Introduction to CUDA Parallel Programming
CUDA平行計算導論

https://ceiba.ntu.edu.tw/1092Phys8061_CUDA

Professor Ting-Wai Chiu (趙挺偉)
Email: twchiu@phys.ntu.edu.tw
Physics Department
National Taiwan University

This lecture will cover:

- CUDA with multi-GPUs
- Vector Addition with multi-GPUs
- P2P (Peer-To-Peer) Communications between GPUs

Basic Issues of Using multi-GPUs

- For large-scale problems, we need computational resources more than what can be provided by just one GPU.

Basic Issues of Using multi-GPUs

- For large-scale problems, we need computational resources more than what can be provided by just one GPU.
- The perfect scenario of using N GPUs is $\frac{\text{speed-up}}{N} \simeq 1, N \gg 1.$

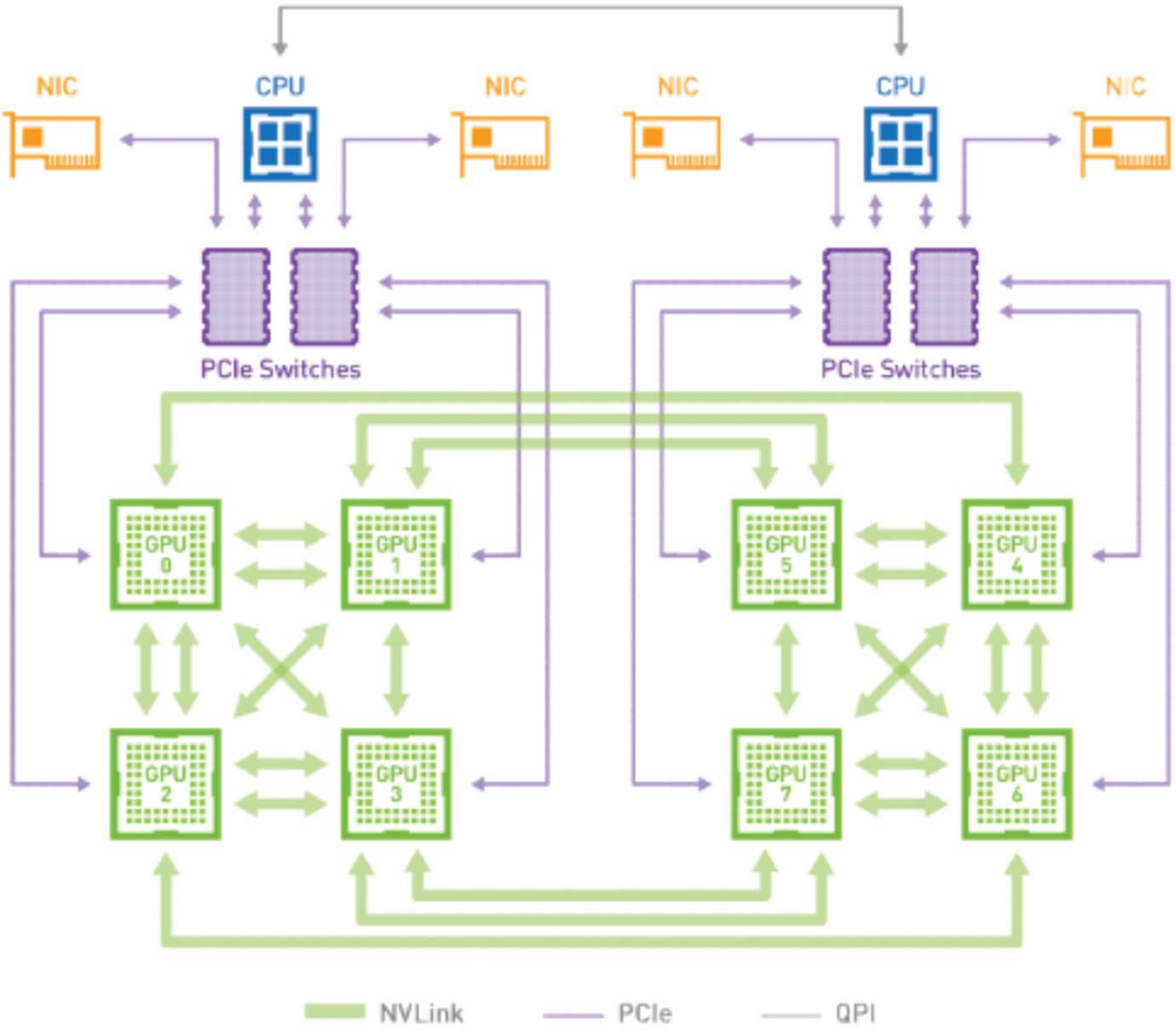
Basic Issues of Using multi-GPUs

- For large-scale problems, we need computational resources more than what can be provided by just one GPU.
- The perfect scenario of using N GPUs is $\frac{\text{speed-up}}{N} \simeq 1, N \gg 1.$
- However, the speed-up versus N becomes saturated at some N_c, and even worse, the speed-up turns out to decrease as $N > N_c$, due to the limited bandwidth for data communications.

Basic Issues of Using multi-GPUs

- For large-scale problems, we need computational resources more than what can be provided by just one GPU.
- The perfect scenario of using N GPUs is $\frac{\text{speed-up}}{N} \simeq 1, N \gg 1.$
