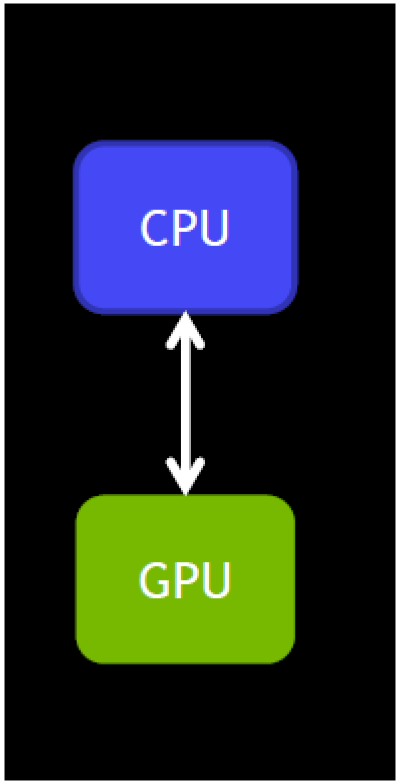
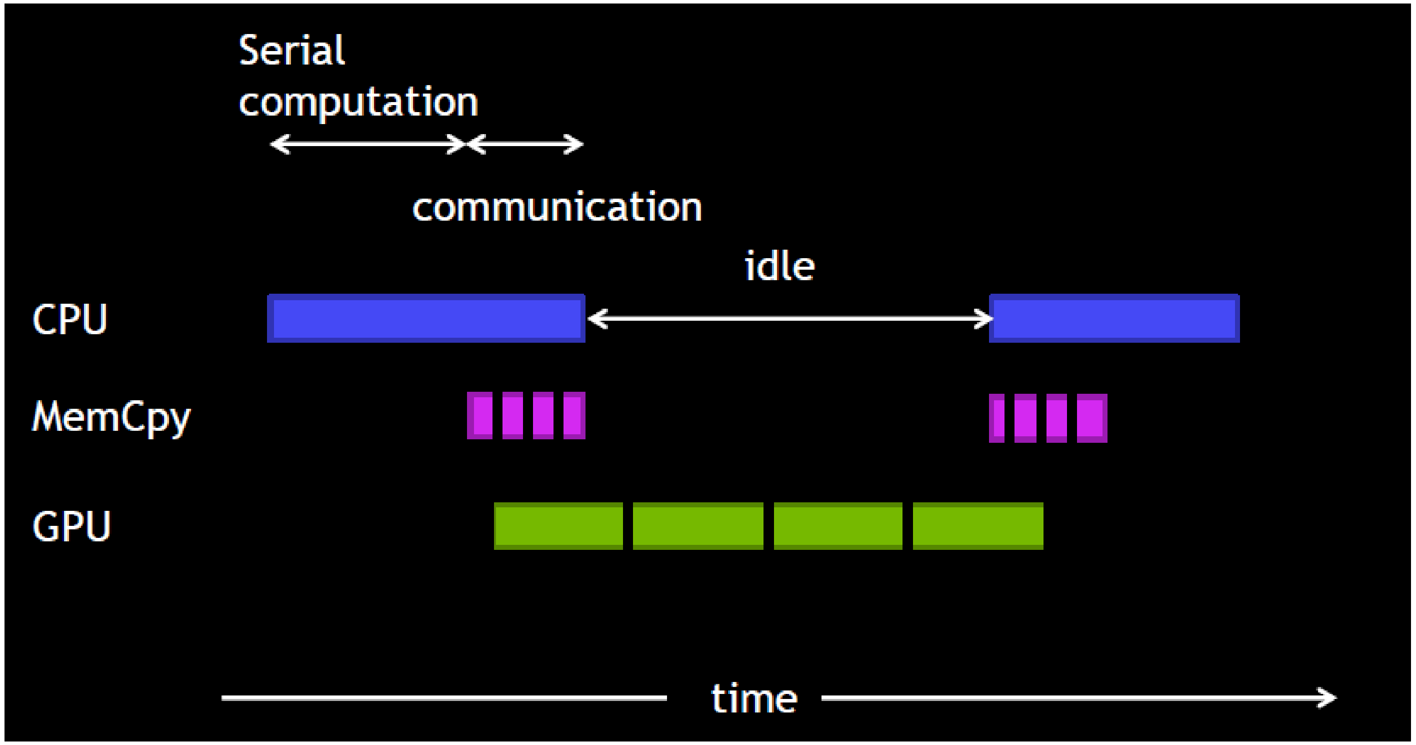
ORTA DOĞU TEKNİK ÜNİVERSİTESİ
MIDDLE EAST TECHNICAL UNIVERSITY

Multi-GPU Programming

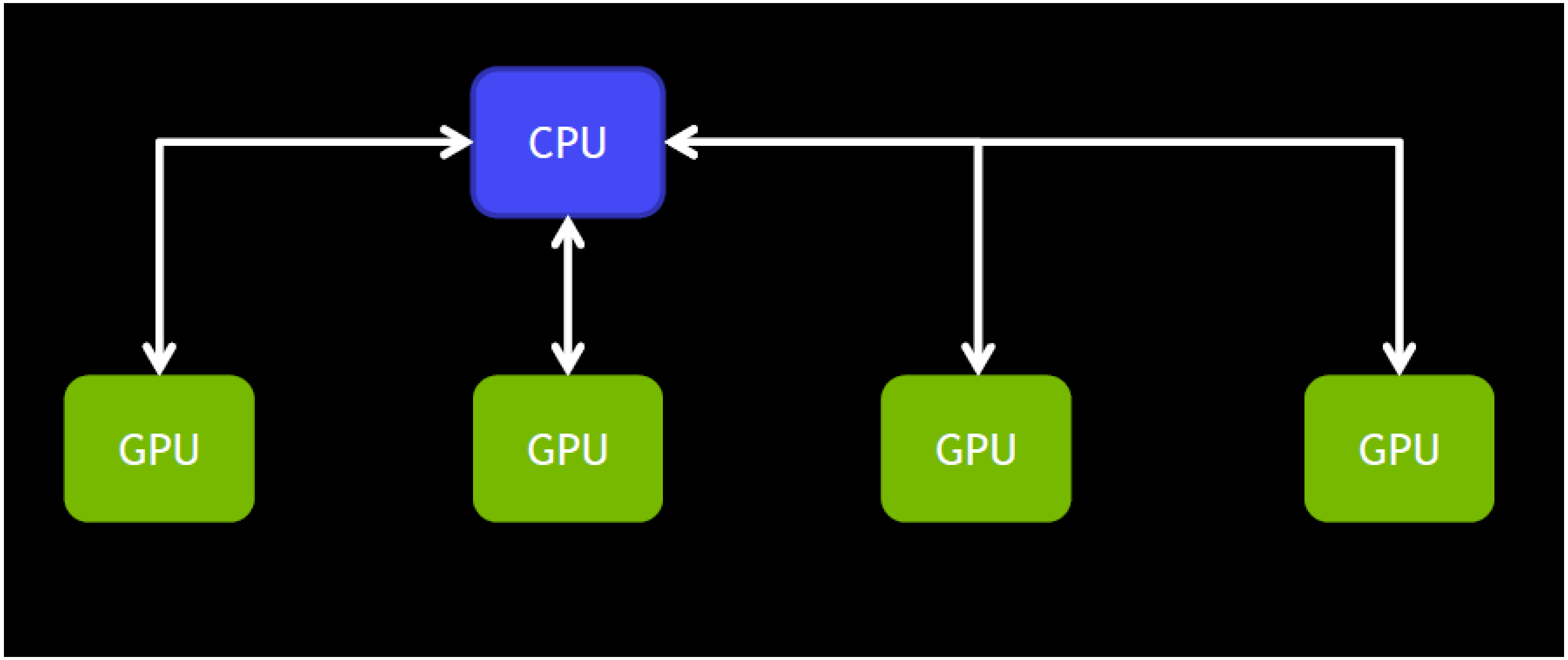
Prof. Dr. Alptekin Temizel
Graduate School of Informatics, METU

