CS302

GPU: Streams, UVA

Spring 2025

Babak Falsafi, Arkaprava Basu

parsa.epfl.ch/course-info/cs302

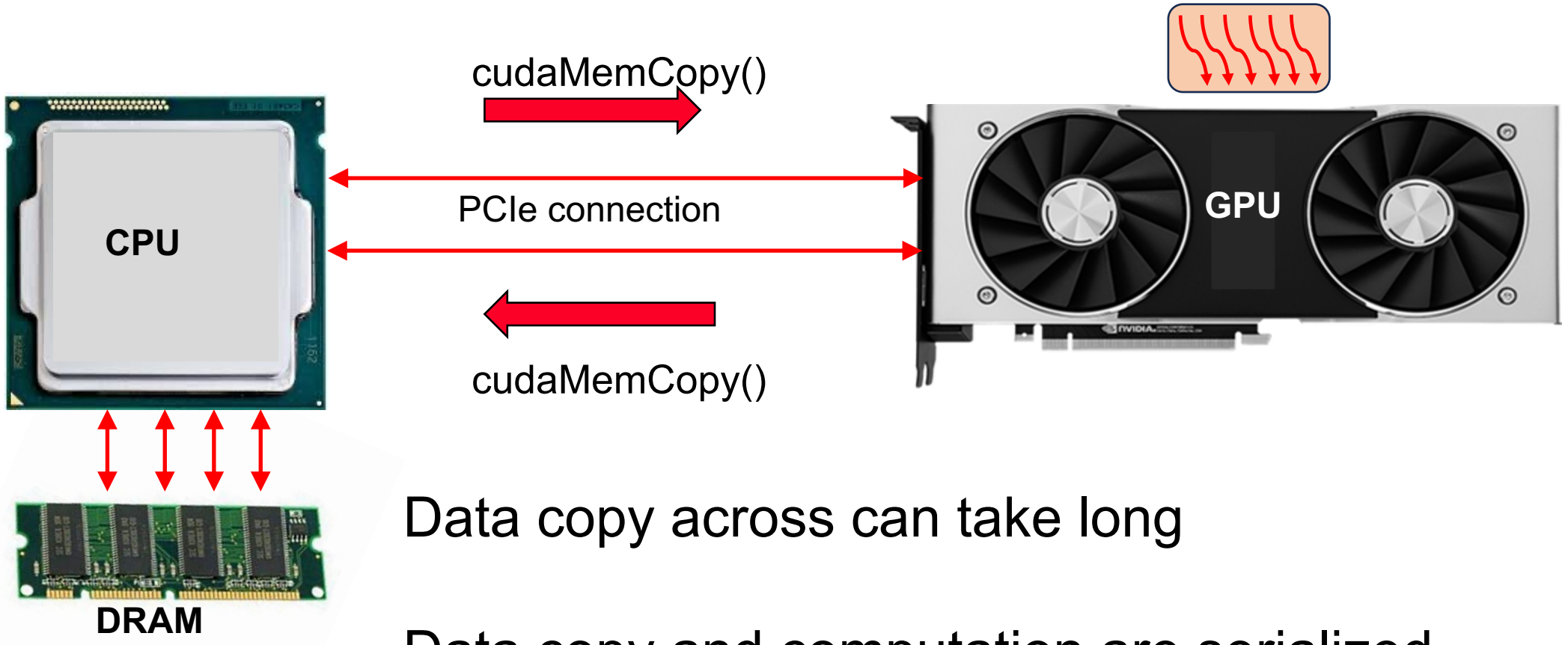
Some of the slides are from Derek R Hower, Adwait Jog, Wen-Mei Hwu, Steve Lumetta, Babak Falsafi, Andreas Moshovos, and from the companion material of the book “Programming Massively Parallel Processors”

Where are We?

M	T	W	T	F
17.Feb	18.Feb	19.Feb	20.Feb	21.Feb
24.Feb	25.Feb	26.Feb	27.Feb	28.Feb
03.Mar	04.Mar	05.Mar	06.Mar	07.Mar
10.Mar	11.Mar	12.Mar	13.Mar	14.Mar
17.Mar	18.Mar	19.Mar	20.Mar	21.Mar
24.Mar	25.Mar	26.Mar	27.Mar	28.Mar
31.Mar	01.Apr	02.Apr	03.Apr	04.Apr
07.Apr	08.Apr	09.Apr	10.Apr	11.Apr
14.Apr	15.Apr	16.Apr	17.Apr	18.Apr
21.Apr	22.Apr	23.Apr	24.Apr	25.Apr
28.Apr	29.Apr	30.Apr	01.May	02.May
05.May	06.May	07.May	08.May	09.May
12.May	13.May	14.May	15.May	16.May
19.May	20.May		22.May	23.May
26.May	27.May	28.May	29.May	30.May

- This lecture
  - Streams
  - Memory oversubscription
- Exercise session
  - Practice sample questions
- Next class
  - Scaling trends
  - 29th May is a holiday!

