GPU COMPUTING
LECTURE 09 - HOST-DEVICE OPTIMIZATIONS

Kazem Shekofteh
kazem.shekofteh@ziti.uni-heidelberg.de
Institute of Computer Engineering
Ruprecht-Karls University of Heidelberg

Inspired from lectures by Holger Fröning

GPUS AS PERIPHERAL DEVICES

Mismatch on-device vs. off-device bandwidth

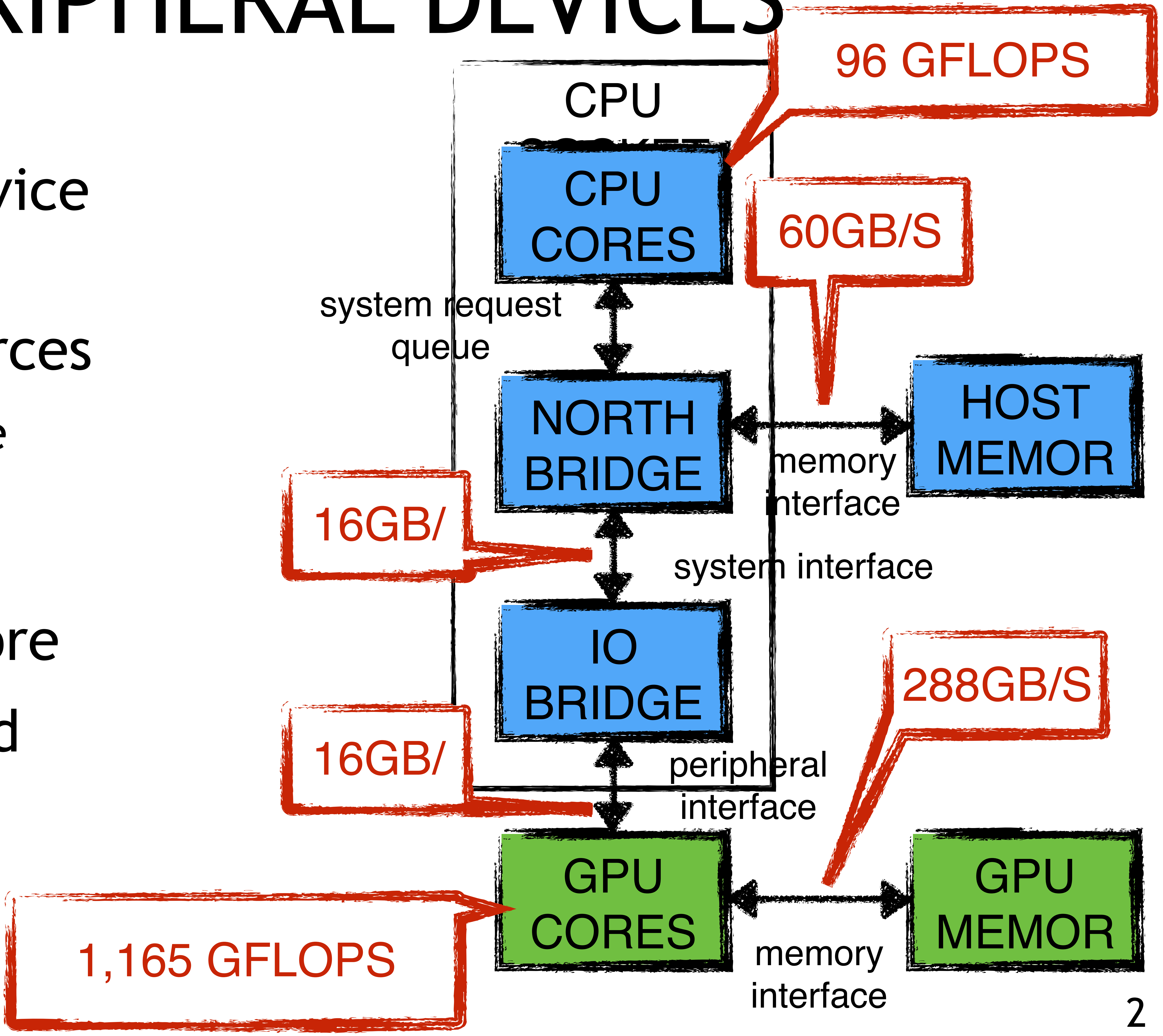
GPU memory is a scarce resources

Big Data/Deep Learning push the requirements dramatically

GPUs typically excel when performing computations in-core

Data movement is orchestrated manually

See last slides though



STREAMS FOR TASK PARALLELISM

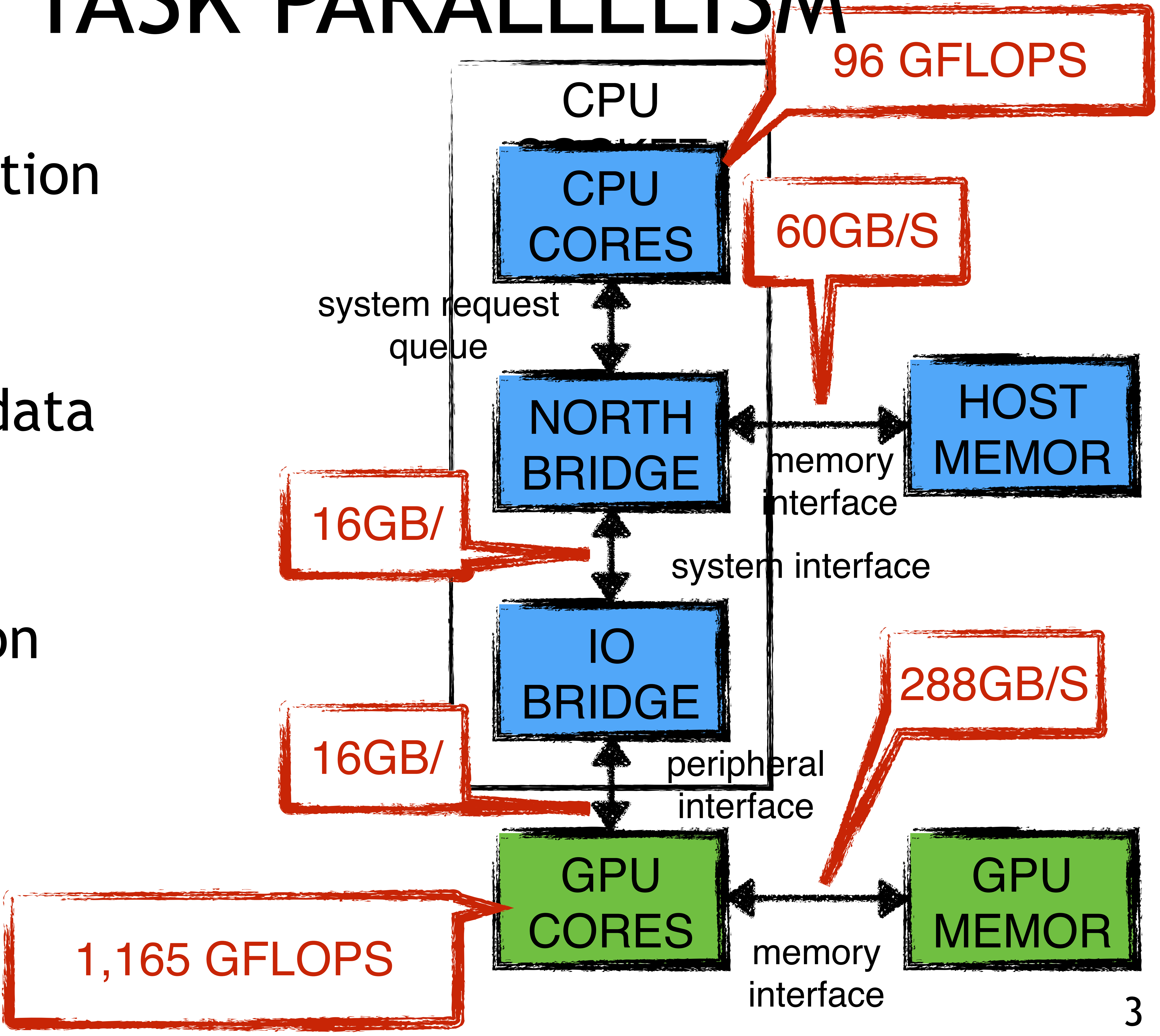
Objective: overlap communication and computation

- Overcome PCIe bottleneck

Up to now: kernels to exploit data parallelism, host code still sequential

Now: exploit task parallelism on the host side

- GPUs are accelerators, thus CPU overhead should be as small as possible



CUDA STREAMS

EXTENDED CONCURRENCY

GPUs are well-known to exploit fine-grained concurrency

- Data-level parallelism
- (-> Instruction-level parallelism/Thread-level parallelism)