Single-GPU Case

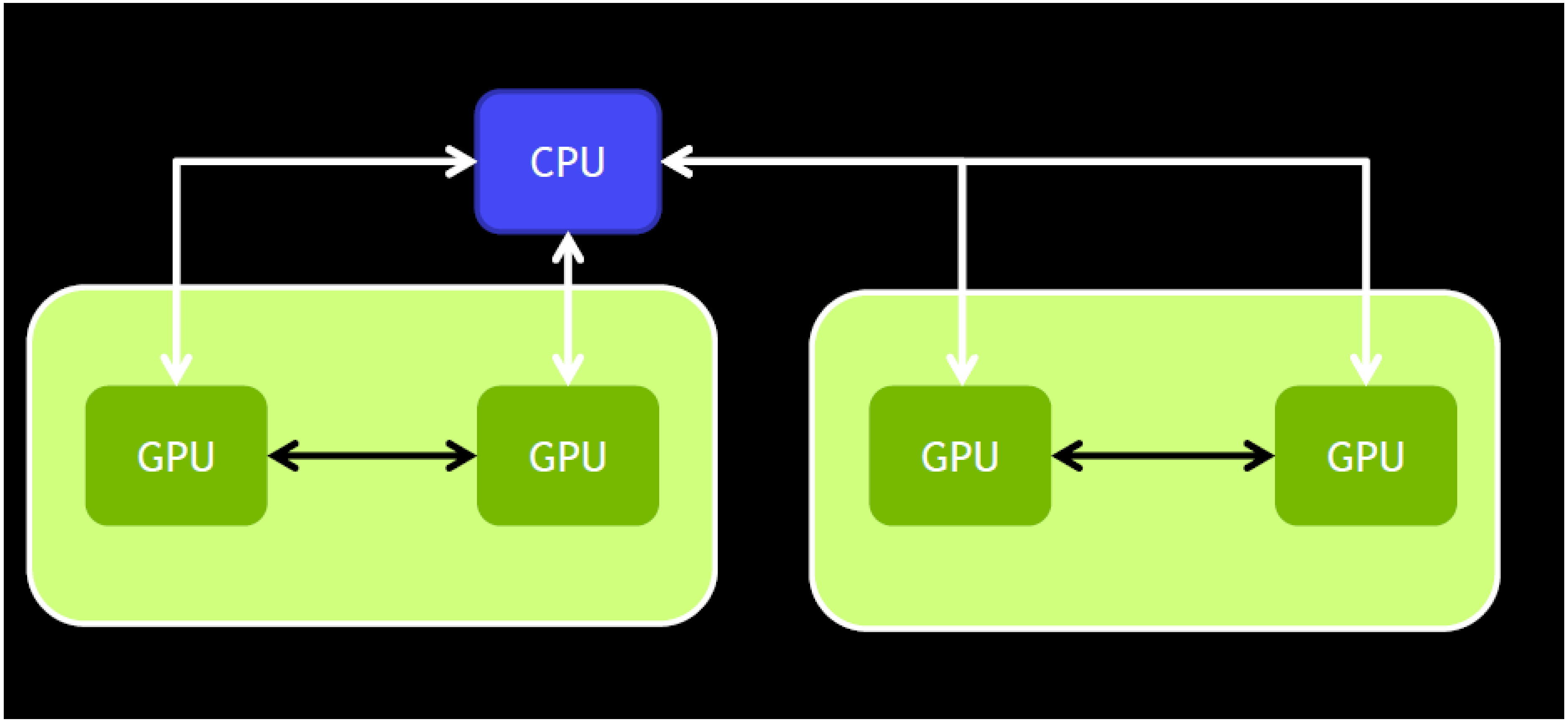




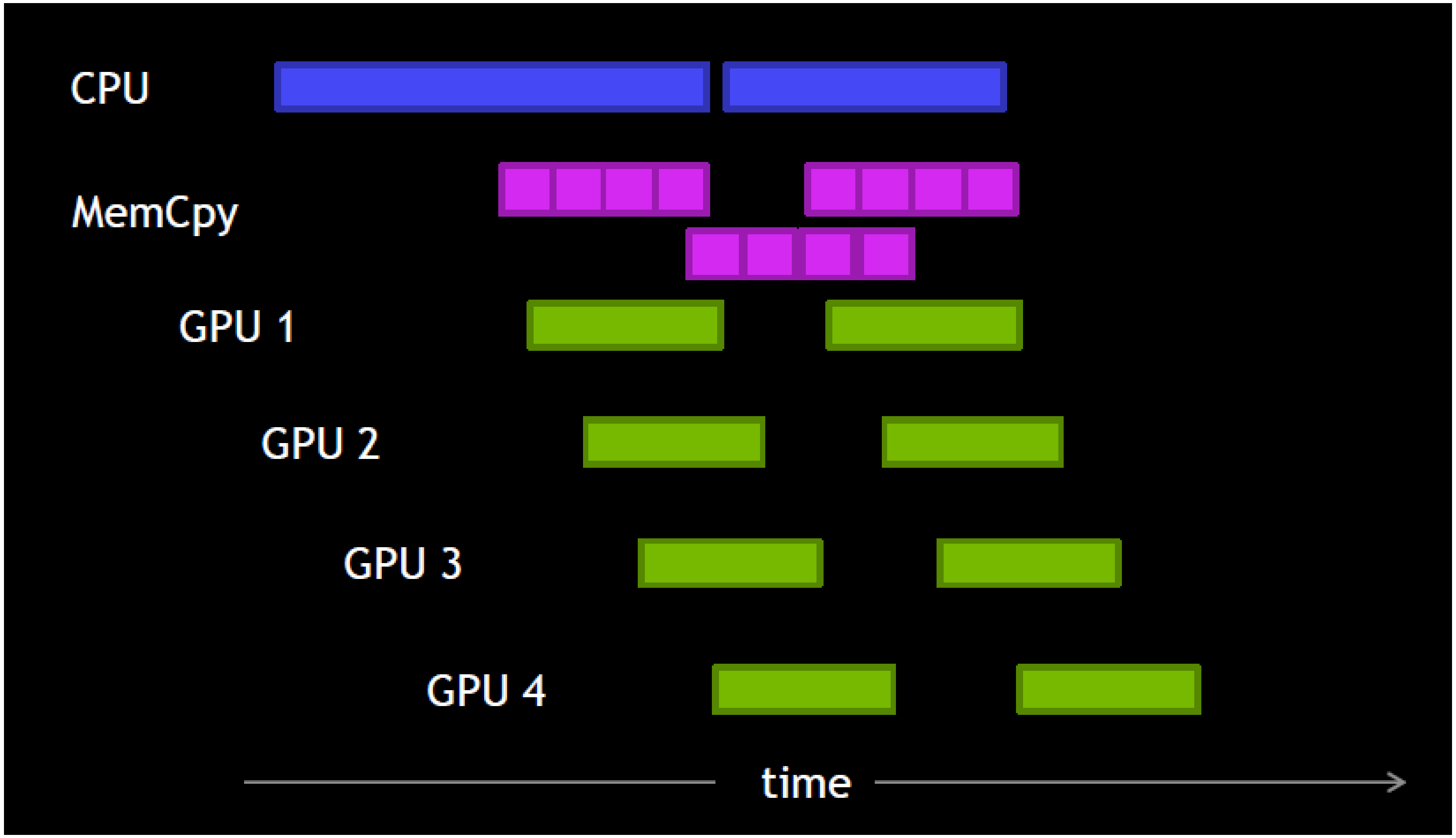
Multi-GPU Case



NVIDIA K10 - Multi-GPU on a Board



Multi-GPU Case



Managing Multiple GPUs

cudaSetDevice() - sets the current GPU

```
cudaSetDevice( 0 );
kernel<<<...>>>(...);
cudaMemcpyAsync(...);
cudaSetDevice( 1 );
kernel<<<...>>>(...);
```

Peer-to-Peer Memcopy

cudaMemcpyPeerAsync(

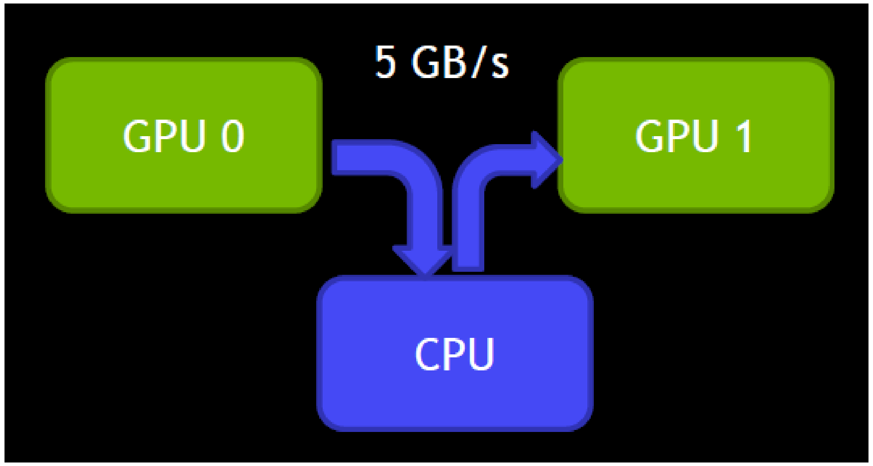
void* dst_addr, **int dst_dev**,

void* src_addr, **int src_dev**,

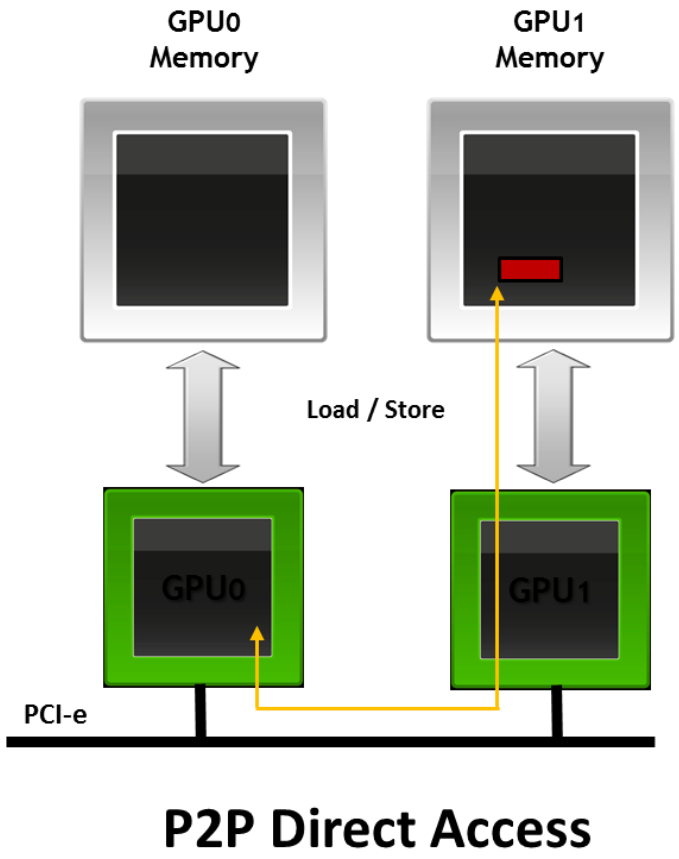
size_t num_bytes,

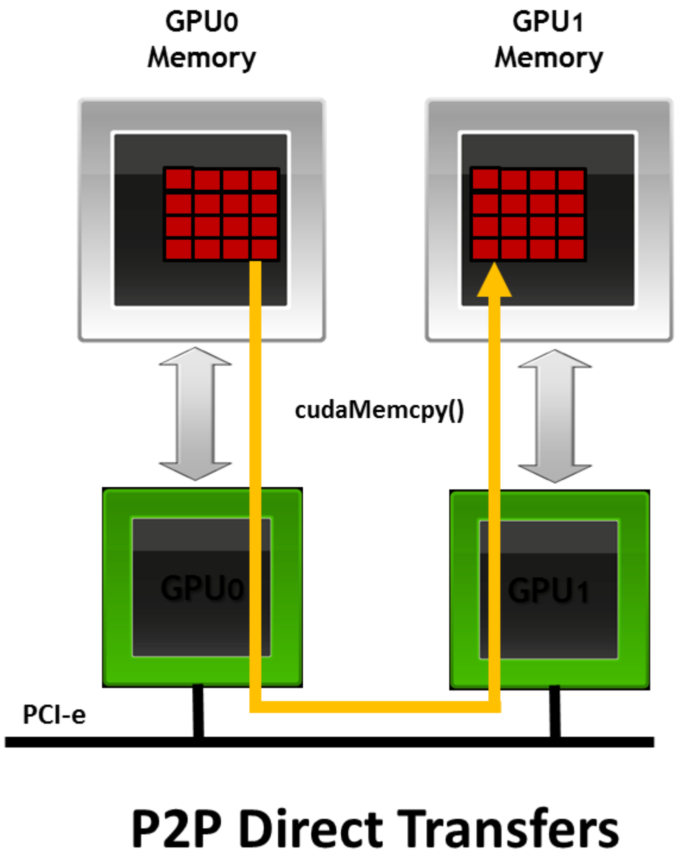
cudaStream_t stream)

If you analyze with the profiler, you'll see two separate copies:
D2H and then H2D



GPUDirect Peer-to-Peer (P2P) Communication Between GPUs on the Same PCIe Bus



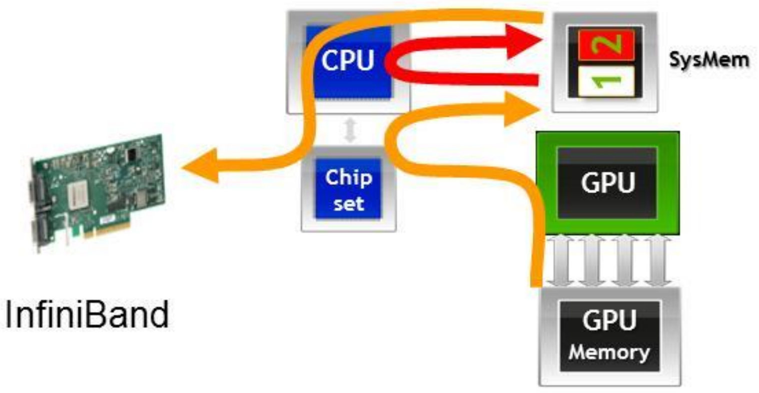


GPU Direct

Without GPUDirect

Same data copied three times:

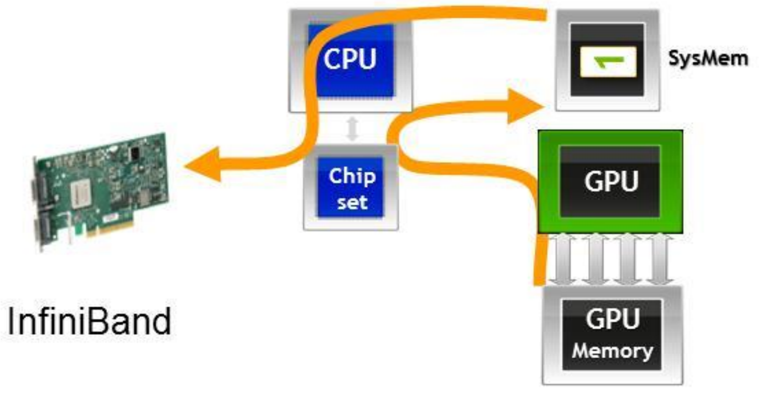
1. GPU writes to pinned sysmem1
2. CPU copies from sysmem1 to sysmem2
3. InfiniBand driver copies from sysmem2



With GPUDirect

Data only copied twice

Sharing pinned system memory makes sysmem-to-sysmem copy unnecessary



Enabling GPU Direct

- cudaDeviceEnablePeerAccess(peer_device, 0)
  - Enables current GPU to access addresses on *peer_device* GPU

- cudaDeviceCanAccessPeer(&accessible, dev_X, dev_Y)
  - Checks whether dev_X can access memory of dev_Y

*Need to have at least FermiCUDA 4.0 and same type devices (not possible between Fermi to Kepler etc.)