GPU Is a Co-Processor: Needs a Companion CPU

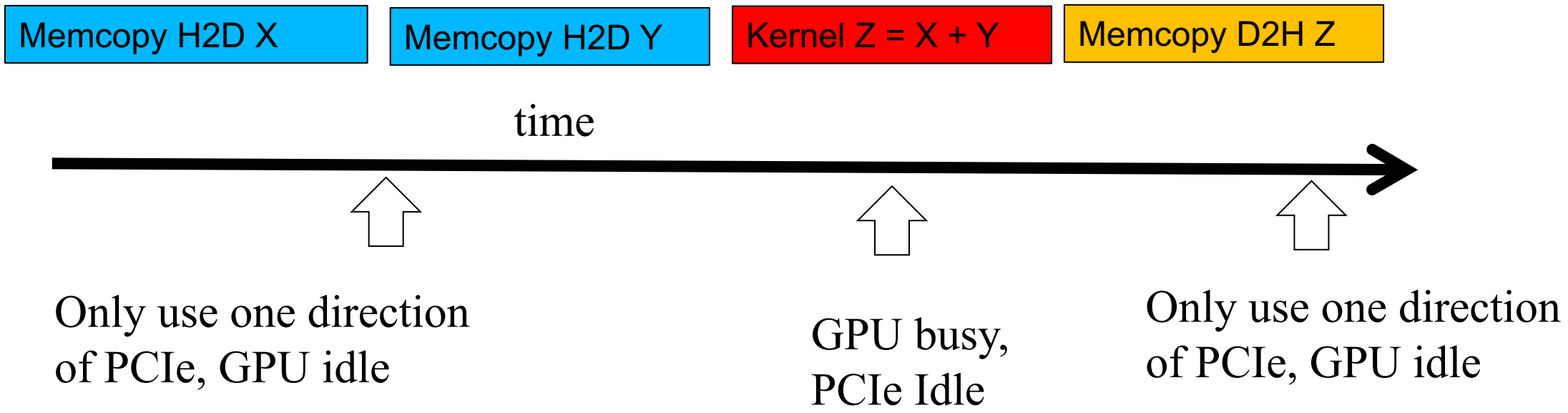


Data copy across can take long

Data copy and computation are serialized

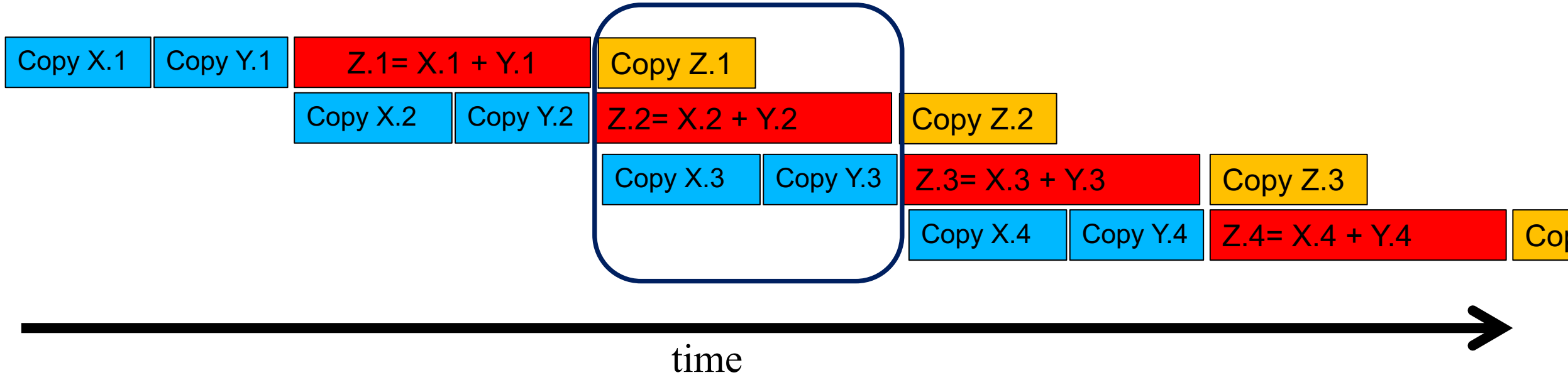
Can we overlap data copy with kernel execution?

Serialization in VectorAdd Example



- Data copy and GPU computation are serialized

Pipelining to Overlap Data Copy with Compute



- Partition the vectors in segments
- Pipeline copy and compute of segments
- Overlap data copy with compute on GPU

CUDA Features for Overlapping Data Transfer and Compute

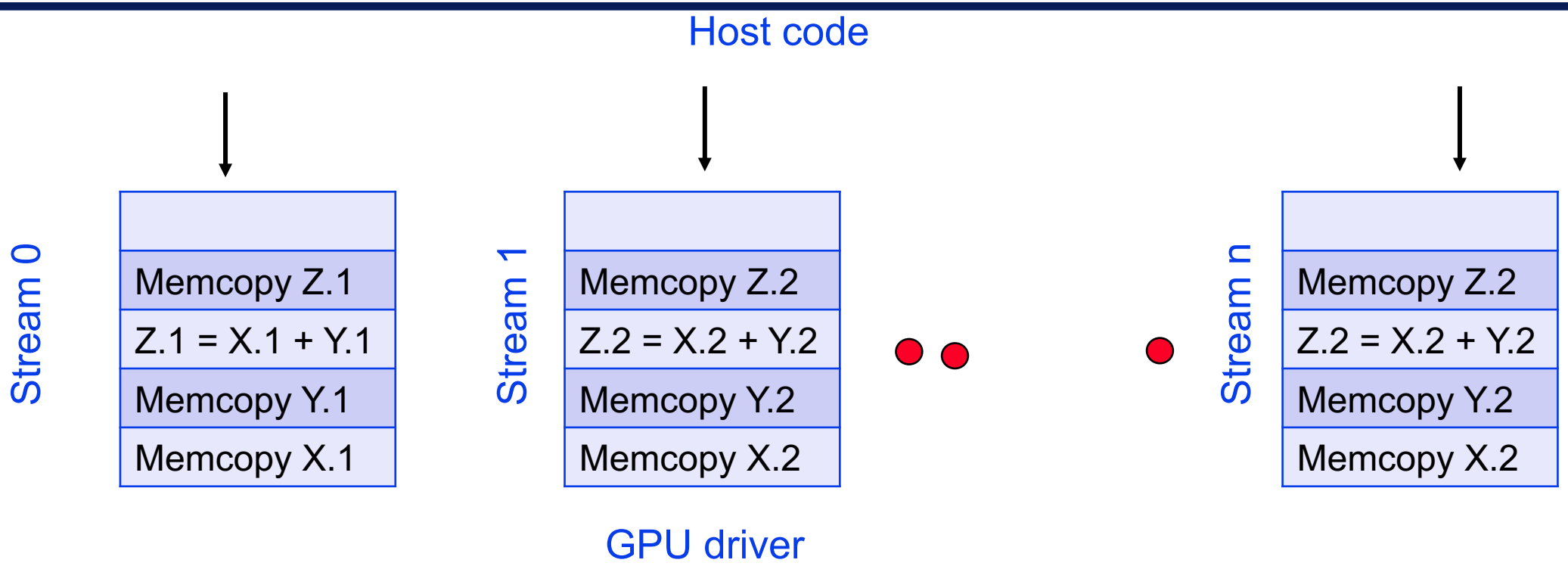
- Streams: Workqueues to queue independent GPU-related tasks
  - Need separate queues for independent tasks

- CUDAmemcopyAsync: Asynchronous data transfer over PCIe
  - Allows host to proceed without waiting for copy to finish (non-blocking)

What are Streams?

- A stream is a queue for GPU related tasks (e.g., kernel launch, memcopy)
- Host (CPU) queue tasks on a stream
- Tasks in the same stream execute sequentially
- Tasks in different streams can execute concurrently

Use Multiple Streams for Overlapping of Copy and Compute



- GPU related requests are placed in streams (FIFO) by host code
- Queue entries processed asynchronously by GPU driver
- No ordering between entries in different streams

Declaring and Creating Streams

- Creating a stream
  cudaError_t cudaStreamCreate (cudaStream_t * pStream)
  pstream → pointer to new stream identifier

- Launching kernels in a specific stream

  kernel_name <<<grid, threadblock, smem, stream id>>>

Streams to Overlap Compute and Data copy in VectorAdd

```
void vecadd(float* x, float* y, float* z, int N) {

    float *x_d, *y_d, *z_d;
    cudaMalloc((void**) &x_d, N*sizeof(float));
    cudaMalloc((void**) &y_d, N*sizeof(float));   // Allocate GPU memory –No change
    cudaMalloc((void**) &z_d, N*sizeof(float));

    cudaStream_t streams[NUM_STREAM];  //Declare multiple streams

    for (unsigned int n; n < NUM_STREAM; n++) // Create multiple streams
       cudaCreateStream(&streams[n]);

   //To continue to the next slide
```

Streams to Overlap Compute and Data copy in VectorAdd

```
unsigned int segmentSize = (N + NUM_STREAM - 1)/ NUM_STREAM; //Segment of data
 for (unsigned int n = 0; n < NUM_STREAM; n++) {
    unsigned int start = n*segmentSize;
    unsigned int end = (start + segmentSize < N)?: (start + segmentSize) : N;
    unsigned int Nsegment = end - start;

    cudaMemcpy(&x_d[start], &x[start], Nsegment*sizeof(float),
               cudaMemcpyHostToDevice, stream[s]); //Memcpy to different streams
    cudaMemcpy(&y_d[start], &y[start], Nsegment*sizeof(float),
               cudaMemcpyHostToDevice, stream[s]);

    numThreadsPerBlock = 512;
    numBlocks = (N + numThreadsPerBlock - 1)/numThreadsPerBlock;
    vecadd_kernel <<< numBlocks, numThreadsPerBlock, 0, stream[s] >>>
    (&x_d[start], &y_d[start], &z_d[start], Nsegment);  //Kernels to streams

    cudaMemcpy(&z[start], &z_d[start], Nsegment*sizeof(float),
          cudaMemcpyDeviceToHost, stream[s]);
 } //End of loop
```

Will block until the computation is complete!

cudaMemcpyAsync: Need Asynchronous Data Copy

- cudaMemcpy() is a blocking
- Host code cannot proceed until the data copy is done
- Host code will not enqueue tasks to streams until the previous segment finishes!

