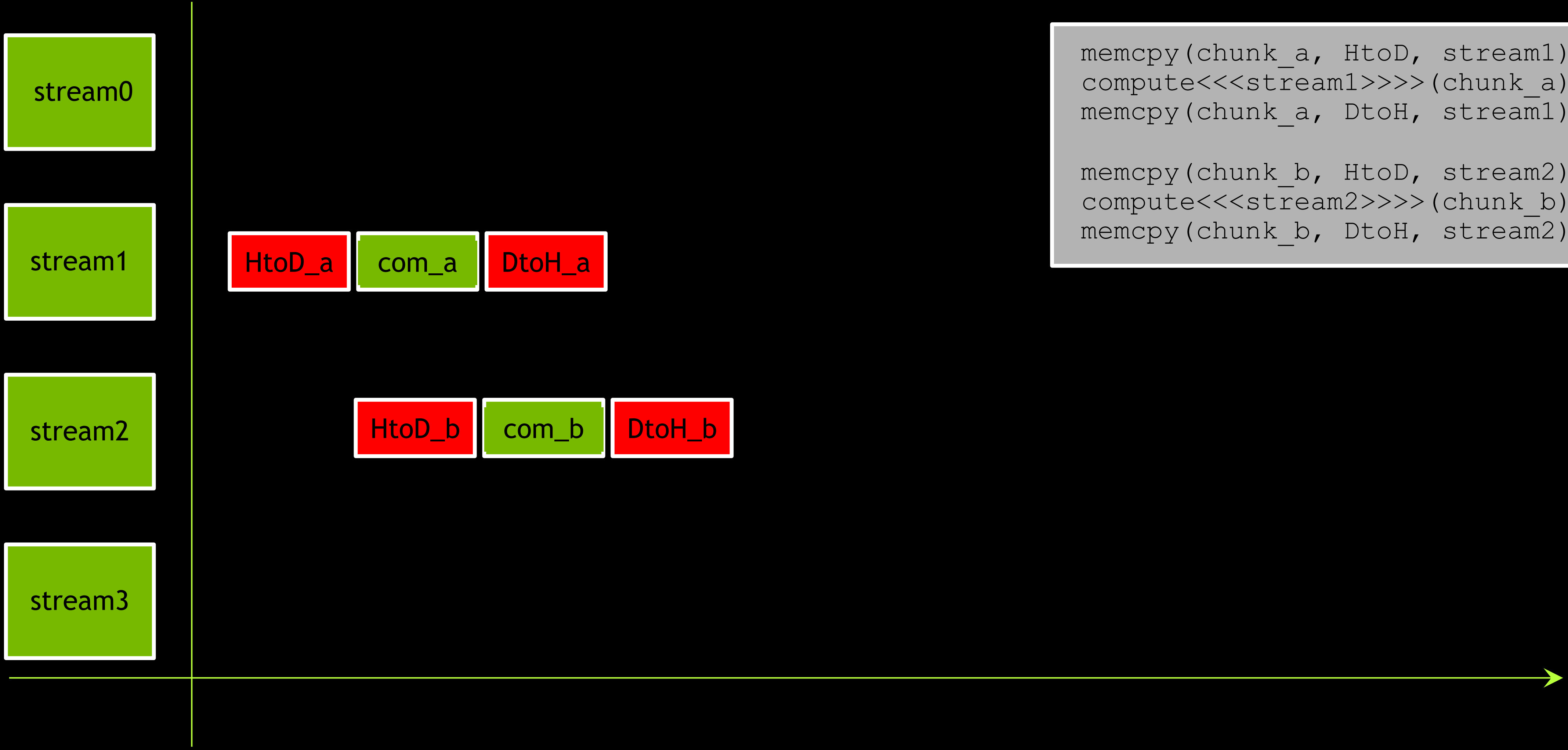
If we now move all operations for each chunk into their own separate non-default stream…

```
memcpy(chunk_a, HtoD, stream1)
compute<<<stream1>>>>(chunk_a)
memcpy(chunk_a, DtoH, stream1)

memcpy(chunk_b, HtoD, stream2)
compute<<<stream2>>>>(chunk_b)
memcpy(chunk_b, DtoH, stream2)
```



…data/compute order is maintained, and,
we can achieve some overlap

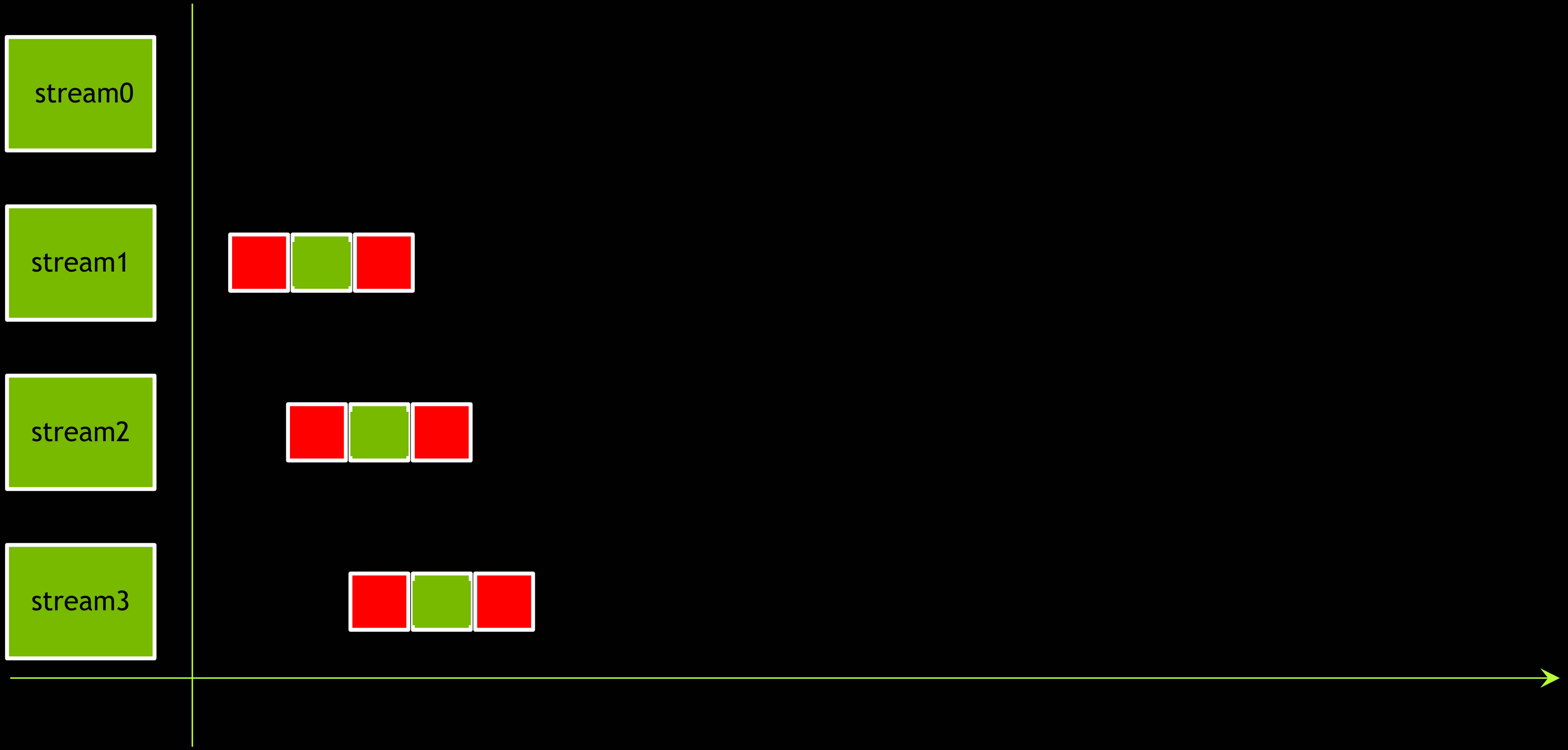
```
memcpy(chunk_a, HtoD, stream1)
compute<<<stream1>>>>(chunk_a)
memcpy(chunk_a, DtoH, stream1)

memcpy(chunk_b, HtoD, stream2)
compute<<<stream2>>>>(chunk_b)
memcpy(chunk_b, DtoH, stream2)
```