- However, the speed-up versus N becomes saturated at some N_c, and even worse, the speed-up turns out to decrease as $N > N_c$, due to the limited bandwidth for data communications.
- For multi-GPUs, the best solution is to use Nvidia NVLink (fastest ~ 600 GB/s) to connect all GPUs on the same motherboard, but, only applicable to Nvidia A100, V100, GTX2080Ti, etc.

Nvidia NVLink in DGX-1(8 V100)



NVLink 2.0, data rate ~ 300 GB/s

2 GTX1060 on the PCIe Bus

PCIe 3.0, data rate ~ 30 GB/s

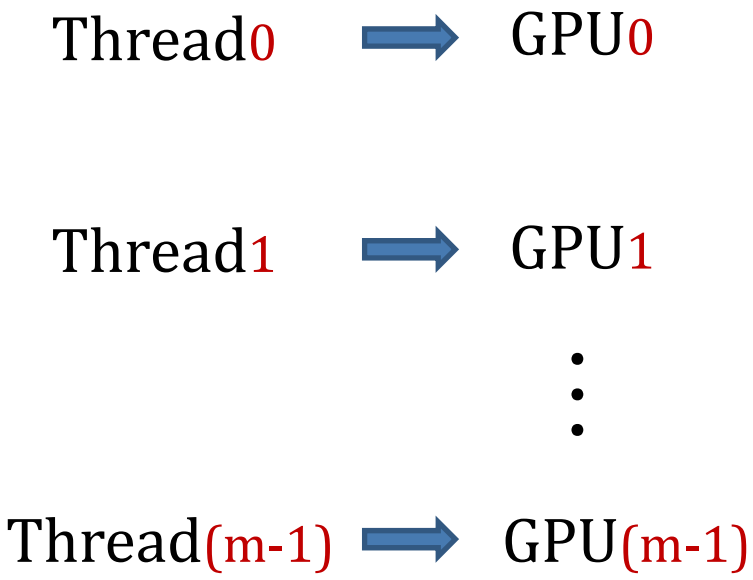
deviceQuery (GTX1060)

```
Device 1: "GeForce GTX 1060 6GB"
  CUDA Driver Version / Runtime Version          10.2 / 10.2
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 6078 MBytes (6373572608 bytes)
  (10) Multiprocessors, (128) CUDA Cores/MP:     1280 CUDA Cores
  GPU Max Clock rate:                            1759 MHz (1.76 GHz)
  Memory Clock rate:                             4004 Mhz
  Memory Bus Width:                              192-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536),
                                                 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  ...
  ...
> Peer access from GeForce GTX 1060 6GB (GPU0) -> GeForce GTX 1060 6GB (GPU1) : Yes
> Peer access from GeForce GTX 1060 6GB (GPU1) -> GeForce GTX 1060 6GB (GPU0) : Yes
```

P2P is crucial for using multi-GPUs via PCIe bus

CUDA with multi-GPUs

A viable way to design CUDA code for multi-GPUs on the same motherboard is to use OpenMP, with one OpenMP thread handling one GPU.