Overlapping threads & instructions

Streams extend this to coarse-grained concurrency

- CPU/GPU concurrency
- Concurrent copy & execute (memcpy & kernel execute)
- Kernel concurrency (CC 2.x and later can run multiple kernels in parallel)
- Multi-GPU concurrency (multiple GPUs in one host can operate in parallel)

Overlapping kernels & memcpys

CUDA STREAMS: HOST-DEVICE SYNCHRONIZATION

Context-based

- Block until all outstanding CUDA operations have completed
  - `cudaMemcpy()`, `cudaDeviceSynchronize()`, …

Stream-based

- Block until / test if all outstanding CUDA operations in a stream have completed
  - `cudaStreamSynchronize (stream)`
  - `cudaStreamQuery (stream)` -> `cudaSuccess` or `cudaErrorNotReady`

Event-based

- Record an event in a stream; when event is dequeued it is time-stamped
  - `cudaEventRecord (event, stream)`
- Block until / test if event has been dequeued (recorded)
  - `cudaEventSynchronize (event)` or `cudaEventQuery (event)`

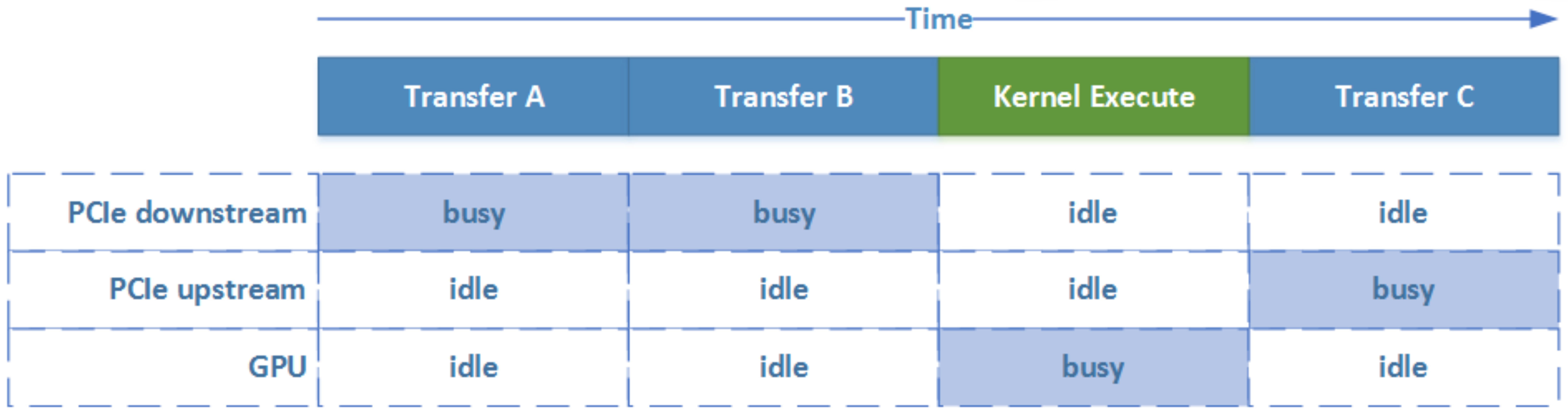
SERIALIZED DATA TRANSFER AND KERNEL EXECUTION

Data movements and kernel executions are serialized by the way we currently use cudaMemcpy

Remember Amdahl‘s law (serial and parallel fraction)

Example of a SAXPY operation

$$y[i] = alpha \cdot x[i] + y[i]$$



	Transfer A	Transfer B	Kernel Execute	Transfer C
PCIe downstream	busy	busy	idle	idle
PCIe upstream	idle	idle	idle	busy
GPU	idle	idle	busy	idle

DEFAULT STREAM & DEVICE OVERLAP

```
cudaMemcpy ( dx, hx, numBytes,
             cudaMemcpyHostToDevice);
saxpy <<<numBlocks,blockSize>>> (dx, ..);
cudaMemcpy ( hx, dx, numBytes,
             cudaMemcpyDeviceToHost);
```

Naming no stream means all memcpys/kernel launches operate on the default stream

=> Inherent synchronization

```
int dev_count;
cudaDeviceProp prop;

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

    if (prop.deviceOverlap) ...
```

Most recent CUDA devices support *Device Overlap*

Called “Concurrent copy and execute” in DeviceQuery

Simultaneously execute a kernel and a H2D/D2H copy

PIPELINING DATA TRANSFERS WITH KERNEL EXECUTION

Divide large data structures into segments

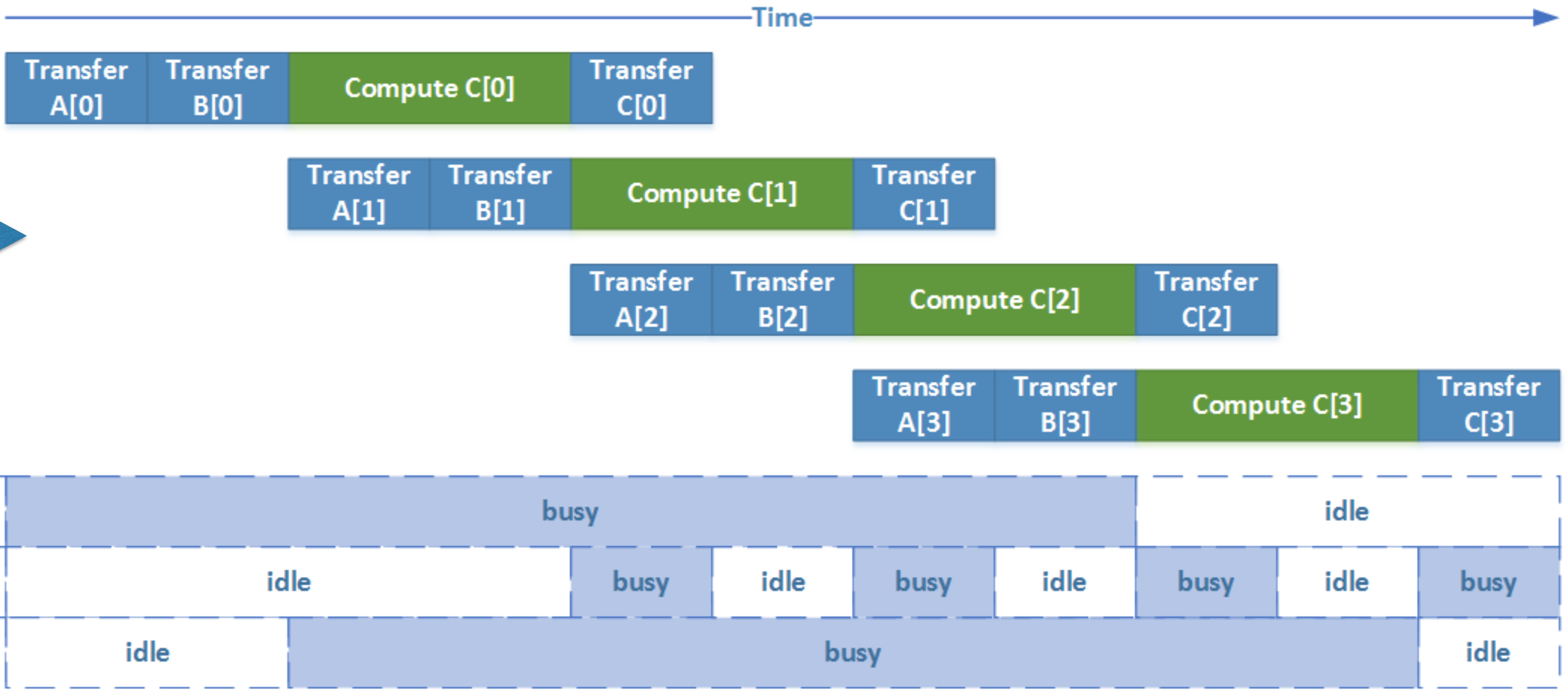
Identify independent segments

Overlap data transfer with kernel execution

Issues

- Kernel launch overhead
- Computational intensity

Pipeline phases
Fill
Steady state
Drain



PCIe downstream	busy						idle		
PCIe upstream	idle	busy	idle	busy	idle	busy	idle	busy	
GPU	idle	busy							idle

