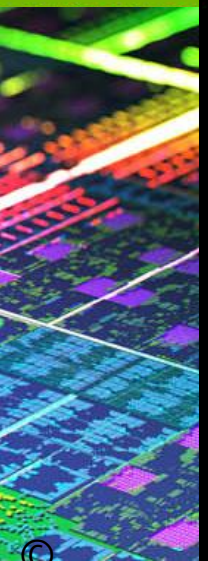
Multi-GPU Programming

Levi Barnes
Developer Technology, NVIDIA



Hands-on Multi-GPU training

S3529

Hands-on Lab: Multi-GPU Acceleration Example

Justin Luitjens

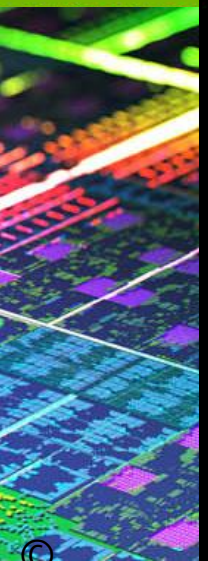
Wednesday 17:00-17:50

Rm 230A1

Where can I find multi-GPU nodes?

GPU Test Drive

http://www.nvidia.com/GPUTestDrive



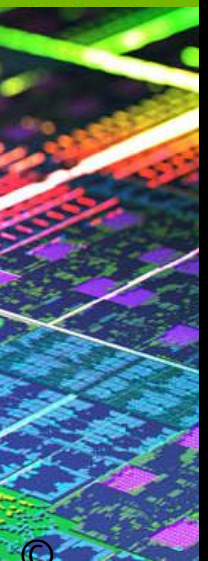
Where can I find multi-GPU nodes?

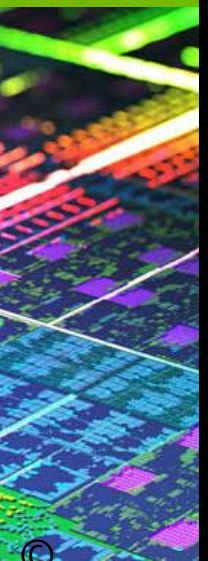
GPU Test Drive

http://www.nvidia.com/GPUTestDrive

Commercial/Academic Clusters

- Emerald (8 x M2090)
- Keeneland (3/2 x M2070)
- Tsubame (3 x M2050)
- JSCC RAS (8x M2090)
- Amazon Cloud (2 x C2050)

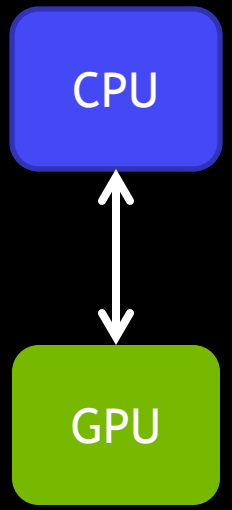




Outline

- Why More GPUs?
- Executing on multiple GPUs
- Hiding inter-GPU communication
- Tuning for node topology
- CUDA C Case study
  - One Host, Many GPUs
  - Many Host, Many GPUs