Peer-to-Peer Memcopy with GPU Direct

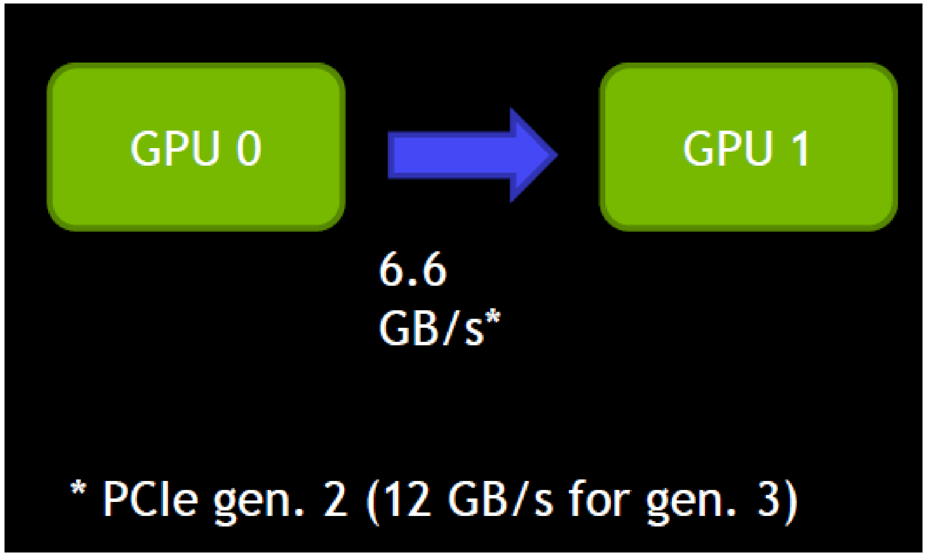
cudaMemcpyPeerAsync(

void* dst_addr, **int dst_dev**,

void* src_addr, **int src_dev**,

size_t num_bytes,

cudaStream_t stream)



Peer-to-Peer Memcopy with GPU Direct

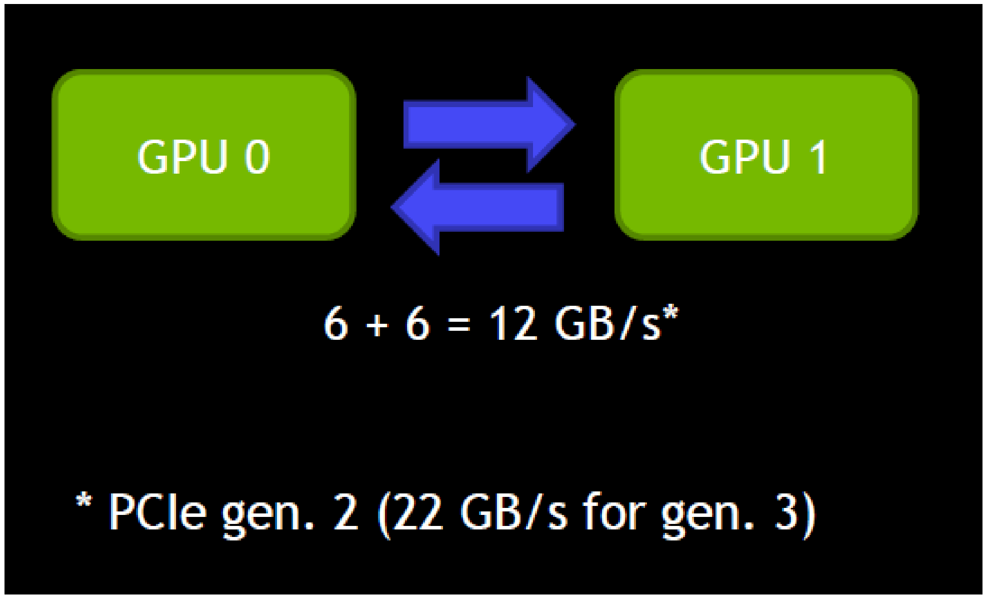
cudaMemcpyPeerAsync(

void* dst_addr, **int dst_dev**,

void* src_addr, **int src_dev**,

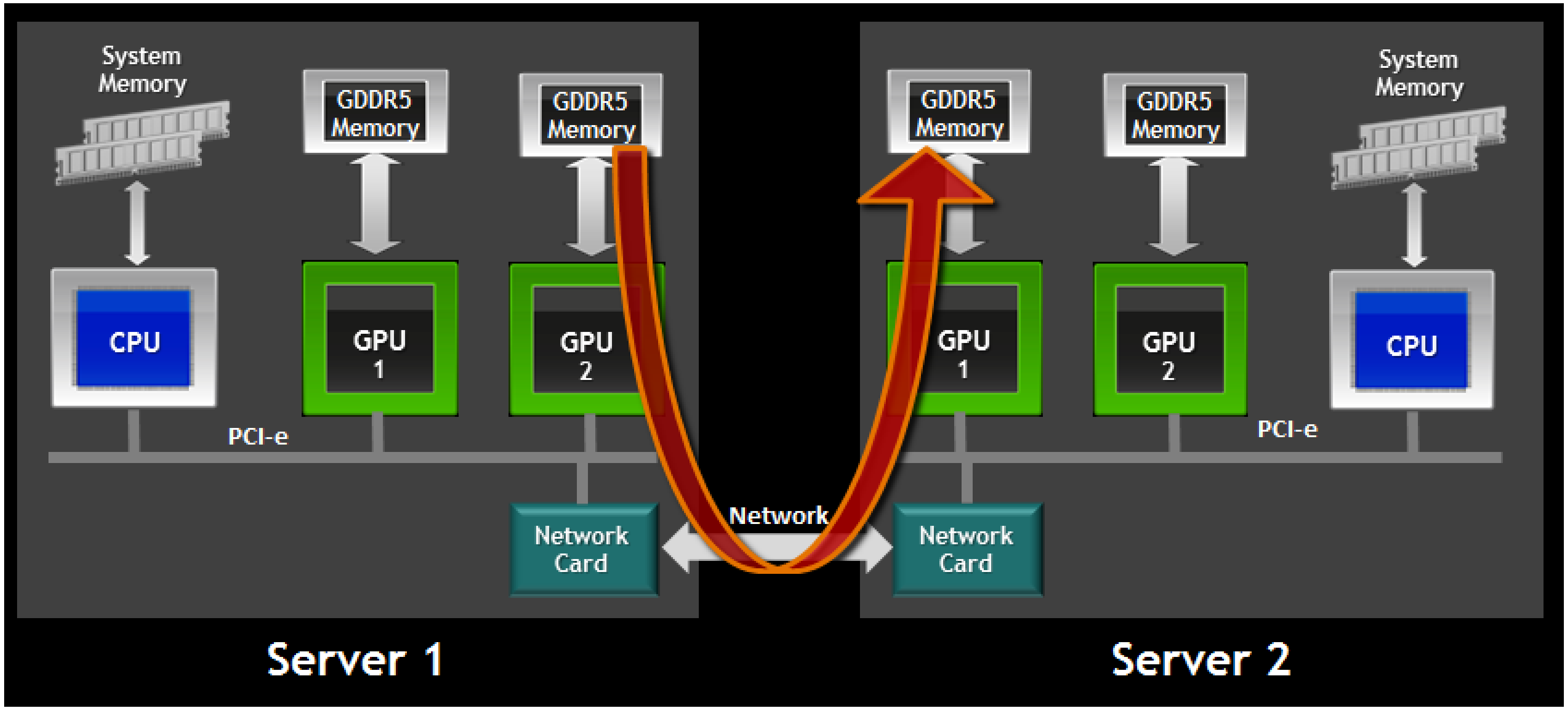
size_t num_bytes,

cudaStream_t stream)



Peer-to-Peer Memcopy with GPU Direct

- GPU Direct RDMA enables third party PCI Express devices to directly access GPU bypassing CPU host memory altogether.
  - Results in significantly improved MPISendRecv efficiency between GPUs and other nodes)



Peer-to-Peer Memcopy with GPU Direct

- GPU Direct RDMA enables third party PCI Express devices to directly access GPU bypassing CPU host memory altogether.
  - Results in significantly improved MPISendRecv efficiency between GPUs and other nodes)
- GPUDirect for Video enables optimized pipeline for frame-based devices such as frame grabbers, video switchers, HD-SDI capture, and CameraLink devices.