CONCEPT OF CUDA STREAMS

CUDA streams allow for simultaneous copy and execute

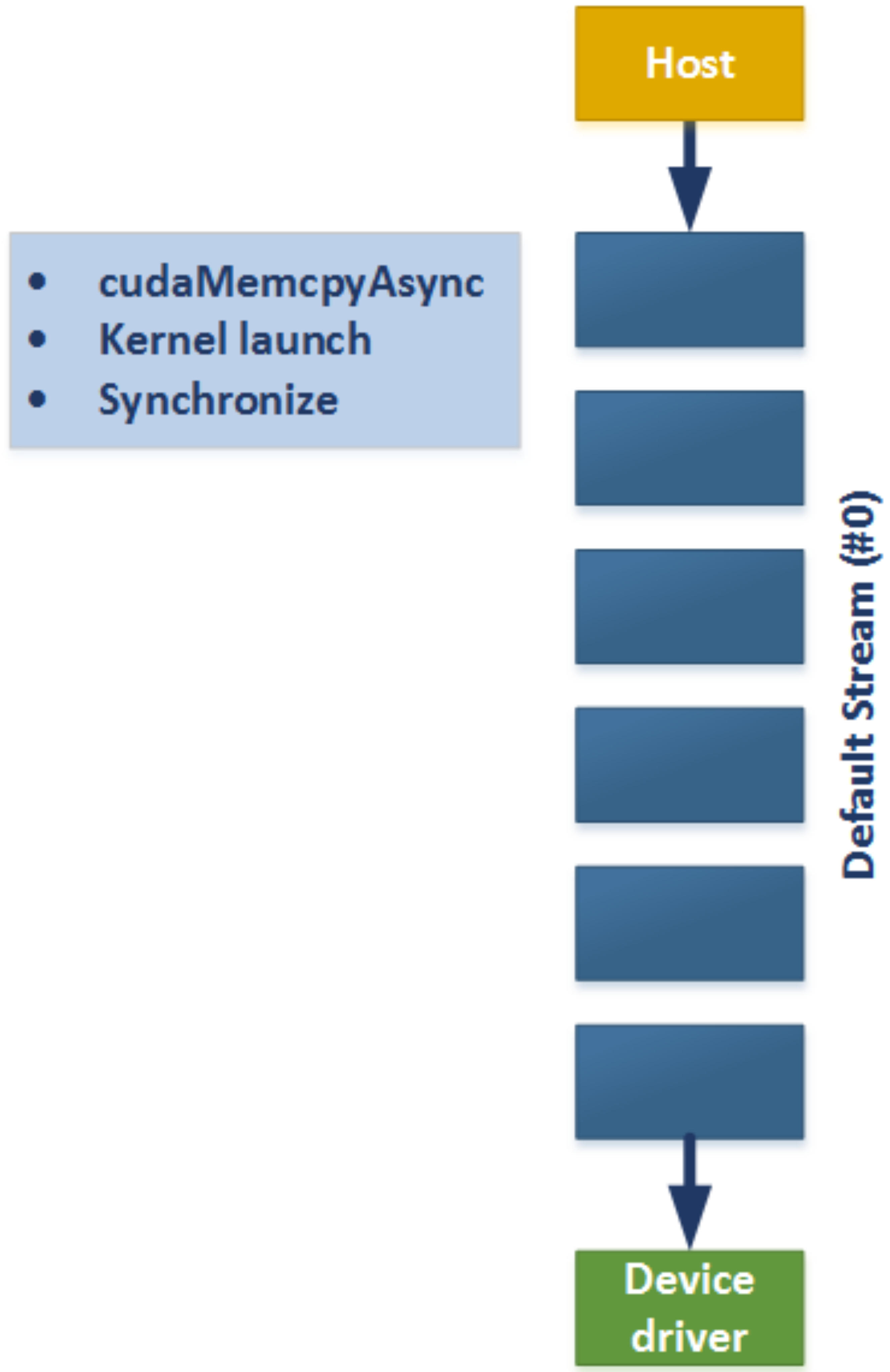
- Asynchronous cudaMemcpyAsync only

A CUDA stream is an **ordered queue** of operations

- Kernel launches, cudaMemcpyAsync, synchronizations

Ordering within a queue is maintained
-> resolving dependencies

- Independent operations should go into different streams
- API calls are asynchronous
- Return after queued, but not necessarily completed



MULTIPLE CUDA STREAMS

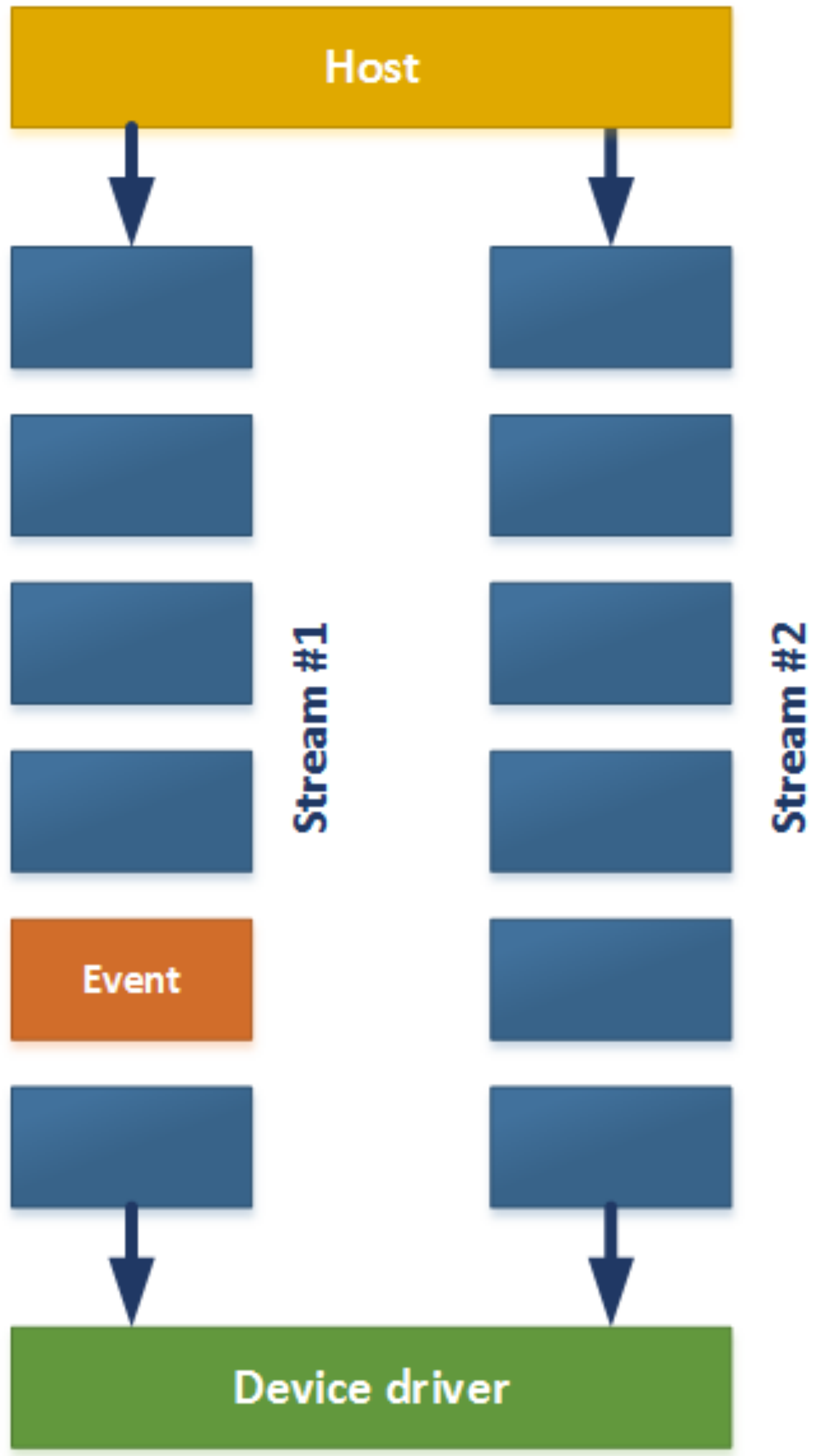
Use different streams to allow for overlap regarding copy and execute -> multiple queues

