ECE408/CS483/CSE408 Fall 2022

Applied Parallel Programming

# Lecture 22: Data Transfer and CUDA Streams (Task Parallelism)

# Course Reminders

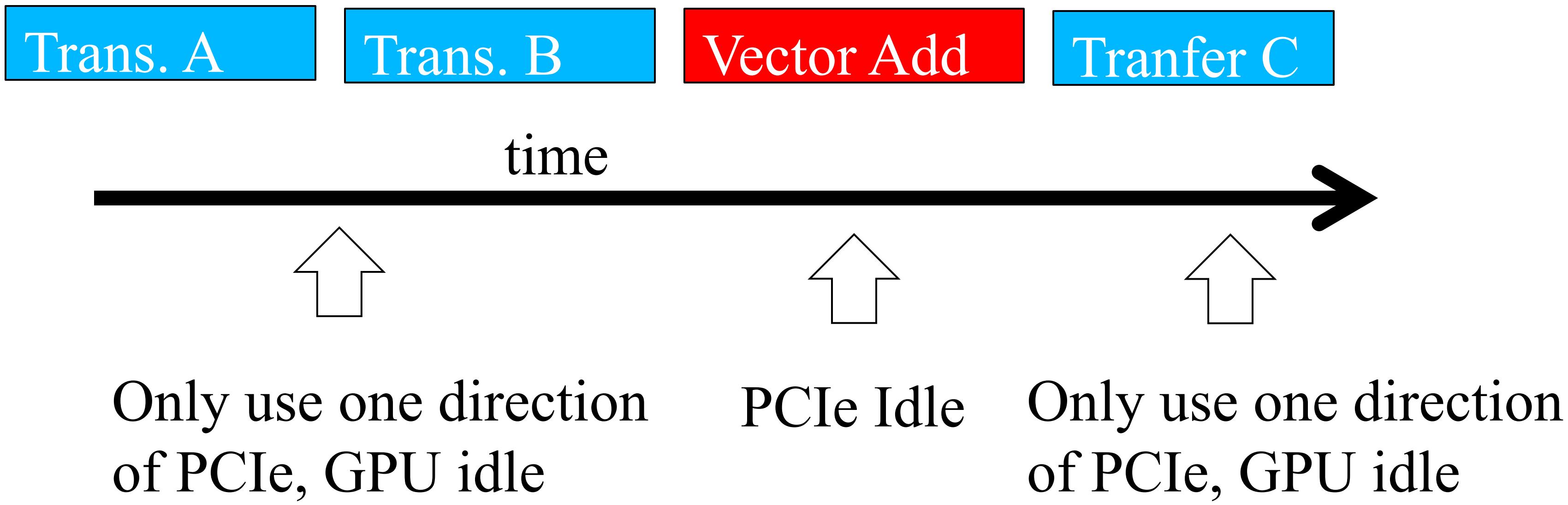
- Lab 6 is due this Friday
- PM3 is out soon
  - This is a much more involved PM, start early
  - Do not wait to submit until last minute!

# Objective

- To learn 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

- 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*

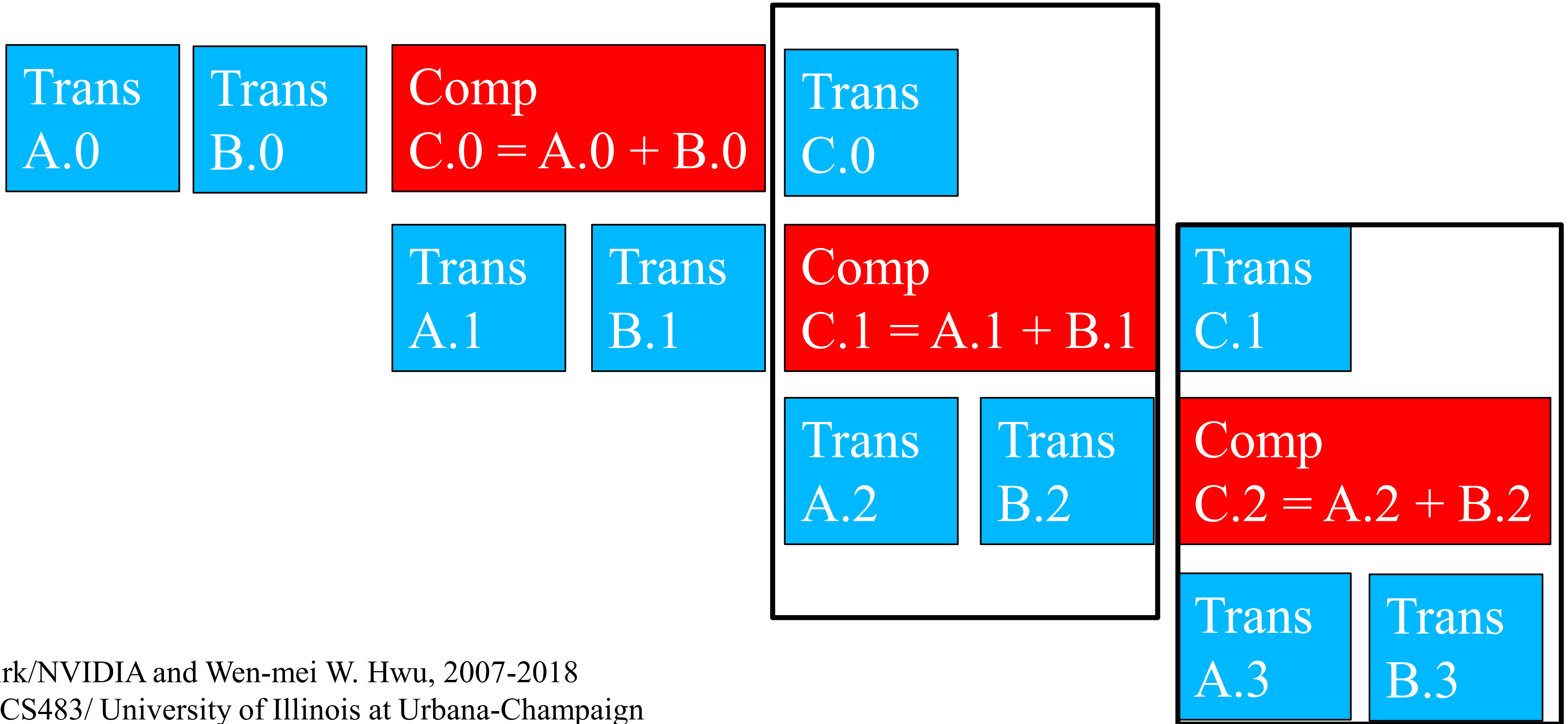
```
int dev_count;
cudaDeviceProp prop;

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

  if (prop.deviceOverlap) …
```