Vector Addition with multi-GPUs (1)

$$C = 1/A + 1/B$$

$$\begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_{N-1} \\ c_N \end{pmatrix} = \begin{pmatrix} 1/a_1 \\ 1/a_2 \\ 1/a_3 \\ \vdots \\ 1/a_{N-1} \\ 1/a_N \end{pmatrix} + \begin{pmatrix} 1/b_1 \\ 1/b_2 \\ 1/b_3 \\ \vdots \\ 1/b_{N-1} \\ 1/b_N \end{pmatrix}$$

Vector Addition with multi-GPUs (2)

$$C = 1/A + 1/B$$

$$\begin{matrix} \text{GPU0} \\ \\ \text{GPU1} \\ \\ \vdots \\ \\ \text{GPU(m-1)} \\ \\ \end{matrix} \left\{ \begin{matrix} \\ \\ \\ \\ \\ \\ \\ \\ \end{matrix} \right. \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_{N-1} \\ c_N \end{pmatrix} = \begin{pmatrix} 1/a_1 \\ 1/a_2 \\ 1/a_3 \\ \vdots \\ 1/a_{N-1} \\ 1/a_N \end{pmatrix} + \begin{pmatrix} 1/b_1 \\ 1/b_2 \\ 1/b_3 \\ \vdots \\ 1/b_{N-1} \\ 1/b_N \end{pmatrix}$$

Vector Addition with multi-GPUs (3)

$$C = 1/A + 1/B$$

Thread0 ⇒ GPU0

Thread1 ⇒ GPU1

Thread(m-1) ⇒ GPU(m-1)

$$\begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_{N-1} \\ c_N \end{pmatrix} = \begin{pmatrix} 1/a_1 \\ 1/a_2 \\ 1/a_3 \\ \vdots \\ 1/a_{N-1} \\ 1/a_N \end{pmatrix} + \begin{pmatrix} 1/b_1 \\ 1/b_2 \\ 1/b_3 \\ \vdots \\ 1/b_{N-1} \\ 1/b_N \end{pmatrix}$$

Vector Addition with multi-GPUs (4)

```
//    A template of using multi-GPUs with OpenMP
        :
#include <omp.h>       // header of OpenMP

int main(void) {
        :
omp_set_num_threads(NGPU);       // set the no. of threads = no. of GPUs
#pragma omp parallel private(cpu_thread_id)    // start of the OpenMP
{
        :
   cpu_thread_id = omp_get_thread_num();    // each thread gets its own id
   cudaSetDevice(cpu_thread_id);    // set device no.  equal to the thread id

   // each thread works on the code with its own cpu_thread_id


}  // end of the OpenMP
```

Vector Addition with multi-GPUs (5)

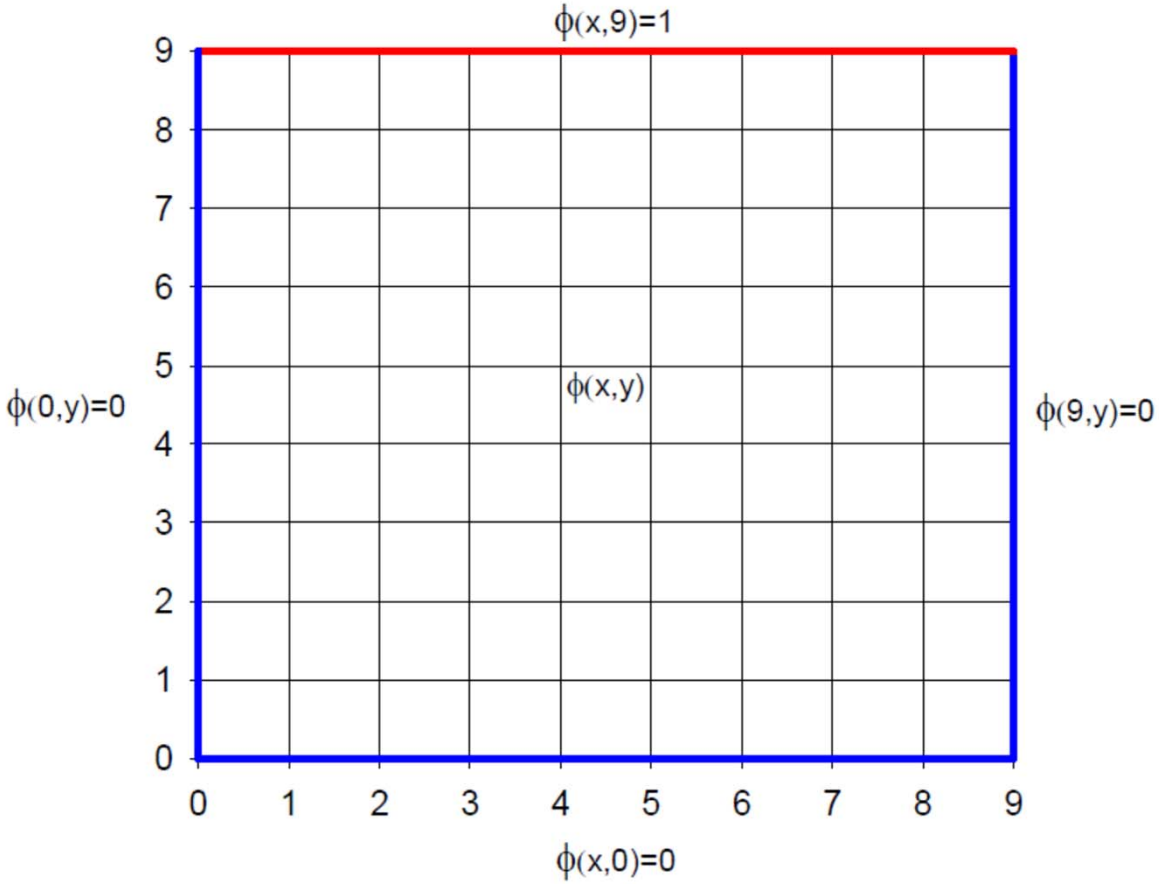
```
#include <omp.h>   // header for OpenMP

int main(void) {              //  code fragment to outline the essential features
 …                            //  of vector addition with multi-GPUs
omp_set_num_threads(N_GPU);
#pragma omp parallel private(cpu_thread_id)
{
  cpu_thread_id = omp_get_thread_num();
  cudaSetDevice(cpu_thread_id);    // set device no.  equal to cpu_thread_id
  cudaMalloc((void**)&d_A, size/N_GPU); // allocate vectors in device memory
  cudaMalloc((void**)&d_B, size/N_GPU);
  cudaMalloc((void**)&d_C, size/N_GPU);
            // Copy vectors from host memory to device memory
  cudaMemcpy(d_A,  h_A+N/N_GPU*cpu_thread_id,  size/N_GPU,
                   cudaMemcpyHostToDevice);
  cudaMemcpy(d_B,  h_B+N/N_GPU*cpu_thread_id,  size/N_GPU,
                   cudaMemcpyHostToDevice);
                                          /* to continue in the next page  */
```