Host needs to query and synchronize on operations in the queue -> events

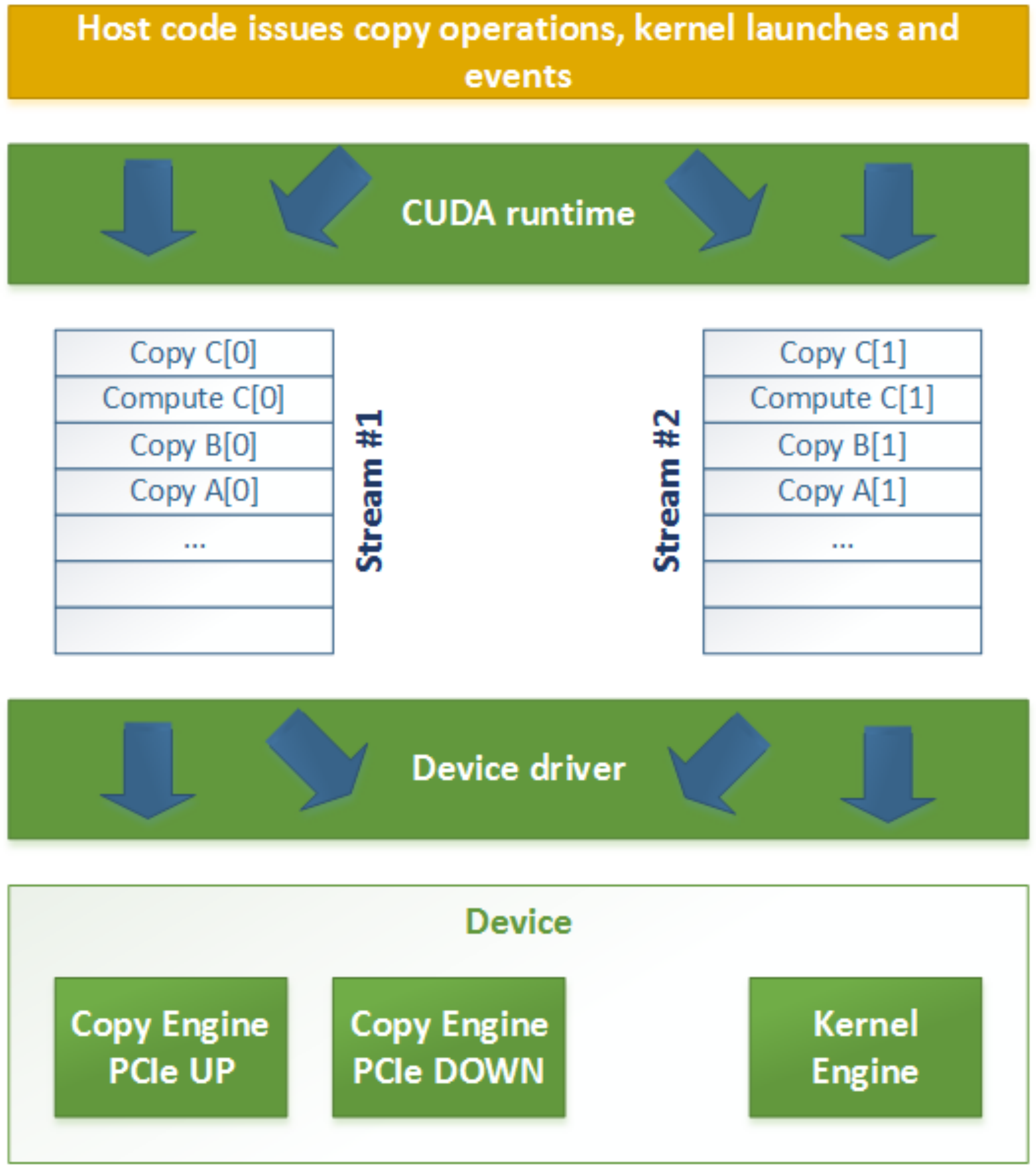
> If CUDA events pop out of the queue, previous operations have completed (FIFO)

Extend kernel launch call by stream ID

```
kernel_function <<< dim3 grid_dim,
                    dim3 block_dim,
                    size_t shmem_size,
                    cudaStream_t stream >>>

cudaMemcpyAsync(void* dst, const void* src,
                size_t count, cudaMemcpyKind kind,
                cudaStream_t stream = 0)
```



MULTIPLE CUDA STREAMS – CONCEPTUAL VIEW



MULTI-STREAM HOST CODE – VERSION 1

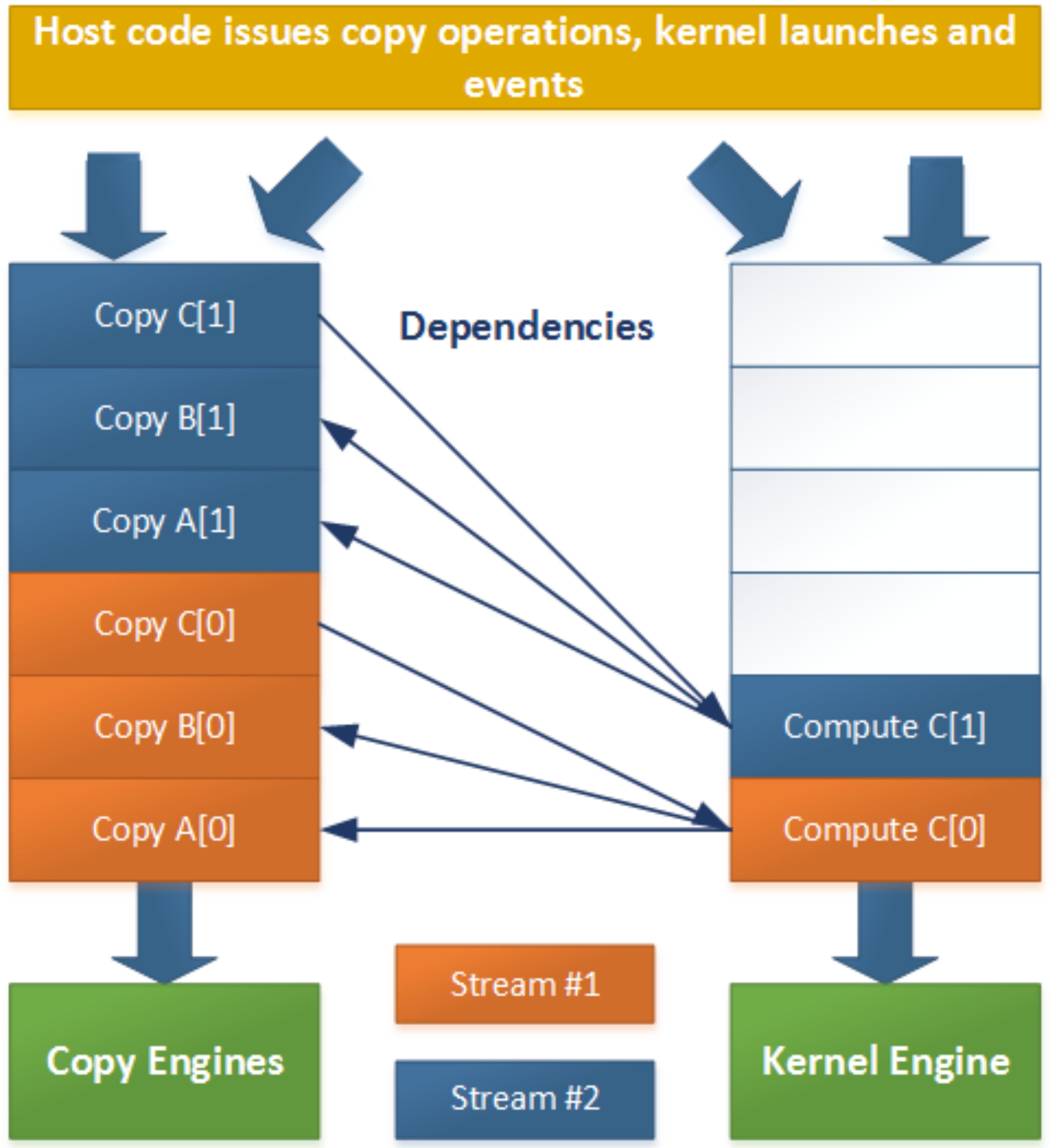
```
cudaStream_t stream0, stream1;
cudaStreamCreate ( &stream0 );
cudaStreamCreate ( &stream1 );
float *d_A0, *d_B0, *d_C0;  // device memory for stream 0
float *d_A1, *d_B1, *d_C1;  // device memory for stream 1

<snip> // cudaMallocs go here

for ( int i = 0; i < n; i += segSize * 2 ) {
  // stream 0
  cudaMemCpyAsync ( d_A0, h_A + i, segSize*sizeof(float),.. , stream0 );
  cudaMemCpyAsync ( d_B0, h_B + i, segSize*sizeof(float),.. , stream0 );
  saxpy <<< segSize/256, 256, 0, stream0 >>> ( d_A0, d_B0, ... );
  cudaMemCpyAsync ( d_C0, h_C + i, segSize*sizeof(float),... , stream0 );

  // stream 1
  cudaMemCpyAsync ( d_A1, h_A + i + segSize, segSize*sizeof(float), ..., stream1 );
  cudaMemCpyAsync ( d_B1, h_B + i + segSize, segSize*sizeof(float), ..., stream1 );
  saxpy <<< segSize/256, 256, 0, stream1 >>> ( d_A1, d_B1, ... );
  cudaMemCpyAsync ( d_C1, h_C + i + segSize, segSize*sizeof(float), ..., stream1 );
}
```