# Overlapped (Pipelined) Timing

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

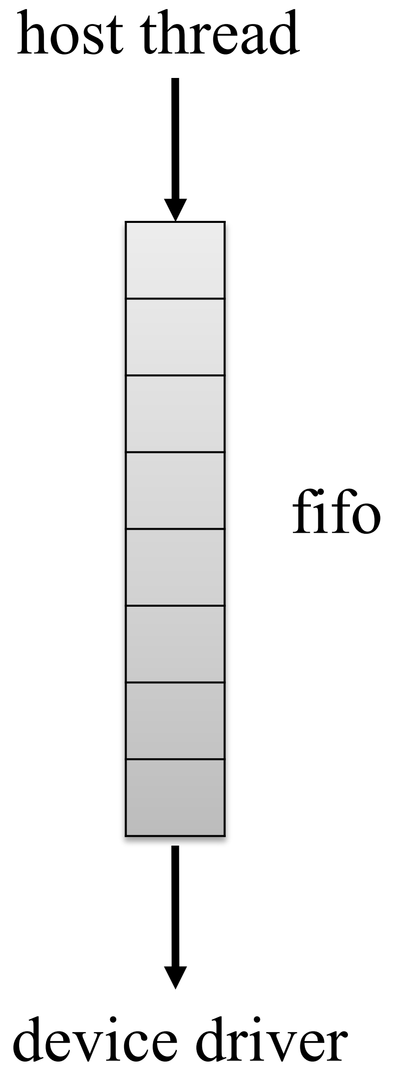


# Using CUDA Streams and Asynchronous Memcpy

- 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**

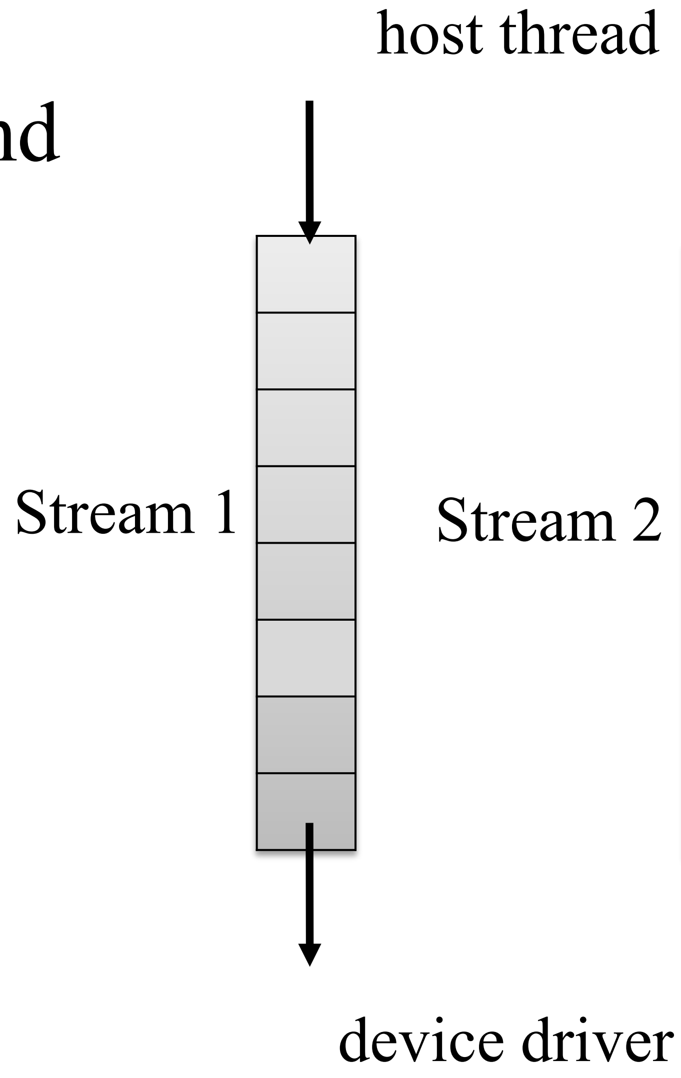
# Streams

- Device requests made from the host code are put into a queue
  - Queue processed asynchronously by the driver and device. Called a “Stream”
  - Driver ensures that commands in the queue are processed strictly in sequence. Memory copies end before kernel launch, etc.

