# CPSC/ECE 4780/6780

# General-Purpose Computation on Graphical Processing Units (GPGPU)

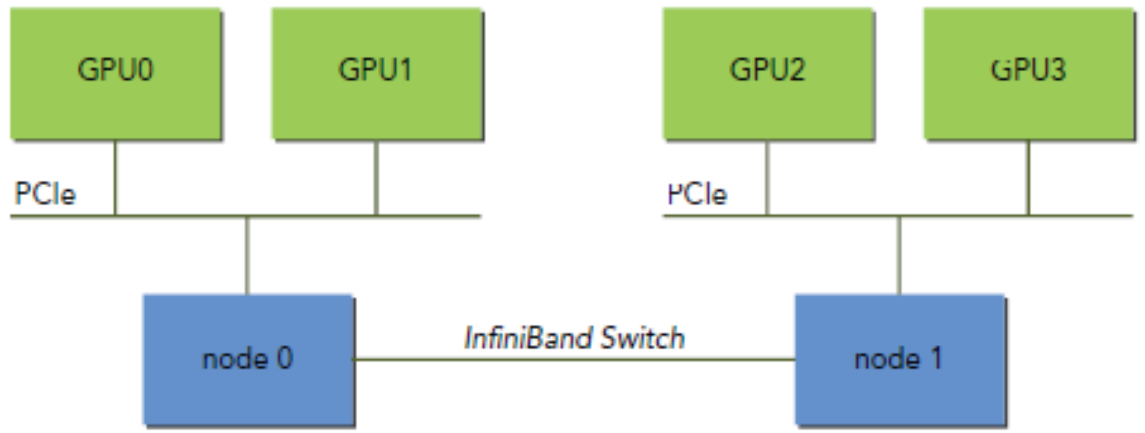
## Lecture 10: Multi-GPU Programming

## Recap of Last Lecture

- What are CUDA streams?
- What is Null stream? And non-Null stream?
- What are the conditions to be satisfied when using streams to overlap device execution with data transfer?
- What is nvvp?

# Multiple GPUs

- Why multiple GPUs?
  - Problem domain size
  - Throughput and efficiency
- Connectivity of multi-GPU systems:
  - Multiple GPUs connected over the PCIe bus in a single node
  - Multiple GPUs connected over a network switch in a cluster



## Inter-GPU Communication Patterns

- Two common inter-GPU communication patterns:
  - No data exchange between problem partitions
    - No data shared across GPUs
    - Each partition can run independently
  - Partial data exchange between problem partitions
    - Need to move data between devices
    - Avoid staging data though host memory

## Executing on Multiple GPUs

- **cudaError_t cudaSetDevice(int id)** sets the current device
- All CUDA operations applies to that device
  - Any device memory allocated from the host thread will be physically resident on that device
  - Any host memory allocated with CUDA runtime functions will have its lifetime associated with that device
  - Any streams or events created from the host thread will be associated with that device
  - Any kernels launched from the host thread will be executed on that device

## Managing Multiple GPUs From A Single CPU Thread

- First, determine the number of CUDA-enabled devices available in a system

```
cudaError_t cudaGetDeviceCount(int *count);
cudaError_t cudaGetDeviceProperties(struct cudaDeviceProp *prop, int device);
```

- Next, iterate over devices to execute kernels and memory copies

```
for (int i = 0; i < ngpus; i++) {
  // set the current device
  cudaSetDevice(i);
  // execute kernel on current device
  kernel<<<grid, block>>>(...);
  // asynchronously transfer data between the host and current device
  cudaMemcpyAsync(...);
}
```

- Current GPU can be changed while asynchronous calls (kernels, memory copies) are running

