CMSC 691
High Performance Distributed Systems

CUDA Asynchronous Concurrent Execution

Dr. Alberto Cano
Assistant Professor
Department of Computer Science
acano@vcu.edu

GPU processing flow so far

1. Copy input data from CPU memory to GPU memory
2. Launch a GPU kernel
3. Copy results from GPU memory to CPU memory

cudaMemcpy(HtD)	Kernel <<< >>>	cudaMemcpy(DtH)

- Kernel needs to wait input data to be transferred
- Results cannot be copied back until kernel finished

- What if we launch multiple kernels?
- Need to wait until all input data is copied to start the kernel?
- Need to wait until kernel finishes to start copying results?

Execution of multiple kernels with no dependencies

time →

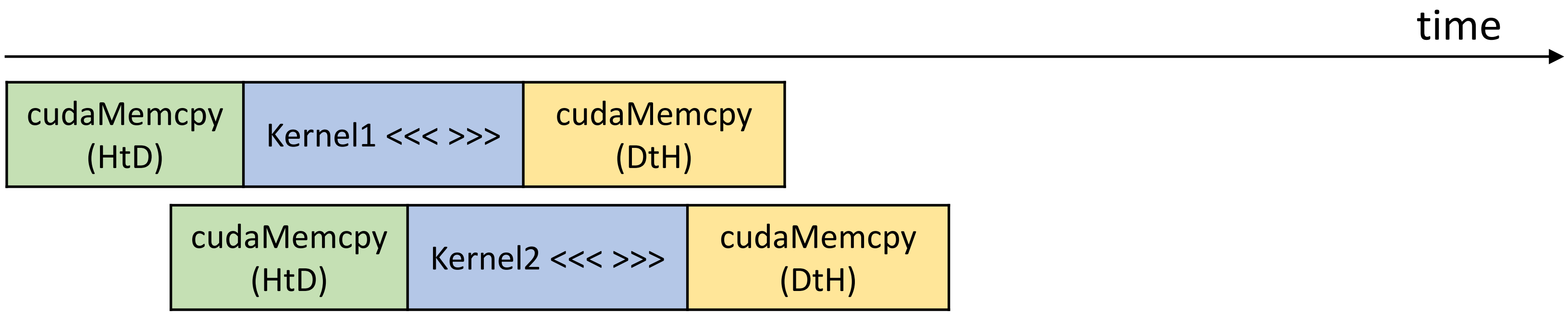
cudaMemcpy (HtD)	Kernel1 <<< >>>	cudaMemcpy (DtH)	cudaMemcpy (HtD)	Kernel2 <<< >>>	cudaMemcpy (DtH)

```
cudaMemcpy(d_input1, h_input1, size, cudaMemcpyHostToDevice);
Kernel1<<<blocks1, threads1>>>(d_input1, d_output1);
cudaMemcpy(h_output1, d_output1, size, cudaMemcpyDeviceToHost);

cudaMemcpy(d_input2, h_input2, size, cudaMemcpyHostToDevice);
Kernel2<<<blocks2, threads2>>>(d_input2, d_output2);
cudaMemcpy(h_output2, d_output2, size, cudaMemcpyDeviceToHost);
```