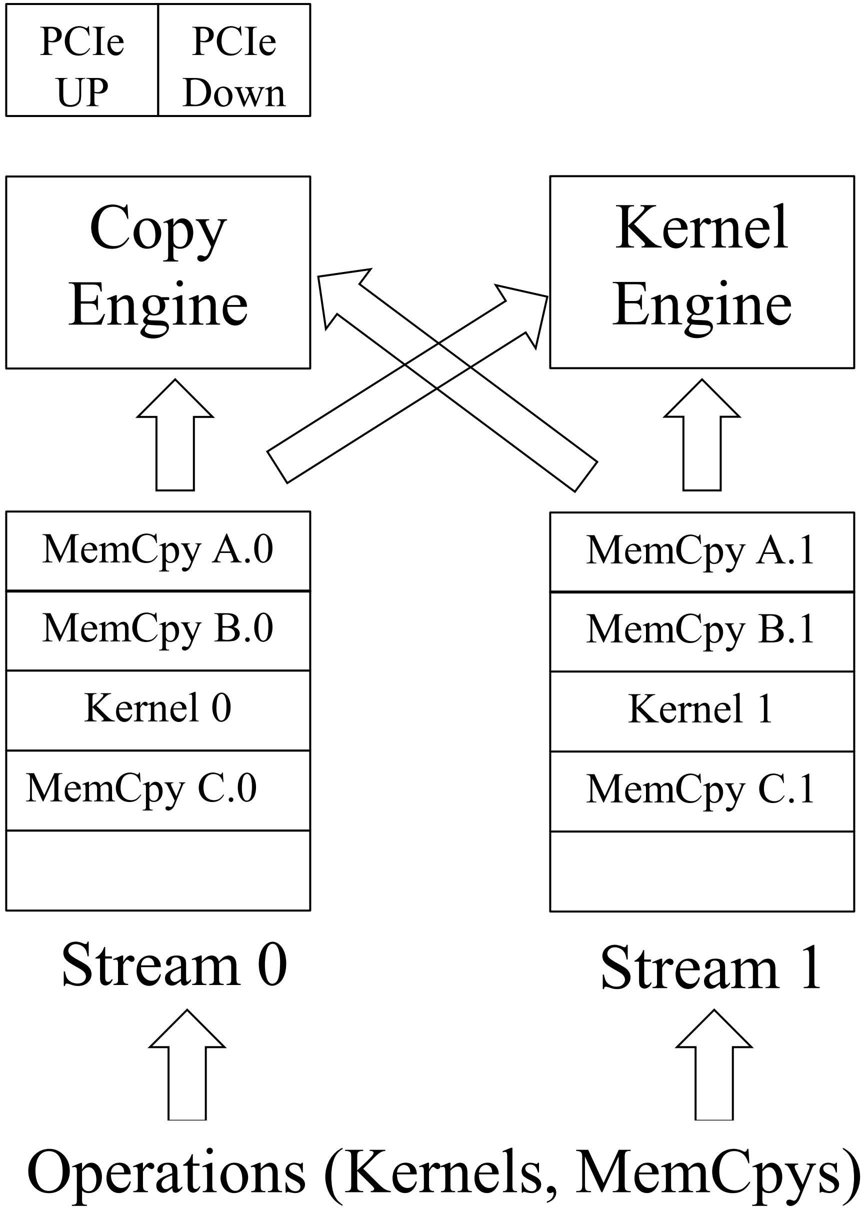


# Streams cont.

- To allow concurrent copying and kernel execution, multiple queues are required



# 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
  // lunch kernel in stream0
  // copy results in stream0
  // copy data in stream1
  // lunch kernel in stream1
  // copy results in stream1
}
```

# A Simple Multi-Stream Host Code

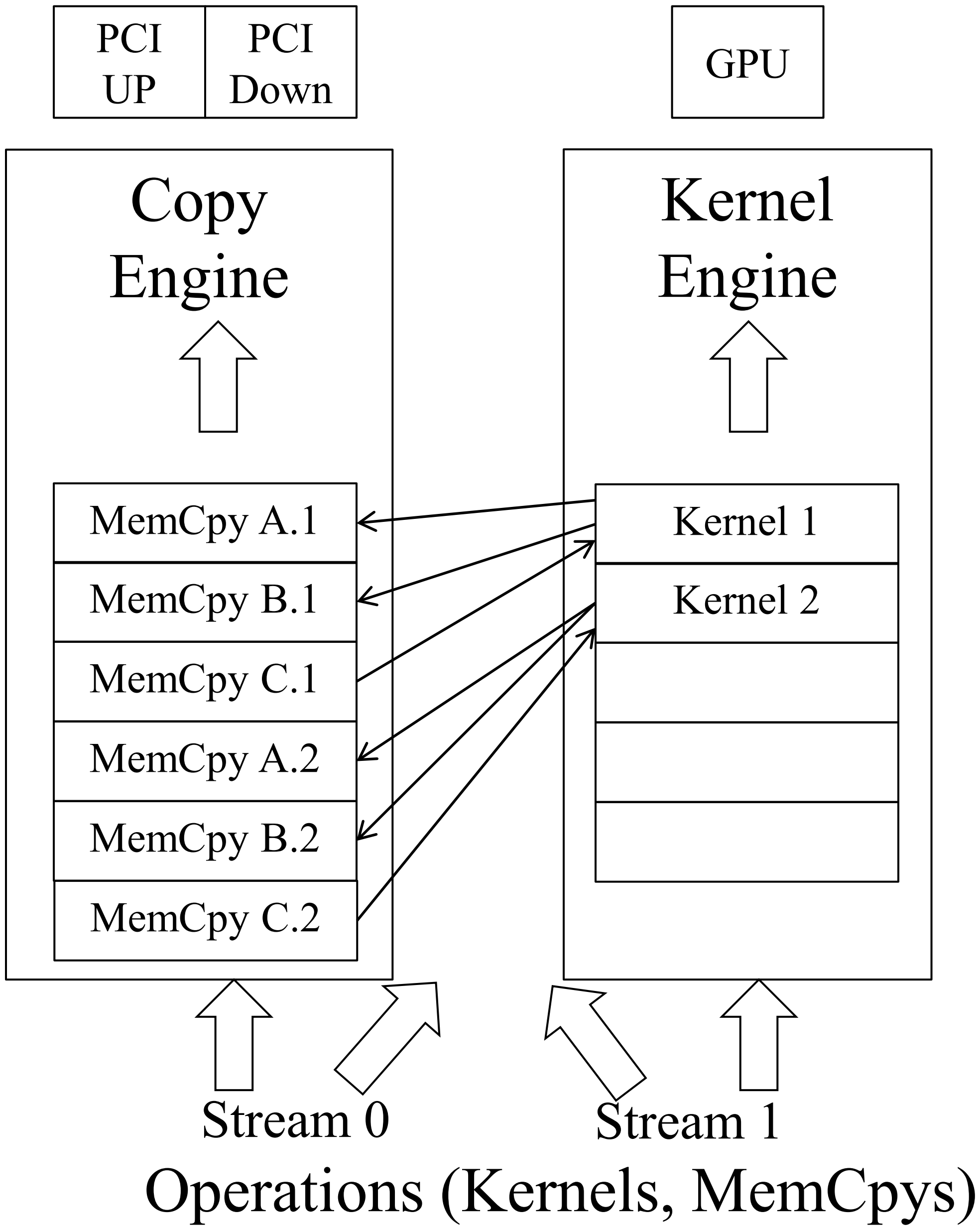
```
for (int i=0; i<n; i+=SegSize*2)
{
  cudaMemcpyAsync(d_A0, h_A+i, SegSize*sizeof(float),.., stream0);
  cudaMemcpyAsync(d_B0, h_B+i, SegSize*sizeof(float),.., stream0);
  vecAdd<<<SegSize/256, 256, 0, stream0>>>(d_A0, d_B0, …);
  cudaMemcpyAsync(h_C+i, d_C0, SegSize*sizeof(float),.., stream0);