Unified Virtual Addressing (UVA)

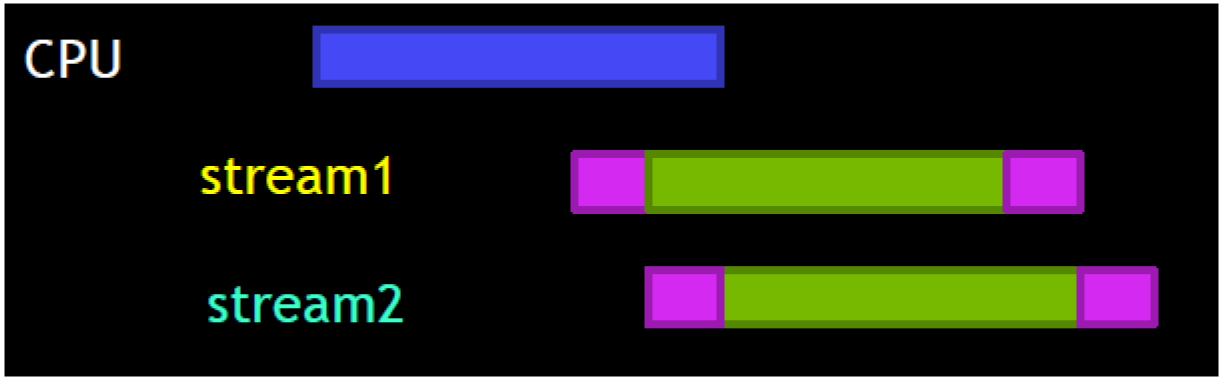
- Driver/GPU can determine from an address where data resides (whether in CPU or GPU and on which GPU as well)
  - Requires:
  - 64-bit Linux or 64-bit Windows with TCC (Tesla Compute Cluster) driver
  - Fermi or later architecture GPUs (compute capability 2.0 or higher)
  - CUDA 4.0 or later

Peer-to-peer memcopy

- cudaMemcpyAsync(void* **dst_addr**, void* **src_addr**, size_t num_bytes, cudaStream_t stream, cudaMemcpyDefault)
  - Just give dst and src addresses and it'll decide on to copy from where to where

Reminder – Overlapping kernel and memcopy

```
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaHostAlloc(&src, size, 0);
…
cudaMemcpyAsync( dst, src, size, dir, stream1 );
kernel<<<grid, block, 0, stream1>>>(…);
cudaMemcpyAsync( dst, src, size, dir, stream2 );
kernel<<<grid, block, 0, stream2>>>(…);
```



Reminder – Creating CUDA Events

- cudaEventCreate(cudaEvent_t *event)
  - Creates an event object
- cudaEventDestroy(cudaEvent_t event)
  - Destroys the specified event
- cudaEventRecord(cudaEvent_t event, cudaStream_t stream=0)
  - Records an event.
  - If stream is non-zero, the event is recorded after all preceding operations in the stream have been completed;
  - Otherwise, it is recorded after all preceding operations in the CUDA context have been completed.
  - cudaEventQuery() or cudaEventSynchronize() must be used to determine when the event has actually been recorded.

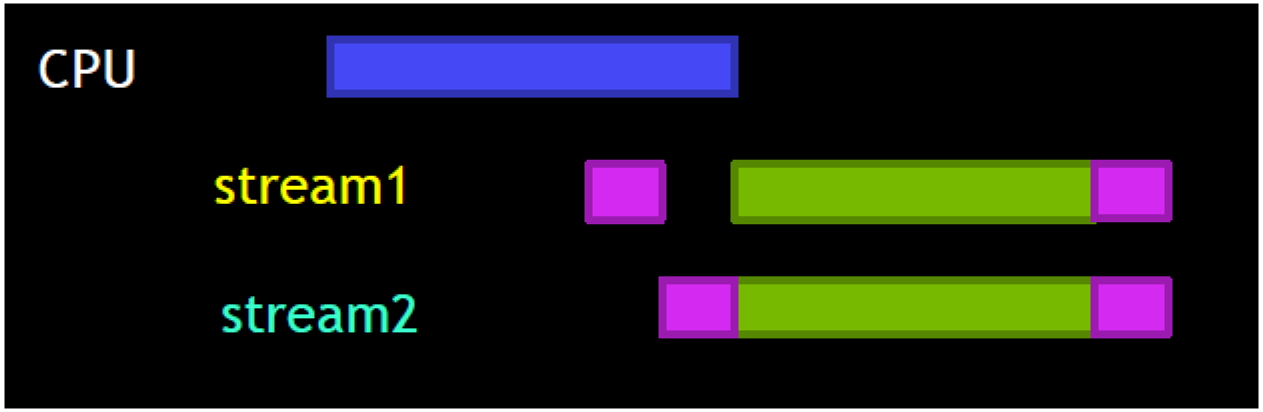
Reminder – Using CUDA Events

- cudaEventSynchronize(cudaEvent_t event)
  - Wait until the completion of all device work preceding the event

- cudaStreamWaitEvent(cudaStream_t stream, cudaEvent_t event)
  - Makes all future work submitted to the stream wait until event reports completion before beginning execution.
  - This synchronization will be performed efficiently on the device.

Reminder – Using CUDA Events

```
cudaEvent_t ev;
cudaEventCreate(&ev);
…
cudaMemcpyAsync( dst, src, size, dir, stream1 );
cudaMemcpyAsync( dst2, src2, size, dir, stream2 );
cudaEventRecord( ev, stream2 );
cudaStreamWaitEvent(stream1, ev);
kernel<<<grid, block, 0, stream1>>>(…);
kernel<<<grid, block, 0, stream2>>>(…);
```



Multi-GPU Streams and Events

- CUDA streams and events are *per device* (GPU)
    - Each device has its own *default* stream (stream 0)
- **Kernels:** can be launched to a stream only if the stream's GPU is current
- **Memcopies:** can be issued to any stream
- **Events:** can be recorded only to a stream if the stream's GPU is current
- It is OK to query or synchronize with any event/stream

Multi-GPU Streams and Events

```
cudaStream_t streamA, streamB;
cudaEvent_t eventA, eventB;
cudaSetDevice( 0 );
cudaStreamCreate( &streamA ); // streamA and eventA belong to device-0
cudaEventCreate( &eventA );
cudaSetDevice( 1 );
cudaStreamCreate( &streamB ); // streamB and eventB belong to device-1
cudaEventCreate( &eventB );
kernel<<<..., streamB>>>(...);
cudaEventRecord( eventB, streamA );
cudaEventSynchronize( eventB );
```

NOT OK:
- device 1 is current
- streamA belongs to device 0

Multi-GPU Streams and Events

```
cudaStream_t streamA, streamB;
cudaEvent_t eventA, eventB;
cudaSetDevice( 0 );
cudaStreamCreate( &streamA ); // streamA and eventA belong to device-0
cudaEventCreate( &eventA );
cudaSetDevice( 1 );
cudaStreamCreate( &streamB ); // streamB and eventB belong to device-1
cudaEventCreate( &eventB );
kernel<<<..., streamA>>>(...);
cudaEventRecord( eventB, streamB );
cudaEventSynchronize( eventB );
```

NOT OK:
- device 1 is current
- streamA belongs to device 0

Multi-GPU Streams and Events

```
cudaStream_t streamA, streamB;
cudaEvent_t eventA, eventB;
cudaSetDevice( 0 );
cudaStreamCreate( &streamA ); // streamA and eventA belong to device-0
cudaEventCreate( &eventA );

cudaSetDevice( 1 );
cudaStreamCreate( &streamB ); // streamB and eventB belong to device-1
cudaEventCreate( &eventB );
kernel<<<..., streamB>>>(...);
cudaEventRecord( eventB, streamB );

cudaSetDevice( 0 );
cudaEventSynchronize( eventB );
kernel<<<..., streamA>>>(...);
```

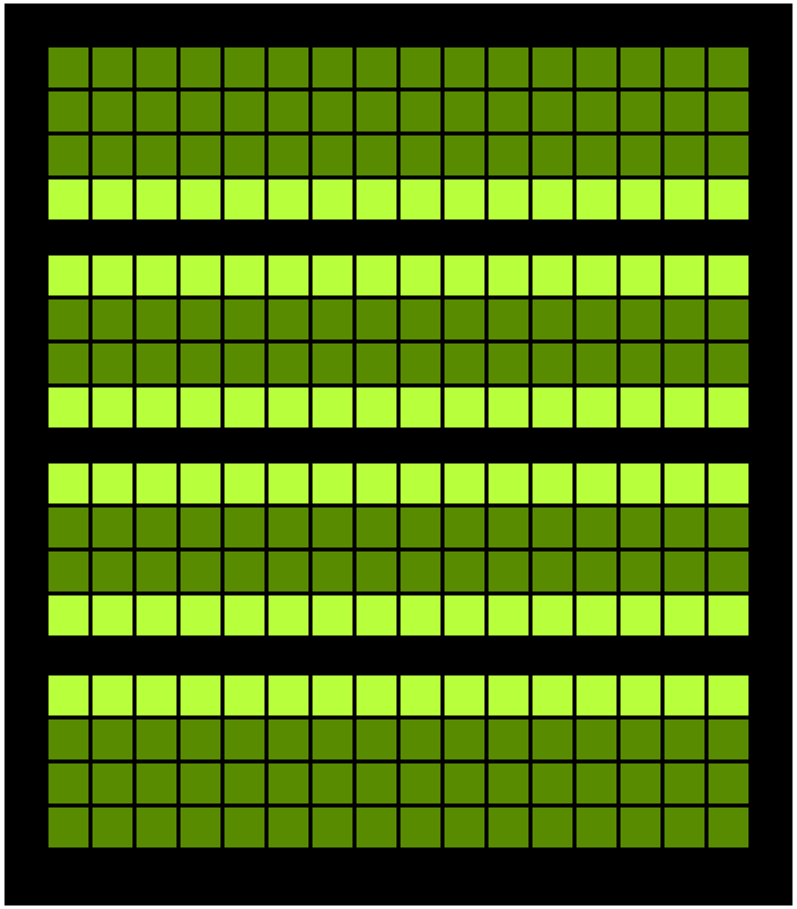
- Synchronizing/querying events/streams of other devices is allowed

Multi-GPU Streams and Events

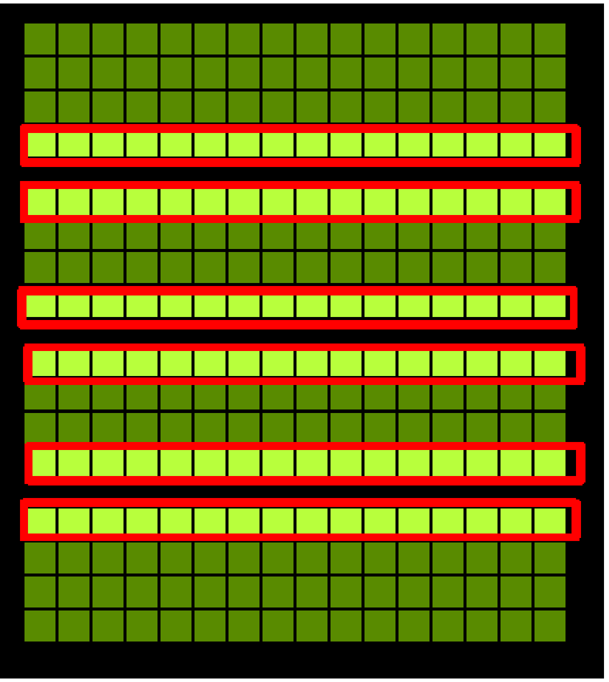
```
cudaSetDevice( 0 );
cudaMalloc( &d_A, num_bytes );
int accessible = 0;
cudaDeviceCanAccessPeer( &accessible, 1, 0 );
if( accessible )
{
        cudaSetDevice( 1 );
        cudaDeviceEnablePeerAccess( 0, 0 );
        kernel<<<...>>>( d_A);
}
```

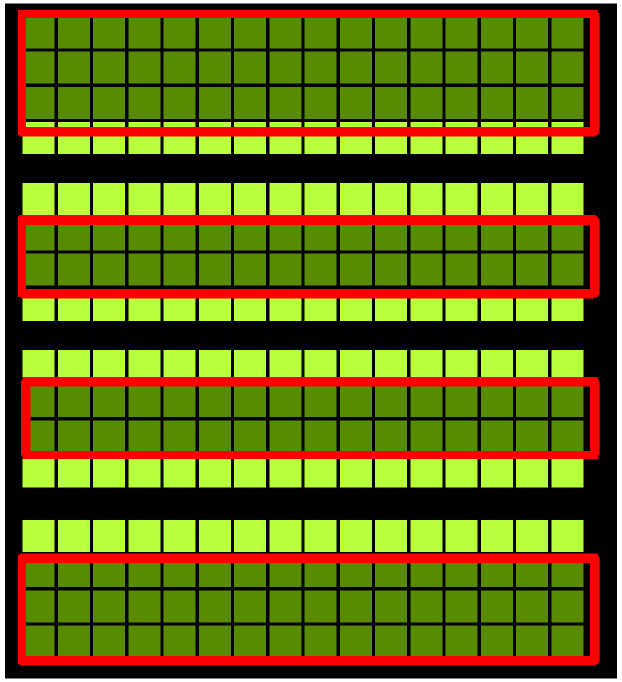
- Kernel executes on Device 1
- It will access (via PCIe) memory on Device 0