# Example: Subdividing Vector Addition Across Multiple GPUs

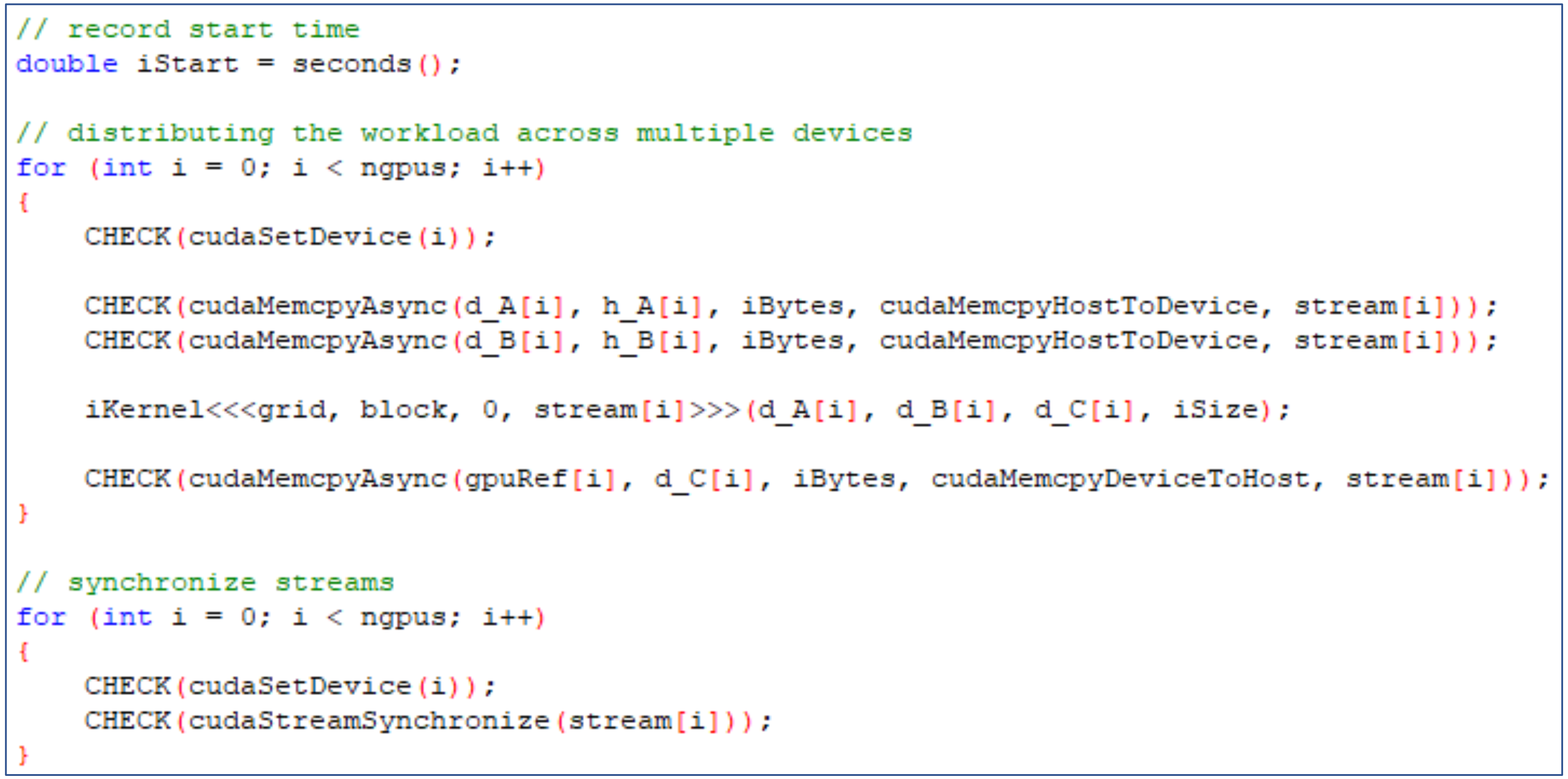
- Allocating memory on multiple devices

```
for (int i = 0; i < ngpus; i++)
{
    // set current device
    CHECK(cudaSetDevice(i));

    // allocate device memory
    CHECK(cudaMalloc((void **) &d_A[i], iBytes));
    CHECK(cudaMalloc((void **) &d_B[i], iBytes));
    CHECK(cudaMalloc((void **) &d_C[i], iBytes));

    // allocate page locked host memory for asynchronous data transfer
    CHECK(cudaMallocHost((void **) &h_A[i],     iBytes));
    CHECK(cudaMallocHost((void **) &h_B[i],     iBytes));
    CHECK(cudaMallocHost((void **) &hostRef[i], iBytes));
    CHECK(cudaMallocHost((void **) &gpuRef[i],  iBytes));

    // create streams for timing and synchronizing
    CHECK(cudaStreamCreate(&stream[i]));
}
```

```
int size = 1 << 24;
int    iSize  = size / ngpus;
size_t iBytes = iSize * sizeof(float);
```

