Optimizing Host-Device Data Communication III - *Code Examples*

Stefano Markidis

One Key-Point

1. We use CUDA streams to overlap communication and computation on GPU

Problem

We want to overlap communication and computation in the sequential code

```
...
cudaMemcpy(d_a, a, bytes, cudaMemcpyHostToDevice);
kernel<<<n/blockSize, blockSize>>>(d_a, 0);
cudaMemcpy(a, d_a, bytes, cudaMemcpyDeviceToHost);
...
```

Solution – CUDA Streams

```
...
const int streamSize = n / nStreams;
const int streamBytes = streamSize * sizeof(float);
...
cudaStream_t stream[nStreams];
for (int i = 0; i < nStreams; ++i)
      cudaStreamCreate(&stream[i]);
...
for (int i = 0; i < nStreams; ++i)
      cudaStreamDestroy(stream[i]);
```

- We break up the array of size `N` into chunks of `streamSize` elements.
- The number of non-default streams used is `nStreams=N/streamSize`
- There are two main approaches to domain decomposition and processing

1st Approach

```
...
for (int i = 0; i < nStreams; ++i) {

    int offset = i * streamSize;
    cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes,
                    cudaMemcpyHostToDevice, stream[i]);
    kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);
    cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes,
                    cudaMemcpyDeviceToHost, stream[i]);

}
...
```

We loop over all the operations for each chunk of the array as in this example code.

2nd Approach

```
...
for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes, cudaMemcpyHostToDevice,
                  cudaMemcpyHostToDevice, stream[i]);
}

for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);
}

for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes,
                  cudaMemcpyDeviceToHost, stream[i]);
}
...
```

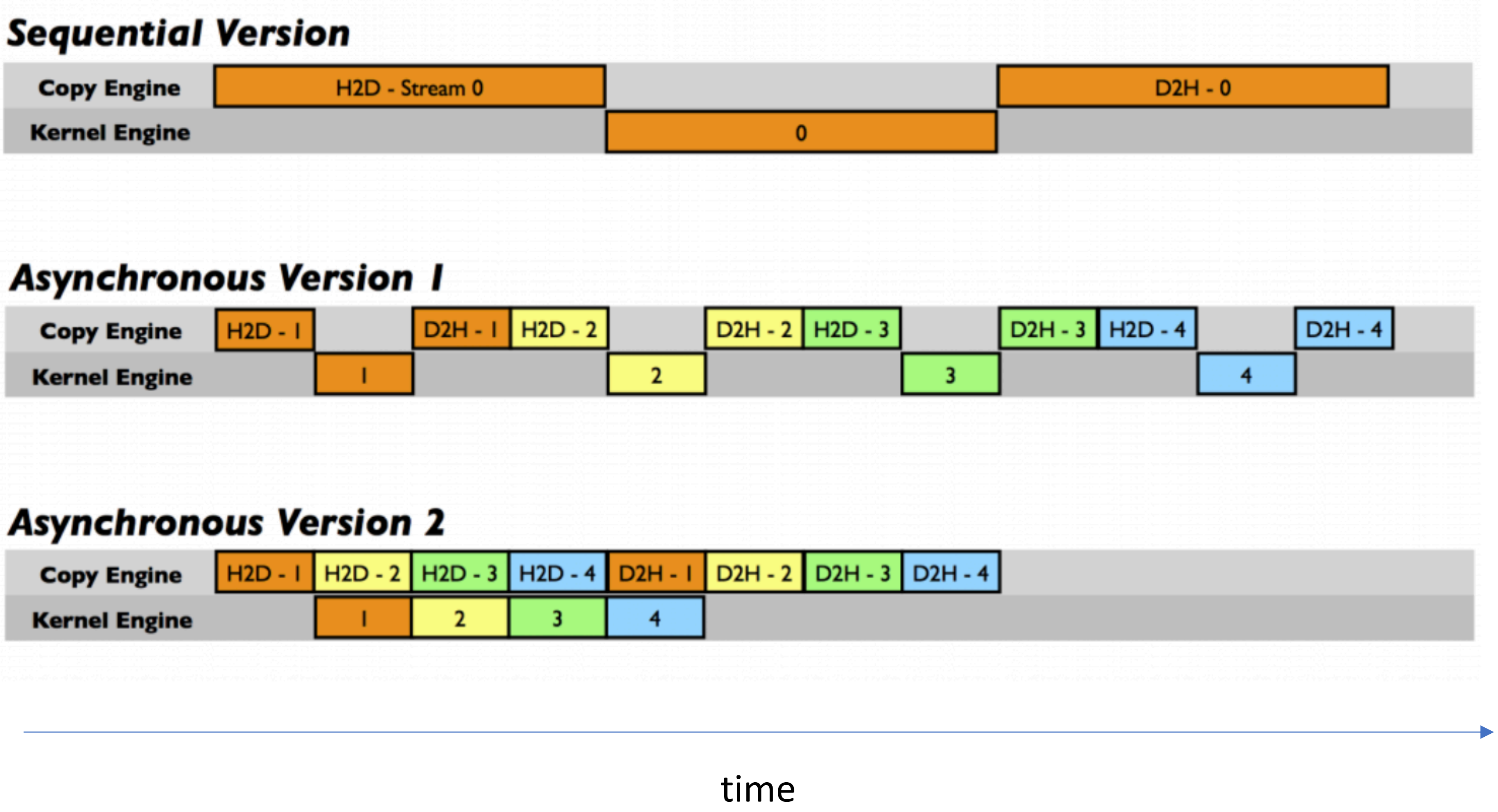
We batch similar operations together, issuing all the host-to-device transfers first, followed by all kernel launches, and then all device-to-host transfers.

Which version performs better?