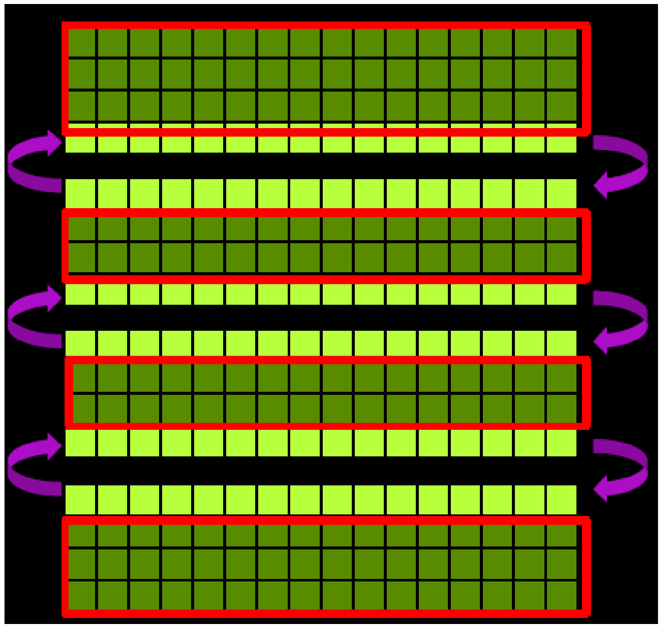
Subdividing the Problem



Subdividing the Problem





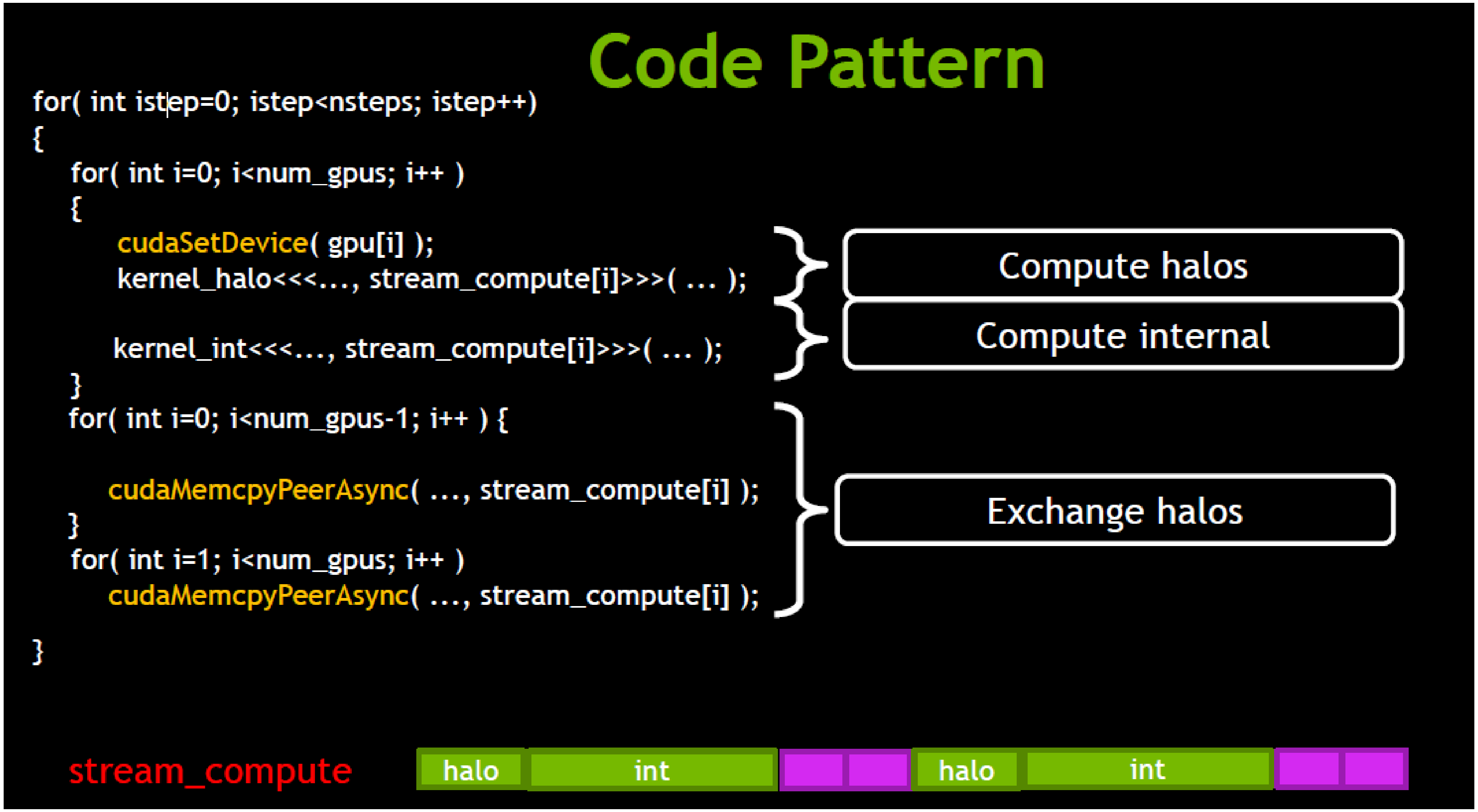


Step 1
Compute Halo Regions

Step 2
Compute Internal Regions

Step 3
Exchange Halo Regions

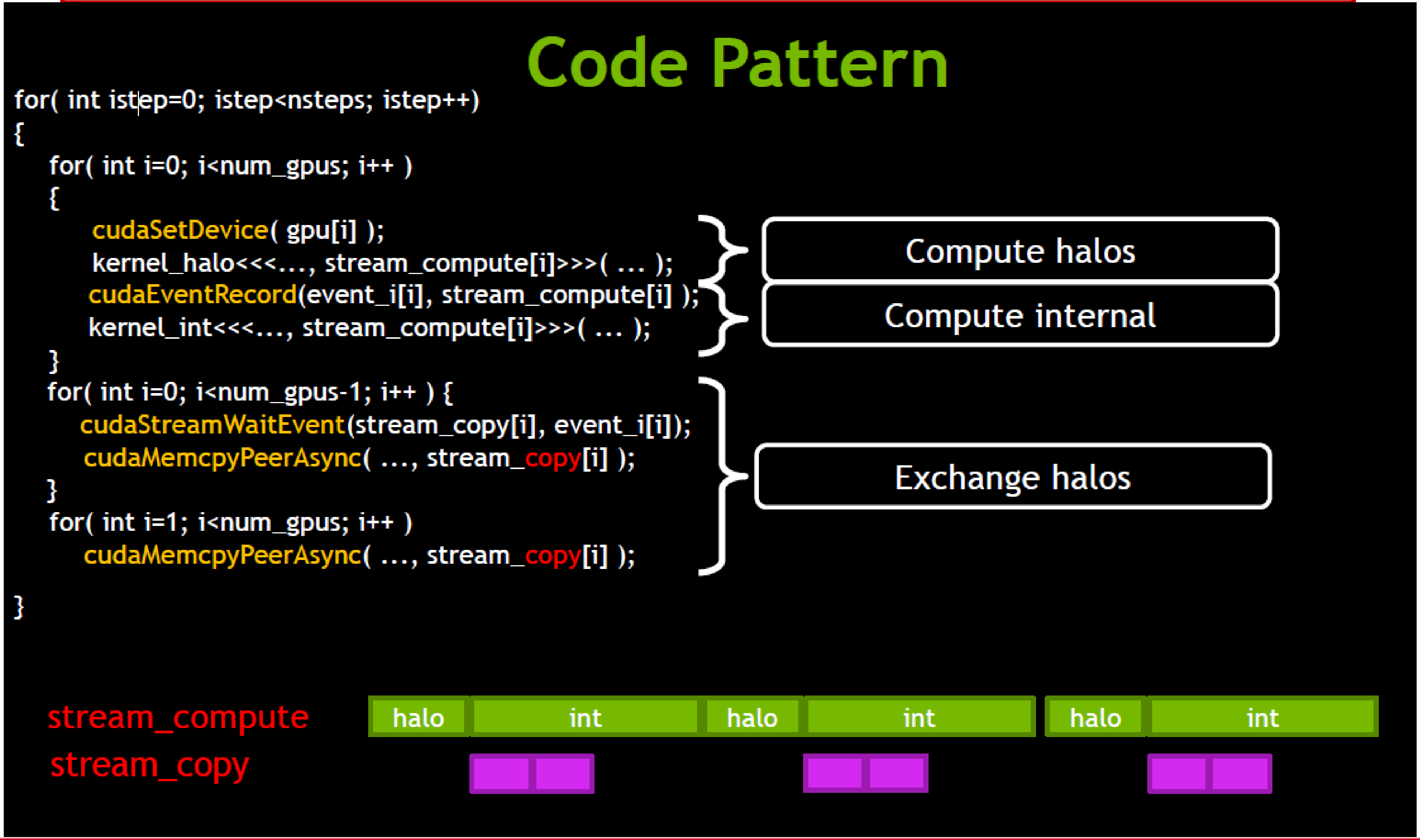
Subdividing the Problem



Code Pattern

```
for( int istep=0; istep<nsteps; istep++)
{
   for( int i=0; i<num_gpus; i++ )
   {
       cudaSetDevice( gpu[i] );
       kernel_halo<<<..., stream_compute[i]>>>( ... );

      kernel_int<<<..., stream_compute[i]>>>( ... );
   }
   for( int i=0; i<num_gpus-1; i++ ) {

      cudaMemcpyPeerAsync( ..., stream_compute[i] );
   }
   for( int i=1; i<num_gpus; i++ )
      cudaMemcpyPeerAsync( ..., stream_compute[i] );

}
```