## Example: Subdividing Vector Addition Across Multiple GPUs

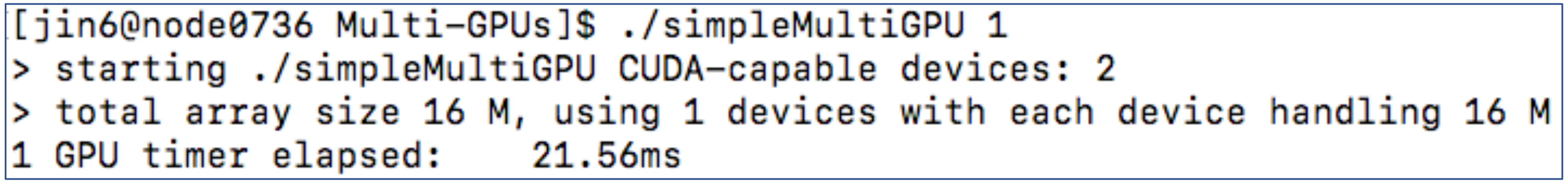
- Distributing work from a single host thread



```
// record start time
double iStart = seconds();

// distributing the workload across multiple devices
for (int i = 0; i < ngpus; i++)
{
    CHECK(cudaSetDevice(i));

    CHECK(cudaMemcpyAsync(d_A[i], h_A[i], iBytes, cudaMemcpyHostToDevice, stream[i]));
    CHECK(cudaMemcpyAsync(d_B[i], h_B[i], iBytes, cudaMemcpyHostToDevice, stream[i]));

    iKernel<<<grid, block, 0, stream[i]>>>(d_A[i], d_B[i], d_C[i], iSize);

    CHECK(cudaMemcpyAsync(gpuRef[i], d_C[i], iBytes, cudaMemcpyDeviceToHost, stream[i]));
}

// synchronize streams
for (int i = 0; i < ngpus; i++)
{
    CHECK(cudaSetDevice(i));
    CHECK(cudaStreamSynchronize(stream[i]));
}
```

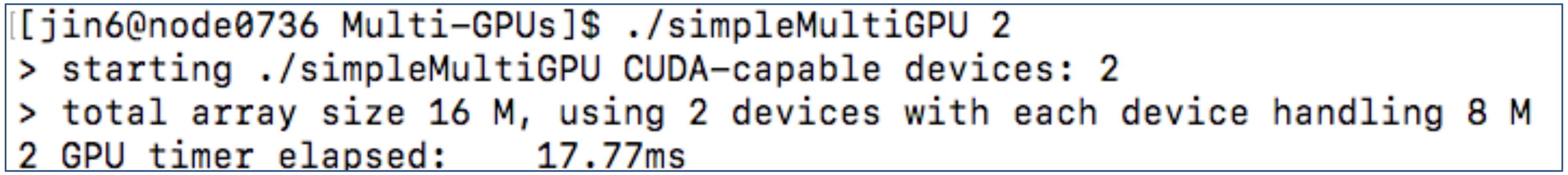
# Example: Subdividing Vector Addition Across Multiple GPUs

- Running with only one GPU



```
[jin6@node0736 Multi-GPUs]$ ./simpleMultiGPU 1
> starting ./simpleMultiGPU CUDA-capable devices: 2
> total array size 16 M, using 1 devices with each device handling 16 M
1 GPU timer elapsed:    21.56ms
```

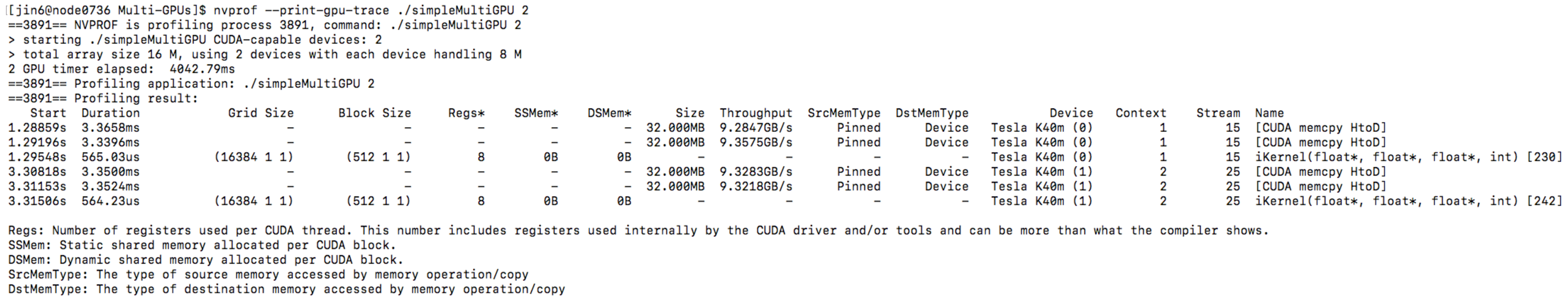
- Running with two GPUs



```
[jin6@node0736 Multi-GPUs]$ ./simpleMultiGPU 2
> starting ./simpleMultiGPU CUDA-capable devices: 2
> total array size 16 M, using 2 devices with each device handling 8 M
2 GPU timer elapsed:    17.77ms
```

# Example: Subdividing Vector Addition Across Multiple GPUs

- More details about each device’s behavior using nvprof:

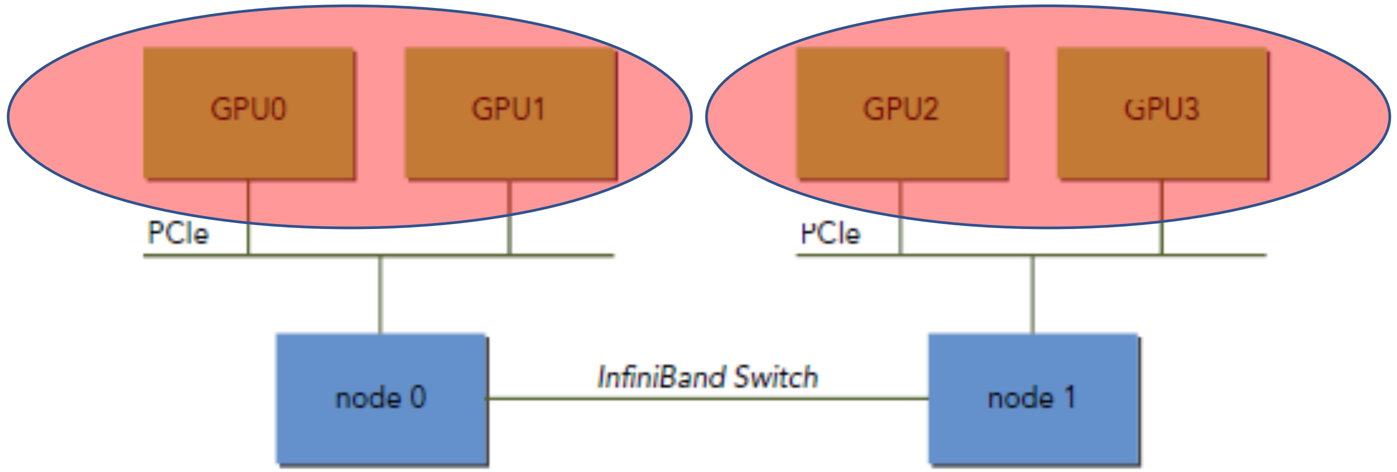


```
[jin6@node0736 Multi-GPUs]$ nvprof --print-gpu-trace ./simpleMultiGPU 2
==3891== NVPROF is profiling process 3891, command: ./simpleMultiGPU 2
> starting ./simpleMultiGPU CUDA-capable devices: 2
> total array size 16 M, using 2 devices with each device handling 8 M
2 GPU timer elapsed:  4042.79ms
==3891== Profiling application: ./simpleMultiGPU 2
==3891== Profiling result:
   Start  Duration            Grid Size      Block Size     Regs*    SSMem*    DSMem*      Size  Throughput  SrcMemType  DstMemType           Device   Context    Stream  Name
1.28859s  3.3658ms                    -               -         -         -         -  32.000MB  9.2847GB/s      Pinned      Device  Tesla K40m (0)         1        15  [CUDA memcpy HtoD]
1.29196s  3.3396ms                    -               -         -         -         -  32.000MB  9.3575GB/s      Pinned      Device  Tesla K40m (0)         1        15  [CUDA memcpy HtoD]
1.29548s  565.03us          (16384 1 1)       (512 1 1)         8        0B        0B         -           -           -           -  Tesla K40m (0)         1        15  iKernel(float*, float*, float*, int) [230]
3.30818s  3.3500ms                    -               -         -         -         -  32.000MB  9.3283GB/s      Pinned      Device  Tesla K40m (1)         2        25  [CUDA memcpy HtoD]
3.31153s  3.3524ms                    -               -         -         -         -  32.000MB  9.3218GB/s      Pinned      Device  Tesla K40m (1)         2        25  [CUDA memcpy HtoD]
3.31506s  564.23us          (16384 1 1)       (512 1 1)         8        0B        0B         -           -           -           -  Tesla K40m (1)         2        25  iKernel(float*, float*, float*, int) [242]

Regs: Number of registers used per CUDA thread. This number includes registers used internally by the CUDA driver and/or tools and can be more than what the compiler shows.
SSMem: Static shared memory allocated per CUDA block.
DSMem: Dynamic shared memory allocated per CUDA block.
SrcMemType: The type of source memory accessed by memory operation/copy
DstMemType: The type of destination memory accessed by memory operation/copy
```

- Changes need to be made in the code:
  - #include <cuda_profiler_api.h>
  - Wrap your kernel with cudaProfilerStart(); and cudaProfilerStop();
    - cudaProfilerStart();
    - yourKernel<<<…>>>(…);
    - cudaProfilerStop();

# Direct Inter-Device Communication

- Kernels executing on one device can directly access the global memory of any GPU connected to the same PCIe root node
- Two modes supported by **CUDA Peer-to-Peer (P2P) API**
  - **Peer-to-peer access**: directly load and store address within a CUDA kernel and across GPUs
  - **Peer-to-peer transfer**: directly copy data between GPUs



## Enabling Peer Access

- Check if a device supports P2P access by

```
cudaError_t cudaDeviceCanAccessPeer(int* canAccessPeer, int device, int peerDevice);
```

