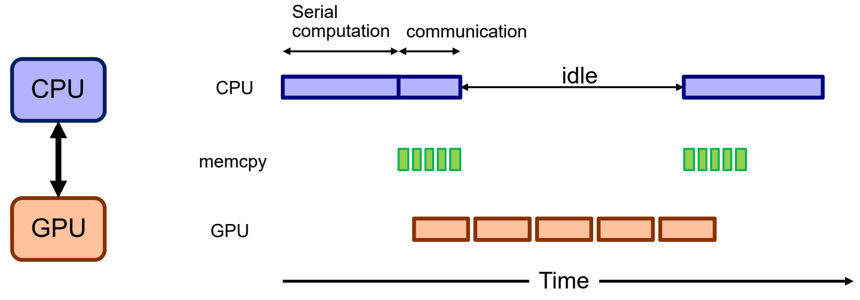
MIC-GPU:
High-Performance Computing for Medical Imaging on Programmable Graphics Hardware (GPUs)

Multi-GPU CUDA Programming

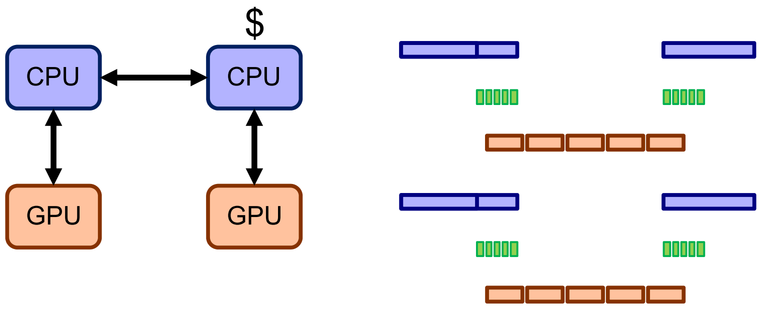
Klaus Mueller and Sungsoo Ha

Stony Brook University
Computer Science
Stony Brook, NY

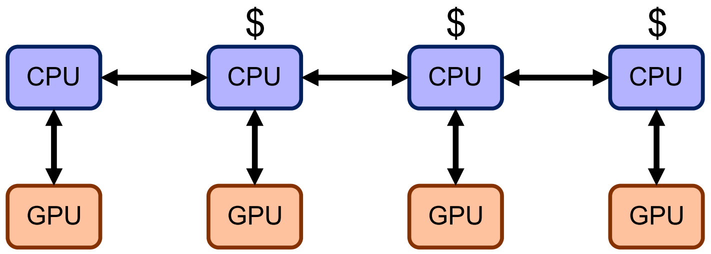
Move to multiple GPUs



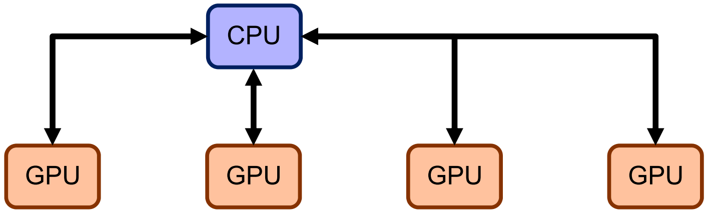
Move to multiple GPUs



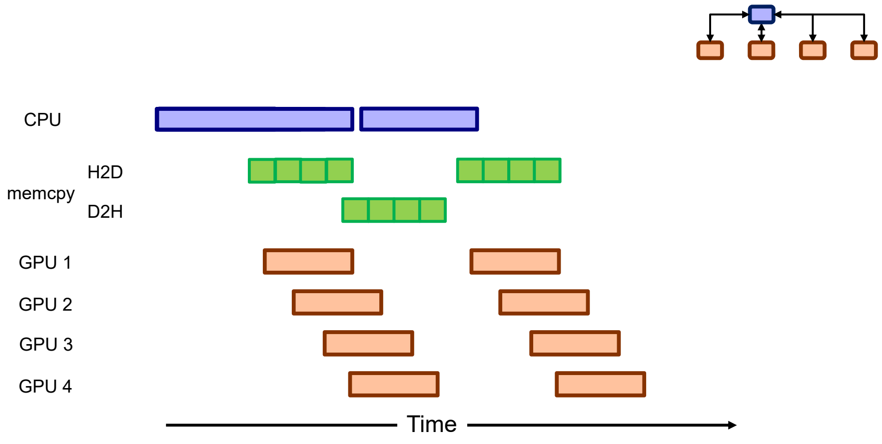
Move to multiple GPUs



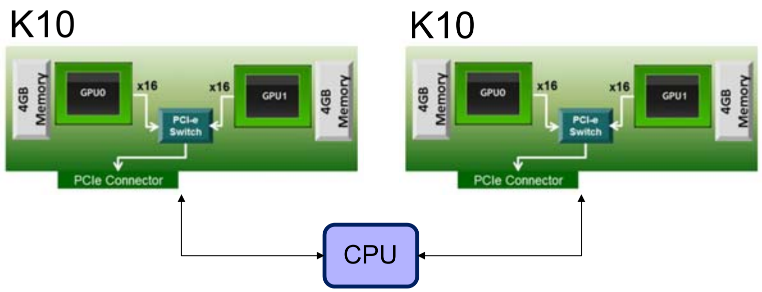
Move to multiple GPUs



Move to multiple GPUs



Example: Nvidia K10



- Number of processor cores: 1536 per GPU
- PCI Express Gen3 x16 system interface
- Total board memory: 8 GB (4 GB per GPU)
- More details in Board Specification of TESLA K10 GPU

Multi GPUs from a single CPU thread

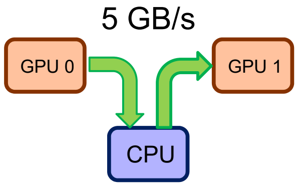
cudaSetDevice (int device) – sets the current GPU

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

Peer-to-peer memcopy

SPIE Medical Imaging

```
cudaMemcpyPeerAsync( void* dst_addr, int dst_dev,
                     void* src_addr, int src_dev,
                     size_t num_bytes, cudaStream_t stream)
```


5 GB/s

GPU 0

GPU 1

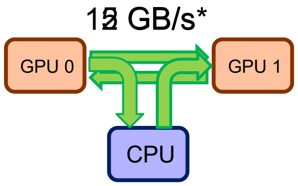