Move to multiple GPUs



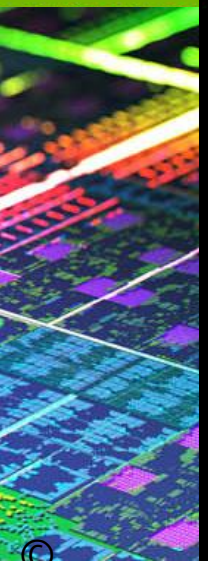
Move to multiple GPUs

CPU

MemCpy

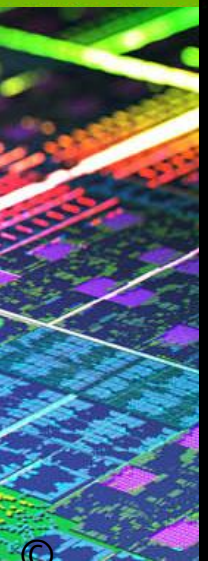
GPU

time

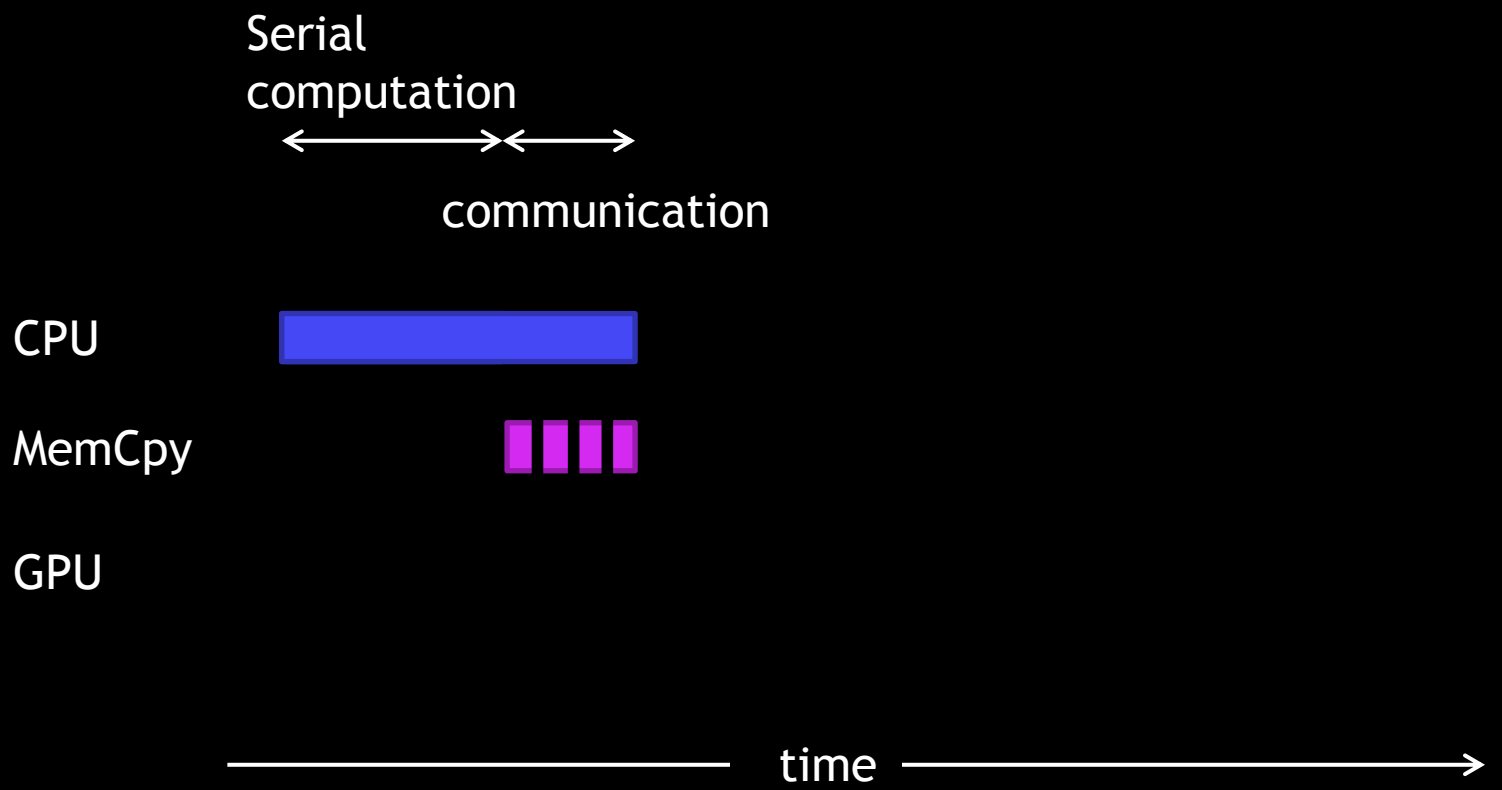


Move to multiple GPUs

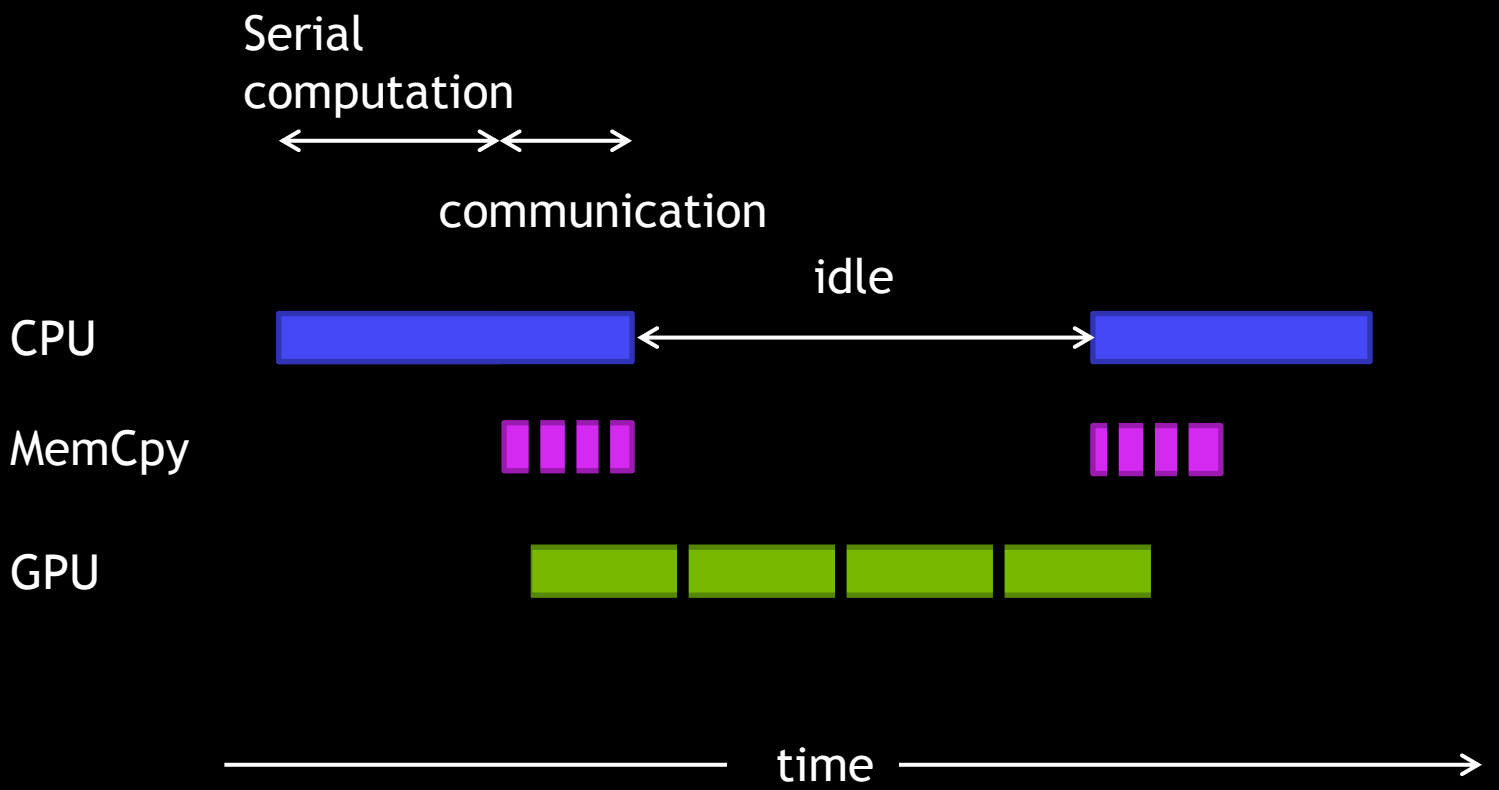
Serial computation

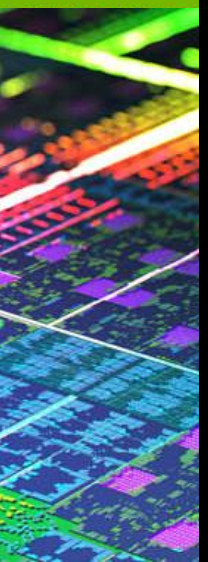


Move to multiple GPUs

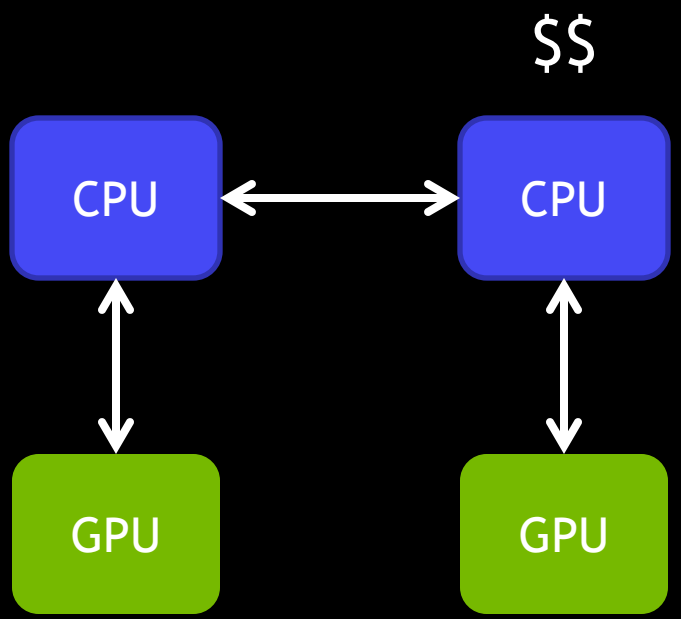


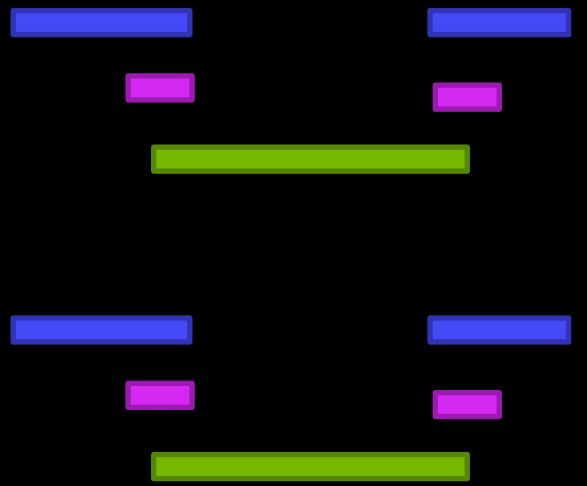
Move to multiple GPUs



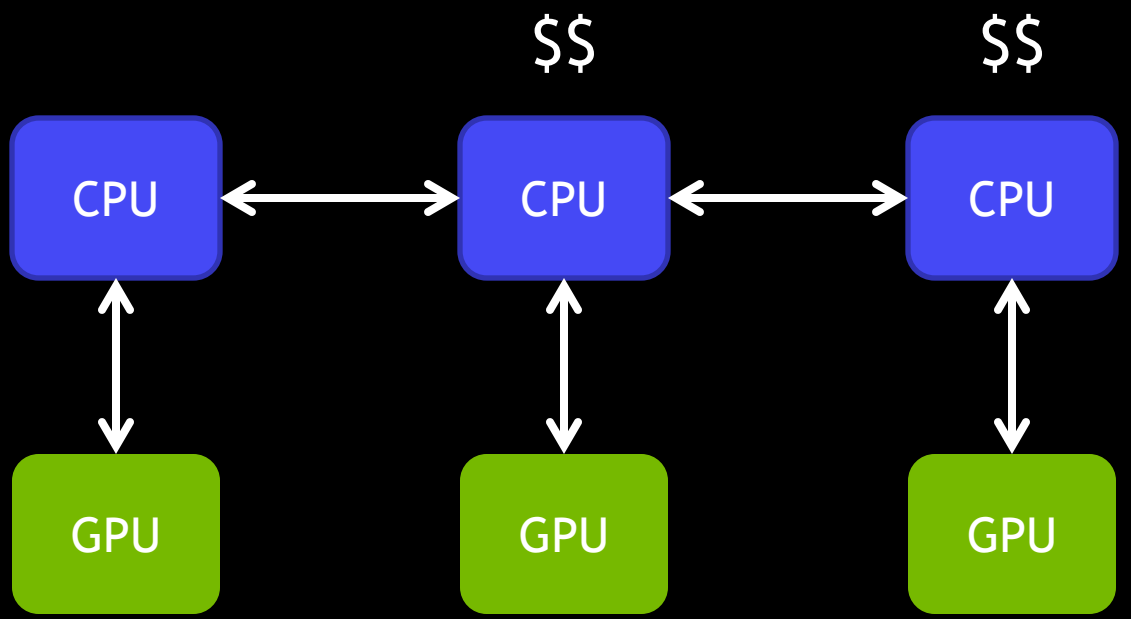


Move to multiple GPUs

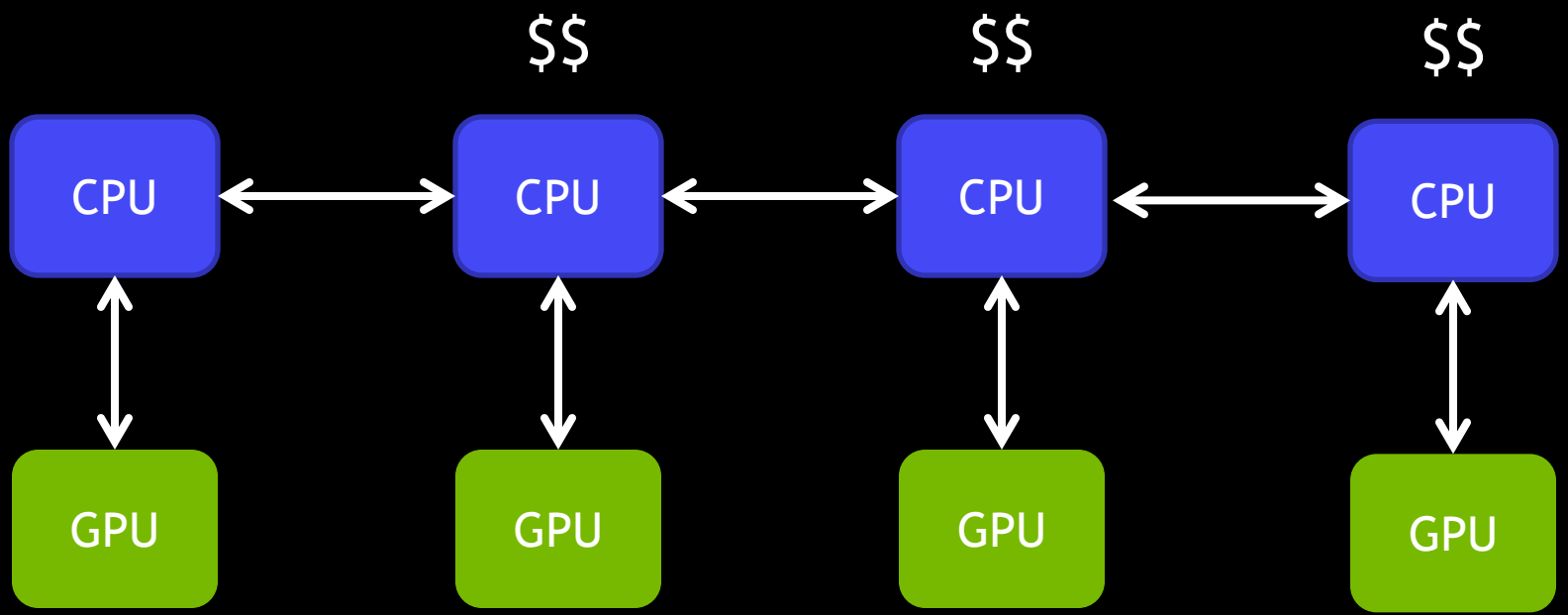




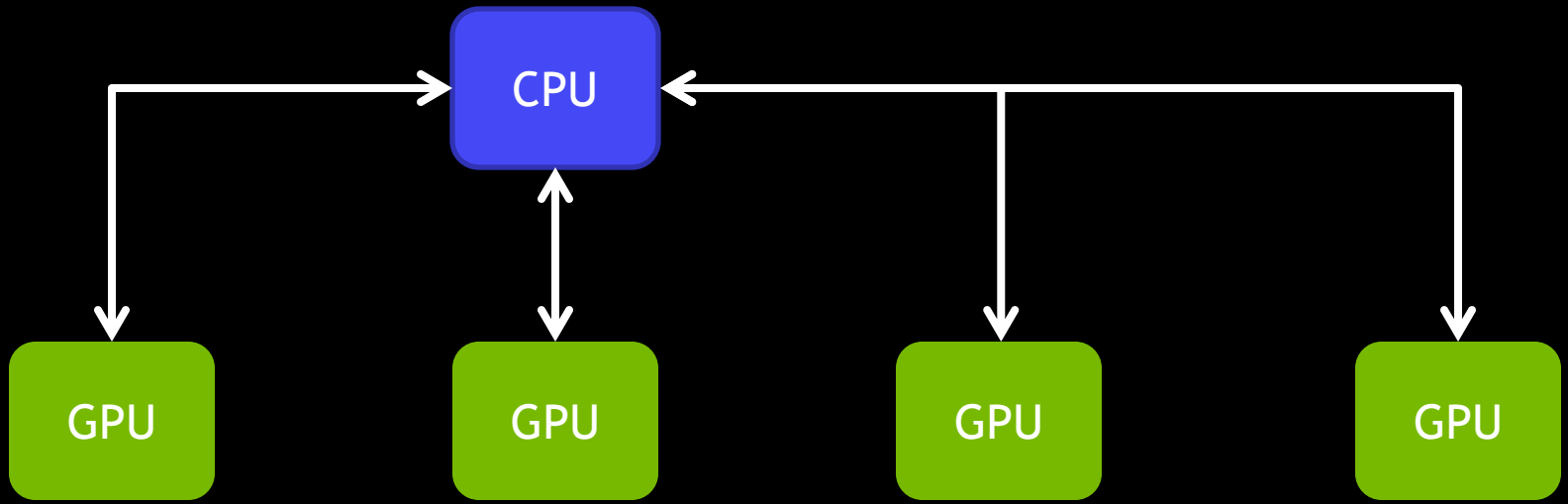
Move to multiple GPUs



Move to multiple GPUs

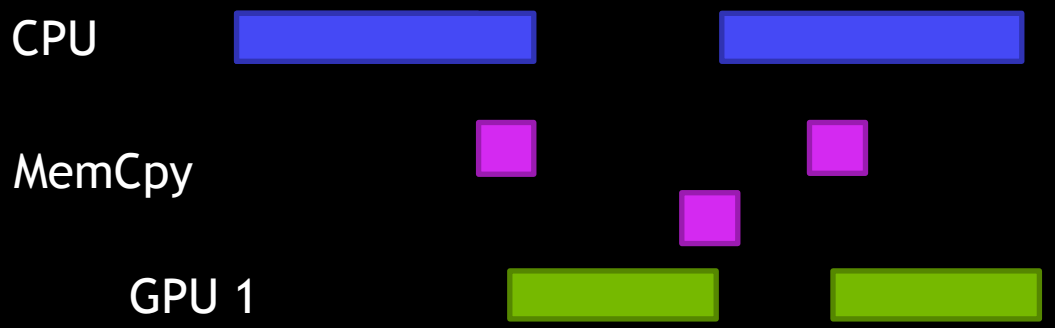


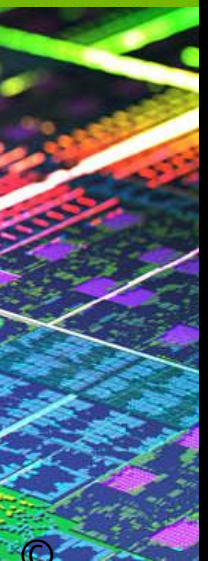
Move to multiple GPUs



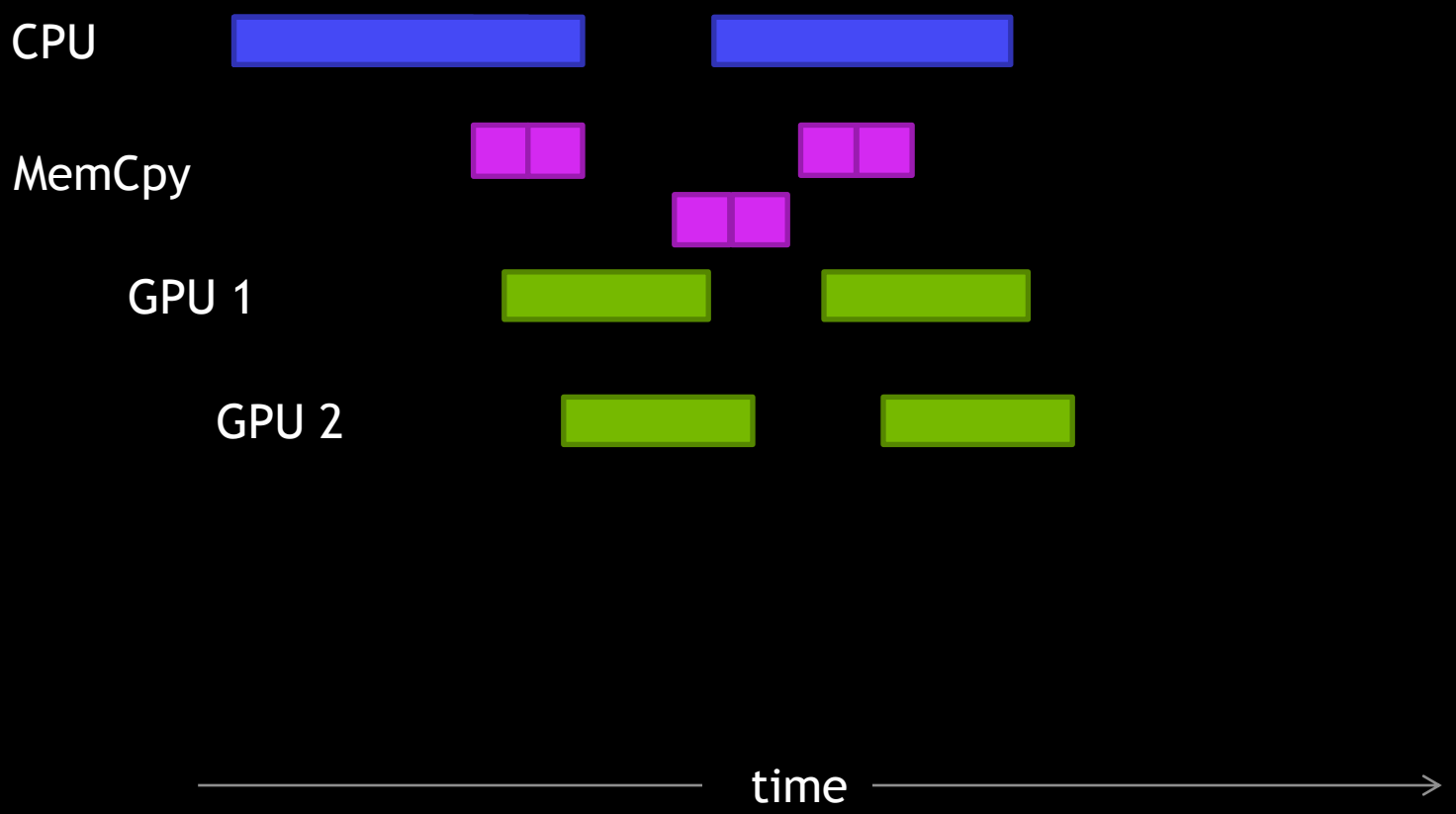
Move to multiple GPUs

Move to multiple GPUs

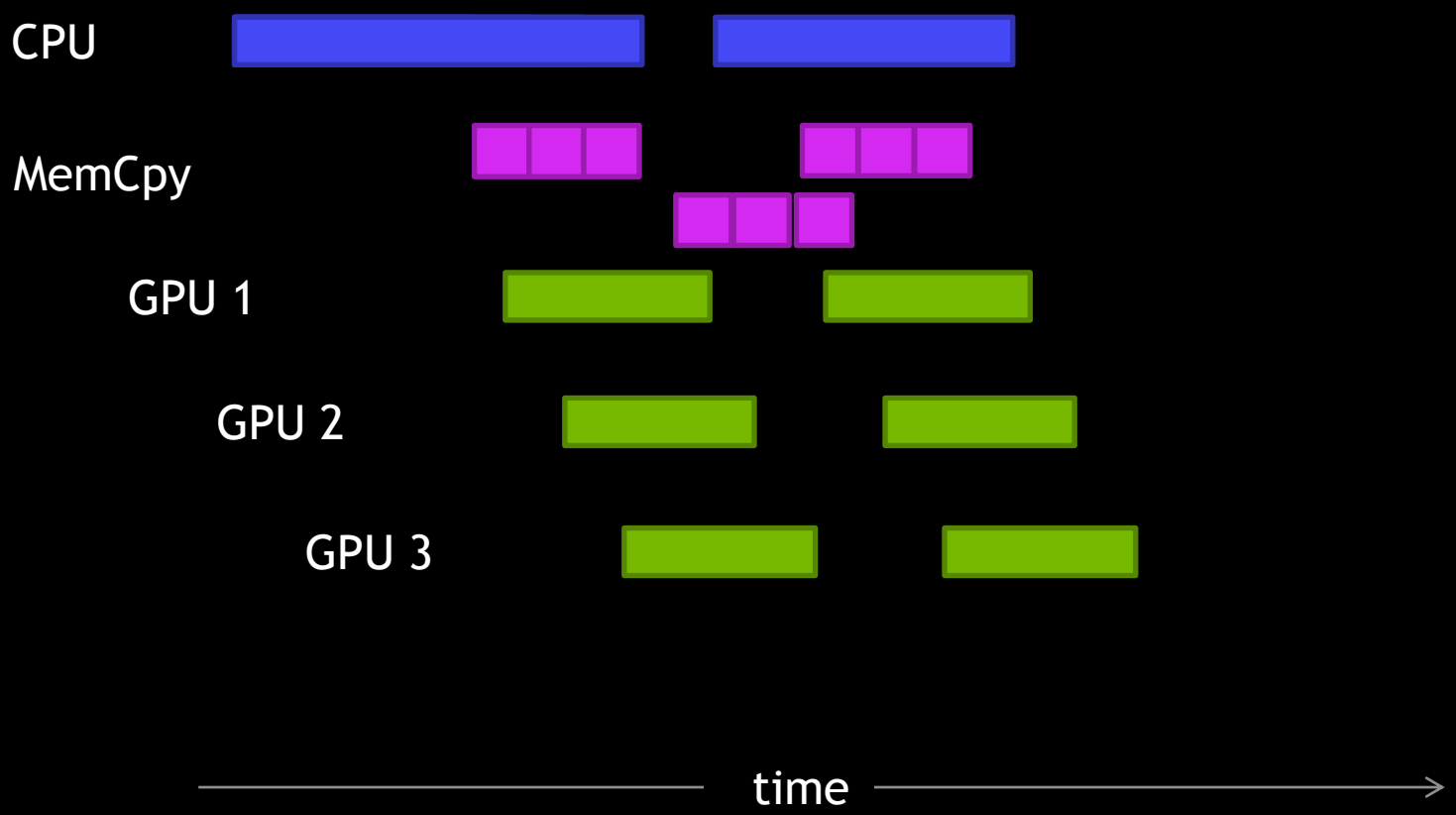




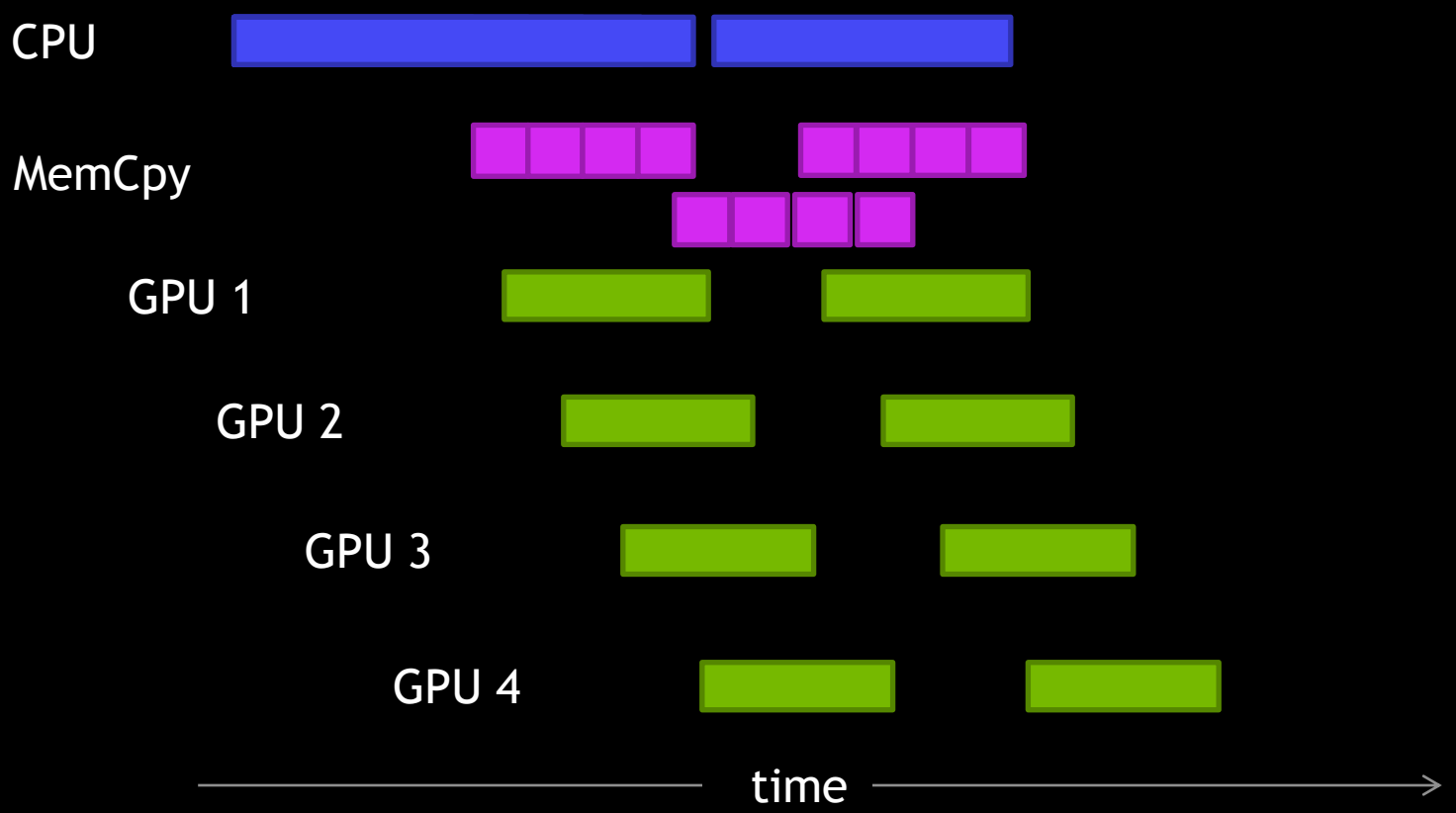
Move to multiple GPUs



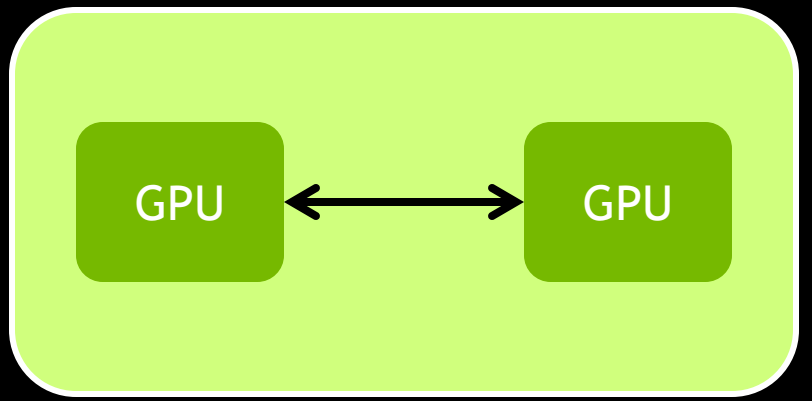
Move to multiple GPUs



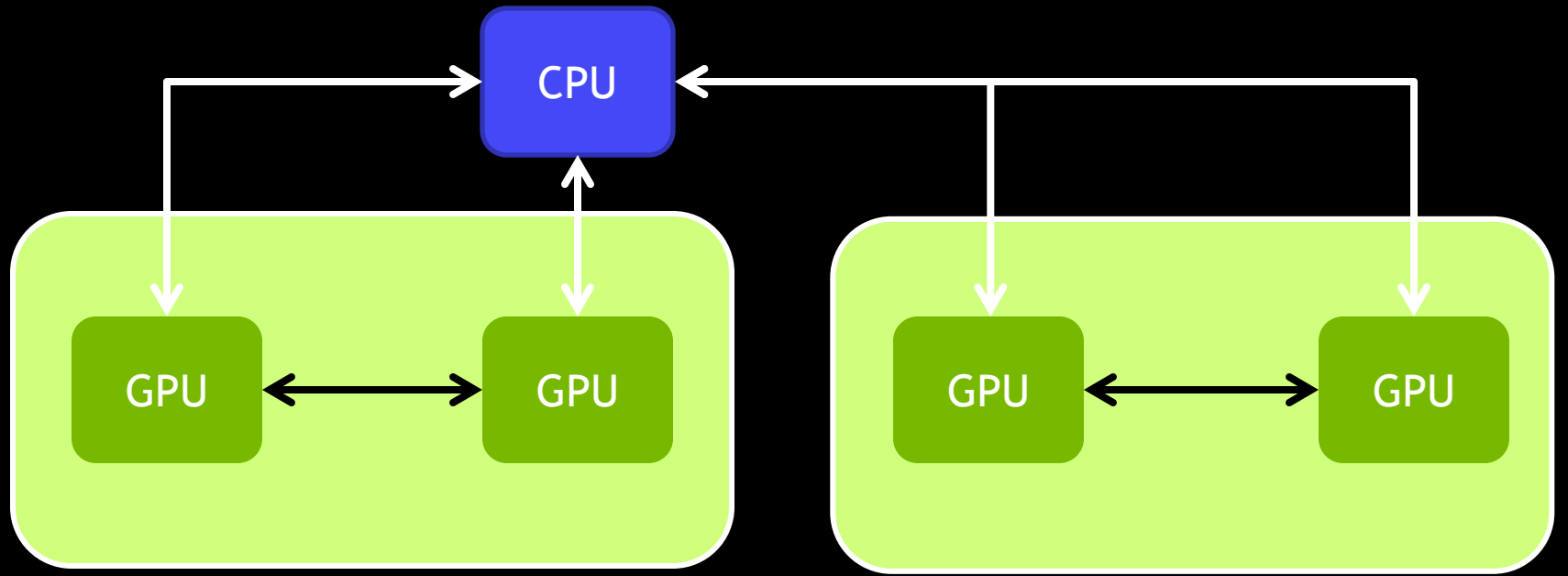
Move to multiple GPUs

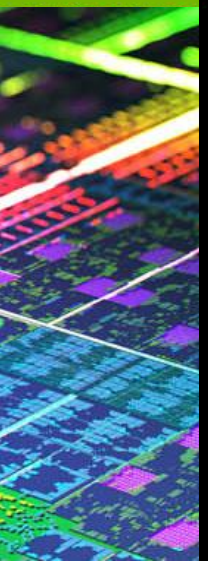


Nvidia K10 - multiple GPUs on a board

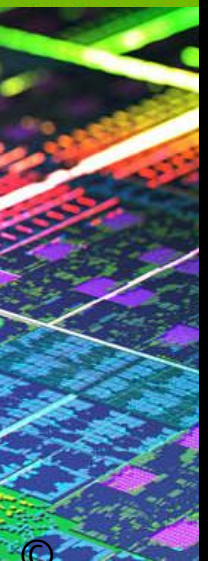


Nvidia K10 - multiple GPUs on a board





Executing on multiple GPUs



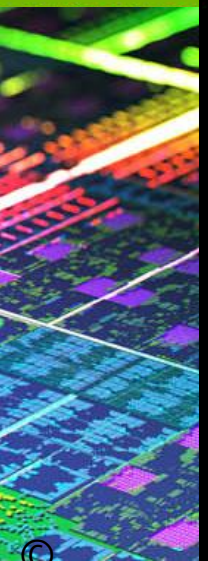
Managing multiple GPUs from a single CPU thread

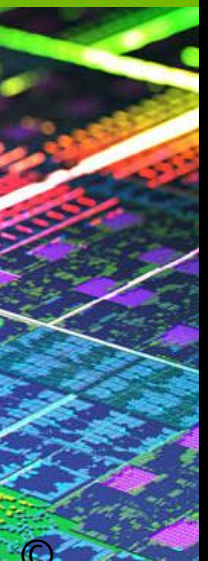
cudaSetDevice() - sets the current GPU

Managing multiple GPUs from a single CPU thread

cudaSetDevice() - sets the current GPU

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



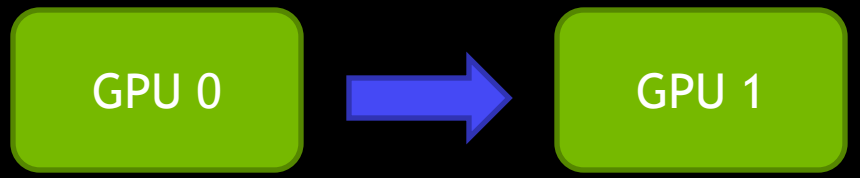


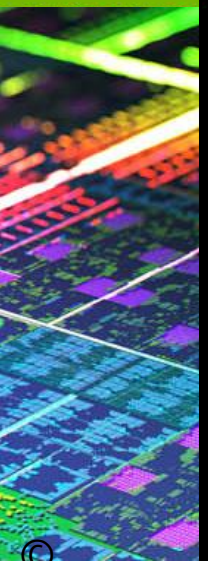
Peer-to-peer memcopy

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

Peer-to-peer memcopy

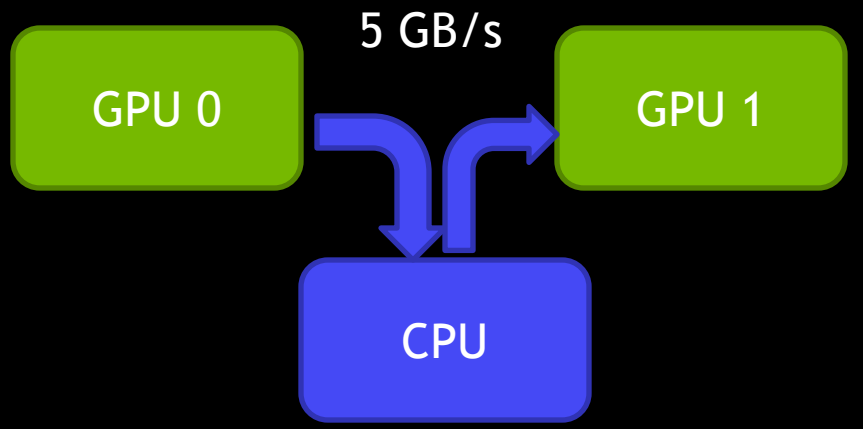
```
cudaMemcpyPeerAsync( void* dst_addr, int dst_dev,
                     void* src_addr, int src_dev,
                     size_t num_bytes, cudaStream_t stream )
```





Peer-to-peer memcopy

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



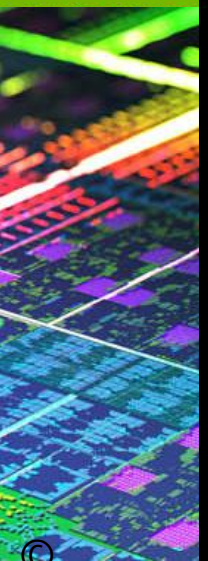
Enabling GPUDirect

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



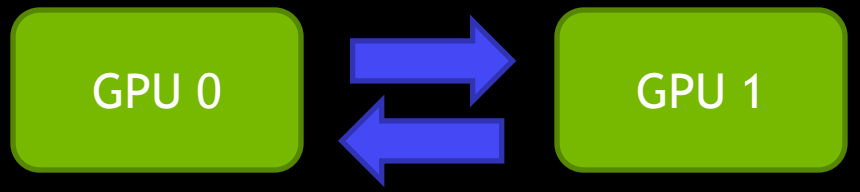
Peer-to-peer memcopy

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

* PCIe gen. 2 (12 GB/s for gen. 3)

