ECE408 / CS483 / CSE408
Summer 2025

Applied Parallel Programming

Lecture 20: Transfer and CUDA Streams
(Task Parallelism)

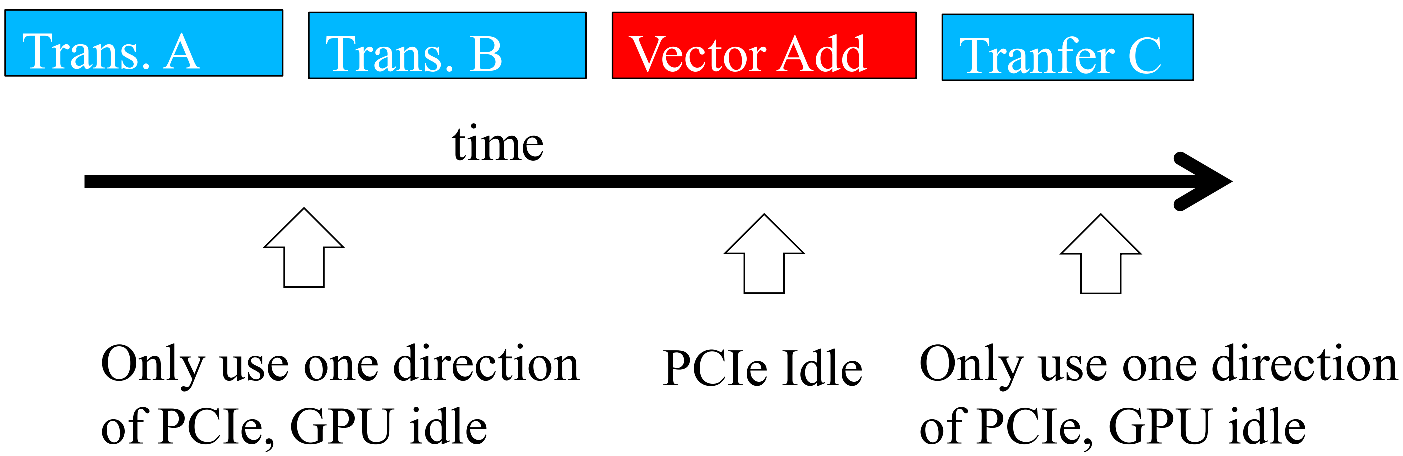
What Will You Learn Today?

more advanced features of the CUDA APIs
for data transfer and kernel launch

- task parallelism for overlapping data transfer with kernel computation
- CUDA streams

Serialized Data Transfer and GPU computation

- So far, the way we use cudaMemcpy serializes data transfer and GPU computation



Device Overlap

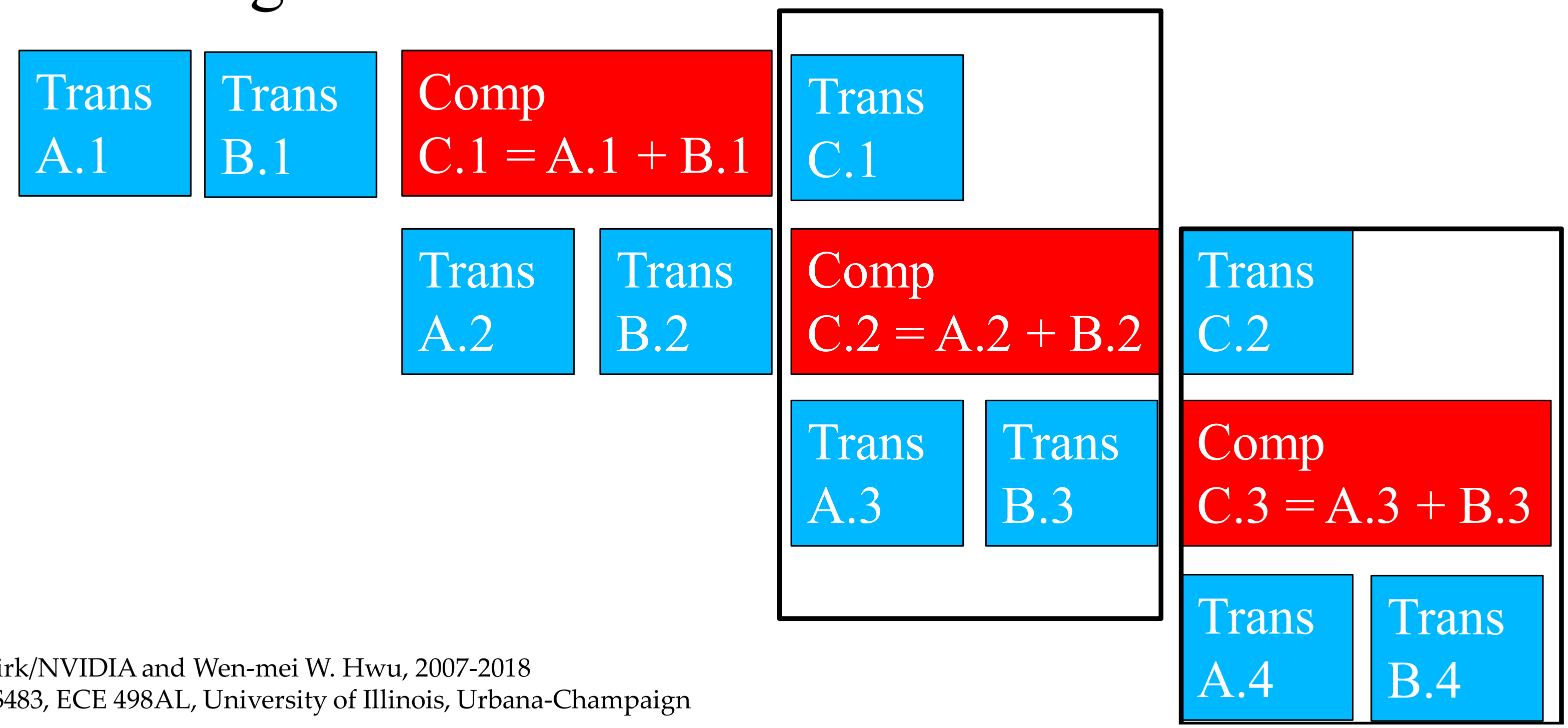
- Most CUDA devices support *device overlap*
  - *Simultaneously execute a kernel while performing a copy between device and host memory*

```
int dev_count;
cudaDeviceProp prop;

cudaGetDeviceCount( &dev_count);
for (int i = 0; i < dev_count; i++) {
  cudaGetDeviceProperties(&prop, i);

  if (prop.deviceOverlap) …
```


© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2018
ECE408/CS483/ University of Illinois at Urbana-Champaign

Overlapped (Pipelined) Timing

- Divide large vectors into segments
- Overlap transfer and compute of adjacent segments



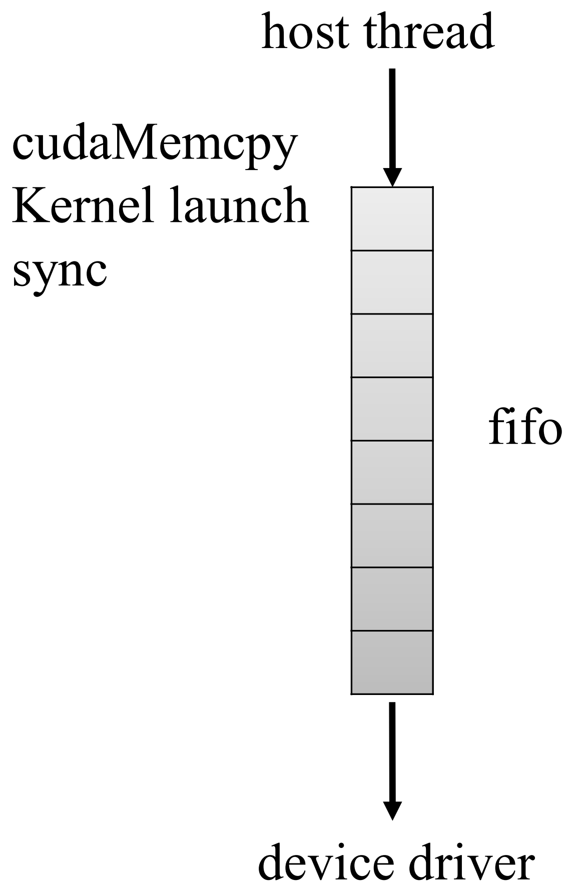
Using CUDA Streams and Asynchronous Memcpy

- CUDA supports parallel execution of kernels and cudaMemcpy with **streams**
- Each stream **is a queue of operations** (kernel launches and cudaMemcpy's)
- Operations (tasks) in different streams
  - can execute in parallel
  - a version of **task parallelism**

Streams

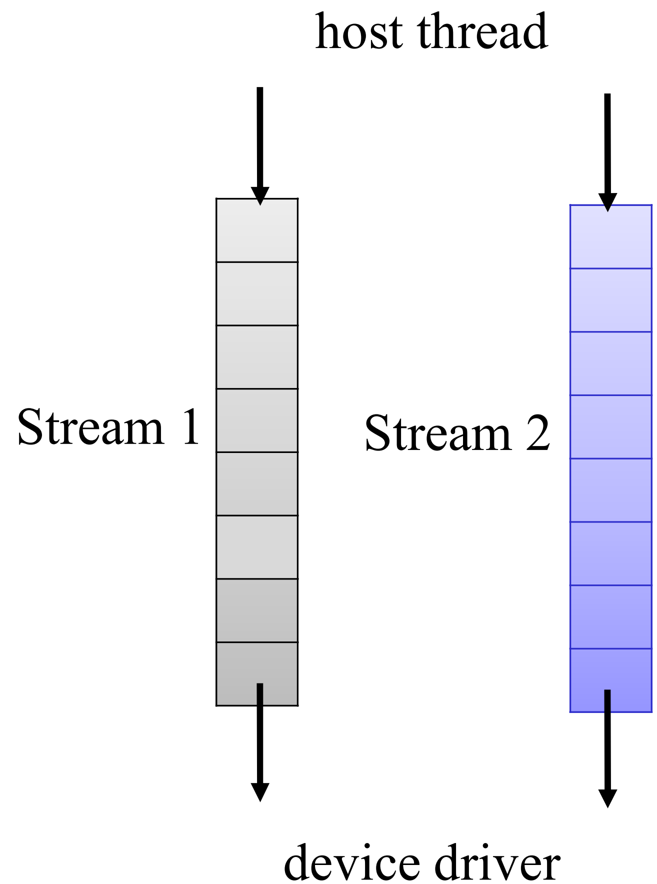
Queue operation:

1. device requests (host code) placed into queue
2. queue processed asynchronously by driver and device
3. each queue processed in order (no overlap), so memory copies end before kernel launch, and so forth