- cudaMemcpyAsync: *Asynchronous* version of cudaMemcpy
  Does not wait for the copy to complete
  Control returns to the host code immediately after queuing the call

```
cudaError_t cudaMemcpyAsync (void * dst, void * src, size_t count,
 cudaMemcpyKind, cudaStream_t);
```

Streams to Overlap Compute and Data copy in VectorAdd

```
unsigned int segmentSize = (N + NUM_STREAM - 1)/ NUM_STREAM; //Segment of data
 for (unsigned int n=0; n < NUM_STREAM; n++) {
    unsigned int start = n*segmentSize;
    unsigned int end = (start + segmentSize < N)?: (start + segmentSize) : N;
    unsigned int Nsegment = end - start;

    cudaMemcpyAsync(&x_d[start], &x[start], Nsegment*sizeof(float),
              cudaMemcpyHostToDevice, stream[s]); //Memcpy to different streams
    cudaMemcpyAsync(&y_d[start], &y[start], Nsegment*sizeof(float),
              cudaMemcpyHostToDevice, stream[s]);

    numThreadsPerBlock = 512;
    numBlocks = (N + numThreadsPerBlock - 1)/numThreadsPerBlock;
    vecadd_kernel <<< numBlocks, numThreadsPerBlock, 0, stream[s] >>>
    (&x_d[start], &y_d[start], &z_d[start], Nsegment);  //Kernels to streams

    cudaMemcpyAsync(&z[start], &z_d[start], Nsegment*sizeof(float),
           cudaMemcpyDeviceToHost, stream[s]);
 } //End of loop
```

Need to Wait for Streams to Finish Work

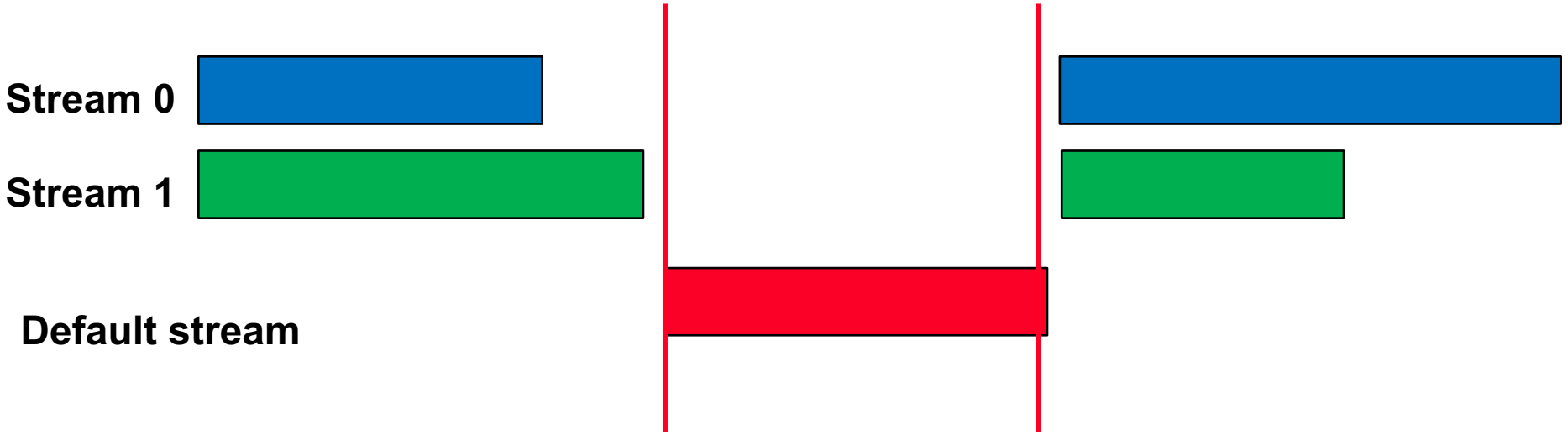
- The host code only queued work on streams
- Ultimately, the host must wait for the entire work to complete

```
    …..
    vecadd_kernel <<< numBlocks, numThreadsPerBlock, 0, stream[s] >>>
    (&x_d[start], &y_d[start], &z_d[start], Nsegment);  //Kernels to streams

    cudaMemcpyAsync(&z[start], &z_d[start], Nsegment*sizeof(float),
          cudaMemcpyDeviceToHost, stream[s]);
} //End of loop

cudaDeviceSynchronize();  //Ensures ALL streams finish their work

//Free the memory using cudaFree();
```

The “Null” (default) Stream

- All GPU-related tasks that do not specify stream (NULL) use the **“default”** stream

- Default (NULL) stream synchronizes all streams
  - Allows no concurrency across other streams

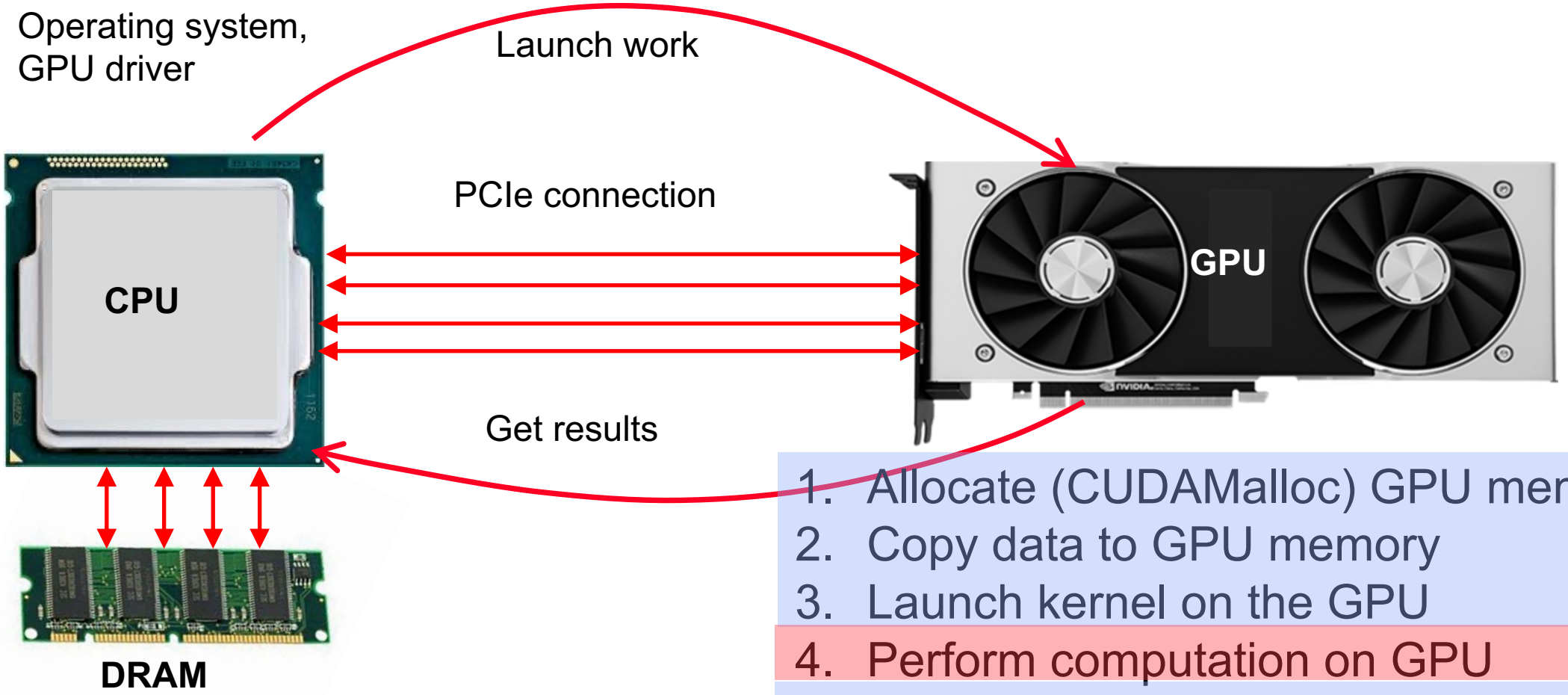


Summary of Streams

- Streams allow concurrency in GPU-related tasks
- Streams are queues of GPU-related tasks
- Tasks in separate queues execute in order
- No ordering amongst tasks in different streams