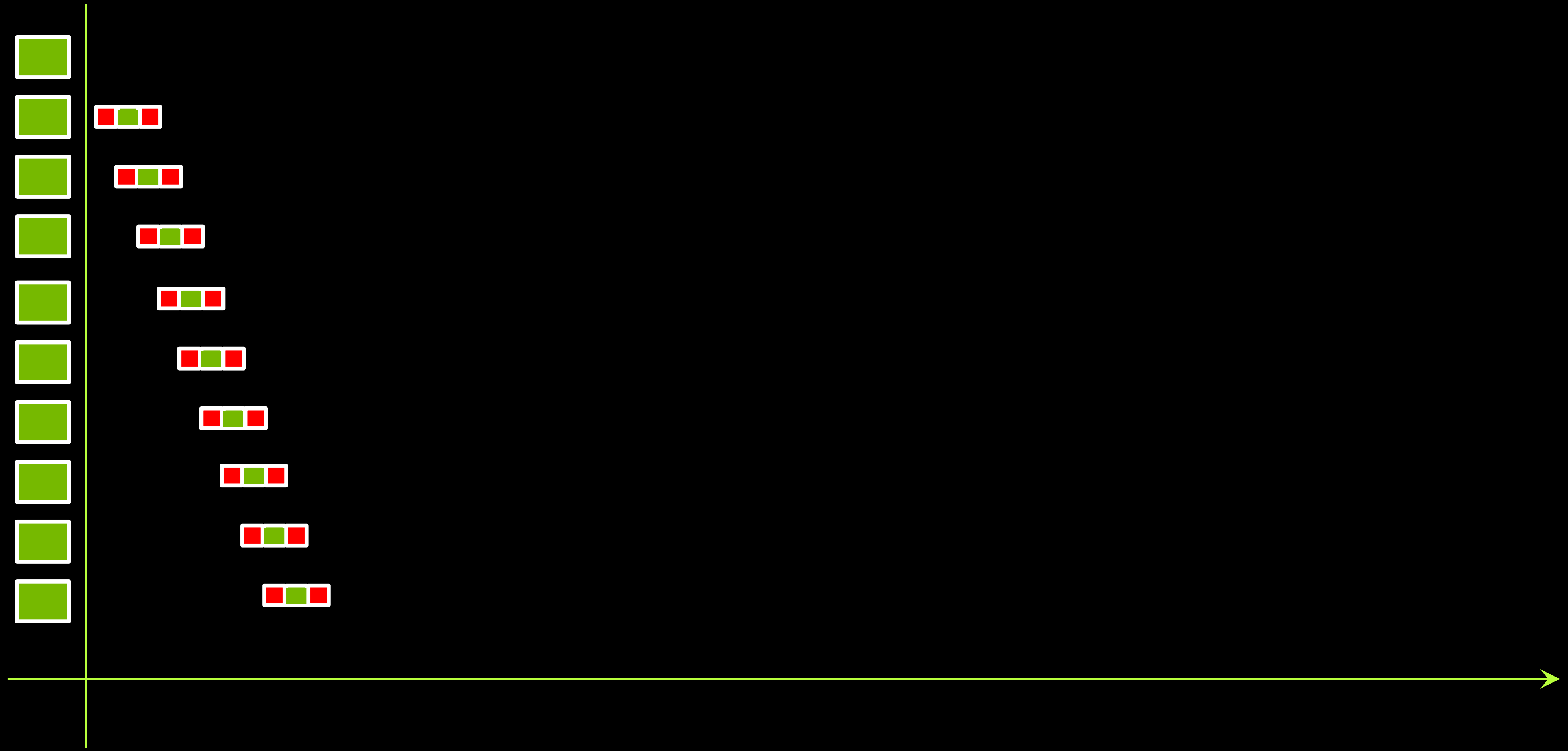
…data/compute order is maintained, and,
we can achieve some overlap

```
memcpy(chunk_a, HtoD, stream1)
compute<<<stream1>>>>(chunk_a)
memcpy(chunk_a, DtoH, stream1)

memcpy(chunk_b, HtoD, stream2)
compute<<<stream2>>>>(chunk_b)
memcpy(chunk_b, DtoH, stream2)
```