Peer-to-peer memcopy

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

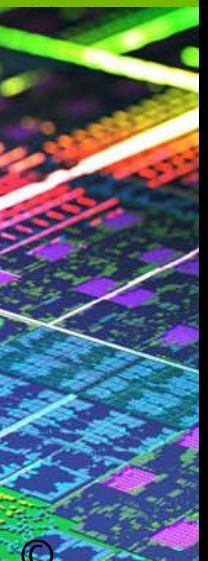


6 + 6 = 12 GB/s*

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

Unified Virtual Addressing (UVA)

Driver/GPU can determine from an address where data resides

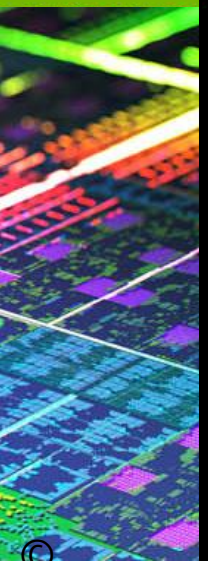


Unified Virtual Addressing (UVA)

Driver/GPU can determine from an address where data resides

Requires:

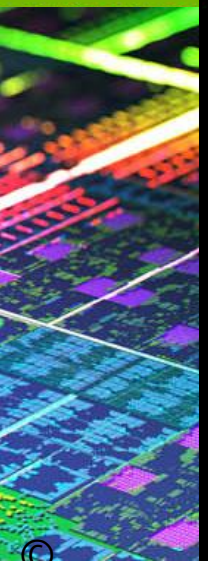
- 64-bit Linux or 64-bit Windows with TCC driver
- Fermi or later architecture GPUs (compute capability 2.0 or higher)
- CUDA 4.0 or later

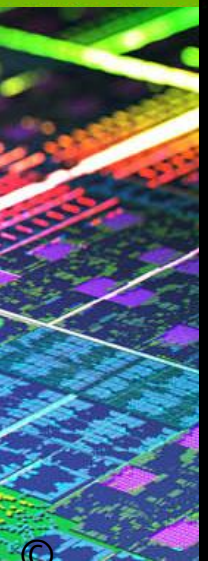


Peer-to-peer memcopy

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

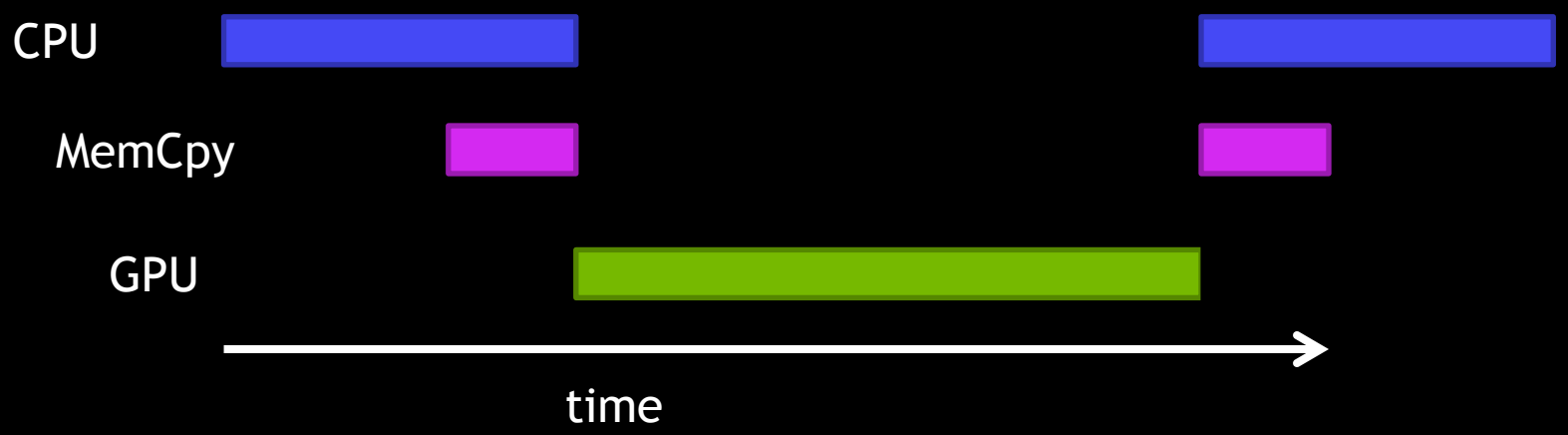
```
cudaMemcpyAsync( void* dst_addr, void* src_addr,
                 size_t num_bytes, cudaStream_t stream,
                 cudaMemcpyDefault )
```



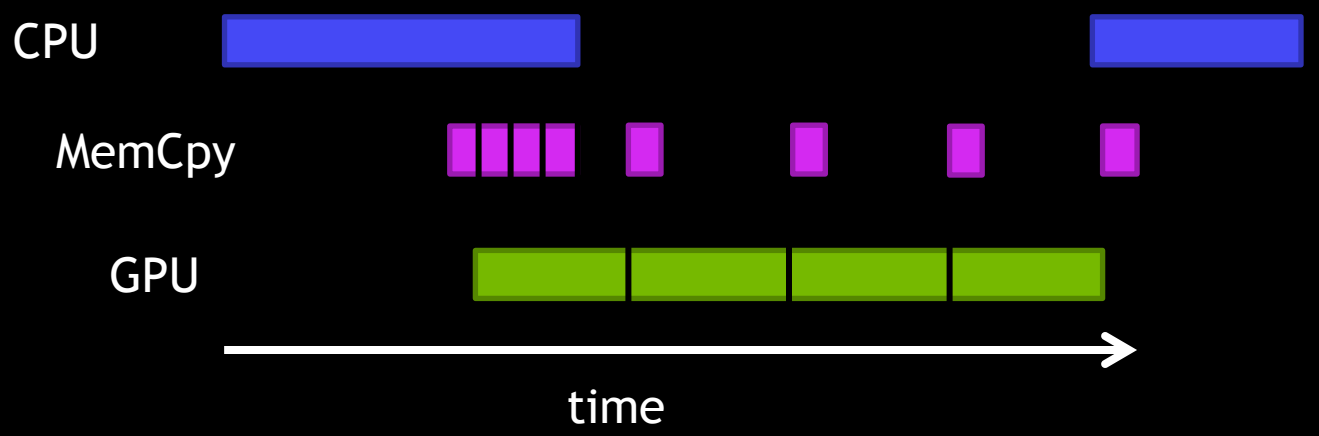


Hiding inter-GPU communication

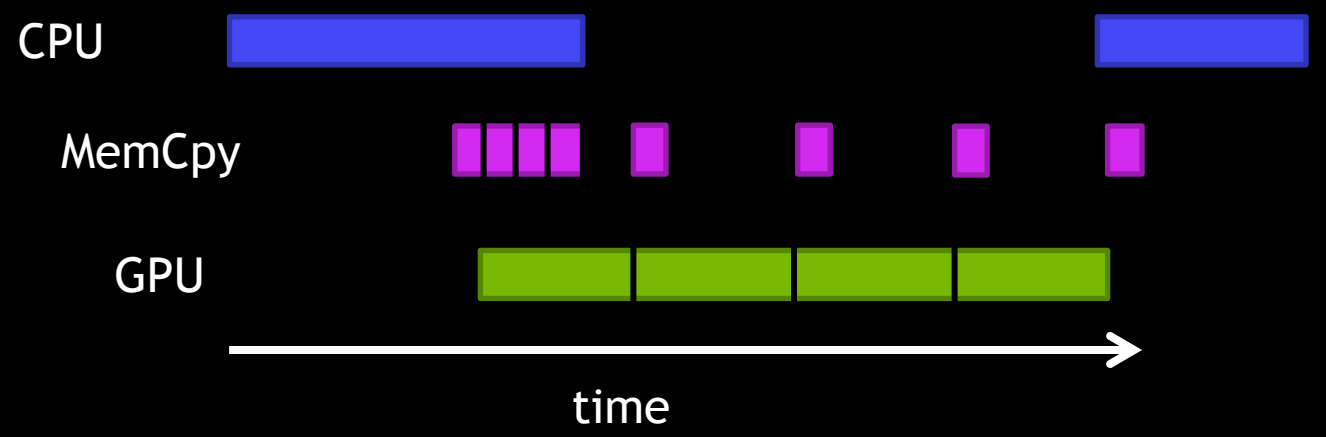
Overlap kernel and memory copy



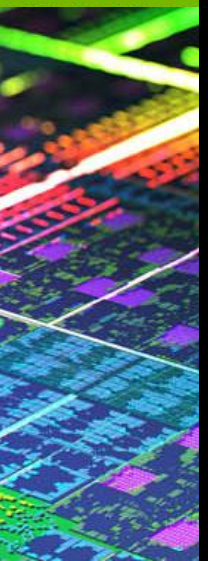
Overlap kernel and memory copy



Overlap kernel and memory copy



- **Requirements:**
  - D2H or H2D memcopy from pinned memory
  - Device with compute capability ≥ 1.1 (G84 and later)
  - Kernel and memcopy in different, non-0 streams

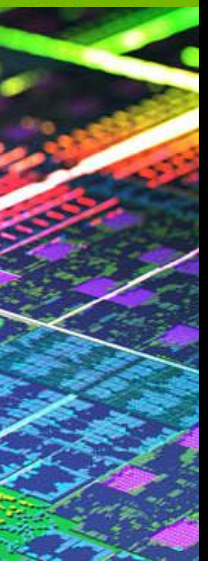


Pinned memory

cudaHostAlloc(void** pHost, size_t size, int flags)

cudaFree(void * pHost)

allocates pinned (page-locked) CPU memory



Pinned memory

cudaHostAlloc(void** pHost, size_t size, int flags)

cudaFree(void * pHost)

allocates pinned (page-locked) CPU memory

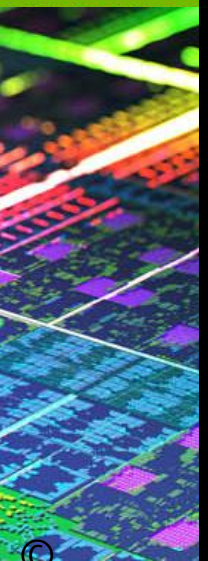
cudaHostRegister(void* pHost, size_t size, int flags)

cudaHostUnregister(void* pHost)

pins/unpins previously allocated memory

Overlap kernel and memory copy

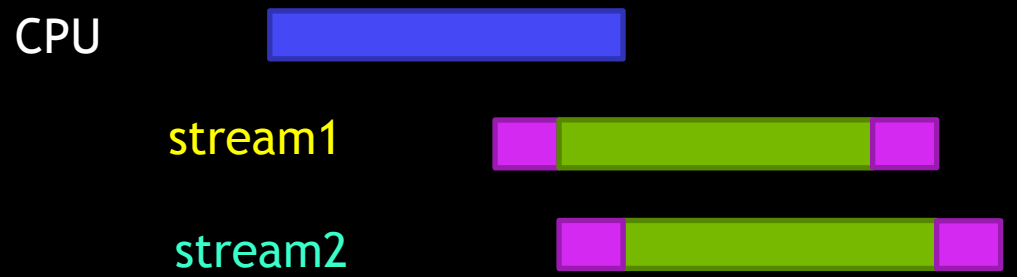
```
cudaStream_t   stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaHostAlloc(&src, size, 0);
…
cudaMemcpyAsync( dst, src, size, dir, stream1 );
kernel<<<grid, block, 0, stream1>>>(…);
cudaMemcpyAsync( dst, src, size, dir, stream2 );
kernel<<<grid, block, 0, stream2>>>(…);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

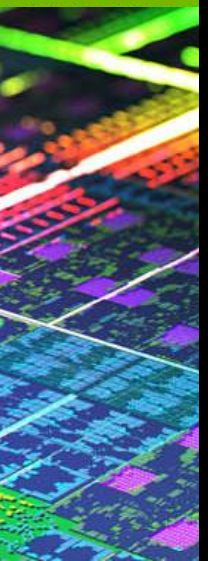


Overlap kernel and memory copy