CPU

GPUDirect

SPIE Medical Imaging

```
cudaMemcpyPeerAsync( void* dst_addr, int dst_dev,
                     void* src_addr, int src_dev,
                     size_t num_bytes, cudaStream_t stream)
```

```
cudaDeviceEnablePeerAccess (peer_device, 0)
cudaDeviceCanAccessPeer (&accessible, dev_x, dev_y)
```


12 GB/s*

GPU 0

GPU 1

CPU

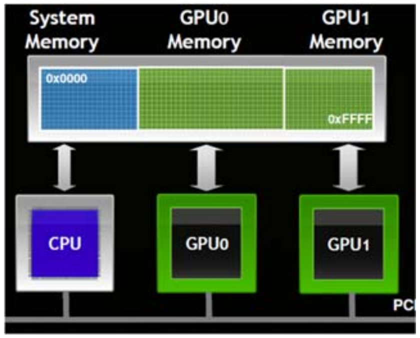
* PCI2 gen.2 (22 GB/s for gen. 3)

Unified Virtual Addressing (UVA)

SPIE Medical Imaging

GPU can determine from an address where data resides

- 64-bit Linux (or Windows) with TCC driver
- Fermi or later architecture GPUs (compute capability 2.0 or higher)
- CUDA 4.0 or later



Unified Virtual Addressing (UVA)

SPIE Medical Imaging

Peer-to-Peer (P2P) Communication

- Eliminates system memory allocation & copy overhead
- More convenient multi-GPU programming

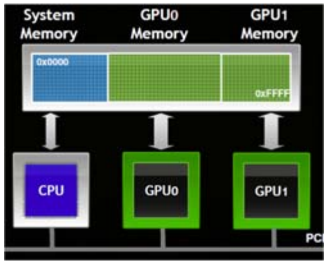
Unified Virtual Addressing (UVA)

Peer-to-Peer (P2P) Communication

- Eliminates system memory allocation & copy overhead
- More convenient multi-GPU programming

One address space for all CPU and GPU memory

- Determine physical memory location from pointer value
- Enables libraries to simplify their interfaces