Loop variable increases by segSize * no_of_streams

Loop unrolling, however here into different queues

ISSUES USING STREAMS

MULTI-STREAM HOST CODE – VERSION 1



Goal: overlap A[1] & B[1] with Compute C[0]

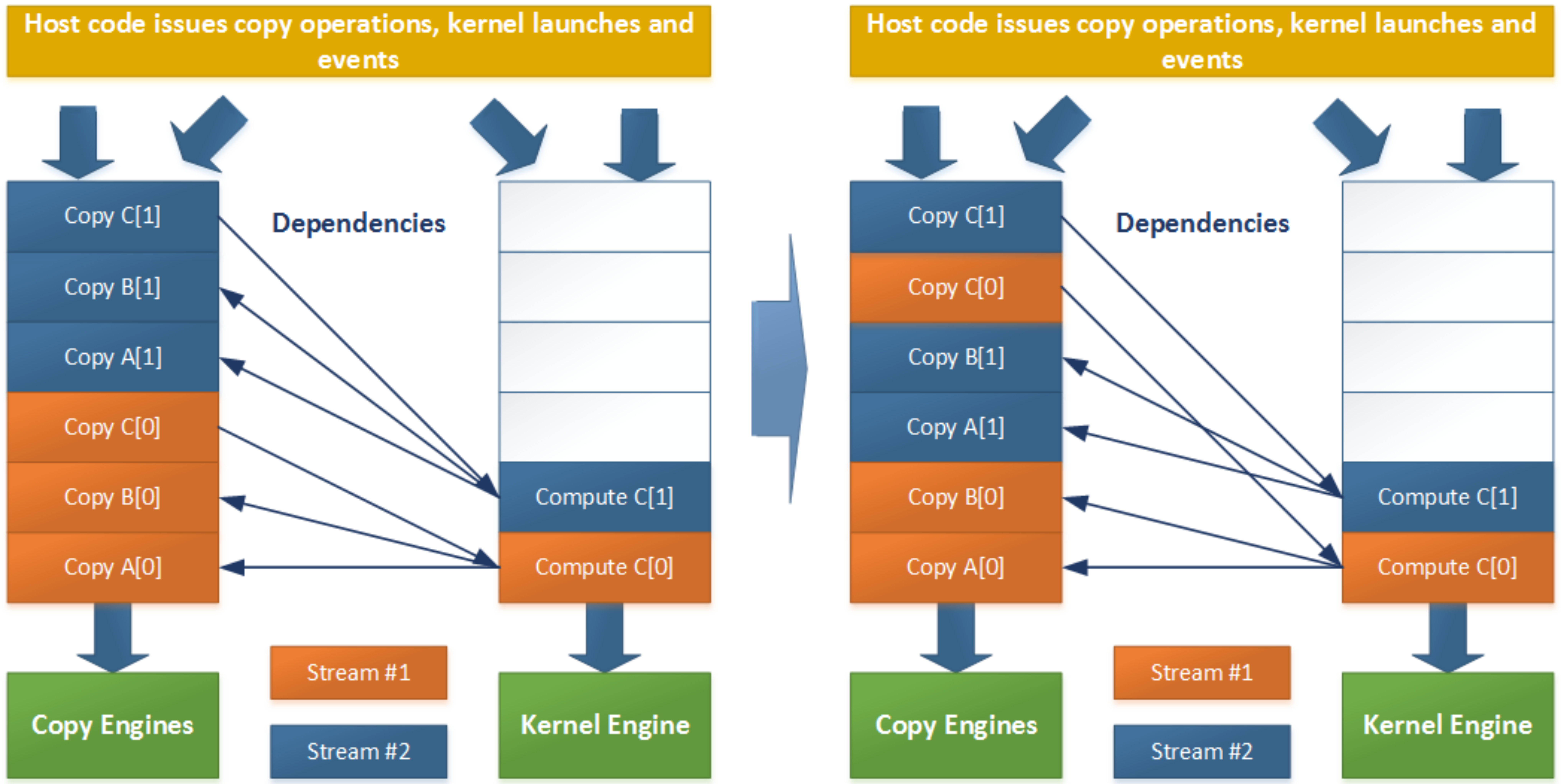
Data structure in device driver to maintain dependencies

- Single device-level queues for copy and execute, respectively
- (Fermi or older)

That’s not what we want

- Copy C[0] blocks A[1] and B[1] in the copy engine queue

MULTI-STREAM HOST CODE – VERSION 2

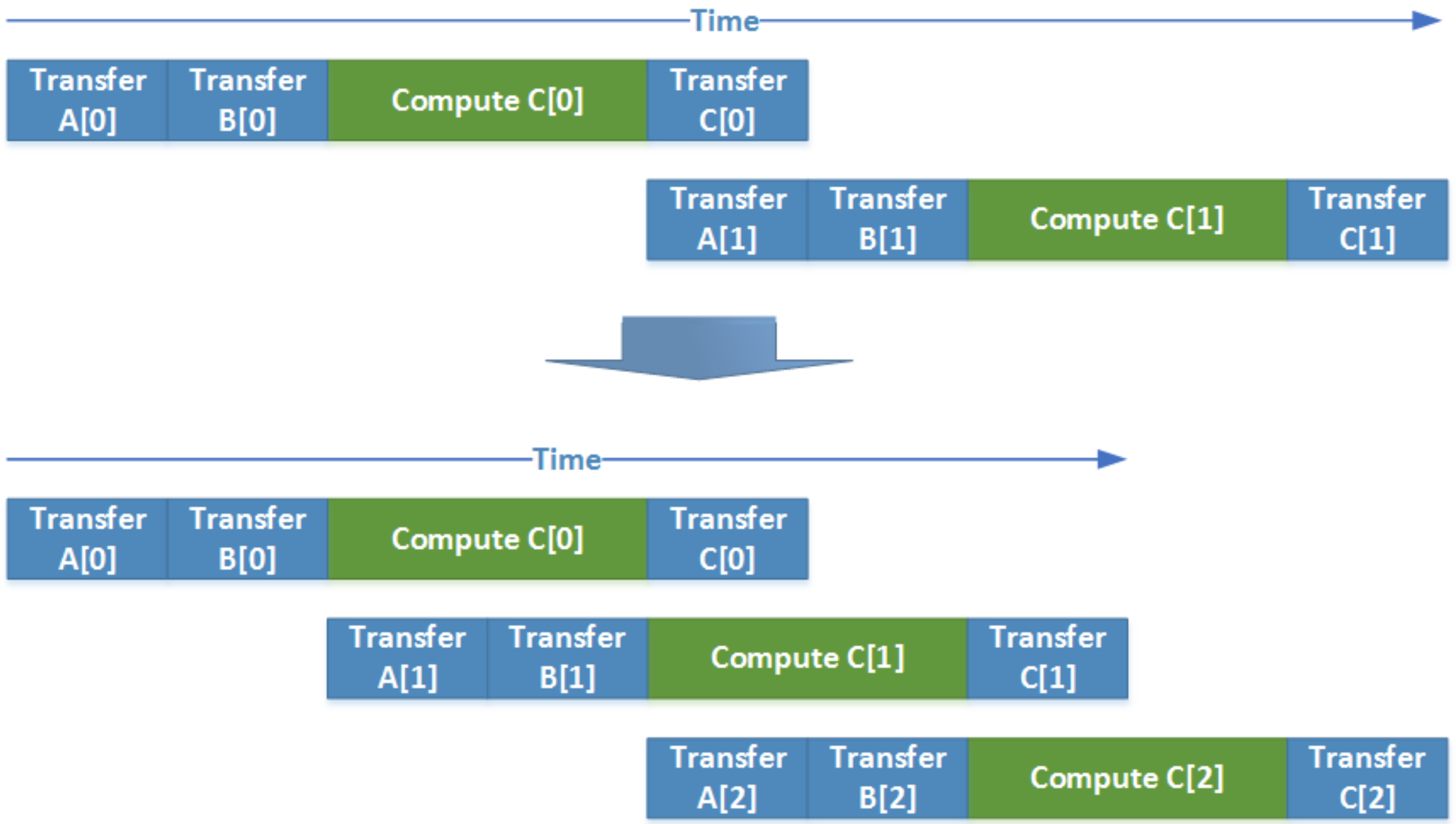


MULTI-STREAM HOST CODE – VERSION 2

```
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

<snip> // cudaMallocs go here

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 );

  saxpy <<< segSize/256, 256, 0, stream0 >>> ( d_A0, d_B0, ... );
  saxpy <<< 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 );
}
```