Vector Addition with multi-GPUs (6)

```
VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N/N_GPU);
 cudaDeviceSynchronize();

 // copy result from device memory to host memory
 // h_C  contains the resulting vector in the host memory

 cudaMemcpy(h_C+N/N_GPU*cpu_thread_id, d_C, size/N_GPU,
                    cudaMemcpyDeviceToHost);

}     // end of the OpenMP


// Complete code at twqcd80:/home/cuda_lecture_2021/vecAdd_NGPU
```

How to Solve Laplace Equation with multi-GPUs

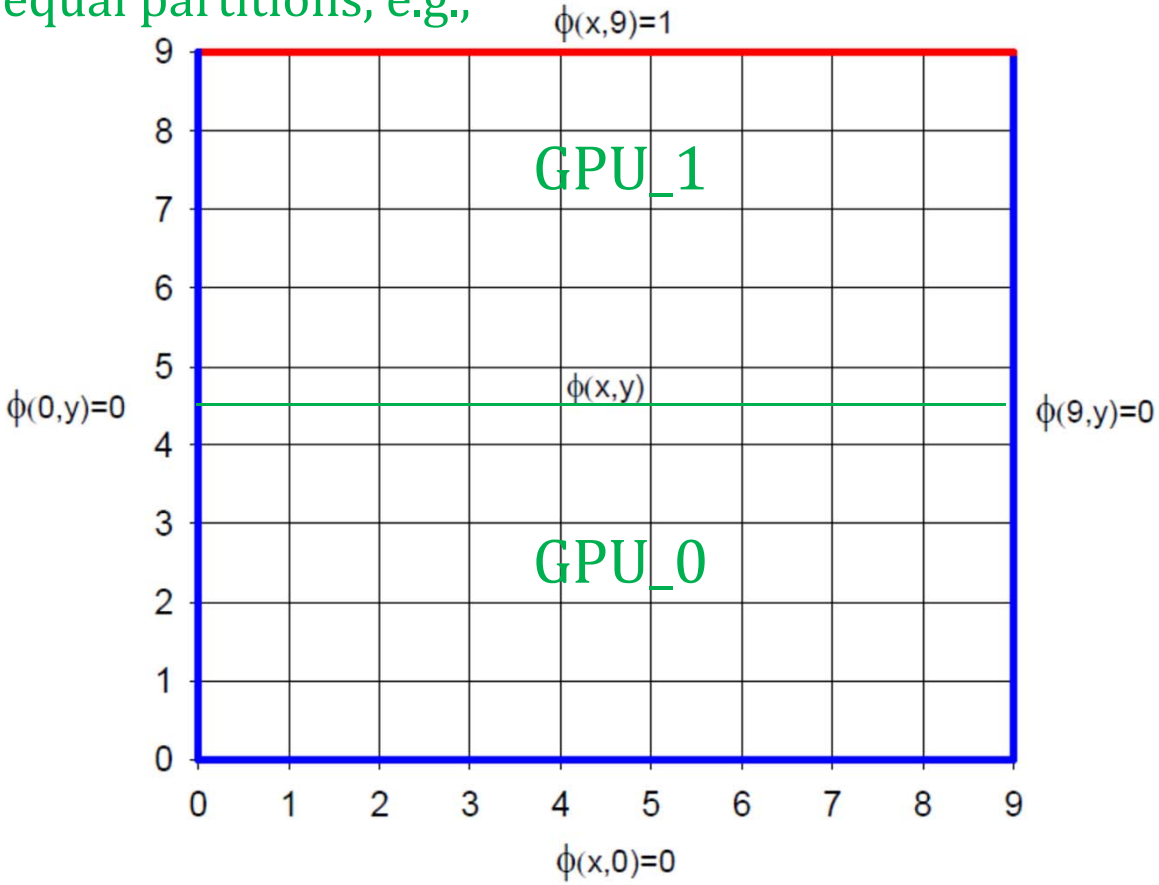
$$\phi(x,y) = \frac{1}{4}\left[\phi(x+1,y) + \phi(x-1,y) + \phi(x,y+1) + \phi(x,y-1)\right]$$



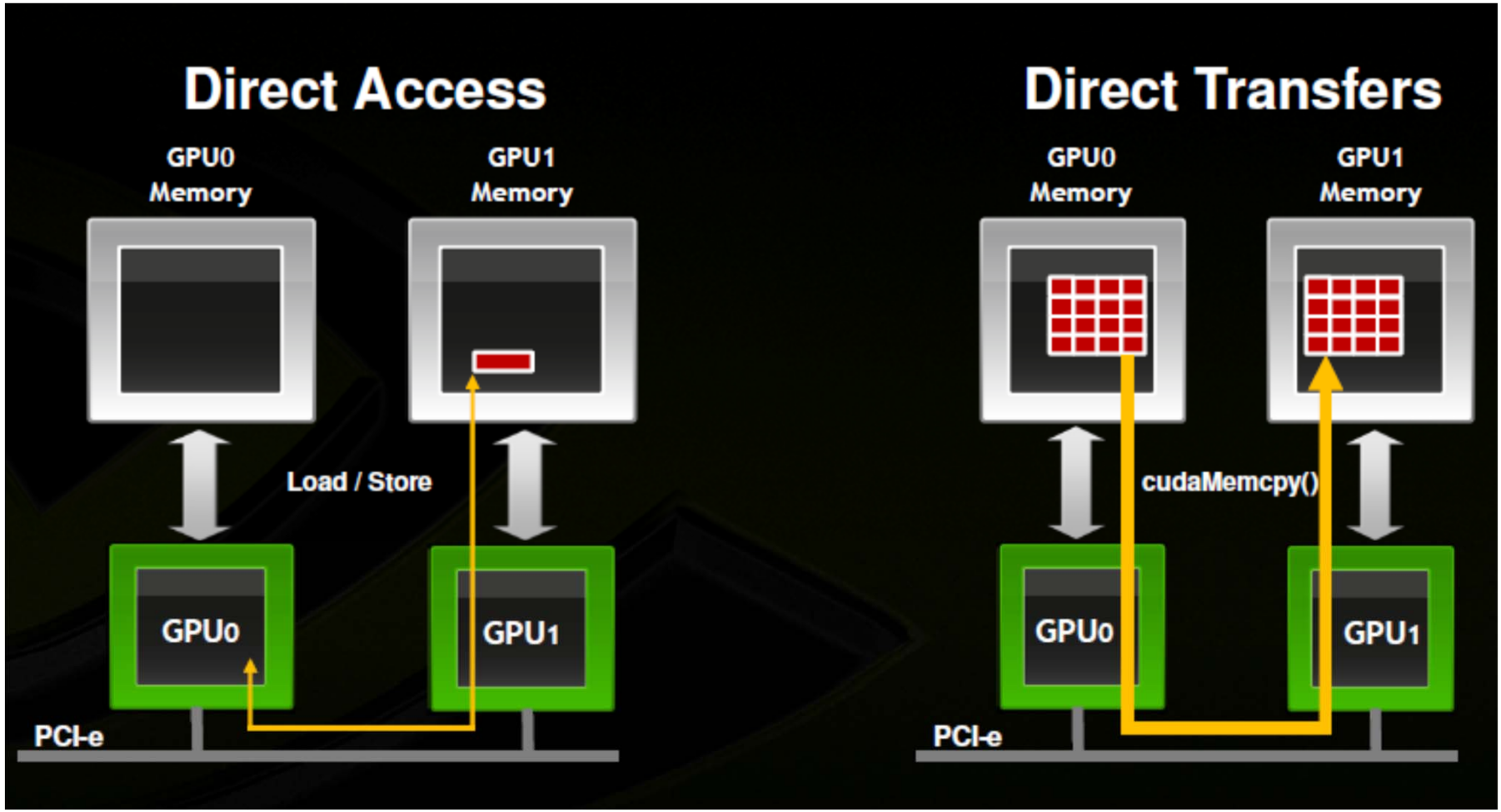
How to Solve Laplace Equation with multi-GPUs