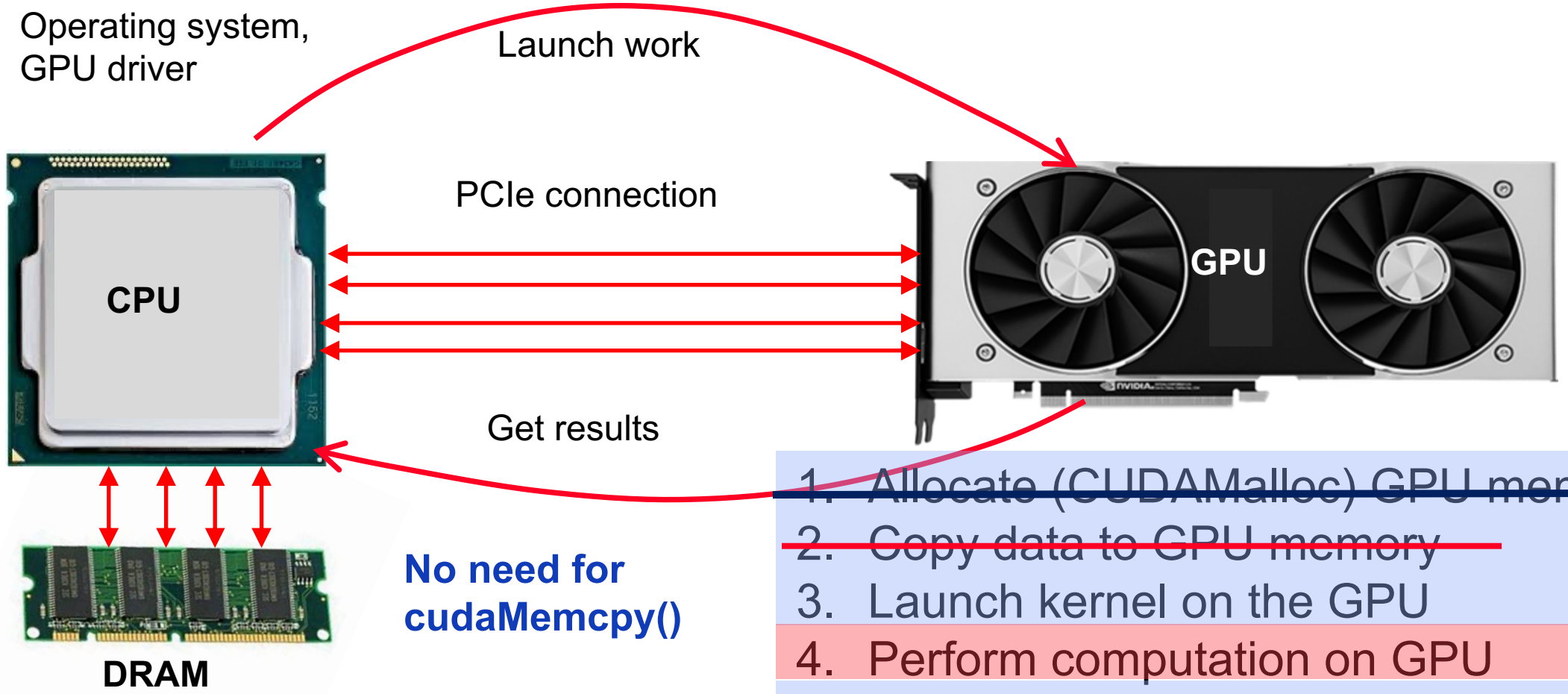
UNIFIED VIRTUAL MEMORY (UVM)

Review: Traditional GPU Memory Allocation and Copy



1. Allocate (CUDAMalloc) GPU memory
2. Copy data to GPU memory
3. Launch kernel on the GPU
4. Perform computation on GPU
5. Copy results from GPU memory
6. Deallocate GPU memory

Unified Virtual Memory (UVM): Simplifying Programming

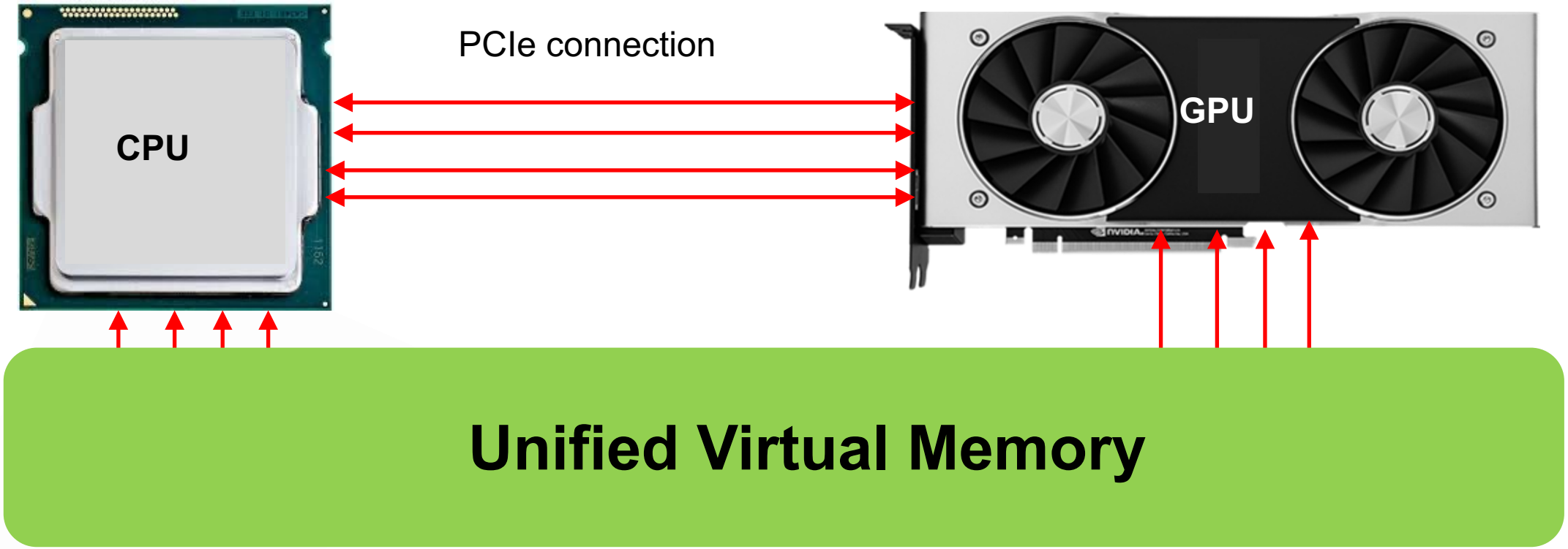


No need for cudaMemcpy()

Joint CPU +GPU allocation

1. ~~Allocate (CUDAMalloc) GPU memory~~
2. ~~Copy data to GPU memory~~
3. Launch kernel on the GPU
4. Perform computation on GPU
5. ~~Copy results from GPU memory~~
6. Deallocate GPU memory

Unified Virtual Memory (UVM): Simplifying Programming



- CUDA runtime and GPU hardware work together to enable UVM

Simple Example of Using UVM's Programming Interface

```
void sortFile(FILE *fp, int N) {
  char *data, *d_data;
  data = malloc (char*) malloc(N);
  cudaMalloc(&d_data, N);
  fread(data, 1, N, fp);
  cudaMemcpy(d_data, data, N,..);
  sortOnGPU <<<…>>> (d_data, N, 1);
  cudaMemcpy(data, d_data, N…);
  useSortedData(data);
  cudaFree(d_data);
  free(data);
}
```

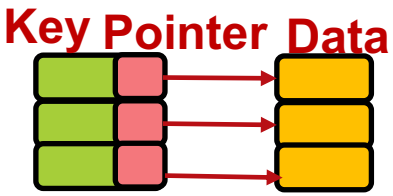
WITHOUT UVM

```
void sortFile(FILE *fp, int N) {
  char *data;
  cudaMallocManaged(&data, N);

  fread(data, 1, N, fp);

  sortOnGPU <<<…>>> (data, N, 1);
  cudaDeviceSynchronize();
  useSortedData(data);
  cudaFree(data);

}
```