Peer-to-peer memcopy

```
cudaMemcpyPeerAsync( void* dst_addr, int dst_dev,
                     void* src_addr, int src_dev,
                     size_t num_bytes, cudaStream_t stream)


cudaMemcpyAsync( void* dst_addr, int dst_dev,
                 void* src_addr, int src_dev,
                 size_t num_bytes, cudaStream_t stream,
                  cudaMemcpyDefault )
```

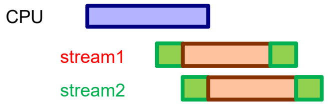
CUDA Streams and Events

Steam

- A sequence of operations that execute in issue-order on the GPU

```
cudaStream_t stream1, stream2
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaHostAlloc(&src, size, 0);               // pinned memory required on host
…
cudaMemcpyAsync(dst, src, size, H2D, stream1);
kernel<<<grid, block, 0, stream1>>>(…);
cudaMemcpyAsync(dst, src, size, H2D, stream2);
kernel<<<grid, block, 0, stream2>>>(…);
…
```

potentially overlapped



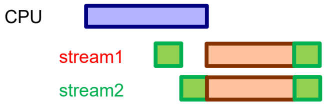
CUDA Streams and Events

Events

- Expressing dependency explicitly

```
cudaEvent_t ev;
cudaEventCreate(&ev);
…
cudaMemcpyAsync(dst, src, size, H2D, stream1);
cudaMemcpyAsync(dst, src, size, H2D, stream2);
cudaEventRecord(ev, stream2);               // Record an event for stream2

cudaStreamWaitEvent(stream1, ev);           // stream1 wait for the event to finish
kernel<<<grid, block, 0, stream1>>>(…);
kernel<<<grid, block, 0, stream2>>>(…);
…
```



Multi-GPUs Streams and Events

```
cudaStream_t streamA, streamB
cudaEvent_t eventA, eventB;

cudaSetDevice(0);                    // current device is 0
cudaStreamCreate(&streamA);
cudaEventCreate(&eventA);

cudaSetDevice(1);                    // current device is 1
cudaStreamCreate(&streamB);
cudaEventCreate(&eventB);

kernel<<<…,streamB>>>(…);            // run kernel with device 1
cudaEventRecord(eventB, streamB);

cudaEventSynchronize(eventB);        // CPU waits for finishing eventB
```

Multi-GPUs Streams and Events

```
cudaStream_t streamA, streamB
cudaEvent_t eventA, eventB;

cudaSetDevice(0);
cudaStreamCreate(&streamA);
cudaEventCreate(&eventA);

cudaSetDevice(1);
cudaStreamCreate(&streamB);
cudaEventCreate(&eventB);

kernel<<<…, streamA >>>(…);
cudaEventRecord(eventB, streamB);

cudaEventSynchronize(eventB);
```

ERROR:
- Device 1 is current
- streamA belongs to device 0

Multi-GPUs Streams and Events

```
cudaStream_t streamA, streamB
cudaEvent_t eventA, eventB;

cudaSetDevice(0);
cudaStreamCreate(&streamA);
cudaEventCreate(&eventA);

cudaSetDevice(1);
cudaStreamCreate(&streamB);
cudaEventCreate(&eventB);

kernel<<<…, streamB >>>(…);
cudaEventRecord(eventA, streamB);

cudaEventSynchronize(eventB);
```

ERROR:
- eventA belongs to device 0
- streamB belongs to device 1

Multi-GPUs Streams and Events

```
cudaStream_t streamA, streamB
cudaEvent_t eventA, eventB;

cudaSetDevice(0);
cudaStreamCreate(&streamA);
cudaEventCreate(&eventA);

cudaSetDevice(1);
cudaStreamCreate(&streamB);
cudaEventCreate(&eventB);

kernel<<<…, streamB >>>(…);
cudaEventRecord(eventB, streamB);

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

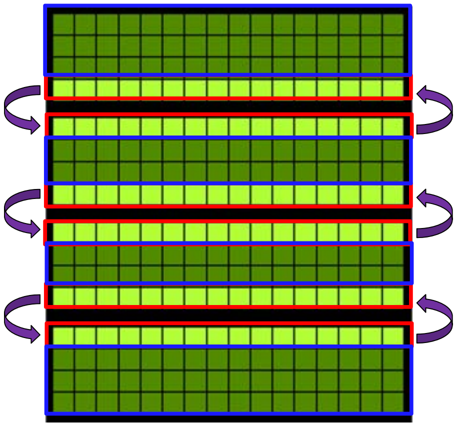
OK:
- Device 0 is current
- Synchronizing/querying events/streams of other devices is allowed

Multi-GPUs Streams and Events

The Rules

- CUDA streams and events are *per device* (GPU)
  - Each device has its own *default* stream (aka 0- or NULL-stream)
- Stream and:
  - **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
- Synchronization/query:
  - It is OK to query or synchronize with any event/stream

Multi-GPUs CUDA Program by example



1. Compute halo regions
2. Compute internal regions
3. Exchange halo regions

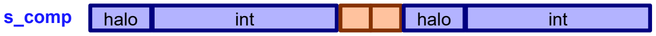
Multi-GPUs CUDA Program by example