  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

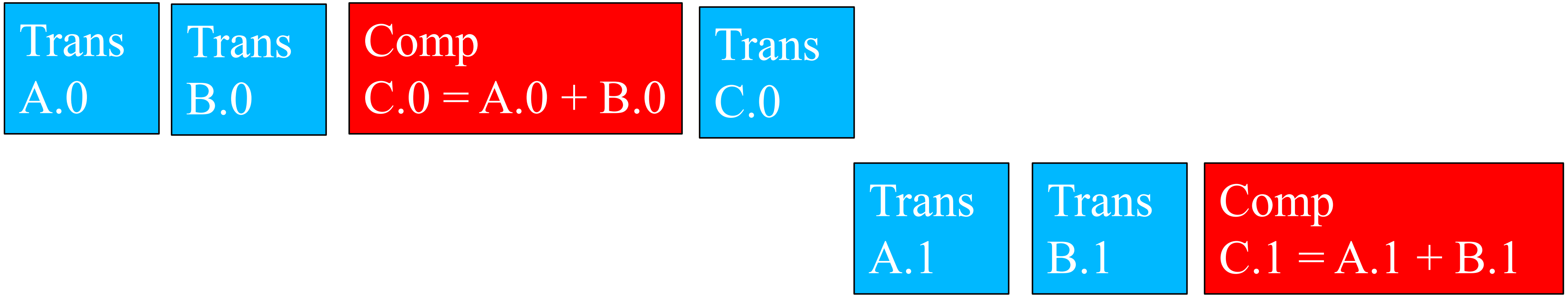
Task at head of queue waits for dependencies (arcs) before executing.

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



# Not quite the overlap we want

- C.0 blocks A.1 and B.1 in the copy engine queue



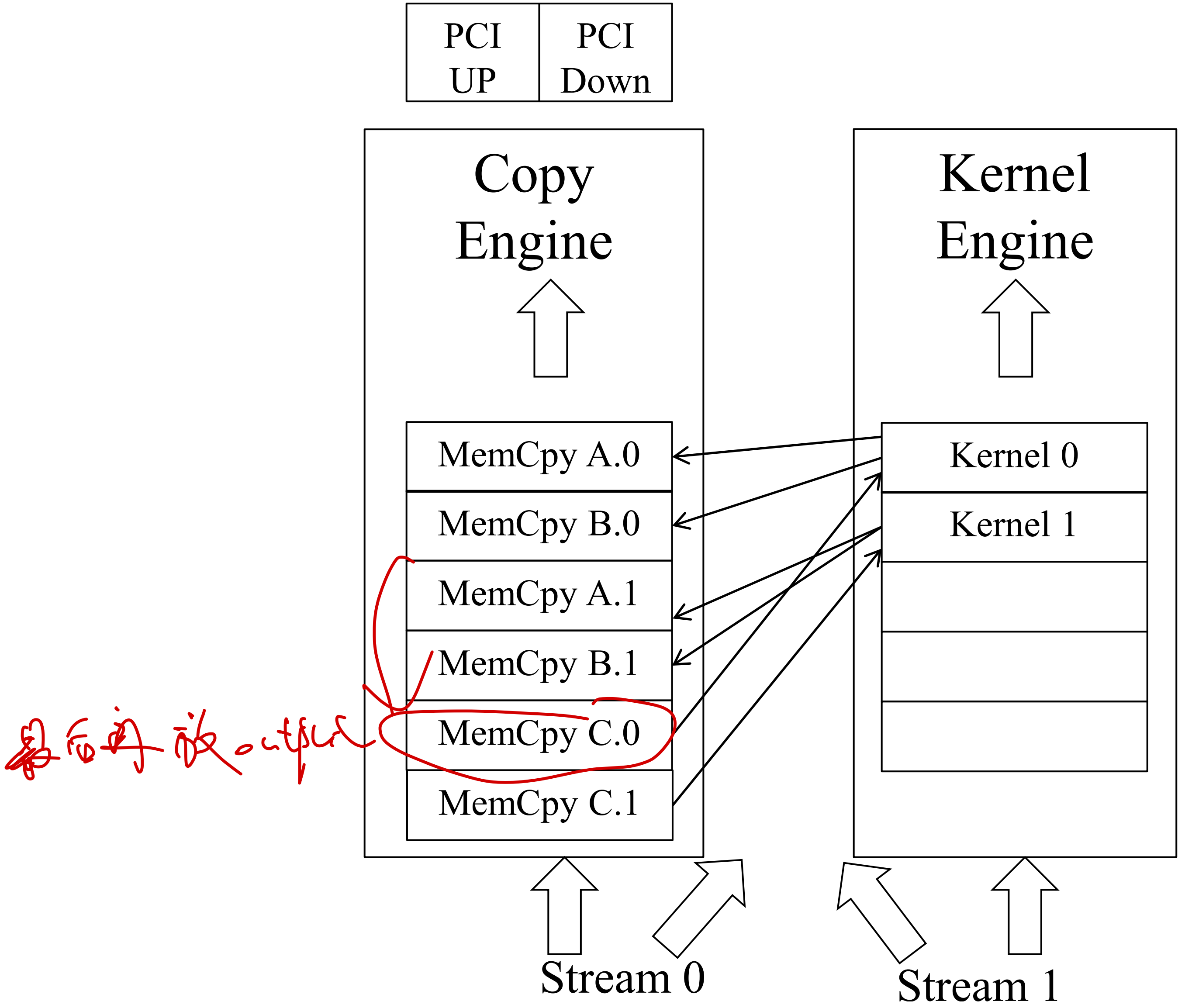
# A Better Multi-Stream Host Code

```
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);
}
```