```
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>>>(…);
```

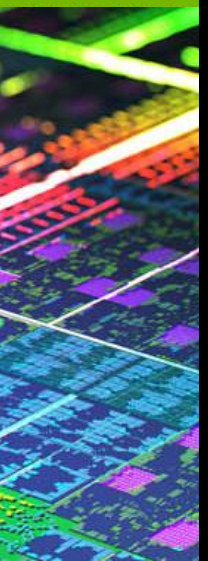
} potentially overlapped





Creating CUDA Events

```
cudaEventCreate(cudaEvent_t *event)
cudaEventDestroy(cudaEvent_t event)

cudaEventRecord(cudaEvent_t event,
                cudaStream_t stream=0)
```

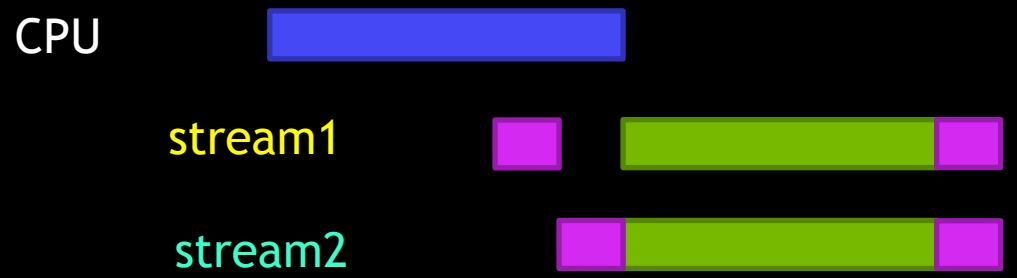


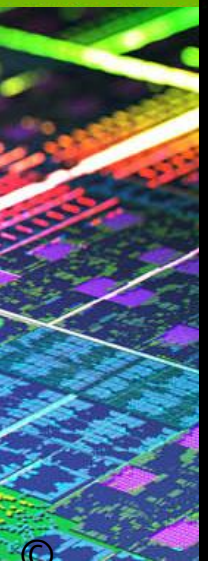
Using CUDA Events

```
cudaEventSynchronize(cudaEvent_t event)

cudaStreamWaitEvent(cudaStream_t stream,
                    cudaEvent_t event)
```

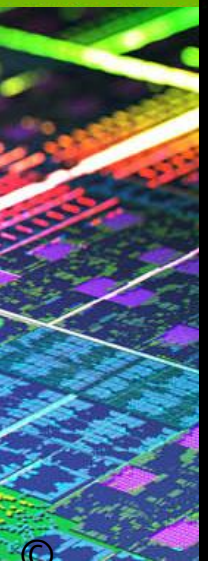
Expressing dependency through events

```
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

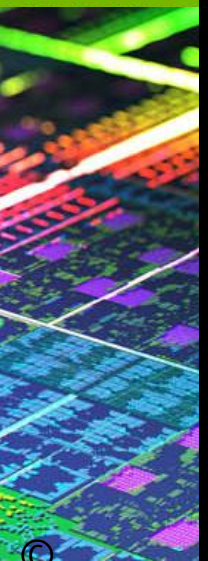


The Rules

- CUDA streams and events are *per device* (GPU)
  - Each device has its own *default* stream (aka 0- or NULL-stream)

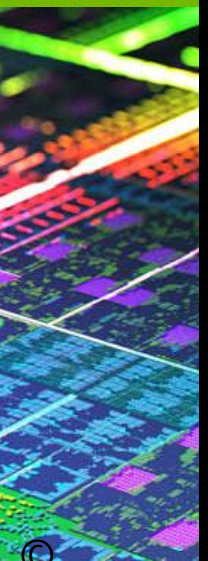
The Rules

- CUDA streams and events are *per device* (GPU)
  - Each device has its own *default* stream (aka 0- or NULL-stream)
- Streams and:
  - **Kernels:** can be launched to a stream only if the stream's GPU is current



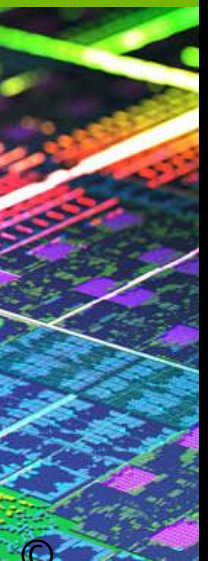
The Rules

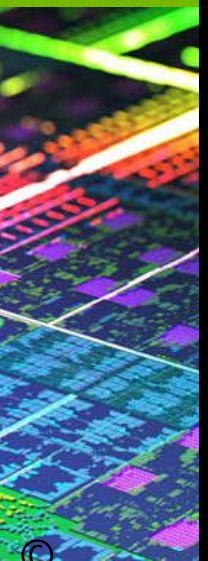
- CUDA streams and events are *per device* (GPU)
  - Each device has its own *default* stream (aka 0- or NULL-stream)
- Streams and:
  - **Kernels:** can be launched to a stream only if the stream's GPU is current
  - **Memcopies:** can be issued to any stream



The Rules

- CUDA streams and events are *per device* (GPU)
  - Each device has its own *default* stream (aka 0- or NULL-stream)
- Streams 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





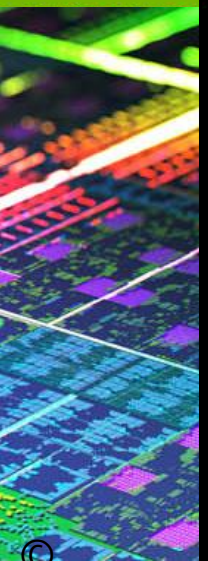
The Rules

- CUDA streams and events are *per device* (GPU)
  - Each device has its own *default* stream (aka 0- or NULL-stream)
- Streams 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

Example 1

```
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 );

cudaEventSynchronize( eventB );
```

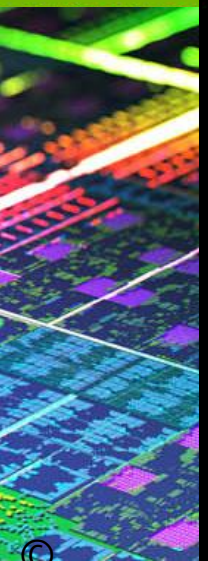
OK:
- device 1 is current
- eventB and streamB belong to device 1



Example 2

```
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 );
```

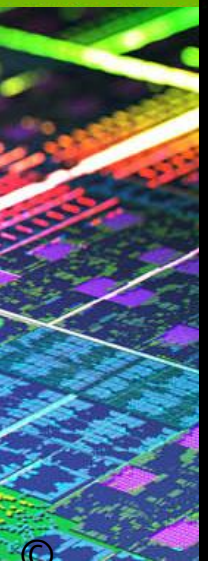
ERROR:
- device 1 is current
- streamA belongs to device 0



Example 3

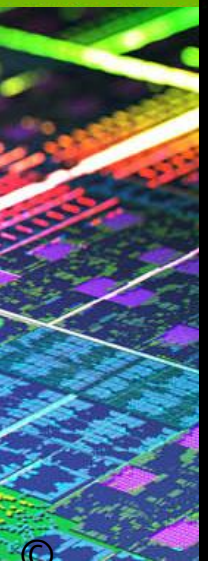
```
cudaStream_t streamA, streamB;
cudaEvent_t eventA, eventB;

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( eventA, streamB );

cudaEventSynchronize( eventB );
```

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



Example 4

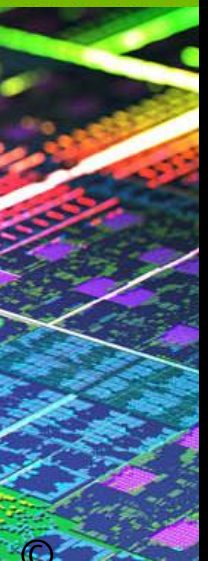
```
cudaStream_t streamA, streamB;
cudaEvent_t eventA, eventB;

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>>>(...);                 } device-1 is current
cudaEventRecord( eventB, streamB );

cudaSetDevice( 0 );
cudaEventSynchronize( eventB );                } device-0 is current
kernel<<<..., streamA>>>(...);
```



Example 4

```
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>>>(...);
```

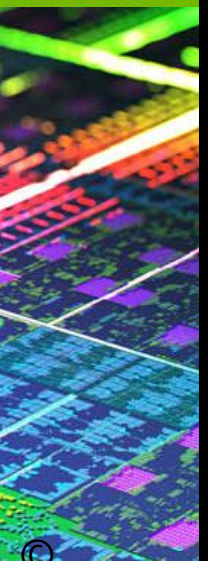
OK:
- device-0 is current
- synchronizing/querying events/streams of other devices is allowed

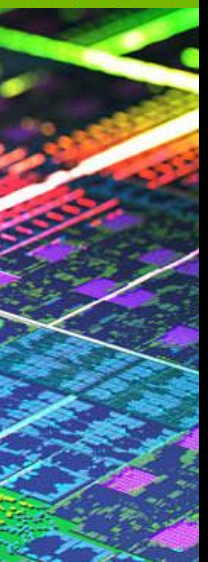


Example 5

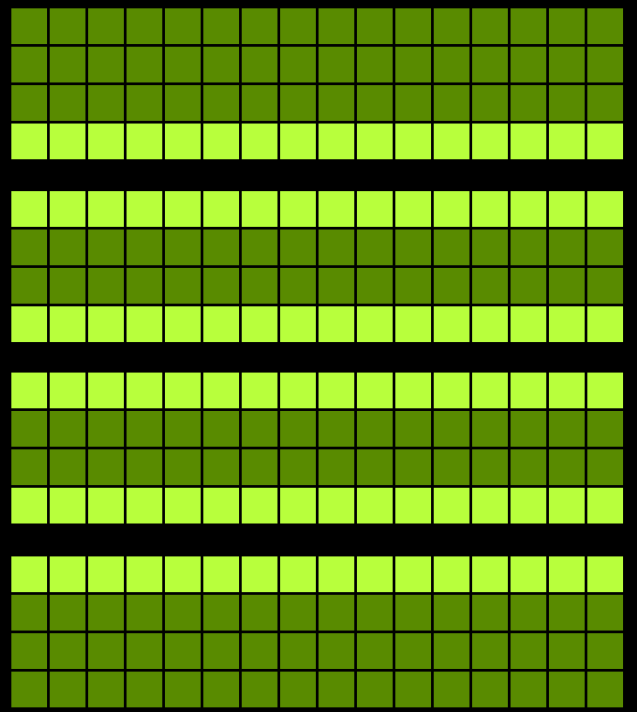
```
int gpu_A = 0;
int gpu_B = 1;

cudaSetDevice( gpu_A );
cudaMalloc( &d_A, num_bytes );

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

Even though kernel executes on gpu2, it will access (via PCIe) memory allocated on gpu1

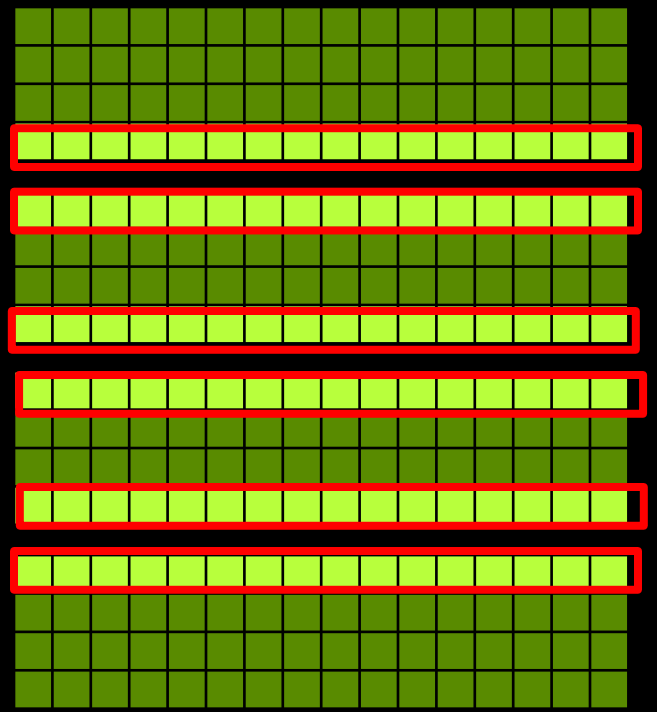




Subdividing the problem

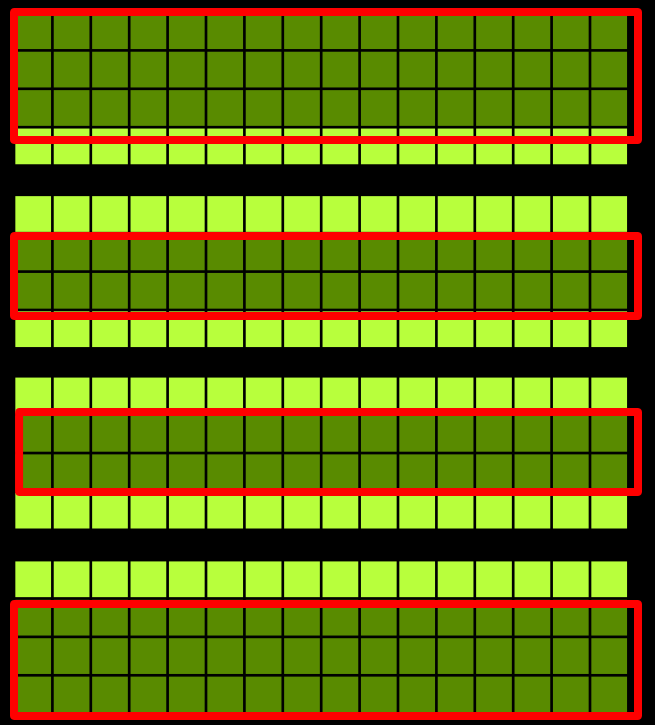


Stage 1



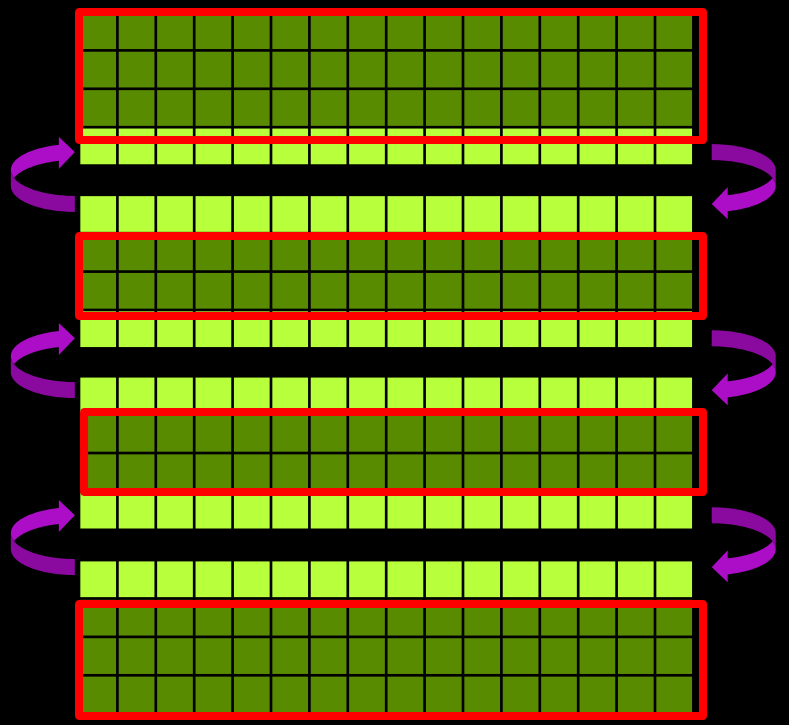
Compute halo regions

Stage 2



Compute internal regions

Stage 2

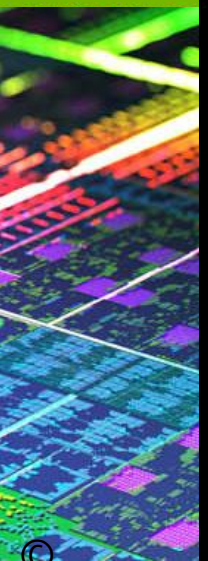


Compute internal regions
Exchange halo regions

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]>>>( ... );
    }
}
```

Compute halos

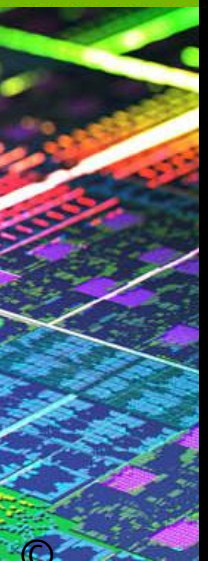


Code Pattern

```
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]>>>( ... );
    }

}
```

Compute halos

Compute internal



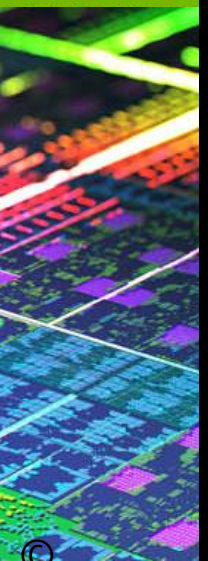
Code Pattern

```
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] );

}
```

Compute halos

Compute internal

Exchange halos



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] );