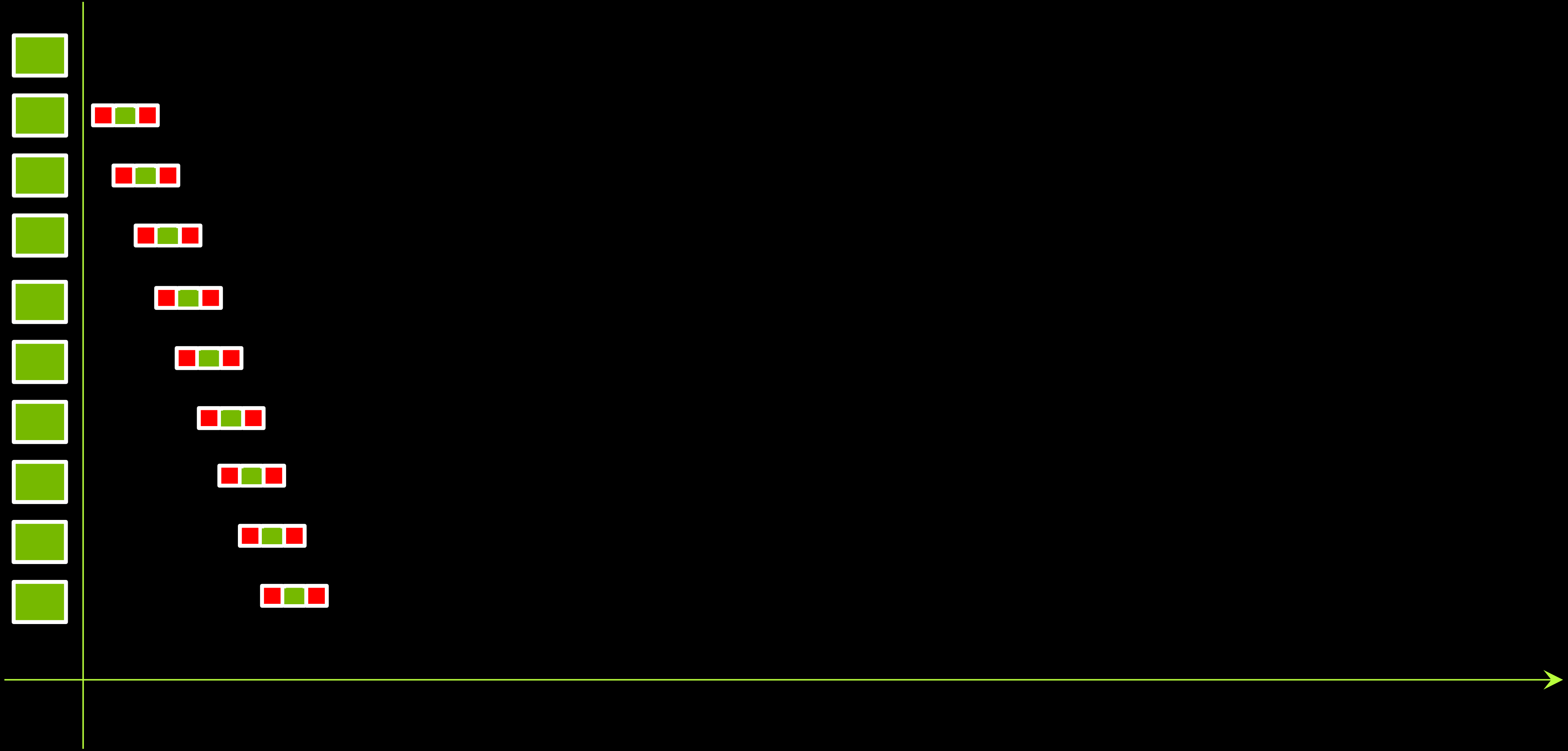
Hypothetically, the number of chunks could be increased for perhaps even better overlap



Hypothetically, the number of chunks could be increased for perhaps even better overlap



The ideal chunking is best learned by observing program performance



COPY/COMPUTE OVERLAP INDEXING

When chunking data to use in multiple streams, indexing can be tricky

Let’s look at a couple examples of how it can be done

We will start by allocating the data needed for all chunks, here a small size to make the example clear

0	1	2	3	4	5	6	7	8	9

We will start by allocating the data needed for all chunks, here a small size to make the example clear

```
cudaMallocHost(&data_cpu, N)
```

N	10	10

0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

Of course we would allocate for the GPU as well, but here we will only present one image of the data

```
cudaMallocHost(&data_cpu, N)
cudaMalloc(&data_gpu, N)
```

N	10	10

0	1	2	3	4	5	6	7	8	9

Of course we would allocate for the GPU as well, but here we will only present one image of the data

```
cudaMallocHost(&data_cpu, N)
```

N	10	10

0	1	2	3	4	5	6	7	8	9

Next, we will define the number of streams, and loop to create and collect them in an array

N	10	10

0	1	2	3	4	5	6	7	8	9

Next, we will define the number of streams, and loop to create and collect them in an array

```
num_streams = 2
// for stream_i in num_streams
     cudaStreamCreate(stream)
     streams[stream_i] = stream
```

N	10	10
num_streams	2	2

0	1	2	3	4	5	6	7	8	9

The size of each chunk of data will depend on the number of data entries and the number of streams

N	10	10
num_streams	2	2

0	1	2	3	4	5	6	7	8	9

The size of each chunk of data will depend on the number of data entries and the number of streams

```
chunk_size = N / num_streams
```

N	10	10
num_streams	2	2
chunk_size	N/num_streams	5

0 1 2 3 4 5 6 7 8 9

Each stream will need to handle one chunk of data. We need to calculate its index into the whole data set

N	10	10
num_streams	2	2
chunk_size	N/num_streams	5

0	1	2	3	4	5	6	7	8	9

To do this we will range over the number of streams, starting at 0...

```
// for stream_i in num_streams
```

N	10	10
num_streams	2	2
chunk_size	N/num_streams	5
stream_i	0	0

0 1 2 3 4 5 6 7 8 9

...and multiply by chunk size

```
// for stream_i in num_streams
     lower = chunk_size*stream_i
```

N	10	10
num_streams	2	2
chunk_size	N/num_streams	5
stream_i	0	0
lower	chunk_size*stream_i	0

0	1	2	3	4	5	6	7	8	9

Starting at the **lower** index and utilizing a chunk size worth of data will give us the stream's data within all data

```
// for stream_i in num_streams
     lower = chunk_size*stream_i
```