By re-ordering enqueue operations, we don't get blocked -> task parallelism

MULTI-STREAM HOST CODE – VERSION 2



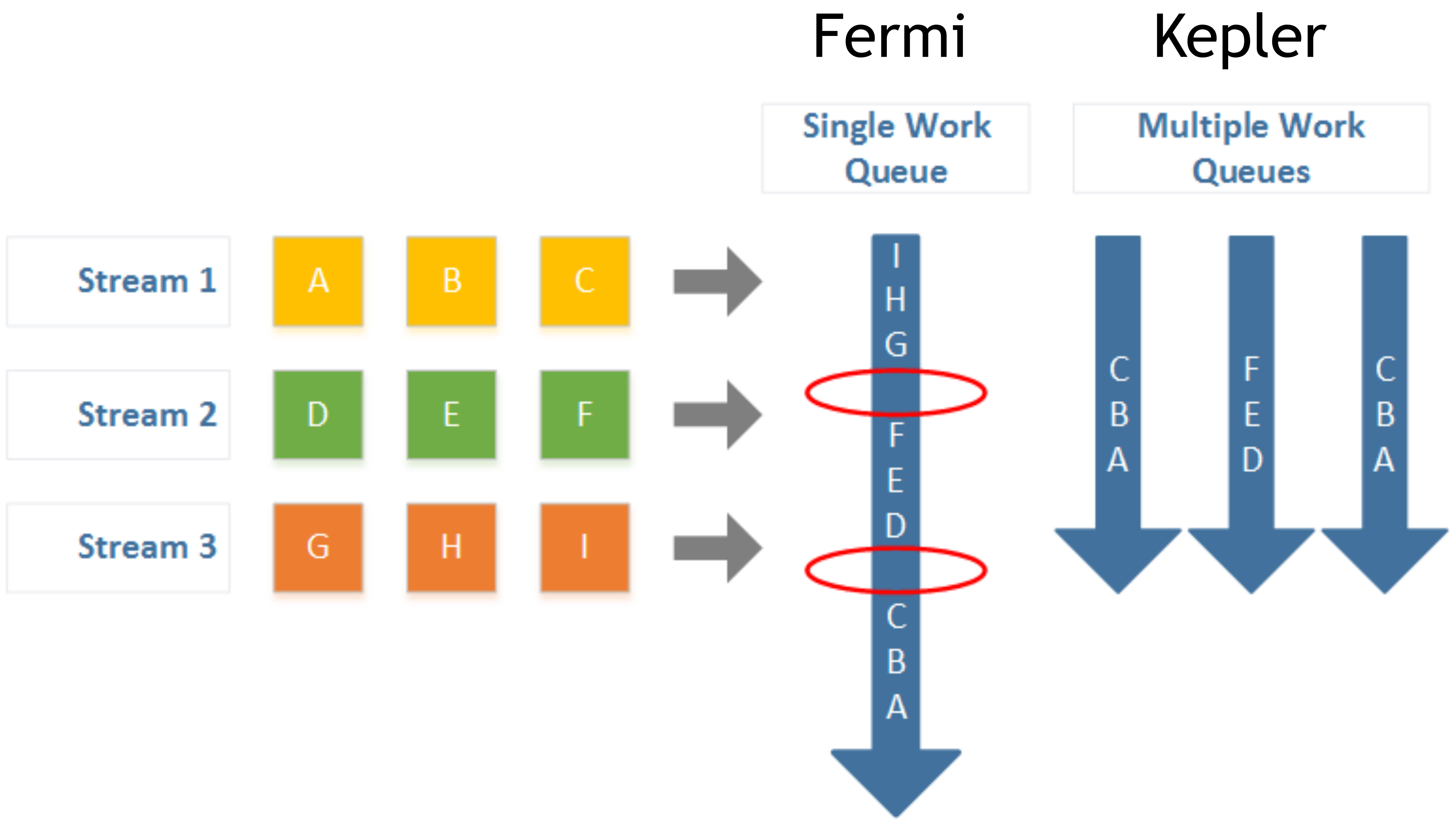
CUDA STREAMS: FERMI VS. KEPLER (OR NEWER)

Fermi: one queue for each engine (copy and execute)

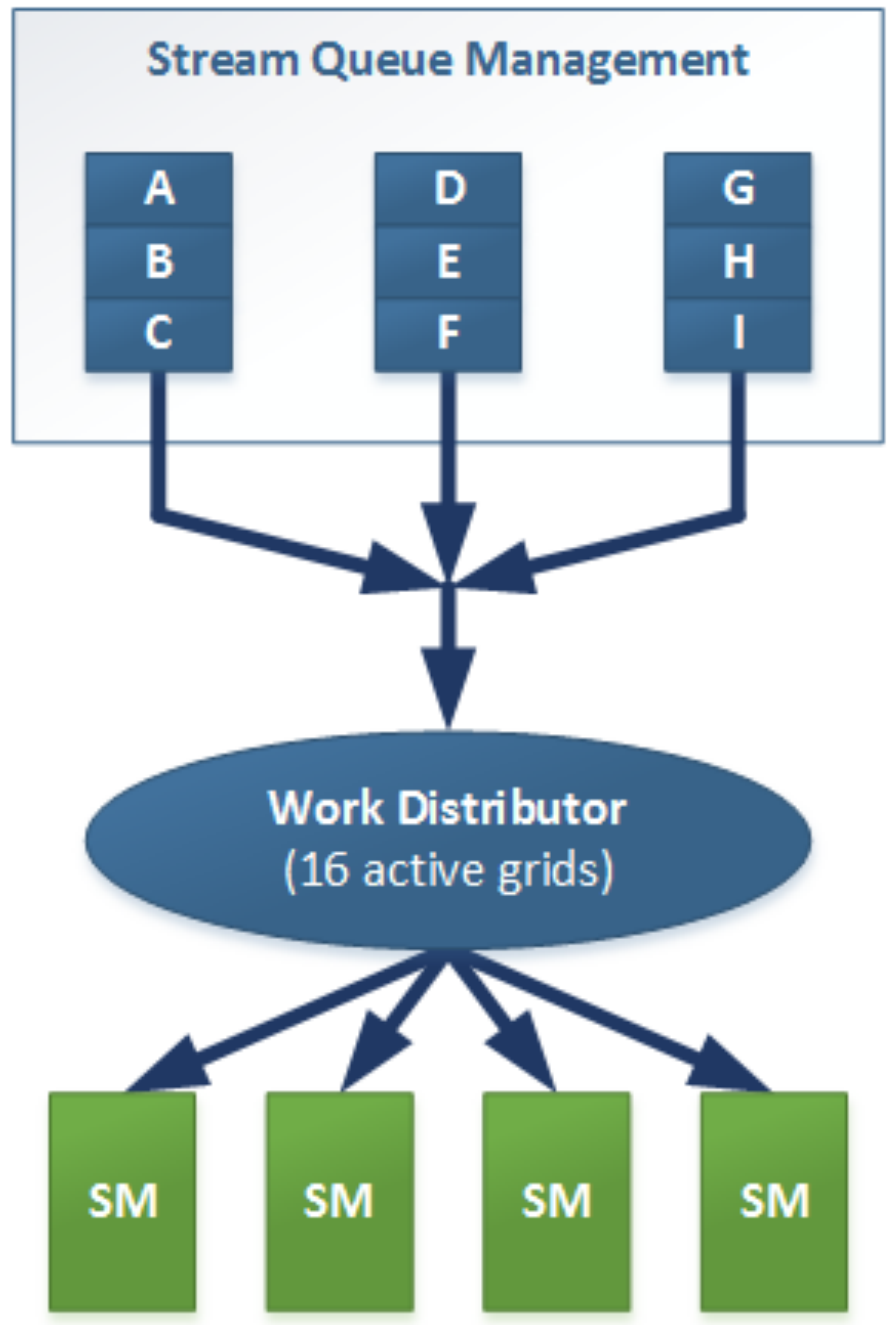
- Careful unrolling required
- See example
- -> Single work queue

Kepler: multiple queues for each engine (copy and execute, 32 queues each)