}
```

Compute halos

Compute internal

Exchange halos

stream_compute

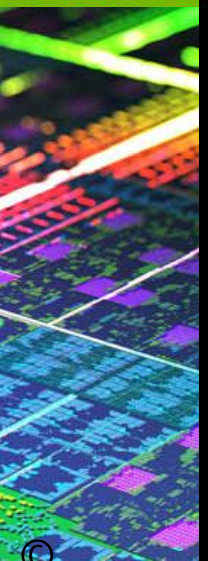
Code Pattern

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

}
```

Compute halos

Compute internal

Exchange halos



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

}
```

Compute halos

Compute internal

Exchange halos

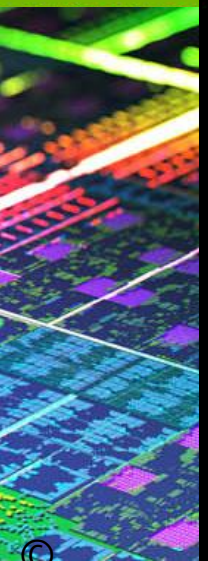
Code Pattern

```
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] );

}
```

Compute halos

Compute internal

Exchange halos



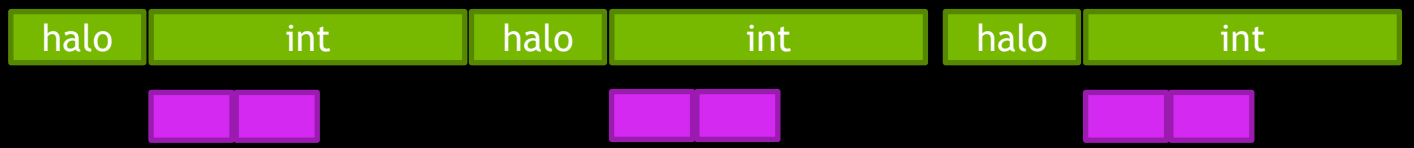
Code Pattern

```
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] );

}
```

Compute halos

Compute internal

Exchange halos



Code Pattern

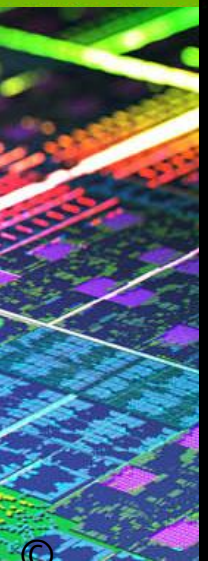
```
for( int istep=0; istep<nsteps; istep++)
{
    for( int i=0; i<num_gpus; i++ )
    {
        cudaSetDevice( gpu[i] );
        kernel_halo<<<..., stream_compute[i]>>>( ... );
        cudaEventRecord(event_i[i], stream_compute[i] );
        kernel_int<<<..., stream_compute[i]>>>( ... );
    }
    for(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

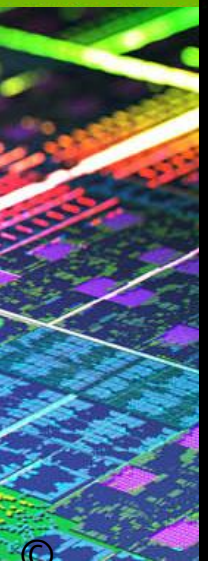
Exchange halos

Synchronize



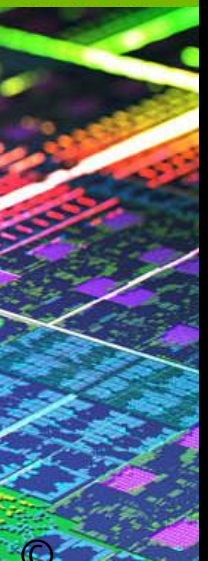
OpenACC in 1 slide

```
for (int i=1; i< SIZE+1; i++) {
    out[i] = 0.5*in[i] +
                0.25*in[i-1] +
                0.25*in[i+1];
}
```



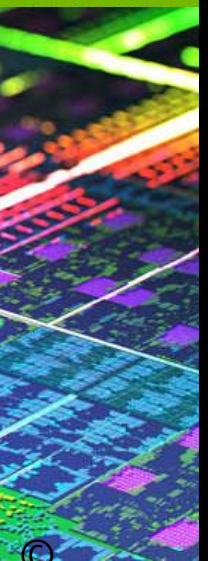
OpenACC in 1 slide

```
#pragma acc data copyin(in[0:SIZE+2])
#pragma acc kernels loop present(out, in)
for (int i=1; i< SIZE+1; i++) {
    out[i] = 0.5*in[i] +
                0.25*in[i-1] +
                0.25*in[i+1];
}
#pragma end kernels
#pragma acc update host(out)
```



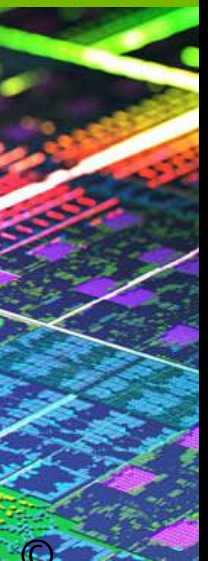
Multiple GPUs with OpenACC

```
#pragma acc data copyin(in)
#pragma acc kernels loop present(in)
for (int i=1; i< 1+SIZE; i++) {
    out[i] = 0.5*in[i] +
                0.25*in[i-1] +
                0.25*in[i+1];
}
#pragma end kernels
#pragma acc update host(out)
```



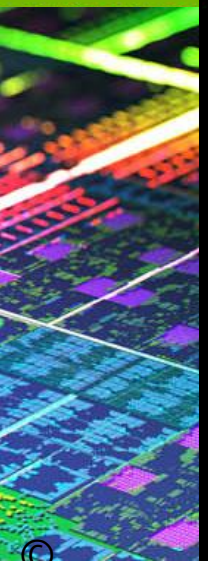
Multiple GPUs with OpenACC

```
#pragma omp parallel num_threads(4)
{
   int t = omp_get_thread_num();
   acc_set_device_num(t, acc_device_nvidia);
#pragma acc data copyin(in)
#pragma acc kernels loop present(in)
for (int i=1; i< 1+SIZE; i++) {
    out[i] = 0.5*in[i] +
                0.25*in[i-1] +
                0.25*in[i+1];
}
#pragma end kernels
#pragma acc update host(out)
}
//compile with -mp -acc
```



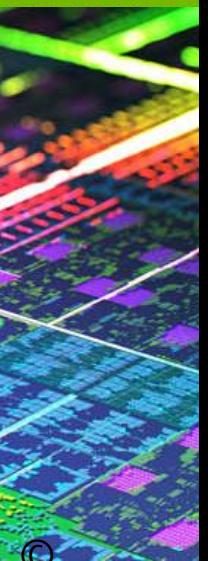
Multiple GPUs with OpenACC

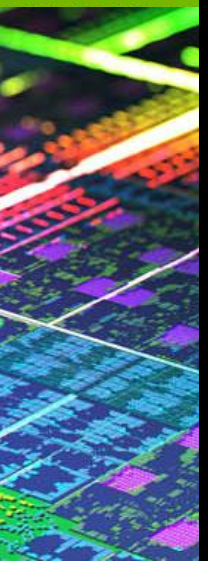
```
#pragma omp parallel num_threads(4)
{
   int t = omp_get_thread_num();
   acc_set_device_num(t, acc_device_nvidia);
#pragma acc data copyin(in)
#pragma acc kernels loop present(in)
for (int i=1+t*SIZE/4; i< 1+(t+1)*SIZE/4; i++)
    out[i] = 0.5*in[i] +
                0.25*in[i-1] +
                0.25*in[i+1];
}
#pragma end kernels
#pragma acc update host(out)
}
//compile with -mp -acc
```



Multiple GPUs with OpenACC

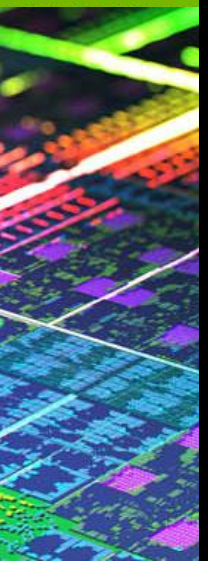
```
#pragma omp parallel num_threads(4)
{
   int t = omp_get_thread_num();
   acc_set_device_num(t, acc_device_nvidia);
#pragma acc data copyin(in[t*SIZE/4:SIZE/4+2])
#pragma acc kernels loop present(in[t*SIZE/4:SIZE/4+2])
for (int i=1+t*SIZE/4; i< 1+(t+1)*SIZE/4; i++)
    out[i] = 0.5*in[i] +
                0.25*in[i-1] +
                0.25*in[i+1];
}
#pragma end kernels
#pragma acc update host(out[t*SIZE/4+1:SIZE/4])
}
//compile with -mp -acc
```

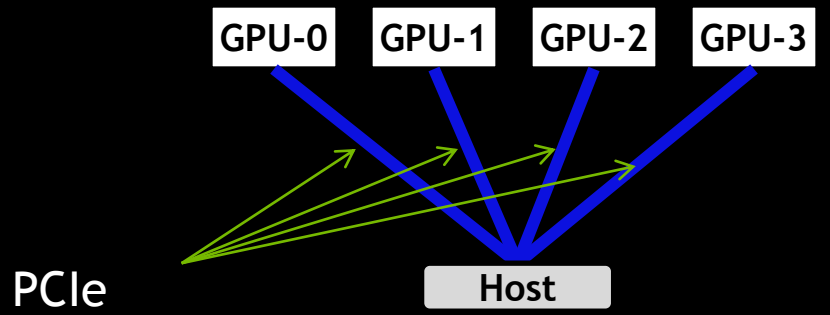




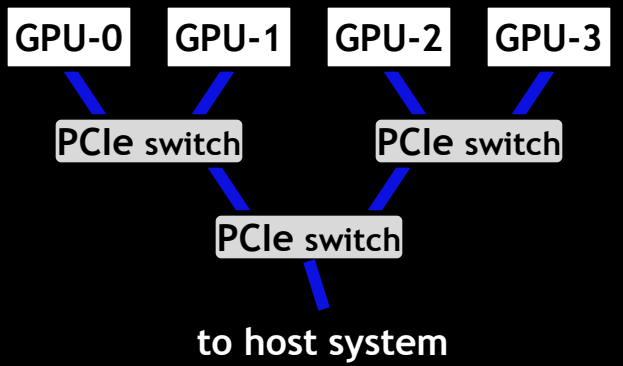
Tuning for your topology

CPU/GPU interface

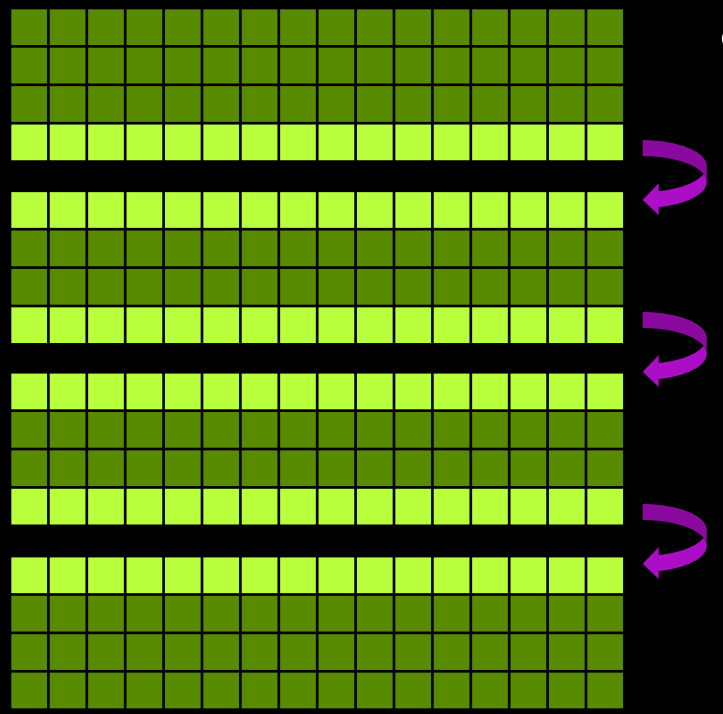




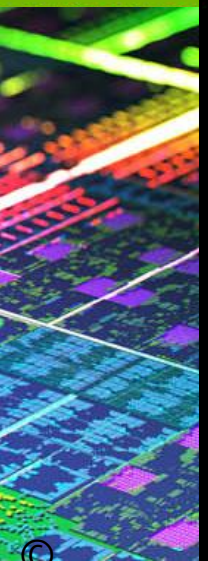
Example: 4-GPU Topology



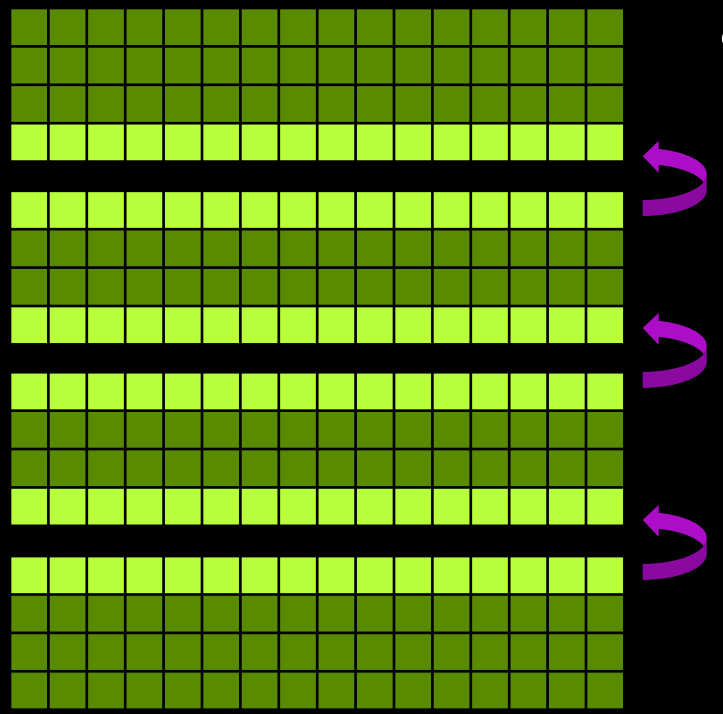
Example: 4-GPU Topology



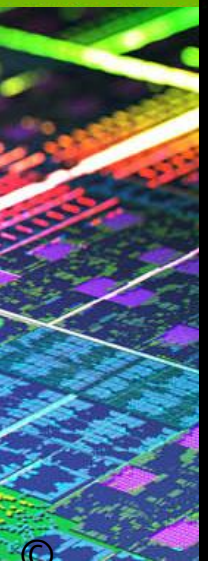
- Two ways to exchange
  - Left-right approach



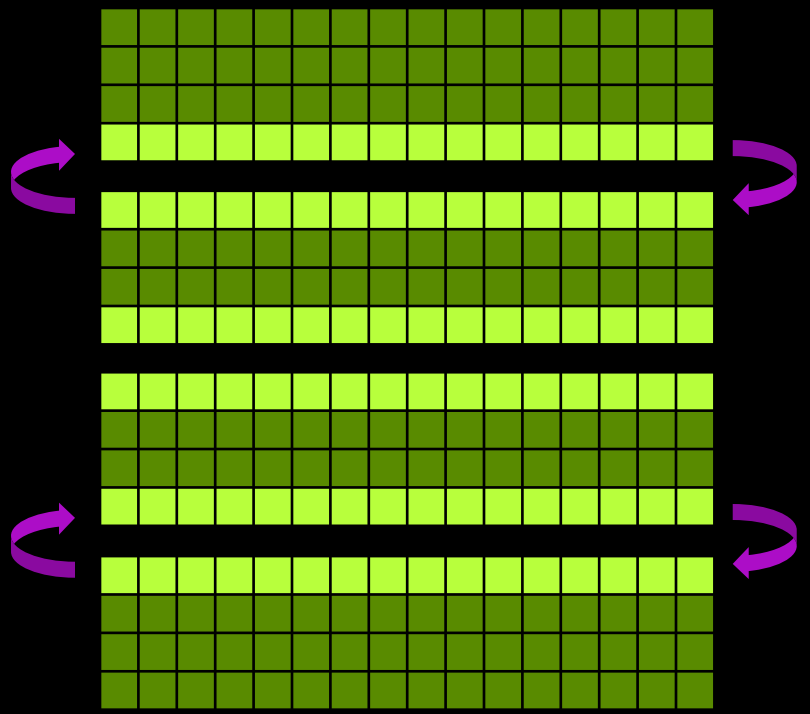
Example: 4-GPU Topology



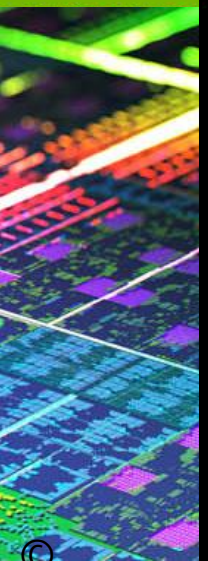
- Two ways to exchange
  - Left-right approach



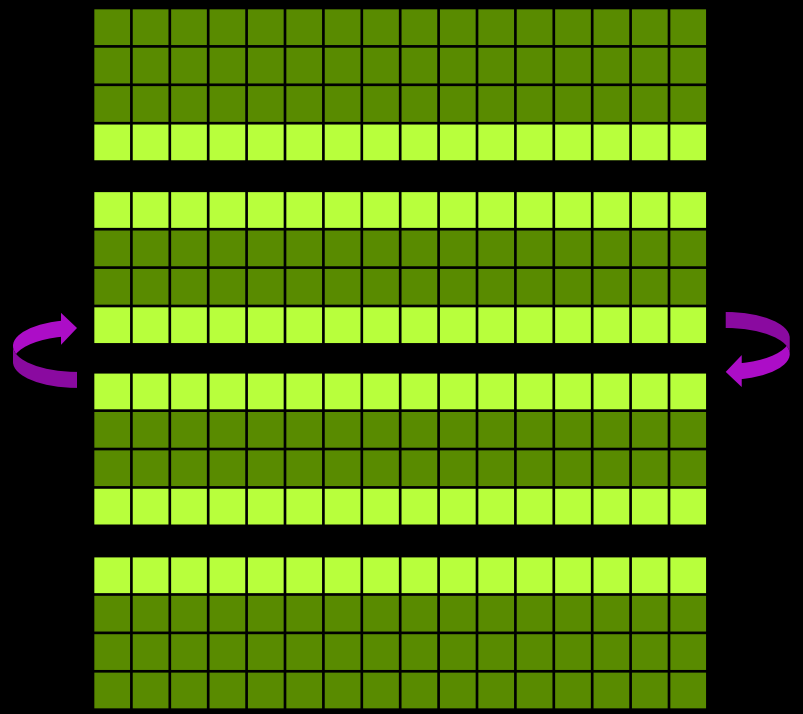
Example: 4-GPU Topology



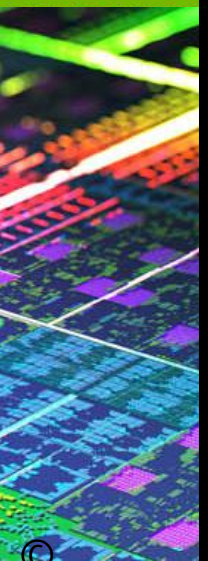
- Two ways to exchange
  - Left-right approach
  - Pairwise approach