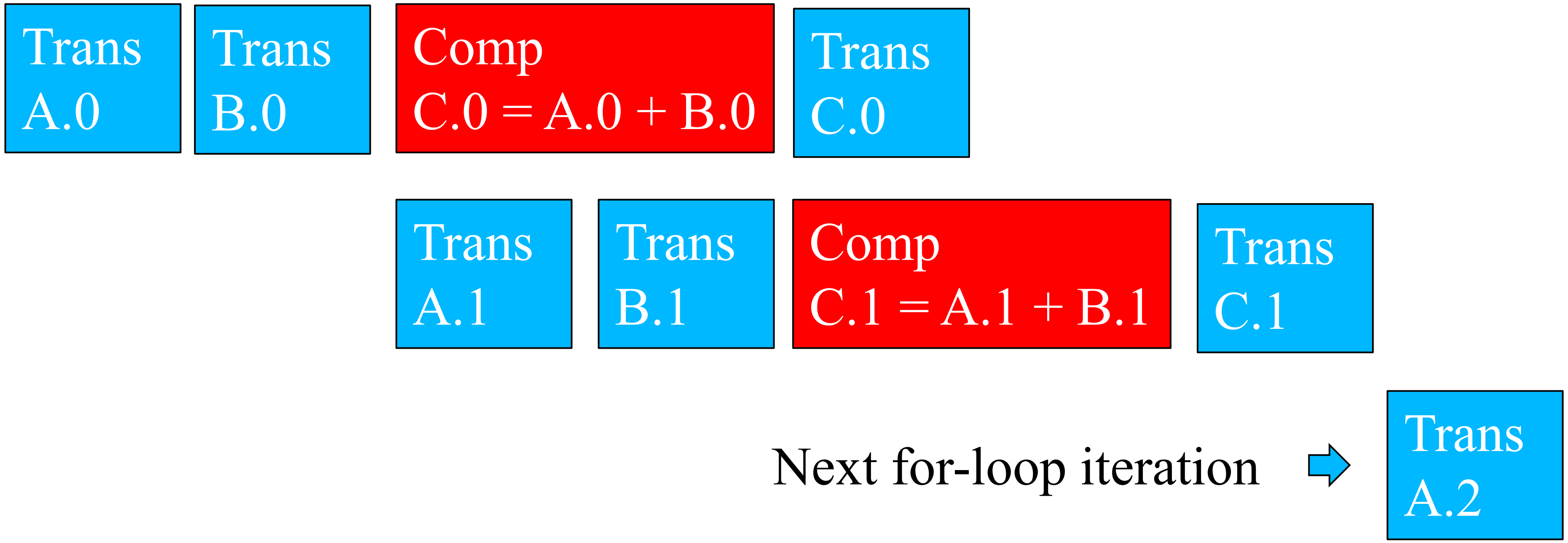
# A View Closer to Reality




Operations (Kernels, MemCpys)

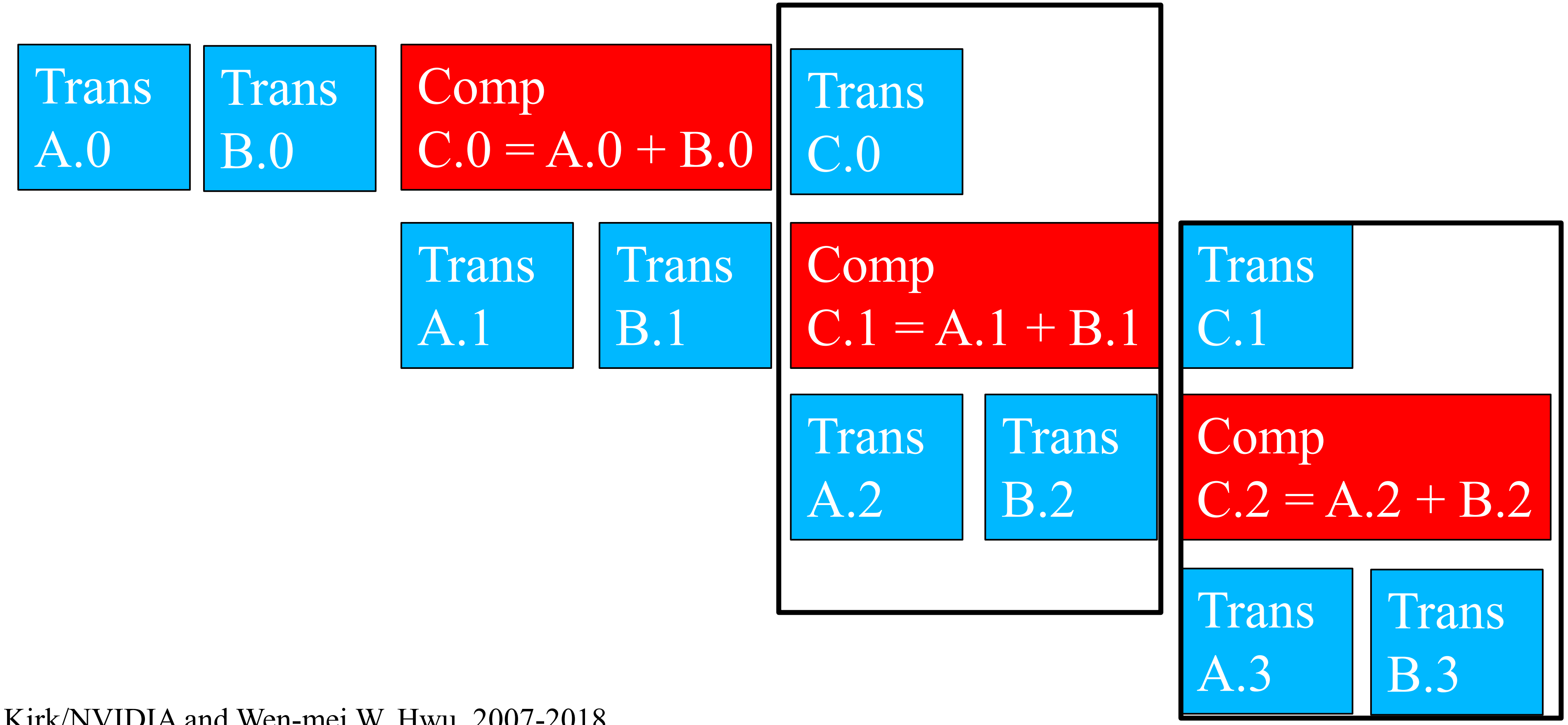
# Better Overlap with Two Streams

- C.0 no longer blocks A.1 and B.1 in the copy engine queue
- However, C.1 still blocks A.2 and B.2 from the next iteration – PCIe used for only one direction