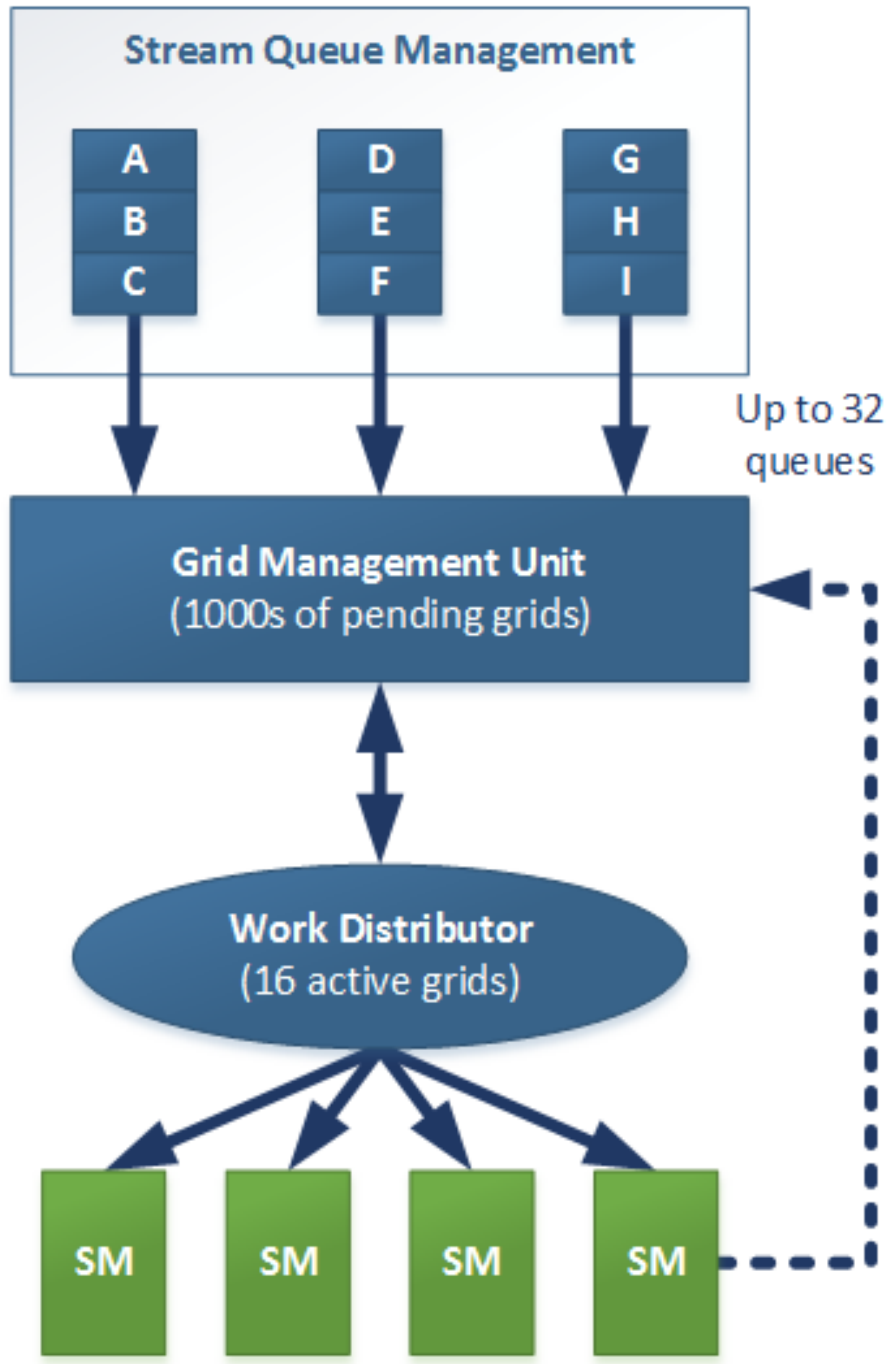
- "Hyper-Queuing"
- Prioritized scheduling between ready streams
- Explicit re-ordering unnecessary



GRID MANAGEMENT



Fermi



Kepler

CUDA STREAMS: FALLACIES

Some operations implicitly synchronize all other CUDA operations

Page locked memory allocation

`cudaMallocHost()` or `cudaHostAlloc()`

Device memory allocation

`cudaMalloc()`

Non-Async versions of memory operations

`cudaMemcpy*()` (no Async suffix)

`cudaMemset*()` (no Async suffix)

Change to L1/shared memory configuration

`cudaDeviceSetCacheConfig()`

REMINDER: LATENCY TOLERANCE TECHNIQUES

streaming

Property	Relaxed Consistency Models	Prefetching	Multi-Threading	Block Data Transfer
Types of latency tolerated	Write (blocking read processors) Read and write (dynamically scheduled processors)	Write Read	Write Read Synchronization	Write Read
Software requirements	Labeling synchronization operations	Predictability	Explicit extra concurrency	Identifying and orchestrating block transfers
Extra hardware support	Little	Little	Substantial	Not in processor, but in memory system
Supported in commercial systems?	Yes	Yes	Yes	(Yes)

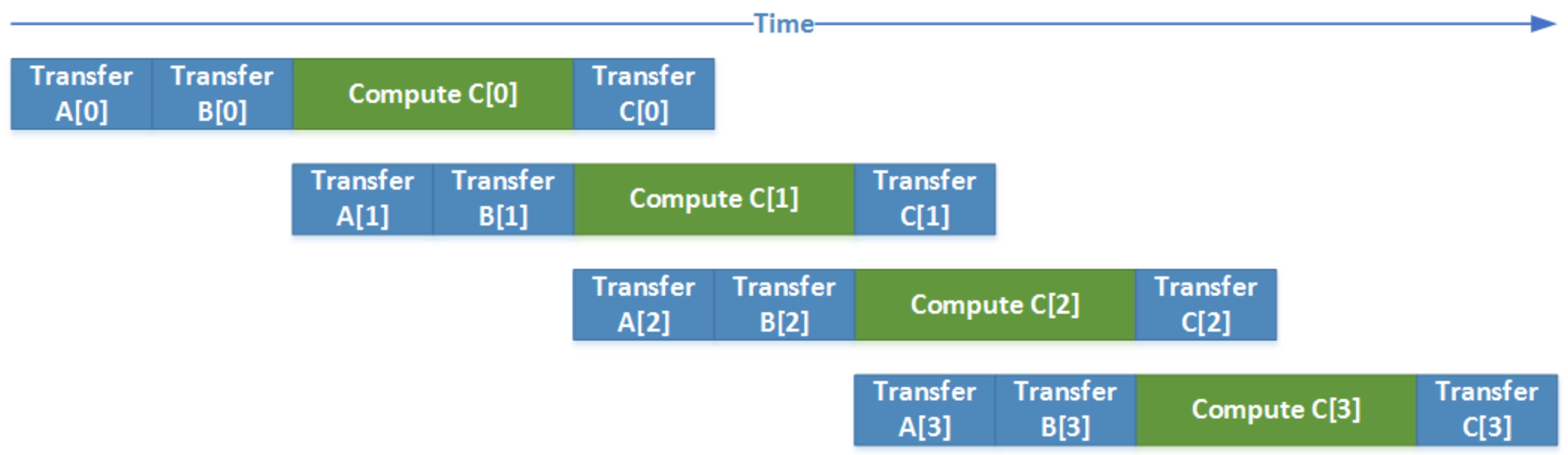
David E. Culler, Jaswinder Pal Singh, Anoop Gupta, Parallel Computer Architecture: A Hardware/Software Approach, Morgan Kaufmann, 1998