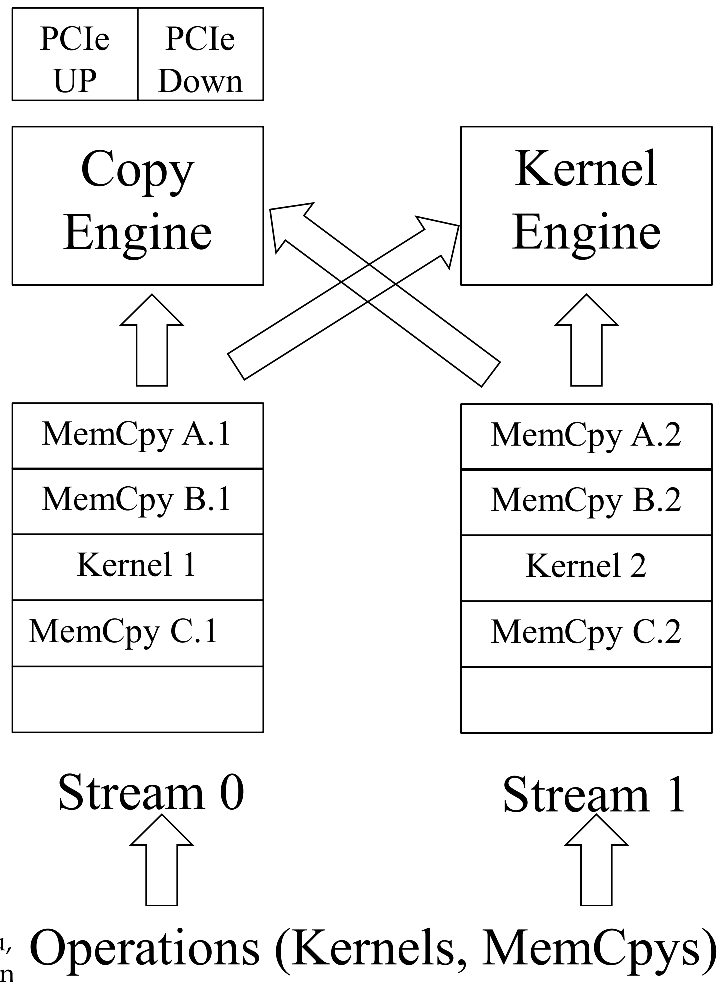


Multiple Streams Enable Parallelism

- To allow concurrent copying and kernel execution, you need to use multiple streams.



Conceptual View of Streams



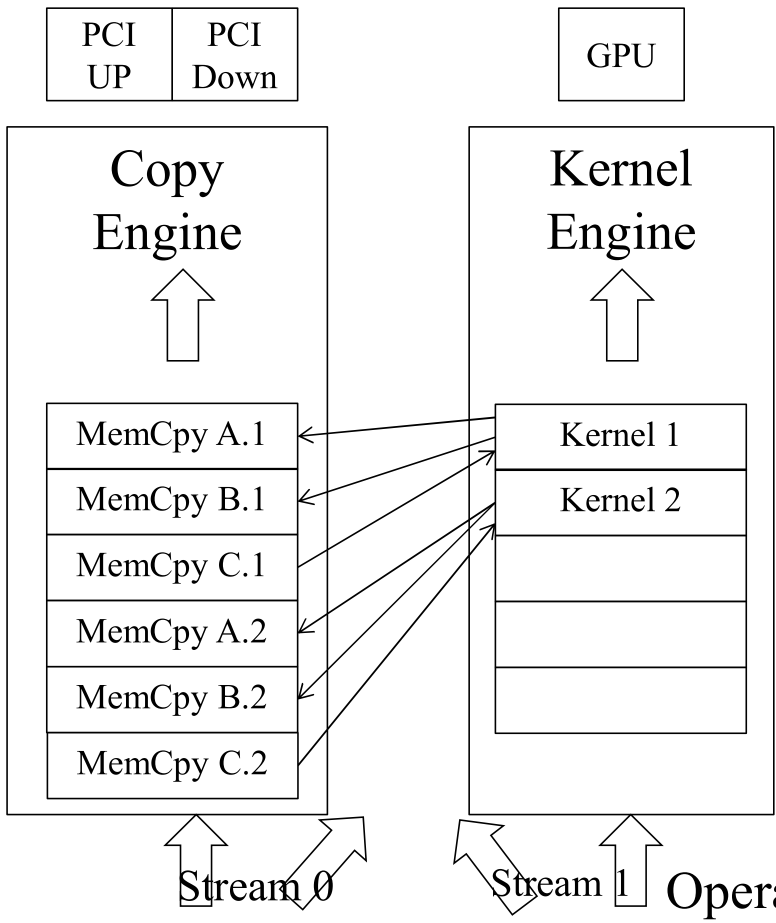
A Simple Multi-Stream Host Code

```
cudaStream_t      stream0, stream1;
cudaStreamCreate(&stream0);
cudaStreamCreate(&stream1);
float *d_A0, *d_B0, *d_C0;    // device memory for stream 0
float *d_A1, *d_B1, *d_C1;    // device memory for stream 1

// cudaMalloc for d_A0, d_B0, d_C0, d_A1, d_B1, d_C1 go here

for (int i=0; i<n; i+=SegSize*2) {
  // copy data in stream0
  // launch kernel in stream0
  // copy results in stream0
  // copy data in stream1
  // launch kernel in stream1
  // copy results in stream1
}
```