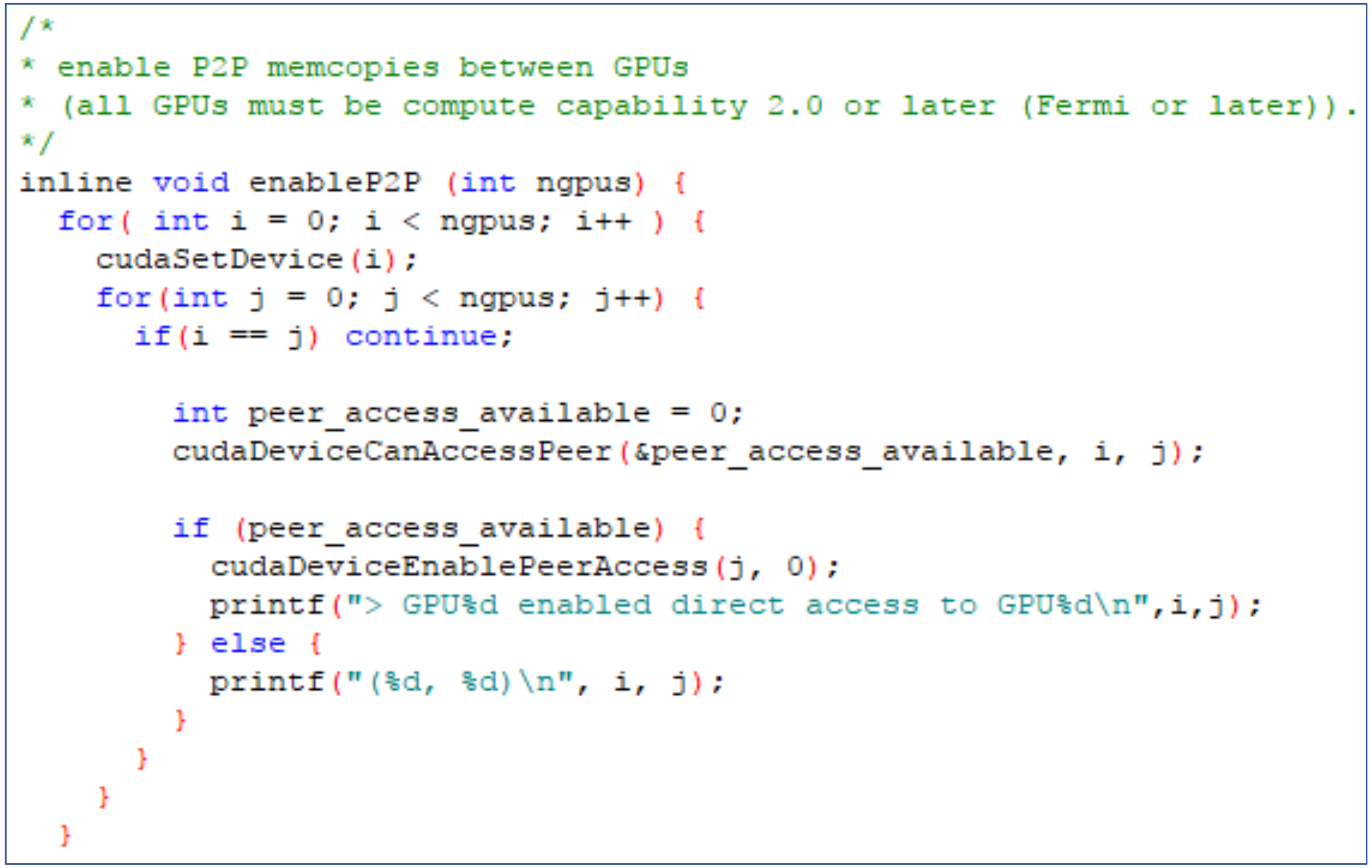
  - Checks whether device can access memory of peerDevice
  - Returns 0/1 via the first argument
- To enable P2P memory access between two devices by

```
cudaError_t cudaDeviceEnablePeerAccess(int peerDevice, unsigned int flag);
```

  - Enables access from the current device to peerDevice
  - The access is unidirectional
  - Access remains enabled until explicitly disabled by

```
cudaError_t cudaDeviceDisablePeerAccess(int peerDevice);
```

# Example: Enabling Bi-Directional Peer Access



```
/*
* enable P2P memcopies between GPUs
* (all GPUs must be compute capability 2.0 or later (Fermi or later)).
*/
inline void enableP2P (int ngpus) {
 for( int i = 0; i < ngpus; i++ ) {
  cudaSetDevice(i);
  for(int j = 0; j < ngpus; j++) {
   if(i == j) continue;

    int peer_access_available = 0;
    cudaDeviceCanAccessPeer(&peer_access_available, i, j);

    if (peer_access_available) {
     cudaDeviceEnablePeerAccess(j, 0);
     printf("> GPU%d enabled direct access to GPU%d\n",i,j);
    } else {
     printf("(%d, %d)\n", i, j);
    }
   }
  }
 }
```