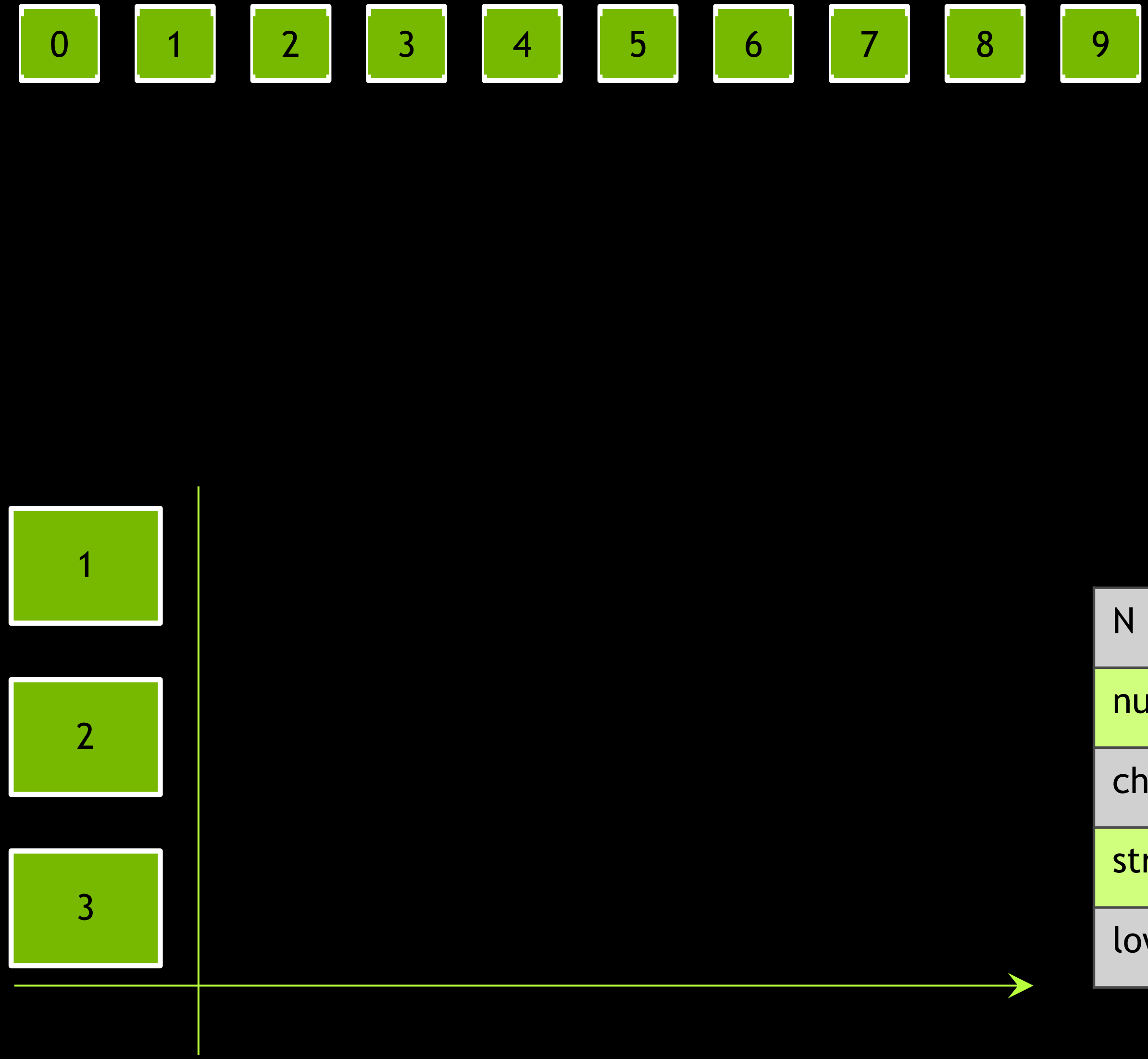
N	10	10
num_streams	2	2
chunk_size	N/num_streams	5
stream_i	0	0
lower	chunk_size*stream_i	0

0	1	2	3	4	5	6	7	8	9

This method will work for each stream_i

```
// for stream_i in num_streams
     lower = chunk_size*stream_i
```

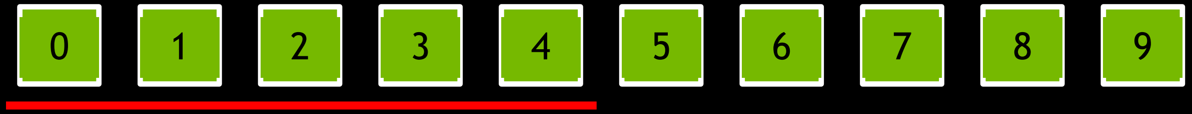
N	10	10
num_streams	2	2
chunk_size	N/num_streams	5
stream_i	1	1
lower	chunk_size*stream_i	5



Having calculated these values, we can now perform non-default stream HtoD memory copies...

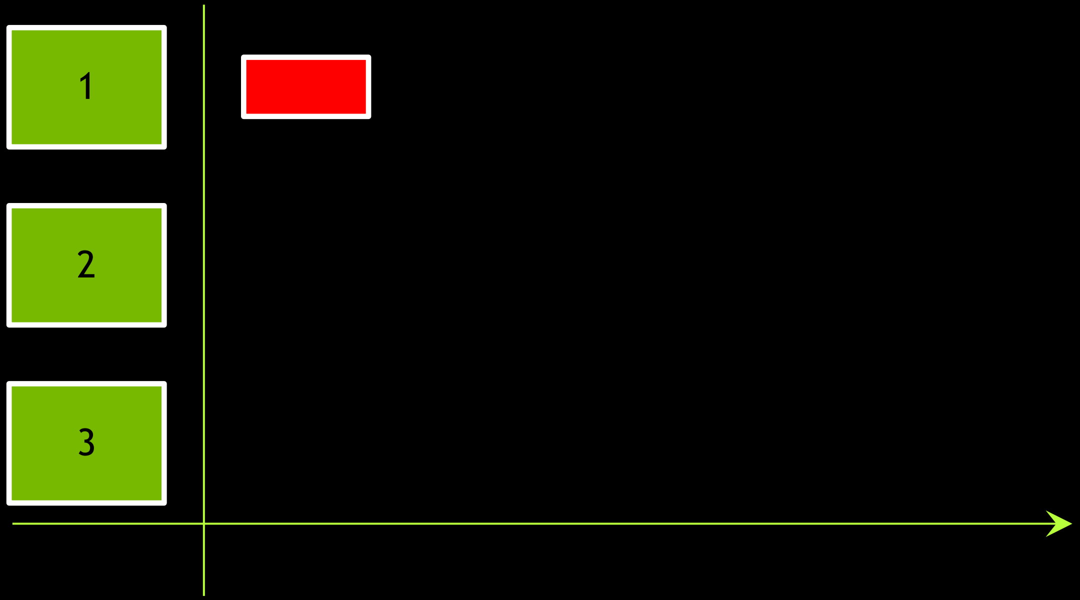
```
cudaMemcpyAsync(
    data_cpu+lower,
    data_gpu+lower,
    sizeof(uint64_t)*chunk_size,
    cudaMemcpyHostToDevice,
    streams[stream_i]
)
```