A Simple Multi-Stream Host Code

```
for (int i=0; i<n; i+=SegSize*2)
{
  cudaMemcpyAsync(d_A0, h_A+i, SegSize*sizeof(float),..., stream0);
  cudaMemcpyAsync(d_B0, h_B+i, SegSize*sizeof(float),..., stream0);
  vecAdd<<<SegSize/256, 256, 0, stream0>>>(d_A0, d_B0, …);
  cudaMemcpyAsync(h_C+i, d_C0, SegSize*sizeof(float),..., stream0);

  cudaMemcpyAsync(d_A1, h_A+i+SegSize;
                            SegSize*sizeof(float),..., stream1);
  cudaMemcpyAsync(d_B1, h_B+i+SegSize;
                            SegSize*sizeof(float),..., stream1);
  vecAdd<<<SegSize/256, 256, 0, stream1>>>(d_A1, d_B1, …);
  cudaMemcpyAsync(h_C+i+SegSize, d_C1,
                            SegSize*sizeof(float),..., stream1);
}
```

Older GPUs Support Streams in Software



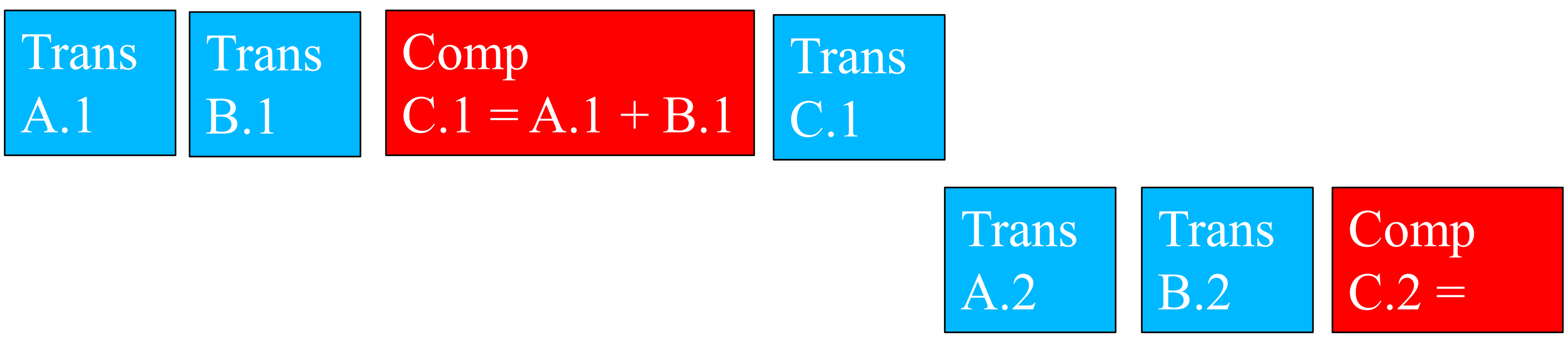
Task at head of queue waits for dependencies (arcs) before executing.

For example, Kernel 1 waits for MemCpy A.1 and MemCpy B.1 to finish.

Operations (Kernels, MemCpys)

Not quite the overlap we want

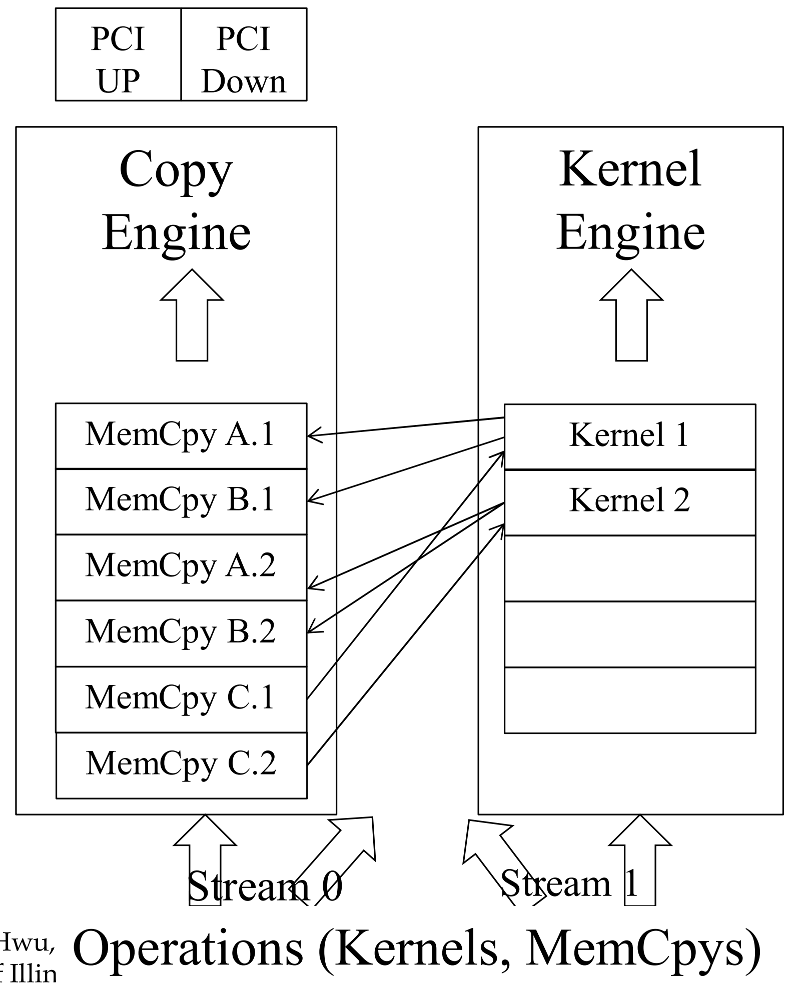
- C.1 blocks A.2 and B.2 in the copy engine queue (head-of-line blocking).