# Peer-to-Peer Memory Copy

```
cudaError_t cudaMemcpyPeerAsync(void* dst, int dstDev, void* src, int srcDev,
size_t nBytes, cudaStream_t stream);
```

- Copies data between two devices
- Currently performance is maximized when stream belong to the source GPU

- If peer-access is enabled
  - Data are transferred along the shortest PCIe path
  - No staging through host memory
- If peer-access is not available
  - CUDA driver stages the transfer via CPU memory => reducing performance

# Example: A Ping-Pong Synchronous Memory Copy

```
// ping pong unidirectional gmem copy
for (int i = 0; i < 100; i++) {
  if (i % 2 == 0) {
    cudaMemcpy(d_src[1], d_src[0], iBytes, cudaMemcpyDeviceToDevice);
  } else {
    cudaMemcpy(d_src[0], d_src[1], iBytes, cudaMemcpyDeviceToDevice);
  }
}
```

Elapsed time and bandwidth:

```
Ping-pong unidirectional cudaMemcpy: 13.41ms performance: 5.00 GB/s
```

## Example: A Ping-Pong Asynchronous Memory Copy

```
// bidirectional asynchronous gmem copy
for (int i = 0; i < 100; i++) {
  cudaMemcpyAsync(d_src[1], d_src[0], iBytes, cudaMemcpyDeviceToDevice, stream[0]);
  cudaMemcpyAsync(d_rcv[0], d_rcv[1], iBytes, cudaMemcpyDeviceToDevice, stream[1]);
}
```

Elapsed time and bandwidth:

```
Ping-pong bidirectional cudaMemcpyAsync: 13.39ms performance: 10.02 GB/s
```

Note: If you disable peer-to-peer access by removing the call to enableP2P(), then both the unidirectional and bidirectional examples still complete without error, but the measured bandwidth will drop as transfers are staged through host memory.

# Multi-GPUs, Streams, and Events

- The typical workflow for using streams and events in a multi-GPU application is:
  - 1. Select the set of GPUs this application will use
  - 2. Create streams and events for each device
  - 3. Allocate device resources on each device (for example, device memory)
  - 4. Launch tasks on each GPU through the streams (for example, data transfers or kernel executions)
  - 5. Use the streams and events to query and wait for task completion
  - 6. Cleanup resources for all devices