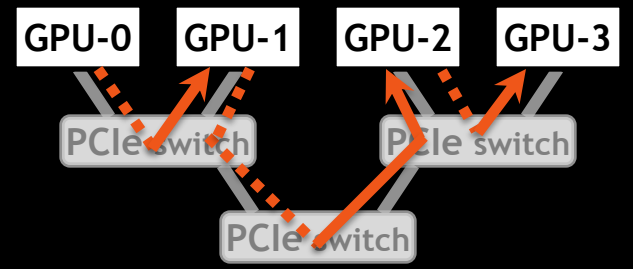
Example: 4-GPU Topology



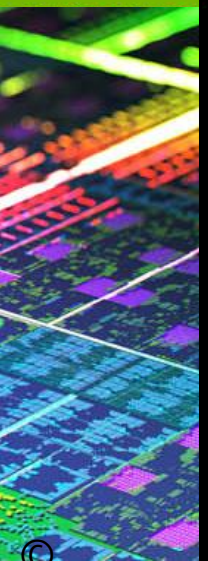
- Two ways to exchange
  - Left-right approach
  - Pairwise approach



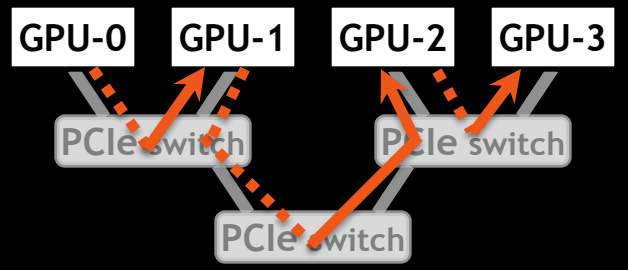
Example: Left-Right Approach for 4 GPUs



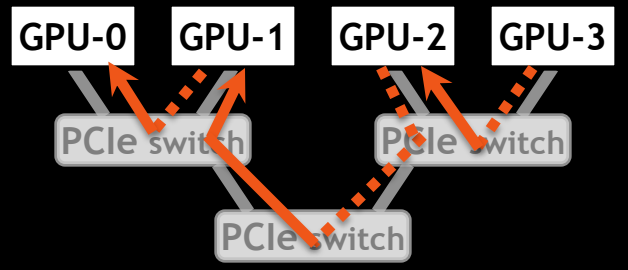
Stage 1: send “right” / receive from “left”



Example: Left-Right Approach for 4 GPUs

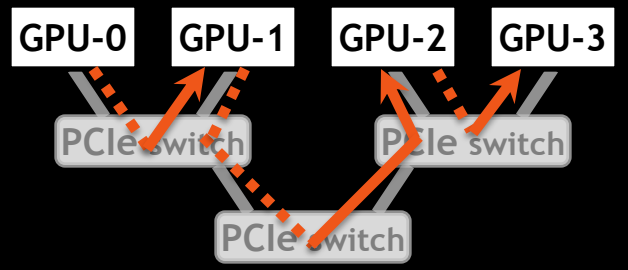


Stage 1: send “right” / receive from “left”

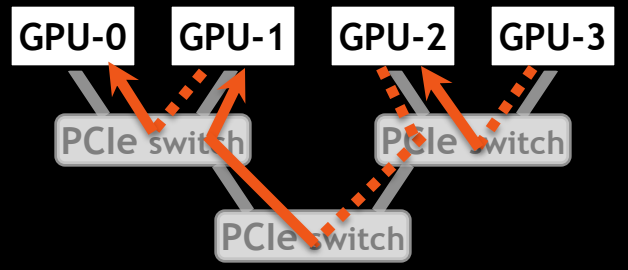


Stage 2: send “left” / receive from “right”

Example: Left-Right Approach for 4 GPUs



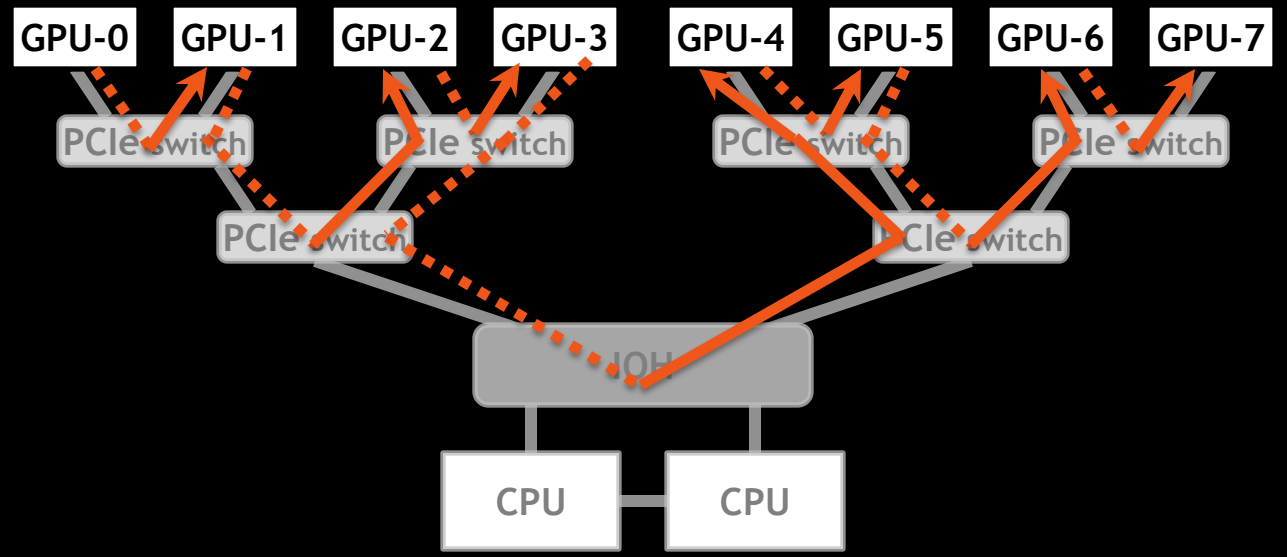
Stage 1: send “right” / receive from “left”



Stage 2: send “left” / receive from “right”

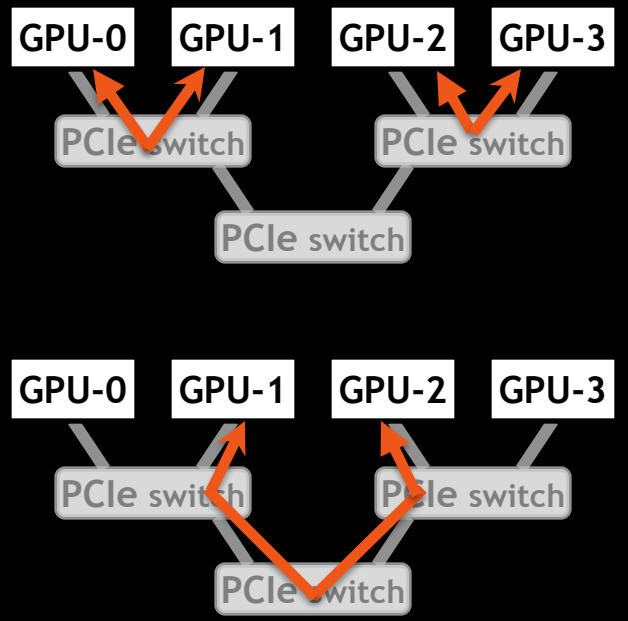
Achieved throughput: ~15 GB/s

Example: Left-Right Approach for 8 GPUs



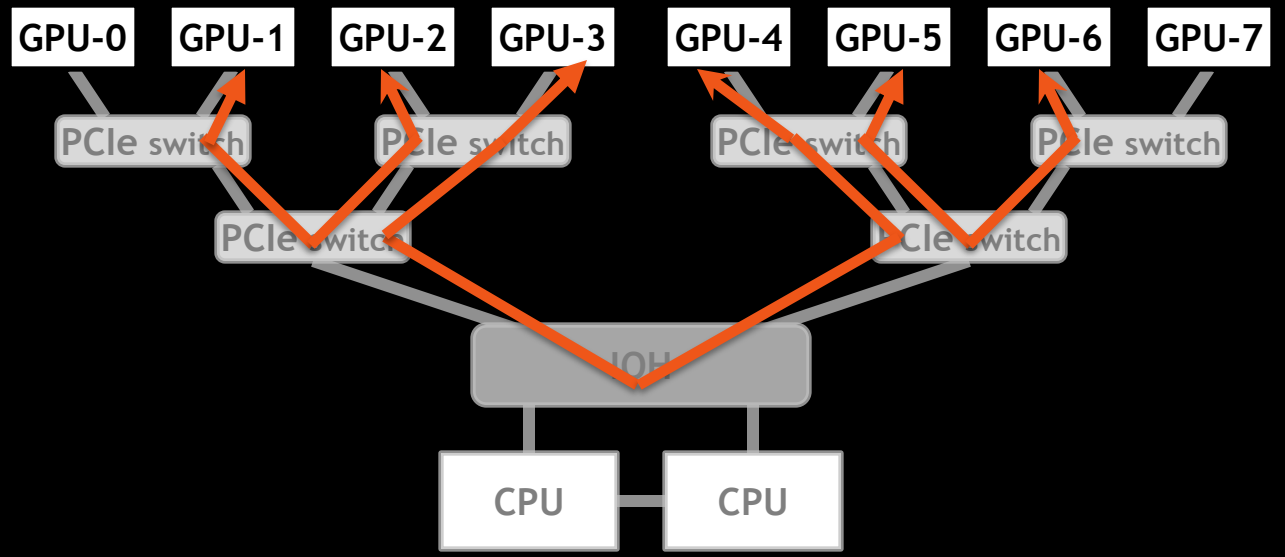
Achieved aggregate throughput: ~34 GB/s

Example: Pairwise Approach for 4 GPUs

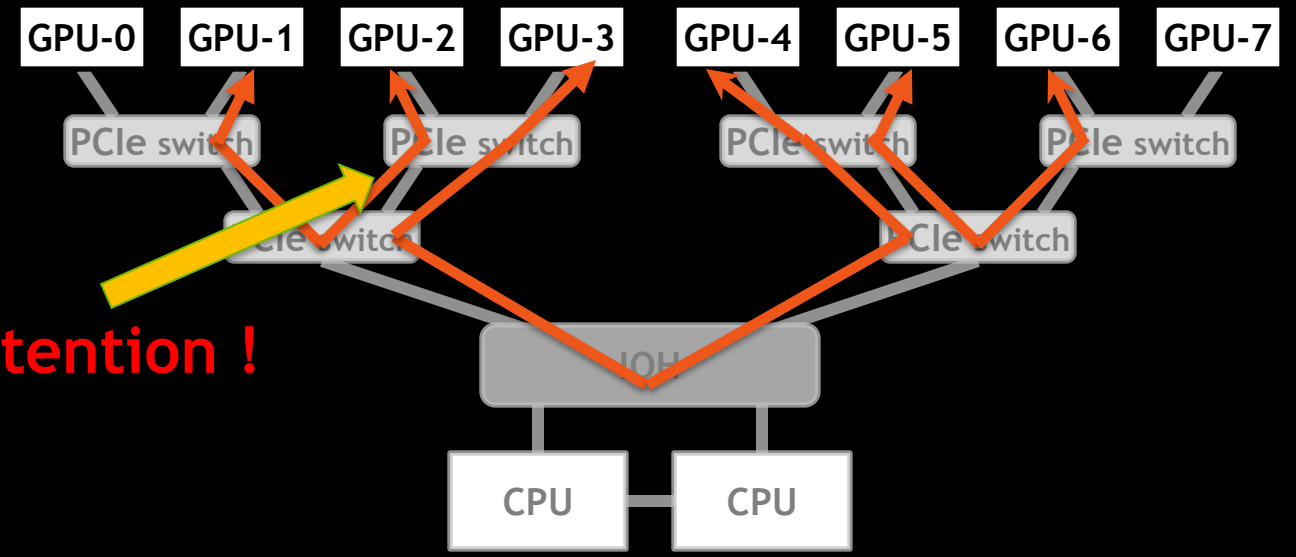


- No contention for PCIe links

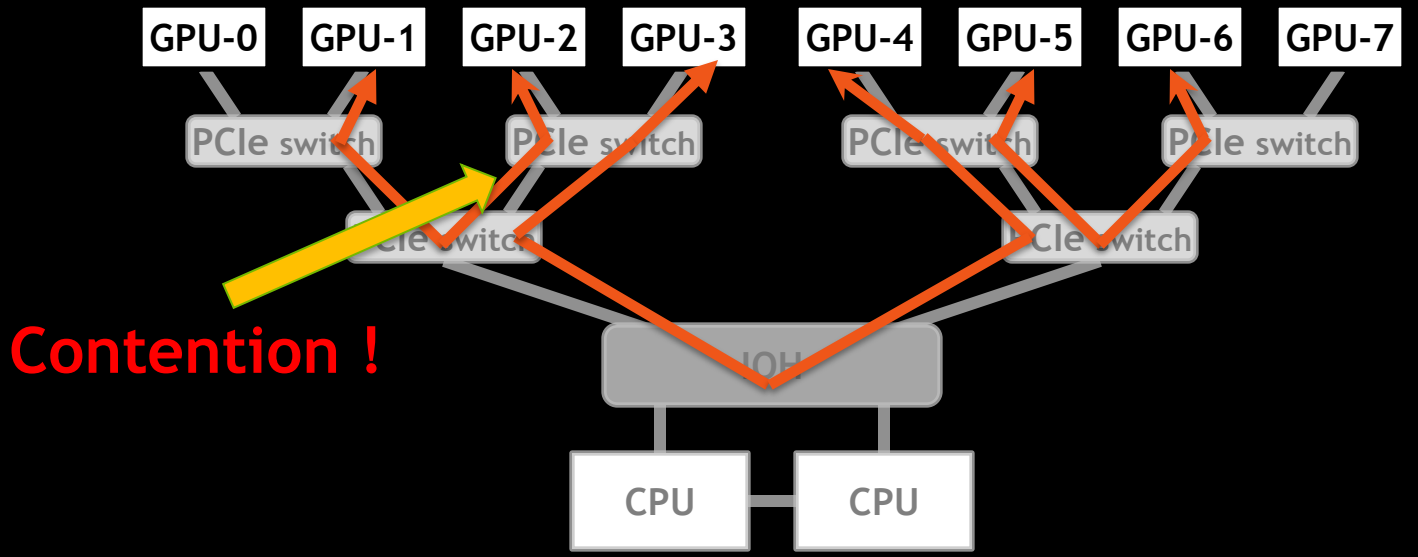
Example: Even-Odd Stage of Pairwise Approach for 8 GPUs



Example: Even-Odd Stage of Pairwise Approach for 8 GPUs



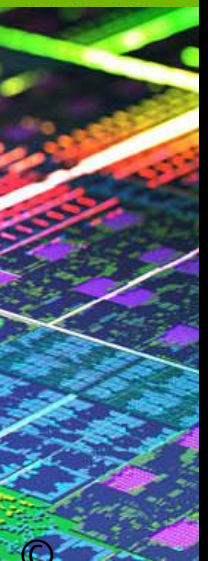
Example: Even-Odd Stage of Pairwise Approach for 8 GPUs



- Odd-even stage:
  - Will always have contention for 8 or more GPUs
- Even-odd stage:
  - Will not have contention

1D Communication

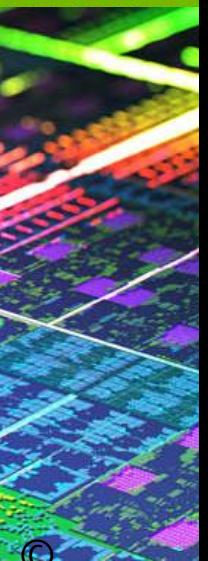
- For 2 GPUs, the approaches are equivalent
- Left-Right approach better for the other cases
- In all cases, duplex improves bandwidth



Code for the Left-Right Approach

```
for( int i=0; i<num_gpus-1; i++ )      // “right” stage
   cudaMemcpyPeerAsync( d_out[i+1], gpu[i+1], d_in[i], gpu[i], num_bytes, stream[i] );
```

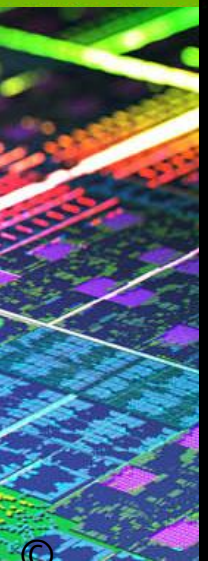
gpu[] = device ids
d_out[], d_in[] = gpu memory



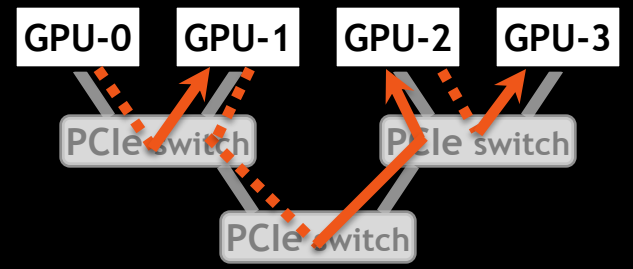
Code for the Left-Right Approach

```
for( int i=0; i<num_gpus-1; i++ )      // “right” stage
   cudaMemcpyPeerAsync( d_out[i+1], gpu[i+1], d_in[i], gpu[i], num_bytes, stream[i] );
```

```
for( int i=1; i<num_gpus; i++ )        // “left” stage
   cudaMemcpyPeerAsync( d_out[i-1], gpu[i-1], d_in[i], gpu[i], num_bytes, stream[i] );
```