A Better Multi-Stream Host Code

```
for (int i=0; i<n; i+=SegSize*2) {

  cudaMemCpyAsync(d_A0, h_A+i; SegSize*sizeof(float),.., stream0);
  cudaMemCpyAsync(d_B0, h_B+i; SegSize*sizeof(float),.., stream0);
  cudaMemCpyAsync(d_A1, h_A+i+SegSize;
                              SegSize*sizeof(float),.., stream1);
  cudaMemCpyAsync(d_B1, h_B+i+SegSize;
                              SegSize*sizeof(float),.., stream1);

  vecAdd<<<SegSize/256, 256, 0, stream0>>>(d_A0, d_B0, …);
  vecAdd<<<SegSize/256, 256, 0, stream1>>>(d_A1, d_B1, …);
  cudaMemCpyAsync(d_C0, h_C+I; SegSize*sizeof(float),.., stream0);
  cudaMemCpyAsync(d_C1, h_C+i+SegSize;
                              SegSize*sizeof(float),.., stream1);
}
```

A View Closer to Reality



Better Overlap with Two Streams

- C.1 no longer blocks A.2 and B.2 in the copy engine queue
- However, C.2 still blocks A.1 and A.2 from the next iteration – PCIe used for only one direction

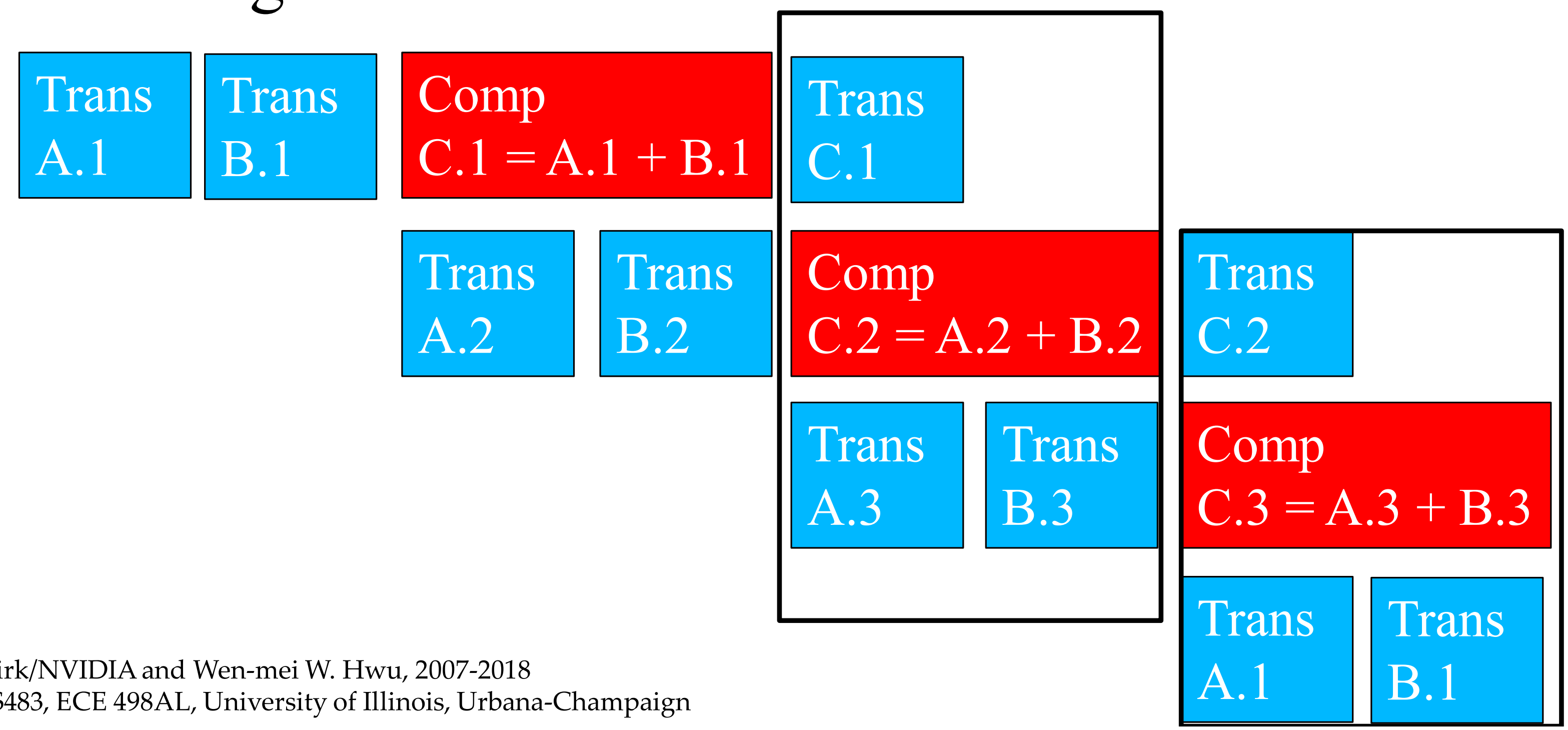
Trans A.1 | Trans B.1 | Comp C.1 = A.1 + B.1 | Trans C.1