APPLICABILITY OF STREAMING

Technique to hide PCIe latency

$$T_{compute} \geq T_{PCI}$$

Assume segment size = N floats



Machine behaviour

$B : Bandwidth(GB/s)$

$C : PeakPerformance(GFlops/s)$

Program behaviour

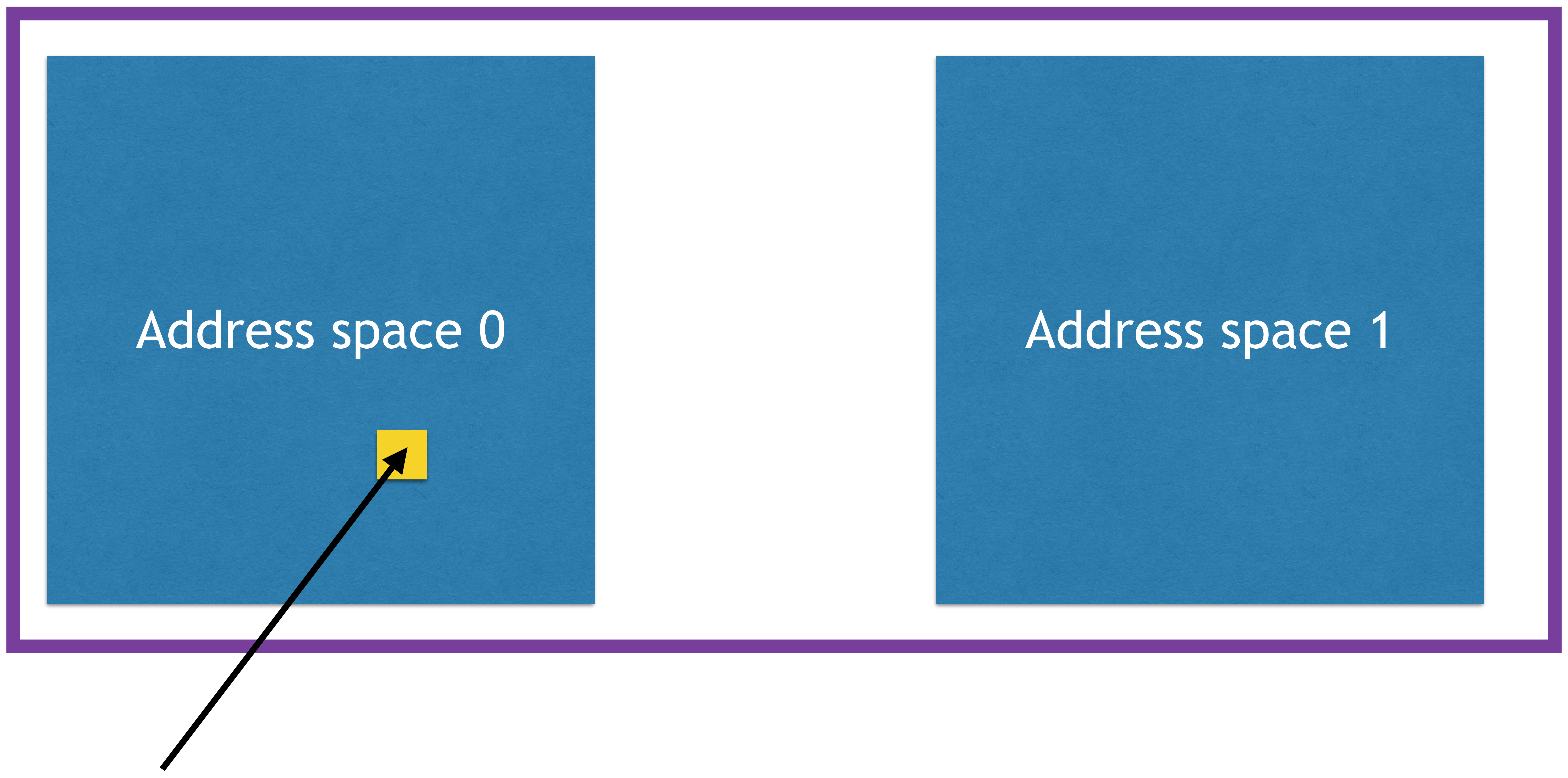
$$ComputationalIntensity : r = \frac{Flops}{Byte}$$

$$T_{PCI} = \frac{4 \times N}{B}, T_{compute} = \frac{r \times 4 \times N}{C}$$

$$\frac{r \times 4 \times N}{C} \geq \frac{4 \times N}{B} \Rightarrow r \geq \frac{C}{B}$$

VIRTUAL SHARED MEMORY

VIRTUAL SHARED MEMORY CONCEPT



0xb62ae…

UVA/UVM

Unified Virtual Addressing (UVA)

Single virtual address space for all memory in the system

GPU code can access all memory

Manual locality optimizations (cudaMemcpy)

```
cudaDeviceCanAccessPeer(&result, gpuid_0,
                                 gpuid_1);
cudaSetDevice (gpuid_0);
cudaDeviceEnablePeerAccess (gpuid_1, 0);
cudaSetDevice (gpuid_1);
cudaDeviceEnablePeerAccess (gpuid_0, 0);
cudaMemcpy ( gpu0_buf, gpu1_buf, buf_size,
             cudaMemcpyDefault);

// or: operate directly on remote memory
 gpu0_buf[idx] = gpu1_buf[idx];
```

Unified Memory (UM)

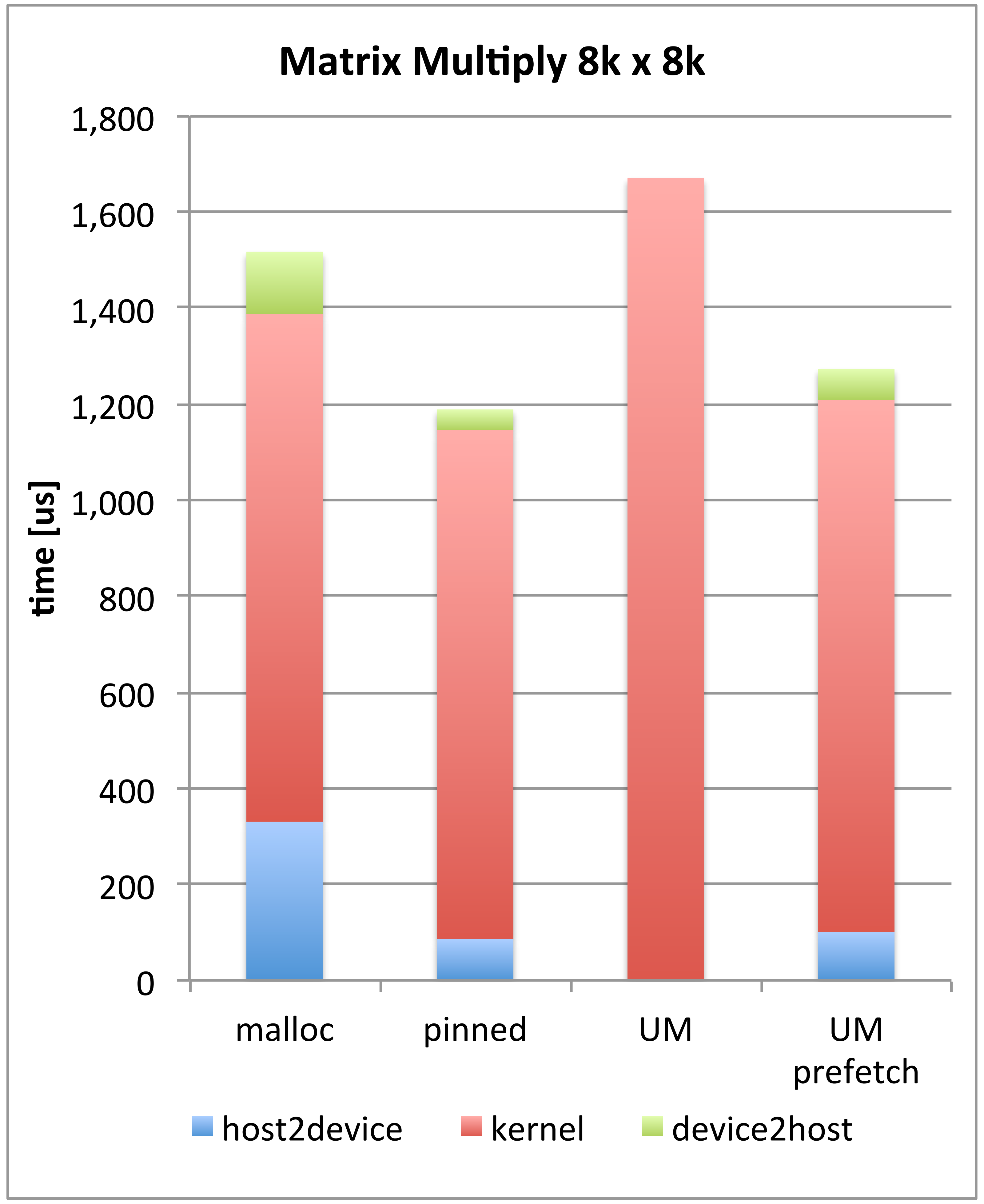
Pool of managed memory that is shared between CPU and GPU

Single pointer sufficient

Automatic (page) migration between CPU & GPU domains (Pascal/Volta)

```
float *X, ...;  // unified pointers
cudaMallocManaged (X, N * sizeof (float));

...
saxpy <<<numBlocks,blockSize>>> (X, ...);
...

cudaDeviceSynchronize ();
use_data ( X );
cudaFree ( X );
```

PASCAL - UNIFIED MEMORY



WRAPPING UP

SUMMARY

More parallelism: streams

- New: task-level parallelism
- + Hide data movement costs (PCIe)
- - More work for the programmer

Careful work dispatch required for older GPUs

Overlap for latency hiding requires certain computational intensity

Alternative #1: Unified virtual addressing (UVA)

- Threads can access CPU or other GPUs' memory
- Access costs can be huge

Alternative #2: Unified memory (UM)

- Automated data movement based on page migration
- Certainly some overhead, but overall concept promising