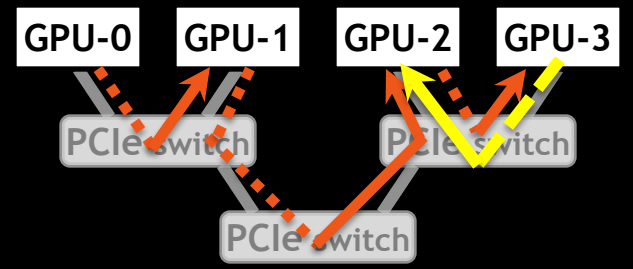
PCIe contention




```
for( int i=0; i<num_gpus-1; i++ )      // “right” stage
  cudaMemcpyPeerAsync( d_out[i+1], gpu[i+1], d_in[i], gpu[i], num_bytes, stream[i] );
```

```
for( int i=1; i<num_gpus; i++ )        // “left” stage
  cudaMemcpyPeerAsync( d_out[i-1], gpu[i-1], d_in[i], gpu[i], num_bytes, stream[i] );
```

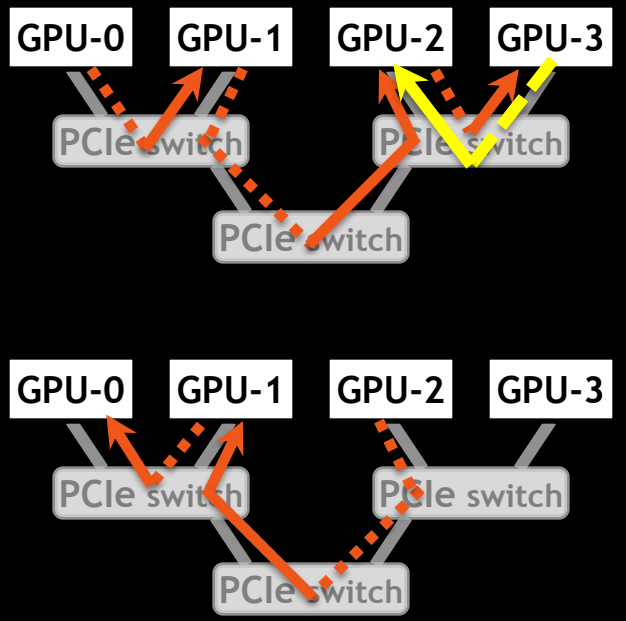
PCIe contention




```
for( int i=0; i<num_gpus-1; i++ )      // “right” stage
  cudaMemcpyPeerAsync( d_out[i+1], gpu[i+1], d_in[i], gpu[i], num_bytes, stream[i] );
```

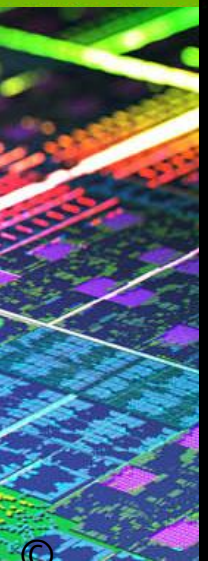
```
for( int i=1; i<num_gpus; i++ )        // “left” stage
  cudaMemcpyPeerAsync( d_out[i-1], gpu[i-1], d_in[i], gpu[i], num_bytes, stream[i] );
```

PCIe contention

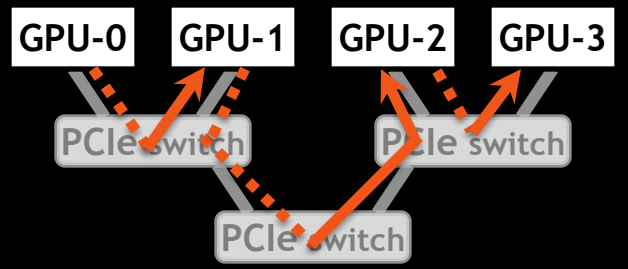


Fix

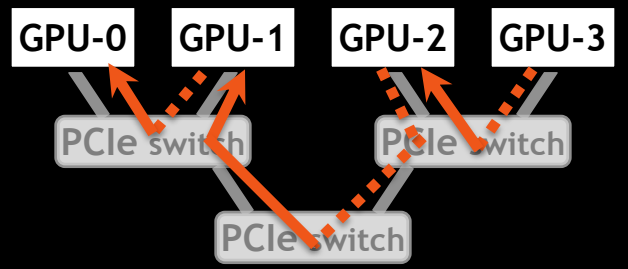
```
for( int i=0; i<num_gpus-1; i++ )      // “right” stage
   cudaMemcpyPeerAsync( d_out[i+1], gpu[i+1], d_in[i], gpu[i], num_bytes, stream[i] );

for( int i=0; i<num_gpus; i++ )
    cudaStreamSynchronize( stream[i] );

for( int i=1; i<num_gpus; i++ )        // “left” stage
   cudaMemcpyPeerAsync( d_out[i-1], gpu[i-1], d_in[i], gpu[i], num_bytes, stream[i] );
```



Better performance with Synchronize

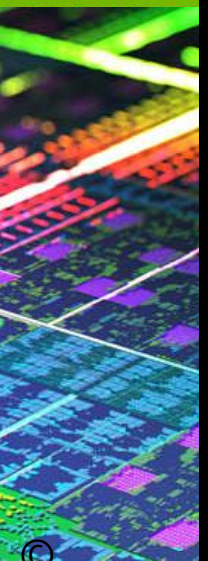


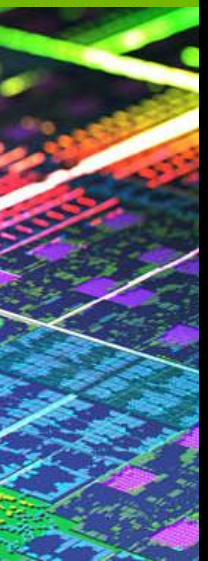
Synchronize



Determining Topology/Locality of a System

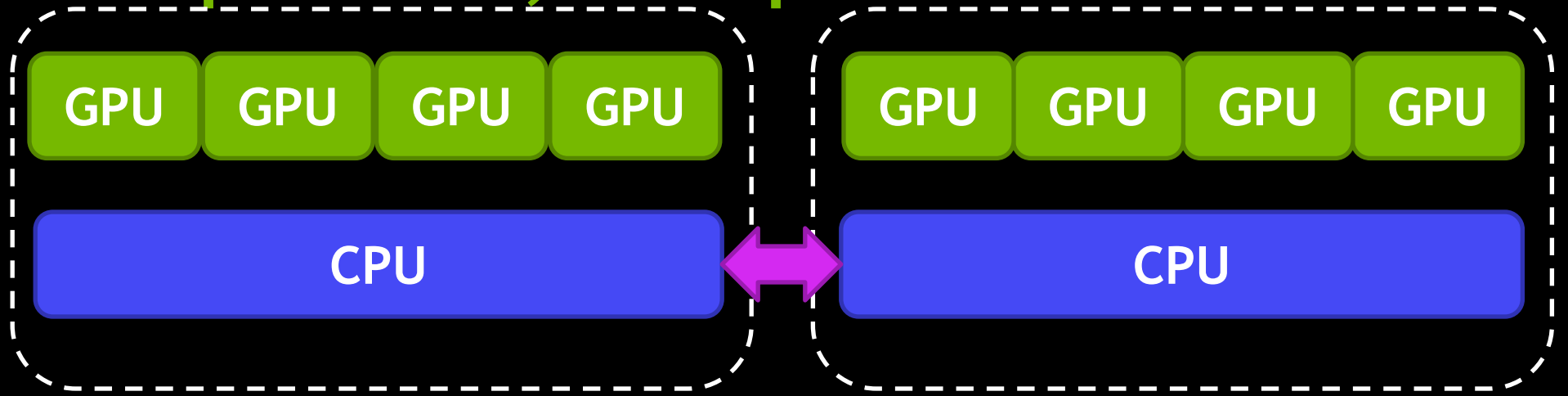
- Hardware Locality tool:
  - http://www.open-mpi.org/projects/hwloc/
  - Cross-OS, cross-platform
- lspci -tvv



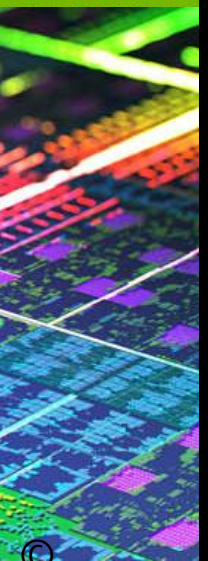


Hiding inter-node communication

Multiple GPUs, multiple nodes

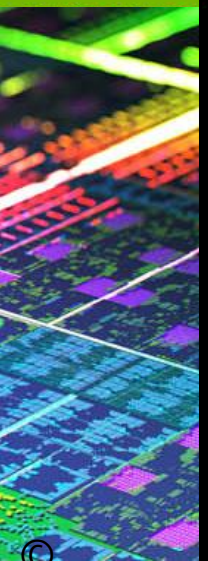


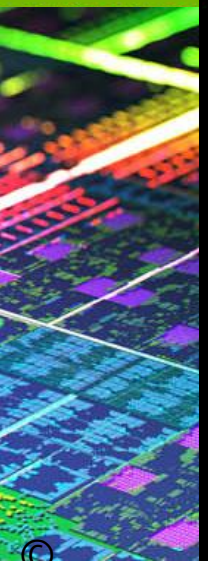

```
cudaMemcpyAsync( ..., stream_halo[i] );
cudaStreamSynchronize( stream_halo[i] );
MPI_Sendrecv( ... );
cudaMemcpyAsync( ..., stream_halo[i] );
```



GPU-aware MPI

- MVAPICH
- OpenMPI
- Cray





GPU-aware MPI

S3047

Introduction to CUDA-aware MPI and NVIDIA GPUDirect™

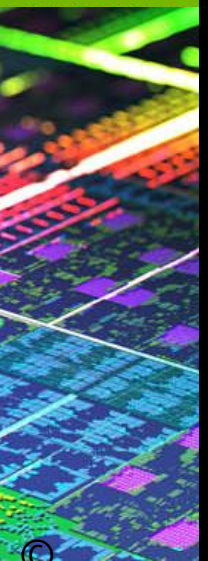
Jiri Kraus

Wednesday 16:00-16:50

Rm 230C

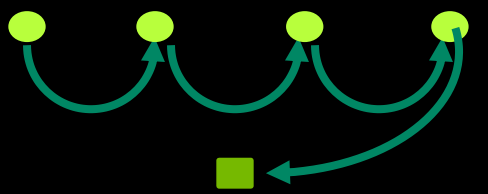
Overlapping MPI and PCIe Transfers

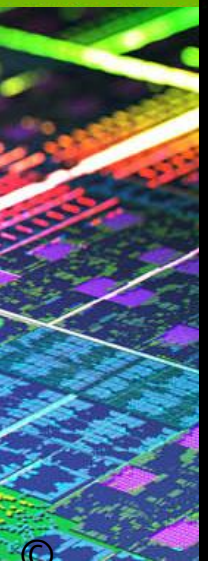
```
for( int i=0; i<num_gpus-1; i++ )
    cudaMemcpyPeerAsync( ..., stream_halo[i] );


for( int i=0; i<num_gpus; i++ )
    cudaStreamSynchronize( stream_halo[i] );
```



Overlapping MPI and PCIe Transfers

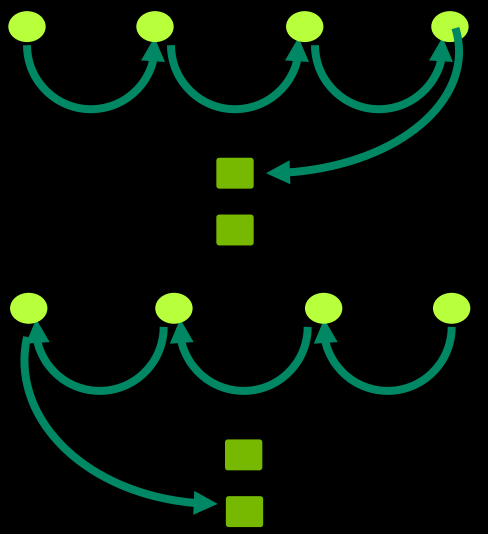
```
for( int i=0; i<num_gpus-1; i++ )
    cudaMemcpyPeerAsync( ..., stream_halo[i] );
cudaSetDevice( gpu[num_gpus-1] );
cudaMemcpyAsync( ..., stream_halo[num_gpus-1] );

for( int i=0; i<num_gpus; i++ )
    cudaStreamSynchronize( stream_halo[i] );
```

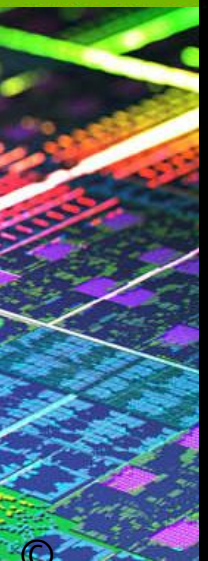




Overlapping MPI and PCIe Transfers

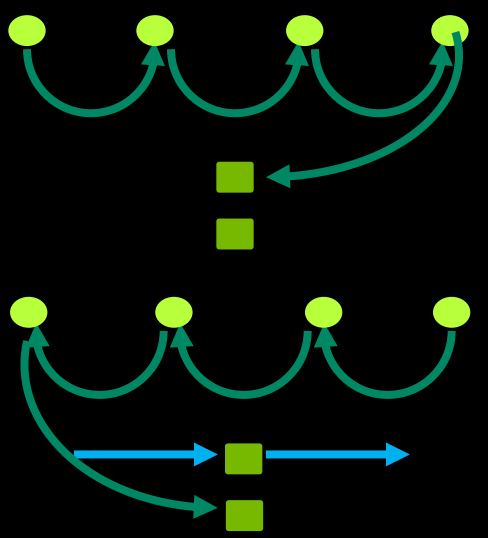
```
for( int i=0; i<num_gpus-1; i++ )
    cudaMemcpyPeerAsync( ..., stream_halo[i] );
cudaSetDevice( gpu[num_gpus-1] );
cudaMemcpyAsync( ..., stream_halo[num_gpus-1] );

for( int i=0; i<num_gpus; i++ )
    cudaStreamSynchronize( stream_halo[i] );

for( int i=1; i<num_gpus; i++ )
    cudaMemcpyPeerAsync( ..., stream_halo[i] );
cudaSetDevice( gpu[0] );
cudaMemcpyAsync( ..., stream_halo[0] );
```

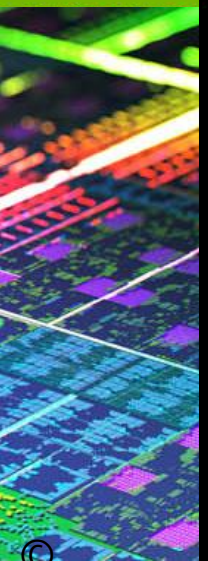




Overlapping MPI and PCIe Transfers

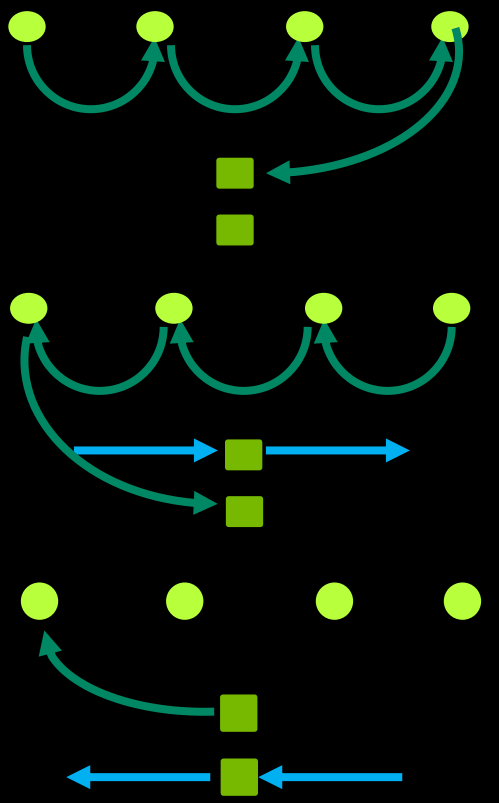
```
for( int i=0; i<num_gpus-1; i++ )
    cudaMemcpyPeerAsync( ..., stream_halo[i] );
cudaSetDevice( gpu[num_gpus-1] );
cudaMemcpyAsync( ..., stream_halo[num_gpus-1] );

for( int i=0; i<num_gpus; i++ )
    cudaStreamSynchronize( stream_halo[i] );

for( int i=1; i<num_gpus; i++ )
    cudaMemcpyPeerAsync( ..., stream_halo[i] );
cudaSetDevice( gpu[0] );
cudaMemcpyAsync( ..., stream_halo[0] );
MPI_Sendrecv( ... );
```

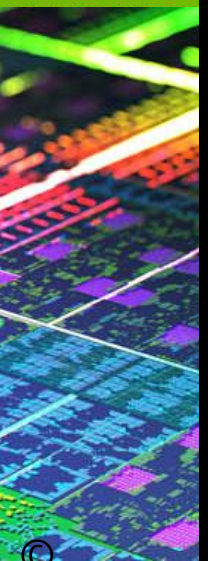




Overlapping MPI and PCIe Transfers

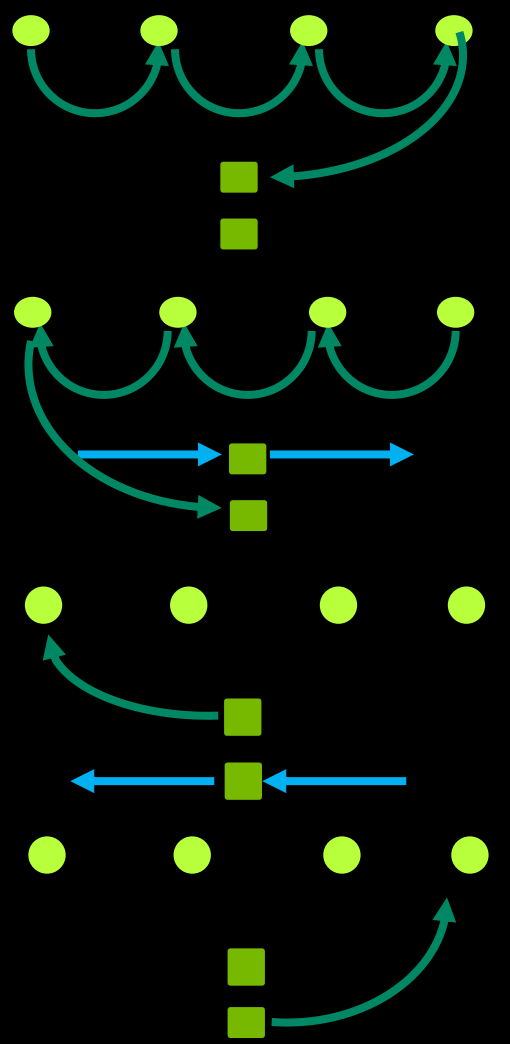
```
for( int i=0; i<num_gpus-1; i++ )
    cudaMemcpyPeerAsync( ..., stream_halo[i] );
cudaSetDevice( gpu[num_gpus-1] );
cudaMemcpyAsync( ..., stream_halo[num_gpus-1] );

for( int i=0; i<num_gpus; i++ )
    cudaStreamSynchronize( stream_halo[i] );

for( int i=1; i<num_gpus; i++ )
    cudaMemcpyPeerAsync( ..., stream_halo[i] );
cudaSetDevice( gpu[0] );
cudaMemcpyAsync( ..., stream_halo[0] );
MPI_Sendrecv( ... );

for( int i=0; i<num_gpus; i++ )
    cudaStreamSynchronize( stream_halo[i] );

cudaSetDevice( gpu[0] );
cudaMemcpyAsync( ..., stream_halo[0] );
MPI_Sendrecv( ... );
```