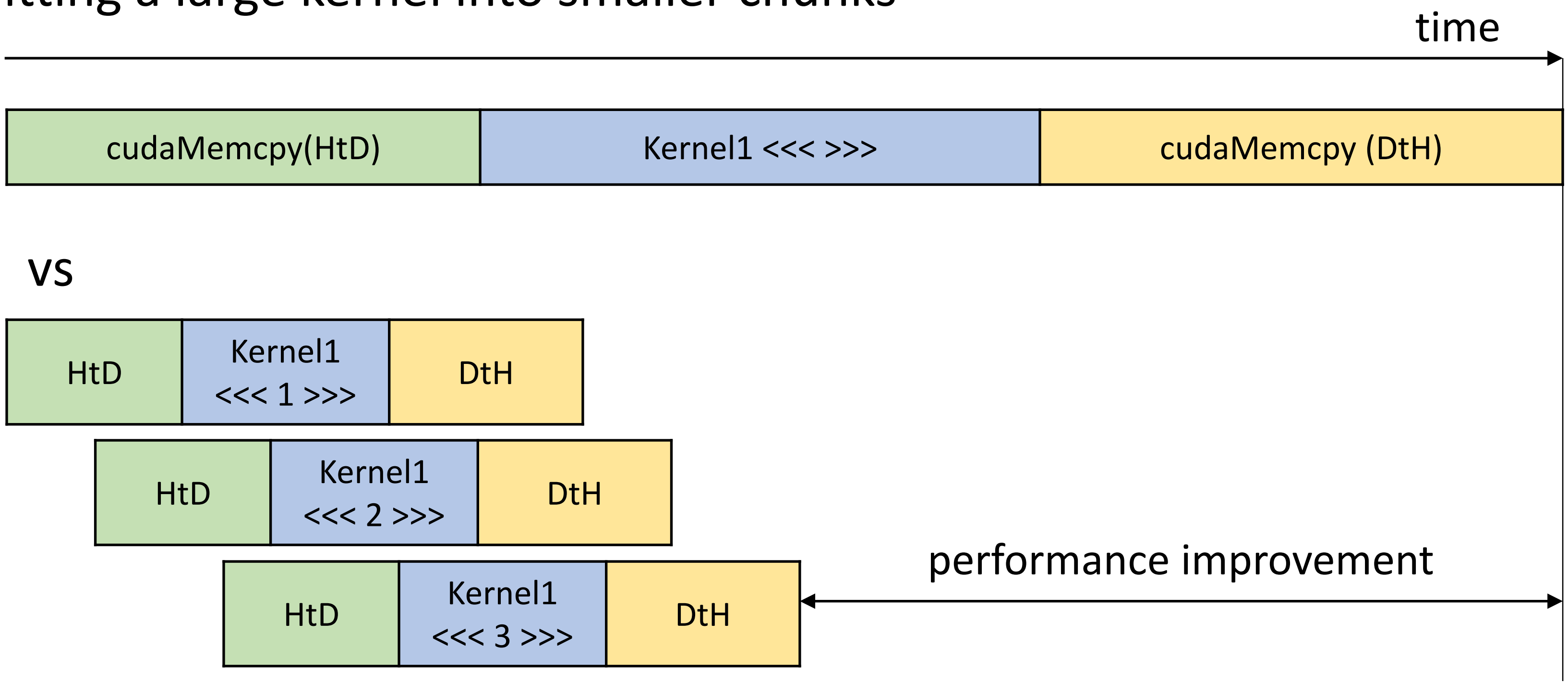
- Assuming Kernel1 and Kernel2 are data-independent
- Is it necessary to serialize the kernel executions?
- Is it necessary to serialize the data transfers?
- Can we overlap them?

Pipelining using streams



- Independent data transfer and kernel execution flows
- Maximize the occupancy of the GPU resources
- Minimize the latency of the program
- Overlapping of data transfer and execution using CUDA streams
- Streams simulate multiple pipelines

Splitting a large kernel into smaller chunks



- A given kernel is executed on a large data
- Divide into smaller chunks
- Multiple concurrent streams to process each chunk
- Overlapping of data transfer (HtD and DtH) and execution

CUDA streams

- A stream is a queue of device work
- The host places work in the queue and continues on immediately
- The device schedules work from streams when resources are free
- Operations within a stream are ordered (FIFO) and cannot overlap
- Operations in different streams are unordered and can overlap