# Three streams needed for continuous pipelining

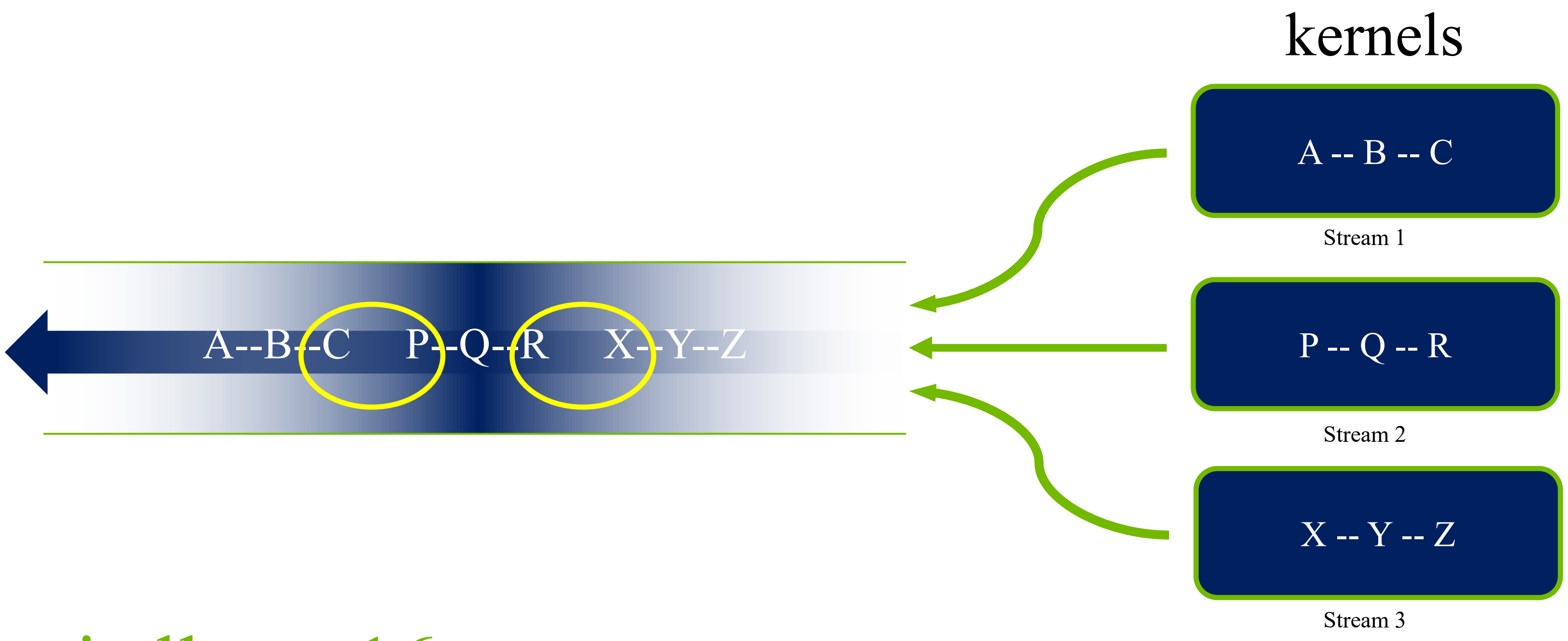
- Divide large vectors into segments
- Overlap transfer and compute of adjacent segments