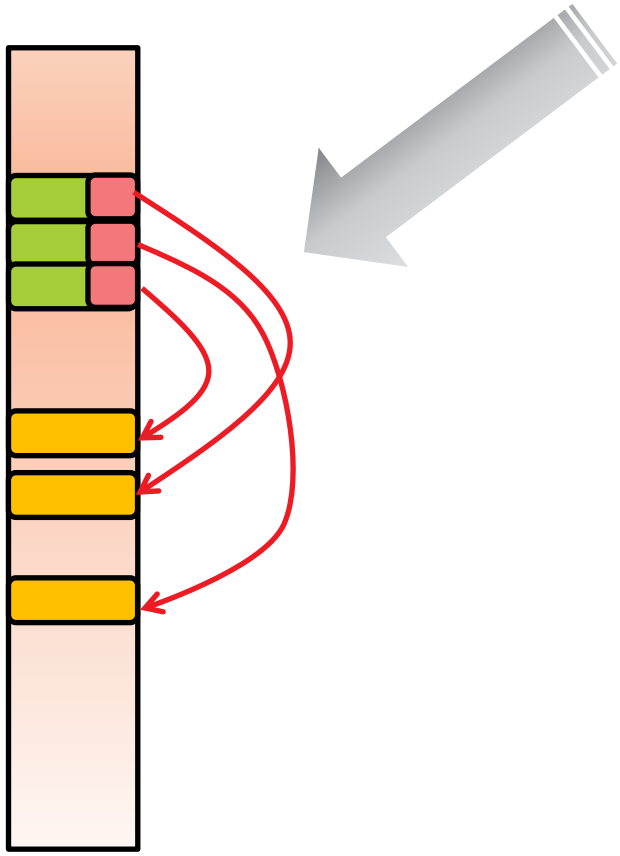
WITH UVM

Programming Pointer-based Data Structures Hard Without UVM

Example: Map data-structure

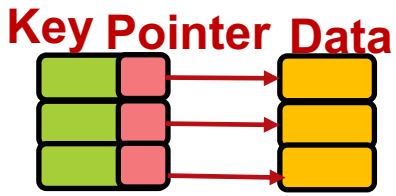


CPU address space

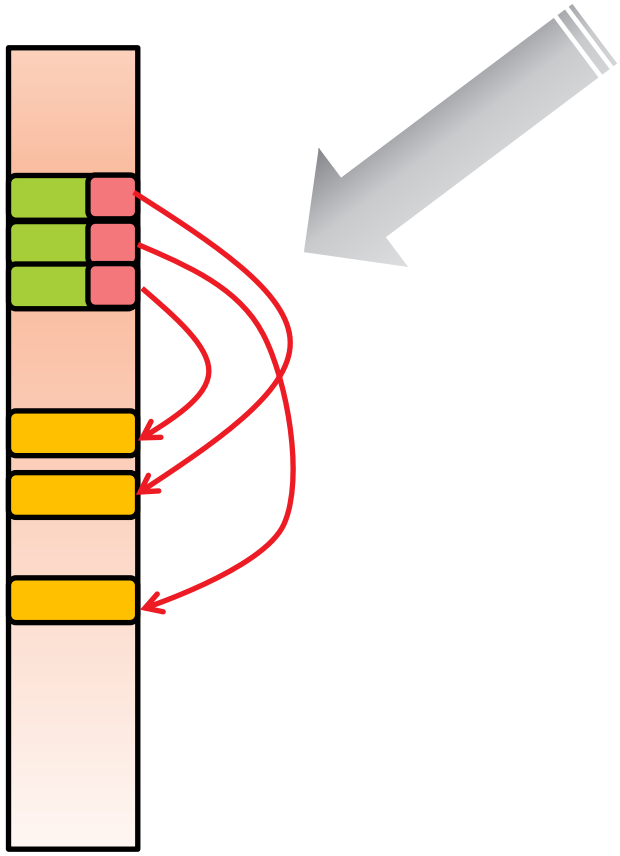


Programming Pointer-based Data Structures Hard Without UVM

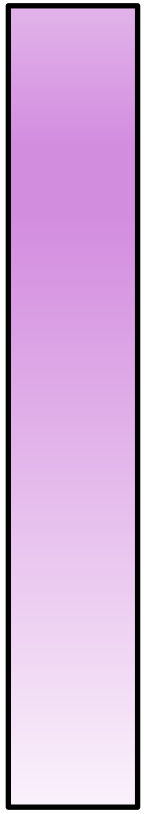
Example: Map data-structure



CPU address space

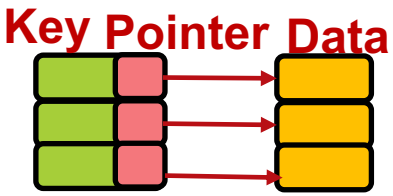


GPU address space



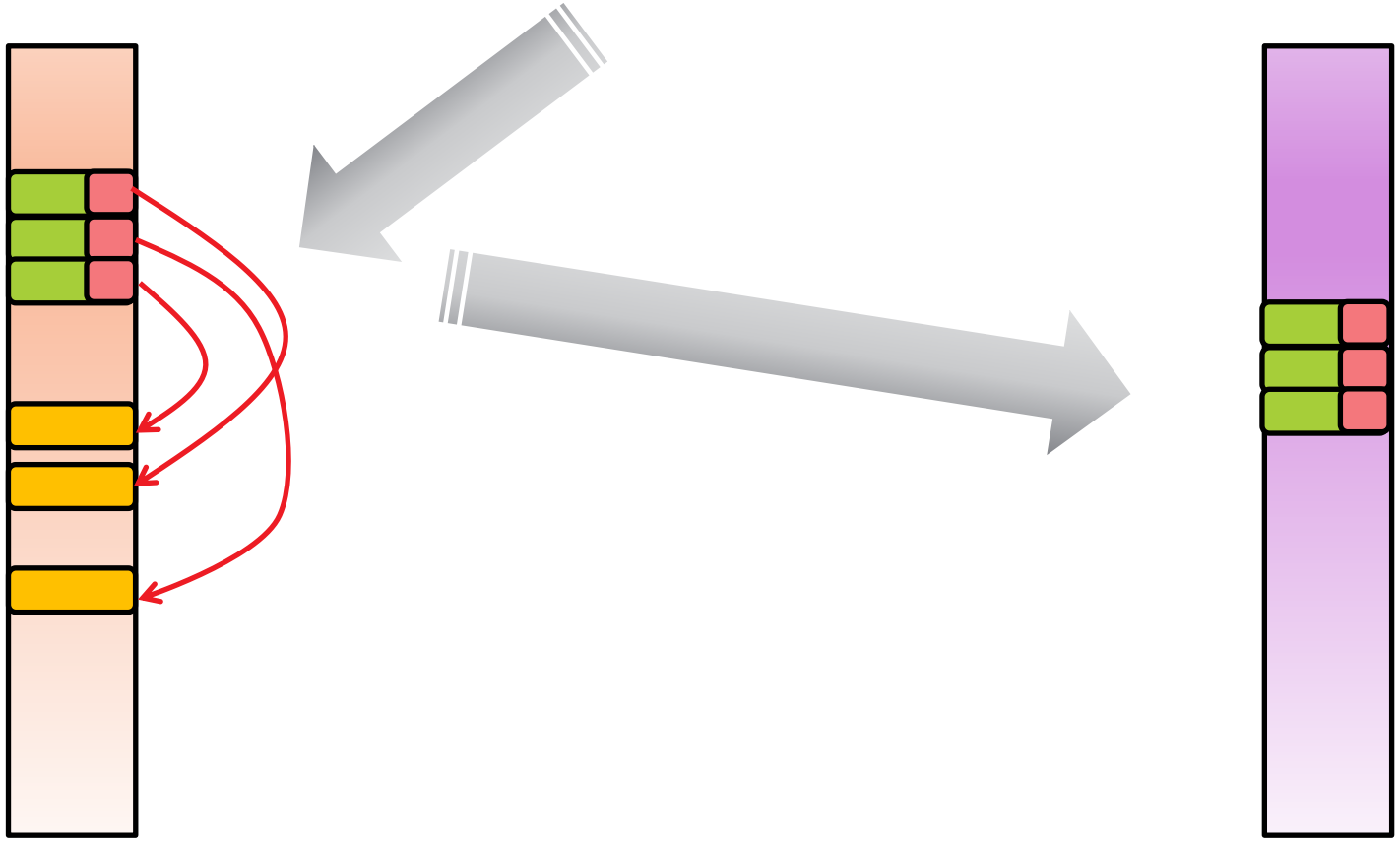
Programming Pointer-based Data Structures Hard Without UVM