Trans A.2 | Trans B.2 | Comp C.2 = A.2 + B.2 | Trans C.2

Next for-loop iteration ⇨ Trans A.1

Three streams needed for continuously pipelined) timing

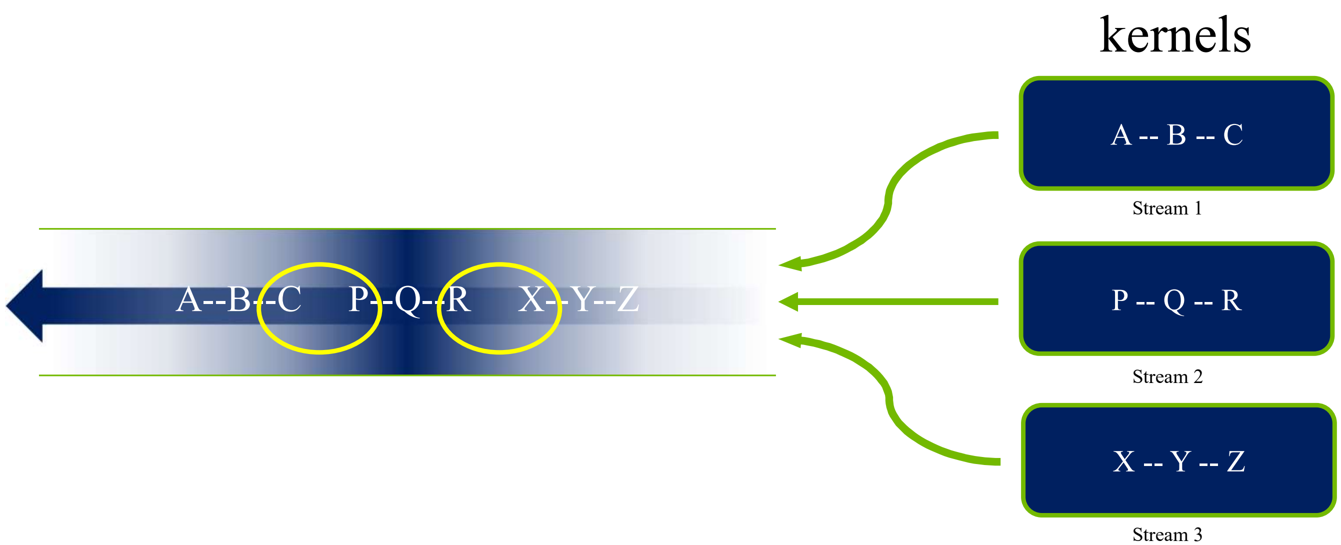
- Divide large vectors into segments
- Overlap transfer and compute of adjacent segments



Hyper Queue

- Provide multiple real stream queues for each engine
- Allow more concurrency by allowing some streams to make progress for an engine while others are blocked

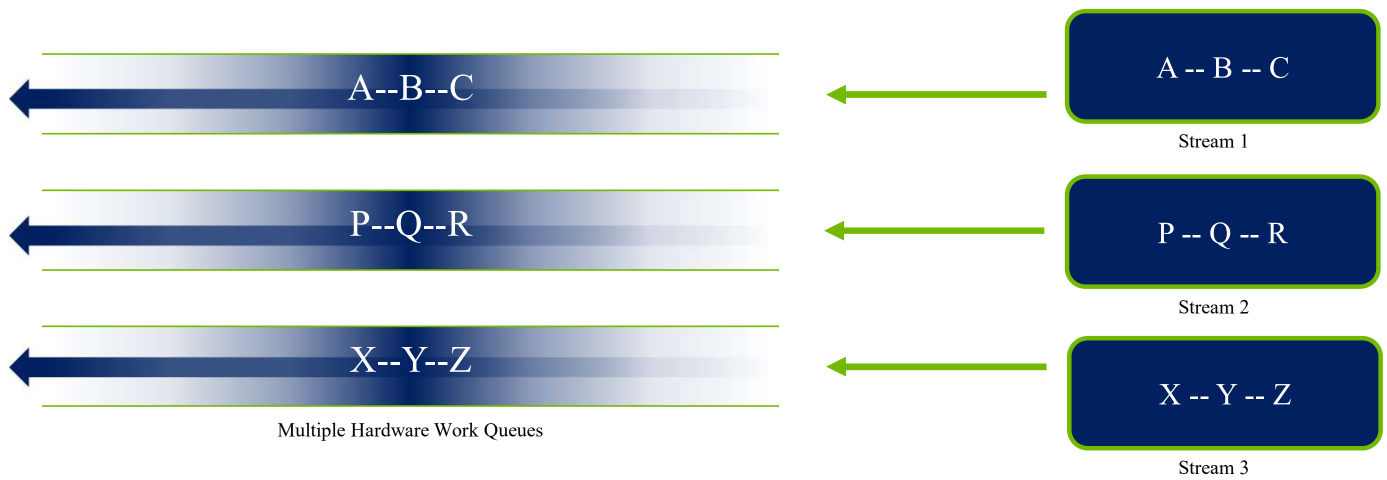
Fermi (and older) Concurrency



Fermi allows 16-way concurrency

- Up to 16 grids can run at once
- But kernels from CUDA streams multiplex into a single queue
- Overlap only at stream edges

Kepler Improved Concurrency



Kepler allows 32-way concurrency

- **One work queue per stream**
- **Concurrency at full-stream level**
- **No inter-stream dependencies**