```
for( int istep=0; istep<nsteps; istep++) {
  for(int i=0; i<num_gpus; i++) {
    cudaSetDevice(gpu[i]);
    kernel_halo<<<…, s_comp[i]>>>(…);
    kernel_int<<<…,s_comp[i]>>>(…);
  }
  for(int i=0; i<num_gpus-1; i++)
    cudaMemcpyPeerAsync(…, s_comp[i]);
  for(int i=1; i<num_gpus; i++)
    cudaMemcpyPeerAsync(…, s_comp[i]);
}
```

Compute halos

Compute internal

Exchange halos



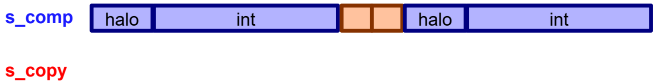
Multi-GPUs CUDA Program by example

```
for( int istep=0; istep<nsteps; istep++) {
  for(int i=0; i<num_gpus; i++) {
    cudaSetDevice(gpu[i]);
    kernel_halo<<<…, s_comp[i]>>>(…);
    kernel_int<<<…,s_comp[i]>>>(…);
  }
  for(int i=0; i<num_gpus-1; i++)
    cudaMemcpyPeerAsync(…, s_copy[i]);
  for(int i=1; i<num_gpus; i++)
    cudaMemcpyPeerAsync(…, s_copy[i]);
}
```

Compute halos

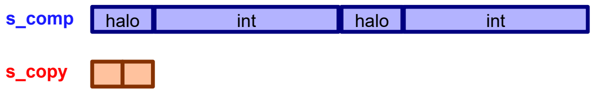
Compute internal

Exchange halos



Multi-GPUs CUDA Program by example

```
for( int istep=0; istep<nsteps; istep++) {
  for(int i=0; i<num_gpus; i++) {
    cudaSetDevice(gpu[i]);
    kernel_halo<<<…, s_comp[i]>>>(…);
    cudaEventRecord(ev[i], s_comp[i]);                // Record halo computation event
    kernel_int<<<…,s_comp[i]>>>(…);
  }
  for(int i=0; i<num_gpus-1; i++) {
    cudaStreamWaitEvent(s_copy[i], ev[i]);             // Wait exchange until the event is completed
    cudaMemcpyPeerAsync(…, s_copy[i]); }
  for(int i=1; i<num_gpus; i++)
    cudaMemcpyPeerAsync(…, s_copy[i]);
}
```



Multi-GPUs CUDA Program by example

```
for( int istep=0; istep<nsteps; istep++) {
  for(int i=0; i<num_gpus; i++) {
    cudaSetDevice(gpu[i]);
    kernel_halo<<<…, s_comp[i]>>>(…);
    cudaEventRecord(ev[i], s_comp[i]);
    kernel_int<<<…,s_comp[i]>>>(…);
  }
  for(int i=0; i<num_gpus-1; i++) {
    cudaStreamWaitEvent(s_copy[i], ev[i]);
    cudaMemcpyPeerAsync(…, s_copy[i]); }
  for(int i=1; i<num_gpus; i++)
    cudaMemcpyPeerAsync(…, s_copy[i]);
  for(int i=0; i<num_gpus; i++) {
    cudaSetDevice(gpu[i]);
    cudaDeviceSynchronize();
    // swap input/output pointer
  }
}
```

Compute halos

Compute internal

Exchange halos

Synchronize

Summary

- Multiple GPUs can stretch your compute dollar
- PeerToPeer and can move data directly between GPUs
- Streams and asynchronous kernel/copies facilitate concurrent execution