Example: Map data-structure



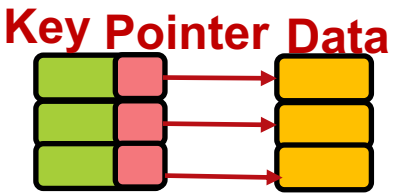
CPU address space

GPU address space



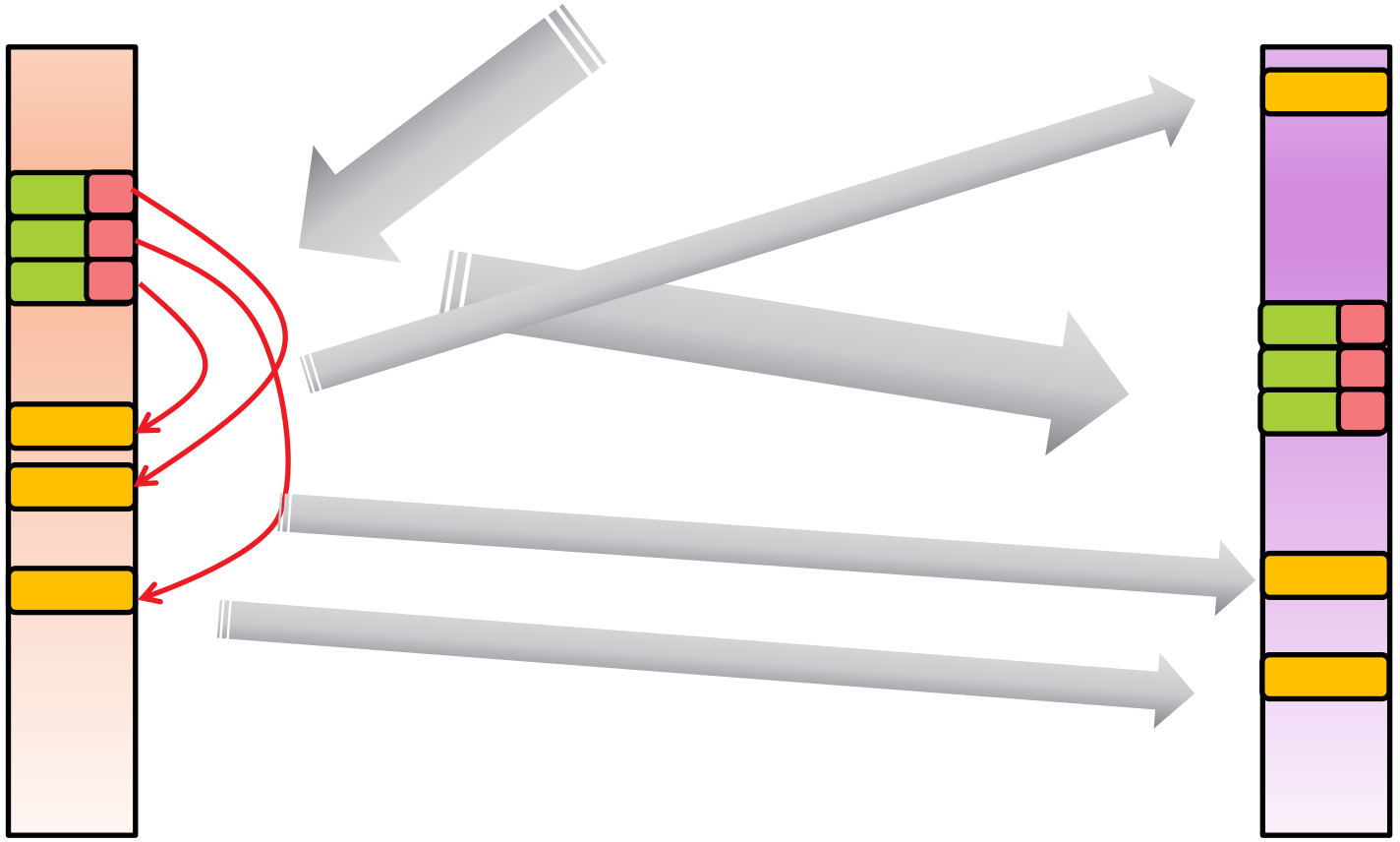
Programming Pointer-based Data Structures Hard Without UVM

Example: Map data-structure



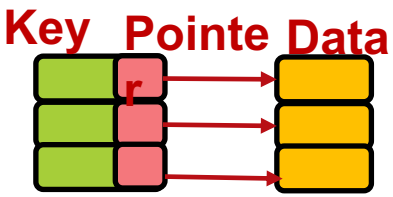
CPU address space

GPU address space



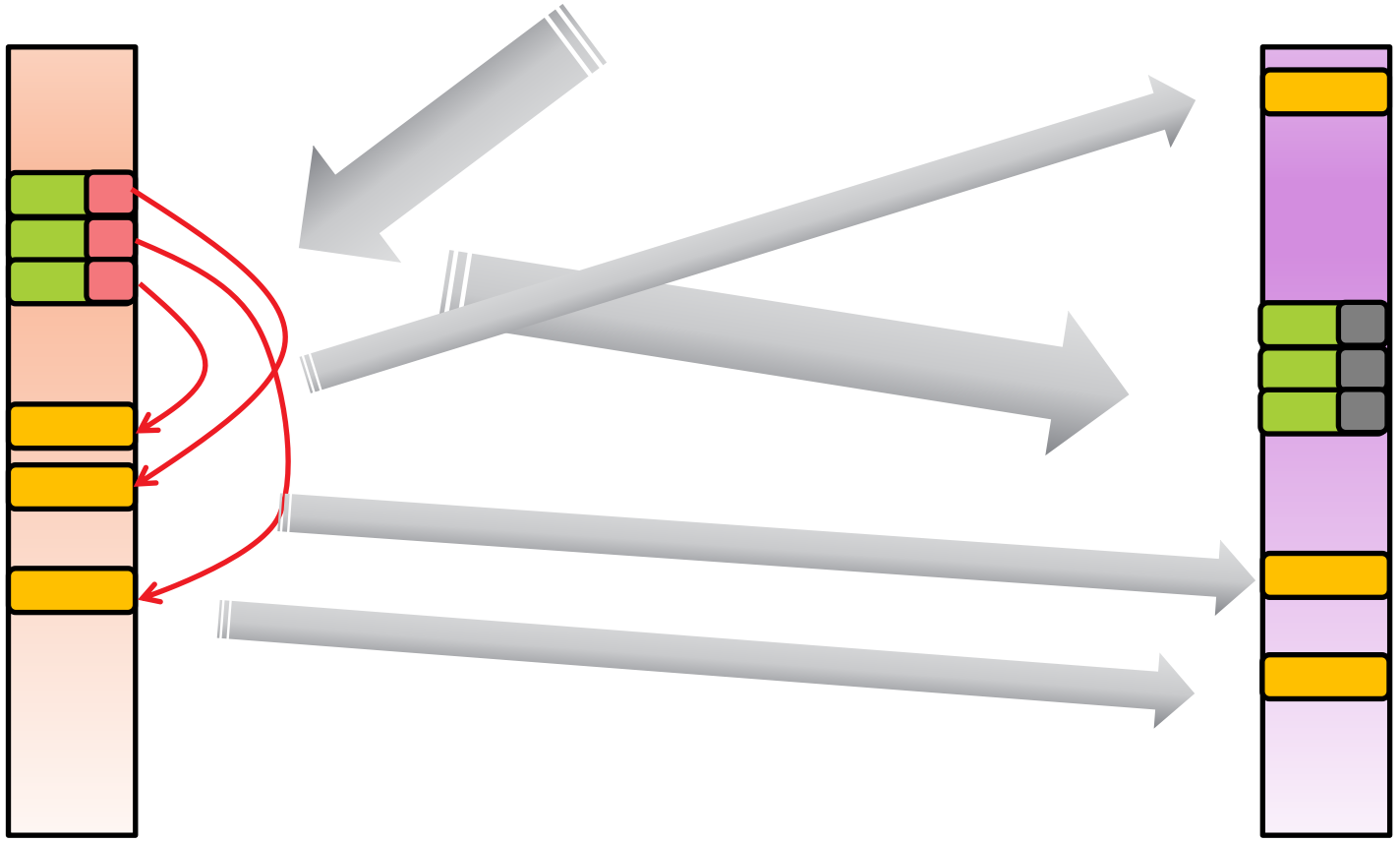
Programming Pointer-based Data Structures Hard Without UVM