Smaller Segments Reduce Boundary Effects

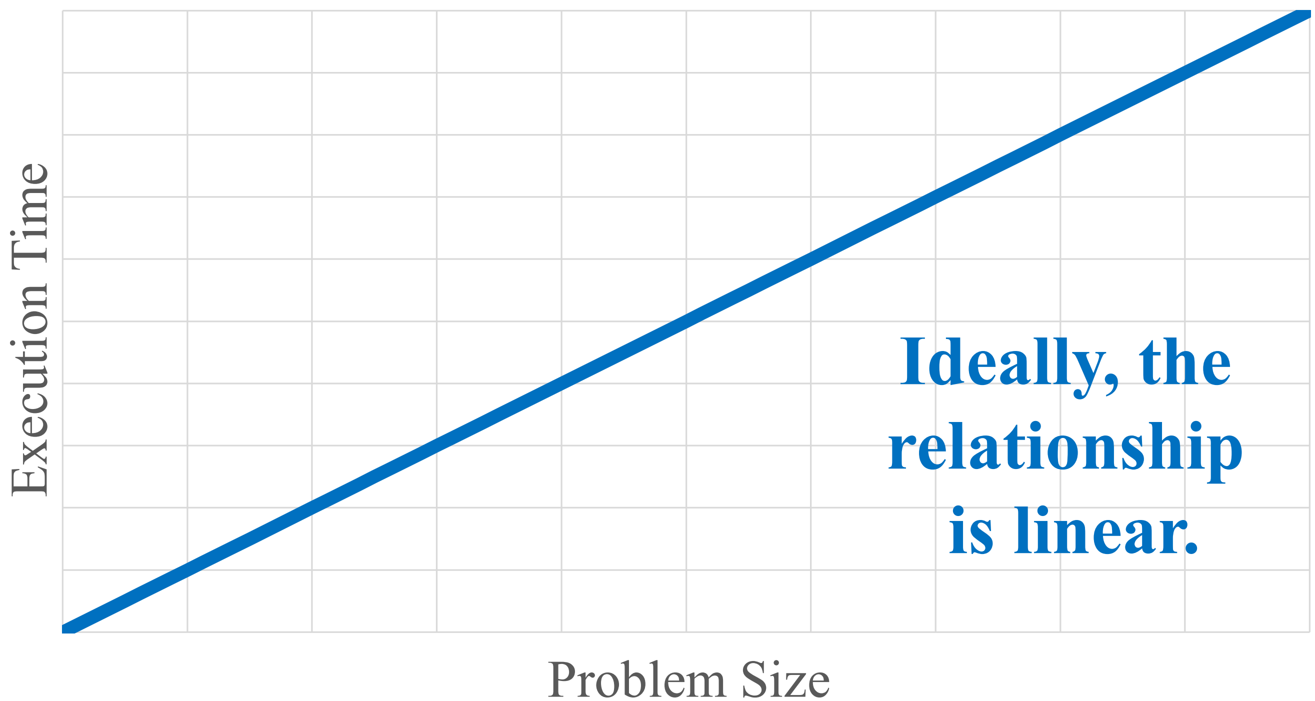
How small should segments be?

- If we **overlap**
  - **transfer of** segment N's **inputs**,
  - **computation** of segment N – 1, **and**
  - **transfer** of segment N – 2's **results**,
- we **still have non-overlapping work** at the beginning and the end.

So segments should be really small?

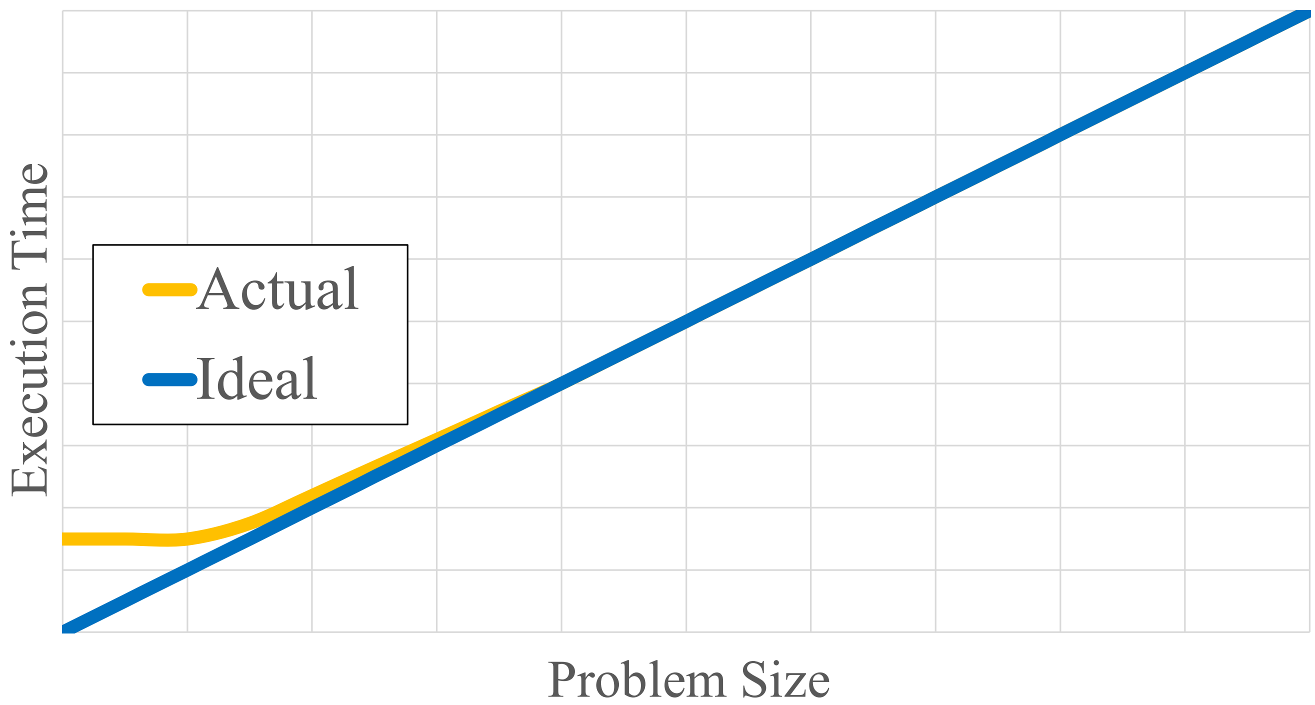
Execution Time is Ideally Linear in Size

Think about execution time as a function of segment size.