- The two approaches might perform differently depending on the specific generation of GPU used.
  - CUDA devices **contain engines for various tasks**
    - Dependencies between tasks in different engines are maintained
    - Within any engine all external dependencies are lost
    - Tasks in each engine’s queue are executed in the order they are issued.

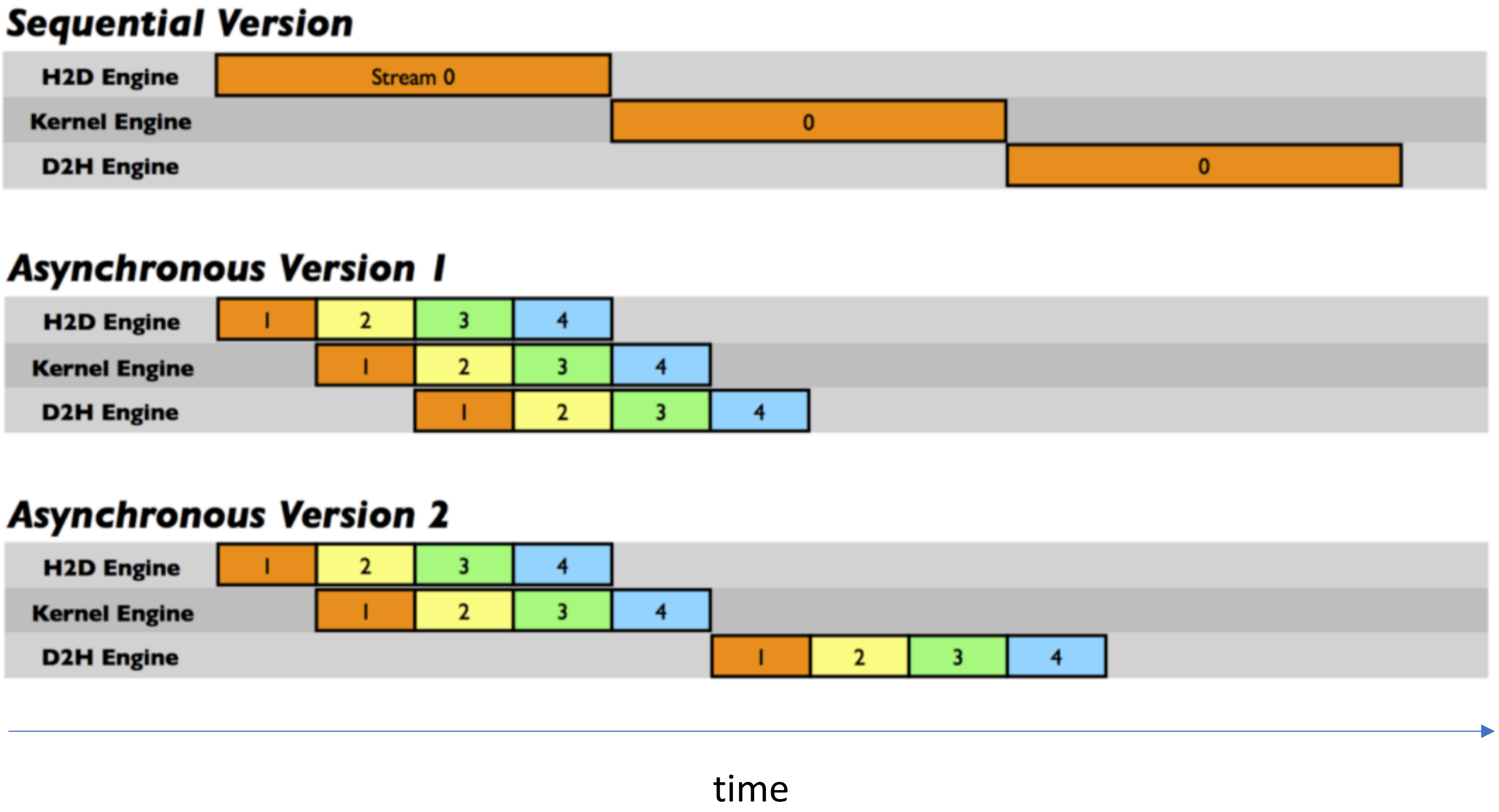
Understanding Difference Performance
1 Copy Engine and 1 Kernel Engine, e.g. Tesla C1060

2nd Approach
Faster!



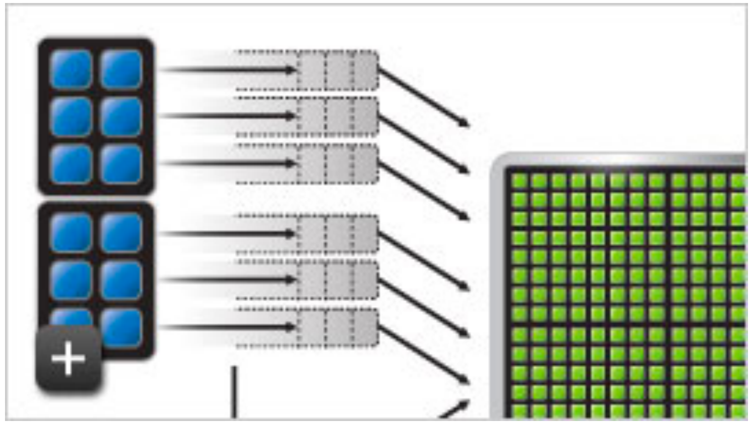
Understanding Difference Performance
2 Copy Engines and 1 Kernel Engine,e.g. Tesla C2050

1st Approach
Faster!



Hyper-Q

- The new **Hyper-Q** GPU feature eliminates the need to tailor the launch order,
- Either approach will perform for devices with **compute capability > 3.5** (the K20 series)



To Summarize

- We showed that CUDA streams can be used to overlap communication and computation in a simple example