N	10	10
num_streams	2	2
chunk_size	N/num_streams	5
stream_i	0	0
lower	chunk_size*stream_i	0

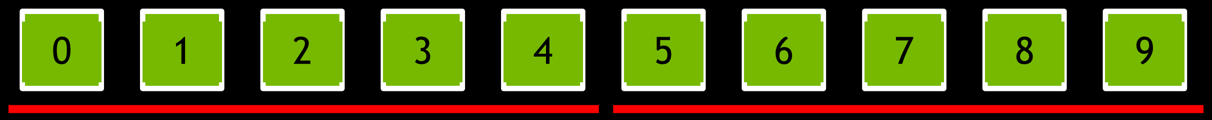


Having calculated these values, we can now perform non-default stream HtoD memory copies...

```
cudaMemcpyAsync(
    data_cpu+lower,
    data_gpu+lower,
    sizeof(uint64_t)*chunk_size,
    cudaMemcpyHostToDevice,
    streams[stream_i]
)
```

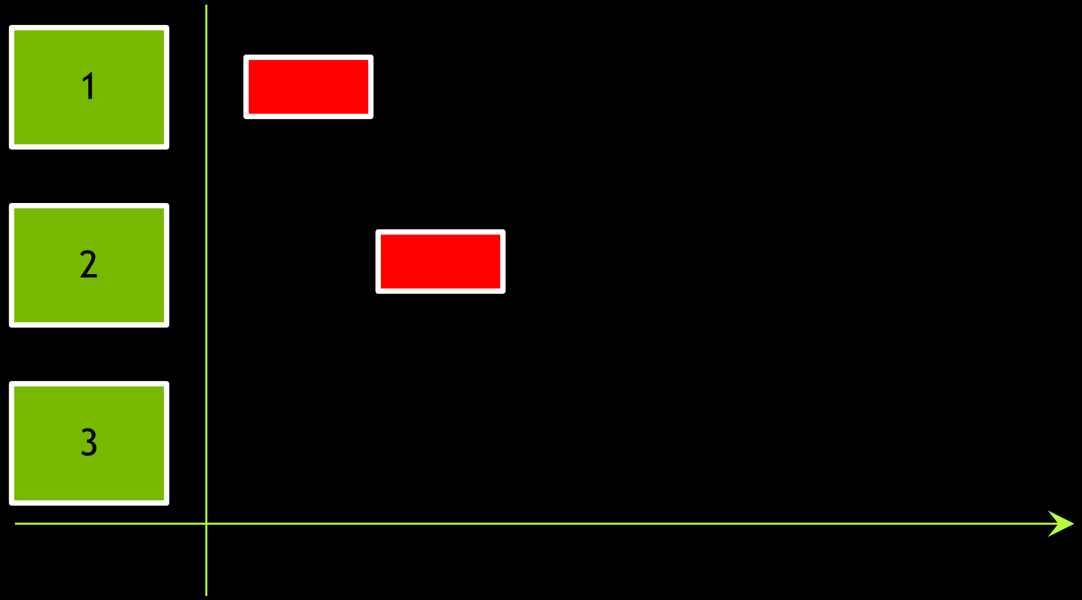


N	10	10
num_streams	2	2
chunk_size	N/num_streams	5
stream_i	0	0
lower	chunk_size*stream_i	0