$$\phi(x,y) = \frac{1}{4}\left[\phi(x+1,y) + \phi(x-1,y) + \phi(x,y+1) + \phi(x,y-1)\right]$$

With 2 GPUs, the natural decomposition is to cut the 2D lattice into 2 equal partitions, e.g.,



P2P Communications between GPUs



P2P eliminates system memory allocation & copy overhead

P2P Memory Copy

1. Check for peer-to-peer access between participating GPUs:

```
cudaDeviceCanAccessPeer(&can_access_peer_0_1, gpuid_0, gpuid_1);
cudaDeviceCanAccessPeer(&can_access_peer_1_0, gpuid_1, gpuid_0);
```

2. Enable peer access between participating GPUs:

```
cudaSetDevice(gpuid_0);
cudaDeviceEnablePeerAccess(gpuid_0, 0);
cudaSetDevice(gpuid_1);
cudaDeviceEnablePeerAccess(gpuid_1, 0);
```

3. Perform P2P memory copy:

```
cudaMemcpy(gpu1_buf, gpu0_buf, buf_size, cudaMemcpyDeviceToDevice)
```

4. Shutdown P2P at the end:

```
cudaSetDevice(gpuid_0);
cudaDeviceDisablePeerAccess(gpuid_0, 0);
cudaSetDevice(gpuid_1);
cudaDeviceDisablePeerAccess(gpuid_1, 0);
```

P2P Direct Access (1)

1. P2P initializations (same as before)

```
cudaDeviceCanAccessPeer(&can_access_peer_0_1, gpuid_0, gpuid_1);
cudaDeviceCanAccessPeer(&can_access_peer_1_0, gpuid_1, gpuid_0);

cudaSetDevice(gpuid_0);
cudaDeviceEnablePeerAccess(gpuid_0, 0);
cudaSetDevice(gpuid_1);
cudaDeviceEnablePeerAccess(gpuid_1, 0);
```

2. Now any GPU can read/write data in the memory of any participating GPUs

```
cudaSetDevice(gpuid_0);
SimpleKernel<<<blocks, threads>>> (gpu0_buf, gpu1_buf);

cudaSetDevice(gpuid_1);
SimpleKernel<<<blocks, threads>>> (gpu1_buf, gpu0_buf);
```

P2P Direct Access (2)

Example

```
__global__ void SimpleKernel(float *src, float *dst)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    dst[idx] = src[idx];
}

cudaSetDevice(gpuid_0);
SimpleKernel<<<blocks, threads>>> (gpu0_buf, gpu1_buf);   // write to gpu1
cudaSetDevice(gpuid_0);
SimpleKernel<<<blocks, threads>>> (gpu1_buf, gpu0_buf);   // read from gpu1

cudaSetDevice(gpuid_1);
SimpleKernel<<<blocks, threads>>> (gpu0_buf, gpu1_buf);  // read from gpu0
cudaSetDevice(gpuid_1);
SimpleKernel<<<blocks, threads>>> (gpu1_buf, gpu0_buf);  // write to gpu0
```

3. Shutdown P2P at the end (same as before)