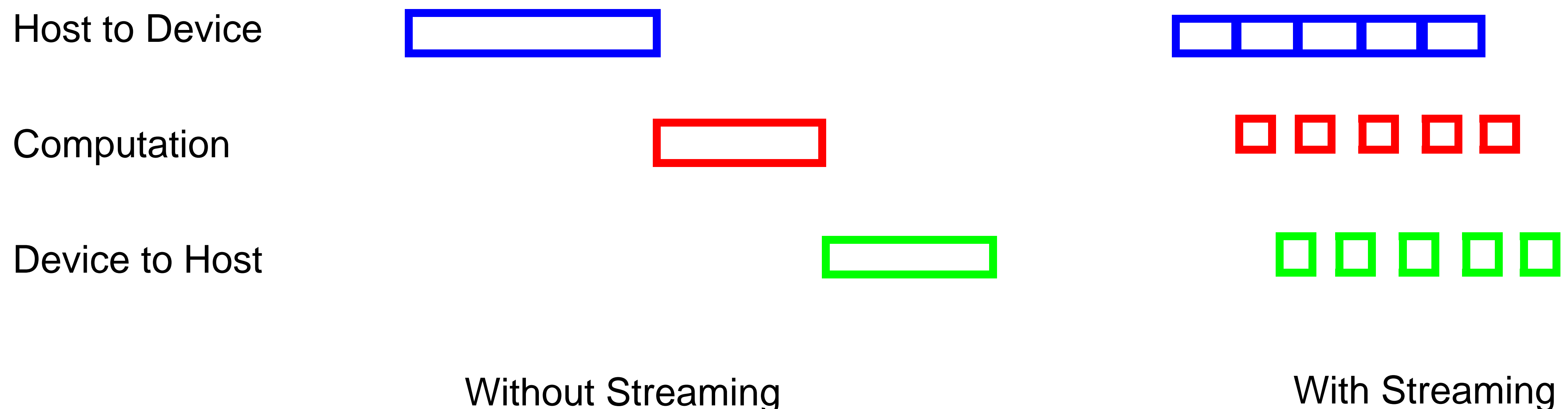
Streaming

Dr Eric McCreath
Research School of Computer Science
The Australian National University

Introduction

The GPU is often described as a "streaming" processor. As it is designed to stream a large amount of data, particularly graphics data. Key characteristics of streaming is overlapping transfers with computation and also reusing buffers such that the total amount of data processed can be considerably greater than the amount that can be stored. If our problem is data parallel we can break up the computation overlap computation and transfers and improve overall performance.



Streams

Cuda provides streams which enables us to asynchronously launch kernels and asynchronously start host/device transfers. If a stream is not specified the executions get added to the default stream and thus serializing all this activity. To create a stream you:

```
cudaStream_t stream;
cudaStreamCreate(&stream);
```

Once this is done you can start a transfer on this stream you use:

```
cudaMemcpyAsync(data_d, data_h, size,  cudaMemcpyHostToDevice, stream);
```

Then to launch a kernel on this stream you:

```
kernel<<<grid,blocks,0,stream>>>();
```

The stream acts like a queue so as items are added they are done in that order, however, computation/transfers can overlap with items on different streams.

Streams

Note also you have queues for:

- host to device transfers
- a compute queue, and
- device to host queue.

So the order in which you make the kernel launch and async transfers will also be maintained.

Breaking up into streams

A simple approach for breaking up your code into streams is to have a different stream for every section. And then use that stream for the: transfer to the host, kernel launch, and the transfer back.

```
cudaStream_t streams[PARTS];

for (int i = 0; i<PARTS; i++) cudaStreamCreate(&streams[i]);

for (int i = 0; i<PARTS; i++) {
    int off = i * sectionSize;
    cudaMemcpyAsync( &data_d[off], &data_h[off], datasize,
                     cudaMemcpyHostToDevice, streams[i]);
    kernel<<<grid,block,0,stream[i]>>>(data_d, off, sectionSize);
    cudaMemcpyAsync( &data_h[off], &data_d[off], datasize,
                     cudaMemcpyDeviceToHost, streams[i]);
}
cudaDeviceSynchronize();  // wait for all streams to finish
```

If we wish to wait for a particular stream to finish you can:

```
cudaStreamSynchronize(streams[2]);
```

Streaming - reusing streams and buffers

To properly coordinate the reuse of buffers requires some careful programing. One approach for doing this is create a fixed number of buffers and a fix number of streams and use "modulo" to rotate between their use. So on the device we would typically have some buffers for input and some for output:

```
#define NUMBUF 3
int *buffer_in_d[NUMBUF];
int *buffer_out_d[NUMBUF];
cudaStream_t streams[NUMBUF];
for (int i=0;i<NUMBUF;i++) cudaStreamCreate(&streams[i]);

int pos = 0;

while (there_is_still_data_to_process) {
    cudaMemcpyAsync( data_in_d[pos], host_data_location, datasize,
                     cudaMemcpyHostToDevice, streams[pos]);
    kernel<<<grid,block,0,streams[pos]>>>(data_in_d[pos], data_out_d[pos]);
    cudaMemcpyAsync( host_result_location, data_out_d[pos], datasize,
                     cudaMemcpyDeviceToHost, streams[pos]);
    pos = (pos+1) % NUMBUF;
}
```

References

- GPU Pro Tip: CUDA 7 Streams Simplify Concurrency, Mark Harris,

https://devblogs.nvidia.com/gpu-pro-tip-cuda-7-streams-simplify-concu

- CUDA C/C++ Streams and Concurrency, Steve Rennich,

https://developer.download.nvidia.com/CUDA/training/StreamsAndCor