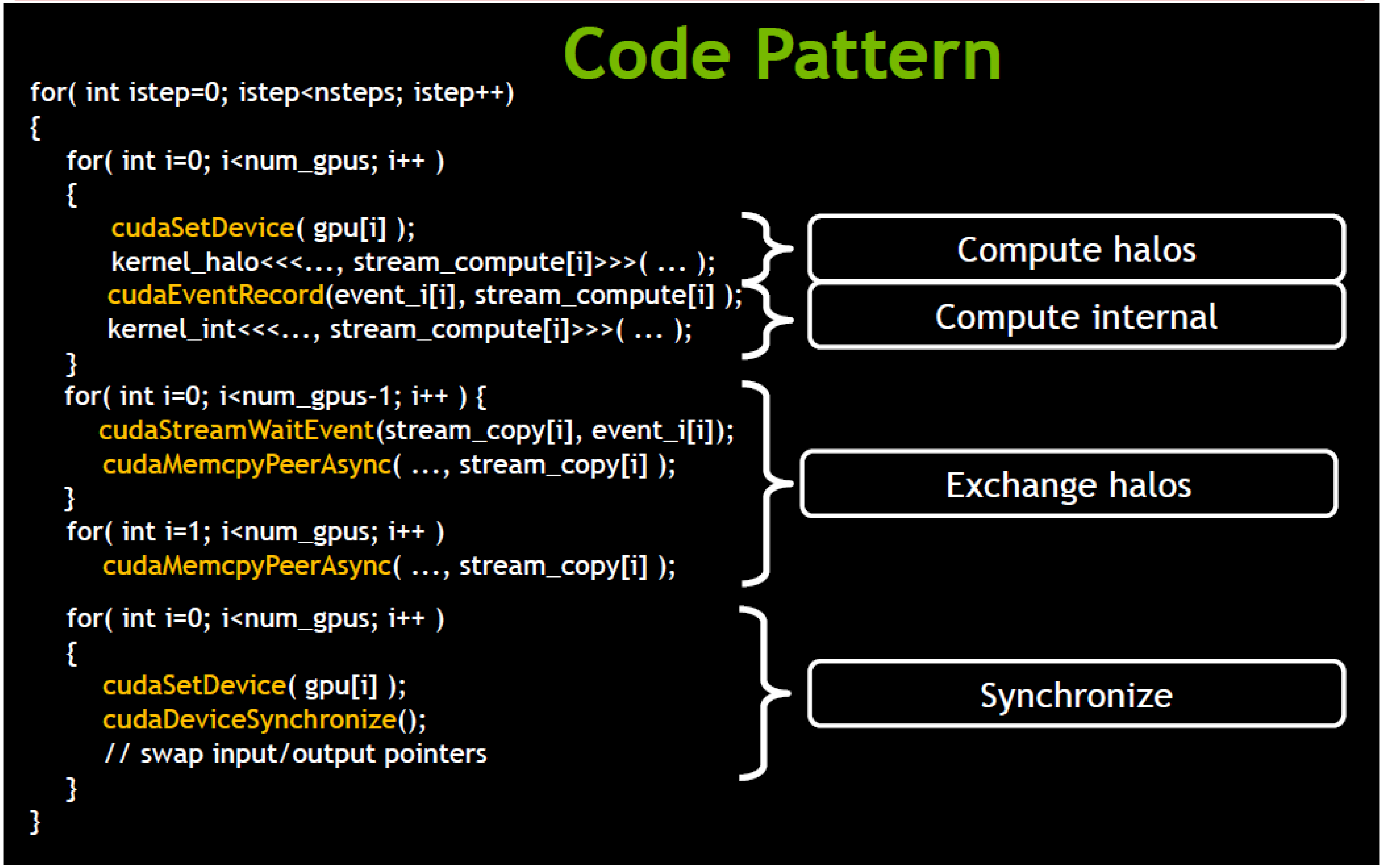
Subdividing the Problem



Code Pattern

```
for( int istep=0; istep<nsteps; istep++)
{
    for( int i=0; i<num_gpus; i++ )
    {
        cudaSetDevice( gpu[i] );
        kernel_halo<<<..., stream_compute[i]>>>( ... );
        cudaEventRecord(event_i[i], stream_compute[i] );
        kernel_int<<<..., stream_compute[i]>>>( ... );
    }
    for( int i=0; i<num_gpus-1; i++ ) {
        cudaStreamWaitEvent(stream_copy[i], event_i[i]);
        cudaMemcpyPeerAsync( ..., stream_copy[i] );
    }
    for( int i=1; i<num_gpus; i++ )
        cudaMemcpyPeerAsync( ..., stream_copy[i] );

}
```

Subdividing the Problem



Code Pattern

```
for( int istep=0; istep<nsteps; istep++)
{
    for( int i=0; i<num_gpus; i++ )
    {
        cudaSetDevice( gpu[i] );
        kernel_halo<<<..., stream_compute[i]>>>( ... );
        cudaEventRecord(event_i[i], stream_compute[i] );
        kernel_int<<<..., stream_compute[i]>>>( ... );
    }
    for( int i=0; i<num_gpus-1; i++ ) {
        cudaStreamWaitEvent(stream_copy[i], event_i[i]);
        cudaMemcpyPeerAsync( ..., stream_copy[i] );
    }
    for( int i=1; i<num_gpus; i++ )
        cudaMemcpyPeerAsync( ..., stream_copy[i] );

    for( int i=0; i<num_gpus; i++ )
    {
        cudaSetDevice( gpu[i] );
        cudaDeviceSynchronize();
        // swap input/output pointers
    }
}
```

Compute halos

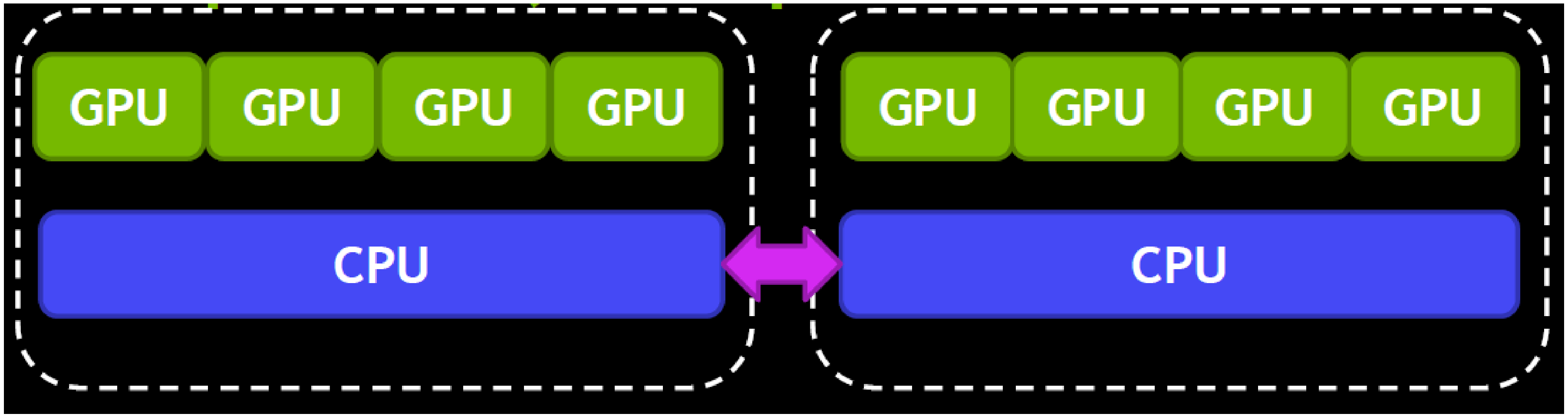
Compute internal

Exchange halos

Synchronize

MPI

- MPI (Message Passing Interface) is a standardized and portable API for communicating data via messages (both point-to-point & collective) between distributed processes.
- MPI is frequently used in HPC to build applications that can scale on multi-node computer clusters.
- In most MPI implementations, library routines are directly callable from C, C++, and Fortran.



Combining MPI with CUDA

- To solve problems with a data size that is too large to fit into the memory of a single GPU.
- To solve problems that would require unreasonably long compute time on a single node.
- To accelerate an existing MPI application with GPUs.
- To enable a single-node multi-GPU application to scale across multiple nodes.

CUDA Aware MPI

- Regular MPI implementations pass pointers to host memory, staging GPU buffers through host memory using cudaMemcpy.

- With CUDA-aware MPI, the MPI library can send and receive GPU buffers directly, without having to first stage them in host memory.

- Implementation of CUDA-aware MPI was simplified by Unified Virtual Addressing (UVA) in CUDA 4.0 – which enables a single address space for all CPU and GPU memory.

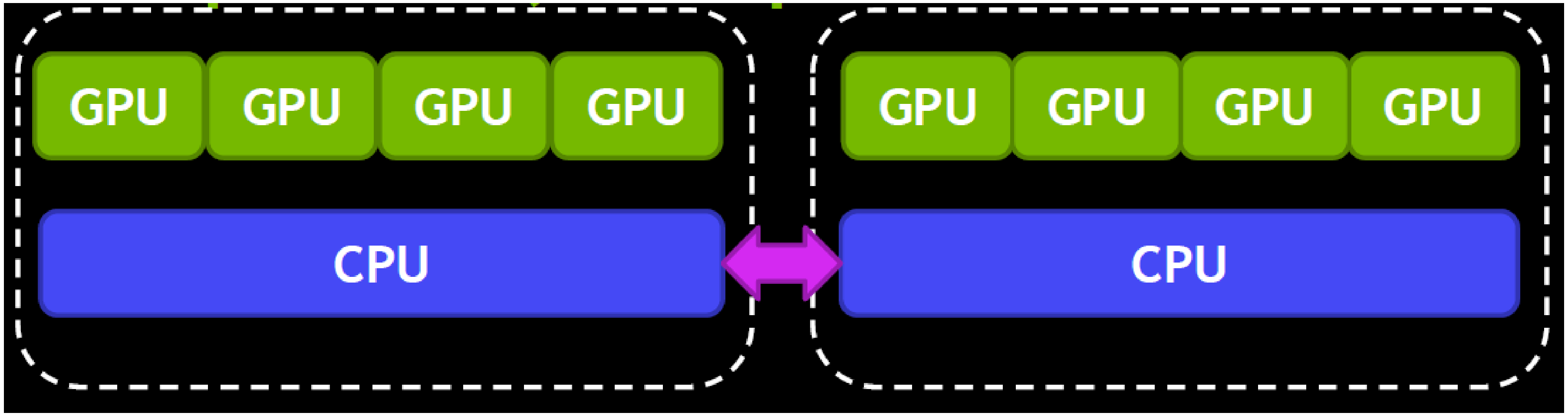
CUDA Aware MPI

- MVAPICH2 is an open source MPI implementation
  - Simplifies the task of porting MPI applications to run on clusters with NVIDIA GPUs by supporting standard MPI calls from GPU device memory.

- IBM® Platform™ MPI is a high performance, production–quality MPI implementation
  - Supports a broad range of industry standard platforms, interconnects and operating systems.

- The Open MPI Project is an open source MPI-2 implementation that is developed and maintained by a consortium of academic, research, and industry partners.
  - GPUs are supported by version 1.7 and later.