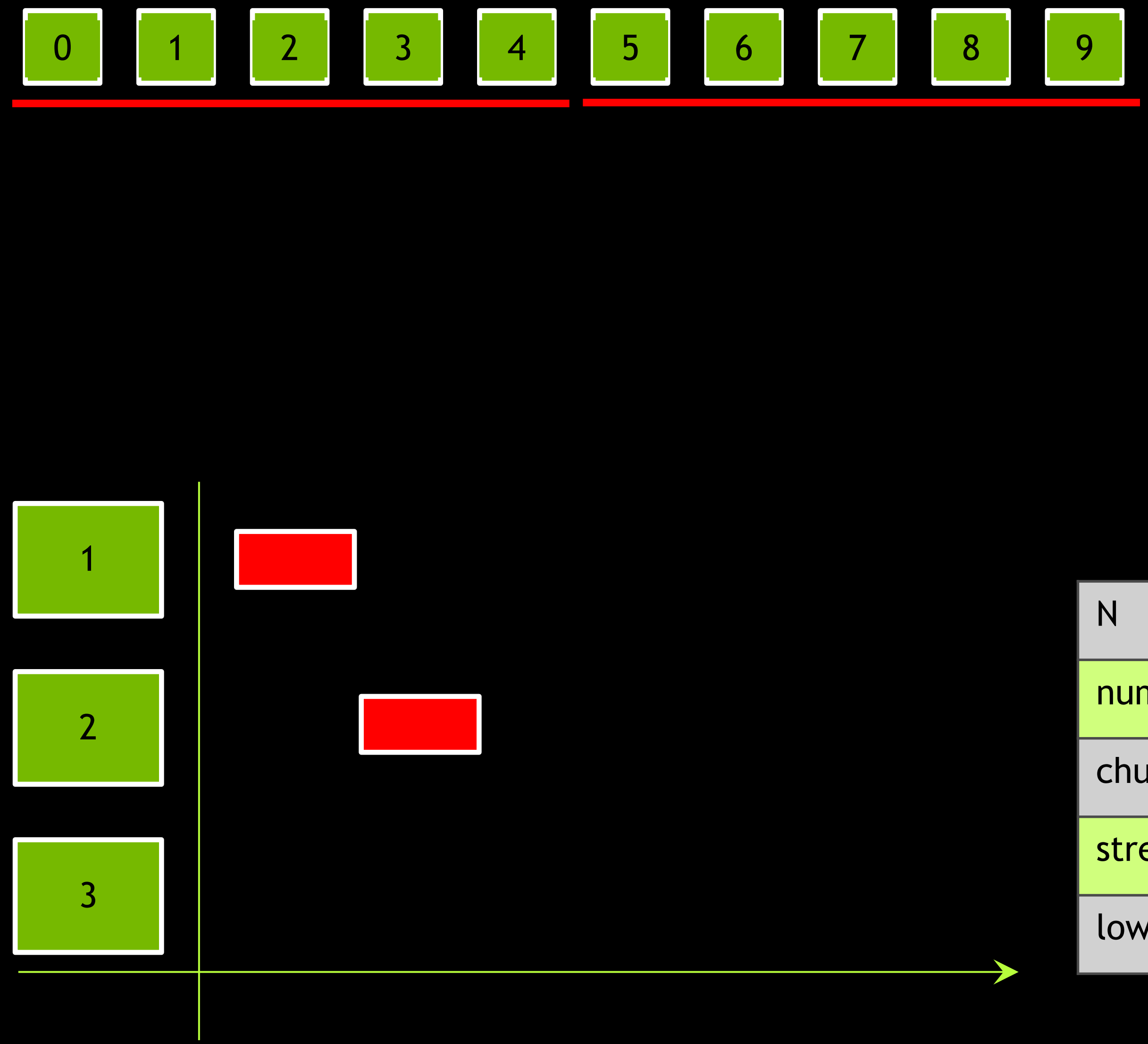


Having calculated these values, we can now perform non-default stream HtoD memory copies...

```
cudaMemcpyAsync(
    data_cpu+lower,
    data_gpu+lower,
    sizeof(uint64_t)*chunk_size,
    cudaMemcpyHostToDevice,
    streams[stream_i]
)
```



N	10	10
num_streams	2	2
chunk_size	N/num_streams	5
stream_i	1	1
lower	chunk_size*stream_i	5



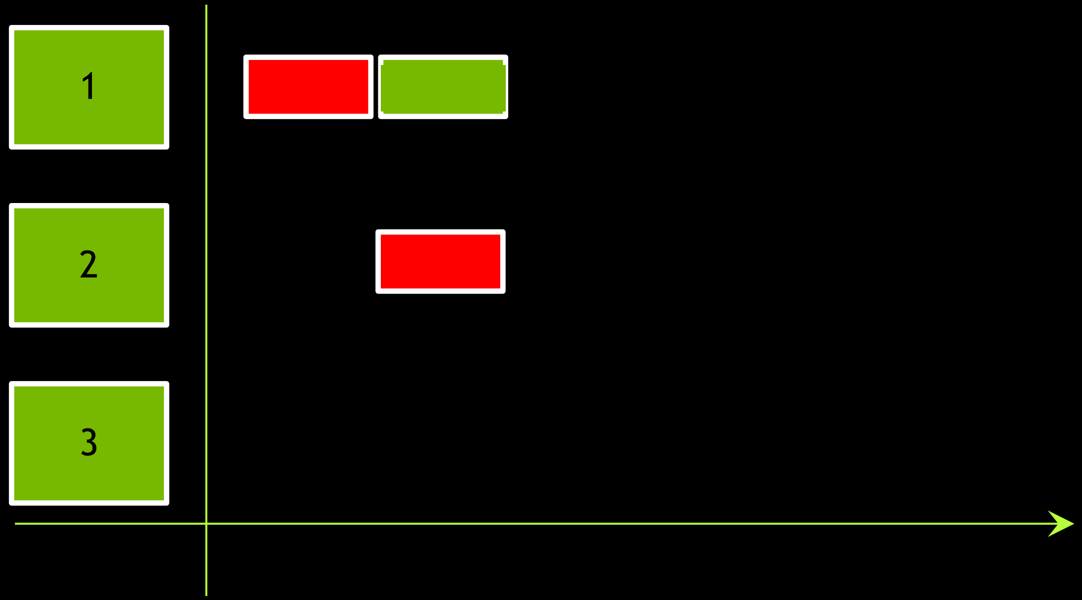
…non-default stream kernel launches…

```
kernel
<<<G, B, 0,streams[stream_i]>>>(
     data_gpu + lower,
     chunk_size
)
```

N	10	10
num_streams	2	2
chunk_size	N/num_streams	5
stream_i	0	0
lower	chunk_size*stream_i	0

...non-default stream kernel launches...

```
kernel
<<<G, B, 0,streams[stream_i]>>>(
     data_gpu + lower,
     chunk_size
)
```



N	10	10
num_streams	2	2
chunk_size	N/num_streams	5
stream_i	0	0
lower	chunk_size*stream_i	0

…non-default stream kernel launches…

```
kernel
<<<G, B, 0,streams[stream_i]>>>(
     data_gpu + lower,
     chunk_size
)
```

N	10	10
num_streams	2	2
chunk_size	N/num_streams	5
stream_i	1	1
lower	chunk_size*stream_i	5

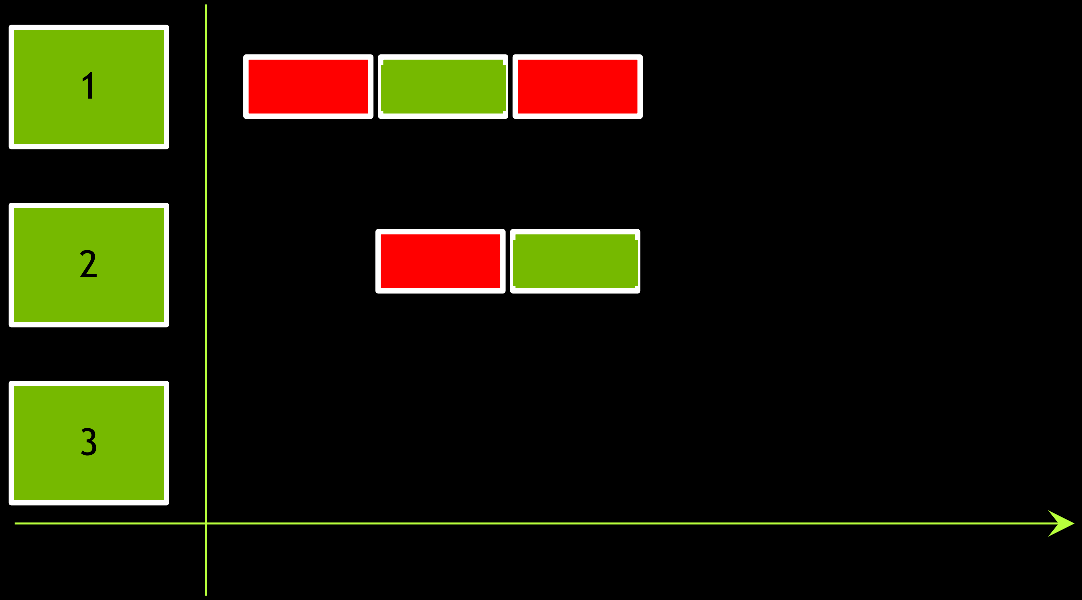
...and non-default stream DtoH memory copies

```
cudaMemcpyAsync(
    data_cpu + lower,
    data_gpu + lower,
    sizeof(uint64_t)*chunk_size,
    cudaMemcpyHostToDevice,
    streams[stream_i]
)
```