Declaration and allocation

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

cudaStream_t *streams = (cudaStream_t*)malloc(n * sizeof(cudaStream_t));
for(int i = 0; i < n; i++) // multiple streams
    cudaStreamCreate(&streams[i]);
```

Scheduling a kernel to execute in a stream

- Kernels parameters:
  - Blocks setup (1D, 2D, 3D blocks)
  - Threads setup (1D, 2D, 3D threads)
  - Shared memory amount
  - Stream

Kernel <<< blocks, threads, smem, stream >>> ();

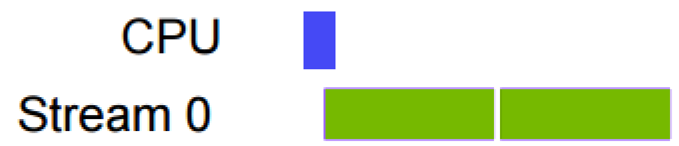
Default stream

- Default stream used when no stream is specified, referred as 0
- Completely synchronous w.r.t. host and device

Kernel concurrency

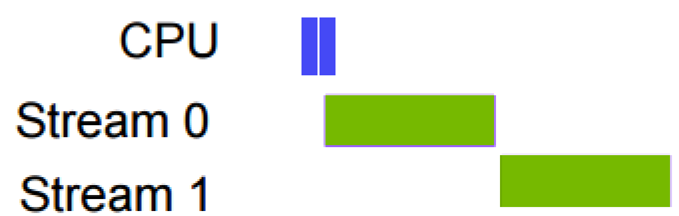
- Assume *kernel* only utilizes 50% of the GPU
- Default stream

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



- Default & user streams

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



Kernel concurrency

- Assume *kernel* only utilizes 50% of the GPU
- User streams

cudaStream_t stream1, stream2;

cudaStreamCreate(&stream1);

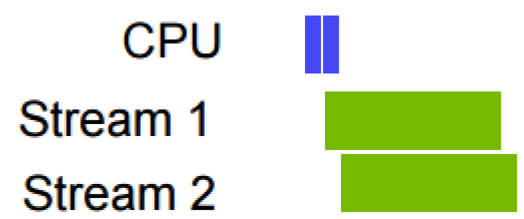
cudaStreamCreate(&stream2);

kernel <<<blocks,threads,0,stream1>>>();

kernel <<<blocks,threads,0,stream2>>>();

cudaStreamDestroy(stream1);

cudaStreamDestroy(stream2);



Requirements for concurrency

- CUDA operations must be in different streams
- *cudaMemcpyAsync* with host from pinned/page-locked memory
  - Allocate with *cudaMallocHost()* or *cudaHostAlloc()*
  - Release with *cudaFreeHost()*
- Sufficient resources must be available
  - *cudaMemcpyAsyncs* in different directions
  - Device resources (multiprocessors, registers, blocks, etc)

- Careful *cudaMemcpyAsync* are **non-blocking**!

Asynchronous memory transfers

- Transfers must be in a non-default stream
- Must use async memcpy
- 1 transfer per direction at a time
- Memory on the host must be pinned

```
cudaMallocHost(&h_ptr, bytes);
cudaMalloc(&d_ptr, bytes);
…
cudaMemcpyAsync(d_ptr,h_ptr,bytes,cudaMemcpyHostToDevice, &stream);
kernel << blocks, threads, smem, stream >>> (d_ptr);
cudaMemcpyAsync(h_ptr,d_ptr,bytes,cudaMemcpyDeviceToHost, &stream);
…
```

Concurrency examples

- Synchronous

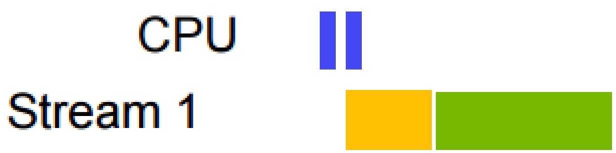
  cudaMemcpy(...);

  kernel <<<...>>>();

- Asynchronous Same Stream

  cudaMemcpyAsync(...,stream1);

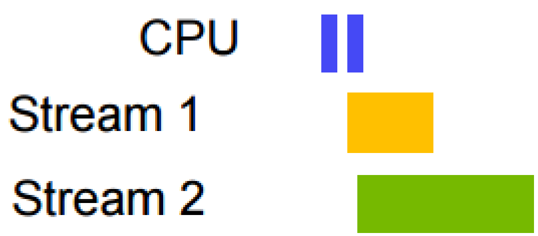
  kernel <<<...,stream1>>>();



- Asynchronous Different Streams

  cudaMemcpyAsync(...,stream1);

  kernel <<<...,stream2>>>();



Synchronization

- Synchronize everything

  cudaDeviceSynchronize()

  - Blocks until all issued CUDA calls are complete (all streams)

- Synchronize host w.r.t. a specific stream

  cudaStreamSynchronize(stream)

  - Blocks until all issued CUDA calls in stream are complete

- Synchronize host or devices using events

  - We already used events to measure elapsed time

CMSC 691
High Performance Distributed Systems

CUDA Asynchronous Concurrent Execution

Dr. Alberto Cano
Assistant Professor
Department of Computer Science
acano@vcu.edu