Example: Map data-structure



CPU address space

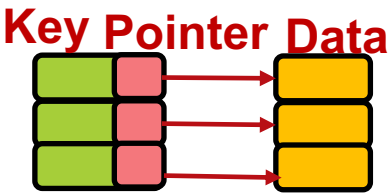
GPU address space



Pointers rendered stale

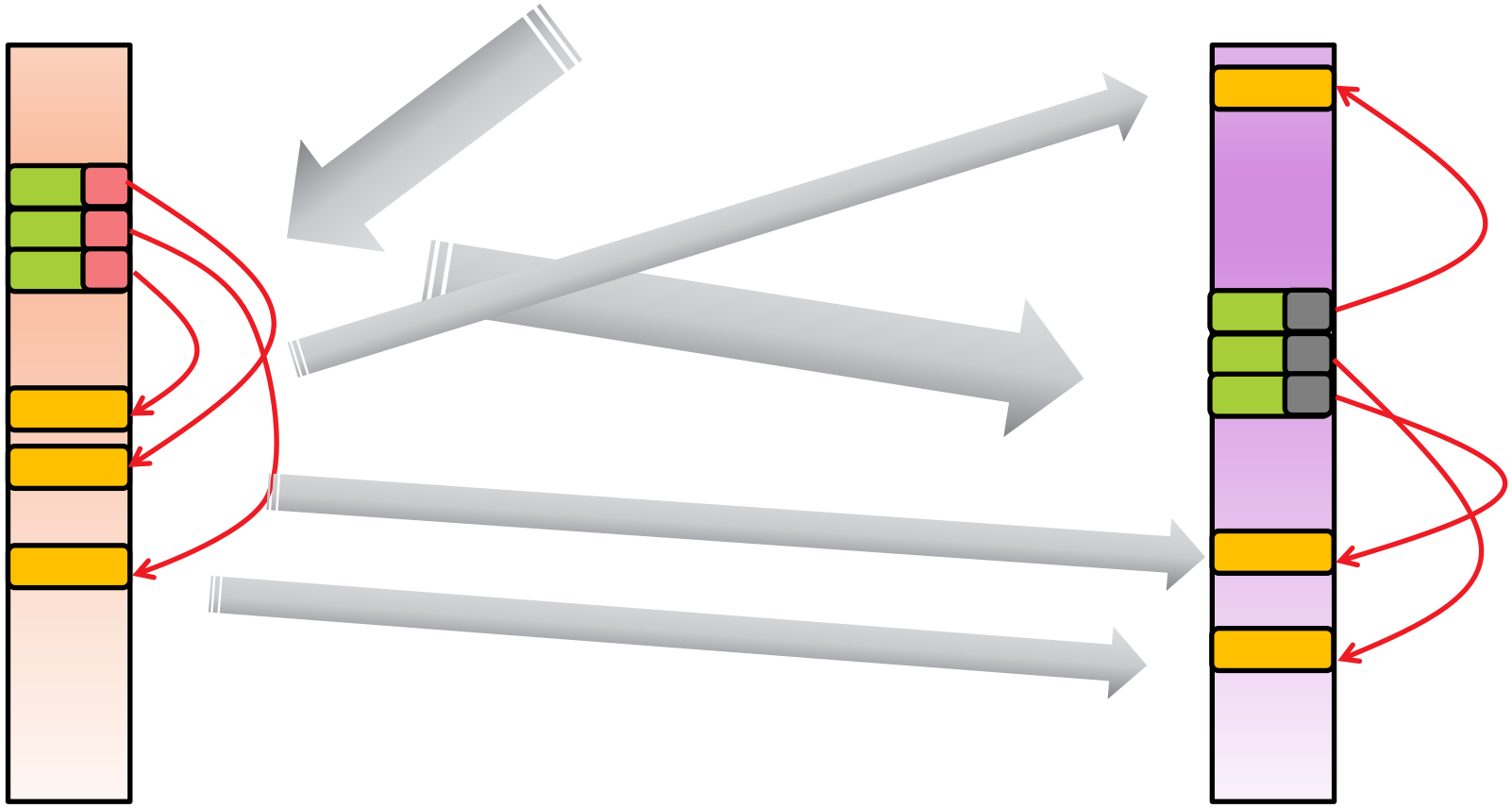
Programming Pointer-based Data Structures Hard Without UVM