# 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

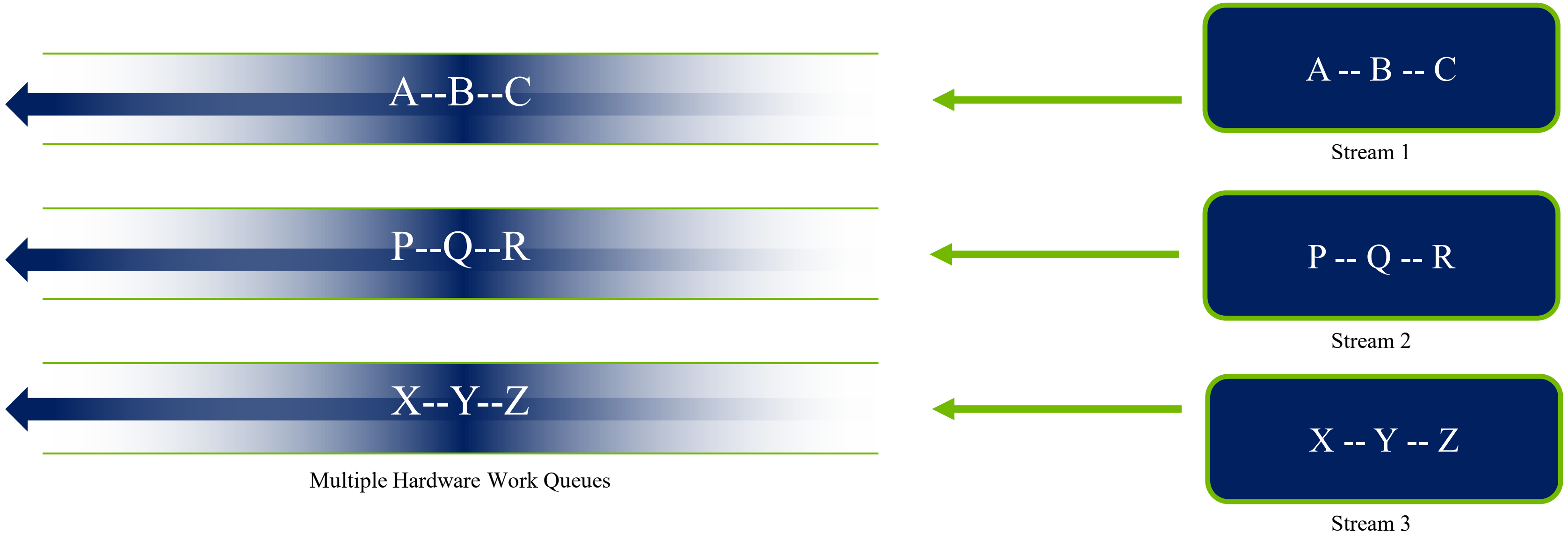
# Fermi (and older) Concurrency




## 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

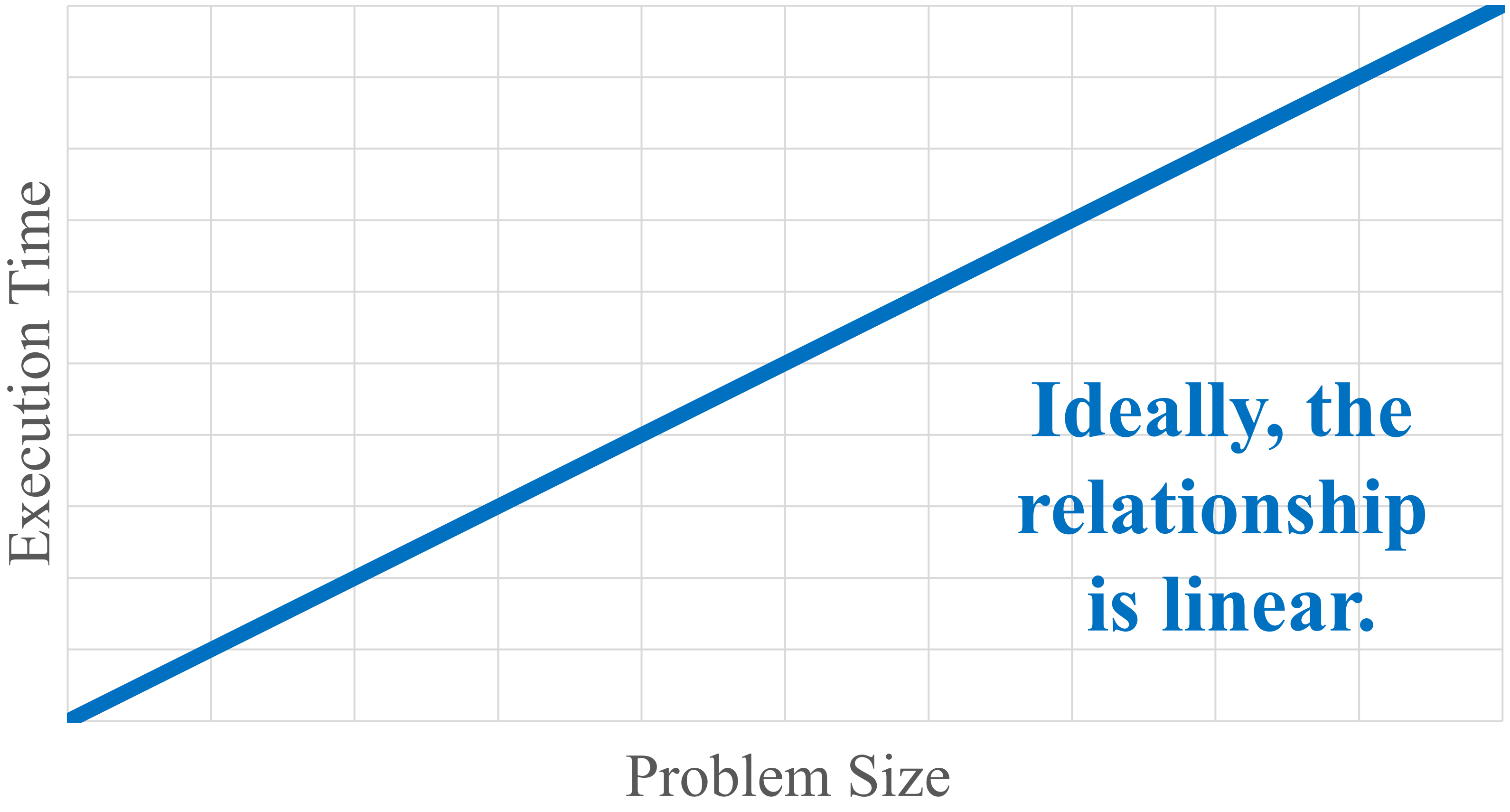
**How small should segments be?**

- 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?**