Multiple GPU, Multiple Nodes



```
cudaMemcpyAsync( ..., stream_halo[i] );
cudaStreamSynchronize( stream_halo[i] );
MPI_Sendrecv( ... );
cudaMemcpyAsync( ..., stream_halo[i] );
```

NVLink

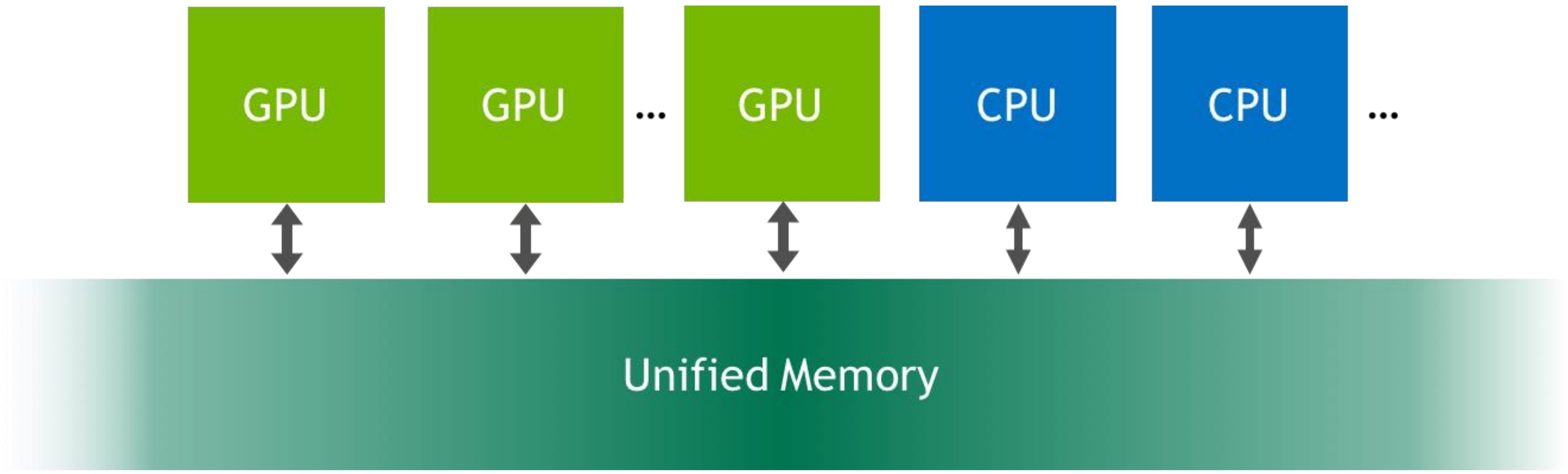
- NVLink is NVIDIA's new high-speed data interconnect. NVLink can be used to significantly increase performance for both GPU-to-GPU communication and for GPU access to system memory. GP100 supports up to four NVLink connections with each connection carrying up to 40 GB/s of bi-directional bandwidth.

- NVLink is available between GPU to GPU transfers
  - x86 devices: CPU to GPU transfers are through PCIe
  - Power8/Power9: CPU to GPU transfers are also through NVLink

NVLink

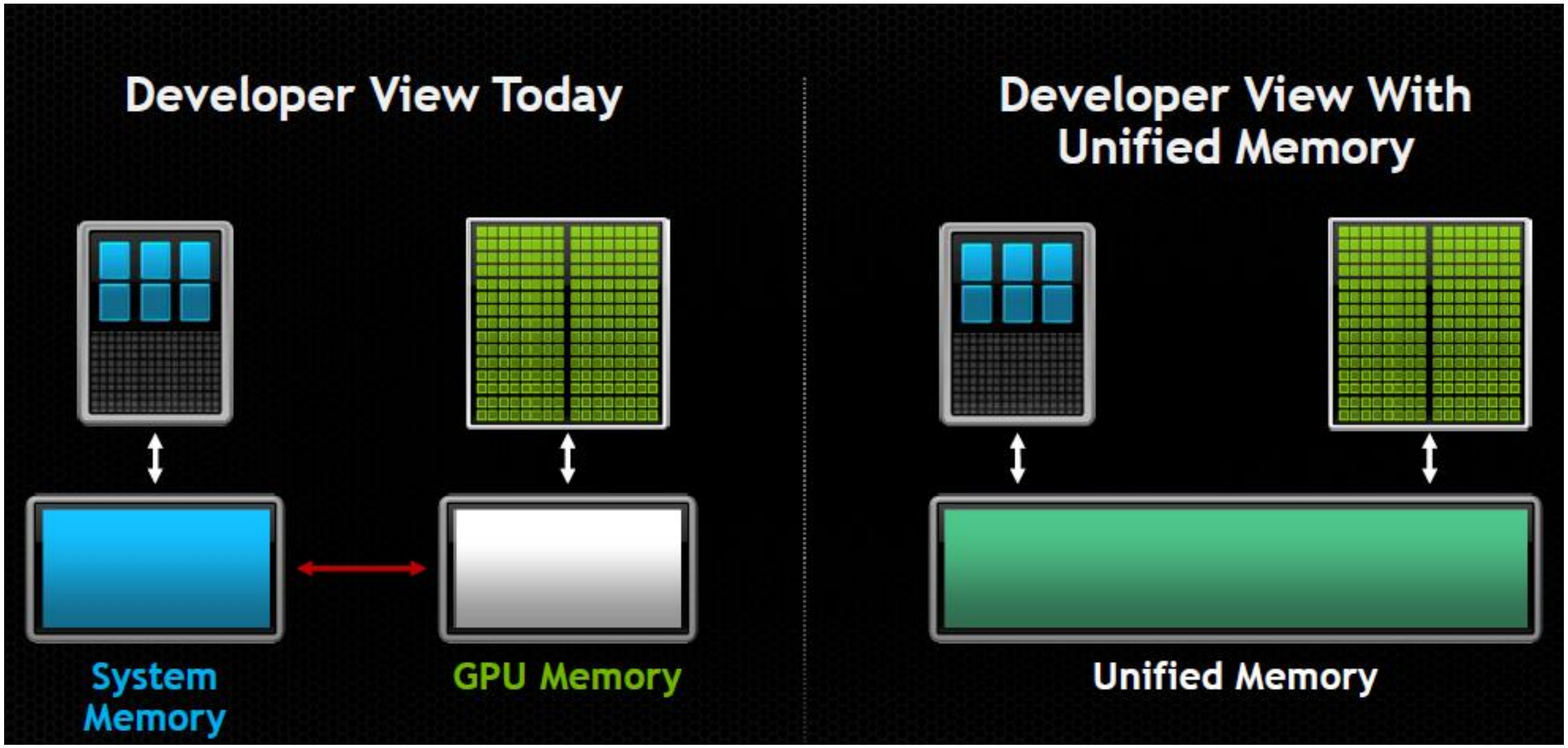
- NVLink operates transparently within the existing CUDA model. Transfers between NVLink-connected endpoints are automatically routed through NVLink, rather than PCIe.
- The cudaDeviceEnablePeerAccess() API call remains necessary to enable direct transfers (over either PCIe or NVLink) between GPUs. The cudaDeviceCanAccessPeer() can be used to determine if peer access is possible between any pair of GPUs.

Unified Memory



Unified Memory is a single memory address space accessible from any processor in a system

Unified Memory



Unified Memory

CPU Version

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);
  fread(data, 1, N, fp);
  qsort(data, N, 1, compare);
  use_data(data);
  free(data);
}
```

GPU Unified Memory Version

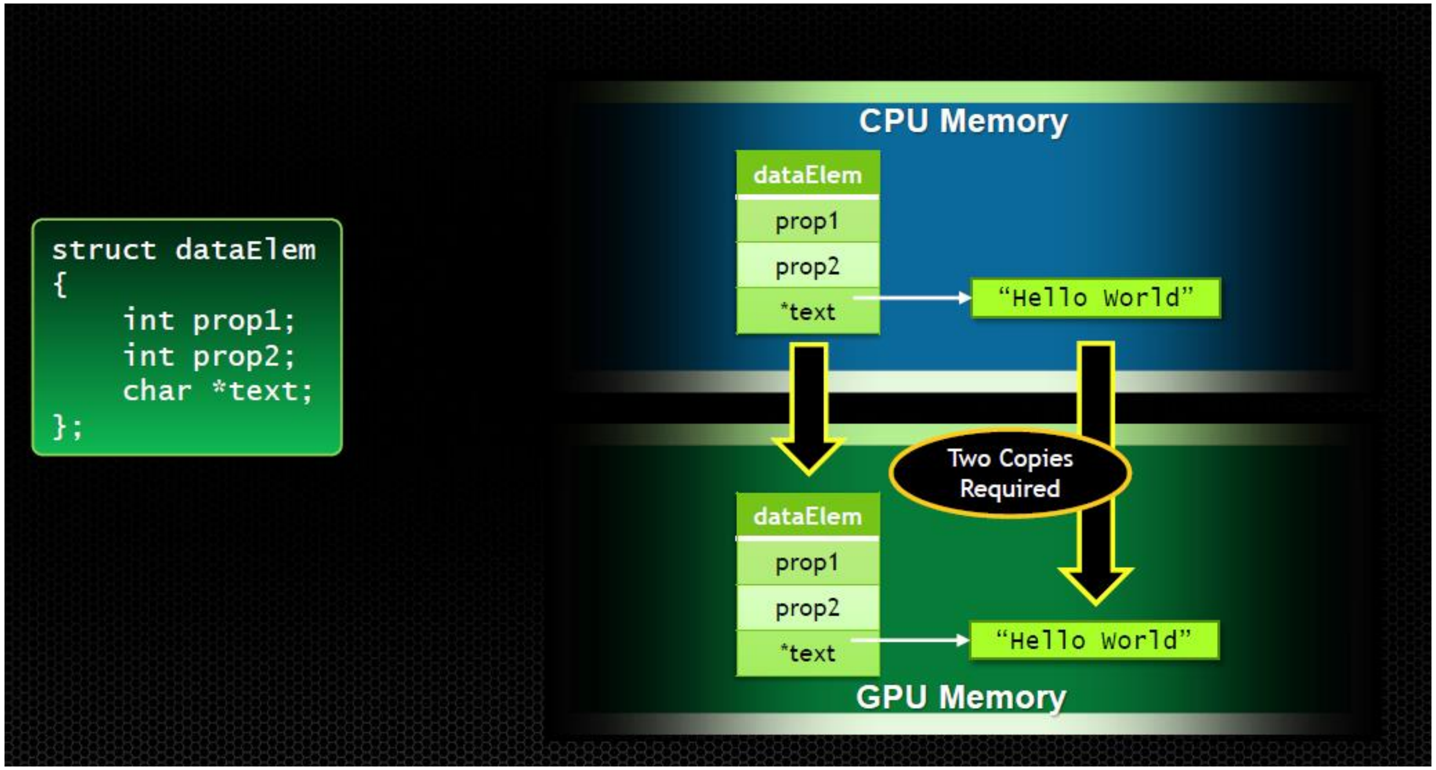
```
void sortfile(FILE *fp, int N) {
  char *data;
  cudaMallocManaged(&data, N);
  fread(data, 1, N, fp);
  qsort<<<...>>>(data,N,1,compare);
  cudaDeviceSynchronize();
  use_data(data);
  cudaFree(data);
}
```

Unified Memory

- Single pointer to data, accessible anywhere
- Guarantee global coherency
- Still allows *cudaMemcpyAsync()* hand tuning
- Eliminates the need for deep copying when using structs

Unified Memory

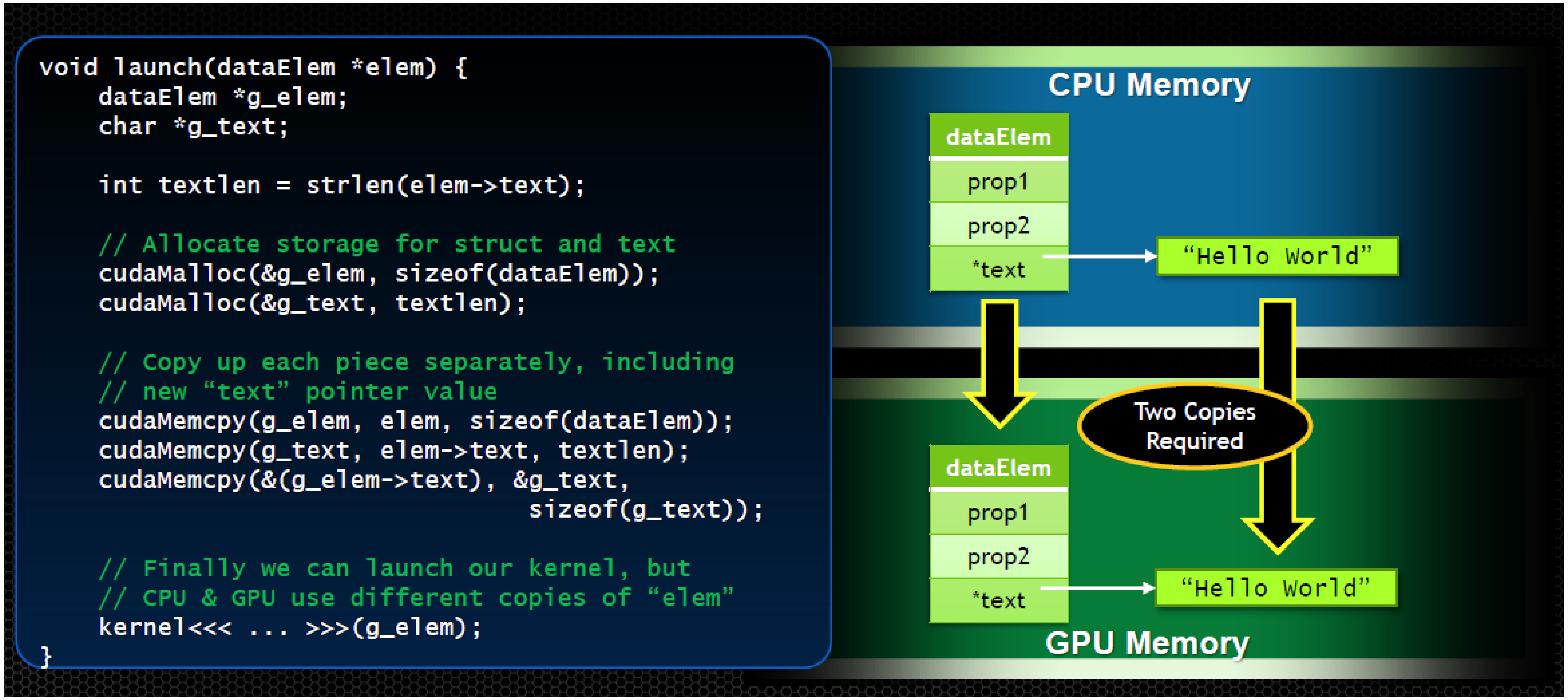
- Eliminates the need for deep copying when using structs