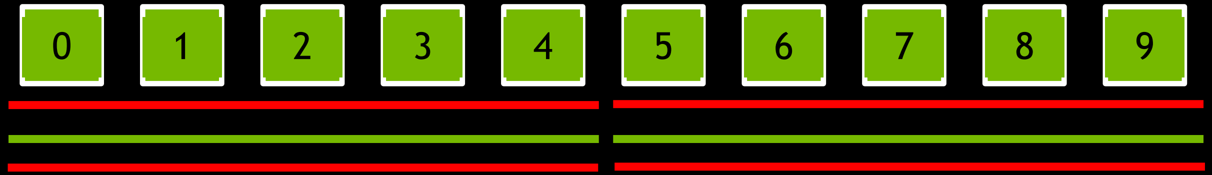
N	10	10
num_streams	2	2
chunk_size	N/num_streams	5
stream_i	0	0
lower	chunk_size*stream_i	0

...and non-default stream DtoH memory copies

```
cudaMemcpyAsync(
    data_cpu + lower,
    data_gpu + lower,
    sizeof(uint64_t)*chunk_size,
    cudaMemcpyHostToDevice,
    streams[stream_i]
)
```

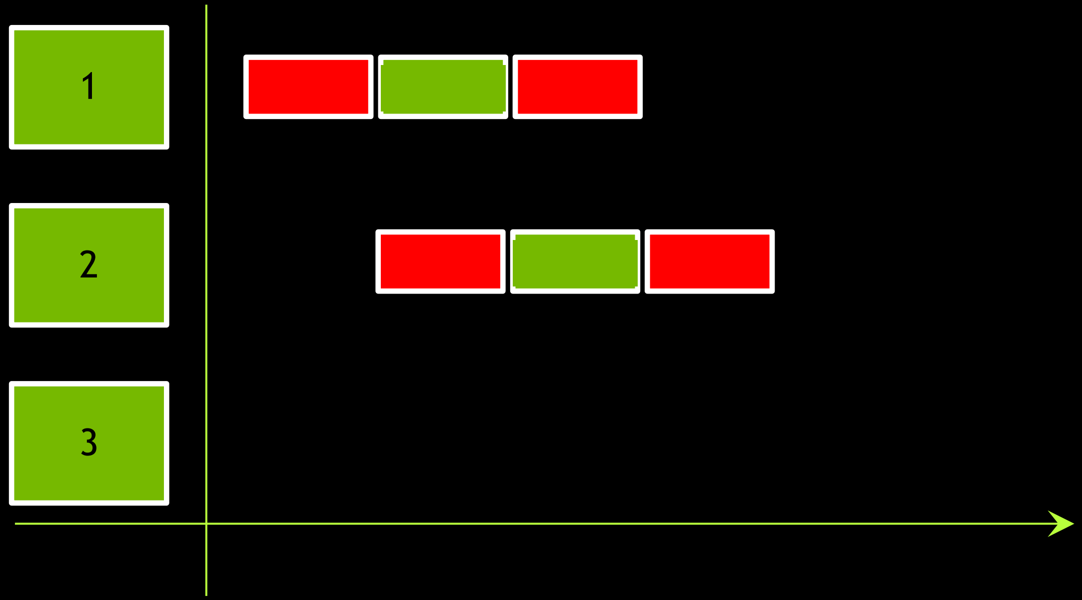


N	10	10
num_streams	2	2
chunk_size	N/num_streams	5
stream_i	0	0
lower	chunk_size*stream_i	0



...and non-default stream DtoH memory copies

```
cudaMemcpyAsync(
    data_cpu + lower,
    data_gpu + lower,
    sizeof(uint64_t)*chunk_size,
    cudaMemcpyHostToDevice,
    streams[stream_i]
)
```



N	10	10
num_streams	2	2
chunk_size	N/num_streams	5
stream_i	1	1
lower	chunk_size*stream_i	5

0 1 2 3 4 5 6 7 8 9

For this example, N was evenly divided by number of streams

N	10	10
num_streams	2	2
chunk_size	N/num_streams	5
stream_i	0	0
lower	chunk_size*stream_i	0

0 1 2 3 4 5 6 7 8 9

But what if this is not the case?

N	10	10
num_streams	3	3
chunk_size	N/num_streams	?
stream_i	0	0
lower	chunk_size*stream_i	?

0	1	2	3	4	5	6	7	8	9

Dividing two ints will result in an int, rounded down if necessary, as in this case

N	10	10
num_streams	3	3
chunk_size	N/num_streams	3
stream_i	0	0
lower	chunk_size*stream_i	0

0	1	2	3	4	5	6	7	8	9

Now as we iterate through the streams to get lower, and then apply chunk size…

N	10	10
num_streams	3	3
chunk_size	N/num_streams	3
stream_i	0	0
lower	chunk_size*stream_i	0

0 1 2 3 4 5 6 7 8 9

Now as we iterate through the streams to get lower, and then apply chunk size...

N	10	10
num_streams	3	3
chunk_size	N/num_streams	3
stream_i	0	0
lower	chunk_size*stream_i	0

0 1 2 3 4 5 6 7 8 9

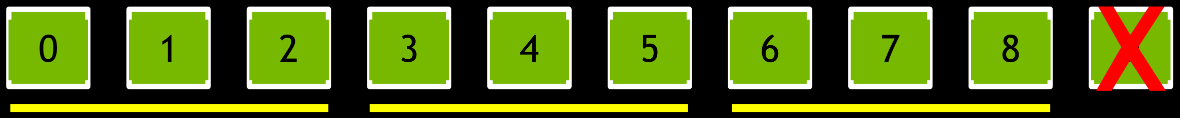
Now as we iterate through the streams to get lower, and then apply chunk size...