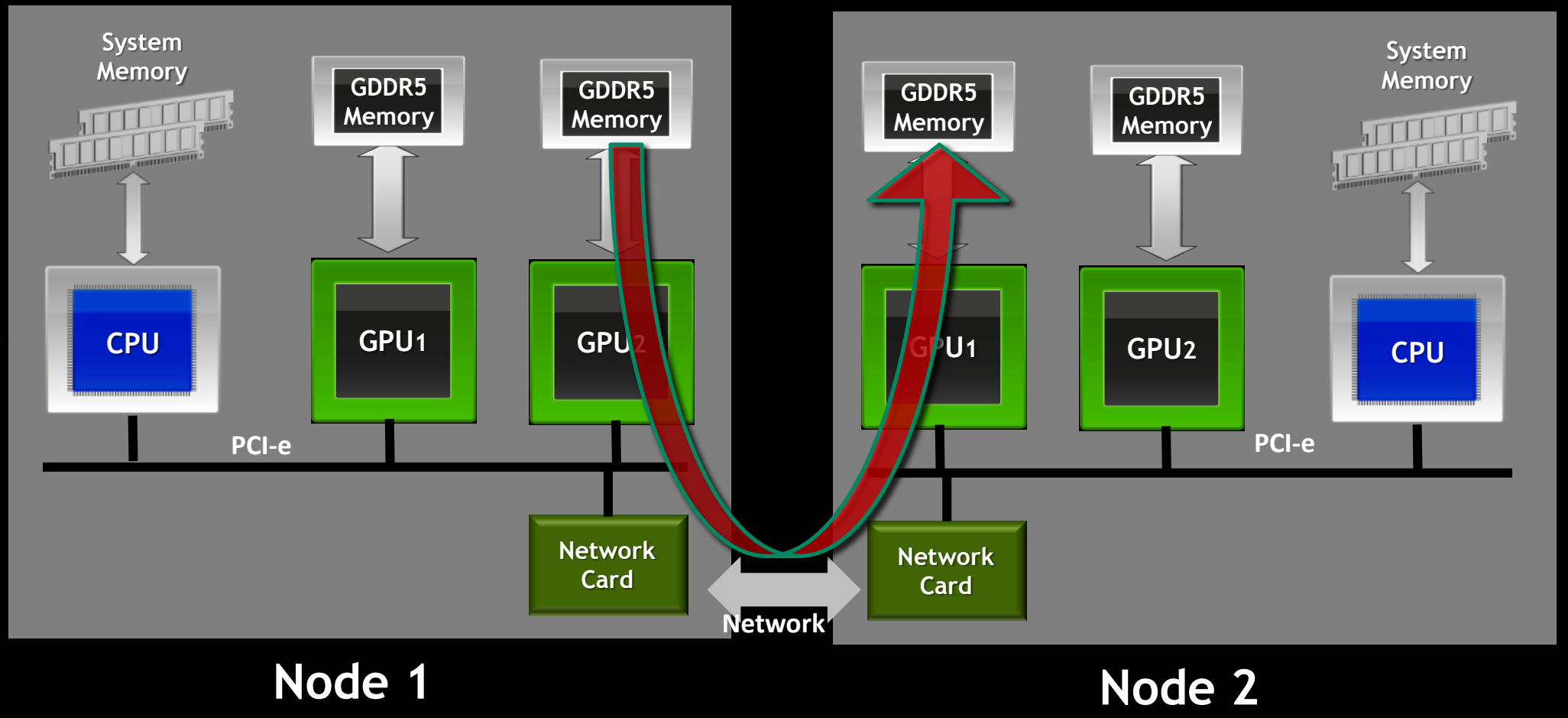


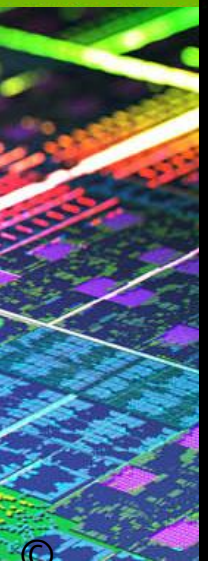
Overlapping MPI and PCIe Transfers

```
for( int i=0; i<num_gpus-1; i++ )
    cudaMemcpyPeerAsync( ..., stream_halo[i] );
cudaSetDevice( gpu[num_gpus-1] );
cudaMemcpyAsync( ..., stream_halo[num_gpus-1] );

for( int i=0; i<num_gpus; i++ )
    cudaStreamSynchronize( stream_halo[i] );

for( int i=1; i<num_gpus; i++ )
    cudaMemcpyPeerAsync( ..., stream_halo[i] );
cudaSetDevice( gpu[0] );
cudaMemcpyAsync( ..., stream_halo[0] );
MPI_Sendrecv( ... );

for( int i=0; i<num_gpus; i++ )
    cudaStreamSynchronize( stream_halo[i] );

cudaSetDevice( gpu[0] );
cudaMemcpyAsync( ..., stream_halo[0] );
MPI_Sendrecv( ... );

cudaSetDevice( gpu[num_gpus-1] );
cudaMemcpyAsync( ..., stream_halo[num_gpus-1] );
```



NVIDIA® GPUDirect™ Support for RDMA

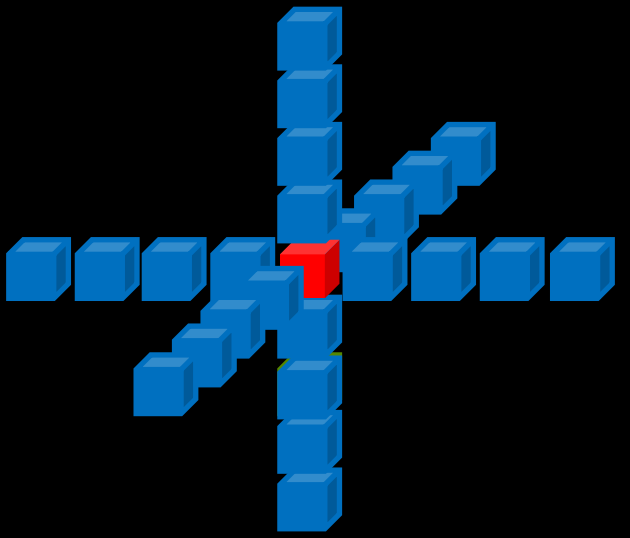




Case Study

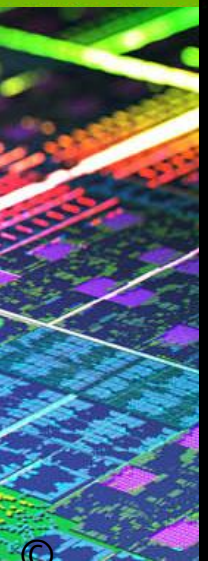
Case Study: TTI FWM

- TTI Forward Wave Modeling
  - Fundamental part of TTI RTM
  - 3DFD, 8th order in space, 2nd order in time
  - 1D domain decomposition



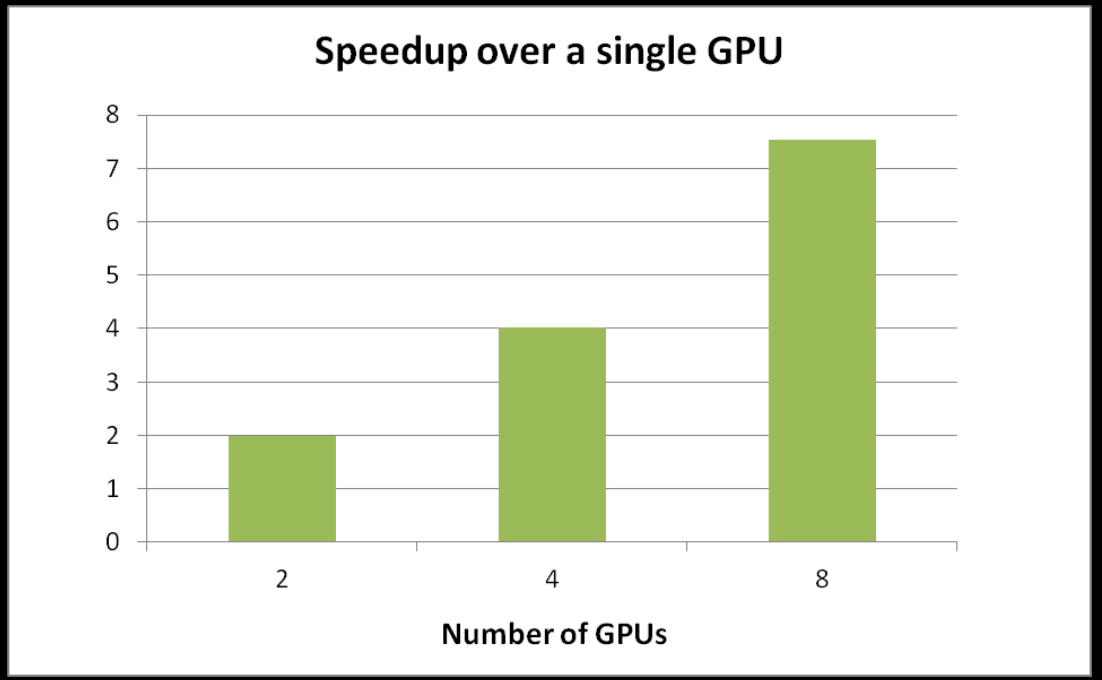
Case Study: TTI FWM

- Experiments:
  - 512x512x512 cube
  - Requires ~7 GB working set
  - Single node, 4-GPU “tree”



Case Study: TTI FWM

- Experiments:
  - 512x512x512 cube
  - Requires ~7 GB working set
  - Single node, 4-GPU “tree”
  - 95% scaling to 8 GPUs/Node

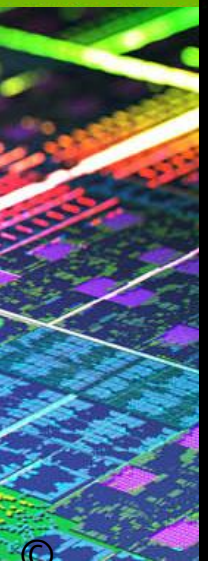


Case Study: Time Breakdown

- Single step (single 8-GPU node):

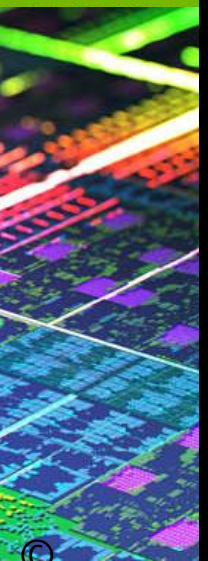
Halo computation:	1.1 ms
Internal computation:	12.7 ms
Halo-exchange:	5.9 ms
Total:	13.8 ms

- Communication is completely hidden
  - ~95% scaling: halo+internal: 13.8 ms (13.0 ms if done without splitting computation into halo and internal)
  - Time enough for slower communication (network)



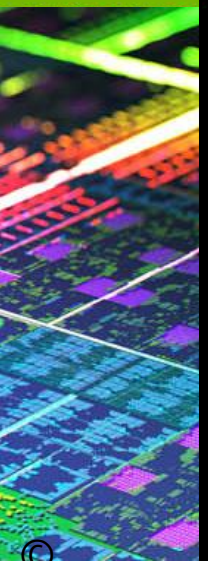
Case Study: Multiple Nodes

- Test system:
  - 3 servers, each with 2 GPUs, Infiniband DDR interconnect



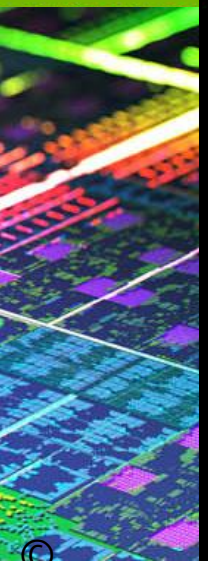
Case Study: Multiple Nodes

- Test system:
  - 3 servers, each with 2 GPUs, Infiniband DDR interconnect
- Performance:
  - 512x512x512 domain:
    - 1 node x 2 GPUs: 1.98x
    - 2 nodes x 1 GPU: 1.97x
    - 2 nodes x 2 GPUs: 3.98x



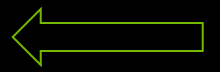
Case Study: Multiple Nodes

- Test system:
  - 3 servers, each with 2 GPUs, Infiniband DDR interconnect
- Performance:
  - 512x512x512 domain:
    - 1 node x 2 GPUs: 1.98x
    - 2 nodes x 1 GPU: 1.97x
    - 2 nodes x 2 GPUs: 3.98x
    - 3 nodes x 2 GPUs: 4.50x

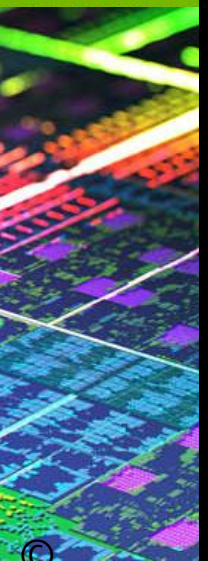


Case Study: Multiple Nodes

- Test system:
  - 3 servers, each with 2 GPUs, Infiniband DDR interconnect
- Performance:
  - 512x512x512 domain:
    - 1 node x 2 GPUs: 1.98x
    - 2 nodes x 1 GPU: 1.97x
    - 2 nodes x 2 GPUs: 3.98x
    - 3 nodes x 2 GPUs: 4.50x

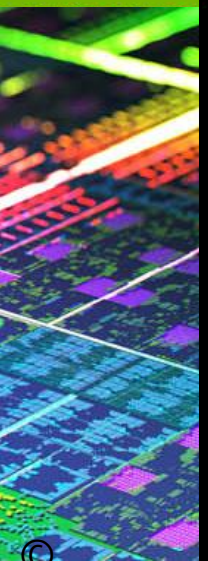


Communication takes longer than internal computation



Case Study: Multiple Nodes

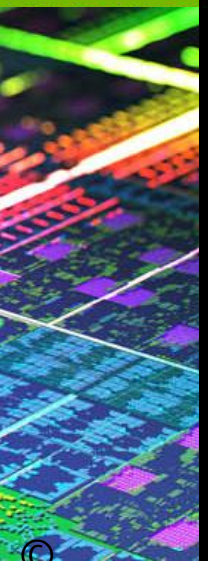
- Test system:
  - 3 servers, each with 2 GPUs, Infiniband DDR interconnect
- Performance:
  - 512x512x512 domain:
    - 1 node x 2 GPUs: 1.98x
    - 2 nodes x 1 GPU: 1.97x
    - 2 nodes x 2 GPUs: 3.98x
    - 3 nodes x 2 GPUs: 4.50x
  - 768x768x768 domain:
    - 3 nodes x 2 GPUs: 5.60x



Communication takes longer than internal computation

Case Study: Multiple Nodes

- Test system:
  - 3 servers, each with 2 GPUs, Infiniband DDR interconnect
- Performance:
  - 512x512x512 domain:
    - 1 node x 2 GPUs: 1.98x
    - 2 nodes x 1 GPU: 1.97x
    - 2 nodes x 2 GPUs: 3.98x
    - 3 nodes x 2 GPUs: 4.50x
  - 768x768x768 domain:
    - 3 nodes x 2 GPUs: 5.60x



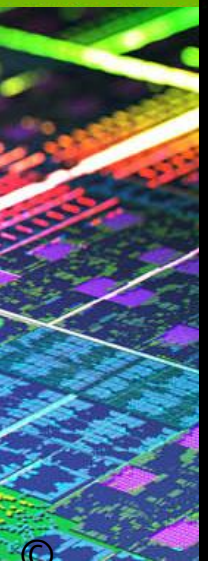
Communication takes longer than internal computation

Case Study: Multiple Nodes

- Test system:
  - 3 servers, each with 2 GPUs, Infiniband DDR interconnect
- Performance:
  - 512x512x512 domain:
    - 1 node x 2 GPUs: 1.98x
    - 2 nodes x 1 GPU: 1.97x
    - 2 nodes x 2 GPUs: 3.98x
    - 3 nodes x 2 GPUs: 4.50x
  - 768x768x768 domain:
    - 3 nodes x 2 GPUs: 5.60x

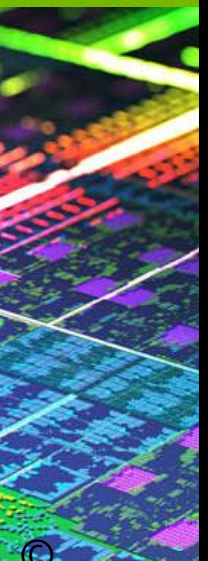
Communication takes longer than internal computation

- Conclusions
  - Communication (PCIe and IB DDR2) is hidden when each GPU gets ~100 slices
    - Network is ~68% of all communication time
  - IB QDR hides communication when each GPU gets ~70 slices



Summary

- Multiple GPUs can stretch your compute dollar
- PeerToPeer and can move data directly between GPUs
- Streams and asynchronous kernel/copies facilitate concurrent execution
- Many apps can scale to 8 GPUs and beyond



Get started

- Hands on Training

  S3529 Wednesday 5pm

- GPU Test Drive

  http://www.nvidia.com/GPUTestDrive

- Visit us at the CUDA expert table