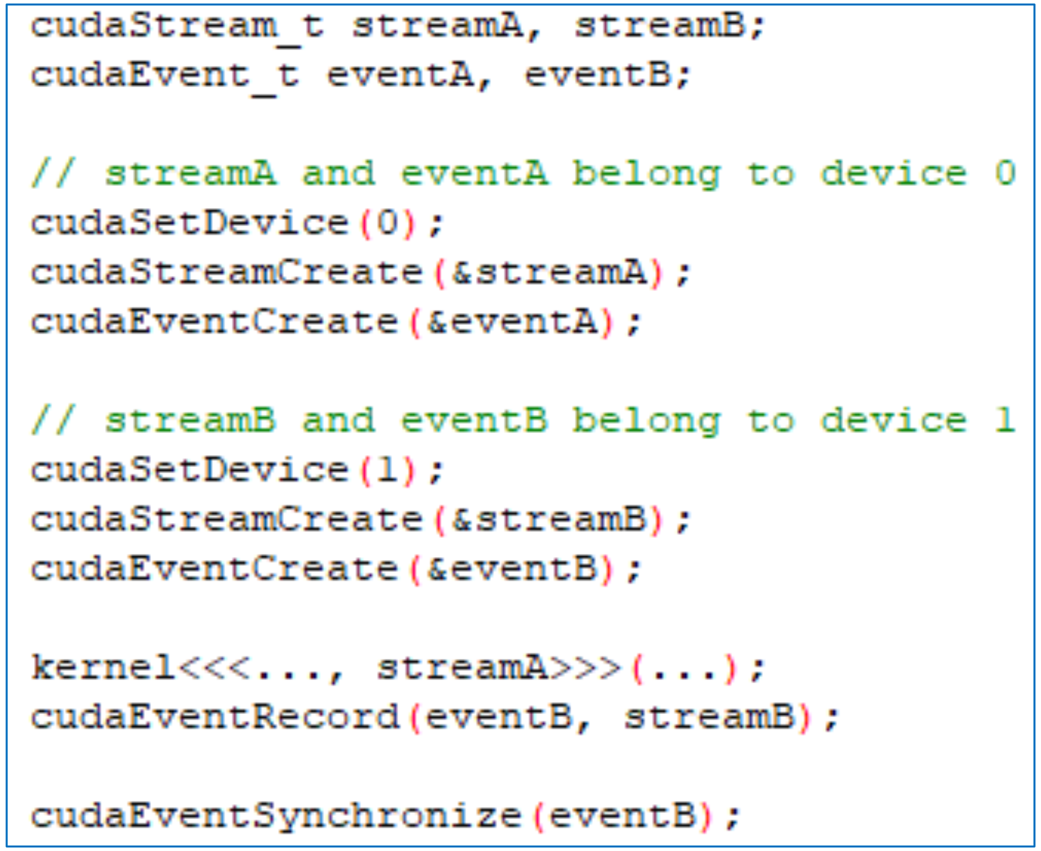
## Multi-GPUs, Streams, and Events

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

// streamA and eventA belong to device 0
cudaSetDevice(0);
cudaStreamCreate(&streamA);
cudaEventCreate(&eventA);

// streamB and eventB belong to device 1
cudaSetDevice(1);
cudaStreamCreate(&streamB);
cudaEventCreate(&eventB);

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

cudaEventSynchronize(eventB);
```

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

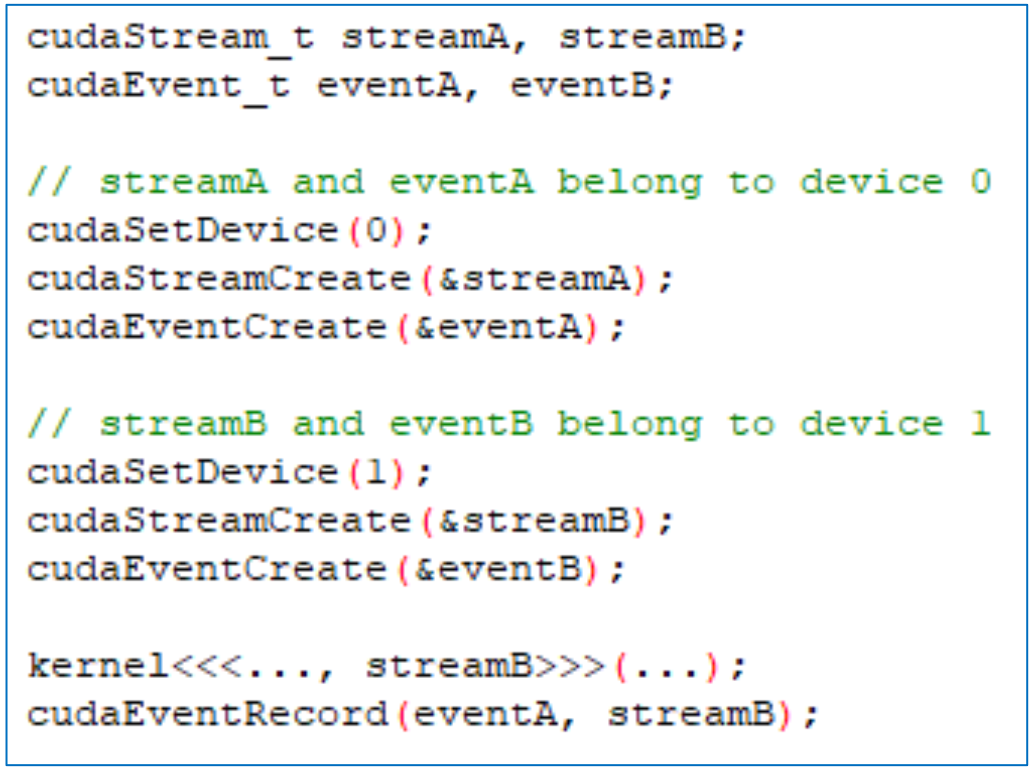
- You can launch a kernel in a stream only if the device associated with that stream is the current device
- You can record an event in a stream only if the device associated with that stream is the current device

# Multi-GPUs, Streams, and Events



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

// streamA and eventA belong to device 0
cudaSetDevice(0);
cudaStreamCreate(&streamA);
cudaEventCreate(&eventA);

// streamB and eventB belong to device 1
cudaSetDevice(1);
cudaStreamCreate(&streamB);
cudaEventCreate(&eventB);

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

cudaEventSynchronize(eventB);
```

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



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

// streamA and eventA belong to device 0
cudaSetDevice(0);
cudaStreamCreate(&streamA);
cudaEventCreate(&eventA);