N	10	10
num_streams	3	3
chunk_size	N/num_streams	3
stream_i	1	1
lower	chunk_size*stream_i	3

0	1	2	3	4	5	6	7	8	9

Now as we iterate through the streams to get lower, and then apply chunk size...

N	10	10
num_streams	3	3
chunk_size	N/num_streams	3
stream_i	2	2
lower	chunk_size*stream_i	6



...we fail to access all values in the data

N	10	10
num_streams	3	3
chunk_size	N/num_streams	3
stream_i	-	-
lower	chunk_size*stream_i	-

0	1	2	3	4	5	6	7	8	9

To fix this we use round-up division to calculate chunk size

N	10	10
num_streams	3	3
chunk_size	ceil_div(N/num_streams)	4
stream_i	0	0
lower	chunk_size*stream_i	0

0	1	2	3	4	5	6	7	8	9

Now as we iterate...

N	10	10
num_streams	3	3
chunk_size	ceil_div(N/num_streams)	4
stream_i	0	0
lower	chunk_size*stream_i	0

0 1 2 3 4 5 6 7 8 9

Now as we iterate...

N	10	10
num_streams	3	3
chunk_size	ceil_div(N/num_streams)	4
stream_i	0	0
lower	chunk_size*stream_i	0

Now as we iterate…

N	10	10
num_streams	3	3
chunk_size	ceil_div(N/num_streams)	4
stream_i	1	1
lower	chunk_size*stream_i	4

We actually access all data, but we have a new problem: chunk size is too large for our last chunk of data

N	10	10
num_streams	3	3
chunk_size	ceil_div(N/num_streams)	4
stream_i	2	2
lower	chunk_size*stream_i	8

0	1	2	3	4	5	6	7	8	9

If, however, we calculate an **upper** index for each chunk that is bound by N...

```
upper = min(lower+chunk_size, N)
```

N	10	10
num_streams	3	3
chunk_size	ceil_div(N/num_streams)	4
stream_i	0	0
lower	chunk_size*stream_i	0
upper	min(lower + chunk_size, N)	4

0	1	2	3	4	5	6	7	8	9

And then calculate a chunk **width** using **upper** and **lower**...

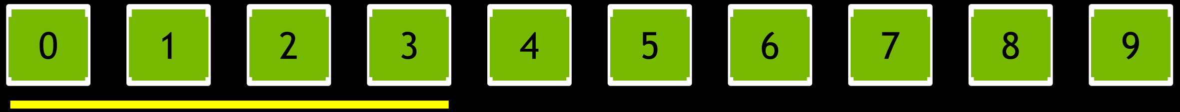
```
width = upper - lower
```

N	10	10
num_streams	3	3
chunk_size	ceil_div(N/num_streams)	4
stream_i	0	0
lower	chunk_size*stream_i	0
upper	min(lower + chunk_size, N)	4
width	upper - lower	4

0 1 2 3 4 5 6 7 8 9

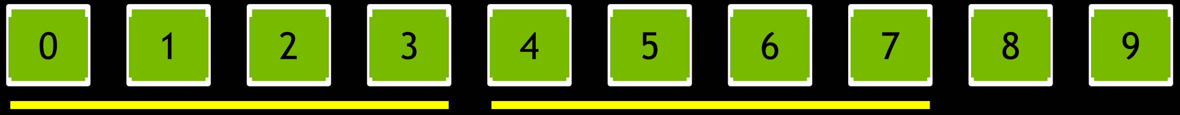
…now when we iterate using width instead of chunk size…

N	10	10
num_streams	3	3
chunk_size	ceil_div(N/num_streams)	4
stream_i	0	0
lower	chunk_size*stream_i	0
upper	min(lower + chunk_size, N)	4
width	upper - lower	4



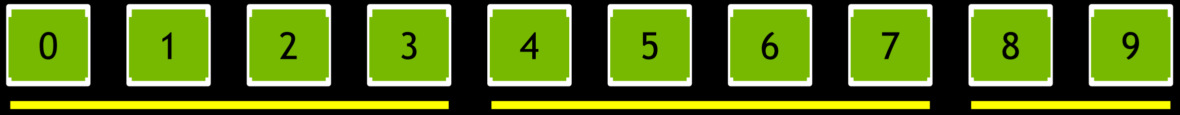
...now when we iterate using width instead of chunk size...

N	10	10
num_streams	3	3
chunk_size	ceil_div(N/num_streams)	4
stream_i	0	0
lower	chunk_size*stream_i	0
upper	min(lower + chunk_size, N)	4
width	upper - lower	4



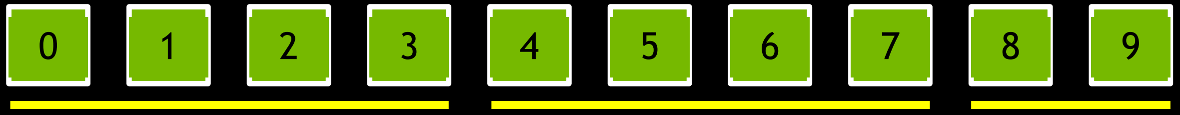
...now when we iterate using width instead of chunk size...

N	10	10
num_streams	3	3
chunk_size	ceil_div(N/num_streams)	4
stream_i	0	1
lower	chunk_size*stream_i	4
upper	min(lower + chunk_size, N)	8
width	upper - lower	4



...now when we iterate using width instead of chunk size...

N	10	10
num_streams	3	3
chunk_size	ceil_div(N/num_streams)	4
stream_i	0	2
lower	chunk_size*stream_i	8
upper	min(lower + chunk_size, N)	10
width	upper - lower	2



...we fit the data perfectly, no matter its size or the number of streams

N	10	10
num_streams	3	3
chunk_size	ceil_div(N/num_streams)	4
stream_i	0	2
lower	chunk_size*stream_i	8
upper	min(lower + chunk_size, N)	10
width	upper - lower	2

NVIDIA

DEEP LEARNING INSTITUTE

www.nvidia.com/dli