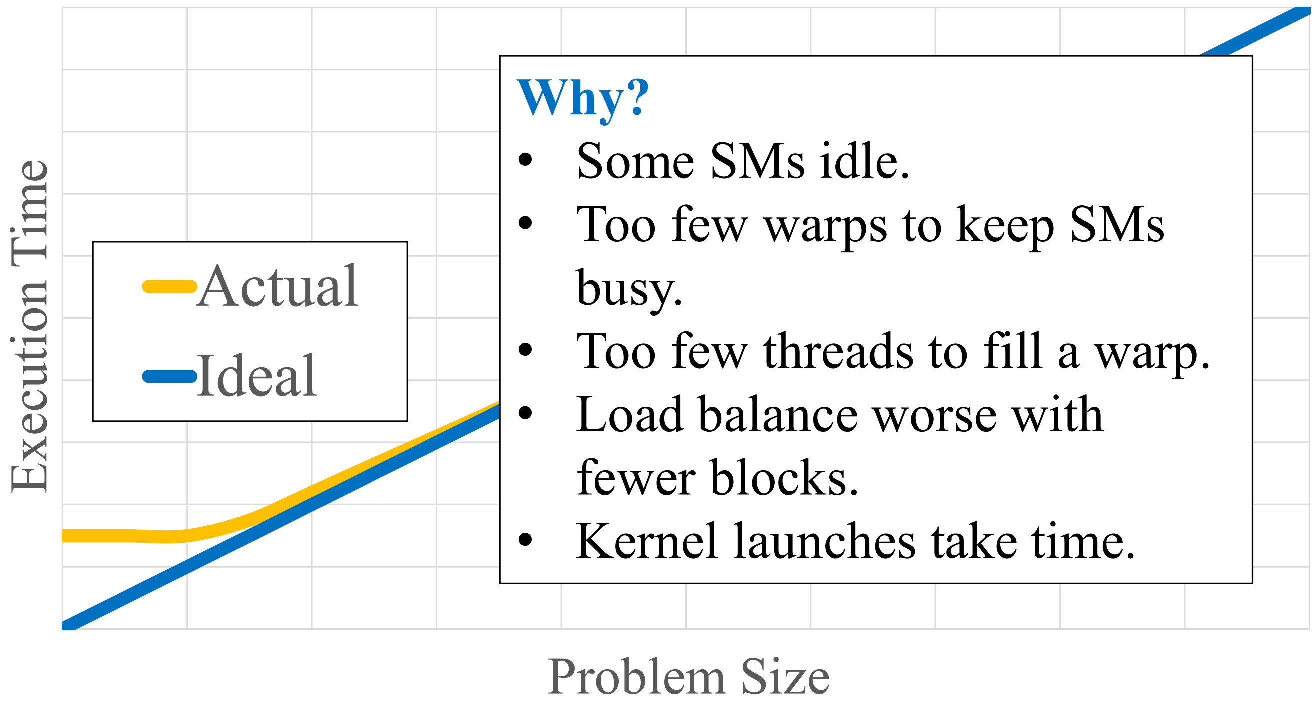
Execution Time Never Reaches Zero

But real execution time has a minimum.



Execution Time Never Reaches Zero

But real execution time has a minimum.



Why?

- Some SMs idle.
- Too few warps to keep SMs busy.
- Too few threads to fill a warp.
- Load balance worse with fewer blocks.
- Kernel launches take time.

Use Moderate Segment Size and Device Query

Data transfers

- **have similar non-linearities** for small sizes
- due to startup costs on host and DMA.

So how small should segments be?

Moderately sized.

Best size likely to **depend on GPU**.

QUESTIONS?

READ CHAPTER 13!

Problem Solving

- You are tasked with performing some operations on a very long vector. Upon profiling your kernel code with nsys, you realize that a significant portion of the execution time is taken up by serial memory transfers to and from the device. You decide to use CUDA streams to overlap the memory transfers with computation to decrease the total execution time. There is a constant overhead for starting a kernel computation on the GPU, which is independent of the size of the data that the kernel is processing. (Note that kernel launch cannot overlap with kernel execution.) The following diagram illustrates the timeline for a single stream execution (not to scale).

Input Transfer A (Host to Device)	Kernel Launch Overhead	Compute Time	Output Transfer B (Device to Host)

  The times for each of the section shown above is as follows:
  - Total Input Transfer time: 10 sec
  - Constant Kernel Launch overhead: 1 sec
  - Total Compute time: 10 sec
  - Total Output Transfer Time: 10 sec
- Q: Assume that you have access to 3 hardware streams that enable continuous pipelining. How many segments should the input vector be divided into so as to minimize the total execution time where continuous pipelining is possible?
- A: 4 or 5 segments (time for N segments is (20 / N) + 10 + N, so time for 4 or 5 is 19 sec)