Example: Map data-structure



CPU address space

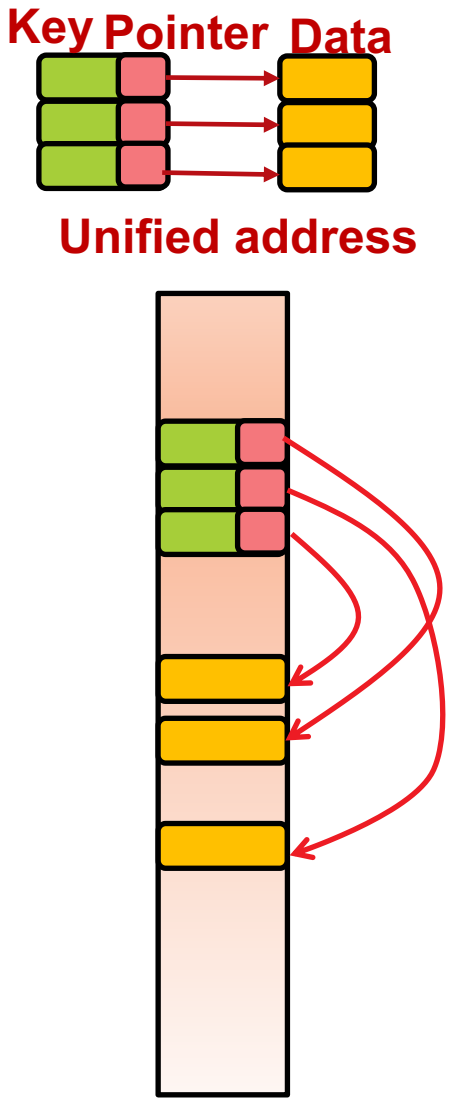
GPU address space



Pointers need to be updated

Programming Pointer-based Data Structures Hard With UVM

Example: Map data-structure

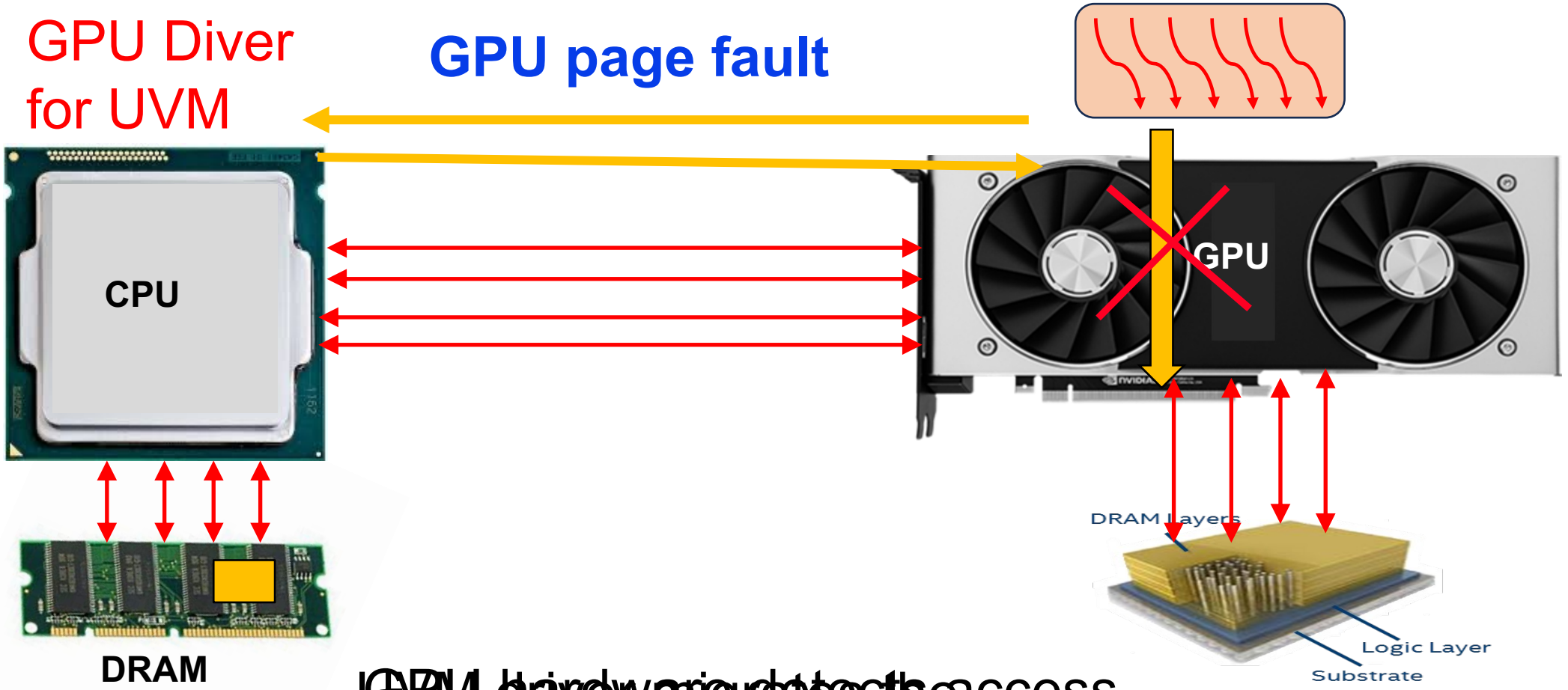


No need for separate memory allocation

No need for pointer update

No need for explicit memcopy

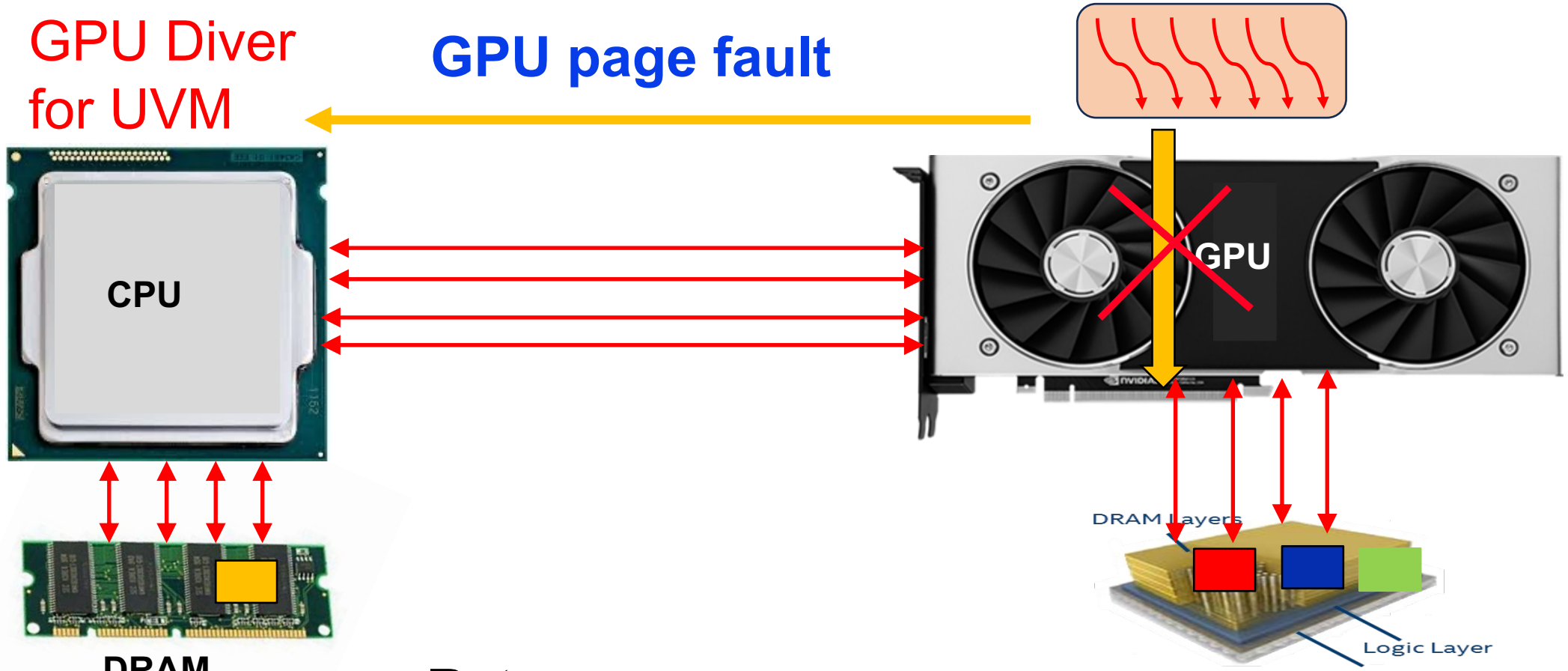
Under the Hood: How Does UVM Work?



Additional Benefits of UVM: Oversubscribing Global Memory

- GPU's global memory limited to a few 10s of GBs
- CUDAmalloc fails if you try to allocate beyond Global memory capacity
- UVM on-demand migration to oversubscribe Global memory capacity

GPU Memory Oversubscription Through On-Deman Migration



Migrate data (page) from Global Memory to DRAM

Retry access

Evict data (page) from Global Memory to DRAM

GPU global memory is full

Summary of Unified Virtual Memory (UVM)

- UVM simplify programming
- UVM enables oversubscription of limited global memory capacity
- UVM uses a combination of hardware and driver to perform on-demand page migration

Key Takeaways of GPU Lectures

- GPUs are accelerator for massive data-parallel processing
- CPU allocates GPU memory, copies data and launches work
- GPU follows Single Instruction Multiple Thread (SIMT) execution
- GPU is programmed using Single Program Multiple Data (SPMD)
- GPU programs must try avoid control flow divergence

Key Takeaways of GPU Lectures

- GPUs have plenty of compute but relatively limited memory capacity and bandwidth
- GPU programs must leverage shared memory, memory access coalescing for performance
- GPUs allows both barrier and fine-grain synchronizations, e.g., atomics, fence
- Tensor cores are specialized to accelerate matrix multiplication