// streamB and eventB belong to device 1
cudaSetDevice(1);
cudaStreamCreate(&streamB);
cudaEventCreate(&eventB);

kernel<<<..., streamB>>>(...);
cudaEventRecord(eventA, streamB);
```

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

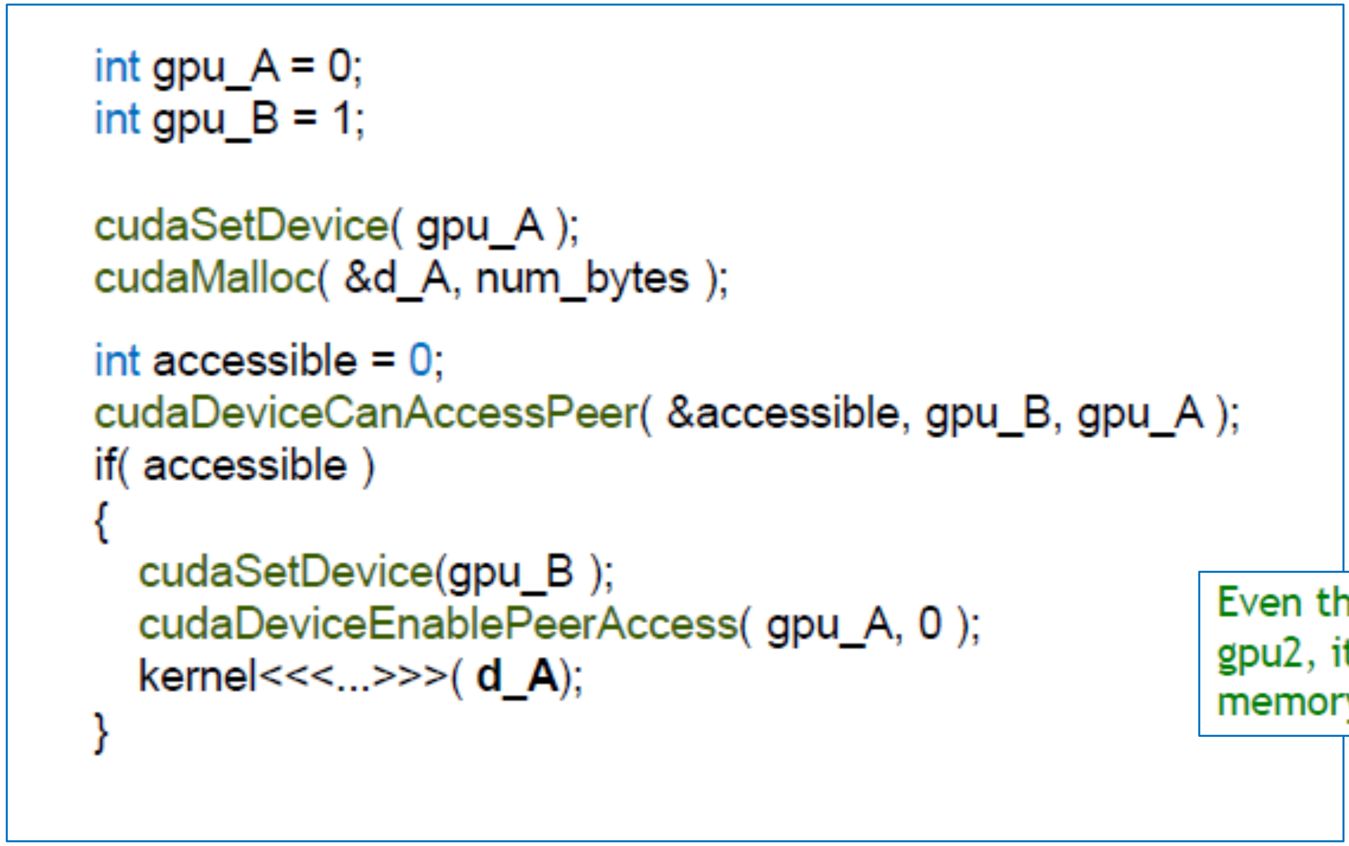
## Multi-GPUs, Streams, and Events

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

// streamA and eventA belong to device 0
cudaSetDevice(0);
cudaStreamCreate(&streamA);
cudaEventCreate(&eventA);

// streamB and eventB belong to device 1
cudaSetDevice(1);
cudaStreamCreate(&streamB);
cudaEventCreate(&eventB);

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

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

- You can query or synchronize any event or stream, even if they are not associated with the current device

OK:
- device-0 is current
- synchronizing/querying events/streams of other devices is allowed
- here, device 0 won't start executing the kernel until device 1 finishes its kernel

## Multi-GPUs, Streams, and Events



```
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

- You can issue a memory copy in any stream at any time, regardless of what device it is associated with or what the current device is

## Conclusions

- Multi-GPU systems are useful for solving problems with very large data sets that cannot fit into a single GPU device, or whose throughput and efficiency can be improved by executing multiple GPU tasks concurrently
- You can execute multi-GPU applications with multiple devices in a single node, or multiple devices in a multi-node GPU-accelerated cluster
- P2P device memory access is available to exchange data among multiple devices without staging data through CPU memory (compute capability 2.0 and higher is required for the devices)