```
struct dataElem
{
    int prop1;
    int prop2;
    char *text;
};
```

Unified Memory

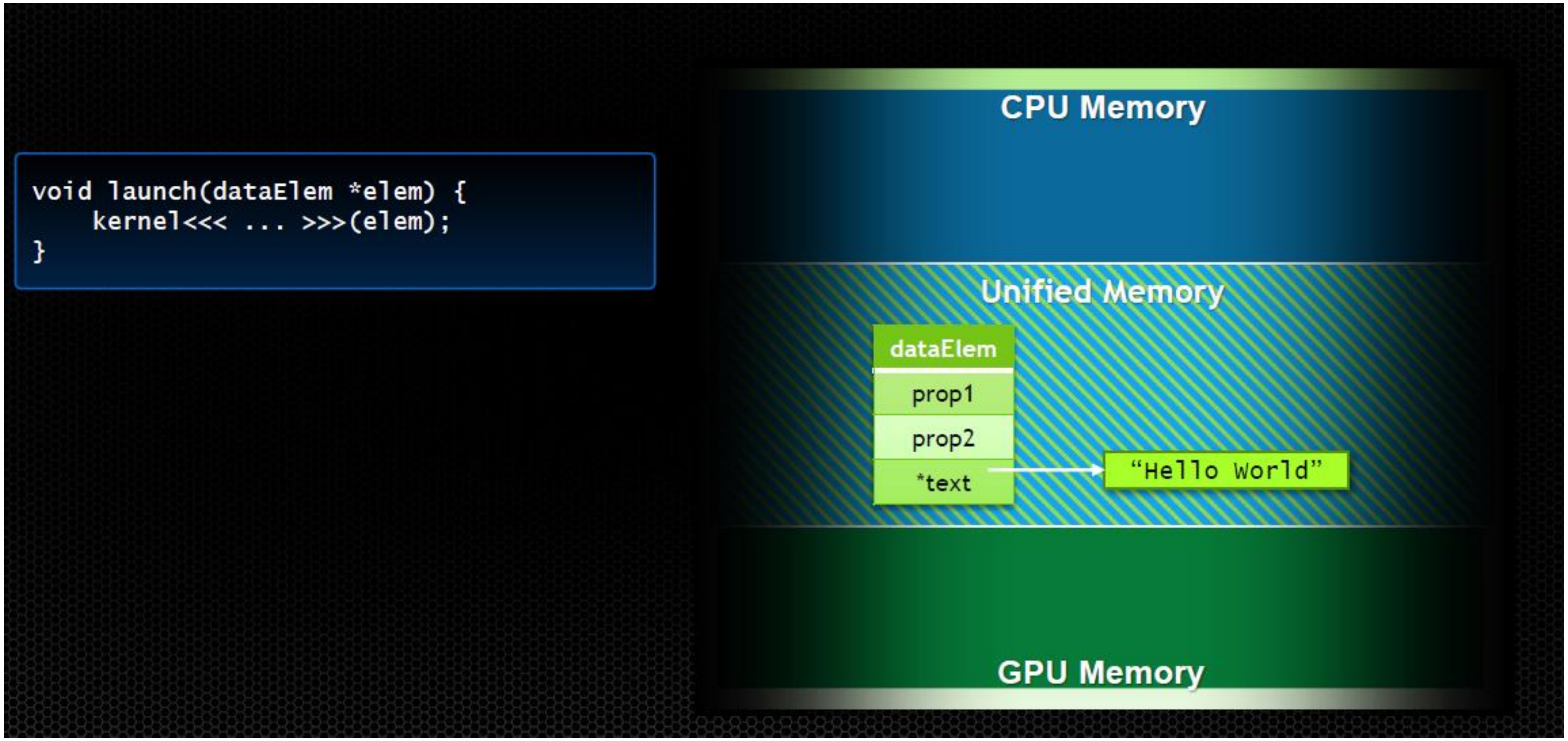
- Eliminates the need for deep copying when using structs



```
void launch(dataElem *elem) {
    dataElem *g_elem;
    char *g_text;

    int textlen = strlen(elem->text);

    // Allocate storage for struct and text
    cudaMalloc(&g_elem, sizeof(dataElem));
    cudaMalloc(&g_text, textlen);

    // Copy up each piece separately, including
    // new “text” pointer value
    cudaMemcpy(g_elem, elem, sizeof(dataElem));
    cudaMemcpy(g_text, elem->text, textlen);
    cudaMemcpy(&(g_elem->text), &g_text,
                                sizeof(g_text));

    // Finally we can launch our kernel, but
    // CPU & GPU use different copies of “elem”
    kernel<<< ... >>>(g_elem);
}
```

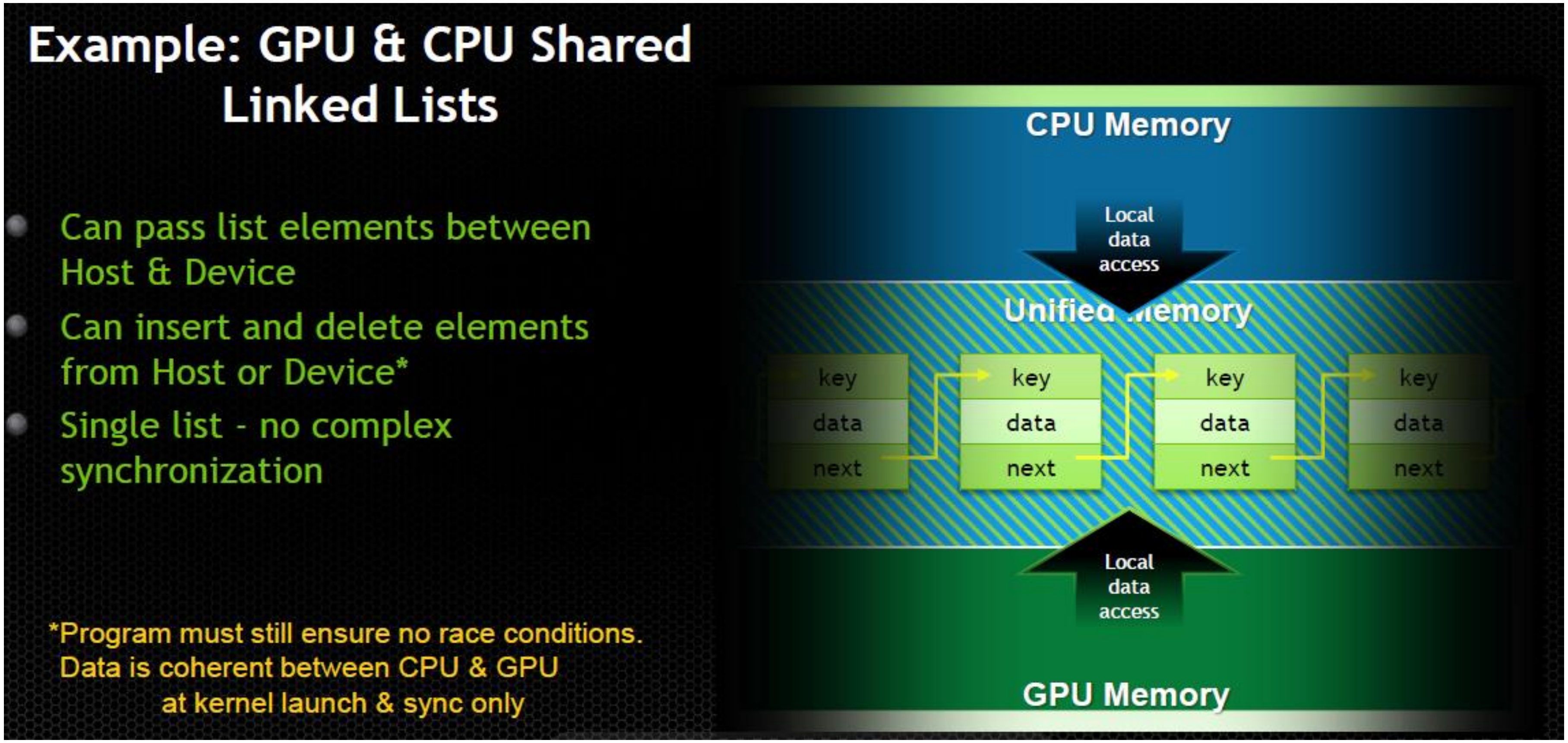
Unified Memory

- Eliminates the need for deep copying when using structs



```
void launch(dataElem *elem) {
    kernel<<< ... >>>(elem);
}
```

Unified Memory



Example: GPU & CPU Shared Linked Lists

- Can pass list elements between Host & Device
- Can insert and delete elements from Host or Device*
- Single list - no complex synchronization

*Program must still ensure no race conditions.
Data is coherent between CPU & GPU at kernel launch & sync only