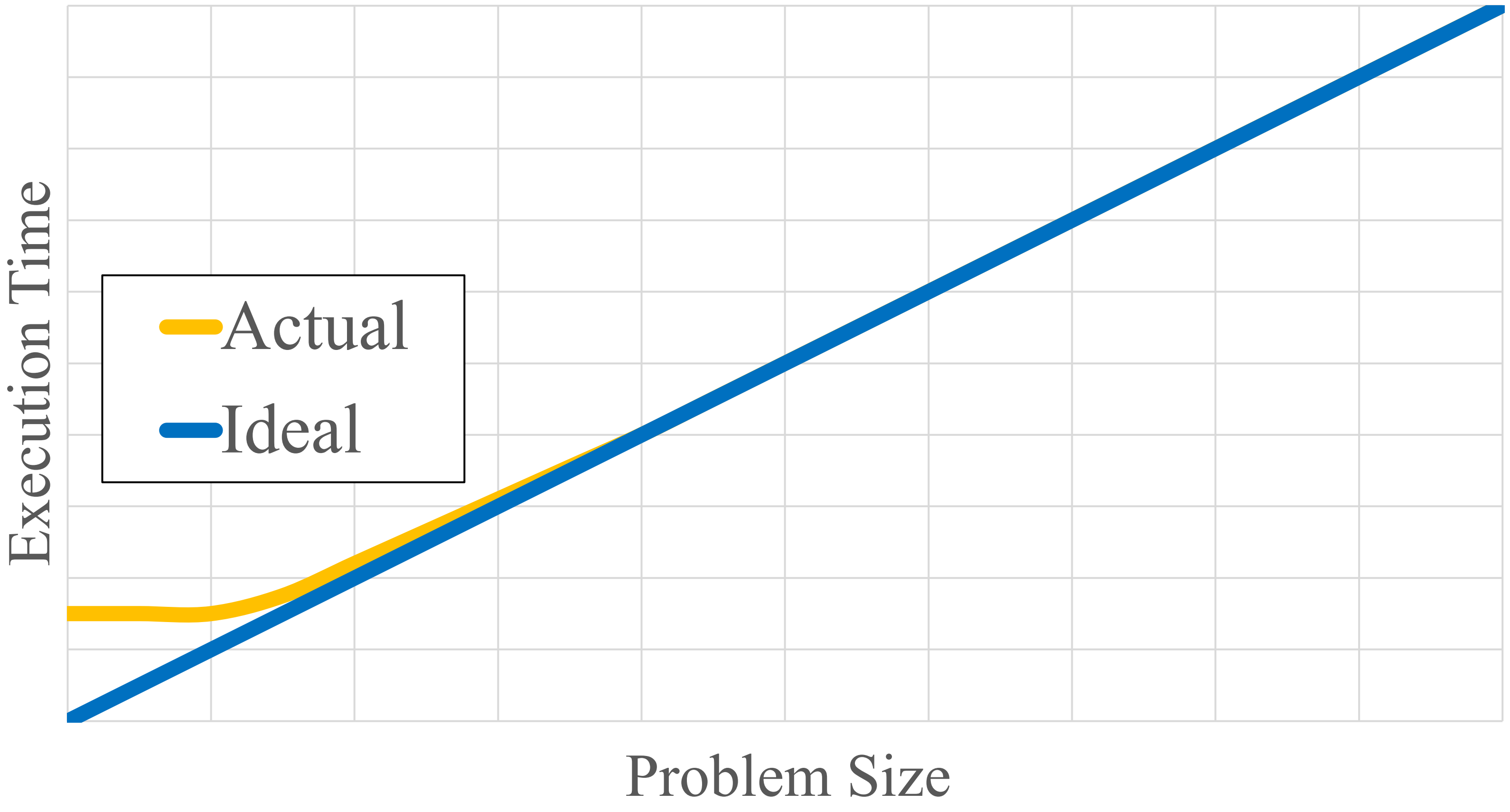
# Execution Time is Ideally Linear in Size

**Think about execution time as a function of segment size.**



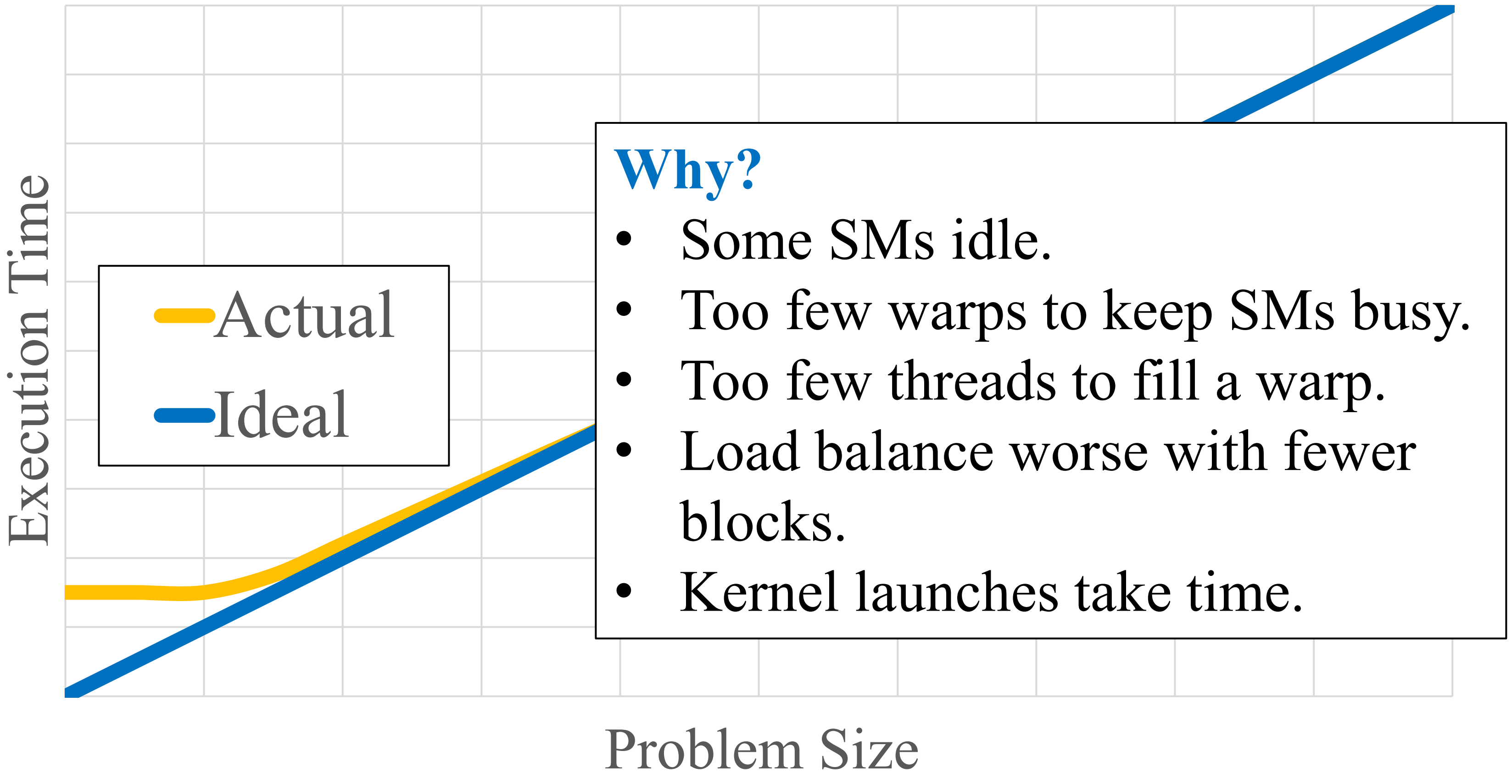
# Execution Time Never Reaches Zero

**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**.

# ANY MORE QUESTIONS
# READ CHAPTER 13